Embedded Systems

6. Real-Time Operating Systems

Lothar Thiele
Contents of Course

1. Embedded Systems Introduction

2. Software Introduction

3. Real-Time Models

4. Periodic/Aperiodic Tasks

5. Resource Sharing

6. Real-Time OS

7. System Components

8. Communication

9. Low Power Design

10. Models

11. Architecture Synthesis

12. Model Based Design

Software and Programming

Processing and Communication

Hardware
Embedded OS

Why an OS at all?
- Same reasons why we need one for a traditional computer.
- Not all services are needed for any device.

Large variety of requirements and environments:
- Critical applications with high functionality (medical applications, space shuttle, process automation, …).
- Critical applications with small functionality (ABS, pace maker, …)
- Not very critical applications with varying functionality (smart phone, smart card, microwave oven, …)
Embeded OS

Why is a *desktop OS not suited*?

- Monolithic kernel is too feature reach.
- Monolithic kernel is not modular, fault-tolerant, configurable, modifiable, … .
- Takes too much memory space.
- It is often too resource hungry in terms of computation time.
- Not designed for mission-critical applications.
- Timing uncertainty too large.
Embedded Operating Systems

Configurability

- No single RTOS will fit all needs, no overhead for unused functions/data tolerate: configurability is needed.
- For example, there are many embedded systems without external memory, a keyboard, a screen or a mouse.

Configurability examples:

- Simplest form: remove unused functions (by linker for example).
- Conditional compilation (using #if and #ifdef commands).
- Validation is a potential problem of systems with a large number of derived operating systems:
  - each derived operating system must be tested thoroughly;
  - for example, eCos (open source RTOS from Red Hat) includes 100 to 200 configuration points.
Automatic dependency analysis and size calculations allow users to quickly custom-tailor the VxWORKS operating system.

© Windriver
Embedded operating systems

*Device drivers handled by tasks* instead of hidden integrated drivers:

- Improve predictability; everything goes through scheduler
- Effectively no device that needs to be supported by all versions of the OS, except maybe the system timer.

<table>
<thead>
<tr>
<th>RTOS</th>
<th>Standard OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>application software</td>
<td>application software</td>
</tr>
<tr>
<td>middleware</td>
<td>middleware</td>
</tr>
<tr>
<td>device driver</td>
<td>device driver</td>
</tr>
<tr>
<td>real-time kernel</td>
<td>operating system</td>
</tr>
<tr>
<td></td>
<td>device driver</td>
</tr>
</tbody>
</table>
Embedded Operating Systems

Interrupts can be employed by any process

- For standard OS: this would be serious source of unreliability.
- But embedded programs can be considered to be tested . . .
- It is possible to let interrupts directly start or stop tasks (by storing the tasks start address in the interrupt table). More efficient and predictable than going through OS interfaces and services.

- However, composability suffers: if a specific task is connected to some interrupt, it may be difficult to add another task which also needs to be started by the same event.
- If real-time processing is of concern, time to handle interrupts need to be considered. For example, interrupts may be handled by the scheduler.
Embedded Operating Systems

**Protection mechanisms are not always necessary:**

- Embedded systems are typically designed for a single purpose, untested programs rarely loaded, software considered reliable.
- *Privileged* I/O instructions not necessary and tasks can do their own I/O.

  Example: Let `switch` be the address of some switch. Simply use
  
  ```
  load register,switch
  ```
  
  instead of a call to the underlying operating system.

- However, protection mechanisms may be needed for safety and security reasons.
Real-time Operating Systems

- A real-time operating system is an operating system that supports the construction of real-time systems.

- Three key requirements:
  1. The timing behavior of the OS must be **predictable**.
     \( \forall \) services of the OS: Upper bound on the execution time!

RTOSs must be deterministic (unlike standard Java for example):
  - upper bounds on blocking times need to be available, i.e. during which interrupts are disabled,
  - almost all activities are controlled by a real-time scheduler.
Task Management Services

External interrupt

Timer interrupt

System calls (trap)

Interrupt dispatch

Interrupt service

Time service & events

Scheduling & dispatcher

Services (create thread, sleep, notify, send, …)

Task execution

kernel
Real-time Operating Systems

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

3. The OS must be *fast*
   - Practically important.
Main Functionality of RTOS-Kernels

- **Task management:**
  - Execution of *quasi-parallel tasks* on a processor using processes or threads (lightweight process) by
    - maintaining process states, process queuing,
    - allowing for preemptive tasks (fast context switching) and quick interrupt handling
  - CPU *scheduling* (guaranteeing deadlines, minimizing process waiting times, fairness in granting resources such as computing power)
  - Process *synchronization* (critical sections, semaphores, monitors, mutual exclusion)
  - Inter-process *communication* (buffering)
  - Support of a *real-time clock* as an internal time reference
Task Management

★ **Task synchronization:**
- In classical operating systems, synchronization and mutual exclusion is performed via semaphores and monitors.
- In real-time OS, special semaphores and a deep integration into scheduling is necessary (priority inheritance protocols, ....).

★ **Further responsibilities:**
- Initializations of internal data structures (tables, queues, task description blocks, semaphores, …)
Task States

**Minimal Set of Task States**:

- **run**
- **idle**
- **ready**
- **wait**
- **dispatch**
- **end_cycle**
- **preemption**
- **resume**
- **signal**
- **activate**
- **terminate**

Diagram:

- **TIMER**
- Arrows indicate transitions between states:
  - From **run** to **idle**
  - From **run** to **wait**
  - From **run** to **ready**
  - From **idle** to **run**
  - From **ready** to **run**
  - From **ready** to **idle**
  - From **ready** to **wait**
  - From **wait** to **run**
  - From **wait** to **idle**
  - From **signal** to **ready**
  - From **preemption** to **ready**
  - From **resume** to **ready**
  - From **terminate** to **destroy**
Task states

- **Run:**
  - A task enters this state as it starts executing on the processor

- **Ready:**
  - State of those tasks that are ready to execute but cannot be executed because the processor is assigned to another task.

- **Wait:**
  - A task enters this state when it executes a synchronization primitive to wait for an event, e.g. a wait primitive on a semaphore. In this case, the task is inserted in a queue associated with the semaphore. The task at the head is resumed when the semaphore is unlocked by a signal primitive.

- **Idle:**
  - A periodic job enters this state when it completes its execution and has to wait for the beginning of the next period.
Threads

A thread is the smallest sequence of programmed instructions that can be managed independently by a scheduler; e.g., a thread is a basic unit of CPU utilization.

Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources:

- Typically shared by threads: memory.
- Typically owned by threads: registers, stack.

Thread advantages and characteristics:

- Faster to switch between threads; switching between user-level threads requires no major intervention by the operating system.
- Typically, an application will have a separate thread for each distinct activity.
- Thread Control Block (TCB) stores information needed to manage and schedule a thread.
Context Switching

process control block

process $P_0$

executing

operating system

interrupt or system call

save state into PCB$_0$

... 

reload state from PCB$_1$

idle

executing

process $P_1$

executing

interrupt or system call

save state into PCB$_1$

... 

reload state from PCB$_0$
Multiple Threads within a Process
Communication Mechanisms

*Problem*: the use of shared resources for implementing message passing schemes may cause priority inversion and blocking.
Communication mechanisms

- **Synchronous communication**: Whenever two tasks want to communicate they must be synchronized for a message transfer to take place (**rendez-vous**)
  - They have to wait for each other.
  - **Problem** in case of dynamic real-time systems: Estimating the maximum blocking time for a process rendez-vous.
  - In a **static** real-time environment, the problem can be solved off-line by transforming all synchronous interactions into precedence constraints.
Communication mechanisms

- **Asynchronous communication:**
  - Tasks do not have to wait for each other
  - The sender just deposits its message into a channel and continues its execution; similarly the receiver can directly access the message if at least a message has been deposited into the channel.
  - More suited for real-time systems than synchronous comm.
  - **Mailbox:** Shared memory buffer, FIFO-queue, basic operations are send and receive, usually has fixed capacity.
  - **Problem:** Blocking behavior if channel is full or empty; alternative approach is provided by cyclical asynchronous buffers.
Class 1: Fast Proprietary Kernels

**Fast proprietary kernels**

For hard real-time systems, these kernels are questionable, because they are designed to be fast, rather than to be predictable in every respect.

Examples include

- FreeRTOS
- QNX
- eCOS
- RT-LINUX
- VxWORKS
- LynxOS
Class 2: Extensions to Standard OSs

*Real-time extensions to standard OS:*
Attempt to exploit comfortable main stream OS.
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>
<tr>
<td>real-time kernel</td>
<td></td>
<td></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

revival of the concept: hypervisor
Example: Posix 1.0b RT-extensions to Linux

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

- RT-Task
- RT-Task
- Init
- Bash
- Mozilla

Special RT-calls and standard OS calls available.
Easy programming, no guarantee for meeting deadline
Example: RT Linux

RT-tasks cannot use standard OS calls.
Commercially available from fsmlabs (www.fsmlabs.com)

Diagram:
- Init
- Bash
- Mozilla
- Linux-Kernel
  - scheduler
  - driver
- RT-Task
- RT-Tasks
- RT-Linux
- RT-Scheduler
- Hardware
- I/O
- Interrupts
Class 3: Research Systems

*Research systems trying to avoid limitations:*
- Include L4, seL4, NICTA, ERIKA, SHARK

*Research issues:*
- low overhead memory protection,
- temporal protection of computing resources
- RTOSes for on-chip multiprocessors
- quality of service (QoS) control (besides real-time constraints)
- formally verified kernel properties

List of current real-time operating systems: