VL 6 – Generative Programming: The SLOTH Approach

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 13 – 2013-06-13

http://www4.informatik.uni-erlangen.de/Lehre/SS13/V_KSS
About this Lecture

Problem Space

Domain Expert

Features and Dependencies

Solution Space

Architect / Developer

Architecture and Implementation

Features and Dependencies

Class

Aspect

System User

Variant

Specific Problem

intentional side

intended properties

System User

Specific Solution

extensional side

actual implementation

instance level

model level
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
  - OOP, AOP, Templates

Generative Approaches
- Metamodell-based generation of components (typed)
- MDD, C++ TMP, generators

“\textit{I’d rather write programs to write programs than write programs.}”

Dick Sites (DEC)
6.1 Motivation: OSEK and Co
6.2 SLOTH: Threads as Interrupts
6.3 SLEEPY SLOTH: Threads as IRQs as Threads
6.4 SLOTH ON TIME: Time-Triggered Laziness
6.5 SLOTH* Generation
6.6 Summary and Conclusions
6.7 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 SLOTH ON TIME: Time-Triggered Laziness
6.5 SLOTH* Generation
6.6 Summary and Conclusions
6.7 References
The OSEK Family of Automotive OS Standards

- **1995** OSEK OS (OSEK/VDX) [8]
- **2001** OSEKtime (OSEK/VDX) [10]
- **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 ~ backwards compatible
- additional time-triggered abstractions (schedule tables, timing protection)
- intended as a successor for both OSEK OS and OSEKtime
OSEK OS: Abstractions [8]

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 ([11]):
    - `GetResource()` → priority is raised to that of highest participating task
  - pre-defined `RES_SCHED` has highest priority (→ 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**
  - `ActivateTask(task)` → `task` is active (↔ ready), counted
  - `TerminateTask()` → running task is terminated
  - `Schedule()` → active task with highest priority is running
  - `ChainTask(task)` → atomic
    - `{ ActivateTask(task) TerminateTask() }

- **Resource-related services**
  - `GetResource(res)` → current task has `res` ceiling priority
  - `ReleaseResource(res)` → current task has previous priority

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

- **Alarm-related services**
  - `SetAbsAlarm(alarm, ...)` → arms `alarm` with absolute offset
  - `SetRelAlarm(alarm, ...)` → arms `alarm` with relative offset
OSEK OS: Conformance Classes [8]

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:

- **Control Flows**
  - ISRs Cat. 2
  - ISRs Cat. 1
    - Kernel Sync
    - Full Preemption
    - Mixed Preemption
    - No Preemption

- **Alarms**
  - Tasks
    - Activate Task
    - Set Event
      - ECC1, ECC2
    - Exec Callback
  - Multiple Tasks Per Prio
    - BCC2, ECC2
    - Multiple Activations
      - BCC2, ECC2

- **Coordination**
  - Resources
    - BCC2, ECC1, ECC2
  - Events
    - ECC1, ECC2
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 ➔ 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 [9, p. 5]

Optional OSEK Builder

Application configuration files (OIL)

System Generator (SG)

Files produced by SG

Compiler

Linker

Executable file

C code

User’s source code

OSEK OS Kernel

OSEK COM

Make tool

Third party tools & related files

OSEK components, tools & related files

User written/defined

© dl  KSS (VL 6 | SS 13)  6 The TL;DR Approach | 6.1 Motivation: OSEK and Co
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!
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 SLOTH ON TIME: Time-Triggered Laziness

6.5 SLOTH* Generation

6.6 Summary and Conclusions

6.7 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
**SLOTH: Example Control-Flow**

![Diagram showing control-flow with CPU priority levels and tasks]

- **init()**
- **enable()**
- ** ISR2**
- **GetRes(Res1)**
- **RelRes(Res1)**
- **SetAlarm(Al1)**
- **iret**
- **Term()**
- **Task1**
- **Task4**
- **Alarm1**
- **Act(Task1)**
- **idle()**

**CPU Prio Level**

0 1 2 3 4

**t**

\(_t_1\) \(_t_2\) \(_t_3\) \(_t_4\) \(_t_5\) \(_t_6\) \(_t_7\) \(_t_8\) \(_t_9\) \(_t_{10}\)
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, 3]

- 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

![Bar chart showing performance evaluation results for various functions with and without dispatch]

- **Activate()**:
  - With dispatch: \( \approx 2x \)
  - Without dispatch: \( \approx 4x \)

- **Terminate()**:
  - \( \approx 20x \)

- **Chain()**:
  - \( \approx 5x \)

- **GetRes()**:
  - \( \approx 3x \)

- **ReleaseRes()**:
  - \( \approx 8x \)

- **Speed-Up**:
  - Commercial OSEK: \( \approx 8x \)
  - SLOTH best case: \( \approx 2x \)
  - SLOTH worst case: \( \approx 4x \)
  - CiAO: \( \approx 20x \)

© dl KSS (VL 6 | SS 13) 6 The SLOTH Approach | 6.2 SLOTH: Threads as Interrupts
Limitations of the SLOTH Approach

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

- No multiple tasks per priority (\(\rightarrow\) OSEK BCC2 / ECC2) \(\leftarrow\) 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
   - Motivation
   - Design
   - Results
6.4 SLOTH ON TIME: Time-Triggered Laziness
6.5 SLOTH* Generation
6.6 Summary and Conclusions
6.7 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 [5]</td>
<td>HW or SW</td>
<td>by HW</td>
<td>RTC</td>
</tr>
<tr>
<td>SLEEPY SLOTH [6]</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**

- **activate(Task1)**
- **HW IRQ**
- **Timer System**
- **IRQ Source ExtTask1**
  - **prio=1**
  - **request**
  - **IE**
- **IRQ Source ISR2**
  - **prio=2**
  - **request**
- **IRQ Source Task3**
  - **prio=3**
  - **request**
- **IRQ Source ExtTask4**
  - **prio=4**
  - **request**
  - **IE**

**IRQ Arbritration Unit**

- **CPU**
  - **curprio=X**
- **IRQ Vector Table**
  - **pro1()**
  - **task1()**
  - **isr2()**
  - **pro3()**
  - **task3()**
  - **pro4()**
  - **task4()**

**Task Stack**

**Stack ET1**

**Stack ET4**

**Hardware Periphery**

**Timer System**

**Alarm Exp.**
**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

1. Task BT1, act(ET3)
2. Prologue ET3, Task ET3
3. Prologue BT1, Task BT1 (ctd.)
4. Prologue BT2, Task BT2
5. Prologue ET3, Task ET3 (ctd.)
6. block()
7. Prologue ET3, Task ET3 (ctd.)
8.ᴜnblock(ET3)
9. save(stk bt)
10. load(stk bt)

**IRQ Source**
- Task1, prio=1, request
- Task2, prio=2, request
- ExtTask3, prio=3, req IE

**Stacks**
- Basic Stack
- Stack ET3

**CPU**
- curprio=3

**IRQ Table**
- pro11() \(\rightarrow\) task1()
- pro12() \(\rightarrow\) task2()
- pro13() \(\rightarrow\) task3()
**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

<table>
<thead>
<tr>
<th>Basic Switches</th>
<th>BT → BT</th>
<th>BT → ET</th>
<th>ET → BT</th>
<th>BT → ET</th>
<th>BT → BT</th>
<th>ET → ET</th>
<th>BT → BT</th>
<th>BT → BT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Act()</strong></td>
<td>3.6</td>
<td>2.5</td>
<td>1.3</td>
<td>1.7</td>
<td>9.7</td>
<td>3.7</td>
<td>3.3</td>
<td>4.0</td>
</tr>
<tr>
<td><strong>Term()</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Chain()</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Average Speed-Up:** 4x

Basic switches in a mixed system only slightly slower than in purely basic system.
6.1 Motivation: OSEK and Co
6.2 SLOTH: Threads as Interrupts
6.3 SLEEPY SLOTH: Threads as IRQs as Threads
6.4 SLOTH ON TIME: Time-Triggered Laziness
6.5 SLOTH* Generation
6.6 Summary and Conclusions
6.7 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 [10] 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)

<table>
<thead>
<tr>
<th>address</th>
<th>500us</th>
<th>725us</th>
<th>950us</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_TricoreIdle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTOR_STM_SRC0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task1Usercode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task2Usercode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task3Usercode</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SLOTH ON TIME:** avoids this by design!

<table>
<thead>
<tr>
<th>address</th>
<th>500us</th>
<th>725us</th>
<th>950us</th>
</tr>
</thead>
<tbody>
<tr>
<td>demo\app\idle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>emo\app\irq_1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>functionTask1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>emo\app\irq_2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>functionTask2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>emo\app\irq_3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>functionTask3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“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

6.1 Motivation: OSEK and Co
6.2 SLOTH: Threads as Interrupts
6.3 SLEEPY SLOTH: Threads as IRQs as Threads
6.4 SLOTH ON TIME: Time-Triggered Laziness
6.5 SLOTH* Generation
6.6 Summary and Conclusions
6.7 References
**SLOTH** Generation

- Two generation dimensions
  - Architecture
  - Application

Generator is implemented in Perl
- Templates
- Configuration
**Static Application Configuration:**

- `roundLength = 1000;`
- `expiryPoints = {`
  - `100 => Task1,`
  - `200 => Task2,`
  - `600 => Task1`;
- `deadlines = {`
  - `450 => Task1,`
  - `350 => Task2,`
  - `950 => Task1`;
- `availableTimerCells = {Cell7, ..., Cell12, Cell42};`

**Cell and IRQ Map:**

- `100 => Cell7 // Activation`
- `200 => Cell8 // Activation`
- `600 => Cell10 // Activation`
- `450 => Cell10 // Deadline`
- `350 => Cell11 // Deadline`
- `950 => Cell12 // Deadline`
- `Cell7 => IRQTask1`
- `Cell8 => IRQTask2`
- `Cell9 => IRQTask1`

**Cell Initialization Code:**

```c
void initCells(void) {
    Cell7.compare = 1000;
    ... Cell7.value = 1000 - 100;
    ...}
void startDispatcher(void) {
    #ifndef CONTROLCELLS
    Cell7.enable = 1;
    ... #else
    // Control Cell 42 for Cells 7-12
    Cell42.output = 1;
    #endif
}
```

**Timer Hardware Description:**

- `TimerArray0 = {
  Cell0 = {
    irqSource => 128,
    isMaster => false,
    controls => {},
  },
  ... Cell42 = {
    irqSource => 170,
    isMaster => true,
    controls => {7, ..., 12},
  },
  ...}
```

**Code Generation**

- `void handlerTask1(void) {
    // Prologue
    savePreemptedContext();
    setCPUPro(execPrio);
    Cell10.reqEnable = 1;
    Cell12.reqEnable = 1;
    userTask1();
    // Epilogue
    Cell10.reqEnable = 0;
    Cell12.reqEnable = 0;
    restorePreemptedContext();
    iret();
    }
```

**Input**

**Analysis and Cell Mapping**

**Intermediate**
Agenda

6.1 Motivation: OSEK and Co
6.2 SLOTH: Threads as Interrupts
6.3 SLEEPY SLOTH: Threads as IRQs as Threads
6.4 SLOTH ON TIME: Time-Triggered Laziness
6.5 SLOTH* Generation
6.6 Summary and Conclusions
6.7 References
Summary: 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
Referenzen


