
# Using Standard Operating Systems for Time Critical Applications with special emphasis on LINUX

von

**Arnd Christian Heursch**

**Dissertation**
**zur Erlangung des akademischen Grades**
**eines Doktors der**
**Naturwissenschaften (Dr. rer.nat.)**

**Neubiberg, 10. November 2006**

**1. Gutachter: Prof. Dr. rer. nat. Helmut Rzehak**
**2. Gutachter: Prof. Dr. rer. nat. Gunnar Teege**

**Fakultät für Informatik**
**Universität der Bundeswehr München**

Universität der Bundeswehr München
Fakultät für Informatik

| | |
|---|---|
| Thema der Dissertation: | Using Standard Operating Systems for Time Critical Applications with special emphasis on LINUX |
| Verfasser: | Arnd Christian Heursch |
| Promotionsausschuss: | |
| Vorsitzender: | Prof. Dr. Andreas Karcher |
| 1. Berichterstatter: | Prof. Dr. Helmut Rzehak |
| 2. Berichterstatter: | Prof. Dr. Gunnar Teege |
| Weiterer Prüfer: | Prof. Dr. Klaus Buchenrieder |
| Weiterer Prüfer: | Prof. Dr. Mark Minas |
| Tag der Prüfung: | 10. November 2006 |
| Mit der Promotion erlangter akademischer Grad: | Doktor der Naturwissenschaften (Dr. rer. nat.) |

Neubiberg, den 12. November 2006

# Contents

# Acknowledgements

This thesis has been carried out under the guidance of Professor Helmut Rzehak while I worked as a research and teaching assistant at the Institute of Information Technology Systems at the Department of Computer Science of the University of the Federal Armed Forces Munich, Germany.

In this place, I would like to express my profound gratitude to Professor Rzehak for providing me such an interesting theme for which the topicality even grew in recent years. His kind support throughout the years, even when emerited, made this thesis possible.

Working to make a standard operating system like Linux more suitable for time-critical tasks, is a very interesting task, in which a lot of people, i.e. the Linux kernel Open Source community, especially researchers of operating systems and developers of embedded systems, are interested and take part in. So this thesis is embedded in the world wide and ongoing research and development how to make the Linux kernel more compliant for time-critical tasks.

Therefore it was never difficult to publish results on a conference.

It was fun to watch the Linux kernel developping during recent years and also very interesting to take small part in its development a few times.

Prof.Dr. Gunnar Teege, who followed Prof.Dr. Rzehak after his emeritation, I thank for his kind support during the recent years that made it possible to me to continue my work on this thesis. I also thank him very much for proof-reading this thesis.

My special thanks go to Prof.Dr. Burkhard Stiller - who has been the head of our institute for several years - for the good working and learning atmosphere at our institute and for always being addressable.

Many thanks also go to my predecessor, Prof.Dr. Michael Mächtel, who developped in his thesis a software monitor that was helpful to measure and visualize time-critical internal operations of the Linux kernel.

I thank all fellow research assistants for the cooperative working atmosphere.

Especially, I thank Mr.J.P.Sauerland for his kind technical help, and for providing test beds and computers needed to execute this thesis always in time.

Many thanks go to those students of my University who supported this thesis while achieving their diploma thesis or student's research projects in this field of research, namely to Dirk Grambow, Alexander Horstkotte, Witold Jaworski, Alexander Koenen, Mirko Pracht, Dirk Roedel and Robert Koch.

I thank my wife for supporting me and for her loving understanding when this thesis took a lot of time to do, that i couldn't spend with her and the children.

*Arnd Christian Heursch, Neubiberg near Munich, Germany, November 2006*

## Disclaimer

All pieces of information and statements in this document have been carefully composed and are believed to be accurate, they reflect the current state of research. Nevertheless it is possible that parts of it are old, incomplete or even incorrect. The document is provided "as is", without warranty of any kind, without even the implied warranty of merchantability or fitness for a particular purpose. Users must take full responsibility for their application of any products, named herein.

In no event shall the author, the reviewers, the university or any other party be liable for any damages, including any indirect, general, special, consequential, or incidental damages, arising out of the use or inability to use this document, even if the author, the reviewers, the university or any other party has been advised of the possibility of such damages.

Please take into account also the notice at the start of the bibliography.

Linux is a registered trademark of Linus Torvalds. All other trademarks or service marks mentioned in this document are the property of their respective owners.

# Abstract

Traditionally, time-critical tasks have been carried out only by proprietary realtime operating systems running on embedded systems with proprietary hardware, available only for industry or military organizations. Others could normally not afford such systems and thus couldn't execute time-critical tasks on the computer systems available to them.

But in recent decades hardware in computer systems evolved in a fast way and got cheaper, so today many users own off-the-shelf PCs at home that are able to perform so called soft realtime tasks like replaying music and films. While in former times a realtime system was designated only to work for its purpose, the user of today wants to do traditional PC work like writing text, executing calculations, playing games and on the same time for instance hear a music CD or DVD. In order to make an off-the-shelf PC able to do all these tasks at the same time in a quality sufficient to the user, not only the hardware has to evolve, but also changes in the standard operating systems, home users use are necessary.

One can observe a second development: Because of their broad functionality, the huge amount of software available, and the many people supporting Linux and Java by porting it to new hardware or providing new software, standard operating systems like Linux are used more and more even in Embedded Systems like mobile phones, settop boxes or DVD players. Software developers even try to fulfill time-critical tasks with soft-realtime requirements using cheap standard operating systems in order to spare money or to shorten the development period of a product by using available software, that is cheap or even free of charge.

That way, there is a big need to enhance standard operating systems to perform a few time-critical tasks in a more timely manner, while running in parallel to many other tasks without stringent timing requirements.

In this thesis the standard operating system Linux is considered as example because - as an Open Source operating system - the source code can be analyzed and - according to the terms of the GNU Public License (GPL) [15]- it is also allowed to modify its source code.

Furthermore, there's a sort of world-wide development: People working at universities or at software companies create ideas and new concepts, implement them for Linux and publish them as Linux kernel patches/enhancements or new applications available in the Internet.

Users anywhere in the world, who want to tackle the same problems, download these enhancements, try them out and comment whether they are useful or not. Therefore, this world-wide community of users, also called Open Source community, does select the best ideas and concepts out of all ideas available.

That way, good concepts survive, they are implemented, tested and optimized, and in the field of Linux research the best, that can happen to a concept, is that it is judged to be worth to be a part or a standard enhancement of the Linux kernel. The Linux kernel is something like the Digital DNA of a Linux box, i.e. a storage of information, that is worth to be maintained and that is evolving in fast steps in order to serve human beings to run their computers in a better way.

During the time this thesis has been carried out, the stable versions of the Linux kernel 2.2, 2.4 and early versions of 2.6 were in use.

This thesis focuses on tasks with soft-realtime requirements that have to run in parallel to all other tasks the user wants to run on a normal Linux PC.

It analyses in which way the standard operating system Linux can be modified in order to fulfill time-critical tasks in a timely manner, better than the stable Linux kernel versions before could do.

Measurements and evaluations of available concepts as well as new ideas and new concepts and implementations of this thesis have been published throughout the years [22, 23, 24, 25, 26, 27, 28, 29, 30, 60, 89], as it is interesting, useful and necessary to get comments of other developers regarding the own work. It is impossible in this field of research to publish all only after a period of five or six years of thinking, implementing and testing, because the feedback of other researchers and users is the engine that keeps the development of the Linux kernel running as fast as it is.

An important constraint to this thesis is that only the kernel of the operating system shall be modified, while the system call interface used by the applications should remain the same. This is because all

applications running on the standard Linux vanilla kernel should also run on its modified version without having to be changed or recompiled. Furthermore, the performance of the system shouldn't decrease in a noticeable way.

The most important reason for latencies up to the order of fifty or even hundreds of milliseconds on Intel compatible personal computers is that the standard Linux kernels 2.0, 2.2, 2.4 and 2.6 aren't preemptible.

When developing the Linux kernel in recent years, kernel developers carried out two concepts to increase its preemptability:

First, instead of locking the whole kernel every time it works, kernel developers made the Linux kernel preemptible except for the execution of so called critical sections, wherein preemption is still forbidden. This step of development has been called 'Preemptible kernel' and has been developed for Linux 2.4 as additional kernel patches. Linus Torvalds, founder and father of the Linux kernel, accepted the Preemptible kernel to be a configurable element of the standard vanilla Linux kernel 2.6.

At second, another group of Linux kernel developers around Ingo Molnar and Andrew Morton introduced additional preemption points to reduce the length of the remaining critical sections. Some of these additional preemption points are already part of the standard Linux 2.6 kernel, others can be made available when using a kernel patch named 'voluntary preemption' for Linux 2.6 [66].

In this thesis both strategies are compared regarding their theoretical concept, their restrictions and the performance issues of their implementations in the Linux kernel [23].

A monitor for measurements in operating systems, developed in a former thesis at our institute [58], has been ported, used and adapted to new, better hardware to visualize internal procedures of the operating system. In this way it is possible to judge and compare the quality of implemented concepts based on measurements.

A further concept to improve preemptability, discussed in this thesis [25], is to protect the remaining critical sections by mutexes instead of preemption locks.

The use of mutexes causes some problems - priority inversion and the risk of deadlocks for instance as well as interrupt handlers have to be handled by kernel threads, first implemented semi-commercially by Timesys Inc. [96].

This thesis points out how the problems of mutexes in the Linux kernel can be tackled. Different priority inheritance protocols have been analyzed, implemented and tested.

As proof of concept two reference mutex implementations with different priority inheritance protocols for the Linux kernel have been carried out and released under the GNU Public license (GPL) in this thesis [31].

In cooperation with MontaVista Software Inc. [67, 68], an American distributor of Linux for the real-time and embedded market, the 'MontaVista Linux Open Source Real Time Project' has been set up in October 2004 [64]. In this project MontaVista replaced most of the preemption locks, i.e. spinlocks for Multiprocessor systems, of the Linux kernel 2.6 with mutexes using a priority inheritance protocol implemented in this thesis. The Linux kernel patch has been published in October 2004 with the University of Federal Armed Forces, Munich, as early contributor [19, 64].

This first patch led to various publications and activities in this field of research, and the Open Source Community, namely the kernel developer Ingo Molnar, working at Red Hat Inc., and other kernel developers, adopted the idea of a so called 'fully preemptible Linux kernel' and developed a so called RT-patch for the Linux kernel 2.6, that replaces all preemption locks / spinlocks of Linux 2.6 by modified Linux kernel mutexes [66].

This thesis points out that the concept of converting the locks of the Linux kernel does indeed reduce Linux preemption latencies in general, but can't satisfy hard-realtime requirements [25].

A more preemptive kernel is also helpful when providing a more precise time base for the Linux kernel. Up to the Linux 2.4 kernel the periodical timer interrupt is fired every 10 milliseconds.

In the Linux 2.6 kernel, the period of the timer interrupt has been decreased to 1 millisecond. This is an easy, but not very efficient solution regarding performance issues, as measurements show.

In this thesis another concept is presented, that has been implemented for Linux 2.4 providing a much higher timing precision for tasks that shall be scheduled at a predefined time [22]. This kernel patch is based on the Time Stamp Clock (TSC), that has been introduced by Intel with the Pentium architecture.

A reference implementation combining the advances of both focal points of this thesis - high preemptability and a good timing resolution, based on the former UTIME patch [41] - called 'Rapid Reaction Linux' - has been carried out and published in 2001 as a kernel patch for Linux 2.4 and can be downloaded from the web page of our institute [32].

Other kernel developers have pursued the same concept of precise timing, for instance the High Resolution Timer Patch [2]. Today the High Resolution Timer patch is also part of the RT-patch of Ingo Molnar for Linux 2.6 [66].

On the whole, the conclusion can be drawn that standard operating systems like Linux are still not appropriate for hard-realtime tasks, where missed deadlines cause catastrophic results.

But since enhancements - as developed, measured or described in this thesis - are continously developed all over the world, off-the-shelf operating systems are increasingly able to perform time-critical tasks with so called soft-realtime requirements. Examples are mobile phones, audio and video/DVD players or router for IP packets, that are already today based on Linux.

The history of Linux kernel development shows that the best enhancements world-wide can become a part of future standard Linux kernels.

So standard operating systems like Linux could play an important role in the embedded market of the future and diminish the portion of proprietary realtime operating systems, because of their increasing soft-realtime capabilities and the amount of free software available for these standard operating systems.

# Chapter 1

# Introduction

## 1.1 Motivation

The penetration of more and more parts of everyday life by personal computers enlarges the demands made to the operating systems used.

In recent years an increasing number of people use multimedia applications on their personal computers at home or at work, e.g looking TV, playing video or audio data or phoning via IP-based networks.

Time critical tasks in general are characterized by the fact that the user or any other entity require temporal upper limits for these services to be delivered by the operating system.

Hard realtime systems are systems, where missing a deadline leads to disastrous consequences and thus can't be tolerated. Such applications are normally running on specialized hardware using hard realtime operating systems.

On the other hand the above named multimedia applications are referred to as soft-realtime tasks. This means that users can tolerate missing a deadline a couple of times if their over-all performance meets their requirements. Therefore, soft-realtime applications at least meet the user's requirements in a statistical manner. For example, a user communicating via an IP phone using Voice over IP will tolerate a short snap due to a moment of silence of about 40 milliseconds, but he won't accept frequent delays of about 150 milliseconds that hamper a normal conversation and are only accepted by users of radio communication. In case of such high delays he would simply stop phoning over IP networks and revert to normal phones.

For such applications used at work or at home - which are only an add-on to normal applications running on a computer - nobody will pay for expensive proprietary realtime operating systems. As a further drawback, often there is only a small number of software programs available for realtime operating systems, because most of them provide proprietary application interfaces, that don't conform to a special standard.

Therefore standard operating systems have to be extended to meet the requirements of time-critical tasks using off-the-shelf hardware components.

In the field of automation technology, proprietary operating systems are partly superseded by standard operating systems, especially if only soft-realtime requirements have to be met. An important advantage of standard operating systems like Windows or Linux is the broad range of software products that are provided for them. Apart from that, these operating systems already support many different hardware architectures, i.e. main boards and processors; furthermore drivers for many peripheral devices are available. Suppliers port standard operating systems to new processor generations; drivers for new buses and interfaces are normally early available at a cheap price. Enterprises are also benefitting from the fact that many programmers already have skills in the field of standard operating systems. Moreover, standard operating systems are often scalable: For instance the Linux kernel can run on an embedded system as well as on a multiprocessor server. That way applications can easily be ported between different hardware platforms. This advantage exists also for applications that run on the Windows operating system family as well as those running on Java. Furthermore, in the case of Open Source operating systems there are also no royalties to pay.

## 1.2 Aim of this thesis

### 1.2.1 The origin of latencies

Several other works are dedicated to the determination of the maximum execution time of a routine under different conditions [76] on modern desktop computers, also called Worst Case Execution Time or WCET.

This thesis focuses on the problem, that a task of high priority, which has to fulfill a time-critical duty, sometimes has to wait a certain time for the resources it needs. Such resources can be peripheral devices or as well internal data structures of the operating system.

The time until such resources are available to the task of highest priority is called *'latency time'* or simply just *'latency'*.

Such latencies have their origin in the multitasking nature of modern operating systems. The processor is shared among different tasks that are executed one after the other on the same PC. Furthermore, the tasks possibly share a lot of other resources like external medium access, memory and internal lists of the operating system. To maintain the data consistency of stored data on theses resources normally the serialization of parallel access demands is necessary. Therefore an access limitation mechanism is needed, in many cases data consistency requires mutual exclusion.

So sharing multiple resources is a very efficient way to use existing hardware, but it also raises the risk of generating latencies for time-critical processes.

On a multitasking operating system many different processes fulfill different duties: Some are working directly for the users, some are serving to maintain the system. They all have their own timetables and possibly deadlines. Although they are working independently, the fact that they are sharing resources and they are working on the same operating system can lead to dependencies in between them and thus cause latencies. That way other processes can cause latencies of a time-critical task. Seen from the task's point of view latencies arise accidentally and they are statistically distributed in time.

The architecture and synchronization mechanisms used in the operating system have a major impact on the number and endurance of such latencies. The analysis and improvement of architecture and synchronization primitives of a standard operating system regarding time-critical tasks is the starting point of this thesis.

### 1.2.2 Operating systems for different purposes

The duty of a standard operating system is to share the hardware resources available in a fair and efficient manner among the tasks that require them. What's important is that the standard desktop user is satisfied with the speed the system offers to solve the tasks, he wants it to be done. This is also called the *Quality of service* the operating system provides. Furthermore, users want to have a stable and long-lasting system providing the ability to integrate new computer peripherals like video cameras, audio systems, mass storages and fast network cards. Thus the operating system must be fully featured and open minded, i.e. offering many different interfaces.

For understanding the architecture of modern operating systems it's furthermore important to take into account the big differences in speed of the various devices that are part of a modern personal computer.

In the last decades it turned out that the central processing units, called CPUs, can today process data very fast and they need special infrastructures like cascaded caches to get the necessary data in time from relatively slow mass storage devices like RAM, ROM and hard disks via buses.

Because of that speed differences Linus Torvalds, the father of the Linux operating system, once said, a standard operating system has to keep the I/O systems going, because input and output periphery is still today the slowest part of desktop computers. The idea behind this sentence is, that the architecture of the operating system must be optimized in a way that the faster parts of the system always deliver data to the slower system parts as soon as they are able to process them. That way the overall performance of the computer system is the highest, as the bottlenecks are used in the most possible efficient way.

On the other hand a realtime operating system has to respect deadlines of tasks, which are for a time-critical task more important than a good over-all performance and a good utilization ratio of the whole system. A realtime operating system has to provide guarantees, that special time-critical tasks of high priority will be finished before a certain point in time. While a standard operating system has to optimize the cases occurring most frequently, the duty of a realtime operating system is to optimize the Worst Case in order to avoid huge latencies.

A standard operating system has to offer a generally efficient scheduler, while the scheduler of a realtime operating system must act predictable.

While a standard operating system should offer as many services as possible to its user, a realtime operating system offers only a few essential services with a reliable temporal behaviour.

The intention of realtime operating systems is to minimize latencies, while desktop operating systems maximize the average throughput of data.

Of course these two optimization goals are divergent, so one operating system can't serve both goals simultaneously as good as an operating system can do that is optimized to fulfill only one of these goals.

### 1.2.3 Possibilities to combine operating systems

For a long time developers have tried to run a realtime operating systems and a standard operating system on the same computer system.

For this approach the task to be done has to be split in a realtime-compliant part and a non-realtime part. Normally there are some ways to exchange data in between these operating systems running on the same computer.

This approach has the advantage that the deadlines of time-critical tasks that run on the realtime operating are always met, if the computer system is sized in an appropriate way. Moreover the standard operating system is full featured and can display data of the realtime components to the user or do for instance network communication to other computers.

The disadvantage is that the computer must have enough performance to run two operating systems at once and that both operating systems have to be configured and maintained by a skilled software developer. Often the realtime operating system also needs an extra license, that has to be payed. Thus this solution is hard-realtime capable, but it's a solution that applies only for special systems in industry or research. Such a concept will not be applied by a user at home. The sum of expenses and time necessary to use it doesn't differ much from using a fully featured realtime operating system. In conclusion this concept isn't analyzed in detail in this thesis.

The focus of this thesis is to find out how standard or off-the-shelf operating systems like for instance Windows, Linux or Java can be enhanced to accomplish time-critical tasks in sufficient time.

### 1.2.4 The approach of this thesis: Making standard operating systems more suitable for time critical tasks

This thesis analyzes how a standard or off-the-shelf operating system can be modified, so that it can meet the needs of time-critical tasks in a better way. The Linux operating system is used as an example hereafter.

There are two constraints to this thesis:

- As a reasonable constraint the interface of the operating system to the applications, i.e. the system call interface, shouldn't be changed or extended. This means all applications should be able to run on the modified and improved kernel as they did on the standard kernel without need of compilation.

- Another important constraint is that the performance of the standard kernel shouldn't suffer too much from the changes when the Linux kernel is made more realtime-compliant. That constraint is important, since many users wouldn't accept a decrease of performance of the standard Linux kernel.

First the capabilities and weaknesses of the Linux kernel regarding time-critical tasks have to be analyzed and reasons why latencies arise need to be found. This can be done by measurements. The results have to be analyzed and compared to results expected theoretically.

Then theoretical concepts to improve the temporal behaviour of the Linux kernel have to be found and implemented and the performance of such implementations must be measured and compared to that of the standard Linux kernel.

### 1.2.5 Realistic preview of possible results

For instance a good way to reduce latencies occurring in a standard operating system is to decouple its subsystems and to revise access limitations carefully.

Probably a standard operating system will not be hard realtime-compliant. Hard realtime systems are those systems where missing a deadline will lead to catastrophic consequences.

Realtime operating systems are carefully designed and time-consuming tests have to be done after every change or extension. This often implies that the code basis of the operating system can't be changed that fast and that not too many people and companies can easily write drivers for it, because they have to be tested not only to function in the right way but also to show an acceptable temporal behaviour, i.e. they mustn't introduce new latencies into the kernel.

Obviously rapid code changes and the possibility to integrate third party drivers are key features of standard operating systems. Furthermore such system haven't been designed to comply to realtime requirements originally, so they often contain long interrupt locks for example. Apart from that users of standard operating systems are not willing to loose too much performance only to comply to realtime requirements.

Therefore standard operating systems will only comply with soft-realtime requirements. But a soft-realtime compliance of standard operating systems will be sufficient for many applications as many users only want their personal computers to play music and video data providing an acceptable quality.

### 1.2.6 The choice of Linux as example

The Linux kernel has been chosen in this thesis, because its source is available under the GNU Public License, the GPL. Thus its code can be changed and results of this thesis can be published again under GPL, currently using version 2 of the GPL. In this way others can test the improvements and possibly use them. Furthermore many researchers and an increasing number of companies are using and supporting Linux. So, this thesis is done in an environment where others are interested in the work that is done. Thus it is possible to exchange not only ideas, but also implementations with other members of the Linux Open Source Community. That way Linux is the ideal platform for conducting experiments on an operating system.

Java would be a possibility too, but its license isn't that open and the preconditions of standard Java are a bit worse than those of Linux as for instance the Java garbage collector isn't realtime compliant. Therefore Linux has been chosen.

## 1.3 Preconditions: The hardware platform chosen

In this thesis only Intel compatible computers have been used. This is sufficient, since the main goal of this thesis is to find, compare, implement and verify concepts to make a standard operating system

like Linux more realtime-compliant. Since an implementation or a test of a concept on more than one platform would be very time-consuming, this hasn't been done in this thesis. Furthermore, for a proof of concept one platform is sufficient. Apart from that, the Intel platform is the most important one for Linux and other desktop operating systems and the biggest part of the Linux kernel source code is platform-independent, so at least parts of the changes done in this thesis will run on other platforms too. The non-platform independent parts could at least be ported, if another platform provides hardware devices satisfying the needs, e.g. the needed timing precision.

## 1.4 Outline

**Chapter 2** discusses the hardware model of personal computers and latencies that can arise from that structure. Furthermore, basic elements and concepts of operating systems are presented.

**Chapter 3** presents metrics to evaluate operating systems with regard to their performance and temporal determinism. Measurements methods and available hardware used for performance analysis in this thesis are introduced in this chapter.

**Chapter 4** outlines requirements to fulfill time-critical tasks. An overview over approaches to fulfill realtime requirements using personal computers running standard operating systems is presented.

Furthermore, sources of latencies in standard operating systems are discussed. Special architectures to avoid such latencies are shown. The chapter focuses on means to enhance the preemptability of an operating system.

Since in this thesis the Linux kernel is analyzed and enhanced to fulfill time-critical tasks in a more deterministic way, **Chapter 5** starts with an overview about problems of the standard Linux vanilla kernel regarding realtime requirements.

Since scheduling and priorities are one of the main issues of time-critical applications, the Linux scheduler, its development in recent years as well as interrupts and soft-interrupts are discussed in **Chapter 6**.

**Chapter 7** describes how the preemptability of the Linux kernel has been enhanced in recent years. Measurements are presented that show, how efficient the theoretical concepts to improve the preemptability of the Linux kernel discussed in chapter 4 have been implemented. Furthermore, measurements of a further improvement developed in this thesis are presented in this chapter.

**Chapter 8** describes the limits of the timing system of Linux 2.4 and 2.6 and an extension, called 'Rapid Reaction Linux', that allows precise timing in the range of tens of microseconds combined with a higher preemptability of the Linux kernel.

**Chapter 9** draws the conclusion from this thesis with regard to the development and ability of the Linux kernel in recent years and in future to handle a few time-critical tasks running together with many other non-time-critical tasks on the same computer system.

# Chapter 2

# Basic model of hardware and operating system used in modern desktop computers

## 2.1 Hardware model and arising latencies

In 1944 John v. Neumann presented the concept of a computer that stores the program code and the data to process in the same memory. His computer consists of the following elements:

- a central processing unit
- a memory
- input and output units
- and a bus, where the above named units exchange data

The v. Neumann computer has been designed to achieve a minimal hardware configuration. Due to technological development the following modifications to the v.Neumann architecture have been made meanwhile:

- In 1944 the CPU still needed more time to process data than to fetch new data from the memory using the bus. Due to new materials and techniques and the enormous speed-up of processors today memory access needs much more time than processing the already fetched data. To avoid that the CPU is idle most of the time, a hierarchy of small fast memory arrays - so-called `'caches'` - has been introduced in between the CPU and the main memory in order to hold often used data near the CPU, thus providing short access times.
- The CPU of the v.Neumann computer executed instructions strictly one after the other, without any parallelism. The need to fetch the next data while the CPU is still calculating led to the invention of pipelines. The pipeline of a modern CPU consists of several execution units working in parallel, the first one pre-fetching the next data from caches and main memory while the other units still process data.
- In the v.Neumann architecture the CPU controlled the bus, so if data had to be transferred from memory to a storage device via the bus, the CPU had to handle every data package. This method is called CPU I/O. On the other hand in modern computers data is transferred between memory and storage devices by Direct Memory Access (DMA) Controllers, so that the CPU is free to calculate while the transfer takes place.

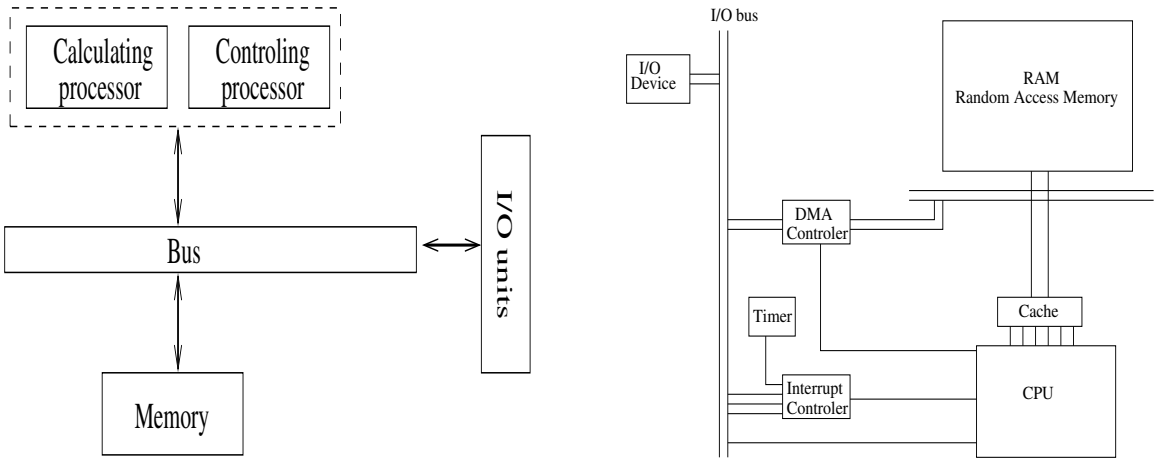



Figure 2.1: *Left part: Structure of a v.Neumann computer [62], right part: Structure of a today's PC*

### 2.1.1 Latencies caused by hardware

The execution time of a program on a PC normally varies a bit, each time it is executed. The reason for these differences are the current internal states of the different components of a computer system, that may require some extra work before the program can be executed or even during its execution. The large number of components in a modern computer causes these runtime differences. These components may act independently from each other; sometimes some of them are coupled in any way. Without a deep understanding and model of the particular computer and its tasks and without introducing constraints it can't be predicted exactly how much time the next run of the program will need.

Differences of the execution time of a program, if it runs several times on the same system, can be caused by elements of hardware and software. In the following subsections the temporal behaviour of some hardware components of a modern personal computer is analyzed.

The following hardware components can - due to their current internal state - cause latencies:

- Caches and RAM
- Pipelines and branch prediction
- Data transfer via a bus to external devices

### 2.1.2 The role of caches

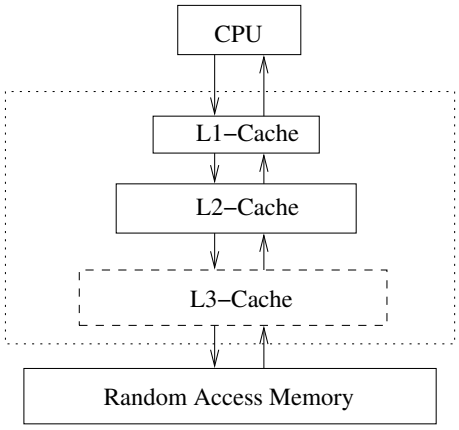



Figure 2.2: *Structure of caches in a modern PC*

Modern PCs include several cache levels: first, second and sometimes third level caches. The reason for this complex architecture is that processors have speeded up enormously, while the Random Access Memory, the RAM, used for the main memory of the PC, couldn't be accelerated that much. To avoid, that new fast processors are lacking data and thus can't work most of the time, small, fast static RAM chips (SRAM) or even on-die caches have been developed.

There is a cache hierarchy. Next to the processor, mostly on the same chip, there is the L1 cache which is divided into instruction and data cache. Further away from the processor there are the L2 and sometimes also a L3 cache.

Important parameters of caches are the dimension of their set, their cache line size and their associativity.

Different cache strategies to cope with incoherences between cache and memory have been developed.

Caches and the access to the memory are an important source of latency in every modern computer. The operation that causes the highest latency is the displacement of data out of the cache, especially if there is data that has to be written into the slower main memory.

### 2.1.3 Unpredictable execution time due to the super-pipelined architecture of modern processors

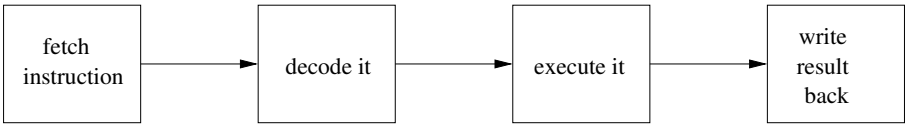



Figure 2.3: *Structure of a pipeline*

Modern processors use mechanisms to run different instructions in parallel in one CPU. To avoid that the CPU can't proceed because first new data has to be fetched from the caches or the main memory, a unit has been installed that only does fetching the instructions in advance. As can be seen in fig.2.3 another unit normally decodes the instructions, the next one executes them and finally the results have to be written back to memory. That's the basic structure of a pipeline. As the phases mentioned above are normally independent from each other these units work like a production line. After an instruction leaves one station it is processed by the next one. Ideally all units in a pipeline would need the same time to process an instruction. In reality a machine cycle is defined by the formula:

$$CPUcycle = \frac{1}{Max(\Delta t_1, ..., \Delta t_2)} \tag{2.1}$$

A CPU with pipeline doesn't execute a single instruction faster than a CPU without, but the number of instructions that are processed by the CPU in a certain time interval is increased by a pipeline as the number of wait cycles is reduced. The throughput is measured by the average number of CPU cycles that are statistically necessary per CPU instruction:

$$CPI_A = \frac{\sum_A CPUcycles}{\sum_A instructions} \tag{2.2}$$

When using pipelines the CPI-value can be decreased. Ideally processing an instruction would need on the average only 1 CPU cycle, but this value is not reached because of conflicts that can arise in the pipeline. These conflicts are explained hereafter.

### Pipeline hazards

Normally every instruction in the pipeline proceeds to the next station of the pipeline every CPU cycle. In certain situations it's possible that the advancement to the next station is not possible for a certain instruction. These disturbances are called '*hazards*' and they can lead to waiting cycles, so called '*stalls*'. During this stalls only some units of the pipeline can proceed in order to resolve the hazard, while the other units have to wait. According to different sorts of conflicts one distinguishes 3 different types of hazards:

- Structural hazards

  In order to make it possible to run all possible instructions in parallel many resources in the CPU and also the system bus and caches should be duplicated, which isn't possible for technical and economical reasons. Therefore there can be components that act as a bottle-neck when certain instructions run in parallel. A way to reduce structural hazards are dynamic scheduling mechanisms that reorder instructions in a way that they don't cause a hazard. Therefore a pool of instructions, that shall be executed by the CPU in near future, is formed. This out-of-order execution can avoid structural hazards in many cases but it also leads to the necessity to reorder the results after the calculations have been done, for instance using the Tomasulo algorithm.

- Data hazards

  Running instructions in parallel can lead to another sort of conflict if the next operation has to work with the result of its predecessor. Since in a pipeline unit 2 executes an instruction, e.g. performs the addition of two variables, while unit 1 already fetches the value of the variables out of cache or memory to prepare the next addition, there is no way to use the result of the addition as input for the next calculation as there is no cycle to write it back into cache or memory. A solution for such conflicts called '*data hazards*' is to install a logical bypass, so that such values can be exchanged in between pipeline units.

- Control hazards

  Such hazards occur when the normal flow of instructions is changed by a special instruction, for instance by a jump, by a branch, by a procedure call or a procedure return.

  Decision branches, i.e. if/then/else instructions, can lead to higher delays. Because of the binary decision there are two directions or branches the instruction flow can continue. Whether the branch is taken or not is clear only when the condition has been evaluated. If for instance the decision has been taken to execute the branch and the pipeline units behind already fetched and decoded the instructions for the non-taken branch case, the early stages of the pipeline have to be erased and the right ones have to be fetched and decoded. This will cause a delay. There are different ways that try to minimize this delay. If the compiler is well adapted to the architecture of the CPU it can execute instructions that are independent from the condition until it is evaluated. This method is called '*Delayed branch*'. Another possibility is to execute *No operation* instructions until the condition has been evaluated. A further mechanism is to try a branch prediction. A branch prediction algorithm tries to predict whether the 'if' or the 'else' branch will be executed. Based on this decision it starts fetching the next instructions into the pipeline in order to avoid a possible pipeline stall after evaluating the condition. Doing this a latency only occurs if the condition is wrongly predicted.

### Super-pipelining

In order to decrease the time a CPU cycle takes, processor manufacturers increase the number of units in a pipeline, cp. eq.(2.1). That way every unit of the pipeline finishes faster because its task to fulfill is smaller, but at the same time the complexity and the number of hazards rises. Therefore every processor architecture has an optimum number of pipeline units, where the pipeline reaches its optimal performance, as fig.2.4 shows.

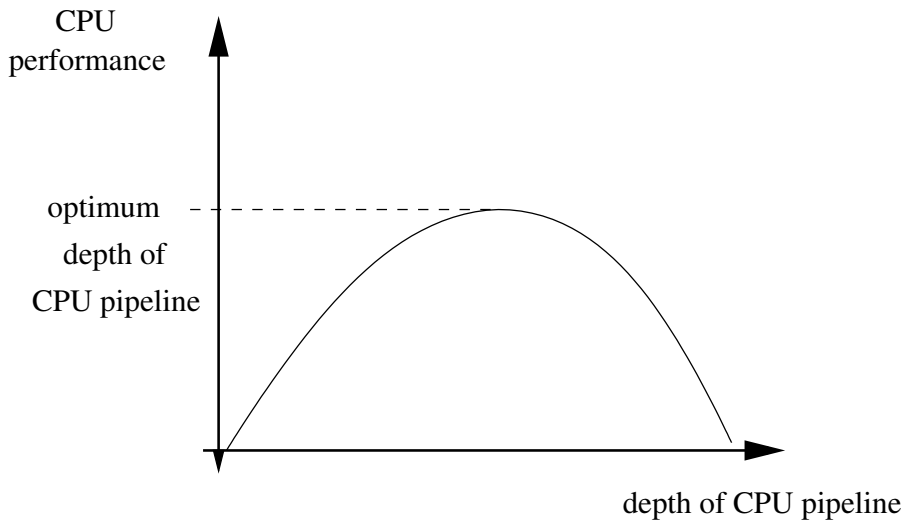



Figure 2.4: *Performance of a pipeline depending on the number of stages it provides*

**Super-scalar architectures**

In order to increase the CPU performance, processor manufacturers implemented in modern CPUs multiple pipelines that are working in parallel. In some processor designs the pipeline share special functional units, so that it is impossible for special combinations of instructions to run in parallel through the pipelines. To avoid such situations and pipeline stalls a technique so-called '*scoreboard*' has been introduced, that forms an instruction pool and thus allows the out of order execution of instructions and pairs and reorders instructions in a way that no hazards occur and pipeline stalls are avoided. After the execution, a functional unit, called '*retirement*', reorders the results the instructions produced.

Latencies caused by the Pipeline and Branch Prediction errors are less important, but they get more important for computers with longer pipeline, e.g. the Pentium 4.

### 2.1.4 Unpredictable execution time due to data buses and external devices

There are also other hardware components that can induce latencies, e.g.

- The connection CPU-chip set to RAM (Host bridge)
- Communication over a bus, e.g. the PCI bus
- Access on storage devices, e.g. a hard disk, or exchanging traffic with a network interface card

The PCI-bus can be referred to as the standard bus of today's personal computers. It is used on many different platforms. Some of its features are:

- High data throughput due to a data bus width of 32 or 64 bit and a frequency of 33 or 66 MHz as well as burst transfers
- Assignment of resources without conflict
- Multi master capability
- Standardized configuration interface with special registers for information and configuration
- PCI devices can use Direct Memory Access, DMA, to transfer data from a device directly into the main memory.

A PCI cycle consists of an address and several data phases. During such a PCI-cycle the CPU can't access the main memory of the computer. Furthermore, the PCI-bus has to be arbitrated before every access. All this can lead to latencies caused by the PCI-bus.

Thus data transfer on the PCI-bus can block CPU activities and PCI-bus activities can even lead to a jitter of the displacement times for cached data.

It's of great interest, how long this latencies can be. In [3] measurements are presented that show that the access of RAM on a personal computer without PCI load needed in average about 187 microseconds with a jitter of 12 microseconds. The same memory access with PCI load in parallel increased to 217 microseconds with a jitter of 30 microseconds. From that measurement one can see that PCI bus activities can delay the execution time of programs by blocking the CPU's memory access for short periods.

### 2.1.5 Summary of hardware caused latencies

For a detailed view of hardware caused latencies the processor architecture, the cache hierarchy and the memory access has to be taken into account as well as the influence of external buses like the PCI-bus.

Latencies can also occur because of a wrong branch prediction or data hazards. Compared to latencies caused by caches these are shorter.

## 2.2 Basic elements and mechanisms of modern operating systems

### 2.2.1 A model of operating system

An operating system can be seen as a collection of programs that act as a mediator between the hardware of the computer and the applications running on top.

Although it is possible that a task operates directly on the hardware without the interference of an operating system, this is nearly never done because of the major advantages an operating system running on a computer provides.

There are two ways to look on it:

From the point of view of the user applications, the operating system offers system calls and applications interfaces, that an application must use if it wants to obtain certain well specified services from the operating system. The advantage of these interfaces is that they partly hide the complexity of the hardware and hardware peripherals used in the computer system. Furthermore, the operating system provides synchronization mechanisms that allow different applications to share the hardware components of the computer. Synchronization objects limit the access of the applications to the hardware in order to avoid access conflicts.

To fulfill this task the operating system normally has the privilege to access the hardware components while user applications can do that only by using the system calls the operating system provides to them.

Some of the main duties of an operating system are listed below. At the same time these are the functional subsystems operating systems consist of.

- Scheduling and dispatching tasks that are running in parallel on the computer
- Memory management
- Timing services
- Synchronization and intertask communication
- Handling of events like interrupts and signals

- Input and output operations
- Administration of file systems, storage devices and other peripherals

Modern operating systems are Multitasking systems. This means the processor is shared between several applications that all get the processor for short periods of time successively. These periods are not long enough to finish the applications. As the applications are switched very fast on the processor, a human being gets the impression that they are running in parallel, although they are on an uniprocessor system.

An operating system has many different hardware and software resources to administrate and share among user processes:

- CPU
- Memory
- Input/Output devices and other peripherals
- Synchronization and intertask communication primitives
- Events like interrupts, signals and timers

Operating systems also can be classified according to their purpose. Regarding the purpose there are standard operating systems for personal computers like Windows. For special purposes, special operating systems have been developed, e.g. Sun Solaris -and Linux as well- is often used on server systems, and there are also operating systems designed for telecommunications or realtime purposes. A trend of the recent years is that standard operating systems are enhanced more and more to be able to satisfy even special purposes. E.g. a server version of Windows operating system is provided, and furthermore there is an embedded branch of Windows, called Windows CE. Also Linux - formerly primarily used as a server system - is used as a desktop operating system today and can be used even for embedded systems [93].

There are several advantages of this policy taking off-the-shelf operating systems for special purposes:

- An off-the-shelf operating system is used by many persons and thus has been tested in detail by many people using different applications. On the contrary a system that has been developed only for one purpose is used by only a few users and thus is often less reliable.
- Since so many people use standard operating systems, drivers and extensions for new hardware are early available for such systems. On a proprietary system, the costs for maintaining the system and adapting it to new technologies has to be payed by a small user group, making it more expensive.

Operating systems can also be distinguished with regard to their structure. Most operating systems are so-called monolithic. This means their structure is not divided into independent modules. Nevertheless they are often internally divided into different subsystems, like the timing subsystem, file systems, etc. An advantage of the monolithic approach is that it allows to use tricky optimization techniques even in between the different subsystems in order to provide a better performance. The tradeoff is that the code becomes more complex and thus is more difficult to maintain and to improve.

On the other hand there are operating systems called micro-kernels. These micro-kernels only provide a core functionality in the kernel like intertask communication, synchronization primitives and task management including the context switch and all privileged operations. Many other parts of the operating system like memory management, hard disk drivers, file systems and even the scheduler are put into tasks, the so-called servers. The advantage is that in a micro-kernel those parts of the operating system that are tasks can fail, without compromising or put at risk the whole kernel. An example of a micro-kernel is the QNX operating system.

On the contrary in a monolithic operating system a failing component, like a driver for example, nearly always leads to a crash of the whole kernel.

Since monolithic operating systems are also divided into subsystems there are also hybrid forms in between these two types. E.g. the kernel modules of the Linux kernel that can be linked and unlinked to a kernel at runtime are at least a step towards a micro-kernel as they are connected to the kernel using a well defined interface and they can be even stacked.

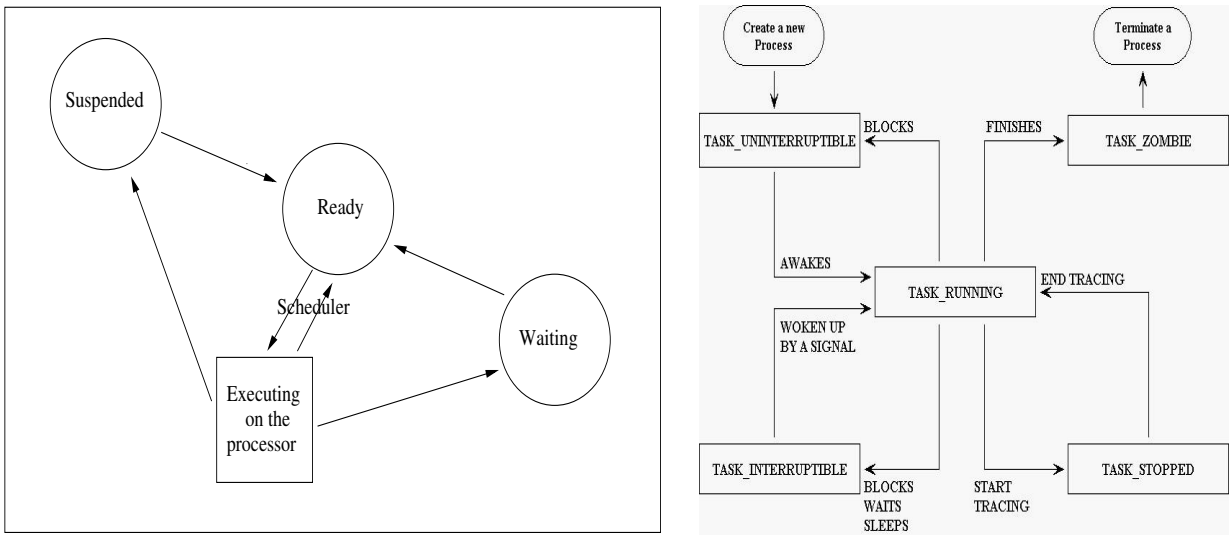



Figure 2.5: *The left part of this figure shows the task state diagram of a general operating system, while the right part shows the task state diagram of Linux 2.4.*

### 2.2.2 The Tasks - the operating units of a system

Simple operating system on small embedded devices often only have one operational unit, called a task, that performs operations.

Modern Operating systems support *multitasking*, i.e. the operating system administrates several tasks that are working independently from each other.

Every task is executing its own code and manages its own resources. Several states can be assigned to a task forming an extended finite state machine.

The left side of fig.2.5 shows the possible states of a task in a general operating system together with the possible transitions between states. The right side shows the extended finite state machine of the Linux kernel.

- **Running**: If a task is currently executing on the processor, its state is *Running*. But since there are several tasks in a multitasking system the processor is shared between these tasks and thus there are always tasks that are currently in one of the other states, *Ready*, *Suspended* or *Waiting*.
- **Ready**: The state *Ready* means that the task is currently waiting for the processor; it is ready to run. Normally such tasks are stored in a waiting queue. The subsystem which decides which task is brought to execution next is called the *scheduler*.
- **Waiting**: The state *Waiting* means that a task is currently waiting for another resource than the CPU. On a personal computer there are many different resources that can be assigned to a process exclusively and thus other processes have to wait until they are freed. Such resources can be peripheral devices, lists of the operating system, synchronization objects or data objects that are shared in between different tasks.
- **Suspended**: The state *Suspended* means that a task is currently neither waiting for the processor nor for another resource, it has been put asleep and will only run again, if it is reawakened by a signal or an interrupt.

### 2.2.3 The concept of virtual memory

The hardware architecture of modern personal computers supports the concept of virtual memory. Virtual memory allows to assign a linear address space exclusively to every task. A task together with its address space is often called a process. This concept has several advantages:

- Address spaces allow a task to be programmed and executed as if it was the only task on the system. Thus a task possessing its own address space is independent from other tasks.
- Since the hardware and the operating system provide a memory protection mechanism, no process can read or write the memory of other processes. Thus different processes are isolated and protected from each other.
- A failure of one process, e.g. a division by zero, normally won't affect other processes except if they have been bound to common resources.

Modern standard operating systems like Windows and Linux normally only work on hardware platforms, that support virtual memory and memory protection by providing a MMU, a memory management unit, that performs the conversion of virtual memory addresses into physical memory addresses.

Today only small embedded devices that contain microcontrollers instead of full featured processors don't provide a MMU. Thus standard operating systems have been ported to such systems, forming Windows CE or $\mu$CLinux, called Microcontroller Linux. Since memory protection doesn't work there, the tasks are sharing the same address space on such systems.

### 2.2.4 Scheduling and context switch

On a multitasking operating system like Linux or Windows, the processor of the computer has to be shared between different processes, that want to execute their code on the processor.

One processor can only execute one process at a time. If more than one process shall run on one processor, this is only possible by a time multiplexing of the processes on the processor.

This means every process gets the processor for a certain period of time. How long this period is and in which sequence the processes are enabled to run, is subject to the scheduling algorithm, the multitasking operating system uses.

Many scheduling algorithms have been presented in literature, and some of them are implemented in all operating systems.

It depends on the purpose of the system which scheduling algorithm is the right one. So in general, real time systems provide other scheduling algorithms than standard operating systems.

Scheduling algorithms like 'explicit plan' or 'rate monotonic priority assignment' are appropriate to realtime systems, which have to perform periodic tasks. Explicit plan means that every task is scheduled at explicit points in time. Rate monotonic scheduling means that the tasks with the shortest period gets the highest priority in the system. Although this scheduling algorithm is not optimal, it is widely used in commercial products. This is because it's an efficient scheduling algorithm, that is easy to implement.

In general one distinguishes preemptive and non-preemptive scheduling, also called cooperative scheduling. The notion of *preemptability* is discussed in sections 4.4, 4.5 and chapter 7.

In the majority of standard operating systems, priority based schedulers are used. This scheduling algorithm finds out the process of highest priority and makes a context switch to execute it next on the processor. In all these systems the scheduling algorithm or strategy deals with how to assign priorities to the tasks either statically or dynamically.

Therefore between 2 tasks running on the processor always the scheduler has to be invoked. Its goal is to find out which task should run next. When it found out the process to run next, the 'context' of the process that run before has to be saved.

In a context switch between two tasks several actions have to be done:

- The processor registers of the process that ran before have to be saved
- A context switch to the address space of the new process has to be done.
- The processor registers of the new process have to be loaded
- The Translation-Lookaside Buffer, the TLB, that provides fast access to some memory pages of the process running before, has to be invalidated.

### 2.2.5 A task - a process or a thread

At first, multitasking operating systems only contained processes. Later on the concept of *threads* has been invented. A *thread*, also called light-weighted task, is a unit that operates in the address space of a process. The execution unit of the object formerly called 'process' is now called 'main thread' or first thread of the process. Every thread executes its own code. Therefore every thread has its own registers and its own stack. But it works in the address space of a process and thus has access to the data of the process.

This concept has several consequences:

- When creating a new process in Unix, the content of the address space of the old process is partly copied into the address space of the new process. Therefore setting up a new process needs a lot of time in Unix compared to the time needed by other system calls.
- Using threads more than one task can work on the same data. That's possible also using processes but more complicated since to work efficiently on a big amount of data a common address space, called shared memory, needs to be declared. Of course, synchronization mechanisms are used when accessing common data.
- Using threads it can be avoided that a process performs a blocking operation and is suspended. When using threads only one thread is suspended while other threads can continue working on the same data of the process.
- Since the address space isn't switched between different threads of the same process, threads can be scheduled a little faster than processes. But the main advantage regarding performance is that the TLB isn't invalidated in between the context switch of threads of the same process. This allows all threads to access memory pages faster and increases the overall performance of the computer system.

### 2.2.6 Kernel and User space

Intel compatible personal computers of today provide different levels of access to hardware, so called rings, see fig.2.6.

Such rings are levels with different privileges, i.e. different access rights to the hardware. At Ring 0 all hardware peripherals are accessible. On the other hand, at ring 3 normally some special assembly commands that access ports or directly manipulate peripheral devices are forbidden.

These hardware mechanisms of Intel compatible personal computers are used by operating system programmers to force user applications to use routines and drivers of the underlying operating system to access the hardware. That way the operating system can administrate the hardware and protect it from wrong commands that could damage it.

User applications work in ring or processor mode 3, while the kernel, i.e. the operating system, work in ring 0, with full access to the hardware.

Several hardware platforms support this separation between the operating system and the applications - the user space programs, as they are called.

Another important feature to separate and protect applications from the kernel and vice versa is the use of separate virtual address spaces for the operating system - the kernel - and the user space programs - the applications - which is common in most modern operating systems like Windows and Linux.

Fig. 2.7 shows that the operating system possesses its own virtual kernel address space. Every user process works on its own virtual address space, too.

This concept enables to protect the processes one from the other. Furthermore, also the kernel address space is protected from being accessed by processes.

Using this memory protection concept, all processes are confined to their own address spaces. Operating systems that strictly use this concept are more stable and less vulnerable to errors in software, since a faulty application doesn't affect other processes or the kernel.

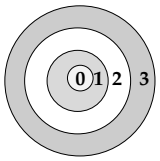



Figure 2.6: *Rings, i.e. modes of an Intel processor with different levels of privileges*

### 2.2.7 System calls

Since applications cannot access hardware devices directly they need services of the operating system to do this.

Every operating system offers an interface with a limited number of system calls, that can be invoked by user programs providing a parameter list. This list of system calls should be stable, as all libraries and user applications rely on that interface. Sometimes a new version of an operating system extends the list of system calls, but doesn't change system calls introduced in previous versions.

All kernel functions, that aren't part of the system call interface are internal, thus not available to application programmers. They can be changed without affecting the interface to the user applications. This approach has been chosen in this thesis.

On an Intel compatible personal computer a system call means that the processor also changes from user mode to kernel mode, because system calls are executed in the protected kernel mode. To do this transition the soft interrupt 0x80 has to be triggered on an Intel platform. Section 2.2.8 discusses the concept of interrupts.

Applications can call system functions directly or they can call them via routines in libraries that wrap the system calls in order to provide new functionality or only to enhance compatibility to other interfaces or standards.

Linux as a standard operating system has about 200 different system calls. They are expanded slightly in every major kernel release.

### 2.2.8 Interrupts

Electrically, an interrupt is just a transition from logical low to logical high on a special pin of the processor.

It instructs the processor to stop the normal flow of operations of the current working task and brings a part of code called interrupt service routine (ISR) to execution on the processor.

- Synchronous interrupts are often called exceptions or soft interrupts. They are generated in between processor instructions by the CPU itself, if the CPU can't continue the normal flow of operations, e.g. because a division by zero encountered or an unknown instruction has been loaded.
- Asynchronous interrupts are generated by external peripheral devices. They are used to inform the processor about events that took place in the external world outside the processor.

For time-critical tasks asynchronous interrupts are of special interest.

The first reaction of the operating system to the hardware interrupt is the interrupt service routine (ISR), also often called interrupt handler. The interrupt service routine (ISR) is not a process or a thread. It belongs to a category of code - often called a *tasklet* - which is executed on a processor without a task context in kernel mode. This means that interrupt service routines execute in kernel address space.

Since the ISR executes as a *tasklet* without a task context, the finite state model of tasks presented in section 2.2.2 does not apply to ISRs.

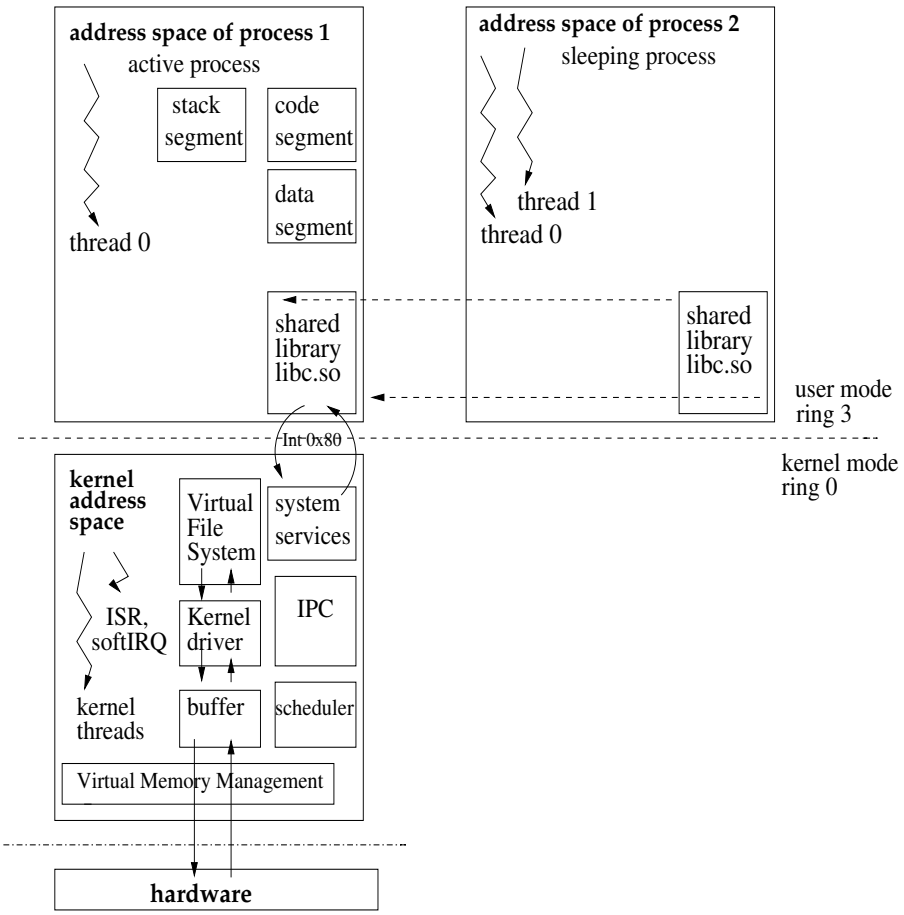



Figure 2.7: *Address spaces of the kernel and different processes in an operating system*

Therefore in a 'tasklet' programmers are not allowed to perform operations that would bring a interrupt service routine into the state *Waiting*. For an ISR it is forbidden to wait or sleep and an interrupt service routine is not scheduled.

The interrupt service routine (ISR) is often also referred to as top half of the interrupt handler. An ISR should be finished fast, it should not use much time, because its priority is that high, that they intercept even important processes of high priority.

Normally peripheral devices like the hard disk controller announce via an interrupt that they finished performing an action that a driver has initiated some time before.

### Bottom halves and other tasklets

Often there are actions to be done some time after an interrupt occurred or after a driver has required or initiated an operation. A typical example is the floppy drive or any other peripheral device, which doesn't send an interrupt when finishing a work order. Thus a driver has to poll periodically, whether the device already finished the initiated operation.

This can be done by any routine operating asynchronously. Such routines are called tasklets. In the Windows family of operating systems they are called *deferred procedure calls* (DPC). In Linux 2.2 they are called *bottom halves*, afterwards *SoftIRQs*, which means soft interrupts. These tasklets execute in kernel mode and space. This means they can't be intercepted by any process or thread. They are treated similar to interrupt service routines by the kernel.

As the top half is the first reaction of the operating system executed normally soon after the interrupt signal reached the processor, a bottom half can execute a part of code, which has to be executed later than the original ISR.

To give an example: The network interface card sends an interrupt to the CPU in order to announce that new network packets have been received. The interrupt service routine of the Linux network driver first copies the octets, i.e. the bytes, from the network interface card into the memory of the computer. Further processing on them is done in a bottom half or SoftIRQ later on.

#### Kernel threads

Kernel threads are threads, that operate in kernel space. That way they can access all internal kernel variables. In Linux they are scheduled like the other tasks, as the Linux kernel supports threads in kernel space and in user space as well.

Kernel threads are often used to do important tasks in an operating system asynchronously, e.g. a kernel thread called *ksoftirqCPUID* executes activated SoftIRQs since Linux kernel version 2.4.10. In previous versions all SoftIRQs were called Bottom Halves and executed every time the scheduler started.

#### The hierarchy of hardware and software priorities

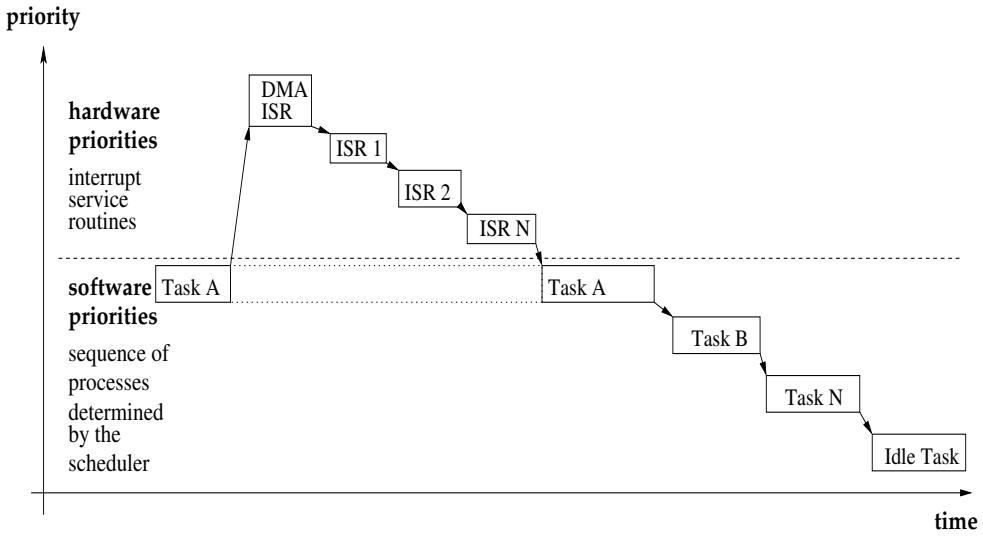



Figure 2.8: *The hierarchy of hardware- and software-priorities*

Fig.2.8 shows hard and software priorities in one priority scheme. In general hardware priorities are more important than software priorities. So processes or threads like 'Task A' or 'Task B' are interrupted by any Interrupt Service Routine (ISR) that reacts to an incoming interrupt. The process of the lowest priority normally is the idle process that executes on the processor, if no other task wants to run there.

The notions of *preemptability* and *locks* are discussed in chapter 4.

## 2.3 The POSIX standard

POSIX is an acronym for 'Portable Operating System Interface'. This standard has been developed by IEEE, and standardized by ANSI and ISO.

The POSIX standard is an effort to normalize the Application Interface (API), the libraries of an operating system could offer. The POSIX standard itself is structured and split into substandards, called POSIX option groups, serving different purposes.

These standards have two effects:

- At first, the functions defined in a POSIX standard can be used as a guideline of best practice when developing operating systems or libraries, because they have been carefully chosen and designed. If all the functions of a POSIX Option Group are implemented in an operating system, a powerful API is at the application programmer's disposal.
- Secondly, POSIX permits developers to port applications easily to other platforms. Ideally, when a platform complies to a POSIX standard or option group, a POSIX-conform application has only to be recompiled in order to run on the new platform, since the libraries of the new platform provide the same POSIX API calls. But as most operating systems only implement a subset of the POSIX option groups, often the source code of the program has to be slightly changed until it runs on the new platform and provides the same functionality and runtime behaviour.

The POSIX option groups are mostly implemented in the libraries of the operating system. But some of them need support by the kernel. A minimal set of POSIX interfaces and functionality is required for all POSIX systems. But the majority of the proposed functions is optional. In C programs, certain POSIX macros are defined after the `'<unistd.h>'` is included in order to signal that the functions of this POSIX option group are available.

The POSIX conformity of the Linux kernel is discussed in detail in section 5.2.

# Chapter 3

# Metrics for evaluating operating systems

## 3.1 Fundamentals of computer performance analysis

The notion of performance analysis covers the fields of measurement, modeling, evaluation and optimization of computer systems [17]. The aim of performance analysis is to investigate the dependencies between influencing variables and performance characteristics of a computer system. Performance analysis is an important part of modern computer design, since the complexity of computer systems increases and the structure of hardware and software and their interaction becomes more difficult to overview.

Performance isn't used in terms of the physical magnitude, but in terms of the potential of a computer system, that consists of many inhomogeneous components, to fulfill its purpose. The metric chosen to evaluate performance always depends on the purpose the system is used for. For instance an interesting parameter to evaluate the CPU performance is the duration of a CPU cycle. Generally performance characteristics or metrics permit a qualitative or quantitative evaluation of the performance of a computer system.

For computer systems that perform time-critical tasks performance analysis is very important since performance characteristics are an important factor when choosing such a system to perform a certain task.

For time-critical tasks a user is mainly interested when the computer system finishes the task the user has requested it to do, this duration is called the *response time* $t_R$ of the system.

According to Mächtel [58], the *response time* is defined as the duration between the request and the completion of the task requested:

$$t_R = t_V + t_W + \sum_{i=1}^{n} t_L(i) \tag{3.1}$$

The *response time* consists of the time $t_V$ the CPU needs to complete the requested task itself, the time to wait $t_W$ and the sum of the latencies $t_L$. The term $t_V$ contains all the time that is needed to process the task assigned, regardless of the time that is needed in user mode to calculate or the time to execute a system call of the operating system in system mode. The waiting time $t_W$ consists of the time $t_U$ the requested task can't be executed, because the processor executes tasks of a higher priority, and the time $t_B$ the task has to wait until resources, that are currently used by other tasks, are freed.

If further delays occur because of the internal state of the system or because this state has to be changed these delays are called latencies $t_L$. A better definition for the notion of latency is given in section 3.2.1.

## 3.2 Real-time - definitions and explanations

In this section different aspects of realtime are presented and looked at in detail. First the metrics that are important for realtime are given.

### 3.2.1 Timeliness

Applications or programs are usually judged whether they fulfill their purpose. Normally this means they must execute logically correct algorithms that lead to a correct result.

Apart from working logically correct realtime tasks also have to satisfy timing requirements, i.e. to present their correct result within a timing interval, that is already known before the realtime task starts.

When a realtime task presents the correct result after a timing constraint, called a deadline, has been missed, the realtime task failed, although it has calculated a correct result.

#### Hard Realtime and Soft Realtime Tasks

A further important distinction between tasks with hard realtime requirements and soft-realtime requirements has to be made.

The difference can be shown regarding the benefit function in fig.3.1. The benefit function expresses the benefit the user has if an application fulfills its task in a certain time interval.

Hard realtime applications have to fulfill their task before a definite point in time, for such a task it is forbidden to miss a deadline. Their maximal response time $t_R$, see eq.(3.1), is well known and guaranteed. On the other hand a hard realtime task is not allowed to present their result too early either, so there is a well defined interval, the correct result has to be presented in.

In a hard realtime system the benefit the user gets when the task is performed outside of the interval $[t_{R,min}, t_{R,max}]$ is zero or even negative, i.e. a damage occurs that is not tolerable, for instance if human life is put at risk. So the benefit function of a hard realtime task has the shape of a rectangle.

The most striking example for this kind of task is the task to moderate an nuclear reactor or to pilot a pursuit planes, which fly in an instable aerodynamic balance. In the first case the reactor core will melt and cause possibly a big atomic disaster, if the moderation rods are not brought into the nuclear reactor within a very short timing interval, in the other case the aeroplane will simply fall down onto the bottom and crash, if the piloting system fails to react in time.

This means hard realtime tasks do not tolerate a single missed deadline. If a deadline in a hard realtime system is missed, a disaster is often inevitable.

On the other hand soft-realtime applications mean that the application has to fulfill its timing requirements only statistically. In a timing interval a response time can be higher than it is normally acceptable to fulfill the goal of the application, but a soft-realtime task still guarantees that a certain percentage rate of the task repeated periodically will fall into a certain timing interval. The difference in comparison to a hard-realtime task is that little deviations of the optimum response time reduce the user's benefit but don't lead to a disaster. This is expressed by a benefit function that slowly approaches zero. Nevertheless often there is a limit for the response time where the benefit function suddenly drops to zero since the benefit is not acceptable any more for the user at that point.

A well known example for soft-realtime are tasks to play video or audio data. A video has to be replayed with a certain frame rate. If one frame isn't replayed totally in the appropriate time, it is dropped and the user recognizes this, but normally the next frame will be displayed in the right manner and the user accepts this small loss of quality. Of course, if more and more video frames are not displayed the loss of quality of the replay is considerable, so that the user will decide to stop the video. To avoid this the video replay task has to guarantee that it will display a certain percentage, e.g. 98 %, of the frames at the right time, which will satisfy the user's quality requirements. A single frame that is displayed too late or even dropped won't produce disastrous results like in the hard realtime case.

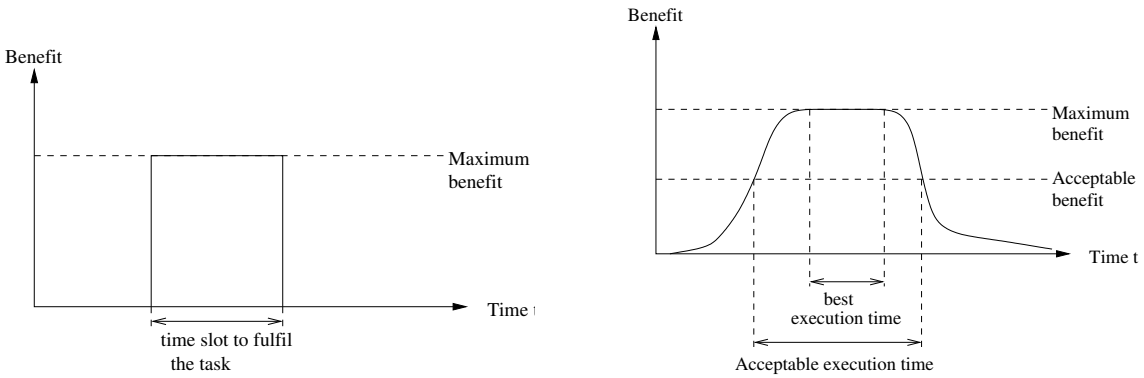



Figure 3.1: Benefit functions of hard and soft realtime systems

**The notion of 'latency'**

Unexpected delays of a task are called latencies, see eq.(3.1), p.31. They are caused by internal states of the operating system or hardware, that isn't known to the user [58]. Latencies arise statistically and often a deep knowledge and understanding of the internal architecture of the system and its possibly parallel operations are required to figure out their reasons.

For realtime systems which shall have a deterministic behaviour it is important to know the maximum values of such latencies.

These maximum values of latencies show up as rare outlier of a test series. Often they are not shown when the result of a benchmark value, see section 3.6.1, is presented.

In practice often the execution time of required tasks is measured and from these values - multiplied with a security factor bigger than 1 - one estimates how powerful a system must be to fulfill the required tasks without violating any deadline.

A further problem when doing such estimations is that there can be even longer latencies than the outlier of a given test series show, which only occur in an longer or infinite test series.

So to estimate the maximum values of a special latency or to set up a measurement situation in order to find out the maximum duration of a special latency in the worst case often a deep analysis of the system is helpful.

A whole discipline, so-called Worst Case Execution Time analysis - or WCET analysis in short -, researches worst case scenarios where several latencies lead to a longer one. The difficulty is to know in which way such latencies can be combined in a realistic way, without making too negative predictions. This is discussed in section 3.3.

Since some well known latencies serve as important performance indicators for operating systems they are defined in the next paragraph.

### 3.2.2 Responsiveness

Responsiveness means the ability of a system to react to events that took place outside the computer system in the real world. A realtime system is normally used to react in due time to technical processes, that act in the real world. Such technical processes follow the laws of nature and deliver data at special points in time to the computer system regardless of the current internal state of the computer system.

**Getting pieces of information from the outer world**

There are different mechanisms by which a computer system is informed about events in the outer world. One possibility is that there is a task that is querying periodically for new data available at a peripherical device.

This is rather time-consuming, thus for all cases, when it isn't known when a new value will be available, the use of the interrupt mechanism is more effective.

#### Interrupts - Fast reactions to signals of the outer world

Interrupts serve to inform a system about changes in the outer world. That's why many peripheral devices like the mouse, keyboard and the network interface card send interrupts to the CPU, cp. section 2.2.8, p.27.

Since changes of the outer world can be of high importance for the system the normal flow of operation of the central processing unit (CPU) is interrupted by an interrupt and after a short time as a first reaction to the interrupt a short piece of code is executed. This code is called interrupt handler or interrupt service routine (ISR). Most operating system execute this code in kernel space, so an interrupt service routine is never allowed to be put asleep, contrary to a task.

Since the ISR interrupts even tasks of high priority currently running on the CPU, their execution time shall be short and normally they awake a task, that has been blocked before waiting for the interrupt.

This task is called the second reaction to the interrupt. It can execute more complex operations in reaction to the interrupt and since it has its own memory context, it can manage more data in a secure way. Furthermore such a task is allowed to operate for longer times and is also allowed to perform blocking operations.

In the following paragraphs simple metrics are presented that characterize the responsiveness of an operating system.

#### IRT - Interrupt Response Time

The interrupt response time is the time from the point of time an external device generated an interrupt until the execution of the first instruction of the interrupt service routine.

The interrupt response time consists of several latencies of different origin:

- the very short time while the interrupt signal runs through the wires
- interrupts may be locked for some time by the operating system
- possibly other interrupt service routines have to be finished
- the microcode cycle currently executed by the processor has to be terminated, the rest of the instructions in the processor pipeline will be discarded
- saving of the context of the current process and the partial context switch to the ISR

#### PDLT - Process Dispatch Latency Time

The PDLT is an important measure to characterize the responsiveness of a system. It's the time between the occurrence of an interrupt and the first command of a process that has been awakened by the interrupt service routine.

The different latencies forming the PDLT are of three origins: the hardware, the operating system, and especially the duration of the Interrupt Service Routine depends on the used driver.

Later on, in section 7.2, a formula for the PDLT of the Linux kernel as an example for a standard operating system will be given.

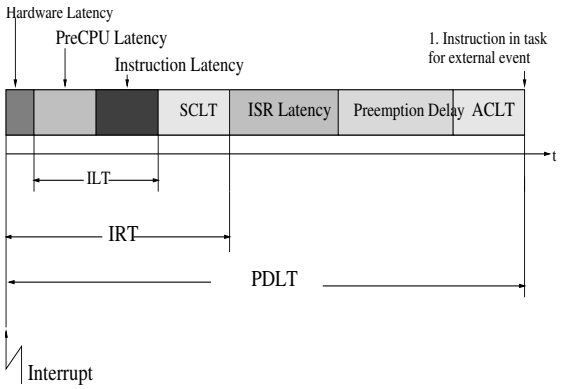



Figure 3.2: The PDLT contains several latencies

### 3.2.3 Concurrency and utilization ratio

Computer systems of today are able to run multiple tasks in parallel. A possibility to do this is to install multiple processors, but that's rather expensive and inflexible, since the number of processors can't be adapted to the number of tasks.

Thus the capability of multiplexing several tasks normally is part of the operating system, i.e. of the software. This is reasonable, since modern processors can execute multiple tasks in a very fast way using time-sharing mechanisms. In case the processor controls a peripheral device, e.g. a printer, the processor can execute a lot of tasks, before it has to interact with the peripheral device again, since the clock rate of modern processors is very high, compared to the timing intervals needed to control a peripheral device.

Nevertheless, also the fastest processor has a limit of tasks it can execute during a certain time span. For $N$ periodical tasks with period $t_{p,i}$ and execution time $t_{ex,i}$ the following formula must be satisfied in order not to overload the system, i.e. have an utilization ratio less than 1:

$$utilisation \quad ratio = \sum_{i=1}^{N} \frac{t_{ex,i}}{t_{p,i}} \leq 1 \tag{3.2}$$

This is only a necessary, not a sufficient condition that the tasks 1 to $N$ can be scheduled by an operating system. Section 3.3 'schedulability' discusses sufficient conditions, see p.35.

## 3.3 Schedulability and WCET-analysis

Worst-case execution time analysis (WCET analysis) form - together with schedulability analysis the basis for establishing confidence into the timely operation of a real-time operating system [76].

The WCET analysis aims at computing upper bounds for the execution times of tasks in a given target system. Using these bounds an allocation of CPU time for the tasks can be made.

Based on this data, methods and tools can be used to analyze the schedulability of a set of tasks on a given target system. The schedulability analysis reveals whether a set of tasks is schedulable and will meet the timing requirements on a given system.

## 3.4 Performance analysis

On the first level measurement methods and techniques to create models of the object to analyze have to be distinguished. In the category of measurement, benchmarks and monitors have to be distinguished. On the other hand modeling techniques and simulation are used to build up a model of a computer system in order to make theoretical forecasts about the behaviour of a system.

The following paragraphs examine the different approaches more closely.

## 3.5 Modeling

To optimize a complex piece of software like an operating system with regard to some special qualities like performance or timeliness one needs a set of measurement procedures that provide the necessary insight into internal operations of the system and a model to describe the system. Different techniques of modeling can be distinguished. *Structural models* describe the active and passive components of a system and their interfaces and interactions. Another way of characterizing a system is to describe it using a *behavioural model*. In such a model the computer system is seen as a black box, i.e. the internal structure of the system doesn't really matter. The system is characterized by the reactions it provides at the outputs when certain input signals have been put in and the system has been in a certain internal state before depending on its history. The temporal behaviour of computer systems is described best by a discrete event model.

#### Analytical modeling

In analytical modeling different methods are distinguished:

In *deterministic analysis* the system parameters have well-defined values at every point in time since they are described by deterministic processes and equations. Since the system parameters of real systems described by the equations are often statistically distributed, the values that are put into the equations often are mean or extreme values.

In *stochastic analysis* the behaviour of the system is described by stochastic processes. The system parameters are described by random variables which often have a statistical distribution. The data flow is often modeled by waiting queues. Although many real systems have a stochastic character, stochastic analysis is complex and not possible in all cases.

### 3.5.1 Statistical distributions in Queuing theory

When latencies like the IRT or the PDLT are measured for many times, the measurement results often can be approximated by a mathematical distribution.

In this section, statistical distributions that are important for the measurements in this thesis are presented, for further references, see [6].

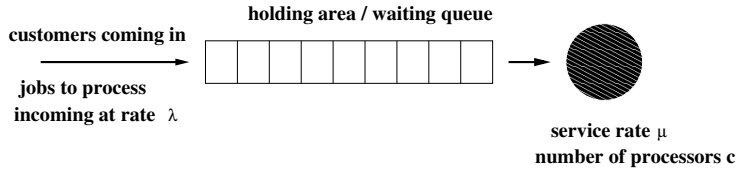



Figure 3.3: *Basic model of queuing theory*

The queuing theory uses a simple basic model to describe a system. This model consists of a working unit, that consists of one or more service stations of the same type and of a waiting room or waiting queue. The customers arrive separated and at random points in time in the waiting room.

If at least one of the service stations is currently free, the customer is served, if not, he is put into the waiting queue. The notions 'customer' and 'service station' can have different meanings in practical applications of queuing theory.

In operating systems the customers are tasks or interrupt service routines which want some labor to be done and the serving stations are controller chips or the CPU of the computer.

### The exponential distribution

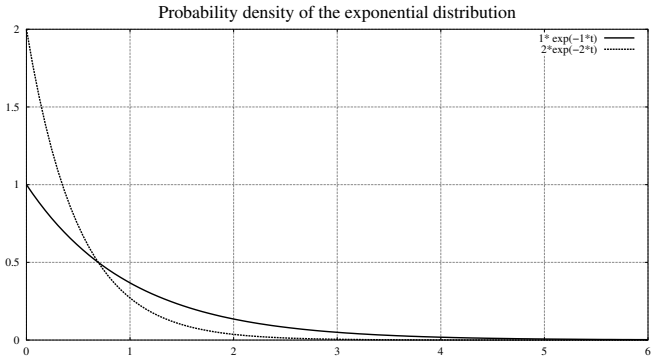



Figure 3.4: *Probability density of the exponential distribution*

The exponential distribution, see fig.3.4, is often used in statistics to describe the duration of continuous operations. For instance, such operations can be the time to wait or the processing time of technical devices.

### The Erlang distribution

The exponential distribution is often used in practical applications. Its highest probability is situated at $t = 0$. But in the real world there are often situations that are similar to a footrace, where a minimum time is needed to cover the distance, then the first reaches the aim, later on most of the people reach it and later the last ones finish.

The Erlang distribution, see fig.3.6, p.42, is one of the distributions that can describe such situations:

The probability density of the Erlang distribution is given by:

$$f(t, \mu, k) = \frac{(k\mu)^k}{(k-1)!} * t^{k-1} * e^{-k\mu t} \tag{3.3}$$

Eq.(3.3) describes $k$ service stations that are distributed following an exponential distribution with the parameter $\mu$. In case $k = 1$ eq.(3.3) describes an exponential distribution.

## 3.6 Measuring methods

Measurements permit statements about the performance of a special system and furthermore they help to validate and improve models of the object in question. They also permit to judge the performance of new algorithms and this an important means when optimizing a system.

In general measuring methods must adapt to the quantities of the systems that are to be measured. Before a measurement procedure can be designed one must have an at least simple model of the object to measure and of the frequency and volume of the data that shall be acquired. This is necessary to determine the hardware that is needed to execute the measurements. For the evaluation of the measured data often statistical routines can be used.

Operating systems are determined by static as well as by dynamic qualities [58]. Static qualities are e.g. the number of lines of source code, the number of system calls and other kernel parameters. Performance analysis concentrates on the dynamical, especially on the temporal behaviour of the system.

To compare the performance of different systems often so called benchmarks are used to provide special performance parameters. Benchmarks are discussed in sec.3.6.1.

On the other hand to control whether a computer system operates as specified or to gather pieces of information about the internal operations and its temporal behaviour so called monitors are used.

In this thesis monitors have been used to provide an insight into programs or into an operating system. This is necessary to be able to optimize complex pieces of software. Furthermore, monitors can visualize internal operations and parameters calculated out of the measured data can help to evaluate and quantify the success of an optimization. Sec.3.8 takes a look on these special measurement tools.

### 3.6.1 Benchmarks

Benchmarks are an approach to determine special parameters of a complex computer system that can characterize its performance regarding a special quality or component.

There are many different benchmark programs to test and evaluate different components of a computer system. Typically the measured performance parameters characterize the performance of the whole system regarding a special quality, i.e. the performance of the hardware and the software together. All benchmarks that are executed as applications, i.e. in user mode, measure the performance that the operating system provides on the given hardware of the computer system. If the performance of the hardware alone without the influence of the operating system shall be measured, this can be done more exactly by routines that are part of the kernel code of the operating system. Those routines have a more direct access to the hardware.

When evaluating a computer system benchmarks must be chosen according to the purpose of the system.

Since performance means - according to its definition - to do a special work in a certain amount of time, the easiest and most obvious method of performance evaluation is measuring the execution time for a special task.

Often a single operation is executed very fast on a computer system, so to avoid measurement errors often a benchmark executes the same operations for many times and calculates statistical measures like average, variance, maximum and minimum out of this data.

One advantage of measuring repeatedly is that measurement errors of the clock used are reduced. Another one is that the values measured are often statistically distributed and thus repeated measurements provide more reliable values than a single measurement does.

Two categories of benchmarks for operating systems can be distinguished:

- application benchmarks
- synthetical or fine grained benchmarks

Application benchmarks are useful to test a system for a special purpose, while synthetical benchmarks work with a standardized load profile. Such load profiles can be ported to other platforms, thus they make a comparison of different platforms possible.

An example of an application benchmark to test realtime performance is the *Hartstone Benchmark*. This benchmark executes a number of periodical tasks with different frequencies, some one-shot tasks with deadlines and some synchronizing processes. The computer system should meet the deadlines of all tasks. During this benchmark the load is increased until the maximum load is reached and the system starts to miss a deadline.

An example for synthetical benchmarks are the ones that are proposed by the SPEC organization, the 'Standard Performance Evaluation Corporation'. This assembly provides a set of standardized programs and load profiles to compare the performance of processors.

The main challenge of a synthetical benchmark is to define a load that gives results that are meaningful and reproducible even on very different architectures and platforms. To generate such a load often a representative mix of operations is executed.

Often the same measurement is repeated while the surrounding conditions, e.g. the load or some other system parameter, are changed [17]. Such a series of measurements using the same benchmark makes a comparison of different platforms or computer systems possible.

While a benchmark defines its own measurement procedure, it is left to the person who measures to determine and write down the environmental conditions of the measurement, which can have a significant impact on the measured performance parameters. E.g. the overall load of the system during the measurement or hardware details of the architecture can be such important environmental conditions that are needed to avoid misinterpretations of measured benchmark data.

Benchmarks in general are not appropriate to find out worst case latencies or worst case situations. Since the operations of a benchmark are executed repeatedly statistical analysis of the measured data provides a range of values with a minimum and a maximum. But as the measurement is statistical it cannot be determined whether the measured maximum is really the maximum that can ever occur. Often the measured worst case execution time depends also on the current load of a computer system and the maximum possible load of a system differs in between different systems. Thus analytical analysis is much more appropriate for WCET analysis than benchmarks are, see sec.3.3.

### 3.6.2 The Rhealstone Real-Time Benchmark

The Rhealstone Benchmark [43] is an example of a fine grained benchmark. Fine grained benchmarks are in general collections of simple programs that measure special qualities of an operating system. This benchmark analyzes several capabilities and the efficiency of an operating system that are important when dealing with time-critical tasks.

The Rhealstone Benchmark is a well known benchmark for Real-Time operating systems. It has been developed in 1989 [43] and since that ported to different architectures [42, 7]. It belongs to the class of ´fine grained´ benchmarks that measure the average duration of often used basic operations of an operating system with respect to responsiveness, interrupt capability and data throughput.

In this thesis the Rhealstone Benchmark has been ported to Linux [23].

All benchmark programs use the scheduling policy SCHED_FIFO and the processes are locked into memory.

- **Context Switch Time**
  In some newer Linux kernels like the kernel 2.4.5, the system call `sched_yield()` does not work sufficiently for SCHED_FIFO, i.e. for soft realtime processes. Therefore when porting the benchmark to Linux the intended call of `sched_yield()` in some benchmark programs had to be replaced by the call of `sched_setscheduler()` with a different priority, but always with the policy SCHED_FIFO. Because `sched_yield()` did not work, it is currently not possible to measure the ´Task or Context Switch Time´ on Linux according to the propositions of the reference implementation [42].

- **Preemption Time**
  This is the average time a high priority process needs to preempt a process of lower priority, if the second one executes code in user space and does not hold any locks. In the benchmark program the lower priority process just executes a loop. Herein included is the time the scheduler needs to find out which process to execute next.

- **Intertask Message Passing Time (System V)**
  This is the average time, that passes in between one process sends a message of nonzero length to another process and the other process gets the message. The port done in this thesis uses the System V Message Queues implemented in Linux.

- **Semaphore Shuffle Time (System V)**
  This means the average time in between a process requests a semaphore, that is currently held by another process of lower priority, until it obtains the semaphore. The time the process of lower priority runs until it releases the semaphore is not included in the measurement. So here the implementation of the semaphores is measured. In this thesis the benchmark program has been adapted to the System V semaphores for processes, that Linux offers. Since the Rhealstone

benchmark does not measure interthread-communication, the POSIX semaphores implemented in the `libpthread` library to be used with threads only are not measured here.

- **Interrupt Response Time (IRT)**
  This is the average time in between an external peripheral device generates an interrupt and the first command of the interrupt service routine (ISR) as first reaction to the interrupt, see section 3.2.2, p.34. This time is especially affected by the hardware used. The measurement has been carried out using the measurement principle described in section 3.6.4.

  The original benchmark measures the ILT - the interrupt latency time -, the time in between the CPU gets an interrupt and the execution of the first line of the interrupt handler.

- **Deadlock Breaking Time**
  Standard Linux does not possess an implementation of semaphores with ´priority inheritance´ [103], which is inevitable for this benchmark program to make sense. So this benchmark couldn't be applied to Linux.

Applying the formula of the Rhealstone Benchmark to unify all the results up to one single value, seems to be only meaningful to compare different platforms.

### 3.6.3 Periodical Task

A periodical task can be used to benchmark some realtime qualities of an operating system. To understand what is tested an understanding of the internal operations of an operating system is necessary. A task that shall be executed periodically is put asleep after execution and invoked in most implementations of operating systems by routines following the timer interrupt that updates the internal time base of the system. This implementation assumed the question whether a periodical task is reawakened and scheduled in time is in fact a measurement of the IRT and PDLT of the timer interrupt, see section 3.2.2, p.34, that the operating system provides on a given hardware.

### 3.6.4 Measuring IRT and PDLT using the parallel port

As mentioned in sec.3.2.2 the IRT and the PDLT are important metrics to characterize the responsiveness of an operating system. To measure them, one must generate an interrupt at a given and known point in time. A device that is very easy to handle on the Intel platform and possesses an interrupt line is the parallel port. Thus some of the measurement tools that have been designed or enhanced during this thesis make use of the parallel port. A similar technique already has been used in [58] and [87].

#### Principle of measurement

**Interrupt Response Time (IRT)**
This is the average time in between an external peripheral device generates an interrupt and the first command of the interrupt service routine (ISR) as first reaction to the interrupt. This time is especially affected by the hardware used.

To generate an interrupt from the parallel port the measurement program writes a '1' to the highest bit of the output port of the parallel port, using an `outb()` call in Linux. It's important that before there was a zero written on that bit, because only the change from zero to one or from low to high is interpreted as an interrupt at the IRQ-pin.

A simple wire leads the electrical signal out of that bit into the interrupt input pin of the parallel port. Fig. 3.5 shows the way of the signal in the computer system.

The outb() call can be done out of a device driver in kernel mode in Linux or with root permissions even out of user mode.

The measurement is done by taking 2 time steps:

$$\delta t_{Latency} = t_2 - t_1 \tag{3.4}$$

The time stamp $t_1$ is taken just before the `outbyte()` is written to the parallel port. The time stamp is taken from the Time Stamp Counter, the TSC, which provides a timing resolution being the inverse of the processor frequency and thus being in the range beneath microseconds, see sec.3.7.1. Since the IRT is normally in the range of some microseconds that's a sufficient resolution. To measure the IRT the second time stamp $t_2$ is taken from the TSC by the first instruction of the interrupt service routine.

Also the PDLT can be measured this way. In case of the PDLT the second time stamp $t_2$ is taken by the first instruction of the task that has been awakened by the interrupt service routine.

As writing to the parallel port of a current Intel compatible PC using `outb()` needs in between 1.4 and 1.8 microseconds and this time is included in the measurements, the measurements of the IRT and PDLT are not more precise than up to 1 or 2 microseconds. How fast an `outb()` command is executed can be estimated by measuring the time 2 `outb()` commands need to be done.

Generating an interrupt this way, the interrupt is synchronized to the kernel-thread performing the `outb()` call to trigger the interrupt.

But the IRT-results measured with this method turn out to be in the same order of magnitude as the IRT measured with other tools, so this is a valid measurement principle.

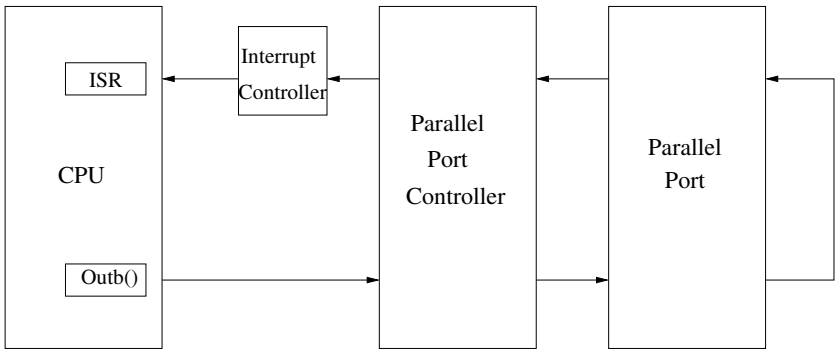



Figure 3.5: *Signaling pathway to generate an interrupt at the parallel port programmed by the CPU*

### Evaluation of the measurements

The results of the $N$ measurements are the measured latencies $t_i$. For presentation the latencies are divided into classes. Each class has a frequency $H_i$ and a relative frequency $relH_i$ of

$$relH_i = \frac{H_i}{N}, \tag{3.5}$$

with $relH_i < 1$ and

$$\sum_{i=1}^{N} relH_i = 1. \tag{3.6}$$

### Measurement of the IRT of the standard Linux 2.4 kernel

In this paragraph an example for the measurement of the IRT shall be presented.

Fig.3.6 shows the relative frequencies of a measurement series of $N = 500$ of the standard Linux 2.4.18 kernel without a significant load. The figure also shows an Erlang distribution that has been correlated with the measured data by fitting the parameters $k$ and $\mu$ of eq.(3.3). All those correlations about

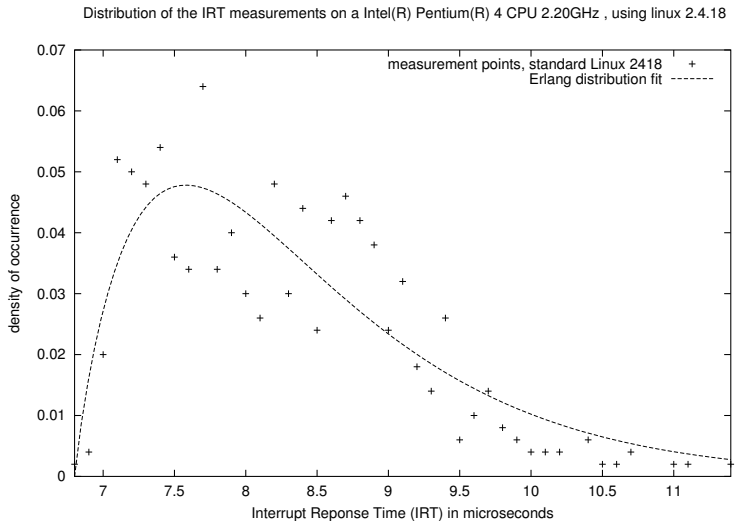



Figure 3.6: *500 measurements of the IRT of a standard Linux 2.4.18 kernel*

different series resulted in a parameter $k \approx 2$. If one puts a model of queuing theory beyond, this means that the interrupt signal passed two service stations with exponentially distributed service times.

Since the interrupts are generated by software and the start of the IRT measurement is well known, the arrival times are of no importance. The duration of the IRT latency thus follows from the statistical times the stations need to process them.

In this model at every controller the interrupt signal has to wait. In the model waiting times are exponentially distributed at every controller. Fig.3.5 shows schematically the pathway of the signal in the computer. The result of the different exponentially distributed waiting delays is an Erlang distributed delay.

If one looks at fig.3.5 this two service stations could be the Interrupt Controller (IC) and the CPU. At both chips the interrupt signal can suffer from variable delays until it can be processed, when the controller are still doing other tasks. On the other hand, since the parallel port is used exclusively and there is always only one signal passing it, the delay of the parallel port controller should be nearly constant and thus irrelevant for the probability distribution.

## 3.7 Measurements of time rely on precise clocks

Measurements to evaluate the performance of a computer system rely on precise clocks. More and more devices are integrated in modern computers that allow a performance evaluation of the system. Since all the measurements of this thesis have been executed on Intel compatible personal computers, the clocks of this hardware platform are characterized and compared hereafter.

### 3.7.1 Different clocks in modern Intel compatible PCs

#### The Programmable Interval Timer (PIT)

The Programmable Interval-Timer (PIT) generates out of the external cycle signal of an oscillator an interrupt after a timing interval of a definite and programmable length. Its cycle is independent of the clock cycle of the processor.

The PIT is realized as a CMOS chip of the series 8253, later on 8254. The chip contains 3 independent counters that can be programmed separately from each other. Every counter consists of a 16 bit register. The external quartz oscillator generate a frequency of 14.317 MHz, which is the historical processor cycle

rate of the IntelX86 processors. To decrease the error the rate is divided by 12, so that the frequency at the input of the 3 counters (CLK0 bis CLK2) is only 1.19318 MHz. This means that all 3 counters are fed with a time basis of 838 nanoseconds.

The PIT is programmed using the port 043h.

The 3 counters of the interval timer can be programmed in 6 different modes, but only the first 4 modes are of interest here:

1. One-shot Mode
   In this mode the counter is initialized with a value and every clock cycle the counter is decremented. When it reaches zero an output is raised from low to high which can be used as a timer interrupt. The high value remains at the output until the counter is reinitialized. This mode allows to fire a single interrupt after a definite time interval, therefore its name *One-shot mode*.

2. Programmable Monoflop
   As initial value the number of clock cycles is given, while the output shall be set to Low. If during this period a second trigger impulse occurs the counter is reinitialized.

3. Periodic Mode
   The counter is initialized with a value N and starts to count down after it has been loaded. If it reaches zero a negative peak, a Low value for only one clock cycle is generated at the output and the counter is reloaded with the same value N. The temporal distance between to impulses at the output is N clock cycles of the PIT. This mode is used in an Intel compatible personal computer to generate periodical interrupts, thus it is called Periodic Mode.

4. Square-pulse generator
   In this mode the initial value is twice the length of the rectangle. During the first half period a High value is generated at the output, after half of the initial value a Low value is generated until the counter reaches zero and is reloaded with the same value. That way a square wave voltage is generated at the output of the counter.

An operating system, like Windows and Linux, programs the PIT at boot time to fire an interrupt - the timer interrupt - periodically. This interrupt defines the time base of the operating system, since the operating system can't keep track of every processor clock cycle, but nevertheless is in need of a periodical interrupt when another time interval of definite length passed by. Via the timer interrupt an operating system offers a time base to its applications and it perform certain periodic maintainance tasks on a regular time basis.

#### The Realtime Clock (RTC)

The Realtime clock of a personal computer is a chip of the series Motorola MC146818 or a compatible chip, which is today integrated in the South Bridge. The heart of the Realtime Clock is a 64-Byte-CMOS-RAM. The first 14 Bytes are reserved for time, date and data for status and supervision. The counter in its standard resolution is increased every 976,56 microseconds; this means a frequency of 1024 Hz.

The RTC can be programmed to generate frequencies that are multiples of 2 and in between 2 and 8.192 kHz. With the programmed frequency the RTC triggers the interrupt 70h. The RTC is independent of the timer interrupt and thus does not interfere with the timing base of the operating system.

#### The Advanced Programmable Interrupt Controller (APIC)

The APIC chip has been designed to replace the older standard interrupt controller, the XT-PIC (8259A). The APIC is part of modern Intel compatible 32 bit personal computers. It is part of the processor and the APIC communicates using the system bus. The APIC can fire periodic interrupts, the highest frequency possible is the one of the system bus. In multiprocessor systems it is used to generate the timer interrupt. In single processor systems that is an option too. Furthermore, the APIC contains a counter that is decremented with bus frequency and can fire a single interrupt when it reaches zero.

### The Time Stamp Counter (TSC)

With the Intel Pentium-S-Pro INTEL started to implement a *Time Stamp Counter*. This counter is a 64 bit register, that is initialized to zero when the computer is booted. After booting this register is incremented by one every processor cycle. AMD integrated this hardware counter into the K6 processor and all following processor architectures.

Since modern processors work with high tact rates, the resolution of this counter is currently in between $10^{-9}$ and $10^{-10}$ seconds. With the commands *rdmsr* and *rdtsc* since the Pentium Pro the value of the register can be read. The high resolution of this counter makes it possible to measure also the execution time of only short assembly routines.

On a 32 bit architecture the Time Stamp counter consists of two 32 bit registers. Since the higher register is incremented only all $2^{32}$ processor cycles for short measurements that are executed repeatedly it's enough to read out only the low register, if measurements with an overflow are excluded from the evaluation.

The command to read out the 64 bit TSC counter *rdtsc()* is an assembly command. On an AMD K6 with 400 MHz, as described in appendix A, this assembly command needs about 38 processor cycles to be executed. The pseudo code to measure this value is listed here:

```
rdtsc(t1);
rdtsc(t2);
diff = t2-t1;
```

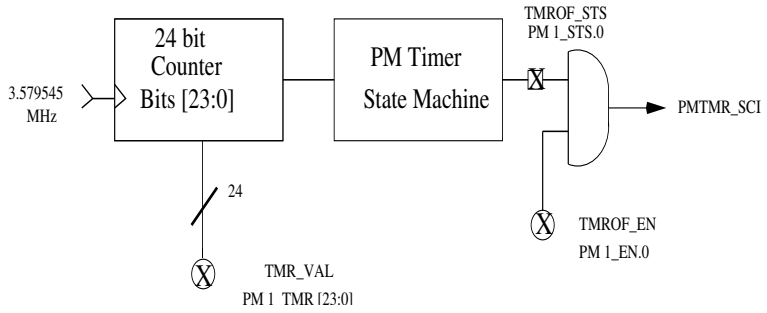



Figure 3.7: *Schematic structure of the ACPI power management chip*

### The Advanced Configuration and Power Interface Power Management (ACPI pm)

The power-management-Timer are used in modern operating systems to measure timing intervals, during which special devices have not been used. The states of such devices can be stored and some of the devices can be shut down in order to save energy.

The power management currently consists of a counter running with a frequency of 3.58 MHz. The power management can fire interrupts below or at this frequency.

### Comparison of the clocks of an Intel compatible PC

Table 3.7.1 shows a comparison of the clocks currently provided by most Intel compatible personal computers.

At first, a distinction has to be made whether a clock can just be read out like the Time Stamp Counter (TSC) or whether it can fire an interrupt. Those that can normally have an one-shot-mode that means the clock is initialized with a value, counts down and generates once an interrupt when the clock reaches zero. On the other hand in periodic mode this procedure is executed repeatedly and the interrupt is generated periodically, e.g. the timer interrupt to update the time basis of the operating system. Some

| Quality | RTC | PIT | APIC | ACPI pm | TSC |
|---|---|---|---|---|---|
| Fire Interrupts | + | + | + | + | - |
| One-shot M. | ++ | ++ | + | + | - |
| Periodic M. | + | + | ++ | | - |
| Programmed | IOCTL | Register | MM-I/O | Register | - |
| Range | 2 Hz - 8 kHz | up to 1.19 MHz | up to bus freq. | up to 14.3 MHz | - |
| Theo. Res. | $2^{1..13}$ Hz steps | approx. 0.001 Hz | 1 bus Cy.* | approx. 0.0001 Hz | 1 CPU-Cy. |
| Real Res. | $2^{1..13}$ Hz steps | 1 Hz | 10 - 20 bus Cy.* | | 38 CPU-Cy. |
| Handling | simple | medium | difficult | medium | simple |

Table 3.1: Qualities of different clocks on an Intel compatible PC, here a Pentium 4 with a bus cycle of 100 MHz, with a Linux 2.4.18 kernel as operating system running

timer chips are easy to program, some are more difficult, because they are more complex. Some have special qualities, e.g. some APICs have a longer starting phase and thus are less adequate for the one-shot mode. At last the resolution of the clocks differs, a rule of thumb is the nearer they are to the processor the higher the resolution is.

The traditional device of Intel compatible personal computers to generate a periodic time interrupt is the PIT. For SMP systems the APIC has been developed which is also part of modern Intel compatible PCs and thus can replace the PIT. The RTC can be used to generate periodical interrupts in between 1 and 8192 Hz, it is not needed for the normal operation of the system.

## 3.8 Monitors

### 3.8.1 Monitoring an Operating System

As introduced in section 3.2.2 the IRT, PDLT and KLT are important latencies to characterize the realtime capabilities of an operating system. For realtime systems the Worst Case Execution Time (WCET) of these latencies is of special interest.

Using monitors it is possible to collect data that provide a deeper understanding about the internal operations of a computer system. Every monitor consists of parts to collect and possibly visualize data as well as the objects to measure, in this thesis mostly the operating system.

Monitors can be classified according to different criteria:

- **Object to measure:** As object to measure every components of a computer system is possible, the hardware, the operating system, a database, an application program or a network stack. Measurement categories can be all measurable quantities, events or operating sequences.
- **Measurement tools:** There are different ways to collect the needed data. If the data is collected by programs, it's called a software monitor. If there is a hardware tool collecting the data it is a hardware monitor. There are also hybrid forms, called hybrid monitors. When measuring software components an advantage of hardware monitors is that they are completely separated from the object to measure. Therefore hardware monitors don't influence the object of interest while measuring. On the other hand hardware monitors require special equipment to collect the data, which software monitors do not.
- **When is the data collected ?** Regarding at what points in time the measured data is taken one distinguishes time-driven and event-driven monitors. Time-driven monitors simply sample interesting data at periodic time intervals. The frequency to sample can be adjusted to the needed precision, if the volume of the sampled data can be handled by the measurement tool. When using a software monitor the load of the system caused by the measurement has to be considered. Event driven monitors on the other hand record only events. Some monitors only count events, others record a complete description of an event. Event driven monitors allow an easy reconstruction of the internal operations of the system.

- **Online or offline evaluation ?** The evaluation can take place online or offline. Online evaluation means that the data is presented to the user while the measurement is still running. Complex calculations are often performed offline after the measurement has been finished. If the data volume is huge, possibly a data reduction is necessary before the data can be stored, for instance sometimes only average and variance are stored.

There are two different monitor schemes that are used in this thesis to find long latencies in an operating system.

- **Time-driven monitors**
  An often used method is to find long latencies statistically, repeating the same measurements frequently. Long latencies like a long PDLT will especially occur to the high priority measurement process, when other processes perform many operations at the same time. So normally these tools measure while the computer deals with a heavy load produced by for instance benchmark programs in the background over long times. The hope is to measure long latencies that way by chance. An example for this measurement technique running on Linux is the tool 'realafeel' by A.Morton, see [100]. In fact, one is never sure, one found the Worst Case latency, but several big latencies have been found on Linux that way. Using special Linux bug trace features of the kernel it is sometimes also possible to identify what part of the code caused this latency.

- **Event-driven monitors**
  If weaknesses of the operating system concerning real time constraints are already theoretically known, it's possible to create special situations that long latencies will be measurable. For instance in standard Linux kernel code is not preemptible, so [58] developed a method called 'Software Monitor', which permits to generate and measure latencies caused by non-preemptible kernel code. This method is rather powerful, since dedicated parts of the operating system like for instance special system calls can be measured and the execution time of these parts of the code is scanned and can be visualized. In Linux as an open source operating system the code that caused the latencies can be viewed and so a deeper understanding of the operating system principles is possible. It's also possible to test improvements that have been made in the kernel code.

- The third possibility is only a theoretical one, a total cover of all code parts of the operating system -even interfering code parts- isn't possible in general. Not even a total cover of sensible parts like executing all the code of the Linux kernel system calls is possible, because depending of the input parameters different parts of kernel code are executed in the system calls because of if/else or case statements. Also, the execution time of loops varies heavily on the input data of a system call or on the history of the system. For example a malloc(memory_amount) on a system working with a heavy computational code using most of the RAM of the PC could cause a long latency, if the operating system has to iterate through long lists to find which memory page it should write to the swap device and then allocate this memory page for the new malloc. To stay with the example, also the input parameter of malloc, 'memory_amount' can influence the latency, because the operating system needs longer to find a bigger memory area not already used or to swap out several memory pages. But the latency will not increase continously when 'memory_amount' is raised. Since Linux always allocates multiples of the memory page size - 4 kilobyte on Intel PC's - only if 'memory_amount' crosses the border of a memory page size. From this example we see that the duration of latencies can vary because of input data and history and that often the duration of latencies depend heavily on implementation details.

### 3.8.2 A special Software-Monitor

The software monitor is a useful method developed by [58] that allows to measure several latencies like the IRT, the PDLT and the KLT in special situations. To use the software monitor a special system function or another part of the operating system is programmed. So this tool allows detailed timing analysis of special parts of the operating system and possible improvements.

Most of its software runs in user space using normal threads or processes, so the source code of the operating system is not needed for the measuring method. But in most operating systems - also in Linux - it's necessary to implement a small kernel driver to handle an interrupt.

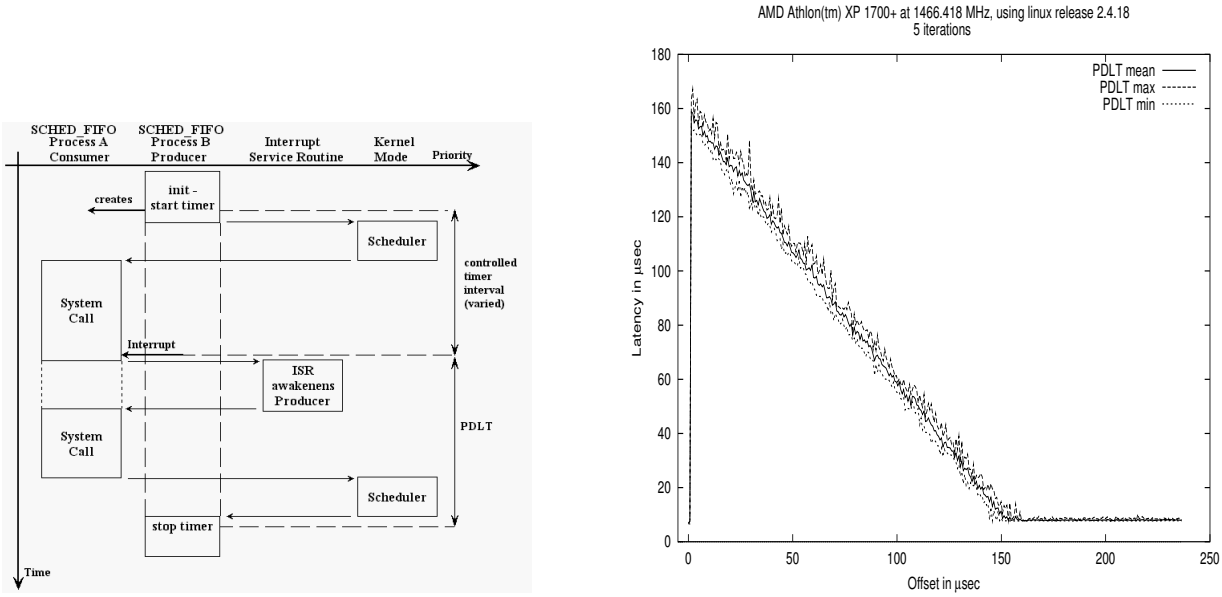



Figure 3.8: Left part: *Illustration of the measurement method, called the 'Software Monitor'*, right part: *A measurement of the PDLT of the standard Linux system call* `write(64kB);` *made with that software monitor*

The basic measurement principle of the software monitor is to generate an interrupt at a well known time. When the interrupt handler comes to execution it takes a time stamp to calculate the IRT later on and it awakes a sleeping high priority soft realtime process. When this process gets the processor and starts its execution after the scheduler has chosen it, a further time stamp is taken in order to calculate the PDLT later on. In order to observe how the operating system works internally, the interrupt shall come, when another thread is fulfilling a special routine of the operating system, e.g. a system call. In a preemptible operating system the thread executing the system call will be preempted and will not affect the PDLT. But in a non-preemptive operating system like Linux the soft realtime process will have to wait, until the thread finishes the execution of the system call, before it can proceed. This means a higher PDLT.

### Non-preemptible regions show up as triangles in the diagrams

A low priority SCHED_FIFO, i.e. a soft realtime process - called 'Consumer' in Fig.3.8 -, executes a system call, which runs in kernel mode. When an interrupt occurs a timer on a PCI timer card [63] is started. The interrupt service routine (ISR) following the interrupt interrupts the low priority SCHED_FIFO process - i.e. the 'Consumer' -, even if it executes kernel code, and awakens a high priority process - the 'Producer' in Fig.3.8 -, that preempts the low priority process as soon as possible. We assume here that the system call does not disable interrupts, if it does it disables them mostly only for short periods on the order of microseconds. After the system call is finished or a preemption point in the code of the system call is reached, the scheduler is invoked and starts the high priority SCHED_FIFO process - called 'Producer' - as the process of highest priority in the system ready to run.

As its first instruction the high priority process stops the timer and calculates the value of the PDLT, as shown in Fig.3.8. The PDLT is the time in between the interrupt occurs and the high priority process obtains the processor. This measurement procedure is repeated varying the length of the 'Controlled Timer Interval' in Fig.3.8, i.e. varying the moment the interrupt occurs related to the start of the Consumer's system call. The interrupts are generated by an external PCI-Timer-Card [63]. That way the system call is scanned by generating an interrupt every microsecond. Every measurement generates one point in the diagram, see figure 7.3, p.102. Fig.3.8 shows the PDLT of the system call `write` when it writes 64 kByte into a shared memory area, which produces a maximum PDLT latency of 160 microseconds on an AMD Athlon XP with 1,4 GHz running a Linux 2.4.18 standard kernel.

If the interrupt occurs just after the system call has been started, the PDLT triangle reaches its highest point, because the whole system call has to be fulfilled before the high priority process can start running.

Interrupting later on, only shorter parts of the system call have still to be fulfilled, so the triangle decreases until it reaches the baseline of the triangle in the end of the system call. In the Figures 7.3-7.5 the execution time of the system call is marked by a thick line on the X-axis.

### Problems of the software monitor using ISA/PCI cards

Disadvantages of the old software monitor

- The software monitor first realized was difficult to port, because it contained a big driver for the ISA or PCI card and the PCI implementation of the Linux kernel has been changed several times from Linux 2.0 to 2.2 and 2.4.
- Another problem was the enormous increase of CPU speed compared to the moderate increase of the buses. The processor frequency of a typical Intel 486 PC, was about 33 MHz and the ISA bus speed was 8 MHz. Modern PC's have CPU speeds of 2.2 GHz and PCI bus speed of 66 MHz up to 133 MHz.
- The measurement interval was not longer than $2^{16}$ usecs, which means about 65 msec, which is shorter than the longest latencies of a Linux 2.2 and 2.4 kernel on several PC's.

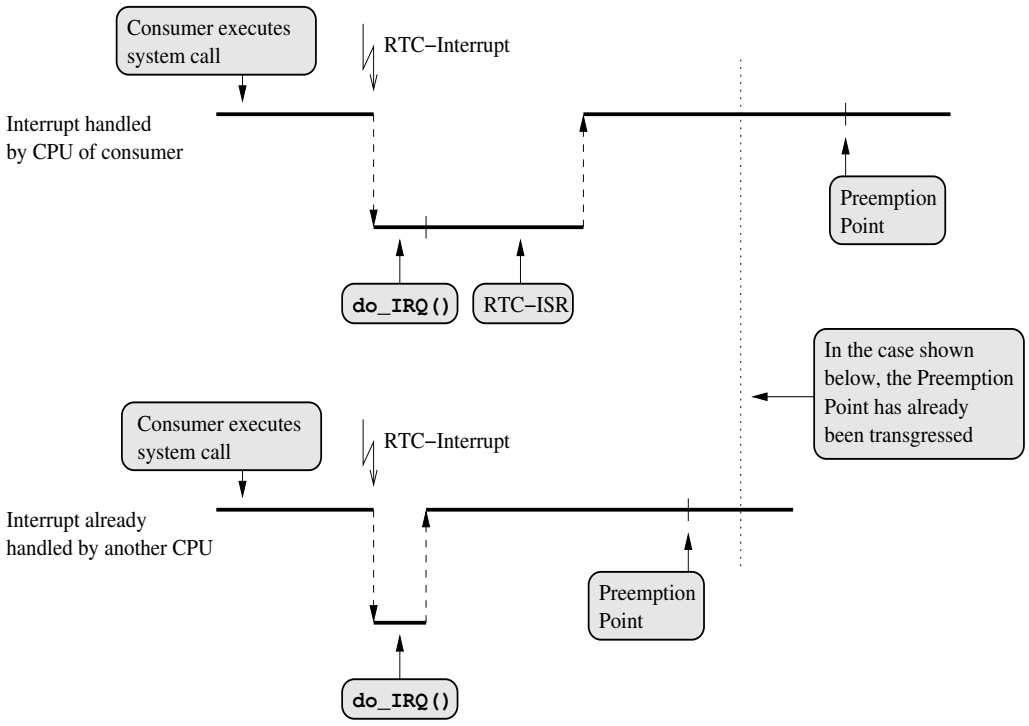



Figure 3.9: *Situation on an multiprocessor system: If the interrupt is handled on another CPU, the measurement does not represent the worst case and isn't taken into account.*

### Implementing a new software-monitor using the Real Time Clock and the Time Stamp Counter of modern PC's

Looking at the principle of the software-monitor [59] there are 2 basic items needed to reimplement it:

- a global time base
- a precise interrupt source

Since the Pentium Pro Intel PC's have a Time Stamp Clock (TSC) processor register, which counts cycles of the processor clock and therefore provides a global time base since booting, which can be used for measurements. Since the TSC register is increased by the processor clock, it's speeded up automatically when processors got more and more faster in recent years. The TSC register normally provides the best timing resolution available on the PC. The Real Time Clock (RTC) is a part of every x86 PC, too, and it's capable to generate interrupts periodically with a frequency range in between 1 Hz and 8192 kHz.

The old software monitor [58] prepared a timer on the external ISA or PCI card to generate an interrupt at e.g. $\delta$t = 54 microseconds from now on, then it started the execution of a system function, that was interrupted by the interrupt 54 usecs later on. This measurement has been repeated varying $\delta$t in between 1 and about 100 usecs, which was enough to scan most of the system functions on an AMD K6 with 400 MHz, as described in appendix A.

The new software-monitor works with the interrupt coming from the RTC periodically. At first it determines exactly when an interrupt from the RTC comes using the RTC. As it comes periodically with e.g. a frequency of 2 KHz, it assumes that the next interrupt will come in exactly 500 usecs. Now a thread is started, which uses the scheduling policy $SCHED\_FIFO$. It performs a busy wait executing a loop until e.g. $\delta t_{wait} = 444usecs = 500usec - \delta t$ have elapsed. After this time $\delta t_{wait}$ of 444 usecs the system function to measure is executed. It is interrupted by the periodical RTC interrupt 56 usecs later on. The Interrupt Service routine wakes up the Producer as in the old software monitor and the IRT and the PDLT are measured. This measurement is repeated varying the sleeping interval of the idle thread e.g. from $\delta t_{wait} = 400$ usec to 500 usecs, so again the system function is scanned in between 0 and 100 usecs after it has been started. Using this principle the maximal duration of a system function or latency to be scanned is $T = \frac{1}{RTC frequency}$ and as the RTC frequency can be varied in between 1 and 8192 kHz, the maximum measurable latency would be 1 s long, which is more than sufficient.

This new software-monitor produced measurements with a precision of about 1 microseconds and less noise than the older one also because of the higher accuracy of the TSC and various error corrections built in [16].

#### Implementation of a software-monitor for SMP systems

At first the concept how the software monitor can be extended to multiprocessor systems has to be discussed:

The worst situation for a soft-realtime process on a SMP system with N processors occurs when N-1 processes are currently executing on the other N-1 processors. Only in this case consumer and producer thread will execute on the same CPU and so the Consumer is able to delay the Producer thread by executing special routines like e.g. a system call or a critical region. An easy way to block all other N-1 CPU's is to make them execute soft realtime processes of a higher priority than the consumer thread and the producer process.

On a multiprocessor system the interrupt controller informs all CPU's about a new interrupt. The CPU's start racing for the interrupt, that one that enters the Linux kernel function `'do_IRQ()'` first will lock a spinlock to enter a critical section, in which it sets the `IRQ_INPROGRESS` flag. This flags tells all other CPU's that this interrupt is already processed by a CPU.

Since the software monitor tries to generate worst case latencies, PDLT measurements on SMP systems where the Interrupt Service Routine has been executed on a different CPU are ignored by the measurement tool, because these cases do not represent the worst case, see fig.3.9.

This way, the SMP software monitor blocks all other N-1 processors by processes using the scheduling policy SCHED_FIFO and proceeds like the software monitor measuring on a single processor system.

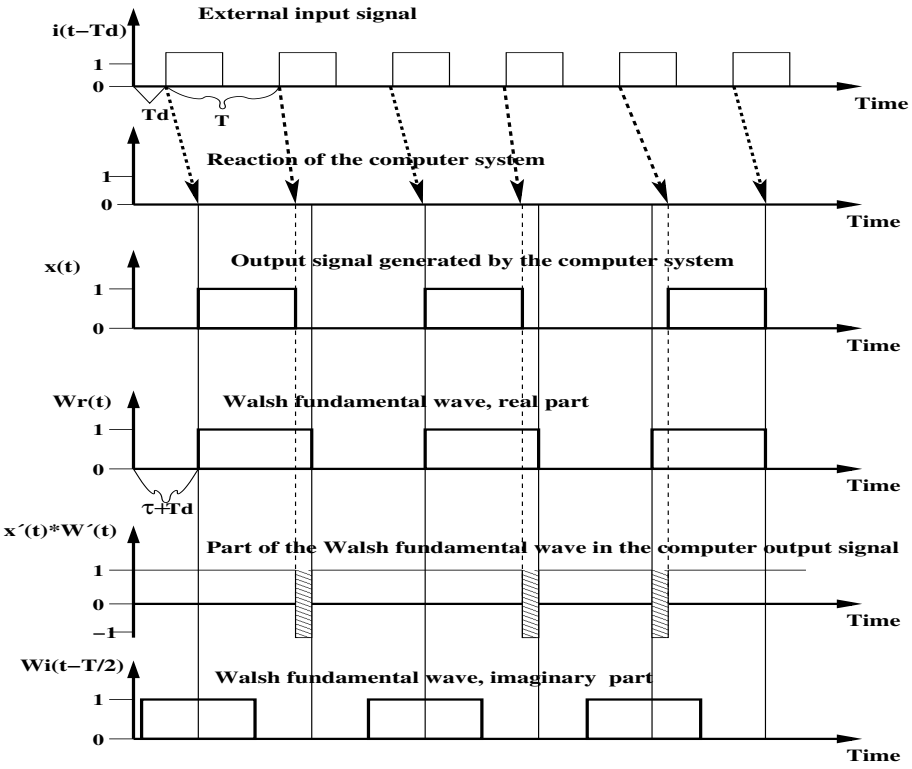



Figure 3.10: *Orthogonal correlation of computer output signal and Walsh fundamental wave*

## 3.9 Orthogonal Walsh correlations to evaluate the responsiveness of operating systems

### 3.9.1 Basic principle

The Walsh-correlations [12] are another metrics to evaluate the responsiveness of an operating system. They have been developed from the idea that a computer performing time-critical tasks could be described by a frequency response locus like an electronic device which reacts with a delay to a periodical external signal.

Thus the Walsh-correlations are founded on the behavioural model, that looks on a computer system as a black box, that behaves in a certain way, as described in sec.3.5.

To analyze the Walsh frequency response the operating system is excited by a periodical pulse string of a certain frequency, e.g. a square waveform.

The challenge the computer system has to fulfill is to invert a signal at one of its outputs, every time the square wave signal at the input has a rising edge. So the operating system works like a toggling flip-flop. The following 3 properties of this system shall be evaluated:

- Does a reaction of the computer really follows to each incoming pulse or does the system leave out to toggle after a pulse sometimes ?
- How delayed does the system react in average ?
- How reliable is the timing behaviour of this reaction ?

To get a quantitative measure the degree of reliability $DoR$ is defined:

$$DoR = \lim_{n\to\infty} \frac{1}{n*T} \int_0^{n*T} x(t)w(t-\tau)dt \tag{3.7}$$

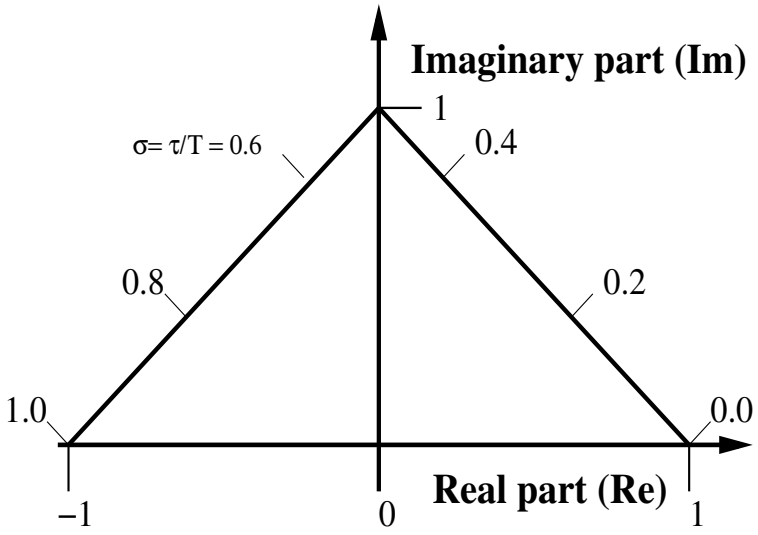



Figure 3.11: *Frequence response locus of the Walsh correlations for different frequencies*

$x(t)$ is the output signal of the computer system to analyze. $w(t-\tau)$ is called the Walsh fundamental wave. In fact $w(t-\tau)$ is the signal that has been used to excite the computer system only delayed by a time $\tau$. $\tau$ means the average delay the output signal of the computer system $x(t)$ shows in comparison to the input signal.

Eq.(3.7) is called a Walsh correlation. The two values of $x(t)$ and $w(t-\tau)$ must be standardized to either -1 or 1 in eq.(3.7).

For an ideal computer system the output signal $x(t)$ equals $w(t-\tau)$, so the degree of reliability $DoR$ is 1 in that case. For all other systems $DoR$ will be in between 0 and 100 %. Of course every computer system has a cut-off frequency, that means the computer system can't react to frequencies above. So the frequency must be in between: $0 < f < f_{cut-off}$.

If one studies the behaviour of $DoR$ more carefully, one sees that the value of $DoR$ can be in between -1 and 1. But values in between -1 and 0 only indicate that a wrong value for $\tau$ has been chosen.

So an exact way to get a reliable value for the average delay of the computer system $\tau$ has to be found.

This problem is solved [12] by the correlation of the output signal $x(t)$ with the orthogonal Walsh fundamental wave $w(t-\tau-\frac{T}{2})$. Thus analogously to other electrical devices a real part and an imaginary part of a value - here the degree of reliability $DoR$ - can be defined:

$$Re(DoR) = \lim_{n\to\infty} \frac{1}{n*T} \int_0^{n*T} x(t)w(t)dt \tag{3.8}$$

$$Im(DoR) = \lim_{n\to\infty} \frac{1}{n*T} \int_0^{n*T} x(t)w(t-T/2)dt \tag{3.9}$$

For an ideal system, that reacts to every pulse with a constant delay $\tau$ the frequency response locus is shown in fig.3.11.

In this figure special points can be identified, where either the imaginary part or the real part of the degree of reliability $DoR$ equals zero. This happens at special frequencies:

1. For the frequency $f = \frac{0.5}{\tau}$ $Re(DoR) = 0$ and $Im(DoR) = 1$.
2. For the cut-off frequency $f = \frac{1}{\tau}$, $Im(DoR)$ is zero and $Re(DoR)$ is $-1$.

A possibility to measure more points of the frequency response locus is to delay the incoming signal for a certain dead time $T_d$ before it is fed into an input of the computer system.

Therefore the system of the delay element and the computer system will react like a computer system which reacts with an average delay of $(T_d + \tau)$.

With the delay element the equation for the special point where $Re(DoR)$ equals zero, is changed into

$$f = \frac{0.5}{\tau + T_d} \tag{3.10}$$

The average delay time of the reaction of the computer system $\tau$ can't be chosen deliberately, but the dead time $T_d$ can. That way for a special chosen $T_d$ it's easy to measure those 2 points of the frequency response locus, where $Re(DoR)$ or $Im(DoR)$ equal zero. This can be achieved by adjusting the right frequency belonging to a chosen dead time $T_d$. If $Re(DoR)$ equals zero, $Im(DoR)$ equals the absolute value of the degree of reliability.

### 3.9.2 Measurement setup and results

A measurement setup to measure orthogonal Walsh correlations of a computer system has to be done in hardware, because the expected delays are of the order of microseconds.

Such a measurement setup is described in [12]. The frequency of a function generator offers a square waveform. A frequency divider divides this frequency by 16 which reduces also possible fluctuations of the cycle of the function generator. This square wave signal $i(t)$ triggers two flip-flops, one triggers on the rising flank of the signal $i(t)$ thus generating the Walsh fundamental wave $w_R(t)$. Another flip-flop triggers on the falling flank, thus generating the orthogonal Walsh fundamental wave $w_I(t - \frac{T}{2})$. Except from the 2 flip-flops the square waveform $i(t)$ is delayed by an adjustable delay element, in [12] it can be adjusted in $\frac{1}{16}$th of the period $T$ of the signal $i(t)$. The output of the delay element $i(t - T_d)$ is fed into the input of the computer system which tries to toggle its output $x(t)$ on the rising flank of the incoming signal. The output $x(t)$ of the computer system has half the frequency of the incoming signal.

The input values of the inverted XOR element can only be 0 or 1. So the multiplication of $x(t)$ and $w(t)$ in eq.(3.8) and (3.9) can be replaced by an inverted XOR element, because it generates a 1 at its output when both inputs are equal and a zero when they are not.

Therefore the output signal of the computer system is XOR'ed and inverted with the Walsh fundamental wave $w_R(t)$, see eq.(3.8). On the other hand $x(t)$ is XOR'ed and inverted with the orthogonal Walsh fundamental wave according to eq.(3.9).

Time counters are used to count the value of $Re(DoR)$ and $Im(DoR)$ doing the summation, which replaces the integral of eq.(3.8) and (3.9) for discrete values.

Using this measurement setup [12] for every adjustable value of the dead time $T_d$ the frequency exciting the computer system has to be searched for, that makes the real part of the degree of reliability $Re(DoR)$ to 0. The $Im(DoR)$ then equals the absolute value of the $DoR$, that should be measured. It depends on how many steps of the dead time $T_d$ the adjustable delay element provides how many measurements can be taken with this setup to find out the frequency according to eq.(3.10).

At the cut-off frequency $f = \frac{1}{\tau}$, Im$(DoR)$ is zero, but most computer systems will loose pulses at a period that is near their own average delay time. So the degree of reliability $DoR$ decreases to zero before it reaches that point.

Measurements with this setup presented in [12] show that the cut-off frequency for the standard operating system used there was around 20 kHz, while a true real time operating system, called RTOS-UH, went up to 120 kHz and more on the same PowerPC hardware. The degree of reliability $DoR$ decreased only to 90 percent until the realtime operating system reached its cut-off frequency. The $DoR$ of the standard operating system on the other hand decreased more quickly to only about 80 percent at its cut-off frequency, although its cut-off frequency was only the sixth part of that of the realtime operating system, all measured on the same PowerPC hardware.

# Chapter 4

# Architecture of systems serving time-critical tasks

## 4.1 Architectures supporting standard as well as time-critical applications

Different approaches to perform time-critical tasks along with other tasks on the same computer are discussed in this section.

Hard real time operating systems, like for instance VxWorks, QNX or LynxOS, are able to handle time-critical and other tasks on the same computer [81, 79, 57]. But these systems often only have a limited number of software programs available for them. Besides, they are often subject to runtime licenses, so called 'royalties', or only provide developer licenses for thousands of Euros or US-dollars.

Developers of Embedded Systems tend to use standard operating systems today, because of the wide spectrum of software available for them. Apart from that, such off-the-shelf operating systems are relatively cheap. This is important especially for mass production.

A proprietary operating system often hasn't been ported to run on new or special processors or to support specialized hardware. So customers of such proprietary operating systems have to write drivers on their own to use special peripheral devices. The use of standard operating systems often makes it superfluous to invest money for these purposes, since there is a wide variety of processors, mainboards and peripheral devices available, standard operating systems provide drivers for. Hardware manufacturers normally sell their devices including drivers for standard operating systems, in order to increase the number of units sold.

Often applications in Embedded Systems also have to fulfill realtime requirements. But standard operating systems often do not provide a good preemptability, high timing resolution or appropriate scheduling policies for time-critical tasks. So normally extensive tests have to be done before it can be stated that an operating system running on a chosen hardware is qualified to execute a task with soft-realtime constraints.

One of the main disadvantages of using off-the-shelf operating systems for time-critical tasks is that their internal subsystems aren't decoupled and thus accidentally any program of minor importance running on the same computer system can cause suddenly arising latencies for the time-critical tasks. Furthermore these latencies also depend on the underlying hardware and the quality of the drivers the operating system provides. The problems performing time-critical tasks on standard operating systems will be discussed in more detail using the operating system Linux as an example in chapter 5.

There are several other possibilities to combine the advantages of a real time and a standard operating system on the same computer [88]:

So there are several strategies how both requirements can be met by one computer system:

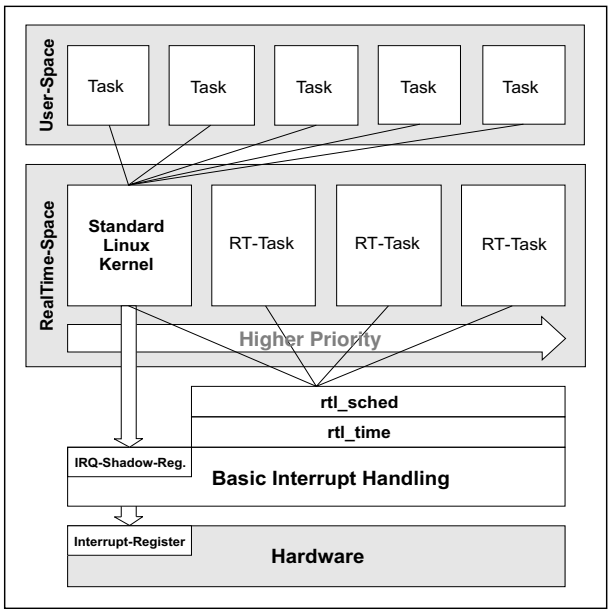



Figure 4.1: *Architecture of RTLinux and first versions of RTAI*

### 4.1.1 Standard operating system scheduled by a small hard realtime kernel

One approach is to put a hard realtime capable operating system in between the hardware of the computer and the standard operating system, see fig.4.1. The hard realtime system installs its own scheduler that is managing the hard realtime threads, which normally operate in the kernel address space of the standard operating system. When no hard realtime thread is ready to run, the standard operating system is executed as task of lowest priority by the underlying realtime scheduler. So the standard operating system only gets the processor, if there are no time-critical tasks to process. That way the standard operating system does not know that there is a hard real time system underneath performing hard-realtime operations.

Apart from its own scheduler the hard realtime operating system normally introduces its own precise timing modules and its own system calls and API.

The hard realtime system must have direct access to the interrupts and all hardware it deals with. Only those interrupts that are not used by the hard realtime system are forwarded to the standard operating system. Interrupt locks done by the standard operating system do not affect the realtime system, since the standard operating system only works on

Since these both operating systems don't share much code, the realtime system can be programmed for hard real time, although the standard operating system does not fulfill these requirements. It normally does not use any services of the standard operating system, because these services are not realtime-compliant. Only at boot time the hard realtime operating system normally let the standard operating system routines boot the PC, until it can take over control.

A small disadvantage is that the embedded system engineer has to divide his application clearly into time-critical tasks working on the hard realtime operating system and other non-realtime tasks like visualizing or configuration and networking that can be done in the full-featured standard operating system. There are normally special communication facilities like pipes, shared memory or signal handlers to exchange data in between the hard realtime system and the standard operating system.

It can be said, that the standard operating system runs in a virtual machine, because hardware devices that are used by both systems have to be emulated for the standard operating system. It's possible that some peripheral devices are managed exclusively by the realtime operating systems, others exclusively by the standard operating system. The realtime operating system must always keep the control of all shared devices in order to be able to guarantee hard-realtime requirements.

The architecture and technique of the dual kernel approach are complex and only skilled operating system developers can implement such a system. A deep understanding of both kernels and the underlying

hardware is necessary to develop and maintain such a solution. Often the environment emulated for the standard operating system has to be slightly changed, if a new version of the standard operating system is used. Both kernels - that of the hard realtime operating system and of the standard operating system - work in ring 0 of the Intel architecture, cp. section 2.2.6.

A further disadvantage is, that the underlying hard realtime system often provides only a few drivers for peripheral hardware, that its functions are often limited and that it sometimes uses another application programming interface (API) than the standard operating system.

Examples of this approach are RTLinux (RealTime Linux) [85] and first versions of RTAI (RealTime Application Interface) as hard realtime operating systems scheduling the standard operating system Linux as task of lowest priority, see fig. 4.1. The Real Time Extension (RTX) is an example of this approach for Windows NT [80]. The dual kernel approach - as described above - is subject to a patent in the State of New Mexico, USA, hold by FSMLabs Inc., the developer and distributor of RTLinux [85].

### 4.1.2 The nanokernel approach: Adeos and PikeOS

There is another interesting approach to make a hard realtime kernel work on the same hardware as a standard operating system like for instance Linux:

Systems like Adeos, Adaptive Domain Environment for Operating Systems [102], introduce a small software layer - that could be called a nanokernel - on top of the hardware in order to get control over the hardware resources of the computer. Afterwards, they allow different operating systems to run on the same computer and to share its hardware resources. Such nanokernels also install a priority order among the operating systems running in their so called 'domains': A realtime operating will be posed in the first domain and thus get every hardware interrupt first. After that, other operating systems in other domains will receive the same interrupt. That way the RT operating system always has first access to hardware resources. The nanokernel - for instance Adeos - resides in ring 0 of the Intel architecture, cp. section 2.2.6. Normally the kernel of standard operating systems like the Linux kernel works at ring 0, too, but with the Adeos nanokernel they must work at ring 1, which is normally unused in the current Intel architecture. That way Linux working in ring 1 has only limited hardware access. Therefore, hardware processor instructions made by Linux can be trapped by Adeos. Adeos can interfere, simulate the results or execute the Linux instructions using a single stepping mechanism. So, Adeos retains the full control of the hardware. Adeos itself is no operating system, as it does not provide processes or memory management or other functions. It only enables several operating systems to run on the same hardware.

One of the first realtime operating systems using the Adeos technique has been RTAI [83]. In order to avoid violating the RTLinux patent, RTAI has been built on top of the Adeos nanokernel since version rtai-24.1.11 for Linux 2.4. RTAI is a small hard realtime operating system that is provided as a kernel module for the Linux kernel. It works in the first domain of the Adeos kernel. When it is idle, Adeos takes back control to let for instance Linux work. This technique enables a hard realtime operating system - or a kernel debugger - to work in parallel, but being prioritized to a standard operating system.

Another example is PikeOS, sharing resources of a computer system among different operating systems, such approaches are used in aircraft industry [40].

### 4.1.3 'User space realtime concepts'

So called 'User space hard realtime concepts' are normally based upon the dual kernel approach described in the last paragraph. They implement further services on top of the underlying hard realtime kernel in order to offer also realtime capable services in user space. But the new realtime services in user space often offer a new application programming interface, a new API. So standard software written for the standard operating system normally doesn't use this new API. From this realtime capable API user processes can't use routines of the standard operating system, if they want to fulfill realtime constraints. To increase the preemptability the standard operating system can be preempted at any time by the user level realtime services. A condition for this ability is that the user level realtime services do not share any routines or resources with the standard operating system. If the user level realtime services shared such resources with the standard operating system, preemptability locks of the latter would apply also to the

user level realtime services. Therefore routines of the operating system can be executed only by helper processes or threads, that return when the work is done. As the functional range of the underlying hard realtime operating system is limited, also the range of the offered API in user mode is limited.

An example for that approach is LXRT which is built on top of RTAI in Linux.

#### RTAI/Fusion and the Xenomai project

RTAI/fusion [83] is a new experimental branch of RTAI: It consists of Adeos [102], RTAI, and the Xenomai project [13]. The Xenomai nanokernel relies on the RTAI kernel and RTAI services, which itself works using the domain technology of Adeos. Thus Xenomai is hard realtime capable. The Xenomai nanokernel provides basic services of a hard realtime operating system in order to be able to emulate the API of different well known Real Time Operating Systems (RTOS) on top of Xenomai. The Xenomai / RTAI / Fusion project shall help to port applications with hard realtime requirements to RTAI/Linux, i.e. Linux/GNU based operating systems, without having to change the API and entire design or behaviour of those existing realtime applications. Such an application running on RTAI/Fusion can be in the sandbox of a (soft)realtime User-Space-Virtual Machine (UVM) in user space based on a multithreading Linux process or they can be implemented using hard RT capable kernel modules.

### 4.1.4 Sharing the processor among operating systems using hardware support

Another concept is the coexistence of two operating systems on the same processor supported by hardware mechanisms.

This approach can for instance make use of the Non Maskable Interrupt (NMI) of an Intel compatible PC, which interrupts the processor periodically and can't be masked by software.

Using a chip attached to the PCI-bus the NMI interrupt is used to switch periodically in between a hard realtime operating system and a standard operating system that don't share common code. The other hardware resources have to be assigned statically to the both operating systems. This means that both operating systems have only a loose coupling, the point in time, when the processor is switched is independent of the internal operations of both systems, thus they can't use for instance the same file system.

The only possibility of communication in between these separated operating systems normally is a shared memory region, used by a special protocol, a TCP/IP stack or a serial interface.

Using this mechanism for instance industrial robots of KUKA incorporation are controlled by a hard realtime system using Windows 95 as standard operating system for visualization and networking [49].

### 4.1.5 Resource kernels

Resource kernels assign time intervals in percent of the available CPU time or other resources like for instance network bandwidth to dedicated time-critical tasks, so that they can fulfill their services independent of other tasks and of the workload of the whole system. Often resource kernels also provide admission control mechanisms to avoid that the system is overloaded and thus misses deadlines. This approach is useful for periodical tasks, but it doesn't help much for acyclic tasks that have to react fast to external influences like interrupts. Further problems are how to know before, how much CPU time a special time-critical task will consume. An adaptive resource or feedback driven scheduler can resolve that problem for periodical jobs, but not for a task that is executed only once. Furthermore, it's not clear what happens, if a periodical task does not finish in the processor interval that has been assigned to that task. So this solution is only soft-realtime capable.

An example of this type is the Linux Resource Kernel, called Linux/SRT [54].

### 4.1.6 The purpose of this thesis - Modifying the Linux kernel to make it serve time-critical tasks in a better way

The purpose of this thesis is to improve the kernel of a standard operating system so that it can perform time-critical operations in a more timely way than the standard operating system on the same hardware does. The application program interface of the kernel, i.e. its system calls, shall not be altered, so that applications and libraries don't have to be rewritten or adapted.

Therefore changes in the kernel of the operating system have to be done. For such a purpose the source code of the operating system has to be freely available and changeable. Linux therefore is an ideal choice.

#### Conclusion

One can conclude from the concepts presented above that performing time-critical tasks and tasks without timing requirements on the same computer often means some extra work for the developers. There is probably no ideal solution to this problem, as in nearly every case developers have to make compromises to reach their goals. Nevertheless, there has been a broad variety of implementations of the concepts presented above throughout recent decades. This shows that the advantages of each approach are big enough to find the interest of some researchers and to meet the needs of some users to solve their time-related problems.

## 4.2 Combining divergent goals

### 4.2.1 The change of requirements operating systems shall comply with

In recent two decades personal computers have penetrated the business world and the use of computers in private area is still increasing. With the introduction of digital photography computers start playing a major role in this field too and the substitution of the old VHS video technique by DVD-players opens the field of home entertainment also for standard operating systems. In fact today DVD-players often are tailored, specialized personal computers. On the other hand multimedia PCs, equipped with TV cards and DVD burners, all run by enhanced standard operating systems like Windows or Linux can record and replay movies, cut films or advertisements and even provide a cinema atmosphere at home.

These new fields of usage for personal computers also changes the requirements a modern standard operating system has to fulfill. Recording and replaying audio and video data requires that standard operating systems of today have to accomplish the requirements of soft-realtime tasks, cp. section 3.2. Especially audio tasks are demanding and don't tolerate long latencies: When replaying sound samples e.g. for voice over IP, the human ear hears already latencies of around 40 milliseconds as a snap.

In the second part of this thesis Linux is taken as a typical standard operating system of today and it is analyzed, which steps have to be done to make it more compliant with the needs of time-critical tasks.

### 4.2.2 Synchronous and asynchronous systems

Operating systems can be distinguished regarding different criteria. One of this criteria is whether an operating system is synchronous or asynchronous.

Many realtime operating systems are synchronous operating systems. Those systems only work at predefined programs, no signals of the outer world reach them and thus they have a predefined deterministic, often cyclic application flow that is well known before. The only possibility for a process in such a system to interact with its surroundings is a mechanism called *polling*. In this case a task periodically asks a peripheral device whether it can deliver a new value.

Since this - in a way autistic - systems show a deterministic behaviour, that is known before, there are only usable for predefined tasks and systems, that have regular periods and well defined limited duties. They don't offer flexibility to adapt to events in the external world, that haven't been foreseen.

Nevertheless, many systems with realtime constraints are synchronous systems. E.g. even some combat airplanes that fly in an instable mode are controlled by a computer running a synchronous multitasking operating system with periodical tasks, that are calculating always in the same sequence. In such systems the execution times of tasks only vary due to different amount of data they might use in different loops or due to different cache allocations.

A standard operating system is an asynchronous system. This means it contains different asynchronous elements like signals or interrupts, see section 3.2.2, that are used to transfer messages from peripheral devices to the CPU and to applications running on it. Signals and interrupts can change the sequence of the applications that run on the processor.

On the one hand operating systems using interrupts work more efficient than systems that have to poll. On the other hand in asynchronous systems the time until a special task is fulfilled depends on more parameters than in a synchronous system.

Using an asynchronous operating system often even the time until the first reaction to an interrupt takes place depends on several parameters and thus is not predictable in general. For instance, before the Interrupt Service Routine is started after an interrupt occurred, there can be an interrupt lock, a preemption lock or other interrupts of higher priority can be handled in between. Thus in asynchronous operating systems even the first reaction to the interrupt is not deterministic.

The fact that the operating system reacts first to interrupts and signals, that occur stochastically, makes it even more difficult to predict the time when a special task, that has been awakened by the interrupt service routine, can be executed as second reaction to the interrupt. Also the completion of this task can be delayed by various interrupt service routines, that interrupt the task, or by a task of higher priority.

So asynchronous systems show a much bigger flexibility, which on the same time causes a smaller predictability.

### 4.2.3 Comparing standard and real time operating systems

In general standard and real time operating systems have to meet the same requirements, but - according to the different goals of the systems - the single items are seen from different angles:

- `Temporal behaviour and data throughput:`
  The goal of standard operating systems is to permit a good average utilisation of the computer resources.

  Since normally the I/O systems of a computer like the hard disk nowadays act much more slowly than the CPU, Linux Torvalds once said that the primary task of a good standard operating system is to keep the I/O systems running, so that the CPU can do as much work as possible.

  On the other hand, realtime operating systems shall assure that time-critical tasks can meet their deadlines under all circumstances. The most important aims of a realtime operating system are predictability and determinism, even in so called worst case situations.

- `Economical utilisation of resources:`
  Especially realtime operating systems often shall run on cheap embedded systems without requiring maintenance operations. So they must work with less resources than standard operating systems. Economical aspects are e.g. amount of memory, CPU frequency and power consumption.

- `Stability and Reliability:`
  The failure of an application shouldn't have an effect on other applications, or on the stability of the operating system. Especially realtime operating systems must provide a good stability.

- `Flexibility: Integration of hardware components:`
  The success of standard operating systems like Windows or Linux is based on their ability that code for drivers to run new devices or code for new concepts can be integrated into the core of the operating system in an easy and fast manner.

  On the other hand, realtime operating systems normally consist of code, that has been tested or sometimes even certified. Therefore, such operating systems aren't subject to fast code changes.

- `Compatibility and amount of available software:`
  Applications are easier and thus faster to port to operating systems that comply with standards like POSIX, cp. section 2.3. The amount of available software is important for both classes of operating systems. Normally a lot more software is available for standard operating systems than for specialized operating systems.

- `Offered user interfaces and kernel size:`
  A standard operating system offers different APIs and interfaces in order to satisfy as many of its user's different needs as possible. This enlarges the kernel of the operating system as well as its libraries.

  For a realtime operating system it's important to have only the parts in the kernel that are necessary to accomplish the specified task, they have been built for. That way other parts can't interfere and the operating system is small enough to fit even on cheap hardware with limited resources.

- `Scalability:`
  This means the same operating system shall run on a uniprocessor system as well as on multiprocessor systems. Furthermore, it shouldn't show a decreasing performance when processing a lot of network traffic.

- `Security:`
  Protection against unauthorized access or use of data is important, the importance of this topic will probably even rise in next years.

Thus, the goals of standard and realtime operating systems are different and divergent and therefore difficult to combine. One can expect that no operating system can fulfill both goals perfectly at the same time.

Realtime operating systems aim at maximum reliability while standard operating systems try to combine maximum flexibility with sufficient runtime stability and a high data throughput.

There are a number of reasons why today more and more often standard operating systems are used also for applications with timing constraints, where before only real time operating systems have been used.

Standard operating systems provide the following advantages:

- Many users and software developers already possess knowledge how to handle standard operating systems
- Standard operating systems provide tools for an easy visualization of data provided by technical processes.
- A huge amount of available software, good development tools and graphical user interfaces, many users are already accustomed to.
- Standard operating systems run on cheap off-the-shelf hardware. They provide drivers for many peripheral devices.
- Some standard operating systems like e.g. Linux don't require license fees, they are free operating systems, although they offer a huge amount of functionality.

Nevertheless, there are also disadvantages, when using standard operating systems for time-critical tasks:

- Standard operating systems often need more powerful and more expensive hardware to run, e.g. normally standard operating systems require a MMU, i.e. a Memory Management Unit. Sometimes they don't run without keyboard and monitor -so called 'headless'- or diskless, i.e. without hard disk.
- Standard operating systems normally don't provide hard realtime capabilities. Also they provide only limited soft-realtime capabilities.

What seems to be desirable and achievable is a better statistical compliance of standard operating systems with temporal deadlines. This means standard operating systems shall be enhanced to meet soft-realtime requirements.

But a strong fulfillment of realtime requirements could also lead to a degradation of the average performance of the operating system. This is not wished by users, who use it only as their desktop operating system. Apart from that, also realtime systems must be economic, especially the power consumption should be low, also in order to avoid to overheat the processor.

## 4.3 Special requirements for realtime operating systems

There are 2 important requirements that have to be fulfilled by a realtime operating system [58]:

1. The reaction of a realtime system to an event must be within a definite period of time, which has been specified before.
2. The task must be done in a certain predetermined span of time.

It's obvious that the second requirement can't be satisfied without the first being true. The first one depends mostly on the qualities of the operating system. For the second one it is important to know whether the task can still be intercepted by e.g. Interrupt Service Routines.

Some further requirements have to be satisfied regarding virtual memory management, scheduling and the preemptability of a realtime system.

Standard operating systems often don't fulfill all of the following requirements. For these reasons and for their inherent design and the high complexity and flexibility of standard operating systems, these systems cannot meet the requirements of hard realtime tasks.

### 4.3.1 Virtual Memory Management of realtime tasks

Virtual memory management permits user applications to need more memory than is physically installed on the computer by saving parts of the memory content on disk space of the hard disk. In the past, operating systems used to put the address space of a whole process to hard disk, that is called *Swapping*.Today only parts of memory of a certain size, called *pages*, that can belong to any process, are put on hard disk, this is called *Paging*.

Since a hard disk is a rather slow device compared to memory, it is important for realtime tasks, that no part of their allocated memory is ever put to a hard disk. For the same reason in most systems parts of the kernel address space are never put on a hard disk in order not to decrease the performance of the system.

Modern operating systems provide system calls to assure a time-critical task can lock itself into memory and avoid that its memory pages are sent to disk. In Linux this system call is called `mlockall(..)`.

### 4.3.2 File I/O

File I/O can be non-deterministic for different reasons. At first, the used hardware peripherals like hard disks often don't behave in a deterministic manner, so different latencies occur. For instance, the time to move the head to the data that shall be read next on a magnetic hard disk is a latencies with statistical distributed values.

Furthermore, standard operating systems apply several strategies to enhance the throughput of I/O subsystems, that can conflict with the temporal requirements of time-critical tasks. For instance, some hard disk drivers first collect data and then reorder it to avoid unneeded movements of the heads of a hard disk.

Therefore, realtime operating systems often provide so called realtime filesystems to make file I/O more realtime-compliant.

### 4.3.3 Scheduling of realtime tasks

Many different scheduling policies and algorithms exist, not all of them are adequate for realtime tasks. In practice, for realtime tasks often priority scheduling is used. Round Robin schedulers that work with time slices and let every task run for only a certain period of time can't be used for realtime tasks, because using such a policy it can't be granted that requirement 2 is fulfilled, i.e. that the tasked is finished in a certain space of time. Section 6.1 will analyze the scheduler used in Linux in more detail.

### 4.3.4 Preemptability

An important precondition for requirement 1, see above, is that user tasks - or the part of the operating system, that is currently working on the processor - can be intercepted by a part of code that has a higher priority and thus should be executed first on the processor. Such interceptions can be interrupt routines, as well as tasks of higher priority that became ready to run after an interrupt occurred. This possibility to stop the currently running task or tasklet and to replace it by one of higher priority is called the *preemptability* of an operating system.

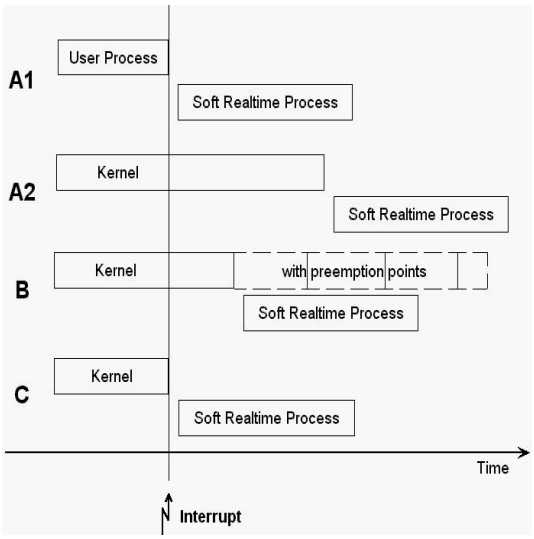



Figure 4.2: *Concepts of Preemption*

Fig.4.2 shows how different operating system react, when an interrupt awakens a soft realtime process [46].

Regarding preemptability different cases have to be distinguished. This paragraph and fig.4.2 look at how different operating systems react to an interrupt. It's assumed hereafter, that interrupts are not currently blocked and the interrupt service routine (ISR) can run soon after the interrupt signal reached the processor. The ISR wakes up a time-critical task, which has the highest priority in the whole system.

- If the interrupt intercepts a user process, cp. case **A1** of fig.4.2, directly after the interrupt service routine has been done the scheduler can be invoked. If the realtime process has the highest priority in the system, it starts executing soon, i.e. with a small PDLT. That's the case for most operating systems, also standard operating systems like Windows and Linux behave like that.

If kernel code is currently executed on the processor, when the interrupt occurs, the situation is different:

- A realtime operating system reacts as can be seen in case **C** of fig.4.2. Even when the interrupt intercepts kernel code like a system call or a kernel thread, the realtime process starts directly after the interrupt service routine finished and the scheduler has been executed. An operating system with an ideal preemptability behaves this way. Operating systems behaving like this can be called 'fully preemptible'.

- Standard operating systems of today mostly behave as can be seen in the cases **A1** and **A2** of Fig.4.2, i.e. kernel code can't be preempted and a long PDLT occurs, since the system call has to be finished first, before the scheduler can be invoked and the process of higher priority can be executed.

Standard operating systems often have a non-preemptible kernel to avoid all synchronization problems that could occur, when kernel routines, i.e. system calls, are intercepted by user tasks. The standard Linux kernel is a typical example for such a policy. This technique avoids conflicts of shared resources, used by more than one instance, and it avoids inconsistencies of kernel data structures, but it also leads to a long PDLT time-critical tasks suffer from on such a system.

The following sections deal with mechanisms to improve the temporal behaviour of operating systems, which have a kernel that is not fully preemptible, i.e. behave like case **A2** in fig.4.2.

**Section 4.4** analyzes the effect of Preemption Points in the kernel of an operating system, this is case **B** in fig.4.2.

**Section 4.5** deals with the introduction of critical sections into the kernel of an operating system. These critical sections can be protected by preemption locks. During a preemption lock is held, it is forbidden to invoke the scheduler. After doing that, the kernel can be made preemptible in general except for these critical sections. If a critical section is used by an interrupt, too, it has to be protected additionally by an interrupt lock to avoid race conditions.

If the critical sections are protected by mutexes instead of preemption locks, preemption is allowed even in critical sections. That way the kernel of an operating system can be made 'fully preemptible', this is case **C** in fig.4.2. **Section 4.6** reports some difficulties that have to be tackled before this solution can come true.

### 4.3.5 Stop a kernel routine and restart it later on

A possibility to allow a time-critical task to preempt the kernel is that a kernel function is stopped, whenever there is a time-critical task that requires the processor. To avoid data inconsistencies all actions a stopped kernel function has done so far have to be made undone, so that the state before the execution of the system function is reestablished. This method is also used in databases and called a *rollback* to a previous state that has been saved before. In this case kernel routines must act similar to database transactions, that take only effect, if the final *commit* is executed.

While this method can provide a short PDLT for the task of highest priority, it wastes nevertheless the performance resources of the computer, because all the work that has been done so far in the kernel routine will have to be redone later on. Thus the latencies for other tasks increase using this method and therefore it is hardly used in operating systems, also because the transaction mechanisms needed would cause some overhead in the operating system.

## 4.4 Preemption Points

The easiest way to make it possible that the operating system can provide fast reactions to tasks with a high priority is to enable routines of the operating system to be intercepted at least at certain points, also when kernel code is currently executed.

This can be done by introducing so called *Preemption Points* into the kernel code. Case B of fig.4.2 shows that the soft realtime process can come earlier to execution using a preemption point than in case A2 where the kernel of the operating system doesn't contain preemption points.

A preemption point means calling the scheduler out of a kernel routine at a certain point in the kernel code. So pseudocode for a Preemption Point looks like this:

```
system_call {
   do_something_in_kernel_mode();
   if(needed) call_scheduler();    // Preemption Point
   do_something_more_in_kernel_mode();
}
```

But a Preemption Point can be placed only at those special points in kernel code, where no data inconsistencies can occur. For instance, if a process shall be deleted from the runqueue, then at first all pointers of the linked list have to be changed, before the scheduler can be invoked. If not the linked list would be in an inconsistent state, possibly causing a system crash [99].

So Preemption Points make the kernel only preemptible at special points, not in general.

### 4.4.1 An analytical formula describing the effect of Preemption Points

Introducing Preemption Points into system calls reduces the $PDLT\_max$. The measurements shown in section 7.3.1 suggest that introducing more and more preemption points does not decrease the PDLT in a linear manner.

In this section a formula is derived that allows a prognostication of the reduction of the PDLT depending on how much Preemption Points have been introduced.

To analyze this question a test kernel driver is used as in section 7.3.1 described running on Linux.

The (i%N) function, i modulo N, defines the Number of equidistant Preemption Points in the loop of the kernel driver. Changing the N in the code above varies the number of Preemption Points. Using the software monitor the $PDLT_{max}$ has been measured. The results are shown in table and fig.4.3.

Fig.4.3 shows, the more Preemption Points are introduced the more the maximum PDLT decreases, but the absolute value of the decrease gets smaller. With eleven Preemption Points and more there is nearly no further decrease. The baseline of the PDLT is nearly reached. In the case shown here the most reasonable amount of Preemption Points would be between seven and eleven.

| Number of introduced Preemption Points | Maximum of PDLT $t_{PDLTmax}$ [$\mu$s] |
|---|---|
| 0 | 66 |
| 1 | 43 |
| 2 | 35 |
| 3 | 33 |
| 7 | 26 |
| 11 | 24 |
| 14 | 22 |
| 23 | 24 |
| 29 | 22 |

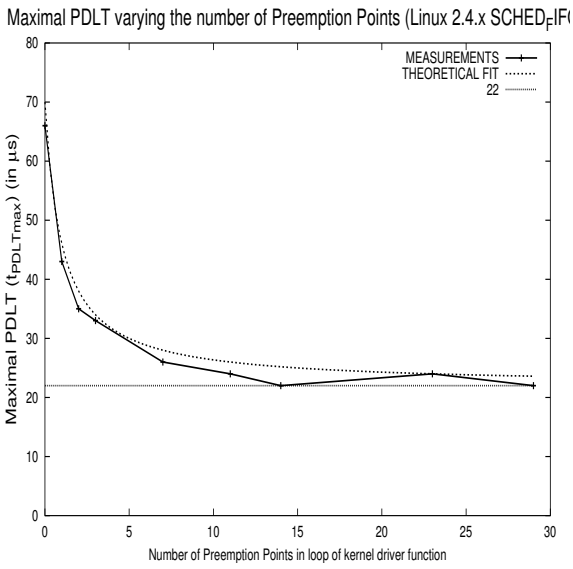



Figure 4.3: *Connection between the amount of equidistant Preemption Points and maximum PDLT, measure points and a theoretical curve. Decrease of the maximum PDLT with an increasing amount of runtime equidistant Preemption Points in a loop*

From the measurements the following formula can be derived: The formula shows the maximum PDLT depending on the number of introduced Preemption Points:

$$t_{PDLTmax} = \frac{t_{preemption-delay}}{n_{pp}+1} + t_{bottom} \tag{4.1}$$

- $t_{preemption-delay}$ := execution time of the non-preemptible system function. $= PDLT_{max} - PDLT_{Bottom} = 66 - 22 = 44$ in table and fig.4.4.
- $t_{bottom}$ := base value of the PDLT, results out of hardware delay, execution time of the ISR, time of context switches and the overhead of the measurement method, is equal to $PDLT_{Bottom}$ in fig.4.4.
- $n_{pp}$ := number of Preemption Points in the system function
- $$n_{sections} := n_{pp} + 1$$
  $n_{sections}$ is the number of non-preemptible sections within the system function.
- $t_{PDLTmax}$ := value of the maximum PDLT

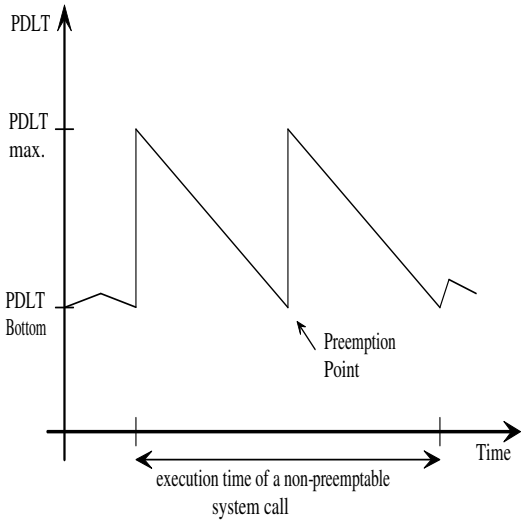



Figure 4.4: *Schematic illustration of the formula variables*

Instead of $t_{preemption-delay}$ one could also write $t_{Latency}^{standard-kernel}$. Equation (4.1) shows the connection between the size of the maximum PDLT and the amount of equidistant inserted Preemption Points. Figure 4.3 shows the graph of equation (4.1) for the measured values of $t_{preemption-delay} = 66-22 = 44\mu$s and $t_{bottom} = 22\mu$s:

If the system function contains no Preemption Point it consists of only one non-preemptible section. The system function is not preemptible at all. The execution time of the system function is part of the PDLT, more precisely of the "Preemption Delay", see figure 3.2 on p.35. Looking at this figure it's clear that Preemption Points only reduce the 'Preemption Delay', so $PDLT_{max}$ can only reach a baseline caused by the other latencies.

Equation (4.1) for the maximum PDLT only apply, if Preemption Points are inserted equally distributed over the execution time of the system function. If they are not equally distributed, eq.(4.1) is the lower limit of the $t_{PDLTmax}$. Without knowledge about the distribution of Preemption Points and consequently the duration of the longest non-preemptible section it is not possible to conclude from the amount of Preemption Points to the maximum PDLT.

One can conclude from equation (4.1) that for long Linux system calls, i.e. $t_{preemption-delay} >> t_{bottom}$ the introduction of only 1 Preemption Point in the middle of the system call at runtime will cut the original $PDLT_{max}$ into half on every Linux system. 9 equidistant Preemption Points would cut it into a tenth. So introducing Preemption Points into the kernel only decreases the $PDLT$ relatively to what it has been before with a standard kernel without Preemption Points. Absolute values depend on the system's processor speed and architecture.

## 4.5 Making an operating system preemptible except for critical sections

### 4.5.1 Critical sections to avoid race conditions

A critical section is a part of code that will cause a data inconsistency, if a task is intercepted by another task while executing the code of the critical section and if the second tasks executes the same code, before the first one has finished the critical section.

A typical example is the following: A new element, called 'new', shall be introduced into a linear list:

```
(1) new->next = element->next;
(2) element->next = new;
```

A process, called 'A', can start a system call of the operating system which executes line (1). If the scheduler of the operating system interrupts the code after line (1), and starts another process 'B', it's possible that 'B' manipulates the same list and for instance deletes the element of the list called 'element'. When later on process 'A' comes to execution again and proceeds in line (2), its pointer to 'element' would be invalid. The assignment in line (2) could write into a part of memory already used for another purpose, which could cause the operating system to crash. Furthermore, the new element 'new' hasn't been inserted into the list as intended.

This example shows that there must be a mechanism of mutual exclusion like a mutex around lines (1) and (2) of the code above in order to avoid concurrent access to the linear list or race conditions in general. Lines of code that have to be executed at once without interruption are called **'critical sections'**.

The next subsection and section 4.6 present different ways to avoid such so called 'race conditions' in an operating system.

### 4.5.2 Protecting critical sections using preemption locks, i.e. blocking the scheduler

Race Conditions can be avoided by temporary forbidding preemption. This means that the scheduler can't be invoked while a critical section is processed, so that no context switch can take place. This avoids a critical section from being called by two instances and thus avoid concurrent access to resources.

This method of avoiding race conditions by blocking the scheduler is referred to as **NPCS**, **N**on-**P**reemptive **C**ritical **S**ections.

NPCS have some advantages compared to other concepts of avoiding race conditions. It is a concept that is easy to implement, since no pieces of information about which task requires which resource at which time is needed. NPCS is best suited when the execution time of critical sections is short and if there are many resource conflicts in between instances that are running in parallel.

A disadvantage is that a task of high priority is blocked until any other task has finished the critical section, even if the task doesn't need access to that special resource. Especially if there are many long critical sections this leads to long and frequent latencies.

### 4.5.3 Interrupt locks: Blocking interrupts

A preemption lock affects time-critical operations less than blocking interrupts, since interrupt handlers as first reactions to an interrupt can still be executed providing a first fast reaction, even when a critical section is processed.

But if a special critical section is also accessed by interrupt service routines, i.e. by interrupt handlers, additionally to the preemption lock there must be also an interrupt lock around the critical section, for instance to avoid race conditions in between interrupt handlers and a task.

Often all interrupts are blocked, because this is more simple than blocking only special interrupts.

Interrupts should be blocked only for very short periods of time, since no pieces of information from peripheral devices, i.e. the outer world, can reach the CPU during this period.

## 4.6 Using mutexes to protect critical sections - Creating a fully preemptible kernel

Alternatively to preemption locks, i.e. NPCS, that block preemption in general, critical sections can also be protected from concurrent access by mutual exclusion mechanisms.

Mutual exclusion can be realized by a mutex or a binary semaphore. The binary semaphore is marked occupied when a task enters the critical section and only freed again, when the same process leaves the critical section.

Since a binary semaphore does not inhibit preemption or block interrupts, task switches can take place and even interrupts can be processed while the task is working in the critical section. That way the kernel of an operating system can be made fully preemptible. In such a fully preemptible kernel global preemption locks or global interrupt locks can be abandoned, they are necessary only in very special and rare cases, for instance to protect the mutex implementation itself from concurrent access.

Data inconsistencies can't occur because the critical sections are protected by the mutexes or semaphores from concurrent access.

Longer latencies can only occur, when a second task wants to access the same critical section or another section protected by the same binary semaphore. This situation is discussed in section 4.6.1.

So seen from the point of latencies this seems to be a good solution.

But there are some problems, that can occur when using mutexes or semaphores. These problems are discussed hereafter and possible solutions are presented:

- The *kernel latency time KLT*, discussed in section 4.6.1 on page 66.
- *Priority inversion* problems, discussed in section 4.6.4 on p.69.
- Possible *deadlocks*, if semaphores are nested, in section 4.6.6 on p.73
- Preemption locks protecting critical sections that are accessed by interrupt handlers can't be easily changed into mutexes, since interrupt handlers are not allowed to sleep. This problem is tackled in section 4.6.7 on p.76.

### 4.6.1 The Kernel Latency Time (KLT)

In the case of a non-preemptible kernel the longest execution time of a kernel routine contributes to the worst case PDLT. When critical sections are introduced into such a kernel, which is made preemptible in general after that apart from the critical sections, the longest kernel induced latency is reduced to the execution time of the longest critical section. That way normally the worst case PDLT is reduced, but as [58] has pointed out, shared mutexes protecting the critical sections can lead to a further latency, called the KLT, the kernel Latency:

Thread A in fig.4.5 executes a system call which contains several critical sections protected by different mutexes. While thread A is executing code in a critical section, i.e it holds the mutex `S2`, an interrupt occurs and the interrupt service routine awakens thread B which has a higher priority. So thread A is preempted right after the interrupt service routine while still holding the mutex `S2`. The scheduler starts the execution of thread B.

If the system call were protected by a general preemption lock, not by mutexes, then the PDLT would be much longer because the scheduler could not be invoked during the whole system call.

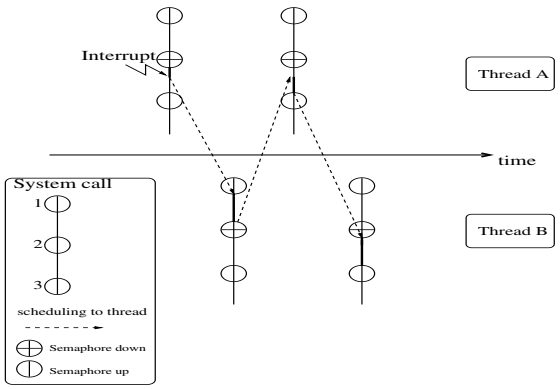



Figure 4.5: *The Kernel Latency Time, KLT, [58]*

So the conversion of general preemption locks into mutexes leads to a shorter PDLT, because not only in user mode but also in kernel mode the kernel is preemptive.

Nevertheless, in the rare case, that thread B calls the same system call - or a critical section protected by the same mutex `S2` - as thread A did before, thread B will be put asleep, when it tries to enter the mutex that thread A currently holds.

The scheduler executes and thread A comes to execution again, finishes the critical section and releases the mutex `S2`. Releasing the mutex `S2` reawakens thread B that is now allowed to get hold of `S2` and to enter the critical section. The scheduler is executed again and thread B comes to execution and finishes the execution of the system call. The whole time thread B has to sleep until thread A - the thread of lower priority - finishes the critical section and releases the mutex is called **the 'kernel latency time', the KLT**. The KLT includes the two times the scheduler executes, so the following formula applies:

$$KLT_{max} = 2 * T_{scheduler} + T^{max}_{criticalsection} \tag{4.2}$$

The kernel latency time only occurs in preemptive operating systems which contain critical sections protected by mutexes or semaphores. So one sees that the use of mutexes shortens the PDLT, but it causes also a new latency called KLT to arise in rare cases. Of course these cases are rare and so on the whole such an operating system shows less latencies than one with general preemption locks.

If there is no priority inheritance protocol implemented in the mutexes, then the KLT can be much longer, see section 4.6.4.

The fact, that a KLT can still occur, means that a transformed Linux kernel, that uses such a technique shouldn't be more hard realtime capable than standard Linux, because in this rare worst case it could behave similar to standard Linux. Its soft-realtime capabilities of course are better than standard Linux.

Therefore the so called hard realtime capabilities of the Timesys Linux 2.4.7 kernel should be checked carefully. To get hard realtime capabilities one must cut long critical sections of the standard Linux kernel into pieces of shorter critical sections. This is the only way to really avoid long latencies in every case.

### 4.6.2 Complex KLT

The complex KLT arises from the fact that one system call can contain several critical sections that can in a disadvantageous case be held by different processes.

In this case, shown in [58], the fulfilling of the execution of a system call can be delayed at the most for the duration of the execution of all `m` critical sections it contains plus the time of some scheduler runs and context switches of the n processes working on the same semaphores causing the KLT.

$$KLT_{max} = 2(n-1) * T_{scheduler} + \sum_{i=1}^{m} T^{max}_{criticalsection} \tag{4.3}$$

So every operating system with many or long critical sections will obviously cause longer latencies in executing time-critical tasks.

Of course, for average performance many small critical sections are better than only a few longer ones, because only rarely all the latencies of the smaller critical sections arise in the same system call causing a big worst case latency.

### 4.6.3 Mathematical modeling

From the existence of the KLT, see section 4.6.1, a simple mathematical model can be derived, that shows in which cases it is useful to replace global preemption locks -or spinlocks in the SMP case- by mutexes or mutual exclusions.

This mathematical model compares the latency a critical section causes on average when it is non-preemptible compared to the latency of the same critical section protected by mutexes or binary semaphores. The possibility that the kernel latency occurs has been taken into account, but not possible overhead due to priority inheritance protocols and deadlock avoiding protocols. But as section 4.6.4 shows, the latter phenomena are of an higher order and thus occur statistically less frequent.

The critical section with the execution time $T_{critical-section}$ must be protected from concurrent accesses by several tasks. The probability $p$ that an interrupt occurs and awakens a process of higher priority $B$, while there is a process of lower priority $A$ working in the critical section - see fig.4.5 -, shall be constant during the execution time of the critical section.

The aim is to minimize the average delay, the process $B$ has to wait until process $A$ finishes the execution of the critical section. That's an optimization for soft-realtime tasks.

If global preemption locks or spinlocks are used to protect the critical section, then the maximum $delay_{critical-section}$ of $B$ will be $T_{critical-section}$, and the average delay will be $T_{critical-section}/2$, because at average $B$ awakens when $A$ already executed half of the critical section.

If mutexes are used, that allow $B$ to preempt $A$ during the execution of the critical section, then there will be only a $delay_{critical-section}$ in the conflict case, that the process $B$ of higher priority wants to enter into a critical section, which is protected by the same mutex, that is currently held by $A$. The probability for this conflict case shall be denoted as $p$.

In this conflict case, the KLT occurs, which has been described in section 4.6.1. Eq.(4.2) shows the maximum KLT that can occur. The average KLT is

$$KLT_{average} = 2 * T_{scheduler} + \frac{T_{critical-section}}{2} \tag{4.4}$$

But the $KLT_{average}$ only occurs with the probability $p$ of the mutex conflict case.

So in order to minimize the average delay time, caused by a critical section, it makes sense to replace a global preemption lock by mutexes in the case that

$$delay^{preemption-lock}_{critical-section,average} > p * KLT_{average} \tag{4.5}$$

$$\frac{T_{critical-section}}{2} > p * (2 * T_{scheduler} + \frac{T_{critical-section}}{2}) \tag{4.6}$$

$$T_{critical-section} > 4 * T_{scheduler} * \frac{p}{1-p} \tag{4.7}$$

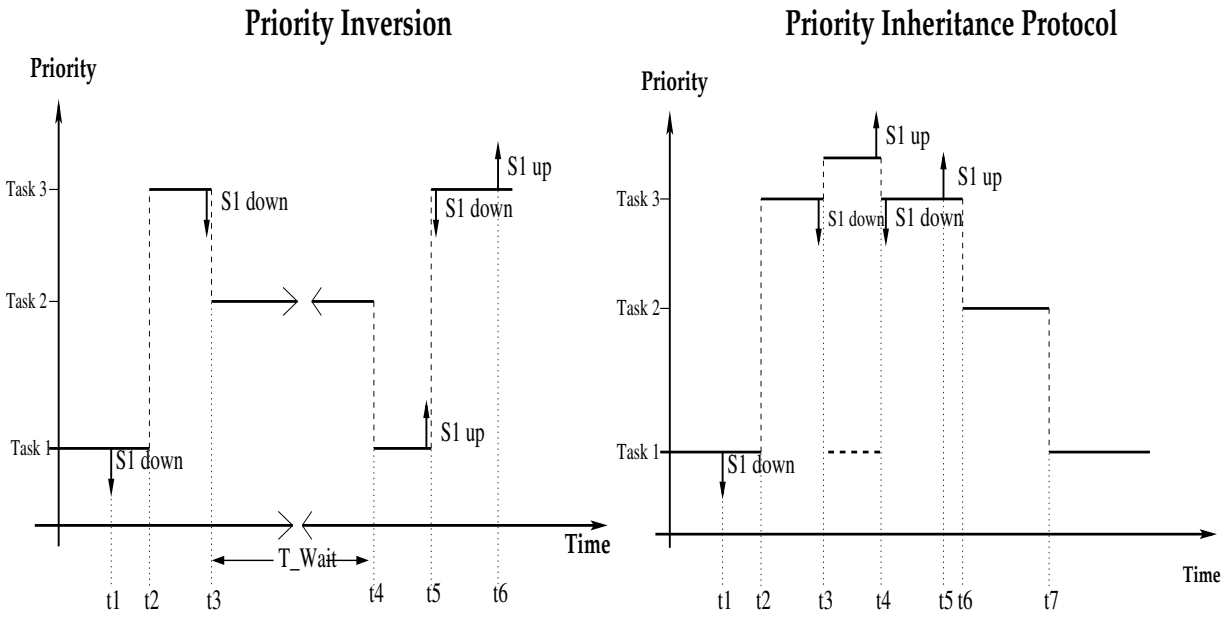



Figure 4.6: *The left figure shows the situation of priority inversion and the right figure shows the same situation when the operating system provides mutexes with a priority inheritance protocol.*

If $T_{scheduler}$ is assumed as constant as it is in Linux 2.6, it only depends from the probability $p$ of the mutual exclusion conflict case and the execution time of the critical section $T_{critical-section}$ itself, whether replacing of a global preemption lock by a mutex is useful for a critical section in order to decrease the average delay it causes.

The most useful this will be for long critical sections which are not accessed too frequently, so that $p$ isn't too big. It's not useful to protect critical sections by mutexes that have a very short execution time, but are used very often.

For instance if for $p = 1/3$ replacing of the spinlock will be useful if $T_{critical-section} > 2 * T_{scheduler}$. For $p = 1/5$ already $T_{critical-section} > T_{scheduler}$ is enough.

Real systems can be more complex than this easy model, for instance the execution time of the critical section $T_{critical-section}$ will vary, if there is work to do with dynamical data structures like lists in the critical section.

Nevertheless, the model shows the principle: The longer the execution time of the critical section, the more the average delay of the high priority process $B$ caused by the critical section can be reduced by a mutex.

Another important conclusion from fig.4.5, eq.(4.6) and (4.7) is: The maximum overhead caused by replacing preemption locks by mutexes in a critical section is $2 * T_{scheduler}$, two times the execution time of the scheduler.

### 4.6.4 Priority Inversion - Starvation

Another important issue one has to deal with when using mutexes or semaphores is a situation called *priority inversion.*

According to the left part of fig.4.6 a situation called priority inversion can occur, if a task of lower priority like task1 holding a mutex is preempted by a task of higher priority like task 3 in the left part of fig.4.6 that requests the same mutex `S1` and is therefore put asleep. If after that a task of middle priority is runnable, when task 3 is put asleep, this task can run for a long time, preventing task 1 from executing any further so that task 1 can't release the mutex. That way task 2 also prevents the task 3 of higher priority from proceeding. This situation is called 'priority inversion'. It leads to a starvation of the high priority task 3 for the time $T_{Wait}$, task 2 of medium priority is running. In Linux task 1 could

be a process or thread scheduled with the SCHED_NORMAL policy while task 3 must be scheduled by the fixed priority scheduler - SCHED_FIFO - as a soft realtime process. In between task 1 and task 3 there could be even more unrelated tasks of intermediate priority that cause priority inversion, so it is sometimes called 'unbounded priority inversion', because one can't say in general how long the starvation will last.

Priority inversion problems are of a higher order than the kernel latency KLT, discussed in section 4.6.1, since the KLT occurs when 2 tasks want to acquire the same mutex, while a priority inversion situation requires at least 3 processes. In the end of section 4.6.5 priority inversion situations are described where even more processes interact using semaphores, thus being of a higher order. In general such latencies of higher order, that require more tasks to interact in a special unfavorable manner, occur less frequent, but if they occur they bear the risk of longer latencies.

### 4.6.5 Priority inheritance and priority ceiling protocols

To avoid priority inversion situations several protocols for mutexes and binary semaphores have been proposed.

#### A simple priority inheritance protocol and its limits

To avoid priority inversion at first a simple priority protocol (SPIP/EPV2) is presented, which is according to [86] used in VxWorks and many other Realtime operating systems:

- Simple Priority Inheritance Protocol (SPIP/EPV2):
  If a task $P_3$ requests a mutex, that is currently held by a task $P_1$ of lower priority, then the priority of $P_1$ is raised up to the priority of $P_3$. As soon as a task released all previously held mutexes, it gets back its initial priority.

In the right part of fig.4.6 the way a priority inheritance protocol works is shown. Slightly different from the SPIP/EPV2 protocol Task $P_1$ is raised to a priority level even one above the priority level of Task $P_3$ in the right part of fig.4.6.

Using such a simple priority inheritance protocol simple priority inversion situations with only 2 tasks of fixed priority involved can be avoided and again the formulas 4.2 and 4.4 apply for the time the high priority task $P_3$ has to wait until task $P_1$ has left the critical section protected by mutexes in the right part of fig.4.6. This $KLT$ latency can't be avoided, if mutexes or any kind of mutual exclusion like semaphores have to be used to avoid race conditions.

This simple protocol works as long as every process holds only one mutex at a particular time, and not more than one. So it already solves a lot of priority inversion situations.

If a process holds more than one mutexes at a time, even with this protocol priority inversion can occur as the following examples show:

1. The first problem is that a process holds the highest inherited priority until it releases all reserved mutexes:

   In these examples processes with a higher index have a higher priority. The process $P_1$ acquires the mutexes A and B. $P_3$ of higher priority starts working, requests mutex B and is put asleep. The priority of $P_1$ is raised to the priority level of process $P_3$. If $P_1$ releases the mutex B, it nevertheless runs at the higher priority of $P_3$ until it releases also mutex A. This way $P_1$ can prevent a third process $P_2$ of middle priority from running for a certain time.

2. Chained mutexes can still produce priority inversion with SPIP/EPV2:

   A set $M$ of mutexes is called 'chained', if there is a series of processes $P_i$ with $(i \geq 0)$, so that every process $P_i$ already holds a mutex of $M$ and all processes except $P_0$ request furthermore one mutex out of $M$, that is currently hold by $P_{i-1}$.

The following example in fig.4.7 clarifies the problem of chained mutexes: Again processes with a higher index have a higher initial priority. At first process $P_1$ acquires mutex A. $P_2$ gets ready to work, preempts $P_1$ because of his higher priority and acquires mutex B. Now $P_2$ requests mutex A and is put asleep. $P_1$, the owner of A, inherits its priority and proceeds. Then the high priority process $P_4$ gets runnable and requests mutex B and is put asleep. The priority of the sleeping process $P_2$, as the owner of B, is raised to the priority level of $P_4$, but process $P_1$, that blocks $P_2$, is not raised. Now every process $P_3$ preempts $P_1$ and therefore indirectly also delays the high priority process $P_4$. So a better priority inheritance protocol must be transitive.

Like the Non-preemptive critical section (NPCS) protocol also priority inheritance (PI) protocols need no information about resources tasks will request in advance.

But when using a PI-protocol a further rule has to be taken in account to avoid deadlocks, this is discussed in section 4.6.6, p.73. On the contrary when using NPCS deadlocks cannot occur and no further rule treating deadlocks has to be considered.

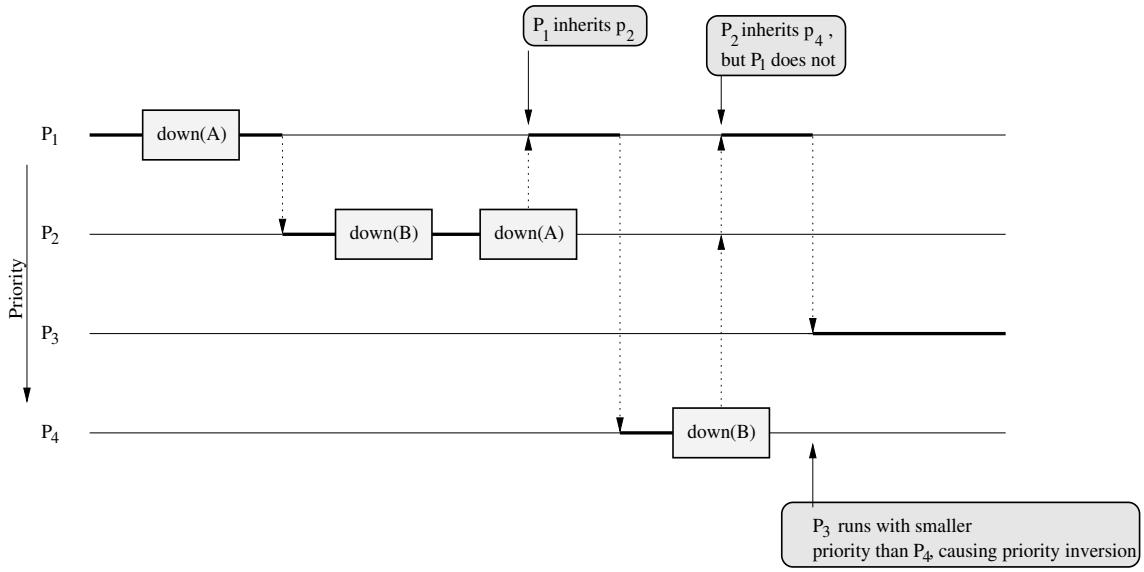



Figure 4.7: *Priority inversion situation using chained mutexes*

### Priority inheritance protocols for more complex situations

A more sophisticated protocol is needed for priority inversion situations where one process can hold more than one mutex at a time.

Such a protocol has been developed 1990 in [91]. The proposed protocol (PIP_Yod2) extends the simple protocol (SPIP/EPV2) of section 4.6.5 as follows:

1. When a process releases a mutex, it is put back to the priority it had just before it requested this mutex.
2. The priority inheritance is transitive. Assume a process $P$ owns a mutex A and waits for a further mutex B, that is currently hold by process $P'$. Every process that requests A and raises the priority of process $P$, must raise also the priority of $P'$ and all other processes that indirectly block process $P$.

But also this protocol is not perfect, as the following counter-example shows, that Victor Yodaiken published in 2001 [86]:

The problem is the first point:

A process $P_1$ with low priority holds mutex A and B. Then process $P_2$ with a higher priority comes to execution, requests mutex B and is put asleep. $P_1$ inherits its priority. Then a high priority process $P_3$ requests mutex A and is blocked. $P_1$ now inherits the priority of $P_3$ and calculates until it can release mutex B. But according to the PIP protocol now $P_1$ is put back to its initial low priority, while still holding mutex A and blocking $P_3$. Because $P_1$ released mutex B, $P_2$ is reawakened and preempts $P_1$, so that again there is a priority inversion situation.

Prof. Victor Yodaiken repairs the protocol of [91] in the following way:

A process, that releases a mutex, is set to the highest priority, that it has inherited from all mutexes still held.

An implementation of such a strategy must hold for every process a set, that consists of pairs like (mutex, highest inherited priority).

When a process $P'$ holds a mutex A and a process $P$ requests the same mutex, blocks and hands down its priority $p_A$ to the process $P'$, then the pair $(A, p_A)$ is added to the set of $P'$. If there is already a pair $(A, p)$ in the set of $P'$, then it will be updated, if the new priority $p_A$ is higher than the older one. When process $P'$ releases mutex A, then the pair containing A will be removed, if it exists, and the priority of $P'$ is set to the highest remaining priority of its set.

This strategy avoids priority inversion, even if a process holds more than one mutex at a time.

To implement this protocol, further details had to be specified: It is not fixed at which priority a process is set, after it released its last mutex. Furthermore, it must be specified at which conditions a new pair $(A, p_A)$ is added to the set of a process, when it inherits a new priority.

The following could happen: Again the priority of the processes shall increase with their index. Process $P_1$ holds the mutexes A and B. Process $P_3$ preempts process $P_1$, requests mutex A, blocks and hands its priority to process $P_1$. Therefore, a pair is added at the mutex set $M$ of $P_1$. Now the process $P_2$ gets ready, but doesn't get the processor, because $P_1$ inherited the high priority of $P_3$. If now $P_1$ performs a blocking operation, it is put asleep for a certain time. Then $P_2$ starts running, requests B and is put asleep. In this case a new pair of the form $(B, p_2)$ must be added to the set of $P_1$, because after releasing mutex A, $P_1$ must be set to the priority of $P_2$.

Although a new pair has been added to the set $M$ of $P_1$, its current priority, which is higher than that of $P_2$, is not changed.

If the initial priority of $P_2$ were lower than the initial priority of $P_1$, then no new pair would have been added to the set of $P_1$, to avoid that $P_1$ would be set to a lower priority than its own initial one when releasing the mutex A.

Therefore, there are 3 different possibilities when a process $P$ with priority p requests for a mutex A held by the process $P'$.

- The priority of $P'$ must be raised to the priority p of $P$ and a new pair (A, p) is added to the set $M$ of $P'$
- It's only necessary to add a new pair (A, p) to the set $M$ of $P'$
- The priority and the set $M$ of $P'$ remain unchanged.

To define a protocol as proposed by Victor Yodaiken [86], the notion of the initial priority has to be defined.

The initial priority of a process is the priority it has just before it claims its first mutex for the first time.

The protocol (PIP_Yod2) implemented hereafter assumes that this initial priority is never changed while the process holds any mutex except for priority inheritance reasons as defined by the protocol. After releasing all mutexes the process will get back its initial priority. Changes of the initial priority for other than priority inheritance reasons are only allowed, while a process holds no mutex.

Using this agreement the following priority inheritance protocol (PIP_Yod2) can be defined:

- Every process P has a set S(P), that contains pairs of the form $(M, p_M)$. If a process $P_2$ with priority $p_2$ requests a mutex M, that is currently held by a process $P_1$ with the lower priority $p_1$, then the priority of $P_1$ is raised to that of process $P_2$. If $p_2$ is higher than the initial priority of $P_1$, then a distinction has to be made: If the set $S(P_1)$ of the low priority process $P_1$ does not contain an element with first component M, then a new element $(M, p_2)$ is added. If on the other hand such an element $(M, p)$ already exists for mutex M, then it is replaced by $(M, p_2)$ in the case that the condition $p_2 > p$ is true.

  The priority inheritance is transitive: If a process $P_2$ inherits the priority $p_3$ of a process $P_3$, and if the process $P_2$ has to wait for another mutex currently held by a further process $P_1$ of lower priority, then $P_1$ also gets the priority $p_3$ from process $P_2$.

  When a process P releases a mutex M there are 3 possibilities. If the set S(P) does not contain an element with first component M, nothing is done. If there is such an element it is removed from the set. After that it has to be distinguished whether the set S(P) is empty or not. If it's empty, the process P gets back its initial priority. Otherwise, the process P gets the highest value of the second components of the elements of S(P), i.e. the highest remaining priority in this set.

#### Priority ceiling protocols (PCP)

The priority ceiling protocol avoids priority inversion by raising the priority of a process, that requests a mutex, to the highest priority of all threads that ever use this mutex.

To find out the highest priority static analysis of the whole system is necessary. This is only applicable or cost-effective for systems with a limited number of processes and limited dynamic operations. Static analysis is much easier to do when the scheduler only assigns static priorities to tasks, that don't change during execution. Furthermore, it must be known before, at which time which task requires which resource.

Alternatively, one could set the process to the highest priority in the system while holding the mutex. This requires less analysis. In both cases this process could prevent unrelated processes of high priority from running.

Compared to Priority Inheritance Protocols (PIP) Priority Ceiling protocols (PCP) sometimes prevent a task from accessing a resource, even if that resource is free. This additional locking of a resource sometimes is referred to as 'avoidance blocking', since this inhibit deadlocks.

The maximal blocking time of PC protocols is higher than with PIP. In average the blocking time is a little higher, but this avoids deadlocks as PCP defines an order of access for the resources available.

Since Priority Ceiling (PC-) protocols are much easier to implement with static priorities and require a static analysis of the system before, they aren't the right choice to implement in a standard operating system.

### 4.6.6 Possibilities to avoid deadlocks

At first a simple deadlock situation shall be shown:

If mutexes or binary semaphores are nested, then in between 2 tasks a situation called *deadlock* can arise. An example for a deadlock is written in pseudo code:

```
DCL Sema1, Sema2 Semaphore;
Relais TASK;
        REQUEST Sema1;
        REQUEST Sema2;
        .....
        RELEASE Sema2;
        RELEASE Sema1;
END; /* TASK Relais */
```

```
Control TASK;
        REQUEST Sema2;
        REQUEST Sema1;
        .....
        RELEASE Sema1;
        RELEASE Sema2;
END; /* TASK Control */
```

The *deadlock* can be avoided, when semaphores are always requested in the same sequence. Since this is not always the case in a big kernel, some mutex implementations implement a deadlock recognition algorithm that refuse to acquire a semaphore as long as a deadlock would occur, cp. the next section or [67, 94, 73].

## Preconditions to a deadlock

According to [94] deadlocks can only arise, if several preconditions are fulfilled:

1. Exclusively used resource: A resource can be used only exclusively by one task, an example is a printer, where only one print job can be printed at a given time.
2. Resource can't be deprived: Other tasks can't deprive a task of a resource, it has already reserved. They must wait until the task releases the resource voluntarily.
3. Incremental request: A task, that already has reserved a special resource, can claim other resources.
4. Circular waiting: There must be a circular chain of tasks, each requesting a resource, that the succeeding task in the chain has already reserved. An example is presented in the next paragraph.

A possibility to avoid deadlocks is to exclude that at least one of the conditions above can ever come true.

## Deadlock detection and how to avoid them

From the preconditions above it follows that several possibilities exist to avoid deadlocks:

The first condition - a resource that can only used exclusively - is a quality of some resources that can't be changed, the easiest example is a printer currently printing a document.

### Treatment of deadlocks at runtime

The second condition - that other resources can't deprive a task of a resource - is difficult to change, because it means that a deadlock that already occurred or is about to occur and has been detected by the software has to be undone. This can only done dynamically by interfering in the normal mode of operations at runtime and thus leads to performance losses.

The problem of deadlocks caused by chained mutexes can be modeled using a directed graph. When a task reserves a resource, an arrow, a directed edge, from the resource to the symbol is written. Before a resource is reserved, it is checked whether this reservation would produce a cycle in the graph. If so, this reservation would cause a deadlock and the reservation has to be delayed until other tasks free their resources, so that the deadlock is avoided. This algorithm can be performed using matrix operations and thus an analysis can be done by a computer, but it consumes some time. Furthermore, delaying a task means a latency.

The Banker's Algorithm [94] detects cycles in advance, before they are completely closed. It then provides a scheduling of tasks that prevents deadlocks. The state of the system is free of deadlocks, as long as one task can be found, that is able to reserve all resources it requires. If such a task can't be found, a deadlock can occur. In the worst case a task is blocked by this algorithm, until all resources have been freed again. Therefore the blocking time is the sum of the execution times of all critical sections of the other tasks.

If a deadlock occurred or is about to occur one possibility is to cancel a process. This means of course a waste of performance and work already done. Another possibility is to store at regular checkpoints the current state of a process, so that it is possible to restore a state of the process before the deadlock occurred, if a deadlock occurs. A problem can be, that operations of the process in the operating system have to be undone, which is not always possible and useful. This means a huge operating expense and thus isn't done normally in operating systems, especially not if they shall meet soft-realtime requirements.

An even more important obstacle to apply the Banker's Algorithm in operating systems than its possible latencies is that it needs to know in advance all resources of the system, the distribution of already allocated resources that already belong to the tasks of a system and the maximum requirements of resources, that can be done by every task during their lifetime. This last piece of information is only known for small or at least predictable systems, not for an entire operating system with user-defined tasks. Thus the Banker's Algorithm isn't used in standard operating systems.

#### Possibilities to avoid deadlocks at compile time

Nevertheless, there are more easy possibilities to avoid the other two preconditions. These mechanisms are done while programming, so at compile time. Therefore they don't cost performance at runtime, which makes them suitable to be deployed in an operating system.

- With regard to the third condition: The problem of incremental request means for the use of semaphores, that deadlocks can only happen, if semaphores are nested.

  One possibility to avoid deadlocks in this case is to request all resources or semaphores at once. So if a task can get hold of all of them at once, it reserves them all, if not, it releases them all and tries again later on. For instance, this possibility is offered by the Win32 API and by some Linux semaphore implementations `sem_op` for application programmers. This avoids incremental request. Small disadvantages are that multiple tries can need more time and that resources are also reserved longer, if they are held until all of them can be freed.

- With regard to the forth condition: Another possibility is to request resources always in the same linear order. This avoids circular chains and thus circular waiting. To do this a total linear ranking of all resources - or at least at all resources that can ever be requested by one task - is necessary. Small disadvantages are that programmers must be aware of this linear ranking of resources, e.g. semaphores, for instance when holding a semaphore and calling a subroutine that also reserves any semaphore.

### Debugging and avoiding deadlocks in the Linux kernel

In operating systems also data structures of the operating system can be considered as resources. For instance, operations on a linear list must be protected by a semaphore or mutex in order to avoid data inconsistencies, that could occur when two tasks work in the same critical section at the same time, cp. section 4.5.

Since the Linux kernel contains nested semaphores or mutexes, deadlocks could also occur in the Linux kernel.

Therefore, kernel developers implemented a configuration option at compile time in the Linux kernel that includes a deadlock detection in the Linux kernel. If a deadlock occurs in the kernel, the deadlock detection code prints a warning message into a log file. The kernel developers then search the semaphores or mutexes, that caused the deadlock, by hand in the source code of the Linux kernel. After the position

has been found, they try to reorder these semaphores in a way, a deadlock can't occur any more. This operation requires rather skilled and well-educated programmers. This means the Linux kernel avoids deadlocks by maintaining a linear ranking of semaphores or mutexes in the kernel.

### 4.6.7 Handling of interrupt handlers that shall make use of mutexes

As stated at the beginning of section 4.6, it is possible to replace all those preemption locks with mutexes, which are never entered by any code from an interrupt service routine (ISR) or a SoftIRQ/Bottom Half. But it is not possible to replace those preemption locks with mutexes, which at the same time also lock interrupts, because such critical sections are used also by interrupt handlers.

The problem is that a mutex puts asleep the caller, if it has already been locked by any other task before. For an interrupt handler sleeping is forbidden by definition, at maximum it's allowed to execute a short busy wait when accessing an already locked spinlock on a multiprocessor system.

#### Transforming interrupt handlers into kernel threads

A possible solution for this problem is to make each interrupt handled by its own kernel thread. This solves the problem, because kernel threads are handled by the scheduler and thus they are allowed to sleep, while a classical interrupt service routine is not. Since these kernel threads that execute the former interrupt handlers can sleep, they can access mutexes. If a mutex is already occupied, the kernel thread is put asleep, until the task that locked the mutex left this critical section and the mutex implementation reawakens the kernel thread. Since the kernel threads that handle interrupts get a very high priority, probably the highest possible in the system, the scheduler will normally elect this kernel thread to be executed on the processor immediately, again.

When the operating system boots, for every assigned interrupt an own kernel thread is started. The normal interrupt routine is assigned to it as thread function, its priority is raised to a high soft-realtime priority and then it is put asleep.

When an interrupt comes in, the general interrupt service routine stub wakes up the appropriate kernel thread, that executes the interrupt service routine (ISR). After being brought to execution on the processor by the scheduler, the interrupt service routine is executed. After that the kernel thread is put asleep again. Of course, interrupts not used can be assigned later dynamically, which will start a new kernel thread, serving the interrupt.

An advantage of such a threaded IRQ-handling is the possibility to assign soft-realtime priorities to interrupts and thus to prioritize a real-time task higher than a special interrupt, if a time-critical application demands that. Also interrupts can be prioritized individually.

An application using such a concept could be a GSM handy, where signal processing of incoming or outgoing messages due to the Time division multiplex (TDMA) of GSM is even more important than any low level interrupt.

A patch for the Linux kernel implementing this concept has been developed in this thesis [27], based on parts of the code of the Timesys GPL Linux kernel 2.4.7 [96]. Section 6.2.4 on p.93 contains details of the patch implementation for Linux.

### 4.6.8 Read-Write Locks

Read-Write locks are used to protect data that is more often read than written. If such a critical section were protected by a normal preemption lock, only one task could enter the critical section either reading or writing to it.

But since data inconsistencies can only occur, if there is a writer, multiple readers reading the data of the critical section in parallel can be allowed. On the other hand any writer must use the critical section exclusively in order to avoid data inconsistencies caused by a read-write or a write-write conflict.

For this sake read-write locks have been created, allowing multiple task to read data in parallel, but granting exclusive access to tasks willing to write.

In the standard Linux kernel exclusive writing access to such critical sections is provided by a preemption lock. When creating a fully preemptible Linux kernel, these read write locks can be converted into (counting) semaphores that distinguish between reading and writing access to the critical section.

The implementation of the read-write locks in this thesis is similar to the implementation of mutexes. Details of the implementation of read-write locks developed and published in this thesis are listed hereafter:

- The maximum number of tasks that are allowed to read a critical section in parallel is limited. In this implementation [39], the maximum is 4 readers. For time-critical tasks this is important, as a writer has to wait until all readers have left the critical section, before access is granted to it exclusively. Limiting the number of readers is the way to avoid delays caused by read-write locks that are described in [104].
- Readers can enter the critical section until the maximum number of allowed readers is reached. Further readers requiring the read-write lock will be blocked and queued in a waiting queue. **Only one** reading task of lower priority holding the lock inherits the higher priority of the blocked reader. That way, a reader of high priority will be able in general to acquaint the read-write lock soon.
- A writer can acquire the lock only if there is no reader or other writer holding the read-write lock currently. A writer locks the critical section exclusively, so all other tasks are blocked until the writer unlocks the read-write lock.
- The read-write lock is given to the waiting task of highest priority.
- Waiting writers are not preferred to readers: Waiting writers of lower priority have to wait, when readers of higher priority want to acquire the lock. But when a writer is waiting new readers of lower priority are blocked. **Every** reader holding the lock inherits the priority of a blocked writer, if the writer's priority is higher. That way the writer can enter the read-write lock according to its priority as soon as possible.

The read-write locks declared above have been implemented and tested in the Linux 2.6.9 kernel in this thesis and published using the GPL2 license [31, 39, 30].

Timesys implemented a similar concept, as can be seen when analyzing the source code of the fully preemptible Timesys GPL kernel 2.4.7, i.e. Timesys 3.1. Only the Timesys priority inheritance protocol is commercial and its source code is encrypted.

### 4.6.9 Discussion - Fully Preemptible Kernel or not

There are several further conditions, that have to be respected as Victor Yodaiken states in [104]:

Critical sections should not contain blocking operations.

As we have seen, a priority inheritance protocol that avoids priority inversion, even if the operating system contains nested mutexes - as the Linux kernel does - must be transitive, as the presented protocol PIP_Yod2 is. That way, [104] states that such a priority inheritance protocol makes an operating system much more complex, and complex inheritance algorithms are time-consuming. Therefore, priority inheritance protocols are not suited to implement hard realtime operating systems (OS), if these operating systems make use of nested mutexes. The best way for such a hard realtime OS is to use preemption locks, i.e. NPCS, and to keep such non-preemptive critical sections as short as possible. This method can be applied to standard operating systems, too.

For operating systems running tasks that have to fulfill no special timing requirements or only soft realtime requirements - as standard OS do - a fully preemptible kernel relying on mutexes with priority inheritance might be a possibility. Some of the problems presented in [104] can be solved:

For instance, as we have seen in section 4.6.8 the number of readers in a RT operating system that are allowed to read such a critical section simultaneously must be smaller than a limit, for instance limited to the maximum of 4 or less tasks. That way, if a writer of high priority blocks, all readers can inherit its high priority without significant performance loss, if the critical section isn't that long [96].

Although [104] argues from a theoretical point of view, priority inheritance shouldn't be used in operating systems, he admits, that even Sun Solaris uses read / write locks with priority inheritance protocol, and during this thesis there have been 3 experimental implementations of a fully preemptible Linux kernel, as documented in section 7.5.4. Furthermore, also hard realtime OS like VxWorks implement a simple priority inheritance protocol similar to SPIP/EPV2 at least for semaphores and mutexes in user space, i.e. used by applications.

Therefore, the development in this field is still open and next years will show, whether mutex based fully preemptible kernels will become a part of standard operating systems of today to serve soft-realtime tasks. At least, as discussed in section 7.5.4 different implementations of a fully preemptible Linux kernel exist meanwhile. These implementations are tested by people belonging to the Open Source Community or to industry world-wide. One implementation is maintained [66] and thus a candidate to be a configurable part of a future standard Linux kernel, cp. section 7.5.4, p.117.

# Chapter 5

# Improving Linux to be more suitable for time-critical applications

## 5.1 Soft-realtime requirements for the Linux kernel

This paragraph outlines briefly the differences between a standard operating system and a realtime operating system when processing time-critical tasks. Furthermore, the capabilities of the Linux kernel regarding realtime requirements are analyzed.

Linux has been chosen in this thesis, because the source of the operating system is provided, can be analyzed, changed and the changes can be redistributed under the terms of the GNU Public License (GPL), currently version 2 [15]. The GPL license and its derivatives, e.g. the LGPL, are the heart of the Open Source Community.

Therefore, choosing Linux it is not only possible to measure its capabilities regarding time-critical tasks, but additionally the source code of the operating system can be analyzed to understand, explain and state the temporal behaviour of the Linux kernel.

Apart from that, since changes of the Linux kernel code are allowed by the GPL2, the Linux kernel offers a very good chance to apply, implement and testify concepts that can lead to a better temporal behaviour of the Linux kernel and thus make Linux more compliant to realtime requirements.

The first step is to analyze the Linux kernel regarding realtime requirements. This thesis covers the development of the Linux kernel from version 2.2 up to version 2.6.

In the following features are listed that standard Linux 2.6 already provides and that are necessary, but not sufficient premises to process time-critical tasks in Linux.

| | SCHED_NORMAL | SCHED_FIFO/SCHED_RR |
|---|---|---|
| fixed priority scheduler | no | yes |
| ISR $->$ wake up user processes | yes | yes |
| Realtime Clock with high precision | no | no |
| IPC Kernel/User | yes/yes | yes/yes |
| Shared Memory for processes | yes | yes |
| Memory Locking | yes | yes |
| files resident in RAM | yes | yes |
| E/A using Busy Waiting & Interrupts | yes | yes |
| Partly Preemptible kernel | yes | yes since Linux 2.6 |
| Global Interrupt locks in the kernel | yes | yes |

Table 5.1: *Capabilities of Linux regarding soft realtime requirements*

- Linux is a multitasking operating system. So time-critical tasks and tasks without timing requirements can run in parallel.

- Linux provides two fixed priority scheduler policies that can be assigned to individual processes or threads. Linux provides 99 priority levels and all these tasks are preferred to normal tasks using the standard UNIX Round Robin scheduler. Basically the fixed priority scheduler policies in the Linux kernel are appropriate for time-critical tasks.

- Linux provides a thread concept that contains kernel supported threads, also called a 'One-to-One' thread implementation concept. This means Linux provides kernel supported threads, i.e. the kernel schedules threads and processes in the same way and no time-consuming multi-level scheduling is needed. Therefore the Linux thread concept is appropriate for time-critical tasks.

  Advantages of using threads are that the context switch in between different threads of the same process is very fast, since the address space hasn't to be changed. Furthermore, the Translation Lookaside Buffer (TLB), that stores the most recent memory address transformations, isn't invalidated and hasn't to be refilled for every new thread of the same process. after every context switch. Also, global variables for threads are easy to implement, although they must be protected by a synchronization concept like threadsafe semaphores or mutexes.

- Linux provides - as most UNIX operating systems - interprocess and interthread communication mechanisms like signals, message queues and shared memory. Furthermore, also synchronization primitives like binary and counting semaphores are provided. These basic mechanisms for time-critical tasks can be inspected using the Rhealstone benchmark, cp. section 3.6.2, p.39. Linux semaphores or other synchronization primitives currently don't implement protocols to solve priority inversion situations. On the contrary, this feature is provided by the most realtime operating systems like e.g. VxWorks [81].

- Applications running in user space can be preempted by a process running at a higher priority at any time. The conditions in kernel mode are discussed in the next paragraph.

Nevertheless, there are some major issues regarding Linux realtime capabilities that should be improved in order to make Linux more realtime-compliant. Such issues are listed hereafter [89]:

- A realtime operating system must have a preemptible kernel, i.e. offer preemptive scheduling, in order to be able to provide short latencies for tasks of high priority. Preemptive scheduling leads to a smaller PDLT.

  Although preemptive scheduling is already implemented for user space applications in Linux 2.2, the standard kernel of Linux 2.2 and 2.4 is non-preemptible on single processor computers. This means, when a time-critical task shall be brought to execution, while the processor is currently in kernel mode, this task of high priority will suffer from a long latency, i.e. from a long PDLT.

  A first concept of preemption has been introduced for the multiprocessor Linux kernel. In the Linux 2.2 multiprocessor kernel critical sections have been introduced, that allow other processors to run the same kernel functions on another processor, if they don't contain a critical section, that is protected by a spinlock.

  Chapter 7, p. 97, is dedicated to the preemptability of the Linux kernel and how it can be improved.

- In Linux - as in nearly every operating system of today - interrupt service routines intercept the currently running process and even the kernel, when the processor is currently running in kernel mode, for instance executing a system call for a task.

  But - as in nearly all standard operating systems - there are interrupt locks also in the Linux kernel code, which lead accidentally to a longer Interrupt Response Time (IRT). Therefore in Linux - as in Windows - even the duration of the IRT is non-deterministic.

  Looking at Linux device drivers the time the interrupts are locked depends sometimes on hardware parameters. The duration of interrupt locks in the core system often depend on the load of the computer system.

  Interrupt locks are discussed and analyzed in detail in section 6.2 on p.90.

- Linux 2.0 up to 2.4 provide only a timer granularity of 10 milliseconds for the Intel platform and 1 millisecond for SUN SPARC and DEC Alpha. Especially 10 milliseconds is not sufficient for all time-critical tasks. Using platform dependent clocks it is in principle possible to get a higher timing resolution. Another possibility is to increase the frequency of the timer interrupt. A disadvantages of this solution is that the performance of the system available to the user decreases.

  Chapter 8, p.119, deals with the Linux timing granularity and how it can be improved.

- As stated above the Linux kernel offers a fixed priority scheduler, that is suited for processing time-critical tasks. Nevertheless, the time used to schedule in Linux 2.2 and 2.4 degrades with the number of tasks ready to run, i.e. the load of the system. This theme is considered in section 6.1 on p.85.

- The bottom halves or SoftIRQs are not prioritized in standard Linux. The concept to prioritize and thus balance time-critical tasks compared to tasklets executed in interrupt context asynchronously, is only rudimental and not sufficient for realtime requirements. Section 6.2 on p.90 deals with this theme.

- Although some parts of the GNU C library `glibc` and other libraries have been implemented following the POSIX standards, the POSIX API isn't fully implemented in standard Linux. For instance concepts of Prioritized I/O are yet not fully implemented in the Linux kernel. This is discussed in detail in section 5.2 on p.81 .

Table 5.1 summarizes the functionality provided by Linux regarding realtime requirements.

## 5.2 Parts of the POSIX standard supported by Linux

The POSIX standard in general has been presented in section 2.3.

The function of the POSIX standard is to standardize the application interface different operating systems provide in a way, that porting the source code of a program from one platform to the other is easy. The POSIX standards define interfaces in the form of C-functions, so that a source code written conform to the POSIX standards can be compiled without changes on all POSIX conform operating systems.

Ideally, if both operating system implemented the POSIX standards fully or at least those parts needed by the application, it would be enough to just recompile the application to get it running on the new platform.

But many operating systems only implement the large POSIX standards only partly, so that slight changes of the source code are necessary, until the application runs and shows the same behaviour on the new platform.

The following standards [108] have been made, also referred to as POSIX.1:

- IEEE 1003.1 System API (C language)
- IEEE 1003.1a System API extensions (symlinks etc.)
- IEEE 1003.1b Real-time & I/O Extensions (formerly POSIX.4)
- IEEE 1003.1c Threads (formerly POSIX.4a)
- IEEE 1003.1d further Realtime extensions (formerly POSIX.4b)
- IEEE 1003.1e bis IEEE 1003.1q further definitions (security, corrections, etc.)

Furthermore, there is also the POSIX.2 Standard:

- IEEE 1003.2 Shell and further programs belonging to the operating system

- IEEE 1003.2a until IEEE 1003.2e

The POSIX standard has been continously advanced since its first publication in 1986. In 2002 it has been revised and merged with the 'Single Unix Specification' of the 'Open Group' and referred to as 'Austin Group Specification'. One of the last changes of POSIX is the 'Open Group Base Specifications Issue 6'. The core of this specification is the IEEE Standard 1003.1-2001 This standard consists of four components:

- `Base definitions:` General notations, concepts, interfaces of the whole standard
- `System interfaces:` System call interface descriptions, subroutines of the programming language C, return types of functions and error processing
- `Shell and Utilities:` Definitions of a source code interface of a command shell interpreter
- `Rationale:` Other pieces of information, that don't fit in the rest of the document structure.

The POSIX 1003.1b standard, that has been referred to as POSIX.4 before 1993, aims at enabling an operating system for soft-realtime tasks.

### 5.2.1 The Linux POSIX Realtime conformity

In this section the conformity of the Linux versions to the existing POSIX 1003.1b standard for soft-realtime tasks shall be analyzed [33].

The POSIX standard 1003.1b-1993 extended the glibc 2.1 library, used with Linux 2.0 and 2.2 by adding the following functions [11, chapter 2]:

The glibc 2.1, used with Linux 2.0 and 2.2 has been compatible to the POSIX.1 and POSIX.2 standards of 1990. The following standard of POSIX.1b-1993 covers the following areas [11, chapter 2]:

- Signals,
- Interprocess communication (IPC) and memory mapped files,
- Memory locking,
- Synchronous I/O,
- Scheduling,
- Asynchronous I/O,
- POSIX macros.

Furthermore, one can state that the following POSIX 1003.1b macros are defined in glib2.1. Thus since this library version, that started working on Linux 2.0 and Linux 2.2, the functionality, that is associated with this POSIX macros by the standard, is provided in Linux:

1. _POSIX_FSYNC[1] ,
2. _POSIX_MAPPED_FILES[2] ,
3. _POSIX_MEMLOCK[3] ,
4. _POSIX_MEMLOCK_RANGE[4] ,
5. _POSIX_MEMORY_PROTECTION[5] ,
6. _POSIX_PRIORITY_SCHEDULING[6] ,

7. _POSIX_REALTIME_SIGNALS[7] ,

The following list shows POSIX macros, that have been partly implemented in the glibc version 2.3 running on the Linux kernel 2.4 and 2.6:

- _POSIX_SHARED_MEMORY_OBJECTS[8]
  `Shared Memory has been supported already by glibc 2.1`
- _POSIX_SEMAPHORES[9]
  `The library glibc 2.3 with NPTL (Native POSIX Thread Library) offers complete support for semaphores.`
- _POSIX_SYNCHRONIZED_IO[10]
  `Since Linux kernel 2.4 the functionality of this macro is fully supported`
- _POSIX_TIMERS [11]
  `POSIX Timer are supported since Linux 2.6`
- _POSIX_ASYNCHRONOUS_IO[12]
  `In the Linux 2.6 kernel there is an implementation which is not fully compatible to the POSIX AIO interface, so that applications must be adapted to that special interface.`
- _POSIX_MESSAGE_PASSING[13]
  `There is still also in Linux 2.6 no implementation of the POSIX message waiting queues, although the code has been already written for that.`
- _POSIX_PRIORITIZED_IO[14]
  `Not implemented in the kernel yet.`

Therefore, the Linux kernel 2.4/2.6 using the standard GNU C library glibc2.3 is still not 100 % compatible to the POSIX 1003.1b standard. Applications that conform to the POSIX 1003.1b standard don't compile and run in all cases on Linux without changes. Nevertheless, kernel 2.4 and 2.6 conform to more parts of the POSIX standard 1003.1b than the older kernels 2.0 and 2.2 did ever before. A good overview about the current Linux POSIX conformity is provided by [74]. Since the 'glibc 2.0' has been introduced, Linux conforms partly to the POSIX 1003.1c realtime standard [15]. In Linux 2.6 the implementation of threads has been redone completely.

But of course, implementing the API does not mean, that the Linux kernel provides also a good scheduling scheme for soft-realtime tasks or a good preemptability or responsiveness to external events. So the next sections and chapters discuss these issues.

---

[1] The fsync(). function
[2] Functionality to map files to memory: mmap, munmap, ftruncate, msync.
[3] API to lock the address space of a whole process into RAM to avoid paging/swapping: mlockall, munlockall.
[4] API to lock memory pages into RAM to avoid paging/swapping: mlock, munlock.
[5] Capability to protect memory: mprotect
[6] process scheduling control: sched_setparam, sched_getparam, sched_setscheduler, sched_getscheduler, sched_yield, sched_get_priority_max, sched_get_priority_min, sched_rr_get_interval.
[7] further signaling functions: sigwaitinfo, sigtimewait, sigqueue.
[8] Shared Memory, shm_open, shm_close, shm_unlink, Yet implemented: ftruncate, mmap, munmap, .
[9] Counting semaphores, sem_open, sem_close, sem_unlink. Already implemented: sem_init, sem_destroy, sem_wait, sem_trywait, sem_post, sem_getvalue.
[10] assures that data of a file are always on the hard disk fdatasync, msync, aio_fsync.
[11] Clocks and timer implemented since Linux 2.6: clock_settime, clock_gettime, clock_getres, timer_create, timer_delete, timer_settime, timer_gettime, timer_getoverrun. Already implemented in Linux 2.4 and before: nanosleep.
[12] aio_read, aio_write, aio_syspend, lio_listio, aio_cancel, aio_error, aio_return, aio_fsync.
[13] Message queues, mq_open, mq_close, mq_unlink, mq_send, mq_receive, mq_notify, mq_setattr, mq_getattr.
[14] Prioritized asynchronous I/O.

# Chapter 6

# Metrological evaluation of the Linux scheduler and interrupt subsystem

## 6.1 The scheduler of the Linux kernel

### 6.1.1 Scheduling Policies - fixed priority scheduling for time-critical tasks

This section deals with the scheduler of the Linux kernel. The scheduler - as an important part of every operating system - is invoked very often.

In standard Linux there are 2 scheduling classes or policies implemented according to POSIX 1003.1b:

- The standard Linux scheduling algorithm, called SCHED_OTHER until Linux 2.4 and SCHED_NORMAL according to POSIX since Linux 2.6, is a Round Robin scheduler, which portions a time slice of the processor to any process. SCHED_NORMAL processes are useless for time-critical tasks, because they can't be prioritized to guarantee that they are run next by the scheduler, after an interrupt occurs. On a Linux system there are often only SCHED_NORMAL processes.
- Time-critical tasks, i.e. soft-realtime tasks, instead should be scheduled using the SCHED_FIFO or SCHED_RR policies, that provide fixed priorities and are preferred to any SCHED_NORMAL process.

The fact that there is a scheduling policy which prefers time-critical tasks to all other task is a precondition that the Linux kernel is able to deal with time-critical tasks at all.

### 6.1.2 Latencies of the Linux Scheduler

The scheduler has a big influence on the behaviour of the system, as it defines what processes are preferred and which are run only as background tasks.

As the scheduler is invoked very often, a scheduler working efficiently also increases the overall performance of an operating system.

Tasks with realtime constraints will have one more requirement: In a realtime operating system the scheduling time, i.e. the time to find the process of highest priority out of a set of processes that are all ready to run, should be constant, independent how many processes are currently ready to run. If scheduling is done efficiently, it also increases the overall performance of the operating system.

Equation 7.1, p.98, shows that the latency of the scheduler $t_{scheduling}$ is part of the PDLT. So an improvement of the scheduler will also decrease the PDLT of every task.

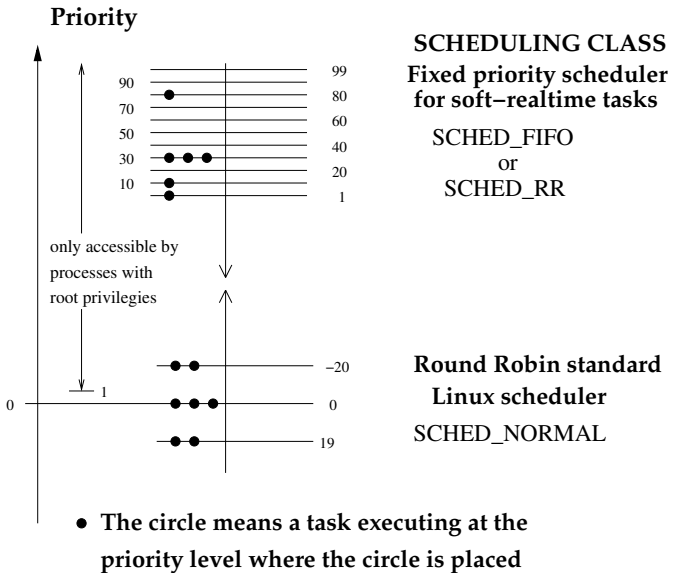



Figure 6.1: *Classes of the Linux scheduler*

To analyze whether Linux can fulfill this requirement a Linux single processor PC is set up with only one soft-realtime process, using the scheduling policy SCHED_FIFO, and many load processes, scheduled using the the standard policy SCHED_NORMAL.

The load processes run in an infinite loop performing a mathematical operation. This way all load processes are always able to run and they are always part of the runqueue, so that they can be chosen by the scheduler to be run next on the processor.

The soft-realtime process puts itself asleep for one second by calling `sleep(1);` The operating system reawakens the soft-realtime process after its sleeping period and puts it back into the runqueue, which is the waiting queue of all processes currently ready to run.

The scheduler then has to look through the runqueue to find out which process has the highest priority. Once found this process is brought to execution by the scheduler.

To measure the execution time of the scheduler two time stamps are taken, one at the start and one at the end of the scheduler function `'schedule()'` in the kernel code file `'sched.c'`. The time stamps are taken from the 64 bit Time Stamp Counter (TSC) register of the processor, see sec. 3.7.1, p.44.

The function `schedule()` then runs to find out the process of highest priority, which is here the SCHED_FIFO process. Just before the `schedule()` function ends and gives the processor to the SCHED_FIFO process, a second time-stamp is taken from the TSC.

The difference in between both is the execution time of the scheduler which has been printed with a `printk()` into the kernel ring buffer and from there later on by the Linux syslog kernel daemon into the file `/var/log/messages` to be analyzed.

This measurement is repeated 200 times with a definite number of load processes runnable, as the SCHED_FIFO process sleeps for 1 second for 200 times. Every such measurement is a point in figure 6.2. Varying the number of load processes fig. 6.2 has been drawn.

### 6.1.3 The Complexity $O(N)$ of the Linux 2.4 scheduler

The measurements presented in fig. 6.2 show that the scheduler execution time $t_{scheduling}$ of the Linux kernel 2.2 and 2.4 has a linear complexity $O(N)$ with $N$ being the number of processes ready to run on the processor, even for a soft-realtime SCHED_FIFO process [52].

This linear dependence of the scheduling time on the number of processes ready to run is due to a loop through the runqueue, that the Linux 2.4 scheduler performs to find out the process with the highest priority to run next:

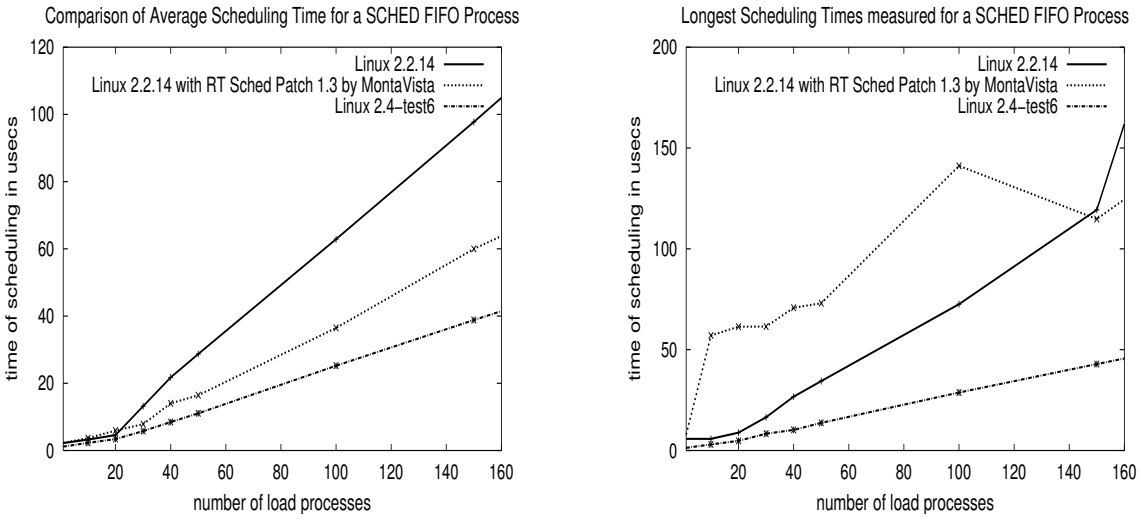



Figure 6.2: *Comparison of average scheduling times on the left and longest scheduling time on the right, measured for a SCHED_FIFO process varying the number of load processes ready to run. Shown are measurements of Linux kernels 2.2.14 and 2.4.0-test6 as well as measurements of Linux kernel 2.2.14 with the 'RT-Sched 1.3' scheduler patch of MontaVista [67].*

```
list_for_each(tmp, &runqueue_head) {
      p = list_entry(tmp, struct task_struct,
                                    run_list);
      if (can_schedule(p, this_cpu)) {
         int weight = goodness(p, this_cpu,
                          prev->active_mm);
         if (weight > c)
            c = weight, next = p;
      }
}
```

`list_for_each` is a preprocessor macro, defined in /include/linux/list.h, that expands to a 'for'-loop over all processes and threads that are currently ready to run.

```
#define list_for_each(pos, head) \
      for (pos = (head)->next; pos != (head);
                               pos = pos->next)
```

In the `goodness()` function,that has been used to calculate priorities up to Linux 2.4 the priority of the process `p` is calculated, SCHED_FIFO and SCHED_RR processes get a weight of 1000 added to their realtime priority in order to be preferred to all SCHED_OTHER processes.

So $max(t_{scheduling})$ in eq. 7.3 in Linux 2.2 and 2.4 is the time the scheduler needs to find the process of highest priority out of the runqueue of all processes ready to run. So the maximum of this term depends on the load of the system in Linux 2.2 and 2.4

The measurements of fig. 6.2 show that the Linux 2.4 scheduler works more rapidly than the scheduler of Linux 2.2, especially for more than 20 SCHED_OTHER load processes ready to run, which represents already a considerable load.

Comparing the source code of both kernel versions, we noted that in the kernel 2.4 the queuing mechanism has been reimplemented for all queues, also for those in the scheduler. The calculation of the dynamic priorities of SCHED_FIFO processes has been altered, the bottom halves have been endorsed by the concept of Soft IRQ´s in Linux 2.4.

The left part of the figure shows the average time the scheduler needs to find out the process of highest priority, while the right part shows the maximum scheduler execution time measured. The PC used for these measurements was an AMD K6 with 400 MHz, as described in appendix A.

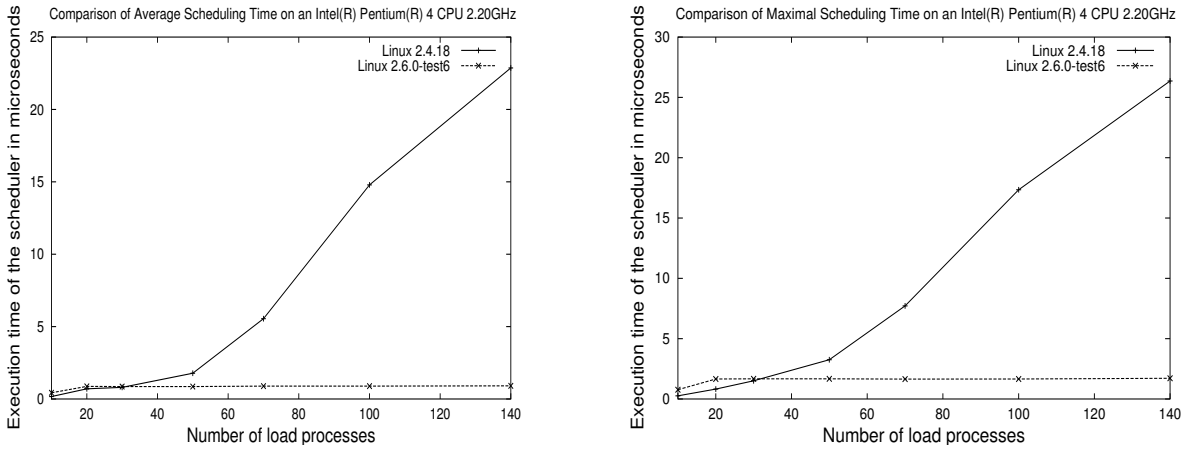



Figure 6.3: *Left image: Comparison of the* **average** *measured execution times of the scheduler of the Linux kernel 2.4.18 and 2.6.0-test6 versus the number of SCHED_NORMAL load processes ready to run, the right part shows the* **maximum** *measured execution times.*

Furthermore, MontaVista released a scheduler Linux kernel patch, later maintained as a project on sourceforge. This kernel patch maintains different smaller queues sorted by priority and a bitmap which shows which queues are empty and which are not. All these changes aim at minimizing the time the scheduler needs to find the process of highest priority ready to run.

Measuring the MontaVista Scheduler Patch for Linux 2.2, the left part of fig. 6.2 shows that the patch shortens the execution time of the scheduler for the average case of Linux 2.2, but there were some rare cases in the measurements, when the scheduling time was longer than in standard Linux 2.2, see the right part of fig. 6.2.

So in standard Linux the scheduling time even for soft realtime processes has been found to depend on the number of time slice processes ready to run, but Linux 2.4 could lower this effect compared to Linux 2.2 kernels, so that the effect should be important only for computers charged with many load processes, like server systems.

### 6.1.4 The Complexity $O(1)$ of the Linux 2.6 scheduler

The new scheduler of Linux 2.6 has been designed by Ingo Molnar [66], to be of constant complexity or order, i.e. O(1). This means the execution time of the scheduler is constant and does not depend on the number of other processes or threads that are currently ready to run any more.

The measurements presented in fig.6.3 show that this goal has been reached with the new scheduler of Linux 2.6.

To reach that goal the internal structure of the scheduler has been redesigned for Linux 2.6. Now the runqueue consists of 2 queues, one, called the 'active' one, with the processes that are still waiting to get the processor in order to execute their timeslice and another one, called 'expired' with processes that already consumed their timeslice. Both queues are sorted by the process priority and are referenced to by pointers. Processes that consumed their timeslice are put into the sorted 'expired' queue. If the 'active' queue is empty, just the pointers to the queues are exchanged, so that the former 'expired' queue becomes now the 'active' queue and vice versa. Since the duration of the new timeslice is calculated when a process enters the 'expired' queue, this is an O(1) design, that is also supported by using bit fields to determine the process of highest priority.

Apart from that, also in Linux 2.6 the POSIX soft-realtime scheduling policies SCHED_FIFO and SCHED_RR exist like in the kernels before.

Another new item is that the Linux 2.6 scheduler privileges slightly so called interactive processes, these are processes that sleep for longer times, for instance because they are waiting for user input from the

mouse or keyboard. The scheduler tries to keep such processes or threads in the 'active' queue, so that these tasks can get the processor faster in order to be able to react to user requests fast and timely. Mostly interactive processes don't need long execution times until they fall asleep again.

To avoid a bottle-neck regarding symmetrical multiprocessing (SMP) systems, in Linux 2.6 there is not only one runqueue, but one per processor in a SMP system.

The measurements shown in fig.6.3 have been executed in runlevel 1, with 10 running kernel threads, which were normally not active. The number of load processes ready to run has been varied from 0 up to 140 processes. When the graphical interface KDE is running on Linux there are normally about 50 processes or threads started.

### 6.1.5 Summary regarding time-critical tasks scheduled by Linux

The measurements of Linux 2.2 and 2.4 in fig.6.2 and fig.6.3 show that the execution time of the scheduler increases with the number of processes or threads running on the whole system. This is not appropriate for soft-realtime tasks.

The execution time of the scheduler in fig.6.3 is less than in fig.6.2, because fig.6.3 has been measured on a Pentium 4 with a processor frequency of 2.2 GHz, while fig.6.2 has been measured on an AMD K6 with only 400 MHz, cp. appendix A.1 and A.2.

The measurements shown in fig.6.3 for Linux 2.6 show that the goal to create a scheduler with a constant execution time has been reached with the new scheduler design in Linux 2.6.

The measurements show that the maximum execution time of the scheduler in Linux 2.6 on a Pentium 4 with 2.2 GHz for a soft realtime task -scheduled using the scheduling policy SCHED_FIFO - is constantly less than 4 microseconds.

Therefore, the scheduler since Linux 2.6 is appropriate to schedule soft-realtime tasks. Scheduler development in the Linux kernel is a good example that kernel developers have optimized the standard Linux vanilla kernel in recent years to be able to execute soft-realtime tasks in an adequate and more timely manner.

## 6.2 Interrupts and Soft-Interrupts

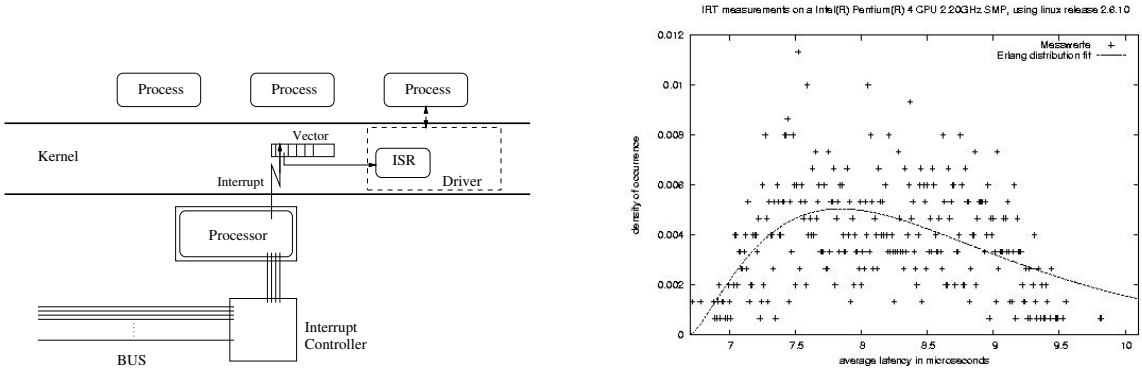



Figure 6.4: Left part: *Interrupt handling in the Linux kernel*, right part:*1500 Measurements of the Interrupt Response Time (IRT) of the parallel port of standard Linux 2.6.10*

This section deals with the interrupt handling of the Linux kernel and in particular with possible latencies caused by interrupts like the Interrupt Response Time (IRT) $t_{IRT}$ in equation (7.1).

The length of the IRT, the Interrupt Response Time, depends also on the underlying hardware. For Intel compatible x86 PCs it's possible on single processor systems to use either the older PIT chip, the Programmable Interval Timer, or the newer Local APIC chip, the Advanced Programmable Interrupt Controller, in order to generate the periodic timer interrupts for the Linux kernel, see section 3.7.1. The Local APIC chip produces a smaller IRT than the older Programmable Interval Timer chip.

In standard Linux the Interrupt Service Routines are preferred to all processes or threads as shown in fig.2.8, p.29.

The left part of fig. 6.4 shows a schematic overview how interrupts are processed in Intel compatible PC's.

The right part of fig.6.4 shows a measurement of the Interrupt Response Time (IRT) of 1500 interrupts generated by the parallel port. The distribution of the IRT measurements is similar to that of the standard Linux kernel 2.4.18, cp. fig.3.6. The measurement method has been described in section 3.6.4, the measurement has been executed using Linux 2.6.10 on a Pentium 4, as described in appendix A.

The IRT of standard Linux 2.6.10 differs from about 6.5 microseconds (usec) up to about 10 usecs in this measurement. In the measurement the Local APIC of the Pentium 4 processor is used as interrupt controller. The delays, that are much longer than 7.5 or 8 microseconds are probably caused by interrupt locks in the Linux kernel. The measurements are fitted by an Erlang distribution according to queuing theory.

### 6.2.1 Bottom-Halves and SoftIRQs

A common concept to reduce the duration of long interrupt handlers, i.e. the term $t_{interrupts}$ in eq.(7.1),p. 98, is to execute less time-critical code of the ISR later on. Therefore the interrupt handler is divided into a *top half*, called interrupt service routine ISR, that is executed directly after the interrupt reached the processor, and a *bottom half*, which is executed later on asynchronously.

In Linux 2.2 the Bottom Halves have been executed every time before the scheduler ran. That way, their execution time still always contributed to the PDLT of a soft-realtime process, cp. term $t_{bottom-halves}$ for Linux 2.2 in eq.(7.1), p.98, since a new time-critical tasks could only come to execution after top and bottom halves had been executed. So the advantage of this concept was only, that other top half ISRs could run before all bottom halves had been executed.

In Linux 2.4 the Bottom Halves have been replaced by soft Interrupts, so called SoftIRQs, that execute the same routines as the bottom halves did before, always in interrupt mode.

A further step to reduce the PDLT of a time-critical task is to make a kernel thread executing the bottom halves/SoftIRQs asynchronously later on.

If SoftIRQs are handled by a kernel thread, i.e. scheduled by the normal scheduler, the former bottom-halves don't contribute any more to the PDLT of a soft realtime process. That's an improvement for realtime tasks, cp. eq.(7.1) at p.98.

This concept has been implemented since Linux kernel version 2.4.10. Since this kernel version the SoftIRQs are handled by a kernel thread, called [ksoftirqd_CPU0]. There is one such kernel thread per CPU, so on a dual processor machine there is also a [ksoftirqd_CPU1].

The kernel threads [ksoftirqd_CPUn] have to do more than to process only the former bottom halves. They process all tasklets and there are four priority levels, new tasklets can be put into:

The former bottom halves are processed at the highest level. Beyond there are two levels that deal with networking, the higher one transmits network packets, the lower one receives network packets. Beyond there's a level, where less important tasklets are scheduled.

But since these kernel threads are scheduled only using the SCHED_OTHER - or since Linux 2.6 the SCHED_NORMAL - scheduling policy, they are not useful for tasklets, which face time-critical problems.

Since also the networking traffic with the network interface card is handled by the ksoftirqd thread, the network traffic even of soft-realtime tasks is delayed often for some milliseconds.

The following paragraph shows how this disadvantage can be changed.

#### Changing the priority of Soft-IRQ kernel threads

To make it possible to transmit and receive network traffic as fast as possible it is also necessary to schedule the SoftIRQ kernel thread using the policy SCHED_FIFO as a soft-realtime task.

In this thesis such a kernel patch has been developed and also the Timesys Linux 2.4.7 kernel executes the Soft-IRQ kernel threads using the scheduling policy SCHED_FIFO.

### 6.2.2 Interrupt locks and their duration

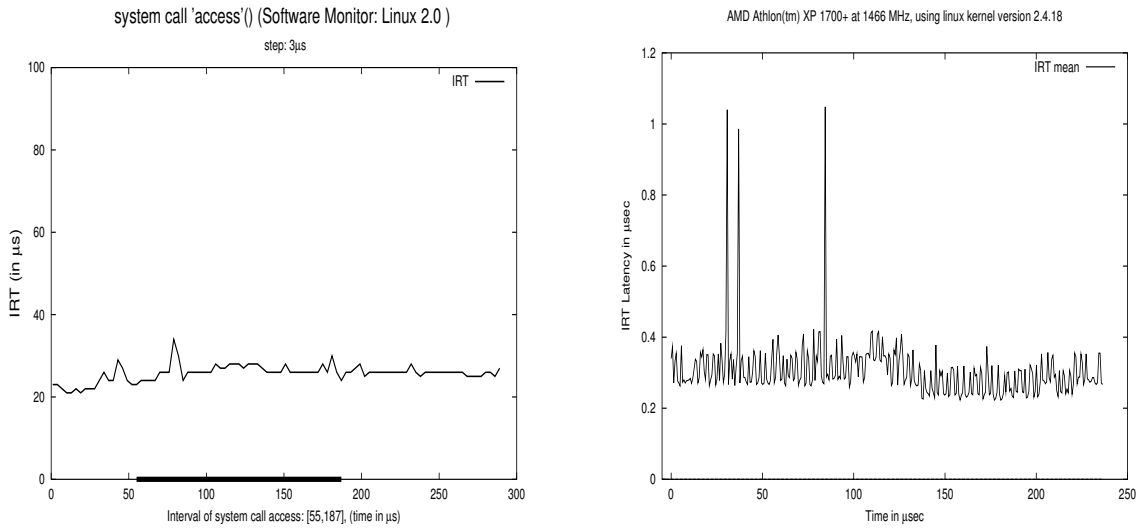



Figure 6.5: *Duration of the Interrupt Response Time (IRT), measured in Linux 2.0 on a Intel 486 PC running at 66 MHz on the left side and in Linux 2.4.18 on the right side, measured on an AMD Athlon XP 1700+. The peaks are probably caused by interrupt locks.*

Interrupt handler run in parallel to other kernel code and so data structures that are changed by system functions and by interrupt handlers have to be protected by interrupt locks.

This paragraph discusses the duration of such interrupt locks.

The execution time of critical sections and interrupt-locks, i.e. the duration of the latencies $t_{critical-sections}$ and $t_{interrupt-locks}$ in equation (7.4), p.107, doesn't depend only on the processor frequency.

The latency $t_{interrupt-locks}$ occurs accidently, when an interrupt has been triggered by an external device, and the status register of the interrupt controller indicates the interrupt, but the CPU can't process it, because interrupts are currently globally disabled at the local processor. Many standard operating systems like Linux and Windows disable interrupts globally at the processor to ensure the integrity of internal data structures.

Furthermore, regarding the duration of interrupt locks can be said the same as can be said regarding the execution time of critical sections in section 7.5.1, p.110:

Their execution time of a code while interrupts are blocked, can depend on parameters that have been given by the calling routine, if they contain loops, controlled by variables. The execution time can also depend on hardware parameters or on the value of internal variables or states of the operating system, if they are touched by the code executed with interrupts blocked.

Therefore, interrupt locks are one important reason, why in standard Linux guarantees for hard realtime tasks can't be given.

Some examples of interrupt locks in the Linux 2.4 kernel code:

- In the scheduler of the Linux kernel up to Linux 2.4 there is a `spin_lock_irq(..)` locking the runqueue and the interrupts while a loop is executed over all processes, that are ready to run. The execution time of the loop depends linear on the number of processes N in the ready queue of the processor, see section 6.1, p.85.
- On Linux 2.2 interrupt lock latencies of about 38 microseconds and more have been measured using the `printk('message')` on a Pentium II with 266 MHz [90, 67], because the kernel function `printk(message)` does string parsing in loops. So the endurance of this lock depends on the message length.
- Many interrupt locks and critical sections, i.e. preemption locks, can be found in the drivers section of the Linux kernel, especially when accessing peripheral hardware devices. This may lead to long kernel latencies, depending on hardware parameters. For instance such an interrupt lock including communication with a hardware device can be seen in the 'sidewinder joystick' driver of the Linux 2.4 kernel.

Fig. 6.5 shows the duration of interrupt locks on different personal computers and different kernels. The measurement have been made with the software monitor, presented in section 3.8.2. The peaks in both figures are due to interrupt locks in the Linux kernel. The left figure has been measured on an Intel 486 personal computer, that uses an older slower interrupt controller, the 82C59A-2, while the right figure has been measured on an AMD Athlon PC, 1500 MHz, using the modern APIC chips to manage interrupts.

Nevertheless, during measurements in this thesis interrupt locks of the endurance of up to 207 microseconds have been measured in rare cases [90]. Also [47] measured that in rare cases, Linux 2.4.23 on an AMD Athlon XP 2400+ showed a maximum interrupt response time of 501 microseconds.

### 6.2.3 'Hard realtime interrupts' - Mirroring the interrupt controller into memory in standard Linux

As stated in section 4.1, p.53, RTLinux mirrors the registers of the interrupt controller into memory. Thus when the Linux kernel sets an interrupt lock, this means only a bit in memory is set, since the Linux kernel doesn't work on the interrupt controller any more. That way interrupts always reach the processor, as they are not blocked at the hardware interrupt controller.

In 2004 some Linux kernel patches and kernel derivatives arose, that are doing the same also in standard Linux [14, 47], namely the RTIRQ patch.

Interrupts can be processed by a so called 'hard realtime interrupt handlers'. If any so called hard realtime ISR is assigned to that interrupt, it can be carried out.

On the other hand, if there is no hard realtime interrupt handler assigned, the hard realtime ISR stub will start the normal Linux interrupt handler, if Linux hasn't currently interrupts blocked. If it currently has, the interrupt is queued, so that it can be processed by Linux later on. That way, if more than one interrupt occur, while Linux blocks interrupts, interrupts are queued to be treated later on and thus don't get lost as they would do in standard Linux.

That way, so called hard realtime interrupts are possible even on Linux, and their Interrupt Response Time should be deterministic and in the range of some microseconds on current Intel compatible PCs, as they don't suffer from Linux interrupt locks. [47] reports the maximum IRT out of 60000 samples using Linux 2.4.23 with the RTIRQ patch to be 5.2 microseconds measured on an AMD Athlon XP 2400+. This is two order of magnitudes better than the worst case IRT of standard Linux 2.4.23 on the same PC, which was 501 microseconds.

In such a hard realtime ISR all Linux functions can be invoked and carried out that don't contain critical sections, that block interrupts like `spin_lock_irqsave(..)` in standard Linux does. The reason is that the standard Linux interrupt locks don't block hard realtime interrupts.

Therefore, data shared in between Linux functions and a hard realtime ISR or critical sections that block interrupts in normal Linux, must be protected by a new synchronization primitive, a new `spin_lock_irqsave(hard_RT_ISR)` that permits mutual exclusion in between Linux and the hard realtime ISR, that is using that function, too. Doing this, also a hard realtime ISR will suffer from latencies due to necessary synchronization. Furthermore, since the hard realtime ISR runs at the highest priority in the system and the Linux process using the new lock will run on a smaller priority level, also priority inversion situations can arise, which would need a priority inheritance protocol to be solved. That way it can be easily seen, that hard realtime IRQs doesn't make the whole Linux functionality available at hard realtime conditions. Priority inheritance protocols (PIP) are discussed in section 4.6.5.

Nevertheless, hard realtime ISRs could do e.g. operations on peripheral devices, that aren't managed by the Linux kernel. So a further development of the RTIRQ patch [47] could e.g. be useful for hard realtime applications as data acquisition and control loops.

But it has to be stated, that RTLinux, which uses a dual kernel approach, is a solution that deserves much more confidence regarding hard-realtime constraints, because in the hard-realtime context of RTLinux only such functions are executed, that do not share any data or code with standard Linux and thus can't interfere with it. Thus the concept of the dual kernel approach is clearer, less error prone and therefore usable by less educated and less experienced developers.

### 6.2.4 The IRQ-Thread patch - Transforming interrupt handlers into kernel threads

As stated in section 4.6.7, in this thesis a kernel patch has been developed, but not published, since it was based on parts of the code of the Timesys GPL Linux kernel 2.4.7 [96], that makes interrupt service routines (ISR) handled by kernel threads. Similar patches have been developed at Fujitsu-Siemens or by Ingo Molnar at Red Hat Inc., who made it part of the standard vanilla Linux kernel 2.6.

So called 'threaded IRQs' or 'threaded interrupt service routines' allow to convert even preemption locks into mutexes that are accessed by interrupt service routines, cp. section 4.6.7.

The so called 'IRQ-Thread patch', i.e. the Interrupt kernel Thread patch, makes each interrupt handled by its own kernel thread, so that it can be prioritized individually. In some implementations, only the timer interrupt remains to be handled as a classical non-threaded interrupt by the original routines of the standard Linux kernel. That way the timer interrupts remains always the routine of the highest priority in the whole system, which is useful, because precise timing of the system is a basic requirement for all actions to be done under realtime constraints, cp. section 2.2.8.

When the kernel boots, for every assigned interrupt besides the timer interrupt an own kernel thread is started. The normal interrupt routine is assigned to it as thread function, its priority is raised to a high soft-realtime SCHED_FIFO priority, for instance to the highest level 99, and then it is put asleep.

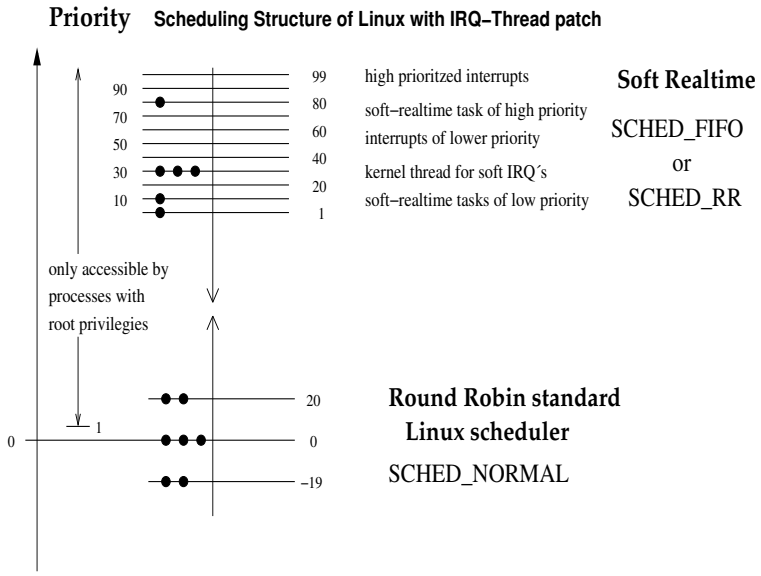



Figure 6.6: *Structure of a Linux kernel with IRQ-Thread patch, which can - in case of full preemptability - serve time-critical tasks in a better way, because interrupts and SoftIRQs can be prioritized individually*

For all interrupt kernel threads the POSIX fixed priority scheduler SCHED_FIFO implemented in the standard Linux kernel, cp. section 6.1 on p.85, is used to give a higher priority to the interrupt handlers and thus prefer them to any of the normal tasks that are scheduled by the standard Unix Round Robin scheduler SCHED_NORMAL.

When an interrupt comes in, the general interrupt service routine stub `do_IRQ()` wakes up the appropriate kernel thread, that executes the interrupt service routine (ISR). After being brought to execution on the processor by the scheduler, the interrupt service routine is executed. After that, the kernel thread is put asleep again. Of course, interrupts not used can also be assigned later dynamically using `request_irq(...)` which will start a new kernel thread.

### Measurements of the IRT of a Linux kernel with IRQ-Thread patch

Section 4.6 deals with replacing all preemption locks in the kernel of an operating system with mutexes and thus creating a 'fully preemptible kernel'.

There have been different implementations of this concept. They are discussed in detail in section 7.5.4.

One implementation has been done by Timesys Inc. for the Linux 2.4.7 kernel - partly protected by a commercial license and providing partly only binary, crypted code.

Another implementation has been done by MontaVista Software Inc. MontaVista implemented it their own way using a simple priority inheritance protocol (SPIP/EPV2) developed in this thesis, cp. section 4.6.5 and presented it using the GPL2 license in the 'MontaVista Linux Open Source Real Time Project' for early Linux 2.6 kernels.

A third implementation has been done by Ingo Molnar, kernel developer at Red Hat Inc., published under GPL2 and integrated into early 2.6 kernels as a configurable option at compile time.

These implementations are presented in detail in section 7.5.4. All of them do need an implementation of the IRQ-Thread patch, for reasons given in section 4.6.7. The MontaVista Linux Open Source Real Time Project is built upon the IRQ-Thread patch, implemented by Ingo Molnar for the Linux 2.6 kernel, as MontaVista Inc. did want to use existing code of the kernel developer community.

Hereafter, the effects of the IRQ-Thread patch have been measured regarding the IRT, the Interrupt Response Time. The measurements have to be compared to the IRT of the standard vanilla Linux kernel of the major version 2.4 or 2.6.

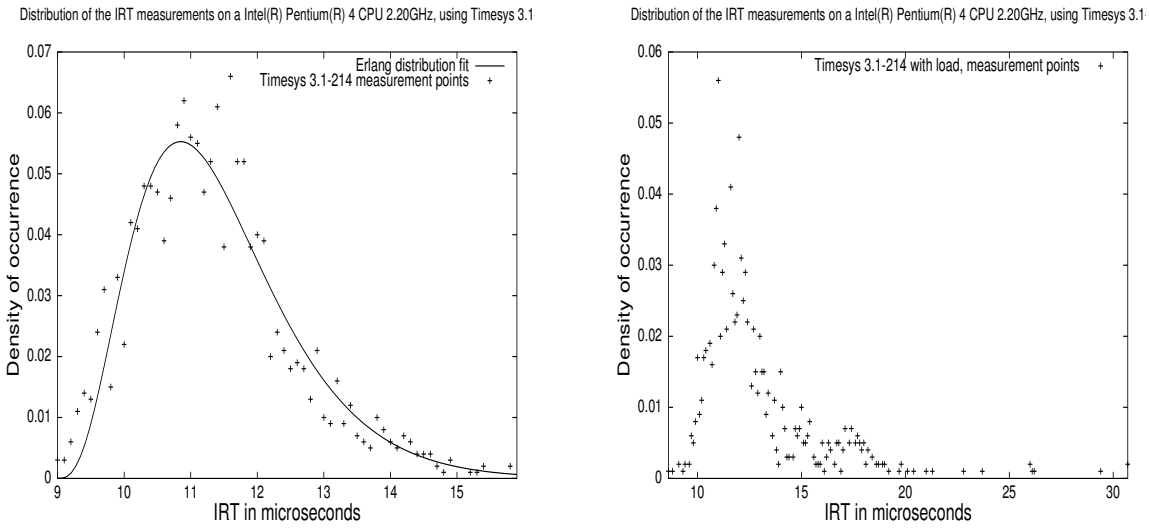



Figure 6.7: *Left part: Distribution of 1000 measured values of the Interrupt Response Time (IRT) of the parallel port of a* **Timesys Linux GPL kernel 2.4.7***, Right part: Same measurement as on the left side, but* **with the load** *of kernel compiling, find, tty and drag & drop operations.*

All measurements have been carried out by generating interrupts on the parallel port of a Pentium 4, see appendix A.2, as described in section 3.6.4, p.40.

Fig.6.7 shows the IRT measurements of a Timesys GPL kernel 2.4.7, i.e. Timesys version 3.1. The comparison of the measurements of the left part of fig.6.7 and fig.3.6, p.42, shows that the IRQ-Thread Patch with its kernel threads in the Timesys kernel leads to a higher average and maximum Interrupt Response Time of the Timesys GPL kernel 2.4.7, compared to standard vanilla Linux 2.4.18 without IRQ-Thread patch in fig.3.6, p.42.

This is due to scheduling, which is necessary for kernel threads, that execute the ISR in the Timesys kernel, but not in standard Linux.

The right part of fig.6.7 shows that under load in rare cases the IRT of the Timesys GPL kernel 2.4.7 on our Pentium 4 reaches up to 25 or 30 microseconds, which is much higher than the standard Linux kernel, cp. fig.3.6, p.42.

This might be due to a special mutex often used in the Timesys kernel causing the KLT, when an interrupt handler accidentally tries to access an already locked mutex, cp. section 4.6.1 on p.66.

Apart from that, the Timesys 2.4.7 kernel, also named Timesys 3.1, does still contain some preemption locks, i.e. spinlocks in the SMP case, for instance to protect its own mutex implementation.

Fig.6.8 shows the IRT of Linux 2.6.9 developer kernels using the IRQ-Thread patch, implemented by Ingo Molnar. The right part of fig.6.8 is one of the first fully preemptible Linux kernel of the MontaVista Linux Open Source Real Time Project with most of its preemption locks converted to mutexes, as presented in section 7.5.4.

The IRT of Linux with IRQ-Thread patch, see fig.6.8, is again higher compared to that of standard Linux 2.6.10, shown in the right part of fig.6.4, p.90. The average IRT of Linux 2.6.10, see fig.6.4, is about 8 microseconds, all measurements are in between 6 and 13 microseconds. The IRT of Linux with IRQ-Thread patch, see fig.6.8, is in between 10 and 27 microseconds with an average of 13.7 microseconds.

It is important to say, that the IRQ-Thread patch applied to the Linux kernel without converting all preemption locks into mutexes lead to many higher interrupt response times, IRTs, because the IRQ kernel threads, that are subject to the scheduler, are - in kernels with IRQ-Thread patch - also blocked by preemption locks. Therefore, it is important, that the few remaining critical sections protected by preemption locks, and not converted into mutexes, in a fully preemptible kernel are kept very short.

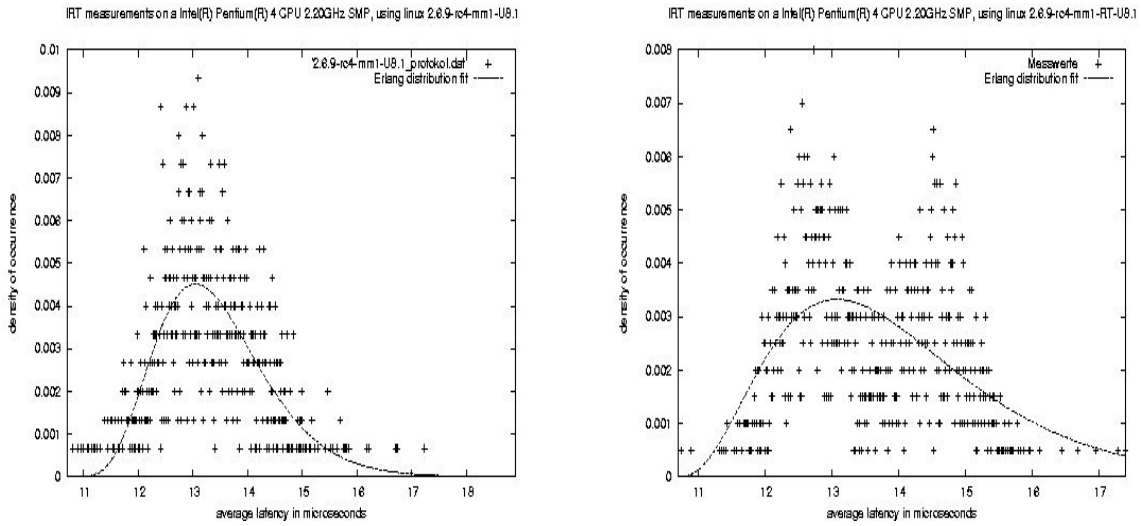



Figure 6.8: *Left part: Distribution of 2000 measured values of the Interrupt Response Time (IRT) of the parallel port of a* **Linux development kernel 2.6.9-rc4-mm1-U8.1**, *Right part: Same measurement as on the left side, but* **with RT patch of the MontaVista Linux Open Source Real Time Project as fully Preemptible Kernel**. *Both development kernels use the IRQ-Thread patch.*

In one of the first fully preemptible kernels of the MontaVista Linux Open Source Real Time Project as shown in the right part of fig.6.8, there are still some preemption locks, that couldn't be converted into mutexes yet.

Probably, the IRT of future fully preemptible Linux kernels - will be less than the IRT of the right part of fig.6.7. But because of the necessary scheduling the IRT of fully preemptible kernels will always be some microseconds higher than in a standard Linux kernel. This is the price to pay for the smaller PDLT of a fully preemptible Linux kernel.

# Chapter 7

# Preemptability - Measuring to judge implemented concepts

## 7.1 Non-preemptible regions in the Linux kernel

This section deals with the preemptability of the Linux kernel and how it can be improved. The general concepts have already been outlined in section 4.3.4, p.61, up to the end of chapter 4.

Equation (7.1), p.98, shows the different terms that influence the PDLT of a soft-realtime process. The standard Linux kernel is not preemptible in general. Thus the Linux kernel can cause long latencies called *preemption-delay*. Fig.4.2, p.61, shows how such long latencies caused by the Linux kernel can occur and will be explained in the following.

Different cases have to be distinguished, also in the Linux kernel:

1. If the processor is currently executing code in user mode, when the interrupt occurs (see line A1 in Fig.4.2, p.61), then the scheduler can be invoked just after the interrupt has been processed by the CPU. In this case there is no kernel latency, $t_{preemption-delay}$ is zero in equation (7.1). The time to preempt a process running in user-space takes only a few microseconds on current hardware [52].

2. If an interrupt occurs while the kernel executes a system call for any process, then the process normally is interrupted in order to process the Interrupt Service Routine (ISR), if interrupts aren't currently blocked. If the ISR wakes up a soft-realtime process, the scheduler isn't invoked instantly, because at first the kernel has to finish the system call it was currently processing before the interrupt occurred. This situation is shown in line A2 of fig.4.2, the appropriate kernel latency is called $t_{preemption-delay}$ in equation (7.1). Since Linux kernel code is not preemptible in general, also kernel threads like the Linux kernel daemons cause kernel latencies.

   As several measurements [101, 99, 70] have shown, the longest kernel latencies produced by kernel code of Linux 2.2 on personal computers used in year 2000 were on the order of tens up to hundreds of milliseconds [59]. So $t_{preemption-delay}$ is the most important term in equation (7.1).

The following sections show in which way and with which success the concepts to improve preemptability, discussed in section 4.3.4, p.61, and in the following sections of chapter 4 have been implemented in the Linux kernel.

## 7.2 Analytical formula of the PDLT for a soft-realtime task in Linux

To discuss the latencies that occur in the Linux kernel hereafter a formula is presented that describes the PDLT of the process of highest priority, that just became ready to run, on a computer system running Linux.

Since that process shall be time-critical it is scheduled using the fixed priority scheduler referred to as SCHED_FIFO, see section 6.1.

The PDLT of this soft-realtime process consists of several independent latencies, see also fig.3.2, p.35:

$$PDLT_{soft-realtime} = t_{IRT} + t_{interrupts} + t_{preemption-delay} + t_{scheduling} + \{t_{bottom-halves}\}_{Linux2.2} \tag{7.1}$$

The time in between a peripheral device sends an interrupt and the first instruction of the Interrupt Service Routine (ISR) is called Interrupt Response Time $t_{IRT}$. As fig. 3.2 shows, $t_{IRT}$ can be divided into 2 parts, hardware caused latencies and a latency caused by interrupt-locks in the Linux kernel, see section 6.2.2:

$$t_{IRT} = t_{IRT-Hardware} + t_{interrupt-locks} \tag{7.2}$$

In the Interrupt Service Routine (ISR) the soft-realtime process of the highest priority in the system is awakened. The duration of the ISR contributes to the term $t_{interrupts}$ in equation (7.1). Generally, $t_{interrupts}$ contains the duration of all ISRs that are executed in between the interrupt and the first instruction of the soft-realtime process that shall be awakened. The interrupt subsystem is discussed in detail in section 6.2.

The term $t_{preemption-delay}$ refers to the preemptability of the kernel. It is discussed in this chapter.

The term $t_{scheduling}$ means the time the scheduler needs to find out the process of highest priority and the normally much shorter time of the context switch to this process. The scheduler is discussed in section 6.1.

A soft-realtime process should protect itself from being swapped out by locking its address space into the RAM using `mlockall()`.

The latency $t_{bottom-halves}$ is only relevant for the Linux 2.2 kernels, not for the Linux 2.4 kernels later than 2.4.10. In these newer kernels the bottom-halves are replaced by SoftIRQs, i.e. soft interrupts, that are processed by a kernel daemon called 'ksoftirqd_CPUID'. So the SoftIRQs/bottom-halves don't contribute to the PDLT of a soft-realtime process any more. In Linux 2.2 the 32 bottom-halves are executed as first part of the scheduler, so every bottom half could be executed only once during the $PDLT_{Linux2.2}$. Bottom halves are only executed, if they have been activated by an ISR for instance. In the worst case, if a long system-call latency occurs, we must assume that all bottom halves have been activated and that they all execute under those circumstances, that they use their maximum time. As there are at maximum 32 bottom halves we get the formula:

$$\{max(t_{bottom-halves})\}_{Linux2.2} = \sum_{a=1}^{32}(t_a)$$

Here $t_a$ means the execution time of one bottom half.

Since the terms or latencies of equation (7.1) can be considered as largely independent from each other, the maximum value of every term in equation (7.1) leads to the maximum PDLT:

$$max(PDLT_{soft-realtime}) = max(t_{IRT}) + max(t_{interrupts}) + max(t_{preemption-delay}) + \\ + max(t_{scheduling}) + \{max(t_{bottom-halves})\}_{Linux2.2} \tag{7.3}$$

## 7.3 Introducing Preemption Points into the kernel

As already stated the Linux kernels 2.0, 2.2, 2.4 and even 2.6 without a special configuration option are non-preemptible kernels.

One of the first attempts to improve the responsiveness of the standard Linux kernel has been made by Ingo Molnar, who offered a so called 'Low Latency kernel patch' for the Linux 2.2 kernel in 2000 [65]. This patch has been used by people who aimed at listening to music on Linux and therefore wanted to reduce the latencies on Linux beyond 40 milliseconds [52]. Later on Andrew Morton proceeded to develop the 'Low Latency Patches' [69].

The main method of the ´Low Latency Patches´ is to introduce Preemption Points at certain positions in kernel code in order to make certain system calls or kernel threads with a long execution time at least preemptible at certain points, see line B of fig.4.2.

In the Linux kernel the code of a Preemption Point looks like this [69]:

```
#define conditional_schedule()
  do {
       if (current->need_resched) {
        current->state = TASK_RUNNING;
        schedule();
       }
     } while (0)
```

These Preemption Points only take effect, when the flag `current→need_resched` has been set, this could have been done by an interrupt service routine (ISR) that awakened the process of high priority. Setting this flag to 1, forces the Linux kernel to reschedule as soon as possible. Since the scheduler isn't called, if the flag is not set, these points are called *Conditional Preemption Points*.

### Where to place such Preemption Points

| Functional sections of Linux as operating system | Number of Preemption Points (PP) in Low Latency Patches | |
|---|---|---|
| | Ingo Molnar 2.4.0-test6 Patch E2 | Andrew Morton Linux 2.4.2 |
| process administration | 10 | 3 |
| memory management | 48 | 31 |
| file system<br>N of them in ReiserFS | 29 | 30<br>12 |
| Input/Output<br>N of them in keyboard driver | 9<br>3 | 2 |
| Total number of PP | 96 | 66 |
| conditional exit points in loops/functions | 5 | 5 |

Table 7.1: *New Preemption Points introduced into the Linux kernels 2.2 and 2.4 in I. Molnar's and A. Morton's Low Latency Patches*

Since especially loops, for instance to copy and erase much data or to pass through or modify long lists, cause long latencies, both 'Low Latency patches' place many Preemption Points in loops.

I. Molnar's patch divides copying or deleting of big data blocks into parts as big as a memory page, followed by a Preemption Point. Big loops, e.g. to shrink caches, are reduced by the patch to smaller loops, so that the processor works shorter in an non-preemptible zone.

Therefore, both patches introduce five additional exit conditions to terminate long loops or functions, which make the code work more efficient.

Both patches focus on long latencies in the functional subsystems ´memory page management´ and ´file system´ of the Linux kernel. Andrew Morton's patch is focused on the network area and the Reiser file system, too.

Ingo Molnar [65] identified six sources of long latencies in Linux 2.2: Calls to the disk buffer cache, memory page management, calls to the /proc file system, VGA and console management, fork and exit of large processes and the keyboard driver. Furthermore, Ingo Molnar's patch made busy waiting interruptible and rewrote the keyboard driver to make it reentrant.

Table 7.1 shows in which functional sections the two patches introduce new Preemption Points. From the table it can be seen that Ingo Molnar introduces more Conditional Preemption Points into the Linux kernel than Andrew Morton.

Additionally features like the debug mode, the robustness test and system control make A.Morton's patch more user-friendly. Also his source code is better structured and commented, which increases its maintability. A.Morton's patch also offered information to developers at runtime providing a debug mode and using the interface `/proc/sys`. Apart from that, the patch could be activated or deactivated in the `.config` file, when compiling a Linux kernel. This patch is available also for recent Linux 2.4 kernel versions [70].

Ingo Molnar's patch has proven to reduce several long latencies of the Linux kernel successfully [101, 70] to the order of 5 to 10 milliseconds on current hardware in 2000. But I. Molnar's original patch has not been ported since the Linux kernel version 2.4.0-test7.

The concept of introducing Preemption Points, see section 4.4, does not worsen the performance of the Linux kernel much as some test measurements using the Rhealstone Benchmark [52] have shown.

In Linux 2.4 and Linux 2.6 some of these Preemption Points have been incorporated into the standard Linux kernel and many other kernel patches [70, 59]. The standard Linux kernel 2.4.3 contained already 34 Preemption Points for the x86 architecture. Because of this and because of the Preemptible kernel, that is discussed in section 7.4, and the incorporation of some of these preemption points the 'Low Latency Patches aren't maintained any more since Linux 2.6 has been released in 2003.

### Change of task state transitions

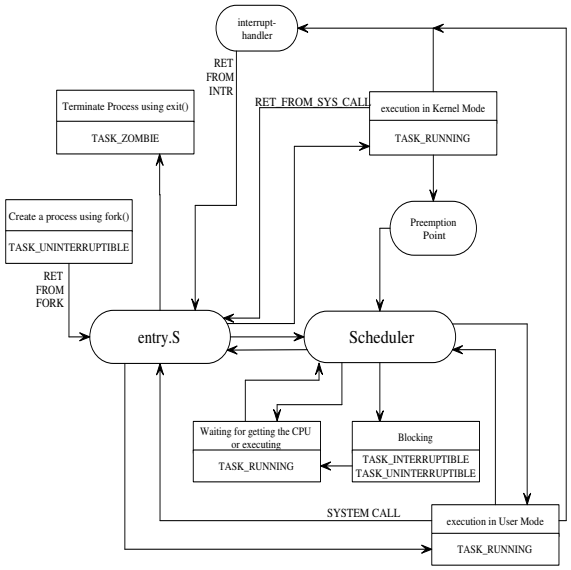



Figure 7.1: *The file 'entry.S' with its assembler routines used for task transitions in standard Linux*

Task state transitions take place before and after an interrupt service routine is executed and before and after a system call in kernel mode is invoked.

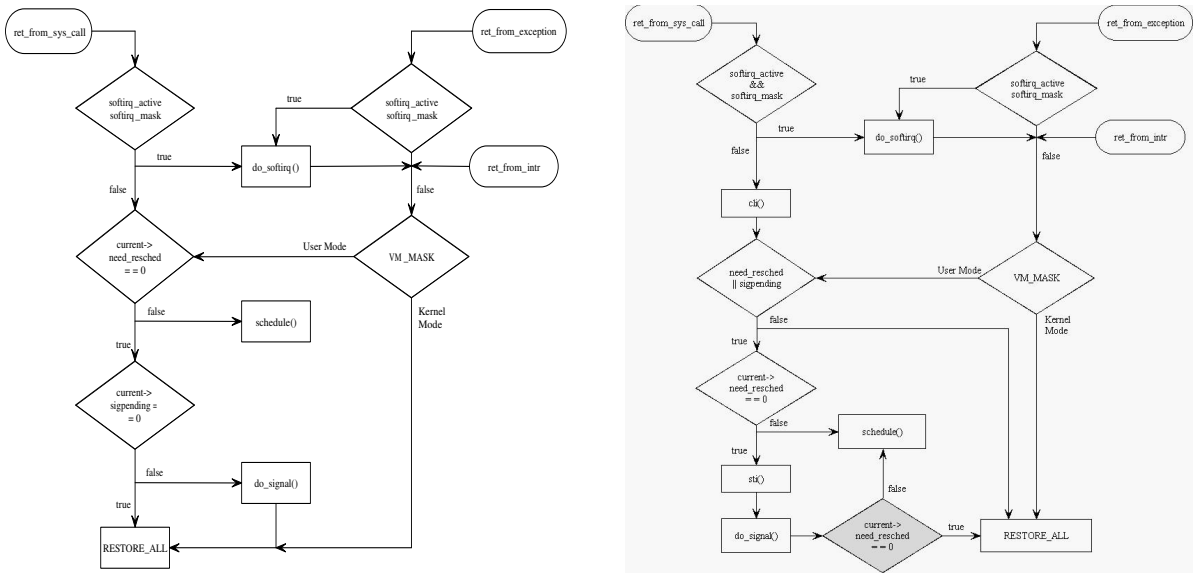



Figure 7.2: *Flowchart of the file 'entry.S', on the left side of the standard Linux kernel 2.4, on the right side of the LowLatency patch.*

The left part of fig.7.1 shows the central position of the file 'entry.S' in the Linux kernel. At many task state transition one of the assembler routines in the file 'entry.S is invoked.

The left part of fig.7.2 shows the flowchart of the file `'entry.S'` of Linux 2.4.2, the right image shows that Andrew Morton's Low Latency patch changes this often used code of the file `'entry.S'` slightly.

The Low Latency Patch tests for signals to process and a need of rescheduling on the same time, in order to make the common case work faster when there is no signal to process and no rescheduling necessary.

#### Pros and Cons of Introducing Preemption Points into the kernel

At first the technique of introducing preemption points into the code needs a theoretical idea, where latencies in operating systems do occur, a detailed analysis of the source code and many measurements at runtime. Analyzing the code is necessary on the one hand to find the right place for a Preemption Point, where the best efficiency is reached, on the other hand to avoid data inconsistencies to occur, that could lead to a crash of the system.

So setting Preemption Points needs a lot of experience and they are also costly to maintain since after every code change these checks have to be redone.

Another disadvantage is that a Preemption Point only decreases the latency of the routine it is placed in. Although it has been searched carefully, there may always be still long latencies in routines, that didn't show up often, aren't known or only occur in special configurations and circumstances. It's also possible that someone adds a new driver which adds a long latency to the system.

So from section 4.4.1 it's clear that Preemption Points reduce very efficiently special long latencies, but they do not provide a general approach like other concepts do, for instance making the kernel of an operating system preemptible in general.

### 7.3.1 Practical measurements using the Software Monitor

In this section measurements are presented in order to visualize the different concepts and to test how they work in a special case. This also serves as proof of concept for the different ideas. The measurements have been done with the software monitor that has been presented in section 3.8.2. These measurements have been carried out on an Intel x86 compatible personal computer with an AMD K6 processor running at 400 MHz, as described in appendix A.

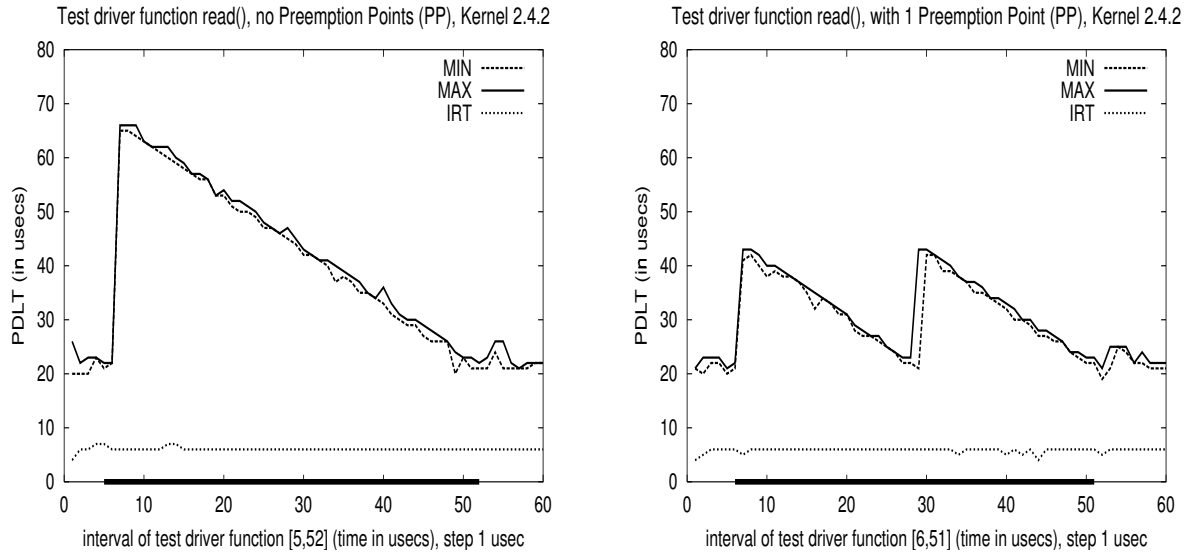



Figure 7.3: *PDLT of test driver function* `read()`*, for-1200-loop without Preemption Points, Linux 2.4.2. Shown are Minimum and Maximum of the PDLT, the Process Dispatch Latency Time, and the IRT, the Interrupt Response Time. The thick line on the X-axis indicates the time during the test driver function read() has been executed. The left part shows the PDLT latency of the system function without preemption point, the right part with 1 preemption point.*

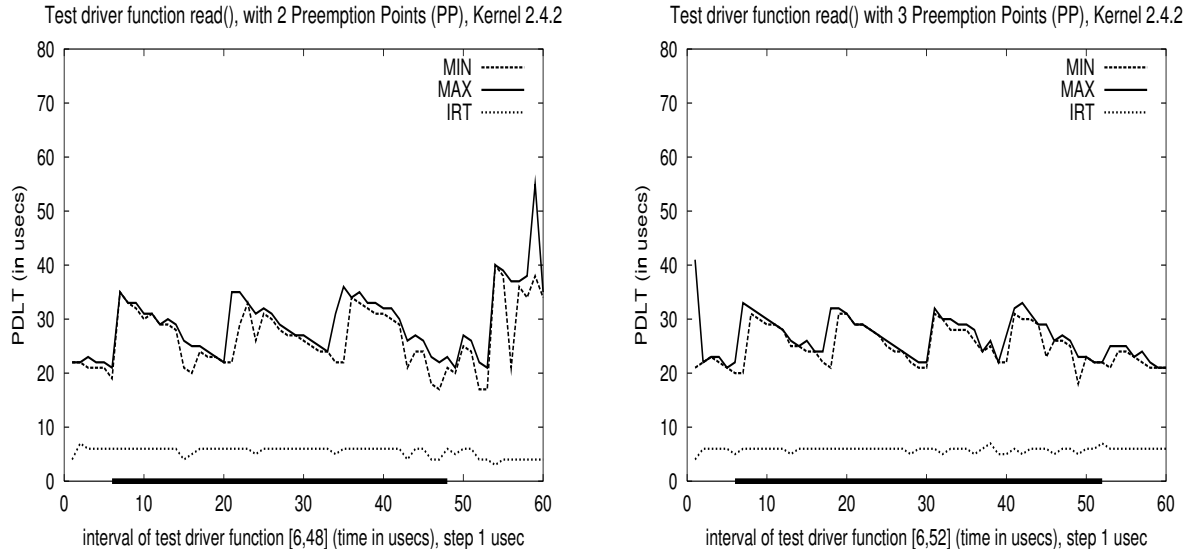



Figure 7.4: Left part: *PDLT latency of the same system function as in fig. 7.3, but with 2 Preemption Points.* Right part: *The same system function with 3 preemption points.*

### Preemption of a kernel driver method

In the measurement method presented in section 3.8.2 the low priority SCHED_FIFO process - the 'Consumer' - invokes a system call.

The measurement setup presented there has to be altered slightly:

A new kernel device driver in a kernel module has been introduced to serve to a new device `'/dev/measure'`. The 'Consumer' process invokes the `read()` function, a system call which is handled by the Virtual File System (VFS) to the `read()` function of the new kernel driver, executing the code in kernel mode, which is not preemptible in standard Linux.

The source code of the read-function of the test kernel driver, executes only a loop, it does not read any

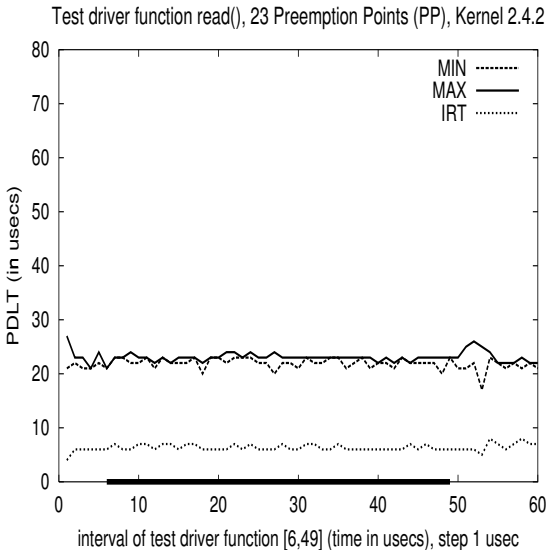



Figure 7.5: *PDLT latency of the same system function as in fig. 7.3, but with 23 Preemption Points.*

data:

```
static ssize_t read(struct file *file,
          char *user_buf, size_t count,
          loff_t *offset)
{
  int i;
  for(i=1; i<1200; i++)
  {
    if((i%1200)==0)  // modulo n function
    {
      if(current->need_resched)
          schedule();  // Preemption Point
    }
  }
  return(0);
}
```

This is the loop, the 'Consumer' process in Figure 3.8 executes in the kernel driver. One can control the amount of Preemption Points by altering the argument of the "modulo" function in the "if" clause. An "$i$%1200" means no Preemption Point whereas "$i$%600" stands for one Preemption Point in the middle of the "for" loop. The "modulo" function has to be checked in every loop.

Now using the software monitor shown in fig.3.8 on p.47 different experiments with different lines of code in the `read()` function of the kernel driver have been made:

1. At first the for-loop without any Preemption Points is measured. As the driver is part of the kernel and `read()` is a system call, it can´t be preempted by the high priority process - the 'Producer'. Thus the left part of Fig.7.3 shows a triangle of about 65 microseconds (usecs) with a minimum PDLT baseline of about 22 usecs. This baseline is defined by the preemption time including scheduling and the measurement overhead. The execution of the for-loop costs about $65 - 22 = 43$ microseconds.

2. In the next steps Preemption Points have been introduced into the for-loop.

   (a) First in the right part of fig.7.3 only 1 Preemption Point has been introduced at half-time of the for-loop. There one can see, that the one and only triangle of the left part of fig.7.3 is split into two smaller triangles in the right part, i.e. into two non-preemptible regions.

(b) Next 2 and 3 Preemption Points have been introduced into the `read()` function of the test kernel driver. $N$ preemption points divide the code of the system function into $N + 1$ parts. So fig.7.4 shows 3 PDLT-triangles in the left part and 4 in the right. Fig.7.3 up to 7.5 show the more preemption points are introduced into the code, the smaller the PDLT triangles become. That way they confirm eq.(4.1) on p.64.

(c) At last 23 Preemption Points have been introduced into the for-loop. Fig.7.5 shows the result: The triangles nearly disappear.

An Preemption Point is only executed, if the variable `current→need_resched` is set to 1 by the ISR. The code of the `read()` function in the kernel driver is divided into 24 parts by the 23 Preemption Points. When the interrupt occurs, the code is executed up to the next Preemption Point, where the scheduler is invoked. The scheduler then preempts the 'Consumer' process and gives the processor to the 'Producer'. All other 'Conditional Preemption Points' do not affect the code, except for testing `current→need_resched` at every Preemption Point.

## 7.4 The 'Preemptible Linux Kernel'

### 7.4.1 The Preemptible Kernel Patch or 'Preemption Patch'

#### Making the kernel preemptible in general - introduction of non-preemptive critical sections (NPCS) into the kernel

An important idea to reduce the term $max(t_{preemption-delay})$ in eq.(7.3), p.98, has been presented by Embedded System vendor MontaVista [67] and has been referred to as *'Preemptible Kernel'* or *'Preemption Patch'*. The technique used in this patch partly stems from the Unix operating system Real/IX, which has been optimized for realtime purposes in the eighties of the last century. In literature this method is often referred to as the introduction of **N**on-**P**reemptive **C**ritical **S**ections -**NPCS**-, into a kernel. cp. section 4.5.2, p.65.

After its presentation the Preemption Patch has been developed as the Open Source project 'kpreempt' by Robert Love [100, 55]. Since Linux 2.6 the patch has been integrated into the official Linux kernel and thus it can be activated when configuring the Linux 2.6 kernel by a switch before compiling the kernel. If it is activated the kernel identifier, that can be shown using the bash command `uname -r`, contains the string `preempt`.

The Preemption Patch has the intention to make Linux kernel code preemptible in general - as shown in line C of Fig.4.2, p.61 - in order to improve the responsiveness of the Linux kernel on single and also multiprocessor systems.

The Preemptible Kernel takes advantage of the critical sections protected by spinlocks that have been introduced into the Linux 2.2 kernel in order to make Linux running in Symmetrical Multi-Processor Mode (SMP) on multiprocessor systems.

#### Spinlocks and Symmetrical Multi-Processor Mode (SMP) in the standard Linux kernel

The standard Linux kernel 2.2 and 2.4 support up to 4 multiprocessors in a Symmetrical Multi-Processor (SMP) System.

In addition to interrupt handlers running in parallel to system calls, on a multiprocessor system several processors can execute kernel code, e.g. system functions, at the same time. To avoid data structures are changed inconsistently, i.e. to avoid race conditions, the standard Linux kernel provides critical sections protected by spinlocks in SMP mode.

Spinlocks assure that only one task can enter a critical section at a time on a multiprocessor system. This means they act like a mutex or a binary semaphore. But there is a difference compared to mutexes and semaphores:

| | return from interrupt or exception to **User-Mode** Preemption allowed ? | return from interrupt or exception to **Kernel-Mode** Preemption allowed ? | return from interrupt or exception to **Kernel-Mode within critical section** Preemption allowed ? |
|---|---|---|---|
| Standard Linux | yes | no | no |
| Preemptible Kernel | yes | yes | no |
| Fully Preemptible Kernel using mutexes instead of preemption locks | yes | yes | yes |

Table 7.2: *This table describes when preemption can take place. The Preemptible Kernel Patch enables preemption of kernel code in general, unless the processor is currently executing a critical section, protected by a C-macro called 'spin_lock', i.e. a preemption lock on a single processor. A 'Fully Preemptible Kernel' will allow preemption even in critical sections using mutexes instead of preemption locks.*

If a task tries to lock a spinlock, that is already locked by another process, the second task isn't put into a waiting queue to fall asleep as it would be when trying to enter a mutex that has been already reserved by another task. If a task tries to enter a spinlock as the second one, it will make a busy wait, 'spinning around', until the first task unlocks the spinlock. Directly after that it can enter the critical section.

#### Single processor mode

Linux provides only one source code for SMP and uniprocessors, so the spinlocks are placed as a C-precompiler instruction, a `#define spin_lock` macro, in the source code. If the kernel is compiled without the option SMP, i.e. for a single processor, then the spinlock macros are all empty, so that there are no spinlocks on single processor systems at all.

In fact there can't be spinlocks on single processors, because a single processor isn't allowed to go in a mode of spinning, i.e. a busy wait, which wouldn't make sense as a synchronization primitive on a single processor system.

Furthermore, in the standard Linux kernel 2.2 and 2.4 such synchronization mechanisms aren't needed, because the processor normally can work on only one path in the kernel at a time, since the standard kernel is not preemptible. Even on single processor systems it is possible, that there is more than one kernel path active, because the scheduler can be called out of a kernel routine at a Preemption Point. But Preemption Points are always placed at secure points, where the code is reentrant.

The Preemptible Kernel Patch allows preemption of Linux kernel code in general. So, in order to avoid race conditions, the Preemptible Kernel fills the empty spinlock macros in the Linux kernel compiled for a single processor system by CPU-local preemption locks. That way the Preemptible Kernel uses the same critical sections as the Linux SMP kernel. It uses them as NPCS, as non-preemptive critical sections, cp. section 4.5.2.

This way, in general the kernel code, which isn't preemptible in standard Linux, is made preemptible.

The preemption lock of the Preemptible Kernel is implemented as follows:

A new counting semaphore called Preemption Lock Counter (PLC) is introduced that allows preemption of kernel code, if it is zero. If it has any value bigger than zero, it prohibits preemption of kernel code. One can see the functioning of the Preemption Lock Counter in figure 7.6, which shows the Preemption Lock Counter (PLC) in fields with a gray background. That way, all kernel functions and kernel threads are made preemptible, but there are some exceptions to avoid race conditions:

Preemption can´t take place:

- while handling interrupts, i.e. executing ISRs
- while doing Soft-IRQ or 'Bottom Half' processing

| | kernel source code C-macro to expand | single processor | Symmetrical Multi-Processor (SMP) All locks are CPU-local only |
|---|---|---|---|
| not preemptible standard Linux kernel 2.6 | 'spinlock_irqsave()' | interrupt lock | interrupt lock & spinlock, i.e. Busy Wait |
| Preemptible Kernel | 'spinlock_irqsave()' | interrupt lock & preemption lock | interrupt lock & spinlock, i.e. Busy Wait |
| Fully Preemptible Kernel | 'spinlock_irqsave()' | mutex ⇒ preemption & interrupts allowed | mutex ⇒ preemption & interrupts allowed |

Table 7.3: *The Standard Linux Kernel and the Preemptible Kernel expand the C-macro* `'spinlock_irqsave()'` *in the kernel sources into different commands at compile time. In the standard kernel critical sections are only protected from interrupts, if there are interrupt handler that manipulate those data structures, too. Furthermore, in the standard SMP kernel they are also protected by spinlocks, that force other CPUs into a busy wait, until the first processor left the critical section. The Preemptible Kernel just adds a preemption lock on the local CPU for the single processor Linux kernel in order to protect critical sections from concurrent access which can occur, since the Preemptible Kernel makes kernel code preemptible except for critical sections. In a fully preemptible kernel, that uses mutexes instead of preemption or spinlocks to protect critical sections from race conditions, preemption and even interrupts - handled by kernel threads - are allowed during critical sections*

- while holding a spinlock, writelock or readlock, i.e. while the processor is working in a non-preemptive critical section (NPCS) of the kernel
- while the kernel is executing the scheduler
- while initializing a new process in the `fork()` system call

At all other times the Preemptible Kernel allows preemption, so that the Preemptible Kernel provides a shorter PDLT in general.

When the system enters one of the states shown above, e.g. a spinlock, the global Preemption-Lock-Counter (PLC) is incremented, indicating that preemption is now forbidden. When leaving that state, e.g. the spinlock is released, or after an interrupt handler has been processed, the Preemption-Lock-Counter is decremented and a test is made to see, whether preemption has been called for meanwhile. This means, after every critical section, there is a Conditional Preemption Point in the spinlock-macro included. If preemption has been called for meanwhile, the current task is preempted.

This ´polling for preemption´ at the end of a preemption-locked section can happen tens of thousands of times per second on an average computer system. With the Preemption Patch Linux has a more deterministic temporal behaviour with smaller and less maximum latencies. The Preemption Patch makes Linux more reactive and suitable for time-critical tasks, although it causes a small overhead.

When Preemption in kernel mode takes place a large value (0x4000000) is added to the Preemption-Lock-Counter (PLC) in order to bypass some code in the scheduler which isn't useful in this case. Then the scheduler is called and after that the value is subtracted from the PLC.

#### Change of task state transitions

The source code for the most task state transitions is placed in the file `'entry.S'` in the Linux kernel source code. Even in the source code its routines are written in assembly code, since this avoids that compiler have to translate it. Linux kernel developers optimize this code by hand since these routines are used many times per second at every task state transition.

In order to improve the responsiveness of the kernel, the interrupt return code, i.e. assembly language code, is modified by the Preemptible Kernel Patch. That way now preemption is possible and tested

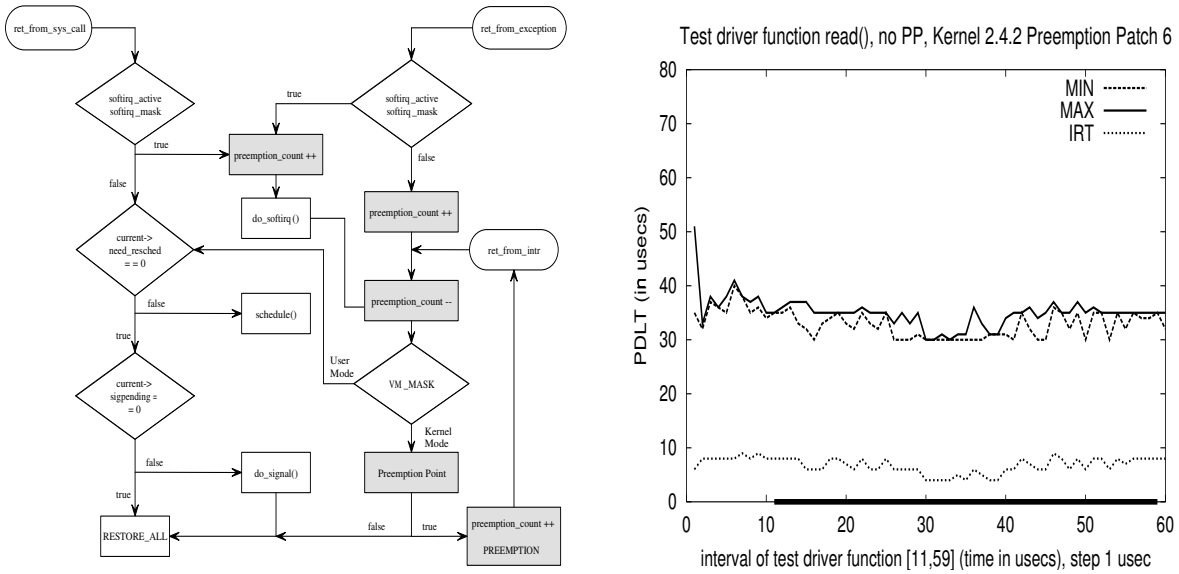



Figure 7.6: *Left part: Flowchart of the file 'entry.S' modified by the Preemption kernel, right part: PDLT of test driver function read(), for-1200-loop without Preemption Points on Linux 2.4.2, but with Preemption Patch 6 by MontaVista Inc.*

for, not only if the interrupt or exception came while the processor has been working in user mode as in standard Linux, but also if the interrupt came while the processor has been working in kernel mode.

The left part of fig.7.6 shows the flowchart of the file `'entry.S'` of the Preemptible Kernel for Linux 2.4.2 with Preemption Patch 6. The Preemption Lock Counter in the fields with a gray background is increased before an interrupt or soft interrupt is processed and it is decreased afterwards. Then the Preemption Lock Counter is tested whether it is zero and it is tested whether preemption has been called for meanwhile. If so, the scheduler is invoked. This figure has to be compared to the flowchart of the standard Linux kernel in fig.7.2, p.101.

#### The PDLT formula for the Preemptible Kernel

For the Preemptible kernel the formulas (7.1) and (7.3) on p.98 have to be modified. Using SMP code also on single processor systems $t_{preemption-delay}$ is replaced by $t_{critical-sections}$ :

$$max(PDLT_{soft-realtime}) = max(t_{IRT}) + max(t_{interrupts}) + max(t_{critical-sections}) + max(t_{scheduling}) \tag{7.4}$$

Since $t_{critical-sections}$ is in most cases much smaller than $t_{preemption-delay}$ in equation (7.1), the Preemptible Kernel improves the soft-realtime capabilities of standard Linux.

But in the Linux 2.4 kernel there are still some very long critical sections, so that $max(t_{critical-sections})$ in equation (7.4) might be not much smaller than $max(t_{preemption-delay})$ in equation (7.3) [101].

Newer versions of the Preemption Patch are also SMP safe, because newer patches also protect - besides spinlocks- per-CPU variables by a preemption lock.

### 7.4.2 Measuring the Preemptible Kernel using the Software Monitor

Hereafter the same measurement principle is used as shown in section 7.3.1. These measurements have been carried out on the same AMD K6 personal computer running at 400 MHz, as used in section 7.3.1 and described in appendix A.

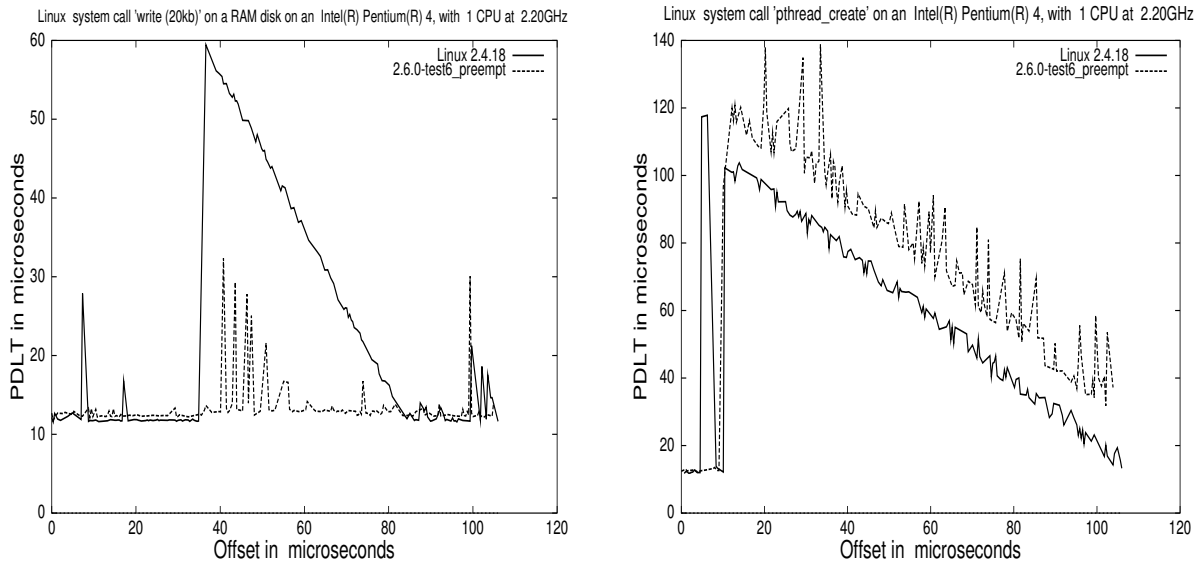



Figure 7.7: *Left part: This measurements shows that Linux 2.6 with activated Preemptible Kernel does indeed reduce the PDLT latency of the system call* `write(20kB)` *on a RAM disk, which is about 60 microseconds in Linux 2.4.18. The smaller spikes of Linux 2.6 could be due to short non-preemptive critical sections. Right part: The figure on the right side shows that even Linux 2.6 with Preemptible Kernel contains long non-preemptive critical sections (NPCS), the one measured here is the system call* `pthread_create` *that creates a new thread and consumes about 120 microseconds on a computer as described in appendix A.2.*

The measurement principle is based on the software monitor, mentioned in section 3.8.2. A thread with higher priority tries to preempt a thread of lower priority, that actually executes a driver in kernel mode. In standard Linux kernel code isn't preemptible, if there are no explicit preemption points in the kernel code, see the left part of fig. 7.3 on p.102.

The test driver function `read()` is executed in kernel mode and it does not contain any spinlock. So the Linux kernel 2.4.2 with Preemption Patch 6 applied to it, should be able to preempt it. The right part of fig.7.6 shows that the Preemptible Kernel patch fulfills the expectations: The system call indeed is preempted, there is no latency triangle shown any more. In the whole diagram of the right part of fig.7.6 the PDLT is in between 30 and 40 microseconds (usec).

When fig.7.5 - the driver function with 23 preemption points - and the right part of fig.7.6 are compared - the for-loop of the driver without preemption points but on a Linux 2.4.2 kernel with the Preemption Patch 6 by MontaVista -, one sees that both methods result in reducing the PDLT triangle of Fig. 7.3, that shows a maximum PDLT of 65 microseconds (usecs). The Preemptible Kernel manages to reduce it to 40 usecs without changing the code of the test driver. Also on higher latencies caused by the test driver the Preemptible Kernel has been able to reduce the PDLT to about 30 up to 40 usecs. By introducing 23 preemption points in the test driver function one reaches a PDLT of about 22 up to 25 usecs during the system call on the same computer system, but as a disadvantage the code of the driver has to be changed to reach that goal.

Comparing the thick lines on the X-Axis of Fig. 7.3 and the right part of fig.7.6 one sees that neither the introduced Conditional Preemption Points change the execution time of the system call significantly, nor the Preemption Patch 6 does.

Fig.7.7 shows on the left side that Linux 2.6 with activated Preemptible Kernel reduces the PDLT latency of the system call `write(20kB)` on a RAM disk, which is about 60 microseconds in Linux 2.4.18. The smaller spikes of Linux 2.6 could be due to short non-preemptive critical sections. The figure on the right side of fig.7.7 shows that even Linux 2.6 with Preemptible Kernel contains long non-preemptive critical sections (NPCS), the one measured here is the system call `pthread_create` that creates a new thread and consumes about 120 microseconds on a computer as described in appendix A.2.

**Impacts of the patches on the performance of the kernel**

The Rhealstone Benchmark, see section 3.6.2, p.39, and its measurement results in section 8.3, p.138, show that the Preemption Patches reduce the performance of the Linux kernel slightly on single processor systems. This is due to the fact, that the Preemption Patch often executes additional code like increasing or decreasing the Preemption Lock Counter (PLC) while being in a critical section, like testing for preemption after a critical section or testing for signals or preemption after an interrupt handler has been executed, cp.fig.7.6.

### 7.4.3 Discussion

Of course, all soft realtime applications should be in general scheduled using the SCHED_FIFO or SCHED_RR policy and be locked into memory, if possible.

Kernel code of the standard Linux kernel 2.2, 2.4 and 2.6 is not preemptible in general. This may produce long latencies on the order of hundreds of milliseconds on current hardware. This section presented the pros and cons of 2 general approaches to reduce the latencies of the non-preemptible Linux kernel code.

On the one hand it is possible to make the Linux kernel 2.2 and 2.4 more responsive by introducing Preemption Points into the kernel. Ingo Molnar's and Andrew Morton's 'Low Latency Patches' follow this concept reducing the longest known Linux kernel latencies by about a factor 10 from the order of hundreds of milliseconds to the order of tens of milliseconds on current hardware. These patches have already proven to be useful for instance to hear audio streams using Linux 2.2 [52].

On the other hand using the SMP critical sections on single processor machines MontaVista presented a kernel Preemption Patch which makes Linux kernel code preemptible except for critical sections. This is a promising approach, but because there are long critical sections in the kernel - long in comparison to 2 context switches - latencies are reduced more efficiently if this patch is combined with a LowLatency Patch for Linux 2.2 and 2.4.

Meanwhile both approaches have been incorporated into the standard Linux kernel 2.6.

Linus Torvalds, the inventor and chief developer of the Linux kernel, stated in an e-mail reply to a Linux audio development group that putting a preemption point into every critical section of the kernel isn't an acceptable concept to make the Linux kernel more realtime-compliant [52]. Nevertheless he accepted to incorporate a few preemption points of the Low Latency patches into the Linux kernel 2.4 to cut down some very long latencies.

A clear disadvantage of introducing preemption points is that it is necessary to patch a Conditional Preemption Point into every part of the kernel code that could introduce a latency longer than e.g. 10 milliseconds (msec). Thus a ´LowLatency Patch´ developer had to test every new driver with the appropriate hardware to avoid latencies.

As a more general concept Linus Torvalds accepted to integrate the code of the Preemptible kernel patch into the standard kernel 2.6. There it can be activated by a compiler switch when configuring the Linux kernel at compile time.

The concept of the Preemption Patch or - 'Preemptible Kernel' since Linux 2.6 - is more general, because a preemptible kernel can preempt all functions without critical sections, although the developers never saw or optimized the code of a special driver for instance.

Nevertheless, the concept of preemption points also in the Preemptible kernel is still useful to cut critical sections into half or more parts, if possible. Therefore some 'spinlock breaking preemption points' have been introduced in the Linux 2.6 kernel in order to decrease some latencies caused by some special critical sections [55].

Nevertheless a guarantee about the value of the longest latency in the Linux kernel - with or without the kernel patches - can't be given, because nobody can test all paths of the Linux kernel code and their mutual interferences.

Also with the Preemptible Kernel the execution time of the longest critical section is difficult to measure, since the duration of a critical section can depend on many parameters including hardware parameters.

Thus any hard realtime guarantees can't be given for the Linux kernel.

## 7.5 Steps towards a 'Fully Preemptible Linux kernel' : Analysis and treatment of remaining critical sections

### 7.5.1 Reducing Latencies caused by long held interrupt & preemption locks

#### The duration of critical sections and interrupt locks

'Long held' preemption locks - or spinlocks in the SMP case - are locks held long compared to the duration of 2 context switches.

Often the same critical sections produce quite different lock durations, i.e. latencies. Looking at the source code one finds the reason:

In the code in between locking and unlocking interrupts or preemption globally there can be `if/else, switch/case` statements or even loops. Which branch -`if` or `else`- is taken or how many times a loop is repeated often depends on the following items:

- The values of the parameters the kernel routine, the lock is part of, has been called with can influence the duration of the critical section.
- The internal state of the operating system. The macro-state depends on maybe thousands of software states and hardware parameters, it depends on kernel options, user behaviour, the load of the system and the history of the system since boot time. Of course not all of these variables influence the latency of a single critical sections, but some of them may do for a certain one.

The way patches try to cope with these latencies in a Linux kernel with Preemption Patch are discussed in the next paragraph. These approaches have been analyzed theoretically in chapter 4.

#### Lock breaking patches - 'Voluntary Preemption' in Linux 2.6

The first possibility to reduce latencies of critical sections is to combine the Preemptible Kernel Patch with a LowLatency Patch.

Those preemption points of the LowLatency Patch which are in the normal kernel code, made preemptible by the Preemptible Kernel Patch, are needless.

But the Low Latency Patches - or Ingo Molnar's 'Voluntary Preemption Patch' for Linux 2.6 - also contain preemption points that break preemption locks on single processors and spinlocks in the SMP case:

```
spin_lock(&lock);
...
if (current->need_resched) {
   spin_unlock(&lock);
   current->state = TASK_RUNNING;
   schedule();
   spin_lock(&lock);
}
...
spin_unlock(&lock);
```

In order to avoid race conditions the preemption lock or spinlock has first to be unlocked and after scheduling, when the task comes to execution again, the preemption or spinlock has to be locked again. This results in more fine grained locks and smaller critical sections.

Several researchers recommended to combine the LowLatency and the Preemption patch [100]. For Linux 2.4 Robert Love offered so called 'spinlock breaking patches' [55], based on the preemption points of the LowLatency patches, that were placed in critical sections. In Linux 2.6 many of these preemption points have been introduced into the official standard Linux kernel.

But since the Linux kernel 2.4 with all its drivers contains about 400 calls to critical sections, it is a lot of work to place preemption points in every critical section. Additionally, it causes a huge test effort, because wrongly placed preemption points can cause data inconsistencies and thus crash the kernel.

For Linux 2.6 Ingo Molnar offers the so called 'Voluntary Preemption Patch', that activates preemption points that have been introduced into the Linux 2.6 kernel for debugging reasons in a C-macro called 'might_sleep'.

#### Conversion of long held spinlocks into mutexes

In this paragraph a more general approach is presented. The second possibility to reduce these latencies is changing long held preemption locks/spinlocks into mutexes in the Preemptible Kernel, first discussed for Linux in the 'kpreempt' project [56]. The advantage of mutex locks is that global locks like preemption locks are replaced by mutual exclusion locks to resources. This replacement does not decrease the term $max(t_{critical\_sections})$ of eq.(7.4), but it reduces its frequency of occurrence, as has been discussed in section 4.6.1, p.66. In the SMP case additionally another task entering a mutual exclusion as the second one on another CPU is put asleep instead of blocking this CPU as a spinlock would do. Therefore mutexes lead to a better average preemptability on single and on SMP systems.

But there are several difficulties to overcome as also stated theoretically in section 4.6 :

- It is possible to replace all those preemption locks by mutexes, which are never entered by any code from an interrupt service routine (ISR) or a SoftIRQ/Bottom Half. But it is not possible to replace those preemption locks, which at the same time also lock interrupts, called `spin_lock_irqsave()`, because such critical sections are used also by interrupt handlers. The problem is that a mutex puts asleep the caller, if it has already been locked before. For an interrupt handler sleeping is forbidden by definition, at maximum it's allowed to execute a short busy wait when accessing an already locked spinlock on a SMP system. A solution for this problem is the IRQ-Thread patch, presented in section 6.2.4 and discussed theoretically in section 4.6.7.
- Apart from that, it's important that the Linux kernel programmer rule to access preemption locks or spinlocks always in the same sequence, determined by their addresses, is extended to mutexes, too, in order not to cause deadlocks, cp. section 4.6.6.
- Another difficulty to overcome caused by the introduction of mutexes into the kernel is the possibility of so called 'unbounded' priority inversion in between 2 tasks using the same kernel mutex in perhaps different kernel routines and other tasks with priorities in between them, cp. section 4.6.4, p.69.

  Since the priority ceiling protocol is only usable for tasks with priorities, which can be statically analyzed, and the calling of Linux system functions can't be previewed, mutexes with a priority inheritance protocol (PIP) have to be implemented in order to avoid the starvation of highly prioritized tasks by less important ones, cp. section 4.6.5, p.70.

  In this thesis two PI-protocols have been implemented: A simple PI-protocol [31], which works fine, when a kernel routine only accesses one mutex at a time, but can still cause priority inversion when a kernel routine accesses nested mutexes.

  Since it isn't sufficient for more complex situations, also a PI-protocol specified by Prof. Victor Yodaiken [103] has been implemented in this thesis that allows nested access to mutexes. Both protocols have been described theoretically in section 4.6.5, p.70.

  A measurement, showing that the priority inheritance protocol works - at least in a special situation - is presented in the next section.

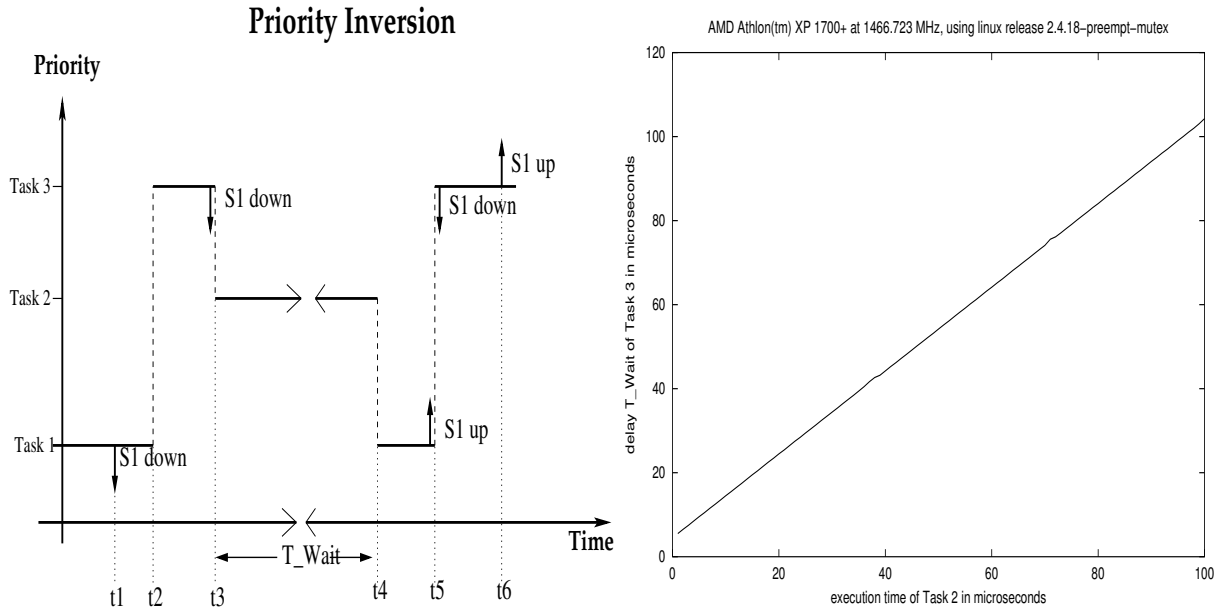



Figure 7.8: *The left figure shows the situation of priority inversion and the right figure shows the time $T_{Wait}$, the high priority task 3 has to wait for the mutex. This waiting time increases proportionally to the execution time of task2 of middle priority.*

### 7.5.2 Priority inheritance protocol for Linux semaphores

Priority inversion situations and priority inheritance protocols have been discussed theoretically in section 4.6.4 on p.69 and section 4.6.5.

In these sections 2 priority inheritance protocols have been presented, one, called SPIP/EPV2, that solves only simple priority inversion problems and one, called PIP_Yod2 according to [103], that solves also priority inversions called by chained mutexes. Both protocols have been implemented as a kernel patch for Linux 2.4 and 2.6 in this thesis, [16, 27] and have been made available under the *GPL2* license [31].

This kernel patch introduces new kernel mutexes with priority inheritance protocol into the Linux kernel. One possibility is to use them to transform kernel preemption locks - or spinlocks in the SMP case - into mutexes. Another interesting possibility is to make them available for applications in user space, e.g. via a Linux kernel module, the device file `/dev/mutex` of the System V Virtual File System (VFS), and a library the applications have to link with.

It is impossible to set up and measure all possible priority inversion situations, especially for chained mutexes, and proving the mathematical correctness isn't easy as all [103]. Thus, in this section only measurements for special test cases are presented, to prove that the priority inheritance mechanisms work as specified at least in these cases.

In user applications priority inheritance can be useful, if e.g. a Linux semaphore or mutex is used by several SCHED_FIFO, i.e. soft-realtime tasks of different priority or by soft-realtime tasks and tasks of normal priority SCHED_NORMAL.

In the latter case, if a SCHED_NORMAL process allocates a mutex or semaphore, a SCHED_FIFO task that requires the same mutex is put asleep. That way a number of SCHED_NORMAL tasks can cause a priority inversion situation as described in section 4.6.4.

With the mutexes with priority inheritance the priority of the SCHED_NORMAL process can be increased to the priority of the SCHED_FIFO process while it holds the mutex, which avoids priority inversion situations.

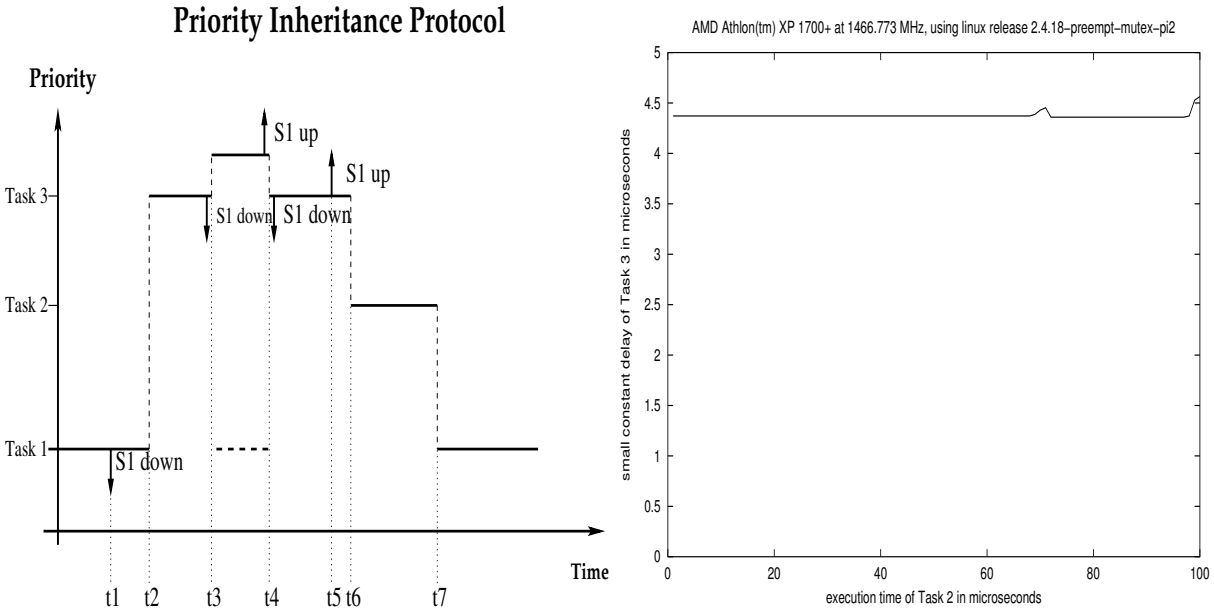



Figure 7.9: *The left figure shows how the priority inversion protocol does prevent priority inversion. Therefore the time $T_{wait}$, task 3 waits for the mutex now only depends on the constant execution time of the critical section, needed by task 1 until it releases the mutex.*

#### A program to test the priority inheritance protocol

The situation of the test program using mutexes without priority inversion is shown in the left part of fig.7.8.

In the priority inversion test program [27] task 2 is also a SCHED_FIFO process. For the test the execution time of task 2, the task of middle priority, varies. The test program measures the time $T_{Wait}$ the high priority task 3 has to wait for the binary semaphore or mutex. The right part of fig.7.8 shows that standard Linux semaphores show priority inversion, since $T_{Wait}$ increases linear with the execution time of task 2, that is not holding or requiring the semaphore.

On the other hand, if the test program uses the new Linux kernel mutexes with priority inheritance protocol, the time $T_{Wait}$ remains stable at only 4.4 microseconds, as the right part of fig. 7.9 shows, even if the execution time of task 2 increases.

4.4 microseconds is the constant time, that is really used by task 1 to execute the critical section, protected by mutexes. Thus, this is the minimum time, the high priority task 3 has to wait due to the mutual exclusion mechanism. In section 4.6.1, p.66, this minimum time has been referred to as the Kernel Latency Time (KLT) [58].

With the priority inheritance protocol the length of the execution time of task 2 doesn't matter any more in fig.7.9, since the priority of task 1 is increased up to the priority of task 3, until task 1 releases the mutex. This mechanism of priority inheritance is shown in the left part of fig.7.9.

These measurements have been done for both priority inheritance protocols. The more complex PI-protocol PIP_Yod2, as proposed by [103], has been tested for a case of chained mutexes, too. The results were all successful, and thus similar to the right part of fig.7.9, [31, 16].

### 7.5.3 Further Mutex implementations: Robust Mutexes/FuSYNs of the OSDL Laboratories

The OSDL Laboratories [72], the Open Source Development Laboratories, developed a special Linux for telecommunication companies, named Carrier Grade Linux. This project implemented so called 'Robust Mutexes' with the following qualities:

- A resource protected by a locked mutex is unlocked by the mutex implementation itself, if the process that locked the mutex dies without unlocking the mutex before. This is an important feature. Mutexes with that quality are called 'Robust Mutexes'.
- Robust Mutexes implement a priority inheritance protocol
- and a deadlock detection algorithm for complex deadlock situations, but as discussed in section 4.6.6 the occurrence of deadlocks can be avoided by a total order of mutexes and thus such an algorithm adds an unnecessary complexity to the mutex implementation.
- Robust Mutexes are POSIX compatible
- If there are several tasks waiting on a robust mutex, that one of highest priority is reawaken as the first one
- Robust Mutexes are also called FUSYN, i.e. an abbreviation for 'Fast User SYNChronization Primitives', or Futex, it means Fast Userspace Mutex. That name expresses that in the normal case, if the mutex is free and a process can lock it, no system call has to be done and therefore all actions necessary can be done in a very fast way remaining in user space. Kernel support is only needed, if the mutex is already locked and the second task must be blocked.
- Robust Mutexes or FUSYN consist of two basic elements, a binary semaphore called 'fulock' and a waiting queue called 'fuqueue'.
- Because of the complex algorithms the source code is rather complex and the features above can't be easily separated when looking at the source code. This was the reason, MontaVista did not choose this code to implement their first fully preemptible Linux kernel in their 'MontaVista Linux Open Source Real Time project', cp. section 7.5.4 on p. 115.

Robust mutexes / FUSYNs can be used in user space or in kernel space. They have been developed to be used in projects of the telecommunication industry, this means mainly in userspace applications. The project did not intend to create a fully preemptible kernel based on Robust Mutexes / FUSYNs.

#### The PDLT formula for the Fully Preemptible Kernel

For the Fully Preemptible kernel with IRQ-Thread patch and priority inheritance protocol the formulas (7.1) on p.98 and (7.4) on p.107 have to be modified.

For a fully preemptible kernel the formula is:

$$PDLT_{soft-realtime} = t_{IRT,IRQ-Threads} + t_{interrupts} + t_{scheduling} \tag{7.5}$$

The interrupt response time (IRT) of the scheduled IRQ-threads consists of

$$t_{IRT,IRQ-Threads} = t_{IRT} + t_{scheduling} \tag{7.6}$$

$t_{IRT,IRQ-Threads}$ contains an additional $t_{scheduling}$, because of the IRQ-Thread patch that makes interrupt service routines subject to the scheduler.

The advantage of eq.(7.5), and thus of a fully preemptible kernel, compared to eq.(7.1) is that the term $t_{preemption-delay}$ or $t_{critical-sections}$ is missing. That way, the Fully Preemptible Kernel improves the soft-realtime capabilities of standard Linux in general.

Combining the last 2 equations leads to the following:

$$PDLT_{soft-realtime} = t_{IRT} + t_{interrupts} + 2 * t_{scheduling} \tag{7.7}$$

$$max(PDLT_{soft-realtime}) = max(t_{IRT}) + max(t_{interrupts}) + 2 * max(t_{scheduling}) \tag{7.8}$$

But a fully preemptible kernel has also a new latency that occurs in rare cases:

In the probably rare cases, when the mutex is already locked by another task, when it is required by the soft-realtime task, the $KLT_{max}$, cp. eq.(4.2) on p.67, will arise, a latency that never occurs in a kernel that only uses NPCS, i.e. a global preemption lock.

This means that a fully preemptible kernel does not improve the determinism or the hard realtime capabilities of a kernel, as in the worst case, when the mutex is locked, the (soft-)realtime task must still wait for $max(t_{critical-sections}) + 2 * max(t_{scheduling})$ more than in the Preemptible Kernel using only non-preemptive critical sections (NPCS), cp. eq.(7.4) on p.107, until it finishes what it had to do. Therefore, it's clear, that a hard realtime operating system shouldn't use mutexes in its kernel, as [104] states, too. Instead of mutexes in hard RT operating systems only short NPCS should be used, so that eq.(7.4), p.107, applies to such systems.

If the mutexes didn't contain a priority inheritance protocol (PIP) a term of possibly unbounded priority inversion $max(t_{priority\_inversion})$, showing up only in rare cases, had to be added to eq.(4.2) on p.67.

If the operating system contains nested mutexes, a transitive priority inheritance protocol as PIP_Yod2, cp. section 4.6.5, p.70, is needed. Unfortunately, kernels of standard operating systems like the Linux kernel, contain nested mutexes.

In this case, eq.(4.2) on p.67 must - for the worst case of transitive priority inheritance - be modified in the following way:

$$max(KLT) = 2 * max(t_{scheduling}) + max(t_{critical-sections}) + \sum_{i=1}^{m} \{max(t_{scheduling}) + max(t_{critical-sections})\} \tag{7.9}$$

where $m$ is the number of times the priority is inherited to a new task, because of the transitivity of the priority inheritance protocol, cp. section 4.6.5, p.70.

Considering eq.(7.9) it's clear, that a fully preemptible kernel can't be used in a hard realtime operating systems and that it can also have disadvantages for time-critical tasks.

Nevertheless, implementations of the fully preemptible Linux kernel exist. They are now on the test bench, and the future will show whether the soft-realtime advantages of running faster in general, cp. eq.(7.5) will be more valuable to the users than being slower in rare cases, described by eq.(7.9).

### 7.5.4 Implementations on the way to a fully preemptible Linux kernel

In this section different implementations of a 'fully preemptible' Linux kernel are presented. All of them use the principle of the IRQ-Thread patch, that has been described in sections 4.6.7 and 6.2.4. All of them convert most of the preemption locks - spinlocks in the SMP case - of the Linux kernel into mutexes. Advantages and disadvantages of the concept of a fully preemptible kernel have been discussed in section 4.6.

#### The Timesys approach - an implementation

Timesys Inc.[96] adopted the Preemptible Kernel and implemented some additional changes to the standard Linux kernel 2.4.7, released partly under GPL/LGPL and partly under a proprietary license:

- The developers at Timesys Inc. changed all Interrupt Service Routines (ISR) - besides the timer ISR - into kernel threads and scheduled these kernel threads at a very high soft-realtime SCHED_FIFO priority. Therefore, with this Timesys Linux kernel it's possible to assign priorities to interrupts or even prioritize a realtime process higher than some interrupts.

  Furthermore, Timesys extended the fixed priority soft realtime scheduler (SCHED_FIFO) to schedule them, normally at its new highest priority 509.

- Since every ISR, i.e. every interrupt handler, is a scheduled kernel thread in the Timesys kernel, it's possible to put even an interrupt handler asleep in the Timesys kernel. So the Timesys kernel could replace even the joined spinlock-interrupt locks `spin_lock_ irqsave()` by mutexes.

  Timesys Inc. converted 276 preemption locks - spinlocks in the SMP case - of Linux 2.4.7 into mutexes by redefining the kernel macro of the Linux kernel source code: `#define spinlock_t mutex_t`. 48 preemption locks remained unchanged. They were only renamed to `old_spinlock_t`. That way, even the Timesys kernel possesses preemption locks or spinlocks as synchronization primitives, called `old_spin_lock_irqsave`, because a few atomic operations with preemption lock are needed in every operating system. The Timesys kernel for instance uses them to protect operations on its own mutexes from being intercepted. But spinlocks are used only to protect very small critical regions in the Timesys Linux kernel, most longer ones are protected only by mutexes.

- Such a mutex implementation needs -as already said- a priority inheritance protocol (PIP) at least for realtime constraints. But while the other parts are available under GPL/LGPL, the Timesys PI-protocol is plugged into the Linux kernel using pointers to functions out of the bulk Linux kernel into Timesys kernel modules, that are not freely - only commercially - available. The source code of these modules is encrypted. Therefore, it was not possible to analyze their PI-protocol in this thesis. In one of their white papers Timesys mentioned the protocols of Sha, Rjakumar and Sathaye [91].

#### The MontaVista Linux Open Source Real Time Project using the mutexes with priority inheritance implemented in this thesis

Since the end of 2003 MontaVista prepared a new project, that has been called 'MontaVista Linux Open Source Real Time Project' [64].

This project had two goals:

- Replacing all preemption locks - spinlocks in the SMP case - with mutexes in the new Linux 2.6 kernel. The mutexes should implement a priority inheritance protocol.

- MontaVista did always intend to give the development later on away to the Linux Open Source kernel community, i.e. to other kernel developers in order to get rid of the costs of maintaining this kernel mutex patch and having to port them to every new kernel version on different platforms the Linux kernel runs on. This concept means that both would have an advantage: The Linux Open Source community will benefit from the initial work, done at MontaVista Software Inc., and later on MontaVista and the world will benefit from a fully preemptible Linux kernel optimized by kernel developers all over the world.

As this thesis at the University of Federal Armed Forces of Germany (UniBwM) had published a mutex implementation with priority inheritance protocol for the Linux kernel 2.4 using the GPL2 license of the Linux kernel in May 2003 [16, 25], that can be downloaded by everyone in the Internet [31], MontaVista Software Inc. decided in the beginning of 2004 to use the implementation of mutexes of this thesis for their fully preemptible Linux kernel of their *'MontaVista Linux Open Source Real Time Project'*. For this cooperation we ported the PMutex implementation - see the SPIP/EPV2 protocol in section 4.6.5

- to work also in Linux 2.6. After this had been accomplished and tested, MontaVista published the 'MontaVista Linux Open Source Realtime Project' in October 2004, and named our researching group at UniBwM as an early contributor [19].

MontaVista Software Inc. converted 603 preemption locks - spinlocks in the SMP case - of Linux 2.6.9 into mutexes by redefining the kernel macro of the Linux kernel source code. 30 preemption locks remain unchanged.

Measurements of the IRT of the MontaVista kernel have been presented in section 6.2.4, p.93. Measurements of the simple priority inheritance protocol SPIP/EPV2, used in the MontaVista Linux kernel, discussed theoretically in section 4.6.4, p.69, and section 4.6.5., have been presented in section 7.5.2.

### Ingo Molnar's Realtime (RT) patches for Linux 2.6

The Timesys kernel 2.4.7 and the first MontaVista kernel 2.6.9 as well didn't run stable on every possible hardware. Probably, too many preemption locks had been converted into mutexes. Ingo Molnar, Linux kernel developer at Red Hat Inc., estimated, there are about 90 preemption locks in the Linux 2.6 kernel that can't be changed into mutexes without risking the stability of the Linux kernel, as they protect actions that have to be done in one atomic operation, for instance on timer chips or the mutex implementation itself.

After the world-wide presentation of the 'MontaVista Linux Open Source Real Time Project' Ingo Molnar liked the idea of replacing all preemption locks / spinlocks in the SMP kernel by mutexes in order to create a fully preemptible Linux kernel [105, 106].

But he preferred to start from the existing semaphore implementation of the standard Linux kernel to replace most of the preemption locks / spinlocks of the standard kernel, probably because the 'PMutexes' of this thesis used assembler code on the x86 to perform better in the common case [31, 25, 16]. This restricted the use of 'PMutexes' to Intel compatible PCs. Because the Linux kernel runs on many other platforms too, Ingo Molnar chose the existing Linux kernel semaphores to build his new mutex implementation on.

These modified Linux kernel mutexes have been part of Ingo Molnar's RT (Realtime) patches for Linux 2.6 [66].

Ingo Molnar maintains and optimizes his RT patch, published using the GPL2 license the Linux kernel uses, too [66]. After patching the Linux kernel code, the kernel configuration PREEMPT_RT has to be chosen in order to create a 'Fully Preemptible Linux kernel'. Then the kernel must be built, linked and booted.

Of course, Ingo Molnar's implementation needs also interrupts, executed by kernel threads. Therefore the kernel configuration option 'PREEMPT_RT' implies the option 'PREEMPT_HARDIRQS' in order to create the needed IRQ-Threads. Since the early Linux 2.6 kernel versions the IRQ-thread patch is a configurable option at compile time of the standard Linux kernel.

Some kernel developers like Thomas Gleixner working at the LiberRTOS project [14], that is partly based on ideas of UTIME [41, 50], helped Ingo Molnar to reimplement the interrupt subsystem of the Linux kernel and to integrate the High Resolution Timer Patch [2] into his newly created RT patch [66], i.e. the RealTime patch.

So this RT patch - MontaVista shows a list of contributors [68] - is a collection of extensions to the current Linux 2.6 kernel, which could become part of one of the future stable Linux kernel releases.

In fact, there are already former parts of the RT patch for early Linux 2.6 kernels that are now part of the standard vanilla Linux 2.6.18 kernel, namely Ingo Molnar's mutex implementation with a priority inheritance protocol [107]. This PI-protocol can be used by mutexes in userspace applications as well as by the mutexes of the RT patch in kernel space when building a fully preemptible Linux kernel. That way, the RT patch for newer versions of the Linux kernel is smaller than the versions of the patch before have been.

Probably, in future more and more parts of the RT patch will be integrated into the standard Linux kernel as options that can be chosen before the kernel is compiled.

# Chapter 8

# Analyzing the Linux timing subsystem to suggest improvements

## 8.1 Time base of the standard Linux kernel

### 8.1.1 Tasks with timing requirements

Time-critical tasks often have to run periodically or have to be started at special points in time. To fulfill such requirements every operating system keeps track of the time. This is done by different operating systems with a different precision. In this section latencies of the Linux time management are examined in detail.

Furthermore many applications and algorithms are in need of exact timing information.

### 8.1.2 The timer interrupt - the heart beat of an operating system

The common way to keep track of the proceeding time is to count the number of oscillations of an oscillator which is known to run with a certain rather stable frequency. Several chips of a modern PC contain such an oscillator, providing clock functionalities for some parts of the PC or generating a clock pulse like the processor clock cycles or the bus clock cycles.

While computer chips of today keep track on very short clock cycles in the range of nanoseconds and beyond, an operating system like e.g. Windows or Linux, is informed periodically only at special points in time, e.g. once every N milliseconds (msec), by a timer interrupt that the time has proceeded. That way the timing base of the operating system normally is less precise than the timing of the underlying hardware.

The standard Linux Kernel programs one of the timer chips of the PC to generate a timer interrupt periodically.

The sources of the Linux kernel do not contain the period of the timer interrupt, but the inverse, its frequency. The file `linux/arch/i386/lib/param.h` contains the architecture dependent C-macro HZ which means the frequency of the Linux timer interrupt in Hertz.

For instance up to Linux 2.4 `HZ` equals 100 on the Intel architecture, indicating that the timer interrupt comes periodically all 10 milliseconds (msec).

In Linux 2.6 `HZ` equals 1000, i.e. all 1 msec a timer interrupt is executed on the Intel platform. After every timer interrupt the timer interrupt service routine increments the global kernel variable `unsigned long volatile jiffies` by 1. This counter `jiffies` is set to zero only once at boot time.

Since many important timing services of the standard Linux kernel are calculated on the basis of `jiffies` they have a timing resolution not more precise than the interval of the timer interrupt, i.e. in the range of milliseconds.

### 8.1.3 Getting precise time stamps in standard Linux

Since some chips like the CPU have a higher timing resolution, see section 3.7.1, also in standard Linux it is possible to get a time stamp from these clocks using e.g. platform dependent assembler instructions of the platform. Some Linux timing services already do so, e.g. the function `gettimeofday()`.

### 8.1.4 The timing services of the standard Linux kernel

An important requirement an operating system has to fulfill is to schedule, reawake and execute a task at a certain point of time in the future.

When a Linux user task wants to sleep for a period of e.g. 50 msec, it invokes the system call `nanosleep()` with the right parameter. The invoked system call fills the struct of a Linux kernel timer with the appropriate values and puts it into a list of kernel timers, i.e. a list of all events in the system to be scheduled at a fixed time in future. After that the task is put asleep. It's the duty of the operating system to reawake the task in time and to bring it to execution on the processor without a long delay or latency.

The following pseudo code shows the structure of a timer of the Linux 2.4.4 kernel:

```
struct timer_list {
        struct list_head list;
        unsigned long expires;
        unsigned long data;
        void (*function)(unsigned long);
};
```

The field `´expires´` is set to a `jiffie` value in the future, when the timer shall expire. Every time a timer interrupt occurs in Linux, the timer Interrupt Service Routine (ISR) activates the timer Soft-Interrupt, the Soft-IRQ, see section 6.2.1. When the Soft-Interrupt is handled, the kernel timers in the kernel timer list are checked, whether the value of their `expire`-field is smaller than the actual `jiffies` value. If so, a certain timer has expired and the Timer Soft-IRQ starts executing the function the pointer of the `'function'` field points to. The field `'data'` contains the parameter that is passed to this function. In the case of `nanosleep()` the `data` field contains a pointer to the process that has to be reawaken, i.e. to be put into the queue of tasks that are ready to run. This is done by the kernel routine, the `function` field points at.

This way, the task that called `nanosleep(..)` before, is awakened after sleeping for the predefined time and some *delay*.

In Linux a task, that is currently the soft-realtime task of highest priority in the system, and that has been scheduled to reawake at a certain time, will be delayed according to the following formula:

$$delay_{task}^{timed} = timing\_delay + PDLT \tag{8.1}$$

As discussed in section 7.1 any latency caused by the lack of preemptability of the operating system will contribute to the $PDLT$.

This section focuses on the *timing_delay*, that is caused by the timer management of the Linux kernel and how it can be reduced.

The accuracy of the Linux timing base can be tested in two ways: The first is to measure the accuracy of the start of periodical tasks, the second is to do the same for so called 'one-shot' tasks, i.e. tasks that are started only once at a certain time.

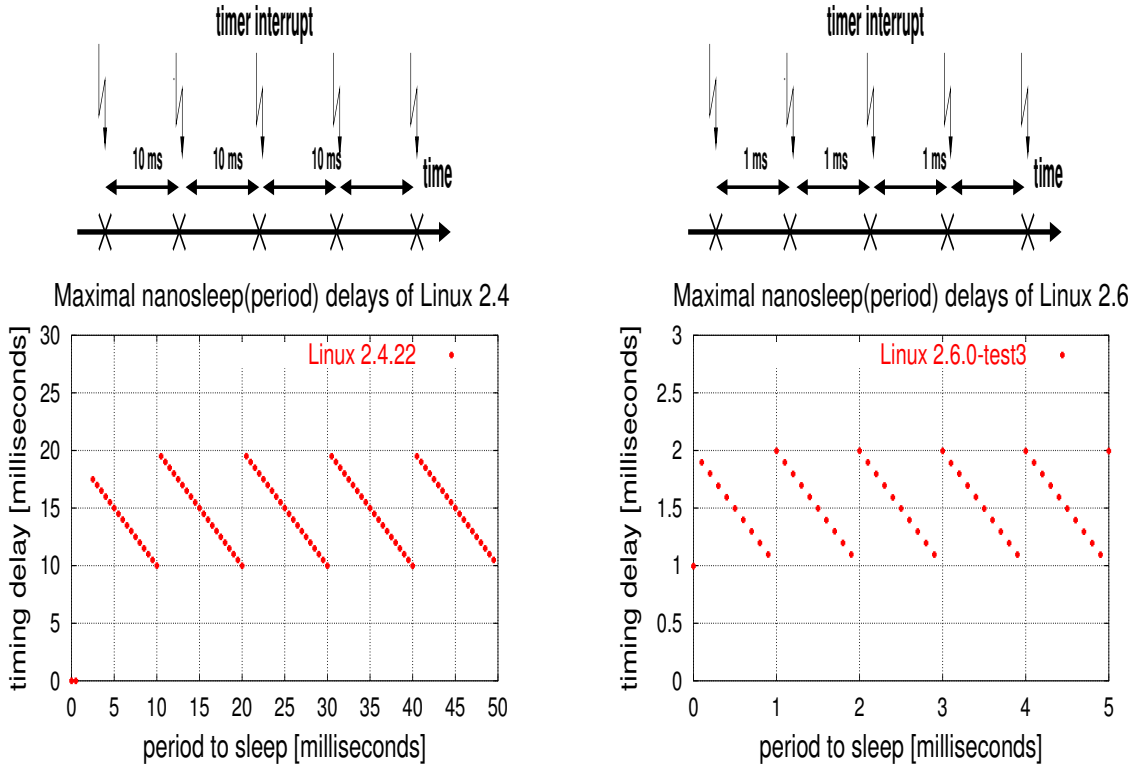



Figure 8.1: Left part: *Linux 2.4, Maximal timing_delay of a periodic soft-realtime task in the standard Linux kernel 2.4. The sleeping period of the task is varied from 0 up to 50 msec.* Right part: *The same program, but here executed on Linux 2.6. The latencies in Linux 2.6 are only a tenth of the delays in Linux 2.4 on the left side, since in Linux 2.6 the internal time base is updated 10 times more often by the timer interrupt.*

### 8.1.5 Timing in Linux 2.4

Tracking of time in computers is discrete like any other operation. In Linux 2.0, 2.2 and 2.4 every 10*msec* the clock of an Intel compatible PC - i.e. the 8254 timer chip or the APIC, see section 3.7.1 - generates a timer interrupt. Every time a timer interrupt comes, an internal variable of the kernel called '`jiffies`' is incremented by one, indicating that another 10 msec have passed.

The general problem with discrete time tracking is that the real time flows continously. Therefore, since the timing variable `'jiffies'` is updated only in intervals, it is already outdated, when its value is read by a process.

In the case of Linux 2.4 the value in the timing variable '`jiffies`' can be outdated for in between 0 msec, when the timer interrupt just came, and up to 10 msec, if it's just the point in time before the next timer interrupt occurs.

The policy of the Linux kernel is to assure a process, that wants to sleep for a certain interval, is never woken up too early.

The timing behaviour of a system can be studied by regarding a periodical process running on the standard Linux kernel, that intends to sleep for an interval called `period` by calling `nanosleep(period)`:

The test program used for the measurements in this and the following sections has been presented by Phil Wilshire at the second Real Time Linux Workshop [101]. The process is scheduled using the SCHED_FIFO scheduling policy. Typical values for `period` are in between 0 and 50 milliseconds. The pseudo code is given below:

```
for(int period = 0; period<= 50 msec; period++)
{
        get_time_stamp_from_TSC(t1);
        nanosleep(period)
        get_time_stamp_from_TSC(t2);
        time_slept = t2-t1;
        delay_timed_task = time_slept - period;
}
```

The time `t2-t1` is determined using the 64 bit Time Stamp Register (TSC) of the x86 processor, see section 3.7.1.

Even on a Linux system which is idle or has a preemptible kernel, there often is a considerable delay, i.e. the process isn't waked up by the Linux kernel in time.

When calling `nanosleep(period)` the kernel adds the `period` to sleep to the current `jiffies` value:

$$expired = jiffies + rounded\_period + security\_interval; \tag{8.2}$$

The *security_interval* is 10 milliseconds (ms) for Linux 2.4 and 2 ms for Linux 2.6. The intention of the *security_interval* is that a oneshot-task never sleeps for a shorter period than it has wished to sleep, although the $jiffies$ value is already outdated when the task puts itself asleep.

After that, the process is put into a list, that the timer interrupt handler will look through later on. The process is put asleep and the scheduler chooses another process to run. Every timer interrupt the timer interrupt handler looks through the list and will wake up the process, if the current `jiffies` value is higher than the `expired` value of the sleeping process.

Figures 8.1 to 8.4 show measurements using the test program above while varying the `period`, the process intends to sleep between 0 and 50 milliseconds, measured on different Linux kernels and with different kernel patches.

The left part of fig.8.1 shows the maximum *timing_delay* that occurs on a Linux 2.4. This delay is only caused by the Linux timing system, so it's best called *timing_delay*.

$$delay_{task}^{timed} = timing\_delay + PDLT = time\_slept - period \tag{8.3}$$

When the computer is idle, the PDLT can be assumed as constant and only in the range of some tens of microseconds on modern personal computers. Such a small PDLT is very low compared to *timing_delays* of several milliseconds in fig.8.1 and 8.2. So for *timing_delays* of 10 or 1 milliseconds there is:

$$delay_{task}^{timed} = timing\_delay + PDLT \approx timing\_delay \tag{8.4}$$

That way, eq.(8.7) also explains the left part of fig.8.1.

Cases with load, where the PDLT isn't small any more, are discussed in section 8.2.6.

Ideally *timing_delay* should be zero, but this isn't the case in real computers and operating systems.

On the contrary the measurements show that even on a Linux system which is idle, there is a considerable *timing_delay* for a short periodical task, i.e. the task isn't waked up by the Linux kernel in time.

The measurement data in the left part of fig.8.1 is described by the following formula for a periodic task with a short execution time in Linux 2.4:

$$delay_{task}^{timed} = time\_slept_{max}^{Linux2.4} - period = 10msec + ceil\left(\frac{period}{10msec}\right) * 10msec + PDLT - period \tag{8.5}$$

The function $ceil(x)$ means rounding up to the next integer, that is bigger or equals the real number $x$. The last equation can be simplified:

$$time\_slept_{max}^{Linux2.4} = 10msec + ceil\left(\frac{period}{10msec}\right) * 10msec + PDLT \tag{8.6}$$

Combining eq.(8.3) and eq.(8.5) it follows, cp. the left part of fig.8.1:

$$timing\_delay_{max}^{Linux2.4} = 10msec + ceil\left(\frac{period}{10msec}\right) * 10msec - period \tag{8.7}$$

How can the terms in eq.(8.7) be explained ?

- Oneshot task: Since the `period` the process wants to sleep is added to the current value of `jiffies`, that is already outdated from 0 msec up to 10 milliseconds (msec), the Linux 2.4 kernel always adds 10 msec as a default value. This is done in order never to wake up a task too early. This is the explanation of the first term of eq.(8.7).

  Thus, a 'one-shot' task that wants to sleep only once for 50 msec, is put asleep for any time in between 50 and 60 msec.

- Periodical task: Periodical tasks do synchronize themselves with the timer interrupt coming all 10 ms in Linux 2.4, as they are awakened by the timer interrupt. If the intended `period` to sleep is a multiple of $10msec$, the second term in eq.(8.7) equals this period and so doesn't contribute to the $timing\_delay^{Linux2.4}$ of eq.(8.7).

  That way in Linux 2.4 the maximum *timing_delay*, i.e. the time the process sleeps too long, for a short periodical task, that intends to sleep for a multiple of 10 msec, is 10 milliseconds (msec), cp. the left part of fig.8.1:

$$timing\_delay_{max}^{Linux2.4}(period = n * 10msec) = 10msec \tag{8.8}$$

- A one-shot task, that wants to sleep for 3 msec, will be put asleep in between 10 up to 20 msec until it is reawakened by the timer interrupt in Linux 2.4.

  If a program wants to sleep periodically for a fraction of $1/HZ$, for ex. for 5 or 24 milliseconds in Linux 2.4, then the second term rounds up the desired period, the process wants to sleep, to the next $10msec$, because the sleeping process can only be woken up by the periodical timer interrupt.

  As the left part of fig. 8.1 shows, it's not very useful to make a task to sleep a fraction of $1/HZ$, because the Linux kernel 2.4 rounds the fraction up to the next 10 millisecond border, so that a one-shot task, that wants to sleep once for 14 msec, will actually sleep in between 6 to 16 msec too long, depending when after the last timer interrupt the task has started to sleep.

### Busy Waits

The only way to avoid the *timing_delay* is a busy wait that is implemented for soft-realtime tasks - i.e. for tasks using the scheduling policy: SCHED_FIFO or SCHED_RR - that want to wait 2 msec or beyond in Linux 2.4. The biggest disadvantage of a busy wait is that it blocks the processor for the whole time, so that no other process can get running on this processor meanwhile.

One can see that the busy wait doesn't cause *timing_delays* in fig.8.1, as the timing delay is zero or in the range of microseconds for a period of 1 and 2 msec on the left side of fig.8.1.

In the source code of Linux 2.6 and in the right part of fig.8.1 one can see that Linux 2.6 doesn't contain a busy wait any more.

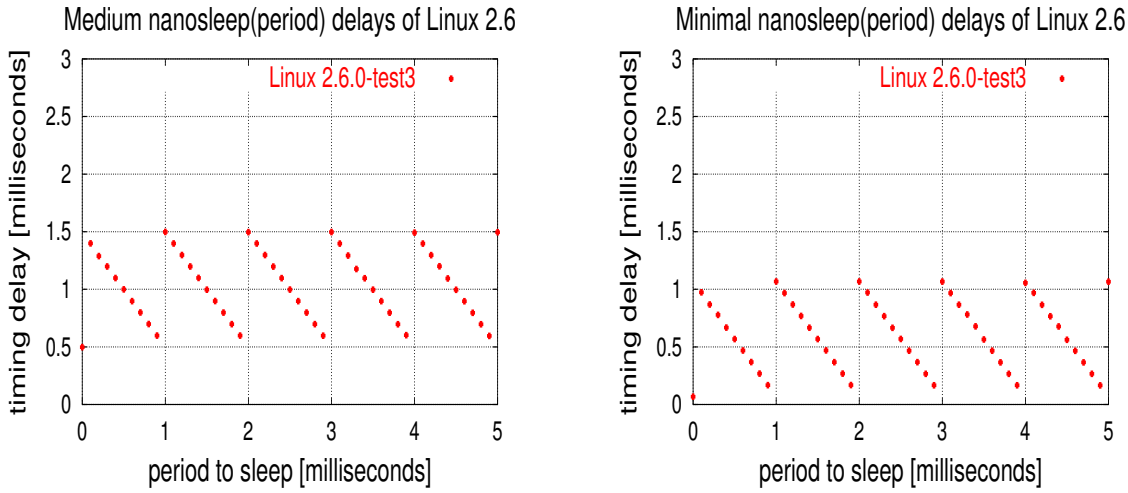



Figure 8.2: Left part: *Linux 2.6, periodical task running itself for 0.5 msec, with an average timing_delay in Linux 2.6,* Right part: *Linux 2.6, periodical task running itself for 0.93 msec, this is about the minimum timing_delay for every task in Linux 2.6*

### 8.1.6 Timing in Linux 2.6

Since time-critical tasks need a precise time base and the hardware of modern personal computers already provides a more precise timing, improving the granularity of the Linux timing base is an important goal in order to make the Linux kernel more realtime compliant.

#### One timer interrupt every millisecond

In standard Linux 2.6 therefore the macro `HZ` in the file `linux/arch/i386/lib/param.h` has been changed to 1000 Hertz, so that the timer interrupt now comes every $\frac{1}{HZ} = 1$ msec.

This way the timing resolution of the Linux kernel 2.6 is ten times higher than that of the stable Linux kernels before.

This leads to a considerable decrease of the delay of the Linux timing system, as the right part of fig.8.1 and eq.(8.9) show.

Additionally since Linux 2.6 the new code of the POSIX Timer project - see section 8.2.2, [2] - is used by `nanosleep()`. Therefore eq.(8.6) has to be changed for Linux 2.6 into:

$$time\_slept_{max}^{Linux2.6} = 2msec + floor\left(\frac{period}{1msec}\right) * 1msec + PDLT \tag{8.9}$$

The function $floor(x)$ means rounding down to the next integer, that is smaller or equal the real number $x$. *period* must be put into the formula in the unit of `milliseconds`.

Using eq.(8.3) it follows for a periodical task with an execution time much smaller than 1 millisecond the maximum *timing_delay*, cp. the right part of fig.8.1 :

$$\begin{aligned} timing\_delay_{max}^{Linux2.6} &= time\_slept - period - PDLT \\ &= 2msec + floor\left(\frac{period}{1msec}\right) * 1msec - period \end{aligned} \tag{8.10}$$

For periods $period = n * 1msec$, that are multiples of 1 msec, eq.(8.10) simplifies to a maximum *timing_delay* of a short periodical task in Linux 2.6:

$$timing\_delay_{max}^{Linux2.6}(period = n * 1msec) = 2msec \tag{8.11}$$

In fig.8.2 it can be seen that the *timing_delay* for periodical tasks with longer execution times is less than the maximum of 2 msec.

So a oneshot task that wants only once to sleep for 1 msec will sleep a random duration in between 2 and 3 milliseconds in Linux 2.6, depending on how outdated the $jiffies$ value already is. This means a *timing_delay* of in between 1 and 2 msec for all tasks, which have a wished sleeping time of $n * 1msec$ in Linux 2.6, cp. the right part of fig.8.1 and fig.8.2. Whether the *timing_delay* is 1 or 2 msec depends on the duration of the periodical task which is different for all these 3 figures.

If a oneshot process intends to sleep e.g. for a duration of 33 msec, it will sleep in between 34 and 35 msec.

The second term of eq.(8.9) rounds down to the previous full millisecond, so if a process wants to sleep for 1.3 milliseconds it in fact sleeps in between 2 and 3 msec.

| | 100 Timer interrupts per second Linux 2.4.18 | 1000 Timer Interrupts per second Linux 2.6.0-test6 | 10000 Timer Interrupts per second **hypothetical** |
|---|---|---|---|
| Period of timer interrupt | 10 msec | 1 msec | 0.1 msec |
| HZ, number of interrupts/sec | 100 | 1000 | 10000 |
| number of TimerSoftIRQ/sec | 100 | 1000 | 10000 |
| Duration of 1 ISR | 6.9 usec | 8.3 usec | |
| Duration of 1 TimerSoftIRQ | 1.0 usec | 0.6 usec | |
| Elapsed time per second | 0.79 msec | 8.9 msec | |
| **CPU usage** | **0.08 %** | **0.9 %** | **9 %** |

Table 8.1: *Overhead of different Linux timer-interrupt frequencies on a Pentium 4 with 2.2 GHz*

### Quantifying the overhead when increasing HZ

Considering performance issues, it is a disadvantage, that with $HZ = 1000$ in Linux 2.6 the timer interrupt comes with a frequency 10 times higher than in Linux 2.4.

To examine the overhead the time the Linux timing subsystem needs to fulfill its task after a timer interrupt occurred has been measured using the Time Stamp Counter (TSC) of an Intel Pentium 4 processor. The results are shown in table 8.1.

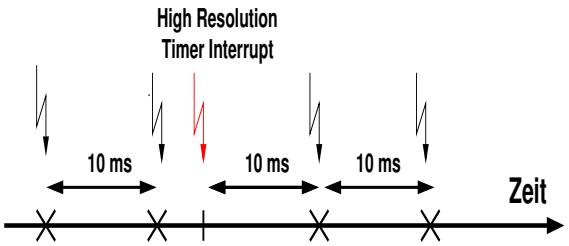
Every time, the timer interrupt arrives at the CPU, the Timer Interrupt Service Routine (ISR) is executed. After that also the SoftIRQ of the Timer Interrupt, historically called 'Timer Bottom Half', is executed.

The 'elapsed time per second' in table 8.1 has been calculated using the following formula:

$$\frac{elapsed\_time}{second} = \frac{NInterrupts}{second} * (t_{ISR} + t_{TimerSoftIRQ}) \tag{8.12}$$

$(t_{ISR}+t_{TimerSoftIRQ}) = (6.9+1.0)microseconds$ - see table 8.1 - is the average time the Timer Interrupt Service Routine and the Timer-SoftIRQ need to be executed once on the Pentium 4, running with a CPU frequency of 2.2 GHz, that has been used for these measurements. So the Linux timing system consumes only about 0.08 percent of the whole CPU time per second on this relatively fast CPU, when the timer interrupt comes only 100 times per second in standard Linux 2.4.

On the other hand in Linux 2.6 the execution of the timer interrupt handler and the timer SoftIRQ need $(t_{ISR}+t_{TimerSoftIRQ}) = (8.3+0.6)microseconds$ every time the timer interrupt comes. With 1000 timer



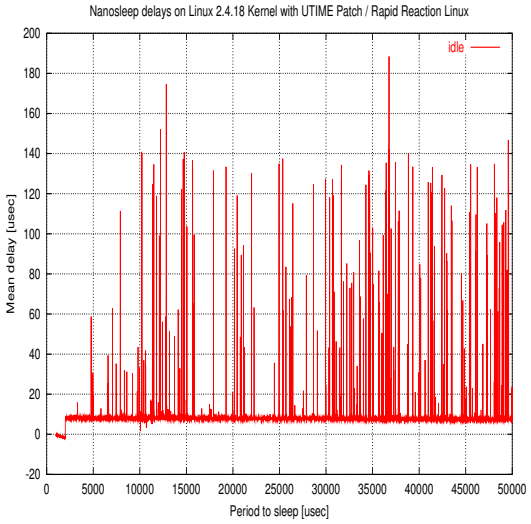




Figure 8.3: Left part: *Linux 2.4 with UTIME, HRT patch or Rapid Reaction Linux Patch: A high resolution timer interrupt is an additional interrupt, fired by the timer interrupt chip at the precise time as required by an application in advance.*
Right part: *Average delay of a periodic soft-realtime task in the modified Linux kernel 2.4.18 with UTIME Patch / Rapid Reaction Linux. Compared to fig.8.1 the UTIME Patch / Rapid Reaction Linux improves the time resolution of standard Linux from 10 msec to the range of about 20 usecs average on standard Intel hardware. The 'mean delay' in microseconds (usec) contains the timing delay and the PDLT. The bigger values are probably caused by the non-preemptive Linux kernel, that causes PDLT values of up to 190 usecs in this measurement.*

interrupts per second this means a CPU consumption of about 9 milliseconds per second, i.e. 0.9 per cent of the whole CPU time.

Of course, the absolute values differ and the same code will execute faster on a processor running at a higher CPU frequency. Also the percentage of the CPU used for the Linux timing system will be less there. On the other hand on a slower processor the percentage of the CPU usage for the timer interrupts will be higher.

Looking at table 8.1 it is clear that a timer interrupt every 100 microseconds would probably lead to a CPU usage of about 9 percent on an Pentium 4, which is not acceptable at all. So using current personal computers another way to increase the timing resolution any further must be chosen.

### 8.1.7 UTIME Patch

As experiments in section 8.1.6 and table 8.1 showed, increasing HZ costs performance and on current hardware a timer interrupt more often than all 1 milliseconds causes too much overhead.

So this section analyses an alternative mechanism to improve the timing resolution of the standard Linux kernel.

#### History of the UTIME Patch

The name UTIME stands for Microsecond (usec) Resolution Timing. This patch has first been developed or ported to Linux on the Intel x86 architecture at the University of Kansas [41], in April 1998 the UTIME stand-alone patch using the Linux kernel version 2.2.13 had been developed. Afterwards the team of Dr. D.Niehaus integrated the UTIME patch into its bigger KURT patch [50], another realtime enhancement for the Linux kernel. In this thesis the UTIME patch has been ported to Linux 2.4 and integrated with a Low Latency patch to form 'Rapid Reaction Linux', cp. section 8.2.

Another project, called High Resolution Timer (HRT) Patch [2], has been built on the experiences of the UTIME patch, too, cp. section 8.2.2.

### Technical realization

The UTIME Patch [41] takes advantage of the fact that soft realtime tasks may have stringent timing requirements on the order of 100 microseconds (usec), but it is very unusual that there is such an requirement every 100 usec. The solution provided by UTIME is to reprogram the timer chip after every timer interrupt to the next foreseeable event. If there is any task, e.g. requiring scheduling in 7 milliseconds (msec), the timer chip is reprogrammed to generate an interrupt at exactly that time. If there is no such timing event the timer chip is reprogrammed to generate the next timer interrupt just in 10 msec. So it´s guaranteed that the interval in between two timer interrupts is never longer than 10 msec, the period of the timer interrupt in standard Linux 2.4.

UTIME extends the global kernel variable `unsigned long volatile jiffies` by a variable called `jiffies_u`, that can be adjusted whenever necessary to the appropriate value in between 0 and 9999 microseconds (usec) to indicate how much time there is until the variable `jiffies` has to be increased again. The Linux time base does not suffer from the jitter of the 8054 timer chip on the order of microseconds, because `jiffies_u` is set according to the current value of the Time Stamp Clock Register (TSC), cp. section 3.7.1. Since this is a clock driven at the processor cycle rate, it is best suited to improve the timing resolution of the standard Linux kernel.

At boot time, the UTIME patch calibrates the TSC clock against the timing interval of the Programmable Interrupt Timer, the PIT, cp. section 3.7.1. That way a factor is determined used to convert the processor clock cycles of the TSC into microseconds.

The patch introduces also the field `signed long usec;` into the kernel timer structure, see section 8.1.4, in order to raise its precision to the range of microseconds (usec).

The UTIME patch provides the following big advantages compared to the timing of standard Linux:

- It allows to nearly eliminate the term *timing_delay* in eq.(8.3). If *timing_delay* is zero, then $delay_{task}^{timed}$ only depends on the PDLT which can be in the best case in the range of only twenty microseconds on modern personal computers, if they are idle. So the UTIME patch makes it possible to execute tasks with timing requirements in the range of 100 microseconds in time.
- The UTIME patch has a very small overhead as there are only additional interrupts generated, when they are really needed to meet soft-realtime constraints.
- Even compared to a timer interrupt, that comes all 1 msec, the UTIME patch can provide a 10 up to 50 times higher timing resolution and a smaller overhead.
- It's assured by the UTIME patch, that a timer interval is never longer than the 10 msec timer interval of standard Linux 2.4. So all applications have at least the same timing resolution.

### Porting the UTIME Patch from Linux 2.2.13 to 2.4

Since there was only a version of UTIME for Linux 2.2.13 available to download [41] the UTIME patch has been ported in this thesis to the Linux 2.4 kernel.

To adapt the UTIME patch to the Linux 2.4 kernel, some changes were necessary, the most important were the following ones:

- Linux 2.2 and Linux 2.4 contain a complex management for time-critical tasks, containing a vector with six lists of events to execute at specified times. The UTIME patch had to be adapted to this structure, since the UTIME 2.2.13 still used the `'old timer list'` mechanism of Linux 2.0 and 2.2. This has been done looking also at the source code of the newer KURT patch versions.
- Linux 2.2 contains a kernel variable `lost_ticks`, which serves to store the `jiffies` for the time in between the timer interrupt occurs and the timer bottom half updates the time basis. This variable has been erased for the x86 platform in Linux 2.4 and replaced by the difference `'jiffies-wall_jiffies'`, containing the new variable `'wall_jiffies'`. The UTIME patch had to be adapted to these changes.

- Many other slight changes were necessary.

The effects of the UTIME Patch on the timing behaviour can be seen in fig.8.3. There it can be seen, the delays caused by the UTIME Patch are only in the range of 20 up to 190 microseconds which is smaller than the *timing_delay* of 1 or 2 milliseconds of Linux 2.6 and much smaller than the *timing_delay* of 10 or 20 milliseconds of Linux 2.4.

## 8.2 Rapid Reaction Linux

### 8.2.1 UTIME and LowLatency patch forming Rapid Reaction Linux

Rapid Reaction Linux combines the 'LowLatency Patch' with the 'UTIME Patch' to be able to serve time-critical tasks in a better way.

In the following it shall be explained, why it has been useful to combine just these 2 kernel patches:

- A good preemptability is a basic feature for every operating system that shall fulfill time-critical tasks. One of the most efficient ways to increase the preemptability of the Linux 2.4 kernel is the LowLatency Patch, as described in section 7.3. Due to the measurements in this thesis, and the study of literature [101, 67] it has been found that the 'LowLatency Patch' reduces many long kernel latencies of the Linux 2.4 kernel best.
- Since the LowLatency Patch is able to reduce many known worst case latencies to the order of tens of milliseconds on current personal computers, it is worth to improve the resolution of Linux timing services beyond the 10 milliseconds provided by the standard Linux 2.4 kernel, too. That´s what the UTIME patch does, without generating much overhead.
- The code changes of the ´LowLatency´ Patch are somehow orthogonal to the changes the UTIME patch makes, because both patches aim at different goals. So the two patches don´t interfere.

There are two further changes that have been made to the kernel code to form 'Rapid Reaction Linux', worth mentioning. They are presented in the two following subsections and have been necessary to combine the advantages of both patches.

#### Improving UTIME kernel timers one-shot behaviour

Since Rapid Reaction Linux incorporates the UTIME Patch, eq.(8.6) on p.123 does not apply to Rapid Reaction Linux and the term *timing_delay* in eq.(8.3) for periodical tasks is reduced to values in the range of some microseconds, not more.

But using the UTIME patch the first `nanosleep(period)` or the `nanosleep()` of a one-shot task often ends in between 0 and 10 milliseconds too soon, because of the outdated system time used. To solve this problem it is necessary to have the Linux time base updated before adding the period, the process wants to sleep, to the current value of `jiffies`. Regarding `nanosleep()` Rapid Reaction Linux calls in kernel code the UTIME function `update_jiffies_u()` to update the system time before starting the sleeping period. `update_jiffies_u()` uses the TSC register of the processor. That way the first jitter of `nanosleep()` is avoided.

A proof that changes show effect can be found in table 8.2: There in line E Rapid Reaction Linux shows a minimum delay of ´0.019 msec´ and a mean delay of ´0.341 msec´, which is much better compared to e.g. the ´-9.8 msec´ of the UTIME Patch in line B of the same table.

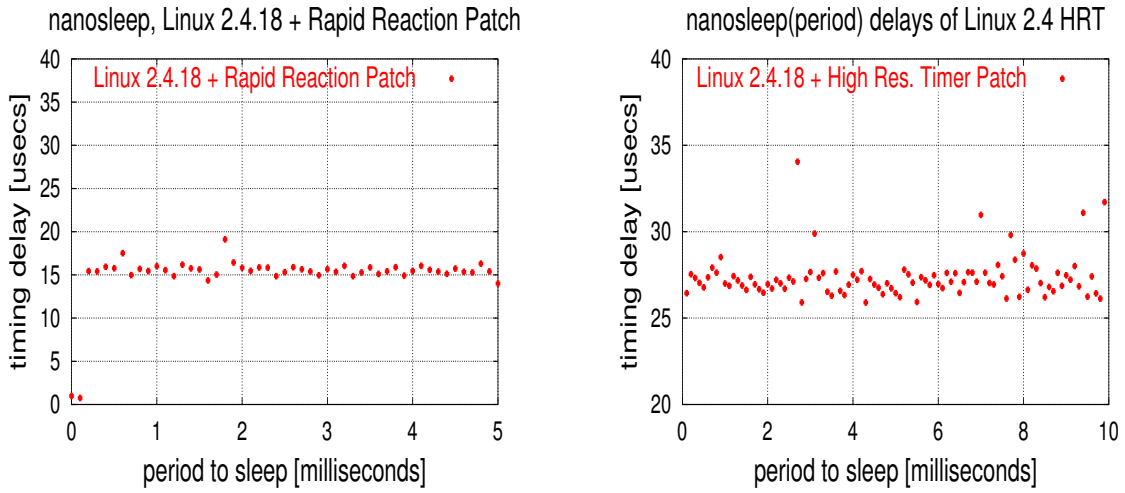



Figure 8.4: Left part: *The measurement program is the same as in fig.8.1, but here executed on Linux 2.4.18 with* **Rapid Reaction Patch**. *The timing_delay with the Rapid Reaction Patch is beyond about 20 microseconds [usec] which is in another order of magnitude compared to the 2 msec of standard Linux 2.6 in the right part of fig.8.1,* Right part: *The same measurement as in fig.8.1 to 8.4 , but here executed on Linux 2.4.18 with* **High Resolution Timer Patch**. *The timing_delay with the HRT-Patch is beyond 35 microseconds [usec], which is also much better than standard Linux*

#### Introduction of soft realtime timers

Since Rapid Reaction Linux can be programmed to fire an interrupt at a certain point in time with the precision of about 20 microseconds on a current Intel compatible PC, it is reasonable that the scheduler is started as soon as possible after the Interrupt service routine (ISR) is done. This is useful since often Interrupt handlers are used to awake a time-critical task as a second reaction to the interrupt, that is working in its own memory context.

To achieve this goal in Rapid Reaction Linux the field '`timer.need_resched`' has been added to the kernel timer structure, cp. section 8.1.4. In Rapid Reaction Linux this field will be set to 1, every time a soft realtime task wants to sleep for a precise time interval. When the affiliated timer interrupt occurs, the variable `current→need_resched` is set to 1 in the ISR to reschedule as soon as possible in order to make potentially use of a Conditional Preemption Point, cp. section 7.3.

If `current→need_resched = 1` is set to 1 in the timer ISR the Linux kernel is able to use all the conditional preemption points, the 'LowLatency Patch' introduced into Linux, to reduce kernel latency times, caused by long term system calls that normally cannot be preempted in Linux 2.4.

The success of this change can be seen in the better results of Rapid Reaction Linux, shown at line E of table 8.2 and 8.3, compared to a combination of the best values of line B, the UTIME Patch, and line D, the LowLatency Patch.

The success of Rapid Reaction Linux can also be seen in the left part of fig.8.4. There Rapid Reaction Linux shows a *timing_delay*, - here the minimum PDLT is even included -, of only 15 to 20 microseconds, which is much smaller than 1 or 2 milliseconds of Linux 2.6 and much smaller than 10 or 20 milliseconds of Linux 2.4, cp. fig.8.1. Furthermore, as fig.8.4 shows, in Rapid Reaction Linux any period that isn't a multiple of 1 ms can be chosen to sleep for without loosing the minimal *timing_delay* of only 15 to 20 microseconds. Fig.8.4 has been measured on an idle computer, so with a minimum PDLT. In section 8.2.4 Rapid Reaction Linux is tested with a higher load leading to a higher PDLT.

### 8.2.2 The High Resolution Timer Kernel Patch

The High Resolution Timer kernel patch (HRT patch) [2] is a further development of the UTIME patch. The basic idea behind the HRT-Kernel Patch is the same as in Rapid Reaction Linux using the UTIME

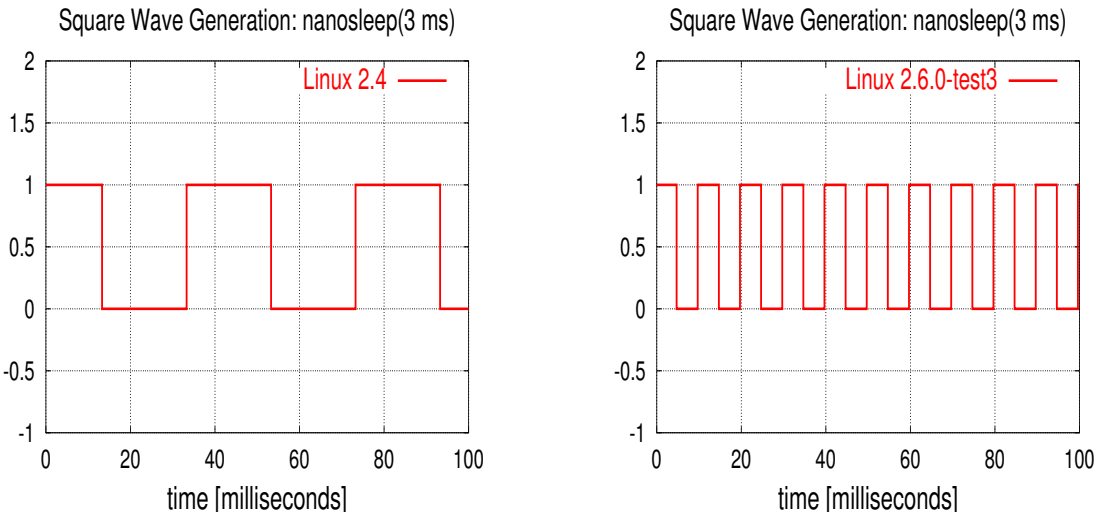



Figure 8.5: Left part: *The process tries to sleep for only 3 milliseconds on a Linux 2.4.18 standard kernel (vanilla). The measurement shows that it is put asleep for 20 milliseconds, as it is predicted by eq.(8.6),* Right part: *The process tries to sleep for only 3 milliseconds on a Linux 2.6.0-test3 kernel. In fact it sleeps for 5 milliseconds, as also predicted by eq.(8.9) and (8.11)*

patch. The chip generating the timer interrupt is reprogrammed to the next foreseeable event, thus allowing a more precise timing, limited by the hardware delay of the used clock.

On the Intel x86 platform the HRT patch offers the possibility to use the clock of the power management, called the ACPI-pm Timer, instead of the Time Stamp Register (TSC) of the processor to calculate the current time with a resolution of microseconds between two timer interrupts, if needed, cp. section 3.7.1.

The measurements in fig.8.4 and fig.8.6 show that the High Resolution Timer Patch has a similar performance as Rapid Reaction Linux, as could be expected since they both follow a similar concept.

### POSIX Timer

Via a POSIX interface the High Resolution Kernel Timer project makes all interesting hardware clocks of the platform available to the software developer using the different POSIX clock types and timer API functions as specified in POSIX 1003.1b, section 14, earlier referred to as POSIX.4.

So with the help of a library, which can be downloaded for Linux 2.4 [2], the following functions are implemented: `clock_settime, clock_gettime, clock_getres, timer_create, timer_delete, timer_settime, timer_gettime, timer_getoverrun`.

The first three functions read the time of a clock, its resolution and allow the superuser to set the time of the clock. The other functions allow to create timers per process, that send a signal that can be handled by a signal handler when they expire.

This POSIX Timer interface has been already put into the Linux 2.6 kernel to make Linux 2.6 more POSIX compliant. But the resolution of these timers and clocks is not higher than $\frac{1}{HZ} = 1msec$ in standard Linux 2.6.

Therefore, to get a higher timing resolution, Linux 2.6 has still to be patched with the now smaller HRT-Patch for Linux 2.6 [2].

The Linux kernel 2.6 in general is moving towards a higher POSIX conformity [58, 103] compared to Linux 2.4, as mentioned in section 5.2.

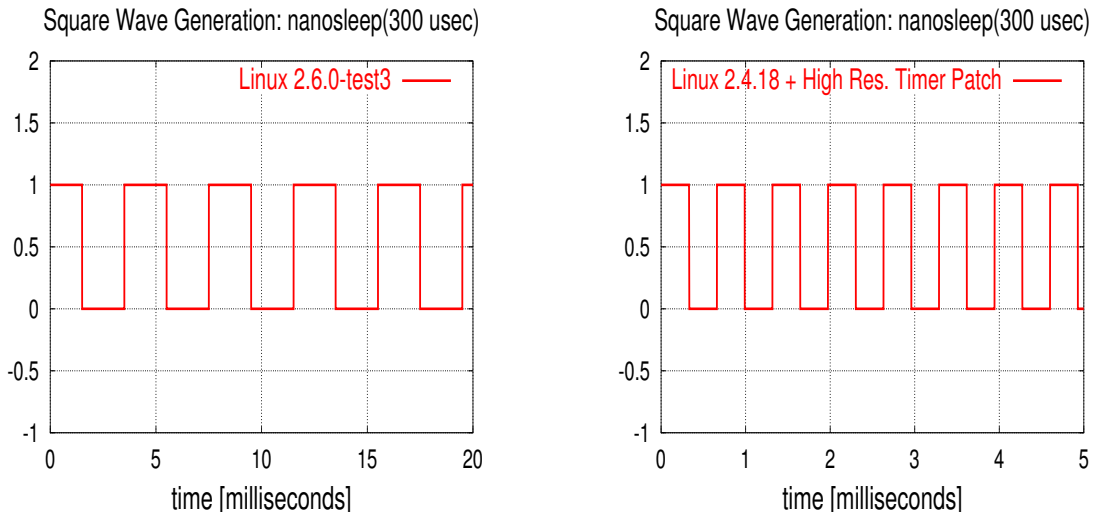



Figure 8.6: Left part: *The process tries to sleep for 300 microseconds on a* **Linux 2.6.0-test3 kernel**, *which isn't possible. According to eq.(8.9) the process will sleep for 2 milliseconds and that's also the value that is measured here.* Right part: *The process tries to sleep for only 300 microseconds on a* **Linux 2.4.18 kernel patched with the High Resolution Timer Patch**. *The measurement show that the tasks sleeps for about 330 usecs instead of 300 usec each time, which is fairly good for such a small period, only a small error of 10 percent.*

### 8.2.3 A software oscilloscope

Another way to visualize the behaviour and frontiers of the timing system of the different Linux kernels and patches is to make the test program introduced in section 8.1.5 writing a `0` or a `1` alternately together with a time stamp of the TSC into memory and later into a file. Later on, this data can be visualized like data on an oscilloscope, so it is called a 'software oscilloscope'.

The measurements shown in fig.8.5 to 8.6 prove that eq.(8.6), p.123, and (8.9), p.124, describe the behaviour of the different Linux kernel versions in the right way.

### 8.2.4 Comparing the preemptability of standard Linux and Rapid Reaction Linux

As already stated in eq.(8.3) in addition to the *timing_delay* a task is always delayed by the $PDLT$ latency, needed to preempt the currently running process. In the case of an idle system the $PDLT$ is very small, only about 10 or 20 microseconds (usec) on a modern Pentium 4, as specified in appendix A.2. This is the time that is always necessary to preempt a process.

But since the Linux kernel 2.4 is not fully preemptive, this latency can also be high in rare cases. Fig.8.7 and 8.8 show results of the same test program, presented in section 8.1.5. In these figures delays of the program are shown when the sleeping period is varied.

The results can be compared to the left part of fig.8.1, where the results of the same test program on a nearly idle PC are shown. On the contrary in fig.8.7 and 8.8 the measurements are done on a PC with high disk load without DMA.

The load causing such long latencies has been generated by the following program, that writes a lot of data to the IDE hard disk without direct memory access (DMA), so the CPU has to handle all data on its own. This method is called CPU I/O.

```
#!/bin/sh
hdparm -d0 /dev/hda1
mount /dev/hda1 /u01
```

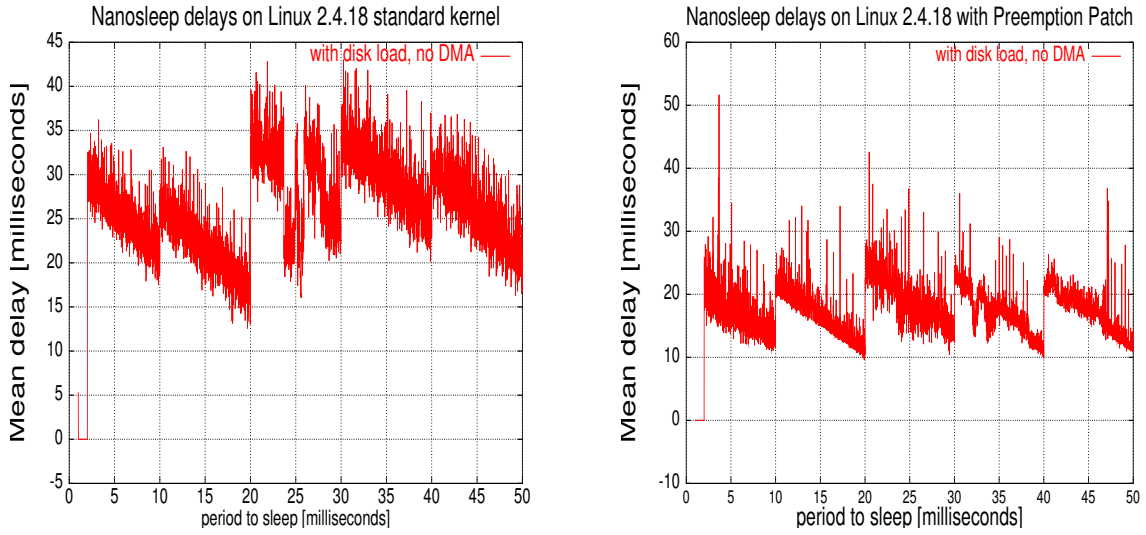



Figure 8.7: Left part: *Average* $delay_{task}^{timed}$ *of the test program on standard Linux 2.4.18 under high disk load without DMA. Compared to the left part of fig.8.1 the delays are higher here.*, Right part: *The same as on the left part, but this time the kernel has been patched with the Preemption Patch which reduces the occurring latencies.*

```
while(true) do
        dd if=/dev/zero of=/u01/disk.dummy bs=1024 count=4096 #> /dev/null
        sync
        sync
done
```

The command `hdparm -d0 /dev/hda1` stops the DMA controller of the IDE hard disk.

On the left part of fig.8.7 standard Linux is measured. The load script running in parallel causes considerable high delays $delay_{task}^{timed}$ of the measurement program under high load compared to the left part of figure 8.1, which shows an idle standard Linux 2.4.18. These latencies occur, because the load script uses long system calls, that are not preemptible in standard Linux, as shown in case A2 of fig.4.2, p.61. Nevertheless, the leaps in both figures occur at the same locations every 10 msec on the x-axis, since they are caused by the Linux 2.4 timing behaviour described by eq.(8.6).

In fig.8.8 the same program with the same load is executed, but on a Linux 2.4.18 with Rapid Reaction Patch. Eq.(8.6) doesn't apply for Rapid Reaction Linux, because in Rapid Reaction Linux *timing_delay* is constant and it's not bigger than some microseconds, so the right part does show neither a *timing_delay* nor the characteristic bounds every 10 msec.

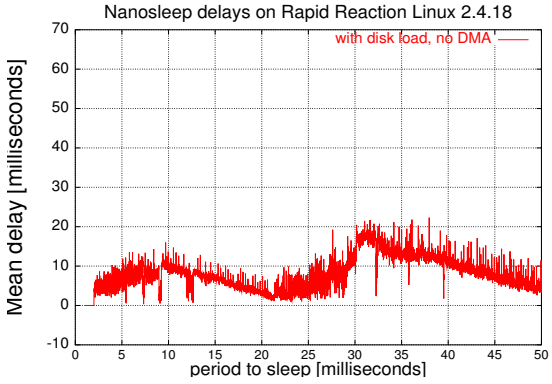



Figure 8.8: *Average* $delay_{task}^{timed}$ *of the test program on Linux 2.4.18 with Rapid Reaction Patch under high disk load*

The $delay_{task}^{timed}$ one sees in fig.8.8 is caused nearly only by the PDLT. The Preemption Points introduced by the LowLatency patch into Rapid Reaction Linux shorten the execution time of the long system calls, where preemption can't take place as shown in line B of fig.4.2, p.61.

Therefore one can see from the left part of fig.8.7, that the load on the system leads to considerable higher latencies in Linux 2.4.

Fig.8.8 shows that to combine a higher timing resolution with a better preemptability of the Linux kernel leads to the best result, i.e. Rapid Reaction Linux shows the smallest $delay_{task}^{timed}$ for the measurement program under high load.

A further improvement can only be reached by increasing the preemptability of the Linux kernel, cp. section 7.1.

### 8.2.5 Reasonable periods of periodical tasks

If a soft realtime tasks has to run periodically on a system, the possible period in standard Linux is restricted downwards by the following formula, if the chosen period of the task shall make sense:

$$period_{min} = factor * delay_{task}^{timed} + execution\_time = \\ factor * (timing\_delay_{max} + PDLT_{max}) + execution\_time \tag{8.13}$$

The term *execution_time* means the time the periodical soft realtime task needs to execute in one period. It can be measured, but depends of course on the task, the compiler and also on the hardware it runs on.

Since a periodical task is awakened by a timer interrupt at the previewed times in order to execute, occurring latencies can be split into the latency produced by the timer management of the operating system *timing_delay* and the normal *PDLT* that follows the timer interrupt until the periodic soft realtime task is running, cp. eq.(8.1).

Equation (8.13) shows the minimum period, a soft realtime process should be scheduled on a given computer system. In the following a reasonable size of the *factor* in equation (8.13) shall be found. Of course *factor* must be always bigger than 1, as the period must be bigger than the execution time and the average latency. Longer latencies occur at random.

Assumed it's a short task with an *execution_time* short compared to average or maximum latencies, expressed by the term $delay_{task}^{timed}$. Assumed furthermore, that this *execution_time* can be considered as to be constant with an error that is small compared to the value of *execution_time*.

In this case if $period_{min}$ is chosen to be in the range of the maximum latencies, the time in between two runs of the program can be as long as two periods, so the frequency of such a time-critical task wouldn't be stable at all. This wouldn't be a usable system.

So a rule of thumb is to choose a factor of 3 in equation (8.13), to choose the minimum period at least 3 times of the longest jitter possible, i.e. the $delay_{task}^{timed}$. But 33 percent of the period is still a big error. If the value of *factor* is chosen to equal 10, then the latency is at maximum 10 percent of the minimum period possible. This will be an acceptable error for many time-critical applications. Thus, the value of the factor in equation (8.13) has to be chosen according to the needs and tolerance limits of the time-critical application.

### 8.2.6 Testing Linux and Linux realtime enhancements

In this section the behaviour of the standard Linux kernel 2.4 and the different kernel patches is shown by presenting and interpreting measurements results. Again the measurement program, first presented in section 8.1.5, has been used to obtain the measurement data.

| | *delay* in msec<br><br>**load: disk stress** | Min *delay* msec | Mean *delay* msec | Max *delay* msec |
|---|---|---|---|---|
| A | Linux 2.2.13 | 0.2 | 11.2 | 119.6 |
| B | A+UTIME Patch | -9.8 | 1.7 | 125.2 |
| C | Linux 2.4.0-test6 | 0.2 | 10.9 | 103.5 |
| D | C+LowLatency Patch C4 | 3.1 | 9.9 | 11.9 |
| E | C + Rapid Reaction Patch | 0.019 | 0.341 | 5.2 |
| F | C+Preemption Patch 1.5 | 0.8 | 12.2 | 119.9 |

Table 8.2: *Results of a task calling* `nanosleep()` *for a period of 50 msec periodically on an AMD K6 with 400 MHz. The columns {Min, Max, Mean} mean the delay, i.e. the time interval the kernel variant caused the soft-Realtime Process to sleep longer than the period of 50 msec (milliseconds). Ideally delay should be 0 msec for {Min, Max, Mean}. Shown are the minimum and maximum times measured {Min delay , Max delay } and the mean delay measured {Mean delay }. The test program ran over 1000 periods in init 5. The IDE hard disks were not tuned using the program "hdparm". During all these measurements a program performed hard disk stress [101] in the background.*

The standard lag on `nanosleep()` in the standard Linux 2.4 kernel, even for a soft realtime task, is 10 msec. Furthermore, a worst case latency on the order of hundreds of milliseconds on current hardware can occur, as show the measurements in lines A,B,C of Table 8.2.

The periodical test program in section 8.1.5 reveals `3 problems` of Standard Linux 2.4:

1. Linux 2.4 has a standard lag on `nanosleep()` of 10 msec, i.e. 1 jiffie, when `nanosleep()` is called periodically, i.e. instead of a period of 50 msec, Linux sleeps about 60 msec in the sample above. This can be seen from the measurements, regarding column ´mean delay´ in Table 8.3, best to be seen in lines A,C,D and F because of short latencies, and from the variable ´rawdata´ in the Linux test program given in [101].

2. At the first time `nanosleep(50 msec)` is executed, the lag is often much shorter than the 'mean delay' of 10 milliseconds (msec), because the start of the first period is not synchronized to the timer interrupt, reawakening the sleeping process. This problem can be observed at the column 'min delay' in the lines A,B,C,D and F in Table 8.3. The test program given in [101] has been changed to evaluate also this first measured value.

3. When running a `dd` job and writing the buffers to the hard disk using `sync` in background, causing heavy hard disk activity - as proposed in [101] -, Linux shows long worst case latencies on the order

| | *delay* in msec<br><br>**ping load** | Min *delay* msec | Mean *delay* msec | Max *delay* msec |
|---|---|---|---|---|
| A | Linux 2.2.13 | 1.7 | 10.0 | 10.1 |
| B | Linux 2.2.13 + UTIME Patch [41] | -0.7 | 0.012 | 0.3 |
| C | Linux 2.4.0-test6 | 4.6 | 10.0 | 10.2 |
| D | Linux 2.4.0-test6+LowLatency [65] Patch C4 | 8.4 | 10.0 | 10.1 |
| E | Linux 2.4.0-test6+ Rapid Reaction Patch | 0.020 | 0.031 | 0.141 |

Table 8.3: *Results of the same measurement program as used in Table 8.2. The only difference to Table 8.2 is that this time the background load consists of a program generating net stress by pinging another host [101] during the measurements. As we can see from the fact that the max delays do not differ much from the mean delays, the "ping load" is no serious load compared to the "disk load" of Table 8.2*

| *delay* in msec E: **Rapid Reaction Linux, no load** | period<br>msec | min *delay*<br>msec | mean *delay*<br>msec | max *delay*<br>msec |
|---|---|---|---|---|
| Busy Wait → | 2.000 | -0.001 | 0.00086 | 0.080 |
| | 3.250 | 0.018 | 0.020 | 0.110 |
| | 4.000 | 0.017 | 0.020 | 0.126 |
| | 13.350 | 0.017 | 0.031 | 0.208 |
| | 27.500 | 0.009 | 0.019 | 0.112 |
| | 38750 | 0.010 | 0.019 | 0.156 |
| | 50000 | 0.013 | 0.019 | 0.078 |
| | 53.350 | 0.010 | 0.025 | 0.393 |

Table 8.4: *This table shows other periods of a periodical task, that are possible in Rapid Reaction Linux, here based on the kernel Linux 2.4.0-test6, if there is no heavy disk load on the system. The test program ran on an AMD K6 400 MHz in runlevel init 5 for 1000 periods for each measurement, without any additional load program. For SCHED_FIFO tasks with a period of 2 msec and beyond, Standard Linux and Rapid Reaction Linux perform a Busy Wait, which produces highest accuracy, but no other process can be executed in these 2 milliseconds.*

of hundreds of milliseconds, see lines A,B,C and F of Table 8.2.

Rapid Reaction Linux improves all 3 problems, see line E of Tables 8.2 and 8.3.

In Table 8.2 and 8.3, a periodical task with a period of 50 msec shall be executed. Instead of the desired period of 50 msec, all standard Linux kernels execute this task normally with an delay of 10 msec, i.e. all 60 msec. Only the original UTIME Patch [41] for Linux 2.2.13 and "Rapid Reaction Linux" normally execute the periodical task with the desired period, i.e a mean delay of nearly 0 milliseconds, see Table 8.3. The 31 microseconds mean *delay* in line E of Table 8.3 might be partly due to the Linux 2.4 scheduler, that selects the SCHED_FIFO process among all other process. Except for Rapid Reaction Linux the first period after starting the measurement normally produces the minimum value, that can differ up to 10 msec from the mean value, because the start of the measurements and the first `nanosleep(50 msec)` is not synchronized to the timer interrupt. The "ping" load in Table 8.3 isn't a heavy load for the system, so the difference in between the 'maximum *delay*' and the 'mean *delay*' in Table 8.3 is only in between 100 and 300 microseconds. But when - as shown in Table 8.2 - a heavy disk load without DMA is executed on the system as background load, for the most Linux versions, the maximum delay measured differs from the mean delay about 100 msec. This is due to latencies caused by non-interruptible system calls invoked by the disk load program, probably to write the buffers to disk (sync). Only the "LowLatency Patch", Line D, and Rapid Reaction Linux, Line E, that incorporates it, can reduce these latencies to the order of 5 msec on the AMD K6 PC, as specified in appendix A.1. As "Rapid Reaction Linux" combines the UTIME Patch, ported to 2.4, with the "LowLatency Patch" and because it applies some changes described in section 8.2.1 and 8.2.1, "Rapid Reaction Linux" can provide the lowest (maximum - minimum) *delay* values with a very good mean *delay* near 0 msec in both Tables 8.2 and 8.3.

In standard Linux 2.4 the period of a periodical task has to be a multiple of 10 milliseconds. In Table 8.3 one sees, that standard Linux 2.4 adds 10 msec by default to the period. In Table 8.5 it can be seen - if the period is not a multiple of 10 msec - Linux 2.4 rounds it up to the next 10 msec boundary. $delay_{task}^{timed}$ normally is a fixed value in between 10 up to 20 milliseconds, when there is no heavy load on the system. In standard Linux 2.4 it is not possible to run a task with a period less than 20 msec, if you don't want to perform 'busy waits' (see Table 8.5). A 'busy wait' is performed in standard Linux 2.4 and in Rapid Reaction Linux, if the sleeping period is 2 milliseconds and beyond and if the process is scheduled by a soft-realtime policy, i.e. by the scheduling policies SCHED_FIFO or SCHED_RR.

In Rapid Reaction Linux the period hasn't to be a multiple of 10 msec, although the period of the Linux timer interrupt remained unchanged at 10 milliseconds. Table 8.4 shows that in Rapid Reaction Linux it is possible to choose other periods. It is possible to schedule tasks with a period of e.g. 3.250 msec, 4 msec, 13.350 msec, ... and the standard *delay* is only in between 20 to 30 microseconds. The maximum *delay* measured was about 400 microseconds, while there was no heavy load on the system. On the AMD K6, 400 MHz, only a X as graphical user interface with some terminals was running.

| *delay* in msec C: **Linux 2.4.0-test6 no load** | period msec | min *delay* msec | mean *delay* msec | max *delay* msec |
|---|---|---|---|---|
| Busy Wait → | 2.000 | -0.005 | -0.0046 | 0.003 |
| | 3.250 | 11.3 | 16.7 | 16.8 |
| | 4.000 | 11.6 | 16.0 | 16.0 |
| | 13.350 | 15.5 | 16.6 | 16.7 |
| | 27.500 | 6.7 | 12.5 | 16.7 |
| | 38.750 | 7.2 | 11.2 | 11.3 |
| | 50.000 | 6.4 | 10.0 | 10.05 |
| | 53.350 | 8.9 | 16.6 | 16.7 |

Table 8.5: *This table shows other periods of a periodical task, on the standard Linux 2.4.0-test6 kernel, if there is no heavy disk load on the system. In standard Linux it is not possible to run a task with a period less than 20 msec, except you use Busy Waiting for SCHED_FIFO processes that only have to wait for 2 msec or less. The standard lag of standard Linux is 10 msec. Tasks with a period, thats not a multiple of 10 msec are delayed to the next bigger period that is a multiple of 10 msec, plus 10 msec. F.ex a task with a period of 50 msec is scheduled with a mean delay of 10 msec, so it is in fact scheduled with an period of 60 msec. A task with a period of 53.350 has a mean delay of 16.6, so its in fact scheduled with a period of about 70 msec. So, in standard Linux it is not possible to run a task with a period less than 20 msec. The test program ran on an AMD K6, 400 MHz, in runlevel init 5 for 1000 periods for each measurement,without any additional load program running.*

Table 8.6 shows once more the difference between standard Linux and Rapid Reaction Linux, but this time with a heavy disk load, produced by the program `dd` and `sync` operations, that write the buffers to the IDE hard disk [101]. standard Linux again shows its standard delay of 10 msec as a period of 50 msec is a multiple of 10 msec. The heavy disk load produces some very long latencies up to 110 msec in Standard Linux.
Rapid Reaction Linux can provide a mean delay of only 300 microseconds to the periodical task. The maximum delay measured here was about 3.3 milliseconds The minimum delay is again at 20 microseconds, as in the case without load for Rapid Reaction Linux (see Table 8.4).
The mean delay increases slightly for standard Linux from about 10 to 10.9 msec and for Rapid Reaction Linux from near 0 to 0.3 msec, because the increase of the maximum delays increases the mean, too.

| *delay* in msec **load: disk stress** | period msec | min *delay* msec | mean *delay* msec | max *delay* msec |
|---|---|---|---|---|
| C: Linux 2.4.0-test6 | 50.000 | 0.242 | 10.9 | 110 |
| E: Rapid Reaction Linux | 50.000 | 0.018 | 0.3 | 2.9 |
| E: Rapid Reaction Linux | 53.530 | 0.020 | 0.3 | 3.3 |

Table 8.6: *This table compares Rapid Reaction Linux to standard Linux once more: In Rapid Reaction Linux it is due to the UTIME patch possible to choose a frequency which isn't a whole number. In standard Linux this is not possible. Due to the LowLatency Patch latencies are greatly reduced in Rapid Reaction Linux. All measurements in this table have been made while Phil Whilshire's program generated heavy disk load. This explains why the max delay is higher as in Table 8.4, where there is no load on the system. At frequencies beyond 50 msec it has not been possible to execute a sync of our IDE-hard disk in between.*

### 8.2.7 Availability of kernel patches and source code developed in this thesis

The kernel patch for Rapid Reaction Linux and other patches and tools developed in this thesis have been licensed under the terms of *GPL2* [15]. They can be downloaded from the web page of the Institute of Information Technology Systems of the University of the Federal Armed Forces Munich, Germany [32].

### 8.2.8 Discussion

#### Rapid Reaction Linux 2.4 compared to the standard kernel Linux 2.4

The measurements presented in this thesis and the measurements of many others showed that the 'LowLatency Patch' [65] can reduce many long latencies very efficiently. Having a Linux kernel with reduced latencies it is even more interesting to have also an accurate time basis.

Therefore in this thesis the UTIME Patch [41] has been selected, ported from Linux 2.2 to Linux 2.4 and combined with the "LowLatency Patch".

Furthermore, the accuracy of the UTIME timers has been improved in this thesis and 'soft realtime timers' have been introduced. The resulting Linux kernel patch has been called 'Rapid Reaction Linux'. In the measurements it shows a very good timing accuracy when executing periodical tasks or one-shot tasks, too.

Standard Linux 2.4 shows a standard lag of 10 msec for periodical tasks, Rapid Reaction Linux does not. In standard Linux the first period of the task is often shorter than the following periods, in Rapid Reaction Linux the first period has the correct length like all the following periods. Standard Linux 2.4 can't schedule periods in between 2 and 20 msec, Rapid Reaction Linux can.

Standard Linux schedules tasks with periods which are not a multiple of 10 msec, - the period of the timer interrupt -, with a delay in between 10 and 20 msec - not including possible latencies - to round up their period to a 10 msec boundary. Rapid Reaction Linux can schedule these periods - without rounding their period - , with a mean delay on the order of tens of microseconds. So Rapid Reaction Linux is best suited for periodical tasks and/or for waiting only once an desired amount of time.

A guarantee about the value of the longest latency in the Linux kernel - with or without the kernel patch of Rapid Reaction Linux - can't be given at the present stage of development - in our opinion -, because nobody can test all paths of the Linux kernel code and their mutual interferences.

Because of the 'LowLatency patch' Rapid Reaction Linux has got a higher preemptability, so the higher timing accuracy is disturbed much less by latencies induced by Linux system calls or kernel threads. This is an important reason for the higher timing accuracy to make sense.

#### The situation in Linux 2.6

The first kernel version of Linux 2.6 has been released in January 2004, by Linus Torvalds. Linus Torvalds and the core developer had decided to incorporate many of the Preemption Points of the 'LowLatency Patches' into the Linux 2.6 kernel. That's what had been done in Rapid Reaction Linux before, too [22].

Furthermore, the Linux 2.6 kernel can be configured to incorporate also the Preemptible Kernel patch at compile time. Using this configuration the standard Linux 2.6 kernel has a much better preemptability, much better than standard Linux 2.4 ever had.

So when porting Rapid Reaction Linux to Linux 2.6, only the port of the UTIME kernel patch and the improvements to combine LowLatency and UTIME patch is necessary. Nevertheless, meanwhile the High Resolution Timer (HRT) Patch follows the same concept and also ported the UTIME patch to Linux 2.6 [2]. The HRT patch still is no part of the early versions of the standard kernel 2.6. On the same time the HRT patch also implements the POSIX timer API. So porting the Rapid Reaction Patch to Linux 2.6 would be like doing the same work twice and therefore it is not very useful.

So there have been others following the same concepts, but at least this thesis has proposed and realized the right concepts at the right time and these ideas have been presented and spread in papers, supported by downloadable code [31, 32], on Linux and Realtime conferences all over the world, also in the US [22], where a lot of Linux development is done [23, 27, 24, 25, 26, 89, 28, 60]. Also there have been contacts to science institutions and industry who contacted us asking for code and advice. The concepts of a better preemptability using mutexes for the Linux 2.6 kernel and the priority inheritance protocol have been asked by several kernel developers meanwhile [19]. The development of future stable Linux kernel versions is still open.

## 8.3 Examining performance using the Rhealstone Benchmark

To test whether and in which way the patches affect the performance of the Linux kernel the Rhealstone Benchmark - introduced in section 3.6.2 - has been executed on different Linux versions and kernel patches.

| | Linux kernel version & patches | Preemption Time [usec] | Intertask Message Passing Time [usec] | Semaphore Shuffle Time [usec] | Interrupt Response Time [usec] |
|---|---|---|---|---|---|
| $\mathcal{A}$ | Linux 2.2.13 | 1.5 | 3.1 | 5.6 | 3.5 |
| $\mathcal{B}$ | $\mathcal{A}$+UTIME Patch [41] | 1.6 | 3.4 | 5.2 | 3.5 |
| $\mathcal{C}$ | Linux 2.4.0-test6 | 1.1 | 3.4 | 5.4 | 3.7 |
| $\mathcal{D}$ | Linux 2.4.2 | 1.0 | 3.6 | 5.4 | 3.7 |
| $\mathcal{E}$ | Linux 2.4.9 | 1.1 | 3.4 | 5.3 | 3.5 |
| $\mathcal{F}$ | $\mathcal{C}$ + I.Molnar [65] LowLatency E2 Patch | 1.2 | 3.4 | 5.6 | 3.7 |
| $\mathcal{G}$ | $\mathcal{C}$ + Rapid Reaction Linux Patch [22] | 1.2 | 3.6 | 5.8 | 3.6 |
| $\mathcal{H}$ | $\mathcal{D}$ + A.Morton [69] LowLatency Patch | 1.0 | 3.9 | 5.3 | 3.8 |
| $\mathcal{I}$ | $\mathcal{C}$ + Preemption Patch 1.5 [67] | 1.5 | 5.1 | 9.0 | 4.7 |
| $\mathcal{J}$ | $\mathcal{D}$ + Preemption Patch 6 [67] | 3.5 | 5.4 | 11.6 | 5.2 |
| $\mathcal{K}$ | $\mathcal{J}$ + A.Morton [69] LowLatency Patch | 5.4 | 5.5 | 12.5 | 5.1 |

Table 8.7: *Results of some of the RHEALSTONE Benchmark programs, measured on an AMD K6, 400 MHz processor on different versions of the Linux kernel, measured in the runlevel 'init 1'. All results are shown in microseconds.*

### The measurement set

Several programs of the Rhealstone Benchmark have been ported in this thesis from other Unix implementations to Linux [43, 42, 7, 36]. The measurement results of these programs of the Rhealstone Benchmark are shown in table 8.7. All times shown there are in microseconds.

Since the underlying hardware influences the measurement results, all measurements in table 8.7 have been performed on the same system, on an AMD K6 with a processor frequency of 400 MHz, as described in appendix A. Every measurement, except the IRT measurement, has been repeated for 7 million times, thereafter the average values of the measured times have been calculated. All measurements have been executed at the runlevel `´init 1´` according to the Unix System V initialization levels. i.e. in single user mode. All benchmark programs have been compiled with the compiler option -O2 of the gcc compiler to optimize them. All results in table 8.7 are shown in microseconds.

### Results of the Rhealstone Benchmark Measurements

Of course, as stated in section 3.6.1 every benchmark program can only measure a few often used execution paths in the kernel.

- The fine grained Rhealstone benchmark programs don´t show significant different results on the different versions of the Linux 2.4 kernel. The Preemption Time shows smaller values for all Linux 2.4 kernels compared to the 2.2 kernels. This may be due to the fact, that the Linux scheduler efficiency for SCHED_FIFO processes has been improved in Linux 2.4, as measured in section 6.1, fig.6.2, p.87. Also the ´LowLatency Patches´ by Ingo Molnar [65] and Andrew Morton [69] - introducing mostly Preemption Points - don´t show an impact on the performance of the benchmark programs.

- A Linux kernel with one of MontaVista´s Preemption Patches [67] needs in all benchmark programs longer execution times than the standard kernel. This indicates that this method of making the Linux kernel preemptible is not free of performance costs.

  The reason for this is that the Preemptible Kernel enables additional parts of the multiprocessor code on single processors, which is a little bit more code to execute than the normal Linux single processor has to do. As can be seen from the comparison of fig.7.6 on p.107 to fig.7.2 on p.101, at every task transition the Preemptible Kernel has to check the Preemption Lock Counter, to test for signals, etc. in order to increase the responsiveness of the Linux kernel. While this makes the Linux kernel more real-time compliant it costs a little performance, which is in fact affordable regarding the performance enhancement processor developers have achieved in the last decades.

- In table 8.7 one can see, that 'Rapid Reaction Linux' has an overhead of about 0.05 to 0.1 microseconds at the Preemption Time, but the value is still smaller than the values of the Linux 2.2 kernels. For the Intertask Message Passing Time it shows up to 0.2 microseconds of overhead. In table 8.7 the 'semaphore shuffle time' of Rapid Reaction Linux is nearly the same as for Linux 2.2.13, and better than Linux 2.4.0-test6. Since Rapid Reaction Linux in line $\mathcal{G}$ is based on Linux 2.4.0-test6, one can say it decreases the Semaphore Shuffle Time, although its value is higher than the one of Linux 2.4.9. Although the values of Rapid Reaction Linux tend to be a little bit higher than in Linux 2.4, - they can be compared to those of Linux 2.2 and there is no reason to fear a major performance loss in 'Rapid Reaction Linux'. Once again one can conclude from these measurements that for the higher timing precision Rapid Reaction Linux provides a little more performance is consumed by the operating system itself since additional code has to be executed.

  Nevertheless, using Rapid Reaction Linux or the High Resolution Timer Patch [2] is much more efficient and less performance consuming than the solution of increasing the standard Linux timer frequency by 10, as has been chosen to done in Linux 2.6. This can be seen from table 8.1, p.125, in section 8.1.6.

- The results of the Interrupt Response Time (IRT) shown in table 8.7 are of the same order of magnitude as the IRT in the figures 7.3 and 7.6 is. So these measurements are consistent with those on p.102 and p.107.

# Chapter 9

# Summary

## 9.1 Conclusion

To an increasing degree users want their personal computers to accomplish time-critical tasks in addition to normal tasks that are not prioritized in any way.

A good example is the - so called - home sector, i.e. private households, where most people run Windows or - more seldom - Linux as standard operating systems.

Nowadays, many people buy music files, e.g. MP3 files, in the Internet, save it on the hard disk of their computer and replay them using their sound card. Furthermore, today some households use TV-cards receiving the TV-signal from a satellite receiver, converting it, so that it can be displayed by a standard operating system on the computer screen. Many others use their personal computer as a DVD player to watch movies at home or to play video games with a high graphical resolution and many frames per second.

Such audio and video tasks are demanding applications: Video streams have a lot of data throughput, while audio tasks have more stringent timing requirements, as the human ear detects already a break of 40 milliseconds as a snap.

Both application classes - video and audio - have soft-realtime requirements. So, if the operating system fails a deadline the user will tolerate, but of course, if the Quality of Service, the operating system can provide, is not sufficient, the user will stop the application and might look for another application, perhaps running on another operating system to fulfill his needs.

Even in embedded devices an operating system like Linux or - Java running on top of a very small operating system - are used more often nowadays than some years before, as they provide a broad functionality, a huge amount of available software, sometimes even free of charge. Additionally, more and more software developers possess knowledge about these operating systems. This makes it easier and cheaper for companies to use them. Formerly, proprietary operating systems were needed to implement devices like firewalls, routers, DVD players and mobile phones, but nowadays such systems can be based on the Linux kernel.

For instance, the big American enterprise 'Motorola' has announced in July 2005, that more than half of the new Motorola mobile phone models in the upper and medium price range will run a Linux kernel using the MontaVista Inc. distribution [21]. A GSM mobile phone - when phoning - receives or sends all 4,6 milliseconds a message for a period of 0,577 milliseconds using a Time Division Multiplex.

Apart from that, Open Source operating systems like Linux have the advantage, that software developers at specialized Linux or at big computer enterprises, at universities and even private persons implement new and better concepts into the kernel or port Linux to new processor generations or write drivers for new peripheral components.

That way, small companies get rid of these costs, they will have to invest on their own, if they use or even maintain a proprietary operating system with only a few customers.

Therefore, Linux partly superseded proprietary operating systems in recent years in the embedded system market. Since Embedded systems often run also time-critical tasks, the abilities of Linux in this field play a decisive role, although Linux - and standard Java as well - can't fulfill hard realtime requirements.

This thesis analyzes the Linux kernel considering it as an example, how a standard operating system can be changed in order to serve time-critical tasks in a better way, i.e. to accomplish such tasks timely.

All efforts and changes to make the Linux kernel more realtime-compliant have been done taking into account two reasonable constraints:

- At first, the interface provided by the operating system for the applications shouldn't be changed.
- At second, the performance of the operating system shouldn't decrease in a way, perceivable to the user.

These two requirements are important, as users wouldn't accept changes to the Linux kernel, if normal applications didn't run on the system any more or only applications using a special API could benefit from the improvements.

Linux - as an Open Source operating system - is suited best for the purpose of this thesis as the source code can be reviewed and also changed and improved, as its license - the GNU Public License of the Free Software Foundation - permits and supports such changes [15].

This thesis has been done from the end of 1999 until the end of 2006, during this time the Linux kernels 2.2 up to 2.6 have been used.

The main reason for latencies of the order of magnitude of ten up to hundreds of milliseconds on Intel compatible personal computers is the non-preemptible Linux kernel of the stable Linux versions 2.0, 2.2, 2.4 and 2.6 on single processor systems.

The Linux kernel had been implemented this way, since synchronization problems like race conditions couldn't occur in a non-preemptible kernel. Later on, when Linux had become more common, software developers started to optimize the Linux kernel to serve time-critical tasks in a better way.

Two different ways to increase the preemptability of the Linux kernel have been carried out:

The first is to introduce so called 'Preemption Points' into the Linux kernel. At such Preemption Points the scheduler can be called without risking a data inconsistency and thus a time-critical task can acquire the processor. These Preemption Points could be added to the kernel code by patching the source code of the Linux kernel with the so called ´LowLatency´ patches, designed by Ingo Molnar and later on by Andrew Morton [65, 69].

The second way is to make the Linux kernel preemptible in general except for critical sections. Such critical sections have originally been introduced into the Linux kernel 2.2 to enable Linux to run in Symmetrical Multi-Processing mode, i.e. to avoid race conditions on Multiprocessor systems. In the Linux 2.2 multiprocessor kernel critical sections are protected by spinlocks.

The so called ´Preemption Patch´ for Linux 2.4, since Linux 2.6 named ´Preemptible kernel´, makes use of the same critical sections protecting them with preemption locks in the single processor version of the Linux kernel. If such a critical section can also be used by an interrupt service routine (ISR), an additional interrupt lock is necessary to avoid race conditions and data inconsistencies.

The concept of the Preemptible Kernel with critical sections is a more general one than introducing Preemption Points.

Therefore, Linus Torvalds, the founder, chief maintainer and chief developer of the Linux kernel, integrated the 'Preemptible kernel' as a kernel configuration option at compile time into the standard Linux vanilla kernel 2.6. This was a big step towards a higher preemptability since the kernel is now preemptible except for critical sections and interrupt service routines.

Nevertheless, also the Preemptible Linux 2.6 kernel shows long latencies due to some remaining long critical sections. Some of them have been cut into pieces by introducing preemption points. Kernel

patches doing that have been named ´spinlock breaking patches´ for Linux 2.4, as they release the locks at points in a critical section, where no data inconsistencies can evolve, and then call the scheduler to execute a time-critical task, if needed. For the Linux 2.6 kernel Ingo Molnar named his lock breaking patches ´voluntary preemption patches´ as they make preemption points available, that had originally been introduced into Linux 2.6 only for debugging [66].

Latencies in Linux 2.6, even with ´Preemptible kernel´ option enabled are due to the remaining non-preemptible critical sections and interrupt locks.

Another possible solution to reduce these latencies is to convert nearly all preemption locks - spinlocks in the SMP case - into mutexes. A mutual exclusion mechanism allows preemption inside a critical section in general. Therefore, it reduces preemption latencies in the common case, and it is the first concept to make the Linux kernel 'fully preemptible'. For every data structure that has to be protected another mutex can be implemented. Only in the rare case that two processes or threads need the same mutex, the second one has to wait and is put asleep. The time it has to wait for the semaphore is named ´kernel latency time´ (KLT) [58].

Therefore, generally the mutex concept reduces latencies and enhances Linux soft-realtime capabilities. Research in this direction has been carried out in this thesis since 2002 [16, 25, 27]. Nevertheless, the mutex concept doesn´t make Linux hard realtime capable.

There were a number of problems to overcome before this lock conversion can take place in the whole Linux kernel:

Using mutexes deadlocks can arise: Deadlocks can be avoided by defining an order of mutexes, every task has to comply with when acquainting a mutex. Because the Linux kernel runs on SMP systems since Linux 2.2, this order of locks has already been established for Linux 2.2 and is maintained.

Moreover, it turned out that - without further changes - preemption and interrupt locks, which protect critical sections also used by interrupt service routines, can´t be converted into mutexes, because an interrupt handler is never allowed to sleep.

A solution for this problem has been presented probably first by Scott Woods and his team at Timesys Inc. who presented an enhanced Linux kernel 2.4.7 for realtime purposes [96]. This kernel handled Linux interrupt service routines by kernel threads of high realtime-priority. Fujitsu Inc. and some other kernel developers have developed similar patches. The kernel developer Ingo Molnar adopted the idea and added a kernel configuration option named ´IRQ thread patch´ to Linux 2.6 and thus made this option available for the standard Linux kernel.

Using interrupt handlers executed by kernel threads also those preemption and interrupt locks can be converted into mutexes, that are accessed by interrupt handlers. The reason is that every kernel thread is allowed to sleep, in the rare case another instance has acquainted the mutex before. This is possible only with so called ´threaded interrupt handlers´, not with the interrupt service routines of the standard Linux kernels, which are generally not allowed to sleep. Of course, these interrupt handler kernel threads are preferred by the scheduler, because they are scheduled using the SCHED_FIFO realtime scheduling class. A further advantage the concept of threaded interrupt handlers provides is that kernel threads can be prioritized, that way a very important realtime task could be prioritized even higher than an interrupt of a lower priority.

Another important advantage of using mutexes instead of preemption locks / spinlocks is that all former interrupt locks around critical sections can be dropped in such a fully preemptible kernel. This means that a fully preemptible Linux kernel contains much less interrupt locks than any other Linux kernel before.

When using mutexes instead of preemption locks latencies like the kernel latency time (KLT) as well as latencies due to priority inversion situations can arise.

In this thesis, in order to overcome priority inversion situations different priority inheritance protocols, one proposed by [103], have been chosen, supplemented and implemented for the Linux kernel and published under the GNU Public license of version 2, the GPL2 in 2003 for Linux 2.4 [25, 27, 31]. As proof of concept one preemption lock / spinlock of the Linux kernel has been converted into a mutex and interrupt handlers have been executed by threads [16]. This modified Linux 2.4 kernel showed to be stable on uniprocessor systems as well as on dual processors tested at our institute.

Also the modified Linux kernel 2.4.7, presented by Timesys Inc., contained a commercial and encrypted priority inheritance protocol (PIP). The implementation, i.e. the source code of this PIP is not known to the public. This made the Timesys Linux kernel semi-commercial and kernel developers doubt whether this is compliant with the GPL. Timesys Inc. replaced all preemption locks of the ´Preemptible kernel patch´ with critical sections protected by mutexes using their own semi-commercial priority inheritance protocol.

In 2004, the priority inheritance protocols, developed and published in this thesis, have been tested by MontaVista Inc., a well-known Californian distributor of a Linux distribution for the embedded and realtime market [67]. Sven-Thorsten Dietrich at MontaVista chose the easier priority inheritance protocol SPIP/EPV2, implemented in this thesis, named 'PMutex' [16, 31], to convert nearly all preemption locks of the Linux kernel into mutexes and to protect them from priority inversion situations.

In cooperation with MontaVista Software Inc. the priority inheritance protocols have been ported to Linux 2.6 [39] and in October 2004 MontaVista released and published their first fully preemptible Linux kernel built using the mutexes with priority inheritance protocol implemented in this thesis [19, 64, 68]. The project has been named ´MontaVista Linux Open Source Real Time Project´ with UniBwM as an early contributor.

As MontaVista Software Inc. is famous in the Linux realtime and embedded community for their work at the ´Preemptible kernel´, the High Resolution Timer Patch and some other realtime enhancements to the Linux kernel [67, 2], kernel developers interested in realtime noticed this project to create a fully preemptible Linux kernel under GPL2 [105], not under a semi-commercial license.

After the presentation of the MontaVista project [64] and another similar one, Ingo Molnar, kernel developer at Red Hat Inc., started to provide and maintain a so called RT-patch or Realtime Patch for Linux 2.6 [66].

The RT-patch replaces all preemption locks / spinlocks by modified Linux kernel mutexes and uses threaded interrupt handlers. This fully preemptible kernel could become a configurable part of future Linux kernels, but that hasn´t been decided yet.

This thesis analyzes in detail advantages, disadvantages and frontiers of all solutions to improve the preemptability of the Linux kernel, as presented above, evaluating their theoretical concepts and measuring and comparing available implementations, also regarding their impact on the performance. A measurement tool, called 'software monitor', developed in a former thesis at our institute [58] has been of great help to analyze internal operations of the Linux kernel. During this thesis it has been ported to use the Time Stamp Counter (TSC) of modern Intel compatible processors instead of an ISA or PCI-timer card. Furthermore, the 'software monitor' has been extended to work also on multiprocessor systems.

The results have been published on different conferences, so they were available to the community [23, 25, 30]. That was preferable than publishing all only in the end, because such results were of greatest use to the community of Linux kernel developers during the development process took place and decisions had to be made.

Since the concepts to make the Linux kernel more preemptible made good progress, it seemed reasonable to combine these efforts with a higher frequency to actualize the Linux time basis. In Linux 2.4 and the kernel versions before the timer basis is updated only every 10 milliseconds.

Therefore, in this thesis the UTIME patch, originally ported to Linux by the Kansas university [41, 50], has been ported from Linux 2.2 to 2.4, combined with one of the Low Latency patches and published under GPL2 as 'Rapid Reaction Linux' [22, 24, 32]. This Linux kernel patch for the Intel x86 architecture provides a much better timing resolution for foreseeable events, as it allows to program the timer interrupt to the next event in time, that is known before. That's a very efficient way as additional interrupts are only generated when needed, e.g. to schedule a task precisely. Measurements made with the implementations developed in this thesis showed that on average a timing resolution of about 20 to 30 microseconds to schedule an process at a predefined time is possible on current hardware, although there can be higher delays if the kernel currently holds any locks. This timing resolution is possible, since Rapid Reaction Linux makes use of the TSC processor register. The TSC register has been introduced by Intel with the Pentium architecture. This way Rapid Reaction Linux, developed for the Linux 2.4 kernel first in 2001 [22], combined both advantages and focal points of this thesis: A higher preemptability and a better timing granularity of the Linux kernel.

One similar project has been presented to the Linux community, called High Resolution Timer patch [2]. This patch is based on the same UTIME patch as Rapid Reaction Linux. Although this patch is a competitor, it assures that the concept has been worth the effort. Later on, in 2005, the LibeRTOS Project, also based on UTIME and KURT [41, 50], worked in the same direction [14]. Parts of both projects have been integrated into Ingo Molnar's RT-patches for Linux 2.6 [66] and perhaps will find step by step their way into the standard Linux kernel.

The Linux 2.6 standard vanilla kernel multiplied the frequency of the timer interrupt of Linux 2.4 by ten, in order to provide a higher timing resolution of about 2 milliseconds for the user. Measurements in this thesis showed that the overhead of this rather simple method is not negligible [24].

During this thesis the scheduling subsystem of the Linux kernel has been analyzed in detail. A latency that increased in a linear way depending on the number of processes ready to run - due to a loop over the ready queue in Linux 2.2 and 2.4 - has been measured [23]. The scheduler of the Linux standard kernel has been improved to show only a constant time to schedule in Linux 2.6 by the kernel developer Ingo Molnar and others [65] as the measurements show [65, 27].

A further development regarding time-critical tasks done by Linux kernel developers has been to make the so called Soft-Interrupts executed by kernel threads since Linux 2.4.10. Since that version they are executed asynchronously. Before they have been a part of the scheduler and thus contributed to the scheduling latency, when a task had to be preempted.

To detect latencies and to evaluate implementations done in this thesis several measurement tools have been implemented, enhanced or ported from other operating systems to Linux, amongst others e.g. a software monitor, developed by [58, 33], that provided an insight into the operating system, and the well known Rhealstone Benchmark [43, 42, 7, 23] for a performance evaluation. The Linux development centers of several big companies asked for the code of this Rhealstone implementation.

## 9.2 Future prospects

This thesis focuses on how to change the Linux kernel in order to make it more suitable to perform time-critical tasks. The summary above resumed the development that has been done in Linux 2.2, 2.4 and 2.6.

Although Linux 2.6 with its higher timing precision of 2 milliseconds, cp. chapter 8, and its better preemptability - if compiled with 'Preemptible Kernel' option - is more suited for time-critical tasks than Linux 2.2 or 2.4, there are still a lot of things to do.

Even Linux 2.6 - compiled with configuration option 'Preemptible Kernel' - still has some long critical sections where the Linux kernel is not preemptible. Therefore, Ingo Molnar's 'Voluntary Preemption' Patch, which is a configurable part of his RT-kernel patch [66], introduces preemption points into these long critical sections in order to reduce their latencies.

Another configurable part of the RT-patch for Linux 2.6 contains the modified kernel mutexes to make the Linux 2.6 kernel fully preemptible. Ingo Molnar and other kernel developers recently managed to add a priority inheritance protocol [106, 107] as a configurable kernel option at compile time to the stable vanilla standard Linux kernel in version 2.6.18, that can be used for mutexes in user space as well as for those in the kernel. Already before, the IRQ-Thread patch, i.e. interrupt handlers executed by kernel threads, had become a configurable option of the stable Linux kernel. That way, the vision of this thesis of a fully preemptible stable mainstream Linux kernel could become true in the nearer future, if even more parts of the RT-patch, f.ex. the modified kernel mutexes, go into the standard Linux kernel [106].

Therefore, it's possible that one of the next stable version of the Linux kernel contains a compiler option 'PREEMPT_RT' to make the Linux kernel fully preemptible, as the already cited RT-patch does.

Possibly, also more and more parts of the High Resolution Timer Patch [2] - a competitor of 'Rapid Reaction Linux', based on the same principles and predecessors [41, 50] - will be a configurable part of future stable version of the Linux kernel. Furthermore, parts of the preemptible interrupt handling of the 'LibeRTOS' project [14] already became and other parts could become parts of a future stable Linux kernel release, as the development of the Linux kernel is a very dynamic process.

Another issue that hasn't been covered by this thesis completely is to implement so called 'Prioritized I/O', for instance as proposed by the POSIX standard. Prioritized I/O means that data of input/output operations done by a task with soft-realtime or simply higher priority is preferred compared to data sent to a device by a task of lower priority. An example for such I/O operations is data transfer to a hard disk or to a network interface card. A first step to accelerate network traffic has been proposed in this thesis as well as by Timesys Inc. by raising the SoftIRQ Thread(s) to soft-realtime priority. But this will make all network I/O to be processed faster, not only that of processes of high priority and therefore this isn't a general solution. First approaches and implementations for Prioritized I/O exist, but have not yet been integrated into the standard Linux kernel or the 'glibc' library.

Also a realtime file system hasn't been integrated into the Linux kernel yet. As shown, also some POSIX macros are not implemented in the current Linux 2.6 kernel [74].

Hard realtime requirements can't be met by the Linux kernel, this is a goal standard operating systems don't reach so far. Perhaps they never will, because many different developers contribute for instance driver code for peripheral devices, that isn't normally checked whether producing high latencies or not. Another important reason is that most users of standard operating systems don't demand for hard-realtime capabilities, but for as much average performance and efficiency as possible.

Solutions offering hard realtime available already today are dual kernel systems like RTLinux [85], which schedule the Linux kernel as the task of lowest priority of a small RTOS, a RealTime Operating System. As LinuXRT of RTAI show [83], it's possible that the hard RTOS kernel of such a dual kernel approach offers its limited services also to time-critical tasks in the Linux userspace. But often time-critical tasks have to use a special API to communicate with the RT-kernel, and apart from that, these dual kernel solutions don't offer the full functionality of Linux under realtime conditions. Another possibility are systems like Adeos [83, 102], Xenomai [83, 13] or PikeOS [40], which can host several operating systems, for instance a RTOS and a non-RTOS like Linux, on the same computer hardware in parallel.

Although Linux cannot fulfill hard realtime requirements, as shown in this thesis, Linux has made good progress to process time-critical tasks with soft-realtime requirements in last years and will probably proceed this way in the next years to come. Therefore, to do time-critical operations using Linux like looking TV, listening to music while the PC is doing additional other tasks in the background will become even more common than it is today.

Furthermore, it is to expect that industry will offer more Embedded Devices running Linux. Even today PDA's and some GSM mobile phones with their stringent timing requirements are running Linux [21]. There are big industrial cooperations like [5] that work to use Linux in Consumer Electronics, i.e. as multimedia platform in embedded devices.

Much likely, users will proceed to require standard operating systems to show a better soft-realtime performance and higher data throughput. That way, there are still a lot of things to do, since optimizing complex systems is nearly always possible. Furthermore, new peripheral hardware devices will allow more precise timing in future. On the whole, an operating system like every software program isn't statical, as beyond the hardware is advancing and above users and applications create new needs and requirements.

## General perspectives of the Linux kernel

The Linux kernel - as one of its predecessor called 'Minix' - is among the first Unix kernels running on relatively cheap IBM compatible personal computers. Furthermore, it's the first Unix kernel with a huge number of installations, not only used by companies, but also by home users.

Since the Linux kernel is distributed using the GPL, the GNU Public License of the Free Software Foundation [15], it made and still makes the rich possibilities of UNIX, including its broad shell-, multiuser- and networking capabilities, available to home users and industrial users, free of license costs. Additionally, the GPL allows every human being to propose improvements to the Linux kernel. Whether these are useful or not is decided by the so called 'Open Source Community' of Linux kernel developers.

That way, the Linux kernel can be considered as a sort of Digital DNA. It's a storage of the best ideas and concepts available worldwide to run computers of today the most efficient possible. The Linux kernel can be seen as one of the positive effects of globalization, using the GPL License it's part of the heritage

of human culture, free of patent rights. The fast development of the Linux kernel has only been possible using the Internet, this new global communication medium, that allows to build up virtual global groups with same interests, for instance the 'village' of Linux kernel developers.

An interesting phenomenon is that the project to develop the Linux kernel is supported as well by big companies like IBM or Siemens for instance or by smaller ones, as well as by universities, other governmental organizations and by communities of private persons, often called 'Linux user groups', and private individuals themselves.

The fast development of the Linux kernel shows that globalization can unite supra-nationally the best forces of mankind, those with commercial, with scientific, teaching or with different interests. An interesting point for individuals as well as for scientists is that the GPL license names the person who has written or changed some lines of code in the Linux kernel. For researchers and teachers Linux is interesting as new concepts can be implemented, published, tried out and judged by others. When researching the Linux kernel one works on a system that is also commercially important, so no researcher risks to work in a field no others are interested in. On the other hand, progress is fast and many researcher work in the same field.

Because of the GPL, the Linux kernel is used increasingly in countries with a fast growing market, for instance China - with its distribution named 'Red Flag Linux' - and Brasilia as well, that has announced to use Linux in governmental organizations. Therefore, the Linux kernel will help also the citizens of these countries to take part in the global development of informatics, of the Internet, and therefore to take part in global communication and global development.

# Appendix A

# Measuring systems

The measurements done in this thesis have been carried out on the following Intel compatible personal computers, which our institute provided thankworthy.

## A.1 AMD K6

- AMD K6 CPU, running at 400 MHz,
- 256 KB Level 1 Cache
- 128 MB of Random Access Memory (RAM)
- IDE hard disk,
- PCI bus speed: 33 MHz
- bought in 2000

## A.2 Pentium 4

- Pentium 4 CPU, running at 2.20 GHz,
- 512 MB of Random Access Memory (RAM)
- 60 GB IDE hard disk,
- bought in 2003

# Bibliography


[1] **Please notice: Some of the following entries of this bibliography contain links to the Internet or to publications. I declare that I'm not responsible of the content of the web-sites or publications these links point at and that they do only reflect the opinions of their respective owners and authors, but not mine. The content of these links may be changed by their respective owners.**

[2] Anzinger, Salisbury, de Vries, *High Resolution Timer Project (HRT)*, 2001, `http://high-res-timers.sourceforge.net` `http://sourceforge.net/projects/high-res-timers`

[3] Alexander v. Bülow, Jürgen Stohr, *Realzeit mit Multiprozessor-Systemen auf PC-Basis*, Lehrstuhl für Realzeit-Computersysteme, Technical University Munich, Germany, Linux automation conference, Hannover, Germany, March 2004, `http://www.linux-automation.de/konferenz_2004/papers/index.html`

[4] Comedi project, Linux and RTLinux Device Drivers and Measurement Control, `http://www.comedi.org`

[5] Consumer-Electronics-Linux-Forum, `http://www.celinuxforum.org`

[6] M.H.DeGroot, *Probability and Statistics*, Second Edition, Addison-Wesley Publishing Company

[7] Digital Equipment Corporation, Maynard, Massachusetts, revised July 1998, *Performance Evaluation: DIGITAL UNIX Real-Time Systems*, `http://citeseer.ist.psu.edu/274569.html`

[8] U.Drepper, I.Molnar, *White Paper of the New POSIX Thread Library, NPTL*, `http://people.redhat.com/drepper/nptl-design.pdf`

[9] Romesch Düe, *Analyse der Interruptantwortzeit und Preemptivität von Linux 2.6 Kerneln*, Student Research Project, University of German Federal Armed Forces Munich, 2005

[10] Embedded Linux Portal, Newsserver, `http://www.linuxdevices.com`

[11] *Programming for the real world, POSIX.4*, Bill O. Gallmeister, O'Reilly & Associates, Inc. Jan. 1995

[12] Gerth W., Wolter B., *Orthogonale Walsh-Korrelation zur qualitativen Beurteilung der Reaktivität von Betriebssystemen*, P. Holleczek (Hrsg.): PEARL 2000 - Workshop über Realzeitsysteme, Springer Verlag Berlin Heidelberg, Germany, 2000, pp.33-42, ISBN 3-540-41210-7

[13] Philippe Gerum, *Xenomai project, Implementing a RTOS emulation framework on GNU/Linux*, August 2001, `http://www.xenomai.org`

[14] Thomas Gleixner, Linutronix, *LibeRTOS, Linux Based Enhanced RealTime Operating System*, Linux automation conference, Hannover, Germany, March 2004, `http://www.linux-automation.de/konferenz_2004/papers/index.html`

[15] GNU Public License, Free Software Foundation, `http://www.gnu.org/copyleft/gpl.html` `http://www.gnu.org/software/libc/libc.html`

[16] Dirk Grambow, *Untersuchung zur Verbesserung der Preemptivität aktueller Linux Kernel*, Master's thesis, University of German Federal Armed Forces Munich, ID 17/02, 2002

[17] W.Haas, M.Zorn *Methodische Leistungsanalyse von Rechensystemen*, Oldenbourg Verlag, Germany, 1995

[18] William von Hagen, *Migrating to Linux kernel 2.6 – Part 2: Migrating device drivers to Linux kernel 2.6*, 02/2004, http://linuxdevices.com/articles/AT4389927951.html

[19] Heise-News: *Echtzeit-Initiative für Linux von MontaVista, MontaVista Linux Open Source Real Time Project*, October 12, 2004,
http://www.heise.de/newsticker/meldung/52063 and /52051,
http://www.pro-linux.de/news/2004/7375.html

[20] Heise-News: *MontaVista stellt Echtzeit-Patches für Linux vor,*
http://www.heise.de/newsticker/meldung/62374

[21] Heise-News: New Motorola mobile phone models based on Linux, July 2005,
http://www.heise.de/mobil/newsticker/meldung/61336

[22] A.Heursch, H.Rzehak, *Rapid Reaction Linux: Linux with low latency and high timing accuracy*, 5th Annual Linux Showcase & Conference of the USENIX Association & the Atlanta Linux Showcase, Inc., ALS 2001, Oakland, California, USA, November 5-10, 2001
http://inf3-www.informatik.unibw-muenchen.de/research/linux/rrlinux/rapid.html
http://www.linuxshowcase.org/2001/heursch.html
http://citeseer.ist.psu.edu/568357.html

[23] A.Heursch, A.Horstkotte, H.Rzehak, *Preemption concepts, Rhealstone Benchmark and scheduler analysis of Linux 2.4*, Real-Time & Embedded Computing Conference, RTEC 2001, Milan, Italy, Nov. 26-29., 2001, http://inf3-www.informatik.unibw-muenchen.de/research/linux/milan/measure_preempt.html

[24] Arnd C. Heursch, Alexander Koenen, Witold Jaworski and Helmut Rzehak, *Improving Conditions for Executing Soft Real-Time Tasks Timely in Linux*, Fifth Real-Time Linux Workshop, Universidad Politecnica de Valencia, Spain, November 9-11, 2003, proceedings, p.131-138,
http://www.realtimelinuxfoundation.org/events/rtlws-2003/papers.html#PAPER_heursch

[25] Arnd C. Heursch, Dirk Grambow, Alexander Horstkotte, Helmut Rzehak, *Steps towards a fully preemptable Linux kernel*, 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming, WRTP 2003, Lagow, Poland, May 14-17, 2003, proceedings, p.159-164, Editors: COLNARIC, ADAMSKI & WEGRZYN, title: Real-Time Programming 2003, publication date: mid November 2003, Elsevier Science Ltd., ISBN: 008 044203 X,
http://inf3-www.informatik.unibw-muenchen.de/research/linux/milan/WRTP03.pdf,
http://www.iie.uz.zgora.pl/wrtp03/

[26] Arnd Heursch, Michael Mächtel, *Ganz pünktlich!, Kernel-Erweiterungen, die Linux für zeitkritische Aufgaben qualifizieren*, journal 'Linux Enterprise', Magazin für Professional Linux & Open Source Development, editor: Software & Support Verlag GmbH, Germany, edition 4/2003, pp.79-85

[27] Arnd C. Heursch, Dirk Grambow, Dirk Roedel and Helmut Rzehak, *Time-critical tasks in Linux 2.6*, Linux automation conference, Hannover, Germany, March 2004,
http://www.linux-automation.de/konferenz_2004/papers/index.html

[28] A. Heursch, H. Rzehak, M. Will et al., *Linux für eingebettete Systeme*, GI- Workshop über Realzeitsysteme Boppard, Germany, November,23-24, 2000, Tagungsband Reihe Informatik aktuell des Springer- Verlags 2000, pp.123-132, ISBN 3-540-41210-7

[29] Arnd Heursch, *Zeitig ans Ziel, Zeitkritische Aufgaben mit dem neuen Linux-Kernel 2.6 bearbeiten*, journal 'Linux Enterprise', Magazin für Professional Linux & Open Source Solutions, editor: Software & Support Verlag GmbH, Germany, edition 2/2005, pp.39-43, ISSN: 1619-7968

[30] Arnd Heursch, Witold Jaworski, Romesch Düe and Helmut Rzehak, *Konzepte zur Steigerung der Preemptivität des Linux Kernels 2.6*, Linux automation conference, Hannover, Germany, March 2005,
http://www.linux-automation.de/konferenz/papers/index.html

[31] Download of GPL'ed implementation of mutexes for Linux 2.4 and 2.6, INF3, Department of Computer Science, University of German Federal Armed Forces Munich, 2003,
http://inf3-www.informatik.unibw-muenchen.de/research/linux/mutex/mutex.html

[32] Rapid Reaction Linux, homepage of the Institute of Information Technology Systems, Department of Computer Science of the University of German Federal Armed Forces Munich, 2001
`http://inf3-www.informatik.unibw-muenchen.de/research/linux/rrlinux/rapid.html`

[33] Sven Heursch, *Latenzzeitmessungen des Betriebssystems Linux (Real-Time Linux/KURT Linux)*, Master's thesis at the University of German Federal Armed Forces Munich, ID 07/1999, `http://inf3-www.informatik.unibw-muenchen.de/research/linux/dipl_heursch/`

[34] S. Heursch, M. Mächtel, 1/2000, *Linux in Eile: Linux im Real-Time-Einsatz*, Germany, iX 1/2000, pp.141-145

[35] Alexander Horstkotte, *Reengineering and analysis of the realtime capabilities of the Linux scheduler*, Student Research Project, University of German Federal Armed Forces Munich, IS 03/2001

[36] Alexander Horstkotte, *Untersuchung und Bewertung von Echtzeiterweiterungen für Linux*, Master's thesis, University of German Federal Armed Forces Munich, ID 11/01, 2001

[37] IEEE, Institute of Electrical and Electronics Engineers, Inc., `http://www.ieee.org`

[38] Witold Jaworski, *Zeitverwaltung in aktuellen Linux Kerneln*, Student Research Project, University of German Federal Armed Forces Munich, IS 07/03, 2003

[39] Witold Jaworski, *Schutz kritischer Bereiche durch Mutexe als Konzept zur Verbesserung der Preemptivität des Linux Kernels 2.6*, Master's thesis ID 44/2004, University of German Federal Armed Forces Munich

[40] Robert Kaiser, Sysgo AG, *PikeOS*, Linux automation conference, Hannover, Germany, March 2004, `http://www.linux-automation.de/konferenz_2004/papers/index.html`

[41] Kansas University, UTIME Patch - *Micro-Second Resolution Timers for Linux*, `http://www.ittc.ku.edu/kurt/`

[42] Kar, R., April 1990, *Implementing the Rhealstone Real-Time Benchmark*, Dr.Dobb´s Journal, `http://www.ddj.com/articles/search, Search for notion Rhealstone`

[43] R. P. Kar and K. Porter, Feb. 1989, *Rhealstone: A Real-Time Benchmarking Proposal,* Dr. Dobbs Journal, vol. 14, pp.14-24, `http://www.ddj.com/articles/search, Search for notion Rhealstone`

[44] Robert Koch, *Erstellung und Kombination von Messtools unter Nutzung der Möglichkeiten neuerer Hardware-Komponenten*, Master's thesis, University of German Federal Armed Forces Munich, ID 18/02, 2002

[45] Alexander Koenen, *Untersuchung und Verbesserung des Zeitverhaltens aktueller Linux Kernel*, Master's thesis, University of German Federal Armed Forces Munich, ID 19/02, 2002

[46] W. Kriechbaum, *Adding real-time capabilities to a standard Unix implementation: The AIX Version 3.1.5 approach*, in Helmut Rzehak (Hrsg.) 'Echtzeitsysteme und Fuzzy Control', Konzepte, Werkzeuge, Anwendungen, ISBN 3-528-05432-8, Vieweg Verlag, Germany, 1994

[47] Bernhard Kuhn, Metrowerks, RTIRQ Patch, 2004
`http://www.ussg.iu.edu/hypermail/linux/kernel/0401.1/0300.html`

[48] Marco Kuhrmann, *Untersuchung des Realzeitverhaltens aktueller Linux Kernel und Kernel-Varianten*, Student Research Project, University of German Federal Armed Forces Munich, IS 24/03, 2003

[49] Kuka Inc., `http://www.kuka-controls.com/`

[50] KURT-Linux, Kansas University Real-Time Linux,
`http://www.ittc.ku.edu/kurt/`

[51] Lineo Inc., `www.lineo.com/`

[52] Linux Audio, Latency Graph by Benno Senoner
`http://www.gardena.net/benno/linux/audio/`,
`http://www.kernel-traffic.org/kernel-traffic/quotes/Benno_Senoner.html`

[53] Some postings of the Linux Kernel Mailing List concerning MontaVista´s Preemption Patch, Sept. 2000,
George Anzinger: http://lkml.org/lkml/2000/9/6/4, http://lkml.org/lkml/2000/9/12/38
Andrea Arcangeli: http://lkml.org/lkml/2000/9/11/84, http://lkml.org/lkml/2000/9/11/104

[54] Linux/SRT, a Linux Resource Kernel, http://www.srcf.ucam.org/~dmi1000/linux-srt/

[55] Robert Love's Linux Kernel Patches, http://www.kernel.org/pub/linux/kernel/people/rml/, http://www.linuxdevices.com/articles/AT8267298734.html

[56] Open Source Linux Kernel Project 'kpreempt', http://kpreempt.sourceforge.net/, http://sourceforge.net/projects/kpreempt

[57] LynxOS by LYNUXWORKS Inc., http://www.lynuxworks.com/

[58] M. Mächtel, 2000, *Entstehung von Latenzzeiten in Betriebssystemen und Methoden zur meßtechnischen Erfassung*, PhD thesis at University of German Federal Armed Forces Munich, VDI Verlag, ISBN 3-18-380808-0

[59] M. Mächtel, *Unter der Lupe: Verfahren zum Vergleich von Echtzeitsystemen*, journal iX 9/2000, Germany, pp.120-122

[60] M. Mächtel, A. Heursch: *Kommt Zeit, kommt Linux - Der Einsatz von Linux in Echtzeitsystemen*, journal ´Linux Enterprise´, Magazin für Professional Linux & Open Source Development, editor: Software & Support Verlag GmbH, Germany, edition 4/2000, pp.56-61

[61] Michael Mächtel, Vorlesung *Echtzeitbetriebssysteme*, Germany, 2004

[62] H.Malz, *Rechnerarchitektur - Aufbau, Organisation und Implementierung*, Vieweg Verlag, Germany, 1998

[63] Timerkarte PCI-CTR05 by Measurement Computing, formerly Computerboards, manufacturer, http://www.measurementcomputing.com/

[64] John Mehaffey, MontaVista Software Inc., *White Paper for the MontaVista Linux Open Source Real Time Project*, October 2004, http://source.mvista.com/linux_2_6_RT.html

[65] Ingo Molnar, *Linux Low Latency Patch for multimedia applications*, 2000, http://people.redhat.com/mingo/lowlatency-patches/

[66] Ingo Molnar, *RT (Realtime) patches for Linux 2.6*, Red Hat Inc.,2005, http://people.redhat.com/mingo/realtime-preempt/

[67] MontaVista Software Inc., *RT-Scheduler and Preemption-Patch for the Linux Kernel*, 2000 http://www.linuxdevices.com/articles/AT4185744181.html, ftp://ftp.mvista.com/pub/Real-Time

[68] MontaVista Software Inc., California, USA, Realtime web pages, 2005, http://www.mvista.com/products/realtime.html

[69] Andrew Morton, *Linux scheduling latency*, 2001 http://www.zip.com.au/~akpm/linux/schedlat.html

[70] Andrew Morton, *Low Latency Patches*, University of Wollongong, Australia, 2001, http://www.uow.edu.au/~andrewm/linux/schedlat.html

[71] The Open Group, http://www.opengroup.org

[72] OSDL Labs, USA, Carrier Grade Linux, Linux for telecommunication companies, http://www.osdl.org/lab_activities/carrier_grade_linux/

[73] Inaky Perez-Gonzalez, Intel Corporation, *Mutexes in Carrier Grade Linux, in the telecommunication sector*, http://developer.osdl.org/dev/robustmutexes/

[74] POSIX Option Groups, http://people.redhat.com/~drepper/posix-option-groups.html

[75] Mirko Pracht, *Analyse von hardware-bedingten Latenzzeiten auf modernen Prozessoren*, Master's thesis, University of German Federal Armed Forces Munich, ID 20/02, 2002

[76] Peter Puschner, Alan Burns, *A Review of Worst-Case Execution-Time Analysis*, Journal of Real-Time Systems, May 2000, number 2/3, vol. 18, pp.115-128

[77] G. Rabbat, B. Furht, R. Kibler, *Three-Dimensional Computer Performance*, July 1988, IEEE Computer, Vol.21, No.7, pp.59-60

[78] Realtime Linux foundation, `http://www.realtimelinuxfoundation.org`

[79] Realtime Operating System QNX by QNX Software Systems, `http://www.qnx.com/`

[80] Real Time Extension (RTX) for Windows NT by VenturCom Inc., `http://www.vci.com, http://www.ardence.com/`

[81] Realtime Operating System VxWorks by Windriver Inc., `http://www.windriver.com`

[82] Dirk Roedel, *Untersuchung von Konzepten zur Verbesserung des Realzeitverhaltens des Linux Kernel*, Master's thesis, University of German Federal Armed Forces Munich, ID 17/03, 2003

[83] RTAI - Realtime Application Interface of DIAPM, `http://www.rtai.org`

[84] Realtime Networking for RTAI, `http://www.rts.uni-hannover.de/rtnet/`

[85] RTLinux - RealTime Linux of FSMLabs, `http://www.rtlinux.org, http://www.fsmlabs.com`

[86] RTLinux mailing list, `http://www.rtlinux-gpl.org`

[87] Alessandro Rubini, Jonathan Corbet, 2001, *Linux Device Drivers*, 2nd Edition, publisher O'REILLY

[88] H.Rzehak, *Standardbetriebssysteme für zeitkritische Anwendungen ?*, GI-Workshop über Realzeitsysteme, Boppard, Germany, Nov.1997, Reihe Informatik aktuell, Springer Verlag, pp.1-9, ISBN 3-540-63562-9

[89] H. Rzehak, A. Heursch, *Die Eignung von Linux für zeitkritische Anwendungen*, GI-Workshop über Realzeitsysteme, Boppard, Germany, November, 23-24, 2000, Tagungsband Reihe 'Informatik aktuell' des Springer-Verlags 2000, pp.1-11, ISBN 3-540-41210-7

[90] Daniel Schepelmann, *Untersuchung von Interruptsperren im Linux Kernel*, student research project, University of German Federal Armed Forces Munich, IS 18/01, 2001

[91] Sha L., Rjakumar R. and S.Sathaye, *Priority inheritance protocols: An approach to real-time synchronization*, IEEE Transactions on computers, 39, pp.1175-1185, 1990

[92] Balaji Srinivasan, *A Firm Real-Time System Implementaion using Commercial Off-The-Shelf Hardware and Free Software*, Master's thesis, University of Kansas, 1998, `http://www.ittc.ku.edu/kurt/`

[93] SSV Embedded, Boards und Software, `http://www.ssv-embedded.de`

[94] Tanenbaum: Moderne Betriebssysteme, 2. Auflage Hansa Verlag, München, 1995

[95] Gunnar Teege, Vorlesung *Betriebssysteme 1*, University of German Federal Armed Forces Munich, 2004, `http://www.unibw.de/inf3/lehre/vorlesungen/vorl07/bs/index_html/base_view`

[96] Timesys Inc., `http://www.timesys.com`

[97] Linus Torvalds et al., Website and archive of the Linux kernel and Linux kernel source code, `http://www.kernel.org`

[98] Analyzing the Linux kernel source code, `http://lxr.linux.no`

[99] Yu-Chung Wang and Kwei-Jay Lin, *Some Discussion on the Low Latency Patch for Linux*, Workshop on Real Time Operating Systems and Applications and second Real Time Linux Workshop, Florida, USA, Download: see [101], 2000

[100] Clark Williams, Red Hat: *Linux scheduler latency*, article in journal 'Embedded Systems', 28.06.2002, http://www.embedded.com/showArticle.jhtml?articleID=23901605

[101] Phil Wilshire, *Real-Time Linux: Testing and Evaluation*, Workshop on Real Time Operating Systems and Applications and second Real Time Linux Workshop, Florida, USA, 2000, http://www.realtimelinuxfoundation.org/events/rtlws-2000/presentations.html

[102] Karim Yaghmour, Opersys Inc., *Adeos project, Adaptive Domain Environment for Operating Systems*, http://home.gna.org/adeos/

[103] Victor Yodaiken, USA, 2001, *The dangers of priority inheritance*, Draft, http://citeseer.ist.psu.edu/yodaiken01dangers.html

[104] Victor Yodaiken, FSMLabs Technical Report, *Against priority inheritance*, December 28, 2005, http://www.fsmlabs.com/against-priority-inheritance.html

[105] Jonathan Corbet, *Approaches to realtime Linux*, October 12, 2004, http://lwn.net/Articles/106010/

[106] Jonathan Corbet, *Kernel Summit 2006: Realtime*, July 18, 2006, http://lwn.net/Articles/191782/, http://lwn.net/Articles/191979/ http://lwn.net/Articles/138174/

[107] Steven Rostedt, *[PATCH] Document futex PI (Priority Inheritance) design* , May 9, 2006, http://article.gmane.org/gmane.linux.kernel/404708

[108] *IEEE standards*, http://stdsbbs.ieee.org/descr

[109] VMWare Inc., http://www.vmware.com/