FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Department Informatik

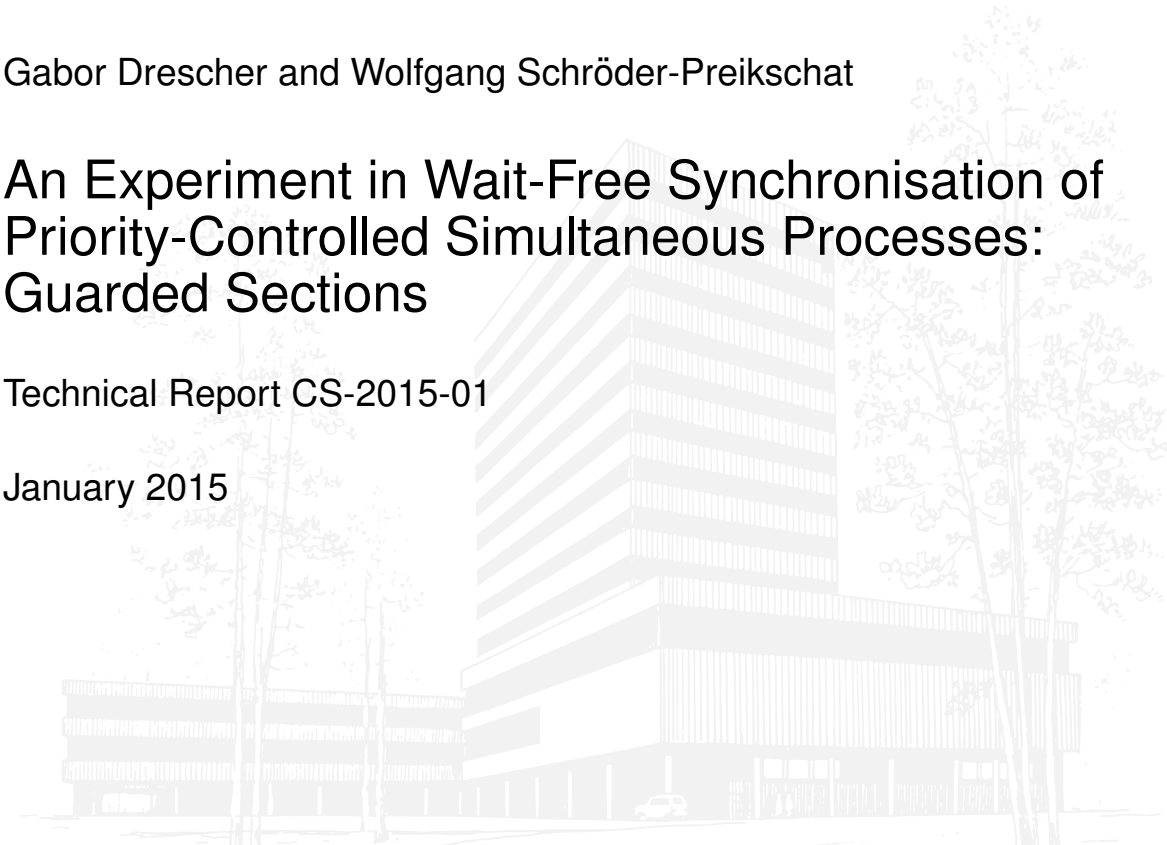
Technical Reports / ISSN 2191-5008

Gabor Drescher and Wolfgang Schröder-Preikschat

An Experiment in Wait-Free Synchronisation of Priority-Controlled Simultaneous Processes: Guarded Sections

Technical Report CS-2015-01

January 2015



Please cite as:
Gabor Drescher and Wolfgang Schröder-Preikschat, “An Experiment in Wait-Free Synchronisation of Priority-Controlled Simultaneous Processes: Guarded Sections,” Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science, Technical Reports, CS-2015-01, January 2015.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.cs.fau.de

An Experiment in Wait-Free Synchronisation of Priority-Controlled Simultaneous Processes: Guarded Sections

Gabor Drescher and Wolfgang Schröder-Preikschat
Dept. of Computer Science, University of Erlangen, Germany
{drescher,wosch}@cs.fau.de

***Abstract*—Wait-free synchronisation gives any process in the system strong progress guarantees, irrespective of number and behaviour of other processes simultaneously competing for shared resources (i.e., data structures and code sections). It ensures completion of any operation in a finite number of steps and, thus, provides the basis to derive bounded above or even constant executions times for non-sequential programs. This characteristic is of special meaning for time-dependent processes typical for real-time (embedded) systems. But wait-free synchronisation against the background of especially arbitrary data and code structures is no bed of roses.**

This paper is about organising non-sequential programs to the benefit of wait-free synchronisation. Conventional critical sections are designed as so called *guarded sections*. Unlike critical sections, preferential processes never block at entrance to a guarded section though only one process at a time is allowed to pass through. Competing processes are forced into bypass but, if necessary and by using *futures*, they can synchronise on concurrent state changes inside the respective section. In consequence of this measure, the execution model of guarded sections constrains the overlapping pattern of interacting (simultaneous) processes. Thereby, efficient wait-free synchronisation of the "guarding operations" is a gratifying by-product. First experiments on a 80-way multi-core system show that non-blocking wait-free synchronised guarded sections outperform lock-based protection schemes such as MCS-locks.

I. Introduction

Multi-core processors have conquered the domain of real-time systems in general and embedded systems in particular. The technological change towards the multiplication of (identical/different) processing units integrated on a single processor chip was mainly due to diminished performance gains in face of increasing operating frequency, a phenomena that was largely called forth by three handicaps as to memory, instruction-level parallelism, and power and summarised as *brick wall* [1], [2]. This, in turn, caused software to be confronted with the problem of exploiting the explicit parallelism made available by the hardware for being processed with good or even high-performance. An undertaking that evoked and still brings "dire straits" to general-purpose computing, but appears to be much more challenging for special-purpose computing as to time-dependent systems.

The constraint of real parallelism not only intensified the already existing problem of non-deterministic operation, but it also amplified interference with real-time scheduling decisions due to more difficult synchronisation methods needed to ensure consistent operation in the data as well as time domain. A fact that is not only limited to event-triggered systems but addresses time-triggered systems equally.

A. Background

Synchronisation of simultaneously interacting processes can be blocking or non-blocking, whereby the latter is inherently free of priority violation, priority inversion, and process deadlock. By concept, direct interference with process scheduling is excluded.[1] In addition, non-blocking synchronisation is differentiated as to the following progress guarantees:

- *obstruction-free*, if any process eventually in isolation (i.e., absence of simultaneously interacting processes) can complete any operation in a finite number of steps [5]

An abbreviated version of this document is republished under the title "*Guarded Sections: Structuring Aid for Wait-Free Synchronisation*" in the Proceedings of the 18th IEEE Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2015).

[1]Disregarding backoff [3], [4] for the resolution of potential contention at atomic machine instructions, which is an issue for blocking synchronisation (e.g., to resolve lock contention) but should also be considered for non-blocking synchronisation. There is no real difference between both paradigms in this respect.

- *lock-free*, if "some process will complete an operation in a finite number of steps, regardless of the relative speeds of the processes" [6]
- *wait-free*, if "any process can complete any operation in a finite number of steps, regardless of the relative speeds of the other processes" [6]

In respect of real-time capabilities, wait-free synchronisation gives strong progress guarantees in that it ensures starvation freedom for each single process and bounded above or constant execution times of the synchronisation protocols. Also supportive of *worst-case execution time* (WCET) analysis [7], all these features are highly desirable for hard real-time systems. In contrast, lock-free synchronisation gives a whole system of processes progress guarantee, yet is prone to starvation of single processes. As event-triggered, thus, priority-based real-time scheduling lets low-priority processes starve anyway, lock-freedom is acceptable but not preferred for corresponding hard real-time systems. Obstruction-free synchronisation provides the weakest progress guarantee. If at all, it becomes an option only for time-dependent processes being subject to strict cooperative scheduling, that is, when any process in the system is capable of processor control in direct responsibility.

For soft/firm real-time systems, wait-free synchronisation is a desirable feature because, on the one hand, of the much higher quality criteria (e.g. in terms of process jitter) and, on the other hand, it establishes in particular *predictability*. The other side of the coin is the structural complexity behind wait-free synchronisation protocols, compared to lock-/obstruction-free solutions as to the same problem. While non-blocking synchronisation in general is preferable for smaller data structures or problems compared to blocking synchronisation, the complexity issue makes wait-free approaches rather suitable only for larger or more complex structures [8]—the exception proves the rule.

Recent discussions about multi-core synchronisation indicates the importance of hardware, rather than software, properties as far as scalability of synchronisation techniques is concerned [9]. But this examination is limited to (1) blocking synchronisation, (2) scalability ("scale out"), and (3) time-independent (general purpose) systems, which makes general transferability of the outcome largely a realm of speculation. In addition, even if optimised solutions for "old-fashioned" parallel processors [10], [11] nowadays no longer differ in results [9] it will be a hasty conclusion that fundamental problems as to the coordination of simultaneously interacting processes disappeared with contemporary hardware platforms. Optimisation for scalability is one aspect, others are energy efficiency, timeliness, and predictability.

B. Contribution

The paper is about an efficient approach of "guarding" critical sections, so called *guarded sections*, which show all benefits of wait-free synchronisation and at the same time are void of the methodical and technological complexity [12] of that paradigm in functional and non-functional terms. Other than conventional critical sections, *preferential processes* (i.e., in static or dynamic terms prior-ranking jobs/tasks) never block at entrance to a guarded section although only one process at a time is allowed to pass through. Guarded sections may be active and, thus, occupied by exactly one process, or inactive (i.e., unoccupied). When a process approaches an active guarded section it gets redirected around it while leaving a *passage request* that becomes operated by the occupying process in due time. More specifically, a preferential process attempting to enter an active guarded section waits only until the passage request has been composed and made available (like *no-wait send* [13]).

If necessary and by using a *future* [14], a redirected process can synchronise on concurrent state changes that have to take place inside the guarded section. In that case, the passage-request order contains a future reference that is used by the occupying process in due time to signal completion of the state change (e.g., availability of a computational result). Similar to issuing a passage request, the process producing the result of a guarded section waits only until the future value has been composed (do. *no-wait send*). Furthermore, signalling of a future result is not only non-blocking as to the signaller but also effective even if the signallee is not yet synchronised on that event (prevention of *lost-wakeup problem*).

This processing model of guarded sections results in a *multiple-enqueue/single-dequeue* mode of operation of the passage-request (i.e., guard) queue and, thereby, paves the way for a highly efficient and wait-free synchronised dynamic data structure shared by many producers (i.e., directed processes) and a single consumer (i.e., occupying process) of passage-request orders. Basis is a *constructional approach* that eventually relieves processes from potential blocking at critical sections at any time. More precisely, processes will potentially block only because of *unilateral* (i.e., logical or conditional) *synchronisation* according to well-defined and explicit data dependencies, but never by reason of multilateral synchronisation. This substantially constrains extent and

structure of the group of interacting processes for a blocking-time analysis, if required.

C. Outline

The rest of the paper is organised as follows. Section II motivates the presented approach by getting deeper into the problem of blocking synchronisation. Section III explains the principle design of guarded sections, followed in Section IV by a description of the implementation and, in Section V, an analysis of first experimental results made with a 80-way multi-core system. Section VI discusses related approaches and Section VII draws conclusions and sketches future work. In order to demonstrate structure and complexity of a dedicated and wait-free run-time support system built from scratch, Appendix A introduces a time-predicatable operating-system executive for guarded sections that are subject to logical or conditional, respectively, synchronisation.

II. Motivation

Enforcement of scheduling decisions is an important aspect in any computing system, but of vital requirement for real-time computing systems. A critical *interference factor* in this regard is (1) synchronisation of simultaneously interacting processes and (2) contention resolution. In case of *mutual exclusion* for blocking (multilateral) synchronisation, the former is prone to *priority inversion* [15]. As a matter of principle, non-blocking synchronisation is free of this problem and, thus, makes counteractive measures [16], [17] such as non-preemptive critical sections, priority inheritance, or (stack-based) priority ceiling protocols unnecessary due to absence of "non-preemptive reusable resources" in shape of conventional critical sections. As the case may be, of course, those measures are necessary only to control sharing of respective resources different from critical sections. But the use of non-blocking synchronisation substantially reduces the problem space in this aspect.

Both of the above-mentioned factors may cause *priority violation* because of the waitlist of blocked processes associated with each critical section or retry of an atomic read-modify-write instruction (e.g., TAS or CAS) applied to the lock variable, respectively. While management of the critical-section waitlist can be easily adapted to the processor waitlist maintained by the process scheduler, corresponding measures as to contention resolution are difficult. For the latter, a *backoff* [3], [4] is common to disperse repeated execution of an atomic instruction. Whether static, (truncated) exponential, or proportional (e.g., ticket spin-lock) backoff, all these methods result in a serialisation of processes according to arrival time at the "hot spot." Similar holds for queueing locks [11]. Such schemes interfere with all but one (FCFS) scheduling discipline, but cause harmful background noise and disruptive behaviour only in a real-time environment.

As far as obstruction-/lock-free methods are concerned, due to and in the course of retries, even non-blocking synchronisation is faced with the priority-violation problem just discussed. If contention resolution namely becomes necessary, only wait-free synchronisation enables processing free of interference with real-time process scheduling—but this synchronisation method is no walk in the park. In a number of cases, wait-free synchronisation largely benefits from *helping schemes* [18]. Normally, these schemes rely on the cooperativeness of interacting processes in order to complete a certain operation in finite time. A supportive measure can be a "software architecture" that forces interacting processes into a dedicated *overlapping pattern* and, thus, founds the basis for simpler synchronisation protocols. As explained next, the concept of guarded sections follows such a constructional approach. Besides providing a general structuring aid for non-sequential programs, this concept also provides a migration path towards wait-free synchronisation as it becomes easily amenable to complex software structures, particularly legacy software.

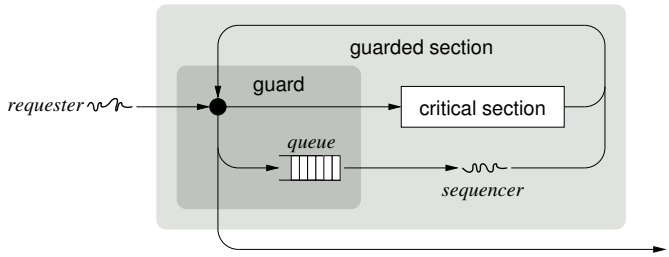
III. Design

In structural respect, guarded sections are not unlike conventional critical sections but as to its flow model very different. Key aspect of the concept is that a preferential process never blocks incoming a guarded section, though its request to pass through that section may be delayed. The model comes up to a *conditional fire-and-forget pattern* of orders to execute a particular program section repeated sequentially. Depending on the (application-specific) function of this program section, different types of guarded sections exist: *non-blocking*, *direct-result*, and *explicit-blocking*.

A. Non-Blocking Guarded Sections

The basic configuration assumes *run-to-completion* processes inside a guarded section. Suchlike processes are free from self-induced wait states as to the possible non-availability of reusable or consumable resources. Those processes will never block inside a guarded section but they are subject of preemption by high-priority processes. Fig. 1 shows this basic model.

In Fig. 1(a), *requester* stands for the process approaching a guarded section and having order to execute a



(a) Flowchart.

1: **if** $(task \leftarrow \text{VOUCH}(guard, order)) \neq 0$ **then**
2: **repeat** ▷ sequential part
3: $handle(task)$
4: **until** $(task \leftarrow \text{CLEAR}(guard, TRUE)) = 0$
5: **end if**

(b) Sample: adopting role as a sequencer.

Fig. 1. Non-blocking guarded section.

critical section. Make out *order* (line 1(b).1) as an object that specifies the "actual parameters" of a particular cycle (line 1(b).3) of a guarded critical section. The *guard* takes care of "traffic control" as to that section. If inactive, the requester is allowed to pass through, thus, activate the guard, occupy the guarded section and handle the order (line 1(b).3). In case of an active guard, the requester's order gets queued and the requester itself is forced to bypass the guarded critical section. All steps necessary for requester control are executed by VOUCH (line 1(b).1), whose return value is a reference (*task*) to the order that shall be processed next.

At the end of a guarded critical section, the occupying process checks the queue for pending orders. If the queue is filled, that process removes the next order from the queue and handles it on behalf of the process having originally ordered critical-section execution. That is, the particular process occupying a guarded section takes the role of a *sequencer* for pending orders as long as the guard queue is filled. Sequencer control is the function of CLEAR (line 1(b).4), whose return value is a reference to the order that shall be processed next.

In this processing model, only a single process, namely the sequencer, is in charge of removing orders from the guard queue. In contrast, on the input side, many requester processes may add orders to that queue. Thus, the guard queue is invariably accessed in a *multiple-enqueue/single-dequeue* style, which significantly eases *wait-free solutions* (cf. Sec. IV) when compared to more general answers [18].

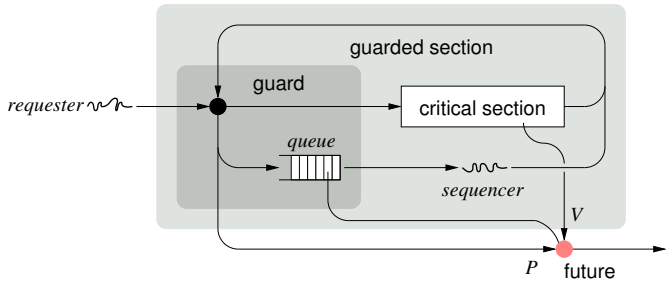
Obviously, any sequencer potentially incurs a delay determined by the number and individual processing time of orders pending in the guard queue. That means, a process passing through a guarded section can be held up in making further progress depending on the incidence of other processes simultaneously approaching this very section. Such a behaviour apparently provokes *lock-free* progress guarantee of a guarded critical section although underneath of it the "guarding operations" are carried out in wait-free manner: a sequencer could be prone to starvation. But with prioritised real-time processing assumed all these orders must have been issued by high-priority processes: none of these high-priority processes was blocked. In priority-based systems, low-priority processes always are subject to starvation.

The fact that low-priority requester processes will not afflict a high-priority sequencer (in a preemptive prioritised system) is obvious as to uni-processor mode of operation—unless the priority-based process scheduler would favour low-priority processes, which is a contradiction in terms. In case of a multi-processor system, however, low-priority processes residing on processors different from a high-priority sequencer could indeed deliver further orders and, thus, cause interference. In order to prevent potentially unbounded delay, a pragmatic approach is to conditionally block a low-priority requester out until the high-priority sequencer leaves the guarded section. That is to say, low-priority requesters arriving at a guarded section then will be subject to *logical synchronisation* with the high-priority sequencer just completing this very guarded section. This is considered as an optional feature to assist a priority-based multiprocessor scheduler in saving *priority loyalty*, namely by giving instructions to unschedule simultaneous low-priority processes for a certain period of time.

In addition to the aforementioned, likewise configuration dependent, a high-priority process (1) occupying a guarded section and (2) having pre-empted a low-priority process in the course of clearing a yet filled guard queue (cf. Sec. IV-B) can refuse role adoption of a sequencer and, thus, never would be delayed because of order processing. For low-priority processes, the guard queue contains as many orders as could have been issued meanwhile by high-priority processes. Based on preliminary knowledge as to process organisation, priority mapping, minimal interarrival times of requesters, and WCET of the guarded section, length of stay of a sequencer can be computed and, thus, bounded above.

B. Direct-Result Guarded Sections

The basic configuration treated so far forms a pure fire-and-forget pattern: orders to a sequencer are not only



(a) Flowchart.

1: **if** $(task \leftarrow \text{VOUCH}(guard, order)) \neq 0$ **then**
2: **repeat** ▷ sequential part
3: $data \leftarrow handle(task)$
4: $\text{PROVE}(task.tobe, data)$ ▷ promise value
5: **until** $(task \leftarrow \text{CLEAR}(guard, TRUE)) = 0$
6: **else** ▷ occupied, bypass
7: $other(\ldots)$
8: $value \leftarrow \text{EXACT}(order.tobe)$ ▷ await future
9: **end if**

(b) Sample: adopting role as a sequencer.

Fig. 2. Direct-result guarded section.

fired but also forgotten by the requester. With VOUCH, the requester assumes a "warranty claim" as to order processing through the sequencer (i.e. oneself, as the case may be) in due time and continues concurrently. Parallel operation is improved without the need to meet the challenge of a redesign of the sequential program section towards non-blocking synchronisaton. This pattern goes very well with "void-type" guarded sections, that is, if the respective program section is not expected to provide a computational result to the requester. If, however, the requester needs to interact with the sequencer in order to receive a concurrently computed value from some action inside the guarded section, *unilateral* (i.e., logical/conditional) *synchronisation* becomes necessary.

A minimal extension of the previous configuration in respect of a "vaule-returning" guarded section is shown in Fig. 2. The main difference to a non-blocking guarded section is the use of a *future* [14] to enable a requester to capture a value from the sequencer while going past a guarded critical section. Basically, a "future value" is of a multi-elementary abstract data type consisting of a single-assignment container (*promise* [19]) as placeholder of problem-specific type consistent with the guarded critical section, a promise *indicator* (kept, broken, pending), and a notification mechanism. The latter, for example, could be a *binary semaphore* [20] set up in a producer/consumer mode of operation (cf. Fig. 2(a)) or a simple per-process *software latch*. Whatsoever, key point is that the mechanism provides for a "sticky bit" that holds a signal once produced and gets reset once consumed.

When the sequencer finished computation of a future value, it keeps the promise to deliver the result and notify the requester (line 2(b).4). That is, (1) the future value gets defined, (2) the promise indicator changes from "pending" to "kept", and (3) a signal is produced to prove promise assignment. The requester exacts (i.e., probes and, if need be, awaits) the future value in due course (line 2(b).8), latest when it requires the data for the own ongoing computation. Thereto, the requester (1) consumes the signal and (2) makes use of the deposited value depending on the promise indicator: if "kept", unloads the container; otherwise, raises an exception.

Such direct-result guarded sections benefit from any form of notification mechanism as long as signalling is (1) still effective (i.e., "sticky") even in case the requester is not yet synchronised on receiving the future value and (2) a throughout non-blocking operation for the sequencer. The former is to prevent lost-wakeup of the requester, which is readily facilitated using a semaphore or latch. The latter is to get around blocking of the sequencer due to hidden conventional critical sections. That is to say, in the present case (Fig. 2(a)), V as well as all operations downward the call hierarchy are void of mutual exclusion for synchronisation measures (cf. case study in App. A). According to the idea of guarded sections, *implicit blocking* of processes would not only be counterproductive—purpose is to support non-blocking synchronised interacting processes—but also prejudice predictability of system behaviour in general.

C. Explicit-Blocking Guarded Sections

While implicit blocking is made impossible by falling back on a throughout non-blocking synchronised runtime or operating-system, respectively, platform for the processing of guarded sections, *explicit blocking* of processes that need to take the role of a consumer must be possible indeed. A corresponding scenario would be, for example, by simply exchanging the use of P and V in Fig. 2(a), that is, with a sequencer calling P and a requester calling V. Another, maybe more common use case, could be an order series that charges the sequencer with the execution of P and V on its own.

Anyway, explicit blocking within a guarded section may occur either instantaneously or deferred. The former assumes that the sequencer (consumer) receives notification by some requester (producer). In contrast, the latter acts on the assumption that the sequencer can also take

a double role both as consumer and producer. Especially for this particular case, unfavourable line-up of orders that possibly entail signalling operations could deadlock the sequencer. A critical sequence, for example, is a P order and a subsequent V order (1) coexistent on the guard queue and (2) applied to the same semaphore. If the P causes the sequencer to block instantaneously, potential deblocking by downstream execution of the subsequent V order may be hold off. A way out of this awkward position is to let the sequencer block only in case of a drained guard queue, thus, defer blocking until the sequencer is empty running and would relinquish guarded-section control in any event.

As the correct procedure is application dependent, both kinds of explicit blocking are provided.[2] A third option, which is currently investigated, is to create an "on-the-fly context" for the sequencer to enable awaiting of notifications concurrently to serialised order processing. Corresponding to a *continuation* [21], a minimal activity medium is spontaneously generated and instructed to take a part in the further processing of pending orders. Suchlike sequencer "offspring" then is in charge of attempting to resume guarded-section execution when the notification has been produced and the sequencer "original" proceeds order processing.

Using sequencer continuations instead of sequencer blocking is attractive as it maintains run-to-completion semantics of processes inside a guarded section. It also renders fiddly measures for safely releasing guarded sections during sequencer wait time unnecessary, such that lost-wakeup of the sequencer will never occur—and keeps guarded sections *event-based*.

IV. Implementation

A dynamic and a static variant of guarded sections were implemented in a completely wait-free manner. The former allows for any number of processes (in terms of threads) and future objects, its implementation is shown in Fig. 3. Note also that the application-specific code of the algorithms shown for the non-blocking and direct-result guarded sections may be wrapped in a subroutine that gets called indirectly through a pointer (cf. lines 1(b).3 and 2(b).3). This would allow for multiple critical sections protected by the same guard.

[2]In a nutshell: made explicit in the guarded-section code, a *phase* operation indicates instantaneous blocking and a *trail* operation defers blocking to the moment when CLEAR drained the guard queue.

```
1: function VOUCH(guard, order)
2:     ENQUEUE(guard, order)
3:     task ← 0
4:     if FAS(guard.flag, 1) = 0 then
5:         task ← DEQUEUE(guard)
6:     end if
7:     return task
8: end function
```

(a) Wait-free entry protocol.

```
1: function CLEAR(guard, adopt)
2:     guard.flag ← 0
3:     task ← 0
4:     if adopt then                ▷ take a part as sequencer
5:         if ¬EMPTY(guard) then
6:             if FAS(guard.flag, 1) = 0 then
7:                 task ← DEQUEUE(guard)
8:             end if
9:         end if
10:    end if
11:    return task
12: end function
```

(b) Wait-free exit protocol.

Fig. 3. Sequence control of guarded sections.

A. Sequence Control

VOUCH implements the entry protocol, which maintains a linked-list of *order* objects (line 3(a).2). The atomic *fetch-and-store* (FAS)[3] instruction in line 3(a).4 ensures that only one thread at a time enters and occupies the guarded section. The respective occupant receives the initial set of parameters (*task*), all other competing threads get *0* on return and will bypass the guarded section (cf. lines 1(b).1 and 2(b).1).

CLEAR implements the exit protocol. In line 3(b).2 the guard is released, followed by a check for an empty guard queue (line 3(b).5). If the queue is filled, the current thread (now no longer occupant) attempts to reoccupy the section by means of FAS. In case of success, this very thread becomes sequencer: the next *task* is retrieved from the queue and the outer guard-loop continues (cf. lines 1(b).4 and 2(b).5). Otherwise, some other thread entered the guard and, maybe, will be in charge of further order sequencing.

B. Race Hazard

Overlapped execution of the entry and exit protocols against the background of such processing patterns has

[3]GCC intrinsic `__sync_lock_test_and_set(ref, val)`.

potential of the *lost-update problem*, the prevention of which needs special care. A lost-update may occur when an enqueued item is ignored and no thread executes the outer guard-loop. The implementation shown in Fig. 3 effectively prevents this problem. All threads executing VOUCH first enqueue their order and strictly after that try to set the guard flag. On the sequencer side, that is within CLEAR, the flag is first reset and then the queue is checked: reverse order would make it possible to have a refilled queue although the check indicated an empty queue (*lost-wakeup problem*). Either an enqueuing thread enters the guarded section by oneself and dequeues the next order or the sequencer notices that the queue is not empty before trying to set the flag.

C. Queue Operations

As mentioned before, the specific processing pattern of guarded sections supports a wait-free synchronised queue. This pattern causes a multiple-enqueue/single-dequeue mode of operation. Fig. 4 shows the corresponding queue operations. Enqueuing (lines 4(b).2–4) follows the same pattern as the MCS queue-based lock [11] and uses FAS. To prevent spinning in the dequeue operation, as with the MCS algorithm, a different dequeuing technique is used. This technique relies on a *dummy* element in the queue. An empty queue therefore always contains a single element. In DEQUEUE, the head pointer is advanced if the head element is followed by another element. Lines 4(b).8–15 take special care for a possibly dequeued dummy element, which needs to be put back on the queue (line 4(c).9). Afterwards, if the queue is still filled, head is advanced to it and the old head-pointer value is returned.

As there is always exactly one dequeuer, only race conditions with respect to simultaneous enqueue operations have to be investigated. The critical machine word that might be accessed simultaneously is the link pointer in the last item. Critical statement in ENQUEUE is line 4(b).4, while lines 4(c).3 and 4(c).10 make up the critical DEQUEUE statements. In both cases the value read is used to determine if the queue is empty. Assuming atomic write operations, the dequeuer will see either an empty queue and return or the next valid item. As writing to the link pointer is really the last operation in ENQUEUE, the dequeuer will never see an invalid item. Further, no dequeue will be performed if not at least two elements are in the queue (e.g., dummy plus useful item). Therefore, no data will be written to already dequeued elements that, thus, can be freed immediately. This eliminates the need for hazard pointers

```
1: dummy.next ← 0
2: head ← ref dummy
3: tail ← ref dummy
```

(a) Queue initialisation.

```
1: procedure ENQUEUE(item)
2:     item.next ← 0
3:     prev ← FAS(tail, item)
4:     prev.next ← item
5: end procedure
```

(b) Add element to the queue (FIFO).

```
1: function DEQUEUE
2:     item ← head
3:     next ← head.next
4:     if next = 0 then
5:         return 0
6:     end if
7:     head ← next
8:     if item = ref dummy then
9:         ENQUEUE(item)
10:        if head.next = 0 then
11:            return 0
12:        end if
13:        head ← head.next
14:        return next
15:    end if
16:    return item
17: end function
```

(c) Remove element from the queue (FIFO).

```
1: function EMPTY
2:     return head.next = 0
3: end function
```

(d) Check for drained queue.

Fig. 4. Multiple-enqueue/single-dequeue wait-free queue.

[22] or garbage collection. As can be seen, the discussed entry and exit protocols forgo direct and indirect loop constructs and, thus, are completely wait-free.

D. Alternative Solution

In the static variant, the ENQUEUE and DEQUEUE operations map to bit operations on a fixed-length bitset, where the bit position is derived from the thread identification. The necessary bit operations are carried out by using atomic OR/AND processor instructions. As a consequence, the provided algorithms are also wait-free in the static case. Besides the outer guard-loop, no further loops are used and only a single atomic bit-instruction is carried out per queue operation.

TABLE I
BASIC OVERHEAD, UNCONTENDED CASE.

Algorithm	Cycles
Dynamic	128
Dynamic NB	116
Static	84
MCS-Lock	39
Read-Spinlock	39

As the present dynamic variant is limited to FIFO-order, scheduling interference may occur. This is not the case with the static variant, which namely implements a priority-based protocol. Thereby, the priority of a request to pass a guarded section corresponds to the bit position derived from the thread identification and, thus, reflects the thread priority. The passage request with highest priority will be executed first.

V. EVALUATION

As a proof of concept, prototype guarded sections are currently made available as *guest-level implementation* above Linux. The overhead of contended and uncontended guarded sections were measured. Timings include direct-result and non-blocking guarded sections, dynamic and static variants.[4] The static configuration employs priority-based execution of requests. Measurements were performed on a 80 core Intel Xeon E5-4640v2 server running at 2.2 GHz partitioned into four cache-coherent sockets with 10 physical or 20 logical cores (through hyper-threading), each.

Since POSIX-semaphores induce a very high overhead, receive of signals as to future objects was done by spinning. As a frame of reference, numbers for the MCS spin-lock [11] are given. These locks shall perform well under high contention. Table I shows the overhead for uncontended acquisition and release of a guarded section and respectively a MCS lock/unlock pair. The critical section itself was void. Processor cycles were averaged over 10^5 executions with hot caches.

Measurements were also performed for high-contended cases using up to 64 processing elements (i.e., cores), as the implementation of the static variant allows for exactly that maximal number of threads. Fig. 5 shows the results in number of cycles. In general, performance decreases dramatically for high contention in all cases. Especially when crossing socket boundaries

[4]Due to the lack of blocking critical sections in the benchmarks, blocking guarded sections were not evaluated.

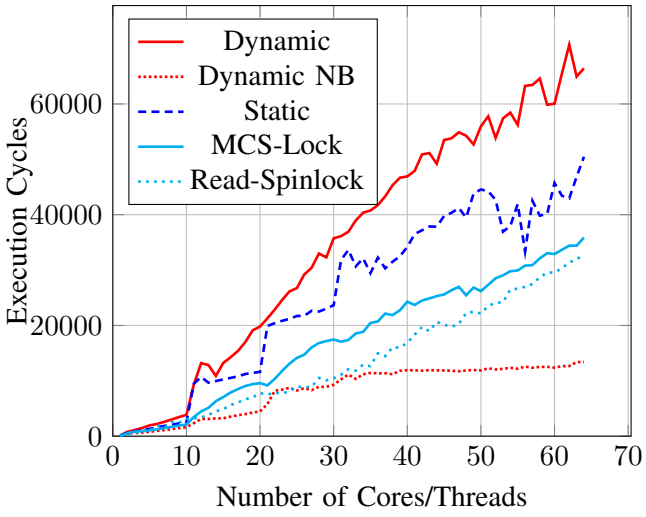


Fig. 5. High contended case, range 1:64 cores.

as can be seen in more detail in Fig. 6, where an abrupt rise of the number of cycles can be stated 10 cores off. This boost is known from earlier experiments

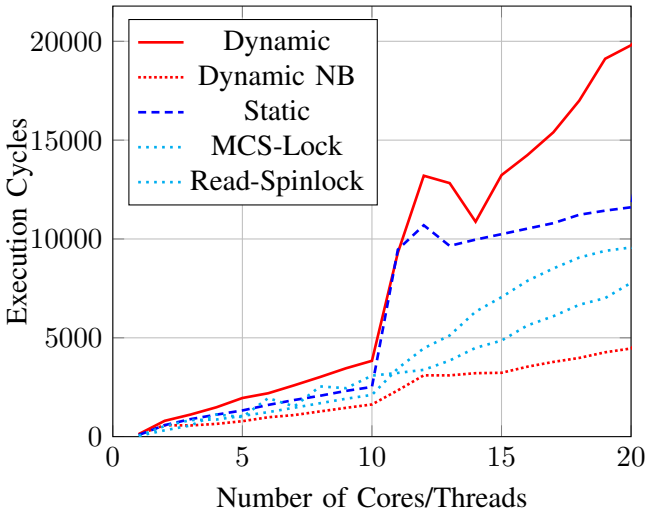


Fig. 6. High contended case, detail, range 1:20 cores.

in which hyper-threaded cores were allocated first before allocating processors/cores on a different socket. The overhead seems to be very similar and is within measurement accuracy.

As can be observed, the overheads for the dynamic and static variants, when directly awaiting the result of the guarded section, are higher compared to the MCS-lock version. However, one has to keep in mind that guarded sections are not only a drop-in replacement for locks but rather provide all the benefits mentioned as to non-blocking synchronisation. MCS-lock is a blocking

technique and, thus, does not feature any of these properties. Contrariwise, the non-blocking dynamic variant employs the best performance, since no thread has to wait on results and threads are either enqueuing further orders or execute requests in sequence. The static non-blocking variant could not easily be measured under high contention, because the number of requests is limited to the number of threads and therefore no high contention scenario can be generated for thousands of iterations. However, as the direct-result version of the static variant is faster than the dynamic variant, similar behavior can be expected in the non-blocking high-contention case.

VI. Related Work

The presented concept was inspired by the early work on *guarded commands*, which were "introduced as a building block for alternative and repetitive constructs that allow nondeterministic program components" [23]. But guarded sections are neither a programming-language construct nor are they supported by a dedicated compiler. Rather, they can be considered and are very well suited as run-time system support for a guarded-command language. Resolution of the boolean expression that must evaluate to true in order to make a guarded command eligible for execution would be a typical case for a guarded section as introduced here.

Synthesis used *procedure chaining* [24] for serialised execution of program sections when non-blocking synchronisation proved to be impractical. This concept avoided synchronisation inside chained procedures, which also applies for non-blocking or direct-result, respectively, guarded sections described in the present work. In contrast to Synthesis, by means of explicit-blocking guarded sections unilateral (logical/conditional) synchronisation is supported as well. Beyond that, in Synthesis, procedure chaining fell back on lock-free synchronised queues, whereas guard-queue synchronisation is wait-free.

Kernel-level synchronisation in the PEACE parallel operating system was based on the serialised execution of so called *epilogues* [25]. Epilogue execution was triggered by first-level trap/interrupt handling, referred to as *prologue*, but carried out synchronous with the current process in kernel mode. Prologues released and acted on behalf of user-mode processes caused instantaneous execution of dedicated epilogues, thus, starting a kernel-mode process. In contrast, prologues catching a kernel-mode process caused deferred epilogue execution. In doing so, the respective epilogue was added to a global queue and removed therefrom whenever the kernel-mode process finished execution of a predecessor epilogue. In contrast to guarded sections, only a single "epilogue guard" (for the whole PEACE kernel) was supported.

A refinement of epilogue-queue management was made for the PURE embedded operating system [26]. Dedicated use in an interrupt-handling context, only, gave rise to a stacking arrangement of multiple-enqueue/single-dequeue operations. This constrained overlapping pattern facilitates lock-free synchronisation of the queue by exclusive use of atomic memory load/store instructions. In contrast to the present work, the epilogue-queue implementation did not scale-out to multi-core/processor systems.

Another similar approach is *flat combining* [27], which is based on coarse locking. A single thread holds a lock when it performs combined access requests to a critical section, while simultaneously requesting threads will block at that lock for the duration of the respectively serialised operations. In contrast, guarded sections never cause arriving preferential threads to block. Furthermore, due to its lock-based solution, flat combining is prone to scheduling interference, priority inversion, and deadlock.

Main motivation behind flat combining was to lower the cost of synchronisation especially as to fine-grained shared data-structure operations. A direct consequence of having several of those operations serialised by a single thread is a significant decrease in overall cache traffic because of a reduction in number and frequency of invalidation messages. This observation led to *remote core locking* (RCL, [28]). Very similar to flat combing, RCL has a single thread in charge of executing a particular critical section if contention at entrance to that critical section exceeds a certain threshold. In difference to flat combing, RCL always seizes a processor/core dedicated of critical-section execution. In contrast to RCL, guarded sections are free of implicit blocking of threads at entrance to an "off-loaded" critical section. But, common to RCL, a similar approach will be considered next to roll a sequencer out to a private core for cache-friendly execution of guarded critical sections subject to high contention.

VII. Conclusion

The uniqueness of guarded sections is that preferential processes never block at entrance to a critical region although only one process at a time is allowed to pass through. Requests for passing a guarded section though are processed in serial manner, but not necessarily the processes that issued these requests. This is the fundamental difference to conventional critical sections, where

mutual exclusion is realised in a way that bestows any process a potential delay at critical-section entrance.

Guarded sections are a means to an end, namely to increase parallelism in non-sequential programs of legacy but also "from scratch" new software. Blocking of processes is reduced to logical/conditional synchronisation and, thus, happens exclusively unilateral according to the data flows between the processes. Guarded sections combine the convenience of critical sections in terms of software structure with the power of non-blocking synchronisation in terms of performance. They are a structuring aid by means of which wait-fee synchronisation of the guarding operations is supported.

The absence of multilateral blocking synchronisation is to the best advantage for real-time systems, above all of those that follow an event-triggered mode of operation. All platform operations used for the implementation of guarded sections are void of priority violation and inversion. Although solutions to these problems are well-known and very well established, none of that is needed for non-sequential time-dependent programs based on guarded sections—unless application code performs (multilateral) blocking synchronisation on its own. Doing without reduces complexity as well as sources of background noise, interference, and overhead from real-time systems. Direct consequence therefrom is improved predictability not only of the system software but also application programs.

Run-time system support for guarded sections is still in its infancy, just as an operating system built around that concept. First experiments on a 80-way multi-core system are encouraging that guarded-section based software systems achieve predictable performance as to the properties of the underlying hardware. Besides tuning, future work focusses on sequencer "off-loading" to spare processor cores and real-time capable energy-awareness of the guarding operations.

APPENDIX A
RUN-TIME SUPPORT SYSTEM

In order to get an impression on the structural as well as computational complexity of a run-time environment needed to support direct-result and blocking guarded sections, the prototype of a mostly *native implementation* of a (time-) *predictable operating-system executive* (POSE) is sketched in the following paragraphs. This implementation is suited for a "bare-metal system" that controls processes of a real processor (e.g., Intel Xeon) as well as for a "guest-level system" that assumes some host operating system (e.g., Linux) underneath and whose process instances serve as virtual processors. POSE features *featherweight processes* in the form of threads sharing a single address space, a latch-based signalling mechanism for the support of logical/conditional synchronisation of simultaneous processes, full-preemptive process scheduling using static priorities, processor dispatching (i.e., context switching), and processor idle-loop control. It is sample of a *threading infrastructure* that gives *wait-free* progress guarantee to processes. Presentation happens top-down, stepwise closing the *semantical gap* between direct-result or blocking, respectively, guarded sections at the top and a physical/logical processor at the bottom.

```
1: procedure PROVE(bond, data)
2:     bond.value ← data
3:     bond.state ← {KEPT}
4:     V(bond.sema)
5: end procedure
```

(a) Deliver promised value (no-wait send).

```
1: function EXACT(bond)
2:     P(bond.sema)
3:     if bond.state ∋ {KEPT} then
4:         return bond.value
5:     else
6:         return undefined
7:     end if
8: end function
```

(b) Retrieve promised value (blocking receive).

Fig. 7. Future control on basis of a signalling semaphore.

A. Future Control

The major difference between non-blocking and direct-result guarded sections is logical or conditional, respectively, synchronisation of interacting processes. As described in Sec. III-B, the *future* concept is used to synchronise requester processes on the availability of data produced by a sequencer process. By using PROVE, the latter process makes the produced data explicitly available to the former process that, in turn, uses EXACT to receive that data for consumption purposes. The implementation of this concept is shown in Fig. 7, which goes back on a simple *signalling* (i.e., binary) *semaphore* for the synchronisation of the two types of processes and a *data container* (i.e., promise) for intermediate buffering of a result value.

Clearly, the memory footprint of a future object is problem-specific and the structure behind is inherently determined by some application-level data type. This particularly effects the actions to buffer (line 7(a).2) and

```
1: procedure P(sema)
2:     LATCH(sema)
3: end procedure
```

(a) Consume signal, conditionally wait.

```
1: procedure V(sema)
2:     NUDGE(sema.task)
3: end procedure
```

(b) Produce signal and store into latch.

Fig. 8. Signalling semaphore mapped to a software latch.

retrieve (line 7(b).4) the data. The generic and, thus, system-specific part of future control relates to sending (lines 7(a).3–4) and receiving (lines 7(b).2–3) the signal that a promised value has been made available and a successful state change took place.

B. Signalling Semaphore

Main task of process interaction from a systems point of view is hidden inside the P and V operations of the (binary) semaphore. In the example shown (Fig. 8), these operations are mapped to respective operations of a per-process software latch. Using LATCH (line 8(a).2), the calling process receives a signal. The process will be blocked only if the signal to be received has not yet been sent using NUDGE (line 8(b).2).

The mapping shown in Fig. 8 is typical for interfacing to the *operating-system machine level* of a computing system. Here, an invocation of LATCH and NUDGE corresponds to *system calls*—in logical or physical respect, depending on the actual operating principle of an operating system in terms of an abstract processor (i.e., virtual machine).[5]

C. Interior Structure

Beneath that mapping interface, a structure of features is found by means of which time-predictable operation of a process management subsystem is provided. This structure shows Fig. 9. Shaded areas likewise embrace features of a specific layer, the number of which is indicated left-justified. Except the bottom layer, which is made of special instructions of the particular (hardware) processor, higher layers consist of procedures or functions, respectively, implemented in some programming

[5]That is to say, the system call happens either as a normal procedure call (logical) in case of a *library-based operating system* or through specialised processor instructions (physical; software trap or SYSENTER/SYSLEAVE-style of actions, resp.) for contemporary *kernelised operating systems*. The former assumes integrated and the latter assumes isolated protection domains.

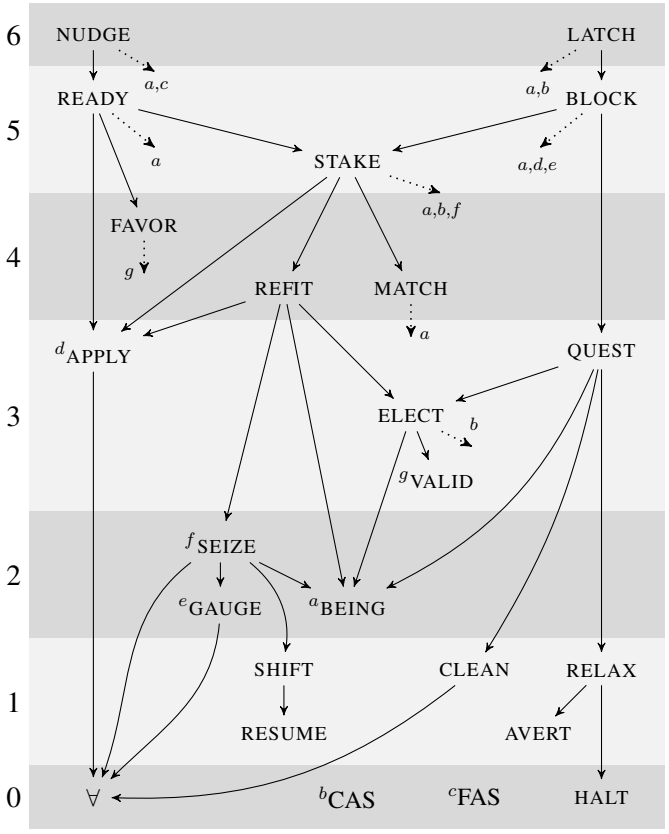


Fig. 9. Functional hierarchy of POSE.

language. The meaning of these layers is as follows (listed from top down to bottom layer):

6 logical/conditional synchronisation
5 basic process control
4 priority control and inheritance
3 process scheduling and selection
2 process dispatching
1 processor control
0 elementary (i.e., atomic) operations

Layers 0 and 1 as well as function BEING and all those operations directly referring to the "for-all" operation (∀, at layer 0) are hardware dependent and, thus, come in different versions as to the respective (physical) processor in use. Except for the memory model, all other layers or operations, respectively, are hardware independent. Nonetheless, even those hardware-independent features show a choice of options that manifest in specific software versions as to different modes of operation required by the particular application scenario.

The solid arrows indicate call relations, whereby a "call" may have been resolved statically at compile time by manual instruction or automatically in the course of a compiler optimisation pass. Same holds to the dotted

```
1: procedure LATCH(mark)
2:     task ← BEING(ONESELF)
3:     if CAS(task.latch, FALSE, ∞) then
4:         BLOCK(mark)
5:     end if
6:     task.latch ← FALSE
7: end procedure
```

(a) Latch process onto a signal.

```
1: procedure NUDGE(name)
2:     task = BEING(name)
3:     if task ≠ 0 then                ▷ process is known
4:         if FAS(task.latch, TRUE) = ∞ then
5:             READY(task)
6:         end if
7:     end if
8: end procedure
```

(b) Signal a process.

Fig. 10. Software latch operations as system calls.

arrows, which were introduced for readability purposes, only, to avoid cutting across. Small types are used as decorations and point to the correspondingly annotated operation symbols.

Fig. 9 shows the call relations additionally in terms of a *functional hierarchy* [29]. Purpose of this combined representation is to embody that, for its own correct behaviour, a higher arranged operation depends on the correct functioning of a pointed-to and, as the case may be, called lower arranged operation. Note that, in general, a call relation alone does not implicitly indicate a dependency relation as to correct program behaviour according to a given specification [30]. The representation in Fig. 9, however, assumes both types of relation. Meaning and interplay of these operations are explained in the next subsections by following a top-down practice.

D. Software Latch

The major effort of semaphore signalling comes with the implementation of LATCH and NUDGE (Fig. 10). These two operations provide a functional interface to a per-process software latch. It is assumed that every process instance contains a "latch register" for capturing timing signals directed to the particular process physically represented by that very instance. Quite similar to hardware, the software latch remembers a signal by means of a dedicated write operation (line 10(b).4) until it gets cleared by an explicit read operation (line 10(a).6).

In order to facilitate these functions, a very basic action (needed in almost any operation further discussed) is reading the *instance pointer* to the current (line 10(a).2) or a named (line 10(b).2) process, in short: the *process pointer*. This pointer gives access to the central data structure that reflects the physical characteristics of a process and, in particular, contains the per-process latch. In operating-system terms, that data structure is generally known as *process control block* (PCB).

Technically, the signal is implemented as a "sticky bit" that is sticked to a process using NUDGE and unsticked therefrom using LATCH. In addition to this function and, as consequence, different from hardware, the software latch also takes care of process control. It blocks a process if a signal has not yet been latched in the moment of a read action (line 10(a).4) and, vice versa, unblocks a possibly blocked process in the moment of a write action (line 10(b).5). In this sequence of actions, special care must be taken to prevent the potential of a *lost wakeup* of a tentatively blocking process (line 10(a).3–4). In the given case, this type of race condition is tackled on a state-machine basis as will be explained next.

E. Process Handling

The programmed action of blocking a process (e.g., line 10(a).4) indicates a logical *preemption point* in the context of *logical synchronisation* on the occurrence of a particular event (i.e., timing signal). In principle, any process always needs to take actions on its own to relinquish its processor to some other process. In addition, given a preemptive system, a process can be forced into closing down by means of an external event such as an *interrupt request* (IRQ) or provision of a process (e.g., line 10(b).5). Thereby, a graduation is often made into "preemptive" and "full preemptive". The latter implies processor relinquishment at any time and place, while the former enables revocation of the processor only at specific places shown separately. Most notably, the full preemptive case further implies not only that *response time* is shorter and the degree of concurrency is higher but also that, of necessity, not a single sequential piece of code (i.e., action sequence) is existent in the program—the exception proves the rule, as will be explained later. POSE corresponds to such a full-preemptive approach.

1) Blocking of a Process: At first sight, blocking a process is a straightforward action: select a process from a pool of ready-to-run candidates and then perform a context switch. At second sight, however, things need to be handled in a more differentiated way. An important aspect thereby is that the process going to block may be the last one available for execution on its processor at all. In this case, the process needs to conduct *idle-loop*

control. No less important is *priority loyalty*: the period from the point of having selected the next ready-to-run candidate until the point of switching is prone to *priority inversion*, which must be either prevented or controlled and bounded in time by design. Yet another aspect is that the blocking but still running process may also be immediately re-selected for continuation. A process switch to itself becomes possible—unless prohibited by the switching method (cf. Fig. 16)—but means bare *overhead* and, thus, should be avoided.

Prompt re-selection may happen to a blocking or "running blocked", respectively, process that becomes simultaneously unblocked and, thus, available again due to some external event. In the case of the software-latch implementation just described, simply assume the simultaneous execution of LATCH and NUDGE (cf. Fig. 10) in a way such that the execution of READY overlaps the execution of BLOCK (cf. Fig. 11) in time. Note that this pattern of overlapped execution may cause a "running ready" or "running blocked ready" process: in other words, a still run-for-block process being also a ready-to-run candidate that, accordingly, can be re-selected for execution. Key point in this scenario is that, under no circumstances, re-selection of a process in that particular state must not lead to its prompt continuation on another processor.

As shown in Fig. 11(a), after quest for a follow-up process (line 11(a).4), the current process is switched off only if BLOCK gets aware of another ready-to-run candidate that can be switched on (lines 11(a).7–8), otherwise only "recalibration" as to process-specific system state takes place (line 11(a).6, cf. also Fig. 15(c)) and the process continues operation in that it seamlessly still engrosses the processor. Before QUEST, the current process actually applies for blocking (line 11(a).3), that is to say, the event expected by the current process is remembered for housekeeping purposes and the state of the current process is extended by $BLOCKED$ (cf. Fig. 13(d)). At the same moment, the current process becomes *logically blocked* (i.e., "running blocked"). This transition means that the process is not yet physically suspended but right there on the course, unless not redirected, and announced its readiness to receive a timing signal related to the blocked-on condition ($mark$).

In POSE, a process state is modeled as a *union set* of one or more substates at a time. In the case of BLOCK (more specifically, line 11(a).3), for example, the *composite state* of the current process actually depends on the point in time APPLY becomes effective. The following state settings and action occurrences are possible:

1) $\{RUNNING\}$, neither BLOCK nor READY,
2) $\{RUNNING, READY\}$, READY only,

```
1: procedure BLOCK(mark)
2:     self ← BEING(ONESELF)
3:     APPLY(self, mark, {BLOCKED})
4:     next ← QUEST(R2R)  ▷ ready-to-run candidate
5:     if next = self then          ▷ continue process
6:         GAUGE(self)
7:     else                         ▷ other process, switch
8:         STAKE(next)
9:     end if
10: end procedure
```

(a) Cessation of a process.

```
1: procedure STAKE(task)
2:     MATCH(task)                  ▷ inherit priority
3:     last ← SEIZE(task)
4:     trim ← last.state
5:     if trim = {BLOCKED, READY} then
6:         self ← BEING(ONESELF)
7:         if last.level.soft ≤ self.level.firm then
8:             if CAS(last.state, trim, Ψ) then
9:                 APPLY(self, R2R, {READY})
10:                SEIZE(last)
11:            end if
12:        end if
13:    end if
14:    REFIT(R2R)                   ▷ restore priority
15: end procedure
```

(b) Appointment of a process.

```
1: procedure READY(task)
2:     self ← BEING(ONESELF)
3:     if ¬FAVOR(task, self) then
4:         APPLY(task, R2R, {READY})
5:     else                         ▷ high-priority candidate
6:         APPLY(self, R2R, {READY})
7:         STAKE(task)              ▷ switch process
8:     end if
9: end procedure
```

(c) Provision of a process.

```
1: function FAVOR(task, this)
2:     if this.level.soft > task.level.firm then
3:         return VALID(task)
4:     end if
5:     return FALSE
6: end function
```

(d) Check for high-priority process.

Fig. 11. Basic process control.

3) $\{RUNNING, BLOCKED\}$, BLOCK only,
4) $\{RUNNING, BLOCKED, READY\}$, both.

The substates, thus, not only reflect on the execution history of specific actions to block or ready a selected process but also indicate a particular overlapping scenario. They give decisive hints in order to achieve non-blocking (wait-free) synchronised operation of processes roaming through the whole infrastructure.

In priority-controlled systems like POSE, the whole sequence of actions from having selected a ready-to-run-candidate (by using QUEST) and detected that this candidate differs from the current process (line 11(a).7) and, thus, is to be switched on (by using STAKE or SEIZE, resp.) forms a *critical path*. This path turns out to be prone to *priority violation* that eventually results in *priority inversion*. Note that, by definition, the current process is the highest-priority one on a particular processor. When that process blocks, the priority of the follow-up process returned by QUEST will be lower or equal[6] to the priority of the current (blocking) process that performed the search for its own successor.

Assume that, simultaneously to the actions on that critical path, the current process is made available again because of a succeeding NUDGE (Fig. 10(b)). Then, by means of READY (Fig. 11(c)), the still "running blocked" current process becomes "running blocked ready" but has to proceed as by instruction of BLOCK (lines 11(a).4–8). A just logically blocked high-priority process made ready-to-run (more precisely, ready-to-continue) in the meantime is forced to switch to a low-priority ready-to-run process: priority violation. In the further course, the low-priority process runs although a high-priority process is ready-to-run and, thus, available: priority inversion. This problematic situation endures until the next rescheduling event as to that particular processor occurs. Not till then will the high-priority ready-to-run process be recognised and resumed. The delay for that rescheduling event as to the particular high-priority ready-to-run process must be predictable and bounded in time in order to prevent *uncontrolled priority inversion*.

By falling back only on non-blocking (wait-free) synchronisation, the described problem cannot be solved by the blocking process alone when executing on that critical path. This is in contrast to blocking synchronisation where the whole critical path constitutes a *critical section* that excludes any simultaneous READY action. Here, the high-priority blocking process enters the critical section within which it will select its low-priority follow-up process and eventually switches off. The switched-on low-priority process is in charge of leaving the critical section, only. Yet in the moment when this happens, a potentially delayed READY action becomes effective, pre-empts the switched-on process, and switches back to the high-priority origin process.

Note that priority violation cannot be eliminated in such a scenario. However its adverse effect of priority inversion can be smoothed out. This also applies to the non-blocking (wait-free) synchronised case shown in Fig. 11(b). In that case, any process switch (i.e., SEIZE) is initiated through a call of STAKE (cf. lines 11(a).8 and 11(c).7), namely to assure that all switched-on processes run through the same protocol in order to overcome priority inversion. When looking closely at Fig. 11(b), "normal" priority inversion[7] is avoided neither but its period is limited by means of *priority inheritance* and, thus, uncontrolled priority inversion is made impossible.

At entrance of STAKE and if applicable, the higher priority of the current process is inherited to the process that will be switched on next (line 11(b).2 or Fig. 12(a), resp.). If inheritance became effective, the original priority of the respective process will be restored again at exit from STAKE (line 11(b).14 or Fig. 12(b), resp.). In between, the checks are made in order to immediately resume the origin process (line 11(b).10) if necessary. Resumption of that process takes place if it is "blocked ready" (indicating that a blocking process became ready-to-run in the meantime) and of higher priority than the current process. By means of an atomic *compare-and-swap* (CAS), a specific Ψ-state is assigned to the corresponding process instance (line 11(b).8) if it is still "blocked ready" and, thus, was not selected for continuation by some other rescheduling action in the meantime. The Ψ-state makes sure that the process (a) cannot be re-elected before its next cycle and (b) is pending with the processor that performed the election: $\Psi = \{PENDING\}$. If CAS succeeds, the current (low-priority) process makes itself a ready-to-run candidate for processor allocation and resumes the former (high-priority) process (lines 11(b).9–10).

Within STAKE, each process unconditionally switches off and resumes at its *normal preemption point* (line 11(b).3). In addition, when a low-priority process detects a high-priority ready-to-run origin process, it condi-

[6]The POSE variant presented does not arrange for equal priorities (cf. ELECT, Fig. 13(b)). It is assumes that all processes controlled by prioritised scheduling at a time have a *unique priority*.

[7]That is to say, as given in the particular case, a high-priority ready-to-run process has to wait for a low-priority running process.

```
1: procedure MATCH(task)
2:     self ← BEING(ONESELF)
3:     if self.level.soft < task.level.firm then
4:         task.level.soft ← self.level.soft
5:     end if
6: end procedure
```

(a) Conditionally transfer priority.

```
1: procedure REFIT(pool)
2:     self ← BEING(ONESELF)
3:     if self.level.soft < self.level.firm then
4:         from ← self.level.soft
5:         self.level.soft ← self.level.firm
6:         next ← ELECT(pool, from, self.level.soft)
7:         if next ≠ 0 then   ▷ middle-priority process
8:             APPLY(self, R2R, {READY})
9:             SEIZE(next)
10:        end if
11:    end if
12: end procedure
```

(b) Conditionally restore priority.

Fig. 12. Priority inheritance protocol.

tionally switches back (lines 11(b).8–10) at its *escape preemption point*. This causes the high-priority process to continue (line 11(b).3) and, as the in turn last (i.e., low-priority) process is not in $BLOCKED$ state, return from STAKE. Actually, a "jump back" (from line 11(b).10 to line 11(b).3) takes place indirectly due to *forced cooperation* of the low-priority process.

It is worth to note that, as explained above, these two process switches will also have to happen in case of a blocking synchronised (pessimistic) solution for that critical path. But in contrast to such a solution, which works implicitly and involves central process scheduling, the non-blocking synchronised (optimistic) method as followed by STAKE makes the protocol explicit. Even more, the shown solution (Fig. 11(b)) to the priority-inversion problem is not only less overhead-prone but the residual overhead is also constant in time, determinable by static program analysis and, thus, predictable.

2) Provision of a Process: With prioritised full-preemptive process scheduling, making a process available may entail a process switch as well (Fig. 11(c)). This comes about if the priority of the current process is lower (i.e., has a larger numerical value in the case given) than the priority of the process to be made available. The respective checking is a major part of FAVOR (Fig. 11(d)). If the current process is to be favoured over the applied process (line 11(c).3), the latter only becomes a ready-to-run candidate (line 11(c).4) and the current process continues. Otherwise, the current process becomes a ready-to-run candidate and a process switch takes place (line 11(c).6–7).

Keep in mind that a provided process (i.e., being in state $READY$) could be immediately selected for continuation on the same or another processor due to pseudo or true parallelism, respectively. But note that selection must be rethought for processes that are in $RUNNING$ state at the same time. A process whose state owns the subset $\{RUNNING, READY\}$ is right on the way to relinquish its processor and, when succeeded, become a ready-to-run candidate for processor allocation. Such a process could be very well re-selected on the same processor while still running, although its resumption then would have absolutely no effect but pure overhead as the process would switch to itself and, nevertheless, join the pool of ready-to-run candidates.

Just in case of true parallelism (e.g., given a multi-core system), provisioning of a process must prevent duplicate execution of this very process on different processors. Concerning this matter, FAVOR (Fig. 11(d)) takes care for the respective checks. Assuming that a process in state $\{RUNNING, BLOCKED\}$ (i.e., which has completed line 11(a).3 of BLOCK but not yet relinquished its processor) is made available through READY by a process residing on a different processor. Further assuming that the readied process is of higher priority than the readying process. In a full preemptive system, thus, the readied process will pre-empt the readying process on its processor (line 11(c).5). As a consequence of this, duplicate execution of the just now provided process will happen. In order to prevent such a defective operation, FAVOR particularly checks for validity of a high-priority process. A process to be provided is termed to be valid in such a situation if it shares the same *execution scope* with the providing process. That is to say, in that particular scenario, both processes must reside on the same processor for preemption to become effective. This validation procedure is also an important feature of process selection and will be explained next.

3) Selection of a Process: When a process decides to relinquish its processor to another process, a ready-to-run candidate needs to be selected and switched on. Process selection is controlled by QUEST (Fig. 13). Once called, this function always returns a valid process pointer to a ready-to-run candidate that is eligible for processor allocation. But this also means that QUEST persists in looking for available processes even if there are no

ready-to-run candidates. That is to say, QUEST is also the mechanisation of the *idle loop* of an operating system.

Central action in the selection procedure is the call of ELECT (line 13(a).5), which chooses the highest priority process from all ready-to-run candidates. If there is no such candidate, a changeover into standby, sleep, or halt mode of the processor is initiated. This mode change is performed by RELAX (line 13(a).7), with operator Λ aggregating knowledge on how to "drug" the processor according to a situate treatment plan. Switchback into "normal" operating state takes place only when the processor is waken up by some external event (e.g., an IRQ or afar-addressing of a monitored shared-memory area). Note that the idling process keeps its priority. At first sight, putting down to lowest priority (∞) would have been appropriate to let pass any other process. The solution of which introduces overhead in the "normal" (i.e., non-idle) path,[8] for which reason priority lowering is intentionally not considered in a situation where logically idleness of the processor can be adequately utilised by polling for available low-priority processes.

Of vital importance is to note that the sequence of idle-loop actions discloses another potential race condition that, again, may cause the lost wakeup of a process in case the problem remains untreated. Assume that ELECT indicates nil return and the process is right on the way to call RELAX next (lines 13(a).5–7). If in that situation an external event takes place at once as a result of which a ready-to-run process—of lower or same priority compared to the idling process—is made available, then this very process gets lost. What makes matters even worse, the checkless processor mode change as prescribed by the program may never be reversed.

In order to prevent this erroneous function, a process indicates its possibility to become idle and, thus, control processor-mode changeover (line 13(a).4) *before* both searching for a ready-to-run process and checking a nil return of ELECT. This help by QUEST assists RELAX to prevent the then idle process from shutting down operation at just the wrong moment (cf. Fig. 14).

Process selection lies in responsibility of ELECT (Fig. 13(b)), which performs a "table walk" to capture a ready-to-run candidate. Nil return of this functions indicates an apparently empty pool of ready-to-run processes at the time of the recent table walk. This indication may be a "false negative" as, simultaneously to the table walk, a process could have been made available in the meantime. The higher-level loop to quest for an

```
1: function QUEST(pool)
2:     self ← BEING(ONESELF)
3:     repeat
4:         CLEAN(CPU)
5:         next ← ELECT(pool, self.level.firm, N)
6:         if next = 0 then
7:             RELAX(CPU, Λ)
8:         end if
9:     until next ≠ 0
10:    return next
11: end function
```

(a) Process-level idle cycling.

```
1: function ELECT(pool, from, to)
2:     assert 0 ≤ from ≤ to − 1 < N
3:     self ← BEING(ONESELF)
4:     for rank ← from, to − 1 do
5:         next ← pool[rank]
6:         trim ← next.state
7:         if trim ∋ {READY} then
8:             if VALID(next) then
9:                 if CAS(next.state, trim, Ψ) then
10:                    return next
11:                end if
12:            end if
13:        end if
14:    end for
15:    return 0
16: end function
```

(b) Choose next low-priority ready-to-run process.

```
1: function VALID(next)
2:     if next.state ∋ {RUNNING} then
3:         if next.scope ≠ CPU then
4:             return FALSE
5:         end if
6:     end if
7:     return TRUE
8: end function
```

(c) Validate ready-to-run process.

```
1: procedure APPLY(task, pool, trim)
2:     assert pool[task.level.firm] = task
3:     task.event ← pool
4:     task.state ← ∀(task.state ∪ trim)
5: end procedure
```

(d) Allocation of a process.

Fig. 13. Process selection and inspection.

[8] At the expense of a short latency for the present high-priority idling process to return from QUEST and continue normal operation, READY (Fig. 11(c)) would have to take care of the "exceptional case" of a lower-ranked idle process to be restored in every respect.

available process (lines 13(a).3–9) compensates for a missed ready-to-run candidate.

The table walk is limited to a specific range (line 13(b).2) and, thus, bounded in time. It proceeds with decreasing priority level (i.e., increasing running index). If an inspected process is ready-to-run and also a valid candidate for processor allocation (lines 13(b).7–8), the attempt is made to claim the respective process instance and return the corresponding process pointer (lines 13(b).9–10). A CAS action ensures that an eligible process can be selected only once at a time and, thus, will never be crossposted to several processors. As CAS is a relative costly operation it will be issued only when required, namely in case of a matching candidate. If the state of that candidate is still unchanged, CAS assigns the Ψ-state to the corresponding process instance as was already commented in context of Fig. 11(b).

A process in Ψ-state must always be allocated to the processor with which it is pending. This state denotes a process that will be switched on in a minute. That is to say, the respective process was successfully selected (lines 13(b).9) and will continue once the current process, under control of which the selection was carried out, detected either self-provisioning (line 11(a).6) or has relinquished its processor (line 11(b).3). Such a process whose continuation has been launched and, thus, is as safe as houses must not be re-selected before having become a ready-to-run candidate again. The processor with which that process is pending is the processor on which the selecting process currently proceeds. Note that, besides ELECT, the Ψ-state is also assigned as part of STAKE (Fig. 11(b)) when switch-back to a meanwhile provided high-priority origin process must be performed (lines 11(b).7–8). Characteristic of the Ψ-state of a "soon running" process is the presence of substate $PENDING$ in the union set.

The action of state transition (line 13(d).4) needs some more attention, as its functioning is fundamental. As aforementioned, the logical process state is modeled as a set of substates. In the case of a *logically blocked* process this effectively means a subset of $\{RUNNING, BLOCKED\}$, as a process will be running when it reaches the action that causes the state transition within BLOCK (Fig. 11(a)). Note that the value of exactly this set union as a subset denotes the idle process. Also note that presence or absence, respectively, of substate $RUNNING$ as a member of that substate places an important role in the selection of a ready-to-run candidate (line 13(c).2). Furthermore, the ∀-operator indicates that the complete state transition (line 13(d).4)

```
1: procedure CLEAN(site)
2:     flag ← ∀(flag \ {site})
3: end procedure
```

(a) Reset interrupt indicator flag.

```
1: procedure LABEL(site)
2:     flag ← ∀(flag ∪ {site})
3: end procedure
```

(b) Set interrupt indicator flag.

```
1: procedure RELAX(site, drug)
2:     old ← AVERT(ALL)          ▷ disable all interrupts
3:     if {site} ∉ flag then
4:         "drug" processor to a greater to lesser extent
5:         HALT                  ▷ enter sleep mode
6:     end if
7:     AVERT(old)                ▷ enable old interrupts
8: end procedure
```

(c) Safely suspend processor core.

Fig. 14. Processor-level idle cycling.

needs to be an indivisible operation in logical or physical terms.[9] Execution of this operator as well as saving its result in memory must ensure *data consistency* "for all" operands (here: the $state$ instance variable) applied.

F. Mode Changeover

POSE idle-loop control happens at two levels of abstractions: namely at (higher) process and (lower) processor level. According to the principle of "separation of concerns", the process level is sensitive to job-creating measures that are suited to keep the technical side running while the processor level is sensitive to energy saving or excessive heat, respectively, and aims at slowdown strike or even dead-time. These different characters are reflected in two dedicated actions: QUEST for process level (Fig. 13(a)) and RELAX for processor level (Fig. 14(c)). Process-level idle-loop control has been extensively discussed above. At this point, the interaction with the processor level shall be in the foreground. Key aspect thereby is to overcome a race condition that may cause process-level idle-loop control from shutting down operation.

[9] Assuming that $state$ is represented in C and as a structure of bit fields of eight bits (i.e., a byte) each. The union operator (∪) then is compiled to an atomic memory byte-store operation. Same holds for the difference operator (\). Reading the whole $state$ set while simultaneously writing one of its set members then delivers a consistent value according to the given memory model. If this model, however, appears to be too weak for the intended semantics of the particular ∀-operator, then a suitable atomic *read-modify-write* instruction must be used instead.

It must be pointed out that the idling cycling of a processor is abrogated only by means of *external processes*. These processes are meant to occur somewhere but not on the idling processor itself. Cases in point are input-output operations carried out by peripherals or state changes caused by other processors (cores) in a shared-memory computing system. A general approach to cancel the idle state is by means of an *interrupt request* triggered by the external process. This model is assumed in POSE. Another and complementing approach, particularly suited for multi-core processors, is a store action afar that targets a monitored shared-memory address range at which a processor core is waiting (i.e., a MONITOR/MWAIT-controlled range on the basis of Intel SSE3). Anyway, idle-loop control acts on the assumption that in the wake of an action of a particular external process some *internal process* may have been made available, thus, switched over to ready-to-run state. Therefore, once returning from RELAX the pool of ready-to-run candidates is scanned (by ELECT) on the order of QUEST (Fig. 13(a)).

For each processor a flag is used to indicate whether or not shutdown may become effective. That flag is cleared by instruction of QUEST (line 13(a).4) and checked by RELAX (line 14(c).3). Using LABEL (Fig. 14(b)), the flag is typically set by the *first-level interrupt handler* (FLIH) of the operating system—or a suitable signal handler in case of a guest-level system. If that flag has not been set since nil return of ELECT and recognition of this fact within QUEST (line 13(a).6), then it is assumed that no interrupt handler was executing and, as a consequence of this, no process was made available. In that case, the processor can be made idling (lines 14(c).4–5). Otherwise, processor-level idle-loop control (RELAX) immediately returns to process-level idle-loop control (QUEST) to facilitate selection of a possibly ready-to-run process. Obviously, the whole sequence of actions from checking the flag until switching into standby mode (lines 14(c).3–5) discloses a potential race condition. Therefore, precaution is taken that the cause of this race condition cannot occur: the processor runs with all interrupts disabled during that critical section.

Although generally a poor solution due to the risk of missing hardware signals, there is no way around disabling interrupts in that particular situation—unless the real processor would provide such simple low-level features for idle control (Fig. 14) or one has a good mind to do "hacking" of the program counter saved by the processor with interruption. Alternatives are also imaginable: for instance, attend making a ready-to-run process available (e.g., READY) always by calling LABEL. Such lifting to process level is reasonable in cases where processes can be made available by actions of other processors in a parallel system, for example when calling PROVE (Fig. 7(a)) from within a direct-result guarded region controlled by a sequencer that owns a private processor core for order processing. But even this will not supersede interrupt-controlled termination of the idling cycle of a processor, as an additional *interprocessor interrupt* (IPI) may be necessary to indicate process availability if the respective processor already has stopped (line 14(c).5).

G. Process Switching

Investigation of a run-time support system for guarded sections would have been incomplete without paying regard to *processor dispatching*, especially against the background of shallows due to race conditions that may occur at different levels of abstractions. The process switch is such another neuralgic spot. This central function needs to update (at least) two data structures in the broader sense, namely (1) the pointer to the current process instance and (2) the current processor status. Given state of the art hardware, this update is not an elementary operation and, thus, divisible in time. In unfavourable conditions and without suitable precautionary measure, the processor status of the *physically current* process could be associated with the wrong process instance.

The trick to overcome this potential race condition is to derive the process pointer of the current (i.e., running) process from the *stack pointer* (SP). Assuming that (1) the per-process stack is allocated to a memory partition of fixed size s of a power of two, (2) the numerical value of the memory address of this partition is an integer multiple of s and (3) the per-process PCB is located at start of that partition. Then, process pointer p of the current process can be computed by a simple arithmetic-logical operation (i.e., two's complement AND) from the current value of the SP as follows:

$$p \;=\; \texttt{<SP>} \;\&\; -s. \tag{1}$$

If the operating system in charge of process management is based on a so called *process-based kernel* where several kernel-level threads exist and each of those threads owns a private stack, then a process switch technically means a *kernel-stack switch* by simply swapping the contents of the SP register of the underlying processor. By mapping the PCB in this manner, the race condition of a process switch can be prevented if the programming model of the processor makes reading and writing of that

```
1: function BEING(name)
2:     if name = ONESELF then
3:         return <SP> & PCBMASK
4:     end if
5:     task ← N2P[name mod NPROC]
6:     if task.name = name then
7:         return task
8:     end if
9:     return 0
10: end function
```

(a) Process-instance pointer.

```
1: function SEIZE(task)
2:     this ← BEING(ONESELF)
3:     if task ≠ this then
4:         this.state ← ∀(this.state \ {RUNNING})
5:         this.status ← SHIFT(task)
6:         task.scope ← CPU
7:     end if
8:     GAUGE(task)
9:     return this                          ▷ last process
10: end function
```

(b) Process-instance switch.

```
1: procedure GAUGE(task)
2:     task.state ← {RUNNING}
3: end procedure
```

(c) Process-instance calibration.

Fig. 15. Processor dispatching.

register an elementary operation—which is the case for contemporary hardware. Fig. 15(a) shows this solution (line 15(a).3), with $PCBMASK$ being equal to $-s$ according to formula (1). Selection of a PCB "by name" (line 15(a).5) is non-critical and uses a mapping table of process pointers ($N2P$, "name to process instance").

Alternative is to have the PCB located at the end of the respective (per-process) memory partition, that is to say, as initial local variable on top of a "virgin" push-down stack. In this case, computation of the process pointer involves the following steps subsequent to (1):

$$p' \quad = \quad p + s - sizeof(PCB). \tag{2}$$

While former solution (1) is sufficient under the assumption that a *worst-case stack usage* (WCSU) will never be exceeded by a process, latter solution (2) acts on the assumption that potential stack overflow can be detected by the underlying processor (e.g., by using a memory protection or management, resp., unit). In other words, solution (1) depends on *a priori knowledge* and static program analysis in order to determine the largest (power

```
1: function SHIFT(cor)
2:     memory[<SP>--] ← processor status
3:     aux ← RESUME(cor, TRUE)
4:     processor status ← memory[<SP>++]
5:     return aux
6: end function
```

(a) Stack-based context switch.

```
1: function RESUME(cor, process-based)
2:     if process-based then                ▷ switch stack pointer
3:         aux ← <SP>
4:         <SP> ← cor
5:     else                                 ▷ switch return address
6:         aux ← memory[<SP>]
7:         memory[<SP>] ← cor
8:     end if
9:     return aux
10: end function
```

(b) Stack-based coroutine switch.

Fig. 16. Problem-oriented switching of flows of control.

of two) s that could be generated by any process in the system. In contrast, solution (2) complies with an average value of s for all processes and considers stack overflow as exceptional but tolerable infrequent case.

Last but not least, the lowest level of process switching regards the processor-dependent actions needed to suspend and resume the particular control flow that makes up a process. The basic instrument used for the given case is the *coroutine*. In the following two different views are taken. At lowermost level, namely in terms of the *program counter* (PC), the coroutine is considered on a par with a normal routine: (1) only the PC is saved and restored in the course of a call and (2) the stack is used for safeguarding. Further registers constituting the processor status will be saved/restored by each routine or coroutine, respectively, on its own and as needed. As with routines, this function of *context backup* is considered a minimal extension to PC backup and handled separately depending on the particular requirements of a coroutine. The latter makes up the higher-level view of being able to manage coroutine-specific contexts of different sizes in a problem-oriented manner, while having the same lower-level view of switching between coroutines in common.

In operating systems, stack-dependent safeguarding of a coroutine status is typical for process-based kernel. The alternative would be an *event-based kernel* that only supports a single kernel-level thread and, thus, stack. In that case, several coroutines would have to share the

same kernel stack and the PC—instead of the SP as with a process-based kernel—needs to be safeguarded directly by some PCB-linked coroutine descriptor. The different models are suggested in Fig. 16(b).

Assuming a stack-based processor as well as run-time model for the programming language used, execution of a suspended coroutine is resumed as shown in Fig. 16(b). A call of the RESUME function implicitly saves the contents of the PC on top of the current stack, which in turn belongs to the coroutine that intends to relinquish processor control. The SP value identifies the memory address where the PC contents was stored. In further course of a process-based model (lines 16(b).3–4), the function body then simply switches stacks and returns the SP contents of the coroutine having made the call. Otherwise, the event-based model is assumed and the return address of RESUME gets switched (lines 16(b).6–7).[10] Any way, the return value becomes actual parameter of a future call of RESUME in order to continue the coroutine suspended in this minute. Note that only in the process-based model happens function return und, thus, restore of the PC contents from the stack that has been put on just now.

In contrast to this lowest level, Fig. 16(a) describes a coroutine changeover accompanied by a context switch. Before calling RESUME, the stopping coroutine saves its processor status onto its stack (line 16(a).2). After return from RESUME, the starting coroutine restores its processor status from its stack (line 16(a).4). Thus, every coroutine is in charge of saving/restoring its own context. In other words, a coroutine is unaware of structure and size of the context of another coroutine but, nonetheless, is capable of resuming execution of the latter.

Reflection on the calling hierarchy starting with SEIZE as the root shows an important *optimisation opportunity* when it comes to a concrete implementation of the discussed algorithms. This approach considers streamlining options for coroutine switching[10] in a broader scope and translates these particularly as to the relevant calling sequence: SEIZE → SHIFT → RESUME. Note that each process switch eventually happens through a call of SEIZE. Also note that the POSE variant discussed here follows the model of a process-based operating-system kernel. Bearing this in mind, realisation of SHIFT and RESUME each as *inline function* on the one hand removes unnecessary overhead as to subroutine management (i.e., parameter passing, call, activation-record allocation/deallocation, and return). On the other hand, this measure implies the upward propagation of stack switching from the level of RESUME (lines 16(b).3–4) to the level of SEIZE. If the actual compiler used is incapable of automatic optimisation in this regard, it is worth giving thought to semi-automatic or even manual measures—subject to the condition that the system-programming language used provides adequate linguistic support (such as C).

[10]Particularly for the event-based model, an even more streamlined solution becomes possible if one relies on or enforces *function inlining*. In that case, the following two actions are merely needed to perform the coroutine switch: (1) save the address of the next but one instruction—that exactly follows action two—into placeholder *aux* and (2) jump to the address specified by argument *cor*.

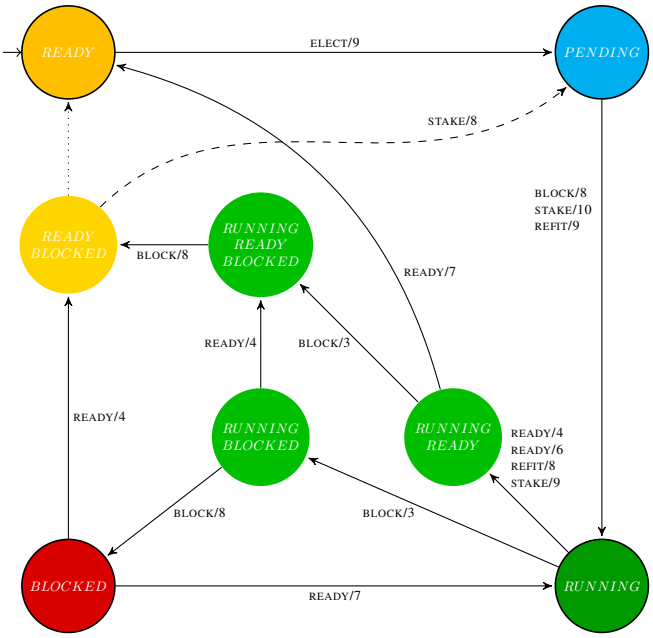


Fig. 17. Non-redundant process states and state transitions. The dotted arrow indicates an instantaneous state change, while the dashed arrow stands for special measures to control priority inversion.

H. State Transitions

Both as summary but also complement, Fig. 17 gives an overview of the (composite) process states that where addressed in the excursus so far. In addition, the state diagram shows the transitions that a process either performs oneself or effectuates for other processes. The corresponding edges are labeled with (1) the name of the operation in the context of which the state transition takes place and (2) the number of the belonging program line that contains the respective state-transition action. By way of example refers ELECT/9 to the CAS operation by means of which a process is removed from the pool of ready-to-run candidates (cf. Fig. 13(b), line 9).

The nodes are labelled with substate names, whereby

each of those names stands for a member of the set union that makes up a process state. In multi-member cases, the substates of a particular node are listed in decreasing dominance concerning process selection in the course of processor allocation. The four nodes forming the vertices of the state diagram denote *principal process states*:

1) $READY$, made available,
2) $PENDING$, selected for resumption,
3) $RUNNING$, proceeding on a processor, and
4) $BLOCKED$, awaiting an event.

The residual nodes represent *subsidiary process states* that help making a non-blocking and, above all, wait-free synchronised system become a reality.

Initial state of a *schedulable process* is $READY$, indicating that all (consumable or reusable, resp.) resources needed by the process to start, continue, or finish (i.e., to proceed) are available except the processor. If selected, the process is $PENDING$ with some processor and will soon proceed, that is, switched on. A pending process will not be selected again before its next cycle.

A process is absolutely suitable for processor allocation if $READY$ is either single or topmost item on the per-node list of substates. In contrast, a process is only suitable to a limited extent if such a composite state comprises subset $\{RUNNING, READY\}$, here with leftmost meaning topmost. Substate $RUNNING$ dominates substate $READY$ in that case—which appears to be logical: resumption of a running process is effectless in functional terms and, thus, implies nothing but overhead in non-functional terms. However, in order to resolve certain race conditions (particularly the lost-wakeup problem, cf. Sec. A-D) it might be necessary to select even such a process. Similar holds in case of the set union $\{RUNNING, READY, BLOCKED\}$ of a composite state, which denotes a blocking (i.e., still running) process whose (1) waiting condition was invalidated in the meantime but (2) processor release did not yet became effective. This state is typical for the idle process, that is, when no ready-to-run candidate is available at the time the current process passed into state $\{RUNNING, BLOCKED\}$ and, while persisting in that state, became ready to run again.

Substate $RUNNING$ is, in union with substate $READY$, of particular importance for the selection procedure. Basically, a ready-to-run process is eligible for resumption on any processor in a homogeneous or symmetrical, respectively, multi-processor/core system. Exception is when such a process also owns substate $RUNNING$. In this situation, processor/core allocation must be rejected as to the respective process except for the processor/core on which this very process currently proceeds. Otherwise that process becomes erroneously duplicated on several processors. Typical example is, again, the idle process. This very process controls the idle loop on a particular processor and, thus, is considered to be immovable for this period. Process selection cares about such a scenario and, therefore, takes the locality (i.e., "execution scope") of a process into account for its decision (cf. Fig. 13(b), line 8).

I. Progress Guarantee

As was shown, most POSE actions are free of loops and will perform in almost constant time. There are only two exceptions to this: (1) process selection by means of ELECT (Fig. 13(b)) and (2) process-level idle loop control by means of QUEST (Fig. 13(a)). Process selection is table-based. As the table has a static size, the table walk in order to select the next ready-to-run process is bounded in time and, thus, is constrained by a predictable WCET using static program analysis.

Process-level idle loop control is a different matter. At first sight, Fig. 13(a) shows an apparently unbounded loop: the search for a ready-to-run process terminates only if ELECT returned a valid process pointer. But such a behaviour is indispensable for any operating system in the absence of run capable processes. In this particular situation of the idle loop, only external processes will cause further progress of internal processes. At nary a point in the programs of Fig. 10–16 are to be found instructions that influence external processes and, thus, might have feedback on internal processes. This *feedback loop* is entirely under control of the application programs. Consequence of which is that, from the systems level point of view, the execution time of QUEST is bounded by the WCET of ELECT.

ACKNOWLEDGMENT

This work was supported by the German Research Foundation (DFG) under grants no. SCHR 603/8-1, /10-1, /13-1, and the Transregional Collaborative Research Centre Invasive Computing (SFB/TR 89, C1). We also thank Benjamin Oechslein as creative director of "on-the-fly contexts" (Sec. III-C) as well as Timo Hönig and Daniel Lohmann for helpful commentaries on the paper.

REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec. 2006.

[2] J. L. Manferdelli, N. K. Govindaraju, and C. Crall, "Challenges and opportunities in many-core computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 808–815, May 2008.

[3] N. Abramson, "The ALOHA system: Another alternative for computer communication," in *Proceedings of the Fall Joint Computer Conference (AFIPS '70)*. New York, NY, USA: ACM, 1970, pp. 281–285.

[4] A. Agarwal and M. Cherian, "Adaptive backoff synchronization techniques," in *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA '89)*. New York, NY, USA: ACM, 1989, pp. 396–406.

[5] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003), May 19–22, 2003, Providence, Rhode Island, USA*. IEEE Computer Society, 2003, pp. 522–529.

[6] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, pp. 124–149, Jan. 1991.

[7] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, Apr. 2008.

[8] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson, "Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?" in *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08)*. IEEE Computer Society, 2008, pp. 342–353.

[9] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the 24th ACM Symposium on Operating System Principles (SOSP 2013)*, M. Kaminsky and M. Dahlin, Eds. New York, NY, USA: ACM, 2013, pp. 33–48.

[10] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6–16, Jan. 1990.

[11] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computing Systems*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[12] S. Ramamurthy, "A lock-free approach to object sharing in real-time systems," Ph.D. dissertation, University of North Carolina, Department of Computer Science, Chapel Hill, 1997.

[13] B. J. H. Liskov, "Primitives for distributed computing," in *Proceedings of the Seventh Symposium on Operating System Principles (SOSP 1979), December 10–12, 1979, Pacific Grove, California, USA*. ACM, 1979, pp. 33–42.

[14] H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," in *Proceedings of the 1977 ACM Symposium on Artificial Intelligence and Programming Languages*, J. Low, Ed. New York, NY, USA: ACM, 1977, pp. 55–59.

[15] B. W. Lampson and D. D. Redell, "Experiences with processes and monitors in mesa," *Communications of the ACM*, vol. 23, no. 2, pp. 105–117, Feb. 1980.

[16] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[17] T. P. Baker, "A stack-based resource allocation policy for realtime processes," in *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS '90)*. IEEE Computer Society, 1990, pp. 191–200.

[18] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proceedings of the 16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. ACM, 2011, pp. 223–234.

[19] D. P. Friedman and D. S. Wise, "The impact of applicative programming on multiprocessing," in *Proceedings of the International Conference on Parallel Processing (ICPP 1976)*. Piscataway, NJ, USA: IEEE Computer Society, 1976, pp. 263–272.

[20] E. W. Dijkstra, "Cooperating sequential processes," Technische Universiteit Eindhoven, Eindhoven, The Netherlands, Tech. Rep. EWD-123, 1965, (Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996).

[21] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean, "Using continuations to implement thread management and communication in operating systems," in *Proceedings of the Thirteenth ACM Symposium on Operating System Principles (SOSP '91)*. ACM, 1991, pp. 122–136.

[22] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004.

[23] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975.

[24] C. Pu and H. Massalin, "An overview of the Synthesis operating system," Department of Computer Science, Columbia University, New York, NY, USA, Tech. Rep. CUCS-470-89, 1989.

[25] W. Schröder-Preikschat, *The Logical Design of Parallel Operating Systems*. Upper Saddle River, NJ, USA: Prentice Hall International, 1994.

[26] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk, "On interrupt-transparent synchronization in an embedded object-oriented operating system," in *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, I. Lee, J. Kaiser, T. Kikuno, and B. Selic, Eds. Washington, DC, USA: IEEE Computer Society, 2000, pp. 270–277.

[27] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*, G. Blelloch and K. Agrawal, Eds. New York, NY, USA: ACM, 2010, pp. 355–364.

[28] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications," in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, G. Heiser and W. Hsieh, Eds. Berkeley, CA, USA: USENIX Association, 2012, pp. 65–76.

[29] A. N. Habermann, L. Flon, and L. W. Cooprider, "Modularization and hierarchy in a family of operating systems," *Communication of the ACM*, vol. 19, no. 5, pp. 266–272, May 1976.

[30] D. L. Parnas, "Some hypothesis about the "uses" hierarchy for operating systems," TH Darmstadt, Fachbereich Informatik, Tech. Rep. BSI 76/1, Mar. 1976.