Evaluation of Real-Time Synchronization in Real-Time Mach

Hideyuki Tokuda and Tatsuo Nakajima
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
hxt@cs.cmu.edu

Abstract

Real-Time Mach provides real-time thread and real-time synchronization facilities. A real-time thread can be created for a periodic or aperiodic activity with a timing constraint. Threads can be synchronized among them using a real-time version of the monitor based synchronization mechanism with a suitable locking protocol. In Real-Time Mach, we have implemented several locking policies, such as *kernelized monitor*, *basic priority priority inheritance protocol*, *priority ceiling protocol*, and *restartable critical section*, for real-time applications. It can also avoid a unbounded *priority inversion* problem.

In this paper, we describe the real-time synchronization facilities in Real-Time Mach and its implementation and performance evaluation. Our evaluation results demonstrated that a proper choice of locking policy can avoid unbounded priority inversions and improve the processor schedulability for real-time applications.

[1] This research was supported in part by the U.S. Naval Ocean Systems Center under contract number N66001-87-C-0155, by the Office of Naval Research under contract number N00014-84-K-0734, by the Defense Advanced Research Projects Agency, ARPA Order No. 7330 under contract number MDA72-90-C-0035, by the Federal Systems Division of IBM Corporation under University Agreement YA-278067, and by the SONY Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NOSC, ONR, DARPA, IBM, SONY, or the U.S. Government.

Contents

1 Introduction

Mach provides synchronization facility among threads using a conventional monitor-based mechanism[4]. However, in real-time applications, it is often encountered a *a priority inversion problem* among synchronizing activities. For instance, if there are many lower priority threads are already waiting on a mutex variable, then a higher priority thread which may have a tighter deadline cannot skip these waiting threads and must wait for their completion. Thus, the higher priority thread may miss its deadline. It often encounters when the system is under heavy load or transient overload.

Furthermore, if a real-time system designer need to determine the worst case blocking delay for a higher priority thread for a given shared resource which is protected by a mutex variable, it is often impossible to compute the bound if a thread in the protected region can be preemptable. The reason is that suppose the lowest priority thread T_L is in the critical region first, then the highest priority thread T_H becomes runnable and attempts to obtain the mutex variable. However, since T_L is in the critical region, T_H must wait for its completion. After T_L resumed, a medium priority thread T_{M1} becomes runnable. Then T_{M1} starts running without using the critical resource and wakes up another medium priority thread T_{M2} and so on. Under this type of interactions, the worst case blocking time of T_H cannot be determined by without knowing all behavior of related medium priority thread T_M's.

In order to bound the worst case blocking time, a simple solution called *priority inheritant* scheme was developed in our group [8, 7, 10]. An priority inheritance scheme is that once T_H blocks on the mutex variable, T_L inherits the priority of T_H. Then, T_{M1} cannot preempt the activity of T_L in the critical region. In this way, the worst case blocking time of T_H can be a function of the duration of the critical region, and not a function of the execution times of the medium priority tasks.

Real-Time Mach [11] provides a set of locking protocols for the mutex variable for sharing resources among real-time threads. We have implemented five locking protocols, namely *basic priority*, *kernelized monitor*, *basic priority inheritance protocol*, *priority ceiling protocol*, and *restartable critical section* to be used for various real-time applications. In this paper, we first describe the real-time synchronization facilities in Real-Time Mach. We then discuss the implementation and performance evaluation of these locking protocols.

2 Real-Time Thread and Synchronization

In this section, we briefly describe a real-time thread model we adopted in Real-Time Mach and the notion of schedulable bound for synchronizing real-time tasks in a single processor environment.

2.1 Real-Time Thread Model

A thread can be defined for a real-time or non-real-time activity. Each thread is specified by at least a procedure name and a stack descriptor which specifies the size and address of the local stack region. For a real-time thread, additional *timing attributes* must be defined by a timing attribute descriptor. A real-time thread can be also defined as a *hard* real-time or *soft* real-time thread. By a hard real-time thread, we mean that the thread must complete its activities by its *hard* deadline time, otherwise it will cause undesirable damage or a fatal error to the system.

The soft real-time thread, on the other hand, does not have such a hard deadline, and it still makes sense for the system to complete the thread even if it passed its critical (i.e. *soft* deadline) time.

A real-time thread can be also defined as a *periodic* or *aperiodic* thread based on the nature of its activity. A periodic thread P_i is defined by the worst case execution time C_i, period T_i, start time S_i, phase offset O_i, and task's semantic importance value V_i. In a periodic thread, a new instantiation of the thread will be scheduled at S_i and then repeat the activity in every T_i. The phase offset is used to adjust a ready time within each period. If a periodic thread is a soft real-time thread, it may need to express the abort time which tells the scheduler to abort the thread. An aperiodic thread AP_j is defined by the worst case execution time C_j, the worst case interarrival time A_j, deadline D_j, and task's semantic importance value V_i. In the case of soft real-time threads, A_j indicates the average case interarrival time and D_j represents the average response time. Abort time can be also defined for the soft real-time thread.

2.2 Real-Time Synchronization Analysis

Analyzing the schedulable bound of real-time tasks in general is a very difficult problem. In this section, we describe a simple yet very applicable case where only real-time periodic threads are synchronizing among them in a single processor environment[1].

To compute a schedulable bound, we must avoid a potential unbounded priority inversion problem among real-time threads. Once we can bound the worst case blocking time, then it can be computed a bound as an extension of the *rate monotonic* [6] scheduling analysis.

There are basically two approaches to bound the worst case blocking time among real-time tasks. One is a *kernelized monitor* and the other is a *priority inheritance* scheme as we described in Section 1.

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 $Thread_i$ and 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*[7].

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

[1]For a multiprocessor case, a simple priority inheritance case may not work and further extensions are necessary. Please look at [7].

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

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

3 Implementation

The current version of RT-Mach is being developed using a network of SUN, SONY workstations, laptop and single board target machines. The pure kernel provided us much better execution environment where we can reduce unexpected delays in the kernel and can run real-time threads without having a UNIX server. The preemptability of the kernel was also improved significantly since UNIX primitives and some device drivers are no longer in the kernel.

3.1 Real-Time Synchronization

The synchronization mechanism is based on mutual exclusion with a lock variable [2]. A thread can allocate, deallocate, and initialize a lock variable with a suitable *lock attribute*. The lock attribute specifies a synchronization policy which determines its queueing policy and priority ordering.

In the current implementation, kernelized monitor (KM), basic policy (BP), basic priority inheritance protocol (BPI), priority ceiling protocol(PCP), and restartable critical region(RCS) are supported. In the KM protocol, if a thread enters to the kernelized monitor region, all preemption is prevented. Thus, the duration of the critical section must be shorter than any real-time thread's deadline.

The BP policy on the other hand, simply enqueues waiting threads in the lock variable based on the thread's priority. BPI provides the inheritant function that a lower priority thread executing the critical section inherits the priority of higher priority thread, when the lock is conflicted. In the PCP protocol, the ceiling priority of the lock is defined as the priority of the highest priority thread that may lock the lock variable. 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. For the RCS policy, a higher priority thread is able to abort the lower priority thread in the critical section and puts it back to the waiting queue with recovering the state of shared variable. 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.

A simple pair of *rt_mutex_lock* and *rt_mutex_unlock* primitives is used to specify mutual exclusion. The *rt_mutex_trylock* primitive is used for acquiring the lock conditionally. A modified version of the condition variable is also supported for specifying a conditional critical region. A pair of *rt_condition_signal* and *rt_condition_wait* primitives is used to synchronize over a condition variable.

The interface of and brief description of the synchronization functions and lock attribute are as follows.

[2] We have also created event-based primitives: **rt_event_send**(event, event_attr), **rt_event_receive**(event, event_attr, timeout). However, we do not describe these primitives in this paper.

```
kval_t = rt_mutex_allocate( lock, lock_attr )
kval_t = rt_mutex_deallocate( lock )
kval_t = rt_mutex_lock( lock, timeout, context )
kval_t = rt_mutex_unlock( lock )
kval_t = rt_mutex_trylock( lock )
kval_t = rt_condition_allocate( cond, cond_attr )
kval_t = rt_condition_deallocate( cond )
kval_t = rt_condition_wait( cond, lock, cond_attr, timeout )
kval_t = rt_condition_signal( cond, cond_attr )
```

```
typedef struct lock_attr {
        type_t       rt_type;          /* lock type */
        priority_t   rt_priority;      /* ceiling priority */
                     ...
} lock_attr_t;

typedef struct cond_attr {
        type_t       rt_type;          /* condition variable type */
        priority_t   rt_priority;      /* for priority inheritance */
                     ...
} cond_attr_t;
```

rt_type indicates a lock policy given by users. *rt_priority* is used for the ceiling protocol and specifies the ceiling priority in *lock_attr*. If NULL value is set as a lock attribute, a default policy (i.e., BP, basic policy) is chosen. In *cond_attr*, *rt_type* indicates the type of condition variable, and *rt_priority* is used for priority inheritance.

4 Evaluation and Cost Analysis

In this section, we first measure the basic cost of our primitives and the cost for managing real-time thread. We then analyze the relation between the cost of various real-time synchronization protocols.

The basic cost of the real-time thread management and synchronization primitives were measured using a Sony NEWS-1720 workstation (25 MHz MC68030) and a FORCE CPU-30 board (20 MHz MC68030). We simply evaluated a single processor environment with a Sony machine. We used an accurate clock on the FORCE board for timing measurement on a NEWS-1720 through VME-bus backplane. This clock enabled us to measure the overheads with resolution of 4 μ s.

4.1 Preemption Cost Analysis

Before we start analyzing the preemptability of the system, let us first determine the basic cost factors. Figure 1 defines the basic cost factors when a higher priority thread preempts a lower priority thread which is executing a system call. C_{opr} specifies the execution cost of the primitive *opr*. C_{non_int} is the worst case execution time of a non-interrupt region where all interrupts are

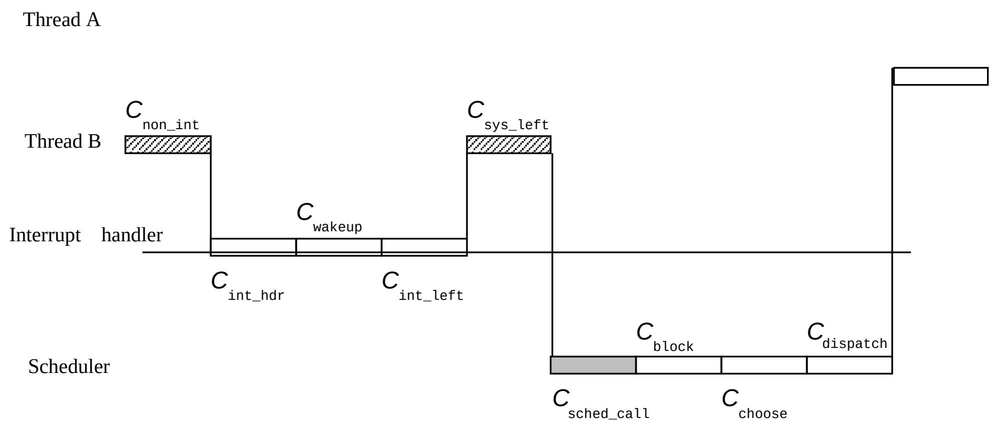

Figure 1: Preemption during a System Call

masked. A critical interrupt may be delayed until the non-critical region is completed. C_{int_hdr} is the worst case execution time of the interrupt handlers. Interrupt handler can be interrupted by a higher priority interrupt. C_{wakeup} is the time to wakeup a blocked thread. C_{int_left} is the remaining time after the wakeup until the interrupt is completed. C_{sys_left} is the remaining execution time of the system call. C_{sched} is the total scheduling delay time and sum of C_{sched_call}, C_{block}, C_{choose}, and $C_{dispatch}$. C_{sched_call} is the delay time to switch to the scheduler. C_{block} is the blocking time for giving up the CPU and C_{choose} is the selection time for a next thread, $C_{dispatch}$ is the context switching time. The results of the measurement are summarized in Table 1.

Now, let us consider the preemption cost in RT-Mach. The total worst case preemption cost can be defined as

$$C_{preempt} = C_{non\_int} + C_{int\_hdr} + C_{wakeup} + C_{int\_left} + C_{sys\_left} + C_{sched}$$
$$C_{sched} = C_{call\_sched} + C_{block} + C_{choose} + C_{dispatch}$$

Under the fixed priority scheduling, $C_{preempt}$ becomes $C_{non\_int}+C_{int\_hdr}+C_{int\_left}+C_{sys\_left}+$ 196 μs. In our target machine, a clock interrupt handler alone requires at least additional $C_{clockint} = C_{int_hdr} + C_{int_left} = 108\mu$s. In a real-time application, the cost of $C_{int\_hdr} + C_{int\_left}$ can be precomputed based on the system configuration, however, the cost for C_{non_int} and C_{sys_left} are operating system specific. In many monolithic kernel-based system, C_{non_int} and C_{sys_left} becomes relatively high. However, in a micro kernel-based system, an ordinary system call becomes preemptive since its function is implemented in a user-level thread. For real-time programs which have shorter deadlines than $C_{preempt}$, we need to reduce each cost factor further down. To reduce C_{non_int} and C_{sys_left}, further kernelization of the current micro kernel is required.

4.2 Synchronization Cost Analysis

In this section, we will analyze the synchronization cost for each locking protocols we have implemented. In general, it would be ideal if the synchronization cost for locking, C_{lock} or unlocking C_{unlock} is independent from the number of waiting threads. Among all implemented policies, KM alone provides this property. However, unlike other policies, KM does not allow

Basic Operation	Cost (μs)
C_{wakeup}	72 †1
C_{sched_call}	72
$C_{int\_left(clock)}$	36
$C_{block_{reincar}}$	672 †1
C_{block}	84 †1
C_{choose}	40 †1
$C_{dispatch}$	48
C_{null_trap}	48
$C_{clockint}$	108 †2

†1 C_{block}, C_{wakeup} and C_{choose}, are measured under a fixed priority scheduling policy (default) and are policy specific numbers.
†2 This includes the cost calling scheduling policy routines, but no thread wakeup cost.

Table 1: The Basic Overhead

	Basic (μs)	KM (μs)	BPI (μ)	PCP (μs)	RCS (μs)
Lock(Acquired)	120	132	196	232	256
Lock (Not Acquired)	208 + C_{block}	N.A.†1	340 + C_{block}	508 + C_{block}	628 + C_{block}
Unlock	144 + $C_{activate}$ × n	156/180 †3	192	256	196

†1 No one can preempt.
†2 m is a number of chains of chained locks.
†3 The cost of right parts includes the cost to wakeup the thread waiting a lock.
†4 n is a number of nest of nested locks.
†5 $C_{activate}$ is the cost for make a blocking thread runnable. The measured cost is 108 μs.

Table 2: The Cost of Locking Primitives

any preemption during a critical section. Thus, a user cannot create a real-time thread whose deadline is shorter than the size of the critical section.

In general, KM tents to be useful for a very short critical section. For medium size critical sections, we can use one of priority inheritance protocols. For very large critical sections, on the other hand, it may be better to use a restartable critical section scheme if the higher priority thread cannot afford to wait for the lower priority thread to complete its critical section.

Table 2 summarizes the execution costs of primitives, *lock and acquired*, *lock and not acquired*, and *unlock* under the five locking protocols; basic default (Basic), kernelized monitor (KM), basic priority inheritance (BPI), priority ceiling protocol (PCP), and restartable critical section (RCS).

Let us now calculate the cost of locking policies, and determine which protocol is better under what conditions. Here, $C^p_{nullcs(1)}$ indicates the time for a single thread to execute a pair of lock and unlock operations under a given locking protocol p and is define as $C^p_{nullcs(1)} = C^p_{syscall} + C^p_{acquired} + C^p_{unlock}$.

The measured results for KM, BPI, and PCP are as follows.

$$C^{km}_{nullcs(1)} = C^{km}_{syscall} + C^{km}_{acquired} + C^{km}_{unlock} = 288\mu s$$
$$C^{bpi}_{nullcs(1)} = C^{bpi}_{syscall} + C^{bpi}_{acquired} + C^{bpi}_{unlock} = 388\mu s$$
$$C^{pcp}_{nullcs(1)} = C^{pcp}_{syscall} + C^{pcp}_{acquired} + C^{pcp}_{unlock} = 488\mu s$$

Let us determine the worst case blocking time of the highest priority thread where n threads are sharing the same critical resource. The worst case for the highest priority thread may occur when a lower priority thread is already in the critical section. Then, under BPI or PCP policies, the highest priority thread becomes runnable and preempts the lowest priority thread and attempts to enter the critical section. Since the resource is already in use, the highest priority thread's lock request will blocks itself and resume the lowest priority thread with the highest priority (due to priority inversion). Additional overhead will occur when all other threads becomes runnable while the lowest priority thread is executing in the critical section.

Under this assumption, we can determine the worst case blocking time $C^{p}_{wait(n)}$ for the highest priority thread under locking protocol p as follows.

$$C^{km}_{wait(n)} = C_{csbody} + (n-2) \times C_{activate} + C^{km}_{unlock} + C_{itds\_csw\_check}$$
$$C^{bpi}_{wait(n)} = C_{csbody} + (C_{wakeup} + C_{switch} + C_{lock\_not\_acquire} + C_{choose} + C_{dispatch}) + (n-2) \times C_{activate} + C^{bpi}_{unlo}$$
$$C^{pcp}_{wait(n)} = C_{csbody} + (C_{wakeup} + C_{switch} + C_{lock\_not\_acquire} + C_{choose} + C_{dispatch}) + (n-2) \times C_{activate} + C^{pcp}_{unlo}$$

where C_{csbody} indicates the cost for executing the body of the critical section alone and $C_{itds\_csw\_check}$ is the cost for determining the next runnable thread with the highest priority among the waiting threads.

From Table 1 and 2, we can determine the cost of thread switching C_{switch} (172 μs $=$ $C_{block} + C_{choose} + C_{block}$), C_{wakeup} is 72 μs, $C_{activate}$ is 108 μs, and C_{unlock} for each protocol. Then, we can derive

$$C^{km}_{wait(n)} = C_{csbody} + 180 + (n-2) \times 108\mu\text{s}$$
$$C^{bpi}_{wait(n)} = C_{csbody} + 948 + (n-2) \times 108\mu\text{s}$$
$$C^{pcp}_{wait(n)} = C_{csbody} + 1180 + (n-2) \times 108\mu\text{s}$$

From these numbers, we can conclude that as long as the system does not have any real-time thread whose deadline is shorter than $C_{csbody} + 288\mu$s and the deadline of a real-time thread which shares the critical section is greater than $2 \times C_{csbody} + 108 \times n - 36\mu$s, then we should be able to use the KM policy among n real-time thread ($n >= 2$).

In general, the BPI and PCP protocols are suitable for a large critical section. For instance, among two threads, the critical section should be longer than $1336 (= 388 + 948)\mu$s for BPI. Otherwise, it is better to use KM, if there is no real-time thread whose deadline is shorter than that value. Similarly for two threads, the minimum length for PCP is 1436μs.

For a restartable critical section, we could get the restarting cost, $C^{rcs}_{restart}$ as follows.

$$C^{rcs}_{restart} = C_{abort}\ (= 816\ \mu\text{s}) + C_{recover}$$

where C_{abort} is a cost for aborting a lower priority thread and $C_{recover}$ is a cost for recovering the state of shared resource. $C_{recover}$ is application specific and there are many different schemes exit. From this restarting cost, we can also conclude that the restartable critical section is a suitable policy if $C_{restart}$ is less than $C^{bpi}_{wait(n)}$ or $C^{pcp}_{wait(n)}$.

5 Conclusion and Future Work

We demonstrated that using new real-time thread and synchronization facility in Real-Time Mach, a user could eliminate unbounded priority inversion problems among synchronizing threads and could perform schedulability analysis for real-time programs in a single CPU environment. The system interface for creating periodic and aperiodic threads, creating a mutex variable, and locking and unlocking mutex was natural and easy to specify a user policy for a specific application.

We also analyzed the performance of proposed locking protocols, KM, BP, BPI, PCP, and RCS, and determined the worst case blocking cost for real-time threads. From the analysis, we could determine which policy should be effective under what conditions.

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. The basic issues has been discussed and evaluated in our experimental tested [9, 10]. For instance, a priority inversion may occur when a client requests for a service to a non-preemptive server. A higher-priority client may face an unbounded priority inversion problem at the server. To avoid this problem, one solution is to add a *port attribute* for each port and allow us to set a different priority inheritance policy, like the lock attribute we implemented. Real-time network support is also important and we are planning to create a new netserver facility where protocol processing will be performed by multiple worker threads based on a message priority.

References

[1] M.J. Accetta, W. Baron, R.V. Bolosky, D.B. Golub, R.F. Rashid, A. Tevanian, and M.W. Young, "Mach: A new kernel foundation for unix development", *In Proceedings of the Summer Usenix Conference,*, July, 1986.

[2] Özalp Babaoglu, "Fault-Tolerant Computing Based on Mach", In *Proceedings of USENIX Mach Workshop*, October, 1990.

[3] R. Chen and T.P. Ng, "Building a Fault-Tolerant System Based on Mach", In *Proceedings of USENIX Mach Workshop*, October, 1990.

[4] E. C. Cooper, and R. P. Draves, "C threads", Technical report, Computer Science Department, Carnegie Mellon University, CMU-CS-88-154, March, 1987.

[5] D. Golub, R. Dean, A. Forin, and R. Rashid, "Unix as an application program", *In the proceedings of Summer Usenix Conference*, June, 1990.

[6] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment", *Journal of the ACM,* Vol.20, No.1, 1973.

[7] R. Rajkumar, "Task Synchronization in Real-Time Systems", Ph.D. Dissertation, Carnegie Mellon University, August, 1989.

[8] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization", Technical Report CMU-CS-87-181, Carnegie Mellon University, November 1987

[9] H. Tokuda and C. W. Mercer, "ARTS: A distributed real-time kernel", *ACM Operating Systems Review*, Vol.23, No.3, July, 1989.

[10] H. Tokuda, C. W. Mercer, Y. Ishikawa, and T. E. Marchok, “Priority inversions in real-time communication”, In *Proceedings of 10th IEEE Real-Time Systems Symposium*, December, 1989.

[11] H. Tokuda, T. Nakajima, and P. Rao, “Real-Time Mach: Towards a Predictable Real-Time System”, In *Proceedings of USENIX Mach Workshop*, October, 1990.