Real-Time Mach:
Towards a Predictable Real-Time System

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

Abstract

Distributed real-time systems play a very important role in our modern society. They are used in aircraft control, communication systems, military command and control systems, factory automation, and robotics. However, satisfying the rigid timing requirements of various real-time activities in distributed real-time systems often requires *ad hoc* methods to tune the system's runtime behavior

The objective of Real-Time Mach is to develop a real-time version of the Mach kernel which provides users with a predictable and reliable distributed real-time computing environment. In this paper, we describe a real-time thread model, real-time synchronization, and the ITDS scheduler in Real-Time Mach. We also discuss the implementation issues, a real-time toolset, and the current status of the system.

1 Introduction

Distributed real-time systems are becoming more common as real-time technology is applied to many real-time applications[20]. However, satisfying the rigid timing requirements of various real-time activities in distributed real-time systems is getting more complex due to the distributed nature of the system.

In many cases, system designers of such complex systems lack systematic development methods and analysis tools, so they resort to *ad hoc* methods to develop, test, and verify real-time systems. For processor scheduling, for instance, the cyclic executive model which uses time line analysis to schedule real-time activities is not suitable for a distributed environment. It is very difficult to test and tune the executive based on some changes in the task set or its timing requirements for complex real-time systems [9]. Message communication scheduling in a network is also difficult since some communication media such as Ethernet do not guarantee bounded communication delay at media access level and do not provide priority-based arbitration.

A new challenge in such real-time systems is to develop a real-time kernel which can provide users with a predictable and reliable distributed real-time computing environment. In particular, the 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 and task interactions (e.g., memory allocation/deallocation, message communications, I/O interactions, etc) can meet their timing requirements.

CMU's ART (Advanced Real-Time Technology) group has been working on a real-time version of the Mach as well as a real-time toolset for system design and analysis. Real-Time Mach is being developed based on a version of the pure kernel [1, 7] using a network of SUN, SONY workstations, and single board target machines. Unlike the standard release 2.5 Mach, this kernel includes new real-time thread management, an integrated time-driven scheduler (ITDS), real-time synchronization, and memory resident objects. The real-time thread model is based on the ARTS real-time thread model [21, 25] and our real-time scheduling theories. Real-Time Mach was also integrated with our real-time toolset, *Scheduler 1-2-3*[23] and Advanced Real-Time Monitor, *ARM*[22].

In this paper, we describe new system facilities in Real-Time Mach and the current status. In Section 2, we first introduce the real-time thread model, real-time synchronization primitives, integrated time-driven scheduler, and support for memory resident objects. Section 3 discusses implementation issues and our solution to priority inversion problems. In Section 4, we also compare our approach and real-time thread model to other operating systems. Section 5 summarizes the development status and considers future work.

[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.

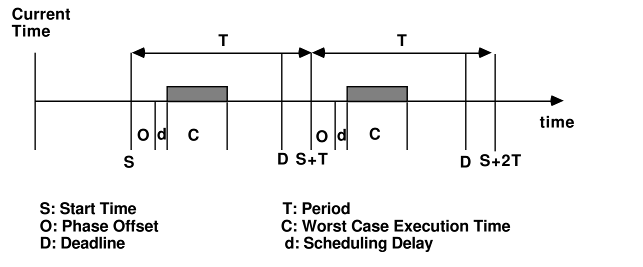


Figure 1: Timing attributes of a periodic thread

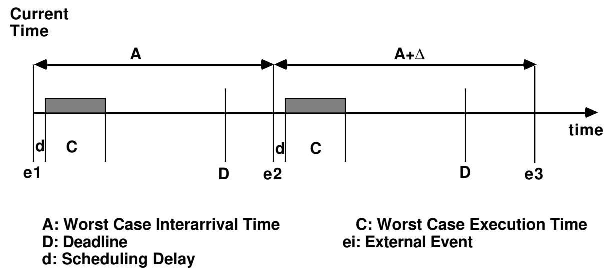


Figure 2: Timing attributes of an aperiodic thread

2 Real-Time Mach

The objective of Real-Time Mach (RT-Mach) is to develop a real-time version of Mach which can support a predictable real-time computing environment together with a real-time toolset. Because of the high portability of Mach, RT-Mach should be able to provide a common real-time computing environment in various machine architectures including single board computer-based targets.

In this section, we will describe the new features of RT-Mach. The current version of RT-Mach supports a real-time thread model, integrated real-time thread scheduler, policy/mechanism separation in the scheduler, real-time synchronization mechanisms, and memory resident objects.

2.1 RT-Thread Model

The objective of the RT-thread model is to support a predictable real-time scheduler and provide a uniform system interface to both real-time and non-real-time threads. Unlike the traditional real-time systems which often use a cyclic executive model, the RT-Mach supports an integrated time-driven scheduler [26] based on a rate monotonic scheduling paradigm [12, 13].

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 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. Figure 1 depicts the timing attributes of a hard periodic real-time 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. Figure 2 depicts the timing attributes of a hard aperiodic real-time thread[2].

2.2 RT-Thread Creation and Termination

A thread can be created, within a task, by using the *rt_thread_create* primitive. As we described in the model, it can be a periodic or aperiodic thread depending on its timing attributes. The timing attributes are specified in the corresponding time descriptor, and the user and kernel stack regions are also given by the stack descriptor. If a creation is successful, a unique thread id will be returned. A thread can be terminated by calling *rt_thread_exit* primitive. If a thread is a periodic thread, a new instantiation of the thread will be scheduled for the next start time and a new thread id will be assigned. The

[2] When a hard real-time thread is aperiodic, we call it a *sporadic* thread where consecutive requests of the task initiation are kept at least Q units of time apart [15].

rt_thread_kill primitive terminates the specified thread while the *rt_thread_wait* primitive blocks the caller thread until the target thread terminates. The *rt_thread_self* primitive returns the thread id of the caller. The *rt_thread_set_attribute* and *rt_thread_get_attribute* primitive are used to assign or get the value of the attribute respectively. The brief description of the thread attribute is shown in below.

```
kval_t = rt_thread_create( parent, child_thread,
                           thread_attr, entry_point, arg )
kval_t = rt_thread_exit( )
kval_t = rt_thread_kill( thread )
kval_t = rt_thread_wait( thread )
thread_t = rt_thread_self( )
kval_t = rt_thread_set_attribute( thread, thread_attr )
kval_t = rt_thread_get_attribute( thread, thread_attr )
```

```
typedef stuct time_desc {
    int rt_type;                      /* periodic or aperiodic thread */
    union {
        struct rt_Periodic {
            time_value_t rt_start;    /* start time */
            time_value_t rt_period;   /* period or response time info */
            time_value_t rt_offset;   /* phase offset */
        } rt_periodic;
        struct rt_Aperiodic{
            time_value_t rt_wcia;     /* worst case interarrival time */
        } rt_aperiodic;
    } rt_attribute;
    time_value_t rt_wcec;             /* worst case exec time */
    time_value_t rt_deadline;         /* deadline */
    time_value_t rt_abort;            /* abort time */
    int rt_value;                     /* semantic_value */
        ...
} time_desc_t

typedef struct stack_desc {
    vm_address_t rt_stack_addr;
    vm_size_t rt_stack_size;
        ...
} stack_desc_t

typedef struct thread_attribute {
    time_desc_t time_desc;
    stack_desc_t stack_desc;
        ...
} thread_attr_t;
```

2.3 RT-Thread Synchronization

Synchronization among threads is necessary since all threads within a task share the task's resources. The synchronization mechanism in RT-Mach is based on mutual exclusion using a lock variable. A thread can allocate, deallocate, and initialize a lock 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 created 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. RT-Mach uses the earliest deadline first (or highest priority first) policy as a queueing policy in both the *rt_mutex* and *rt_condition* primitives. A caller can also control its priority inheritance policy by setting the proper mutex's or condition variable's attribute.

```
kval_t = rt_mutex_allocate( lock, lock_attr )
kval_t = rt_mutex_deallocate( lock )
kval_t = rt_mutex_lock( lock, timeout )
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 )
```

Unlike the ordinary mutual exclusion mechanism, the *rt_mutex_lock* and *rt_mutex_unlock* pair provide a priority inheritance mechanism in order to avoid an unbounded *priority inversion* problem. Priority inversion occurs when a high priority task must wait indefinitely for a lower priority task to execute.

Suppose that a low priority thread τ_L is in the critical region. While thread τ_L is executing, a high priority thread τ_H attempts to enter the critical region by executed the *rt_mutex_lock* primitive. Since τ_L is in the critical region, τ_H must wait for τ_L to exit. Now suppose that other threads $\tau_{M_1} \cdots \tau_{M_k}$ become active. These threads can begin their computation and will preempt thread τ_L, thus we cannot bound the worst case blocking time of τ_H.

In order to bound the worst case blocking time of threads, our group has developed priority inheritance protocols including *Priority Ceiling Protocol* [18]. In this example, once τ_H executes *rt_mutex_lock*, then τ_L will inherit the high priority from τ_H. In this way, the highest priority thread's worst case blocking time is bounded by the size of critical region (See Section 2.5).

2.4 RT-Thread Scheduling

In real-time operating systems, thread scheduling plays an important role in managing the system resources in a timely fashion. However, traditional operating systems do not provide us a flexible and a adaptable scheduling management. We have developed a novel scheduling model, Integrated Time-Driven Scheduler(ITDS) for the ARTS kernel[24], and have extended the model for RT-Mach. This section describes the Mach scheduling mechanism and an extended ITDS model.

2.4.1 Mach Scheduling Mechanism

Mach provides a flexible processor allocation facility. The facility uses two objects: *processor* and *processor set*. A processor object represents a physical processor and a processor set object corresponds to a set of processors. A thread belongs to a processor set and similarly a processor belongs to a processor set. A special processor set called a *default processor set* exists. Before a new processor set is created, all processors belong to a default processor set.

The most important data structure to manage scheduling of thread is the *run queue*. All processor sets have their own respective run queues. When a thread becomes runnable, it is enqueued into the run queue of the processor set to which the thread belongs. Also, when the current running thread in a processor is blocked or preempted, the new thread is chosen from the processor set where the blocked thread belongs. Because threads do not migrate between processor sets, we can choose a suitable scheduling policy for each processor set. However, the current Mach scheduler has only *round-robin* and *fixed priority* policies and manages 32-levels of thread priorities. Mach's fixed priority policy preempts the running thread if there are runnable threads with same priority and its quantum is expired. In real-time computing, such preemption decreases schedulability and we need other scheduling policies, for example, rate monotonic policy, and various aperiodic servers[19].

2.4.2 ITDS Scheduling in RT-Mach

The objective of the integrated time-driven scheduler is to provide predictability, flexibility, and modifiability for managing both *hard* and *soft* real-time activities. The ITDS scheduler allows the system designer to predict whether the given task set can meet its deadlines or not.

The ITDS scheduler adopted a *capacity preservation* scheme to cope with hard and soft types of real-time activities. By bandwidth preservation we mean that we divide the necessary processor cycles between the two types. We first analyze the necessary processor cycles by accumulating the total computation time for the hard periodic and sporadic activities. Then, we will assign the remaining schedulable amount of the unused processor cycles to the soft real-time tasks.

The ITDS scheduler was designed and implemented using an object model and layered structure. The scheduling policy used by the scheduler object is a self-contained object and separated from the mechanism layer used to actually switch threads within the kernel. Using the ITDS scheduler, we can assign five different polices for each processor set in RT-Mach (See Figure 3).

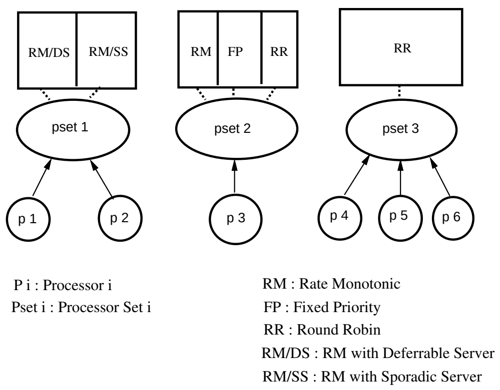


Figure 3: ITDS for RT-Mach

The low level mechanism layer is divided into two sublayers: a processor set management sublayer and a thread dispatching management layer. The processor set management sublayer manages context switching, preemptions of threads and assignment of processors. The thread dispatching management layer controls idle threads and aperiodic servers: the polling server, deferrable server, and sporadic server[19].

Each processor set can have a scheduling policy since each processor set maintains its own run queue and operations for controlling the run queue. Therefore, we can configure the system for the various real-time applications. For example, let us consider the multiprocessor with six processors, and three processor sets: $pset_1$, $pset_2$, and $pset_3$. $pset_1$ has two processors: p_1 and p_2, $pset_2$ has one processor: p_3, and $pset_3$ has a three processors: p_4, p_5 and p_6. $pset_1$ executes real-time applications using two processors using rate monotonic with deferrable server or rate monotonic with sporadic server. $pset_2$ executes real-time and non real-time applications, so the application program may change scheduling policies from rate monotonic, fixed priority, or round-robin. $pset_3$ executes non real-time applications using three processors by round-robin algorithm.

We can change scheduling policies by using the following primitives. The policy attribute is used to pass policy specific arguments such as a server's period, capacity of the server, etc. to the specified policy module in the system.

kval_t = **rt_get_sched_policy**(policy, policy_attr)
kval_t = **rt_set_sched_policy**(policy, policy_attr)

2.5 Memory Object Management

The kernel must also avoid unbounded delay while it manages memory objects for real-time threads. In general, Mach's resource allocation policy is based on *lazy* evaluation technique. For instance, if a thread allocates a region of memory, the system does not allocate the physical memory object unless the thread touches the region and cause a page fault.

In order to eliminate such unpredictable page fault handing delay, a real-time thread can "pin-down" any region of its parent task's virtual address space by using the following *vm_wire* primitive.

kval_t = **vm_wire**(task, start_addr, size, access_type)

2.6 Schedulability Analysis

In the RT-Thread Model, our goal is to provide a better interface to adopt the well-known schedulability analysis techniques. For instance, given a set of periodic, independent tasks in a single processor environment, with the rate monotonic scheduling algorithm the worst case schedulable bound is 69%[13], the average case is 88% [12], and the best case, where threads have harmonic periods, is up to 100% of the CPU utilization.

In the case of a more general task set where threads can synchronize via critical regions, we can also bound the synchronization (blocking) time for each task by using the priority ceiling protocol. Using these inheritance protocols, we can also check schedulable bound for *n* periodic threads as follows.

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

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

These techniques are integrated with our real-time toolset. However, providing end-to-end schedulability analysis for a given task set which communicates over a real-time network still remains as a future challenge[3].

3 Implementation

The current version of RT-Mach is being developed using a network of SUN, SONY workstations and single board target machines. We first experimented with our real-time thread model using a modified version of Release 2.5 Mach kernel which can support fixed priority thread scheduling, a cpu server (i.e., processor set), and a *vm_wire* call. We then moved to the current pure kernel based environment. 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, if necessary. The preemptability of the kernel was also improved significantly since many device drivers are no longer in the kernel.

A platform for RT-Mach has slightly different requirements in terms of its execution environment. For embedded real-time applications, we need to support not only various types of workstations, but also a wide variety of single-board or multiple-board based target machine environments. Booting the target machines also requires a different booting procedure via the system's backplane-bus or via a network. For instance, we are working on a VME-bus based target environment. For real-time communication, we are also supporting a FDDI network in addition to the IEEE 802.5 token ring network.

In this section, we will describe some implementation issues encountered in our first version of the pure kernel-based RT-Mach. We also discuss the capability of the real-time toolset and the current status of RT-Mach.

3.1 Implementation Issues

Our primary focus in the current implementation was to remove unbounded delay in the system and provide better preemptability among real-time threads. However, there are many places we encountered problems in managing system resources in Mach. In many cases, we can recognize differences in resource allocation policy between the time-sharing paradigm and the real-time computing paradigm. Major policy differences are

- "lazy" evaluation vs. "eager" evaluation policy
- FIFO ordering vs. deadline-driven ordering (starvation-free vs. no missed deadlines)
- unbounded delay vs. bounded delay

[3] Using the latest processor architecture such as various RISC chips, it becomes an interesting practical problem in determining the worst case execution time for sections of code (which must take into account cache management and pipelined architectures).

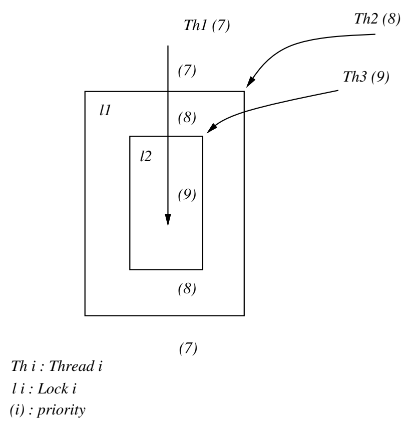


Figure 4: Priority Inheritance in Nested Lock

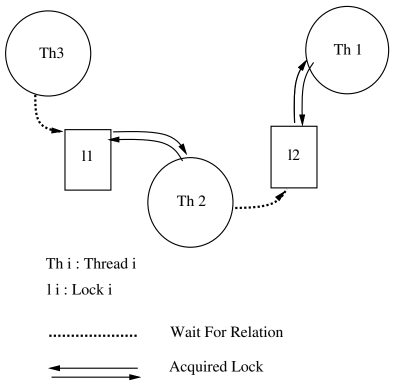


Figure 5: Cascaded Priority Inheritance

In many resource allocation cases, Mach takes advantage in deferring the actual allocation of resources until the requester needs it. *Copy-on-write* and *map-on-reference* techniques are good examples of the lazy evaluation scheme. On the other hand, this type of memory management policy often creates an unpredictable delay in getting the actual resources that the requester needs. For instance, if a thread needs to allocate memory, it calls *vm_alloc* routine, but the actual physical memory may not be allocated unless the thread actually touches that region and causes a page fault. However, if the thread is a real-time thread, it cannot afford to wait for unpredictable page fault service time. Rather, we would like to allocate memory resources in eager fashion. We can then estimate the worst case time for the allocation delay.

Mach uses many queues to manage various system resources such as ready queues, message queues, and free memory list queues. FIFO queuing is often used in these queues since the system can easily avoid the starvation among waiting threads. However, in a real-time environment, FIFO queueing often creates a priority inversion problem. If all of real-time threads can meet their deadlines, then there will be no starvation among these threads.

Similarly, in real-time synchronization, we do not treat waiting threads in FIFO order. For instance, when a real-time thread attempts to enter a critical region, it will be queued in its waiting queue in earliest deadline first (or the highest priority first) order. Then, when a thread exits from the critical region, the highest priority real-time thread will be chosen rather than the oldest waiting thread in the waiting queue.

In RT-Mach, we also use a basic priority inheritance protocol to avoid priority inversion problem in real-time synchronization. Let us describe three interesting cases where priority inheritance requires an additional mechanism.

In the first example, when a priority inversion occurs in a nested critical region, the propagation of the effective priority has to be performed carefully. Let us consider three threads: th_1, th_2, and th_3, and two locks: l_1 and l_2. We assume that th_1 has priority 7, th_2 has priority 8, and th_3 has priority 9. The larger value means higher priority. th_1 acquires l_1 and then l_2. th_3 then tries to acquire l_2, and th_2 tries to acquire l_1. The priority inheritance protocol makes th_1 inherit a priority of th_3, then the priority of th_1 becomes 9. After th_1 unlocks l_2, the priority of th_1 must become 8 because th_2's priority is 8 and th_2 waits for th_1 for unlocking l_1. When th_1 unlocks l_1, the priority of th_1 goes back to the base priority 7.

The second case is where the *waiting for* relation must be maintained among threads so that we can propagate the proper priority to a target thread. Let us consider three threads: th_1, th_2, and th_3, and two locks l_1 and l_2. th_3 has the highest priority, th_2 has middle priority, and the priority of th_1 is the lowest. th_1 acquires l_2. th_2 acquires l_1 and waits for unlocking l_2. When th_3 tries to acquired l_1, th_1 must inherit the priority of th_3. To manage this type of blocking case, the effect of priority inheritance is cascaded[4].

The third case is where the highest priority thread is timed out while waiting for a lock. For instance, in Figure 5, th_3 is inherits the priority of th_1. However, if th_1 is timed out while waiting for l_1, the priority of th_3 must be changed back to the priority of th_2.

The above problems can be solved by maintaining a relation between the lock variables and the threads. In RT-Mach, each thread maintains a pointer to lock variables, and the lock variable also keeps track of the nesting relation to the other locks and the holding thread. For example, in Figure 4, when

[4]The similar cascading problem and its solution was also described in [8]

th_1 acquires l_1, th_1 points to l_1. Next, when th_1 acquires l_2, l_2 points to l_1, and th_1 points to l_2. Then, if th_1 unlocks l_2, the priority of th_1 can degrade to the priority of $th2$ because l_2 knows l_1, and l_1 knows the priority of th_2.

We can solve the timeout problem mentioned above, the following way. From Figure 5, th_1 points to l_1, l_1 points to th_2, th_2 points to l_2, and l_2 points to th_3. So, th_1 can inherit the priority of th_3. If th_1 is timed out while waiting to acquire l_1, the priority of th_2 reverts to its base priority, and th_3 inherit that priority.

3.2 Real-Time Toolset

We have also developed a set of tools which we can use in conjunction with Real-Time Mach for predicting the behavior of the system and for runtime monitoring and debugging. The goal of the toolset is to incorporate a system-wide scheduling analysis which includes communication and synchronization among real-time threads. The toolset consists of *Scheduler 1-2-3* and *ARM*.

Scheduler 1-2-3 is a schedulability analyzer and is an X11-window based interactive tool for creating, manipulating, and analyzing real-time task sets. It employs methods ranging from closed form analysis to simulation to determine whether a feasible schedule exists for a given task set and what the schedulable bound is for that set.

ARM(Advanced Real-Time Monitor) is also an X11-window based tool designed to analyze and visualize the runtime behavior of the target nodes in real time. The ARM allows us to reach into a remote target and view the scheduling events which are extracted using event taps in RT-Mach.

3.3 Current Status

In the current version of RT-Mach, the RT-thread model and extended ITDS scheduler have been implemented. RT-Mach has also been integrated with the real-time tool set. In Figure 6, we show the snapshot of ARM with three periodic threads, and ten aperiodic threads. ARM is useful for monitoring the occurrences of preemption and the order in which threads are executed. Timing bugs can also be deleted easily using ARM.

Figure 7 demonstrates the RT-thread model. In the gmol demo, seven periodic threads are created and every thread represents an atom of a molecule. The threads, while executing, cause the molecule to rotate about an axis passing through the atom at the center. If scheduled correctly, the molecule maintains its integrity, otherwise the atoms move in a random fashion. Gmol visually demonstrates lack of schedulability, when the molecule's rotation is random. If we use a proper scheduling policy, we can ensure schedulability with high CPU utilization and each molecule rotates in a completely synchronized way (the left figure). However, if the scheduling policy is inappropriate, some deadlines are missed, so the synchronized rotation of the molecules is violated, and each molecule rotates in chaotic manner (the right figure). Gmol further demonstrates the importance of the RT-thread model in preserving the schedulability of RT-thread. The traditional system offers the *delay* primitive for representing time description. However, the calculation of delay time and the execution of delay primitives may not be atomic giving rise to invalid time values, resulting in non synchronized rotation of the molecule.

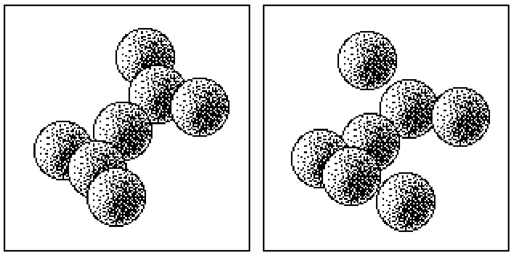


Figure 7: The Snapshot of Gmol

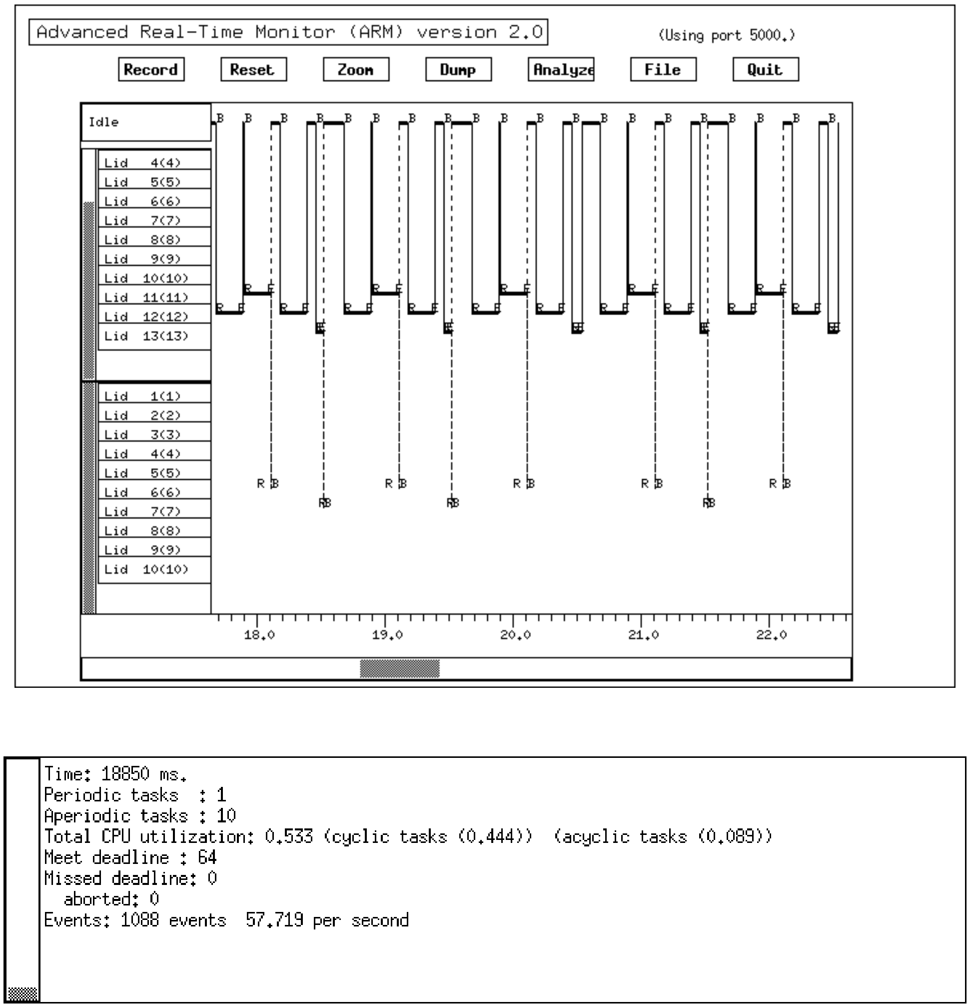
Null Argument Trap	0.03 ms
Null Argument MIG	0.30 ms
Context Switch	0.26 ms
vm_alloc 1KB (no wiring)	0.59 ms
vm_alloc 1KB (wiring)	2.67 ms
Port Allocate/Deallocate	1.2 ms
RT-Thread Creation/Termination	4.258 ms
RT-Lock and Unlock	0.146 ms

Table 1: The Basic Performance of RT-Mach

Table 1 summarizes the basic performance of the current version of RT-Mach.

All measurements are performed on a Sun3/60 workstation with 12 Mbytes by repeating the target function more than 10,000 times. The trap interface to the kernel is about 10 times faster than MIG [5]. The context switch time between threads is acceptable in our application, and we can assume this number as the worst case since all resources are wiring down for these threads. Allocation of memory is normally done without wiring, thus the overhead is relatively small. However, additional wiring cost is not reduce to negligible yet, since we simply reused the original wiring facility in Mach. The creation and termination cost are mainly due to allocation and deallocation of system resources which belongs to a thread. The total cost should be reduced further by re-

[5]Mach uses a MIG(Mach Interface Generator) [10] to call kernel primitives in a object-oriented fashion and generates stubs for user programs.



This figure shows an example of the history execution diagram. The top six boxes indicate the action menus. The top half of threads correspond to the periodic threads and the bottom half correspond to the aperiodic threads. Character 'R' shown in the execution history diagram indicates a periodic thread which becomes runnable and 'E' indicates that it terminates with its deadline being met; 'A' or 'C' indicate that the thread is aborted or canceled due to a missed deadline. 'B' indicates that the thread is blocked waiting for some event, or for preemption. The bottom window shows the various statistical information.

Figure 6: An Example of the ARM Snapshot

placing the allocation and deallocation policies. The current version of the *rt_thread_create*, *rt_thread_exit*, *rt_mutex_lock*, and *rt_mutex_unlock* primitives are implemented using a trap mechanism, rather than MIG.

4 Related Work

The pure kernel-based approach is gaining popularity and several pure (or micro) kernel-based operating systems have been developed for the distributed computing environment [3, 16, 17]. Advantages of using a pure kernel instead of a standard monolithic kernel is that the preemptability of the kernel will be inherently better, the size of the kernel becomes much smaller, and modification of the kernel will be easier. However, only a few micro-kernels were designed for supporting distributed real-time applications.

In many commercially available real-time operating systems and executives, a fixed priority-based preemptive scheduling policy has been used. Emphasis was placed on fast interrupt latency, fast context switching, and small kernel size [6]. Although these factors are important properties for real-time operating systems, users were often forced to create an *ad hoc* scheduling module for each particular application. Further more, under a transient overload, users may loose control over which tasks should complete their computations and which should be aborted or canceled. It is also difficult to remove priority inversion problems in the kernel and bound the worst case blocking time for threads.

The proposed real-time thread model is different from many other thread models. In particular, our model

- distinguishes between real-time and non real-time threads,
- assumes explicit timing constraints for each real-time thread, and
- provides a priority inheritance protocol to avoid unbounded priority inversion.

The POSIX-Thread proposal[11] is very similar to Mach's C-Thread package[5] and it also does not distinguish between real-time threads and non-real-time threads. This poses a problem of identifying the type of threads that can "pinned down" its memory objects. However, it can dynamically select the thread scheduling policy and a thread also contains the thread attributes such as "inherit priority", "scheduling priority", "scheduling policy", and "minimum stack size". Thus, adding the timing attributes would be very simple.

The Ultrix-Thread model[4] does not address real-time thread issues, however, the designer intended to create much lighter threads by leaving the context information of thread at the process level as much as possible. Thus, creation of a new thread can be done by specifying thread's stack page and guard page address: **tfork**(stack_ptr, guard_ptr).

The Topaz-Thread model[14]provides a clean thread interface library at the Modula-2+ language level, however, it does not address real-time thread issues.

5 Summary

The objective of Real-Time Mach is to develop a real-time version of Mach which can support a predictable real-time computing environment together with a real-time toolset. In particular, the kernel should allow a system designer to predict the schedulability of *hard* and *soft* real-time tasks which communicate over a real-time network.

In this paper, we described a real-time thread model, real-time synchronization, integrated time-driven scheduler, and memory resident objects for Real-Time Mach. We also discussed the implementation issues, real-time toolset, and the current status of the system.

We are still improving the system capability in order to provide a system-wide schedulability analysis in Real-Time Mach. In particular, we are working on predictable real-time communication support, priority inversion problems in Mach IPC, and multi-board based multiprocessor targets.

6 Acknowledgments

We would like to thank the members of the ART Project and the Mach group for their valuable comments and inputs to the development of Real-Time Mach.

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] David L. Black, "Scheduling support for concurrency and parallelism in the Mach operating system", *IEEE Computer,* Vol.23, No.5, 1990

[3] 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.

[4] D. S Conde, F. S. Hsu, and U. Sinkewicz, "Ultrix threads", In *Proceedings of Summer Usenix Conference,* June, 1989.

[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] B. Furht, J. Parker, and D. Grostick, "Performance of $REAL/IX^{TM}$ - Fully Preemptive Real Time UNIX", *Operating System Review,* Vol.23, No.4, April, 1989

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

[8] Mark Heuser, "An implementation of real-time thread synchronization", *In Proceedings of Usenix Summer Conference,* June, 1990.

[9] P. Hood and V. Grover, "Designing real time systems in ADA", Tech Report 1123-1, SofTech, Inc., January, 1986.

[10] M.B. Jones, and R.F. Rashid, "Mach and Matchmaker: Kernel and language support for object-oriented distributed system", *In proceedings of the first conference of OOPSLA,* September, 1986

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

[12] J. P. Lehoczky, L. Sha, and Y. Ding, "The rate-monotonic scheduling algorithm: Exact characterization and average case behavior", Department of Statistic, Carnegie Mellon University, 1987.

[13] 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.

[14] P. McJones and Swart P, "Evolving the unix system interface to support multithreaded programs" , Technical report, Tech Report 21, Part I, DEC SRC, September, 1987.

[15] A. K. Mok, "Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment", *PhD thesis*, Massachusetts Institute of Technology, May 1983.

[16] 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

[17] 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

[18] 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

[19] B. Sprunt, L. sha and J. P.Lehoczky, "Aperiodic Task Scheduling for Hard-Real-Time Systems", *The Journal of Real-TIme Systems*, Vol.1, No.1, 1989.

[20] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems", *IEEE Computer*, Vol.21, No.10, October, 1988.

[21] H. Tokuda and M. Kotera, "A real-time tool set for the ARTS kernel", Proceedings of 9th IEEE Real-Time Systems Symposium, December, 1988.

[22] H. Tokuda and M. Kotera, "Scheduler1-2-3: An interactive schedulability analyzer for real-time systems", *In Proceedings of Compsac88,* October 1988.

[23] H. Tokuda, M. Kotera, and C. W. Mercer, "A real-time monitor for a distributed real-time operating system", *In Proceedings of ACM SIGOPS and SIGPLAN workshop on parallel and distributed debugging*, May, 1988.

[24] H. Tokuda, M. Kotera, and C. W. Mercer, "An integrated time-driven scheduler for the ARTS kernel", *In Proceedings of 8th IEEE Phoenix Conference on Computers and Communications*, March, 1989.

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

[26] 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.