Konfigurierbare Systemsoftware (KSS)

VL 6 – Generative Programming: The SLOTH Approach

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 12 – 2012-06-27

http://www4.informatik.uni-erlangen.de/Lehre/SS12/V_KSS

Implementation Techniques: Classification

Decompositional Approaches

- Text-based filtering (untyped)
- Preprocessors

Compositional Approaches

- Language-based composition mechanisms (typed)
- OOP, AOP, Templates

Generative Approaches

- Metamodel-based generation of components (typed)
- MDD, C++ TMP, generators

About this Lecture

I'd rather write programs to write programs than write programs.

Dick Sites (DEC)
The OSEK Family of Automotive OS Standards

- **1995** OSEK OS (OSEK/VDX) [6]
- **2001** OSEKtime (OSEK/VDX) [8]
- **2005** AUTOSAR OS (AUTOSAR) [1]

**OSEK OS**
- 
  > "Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen"

- statically configured, event-triggered real-time OS

**OSEKtime**
- 
  > "Automotive Open System Architecture"

- statically configured, time-triggered real-time OS

- can optionally be extended with OSEK OS (to run in slack time)

**AUTOSAR OS**
- 
  > "Automotive Open System Architecture"

- statically configured, event-triggered real-time OS

- real superset of OSEK OS \(\leadsto\) backwards compatible

- additional time-triggered abstractions (schedule tables, timing protection)

- intended as successor for both, OSEK OS and OSEKtime
OSEK OS: Abstractions [6] (Cont’d)

- Coordination and synchronization
- Resource: mutual exclusion between well-defined set of tasks
  - stack-based priority ceiling protocol ([9]):
    - GetResource() \(\rightarrow\) priority is raised to that of highest participating task
    - pre-defined RES_SCHED has highest priority (\(\rightarrow\) blocks preemption)
    - implementation-optimal: task set may also include cat 2 ISRs
- Event: condition variable on which ETs may block
  - part of a task’s context
- Alarm: asynchronous trigger by HW/SW counter
  - may execute a callback, activate a task or set an event on expiry

OSEK OS: Conformance Classes [6]

- OSEK offers predefined tailorability by four conformance classes
  - BCC1: only basic tasks, limited to one activation request per task and one task per priority, while all tasks have different priorities
  - BCC2: like BCC1, plus more than one task per priority possible and multiple requesting of task activation allowed
  - ECC1: like BCC1, plus extended tasks
  - ECC2: like ECC1, plus more than one task per priority possible and multiple requesting of task activation allowed for basic tasks
- The OSEK feature diagram

OSEK OS: System Services (Excerpt)

- Task-related services
  - ActivateTask(task): \(\rightarrow\) task is active (\(\rightarrow\) ready), counted
  - TerminateTask(): \(\rightarrow\) running task is terminated
  - Schedule(): \(\rightarrow\) active task with highest priority is running
  - ChainTask(task): \(\rightarrow\) atomic \{ ActivateTask(task); TerminateTask() \}
- Resource-related services
  - GetResource(res): \(\rightarrow\) current task has res ceiling priority
  - ReleaseResource(res): \(\rightarrow\) current task has previous priority
- Event-related services (extended tasks only!)
  - SetEvent(task, mask): \(\rightarrow\) events mask for task are set
  - ClearEvent(mask): \(\rightarrow\) events mask for current task are unset
  - WaitEvent(mask): \(\rightarrow\) current task blocks, until event from mask has been set
- Alarm-related services
  - SetAbsAlarm(alarm, ...): \(\rightarrow\) arms alarm with absolute offset
  - SetRelAlarm(alarm, ...): \(\rightarrow\) arms alarm with relative offset

OSEK OS: System Specification with OIL [7]

- An OSEK OS instance is configured completely statically
  - all general OS features (Hooks, ...)
  - all instances of OS abstractions (Tasks, ...)
  - all relationships between OS abstractions
  - described in a domain-specific language (DSL)
- OIL: The OSEK Interface Language [7]
  - standard types and attributes (TASK, ISR, ...)
  - vendor/platform-specific attributes (ISR source, priority, triggering)
  - task types and conformance class is deduced

OIL File for Example System (BCC1)

- Three basic tasks: Task1, Task3, Task4
- Category 2 ISR: ISR2 (platform-specific source/priority)
- Task1 and Task3 use resource Res1 - ceiling pn = 3
- Alarm Alarm1 triggers Task4 on expiry
OSEK OS: System Generation [7, p. 5]

- User’s source code
- Compiler
- Linker
- Executable file
- Files produced by SG
- C code
- Make tool
- Object libraries
- OSEK COM
- OSEK Builder
- OSEK components, tools & related files
- User written/defined
- Third party tools & related files

OSEK OS: Example Control Flow

- Basic tasks behave much like IRQ handlers (on a system with support for IRQ priority levels)
  - priority-based dispatching with run-to-completion
  - LIFO, all control flows can be executed on a single shared stack
- So why not dispatch tasks as ISRs?
  
  ~ Let the hardware do all scheduling!
  ~ Let’s be a SLOTH!

Agenda

6.1 Motivation: OSEK and Co
6.2 SLOTH: Threads as Interrupts
  - Basic Idea
  - Design
  - Results
  - Limitation
6.3 SLEEPY SLOTH: Threads as IRQs as Threads
6.4 Outlook: SLOTH ON TIME
6.5 Summary and Conclusions
6.6 References

“SLOTH: Threads as Interrupts” [3]

- Idea: threads are interrupt handlers, synchronous thread activation is IRQ
  
  Let interrupt subsystem do the scheduling and dispatching work
  - Applicable to priority-based real-time systems
  - Advantage: small, fast kernel with unified control-flow abstraction

**SLOTH Design**

- IRQ system must support priorities and software triggering

![Diagram of IRQ system and task stack]

**SLOTH: Qualitative Results**

- Concise kernel design and implementation
  - < 200 LoC, < 700 bytes code memory, very little RAM
- Single control-flow abstraction for tasks, ISRs (1/2), callbacks
  - Handling oblivious to how it was triggered (by hardware or software)
- Unified priority space for tasks and ISRs
  - no rate-monotonic priority inversion [2]
- Straight-forward synchronization by altering CPU priority
  - Resources with ceiling priority (also for ISRs!)
  - Non-preemptive sections with `RES_SCHEDULER` (highest task priority)
  - Kernel synchronization with highest task/cat.-2-ISR priority

**Performance Evaluation: Methodology**

- Reference implementation for Infineon TriCore
  - 32-bit load/store architecture
  - Interrupt controller: 256 priority levels, about 200 IRQ sources with memory-mapped registers
  - Meanwhile also implementations for ARM Cortex M3 (SAM3) and x86
- Evaluation of task-related system calls:
  - Task activation
  - Task termination
  - Task acquiring/releasing resource
- Comparison with commercial OSEK implementation and CiAO
- Two numbers for SLOTH: best case, worst case
  - Depending on number of tasks and system frequency
Performance Evaluation: Results

![Graph showing performance evaluation results]

### Limitations of the SLOTH Approach

- No extended tasks (that is, events, $\rightarrow$ OSEK ECC1 / ECC2) $\leftrightarrow$ impossible with stack-based IRQ execution model
- No multiple tasks per priority (that is, OSEK BCC2 / ECC2) $\rightarrow$ execution order has to be the same as activation order

### Agenda

- 6.1 Motivation: OSEK and Co
- 6.2 SLOTH: Threads as Interrupts
- 6.3 SLEEPY SLOTH: Threads as IRQs as Threads
- 6.4 Outlook: SLOTH ON TIME
- 6.5 Summary and Conclusions
- 6.6 References
## Control Flows in Embedded Systems

<table>
<thead>
<tr>
<th></th>
<th>Activation Event</th>
<th>Sched./Disp.</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISRs</td>
<td>HW</td>
<td>by HW</td>
<td>RTC</td>
</tr>
<tr>
<td>Threads</td>
<td>SW</td>
<td>by OS</td>
<td>Blocking</td>
</tr>
<tr>
<td>SLOTH [3]</td>
<td>HW or SW</td>
<td>by HW</td>
<td>RTC</td>
</tr>
<tr>
<td>SLEEPY SLOTH  [4]</td>
<td>HW or SW</td>
<td>by HW</td>
<td>RTC or Blocking</td>
</tr>
</tbody>
</table>

(RTC: Run-to-Completion)

### SLEEPY SLOTH: Main Goal and Challenge

**Main Goal**
Support extended blocking tasks (with stacks of their own), while preserving SLOTH's latency benefits by having threads run as ISRs

**Main Challenge**
IRQ controllers do not support suspension and re-activation of ISRs

---

### SLEEPY SLOTH Design: Task Prologues and Stacks

**Hardware Periphery**
- Hardware IRQ
- Alarm Exp.
- Timer System

**Activation Event**
- Activate(Task1)
- HW IRQ
- Task prologue: switch stacks if necessary
- Switch basic task → basic task omits stack switch
- On job start: initialize stack
- On job resume: restore stack

**Task termination**
- Task with next-highest priority needs to run
- Yield CPU by setting priority to zero
- (Prologue of next task performs the stack switch)

**Task blocking**
- Take task out of "ready list"
- Disable task's IRQ source
- Yield CPU by setting priority to zero

**Task unblocking**
- Re-enable task's IRQ source
- Re-trigger task's IRQ source by setting its pending bit

---

### SLEEPY SLOTH: Dispatching and Rescheduling

- Task prologue: switch stacks if necessary
- Switch basic task → basic task omits stack switch
- On job start: initialize stack
- On job resume: restore stack

- Task termination: task with next-highest priority needs to run
  - Yield CPU by setting priority to zero
  - (Prologue of next task performs the stack switch)

- Task blocking: take task out of "ready list"
  - Disable task's IRQ source
  - Yield CPU by setting priority to zero

- Task unblocking: put task back into "ready list"
  - Re-enable task's IRQ source
  - Re-trigger task's IRQ source by setting its pending bit
**SLEEPY SLOTH: Example Control Flow**

- CPU/Task Priority
- Task Priority:
  - Task BT1
    - act(ET3)
    - Prologue ET3
    - save(stk bt)
    - init(stk et3)
  - Task ET3
    - block()
    - Prologue BT1
    - save(stk et3)
    - load(stk bt)
  - Task BT2
    - act(BT2)
    - Prologue BT2
    - nop
  - Task ET3 (ctd.)
    - unblock(ET3)
    - Prologue ET3
    - save(stk bt)
    - load(stk et3)

**SLEEPY SLOTH: Evaluation**

- Reference implementation on Infineon TriCore microcontroller
- Measurements: system call latencies in 3 system configurations, compared to a leading commercial OSEK implementation
  1. Only basic run-to-completion tasks
  2. Only extended blocking tasks
  3. Both basic and extended tasks

**Evaluation: Only Basic Tasks**

- Speed-Up: 2.0 4.6 19.0 4.3 3.7 8.0 7.4
- Average Speed-Up: 7x

**Evaluation: Only Extended Tasks**

- Speed-Up: 2.4 1.6 1.7 5.3 3.4 3.5
- Average Speed-Up: 3x

- **SLEEPY SLOTH outperforms commercial kernel with SW scheduler**
- **SLEEPY SLOTH as fast as original SLOTH**
- **Still faster than commercial kernel with SW scheduler**
- **SLEEPY SLOTH: Extended switches slower than basic switches**
Evaluation: Extended and Basic Tasks

- Basic switches in a mixed system only slightly slower than in purely basic system

SLOTH* Generation
- Two generation dimensions
  - Architecture
  - Application
- Generator is implemented in Perl
  - templates
  - configuration

SLOTH ON TIME: Time-Triggered Laziness
- Idea: user hardware timer arrays to implement schedule tables
- TC1796 GPTA: 256 timer cells, routable to 96 interrupt sources
  - use for task activation, deadline monitoring, execution time budgeting
- SLOTH ON TIME implements OSEKtime [8] and AUTOSAR OS schedule tables [1]
  - combinable with SLOTH or SLEEPYSLOTH for mixed-mode systems
  - up to 170x lower latencies compared to commercial implementations
Summary: The SLOTH* Approach

- **Exploit standard IR/timer hardware to delegate core OS functionality to hardware**
  - scheduling and dispatching of control flows
  - OS needs to be tailored to application and hardware platform
  - generative approach is necessary

- **Benefits**
  - tremendous latency reductions, very low memory footprints
  - unified control flow abstraction
  - hardware/software-triggered, blocking/run-to-completion
  - no need to distinguish between tasks and ISRs
  - reduces complexity
  - less work for the OS developer :-) We are

Referenzen (Cont’d)


