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

	Activation Event	Sched./Disp.	Semantics
ISRs	HW	by HW	RTC
Threads	SW	by OS	Blocking
Sloth [RTSS 09]	HW or SW	by HW	RTC
Sleepy Sloth	HW or SW	by HW	RTC or Blocking

(RTC: Run-to-Completion)

Talk Outline

1. Sloth Revisited
2. Sleepy Sloth Design and Implementation
3. Evaluation
4. Conclusions and Future Work

Sloth Revisited

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.

IRQ Arbi-tration Unit

CPU

curprio=X

IRQ Vector Table

task1()
isr2()
task3()
task4()

Task Stack

- Platform must support IR priorities and software IR triggering

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 Design: Task Prologues and Stacks

activate(Task1)

prio=1	IRQ Source ExtTask1
req IE	

Hardware Periphery → HW IRQ →

prio=2	IRQ Source ISR2
request	

prio=3	IRQ Source Task3
request	

Timer System → Alarm Exp. →

prio=4	IRQ Source ExtTask4
req IE	

IRQ Arbitration Unit

CPU

curprio=X

IRQ Vector Table

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

Task Stack

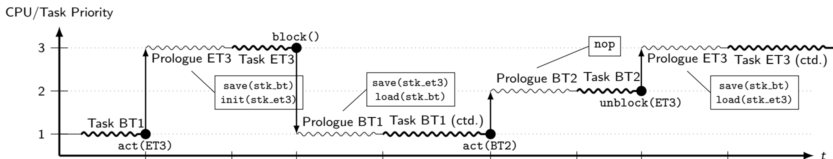
Stack ET1

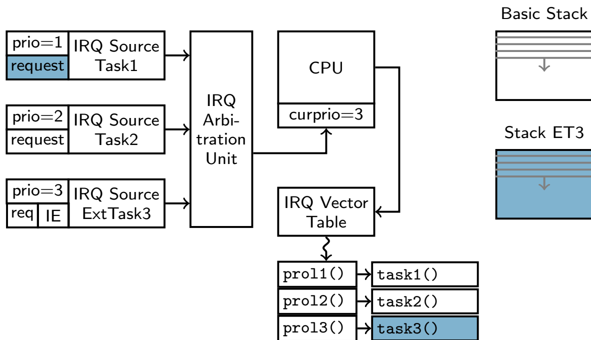
Stack ET4

Sleepy Sloth: Dispatching and Rescheduling

- Task prologue: switch stacks if necessary
  - Switch *basic task* $\rightarrow$ *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





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

Cycles

400
300
200
100
0

	Activate()	Activate() w/ dispatch	Terminate() w/ dispatch	Chain() w/ dispatch	GetRes()	ReleaseRes()	ReleaseRes() w/ dispatch
Speed-Up	2.0	4.6	19.0	4.9	3.7	8.0	7.4

Average Speed-Up: 7x

Sloth
Sleepy Sloth
Commercial OSEK

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

Evaluation: Only Extended Tasks

Cycles

400
300
200
100
0

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

Average Speed-Up: 3x

Sleepy Sloth
Commercial OSEK

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

Evaluation: Extended *and* Basic Tasks

Cycles

400
300
200
100
0

	Act()	Act()	Block()	Unblock()	Term()	Term()	Term()	Chain()
	BT → BT	BT → ET	ET → BT	BT → ET	BT → BT	ET → ET	BT → BT	BT → BT
		stack switch	stack switch	stack switch		stack switch	stack switch	
Speed-Up	3.6	2.5	1.3	1.7	9.7	3.7	3.3	4.0

Average Speed-Up: 4x

Sleepy Sloth
Commercial OSEK

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

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
    $\rightarrow$ flexible
  - ...can be activated by hardware or through a syscall
    $\rightarrow$ flexible
  - ...are scheduled and dispatched by the IRQ controller
    $\rightarrow$ fast (speed-up $\approx$ 2–5)
  - ...run in a single priority space, the IRQ priority space
    $\rightarrow$ 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