Sleepy Sloth: Threads as Interrupts as Threads

Wanja Hofer, Daniel Lohmann, Wolfgang Schröder-Preikschat

32nd IEEE Real-Time Systems Symposium
November 30, 2011
# My Personal Dilemma

## Motivation
Get a PhD in computer science (in operating systems group)

## Problem
Requires lots of work (build own operating system),

but I am a lazy sloth

## Solution
Let the hardware do the work!
## 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 [RTSS 09]</td>
<td>HW or SW</td>
<td>by HW</td>
<td>RTC</td>
</tr>
<tr>
<td>Sleepy Sloth</td>
<td>HW or SW</td>
<td>by HW</td>
<td>RTC or Blocking</td>
</tr>
</tbody>
</table>

(RTC: Run-to-Completion)
Talk Outline

1. Sloth Revisited
2. Sleepy Sloth Design and Implementation
3. Evaluation
4. Conclusions and Future Work
Platform must support IR priorities and software IR triggering
### 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) -> IRQ Source ExtTask1
  prio=1  req IE

Hardware Periphery

HW IRQ -> IRQ Source ISR2
  prio=2  request

Timer System

Alarm Exp. -> IRQ Source ExtTask4
  prio=4  req IE

CPU

curprio=X

IRQ Arbi-tration Unit

IRQ Vector Table

prol1() -> task1()
isr2()
prol3() -> task3()
prol4() -> task4()

Task Stack

Stack ET1

Stack ET4
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: Evaluation

- Reference implementation on Infineon TriCore microcontroller
- 32-bit load/store architecture
- Interrupt controller: 256 priority levels, about 200 IRQ sources with memory-mapped registers

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

Speed-Up

- Activate() w/ dispatch: 2.4
- Block() w/ dispatch: 1.6
- Unblock() w/ dispatch: 1.7
- ClearMask() w/ dispatch: 5.3
- Terminate() w/ dispatch: 3.4
- Chain() w/ dispatch: 3.5

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

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

Average Speed-Up: 4x

<table>
<thead>
<tr>
<th>Speed-Up</th>
<th>3.6</th>
<th>2.5</th>
<th>1.3</th>
<th>1.7</th>
<th>9.7</th>
<th>3.7</th>
<th>3.3</th>
<th>4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Act()</td>
<td>BT → BT</td>
<td>stack switch</td>
<td>Act()</td>
<td>BT → ET</td>
<td>stack switch</td>
<td>Block()</td>
<td>ET → BT</td>
<td>stack switch</td>
</tr>
<tr>
<td>Unblock()</td>
<td>BT → ET</td>
<td>stack switch</td>
<td>Term()</td>
<td>BT → BT</td>
<td>stack switch</td>
<td>Term()</td>
<td>ET → BT</td>
<td>stack switch</td>
</tr>
<tr>
<td>Term()</td>
<td>BT → BT</td>
<td>stack switch</td>
<td>Term()</td>
<td>BT → BT</td>
<td>stack switch</td>
<td>Term()</td>
<td>BT → BT</td>
<td>stack switch</td>
</tr>
<tr>
<td>Chain()</td>
<td>BT → BT</td>
<td>stack switch</td>
<td>Speed-Up</td>
<td>3.6</td>
<td>2.5</td>
<td>1.3</td>
<td>1.7</td>
<td>9.7</td>
</tr>
</tbody>
</table>
Evaluation: Summary

Main Goal

Support **extended blocking tasks** (with stacks of their own) while preserving Sloth’s **latency benefits** by having threads run as ISRs.
Conclusions and Future Work

- Sleepy Sloth threads, which are implemented as ISRs, 
  - ... can be run-to-completion or blocking  
    → flexible
  - ... can be activated by hardware or through a syscall  
    → flexible
  - ... are scheduled and dispatched by the IRQ controller  
    → fast (speed-up ≈ 2–5)
  - ... run in a single priority space, the IRQ priority space  
    → no priority inversion

- Future work: investigate suitability of the Sloth concept for...
  - ... integration with RTLinux/RTAI on x86-APIC
  - ... multi-core IRQ controllers: Pandaboard (dual-Cortex-A9), x86-APIC
  - ... time-triggered scheduling elements: AUTOSAR schedule tables