Konfigurierbare Systemsoftware (KSS)

VL 5 – Generative Programming: The SLOTH Approach

Daniel Lohmann
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität
Erlangen-Nürnberg
SS 16 – 2016-05-09
http://www4.informatik.uni-erlangen.de/Lehre/SS16/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

“Ì’d rather write programs to write programs than write programs.”
Dick Sites (DEC)
Agenda

5.1 Motivation: OSEK and Co
5.2 SLOTH: Threads as Interrupts
5.3 SLEEPY SLOTH: Threads as IRQs as Threads
5.4 SAFER SLOTH: Hardware-Tailored Isolation
5.5 SLOTH ON TIME: Time-Triggered Laziness
5.6 SLOTH+ Generation
5.7 Summary and Conclusions
5.8 References

The OSEK Family of Automotive OS Standards

- **1995** OSEK OS (OSEK/VDX) [9]
- **2001** OSEKtime (OSEK/VDX) [11]
- **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

- 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 a successor for both OSEK OS and OSEKtime

OSEK OS: Abstractions [9]

Control flows

- **Task**: software-triggered control flow (strictly priority-based scheduling)
  - Basic Task (BT) run-to-completion task with strictly stack-based activation and termination
  - Extended Task (ET) may suspend and resume execution (\( \mapsto \) coroutine)

- **ISR**: hardware-triggered control flow (hardware-defined scheduling)
  - Cat 1 ISR (ISR1) runs below the kernel, may not invoke system services (\( \mapsto \) prologue without epilogue)
  - Cat 2 ISR (ISR2) synchronized with kernel, may invoke system services (\( \mapsto \) epilogue without prologue)

- **Hook**: OS-triggered signal/exception handler
  - ErrorHook invoked in case of a syscall error
  - StartupHook invoked at system boot time
  - ...
OSEK OS: Abstractions [9] (Cont’d)

- Coordination and synchronization
  - Resource: mutual exclusion between well-defined set of tasks
    - stack-based priority ceiling protocol ([12]):
      - GetResource() \(\Rightarrow\) priority is raised to that of highest participating task
    - pre-defined RES_SCHED has highest priority (\(\Rightarrow\) blocks preemption)
    - implementation-optional: 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 [9]

- 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) \(\sim\) task is active (\(\Rightarrow\) ready), counted
  - TerminateTask() \(\sim\) running task is terminated
  - Schedule() \(\sim\) active task with highest priority is running
  - ChainTask(task) \(\Rightarrow\) atomic
    - ActivateTask(task)
    - TerminateTask()

- Resource-related services
  - GetResource(res) \(\sim\) current task has res ceiling priority
  - ReleaseResource(res) \(\sim\) current task has previous priority

- Event-related services (extended tasks only!)
  - SetEvent(task, mask) \(\sim\) events in mask for task are set
  - ClearEvent(mask) \(\sim\) events in mask for current task are unset
  - WaitForEvent(mask) \(\sim\) current task blocks until event from mask has been set

- Alarm-related services
  - SetAbsAlarm(alarm, ...) \(\sim\) arms alarm with absolute offset
  - SetRelAlarm(alarm, ...) \(\sim\) arms alarm with relative offset

OSEK OS: System Specification with OIL [10]

- 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 Implementation Language
  - standard types and attributes (TASK, ISR, ...)
  - vendor/plattform-specific attributes (ISR source, priority, triggering)
  - task types and conformance class is deduced

OS ExampleOS {
  STATUS = STANDARD; STARTUPHOOK = TRUE;
}

TASK Task1 {
  PRIORITY = 1; AUTOSTART = TRUE; RESOURCE = Res1;
}

TASK Task2 {
  PRIORITY = 3; AUTOSTART = FALSE; RESOURCE = Res1;
}

RESOURCE Res1 {
  RESOURCEPROPERTY = STANDARD;
}

ISR ISR2 {
  CATEGORY = 3; PRIORITY = 4;
}

ALARM Alarm1 {
  ACTION = ACTIVATE_TASK {
    TASK = Task4;
    AUTOSTART = FALSE;
  }
}

OSEK OS: Abstractions [9] (Cont’d)

- Coordination and synchronization
  - Resource: mutual exclusion between well-defined set of tasks
    - stack-based priority ceiling protocol ([12]):
      - GetResource() \(\Rightarrow\) priority is raised to that of highest participating task
    - pre-defined RES_SCHED has highest priority (\(\Rightarrow\) blocks preemption)
    - implementation-optional: 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: System Generation [10, p. 5]

- Files produced by SG
  - C code
  - Application configuration files (OIL)
- C code
- User's source code
- Optional OSEK Builder
- Compiler
- Linker
- Executable file
- OSEK Builder
- OSEK components, tools & related files
- Third party tools & related files
- Make tool
- Object libraries
- User written/defined

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

5.1 Motivation: OSEK and Co
5.2 SLOTH: Threads as Interrupts
  - Basic Idea
  - Design
  - Results
  - Limitation
5.3 SLEEPY SLOTH: Threads as IRQs as Threads
5.4 SAFER SLOTH: Hardware-Tailored Isolation
5.5 SLOTH ON TIME: Time-Triggered Laziness
5.6 SLOTH+ Generation
5.7 Summary and Conclusions
5.8 References

“SLOTH: Threads as Interrupts” [6]

- 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

**SLOTH: Example Control-Flow**

- CPU Prio Level
  - t0
  - t1
  - t2
  - t3
  - t4
  - t5
  - t6
  - t7
  - t8
  - t9
  - t10
  - t11
  - t12

**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 (SAM3U) 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

**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 [3, 4]

- 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: Results

![Graph showing performance evaluation results for different functions and dispatch types.](image)

- Speed-Up
  - Comm. OSEK: $\approx 2x$
  - SLOTH best case: $\approx 4x$
  - SLOTH worst case: $\approx 20x$
  - CIO: $\approx 3x$
  - Commercial OSEK: $\approx 8x$

Limitations of the SLOTH Approach

- No multiple tasks per priority (OSEK BCC2 / ECC2)
- Execution order has to be the same as activation order
- No extended tasks (that is, events, OSEK ECC1 / ECC2)
- Impossible with stack-based IRQ execution model
- No safety (that is, AUTOSAR-OS memory protection)
- Impossible if everything runs as IRQ handler

Agenda

- 5.1 Motivation: OSEK and Co
- 5.2 SLOTH: Threads as Interrupts
- 5.3 SLEEPY SLOTH: Threads as IRQs as Threads
  - Motivation
  - Design
  - Results
- 5.4 SAFER SLOTH: Hardware-Tailored Isolation
- 5.5 SLOTH ON TIME: Time-Triggered Laziness
- 5.6 SLOTH+ Generation
- 5.7 Summary and Conclusions
- 5.8 References

Control Flows in Embedded Systems

- **ISRs**
  - Activation Event: HW
  - Sched./Disp.: by HW
  - Semantics: RTC
- **Threads**
  - Activation Event: SW
  - Sched./Disp.: by OS
  - Semantics: Blocking
- **SLOTH [6]**
  - Activation Event: HW or SW
  - Sched./Disp.: by HW
  - Semantics: RTC
- **SLEEPY SLOTH [7]**
  - Activation Event: HW or SW
  - Sched./Disp.: by HW
  - Semantics: RTC or Blocking

(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: 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 Design: Task Prologues and Stacks**

**SLEEPY SLOTH: Example Control Flow**
Evaluation: Extended and Basic Tasks

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Act() BT → BT</th>
<th>Act() ET → BT</th>
<th>Block() BT → ET</th>
<th>Unblock() ET → BT</th>
<th>Term() BT → BT</th>
<th>Term() ET → BT</th>
<th>Term() ET → ET</th>
<th>Basic() BT → BT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
</tr>
</tbody>
</table>

Speed-Up: 3.6 2.5 1.3 1.7 9.7 3.7 3.3 4.0

Average Speed-Up: 4x

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

Limitations of the SLOTH Approach

- No multiple tasks per priority (→ OSEK BCC2 / ECC2)
- Execution order has to be the same as activation order
- No extended tasks (that is, events, → OSEK ECC1 / ECC2)
- Impossible with stack-based IRQ execution model
- No safety (that is, AUTOSAR-OS memory protection)
- Impossible if everything runs as IRQ handler

Agenda

5.1 Motivation: OSEK and Co
5.2 SLOTH: Threads as Interrupts
5.3 SLEEPY SLOTH: Threads as IRQs as Threads
5.4 SAFER SLOTH: Hardware-Tailored Isolation
  - Motivation
  - Design
  - Results
5.5 SLOTH ON TIME: Time-Triggered Laziness
5.6 SLOTH+ Generation
5.7 Summary and Conclusions
5.8 References

SAFER SLOTH: Main Goal and Challenge

Main Goal
Support isolation of state and privileges, while preserving SLOTH’s latency benefits by having threads run as ISRs.

Main Challenge
ISRs run in supervisor mode with full privileges. So how to
- Effectively isolate kernel and application
- Maintain design principles of Sloth
Memory Protection in Embedded Systems

- Safety, but not security
- Protect the data, but not the code
- Safety model based on AUTOSAR OS
- MPU-based isolation

Vertically: Protect kernel state and MPU configuration
Horizontally: Isolate applications or even tasks from each other

Protection Modes in SAFER SLOTH

Unsafe
- The original Sloth OS, without isolation

MPU
- MPU active, but tasks execute with supervisor privileges
- Vertical isolation ensured constructively in post-validation

MPU+traps
- Vertical isolation ensured by hardware privilege levels
- System services acquire kernel privileges via syscall mechanism

SAFER SLOTH: Architecture

- Exploit as much knowledge about target hardware as possible
- Tailor kernel to fit both the platform and the application
- Taking into account:
  - Extent and layout of MPU configuration
  - Method for re-programming the MPU
  - Available hardware privilege levels
  - Is MPU active in all levels?
  - Degree of safety required by the application
The Problem with Traps

- SLOTH gains a lot of its benefits through compiler optimizations
  - Inlining of system service calls
  - Removal of dead code
  - Constant propagation
- Traditional traps prohibit such optimizations
  - System services must be standalone functions
  - Jumped to via a syscall dispatcher

Solution Idea

Combine MPU and MPU+traps mode

~ Inline traps as 4th protection mode (MPU+itraps)
Evaluation Results: Total Overheads

**Safer Sloth vs. commercial AUTOSAR OS**

<table>
<thead>
<tr>
<th>Event</th>
<th>Safer Sloth (unsafe mode)</th>
<th>AUTOSAR OS (unsafe)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activate</td>
<td>120</td>
<td>160</td>
</tr>
<tr>
<td>Activate w/o disp.</td>
<td>100</td>
<td>140</td>
</tr>
<tr>
<td>Terminate w/o disp.</td>
<td>80</td>
<td>120</td>
</tr>
<tr>
<td>ChainTask w/o disp.</td>
<td>60</td>
<td>100</td>
</tr>
<tr>
<td>GetResource w/o disp.</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>ReleaseRes w/o disp.</td>
<td>20</td>
<td>60</td>
</tr>
</tbody>
</table>

Evaluation Results: Additional Overheads

**Safer Sloth vs. commercial AUTOSAR OS**

<table>
<thead>
<tr>
<th>Event</th>
<th>Safer Sloth (unsafe mode)</th>
<th>AUTOSAR OS (unsafe)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activate</td>
<td>120</td>
<td>160</td>
</tr>
<tr>
<td>Activate w/o disp.</td>
<td>100</td>
<td>140</td>
</tr>
<tr>
<td>Terminate w/o disp.</td>
<td>80</td>
<td>120</td>
</tr>
<tr>
<td>ChainTask w/o disp.</td>
<td>60</td>
<td>100</td>
</tr>
<tr>
<td>GetResource w/o disp.</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>ReleaseRes w/o disp.</td>
<td>20</td>
<td>60</td>
</tr>
</tbody>
</table>

Limitations of the SLOTH Approach

- No multiple tasks per priority (OSEK BCC2 / ECC2)
  - Execution order has to be the same as activation order
- No extended tasks (that is, events, OSEK ECC1 / ECC2)
  - Impossible with stack-based IRQ execution model
- No safety (that is, AUTOSAR OS memory protection)
  - Impossible if everything runs as IRQ handler

Agenda

- 5.1 Motivation: OSEK and Co
- 5.2 SLOTH: Threads as Interrupts
- 5.3 SLEEPY SLOTH: Threads as IRQs as Threads
- 5.4 SAFER SLOTH: Hardware-Tailored Isolation
- 5.5 SLOTH ON TIME: Time-Triggered Laziness
- 5.6 SLOTH*: Generation
- 5.7 Summary and Conclusions
- 5.8 References
SLOTH ON TIME: Time-Triggered Laziness

- Idea: use 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, time synchronization, and schedule table control
- SLOTH ON TIME implements OSEKtime [11] and AUTOSAR OS schedule tables [1]
  - combinable with SLOTH or SLEEPY SLOTH for mixed-mode systems
  - up to 170x lower latencies compared to commercial implementations

Qualitative Evaluation: AUTOSAR

Commercial AUTOSAR: Priority inversion with time-triggered activation (2,075 cycles each)

SLOTH ON TIME: avoids this by design!

Interrupts are perhaps the biggest cause of priority inversion in real-time systems, causing the system to not meet all of its timing requirements.


Agenda

5.1 Motivation: OSEK and Co
5.2 SLOTH: Threads as Interrupts
5.3 SLEEPY SLOTH: Threads as IRQs as Threads
5.4 SAFER SLOTH: Hardware-Tailored Isolation
5.5 SLOTH ON TIME: Time-Triggered Laziness
5.6 SLOTH* Generation
5.7 Summary and Conclusions
5.8 References

SLOTH* Generation

- Two generation dimensions
  - Architecture
  - Application
- Generator is implemented in Perl
- Templates
- Configuration
**SLOTH ON TIME Generation**

**Summary: The SLOTH+ Approach**

- Exploit standard interrupt/timer/mpu 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
    - no rate-monotonic priority inversion
    - reduces complexity
  - less work for the OS developer :-)

**Agenda**

5.1 Motivation: OSEK and Co
5.2 SLOTH: Threads as Interrupts
5.3 SLEEPY SLOTH: Threads as IRQs as Threads
5.4 SAFER SLOTH: Hardware-Tailored Isolation
5.5 SLOTH ON TIME: Time-Triggered Laziness
5.6 SLOTH+ Generation
5.7 Summary and Conclusions
5.8 References

**Referenzen**


