

STEPS TOWARDS A FULLY PREEMPTABLE LINUX KERNEL

Arnd C. Heursch* Dirk Grambow* Alexander Horstkotte*
Helmut Rzehak*

* Department of Computer Science, University of Federal Armed
Forces, Munich, Werner-Heisenberg-Weg 39, 85577 Neubiberg,
Germany, heursch@informatik.unibw-muenchen.de,
rz@informatik.unibw-muenchen.de

This paper has been published first on the
27th IFAC/IFIP/IEEE Workshop on Real-Time Programming,
WRTP'03, Lagow, Poland, May 14-17, 2003,
<http://www.iie.uz.zgora.pl/wrtp03/action.php>

Abstract: This paper investigates the real-time capabilities of the Linux kernel and special Linux real-time kernel patches to enhance the preemptability of the Linux kernel. While different latencies contributing to the Process Dispatch Latency Time (PDLT) are discussed, patches to reduce those latencies are examined in detail. *Copyright © 2003 IFAC*

Keywords: real time, performance analysis, jitter, critical areas, priority, inversion, real-time tasks

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.

2. THE PROCESS DISPATCH LATENCY TIME AND LINUX SCHEDULING POLICIES

The Process Dispatch Latency Time (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, see fig.1.

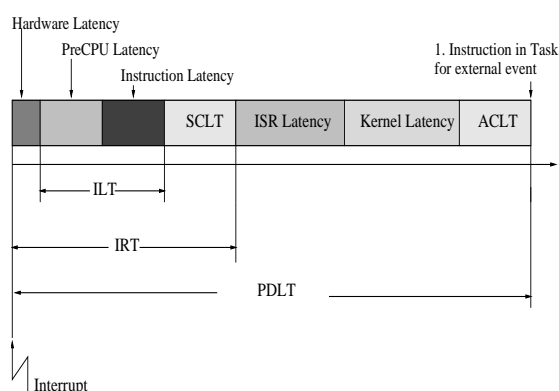


Fig. 1. The PDLT contains several latencies

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_OTHER, 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_OTHER processes.

- 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_OTHER process by the scheduler.

3. LATENCIES CAUSED BY THE LINUX KERNEL

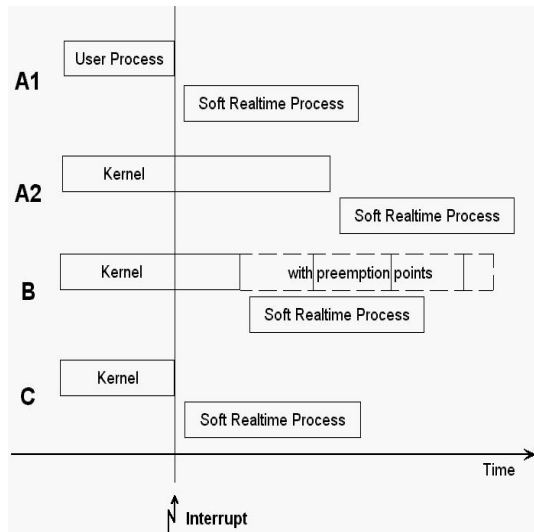


Fig. 2. Concepts of Preemption

Formula (1) shows the latencies, contributing to the PDLT of the soft-real-time process on a single processor system in standard Linux, which has the highest priority of all processes currently ready to run:

$$PDLT_{soft-real-time} = t_{IRT} + t_{ISR-latency} + t_{kernel-latency} + t_{scheduling} + \{t_{bottom-halves}\}_{Linux2.2} \quad (1)$$

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.1 shows, t_{IRT} can be divided into 2 parts, hardware caused latencies and a latency caused by interrupt-locks in the Linux kernel.

In the sample program the Interrupt Service Routine (ISR) awakens the soft-real-time process of the highest priority in the system. The duration of the ISR contributes to the term $t_{ISR-latency}$ in equation (1) and fig.1. Generally $t_{ISR-latency}$ contains the duration of all ISR's that are executed in between the interrupt and the first instruction of the soft-real-time process that shall be awakened. Different cases have to be distinguished:

- (1) If the processor is currently executing code in user mode, when the interrupt occurs (see line A1 in fig. 2), then the scheduler can be invoked

Table 1. Comparison of 2 LowLatency Patches

Functional sections of Linux as operating system	Number of Preemption Points (PP) in Low Latency Patches	
	I.Molnar 2.4.0-test6 Patch E2	A.Morton Linux 2.4.2
	process administration	10
memory management	48	31
file system	29	30
Input/Output	9	2
Total number of PP	96	66
conditional exit points in loops/functions	5	5

just after the interrupt has been processed by the CPU. In this case there is no kernel latency; $t_{kernel-latency}$ is zero in eq. (1) and fig.1. The time to preempt a process running in user-space takes only a few microseconds on current hardware (Heursch *et al.*, 2001), since there is only a scheduling and a context switch necessary, causing the Application Context switch Latency (ACLT) in fig.1.

- (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. Although the ISR wakes up a soft-real-time 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.2, the appropriate kernel latency is called $t_{kernel-latency}$ in eq.(1). Since Linux kernel code is not preemptable in general, also kernel threads cause kernel latencies. As several measurements (Morton, 2001; Wilshire, 2000; Williams, 2002) have shown, the longest kernel latencies produced by kernel code on current PC's are on the order of tens up to hundreds of milliseconds (Heursch and Rzehak, 2001). So, regarding Worst Case Latencies $t_{kernel-latency}$ is the most important term in eq. (1).

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. A soft-real-time process should protect itself from being swapped out by locking its address space into the RAM using `mlockall()`. Because of the implementation of the scheduler of the standard Linux kernel as one list, the term $t_{scheduling}$ depends linear on the number of processes ready to run on the system (Heursch *et al.*, 2001).

Since the terms or latencies of eq. (1) can be considered as largely independent from each other, the maximum PDLT value of eq. (1) is the sum of the maximums of each of its terms.

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 softirq's, i.e. soft interrupts, that are processed by a kernel daemon called 'ksoftirqd_CPUID'. So the softirq's/bottom-halves don't contribute to the PDLT of a soft-real-time 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 $PDLT_{Linux2.2}$.

In the following different kernel patches, that improve the soft real-time capabilities of the standard Linux kernel, are examined. In general nobody can guarantee the total correctness of any kernel patch as of the standard kernel itself, because nobody can test all execution paths in the kernel.

4. LOW LATENCY PATCHES

The main method of the 'Low Latency Patches' is to make system calls or kernel threads with a long execution time at least preemptable at certain points, see line B of fig. 2. Therefore Preemption Points are introduced into long kernel routines:

```
#define conditional_schedule()
do { if (current->need_resched) {
    current->state = TASK_RUNNING;
    schedule();
}
} while (0)
```

These so called 'Conditional' Preemption Points only take effect, when the kernel variable `current->need_resched` has been set, e.g. an interrupt service routine often awakens a SCHED_FIFO process and sets this variable to '1' in order to force a rescheduling as soon as possible.

A Preemption Point can be placed only at special points, where no data inconsistencies can occur due to read/write or write/write conflicts (Wang and Lin, 2000). Most of them are placed where long lists have to be changed, i.e. in the memory management or in drivers, see Table 1.

How much Preemption Points should be introduced? To analyze this question a method like this is introduced into a test kernel driver, located in a Linux kernel module:

```
for(int i=1; i<1200; i++)
    if(i % 600 ==0) conditional_schedule();
```

The $(i\%N)$ function, i modulo N , defines the Number of equidistant Preemption Points in the loop of the test kernel driver. Changing the N in the code above from 0 to 29 varies the number of Preemption Points and using a special Software Monitor (Maechtel and Rzehak, 1999; Heursch *et al.*, 2001) the remaining PDLT has been measured, see fig. 3.

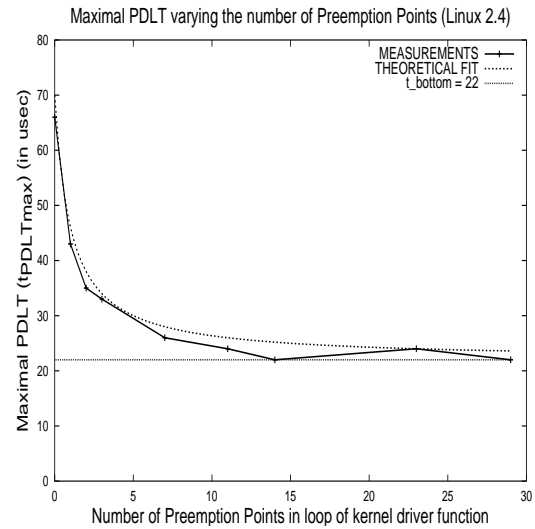


Fig. 3. Maximum PDLT dependent on the amount of equidistant Preemption Points

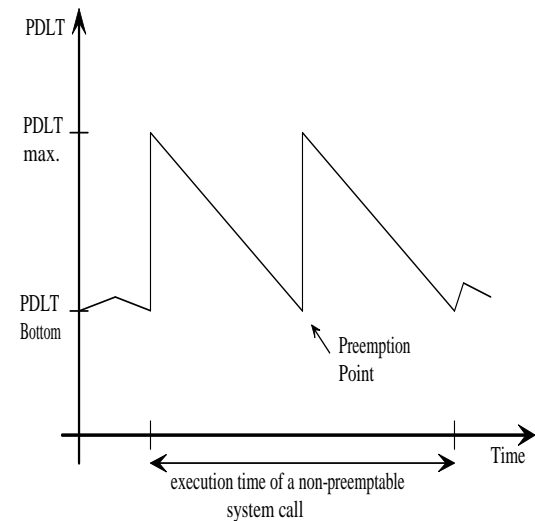


Fig. 4. Schematic illustration of the variables in formula (2)

Fig.3 and formula (2) show, the more Preemption Points (n_{pp}) are introduced the more the maximum PDLT ($tpDLT_{max}$) decreases, but the absolute value of the decrease gets smaller. With eleven Preemption Points and more in fig. 2 there is nearly no further decrease. The baseline of the PDLT t_{bottom} is nearly reached. In fig.3 the most reasonable amount of Preemption Points would be between seven and eleven. The Preemption Points are equally distributed regarding execution time. Instead of $t_{kernel-latency}$ one could also write $t_{kernel-latency}^{standard}$:

$$t_{PDLT_{max}} = \frac{t_{kernel-latency}}{n_{pp} + 1} + t_{bottom} \quad (2)$$

- $t_{kernel-latency}$:= execution time of the non-preemptable system function.
 $= PDLT_{max} - PDLT_{Bottom} = (66 - 22) \text{ usec} = 44\text{usec}$ in fig. 3 and 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.
- n_{pp} := number of Preemption Points introduced into the system function
- $n_{sections}$:= $n_{pp} + 1$ is the number of non-preemptable sections within the system function.
- $t_{PDLT_{max}}$:= value of the remaining maximum PDLT

From equation (2) one can conclude that for long Linux system calls, i.e. $t_{kernel-latency} \gg t_{bottom}$, 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.

5. THE PREEMPTIBLE KERNEL

5.1 Spinlocks in the standard Linux kernel

Spinlocks are the basic synchronization mechanism in the Linux kernel for SMP mode, for Symmetrical MultiProcessor systems. On a SMP system it's possible that currently every processor executes a system call. To avoid race conditions spinlocked regions have been introduced as critical sections into the Linux SMP kernel.

Spinlocks assure that only one process can enter a critical section at a time, even on a SMP system. This means they act like a mutex or a binary semaphore. But there is a difference compared to semaphores:

If a process tries to lock a spinlock, that is already locked by another process, the second process isn't put into a waiting queue to fall asleep as it would be when trying to enter a mutex as the second one. If a process tries to enter a spinlock as the second one, it will make a busy wait, 'spinning around', until the first process unlocks the spinlock.

Linux provides only one source code for SMP and uniprocessors, so the spinlocks are placed as a C-macro in the source code. If the standard kernel is compiled without the option SMP, i.e. for a uniprocessor, then the spinlock macros are all empty, so that there are no spinlocks on uniprocessor systems at all.

On single processor systems spinlocks are not necessary, because the processor normally can work on only one path in the kernel at a time, since the standard kernel is not preemptable.

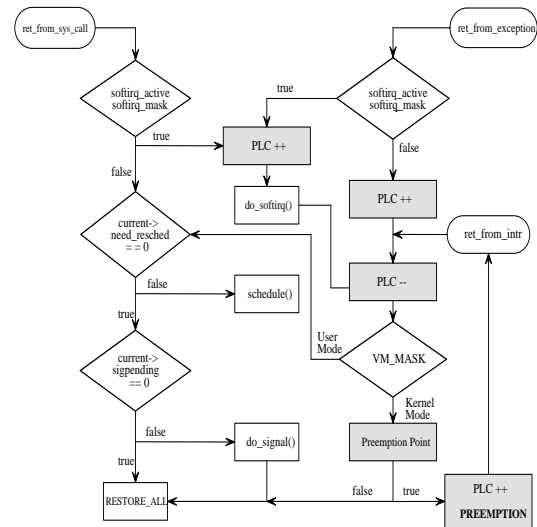


Fig. 5. Flowchart of task state transitions of the Preemptible Kernel for Linux 2.4.2

5.2 The Preemptible Kernel Patch

Another idea to reduce the term $t_{kernel-latency}$ in eq. (1) has been presented by embedded system vendor MontaVista (Inc., 2000) and is now named 'Preemptible Kernel' or 'Preemption Patch'. Today it is developed as the open source project 'kpreempt' ('kpreempt', 2002) and by Robert Love (Love, 2002). It's now a configuration option in the official Linux 2.5 development kernel.

This kernel patch has the intention to make Linux kernel code preemptable in general - see line C of fig.2 - in order to improve the responsiveness of the Linux kernel on single and multiprocessor systems.

The Preemptible Kernel takes advantage of the spinlocks that have been placed in the kernel since Linux 2.1 in order to make Linux running on SMP systems. The patch applies the following changes to the kernel:

- In order to avoid race conditions in the now Preemptible Linux Kernel the spinlocked regions are changed on uniprocessors into critical sections with a preemption lock by redefining the spinlock macro in the Linux kernel source code. Spinlocks itself on uniprocessors are impossible, because a uniprocessor isn't allowed to go into a busy wait, which it couldn't leave any more.
- 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. That way, all kernel functions and kernel threads are made preemptable, but there are some exceptions:
Preemption can't take place:
 - (1) while handling interrupts
 - (2) while processing Soft-IRQ's
 - (3) while holding a spinlock, write- or readlock.
 - (4) while the kernel is executing the scheduler

(5) while initializing a new process in the `fork()` system call

At all other times the algorithm allows pre-emption.

When the uniprocessor system enters one of the states shown above, e.g. a critical section, the global Preemption-Lock-Counter (PLC) is incremented, indicating that preemption is now forbidden. When leaving the critical section, the PLC is decremented and a test is made to see, whether preemption has been called for during holding the spinlock. If so, the current task is preempted directly after releasing the spinlock. This 'polling for preemption' at the end of a preemption-locked region can happen tens of thousands of times per second on an average system. Fig. 5 shows the flowchart of some assembler routines in the file 'entryS', that are invoked at nearly every task state transition. Fields with a gray background indicate a change of the Preemption Lock Counter (PLC). The figure shows that service routines for interrupts and soft-interrupts are handled as critical sections.

- Per-CPU variables have to be protected by a preemption lock.
- In order to improve the responsiveness of the kernel, the interrupt return code, i.e. assembly language code, is modified. That way now pre-emption is possible and tested 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.

For the Preemptible Kernel eq. (1) has to be modified. $t_{kernel-latency}$ is replaced by $t_{critical_sections}$. Since $t_{critical_sections}$ is in most cases much smaller than $t_{kernel-latency}$ in equation (1), the Preemptible Kernel improves the soft-real-time capabilities of standard linux. But there are still some very long critical sections - i.e. spinlocked sections in SMP-, where preemption is forbidden, so that $max(t_{critical_sections})$ might be not much smaller than $max(t_{kernel-latency})$ in equation (1) (Wilshire, 2000).

The Rhealstone Benchmark (Heursch *et al.*, 2001) showed that the Preemption Patch slightly reduces the performance of the Linux kernel on uniprocessors. This is due to the fact, that the Preemptible Kernel often executes additional code, like the gray fields shown in fig. 5, and some code that has been used before only in the SMP case.

5.3 Reducing latencies caused by long held spinlocks

'Long held' preemption locks - or spinlocks in the SMP case - are locks held long compared to the duration of 2 context switches. There are two possibilities to cope with these latencies in an already Preemptible Kernel:

Combining patches

The first possibility is to combine the Preemptible Kernel Patch with a Low Latency Patch. The Low Latency Patches also contain Preemption Points breaking spinlocks: In order not to crash the kernel the spinlock has first to be unlocked and after scheduling, when the process comes to execution again, the spinlock has to be locked again. This results in more fine grained spinlocks.

MontaVista combined the Preemption Patch for Linux 2.4.2 with parts of Andrew Morton's LowLatency patch, today Robert Love offers such patches, too (Love, 2002). A combination of both patches at own risk has been shown to reduce latencies even more efficient than one of the patches used alone (Williams, 2002). From a theoretical point of view this is easy to explain. The Preemption Points of the Low Latency Patch reduce efficiently special known long latencies, partly those in spinlocks, while the Preemption Patch makes preemption possible in general, i.e. also in new added drivers without Preemption Points, if they don't currently hold spinlocks. Even an agreement between the patch maintainers to combine both patches to one new official patch has been reported.

Changing long held spinlocks into mutexes

The second possibility is changing long held pre-emption locks/spinlocks into mutexes in the Pre-emptible Kernel, first discussed for Linux in the 'kpreempt' project ('kpreempt', 2002). The advantage of mutex locks is that global locks like pre-emption locks are replaced by mutual exclusion locks to resources. This replacement does not decrease the term $max(t_{critical_sections})$, which replaces the term $max(t_{kernel-latency})$ in equation (1) in any preemptible kernel, but it reduces its frequency of occurrence. In the SMP case additionally another process entering a mutual exclusion as the second one on another CPU is put asleep instead of blocking this CPU as a spinlock would do. This leads to a better average preemptibility.

It's possible to replace all those spinlocks by mutexes, which are never entered by any code from an interrupt service routine (ISR) or a softIRQ/Bottom Half. But it's not possible to replace the spinlocks which at the same time also lock interrupts, called `spin_lock_irqsave()` in the standard Linux kernel. The reason 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 accessing an already locked spinlock on a SMP system.

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 processes using the same kernel mutex in perhaps different kernel routines and other processes with priorities in between them. Since the priority ceiling protocol is only usable for processes with priorities

which can be statically analyzed, and the calling of Linux system functions can't be previewed, mutexes with a priority inheritance (PI) protocol have to be implemented in order to avoid the starvation of highly prioritized processes by less important ones. Furthermore it's important that the Linux kernel programmer rule to access spinlocks always in the same sequence, determined by their addresses, is extended to mutexes, too, in order not to cause deadlocks.

In (Grambow, 2002) two different PI-protocols have been implemented. The first simple PI-protocol should work fine, when a kernel routine accesses only one mutex at a time. Another PI-protocol described by Yodaiken (2001) allows also access to nested mutexes, but this causes some overhead compared to the spinlocks in the standard Linux kernel.

Apart from kernel mutexes such PI-protocols can also be useful to applications that use mutexes in user space and want to avoid priority inversion. The implementation is available under GPL (Grambow, 2002).

The Timesys approach - an implementation

Timesys (Timesys, 2001) adopted the Preemptible Kernel and implemented some additional changes in the standard kernel, released partly under GPL/LGPL and partly under a proprietary license:

- They changed all Interrupt Service Routines (ISR) and Soft IRQ's /Bottom Halves into kernel threads and extended the fixed priority soft real-time scheduler to schedule them, normally at its new highest priority 509. So with this Timesys Linux kernel it's possible to assign priorities to interrupts or even prioritize a real-time process higher than some interrupts.
- 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. Of course to protect very small critical regions in the Timesys Linux kernel, even this kernel contains non-preemptible combined spinlock interrupt locks, called `old_spin_lock_irqsave`, but these are held only for very short times, e.g. to do atomic operations on the Timesys mutexes.
- Such a mutex implementation needs a priority inheritance (PI) protocol at least for real-time 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 available.

6. SUMMARY

While approaches to increase the preemptability of the Linux kernel like the Low Latency Patches and the Preemptible Kernel Patch have already shown to decrease latencies successfully, a combination of both techniques promises the best results. A further approach that replaces preemption locks or spinlocks by mutexes only reduces the frequency latencies of the remaining long critical sections occur. Such mutexes are in need of a complex priority inheritance protocol to avoid unbounded priority inversion, i.e. possible starvation of soft-real-time tasks. Furthermore for this approach to make sense all interrupt service routines must be transformed into kernel threads. So the alternative idea of the Low Latency Patches to divide long critical sections into several shorter ones still remains a promising approach.

7. REFERENCES

- Grambow, Dirk (2002). Untersuchung zur verbesserung der preemptivitaet aktueller linux kernel. <http://inf3-www.informatik.unibw-muenchen.de/research/linux/mutex/mutex.html>. diploma thesis INF ID 17/2002.
- Heursch, Arnd, Alexander Horstkotte and Helmut Rzehak (2001). Preemption concepts, rhesstone benchmark and scheduler analysis of linux 2.4. In: *Conf. Proc. Real-Time & Embedded Computing Conference, Milan*. http://inf3-www.informatik.unibw-muenchen.de/-research/linux/milan/measure_preempt.html.
- Heursch, Arnd and Helmut Rzehak (2001). Rapid reaction linux: Linux with low latency and high timing accuracy. In: *Conf. Proc. of ALS, Oakland, California*. <http://inf3-www.informatik.unibw-muenchen.de/research/linux/rrlinux/rapid.html>.
- Inc., MontaVista (2000). Linux preemption patch. <http://www.montavista.com>.
- 'kpreempt', Open Source Project (2002). Preemptible kernel. <http://kpreempt.sourceforge.net/>.
- Love, Robert (2002). Preemptible kernel patches. <http://www.tech9.net/rml/linux/>.
- Maechtel, Michael and Helmut Rzehak (1999). Real-time operating systems on the test-bench. In: *Proc. Joint IFAC/IFIP Workshop on Real-Time Programming and Internat. Workshop on Active and Real-Time Database Systems*.
- Morton, Andrew (2001). Low latency patches. <http://www.zip.com.au/~akpm/linux/schedlat.html>.
- Timesys (2001). <http://www.timesys.com>.
- Wang, Yu-Chung and Kwei-Jay Lin (2000). Some discussion on the low latency patch for linux. In: *Conf. Proc. of the Workshop on Real Time Operating Systems and Applications, Florida*. <http://www.thinkingnerds.com/projects/rtos-ww/presentations.html>.

- Williams, Clark, Red Hat Inc. (2002). Linux scheduler latency. *Embedded Systems (Europe)*.
<http://www.allembedded.com>.
- Wilshire, Phil (2000). Real-time linux: Testing and evaluation. same conference as see Wang, Yu-Chung, (2000).
- Yodaiken, Victor (2001). The dangers of priority inheritance.
citeseer.nj.nec.com/yodaiken01dangers.html.