The ARM Instruction Set

Real-Time Operating Systems

Steven P. Smith
Mark McDermott

Real-Time System Organization

Controlled Subsystem
Control Subsystem
User Interface Subsystem

Application Interface
Man-Machine Interface

Classification of Real-Time Systems

Real-time systems are often divided into two groups

- “Hard” real-time systems are those with operational deadlines that must always be met in order for the system to be considered functioning properly.
  - Flight control systems
  - Aviation collision avoidance systems
  - Certain medical devices such as pacemakers
  - Safety-critical plant control systems

- “Soft” real-time systems are those with operational deadlines that, if missed occasionally, do not pose critical risks.
  - Streaming media decoders (set-top boxes)

The distinction between hard and soft real-time systems is driven almost exclusively by the question of the potential risks and costs associated with missed deadlines.

Hard and Soft Real-Time Systems

- The notion of “soft” real-time systems is imprecise at best.
  - A classic engineering trade-off: system cost and complexity versus the impact of a missed deadline.
  - Planes falling from the sky is a bad thing.
  - An extra millisecond or two of cooking time on your microwave popcorn is probably tolerable.

- The optimal point for a soft real-time system on the spectrum from perfect (hard) real-time performance to missing all deadlines is ultimately determined by the market.

- Failure to meet a deadline in a soft real-time system is typically associated with a potential economic penalty rather than the loss of human life.

Hard Real-Time Systems

- Hard real-time systems are typically found in highly regulated areas, and often have precisely defined performance requirements.
  - U.S. Federal Aviation Administration
  - U.S. Food and Drug Administration

- Hard real-time systems performance requirements are typically defined in terms of their maximum acceptable probability of system failure (i.e., a missed deadline).
  - Requirements in the development of the Airbus A-320 were to have a maximum probability of failure of $10^{-6}$ per flight hour.
  - High-quality navigation systems (GPS) exhibit a maximum probability of failure in the $10^{-4}$ to $10^{-3}$ range.

- In practice, hard real-time systems are often also fault tolerant systems.
Tasks in Real-Time Systems
- Real-time system can be characterized as a cooperating collection of tasks, each of which can be categorized as follows:
  - Hard real-time tasks
  - Soft real-time tasks
  - Not real-time tasks
- Tasks may be further classified:
  - Periodic tasks
  - Aperiodic tasks (sometimes yet imprecisely also called asynchronous)
  - Sporadic tasks

Real-Time Operating Systems
- The principal responsibility of a real-time operating system (RTOS) is to guarantee that each hard real-time task schedule is satisfied.
- Nearly as critical but less widely appreciated is the requirement that an RTOS exhibit predictable behavior.
  - RTOS functional and timing behavior must be as nearly deterministic as possible.
  - This aspect is critical in safety-critical hard real-time systems to facilitate accurate analysis of the system with respect to satisfying its requirements.
- Most modern RTOS implementations make extensive use of threads instead of processes for multitasking. More complex and/or higher performance systems will use both.

Key Elements of Real-Time Operating Systems
- Time management
  - Measurement resolution
  - Event timer resolution
  - Synchronization in distributed systems
- Interprocess and interprocessor communications
  - Safe data sharing
  - Hazard prevention
  - Deadlock avoidance
  - Bounded channel access and message transmission delays
- Task scheduling

Scheduling in a Real-Time Operating System
- The objective of the scheduler is to determine an ordering of the execution of tasks that is feasible.
  - One that enables a system’s scheduling requirements to be met.
- Scheduling algorithms are classified as either static or dynamic
- Deterministic system behavior and task execution times enables the use of standard scheduling algorithms.
- Scheduling algorithms (also sometimes referred to as policies) generally fall into one of the following categories:
  - Clock Driven Scheduling
  - Weighted Round Robin Scheduling
  - Priority Scheduling

Scheduling in a Real-Time Operating System
- For scheduling purposes, data must be gathered about the application and its constituent tasks:
  - Number of tasks
  - Resource Requirements of each task
  - Release Time for each task
  - Execution time for each task
  - Deadlines for each task, if applicable
- RTOS and platform-specific information must also be gathered, including:
  - Maximum context-switching time
  - Maximum interrupt service latency
  - Maximum communications delays
The ARM Instruction Set

RTOS Scheduling Approaches

- Clock Driven
  - All parameters about jobs (release time/execution time/deadline) known in advance.
  - Schedule can be computed offline or at some regular time instances.
  - Minimal runtime overhead.
  - Not suitable for many applications.
- Weighted Round Robin
  - Jobs scheduled in FIFO manner
  - Time quantum given to jobs is proportional to its weight
  - Example use: High speed switching network
  - QoS guaranteed.
  - Not suitable for precedence constrained jobs.
- Priority Scheduling
  - Processor never idle when there are ready tasks
  - Processor allocated to processes according to priorities
  - Priorities
    - Static: at design time
    - Dynamic: at runtime

Priority Scheduling of Periodic Tasks

- Priority scheduling generally assumes the ability to preempt executing tasks.
- Earliest Deadline First (EDF)
  - Process with earliest deadline given highest priority
- Least Slack Time First (LSF)
  - Slack = relative deadline − execution time left
- Rate Monotonic Scheduling (RMS)
  - For periodic tasks
    - Tasks priority inversely proportional to its period
    - A feasible schedule can always be created so long as CPU utilization required for tasks with deadlines remains at or below 69.3%.
- Priority inversion issues must be addressed in static priority schemes

Scheduling of Aperiodic Tasks

- Several techniques have been described in the literature for addressing aperiodic tasks in real-time systems.
  - Make aperiodic tasks “background” tasks that run only when the processor would otherwise be idle.
  - Create a periodic process with a fixed (and typically low) priority that executes aperiodic tasks as they arise.
  - Create a high priority task that executes whether an aperiodic task is currently active or not. This is often called “bandwidth preserving” scheduling and is well suited to systems that need to be highly responsive to external events (e.g., user input).
  - Create a periodic server task that inherits the priority of each aperiodic task as it arises.

Evaluating the Quality of a Real-Time Schedule

- Breakdown Utilization (BU)
  - The percentage of resource utilization below which the RTOS can guarantee that all deadlines will be met. Higher is better, of course.
- Normalized Mean Response Time (NMRT)
  - The ratio of the “best case” time interval a task becomes ready to execute and then terminates, and the actual CPU time consumed. Again, higher is better and the value is bounded by 1.0.
- Guaranteed ratio (GR)
  - For dynamic scheduling, the number of tasks whose deadlines can be guaranteed to be met versus the total number of tasks requesting execution.

Taxonomy of Real Time Operating Systems (RTOS)

- Small, fast, proprietary kernels (commercial, home grown)
  - Real-time extensions to commercial operating systems
    - Linux
    - Windows CE
  - Research Operating Systems
    - TinyOS
  - Part of language run-time environments
    - Java (embedded real-time Java)
- In general, there are three types of RTOS implementation:
  - Cyclic Executive
  - Monolithic kernel
  - Microkernel

RTOS Implementation: Cyclic Executive

Advantages
- Simple implementation, low overhead
- Very predictable
- Good for signal processing applications

Disadvantages
- Can’t handle sporadic events
- Everything must operate in lockstep
- Code must be scheduled manually
### The ARM Instruction Set

#### RTOS Organizations: Monolithic Kernel (e.g., Linux)
- **Advantages**
  - Commonality of RT Applications with non-real-time ones
  - Device Driver Base
  - Stability

- **Disadvantages**
  - Interrupt latency can be long
  - No support for handling priority inversion

#### RTOS Organizations: Microkernel
- **Advantages**
  - Very small footprint for kernel
  - Kernel provides scheduling/interrupt handling
  - Kernel plug-ins are available for COTS RTOS
  - TCP/IP stack, filesystem
  - KPIs are multithreaded

- **Disadvantages**
  - User generally has to write their own drivers and managers for non-COTS RTOS

### Some Example Real-Time Operating Systems
- **For tiny embedded systems**
  - PALOS
  - TinyOS
  - pSOS
- **For mid-size embedded systems**
  - eCos
  - µCOS-II
  - VxWorks
  - ThreadX
  - Nucleus
- **For large-size embedded systems**
  - Linux
  - RTLinux
  - VxWorks
  - VxWorks

### Example I: PALOS
- **Power Aware Lightweight Operating System or Parasite real-time, Application-specific Lightweight Operating System initiated by the UCLA Network and Embedded Systems Lab**
- **Structure**
  - PALOS Core
  - Drivers, Managers, and user-defined Tasks
- **PALOS Core**
  - Task control: slowing, stopping, resuming
  - Periodic and aperiodic handlers
  - Inter-task Communication via event queues
  - Event-driven tasks: task routine processes events stored in event queues while (eventQ != 0); (dequeue event: process event)
- **Drivers**
  - Processor-specific: UART, SPI, Timers...
  - Platform-specific: Radio, LEDs, Sensors

### Tasks in PALOS
- A task belongs to the PALOS main control loop
- Each task has an entry in PALOS task table (along with eventQs)
- Execution control
  - A task counter is associated with each task
  - Counters are initialized to predefined values
  - Counts are increment or decrement
  - On-negative: restart
  - On-stop: completion
  - On-reset: terminate
  - On-counter reaches 0: interrupt (push stack)
  - When counter reaches zero, the task routine is called. The counter is reset to initial value.

### Event Handlers in PALOS
- Periodic or aperiodic events can be scheduled using Delta Q and Timer Interrupt
- When event expires, appropriate event handler is called
The ARM Instruction Set

**PALOS Features**
- **Portable**
  - CPUs: ATmega103, ATmega128L, TMS320, StrongThumb
  - Boards: MICA, iBadge, MK2
- **Small Footprints**
  - Core (compiled for ATmega128L)
    - Code Size: 968 Bytes, Mem Size: 148 Bytes
  - Typical: 3 drivers, 3 user tasks
    - Code Size: 8 Kbytes, Mem Size: 1.3 Kbytes
- **Task execution control**
  - Provides means of controlling task execution (slowing, stopping, and resuming)
  - Scheduler: Multiple periodic, and aperiodic functions can be scheduled
- **Preliminary version v0.11 available from sourceforge**
  - [https://sourceforge.net/project/showfiles.php?group_id=61125](https://sourceforge.net/project/showfiles.php?group_id=61125)

**TinyOS Key Claims/Facts**
- Stylized programming model with extensive static information
- Easy migration across h/w/s/w boundary
- Very Small Software Footprint 3.6 KB
- Two level scheduling structure
  - Preemptive scheduling of event handlers
  - Non-preemptive FIFO scheduling of tasks
  - Bounded size scheduling data structure
  - Power-aware: puts processor to sleep when tasks are complete
- Rich and Efficient Concurrency Support
  - Events propagate across many components
  - Tasks provide internal concurrency
  - At peak load 50% CPU sleep
- Power Consumption on Rene Platform
  - Transmission Cost: 1 μJ/bit
  - Inactive State: 5 μA
  - Peak Load: 20 mA
- Efficient Modularity
  - Events propagate through stack <40 μs
  - Programming in C with lots of macros, now moving to NestC

**Example II: TinyOS**
- OS for UC Berkeley’s Motes wireless sensor nodes
- System composed of
  - Time scheduler
  - Task scheduler
  - Command scheduler
  - Events

**Example of TinyOS Application**

**Example III: µCOS-II**
- Portable, ROMable, scalable, preemptive, multitasking RTOS
  - Up to 63 statically declared tasks
- **Services**
  - Semaphores, event flags, mailboxes, message queues, task management, fixed-size memory block management, time management
- **Source freely available for academic non-commercial usage for many platforms**
  - Value added products such as Gull, TCP/IP stack etc.
  - Book "MicroC/OS-II: The Real-Time Kernel" describes the internals
- **Like VxWorks, used in many defense applications**

**Example IV: eCos**
- Embedded, Configurable OS
- Open-source, from RedHat
- Designed for devices lower-end than embedded Linux
- Several scheduling options
  - bit-map scheduler, lottery scheduler, multi-level scheduler
- **Three-level processing**
  - Hardware interrupt (ISR), software interrupt (DSI), threads
  - Inter-thread communication
  - Mutex, semaphores, condition variables, flags, message box
- **Portable**
  - Hardware Abstraction Layer (HAL)
- Based on configurable components
  - Package based configuration tool
  - Kernel sizes from 32 KB to 32 MB
  - Implements ITRON standard for embedded systems
  - OS-neutral POSIX compliant EL/I/IX API
**Example V: Real-time Linuxes**

- **Microcontroller (no MMU)** OSes:
  - uClinux - small footprint Linux (< 512KB kernel) with full TCP/IP
- **QoS extensions for desktop**:
  - Linux-SRT and QLinux (Umass)
  - soft real-time kernel extension
  - target: media applications
- **Embedded PC**
  - RTLinux, RTAI
  - hard real-time OS
  - e.g., RTLinux has Linux kernel as the lowest priority task in a RTOS
  - fully compatible with GNU/Linux
  - HardHat Linux (MontaVista)

**Example VI: RTLinux**

- **Real Time Kernel (at the lowest level)**
  - Native Linux Kernel is a low priority thread
  - Executed only when no real time tasks
- **Real Time Tasks**
  - Executed only when no real time tasks
  - Interrupts trapped by the Real Time Kernel
  - and passed onto Linux Kernel
  - Software simulation to hardware interrupts
  - Interrupts are serviced by RTLinux
  - Software emulation of hardware interrupts

**Conclusions**

- **RTOS selection depends upon many factors:**
  - Regulatory demands
  - Application requirements
  - Platform resources and capabilities
  - Communications capabilities
  - Time management, synchronization
  - Support tools
- **Validating hard real-time performance** is a complex undertaking.
  - Formal methods are powerful but too complex
  - Alternative is extensive testing
  - In practice, hard real-time systems tend to be “over designed” (i.e., low resource utilization).