Real Time Operating Systems

Prof. Tajana Simunic Rosing
Department of Computer Science and Engineering
University of California, San Diego.
Real-time operating systems

Three key requirements

1. **Predictable OS timing behavior**
   - upper bound on the execution time of OS services
   - short times during which interrupts are disabled,
   - contiguous files to avoid unpredictable head movements

2. **OS must be fast**

3. **OS must manage the timing and scheduling**
   - OS possibly has to be aware of task deadlines;
     (unless scheduling is done off-line).
   - OS must provide precise time services with high resolution.
RTOS-Kernels

- Distinction between real-time kernels and modified kernels of standard OSes.

- Distinction between general and RTOSes for specific domains,
- standard APIs (e.g. POSIX RT-Extension of Unix) or proprietary APIs.
How to organize multiple tasks?

• Cyclic executive (Static table driven scheduling)
  • static schedulability analysis
  • resulting schedule or table used at run time

• Event-driven non-preemptive
  • tasks are represented by functions that are handlers for events
  • next event processed after function for previous event finishes

• Static and dynamic priority preemptive scheduling
  • Static schedulability analysis
  • At run time tasks are executed “highest priority first”
  • Rate monotonic, deadline monotonic, earliest deadline first, least slack
RTOS Organization:
Cyclic Executive

Kernel Mode:
- Application
- Device Drivers
- I/O Services
- TCP/IP Stack

Platform:
- Network Drivers

Hardware
RTOS Organization:
Monolithic Kernel

- User Mode (protected)
- Kernel Mode
- Application
  - Filesystems
  - Device Drivers
  - Network Drivers
  - I/O Managers
  - Graphics Subsystem
  - Graphics Drivers
  - Other….

Hardware Interface Layer
Hardware
RTOS Organization: Microkernel

User Mode (protected)

- Application
- Application
- Application

Kernel Mode

- Kernel (tiny)
- Network Manager
- Photon
- Device Manager
- Filesystem Manager
- Network Drivers
- Graphics Drivers
- Device Drivers
- Filesystem Drivers

Hardware
Faster kernels

• Designed to be fast
  – Many are inadequate for complex systems
• Proprietary examples:
  – QNX, PDOS, VCOS
  – VxWORKS
• Open source example:
  – FreeRTOS
Automatic dependency analysis and size calculations allow users to quickly customize the VxWORKS operating system.
VxWorks 7

- 1.5 billion embedded devices use it
  - world’s most widely deployed proprietary RTOS
- Recently introduced virtualization, multicore scheduling, MMU support, security & safety infrastructure
- Supports lots of CPU architectures: ARM, Intel, etc.
Portability with Virtualization
User vs. Kernel Mode & Security

- Protected user mode: applications
- Unprotected kernel mode: OS kernel & drivers
- VxWorks has a lightweight kernel-based threading model

ELOC = effective lines of code; all lines that are not comments, blanks or standalone braces or parenthesis.
VxWorks Task States

- **Suspended** – used primarily for debugging
- **Ready** – waiting for CPU, a single primitive = create + activate
- **Pended** – blocked, wait $\Delta t$ time for recourses.
- **Delay** – asleep for $\Delta t$ time, after $\Delta t$ goes to ready state.

On a context switch, a task’s context is saved in task control block (TCB).
Real-Time Process (RTP) Model

• Processes run in their own memory space
  – The kernel and other processes are protected
• Takes advantage of CPUs with a memory management unit (MMU)
• RTPs are isolated both from the kernel and from each other – error protection
• Application libraries and data memory can be shared between RTPs
• Supports both user and kernel modes
Kernel and RTP Interaction

• A system call trap interface is used to access kernel services
• Tasks in different RTPs may interact using shared data regions and inter-process communication (IPC) mechanisms
Memory Model

- Virtual memory manager
- Kernel heap is reduced to create an area of unmapped physical pages
- Kernel heap size is configurable
- An RTP task running in user mode only has access to the RTPs memory space
- The task can trap into kernel mode via a system call
Scheduling

• A global task scheduler schedules tasks across all RTPs
  • Only tasks are schedulable
• Priority-based preemptive scheduling
  • Each task has priority (0 - highest to 255) with its own queue.
  • If two tasks are ready, a lower priority task is pre-empted
  • Priorities can be changed at runtime.
  • A user can lock a task so that it can’t be preempted even by higher priority tasks or interrupts. This allows the use of the fixed priority response time analysis to check schedulability offline.
• Round-robin scheduling
  • After time slice for a task expires, another task with the same priority will execute during the given time slice.
• Preemption locks
  • They prevent task context switching, but do not prevent interrupts.
• Handles priority inversion with priority ceiling and priority inheritance
VxWorks task scheduling examples

- **Priority-bases preemptive**

- **Round-Robin**
Scheduling: task criticality and CPU affinity

Source: VxWorks
Criticality in VxWorks

- Foreground vs. background RTPs
Interrupts

• Inform the system of external events
• Interrupt Service Routines (ISRs) run outside any task context -> no task context switch
• Limitations of ISR:
  – All ISRs share the same stack
  – ISR has no context that can be suspended
  – Cannot take the semaphore, but can give the semaphores, releasing any task waiting on them.
  – Cannot perform I/O through drivers except pipe.
Shared Code and Reentrancy

- *Shared Code* - a single copy of code executed by multiple tasks.
  - Must be *reentrant*
    - A single copy of the routine can be called simultaneously from several task contexts without conflict.

- Reentrancy Techniques:
  - Dynamic Stack Variables
  - Global and Static Variables Guarded by Semaphores
    - Mutual exclusion – only one task can change data at a time
    - Uses counting semaphore with timeout (real time)
  - Task Variables
    - 4B variables added to TCB
Dynamic Stack Variables

Tasks

Task1()
{
    ....
    comFun(myData1)
}

Task2()
{
    ....
    comFun(myData2)
}

Task Stacks

myData1

myData2

Common Function

comFun(yourData)
{
}

24
Intertask Communication

- VxWorks supports:
  - Shared memory
  - Semaphores (binary & counting)
    - Timeout option: If time > timeout, ERROR occurs
  - Mutexes (POSIX interfaces)
  - Message queues and Pipes
  - Sockets and RPCs
  - Signals

- The mutex semaphore supports the priority-inheritance algorithm

Source: Brandon Miller
Microkernel with VxWorks

• A complete high-performance RTOS in a tiny footprint
  – 20 KB or smaller in size
• Very fast response times
  – Time to boot: 25 µs (on Intel Quark)
  – Time to switch from fiber to ISR execution: 1955 ns (on Intel Quark)
• Multi-threaded
• Multi-core support
• Power management ready
Application Examples

- Flight simulators
- Radio and optical telescopes
- Navigation systems
- Deep sea instrumentation
- Traffic control systems
- Modems

- Printers
- Digital cameras
- Hand-held computing devices
- Routers, switches, etc

… any systems where rigid time requirement have been placed on the operation of a processor or the flow of the data.
Medical application example

• BD Biosciences is a leading global provider of systems for sorting and analyzing CD4 cells to monitor the stages of HIV/AIDS infection and the effectiveness of new vaccines and therapies.

• VxWorks allowed BD to cut its development time by 25%!
VxWorks & Mars Rover Curiosity

- NASA chose VxWorks to run the craft's controls
  - rocket left Earth Nov 2011
  - successfully landed on Mars Aug 2012
- VxWorks supported more than 20 JPL missions
Open Source Fast RTOS: FreeRTOS

- Leading open source RTOS
  - E.g. basis for DuinOS used in Arduino

- Key features:
  - Preemptive and co-operative scheduling, Multitasking, Services, Interrupt management, MMU; Supports stacks for TCP/IP, USB, & basic file systems

- Highly portable C, 24 architectures supported, Ports are freely available in source code.

- Scalable:
  - Only use the services you only need by specifying in FreeRTOSConfig.h
  - Minimum footprint = 4KB
Scheduling & Multitasking

• Preemptive task scheduling:
  o Fully preemptive
  o Always runs the highest priority task that is ready to run
  o Task priorities are dynamic
  o Comparable with other preemptive kernels
  o Cooperative
    o Context switch occurs if a task blocks or yields the CPU

• Multitasking:
  • No software restriction on # of tasks that can be created, # of priorities that can be used
  • Priority assignment
    ▪ More than one task can be assigned the same priority.
    ▪ RR with time slice = 1 RTOS tick
Services & Interrupts

- Services
  - Queues
  - Semaphores
    - Binary and counting
  - Mutexes
    - With priority inheritance
    - Support recursion

- Interrupts
  - An interrupt can suspend a task execution, mechanism is port dependent.
Task status in FreeRTOS

- **Running**
  - Task is actually executing

- **Ready**
  - Task is ready to execute but a task of equal or higher priority is Running.

- **Blocked**
  - Task is waiting for some event.
    - **Time**: if a task calls vTaskDelay() it will block until the delay period has expired.
    - **Resource**: Tasks can also block waiting for queue and semaphore events.

- **Suspended**
  - Much like blocked, but not waiting for anything.
  - Tasks will only enter or exit the suspended state when explicitly commanded to do so via API.

Mostly from http://www.freertos.org/RTOS-task-states.html
Standard OS with real-time extensions

- RT-kernel running all RT-tasks.
- Standard-OS executed as one task.

<table>
<thead>
<tr>
<th>RT-task 1</th>
<th>RT-task 2</th>
<th>non-RT task 1</th>
<th>non-RT task 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>device driver</td>
<td>device driver</td>
<td>Standard-OS</td>
<td></td>
</tr>
</tbody>
</table>

+ Crash of standard-OS does not affect RT-tasks;
- RT-tasks cannot use Standard-OS services;
  less comfortable than expected
Example: RT-Linux

- **Init**
- **Bash**
- **Mozilla**

- **Linux-Kernel**
  - scheduler
  - driver

- **RT-Linux**
  - RT-Scheduler

- **RT-Task**
- **RT-Task**

- **Hardware**

Connections:
- **interrupts** from **Hardware** to **RT-Linux**, **RT-Task**, **Mozilla**, **Bash**
- **interrupts** from **RT-Linux** to **RT-Task**, **Bash**, **Mozilla**
- **I/O** from **Hardware** to **RT-Linux**
RT-Linux philosophy

• Real time programs need to be split into two parts:
  – a small light component with hard real time constraints, scheduled via real time scheduler
  – larger component that does most of the processing, scheduled via normal Linux scheduler

• The two parts communicate through a real-time-FIFO, a non-blocking queue which allows the real time parts to read and write data from normal processes.
RT-Linux schedulers

• Real time tasks are implemented as kernel modules
  – At initialization they send the real time scheduler all the information about their release times, periods and deadlines.

• Rt-Linux allows for swappable real-time schedulers:
  – EDF algorithm, RM algorithm & priority based round-robin
  – Does not schedule dependent real-time tasks
  – Memory is allocated in non-pageable blocks saving the uncertainty of swap file usage
  – All real time tasks run in the same kernel memory space
    • No protected memory benefits
    • Context switches are much faster because we save the cost of flushing the TLB cache and using trap instructions to enter and exit kernel mode

• Linux is the lowest priority process
  – It can be preempted anytime when RT scheduler requires
  – Executes only if the real time scheduler has nothing else to do.
  – Linux uses its own scheduler to schedule tasks running under it.
RT-Linux & interrupts

• A very serious problem is the disabling of interrupts by the Linux kernel
  – Function call to disable interrupts in the kernel source is replaced with a macro that masks these interrupts from the Linux process but not from the other real time processes

• Real-time kernel handles masking all the interrupts
  – It passes the ones intended to the Linux process on to it as software interrupts if Linux enabled interrupts

• RTLinux is very fast in handling interrupts
  – running on a x86 generates runs in less than 15μs from the moment a hardware interrupt is detected to when its interrupt handler starts to execute
RT-Linux - summary

• Demonstrates the “hack in to the architecture” approach to adding real time.

• Example application: Henson Company Creature Shop's Performance Control System
  – The "animatronic" version of the system is capable of precise control of real-world puppets that are actuated by electromechanical or hydraulic servos.
  – The system runs RTLinux on the front end for I/O and timing & communicates serially with a back-end embedded PC residing in the puppet, which communicates serially with microcontroller-based motor driver peripherals.
RTOS for Raspberry Pi

- Limitation of Raspberry Pi for real-time OS support
  - No real-time clock
    - Cannot maintain the actual time and date (should be connected to the internet)
    - Cannot generate deterministic timing pulses
    - Need to connect additional hardware modules
  - There is no support of real-time in standard Linux / BSD kernels
    - OS may be slow due to processors

- RTOS kernel ports for Raspberry Pi
  - FreeRTOS: basic port to Raspberry Pi
  - ChibiOS/RT: efficient and preemptive kernel
  - Xenomai: real-time extension of Linux
  - Machinoid: hard real-time support targeting to Robotics, CNC and 3D Printing
  - RTEMS: similar to VxWorks and ported to Raspberry Pi
RTOS Research systems

- **Research issues**
  - low overhead memory protection,
  - temporal protection of computing resources
  - RTOSes for on-chip multiprocessors
  - support for continuous media
  - quality of service (QoS) control.
Research Kernel Examples

• Small kernels
  – PALOS, TinyOS

• Medium size
  – uCos II, eCos

• Larger
  – Linux-based systems
Small kernel: PALOS

- Structure – PALOS Core, Drivers, Managers, and user defined Tasks
- PALOS Core
  - Task control: slowing, stopping, resuming
  - Periodic and aperiodic handling and scheduling
  - Inter-task Communication via event queues
  - Event-driven tasks: task routine processes events stored in event queues
- Drivers
  - Processor-specific: UART, SPI, Timers..
  - Platform-specific: Radio, LEDs, Sensors
- Small Footprint
  - Core (compiled for ATMega128L) Code Size: 956 Bytes, Mem Size: 548 Bytes
  - Typical (3 drivers, 3 user tasks) Code Size: 8 Kbytes, Mem Size: 1.3 Kbytes
Execution control in PALOS

- Each task has a task counter
- Counters initialized to:
  - 0: normal
  - >>:0 slowdown
  - -1: stop
  - >=0: restart
- Decremented
  1) every iteration (relative timing)
  2) by timer interrupts (exact timing)
- If counter = 0, call tasks; reset counter to initialization 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
Small Kernel: TinyOS

- System composed of
  - scheduler, graph of components, execution context
- Component model
  - Basically FSMs
  - Four interrelated parts of implementation
    - Encapsulated fixed-size frame (storage)
    - A set of command & event handlers
    - A bundle of simple tasks (computation)
  - Modular interface
    - Commands it uses and accepts
    - Events it signals and handles
- Tasks, commands, and event handlers
  - Execute in context of the frame & operate on its state
  - Commands are non-blocking requests to lower level components
  - Event handlers deal with hardware events
  - Tasks perform primary work, but can be preempted by events
- Scheduling and storage model
  - Shared stack, static frames
  - Events preempt tasks, tasks do not
  - Events can signal events or call commands
  - Commands don’t signal events
  - Either can post tasks
TinyOS Overview

• Stylized programming model with extensive static information
  – Compile time memory allocation
• Easy migration across h/w - s/w boundary
• Small Software Footprint - 3.4 KB
• Two level scheduling structure
  – Preemptive scheduling of event handlers
  – Non-preemptive FIFO scheduling of tasks
  – Bounded size scheduling data structure
• Rich and Efficient Concurrency Support
  – Events propagate across many components
  – Tasks provide internal concurrency
• 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
Complete TinyOS Application

Component Name | Code Size (bytes) | Data Size (bytes)
--- | --- | ---
Multihop router | 88 | 0
AM_dispatch | 40 | 0
AM_temperature | 78 | 32
AM_light | 146 | 8
AM | 356 | 40
Packet | 334 | 40
RADIO_byte | 810 | 8
RFM | 310 | 1
Photo | 84 | 1
Temperature | 64 | 1
UART | 196 | 1
UART_packet | 314 | 40
I2C_bus | 198 | 8
Processor_init | 172 | 30
TinyOS scheduler | 178 | 16
C runtime | 82 | 0
Total | 3450 | 226

Source: Hill, Szewczyk et. al., ASPLOS 2000
Medium size: µCOS-II

• Portable, ROMable, scalable, preemptive, multitasking RTOS

• 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 GUI, TCP/IP stack etc.
Medium size: eCos

- Embedded, Configurable OS, Open-source
- Several scheduling options
  - bit-map scheduler, lottery scheduler, multi-level scheduler
- Three-level processing
  - Hardware interrupt (ISR), software interrupt (DSR), 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 size from 32 KB to 32 MB
  - Implements ITRON standard for embedded systems
  - OS-neutral POSIX compliant EL/IX API
Larger Size: Linux-based

- Microcontroller (no MMU) OSes:
  - uClinux - small-footprint Linux (< 512KB kernel) with full TCP/IP

- QoS extensions:
  - Linux-SRT and QLinux
    - soft real-time kernel extension
    - target: media applications
  - Posix 1.b RT

- 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
Soft deadlines: Posix RT

- Standard scheduler can be replaced by POSIX scheduler implementing priorities for RT tasks

- Special RT & standard OS calls available.
- Easy programming, no guarantee of meeting deadlines
Embedded Market Study on Most Frequently Used Operating Systems

Only Operating Systems that had 2% or more are shown.

Copyright © 2013 by UBM/EE Times Group. All rights reserved.
Middleware

- Between applications and OS
- Provides a set of higher-level capabilities and interfaces
- Customizable, composeable frameworks
- Types of services:
  - component – independent of other services
    - E.g. communication, information, computation
  - integrated sets
    - e.g. distributed computation environment
  - integration frameworks
    - Tailor to specific domain: e.g. transaction processing
Integrated sets

- A set of services that take significant advantage of each other
- Example: Distributed Computing Environment (DCE)
  - Provides key distributed technologies – RPC, DNS, distributed file system, time synch, network security and threads service
  - From Open SW Foundation, supported by multiple architectures and major SW vendors
DCE

- DCE Security Service
  - DCE Distributed File Service
  - DCE Distributed Time Service
  - DCE Directory Service
  - Other Basic Services
  - DCE Remote Procedure Calls

- DCE Threads Services

- Operating System Transport Services
Integration frameworks middleware

• Integration environments tailored to specific domain

• Examples:
  – Workgroup framework
  – Transaction processing framework
  – Network management framework
  – Distributed object computing (e.g. CORBA, E-SPEAK, JINI, message passing)
Distributed Object Computing

• Advantages:
  – SW reusability, more abstract programming, easier coordination among services

• Issues:
  – latency, partial failure, synchronization, complexity

• Techniques:
  – Message passing (object knows about network)
  – Argument/Return Passing – like RPC
    • network data = args + return result + names
  – Serializing and sending
    • network data = obj code + obj state + synch info
  – Shared memory
    • network data = data touched + synch info
**SW for access to remote objects**

CORBA (Common Object Request Broker Architecture). Information sent to Object Request Broker (ORB) via local stub. ORB determines location to be accessed and sends information via the IIOP I/O protocol.

Access times not predictable.

**OBJ management architecture**
Real-time CORBA

• End-to-end predictability of timeliness in a fixed priority system.

• respecting thread priorities between client and server for resolving resource contention,

• bounding the latencies of operation invocations.

• RT-CORBA includes provisions for bounding the time during which priority inversion may occur.
Message passing interface

- Message passing interface (MPI): alternative to CORBA
- MPI/RT: a real-time version of MPI [MPI/RT forum, 2001].
- MPI-RT does not cover issues such as thread creation and termination.
- MPI/RT is conceived as a layer between the operating system and non real-time MPI.
Summary

• SW
  – Various apps

• Middleware
  – E.g DCE, CORBA

• RTOS
  – E.g TinyOS, eCos, RT-Linux
Sources and References

• Nikil Dutt @ UCI
• Mani Srivastava @ UCLA
• Brandon Miller, T.B. Skaali
• Amr Ali Abdel-Naby for FreeRTOS
• WindRiver/VxWorks