Time-critical tasks in Linux 2.6
Concepts to increase the preemptability of the Linux kernel

Arnd C. Heursch, Dirk Grambow, Dirk Roedel and Helmut Rzehak
Department of Computer Science
University of Federal Armed Forces, Munich,
Werner-Heisenberg-Weg 39, 85577 Neubiberg, Germany
arnd.heursch@unibw-muenchen.de, rz@informatik.unibw-muenchen.de
http://www.informatik.unibw-muenchen.de/inst3/index_de.php

Linux Kernel, soft realtime, preemptability, critical sections, mutex, priority inversion, priority inheritance, scheduler, timing resolution

Abstract

In the new Linux 2.6 kernel some features that are important for time-critical tasks have been integrated.

In Linux 2.6 the Preemptible Kernel Patch [6] has been integrated as a kernel configuration option. If selected Linux 2.6 kernel code can now be preempted, if a process of higher priority requires the processor. This is a very important feature for soft-realtime tasks. In Linux 2.6 the timer interrupt service routine is preset to be executed every 1 millisecond, ten times more often than in Linux 2.4, this means a 10 times higher timing precision in Linux 2.6, although some overhead. In Linux 2.4 the execution time of the scheduler depends on the number of tasks that are currently ready to run. In Linux 2.6 a new scheduler has been introduced with a constant scheduling time as measurements show.

But even in Linux 2.6 with activated Preemptible kernel option there are still a lot of critical sections in the kernel that produce long latencies. Some of the longest critical sections have been cut into smaller critical sections in Linux 2.6 by introducing some Preemption Points out of the former Spin-lock-breaking [5] and Low Latency patches [2]. Thus on the whole Linux 2.6 is better optimized to execute time-critical or soft-realtime tasks than the Linux kernels before ever were.

Nevertheless there are further concepts to deal with the remaining critical sections: The frequency of occurrence of latencies can be reduced by changing general preemption locks into mutexes as the Linux GPL kernel 2.4.7 of the Timesys company[8] does. But in the rare case, that 2 processes are requiring the same mutex, long latencies arise again, so even using such a solution rare long latencies can occur.

Mutexes can cause priority inversion. The Timesys kernel implements an only commercially available priority inheritance protocol, which sources are not available. We present a GPL kernel patch which implements a priority inheritance protocol following a paper of Victor Yodaiken in 2001 [9].

Another problem is that since interrupt service routines are not allowed to sleep, preemption locks in interrupt handlers normally can't be changed into mutexes. To change preemption locks that are placed in interrupt service routines too, a further patch is presented, that causes all interrupt service routines besides the timer interrupt to be handled by kernel threads. The technics of this patch is similar to what Timesys uses in its Timesys GPL kernel [8].

1 Introduction

In recent years desktop computers with standard operating systems are used more and more often for soft-real-time tasks, e.g. replaying audio- and video-streams. On non-preemptable operating systems other tasks running in parallel can cause latencies that delay the execution of the soft-real-time task. This paper concentrates on solutions that do not change or extend the system call interface the standard Linux kernel offers to libraries and applications. Therefore hard realtime capable solutions like RTLinux or RTAI, that introduce their own subsystem and API, are not covered herein.

1.1 PDLT - Process Dispatch Latency Time

In this section two important latencies shall be introduced.

The interrupt response time IRT is the time from when an external device generated an interrupt until the execution of the first instruction of the interrupt service routine (ISR). As the scheduler is not involved it is normally much shorter than the PDLT and only about some usecs up to about 15 usecs on modern PCs.

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

The PDLT is higher than the IRT, because even if the task owns the highest priority in the system, besides the IRT the PDLT also contains the duration of the interrupt service routine, the delay of possible preemption locks and the duration of the scheduler.

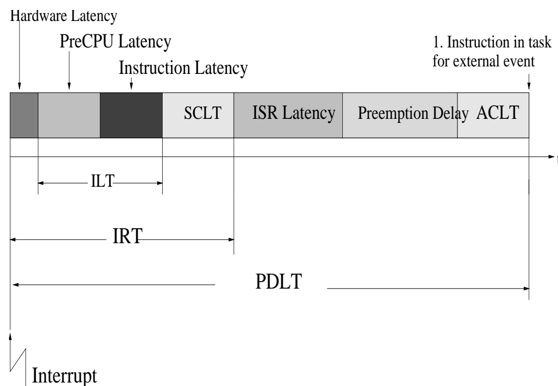

Figure 1: *The PDLT contains several latencies*

1.2 Scheduling Policies

Linux is a standard operating system, not originally designed as a real-time operating system. In Linux there are 2 scheduling classes or policies:

- The standard Linux scheduling algorithm, named SCHED_NORMAL - it has been named SCHED_OTHER in the kernels before Linux 2.6 - is a Round Robin scheduler, which portions a time slice of the processor to any process. On a normal Linux system there are often only SCHED_NORMAL processes.

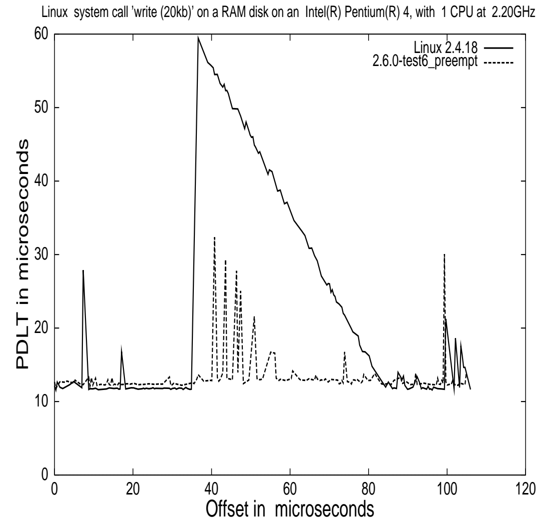

Figure 2: *Calling write(data) with 20 KByte of data on a RAM disk produces in Linux 2.4 and 2.6 a latency of 60 microseconds on our Pentium 4 at 2.2 GHz, a time while this write system call of the Linux kernel can't be preempted. As we can see the Preemptible kernel, here with kernel 2.6.0, reduces that latency to only 30 microseconds. The measurements have been executed using a software monitor described in [7], [4].*

- Time-critical tasks, i.e. soft-real-time tasks, instead should be scheduled using the SCHED_FIFO or SCHED_RR policies, that provide 99 different fixed priorities and are preferred to any SCHED_NORMAL process by the scheduler.

2 Critical sections to avoid race conditions

The Preemption Patch or Premptible kernel patch [6] makes the Linux kernel preemptable in general, so that when an interrupt service routine sets the kernel variable **need_resched** to 1, the scheduler can be invoked in order to bring a task of higher priority to execution, even if the processor is working in kernel mode.

In Linux 2.6 the Preemptible kernel has been integrated as a configuration option that should be chosen before compiling the kernel whenever time-critical tasks have to be done.

The measurement in fig.2 shows that the Preemptible kernel reduces many latencies of the standard Linux kernel 2.4 and 2.6. Often the reduction is even better than in fig.2 as can be seen in fig.3.

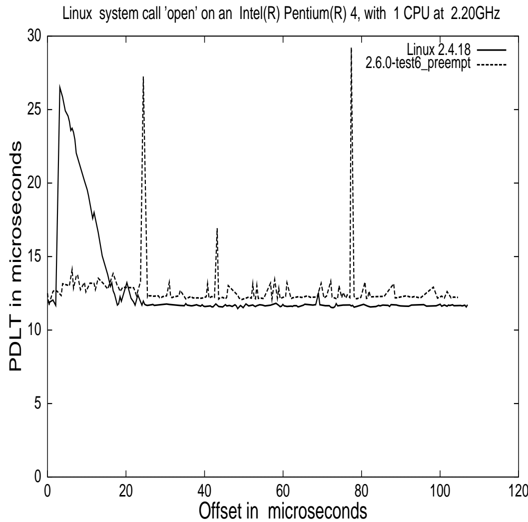

Figure 3: *Calling open to a file on a RAM disk produces in Linux 2.4 and 2.6 a latency of 27 microseconds on our Pentium 4 at 2.2 GHz. As we can see the Preemptible kernel, here with kernel 2.6.0, reduces that latency to the baseline PDLT of beyond 15 microseconds.*

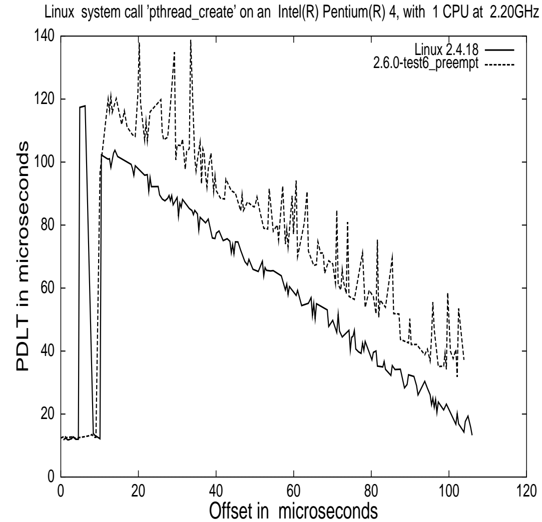

Figure 4: *Calling pthread_create() produces in Linux 2.4 and also in Linux 2.6 even with Preemptible kernel patch a latency of about 120 microseconds on our Pentium 4 running at 2.2 GHz*

In every operating system there are lines of code belonging logically together, which will produce data inconsistencies if they aren't executed atomically at once by only one instance at a time.

As an example the following code inserts a new element called 'new' into a linear list.

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

A process named '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 f.ex. deletes the element of the list named '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' hadn'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 acesses 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'**.

Every operating system contains such critical sections. In standard operating systems like Linux 2.6 with Preemptible kernel they are often protected by general preemption locks instead of mutexes. An example is shown in fig.4, which shows the latency of a `pthread_create`, that is mapped to the `clone` system call in the Linux kernel. As shown by the measurement even the Preemptible 2.6.0 kernel contains that latency caused by a general preemption lock.

General preemption locks are easier to implement, but they lead to a worse temporal behaviour of the whole system compared to mutexes. Mutexes only forbid preemption of a process when 2 tasks are working on the same particular code or subsystem, while preemption locks forbid preemption in general.

Therefore in order to reduce the PDLT many preemption locks should be changed into mutexes, at least if the execution time is longer than 2 scheduler runs.

The following sections are dedicated to show advantages and disadvantages of such a solution using mutexes.

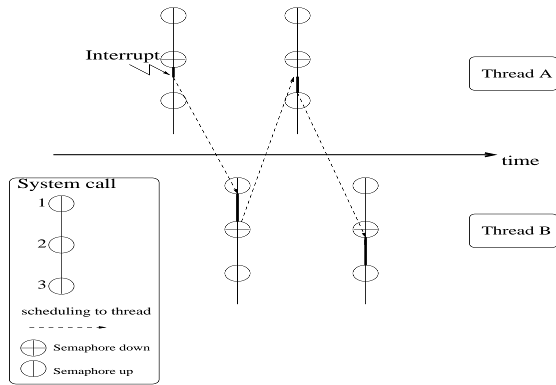

Figure 5: The Kernel Latency Time, KLT

2.1 The Kernel Latency Time (KLT)

Thread A in fig.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.

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:

$$KLT_{max} = 2 * T_{scheduler} + T^{max}_{criticalsection} \qquad (1)$$

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 2.3.

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.

2.2 Mathematical modelling

From the existence of the KLT, see section 2.1, 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.

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.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 desribed in section 2.1. Eq. (1) shows the maximum KLT that can occur. The average KLT is

$$KLT_{average} = 2 * T_{scheduler} + \frac{T_{critical-section}}{2} \quad (2)$$

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

$$delay^{preemption-lock}_{critical-section,average} > p * KLT_{average} \quad (3)$$

$$\frac{T_{critical-section}}{2} > p*(2*T_{scheduler}+\frac{T_{critical-section}}{2}) \quad (4)$$

$$T_{critical-section} > 4 * T_{scheduler} * \frac{p}{1-p} \quad (5)$$

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.

F.ex. 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, f.ex. 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 eq.(4) and (5) 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.

2.3 Priority Inversion - Starvation

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

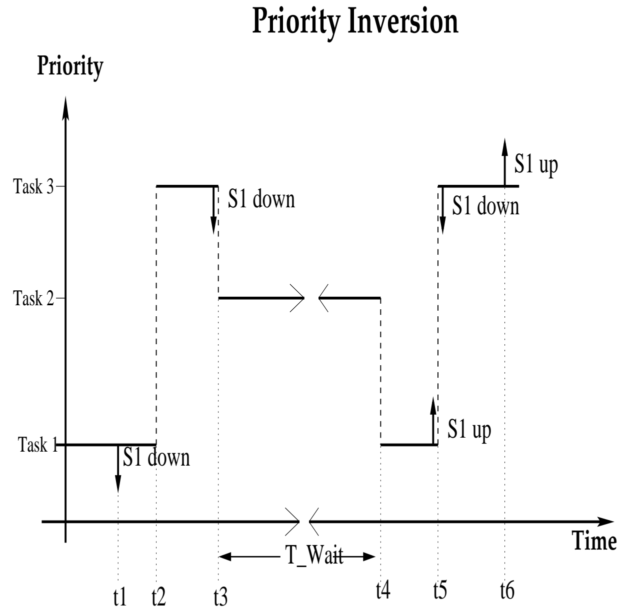

Figure 6: *Priority inversion situation*

According to fig. 6 a situation called priority inversion can occur if a task of lower priority like task1 holding a

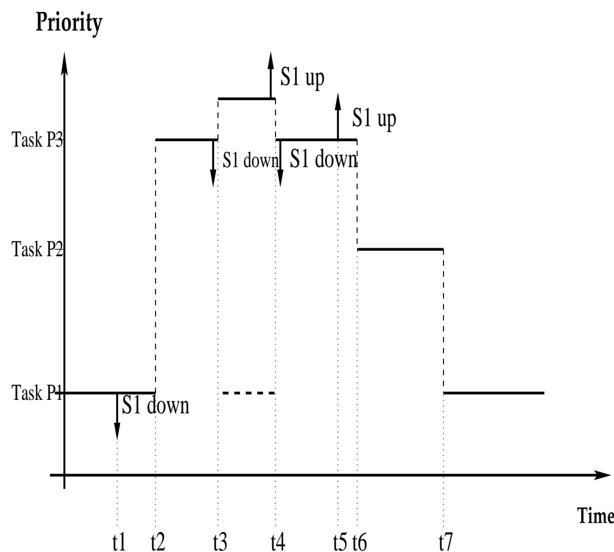

Figure 7: *Priority inheritance protocol to manage simple priority inversion problems*

mutex is preempted by a task of higher priority like task 3 in fig. 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.

2.4 Priority inheritance and priority ceiling protocols

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

2.4.1 Priority ceiling protocol

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

For these reasons priority ceiling isn't the right choice to implement in the Linux kernel.

2.4.2 A simple priority inheritance protocol and its limits

To avoid priority inversion at first a simple priority protocol (SPIP) is presented, which is according to [9] used in VxWorks and many other Realtime operating systems:

- Simple Priority Inheritance Protocol (SPIP):
  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 fig.7 the way a priority inheritance protocol works is shown. Slightly different from the SPIP protocol Task P_1 is raised to a priority level even one above the priority level of Task P_3 in fig.7.

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 1 and 2 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 fig.7. 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:

   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.8 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.

2.4.3 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 [11]. The proposed protocol PIP extends the simple protocol of section 2.4.2 as follows:

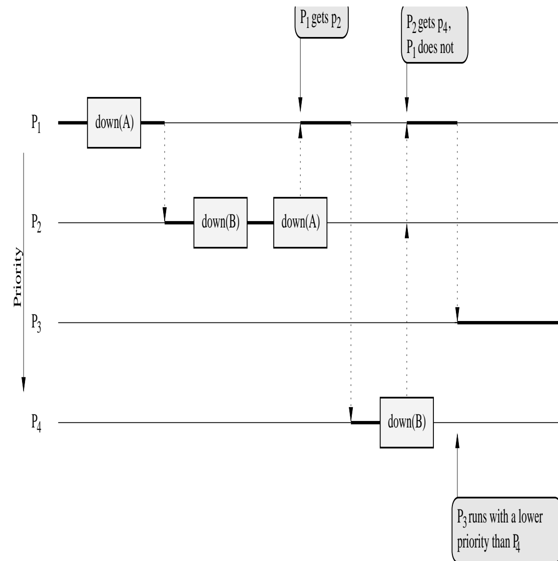

Figure 8: *Priority inversion at chained mutexes*

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 the process P.

But also this protocol is not perfect, as the following counter-example shows, that Victor Yodaiken published in 2001 [9]:

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.

Yodaiken repairs the protocol of [11] 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 helds 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 don't gets 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 [9], 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 implemented hereafter assumes that this inital 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 PV 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, 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 priority in this set.

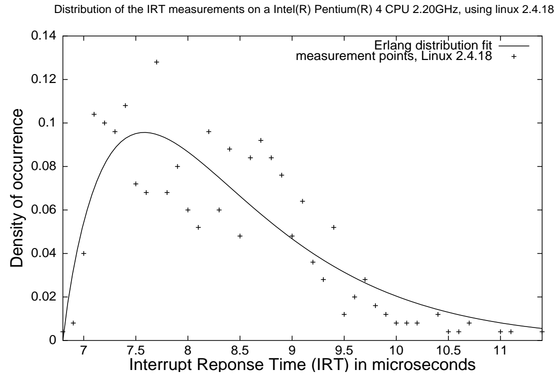

Figure 9: *Distribution of 1000 measured values for the Interrupt Response Time of the paralel port of a* **standard Linux kernel 2.4.18**

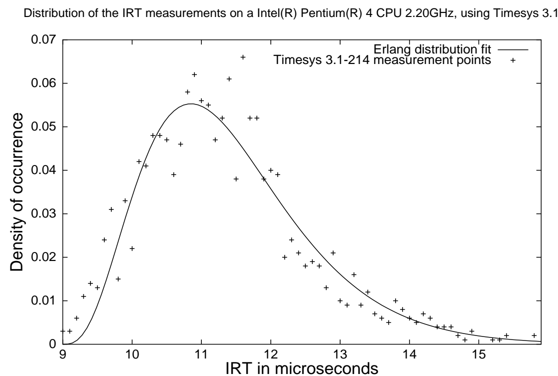

Figure 10: *Distribution of 1000 measured values for the Interrupt Response Time of the paralel port of a* **Timesys Linux GPL kernel 2.4.7**

3 Transforming interrupt service routines into kernel threads

Using the priority inheritance protocol presented above it is possible to replace all those preemption locks by mutexes, which are never entered by any code from an interrupt service routine (ISR). On the other hand it is not possible to replace those preemption locks which at the same time also lock interrupts, called `spin_lock_irqsave()` in the standard Linux kernel. The problem here is that a mutex puts asleep the caller, if it has already been locked before. For an ISR sleeping is forbidden by definition, at maximum it's allowed to execute a short busy wait when acessing an already locked spinlock on a SMP system.

A way out of this problem is to change all Interrupt Service Routines (ISR) besides the timer ISR into kernel threads and schedule these kernel threads at a

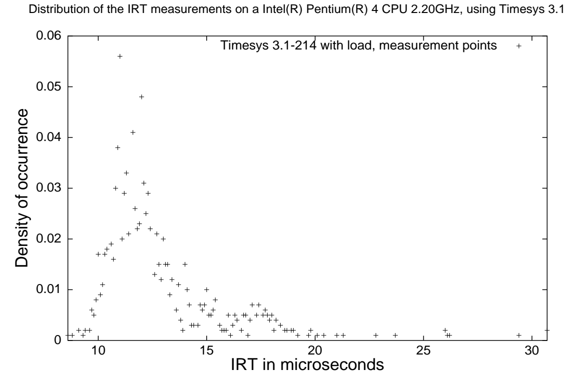

Figure 11: *Distribution of 1000 measured values for the Interrupt Response Time of the paralel port of a* **Timesys Linux GPL kernel 2.4.7 with the load** *of kernel compiling, find, tty and drag & drop operations*

very high soft-realtime SCHED_FIFO priority like the Timesys kernel does. Since kernel threads are allowed to sleep, in a kernel with such an Interrupt-kernel-thread (IKT) patch [10] even the joined spinlock-interrupt locks `spin_lock_irqsave()` can be replaced by mutexes.

Of course to protect very small critical regions in the Timesys Linux kernel, even this kernel contains non-preemptable combined spinlock interrupt locks, but these are held only for very short times, e.g. to do atomic operations on the Timesys mutexes.

As a side effect with such a kernel patch it is possible to assign soft-realtime priorities to interrupts or even prioritize a real-time process higher than some interrupts.

3.1 The implementation of the IKT-patch

The IKT-patch modifies the standard Linux kernel in the following way: 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 and then it is put asleep. 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. After that the kernel thread is put asleep again. Of course interrupts can also be assigned later dynamically using `request_irq(...)` which will start a kernel thread too.

The comparison of the measurements of fig.9 and 10

shows that the IKT-Patch with its kernel threads leads to a higher average and maximum Interrupt Response Time of the Timesys GPL kernel compared to standard Linux 2.4.18. The measurements have been made by generating interrupts on the parallel port of a Pentium 4. Fig.11 shows that under load in rare cases the IRT of the Timesys GPL kernel on our Pentium 4 reaches up to 30 microseconds, probably because of remaining preemption locks, that haven't be changed into mutexes.

4 Latencies of the Linux Scheduler

The scheduler - as an important part of every operating system - is invoked very often. As seen in eq.(1) the use of mutexes will lead to even more scheduler calls. If the IKT Patch is introduced a scheduler call is necessary also before an interrupt handler is handled by a kernel thread.

So the scheduling time for a SCHED_FIFO process, i.e. a soft realtime process is an important characteristics of the system, because this latency is part of every PDLT as can be seen in fig.1.

Therefore the following measurement situation has been set up:

On our single processor PC, a Pentium 4 with 2.2 GHz, we ran only one soft realtime process, using the scheduling policy SCHED_FIFO, and many load processes, scheduled using the policy SCHED_NORMAL, which is the standard Linux time slice scheduler.

After sleeping for one second the SCHED_FIFO process awakens and is queued back into the runqueue. The next time the scheduler is invoked, in the first lines of the kernel function `schedule()` in the file `kernel/sched.c` a time-stamp is taken from the 64 bit Time Stamp Counter (TSC) Register of the Pentium 4, 2.2 GHz processor, which counts a clock cycle every 0.45 nanoseconds since the system was booted. In the Linux kernel there are several macros implemented for x86 processors to take a time stamp from the TSC-Register, f.ex. `rdtsc(low, high)`, where `low` and `high` are of the type `unsigned long`. i.e. 32 bit long.

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 starts executing the SCHED_FIFO process on the processor, a second time-stamp is taken from the TSC. The difference of the 2 time-stamps, which represents the duration of the kernel function `schedule()`, is calculated and printed with a `printk()` into the kernel ringbuffer and from there later on into the file `/var/log/messages`.

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. Out of such a measurement we got every point in fig.12 and 13. Varying the number of SCHED_NORMAL load processes we could draw the lines shown in both figures.

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

4.1 The Complexity of the Linux 2.4 scheduler

It is known that the scheduler of the Linux kernel 2.4 shows 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 processes, as the measurements in fig.12 and 13 confirm.

This linear dependance 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:

```
list_for_each(tmp, &runqueue_head) {
      p = list_entry(tmp, ...)

      if (can_schedule(p, this_cpu)) {
         int weight = goodness(p,...);
         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.

4.2 The Complexity of the Linux 2.6 scheduler

The new scheduler of Linux 2.6 has been designed by Ingo Molnar, 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.12 and 13 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 bitfields to determine the process of highest priority.

Besides 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, often because they are waiting for user input from the mouse or keyboard f.ex. The scheduler tries to keep such processes in the 'active' queue, so that they 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 multi-processing (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.12 and 13 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.

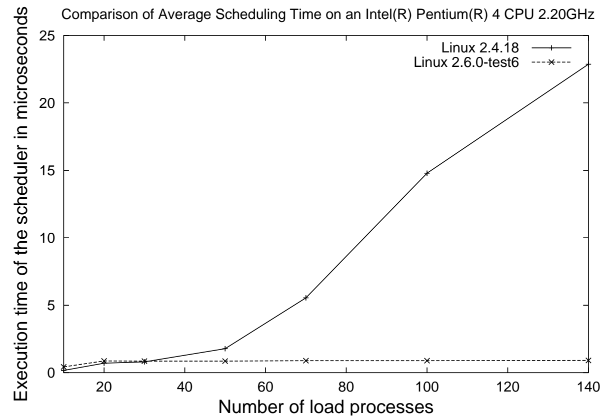

Figure 12: *Comparison of the* **average** *measured execution times of the scheduler of the Linux kernel 2.4.18 and 2.6.0-test6 versus the number of SCHED_NORMAL load processes ready to run*

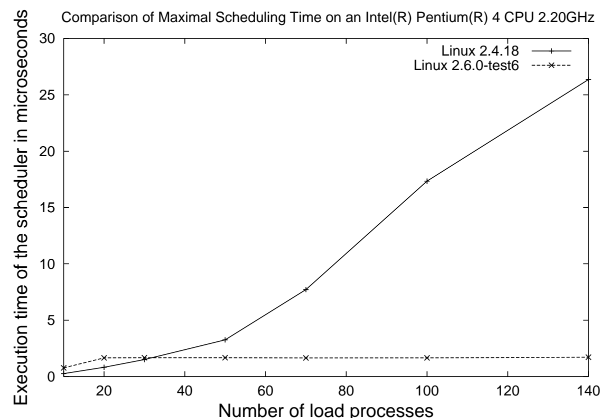

Figure 13: *Comparison of the* **maximum** *measured execution times of the scheduler of the Linux kernel 2.4.18 and 2.6.0-test6 versus the number of SCHED_NORMAL load processes ready to run*

5 Availability

The priority inheritance patch (PV-Patch) presented in this paper and the patch that transforms interrupt service routines into kernel threads (IKT-Patch) can be downloaded under the GPL license from our website [10].

6 Summary

The Linux kernel has made several big steps towards a better soft-realtime compatibility in the last years and some important improvements like the 'Preemptible kernel' [6] are now in Linux 2.6 available as a configuration option of the standard Linux kernel. Others are still only available as additional kernel patch like the 'High Resolution Timer patch' [1] or 'Rapid Reaction patch' [3] for a better timing resolution. In Linux 2.6 also the scheduler shows a constant execution time and the timer interrupt is executed ten times more often providing a more precise time base of about 1 or 2 milliseconds. So on the whole the standard Linux 2.6 kernel is better suited for meeting the requirements of soft-realtime tasks than Linux 2.4

Despite these improvements also the Preemptible Linux kernel 2.6 still contains long critical sections that can cause long PDLT latencies. This latencies can be f.ex. in the range of 1, 2 or 3 milliseconds on a Pentium 4, 2.2 GHz system with load. The upper bound is not known.

In this paper a further concept has been presented and analyzed that is used for ex. by the commercial Timesys Linux kernel [8] to increase the preemptability of the Linux kernel furthermore. A way to reach this goal is to change many of the remaining preemption locks - which are spinlocks in the SMP case - of the Preemptible Linux kernel into mutexes. For these mutexes a priority inheritance protocol should be implemented for soft-realtime reasons. We implemented such a protocol [9] under the GPL license [10], while the Timesys priority inheritance protocol [8] is only commercially available.

Furthermore to transform into mutexes also preemption locks that are used by interrupt service routines, a further patch -called IKT patch- is needed, that makes all interrupt service routines besides the timer interrupt executed by schedulable kernel threads. As a side effect these kernel threads executing interrupt handlers now can be prioritized by different priority levels of the soft-realtime scheduler SCHED_FIFO.

These GPL kernel patches, that can be downloaded from our website [10], alone lead to a higher IRT, since the interrupt handling kernel threads are now also delayed by preemption locks.

But if these concepts are used to convert most of the preemption locks of the Linux kernel into mutexes - as the Timesys kernel does - a better preemptabilityand soft-realtime capability can be reached than in the Preemptible Linux kernel as measurements of the Timesys kernel show, see fig.11. Since the kernel threads handling interrupts are scheduled, a scheduler with a constant execution time like the Timesys scheduler or the one of Linux 2.6 is very important. Nevertheless the need of scheduling before dealing with an interrupt leads to an IRT, that is some microseconds higher than in standard Linux, see fig.10.

Although from a theoretical point of view it could be shown that -because of the new latency KLT- this concept alone can't lead to hard realtime guarantees, it provides a much better statistical preemptablility of the Linux kernel which is an advantage when executing tasks with soft-realtime constraints.

References

[1] George Anzinger, 2001, High Resolution POSIX Timer (HRT Patch)

[2] Andrew Mortons Low Latency Patches, 2001, `www.zip.com.au/~akpm/linux/schedlat.html`

[3] A.Heursch, H.Rzehak, 2001, *Rapid Reaction Linux: Linux with low latency and high timing accuracy*, ALS, Oakland, California, `inf3-www.informatik.unibw-muenchen.de/research/linux/rrlinux/`

[4] A.Heursch, A.Horstkotte, H.Rzehak: *Preemption concepts, Rhealstone Benchmark and scheduler analysis of Linux 2.4*, RTEC, Milan, Nov. 2001, `http://inf33-www.informatik.unibw-muenchen.de/research/milan/measure_preempt.html`

[5] Robert Love's Kernel Patches, `www.tech9.net/rml/linux/`

[6] Nigel Gamble, 2001, Open Source Project 'kpreempt', `http://kpreempt.sourceforge.net/`

[7] M. Maechtel: *Entstehung von Latenzzeiten in Betriebssystemen und Methoden zur messtechnischen Erfassung*, ISBN 3-18-380808-0

[8] Timesys Linux GPL kernel: `www.timesys.com`

[9] Victor Yodaiken, *The dangers of priority inheritance*, `http://citeseer.nj.nec.com/cs`

[10] Mutex implementation with priority inheritance protocol, `http://inf3-www.informatik.unibw-muenchen.de/research/linux/mutex/`

[11] Sha L., Rjakumar R. and S.Sathaye, *Priority inheritance protocols: An approach to real-time synchronization*, IEEE Transactions on computers, 39:1175-1185, 1990