Real Time Operating Systems

Embedded Real Time Systems
Prof. Davide Brugali
Università degli Studi di Bergamo
Characteristics of a RTS

- **Concurrent** control of separate system components — devices operate in parallel in the real-world; better to model this parallelism by concurrent entities in the program.

- **Extreme reliability and safe** — embedded systems typically control the environment in which they operate; failure to control can result in loss of life, damage to environment or economic loss.

- **Guaranteed response times** — we need to be able to predict with confidence the worst case response times for systems; efficiency is important but predictability is essential.
Desirable features of a RTOS

- **Timeliness**: OS has to provide mechanisms for
  - time management
  - handling tasks with explicit time constraints

- **Predictability**: to guarantee in advance the deadline satisfaction to notify when deadline cannot be guaranteed

- **Fault tolerance**: HW/SW failures must not cause a crash

- **Design for peak load**: All scenarios must be considered
Features of Real-Time Kernels

- Most real-time systems do not provide the features found in a standard desktop system
  - Real-time systems are typically single-purpose
  - Real-time systems often do not require interfacing with a user
  - Features found in a desktop PC require more substantial hardware that what is typically available in a real-time system

- In general, real-time operating systems must provide:
  1. Preemptive, priority-based scheduling
  2. Preemptive kernels
  3. Latency must be minimized
Minimizing Latency

- **Event latency** is the amount of time from when an event occurs to when it is serviced.

![Diagram showing event latency with time line and event occurrence points](image)
Interrupt Latency

- **Interrupt latency** is the period of time from when an interrupt arrives at the CPU to when it is serviced.
Dispatch Latency

- **Dispatch latency** is the amount of time required for the scheduler to stop one process and start another.
Real-Time Operating Systems

• Timeliness
  • Achieved through proper scheduling algorithms
    • Core of an RTOS!

• Predictability
  • Affected by several issues
    • Characteristics of the processor (pipelining, cache, DMA, …)
    • I/O & interrupts
    • Synchronization & IPC
    • Architecture
    • Memory management
    • Applications
    • Scheduling!
Achieving Predictability: DMA

- **Direct Memory Access**
  - To transfer data between a device and the main memory
  - Problem: I/O device and CPU share the same bus

2 possible solutions:

- **Cycle stealing**
  - The DMA steals a CPU memory cycle to execute a data transfer
  - The CPU waits until the transfer is completed
  - Source of non-determinism!

- **Time-slice method**
  - Each memory cycle is split in two adjacent time slots
    - One for the CPU
    - One for the DMA
  - More costly, but more predictable!
Achieving Predictability: Cache

To obtain a high predictability it is better to have processors without cache

Source of non-determinism

- cache miss vs. cache hit
- writing vs. reading
Achieving Predictability: Interrupts

One of the biggest problem for predictability

- Typical device driver
  <enable device interrupt>
  <wait for interrupt>
  <transfer data>

- In most OS
  - interrupts served with respect to fixed priority scheme
  - interrupts have higher priorities than processes
  - How much is the delay introduced by interrupts?
    - How many interrupts occur during a task?

- problem in real-time systems
  - processes may be of higher importance than I/O operation!
Interrupts: First Solution Attempt

Disable all interrupts, but timer interrupts

**Advantages**

- All peripheral devices have to be handled by tasks
- Data transfer by polling
- Great flexibility, time for data transfers can be estimated precisely
- No change of kernel needed when adding devices

**Problems**

- Degradation of processor performance (busy wait)
- Task must know low level details of the drive
Interrupts: Second Solution Attempt

- Disable all interrupts but timer interrupts, and handle devices by special, *timer-activated* kernel routines

**Advantages**
- unbounded delays due to interrupt driver eliminated
- periodic device routines can be estimated in advance
- hardware details encapsulated in dedicated routines

**Problems**
- degradation of processor performance (still busy waiting within I/O routines)
- more inter-process communication than first solution
- kernel has to be modified when adding devices
Interrupts: Third Solution Attempt

Enable external interrupts and reduce the drivers to the least possible size

- Driver only activates proper task to take care of device
- The task executes under direct control of OS, just like any other task
- User tasks may have higher priority than device tasks

Advantages

- busy wait eliminated
- unbounded delays due to unexpected device handling dramatically reduced (not eliminated!)
- remaining unbounded overhead may be estimated relatively precisely

State of the art!
Achieving predictability: System Calls

- All system calls have to be characterized by bounded execution time
  - each kernel primitive should be preemptable!
  - non-preemptable calls could delay the execution of critical activities → system may miss hard deadline
Achieving predictability: Semaphore

- Usual semaphore mechanism not suited for real-time applications
- Priority inversion problem
- High priority task is blocked by low priority task for unbounded time
- Solution: use special protocols
  - Priority Inheritance
  - Priority ceiling
Real-Time Linux
Evolution of Linux

• Early Linux Not Designed for Real-Time Processing
  • Early Linux (1.x Kernel) installations on retired Windows PCs
    • Linux outperformed Windows in reliability and uptime (still does)
  • **Linux Design**: Fairness, Throughput and Resource-Sharing
    • Basic Unix development design principles applied in Kernel
    • User tasks should not stall under heavy load
    • Does not drop network connections or starve users / applications
    • System resources must be shared fairly between users
  • Fairness, progress and resource-sharing conflict with the requirements of time-critical applications
  • UNIX systems (and Linux) are historically not Real-Time OS
Real-Time Inhibitor

Linux Kernel Critical Sections

- Critical sections protect shared resources, e.g. hardware registers, I/O ports, and data in RAM
- Critical sections are shared by Processes, Interrupts and CPUs.
- Critical sections must be locked and unlocked
- **Locked critical sections are not pre-emptable**
Real-Time Inhibitor

**Existing Locking Subsystems are not Priority-Aware**

- **System semaphore**
  - Counting semaphore used to wake multiple waiting tasks
  - No support for priority inheritance
  - No priority ordering of waiters

- **Big Kernel Lock (BKL)**
  - Originally non-preemptable, now preemptable using system semaphore
  - Can be released by blocking tasks, re-acquired upon wake-up
  - No priority-awareness, or priority inheritance for contending tasks

- **Read – Write Locks**
  - Classical blocking / starvation issues with no priority awareness
The Fully Preemptible Linux Kernel

- Dramatic Reduction in 2.6 Preemption Latencies
  - Multiple Concurrent Tasks in Independent Critical Sections
  - Generally Fully Preemptible → “No Delays”
    - Non-preemptible: lowest-level interrupt management
    - Non-preemptible: Scheduling and context switching code
Linux Real-Time Technology Overview

- Linux 2.6 Kernel Real-Time Technology Enhancements

- Preemptible Interrupt Handlers in Thread Context
  - Default: IRQs (interrupt request from a device) run in threads
  - IRQ Thread can have private stack
  - Real-Time tasks at Higher Priority than IRQ handlers

- Integrated Kernel Mutex with Priority Inheritance (PI)
  - Preemptible PI Mutex protects Kernel Critical Sections
  - Big Kernel Lock (BKL) converted to PI Mutex
  - Read-Write Locks converted to PI Mutex

- High Resolution Timers
Thread-Context Interrupt Handlers

**Threaded IRQs Pros**

- RT IRQs do not contend with common IRQs
- IRQ Processing does not Interfere with task execution
- Flexible priority assignment
  - can be arranged to emulate hardware-based priorities
- Interrupts run fully preemptible

**Threaded IRQs Cons**

- IRQ-Thread Overhead
  - Scheduler must run to activate IRQ Threads
- IRQ Thread Latency
  - IRQS no longer running at the highest priority
  - Full task switch required to handle IRQ
- Response-Time / Throughput tradeoff
Real-Time Linux Kernel Evolution

- Gradual SMP-Oriented Linux Kernel Optimizations

<table>
<thead>
<tr>
<th>Kernel Version</th>
<th>Preemption Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Early Kernel 1.x</td>
<td>No Kernel preemption</td>
</tr>
<tr>
<td>SMP Kernel 2.x</td>
<td>No Kernel preemption, “BKL” SMP Lock</td>
</tr>
<tr>
<td>SMP Kernel 2.2 - 2.4</td>
<td>No preemption, Spin-locked Critical Sections</td>
</tr>
<tr>
<td>“Preempt” Kernel 2.4</td>
<td>Kernel Preemption outside Critical Sections,</td>
</tr>
<tr>
<td></td>
<td>Spin-locked Critical Sections</td>
</tr>
<tr>
<td>Kernel 2.6</td>
<td>Kernel Preemption outside Critical Sections, Preemptible</td>
</tr>
<tr>
<td></td>
<td>“BKL”, O(1) Scheduler</td>
</tr>
<tr>
<td>“RT-Preempt” Kernel</td>
<td>Kernel Critical sections Preemptible</td>
</tr>
<tr>
<td></td>
<td>IRQ Subsystem Prioritized and Preemptible</td>
</tr>
<tr>
<td></td>
<td>Mutex Locks with Priority Inheritance</td>
</tr>
<tr>
<td></td>
<td>High-Resolution Timers</td>
</tr>
</tbody>
</table>
Efficiency and Responsiveness are Inversely Related

Overhead for Real-Time Preemption

- Mutex Operations more complex than spinlock operations
- Priority Inheritance on Mutex increases task switching
- Priority Inheritance increases worst-case execution time
- Interrupt overhead
- Additional Task Switching
- Interrupt Preemption → Interrupt throughput reduction

Real-Time Response vs. Throughput
RTLinux

- Available as a patch to the regular Linux kernel
- Provides an RT API for developers
- RTLinux is a hybrid OS that runs a Linux kernel as an idle thread (lowest priority) of the real-time kernel.
- Predictable delays.
  - By its small size and limited operations.
- Finer timer resolution.
- RT kernel and RT applications are kept as simple as possible and non-time critical applications (GUIs, file systems) are handled by the standard Linux.
Real time threads and interrupt handlers never delayed by non-realtime operations.

Preemptible kernel.
- Its routines are very small and fast, this does not cause big delays.
- Interrupts from Linux are disabled.

RT-Linux has many kinds of Schedulers.
- FIFO.
  - Used to pass information between real-time process and ordinary Linux process.
  - Designed to never block the real-time task.
- The “earliest deadline first” scheduler.
- Rate-monotonic scheduler.
Figure 1.1: Detail of the bare Linux kernel
Figure 1.2: Detail of the RTLinux kernel
RTAI
Real-Time Application Interface
RTAI (Real Time Application Interface)

- Hard real-time extension to the Linux kernel
- A patch to the Linux kernel which introduces a hardware abstraction layer
- A broad variety of services which make realtime programmers' lifes easier
- RTAI provides deterministic response to interrupts, POSIX compliant and native RTAI realtime tasks.
- Linux application is able to execute without any modification
- RTAI considers Linux as a background task running when no real time activity occurs.
RTAI (Contd)

- RTAI is very much module oriented
- real time scheduler module
  - Task functions
  - Timing functions
  - Semaphore functions
  - Mailbox functions
  - Intertask communication functions
- Fifo services
- Shared memory
- Posix pthread and pqueue(msg queue)
Comparison of Linux implementations
RTLinux and RTAI

- RTAI provides better real-time support than RTLinux
  - soft real-time in user space along with hard real-time in kernel space
  - excellent performance in terms of low jitter and low latency
  - better C++ support and more complete feature set
  - availability of LXRT which allows user space applications in kernel space
- RTAI has the better open source approach with frequent feedback from developers
Vx-Works


http://bwrc.eecs.berkeley.edu/Classes/CS252/Notes/Lec26a-sw.pdf
VxWorks

- Created by Wind River.
- Current Version: VxWorks 6.0
- VxWorks is the most established and most widely deployed device software operating system.
- Currently there are more than 300 million devices that are VxWorks enabled.
- The core attributes of VxWorks, include high performance, reliability, determinism, low latency and scalability.
VxWorks (contd..)

- Enhanced error management
- Backward compatibility to previous version features for exception handling and template support
- Extensive POSIX 1003.1, .1b, .1c compatibility (including pthreads)

- Scheduling
  - Uses preemptive priority with round robin scheduling to accommodate for both
    - Real time processes
    - Non-real time processes
• Memory Protection
  • MMU based memory protection.

• Reduced Context Switch time
  • Saves only those register windows that are actually in use (on a Sparc)
  • When a task’s context is restored, only the relevant register window is restored
  • To increase response time, it saves the register windows in a register cache – useful for recurring tasks
VxWorks (contd..)

- Distinguishing features
  - efficient POSIX-compliant memory management
  - multiprocessor facilities
  - shell for user interface
  - symbolic and source level debugging capabilities
  - performance monitoring

- Mars Exploration Rovers *Spirit* and *Opportunity* and the Mars Reconnaissance Orbiter use the VxWorks operating system
Xenomai
XENOMAI: Real-Time Framework for Linux

- Xenomai is a real-time development framework cooperating with the Linux kernel, in order to provide a pervasive, interface-agnostic, hard real-time support to user-space applications, seamlessly integrated into the GNU/Linux environment.

- Generic RT-core (“nucleus”)
- Kernel-independent, but highly integrated with Linux
- RTOS APIs provided via “skins”

- Xenomai was born in 2001, out of a basic idea: support traditional RTOS APIs over a Linux-based real-time framework, so that existing industrial applications coming from the proprietary world could migrate easily to a GNU/Linux-based environment, while keeping stringent real-time guarantees. To this end, the core Xenomai technology exhibits an abstract real-time nucleus, which provides generic building blocks for implementing real-time APIs, aka "skins". This way, a skin can emulate a proprietary API efficiently, based on a reusable real-time core.
Xenomai Skin

- POSIX
- Native (clean RTAI-like API)
- VxWorks
- VRTX
- pSOS+
- μlTRON
- RTAI
- RTDM

Avionics
ARINC 653?
Automotive
OSEK?
AUTOSAR?

http://www.xenomai.org/documentation/branches/v2.3.x/pdf/xenomai.pdf
The RTDM (Real-Time Driver Model) interface provides a skin-independent framework for writing real-time device drivers, and accessing them through a POSIX interface. RTDM helps to follow the well-known design principle of clean separation between hardware interface and application program, also when strict deterministic behaviour is required.
### Xenomai Featuring:

- **RTnet** *(RT-networking stack)*
- **RT-FireWire** *(RT-IEEE1394 stack)*
- **USB4RT, USB20RT** *(RT-USB stacks)*
- **COMEDI over RTDM** *(DAC driver framework)*
- **OROCOS** *(RT-middleware)*
- **RACK** *(Robotics RT-middleware)*
- **CanFestival** *(CANopen library)*
- **Xeno--** *(C++ & Python wrapping library)*
- **LTTng** *(System event tracing)*
- **kgdb** *(Remote kernel debugger)*