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 13 – 2013-06-13
http://www4.informatik.uni-erlangen.de/Lehre/SS13/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

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

- 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: 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

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: System Specification with OIL [9]

- 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 {
  ID = 1; CATEGORY = 1; ACTION = ACTIVATETASK {
   TASK Task2 {
     ID = 2; CATEGORY = 2; AUTOSTART = FALSE;
   } 
   RESOURCE Res1 {
     CATEGORY = 1; RESOURCEPROPERTY = STANDARD;
   } ISR ISR1 {
     ID = 1; CATEGORY = 1; ACTION = ACTIVATETASK {
       TASK Task3 {
         ID = 3; CATEGORY = 3; AUTOSTART = TRUE;
       } RESOURCE Res1 {
         CATEGORY = 1; RESOURCEPROPERTY = STANDARD;
       } 
       ISR ISR2 {
         ID = 2; CATEGORY = 2; AUTOSTART = FALSE;
       } 
     } 
   } TASK Task4 {
     ID = 4; CATEGORY = 2; AUTOSTART = FALSE;
   } 
   RESOURCE Res1 {
     CATEGORY = 1; RESOURCEPROPERTY = STANDARD;
   } ISR ISR1 {
     ID = 1; CATEGORY = 1; ACTION = ACTIVATETASK {
       TASK Task3 {
         ID = 3; CATEGORY = 3; AUTOSTART = TRUE;
       } RESOURCE Res1 {
         CATEGORY = 1; RESOURCEPROPERTY = STANDARD;
       } ISR ISR2 {
         ID = 2; CATEGORY = 2; AUTOSTART = FALSE;
       } 
     } 
   } 
  } 
}
OSEK OS: System Generation [9, p. 5]

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

OSEK OS: Example Control Flow

Task Prio Level
t
0
1
2
3
4
ISR2
0
1
2
3
4
5
6
7
8
9
10
init()
StartOS()
Task1
GetRes(Res1)
Task1
E
ISR2
SetAlarm(Al1, t8)
iret
Task1
RelRes(Res1)
Task1
Term()
idle()
Task4
E
Alarm1
Act(Task1)
Term()
Task1

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!

“SLOTH: Threads as Interrupts” [5]

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 setup 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, 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

![Graph showing performance evaluation results](image)

**Limitations of the SLOTH Approach**

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

**Control Flows in Embedded Systems**

<table>
<thead>
<tr>
<th>ISRs</th>
<th>Activation Event</th>
<th>Sched./Disp.</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW</td>
<td>HW or SW</td>
<td>by HW</td>
<td>RTC</td>
</tr>
<tr>
<td>SW</td>
<td>SW</td>
<td>by OS</td>
<td>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: 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**

---
**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**

![Graph showing speed-up comparison between Sleepy Sloth and commercial OSEK]

- **Sleepy Sloth outperforms commercial kernel with SW scheduler**
- **Sleepy Sloth as fast as original Sloth**

**Evaluation: Only Extended Tasks**

![Graph showing speed-up comparison between Sleepy Sloth and commercial OSEK]

- Still faster than commercial kernel with SW scheduler
  - **Sleepy Sloth: Extended switches slower than basic switches**

**Evaluation: Extended and Basic Tasks**

![Graph showing speed-up comparison between Sleepy Sloth and commercial OSEK]

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

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

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

**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 :-)


