2015 IEEE 18th International Symposium on Real-Time Distributed Computing

Guarded Sections: Structuring Aid for Wait-Free Synchronisation

Gabor Drescher, Wolfgang Schröder-Preikschat
Department of Computer Science
School of Engineering
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Email: {drescher,wosch}@cs.fau.de

***Abstract*—This paper is about a novel approach of organising non-sequential programs to the benefit of wait-free synchronisation. Other than critical sections, processes never block at entrance to a *guarded section* although 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. In consequence, the execution model constrains the overlapping pattern of interacting (simultaneous) processes. Thereby, in the downstream transactional stage, efficient wait-free synchronisation of the "guarding operations" is gratifying by-product. First experimental results made with a 80-way multi-core system show that non-blocking wait-free synchronised guarded sections outperform MCS-locks.**

I. Introduction

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*. As a matter of principle, non-blocking synchronisation is free of this problem and, thus, makes counteractive measures [1] 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.

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), 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* [2] is common to disperse repeated execution of an atomic instruction. All these methods result in a serialisation of processes according to arrival time at the "hot spot." Similar holds for queueing locks [3]. They all interfere with real-time scheduling disciplines.

Only wait-free synchronisation [4] enables processing free of interference—but this synchronisation method is no walk in the park. In a number of cases, wait-free synchronisation largely benefits from *helping schemes* [5]. 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.

II. Design

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

For lack of space, in the following only non-blocking guarded sections are discussed. Direct-result and explicit-blocking guarded sections are presented in a supplemental publication [6], in addition to a discussion about an implementation sketch of a dedicated and deterministic run-time support system of time-predictable characteristic.

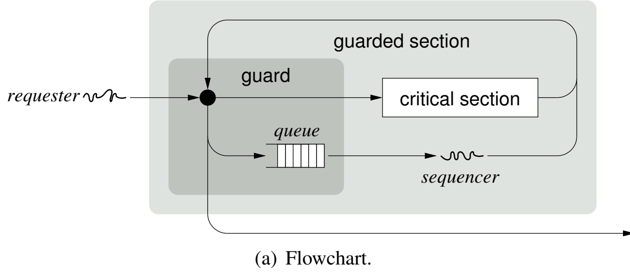
A. Non-Blocking Guarded Sections

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, but they are subject of preemption by high-priority processes. Thus, according to instructions, run-to-completion processes will never block inside a guarded section. Fig. 1 shows this basic model.

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

1555-0885/15 $31.00 © 2015 IEEE
DOI 10.1109/ISORC.2015.17

IEEE computer society



(a) Flowchart.

```
1: if (task ← VOUCH(guard, order)) ≠ 0 then
2:     repeat                                  ▷ sequential part
3:         handle(task)
4:     until (task ← CLEAR(guard, TRUE)) = 0
5: end if
```

(b) Sample: adopting role as a sequencer.

Fig. 1. Non-blocking guarded section.

```
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) Entry protocol: reception inspection and filling of the guard queue.

```
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) Exit protocol: final inspection and emptying of the guard queue.

Fig. 2. Sequence control of guarded sections.

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. But not on the input side, where 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. III) when compared to more general answers [5].

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. This corresponds to *lock-free synchronisation* of a guarded critical section although underneath of it the "guarding operations" are carried out in wait-free manner: thus, a sequencer is prone to starvation. But, on the other hand, 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.

Depending on the guard configuration, a high-priority process (1) occupying a guarded section and (2) having preempted a low-priority process in the course of clearing a yet filled guard queue (cf. Sec. III-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 by a high-priority process. In not a few cases, the potential delays for low-priority processes can be computed and bounded above based on preliminary knowledge as to the process organisation and priority mapping of the given real-time application.

III. 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 (e.g., in terms of threads) and future objects, its implementation is shown in Fig. 2.

A. Sequence Control

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

CLEAR implements the exit protocol. In line 2(b).2 the guard is released, followed by checks for becoming sequencer and an empty guard queue (lines 2(b).4–5). If the queue is filled, the current process (now no longer occupant) attempts to reoccupy the section by means of FAS. In case of success, this very process becomes sequencer: the next *task* is retrieved from the queue and the outer guard-loop continues (cf. line 1(b).4). Otherweise, some other process 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 potential of the *lost-update problem*, the prevention of which

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

```
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. 3. Multiple-enqueue/single-dequeue wait-free synchronised queue.

needs special care. A lost-update may occur when an enqueued item is ignored and no process executes the outer guard-loop. The implementation shown in Fig. 2 effectively prevents this problem. All processes 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. Either an enqueuing process 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. 3 shows the corresponding queue operations. Enqueuing (lines 3(b).2–4) follows the same pattern as the MCS queue-based lock [3] and uses FAS. To prevent spinning in the dequeue operation, as with the MCS algorithm, a novel 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 3(b).8–15 take special care for a possibly dequeued dummy element, which is simply enqueued again if need be (line 3(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 3(b).4, while lines 3(c).3 and 3(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 [7] or garbage collection. As can be seen, the discussed entry and exit protocols (incl. the queue operations) are completely wait-free as all actions are bounded in time.

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 process 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. 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 process identification and, thus, reflects the process priority The passage request with highest priority will be executed first.

IV. 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. Due to the lack of blocking critical sections, blocking guarded sections were not evaluated. 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 [3] 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.

TABLE I. BASIC OVERHEAD, UNCONTENDED CASE.

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

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 processes. Fig. 4 shows the results in number of cycles. In general, performance decreases dramatically for high contention in all cases. Especially when crossing socket boundaries, where an abrupt rise of the number of cycles can be stated 10 cores off. This boost is known from earlier experiments 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.

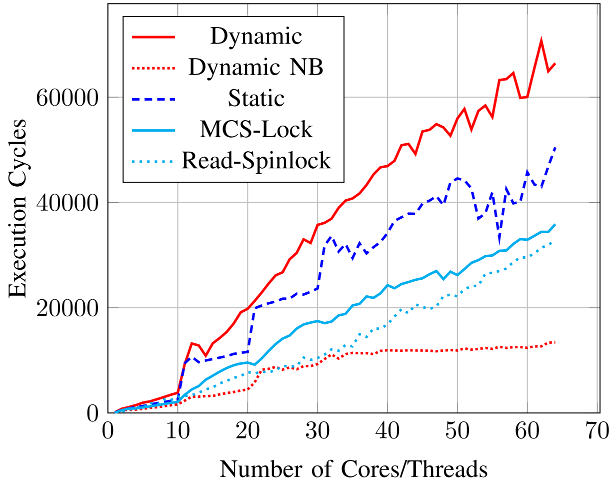


Fig. 4. High contended case, range 1:64 cores.

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 process has to wait on results and processes 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 processs 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.

V. CONCLUSION

The uniqueness of guarded sections is that 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. 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. 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 [6], 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.

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 and Timo Hönig for helpful commentaries on the paper.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] G. Drescher and W. Schröder-Preikschat, "An experiment in wait-free synchronisation of priority-controlled simultaneous processes: Guarded sections," Department Informatik, Friedrich-Alexander-Universität Erlangen-Nürnberg, Tech. Rep. CS-2015-01, Jan. 2015.

[7] 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.