Fast User Space Priority Switching

Alexander Zuepke, Marc Bommert, Robert Kaiser
RheinMain University of Applied Sciences, Wiesbaden, Germany
Email: {alexander.zuepke|marc.bommert|robert.kaiser}@hs-rm.de

***Abstract*—This paper presents fast user space priority switching, a mechanism for tasks to change their scheduling priority without entering the operating system kernel. Instead, tasks and the operating system kernel agree on a shared memory space storing the current task's priority. While the task changes its priority by writing to a variable in the shared memory, the operating system kernel synchronizes its internal scheduling priority with the user task's priority lazily on certain occasions affecting scheduling.**

We discuss two different protocols for fast user space priority switching. For two ARM-based platforms, we compare their implementations with a traditional approach which uses system calls to change scheduling priorities. The presented approach is suitable for systems using partitioned preemptive fixed-priority scheduling.

I. Introduction

Operating system environments, like OSEK[1] and AUTOSAR[2] in the automotive world, ARINC 653[3] in Avionics, and POSIX[4] real-time scheduling for industrial applications, often rely on preemptive fixed-priority scheduling, either on single processor systems or in a partitioned environment comprising a single processor per partition. The system's scheduled active entities, called tasks, processes, or threads, switch their scheduling priorities when entering a critical section, according to the operating system's *priority ceiling protocol* [5] to prevent deadlocks. Using precomputed priorities based on previous analysis, the tasks raise their priorities upon entering a critical section to a specific value, and lower their priorities again to the previous value when leaving the critical section. Moreover, critical sections can be acquired in a nested fashion. Typically, on occasions where priorities are changed, other important things happen as well, so most operating systems use a system call to implement the state changes in the operating system kernel. However, system calls are expensive operations on most processor architectures and in most operating systems, involving a state change of processor privileges, saving and restoring registers, or switching to different stacks.

This paper presents a concept to change task priorities in user space with low cost by avoiding system calls entirely in the common fast path. To this end, we define a shared memory protocol between user space tasks and the operating system kernel.

The rest of this paper is structured as follows: We introduce terminology, the execution environment, and the problem scenario in section II. In section III, we discuss our optimized approaches in detail. Section IV compares the approaches with a traditional implementation using system calls to change priorities on two ARM-based platforms for automotive and industrial / multimedia usage scenarios. We discuss the results in section V. Finally, section VI lists related work and section VII concludes.

II. Problem Definition

We use the following terminology and make the following assumptions: The term *task* refers to the unit of execution scheduled by an operating system. The operating system provides two execution modes: either *user mode* for application contexts, or *kernel mode* for operating system specific activities. Additionally, the distinction of *user space* and *kernel space* denotes separated address spaces for the execution modes, e.g. user code can not access kernel code and data, and kernel code must validate user space pointers before access.

For task scheduling, we assume an execution environment using *partitioned preemptive fixed-priority scheduling*, i.e. independent scheduling on each processor. The higher a priority value, the more favoured a task is selected by the operating system scheduler. For tasks of the same priority, we assume activation in *FIFO-order* based on task arrival times.

Each task τ_i has a current *scheduling priority* $P_i(t)$ and an upper priority bound, the *maximum controlled priority* $Pmax_i$ up to which it can adjust its own priority during task execution time. For brevity, we shorten $P_i(t)$ to P_i.

We assume that a *priority ceiling protocol* is used, such that each *critical section* CS_m has a dedicated ceiling priority $Pceil_m$. This ceiling priority is statically defined as the maximum scheduling priority of all tasks that compete for that specific critical section CS_m.

We further assume that all tasks comply with the priority ceiling protocol, i.e. that no task tries to enter a critical section with a priority lower than the ceiling priority $Pceil_m$ of the section. Conversely, the maximum controlled priority of all tasks that compete for a specific critical section CS_m is higher or equal to the ceiling priority of this critical section $Pceil_m$. That is: $\forall \tau \forall m : Pceil_m \leq Pmax_i$.

As illustrated in Figure 1, on entry into a critical section at t_{enter}, task τ_i raises its priority from its current scheduling priority P_i to $P_i' := max(P_i, Pceil_m)$. On exit of that critical section at t_{leave}, it lowers its priority back to its previously possessed priority P_i. Moreover, critical sections can be entered in a nested fashion.

As Figure 1 shows, in a scenario where task τ_i raised its priority and exclusively performs resource access, while another task τ_j arrives at t_{arrive}, three possible priority relations are to be distinguished:

[1]Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug / Vehicle Distributed Executive [1]
[2]AUTomotive Open System Architecture [2]
[3]Avionics Application Standard Software Interface [3]
[4]Portable Operating System Interface [4]

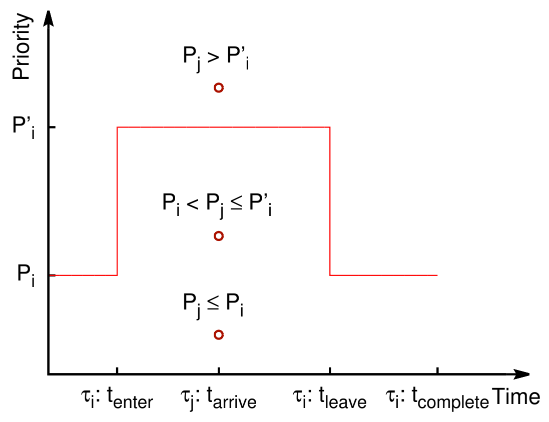

Fig. 1: Cases of interaction: τ_j arrives, while τ_i raised its priority temporarily from P_i to P_i' during a critical section.

- Case $P_j > P_i'$: The priority of τ_j is higher than the raised priority of τ_i. The operating system kernel thus immediately preempts τ_j over τ_i. τ_i will effectively be preempted while still being in the critical section. By definition of the priority ceiling protocol, all tasks with higher priority than P_i', which includes τ_j, do not access the critical section τ_i was preempted in.

- Case $P_i < P_j \leq P_i'$: The priority of τ_j is higher than the scheduling priority P_i initially possessed by τ_i, but is lower than or equal to its raised priority P_i'. This case has to be considered by the operating system kernel since, as soon as τ_i lowers its priority back to P_i, it needs to be preempted immediately in favour of τ_j.

- Case $P_j \leq P_i$: The priority of τ_j is lower than or equal to the initial scheduling priority P_i of τ_i. Here, τ_j can not preempt τ_i regardless of τ_i's current priority. In fact, τ_j will not be selected as currently scheduled task before τ_i completes at $t_{complete}$.

In the scenario depicted in Figure 1, a traditional implementation approach would use two system calls to raise and lower a task's scheduling priority or disable interrupts during execution of the critical section. Due to the low overhead of interrupt disable/enable pairs compared to system calls, such a pragmatic approach is often considered acceptable if the execution environment permits the use of privileged instructions by application code, such as in Automotive. However, in POSIX and ARINC 653, disabling interrupts is not allowed.

To lower the cost of priority ceiling protocols, we consider the following optimization: if no task arrives while task τ_i is in a critical section, the system calls could be omitted. This could be implemented by a *lazy* approach: the step to adjust priorities is delayed to the point in time when a scheduling specific event occurs, e.g. until a new task arrives.

Two scenarios are possible: either the kernel tracks all critical sections τ_i may enter, or it tracks P_i. We consider the latter: τ_i stores its current scheduling priority in the user space variable `uprio` which is known and accessible to the operating system kernel, and the kernel synchronizes τ_i's actual scheduling priority with this value when necessary.

III. Fast User Space Priority Switching Protocols

In this section, we discuss two different protocols for fast user space priority switching, referred to as UPRIO|KPRIO and UPRIO|NPRIO, according to the names of the protocol variables.

A. *The* UPRIO|KPRIO *Protocol*

The UPRIO|KPRIO protocol comprises two variables:

- `uprio` describes the priority of the current task τ_i in user space, *user prio*, and is set by the task on priority change, i.e. when entering or leaving a critical section.
- `kprio`, shorthand for *kernel prio*, reflects the kernel's view of the scheduling priority of τ_i. On kernel entry[5], the kernel updates P_i from `uprio` and writes the resulting priority back to `kprio`.

Using this protocol, τ_i can raise its scheduling priority by issuing single memory store operation only:

```
function Protocol1_RaisePriority(prio)
    uprio := prio
```

To handle preemption correctly when lowering the priority again, τ_i needs to check (by looking at `kprio`) whether it has been interrupted since it entered the critical section and, if so, to issue a system call to trigger rescheduling in the kernel:

```
function Protocol1_LowerPriority(prio)
    uprio := prio
    if uprio < kprio
        SyscallPreempt()
```

The kernel part of the protocol synchronizes the internal scheduling priority on kernel entry:

```
function Protocol1_KernelSyncPriority()
    if uprio > currentTask.maxprio
        uprio := currentTask.maxprio
    kprio := uprio
    if kprio < nextTask.prio
        KernelPreempt()
```

For robustness, the kernel ensures that τ_i scheduling priority does not exceed the task's maximum controlled priority $Pmax_i$. Furthermore, it checks for pending task preemption.

Using this protocol, τ_i can raise and lower its scheduling priority in the fast path by issuing one memory load and two memory store operations only. In the slow path, i.e. when interrupted by the kernel in the critical section, one system call is inevitable.

In a nested critical section scenario, τ_i needs to raise its priority twice: from P_i to P_i' on entering the outer critical section CS_m, and finally to P_i'' on entering the inner critical section CS_n. Using the described protocol, again no system call is involved in this step. The number of system calls on lowering the priority ranges between zero and two: zero for no interruption in the fast path, one for an interruption in the outer

[5]The kernel is entered by a processor architecture specific *trap* mechanism on system calls, asynchronous interrupts, or exceptions like division by zero.

critical section, and two for an interruption in the inner one. In the latter case, the protocol exhibits its worst case behavior: with each transition to a lower priority level, it has to issue a system call, because kprio > uprio.

B. *The* UPRIO|NPRIO *Protocol*

The second protocol also uses two variables, uprio and nprio, but with slightly different semantics:

- uprio describes the current priority of τ_i set by user code.
- nprio, meaning *next priority*, refers to the priority of the next eligible task for scheduling in FIFO and priority order. The kernel provides this information and updates it on every scheduling decision.

Like the first protocol, UPRIO|NPRIO allows rapid priority changes with no system calls in the fast path as long as the next thread's priority remains below P_i. If uprio drops below nprio on lowering the priority, user code issues a system call:

```
function Protocol2_RaisePriority(prio)
    uprio := prio

function Protocol2_LowerPriority(prio)
    uprio := prio
    if uprio < nprio
        SyscallPreempt()
```

The kernel updates nprio accordingly when tasks are inserted to or removed from the ready queue. Additionally, it bounds uprio to $Pmax_i$ and checks whether to preempt:

```
function Protocol2_KernelReadyQueueChange()
    ...
    nprio := nextTask.prio
    ...
    if uprio > currentTask.maxprio
        uprio := currentTask.maxprio
    if uprio < nprio
        KernelPreempt()
```

Despite the similarities, UPRIO|NPRIO only needs a system call if preemption is really required. Therefore, in the nested critical section scenario from above, the number of system calls to lower the priority is at most one.

C. *Protocol Summary*

When comparing UPRIO|KPRIO and UPRIO|NPRIO, both protocols use the same technique to change priorities by writing to a variable in user space, uprio, and checking another variable, kprio or nprio, when lowering the priority to test for (possible) preemption. While UPRIO|KPRIO uses the priority of the currently scheduled thread for that, UPRIO|NPRIO relies on the actual priority of the next eligible thread instead.

Despite being faster than an implementation using system calls to raise *and* lower priorities, implementations of both protocols add overhead to ready queue handling (synchronizing uprio) or context switching (setting uprio and kprio/nprio).

Still, UPRIO|KPRIO bears potential for optimization: instead of synchronizing kprio on every kernel entry, the synchronizing step could be postponed to actual scheduling decisions like in the UPRIO|NPRIO protocol.

IV. EVALUATION

We compare implementations of the approaches presented in section III to a traditional implementation using system calls to change priorities. To evaluate the implementations, we selected two ARM-based platforms from different use cases, but with a similar system architecture.

As a typical representative of an automotive processor, we selected the Hercules TMS570 evaluation board from Texas Instruments. The TMS570 has two ARM Cortex-R4 processor cores operating in lockstep mode at 180 MHz. This Cortex-R4 implementation does not have any caches, but fast, tightly-coupled on-chip SRAM for data storage and flash memory for instruction storage. Also, it supports a memory protection unit (MPU) to isolate applications. On the other end of the spectrum, the AM3358 processor on the BeagleBone Black board represents a system typically used in industrial and multimedia scenarios. Its Cortex-A8 core has split data and instruction caches of 32KB size each, a memory management unit (MMU), and it operates at 550 MHz.

Despite these differences, both processors share the same instruction set architecture and most of the exception handling model. To exclude side effects by memory management, we use a static memory layout on both processors and execute the same benchmark on a small statically configured OSEK-like operating system, based on a custom micro kernel.

We compare the two approaches UPRIO|KPRIO and UPRIO|NPRIO to each other and to an implementation using system calls for each priority change. We use two scenarios for evaluation: Firstly, in subsection A, we provide the execution times of a micro benchmark of all three approaches to determine the overhead of the fast priority switching implementation in subsection B. Secondly, in subsection C, we evaluate the benefit of fast priority switching in a nested locking scenario with and without preemption.

A. *Micro Benchmarks*

The micro benchmark shown in table II comprises multiple functions to evaluate the platform performance in general and

TABLE I: Processor Characteristics

Parameter	**TMS570**	**AM3359**
Typical applications	safety critical transportation applications	industrial automation, consumer electronics
CPU Core	Cortex-R4	Cortex-A8
Micro architecture	ARMv7-R	ARMv7-A
Pipeline	in order, dual-issue 8 stages, 1 ALU	in order, dual-issue 13 stages, 2 ALUs
L1 Caches	none none	2x 32 KB, 4-way 16 word line
L2 Cache	none	256 KB (not used)
Board	TI Hercules board with TMS570LS3137	BeagleBone Black rev. 1 with AM3359AZCZ
CPU Clock	180 MHz	550 MHz
SRAM Clock	180 MHz	275 MHz
Instruction fetch	Flash, 180 MHz	I-Cache, 550 MHz

basic scheduling related activities of an OSEK-like execution environment in particular. In braces, we give the relative performance gain over the traditional system call approach.

To analyze the platform performance, we conducted the following tests:

- NOP (No OPeration) loops to analyze the overhead for decrement-and-branch instructions,
- function calls followed by an immediate return, and
- memory performance of load and store operations.

To evaluate our operating system and its scheduling overhead, we measured the execution time of the following combinations of system calls:

- A *null system call* determines the overhead of system calls in general.
- `Schedule` enforces a round-trip through the scheduler without a context switch. The call puts the current task in `READY` state on the ready queue and reschedules it immediately.
- A `ChainTask` call where the calling task activates itself again: in addition to `Schedule()`, this shows the overhead of resetting the task's state.
- The `ActivateTask` / `TerminateTask` pairs shows task activation of a lower and higher priority task. Activation of the higher priority task causes scheduling and a context switch to this newly activated task, which immediately terminates, followed by another context switch back to the original caller. Activation of the lower priority task just measures the cost of placing a task on the ready queue.
- An event loop: a high priority task waits for incoming events which are signalled by a low priority task in a loop. The high priority task issues two system calls: `WaitEvent` to wait for events in the operating system kernel, and `ClearEvent` to acknowledge the events. This again comprises two context switches.

TABLE II: Micro Benchmark Results in CPU Cycles

Benchmark	System Call	UPRIO|KPRIO	UPRIO|NPRIO
TMS570			
NOP	7.5		
function call	22		
read 1K flash, 32-bit	2474		
read 1K SRAM, 32-bit	2078		
write 1K SRAM, 32-bit	2078		
null system call	157	185 (+17.8%)	157 (+0%)
`Schedule`	329	351 (+6.7%)	359 (+9.1%)
`ChainTask`	416	433 (+4.1%)	443 (+6.5%)
task activation low priority	295	320 (+8.5%)	312 (+5.8%)
task activation high priority	983	1060 (+7.8%)	1060 (+7.8%)
event loop	1166	1262 (+8.2%)	1224 (+5.0%)
AM3359			
NOP	2		
function call	7		
read 1K flash, 32-bit	786		
read 1K SRAM, 32-bit	786		
write 1K SRAM, 32-bit	786		
null system call	144	167 (+16.0%)	144 (+0%)
`Schedule`	251	283 (+12.7%)	279 (+11.2%)
`ChainTask`	295	327 (+10.8%)	324 (+9.8%)
task activation low priority	244	267 (+9.4%)	280 (+14.8%)
task activation high priority	710	776 (+9.3%)	752 (+5.9%)
event loop	917	1031 (+12.4%)	958 (+4.5%)

All tests were run in a loop 16384 times. For measurement, we used the 32-bit cycle counter of the ARM processor's performance monitor unit, which can be read in a single instruction. The cycle counter is configured to run at the processor's core clock speed.

The overall size of the operating system kernel and the benchmarks on the TMS570 board is 26 KB code and 25 KB data. The same values for the AM3359 are only slightly higher, so the overall working set fits into each of the processor's data and instruction caches.

We used the GCC 4.6.3 cross compiler for ARM provided by the Ubuntu 12.04 Linux distribution to compile our C99-based operating system. We let the compiler optimize for size with `-Os -fomit-frame-pointer` and used inline functions where possible.

B. Overhead of Fast Priority Switching Protocols

As table II shows, the Cortex-A8 core of the AM3359 possesses a faster micro architecture and shows roughly 2.5x faster memory performance than the TMS570. Also, the different protocols have no impact on platform performance. When comparing the performance of the system calls, only a performance benefit of 20% to 30% remains.

The UPRIO|KPRIO protocol shows a general overhead of up to 17.8% on the TMS and up to 16.0% on the AM3359. The UPRIO|NPRIO protocol performs better with an overhead of up to 9.1% on the TMS and up to 14.8% on the AM3359. Especially the decision to synchronize user priorities only on scheduling decisions pays off for null system calls.

The operating system kernel is not further optimized for any of the protocols. All three compared implementations use a defined region per task to host the protocol variables[6]. The shared region alone causes an overhead of 2% to 3% on context switches (values not shown).

C. Nested Locking Scenario with Preemption

To determine the overhead of critical sections and measure the effect of possible preemption on lowering a task's priority, the critical section benchmark defines three points in time when to activate another task:

T1	outside the outer critical section
T2	inside the outer, outside the inner critical section
T3	inside the inner critical section

At these trigger points, the benchmark task activates one out of four tasks whose priority value may be:

A) above the inner critical section's ceiling priority,
B) between the outer and the inner critical sections' ceiling priorities,
C) between the benchmark task's normal priority and the outer critical section's ceiling priority, or
D) lower than the benchmark task's normal priority.

Alternatively, no additional task is activated. In total, this provides 15 different procedures, of which we consider only a meaningful subset.

[6]The region between kernel and user task additionally hosts the ID of the currently executing task, which is the only value the system call variant updates

Similarly, we benchmark a non-nested critical section. We again interrupt the benchmark task outside and inside the critical section by none, another higher, medium, or lower priority task. This additionally provides 8 different procedures in total, of which we also consider only a meaningful subset. For the non-nested case, the single critical section is referred to as the outer critical section.

Table III shows the timing of nested and non-nested critical sections in terms of CPU cycles for both platforms including possible interruptions. In braces, we give the relative performance gain over the common system call approach. For both, the nested and non-nested cases, our protocols show good results when the critical sections are not interrupted, achieving a performance gain of more than 72% compared to the system call approach. Naturally, if the measured task is subject to interruptions, its execution time increases.

For interrupted critical sections, table III lists three typical combinations: *(+ highprio task)*, where a higher priority task causes immediate preemption, *(+ lowprio task)*, where a lower priority task does not cause preemption, and *(+ medprio task)* for scenarios where a medium priority task causes preemption as soon as a critical section is left. Where it makes a difference when the benchmark task is interrupted, i.e. in the outer or the inner critical section, the numbers are presented. Especially the Uprio|Kprio protocol is sensitive to this.

TABLE III: Critical Section Benchmark in CPU Cycles

Benchmark	**System Call**	**Uprio|Kprio**		**Uprio|Nprio**
		TMS570		
non-nested	370	T*	101 (-72.7%)	101 (-72.7%)
+ lowprio task	670	T1	437 (-34.8%)	424 (-36.7%)
		T2	659 (-1.6%)	
+ medprio task	1321	T1	1148 (-13.1%)	1136 (-14.0%)
		T2	1382 (+4.6%)	1299 (-1.7%)
+ highprio task	1340	T1	1157 (-13.7%)	1140 (-14.9%)
		T2	1400 (+4.5%)	
nested	747	T*	184 (-75.4%)	184 (-75.4%)
+ lowprio task	1014	T1	504 (-50.3%)	482 (-52.5%)
		T2	730 (-28.0%)	
		T3	947 (-6.6%)	
+ medprio task	1707	T1	1216 (-28.8%)	1197 (-29.9%)
		T2	1455 (-14.8%)	1367 (-19.9%)
		T3	1668 (-2.3%)	
+ highprio task	1694	T1	1223 (-27.8%)	1206 (-28.8%)
		T2	1450 (-14.4%)	
		T3	1669 (-1.5%)	
		AM3359		
non-nested	327	T*	28 (-91.4%)	28 (-91.4%)
+ lowprio task	581	T1	296 (-49.1%)	276 (-52.5%)
		T2	506 (-12.9%)	
+ medprio task	1046	T1	839 (-19.8%)	783 (-25.1%)
		T2	1040 (-0.6%)	935 (-10.6%)
+ highprio task	1061	T1	822 (-22.5%)	787 (-25.8%)
		T2	1046 (-1.4%)	
nested	651	T*	58 (-91.1%)	58 (-91.1%)
+ lowprio task	901	T1	313 (-65.3%)	293 (-67.5%)
		T2	531 (-41.1%)	
		T3	736 (-18.3%)	
+ medprio task	1376	T1	840 (-39.0%)	804 (-41.6%)
		T2	1058 (-23.1%)	957 (-30.5%)
		T3	1275 (-7.3%)	
+ highprio task	1382	T1	846 (-38.8%)	804 (-41.8%)
		T2	1076 (-22.1%)	
		T3	1275 (-7.7%)	

For Uprio|Kprio, activations inside the inner critical section (T3) are most expensive. These entail two system calls to synchronize the task's priority, whereas activations in the outer critical section (T2) just need one system call. For the TMS570, the T2-scenarios perform worse than the standard system call based approach by up to 4.6% in case of the kernel preempting the current task in favour of the interrupting task. As the number of system calls is exactly one in both the standard system call approach and our protocols, this is solely due to protocol overhead.

In contrast, the Uprio|Nprio protocol does not show these effects. As it issues at most one system call (zero for non-nested T2-scenarios) to synchronize its priority with the kernel, its results are always faster than the system call approach.

V. Discussion

In the benchmark results, both protocols show an overhead in system call performance compared to a traditional approach. This overhead needs to be justified by the performance gain of handling critical sections in user space in the fast path. We think the results draw a realistic picture of the protocols, as neither benchmarked approach was specifically optimized.

A. Benchmark Results

For Uprio|Kprio, we expected a constant overhead for each system call and only little impact on scheduling related calls. The former is an effect of synchronizing priorities on every system call, and the latter shows the overhead of updating the protocol variables. We also expected to see a *staircase* pattern due to additional system calls on lowering priorities when interrupting the protocol inside critical sections; the overheads are higher the higher the nesting level is.

However, we did not expect that Uprio|Kprio would perform so poorly and sometimes even shows slower performance than the system call based approach. We also benchmarked an implementation of the Uprio|Kprio protocol which does not synchronize priorities on every system call, but only on scheduling conditions (benchmark results not shown). The constant overhead for every system call disappears, but the worst case condition of a system call when lowering the priority still dominates the performance.

We also benchmarked the cost of updating user variables in a system call based approach to get an estimate of the costs (values not shown). We compared an implementation providing `uprio` and the ID of the current task in user accessible variables, an implementation providing just the ID, and one providing no variables. The overhead for providing the first variable (task ID) was about 4%, the additional overhead for the second variable (`uprio`) was less than 2% on a context switch, and about 4% for calls affecting the task's priority. These results are in line with the micro benchmarks.

The Uprio|Nprio protocol behaves as expected. The benchmark values show that it needs at most one system call when lowering the priority, and also the overall performance gain looks promising, especially on a system with caches. However, the benchmark results of a non-nested critical section interrupted by a medium priority task show, that the worst case timing is still in the range of the pure system call based approach. We assume this effect has prevented the adoption of such protocols in general purpose operating system.

Comparison of the results for TMS570 and AM3359 shows, that the performance gain of the protocols is higher by about 10% on the architecture with caches. We would have expected to see that the handling of the additional protocol variables would show much less impact on a CPU with caches, but the micro benchmark results point into a different direction.

At this point, it also becomes clear that further analysis of the benefits of the protocols is difficult using just these synthetic benchmarks. We need to run real-world workloads or require statistical information on the distribution of nested and non-nested locking and typical preemption patterns to see the overall effect on a long-running system.

As said before, a pragmatic approach in Automotive is to disable interrupts at the beginning of critical sections. This is probably related to higher costs of following the OSEK priority ceiling protocol compared to disabling interrupts. We need to compare the proposed protocols to these pragmatic implementations as well, as upcoming automotive platforms may no longer allow to disable interrupts in user space due to increased function safety requirements.

B. Safety and Security Considerations

From a safety point of view, the following aspects are relevant. A task τ_i can try to exceed its maximum controlled priority $Pmax_i$ by placing a higher priority value into `uprio`. The kernel must check this whenever it reads `uprio` and must bound the value to $Pmax_i$. Additionally, it is possible to enforce a lower priority bound $Pmin_i$ in an implementation, should that be a requirement. Lastly, a task can act as a foul player and not issue a system call on lowering the priority. This behavior has the same effect as a thread not leaving the critical section, because it delays the scheduling of higher priority tasks. This problem is not introduced by the fast priority switching approach: it would also happen with the traditional approach using system calls. Tasks accessing the same resource must mutually trust each other anyway.

Also, both protocols leak scheduling related information: UPRIO|KPRIO reveals that the user task has been interrupted, and UPRIO|NPRIO exposes the priority of the next eligible task for scheduling on the ready queue. This may hinder the adoption in security sensitive operation environments.

VI. Related Work

In real-time scheduling on single processor systems, *priority inversion* problems occur when a medium priority task preempts a low priority task inside a critical section, and is itself preempted by a high priority task which tries to enter that critical section. As the low priority task is not scheduled for execution, it cannot leave the critical section the highest task wishes to enter. The *Priority Inheritance Protocol* (PIP) [5] temporarily raises the priority of the lower priority task to the priority of a higher priority task when the higher priority task is waiting to enter a critical section locked by a lower priority task. The *Priority Ceiling Protocol* (PCP) [5] assigns a ceiling priority to each critical section which has to be assumed on entering the critical section. The *Stack Resource Protocol* (SRP) by Baker [6] solves this for scheduling with dynamic priorities such as Earliest Deadline First (EDF). For these protocols, multiprocessor variants exist [7] [8].

Operating system environments use these protocols or adaptions thereof. OSEK [1] and AUTOSAR [2] use the *OSEK immediate priority ceiling protocol*. POSIX [4] refers to a similar protocol as `PTHREAD_PRIO_PROTECT`.

Some commercial operating systems such as LynxOS with its "Fast Ada Page" [9] employ techniques similar to the described protocols. Previous PikeOS [10] single core implementations used a model comparable to UPRIO|KPRIO.

Linux' vDSO [11] and L4's user-level TCB [12] map a page into user space to share information such as the current thread's control block and IPC arguments (L4), or current CPU and system time (Linux).

The pending event indicator in L4's vCPU concept [13] brings fast interrupt enable / disable pairs to para-virtualized operating systems on top of a micro kernel. Its interface also uses a two-way indicator to signal the interrupt status and pending interrupts like the UPRIO|NPRIO protocol.

Sloth [14] schedules tasks as interrupts and thus delegates scheduling decisions to the interrupt controller. Tasks interface with the interrupt controller directly to change their scheduling priorities. Benchmark results show similar performance benefits for critical sections. However, this performance comes at the price: a malicious task could easily monopolize the CPU. This approach is only feasible if all tasks can trust each other.

Similarly optimized for the fast path, the *Fast User Space Mutex* (Futex) [15] supports mutexes with low overhead, requiring a system call only on contention.

VII. Conclusion

We have shown a concept enabling the currently executing task to change its scheduling priority in user space. When raising priorities, no system calls are needed. For lowering priorities again, system calls are only required when scheduling is necessary. We have described and evaluated two approaches, UPRIO|KPRIO and UPRIO|NPRIO, and discussed the safety impact in case of tasks misbehaving by exceeding their assigned priority ranges.

Both protocols show performance gains for uncontended critical sections and, in case of UPRIO|NPRIO, similar or better performance on contention.

The presented approach is suitable for real-time operating systems with partitioned preemptive fixed-priority scheduling, especially for the OSEK priority ceiling protocol in Automotive.

For future work, we would like to combine the presented approach with the Futex concept of [16] to implement fast *priority ceiling mutexes* and further synchronization primitives in user space for AUTOSAR, ARINC 653, and POSIX use cases. Also, we need to conduct measurements using more complex scenarios to evaluate the impact on real-world applications. Finally, we would like to discuss the applicability to multicore environments, mixed criticality systems, and approaches using dynamic priority scheduling such as EDF.

References

[1] OSEK/VDX, “Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug / Vehicle Distributed Executive.” [Online]. Available: http://www.osek-vdx.org/

[2] AUTOSAR, “AUTomotive Open System ARchitecture.” [Online]. Available: http://www.autosar.org/

[3] ARINC Report 653, “Avionics application software standard interface.”

[4] IEEE, “POSIX.1-2008 / IEEE Std 1003.1-2008 real-time API,” 2008.

[5] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization,” *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[6] T. P. Baker, “Stack-based Scheduling of Realtime Processes,” *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[7] R. Rajkumar, L. Sha, and J. P. Lehoczky, “Real-Time Synchronization Protocols for Multiprocessors,” in *IEEE Real-Time Systems Symposium*. IEEE Computer Society, 1988, pp. 259–269.

[8] P. Gai, G. Lipari, and M. Di Natale, “Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-Chip,” in *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, ser. RTSS ’01, 2001, pp. 73–.

[9] LynuxWorks, “LynxOS RTOS.” [Online]. Available: http://www.lynuxworks.com/

[10] SYSGO, “PikeOS.” [Online]. Available: http://www.pikeos.com/

[11] “vDSO - overview of the virtual ELF dynamic shared object.” [Online]. Available: http://man7.org/linux/man-pages/man7/vdso.7.html

[12] J. Liedtke and H. Wenske, “Lazy process switching,” in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, 2001, pp. 15–18.

[13] A. Lackorzynski and A. Warg, “Virtual Processors as Kernel Interface,” in *Twelfth Real-Time Linux Workshop*, 2010.

[14] W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, “Sloth: Threads as interrupts,” in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, 2009, pp. 204–213.

[15] H. Franke, R. Russell, and M. Kirkwood, “Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux,” in *Proceedings of the Ottawa Linux Symposium*, 2002, pp. 479–495.

[16] A. Zuepke, “Deterministic Fast User Space Synchronisation,” in *OSPERT Workshop*, 2013.