Real-Time Mach

Hiroshi Arakawa, Se’rgio Campos, Stephen Chou, Takuro Kitayama, Tatsuo Nakajima, Stefan Savage, Hide Tokuda

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pa. 15213

What is Real-Time?

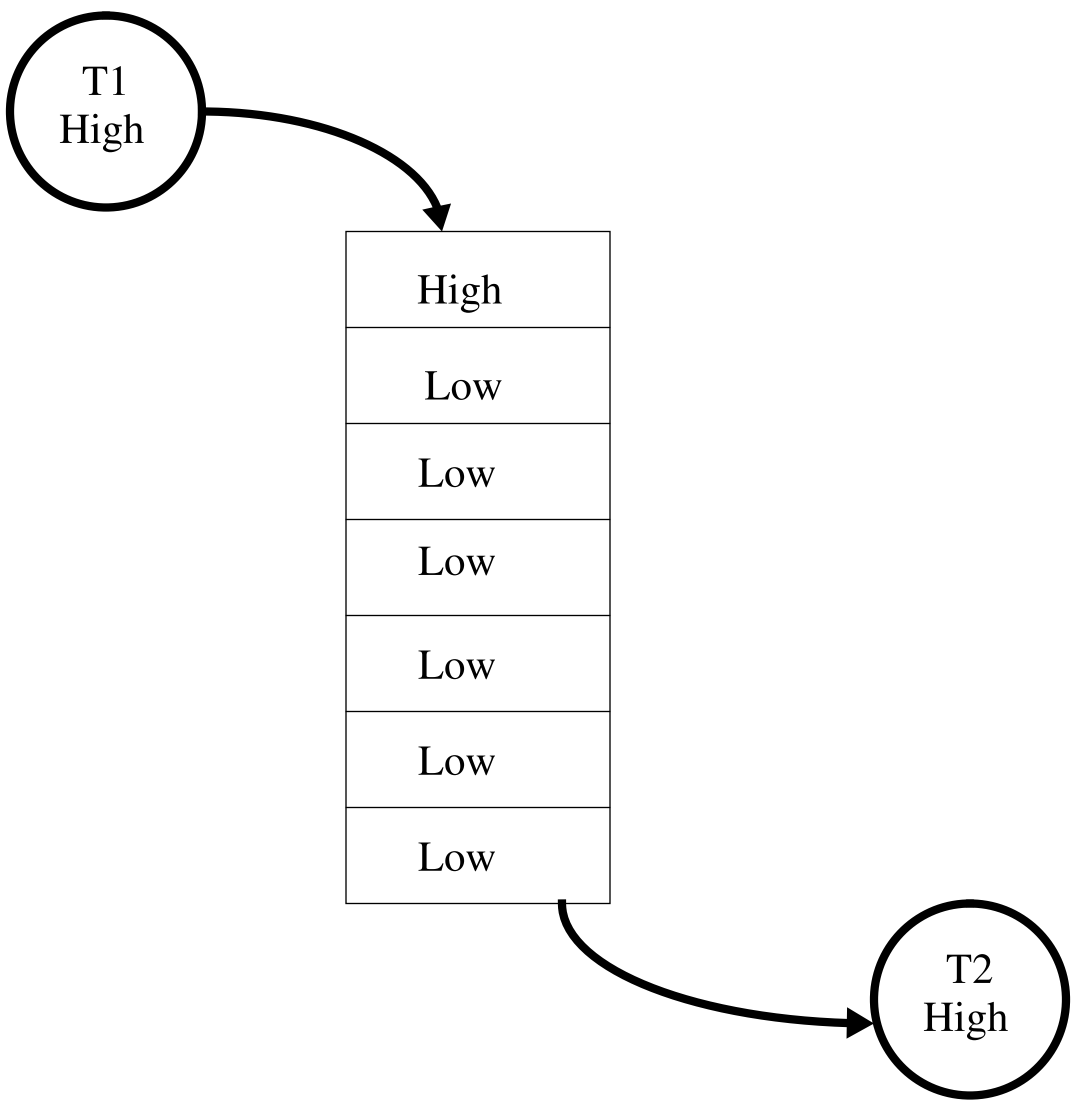
- Speed alone is insufficient!
- Ideally, the correctness of real-time programs should depend on temporal, as well as logical, validity.
- OS services should be provided within a bounded time frame. Priority inversion should be minimized.
- Schedulability of real-time tasks should be maximized (TDM style scheduling is not very interesting)

Real-T ime Liabilities for Mach

- Lazy evaluation philosophy
- Time is not a first-class attribute.
- No priority based synchonization
- Time-sharing scheduler
- FIFO queueing of IPC
- Non-preemptable kernel

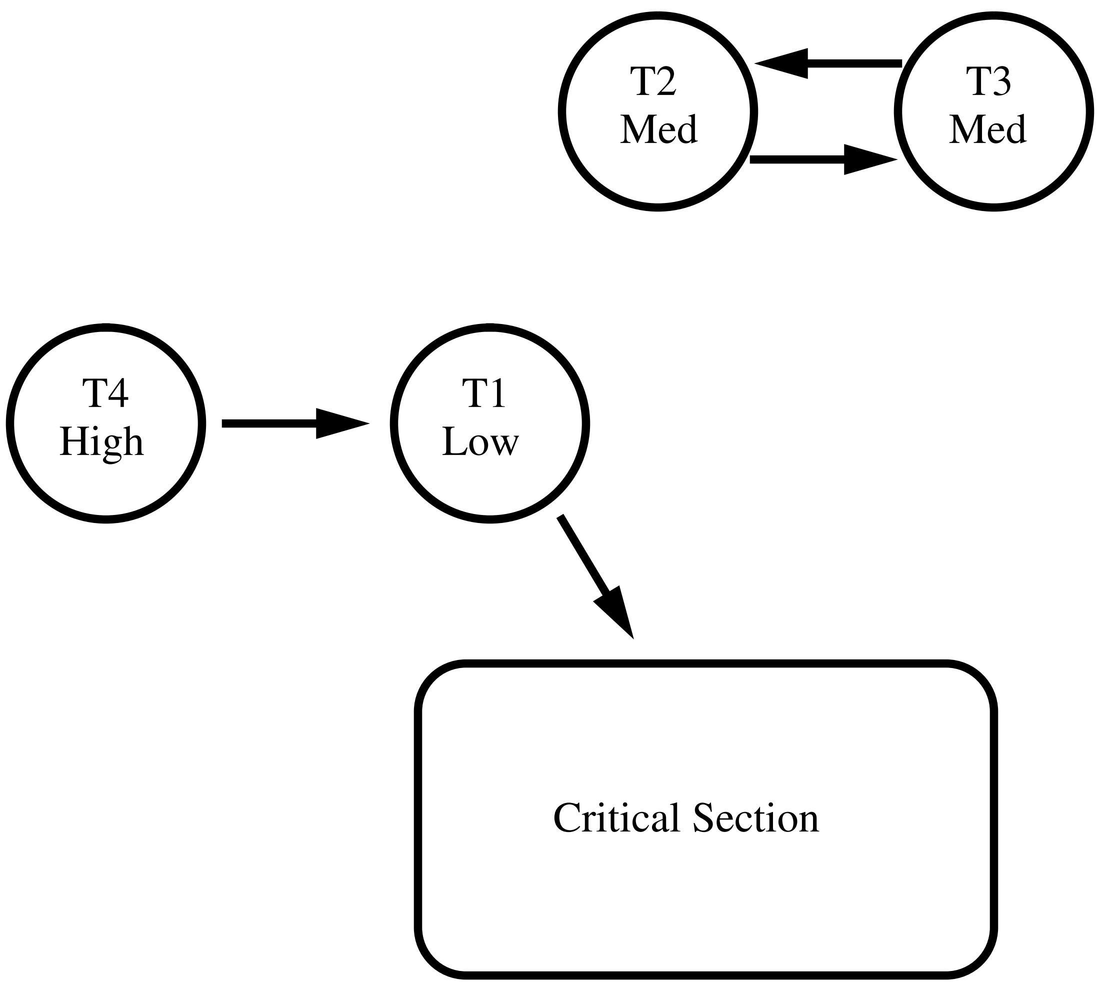
Priority Inversion Problem

- FIFO queuing of messages can result in unpredictable service times.

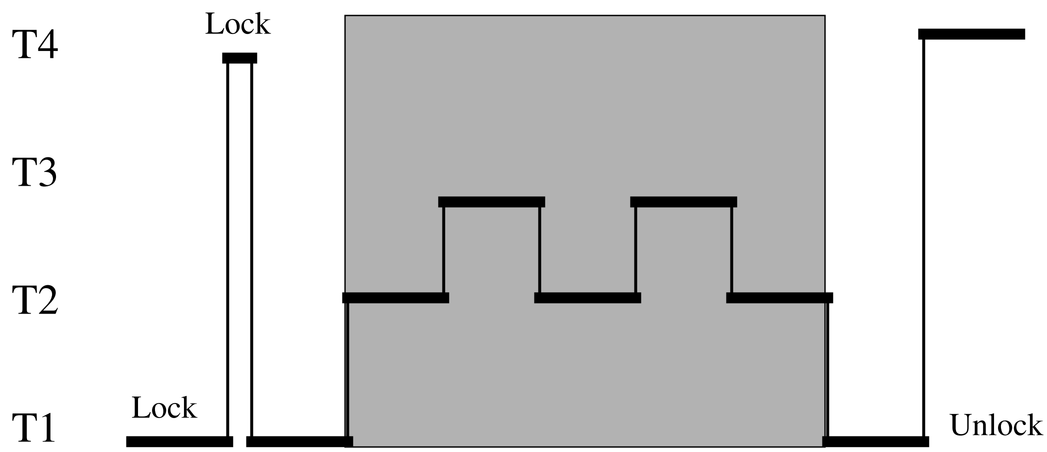


Priority Inversion Problem

- Conventional synchronization can lead to unbounded service times.



Priority Inversion Problem

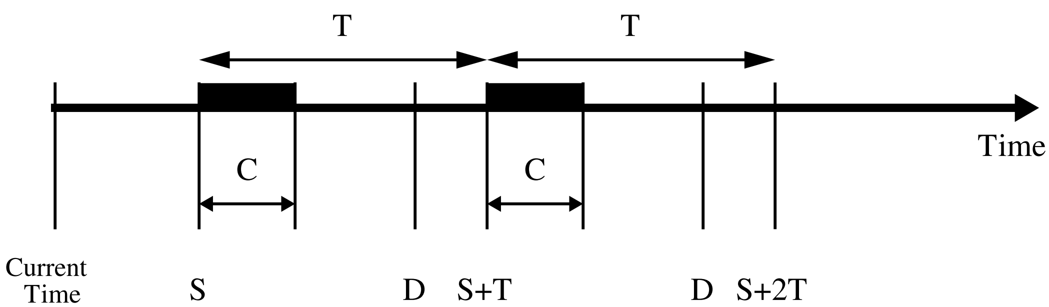


Real-T ime Extensions

- Real-time threads
- Selectable scheduling policies
- Priority based synchronization
- Time violation exceptions
- Prioritized queuing of IPC

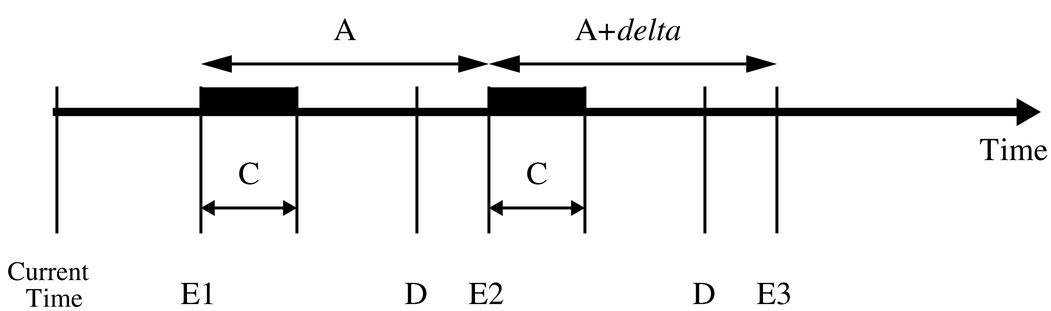
Real-Time Thread Abstractions

Periodic Thread



C: Worst Case Execution Time
S: Start Time
D: Deadline
T: Period

Aperiodic Thread



C: Worst Case Execution Time
Ex: External Event
D: Deadline
A: Worst Case Interarrival Time

Thread Interface

rt_thread_create (parent_task, child_thread, thread_attr);

struct **thread_attr_data_t** {
    long priority;
    long period;
    long deadline;
    long wcet;
    void (*entrypoint)();
    void (*exitpoint)();
    vm_offset_t stack_addr;
    long stack_size;
...
}

thread_get_state (target_thread, flavor, new_state, new_stateCnt);
thread_set_state (target_thread, flavor, new_state, new_stateCnt);

where flavor is **RT_THREAD_ATTR_STATE** and new_state holds a **thread_attr_data_t** structure.

Threads can be terminated using the traditional **thread_terminate** and **task_terminate** primitives.

Periodic Thread Implementation

rt_thread_create ()

- Creates thread (with thread_create)
- Binds thread_attr to thread
- Calculate start time and deadline
- Point PC to entrypoint
- Adjust stack so rt_thread_exit() is called on return
- Enter thread into wakeup and deadline queue

rt_thread_exit ()

- Reset start and deadline times
- Reset PC
- Reset stack
- Put thread into wakeup and deadline queues
- Block thread

The wakeup queue is checked at timer interrupt time and if a wakeup time has arrived the thread is enqueued in a runq.

Scheduling policies

Scheduling policies are selected per processor set:

- Mach T imesharing
- Fixed Priority
- Rate Monotonic
- Deadline Monotonic
- Earliest Deadline First

Aperiodic Servers are a method of capacity preservation for improving the response time of aperiodic threads:

- Deferable Server
- Sporadic Server

Schedulability Analysis

- Rate Monotonic

$$\sum_{j=1}^{n} \frac{C_j}{T_j} \leq n \times (2^{1/n} - 1)$$

- Rate Monotonic with Priority Ceiling Critical Section

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

Ci: The total compuation time fo Thread i

Ti: the Periodc of Thread i

Bi: The worst case blocking time of Thread i

Scheduler Interface

rt_sched_get_policy (pset);

rt_sched_set_policy (pset, policy);

rt_aps_create (pset, aps_attr);

rt_aps_destroy (aps);

rt_aps_bind (aps, thread);

rt_aps_unbind (thread);

Scheduler modifications

Indirected to per-policy functions (eg **(*rt_thread_select)()**):

- **thread_select()**
- **thread_block()**
- **thread_run()**
- **thread_setrun()**
- **rem_runq()**
- **choose_thread()**
- **choose_pset_thread()**
- **csw_needed()** [no longer a macro]

Added:

- **clock_sched()** : to be run every tick... replaces **thread_update_quantum()**
- **ast_sched()** : to check if a csw is needed in AST (only used for timesharing policy)
- Optional logging code for examining scheduler state changes with ARM tool.

Timeouts use time_value_t instead of ticks. This prevents 32bit timeout problem... somewhat more expensive.

Assorted changes to thread deletion functions...

Synchronization Model

- Preemptability
  - Non Preemptable
  - Preemptable
  - Restartable
- Queuing
  - FIFO
  - Priority Based
- Priority Inheritance function

Synchronization Policies

- KM: Kernelized Monitor

  Non preemptable

- BP: Basic Priority

  Preemptable, Priority queuing

- BPI: Basic Priority Inheritance

  Preemptable, Priority queuing, Priority inheritance

- PC: Prority Ceiling

  Preemptable, Priority queuing, Priority inheritance with Priority Ceiling

- RCS: Restartable Critical Section

  Restartable, Priority Queuing

Synchronization Interface

rt_mutex_allocate (lock, lock_attr);

rt_mutex_deallocate (lock);

rt_mutex_lock (lock);

rt_mutex_unlock (lock);

rt_mutex_try_lock (lock);

```
struct lock_attr_data_t {
    long policy;
    long priority_ceiling;
    time_value_t timeout;
    void *saved_state;
    ...

}
```

There is also a similar interface for condition and event variables.

Synchronization Implementation

- Is shrouded in mystery ...
- Is changing...

Basically, priority inheritance is implemented by maintinaing a priority ordered queue of threads which are requesting requesting a lock. If a requested lock is owned and the requesting thread is at the head of this queue then the thread currently holding the lock saves it's current priority and assumes the priority of the requesting thread. When the lock is reliquished the thread resumes it's old priority and the highest priority thread is allowed to make forward progress.

Nested locks are arbitrated by maining a doubly linked list of lock descriptors.

Time V iolation Implementation

Threads which wish to be notified asynchronously of deadline violations implicitly create a shadow thread to take the exception.

Deadlines are entered into deadline queue
(the current implementation incorporates deadline and wakeup events in the same queue)

At timer interrupt time the deadline queue is checked

If a deadline has been missed then an exception is dispatched to the exception port of the specified thread.

At that point the shadow thread can choose to ingore the exception, kill the thread, dynamically change thread attributes (raise the period, etc), or some other user specified action

On machines with programmable countdown timers (such as the i386) it is possible to set deadlines to below the regular timer tick granularity .

The timer can be programmed to interrupt at the next deadline and then logic in the interrupt handler can make it still appear to behave as a 10ms or 1ms interval timer .

Why T rap? (or at least why we manage nasty stuff like user stacks in the kernel)

- Speed!

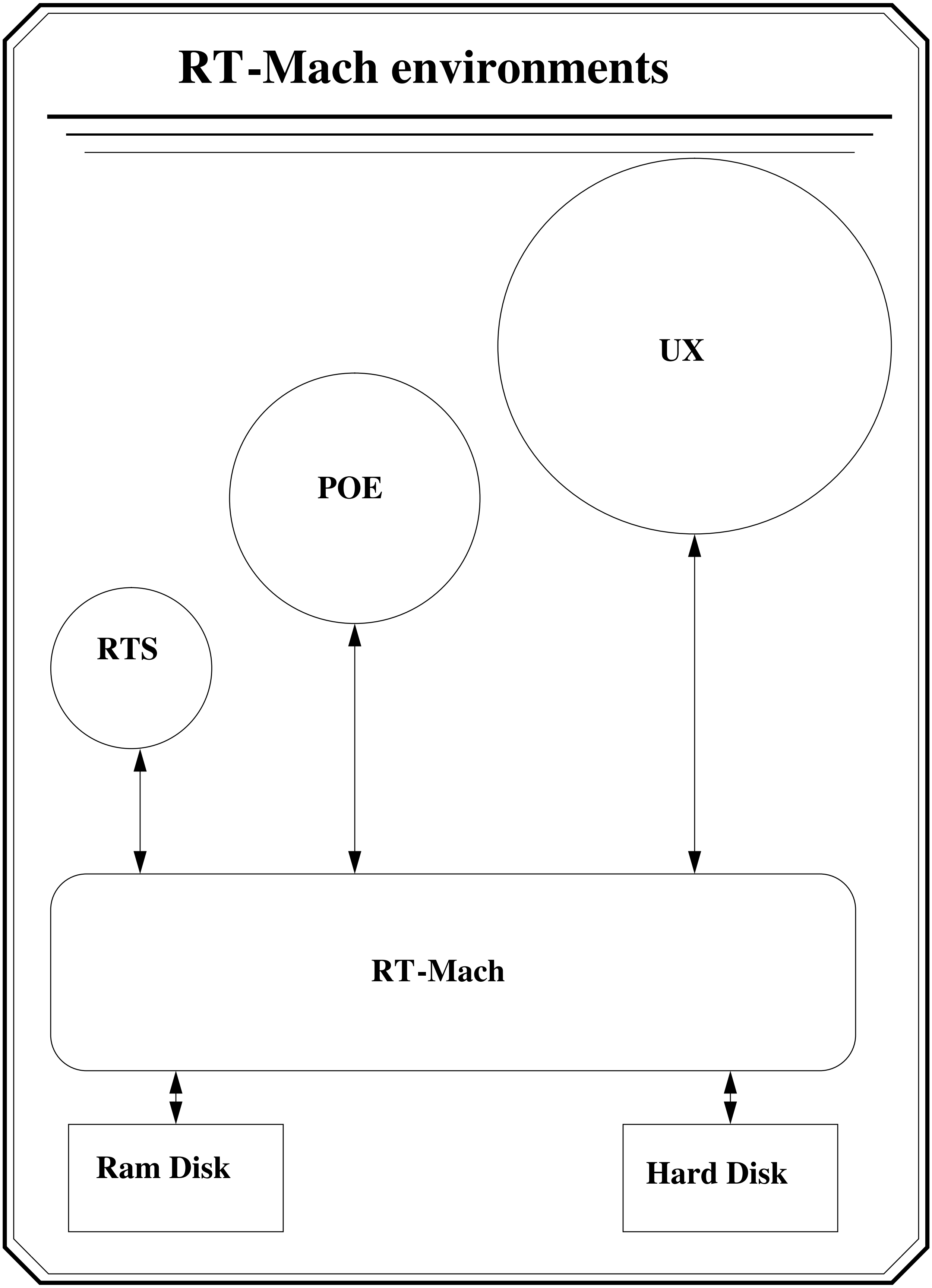
Using the “traditional” Mach approach to create a rel-time thread requires 6 traps (mostly mach_msg as well):

```
thread_create();
vm_allocate();
thread_get_state();
thread_set_state(); /* for registers */
thread_set_state(); /* for thread atributes */
thread_resume();
```

In contrast, rt_thread_create() does the same thing withonly a single trap:

```
rt_thread_create();
```

- Priority inversion in current IPC queueing


RT-Mach environments
UX
POE
RTS
RT-Mach
Ram Disk
Hard Disk

Current Status

- MK73 version of RT -Mach runs on i386 (and shortly mips) and supports multiple scheduling policies, periodic threads, and deadline exceptions

- MK69 and MK73 versions WILL run with the Unix Server

- MK47 version supports priority based synchronization

- Ram disk and VGA devices added

TO DO

- Prioritized IPC
- Remove default pager
- New paeriodic server implmentation.
- Automatically wire down all user memory
- Make synchronization code current with MK73
- Make sure those nasty thread_switch() and set_pri() functions REALL Y don't interact with our scheduler