On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System*

Friedrich Schön
GMD—German National Research Center for Information Technology
Kekuléstraße 7
D-12489 Berlin, Germany
fs@first.gmd.de

Wolfgang Schröder-Preikschat, Olaf Spinczyk, Ute Spinczyk
University of Magdeburg
Universitätsplatz 2
D-39106 Magdeburg, Germany
{wosch,olaf,ute}@ivs.cs.uni-magdeburg.de

Abstract

A crucial aspect in the design of (embedded real-time) operating systems concerns interrupt handling. This paper presents the concept of a modularized interrupt-handling subsystem that enables the synchronization of interrupt-driven, non-sequential code without the need to disabling hardware interrupts. The basic idea is to use non-blocking/optimistic concurrency sequences for synchronization inside an operating-system kernel. Originally designed for the PURE *embedded operating system, the presented object-oriented implementation is highly portable not only regarding the CPU but also operating systems and yet efficient.*

1. Introduction

A crucial aspect in the design of (embedded real-time) operating systems concerns interrupt handling, in particular the synchronization of asynchronously initiated code fractions. The underlying model has to be efficient and should also reduce the periods of time during which interrupts are physically disabled to an absolute minimum. Ideally, at software level, interrupts should never be disabled at all. The embedded (parallel/distributed) operating system family PURE [14] follows this ideal pattern and, thus, ensures *interrupt transparency*: selected members of the PURE family at no times disable (hardware) interrupts, or exploit special CPU instruction, for synchronization purposes. Rather, these family members use *non-blocking/optimistic concurrency* sequences for synchronization and result into an operating-system kernel that never has to disable interrupts.

Disabling of hardware interrupts may be disadvantageous for several reasons. For example, in the case of pipeline processors there may arise the need to flush the pipeline before interrupts can be disabled. This is to ensure that certain pipelined instructions are not executed before the disabling of interrupts becomes effective. As a consequence, performance is lost. Another example regards the worst-case interrupt latency, which corresponds to the longest period of time during which interrupts are disabled. In that case, the capability to react fast on external events is limited. Moreover, the probability of loosing external events is increased. There are many other examples, most of them come out of the (embedded) real-time area. One aspect in the design and implementation of real-time operating systems therefore is to reduce the need for disabling interrupts to an absolute minimum or to provide mechanisms that help avoiding such measures at all [12]. The (general purpose) operating systems available at the market base on interrupt disabling to secure critical sections.

There are a couple of (special purpose) operating systems, e.g. [6, 9], that employ atomic CPU instructions and provide a locking-free kernel. However, the specific algo-

*This work has been partly supported by the Deutsche Forschungsgemeinschaft (DFG), grant no. SCHR 603/1-1 and the Bundesministerium für Bildung und Forschung (BMBF), grant no. 01 IS 903 D 2.

rithms to achieve wait-free synchronization [7] often have negative effects on the software structure and make the development of highly modular systems somewhat difficult. In addition, portability is limited. The processors typically used in the deeply embedded [1] systems area do not provide appropriate atomic CPU instructions, such as *compare-and-swap* or *load-linked/store-conditional*, to ease the implementation of wait-free synchronization. Instead, these instructions must be emulated using more fundamental and often very restricted atomic CPU instructions. This emulation means increased resource consumption in terms of CPU clock cycles and memory consumption which cannot always be tolerated.

In order to improve maintainability and portability of (embedded) operating systems other solutions to the interrupt synchronization problem should be pursued. This paper presents a solution to the synchronization problem of interrupt-driven, non-sequential code by making assumptions about the invocation patterns of an *interrupt service routine* (ISR). In addition, the model proceeds from a specific structure, or modularization, of an interrupt-handling subsystem that dismembers an ISR into two closely related parts. The flow of control from one part to the other one of an ISR is regulated by a synchronized queue. This queue then represents the single point at which interrupt safeness has to be ensured. It is the point where the proposed interrupt-transparent (non-blocking/optimistic) synchronization of shared data structures takes place.

In the following sections, first the synchronization model of PURE is discussed. This is followed by a presentation of the (C++) implementation. Afterwards an analysis of the proposed implementation is presented, discussing a number of arguments regarding the suitability of the concept for real-time embedded systems. Some concluding remarks finish the paper.

2. Synchronization Model

The partitioning of an interrupt handler into two closely related portions is a very common approach: UNIX systems (e.g. SVR4 [5], 4.4BSD [10], or Linux [1]) partition the kernel into a *top half* and a *bottom half* and use an *asynchronous system trap* (AST) to force the scheduling of an event related to the top half; interrupt handlers in MARS [4] are divided into a *minor* and *major* section; PEACE [15], the PURE predecessor, invented the notion of *prologue* and *epilogue* for the two halves; and Windows NT uses a *deferred procedure call* (DPC) to propagate interrupts [3]. In all these systems, one of the two parts (i.e. bottom half, AST, major section, epilogue, and DPC) represents an asynchronously initiated function whose execution need to be synchronized with respect to the overall system operation. The major difference to PURE is the technique applied for a safe interplay of the synchronous and asynchronous (i.e. interrupt-driven) code fractions: PURE provides a general and interrupt-transparent solution.

The interrupt-handling subsystem of PURE is made of two main parts (Figure 1). The features provided by the *interrupt synchronization and linkage environment* (ISLE) enable the system to attach/detach an ISR to/from the exception (i.e. trap/interrupt) vectors of the underlying CPU in a CPU-independent manner. They also support, but do not enforce, the *objectification* of ISR software in a CPU-independent manner. ISLE is extended by an *interrupt-driven embedded addition* (IDEA) that, typically, constitutes *device driver* which, in turn, contain the ISR. IDEA also represents the connecting link to other parts of a PURE operating system, for example to the process management subsystem or the process scheduler. That is to say, IDEA contains, or leads to, critical code sections that have to be secured properly.

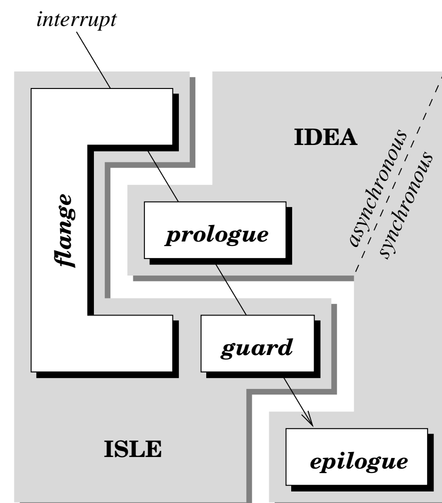

Figure 1. Building blocks of the interrupt-handling subsystem

As indicated in Figure 1, IDEA is subdivided into an asynchronously and synchronously executing part: *prologue* and *epilogue*. The transition from prologue to epilogue is performed by the *guard*. Using *flange*, a prologue is attached to some CPU exception vector and will be immediately started upon signaling a hardware interrupt to the CPU. An epilogue is considered a prologue continuation and will be started only if granted by the guard. The guard takes care of the serialized execution of these epilogues.

[1] The phrase “deeply embedded” refers to systems forced to operate under extreme resource constraints in terms of memory, CPU, and power consumption. The dominating processor technology for these systems is still 8-bit, 16-bit technology is only slowly coming.

Since epilogues are executed with all interrupts enabled, the period of time during which interrupts are blocked is reduced. Since critical sections are not secured by disabling interrupts, the latency of the highest priority interrupt handler can be determined solely basing on the hardware specification of the underlying CPU.

2.1. Prologue Continuations

Epilogues and critical sections are termed *guarded section*. The entire guarded section is controlled by a specific guard. As a consequence, the guard does not only specifically prevent the concurrent execution of epilogues but also of (contemporary) critical sections. These guarded sections, for example, partly implement common operating-system functions such as thread scheduling, event signaling, message communication, and memory management. In Figure 2, these functions constitute POSE, the PURE *operating-system extensions*.

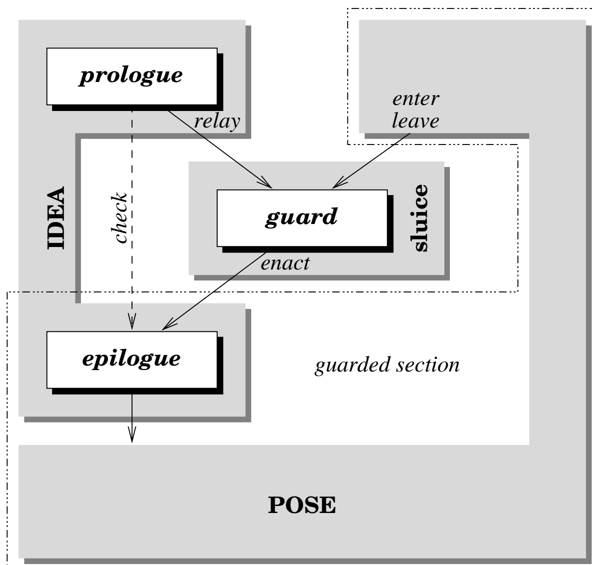

Figure 2. Serialization of interrupt epilogues

Prologues and epilogues are (logically) loosely coupled by the guard. This loose coupling is intimated by the dashed arrow. The guard implements a *sluice* of epilogues and ensures that at a moment in time only one epilogue may be active.

When a prologue wishes to start a continuation (*check*) it hands over to the guard the corresponding epilogue object to be executed (*relay*). The epilogue will be run (*enact*) only when the sluice is free, meaning that the controlling guard is locked. Initially the sluice is free, i.e. all guarded sections are inactive. The free-state of the sluice is canceled when a guarded section becomes active (*enter*), meaning that the controlling guard is unlocked. As long as the guarded section is active, epilogues issued by interrupting prologues are made pending by storing them in a queue. Then the interrupted guarded section continues execution. When the guarded section becomes inactive, the free-state of the sluice will be restored and the queue of pending epilogues will be cleared by processing all pending epilogues (*leave*). Similar holds when a prologue returns to the interrupted thread.

2.2. Phase Transition

A difficulty to be solved by the propagation procedure is to make sure epilogue execution goes along with the lowest possible interrupt priority, i.e. all interrupts are enabled. Since prologues execute on a certain hardware interrupt priority level, the interrupts are to be enabled on the phase transition from prologue to epilogue. At this moment, specific care must be taken such that interrupt handlers will not run the risk of infinite recursive activation and (interrupt) stack overflow.

When the prologue asks the guard for epilogue propagation (*relay*), a test on an active guarded section is performed. The epilogue is made pending, and the interrupted process is resumed, if the request for propagation overlaps with an active guarded section. If overlapping does not take place, a number of follow-up steps take place to make sure that (a) propagated epilogues run interrupts enabled, (b) stack expansion is constrained by a variable but maximum number of nested interrupts, and (c) no pending epilogue has been left after propagation finished. The order of the individual steps is of importance yet: (1) the guard is to be locked, (2) the actual interrupt priority level is to be remembered and then are interrupts to be enabled, (3) all pending epilogues are to be processed, (4) the interrupt priority level is to be restored, and (5) the guard is to be unlocked. Finally, a check on further epilogues that may have been made pending in the meantime is to be performed and, if necessary, the entire sequence is to be repeated.

The fourth step restores a certain interrupt priority level. This also means the disabling of interrupts of the same and lower priority. Nevertheless is the presented approach meant to be interrupt-transparent. Note that the restored interrupt priority level is that of the interrupting prologue that initiated epilogue propagation. This priority level was defined by hardware not by software. So there is nothing "disreputable" to go up to this level when the prologue has finished epilogue propagation and returns to the interrupt-handling code of the ISR: the I/O behavior of the corresponding guard function (*relay*) remains consistent, it starts and ends execution with the same interrupt priority. Restoring the interrupt priority level is mandatory also to prevent the starvation of epilogues.

Since interrupts are enabled after having locked the guard, the growth of the (interrupt) stack is restricted and depends (a) on the maximum number of interrupt levels/priorities and (b) on the stack consumption of each of the maskable interrupt handlers. The maximal expansion of the interrupt stack is fixed and can be computed/approximated before run-time.

3. Implementation

Since the guard operates in an interrupt-driven context, its queue operations need to be synchronized. This happens in an interrupt-transparent manner. A queuing strategy has been implemented that enables the dequeue operation, in order to run delayed epilogues upon leaving a guarded section, to be overlapped by enqueue operations issued by an interrupt handler (i.e. prologue). In addition, enqueue operations are allowed to overlap themselves, e.g. when a high-priority interrupt handler preempts a low-priority interrupt handler. All necessary synchronization measures run with interrupts enabled. Other overlapping scenarios need not be considered.

3.1. Data Structures

PURE is an object-oriented system implemented in C++. In order to be able to keep epilogue objects on a queue, the class implementing an epilogue is inherited from a base class that provides queuing capability. This base class, `Chain`, is providing a `next` pointer to support the implementation of single-linked lists. The head of this list then is implemented by a single `Chain*` instance. Extending the list head by a second `Chain*` instance which implements the pointer to the last list element (i.e. the tail) results in a queue. The resulting class is shown in Figure 3.

```
1: class Cargo : public Chain {
2: protected:
3:   Chain* tail;
4: public:
5:   Cargo ();
6:   void   enqueue (Chain* item);
7:   Chain* dequeue ();
8: };
```

Figure 3. Queue interface

The `Cargo` abstraction implements an *interrupt-transparent queue* according to the first-in, first-out (FIFO) strategy. The critical operations are `enqueue()` and `dequeue()`. They will be discussed below. The constructor `Cargo()` takes care of a specific initial state representing an empty queue with the following properties: (a) `next` (i.e. the head pointer) is nil and (b) `tail` points to `next`. In fact, an empty `Cargo` queue is not really empty in the usual sense. Rather, in this case, `tail` points to a dummy element which is represented by the queue head pointer.

3.2. Concurrency Problem

The data structure described in the previous section makes the distinction whether an element is inserted into an empty or non-empty queue obsolete. Due to the dummy element, the queue never runs empty. As a consequence, queue insertion happens in only two basic steps, namely: (1) `tail->next = item` and then (2) `tail = item`, with `item` being a `Chain*`. A nil head pointer will be automatically set to the first list element when that element is to be placed on a logically empty queue. This is because `tail` points to the `next` pointer of the last element in the queue, and not to the last element. Initially, this `next` pointer is the head pointer itself.

It is obvious that the above mentioned insertion mechanism is not interrupt-safe. An interrupt occurring during or after step (1) and entailing overlapped execution of the same sequence on the same data structure results in an inconsistent state. After step (1) the tail pointer still points to the old last element which, through step (1), became the predecessor of the inserted element. Overlapped execution of the insertion procedure destroys that predecessor relationship and creates a new one: the old last element now points to the element inserted during the interruption phase. At the end of this phase, the tail pointer will point to the most recently inserted element. After return from interrupt, resumed execution of step (2) then lets the tail pointer point to the element that was going to be placed on the queue but whose insertion was interrupted. The effect is that that element, and any other element enqueued afterwards, will never be found on the queue when following the head pointer. A dequeue typically employs the head pointer to remove the next element from the (FIFO) queue. Consequently, due to overlapping enqueue operations, elements may be lost.

3.3. Interrupt-Safe Enqueuing

The implementation of an interrupt-safe `enqueue()` is listed in Figure 4. According to FIFO, a newly to be inserted element becomes the last queue element and has to terminate the list. List termination is handled in line 2 and insertion of an element is handled in line 8. Furthermore the tail pointer needs to be updated. This is done in line 4. In contemporary queue implementations, manipulation of the tail pointer finishes the insertion procedure. In the case discussed here this manipulation takes place before the element is added to the queue.

The fundamental idea for making `enqueue()`

```
1: void Cargo::enqueue (Chain* item) {
2:   item->next = 0;
3:   Chain* last = tail;
4:   tail = item;
5:   while (last->next) {
6:     last = last->next;
7:   }
8:   last->next = item;
9: }
```

Figure 4. Enqueue operation

interrupt-safe is setting the tail pointer to the element to be enqueued before the insertion actually takes place. The assignment is atomic, however it causes the loss of the insertion point. Therefore, the insertion point must be remembered (line 3) before the tail pointer is manipulated. But in this case overlapped execution of `enqueue()` between lines 3 and 4 may cause the insertion of further elements. This means that `last` of the interrupted instance potentially does not really point to the last element of the queue and, thus, no longer remembers the right insertion point when the interruption phase ends. The new insertion point must be found by searching for the end of the list (lines 5–7). Finally, the insertion can take place. The overlapped execution of `enqueue()` still results in an ordered queue. Note that the order in which elements are placed on the queue does not necessarily correspond to the order of `enqueue()` calls, but rather to the order in which the assignments of the overlapping invocations indicated by line 4 are completed.

3.4. Interrupt-Safe Dequeuing

The previous paragraphs discussed the case of overlapping `enqueue()` by itself. This may happen when a low-priority prologue is interrupted by a high-priority prologue and both prologues request the invocation of their associated epilogues. The other case to be handled is that `dequeue()` is overlapped by `enqueue()`. This may happen when a guarded section is going to be left and, thus, pending epilogues are to be removed (employing `dequeue()`) from the queue of the guard. The case that needs not to be handled is when `dequeue()` interrupts itself and `enqueue()`. This kind of overlapping would be possible only if prologues decide to remove epilogues from the queue—and this does not correspond to the pro/epilogue concept of PURE.

The interrupt-safe implementation of `dequeue()` is shown in Figure 5. In this implementation, the particular concurrency problem arises if the queue holds only one element and at the same time `dequeue()` is overlapped by at least one `enqueue()`. Removing this last element from the queue normally would result in the initial ("empty") queue state. However this requires the manipulation of `tail` by `dequeue()`—and `tail` is also manipulated by `enqueue()`. In this particular situation, it may happen that new elements will be added not to the queue but to the element having been dequeued. The idea of an interrupt-safe `dequeue()` now is (1) to check for exactly this case (line 5) and (2) to requeue all elements that have been enqueued by mistake (lines 6–11). Line 3 checks for a non-empty queue, if necessary removes an element from the list, and checks whether this was the last element. Line 4 creates the initial state if the queue run empty.

```
 1: Chain* Cargo::dequeue () {
 2:   Chain* item = next;
 3:   if (item && !(next=item->next)) {
 4:     tail = (Chain*)this;
 5:     if (item->next) {
 6:       Chain* lost = item->next;
 7:       Chain* help;
 8:       do {
 9:         help = lost->next;
10:         enqueue (lost);
11:       } while (lost = help);
12:     }
13:   }
14:   return item;
15: }
```

Figure 5. Dequeue operation

4. Analysis and Discussion

The correctness of the `Cargo` implementation has been validated by model checking. First investigations confirm the correctness of `enqueue()` [13]. The following subsections discuss the proposed interrupt synchronization mechanism in terms of overhead and real-time behavior.

4.1. Overhead

The overall performance of the PURE interrupt-synchronization scheme has been documented earlier at a different place [2]. Table 1 shows the overhead of two variants of the synchronized queue in comparison with the plain, unsynchronized queue. The number of (Pentium II, 300 MHz) CPU instructions (ins.) and clock cycles (cyc.) are presented for the `enqueue()` and `dequeue()` primitives, based on the C++ compiler `egcs-1.0.2` release for i80x86 processors. Method inlining has been disabled.

function	interrupt synchronization					
	none		*blocking*		*transparent*	
	ins.	cyc.	ins.	cyc.	ins.	cyc.
`enqueue()`	10	84	19	127	12	86
`dequeue()`	17	82	21	135	26	90
total	27	166	40	262	38	176

Table 1. Overhead of the queue operations

The numbers given for the interrupt-transparent synchronization indicate the best case: the `while`-clause of `enqueue()` (Figure 4, lines 5–7) and the `if`-clause of `dequeue()` (Figure 5, lines 5–12) are not executed. Only the expressions of the corresponding conditions are evaluated. In addition, the numbers of `dequeue()` in all three cases are for a queue with exactly one stored element. The numbers for the blocking case include instructions and clock cycles needed for disabling/enabling interrupts.

In the worst case of `enqueue()`, additional instructions for searching the end of the list must be taken into account (see also lines 5–7 of Figure 4). The clock cycles consumed, however, depend on the length of that list. The additional overhead for a single iteration amounts to 3 instructions. In case of `dequeue()`, the worst case adds instructions for (1) preparation of element requeuing (line 6 of Figure 5) and (2) element requeuing itself (lines 8–11 of Figure 5). Again, the number of clock cycles consumed for the second step depends on the queue length. Also note that requeuing is implemented using `enqueue()`. The additional `dequeue()` overhead in the worst case, excluding `enqueue()`, is 24 instructions.

Comparing the clock cycles of transparent against blocking interrupt synchronization documents a fairly lightweight implementation. As shown, the interrupt transparent `enqueue()` and `dequeue()` perform about 33 % better than the interrupt-blocking peers. An `enqueue()` performs worser only if been interrupted at the critical point (between line 3 and 4 of Figure 4) for at least three times, creating a list of at least three epilogues. A faster `enqueue()` generally decreases the interrupt latency. This makes the interrupt transparent `enqueue()` somewhat superior to the interrupt-blocking variant.

The best case numbers for the interrupt-transparent implementation of both `enqueue()` and `dequeue()` indicate the overhead of these two primitives for the assumed normal situation in which no interruption takes place. This situation is considered normal not only because a single interrupt at a time is a fairly exceptional event, but also because overlapping interrupts are much more exceptional. The critical and somewhat overhead-prone case comes up only when `enqueue()` (1) gets interrupted at sensitive locations and (2) could be thus overlapped by itself and when (3) the interrupting instance shares the same queue with the interrupted instances. Even for stressed systems this combination of events is of quite low probability.

The clock-cycle difference between blocking and none interrupt synchronization indicates the overhead of disabling and enabling interrupts: 43 cycles for `enqueue()` and 53 cycles for `dequeue()`. Compared to the raw runtime of both primitives (84 resp. 82 clock cycles), this results into a fairly large overhead.

4.2. Worst Case Overhead

A prediction of the *worst case overhead* (WCO) of the interrupt-transparent `enqueue()` and `dequeue()` depends on the actual number of epilogues queued by the guard. This number, N_{epi}, in turn depends (a) on the hardware interrupt priority scheme and/or (b) on the implementation of the device drivers. The following formulas hold for the two functions:

$$\begin{aligned} WCO_{enq} &= O_{ins} + (N_{epi} - 1) * O_{ski} \\ WCO_{deq} &= O_{rem} + O_{pre} + N_{epi} * O_{req} \end{aligned}$$

In these formulas, O_{ins} gives the overhead for list insertion of a single item, O_{ski} for skipping a single item when searching for the list end, O_{rem} for deleting an item from a single-element list, O_{pre} for preparing element requeuing, and O_{req} for requeuing a single element. Given a certain CPU, these all are constant parameters. The values of which can be determined by source code or run-time analysis. The only variable parameter is N_{epi}. However, N_{epi} remains an unknown quantity only if, in case of overlapping operation, every intervening interrupt (i.e. prologue) calls for epilogue processing. This will be the case when prologue activations always ask the guard for continuation as epilogue, independently of the actual execution state of that epilogue. However, a typical protocol between prologue and epilogue will be that the prologue calls only for an inactive epilogue. That is, whenever a prologue interrupts its associated epilogue it will not call for epilogue activation but rather communicate with the active epilogue directly. This limits the maximal value of N_{epi} making it dependent only on case (a) above. Being faced with a hardware that implements an 8-level interrupt priority scheme, the maximum value of N_{epi} will be eight.

4.3. Interrupt Frequency

The PURE synchronization concept does not prescribe any constraints on the interrupt frequency that can be dealt with. When operating-system software never disables interrupts, the duration until which the next interrupt (of the

highest priority level) can be handled mostly depends on the duration of the ISR—and the ISR, i.e. the prologue code, is not considered an integral PURE building block, but rather a component that is strongly related to the application domain. That is to say, PURE is free of any system parameter that limits the frequency by which interrupts of the highest priority can be handled.

4.4. Real-Time Capabilities

At the level of abstraction of the proposed synchronization mechanism, the length of the epilogue queue is virtually unlimited. In a PURE environment, the epilogue queue is given a maximum length by a careful design and implementation of all device drivers. One design issue, for example, is to ensure that a prologue always requests the execution of a single epilogue at a time and only if the execution of this epilogue is not pending at that moment. In this case, the maximal possible queue length corresponds to the number of epilogue-demanding prologues (i.e. interrupt handlers) in the system. In particular, if interrupt-priority levels are not shared by many prologues, the maximal possible queue length corresponds to the maximal possible number of nested prologues at a time. The sum of the worst case execution times of all the queued epilogues then determines the maximal possible delay for a process to leave a guarded section.

In the current PURE implementation, all epilogues are equally qualified for execution. This is enforced by employing a simple FIFO queuing strategy when an epilogue needs to be made pending. When being executed, any of these epilogues may ask for the scheduling of a process of a higher priority than the currently executing process. As a result, the current process will be preempted in favor of the execution of some higher-priority process. Whether or not that process really is of the highest priority, and, thus, preemption at that moment was the right decision, depends on the actions performed by the subsequent epilogues in the queue.

While the guard works according to FIFO, a real-time scheduler typically prefers some priority- or deadline-based scheme. Assuming there are two pending epilogues, E_1 and E_2, and the execution of these epilogues entails the scheduling of two processes, $P(E_1)$ and $P(E_2)$, each of which having a higher priority than the currently executing process. Further assume that the priority of $P(E_1)$ is lower than the priority of $P(E_2)$. In such a situation, and if the execution of any epilogue could result in the preemption of the currently running process, lower priority $P(E_1)$ would be run before higher-priority $P(E_2)$ since FIFO epilogue propagation executes E_1 before E_2.

But, being preempted or not, a process always voluntarily relinquishes the CPU by executing scheduler code. In PURE, the scheduler is controlled by a guard and a process being run, thus, at first always has to leave some guarded scheduler section. Leaving a guarded section eventually causes the process to actively propagate pending epilogues. In the scenario described above, the propagating i.e. currently running process will be $P(E_1)$, which then executes E_2, which schedules $P(E_2)$, which preempts $P(E_1)$.

Thus, the guard occasionally does not really favor a lower-priority process over a higher-priority process, but it may cause a further scheduling latency until the highest-priority process will be run. That latency has an upper bound and depends on the maximal possible number of pending epilogues at a time. The scheduling latency can be avoided only if guard and scheduler are one thing, i.e. if the guard conforms to the strategy of the real-time scheduler. However, this requires the management of an interrupt-transparent queue other than according to FIFO—which tends to increase overhead and somewhat limits the real-time properties. The approach followed by the current PURE implementation therefore is to only register the scheduling decisions during epilogue propagation and then have a single, finishing *scheduler epilogue* that performs preemption. This measure reduces the scheduling latency by avoiding "false preemption" that might be caused by the epilogues.

The problem discussed above may occur only in system configurations of more than one preemption causing epilogue. These configurations are more likely with event-triggered real-time systems. In case of time-triggered systems [4], there will be at most one such kind of epilogue, namely a *clock epilogue*. May be there will be no epilogue at all, because the entire system runs in polling mode and does not rely on interrupts (i.e. prologues). All these scenarios are application-domain specific and supported by tailor-made members of the PURE family of operating systems. A further member is currently developed that eliminates false preemption using integrated scheduling of both epilogues (i.e. passive objects) and processes (i.e. active objects).

5. Conclusion

This paper discussed the design and implementation of a system that supports interrupt-transparent (non-blocking/optimistic) synchronization of interrupt-driven concurrent code. The presented approach assumes the structuring of an interrupt-handling subsystem, e.g. a device driver, into two parts: a prologue and an epilogue. The prologue represents an asynchronously executing interrupt handler. The epilogue represents a synchronously executing prologue continuation allowed to execute critical sections of an operating-system kernel. The connecting link between both parts is implemented by a guard whose purpose is to control the execution of so-called guarded sections. A seri-

alization of the execution of guarded sections is performed by collecting delayed epilogues in a queue. In this setting, the queue represents the single point whose implementation has to be ensured to be interrupt safe.

The presented performance figures of the PURE implementation document a still "featherweight" solution. As a result, the delay of an interrupted process due to queuing overhead can be reduced when compared to more traditional ways of queue synchronization that physically disable/enable interrupts. This aspect is of particular significance for (embedded) real-time systems whose goal, besides others, must be to constrain the effects of disruptive elements such as interrupts to an absolute minimum. Since the PURE concept does without blocking of interrupts, the latency of the highest-priority interrupt handler solely depends on static parameters, such as the processor speed or the number of CPU instructions, and not on dynamic aspects, such as the timed occurrence of interrupts. Thus, that latency can be determined in advance, before run-time.

Critical sections in PURE are so-called guarded sections. A guarded section is never preempted to run epilogues or, because event-driven rescheduling is considered a prologue continuation, another process. These sections may be preempted only to run prologues. Pending epilogues will be processed upon leaving a guarded section. As a consequence, the worst case delay of a guarded section is determined by the number of (a) interrupting prologues and (b) delayed epilogues. The former factor depends on the interruption model of the CPU and the external physical processes, the latter factor is constrained by the design of the device drivers. The PURE kernel consists of only a handful of, and moreover very small, guarded sections.

Prologues and epilogues of PURE may be event-triggered or time-triggered. A criticism of time-triggered systems is to give up the dynamics and flexibility of event-triggered systems in favor of a static and somewhat inflexible solution. Approaches for hard real-time scheduling as followed, for example, with Spring [16] or MARUTI [8] thus are supported as well by PURE. The kernels of Spring and MARUTI, however, perform blocking synchronization of interrupts, while PURE is providing a non-blocking/optimistic solution.

PURE is designed as a *program family* [11]. The family idea particularly accompanied the design and development of the interrupt synchronization and linkage environment of PURE. Selected family members perform interrupt-transparent synchronization, some perform interrupt-blocking synchronization, and others perform no synchronization at all. The highly modular design of highly performance-sensitive parts yet yields a high performance and pure object-oriented system in C++. The goal of PURE is to give applications exactly the resources they need to perform their tasks, no more and no less.

References

[1] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley, 1998. ISBN 0-2011-33143-8.

[2] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, St Malo, France, May 1999.

[3] H. Custer. *Inside WINDOWS-NT*. Microsoft Press, 1993.

[4] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The Real-Time Operating System of MARS. *Operating Systems Review*, 23(3):141–157, 1989.

[5] B. Goodheart and J. Cox. *The Magic Garden Explained — The Internals of UNIX System V Release 4*. Prentice Hall, 1994. ISBN 0-13-098138-9.

[6] M. B. Greenwald and D. R. Cheriton. The Synergy between Non-Blocking Synchronization and Operating System Structure. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 123–136, October 28–31 1996. Seattle, WA.

[7] M. P. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

[8] S.-T. Levi, S. K. Tripathi, S. D. Carson, and A. K. Agrawala. The MARUTI Hard Real-Time Operating System. *Operating Systems Review*, 23(3):90–105, 1989.

[9] H. Massalin and C. Pu. A Lock-Free Multiprocessor OS Kernel. Technical Report CUCS–005–91, October 1991.

[10] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4* BSD *Operating System*. Addison-Wesley, 1996. ISBN 0-201-54979-4.

[11] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-5(2):1–9, 1976.

[12] K. Ramamritham and J. A. Stankovic. Scheduling Algorithms and Operating Systems Support for Real-Time Systems. *Proceedings of the IEEE*, pages 55–67, Jan. 1994.

[13] K. Schneider, M. Huhn, and G. Logothetis. Validation of Object Oriented Concurrent Designs by Model Checking. In *Proceedings of the 10th IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARM99)*, Bad Herrenalb, Germany, September 27–29 1999.

[14] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP WG 9.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES '98)*, Paderborn, 1998.

[15] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.

[16] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Operating Systems. *Operating Systems Review*, 23(3):54–71, 1989.