IRQs: the Hard, the Soft, the Threaded and the Preemptible

Alison Chaiken

Latest version of these slides
alison@she-devel.com

Embedded Linux Conference Europe
Oct 11, 2016

Example code

Version 2, actually presented live

Thursday October 13, 2016 15:30:

Debugging Methodologies for Realtime Issues
Joel Fernandes, Google
this same room

Knocking at Your Back Door (or How Dealing with Modern Interrupt Architectures can Affect Your Sanity)
Marc Zyngier, ARM Ltd
Hall Berlin A

Agenda

- Why do IRQs exist?
- About kinds of hard-IRQ handlers
- About softirqs and tasklets
- Differences in IRQ handling between RT and non-RT kernels
- Studying IRQ behavior via kprobes, event tracing, mpstat and eBPF
- Detailed example: when does NAPI take over for eth IRQs?

"Kunst nicht lehrbar ist.   Sie müssen wieder in der Werkstatt aufgehen." -- Walter Gropius

Sample questions to be answered

- What's all stuff in /proc/interrupts anyway?
- What are IPIs and NMIs?
- Why are atomic operations expensive for ARM?
- Why are differences between mainline and RT for softirqs?
- What is 'current' task while in softirq?
- What function is running inside the threaded IRQs?
- When do we switch from individual hard IRQ processing to NAPI?

Interrupt handling: a brief pictorial summary

one full life, http://tinyurl.com/j25lal5

Dennis Jarvis, http://tinyurl.com/jmkw23h

Top half: the hard IRQ

Bottom half: the soft IRQ

Why do we need interrupts at all?

- IRQs allow devices to notify the kernel that they require maintenance.
- Alternatives include
  - polling (servicing devices at a pre-configured interval);
  - traditional IPC to user-space drivers.
- Even a single-threaded RTOS or a bootloader needs a system timer.

Interrupts in Das U-boot

- For ARM, minimal IRQ support:
  - clear exceptions and reset timer (*e.g.*, arch/arm/lib/interrupts_64.c or arch/arm/cpu/armv8/exceptions.S)

- For x86, interrupts are serviced via a stack-push followed by a jump (arch/x86/cpu/interrupts.c)
  - PCI has full-service interrupt handling (arch/x86/cpu/irq.c)

Interrupts in RTOS: Xenomai/ADEOS IPIPE

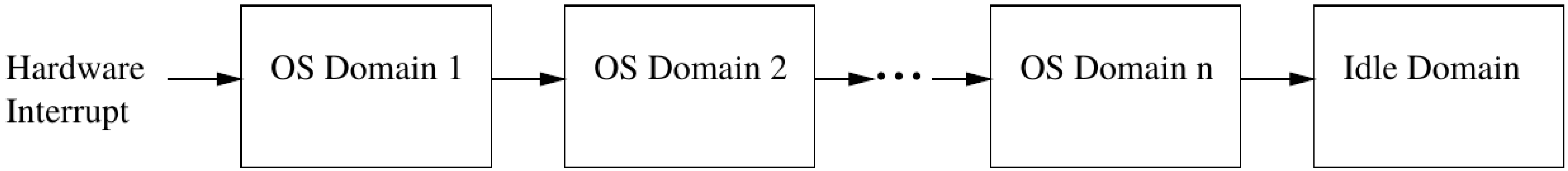


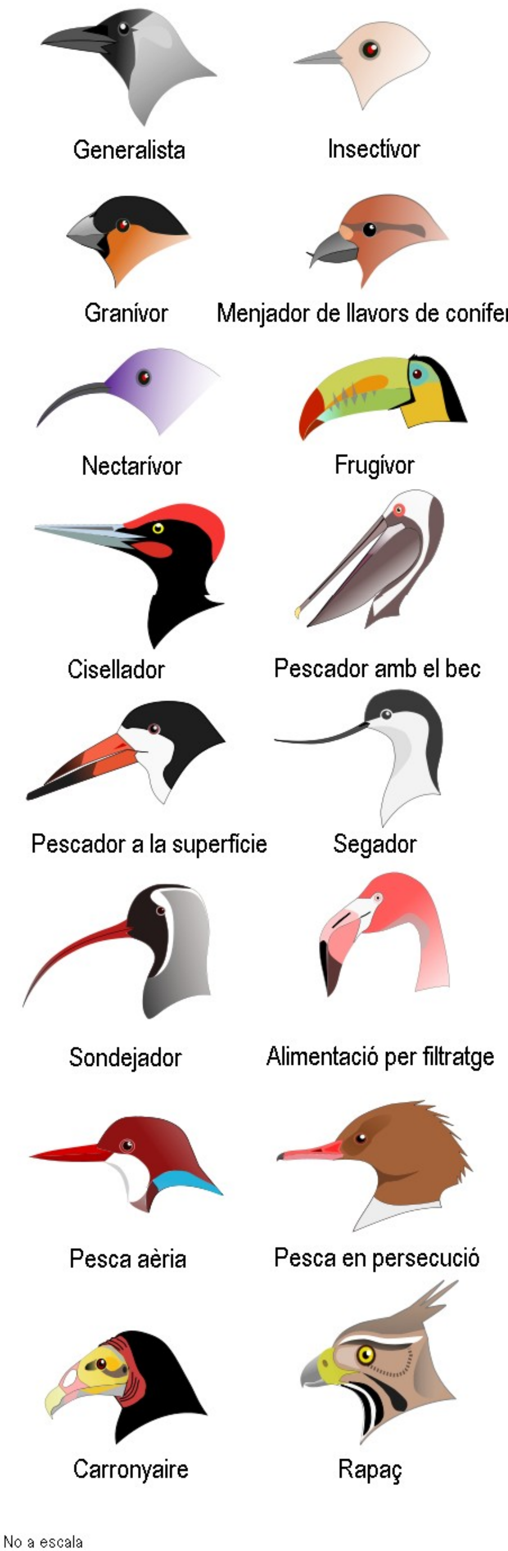
Figure 2: Adeos' interrupt pipe.

From Adeos website, covered by GFDL

Zoology of IRQs

- Hard versus soft
- Level- vs. edge-triggered, simple, fast EOI or per-CPU
- Local vs. global; System vs. device
- Maskable vs. non-maskable
- Shared or not; chained or not
- Multiple interrupt controllers per SOC

→ 'cat /proc/interrupts' or 'mpstat -A'



ARM IPIs, from arch/arm/kernel/smp.c

```
void handle_IPI(int ipinr, struct pt_regs *regs)
    switch (ipinr) {
    case IPI_TIMER:
        tick_receive_broadcast();
    case IPI_RESCHEDULE:
        scheduler_ipi();
    case IPI_CALL_FUNC:
        generic_smp_call_function_interrupt();
    case IPI_CPU_STOP:
        ipi_cpu_stop(cpu);
    case IPI_IRQ_WORK:
        irq_work_run();
    case IPI_COMPLETION:
        ipi_complete(cpu);
    }
```

Handlers are in kernel/sched/core.c

$ # cat /proc/interrupts, look at bottom

SKIP

What is an NMI?

- A 'non-maskable' interrupt is related to:
  - HW problem: parity error, bus error, watchdog timer expiration . . .
  - also used by perf

```
/* non-maskable interrupt control */
#define NMICR_NMIF      0x0001          /* NMI pin interrupt flag */
#define NMICR_WDIF            0x0002  /* watchdog timer overflow */
#define NMICR_ABUSERR         0x0008   /* async bus error flag */
```

From arch/arm/mn10300/include/asm/intctl-regs.h

SKIP

How IRQ masking works

only current core

arch/arm/include/asm/irqflags.h:
```
#define arch_local_irq_enable arch_local_irq_enable
static inline void arch_local_irq_enable(void)
{	asm volatile(
		"cpsie i			@ arch_local_irq_enable"
		::: "memory", "cc"); }
```

"change processor state"

arch/arm64/include/asm/irqflags.h:
```
static inline void arch_local_irq_enable(void)
{	asm volatile(
		"msr	daifclr, #2		// arch_local_irq_enable"
		::: "memory"); }
```

arch/x86/include/asm/irqflags.h:
```
static inline notrace void arch_local_irq_enable(void)
{	native_irq_enable(); }
static inline void native_irq_enable(void)
{	asm volatile("sti": : :"memory"); }
```

x86's Infamous System Management Interrupt

- SMI jumps out of kernel into System Management Mode
  - controlled by System Management Engine (Skochinsky)
- Identified as security vulnerability by Invisible Things Lab
- Not directly visible to Linux
- Traceable via hw_lat detector (sort of)

[RFC][PATCH 1/3] tracing: Added hardware latency tracer, Aug 4
From: "Steven Rostedt (Red Hat)" <rostedt@goodmis.org>
The hardware latency tracer has been in the PREEMPT_RT patch for some time. It is used to detect possible SMIs or any other hardware interruptions that the kernel is unaware of. Note, NMIs may also be detected, but that may be good to note as well.

ARM's Fast Interrupt reQuest

- An NMI with optimized handling due to dedicated registers.
- Underutilized by Linux drivers.
- Serves as the basis for Android's fiq_debugger.

SKIP

IRQ 'Domains' Correspond to Different INTC's

CONFIG_IRQ_DOMAIN_DEBUG:

This option will show the mapping relationship between hardware irq numbers and Linux irq numbers. The mapping is exposed via debugfs in the file "irq_domain_mapping".

Note:

- There are a lot more IRQs than in /proc/interrupts.
- There are more IRQs in /proc/interrupts than in 'ps axl | grep irq'.
- Some IRQs are not used.
- Some are processor-reserved and not kernel-managed.

Example: i.MX6 General Power Controller

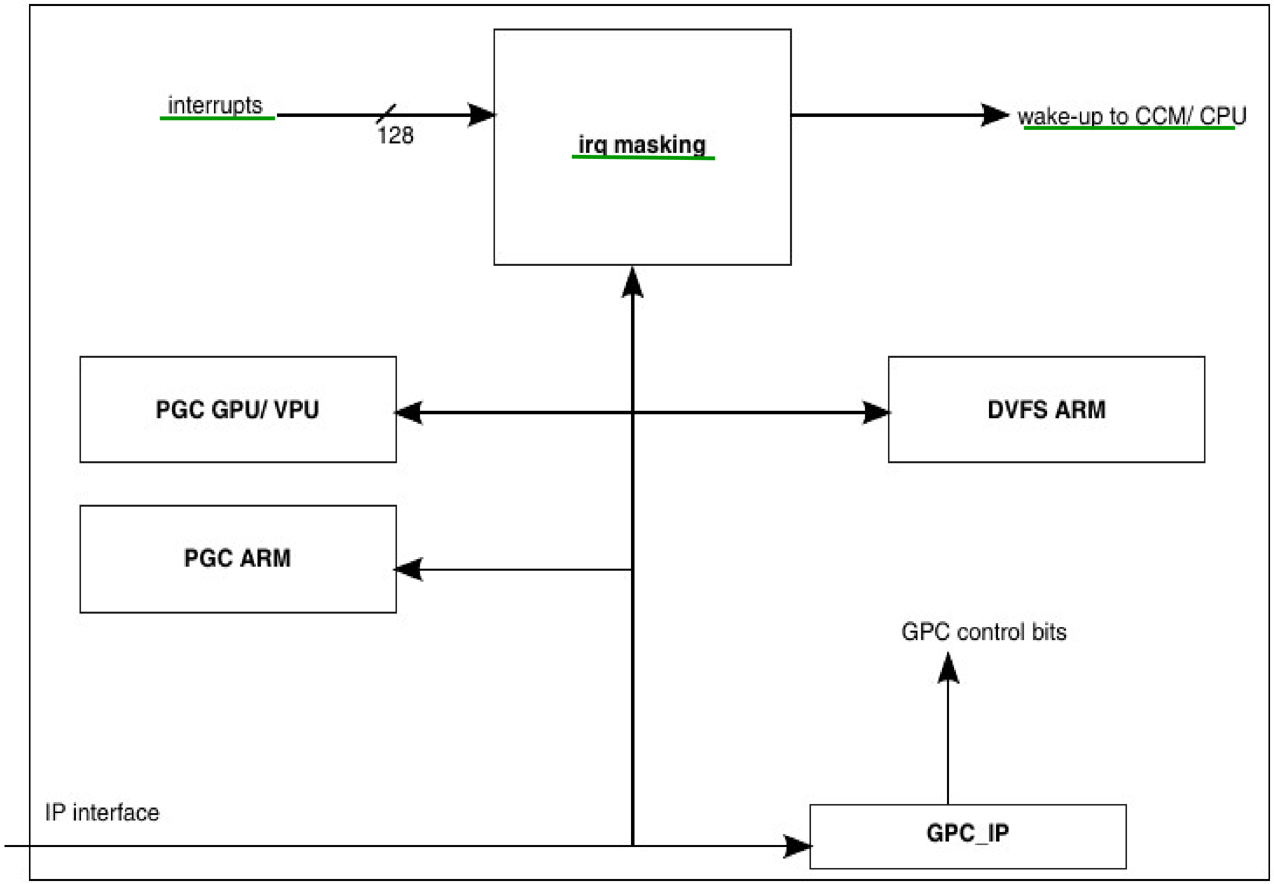


Figure 27-1. GPC Block Diagram

Unmasked IRQs can wakeup sleeping power domains.

Threaded IRQs in RT kernel

ps axl | grep irq

with both RT and non-RT kernels.

Handling IRQs as kernel threads allows priority and CPU affinity to be managed individually.

IRQ handlers running in threads can themselves be interrupted.

Quiz: What we will see with 'ps axl | grep irq' for non-RT kernels? Why?

What function do threaded IRQs run?

```
/* request_threaded_irq - allocate an interrupt line
 *   @handler: Function to be called when the IRQ occurs.
 *     Primary handler for threaded interrupts
 *     If NULL and thread_fn != NULL the default
 *     primary handler is installed
 *
 *   @thread_fn: Function called from the irq handler thread
 *       If NULL, no irq thread is created
*/
```

Even in mainline, request_irq() = requested_threaded_irq() with NULL thread_fn.

EXAMPLE

CASE 0 ***indirect*** invocation of request_threaded_irq()

request_irq(*handler*) → request_threaded_irq(*handler*, NULL)

Result:

-- irq_default_primary_handler() runs in interrupt context.

-- *All* it does is wake up the thread.

-- Then *handler* runs in irq/<name> thread.

CASE 1 ***direct*** invocation of request_threaded_irq()

Result:

-- *handler* runs in interrupt context.

-- *thread_fn* runs in irq/<name> thread.

irq_setup_forced_threading()

Threaded IRQs in RT, mainline and mainline with “threadirqs” boot param

- RT: all hard-IRQ handlers that don't set IRQF_NOTHREAD run in threads.

- Mainline: only those hard-IRQ handlers whose registration requests explicitly call request_threaded_irq() run in threads.

- Mainline with threadirqs kernel cmdline: like RT, but CPU affinity of IRQ threads cannot be set.

genirq: Force interrupt thread on RT

genirq: Do not invoke the affinity callback via a workqueue on RT

Shared interrupts: mmc driver

- Check 'ps axl | grep irq | grep mmc':

```
1     0   122     2 -51     0      -      S    ?          0:00 [irq/16-mmc0]
1     0   123     2 -50      0     -      S    ?          0:00 [irq/16-s-mmc0]
```

- 'cat /proc/interrupts': mmc and ehci-hcd share an IRQ line

```
16:        204        IR-IO-APIC  16-fasteoi   mmc0,ehci_hcd:usb3
```

- drivers/mmc/host/sdhci.c:

handler thread_fn

```
ret = request_threaded_irq(host->irq, sdhci_irq, sdhci_thread_irq,
IRQF_SHARED,mmc_hostname(mmc), host);
```

Why are atomic operations more expensive (ARM)?

arch/arm/include/asm/atomic.h:

```
static inline void atomic_##op(int i, atomic_t *v)      \
{ raw_local_irq_save(flags);    \
v->counter c_op i;         \
raw_local_irq_restore(flags);   }
```

include/linux/irqflags.h:

```
#define raw_local_irq_save(flags)      \
do {  flags = arch_local_irq_save();    } while (0)
```

arch/arm/include/asm/atomic.h:

```
/* Save the current interrupt enable state & disable IRQs */
static inline unsigned long arch_local_irq_save(void) { . . . }
```

Introduction to softirqs

Tasklet interface | Raised by devices | Kernel housekeeping

In kernel/softirq.c:

IRQ_POLL since 4.4

```
const char * const softirq_to_name[NR_SOFTIRQS] = {
  "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "BLOCK_IOPOLL",
      "TASKLET", "SCHED", "HRTIMER", "RCU"
};
```

Gone since 4.1

In ksoftirqd, softirqs are serviced in the listed order.

What are tasklets?

```
const char * const softirq_to_name[NR_SOFTIRQS] = {
 "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "BLOCK_IOPOLL",
       "TASKLET", "SCHED", "HRTIMER", "RCU"
};
```

- Tasklets perform deferred work not handled by other softirqs.
- Examples: crypto, USB, DMA, keyboard . . .
- More latency-sensitive drivers (sound, PCI) are part of *tasklet_hi_vec*.
- Any driver can create a tasklet.
- tasklet_hi_schedule() or tasklet_schedule() are called directly by ISR.

SKIP

```
[alison@sid ~]$ sudo mpstat -I SCPU
```

Linux **4.1.0-rt17**+ (sid) 05/29/2016 _**x86_64**_(4 CPU)

CPU	HI/s	TIMER/s	NET_TX/s	NET_RX/s	BLOCK/s	TASKLET/s	SCHED/s	HRTIMER/s	RCU/s
0	0.03	249.84	0.00	0.11	19.96	0.43	238.75	0.68	0.00
1	0.01	249.81	0.38	1.00	38.25	1.98	236.69	0.53	0.00
2	0.02	249.72	0.19	0.11	53.34	3.83	233.94	1.44	0.00
3	0.59	249.72	0.01	2.05	19.34	2.63	234.04	1.72	0.00

Linux **4.6.0**+ (sid) 05/29/2016 _**x86_64**_(4 CPU)

CPU	HI/s	TIMER/s	NET_TX/s	NET_RX/s	BLOCK/s	TASKLET/s	SCHED/s	HRTIMER/s	RCU/s
0	0.26	16.13	0.20	0.33	40.90	0.73	9.18	0.00	19.04
1	0.00	9.45	0.00	1.31	14.38	0.61	7.85	0.00	17.88
2	0.01	15.38	0.00	0.20	0.08	0.29	13.21	0.00	16.24
3	0.00	9.77	0.00	0.05	0.15	0.00	8.50	0.00	15.32

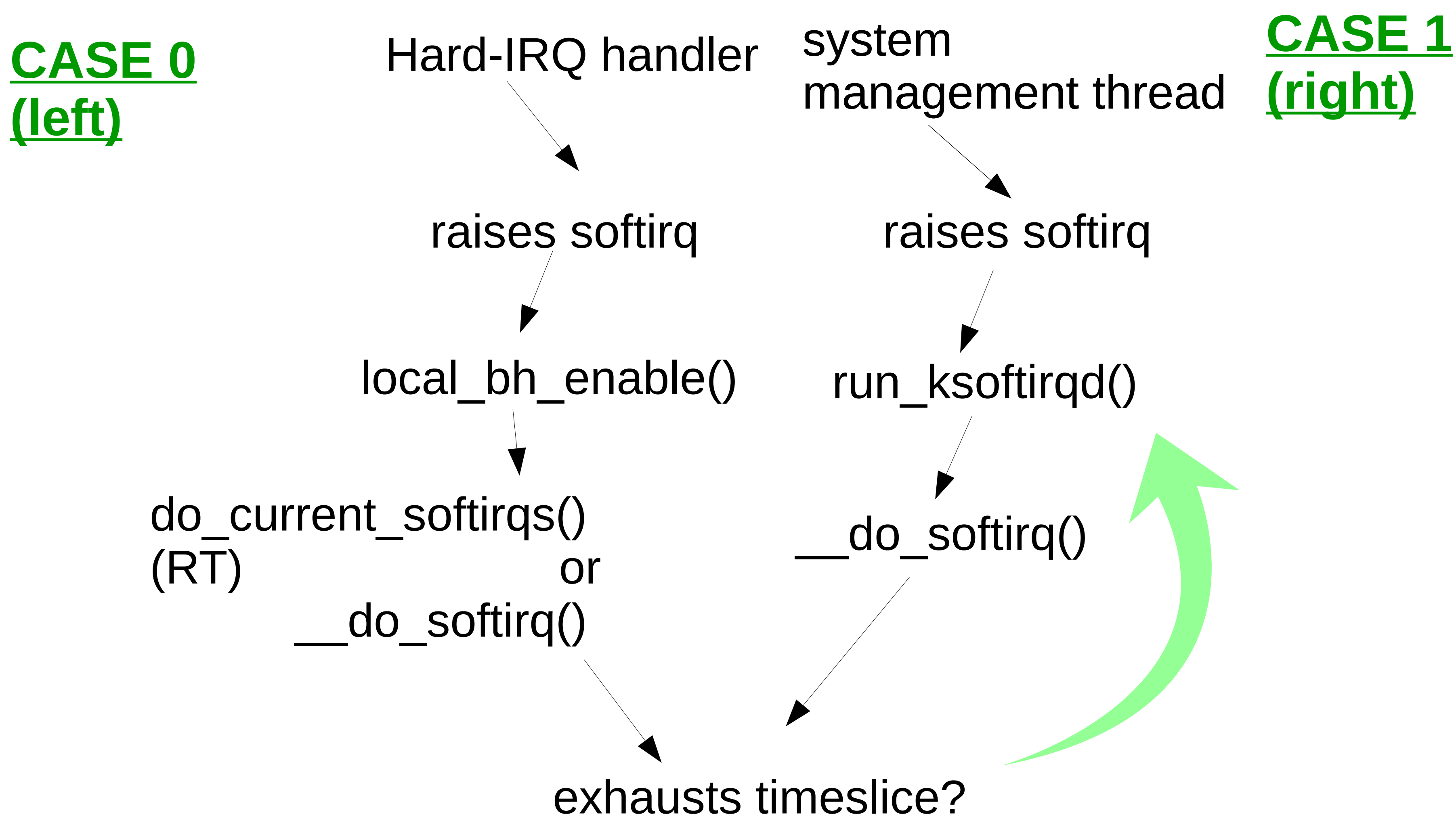
Linux **4.1.18-rt17**-00028-g8da2a20 (vpc23) 06/04/16 _**armv7l**_(2 CPU)

CPU	HI/s	TIMER/s	NET_TX/s	NET_RX/s	BLOCK/s	TASKLET/s	SCHED/s	HRTIMER/s	RCU/s
0	0.00	999.72	0.18	9.54	0.00	89.29	191.69	261.06	0.00
1	0.00	999.35	0.00	16.81	0.00	15.13	126.75	260.89	0.00

Linux **4.7.0** (nitrogen6x) 07/31/16 _**armv7l**_ (4 CPU)

CPU	HI/s	TIMER/s	NET_TX/s	NET_RX/s	BLOCK/s	TASKLET/s	SCHED/s	HRTIMER/s	RCU/s
0	0.00	2.84	0.50	40.69	0.00	0.38	2.78	0.00	3.03
1	0.00	89.00	0.00	0.00	0.00	0.00	0.64	0.00	46.22
2	0.00	16.59	0.00	0.00	0.00	0.00	0.23	0.00	3.05
3	0.00	10.22	0.00	0.00	0.00	0.00	0.25	0.00	1.45

Two paths by which softirqs run



Related demo and sample code

Case 0: Run softirqs at exit of a hard-IRQ handler

RT (4.6.2-rt5)

local_bh_enable();

↓

__local_bh_enable();

↓

do_***current***_softirqs();

↓

while (***current***->softirqs_raised) {
    i = __ffs(***current***->softirqs_raised);
    do_***single***_softirq(i);
}

↓

handle_softirq();

Run softirqs raised
*in the **current** context.*

non-RT (4.6.2)

local_bh_enable();

↓

do_softirq();

↓

__do_softirq();

↓

handle_***pending***_softirqs();

↓

while ((softirq_bit = ffs(***pending***)))
    handle_softirq();

*Run **all** pending softirqs up to*
MAX_IRQ_RESTART.

EXAMPLE

Case 1: Scheduler runs the rest from ksoftirqd

RT (4.6.2-rt5)

run_ksoftirqd();

↓

do_current_softirqs()
[where *current* == ksoftirqd]

non-RT (4.6.2)

run_ksoftirqd();

↓

do_softirq();

↓

__do_softirq();

↓

```
h = softirq_vec;
while ((softirq_bit = ffs(pending)))
{
        h += softirq_bit - 1;
        h->action(h);
}
```

RT vs Mainline: entering softirq handler

SKIP

4.7 **mainline**:

```
[11661.191187]  e1000e_poll+0x126/0xa70 [e1000e]
[11661.191197]  net_rx_action+0x52e/0xcd0
[11661.191206]  __do_softirq+0x15c/0x5ce          <- kick off soft IRQ
[11661.191215]  irq_exit+0xa3/0xd0                  }
[11661.191222]  do_IRQ+0x62/0x110                   } hard-IRQ handler
[11661.191230]  common_interrupt+0x82/0x82          }
```

4.6.2-**rt**5:

```
[ 6937.393805]  e1000e_poll+0x126/0xa70 [e1000e]
[ 6937.393808]  check_preemption_disabled+0xab/0x240
[ 6937.393815]  net_rx_action+0x53e/0xc90                 }
[ 6937.393824]  do_current_softirqs+0x488/0xc30           } kick-off softIRQ
[ 6937.393831]  do_current_softirqs+0x5/0xc30             }
[ 6937.393836]  __local_bh_enable+0xf2/0x1a0
[ 6937.393840]  irq_forced_thread_fn+0x91/0x140           }
[ 6937.393845]  irq_thread+0x170/0x310                    }
[ 6937.393848]  irq_finalize_oneshot.part.6+0x4f0/0x4f0   } hard-IRQ handler
[ 6937.393853]  irq_forced_thread_fn+0x140/0x140          }
[ 6937.393857]  irq_thread_check_affinity+0xa0/0xa0       }
[ 6937.393862]  kthread+0x12b/0x1b0
```

Summary of softirq execution paths

Case 0: Behavior of local_bh_enable() differs significantly between RT and mainline kernel.

Case 1: Behavior of ksoftirqd itself is *mostly* the same (note discussion of ktimersoftd below).

What is 'current'?

include/asm-generic/current.h:

```
#define get_current() (current_thread_info()->task)
#define current get_current()
```

arch/arm/include/asm/thread_info.h:

```
static inline struct thread_info *current_thread_info(void)
{ return (struct thread_info *)  (current_stack_pointer &
~(THREAD_SIZE - 1));
}
```

arch/x86/include/asm/thread_info.h:

```
static inline struct thread_info *current_thread_info(void)
{  return (struct thread_info *)(current_top_of_stack() -
THREAD_SIZE);}
```

In do_current_softirqs(), *current* is the threaded IRQ task.

SKIP

What is 'current'? part 2

arch/arm/include/asm/thread_info.h:

```
/*
 * how to get the current stack pointer in C
 */
register unsigned long current_stack_pointer asm ("sp");
```

arch/x86/include/asm/thread_info.h:

```
static inline unsigned long current_stack_pointer(void)
{
	unsigned long sp;
#ifdef CONFIG_X86_64
	asm("mov %%rsp,%0" : "=g" (sp));
#else
	asm("mov %%esp,%0" : "=g" (sp));
#endif
	return sp;
}
```

Q.: When do system-management softirqs get to run?

Introducing systemd-irqd!![†]

[†]As suggested by Dave Anders

Do timers, scheduler, RCU ever run as part of do_current_softirqs?

Examples:

-- every jiffy,
raise_softirq_irqoff(HRTIMER_SOFTIRQ);

-- scheduler_ipi() for NOHZ calls
raise_softirq_irqoff(SCHED_SOFTIRQ);

-- rcu_bh_qs() calls
raise_softirq(RCU_SOFTIRQ);

These run when ksoftirqd is *current*.

Demo: kprobe on do_current_softirqs() for RT kernel

- At Github
- Counts calls to *do_current_softirqs()* from ksoftirqd and from a hard-IRQ hander.
- Tested on 4.4.4-rt11 with Boundary Devices' Nitrogen i.MX6.

Output showing what task of 'current_thread' is:

```
[   52.841425] task->comm is ksoftirqd/1
[   70.051424] task->comm is ksoftirqd/1
[   70.171421] task->comm is ksoftirqd/1
[  105.981424] task->comm is ksoftirqd/1
[  165.260476] task->comm is irq/43-2188000.
[  165.261406] task->comm is ksoftirqd/1
[  225.321529] task->comm is irq/43-2188000.
```

explanation

Softirqs can be pre-empted with PREEMPT_RT

include/linux/sched.h:

```
struct task_struct {
#ifdef CONFIG_PREEMPT_RT_BASE
    struct rcu_head put_rcu;
    int softirq_nestcnt;
    unsigned int softirqs_raised;
#endif
};
```

RT-Linux headache: 'softirq starvation'

“sched: RT throttling activated” or
“INFO: rcu_sched detected stalls on CPUs”

- ksoftirqd scarcely gets to run.
- Events that are triggered by timer interrupt won't happen.
- *Example*: main event loop in userspace did not run due to missed timer ticks.

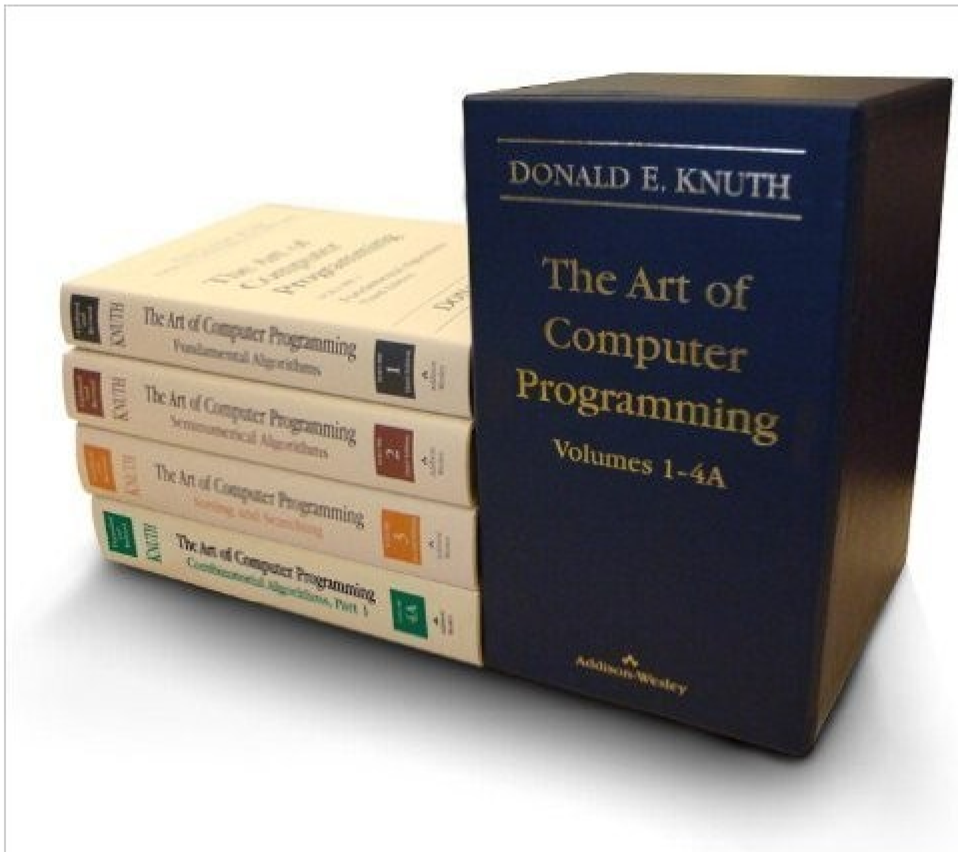
Reference: “Understanding a Real-Time System” by Rostedt, slides and video

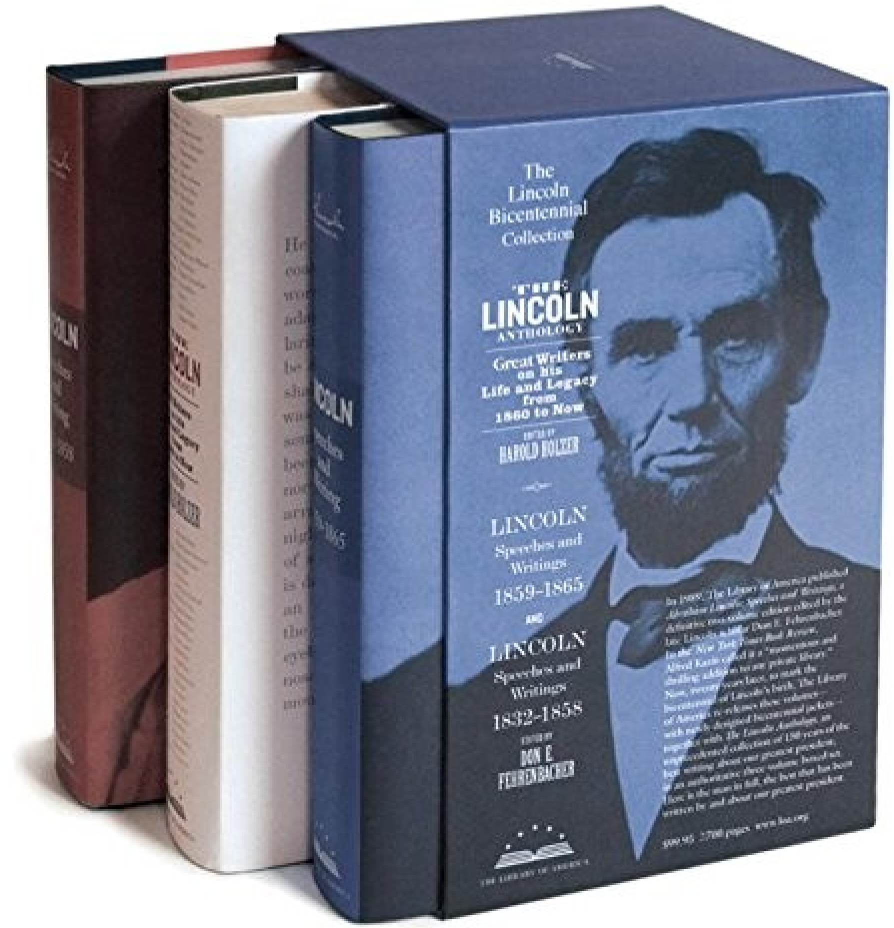
(partial) RT solution: ktimersoftd

Author: Sebastian Andrzej Siewior <bigeasy@linutronix.de>
Date: Wed Jan 20 2016 +0100
softirq: split timer softirqs out of ksoftirqd

With enough networking load it is possible that the system never goes idle and schedules ksoftirqd and everything else with a higher priority. One of the tasks left behind is one of RCU's threads and so we see stalls and eventually run out of memory. This patch moves the TIMER and HRTIMER softirqs out of the `ksoftirqd` thread into its own `ktimersoftd`. The former can now run SCHED_OTHER (same as mainline) and the latter at SCHED_FIFO due to the wakeups. [. . .]

ftrace produces a copious amount of output





Investigating IRQs with eBPF: bcc

- BCC - Tools for BPF-based Linux analysis
- tools/ and examples/ illustrate interfaces to kprobes and uprobes.
- BCC tools are:
  - a convenient way to study arbitrary infrequent events dynamically;
  - based on dynamic code insertion using Clang Rewriter JIT;
  - lightweight due to in-kernel data storage.

eBPF, IOvisor and IRQs: limitations

- JIT compiler is currently available for the x86-64, arm64, and s390 architectures.
- No stack traces unless CONFIG_FRAME_POINTER=y
- Requires recent kernel, LLVM and Clang
- bcc/src/cc/export/helpers.h:

```
#ifdef __powerpc__
[ . . . ]
#elif defined(__x86_64__)
[ . . . ]
#else
#error "bcc does not support this platform yet"
#endif
```

SKIP

bcc tips

- Kernel source must be present on the host where the probe runs.
- /lib/modules/$(uname -r)/build/include/generated must exist.
- To switch between kernel branches and continue quickly using bcc:
  - run 'mrproper; make config; make'
  - 'make' need only to populate include/generated in kernel source before bcc again becomes available.
  - 'make headers_install' as non-root user

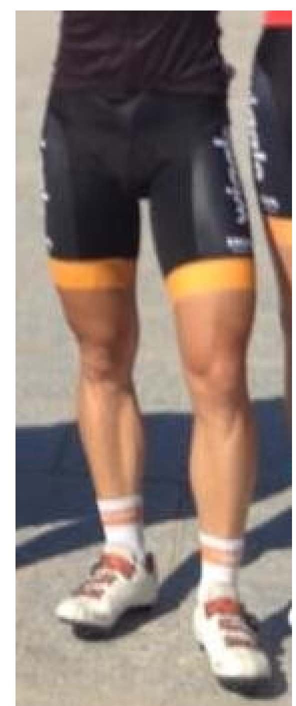
SKIP

Get latest version of clang by compiling from source
(or from Debian Sid)

```
$ git clone http://llvm.org/git/llvm.git
$ cd llvm/tools
$ git clone --depth 1 http://llvm.org/git/clang.git
$ cd ..; mkdir build; cd build
$ cmake .. -DLLVM_TARGETS_TO_BUILD="BPF;X86"
$ make -j $(getconf _NPROCESSORS_ONLN)
```

from samples/bpf/README.rst

Example: NAPI: changing the bottom half



Quick NAPI refresher

The problem:

“High-speed networking can create thousands of interrupts per second, all of which tell the system something it already knew: it has lots of packets to process.”

The solution:

“Interrupt mitigation . . . NAPI allows drivers to run with (some) interrupts disabled during times of high traffic, with a corresponding decrease in system load.”

The implementation:

Poll the driver and drop packets without processing in the NIC if the polling frequency necessitates.

Example: i.MX6 FEC RGMII NAPI turn-on

== irq_forced_thread_fn() for irq/43

```
static irqreturn_t fec_enet_interrupt(int irq, void *dev_id)
[ . . . ]
    if ((fep->work_tx || fep->work_rx) && fep->link) {
        if (napi_schedule_prep(&fep->napi)) {
            /* Disable the NAPI interrupts */
            writel(FEC_ENET_MII, fep->hwp + FEC_IMASK);
            __napi_schedule(&fep->napi);
        }
    }
```

Back to threaded IRQs

Example: i.MX6 FEC RGMII NAPI turn-off

```
static int fec_enet_rx_napi(struct napi_struct *napi, int budget){
[ . . . ]
    pkts = fec_enet_rx(ndev, budget);
    if (pkts < budget) {
        napi_complete(napi);
        writel(FEC_DEFAULT_IMASK, fep->hwp + FEC_IMASK);
    }
}
netif_napi_add(ndev, &fep->napi, fec_enet_rx_napi,
NAPI_POLL_WEIGHT);
```

Interrupts are re-enabled when budget is not consumed.

Using existing tracepoints

- function_graph tracing causes a lot of overhead.
- How about napi_poll tracer in /sys/kernel/debug/events/napi?
  - Fires constantly with any network traffic.
  - Displays no obvious change in behavior when eth IRQ is disabled and polling starts.

The Much Easier Way:

BCC on x86_64 with
4.6.2-rt5 and Clang-3.8

Handlind Eth IRQs in ksoftirqd on x86_64, but NAPI?

4.6.2-rt5

root $ ./stackcount.py e1000_receive_skb
Tracing 1 functions for "e1000_receive_skb"
^C

e1000_receive_skb
e1000e_poll
net_rx_action
do_current_softirqs
run_ksoftirqd
smpboot_thread_fn
kthread
ret_from_fork
1

COUNTS

e1000_receive_skb
e1000e_poll
net_rx_action
do_current_softirqs
__***local_bh_enable***
irq_forced_thread_fn
irq_thread
kthread
ret_from_fork
26469

running from ksoftirqd, not from hard IRQ handler.

Normal behavior: packet handler runs immediately after eth IRQ, in its context.

Switch to NAPI on x86_64

```
[alison@sid]$ sudo modprobe kp_ksoft eth_irq_procid=1
[  ] __raise_softirq_irqoff_ksoft: 582 hits
[  ] kprobe at ffffffff81100920 unregistered

[alison@sid]$ sudo ./stacksnoop.py __raise_softirq_irqoff_ksoft
144.803096056     __raise_softirq_irqoff_ksoft
    ffffffff81100921 __raise_softirq_irqoff_ksoft
    ffffffff810feda9  do_current_softirqs
    ffffffff810ffeae   run_ksoftirqd
    ffffffff8114d255 smpboot_thread_fn
    ffffffff81144a99 kthread
    ffffffff8205ed82 ret_from_fork
```

SKIP

Same Experiment, but non-RT 4.6.2

Most frequent:

```
e1000_receive_skb
 e1000e_poll
 net_rx_action
 __softirqentry_text_start
 irq_exit
 do_IRQ
 ret_from_intr
 cpuidle_enter
 call_cpuidle
 cpu_startup_entry
 start_secondary
   1016045
```

Run in ksoftirqd:

```
 e1000_receive_skb
 e1000e_poll
 net_rx_action
__softirqentry_text_start
 run_ksoftirqd
 smpboot_thread_fn
 kthread
 ret_from_fork
   1162
```

At least 70 other call stacks observed in a few seconds.

Due to handle_pending_softirqs(), any hard IRQ can run before a given softirq (non-RT 4.6.2)

```
e1000_receive_skb
 e1000e_poll
 net_rx_action
 __softirqentry_text_start
 irq_exit
 do_IRQ
 ret_from_intr
 pipe_write
 __vfs_write
 vfs_write
 sys_write
 entry_SYSCALL_64_fastpath
   357
```

```
e1000_receive_skb
 e1000e_poll
 net_rx_action
 __softirqentry_text_start
 irq_exit
 do_IRQ
 ret_from_intr
 __alloc_pages_nodemask
 alloc_pages_vma
 handle_pte_fault
 handle_mm_fault
 __do_page_fault
 do_page_fault
 page_fault
   366
```

Same Experiment, but 4.6.2 with 'threadirqs' boot param

e1000_receive_skb
e1000e_poll
net_rx_action
__softirqentry_text_start
do_softirq_own_stack
do_softirq.part.16
__local_bh_enable_ip
irq_forced_thread_fn
irq_thread
kthread
ret_from_fork
  569174

With 'threadirqs' cmdline parameter at boot.

Note:
no *do_current_softirqs()*

Investigation on ARM:

kprobe with 4.6.2-rt5

Documentation/kprobes.txt

"In general, you can install a probe *anywhere* in the kernel. In particular, you can probe interrupt handlers."

Takeaway: **not** limited to existing tracepoints!

Not *quite* anywhere

root@nitrogen6x:~# insmod 4.6.2/kp_raise_softirq_irqoff.ko
[1749.935955] Planted kprobe at 8012c1b4
[1749.936088] Internal error: Oops - undefined instruction: 0 [#1] PREEMPT SMP ARM
[1749.936109] Modules linked in: kp_raise_softirq_irqoff(+)
[1749.936116] CPU: 0 PID: 0 Comm: swapper/0 Not tainted 4.6.2
[1749.936119] Hardware name: Freescale i.MX6 Quad/DualLite
[1749.936131] PC is at __raise_softirq_irqoff+0x0/0xf0
[1749.936144] LR is at __napi_schedule+0x5c/0x7c
[1749.936766] Kernel panic - not syncing: Fatal exception in interrupt

Mainline stable 4.6.2

code at Github

Adapt samples/kprobes/kprobe_example.c

```
/* For each probe you need to allocate a kprobe structure */
static struct kprobe kp = {
    .symbol_name= "__raise_softirq_irqoff_ksoft",
};
```

in net/core/dev.c

```
/* kprobe post_handler: called after the probed instruction is executed */
static void handler_post(struct kprobe *p, struct pt_regs *regs,unsigned
long flags)
{
     unsigned id = smp_processor_id();
    /* change id to that where the eth IRQ is pinned */
    if (id == 0) { pr_info("Switched to ethernet NAPI.\n");
        pr_info("post_handler: p->addr = 0x%p, pc = 0x%lx,"
            " lr = 0x%lx, cpsr = 0x%lx\n",
        p->addr, regs->ARM_pc, regs->ARM_lr, regs->ARM_cpsr);   }
}
```

Watching net_rx_action() switch to NAPI

```
alison@laptop:~# make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- samples/kprobes/ modules

root@nitrogen6x:~# modprobe kp_ksoft.ko eth_proc_id=1

root@nitrogen6x:~# dmesg | tail
[ 6548.644584] Planted kprobe at 8003344
root@nitrogen6x:~# dmesg | grep post_handler
root@nitrogen6x:~#

 . . . . .  Start DOS attack . . . Wait 15 seconds . . . .

root@nitrogen6x:~# dmesg | tail
[ 6548.644584] Planted kprobe at 80033440
[ 6617.858101] pre_handler: p->addr = 0x80033440, pc = 0x80033444,
lr = 0x80605ff0, cpsr = 0x20070193
[ 6617.858104] Switched to ethernet NAPI.
```

Another example of output

Insert/remove two probes during packet storm:

```
root@nitrogen6x:~# modprobe -r kp_ksoft
[  232.471922] __raise_softirq_irqoff_ksoft: 14 hits
[  232.471922] kprobe at 80033440 unregistered


root@nitrogen6x:~# modprobe -r kp_napi_complete
[  287.225318] napi_complete_done: 1893005 hits
[  287.262011] kprobe at 80605cc0 unregistered
```

Counting activation of two softirq execution paths

```
static struct kprobe kp = {
    .symbol_name= "do_current_softirqs",
};

if (raised == NET_RX_SOFTIRQ) {
        ti = current_thread_info();
        task = ti->task;
        if (chatty)
            pr_debug("task->comm is %s\n", task->comm);

        if (strstr(task->comm, "ksoftirq"))
            p->ksoftirqd_count++;
        if (strstr(task->comm, "irq/"))
            p->local_bh_enable_count++;
    }
```

show you the codez

previously included results

store counters in struct kprobe{}

```
modprobe kp_do_current_softirqs chatty=1
```

Summary

- IRQ handling involves a 'hard', fast part or 'top half' and a 'soft', slower part or 'bottom half.'
- Hard IRQs include arch-dependent system features plus software-generated IPIs.
- Soft IRQs may run directly after the hard IRQ that raises them, or at a later time in ksoftirqd.
- Threaded, preemptible IRQs are a salient feature of RT Linux.
- The management of IRQs, as illustrated by NAPI's response to DOS, remains challenging.
- If you can use bcc and eBPF, you should be!

Acknowledgements

Thanks to Sebastian Siewor, Brenden Blanco, Brendan Gregg, Steven Rostedt and Dave Anders for advice and inspiration.

Special thanks to Joel Fernandes and Sarah Newman for detailed feedback on an earlier version.

Useful Resources

- NAPI docs
- Documentation/kernel-per-CPU-kthreads
- Documentation/DocBook/genericirq.pdf
- Brendan Gregg's blog
- Tasklets and softirqs discussion at KLDP wiki
- #iovisor at OFTC IRC
- Alexei Starovoitov's 2015 LLVM Microconf slides

B1 System Level Programmers' Model
B1.3 ARM processor modes and ARM core registers

ARMv7 Core Registers

Application level view	System level view								
	User	System	Hyp †	Supervisor	Abort	Undefined	Monitor ‡	IRQ	FIQ
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fiq
R9	R9_usr								R9_fiq
R10	R10_usr								R10_fiq
R11	R11_usr								R11_fiq
R12	R12_usr								R12_fiq
SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
LR	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
PC	PC								
APSR	CPSR								
			SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
			ELR_hyp						

Softirqs that don't run in context of hard-IRQ handlers run “on behalf of ksoftirqd”

```
static inline void ksoftirqd_set_sched_params(unsigned int cpu)
{
	/* Take over all but timer pending softirqs when starting */
	local_irq_disable();
	current->softirqs_raised = local_softirq_pending() & ~TIMER_SOFTIRQS;
	local_irq_enable();
}

static struct smp_hotplug_thread softirq_threads = {
	.store			= &ksoftirqd,
	.setup			= ksoftirqd_set_sched_params,
	.thread_should_run	= ksoftirqd_should_run,
	.thread_fn		= run_ksoftirqd,
	.thread_comm		= "ksoftirqd/%u",
};
```

Compare output to source with GDB

```
[alison@hildesheim linux-4.4.4 (trace_napi)]$ arm-linux-gnueabihf-gdb vmlinux
(gdb) p *(__raise_softirq_irqoff_ksoft)
$1 = {void (unsigned int)} 0x80033440 <__raise_softirq_irqoff_ksoft>

(gdb) l *(0x80605ff0)
0x80605ff0 is in net_rx_action (net/core/dev.c:4968).
4963            list_splice_tail(&repoll, &list);
4964            list_splice(&list, &sd->poll_list);
4965            if (!list_empty(&sd->poll_list))
4966                    __raise_softirq_irqoff_ksoft(NET_RX_SOFTIRQ);
4967
4968            net_rps_action_and_irq_enable(sd);
4969    }
```