Efficient Object Sharing in Quantum-Based Real-Time Systems

Summary by Andreas Haeberlen, University of Karlsruhe (haeberlen@ira.uka.de)

James H. Anderson, Rohit Jain, Kevin Jeffay
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175
{anderson,jain,jeffay}@cs.unc.edu

Abstract

The authors present a new scheme for implementing shared objects on a real-time system [1]. They assume that the system allocates processor time in discrete quanta, and that the quantum is large compared to the length of an object call. Under these conditions, most object calls are likely to execute without preemption, thus allowing the usage of simpler and more efficient access mechanisms.

This seminar talk was given on December 21st, 2000, during a real-time systems course at Karlsruhe University. The corresponding slides are available on the web from http://www.haeberlen.de/.

1 Introduction

During analysis of real-time systems, different tasks are often assumed not to interfere with each other. However, it is sometimes necessary for multiple tasks to cooperate on a single job, and this can add considerably to the difficulties a systems designer has to face.

For example, consider a mobile robot controlled by some kind of real-time system. Assume it has three concurrent activities: One uses a laser scanner to determine its current position; another one controls the robot's wheels, and a third one gathers data from the sensors. All of these activities need access to a common environment map, which might reside in some kind of shared memory.

1.1 Why sharing is difficult

In a non-preemptive system, implementing shared objects is easy. Tasks are guaranteed to stay active until they voluntarily relinquish control; therefore, they can be sure that their object calls execute without conflicts. However, this leads to unacceptable latencies for urgent jobs, because there is no way that they can interrupt the task which is currently executing. As a consequence, most real-time systems offer preemption.

If preemption is possible, however, a task might be preempted *while* accessing an object, and other tasks might then invoke conflicting object calls before it has a chance to finish. To avoid inconsistencies, these conflicts must be avoided or at least resolved.

1.2 Some common solutions

A first approach would be to establish mutual exclusion. Tasks would be required to grab a lock before accessing the object and to release it afterwards. However, there are some pitfalls to this solution:

- Possibility of priority inversion. This is a severe problem, especially in time-critical systems. One famous example is the Sojourner rover used in the Pathfinder mission to Mars [6], which occasionally had to be rebooted due to priority inversion.
- Unbounded wait time. Threads might have to wait a long time before acquiring the lock; in some cases, they can even starve completely.
- Deadlocks and other mutex-related problems. In a complex system, these can be really hard to find, and programmers must be very careful to avoid them.

Another lock-related problem is that the concept itself is too *pessimistic*: Applications must pay the cost of acquiring and releasing the lock even when there is little or no contention at all. However, if object calls are short, most of them are likely to execute without conflicts, and a good object sharing scheme should be optimized for this expected case.

There is a different approach which does exactly that. It is called lock-free, because it does not rely on locks to prevent conflicts, but rather detects and *resolves* them when they actually occur. Object accesses must be implemented in a way that they either have the desired effect or no effect at all, and they must be able to detect failures. An object call then uses a loop which retries the access until it succeeds.

This scheme is better than the first one, because lower priority tasks can no longer impend the progress of higher priority tasks, and it has less overhead in the expected case. However, it is still not acceptable for real-time systems because the operation might be involve multiple retries, and thus the execution time cannot be bounded. The reason is that preemptions might occur at any time, even multiple times during a single object call.

1.3 Quantum-based Systems

Fortunately, most real-time systems are not fully preemptive. The reason is that in many systems, dispatching is not done at instruction granularity. Instead, when activated by the dispatcher, every task is given a discrete amount of processing time, which is called a *quantum*. Of course, important tasks might receive more quanta than others, but once a task is assigned to the processor, it is guaranteed to run at least for the length of an entire quantum.

In fact, some real-time scheduling algorithms, such as round robin and the proportional-share [4] scheme, are expressly quantum-based. The other algorithms are often implemented on a kernel that makes scheduling decisions at regular intervals - e.g. because a timer is used to invoke the scheduler - and thus implicitly uses quanta.

Quantum-based systems can be seen as a compromise between non-preemptive and fully preemptive systems. They have the lower scheduling overhead of the former and the lower latencies of the latter. This might explain why they are so popular.

1.4 Approach of the paper

In reality it can be observed that, compared to the length of an object call, the typical quantum is quite large. The authors cite results from one of their earlier papers [3], where they measured various object calls on an old 68030 machine running at 25 MHz. Even then, most calls completed within 100μs in the worst case, whereas a typical quantum is about ten times larger. In fact, if the quantum is smaller, performance can drop significantly because of cache effects. The paper thus makes the following assumption:

> **Preemption axiom:** *The quantum is large enough to ensure that each task can be preempted at most once across two consecutive object calls.*

If this axiom holds, object sharing is facilitated a great deal, because the lock-free scheme can then be implemented with bounded retries, given that an object call can be preempted *at most once*; the second attempt then gets a fresh quantum and can execute without preemption. Furthermore, the sharing code can be simplified because the loop is no longer necessary; if the first attempt fails, the second attempt is guaranteed to complete and does not need any checks.

The remaining part of the talk is structured as follows: In section 2, a few simple and efficient algorithms are presented to illustrate the power of the Preemption Axiom. The following section shows how this principle can be used to compute better bounds on worst-case execution time. Section 4 presents an extension for multiprocessors, and section 5 concludes.

2 Algorithms

To demonstrate the power of the Preemption Axiom, this section presents some basic object access procedures and their implementations. The benefit of the Preemption Axiom in this context is simplicity and efficiency; for schedulability analysis, please refer to section 3.

2.1 Prerequisites

To implement the notion of an atomic object access, at least *some* support from the hardware is required. In this section, we will assume that the hardware has an atomic compare-and-swap instruction with the following semantics:

```
CAS(addr, old, new)
<
  if (*addr==old)
     then *addr = new; return true;
     else return false;
>
```

The angular brackets are used to indicate that this operation is atomic, i.e. it executes completely, or not at all.

2.2 Read-Modify-Write (RMW)

The CAS primitive described above can only be used if the new value to be installed is known in advance and does not depend on the previous value.

If that is not the case, another primitive like RMW is required:

```
RMW(addr, func)
<
  old = *addr;
  *addr = func(old);
  return old;
>
```

Again, the operation must execute atomically. However, there usually is no hardware instruction that supports RMW directly, so it must be implemented by means of CAS and the Preemption Axiom:

```
RMW(addr, func)
{
  var old, new;
  old = *addr;
  new = func(old);
  if(CAS(addr, old, new) == false)
    {
     old = *addr;
     *addr = func(old);
    }
  return old;
}
```

The algorithm first reads the old value from memory and computes the user-defined function; then it tries to commit the result by performing CAS. If this succeeds, the value cannot have been changed in the meantime, and RMW is complete. However, if the CAS operation fails, there must have been a preemption within the first two lines, and another task must have changed the value. Fortunately, this also means that a new quantum has just begun, and by virtue of the Preemption Axiom, the second attempt is guaranteed to execute without preemption. We can therefore omit all the checks and retry using purely sequential code.

The benefit from the Preemption Axiom is that the WCET of this code can easily be bounded (it executes in ’constant’ time), and that the algorithm is so simple.

2.3 Conditional Compare-and-Swap (CCAS)

The CCAS operation is another useful primitive for implementing shared objects. It is different to a simple CAS in that it can check *two* values instead of only one, i.e. it fails if either one does no longer hold its presumed content:

```
CCAS(addr, old, new, version, old_version)
<
  if (*version == old_version)
       then return CAS(addr, old, new);
       else return false;
>
```

CCAS is difficult to implement because for it to succeed, the two conditions must be true *at the same time*. To ensure this, the algorithm assumes that it can tag one of the values while it checks the other one, because it needs to detect any modifications during that time. For this reason, a small part of the memory location holding the value needs to be dedicated to the tag.

In the expected case, the algorithm performs the following steps (see figure 1):

1. It reads the memory location holding value and tag

2. Then it checks the value part against the supplied old value. The version number is also checked.

3. Now the tag field is set to some unique value, e.g. the process ID. The update is committed using compare-and-swap. From this point, CCAS can detect modifications on the value field.

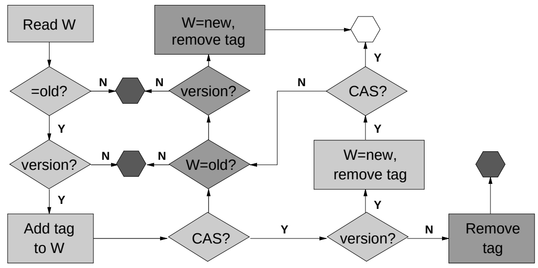

Figure 1. CCAS algorithm

4. Once more, the version number is checked. If this succeeds, CCAS can be sure that both values are correct.

5. The tag is removed, and the new value is installed. Again, the update is performed atomically via compare-and-swap.

Of course, the algorithm must take into account that it might be preempted at any time, and it must be able to recover. However, if a preemption occurs within the first two steps, no special action is necessary because nothing has been changed yet. These checks are performed only to ensure that the operation has a *chance* of succeeding; if one of the values has already been modified, it is best to fail immediately.

If the compare-and-swap in step 3 fails, however, this means that the algorithm was preempted during the second step by another process, and that this process has modified the memory location. However, a fresh quantum is available now, and CCAS can perform a simplified retry:

1. It checks both the value part of the memory location and the version number. If either of them was modified, CCAS fails immediately.

2. It clears the tag, installs the new value and commits the changes by simply writing them back to memory.

A failure during step 5 can be handled in exactly the same way, because the only possible reason is a preemption during step 4, and this also results in a new quantum being issued.

If the version check in step 4 fails, a preemption occurred during the third step. As the version number does no longer match, the operation must fail; however, the tag has already been set at that time and must be removed.

Please note that, although the worst-case execution time of CCAS itself can easily be bounded, the operation as a whole can fail and may have to be retried.

3 Schedulability Analysis

In order to provide hard real-time guarantees to applications, the admission controller needs bounds on the execution times. Better bounds enable it to admit more tasks, which increases processor utilization. In this section, the authors show how the Preemption Axiom can be used to improve these bounds, and what impact this has on schedulability analysis.

3.1 Prerequisites

For simplicity, the authors consider only the rate-monotonic scheme; extending their approach to the EDF scheme is straightforward. A periodic task model is assumed, where each job consists of distinct non-overlapping phases with at most one object call each. Phases containing no object calls are called computational, otherwise they are object access phases.

Furthermore, it is assumed that all tasks can be indexed in a way that each task may be preempted only by tasks with a lower index. One might think of that index as a priority. It can be constructed under both the RM and the EDF scheme.

3.2 Straightforward Solutions

When computing the worst-case execution time of object access code, the main difficulty is to find bounds on the interference cost, i.e. the additional overhead caused by other applications trying to access the same object. Fortunately, because of the Preemption Axiom, this overhead is usually limited to one retry per object call. However, it is still unknown how many retries will be necessary when executing multiple object calls.

A first approach would be to account for every possible retry. This could be done by finding the most expensive retry and adding the respective cost for every quantum boundary crossed, or for every object access phase, whichever is lower. Of course, this might result in another quantum boundary being crossed, so the corresponding formula is a recurrence relation.

Of course, this bound is too high. For example, retry costs might vary across different types of object call, so it might pay to apply them in decreasing order instead of always taking the maximum. But the resulting bounds are still obviously too high.

3.3 Obtaining better bounds

To further improve these bounds, the paper presents a different approach. Let w_j denote the number of phases in task T_j, and assume the additional overhead s_j^v of each phase v is known.

The *exact* worst-case interference cost $E_i(t)$ for a task T_i in an interval of length t can then be computed:

$$E_i(t) \equiv \sum_{j=1}^{i} \sum_{v=1}^{w_j} \sum_{l=1}^{j-1} m_j^v(l,t) \cdot s_j^v(l)$$

where $m_j^v(l,t)$ is the worst-case number of interferences caused in T_j's v^{th} phase by jobs of T_l.

Of course, the values for m_j^v are not known in advance. However, there obviously are some constrains for the m_j^v. For example, remember that a job can preempt jobs of other tasks only if these have a higher index, so every high-priority job can cause *at most one* retry for a low-priority job, i.e. the one that was executing when the new job was released.

The idea is to maximize the m_j^v under these and similar constraints. While this approach is certainly against intuition, it yields the maximum value for each m_j^v that can possibly occur, i.e. it bounds them. To compute the maximum values, the admission controller can use linear programming.

The paper presents six constraint sets, three of which were already introduced in a previous paper [2]. As the goal is understanding and not mathematical accuracy, the exact definitions are not included here; if you need the details, please refer to the original document. The constraint sets describe the following properties of the system:

- Every job of a task with higher priority than T_i can cause at most one retry for T_i.
- The number of retries in $T_1 \ldots T_i$ is smaller or equal than the number of jobs in $T_1 \ldots T_{i-1}$.
- The number of retries in every single object access can be bounded.
- Object accesses are never retried twice (Preemption Axiom).
- Interferences can occur at quantum boundaries only (Preemption Axiom).
- Then number of retries cannot exceed the number of object access phases.

The original paper contains scheduling conditions derived from these constraints and also briefly considers the proportional-share scheme. Details will not be presented here.

3.4 Experimental comparison

To evaluate their suggestion, the authors conducted a series of simulation experiments. They generated 120 task sets, each with 10 tasks accessing up to 10 shared objects. Then they computed the estimated retry cost both with their new algorithm and one from a previous paper [2]. The results show that the new algorithm improves the bounds by about 10% to 20%. It also takes considerably less time to complete.

4 Extension to Multiprocessors

The solutions presented in the previous sections assume that the only cause of interference is preemption. This is certainly not true for a multiprocessor system, because any two processors might be trying to access a certain object *at the same time*. This section describes an extension to the Preemption Axiom for use with multiprocessors.

4.1 New Issues

Because a multiprocessor system has true concurrency, a simple retry mechanism is not sufficient. Object accesses might repeatedly be interfered with, possibly causing an unboundable number of retries. For this reason, accesses must be synchronized, and some kind of lock is required.

However, because of the problems mentioned earlier, that lock should be implemented with care. In a real-time system, the time a thread spends waiting for the lock needs to be bounded, so it is advisable to use a queue lock. It is also important that the waiting thread does not impend the progress of other threads, so the lock should be preemptable.

Various preemptable queue locks have been proposed, e.g. [5]. The benefit of the Preemption Axiom in this context is that the resulting lock is considerably simpler. For example, the lock proposed in [5] has a total of 63 executable statements, while the one discussed here needs only 17.

4.2 Prerequisites

For this approach to work, the Preemption Axiom must be extended to ensure that the quantum is long enough to hold $P + 1$ consecutive object calls, where P is the number of processors across which the object is shared. This is still not unreasonable, because it is very challenging to implement real-time systems on large-scale multiprocessors and even then, it is unlikely that a single object would be shared by a large number of processors.

The approach also requires some OS support; the system must be able to set an user-accessible flag on every preemption. Additionally, the algorithm can benefit from the ability to postpone preemption for just a small amount of time, but this is not essential.

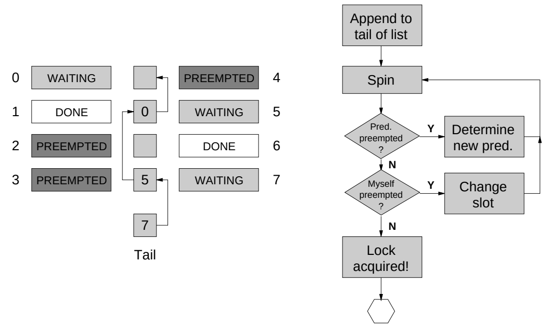

Figure 2. Preemptable queue lock

4.3 Implementing Objects

Let N denote the number of tasks that can have access to the object. The queue lock needs three shared variables: an array of size N, which is used to store the predecessor of each waiting task, a pointer to the tail of the queue, and another array of $2N$ memory locations called "slots". Each task T_i owns two of these slots, i and $i + N$, and each of them can hold one of the values 'waiting', 'preempted' and 'done'.

If a task needs the lock, it chooses one of its slots and sets it to 'waiting' (see figure 2). It also tells the operating system to change that value to 'preempted' when it activates another task. Then it appends itself to the end of the queue by performing a fetch-and-store operation on the tail pointer and storing the old value to its own predecessor pointer.

Now it starts spinning on two slots, its own active slot and that of its predecessor. This spin will stop for one of the following reasons:

1. The predecessor is preempted. In this case, the waiting task determines the predecessor of that task and uses it as its new predecessor.

2. The waiting task itself is preempted. Then it switches to its other slot and re-appends itself to the tail of the list.

3. The predecessor releases the lock by setting its slot to 'done'. Now the task can safely access the object.

Please note that one spin location per task is not sufficient because if a preemption occurs, the direct successor must notice this and advance in the queue, which might not be possible if 'waiting' was re-established directly afterwards. On the other hand, no more than two spin locations are required because when a task is forced to retry, the extended version of the Preemption Axiom ensures that the remaining part of the algorithm executes without preemption.

Another issue is that a task might be preempted during the object access itself. The original version of the algorithm prevents this by executing the object access as a critical section, i.e. with interrupts disabled. However, the lock-free scheme from section 2 could be used instead.

4.4 Experimental Comparison

The authors conducted performance experiments to compare their scheme against another preemptable queue lock presented on the previous RTSS conference by H. Takada and K. Sakamura [5]. For their experiments, they used a parallel architecture Simulator called Proteus, which enabled them to customize the kernel interface and provide the extra functionality needed by the algorithms.

The results show that on average, the algorithm presented in the paper needs about 25% less time to acquire a lock than the one designed by Takada and Sakamura; the best-case time was even reduced by 50%.

5 Conclusion

The paper presents a new approach to implementing shared objects on quantum-based real-time systems. An optimistic, retry-based mechanism is combined with certain assumptions about when preemptions can occur. The resulting algorithms are efficient and surprisingly simple.

The authors discuss implications on schedulability analysis and demonstrate that their scheme can be used to achieve better bounds on worst-case execution time. They also extend their approach to multiprocessor systems.

References

[1] J. Anderson, R. Jahin, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 346–355, December 1998.

[2] J. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 92–105, 1996.

[3] S. Ramamurthy. A lock-free approach to object sharing in real-time systems. PhD thesis, University of North Carolina, Chapel Hill, North Carolina, 1997.

[4] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288–299, 1996.

[5] H. Takada and K. Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 134–143, 1997.

[6] Time Rover Corp. *Priority Inversion and the Mars Pathfinder*. http://www.time-rover.com/Priority.html.

About the Authors

James H. Anderson *is an associate professor at the University of North Carolina. His research topics include distributed and concurrent algorithms, formal methods, and real-time systems. He is also the supervisor of* ***Rohit Jain****, a co-author of this paper, and Srikanth Ramamurthy, who contributed to a number of other papers that are cited here.*
Kevin Jeffay*, S. Shephard Jones Professor at UNC, has been present at virtually every important real-time conference since 1996; he even held the program chair twice, at RTAS 1996 and at RTSS 2000. His current projects include multimedia networking, computer-supported collaborative work, and real-time systems. Together with Anderson, he is a member of the DiRT group at the University of North Carolina.*