Evaluation of Real-Time Synchronization in Real-Time Mach

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

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

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 Introduction

A new challenge in advanced real-time systems is not only creating a fast and responsive kernel, but providing a predictable and analyzable real-time computing environment. An advanced real-time kernel should allow a system designer to analyze the runtime behavior at the design stage and predict whether the given real-time tasks having various types of system interactions can meet their timing requirements.

The objective of Real-Time Mach is to create a real-time version of Mach operating system which can provide such predictable real-time computing environment. However, in the Mach kernel, it is often difficult to analyze the runtime behaviour of the time critical activities due to lack of real-time scheduling and synchronization facilities. For instance, Mach provides synchronization facility among threads using a conventional monitor-based mechanism[5]. Threads can enter a critical section in FIFO order, however, this may cause a *a priority inversion problem* among threads. If many lower priority threads are already waiting on a mutex variable, a higher priority thread which may have a tighter deadline must wait for their completion. Thus, the higher priority thread may miss its deadline.

Furthermore, a real-time system designer need to determine the worst case blocking delay for a higher priority thread for the shared resource. It is often impossible to compute the bound if a thread in the protected region can be preemptable. Let us consider the following case. Suppose that 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 inheritance* scheme were developed in our group [12, 10, 14]. 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 [15] provides a set of locking protocols for the mutex variable for sharing resources among real-time threads. We have implemented five locking protocols, namely *kernelized monitor*, *basic priority*, *basic priority inheritance*

[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.

protocol, *priority ceiling protocol*, and *restartable critical section* to be used for various real-time applications.

In this paper, we describe the implementation and performance evaluation of real-time synchronization facilities in Real-Time Mach. In Section 2, we first introduce the real-time thread model and synchronization facilities in Real-Time Mach. Section 3 discusses the extended system interface for real-time synchronization and the implementation of synchronization mechanisms and policy modules. In Section 4, we also describe the performance evaluation of thread preemption and the locking protocols. Related work is discussed in Section 5 and we describe the conclusion and future work in Section 6.

2 Real-Time Thread and Synchronization Model

In this section, we briefly describe a real-time thread and synchronization 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 Model

In a real-time synchronization model, we have created various synchronization policies based on two basic factors: one is a queueing order among waiting threads and the other is preemptability of the running thread in the critical section.

Traditional synchronization primitives use FIFO ordering among waiting threads 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 thread must wait for the completion of all low priority threads in the waiting queue. If all of real-time threads can meet their deadlines, then there will be no starvation among these threads. Thus, in real-time synchronization, the system should provide a deadline based (or priority based) ordering to avoid unbound priority inversion problems.

Preemptability of the running thread in the critical section also 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 for schedulability and a different cost to enter to and exit from critical sections. In the following section, we will show the difference between synchronization protocols. The result allows application programmer to select suitable synchronization protocols for their applications.

2.3 Real-Time Synchronization Analysis

Analyzing the schedulable bound for real-time tasks is a very difficult problem. We have developed a simple yet very applicable scheme for analyzing real-time periodic threads which are synchronizing 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 could bound the worst case blocking time, then we can compute a bound as an extension of the *rate monotonic* scheduling analysis [8].

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*[10].

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 Implementation

The current version of Real-Time Mach is being developed using a network of SUN, SONY workstations, laptop and single board target machines. The pure kernel 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. In this section, we will describe the extended system interface for real-time synchronization and the implementation of synchronization mechanisms and policy modules.

3.1 System Interface for 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 and priority ordering.

A simple pair of *rt_mutex_lock* and *rt_mutex_unlock* primitives is used to specify mutual exclusion. The *rt_mutex_trylock*

[1]For a multiprocessor case, a simple priority inheritance case may not work and further extensions are necessary. Please refer[10].

[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.

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 the synchronization functions and lock attribute are summarized as follows.

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

3.2 Synchronization Module

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 (or an abstract data type). The communication between the policy object and its mechanism was done through function calls, not message passing.

The synchronization module is divided into the common lock object and lock policy objects. Figure 1 shows the relationship between the common lock object and various lock policy objects. Only one common lock object is created, while the lock policy object is created for each lock object and controls the priority of the thread which is holding the lock object.

The common lock object offers the mechanism to manage the blocking and unblocking of threads for exclusive access It also manipulates the queues of threads waiting for the releasing the lock according to the current processor scheduling policy. The lock object up calls its associated lock policy object. Some of lock policy objects also controls the thread's effective priority for controlling a priority inheritance scheme.

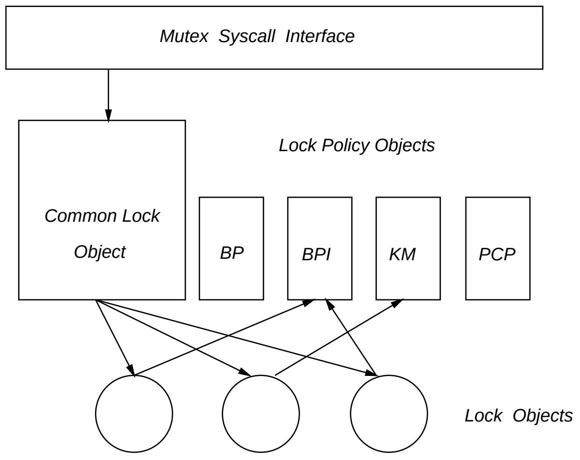

Figure 1: Lock Policy Module

3.2.1 Interface for Policy Control

The policy operations for the lock object are triggered by a system call from a user. The thread executes these operations without blocking. If the thread must be blocked inside these operations, it returns to the common lock object. The common lock object determines whether to block the thread or not, and re-execute the policy operations.

The lock object and policy object provide the following operations.

kval_t = **rt_mutex_policy_acquired**(lock)
kval_t = **rt_mutex_policy_notacquired**(lock)
kval_t = **rt_mutex_policy_conflict**(lock)
kval_t = **rt_mutex_policy_unlock**(lock, lock_attr)
kval_t = **rt_mutex_policy_abort**(lock, mode, who)
kval_t = **rt_mutex_policy_control**(lock, cmd, arg, argsize)

rt_mutex_lock_acquired is called when the mechanism layer needs to acquire the lock. This operation keeps track the current thread's priority for the lock object. *rt_mutex_lock_not_acquired* is used when the mechanism module cannot acquire the lock. In the case of basic priority inheritance protocol, the operation inherits the higher thread's priority. *rt_mutex_lock_unlock* is invoked from *rt_mutex_unlock*. It resets the lock structure. In the case of priority inheritance, the operation recovers the priority of the thread which executes *unlock*. *rt_mutex_conflict* is invoked when *lock* is called. It checks whether we need to abort the current critical section. *rt_mutex_lock_abort* is used when the lock is aborted. The operation is called in two ways: one is when *lock* or *unlock* operation fails and the other is when the thread in the locked region need to be aborted. The operation is also called when timeout occurred. *rt_mutex_control* is invoked to control policy module such as set and get default lock policy.

3.2.2 Policy Module

A lock policy module is implemented as an object similar to the scheduling policy object [15]. The following lock policy objects are supported.

Kernelized Monitor (KM): No preemption is allowed while a thread is in the kernelized critical section. The duration of the priority inversion is bounded by the size of the critical region. *rt_mutex_policy_aquired* notifies the processor scheduler for entering the kernelized monitor and *rt_mutex_policy_unlock* tells the thread to exit from the kernelized critical section.

Basic Priority (BP): All operations of this policy object are null functions. The waiting threads are enqueued in the lock object based on the thread's priority.

Basic Priority Inheritance (BPI): The lower priority thread executing the critical section inherits the priority of higher priority thread, when the lock is conflicted. Priority inheritance is processed in *rt_mutex_policy_noacquired* and *rt_mutex_policy_unlock* recovers the inherited priority to the thread's original priority.

Priority Ceiling Protocol (PCP): The execution of 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 policy module has a global queue to hold the currently acquired locks. *rt_mutex_policy_acquired* checks the ceiling priority of locks in the global lock queues with the thread which will enter the critical section to determine the ceiling block of the thread.

Restartable Critical Section (RCS): The critical section is aborted by the higher priority threads and restarted after the higher priority thread is completed. *rt_mutex_conflict* is called in *rt_mutex_lock* and aborts the execution of the critical section of the current running thread if lower priority thread is executing the critical section.

The lock policy object is easy to define. Among all policies, the priority ceiling protocol is the largest policy module. The priority inheritance scheme is used in both the basic priority inheritance protocol and priority ceiling protocol. The code can be shared in both object. The advantage of the approach is not only the reduction of code size, but also it makes clear definition of each algorithm. For instance, priority ceiling algorithm is separated into priority inheritance management and priority ceiling management part.

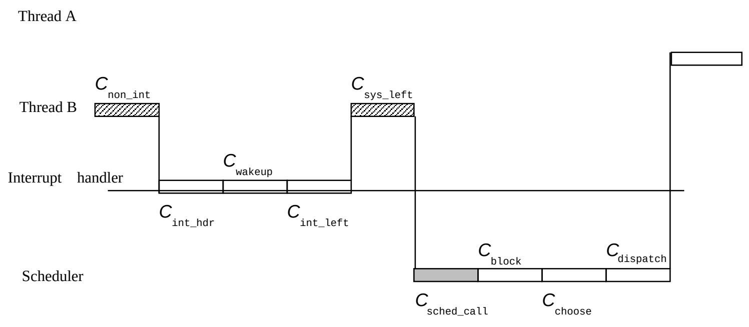

Figure 2: Preemption during a System Call

4 Evaluation and Cost Analysis

In this section, we first discuss the cost for managing real-time thread and the synchronization cost under various synchronization policies. We then analyze the relation between the schedulability and 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 μs.

4.1 Preemption Cost Analysis

Before we start analyzing the preemptability of the system, let us first determine the basic cost factors. Figure 2 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 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

Basic Operation	Cost (μs)
C_{wakeup}	72 †1
C_{sched_call}	72
$C_{int\_left(clock)}$	36
$C^{\dagger 1}_{block_{reincar}}$	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.

†3 $C_{block_{reincar}}$ is the blocking cost at the reincarnation of a period thread.

Table 1: The Basic Overhead

summarized in Table 1.

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

$$
\begin{aligned}
C_{preempt} &= C_{non\_int} + C_{int\_hdr} + C_{wakeup} \\
&\quad + C_{int\_left} + C_{sys\_left} + C_{sched} \\
C_{sched} &= C_{sched\_call} + C_{block} + C_{choose} + C_{dispatch}
\end{aligned}
$$

Under the fixed priority scheduling, $C_{preempt}$ becomes $C_{non_int} + C_{int_hdr} + C_{int_left} + C_{sys_left} + 316\ \mu$s. 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 systems, 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 protocol we have implemented.

In general, KM tends 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 Priority (BP), kernelized monitor (KM), basic priority inheritance (BPI), priority ceiling protocol (PCP), and restartable critical section (RCS). The *lock and acquired* cost includes the cost of trap handling, *rt_mutex_lock* in the common lock object, and *rt_mutex_policy_aquired* function in a policy object. The *lock and not acquired* cost includes only the cost for executing *rt_mutex_policy_notacquired* in a policy module. The *unlock* cost includes the cost of trap handling, *rt_mutex_unlock* function in the common lock object, and *rt_mutex_policy_unlock* function in a policy object.

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.

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

From the above results, it was clear that the implementation cost increased proportional to the complexity of the locking protocol.

Let us then determine the worst case blocking time of the highest priority thread where n threads are sharing the same critical lock 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 lower 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 lower priority thread with the highest priority (due to priority inversion). Additional overhead may occur when all other threads become runnable while the lower priority thread is executing in the critical section[1].

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.

$$
\begin{aligned}
C^{km}_{wait(n)} &= C_{csbody} + (n-1) \times C_{activate} + C^{km}_{unlock} \\
C^{bpi}_{wait(n)} &= C_{csbody} + (C_{wakeup} + C_{switch} + C_{lock\_not\_acquire} \\
&\quad + C_{choose} + C_{dispatch}) + (n-2) \times C_{activate} + C^{bpi}_{unlock} \\
C^{pcp}_{wait(n)} &= C_{csbody} + (C_{wakeup} + C_{switch} + C_{lock\_not\_acquire} \\
&\quad + C_{choose} + C_{dispatch}) + (n-2) \times C_{activate} + C^{pcp}_{unlock}
\end{aligned}
$$

where C_{csbody} indicates the cost for executing the body of the critical section alone, C_{switch} represents the total thread switching cost, and $C_{activate}$ is the cost for making a blocked thread runnable.

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

[1]Assuming that the system does not disable all interrupts in KM.

	BP (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 + $C_{activate} \times$ 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 making a blocked thread runnable. The measured cost is 108 μs.

Table 2: The Cost of Locking Primitives

$$C^{km}_{wait(n)} = C_{csbody} + 180 + (n - 1) \times 108\mu s$$
$$C^{bpi}_{wait(n)} = C_{csbody} + 948 + (n - 2) \times 108\mu s$$
$$C^{pcp}_{wait(n)} = C_{csbody} + 1180 + (n - 2) \times 108\mu s$$

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} + 288μs and the deadline of the real-time thread which shares the critical section is greater than $2 \times C_{csbody} + 108 \times n + 72\mu$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 synchronizing threads, the deadline of the threads should be longer than $2 \times C_{csbody} + 1336 (= 948 + 388)\mu$s for BPI. Otherwise, it may need to use KM, if there is no real-time thread whose deadline is shorter than $2 \times C_{csbody} + 288 (= 216 + 72)\mu$s. Similarly for two threads under PCP, the deadline should be larger than $2 \times C_{csbody} + 1668\mu$s.

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

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

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 $C^{pcp}_{wait(n)}$.

4.3 Schedulability Test: Case I

In the following section, we will compare the schedulability of various periodic threads under different locking protocols: KM, BP, BPI, PCP, and RCS using few benchmark programs. In this Benchmark-1 program, there are four periodic threads: Th_A, Th_B, Th_C and Th_D. Th_A has the highest priority, Th_B and Th_C are medium, and Th_D is the lowest priority. Th_D is executed first, Th_B and Th_C are executed next. Lastly, Th_A is executed. Th_A and Th_D share the same object. At the beginning of the execution, each thread locks the shared object and releases the object at the end. The timing attributes of each thread are given in Table 3. C_i represents the worst case execution time of thread Th_i. B_i indicates the worst case blocking time of Th_i and T_i is the period of Th_i. U_i is the processor utilization of Th_i and S_i indicates the start time of aperiodic thread Th_i.

Thread	T_i (ms)	C_i (ms)	S_i (ms)	U_i (%)
Thread A	100	10	20	10
Thread B	300	EXE	10	$\frac{EXE}{300}$
Thread C	400	60	10	15
Thread D	1000	30	0	3

Table 3: Thread Attributes of Benchmark-1

Lock Policy	Max Exec Time (ms)	U_i (%)
BP	14	32.6
BPI	181	88.3
KM	183	89
PCP	181	88.3
RCS	202	95.3

Table 4: Breakdown Utilization under Benchmark-1

For each locking protocol, we compared the breakdown processor utilization while we vary the execution time of Th_C (i.e., EXE in Table 3). Table 4 shows the execution results of Benchmark-1. The result indicates that the priority inversion degrades the schedulability significantly and priority queueing alone does not solve the priority inversion problem. Thus, KM, BPI, PCP, or RCS is a very effective policy in real-time synchronization. Also, this benchmark result indicates that RCS policy could achieve the highest schedulability among these policies, if $C^{rcs}_{restart}$ is significantly small. Since it can abort the critical section of the lower priority thread, the higher priority thread need not to wait for the termination of the lower priority thread.

4.4 Schedulability Test: Case II

The previous benchmark shows that KM, BPI, and PCP policy may achieve relatively high schedulability for the given taskset.

Benchmark-2 presents that this is not always true under different taskset. In this program, we consider two threads, Th_A and Th_B. Th_A is the highest priority thread, and it is started 10

Thread	T_i (ms)	C_i (ms)	S_i (ms)	U_i (%)
Thread A	100	20	10	20
Thread B	200	EXE	0	$\frac{EXE}{200}$

Table 5: Thread Attributes of Benchmark-2

Lock Policy	Max Exec Time (ms)	U_i (%)
BPI	155	97.5
KM	88	64
PCP	87	63.5

Table 6: BPI, KM, PCP under Benchmark-2

ms after Th_B starts. The timing attributes of Th_A and Th_B are given in Table 5. Th_B contains a critical section, and we vary the length of the critical section during the test.

We compared BPI with KM and PCP under Benchmark-2. Table 6 summarize the breakdown utilization of each policy. Although PCP has nice properties such as deadlock free property, the result indicates that PCP could not achieve at the same level as BPI scheme. If the taskset is deadlock free in nature, then BPI policy is the best scheme.

We also compared BPI and RCS policy under Benchmark-3 program. In general, RCS becomes useful if a higher priority thread cannot afford to wait for a lower priority thread to complete the critical section. However, restarting a critical section requires not only the cost of aborting of the critical section but also the restarting cost. The schedulability may be decreased, if these costs dominates the total overhead.

Unlike Benchmark-1, Benchmark-3 indicates the case where RCS has significantly lower break down utilization than BPI.

Various benchmark programs we presented in this section indicates that there is no best policy for all real-time applications. A system designer needs to select a suitable policy for their applications.

5 Related Work

The micro kernel-based approach is gaining popularity and several micro kernel-based operating systems have been developed for an advanced distributed computing environment [4, 9, 11].

Advantages of using a micro kernel instead of a standard monolithic kernel is its high preemptability, small size, and extensibility. However, only a few micro kernels were designed

Thread	T_i (ms)	C_i (ms)	S_i (ms)	U_i (%)
Thread A	100	20	50	20
Thread B	150	EXE	60	$\frac{EXE}{150}$
Thread C	200	60	0	30

Table 7: Thread Attributes in Benchmark-3

Lock Policy	Max Exec Time (ms)	U_i (%)
BPI	60	90
RCS	15	60

Table 8: BPI vs. RCS under Benchmark-3

for supporting predictable distributed real-time computing environment.

For instance, Chorus's micro kernel[11] was designed for real-time applications. However, their emphasis was placed at rather low-level kernel functions such as providing a user-defined interrupt handler and preemptive kernel. The kernel uses the wired-in fixed priority preemptive scheduling policy and there is no additional features reported to avoid priority inversion problems.

The V kernel's emphasis was also intended for supporting high speed real-time applications[4]. V's optimized message passing mechanism and VMTP protocol can provide basic functions for building distributed real-time applications. However, the wired-in scheduling policy and locking protocol may cause us a potential inversion problem.

Amoeba's advantage is its high performance RPC and was used for remote video image transmission using Ethernet. Like Real-Time Mach, Amoeba can support a set of single board computers without having local disks, however, it does not provide any safe mechanisms for creating a periodic thread and avoiding priority inversion problems.

The POSIX-Thread proposal[7] is very similar to Mach's C-Thread package[5]. However, it also does not distinguish between real-time threads and non-real-time threads. Like Real-Time Mach, it can dynamically select the thread scheduling policy. A POSIX thread also contains the thread attributes such as *inherit priority*, *scheduling priority*, *scheduling policy*, and *minimum stack size*. Thus, adding our timing attributes into the proposed POSIX interface would be very simple.

Uniqueness of Real-Time Mach is based on our real-time thread and synchrozation model. The proposed real-time thread model is different from many other thread models. Our model distinguishes between real-time and non real-time threads and we provide explicit timing constraints for each real-time thread. A suitable synchrozation policy such as KM, BPI, PCP, RCS can be selected to avoid unbounded priority inversion and to improve the schedulability of the given taskset.

6 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. We also described the 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 [13, 14]. 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] D.R. Cheriton, G.R. Whitehead and E.D. Sznyter, "Binary emulation of UNIX using V Kernel", *In proceedings of Summer Usenix Conference,* June, 1990.

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

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

[7] IEEE, "Realtime Extension for Portable Operating Systems", P1003.4/Draft6, February, 1989.

[8] 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.

[9] S.J. Mullender, G.V. Rossum, A.S. Tanenbaum, R. Renesse and H. Staveren, "Amoeba: A Distributed Operating System for the 1990s", *IEEE Computer* Vol.23, No.5, May, 1990

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

[11] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemount, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, "Chorus distributed operating system", *Computing Systems Journal*, The Usenix Association, December, 1988

[12] 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

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

[14] 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.

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