View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by E-LIB Dokumentserver - Staats und Universitätsbibliothek Bremen

Universität Bremen
Fachbereich 3

Hard Real-Time Linux for Off-The-Shelf Multicore Architectures

Dissertation zur Erlangung des Grades eines Doktors der Ingenieurwissenschaften – Dr.-Ing. –

vorgelegt von **Dirk Radder**
im Fachbereich 3 der Universität Bremen
am 19. Mai 2015

Datum des Promotionskolloquiums: 10.11.2015

Gutachter: Prof. Dr. Jan Peleska (Universität Bremen)
Prof. Dr.-Ing. Görschwin Fey (Universität Bremen)

Zusammenfassung

In dieser Dissertation werden die Forschungsergebnisse bezüglich der Entwicklung einer Echtzeiterweiterung für das Linux Betriebssystem vorgestellt. Die Arbeit beschreibt eine vollständige Erweiterung das Kernels, welche hartes Echtzeitverhalten auf einer 64 Bit x86 Architektur ermöglicht.

Im ersten Teil dieser Arbeit werden Echtzeit-Systeme kategorisiert und Konzepte von Echtzeit-Betriebssystemen eingeführt. Im Weiteren werden zahlreiche bekannte Echtzeit-Betriebssysteme betrachtet. QNX Neutrino, `RT_PREEMPT` Linux Patch sowie HLRT Linux Patch werden detailiert analysiert und deren Kernkonzepte ausführlich diskutiert. Darüber hinaus wird eine Test-Suite erarbeitet, mit der aussagekrätige Benchmarks der analysierten Systeme erstellt werden. Die Systeme werden anhand dieser Benchmarks evaluiert und mit der in dieser Arbeit entwickelten Echtzeit-Erweiterung verglichen.

Anhand der vorausgegangenen Analysen der genannten Systeme wird ein Katalog von Anforderungen definiert, den die entwickelte Echtzeit-Erweiterung umsetzen wird. Basierend auf diesem Anforderungs-Katalog und den identifizierten Kernkonzepten der analysierten Systeme wird der Entwurf der Echtzeit-Erweiterung erarbeitet und deren konkrete Implementierung dargestellt. Abschließend werden die Benchmarks aller analysierten Systeme, einschließlich der erarbeiteten Echtzeit-Erweiterung, miteinander verglichen und bewertet.

Abstract

This document describes the research results that were obtained from the development of a real-time extension for the Linux operating system. The paper describes a full extension of the kernel, which enables hard real-time performance on a 64-bit x86 architecture.

In the first part of this study, real-time systems are categorized and concepts of real-time operating systems are introduced to the reader. In addition, numerous well-known real-time operating systems are considered. QNX Neutrino, RT_PREEMPT Linux Patch and HLRT Linux Patch are analyzed in detail. The core concepts of these systems are shown and discussed. Furthermore, a test suite is developed, which is used to obtain expressive benchmarks from the systems that were analyzed before. The systems are evaluated on the basis of these benchmarks and compared to the real-time extension which is developed in this work.

A requirements catalogue is defined based on the analysis of the stated operating systems. The design of a real-time extension is developed based on the specification catalogue and the identified core concepts. Furthermore, the concrete implementation of the developed real-time extension is presented in detail. Finally, the benchmarks of all analyzed systems, including the developed real-time extension, are compared to each other and evaluated.

Contents

List of Figures xiii

List of Tables xv

1 Introduction 1
1.1 Objectives 2
1.2 Main Contributions 2
1.3 Related Work 3
1.4 Structure of this Document 6

I Background and Components 9

2 Concepts of Real-Time Systems 11
2.1 Definition and Classification of Real-Time Systems 11
2.1.1 Proprietary versus Open 12
2.1.2 Centralized versus Distributed 13
2.1.3 Fail-Safe versus Fail-Operational 13
2.1.4 Hard versus Soft Real-Time 14
2.1.5 Event-Triggered versus Time-Triggered 15
2.2 Requirements of Real-Time Systems 16
2.2.1 Functional Requirements 17
2.2.2 Temporal Requirements 17
2.2.3 Dependability Requirements 18
2.2.4 Architectural Requirements 18
2.3 Hardware for Real-Time Systems 19
2.4 x86 Instruction Set Architecture 20
2.4.1 General Remarks and Design 20
2.4.2 Hyper-Threading Technology 21
2.4.3 APIC Architecture 21
2.4.4 Time Stamp Counter 22
2.4.5 Problems with Hard Real-Time on the x86 Architecture 22

3 Real-Time Operating Systems 23
3.1 Task Management 23

3.1.1 Task States 24
3.1.2 Task Hierarchy 24
3.1.3 Task Types 25
3.2 Scheduling 25
3.2.1 Clock Driven Scheduling Strategies 26
3.2.2 Event Driven Scheduling Strategies 26
3.2.2.1 Earliest-Deadline-First Scheduling 26
3.2.2.2 Rate-Monotonic Scheduling 26
3.2.3 Hybrid Scheduling Strategies 27
3.2.3.1 First-In-First-Out Scheduling 27
3.2.3.2 Round-Robin Scheduling 28
3.3 Partitioning 28
3.4 Main System Services 29
3.5 POSIX Standard 29
3.5.1 Real-Time System Profiles 30
3.6 Real-Time and Linux 31
3.6.1 Real-Time Kernel 31
3.6.2 Kernel Preemption 32
3.6.3 Resource Reservation 32

II Real-Time Operating Systems Analysis 33

4 Overview of Available Real-Time Operating Systems 35
4.1 Atomthreads 39
4.1.1 Structure of the Kernel 39
4.1.2 CPU Architecture Ports 41
4.2 eCos 41
4.2.1 Design 42
4.3 VxWorks 42
4.3.1 Protection Domains Architecture 43
4.4 μC/OS-III 43
4.4.1 Design 44
4.5 MontaVistaLinux 44
4.6 ThreadX 44
4.6.1 Pico Kernel 45
4.7 RTLinux 45
4.7.1 Structure of the Kernel 45
4.8 QNX, HLRT and RT-Preempt 46

5 Evaluating (POSIX) Real-Time Operating Systems 49
5.1 Unconsidered Aspects 49
5.2 Identify Technical Values 50
5.2.1 Benchmark Methodology 51

5.3 Case Scenarios for Real-Time Systems . . . 53
5.3.1 Development Board . . . 54
5.4 Benchmark Test Framework . . . 55
5.4.1 Measurement Details . . . 55
5.4.2 Operating System Overhead . . . 57
5.4.3 Test Design . . . 58
5.4.3.1 Task Period Accuracy . . . 59
5.4.3.2 Task Change Times . . . 60
5.4.3.3 Task Creation Time . . . 62
5.4.3.4 Interrupt Times . . . 63

6 Case Study 1: RT-Preempt Patch 67
6.1 Background and Overview . . . 67
6.2 Preemptable In-Kernel Locking Primitives . . . 68
6.2.1 Priority Inheritance for In-Kernel Locking Primitives . . . 70
6.3 Interrupt Handlers as Kernel Threads . . . 72
6.4 Real-Time Application Programming . . . 75
6.5 Benchmarking . . . 75
6.5.1 Task Period Tests . . . 76
6.5.2 Task Switch Tests . . . 79
6.5.3 Task Creation Test . . . 84
6.5.4 Interrupt Tests . . . 86
6.6 Summary . . . 89

7 Case Study 2: HaRTLinC 91
7.1 Background and Overview . . . 91
7.2 CPU Reservation . . . 92
7.2.1 The `SCHED_HLRT` Scheduling Policy . . . 92
7.2.2 Interrupt Routing . . . 96
7.2.3 Necessary Adjustments . . . 97
7.3 Time-Triggered Architecture . . . 98
7.4 Real-Time Application Programming . . . 99
7.5 Benchmarking . . . 100
7.5.1 Task Period Tests . . . 100
7.5.2 Interrupt Tests . . . 103
7.6 Summary . . . 105

8 Case Study 3: QNX Neutrino 107
8.1 Background and Overview . . . 107
8.2 Microkernel Architecture . . . 107
8.2.1 Process Management . . . 108
8.2.2 Interrupt Handling . . . 110
8.2.3 Message Passing . . . 110
8.3 Adaptive Partitioning Scheduler . . . 111

8.4 Benchmarking . . . 114
8.4.1 Task Period Tests . . . 115
8.4.2 Task Switch Tests . . . 117
8.4.3 Task Creation Test . . . 121
8.4.4 Interrupt Tests . . . 121
8.5 Summary . . . 124

III A Hard Real-Time Linux Operating System **125**

9 Requirements Discussion **127**
9.1 Analysis . . . 130
9.2 Coverage . . . 131

10 Hard Real-Time Linux System Design **133**
10.1 Overview . . . 133
10.2 CPU Reservation . . . 134
10.2.1 Interrupt Routing . . . 135
10.2.2 CPU Profiles . . . 135
10.2.3 Housekeeping . . . 135
10.3 Partitioning . . . 136
10.3.1 Scheduling Groups . . . 136
10.3.2 Group Cycles . . . 138
10.3.3 Threaded Interrupt Handling . . . 139
10.3.4 System-Call Redirection . . . 139
10.4 Task Management . . . 140
10.4.1 Events . . . 140
10.4.1.1 Synchronisation . . . 140
10.4.1.2 Deadlines . . . 140
10.4.2 Kernel Preemption . . . 141
10.4.2.1 System-Call Handler Threads . . . 141
10.4.2.2 Preemptible Critical Sections . . . 142
10.4.3 Scheduling in Static Groups . . . 143
10.4.4 CPU Budget Based Scheduling . . . 145

11 Description of the HRT Linux Implementation **147**
11.1 Global Objects and Data Types . . . 147
11.2 Management of Memory Objects . . . 149
11.2.1 Id and Key Pool . . . 150
11.2.2 Entity System-Call Multiplexer . . . 151
11.3 Subsystems . . . 151
11.3.1 Time Base and Clock Sources . . . 152
11.3.2 Timer and Interrupts . . . 152
11.3.3 Events . . . 153

11.4 CPU Reservation 154
11.4.1 Per CPU Idle Task 155
11.4.2 CPU States 156
11.4.3 CPU Takeover 157
11.4.4 Necessary Adjustments 159
11.5 Scheduling Class (SCHED_HRTL) 163
11.5.1 Linux Scheduler Integration 163
11.5.2 Adding Tasks to SCHED_HRTL 164
11.5.3 Necessary Adjustments 165
11.6 Static Scheduling Plan 166
11.6.1 Cycles 166
11.6.2 Partition Management 168
11.6.3 Interrupt Handlers 168
11.6.4 Interface for Scheduler Modules 169
11.6.5 Deadline Events 171
11.7 Balancing Dynamic Partitions 172
11.7.1 CPU Usage Measurement 172
11.7.2 Group Distribution 173
11.7.3 SCHED_HRTL Integration 174
11.8 System-Call Handler Threads 175
11.8.1 System-Call Redirection 176
11.8.2 Work Package Scheduling 177
11.8.3 Spin-Lock Replacement 180
11.8.4 Necessary Adjustments 181
11.9 Real-Time Application Programming 182
11.9.1 User-Space Interface 182
11.9.1.1 Library 183
11.9.2 Benchmarking 193
11.9.2.1 Task Period Tests 193
11.9.2.2 Task Switch Tests 196
11.9.2.3 Task Creation Test 199
11.9.2.4 Interrupt Tests 201

IV Evaluation **205**

12 Benchmark Results Comparison **207**

13 Conclusion and Outlook **211**

Bibliography **213**

List of Figures

1.1 Overview of the thesis 7

2.1 Real-time system 12
2.2 Linking interface 13
2.3 Timing requirements of jobs 14
2.4 APIC architecture 21

3.1 Internal and external view on processes 23
3.2 Preemptive tasks 24
3.3 Real-time kernel design 31

4.1 Typical memory footprints of real-time operating systems 38
4.2 VxWorks protection domains architecture 43
4.3 RTLinux kernel design 46

5.1 Task switch latency 51
5.2 Interrupt times 52
5.3 Interrupt to task latency 53
5.4 Benchmark test behaviour 58
5.5 Periodic task benchmark test 59
5.6 Task switch/preemption benchmark test 61
5.7 Task creation benchmark test 62
5.8 Interrupt benchmark test 63
5.9 Interrupt latency benchmark test 64
5.10 Interrupt dispatch latency benchmark test 64
5.11 Interrupt to task latency benchmark test 65

6.1 Priority inversion 71
6.2 RT-Preempt scheduling hierarchy 73

8.1 QNX Neutrino adaptive partitioning 112

9.1 Real-time operating system requirements 130

10.1 Hard real-time Linux scheduling groups (Example 1) 134
10.2 FIFO scheduling in Example 1 143
10.3 RM scheduling in Example 1 144

10.4 Available CPU time (Example 2) 146

List of Tables

2.1 Characteristics of real-time systems 12
2.2 Hard real-time versus soft real-time 15
2.3 Requirements of real-time systems 16

4.1 Real-time operating systems overview 37
4.2 Real-time operating systems hardware characteristics 40
4.3 Real-time operating systems technical characteristics 40

5.1 Development board specification 54

6.1 Benchmark test results [μs]: RT-Preempt period task (500μs) 78
6.2 Benchmark test results [μs]: Linux 3.4.104 period task (500μs) 78
6.3 Benchmark test results [μs]: RT-Preempt period task (10ms) 78
6.4 Benchmark test results [μs]: Linux 3.4.104 period task (10ms) 78
6.5 Benchmark test results [μs]: RT-Preempt period task (100ms) 79
6.6 Benchmark test results [μs]: Linux 3.4.104 period task (100ms) 79
6.7 Benchmark test results [μs]: RT-Preempt period task (1sec) 79
6.8 Benchmark test results [μs]: Linux 3.4.104 period task (1sec) 79
6.9 Benchmark test results [μs]: RT-Preempt preempt task (signal) 81
6.10 Benchmark test results [μs]: Linux 3.4.104 preempt task (signal) . . . 82
6.11 Benchmark test results [μs]: RT-Preempt switch task (2 tasks) 83
6.12 Benchmark test results [μs]: Linux 3.4.104 switch task (2 tasks) 84
6.13 Benchmark test results [μs]: RT-Preempt switch task (16 tasks) 84
6.14 Benchmark test results [μs]: Linux 3.4.104 switch task (16 tasks) . . . 84
6.15 Benchmark test results [μs]: RT-Preempt switch task (128 tasks) . . . 84
6.16 Benchmark test results [μs]: Linux 3.4.104 switch task (128 tasks) . . 84
6.17 Benchmark test results [μs]: RT-Preempt switch task (512 tasks) . . . 84
6.18 Benchmark test results [μs]: Linux 3.4.104 switch task (512 tasks) . . 84
6.19 Benchmark test results [μs]: RT-Preempt task creation 85
6.20 Benchmark test results [μs]: Linux 3.4.104 task creation 86
6.21 Benchmark test results [μs]: RT-Preempt interrupt latency (ISR) . . . 88
6.22 Benchmark test results [μs]: Linux 3.4.104 interrupt latency (ISR) . . 88
6.23 Benchmark test results [μs]: RT-Preempt interrupt latency (dispatch) 89
6.24 Benchmark test results [μs]: Linux 3.4.104 interrupt latency (dispatch) 89
6.25 Benchmark test results [μs]: RT-Preempt interrupt latency (SLIH) . . 89

6.26 Benchmark test results [μs]: Linux 3.4.104 interrupt latency (SLIH) . 89

7.1 Benchmark test results [μs]: HLRT period task (500μs) 102
7.2 Benchmark test results [μs]: HLRT period task (10ms) 102
7.3 Benchmark test results [μs]: Linux 2.6.27 period task (10ms) 102
7.4 Benchmark test results [μs]: HLRT period task (100ms) 102
7.5 Benchmark test results [μs]: Linux 2.6.27 period task (100ms) 103
7.6 Benchmark test results [μs]: HLRT period task (1sec) 103
7.7 Benchmark test results [μs]: Linux 2.6.27 period task (1sec) 103
7.8 Benchmark test results [μs]: HLRT interrupt latency (ISR) 104
7.9 Benchmark test results [μs]: Linux 2.6.27 interrupt latency (ISR) . . . 104
7.10 Benchmark test results [μs]: HLRT interrupt latency (dispatch) 105
7.11 Benchmark test results [μs]: Linux 2.6.27 interrupt latency (dispatch) 105

8.1 Benchmark test results [μs]: QNX period task (500μs) 117
8.2 Benchmark test results [μs]: QNX period task (10ms) 117
8.3 Benchmark test results [μs]: QNX period task (100ms) 117
8.4 Benchmark test results [μs]: QNX period task (1sec) 117
8.5 Benchmark test results [μs]: QNX preempt task (event) 120
8.6 Benchmark test results [μs]: QNX preempt task (signal) 120
8.7 Benchmark test results [μs]: QNX switch task (2 tasks) 121
8.8 Benchmark test results [μs]: QNX switch task (16 tasks) 121
8.9 Benchmark test results [μs]: QNX switch task (128 tasks) 121
8.10 Benchmark test results [μs]: QNX switch task (512 tasks) 121
8.11 Benchmark test results [μs]: QNX task creation 122
8.12 Benchmark test results [μs]: QNX interrupt latency (ISR) 122
8.13 Benchmark test results [μs]: QNX interrupt latency (dispatch) 123
8.14 Benchmark test results [μs]: QNX interrupt latency (SLIH) 123

9.1 Listing of determined requirements 129
9.2 Requirements coverage 131

10.1 RM scheduling in Example 1 145
10.2 RM scheduling in Example 1 with unworkable task set 145

11.1 CPU states 157
11.2 Listing of HRTL systemcalls 183
11.3 Benchmark test results [μs]: HRTL period task (500μs) 194
11.4 Benchmark test results [μs]: Linux 3.5.7 period task (500μs) 194
11.5 Benchmark test results [μs]: HRTL period task (10ms) 195
11.6 Benchmark test results [μs]: Linux 3.5.7 period task (10ms) 195
11.7 Benchmark test results [μs]: HRTL period task (100ms) 195
11.8 Benchmark test results [μs]: Linux 3.5.7 period task (100ms) 195
11.9 Benchmark test results [μs]: HRTL period task (1sec) 195
11.10 Benchmark test results [μs]: Linux 3.5.7 period task (1sec) 196

11.11Benchmark test results [μs]: HRTL preempt task (signal) 197
11.12Benchmark test results [μs]: Linux 3.5.7 preempt task (signal) 197
11.13Benchmark test results [μs]: HRTL preempt task (event) 198
11.14Benchmark test results [μs]: HRTL switch task (2 tasks) 199
11.15Benchmark test results [μs]: Linux 3.5.7 switch task (2 tasks) 199
11.16Benchmark test results [μs]: HRTL switch task (16 tasks) 199
11.17Benchmark test results [μs]: Linux 3.5.7 switch task (16 tasks) 199
11.18Benchmark test results [μs]: HRTL switch task (128 tasks) 199
11.19Benchmark test results [μs]: Linux 3.5.7 switch task (128 tasks) 199
11.20Benchmark test results [μs]: HRTL switch task (512 tasks) 199
11.21Benchmark test results [μs]: Linux 3.5.7 switch task (512 tasks) 200
11.22Benchmark test results [μs]: HRTL task creation 201
11.23Benchmark test results [μs]: Linux 3.5.7 task creation 201
11.24Benchmark test results [μs]: HRTL interrupt latency (ISR) 203
11.25Benchmark test results [μs]: Linux 3.5.7 interrupt latency (ISR) 203
11.26Benchmark test results [μs]: HRTL interrupt latency (dispatch) 204
11.27Benchmark test results [μs]: Linux 3.5.7 interrupt latency (dispatch) . 204
11.28Benchmark test results [μs]: HRTL interrupt latency (SLIH) 204
11.29Benchmark test results [μs]: Linux 3.5.7 interrupt latency (SLIH) . . . 204

12.1 Period benchmark results overview [μs] 207
12.2 Process creation benchmark results overview [μs] 208
12.3 Preemption benchmark results overview [μs] 209
12.4 Task switch benchmark results overview [μs] 209
12.5 HRTL task switch results without caching 210
12.6 Interrupt benchmark results overview [μs] 210

1

Introduction

Embedded systems are present in a wide variety of electronic products. In the day-to-day life we are habituated to use them easily. Many of these systems must continually react to changes in the system's environment and must compute certain results at a specific time without delay. For some systems, there is no value to a certain result if it is not available at a defined time. In some cases, the effects of a late computation may be catastrophic. A flight control system presents a good example.

Most embedded real-time systems are designed to have very low memory footprint and high modularity. They are developed to solve a clearly structured task in a unique situation. In order to test the conformance of embedded real-time systems to their specified requirements a test system must be able to stimulate inputs within the required time bounds. In contrast to embedded real-time applications like the flight controller that served as a safety-critical example, some applications require a more flexible computer system with strict real-time constraints. For instance, high latency will ruin a good performance for an audio recording system.

In recent years, the Linux operating system has raised some interest in both industrial and private spheres. It was initially designed around fairness and good average performance and provides only *soft* real-time capabilities. Various specialised hardware and software solutions are available for systems with stricter real-time constraints. However, the costs for such real-time computer systems can be very high. Using off-the-shelf computer hardware in combination with Linux is a much more cost-effective approach.

The Linux kernel design conflicts with real-time requirements in many aspects and lacks some features associated with traditional real-time operating systems. It is a monolithic kernel [BC05, Chap. 1], which means that device drivers, scheduler and operating system services reside side by side in what is referred to as kernel space. A program that requests an operating system service can generally not be interrupted before it has finished the interaction. Furthermore, interrupt handlers can cause latency on high priority task execution. Thus, a deterministic priority scheduling of tasks is not provided. As a consequence, the kernel needs to be modified in order to support *hard* real-time applications. Several extensions pursuing different approaches already exist. They all introduce different real-time features to Linux, but none of them can be said to successfully transform Linux into a full modern real-time

operating system.

This first chapter introduces the objectives of this dissertation. Furthermore, the main contributions are explained in detail. The chapter concludes with an overview of the structure of this document.

1.1. Objectives

Two distinctive approaches to add real-time capabilities to Linux can be identified. One uses a dual kernel design where Linux runs at a task on top of a real-time kernel [BM14, Sect. 18.4], while the other improves the Linux kernel itself so that it fulfills real-time requirements [YMBYG08, Chap. 12]. In a dual kernel design, tasks running in the real-time kernel can make only limited use of existing system services in Linux. Drivers and system services must be created specifically for the real-time kernel. On the other hand, known extensions that introduce real-time capabilities to Linux only deal with some aspects of *real-time*. A fully featured real-time operating system is not provided by those extensions.[1]

The aim of this work is to develop an extension for the Linux operating system that adds real-time capabilities and extends the kernel with new features. The design of this extension is based on investigations of already existing approaches that extend Linux with real-time features as well as investigations of traditional real-time operating systems. As aforementioned in the introduction to this chapter, the resultant real-time operating system will be implemented for off-the-shelf computer hardware and represents an alternative to commercial software solutions.

1.2. Main Contributions

The following main contributions are presented in this dissertation:

→ detailed analysis of real-time operating systems with main focus on Linux real-time extensions

→ design and implementation of a completely new approach to enhance Linux with real-time capabilities

→ development of a benchmark test suite for POSIX like real-time operating systems

As a first step, it was necessary to identify the features that are required for real-time operating systems. These requirements derive from the investigations of traditional real-time operating systems like QNX Neutrino (see Chapter 8) as well as from real-time Linux extensions like the `RT_PREEMPT`-Patch (see Chapter 6) and the HLRT-Patch (see Chapter 7).

[1] Chapter 6 and Chapter 7 discuss two Linux real-time extensions. An overview of supported real-time features is given in Chapter 9.

Based on the identified requirements, a system design was developed that extends the Linux kernel with real-time capabilities. The extension turns Linux into a fully preemptible operating system and introduces partitioning concepts in different variants to the kernel. Balancing system-calls between partitions allow critical sections to be preempted inside the kernel and provide a highly flexible and low latency scheduling of tasks and partitions.

In order to evaluate the implementation of the new extension, a method that allows a reliable comparison to other operating systems was developed. The benchmark test suite was applied to the new extension and the other investigated real-time solutions.

1.3. Related Work

The design and implementation of real-time systems poses special requirements to both, the hardware the operating system is executed on and the software implementation of the operating system itself. Often, dedicated hardware is designed to guarantee real-time properties of the system directly on the hardware level.[2] An example of such system from the aerospace domain is the *Integrated Modular Avionics* (IMA) architecture [Efk14]. The IMA concept includes an assembly of common hardware modules capable of supporting numerous applications of differing criticality levels.

Our work, however, is concerned with the design and implementation of real-time capable operating systems solely on the level of software. We base our implementation on standard x86 architectures, and extend a Linux-based operating system kernel so that it delivers real-time capabilities. The fundamentals of real-time system design have been discussed in [Kop97, Liu00, BW09]. In principle, the system architecture can be classified into either time- or event-triggered, a classification that depends on the types of internal events which cause some action of the system. In event-triggered systems, all activities are started whenever a relevant event occurs, which is classically realized using interrupts [Kop97, Sect. 1.5.5]. By way of contrast, in a time-triggered system, all actions are initiated by the progression of time [Kop97, Sect. 1.5.5]. Our own real-time extension of the Linux kernel implements a time-triggered architecture [Kop97, Chap. 14] [KB03].

The evaluation of real-time operating systems has been discussed extensively in the literature. Measuring the execution time and performance in real-time operating systems is described in [Ste02]. The paper presents a variety of techniques, which mostly aim at measuring the timing behavior of applications within a real-time environment. However, in the context of our work, measuring the operating system overhead is the interesting part. Two different kinds of operating system overhead are introduced in [Ste02, Sect. 4.3]: context switch time and interrupt latency. Context switching from one task to another is measured by two (or more) alternating tasks on different priority levels. Each context switch triggers an externel logic analyzer.

[2] A comparison of commercial off-the-shelf and special-to-purpose platforms is given in [Ott06, Sect. 1.5].

Measuring the overhead of an interrupt handler is done in a similar way, where an interrupt handler toggles an input bit to the logic analyzer.

In this thesis we develop our own test suite for benchmarking real-time operating systems based on the Rhealstone method [KP89]. For our test suite we refine the two methods from [Ste02] (task switching and interrupt latency) by adding benchmarks for the *task preemption latency* and the *interrupt dispatch latency*. The latter benchmark has been discussed in [RB10, Sect. 3.2]. The authors identify several components and show that the dispatch latency is composed from the runtimes of these components. In order to measure all latency sources on the interrupt dispatch path, an instrumented kernel is needed. However, our benchmark framework is designed to optain benchmark results from different operating systems without affecting the operating system kernels, and hence the interrupt dispatch latency measurement is not divided into sub-components.

[KP89, Ste02] describe a set of benchmark programs, on which our test suite is based. Most notably, we have integrated benchmarks for task switch latency, task preemption latency, interrupt latency, interrupt dispatch latency, and interrupt to task latency into our benchmark framework. Additionally, we have added benchmarks for process creation and accuracy of task periods (compare Section 5.2.1). We defer the discussion of the details of our benchmark test suite to Section 5.4.3. Technical aspects of how to retrieve precise and reliable time values are discussed in Section 5.4.1. The time accounting is based on the *time stamp counter* (TSC) register. [Pao10] describes the technique that is used in this thesis to measure the clock cycles from the TSC in a Linux environment running on a x86 architecture.

Our extension of the Linux kernel with real-time capabilities presented in this thesis is based on existing work that targets partitioning, scheduling, preemption and techniques for system-call distribution. In the following, we discuss the relation of the state-of-the-art with respect to these techniques to our work. These issues have, at least to some extent, been addressed by existing implementations of real-time operation systems. Examples of such systems are the RT-Preempt Linux extension [MG] and QNX [Bla]. We defer the discussion of the details of these systems and how their approach deviates from ours to Chapter 6 and Chapter 8.

Partitioning

The term partitioning refers to techniques which assign shared resources (such as computation time) of the system to tasks in a predictable way. Partitioning is thus a core aspect for multi-task real-time operating systems. One important technique, which is implemented in real-time operating systems such as QNX [JCLC06], is referred to as *adaptive partitioning scheduling*. Using this technique, partitioning may be adapted at runtime as long as the adaptation does not prevent processes from meeting their deadlines. Each partition has an fixed amount of available computation time allotted. A task that overruns the partition's time budget can only affect other tasks in the same partition [DB11, Chap. 5]. The approach can thus be seen as

opposed to traditional partitioning that uses static assignments of time slots (as implemented for operating systems such as PikeOS [SYS] or RT-Mach [TNR90]). The paper [MFL+09] describes a implementation of a similar idea for Minix 3 [HBG+06]. In their approach, a partition can request additional CPU time from a so-called *virtual resource server* in order to meet its deadlines [MFL+09, Sect. 3.1]. In Section 10.3.1, we describe partitioning by *dynamic scheduling groups*, which is based on similar ideas.

According to [RTC92], an implementation of an partitioning operating system assigns static time slots to its software partitions. Each partition receives a fixed amount of CPU time. The operating system's scheduler ensures that no partition can spend CPU time allotted to another partition. [KW07] describes the concepts of how time partitioning can be realized through a fixed-cyclic time-slice scheduler, which allocates periods of time to each partition. The paper investigates the partitioning strategies of PikeOS [SYS] where scheduling decisions are made according to a pre-configured static schedule plan. We will describe *static scheduling groups* in Section 10.3.1, which are based on these concepts.

A comparison of different operating systems that provide partitioning is given in [LSOH07].

Task Scheduling

Based on the partitioning concepts as introduced above, various priority and deadline orientated task scheduling strategies are implemented for our real-time operating system. The task scheduling for partitions is realised by *scheduling modules*. A scheduling module can be implemented as a Linux loadable module and can be registered and deregistered at system runtime. Each (static) partition has to be connected with one task scheduling module. [GAGB01] introduces the concept of flexible scheduling modules for Linux like operating system kernels. The implementation of a scheduling algorithm should possibly be independent and handle only the scheduling behavior. The paper presents the details of a *soft and hard real-time kernel* (S.Ha.R.K.) which was purposely designed to support the implementation and testing of new scheduling algorithms. In Section 11.6.4, we describe scheduling modules for our operating system. The interface for these modules is based on similar ideas.

Current literature is rich in techniques and algorithms for real-time scheduling [dMC00]. Deadline orientated task scheduling like rate-monotonic scheduling (see Section 3.2.2.2) is discussed in detail in [LL73] and [AB98]. We will describe the details of a rate-monotonic scheduling module for our real-time operating system in Section 10.4.3. [FCTS09] presents an enhancement of the Linux scheduler through the implementation of another deadline orientated scheduling strategy. The paper describes the adaptations of the Linux kernel that are necessary to support earliest-deadline-first scheduling (see Section 3.2.2.1). The description of the Linux modular scheduling framework [FCTS09, Chap. 3] and [Mol07] contributes to our implementation of an interface for scheduling modules (see Section 11.6.4).

A priority orientated scheduling module is discussed in Section 10.4.3. It is based

on First-In-First-Out (FIFO) and Round-Robin (RR) scheduling methods. Details on these scheduling strategies can be found in [Tan08, Sect. 2.4] and [Noe05, Sect. 9.2].

Preemption and System-Calls

In monolithic operating system kernels, system-calls are a major obstacle when it comes to providing real-time guarantees because they often exhibit unpredictable timing behavior. For example, a system-call may be used to read inputs from a device driver, the exact timing of which is unknown. Furthermore, the access to the device may block access for other tasks; the call can thus be seen as a critical section. The notion of real-time thus entails that system-calls, and thereby the operating system kernel, have to be preemptable. The state-of-the-art solution to this problem is implemented by the RT-Preempt Linux extension [DW05], the key idea of which is to secure critical sections using mutexes instead of spin-locks. The RT-Preempt Linux extension thereby avoids busy waiting and allows preemption within critical sections in the kernel. However, this strategy suffers from priority inversion [LR80, Sect. 4], where the execution of a high priority task is delayed by a medium priority task [BMS93]. This is indeed a problem with most operating system kernels that implement some form of priority-based task model [MT92]. We will discuss this situation in Section 6.2.1 and investigate how the problem is solved for the RT-Preempt Linux extension.

Our work, in contrast, allows to redirect system-calls from tasks to so-called *system-call handler threads*. Whenever such a call is invoked, its execution is redirected to the handler, which introduces a form of asynchronous execution of system-calls (even though the caller still has to wait for termination of the respective system-call). Such handler threads are preemptible even in most critical sections. Similar ideas can be found in the area of secure system design, where *system-call interpositioning* [Gar03] is frequently used in sandbox environments. For example, Ostia [GPR04] introduces a *delegating architecture*, which allows to delegate calls to distinguished handlers. We defer the discussion of the details of system-call handler threads and how the problem of priority inversion is treated to Section 10.4.2.1 and Section 11.8.

1.4. Structure of this Document

This document is structured into four parts as illustrated in Figure 1.1. The figure also presents the overall research process of this dissertation and shows dependencies between sections.

Part I describes real-time computer systems and defines the role of real-time operating systems. Basic terms and concepts are introduced which are used in later chapters. This includes a definition of the term *off-the-shelf* architecture. General implementation details on the Linux kernel are not discussed in this part.

In Part II several real-time operating systems are discussed. This part provides an in-depth examination of three systems. A method for evaluating operating systems is developed and applied to the analysed systems.

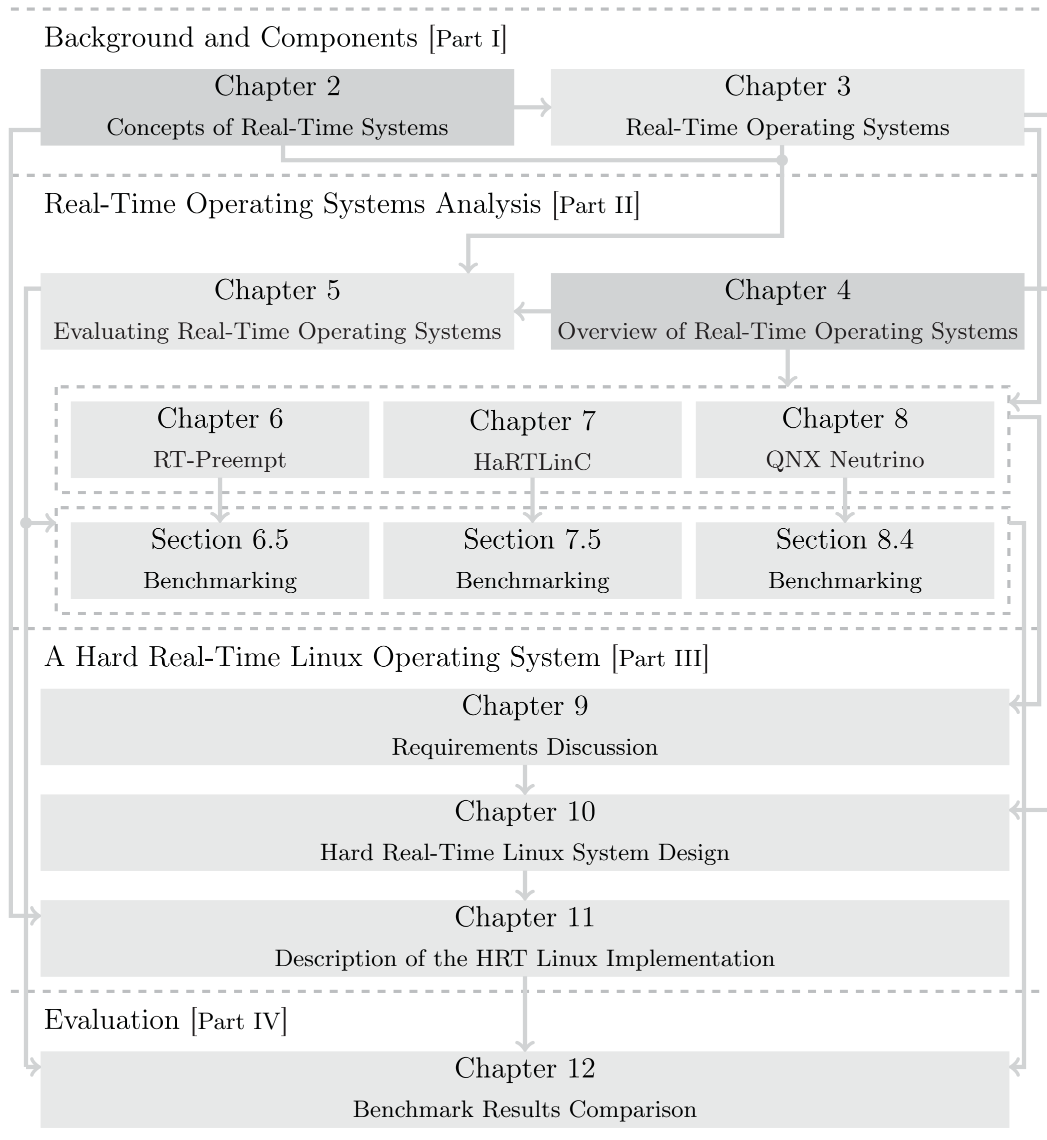

Figure 1.1.: Overview of the thesis

Part III identifies and summarizes requirements from the previous system analyses. Based on this requirements discussion, the design for a new real-time Linux extension is developed. The last chapter in this part describes technical details on the implementation of the design.

Part IV compares the developed real-time extension to other systems and gives a comprehensive conclusion of the results elaborated in this thesis.

Part I.

Background and Components

2

Concepts of Real-Time Systems

This chapter introduces real-time systems to the reader and explains the concepts and classifications of such systems. Later in this thesis we will concentrate on computer systems inside a larger system, the real-time system, and will isolate the part of a real-time operating system from the real-time computer-system. As an introduction we will start with a view at the whole real-time system. According to Kopetz [Kop97, Sect. 1.1], it is reasonable to separate a real-time system into three distinguished sub-systems called clusters:

- Operator
- Real-time computer system
- Controlled object

The operator and the controlled object are regarded as the environment of the real-time computer system. The environment and the computer system are connected with two interfaces. The computer system is influenced by the operator via the man-machine interface and must react to stimuli from the controlled object via the instrumentation interface. The real-time computer system itself can be divided into a real-time operating system and one or more real-time jobs respectively real-time tasks[1] as shown in Figure 2.1.

In the following sections we will concentrate on real-time computer systems. The area of the environment of real-time computer systems is not considered. Information about the man-machine interface respectively the instrumentation interface, the operator and the controlled object can be found in [Kop97, Sect. 1.2]. For the purpose of simplicity real-time computer systems are referred to as real-time systems in the following chapters. Hardware issues of real-time systems are discussed in Section 2.3 and Section 2.4.

2.1. Definition and Classification of Real-Time Systems

A real-time system is an information processing system which has to respond to stimuli within a finite and specified time. This basically means that results of computations

[1] A technical definition of the term task is given in Section 3.1.

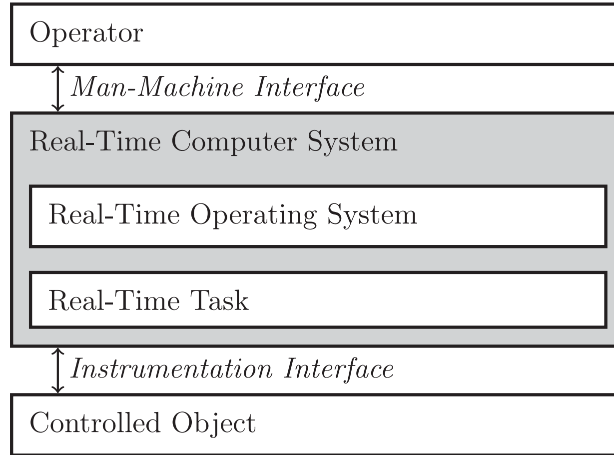

Figure 2.1.: Real-time system

not only must be correct but also be present at a given time. This point in time is typically referred to as the *deadline*. A computation can be deemed to have failed if it is not completed before the deadline for this result has lapsed. A (hard) real-time deadline must be met, regardless of system load.[2]

In the following sections real-time systems are considered from different perspectives. Several properties can be assigned to each perspective. The first perspective (*general*) describes a common view of the system. The second perspective (*application*) includes characteristics of the system depending on the application (outside the computer system). The last perspective (*design and implementation*) depends on the design and implementation of the system (inside the computer system). Additional information on this topic can be found in [FGR+90, Sect. 1.2] and [Kop97, Sect. 1.5].

Perspective	Characteristic
General	– Proprietary versus open
	– Centralized versus distributed
Application	– Fail-safe versus fail-operational
	– Hard versus soft real-time
Design and implementation	– Event-triggered versus time-triggered

Table 2.1.: Characteristics of real-time systems

2.1.1. Proprietary versus Open

This characteristic is probably of minor importance for the development of a real-time system. Nevertheless, it should be mentioned at this point that real-time systems are quite differentiated at this criterion.

[2] In Section 2.1.4 real-time deadlines will be discussed in detail.

2.1.2. Centralized versus Distributed

At this point some concepts of distributed real-time system are briefly explained. A demarcation to centralized systems is shown. There will be no evaluation of the advantages or disadvantages of distributed systems to centralized systems. An answer to the question, why and when a distiubuted solution is a good approach for a real-time system, is given in [Kop97, Chap. 2] and [Lei07, Sect. 2.1.2].

A distributed system is, as the name clearly suggests, divided into several subsystems. In contrast to centralized systems, there is no shared address space over the whole system. This raises the need for defining an interface to synchronize the subsystems, to be able to share data and to link subsystems together so that properties that have been established at subsystem level will hold at the system level. What is referred to as the *linking interface* ensures encapsulation of the subsystems, hiding all internal details and local interfaces.

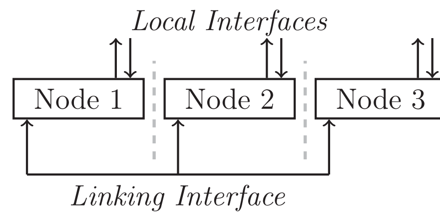

Figure 2.2.: Linking interface

The linking interface requires a *communication system* which has to fulfill some requirements to be qualified for a real-time system:

protocol latency The time offset between sending and receiving a message has to be short enough. For an example, the time interval after sending a message until receiving the acknowledgement from the opposition can be denominated as protocol latency.

error detection It is extremely important for a communication system in the real-time array to take note of bad transmissions. Error detection techniques allow detecting such errors, while error correction enables reconstruction of the original data.

end-to-end acknowledgment To provide a dependable service it is essential for the receiver to send an acknowledgement back to the sender of the message that the message was received after successful delivery.

2.1.3. Fail-Safe versus Fail-Operational

Fail-safeness is a characteristic of the application, not the computer system. A system is fail-safe when there are one or more safe states in the environment that can be reached in case of an error. At a railway signaling system for example, in case of a system failure all signals are set to red to stop all trains and prevent a disaster. In

fail-safe applications the computer system must have a high error detection coverage. This means that the probability that an existing error is detected has to be as high as possible. The interval between the start of an error and the detection of the same is called error detection latency. Fail-safe applications are characterized by a low error detection latency.

A system is fail-operational if no safe state can be achieved in the event of a system failure. For example, a safe error state can not be identified for an aircraft. For fail-operational applications the computer system must maintain a minimum level of service, even after encountering an error.

2.1.4. Hard versus Soft Real-Time

The term real-time describes the property of a system to deliver certain results within a predetermined period of time. The processing of the data does not necessarily have to be carried out in a particularly fast way. It is essential that it is fast enough to fulfill the timing constraints made by the real-time application.

The precise point in time at which a job[3] becomes ready for execution is called the release time of the job. The job can be scheduled by the system at or after the release time of the job. If a job is released when the system begins execution, the job has no release time. In contrast, the deadline of a job is the precise point in time at which its computation (execution) must be completed. In this context, the relative deadline of a job is the time between the release time and the deadline. A job has no deadline if its deadline is at infinity.

As an example, a system which has to schedule a specific job periodically is considered (Figure 2.3). A job (J) of this stream ($J_0, J_1, \ldots, J_n$) is executed every 50 milliseconds. Assuming that the system starts execution of the first job 10 milliseconds after system start, the release time of the job J_i is $10+i\cdot 50$ milliseconds, for $i = 0, 1, \ldots$. Supposing each job must be completed before the subsequent job starts execution, the deadline for J_i is the start of J_{i+1}. If the jobs must be completed sooner, maybe after 20 milliseconds, the deadline for J_i is the start of $J_i + 20$ milliseconds. The time between the release time of a job and its deadline (the relative deadline) is the response time of a job. In this case the response time is 20 milliseconds.

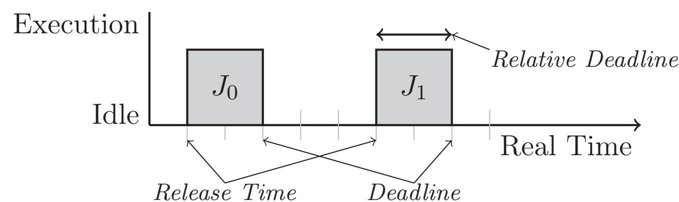

Figure 2.3.: Timing requirements of jobs

To describe the quality of real-time, the reliability of the fulfillment of deadlines

[3] A job is an unit of work that can be scheduled by the system. At this point it is not important to distinguish between variety levels of jobs like Task, Thread, etc.

has to be defined. For this purpose, usually a distinction between hard real-time and soft real-time is made:

Soft The response time is guaranteed only in a statistical way. The reliability of meeting a deadline reaches an acceptable average or another statistical criterion. The result is useful (can still be used) even after the deadline has passed.

Firm The response time is strict. A result is of no value after the deadline has passed. Late results have no benefits for the system.

Hard The response time is strict. A result is of no use after the deadline has passed. Missing a deadline has fatal consequences for the system.

A short comparison of hard real-time and soft real-time characteristics is given in Table 2.2. The terms firm real-time and hard real-time are summarized in one group. Differentiations are only made between soft real-time and hard real-time. More information about these characteristics can be found in [Kop97, Sect. 1.5.1].

Characteristic	**Hard real-time**	**Soft real-time**
Response time	Hard-required	Soft-desired
Peak-load performance	Predictable	Degraded
Control of pace	Environment	Computer
Safety	Often critical	Non-critical
Size of data files	Small	Large
Redundancy type	Active	Checkpoint-recovery
Data integrity	Short-term	Long-term
Error detection	Autonomous	User assisted

Table 2.2.: Hard real-time versus soft real-time

Soft real-time response means that the duration of a job usually does not exceed the specified time limit. This can be shown by measurements and static calculations. A soft real-time system is allowed to miss its deadline infrequently. Soft real-time is for example usually sufficient for multimedia applications. In contrast to that, hard real-time behavior means that a provable upper bound for the duration of a job can be given based on hardware specifications and model calculations. This is important for critical applications such as for controlling engineering.

2.1.5. Event-Triggered versus Time-Triggered

In short it can be said that a real-time system is time-triggered, if control signals are based solely on the progress of a global time notation. On the other hand a real-time system is event-triggered, if control signals depend only on the occurrence of events.

As described in [Kop97, Chap. 14], a time-triggered system starts all of its activities at predetermined points in time. No interrupts are allowed to occur[4]. For instance, the state of an input signal could be checked periodically every ten milliseconds. The system reacts accordingly, if a state change is detected. In contrast, an event-triggered system starts acting whenever a significant event occurs. For example, a change of the input signal from above would generate an interrupt request, which causes the interrupt service routine to be executed. The interrupt service routine reacts to the event.

In a time-triggered real-time system activities are initiated at predefined points in time. This requires intensive knowledge about the system, since all sorts of events have to be predictable. Anything that is not completely known at the system design stage can not be managed at all. The event-triggered system with its interrupt mechanism offers flexibility and low response time. The described limitation of the time-triggered system is not valid for an event-triggered system. If there is no upper limit to the frequency of interrupts (the occurrence of events), the event-triggered system can be completely occupied with interrupt handler execution. The load can become too heavy and the system cannot respond in time [KB03].

2.2. Requirements of Real-Time Systems

In [Kop97, Chap. 1] several general functional and metafunctional requirements for real-time systems are identified. An overview of these requirements is given in Table 2.3.

Group	**Requirement**
Functional	– Data collection and monitoring – Process control – Direct digital control – Man-machine interaction – Error handling – Archiving
Temporal	– Predictability – Error detection latency
Dependability	– Reliability – Safety – Maintainability – Availability
Architectural	– Testability – Backwards and forwards compatibility – Standardization of components

Table 2.3.: Requirements of real-time systems

[4] With the exception of a periodic clock interrupt.

To describe the requirements of real-time systems it is necessary to introduce some new concepts. The state of a controlled object (Figure 2.1) at any point in time can be described by its state variables at that moment. A subset (or all) of these state variables that are of particular interest are called *real-time entities*. A real-time entity which can be changed by a subsystem is in the *sphere of control* of that subsystem. The real-time entity can not be modified but observed outside of the sphere of control. A *real-time image* is a temporally accurate picture of a state variable at a particular moment in time. The real-time image is only accurate for a limited time interval [Kop97, Chap. 5].

2.2.1. Functional Requirements

Functional requirements of a real-time system deal with the functions that a real-time system must perform. Data observation, collection and monitoring are the main topics in this field. An observation of a real-time entity is represented by a real-time image. The scope of that image depends on the dynamics of the controlled object. After a certain time has elapsed the data has to be marked as outdated. The collection and the transformation of a sequence of real-time entities to standard measurement units for averaging and measurement error reducing is called *signal conditioning*. Signal conditioning requires the data to be checked for accuracy afterwards. If the resulting real-time image is correct the data element is called an *agreed data element*.

A real-time system must inform the operator of the current state of the controlled object and must also assist the operator in controlling the object. In this context an extensive data logging and data reporting subsystem is often included as part of the man-machine interface. This subsystem requires that every data entry is connected to an exact time stamp.

2.2.2. Temporal Requirements

One important requirement on a real-time system is predictability. The classical hard real-time design paradigm is based on worst-case assumptions and static resource allocation. Because of this paradigm, unlike general purpose systems, real-time systems may cause a waste of the available resources of the computer system and thus increasing the overall system cost. Real-time systems must be predictable enough to guarantee the required performance.

It is important for a real-time system to detect any error within the control system within a short time and with a high probability. It is then possible to bring the system into a safe state or to perform some corrective actions. The *error detection latency* is of particular interest in the area of distributed real-time systems in context with the linking interface.

To update a real-time image with the data of the corresponding real-time entity, some actions have to take place (e.g. request sensor data; for distributed real-time systems messages containing observation information have to be sent). The time delay between the start of the observation event until the real-time image gets updated is

called *lag*. For the periodical observation of data (respectively the periodical writing of signals) the lag for each observation is variable. This is referred to as *delay jitter* and it brings an additional uncertainty into the real-time system. It is desirable to keep the delay jitter as small as possible and hence the lag as a constant delay.

2.2.3. Dependability Requirements

To describe the dependability of a real-time system the following five attributes are of importance [LAK92]:

Maintainability rates the time that is needed to *repair* a system after a failure occurred. To define a maintainability measure a *mean-time to repair* is introduced.

Availability is the assessment of correct service of the system. With respect to the occurrence of failures according to a service and the time to repair the failures, the availability is measured by the fraction of time that the system is ready to provide the service.

Reliability describes the probability of a system to provide a specified service within a defined timeframe. In relation to a constant failure rate of the system ($\lambda failures/hour$) the reliability at time t is given by $R(t) = exp(-\lambda(t - t_0))$.

Safety in this context means dealing with critical failure modes. The two failure modes malign and benign are distinguished. The system and the subsystems that are critical for the safe operation of the system must be protected by stable interface to eliminate the possibility of error propagation.

Security is the ability of a system to prevent unauthorized access to information or services.

These metafunctional attributes cover the quality of service a real-time system delivers to its users. A detailed view on dependability requirements can be found in [Kop97, Sect. 1.4].

2.2.4. Architectural Requirements

The field of architectural requirements for a real-time system describes properties like testability and compatibility. In a market economy, these attributes are essential for a project to choose between a variety of operating systems. The entire real-time system has to provide a strategy to meet the following needs:

Testability is the property of a system to support the integration into a given test context. It is the prerequisite for a successful validation of the system. Most safety-critical systems require a prestigious certification.

Backwards and forwards compatibility assumes that the system can work with input generated by an older product. Forward compatibility aims at the ability of a design to easily accept input intended for later versions of itself.

Standardization of components increases the compatibility of the system with other components. In addition, the know-how that is needed to deal with the system is unified compared to similar systems.

2.3. Hardware for Real-Time Systems

A wide range of application for real-time operating systems can be found in the sphere of embedded systems. Embedded systems are included in a variety of applications and devices mostly invisible to the user (for instance medical supplies, electrical household appliances, consumer electronics, automobiles, aircrafts, ...). Complex systems which can be found in vehicles or aircrafts are built up from a variety of connected embedded systems. Embedded systems are usually adapted specifically to a certain job or scope. An optimised mix of specialized hardware and software is often chosen for the dislocation of an embedded system. In general, such a mix is optimised concerning the scope of the entire system and is highly restricted under the following boundary conditions:

- Minimum costs
- Low place consumption
- Low memory footprint
- Low energy intake

Single components like processor and main memory are often based on advancements of older components. This facilitates the long-term applicability and the acquisition of replacement parts. In many applications, the use of an older processor architecture can help to reduce costs.

Hardware which was developed for embedded systems is often designed to ensure compliance with real-time requirements (also see [YMBYG08, Chap. 3]). High availability and defined response times are frequently requested requirements for an embedded system and thus for its operating system. Moreover, many embedded systems are permanent operational which expects a low delinquency rate.

The structure of a processor is called the processor architecture. The various architectures differ from one another primarily on the nature and level of some functional units of the processor. A series of processors of an architecture can form a processor family if they only differ in some peripheral properties. The term off-the-shelf processor refers to architectures which are used primarily in desktop computers. These processors belong mainly to the x86 family and are developed as hybrid CISC/RISC architectures. In Section 2.4.1 we will have a look at the x86 family processors. Processors which are specifically designed for real-time systems can be contrasted with off-the-shelf processors generally associated with the embedded application. They have significant advantages in terms of cost and power consumption.

2.4. x86 Instruction Set Architecture

Since the x86 micro processor architecture family is the main architecture for the aforementioned off-the-shelf systems we will examine this architecture more closely below. For more detailed information please refer to [Wik13].

2.4.1. General Remarks and Design

The x86 architecture has been implemented in processors from Intel, Cyrix, AMD, VIA and many other companies. Derived from the 8086 processor from Intel many additions and extensions have been added to the x86 instruction set over the time. For some advanced features, x86 compatible processors may require license from Intel and AMD. The term x86 derived from the fact that early successors to the 8086 chip also had names ending in *86*. For licensing reasons, word marks were used as identifiers for later developments (e.g. Pentium, Athlon, ...). In the following section different identifiers are introduced for variations of the x86 architecture. They are sometimes used as synonymous in literature. In this work, only the identifiers x86 and x86_64 will be used to refer to the architecture.[5]

x86, x86_32 is the name for the x86 32 bit architecture as introduced above in this section.

i386, IA-32 was the first incarnation of x86 to support 32 bit computing. Technically the term i386 refers to the 80386 processor from Intel. In literature the names x86, x86_32, i386 and IA-32 are used interchangeably to describe the x86 32 bit architecture.

IA-64, Itanium is a 64 bit architecture introduced by Intel for explicitly parallel instruction computing (EPIC). This architecture can not be compared to the x86 family at all and is only listed here to avoid confusion.

x86_64, x64 is an extension of the x86 architecture that provides 64 bit computing. It supports larger amounts of virtual and physical memory compared to its predecessors. x86_64 also provides 64 bit registers and numerous other enhancements which are not considered here.

Intel 64, EM64T, IA32e is also known as *Extended Memory 64 Technology* and represents an extension from Intel to the IA-32 (x86) architecture. With the introduction of Intel 64 as implementation of the x86_64 architecture the Itanium architecture is diminishing in importance.

AMD64 is the implementation of the x86_64 architecture from AMD. It has some technical differences to Intel 64. Since the gap between AMD64 and Intel 64 is hardly noticeable in the context of the operating system, it is not of importance to investigate this alterations in detail here.

[5] If it is needed to distinguish between company-related dialects, the name of the company will be attached.

Some instructions differ in availability and functionality between Intel 64 and AMD64. However, these are mainly relevant for compiler construction. Another difference between both architectures is the missing I/O memory management unit (IOMMU) on Intel 64 chips. This leads to the fact that direct memory accesses (DMA) above 4 gigabyte for devices which do not support 64 bit addressing is not possible on Intel 64 processors. In order to compensate this disadvantage, Intel introduced the Intel Virtualization Technology (Intel VT, IVT) which can be partly compared to IOMMU. Both techniques are not of significant importance for this thesis and hence are not considered further.

2.4.2. Hyper-Threading Technology

The HT technology is used to improve parallelization of computations performed on x86 processors. Two virtual cores are addressed for each processor core that is physically present. The operating system can schedule two threads on one processor core but the descision which thread is actually executed can not be influenced. Most real-time operating systems (including this thesis) require hyper-threading to be deactivated.

2.4.3. APIC Architecture

The advanced programmable interrupt controller (APIC) is designed to solve interrupt routing efficiency issues in (x86) multicore systems. It consists of a local component (local APIC) integrated into the processor itself, and an I/O APIC on a system bus (Figure 2.4).

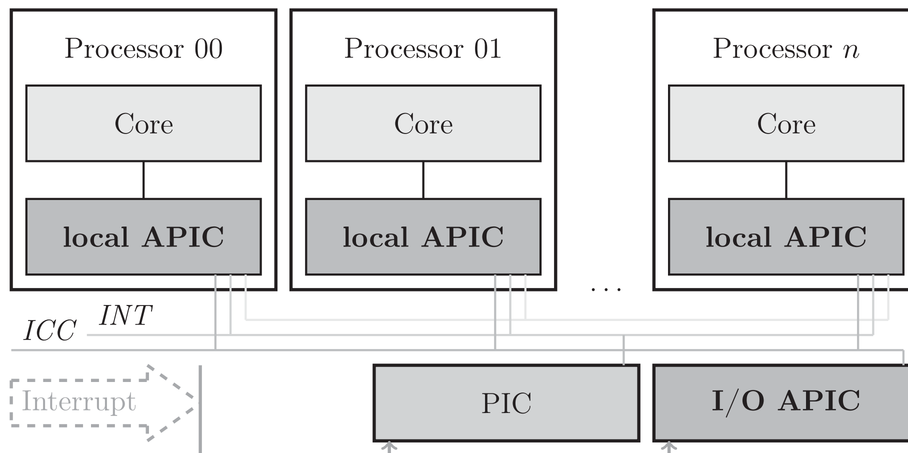

Figure 2.4.: APIC architecture

A local APIC manages all external interrupts for a specific processor in an SMP system. There is one local APIC in each CPU in the system. The details of the APIC design are not important for this thesis. We will only list two features here which are important for the implementation of the operating system presented in Part III.

Inter processor interrupts (IPI) are used for one processor interrupting another processor in a multiprocessor system. Details on IPI will be discussed later in Section 10.2.1 and Section 11.4.4.

APIC timer is a high-resolution timer that can be used in different modes. The operating system described in Part III can make use of this timer. Technical details on the implementation can be found in Section 11.3.2.

2.4.4. Time Stamp Counter

The time stamp counter stores (or counts) the number of cycles since system start. It is a 64 bit register available on each processor in the system. With this register a high-resolution and low-overhead way of getting CPU timing information is provided. There is no assurance that the time stamp counters of multiple CPUs are synchronized. In such cases, getting reliable results is only possible by locking a thread to a single CPU. Details for using the time stamp counter register are presented in Section 5.4.1.

2.4.5. Problems with Hard Real-Time on the x86 Architecture

Real-time behavior on x86 machines can be highly influenced by device drivers. For example, if a device grabs the PCI bus for long periods during DMA (direct memory access) activity, it can introduce significant latencies in the system. In Linux operating systems a device driver is part of the kernel respectively is executed in the kernel mode and has almost unrestricted access to kernel data structures. A badly programmed device driver module can disrupt the behavior of real-time applications with constrained timing requirements.

The x86 architecture supports the so called system management mode (SMM) as an operating mode in which all normal execution of the operating system is suspended. Special separate software like firmware is executed in high-privilege mode and takes CPU time away from the operating system. SMM is entered via the system management interrupt (SMI) which cannot be overridden or disabled by the operating system. SMI can destroy real-time behavior.

3

Real-Time Operating Systems

Real-time operating systems (RTOS) are operating systems with additional real-time functions for the unconditional adherence to time constraints and the predictability of process behavior. Such operating systems must be able to guarantee the compliance of defined response times even in the worst case. This mainly concerns the areas of scheduling, memory management and interrupt handling. In addition to this a RTOS must support an analytical analysis of its temporal behavior under all specified load and fault conditions [Kop97, Chap. 10].

This chapter describes basic techniques and structures of real-time operating systems and discusses some dynamic properties. Basic scheduling concepts are discussed in Section 3.2. Section 3.5 gives an overview of the POSIX standard for RTOS. Finally, Section 3.6 deals with capabilities and necessary enhancements to qualify the Linux kernel to be real-time. Architectural features of the kernel will also be discussed.

3.1. Task Management

In the following the term task is understood as a process from kernel view. A process is a logical relationship of different threads which all serve a common purpose. A process is an instance of a computer program that is being executed. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. Threads are different execution points of the same process respective task. Figure 3.1 illustrates the meaning of the terms process, task and thread.

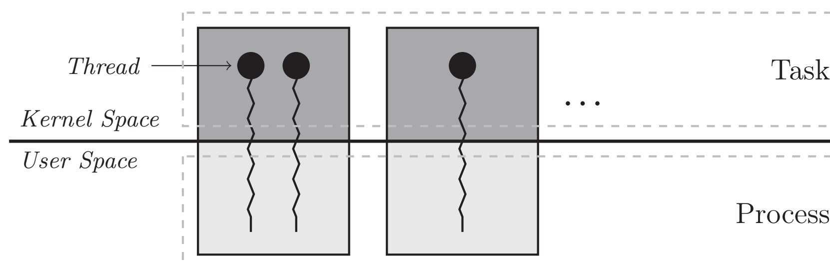

Figure 3.1.: Internal and external view on processes

Below it is assumed that a task always has exactly one thread. In other words, for the sake of simplicity, the terms task and thread are synonymous unless specified otherwise. The execution of a task is managed by the kernel, particularly the scheduler which will be discussed later in this chapter (Section 3.2).

3.1.1. Task States

In most operating systems, a task can be in the state *inactive*, *running*, *ready* or *blocked*. In state *running* the execution of commands on the CPU takes place. The task is *blocked* if it waits on an event. *Ready* means that the task could be executed but the resource (CPU) is busy with the execution of another task. During the creation of the task and after termination the task is in state *inactive*. Figure 3.2 illustrates the different states a task can have. The activation of the task and the transition from *ready* to *running* is decided by the scheduler. The scheduler can also put the task back from *running* to *ready* state if for example the time slice allocated for this thread has expired. If the execution cannot proceed for some reason the kernel puts the thread in the suspension queue and the task enters the *blocked* state.

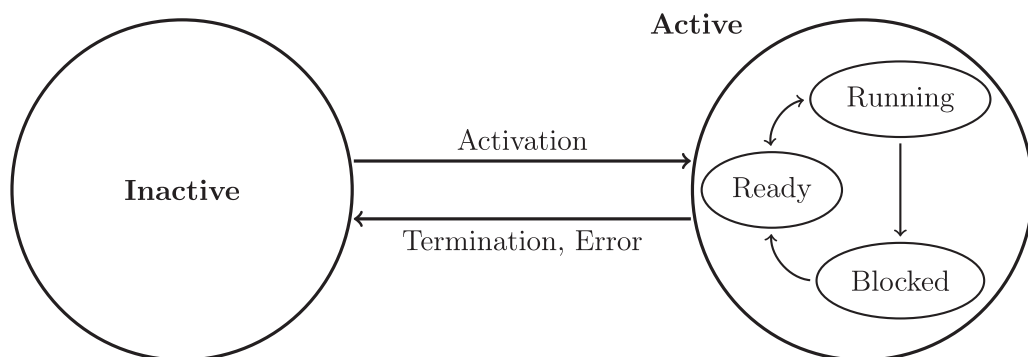

Figure 3.2.: Preemptive tasks

When a new task is activated by the kernel, the kernel allocates memory for the new task and loads the code of the process into the allocated segment if needed (threads inside the same process share one instance of the code). A data structure is needed by the kernel for each thread to control and schedule the tasks. This data structure is called the thread control block (TCB). It contains stack and code pointer, status variables and a reference to the process the thread lives in (process control block, PCB). Additional information on process and thread relations can be found in [Tan08, Sect. 2.1.6].

3.1.2. Task Hierarchy

In POSIX like operating systems, tasks are bounded into a distinct task hierarchy. If a new process is created, the new process is associated as a *child* with the *parent* process. Child processes of one parent process are related as siblings. Depending on the operating system a child process inherits a copy of the parents process control

block on creation. Task hierarchy and process creation will be discussed later in Chapter 11. Further information can be found in [Lov10, Chap. 3].

3.1.3. Task Types

Based on the way real-time tasks recur over a period of time, it is possible to classify them into three main categories. We assume all tasks in the system to be preemptible at any point in time.

Periodic tasks are characterized by three parameters: the period P, the computation time C and the relative deadline D. The task T generates a job at each integer multiple of P, and each such job has an execution requirement of C execution units, and must complete execution before its deadline D. The jobs are independent from each other. Each task does not interact (e.g. exchanging messages) with other jobs.

A schedulability test for a set of n periodic tasks states that the utilization factor:

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \tag{3.1}$$

must be less than or equal to 1 ($U \leq 1$). The concept of periodic tasks was first introduced by Chung Laung Liu [LL73] and has shown remarkable use for the modeling of recurring processes in real-time systems.

Sporadic tasks recur at random points in time. Like periodic tasks, they are characterized by three parameters: the minimum separation G, the computation time C and the relative deadline D. The minimum separation between two consecutive instances of the task implies that once an instance of a sporadic task occurs, the next instance cannot occur before G time units have elapsed.

Aperiodic tasks can arise at random points in time. They are in many ways similar to a sporadic task. The minimum separation time between two consecutive instances can be 0. That is, two or more instances of an aperiodic task might occur at the same time.

3.2. Scheduling

In an environment with more than one task being in the *ready* or *running* state simultaneously, a method by which these tasks are given access to CPU time is needed. A task or process scheduler handles the execution of tasks on one or more CPUs and balances the load of the system. In this section we will focus on task scheduling in real-time operating systems.

Several classifications of real-time task scheduling algorithms exist. Based on how the scheduling points are defined a scheduler can be classified into one of three main types which will be discussed in this section.

3.2.1. Clock Driven Scheduling Strategies

A clock driven scheduler determines the scheduling points by the interrupts received from a clock. This kind of scheduling strategy is simple and efficient. They follow a static scheduling plan which is developed offline. None or only a few decisions need to be made at runtime. Therefore, these schedulers incur very little run time overhead. Handling periodic and sporadic tasks is not possible since the exact time of occurrence of these tasks can not be predicted. For this reason, this type of scheduler is also called static scheduler.

A clock driven cyclic scheduler is discussed in Section 10.3.2.

3.2.2. Event Driven Scheduling Strategies

For event driven schedulers, scheduling points are defined by certain events which precludes clock interrupts. In event driven scheduling, the scheduling points are defined by task events. In contradistinction to clock driven strategies event driven schedulers can handle aperiodic and sporadic tasks with some restrictions. In this section we will briefly outline two types of event driven schedulers.

3.2.2.1. Earliest-Deadline-First Scheduling

Earliest-deadline-first (EDF) or least-time-to-go is a dynamic scheduling algorithm which makes its decisions so that deadlines for all tasks are met. The time points are always considered for scheduling, which either a new task is started or a active task is finished.

- All waiting tasks are sorted in ascending order of deadlines.
- The task which must be completed first, receives the CPU.

A task set is schedulable under EDF, if it satisfies the condition that the total processor utilization (Equation 3.2) due to the task set is less than 1.

EDF is a well discussed algorithm in literature. Readers with further interest can find information in [Liu00, Sect. 6.2.2], [Kop97, Sect. 11.3.1] and [SRS98].

3.2.2.2. Rate-Monotonic Scheduling

A classic dynamic preemptive algorithm for scheduling a set of periodic independent tasks with static priorities is the Rata-Monotonic algorithm (RM) [LL73]. The following assumptions are valid for all tasks of the set:

- All tasks in the set (for which hard deadlines exist) are periodic.
- The deterministic deadline of each task T_i is equal to its period P_i.
- No resource sharing; all tasks are independent from each other.
- The computation time C_i of each task is known and constant.

- Context switch times and other thread operations are free and have no impact on the model.

The static priorities are assigned on the basis of the cycle duration P_i of the task. The shorter the cycle duration is, the higher is the task's priority $p()$. The priorities are linear with the rate. At runtime the process with the highest priority is always selected.

$$p(T_i) = \frac{1}{P_i}$$

Liu and Layland [LL73] proved that for a set of n periodic tasks with unique periods, a feasible schedule that will always meet deadlines exists if the CPU utilization (see Equation 3.1) is below a specific bound:

$$U \leq n \cdot \left(\sqrt[n]{2} - 1 \right)$$

It follows that the schedulability test for RM is:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n \cdot \left(\sqrt[n]{2} - 1 \right) \tag{3.2}$$

Equation 3.2 offers a worst-case condition that characterizes schedulability of a set of tasks under the rate-monotonic algorithm. When the number of processes tends towards infinity the bound will tend towards:

$$\lim_{n \to \infty} n \cdot \left(\sqrt[n]{2} - 1 \right) = ln2 \approx 0.693$$

This means that with the rate-monotonic algorithm if not more than 69.3% of available processor cycles are used, an optimal scheduling can be guaranteed where each task meets its deadline [Liu00]. This is, in fact, quite a pessimistic scenario. Task sets are often schedulable by the rate-monotonic algorithm at much higher utilization levels, even with worst-case phasing. In general, it has been shown that when periods are generated from a uniform distribution and P_j evenly divides P_i for $1 \leq j \leq i$, the breakdown utilization will be in the 88% to 92% range [Leh90].

3.2.3. Hybrid Scheduling Strategies

A hybrid scheduler uses both clock interrupts as well as event occurrences to define its scheduling points. This class of scheduling strategies can fully handle aperiodic and sporadic tasks but loses some accuracy while treating periodic tasks. We will discuss two types of scheduling strategies in this chapter.

3.2.3.1. First-In-First-Out Scheduling

The First-In-First-Out (FIFO) scheduling strategy, equivalent to First-Come-First-Served (FCFS), describes the principle of a queue processing technique by ordering

tasks where they leave the queue in the order they arrived. Each task in this scheduling class is executed until it is suspended by a task at a higher priority level or it releases the CPU by itself. Since context switches inside the class only occur in these situations, and no reorganization of the task queue is required, scheduling overhead is minimal. As long as every task eventually completes or releases the CPU by itself, there is no starvation. In an environment where some tasks might not complete, there can be starvation. In addition, throughput can be low, since long tasks can hog the CPU for a long time.

3.2.3.2. Round-Robin Scheduling

The Round-Robin (RR) method allows all tasks (each one at a time) to run on the CPU for a short period (timeslice). The tasks are managed in a queue. The foremost task is given access to the CPU until its timeslice expires. After that the task is added to the tail of the queue. The next task is selected according to the FIFO principle. A task can release the CPU before the timeslice expires. In this case, the resources are immediately reallocated and the CPU is occupied by the next task.

To add priority level scheduling to this method the scheduler holds a task queue for each priority level. Only the queue with the highest priority is treaded. If there are no tasks in this queue then the next priority level is considered.

3.3. Partitioning

Many real-time operating systems implement a concept called partitioning. Each partition has its own memory space and hosts at least one process. Processes within one partition should not cause adverse effects to other partitions. We will concentrate on the scheduling aspects of partitioning mechanisms. Further information concerning memory protection, resource allocation and inter-partition communication can be found in [LSOH07].

A partition scheduler will execute processes on a CPU inside a partition according to a specified sequence of time windows. The tasks in each partition run on that CPU only during the time window for that partition. All tasks in all other partitions are not allowed to run during this time window. The key aspect for the partition scheduling is the ability to guarantee CPU time for a set of tasks. The tasks inside a partition are scheduled by a secondary scheduler (Section 3.2). There are two major varieties of partition scheduling:

Static partition scheduling assigns a fixed bound of CPU time within a given period to a partition. This time is seen as an execution window in which no other task from any other partition can be scheduled.

Dynamic partition scheduling requests that a percentage of processing resources within a given period be reserved for a particular partition. A running task can be preempted by a task from another dynamic partition (e.g. higher task priority).

3.4. Main System Services

Apart from scheduling strategies a real-time operating system has to provide other important features. Most of them are highly dependent on the operating-system design and implementation. At this point we will only list some important features required of a real-time operating system, and in particular those that are normally absent in traditional operating systems. In Part II these features are examined for a range of concrete real-time operating systems.

Timer support with adequate resolution is one of the most important issues for an real-time operating system. Real-time applications often require the support of timer services with resolution of the order of a few microseconds. Traditional operating systems often do not provide time services with sufficiently high resolution.

Task priority levels which are not changed by the operating system to balance the system load or for other reasons are required. These static priority levels separate real-time tasks from *normal* system tasks.

Fast task preemption is needed to ensure that whenever a high priority task becomes ready for execution, an executing low priority task instantly yields the CPU to it. The time duration for which a higher priority task waits before it is allowed to execute is quantitatively expressed as the corresponding task preemption time.

Predictable and fast interrupt latency is required to keep the time delay between the occurrence of an interrupt and the corresponding treatment as short as possible. Furthermore, the treatment should be preemptive to ensure task preemption.

Interprocess communication shares critical resources among real-time tasks. This includes events and other synchronizing mechanisms.

Memory management with support for a real-time task to control paging and swapping is needed. Memory locking prevents a page from being swapped from memory to hard disk. In the absence of a memory locking feature, memory access times of real-time tasks can show large jitter, as the access time would greatly depend on whether the required page is in the physical memory or has been swapped out.

Error detection in case a real-time task has missed its deadline. The system should provide a mechanism for reporting such violations.

3.5. POSIX Standard

The *Institute of Electrical and Electronics Engineers* (IEEE) has specified a family of related standards to define the application programming interface (API) for UNIX like operating systems. The so called portable operating system interface (POSIX)

describes the interaction between the operating system and its applications. POSIX also defines a standard threading library API which is supported by most modern operating systems.

POSIX:2008 (IEEE Std 1003.1-2008) represents the current version of the standard. The specification of the user and software interface of the operating system is divided into four parts [Com08]:

Base definitions General terms, concepts, and interfaces common to all volumes of this standard, including utility conventions and C-language header definitions.

System interfaces and headers Definitions for system service functions and subroutines, language-specific system services for the C programming language, function issues, including portability, error handling, and error recovery.

Commands and utilities Definitions for a standard source code-level interface to command interpretation services (shell) and common utility programs for application programs.

Explanations Extended rationale that did not fit well into the rest of the document structure.

3.5.1. Real-Time System Profiles

A group of four profiles for real-time POSIX characterizing single-task and multi-task operating systems with and without file system support are known as IEEE Std 1003.13-2003 [Com08] (revision of IEEE Std 1003.13-1998) PSE51 to PSE54. This profiles are appropriate for the development and execution of realtime applications.

Minimal realtime system profile (PSE51) These systems are designed for unattended control of one or more I/O devices. Neither the interaction with users nor the access and management of file systems is required. The system consists of a single POSIX process, which contains one or more threads of control. The process is executed by a single CPU without memory management unit (MMU).

Realtime controller system profile (PSE52) PSE52 systems are an extension of the PSE51 profile. Support for a file system interface and asynchronous (nonblocking) I/O interfaces has been added to this profile. The file system can be implemented in memory (no mass storage is required).

Dedicated realtime system profile (PSE53) These systems are an extension of the PSE52 profile. Support for multiple processes has been added. The profile contains a uniform interface for device drivers and files, but no hierarchical file system. Since memory management hardware may be provided, the functionality of memory locking is provided. The hardware model for this profile assumes one or more processors, each with its own MMU, in the same system.

Multi-purpose realtime system profile (PSE54) PSE54 systems include all the functionality of the other three profiles. A mix of real-time and non-real-time tasks can be run at this level, some being interactive user tasks. Support for multiple multithreaded processes is required so that multitasking may be done by threads, processes, or both.

3.6. Real-Time and Linux

Many approaches to equip the basic Linux operating system with real-time features have been published. In this chapter we will have a look on the most distinct techniques. A more detailed discussion on this can be found in Chapter 6 and Chapter 7.

3.6.1. Real-Time Kernel

This method provides a compact real-time kernel placed between the hardware layer und the standard Linux kernel. The real-time kernel controls the execution of real-time tasks, intercepts the hardware interrupts and runs the standard Linux kernel as a background task. The real-time tasks run at the kernel level with kernel privileges. Linux and its user tasks run whenever no real-time task is ready to execute. Linux is preempted whenever a higher priority task (real-time task) becomes ready for execution.

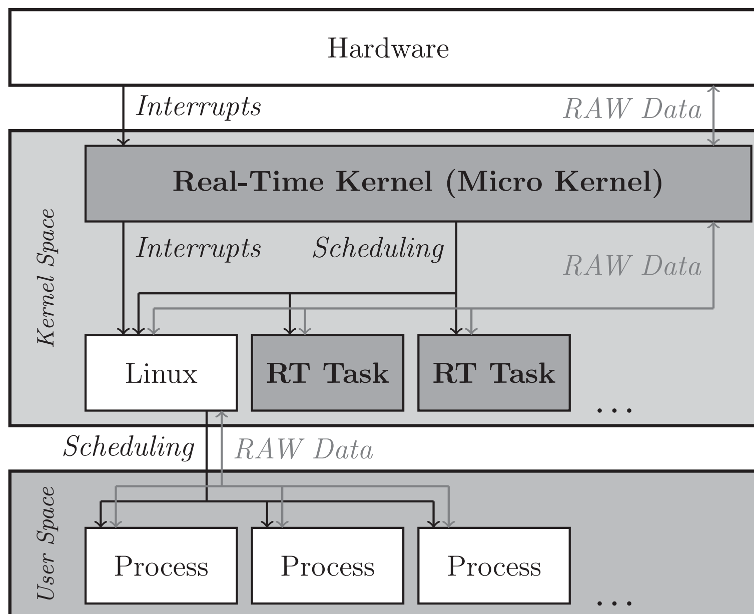

Figure 3.3.: Real-time kernel design

As shown in Figure 3.3 the hardware interrupt communication is completely abstracted by the real-time kernel (micro kernel). The standard Linux kernel cannot preempt any interrupt processing in the micro kernel. The original Linux functions for enabling and disabling interrupts (`cli`, `sti` and `iret`) have to be replaced by

emulating macros inside the Linux kernel. This will permit interrupts to be still available for the real-time tasks, even if the Linux kernel has disabled them. The micro kernel design is implemented in real-time operating systems like RTLinux [YB97].

3.6.2. Kernel Preemption

This strategy modifies the Linux kernel in a way that tasks are preemptible at almost every time. Normally, tasks inside the kernel mode (e.g. system-call) are not preemptible. Thus it is possible that the execution of a high priority task is delayed by a lower priority task in kernel mode. Additionally, threaded interrupt handlers are introduced to Linux. As a consequence, interrupts have an own priority level and can be interrupted by higher priority tasks. This design is implemented by the RT_PREEMPT kernel patch (Chapter 6).

3.6.3. Resource Reservation

Giving a resource exclusivity to a certain task eliminates the need for the task to wait for this resource. If a CPU is assigned to a task and no other task is able to be executed on this CPU the task can not be interrupted. In addition to the exclusion of other tasks from being executed on a certain CPU a more strict routing of interrupts is introduced to the Linux kernel. In Chapter 7 we will discuss this approach in detail.

Part II.

Real-Time Operating Systems Analysis

4

Overview of Available Real-Time Operating Systems

The variety and number of available real-time operating systems is enormous. This chapter provides a detailed overview of the current most popular real-time operating systems. The overview shown in Table 4.1 offers a first comparison of different real-time operating systems on the basis of some general characteristics.

Name Name and company name

Version Latest stable release and date

Source Source code is available

Hardware Supported platforms (selection)

64 64 bit architectures are supported

TPL Thread priority levels

Usage Marketing target

Kernel Kernel architecture

MP Multiprocessor support

Name	Version	Source	Hardware	64	TPL	Usage	Kernel	MP	Notes	Ref.
Atomthreads *community*	1.3 (2010)	open	AVR, STM8	no	256	embedded	micro	no	scheduler for embedded systems	[Law]
BeRTOS *Develer*	2.7.0 (2011)	open	ARM, AVR, PowerPC	no	2^{32}	embedde	micro	no	x86 emulated	[Dev]
ChibiOS/RT *community*	2.4.1 (2012)	open	x86, ARM, AVR, PowerPC, ...	no	128	embedded	micro	no	Intel 80386	[Sir]
Deos *DDC-I*	Q1 (2009)	closed	x86, PowerPC	no	2^{32}	embedde	micro	no	DO-178B level A certified	[DDCa]
DSPnano *RoweBots*	2 (2008)	available	PIC, M16C, ARM	no	2^{32}	embedde	micro	no	based on Unison, system on a chip	[Rowa]
eCos *community*	3.0 (2009)	open	x86, ARM, PowerPC, MIPS	yes	32	embedded	micro	no		[eCo]
ERIKA *Evidence*	1.5.1 (2010)	open	ARM, AVR, ...	no	1	embedded	monolithic	yes	OSEK/VDX RTOS	[Evi]
embOS *Segger*	3.86d (2012)	closed	ARM, AVR, ...	no	256	embedded	micro	no		[SEG]
FreeRTOS *Real Time Engineers*	7.1.0 (2011)	open	x86, ARM, AVR, ...	no	256	embedded	micro	no		[Rea]
HeartOS *DDC-I*	Q1 (2009)	closed	x86, ARM, PowerPC	no	2^{32}	embedde	micro	no		[DDCb]
HLRT *community*	2.6.27.19 (2013)	open	x86	no	64	general	monolithic	yes	Linux patch	[Efk05]
KURT *Kansas University*	1.23 (2000)	open	x86	no	256	general	monolithic	no	no hard real-time, defunct	[Uni]
Integrity-178B *Green Hills Software*	11 (2012)	closed	x86, MIPS, PowerPC, ARM	yes	512	embedded	micro	yes	DO-178B level A certified	[Gre]
LynxOS *LynuxWorks*	5.0 (2007)	available	ARM, Mips, PowerPC	no	512	embedded	micro	yes	full 1003.1, .1b & .1c	[Lyn]

Name	Version	Source	Hardware	64	TPL	Usage	Kernel	MP	Notes	Ref.
MontaVistaLinux *MontaVista*	6 (2009)	open	x86, ARM, ...	yes	1024	embedded	micro	yes		[Mon]
Neutrino *QNX Systems*	6.5.0 (2010)	closed	x86, ARM, PowerPC, ...	yes	64	embedded	micro	yes	full 1003.1, .1b & .1c	[Bla]
OpenComRTOS *Altreonic NV*	1.4 (2011)	available	ARM, PowerPC	yes	256	embedded	micro	yes		[Alt]
OSE *ENEA AB*	5.5 (2008)	closed	ARM, MIPS, PowerPC	no	32	embedded	micro	yes		[Ene]
PikeOS *SYSGO*	3.2 (2011)	closed	x86, MIPS, ARM, PowerPC	no	256	embedded	micro	yes	1003.13 PSE52	[SYS]
RTLinux *Wind River Systems*	3.2rc1 (2007)	open	same as Linux	yes	1024	general	micro	yes	1003.13 PSE51	[YB]
`RT_PREEMPT` *community*	2.6.31.6 (2011)	open	x86, ARM, SH	yes	64	general	monolithic	yes	Linux patch	[MG]
ThreadX *Express Logic*	G5.5.5 (2008)	available	x86, ARM, SH, PowerPC, MIPS	no	32	embedded	micro	yes		[Exp]
μC/OS-III *Micrium*	3 (2009)	closed	ARM, AVR	yes	64	embedded	micro	no		[Mica]
Unison *RoweBots*	4.0 (2008)	open	PIC, M16C, ARM, SHARC	no	2^{32}	embedde	micro	yes	system on a chip	[Rowb]
VxWorks *Wind River Systems*	6.7 (2009)	closed	x86, ARM, SH, PowerPC, MIPS	yes	256	embedded	micro	yes	1003.13 PSE52	[Win]
Windows CE *Microsoft*	7 (2011)	closed	x86, ARM, SH, PowerPC, MIPS	no	256	embedded	monolithic	yes	no UNIX like OS	[Micb]

Table 4.1.: Real-time operating systems overview

Some systems from Table 4.1 like DSPano are designed to have very low memory footprint and high modularity. This allows the systems to be used on a variety of different CPUs, even digital signal controllers. Figure 4.1 shows a simplified overview of memory footprints of different systems. To keep the size of such a system small, the system often consists only of a kernel containing a real-time scheduler and some task synchronization primitives. These *low memory systems* are not designed to deal with a variety of hardware nor to satisfy any dynamic terms. Real-time operating systems like VxWorks and PikeOS have also been developed to fit a small footprint in an embedded environment. However, they support a larger selection of hardware and have their own driver layer. QNX Neutrino also falls into this category of embedded non Linux operating systems. However, Neutrino has some properties that should be examined in greater detail later (e.g. micro kernel architecture and dynamic partitioning).

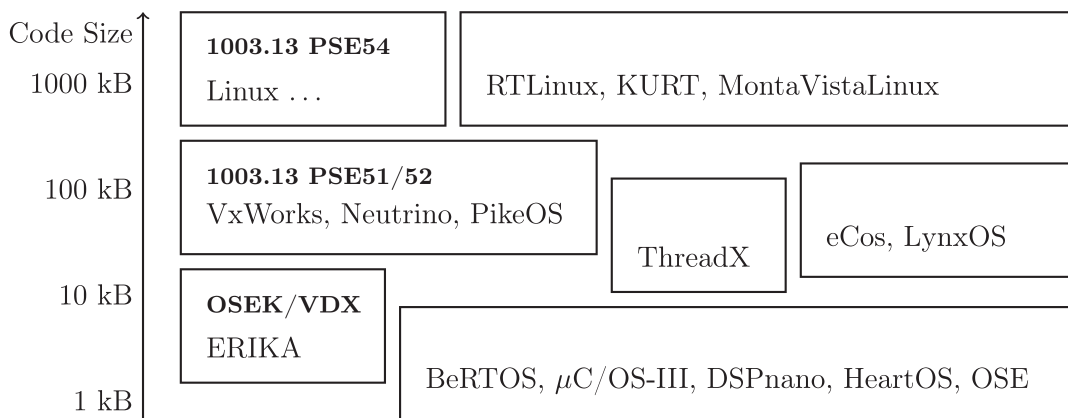

Figure 4.1.: Typical memory footprints of real-time operating systems

The ERIKA operating system was developed to support multicore devices for the automotive markets. ERIKA provides a minimal footprint real-time kernel and supports various code generation tools. Only fixed priority scheduling is implemented. ERIKA is a minimal system with the aim to do mostly I/O processing in a highly embedded environment. The characteristics of the system are not discussed further here.

Real-time operating systems like Deos, Integrity-178B, PikeOS and LynxOS are designed to be DO-178B certified for safety-critical applications. These systems are commonly used in avionics applications. The kernel sources are not available for any examinations[1].

Windows CE is optimized for having a small memory footprint. It is mainly used for devices that do not allow for end-user extensions and can be stored into ROM for example. Some parts of the source code are available to customers.

The systems discussed in the following sections are of greater importance. The chapter concludes with a detailed examination of some selected real-time operating systems.

[1]Excluding LynxOS, the source code is available in part with appropriate license.

Table 4.2 provides a comparison of different hardware characteristics of these systems.

Name Real-time operating system name

Hardware Support for hardware busses

Networking Support for network protocols

File systems Support for file systems (RTFS stands for real-time file system)

In addition to hardware characteristics Table 4.3 shows some technical properties.

Name Real-time operating system name

Development hosts Development platform

Components Support for floating point operations, math library, memory management

Standards/API Standards

4.1. Atomthreads

Due to its extensive documentation and easy readable code Atomthreads provides a good platform for learning RTOS internals. The entire operating system fits into a few source files. No file system, IP stack or device drivers are included. All of the architecture-dependent aspects are encapsulated into what are called CPU architecture ports. Only 2 architectures are supported by default (AVR, STM8).

Atomthreads is released under the open source BSD license and is free to use for commercial or educational purposes without restriction. It was designed and written by Kelvin Lawson in 2010.

4.1.1. Structure of the Kernel

The Atomthreads kernel consists of 6 modules implemented as libraries. The starting routine of the system is part of the CPU architecture port. This routine initializes the operating system by using the libraries.

Kernel Core kernel functionality such as managing the queue of ready threads, creating threads and context-switch decisions.

Architecture depending parts Entry point of the system and architecture dependent parts such as interrupt routines.

Mutex A mutual exclusion library.

Semaphore A counting semaphore library.

Name	Development hosts	Components			Standards/API
		FP	ML	MMU	
MontaVistaLinux	Linux	yes	yes	yes	ANSI and POSIX
eCos	Windows, Linux	yes	yes	yes	ANSI C-89, POSIX.1a, 1b, 1c, 1d, μITRON 3.02
KURT	Linux	yes	yes	yes	
Neutrino	Windows, Linux, Solaris, self-hosted	yes	yes	yes	POSIX.1a, 1b, 1c, 1d, PSE52
HLRT/RT_PREEMPT	Linux	yes	yes	yes	ANSI C-89
RTLinux	Linux	yes	yes	yes	ANSI C-89, POSIX.1a, 1b, 1c, 1d, PSE51
ThreadX	Windows, Linux	yes	yes	no	ANSI C-89, DO-178B, IEC-61508, ...
VxWorks	Windows, Linux, Solaris	yes	yes	yes	ANSI C-89, DO-178B, IEC-61508, POSIX.1a, 1b, 1c, 1d, PSE52

Table 4.2.: Real-time operating systems hardware characteristics

Name	Hardware							Networking			File systems			
	IDE	SATA	SCSI	USB	PCI	PCIe	CAN	IP	TCP	UDP	FAT	NTFS	EXT	RTFS
MontaVistaLinux	yes		yes	yes	yes		yes	yes	yes	yes	yes	no	yes	no
eCos	yes	no	no	yes	yes	no	yes	yes	yes	yes	yes	no	no	no
KURT	yes	no	yes	no	yes	no	no	yes	yes	yes	yes	no	yes	no
Neutrino	yes	yes	yes	yes	yes	no	no	yes	yes	yes	yes	no	yes	yes
HLRT/RT_PREEMPT	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no
RTLinux	yes	no	yes	no	yes	no	yes	yes	yes	yes	yes	no	yes	no
ThreadX	yes	no	no	yes	no	no	no	yes	yes	yes	yes	no	no	no
VxWorks	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	no	yes

Table 4.3.: Real-time operating systems technical characteristics

Queue A queue and message-passing library.

Timer Kernel system clock functionality and timer functionality for kernel and application code.

4.1.2. CPU Architecture Ports

A port to a CPU architecture can comprise just one or two modules which provide the architecture specific functionality, such as the context switch routine which saves and restores processor registers on a thread switch. The kernel port is split into a few modules (files):

`atomport.c` Those functions which can be written in C.

`atomport-asm.s` Main register save/restore assembler routines.

`atomuser.h` Port-specific header required by the kernel for each port.

A context switch is performed by the simple strategy below. After the context switch the return address on the stack will be the new thread's return address.

- Save the CPU registers to the stack.
- Save the final stack pointer to the TCB (thread control block).
- Get the new thread's stack pointer off the TCB.
- Restore the CPU registers from the stack.

4.2. eCos

Since 2004, eCos (embedded configurable operating system) is distributed under the GPL license with an exception which permits proprietary application code. The non-free version of the eCos real-time operating system, eCosPro, is a commercial distribution which incorporates proprietary software components and with additional features that are not released as free software. eCos is designed to be portable to a wide range of target architectures. It has a small memory footprint and is used in embedded systems. The kernel is open source and only free available open source GNU development tools are used to build an image. Only the FAT file system and some file systems for flash memory devices are supported by eCos.

eCos was developed for the embedded market. The compiled system consists only of one binary image including all applications statically. eCos can not be compared with systems such VxWorks or QNX Neutrino. In such systems it is possible to add and remove applications at runtime. Moreover, only a very small selection of modern hardware for off-the-shelf multicore architectures is used by eCos. For example, no SATA, PCIe or even multicore is supported.

4.2.1. Design

The eCos core consists of a suite of functions needed in many embedded applications.

Device drivers Include standard serial, Ethernet, USB and others. CAN bus support is also included in this section.

Kernel Like many real-time operating systems on the embedded context, the eCos kernel was designed to satisfy the following objectives:

- Low interrupt latency
- Low task switching latency
- Small memory footprint
- Deterministic behavior

eCos implements a classic multi-threaded architecture with a set of synchronization primitives. It is intended for embedded systems and applications which need only one process with multiple threads.

ANSI C-89 and math library Provide standard compatibility with function calls.

Hardware abstraction layer Provides a software layer that gives access to hardware. It allows programmers to write device-independent applications by providing standard operating system calls (POSIX) to hardware.

4.3. VxWorks

VxWorks started as a set of enhancements to a simple real-time operating system called VRTX. VxWorks was created to turn the VRTX kernel into a full embedded real-time operating system and development environment. Since 1987, VxWorks is designed for use in deeply embedded systems. Typical fields of application include, for example, machine control, medical equipment and network infrastructure.

Like many other embedded operating systems, the development of applications for VxWorks takes place on a more powerful host computer. The integrated development environment for software in VxWorks is called Tornado. It consits of the operating system itself, development tools as Wind River Compiler and Wind River GNU Compiler, graphical user interface (based on Eclipse) which establishes host-target communication, and a VxWorks simulator called VxSim. Supported development platforms are Linux, Windows and Solaris.

VxWorks supports the POSIX specification and basic system calls including the pthreads extensions. At this time only the x86 64 bit architecture is supported. In addition to the FAT file system VxWorks provides its own file system and some file systems for flash memory devices.

The kernel source is not supplied within a VxWorks license.

4.3.1. Protection Domains Architecture

In VxWorks user-space applications are isolated from other user-space applications as well as the kernel via memory protection mechanisms. Comparable with QNX Neutrino VxWorks also uses different separated and protected memory areas. A separated memory area in VxWorks is called protection domain. Figure 4.2 illustrates the protection domains architecture of VxWorks. The microkernel provides multitasking, interrupt support and scheduling. The intertask communications mechanisms supported by VxWorks include shared memory, message queues, semaphores, events and pipes, sockets and remote procedure calls, and signals.

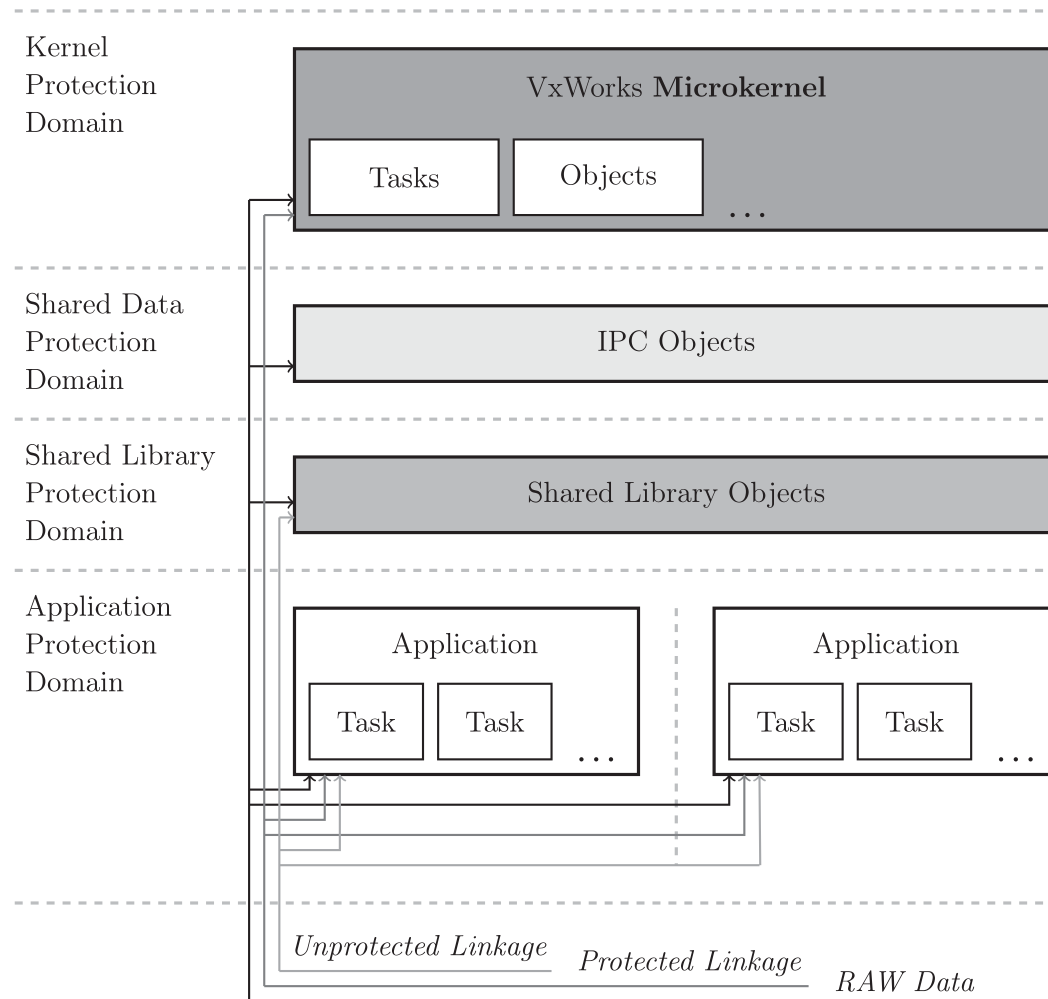

Figure 4.2.: VxWorks protection domains architecture

4.4. μC/OS-III

The Micro-Controller Operating Systems Version 3 is a priority-based real-time multitasking operating system kernel for microprocessors, written mainly in the C

programming language. It was introduced in the year 2009 by Micrium Embedded Software.

4.4.1. Design

μC/OS-III can manage up to 64 tasks. The four highest priority tasks and the four lowest priority tasks are reserved for system services. Each task is an infinite loop and can be in one of the states described in Section 3.1.1 and have a unique priority level assigned. The task priority level also serves as the task identifier.

4.5. MontaVistaLinux

The MontaVistaLinux (formerly known as Hard Hat Linux) operating system is tailored to the needs of embedded software developers. MontaVista sells subscriptions, which consist of software, documentation, and technical support. An integrated development environment is included known as DevRocket. The environment is a set of Eclipse plug-ins for facilitating application and system development with MontaVistaLinux. DevRocket runs on Linux, Solaris and Windows.

The software includes a Linux kernel and toolchain aimed at a specific hardware configuration. The distribution is available in three editions, each aimed at different market segments.

MontaVista Carrier Grade Edition The commercial CGA Linux development platform ensures long-term support and high availability.

MontaVista Professional Edition The commercial professional edition has some benefits which are missing in the other versions including integration with open source tools for a particular hardware architecture, and support.

MontaVista Mobilinux The Mobilinux version is a highly embedded operating system mainly targeted at smartphones.

4.6. ThreadX

ThreadX was designed specifically for deeply embedded applications. The system is implemented as a C library. Only the features used by the application are brought into the final image. The name ThreadX is derived from the fact that threads are used as the executable modules. ThreadX is widely used in a variety of consumer electronics (mainly ink and laser printers). The operating system is delivered with complete C and assembly source code.

The operating system ships with its own application interface, currently ThreadX V5 API. Several compatibility kits for ThreadX are available. The POSIX kit defines the basic pthread services.

Developing embedded systems using ThreadX is usually done on a host machine running Linux or Windows. Several commercial development tools are available for ThreadX.

4.6.1. Pico Kernel

The operating system uses a pico kernel design which is another name for micro kernel. The term pico kernel is used to further emphasize its small size. The ThreadX package can be extended by some extra modules. FileX is a FAT compatible file system. Networking support such TCP and UDP is included in the NetX module. The USBX module is required for USB support.

4.7. RTLinux

RTLinux [YB] is an extensions for Linux to achieve real time capability using the RTAI API [Dip]. RTAI was founded by Paolo Mantegazza from the Technical University of Milan. RTLinux was developed by Victor Yodaiken and Michael Barabanov [YB97] at the University of New Mexico in 1996 and was originally sold by FSMLabs. In 2007 the rights on RTLinux were acquired by the embedded Linux specialist Wind River Systems.

Real-time tasks are implemented as kernel tasks. In RTLinux a user space process can communicate with a real-time task via real-time signals. Hardware interrupts and timed signals can be handled at user space signal handlers. It is not possible to call any Linux system calls nor RTLinux services from these handlers.

Because the latest stable version of RTLinux is designed for the Linux kernel version 2.4.29 it lacks some important features of modern computer systems. For instance, support for SATA devices is not provided. Further, new x86 CPU architectures are also not supported. The latest CPU that is known to be functioning is from the Intel Pentium series.

4.7.1. Structure of the Kernel

RTLinux is based on a micro kernel design which provides a real-time executive underneath the original Linux kernel (Section 3.6.1). The micro kernel allows running of real-time tasks at the kernel level, and turns the original Linux into one of these tasks [2]. An (real-time) application can be defined as a collection of (real-time) tasks, and Linux would be the one with the lowest priority. Interrupts are intercepted by RTLinux. The interrupt handling of the Linux kernel is adjusted so that Linux has no control over hardware interrupts. Instead, these interrupts are intercepted by RTLinux and passed to Linux as soft interrupts when they intended for Linux. Interrupts to real-time tasks (eg. timer) will be treated by RTLinux and can not be turned off by Linux.

[2] *Original Linux task* includes the Linux kernel and all Linux processes running above it.

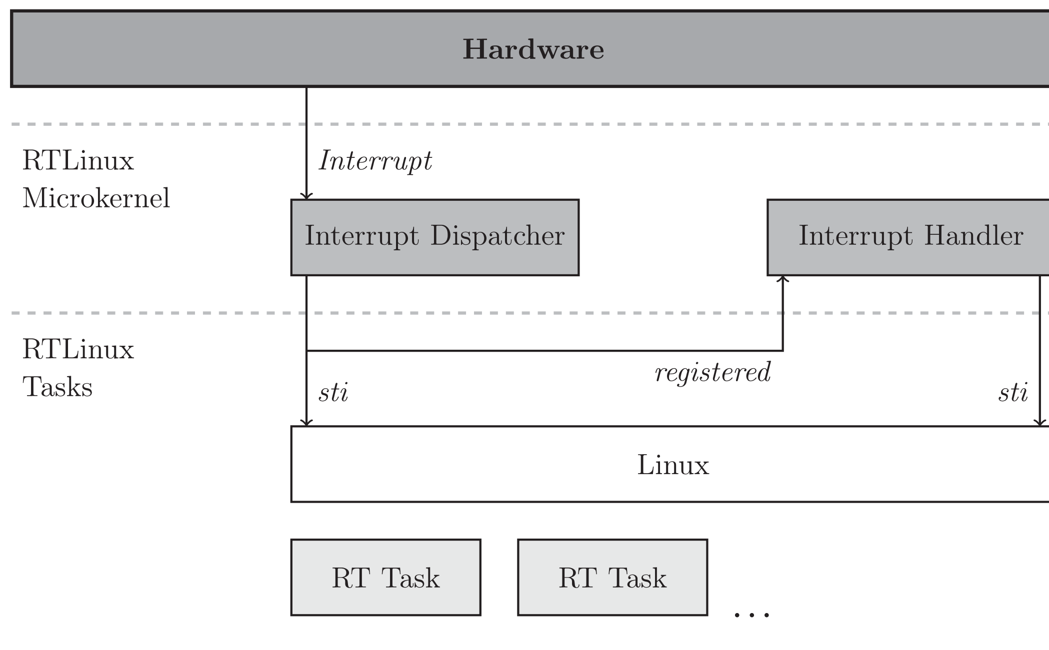

Figure 4.3.: RTLinux kernel design

Real-time tasks in RTLinux run as kernel modules. They can be loaded into the micro kernel from Linux at run time. It is possible to load and unload RTLinux modules without the need of recompiling the kernel or even rebooting the computer. RTLinux offers a bidirectional communication mechanism between real-time tasks and Linux processes. What are referred to as `RT-FIFO`s can be accessed from the RTLinux side by some atomic and non blocking kernel functions. From the Linux processes point of view, `RT-FIFO`s are accessed like traditional UNIX character devices.

RTLinux consists of a set of kernel modules, along with some patches to body of the Linux kernel. The real-time system is loaded as a set of modules into the patched linux kernel.

4.8. QNX, HLRT and RT-Preempt

This thesis discusses three real-time operating systems in detail. We will begin with the RT-Preempt extension, a popular real-time extension for Linux, which is developed and maintained by a small group of developers from the *Linux kernel mailing list* [3]. The second real-time operating system is the HLRT patch, which was originally created at the University of Bremen. The RT-Preempt extension and the HLRT patch pursue completely different strategies in order to enable real-time behavior. However, both improve the Linux kernel in a fundamental way, which makes these extensions highly interesting for this thesis. The third real-time operating system is QNX Neutrino,

[3] `http://www.tux.org/lkml/`

one of the most widely used real-time operating systems. Particularly Neutrino's partitioning strategies and the microkernel design are significant for this thesis.

Detailed discussions about QNX Neutrino (Chapter 8), the HLRT (Chapter 7) and RT-Preempt (Chapter 6) patches are provided in the next chapters.

5

Evaluating (POSIX) Real-Time Operating Systems

Evaluating a real-time operating system in a way that allows a reliable comparison to other operating systems is a difficult task. Depending on the application area (and often the aim of a project) the various attributes of the systems must be weighted differently. For instance, one of the first things that must be done when selecting a real-time operating system for a project is to identify the real-time constraints and categorize them. In some cases an accurate timing with a deviation of some nano seconds is highly important, in other cases the timing behavior of a general-purpose operating system might be sufficient. The methods and strategies described in this chapter do not consider any project related needs for real-time operating systems.[1] The intention of this chapter is to work out a comparable unit for the different technical factors of a real-time operating system. After identification and categorization of these technical factors in Section 5.2 a framework for how to measure dynamic sizes and interpret values in the described categories is introduced in Section 5.4.

The method presented is applied in the following chapters. Each of the detailed investigations of real-time operating systems[2] is deposited with the results of this method.

5.1. Unconsidered Aspects

For a convincing comparison of real-time operating systems it is necessary to mention technical features as well as non-technical aspects like economic reasons or contract relations. It is quite necessary to compare the cost of licenses of different systems when choosing a real-time operating system for a particular project. In addition, software companies offer commercial real-time operating systems often in different models for support, training and consulting. These factors may affect the development time of software for the project under favorable circumstances.

Furthermore, the development of software for an operating system is limited by the availability of tools (e.g. compilers and debuggers). It should also be noted which

[1] Section 5.1 gives an impression of such characteristics. Further information can be found in [CL07].
[2] Chapter 6, Chapter 7, Chapter 8 and Chapter 12

programming languages are available for the predestinated platform. Especially in the embedded systems field in most cases a complete tool chain is offered by the manufacturers, which also includes a (graphical) development environment. This environment may also contain communication channels to interact with the running system on the embedded hardware.

Supported standards (and in general compatibility of a system) may be particularly relevant when porting an existing project to another platform i.e. operating system. A high correlation between different programming interfaces (API) may lead to less effort in adapting existing software.

During the comparison of real-time operating systems and technical evaluation according to the method developed in this chapter, the above-mentioned characteristics are not considered.

5.2. Identify Technical Values

In Chapter 4 some statical properties of real-time operating systems were presented. For an evaluative comparison of different systems, it is necessary to extend the list of above properties by a couple of dynamic aspects. These dynamic properties are visible at system runtime and are strongly influenced by the underlying platform.[3] These aspects can commonly be referred to as performance values:

- Task period accuracy
- Task switch latency
- Task preemption latency
- Task creation latency
- Interrupt latency
- Interrupt dispatch latency
- Interrupt to task latency

Performance benchmarking is the process for determining and comparing the above dynamic properties. Depending on the benchmarking method, various properties are identified. The method used in this work to collect performance metrics derives directly from the Rhealstone variant [KP89]. Section 5.2.1 discusses which values are collected and analyzed. The approach described in the Rhealstone paper requires the discovery of six different reference values which are known as *Rhealstone components*. For benchmarking, these components are measured separately and independently from each other. The following comparison of the measurement values implies that the measurements are all performed under homogenous conditions. This relates primarily to the hardware, which is identical for all measurements. The obtained

[3] A description of the used hardware setup can be found in Section 5.3.1.

values can be compared within a component among the various systems. Furthermore, the Rhealstone method describes how the empirical results can be combined to a representative figure of merit (*Rhealstone per time unit*).

For this thesis, not all of the components described in the Rhealstone paper are important for the intended comparison. The *semaphore-shuffle time* and the *deadlock-break time* are not considered. As can be seen in Chapter 10 and Chapter 11 no functionality to the Linux kernel has been extended or altered for the environments concerned. The semaphore-shuffle time is taken into account indirectly in some test introduced in Section 5.4.3. In modern operating systems (multicore mode) the deadlock-break time is not applicable in the way it is introduced in the Rhealstone paper. Some new components (see also [HGCD00]) have been added to the Rhealstone method which will be discussed in the following sections.

5.2.1. Benchmark Methodology

As already described in the introduction, identifying the considered reference values is based on the Rhealstone method. The technical realization of the measurements is explained in Section 5.4. The performance values which are to be determined are listed below:

Task period accuracy The accuracy of the period defined for a task. Depending on the operating system the appropriate technique of the system for the realization of periodic tasks must be selected.

Task switch latency The time required for a task switch on a CPU (Figure 5.1). All for the measurement relevant tasks are executed with the same priority and are scheduled according to the FIFO policy (Section 3.2.3.1). A task initiates a task switch by calling the `yield()` function.

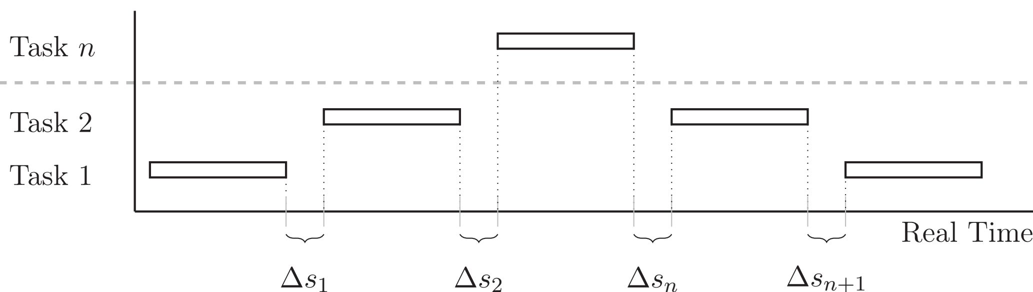

Figure 5.1.: Task switch latency

The task switch latency is measured in four stages. In each stage the number of active tasks is increased (2, 16, 128, 512). Therewith, effects of the cache on the scheduling can be made visible.

Task preemption latency Similar to the task switch latency measurement, the time that is needed for a task switch is measured with the task preemption latency

measurement. However, a running task is preempted by a higher prioritized task. The task switch is performed by putting a higher prioritized task to the ready state. As will be seen in Section 5.4.3, this measurement requires a certain degree of synchronization.

Task creation latency The time needed for generating a new task is measured with the task creation latency. Depending on the operating system the newly created task must have a higher priority than the task which is currently running. In Linux a newly created task inherits some properties of the parent process such as the task priority level. However, the created tasks are placed at the head of the FIFO queue, whereby the executed task is interrupted by the new task.

Interrupt latency The time which elapses from the occurrence (or triggering) of an interrupt to execution of the handler code (interrupt-service routine, ISR) (Figure 5.2) is measured with the interrupt latency.

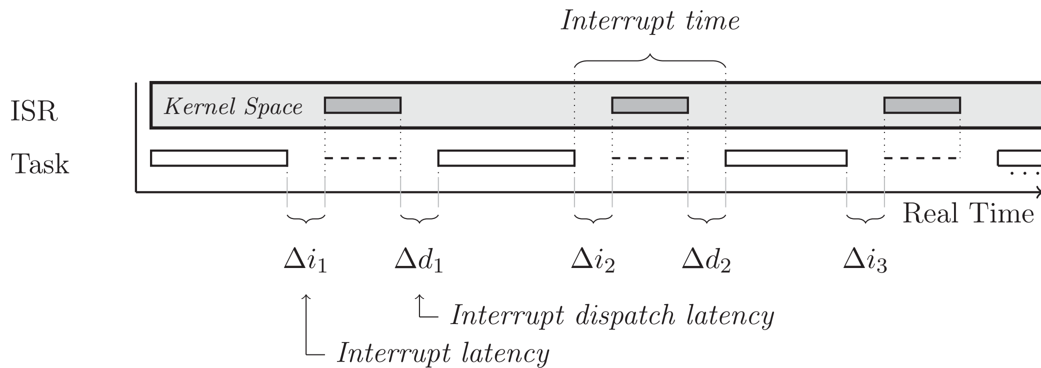

Figure 5.2.: Interrupt times

Interrupt dispatch latency The time required by the execution of an interrupt handler to switch back to the interrupted task (Figure 5.2) is measured with the interrupt dispatch latency.

Interrupt to task latency The time that elapses from the occurrence of an interrupt to execution of a second level handler interrupt (Figure 5.3). The second level interrupt handler (SLIH) is running in task context. Thus, the measuring includes at least one task switch. Similar to the task period accuracy benchmark test the appropriate technique of the system for the realization of second level interrupt handlers must be selected.

A series of measurements consists of the independent multiple repetition n of a measurement $\{a_1, \ldots, a_n\}$. The result of a measurement series is represented by two values:

Average or arithmetic mean of all values in one series

$$A = \frac{1}{n} \sum_{i=1}^{n} a_i$$

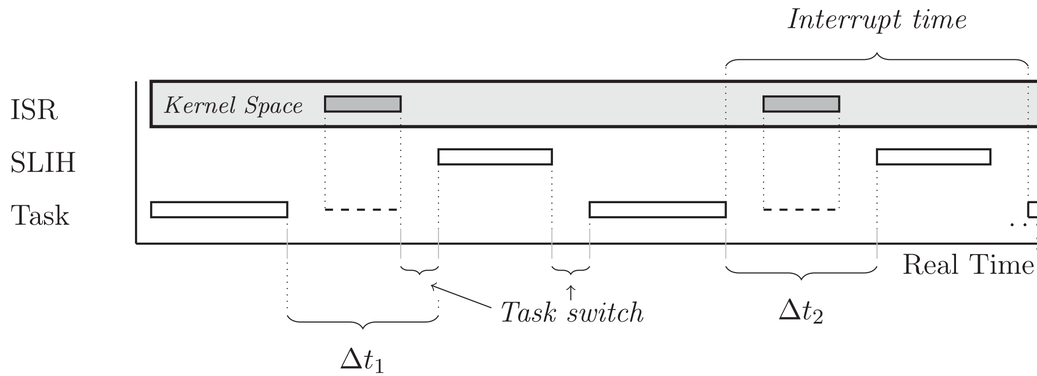

Figure 5.3.: Interrupt to task latency

Standard deviation from the average

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(a_i - A)^2}$$

5.3. Case Scenarios for Real-Time Systems

The determination of the reference values introduced above has to take place under the very same conditions for all systems that are going to be considered. For each system, a set of tests is implemented and is executed on identical hardware. Section 5.3.1 discusses the underlying platform for the tests performed in later chapters.

The tests are performed in various scenarios. In addition to the unloaded application of the tests[4] two other scenarios are considered:

Task schedule utilization Each CPU in the system is utilized by various computational tasks. This increases the number of active processes and affects the load balancing behavior of the scheduler. The utilization is achieved by starting four (for each CPU in the system) computationally intensive processes.

I/O utilization The I/O utilization is achieved by starting two (for each CPU in the system) processes only performing operations on the file system and the underlying hardware. This increases the number of active processes in the system and affects the quantity of interrupt occurrences.

These scenarios are intended to simulate a realistic environment during test execution. Especially for real-time operating systems, it is necessary to know the effects of the environment on the running system. The measurement of a reference value can only be meaningfully assessed by taking account of the described factors.

For the simulation of these scenarios, the *stress*[5] application is used. The program is a deliberately simple workload generator for POSIX systems. It imposes a configurable

[4] A setup as neutral as possible where only services required by the system are active.
[5] `http://freecode.com/projects/stress`

amount of CPU, memory, I/O, and disk stress on the system. Because of its simplicity, the application is well suited to be compiled and started on all systems to be tested without adjustments. A separate process is started for every parameter passed to the application. Each process fulfills only the desired jop until the main application is terminated. By distributing the load across several processes a multi-processor system can be evenly loaded. The scenarios are realized with the stress application as follows:

- `./stress --vm 8 --cpu 8`
  This spawns 8 processes spinning on the `sqrt()` function call and 8 processes spinning on `malloc()` and `free()`. As described in Section 5.3.1 the used hardware has 4 CPU cores. On each CPU core at least 2 processes causing load are running and affect the test.
- `./stress --hdd 4 --io 4`
  This spawns 4 processes spinning on `sync()` and 4 processes spinning on `write()` and `unlink()`. As a consequence, each CPU in the system will receive a lot of I/O work.

Tests are executed in three different scenarios. These scenarios are equal on each system.

5.3.1. Development Board

All benchmark tests run on the same x86_64 platform. The different operating systems are installed on a Precision T3500 Workstation (Table 5.1). The CPU used for benchmarking comprises 4 separate cores. The hyper-threading technology is not used and is therefore disabled (Section 2.4.2).

CPU	Intel Xeon W3530			
	Clock rate	2.8 GHz	Family	6
	CPU cores	4	Model	26
	L1 Cache	8192 KB	Stepping	5
	instruction set	64 bit (SSE4.2)		
Board	Dell XPDFK Mainboard			
	Chipset	Intel X58		
	Memory	3 x 1 GB DDR3 1066 MHz		
Additional				
	Network	Broadcom 5754 Gigabit Ethernet-Controller		
	Measurement	Meilhaus ANT8 Logic Analyser		

Table 5.1.: Development board specification

5.4. Benchmark Test Framework

The structure and the process for the individual benchmark tests will be discussed in the following sections. The implementations of the tests are explained only to a limited extend since the test implementations are strongly dependent upon the used systems. The complete framework including the implementations of the tests can be accessed at [Rad14]. The underlying framework is identical in all tests. In particular the technique used for determining measurement values (Section 5.4.1) and specific configuration requirements of the tests are outsourced to header-files. The iteration of the test procedures is set by a constant `ITERATIONS`. The evaluation of the operating systems is executed with a fixed iteration size of 250. In addition to the iteration constant the benchmark tests are influenced by the following parameters:

`ITERATIONS` (`benchmark.h`) Number of test cycles

`BENCHMARK_CPU` (`benchmark.h`) Starting CPU for the benchmark test

`RT_PRIO` (`benchmark.h`) Initial priority level for the benchmark test (depending on the test it is necessary to create processes in the range of `RT_PRIO` ± 2)

`PARPORT_ADDR`, `PARPORT_IRQ` (`benchmark_linux.h`, `benchmark_qnx.h`) Parameters for the parallel port interface (Section 5.4.3.1 and Section 5.4.3.4)

`PARPORT_DEV` (`benchmark_linux.h`) The device-file of the parallel port on Linux based systems (Section 5.4.3.1)

`PARTITION_NAME`, `BUDGET_PERCENT` (`benchmark_qnx.h`) Name and budget for partitioning on a QNX system

During execution the benchmark test and all created processes are bound to a certain CPU (`BENCHMARK_CPU`). The reason for this is explained in Section 5.4.1. During test initialisation it is necessary to arrange an affinity to this CPU by using the interface provided by the used operating system. Additionally a fixed clock speed for the `BENCHMARK_CPU` is required. The clock speed must not vary during test execution. A deviation from the defined clock speed or migrating to another CPU will cause distorted results.

5.4.1. Measurement Details

The measured values, which are determined during the benchmark test execution are relative temporal information unrelated to the actual system time. An absolute time value which is valid after test termination is not required for the benchmark tests. It is sufficient to measure the time interval between two events unrelated to the system time. Thus, an overhead can be saved that would otherwise be required for precise determination of the time points.

The *time stamp counter* (TSC) register (Section 2.4.4) is used to determine time values inside a test. The register is accessible without the involvement of the operating

system. Due to its precision, the TSC register is well suited for the time measurements performed by a test. Since each CPU has its own TSC register which is not synchronized with other CPUs[6], the measured values from one CPU can not be compared with measured values from other CPUs. In addition to the overhead of the operating system for task migration this is the main reason why the test execution is bound with all active components to a certain CPU. The frequency of the TSC is bound to the timing of the CPU clock. The interpretation of the values of the TSC register change if the CPU clock speed is scaled. In order to obtain an accurate and comparable difference between two TSC register values the CPU clock speed must not be modified. For compatibility reasons, only the lower (32 bit) part of the TSC register is evaluated. Overruns of such 32 bit value are easy to recognize and can easily be fixed (Listing 5.1) after test execution during processing of measurement results.

period_x_ms.c

```
...
if (stop > start)
    step = stop - start;
else
    step = (((uint32_t)-1) - start) + stop;
...
```

Listing 5.1: Test result overrun

The values of a TSC based time tracking are measured in cycles. The framework previously introduced in this chapter provides a macro `rdtsc_32()` which is able to determine the current value of the register (Listing 5.2). `rdtscp` and `cpuid` are serializing calls. They prevent the CPU from reordering instructions around these calls. [Pao10] describes how code execution should be measured on x86 architecures:

1. Call `cpuid()` and `rdtsc_32()` to get cycle count when starting the measurement.
2. Call `rdtscp_32()` and `cpuid()` to get the counter when measurement finishes.

benchmark.h

```
...
#define rdtsc_32(tsc)                                                  \
    __asm__ __volatile__("rdtscp" : "=a" (tsc) : : "edx")

#define rdtscp_32(tsc)                                                 \
    __asm__ __volatile__("rdtscp" : "=a" (tsc) : : "edx")

#define cpuid()                                                        \
    __asm__ __volatile__("cpuid" ::: "rax", "rbx", "rcx", "rdx")
...
```

Listing 5.2: Read time stamp counter

During test execution, values are stored in a table in memory, the structure of which depends on the benchmark test. No data is stored in files or transferred to a console while the test is in progress. This would include parts of the operating system that

[6]In fact, some operating systems synchronize the TSC register of all CPUs in the system. It cannot be assumed that all operating systems perform a synchronization with sufficient precision.

would otherwise not be relevant for the test execution and the measurement would be distorted unnecessarily. As already mentioned, two values of the TSC register are always necessary for a measurement. The table that receives the measurement data needs to have at least two fields for 32 bit values per measurement step. If several tasks are involved in a measurement, the table of results is placed in a memory area where it can be accessed by different processes (IPC shared memory). It is important to ensure that no access to this memory area is performed within a measuring step. Depending on the operating system, dereferencing a memory address within a shared memory segment involves mechanisms of the operating system of which the overhead cannot be estimated. The measured values are first administered locally in the relevant process memory and then later are transferred to the shared memory segment. This is done either after completion of all measurement steps or between two measurement steps if no impact on the following measurement step can occur. The management of a table inside a shared memory segment may require synchronized access to this area.

POSIX operating systems provide a mechanism for excluding memory pages of a process from being swapped out (`mlockall()`). For the tasks involved in a benchmark test, this technique is used. In addition, all used memory areas located in the stack segment of the process are initialized in order to achieve a stack pre-fault. This causes all memory areas to be available at test start and be permanently located in the memory.

In addition to the measurements based on the TSC register, a second series of measurements is performed for the benchmark tests to measure the accuracy of periodic tasks. With the support of a logic analyzer, a pulse is measured on external hardware that should correspond exactly to the period of the task. Instead of determining the TSC register value the test stimulates a trigger. The pulse is sent through the parallel port. In each task period a data pin of the parallel port is alternated by the test. The implementation of this test is used to verify the first measurement series values on a system which is not directly involved. From the results of the logic analyzer measurements it can be concluded that the accuracy and quality of the TSC measurements are sufficient.

5.4.2. Operating System Overhead

To some extent, the operating system overhead constitutes exactly what is to be measured by the presented benchmark tests. However, the test itself during test execution is also a component of the system and affects the overhead of the operating system in some areas. It is not always possible to substantiate the overhead caused by the test from the operating system overhead. For example, the method described above for measuring the period of a task by using a logic analyzer involves various operating system components that can delay the transmission of the pulse through the parallel port. In Linux based operating systems, a device file is used for stimulating the parallel port. Thus the file system layer of the kernel is involved into test execution. This overhead, however, is applicable for all Linux based systems and therefore is of

no consequence. In QNX, a similar technique is used for transmitting a pulse through the parallel port so that the overhead is also produced in a comparable manner.

In the following sections overhead that can occur is described together with the test procedure explanation if necessary. Delays that occur between the execution of two measuring steps are not relevant for the measurement and can therefore be ignored.

5.4.3. Test Design

The following section will show how the reference values as identified in Section 5.2.1 are determined. Figure 5.4 shows the principal sequence of a benchmark test. The step *OS setup* is determined by the conditions of the running operating system. The necessary tasks here will be explained later in the related chapters. The initialization of the hardware is particularly important for the interrupt-related tests. This step prepares the used interface for generating interrupts from inside the test procedure. The additional hardware for the task period accuracy benchmark tests is also initialized in this step. The individual steps required by *init HW* are explained later in this chapter and will be discussed again in the chapters related to operating systems. The steps *shutdown HW* and *OS postprocessing* form the counterpart to the initialization step. After completion of the tests reserved hardware and operating system resources are properly released in this step.

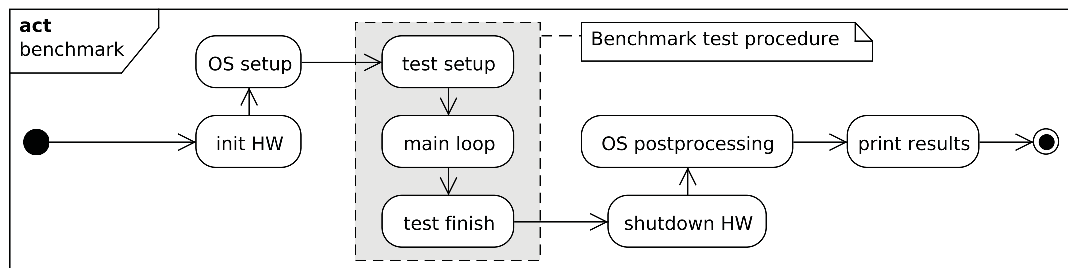

Figure 5.4.: Benchmark test behaviour

Once the hardware and the operating system are prepared for the subsequent test, an individual initialization phase (*test setup*) is performed for each test. This is part of the actual benchmark test and prepares the used memory instances depending on the test procedure. Depending on the number of tasks involved in the test execution, the process stack, heap or a more complex inter process communication structure is used for storing the measured data. The memory organization is illustrated for each test in the following sections. It may be necessary to release previously initialized memory structures after test execution (*test finish*). The order of the individual steps shown in Figure 5.4 does not apply equally to all tested operating systems. It may be necessary to *print results* before releasing previously initialized memory structures. In addition to the memory organization, the creation of other tasks that are used for the test execution is realised within the test setup phase.

Each benchmark test involves a cyclical component in which the measurement takes place (*main loop*). The test loop and parts of the setup and finishing parts are

explained in the following sections. Workflows are represented using UML sequence diagrams.

No output of test results (e.g. to a terminal) is made during test execution. The results are printed (*print results*) to *stdout* after all data has been received.

5.4.3.1. Task Period Accuracy

Figure 5.5 illustrates the benchmark test for measuring the task period accuracy. How a periodic task is initialized depends on the operating system which is used. The process described here outlines the initialization of a periodic tasks after storage preparation. The aspects of individual operating systems for generating periodic tasks are referred to in the appropriate chapters.

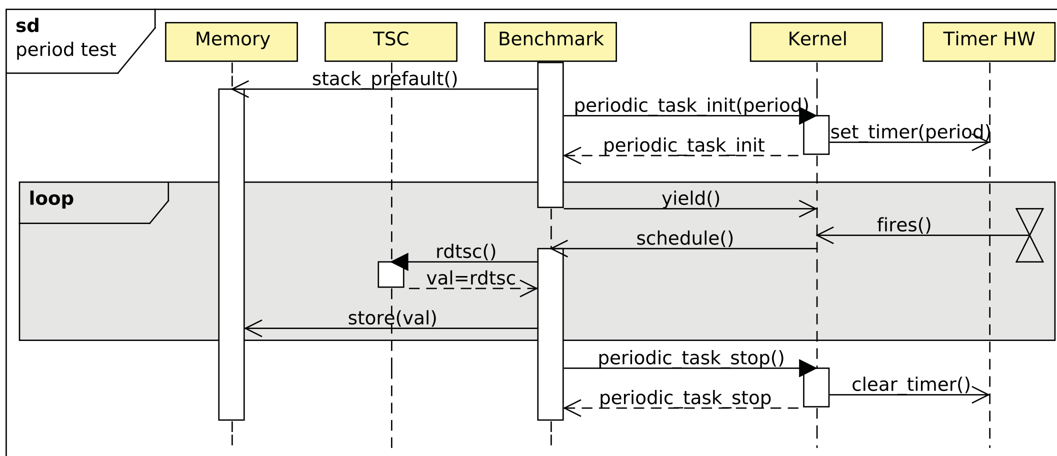

Figure 5.5.: Periodic task benchmark test

The call to `yield()` indicates the end of a period to the operating system. The function is replaced by a version that is required by the operating system to signal the end of a period if necessary. The periodic task is suspended until the start of the next period. The occurrence of the timer signal (`fires()`) constitutes the beginning of the next period. The previously suspended task continues and immediately determines the current value of the TSC register.

In this measurement, the accuracy of the clock source used by the operating system and the latency for continuing the task (`schedule()`) is weighted. The continuing task must have the highest priority so that the CPU is immediately given to that task. A measurement result is stored in the memory structure described above. A result complies to the difference between two contiguous readings. The expectation for a result is exactly the task period time. Thus, for n desired results the measuring is repeated $n + 1$ times.

The test is performed with different period time values: $500\mu s$, $10ms$, $100ms$ and $1sec$. The values selected for the period time do not reflect the limits of possible values. Limits for task period times vary according to the operating system. However, it was ensured that at least one value is below the global timer frequency intended by

the operating system (if available). For Linux-based systems, this value describes the frequency of timer interrupt and is usually defined as $10ms$.

The macro `TRIGGER_PARPORT` (which must be defined at compile time of the test) configures the benchmark program so that the previously described logic analyzer is addressed.

5.4.3.2. Task Change Times

The sequence shown in Figure 5.6 illustrates the benchmark tests for preemption latency and task switch latency measurement. Since both tests are similar in many aspects they are presented here together in one diagram. Differences in both tests are shown inside the *alt* block.

At least two new processes are created before the test main loop starts. Scheduling for the new processes is managed under the FIFO policy. As described in Section 5.2.1 the number of processes that are created varies for later test cases. The created processes start with a priority level lower than the initial benchmark process (thus the initial task is not interrupted). The measurements in the test main loop are based on interactions between the new created processes. To complete the initialization phase, the priority of the initial benchmark process is lowered. Thus, the processes previously generated are executed according to the priority and order of the operating system and preempt the initial process. Each task starts with its own initialization phase and then blocks by waiting for an event (`wait_for_signal()`). Since all higher prioritized tasks are *blocking* now, the initial benchmark task comes back to life and initiates the start of the main loop by sending an event (`send_signal()`). The event preempts the initial task immediately. After test completion the additional created processes terminate and the initial process continues with the *finish* phase.

The preemption latency measurement takes place between two tasks with different priorities (*high* and *low*). However, both tasks need to have a priority level higher than other active tasks in the system (apart from the initial benchmark task during initialization). The *low* task can only be preempted by the *high* task. If the *high* task gives up the CPU, the *low* task is scheduled instantly. Inside the test main loop the *high* task is blocked by waiting for a signal (`wait_for_signal()`) which can only be sent by the *low* task. The current TSC register value is stored by the *low* task before the signal is sent. After the event is sent the *high* task preempts the *low* task and also stores the current TSC register value. Thus, the time between the arrival of an event on which a higher priority task is waiting and the continuation of the previously suspended task is measured. Depending on the operating system, the synchronization events can be implemented in different ways. If an operating system offers a special method for process synchronization both variants will be implemented. At least one version of the preemption latency measurement test is implemented for each operating system (using POSIX signals).

In order to measure the task switch latency two (or more) processes are needed, each with the same priority level. Similar to the preemption latency tests, the created processes for the task switch latency measurement need to have a priority level higher

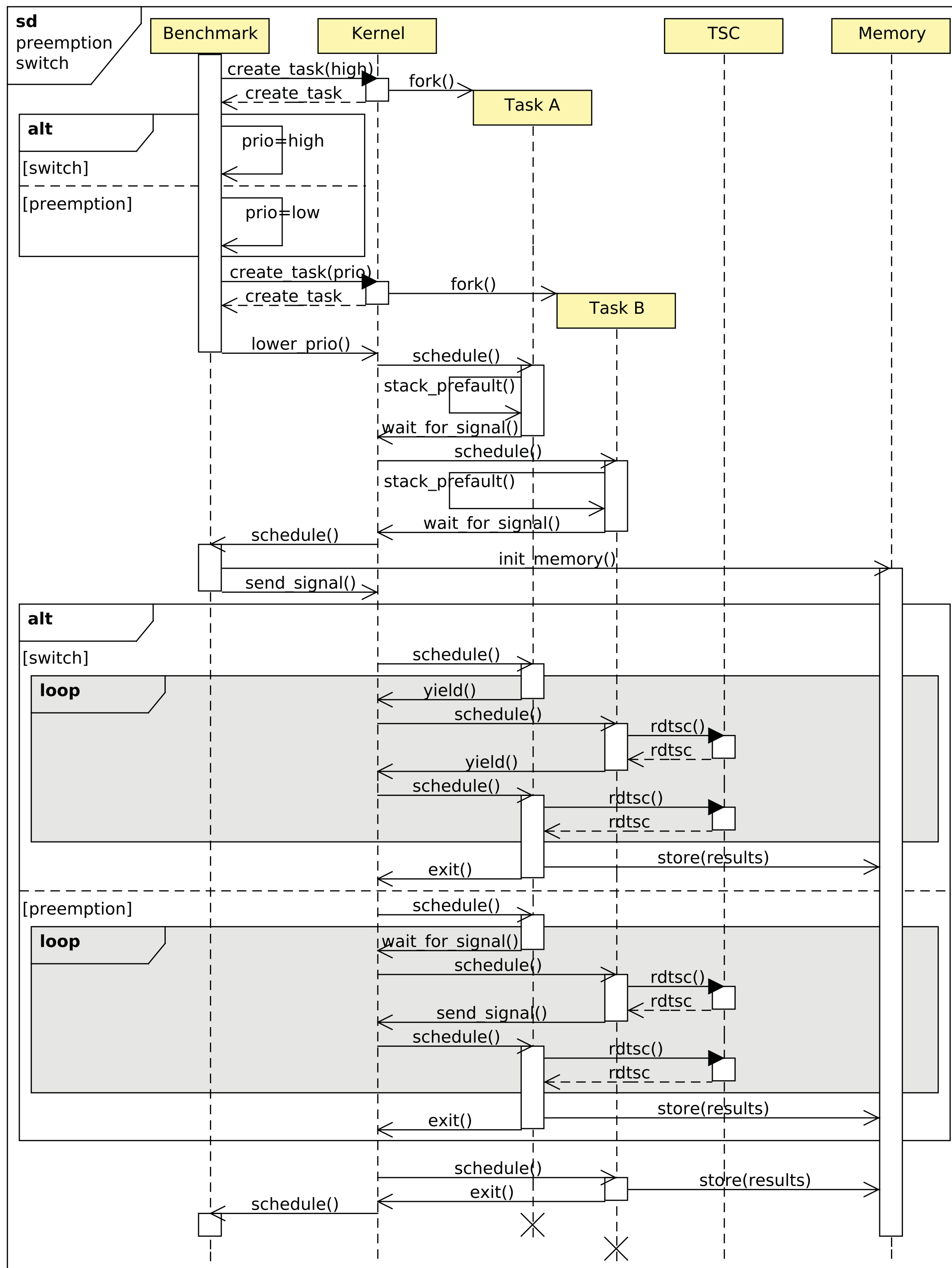

Figure 5.6.: Task switch/preemption benchmark test

than other active tasks in the system (apart from the initial benchmark task during initialization). The additional tasks for this test are absolutely identical in behavior. In each test step, the current value of the TSC register is stored before a task switch is initiated by calling `yield()`. From the measurement values the time between the preemption of one task and the continuation of another one can be calculated.

For each task involved in the test the measurement data is collected locally. After test completion (main loop exit) the data is transferred to a shared memory area. Each test steps contains two TSC register values: start and end. The process Id is additionally stored for each value.

5.4.3.3. Task Creation Time

For the determination of the task creation time a new process is created within the main loop of the test for each test step. The newly created process has a higher priority level than the main benchmark process. Thus, the main task is preempted immediately by the new task.[7] The current value of the TSC register is determined immediately before a new process is created (`fork()`). The values are stored locally. The new task determines the current value of the TSC register again and stores the value in a shared memory segment. After the additional task is terminated the main benchmark task transfers the previously stored TSC register value to the shared memory segment.

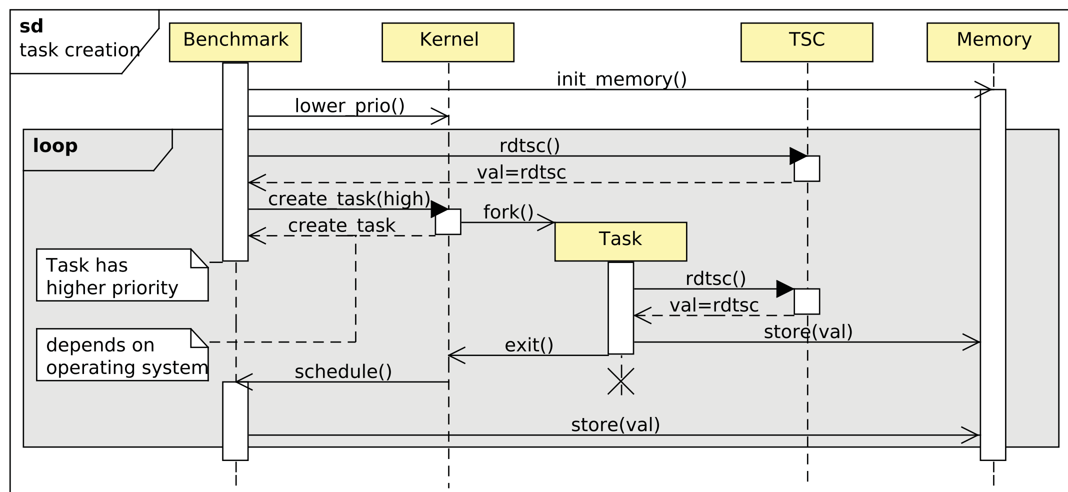

Figure 5.7.: Task creation benchmark test

Requesting the TSC register values takes place as early as possible in the newly created task. A small overhead follows from the `if-fork()` construction (Listing 5.3) which is common for POSIX operating systems. However, this overhead applies to all the operating systems analysed in later chapters and therefore can be ignored.

fork_higher_prio.c

```
...
rdtsc_32(tsc);

if (0 == fork()) {
    rdtsc_32(tsc);
...
```

Listing 5.3: Fork new process in task creation benchmark test

[7] For Linux based systems it is sufficient if the new task has the same priority like the main benchmark task (compare Section 5.2.1).

5.4.3.4. Interrupt Times

The basic structure for a benchmark test to determine the interrupt latency time, interrupt dispatch latency time and interrupt to task latency time is shown in Figure 5.8. The figure illustrates all components involved in test execution. The individual specifications of the above mentioned test characteristics are explained below.

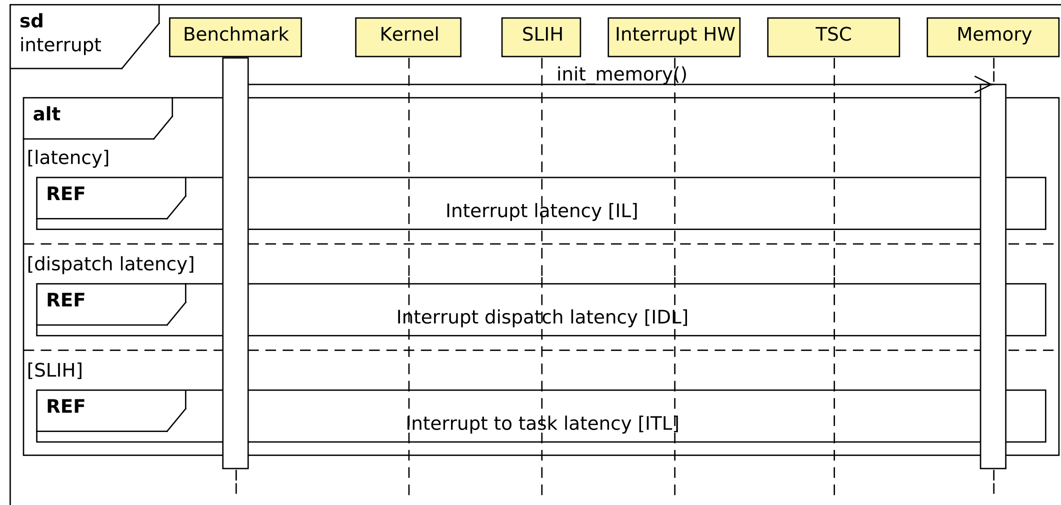

Figure 5.8.: Interrupt benchmark test

The parallel port is used for triggering interrupts. In a particular configuration the port allows the test to provoke interrupts. The specification for the parallel port describes that bit 4 of the control register (port 3) turns interrupts on. The parallel port creates an interrupt each time the electrical signal at pin 10 (ACK bit) changes from low to high. To actually generate interrupts one of the data bits is connected with the ACK bit. The test then stimulates this data bit and thus triggers an interrupt. For the interrupt latency and interrupt dispatch latency measurements it is necessary to implement an interrupt handler. The handler is designed according to the conditions of the used operating system and is connected with the parallel port interrupt (usually number 7).

The interrupt latency benchmark test is shown in Figure 5.9. After storing the current TSC register value in a local memory area the parallel port (*Interrupt HW*) is invoked to cause an interrupt (`trigger()`). The interrupt handler then determines the current time value from the TSC register. As well as the interrupt latency, the measurement includes the overhead needed to write data to the parallel port.

Similar to the interrupt latency benchmark test, an interrupt handler needs to be implemented for the interrupt dispatch latency measurement. The process of the interrupt dispatch test is shown in Figure 5.10. Like before interrupts are generated by calling the `trigger()` function. However, the TSC register value determined from the interrupt handler represents the start value for the measurement. After returning to the interrupted task the second TSC register value is ascertained. Thus, the measurement includes the time between exiting the interrupt handler and continuing

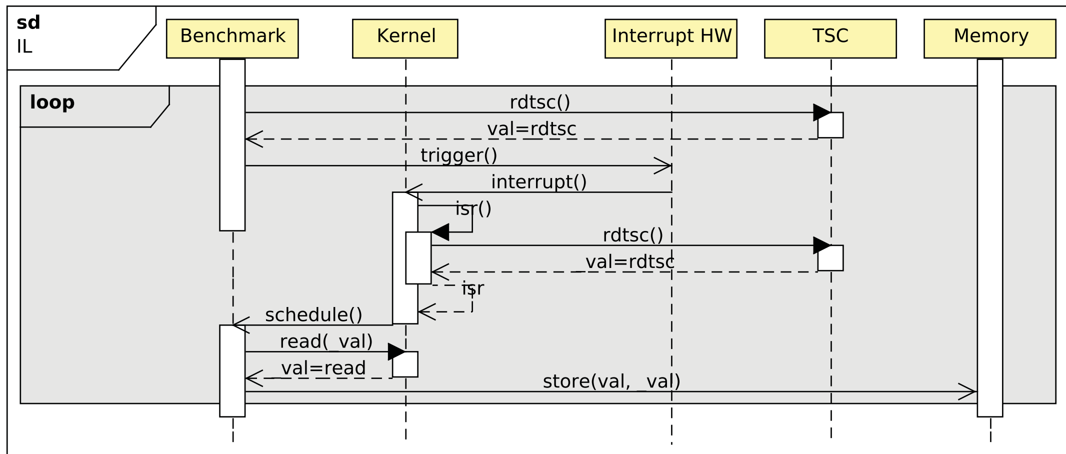

Figure 5.9.: Interrupt latency benchmark test

the previously interrupted task.

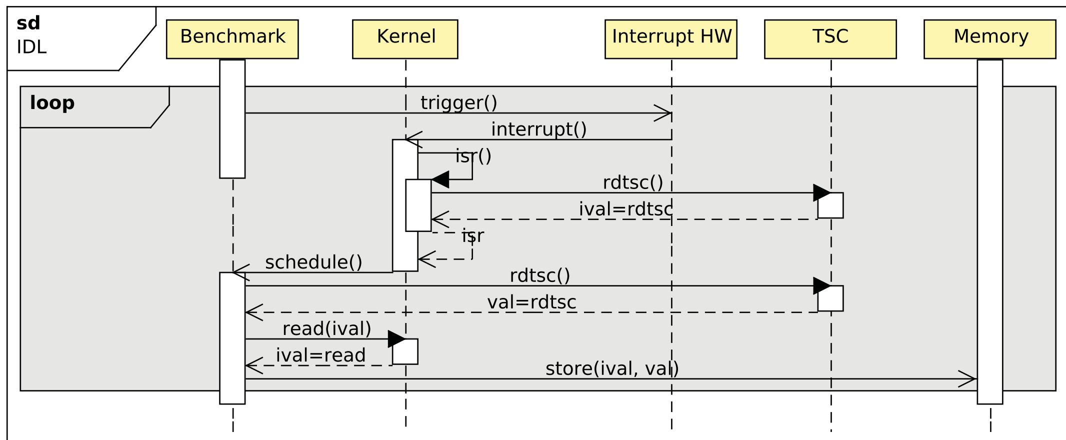

Figure 5.10.: Interrupt dispatch latency benchmark test

The test for the interrupt to task latency measurement as shown in Figure 5.11 requires a second level interrupt handler. Since the procedure for installing a SLIH strongly depends on the operating system used, details for SLIH are nor discussed here. Similar to the interrupt tests introduced above an interrupt is generated by calling the `trigger()` function. The task that triggered the interrupt is preempted by the handler. After a high level interrupt handler has treated the interrupt a SLIH task is scheduled. Inside the SLIH the current value of the TSC register is determined. This value represents the measurement together with the TSC register value determined before the interrupt was triggered. The measurement includes the time that is needed for a switch to the SLIH task.

Depending on the operating system different strategies have to be implemented for transferring measurement data between interrupt handler and benchmark task. In a QNX system a global variable can be used to share values from inside an interrupt handler and the benchmark task. In a Linux based system the `proc` interface is used.

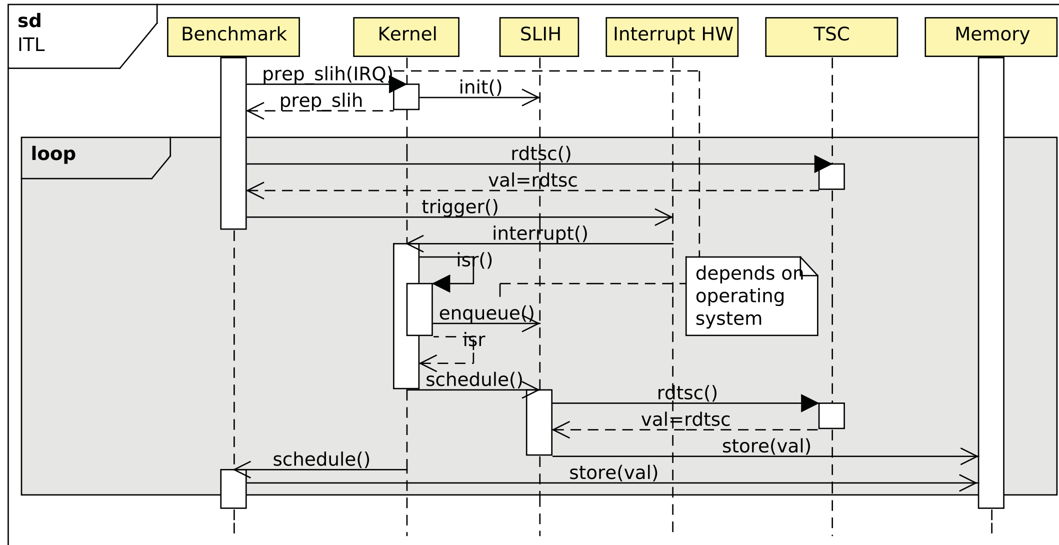

Figure 5.11.: Interrupt to task latency benchmark test

The actual implementation is explained in the chapters assigned to the operating systems.

6

Case Study 1: RT-Preempt Patch

The first detailed operating system analyses in this thesis refers to the RT-Preempt Linux extension. We will discuss the main contributions of the patch and analyse how real-time behaviour is achieved (see Section 3.6.2). Later in this chapter the implementation of the benchmark tests introduced in Section 5.4.3 for the RT-Preempt Linux system will be explained.

6.1. Background and Overview

The RT-Preempt extension for Linux was introduced by Ingo Molnar and Thomas Gleixner and is maintained by a small group of core developers. This extension allows many parts of the kernel to be preempted, with the exception of some regions of code. This is done by moving interrupts and software interrupts to kernel threads, as well as replacing most kernel spin-locks with mutexes that support priority inheritance. The key point of the extension is to minimize the amount of kernel code that is non-preemptible. In particular, critical sections and interrupt handlers are preemptible in a RT-Preempt Linux kernel. Real-time support is activated in the Kernel by setting few parameters. After configuration, the kernel can be compiled and installed as usual.

`CONFIG_PREEMPT = y` enables non-critical-section kernel preemption.

`CONFIG_PREEMPT_RT = y` enables full preemption, including critical sections. This option also enables: `CONFIG_PREEMPT_SOFTIRQS`, `CONFIG_PREEMPT_HARDIRQS`, and `CONFIG_PREEMPT_RCU`

`CONFIG_HIGH_RES_TIMERS = y` enables the High-Resolution-Timer option.

`CONFIG_ACPI = n` disables all power management options like ACPI or APM.

The RT-Preempt extension has raised quite some interest throughout the industry. This is mainly based on the contemporary availability of the extension for new Kernel versions. Furthermore, the extension can be integrated almost seamlessly into the Kernel. Real-time tasks make use of the standard Posix API. The extension only has impact on the kernel, userspace tasks do not notice the difference. However,

important things to keep in mind while writing realtime applications for the RT-Preempt extension are discussed in Section 6.4. Some techniques that were previously part of the RT-Preempt extension have been inherited by the mainstream kernel. So, the Linux timer API was separated into infrastructures for high resolution timers and timeouts, leading to user space POSIX timers with high resolution.

The RT-Preempt extension for Linux arrives as a patch that can be applied to the mainline kernel and is available for various kernel versions. The version of the patch on which this examination is based affects 652 files of the kernel source tree. Since many architectures included in the Linux kernel are supported by the patch, the number of changes in the architecture dependent files is quite high. While applying the patch 228 files are touched in the kernel arch source tree. 86 files among them belong to the `x86` (`x86_64`) architecture. Another big block of 186 changed files concerns drivers, filesystems and network depended code. In the following, the main focus is on the core kernel sections, such as scheduling and timing.

6.2. Preemptable In-Kernel Locking Primitives

The real-time patch converts most spin-locks in the system to mutexes. This reduces overall latency at the expense of slightly reduced throughput. The benefit of converting spin-locks to mutexes is that they can be preempted. When a task successfully acquires a spin-lock in a *normal* unpatched kernel, preemption is disabled and the task that acquired the spin-lock is allowed to enter the critical section. No task switches can occur until a `spin_unlock()` operation takes place. The `spin_lock()` function is actually a macro that has several forms, depending on the kernel configuration. They are defined at the architecture-independent top level definitions in `linux/spinlock.h`. When the kernel is patched with the RT-Preempt patch, these spin-locks are promoted to mutexes to allow preemption of higher-priority processes when a spin-lock is held .

spinlock.h

```
#define spin_lock(lock) rt_spin_lock(lock)
```

rtmutex.c

```
static void  noinline __sched
rt_spin_lock_slowlock(struct rt_mutex *lock)
{
...
        atomic_spin_lock_irqsave(&lock->wait_lock, flags);
...
        saved_state = rt_set_current_blocked_state(current->state);

        for (;;) {
                int saved_lock_depth = current->lock_depth;

                /* Try to acquire the lock */
                if (do_try_to_take_rt_mutex(lock, STEAL_LATERAL))
                        break;
...
                if (!waiter.task) {
                        task_blocks_on_rt_mutex(lock, &waiter, current,0,
```

```
                                                    flags);
...
                }
...
                atomic_spin_unlock_irqrestore(&lock->wait_lock, flags);
...
                if (adaptive_wait(&waiter, orig_owner)) {
                        put_task_struct(orig_owner);

                        if (waiter.task)
                                schedule_rt_mutex(lock);
                } else
                        put_task_struct(orig_owner);

                atomic_spin_lock_irqsave(&lock->wait_lock, flags);
...
        }

        rt_restore_current_state(saved_state);
...
        atomic_spin_unlock_irqrestore(&lock->wait_lock, flags);
...
}

void __lockfunc rt_spin_lock(spinlock_t *lock)
{
        rt_spin_lock_fastlock(&lock->lock, rt_spin_lock_slowlock);
        spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
}
```

Listing 6.1: RT-Preempt `spin_lock()`

Listing 6.1 shows the definition of the `spin_lock()` macro for a RT-Preempt patched kernel. The call to `rt_spin_lock_fastlock(..., f())` in `rt_spin_lock()` intercepts some error states and continues with calling the committed function `f()` (`rt_spin_lock_slowlock()`). Spin-locks in the *normal* Linux kernel are realised with busy waiting. The method avoids context switches that can be very time consuming. The locking task runs continually until the lock is acquired. Generally, a spin-lock should be held for a short amount of time because the waiters for the lock will be using CPU time. Blocking and preemption are illegal while holding a lock. A spin-lock inside a RT-Preempt kernel is implemented as a mutex (Listing 6.1). It causes waiters to sleep if the lock is held by a different task. With this it is possible (and valid) for a task to be suspended inside the spin-lock. Mutex based spin-locks can be acquired in some contexts inside a RT-Preempt patched kernel where blocking operations are not allowed in *normal* Linux.

+ A mutex can be acquired in a preemptable RCU read section.

+ A mutex can be acquired in most interrupt handlers and softirqs because these sections occur in thread context.[1]

[1]See Section 6.3 for details.

– A mutex cannot be used in interrupt context and must not be in atomic paths.

Critical sections protected by `spinlock_t` (respectively `rwlock_t`) objects are now preemptible. The creation of non-preemptible sections is still possible with the `raw_spinlock_t` type. Most *normal* kernel spin-locks are converted into RT-Preempt spin-locks. `raw_spinlock_t` should only be used for scheduler code, mutex implementation and for low level interrupt handlers (e.g. timer).

6.2.1. Priority Inheritance for In-Kernel Locking Primitives

Both methods, busy waiting spin-lock and real-time mutex, ensure that only one execution path enters a critical section. If a task tries to enter an already occupied section, it will be suspended (or blocked) until the task that has previously locked the section releases it. Since tasks which have locked a section using a mutex are preemptible a problem arises that is commonly known as priority inversion (Figure 6.1 a).

- Low-priority task 1 acquires a mutex.
- High-priority task 3 attempts to acquire the lock held by low-priority task 1 and blocks.
- Medium-priority task 2 starts executing and preempts low-priority task 1.

In the situation described above, medium-priority task 2 is executed before high-priority task 3. This violates the priority model that tasks can only be prevented from running by higher priority tasks (and briefly by low priority tasks which will quickly complete their use of a resource shared by the high and low priority tasks). Such priority inversion can indefinitely delay a high-priority task. There are two main ways to address this problem.

Suppressing preemption In this case, since there is no preemption, task 2 cannot preempt task 1, preventing priority inversion from occurring.

Priority inheritance High-priority task 3 temporarily donates its high priority to lower-priority task 1 that is holding the critical lock (Figure 6.1 b).

Priority inheritance is transitive. In the example above, if an even higher priority task 4 attempted to acquire a second lock that high-priority task 3 was already holding, then both tasks 3 and 1 would be temporarily boosted to the priority of task 4.

It may take some time for task 3 to run, and it is quite possible that another higher-priority task 5 will try to acquire the lock in the meantime. If this happens, task 5 will *steal* the lock from task 3, which is legal because task 3 has not yet run, and has therefore not actually acquired the lock. On the other hand, if task 3 gets to run before task 5 tries to acquire the lock, then task 5 will be unable to *steal* the lock, and must instead wait for task 3 to release it. So, the priority of task 3 will be boosted to the level of the priority of task 5 in order to expedite matters.

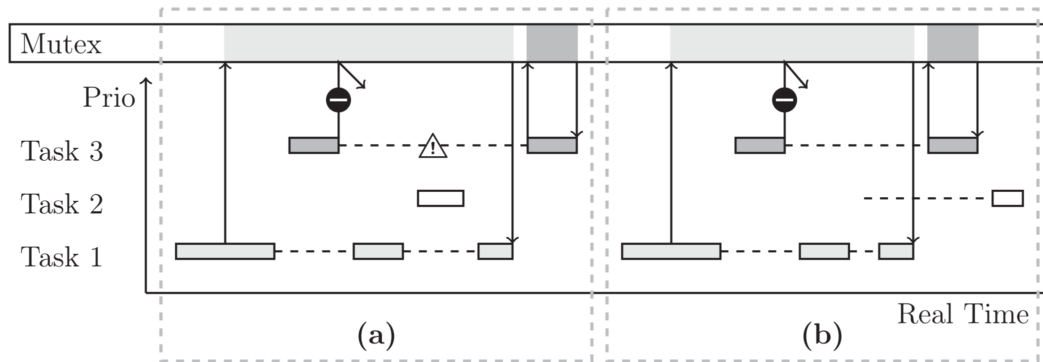

Figure 6.1.: Priority inversion

In the RT-Preempt extension implementation of priority inheritance the priority of a task is raised by a new task that tries to get a lock. The task that holds the lock gets the priority of the new task. This is done inside `task_blocks_on_rt_mutex()` which is called from `rt_spin_lock_slowlock` (Listing 6.1). After releasing the lock the previously boosted task falls back to its normal priority level (Listing 6.2). In both cases `rt_mutex_adjust_prio()` is responsible for changing the task priority. The details of the implementation are not shown here, since they are not of importance for the understanding of the RT-Preempt extension realisation of priority inheritance.

rtmutex.c

```
tatic void  noinline __sched
rt_spin_lock_slowunlock(struct rt_mutex *lock)
{
...
        atomic_spin_lock_irqsave(&lock->wait_lock, flags);
...
        wakeup_next_waiter(lock, 1);
...
        atomic_spin_unlock_irqrestore(&lock->wait_lock, flags);

        /* Undo pi boosting.when necessary */
        rt_mutex_adjust_prio(current);
}
```

Listing 6.2: RT-Preempt `spin_unlock()`

In a scenario with read/write locks it turns out that priority inheritance is particularly problematic. In such a situation a lock can be held by more than one task (reader) at a time. Since priority inheritance is transitive numerous tasks can be involved in this situation. To change and unchange the priorities of all these tasks can cause a indeterminate latency impact on the scheduling behavior. The RT-Preempt extension simplifies the problem by permitting only one task at a time to read-hold a read/write lock. Listing 6.3 shows that read/write locks with the RT-Preempt extension are implemented as rt-spin-locks (mutex) internally. Also see [RH07] for further information on read/write locks inside the RT-Preempt Linux extension.

rwlock.h

```
#define write_lock(lock)          rt_write_lock(lock)
```

```
#define read_lock(lock)         rt_read_lock(lock)
```

rt.c

```
void __lockfunc rt_write_lock(rwlock_t *rwlock)
{
        rwlock_acquire(&rwlock->dep_map, 0, 0, _RET_IP_);
        __rt_spin_lock(&rwlock->lock);
}

void __lockfunc rt_read_lock(rwlock_t *rwlock)
{
        struct rt_mutex *lock = &rwlock->lock;

        rwlock_acquire_read(&rwlock->dep_map, 0, 0, _RET_IP_);
...
        if (rt_mutex_real_owner(lock) != current)
                __rt_spin_lock(lock);
        rwlock->read_depth++;
}
```

rtmutex.c

```
void __lockfunc __rt_spin_lock(struct rt_mutex *lock)
{
        rt_spin_lock_fastlock(lock, rt_spin_lock_slowlock);
}
```

Listing 6.3: RT-Preempt `rw_lock()`

6.3. Interrupt Handlers as Kernel Threads

With the RT-Preempt extension many interrupt handlers run in task context. This threaded handlers have (like other tasks) a priority level and are thus integrated into the Linux task scheduling. It is now possible for a normal real-time tasks to run with higher priority than interrupt handlers . Therefore, most interrupt handlers are preemptible. Figure 6.2 shows the scheduling hierarchy in the RT-Preempt extension Linux.

Not all handlers can be delayed. Some interrupts like for instance per-CPU timer interrupts have to be treated immediately. There is still the need for such handlers to be executed in hardware-interrupt context. For this reason interrupts are devided into two categories :

IRQ_NODELAY Interrupts marked with the `IRQ_NODELAY` flag are caused to run in hardware-interrupt context (ISR). In the standard RT-Preempt extension configuration, only per-CPU timer interrupts (scheduler tick) and floating-point co-processor interrupts have this flag specified. In this category the preemptable spin-locks must not be used for synchronization, since hardware-interrupts are not preemptible. The `raw_spinlock_t` type can be used for mutual exclusion instead.

A Linux kernel module (e.g. driver) can register a new interrupt service routine (ISR) that is forced to be executed in hardware-interrupt context by passing

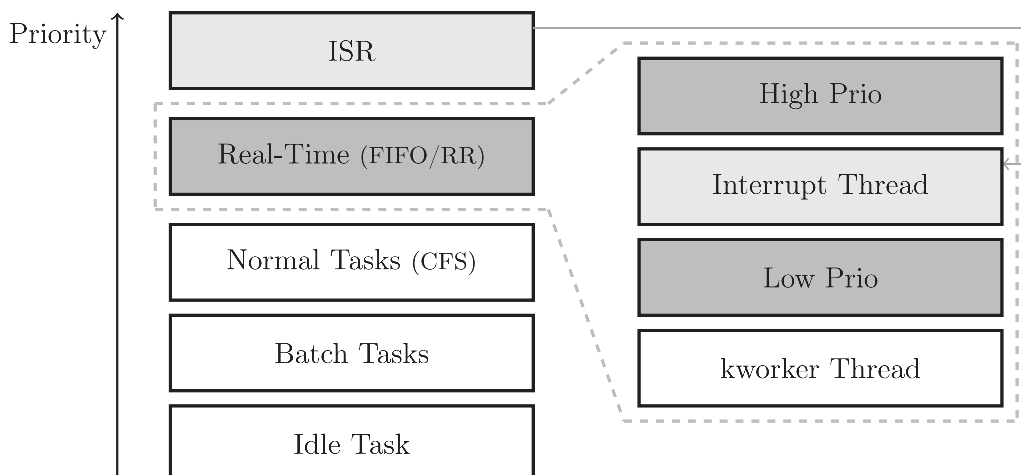

Figure 6.2.: RT-Preempt scheduling hierarchy

the `IRQ_NODELAY` flag. Both, interrupt and scheduling latencies can be greatly degraded with the increased occurance of non preemptible interrupts.

The implementation of `IRQ_NODELAY` interrupt handlers and code sequences interacting with such handlers need to follow some special rules. Code that must interact with `IRQ_NODELAY` interrupts cannot use `local_irq_save()`, since this does not disable hardware interrupts in the RT-Preempt extension. Instead, `raw_local_irq_save()` should be used. Similarly, raw spin-locks need to be used when interacting with `IRQ_NODELAY` interrupt handlers.

Virtualized A virtualized interrupt is split into a hardware-interrupt and a threaded part. The actual service routine (ISR) is evacuated to a handler task. Like mentioned before this interrupt task is preemptible. The hardware part is responsible for storing the interrupt in a queue and activating the handler task. When the handler task is scheduled each ISR in the queue is processed. A real-time task with a higher priority level than the handler task can preempt the processing of the ISR queue.

To handle races with interrupt handlers a threaded interrupt can be delegated to another CPU in the system. Any code that interacts with an interrupt handler must be prepared to deal with that interrupt handler running concurrently on some other CPU.

The standard Linux API provides `request_irq()` for introducing interrupt handlers. It is actually a wrapper to `request_threaded_irq()` which supports the concept of threaded interrupt handlers already. However, almost all drivers in the Kernel use `request_irq()` for requesting hardware-interrupts. Detailed information about how interrupts are treated in the Linux kernel can be found at [BC05, Chap. 4]. In general, the handler function is invoked in hardware-interrupt context and tells the Linux framework either to wake up the associated interrupt task or not. The interrupt task calls the thread function to finish the interrupt handling.

Listing 6.4 outlines the process of registering an interrupt handler. The reference to `thread_fn` is `NULL` if a new handler is registered with the `request_irq()` method. This causes `preempt_hardirq_setup()` to define the thread function to be the handler function if the `IRQ_NODELAY` flag is not set. The handler function is replaced by the standard handler function which just returns with `IRQ_WAKE_THREAD`.

interrupt.h

```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char *name, void *dev)
{
        return request_threaded_irq(irq, handler, NULL, flags, name,dev);
}
```

manage.c

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                         irq_handler_t thread_fn, unsigned long irqflags,
                         const char *devname, void *dev_id)
{
...
        action->handler = handler;
        action->thread_fn = thread_fn;
        action->flags = irqflags;
...
        retval = __setup_irq(irq, desc, action);
...
}

static int
__setup_irq(unsigned int irq,struct irq_desc *desc,struct irqaction *new)
{
...
        /* Preempt-RT setup for forced threading */
        preempt_hardirq_setup(new);
...
        if (new->thread_fn && !nested) {
                struct task_struct *t;

                t = kthread_create(irq_thread, new, "irq/%d-%s", irq,
                                   new->name);
...
                new->thread = t;
        }
}

static void preempt_hardirq_setup(struct irqaction *new)
{
        if (new->thread_fn || (new->flags &(IRQF_NODELAY | IRQF_PERCPU)))
                return;

        new->flags |= IRQF_ONESHOT;
        new->thread_fn = new->handler;
        new->handler = irq_default_primary_handler;
```

```
}
```

Listing 6.4: RT-Preempt interrupt request

An interrupt will be handled in task context, if the handler is registered with a call to `request_irq()` and the `IRQ_NODELAY` flag is not set. The new task is created in `__setup_irq()` if necessary and runs at a priority level of $\frac{MAX_USER_RT_PRIO}{2}$. In combination with the RT-Preempt spin-lock implementation interrupt handlers are fully preemptable.

6.4. Real-Time Application Programming

No special API is required in the RT-Preempt extension. Real-time tasks make use of the standard Posix API. The RT-Preempt internet presence[2] mentions some important things to keep in mind while writing real-time applications:

- A real-time application uses the `SCHED_RR` or the `SCHED_FIFO` scheduling policy.
- A call to `mlockall()` is requiered as soon as possible in order to lock the calling process's virtual address space into RAM and preventing that memory from being paged to the swap area.
- All threads should be created at the startup time of the application. Creating threads during rel-time execution will ruin the real-time behavior.
- Each memory page of the entire stack of each thread should be touched as soon as possible (after calling `mlockall()`) in order to cause a stack fault at a determined time.
- A real-time application should not run on the highest priority level since there are a few management threads which need to run with higher priority then the application.
- When the system is fulfilling its real-time requirements the VGA text console must be left untouched. Nothing is allowed to be written to that console. (Using a graphical interface based on the X window system has no impact on the real-time behavior.)

Example applications can be seen in Section 6.5.

6.5. Benchmarking

The benchmark tests described in Section 5.4.3 are going to be discussed in this section in concrete implementation for the Linux 3.4.104 kernel. Since the API for a kernel with RT-Preempt extension does not differ from the standard kernel, the implemented

[2] `https://rt.wiki.kernel.org`

tests are executable for both kernels. The presented values in this section are the results from executing the same tests for a patched and a non-patched kernel. This makes it easy to see the impact of the RT-Preempt extension. In Chapter 12 the benchmark results are compared and evaluated to other real-time operating systems. All benchmark results can be accessed at `http://www.informatik.uni-bremen.de/agbs/dirkr/HRTL/benchmark_results.tgz`.

Listing 6.5 shows the main setup for a real-time process in the RT-Preempt extension operating system. Each task calls this function (in a slightly modified version with respect to the process priority) to become a real-time task.

period_....c

```
int setup(void) {
    cpu_set_t set;
    struct sched_param schedp = { .sched_priority = RT_PRIO };

    CPU_ZERO(&set);
    CPU_SET(BENCHMARK_CPU, &set);

    /* declare as a real time task */
    if (0 > sched_setscheduler(0, SCHED_FIFO, &schedp)) {
...
    /* lock CPU */
    if (0 > sched_setaffinity(0, sizeof(cpu_set_t), &set)) {
...
    /* lock memory */
    if (0 > mlockall(MCL_CURRENT | MCL_FUTURE)) {
...
}
```

Listing 6.5: RT-Preempt benchmark test setup

As mentioned in Section 5.4.1 a task has to arrange some sort of affinity to a certain CPU in order to keep the TSC register values comparable. This is done by calling `sched_setaffinity()` with a prepared `cpu_set_t` variable. Further, this real-time task determines its scheduling policy to be `SCHED_FIFO` and locks its memory by calling `mlockall()`.

6.5.1. Task Period Tests

The periodic task benchmark test is presented in Listing 6.6. As can be seen, the setup function (Listing 6.5) is called early in the main function. The parallel port initialisation takes place before calling `setup()` and is not shown in the listing. The memory area for the measurement results (`tsc[]`) is touched before executing the main loop. As described in Section 6.4 this is necessary to cause a stack fault before the test starts.

period_....c

```
int main(int argc, char **argv) {
    uint32_t tsc[LOOP_COUNT];
...
    if (0 > setup())
        exit(EXIT_FAILURE);
```

```
    /* pre-fault stack */
    for (i = 0; i < LOOP_COUNT; i++)
        rdtsc_32(tsc[i]);

    if (0 > start_timer(atoi(argv[1]), atoi(argv[2])))
        exit(EXIT_FAILURE);

    /* benchmark */
    for (i = 0; i < LOOP_COUNT; i++) {
        select(0, NULL, NULL, NULL, NULL);
        rdtscp_32(tsc[i]);
        cpuid();
#ifdef TRIGGER_PARPORT
        parport_toggle();
#else
        busy();
#endif
    }
...
}

int start_timer(unsigned int runtime_sec, unsigned int runtime_us) {
    struct itimerval ival = {
        .it_interval = { .tv_sec = runtime_sec, .tv_usec = runtime_us},
        .it_value =    { .tv_sec = runtime_sec, .tv_usec = runtime_us}
    };

    if (0 > setitimer(ITIMER_REAL, &ival, NULL)) {
...
}
```

Listing 6.6: RT-Preempt period task benchmark test

In the test the periodic task behavior is realised by using an interval timer, programmed with `setitimer()`. Upon expiration of the programmed timer, a `SIGALRM` signal will be generated. The signal will be delivered immediately when generated. The appropriated `select()` system call in the main loop blocks until the signal is received by the process. This signal is captured in a signal handler and indicates the start of a new period.

The first benchmark test for the RT-Preempt extension operating system measures the scheduling precision of a periodic task with a timer of 500 μs (Table 6.1). The test was executed in the 3 scenarios described in Section 5.3.

As one can see the precision of the 500 μs timer is almost met. The introduced measurement details in Section 5.4.1 explain the acquisition of the time values. Results in the table were converted to the μs unit. The translation caused some inaccuracy in the precision of the values due to rounding errors.

Compared with the results of the same benchmark test on a native Linux kernel system (Table 6.2) it is easy to see, that the patch brings a higher precision for a small timer to the Linux operating system. Like mentioned before the test was executed

Scenario	Average	Min	Max	Gap	Deviation
Normal	499.931	498.245	501.202	2.957	0.969
CPU utilization	499.952	497.342	502.438	5.096	1.125
I/O utilization	499.919	495.025	505.939	10.914	1.198

Table 6.1.: Benchmark test results [μs]: RT-Preempt period task (500μs)

with the same kernel without applying the RT-Preempt patch.

Scenario	Average	Min	Max	Gap	Deviation
Normal	499.968	492.642	505.692	13.050	1.882
CPU utilization	499.954	476.227	524.673	48.446	2.748
I/O utilization	499.978	491.964	508.623	16.659	2.250

Table 6.2.: Benchmark test results [μs]: Linux 3.4.104 period task (500μs)

Table 6.3 and Table 6.4 show the results of the periodic benchmark test with a 20 times larger timer. In the CPU and I/O utilization scenarios the RT-Preempt system loses some accuracy with a 10 ms timer.

Scenario	Average	Min	Max	Gap	Deviation
Normal	9998.973	9994.927	10003.621	8.694	0.681
CPU utilization	9999.048	9990.908	10005.985	15.076	2.142
I/O utilization	9999.041	9988.103	10010.922	22.820	1.722

Table 6.3.: Benchmark test results [μs]: RT-Preempt period task (10ms)

Scenario	Average	Min	Max	Gap	Deviation
Normal	9999.043	9994.372	10004.310	9.938	1.523
CPU utilization	9999.083	9931.113	10077.274	146.161	12.243
I/O utilization	9999.041	9988.460	10009.622	21.161	3.551

Table 6.4.: Benchmark test results [μs]: Linux 3.4.104 period task (10ms)

The same test is repeated with a 100 ms (Table 6.5 and Table 6.6) and a 1 second timer (Table 6.7 and Table 6.8). Nevertheless, the test is only executed in the normal scenario. What is striking is the continuous loss of precision in the RT-Preempt system with increasing timer values. In contrast, the normal Linux kernel gains some accuracy with higher timer values.

Scenario	Average	Min	Max	Gap	Deviation
Normal	99990.428	99983.862	99997.116	13.253	2.187

Table 6.5.: Benchmark test results [μs]: RT-Preempt period task (100ms)

Scenario	Average	Min	Max	Gap	Deviation
Normal	99990.394	99985.707	99995.328	9.621	0.876

Table 6.6.: Benchmark test results [μs]: Linux 3.4.104 period task (100ms)

Scenario	Average	Min	Max	Gap	Deviation
Normal	999903.946	999899.019	999909.010	9.991	2.526

Table 6.7.: Benchmark test results [μs]: RT-Preempt period task (1sec)

Scenario	Average	Min	Max	Gap	Deviation
Normal	999903.939	999898.414	999908.540	10.126	1.163

Table 6.8.: Benchmark test results [μs]: Linux 3.4.104 period task (1sec)

6.5.2. Task Switch Tests

As described in Section 5.2.1 two different tests for measuring task switch latency are implemented. Like before in the periodic task benchmarking these tests are performed in the described scenarios for a native Linux kernel and a RT-Preempt system.

Listing 6.7 shows the implementation of the startup routine for the task preemption latency benchmark test. Two semaphores (`SEM_FORK` and `SEM_WAIT`) are used as events for synchronising the start of the test. Since, actually three different processes are involved in test executing, the results are stored in a shared memory segment. The initialisation of the semaphores and the shared memory segment are not shown in the listing.

switch_...signal.c

```
int main(void) {
...
    struct sched_param schedp = { .sched_priority = RT_PRIO -2};
...
    if (0 > setup(RT_PRIO +1))
        exit(EXIT_FAILURE);
...
    if (0 == (high = fork())) {
        if (0 > setup(RT_PRIO))
            exit(EXIT_FAILURE);

        task_high(1);
    } else
        semop(sem_id, &sem_op_fork, 1);
```

```
    semctl(sem_id, SEM_FORK, SETVAL, 1);

    if (0 == fork()) {
        if (0 > setup(RT_PRIO -1))
            exit(EXIT_FAILURE);

        task_low(0, high);
    } else
        semop(sem_id, &sem_op_fork, 1);

    semop(sem_id, &sem_op_wait, 1);
    sched_setparam(0, &schedp);
...
}
```

Listing 6.7: RT-Preempt task preemption benchmark test startup

Two processes are forked during test startup. Each created process calls the `setup()` function at first. The process related startup routines are shown later in this section. One of the semaphores (`SEM_FORK`) is used here to let the main process block until the new created process finishes its own setup phase. After both processes have finished their startup the main process fires the second event (`SEM_WAIT`) and lowers its priority level. Both forked processes have higher priority than the main process now. If they terminate, the main process comes back to life and finishes the benchmark test by printing the results.

The main test takes place between the newly created processes.[3] Listing 6.8 shows the main routines for both tasks. After performing the startup synchronisation as mentioned above (semaphores `SEM_FORK` and `SEM_WAIT`), the benchmark test starts with entering the `for` loop. The POSIX signal mechanism, which is implemented in the Linux operating system, is used for triggering the higher priority task. `task_low` sends a signal (`SIGALRM`) to `task_high` with the `kill()` system call for waking up that task. `task_high` previously blocked on the `pause()` system call which causes the calling process to sleep until a signal is delivered that causes the invocation of a signal-catching function.

switch_...signal.c

```
void task_low(int idx, pid_t high) {
...
    for (i = 0; i < LOOP_COUNT; i++)
        rdtsc_32(tsc[i]);

    semop(sem_id, &sem_op_fork, 1);
    semop(sem_id, &sem_op_wait, 1);

    for (i = 0; i < LOOP_COUNT; i++) {
        busy_long();
        cpuid();
        rdtsc_32(tsc[i]);
        kill(high, SIGALRM);
    }
```

[3] See Section 5.4.3.2 for further explanation.

```
...
    if (NULL == (res = shmat(shm_id, 0, 0))) {
...
    }

    for (i = 0; i < LOOP_COUNT; i++)
        res->tsc[idx][i] = tsc[i];
...
}

void task_high(int idx) {
...
    for (i = 0; i < LOOP_COUNT; i++)
        rdtsc_32(tsc[i]);

    signal(SIGALRM, task_high_sighandler);
    semop(sem_id, &sem_op_fork, 1);
    semop(sem_id, &sem_op_wait, 1);

    for (i = 0; i < LOOP_COUNT; i++) {
        busy_long();
        pause();
        rdtscp_32(tsc[i]);
        cpuid();
    }

    if (NULL == (res = shmat(shm_id, 0, 0))) {
...
    }

    for (i = 0; i < LOOP_COUNT; i++)
        res->tsc[idx][i] = tsc[i];
...
}
```

Listing 6.8: RT-Preempt task preemption benchmark test

The benchmark test was executed in the 3 scenarios described in Section 5.3. Table 6.9 and Table 6.10 show the results of the test. As one can see the RT-Preempt extension slows down the preemption of a task by a factor of 2.5 compared to a native Linux kernel.

Scenario	**Average**	**Min**	**Max**	**Gap**	**Deviation**
Normal	4.699	4.620	8.084	3.463	0.222
CPU utilization	4.728	4.665	8.639	3.973	0.253
I/O utilization	4.776	4.653	8.900	4.248	0.276

Table 6.9.: Benchmark test results [μs]: RT-Preempt preempt task (signal)

The second benchmark test for measuring the task switch latency in a RT-Preempt extension kernel is also described in Section 5.4.3.2. To simplify the complexity of

Scenario	Average	Min	Max	Gap	Deviation
Normal	1.538	1.501	1.709	0.208	0.026
CPU utilization	1.543	1.491	1.737	0.245	0.025
I/O utilization	1.506	1.486	1.641	0.155	0.023

Table 6.10.: Benchmark test results [μs]: Linux 3.4.104 preempt task (signal)

the program, the arrangement of the shared memory segment is slightly more tricky compared to the task preemption benchmark test above. Because up to 512 processes are involved in test execution, the process ID is stored together with each measurement value to make the printed results more expressive. Listing 6.9 shows the data structure for the shared memory segment.

switch_same....c

```
struct tsc_tab_entry {
    pid_t pid;
    uint32_t tsc;
};

struct tsc_tab {
    unsigned int idx_start;
    unsigned int idx_stop;

    struct tsc_tab_entry start[LOOP_COUNT];
    struct tsc_tab_entry stop[LOOP_COUNT];
};

void tsc_tab_init(struct tsc_tab *tab) {
    tab->idx_start = 1;
    tab->idx_stop = 0;
}
```

Listing 6.9: RT-Preempt task switch benchmark test data structure

Each process that is part of the benchmark test stores two values in the table within each iteration of the test main loop (Listing 6.10). The following functions are provided for manipulating values in the table:

`tsc_tab_write_start(..., int idx, ...)` Write the given value to the `start` table at the position marked by `idx`.

`tsc_tab_write_stop()` Write the given value to the `stop` table. The index for the `stop` table is incremented by 1 afterwards.

`tsc_tab_get_start_idx()` Get the next index for the `start` table. The index for the `start` table is incremented by 1 afterwards. If the end of the table is reached, -1 is returned and the calling process will terminate (Listing 6.10).

Inside the test main loop, the process reserves an index for the table by calling `tsc_tab_get_start_idx()`. In the next turn the benchmark values are stored

locally in the process memory stack. In the section afterwards the previously stored values are transmitted to the table by using the reserved start index outside the time tracking. The actual task switch is invoked by calling the `sched_yield()` system call.

switch_same....c

```
void task(void) {
...
    do {
        busy_long();
        cpuid();
        rdtsc_32(tsc_start);
        sched_yield();
        rdtscp_32(tsc_stop);
        cpuid();

        if (-1 != idx)
            tsc_tab_write_start(res, idx, pid, tsc_start);

        tsc_tab_write_stop(res, pid, tsc_stop);
        idx = tsc_tab_get_start_idx(res);
    } while (-1 != idx);
...
}
```

Listing 6.10: RT-Preempt task switch benchmark test

The test startup is almost the same as for the task preemption benchmark test before. Details are not printed here. All processes needed for the test execution are forked within the main process and use the same synchronisation mechanism (semaphores as events).

Table 6.11 and Table 6.12 present the results of the task switch latency benchmark test on a RT-Preempt extension kernel respectively native Linux kernel with two alternating processes. The time required for a task switch is slightly higher in the RT-Preempt extension kernel. However, the smaller deviation value shows a minimal better reproducibility.

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.617	0.607	0.634	0.027	0.004
CPU utilization	0.616	0.605	0.658	0.052	0.007
I/O utilization	0.623	0.602	0.640	0.037	0.004

Table 6.11.: Benchmark test results [μs]: RT-Preempt switch task (2 tasks)

The same test was repeated with 16 (Table 6.13 and Table 6.14), 128 (Table 6.15 and Table 6.16) and 512 (Table 6.17 and Table 6.18) switching processes. The time required for a task switch increases with more involved processes.

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.453	0.450	0.498	0.048	0.006
CPU utilization	0.461	0.451	0.510	0.059	0.006
I/O utilization	0.460	0.449	0.492	0.043	0.008

Table 6.12.: Benchmark test results [μs]: Linux 3.4.104 switch task (2 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.698	0.659	0.752	0.093	0.016

Table 6.13.: Benchmark test results [μs]: RT-Preempt switch task (16 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.571	0.520	0.615	0.095	0.015

Table 6.14.: Benchmark test results [μs]: Linux 3.4.104 switch task (16 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.910	0.828	1.068	0.240	0.037

Table 6.15.: Benchmark test results [μs]: RT-Preempt switch task (128 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.776	0.708	0.891	0.183	0.027

Table 6.16.: Benchmark test results [μs]: Linux 3.4.104 switch task (128 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	1.309	1.073	1.769	0.695	0.114

Table 6.17.: Benchmark test results [μs]: RT-Preempt switch task (512 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	1.257	0.936	1.622	0.686	0.126

Table 6.18.: Benchmark test results [μs]: Linux 3.4.104 switch task (512 tasks)

6.5.3. Task Creation Test

The time it takes for creating a new process is measured by the task creation benchmark test. According to the description in Section 5.4.3.3 a new task is spawned in each test step within the test main loop by calling the `fork()` system call. Time is measured immediately before and after (in the new process) invoking `fork()`. To transfer the

second measurement value to the main process a shared memory segment is used. Listing 6.11 shows the implementation of the task creation benchmark test for the RT-Preempt extension operating system. The startup procedure is not shown in the listing, since there are no new steps to be introduced. All necessary preparation steps for the test execution were introduced before with the explanations of the other benchmark tests.

fork_same_prio.c

```
int main(void) {
...
    for (i = 0; i < LOOP_COUNT; i++) {
        res->start[i].pid = pid;
        cpuid();
        rdtsc_32(res->start[i].tsc);

        if (0 == fork()) {
            rdtscp_32(res->stop[i].tsc);
            cpuid();
            res->stop[i].pid = getpid();
            exit(EXIT_SUCCESS);
        }
...
    }
...
}
```

Listing 6.11: RT-Preempt task creation benchmark test

In a Linux system (and RT-Preempt extension kernel) a newly created process inherits the priority level and the scheduling policy of the parent process. The new process is an exact duplicate of the calling process except some points that are not discussed here. Subsequently the created process is put at the start of the FIFO run-queue within the kernel.[4] Therefore, the new created process will preempt the currently running process (parent).

The results of the task creation benchmark test are shown in Table 6.19 and Table 6.20. There are no significant differences between the results of a RT-Preempt extension kernel and a native Linux kernel.

Scenario	Average	Min	Max	Gap	Deviation
Normal	35.031	29.954	39.254	9.300	2.228
CPU utilization	37.148	31.011	39.269	8.259	1.478
I/O utilization	35.269	30.640	39.264	8.624	2.253

Table 6.19.: Benchmark test results [μs]: RT-Preempt task creation

[4] The POSIX standard specifies that the thread should go to the end of the list.

Scenario	Average	Min	Max	Gap	Deviation
Normal	33.482	29.731	35.695	5.964	1.480
CPU utilization	35.282	30.205	39.198	8.993	2.294
I/O utilization	33.892	30.595	35.707	5.112	1.121

Table 6.20.: Benchmark test results [μs]: Linux 3.4.104 task creation

6.5.4. Interrupt Tests

The implementation of the three interrupt benchmark tests as described in Section 5.4.3.4 are explained in this section. For the realisation of the tests, it is necessary to enhance the kernel with two modules. These modules implement the interrupt handlers that will be registered on the parallel port interrupt.

Interrupt latency, interrupt dispatch latency For these tests the interrupt handler just captures the current value of the TSC register and returns. The handler is shown in Listing 6.12.

irq_benchmark.c

```
static irqreturn_t irq_handler(int irq, void *__hw_irq)
{
        irqreturn_t result = IRQ_HANDLED;

        rdtsc_32(tsc);
        return result;
}
```

Listing 6.12: RT-Preempt interrupt benchmark test handler

Interrupt to task latency For this test the Linux tasklet mechanism is used. The handler enqueues the tasklet with the occurrence of the interrupt and returns. The kernel will schedule the tasklet kernel thread. Inside the tasklet the current value of the TSC register is determined. Listing 6.13 shows the implementation of the handler and the tasklet.

irq_benchmark.c

```
static irqreturn_t irq_handler(int irq, void *__hw_irq)
{
        irqreturn_t result = IRQ_HANDLED;

        tasklet_schedule(&interrupt_latency_tasklet);
        return result;
}

static void interrupt_latency_do_tasklet(unsigned long unused)
{
        rdtscp_32(tsc);
        cpuid();
}
```

Listing 6.13: RT-Preempt interrupt benchmark test tasklet handler

Values between the main benchmark test and the measurements inside the interrupt handler respective tasklet are transmitted via the *proc* interface. The initialisation of the proc interface, the interrupt registration and the declaration of the tasklet are not shown here. Both modules will create a file /proc/interrupt_latency. A simple read on that file will return the result of the last measurement. It is important for test execution to bind the interrupt treatment of the parallel port interrupt to a certain CPU. This is done in the Linux system by writing a affinity mask to /proc/irq/IRQ_NUMBER/smp_affinity.[5] smp_affinity defines the CPU cores that are allowed to execute the ISR for an interrupt. The value stored in this file is a hexadecimal bit-mask representing all CPU cores in the system.

With the introduced interrupt handler, measuring the interrupt latency is quite simple. Listing 6.14 shows the implementation of the interrupt latency benchmark test main loop. The interrupt is triggered with the benchmark framework functions parport_low() and parport_high().

interrupt_isr.c

```
int main(void) {
...
    for (i = 0; i < LOOP_COUNT; i++) {
...
        parport_low();
        busy();
        cpuid();
        rdtsc_32(res.start[i]);
        parport_high();

        do {
...
            fscanf(fh, "%s\n", tmp);
            res.stop[i] = strtoll(tmp, NULL, 10);
        } while (old == res.stop[i]);

        old = res.stop[i];
    }
...
}
```

Listing 6.14: RT-Preempt interrupt latency benchmark test

The results of the benchmark test are shown in Table 6.21 and Table 6.22. Here one can clearly see the impact of the RT-Preempt extension. The overall time for handling a (threaded) interrupt in a RT-Preempt kernel is much longer compared to a native Linux kernel, but under heavy I/O load the RT-Preempt extension enables the kernel to react still within accurate time.

The interrupt dispatch latency benchmark test is similar to the interrupt latency benchmark test except for the time measurement points. For this test the first value is captured within the kernel. The second time value is gathered when returning from interrupt. Listing 6.15 illustrates the interrupt dispatch latency benchmark test.

int...dispatch.c

```
int main(void) {
```

[5]The IRQ_NUMBER for the parallel port interrupt is 7 for most systems.

Scenario	Average	Min	Max	Gap	Deviation
Normal	9.282	8.165	10.620	2.454	0.651
CPU utilization	9.321	8.184	11.238	3.054	0.657
I/O utilization	9.378	8.210	10.828	2.617	0.668

Table 6.21.: Benchmark test results [μs]: RT-Preempt interrupt latency (ISR)

Scenario	Average	Min	Max	Gap	Deviation
Normal	4.558	3.190	5.921	2.731	0.693
CPU utilization	5.848	3.275	8.467	5.192	1.066
I/O utilization	4.931	3.227	55.150	51.923	3.602

Table 6.22.: Benchmark test results [μs]: Linux 3.4.104 interrupt latency (ISR)

```
...
    for (i = 0; i < LOOP_COUNT; i++) {
        busy();
        parport_low();
        busy();
        parport_high();

        do {
            rewind(fh);
            fscanf(fh, "%s\n", tmp);
            res.start[i] = strtoll(tmp, NULL, 10);
        } while (old == res.start[i]);

        rdtscp_32(res.stop[i]);
        cpuid();
        old = res.start[i];
    }
...
}
```

Listing 6.15: RT-Preempt interrupt dispatch latency benchmark test

One problem here is the slight delay when triggering the interrupt. For this reason the test *busy waits* for the measurement value provided with the proc system file to change. The test will be interrupted within this loop, but an additional comparison for leaving the `while` loop takes place before the measurement is completed. Another problem is that the file system layer is involved in the measurement. Since the same test is executed here for both operating systems, the overhead of the `read()` system call is negligible. This issue will be discussed in Chapter 12 again when comparing the RT-Preempt extension with other operating systems.

The results of the interrupt dispatch latency benchmark test are provided in Table 6.23 and Table 6.24.

Scenario	Average	Min	Max	Gap	Deviation
Normal	6.573	5.505	7.498	1.993	0.678
CPU utilization	6.616	5.514	7.497	1.983	0.666
I/O utilization	6.625	5.520	7.499	1.979	0.682

Table 6.23.: Benchmark test results [μs]: RT-Preempt interrupt latency (dispatch)

Scenario	Average	Min	Max	Gap	Deviation
Normal	1.950	1.166	3.111	1.945	0.445
CPU utilization	12.527	1.148	20.620	19.472	4.809
I/O utilization	1.889	1.084	4.679	3.595	0.530

Table 6.24.: Benchmark test results [μs]: Linux 3.4.104 interrupt latency (dispatch)

The interrupt to task latency benchmark test is identical to the interrupt latency benchmark test and is not listed here. The results of the test are shown in Table 6.25 and Table 6.26. Again, the enrichment of the RT-Preempt extension can be seen in the results for an I/O loaded system.

Scenario	Average	Min	Max	Gap	Deviation
Normal	9.337	8.217	10.505	2.288	0.639
CPU utilization	9.392	8.170	10.499	2.329	0.649
I/O utilization	9.460	8.166	11.498	3.332	0.695

Table 6.25.: Benchmark test results [μs]: RT-Preempt interrupt latency (SLIH)

Scenario	Average	Min	Max	Gap	Deviation
Normal	4.848	3.265	23.128	19.863	1.325
CPU utilization	7.546	5.061	13.107	8.046	1.099
I/O utilization	5.193	3.432	55.033	51.600	4.579

Table 6.26.: Benchmark test results [μs]: Linux 3.4.104 interrupt latency (SLIH)

6.6. Summary

The RT-Preempt Linux extension introduces some significant changes to the Linux kernel. The fully preemptable design of the kernel makes it possible to satisfy the period of a task with a failure in the range of microseconds. Later in Chapter 11 we will see how the approach of preemptible spin-locks is adapted by the operating

system developed in this thesis.

Besides preemptible spin-locks the concept of threaded interrupt handlers represents another major contribution to the normal Linux kernel. Since many aspects of the RT-Preempt patch are already adapted by the native kernel, this concept is not completely new. Together with other strategies discussed in the next chapters a threaded interrupt handler model will be developed for the operating system introduced in Part III.

7

Case Study 2: HaRTLinC

In this chapter we will discuss the technical details of the HaRTLinC Linux extension. Compared to the RT-Preempt patch (Chapter 6) the HaRTLinC system is pursuing a different strategy to achieve real-time behavior (see Section 3.6.3). Later in this chapter the implementation of the benchmark tests introduced in Section 5.4.3 for the HaRTLinC Linux system will be explained.

7.1. Background and Overview

The HaRTLinC project (HLRT) was originally created as a student project at the University of Bremen. It is an improvement of the Linux kernel modifications described by Klaas-Henning Zweck in his diploma thesis *Kernelbasierte Echtzeiterweiterung eines Linux-Multiprozessor-Systems* [Zwe02]. Zweck introduced a new scheduling policy. A task using this scheduling policy runs on its own exclusively reserved CPU and is not suspended by any hardware interrupts. Christof Efkemann developed a Linux extension based on the HLRT project and released a patch for the kernel with his diploma thesis *Development and evaluation of a hard real-time scheduling modification for Linux 2.6* [Efk05]. The main features of the HLRT extension are a CPU reservation mechanism and a periodic scheduling mechanism. The first one provides real-time scheduling for a task in a patched Linux system (Section 7.2). The periodic scheduling mechanism provides periodic execution of tasks with fixed intervals and detection of missed deadlines (Section 7.3).

After applying the HLRT patch to the kernel, real-time support is activated in the Kernel by setting few parameters. After configuration, the kernel can be compiled and installed as usual.

`CONFIG_HLRT = y` enables the HLRT extension features.

`CONFIG_HLRT_NOROOT = y` allows all users to use the HLRT scheduling policy.

`CONFIG_HIGH_RES_TIMERS = n` not supported by HLRT.[1]

`CONFIG_ACPI = n` disables all power management options like ACPI or APM.

[1] See the kernel configuration menu for the HLRT extension `arch/x86/Kconfig`.

The HLRT Linux extension is used and maintained by *Verified Systems International*[2] for computer cluster based embedded systems testing. Unlike the RT-Preempt extension (Chapter 6) real-time tasks in a HLRT system need to use a special API. The API for the HLRT extension is provided with the patch and described in Section 7.4.

The HLRT extension for Linux arrives as a patch that can be applied to the mainline Kernel. It is only available for the 32 bit `x86` architecture. The patch affects 40 files of the Kernel source tree.

7.2. CPU Reservation

A task in the *normal* unpatched Linux kernel is interrupted regularly by hardware interrupts and the scheduler. The scheduler can decide to preempt the running task and select a different one. These delays and interruptions are not predictable for the task and introduce an indeterminism which prevents real-time execution. The HLRT extension eliminates this indeterminism by allowing a task to run without interruptions.

On a system with n CPUs the HLRT kernel allows $n-1$ CPUs to be reserved. A CPU is reserved by a task when all other tasks are excluded from using this CPU . The task that reserved a CPU is bound to that CPU and can not be migrated to any other CPU in the system . Further, the task can not be interrupted by the execution of any other task since no other task is allowed to be scheduled on that CPU. In addition to the exclusion of other tasks from being executed on a reserved CPU the HLRT extension allows a real-time task to control which interrupts it receives on its reserved CPU.

7.2.1. The `SCHED_HLRT` Scheduling Policy

A new scheduling policy is introduced by the HLRT extension. Each real-time task in the system is scheduled with that policy. If a task switches to the `SCHED_HLRT` policy, the scheduler must determine a currently unreserved CPU and reserve it for the task. The real-time task's CPU affinity mask is set to contain only its reserved CPU. A task can switch to the `SCHED_HLRT` policy by calling the `sched_setscheduler()` system call. The function is extended by the patch and supports the new scheduling policy. The permission checks within `sched_setscheduler()` have been changed to allow the use of `SCHED_HLRT` only if the task has the required capability (i.e. super user) or `CONFIG_HLRT_NOROOT` was enabled in the kernel configuration. All tasks connected with the `SCHED_HLRT` policy are managed by the *fair* scheduling class. We will refer to a task that is scheduled with the `SCHED_HLRT` policy and is managed by the fair scheduler module as a task inside the `SCHED_HLRT` class.

The new global variable `hlrt_cpus_allowed` contains all CPUs that are not currently reserved. A (non real-time) task can run on a CPU if the task CPU affinity and `hlrt_cpus_allowed` contain this CPU. When a CPU is reserved for a task,

[2] `http://www.verified.de`

the scheduler must determine a currently unreserved CPU and reserve it for the task by removing it from `hlrt_cpus_allowed`. As mentioned above, at least one CPU must always be left unreserved for all normal tasks to run on. The two functions listed below are used for administrating the `hlrt_cpus_allowed` mask. Both operations are protected by a spin-lock to avoid having two different tasks reserve the same CPU concurrently.

`hlrt_allocate_next_cpu()` finds a free CPU and deletes it from the mask. The first CPU is always skipped to ensure that at least one CPU is left for normal tasks to continue execution.

`hlrt_release_cpu()` releases a CPU by setting its bit in the `hlrt_cpus_allowed` mask.

A task is a real-time task, if it is scheduled with the `SCHED_HLRT` policy. Real-time tasks are not allowed to migrate between reserved CPUs. Joining the `SCHED_HLRT` policy is only possible with a successful call to `hlrt_allocate_next_cpu()`. The task is then bound to the reserved CPU.

Before running a real-time task on its reserved CPU it must be ensured that all other tasks have left this CPU. This is done during the CPU reservation process. Listing 7.1 shows how a task is added to the `SCHED_HLRT` scheduling class.

sched.c

```
static int hlrt_set_hlrt_policy(struct task_struct *p)
{
...
        cpu = hlrt_allocate_next_cpu();
        if (cpu < NR_CPUS) {
...
                /* change scheduling policy */
...
                __setscheduler(rq, p, SCHED_HLRT, 0);
...
                /* move process to its reserved CPU */
                set_cpus_allowed(p, cpumask_of_cpu(cpu));

                /*
                 * kick all other processes from the reserved
                 * CPU's runqueue
                 */
                migrate_live_tasks(cpu);

                /* move away pending timers */
                hlrt_move_timers(smp_processor_id(),
                                 first_cpu(cpu_possible_map));
                hlrt_move_hrtimers(smp_processor_id(),
                                   first_cpu(cpu_possible_map));

                /* move pending tasklets */
                hlrt_move_tasklets(smp_processor_id(),
                                   first_cpu(cpu_possible_map));
```

```
                /* move block done softirq list */
                hlrt_move_blk_done(smp_processor_id(),
                                   first_cpu(cpu_possible_map));

                /* move pending RCU callbacks */
                hlrt_move_rcu_callbacks(smp_processor_id(),
                                        first_cpu(cpu_possible_map));
...
                return 0;
        }
        return -EBUSY;
}

static int __sched_setscheduler(struct task_struct *p, int policy,
                                struct sched_param *param, bool user)
{
...
#ifdef CONFIG_HLRT
        if (policy == SCHED_HLRT) {
                return hlrt_set_hlrt_policy(p);
        }
...
#endif
...
}
```

Listing 7.1: Add a task to `SCHED_HLRT`

HLRT makes use of the `migrate_live_tasks()` function to migrate tasks away from the newly reserved CPU. `migrate_live_tasks()` is part of the Linux CPU hotplug mechanism. The function runs through the CPU run-queue of assigned tasks and migrates them (all but not the current running task) to another CPU. The HLRT extension ensures that the new assigned CPU for a migrated task is not a reserved CPU. It is actually the first CPU in the task CPU affinity mask or, if no valid CPU can be found in that mask, the first CPU in the system (which is never reserved).

Non real-time tasks bound to a reserved CPU must be treated in a special way. If the CPU of a bound task becomes reserved, the task cannot be executed. It should then be allowed to run on another unreserved CPU. A normal process is just released from that CPU and executed on any other CPU. This, however, can be dangerous if the task is a kernel thread that makes use of CPU bound variables. Special care must be taken to avoid concurrency and race conditions. Variables that may be accessed from different CPUs must be protected by locks. The five calls to `hlrt_move_*()` routines in Listing 7.1 handle a set of kernel threads that fall into this category. They will be discussed in Section 7.2.3.

A real-time task can release a CPU that was previously reserved. In this case the task is removed from the `SCHED_HLRT` scheduling class. The new assigned policy is `SCHED_NORMAL`. Further information about releasing a reserved CPU can be found in [Efk05, Sect. 4.1.1.3].

Load balancing in the Linux kernel is a way for a task to change its CPU. The load balancing mechanism is patched by the HLRT extension in order to not violate any CPU reservations. The Linux scheduler pulls and pushes tasks from a overloaded CPU run-queue to a less busy CPU. HLRT ensures that a reserved CPU does not pull tasks from other run-queues and does push tasks from the run-queue of a reserved CPU to another CPU. The function `find_busiest_runqueue()` which is called from `load_balance()` is modified in a way, that CPU not included in the `hlrt_cpus_allowed` mask are ignored by the algorithm. The details of these changes are not considered here. Further information about the HLRT adjustments to the Linux load balancing mechanism can be found in [Efk05, Sect. 4.1.2.4].

It must be assured that the child does not inherit the scheduling policy and the CPU affinity if a real-time task uses the `fork()` or `clone()` system calls. Therefore it is not allowed for the `SCHED_HLRT` scheduling policy to be inherited (Listing 7.2). Normally, a forked task starts its execution on the CPU on which the parent task is running at the time of the fork. In case of an HLRT task, the CPU for the child is determined by taking any CPU from the intersection of the child's `cpus_allowed` mask and the `hlrt_cpus_allowed` mask. If no valid CPU can be found, the first CPU in the system is used.

sched.c

```
void sched_fork(struct task_struct *p, int clone_flags)
{
...
#ifdef CONFIG_HLRT
        /* no inheritance of hard real-time scheduling class */
        if (p->policy == SCHED_HLRT) {
...
                p->policy = SCHED_NORMAL;
...
        }
#endif
...
}
```

Listing 7.2: `SCHED_HLRT` policy is not inherited

When a previously suspended task becomes ready for execution again (waking up), the selected run-queue must not be on a reserved CPU for a non real-time task. A real-time task should wake up and be enqueued in the run-queue of its reserved CPU. The HLRT extension patches the `try_to_wake_up()` function, which is invoked by the various other functions that wake up tasks in the Linux kernel. In case that the waking task is a not a real-time task and shall be executed on the current CPU because it is not in a run-queue, `try_to_wake_up()` is extended to ensure that the current CPU is not reserved. If the waking task is a member of the `SCHED_HLRT` class the function must ensure that the task is enqueued on its assigned CPU. In case that a non real-time task previously ran on a CPU that has now become reserved by another task and is now supposed to be woken on that CPU, it is necessary to find a new CPU for the task.

7.2.2. Interrupt Routing

The HLRT extension allows an real-time task to control which interrupts it receives on its reserved CPU . By default, the interrupt routing for a CPU is not touched when that CPU is reserved for a task. The two system calls `hlrt_request_irq()` and `hlrt_release_irq()` are provided to modify the interrupt routing for a reserved CPU (Listing 7.3).

hlrt.c

```
static inline cpumask_t hlrt_get_irq_affinity(int irq)
{
        return irq_desc[irq].affinity;
}

asmlinkage int sys_hlrt_request_irq(unsigned int irq)
{
...
        affinity = hlrt_get_irq_affinity(irq);
        cpus_complement(reserved_cpus, hlrt_cpus_allowed);
        cpus_and(result, affinity, reserved_cpus);
        if (cpus_empty(result))
                retval =hlrt_set_irq_affinity(irq,current->cpus_allowed);
        else
                retval = -EBUSY;
...
        return retval;
}

asmlinkage int sys_hlrt_release_irq(unsigned int irq)
{
...
        affinity = hlrt_get_irq_affinity(irq);
        cpus_complement(reserved_cpus, hlrt_cpus_allowed);
        cpus_and(result, affinity, reserved_cpus);
        cpu_clear(smp_processor_id(), result);
        if (cpus_empty(result)) {
                cpumask_t tmp = TARGET_CPUS;

                cpus_and(affinity, hlrt_cpus_allowed, tmp);
                retval = hlrt_set_irq_affinity(irq, affinity);
        } else
                retval = -EBUSY;
...
        return retval;
}
```

Listing 7.3: HLRT interrupt routing

At can be seen in Listing 7.3 requesting an interrupt is only allowed if that interrupt is not already bound to another reserved CPU. An interrupt can only be released if was bound to the current CPU before. If an interrupts is released, the routing for the interrupt is set to all non reserved CPUs. `hlrt_set_irq_affinity()` performs a check if the requested interrupt is already routed to a reserved CPU. In this case the

operation fails. If the interrupt can be assigned to the given CPU mask the function invokes `irq_set_affinity()`.

7.2.3. Necessary Adjustments

SoftIRQs are used in the Linux kernel to defer work to a later point in time . The work is bound to a given CPU (the CPU on which the SoftIRQ appears in most cases). A SoftIRQ is a function that gets called some time after it has been activated (raised). It can be executed when returning from a hardware interrupt or by the kernel SoftIRQ thread. Both cases are not applicable to a reserved CPU. Because SoftIRQs are raised on each CPU individually, they must be transferred to the first (unreserved) CPU in case that the raising CPU is reserved . The HLRT extension solves this issue by patching the Linux SoftIRQ mechanism. The changes are not discussed here. A detailed analysis of the HLRT SoftIRQ patch can be found in [Efk05, Sect. 4.1.2.6].

In Listing 7.1 it is shown that during CPU reservation several operations take place after a CPU was deleted from the `hlrt_cpus_allowed` mask and all other tasks have left that CPU. A short overview of necessary adjustments to the Linux kernel for a reserved CPU is given in this section. A detailed description of the `hlrt_move_*()` operations can be found in [Efk05, Sect. 4.1.2].

`hlrt_move_timers()`, `hlrt_move_hrtimers()` Timers (and hrtimers[3]) which have already been activated must be moved away when a CPU becomes reserved. New timers must be activated on a different CPU. This is necessary, because the local APIC timer interrupt may be disabled for a reserved CPU. Timers in the Linux kernel are processed by the interrupt handler of the local APIC timer. Another reason is the fact that real-time tasks should not be interrupted by other code (in this case timer processing).

`hlrt_move_tasklets()` Pending tasklets for the reserved CPU must be moved to another CPU (actually the first CPU). Tasklets are implemented via two SoftIRQs. Since SoftIRQs are also directed to an unreserved CPU, new tasklets must be enqueued on that CPU.

`hlrt_move_blk_done()` Blockoperation done SoftIRQs must be moved away when a CPU becomes reserved. The *block done* SoftIRQ is raised by a driver after a finished block command is enqueued into the block done queue. Any pending command in the done queue must be moved to another CPU. Further, new finished block commands must be directed to that CPU.

`hlrt_move_rcu_callbacks()` Similar to finished block commands, read-copy update (RCU) operations are finished with a callback. The callbacks are processed by the RCU tasklet. The per CPU queue of these pending finished operations must be moved away when a CPU becomes reserved. New RCU callbacks must be enqueued on that CPU.

[3] It was originally planned to support high resolution timers.

The read-copy update[4] (RCU) implementation in the Linux kernel needs each CPU to achieve a quiescent state at times. A quiescent state describes a state where all local references shared to data structures have been lost and no assumptions are made based on their previous contents. For instance, a CPU goes through a quiescent state if a context switch (schedule) takes place. The RCU callbacks are processed in batches. This happens as soon as all CPUs have gone through a quiescent state. Allocated memory is freed in such a callback for the calling CPU. If the timer interrupt is disabled for a reserved CPU and so the scheduler is not invoked regularly, it is not detected that the CPU is in a quiescent state. Additionally, the RCU tasklet (processing the callback) will never be scheduled on a reserved CPU. As a consequence, memory allocated in RCU operations is never freed for that CPU. The HLRT extension signals that a CPU passed through a quiescent state when a process returns from a system call back to user space. The system call handler[5] is patched in order to implement this behavior (Listing 7.4).

entry_32.S

```
sysenter_do_call:
...
        call *sys_call_table(,%eax,4)
...
#ifdef CONFIG_HLRT
...
        call hlrt_quiescent_state
...
#endif
...
```

Listing 7.4: Return from system call

7.3. Time-Triggered Architecture

The HLRT extension provides the periodic execution of tasks with fixed intervals . These intervals can be seen as deadlines. A mechanism for detecting missed deadlines is also included in the extension. A real-time task can be scheduled with fixed but user defined intervals . If a real-time task chooses to activate periodic execution it must call the newly introduced `hlrt_set_periodic_local_timer_irq()` system call to define a period length. Another new system call `hlrt_yield()`[6] is used by the task to wait for the next period (puts the task to sleep) . If the task does not invoke `hlrt_yield()` within a period, a `SIGALRM` signal will be sent to the task. In this case the task has missed its deadline because it is still executing although the next period has started . A task that uses this feature must not use any system calls that might sleep because it would certainly miss its deadline. Furthermore, only tasks that

[4]Detailed information about the read-copy update mechanism can be found in [CRKH05, Chap. 5] and [BC05, Chap. 5].

[5]The HLRT extension uses an additional vector. Both handlers are patched.

[6]Wrapper for `sys_hlrt_wait_for_apic_timer()`.

have a reserved CPU can be allowed to use this feature. Otherwise the timer interrupt would be in use by the Linux kernel (timer, scheduler, ...).

The HLRT extension makes use of the local APIC timer interrupt to realise periodic tasks. A new interrupt vector is provided by HLRT. `hlrt_set_periodic_local_timer_irq()` programs the local APIC hardware in a way that the new vector is used. Upon initialisation of the local APIC at boot time an interrupt gate is added. The handler for the interrupt gate is shown in Listing 7.5. It is called whenever the local APIC timer creates an interrupt.

```
void smp_hlrt_apic_timer_interrupt(struct pt_regs regs)
{
...
        ack_APIC_irq();
        /* use nmi_enter/_exit because irq_exit would execute pending
           SoftIRQs, which we want to avoid */
        nmi_enter();
        q = &__get_cpu_var(hlrt_apic_timer_wait_queue);
        if (waitqueue_active(q)) {
                wake_up(q);
        } else {
                send_sig(SIGALRM, current, 0);
...
        }
        nmi_exit();
}
```

hlrt.c

Listing 7.5: HLRT local APIC interrupt handler

If the timer interrupt fires and the corresponding task can not be found in the wait queue, a `SIGALRM` is sent to the task. If the task is in the wait queue, it has performed a `hlrt_yield()` system call before (Listing 7.6). The wait queue is a per CPU variable. Since only one real-time task per CPU is allowed (and possible) in a HLRT system, the queue has either one or no element. If the task is not in the wait queue, `current` is always the task that missed the deadline.

```
asmlinkage int sys_hlrt_wait_for_apic_timer(void)
{
...
      interruptible_sleep_on(&__get_cpu_var(hlrt_apic_timer_wait_queue));
      return 0;
}
```

hlrt.c

Listing 7.6: Wait for next period

7.4. Real-Time Application Programming

As mentioned in Section 7.1 a real-time task in the HLRT operating system uses a special API. The CPU-reservation and interrupt-redirection functions are described below.

rtthlrt_begin_rt_process(), rtthlrt_end_rt_process() Reserves or releases a CPU for the calling task. `rtthlrt_begin_rt_process()` adds the calling task to the `SCHED_HLRT` scheduling class with a call to `sched_setscheduler()`. On success, the memory is locked by calling `mlockall()` . All interrupts for the reserved CPU are deactivated, including the APIC timer interrupt. When releasing CPU, the interrupts are reactivated for the CPU.

rtthlrt_enable_irq(), rtthlrt_disable_irq() After a task has been given real-time status by calling `rtthlrt_begin_rt_process()`, no interrupt requests are relayed to the reserved CPU any more. If the task still wants to receive interrupt requests, it can enable (or disable) single interrupts. Requested interrupts will be delivered exclusively to the reserved CPU.

rtthlrt_disable_local_timer_irq(), rtthlrt_enable_local_timer_irq() These functions allow the calling task to adjust the local APIC timer interrupt. This interrupt is not handled by `rtthlrt_enable_irq()`.

hlrt_set_periodic_local_timer_irq(), hlrt_yield() The calling real-time task runs on in periodic mode with the specified period. When the period elapses, the task must have performed a `hlrt_yield()` call, from which it will then be woken.

An example HLRT real-time application is given in [Efk05, Chap. 5]. The benchmark tests implementation in Section 7.5 can be taken as example applications for the HLRT extension as well.

7.5. Benchmarking

The benchmark tests described in Section 5.4.3 will be discussed in this section in concrete implementation for the HLRT operating system. The presented values in this section are the results from executing the same tests for a patched and a non-patched kernel[7]. This makes it easy to see the impact of the HLRT extension. In Chapter 12 the benchmark results are compared and evaluated to other real-time operating systems. The tests for the unpatched kernel were already described in Section 6.5. All benchmark results can be accessed at `http://www.informatik.uni-bremen.de/agbs/dirkr/HRTL/benchmark_results.tgz`.

Not all benchmark tests as described in Section 5.4.3 can be applied to the HLRT operating system. The restriction that only one real-time task can run on one CPU makes all task switching tests inapplicable.

7.5.1. Task Period Tests

Listing 7.7 shows the HLRT periodic task benchmark test implementation. The setup function (`start_rt()`) is called early in the main function. As mentioned in

[7]Linux kernel version 2.6.27.19

Section 7.4 the library function `rtthlrt_begin_rt_process()` gives the calling task real-time properties. The memory area for the measurement results (`tsc[]`) is touched before executing the main loop. As described in Section 6.4 this is necessary to cause a stack fault before the test starts.

period_....c

```
int start_rt(void) {
    if (0 > rtthlrt_begin_rt_process()) {
        perror("hlrt_begin_rt_process failed");
        return -1;
    }

    return 0;
}

int main(int argc, char **argv) {
    uint32_t tsc[LOOP_COUNT];
...
    if (0 > start_rt())
        exit(EXIT_FAILURE);

    /* pre-fault stack */
    for (i = 0; i < LOOP_COUNT; i++)
        rdtsc_32(tsc[i]);
...
    /* start local APIC timer in periodic mode */
    hlrt_set_periodic_local_timer_irq(atoi(argv[1]));

    /* benchmark */
    for (i = 0; i < LOOP_COUNT; i++) {
        hlrt_yield();
        rdtscp_32(tsc[i]);
        cpuid();
#ifdef TRIGGER_PARPORT
        parport_toggle();
#else
        busy();
#endif
    }
...
}
```

Listing 7.7: HLRT period task benchmark test

The test makes use of the HLRT time-triggered architecture (Section 7.3) to realise periodic task behavior. The `hlrt_yield()` library call blocks until the programmed period expires. A signal handler function is also implemented and not shown in the listing. Different from the Linux periodic task benchmark test implementation (Section 6.5.1) a `SIGALRM` signal is only sent in case the real-time task misses its deadline.

The first benchmark test for the HLRT extension operating system measures the scheduling precision of a periodic task with a period of 500 μs (Table 7.1). The test was executed in the 3 scenarios described in Section 5.3.

Scenario	Average	Min	Max	Gap	Deviation
Normal	499.936	499.787	500.087	0.300	0.035
CPU utilization	499.939	497.863	502.017	4.154	0.558
I/O utilization	500.101	499.397	504.599	5.202	0.837

Table 7.1.: Benchmark test results [μs]: HLRT period task (500μs)

Like for the RT-Preempt extension the results in the table were converted to the μs scala. The translation caused some inaccuracy in the precision of the values due to rounding errors. The test is not applicable for an unpatched Linux 2.6.27.19 kernel. Timers smaller than the Linux scheduling interval (10 ms in this case) are only provided in the Linux kernel with the use of high resolution timers. As described in Section 7.1 the HLRT extension does not support this feature. For the comparability of (all) HLRT benchmark results, the configuration was not adjusted for the unpatched Linux kernel.

Table 7.2 and Table 7.3 show the results of the periodic benchmark test with a period of 10 ms. The results are also shown in **??**.

Scenario	Average	Min	Max	Gap	Deviation
Normal	9999.051	9998.762	10003.404	4.642	0.478
CPU utilization	9999.007	9995.378	10004.722	9.343	1.753
I/O utilization	10002.072	9997.962	10011.605	13.643	3.265

Table 7.2.: Benchmark test results [μs]: HLRT period task (10ms)

Scenario	Average	Min	Max	Gap	Deviation
Normal	9998.832	9915.862	10086.177	170.315	10.598
CPU utilization	9998.833	9781.798	10226.955	445.157	44.156
I/O utilization	10002.902	9916.873	10088.423	171.550	13.792

Table 7.3.: Benchmark test results [μs]: Linux 2.6.27 period task (10ms)

The same test is repeated with a 100 ms (Table 7.4 and Table 7.5) and a 1 second period (Table 7.6 and Table 7.7). Nevertheless, the test is only executed in the normal scenario.

Scenario	Average	Min	Max	Gap	Deviation
Normal	99990.496	99989.767	99994.561	4.794	1.314

Table 7.4.: Benchmark test results [μs]: HLRT period task (100ms)

Scenario	Average	Min	Max	Gap	Deviation
Normal	99988.119	99986.510	99993.076	6.567	1.320

Table 7.5.: Benchmark test results [μs]: Linux 2.6.27 period task (100ms)

Scenario	Average	Min	Max	Gap	Deviation
Normal	999905.152	999904.888	999909.213	4.324	0.682

Table 7.6.: Benchmark test results [μs]: HLRT period task (1sec)

Scenario	Average	Min	Max	Gap	Deviation
Normal	999881.393	999879.947	999886.276	6.329	1.169

Table 7.7.: Benchmark test results [μs]: Linux 2.6.27 period task (1sec)

7.5.2. Interrupt Tests

The implementation of the interrupt benchmark tests as described in Section 5.4.3.4 are explained in this section. Since the HLRT extension allows only one real-time task per CPU, the interrupt to task latency benchmark test is not applicable for HLRT. For the realisation of the tests, it is necessary to enhance the kernel with a module. This module implements the interrupt handler for the parallel port interrupt. Like for a normal Linux kernel (Listing 6.12) the interrupt handler just captures the current value of the TSC register and returns. Values between the main benchmark test and the measurements inside the interrupt handler are transmitted via the proc interface. Listing 7.8 shows the implementation of the interrupt latency benchmark test main loop. The interrupt is triggered with the benchmark framework functions `parport_low()` and `parport_high()`.

interrupt_isr.c

```
int main(void) {
...
    if (0 > start_rt())
        exit(EXIT_FAILURE);

    parport_init_irq(-1);
...
    for (i = 0; i < LOOP_COUNT; i++) {
        busy();
        parport_low();
        busy();
        cpuid();
        rdtsc_32(res.start[i]);
        parport_high();

        do {
...
            fscanf(fh, "%s\n", tmp);
            res.stop[i] = strtoll(tmp, NULL, 10);
```

```
        } while (old == res.stop[i]);

        old = res.stop[i];
    }
...
}
```

bench...linux.h

```
static inline void parport_init_irq(int cpu) {
...
#ifdef BENCHMARK_HLRT
    cpu = cpu;
    rtthlrt_enable_irq(PARPORT_IRQ);
#endif
}
```

Listing 7.8: HLRT interrupt latency benchmark test

The results of the benchmark test are shown in Table 7.8 and Table 7.9.

Scenario	Average	Min	Max	Gap	Deviation
Normal	4.610	3.245	6.005	2.760	0.690
CPU utilization	5.383	3.395	7.089	3.694	0.782
I/O utilization	4.625	3.274	5.968	2.694	0.690

Table 7.8.: Benchmark test results [μs]: HLRT interrupt latency (ISR)

Scenario	Average	Min	Max	Gap	Deviation
Normal	4.785	3.350	6.106	2.757	0.695
CPU utilization	5.284	3.350	11.919	8.569	0.928
I/O utilization	4.814	3.372	6.741	3.368	0.702

Table 7.9.: Benchmark test results [μs]: Linux 2.6.27 interrupt latency (ISR)

The interrupt dispatch latency benchmark test is similar to the interrupt latency benchmark test except of the time measurement points. For this test the first value is captured within the kernel. The second time value is gathered when returning from interrupt. Listing 7.9 illustrates the interrupt dispatch latency benchmark test.

int...dispatch.c

```
int main(void) {
...
    for (i = 0; i < LOOP_COUNT; i++) {
        busy();
        parport_low();
        busy();
        parport_high();

        do {
            rewind(fh);
```

```
            fscanf(fh, "%s\n", tmp);
            res.start[i] = strtoll(tmp, NULL, 10);
        } while (old == res.start[i]);

        rdtscp_32(res.stop[i]);
        cpuid();
        old = res.start[i];
    }
...
}
```

Listing 7.9: HLRT interrupt dispatch latency benchmark test

The test is similar to the RT-Preempt interrupt dispatch latency benchmark test (Listing 6.15). The same problem with the slight delay when triggering the interrupt can be found in this scenario too. The test will be interrupted within the loop, but an additional comparison for leaving the `while` loop takes place before finishing the measurement.

The results of the interrupt dispatch latency benchmark test are provided in Table 7.10 and Table 7.11.

Scenario	**Average**	**Min**	**Max**	**Gap**	**Deviation**
Normal	1.387	0.810	1.930	1.120	0.317
CPU utilization	5.861	0.843	13.197	12.354	2.911
I/O utilization	1.415	0.831	1.950	1.118	0.322

Table 7.10.: Benchmark test results [μs]: HLRT interrupt latency (dispatch)

Scenario	**Average**	**Min**	**Max**	**Gap**	**Deviation**
Normal	1.267	0.721	1.770	1.048	0.301
CPU utilization	4.495	0.717	10.435	9.717	2.474
I/O utilization	1.291	0.718	2.243	1.524	0.320

Table 7.11.: Benchmark test results [μs]: Linux 2.6.27 interrupt latency (dispatch)

7.6. Summary

The HLRT patch achieves real-time behavior by isolating each real-time task on an exclusevly assigned CPU. There is no need to introduce extra preemption points since no other task can interrupt a running real-time task.

In Chapter 11 we will see how the basic idea of CPU reservation and time-triggerd architecture are used to build a task partitioning system together wither other approaches from the next chapter. The concepts of threaded interrupt handlers (Section 6.3)

together with the interrupt routing introduced in this chapter will be modified to fit into our partitioning system.

8 Case Study 3: QNX Neutrino

QNX is one of the most widely used real-time operating systems. In this chapter we will discuss some of the main features of the QNX kernel and analyse technical aspects. Section 8.4 explains the implementation of the benchmark tests introduced in Section 5.4.3 for the QNX Neutrino operating system.

8.1. Background and Overview

Neutrino is the evolution of the operating system formerly known as QUNIX (abbreviated QNX). To meet the increasing market orientation on POSIX models, the kernel of the QUNIX operating system was completely redeveloped. The result of these efforts is the inherently POSIX compliant and SMP compatible real-time operating system Neutrino. Typically, the system comes with an integrated graphical user interface (Photon microGUI), a development environment based on Eclipse, various GNU tools and internet software (Mozilla). Neutrino has been ported to many architectures and now runs on almost any modern CPU. It is primarily aimed at the embedded market.

In Neutrino the QNX Momentics Development Suite is included. Neutrino is currently not available without this commercial development environment. Momentics allows the development of Neutrino applications under Linux, Windows, Solaris and existing Neutrino based systems.

Neutrino is one of the most popular real-time operating systems available for critical tasks. It has a Unix-like structure and is fully POSIX compliant. With an API wrapper, it also supports software for Linux. Its strengths lie in the easy development of software, control of industrial robots and micro-computers. A variety of hardware is supported by Neutrino. However, support for PCIe has not been made available yet.

8.2. Microkernel Architecture

The Neutrino microkernel provides some minimal services used by a set of cooperating tasks, which in turn provide the higher-level OS functionality. Services like filesystems and partitioning are provided by optional tasks. The kernel itself is dedicated to a

few fundamental services:[1]

Message-passing services handle the routing of all messages between all tasks (threads) throughout the entire system. POSIX signal primitives are also included in this service.

Synchronization services handle in addition to message passing POSIX conform thread-synchronization primitives.

Timer services provide a set of POSIX timer services.

Process management provides scheduling of tasks for execution using the various POSIX realtime scheduling algorithms (First-In-First-Out and Round-Robin). The process manager portion is responsible for managing processes, memory, and the *pathname space*.

Low-level interrupt handling receives all hardware interrupts and faults, then passes them on to the appropriate driver task.

All operating system services, except those provided by the mandatory microkernel, are handled via standard processes. Drivers and applications are isolated to their own memory address, and operate independently. In a multicore system, QNX allows only one thread at a time to enter the kernel. The kernel has a *restart* model for kernel call preemption. A task inside a kernel call can be preempted by a higher priority task. If the lower priority task is executed again, the first thing it will do is to re-execute the system call instruction. This will restart the kernel call that was being worked on when it was interrupted in the first place.

This section gives an overview of the main QNX Neutrino microkernel services. For more information on QNX, see [Sys05, Chap. 2] and [Hil92].

8.2.1. Process Management

Process management in Neutrino is split into two components. The scheduling of tasks according to the First-In-First-Out and Round-Robin policies is handled directly by the kernel. Creation and destruction of tasks is done in a single module called `procnto`. This module is required for all runtime systems. `procnto` runs as a true process and is scheduled to run by the kernel like all other processes. It is the only process which shares the same address space as the kernel. Communication with other processes takes place via the kernel's message passing primitives (Section 8.2.3). `procnto` is responsible for creating new processes in the system and managing the most fundamental resources associated with a process. These services are all provided via messages.[2]

[1] QNX can be driven as a distributed system. The related network services are not listed here.

[2] Since messages are network-wide, it is easy to create a process on another node by sending the process-creation message to the instance of `procnto` running on that node.

`procnto` is capable of creating multiple POSIX processes, each of which may contain multiple POSIX threads. Like Linux systems, QNX supports some process creation primitives (`fork()`, `exec()` and `spawn()`). Both `fork()` and `exec()` are defined by POSIX, while the implementation of `spawn()` is unique to QNX. It can create processes on any node in the network. When a process is created by one of the three primitives, it inherits much of its environment from its parent. A process goes through four phases:

Creation of a process consists of allocating a process ID for the new process. The information that defines the environment of the new process is initialized. Most of it is inherited from the parent process.

Loading of a process image. This is done by a separate *loader thread*. Actually, the loader thread is the newly created task in an early state. The loader code resides in the `procnto` module, but the thread runs under the process ID of the new process.

Execution of the new process. All processes run concurrently with their parents. In addition, the termination of a parent process does not automatically cause the termination of its child processes.

Termination in either of two ways: a signal whose defined action is to cause process termination is delivered to the process or the process invokes `exit()`.

Similar to Linux systems, every task in Neutrino is assigned a priority level . The scheduler selects the next task to run by looking at the priority assigned to every task that is ready for execution. The task with the highest priority is selected to run. The default priority for a new process is inherited from its parent. The priority level can be adjusted at runtime with the two system calls `getprio()` and `setprio()`. Although a task inherits its scheduling class from its parent process. It can be changed with the two system calls `getscheduler()` and `setscheduler()`. `procnto` has no priority assigned. It listens on a channel for incoming messages. When `procnto` receives a message it *floats* to the same priority as the client thread.

The POSIX semantics for device and file access is presented in QNX by the so called *pathname space*. Managing the pathname space is part of the `procnto` module. The details of the pathname space implementation can be found at [Sys05, Chap. 5]. QNX encourages the development of applications that are split up into cooperating processes. Processes can register names within the pathname space. Other processes can then ask the process manager for the process ID associated with that name.

The process manager is responsible for memory management. On task creation, the loader thread starts in a new virtual memory space. The process manager will then take over this environment, changing the mapping tables as required by the processes it starts. POSIX memory locking is supported by QNX, so that a process can avoid the latency of fetching a page of memory, by locking the memory.

8.2.2. Interrupt Handling

User-level processes can attach (and detach) hardware interrupt handlers to (and from) interrupt vectors at runtime. When the hardware interrupt occurs, the processor will enter the interrupt redirector in the microkernel. Here the processor context is set so that the handler has access to the code and data that are part of the thread the handler is contained within. This allows the handler to use the buffers and code in the user-level thread. If higher-level work is required, QNX provides an API for queuing events to the thread the handler belongs to . Since interrupt handler run with the memory-mapping of the thread containing it, the handler can directly manipulate devices mapped into the thread's address space, or directly perform I/O instructions. As a result, device drivers that manipulate hardware don't need to be linked into the kernel.

The interrupt redirector code in the microkernel will call each interrupt handler attached to that hardware interrupt. If the handler indicates that an event has to be passed to a process, the kernel will queue the event. When the last handler has been called for that vector, the kernel interrupt handler will finish manipulating the interrupt control hardware. If a queued event causes a higher-priority thread to become ready for execution, the interrupt return will switch into the context of the higher-priority thread.

While the interrupt handler is executing, it has full hardware access, but cannot invoke other kernel calls. If necessary, the handler can cause a thread to be scheduled at some user-specified priority to do further work.

Example programs that use the QNX interrupt handling can be found in Section 8.4.4.

8.2.3. Message Passing

Message passing is the primary form of inter process communication (IPC) in QNX Neutrino. Other forms of IPC are built over the native message passing. In a multicore system, QNX uses interprocessor interrupts (IPI) for inter processor communication.

Messages are passed via communication channels. The `ChannelCreate()` system call creates a channel that can be used to receive messages. Once created, the channel is owned by the process and is not bound to the creating thread. To establish a connection between a process and a channel, the system call `ConnectAttach()` is used. In addition, the system calls `name_attach()` and `name_open()` can be used in order to create or open a channel and associate a name with it. Communication over channels is synchronized . It always follows the *send* - *receive* - *reply* scheme. A reader task is blocked until it receives a message. The sending task then is suspended until the receiving task sends a reply message.

To dequeue and read messages from a channel, the `MsgReceive()` system call is used. On the other side, the `MsgSend()` system call is used to enqueue messages on the channel. Messages are enqueued in priority order. An example application that uses message passing can be seen in Section 8.4.2.

In QNX, a special message type exists that can be passed via communication channels. A *pulses* is a tiny message that can carry only 40 bits of payload. The most important difference compared to normal messages is, that sending a pulse is non-blocking for the sender. Receiving a pulse is done by either system call `MsgReceive()` which has already been introduced, or the `MsgReceivePulse()` system call. `MsgReceivePulse()` will receive only pulses and ignores normal messages (they stay in the queue).

As mentioned above in this section, IPC mechanisms are built on top of message passing in QNX. The microkernel delivers events for various system states via pulse/channel communication. A task can register to receive an event as a timer . In Section 8.4.1 it can be seen how pulses can be used as timers.

8.3. Adaptive Partitioning Scheduler

As a microkernel operating system Neutrino is based on the strategy to run the bulk of the system in the form of processes. Here, each process runs in a separate and protected memory area. The system includes several segments called partitions in which processes are trapped . Figure 8.1 illustrates the partitioning mechanism of Neutrino. System services such as hardware drivers run in the system partition. This significantly increases the stability of the kernel . A faulty device driver can not affect the overall system. Developers and integrators can isolate downloadable content to the secure downloadable partition. Malicious or unvalidated software that monopolizes system resources can not starve critical functions of CPU time. Unlike fixed partitioning mechanisms, the adaptive partitioning of Neutrino ensures that spare CPU capacity is used when it is available.

The Neutrino kernel (like primarily all microkernel architectures) includes only the most fundamental services, such as signals, timers and schedulers. All other components run as processes in the system partition. The components and kernel communicate via a single messaging layer (linking interface) as described in Section 8.2.3. This virtual software bus permits the addition and removal of software components at any time.

Adaptive partitions are a concept in QNX to separate applications into dynamic sets (partitions). Each partition has a budget of CPU time allocated to it that guarantees its minimum share of the CPU's resources . A partition's unused budget is distributed among partitions that require extra resources when the system isn't loaded. Partitions can be dynamically added and configured at system run time. Also, threads can be added to and deleted from a partition at run time. Child threads automatically run in the same partition as their parent. However, adaptive partitions can not be dynamically deleted. Once created, a partition can only be removed by a system restart.[3] The following properties describe an adaptive partition:

`name` The name of the partition.

[3] Provided that the partition is not included in the system specification.

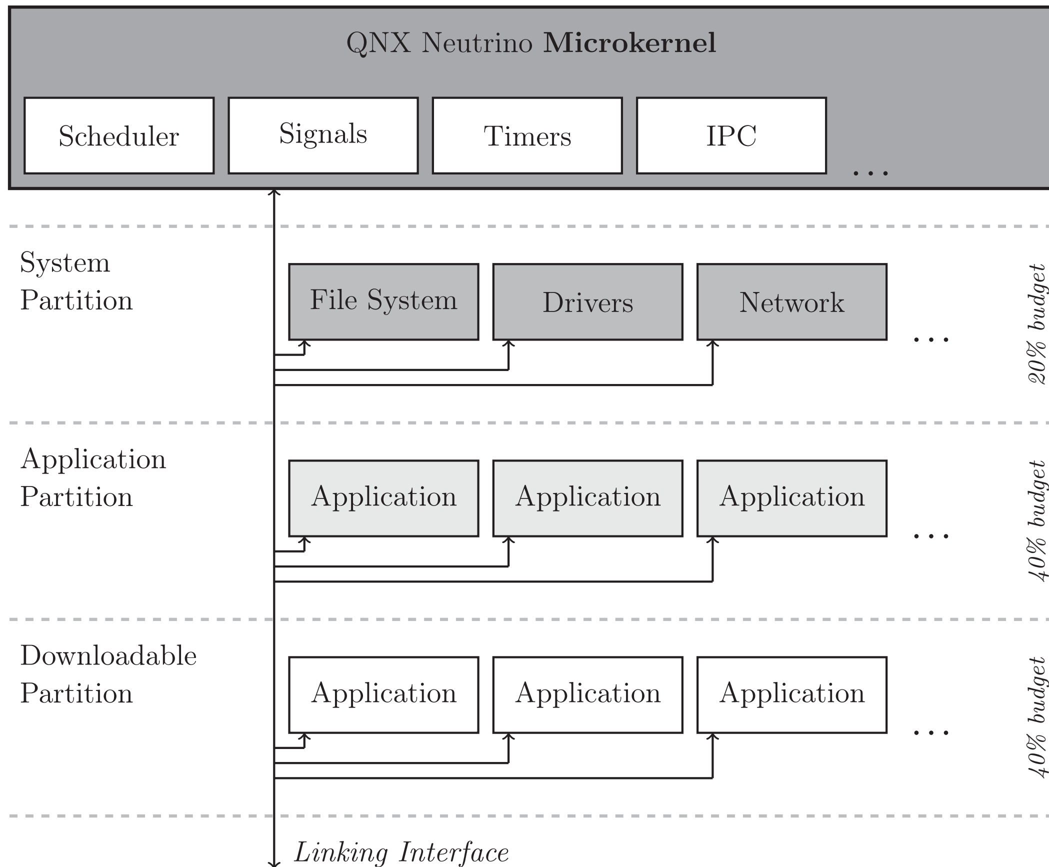

Figure 8.1.: QNX Neutrino adaptive partitioning

`budget_percent` The percentage CPU budget for the partition. Budgets given to a new partition are subtracted from the parent partition.

`critical_budget_ms` The critical budget for the partition.

`id` The partition's ID number (chosen by the system).

Adaptive partitions are handled by the adaptive partitioning scheduler (APS). APS is a thread scheduler that is available as an optional module for the process manager (Section 8.2.1). On system start, an initial partition is created (system partition). The System partition initially has a CPU budget of 100%. When creating a new partition, its budget is taken from its parent partition's budget; which is usually the system partition. The following properties describe the overall parameters of the adaptive partitioning scheduler:

`cycles_per_ms` The number of machine cycles in a millisecond. This is a constant given by the system.

`windowsize_cycles` The length, in CPU cycles, of the averaging window used for scheduling. By default, this corresponds to 100 ms.

windowsize2_cycles The length of window 2. Typically 10 times the window size.

windowsize3_cycles The length of window 3. Typically 100 times the window size.

scheduling_policy_flags These flags set options for the adaptive partitioning scheduling algorithm.

sec_flags Defines the security options of APS.

bankruptcy_policy Defines how the APS behaves if a partition exhausts its critical budget.

num_partitions The number of partitions defined.

max_partitions The largest number of partitions that may be created at any time. This is a system constant and is set to eight.

The adaptive partitioning scheduler throttles CPU usage by measuring the average CPU usage of each partition. The average is computed over the averaging window (`windowsize_cycles`). The window size defines the time over which APS balances the partitions to their CPU limits. The window moves forward as time advances. The two additional windows `windowsize2_cycles` and `windowsize3_cycles` allow to keep statistics over a longer period. They are meaningless for partition balancing.

APS guarantees that partitions receive a defined minimum amount of CPU time. It also supports task priority levels and preempts a lower prior task if a higher priority task becomes ready. Both requirements can be satisfied as long as there is no need to limit a partition in order to guarantee some other partition's budget. The following three scenarios can be distinguished:

Underload Partitions are demanding less CPU time than their defined budgets allow. APS chooses between them by picking the highest-priority running thread.

Idle time One or more partitions are not running. APS then gives that partition's time to other running partitions. If the other running partitions demand enough time, they're allowed to run over budget.

Overload All partitions are demanding their full budget. In this case, the requirement to meet the partitions' guaranteed budgets takes precedence over priority.

APS allows a task to run even if its partition is over budget. `critical_budget_ms` must be defined and available for that partition. A *critical task* is automatically identified if it was initiated by an I/O interrupt. However, the `SchedCtl()` system call can be used to mark a thread as critical. A critical thread is allowed to violate the budget rules of its partition and run immediately. A thread that receives a message from a critical thread automatically becomes critical as well. The critical time budget

(critical_budget_ms) is specified in milliseconds. It's the amount of time all critical threads may use during an averaging window. A critical thread will run as long as its partition still has critical budget. The critical budget of a partition is not billed if the partition still has CPU budget or no other partition is competing for CPU time. The critical threads run whether or not the time is billed as critical. The only time critical threads would not run is when their partition has exhausted its critical budget. The number of critical threads in the system must be few or APS will not be able to guarantee all partitions' their minimum CPU budgets.

If the critical CPU time billed to a partition exceeds its critical budget the previously mentioned bankruptcy_policy becomes important. A so called *bankruptcy* is always considered to be a design error on the part of the application. QNX provides several strategies to handle bankruptcy. It reaches from the default behaviour where it is not allowed for a partition to run again until it gets more budget to a forced system reboot.

Threads can be added to and removed from partitions at runtime using the SchedCtl() function .

8.4. Benchmarking

The benchmark tests described in Section 5.4.3 are going to be discussed in this section in concrete implementation for the QNX Neutrino operating system. In Chapter 12 the benchmark results are compared and evaluated to other real-time operating systems. All benchmark results can be accessed at http://www.informatik.uni-bremen.de/agbs/dirkr/HRTL/benchmark_results.tgz.

Listing 8.1 shows the system wide setting for the benchmark testing. This stand alone application is executed before the testing starts. After adjusting the period of the timer interrupt, one partition is created that can use up to 100% of the available CPU time. All tests are executed in this partition.

setup.c

```
int main(void) {
...
    /* set the clock period to 100 us */
    if (0 > ClockPeriod(CLOCK_REALTIME, &clkper, NULL, 0)) {
...
    memset(&creation_data, 0, sizeof(creation_data));
    creation_data.budget_percent = BUDGET_PERCENT;
    creation_data.critical_budget_ms = CRITICAL_MS;
    creation_data.name = PARTITION_NAME;

    /* create adaptive partition */
    ret = SchedCtl(SCHED_APS_CREATE_PARTITION, &creation_data,
                   sizeof(creation_data));
...
}
```

Listing 8.1: QNX benchmark test system wide settings

Listing 8.2 shows the main setup for a benchmark process in the Neutrino operating system. Each task calls this functions before executing any measurements. As mentioned in Section 5.4.1 a task has to arrange some sort of affinity to a certain CPU in order to keep the TSC register values comparable. Further, the real-time task determines its scheduling policy to be `SCHED_FIFO` and locks its memory by calling `mlockall()` .

bench...qnx.h

```
static inline int setup_rt(int prio) {
...
    param.sched_priority = prio;
    if (0 > sched_setscheduler(0, SCHED_FIFO, &param)) {
...
    if (0 > mlockall(MCL_CURRENT | MCL_FUTURE)) {
...
    APS_INIT_DATA(&lookup_data);
    lookup_data.name = PARTITION_NAME;
    ret = SchedCtl(SCHED_APS_LOOKUP, &lookup_data, sizeof(lookup_data));
    if (0 > ret) {
...
    APS_INIT_DATA(&join_data);
...
    ret =SchedCtl(SCHED_APS_JOIN_PARTITION,&join_data,sizeof(join_data));
    if (0 > ret) {
...
}

static inline int set_affinity(int cpu) {
...
    rmaskp = (unsigned *)(&(((int *)my_data)[1]));
    imaskp = rmaskp + num_elements;
    RMSK_SET(cpu, rmaskp);
    RMSK_SET(cpu, imaskp);

    retval = ThreadCtl(_NTO_TCTL_RUNMASK_GET_AND_SET_INHERIT, my_data);
...
}
```

Listing 8.2: QNX benchmark test setup

8.4.1. Task Period Tests

The periodic task benchmark test is presented in Listing 8.3. As can be seen, the setup functions (Listing 8.2) are called early in the main function. The memory area for the measurement results (`tsc[]`) is touched before executing the main loop. As described in Section 6.4 this is necessary to cause a stack fault before the test starts.

period_....c

```
int main(int argc, char **argv) {
    uint32_t tsc[LOOP_COUNT];
...
    if (0 > setup_rt(RT_PRIO))
        exit(EXIT_FAILURE);
```

```
    if (0 > set_affinity(BENCHMARK_CPU))
        exit(EXIT_FAILURE);
...
    /* pre-fault stack */
    for (i = 0; i < LOOP_COUNT; i++)
        rdtsc_32(tsc[i]);

    if (0 > (chid = ChannelCreate(0))) {
...
    if (0 > (coid = ConnectAttach(ND_LOCAL_NODE, 0, chid, 0, 0))) {
...
    SIGEV_PULSE_INIT(&event, coid,
                     SIGEV_PULSE_PRIO_INHERIT,
                     _PULSE_CODE_MINAVAIL,
                     (void*)pulse_id);

    if (0 > (timer_id = start_timer(&timer, &event)))
...
    for (i = 0; i < LOOP_COUNT; i++) {
        MsgReceivePulse(chid, &pulse, sizeof(pulse), NULL);
        rdtscp_32(tsc[i]);
        cpuid();
#ifdef TRIGGER_PARPORT
        parport_toggle();
#else
        busy();
#endif
    }
...
}
```

bench...qnx.h

```
static inline int start_timer(struct timespec *time,struct sigevent *e){
...
    retval = timer_create(CLOCK_REALTIME, e, &timer_id);
...
    timer.it_value.tv_sec = time->tv_sec;
    timer.it_value.tv_nsec = time->tv_nsec;
    timer.it_interval.tv_sec = time->tv_sec;
    timer.it_interval.tv_nsec = time->tv_nsec;

    retval = timer_settime(timer_id, 0, &timer, NULL);
...
    return timer_id;
}
```

Listing 8.3: QNX period task benchmark test

Periodic task behavior is realised by using an interval timer, programmed with `timer_settime()`. Upon expiration of the programmed timer, a pulse is sent to the benchmark task. The appropriated `MsgReceivePulse()` call in the main loop blocks until the pulse is received by the process. The pulse indicates the start of a new period.

The first benchmark test for the QNX Neutrino operating system measures the scheduling precision of a periodic task with a timer of 500 μs (Table 8.1). The test was executed in the 3 scenarios described in Section 5.3.

Scenario	Average	Min	Max	Gap	Deviation
Normal	499.807	493.048	599.774	106.726	10.966
CPU utilization	499.817	496.591	599.546	102.955	10.895
I/O utilization	499.808	465.293	598.909	133.615	11.523

Table 8.1.: Benchmark test results [μs]: QNX period task ($500\mu s$)

Table 8.2 shows the results of the periodic benchmark test with a 20 times larger timer. In the CPU and I/O utilization scenarios Neutrino system loses some accuracy with a 10 ms timer.

Scenario	Average	Min	Max	Gap	Deviation
Normal	9999.100	9969.021	10075.079	106.058	44.213
CPU utilization	9999.107	9970.339	10074.667	104.328	44.255
I/O utilization	9999.108	9950.242	10085.792	135.551	44.161

Table 8.2.: Benchmark test results [μs]: QNX period task ($10ms$)

The same test is repeated with a 100 ms (Table 8.3) and a 1 second timer (Table 8.4). Nevertheless, the test is only executed in the normal scenario.

Scenario	Average	Min	Max	Gap	Deviation
Normal	99990.666	99919.991	100025.977	105.986	46.676

Table 8.3.: Benchmark test results [μs]: QNX period task ($100ms$)

Scenario	Average	Min	Max	Gap	Deviation
Normal	999907.835	999827.273	999934.190	106.916	41.930

Table 8.4.: Benchmark test results [μs]: QNX period task ($1sec$)

8.4.2. Task Switch Tests

As described in Section 5.2.1 two different sorts of tests for measuring task switch latency are implemented (task preemption time and task switch time). The task preemption time is measured in two variants. Listing 8.4 shows the implementation of the startup routine for the task preemption latency benchmark test.

switch_...ev

```
int main(void) {
...
    if (0 > setup_rt(RT_PRIO +1))
        exit(EXIT_FAILURE);

    if (0 > set_affinity(BENCHMARK_CPU))
        exit(EXIT_FAILURE);
...
    /* register a name in the namespace and create a channel */
    if ((attach = name_attach(NULL, MY_SERV, 0)) == NULL) {
...
    if (0 > (coid = ConnectAttach(0, 0, attach->chid, _NTO_SIDE_CHANNEL,
         0))) {
...
    if (0 == fork()) {
        if (0 > set_affinity(BENCHMARK_CPU))
            exit(EXIT_FAILURE);

        if (0 > setup_rt(RT_PRIO))
            exit(EXIT_FAILURE);

        task_high(1, res);
...
    } else
        t1 = MsgReceive(attach->chid, NULL, 0, NULL);

    if (0 == fork()) {
        if (0 > set_affinity(BENCHMARK_CPU))
            exit(EXIT_FAILURE);

        if (0 > setup_rt(RT_PRIO -1))
            exit(EXIT_FAILURE);

        task_low(0, res);
...
    } else
        t0 = MsgReceive(attach->chid, NULL, 0, NULL);

    MsgReply(t1, 0, NULL, 0);
    MsgReply(t0, 0, NULL, 0);
    sched_setparam(0, &schedp);
...
}
```

Listing 8.4: QNX task preemption benchmark test startup

Two channels are used as events for synchronising the start of the test. Since, actually three different processes are involved in test executing, the results are stored in a shared memory segment. The initialisation of the shared memory segment is not shown in the listing. Two processes are forked during test startup. As one can see, each created process calls the previously introduced setup functions (Listing 8.2) at first. The process related startup routines are shown later in this section. A channel

(t0 or t1) is used here to let the main process block until the new created process finishes its own setup phase. The created process sends a message on that channel and waits for a reply. After both processes have finished their startup the main process send a reply on both channels and lowers its priority level. Both forked processes have higher priority than the main process now. If they terminate, the main process comes back to life and finishes the benchmark test by printing the results.

The main test takes place between the new created processes.[4] As mentioned above, the main test is implemented in two different variants. One uses the POSIX signal mechanism for triggering the higher priority task. The implementation is not shown here, since it is similar to the task preemption latency benchmark test for the Linux operating system (see Listing 6.8). The other version of the test uses a QNX channel for triggering the higher priority task. Listing 8.5 shows the main routines for both tasks. After performing the startup synchronisation as mentioned above (`MsgSend()`), the benchmark test starts with entering the `for` loop. `task_low` writes to a channel (`srv_coid`) for waking up the higher priority task `task_high` which was previously blocked on reading from that channel (`MsgReceive()`).

```
void task_low(int idx, struct tsc_tab *res) {
...
    for (i = 0; i < LOOP_COUNT; i++)
        rdtsc_32(tsc[i]);

    MsgSend(srv_coid, NULL, 0, NULL, 0);
...
    for (i = 0; i < LOOP_COUNT; i++) {
        busy_long();
        cpuid();
        rdtsc_32(tsc[i]);
        MsgSend(task_coid, NULL, 0, NULL, 0);
    }

    for (i = 0; i < LOOP_COUNT; i++)
        res->tsc[idx][i] = tsc[i];
...
}

void task_high(int idx, struct tsc_tab *res) {
...
    for (i = 0; i < LOOP_COUNT; i++)
        rdtsc_32(tsc[i]);

    MsgSend(srv_coid, NULL, 0, NULL, 0);

    for (i = 0; i < LOOP_COUNT; i++) {
        busy_long();
        m = MsgReceive(attach->chid, NULL, 0, NULL);
        rdtscp_32(tsc[i]);
        cpuid();
```

switch_...event.c

[4]See Section 5.4.3.2 for further explanation.

```
        MsgReply(m, 0, NULL, 0);
    }

    for (i = 0; i < LOOP_COUNT; i++)
        res->tsc[idx][i] = tsc[i];
...
}
```

Listing 8.5: QNX task test preemption benchmark test

The benchmark test was executed in the 3 scenarios described in Section 5.3. Table 8.5 and Table 8.6 show the results of the test. As one can see the QNX implementation of the POSIX signaling mechanism has an impact on the execution time. Using a QNX channel for triggering the higher priority task is faster by a factor of about 2.5.

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.419	0.401	1.858	1.457	0.128
CPU utilization	0.456	0.416	3.944	3.528	0.231
I/O utilization	1.461	0.434	6.783	6.349	0.591

Table 8.5.: Benchmark test results [μs]: QNX preempt task (event)

Scenario	Average	Min	Max	Gap	Deviation
Normal	1.004	0.946	3.645	2.700	0.316
CPU utilization	1.030	0.946	4.851	3.905	0.384
I/O utilization	3.023	1.056	8.169	7.113	0.850

Table 8.6.: Benchmark test results [μs]: QNX preempt task (signal)

The second benchmark test for measuring the task switch latency is also described in Section 5.4.3.2. As explained in Section 6.5.2 the arrangement of the shared memory segment is more complicated compared to the task preemption time benchmark test. The layout for the shared memory segment can be seen in Listing 6.9.

The test startup is almost the same as for the task preemption benchmark test before. Details are not printed here. All processes needed for the test execution are forked within the main process and use the same synchronisation mechanism (channels as events). The actual task switch is invoked by calling the `sched_yield()` system call.

Table 8.7 presents the results of the task switch latency benchmark test for the QNX Neutrino operating system with two alternating processes. The same test was repeated with 16 (Table 8.8), 128 (Table 8.9) and 512 (Table 8.10) switching processes. As can be seen, the time required for a task switch increases with more involved processes.

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.440	0.406	3.042	2.637	0.176
CPU utilization	0.463	0.423	3.450	3.027	0.196
I/O utilization	1.129	0.434	3.607	3.172	0.547

Table 8.7.: Benchmark test results [μs]: QNX switch task (2 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.465	0.420	2.750	2.330	0.196

Table 8.8.: Benchmark test results [μs]: QNX switch task (16 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.639	0.507	6.222	5.715	0.583

Table 8.9.: Benchmark test results [μs]: QNX switch task (128 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.824	0.539	20.260	19.722	2.008

Table 8.10.: Benchmark test results [μs]: QNX switch task (512 tasks)

8.4.3. Task Creation Test

The task creation benchmark test measures the time it takes for creating a new process. According to the description in Section 5.4.3.3 a new task is spawned in each test step within the test main loop by calling the `fork()` system call. Time is measured immediately before and after (in the new process) invoking `fork()`. For transferring the second measurement value to the main process a shared memory segment is used. The implementation for the task creation benchmark test is not shown here, since it is almost the same as already described in Listing 6.11. Like for a Linux system, a new created process in QNX inherits the priority level and the scheduling policy of the parent process. The new process is an exact duplicate of the calling process except some points which are not discussed here. The new process will also start in the same partition like the parent process. Unlike the Linux operating system, the new created process is not put at the start of the FIFO run-queue. A additional call of the `sched_yield()` system call is needed and thus included in the time measuring.

The results of the task creation benchmark test are shown in Table 8.11.

8.4.4. Interrupt Tests

The implementation of the three interrupt benchmark tests as described in Section 5.4.3.4 are explained in this section. For the interrupt latency benchmark test and

Scenario	Average	Min	Max	Gap	Deviation
Normal	186.772	174.839	206.366	31.527	5.008
CPU utilization	170.843	147.151	251.214	104.064	18.120
I/O utilization	329.973	229.736	379.933	150.197	31.121

Table 8.11.: Benchmark test results [μs]: QNX task creation

the interrupt dispatch latency benchmark test an interrupt handlers that will be registered on the parallel port interrupt is needed. Listing 8.6 shows the implementation of the interrupt handler for the interrupt latency benchmark test.

interrupt_isr.c

```
const struct sigevent *handler(void *arg, int id) {
...
    InterruptLock(&spinlock);
    rdtscp_32(tsc);
    cpuid();
    clear_parport_interrupt();
    *((uint32_t *)arg) = tsc;
    id = id;
    InterruptUnlock(&spinlock);
    return NULL;
}
```

Listing 8.6: QNX interrupt benchmark test handler

The interrupt handler just captures the current value of the TSC register as early as possible and returns. Values between the main benchmark test and the measurements inside the interrupt handler are transmitted via a shared variable. This is possible because the interrupt handler is connected to the context of the thread it is contained within (Section 8.2.2).

With the introduced interrupt handler, measuring the interrupt latency is quite simple. The interrupt latency benchmark test main loop is similar to the test for the Linux operating system (Listing 6.14). The results of the benchmark test are shown in Table 8.12.

Scenario	Average	Min	Max	Gap	Deviation
Normal	6.291	4.724	7.772	3.048	0.436
CPU utilization	6.287	4.735	7.802	3.067	0.380
I/O utilization	5.871	4.801	7.840	3.040	0.626

Table 8.12.: Benchmark test results [μs]: QNX interrupt latency (ISR)

The interrupt dispatch latency benchmark test is similar to the interrupt latency benchmark test except of the time measurement points. For this test the first value is captured within the handler. The second time value is gathered when returning from interrupt. The interrupt handler for this test is modified in the way, that the

TSC register is taken as late as possible. The results of the interrupt dispatch latency benchmark test are provided in Table 8.13.

Scenario	Average	Min	Max	Gap	Deviation
Normal	1.348	1.161	5.232	4.071	0.514
CPU utilization	1.429	1.157	7.528	6.371	0.782
I/O utilization	2.271	1.218	6.635	5.416	1.130

Table 8.13.: Benchmark test results [μs]: QNX interrupt latency (dispatch)

For the interrupt to task latency benchmark test an additional thread is created which is acting as a second level interrupt handler. As described in Section 8.2.2 QNX provides a signaling mechanism for interrupts which occur. The threaded handler registers for events from the parport interrupt. A call to `InterruptWait` will block the thread until an event from this interrupt is sent. The implementation of the interrupt handler thread is shown in Listing 8.7.

interrupt_slih.c

```
void *int_thread(void *arg) {
...
    if (0 > setup_rt(RT_PRIO +1))
        exit(EXIT_FAILURE);

    if (0 > set_affinity(BENCHMARK_CPU))
        exit(EXIT_FAILURE);

    SIGEV_INTR_INIT(&event);
    id = InterruptAttachEvent(PARPORT_IRQ, &event, 0);

    while (1) {
        InterruptWait (NULL, NULL);
        rdtscp_32(data);
        cpuid();
        InterruptUnmask(PARPORT_IRQ, id);
    }
}
```

Listing 8.7: QNX interrupt benchmark test threaded handler

The interrupt to task latency benchmark test is identical to the interrupt latency benchmark test and is not listed here. The results of the test are shown in Table 8.14.

Scenario	Average	Min	Max	Gap	Deviation
Normal	9.967	9.127	13.313	4.185	0.997
CPU utilization	10.109	9.157	14.078	4.921	1.092
I/O utilization	10.607	9.123	14.971	5.848	1.228

Table 8.14.: Benchmark test results [μs]: QNX interrupt latency (SLIH)

8.5. Summary

The kernel structure of Neutrino has little in common with the Linux operating system. The microkernel design is taken as an inspiration for some aspects of the operating system extension introduced later in Chapter 11 (system-call handler threads). Further, the partitioning mechanism and the interrupt handling explained in this chapter will be discussed again in a modified form for our new operating system.

Part III.

A Hard Real-Time Linux Operating System

9

Requirements Discussion

Traditional real-time operating systems like QNX Neutrino have been designed to fulfill real-time requirements inherently. In contrast to QNX, Linux was initially designed around fairness and good average performance. It does not provide (hard) real-time capabilities to its applications. In the previous case studies we have analysed two different variants of how real-time can be introduced to Linux. Based on these analyses together with the QNX Neutrino case study we can outline a list of features required for real-time operating systems:

Preemtable kernel If a high priority task becomes ready for execution the operating system has to be able to switch as soon as possible from a low priority task to the new higher priority task. In older Linux versions a task inside the kernel (e.g. system-call) could not be preempted by any other task before it has left the kernel. Since Linux version 2.6 the kernel implements the *preemptible kernel* design whereby a task can be scheduled even if it is currently on a kernel path. Kernel preemption remains disabled in many situations inside the kernel, for instance if a task holds any exclusive resource (spin-lock). The RT-Preempt Linux extension addresses this issue by turning many spin-locks into preemptible alternatives. For the HLRT approach this situation can not occur since tasks are isolated on CPUs. QNX Neutrino solves this problem by executing almost all operating system services in task context (microkernel design).

Strict task priorities Sharing resources in the system, especially CPU time, between various tasks need to be adjustable. Normally this is done by assigning a priority level to a task. The tasks are scheduled according to their priorities. In QNX a more refined method is shown which allows the combination of tasks to groups (partitions). These groups introduce a new priority level on top of the task priorities and provide greater adjustability in the system wide scheduling. HLRT also implements a simple way of partitioning. Each real-time task constitutes its own partition which is assigned exclusively to one CPU.

The extension of the Linux kernel with real-time features means that real-time applications interact with other non-real-time components within the same system. A strict distinction between real-time and non-real-time is required for a real-time

Linux system. Grouping tasks to partitions will be the basis for addressing this requirement in the operating system introduced in the next chapters.

The previously introduced preemptible kernel design allows a low priority task to be preempted whilst holding an exclusive resource. A higher priority task which is requesting this resource can be delayed by the lower priority task (priority inversion, Section 6.2.1). Because exclusive resources can be found at many places inside the Linux kernel, it is absolutely necessary to deal with this problem.

Interrupt routing In addition to strict task priorities it is necessary for a real-time operating system to provide a way to reduce the opportunity for unrelated activities to interrupt a task on a high priority level. Furthermore some sort of prioritization for interrupt (handlers) is needed in order to assure stable interrupt latency. The RT-Preempt patch implements interrupt handlers as threads which have a static priority assigned and thus can be sorted into the task priority hierarchy. However, some interrupts cannot be executed in task context and remain as non-preemptible kernel paths. Since interrupt handlers also request locks for exclusive resources, this interferes with strict task priority scheduling. The HLRT extension provides an interrupt routing mechanism which addresses the problem of interrupting high priority tasks but does not support any solution for assigning priority levels to interrupt handlers.

The previous analyzes of existing real-time solutions have shown how these features are implemented for the particular systems. From these implementations we can identify some requirements for real-time operating systems, which will be discussed in this chapter. The requirements are derived from the investigation of these systems and summarized in Table 9.1.

Id	Name	Text
`REQ_0001`	Support for real-time tasks	A task can be defined to be a real-time task. This sets the real-time task apart from non-real-time tasks. A distinction between real-time tasks and non-real-time tasks is possible.
`REQ_0002`	Priority levels for real-time tasks	A priority level can be assigned to a real-time task. The priority of a real-time task reflects its importance in the system.
`REQ_0003`	Real-time tasks are preemptive	A real-time task with a higher priority level interrupts the execution of a real-time task with a lower priority. A lower prioritized task cannot interrupt a real-time task on a higher priority level.
`REQ_0004`	Real-time tasks are preferred	Real-time task priority levels are always at a higher level compared to non-real-time task priority levels.
`REQ_0005`	Synchronisation of real-time tasks	Real-time tasks can be synchronized in a way that does not affect other real-time tasks.
`REQ_0006`	Timing constraints of real-time tasks	A deadline can be specified for a real-time task. When the deadline passes, the task must have performed a special call to signal the compliance of the deadline.

Id	Name	Text
REQ_0007	Periodic real-time tasks	The deadline for a real-time task can be taken as a period. The task can be scheduled according to that period. The task is suspended from the compliance of the deadline to the start of the next period.
REQ_0008	Detection of missed deadlines	Missed deadlines of a real-time task must be visible to that task. A signalling mechanism is needed to interrupt the normal execution of the task in case of a missed deadline.
REQ_0009	Real-time task partitioning	Real-time tasks can be combined to groups.
REQ_0010	Grouped real-time tasks are isolated	A real-time task in one partition can not handicap the execution of real-time tasks in other partitions.
REQ_0011	Timing constraints of partitions	The task scheduling within a partition can be adjusted and can be distinguished between other partitions.
REQ_0012	Resource management	System resources can be assigned exclusive to a certain partition and thus to the appropriate real-time tasks.
REQ_0013	Interrupt splitting	Interrupts are divided into a hardware depended and a processing section.
REQ_0014	Routing of hardware depended interrupt sections	Treating the hardware depended section of occurring interrupts can be linked to specific CPUs in the system.
REQ_0015	Routing of interrupt processing sections	The processing section of an interrupt can be linked to specific CPUs in the system.
REQ_0016	Scheduling of interrupt processing sections	The processing section of an occurring interrupt can be scheduled in a way that a real-time task cannot be interrupted by that interrupt.
REQ_0017	User control of system resources	A real-time task must be able to adjust all parameters that are significant for executing that task.
REQ_0018	Adjustable priority levels for real-time tasks	The Priority level of a real-time task can be adjusted at runtime.
REQ_0019	Specify paging or process swapping of real-time tasks	A mechanism for memory locking is provided to prevent the memory of a real-time task from being paged to the swap area.
REQ_0020	Composition of real-time tasks	The grouping of real-time tasks to partitions can be changed at runtime. A real-time task can be transferred between partitions.
REQ_0021	CPU affinity of real-time tasks	A real-time task can be bound to a certain CPU in the system. The task will not be executed on another CPU. This prevents the task from being migrated between CPUs.
REQ_0022	Availability of runtime properties of real-time tasks	Runtime properties of a real-time task must be visible for that task. The task can know its priority level, the partition assignment and the current time within a specified task period.

Table 9.1.: Listing of determined requirements

There are different ways for an operating system to obtain real-time capabilities. Not all of the listed requirements in Table 9.1 have to be implemented to provide at least partial real-time support. For example, an operating system can support real-time scheduling with compromise on real-time task partitioning. However, this would mean a lack of functionality, as will be seen later in this chapter.

9.1. Analysis

The requirements from Table 9.1 and their relationships are shown in Figure 9.1. Three main paths can be identified, each starting from REQ_0001, REQ_0013 and REQ_0017.

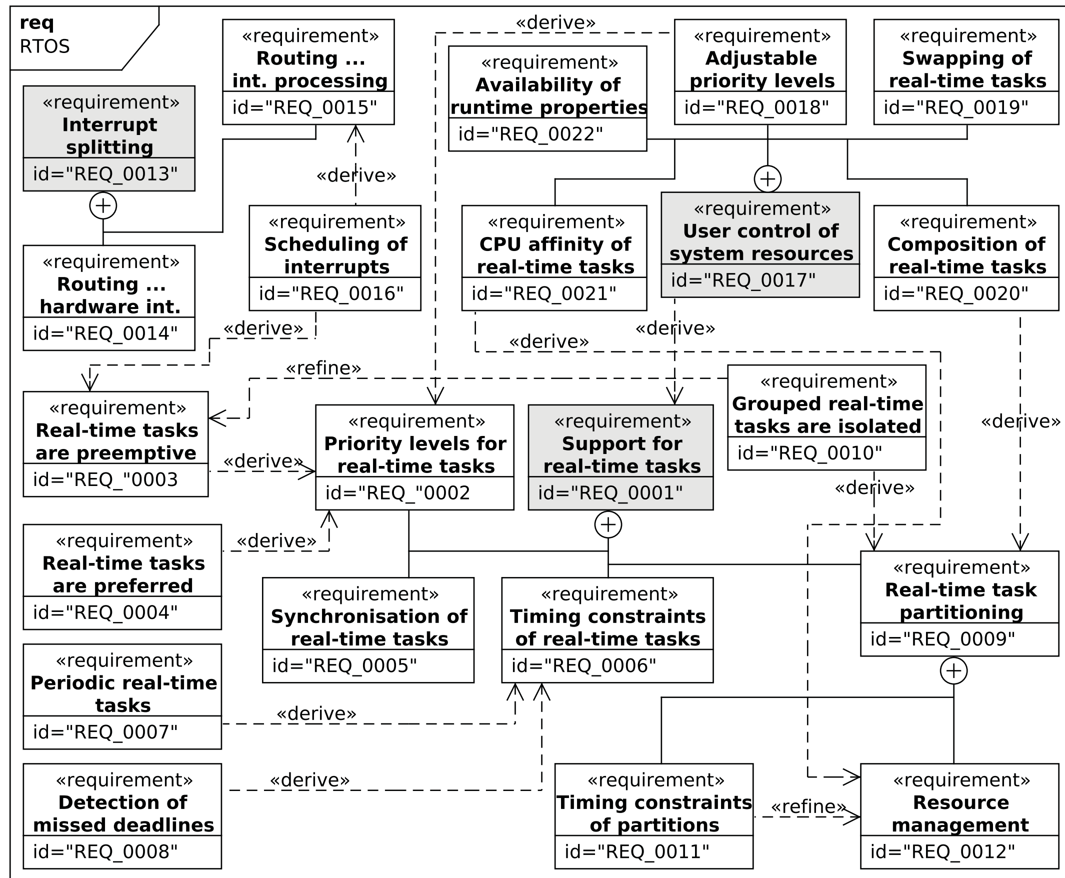

Figure 9.1.: Real-time operating system requirements

As mentioned above it is necessary to distinguish between real-time and non-real-time tasks (REQ_0001) in a patched Linux kernel. (This requirement is meaningless for the QNX Neutrino operating system.) It constitutes the needs for task priority levels (REQ_0002) and task preemption (REQ_0003) and introduces a requirement to provide task partitioning (REQ_0009). Partitioning allows to assign system resources exclusivly to a group of tasks (REQ_0012) and provides strict separation of task execution in different partitions (REQ_0010).

Since real-time tasks have an assigned priority level it is necessary to allow this priority to be adjustable (REQ_0018). Together with other requirements like preventing the memory of a real-time task from being paged to the swap area (REQ_0019) a requirement can be identified that grants access to runtime properties of a real-time task (REQ_0017).

The need to control interrupts in a more flexible way than it is supported by the native Linux kernel is presented in REQ_0013. The hardware depended section of an interrupt (REQ_0014) as well as the processing section (REQ_0015) can be assigned to certain CPUs in the system. In addition to it the processing section of an interrupt can be scheduled in a way that a real-time task cannot be preempted by that interrupt (REQ_0016).

Requirement REQ_0006 describes that a real-time task can have an assigned deadline. Based on this the periodic task model (Section 3.1.3) is introduced into the system (REQ_0007). Furthermore it must be possible for a real-time task to react accordingly if a deadline was missed (REQ_0008).

9.2. Coverage

Table 9.2 shows the requirement coverage for the operating systems which have been described previously.

RTOS	REQ_0001	REQ_0002	REQ_0003	REQ_0004	REQ_0005	REQ_0006	REQ_0007	REQ_0008	REQ_0009	REQ_0010	REQ_0011	REQ_0012	REQ_0013	REQ_0014	REQ_0015	REQ_0016	REQ_0017	REQ_0018	REQ_0019	REQ_0020	REQ_0021	REQ_0022
RT-Preempt	✓	✓	✓	✓			✓						✓	✓	✓	✓	✓	✓	✓		✓	✓
HLRT	✓			✓		✓	✓	✓					✓	✓	✓		✓		✓		✓	
Neutrino		✓	✓		✓		✓		✓	✓		✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
HRTL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 9.2.: Requirements coverage

The RT-Preempt extension does not provide the definition and tracking of deadlines for real-time tasks. Moreover, no partitioning is implemented. Thus, the appropriate requirements are missing for the RT-Preempt Linux system. Since real-time tasks use the SCHED_RR or the SCHED_FIFO scheduling policy, they are separated from non-real-time tasks. Periodic real-time tasks can be realised by using an interval timer, programmed with setitimer().

Real-time tasks in a HLRT patched kernel do not need any priority levels, since they are executed on a reserved CPU and cannot be interrupted. The question if another task has a higher priority does not arise. The kernel is not changed by the patch in order to introduce preemption. Instead the Linux based preemption model that comes with all kernel versions starting from 2.6 needs to be disabled. As mentioned above,

HLRT does not provide any features for scheduling threaded interrupt handlers with real-time priorities. The periodic task model is implemented by the HLRT kernel but there are no possibilities for a real-time task to determine the current elapsed time of the current period.

QNX Neutrino does not distinguish between real-time and non-real-time tasks. Neutrino cannot be compared to the Linux patch variants in this case, because it implements a completely different design. However, similar to Linux systems, every task in Neutrino has an assigned priority level and is scheduled according to that priority.

10
Hard Real-Time Linux System Design

Based on the previous discussions of real-time operating systems, a list of requirements has been identified. This chapter presents a design for an extension to the Linux kernel that combines all these requirements.

10.1. Overview

In order to meet the identified requirements in Chapter 9, the Linux kernel must be modified in several ways. Each of the previously studied real-time operating systems have contributed to the system design. For example, the partitioning concepts discussed in Section 10.3 are inspired by the QNX Neutrino adaptive partitioning scheduler (Section 8.3). The CPU reservation mechanism from the HLRT extension (Section 7.2) builds the basis for the concepts introduced in Section 10.2. On top of the CPU reservation paired with a partitioning strategy the idea of threaded interrupt handlers from the RT-Preempt extension (Section 6.3) gets introduced into the system design.

In the hard real-time Linux design discussed in this chapter, a higher-level scheduling unit is introduced. Each real-time task can be assigned to a defined *scheduling group*; a task becomes a real-time task when it is assigned to a scheduling group. The time-triggered static schedules for this groups (*static scheduling groups*) are developed off-line. Each group is assigned to one CPU and has a defined computation time. A *group cycle* for a CPU is the sum of all computation times of the groups associated to this CPU. An example for a schedule with groups for several CPUs is shown in Figure 10.1.

Group cycles on different CPUs are independent of each other. However, they can be synchronised under the consideration of some limitations. Each (static) group belongs specifically to one cycle and thus is assigned specifically to one CPU. The *system group* presents a special case. This group is reserved for the basic Linux system and *dynamic scheduling groups*. The system group can be defined on several CPUs and can therefore belong to more than one cycle. At least one CPU (*system CPU*)

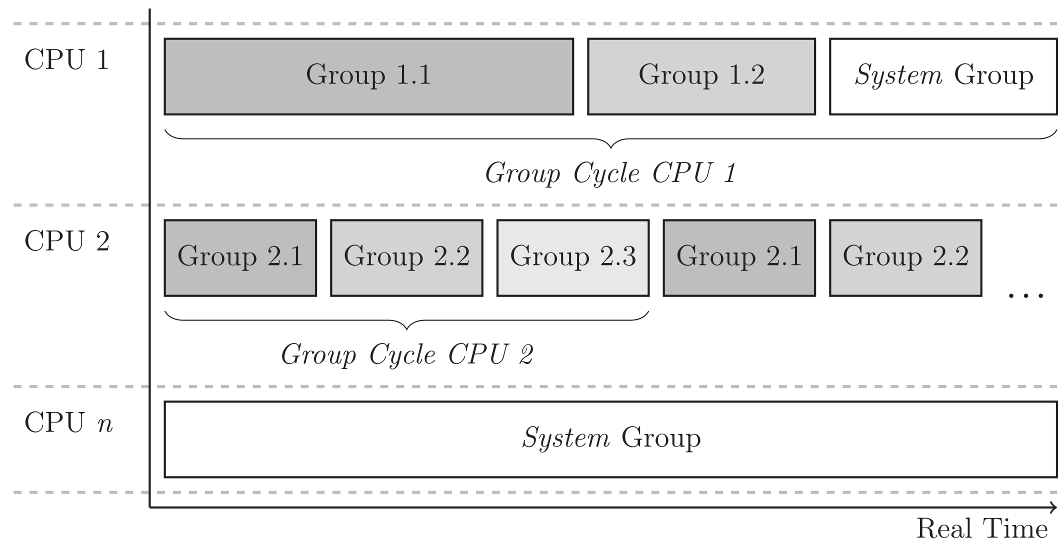

Figure 10.1.: Hard real-time Linux scheduling groups (Example 1)

must not host any groups (not shown in Figure 10.1). This is important to handle sporadic Linux tasks and events such as interrupt handler.

A group cycle is combined with some other properties to an object called *core*. Each CPU in the system can host exactly one core (except of the system CPU). If a core is assigned to a CPU, the *normal* Linux scheduling for that CPU is deactivated (CPU reservation). A reserved CPU is completely isolated from the Linux system including interrupt handling. If the assigned core contains one or more system groups, the Linux scheduler will be free to schedule tasks within the related time slot of these groups. In order to reserve a CPU, a core must be selected which will be assigned to that CPU together with a *CPU profile*. A profile can be assigned to several CPUs while a core can only be hosted by one CPU.

10.2. CPU Reservation

On a system with n CPUs the HRTL system allows $n-1$ CPUs to be reserved. Unlike the HLRT extension a reserved CPU is not claimed by a single task. Instead, reserving a CPU extends the running HRTL partitioning core (Section 10.3) with an additional resource for scheduling groups. It is not possible for the Linux system to schedule any task on a reserved CPU and thus interrupt the partitioning scheduler.[1] The modifications to the Linux kernel that are needed to allow the reservation of a CPU are discussed in Section 11.4 and Section 11.5.

To reserve a CPU, a static group scheduling plan is required. This plan is provided by a group cycle (core). Details for group cycles can be found in Section 10.3.2. A CPU can only be reserved in combination with such a scheduling plan.

[1]This constraint is discussed in Section 10.3.1.

10.2.1. Interrupt Routing

All maskable interrupts are disabled for a reserved CPU. The running static group scheduling plan can reenable interrupts for its CPU. This can be triggered from a real-time task running in one of the groups included in the plan. Thus it is possible to book individual CPUs for the processing of certain interrupts.

The clock interrupt is disabled after a CPU becomes reserved. This depends on the underlaying hardware. A last time tick interrupt is received after the reservation phase is completed. This tick initiates the group cycle. The group scheduler is then responsible for adjusting and calculating timer interrupts in order to keep the scheduling plan working.

The Linux system is not allowed to send any kind of interruption to a reserved CPU. For that reason, APIC inter processor interrupts (IPI) must be intercepted and handled in a special way. Since the HRTL task scheduling subsystem and the group scheduler use IPI calls for synchronisation, they can not just be ignored. Whenever a non reserved CPU tries to interrupt a reserved CPU by sending an IPI call, this call is taken to a queue and handled later at a time when the core running on that reserved CPU decides it.

10.2.2. CPU Profiles

A CPU profile defines the clock source and the time keeping that should be used for a reserved CPU. In the HRTL system it is possible to define several modules for these services. They will be considered in Section 11.3. One CPU profile can be taken for several reserved CPUs. Once constituted, a CPU profile can not be changed on a reserved CPU. It is selected together with a static group scheduling plan in the reservation process.

Besides the timing services a CPU profile also handles deadline tracking for real-time tasks running in combination with this profile. The HRTL system provides an interface to get information about real-time deadlines from user space. The verbose level for notifications about deadlines can be adjusted. It is possible to see which deadlines are defined and if they are met or missed together with timing information. Since CPU profiles can be used for several reserved CPUs simultaneously, special care must be taken when transmitting deadline states from a CPU to a profile. A method which enables handling concurrent deadline states without blocking the transmission of a deadline is presented in Section 11.6.5.

A CPU profile represents the bottom level of the task creation hierarchy as described in Section 10.4. Here, it can be defined in which scheduling group new created tasks should be placed. This rule can be overwritten at two other points in the hierarchy as will be shown later.

10.2.3. Housekeeping

Some mechanisms in the Linux kernel require a periodic treatment even on reserved CPUs. For instance, the read-copy update implementation in the Linux kernel needs

each CPU to achieve a quiescent state at times. The right time to go through such a state is during a housekeeping phase. It is up to the core running on a reserved CPU to invoke a housekeeping phase. The work to be done in a housekeeping phase is organized in event queues. Achieving a quiescent state is only one event in a queue. The queue for IPI calls, which was previously described is also handled during housekeeping. The CPU houskeeping framework for the HRTL system provides the following housekeeping events:

Inter processor interrupts call pending IPI calls for this CPU.

Read-copy update achieving a quiescent state. All local references shared to data structures have been lost and no assumptions are made based on their previous contents.

Kernel print flushing the kernel message buffer. This will provide pending kernel messages for the log daemon.

Scheduling (group and core) perform group scheduling related housekeeping actions. This will be discussed in Section 10.3.2.

Each reserved CPU has its own housekeeping object that manages the queues described. Which of the queues are going to be processed during a housekeeping phase can be defined with a bit mask. This means that not all queues have to be treated in each housekeeping phase. Further, the events inside a queue are sorted by priorities.

10.3. Partitioning

In the hard real-time Linux system design, a partition is represented by a scheduling group. Partitions are protected memory areas with distinct task scheduling strategies. Different types of scheduling groups are discussed in Section 10.3.1. Each real-time task belongs to a group. The task behavior and scheduling inside a group are managed by a real-time task scheduler which is discussed in Section 10.4. Further, communication between real-time tasks and kernel subsystems (via system-calls) can be directed to other partitions (Section 10.3.4). Tasks are completely isolated in their group and can not interfere with the overall group scheduling.

10.3.1. Scheduling Groups

A scheduling group defines the overall time behavior for real-time tasks which are associated with this group. Further, CPU affinity and system-call redirection are specified for real-time tasks through adjustable parameters of scheduling groups. Two different types of scheduling groups can be differentiated:

Static groups have fixed time slices in which associated tasks can be scheduled. A static group is connected to a core object at runtime and receives a static time slot

inside the core scheduling plan which conforms with the group's time slice. The length of the core scheduling plan (group cycle time) dictates the period of the group. Once assigned to a core object, the group's time slice is active in every core cycle during the associated time slot and will not move inside the core scheduling plan. Thus, each static group has a fixed runtime (time slice) and a fixed period (runtime of the core scheduling plan).

The overall time slice for a group (runtime) is divided into an execution and an idle part. The execution part represents the available time slice for scheduling tasks connected to the group. Generally, the idle part is much smaller than the execution part. It can be defined to be zero which means that the execution part equals the group's runtime. If defined, the idle part provides time for performing housekeeping events (Section 10.2.3).

As mentioned in the introduction to this chapter, two kinds of static groups are distinguished. The properties described above are valid for both group types:

System groups All tasks that are not defined to be real-time run as *normal* Linux tasks. Together with Linux main system kernel threads they are executed in system groups. It is the role of the Linux scheduler to balance these tasks to different CPUs. The Linux scheduler needs to be extended that the execution time of each system group and the group cycle time of the corresponding core can be taken into consideration for load balancing. For example, the situation shown in Figure 10.1 defines three different groups to CPU 1. A load level of about 75% must be shown to the Linux scheduler for this CPU. Ground up using this basis, a reasonable load balance can be made by the Linux scheduler for the system.

System groups provide time slots for the Linux scheduler as well as for dynamic real-time partitions. Actually, the Linux scheduler is seen as a dynamic group which shares the available runtime together with other dynamic groups. If a system group is replaced by another group inside a core object, the system group becomes invalid and the associated memory objects are freed.

Real-time groups Real-time groups can exist beside system groups on different CPUs. For the group scheduler there is no difference for handling system groups and real-time groups. However, both types behave differently in detail. The task behavior inside a real-time group is managed by a real-time scheduler. Unlike system groups real-time groups are not managed or influenced by the core Linux system or the dynamic group scheduling. Moreover, they are either active and thus included into a core scheduling plan[2] or they are inactive. In contrast to system groups a real-time group is valid if it is not assigned to a core object.

Dynamic groups have a minimum percentage of available CPU time allotted (budget). The available CPU time is given by the time slots provided by system groups. Each system group extends the available CPU time by its execution time slot (excluding the idle time). The normal Linux task scheduler is executed inside the

[2]The scheduling plan can still be inactive in case it is not assigned to a CPU.

dynamic *Linux group*. Unlike static groups a dynamic group is not assigned to a certain core object and thus it is not bound to a specific CPU. The dynamic group scheduler manages all available time slots and decides which dynamic group will receive how much of CPU time (Section 10.4.4). The budget for a dynamic group is given by two values:

Budget for reserved CPUs ensures a specified minimum share of time in system groups when the system is overloaded. It is guaranteed that a dynamic group will receive this budget. If the system is not overloaded, the remaining CPU time is shared between all dynamic groups. The Linux scheduler is executed if no dynamic group can be executed. Thus, the Linux group has a budget for reserved CPUs equal to zero and is only executed on reserved CPUs if the system is not overloaded.

Budget for non-reserved CPUs specifies a maximum share of time on unreserved CPUs. A group that has runnable tasks will receive exactly this percentage of CPU time provided enough tasks for all available CPUs are ready for execution. The concept of sharing the remaining CPU time in a system which is not overloaded is not followed here. The Linux group always receives CPU time that is not used by other groups.

Several properties can be assigned to scheduling groups. They will be discussed in the following sections. Some features only make sense for one kind of scheduling group. For instance, dynamic groups do not have assigned idle time slots.

10.3.2. Group Cycles

The static groups assigned to a core object build a group cycle (or group scheduling plan). Each static group associated with a scheduling plan receives a static time slice inside the plan. The order of time slices (for static real-time groups) of a plan will not change. The group scheduler which is responsible for changing the active group uses a timer interrupt to preempt the execution of the current group. A running group is only preempted by the group scheduler in order to change the active time slot. If there is only one group defined for a scheduling plan, this group is not interrupted by the group scheduler (provided the group and the core object do not define any idle time slots). The timer interrupts (scheduling points) are calculated once before the scheduling plan is started. Each scheduling point is defined by an offset which will be added to the last scheduling point in order to determine the exact time for the next interrupt.

A static group contains four scheduling points. Two for each time slice (execution and idle) to denote the start and stop of the time slice. A scheduling point is implemented as an event which is included in an event queue. Each event queue has an associated timer which is programmed according to the description above. Thus, the four scheduling points for a static group will invoke three timer interrupts, because the stop of the execution part and the start of the idle part are located in the same

event queue. The next static group will share one event queue for its start event with the stop event from the previous one. A detailed discussion of the group scheduler implementation can be found in Section 11.6.

Each point in time in a group cycle is associated with a time slice of one static group, whereby it applies that a static group is represented only once inside one cycle. Free space of a scheduling plan is filled up by system groups. If a static real-time group is added to a scheduling plan, the most appropriate system group is removed. On the other hand, removing a real-time group from the plan will create a system group that fits inside the blank. Bordering system groups are allocated into one group.

The group scheduler provides various events for the subsystems enclosed in a static group. An event can occur inside the time slice of the group. Besides the start and stop of a period, dynamic timers are also implemented. To define a timer a certain point in time must be specified. This point in time is given by a combination of the period's number and the offset inside this period (this will be discussed in Section 11.1). If the scheduling plan is already running on a CPU adding and removing groups generates overhead, because expected timers may need to be canceled.[3] Furthermore the fixed scheduling points for a new group have to be calculated. These group scheduling related housekeeping actions take place during the idle phase.

Each CPU has its own idle task. This task is scheduled in idle phases and every time an active real-time group can not schedule a real-time task. Housekeeping actions are also performed by this task.

10.3.3. Threaded Interrupt Handling

A processing section of an interrupt (Chapter 9) can be assigned to a static scheduling group or a scheduling plan. The interrupt is then executed by the idle task (not during housekeeping). The execution of the normal group task scheduling is interrupted and the idle task is executed (if the priority level allows it). Thus, the idle task implements a queue for occurring interrupts and can alter its priority. The interrupt handler (hardware dependent part) enqueues a work package for the idle task connected to the specified object (group or core) when the associated interrupt occurs.

10.3.4. System-Call Redirection

System-calls of tasks inside a real-time group (static real-time groups and dynamic groups) can be directed to handler threads (Section 10.4.2.1). The group object defines which system-calls are routed to which handler thread.[4] A task that executes a redirected system-call is suspended for this time and another task from the group can be executed. The system-call invocation has an assigned priority level based on the task priority inside the group. System-call redirection to handler threads allows task (and group) scheduling with very low latency, because no kernel preemption is

[3]Adding a new group will replace a running system group.
[4]A similar routing exists for normal Linux tasks.

needed for a task that never follows a kernel path. The handler thread itself is treated differently compared to normal real-time tasks and allows preemption at a high level.

10.4. Task Management

A task in the hard real-time Linux design becomes real-time when it is added to a scheduling group. The task is then subject to the rules of the respective group. Depending on the scheduling group different properties apply to real-time tasks which will be discussed later in this section. A task can be added to a group either because it is *moved* there as a running task or it is created from a task that is already running with real-time scheduling. In both cases it must be specified where the new task should be placed and which properties should be applied (e.g. deadlines, priority, . . .). Since the POSIX API for creating new processes should not be changed in order to adopt every application by the operating system, a standard configuration for new tasks is located in the group object and the CPU profile. Also, every task can have a configuration for processes it creates. If no standard configuration can be found, a new task will inherit all properties including the scheduling group from its parent.

10.4.1. Events

For task synchronization and time tracking, the hard real-time linux operating system provides some mechanisms which are accessible for real-time tasks. Other communication objects which are provided by the Linux core system are still available. As described above, each static scheduling group has a assigned period and a fixed runtime. A real-time task can get information about the current time inside its period. The API for time tracking and the features described below are discussed in Section 11.9.1.

10.4.1.1. Synchronisation

Several fixed synchronisation points are provided by the operating system. For instance, a real-time task can wait for the start of a new period of a static scheduling group (including its own group). Besides this, events can be allocated from user space. Each event has a unique Id and a key assigned. A task can register itself for an event, the task will be suspended till the event occurs. The occurrence of an event can be triggered from another task, from a scheduling point or from an interrupt.

10.4.1.2. Deadlines

A task can specify a deadline. Depending on the real-time scheduler implementation it will be scheduled according to its deadlines. How a task signals the operating system that it has met its deadline also depends on the scheduler module. The task that defines a deadline receives a signal (`SIGALRM`) if it has missed its deadline. Thus, the application can react on miscounted time behavior.

The previously introduced CPU profiles allow the connection of a *deadline watchdog* process to a profile. Different deadline events are generated from the tasks associated with such a CPU profile. The watchdog collects these events periodically and makes them available from user space. A deadline event consists of a timestamp and the process Id of the task which caused the event. Additionally, a second time value indicates the deviation from the expected time. Four different deadline events are provided:

DEFINED A real-time task defines a deadline

UNDEFINED A real-time task releases a deadline

MISSED A real-time task missed its deadline

MET A real-time task met its deadline

10.4.2. Kernel Preemption

Chapter 9 introduced the need for a preemptible kernel. Unlike the QNX Neutrino operating system, the Linux kernel is implemented as a monolithic kernel where all operating system services are realised as system-calls. Based on the Linux preemptible kernel design a strategy has been implemented to add greater flexibility for preempting a task inside the kernel during executing a system-call.

10.4.2.1. System-Call Handler Threads

A thread that is not actually running in kernel mode can be preempted at any time. Only when invoking a system-call the thread can be on a kernel path that may not be preemptible. The hard real-time Linux design discussed in this chapter provides the possibility of redirecting system-calls from tasks to *system-call handler threads*. These handlers are threads which can be added to scheduling groups like every other task too. A scheduling group can be connected with one or more system-call handlers, as a result the addressed system-call sections (e.g. file system, IPC, . . .) will be redirected to the handler threads. A scheduling group can be connected with a handler thread running in that group.

Whenever a thread invokes a system-call a work package is sent to the associated handler. According to the priority level of the sending task the package receives its own priority level and is stored in a list. The calling thread is suspended during system-call execution and another task can be scheduled in the group where the calling thread belongs. The suspended task is woken up by the handler (put to the ready queue) if the system-call execution is completed. When the thread is scheduled again inside its group the invoked system-call returns with a result and the thread continues running in user space.

System-call work packages have a specified entry point defined as a function. The execution of a packages starts with calling the entry point function and ends with returning from that function. During execution the stack segment of the suspended

thread (sending task) is used for local variables and function calls.[5] Thus, it is possible to preempt the execution of a work package and resume or start another one. If a package is preempted, the current environment is saved in the work package memory object and the stack segment is switched back. Depending on the cause for the interruption the work package is put back to the package list or stored in another place. The handler thread continues running with its own stack segment.

A system-call handler thread takes the work package with the highest priority from its list of packages and either calls the entry point function in order to start a new work package or switches to the previously saved environment in order to continue a preempted work package. In case a new package arrives with a higher priority than the one that is currently running, the current work package will be preempted and the handler switches to the new package. Because a handler can be connected to several partitions, a scheduling group has a own priority level assigned which is only considered for system-call work packages.

A system-call handler thread executing a work package must not be suspended or call the Linux scheduler function. The kernel must be prepared in several cases to intercept these situations. If a work package suspends itself (i.e. changing the task state) the package is marked to be suspended and is not put back to the package list. A *wake up* on a suspended work package will put the package back to the list. If a task switch is needed, the work package is put back to the list before the handler thread calls the scheduler function.

Only work packages are seen by a system-call handler. No information about the system-call itself (which is represented by the package) is needed. All details required for executing a work package are stored inside the work package memory object. This makes it possible to move packages between handlers. A handler can adopt a work package from another handler. A preemption of an adopted package will put it back to the list of the original handler thread.

10.4.2.2. Preemptible Critical Sections

The technique of directing system-calls to handler threads as described in the previous section allows tasks to be preempted at any time. However, system-call handler threads are only preemtible according to the standards of the Linux preemptible kernel design. In order to achieve greater flexibility and to reduce scheduling latency, the hard real-time linux design allows threads inside critical sections (spin-locks) to be preempted under certain conditions. Only system-call handler threads can be preempted inside a spin-lock if the thread does not explicitly disable preemption before requesting the lock. It is still ensured that only one execution path enters a critical section at the same time. To address the problem of priority inversion (Section 6.2.1) a suspended work package which is inside a critical section can be adopted by every other thread. A task that invokes a system-call without the involvement of a system-call

[5] Other segments and several memory references have to be switched. This will be discussed in Section 11.8

handler is also able to adopt the corresponding work package.

If a task is requesting a lock that is already blocked, the task either has to wait until the critical section is released or adopts the work package that holds the lock in case the package is preempted. If the requesting task has not explicitly disabled preemption it will be suspended. Otherwise, the task is busy waiting on the lock which means no scheduling takes place. Waiting tasks are organised in a priority queue for each spin-lock. If a task releases a lock the highest priority task in the queue gets the lock and continues executing. If a work package inside a lock gets preempted the highest priority task is triggered in order to adopt this work package.

10.4.3. Scheduling in Static Groups

For handling tasks assigned to a static group, different strategies arise. One of these strategies must be chosen for each group except system groups. The hard real-time Linux implementation provides a modular design to extend the scheduling core with new variants. In this section two variants of scheduling strategies will be discussed:

POSIX approach Each task has a priority level assigned between 0 and 64. The scheduler arranges the tasks in a ready queue in order of their priority. Lower priority tasks get interrupted by incoming higher priority tasks. Whenever a task is ready to compute no task with a lower priority will be executed. Therefore starvation of lower priority tasks is possible with large amounts of high priority tasks queuing for CPU time. Waiting time and response time depend on the priority of the task. Higher priority tasks have smaller waiting and response times (see also Section 3.2.3.1).

Figure 10.2 illustrates a possible situation in example 1 (Figure 10.1). Two tasks with the same priority level in Group 1.1 are scheduled with the FIFO strategy.

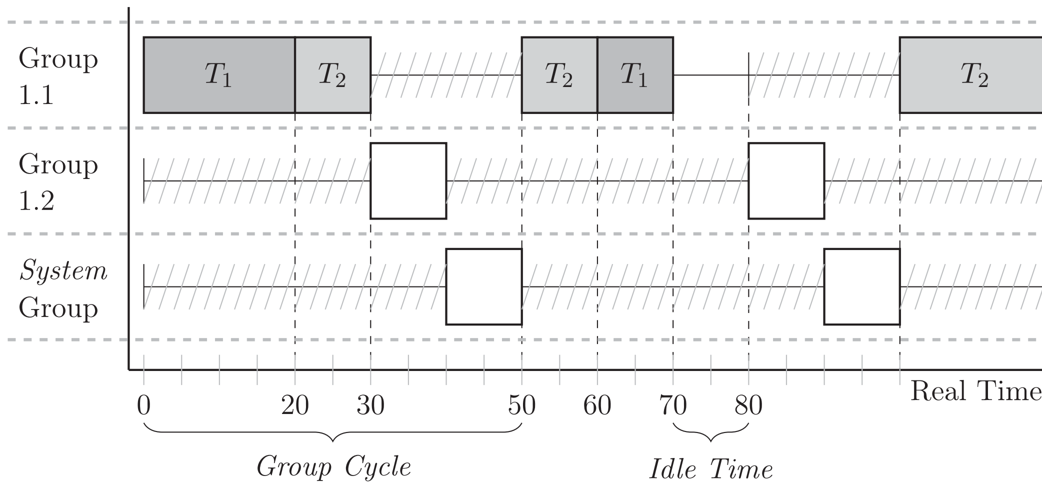

Figure 10.2.: FIFO scheduling in Example 1

First, the scheduler decides to execute T_1 until the task is blocked for some reason at $20ms$. At this time T_2 starts execution. At $50ms$ a new group cycle for Group

1.1 starts. Assumed that T_1 is ready for execution again, T_2 can not be interrupted by T_1, because the task in execution can only be interrupted by a task with a higher priority level. At $60ms$ T_2 is blocked for some reason and T_1 is executed again.

Rate-Monotonic scheduling The scheduling strategy must ensure that each task can run for the denoted time in its period. The parameters of all periodic tasks are known from the start (see also Section 3.2.2.2).

In the hard real-time Linux design described in this chapter, the rate-monotonic algorithm is applied to tasks in scheduling groups. Since each group has its own period in a static cycle the period P_i for a task T_i must be in relation to the group's period. A task in a scheduling group is described by a tupel $T_i = (P_i, C_i)$, where C_i is the computation time of the task given in milliseconds and P_i is the task's period given in counts of the group's period. For example, from Figure 10.1 the group 2.1 of CPU 2 is given with 3 tasks and a computation time of 10 milliseconds: $G = T_1, T_2, T_3$. The tasks in this group are defined with: $T_1 = (1, 2)$, $T_2 = (2, 7)$ and $T_3 = (4, 12)$. The situation is shown in Figure 10.3.

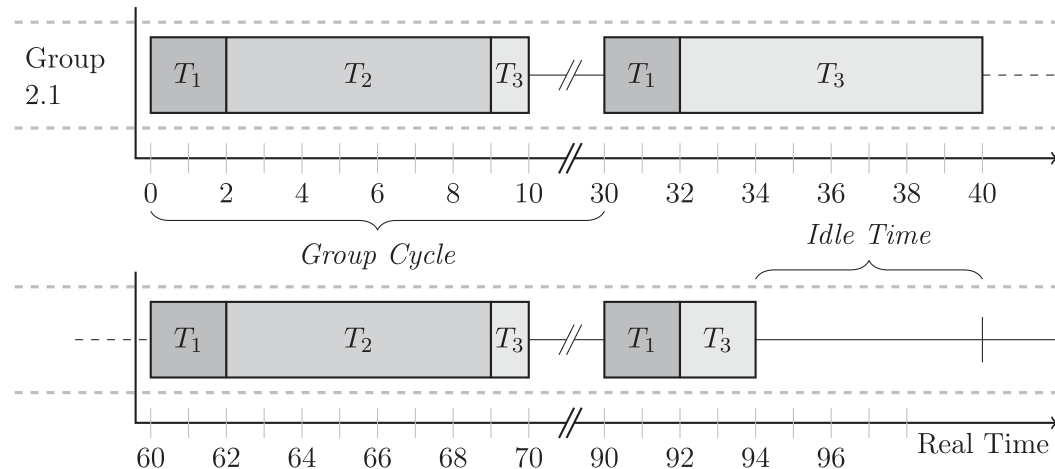

Figure 10.3.: RM scheduling in Example 1

CPU utilization for the given example is calculated with $U = 0.85$ (see Equation 3.1). Equation 3.2 defines the upper bound for guaranteed schedule in a worst-case scenario. One can easily see that the CPU utilization in the example is above this bound: $0.85 \nleq 0.7798$. This point will be discussed later in this section. Table 10.1 shows the calculated values for the 3 tasks of the example.

The algorithm chooses T_1 to be executed at first because of the highest priority. After that T_2 follows before T_3 starts. At $30ms$ T_1 becomes ready for execution again and T_3 is suspended. In the end T_3 finally finishes at $94ms$ and the CPU is idle for $6ms$ before the run begins again. This meets the expectation that there is 15% remaining of available processor cycles given by a CPU utilization of 85%.

A slight change of the example shows that the task set can not be scheduled even if the CPU utilization is below 1 (100%). The period of T_3 now is changed to 3

Task	C	P	U	$p()$	**Group cycle runtime**
T_1	$2ms$	$1 \rightarrow 10ms$	0.20	1	$1[2ms], 2[2ms], 3[2ms], 4[2ms]$
T_2	$7ms$	$2 \rightarrow 20ms$	0.35	0.5	$1[7ms], 3[7ms]$
T_3	$12ms$	$4 \rightarrow 40ms$	0.30	0.25	$1[1ms], 2[8ms], 3[1ms], 4[2ms]$

Table 10.1.: RM scheduling in Example 1

$T_3 = (3, 12)$. The calculations are shown in Table 10.2. The CPU utilization for the new example is now $U = 0.95$.

Task	C	P	U	$p()$	**Group cycle runtime**
T_1	$2ms$	$1 \rightarrow 10ms$	0.20	1	$1[2ms], 2[2ms], 3[2ms]$
T_2	$7ms$	$2 \rightarrow 20ms$	0.35	0.5	$1[7ms], 3[7ms]$
T_3	$12ms$	$3 \rightarrow 30ms$	0.4	0.3	$1[1ms], 2[8ms], 3[1ms]$ **!**

Table 10.2.: RM scheduling in Example 1 with unworkable task set

10.4.4. CPU Budget Based Scheduling

Tasks inside dynamic scheduling groups are scheduled according to their priority level. Each dynamic group has a minimum percentage of available CPU time allotted (budget). If none of the defined dynamic groups are running over budget the task with the highest priority is chosen for execution. Details on the CPU budget scheduler implementation can be found in Section 11.7.

The scheduler throttles CPU usage by measuring the average CPU usage of each group. The average is computed over a time window (per default 500 milliseconds). The current CPU usage is updated every 5 milliseconds (can be configured). This means every 5 milliseconds the usage for this time is added to the usage for the past 495 milliseconds to compute the total CPU usage over the window. Thus, the time window moves forward as time advances. According to the calculated CPU usage the scheduler balances the groups to their guaranteed CPU limits.

The system is overloaded if all partitions are demanding their full budget. The free time in a system which is not overloaded is shared between all partitions. An exception to this is the normal Linux scheduler. It is encapsulated in the so called *Linux group*. The Linux group will only receive additional time if no other task from any dynamic real-time group can be executed.

Whenever there are groups demanding less than their budgets, the scheduler chooses between them by picking the highest priority running task. Tasks are simply scheduled by the FIFO or Round-Robin policy, where Round-Robin only adds a timing property to the FIFO scheduling. In contrast to simple FIFO scheduling, a task (1) can be preempted by a lower priority task (2) from another group, if the group for task 1 has overused its budget and the budget for the group of task 2 is still available. A

group can only use its assigned CPU budget fully, if at least one runnable task for each CPU exists. For instance, a partition hosting only one task can only use 25% of available CPU time if four CPUs are executing system groups simultaneously.

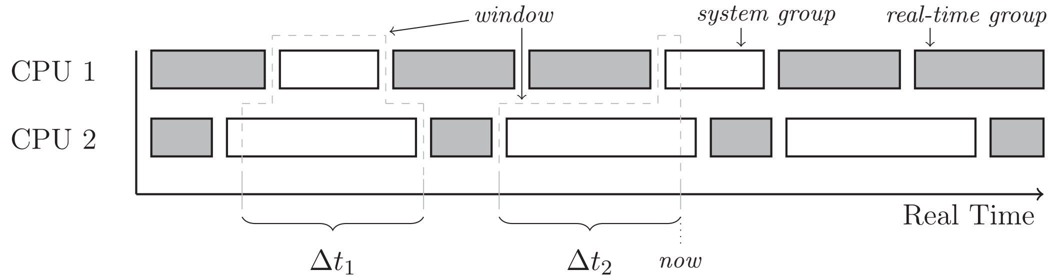

Figure 10.4.: Available CPU time (Example 2)

The available CPU time is provided by system groups. System groups can start and stop at any time, thus the available time for calculating the average CPU usage is not constant. Furthermore, the number of available CPUs can vary at runtime. It is possible that no CPU time is available for a period of time (no system group is running). The time window moves only over available CPU time, if no time is available the window freezes till a system group starts execution. This situation is shown in Figure 10.4. The overall window has a size of 500 milliseconds ($\Delta t_1 + \Delta t_2 = 500ms$). It is also possible that new system groups are added or removed. The time window is reset, if the last system group is removed from all scheduling plans in the system.

11

Description of the HRT Linux Implementation

This chapter describes the implementation details of the design introduced in Chapter 10. Only the key features of the hard real-time extension are discussed here. The entire source code is available for download as a patch for kernel version 3.5.7. Later in this chapter the implementation of the benchmark tests introduced in Section 5.4.3 for the HRTL extension are explained.

The complete HRTL kernel patch can be accessed at [Rad15a].

The real-time patch consists of modifications of the original Linux kernel in several parts of the kernel source code. All modifications are visible as conditional compilation parts enabled by the `CONFIG_HRTL_*` macros. Whenever possible new functions or even totally new sections are placed into separated files inside the source tree. At the very top level of the sources a number of directories containing the main parts of the extension can be seen:

`hrtl` The main HRTL kernel code

`arch/x86/hrtl` The architecture specific HRTL kernel code

`include/hrtl` The header files with global definitions

Almost all globally visible objects and functions have the prefix `hrtl_` assigned. Thus, it is easily possible to identify changes made by the patch. However, some new features do not fit into this scheme (i.e. the `SCHED_HRTL` scheduling class). Changes that cannot be identified by the rules introduced are indicated in this chapter.

Basic system settings are summerized in the `defines.h` header file. According to the description above, the file can be found in `include/hrtl`. The file includes mainly default values for variables which will be part of the discussions in this chapter.

11.1. Global Objects and Data Types

The implementation of the HRTL extension introduces some new basic types and global data objects to the Linux kernel. Some of them are used in several parts of the source code. Basic type definitions can be found in `include/hrtl/types.h`.

hrtl_key_t, hrtl_id_t are unsigned integer types. A key (hrtl_key_t) is a user defined value which must be in a defined range hrtl_key_range (minimum and maximum). An id is always chosen by the HRTL system. An invalid key or id has the defined value HRTL_KEY_INVALID respectively HRTL_ID_INVALID. Keys and ids are explained in detail later in Section 11.2.1.

hrtl_time_t describes a time value in the HRTL system. It is always a multiple of HRTL_TIME_UNIT_NS (defines.h). A value of HRTL_TIME_INVALID indicates an invalid time value. Various functions for translating time values are defined in types.h. Listing 11.1 shows an example of how a clock time value is translated to a HRTL time value and vice versa.

types.h

```
#define hrtl_time_to_ns(t)      ((hrtl_time_t)(t) * HRTL_TIME_UNIT_NS)
#define hrtl_ns_to_time(s)      ((hrtl_time_t)((s) / HRTL_TIME_UNIT_NS))
```

Listing 11.1: HRTL time value translation (nanoseconds)

hrtl_period_t is a complex data type to define a point in time in a periodic context. It includes a time offset and a period counter. Together with the period length a distinct point in time is given (Listing 11.2).

types.h

```
static inline hrtl_time_t hrtl_period_to_time(hrtl_period_t period)
{
        return hrtl_period_get_count(period)
               * hrtl_period_get_runtime(period)
               + hrtl_period_get_time(period);
}
```

Listing 11.2: Period to monotonic time

The HRTL subsystems (Section 11.3) and other parts of the kernel make use of data types for storing time values in a organised queue. Types and access functions for such a queue are defined in include/hrtl/timequeue.h. The implementation is based on *red black trees* which are already provided as a library by the Linux kernel (include/linux/rbtree.h). HRTL implements an interface to Linux red black trees that takes account of the HRTL time type (hrtl_time_t). A time queue is represented by the type hrtl_timequeue_head and contains elements of hrtl_timequeue_node. Each node has an expiration time in the hrtl_time_t format assigned. The nodes are sorted according to this time value. Only the node with the nearest expiration time is accessible in constant time.

Some global variables containing information on the system wide CPU allocation are introduced in include/hrtl/system.h. The use of these CPU masks is explained later in this chapter.

hrtl_cpus_available available CPUs for HRTL

hrtl_cpus_system CPUs used by the Linux system

hrtl_cpus_allowed CPUs that can be reserved by HRTL

hrtl_cpus_reserved CPUs used (reserved) by HRTL

hrtl_cpus_free CPUs not used by HRTL but allowed

hrtl_cpus_linux CPUs used by the Linux scheduler

hrtl_cpus_dyn CPUs available for the budget scheduler

The boot process for HRTL related structures is divided into six stages. Initialisation functions for HRTL structures are assigned to these subsections and executed during system start up:

hrtl_initcall_boot, **hrtl_initcall_early** first run, no dependencies and no memory management available

hrtl_initcall_mainsystem setup main-system structures

hrtl_initcall_subsystem setup sub-system structures

hrtl_initcall_service setup system services

hrtl_initcall_late everything else (HRTL core is fully running)

11.2. Management of Memory Objects

Some data objects in the HRTL extension share the same properties such as access and identification from user space. They are combined to a complex data type `hrtl_entity`. An instance of `hrtl_entity` is placed inside a larger data object and provides the features discussed in this section. The main structure of the `hrtl_entity` type is shown in Listing 11.3.

entity.h

```
struct hrtl_entity {
        char                        name[HRTL_NAME_LEN];
        void                        *context;

        /* -- assigned by the framework -- */
...
        struct {
                struct timeval    create;
                struct timeval    modify;
        } access;

        unsigned int                perms;
        struct hrtl_user            user;
...
        struct hrtl_hashentry       hashentry;
        hrtl_id_t                   id;
        unsigned int                references;
...
};
```

Listing 11.3: `hrtl_entity`

The HRTL system stores time information about creation and modification of hrtl_entity objects. Furthermore, permissions for modification and other operations are checked for the calling process. The concrete use of an hrtl_entity object depends on the superior data structure.

hrtl_entity objects are summarized to groups. An object is not accessible if it is not included in a group. An hrtl_entity_group object represents such a group and provides several functions for manipulating the objects contained. They are discussed in this section. The main management functions are:

hrtl_entity_group_get, hrtl_entity_group_put increases and decreases a counter inside the hrtl_entity object (references). The counter indicates if the object is being used by some kernel code at the moment.

hrtl_entity_attach_group adds an object to a group. The associated timestamps are adjusted. Adding an hrtl_entity object to a group sets the create and modify time stamp to the current time. The current user is set as the owner and the creator of the object.

hrtl_entity_detach_group removes an object from a group. This is only possible if the references counter is zero (the object is not in use). An hrtl_entity object outside a group can not be accessed by the functions discussed in this section.

The functions for creating and deleting hrtl_entity_group objects are not shown here. A group which contains objects can not be deleted. On the other hand, objects can only be added to a group until a defined limit is reached.

11.2.1. Id and Key Pool

The objects of an entity group are stored in linked lists. In order to identify and access the elements of a group, an identifier number (id) for each element can be generated. Each id is unique within the scope of an entity group. The id represents an index in a static array of references to entity objects. Since the size of the array is fixed the number of available ids is limited by the array size. The function hrtl_entity_group_connect_id() creates an id for an object inside a group. If no free entry in the id array can be found, the limit of available ids is reached and the operation fails. An element of a group can be accessed by hrtl_entity_group_get_by_id(). The function returns a reference of an entity element if the id could be found in the id array of the group.

In addition to ids, an entity object can also be accessed by an user defined identification key. Ids are chosen by the HRTL system and are only unique within one entity group. A key is given by the creator of an object and is connected to a hrtl_entity_key_pool. The keys are stored in a hash table. The size of the table and the range of valid keys can be defined. A key pool can handle keys from various entity groups. The HRTL system defines two different hrtl_entity_key_pool objects by default:

hrtl_keys_system for system service objects that need to be accessible from user space (e.g. Section 11.3).

```
#define HRTL_SYSTEM_KEY_MIN           0xFFFF0000                  defines.h
#define HRTL_SYSTEM_KEY_MAX           0xFFFFFFFE
```

Listing 11.4: System key range

hrtl_keys_resource for user defined objects like scheduling groups and profiles (e.g. Section 11.6).

```
#define HRTL_RESOURCE_KEY_MIN         0x00000001                  defines.h
#define HRTL_RESOURCE_KEY_MAX         0xFFFEFFFF
```

Listing 11.5: Resource key range

11.2.2. Entity System-Call Multiplexer

`hrtl_entity` objects can be accessed from user space via the entity system-call multiplexer. An entity group needs to implement an instance of `hrtl_entity_syscall` in order to provide user space access to its elements. The `hrtl_entity_syscall` object defines which operations are supported:

HRTL_ENTITY_SYSCALL_CONTROL allows to create and destroy objects.

HRTL_ENTITY_SYSCALL_CONFIG allows to change the ownership and the access mode of an object.

HRTL_ENTITY_SYSCALL_INFO provides detailed information on every element.

HRTL_ENTITY_SYSCALL_SETTINGS allows to configure settings of an element.

Each of the above listed flags enables one or more functions that can be called from the entity system-call. For instance, if the `CONTROL` flag is set, the functions `alloc()` and `free()` are available. The `SETTINGS` flag enables the functions `settings()` and `details()`. These four functions are accessible by pointers stored in the `hrtl_entity_syscall` object.

The enumeration type `hrtl_entity_domain` defined in `include/hrtl/syscall.h` needs to be extended for each `hrtl_entity_syscall` object. Details on HRTL system-calls will be discussed in Section 11.9.1.

11.3. Subsystems

The partitioning and task scheduling in the HRTL system is based on Linux kernel modifications and enhancements. The low level extensions are denoted as subsystems and are introduced in this section.

11.3.1. Time Base and Clock Sources

On a CPU that is controlled by the HRTL system (Section 11.4) the current time can be determined by the `hrtl_time_base`-functions. The CPU reservation mechanism resets the clock of a CPU to zero. The current time gives the time that passed since the CPU was reserved.

`hrtl_time_base_now()` gives the current time since the CPU was reserved in `hrtl_time_t`.

`hrtl_time_base_now_ns()` gives the current time since the CPU was reserved in nano seconds.

`hrtl_time_base_get_diff()` the distance between two reserved CPUs in nano seconds.

A clock for a reserved CPU is represented by the implementation of an instance of `hrtl_clock_source` (Listing 11.6). The `setup()` function is invoked during the reservation process. `now_ns()` is called by `hrtl_time_base_now_ns()`. The standard clock implementation of the HRTL system can be seen in `arch/x86/hrtl/clock-tsc.c`.

clock_source.h

```
struct hrtl_clock_source {
        struct hrtl_version version;

        unsigned long  (*now_ns)(struct hrtl_clock_source *);
        int            (*setup)(struct hrtl_clock_source *);
        int            (*shutdown)(struct hrtl_clock_source *);

        /* -- assigned by the framework -- */
        struct hrtl_entity entity;
};
```

Listing 11.6: Clock source

11.3.2. Timer and Interrupts

Based on the time base and clock source subsystem a timer system can be built. A timer device implements an instance of `hrtl_timer_device` (Listing 11.7). The `setup()` function is called during the reservation process and performs a timer calibration. The results of the calibration are stored in an `hrtl_timer_device_cpu` object for each CPU.

timer_device.h

```
struct hrtl_timer_device {
        struct hrtl_version version;

        void  (*set)(struct hrtl_timer_device *, unsigned long delta);
        int   (*setup)(struct hrtl_timer_device *,
                       struct hrtl_timer_device_cpu *);
        int   (*shutdown)(struct hrtl_timer_device *);
```

```
        /* -- assigned by the framework -- */
        struct hrtl_entity entity;
        void (*fires)(struct hrtl_timer_device *, unsigned long delta);
};
```

Listing 11.7: Timer device

The `set()` function is called by the timer subsystem to program the time for an interruption. After that time has passed the timer device has to call the `fires()` function.

The standard timer device in the HRTL system can be seen in `timer-lapic.c` in the architecture specific source tree. The timer device uses the local APIC architecture to program interrupts. During the setup process the Linux interrupt handler is replaced by a new HRTL local APIC handler (Listing 11.8).

```
void __irq_entry hrtl_timer_lapic_handler(struct pt_regs *regs) {
...
        add_preempt_count(NMI_OFFSET + HARDIRQ_OFFSET);
        apic_write(APIC_EOI, APIC_EOI_ACK);
        inc_irq_stat(apic_timer_irqs);
        hrtl_lapic_timer.fires(&hrtl_lapic_timer, apic_read(APIC_TMICT));
...
        sub_preempt_count(NMI_OFFSET + HARDIRQ_OFFSET);
        set_irq_regs(old_regs);
}
```

timer-lapic.c

Listing 11.8: HRTL local APIC interrupt handler

A timer in the HRTL system is represented by an instance of `hrtl_timer`. It has to implement at least a callback function `fires()`. The callback is invoked when the timer expires and returns a `hrtl_time` value if the timer should be programmed again or `HRTL_TIME_INVALID`. The timer subsystem handles all timer objects in a per CPU `hrtl_timequeue` sorted by their expiration time. The associated timer device is programmed according the head of the timer queue. If a timer expires, the specified callback function is called and the timer device is programmed for the next timer in the queue.

The two functions `hrtl_timer_start()` and `hrtl_timer_stop()` give access to the timer subsystem. They must be called on the same CPU that should execute (or delete) the timer. The event subsystem (Section 11.3.3) is based on the timer system and provides a more flexible way for programming timed events.

11.3.3. Events

An event (`hrtl_event`) defines a callback function and a trigger. This trigger can either be a call to a special trigger-function, the expiration of a timer or another event (event queue).

Event queue An event can be included in an object of the type `hrtl_eventqueue`. The trigger of this event depends on the trigger of the event queue. The trigger of

an event queue can either be a call to a special trigger-function or a timer. If the trigger for an event queue fires, the associated events are computed according to a given priority (Listing 11.9).

event.h

```
enum hrtl_event_priority {
        __HRTL_EVENT_PRIORITY_FIRST  = 0,
        HRTL_EVENT_PRIORITY_HIGH     = __HRTL_EVENT_PRIORITY_FIRST + 0,
        HRTL_EVENT_PRIORITY_MID      = __HRTL_EVENT_PRIORITY_FIRST + 1,
        HRTL_EVENT_PRIORITY_LOW      = __HRTL_EVENT_PRIORITY_FIRST + 2,
        __HRTL_EVENT_PRIORITY_LAST   = HRTL_EVENT_PRIORITY_LOW,
        __HRTL_EVENT_PRIORITY_NUM    = __HRTL_EVENT_PRIORITY_LAST + 1,
};
```

Listing 11.9: Priorities for events

Event queues are dynamic memory objects that can be created with `hrtl_event-queue_create()`. Some situations in the HRTL system require merging and dividing queues, which will automatically create or destroy event queues (i.e. Section 11.6.1).

Trigger function The two functions `hrtl_event_trigger()` and `hrtl_event-queue_trigger()` signal the occurrence of an event. These functions can only be called for events that are not associated with a timer. If an event (respectively an event queue) has enabled the entity system-call access (Section 11.2.2), the event can be triggered from user space. This will be discussed in Section 11.9.1.

Timer An event (respectively an event queue) can use a timer as trigger source. In this case, an event can be implemented as a periodic event. If the timer for an event fires the timer subsystem will re-queue the timer according to a given time value (the period).

Events provide a wait queue for tasks. A task can register to wait for the occurrence of an event by a call to `hrtl_event_wait()`. The calling task must not hold any locks, since it is going to be suspended. In order to prevent waiting on an event that occurred between releasing a lock and the call to the wait function, a *ticket* for that event must be used in combination with the wait function. The ticket is received inside a protected section (lock) and is valid until the event occurs. If the ticket is not valid while calling `hrtl_event_wait()`, the function just returns since the event had already occurred.

11.4. CPU Reservation

A CPU that is controlled by the HRTL system is called a reserved CPU. It extends the running HRTL partitioning core with an additional resource for task and group scheduling (see Section 11.6). Normally, a task is interrupted regularly by hardware interrupts and the core Linux system. The Linux scheduler can decide to resume or start a different task on each CPU at any time. In order to realise the techniques as

described in Chapter 10, the Linux scheduler and other main system aspects must be disabled or adjusted for reserved CPUs. The reservation process is discussed in this section.

For each reserved CPU a timer device (Section 11.3.2) and a clock source (Section 11.3.1) must be chosen. An instance of the type `hrtl_profile` specifies these properties. One CPU profile can be taken for several reserved CPUs. It is bound to a reserved CPU till the CPU is released.

11.4.1. Per CPU Idle Task

The idle task is scheduled on a reserved CPU if no real-time task can be executed. Each CPU in the system has one defined idle task. For a non-reserved CPU, the idle task is present but never scheduled. During CPU reservation and in the housekeeping phase (Section 10.2.3), the idle task is used to perform some work that must be executed on that specific CPU. Listing 11.10 shows the main loop of the task.

idle_task.c

```
static int hrtl_idle_task_fn(void *data)
...
        while (!kthread_should_stop()) {
                preempt_disable();
                local_irq_save(flags);
...
                if (unlikely(idle_task->flags & HRTL_IDLE_TASK_SETUP)) {
...
                        hrtl_system_cpu_setup();
                } else if (idle_task->flags & HRTL_IDLE_TASK_PERIOD) {
...
                        hrtl_housekeeping_run();
                }

                local_irq_restore(flags);

                if (unlikely(idle_task->flags & HRTL_IDLE_TASK_IRQ)) {
...
                        hrtl_irq_run();
                        set_tsk_need_resched(current);
                }

                while (!need_resched())
                        cpu_relax();
...
                preempt_enable_no_resched();
                schedule();
        }
...
}
```

Listing 11.10: Idle task main loop

An idle task is a member of the HRTL scheduling class (Section 11.5) and bound to a certain CPU. Unlike other tasks in this scheduling class, the idle task does not

belong to any partition. The behavior of the idle task is managed by three different flags (defined in `hrtl/system/system.h`):

`HRTL_IDLE_TASK_PERIOD` This flag is set in the idle phase of a scheduling group. Depending on the housekeeping object[1] different housekeeping tasks are scheduled. Housekeeping tasks are arranged in event queues. The function `hrtl_housekeeping_run()` triggers these queues according to the housekeeping object.

`HRTL_IDLE_TASK_SETUP` This flag is only set during the CPU reservation process. It leads to a call to `hrtl_system_cpu_setup()` where the current CPU state is changed (Section 11.4.2). Since an idle task is bound to a certain CPU, the setup function is executed on that CPU.

`HRTL_IDLE_TASK_IRQ` This flag invokes the idle task to execute an interrupt handler. Threaded interrupt handlers are discussed in Section 11.6.3.

The idle for a specific CPU is chosen for execution if no real-time task can be scheduled. For instance, if the current static scheduling group has no active task in the ready queue. When the current time slice in the scheduling plan is assigned to a system group or a dynamic partition, the idle task is scheduled only if Linux tasks are not activated (Section 11.5). The idle task will be scheduled if the related CPU is set in the CPU mask `hrtl_cpus_reserved` and is not set in the mask `hrtl_cpus_linux`.

11.4.2. CPU States

At any time, each CPU in the system is in one of the states as shown in Table 11.1. The table describes an action that is performed in every state. The column *CPU* indicates which CPU executes the action: system CPU (*sys*), CPU that is going to be reserved/released (*res*). A state change means going to the next state (the next row). The next state for `HRTL_SYSTEM_CPU_FREED` is `HRTL_SYSTEM_CPU_UNUSED`.

State actions are managed by event queues. Entering a state triggers the appropriate queue and executes the included events. The events are executed with three different priority levels (Section 11.3.3).

An instance of `hrtl_cpu_setup_call` defines a certain state action. As can be seen in Listing 11.11 a callback function is defined together with a state and a priority level. The macro `hrtl_cpu_setup_initcall` can be used to define a state action during the system boot process.

cpu.h

```
enum hrtl_cpu_setup_prio {
        HRTL_CPU_SETUP_PRE      = HRTL_EVENT_PRIORITY_HIGH,
        HRTL_CPU_SETUP_POST     = HRTL_EVENT_PRIORITY_LOW,
        HRTL_CPU_SETUP_SYSTEM   = HRTL_EVENT_PRIORITY_MID,
};
```

[1] The housekeeping objects are not explained here. See Section 10.2.3 for more details.

```
struct hrtl_cpu_setup_call {
        enum hrtl_cpu_state state;
        enum hrtl_cpu_setup_prio prio;
        void (*call)(struct hrtl_cpu *);

        /* -- assigned by the framework -- */
        struct hrtl_event event;
};

#define hrtl_cpu_setup_initcall(_c)                                    \
    static int __init __hrtl_hrtl_cpu_setup_initcall_##_c (void) {     \
        hrtl_printk_trace_fkt_boot();                                  \
        WARN_ON(0 > hrtl_cpu_setup_call_register(&(_c))); return 0; }; \
    hrtl_initcall_late(__hrtl_hrtl_cpu_setup_initcall_##_c)
```

Listing 11.11: CPU setup calls

The system CPU hosts a special worker thread which can be triggered to execute a specified work package on that CPU. The worker thread is bound to the system CPU and not included in the HRTL scheduling class. Together with the per CPU idle tasks the state actions described can be scheduled on different CPUs.

State	CPU	Action
HRTL_SYSTEM_CPU_UNUSED	n/a	*non-reserved*
HRTL_SYSTEM_CPU_PREPARATION	sys	clear CPU from system cpus
HRTL_SYSTEM_CPU_SETUP	res	setup devices (CPU profile)
HRTL_SYSTEM_CPU_COMPLETION	sys	prepare Linux subsystems
HRTL_SYSTEM_CPU_RESERVED	res	start into static scheduling plan
HRTL_SYSTEM_CPU_EVALUATING	sys	mark CPU as *free*
HRTL_SYSTEM_CPU_SHUTDOWN	res	stop running scheduling plan
HRTL_SYSTEM_CPU_CLOSING	sys	prepare Linux subsystems (release)
HRTL_SYSTEM_CPU_FREED	res	shutdown devices (CPU profile)

Table 11.1.: CPU states

11.4.3. CPU Takeover

The function `hrtl_system_reserve()` initiates the CPU reservation process (Listing 11.12). If no specific CPU is defined in parameter `rcpu` the next free CPU is taken and removed from `hrtl_cpus_free`. `hrtl_cpu_get_cpu()` returns a reference to the CPU's status object (`hrtl_cpu`) which includes the current CPU state. Only one CPU can be in the reservation process at a time. In order to exclude other CPUs from entering the reservation process while another CPU is in this phase, the calling CPU requests a *ticket* (token). The ticket can only be given to one CPU and must be

returned when the reservation process is completed (or has failed). Besides mutual exclusion the ticket manages various status information during the reservation process. `hrtl_cpu_reserve_ticket_wait()` blocks the calling task until the ticket is available.

system.c

```
int hrtl_system_reserve(int rcpu, hrtl_key_t profile_key,
                        hrtl_id_t core_id, hrtl_id_t sync_core_id)
{
...
        cpu = hrtl_system_take_free_cpu(rcpu);
...
        ccpu = hrtl_cpu_get_cpu(cpu);
        hrtl_cpu_reserve_ticket_wait(ccpu);

        retval = hrtl_profile_prepare_setup(cpu, profile_key);
...
        retval = hrtl_core_prepare_setup(cpu, core_id, sync_core_id);
...
        WARN_ON(hrtl_cpu_ticket_prepare_step(HRTL_SYSTEM_CPU_PREPARATION,
                                             HRTL_SYSTEM_CPU));
        hrtl_sys_worker_trigger();

        if (0 == hrtl_cpu_wait_state(ccpu, HRTL_SYSTEM_CPU_RESERVED)) {
...
        }

        hrtl_cpu_release_ticket(ccpu);
...
}
```

Listing 11.12: CPU reservation

After the ticket is assigned, the given CPU profile and scheduling plan (core) are prepared for the reservation process. This includes mainly a check of the permissions of the calling task. In this step, the *cycles* of the scheduling plan are put together (Section 11.6.1).

`hrtl_cpu_ticket_prepare_step()` defines the next state and the CPU that should execute the related state action events. Since the *preparation* state is entered from the system CPU, the worker thread needs to be activated. Details on the worker thread are not discussed here. The calling task is blocked until the CPU is reserved or the reservation process has failed. Finally, the ticket which was previously acquired is released.

The worker thread realises that the preparation state is defined for a CPU and calls `hrtl_system_cpu_state()`. This function is invoked for every state change from different tasks running on their specified CPU. The implementation of `hrtl_system_cpu_state()` triggers the according event queues for a prepared state. After executing some state related actions the next state is prepared. As can be seen in Listing 11.13 no next state is prepared for the *reserved* state.

system.c

```
static void hrtl_system_cpu_state(struct hrtl_cpu *ccpu,
                                  enum hrtl_cpu_state state)
```

```
{
...
        WARN_ON(hrtl_cpu_ticket_achieved_step(state));

        switch (state) {
        case HRTL_SYSTEM_CPU_PREPARATION:
                hrtl_warn_on(hrtl_cpu_reserved(cpu));
                hrtl_cpu_ticket_prepare_step(state +1, cpu);
                cpumask_set_cpu(cpu, hrtl_cpus_reserved);
                break;

        case HRTL_SYSTEM_CPU_SETUP:
                hrtl_migrate_linux_tasks();
                hrtl_cpu_ticket_prepare_step(state+1, HRTL_SYSTEM_CPU);
                hrtl_sys_worker_trigger();
                break;

        case HRTL_SYSTEM_CPU_COMPLETION:
                hrtl_cpu_ticket_prepare_step(state +1, cpu);
                idle_task = hrtl_cpu_get_idle_task(cpu);
                hrtl_idle_task_set_setup(idle_task);
                set_tsk_need_resched(hrtl_idle_task_get_task(idle_task));
                break;
...
        case HRTL_SYSTEM_CPU_UNUSED:
        case HRTL_SYSTEM_CPU_RESERVED:
                break;
...
        }
}
```

Listing 11.13: CPU state machine

In the preparation step the CPU is activated in the `hrtl_cpus_reserved` mask. The periodic Linux timer interrupt checks if the current CPU is enabled in this mask. On a reserved CPU the timer interrupt calls the function `hrtl_system_takeover()` and disables itself (see Section 11.4.4 for details). The *takeover* function disables the CPU from the `hrtl_cpus_linux` mask and sets the `HRTL_IDLE_TASK_SETUP` flag for the idle task.

11.4.4. Necessary Adjustments

The Linux timer interrupt must be extended for the CPU reservation process in order to call the `hrtl_system_takeover()` function. Furthermore, the periodic Linux timer should be disabled so that the real-time scheduling runs without further interruptions on a reserved CPU. Listing 11.14 shows the necessary adjustments for the Linux time tick system.

tick-sched.c

```
static enum hrtimer_restart tick_sched_timer(struct hrtimer *timer)
{
...
#if defined(CONFIG_NO_HZ) || defined(CONFIG_HRTL)
```

```
   if (unlikely(tick_do_timer_cpu == TICK_DO_TIMER_NONE))
      tick_do_timer_cpu = cpu;
#endif
...
#ifdef CONFIG_HRTL
        if (unlikely(hrtl_system_takeover())) {
                if (tick_do_timer_cpu == cpu)
                        tick_do_timer_cpu = TICK_DO_TIMER_NONE;

                return HRTIMER_NORESTART;
        }
#endif
   hrtimer_forward(timer, now, tick_period);
   return HRTIMER_RESTART;
}
```

Listing 11.14: Linux periodic interrupt

The Linux event handler for periodic ticks is based on high resolution timers. In order to stop the periodic timer for a CPU the callback function just has to return `HRTIMER_NORESTART`. `hrtl_system_takeover()` returns a value other than zero in case the calling CPU is set in the `hrtl_cpus_reserved` mask. If the timer interrupt for the calling CPU should be disabled and the CPU is responsible for jiffies updates, the `tick_do_timer_cpu` variable is released and will be taken by any other non-reserved CPU in the system.

As described in Section 10.2.1 APIC inter processor interrupts (IPI) must be intercepted and handled in a special way. An IPI call to a reserved CPU is taken to a queue and handled during one of the housekeeping phases on that CPU. Listing 11.15 shows the necessary adjustment for the Linux IPI call API.

smp.h

```
static inline void arch_send_call_function_single_ipi(int cpu)
{
#ifdef CONFIG_HRTL
        if (hrtl_cpu_reserved(cpu))
                hrtl_cpu_call_function_single_ipi(cpu);
        else
#endif
   smp_ops.send_call_func_single_ipi(cpu);
}

static inline void arch_send_call_function_ipi_mask(const struct cpumask
     *mask)
{
#ifdef CONFIG_HRTL
        int cpu;
        cpumask_t linux_mask;

        cpumask_clear(&linux_mask);

        for_each_cpu(cpu, mask)
                if (hrtl_cpu_reserved(cpu))
                        hrtl_cpu_call_function_ipi(cpu);
```

```
                else
                        cpumask_set_cpu(cpu, &linux_mask);

        if (!cpumask_empty(&linux_mask))
                smp_ops.send_call_func_ipi(&linux_mask);
#else
   smp_ops.send_call_func_ipi(mask);
#endif
}
```

Listing 11.15: Linux IPI API

Since the Linux periodic timer interrupt is disabled on reserved CPUs, the Linuc timer subsystem (including HR-timer) and the RCU mechanism need to be extended. Timers and RCU are normally processed by the interrupt handler of the periodic tick. If the Linux handler is completely disabled, timers will not work any more.

Linux timers

```
static struct hrtl_cpu_setup_call hrtl_timer_pull_call = {
        .state = HRTL_SYSTEM_CPU_COMPLETION,
        .prio = HRTL_CPU_SETUP_PRE,
        .call = hrtl_timer_pull, };
```

timer.c

```
static struct hrtl_cpu_setup_call hrtl_hrtimer_pull_call = {
        .state = HRTL_SYSTEM_CPU_COMPLETION,
        .prio = HRTL_CPU_SETUP_PRE,
        .call = hrtl_hrtimer_pull, };
```

hrtimer.c

Listing 11.16: Linux timer setup

Timers which have already been activated, running on a reserved CPU, must be moved away when the Linux timer interrupt is deactivated. The two setup callbacks shown in Listing 11.16 are responsible for timer migration. The implementation is not shown here. Pending timers are just removed from the timer queue and append to the queue hosted by the system CPU. This requires some additional locking variables, since a queue may be altered from different CPUs simultaneously.

New timers must be activated on a different CPU if the Linux timer interrupt is deactivated. When a new timer should be started on a reserved CPU, the timer base is changed to the timer base of the system CPU.

Read-copy update

```
static struct hrtl_cpu_setup_call hrtl_move_rcu_callbacks_call = {
        .state = HRTL_SYSTEM_CPU_COMPLETION,
        .prio = HRTL_CPU_SETUP_PRE,
        .call = hrtl_move_rcu_callbacks, };
```

rcutree.c

Listing 11.17: Linux RCU setup

The per CPU queue of pending finished RCU operations must be moved away when a CPU becomes reserved. New RCU callbacks must be enqueued on a non-reserved

CPU (system CPU). The setup callback shown in Listing 11.17 manages RCU migration. The implementation uses the Linux CPU hotplug mechanism.

Since the Linux timer interrupt is disabled the function `rcu_check_quiescent_state()` is never called and it is not detected that the CPU is in a quiescent state. Instead, `hrtl_quiescent_state()` signals that a reserved CPU has passed such a state. It is actually a wrapper function for `rcu_check_quiescent_state()` and is invoked during the housekeeping phase.

Pending tasklets and work packages for a reserved CPU must be moved to another CPU (actually the system CPU). Tasklets are implemented via two SoftIRQs. Since SoftIRQs are also directed to an unreserved CPU, new tasklets must be enqueued on that CPU. SoftIRQs and workqueues are processed by a special kernel thread which is not scheduled on a reserved CPU. The Linux SoftIRQ, tasklet and workqueue mechanism need to be modified in several places. The implementation is always similar to the already introduced timer and RCU modifications. New work packages for a reserved CPU are redirected to the system CPU. Pending work is migrated when a CPU becomes reserved. In all cases additional locking mechanisms are needed, which brings high variety of code changes. The migration of pending packages is introduced via `hrtl_cpu_setup_call` objects to the CPU reservation process. The migration implementations are mainly based on the Linux CPU hotplug mechanism, which already enables Linux to migrate pending packages between CPUs.

A reserved CPU has to be excluded from the load balancing of the Linux scheduler. On the one hand, a reserved CPU must not pull processes from other run-queues, even if these are much longer than its own run-queue. On the other hand, real-time tasks must not be pulled from the run-queue of a reserved CPU to another CPU. `load_balance()` and the `find_idlest_cpu()`/`find_busiest_cpu()` functions need to be prepared to ignore CPUs which are not included in the `hrtl_cpus_linux` mask. Linux tasks are moved from a reserved CPU by the HRTL system during the housekeeping phase.

When waking up sleeping tasks, care must be taken to put them on the right CPU. `select_task_rq()` is called from the wake-up mechanism to determine a CPU for a task. Non-real-time tasks are never woken up on reserved CPUs (respectively if the CPU is not included in the `hrtl_cpus_linux` mask).

core.c

```
static inline
int select_task_rq(struct task_struct *p, int sd_flags, int wake_flags)
{
      int cpu = p->sched_class->select_task_rq(p, sd_flags,wake_flags);
...
      if (unlikely(p->policy != SCHED_HRTL)) {
              hrtl_warn_on(!irqs_disabled());
              read_lock(hrtl_cpus_linux_lock);
              cpumask_and(&allowed,tsk_cpus_allowed(p),hrtl_cpus_linux);
              read_unlock(hrtl_cpus_linux_lock);

              if (unlikely(cpumask_empty(&allowed)))
```

```
                    cpumask_copy(&allowed,cpumask_of(HRTL_SYSTEM_CPU));
      } else
             cpumask_copy(&allowed, tsk_cpus_allowed(p));
...
}
```

Listing 11.18: Select task run-queue

11.5. Scheduling Class (SCHED_HRTL)

A new scheduling class is introduced by the HRTL extension. Each real-time task in the system is a member of that scheduling class. A task can switch to the `SCHED_HRTL` policy by calling the `sched_setscheduler()` system call. The function is extended by the patch and supports the new scheduling class.

`SCHED_HRTL` has the highest priority assigned among all scheduling classes except the *stop-task* class.[2] The macro `for_each_class()` iterates over all classes. Scheduling classes are linked in a list where each class points to the next lower priority class.

stop_task.c

```
const struct sched_class stop_sched_class = {
#ifdef CONFIG_HRTL
        .next                   = &hrtl_sched_class,
#else
        .next                   = &rt_sched_class,
#endif
...
};
```

hrtl.c

```
const struct sched_class hrtl_sched_class = {
        .next                   = &rt_sched_class,
...
};
```

Listing 11.19: Scheduling classes list

A detailed description of the Linux scheduler can be found in [?] and [?].

11.5.1. Linux Scheduler Integration

The main scheduler function `__schedule()` picks up the highest priority task from a scheduling class and switches to the memory segment and thread's register state of the new task. The classes are processed according to their priorities. If a class has no runnable task, the next class in the list is considered. A scheduling class has to implement a `pick_next_task()` function.

hrtl.c

```
static struct task_struct *
pick_next_task_hrtl(struct rq *rq)
{
```

[2]The stop task is the highest priority task in the system, it preempts everything and will be preempted by nothing.

```
        int cpu = smp_processor_id();
        struct task_struct *next;

        /* first, check if an interrupt is pending */
        if (unlikely(NULL != (next = hrtl_curr_interrupt_task())))
                return next;

        /* then, try to get an HRTL task */
        if (likely(NULL != (next = hrtl_curr_get_next_task())))
                return next;

        /* return the idle task if Linux is not welcome */
        if (likely(!hrtl_cpu_dyn(cpu)))
               return hrtl_idle_task_get_task(hrtl_curr_get_idle_task());

        /* get a dynamic group task */
        if (likely(NULL!=(next = hrtl_budget_sched_pick_next_task(rq)))){
                hrtl_budget_run_hrtl(hrtl_task_budget_index(next));
                hrtl_cpu_clear_linux(cpu);
                return next;
        }

        /* finally, Linux can schedule a task */
        hrtl_budget_run_linux();
        hrtl_cpu_set_linux(cpu);
        return NULL;
}
```

Listing 11.20: Pick next task from `SCHED_HRTL`

Listing 11.21 shows the `pick_next_task()` function of the HRTL scheduling class. Calling this function on a non-reserved CPU will return `NULL` (which will cause the main scheduler function to switch to the next scheduling class). `hrtl_curr_get_next_task()` is a link to the currently running time-slot of the static scheduling plan which will be discussed in Section 11.6. On a non-reserved CPU or a CPU currently running a system group the function always returns `NULL`.

The concrete implementation of the `pick_next_task()` function and other functions of the scheduling class depend on the current HRTL scheduling module (Section 11.6.4) respectively the dynamic group scheduler (Section 11.7.3).

11.5.2. Adding Tasks to `SCHED_HRTL`

Static partitions are assigned to a certain CPU. If a task is added to a group, it must be ensured that this task is located on the same CPU. Since running tasks can not migrate to another CPU, it must be suspended and woken up on the right CPU. The HRTL scheduling class member function `switched_to_hrtl()` is called from `sched_setscheduler()` after a task was connected with a static partition. If the current task's CPU and the partition's CPU are different, the task is added to a *pull* queue inside the partition and the task's `resched` flag is set. The `resched` flag will cause the main scheduler function to choose another task for execution

on the original task CPU. The resulting call of the HRTL scheduling class member function `put_prev_task_hrtl()` will signal the partition's CPU that a task switch is needed (in case the partition is running and the currently running task has a lower priority level). The actual task CPU migration is done by `pre_schedule_hrtl()` which gets called by the main scheduler function on the partition's CPU. A comparable technique will be discussed in Section 11.7.2 for the dynamic partitions balancing algorithm.

The HRTL scheduling class member function `task_fork_hrtl()` is called by the Linux task creation mechanism if a task included in the HRTL policy calls the `fork()` system call. The function chooses a partition where the new task should be placed in. A partition for a newly created task can be defined at three different places. The calling task itself can decide where the new task should be spawned. If the task does not define a partition, the group of the calling task and then the group's CPU profile are considered. If none of these places define a valid partition where the new task can be placed in, the new task is added to the `SCHED_NORMAL` Linux class.

11.5.3. Necessary Adjustments

The main scheduler function (especially the sub routine `pick_next_task()`) needs to be modified in order to integrate the HRTL scheduling class. `pick_next_task()` implements an optimisation in the case that all running tasks are connected with the *fair* scheduling class. This optimisation is disabled in a HRTL patched kernel.

core.c

```
static inline struct task_struct *
pick_next_task(struct rq *rq)
{
...
#ifndef CONFIG_HRTL
...
        if (likely(rq->nr_running == rq->cfs.h_nr_running)) {
                p = fair_sched_class.pick_next_task(rq);
                if (likely(p))
                        return p;
        }
#endif
        for_each_class(class) {
                p = class->pick_next_task(rq);
                if (p)
                        return p;
        }
...
}
```

Listing 11.21: Pick next task

The system call `sched_setscheduler()` is modified in order to handle the new HRTL scheduling class. `SCHED_HRTL` is now a valid class and can be assigned to a task in combination with a scheduling group. The group must be able to incorporate the task or the function will return with an error. `sched_setscheduler()` expects

an object of `sched_param` in order to set the tasks priority level.[3] The HRTL patch adds an additional field to that structure in order to specify a partition.

11.6. Static Scheduling Plan

A static scheduling plan consists of on `hrtl_core` object and one or more `hrtl_group` objects. Partitions are represented by (static) groups and include an execution and an idle part (Section 10.3.1). During the CPU reservation process, a scheduling plan is chosen and exclusively connected to that CPU. Partitions can be added to a plan offline (before it is connected to a CPU) and at runtime.

11.6.1. Cycles

A time slot is represented by an instance of `hrtl_cycle`. It has a start and a stop event and a defined runtime. Three callback functions need to be implemented for each cycle object:

start(), stop() are called when the relevant time-slot begins and respectively ends.

handler() is called with different actions (`hrtl_cycle_handler_action`) during the setup phase (reserve CPU) and the shutdown phase (release CPU).

A cycle can contain other cycles. The function `hrtl_cycle_hook()` inserts a cycle (and all included cycles) into a root cycle at a given time offset. Two event queues are created for each of the two cycles, one for each start and stop event. Stop events have a higher priority than start events in a event queue.

cycle.c

```
#define HRTL_CYCLE_START_EVENT  HRTL_EVENT_PRIORITY_MID
#define HRTL_CYCLE_STOP_EVENT   HRTL_EVENT_PRIORITY_HIGH

static int __hrtl_cycle_setup_queue(struct hrtl_cycle *cycle,
                                    struct hrtl_eventqueue *queue,
                                    enum hrtl_event_priority prio)
{
...
        if (HRTL_CYCLE_START_EVENT == prio)
                event = &cycle->event_start;
...
        return hrtl_event_attach_queue(event, queue, prio);
}

static int hrtl_cycle_ensure_queues(struct hrtl_cycle *cycle)
{
...
        if (NULL == hrtl_event_get_queue(&cycle->event_start)) {
```

[3] This priority level is only valid for Linux tasks. Real-time tasks atributes are adjusted with special functions and are discussed in Section 11.9.1.

```
                queue = __hrtl_cycle_alloc_queue(HRTL_CYCLE_START_EVENT);
                if (IS_ERR(queue))
                        return PTR_ERR(queue);

                retval = __hrtl_cycle_setup_queue(cycle, queue,
                                                  HRTL_CYCLE_START_EVENT);
...
        }

        if (NULL == hrtl_event_get_queue(&cycle->event_stop)) {
...
}
```

Listing 11.22: Create queues for cycle events

Each event queue will be triggered by a timer. Two event queues that are triggered at the same time are joined together. Since stop events have higher priorities than start events, stop events are executed before start events if a queue is triggered by a timer. A root cycle that does not contain any sub cycles has only one event queue with one timer at runtime. A root cycle that contains two sub cycles, for instance an execution and an idle time-slot of a partition, has two event queues at runtime.

`hrtl_cycle_start()` starts a defined root cycle with all included sub cycles. During the startup process all timers for the included event queues are programmed. Each timer for any event queue is defined as a periodic timer. The period time is the runtime of the root cycle. After a cycle has started it can not be added as a sub cycle to another cycle. A cycle which has started can still incorporate other cycles.

The function `hrtl_cycle_setup()` is called for each cycle that is included in the root cycle during the startup process. The runtime of the root cycle and other time values are calculated and stored for every cycle. Thus, determining the current period time is easy. `hrtl_cycle_get_period()` (which calls `__hrtl_cycle_get_period()`) translates a given time to the period time format (`hrtl_period_t`) as described in Section 11.1.

cycle.c

```
static hrtl_period_t __hrtl_cycle_get_period(struct hrtl_cycle *cycle,
                                             hrtl_time_t time)
{
...
        time -= cycle->calc.started;
        ret.runtime = cycle->runtime;
        ret.val.time = time % cycle->calc.runtime;
        ret.val.count = time / cycle->calc.runtime;
...
        return ret;
}

static int hrtl_cycle_setup(struct hrtl_cycle *root)
{
...
        if (root->flags & HRTL_CYCLE_INCLUDED) {
                parent = root->included.cycle;
```

```
		if (HRTL_TIME_INVALID == root->calc.started) {
			root->calc.started = parent->calc.started;
			root->calc.started += root->included.offset;
		}

		root->calc.offset = parent->calc.offset;
		root->calc.offset += root->included.offset;
		root->calc.runtime = parent->calc.runtime;
		root->cpu = parent->cpu;
	} else {
		hrtl_warn_on(HRTL_TIME_INVALID == root->calc.started);
		root->calc.runtime = root->runtime;
		root->calc.offset = 0;
	}
...
}
```

Listing 11.23: Get current period time of running cycle

11.6.2. Partition Management

Time-slots of static partitions are represented as cycles. Each partition has one root cycle including one cycle for the execution time-slot and one for the idle time-slot (the idle time-slot is optional). A core object has the same time-slot structure. The execution time-slot of a core objects includes the cycles of the associated partitions.

The beginning of a time-slot of a partition is signaled by `hrtl_group_start()`. The function is associated with the partition's root cycle as the start callback. `hrtl_group_start()` registers the partition at the HRTL scheduler (`hrtl_curr-_get_next_task()`, Section 11.5.1) and sets the resched flag for the currently running thread. `hrtl_group_start()` is called in interrupt context (timer interrupt). Thus, the resched flag will invoke the main scheduler immediately when returning from interrupt. Since the partition has changed on that CPU, the next task will be picked from the new (active) partition.

At any time in the scheduling plan for a reserved CPU, exactly one partition is activated. The space between time-slots of partitions in a core execution cycle is filled by system groups. Two directly adjacent system groups will be combined to one system group. Depending on the HRTL setup the running CPU is added to the `hrtl_cpus_linux` mask when a system group starts. In the idle phase, active Linux tasks are migrated away from the running CPU to the system CPU.

11.6.3. Interrupt Handlers

The per CPU idle thread is used to execute interrupts in task context. An interrupt is split into a hardware handler (`hrtl_hw_irq`) and a threaded handler (`hrtl_irq`). An `hrtl_irq` object can be connected with a partition or a reserved CPU (core). If the handler is assigned to a partition, it will only be executed inside the allotted time-slot. If no hardware handler is defined for an interrupt, the standard HRTL

handler is used to enqueue the threaded handler at the defined position (core or partition). The occurrence of an interrupt will preempt the running task (if the defined time-slot is active) and execute the idle thread.

11.6.4. Interface for Scheduler Modules

The task scheduling for a static partition is realised by scheduler modules. The modules can be implemented as Linux loadable modules and can be registered and deregistered at system runtime. The HRTL system includes four example implementations of scheduler modules:

`HRTL_SCHED_SINGLE` An easy scheduling class that deals with only one running task or none. As soon as a task is added to this class the group closed flag is set. This means that no more tasks can by added by the system. The flag is deleted by removing the previously added task.

`HRTL_SCHED_RM` Rate-monotonic scheduling. This class requires each task to define a computation time and a deadline. According to the descriptions in Section 3.2.2.2 and Section 10.4.3 the task with the shortest deadline is selected for execution.

`HRTL_SCHED_POSIX` FIFO and round-robin scheduling. Provides priority based scheduling with additional timeslices. This scheduling variants were discussed in Section 3.2.3 and Section 10.4.3.

`HRTL_SCHED_SYSTEM` This class can not be selected for user defined partitions. It is assigned to every system group. Interactions with dynamic partitions are discussed in Section 11.7.3.

The API for scheduler modules defines several callback functions that must be implemented by a module. They are part of the `hrtl_sched` structure witch also includes an `hrtl_entity` reference. Once registered, a scheduler module can be selected for a static partition.

`int assign()` The scheduler module is assigned to a partition. Necessary memory segments can be allocated and initialised in this function. A return value other than zero signals that the module can not be connected to the given partition. No tasks are included in the partition when this function is called.

`release()` The scheduler module is disconnected from a partition. At this time, the partition is not running and does not include any tasks. Previously allocated memory segments can be released by this function.

`int add()` A thread should be added to a partition. The partition may not be assigned to a core object. A return value other than zero signals that the thread can not be added to the partition.

del() A thread is removed from a partition. The partition may not be assigned to a core object.

periodic() This function is called in several situations at defined times. It receives a parameter *expected* (`hrtl_period_t`) witch stores the time value when the event was planed.[4] The following events are defined:

- **HRTL_SCHED_PERIODIC_SETUP, HRTL_SCHED_PERIODIC_SHUTDOWN** The partiton starts or stops running, respectively the according core is assigned to (removed from) a reserved CPU or the partition is added to a scheduling plan that is already running.
- **HRTL_SCHED_PERIODIC_START, HRTL_SCHED_PERIODIC_END** This events signals start and stop of the time-slot that is connected with the partition.
- **HRTL_SCHED_PERIODIC_TICK** A defined timer that was programmed by the scheduler module. A timer can be adjusted by `hrtl_group_set_periodic()`, `hrtl_group_forward_periodic()` and `hrtl_group_clear_periodic()`.

Apart from the setup and shutdown events, `periodic()` is always called in interrupt context.

check_preempt() This function checks if a task that entered the runnable state should preempt the task which is currently running.

get_next_task() This function chooses the most appropriate task eligible to run next. If the function returns `NULL` either the idle task or a normal Linux task will be scheduled (depending on the `hrtl_cpus_linux` mask).

yield() The running task performed a `yield()` system call. The HRTL system provides the `HRTL_TASK_YIELD_CYCLE` flag that indicates that a task should be suspended until the next period (time-slot) starts.

enqueue() Called when a task enters a runnable state.

dequeue() When a task is no longer runnable, this function is called.

prio_changed(), runtime_changed(), period_changed(), flags_changed() Task attributes have changed.

The Linux task structure is extended by some variables in order to allow management of assigned tasks in a partition. Thus, no extra memory blocks need to be allocated if a task should be placed in a queue by a module.

sched.h

```
struct sched_hrtl_entity {
...
        struct hrtl_group *group;
...
        unsigned int prio;
```

[4] The current time is available by `hrtl_curr_get_period()`.

```
        hrtl_time_t runtime;
        unsigned int period;
...
        unsigned int flags;
...
        /* HRTL schedule class related data */
        struct {
                unsigned int flags;
                hrtl_time_t slice;
                hrtl_period_t started;
                hrtl_period_t timer;
                struct list_head list;
                struct hrtl_timequeue_node node;
        } class;
};

struct task_struct {
...
#ifdef CONFIG_HRTL
        struct sched_hrtl_entity hrtl;
#endif
...
};
```

Listing 11.24: HRTL task structure

The `HRTL_TASK_DEADLINE` flag can be defined for a thread to indicate that the thread is running in *periodic mode*. The period is defined by the runtime value in the `hrtl_sched_param` structure. A deadline is met by executing a `yield()` system call within the specified time. The task is suspended until the next period (runtime value) starts. A sleep does not signal that the task has met its deadline. A sleeping task can miss its deadline.

The scheduler module can make use of the functions `hrtl_task_deadline_define()` and `hrtl_task_deadline_undefine()` to signal that a deadline for a task was defined or undefined. The functions `hrtl_task_met_deadline()` and `hrtl_task_missed_deadline()` signal that a deadline was met or missed. In case a deadline is missed the corresponding task will receive a POSIX signal (defined by `HRTL_SIGDEADLINE`) if the `HRTL_TASK_SIGNALS` flag is set for the task.

11.6.5. Deadline Events

Deadline events are collected for all tasks running in periodic mode. These events are available for user space applications (*deadline watchdog*). A deadline watchdog must be registered at a profile in order to receive deadline events. Listing 11.25 the events that are available.

types.h

```
enum hrtl_deadline_info_event {
        HRTL_DEADLINE_DEFINE,
        HRTL_DEADLINE_UNDEFINE,
        HRTL_DEADLINE_MISSED,
        HRTL_DEADLINE_MET,
```

```
};

struct hrtl_deadline_info {
        enum hrtl_deadline_info_event event;
        pid_t pid;
        hrtl_time_t runtime;
        hrtl_time_t now;
};
```

Listing 11.25: HRTL task structure

Two variables containing time information are included in a deadline info block. They indicate when a deadline was defined (`runtime`) and when a task signaled that a deadline was met (`now`).

Deadline events are stored in lockless ring buffers. Each CPU defines its own deadline event ring buffer, thus only one writer exists for each buffer. No locking mechanism is needed when an event is put into a buffer, since the commit can not be interrupted by another CPU.

A deadline watchdog needs to request new deadline events. Delivering a POSIX signal is not possible, because the sending task (respectively interrupt handler) may block on a spin-lock while the signal is enqueued.

11.7. Balancing Dynamic Partitions

According to the description in Section 10.4.4 dynamic partitions are scheduled in the time-slots provided by system groups. The HRTL system allows the definition of seven (`HRTL_BUDGET_MAX_GROUP`) dynamic partitions plus one for the Linux system. Runtime information of partitions are managed in an array, thus the maximum number of partitions must be a constant.[5] The update rate and length of the time window can be configured with the `HRTL_BUDGET_WINDOW_SIZE` and `HRTL_BUDGET_WINDOW_UPDATE` constants.

11.7.1. CPU Usage Measurement

The HRTL dynamic partition scheduler maintains two different time windows. One for reserved CPUs and one for unreserved CPUs. Each dynamic partition has a guaranteed minimum percentages of available CPU time allotted for reserved CPUs and a maximum share of time on unreserved CPUs (see also Section 10.3.1). A time window is represented by an object of `budget_window`.

budget.h

```
#define __HRTL_BUDGET_MAX_GROUP   (HRTL_BUDGET_MAX_GROUP * 8)
#define __HRTL_BUDGET_GROUP_LONG BITS_TO_LONGS(__HRTL_BUDGET_MAX_GROUP)

struct budget_window_element {
        unsigned int slices;
        unsigned int groups[__HRTL_BUDGET_MAX_GROUP];
};
```

[5]Dynamic memory allocation for partitons would require some locking mechanisms.

```
struct budget_current_frame {
        cpumask_t budget_cpus;
        unsigned long budget_groups[NR_CPUS][__HRTL_BUDGET_GROUP_LONG];
};

struct budget_window {
        void (*update)(struct budget_window *window,
                       struct hrtl_budget *budget,
                       unsigned int new, unsigned int old);
        void (*frame_init)(struct budget_window *window,
                           struct budget_current_frame *frame);

        raw_spinlock_t lock;
        struct budget_current_frame frame;
        unsigned int pos;
        unsigned int slices;
        struct budget_window_element window[HRTL_BUDGET_WINDOW_SIZE];
};
```

Listing 11.26: Time window

The actual part of a window that moves forward as time advances is `frame` (`budget_current_frame`). A window update (moving the frame) is performed by a (Linux) timer running on the system CPU every `HRTL_BUDGET_WINDOW_UPDATE` microseconds (default is 5000). A frame defines a bit mask for CPUs being available for scheduling between two window updates and a bit mask for the partitions that were running in that time. A time window stores frames for the last `HRTL_BUDGET_WINDOW_SIZE` window updates (default is 80[6]). On a window update, the usage for this frame is added to the usage for previous `HRTL_BUDGET_WINDOW_SIZE` -1 frames to compute the total CPU usage over the time window. Furthermore, a new slot in the `window` array is allocated for the next frame. The outdated window slot is subtracted from the total CPU usage. The new frame inherits the bit mask of available CPUs from the previous frame. The currently running partition on each available CPU is marked as running in the bit mask for active partitions.

Each CPU that is available for scheduling during a frame extends the number of `slices` by one. A slice is distributed in equal parts to the partitions running on that CPU. The number of used slices for a partition is stored in the partitions data object. The total CPU usage for a partition in a time window is given by the percentage of used slices of this partition and the number of available slices during the time window.

11.7.2. Group Distribution

The HRTL dynamic partition scheduler maintains two different queues for available time slices (for reserved and non-reserved CPUs). Time slices are queued according to the priority of the currently running task in that slice. The slice running the task with

[6] The default window length is 400 milliseconds: $5000\mu s \cdot 80$.

the lowest priority is enqueued at the head of the queue. If the task's CPU budget is depleted the priority is lowered so that tasks from partitions with valid CPU budget are always enqueued after partitions that have overused their CPU budgets. If a task becomes ready for execution, it is easy to find a time slot for that task (or none). The new task can run if the combination of task priority and partition budget is higher than the one of the task at the head of the queue. In this case, the queue's head is removed from the queue and prepared for a task switch.[7] The previously removed slice is enqueued again with the new task's properties after the switch took place.

Each CPU in the system provides one time slice at a time or none if the CPU is running a static partition. The slice is either enqueued in the reserved queue or the non-reserved queue. Above, it was explained how a slice can be found for a known task. In three different situations, a suitable task needs to be found for a slice:

1. A time slice changes the queue.
   (reserved ↔ non-reserved)

2. The partition of the task running in a time slice changes the budget state.
   (available ↔ overused)

3. The task running in a time slice is dequeued.
   (suspend, terminate, ...)

Dynamic partitions are organised in four queues. Like the queues for available time slices, partitions are enqueued for reserved and non-reserved CPUs. However, a partition can be present in both categories at the same time, since dynamic partitions have different budget values for both kinds of CPUs. Partitions that have overused their CPU budget are separated from those who have not. Partitions are enqueued according to the task with the highest priority level that is not already running. A task for a free time slice is always found in the head of the queue of partitions that still have a valid CPU budget.

Each dynamic partition maintains a priority based array for the related tasks. Tasks of the same priority are stored in a queue. A task is in that array if it is ready to run but not already scheduled in any time slice. A task that is added to or removed from the array changes the partition ordering.

11.7.3. `SCHED_HRTL` Integration

A time slice is removed from the non-reserved queue by a callback function that is registered in the HRTL CPU reservation process (Section 11.4.2). The slice is enqueued in the reserved queue when a system group starts execution. The `periodic()` callback function of the `HRTL_SCHED_SYSTEM` scheduler module informs the budget accounting core about the new space.

The `pick_next_task()` function from the HRTL scheduling class is always called on every task switch on every CPU (Listing 11.21). The two functions

[7] A task switch is realised by various callbacks from the Linux scheduler core.

hrtl_budget_run_hrtl() and hrtl_budget_run_linux() signal that the calling CPU is currently running either a real-time task or a normal Linux task.

account.c

```
static void hrtl_budget_run_group(int idx, int cpu)
{
...
        if (hrtl_cpu_reserved(cpu))
                window = &hrtl_window;
        else
                window = &linux_window;
...
        per_cpu(budget_group_last, cpu) = idx;
        __set_bit(idx, window->frame.budget_groups[cpu]);
...
}

void hrtl_budget_run_linux(void)
{
...
        hrtl_budget_run_group(linux_budget_idx, smp_processor_id());
}

void hrtl_budget_run_hrtl(int idx)
{
        hrtl_budget_run_group(idx, smp_processor_id());
}
```

Listing 11.27: Register partition as active in frame

The periodic timer that is provided by the HRTL system for scheduler modules is used to realize time slices for real-time tasks within a partition as well as for Linux tasks running on reserved CPUs. The timer calls the hrtl_linux_tick_emulation() function which behaves like the original Linux timer callback. If the time slice of a real-time task running in a dynamic partition is exceeded the task will be put to the tail of the queue in the related priority array.

11.8. System-Call Handler Threads

The HRTL system allows partitions to execute system-calls in *system-call handler threads* (Section 10.4.2). Whenever a thread invokes a system-call a work package is sent to the associated handler. One handler can be connected to multiple partitions. The handler thread main loop is shown in Listing 11.28.

thread.c

```
static int hrtl_handler_thread_fn(void *data)
{
...
        preempt_disable();
...
        for (;;) {
...
                handler->state = HRTL_HANDLER_RUNNING;
```

```
                if (need_resched())
                        handler->flags |= HRTL_HANDLER_SUSPENDED;
                else if (NULL == handler->work) {
                        handler->work = __hrtl_handler_get_next_work(
                             handler);

                        if (NULL == handler->work) {
...
                                handler->state = HRTL_HANDLER_WAITING;
                        } else
                                handler->work_prio = handler->work->prio;
                }
...
                if (handler->state == HRTL_HANDLER_WAITING)
                        hrtl_handler_idle(handler, ticket);
                else if (!(handler->flags & HRTL_HANDLER_SUSPENDED))
                        hrtl_handler_do_work(handler);
                else
                        hrtl_handler_resched(handler);
        }
...
        preempt_enable_no_resched();
...
}
```

Listing 11.28: System-call handler thread main loop

In every main loop cycle one of three different actions is performed. The thread calls the main scheduler function (`hrtl_handler_resched()` → `schedule()`) if the resched flag is set. If the flag is not set and a work package is waiting in the queue, the thread starts or continues executing the package (`hrtl_handler_do_work()`; Section 11.8.2). If no work package is pending, the thread waits sleeping for a trigger (`hrtl_handler_idle()`).

11.8.1. System-Call Redirection

The Linux macros for system-call declaration (`SYSCALL_DEFINE`...) are replaced by HRTL versions. An additional function call is placed between system-call invocation and execution of the system-call function. On each call it is checked if the calling task has defined a system-call redirection. In case the system-call has to be executed by a handler thread, a work package is configured and appended to the task object. The calling task sets the `HRTL_TASK_HANDLER_LOAD` flag, prepares itself for waiting and calls the scheduler main routine. The scheduler function will notice the `HRTL_TASK_HANDLER_LOAD` flag and will then enqueue the appended work package to the handler thread's queue after the task is suspended.[8]

The function `__hrtl_handler_enqueue_new_work()` places a work package in the handler's package queue. If the new package should replace the currently

[8] This step is necessary, because otherwise the work package could be treated by the handler thread before the calling task is suspended.

running package (higher priority), `__hrtl_handler_trigger()` sends a reschedule interrupt to the CPU where the handler is running.

thread.c

```
static void
__hrtl_handler_trigger(struct hrtl_handler *handler, int prio)
{
        if (handler->state == HRTL_HANDLER_WAITING)
                hrtl_event_trigger(&handler->trigger);
        else if (handler->work_prio > prio) {
                handler->flags |= HRTL_HANDLER_WORK_SCHEDULE;
                smp_send_reschedule(task_cpu(handler->thread));
        }
}

static int __hrtl_handler_queue_add_tail(struct hrtl_handler *handler,
                                          struct hrtl_handler_work *work)
{
        work->state = HRTL_HANDLER_WORK_ENQUEUED;
        handler->count_queued++;
        __set_work_prio(work);
        list_add_tail(&work->list,
              &handler->q->prio_queues[work->grp_prio][work->task_prio]);
        __set_bit(work->grp_prio, &handler->q->group_prio_bitmap);

        return !__test_and_set_bit(work->task_prio,
                        &handler->q->task_prio_bitmap[work->grp_prio]);
}

static void inline
__hrtl_handler_enqueue_new_work(struct hrtl_handler *handler,
                                struct hrtl_handler_work *work)
{
        if (__hrtl_handler_queue_add_tail(handler, work))
                __hrtl_handler_trigger(handler, work->prio);
}
```

Listing 11.29: Adding a new work package

If the system-call execution is completed, the previously suspended task is woken up. The result of the system-call function is stored in the work package and returned to user space.

11.8.2. Work Package Scheduling

Work package scheduling is based on the `setjmp()`/`longjmp()` technique known from the C standard library[Jon91, Chap. 5]. The function `hrtl_save_environment()` saves the current environment, at some point of program execution, into a data structure (`__hrtl_environment`) that can be used at some later point of program execution by `hrtl_restore_environment()` to restore the program state. This process can be imagined to be a *jump* back to the point of program execution where the environment was saved. The return value from `hrtl_save_environment()` indicates whether control reached that point normally or from a call to `hrtl_restore_-`

`environment()`. A work package defines one `__hrtl_environment` object in order to store the current execution state. If a running work package should be preempted, the current program state is saved and the handler restores the environment saved before package execution started.

Since work packages can be preempted by other work packages, each work package needs its own stack segment for local variables and function calls. A work package is always connected to a waiting task (the actuator of the system-call). When a work package is scheduled by a handler thread, the current kernel stack segment is changed to the stack that belongs to the waiting task.

In order to keep references to user space memory objects valid, the memory segment of the waiting task is marked to be active. This allows correct user space address translation from kernel space. All relevant data structures are now valid and available during system-call execution. Since the current stack segment was changed to the original one (the one that would be active during normal system-call execution), the reference to the current task is also correct. The Linux macro `current` addresses the task which is currently running. This reference is saved in the stack segment's header. Thus, things like, for instance, the file descriptor table will be used from the waiting task. The system-call execution path does not need to be altered. However, special care must be taken in the Linux scheduler core when dealing with handler threads (Section 11.8.4).

thread.c

```
static void hrtl_handler_switch_to(struct hrtl_handler *handler)
{
...
        this_cpu_write(current_task, handler->work->p);
        hrtl_set_rq_curr(handler->work->p);
        this_cpu_write(kernel_stack,
                       (unsigned long)task_stack_page(handler->work->p) +
                       THREAD_SIZE - KERNEL_STACK_OFFSET);
        switch_mm(handler->thread->active_mm, handler->work->p->mm,
                  handler->work->p);
...
        handler->work->p->state = handler->work->p->hrtl.saved_state;
...
        current_thread_info()->cpu = smp_processor_id();
}

static void hrtl_handler_switch_from(struct hrtl_handler *handler)
{
...
        handler->work->p->hrtl.saved_state = handler->work->p->state;
        handler->work->p->state = TASK_UNINTERRUPTIBLE;
...
}
```

Listing 11.30: Switch to/from work package context

The two functions `hrtl_handler_switch_to()` and `hrtl_handler_switch_from()` implement the context switch between handler thread and work package (Listing 11.30). The current environment is changed in other functions which are

discussed later in this section. When switching back from a work package, the actual task state is saved inside the work package. The task that is connected with the work package is then changed back to be in a waiting state.

As can be seen in Listing 11.28, work package execution is managed by `hrtl_handler_do_work()`. The function is shown in Listing 11.31. `hrtl_save_environment()` returns a value equal to zero if it was invoked to save the current environment and a value other than zero if `hrtl_restore_environment()` was called. The flag `HRTL_HANDLER_WORK_ENTRY` is set by the system-call redirection code and indicates a new call to be started. Each work package starts execution with `hrtl_handler_do_start_work()` and finishes with `hrtl_handler_do_finish_work()`. Work package interruptions are treated by `hrtl_handler_do_handle_jmp_back()` and are continued by `hrtl_handler_do_continue_callback()`[9].

thread.c

```
static void noinline hrtl_handler_do_work(struct hrtl_handler *handler)
{
...
        if (0 != (resched = hrtl_save_environment(handler->_env))) {
                /* jmp back from callback... */
                hrtl_handler_do_handle_jmp_back(handler, resched);
        } else if (handler->work->flags & HRTL_HANDLER_WORK_ENTRY) {
                /* start new callback... */
                handler->work->flags &= ~HRTL_HANDLER_WORK_ENTRY;
                hrtl_handler_do_start_work(handler);
                hrtl_handler_do_finish_work(handler);
        } else {
                /* continue callback... */
                hrtl_handler_do_continue_callback(handler);
                BUG();
        }
...
}
```

Listing 11.31: Manage work package execution

Starting a new work package and continuing a previously preempted package switches the current context by calling `hrtl_handler_switch_to()`. The stack pointer has to be adjusted for a new package before the defined callback function is called. Later callbacks (continuations) will consider the correct stack pointer, since it is included in the saved environment. A call to `hrtl_restore_environment()` will continue a previously preempted work package. When the execution of an work package is completed (return from callback function), the old stack pointer is restored and the context is switched back to the handler thread.

If a work package is preempted, the function `hrtl_handler_do_handle_jmp_back()` restores the previously saved environment. The work package is enqueued back to the handlers work package queue if the task state is still `TASK_RUNNING`.

[9] `hrtl_handler_do_continue_callback()` never returns. The work package will either be preempted (`hrtl_handler_do_handle_jmp_back()`) or will return from `hrtl_handler_do_start_work()`.

Otherwise, the work package was put to a waiting queue during system-call execution. In this case, the package is put back to the queue. The associated *wakeup* call will enqueue the package and trigger the corresponding handler thread. According to the explanation in Section 10.4.2.1, a work package can be preempted in case the superior handler thread shall adopt another preempted package from another handler thread. The work package that initiates the adoption stores (carries) the preempted package and will be put to a waiting state until the adopted package is completed or interrupted.

If a handler thread that is executing a work package invokes the Linux scheduler, the function `hrtl_handler_work_schedule()` is called. It is the counterpart to `hrtl_handler_do_handle_jmp_back()`. The current environment is saved inside the work package data object and the previously saved environment of the handler thread is restored.

A previously suspended work package in a sleeping state can be woken up by the Linux *try-to-wakeup* mechanism. However, instead of reactivating the connected task, the work package is sent back to the handler's queue by `__hrtl_handler_enqueue_new_work()`. A wakeup call to a work package is redirected to `hrtl_handler_work_wake()`. The function behaves like the Linux scheduler function `try_to_wake_up()`.

11.8.3. Spin-Lock Replacement

The *spin-lock* macros as defined in `include/linux/spinlock.h` and `spinlock_types.h` are replaced by HRTL versions. The `raw_` variants are untouched and can still be used. Listing 11.32 shows the spin-lock structure that is introduced by the HRTL patch.

spinlock_types.h

```
typedef struct spinlock {
        struct raw_spinlock rlock;
        struct spinlock *nested;
        struct task_struct *owner;
        unsigned int magic;
        struct gtplist_head wait_list[2];
} spinlock_t;
```

Listing 11.32: HRTL spinlock type

The data type provides two different queues for waiting threads. A thread that tries to acquire a spin-lock which has already been locked is put to one of these queues. One of the queues maintains threads that are allowed to be suspended. The other one holds references to busy waiting threads. The `magic` variable indicates if the spin-lock object was initialised correctly. Some kernel code declares and uses spin-lock objects without using the appropriate spin-lock declaration and initialisation macros.[10] If the `magic` variable shows that the object was not initialised, and thus the queues for

[10] For instance, declaring an spin-lock object as a global variable will initialise the memory area with zero. This can be enough for a traditional spin-lock object.

waiting threads were not set up, the spin-lock needs to be initialised on the first use at runtime.

A spin-lock is claimed when the `owner` pointer is set to the callers task object. The pointer is changed by the `cmpxchg()` command that is provided by `x86` architectures.[11] If the command did not successfully allocate the spin-lock's `owner` pointer, the spin-lock is either already blocked or is in a transition state. In both cases, the `rlock` is taken and the `owner` reference is modified (Listing 11.33). The appropriate spin-lock release code will fail on freeing the spin-lock by `cmpxchg()`, because the `LOCK_HAS_WAITERS` flag is set. This forces the release call to also claim the `rlock`.

handler.h

```
#define LOCK_HAS_WAITERS  1UL
```

spinlock.c

```
static void __lock_set_waiter_flag(struct spinlock *lock)
{
        unsigned long owner, *p = (unsigned long *) &lock->owner;

        do {
                owner = *p;
        } while (cmpxchg(p, owner, owner | LOCK_HAS_WAITERS) != owner);
}
```

Listing 11.33: Set waiter flag

Before a thread is put to one of the wait queues, it is checked if the spin-lock is held by a work package that is preempted (is located on a handler thread package queue). In this case, the calling thread adopts the work package and is suspended until the adopted package has released the lock. If a spin-lock is released and the `LOCK_HAS_WAITERS` flag is set, the thread with the highest priority is triggered.

Once a thread has claimed a spin-lock, no other thread can claim the same lock before the first thread has released it. For a handler thread executing a work package, the preemption counter is not modified. This allows the handler thread (and thus the work package; see Section 11.8.2) inside the lock to be preempted by other threads and work packages.

11.8.4. Necessary Adjustments

The Linux run-queue structure (`rq`) is extended by an additional field for starting system-call work packages. As described in Section 11.8.1 a work package is sent to the handler thread's queue after the calling thread is suspended. The main scheduler routine (`__schedule()`) stores a starting work package in the run-queue object before a task switch takes place. The new task will recognize the pending work package and send it to the handler's queue by calling `hrtl_handler_syscall_start()`. This must be done outside the run-queue lock, since the corresponding wakeup call to the handler thread will also lock the queue.

[11] The command compares the contents of a memory location (`owner`) to a given value (`NULL`) and, only if they are the same, modifies the contents of that memory location to a given new value (`current`).

The public functions schedule(), preempt_schedule(), preempt_schedule_-irq() and __cond_resched() are patched so that the handler thread scheduler function (hrtl_handler_work_schedule()) is called instead of __schedule() in case the calling thread is executing a work package. Listing 11.34 shows the modification for schedule(). The other functions are patched in a similar way.

core.c

```
asmlinkage void __sched schedule(void)
{
...
#ifdef CONFIG_HRTL
        if (current->hrtl.flags & HRTL_TASK_HANDLER_PATH)
                hrtl_handler_work_schedule();
        else
                __schedule();
#else
        __schedule();
#endif
}
```

Listing 11.34: Scheduler modification for work packages

The main wakeup function try_to_wake_up() is extended so that the handler thread wakeup function (hrtl_handler_work_wake()) is called in case the addressed thread is executing a work package. Furthermore, threads waiting on a spin-lock are not woken up by Linux calls.

11.9. Real-Time Application Programming

The HRTL extension introduces new system-calls to the Linux kernel. A user-space library is provided for ease use of the HRTL patch (Section 11.9.1.1). Example applications can be seen in Section 11.9.2.

11.9.1. User-Space Interface

System-calls added by the HRTL extension are listed in Table 11.2.

Name	Number	Parameter		
hrtl_entity	1024	1	enum hrtl_entity_domain	domain
		2	enum hrtl_entity_fct	function
		3	union hrtl_entity_arg	arg1
		4	union hrtl_entity_arg	arg2
hrtl_core	1025	1	enum hrtl_core_fct	function
		2	union hrtl_core_arg	arg1
		3	union hrtl_core_arg	arg2
hrtl_system	1026	1	enum hrtl_system_fct	function
		2	union hrtl_system_arg	arg1
		3	union hrtl_system_arg	arg2

		4	union hrtl_system_arg	arg3
		5	union hrtl_system_arg	arg4
hrtl_sched_getparam	1027	1	pid_t	pid
		2	struct hrtl_sched_param *	param
hrtl_sched_setparam	1028	1	pid_t	pid
		2	struct hrtl_sched_param *	param
hrtl_housekeeping	1029	1	int	flags
hrtl_sched_setfork	1030	1	pid_t	pid
		2	struct hrtl_sched_fork *	fork
hrtl_sched_getfork	1031	1	pid_t	pid
		2	struct hrtl_sched_fork *	fork
hrtl_getgroup	1032	1	pid_t	pid
hrtl_getcore	1033	1	pid_t	pid
hrtl_getperiod	1034	1	hrtl_period_t *	period
hrtl_shutdown	1035		*none*	
hrtl_profile	1036	1	enum hrtl_profile_fct	function
		2	union hrtl_profile_arg	arg1
		3	union hrtl_profile_arg	arg2
hrtl_event_trigger	1037	1	hrtl_id_t	id
hrtl_event_wait	1038	1	hrtl_id_t	id
hrtl_event_core_wait	1039	1	hrtl_id_t	id
hrtl_event_wait_count	1040	1	hrtl_id_t	id
		2	unsigned int	count
hrtl_event_get_count	1041	1	hrtl_id_t	id
hrtl_cache_disable	1042		*none*	
hrtl_cache_enable	1043		*none*	

Table 11.2.: Listing of HRTL systemcalls

The system-calls are used by the HRTL user-space library. Description of datatypes can be found in Section 11.9.1.1.

11.9.1.1. Library

The HRTL user-space library defines 149 functions. Not all of these functions are discussed in this section (e.g. sorting and iteration lists exported by the kernel to user-space). The complete HRTL user-space library can be accessed at [Rad15b].

sched_setparam, sched_getparam set and get scheduling parameters

```
int libhrtl_sched_setparam(pid_t pid,
                           const struct hrtl_sched_param *param);
int libhrtl_sched_getparam(pid_t pid,
                           struct hrtl_sched_param *param);
```

```
struct hrtl_sched_param {
    unsigned int flags;
    unsigned int prio;
    hrtl_time_t runtime;
    unsigned int period;
};
```

`libhrtl_sched_setparam()` sets the scheduling parameters associated with the scheduling module for the process identified by `pid`. If `pid` is zero, then the parameters of the calling process are set.

`libhrtl_sched_getparam()` retrieves the scheduling parameters for the process identified by `pid`. If `pid` is zero, then the parameters of the calling process are retrieved.

The `flags` argument is constructed as the bitwise *or* of one or more of the following constants:

HRTL_TASK_YIELD_CYCLE A call to `sched_yield()` will cause the process to be suspended until the next cycle starts.

HRTL_TASK_DEADLINE Depending on the selected scheduler module, this flag enables the periodic task mode.

HRTL_TASK_SIGNALS The process receives an POSIX signal in case a deadline was missed.

Returns: 0 on success, < 0 on error

Errors:	EINVAL	The argument `param` does not make sense.
	EFAULT	Memory copy from/to user-space failed.
	ESRCH	The process whose ID is `pid` could not be found.
	EPERM	The calling process does not have appropriate privileges.

set_fork, **get_fork** set and get fork parameters

```
int libhrtl_sched_setfork(pid_t pid,
                          const struct hrtl_sched_fork *fork);
int libhrtl_sched_getfork(pid_t pid,
                          struct hrtl_sched_fork *fork);
int libhrtl_profile_set_fork(hrtl_key_t key,
                             const struct hrtl_sched_fork *fork);
int libhrtl_profile_get_fork(hrtl_key_t key,
                             struct hrtl_sched_fork *fork);
int libhrtl_group_set_fork(hrtl_id_t id,
                           const struct hrtl_sched_fork *fork);
int libhrtl_group_get_fork(hrtl_id_t id,
                           struct hrtl_sched_fork *fork);

struct hrtl_sched_fork {
    hrtl_id_t group_id;
    struct hrtl_sched_param param;
};
```

`libhrtl_sched_setfork()` sets the fork parameters for the process identified by `pid`. If `pid` is zero, then the parameters of the calling process are set. `libhrtl_profile_set_fork()` sets the fork parameters for the profile identified by `key`. `libhrtl_group_set_fork()` sets the fork parameters for the group identified by `id`.

`libhrtl_sched_getfork()` retrieves the fork parameters for the process identified by `pid`. If `pid` is zero, then the parameters of the calling process are retrieved. `libhrtl_profile_get_fork()` retrieves the fork parameters for the profile identified by `key`. `libhrtl_group_get_fork()` retrieves the fork parameters for the group identified by `id`.

The argument `param` will be set for a new process via `libhrtl_sched_setparam()`. The new process will be forked inside the group specified by `group_id`.

Returns: 0 on success, < 0 on error

Errors:		
	EINVAL	The argument `param` does not make sense.
		No fork parameters are defined.
	EFAULT	Memory copy from/to user-space failed.
	ESRCH	The process whose ID is `pid` could not be found.
		The group whose ID is `id` could not be found.
		The profile whose KEY is `key` could not be found.
	EPERM	The calling process does not have appropriate privileges.
	EBUSY	The current scheduler module does not allow modifications.

`start_rt`, `stop_rt` set and clear task real-time status

```
int libhrtl_sched_start_rt(pid_t pid,
                           hrtl_id_t group_id);
int libhrtl_sched_start_rt_no_mlock(pid_t pid,
                                    hrtl_id_t group_id);
int libhrtl_sched_stop_rt(pid_t pid);
```

`libhrtl_sched_start_rt*()` adds the process identified by `pid` to the group identified by `group_id`. If `pid` is zero, then the calling process is assigned. On success, `libhrtl_sched_start_rt()` locks all of the identified process's virtual address space into RAM, preventing that memory from being paged to the swap area.

`libhrtl_sched_stop_rt()` removes the process identified by `pid` from the group the process is currently running in. If `pid` is zero, then the calling process is removed. Previously locked memory is unlocked on success.

Returns: 0 on success, < 0 on error

Errors:

ESRCH	The process whose ID is `pid` could not be found.
	The group whose ID is `id` could not be found.
EPERM	The calling process does not have appropriate privileges.
EBUSY	The group's scheduler module does not allow modifications.

`housekeeping` perform housekeeping

```
int libhrtl_housekeeping(unsigned int flags);
```

Perform housekeeping actions according to the description in Section 10.2.3.

Returns: 0 on success, < 0 on error

Errors:

EINVAL	The calling task is running in a dynamic partition.
ECANCELED	The calling task is not running in a real-time partition.

`getgroup`, `getcore`, `getperiod` get task runtime environment information

```
int libhrtl_getgroup(pid_t pid);
int libhrtl_getcore(pid_t pid);
int libhrtl_getperiod(hrtl_period_t *period);

typedef struct hrtl_period_val {
    unsigned long  count;
    hrtl_time_t    time;
} hrtl_period_val_t;

typedef struct hrtl_period {
    hrtl_period_val_t  val;
    hrtl_time_t        runtime;
} hrtl_period_t;
```

`libhrtl_getgroup()` and `libhrtl_getcore()` return the current group ID respectively core ID where the process identified by `pid` is running in. If `pid` is zero, then the parameter for the calling process is returned.

`libhrtl_getperiod()` retrieves timing information of the calling process (Section 11.1).

Returns: 0 or ID on success, < 0 on error

Errors:

EINVAL	The calling task is not running in a real-time partition.
EFAULT	Memory copy to user-space failed.
ESRCH	The process whose ID is `pid` could not be found.
EPERM	The calling process does not have appropriate privileges.

`event_wait`, `event_trigger` event handling (Section 11.3.3)

```
int libhrtl_event_wait(hrtl_id_t id);
int libhrtl_event_wait_count(hrtl_id_t id,
```

```
                                unsigned int count);
int libhrtl_event_core_wait(hrtl_id_t id);
int libhrtl_event_group_wait(hrtl_id_t id);
int libhrtl_event_get_count(hrtl_id_t id);
int libhrtl_event_trigger(hrtl_id_t id);
```

`libhrtl_event_wait*()` suspends the calling process until the event identified by `id` is triggered. `libhrtl_event_wait_count()` allows to specify a counter value of event occurrences. If the counter does not match the current event counter, the calling process will not be suspended.

`libhrtl_event_core_wait()` and `libhrtl_event_group_wait()` suspend the calling process until the group respectively core identified by `id` starts the next cycle.

Returns: 0 or counter on success, < 0 on error

Errors:		
	EINVAL	Parameter `id` is not in range.
	ENOKEY	The event, core or group identified by `id` does not exist.
	EINTR	An unblocked POSIX signal was caught.

`system_get` get system information

```
int libhrtl_system_get_version(hrtl_version_t *version);
int libhrtl_system_get_info(hrtl_system_info_t *info);
int libhrtl_system_get_cpus_available(size_t setsize, cpu_set_t *mask);
int libhrtl_system_get_cpus_allowed(size_t setsize, cpu_set_t *mask);
int libhrtl_system_get_cpus_reserved(size_t setsize, cpu_set_t *mask);

typedef struct hrtl_version {
    unsigned int major;
    unsigned int minor;
} hrtl_version_t;

typedef struct hrtl_system_arg_info {
        struct hrtl_version version;
        unsigned int abi_revision;
        unsigned int timer_unit_ns;
        int system_cpu;

        struct {
                struct hrtl_key_range system_key_range;
                struct hrtl_key_range resource_key_range;
                unsigned int max_core;
                unsigned int max_group;
                hrtl_time_t core_runtime_min;
                hrtl_time_t core_runtime_max;
                hrtl_time_t group_runtime_min;
                hrtl_time_t group_runtime_max;
                hrtl_time_t idletime_min;
                hrtl_time_t idletime_max;
                hrtl_time_t task_runtime_min;
```

```
                hrtl_time_t task_runtime_max;
                unsigned int prio_levels;
        } limit;
} hrtl_system_info_t;
```

`libhrtl_system_get_info()` returns a `hrtl_system_info_t` block with several system constans.

Returns: 0 on success, < 0 on error

Errors: EFAULT Memory copy to user-space failed.

`system_reserve`, `system_release` CPU reservation

```
int libhrtl_system_reserve(int cpu,
                           hrtl_key_t profile,
                           hrtl_id_t core);
int libhrtl_system_reserve_sync(int cpu,
                           hrtl_key_t profile,
                           hrtl_id_t core,
                           hrtl_id_t sync_core);
int libhrtl_system_release(int cpu);
int libhrtl_shutdown(void);
```

`libhrtl_system_reserve*()` claims or reserves the CPU specified by `cpu`. If `cpu` is set to -1, the first free available CPU will be reserved. The parameters `profile` and `core` specify the KEY and the ID of the objects as described in Section 11.4. `libhrtl_system_reserve_sync()` allows to specify an additional core object that is already running on a reserved CPU. The core identified by `core` will run in sync with the core identified by `sync_core`. Both core objects need to have the same runtime parameters.

`libhrtl_system_release()` releases the CPU specified by `cpu`. `libhrtl_shutdown()` releases all CPUs currently reserved.

Returns: 0 or CPU on success, < 0 on error

Errors:		
	ENOSPC	No free CPU is available.
	ENOKEY	The core identified by `core` does not exist.
		The profile identified by `profile` does not exist.
	EALREADY	CPU specified by `cpu` is already reserved.
	EIO	System error.
	EPERM	The calling process does not have appropriate privileges.
	EINVAL	Parameter `cpu` is not in range.
	ECANCELED	CPU specified by `cpu` is not reserved.

`request_irq`, `release_irq` enable and disable interrupts

```
int libhrtl_system_request_irq(int cpu, unsigned int irq);
int libhrtl_system_release_irq(int cpu, unsigned int irq);
```

After a process has been given real-time status, no interrupt requests are relayed to the reserved CPU any more. The calling process can enable single IRQs by means of `libhrtl_system_request_irq()`. The requested interrupts will be delivered exclusively to the reserved CPU.

`libhrtl_system_release_irq()` disables the delivery of the specified IRQ to the reserved CPU. Afterwards, the IRQ will be delivered to all non-reserved CPUs.

Returns: 0 on success, < 0 on error

Errors:	EINVAL	Parameter `cpu` is not in range.
		Parameter `irq` is not in range.
	EALREADY	IRQ specified by `irq` is already assigned/released.
	EIO	System error.
	ECANCELED	CPU specified by `cpu` is not reserved.

`cache_disable`, `cache_enable` enable and disable cache

```
int libhrtl_cache_disable(void);
int libhrtl_cache_enable(void);
```

Disable or enable the cache of the CPU where the function is called.

Returns: 0 on success, < 0 on error

Errors: ECANCELED Calling process is not HRTL process.

`create`, `destroy` create and delete objects

```
int libhrtl_profile_create(const struct hrtl_entity_arg_create *cr,
                           const hrtl_profile_create_data_t *data);
int libhrtl_profile_destroy(hrtl_key_t key);
int libhrtl_handler_create(const struct hrtl_entity_arg_create *cr);
int libhrtl_handler_destroy(hrtl_id_t id);
int libhrtl_group_create(const struct hrtl_entity_arg_create *cr,
                         const hrtl_group_create_t *data);
int libhrtl_group_destroy(hrtl_id_t id);
int libhrtl_core_create(const struct hrtl_entity_arg_create *cr,
                        const hrtl_core_create_data_t *data);
int libhrtl_core_destroy(hrtl_id_t id);
int libhrtl_event_create(const struct hrtl_entity_arg_create *cr);
int libhrtl_event_destroy(hrtl_id_t id);

struct hrtl_entity_arg_create {
    char name[HRTL_NAME_LEN];
    hrtl_key_t key;
};

typedef struct hrtl_entity_profile_create {
    hrtl_key_t clock_source;
    hrtl_key_t timer_device;
} hrtl_profile_create_data_t;
```

```
typedef struct hrtl_entity_core_create {
    hrtl_time_t runtime;
} hrtl_core_create_data_t;

enum hrtl_group_type {
    HRTL_GROUP_STATIC,
    HRTL_GROUP_DYNAMIC,
};

typedef struct hrtl_entity_group_create {
    enum hrtl_group_type type;

    union {
        struct {
            hrtl_time_t runtime;
        } static_grp;

        struct {
            unsigned int rt_budget;
            unsigned int budget;
        } dynamic_grp;
    };
} hrtl_group_create_t;
```

Create or destroy core, group, system-call handler and event objects.

Returns: 0 on success, < 0 on error

Errors:	EINVAL	The argument `cr` ot `data` does not make sense.
	EFAULT	Memory copy from user-space failed.
	EPERM	The calling process does not have appropriate privileges.
	ENOSPC	No space available for a new object.

`by_key`, `by_id` get object identifiers

```
hrtl_id_t libhrtl_handler_by_key(hrtl_key_t key);
hrtl_key_t libhrtl_handler_by_id(hrtl_id_t id);
hrtl_id_t libhrtl_group_by_key(hrtl_key_t key);
hrtl_key_t libhrtl_group_by_id(hrtl_id_t id);
hrtl_id_t libhrtl_core_by_key(hrtl_key_t key);
hrtl_key_t libhrtl_core_by_id(hrtl_id_t id);
hrtl_id_t libhrtl_event_by_key(hrtl_key_t key);
hrtl_key_t libhrtl_event_by_id(hrtl_id_t id);
```

Get KEY respectively ID of an object.

Returns: KEY or ID on success, < 0 on error

Errors:	ESRCH	The object whose ID is `id` could not be found.
		The object whose KEY is `key` could not be found.

`chmod`, `chown` change object owner and mode bits

```
int libhrtl_profile_chmod(hrtl_key_t key, short mode);
int libhrtl_profile_chown(hrtl_key_t key, hrtl_user_t *user);
int libhrtl_group_chmod(hrtl_id_t id, short mode);
int libhrtl_group_chown(hrtl_id_t id, struct hrtl_user *user);
int libhrtl_core_chmod(hrtl_id_t id, short mode);
int libhrtl_core_chown(hrtl_id_t id, struct hrtl_user *user);
int libhrtl_event_chmod(hrtl_id_t id, short mode);
int libhrtl_event_chown(hrtl_id_t id, struct hrtl_user *user);

struct hrtl_user {
    uid_t uid, cuid;
    gid_t gid, cgid;
};
```

These calls change the permissions and the owner of an object. The new permissions are specified in `mode`. The flags for `mode` are described in the documentation of the `chmod()` Linux system-call.

Returns: KEY or ID on success, < 0 on error

Errors:	ESRCH	The object whose ID is `id` could not be found.
		The object whose KEY is `key` could not be found.
	EPERM	The calling process does not have appropriate privileges.

`handler_start`, `handler_stop` start and stop system-call handler

```
int libhrtl_handler_start(hrtl_id_t id);
int libhrtl_handler_stop(hrtl_id_t id);
int libhrtl_handler_getpid(hrtl_id_t id);
```

Start or stop a handler thread identified by `id`.

Returns: 0 or PID on success, < 0 on error

Errors:	ESRCH	The handler whose ID is `id` could not be found.
	EALREADY	Handler is already running/stopped

`add_group`, `del_group` connect and disconnect group with core

```
int libhrtl_core_add_group(hrtl_id_t id, hrtl_id_t grp_id);
int libhrtl_core_del_group(hrtl_id_t id, hrtl_id_t grp_id);
```

Connect or disconnect core and group objects.

Returns: 0 on success, < 0 on error

Errors:	ESRCH	The object whose ID is `id` or `grp_id` could not be found.
	EALREADY	Group with ID `grp_id` is already connected.
	ENOSPC	The specified core object does not have enough free space.

`deadline_events` deadline watchdog interface

```
int libhrtl_profile_register_deadline_watchdog(hrtl_key_t key);
int libhrtl_profile_deregister_deadline_watchdog(void);
int libhrtl_profile_get_deadline_events(struct hrtl_deadline_info *
    buffer,
                                        unsigned int limit);
```

`libhrtl_profile_register_deadline_watchdog()` registers the calling process to be the deadline watchdog for the profile identified by `key`. A process can deregister itself by calling `libhrtl_profile_deregister_deadline_watchdog()`.

A registered deadline watchdog can retrieve deadline events (Section 11.6.5) by calling `libhrtl_profile_get_deadline_events()`. The maximum number of available deadline events as giving in parameter `limit` will be copied to the memory area specified by `buffer`.

Returns: 0 or number of events on success, < 0 on error

Errors:		
	ESRCH	The profile whose KEY is `key` could not be found.
	EALREADY	Profile with KEY `key` is already connected.
		Calling process is already a deadline watchdog.
	ECANCELED	Calling process is not a registered deadline watchdog.

`set_sched`, `clear_sched` set and clear secheduler module for group

```
int libhrtl_group_set_sched(hrtl_id_t id, hrtl_key_t sched_key);
int libhrtl_group_clear_sched(hrtl_id_t id);
```

Assign scheduler module specified by `sched_key` to group identified by `id`. An already assigned module can be removed by `libhrtl_group_clear_sched()`.

Returns: 0 or number of events on success, < 0 on error

Errors:		
	ESRCH	The module whose KEY is `sched_key` could not be found.
		The group whose ID id `id` could not be found.
	EALREADY	Group with ID `id` has already a module assigned.
	ECANCELED	Module can not be assigned to group.
	EPERM	The calling process does not have appropriate privileges.

`syscall_connect`, `syscall_disconnect` connect and disconnect group with system-call handler

```
int libhrtl_group_syscall_connect(hrtl_id_t id, hrtl_id_t handler_id);
int libhrtl_group_syscall_disconnect(hrtl_id_t id);
```

Connect system-call handler thread specified by `handler_id` with group identified by `id`. An already assigned thread can be removed by `libhrtl_group_syscall_disconnect()`

Returns: 0 on success, < 0 on error

Errors:		
	ESRCH	The handler whose ID is `handler_id` could not be found.
		The group whose ID id `id` could not be found.
	EALREADY	Group with ID `id` has already a handler assigned.
	ECANCELED	Handler thread is defined but not started.
	EPERM	The calling process does not have appropriate privileges.

11.9.2. Benchmarking

The benchmark tests described in Section 5.4.3 are discussed in this section in concrete implementation for the HRTL extension. In Chapter 12 the benchmark results are compared and evaluated to other real-time operating systems. The values presented in this section are the results from executing the same tests for a patched and a non-patched kernel[12]. The tests for the unpatched kernel were already described in Section 6.5. All benchmark results can be accessed at `http://www.informatik.uni-bremen.de/agbs/dirkr/HRTL/benchmark_results.tgz`.

11.9.2.1. Task Period Tests

Listing 11.35 shows the HRTL 500μs periodic task benchmark test implementation. The setup function `setup()` is called early in the main function. The function creates a profile, core and a group object with the API calls explained in Section 11.9.1.1. The core object has a defined cycle time of 5 seconds and contains only the defined group object which has the same cycle time. A free CPU is reserved for the core object. After the benchmark process was added to the partition (`libhrtl_sched_start_rt()`), the `HRTL_TASK_YIELD_CYCLE` flag is set (Section 11.6.4). A `sched_yield()` system-call blocks until the programmed period expires (`libhrtl_set_task_deadline()`). The memory area for the measurement results (`tsc[]`) is touched before executing the main loop. As described in Section 6.4 this is necessary to cause a stack fault before the test starts.

period_us.c

```
int main(int argc, char **argv) {
    uint32_t tsc[LOOP_COUNT];
...
    runtime_us = hrtl_us_to_time(atoi(argv[1]));
...
    if (0 > setup())
        exit(EXIT_FAILURE);

    libhrtl_sched_start_rt(getpid(), grp);
    libhrtl_set_task_yield_cycle();

    /* pre-fault stack */
```

[12]Linux kernel version 3.5.7

```
    for (i = 0; i < LOOP_COUNT; i++)
        rdtsc_32(tsc[i]);

    if (0 > libhrtl_set_task_deadline(runtime_us))
      exit(EXIT_FAILURE);

    sched_yield();

    for (i = 0; i < LOOP_COUNT; i++) {
        sched_yield();
        rdtscp_32(tsc[i]);
        cpuid();
#ifdef TRIGGER_PARPORT
        parport_toggle();
#else
        busy();
#endif
    }
...
}
```

Listing 11.35: HRTL period task benchmark test (μs)

The first benchmark test for the HRTL extension operating system measures the scheduling precision of a periodic task with a period of 500 μs (Table 11.3). The test was executed in the 3 scenarios described in Section 5.3.

Scenario	Average	Min	Max	Gap	Deviation
Normal	499.952	499.863	500.026	0.163	0.022
CPU utilization	499.952	499.852	500.052	0.200	0.040
I/O utilization	499.952	499.179	500.573	1.394	0.147

Table 11.3.: Benchmark test results [μs]: HRTL period task (500μs)

Like for the RT-Preempt extension and the HLRT patch the results in the table were converted to the μs scala. The precision of the 500 μs timer is almost met. Compared with the results of the same benchmark test on a native Linux kernel system (Table 11.4) it is easy to see, that the extension brings a higher precision for a small timer to the Linux operating system. Like mentioned before the test was executed with the same kernel without applying the HRTL patch.

Scenario	Average	Min	Max	Gap	Deviation
Normal	499.951	495.025	505.014	9.989	1.669
CPU utilization	499.956	490.280	509.615	19.335	2.667
I/O utilization	499.925	493.415	504.886	11.471	2.151

Table 11.4.: Benchmark test results [μs]: Linux 3.5.7 period task (500μs)

The remaining periodic task benchmark tests implement another approach to realise periodic task behavior. Instead of programming a task period via `libhrtl_set_-task_deadline()` the main cycle of the core object is set to the desired period. A `sched_yield()` system-call blocks until the next cycle of the core object starts.

Table 11.5 and Table 11.6 show the results of the periodic benchmark test with a period of 10 ms.

Scenario	Average	Min	Max	Gap	Deviation
Normal	9999.043	9998.971	9999.118	0.147	0.029
CPU utilization	9999.043	9998.961	9999.104	0.143	0.026
I/O utilization	9999.041	9998.449	9999.636	1.187	0.104

Table 11.5.: Benchmark test results [μs]: HRTL period task (10ms)

Scenario	Average	Min	Max	Gap	Deviation
Normal	9999.043	9994.483	10003.661	9.179	1.347
CPU utilization	9999.062	9918.925	10090.022	171.098	12.500
I/O utilization	9999.049	9985.611	10012.510	26.899	3.559

Table 11.6.: Benchmark test results [μs]: Linux 3.5.7 period task (10ms)

The same test is repeated with a 100 ms (Table 11.7 and Table 11.8) and a 1 second timer (Table 11.9 and Table 11.10). Nevertheless, the test is only executed in the normal scenario.

Scenario	Average	Min	Max	Gap	Deviation
Normal	99990.429	99990.357	99990.501	0.144	0.023

Table 11.7.: Benchmark test results [μs]: HRTL period task (100ms)

Scenario	Average	Min	Max	Gap	Deviation
Normal	99990.427	99985.325	99995.551	10.226	0.809

Table 11.8.: Benchmark test results [μs]: Linux 3.5.7 period task (100ms)

Scenario	Average	Min	Max	Gap	Deviation
Normal	999904.298	999904.204	999904.369	0.165	0.022

Table 11.9.: Benchmark test results [μs]: HRTL period task (1sec)

Scenario	Average	Min	Max	Gap	Deviation
Normal	999904.298	999899.919	999909.573	9.653	0.661

Table 11.10.: Benchmark test results [μs]: Linux 3.5.7 period task ($1sec$)

11.9.2.2. Task Switch Tests

As described in Section 5.2.1 two different tests for measuring task switch latency are implemented. Like before in the periodic task benchmarking these tests are performed in the scenarios described for a native Linux kernel and a HRTL system.

Listing 11.36 shows the implementation of the startup routine for the task preemption latency benchmark test. Two HRTL events (`fork_event` and `wait_event`) are used to synchronise the start of the test. Since, actually three different processes are involved in test executing, the results are stored in a shared memory segment. The initialisation of the semaphores and the shared memory segment are not shown in the listing.

switch_...signal.c

```
int main(int argc, char **argv) {
    struct hrtl_sched_fork hrtl_fork;
...
    if (0 > setup())
        exit(EXIT_FAILURE);
...
    libhrtl_set_task_prio(HRTL_PRIO_MAX);
    hrtl_fork.group_id = grp;
    hrtl_fork.param.prio = HRTL_PRIO_MAX - 1;
    hrtl_fork.param.runtime = HRTL_DEFAULT_TASK_RUNTIME;
    hrtl_fork.param.period = HRTL_PERIOD_MIN;
    libhrtl_sched_setfork(getpid(), &hrtl_fork);

    if (0 > libhrtl_sched_start_rt(getpid(), grp))
        exit(EXIT_FAILURE);

    if (0 == (high = fork()))
        task_high(1);
    else
        libhrtl_event_wait(fork_event);

    hrtl_fork.param.prio = HRTL_PRIO_MAX - 2;
    libhrtl_sched_setfork(getpid(), &hrtl_fork);

    if (0 == fork())
        task_low(0, high);
    else
        libhrtl_event_wait(fork_event);
...
    libhrtl_event_trigger(wait_event);
    libhrtl_set_task_prio(HRTL_PRIO_MIN);
...
```

```
}
```

Listing 11.36: HRTL task preemption benchmark test startup

Two processes are forked during test startup. According to Section 11.5.2 the new processes are placed in the real-time partition with the defined priority levels. One of the HRTL events (`fork_event`) is used here to let the main process block until the new created process finishes its own setup phase. After both processes have finished their startup the main process fires the second event (`wait_event`) and lowers its priority level. Both forked processes have higher priority than the main process now. If they terminate, the main process comes back to life and finishes the benchmark test by printing the results.

The main test takes place between the new created processes. The program code is not shown here. It is almost the same as for the Linux task creation benchmark test (Listing 6.8). Table 11.11 and Table 11.12 show the results of the test.

Scenario	Average	Min	Max	Gap	Deviation
Normal	1.517	1.510	1.526	0.016	0.002
CPU utilization	1.522	1.501	1.554	0.053	0.003
I/O utilization	1.529	1.520	1.544	0.024	0.002

Table 11.11.: Benchmark test results [μs]: HRTL preempt task (signal)

Scenario	Average	Min	Max	Gap	Deviation
Normal	1.624	1.620	1.675	0.056	0.007
CPU utilization	1.616	1.593	2.240	0.647	0.050
I/O utilization	1.596	1.583	1.703	0.120	0.010

Table 11.12.: Benchmark test results [μs]: Linux 3.5.7 preempt task (signal)

The main test is implemented in two different variants. One uses the POSIX signal mechanism to trigger the higher priority task. The other version of the test uses an additional HRTL event for triggering the higher priority task. Listing 11.37 shows the main routines for both tasks. After performing the startup synchronisation as mentioned above, the benchmark test starts with entering the `for` loop. `task_low` triggers the event (`preempt_event`) for waking up the higher priority task.

switch_...event.c

```
void task_low(int idx) {
...
    libhrtl_event_trigger(fork_event);
    libhrtl_event_wait(wait_event);

    for (i = 0; i < LOOP_COUNT; i++) {
        busy_long();
        cpuid();
```

```
        rdtsc_32(tsc[i]);
        libhrtl_event_trigger(preempt_event);
    }
...
}

void task_high(int idx) {
...
    libhrtl_event_trigger(fork_event);
    libhrtl_event_wait(wait_event);

    for (i = 0; i < LOOP_COUNT; i++) {
        busy_long();
        libhrtl_event_wait(preempt_event);
        rdtscp_32(tsc[i]);
        cpuid();
    }
...
}
```

Listing 11.37: HRTL task preemption benchmark test

The benchmark test was executed in the 3 scenarios described in Section 5.3. Table 11.13 shows the results of the test.

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.749	0.740	0.766	0.026	0.011
CPU utilization	0.751	0.736	0.786	0.050	0.014
I/O utilization	0.752	0.737	0.778	0.041	0.012

Table 11.13.: Benchmark test results [μs]: HRTL preempt task (event)

The second benchmark test for measuring the task switch latency is also described in Section 5.4.3.2. As explained in Section 6.5.2 the arrangement of the shared memory segment is more complicated compared to the task preemption time benchmark test. The layout for the shared memory segment can be seen in Listing 6.9.

The test startup is almost the same as for the task preemption benchmark test before. Details are not printed here. All processes needed for the test execution are forked within the main process and use the same synchronisation mechanism (HRTL events). The actual task switch is invoked by calling the `sched_yield()` system call.

Table 11.14 presents the results of the task switch latency benchmark test for the HRTL system with two alternating processes. The same test was repeated with 16 (Table 11.16), 128 (Table 11.18) and 512 (Table 11.20) switching processes. As can be seen, the time required for a task switch increases with more involved processes.

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.471	0.453	0.484	0.031	0.002
CPU utilization	0.469	0.451	0.494	0.043	0.002
I/O utilization	0.474	0.460	0.491	0.031	0.007

Table 11.14.: Benchmark test results [μs]: HRTL switch task (2 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.475	0.471	0.516	0.044	0.006
CPU utilization	0.475	0.471	0.521	0.050	0.007
I/O utilization	0.487	0.479	0.524	0.046	0.005

Table 11.15.: Benchmark test results [μs]: Linux 3.5.7 switch task (2 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.574	0.536	0.611	0.076	0.014

Table 11.16.: Benchmark test results [μs]: HRTL switch task (16 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.582	0.548	0.634	0.086	0.014

Table 11.17.: Benchmark test results [μs]: Linux 3.5.7 switch task (16 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.778	0.710	0.857	0.147	0.030

Table 11.18.: Benchmark test results [μs]: HRTL switch task (128 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	0.811	0.734	0.907	0.173	0.030

Table 11.19.: Benchmark test results [μs]: Linux 3.5.7 switch task (128 tasks)

Scenario	Average	Min	Max	Gap	Deviation
Normal	1.219	0.980	1.543	0.563	0.109

Table 11.20.: Benchmark test results [μs]: HRTL switch task (512 tasks)

11.9.2.3. Task Creation Test

The task creation benchmark test measures the time it takes for creating a new process. According to the description in Section 5.4.3.3 a new task is spawned in each test

Scenario	Average	Min	Max	Gap	Deviation
Normal	1.226	1.041	1.548	0.507	0.091

Table 11.21.: Benchmark test results [μs]: Linux 3.5.7 switch task (512 tasks)

step within the test main loop by calling the `fork()` system call. Time is measured immediately before and after (in the new process) invoking `fork()`. To transfer the second measurement value to the main process a shared memory segment is used. Listing 11.38 shows the implementation of the task creation benchmark test for the HRTL extension.

fork_high_prio.c

```
void task(void) {
...
    hrtl_fork.group_id = grp;
    hrtl_fork.param.prio = HRTL_PRIO_MAX - 1;
...
    libhrtl_sched_setfork(getpid(), &hrtl_fork);
...
    libhrtl_event_trigger(fork_event);
    libhrtl_event_wait(wait_event);
...
    for (i = 0; i < LOOP_COUNT; i++) {
        res->start[i].pid = pid;
        cpuid();
        rdtsc_32(tsc);

        if (0 == fork()) {
            rdtscp_32(tsc);
            cpuid();
            res->stop[i].pid = getpid();
            res->stop[i].tsc = tsc;
            exit(EXIT_SUCCESS);
        }

        res->start[i].tsc = tsc;
        busy_long();
        sched_yield();
    }
...
    libhrtl_event_trigger(wait_event);
...
}

int main(int argc, char **argv) {
...
    libhrtl_set_task_prio(HRTL_PRIO_MAX);
    hrtl_fork.group_id = grp;
    hrtl_fork.param.prio = HRTL_PRIO_MAX - 2;
...
    libhrtl_sched_setfork(getpid(), &hrtl_fork);
...
```

```
    if (0 == fork())
        task();
    else
        libhrtl_event_wait(fork_event);
...
    libhrtl_event_trigger(wait_event);
    libhrtl_event_wait(wait_event);
...
}
```

Listing 11.38: HRTL task creation benchmark test

The startup procedure is almost the same as for the previously introduced tests. `task()` implements the main test function. In each iteration of the `for` loop, a new process is created. The new process starts with a higher priority level according to the definition in `hrtl_fork.param.prio`.

The results of the task creation benchmark test are shown in Table 11.22 and Table 11.23. The measured times in a HRTL extension kernel are slightly higher compared to a native Linux kernel. This is because adding the new task to the defined partition creates some overhead.

Scenario	**Average**	**Min**	**Max**	**Gap**	**Deviation**
Normal	28.011	25.067	31.864	6.798	1.436
CPU utilization	31.898	28.509	37.138	8.629	1.612
I/O utilization	27.861	25.212	33.284	8.072	1.493

Table 11.22.: Benchmark test results [μs]: HRTL task creation

Scenario	**Average**	**Min**	**Max**	**Gap**	**Deviation**
Normal	24.059	23.487	29.045	5.558	0.494
CPU utilization	28.669	26.721	32.790	6.069	1.139
I/O utilization	24.484	23.771	30.203	6.432	0.639

Table 11.23.: Benchmark test results [μs]: Linux 3.5.7 task creation

11.9.2.4. Interrupt Tests

The implementation of the interrupt benchmark tests as described in Section 5.4.3.4 are explained in this section. For the realisation of the tests, it is necessary to enhance the kernel with a module which implements the interrupt handler. The module uses the HRTL interrupt mechanism as explained in Section 11.6.3. Listing 11.39 shows the initialisation function of the module.

irq_benchmark.c

```
static enum
hrtl_irq_result irq_latency_hw_handler(unsigned int irq_nr,
```

```
                                                        struct hrtl_hw_irq *hw_irq)
{
        rdtsc_32(tsc);
        return HRTL_IRQ_HANDLED;
}

static struct hrtl_irq irq_latency_hw_irq = {
        .name = "latency_hw_irq",
        .handler = irq_latency_hw_handler,
};

static int __init init_interrupt_latency(void)
{
...
        hrtl_hw_irq_init(&irq_latency_hw_irq);
        retval = hrtl_hw_irq_connect(&irq_latency_hw_irq, PARALLEL_IRQ);
...
}
```

Listing 11.39: HRTL benchmark interrupt handler

After loading the module, an interrupt handler is registered for the parallel port. For the interrupt latency and interrupt dispatch latency benchmarks the handler just captures the current value of the TSC register and returns. For the interrupt to task latency benchmarks the handler has to be extended (Listing 11.40). This is triggered via the proc interface. A HRTL key for a partition is received by the module's proc `write()` function. The module then registers a threaded interrupt handler in that partition.

irq_benchmark.c

```
static enum
hrtl_irq_result irq_latency_handler(struct hrtl_irq *irq)
{
        rdtscp_32(tsc);
        cpuid();
        return HRTL_IRQ_HANDLED;
}

static struct hrtl_irq irq_latency_irq = {
        .name = "latency_irq",
        .handler = irq_latency_handler,
};

static int
proc_write_interrupt_latency(struct file *file, const char *buffer,
                             unsigned long count, void *data)
{
...
        ret = kstrtou32(tmp, 16, &group_key);
...
        hrtl_irq_init(&irq_latency_irq);
        ret = hrtl_irq_connect(&irq_latency_irq, &irq_latency_hw_irq);
...
        ret = hrtl_irq_register_group(&irq_latency_irq, group_key);
```

```
...
}
```

Listing 11.40: HRTL benchmark threaded interrupt handler

Values between the main benchmark test and the measurements inside the interrupt handler respectively threaded interrupt handler are transmitted via the proc interface. The module will create a file `/proc/interrupt_latency`. A simple read on that file will return the result of the last measurement. It is important for test execution to bind the interrupt treatment of the parallel port interrupt to a certain CPU. This is done inside the benchmark test executable by calling `libhrtl_system_request_irq()`.

With the introduced interrupt handler, measuring the interrupt latency is quite simple. The implementation is similar to the native Linux interrupt latency benchmark test (Listing 6.14). Only the test startup is different in order to create a partition on a reserved CPU. The results of the benchmark test are shown in Table 11.24 and Table 11.25.

Scenario	Average	Min	Max	Gap	Deviation
Normal	4.684	3.347	6.093	2.747	0.677
CPU utilization	4.760	3.385	6.283	2.898	0.668
I/O utilization	4.815	3.392	6.723	3.331	0.727

Table 11.24.: Benchmark test results [μs]: HRTL interrupt latency (ISR)

Scenario	Average	Min	Max	Gap	Deviation
Normal	4.642	3.257	6.069	2.812	0.686
CPU utilization	6.600	3.992	8.443	4.451	0.867
I/O utilization	4.763	3.315	7.423	4.108	0.727

Table 11.25.: Benchmark test results [μs]: Linux 3.5.7 interrupt latency (ISR)

The interrupt dispatch latency benchmark test is similar to the interrupt latency benchmark test except for the time measurement points. For this test the first value is captured within the kernel. The second time value is gathered when returning from interrupt. The results of the interrupt dispatch latency benchmark test are provided in Table 11.26 and Table 11.27.

The interrupt to task latency benchmark test is identical to the interrupt latency benchmark test and is not listed here. The results of the test are shown in Table 11.28 and Table 11.29.

Scenario	Average	Min	Max	Gap	Deviation
Normal	1.616	1.020	2.237	1.217	0.289
CPU utilization	1.610	0.990	2.286	1.296	0.319
I/O utilization	1.649	1.030	2.413	1.383	0.324

Table 11.26.: Benchmark test results [μs]: HRTL interrupt latency (dispatch)

Scenario	Average	Min	Max	Gap	Deviation
Normal	1.429	0.926	2.120	1.194	0.303
CPU utilization	8.403	1.426	13.760	12.334	2.711
I/O utilization	1.708	0.947	5.048	4.101	0.836

Table 11.27.: Benchmark test results [μs]: Linux 3.5.7 interrupt latency (dispatch)

Scenario	Average	Min	Max	Gap	Deviation
Normal	5.178	3.908	6.531	2.623	0.615
CPU utilization	5.212	3.815	6.778	2.963	0.677
I/O utilization	5.270	3.887	7.639	3.752	0.805

Table 11.28.: Benchmark test results [μs]: HRTL interrupt latency (SLIH)

Scenario	Average	Min	Max	Gap	Deviation
Normal	4.961	3.498	6.546	3.048	0.699
CPU utilization	8.471	5.502	13.770	8.267	1.109
I/O utilization	5.140	3.618	7.748	4.129	0.838

Table 11.29.: Benchmark test results [μs]: Linux 3.5.7 interrupt latency (SLIH)

Part IV.

Evaluation

12 Benchmark Results Comparison

In the previous chapters a real-time extension for the Linux operating system was introduced. The results of the evaluation framework from Chapter 5 for the new extension and the systems discussed are compared in this chapter. Detailed information on the test implementation for each operating system and their results can be found in the related sections: RT-Preempt Section 6.5, HLRT Section 7.5, QNX Neutrino Section 8.4 and HRTL Section 11.9.2.

Benchmark results are summarized in the tables included in the following paragraphs. Time values are given in microseconds (μs). The first value for each measuring result is the standard deviation from the average. The average is given in brackets beside the standard deviation: $\sigma(A)$. As described in Section 5.3, some tests were executed in different scenarios which is constituted by the last column in each table.

Period	**HRTL**	**HLRT**	**RT-Preempt**	**QNX**	Scen.
500μs	0.022 (499.952)	0.035 (499.936)	**0.969** (499.931)	10.966 (499.807)	
	0.040	0.558	1.125	10.895	CPU
	0.147	0.837	1.198	11.523	I/O
10ms	0.029	0.478	0.681	44.213	
	0.026	1.753	2.142	44.255	CPU
	0.104	3.265	1.722	44.161	I/O
100m	0.023	1.314	2.187	46.676	
1sec	0.022	0.682	**2.526**	41.930	

Table 12.1.: Period benchmark results overview [μs]

Table 12.1 provides the measuring results of the scheduling precision tests for periodic tasks. Each system meets the expected period in the average with a difference of some nano seconds. The results of the external measurement as described in Section 5.4.1 are not presented here. They are included in the complete measurment results and available to download.

The scheduling precision for a periodic task in the HRTL system is very high

even in the utilization scenarios. The maximum deviation from an expected timer is 147 nanoseconds while the system is under heavy I/O load. Compared to the other systems, the HRTL kernel does not lose precision with increasing timers. Especially in the RT-Preempt kernel the timer precision deteriorates by a factor of 2.6 between 500 nanoseconds and 1 second timers.

The results presented in Table 12.2 show the time it takes to create a new process (`fork()`) in the particular systems. For the HRTL kernel, the test was executed in static partitions as well as in dynamic partitions. In the CPU utilization scenario the required time slightly increases for the Linux based systems. This is due to the fact that several tasks are added to the system within the scenario. However, what is striking is the fact that the required time for task creation in the QNX Neutrino system is much higher than the time required in Linux based systems.

Partition	**HRTL**	**RT-Preempt**	**QNX**	Scenario
static/none	1.436 (28.011)	2.228 (35.031)		
	1.612 (31.898)	1.478 (37.148)		CPU
	1.493 (27.861)	2.253 (35.269)		I/O
dynamic	1.453 (28.916)		5.008 (186.772)	
	1.834 (33.295)		18.120 (170.843)	CPU
	1.777 (29.478)		31.121 (329.973)	I/O

Table 12.2.: Process creation benchmark results overview [μs]

The HRTL system shows a very constant rate when preempting task execution. In contrast to other systems, the CPU and I/O utilization scenarios have almost no impact on the test. The highest effect can be seen in the QNX system between an unloaded system and the I/O utilization scenario. Table 12.3 presents the task preemption benchmark results for the HRTL and the QNX systems in two variants. As described in Section 5.4.3.2, different synchronization events are used for these systems. For both systems, using a special method for process synchronization offered by the operating system, the required time for task preemption is almost halved.

Preempting a task that runs in a dynamic partition in the HRTL kernel takes slightly longer compared to static partitions. The dynamic partition scheduler is more complex than a scheduler module encapsulated in a static partition. The scheduler has to make decisions according to the calculated CPU usage and decides which partition should run in which time slot. The overhead can be seen in the benchmark results.

Table 12.4 summarises the results of the task switch latency tests. The task switch latency is measured in four stages. In each stage the number of active tasks is increased (first column). As can be seen, the time required for a task switch increases with more involved processes. Like before in the task preemption benchmarks (Table 12.3) the HRTL kernel has a very constant rate for switching tasks (within one stage). However, the RT-Preempt system also provides a high reproducibility. Only the QNX system

Type	Partition	**HRTL**	**RT-Preempt**	**QNX**	Scenario
signal	static/none	0.002 (1.517)	0.222 (4.699)		
		0.003 (1.522)	0.253 (4.728)		CPU
		0.002 (1.529)	0.276 (4.776)		I/O
signal	dynamic	0.009 (2.047)		0.316 (1.004)	
		0.012 (2.058)		0.384 (1.030)	CPU
		0.010 (2.040)		0.850 (3.023)	I/O
event	static/none	0.011 (0.749)			
		0.014 (0.751)			CPU
		0.012 (0.752)			I/O
event	dynamic	0.010 (1.252)		0.128 (0.419)	
		0.015 (1.257)		0.231 (0.456)	CPU
		0.015 (1.258)		0.591 (1.461)	I/O

Table 12.3.: Preemption benchmark results overview [μs]

seems to be affected by the utilization scenarios. Particularly a high I/O load of the system has an impact of the task switch latency.

Tasks	Partition	**HRTL**	**RT-Preempt**	**QNX**	Scenario
2	static/none	0.002 (0.471)	0.004 (0.617)		
		0.002 (0.469)	0.007 (0.616)		CPU
		0.007 (0.474)	0.004 (0.623)		I/O
2	dynamic	0.008 (0.723)		0.176 (0.440)	
		0.009 (0.737)		0.196 (0.463)	CPU
		0.009 (0.726)		0.547 (1.129)	I/O
16	static/none	0.014 (0.574)	0.016 (0.698)		
16	dynamic	0.012 (0.828)		0.196 (0.465)	
128	static/none	0.030 (0.778)	0.037 (0.910)		
128	dynamic	0.024 (1.039)		0.583 (0.639)	
512	static/none	0.109 (1.219)	0.114 (1.309)		
512	dynamic	0.106 (1.488)		2.008 (0.824)	

Table 12.4.: Task switch benchmark results overview [μs]

For the HRTL kernel task switches in dynamic partitions need more time compared to static partitions. As already mentioned above, this is because of the overhead

needed to balance dynamic partitions according to used CPU usage in the system.

Table 12.4 shows that the time required for a task switch increases with more involved processes. This is caused by cache misses. The more processes there are to switch between, the more cache misses occur. As the number of active processes increases, the caching effect becomes evident since the thread context will no longer be able to reside in the cache. To illustrate the effect of the cache, the task switch benchmark tests were executed again for the HRTL system with cache disabled. The results are presented in Table 12.5. It can be seen that the overall time requiered for a task switch is much higher compared to Table 12.4, but the number of active processes has no impact on the task switch time.

Tasks in the queue	2	16	128	512
Time requiered in μs	130.214	130.258	130.208	130.265

Table 12.5.: HRTL task switch results without caching

The results of the interrupt related benchmark test are shown in Table 12.6. According to the description in Section 5.2.1 the effect of interrupts is measured in three different variants (first column). For the Linux based systems, the utilization scenarios have a higher impact on the interrupt latency compared to the QNX system. QNX shows a great stability on scheduling second level interrupt handlers. However, the HRTL and the HLRT extension provide a better average time for interrupt latency of low level interrupt handlers.

Type	**HRTL**	**HLRT**	**RT-Preempt**	**QNX**	Scenario
ISR	0.677 (4.684)	0.690 (4.610)	0.651 (9.282)	0.436 (6.291)	
	0.668 (4.760)	0.782 (5.383)	0.657 (9.321)	0.380 (6.287)	CPU
	0.727 (4.815)	0.690 (4.625)	0.668 (9.378)	0.626 (5.871)	I/O
dispatch	0.289 (1.616)	0.317 (1.387)	0.678 (6.573)	0.514 (1.348)	
	0.319 (1.610)	2.911 (5.861)	0.666 (6.616)	0.782 (1.429)	CPU
	0.324 (1.649)	0.322 (1.415)	0.682 (6.625)	1.130 (2.271)	I/O
SLIH	0.615 (5.178)		0.639 (9.337)	0.997 (9.967)	
	0.677 (5.212)		0.649 (9.392)	1.092 (10.109)	CPU
	0.805 (5.270)		0.695 (9.460)	1.228 (10.607)	I/O

Table 12.6.: Interrupt benchmark results overview [μs]

The benchmark test results presented in this chapter show the high potential of Linux operating systems for real-time purposes. It should be mentioned that in the discussion on the QNX Neutrino operating system and presentation of the benchmark result some main features were not considered (e.g. QNX as distributed system). An overall conclusion of this work is presented in the next chapter (Chapter 13).

13 Conclusion and Outlook

In this thesis, a new real-time patch for the Linux operating system has been developed. Design decisions are mainly based on detailed analysis of well known real-time Linux extensions and full real-time operating systems. This last chapter summarizes the main results of this thesis and points out future work.

Over the past decades, significant efforts have been invested in the design and development of real-time operating systems. These efforts have been carried out to either develop such a system from scratch or to extend or adapt an existing operating system such as Linux so as to enhance it with real-time capabilities. Our own work follows the latter line of research. Though it has in principle been known that the Linux kernel can be extended in such ways (RT-Preempt, HLRT and RT-Linux are examples of such approaches) these systems have significant drawbacks in terms of either functionality or performance.

Our own work builds on these works by integrating key features such as (dynamic and static) partitioning, scheduling and asynchronous system-call handlers into the Linux kernel in novel ways. We have shown that this approach can significantly improve the performance of Linux providing real-time features and at the same time extends the feature-set compared to existing approaches. The core technique that leads to these improvements is the combination of partitioning with preemptible kernel paths. Based on this modification of the kernel, features such as asynchronous system call distribution can straightforwardly be implemented.

A consequence of our architecture is that scheduling strategies can dynamically be loaded as kernel modules and applied to process partitions. Our implementation contains several scheduling strategies, namely FIFO, Round-Robin and Rate-Monotonic. The Rate-Monotonic scheduler, however, has been implemented only for static partitioning. Thus, two opportunities for future research directions appear worthwhile:

1. Further scheduling strategies should be integrated and their impact should be evaluated, as it is unclear which approach exhibits the best performance results in practical applications.

2. In particular, deadline-oriented scheduling techniques such as Rate-Monotonic or Earliest-Deadline-First scheduling should be evaluated for dynamic partitioning.

Currently, each partition is assigned exactly one scheduling time-slot. Generalizing this scheme so that each partition may be associated with a collection of time-slots provides additional flexibility such as executing tasks from one static partition on several CPUs in parallel. Furthermore, a generalized mapping from partitions to time-slots would support a greater flexibility for designing static scheduling plans, since one partition could be included in one plan multiple times with different runtimes. It should thus be evaluated how this generalization can effectively be integrated into our kernel extension.

The system call distribution is based on work packages and system-call handlers. It should be evaluated how the design of these handler threads can be extended, so that occourring interrupts can also be managed as work packages. Treating interrupts as work packages would provide a greater flexibility for prioritized interrupt handling.

Bibliography

[AB98] Alia Atlas and Azer Bestavros. Statistical rate monotonic scheduling. In *Proceedings of the 19th Real-Time Systems Symposium*, pages 123–132. IEEE Computer Society, 1998.

[Alt] Altreonic NV. OpenComRTOS. `http://www.altreonic.com`. Retrieved February 2012.

[BC05] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel - from I/O ports to process management*. O'Reilly Media, Inc., 3rd edition, 2005.

[Bla] BlackBerry Ltd. QNX Neutrino. `http://www.qnx.com`. Retrieved February 2012.

[BM14] Ivan Cibrario Bertolotti and Gabriele Manduchi. *Real-Time Embedded Systems: Open-Source Operating Systems Perspective*. CRC Press, Inc., 2014.

[BMS93] Özalp Babaoğlu, Keith Marzullo, and Fred B. Schneider. A formalization of priority inversion. *Real-Time Systems*, 5(4):285–303, 1993.

[BW09] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational, 4th edition, 2009.

[CL07] Walter Cedeño and Phillip A. Laplante. An overview of real-time operating systems. *Journal of Laboratory Automation*, 12(1):40–45, 2007.

[Com08] Portable Application Standards Committee. Portable Operating System Interface - IEEE Std 1003.1-2008, September 2008.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers - where the Kernel meets the hardware*. O'Reilly Media, Inc., 3rd edition, 2005.

[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.

[DDCa] DDC-I, Inc. Deos 653. `http://www.ddci.com/products_deos.php`. Retrieved December 2011.

[DDCb] DDC-I, Inc. HeartOS. `http://www.ddci.com/products_heartos.php`. Retrieved December 2011.

[Dev] Develer s.r.l. BeRTOS - not only a kernel. http://www.bertos.org. Retrieved December 2011.

[Dip] Dipartimento di Ingegneria Aerospaziale. Real Time Application Interface. http://www.rtai.org. Retrieved February 2012.

[dMC00] António J. Pessoa de Magalhães and Carlos J. A. Costa. Real-time scheduling models: an experimental approach. In *4th Portuguese Conference on Automatic Control*. APCA, 2000.

[DW05] Sven-Thorsten Dietrich and Daniel Walker. The evolution of real-time linux. In *Proceedings of the 7th Real-Time Linux Workshop*, 2005.

[eCo] eCosCentric Limited. eCos - embedded configurable operating system. http://ecos.sourceware.org/. Retrieved November 2011.

[Efk05] Christof Efkemann. Development and evaluation of a hard real-time scheduling modification for linux 2.6. Diplomarbeit, Universität Bremen, 2005.

[Efk14] Christof Efkemann. *A Framework for Model-based Testing of Integrated Modular Avionics*. PhD thesis, Universität Bremen, 2014.

[Ene] Enea. OSE - Operating System Embedded. http://www.enea.com/ose. Retrieved November 2011.

[Evi] Evidence Srl. ERIKA Enterprise. http://erika.tuxfamily.org. Retrieved November 2011.

[Exp] Express Logic Inc. ThreadX - Real-Time Operating System. http://rtos.com/products/threadx. Retrieved February 2012.

[FCTS09] Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. An EDF scheduling class for the Linux kernel. In *11th Real-Time Linux Workshop*, pages 1–8, 2009.

[FGR+90] Borko Furht, Dan Grostick, Guy Rabbat, John Parker, David Gluch, and Meg McRoberts. *Real-Time UNIX Systems: Design and Application Guide*. Kluwer Academic Publishers, 1990.

[GAGB01] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo. A new kernel approach for modular real-time systems development. In *13th Euromicro Conference on Real-Time Systems*, pages 199–206, 2001.

[Gar03] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *In Proceedings of the Network and Distributed Systems Security Symposium*, pages 163–176, 2003.

[GPR04] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *In Proceedings of the Network and Distributed Systems Security Symposium*. The Internet Society, 2004.

[Gre] Green Hills Software. Integrity - Real-Time Operating System. `http://www.ghs.com/products/rtos/integrity.html`. Retrieved January 2012.

[HBG+06] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: a highly reliable, self-repairing operating system. *Operating Systems Review*, 40(3):80–89, 2006.

[HGCD00] Wolfgang A. Halang, Roman Gumzej, Matjaz Colnaric, and Marjan Druzovec. Measuring the performance of real-time systems. *Real-Time Systems*, 18(1):59–68, 2000.

[Hil92] Dan Hildebrand. An architectural overview of qnx. *Usenix Workshop on Micro-Kernels & Other Kernel Architectures*, 1992.

[JCLC06] Kerry Johnson, Jason Clarke, Paul Leroux, and Robert Craig. *OS Partitioning for Embedded Systems*. QNX Software Systems, 2006.

[Jon91] R.S. Jones. *The C Programmer's Companion: ANSI C Library Functions*. Silicon Press, 1991.

[KB03] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[KP89] Rabindra P. Kar and Kent Porter. Rhealstone: A real-time benchmarking proposal. *Dr. Dobb's Journal of Software Tools*, 14(2):14–24, 1989.

[KW07] Robert Kaiser and Stephan Wagner. Evolution of the pikeos microkernel. In *1st International Workshop on Microkernels for Embedded Systems*, pages 50–57, 2007.

[LAK92] J.C. C. Laprie, A. Avizienis, and H. Kopetz. *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc., 1992.

[Law] Kelvin Lawson. Atomthreads: Open Source RTOS. `http://www.atomthreads.com`. Retrieved December 2011.

[Leh90] John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the Real-Time Systems Symposium*, pages 201–209. IEEE Computer Society, 1990.

[Lei07] Bernhard Leiner. A partitioning operating system based on rtai-lxrt linux. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, 2007.

[Liu00] Jane W. S. Liu. *Real-Time Systems*. Pearson Education, 2000.

[LL73] Chang L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[Lov10] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[LR80] Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa. *Commun. ACM*, 23(2):105–117, 1980.

[LSOH07] Bernhard Leiner, Martin Schlager, Roman Obermaisser, and Bernhard Huber. A comparison of partitioning operating systems for integrated systems. *Computer Safety, Reliability, and Security*, 4680:342–355, 2007.

[Lyn] Lynx Software Technologies, Inc. LynxOS RTOS. `http://www.lynx.com/products/real-time-operating-systems/lynxos-rtos/`. Retrieved December 2011.

[MFL+09] Antonio Mancina, Dario Faggioli, Giuseppe Lipari, Jorrit N. Herder, Ben Gras, and Andrew S. Tanenbaum. Enhancing a dependable multiserver operating system with temporal protection via resource reservations. *Real-Time Systems*, 43(2):177–210, 2009.

[MG] Ingo Molnar and Thomas Gleixner. RT_Preempt Patch for Linux. `https://rt.wiki.kernel.org`. Retrieved January 2012.

[Mica] Micrium Embedded Software. μC/OS-III. `http://micrium.com/rtos/ucosiii`. Retrieved December 2011.

[Micb] Microsoft Corporation. Windows Embedded CE. `http://msdn.microsoft.com/en-us/windowsembedded`. Retrieved December 2011.

[Mol07] Ingo Molnar. Modular Scheduler Core and Completely Fair Scheduler (CFS). `https://lkml.org/lkml/2007/4/13/180`, 2007.

[Mon] MontaVista Software. MontaVista Linux. `http://www.mvista.com/product_detail_mvl6.php`. Retrieved February 2012.

[MT92] Clifford Mercer and Hideyuki Tokuda. Preemptibility in real-time operating systems. In *Proceedings of the Real-Time Systems Symposuim*, pages 78–87. IEEE Computer Society, 1992.

[Noe05] Tammy Noergaard. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Newnes, 2005.

[Ott06] Aliki Ott. *System Testing in the Avionics Domain*. PhD thesis, Universität Bremen, 2006.

[Pao10] Gabriele Paoloni. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*. Intel Corporation, September 2010.

[Rad14] Dirk Radder. x86 Benchmark Framework. `http://www.informatik.uni-bremen.de/agbs/dirkr/HRTL/benchmark.tgz`, November 2014.

[Rad15a] Dirk Radder. Hard Real-Time Linux - real-time extension kernel patch, Version 0.18. `http://www.informatik.uni-bremen.de/agbs/dirkr/HRTL/patch-3.5.7-hrtl-0.18.gz`, April 2015.

[Rad15b] Dirk Radder. Hard Real-Time Linux - user-space library, Version 0.2. `http://www.informatik.uni-bremen.de/agbs/dirkr/HRTL/libhrtl_0.2.tgz`, April 2015.

[RB10] Sergio A. Rodriguez and Phillip M. Burt. A latency model of linux 2.6 for digital signal processing in real time. In *12th Real-Time Linux Workshop*, pages 1–8. OSADL, 2010.

[Rea] Real Time Engineers Ltd. FreeRTOS. `http://www.freertos.org`. Retrieved January 2012.

[RH07] Steven Rostedt and Darren V. Hart. Internals of the rt patch. *Proceedings of the Linux Symposium*, 2, June 2007.

[Rowa] RoweBots Limited. DSPnano Embedded Real Time Operating System. `http://www.rowebots.com/products/dspnano_rtos`. Retrieved November 2011.

[Rowb] RoweBots Research Inc. Unison RTOS for Microcontrollers. `http://www.rowebots.com/products/unison_rtos`. Retrieved February 2012.

[RTC92] RTCA, Inc. DO-178B: Software Considerations in Airborne Systems and Equipment Certification, December 1992.

[SEG] SEGGER Microcontroller Systems LLC. embOS - real-time operating system. `http://www.segger.com/embos.html`. Retrieved November 2011.

[Sir] Giovanni Di Sirio. ChibiOS/RT. `http://www.chibios.org`. Retrieved December 2011.

[SRS98] John A. Stankovic, Krithi Ramamritham, and Marco Spuri. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[Ste02] David B. Stewart. Measuring execution time and real-time performance. In *Proceedings of the Embedded Systems Conference*, pages 1–15, 2002.

[SYS] SYSGO AG. PikeOS. http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept. Retrieved February 2012.

[Sys05] QNX Software Systems. *QNX Neutrino RTOS V6.3 System Architecture*. QNX Software Systems Co, for release 6.3.0 or later edition, 2005.

[Tan08] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson, 3rd edition, 2008.

[TNR90] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-time mach: Towards a predictable real-time system. In *In Proceedings of the USENIX MACH Symposium*, pages 73–82. USENIX, 1990.

[Uni] University of Kansas. KU Real Time Linux. http://www.ittc.ku.edu/kurt. Retrieved November 2011.

[Wik13] Source Wikipedia. *X86 Architecture: X86, Ia-32, Pentium Fdiv Bug, Hyper-Threading, Wintel, Icomp, X86-64, X86 Assembly Language, X86 Calling Conventions, X86 Virtualiza*. General Books LLC, 2013.

[Win] Wind River Systems Inc. VxWorks. http://www.windriver.com/products/vxworks. Retrieved December 2011.

[YB] Victor Yodaiken and Michael Barabanov. RTLinux. http://www.rtlinuxfree.com. Retrieved December 2011.

[YB97] V. Yodaiken and M. Barabanov. A real-time linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, volume 34, 1997.

[YMBYG08] K. Yaghmour, J. Masters, G. Ben-Yossef, and P. Gerum. *Building Embedded Linux Systems*. O'Reilly Media, Inc., 2nd edition, 2008.

[Zwe02] Klaas-Henning Zweck. Kernelbasierte Echtzeiterweiterung eines Linux-Multiprozessor-Systems. Diplomarbeit, Universität Bremen, 2002.