Introduction to real-time programming concepts

W. Eric Norum

June 24, 2010

What is a real-time system?

- "One in which the timeliness of the results is as important as the value"
- "One which must provide its results within certain deadlines"
- These don't really nail it down very well.....
  - Is a payroll system "real-time"?

What is a real-time system?

- "I shall not today attempt further to define the kinds of material I understand to be embraced within that shorthand description and perhaps I could never succeed in intelligibly doing so.

  But I know it when I see it....."

  – Justice Potter Stewart,

  concurring opinion in Jacobellis v. Ohio 378 U.S. 184 (1964)

- Justice Stewart was talking about something different, but his approach might be reasonable.

Real-time system classifications

- **Hard real-time systems** - Failure to meet response-time constraints results in system failure.
- **Soft real-time systems** - Performance is degraded, but not destroyed by failure to meet response-time constraints.
- **Firm real-time systems** - Systems with hard deadlines where some low probability of missing a deadline can be tolerated.

What characterizes a real-time OS or executive?

- Deterministic response to external stimuli (interrupts)
- No long ‘tails’ in response histogram

And how's that done?

- Interrupts disabled for as little time as possible
- Interrupt handlers as short as possible
- Predictable scheduling
- In extreme cases can even do things like disabling memory caching – all to ensure deterministic response

Real-time Kernel Hierarchy

- Nano-kernel Task dispatching only
- Micro-kernel Task scheduling
- Kernel Intertask communication
- Executive
  - Adds I/O services, network services
- Operating System
  - Generalized user interface, file management, security

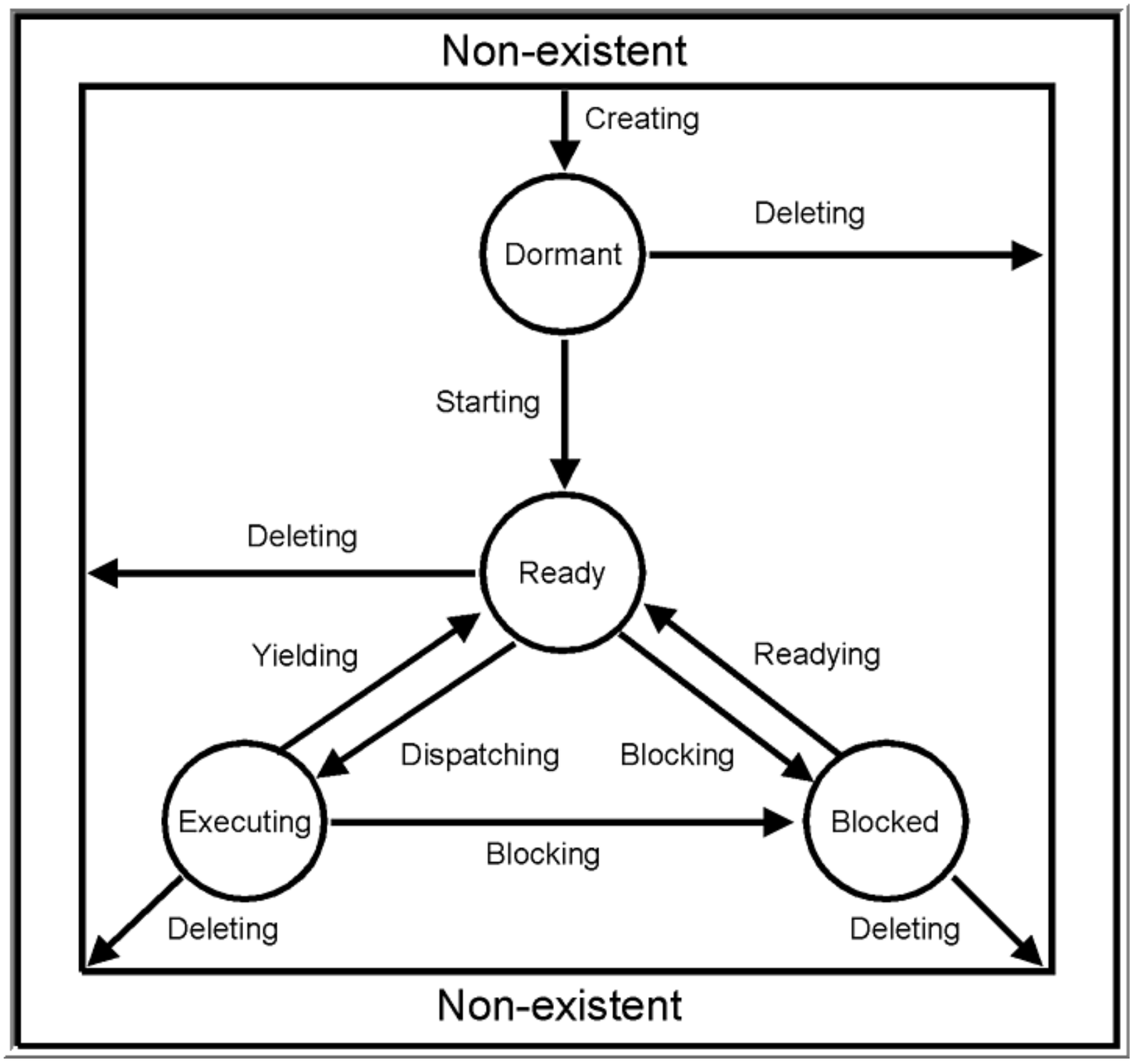
Scheduling

- Multi-processer vs. Multi-process vs. Multi-threaded
- Fair-share
- Strict Priority Based
- Rate Monotonic
- Round Robin

Task States

- Executing
- Ready
- Blocked (waiting for some condition)
- Dormant (created but not net started)
- Non-existent

Task States


Non-existent
Creating
Dormant
Deleting
Starting
Deleting
Ready
Yielding
Readying
Dispatching
Blocking
Executing
Blocked
Blocking
Deleting
Deleting
Non-existent

Task Synchronization

- Events
- Semaphores
- Message Queues

Events

- A task can wait for an event, or group of events, to arrive
- Events are sent to a particular task – the sender must know the task ID of the receiver
- Handy for simple synchronization case like waiting for an interrupt

Semaphores

- Control access to a common resource
- A counter that can be atomically incremented or decremented – but never goes below zero
  - Example – a restaurant

Semaphore options

- Counting/binary
- Queuing – priority or first-come, first served
- Recursive or not?

Mutual Exclusion

- Useful for
  - Piece of code that can be active in only one task at a time
  - Access to a resource (for example an I/O device) by just one thread at a time
- Implement as semaphore with initial count = 1
- Issues
  - Priority inversion
  - Deadlock

Priority Inversion

- Three tasks
  - A (high priority)
  - B (middle priority)
  - C (low priority)
- Resource shared by A and C
  - Protected by a Mutex Semaphore

Priority Inversion

- A and B have blocked so C is able to run
- C acquires the mutex
- B becomes runnable and so preempts C
- A becomes runnable and attempts to acquire the mutex – but blocks because C holds it
- B runs – so C has no opportunity to release mutex

Priority Inversion

- Even though B has nothing to do with the mutex it is preventing A from obtaining it
- Priority of A and B have effectively inverted
- A may be held blocked for too long
- c.f. Mars Pathfinder

Priority Inheritance

- One way of mitigating the problem
- When a higher-priority task attempts to obtain a mutex any task holding the mutex has its priority raised to that of the requesting task
  - The task holding the mutex ‘inherits’ a higher priority

Priority Inheritance

- Same system (A, B, C) but with inheritance
- This time when A attempts to obtain the mutex the priority of C is raised to that of A – so C is the highest-priority runnable task and runs until it releases the mutex
- The priority of C then reverts to its original value and A becomes the highest-priority runnable task and is able to obtain the mutex

Priority Inheritance

- With more tasks and priorities the possibilities become more complex (inherit priority, then inherit even higher priority, then back to first inherited priority and then back to original priority) but RTEMS takes care of all this for you

Deadlock

- Consider a system with two tasks A and B and two mutexes M_A and M_B
- A takes mutex M_A
- B takes mutex M_B
- A attempts to take mutex M_B – and blocks
- B attempts to take mutex M_A – DEADLOCK!
  - Neither A nor B can run

Deadlock Mitigation

- Don’t nest mutex requests – i.e. don’t request a mutex while holding another mutex
  - Not always easy to do but certainly the safest
  - Example is the EPICS ASYN package – a callback to user code is never made while a mutex is held

Deadlock Mitigation

- If you must nest requests always request them in the same order
  - The deadlock described earlier occurs only because A requests M_A and then M_B and B requests M_B and then M_A

Message Queues

- FIFO buffer
- Maximum message size and number set when queue is created
- Receiver can block waiting for message
- What happens when attempting to place a message on a queue that is full?
  - Block?
    - not a good idea in an interrupt handler
  - Return error?

Interrupts

- Specify function that is to be invoked when a particular interrupt occurs
- Function is passed a value which is often used to hold a pointer to a data structure containing a task ID or message queue ID or such like
- Must not call any routine that would block!

Simple I/O

- Memory mapped
- C ‘volatile’ keyword
- Useful to use function/macro

I/O

- Bad version

```
void pulsePin(uint16_t *ioreg)
{
    *ioreg = 1;
    *ioreg = 0;
}
```

I/O

- Better version

```
void pulsePin(volatile uint16_t *ioreg)
{
    *ioreg = 1;
    *ioreg = 0;
}
```

I/O

- Best version

```
void pulsePin(uint16_t *ioreg)
{
    out_be16(ioreg, 1);
    out_be16(ioreg, 0);
}
```

‘volatile’ and busy loops

- Another way that things can break

  int16_t *csr; while ((*csr & 0x80) == 0);

- Gets ‘optimised’ to:

  if ((*csr & 0x80) == 0) { while(1); }

- Adding ‘volatile’ fixes this (but it’s still not a good thing to do):
  - Should have upper bound on number of loops and return error if hardware never comes ready
  - Busy loops tie up the CPU and lock out lower priority tasks

‘volatile’ and shared memory

- Needed when variables are accessed by multiple threads (shared memory)
- Likely need a mutex as well
- ‘Obviously atomic’ operations aren’t necessarily so. Even simple assignments can be split up into multiple instructions on some architectures – what happens if you get an interrupt and task switch in the middle?

‘volatile’ and interrupt handlers

- Needed when variables are accessed by thread(s) and interrupt handler
- Need to think very carefully about where interrupt disable/enable directives are required
- The ‘atomic’ operation problem is even trickier here
- If at all possible, just have interrupt handler unblock a thread to do the real work

RTEMS

- Open-source, real-time executive
- ‘Super Core’ with multiple APIs
  - ‘Classic’
  - POSIX
  - ITRON (moribund)
- BSD network stack, NFS
- Multiprocessor capable (but **not** SMP -- yet)

RTEMS Objects

- RTEMS provides a set of predefined objects
- Tasks, message queues, semaphores, timers, memory regions
- Unique ID
- Arbitrary name

RTEMS Time

- ‘Tick’ based
- Defines granularity of interval and calendar operations
- Some external mechanism provides the periodic clock tick

RTEMS Task Manager

- Priorities 1 (highest) to 255 (lowest)
- Preemptible?
- Time slicing?
- Floating point?
- Stack size

RTEMS Task Directives

- rtems_task_create
- rtems_task_start
- rtems_task_wake_after
- rtems_task_wake_when

RTEMS Task Directives

- Some risky ones
  - rtems_task_suspend
  - rtems_task_resume
  - rtems_task_restart
  - rtems_task_delete

RTEMS Task Creation

- rtems_status_code rtems_task_create(

  rtems_name name,

  rtems_task_priority initial_priority,

  size_t stack_size,

  rtems_mode initial_mode,

  rtems_attribute attribute_set,

  rtems_id *id

);

  - Attributes (local/global, floating-point)
  - Modes (RTEMS_PREEMPT, RTEMS_TIMESLICE, etc.)

RTEMS Task Start

- rtems_status_code rtems_task_start(

  rtems_id id,

  rtems_task_entry entry_point,

  rtems_task_argument argument);

- ‘argument’ is commonly a pointer to a data structure holding per-task values

RTEMS Semaphore Creation

- rtems_status_code rtems_semaphore_create(

  rtems_name name,

  uint32_t count,

  rtems_attribute attribute_set,

  rtems_task_priority priority_ceiling,

  rtems_id *id);

- Attributes (FIFO/Priority, Binary/Counting/Simple, Inherit priority or not, implement priority ceiling or not, local/global)

RTEMS Semaphore Creation

- Attributes for a mutual-exclusion (Mutex) sempahore:

  RTEMS_PRIORITY |

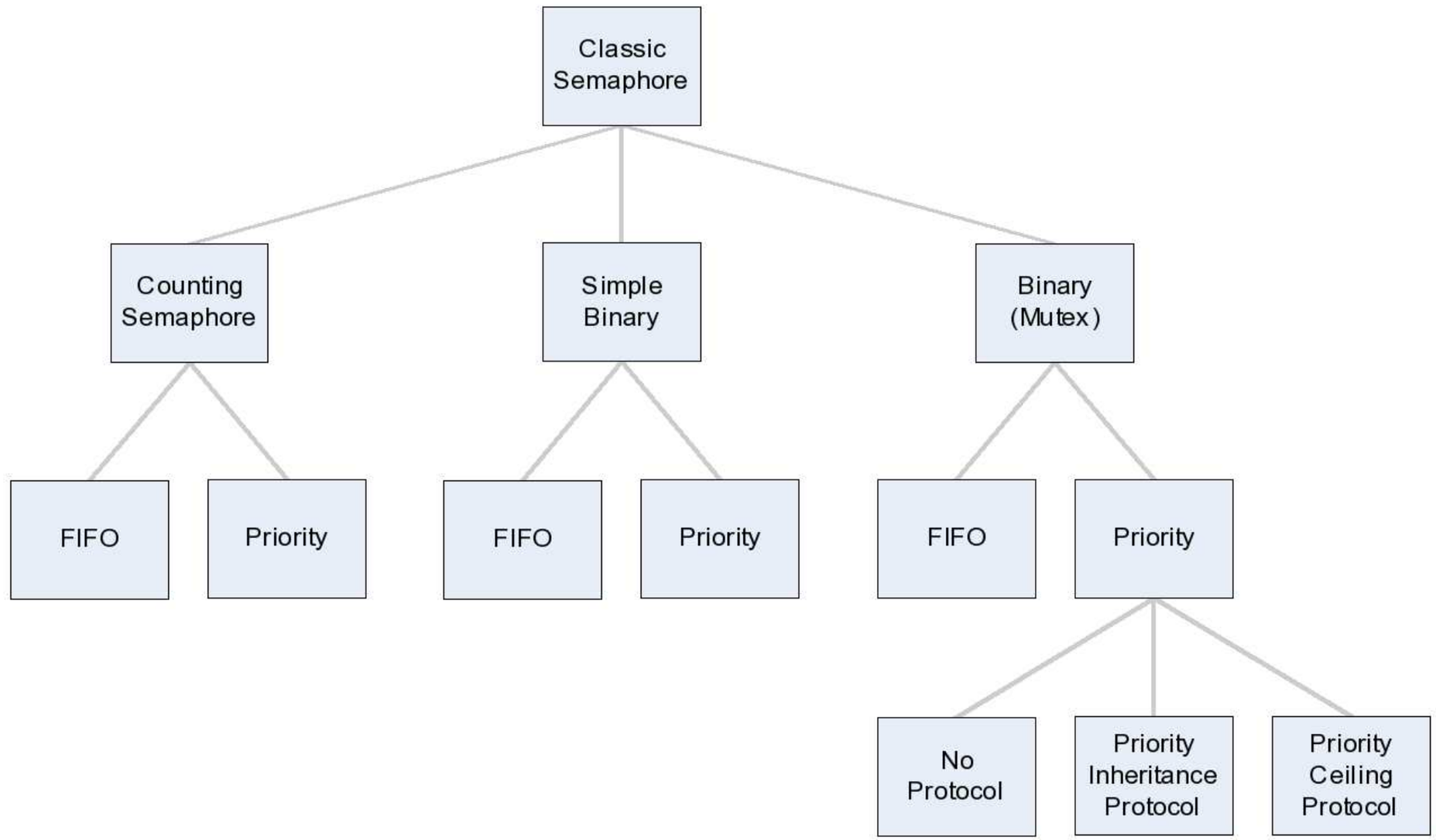
  RTEMS_BINARY_SEMAPHORE |

  RTEMS_INHERIT_PRIORITY |

  RTEMS_NO_PRIORITY_CEILING |

  RTEMS_LOCAL

RTEMS Semaphore Attributes



RTEMS Semaphore Acquisition

- rtems_status_code rtems_semaphore_obtain(

  rtems_id id,

  rtems_option option_set,

  rtems_interval timeout);

- option_set can be RTEMS_WAIT or RTEMS_NOWAIT
- timeout in clock ticks or RTEMS_NO_TIMEOUT to wait ‘forever’
- Release: rtems_status_code rtems_semaphore_release(rtems_id id);

RTEMS Events

- 32 possible events
- Sent to a particular task
- Task can wait (with optional timeout) for any or all events in a set to arrive

RTEMS Status Codes

- All RTEMS directives return an value with type rtems_status_code
- The routine rtems_status_text takes an rtems_status_code value as an argument and returns a pointer to a character string describing the value

RTEMS Status Codes

- Example

```
somefunc(...)
{
    rtems_status_code status;
    ......
    status = rtems_task_create(.....);
    if (status != RTEMS_SUCCESSFUL) {
        fprintf(stderr,“Can’t create task: %s\n”, rtems_status_text(status));
        ......
    }
```

RTEMS Object Names

- In the 'classic' API object names are arbitrary 32-bit values
- The macro rtems_build_name(c3,c2,c1,c0) packs four characters into a single 32-bit value:

rtems_task_create(rtems_build_name('t','s','k','A'), ...

RTEMS Web Site

- http://www.rtems.org
- Links to downloadable code
- Links to complete documentation