

SLOTHFUL LINUX: An Efficient Hybrid Real-Time System by Hardware-Based Task Dispatching

Diplomarbeit im Studiengang Informatik

vorgelegt von

Rainer Müller

geboren am 14. Februar 1987 in Erlangen

Angefertigt am

Lehrstuhl für Informatik 4 – Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:

**Dipl.-Inf. Wanja Hofer
Dr.-Ing. Daniel Lohmann
Dr.-Ing. Fabian Scheler
Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**

Beginn der Arbeit: 01.01.2012
Abgabe der Arbeit: 02.07.2012

Abstract

The particular requirements raised in the embedded market lead to the development of real-time operating systems, which differ from the common general-purpose operating systems by applying strict timing and providing predictable execution of tasks. Some features of a real-time application, such as human interaction and some data transfers to other components, are not time-critical and thus, can be executed by a general-purpose operating system in order to reduce the footprint of the real-time system. As the general-purpose operating system requires access to the data gathered by the real-time operating system, these two should be consolidated on the same platform forming a hybrid system.

This thesis examines how the existing SLOTH approach of interrupt-driven scheduling can be applied to a combined system running both a real-time core and the Linux kernel concurrently on the same hardware. This entails a review of existing approaches towards a real-time capable Linux and the design of a communication channel between the real-time core and processes running in Linux. The evaluation of the resulting SLOTHFUL LINUX is supposed to show the advantages of the SLOTH approach in a hybrid system compared to both the standalone SLOTH and another real-time extension for Linux.

Zusammenfassung

Die speziellen Anforderungen von eingebetteten Systemen haben zur Entwicklung von Echtzeitbetriebssystemen geführt, die sich von den üblichen Allzweckbetriebssystemen durch striktes Einhalten von Zeitvorgaben und vorhersagbarer Ausführung von Tasks unterscheiden. Einige Teile der Echtzeitanwendung, unter anderem Benutzerinteraktionen oder Datentransfer zu anderen Hardwarekomponenten, sind nicht zeitkritisch und können daher in ein Allzweckbetriebssystem ausgelagert werden. Dadurch verringert sich der Ressourcenbedarf der Echtzeitanwendung. Um dem Allzweckbetriebssystem Zugriff auf die erfassten Daten des Echtzeitbetriebssystems zu ermöglichen, sollten diese auf derselben Plattform als hybrides System vereint werden.

Diese Arbeit untersucht, wie das existierende SLOTH-Konzept für das interrupt-gesteuerte Einplanen von Tasks auf ein kombiniertes System übertragen werden kann, in dem neben einem Echtzeitbetriebssystem gleichzeitig der Linux-Kernel auf derselben Hardware ausgeführt wird. Dies umfasst sowohl die Analyse existierender Ansätze für ein echtzeitfähiges Linux als auch die Entwicklung eines Kommunikationskanals zwischen dem Echtzeitkern und Prozessen unter Linux. Die Evaluation des resultierenden SLOTHFUL LINUX soll aufzeigen, welche Vorteile das SLOTH-Konzept in einem Hybridsystem im Vergleich mit dem freistehenden SLOTH und einer weiteren Echtzeiterweiterung für Linux bietet.

Contents

1	Introduction	1
1.1	Real-Time Operating Systems	1
1.2	Consolidation of Real-Time and General-Purpose Operating Systems . . .	2
1.3	Goals of this Thesis	3
1.4	The Linux Kernel	4
1.5	The Sloth Real-Time Kernel	4
1.5.1	The Sloth Concept	4
1.5.2	Requirements on the Interrupt Controller	7
1.6	Outline of this Thesis	7
2	Problem Analysis	8
2.1	Requirements on an RTOS	8
2.2	Existing Projects based on Linux	9
2.2.1	The RT-Preempt Patch	9
2.2.2	The I-Pipe Patch for Generic Interrupt Virtualization	10
2.2.3	Summary	12
2.3	Integration of SLOTH into Linux	12
2.4	Summary	12
3	Design and Implementation of Sloth on the Intel x86	14
3.1	Sloth Design Overview	14
3.2	The Intel x86 Interrupt Subsystem	15
3.3	Utilization of the Local APIC	18
3.3.1	Tasks	19
3.3.2	Resource Management	21
3.4	Summary	21

4	Design and Implementation of Slothful Linux	22
4.1	Design of SLOTHFUL LINUX	22
4.1.1	Tasks and Resources	22
4.1.2	Pipes	23
4.1.3	Real-Time Applications as Linux Kernel Modules	23
4.1.4	Summary	24
4.2	Implementation of SLOTHFUL LINUX	24
4.2.1	Modifications to the Standard Linux Kernel	24
4.2.2	Tasks	27
4.2.3	Resources	29
4.2.4	Pipes	29
4.2.5	Summary	32
5	Evaluation	33
5.1	Evaluation Metrics	33
5.2	Platform Setup	34
5.3	Performance Evaluation	35
5.3.1	Performance Benchmarking with the Time-Stamp Counter	35
5.3.2	Results of the Performance Evaluation	37
5.3.3	Summary of the Performance Evaluation	40
5.4	Evaluation of Interrupt Latency	40
5.4.1	Latency Measurements with the Local APIC Timer	40
5.4.2	Results of the Interrupt Latency Evaluation	41
5.4.3	Summary of the Interrupt Latency Evaluation	44
5.5	Limitations	44
5.6	Summary	45
6	Conclusion	46
	Bibliography	48

Chapter 1

Introduction

In the last decades, computers have become a ubiquity around us as we are using them on a daily basis for both work and entertainment. Furthermore, many items around us are already being controlled by computers hidden from our view. The behavior of machines such as cars, aircrafts, industrial workbenches, and even home appliances is conducted by one or multiple embedded devices that are responsible for controlling the machine's movement. Depending on the operation, this can either happen autonomously or according to user input. When such tasks have strict timing requirements, they are classified as *real-time* and are usually performed by control hardware with well-adapted operating systems.

1.1 Real-Time Operating Systems

Human interaction with the computer has lead to the development of many different operating systems to accomplish the varying tasks required in the system. The well-recognized general-purpose operating systems (GPOS) are being used in the interaction with users and allow us to arrange things like world-wide network communication.

However, in embedded devices different requirements call for different operating systems. Especially, decisions for control in safety-critical systems have to be made in a reliable way to avoid damages or harm to both machines and humans. To ensure these properties, specific design criteria are applied to the class of real-time operating systems (RTOS). Although the term *real-time* in their name might suggest otherwise, these operating systems are not necessarily required to act very fast, instead the key is that the handling of events always takes a limited, predictable amount of time. This allows to plan program execution in the correct order, so that the reaction to physical events is always on time. An RTOS can operate on varying timescales, such as motor controllers operating in a time range of a few microseconds, multimedia devices with requirements in the range of milliseconds, or industrial plants where processes can take seconds, minutes, or even hours.

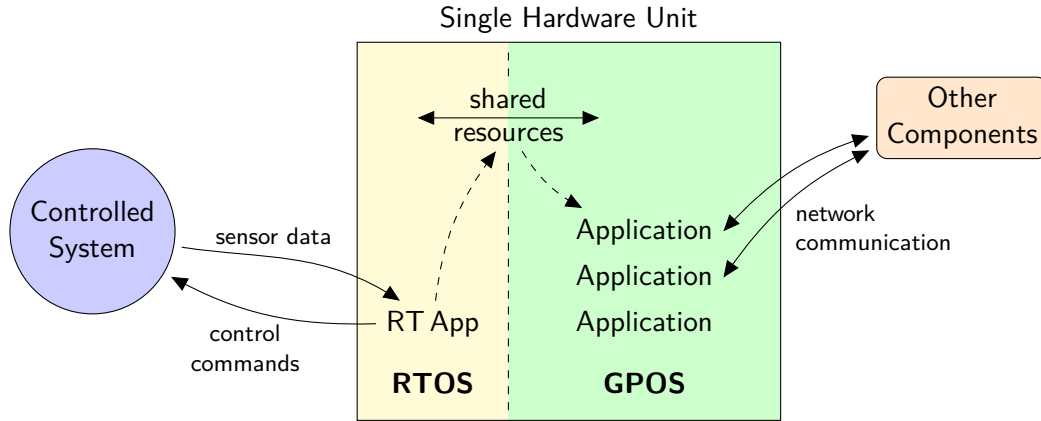


Figure 1.1: Schematic implementation of a hybrid system which runs both an RTOS and a GPOS on a single hardware platform.

In general, RTOS can be divided into the categories of *soft* and *hard*. In a soft real-time system, missing a few deadlines can be tolerated and therefore, the result might be of lower quality. For example in a video display, occasionally losing a single frame will go almost unnoticeable due to the dynamics of the human eye. The average performance of the system will still be acceptable, although a deadline has been missed [1].

In contrast, a hard real-time system guarantees timing constraints with strict deadlines that cannot be missed under any circumstances. In such a system, missing a deadline cannot be tolerated and would raise an exception to be handled by the application. For example, the internal state of a real-time application controlling an assembly line must always be synchronized with the actual state of the manufacturing system. Any control actuations in this system must be issued in a specific time frame to ensure the resulting product will match the desired outcome.

1.2 Consolidation of Real-Time and General-Purpose Operating Systems

In the past, hard real-time applications have usually been implemented using industrial components such as microcontrollers and signal processors dedicated to a single task. For safety-critical applications, the deployment of a real-time system needs to pass through multiple verification steps to ensure the correctness of the system.

However, some tasks related to the implementation of a real-time application do not need to be handled in the RTOS itself and can be moved out to reduce the amount of code to be verified. Such tasks include the gathering of statistics or running visualizations on data collected from sensors. In general, any task meant for human interaction can be

handled in a GPOS as these are not time-critical due to the perception of the human senses. Thus, splitting the real-time part off the whole application leaves two separate parts where the one for controlling can be implemented using a specialized RTOS and the other for human intervention can be run in a GPOS.

As this part implemented in the GPOS still needs to access the data collected by the application running in the RTOS, the data needs to be shared. One approach to this is the consolidation of the two systems on the same hardware platform as that removes the need for transfer mechanisms across devices as depicted in Figure 1.1.

Such communication or visualization efforts can be achieved on standard PC hardware as their compatibility to a lot of other components allows usage in many possible combinations. Since the consolidation effort removes the need for separated components running the real-time system, the bill of materials can be reduced by using standard hardware. Furthermore, the PC hardware based on the Intel x86 processors is widely available, which makes the development of an RTOS on this platform cheaper than using specialized industrial components. Also, a manufacturer can warrant that replacement parts will be available in the whole lifespan of a deployed product.

1.3 Goals of this Thesis

The SLOTH system introduced by Hofer et al. [2] provides a small RTOS with low latencies by using the hardware for components usually implemented in software. The main concept in SLOTH is the use of the interrupt controller hardware for thread scheduling by modeling all threads as interrupt handlers. The SLOTH system has already proven to outperform existing implementations in previous publications [2, 3].

Implementations of SLOTH have already been accomplished on the Infineon TriCore and the ARM Cortex-M3 [2, 4]. While these existing implementations of SLOTH run standalone on a microcontroller platform, this thesis investigates how an RTOS implementing the SLOTH real-time scheduling can be integrated with Linux as a GPOS in order to run them both concurrently on the same PC hardware.

Existing approaches towards a real-time capable Linux need to be reviewed by examination of the implementations of these systems and to identify concepts to be reused for the implementation of a hybrid system using the SLOTH approach for interrupt-driven scheduling for the real-time part. An important factor in the analysis is the handling of interrupts, since latency—a major property of an RTOS—may not be affected by interrupt blocking in the Linux kernel. The real-time extension for Linux designed in this thesis implements the SLOTH approach of interrupt-driven scheduling, where the Linux running simultaneously is supposed to provide user interaction without influencing the real-time guarantees of the SLOTH system. To take full advantage of the fusion, the real-time tasks running in the RTOS need to communicate with processes in the GPOS in order to transfer information between the two systems; thus, an adequate mechanism needs to be invented as well. The resulting SLOTHFUL LINUX provides the functional-

ity of common Linux with an additionally RTOS core that uses the SLOTH concept for scheduling.

In order to compare the implementation of SLOTHFUL LINUX with another existing hybrid system, the Intel x86 was chosen as hardware platform as it is widely supported among available systems. At first, the standalone SLOTH system is ported to the bare-metal Intel x86 to review the available interrupt controller hardware. This proves the fitness of the platform and is also a competitor for evaluations against the combined implementation.

The following two sections provide necessary background information on both the Linux kernel and the SLOTH real-time system, which are combined into SLOTHFUL LINUX in this thesis.

1.4 The Linux Kernel

Since its advent about 20 years ago, Linux [5] has gained a wide user-base and is developed by a large community of programmers around the world supporting many different hardware components. As the source code of Linux is available under the terms of an open source license, everyone can extend the kernel code in numerous ways. Likewise, the availability of the source code makes it possible to understand its control flow paths in detail, which supports both development and verification of modifications to the Linux kernel.

Due to the success of Linux as a base for prior implementations of hybrid real-time systems (further detailed in Section 2.2), this thesis aims to integrate the SLOTH real-time kernel with Linux, in order to run both concurrently on the same computing hardware.

1.5 The Sloth Real-Time Kernel

The SLOTH concept proposes to implement scheduling and dispatching of different control flows in an operating system by using the interrupt controller hardware. This removes the usual distinction between threads and interrupts by implementing all control flows as interrupt handlers and thus, eliminates the need for a software scheduler completely. In this system, scheduling decisions for both synchronous and asynchronous events are made solely in the scope of the interrupt controller, which has a huge improvement on the performance of system services.

1.5.1 The Sloth Concept

The reference implementation of SLOTH is designed as a standalone event-driven real-time system implementing the OSEK¹ operating system specification [6]. An overview

¹OSEK is a German acronym for “Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug”, translates to “open systems and corresponding interfaces for automotive electronics”

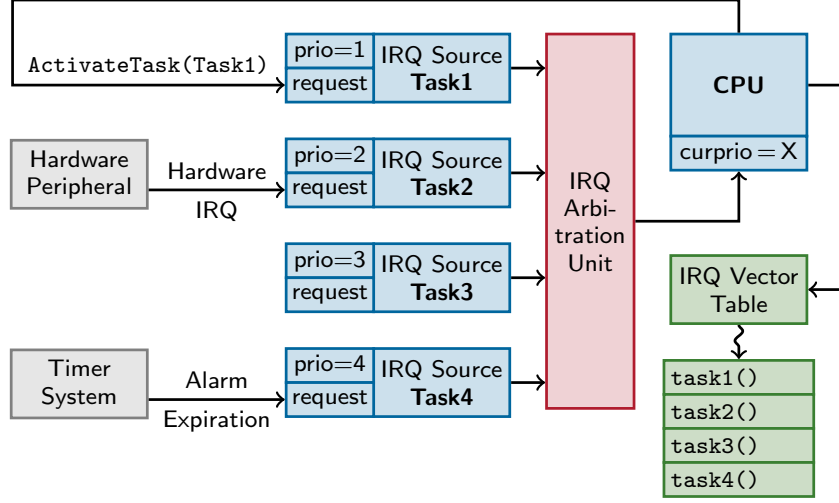


Figure 1.2: Design of the SLOTH system using interrupt handlers for implementation of different control flows.

of the concept can be seen in Figure 1.2. The system functionality includes control flow abstractions in form of *tasks*, where each of them is assigned to a separate interrupt source with a fixed priority. Therefore, the activation of such a task merely means setting the corresponding pending bit in the interrupt controller. This can be done synchronously by software with a system service, or asynchronously by a peripheral hardware device. The scheduling decision is then handled by the IRQ arbitration unit that decides which of the pending interrupts—with the attached *task* control flow—has the highest priority. If the current CPU priority is lower than this highest priority determined in the arbitration step, the interrupt controller signals an interrupt request to the CPU. Otherwise, if the current CPU priority is already higher, then no interrupt request will be signaled until the CPU priority is lowered again. When the CPU is interrupted by the interrupt controller, the corresponding task will be dispatched automatically by looking up its entry point in the vector table. The CPU priority does not always have to match the priority of the executing control flow as it can also be changed in critical sections for mutual exclusion. Such synchronization primitives are implemented as *resources* in OSEK.

The OSEK specification describes two types of tasks: *basic* and *extended*. The former have a strict run-to-completion property, while the latter are able to block and wait for an event. Although another publication [3] already enhanced the SLOTH concept for extended tasks, this thesis focuses on the implementation of basic tasks targeting a basic conformance class of OSEK. Also, common systems with a software-based scheduler differentiate between asynchronous triggered interrupt service routines (ISRs) and synchronous activated tasks. In SLOTH, all tasks are effectively implemented as inter-

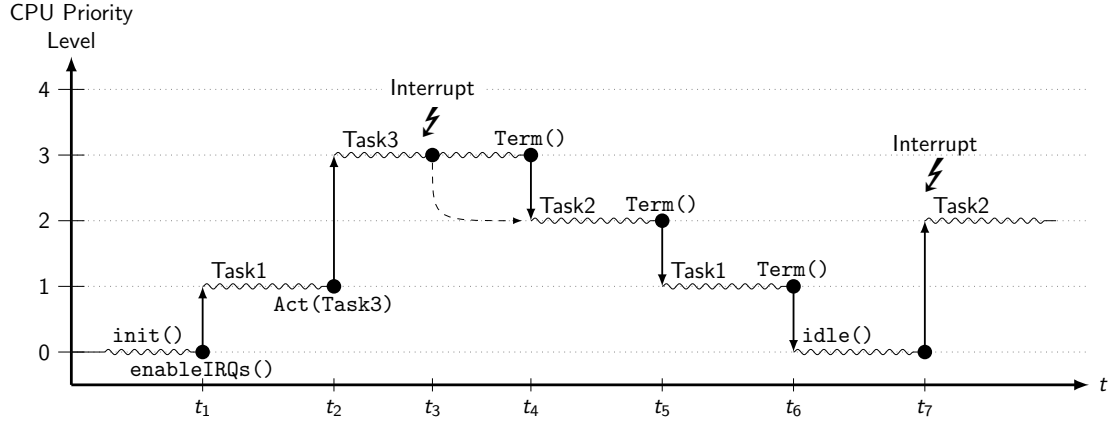


Figure 1.3: Example control flow in a SLOTH system. The execution of most system services leads to an implicit or explicit altering of the current CPU priority level, which subsequently leads to an automatic and correct scheduling and dispatching of the control flows by the hardware.

rupt handlers: both tasks and ISRs are allowed to be triggered asynchronously. Thus, in the following, the term *task* refers to control flows with run-to-completion properties, irrespective whether their activation is synchronous or asynchronous.

The task management of SLOTH is depicted in the example control flow in Figure 1.3. In this configuration, Task1, Task2, and Task3 have the priorities 1, 2, and 3, respectively. After initialization of the system, where the auto-started Task1 is set to pending, the system dispatches Task1 running with priority 1 at t_1 . At t_2 , Task1 activates Task3 using the `ActivateTask()` system service. As Task3 has the higher priority, it is immediately dispatched and preempts Task1; the current CPU priority level is raised to 3. At the time t_3 , an asynchronous task activation occurs by a hardware device that signals a request in the interrupt source of Task2. In this case, the dispatching of Task2 has to be delayed as the current execution priority is 3. The interrupt controller does not interrupt the CPU and the execution of Task3 continues. Only as Task3 terminates itself at t_4 using the `TerminateTask()` system service, the execution priority is lowered and the interrupt controller interrupts the CPU to dispatch Task2 which sets the execution priority to 2. Task2 terminates at t_5 , where the execution priority is lowered again so that the preempted Task1 continues running. At t_6 , where Task1 terminates itself, no other control flow is currently pending. Thus, the system is running an idle function at the lowest priority level waiting for further interrupts until Task2 is triggered again by an interrupt at t_7 . This time it is dispatched at once as no other control flow is running or pending.

1.5.2 Requirements on the Interrupt Controller

The SLOTH concept with utilization of the interrupt controller for scheduling leads to a very concise kernel implementation. The design of SLOTH is not tailored to specific hardware in order to be able to implement such a system on many modern platforms which fulfill the following requirements:

- The interrupt subsystem must provide different priority levels for each task in the system.
- The interrupt subsystem must support software-generated interrupts, allowing synchronous triggering of individual interrupts.

While existing implementations of SLOTH run standalone on embedded platforms, this thesis covers the design and implementation of an efficient hybrid real-time system based on the SLOTH concept, which runs concurrently with Linux on Intel x86 hardware. Section 3.2 details the interrupt controller available on this platform and also shows that it complies with these requirements for the implementation of the SLOTH concept.

1.6 Outline of this Thesis

This chapter gave an overview of the SLOTH concept and introduced the goal of this thesis: SLOTHFUL LINUX, a hybrid system running Linux and a SLOTH system concurrently on the same hardware platform. The following Chapter 2 gives an overview of both the requirements of an RTOS and outlines existing real-time extensions for the Linux kernel. Chapter 3 describes the design and implementation of SLOTH on the bare-metal Intel x86 platform, which will then be integrated with Linux to form SLOTHFUL LINUX in Chapter 4. Both implementations will be evaluated and discussed in Chapter 5; this chapter also compares SLOTHFUL LINUX with existing real-time extensions for the Linux kernel. Chapter 6 concludes this thesis with a summary of the results and an outlook for future work.

Chapter 2

Problem Analysis

This chapter gives an overview of the requirements on an RTOS emphasizing the kernel features related to the scheduling and dispatching of tasks. In the consolidation process of RTOS and GPOS it is important that these real-time properties are being respected. Thus, this chapter presents a review of current existing real-time extensions for the Linux kernel and their approaches, with a focus on interrupt virtualization.

2.1 Requirements on an RTOS

Operating systems are responsible for scheduling multiple tasks according to their priorities and for dispatching new tasks as they become active. Additionally, interrupts triggered by peripheral hardware components can signal events to the kernel that might need to be forwarded to the appropriate tasks.

In an RTOS, these operations need to be carried out in a predictable manner so that the program execution timing becomes deterministic. These timings can subsequently be used to assign a fixed *worst-case execution timing* to the jobs accomplished by the tasks in the RTOS. The response constraints of a real-time application can only be satisfied with reliable execution in the RTOS.

Therefore, task switches should take a fixed amount of time for saving and restoring context of the executing tasks. At best, system services for task switches only introduce a short latency into the application's execution, while scheduling decisions necessary due to task activation or termination need to be made promptly. To react fast on external events, an RTOS should ideally be preemptible at all times, which benefits a low interrupt latency. The scheduler should be able to preempt tasks in the system and give the CPU resources to other tasks according to their priorities by providing many opportunities for rescheduling. The responsiveness of the system can be improved by handling multiple levels of interrupts, where interrupt handlers can preempt not only running tasks but also other interrupt handlers. In order to preserve the integrity of data, applications must be able to synchronize concurrent access to resources by multiple tasks; this task

synchronization mechanism needs to be predictable and may not introduce unbounded priority inversion. Tasks of high priority may be delayed for an unlimited amount of time by other control flows of lower priority. [1, 7]

Overall, as operating systems are only used as an abstraction for applications to the hardware and do not fulfill a purpose of their own, especially real-time operating systems in the embedded market should have a low overhead and a high performance. By reducing the footprint of the operating system, more resources will be available for the application providing the actual functionality in a deployed system.

2.2 Existing Projects based on Linux

Linux has become quite popular over the years and due to its community-based development it gained support for various hardware components. However, the standard Linux kernel fails to provide strict timing guarantees and the responsiveness to hardware interrupts shows variances. Previous measurements of these properties have shown that the standard Linux kernel does not meet the requirements to be used as an RTOS [8, 9].

However, several approaches have already been implemented in order to improve the timing predictability of Linux. These different solutions can be classified into two categories:

- Modify Linux with patches to provide a kernel with real-time behavior.
- Move the real-time activity into a smaller real-time kernel that handles all interrupts and run the Linux kernel with virtualized interrupts as a low-priority task.

The following sections take examples for both categories and explain the different approaches in more detail.

2.2.1 The RT-Preempt Patch

For the first option, the RT-Preempt patch [10, 11] is a well-known example of modifications to the Linux kernel. This patch already poses a long and steady process along the development of Linux, as some features developed as part of this patch were also integrated into the standard Linux kernel. The most important part of the patch tries to make the full kernel code preemptible. This is achieved by allowing preemption of locking primitives synchronizing critical sections in the kernel. Additionally, it implements priority inheritance for kernel spinlocks and semaphores to avoid priority inversions. Even hardware interrupt handlers are made preemptible by wrapping them into a thread context. Overall, the patch includes a variety of features of interest for real-time capabilities within the Linux kernel.

However, despite the features added by this patch, the Linux kernel with the features of the RT-Preempt patch does not qualify for use in *hard* real-time systems. Measurements have shown that the real-time performance still includes a jitter, although it has

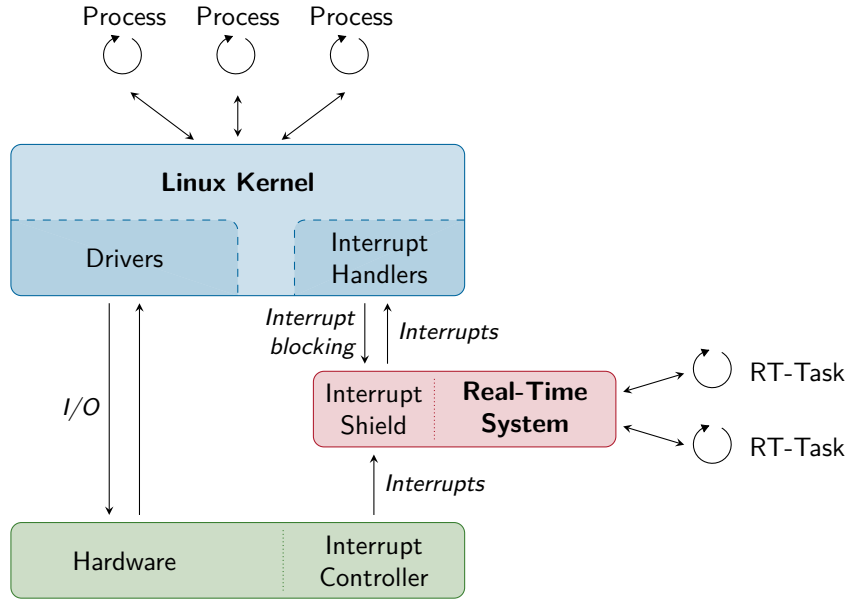


Figure 2.1: Interrupt virtualization adds an abstraction to the Linux kernel which removes direct access to the interrupt controller. This makes it possible to run Linux and an RTOS side-by-side on the same hardware.

been reduced a lot compared to the standard Linux kernel [12]. With this patch, there is also no way to make guarantees for a worst case execution time as the implementation is quite complex and thus, it is not possible to test all control paths a real-time application task might take. Nevertheless, the RT-Preempt patch at least manages to bring real-time support to the Linux kernel that can be used in a scenario with *soft* real-time requirements [12].

2.2.2 The I-Pipe Patch for Generic Interrupt Virtualization

The second approach uses *interrupt virtualization* with an abstraction layer between the Linux kernel and the interrupt controller hardware as shown in Figure 2.1. For this virtualization, all functionality regarding interrupt handling needs to be removed from the Linux kernel. Instructions for blocking interrupts on the hardware—assembly mnemonics `cli/sti` on Intel x86—could inflict unpredictable delays in the interrupt dispatching as they are used for synchronization throughout the Linux kernel code in drivers from many different authors.

The synchronization of code sections using hardware interrupt blocking needs to be replaced by *optimistic interrupt protection* [13], which allows the occurrence of interrupts during these sections but prevents the execution of interrupt handlers until the end of

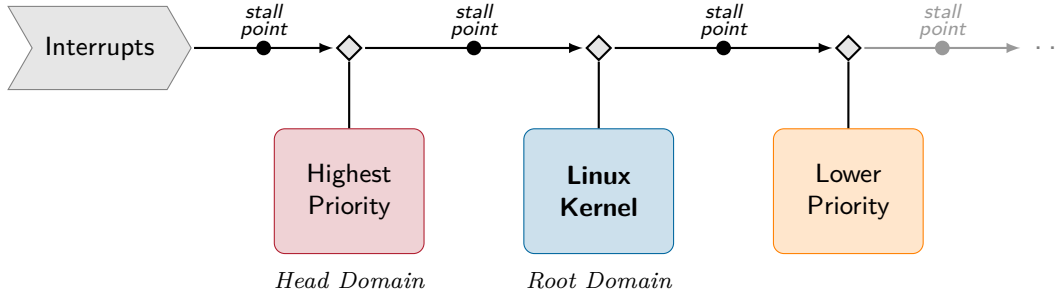


Figure 2.2: The I-Pipe patch arranges multiple domains in the form of a pipeline where each domain may only handle the events passed by the domains of higher priority. The position of the *root* domain containing the Linux kernel might be anywhere along the pipeline. Each domain implements optimistic interrupt protection and can be stalled to block the handling of interrupts in this domain without influencing the domains of higher priority.

the section. Therefore, although they actually can be interrupted, these sections still appear to be “uninterruptible” in the control flow.

To achieve this protection, replacement functions for the entry of such critical sections only mark interrupts as blocked in software. A small real-time kernel receives interrupts from the controller hardware and dispatches their handlers according to this current blocking state. If interrupts are allowed, their handlers can be dispatched immediately in the Linux kernel. Otherwise, they need to be delayed until the synchronized section will be left. At this point in the replacement code, a check is added that dispatches pending interrupt handlers as marked by the real-time kernel.

With these changes, interrupts are always handled by the real-time kernel and although Linux is unable to block interrupts, synchronization of interrupt handlers is still possible. In this system, the real-time kernel always takes precedence as it reacts on the interrupts and thus, the Linux kernel will only run when the real-time application is idle.

This technique was used to implement the first Linux based real-time operating systems [14, 15]; later on, the Adeos patch for Linux [16, 17, 18] aims for a generic way of interrupt virtualization. It is used as base for both RTAI [19] and Xenomai [20], which are two projects adding real-time capabilities to the Linux kernel.

By using the interrupt virtualization mechanism, Adeos arranges multiple domains with static priorities along the form of a pipeline as depicted in Figure 2.2. Interrupt events generated by the hardware flow from the highest to the lowest priority domains. The head of the pipeline is nearest to the hardware and has the highest priority. This interrupt pipeline for Linux as provided by Adeos is commonly referred to as the *I-Pipe patch* for Linux.

The domain stages in the interrupt pipeline can be *stalled*, which inhibits interrupt handling at this stage and only records their occurrence for handling later. The pending

interrupts are being recorded and replayed once the stage is *unstalled*. However, interrupts generated by the hardware are still fed to the domains of higher priority. Therefore, domains of lower priority will be preempted when interrupts occur that need to be handled in higher priority domains. A domain registers handlers for incoming interrupts and either passes interrupt events on to the next stage in the pipeline or ends handling at this stage. If no handler for a specific interrupt has been registered in a domain, it is automatically propagated down to the other domains in the pipeline.

The I-Pipe patch treats Linux as the *root* domain, which is the initial domain brought up at boot time. Afterwards, more domains can be added at arbitrary positions in the pipeline. For the implementation of a real-time system, the most interesting stage is at the head where interrupts are received first. Both RTAI and Xenomai insert a new domain for their real-time core that must have the highest priority at the head. Although the I-Pipe patch allows to install many domains along the pipeline, using two domains—one for the real-time and one for the general-purpose part—is the main use in these implementations.

2.2.3 Summary

The standard Linux kernel does not provide real-time capabilities; however, two different approaches towards providing real-time capabilities with the Linux kernel are available. Of these, the RT-Preempt patch only provides soft real-time performance. This shows that modifications to the Linux kernel can be quite complicated and the resulting behavior is influenced by many factors. However, support for hard real-time can be achieved by using the second approach of interrupt virtualization as for example provided by the I-Pipe patch against the Linux kernel.

2.3 Integration of Sloth into Linux

Interrupt virtualization adds an abstraction to the handling of interrupts in the standard Linux kernel and thus, a real-time extension can schedule and dispatch its tasks independently. This technique removes direct access to the interrupt controller hardware from the Linux kernel. The SLOTH system relies on utilization of the interrupt controller for scheduling and dispatching and requires direct access to the hardware for that. Thus, the interrupt virtualization mechanism perfectly fits for the hybrid SLOTHFUL LINUX which will be detailed in Chapter 4.

2.4 Summary

The main requirement on an RTOS is the deterministic execution to guarantee predictable and strict timing of an application. Of the existing real-time extensions to the

Linux kernel, only those applying interrupt virtualization are able to provide hard real-time guarantees. This mechanism provided as a generic solution by the I-Pipe patch for the Linux kernel can be reused for the integration of SLOTHFUL LINUX.

Chapter 3

Design and Implementation of Sloth on the Intel x86

The SLOTH concept was designed for using the interrupt controller for scheduling purposes in an event-driven embedded system. Before implementing a combined RTOS and GPOS, SLOTH was first ported to the bare-metal Intel x86 platform to see how the available hardware components comply with the requirements defined in Section 1.5.2. After providing an overview of the design of the SLOTH kernel in the first section, this chapter describes the Intel x86 interrupt subsystem and its utilization for the implementation of SLOTH tasks and resources.

3.1 Sloth Design Overview

SLOTH implements the OSEK operating system specification as explained in Section 1.5. It supports tasks as control flow abstractions for which a fixed priority needs to be defined in an application specific configuration before compilation. A system generation step analyzes the application configuration in order to tailor the system to the needs of the application and assigns each task an interrupt source according to its priority. Afterwards it generates corresponding code that only contains features selected in the configuration. Thus, the resulting binary can be reduced in size and will be as small as possible. Table 3.1 gives an overview of the available system services for tasks and resources.

Multiple tasks can be competing for execution on a CPU, although only one of them can be running at the same time. Usually, a software scheduler is responsible to determine the execution order of tasks. However, in SLOTH there is no such software component and the interrupt controller hardware determines the execution sequence instead. These tasks can be activated either synchronously by the `ActivateTask()` system service or asynchronously by a hardware device. As tasks are already implemented as interrupt

System Service	Functionality
<code>ActivateTask(TaskID)</code>	schedules a task for execution
<code>TerminateTask()</code>	ends the current control flow
<code>ChainTask(TaskID)</code>	ends the current control flow and schedules another task ready for execution thereafter
<code>GetResource(ResID)</code>	enters a critical section
<code>ReleaseResource(ResID)</code>	leaves a critical section

Table 3.1: The system services provided by SLOTH for the management of tasks and resources according to the OSEK OS specification.

handlers in SLOTH, there is no need for a separate abstraction for interrupt service routines.

Furthermore, *resources* can be acquired with the system services `GetResource()` and `ReleaseResource()` for mutual exclusion in critical sections using the OSEK priority ceiling protocol. While a resource is held, the current execution priority is raised to the highest priority of all tasks accessing the same resource, which prevents preemption during the synchronized section.

3.2 The Intel x86 Interrupt Subsystem

The Intel x86 architecture is one of the most common hardware platforms with an emphasis on backward compatibility. It implements a CISC instruction set with data stored in little-endian byte order. In its history, the design was subject to many additions, so that modern x86 CPUs are superscalar, featuring pipelining and out-of-order execution.

However, most notable for this work, the original external Programmable Interrupt Controller (PIC) was replaced by Intel with the Advanced Programmable Interrupt Controller (APIC) to support multiple processors in the same computer. This new hardware is specified in two parts, the *local APIC* as an inherent part of modern x86 CPUs and the *I/O APIC* as an external gateway on bus interfaces as shown in Figure 3.1. This new APIC architecture removed the need for interrupt vector sharing by extending the previous 16 available vectors of the cascaded PIC mode to 224 usable vectors in the local APIC. While originally intended for multiple processors, the local APIC and I/O APIC can also be used in uni-processor systems. Programming of the local APIC is achieved by reading and writing memory-mapped registers [21, Ch. 3A].

The I/O APIC is part of the chipset and has multiple input pins on which it receives interrupts from devices. It is responsible to redirect these interrupts to one or more local APICs connected over the system bus by raising the corresponding vectors as programmed in the redirect table of the I/O APIC. Additionally, the local APICs are able to send inter-processor interrupts (IPIs) for any vector to one or more of the connected

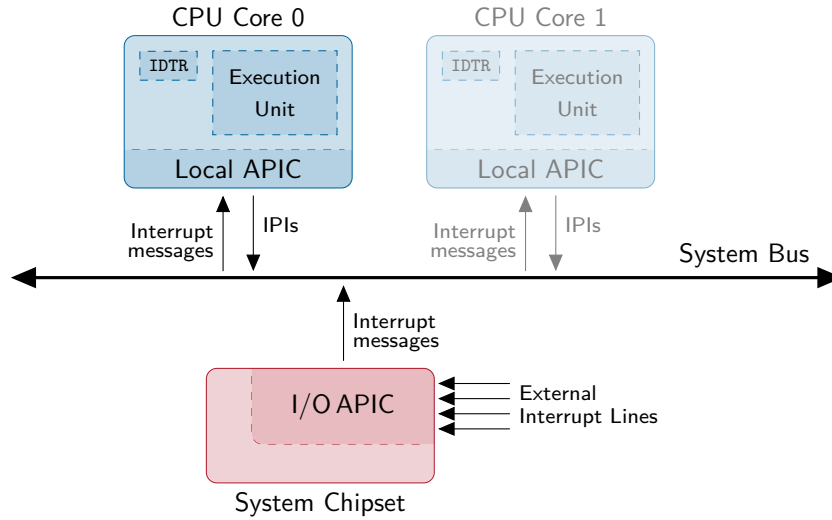
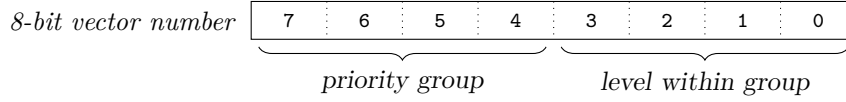


Figure 3.1: The x86 interrupt controller architecture with local APIC and I/O APIC which is meant to support multiple processors, but can also be used in uni-processor systems. The local APIC receives interrupt messages on the system bus and can also send inter-processor interrupts (IPIs) to other processors or to itself—in which case they are handled internally and are not put on the bus.

local APICs. These IPIs can also be configured for delivery to the sender itself. Such a self-IPI can be used for synchronous interrupt triggering by software. IPIs can be issued by writing the Interrupt Command Register (ICR) with the vector number and destination. The local APIC as part of the CPU asserts interrupts as signaled on the system bus and interrupts the execution unit in order to dispatch interrupt handlers using an Interrupt Descriptor Table (IDT) in memory specified by the internal IDTR register. This is basically a vector table in form of a list of addresses to the corresponding handlers and also contains additional flags specifying privilege levels. Of these 256 entries, the first 32 vectors are reserved by Intel for exceptions, leaving 224 vectors to be used for signals from external devices or issuing software-generated interrupts.

Of these vectors delivered through the local APIC each has an implied priority based on the vector number. This priority will be used to determine when specific interrupts can be serviced by the CPU. The interrupt priorities are organized in groups of 16, which means there are 14 different preemption priorities for the 224 usable vectors. Technically, the higher 4 bits of the 8-bit vector number resolves to the priority group and the lower 4 bits specify the priority level inside that group as illustrated in Figure 3.2.

The first two priority groups fall into the range of the reserved vectors in use for system exceptions. For these, the prioritizing does not apply since exceptions need to be handled in a synchronous manner. For the rest of the priority groups applies: the higher



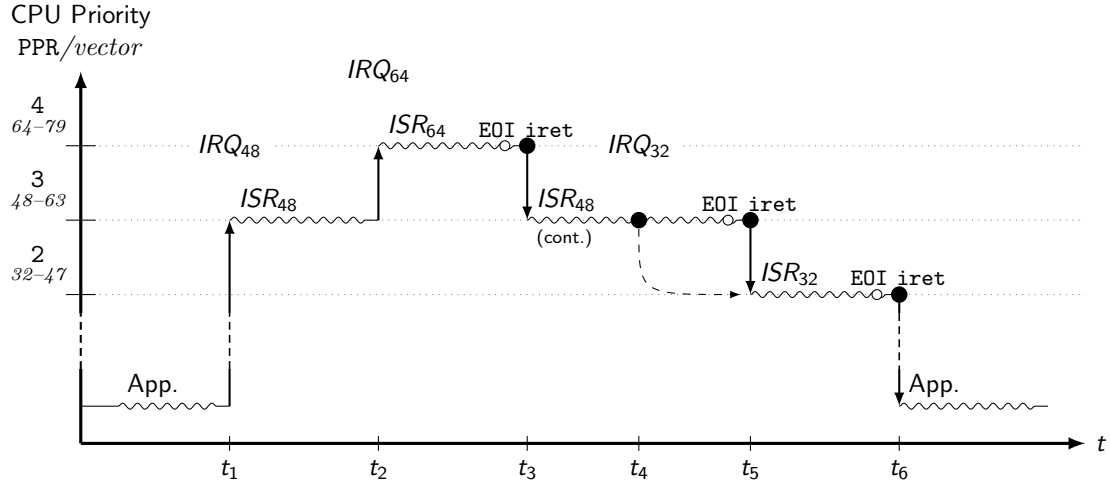


Figure 3.3: Example of nesting of interrupt handlers with different priorities. The interrupt handlers are being dispatched according to their priority groups, where only an interrupt with a higher priority group can preempt a running interrupt handler with a lower priority group.

Whenever an interrupt of a higher priority than the current processor priority is pending and interrupts are enabled in the processor flags, the APIC signals this interrupt immediately to the CPU without waiting for a write to `E0I`. This means that interrupt handlers can be interrupted, effectively leading to a nesting of interrupt handlers as demonstrated in Figure 3.3.

In this example, a normal application is executing when at t_1 , the interrupt for the interrupt vector 48 is triggered. This leads to an immediate dispatch of the corresponding `ISR48` and the CPU priority is raised to the priority group of the interrupt vector. Assume that each of the interrupt handlers in this figure allow interrupts again as their first action. At t_2 , the interrupt for the vector 64 is set to pending. Although another ISR is currently running, `ISR64` is dispatched and preempts the currently running `ISR48` as it has a higher priority group. The execution of `ISR48` continues after `ISR64` returns from the interrupt handler by writing to the `E0I` and executing the `iret` instruction. In contrast, as the vector 32 is requested at t_4 , the `ISR32` is deferred until the `ISR48` with their higher priority group has finished and signals the end of the interrupt and returns at t_5 . Finally, as `ISR32` ends the interrupt at t_6 , the interrupted application can run again.

3.3 Utilization of the Local APIC

The local APIC fulfills the requirements on an interrupt controller in order to implement SLOTH on the hardware platform as listed in Section 1.5.2. Each SLOTH task can be

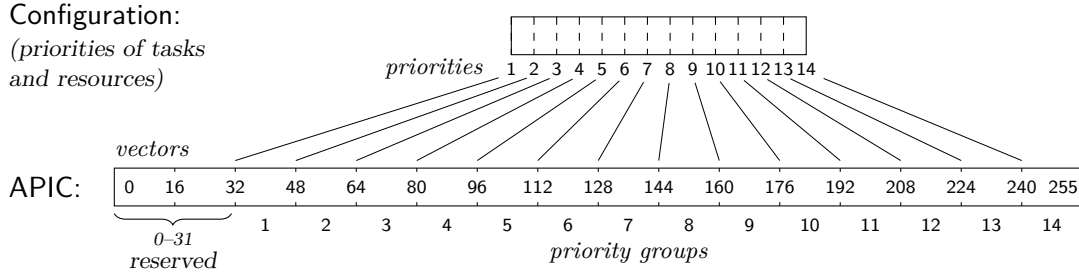


Figure 3.4: The mapping of the priority spaces from the specifications in the application configuration to the physical priorities corresponding to the APIC vector numbers.

assigned to one of the available interrupts vectors with inherent priorities as provided by the local APIC. It is also possible to trigger an arbitrary vector in a synchronous manner from software with a self-IPI. With the mandatory functionality in place, the local APIC can be used to implement tasks and resources on the Intel x86 architecture.

3.3.1 Tasks

In the SLOTH concept, control flows are organized into tasks that are scheduled according to their priority by the interrupt controller. On the Intel x86, this is implemented using the local APIC by assigning tasks to interrupts. As the interrupt vectors have an inherent priority derived from their number, task priorities need to be mapped from their logical specification in the configuration to the physical equivalents in the APIC. Since the priorities in the APIC are organized in groups of 16 and preemption will only occur when the priority group is higher, the vector numbers for the SLOTH system need to be spread through the different groups. This means the system can use up to 14 different priorities as shown in Figure 3.4.

Tasks can be activated synchronously with the system service `ActivateTask()`. This is implemented by issuing a self-IPI with the local APIC, which is only a single write instruction to the ICR. Tasks may also be triggered asynchronously by an external device when the I/O APIC has been programmed accordingly to raise the corresponding vector; both ways mark the vector as pending. If the current CPU priority is less than the priority group of the marked interrupt, this leads to an interruption of the execution unit.

To dispatch an interrupt handler, the CPU first stores the non-volatile register context of the current control flow on the stack and then looks up the entry address in the vector table. In consequence, execution continues at this address, where a small wrapper function saves the remaining context. Subsequently, the wrapper jumps to a generated function prologue which sets the current task variable to this activated task. The prologue also enables interrupts at this point again, which allows preemption of this control flow by tasks with a higher priority level.

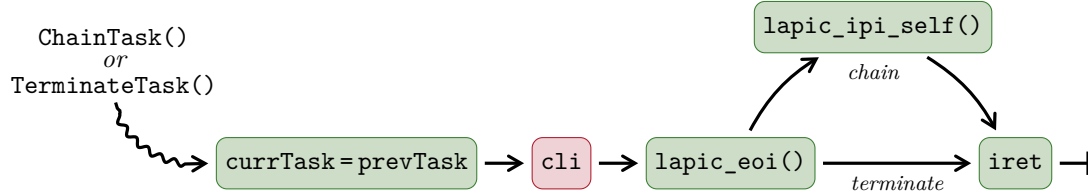


Figure 3.5: Task termination needs to be protected as interrupts could occur between signaling the end-of-interrupt to the APIC and completing the interrupt handler with an `iret`. Chaining another task additionally requires sending a self-IPI to trigger the corresponding interrupt.

The run-to-completion property of tasks matches the semantics of interrupt handlers perfectly, allowing to nest task execution in the same way as interrupt handlers as shown above in Figure 3.3.

A currently running task terminates itself with the system service `TerminateTask()`, which ends the execution of the calling control flow. A task must indicate the termination with either `TerminateTask()` or `ChainTask()` (see below). The previous context needs to be restored, including dropping the CPU priority to the previous level. The latter is achieved automatically by writing to the EOI register, which reverts the current priority in the PPR to either the priority group of the previously interrupted control flow or the value of the TPR, whichever is higher. However, since the end-of-interrupt in the APIC lowers the priority before the current interrupt handler was actually completed with the `iret` instruction, the still-running control flow could again be interrupted immediately by the next pending interrupt. Therefore, interrupts between signaling the end-of-interrupt to the APIC and the `iret` instructions must be prevented by disabling all interrupts beforehand with an `cli` instruction. As the `iret` instruction restores the previous state of the EFLAGS system register from the stack, it automatically enables interrupts again.

Additionally, the system service `ChainTask()` is available to both terminate the current control flow and schedule another task for execution in a single operation. If the chained task has a higher priority than the calling task, it is essential that the newly scheduled task does not start execution until the calling task has been terminated. This means that synchronization is required for the proper chaining of tasks by suppressing the activation request until the task termination has been completed. In SLOTH on Intel x86, this is achieved by first blocking all interrupts and signaling the end-of-interrupt in the same way as in `TerminateTask()`; only then the task to be chained will be activated by sending the corresponding self-IPI. Finally, the `iret` instruction implicitly restores the interrupt-enable flag; thus, the interrupt controller will decide which pending interrupt will be dispatched next according to the priorities. The sequences for task termination and task chaining are depicted in Figure 3.5.

3.3.2 Resource Management

Resources are used for mutual exclusion between different tasks. They allow to coordinate the access to shared components such as memory ranges or hardware peripherals. OSEK specifies a priority ceiling protocol [6, p. 29] that mandates that a resource can only be occupied by one task at a time, avoids any deadlocks by the use of resources and prohibits unbounded priority inversion. A resource acquisition raises the current execution priority to the ceiling priority of the resource—the highest priority level of all tasks allowed to access this resource. When released, the execution priority level is lowered to the task’s previous priority.

The ceiling priority value is statically computed at system generation for each resource in an application. These priorities need to be mapped in the same way as task priorities: from logical values in the configuration to physical equivalents in the APIC. The system service `GetResource()` acquires a resource to enter a critical section in a task. When a task needs multiple resources, the calls to `GetResource()` and `ReleaseResource()` need to be strictly nested, such that the execution priority is raised and lowered in a stack-based manner.

In the SLOTH implementation for Intel x86, resources are implemented using the lower priority threshold provided by the TPR in the APIC. Using this register, the current execution priority can be elevated to a higher level, which prevents interrupt handlers of a lower priority from dispatching. The current priority can only be raised over the priority of the currently executing interrupt handler. Initially, the TPR is set to value 0, which does not influence the arbitration and thus, only the priorities of the interrupts are taken into account.

On acquiring a resource with `GetResource()`, the current priority in the TPR is pushed onto a stack to allow nesting of resource usage. After saving the previous value, the ceiling priority of the acquired resource is written to the TPR. The system service `ReleaseResource()` reverses the operation by restoring the previous execution priority from the stack into the TPR. This leaves the critical section enclosed by the two system services.

3.4 Summary

The SLOTH concept proposes interrupt-driven scheduling by use of the interrupt controller hardware to eliminate the need for a software-based scheduler. The SLOTH implementation for the Intel x86 supports tasks and resources by utilizing the functionality of the local APIC, which is part of the CPU.

Chapter 4

Design and Implementation of Slothful Linux

In the last chapter on the design and implementation of SLOTH on the bare-metal Intel x86 the hardware platform already proved to meet the requirements of the SLOTH concept. This chapter covers the design and implementation of the hybrid SLOTHFUL LINUX, which is capable of running the interrupt-driven SLOTH real-time system along with Linux on a single computer.

4.1 Design of Slothful Linux

When adding SLOTH as a real-time extension to the Linux kernel, the implementation should offer the same functionality as the standalone SLOTH system as presented in Chapter 3. This means that the system should support *tasks* that are scheduled and dispatched by the interrupt controller and *resources* to limit the access to shared components. Additionally, as SLOTHFUL LINUX runs an RTOS in combination with Linux on the same hardware, there also needs to be a channel to transfer data between the two systems. Therefore, SLOTHFUL LINUX implements *pipes* to transfer data from the real-time system to processes running in the general-purpose OS. In an application running on SLOTHFUL LINUX, these could be sensor values used for visualizations drawn by an application running in Linux.

4.1.1 Tasks and Resources

The SLOTH concept takes advantage of the interrupt controller for scheduling and dispatching of tasks. The interrupt virtualization technique described in Section 2.2.2 introduces an abstraction layer which removes direct access to the interrupt controller hardware from the Linux kernel. This is a perfect match for the implementation of a hybrid system running SLOTH concurrently with Linux on the same hardware. The SLOTH

real-time kernel will have direct access to the interrupt controller, using its priority arbitration hardware for scheduling decisions. The interrupts intended for the real-time system can dispatch tasks directly in the SLOTH kernel, while interrupts from peripherals will be passed on to the Linux kernel.

Thus, the I-Pipe patch is used as a basis for the implementation of SLOTHFUL LINUX. Applying this patch to the Linux kernel already removes sections blocking interrupt handling, which allows interrupts—and especially the associated SLOTH tasks—to preempt the Linux kernel at any time. In the interrupt pipeline, the SLOTH real-time core will be inserted as a new domain at the head priority. The Linux domain in the pipeline will be stalled when required, whereas, the SLOTH domain will never be stalled. In a SLOTH system, the priorities are managed directly by the interrupt controller and interrupts will not be signaled by the hardware unless preemption of the current control flow is actually allowed with respect to the current execution priority of the processor.

Therefore, the SLOTH domain does not take advantage of the interrupt pipeline—rather, the additional code in the interrupt dispatching could add unnecessary cycles to the interrupt latency. Therefore, interrupts intended for the SLOTH system should not flow through the interrupt pipeline, but instead should be delivered right to the SLOTH system. Further changes to the Linux kernel will ensure that SLOTH will have exclusive access to the appropriate interrupt vectors and its handlers.

4.1.2 Pipes

In order to take advantage of the consolidation of an RTOS and a GPOS on the same hardware, data needs to be shared between the two. For this purpose, SLOTHFUL LINUX uses persistent pipes to implement a unidirectional communication channel from the real-time core to user space processes in Linux. Of course, this is an extension to the existing SLOTH system and there is no specification in OSEK for this kind of object.

Data transfers from the real-time part to a user space process need to occur across the Linux kernel as the real-time core does not know the user space processes of Linux and—by the UNIX philosophy—the user space process needs a file or device to read from. Therefore, the pipes are implemented as character devices in Linux, which can be read from user space processes and written by real-time SLOTH tasks.

4.1.3 Real-Time Applications as Linux Kernel Modules

The implementation of SLOTHFUL LINUX is supposed to be flexible and to support different real-time applications. In the same way the standalone SLOTH system can be tailored to the needs of an application, the real-time core should only contain the relevant parts as required for the application at hand. Therefore, SLOTHFUL LINUX uses kernel modules to implement real-time applications. This makes it possible to load and unload a real-time application after booting the computer, which also aids development as no reboot is required to make changes to the real-time core. Each configured application

will be compiled into a single loadable Linux kernel module that contains all required information for the deployment of the real-time application.

4.1.4 Summary

SLOTHFUL LINUX applies the interrupt virtualization offered by the I-Pipe patch to deny the Linux kernel the direct access to the interrupt controller. The real-time core schedules and dispatches tasks utilizing the interrupt controller, without being influenced by the Linux kernel. The real-time applications bundled into kernel modules can be loaded at runtime, while pipes using character devices offer a communication channel to user space processes in Linux.

4.2 Implementation of Slothful Linux

The implementation of SLOTHFUL LINUX supports running an interrupt-driven real-time core along with Linux on the same hardware platform. In its current form, it is based on the Linux kernel version 2.6.38.8, which is the latest supported version of the I-Pipe patch for Intel x86 at time of this writing. Due to the use of hardware components systems implementations of the SLOTH system are limited to the hardware platforms they have been designed for. In this case, SLOTHFUL LINUX makes direct use of the local APIC available on the Intel x86 platform. The implementation targets a single processor system, however, the design is not limited to this and could easily be adapted to multi-core systems.

4.2.1 Modifications to the Standard Linux Kernel

SLOTHFUL LINUX uses interrupt virtualization to remove the direct access to the interrupt controller from the Linux kernel. A common implementation of this approach is available for Linux with the I-Pipe patch as part of the Adeos project. However, further modifications to the standard Linux kernel are necessary to implement interrupt-driven scheduling of real-time tasks according to the SLOTH concept.

Using the I-Pipe Patch for Interrupt Virtualization

The I-Pipe patch is a modification to the Linux kernel that implements interrupt virtualization by optimistic interrupt protection for multiple domains arranged along an interrupt pipeline, as described in Section 2.2.2.

As a short reiteration, with the I-Pipe patch interrupts will not be handled directly by Linux, but an abstraction layer between the kernel and the interrupt controller hardware is added. As the direct access to the hardware has been removed, the Linux kernel no longer blocks interrupts directly, but only records the current blocking state in a variable. The new abstraction layer then decides if interrupt handlers can be dispatched

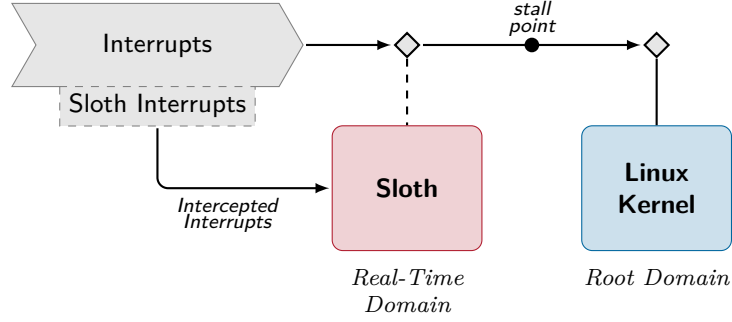


Figure 4.1: The pipeline architecture as used in SLOTHFUL LINUX where the SLOTH domain takes precedence over the domain of the Linux kernel. The Linux domain has a stalling point in front, while the SLOTH real-time core receives the intercepted interrupts directly avoiding the interrupt pipeline.

immediately or need to be delayed until the variable is reset. This allows to preempt the Linux kernel at any time, which is required to run an RTOS concurrently at a higher priority.

In order to implement the interrupt pipeline of the I-Pipe patch, the default interrupt handler of the Linux kernel is replaced with a specific handler for the interrupt pipeline. Each interrupt signaled by the hardware could be awaited by any of the domains in the pipeline. Therefore, each of the domains needs to be checked if they have installed an handler for the incoming interrupt request. This happens in strict order of the pipeline, which stops as soon as a domain accepting this interrupt is found and the corresponding handler is dispatched. If a domain is currently stalled, no interrupt handler can be dispatched here and the interrupt is recorded as pending to be handled later.

For this case of using the I-Pipe patch for the implementation of a hybrid system running both an RTOS core and Linux on the same hardware, there is only need for two separate domains. Of course, the real-time domain needs a higher priority than the Linux kernel. In the specific case of the SLOTH concept, which uses the interrupt controller for the scheduling and dispatching of tasks, the real-time domain needs to be the head domain of the pipeline. At this position, the SLOTH domain has direct access to the interrupt controller and can use it to its full extent. Figure 4.1 shows the interrupt pipeline used by SLOTHFUL LINUX.

Intercepting Interrupts for Sloth

In a SLOTH system, the interrupt subsystem of the hardware platform is in control of the scheduling and dispatching of tasks. Each task is mapped to an interrupt with the appropriate priority that is used by the interrupt controller for arbitration between multiple control flows waiting to be executed. Therefore, the pipeline added by the I-Pipe

patch will only be used for the optimistic interrupt protection within the Linux kernel and not for the SLOTH real-time core.

As latency is a major property of real-time systems, going through the interrupt handlers of the I-Pipe patch walking along the pipeline would add unnecessary cycles to the latency of the SLOTH system. Hence, SLOTH installs its own interrupt handler for the subset of the available interrupt vectors used for the corresponding tasks.

In the standard Linux kernel, all interrupt vectors of the local APIC in the interrupt descriptor table (IDT) point to the single dispatcher function `handle_irq` that checks for registered interrupt handlers and dispatches them. As this single dispatcher function needs to know which interrupt was actually triggered, a trampoline function table is added in between that pushes the interrupt number onto the stack and then jumps to the actual handler function `handle_irq`. This table is actually packed in groups of seven handlers into a single 32-byte sized chunk, as that fits into a single cache line on modern Intel x86 processors. This alignment optimization means that six of the interrupts in this chunk will have to take another jump to the handling function.

For SLOTHFUL LINUX, the interrupt handler addresses in the IDT that usually point to the trampoline table are overwritten with addresses to its own trampoline table `sloth_irq_entries`, which points to the handler function `handle_sloth_irq`. In contrast to the original handler table, it is not packed into smaller chunks with short jumps, but contains the full address for each entry as SLOTH needs to treat each interrupt the same. Otherwise the dispatching of a task could take a different amount of cycles based on its vector number, which is contrary to the realization of the real-time properties.

As the vectors registered here are directly handled in the SLOTH real-time core and can neither reach the I-Pipe dispatcher nor the Linux kernel handler, they are marked as used in the vector allocator. This removes the vectors out of control from the Linux kernel and ensures they will never be assigned to an interrupt and thus, they are not expected to be ever delivered to Linux.

For the design goal of encapsulating each real-time application into a self-contained kernel module, the actual addresses of the task functions are not known at compile time. The dynamic loading of modules assigns memory for the text segments at runtime, placing the functions at unknown locations. Thus, `handle_sloth_irq` takes these addresses from a dispatcher table that is part of the kernel module.

Kernel Module Loading

Real-time applications for use in SLOTHFUL LINUX are compiled into single kernel modules as the Linux kernel module interface already provides the necessary functionality for loading dynamic code and data sections.

Each kernel module contains a small initialization function that registers a new domain in the interrupt pipeline to be used for the SLOTH real-time core. The module also brings the task functions and a dispatcher table that assigns these tasks to the corresponding interrupt vectors with their implied priority. In the initialization function,

the table is registered with the patched interrupt handler in the Linux kernel and is subsequently used for the dispatching of tasks in the real-time system. Furthermore, the initialization also registers the required character devices for the pipes used in the real-time application.

The whole initialization sequence is synchronized by blocking all interrupts to avoid the handling of interrupts until the mandatory setup of the dispatcher table is complete. At all other occasions, the SLOTH real-time core will only block interrupts for synchronization purposes where required for a short and bounded time. In the same way as the kernel modules can be loaded, they can also be removed from the Linux kernel, where an exit function unloads the dispatcher table, removes the SLOTH domain from the interrupt pipeline and cleans the character devices for the pipes. Thus, the changes made at module initialization can be fully reversed to the original state.

Summary of the Modifications to the Linux Kernel

Besides applying the I-Pipe patch, only slight additional modifications to the Linux kernel are required for the SLOTH real-time core. In the interrupt handling, some vectors are intercepted by a custom handler function to dispatch the corresponding real-time tasks. The interrupt handler uses a dispatcher table which is registered during the load of the real-time application that is compiled into a Linux kernel module.

4.2.2 Tasks

Due to interrupt virtualization, the Linux kernel no longer has direct access to the interrupt controller; thus, the SLOTH real-time core can use it to its full extent for the scheduling and dispatching of tasks. In SLOTHFUL LINUX, each real-time task is assigned to an interrupt vector of the local APIC in the same way as on the bare-metal Intel x86. Task priorities need to be mapped from their logical value in the configuration to a vector with an inherent priority derived from its number. As described in the previous section, the Linux kernel has been patched to remove some interrupt vectors out of its control. The SLOTH real-time core intercepts these interrupts to dispatch the corresponding tasks.

The Linux kernel uses the interrupt vectors of the local APIC as depicted in Table 4.1. The interrupts in the range 129 to 223 are used for the SLOTH system, which means—due to the priority organization of the vectors in the local APIC in groups of 16—there can be up to 5 different tasks in a real-time application for SLOTHFUL LINUX as can be seen in the smaller overview shown as part of Table 4.1. The remaining interrupt vectors have been left to the Linux kernel for use with peripheral devices. However, this partition has been chosen arbitrarily and could be extended to support more tasks in the SLOTH real-time core when removing more vectors from the Linux kernel. Special care has to be taken not to change the vector 128, which is traditionally used as the system call gate in Linux.

Vector	Function	
0–19	NMI and exceptions	
20–31	Reserved by Intel	
32–127	External interrupts	
128	System call gate	
129–223	<div> <div> { External interrupts in standard Linux Real-time tasks in SLOTHFUL LINUX </div> <div> } </div> </div>	<div> SLOTHFUL LINUX: </div> <div> 144 Task 1 145–159 (<i>unused</i>) 160 Task 2 161–175 (<i>unused</i>) 176 Task 3 177–191 (<i>unused</i>) 192 Task 4 193–207 (<i>unused</i>) 208 Task 5 209–223 (<i>unused</i>) </div>
224–238	Special system interrupts	
239	Local APIC timer interrupt	
240	Local APIC thermal interrupt	
241–253	IPIs for SMP systems	
254	Local APIC error interrupt	
255	Local APIC spurious interrupt	

Table 4.1: The interrupt vectors of the local APIC as allocated by the Linux kernel. SLOTHFUL LINUX intercepts some of the vectors meant for external devices in order to implement interrupt-driven scheduling of real-time tasks.

The interrupt dispatching of the local APIC occurs according to the inherent priorities of the vectors. As the vectors reserved for the SLOTH system have a higher priority than the vectors left for external devices controlled by the Linux kernel, the real-time system can always preempt the interrupt handlers of the general-purpose system. However, some vectors still have a higher priority than the SLOTH core as these are mostly error conditions that need to be handled by the Linux kernel; for example, one vector of the special system interrupts is used for machine check events or the thermal event interrupt where both indicate possible hardware faults. The Linux kernel configuration used for the deployment of SLOTHFUL LINUX will ensure that most of these vectors will never be triggered. The evaluation setup described in the following Chapter 5 will give some more detail on this topic.

In the SLOTH real-time core, the same system services for tasks as in the standalone system can be used. Tasks are activated synchronously with the `ActivateTask()` system service, which issues a self-IPI for the local APIC. External devices may trigger tasks as well when the I/O APIC has been programmed to trigger this vector for the corresponding interrupt. Of course, the signaling of pending vectors in the local APIC is the same as on the standalone system: the CPU will only be interrupted if the current CPU priority is less than the priority group of the pending interrupt.

The interrupt dispatch goes through the IDT and the `handle_sloth_irq` function as described in the previous section. As in the bare-metal implementation, interrupts are automatically disabled at the entry of an interrupt routine. This SLOTH specific handler first switches to the SLOTH real-time domain in the interrupt pipeline and stores which

domain was preempted. This switch is arranged by setting a global state variable of the interrupt pipeline, which is guarded by a few sanity checks. Although the interrupt could also have occurred while the CPU was already executing code in the real-time domain, this domain switch is always performed in the same way to ensure the same latency is induced for each interrupt handling. Switching domains at this point is necessary so that any exception raised in the SLOTH domain can be handled there, as otherwise it would escalate to a problem in the Linux kernel, which would not know how to handle this as from the perspective of Linux, no interrupt occurred at all. Thus, it would assume the currently control flow marked as running caused this exception—which could be within the kernel or any user space process. At this switch, the root domain—the Linux kernel itself—is marked as stalled to inhibit handling of interrupts. Afterwards, the prologue of the task function provided by the real-time application is called. This prologue records this task as the currently running task and subsequently enables interrupts again.

The task function itself runs with the same privileges as every interrupt handler in the Linux kernel. The dispatching of real-time tasks in SLOTHFUL LINUX does not take any special measures in this regard. The task dispatcher does not reconfigure the MMU or switch to another privilege level in order to reduce latency. Therefore, the programming of real-time applications should be carried out carefully as the current implementation does not provide memory protection.

When the task function has finished and terminates itself with the `TerminateTask()` system service, the end of the interrupt handling is signaled to the local APIC in the same way as in the bare-metal Intel x86 implementation. Afterwards, the context of the preempted domain is restored and at the completion of the interrupt handling with the `iret` instruction, interrupts are automatically enabled again. Thereafter, if more interrupts are pending, the next highest vector number will be dispatched. The implementation of `ChainTask()` follows the same approach, except it additionally triggers of the chained task by sending a self-IPI.

4.2.3 Resources

The real-time core running in SLOTHFUL LINUX manages resources in the same way as the bare-metal system implementing the OSEK priority ceiling protocol. On resource acquisition with `GetResource()`, the current CPU priority is raised by writing the corresponding ceiling priority to the TPR of the local APIC. This ceiling priority for each resource is statically computed in the system generation step according to the configuration.

4.2.4 Pipes

Pipes represent a unidirectional communication channel between the SLOTH real-time core and processes controlled by the Linux kernel. They allow to transfer data from the real-time system to the general-purpose domain in order to take advantage of the

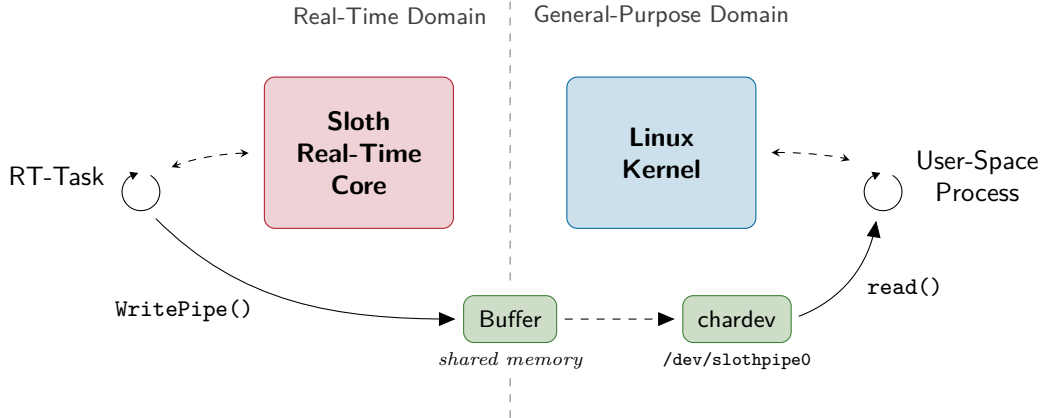


Figure 4.2: In SLOTHFUL LINUX, a *pipe* represents a unidirectional communication channel between the real-time system core and processes running in the general-purpose operating system.

consolidation of the two systems as depicted in Figure 4.2. A user space process running in the Linux kernel cannot interact with the real-time core directly, as the tasks running there are not known to Linux. Thus, data transfers must occur across the Linux kernel and need to be exposed to user space processes through an interface. As the real-time core and Linux share the same address space, the pipe implementation uses a buffer in shared memory to which the real-time core can output its data. The data gathered in this buffer can then be read through a character device by user space processes.

Character devices provided by the Linux kernel appear in the file system as if they were normal files; however, they act as an interface to a device driver—in this case, the SLOTH pipes. Processes running in Linux can open these files with the common file operations that invoke the corresponding callbacks in the driver running in the Linux kernel.

Each real-time application can configure multiple pipes which are mapped to character devices named `/dev/slothpipe0`, `/dev/slothpipe1`, and so on. These character devices are persistent for the runtime of the real-time application in SLOTHFUL LINUX and model the behavior of *named pipes* or FIFOs. User space processes can open them at any time to read the data stream that has been written previously by real-time tasks. The implementation is limited to one direction as the tasks implemented in the SLOTH real-time core have run-to-completion semantics and, thus, cannot block and wait for more data. A bidirectional communication channel would be possible by extending the implementation with a non-blocking read operation in the real-time system in future work.

At load time of the real-time application in form of a Linux kernel module, the required pipes with callbacks are allocated and the corresponding device nodes are created. As pipes implement a unidirectional communication channel, the callback for the open

System Service	Functionality
<code>WritePipe(PipeID, string)</code>	writes a string to a pipe
<code>WritefPipe(PipeID, format, ...)</code>	writes a formatted string to a pipe
<code>WritePipeLocked(PipeID, ResID, string)</code>	same as <code>WritePipe()</code> , synchronizing access on a resource
<code>WritefPipeLocked(PipeID, ResID, format, ...)</code>	same as <code>WritefPipe()</code> , synchronizing access on a resource

Table 4.2: The system services provided by SLOTHFUL LINUX for writing pipes in the real-time system.

system call ensures that Linux user processes can only open them in read-only mode and does not allow write operations from user space. As the pipe storage is persistent, each pipe is backed by a ring buffer that stores the data written from the real-time task until the data is read from the receiving end. The buffer has a static size which is subject to the application configuration.

For the real-time core, new system services have been introduced following the naming scheme of the other system services as shown in Table 4.2. `WritePipe()` writes a simple null-terminated string into the buffer of the pipe. The system service `WritefPipe()` implements common *printf()*-style formatting, which produces output according to directives given in a format string referring to the following arguments.

In these two system services, access to the pipe object is not protected. If a task of higher priority preempts another task that is currently writing a string to a pipe and subsequently, the high-priority task also outputs data to the pipe, the resulting data stream in the pipe will be mangled. Thus, if writes to a single pipe can occur in multiple control flows, the access to the pipe needs to be synchronized. This can be achieved by using resources in SLOTH that provide system services for mutual exclusion. The application programmer is responsible to ensure the correct ordering of the data written to a pipe. However, two separate system services `WritePipeLocked()` and `WritefPipeLocked()` aid the programmer by offering small wrappers that temporarily acquire the given resource while writing to a pipe.

In summary, pipes allow the real-time application running in SLOTHFUL LINUX to transfer data to the general-purpose operating system. Concurrent read and write accesses do not need synchronization between Linux and the real-time core, however, the application programmer needs to ensure only one write operation occurs in the real-time core at the same time. The read operations on the corresponding character devices provided by the Linux kernel allow pipes to retrieve the data written by the real-time tasks without influencing the behavior of the real-time system.

4.2.5 Summary

SLOTHFUL LINUX applies small modifications to the Linux kernel to intercept interrupt vectors in order to bypass the interrupt pipeline of the I-Pipe patch to reduce the latency of task dispatches and take advantage of the interrupt-driven scheduling in the SLOTH real-time core. The task and resource management follows the bare-metal implementation with the same utilization of the local APIC. With new system services, pipes offer a communication channel by use of a shared buffer between the real-time domain and the Linux kernel, which will be read by character devices from user space processes in Linux.

Chapter 5

Evaluation

The implementation of SLOTHFUL LINUX presented in the previous chapter implements a hybrid system where a real-time core runs concurrently with Linux on the same hardware. This chapter evaluates the resulting implementation in comparison with the bare-metal implementation of SLOTH for the Intel x86 and Xenomai [20], another hybrid system based on the Linux kernel. Although real-time systems usually provide a lot of features on the kernel level, this evaluation will focus on performance and latency as these are the properties where the SLOTH concept has already proven its positive effects.

This chapter first presents the metrics used in the evaluation in Section 5.1 and explains the setup of both the hardware platform and the operating systems in Section 5.2. The evaluation of the performance in Section 5.3 includes the system services in both the standalone SLOTH and SLOTHFUL LINUX. Section 5.4 presents measurements on the interrupt latency induced by the handling of interrupts in the competing operating systems. Section 5.5 discusses the current limitations of SLOTHFUL LINUX implemented on the Intel x86.

5.1 Evaluation Metrics

Operating systems are usually only used as a means to an end, as the applications running on them provide the actual functionality of the whole system. Therefore, it is especially important for an embedded operating system to provide high performance in the system services and to use little resources to leave as much as possible for the application, while still fulfilling the intended duties.

In this evaluation, the performance of the system services is assessed by the amount of clock cycles they take for their execution. This includes the system services of a SLOTH system as presented in Section 3.1. The task management includes synchronous task activation with and without dispatch and the termination and chaining of tasks, which inherently cause a task switch as the current control flow is ended. The management of

resources contains the acquisition of a resource as well as releasing the resource again, which can lead to a task dispatch in certain situations.

Additionally, the way an operating system handles interrupts has an influence on the predictability of the system. In order to compare the behavior of the systems, measurements need to be taken on the interrupt level. The *interrupt latency* is defined as the time interval between the point in time where the interrupt request is issued and the time the execution of the corresponding handler function starts. Thus, this is a metric for the overhead the operating system adds for the management of asynchronous triggered interrupts.

In the following experiments, the performance of the system services will be measured by the required clock cycles for their execution that will be obtained from the time-stamping counter of the processor. The interrupt latency will be measured with help of the local APIC timer.

5.2 Platform Setup

The measurements for the evaluation of the system performance were taken on an Intel Embedded Development Board with an Intel Atom D510 [22, 23]. The processor was configured in the BIOS such that sources of indeterminism have been ruled out. Power-saving features usually exposed via ACPI were disabled, the front-side system bus is running at 166.6 MHz, and the CPU clock with a multiplier of 10 runs at a constant rate of 1.6 GHz. This ensures that the measurements are not influenced by dynamic frequency scaling or throttling of the CPU in T-States by omitting duty cycles. Furthermore, although this Atom CPU offers two separate processing cores with Hyper-Threading¹, only one single core has been used in these tests—a uni-processor system with SMT disabled. The implementation of SLOTHFUL LINUX presented in this thesis targets a single processor system and these configuration choices ensure that the measurements are obtained on a single core.

The evaluation includes both systems presented in this thesis—the standalone SLOTH implementation and SLOTHFUL LINUX. Additionally, Xenomai was chosen as an alternative implementation of a real-time extension for Linux. The Linux systems are both based on kernel 2.6.38.8, each with the latest I-Pipe patch version 2.11-02 for x86; the Xenomai system is version 2.6.0. Both Linux systems were booted up in single-user mode and no artificial load was generated during the experiments.

As both SLOTH systems implement the same system services, the performance of task switches and resource management can be evaluated with the same test application. For comparison of SLOTHFUL LINUX with Xenomai, no common task management interface is available. The SLOTH concept targets systems with a static priority space that is configured at compile-time where the priorities of the tasks do not change at runtime of

¹Hyper-Threading is Intel’s marketing name for simultaneous multithreading (SMT), where multiple threads can be executed concurrently in different pipeline stages of a superscalar processor.

the system. In Xenomai, however, tasks are created dynamically such that new tasks can be spawned at any time. Also, Xenomai tasks always provide their own context with their own stack in contrast to the run-to-completion tasks implemented in SLOTHFUL LINUX. Thus, a comparison of the task management provided by Xenomai against the real-time core of SLOTHFUL LINUX would not yield interesting results since the two concepts are too different. The methods used for obtaining the performance measurements of the system services are described in Section 5.3.1.

Nevertheless, the interrupt latency can be compared across all three systems. The latency in the standalone SLOTH system was measured for comparison of the overhead induced in the real-time core of SLOTHFUL LINUX due to the domain switches. Section 5.4.1 explains how the interrupt latency of the systems was observed in the evaluation experiment.

5.3 Performance Evaluation

The system services provided by the systems implementing the SLOTH concept are measured in test cases, which give a comprehension on the performance of the operating system implementation.

5.3.1 Performance Benchmarking with the Time-Stamp Counter

The performance of the system services was measured by the clock cycles spent for their execution. The Intel x86 CPUs provide a time-stamping mechanism that can be used to measure the relative time occurrence between two events. The time-stamp counter is a 64-bit value that starts with the initial value of zero at reset of the processor and increments by each processor clock cycle—effectively running at the same rate. There are slight differences between processors when dynamic frequency scaling is involved [21, Ch. 3B], however, this has been ruled by the BIOS configuration as described in the previous section. For these measurements, the time-stamp counter (TSC) increments at the constant rate of the processor clock. This counter can be read with the `rdtsc` instruction that stores the 64-bit value separated into its high and low 32-bit parts into two registers.

However, due to the superscalar property of modern x86 CPUs, precise measurements of code execution need to be carried out with due consideration. To benchmark the cycles spent for execution in a naive way would be to read the time-stamp counter with the `rdtsc` instruction, then run the code to be measured. Afterwards, read the time-stamp counter again and calculate the difference between the two obtained values to get the amount of elapsed clock cycles. The problem with this approach is that modern x86 CPUs feature out-of-order execution, that modifies the temporal sequence of independent instructions in order to optimize the throughput of the processor [21, Ch. 1].

Thus, to guarantee all instructions between the two calls to `rdtsc` and no instructions outside this range will be measured, serializing instructions need to be added to the

```

uint32_t startLow, startHigh, endLow, endHigh;
uint64_t start, end, duration;
/* Serialize instruction stream and read time-stamp counter at start */
asm volatile (
    "xor %%eax, %%eax\n"
    "cpuid\n"
    "rdtsc\n"
    "mov %%edx, %0\n"
    "mov %%eax, %1\n"
    : "=r" (startHigh), "=r" (startLow)
    : : "%eax", "%ebx", "%ecx", "%edx");

/* ... the code to be measured needs to be placed here ... */
/* Serialize instruction stream and read time-stamp counter at end */
asm volatile (
    "mov %%cr0, %%eax\n"
    "mov %%eax, %%cr0\n"
    "rdtsc\n"
    "mov %%edx, %0\n"
    "mov %%eax, %1\n"
    : "=r" (endHigh), "=r" (endLow)
    : : "%eax", "%edx");

/* Calculate elapsed CPU cycles */
start = (((uint64_t) startHigh) << 32) | startLow;
end   = (((uint64_t) endHigh) << 32) | endLow;
duration = end - start;

```

Figure 5.1: The benchmarking method recommended by Intel for measuring elapsed clock cycles during code execution [24].

instruction stream [24, p. 9]. A serializing instruction forces the CPU to complete any preceding instructions before advancing to the next one. The serializing instruction `cpuid`, which usually returns information about the processor, is added right before the first `rdtsc` instruction. Using the `cpuid` instruction again after the execution of the measured code for serialization would add variance to the measurement that reduces its resolution. Therefore, the recommended alternative is the sequence of reading and writing the current value of the `cr0` control register to itself [24, p. 13]. The resulting measurement code is shown in Figure 5.1. As this method obtains a 64-bit value from the TSC, the measurement duration can amount to years before the counter value overflows, which is definitely enough for this purpose.

In the setup of the test cases it was observed that due to the serialization instructions, the resolution of the measurements appears to be limited to the clock rate of the system bus on the particular Intel Atom board used for the evaluation, as the CPU seems to synchronize on the bus clock at this point. Thus, the accuracy of the measurements is

limited to 10 cycles. The measurements need to be repeated multiple times especially as the fill of the instruction cache at the beginning might introduce deviations.

In order to get specific measurements of the code sections to be tested, the overhead introduced by this benchmarking method needs to be filtered out. This offset was obtained before starting the actual measurements by running the serialized `rdtsc` sequence multiple times. This value was then subtracted from all other measurements to get the actual clock cycles required for the execution of the code only. As the CPU is running at a constant rate of 1.6 GHz, each clock cycle takes 0.6 ns and the results in the following sections are presented in both clock cycles of the CPU and in absolute nanoseconds.

The correctness of this benchmarking approach was verified by taking measurements over a fixed amount of simple instructions, such as a hundred instructions of `xor` or `mov` with register operands. The result showed that both the benchmarking and the calculation of the overhead are correct.

5.3.2 Results of the Performance Evaluation

A test application using the system services related to tasks and resources was used for the measurements on both the standalone SLOTH system and the real-time core of SLOTHFUL LINUX. This application uses multiple tasks, where the performance of task switches and the cost of synchronization using resources can be observed. The selected test cases measure the system services from the point before the invoking statement until the action is completed. For example, a task activation involving preemption of the current task is measured from the point before `ActivateTask()` to the first instruction in the task function of the activated task. If no preemption or another task switch occurs, the measurement is stopped at the instruction right after the invocation of the system service.

Standalone Sloth

The results for the system services in the standalone SLOTH system running on the Intel x86 are presented in Table 5.1. The first test case *S1* takes 162 cycles, as without a dispatch the synchronous task activation by sending a self-IPI is merely a write instructions to the memory-mapped register of the local APIC to set the destination and vector number in order to trigger the IPI. Of course, as this is an synchronous action, a short delay might be caused by the APIC for the arbitration as well. In test case *S2*, a task with a higher priority is activated; thus, the currently running task is preempted and the other task is dispatched. The 601 cycles listed in the table include both the triggering of the self-IPI, the arbitration in the local APIC, the jumps through the IDT and the prologue of the activated task until the first instruction of the application provided function is reached.

The termination of a task in *S3* takes 219 cycles, whereas chaining another task in *S4* requires 810 cycles, the most time-consuming system service measured. However,








Test Case			CPU Cycles	ns	rel. comp.
<i>S1</i>	<code>ActivateTask()</code>	without dispatch	162	97.2	
<i>S2</i>	<code>ActivateTask()</code>	with dispatch	601	360.6	
<i>S3</i>	<code>TerminateTask()</code>	with dispatch	219	131.4	
<i>S4</i>	<code>ChainTask()</code>	with dispatch	810	486.0	
<i>S5</i>	<code>GetResource()</code>	without dispatch	160	96.0	
<i>S6</i>	<code>ReleaseResource()</code>	without dispatch	130	78.0	
<i>S7</i>	<code>ReleaseResource()</code>	with dispatch	560	336.0	

Table 5.1: Performance evaluation of task switching and resources in the standalone SLOTH system. The values specify the measured execution time in number of clock cycles and nanoseconds for the Intel Atom D510 running at 1.6 GHz; the bar charts on the right present a relative comparison.

chaining a task means that the current task is terminated and another task is activated and dispatched. Taking the values of the task termination *S3* and task activation *S2* together, this almost equals the cycles measured for task chaining, where the slight difference is only due to the resolution of the measurements of up to 10 cycles as explained in Section 5.3.1.

The system service for acquiring a resource takes 160 cycles in test case *S5*, this includes a synchronized store of the execution priority on a stack and raising the execution priority with the TPR of the local APIC. For releasing the resource again without a dispatch—that means according to the priorities no task activation is pending—takes 130 cycles in *S6* which is slightly faster than the acquisition as it only needs to restore the previous priority from the top of the stack. If another task was activated in between with a priority between the previous execution priority and the ceiling priority of the acquired resource, a dispatch needs to occur as soon as the resource is released. This situation is measured in test case *S7*, where the dispatch from the release of the resource until the first instruction of the other task function takes 560 cycles. This is slightly faster than an `ActivateTask()` with dispatch in *S2* since the writes to the registers of the local APIC and the assertion of the interrupt request already occurred before the measurement started.

Overall, the measurements of the implementation of SLOTH for the Intel x86 platform show a consistent picture on the performance of the system services. Due to the concise kernel of a system implementing the SLOTH concept, the impact of task switches and the influence of the different actions on the system services can be interpreted quite well.

Slothful Linux

The same test application used for the standalone SLOTH system was also executed on the real-time core of the SLOTHFUL LINUX implementation. Both systems implement








Test Case			CPU Cycles	ns	rel. comp.
<i>L1</i>	<code>ActivateTask()</code>	without dispatch	167	100.2	
<i>L2</i>	<code>ActivateTask()</code>	with dispatch	946	567.6	
<i>L3</i>	<code>TerminateTask()</code>	with dispatch	558	334.8	
<i>L4</i>	<code>ChainTask()</code>	with dispatch	1418	850.8	
<i>L5</i>	<code>GetResource()</code>	without dispatch	166	99.6	
<i>L6</i>	<code>ReleaseResource()</code>	without dispatch	137	82.2	
<i>L7</i>	<code>ReleaseResource()</code>	with dispatch	817	490.2	

Table 5.2: Performance evaluation of task switching and resources in the real-time core of SLOTHFUL LINUX. The values specify the measured execution time in number of clock cycles and nanoseconds for the Intel Atom D510 running at 1.6 GHz; the bar charts on the right present a relative comparison.

the same interface, so no changes to the application source code or configuration were required. The results of the measurements are shown in Table 5.2.

The connections between the different system services are similar to the standalone SLOTH system. The first test case *L1*, a task activation without a dispatch takes 167 cycles in SLOTHFUL LINUX, which is the same as in the standalone SLOTH system when taking the resolution of the measurements into account. However, a task activation with a dispatch in *L2* takes 946 cycles, as compared to the 601 cycles measured in SLOTH. Although bypassing the interrupt pipeline of the I-Pipe patch, the additional approximately 300 cycles in SLOTHFUL LINUX are subject to the switch to the real-time domain, which is always performed even though the interrupted code was already executing in the SLOTH domain as explained in Section 4.2.2. This can be observed in task termination in the same way, where *L3* in SLOTHFUL LINUX takes 558 cycles, which is almost the same amount of cycles more compared to the standalone SLOTH implementation with 219 cycles in test case *S3*. Again, at task termination, the interrupted domain needs to be restored which happens regardless whether the domain to return to already was the real-time domain. For `ChainTask()` in *L4* with 1418 cycles, this domain switching is applied twice. The switch is required once for termination of the current control flow and again in the prologue of the dispatched task. In comparison with *S4* in the standalone SLOTH, task chaining takes approximately 600 cycles more in SLOTHFUL LINUX—which is twice the domain switching overhead observed before.

On the other hand, the resource management in the real-time core of SLOTHFUL LINUX is identical with respect to the measurement resolution. The system services `GetResource()` in *L5* and `ReleaseResource()` without dispatch in *L6* take 166 and 133 cycles, respectively. However, releasing a resource with a task dispatch involved requires 817 cycles in test case *L7*. Once again, compared to the 560 cycles in test case *S7* in the standalone SLOTH, the additional overhead induced by domain switching can be seen in SLOTHFUL LINUX.

5.3.3 Summary of the Performance Evaluation

The performance evaluation of both the standalone SLOTH implementation and the hybrid SLOTHFUL LINUX show similar results. However, task switches in the real-time core of SLOTHFUL LINUX are affected by an additional overhead caused by the domain switching that is required for the interrupt pipeline.

5.4 Evaluation of Interrupt Latency

The interrupt latency indicates how the operating system reacts on interrupt requests. In real-time systems, it is not only important to have fast handling of interrupts; rather, the timeliness of interrupt handling should be predictable with low variability.

5.4.1 Latency Measurements with the Local APIC Timer

Accurate timing measurements of the interrupt latency are difficult to measure by use of external devices since the instant at which an interrupt is requested—an asynchronous event—needs to be determined. The latency measurements in this thesis did not resort to peripheral hardware, instead, the interrupts were generated internally by using the local APIC. Thus, the measurements only determine the interrupt latency that is actually induced by software. Any arbitration and communication usually happening on the buses between the hardware device, the I/O APIC and the local APIC have been neglected. As the handling of interrupts in software is the only factor that can be influenced in an operating system, this metric allows to compare the different implementations.

In addition to the functionality described in Section 3.2, the local APIC contains a timer that is used to trigger a local interrupt based on a counter timeout value. Software sets up the initial count register with the timeout value, which is then copied to the current count register and decremented there at a fixed rate. When the current count register reaches zero, an interrupt is generated. The corresponding vector to be set to pending is configured in its local vector table (LVT) entry. The clock source for the timer is derived from the system bus clock, divided by the value specified in the divide configuration register ranging from 1 to 128.

The local APIC offers two modes for the timer: periodic and one-shot. When driving the local APIC timer in periodic mode, the initial value is loaded into the current count register again and the countdown is repeated. In one-shot mode, the current count remains at zero until the initial count is reprogrammed.

The timer functionality of the local APIC is usually in use by operating systems for timer interrupts. However, as it is able to trigger local interrupts in an asynchronous manner, it will be used to measure the interrupt latency induced by the systems. To prevent the Linux kernel in SLOTHFUL LINUX from using the timer component of the local APIC, the system was booted with the kernel command line argument `nolapic_timer`. Thereby, Linux switches to another time source available on the hardware such as the

High-Precision Event Timer (HPET) provided by the chipset for periodic timer interrupts. For the Xenomai system, the local APIC timer needs to be enabled at startup as otherwise the Xenomai initialization does not succeed. However, as no alarms are set up through the interface, the local APIC timer is actually unused. Thus, the timer of the local APIC can freely be used by test applications for measurement purposes.

The interrupt latency is defined as the time between the instant where an interrupt is triggered and the moment the registered interrupt handler starts its execution. When executing in periodic mode, the current count register of the local APIC is reset at the same time the interrupt is issued. Thus, reading this value returns the time left until the next timer deadline or, subtracting the current count register from the initial count returns the amount of clock ticks that occurred since the last reset.

Therefore, the setup for measuring the interrupt latency programs the local APIC timer to run in periodic mode with a divide value of 1; thus, the timer ticks occur at the same rate of the system bus. The configured handler function for the corresponding vector immediately reads the current count register and calculates the interrupt latency as the difference to the initial count. As the local APIC timer is counting at the system bus clock rate, the resolution of these values is one tenth of the CPU clock cycles. For the presentation, all measurements have been converted to CPU clock cycles to avoid confusion.

5.4.2 Results of the Interrupt Latency Evaluation

The interrupt latency induced by the interrupt handling was measured for the standalone SLOTH implementation, for the real-time core of SLOTHFUL LINUX, and for the Xenomai real-time extension for Linux. For the two systems implementing the SLOTH concept, a task was activated asynchronously by using the local APIC timer as described in the previous section. The measurement of the latency was taken at the first instruction of the task function. In the Xenomai system, an interrupt handler was set up using the `rt_intr_create()` API call, which also measured the interrupt latency at the first instruction of the registered handler function. To get a fair comparison, this was set up as a kernel module in Xenomai, as the SLOTHFUL LINUX system also loads its applications into kernel space. In the two Linux competitors, the general-purpose part was left idle and no artificial load was generated on the systems. The measurements were repeated 5000 times to get a comprehensive view on the interrupt handling of the three systems.

The results of the measurements for all three systems—standalone SLOTH, SLOTHFUL LINUX, and Xenomai—are presented in Figure 5.2.

Standalone Sloth

The SLOTH implementation on the bare-metal Intel x86 was measured for illustration purposes as shown in Figure 5.2(a). As it runs as an RTOS alone on the hardware, it does not need to cut back and respect other systems. As SLOTH has full control over the

hardware it is the fastest of the three measured systems. The interrupt latency is stable at a mean value of 450.5 cycles, with a standard deviation of 6 cycles only.

When put into relation to the values obtained for the synchronous task activations in the standalone SLOTH system presented in Table 5.1, the experiments give a coherent picture. The synchronous task activation with dispatch in test case *S2* amounts to 601 cycles, whereas the synchronous task activation without a dispatch in test case *S1* takes 162 cycles. For the interrupt latency measurement, the activation occurs asynchronously by the local APIC timer. Thus, with respect to the resolution of the two methods for obtaining the measurements, the difference between these two values approximately equals the 450 cycles obtained in the interrupt latency measurement presented here.

Slothful Linux

The second graph Figure 5.2(b) shows the measured values for the real-time core of the SLOTHFUL LINUX system. As can be seen, the values are higher than for the standalone SLOTH system. As already observed in the performance evaluation of the system services in Section 5.3.2, the interrupt dispatching in SLOTHFUL LINUX induces additional 300 cycles for the domain switch. However, the overhead appears to be twice as high in this case with a median value of 1050 cycles in SLOTHFUL LINUX compared to the median value of 450 cycles for the standalone SLOTH system.

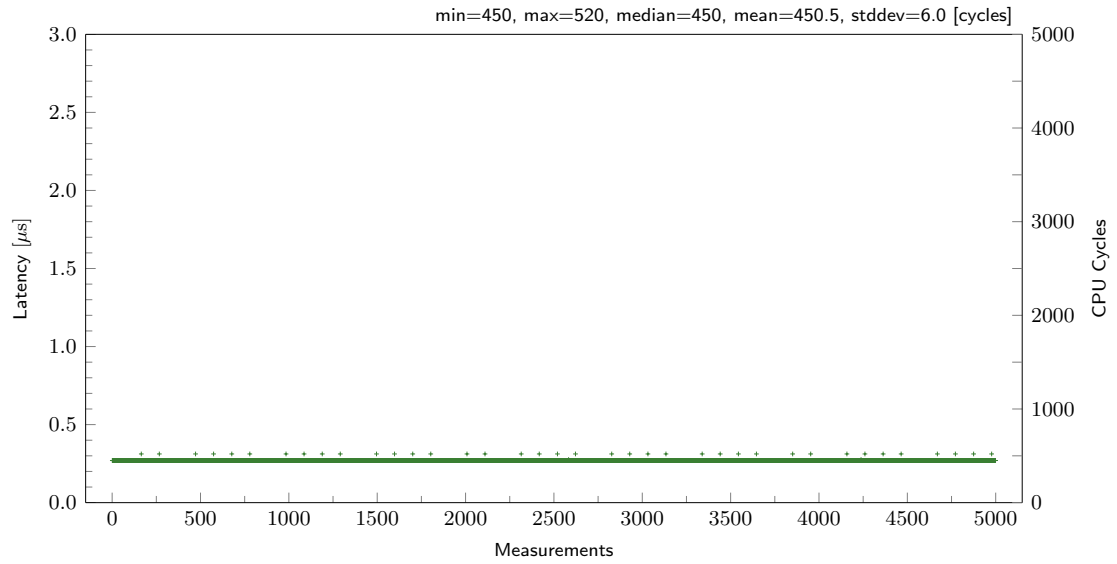
The overhead can also be recognized when applying the same calculations in relation with the synchronous task activations presented in Table 5.2 as above for the standalone SLOTH system. The synchronous task activation with dispatch in test case *L2* takes 946 cycles, which is 779 cycles more than the synchronous task activation without dispatch in test case *L1* with 167 cycles. Compared to the 1050 cycles determined for the interrupt latency here, there are approximately 270 cycles remaining.

There is no obvious source of these additional cycles; thus, further investigation is required on this topic; a possible approach would be to trace the interrupt dispatching sequence. Otherwise, this could also be subject to the multi-level caching of the Intel x86 platform. Unlike in the previous measurements on the synchronous task activations, the Linux kernel runs between the asynchronous task activations in this case. Thus, if the Linux kernel issues cache flushes here, these would also affect the latency of the interrupt handling as the IDT, the SLOTH trampoline, the SLOTH dispatcher table, and the task function itself provided by the real-time application kernel module are loaded from memory. As each memory access passes through the MMU, a TLB miss could induce latency as well. A solution for this problem could be locking relevant instructions in the cache as this would avoid memory accesses and thus, reduce the latency.

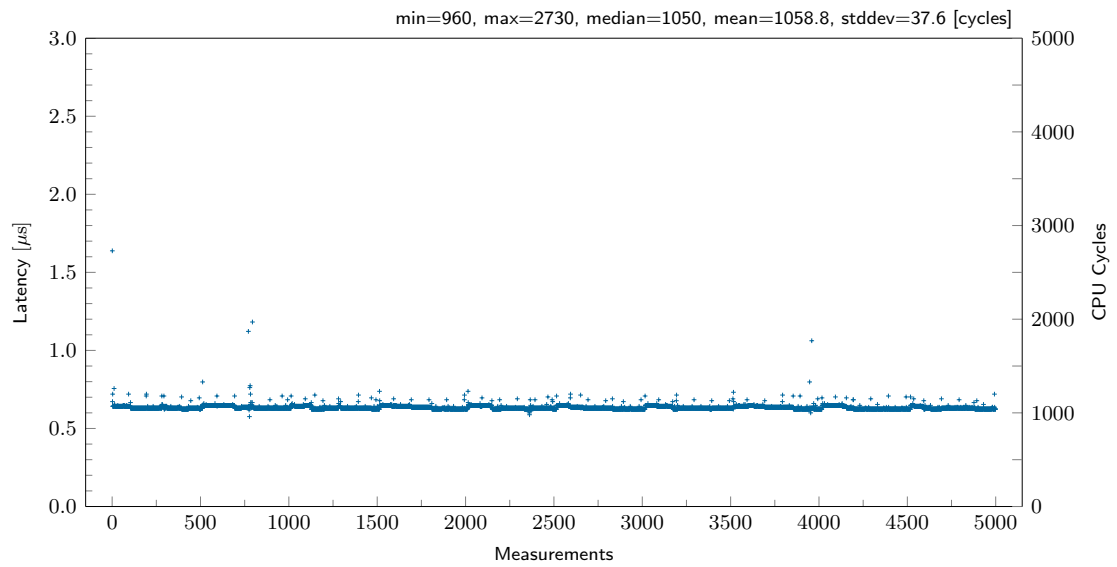
Xenomai

The data gathered for the Xenomai system is presented in the third graph in Figure 5.2(c). With a standard deviation of 85.5 cycles for Xenomai, the values are not

(a) SLOTH



(b) SLOTHFUL LINUX



(c) Xenomai

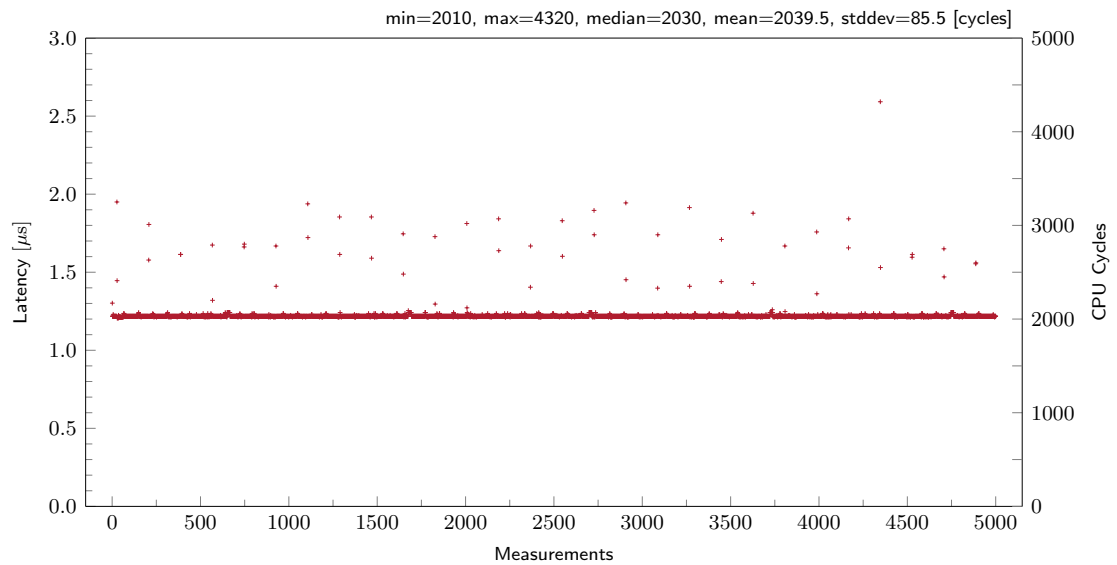


Figure 5.2: Comparison of the interrupt latency induced by the standalone SLOTH system, the real-time core in the SLOTHFUL LINUX system and the Xenomai system.

as stable as the SLOTHFUL LINUX system with a standard deviation of 37.6 cycles. This can also be observed clearly by the distribution of the measurements in the plot. The interrupt latency measured for Xenomai with a median value of 2030 cycles is even higher than in SLOTHFUL LINUX with a median value of 1050 cycles.

Despite the remaining overhead of unknown origin in SLOTHFUL LINUX, it already provides lower interrupt latency compared to Xenomai, which is due to the interception of the vectors specific to the SLOTH system avoiding the interrupt pipeline for their dispatch.

5.4.3 Summary of the Interrupt Latency Evaluation

The standalone SLOTH system has the lowest interrupt latency as it has full control over the hardware and does not require as many indirections in the interrupt dispatching as the real-time core in SLOTHFUL LINUX. This hybrid system is affected by the overhead of the domain switching and probably an additional overhead of uncertain source is added by caching of the memory accesses. The measurements observed both a higher latency and a higher deviation for the Xenomai system than for the SLOTHFUL LINUX system, as the latter circumvents the interrupt pipeline.

5.5 Limitations

As observed in the evaluation, the Intel x86 platform is quite complex due to the pipelining of the CPU and the influence of memory accesses by multi-level caching, which need to be respected in the development of real-time systems for this hardware. The most important factor in favor of using this platform for the actual deployment for real-time applications is the high execution performance.

The SLOTHFUL LINUX implementation presented in this thesis was developed on the Intel x86 as a reference, while the design of SLOTHFUL LINUX is not limited to this platform. In fact, any platform that fulfills the requirements on the interrupt controller defined in Section 1.5.2 and that is supported by the Linux kernel would be a feasible target for the implementation of a hybrid system based on this concept.

However, the maximum number of different tasks in a real-time application in a SLOTH system is limited by the number of priorities offered by the hardware platform. While the Intel x86 offers 224 different interrupt vectors in the local APIC, they are spread into 14 priority groups determining the preemption of interrupt handlers. Thus, as the standalone SLOTH implementation is already restrained on the bare-metal platform, sharing the priority space with the Linux kernel reduces the number of tasks in the real-time core of SLOTHFUL LINUX even more. As triggering an interrupt of the real-time core is equivalent to a task activation, the available interrupt vectors need to be strictly partitioned and cannot be shared with the Linux system.

The implementation of SLOTHFUL LINUX is currently bound to the use of basic tasks with run-to-completion property only, which matches exactly the strictly nested dispatch-

ing of interrupt handlers exactly. The original SLOTH concept was already extended to overcome this limitation on the TriCore platform [3]. However, as the local APIC does not provide a way to disable the dispatch of specific interrupt vectors, the approach presented in the extension for SLOTH on the TriCore could not be applied to the Intel x86 and further research on this topic is necessary.

Also, as the MMU is not reconfigured at the dispatch of real-time tasks and the task functions are running with the privileges of the Linux kernel without memory protection, a badly behaving real-time application could tamper with the internals of the kernel. This should not pose a problem in actual systems; after all, loading a real-time application bundled as a kernel module must be carried out by the superuser, who would have access to the whole system anyway.

The Linux kernel used as the base for SLOTHFUL LINUX supports many different hardware platforms and its development follows a fast pace that arose some problems during the implementation of SLOTHFUL LINUX. While the interface offered to user-space applications is stable and standardized by specifications, the internal interfaces offered to kernel modules are subject to a steady evolution in each kernel version. Thus, for an extension such as SLOTHFUL LINUX, it is necessary to adapt to changes in the kernel and re-evaluate the system on a regularly basis.

5.6 Summary

The evaluation of the performance of the system services showed that the real-time core of SLOTHFUL LINUX could reach the same values as SLOTH. However, the hybrid system is affected by the domain switching of the interrupt pipeline provided by the I-Pipe patch. While the interrupt latency evaluation could not reveal all sources of induced latency in SLOTHFUL LINUX, it already reached a latency lower than the Xenomai system.

The interrupt pipeline is the main source for these additional overhead, although it is only being used for the optimistic interrupt protection in the domain of the Linux kernel. Thus, replacing the interrupt pipeline with a less complex implementation could reduce or remove this overhead.

Chapter 6

Conclusion

In this thesis, the SLOTH concept of interrupt-driven scheduling was applied to the efficient hybrid real-time system SLOTHFUL LINUX, where both an RTOS and a GPOS run concurrently on the same hardware. By use of interrupt virtualization, the presented design of SLOTHFUL LINUX supports execution of tasks in the real-time core in the same way as a standalone SLOTH implementation, where scheduling and dispatching of tasks is handled by the interrupt controller. As the Linux kernel is only running in the idle time of the SLOTH system, it does not influence the real-time properties of the hybrid system.

SLOTHFUL LINUX provides *pipes* as communication channels between the real-time core and user-space processes running in Linux, which can be used to transfer data in order to implement gathering of statistics or visualizations in the general-purpose part of the system. Thus, the Linux part can implement interaction with humans or network connections to components that do not need to occur with real-time properties.

The presented SLOTHFUL LINUX implementation has proven that the SLOTH approach of interrupt-driven scheduling is feasible to be used in a hybrid system achieving both real-time and general-purpose activities on the same hardware platform. The evaluation of the hybrid system showed that the required transition between the general-purpose and the real-time domain for all task switches in SLOTHFUL LINUX induces a constant additional overhead compared to the standalone SLOTH system. However, the control flows of the real-time core in SLOTHFUL LINUX are again managed according to the SLOTH concept by using the interrupt subsystem for scheduling and dispatching of tasks, which has positive effects on the non-functional properties of the system. Although applying the existing I-Pipe patch for optimistic interrupt protection, SLOTH avoids the interrupt pipeline for the dispatching of tasks by intercepting the corresponding interrupts. Thus, the interrupt latency of the concise SLOTH core observed in the evaluation is half as low as the interrupt latency of another real-time extension for Linux. However, SLOTHFUL LINUX does not achieve the same low latencies as the standalone SLOTH system, which needs further investigation on the interrupt dispatching sequence that shows deviations due to caching.

The topic of integrating SLOTH with a GPOS leaves room for enhancements in future work. As the I-Pipe patch is only applied for the optimistic interrupt protection in the Linux kernel and the supplied interrupt pipeline is not used for the dispatching of real-time tasks in the SLOTH core, a less complex implementation could replace the utilized functionality of the I-Pipe patch in order to simplify the structure of the system.

Also, with the ongoing trend towards multi-core systems in the embedded market, SLOTHFUL LINUX could benefit from using multiple cores. A possible approach would be to deploy a different real-time application on each processor core, while the Linux system can run concurrently on all cores by the same technique as presented in this thesis. In contrast, multiple cores could also be used to run both systems in parallel at the same time by arranging the RTOS and the GPOS on distinct cores.

Bibliography

- [1] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ, USA, 2000.
- [2] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. Sloth: Threads as interrupts. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS 2009)*, pages 204–213, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
- [3] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Sleepy Sloth: Threads as interrupts as threads. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS 2011)*, pages 67–77, Los Alamitos, CA, USA, December 2011. IEEE Computer Society.
- [4] Rainer Müller. Implementation of an interrupt-driven OSEK operating system kernel on an ARM Cortex-M3 microcontroller. Study Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, October 2011.
- [5] Linus Torvalds et al. Linux kernel source.
<http://kernel.org>.
- [6] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005.
<http://portal.osek-vdx.org/files/pdf/specs/os223.pdf>.
- [7] Kaushik Ghosh, Bodhisattwa Mukherjee, and Karsten Schwan. A Survey of Real-Time Operating Systems. Technical report, College of Computing, Georgia Institute of Technology, 1994.
- [8] Matteo Marchesotti, Roberto Podestá, and Mauro Migliardi. A measurement-based analysis of the responsiveness of the Linux kernel. In *13th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2006)*, pages 397–408. IEEE Computer Society, March 2006.

- [9] Paul Regnier, George Lima, and Luciano Barreto. Evaluation of interrupt handling timeliness in real-time Linux operating systems. *SIGOPS Operating Systems Review*, 42(6):52–63, October 2008.
- [10] Ingo Molnar et al. Real-Time Linux (CONFIG_PREEMPT_RT). <https://rt.wiki.kernel.org/>.
- [11] Paul McKenney. A realtime preemption overview. <http://lwn.net/Articles/146861/>, August 2005.
- [12] Arther Siro, Carsten Emde, and Nicholas McGuire. Assessment of the realtime preemption patches (RT-Preempt) and their impact on the general purpose performance of the system. In *Proceedings of the 9th Real-Time Linux Workshop*, 2007.
- [13] Daniel Stodolsky, J. Bradley, Chen Brian, and N. Bershad. Fast interrupt priority management in operating system kernels. In *In Second USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 105–110. USENIX, 1993.
- [14] Victor Yodaiken and Michael Barabanov. A Real-Time Linux. *Linux Journal*, 34, 1997.
- [15] Michael Barabanov. A Linux-based real-time operating system. Master’s thesis, New Mexico Institute of Mining and Technology, 1997.
- [16] Karim Yaghmour. Adaptive domain environment for operating systems. Technical report, Opersys, Inc., 2001.
- [17] The Adeos Project. Adaptive Domain Environment for Operating Systems (Adeos). <http://home.gna.org/adeos/>, March 2004.
- [18] Philippe Gerum. Life with Adeos, Revision B. <http://www.xenomai.org/documentation/xenomai-2.6/html/life-with-adeos/>, September 2005.
- [19] The RTAI Project. Real Time Application Interface (RTAI) for Linux. <http://rtai.org>, February 2010.
- [20] The Xenomai Project. Xenomai: Real-Time Framework for Linux. <http://xenomai.org>, May 2012.
- [21] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer Manuals*, May 2012.
<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>.

- [22] Intel Corporation. Embedded Development Boards for Intel Atom Processors.
http://www.intel.com/p/en_US/embedded/designcenter/tools/development-board.
- [23] Intel Corporation. Intel Atom Processor D510.
<http://ark.intel.com/products/43098/>.
- [24] Intel Corporation. *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*.
<http://download.intel.com/embedded/software/IA/324264.pdf>.