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
About this Lecture

Problem Space
- Domain Expert
- Features and Dependencies

Solution Space
- Architect / Developer
- Architecture and Implementation

System User
- Configuration
- Specific Problem
- Specific Solution

Features and Dependencies
- Architect / Developer
- Class
- Aspect

System User
- Variant
- System User

intentional side
- intended properties

extensional side
- actual implementation
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
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

“\[I’d \ rather \ write \ programs \ to \ write \ programs \ than \ write \ programs.\]”
Dick Sites (DEC)
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
6.1 Motivation: OSEK and Co

Background
OSEK OS: Abstractions
OSEK OS: Tailoring and Generation

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
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

- **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

- **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

$\leftrightarrow$ “Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen”

$\leftrightarrow$ “Automotive Open System Architecture”
OSEK OS: Abstractions [6]

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 (→ coroutine)

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

- **Hook**: OS-triggered signal/exception handler
  - ErrorHook: invoked in case of a syscall error
  - StartupHook: invoked at system boot time
  - ...
Coordination and synchronization

- **Resource**: mutual exclusion between well-defined set of tasks
  - stack-based priority ceiling protocol ([9]):
    - `GetResource()` \(\leadsto\) priority is raised to that of highest participating task
  - pre-defined `RES_SCHED` has highest priority (\(\leadsto\) 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 services (excerpt)

- task-related services
  - activate task \((task)\) \(\leadsto\) task is active (\(\mapsto\) ready), counted
  - terminate task \(()\) \(\leadsto\) running task is terminated
  - schedule \(()\) \(\leadsto\) active task with highest priority is running
  - chain task \((task)\) \(\mapsto\) atomic \(\begin{cases} activate task \((task)\) \\ terminate task \(()\) \end{cases}\)

- resource-related services
  - get resource \((res)\) \(\leadsto\) current task has res ceiling priority
  - release resource \((res)\) \(\leadsto\) current task has previous priority

- event-related services (extended tasks only!)
  - set event \((task, mask)\) \(\leadsto\) events in mask for task are set
  - clear event \((mask)\) \(\leadsto\) events in mask for current task are unset
  - wait event \((mask)\) \(\leadsto\) current task blocks until event from mask has been set

- alarm-related services
  - set abs alarm \((alarm, \ldots)\) \(\leadsto\) arms alarm with absolute offset
  - set rel alarm \((alarm, \ldots)\) \(\leadsto\) arms alarm with relative offset
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
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

**OIL File for Example System (BCC1)**
- Three basic tasks: **Task1**, **Task3**, **Task4**
- Category 2 ISR: **ISR2** (platform-spec. source/priority)
- **Task1** and **Task3** use resource **Res1** $\rightarrow$ ceiling pri = 3
- Alarm **Alarm1** triggers **Task4** on expiry

```plaintext
OS ExampleOS {
    STATUS = STANDARD;
    STARTUPHOOK = TRUE;
};
TASK Task1 {
    PRIORITY = 1;
    AUTOSTART = TRUE;
    RESOURCE = Res1;
};
TASK Task3 {
    PRIORITY = 3;
    AUTOSTART = FALSE;
    RESOURCE = Res1;
};
TASK Task4 {
    PRIORITY = 4;
    AUTOSTART = FALSE;
};
RESOURCE Res1 {
    RESOURCEPROPERTY = STANDARD;
};
ISR ISR2 {
    CATEGORY = 2;
    PRIORITY = 2;
};
ALARM Alarm1 {
    COUNTER = Timer1;
    ACTION = ACTIVATETASK {
        TASK = Task4;
    }; 
    AUTOSTART = FALSE;
};
```
OSEK OS: System Generation [7, p. 5]
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
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
_IRQ system must support priorities and software triggering

activate(Task1)

prio=1
request

IRQ Source
Task1

prio=2
request

IRQ Source
ISR2

prio=3
request

IRQ Source
Task3

prio=4
request

IRQ Source
Task4

Hardware Periphery

HW IRQ

Timer System

Alarm Exp.

curprio=X

IRQ Vector Table

task1()
isr2()
task3()
task4()
SLOTH: Example Control-Flow

CPU Prio Level

init()

enable()

getRes(Res1)

Task1

ISR2

RelRes(Res1)

Task1

SetAlarm(Al1)

iret

Task1

Term()

Alarm1

Task4

Act(Task1)

Task1

Term()

idle()

Task1

getRes(Res1)

Task1

 ISR2


**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 (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
Performance Evaluation: Results

Cycles

<table>
<thead>
<tr>
<th>Function</th>
<th>Speedup</th>
<th>SLOTH best case</th>
<th>SLOTH worst case</th>
<th>Commercial OSEK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activate()</td>
<td>≈ 2x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activate() w/ dispatch</td>
<td>≈ 4x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terminate()</td>
<td>≈ 20x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chain()</td>
<td>≈ 5x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetRes()</td>
<td>≈ 3x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReleaseRes()</td>
<td>≈ 8x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReleaseRes() w/ dispatch</td>
<td>≈ 8x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Performance Evaluation: Comparison with CiAO

<table>
<thead>
<tr>
<th></th>
<th>Act() w/o dispatch</th>
<th>Act() w/ dispatch</th>
<th>Term() w/ dispatch</th>
<th>Chain() w/ dispatch</th>
<th>GetRes() w/o dispatch</th>
<th>RelRes() w/o dispatch</th>
<th>RelRes() w/ dispatch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SLOTH best case</strong></td>
<td>34</td>
<td>60</td>
<td>14</td>
<td>67</td>
<td>19</td>
<td>14</td>
<td>36</td>
</tr>
<tr>
<td><strong>SLOTH worst case</strong></td>
<td>48</td>
<td>74</td>
<td>14</td>
<td>81</td>
<td>19</td>
<td>14</td>
<td>36</td>
</tr>
<tr>
<td><strong>CiAO</strong></td>
<td>75</td>
<td>206</td>
<td>107</td>
<td>139</td>
<td>19</td>
<td>66</td>
<td>204</td>
</tr>
</tbody>
</table>
Limitations of the SLOTH Approach

- No extended tasks (that is, events, $\mapsto$ OSEK ECC1 / ECC2)
  $\mapsto$ impossible with stack-based IRQ execution model

- No multiple tasks per priority ($\mapsto$ OSEK BCC2 / ECC2)
  $\mapsto$ execution order has to be the same as activation order
6.1 Motivation: OSEK and Co
6.2 SLOTH: Threads as Interrupts
6.3 SLEEPY SLOTH: Threads as IRQs as Threads
   Motivation
   Design
   Results
   SLOTH* Generation
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 Threads</td>
<td>HW</td>
<td>by HW</td>
<td>RTC</td>
</tr>
<tr>
<td><strong>SLOTH [3]</strong></td>
<td>SW</td>
<td>by OS</td>
<td>Blocking</td>
</tr>
<tr>
<td><strong>SLEEPY SLOTH [4]</strong></td>
<td>HW or SW</td>
<td>by HW</td>
<td>RTC</td>
</tr>
<tr>
<td></td>
<td>HW or SW</td>
<td>by HW</td>
<td>RTC or Blocking</td>
</tr>
</tbody>
</table>

(RTC: Run-to-Completion)
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**

 activates (Task1) 

 **Hardware Periphery** 

 **HW IRQ** 

 **Alarm Exp.** 

 **Timer System** 

 **Task Stack** 

 **Stack ET1** 

 **Stack ET4** 

 **IRQ Source ExtTask1** 

 **prio=1** req IE 

 **IRQ Source ISR2** 

 **prio=2** request 

 **IRQ Source Task3** 

 **prio=3** request 

 **IRQ Source ExtTask4** 

 **prio=4** req IE 

 **IRQ Vector Table** 

 **curprio=X** 

 **activate(Task1)** 

 **IRQ Vector Table** 

 **Prol1()** 

 **Task1()** 

 **isr2()** 

 **Prol3()** 

 **Task3()** 

 **Prol4()** 

 **Task4()**
**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 BT1**
  - `act(ET3)`
  - Prologue ET3: `save(stk bt) init(stk et3)`
  - Task ET3
  - Prologue BT1: `save(stk et3) load(stk bt)`
  - Task BT1 (ctd.)
    - `act(BT2)`
    - Prologue BT2: `nop`
    - Task BT2
    - Prologue ET3: `save(stk bt) load(stk et3)`
    - Task ET3 (ctd.)
      - `unblock(ET3)`

**IRQ Source**

- **Task1** `prio=1` `request`
- **Task2** `prio=2` `request`
- **ExtTask3** `prio=3` `req IE`

**IRQ Arbitration Unit**

- `curprio=3`

**IRQ Vector Table**

- `pro11()` `task1()`
- `pro12()` `task2()`
- `pro13()` `task3()`

**Basic Stack**

- Stack ET3

---

© dl  KSS (VL 6 | SS 12)  6 The SLOTH Approach | 6.3 SLEEPY SLOTH: Threads as IRQs as Threads 6–28
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

Average Speed-Up: 7x

**SLEEPY SLOTH** outperforms commercial kernel with SW scheduler
**SLEEPY SLOTH** as fast as original SLOTH
Evaluation: Only Extended Tasks

Average Speed-Up: 3x

Still faster than commercial kernel with SW scheduler

**SLEEPY SLOTH:** Extended switches slower than basic switches
### Evaluation: Extended and Basic Tasks

![Bar chart showing cycles and speed-up for different operations.]

**Cycles**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sleepy Sloth</th>
<th>Commercial OSEK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Act() BT → BT</td>
<td>3.6</td>
<td></td>
</tr>
<tr>
<td>Act() BT → ET</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>Block() ET → BT</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>Unblock() BT → ET</td>
<td>1.7</td>
<td></td>
</tr>
<tr>
<td>Term() BT → BT</td>
<td>9.7</td>
<td></td>
</tr>
<tr>
<td>Term() ET → ET stack switch</td>
<td>3.7</td>
<td></td>
</tr>
<tr>
<td>Term() BT → BT stack switch</td>
<td>3.3</td>
<td></td>
</tr>
<tr>
<td>Chain() BT → BT</td>
<td>4.0</td>
<td></td>
</tr>
</tbody>
</table>

**Average Speed-Up:** 4x

- **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
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
**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 [8] 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

---

**Diagram:**

- Dispatcher round length
- Task1
- Task2
- idle

**Graph:**

- $t$: time
- 0, 200, 400, 600, 800, 1000, 1200

---

© dl KSS (VL 6 | SS 12) 6 The SLOTH Approach | 6.4 Outlook: SLOTH ON TIME 6–35
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
The SLOTH* Approach

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

We are sloth


