Real-Time Synchronization in Real-Time Mach

Tatsuo Nakajima
Waseda University
446 PUBLICATIONS **4,344** CITATIONS
SEE PROFILE

Hideyuki Tokuda
National Institute of Information and Communications Technology
438 PUBLICATIONS **7,458** CITATIONS
SEE PROFILE

Real-Time Synchronization in Real-Time Mach

Tatsuo Nakajima[‡] Hideyuki Tokuda[†‡]

Abstract: The correctness of a real-time program should be ensured by both the results of computation and the time at which the results are produced. In the program, predictability and analyzability will become important properties for ensuring their correctness. A synchronization facility plays an important role for managing the contention and the cooperation among concurrent activities so that we should pay much attention to real-time synchronization in real-time computing. In this paper, we present the synchronization facilities which provide predictability and analyzability in Real-Time Mach.

1 Introduction

The correctness of a real-time program should be ensured by both the results of computation and the time at which the results are produced. In the program, *predictability* and *analyzability* will become important properties for ensuring their correctness. The real-time systems are predictable when we know the behavior of all system primitives such as the worst case execution time. Also ,the systems are analyzable when we know whether all requirements of applications such as timing constraints are satisfied or not at the design stage.

A synchronization facility plays an important role for controlling the contention and the cooperation among concurrent activities. In real-time systems, the blocking time due to synchronization should be bounded for ensuring predictability and analyzability.

In this paper, we focus on the real-time synchronization for constructing a critical section in a single CPU environment. Real-Time Mach[3] provides several synchronization facilities which has different characteristics for predictability and analyzability. The synchronization facilities enable each application to select a suitable synchronization facility for their purpose. In section 2, we describe a model of real-time synchronization. Section 3 presents the structure of the synchronization module in Real-time Mach. In section 4, we discuss future works on real-time synchronization.

2 Real-Time Synchronization

2.1 Priority Inversion Problem

The contention among programs should be controlled when several activities share protected resources and need to decide the order of their executions. The cooperation of programs should be controlled when the activities need to wait for incoming events, and several activities have a precedence relation. Synchronization may block other activities until the contention is resolved or some conditions are satisfied.

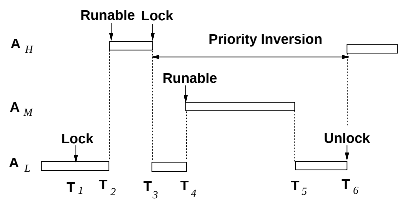

Figure 1: Critical Section with Priority Inversion

A real-time system designer need to determine the worst case blocking time of a higher priority activity for the shared resource. It is often impossible to compute the bound if an activity in the protected region can be preemptable. For realizing predictable and analyzable computing, blocking time should be bounded. Real-time systems control activities according to their deadlines. The unbounded blocking is caused by the preemption of lower priority activities which is acquiring a critical section. The problem is called *priority inversion problem*.

Let us consider the following case. Suppose that the lowest priority activity A_L is in the critical region first at T_1, then the highest priority Activity A_H becomes runnable at T_2 and attempts to enter a critical region at T_3. However, since A_L is in the critical region, A_H must wait for its completion. After A_L resumed, a medium priority thread A_M becomes runnable at T_4. Then A_M starts running without using the critical resource and may wake up another medium priority thread and so on. After A_M is completed at T_5 and then A_L exits the critical section at T_6, A_H can enter the critical section. In the example, priority inversion is occurred from T_3 to T_6. The time of priority inversion cannot be bounded without knowing all behavior of related medium priority A_M's. Figure 1 shows the time sequence in the execution.

In order to bound the worst case blocking time, a simple solution called *priority inheritance* scheme was developed in our group [1]. An priority inheritance scheme is that once A_H enters in a critical section, A_L inherits the priority of A_H. Then,

[1†] Faculty of Environmental Information, Keio University
[‡] School of Computer Science, Carnegie Mellon University

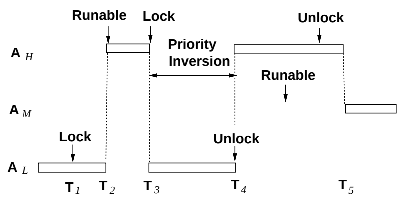

Figure 2: Critical Section with Priority Inheritance

A_M cannot preempt the activity of A_L in the critical region. In this way, the worst case blocking time of A_H can be bounded if only the worst case execution time of the critical section is known. Figure 2 shows the time sequence of the execution.

Traditional synchronization primitives use FIFO ordering among waiting activities to enter a critical section, since FIFO ordering can avoid the starvation. In real-time computing environment, however, FIFO ordering often creates a priority inversion problem. A higher priority activity must wait for the completion of all low priority activities in the waiting queue. If all of real-time activities can meet their deadlines, then there will be no starvation among these threads. Thus, in real-time synchronization, the system should provide a priority based ordering to avoid unbound priority inversion problems.

A client/server model in many operating systems is also a source of an unbounded priority inversion. A Server may have several threads for processing requests from clients at the same time. When all threads in the server are processing previous requests, a high priority request is blocked and is entered in the waiting queue of the server. One of the threads inherits the priority of the blocked request to avoid an unbounded priority inversion.

2.2 Real-Time Synchronization Model

Real-Time Mach provides real-time thread model[3] for the thread management. A thread is classified into a periodic thread and an aperiodic thread. A synchronization facility manages the critical section among the threads.

In Real-Time Mach, the following primitives are defined for constructing critical section.

kr = **rt_mutex_allocate**(&mutex, mutex_attr)
kr = **rt_mutex_deallocate**(mutex)
kr = **rt_mutex_lock**(mutex, timeout, context)
kr = **rt_mutex_unlock**(mutex)
kr = **rt_mutex_control**(mutex, cmd, arg, size)

```
typedef struct mutex_attr {
    rt_policy_t    mutex_policy;   /* mutex policy */
    rt_priority_t  mutex_priority; /* ceiling priority */
                   ...
} rt_mutex_attr_t;
```

A mutex variable is allocated for constructing a critical section. Two primitives *rt_mutex_allocate* and *rt_mutex_deallocate* allocates and deallocates a mutex variable. In *rt_mutex_allocate*, we can specify a mutex attribute. The attribute includes a policy field indicating a locking protocol. In the beginning of a critical section, *rt_mutex_lock* should be called and *rt_mutex_unlock* is called at the end of the critical section. *rt_mutex_control* allows to access an attribute of a mutex.

In the rest of this section, we classify locking protocols for a real-time synchronization. Preemptability of the running thread in the critical section affects the synchronization policies and the schedulability of the task sets. In Real-Time Mach, the following three preemption levels of the running task in the critical section has been supported.

Non Preemptable: No preemption is allowed while a thread is executing in the critical section.

Preemptable: A higher priority thread can preempt the current running thread. If the higher priority thread need to enter the critical section, it will be blocked (i.e., it must wait for the completion of the critical section).

Restartable: A higher priority thread can preempt the current running thread. It then aborts the running thread and puts the thread back to the waiting queue. The higher priority thread executes the critical section without further delay. The preempted lower priority thread will restart later from the beginning of the critical section.

By selecting a queueing ordering and the above preemption choices, the following synchronization policies can be defined:

Kernelized Monitor protocol (KM): KM protocol takes the non preemptable mode.

Basic Priority protocol (BP): BP protocol takes the preemptable mode and the queueing ordering is based on the thread priority.

Basic Priority Inheritance protocol (BPI): BPI protocol is created as BP protocol plus the priority inheritance function. By this function, a lower priority thread executing the critical section inherits the priority of higher priority thread, when the lock is conflicted.

Priority Ceiling protocol (PCP): PCP protocol is an extension of BPI protocol. In PCP, the it ceiling priority of the lock is defined by the priority of the highest priority thread that may lock the lock variable. The execution of the thread is blocked when the priority ceiling of the lock is not higher than all locks which are owned by other threads. The protocol prevents deadlock, and chained blocking. The underlying idea of PCP is to ensure that when a thread

T preempts the critical section of another thread *S* and executes its own critical section *CS*, the priority at which *CS* will be executed is guaranteed to be higher than the inherited priorities of all the preempted critical sections.

Restartable Critical Section protocol (RCS): RCS policy is based on the restartable mode. In RCS, a higher priority thread is able to abort the current running thread in the critical section and puts it back to the waiting queue with recovering the state of shared variable. During the recovery, the higher priority thread's priority is inherited like in priority inheritance protocol. After this recovery action, the higher priority thread can enter the critical section without any waiting in the queue. A user program must be responsible to recover the state of shared variable.

Each synchronization protocol has a different characteristics of schedulability and a different cost entering to and exiting from critical sections.

2.3 Real-Time Synchronization Analysis

To compute a schedulable bound, we must avoid a potential unbounded priority inversion problem among real-time threads. Once we could bound the worst case blocking time, then we can compute a bound as an extension of the *rate monotonic* scheduling analysis.

There are basically two approaches to bound the worst case blocking time among real-time tasks. One is using a *kernelized monitor* protocol and the other is using a *priority inheritance* scheme as we described in the previous section.

In the kernelized monitor protocol, while a task is executing in a critical section, the system will not allow any preemption of that task. Suppose that if *n* tasks are scheduled in the earliest deadline first order, the worst case schedulable bound is defined as follow.

$$\sum_{j=1}^{n} \frac{C_j+CS}{T_j} \leq 1$$

where C_i, T_i, CS represents the total computation time of $Thread_i$, the period of $Thread_i$, and the worst case execution time of the critical section respectively. In general, if the duration of CS is too big, the system cannot satisfy the schedulability test and end up with reducing the number of tasks and running with a very low total processor utilization.

On the other hand, if we relax the kernelized monitor and allow a preemption during the critical section, we may face the unbounded priority inversion problem. Under a priority inheritance scheme, once the higher priority task blocks on the critical section, the low priority task will inherit the high priority from the higher priority task. One of extended priority inheritance protocols is called a *priority ceiling protocol*[1].

Using these inheritance protocols under a rate monotonic policy, we can also check schedulable bound for *n* periodic threads as follows.

$$\forall i, 1 \leq i \leq n, \qquad \frac{B_i}{T_i} + \sum_{j=1}^{i} \frac{C_j}{T_j} \leq i(2^{\frac{1}{i}} - 1)$$

where C_i, T_i, B_i represents the total computation time, the period, and the worst case blocking time of $Thread_i$ respectively.

Restartable critical section protocol makes the blocking time to be smallest among the proposed protocols, if abort and recovery overhead is negligible. The schedulability formula for restartable critical section is similar to the formula for the priority ceiling protocol. The difference is that B_i should represent the abort and recovery time when using restartable critical section.

3 Structure of Synchronization Module

The current version of Real-Time Mach is being developed using a network of SUN3, DEC station and IBM PC/AT workstations and laptop machines. The microkernel provided us a better execution environment where we could reduce unexpected delays in the kernel. The preemptability of the kernel was also improved significantly since UNIX primitives and some device drivers are no longer in the kernel.

The synchronization facility is implemented based on the policy/mechanism separation concept for improving the adaptability of the system. Each synchronization policy module is implemented as an object. A mutex attribute indicates the policy object which controls its critical section.

The synchronization module is divided into three components: *synchronization management module*(SMM), *dispatching management module*(DMM) and *priority management module*(PMM). Figure 4 shows the relationship between the three components. SMM exports the interface to users such as *allocate*, *deallocate*, *lock* and *unlock* and calls DMM and PMM for controlling dispatching and priorities of threads. In DMM and PMM, policy and mechanism are clearly separated.

When a thread will enter into a critical section, DMM determines whether it is allowed or not. When the thread need to be blocked, DMM decides which thread in the critical section should be inherited the priority of the thread which will be enter to the critical section. In DMM, a policy part is separated as an object clearly and can change a policy according to the characteristics of applications. A different policy object encapsulates a different policy to determine which thread should be inherited the priority. In usual environment, we use the policy to select the thread which execute the shorted critical section or the thread which has the highest priority.

PMM decides how to inherit a priority to the thread which is executing a critical section, for instance BPI, PCP or RCS. PPM also separate a policy part clearly. These policies reside in policy objects.

The policy for DMM and PMM is selected at the allocation time for a mutex variable. Each mutex variable can choose a different policy from the characteristics of a critical section.

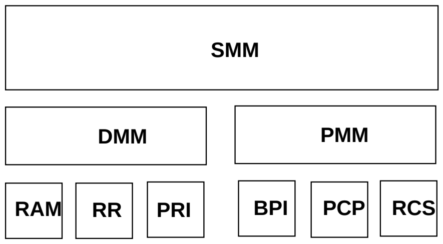

Figure 3: Synchronization Module

4 Discussion

In this section, we discuss the four issues about real-time synchronization.

How to select policy: Each synchronization protocol has a different cost and a different time of priority inversion. For example, the cost of KM is very cheap but it may cause longer priority inversion time and lower the break down CPU utilization of systems. In [4], we show the tradeoff between the cost and schedulability for each locking protocol. The support for multiple locking protocols enables us to select suitable one for their applications. Presently, the selection of protocols is responsible of users. The selection should be done in a compiler or a run-time system using a specification of an application.

Synchronization in Multiple Domains: We focus on only a critical section in this paper, but synchronization is appeared in different domains such as IPC. Priority control mechanism should be integrated in all domains. In our implementation, DMM and PMM may be shared by the synchronization module and the IPC module. In distributed real-time systems, protocol processing is important for predictable and analyzable systems. However, a number of priorities may be different in CPU domains and network domains. The mapping between these domains may cause a big effect to priority inversion so that careful mapping of priorities is important.

User Level Implementation: For real-time computing, high performance is also important. In our implementation, the synchronization module is implemented in a kernel space. The minimum cost is alway more than the cost of null system calls. User level threads make the cost of a thread management cheaper because it allow to work without the interaction with a kernel. Also, we need a high performance synchronization for multi threaded servers such as a network protocol server. Traditional user-level synchronization uses a spin-lock as light-weight synchronization. However, a spin-lock is not suitable in a single processor real-time system because the spin of a high priority thread causes the loss of CPU uilization easily. The cooperation of a run-time system and a kernel promises a better performance by disallowing a preemption in a short critical section.

Concurrency Control: Database systems use a concurrency control to preserve the consistency of data. In this case, a lock is held until the termination of a program. In concurrency control, the length of a critical section is long, then restartable critical section protocol is better when a high priority transaction cannot allow long waiting time. However, the abortion of a long-lived low priority transaction may cause expensive large rollback. We need a more heuristic approach which combines a priority inheritance and a restartable critical section.

5 Conclusion

We demonstrated that new real-time thread and synchronization facility in Real-Time Mach eliminate unbounded priority inversion problems among threads using critical sections. We also described the schedulability analysis for real-time programs in a single CPU environment.

Real-Time Mach provides several synchronization policies so that a suitable synchronization facility is chosen for each application. The tradeoff between the cost of synchronization and the utilization of systems is responsible of users.

However, these primitives alone cannot eliminate priority inversion problems in the systems. Our plan is to remodel the Mach IPC feature so that it can support a priority-based message handling and different types of transport protocols for real-time message transmission. We are implementing a new synchronization module which is integrated with IPC.

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, 1990.

[2] T. Nakajima and H. Tokuda "Implementation of Scheduling Policies in Real-Time Mach", In Proceeding of Second International Workshop on Object-Oriented Operating-Systems, 1992.

[3] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System", In Proceeding of USENIX Mach Workshop, October, 1990.

[4] H. Tokuda and T. Nakajima, "Evaluation of Real-Time Synchronization in Real-Time Mach", In Proceeding of USENIX 2nd Mach Symposium 1991.