Linux Device Drivers and Hardware Interrupts

Important Notice : Courseware - Legal

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here: http://bit.ly/ktcorp

Table of Contents

Interrupt Handling

What is an IRQ?

<< Source: LKD3 >>

When a hardware interrupt occours, the processor saves the current state (by saving registers) and jumps to a predefined entry point. This is arch-dependant and usually in assembly.

Here the OS takes control.
It invokes the 'C' runtime control point for all interrupt-handling: do_IRQ().

do_IRQ() receives the saved registers as a parameter; from here, it extracts the IRQ number (into a variable 'vector').

do_IRQ now initiates *interrupt context* by calling irq_enter().
It next invokes the handle_irq() routine which processes the interrupt, invoking the handler(s). Once done, irq_exit() exits interrupt context placing the kernel back in process context.

File : arch/x86/kernel/entry_32.S

```
...
common_interrupt:
    ASM_CLAC
    addl $-0x80,(%esp)  /* Adjust vector into the [-256,-1] range */
    SAVE_ALL
    TRACE_IRQS_OFF
    movl %esp,%eax
    call do_IRQ
    jmp ret_from_intr
ENDPROC(common_interrupt)
...

unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
    struct pt_regs *old_regs = set_irq_regs(regs);

    /* high bit used in ret_from_ code  */
    unsigned vector = ~regs->orig_ax;
    unsigned irq;

    exit_idle();
    irq_enter();

    irq = __this_cpu_read(vector_irq[vector]);
    if (!handle_irq(irq, regs)) {
        ack_APIC_irq();

        if (printk_ratelimit())
            pr_emerg("%s: %d.%d No irq handler for vector (irq %d)\n",
__func__, smp_processor_id(), vector, irq);
    }

    irq_exit();  << first check for pending softirq's and, if so,
invoke_softirq() is called; then, exit interrupt context >>
    set_irq_regs(old_regs);

    return 1;
}
```

However, before returning to the initial entry point, and therefore having the processor restore context to whomever was interrupted, the kernel, on the return path from interrupt, has a (scheduling and signal-recognition) "opportunity point": the TI_NEED_RESCHED bit is checked; if set, the scheduler is invoked.

SIDEBAR :: irqaction

All registered IRQ handlers are maintained on a linked list; each node is the struct irqaction :

```
/**
 * struct irqaction - per interrupt action descriptor
 * @handler:    interrupt handler function
 * @name:    name of the device
 * @dev_id: cookie to identify the device
 * @percpu_dev_id:  cookie to identify the device
 * @next:    pointer to the next irqaction for shared interrupts
 * @irq:     interrupt number
 * @flags:   flags (see IRQF_* above)
 * @thread_fn:  interrupt handler function for threaded interrupts
 * @thread: thread pointer for threaded interrupts
 * @thread_flags:   flags related to @thread
 * @thread_mask:    bitmask for keeping track of @thread activity
 * @dir:     pointer to the proc/irq/NN/name entry
 */
struct irqaction {
    irq_handler_t        handler;
    void                 *dev_id;
    void __percpu        *percpu_dev_id;
    struct irqaction     *next;
    irq_handler_t        thread_fn;
    struct task_struct   *thread;
    unsigned int         irq;
    unsigned int         flags;
    unsigned long        thread_flags;
    unsigned long        thread_mask;
    const char          *name;
    struct proc_dir_entry   *dir;
} ____cacheline_internodealigned_in_smp;
```

SIDEBAR ***[OPTIONAL]***

Hardware Interrupt Handling on ARM Processors

Source :: "ARM System Developer's Guide – Designing and Optimizing System Software" by Sloss, Symes and Wright, published by Elsevier.

An exception is any condition that needs to halt the normal sequential execution of instructions. Examples are when the ARM core is reset, when an instruction fetch or memory access fails, when an undefined instruction is encountered, when a software interrupt instruction is executed, or when an external interrupt has been raised. Exception handling is the method of processing these exceptions.

Most exceptions have an associated software exception handler—a software routine that executes when an exception occurs. For instance, a Data Abort exception will have a Data Abort handler. The handler first determines the cause of the exception and then services the exception. Servicing takes place either within the handler or by branching to a specific service routine. The Reset exception is a special case since it is used to initialize an embedded system.

…

<< Src: ARM Assembly Language, W Hohl.

ARM v4T cores support seven processor modes:

Mode	**Description**	**Privileged?**
Supervisor (SVC)	Entered on reset and when a Software Interrupt (SWI) instruction is executed	Privileged modes
FIQ	Entered when a high priority (fast) interrupt is raised	
IRQ	Entered when a low priority (normal) interrupt is raised	
Abort	Used to handle memory access violations	
Undef	Used to handle undefined instructions	
System	Privileged mode using the same registers as User mode [support added from ARMv4]	

User	Mode under which most application/OS tasks run	Unprivileged mode

An Abort exception/mode switch occurs when the processor attempts to access a physical memory address correspoing to a virtual address translated by the MMU and the physical address is invalid.

Two types of Aborts actually:

- Data Abort : invalid data memory access
- Prefetch Abort : invalid instruction memory access

>>

ARM Processor Exceptions and Modes

Table 9.1 lists the ARM processor exceptions. Each exception causes the core to enter a specific mode. In addition, any of the ARM processor modes can be entered manually by changing the cpsr. User and System mode are the only two modes that are not entered by a corresponding exception, in other words, to enter these modes you must modify the cpsr.

When an exception causes a mode change, the core automatically :

- saves the cpsr to the spsr of the exception mode
- saves the pc to the lr of the exception mode

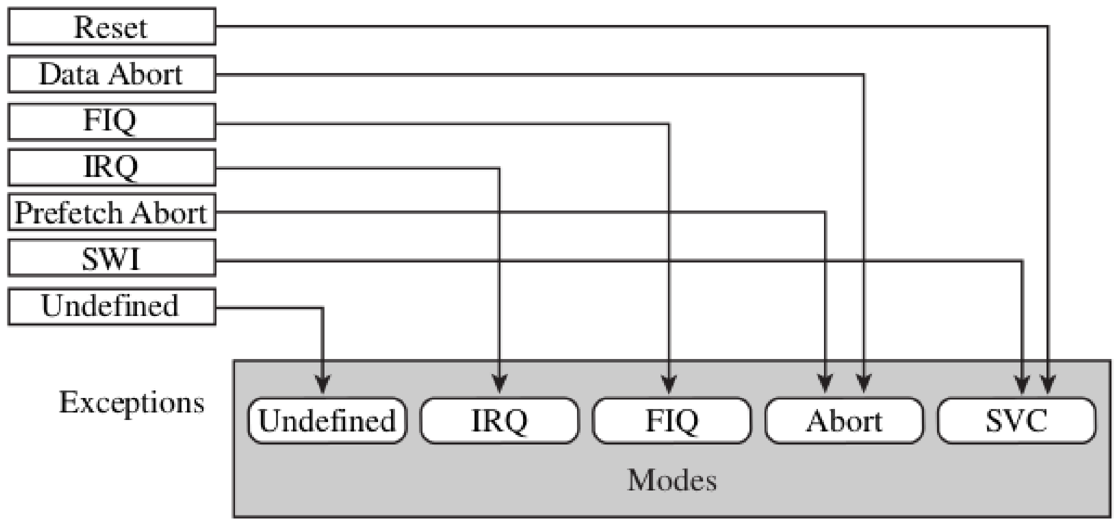


Figure 9.1 Exceptions and associated modes.

…

Exceptions, Interrupts, and the Vector Table

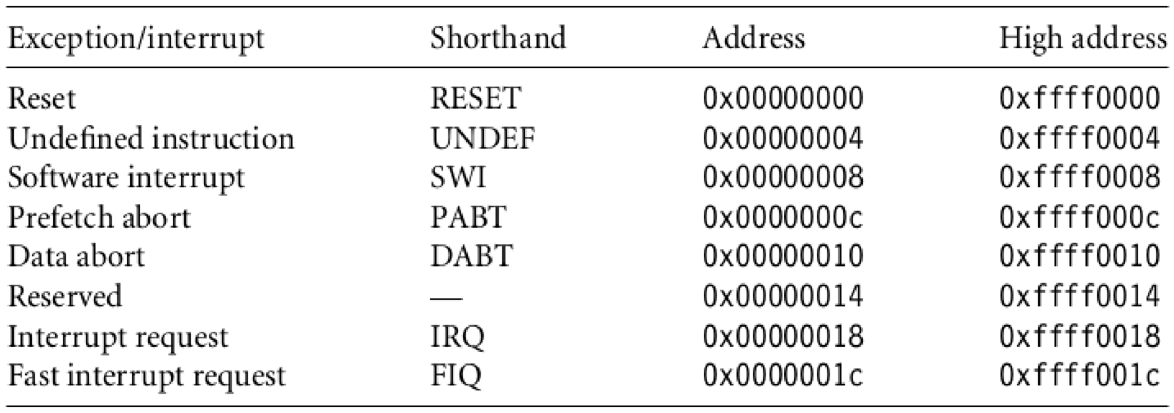
When an exception or interrupt occurs, the (ARM) processor sets the pc to a specific memory address. The address is within a special address range called the vector table. The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt. << equivalent to the IDT on x86 >>.

The memory map address 0x00000000 is reserved for the vector table, a set of 32-bit words. On some processors the vector table can be optionally located at a higher address in memory (starting at the offset `0xffff0000` – virtual address). Operating systems such as Linux and Microsoft's embedded products can take advantage of this feature.

When an exception or interrupt occurs, the processor suspends normal execution and starts loading instructions from the exception vector table (see Table 2.6). Each vector table entry contains a form of branch instruction pointing to the start of a specific routine:

- Reset vector is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
- Undefined instruction vector is used when the processor cannot decode an instruction.
- Software interrupt vector is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine << *on Linux – system calls* >>
- Prefetch abort vector occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.
- Data abort vector is similar to a prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions << *this is what ultimately (from the MMU to OS fault handler to...) leads to your "Segmentation fault, [core dumped]" signals, etc* >>

Table 2.6 The vector table.



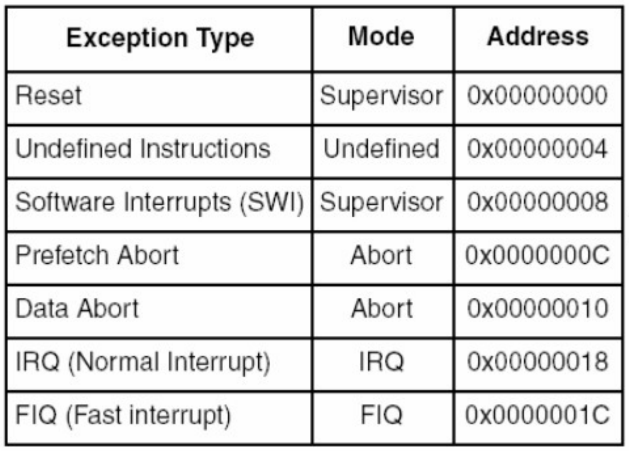
Exception/interrupt	Shorthand	Address	High address
Reset	RESET	0x00000000	0xffff0000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	DABT	0x00000010	0xffff0010
Reserved	—	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c

- Reset vector is the location of the first instruction executed by the processor when power is applied. This instruction branches to the initialization code.
- Interrupt request vector is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the cpsr.
- Fast interrupt request vector is similar to the interrupt request but is reserved for hardware requiring faster response times. It can only be raised if FIQs are not masked in the cpsr.

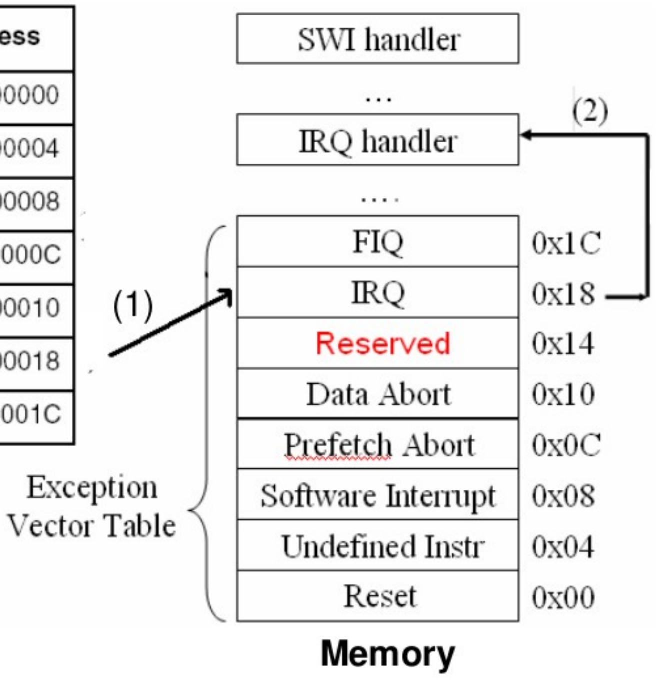
<<
Implementation On Linux

ARM vector table base address is at physical address 0x0, and usually at virtual address 0xffff0000 .

• ARM Exception Vectors



Exception Type	Mode	Address
Reset	Supervisor	0x00000000
Undefined Instructions	Undefined	0x00000004
Software Interrupts (SWI)	Supervisor	0x00000008
Prefetch Abort	Abort	0x0000000C
Data Abort	Abort	0x00000010
IRQ (Normal Interrupt)	IRQ	0x00000018
FIQ (Fast interrupt)	FIQ	0x0000001C



In *arch/arm/mm/mmu.c* :
#define vectors_base() (vectors_high() ? 0Xffff0000 : 0)
...

Where is the ARM vector table initialized and setup in Linux? Short answer: *arch/arm/kernel/traps.c:early_trap_init()*

See this post for more details.

File : arch/arm/kernel/entry-armv.S : *(kernel ver 3.2.21)*

```
...
.LCvswi:
    .word   vector_swi

    .globl  __stubs_end
__stubs_end:

    .equ    stubs_offset, __vectors_start + 0x200 - __stubs_start

    .globl  __vectors_start
__vectors_start:
 ARM(   swi SYS_ERROR0  )
```

```
THUMB( svc #0      )
THUMB( nop         )
   W(b)    vector_und + stubs_offset           ← Undefined instruction
   W(ldr)  pc, .LCvswi + stubs_offset          ← Software interrupt
   W(b)    vector_pabt + stubs_offset          ← Prefetch abort
   W(b)    vector_dabt + stubs_offset          ← Data abort
   W(b)    vector_addrexcptn + stubs_offset    ← Reserved
   W(b)    vector_irq + stubs_offset           ← Interrupt request
   W(b)    vector_fiq + stubs_offset           ← Fast Interrupt request

   .globl  __vectors_end
…
```

>>

OS Low-Level (BSP) Interrupt Handling on ARM Linux

On Linux:

File : arch/arm/include/asm/entry-macro-multi.S

```
…
/*
 * Interrupt handling.  Preserves r7, r8, r9
 */
    .macro  arch_irq_handler_default
    get_irqnr_preamble r6, lr
1:  get_irqnr_and_base r0, r2, r6, lr
    movne   r1, sp
    @
    @ routine called with r0 = irq number, r1 = struct pt_regs *
    @
    adrne   lr, BSYM(1b)
    bne asm_do_IRQ
...
```

File : arch/arm/kernel/irq.c

```
...
/*
 * asm_do_IRQ is the interface to be used from assembly code.
 */
asmlinkage void __exception_irq_entry
asm_do_IRQ(unsigned int irq, struct pt_regs *regs)
{
    handle_IRQ(irq, regs);
}
```

As can be seen from above, the kernel then invokes the 'C' function *asm_do_IRQ()*. This is the C entry point for all interrupt-handling on ARM-Linux.

File : arch/arm/kernel/irq.c

```
...
```

```
void handle_IRQ(unsigned int irq, struct pt_regs *regs)
{
    struct pt_regs *old_regs = set_irq_regs(regs);

    irq_enter();

    /*
     * Some hardware gives randomly wrong interrupts.  Rather
     * than crashing, do something sensible.
     */
    if (unlikely(irq >= nr_irqs)) {
        if (printk_ratelimit())
            printk(KERN_WARNING "Bad IRQ%u\n", irq);
        ack_bad_irq(irq);
    } else {
        generic_handle_irq(irq);
    }

    irq_exit();
    set_irq_regs(old_regs);
}

Basic call-graph:

asm_do_IRQ()                              ← arch/arm/kernel/irq.c
    handle_IRQ()
        irq_enter()
        generic_handle_IRQ()
             generic_handle_irq_desc()
                   desc->handle_irq(irq, desc);   [where 'desc' is a pointer
to struct irq_desc ]
        irq_exit()
```

Some FAQs related to Interrupt processing on Linux

1. How are interrupts (IRQs) distributed among processor cores on a multi-core system?
A.
Source

a) "... It is also possible for any CPU to handle the interrupt. The ACK register will allocate IRQ; the first CPU to read it, gets the interrupt. If multiple IRQs (are) pend(ing) and there are two ACK reads from different CPUs, then each will get a different interrupt. A third CPU reading would get a spurious IRQ. ..."

b) Regarding affinity: Linux uses an IRQ-affinity model; the default is to have all cores enabled to handle an IRQ. This can be changed (as root user). See the IRQ-affinity documentation file for more details on how to do so.

2. Does the kernel maintain a separate IRQ stack? (used for handling hardware interrupts)

A. Short answer, YES -or- it is hardware-dependant:

- IA-32: same kernel-mode stack of interrupted task ('current') is reused
- x86_64: a separate IRQ stack *is* used
- ARM-32 and ARM-64: separate IRQ stacks used
    - ref: [see assembly here: http://stackoverflow.com/questions/18924177/why-does-arm-supervisor-mode-have-its-own-stack] ;
    - and here: arm64: Introduce IRQ stack [patch]
    - also, ARM (32-bit) cores have a "banked" SP register per privilege-level (or mode), so saving and restoring is unecessary and thus much faster!
- PowerPC and SuperH provide the kernel configuration option CONFIG_IRQSTACKS to enable separate stacks for IRQ processing.

Particularly for the x86_64 :

Source: https://www.kernel.org/doc/Documentation/x86/x86_64/kernel-stacks

...

```
x86_64 page size (PAGE_SIZE) is 4K.

Like all other architectures, x86_64 has a kernel stack for every active
thread.  These thread stacks are THREAD_SIZE (2*PAGE_SIZE) big. These stacks
contain useful data as long as a thread is alive or a zombie. While the
thread is in user space the kernel stack is empty except for the thread_info
structure at the bottom.

In addition to the per thread stacks, there are specialized stacks associated
with each CPU.  These stacks are only used while the kernel is in control on
that CPU; when a CPU returns to user space the specialized stacks contain no
useful data.  The main CPU stacks are:

* Interrupt stack.  IRQSTACKSIZE

  Used for external hardware interrupts.  If this is the first external
hardware interrupt (i.e. not a nested hardware interrupt) then the kernel
switches from the current task (stack) to the interrupt stack.
  Like the split thread and interrupt stacks on i386, this gives more room
for kernel interrupt processing without having to increase the size of every
per thread stack.


  The interrupt stack is also used when processing a softirq.
```

--*snip*--

Using 'crash':

```
crash> mach
   MACHINE TYPE: x86_64
           MEMORY SIZE: 7.9 GB
                  CPUS: 4
       PROCESSOR SPEED: 2691 Mhz
                    HZ: 250
             PAGE SIZE: 4096
   KERNEL VIRTUAL BASE: ffff880000000000
   KERNEL VMALLOC BASE: ffffc90000000000
   KERNEL VMEMMAP BASE: ffffea0000000000
      KERNEL START MAP: ffffffff80000000
   KERNEL MODULES BASE: ffffffffa0000000
     KERNEL STACK SIZE: 16384
        IRQ STACK SIZE: 16384
            IRQ STACKS:
                 CPU 0: ffff88023e200000
                 CPU 1: ffff88023e240000
                 CPU 2: ffff88023e280000
                 CPU 3: ffff88023e2c0000
 STACKFAULT STACK SIZE: 4096
--snip--
```

Source : Wrt IA-32, Oct 2012

"...
If a local variable is declared in an ISR, where will it be stored ?
On the kernel stack. The kernel (Linux kernel, that is) does not hook ISRs directly to the x86 architecture's interrupt gates but instead delegates the interrupt dispatch to a common kernel interrupt entry/exit mechanism which saves pre-interrupt register state before calling the registered handler(s). The CPU itself when dispatching an interrupt might execute a privilege and/or stack switch, and this is used/set up by the kernel so that the common interrupt entry code can already rely on a kernel stack being present.

That said, interrupts that occur while executing kernel code will simply (continue to) use the kernel stack in place at that point. This can, if interrupt handlers have deeply nested call paths, lead to stack overflows (if a deep kernel call path is interrupted and the handler causes another deep path; in Linux, filesystem / software RAID code being interrupted by network code with iptables active is known to trigger such in untuned older kernels ... solution is to increase kernel stack sizes for such workloads).
..."

FIQs - Fast Interrupts *Source*

ARM architecture supports two types of interrupts: fast interrupt requests (FIQs), for fast, low latency interrupt handling and Interrupt Request (IRQs), for more general interrupts.[1][2]

An FIQ takes priority over an IRQ in an ARM system. Also, only one FIQ source at a time is supported. This helps reduce interrupt latency as the interrupt service routine can be executed directly without determining the source of the interrupt.

A context save is not required for servicing FIQ since it has its own set of banked registers. This reduces the overhead of context switching.

Related:

(On ARM) What is the difference between FIQ and IRQ interrupt system?
<< Especially read the second answer by Manav M-N. >>

Installing an Interrupt Handler

If you want to actually “see” interrupts being generated, writing to the hardware device isn’t enough; a software handler must be configured in the system. If the Linux kernel hasn’t been told to expect your interrupt, it simply acknowledges and ignores it. Interrupt lines are a precious and often limited resource, particularly when there are only 15 or 16 of them. The kernel keeps a registry of interrupt lines, similar to the registry of I/O ports. A module is expected to request an interrupt channel (or IRQ, for interrupt request) before using it and to release it when finished. In many situations, modules are also expected to be able to share interrupt lines with other drivers, as we will see. The following functions, declared in <*linux/interrupt.h*>, implement the interrupt registration interface:

```
int request_irq(unsigned int irq,
      irq_handler_t (*handler)(int, void *),
       unsigned long flags,
       const char *name,
       void *dev);
void free_irq(unsigned int irq, void *dev);
```

The value returned from request_irq to the requesting function is either 0 to indicate success or a negative error code, as usual. It’s not uncommon for the function to return -EBUSY to signal that another driver is already using the requested interrupt line. The arguments to the functions are as follows:

`unsigned int irq`
The interrupt number being requested.

`irq_handler_t (*handler)(int, void *)`
The pointer to the handling function being installed. We discuss the arguments to this function and its return value later in this chapter.

`unsigned long flags`
As you might expect, a bit maskof options (described later) related to interrupt management.

`const char *dev_name`
The string passed to request_irq is used in /proc/interrupts to show the owner of the interrupt (see the next section).

`void *dev_id`

Pointer used for shared interrupt lines. It is a unique identifier that is used when the interrupt line is freed and that may also be used by the driver to point to its own private data area (to identify which device is interrupting). dev_id must be globally unique.

If the interrupt is not shared, dev_id can be set to NULL, but it a good idea anyway to use this item to point to the device structure.

<< LKD3 >>

Interrupt Handler Flags

The **third parameter, flags**, can be either zero or a bit mask of one or more of the flags defined in *<linux/interrupt.h>*. Among these flags, the most important are :

IRQF_DISABLED —When set, this flag instructs the kernel to disable all interrupts when executing this interrupt handler.When unset, interrupt handlers run with all interrupts except their own enabled.
Most interrupt handlers do not set this flag, as disabling all interrupts is bad form. Its use is reserved for performance-sensitive interrupts that execute quickly. **[*]**

This flag is the current manifestation of the SA_INTERRUPT flag, which in the past distinguished between "fast" and "slow" interrupts.

[*]

(Fairly) Recent Changes in Hardware Interrupt Handling in the Linux Kernel

Summary of Big (Fairly) Recent Changes:

- **Threaded Interrupt Handlers** : Interrupts handlers have the possibility to become Threaded !
  - use *request_irq* to use traditional 'top-half' handler style
- use *request_threaded_irq* to use the new threaded handler style (obviates the need for top/bottom half dichotomies)

Note that most distros (and the defconfig) nowadays use threaded handlers by default (by setting the kernel configuration option CONFIG_ IRQ_FORCED_THREADING !

```
# grep IRQ_FORCED /boot/config-4.4.0-59-generic
CONFIG_IRQ_FORCED_THREADING=y
```

Thus all interrupts except those that use the IRQF_NO_THREAD flag in request_irq() will *be threaded.*

[Details follow in a later section]

- Interrupt Handlers runs with all other interrupts Disabled
  - **IRQF_DISABLED is a nop**

[FYI, DETAIL]

Disabling IRQF_DISABLED
--Ref: http://lwn.net/Articles/380931/

- Classic behaviour: distinguish between "fast" and "slow" handlers; fast handlers run with all interrupts disabled, slow handlers run with all other interrupts enabled.

- Need to distinguish is fading nowadays:
  - hardware is better
  - faster cpu's (can execute a lot more code in the same time)
  - worst interrupt-time offenders have been fixed (mostly from RT tree work)
  - more work moved out of top-half handlers

Snippets from the above article link:

...
Simultaneously, problems associated with the fast/slow dichotomy have been growing. There is no way to run handlers for interrupts on shared lines (found on any system with a PCI bus) with interrupts disabled, because any other handler for a device on the same line can enable interrupts. Allowing interrupt handlers to interrupt each other leads to worse cache behavior and unpredictable completion times. What set off the recent discussion, though, was this patch from Andi Kleen which was aimed at addressing another problem: deeply nested interrupt handlers can overflow the processor's interrupt stack - a situation from which good things cannot be expected to ensue.

Andi's solution is to monitor the depth of the interrupt stack within the core kernel's interrupt-handling code. Should the stack become more than half full, the core code will no longer enable interrupts before calling slow handlers. In effect, it treats slow handlers as if they were fast handlers for the duration of the stack-space squeeze. This patch solved the problem that was being observed, but it ran into some trouble; in particular, Thomas Gleixner did not hesitate to make his dislike for the patch known. Your editor will try to rephrase the argument in slightly more polite terms; according to Thomas, the patch implemented a solution which was unreliable at best, was liable to create significant latencies in the system, and which ignored the real problem.

Said real problem, according to Thomas, is the fact that slow handlers exist at all. He would like to see a world where all interrupt handlers are run with interrupts disabled, and where all of those handlers get their work done quickly. Any extended interrupt processing should be moved to threaded handlers. In summary:

So what's the point of running a well written (short) interrupt handler with interrupts enabled ? Nothing at all. It just makes us deal with crap like stacks overflowing for no good reason.

Linus initially squashed the idea, saying that a world where we only have fast handlers is not really possible:

So Thomas, you're wrong. We can't fix all irq handlers to be really quick, and we MUST NOT run them with all other irq's disabled if they are not - because that obviates the whole point.

It is interesting to note, though, that this position shifted over time. Linus (and others) expressed a number of concerns about running all handlers with interrupts disabled:

The handlers for some devices simply have to do a lot of work, and that cannot be easily changed. Embedded systems, in particular, can have fussy hardware and slow processors.
Some handlers will not work properly if interrupts are not enabled. In the past, some drivers have done things like waiting for a certain amount of time to pass (as reflected in changes to the jiffies variable). This dubious practice fails outright if interrupts are disabled: the timer interrupt will be blocked, and jiffies will not advance.
Some hardware simply has strict latency requirements which cannot wait for another interrupt handler to finish its job.

Looking at all these worries, one might well wonder if a system which disabled interrupts for all handlers would function well at all. So it is interesting to note one thing: any system which has the lockdep locking checker enabled has been running all handlers that way for some years now. Many developers and testers run lockdep-enabled kernels, and they are available for some of the more adventurous distributions (Rawhide, for example) as well. So we have quite a bit of test coverage for this mode of operation already.

Another thing that happened over the last few years was the integration of the dynamic tick code, which disables the clock tick when the system is idle. Clock ticks are not turned back on for interrupt handlers. So any handler which expects jiffies to change while it is running will, sooner or later, go into a rather undignified infinite loop. Users tend to notice that kind of behavior, so most drivers which behave this way have long since been fixed.

Finally, the realtime tree developers have spent a great deal of time tracking down sources of latency; excessive time spent in interrupt handlers is one of the worst of those. So drivers which control hardware of interest have generally been fixed. The addition of threaded interrupt handlers has made it easier to fix drivers; most of the code can simply be pushed into the threaded handler with no other change at all.

Given all of this, Ingo Molnar felt confident in saying:
I'm fairly certain, based on having played with those aspects from many angles that disabling irqs in all drivers should work just fine today already.

After hearing this from a few core developers, and after doing some research of his own, Linus eventually stopped opposing the idea and started talking about how it should be implemented. Thomas then posted a patch implementing the change. With this patch, the IRQF_DISABLED flag (used to indicate a fast handler) becomes a no-op; it is expected to be removed altogether in 2.6.36.

There are still some concerns about the change, especially with regard to slow hardware on embedded systems. In some of these cases, the problem can be solved with threaded interrupt handlers. Some developers worry, though, that threaded handlers impose too much latency on interrupt response. Improving on that situation is a task for the future; in the mean time, some interrupt handlers may just have to enable interrupts internally to get the required behavior. The preferred function for this purpose is local_irq_enable_in_hardirq(); its use can already be found in the IDE layer.

Code View

request_irq()* is a wrapper over *request_threaded_irq(), with the third parameter – the *threaded handler* – set to NULL.

NO irq thread created by default

include/linux/interrupt.h
```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, uns     d long flags,
const char *name, void *dev)
{
    return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}
```

kernel/irq/manage.c
```
/**
 *  request_threaded_irq - allocate an interrupt line
 *  @irq: Interrupt line to allocate
 *  @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
 *  @irqflags: Interrupt type flags
 *  @devname: An ascii name for the claiming device
 *  @dev_id: A cookie passed back to the handler function
...
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
irq_handler_t thread_fn, unsigned long irqflags,
             const char *devname, void *dev_id)
{
...

}
```

The job of the primary handler (or "hard" handler or ISR or top half) is to :
a) if it's a shared interrupt (happens quite often), check that it's for your device, else return (with IRQ_NONE)
b) acknowledge and turn off the interrupt on the hardware, and
c) bring in data (as appropriate)

See this SO discussion: Clarification about the behavior of request_threaded_irq

Here's the commit by Thomas Gleixner (github) [2.6.30-rc1] Mar 23, 2009:

genirq: add threaded interrupt handler support

Add support for threaded interrupt handlers:

A device driver can request that its main interrupt handler runs in a thread. To achive this the device driver requests the interrupt with request_threaded_irq() and provides additionally to the handler a thread function. The handler function is called in hard interrupt context and needs to check whether the interrupt originated from the device. If the interrupt originated from the device then the handler can either return IRQ_HANDLED or IRQ_WAKE_THREAD. IRQ_HANDLED is returned when no further action is required. IRQ_WAKE_THREAD causes the genirq code to invoke the threaded (main) handler. When IRQ_WAKE_THREAD is returned handler must have disabled the interrupt on the device level. This is mandatory for shared interrupt handlers,

```
but we need to do it as well for obscure x86 hardware where disabling
an interrupt on the IO_APIC level redirects the interrupt to the
legacy PIC interrupt lines.

Signed-off-by: Thomas Gleixner <tglx@linutronix.de>
Reviewed-by: Ingo Molnar <mingo@elte.hu>
...
```

IRQF_SAMPLE_RANDOM — << ***Deprecated*** >>

From kernel ver 3.6 onward, the kernel handles interrupt randomness contribution to the entropy pool itself; this flag is thus deprecated.

<< Older: < 3.6.x >>
This flag specifies that interrupts generated by this device should contribute to the kernel entropy pool. The kernel entropy pool provides truly random numbers derived from various random events. If this flag is specified, the timing of interrupts from this device are fed to the pool as entropy.

Do not set this if your device issues interrupts at a predictable rate (for example, the system timer) or can be influenced by external attackers (for example, a networking device). On the other hand, most other hardware generates interrupts at nondeterministic times and is, therefore, a good source of entropy.

IRQF_TIMER —This flag specifies that this handler processes interrupts for the system timer.

<linux/interrupt.h>

```
…
/* ...
 * IRQF_TIMER - Flag to mark this interrupt as timer interrupt
…
 * IRQF_NO_SUSPEND - Do not disable this IRQ during suspend
 * IRQF_NO_THREAD - Interrupt cannot be threaded
 */
…
#define IRQF_TIMER      (__IRQF_TIMER | IRQF_NO_SUSPEND |
IRQF_NO_THREAD)
…
```

IRQF_SHARED —This flag specifies that the interrupt line can be shared among multiple interrupt handlers. Each handler registered on a given line must specify this flag; otherwise, only one handler can exist per line. More information on shared handlers is provided in a following section.

<<
Level or Edge Triggered Interrupt

Source
Level triggered: as long as the IRQ line is asserted, you get an interrupt request. When you serve the interrupt and return, if the IRQ line is still asserted, you get the interrupt again immediately.

Edge triggered: You get an interrupt when the line changes from inactive to active state, but only once. To get a new request, the line must go back to inactive and then to active again.
…

There are IRQ flags to control this; having said that, it's generally (usually) expected that this seeting is already configured (via firmware or per machine init code).

USEFUL:

- *PCI interrupts are specified to always be level-low sensitive << level-triggered, low => intrerrupt >>*
- *See this discussion on SO: Concurrent interrupt handling in Linux (regarding usage of level-triggered interrupts when sharing is used).*

>>

The **fourth parameter**, name, is an ASCII text representation of the device associated with the interrupt. For example, this value for the keyboard interrupt on a PC is keyboard.These text names are used by /proc/irq and /proc/interrupts for communication with the user, which is discussed shortly.

The **fifth parameter**, dev, is used for shared interrupt lines.When an interrupt handler is freed (discussed later), dev provides a unique cookie to enable the removal of only the desired interrupt handler from the interrupt line. Without this parameter, it would be impossible for the kernel to know which handler to remove on a given interrupt line. You can pass NULL here if the line is not shared, but you must pass a unique cookie if your interrupt line is shared. (And unless your device is old and crusty and lives on the ISA bus, there is a good chance it must support sharing.)

This pointer is also passed into the interrupt handler on each invocation. A common practice is to pass the driver's device structure:This pointer is unique and might be useful to have within the handlers.

Tip: See http://stackoverflow.com/questions/15245626/simple-interrupt-handler-request-irq-returns-error-code-22?rq=1 as an example of why we require dev_id.

On success, *request_irq()* returns zero. A nonzero value indicates an error, in which case the specified interrupt handler was not registered.A common error is -EBUSY, which denotes that the given interrupt line is already in use (and either the current user or you did not specify IRQF_SHARED).

Note that *request_irq()* can sleep and therefore cannot be called from interrupt context or other situations where code cannot block. It is a common mistake to call request_irq() when it is unsafe to sleep. This is partly because of why *request_irq()* can block: It is indeed unclear:

On registration, an entry corresponding to the interrupt is created in */proc/irq*. The function *proc_mkdir()* creates new procfs entries.This function calls *proc_create()* to set up the new procfs entries, which in turn calls *kmalloc()* to allocate memory. As you will see, *kmalloc()* can sleep. So there you go!

<<
Can use ***/proc/interrupts*** to lookup currently installed IRQ handler information. Eg.

a) on an x86_64 running Ubuntu 17.04

```
$ uname -r
4.10.0-32-generic
$
$ cat /proc/interrupts
            CPU0       CPU1       CPU2       CPU3
   0:         25          0          0          0  IR-IO-APIC    2-edge      timer
   1:     113587         29       7323       1053  IR-IO-APIC    1-edge      i8042
   8:          1          0          0          0  IR-IO-APIC    8-edge      rtc0
   9:      91482        121      42945       2826  IR-IO-APIC    9-fasteoi   acpi
  12:     313243         28      12900       1379  IR-IO-APIC   12-edge      i8042
 120:          0          0          0          0  DMAR-MSI    0-edge      dmar0
 121:          0          0          0          0  DMAR-MSI    1-edge      dmar1
 122:       1019      11502     379184      53038  IR-PCI-MSI 327680-edge      xhci_hcd
 123:        100          0         13          5  IR-PCI-MSI 1048576-edge      rtsx_pci
 124:     577618        843    2558372    2970144  IR-PCI-MSI 376832-edge
ahci[0000:00:17.0]
 125:          2          0         21         11  IR-PCI-MSI 360448-edge      mei_me
 126:          3          3         47          4  IR-PCI-MSI 3145728-edge
 127:    6372021    2234906         33     740500  IR-PCI-MSI 32768-edge      i915
 128:          0          0         87          9  IR-PCI-MSI 514048-edge
snd_hda_intel:card0
 129:      77239     145614     469419     443582  IR-PCI-MSI 2097152-edge      iwlwifi
 130:          0        110       1530        170  IR-PCI-MSI 520192-edge      enp0s31f6
 NMI:         92        500        518        526   Non-maskable interrupts
 LOC:   17722501   14387780   14641624   14337857   Local timer interrupts
 SPU:          0          0          0          0   Spurious interrupts
 PMI:         92        500        518        526   Performance monitoring interrupts
 IWI:          3          2          9          2   IRQ work interrupts
 RTR:         12          1          0          0   APIC ICR read retries
 RES:     945135     850296     788583     658663   Rescheduling interrupts
 CAL:     555888     510295     505432     508562   Function call interrupts
 TLB:     471685     476995     474196     478524   TLB shootdowns
 TRM:        402        402        402        402   Thermal event interrupts
 THR:          0          0          0          0   Threshold APIC interrupts
 DFR:          0          0          0          0   Deferred Error APIC interrupts
 MCE:          0          0          0          0   Machine check exceptions
 MCP:        252        249        249        249   Machine check polls
 ERR:          0
 MIS:          0
 PIN:          0          0          0          0   Posted-interrupt notification event
 PIW:          0          0          0          0   Posted-interrupt wakeup event
$
```

b) on an Android (S7) phone (multicore, 6 CPU cores)

```
$ adb shell
herolte:/ $ uname -a
Linux localhost 3.18.14-11104523 #1 SMP PREEMPT Mon Jul 10 17:36:37 KST 2017 aarch64
herolte:/ $ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5
  1:        427          0          0          0          0          0
exynos_wkup_irq_chip   0  abov-touchkey
  2:          7          0          0          0          0          0
exynos_wkup_irq_chip   2  sec-nfc
  3:          0          0          0          0          0          0   PCI-MSI   0  PCIe
PME, aerdrv
  5:        542          2          0          0         27         19
exynos_wkup_irq_chip   5  sx9310_irq
  6:        192          0          0          0          0          0
exynos_wkup_irq_chip   6  arizona
  7:         83          0          0          0          0          0
exynos_wkup_irq_chip   6  max77854-irq
  8:      20947          1          0          0          0          2
exynos_wkup_irq_chip   7  hrm_sensor_irq
  9:       2366          0          0          0          0          0
exynos_wkup_irq_chip   6  sec-pmic-irq
 27:          0          0          0          0          0          0       GIC  27
arch_timer
```

```
 32:          0          0          0          0          0          0       GIC  32
10580000.pinctrl
 53:          0          0          0          0          0          0       GIC  53
cp_fail
 89:          0          0          0          0          0          0       GIC  89
114e0000.sysmmu
 94:          0          0          0          0          0          0       GIC  94
lpass-ca5
 98:    1007340    1054909     744827     464132          0          0       GIC  98
11420000.adma

...

492:          0          0          0          0          0          0       GIC 492
14cc0000.pinctrl
516:          0          0          0          0          0          0  max77854  bypass-
irq
517:          0          0          0          0          0          0  max77854  batp-irq
520:          0          0          0          0          0          0  max77854  wpc-int
521:         11          0          0          0          0          0  max77854  chgin-
irq
522:         56          0          0          0          0          0  max77854  aicl-irq
523:          0          0          0          0          0          0  max77854  muic-irq
...
539:          0          0          0          0          0          0  max77854
fuelgauge-irq
550:       2361          0          0          0          0          0  sec-pmic  rtc-
alarm0
...
593:          3          0          0          0          0          0
exynos_wkup_irq_chip   7  sec-nfc
594:      97290          0          0          0          0          0
exynos_wkup_irq_chip   0  sec_ts
595:          0          0          0          0          0          0
exynos_gpio_irq_chip   1  SMPL WARN
596:       2861          0          0          0          2          0
exynos_wkup_irq_chip   3  bt_host_wake
597:          1          0          0          0          0          0
exynos_wkup_irq_chip   5  tflash_det
...
606:          0          0          0          0          0          0  moon IRQ  16
ADSP2 interrupt 1
607:          0          0          0          0          0          0  moon IRQ  83
ADSP2 bus error
608:          0          0          0          0          0          0  moon IRQ  84
ADSP2 bus error
609:          0          0          0          0          0          0  moon IRQ  85
ADSP2 bus error
610:          0          0          0          0          0          0  moon IRQ  86
ADSP2 bus error
611:          0          0          0          0          0          0  moon IRQ  87
ADSP2 bus error
612:          0          0          0          0          0          0  moon IRQ  88
ADSP2 bus error
613:          0          0          0          0          0          0  moon IRQ  89
ADSP2 bus error
614:          0          0          0          0          0          0  moon IRQ  17
ADSP2 Spk prot
```

```
615:    1028778          0          0          0          0          0
exynos_gpio_irq_chip   0  13960000.decon_f
616:          0          0          0          0          0          0
exynos_gpio_irq_chip   1  pcd-irq
...
IPI0:  14918903   17689502   16316164   12097562    4110015    4022712       Rescheduling
interrupts
IPI1:     41277      41960      37889      35593      44302      45120       Function call
interrupts
IPI2:     61899     170855     160924     105366     956597     733718       Single
function call interrupts
IPI3:         0          0          0          0          0          0       CPU stop
interrupts
IPI4:         0          0          0          0          0          0       Timer
broadcast interrupts
IPI5:         0          0          0          0       4035       4003       IRQ work
interrupts
IPI6:         0       7342       7342       7342       7342       7342       CPU Wakeup by
interrupts
Err:          0
herolte:/ $
```

>>

<< Source: LDD3 >

Implementing an Interrupt Handler

So far, we've learned to register an interrupt handler but not to write one. Actually, there's nothing unusual about a handler—it's ordinary C code. The only peculiarity is that a handler runs at interrupt time and, therefore, suffers some **restrictions** on what it can do.

- A handler *can't transfer data* to or from user space, because it doesn't execute in the context of a process, and the data transfer could generate a page fault.
  << *So?* It is a design requirement that the page fault handling code runs in process context only. >>
- Handlers also *cannot do anything that would sleep*, such as calling wait_event, allocating memory with anything other than GFP_ATOMIC, or locking a semaphore.
- Finally, handlers *cannot call schedule*.

Effectively, it is illegal for interrupt context code to call *schedule()* (or any variant of it) either directly or indirectly.

Why? ***A nice clear explanation:*** ***Why can't you sleep in an interrupt handler in the Linux kernel? Is this true of all OS kernels?***

Why kernel code/thread executing in interrupt context cannot sleep?

<<

SIDEBAR :: An experiment: Calling schedule() in interrupt context

GitHub repo: https://github.com/kaiwan/trg_linux_dd_L3.git
intr_ctx/intr_work.c

We run a kernel module that deliberately invokes *schedule()* within interrupt context! (We leverage the kernel's *irq_work* functionality to run some code in interrupt context. Output below is on a x86_64 Ubuntu 18.04 LTS guest running on Oracle VirtualBox 5.2 on an x86_64 Ubuntu 18.10 host system):

```
$ sudo insmod ./intr_work.ko

...

[12795.931651] intr_work: attempting sleep in irq ctx now...
[12795.931652] BUG: scheduling while atomic: insmod/11347/0x00010000
[12795.931655] Modules linked in: intr_work(OE+) nls_utf8 isofs vboxsf
crct10dif_pclmul crc32_pclmul ghash_clmulni_intel snd_intel8x0 snd_ac97_codec
ac97_bus joydev pcbc snd_pcm aesni_intel aes_x86_64 crypto_simd glue_helper
cryptd vboxvideo(CE) snd_seq_midi snd_seq_midi_event intel_rapl_perf ttm
snd_rawmidi input_leds drm_kms_helper serio_raw mac_hid snd_seq snd_seq_device
snd_timer drm fb_sys_fops syscopyarea sysfillrect snd soundcore vboxguest
sysimgblt sch_fq_codel parport_pc ppdev lp parport ip_tables x_tables autofs4
hid_generic usbhid hid psmouse e1000 ahci libahci pata_acpi i2c_piix4 video [last
unloaded: intr_work]
[12795.931709] CPU: 0 PID: 11347 Comm: insmod Tainted: G         C OE    4.15.0-
43-generic #46-Ubuntu
[12795.931710] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox
12/01/2006
[12795.931711] Call Trace:
[12795.931712]  <IRQ>
[12795.931717]  dump_stack+0x63/0x8b
[12795.931720]  __schedule_bug+0x55/0x70
[12795.931722]  __schedule+0x5fd/0x8a0
[12795.931724]  ? printk+0x52/0x6e
[12795.931726]  schedule+0x2c/0x80
[12795.931728]  my_irq_work+0x117/0x15e [intr_work]
[12795.931731]  irq_work_run_list+0x52/0x80
[12795.931733]  irq_work_run+0x18/0x40
[12795.931734]  smp_irq_work_interrupt+0x3e/0xd0
[12795.931736]  irq_work_interrupt+0x84/0x90
[12795.931736]  </IRQ>
[12795.931739] RIP: 0010:native_write_msr+0xa/0x30
[12795.931740] RSP: 0018:ffffc2ef028dfc10 EFLAGS: 00000246 ORIG_RAX:
ffffffffffffff09
[12795.931742] RAX: 00000000000000f6 RBX: 0000000000000001 RCX: 000000000000083f
[12795.931743] RDX: 0000000000000000 RSI: 00000000000000f6 RDI: 000000000000083f
[12795.931744] RBP: ffffc2ef028dfc10 R08: 0000000000000001 R09: 00000000000003f2
```

```
[12795.931745] R10: 0000000000000000 R11: 0000000000000000 R12: ffffffffc028d0ff
[12795.931745] R13: ffff9f857bc85660 R14: 0000000000000001 R15: ffff9f85794685a0
[12795.931748]  native_apic_msr_write+0x2b/0x40
[12795.931750]  x2apic_send_IPI_self+0x20/0x30
[12795.931752]  arch_irq_work_raise+0x2a/0x40
[12795.931753]  irq_work_queue+0x8d/0xa0
[12795.931756]  intr_drv_init+0x145/0x1000 [intr_work]
[12795.931757]  ? 0xffffffffc01b4000
[12795.931759]  do_one_initcall+0x52/0x19f

...

[12795.931789] R13: 000055e78a167760 R14: 0000000000000000 R15: 0000000000000000
[12795.931794] ------------[ cut here ]------------
[12795.931795] kernel BUG at
/build/linux-vxxS7y/linux-4.15.0/kernel/irq_work.c:138!
[12795.931800] invalid opcode: 0000 [#1] SMP PTI
...
```

>>

(continuing from above ...)

The role of an interrupt handler is to give feedback to its device about interrupt reception and to read or write data according to the meaning of the interrupt being serviced. The first step usually consists of clearing a bit on the interface board; most hardware devices won't generate other interrupts until their "interrupt-pending" bit has been cleared. Depending on how your hardware works, this step may need to be performed last instead of first; there is no catch-all rule here. Some devices don't require this step, because they don't have an "interrupt-pending" bit; such devices are a minority, although the parallel port is one of them.

A typical task for an interrupt handler is awakening processes sleeping on the device if the interrupt signals the event they're waiting for, such as the arrival of new data. To stick with the frame grabber example, a process could acquire a sequence of images by continuously reading the device; the read call blocks before reading each frame, while the interrupt handler awakens the process as soon as each new frame arrives. This assumes that the grabber interrupts the processor to signal successful arrival of each new frame.

The programmer should be careful to write a routine **that executes in a minimum amount of time**, independent of it being a fast or slow handler. If a long computation needs to be performed, the best approach is to use a tasklet or workqueue (deferred functionality mechanisms) to schedule computation at a safer time.

Handler Arguments and Return Value

Signature:

`static irqreturn_t interrupt_handler(int irq, void *dev_id)`

Two arguments are passed to an interrupt handler: *irq* and *dev_id*. Let's look at the role of each.

The **interrupt number** (*int irq*) is useful as information you may print in your log messages, if any.

The second argument, *void *dev_id*, is a sort of **client data**; a void * argument is passed to *request_irq*, and this same pointer is then passed back as an argument to the handler when the interrupt happens. You usually pass a pointer to your device data structure in dev_id, so a driver that manages several instances of the same device doesn't need any extra code in the interrupt handler to find out which device is in charge of the current interrupt event.

Also, many drivers (typically PCI) share irq lines; hence we need a way to know whether it's really our driver that should service this interrupt instance. dev_id could be used to help figure this out.
More often though, a hardware-specific check is done to see whether the interrupt is really meant for this device. For example, from the interrupt handler of the PCI Ethernet RTL8139 driver (*drivers/net/8139too.c:rtl8139_interrupt*):

```
...
static irqreturn_t rtl8139_interrupt (int irq, void *dev_instance)
{
        struct net_device *dev=(struct net_device *)dev_instance;
        struct rtl8139_private *tp = netdev_priv(dev);
        void __iomem *ioaddr = tp->mmio_addr;
        u16 status, ackstat;
        int link_changed = 0; /* avoid bogus "uninit" warning */
        int handled = 0;

        spin_lock (&tp->lock);
        status = RTL_R16 (IntrStatus);

        /* shared irq? */
        if (unlikely((status & rtl8139_intr_mask) == 0))
                goto out;

        handled = 1;
...
```

Return Value:

Interrupt handlers should return a value indicating whether there was actually an interrupt to handle. If the handler found that its device did, indeed, need attention, **it should return IRQ_HANDLED** ; otherwise the return value should be IRQ_NONE . You can also generate the return value with this macro:
`IRQ_RETVAL(handled)`

where handled is nonzero if you were able to handle the interrupt. The return value is used by the kernel to detect and suppress spurious interrupts. If your device gives you no way to tell whether it really interrupted, you should return IRQ_HANDLED .

File : [3.10.24] include/linux/irqreturn.h

```
#ifndef _LINUX_IRQRETURN_H
#define _LINUX_IRQRETURN_H

/**
 * enum irqreturn
 * @IRQ_NONE        interrupt was not from this device
 * @IRQ_HANDLED     interrupt was handled by this device
 * @IRQ_WAKE_THREAD handler requests to wake the handler thread
 */
enum irqreturn {
    IRQ_NONE        = (0 << 0),
    IRQ_HANDLED     = (1 << 0),
    IRQ_WAKE_THREAD     = (1 << 1),
};

typedef enum irqreturn irqreturn_t;
#define IRQ_RETVAL(x)   ((x) != IRQ_NONE)

#endif
```

SIDEBAR : Reentrancy and Interrupt Handlers

Interrupt handlers in Linux need not be reentrant. When a given interrupt handler is executing, the corresponding interrupt line is masked out on all processors, preventing another interrupt on the same line from being received. Normally all other interrupts are enabled, so other interrupts are serviced, but the current line is always disabled.

Consequently, the same interrupt handler is never invoked concurrently to service a nested interrupt. This greatly simplifies writing your interrupt handler.

Threaded Interrupt Handlers

Quick Summary:

Interrupts handlers now (2.6.30, June 2009) have the possibility to become Threaded :

- You should use this if you *require* the interrupt handling to be running in a task context that can block. To do so, explicitly call `request_threaded_irq()` passing the thread handling function pointer (3rd parameter).
- If you use the “usual” / traditional `request_irq()` function, you get the traditional style “top-half / bottom-half” handling mechanism
- Additionally, one could pass the `IRQF_NO_THREAD` flag to use traditional 'top-half' handler style.

Thomas Gleixner merged threaded IRQs into the mainline Linux kernel (from the PREEMPT_RT patch). The commit SHA1 (starts with) 3aa551c9 dated Mon 23 March 2009. Kernel ver: 2.6.30-rc1
[
An interesting 'git' Aside:
How does one figure the kernel version given only the patch (SHA1) id?

Ref: Given a git patch id (or commit SHA1 hash), how to find out which kernel release contains it?

Short Ans:
Use '`git describe`' like so (on the local repo of course; note that it can take a while!):

```
$ git describe --contains 3aa551c9
v2.6.30-rc1~3^2~5
$
```

BTW, 3aa551c9 is the (first 8 digits) of the unique SHA1 id of the "Add support for threaded interrupt handlers - V3" by Thomas Gleixner !
]

[Link to the patch]

[FYI, Details]:

Moving Interrupts to Threads

--Ref: http://lwn.net/Articles/302043/

Processing interrupts from the hardware is a major source of latency in the kernel, because other interrupts are blocked while doing that processing. For this reason, the realtime tree has a feature, called threaded interrupt handlers, that seeks to reduce the time spent with interrupts disabled to a bare minimum—pushing the rest of the processing out into kernel threads. But it is not just realtime kernels that are interested in lower latencies, so threaded handlers are being proposed for addition to the mainline.

Traditionally, interrupt handling has been done with top half (i.e. the "hard" irq) that actually responds to the hardware interrupt and a bottom half (or "soft" irq) that is scheduled by the top half to do additional processing. The top half executes with interrupts disabled, so it is imperative that it do as little as possible to keep the system responsive. Threaded interrupt handlers reduce that work even further, so the top half would consist of a "quick check handler" that just ensures the interrupt is from the device; if so, it simply acknowledges the interrupt to the hardware and tells the kernel to wake the interrupt handler thread.

In the realtime tree, nearly all drivers were mass converted to use threads, but the patch Gleixner proposes makes it optional—driver maintainers can switch if they wish to. Automatically converting drivers is not necessarily popular with all maintainers, but it has an additional downside as Gleixner notes: "Converting an interrupt to threaded makes only sense when the handler code takes advantage of it by integrating tasklet/softirq functionality and simplifying the locking."

A driver that wishes to request a threaded interrupt handler will use:

```
  int request_threaded_irq(unsigned int irq,
                      irq_handler_t handler,
                         irq_handler_t quick_check_handler,
                       unsigned long flags, const char *name,
                      void *dev)
```

<< the *quick_check_handler* ultimately invokes *schedule_work()*; this is the **work queue** mechanism of getting kernel threads to consume some work! >>

This is essentially the same as request_irq() with the addition of the quick_check_handler. As requested by Linus Torvalds at this year's Kernel Summit, a new function was introduced rather than changing countless drivers to use a new request_irq().

The quick_check_handler checks to see if the interrupt was from the device, returning IRQ_NONE if it isn't. It can also return IRQ_HANDLED if no further processing is required or IRQ_WAKE_THREAD to wake the handler thread. One other return code was added to simplify converting to a threaded handler. A quick_check_handler can be developed prior to the handler being converted; in that case, it returns

IRQ_NEEDS_HANDLING (instead of IRQ_WAKE_THREAD) which will call the handler in the usual way.
--

So, from 2.6.36 (??), threaded interrupt handlers are a reality.
Driver developers still have a choice though: use ***request_irq()* directly to use the "traditional / classic" approach.**

```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
const char *name, void *dev)
{
    return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}

...
```

NULL by default =>
***no** threaded handler*

If a threaded handler function pointer is passed:

```
request_threaded_irq   calls  __setup_irq(irq, desc, action);

...

static int
__setup_irq(unsigned int irq, struct irq_desc *desc, struct irqaction *new)
{
...
    /*
     * Create a handler thread when a thread function is supplied
     * and the interrupt does not nest into another interrupt
     * thread.
     */
    if (new->thread_fn && !nested) {
        struct task_struct *t;

        t = kthread_create(irq_thread, new, "irq/%d-%s", irq,
                   new->name);
        if (IS_ERR(t)) {
            ret = PTR_ERR(t);
            goto out_mput;
        }
...
```

Eg.: On an IA-32 x86 running Ubuntu 13.10 kernel ver 3.11.0-14-generic :

```
# ps aux|grep --color=auto 'irq/'
root       348  0.0  0.0      0     0 ?        S    Jan01   0:40 [irq/49-iwlwifi]
root      5897  0.0  0.0      0     0 ?        S    07:21   0:00 [irq/45-mei_me]
root      7551  0.0  0.0   4452   820 pts/5    S+   08:17   0:00 grep --color=auto irq/
#
```

<<

2.6.39: Add a commandline parameter "threadirqs" which forces all interrupts except those marked IRQF_NO_THREAD to run threaded (commit)

From https://www.kernel.org/doc/Documentation/kernel-parameters.txt

...

```
threadirqs      [KNL]
                Force threading of all interrupt handlers except those
                marked explicitly IRQF_NO_THREAD.
```

...

>>

Further to the above, some interesting Kprobes work (we inserted a kprobe on *schedule()*), can actually show us that threaded interrupts are process context and are therefore scheduled similar to other "normal" threads:

```
  irq/49-iwlwifi-348   [002] d...  7941.410786: handler_pre.part.0:
handler_pre:59 : irq/49-iwlwifi:348: Pre 'schedule'. Intr? n.

  irq/49-iwlwifi-348   [002] d...  7941.410794: handler_post:
handler_post:74 : irq/49-iwlwifi:348. Post 'schedule'.

  irq/49-iwlwifi-348   [002] d...  7941.410796: handler_post:  delta: 13 us,
avg:13


   kworker/u16:0-16607 [000] d...  7941.410853: handler_pre.part.0:
handler_pre:59 : kworker/u16:0:16607: Pre 'schedule'. Intr? n.

   kworker/u16:0-16607 [000] d...  7941.410860: handler_post:
handler_post:74 : kworker/u16:0:16607. Post 'schedule'.

   kworker/u16:0-16607 [000] d...  7941.410862: handler_post:  delta: 13 us,
avg:13


  irq/49-iwlwifi-348   [002] d...  7941.412689: handler_pre.part.0:
handler_pre:59 : irq/49-iwlwifi:348: Pre 'schedule'. Intr? n.

  irq/49-iwlwifi-348   [002] d...  7941.412696: handler_post:
handler_post:74 : irq/49-iwlwifi:348. Post 'schedule'.

  irq/49-iwlwifi-348   [002] d...  7941.412697: handler_post:  delta: 11 us,
avg:12


   kworker/u16:0-16607 [000] d...  7941.412748: handler_pre.part.0:
handler_pre:59 : kworker/u16:0:16607: Pre 'schedule'. Intr? n.

   kworker/u16:0-16607 [000] d...  7941.412755: handler_post:
handler_post:74 : kworker/u16:0:16607. Post 'schedule'.
```

```
   kworker/u16:0-16607 [000] d...  7941.412756: handler_post:  delta: 11 us,
avg:11
[…]
```

FAQ > In interrupt context code, how do i know if it's running in "classic" hardirq (top-half) or "bottom-half" or threaded interrupt (process) context?

Ans > Use *in_interrupt();* if it returns True, you are in interrupt context. If false, you're running in process context.
Tip: a *threaded* interrupt runs in process context of course (in the context of a kernel thread to be precise).

Several other macros exist as well providing more detail (see *include/linux/hardirq.h*).

A (wrapper) macro I find useful:

```
#ifdef __KERNEL__
#include <linux/sched.h>
#include <linux/interrupt.h>
#define PRINT_CTX() do {                                                    \
        printk_ratelimited("PRINT_CTX:: [cpu %02d]%s-%d\n",                  \
            smp_processor_id(), __func__, current->pid);                    \
       if (!in_interrupt())                                                 \
               printk(" in process context:%c%s%c:%d\n",                      \
                 (!current->mm?'[':' '), current->comm,                       \
               (!current->mm?']':' '), current->pid);                       \
      else                                                                  \
        printk(" in interrupt context: in_interrupt:%3s. in_irq:%3s.    \ in_softirq:%3s.
in_serving_softirq:%3s. preempt_count=0x%x\n",               \
          (in_interrupt()?"yes":"no"), (in_irq()?"yes":"no"),               \
          (in_softirq()?"yes":"no"), (in_serving_softirq()?"yes":"no"), \
          (preempt_count() && 0xff));                                       \
  }                                                                         \
} while (0)
#endif
```

This, and other useful macros, can be found in the header provided – convenient.h ! (in a blog article).

Using the powerful 'crash' tool to investigate IRQs :

```
crash> irq
 IRQ   IRQ_DESC/_DATA      IRQACTION      NAME
  0   ffff880235808400  ffffffff81c1dac0  "timer"
  1   ffff880235808600  ffff88022f846400  "i8042"
  2   ffff880235808800      (unused)
  3   ffff880235808a00      (unused)
  4   ffff880235808c00      (unused)
  5   ffff880235808e00      (unused)
  6   ffff880235809000      (unused)
  7   ffff880235809200      (unused)
  8   ffff880235809400  ffff8800b4e72980  "rtc0"
  9   ffff880235809600  ffff88023276e300  "acpi"
...
crash>
crash> irqaction ffff8800b4e72980
struct irqaction {
  handler = 0xffffffff81054170 <hpet_rtc_interrupt>, << irq handler >>
  dev_id = 0xffff88022f85b800,
  percpu_dev_id = 0x0,
  next = 0x0,
  thread_fn = 0x0,       << handler is not a thread >>
  thread = 0x0,
  irq = 8,
  flags = 0,
  thread_flags = 0,
  thread_mask = 0,
  name = 0xffff8800b4e6a8b0 "rtc0",
  dir = 0xffff8800b4e72a00
}
crash>
 crash> irq 49          << iwlwifi >>
 IRQ   IRQ_DESC/_DATA      IRQACTION      NAME
 49   ffff8802301b6000  ffff88022f6f9600  "iwlwifi"
crash>
crash> irqaction ffff88022f6f9600
struct irqaction {
  handler = 0xffffffffc04a8fd0,
  dev_id = 0xffff880230774000,
  percpu_dev_id = 0x0,
  next = 0x0,
  thread_fn = 0xffffffffc04a7780,  << irq handler is a (kernel) thread ! >>
  thread = 0xffff88022f987010,    << thread task_struct ptr >>
  irq = 49,
  flags = 128,
  thread_flags = 0,
  thread_mask = 0,
  name = 0xffffffffc04bcbce "iwlwifi",
  dir = 0xffff8800b5283f00
}
crash>
```

Lets retrieve information on the thread!

```
crash> task 0xffff88022f987010 |grep comm
      start_comm = "\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000"
  comm = "irq/49-iwlwifi\000",
crash>
crash> task 0xffff88022f987010 |grep sched
  sched_class = 0xffffffff81815700 <rt_sched_class>,
  sched_task_group = 0xffffffff81ed1280 <root_task_group>,
  sched_info = {
  sched_reset_on_fork = 0,
  sched_contributes_to_load = 0,
crash> task 0xffff88022f987010 |grep prio
  prio = 49,
  static_prio = 120,
  normal_prio = 49,
  rt_priority = 50,
    prio = 49,
    prio_list = {
crash>
...
```

<<
Another example of a “hard” interrupt handler: from Intel's 10 Gbps “ixgb” device driver:

```
drivers/net/ethernet/intel/ixgb/ixgb_main.c
[...]
 err = request_irq(adapter->pdev->irq, ixgb_intr, irq_flags,
                      netdev->name, netdev); << Note: not a
                                          threaded handler! >>
[...]
static irqreturn_t
ixgb_intr(int irq, void *data)     << this is the hardirq 'top
                                      half' handler >>
{
    struct net_device *netdev = data;
    struct ixgb_adapter *adapter = netdev_priv(netdev);
    struct ixgb_hw *hw = &adapter->hw;
    u32 icr = IXGB_READ_REG(hw, ICR);

    if (unlikely(!icr))
        return IRQ_NONE;  /* Not our interrupt */

    if (unlikely(icr & (IXGB_INT_RXSEQ | IXGB_INT_LSC)))
        if (!test_bit(__IXGB_DOWN, &adapter->flags))
            mod_timer(&adapter->watchdog_timer, jiffies);

<< Send the packet up the protocol stack via NAPI >>
    if (napi_schedule_prep(&adapter->napi)) {

        /* Disable interrupts and register for poll. The flush
          of the posted write is intentionally left out.
        */

        IXGB_WRITE_REG(&adapter->hw, IMC, ~0);
        __napi_schedule(&adapter->napi);
    }
    return IRQ_HANDLED;
}
[...]
>>
```

Interrupt Control

The Linux kernel implements a family of interfaces for manipulating the state of interrupts on a machine. These interfaces enable you to disable the interrupt system for the current processor or mask out an interrupt line for the entire machine. These routines are all architecture-dependent and can be found in <asm/system.h> and <asm/irq.h>. See Table 7.2, later in this chapter, for a complete listing of the interfaces.

Reasons to control the interrupt system generally boil down to needing to provide synchronization. By disabling interrupts, you can guarantee that an interrupt handler will not preempt your current code. Moreover, disabling interrupts also disables kernel preemption.

Neither disabling interrupt delivery nor disabling kernel preemption provides any protection from concurrent access from another processor, however. Because Linux supports multiple processors, kernel code more generally needs to obtain some sort of lock to prevent another processor from accessing shared data simultaneously. These locks are often obtained in conjunction with disabling local interrupts. The lock provides protection against concurrent access from another processor, whereas disabling interrupts provides protection against concurrent access from a possible interrupt handler. Chapters 9 and 10 discuss the various problems of synchronization and their solutions. Nevertheless, understanding the kernel interrupt control interfaces is important.

Disabling and Enabling Interrupts

To disable interrupts locally for the current processor (and only the current processor) and then later reenable them, do the following:

```
local_irq_disable();
/* interrupts are disabled .. */
local_irq_enable();
```

--*snip*--

The local_irq_disable() routine is dangerous if interrupts were already disabled prior to its invocation. The corresponding call to local_irq_enable() unconditionally enables interrupts, despite the fact that they were off to begin with.
Instead, a mechanism is needed to (save and subsequently) restore interrupts to a previous state. This is a common concern because a given code path in the kernel can be reached both with and without interrupts enabled, depending on the call chain.

--*snip*--

```
unsigned long flags;
local_irq_save(flags);
/* interrupts are now disabled */
/* ... */
```

```
local_irq_restore(flags); /* interrupts are restored to their
                             previous state */
```
--snip--

All the previous functions can be called from both interrupt and process context.

Disabling a Specific Interrupt Line

In the previous section, we looked at functions that disable all interrupt delivery for an entire processor. In some cases, it is useful to disable only a specific interrupt line for the entire system. This is called masking out an interrupt line. As an example, you might want to disable delivery of a device's interrupts before manipulating its state. Linux provides four interfaces for this task:

```
void disable_irq(unsigned int irq);
void disable_irq_nosync(unsigned int irq);
void enable_irq(unsigned int irq);
void synchronize_irq(unsigned int irq);
```

The first two functions disable a given interrupt line in the interrupt controller. This disables delivery of the given interrupt to all processors in the system. Additionally, the disable_irq() function does not return until any currently executing handler completes. Thus, callers are assured not only that new interrupts will not be delivered on the given line, but also that any already executing handlers have exited.

The function disable_irq_nosync() does not wait for current handlers to complete.

The function synchronize_irq() waits for a specific interrupt handler to exit, if it is executing, before returning.

Calls to these functions nest. For each call to disable_irq() or disable_irq_nosync() on a given interrupt line, a corresponding call to enable_irq() is required. Only on the last call to enable_irq() is the interrupt line actually enabled.

For example, if disable_irq() is called twice, the interrupt line is not actually reenabled until the second call to enable_irq(). All three of these functions can be called from interrupt or process context and do not sleep. If calling from interrupt context, be careful! You do not want, for example, to enable an interrupt line while you are handling it. (Recall that the interrupt line of a handler is masked out while it is serviced.)

It would be rather rude to disable an interrupt line shared among multiple interrupt handlers. Disabling the line disables interrupt delivery for all devices on the line. Therefore, drivers for newer devices tend not to use these interfaces.[3] Because PCI devices have to support interrupt line sharing by specification, they should not use these interfaces at all. Thus, disable_irq() and friends are found more often in drivers for older legacy devices, such as the PC parallel port.

[3] *Many older devices, particularly ISA devices, do not provide a method of obtaining whether they generated an interrupt. Therefore, often interrupt lines for ISA devices cannot be shared. Because the PCI specification mandates the sharing of interrupts, modern PCI-based devices support interrupt sharing. In contemporary computers, nearly all interrupt lines can be shared.*

<<

- This discussion also gives rise to the question: how does hardware-specific IRQs as seen by the interrupt controller chip (PIC / GIC) get mapped to Linux IRQ numbers?
  This is done via an interrupt number mapping library feature called "IRQ Domains". See the irq-domain documentation file for more information.

- On SMP, how are interrupts delivered to individual processor cores? Linux uses an IRQ-affinity model; the default is to have all cores enabled to handle an IRQ. This can be changed via proc (as root user). See the IRQ-affinity documentation file for more details on how to do so.

>>

Status of the Interrupt System

It is often useful to know the state of the interrupt system (for example, whether interrupts are enabled or disabled) or whether you are currently executing in interrupt context. The macro irqs_disabled(), defined in <asm/system.h>, returns nonzero if the interrupt system on the local processor is disabled. Otherwise, it returns zero.
Two macros, defined in <linux/hardirq.h>, provide an interface to check the kernel's current context.They are

```
in_interrupt()
in_irq()
```

The most useful is the first: It returns nonzero if the kernel is performing any type of interrupt handling.This includes either executing an interrupt handler or a bottom half handler. The macro in_irq() returns nonzero only if the kernel is specifically executing an interrupt handler (meaning, a "top half" handler).

More often, you want to check whether you are in process context. That is, you want to ensure you are not in interrupt context. This is often the case because code wants to do something that can only be done from process context, such as sleep. **If in_interrupt() returns zero, the kernel is in process context. Treat in_interrupt() return value as a boolean.**

Yes, the names are confusing and do little to impart their meaning.
Table 7.2 is a summary of the interrupt control methods and their description.

Table 7.2 ***Interrupt Control Methods***

Function	*Description*
local_irq_disable()	Disables local interrupt delivery
local_irq_enable()	Enables local interrupt delivery
local_irq_save()	Saves the current state of local interrupt delivery and then disables it
local_irq_restore()	Restores local interrupt delivery to the given state
disable_irq()	Disables the given interrupt line and ensures no handler on the line is executing before returning
disable_irq_nosync()	Disables the given interrupt line
enable_irq()	Enables the given interrupt line

irqs_disabled()	Returns nonzero if local interrupt delivery is disabled; other- wise returns zero
in_interrupt()	Returns nonzero if in interrupt context and zero if in process context
in_irq()	Returns nonzero if currently executing an interrupt handler and zero otherwise
in_softirq()	Returns nonzero if currently executing in a softirq handler and zero otherwise.

<< LDD3 >>

Tasklets and Bottom-Half Processing

One of the main problems with interrupt handling is how to perform longish tasks within a handler. Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long. **These two needs (work and speed) conflict** with each other, leaving the driver writer in a bit of a bind.

Linux (along with many other systems) resolves this problem **by splitting the interrupt handler into two halves**. The so-called **top half** is the routine that **actually responds to the interrupt** -- the one you register with *request_irq*.

The **bottom half** is a routine **that is scheduled by the top half to be executed later, at a safer time**. The use of the term bottom half in the 2.4 kernel can be a bit confusing, in that it can mean either the second half of an interrupt handler or one of the mechanisms used to implement this second half, or both. When we refer to a bottom half we are speaking generally about a bottom half; the old Linux bottom-half implementation is referred to explicitly with the acronym BH.

But what is a bottom half useful for?

The big difference between the top-half handler and the bottom half is that **all interrupts are enabled during execution of the bottom half** -- that's why it runs at a safer time.

In the typical scenario, the **top half saves device data to a device-specific buffer, schedules its bottom half, and exits**: this is very fast. The bottom half then performs whatever other work is required, such as awakening processes, starting up another I/O operation, and so on. This setup permits the top half to service a new interrupt while the bottom half is still working.

Every serious interrupt handler is split this way. For instance, when a network interface reports the arrival of a new packet, the handler just retrieves the data and pushes it up to the protocol layer; actual processing of the packet is performed in a bottom half.

One thing to keep in mind with **bottom-half processing is that all of the restrictions that apply to interrupt handlers also apply to bottom halves**. Thus, bottom halves **cannot sleep, cannot access user space, and cannot invoke the scheduler**.

<<
Technically, softirq's _do_ run in *an* interrupt context - the "softirq" context; it's just that it's not "hard-irq" context (which is the context when a hardware interrupt occurs).

So, in a softirq handler, in terms of the 'lookup' macros Linux provides:

in_interrupt*: yes | in_irq: no | in_softirq: yes | in_serving_softirq: yes.

Yet, as clearly mentioned above, "**all of the restrictions that apply to interrupt handlers also apply to bottom halves**. Thus, **bottom halves cannot sleep, cannot access user space, and cannot invoke the scheduler."**

(Participants can try out the 'irqsee' kernel module provided).
>>

<<
* From *include/linux/hardirq.h :*
```
...
#define hardirq_count() (preempt_count() & HARDIRQ_MASK)
#define softirq_count() (preempt_count() & SOFTIRQ_MASK)
#define irq_count()     (preempt_count() & (HARDIRQ_MASK |
                            SOFTIRQ_MASK | NMI_MASK))

/*
 * Are we doing bottom half or hardware interrupt processing?
 * Are we in a softirq context? Interrupt context?
 * in_softirq - Are we currently processing softirq or have bh disabled?
 * in_serving_softirq - Are we currently processing softirq?
 */
#define in_irq()             (hardirq_count())
#define in_softirq()         (softirq_count())
#define in_interrupt()       (irq_count())
#define in_serving_softirq() (softirq_count() & SOFTIRQ_OFFSET)
…
<linux/preempt.h>
...
#define preempt_count() (current_thread_info()->preempt_count)
```

>>

<<
Note:
preempt_count is a bitmask. To fetch the actual preemption counter value (i.e. the integer that's incremented every time a (spin)lock is taken and/or the kernel enters a non-preemptible section, and decremented when unlocked and/or the kernel enters a preemptible section), use:

```
preempt_count() & 0xFF
```

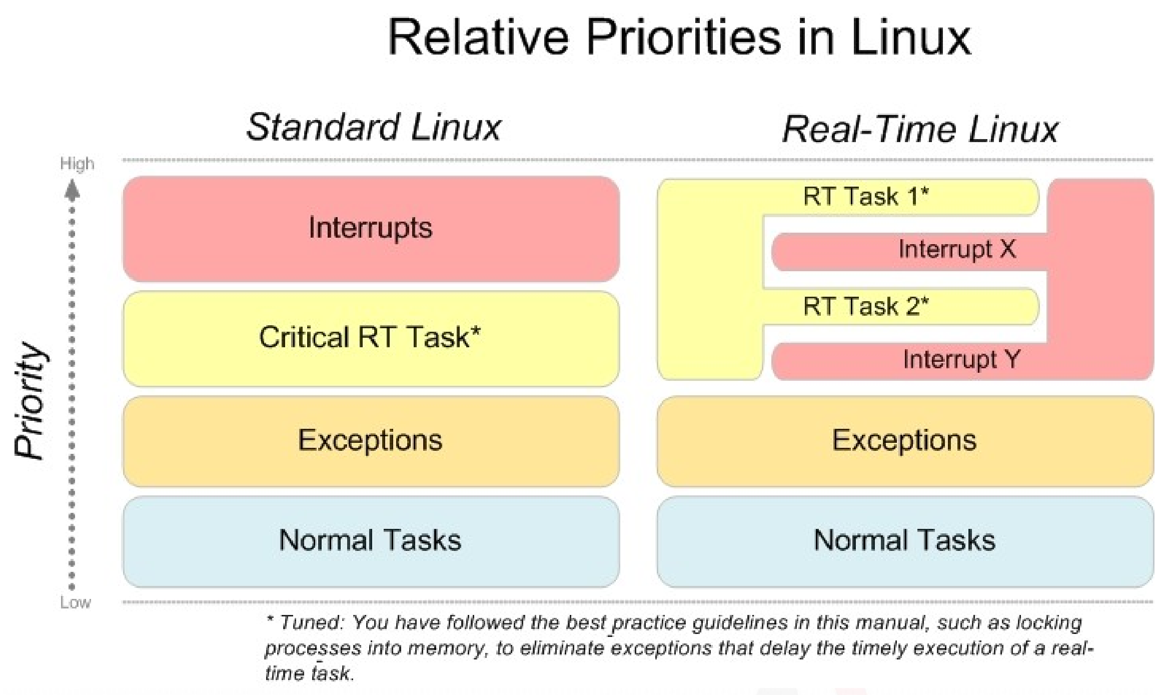
>>

An FAQ: **Which context are softirq and tasklet in?**

<<
OPTIONAL / FYI

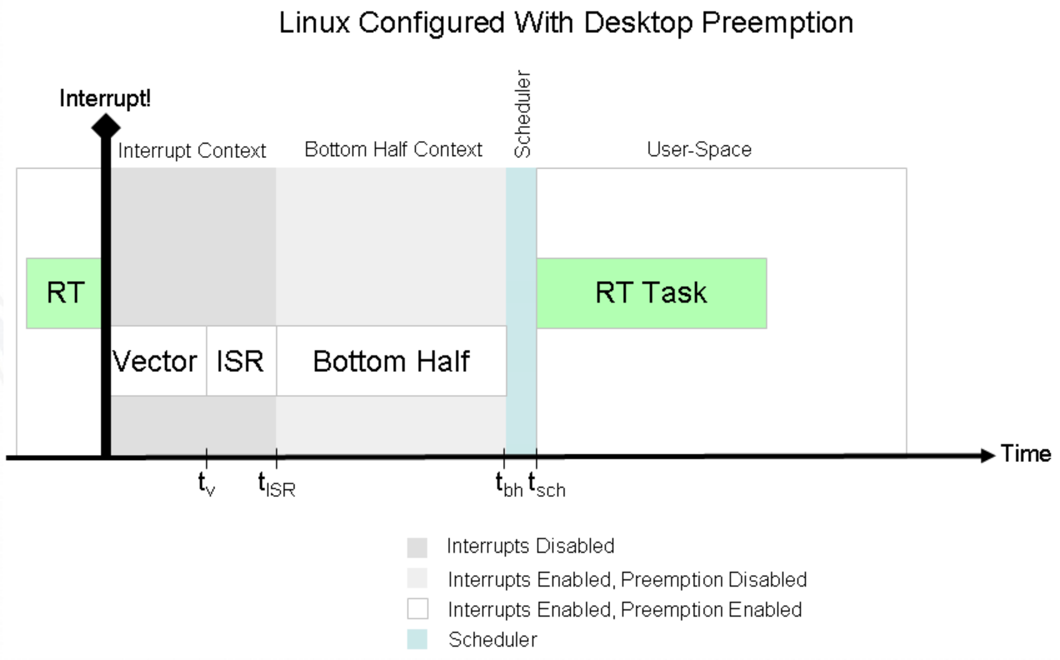
An informative resource on Linux interrupt-handling with emphasis on (hard) real-time:
https://export.writer.zoho.com/public/rreginelli/Chapter-10---Interrupts1/fullpage
(See the diagrams).

…


Relative Priorities in Linux
Standard Linux
Real-Time Linux
Priority
High
Low
Interrupts
Critical RT Task*
Exceptions
Normal Tasks
RT Task 1*
Interrupt X
RT Task 2*
Interrupt Y
Exceptions
Normal Tasks
* Tuned: You have followed the best practice guidelines in this manual, such as locking processes into memory, to eliminate exceptions that delay the timely execution of a real-time task.

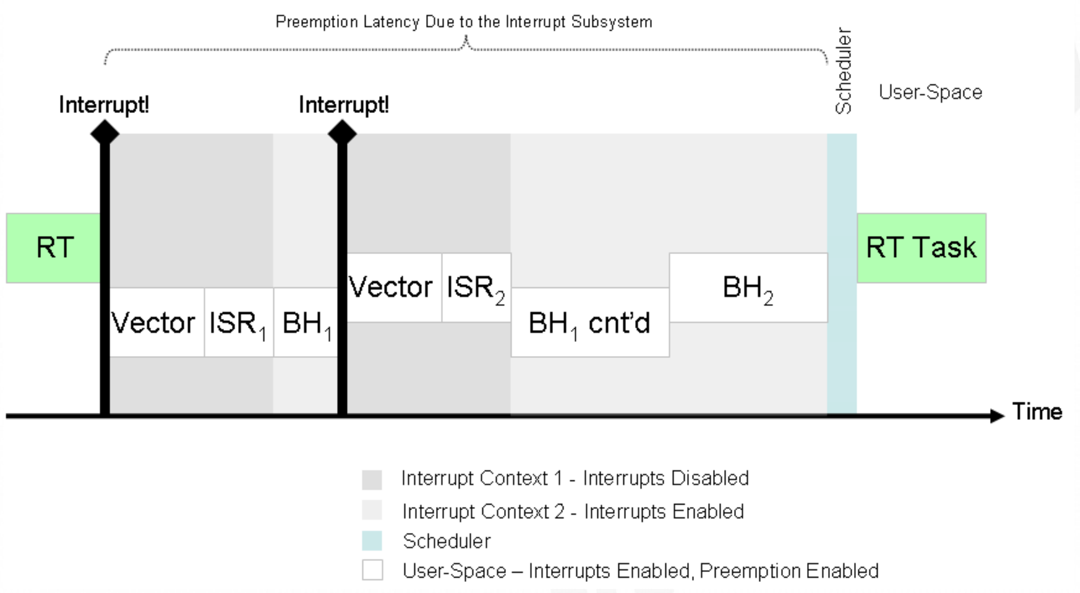
…

The diagram below depicts a simplified timeline for handling a single interrupt in the non-real-time kernel.



…
Handling a single interrupt is generally very fast in Linux and does not contribute very much to preemption latency. Preemption latency due to the interrupt subsystem can become a factor, however, when there is a flood of interrupts. The diagram below depicts the scenario where two interrupts fire in close succession.

Linux Configured With Desktop Preemption


Preemption Latency Due to the Interrupt Subsystem
Scheduler
User-Space
Interrupt!
Interrupt!
RT
Vector
ISR1
BH1
Vector
ISR2
BH1 cnt'd
BH2
RT Task
Time
Interrupt Context 1 - Interrupts Disabled
Interrupt Context 2 - Interrupts Enabled
Scheduler
User-Space – Interrupts Enabled, Preemption Enabled

…
The Linux kernel community took care to prevent starvation and forward-progress of user-space applications under heavy interrupt load with ksoftirqd. ksoftirqd is a set of per-CPU threads that are called on to process softirqs when the system is overloaded with softirqs. ksoftirqd is activated when a softirq is already being processed and it is reactivated. With the softirqs being handled by threads, other user-space tasks are given the opportunity to run and compete for the CPU's attention. In this design, if user-space is idle, ksoftirqd can execute immediately. If there are tasks ready to run in user-space, they are allowed to compete for time with ksoftirqd.

…

<<
Source

Tasklets Bottom half Scheduling

--snip--

Each CPU has queues for scheduled (high-priority and normal) tasklets.

When a CPU is about to return to user space from an interrupt or from a system call, it checks for scheduled tasklets, and executes them. The same checks are done again after a tasklet has finished. (If there are too many scheduled tasklets, they are not all executed at once, but moved into a kernel thread, ksoftirqd; but the principle stays the same.)

Therefore, a tasklet will never interrupt another tasklet, but a high-priority tasklet will be executed before any scheduled 'normal' tasklets.

answered 23 hours ago
CL. 36.3k 4 17 40

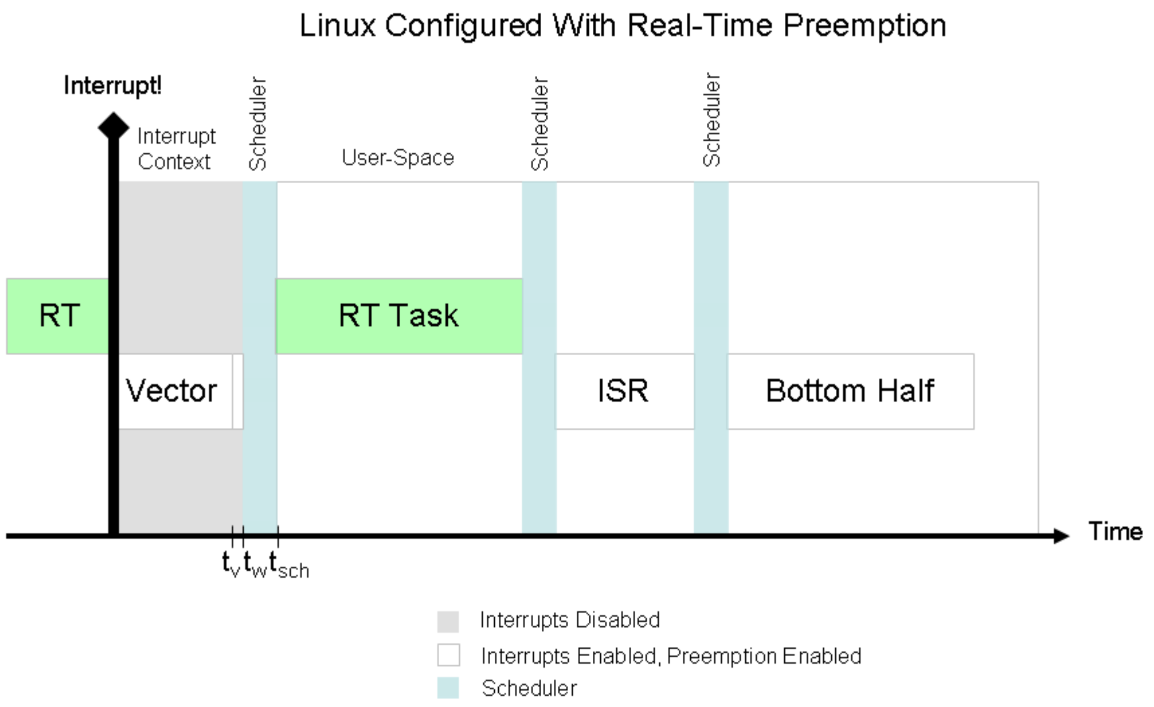
FYI, conversely, a hardware interrupt (or top-half) <i>can</i> interrupt or preempt a bottom-half routine. – kaiwan.
>>

Real-Time Linux: Processing an Interrupt

To mitigate delays to scheduling the highest priority task, real-time Linux runs interrupts into thread context. Instead of handling the entire interrupt in interrupt context, only a short function executes to awaken the kernel thread that contains the ISR. The ISR is then scheduled to run according to its priority.

In real-time Linux, one thread is created per interrupt line. Each thread is assigned the FIFO scheduling policy and a real-time priority of 50. Shared interrupts are handled by a single thread. All interrupt activity, with the exception of the timer interrupt, execute in thread context by default. All bottom halves (softirqs) also run in thread context with the FIFO scheduling policy and a real-time priority of 50.

The diagram below shows the improved determinism brought to Linux with real-time preemption enabled in the kernel. In this scenario, the system designer has assigned a higher priority to the real-time task than to the handlers for which the interrupt has just fired. The high priority real-time task is held off from executing for only a very short period of time while the kernel masks the interrupt line and wakes the appropriate ISR thread.



…

One of the most important steps in deploying a real-time system is correctly prioritizing the various high priority,time-critical tasks (interrupt handlers, system tasks, your real-time application) against each other. You need to be especially careful when the kernel is configured for real-time because interrupt handlers, with their high priority and FIFO scheduling policy, can hold off lower priority tasks for as long as they choose. The kernel does not intervene, like it does in the non-real-time kernel, to prevent the starvation of user-space tasks. It is also easy to create a deadlock situation where a high priority consumer thread starves a producer thread on which it depends. There is a delicate inter-connectedness between all the new real-time threads that are created (including interrupt handlers) in the real-time configuration of the kernel, and they need to be correctly balanced against the deadlines of your real-time task.

...

All rights with the original author(s).

>>

The Linux kernel has two different mechanisms that may be used to implement bottom-half processing. **Tasklets** are often the preferred mechanism for bottom-half processing; they are very fast, but all tasklet code must be **atomic**.

Tasklets, bottom-halves and softirqs are all examples of the **"deferred function"** mechanism of the Linux kernel. Bottom Halves are built upon tasklets; tasklets are built on softirqs.

Tasklets are the preferred way to implement deferrable functions in I/O devices. Tasklets are built over two softirqs (seen later) – HI_SOFTIRQ and TASKLET_SOFTIRQ. Several tasklets may be associated with the same IRQ, each tasklet having it's own function.

SIDEBAR - Work Queues and Tasklets

The alternative to tasklets is **work queues** (2.6 equivalent to 2.4's task queues).

The main difference between deferrable functions and work queues is that deferrable functions (like tasklets) run in interrupt context while functions *in work queues run in process context*. Running in process context is the only way to execute functions that *can block*, because no process switch can take place in interrupt context.

Neither deferrable functions nor functions in a work queue can access the User Mode address space of a process. In fact, a deferrable function cannot make any assumption about the process that is currently running when it is executed. On the other hand, a function in a work queue is executed by a kernel thread (typically keventd, when one uses the predefined work queue functionality), so there is no User Mode address space to access.

In most cases, creating a whole set of worker threads in order to run a function is overkill. Therefore, the kernel offers a predefined work queue called ``events", which can be freely used by every kernel developer. The predefined work queue is nothing more than a standard work queue that may include functions of different kernel layers and I/O drivers.

From LDD3:
“ A device driver, in many cases, does not need its own workqueue. If you only submit tasks to the queue occasionally, it may be more efficient to simply use the shared, default workqueue that is provided by the kernel. If you use this queue, however, you must be aware that you will be sharing it with others.

Among other things, that means that you should not monopolize the queue for long periods of time (no long sleeps), and it may take longer for your tasks to get their turn in the processor."

One way to see runtime usage of workqueues is to use the trace-cmd front-end to the ftrace facility:

```
# trace-cmd record -e workqueue
Hit Ctrl^C to stop recording
^C
...
# trace-cmd report
…
cpus=8
          <idle>-0     [005]  7995.171325: workqueue_queue_work: work
struct=0xffffffff81a64360 function=console_callback workqueue=0xffff8801376eb000
req_cpu=5 cpu=5
          <idle>-0     [005]  7995.171327: workqueue_activate_work: work struct
0xffffffff81a64360
          <idle>-0     [005]  7995.171336: workqueue_queue_work: work
struct=0xffff880133e5c110 function=flush_to_ldisc workqueue=0xffff8801376eb000
req_cpu=5 cpu=5
          <idle>-0     [005]  7995.171336: workqueue_activate_work: work struct
0xffff880133e5c110
     kworker/5:1-47    [005]  7995.171409:
…
…
      <idle>-0     [001]  7998.694408: workqueue_activate_work: work struct
0xffff88009a8cc110
     kworker/1:0-8     [001]  7998.694420: workqueue_execute_start: work struct
0xffff8800bb251188: function do_dbs_timer
     kworker/1:0-8     [001]  7998.694424: workqueue_execute_end: work struct
0xffff8800bb251188
     kworker/1:0-8     [001]  7998.694425: workqueue_execute_start: work struct
0xffff88009a8cc110: function flush_to_ldisc
     kworker/1:0-8     [001]  7998.694471: workqueue_execute_end: work struct
0xffff88009a8cc110
#
```

Tasklets are a special function that may be scheduled to run, **in interrupt context**, at a system-determined safe time. However, if your driver has **multiple tasklets**, they must employ some sort of **locking** (spinlocks) to avoid conflicting with each other.

Tasklets are also guaranteed to run on the same CPU as the function that first schedules them. An interrupt handler can thus be secure that a tasklet will not begin executing before the handler has completed. However, another interrupt can certainly be delivered while the tasklet is running, so locking between the tasklet and the interrupt handler may still be required.

Tasklets must be declared and initialized with the *tasklet_init()* function:

```
tasklet_init(struct tasklet_struct *t,
              void (*func)(unsigned long), unsigned long data);
```

“t” is the new tasklet_struct data structure which you must allocate memory for (it will be initialized internally by this function), “func” is the function that is called to execute the tasklet (it takes one unsigned long argument and returns void), and “data” is an unsigned long value to be passed to the tasklet function.

The function ***tasklet_schedule*** (or *tasklet_hi_schedule*) is used to activate a tasklet for running.

```
void tasklet_schedule(struct tasklet_struct *t);
```

Also available:

```
void tasklet_hi_schedule(struct tasklet_struct *t);
```

A given tasklet cannot run simultaneously on different CPUs. Different CPUs can run *different* tasklets, however.

Also, realize that: a top-half IRQ handler (say, IRQ 'n') can run and schedule a tasklet. The tasklet is now running; while it is executing, nothing prevents the same top-half IRQ 'n' occuring; thus, the top-half handler and bh tasklet can now overlap - hence the need to protect any shared data between them using spinlocks.

Softirqs

Softirqs are pre-defined (static) deferred processing (“bottom-half”) mechanisms that usually run in an interrupt context. After handling a hardware interrupt, the first thing kernel code does is check for pending softirqs.

```
do_IRQ() → handle_irq() → <Intr top-half runs> → irq_exit() →
  invoke_softirq() → do_softirq() → <Intr bottom half runs: tasklet/softirq>
```

Linux uses several kinds of software IRQs: (*<linux/interrupt.h>*)

Softirq Type	**Prio rity**	**Comment**
HI_SOFTIRQ	0	Handles high priority tasklets
TIMER_SOFTIRQ	1	Timers
NET_TX_SOFTIRQ	2	Transmits packets to network cards
NET_RX_SOFTIRQ	3	Retrieves packets from network cards
BLOCK_SOFTIRQ (+ BLOCK_IOPOLL_SOFTIRQ)	4	Block devices
TASKLET_SOFTIRQ	5	Handles regular tasklets
SCHED_SOFTIRQ	6	Scheduler
HRTIMER_SOFTIRQ	7	HRT (High resolution timers)
RCU_SOFTIRQ	8	RCU locking

include/linux/interrupt.h

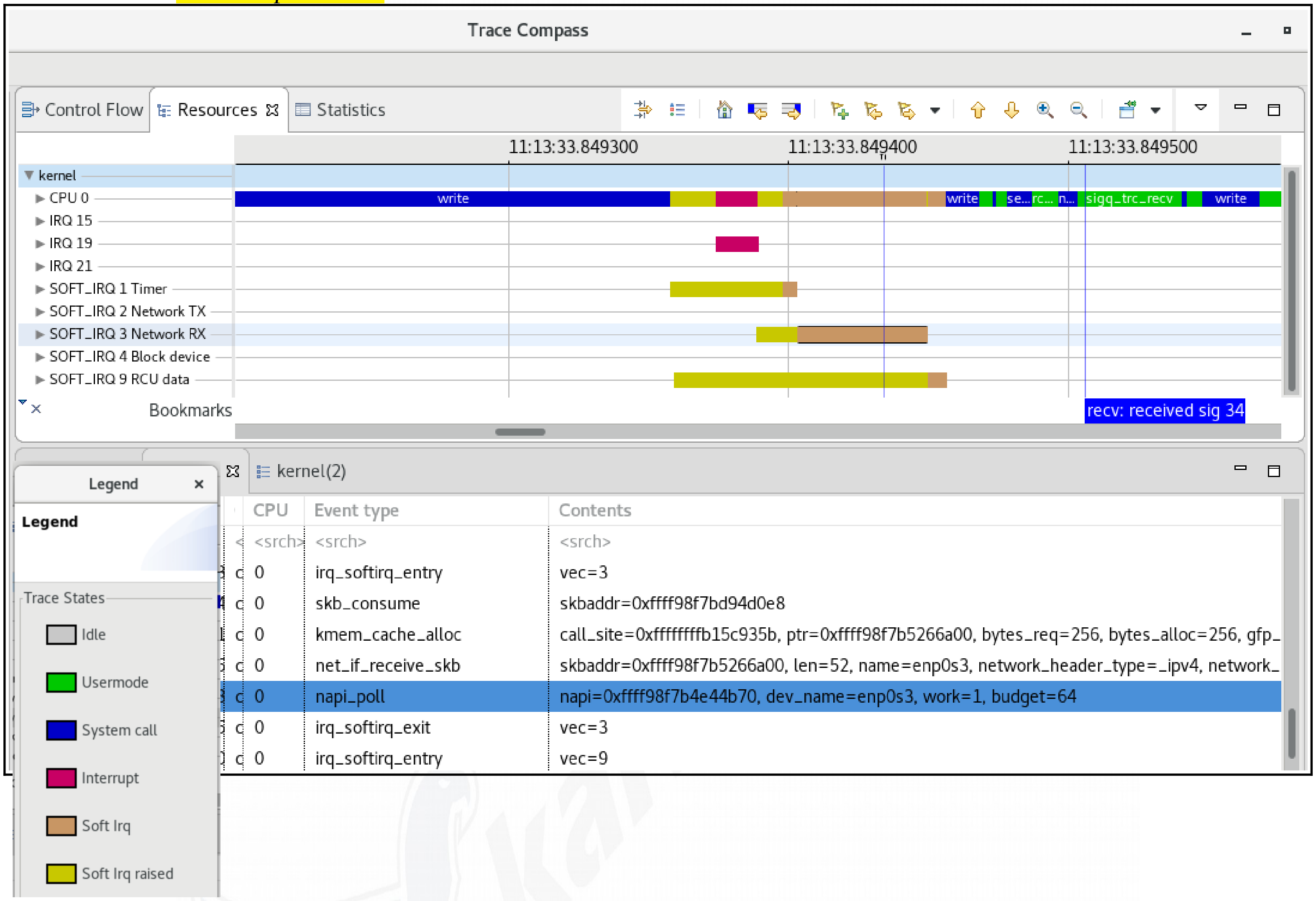
```
...
/* PLEASE, avoid to allocate new softirqs, if you need not _really_ high
   frequency threaded job scheduling. For almost all the purposes
   tasklets are more than enough. F.e. all serial device BHs et
   al. should be converted to tasklets, not to softirqs.
 */

enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    BLOCK_SOFTIRQ,
    BLOCK_IOPOLL_SOFTIRQ,
    TASKLET_SOFTIRQ,
    SCHED_SOFTIRQ,
        HRTIMER_SOFTIRQ,
        RCU_SOFTIRQ,    /* Preferable RCU should always be the last softirq */

        NR_SOFTIRQS
};
...
```

```
/ # cat /proc/softirqs     # On an OMAP4 board running Linux 3.1.5
                    CPU0       CPU1
          HI:          0          0
       TIMER:       2378        765
      NET_TX:          0          0
      NET_RX:          0          0
       BLOCK:          0          0
BLOCK_IOPOLL:          0          0
     TASKLET:        382          1
       SCHED:       1976        507
     HRTIMER:          0          0
         RCU:       2453       1740
/ #
```

On an Android phone:

```
herolte:/ $ cat /proc/softirqs
                    CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
          HI:      25458          5       3326          0       7479       6880         44          4
       TIMER:    2900240    2777259    2573835    2309919     892684     835091     125929     123222
      NET_TX:     216236      35164      30536      19417      49293       9695       1059       1009
      NET_RX:     101548      58998      45216         80     172988        124         41         40
       BLOCK:    4683424        264       3460        148     688744     680112     266198     265760
BLOCK_IOPOLL:      25454          5       3326          0       7479       6882         44          4
     TASKLET:    1879043       1212       3817        763       9111       7788         94         58
       SCHED:    2743332    2638272    2471962    2271197     655722     594580      96409      96611
     HRTIMER:       7167       6037       5630       5125        686        644         43         61
         RCU:    3236520    2978406    2602809    2023796    1702240    1673544     409153     388072
herolte:/ $
```

Screenshot below: part of a *LTTng* session capture, visualized using the superb *tracecompass* GUI:

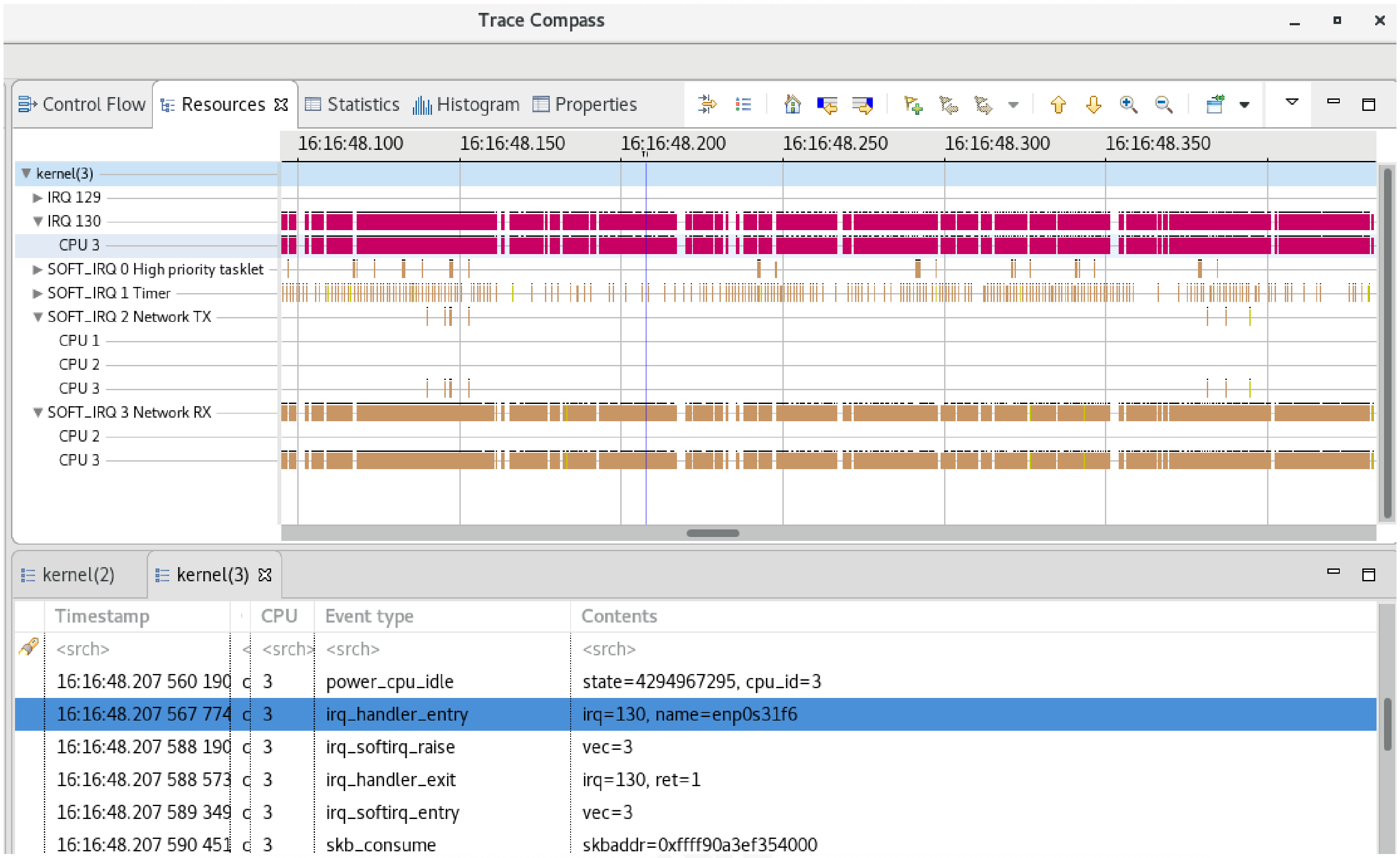


CPU	Event type	Contents
<srch>	<srch>	<srch>
0	irq_softirq_entry	vec=3
0	skb_consume	skbaddr=0xffff98f7bd94d0e8
0	kmem_cache_alloc	call_site=0xffffffffb15c935b, ptr=0xffff98f7b5266a00, bytes_req=256, bytes_alloc=256, gfp_
0	net_if_receive_skb	skbaddr=0xffff98f7b5266a00, len=52, name=enp0s3, network_header_type=_ipv4, network_
0	napi_poll	napi=0xffff98f7b4e44b70, dev_name=enp0s3, work=1, budget=64
0	irq_softirq_exit	vec=3
0	irq_softirq_entry	vec=9

Legend

Trace States	
⬜	Idle
🟩	Usermode
🟦	System call
🟥	Interrupt
🟫	Soft Irq
🟨	Soft Irq raised

#2



Timestamp	CPU	Event type	Contents
<srch>	<srch>	<srch>	<srch>
16:16:48.207 560 190	3	power_cpu_idle	state=4294967295, cpu_id=3
16:16:48.207 567 774	3	irq_handler_entry	irq=130, name=enp0s31f6
16:16:48.207 588 190	3	irq_softirq_raise	vec=3
16:16:48.207 588 573	3	irq_handler_exit	irq=130, ret=1
16:16:48.207 589 349	3	irq_softirq_entry	vec=3
16:16:48.207 590 451	3	skb_consume	skbaddr=0xffff90a3ef354000

The *crash* tool can show us the defined *softirqs*:

```
crash> irq -b
SOFTIRQ_VEC      ACTION
    [0]      ffffffff81075160  <tasklet_hi_action>
    [1]      ffffffff8107e540  <run_timer_softirq>
    [2]      ffffffff816826c0  <net_tx_action>
    [3]      ffffffff81683850  <net_rx_action>
    [4]      ffffffff81374b20  <blk_done_softirq>
    [5]      ffffffff813752e0  <blk_iopoll_softirq>
    [6]      ffffffff81075260  <tasklet_action>
    [7]      ffffffff810b44f0  <run_rebalance_domains>
    [8]      ffffffff81098e00  <run_hrtimer_softirq>
    [9]      ffffffff810da410  <rcu_process_callbacks>
crash>
```

Softirq's are statically assigned at compile-time; they ***cannot*** *be dynamically registerd and de-registered like a tasklet.*
open_softirq() registers a softirq.

File : kernel/softirq.c

```
...
 static struct softirq_action softirq_vec[NR_SOFTIRQS]
__cacheline_aligned_in_smp;
...
void open_softirq(int nr, void (*action)(struct softirq_action *))
{
    softirq_vec[nr].action = action;
}
```

Using cscope:

```
Functions calling this function: open_softirq

  File          Function          Line
0 blk-iopoll.c  blk_iopoll_setup   223 open_softirq(BLOCK_IOPOLL_SOFTIRQ,
                                         blk_iopoll_softirq);

1 blk-softirq.c blk_softirq_init   171 open_softirq(BLOCK_SOFTIRQ,
                                         blk_done_softirq);

2 hrtimer.c     hrtimers_init     1741 open_softirq(HRTIMER_SOFTIRQ,
                                         run_hrtimer_softirq);

3 rcutree.c     rcu_init          2076 open_softirq(RCU_SOFTIRQ,
                                         rcu_process_callbacks);

4 sched.c       sched_init        8141 open_softirq(SCHED_SOFTIRQ,
                                         run_rebalance_domains);

5 softirq.c     softirq_init       736 open_softirq(TASKLET_SOFTIRQ,
                                         tasklet_action);

6 softirq.c     softirq_init       737 open_softirq(HI_SOFTIRQ,
                                         tasklet_hi_action);

7 timer.c       init_timers       1740 open_softirq(TIMER_SOFTIRQ,
                                         run_timer_softirq);

8 dev.c         net_dev_init      6500 open_softirq(NET_TX_SOFTIRQ,
                                         net_tx_action);

9 dev.c         net_dev_init      6501 open_softirq(NET_RX_SOFTIRQ,
                                         net_rx_action);
```

It's very interesting to see **Ftrace** output (in 'latency format'); clearly shows up interrupt paths and type:

Help for interpreting the output of the ftrace latency output format

```
# -----------------------------------------------
# Help for interpreting the output of the latency:
# from Documentation/trace/ftrace.txt
--snip--
# The next lines after the header are the trace itself. The header
# explains which is which.
#
#   cmd: The name of the process in the trace.
#
#   pid: The PID of that process.
#
#   CPU#: The CPU which the process was running on.
#
#   irqs-off: 'd' interrupts are disabled. '.' otherwise.
#           Note: If the architecture does not support a way to
#                 read the irq flags variable, an 'X' will always
#                 be printed here.
#
#  need-resched: 'N' task need_resched is set, '.' otherwise.
#
#   hardirq/softirq:
#       'H' - hard irq occurred inside a softirq.
#       'h' - hard irq is running
#       's' - soft irq is running
#       '.' - normal context.
#
#   preempt-depth: The level of preempt_disabled
#
# The above is mostly meaningful for kernel developers.
#
#   time: When the latency-format option is enabled, the trace file
#       output includes a timestamp relative to the start of the
#       trace. This differs from the output when latency-format
#       is disabled, which includes an absolute timestamp.
#
#  delay: This is just to help catch your eye a bit better. And
#        needs to be fixed to be only relative to the same CPU.
#        The marks are determined by the difference between this
#        current trace and the next trace.
#         '!' - greater than preempt_mark_thresh (default 100)
#         '+' - greater than 1 microsecond
#         ' ' - less than or equal to 1 microsecond.
#
#  The rest is the same as the 'trace' file.
#
--snip--
```

A sample ftrace session on an ARM v7 (emulated using QEMU) running Linux 3.10.24 :

```
--snip--

# tracer: function_graph
#
# function_graph latency trace v1.1.5 on 3.10.24
# --------------------------------------------------------------------
# latency: 0 us, #44942/144640, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:1)
#    -----------------
#    | task: -0 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
#
#                        _-----=> irqs-off
#                       / _----=> need-resched
#                      | / _---=> hardirq/softirq
#                      || / _--=> preempt-depth
#                      ||| /
# CPU  TASK/PID        ||||  DURATION                  FUNCTION CALLS
# |     |    |         ||||   |   |                     |   |   |   |
           << unfortunately, the indented function names wrap around sometimes
>>
  0)   sleep-533    |  d.s5  3.292 us    |      << notice we're in softirq
context >>
wakeup_preempt_entity();
  0)   sleep-533    |  d.s5  4.292 us    |
resched_task();
  0)   sleep-533    |  dNs5! 169.791 us  | << notice the TIF_NEED_RESCHED flag
                                   set implying we need to reschedule ASAP! >>
}
  0)   sleep-533    |  dNs5! 224.209 us  |
}
  0)   sleep-533    |  dNs5! 279.875 us  |
}
  0)   sleep-533    |  dNs5! 833.375 us  |
}
  0)   sleep-533    |  dNs5              |
_raw_spin_unlock() {
  0)   sleep-533    |  dNs5  2.583 us    |
sub_preempt_count();
  0)   sleep-533    |  dNs4+ 52.416 us   | }

--snip--

  0)   sleep-533    |  dN.2! 6794.292 us |                         }
  0)   sleep-533    |  dN.2! 6946.542 us |                       }
  0)   sleep-533    |   <========== |       << the arrow denotes a switch of
ctx >>
  0)   sleep-533    |  .N..              |                          __schedule() {
  0)   sleep-533    |  .N..  1.667 us    |
add_preempt_count();
  0)   sleep-533    |  .N.1  4.167 us    |
rcu_note_context_switch();
  0)   sleep-533    |  .N.1              |
_raw_spin_lock_irq() {
  0)   sleep-533    |  dN.1  2.209 us    |
add_preempt_count();
  0)   sleep-533    |  dN.2+ 55.833 us   |                      }
  0)   sleep-533    |  dN.2  1.750 us    |
update_rq_clock();
 0)   sleep-533    |  dN.2              |
put_prev_task_fair() {
 0)   sleep-533    |  dN.2  2.292 us    |
update_curr();

                       _-----=> irqs-off
                      / _----=> need-resched
                     | / _---=> hardirq/softirq
                     || / _--=> preempt-depth
                     ||| /
CPU  TASK/PID        ||||  DURATION                  FUNCTION CALLS
 |     |    |        ||||   |   |                     |   |   |   |

 0)   sleep-533    |  dN.2  2.875 us    |
__enqueue_entity();
 0)   sleep-533    |  dN.2! 104.833 us  |                         }
 0)   sleep-533    |  dN.2              |
pick_next_task_fair() {
 0)   sleep-533    |  dN.2  2.750 us    |
wakeup_preempt_entity();
 0)   sleep-533    |  dN.2  2.833 us    |
clear_buddies();
 0)   sleep-533    |  dN.2  4.583 us    |
__dequeue_entity();
 0)   sleep-533    |  dN.2! 165.083 us  |                         }
 0)   sleep-533    |  d..2              | atomic_notifier_call_chain() {
 0)   sleep-533    |  d..2              |  << TIF_NEED_RESCHED flag cleared >>
__atomic_notifier_call_chain() {
 0)   sleep-533    |  d..2  1.958 us    |
__rcu_read_lock();
 0)   sleep-533    |  d..2              |
notifier_call_chain() {
 0)   sleep-533    |  d..2  2.875 us    |
vfp_notifier();
 0)   sleep-533    |  d..2+ 49.208 us   |                             }
 0)   sleep-533    |  d..2  2.250 us    |
__rcu_read_unlock();
 0)   sleep-533    |  d..2! 189.292 us  |                           }
 0)   sleep-533    |  d..2! 233.875 us  |                         }
 ------------------------------------------
 0)   sleep-533    =>  kworker-302         << ctx switch has taken place! >>
 ------------------------------------------

 0)  kworker-302   |  d..2              |      finish_task_switch() {
 0)  kworker-302   |  d..2              |        _raw_spin_unlock_irq() {

--snip--

 0)  ftrace_-529   |  ....+ 32.958 us   |                   }
 0)  ftrace_-529   |  ....! 148.834 us  |                 }
 0)  ftrace_-529   |   ==========>      |              << hardware interrupt ! >>
 0)  ftrace_-529   |  d...              |                 gic_handle_irq() {
```

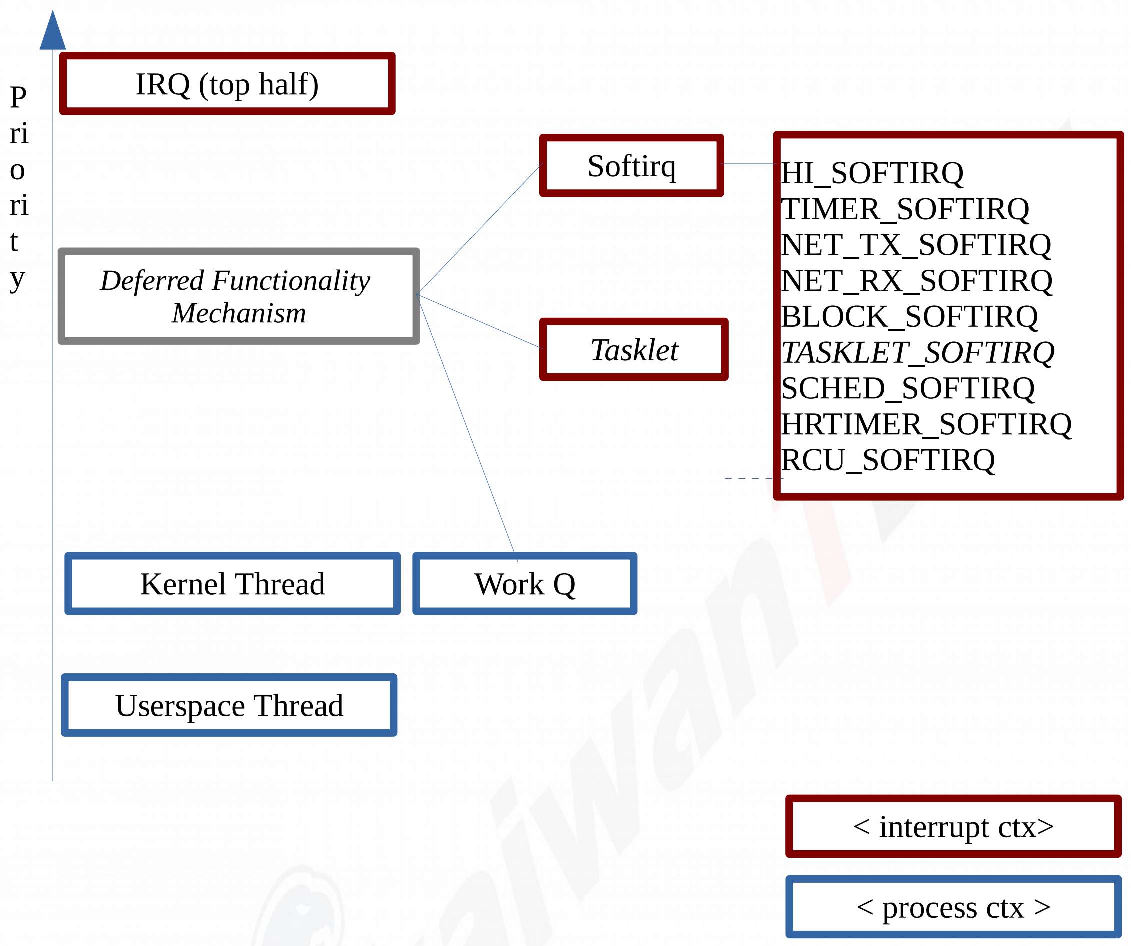
```
 0)  ftrace_-529   |  d...              |                  irq_find_mapping()
{
 0)  ftrace_-529   |  d...  1.416 us    |
irq_domain_legacy_revmap();
 0)  ftrace_-529   |  d...+ 31.708 us   |                  }
 0)  ftrace_-529   |  d...              |                  handle_IRQ() {
 0)  ftrace_-529   |  d...              |                    irq_enter() {
 0)  ftrace_-529   |  d...  2.250 us    |
rcu_irq_enter();
 0)  ftrace_-529   |  d...  1.458 us    |
add_preempt_count();
 0)  ftrace_-529   |  d.h.+ 61.500 us   |                 } << h => hardirq >>
 0)  ftrace_-529   |  d.h.              |
generic_handle_irq() {
 0)  ftrace_-529   |  d.h.  2.000 us    |                      irq_to_desc();

                       _-----=> irqs-off
                      / _----=> need-resched
                     | / _---=> hardirq/softirq
                     || / _--=> preempt-depth
                     ||| /
CPU  TASK/PID        ||||  DURATION                  FUNCTION CALLS
 |     |    |        ||||   |   |                     |   |   |   |

 0)  ftrace_-529   |  d.h.              |
handle_fasteoi_irq() {
 0)  ftrace_-529   |  d.h.              |
_raw_spin_lock() {
 0)  ftrace_-529   |  d.h.  1.042 us    |
add_preempt_count();
 0)  ftrace_-529   |  d.h1+ 33.667 us   |                        }
 0)  ftrace_-529   |  d.h1              |
handle_irq_event() {
 0)  ftrace_-529   |  d.h1              |
_raw_spin_unlock() {
 0)  ftrace_-529   |  d.h1  1.042 us    | sub_preempt_count();

--snip--

 0)  ftrace_-529   |  d.h.! 1974.458 us |                      }
 0)  ftrace_-529   |  d.h.! 2034.417 us |                    }
 0)  ftrace_-529   |  d.h.              |                    irq_exit() {
 0)  ftrace_-529   |  d.h.  1.417 us    |
sub_preempt_count();
 0)  ftrace_-529   |  d...              |                      do_softirq() {
 0)  ftrace_-529   |  d...              |
__do_softirq() {
 0)  ftrace_-529   |  d...  1.000 us    |
msecs_to_jiffies();
 0)  ftrace_-529   |  d...  3.000 us    |
__local_bh_disable();
 0)  ftrace_-529   |  ..s.              |                 << s => softirq >>
run_timer_softirq() {
 0)  ftrace_-529   |  ..s.  1.042 us    |
hrtimer_run_pending();
 0)  ftrace_-529   |  ..s.              |
_raw_spin_lock_irq() {
 0)  ftrace_-529   |  d.s.  1.000 us    | add_preempt_count();
```

```
--snip--

                       _-----=> irqs-off
                      / _----=> need-resched
                     | / _---=> hardirq/softirq
                     || / _--=> preempt-depth
                     ||| /
CPU  TASK/PID        ||||  DURATION                  FUNCTION CALLS
 |     |    |        ||||   |   |                     |   |   |   |

 0)  ftrace_-529   |  d.s.  1.750 us    |
sub_preempt_count();
 0)  ftrace_-529   |  d...+ 33.167 us   |                           }
 0)  ftrace_-529   |  d...! 2401.167 us |                         }
 0)  ftrace_-529   |  d...! 2432.083 us |                       }
 0)  ftrace_-529   |  d...  2.416 us    |                       rcu_irq_exit();
 0)  ftrace_-529   |  d...! 2522.625 us |                     }
 0)  ftrace_-529   |  d...! 4705.041 us |                   }
 0)  ftrace_-529   |  d...! 4798.041 us |                 }
 0)  ftrace_-529   |   <=========== |
 0)  ftrace_-529   |  ....! 5182.375 us |               }
 0)  ftrace_-529   |  ....! 5375.875 us |             }
 0)  ftrace_-529   |  ....              |             terminate_walk() {
 0)  ftrace_-529   |  ....              |               path_put() {
 0)  ftrace_-529   |  ....              |                 dput() {

--snip--
```

FYI, Ftrace Resources

- Documentation/trace/ftrace.txt
- [LWN, Steven Rostedt]
  - Debugging the kernel using Ftrace - part 1
  - Debugging the kernel using Ftrace – part 2
  - Secrets of the Ftrace function tracer

Task Prioritization on Linux



<< Source: Essential Linux Device Drivers, S Venkateshwaran >>

Deferred Functionality Mechanisms in the 2.6 Linux Kernel – A Quick Comparison

--snip--
You have seen the differences between interrupt handlers and bottom halves, but there are a few similarities, too. Interrupt handlers and tasklets are both not reentrant. And neither of them can go to sleep. Also, interrupt handlers, tasklets, and softirqs cannot be preempted.

Work queues are a third way to defer work from interrupt handlers. They execute in process context and are allowed to sleep, so they can use drowsy functions such as mutexes. We discussed work queues in the preceding chapter when we looked at various kernel helper facilities. Table 4.1 compares softirqs, tasklets, and work queues.

Table 4.1. Comparing Softirqs, Tasklets, and Work Queues

	Softirqs	***Tasklets***	***Work Queues***
Execution context	Deferred work runs in interrupt context.	Deferred work runs in interrupt context.	Deferred work runs in process context.
Reentrancy	Can run simultaneously on different CPUs.	Cannot run simultaneously on different CPUs. Different CPUs can run different tasklets, however.	Can run simultaneously on different CPUs.
Sleep semantics	Cannot go to sleep.	Cannot go to sleep.	May go to sleep.
Preemption	Cannot be preempted/ scheduled.	Cannot be preempted/ scheduled.	May be preempted/scheduled.
Ease of use	Not easy to use.	Easy to use.	Easy to use.
When to use	If deferred work will not go to sleep and if you have crucial scalability or speed requirements.	If deferred work will not go to sleep.	If deferred work may go to sleep.

There is an ongoing debate in LKML on the feasibility of getting rid of the tasklet interface. Tasklets enjoy more priority than process context code, so they present latency problems. Moreover, as you learned, they are constrained not to sleep and to execute on the same CPU. It's being suggested that all existing tasklets be converted to softirqs or work queues on a case-by-case basis.

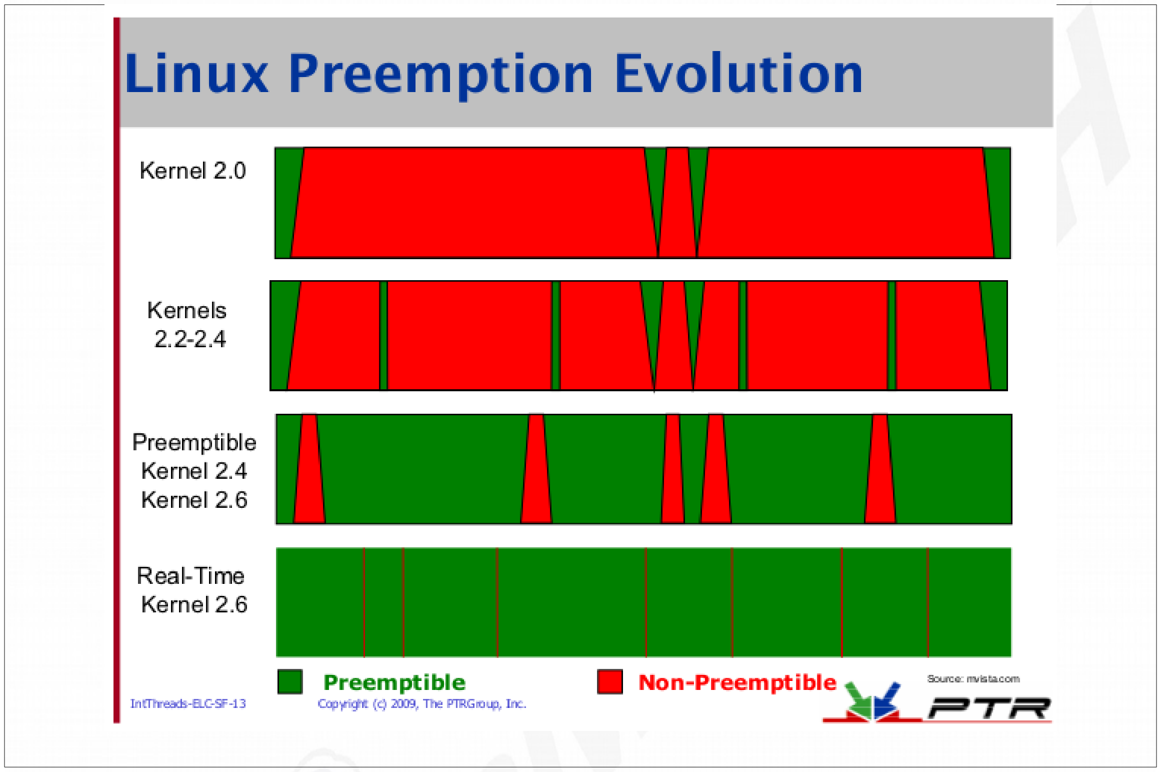
[For more detail on Bottom-Half management, see the LKD3 book.]

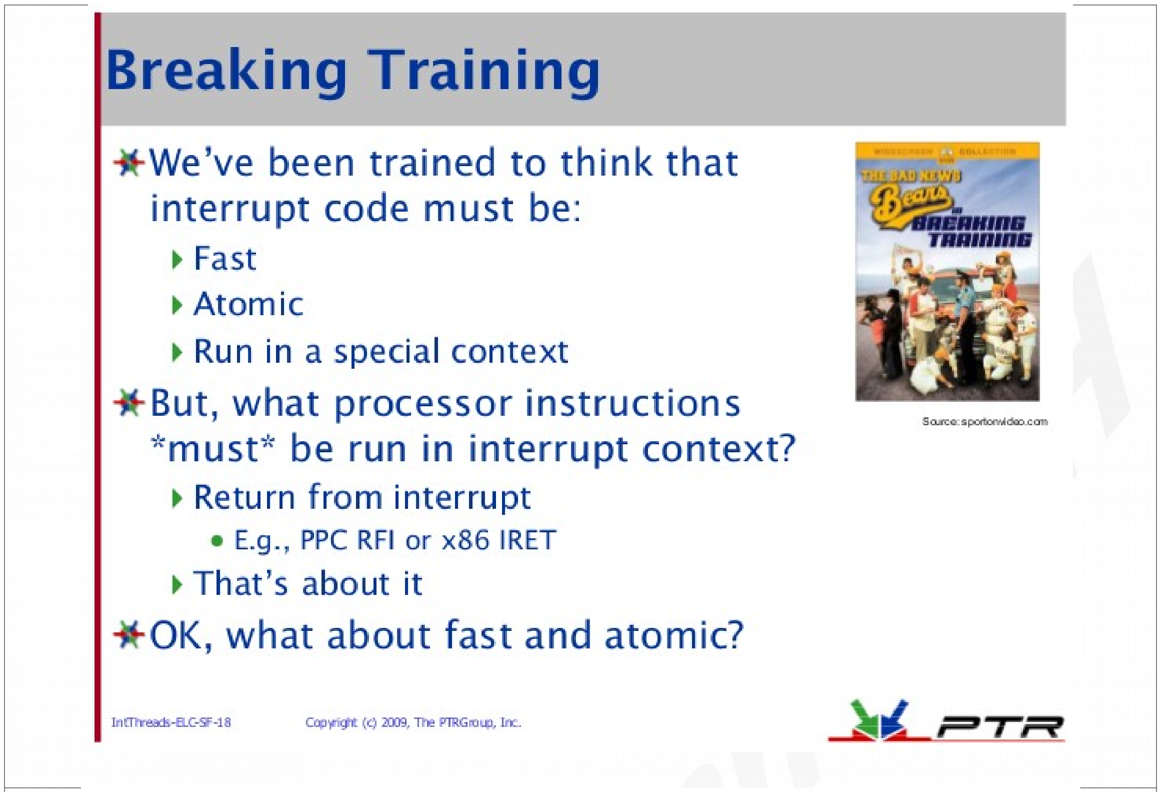
Examples

- *The Linux kernel source tree drivers/ branch !*
- *Creating a Basic LED Driver for Raspberry Pi [GPIO driver]*
- *http://derekmolloy.ie/kernel-gpio-programming-buttons-and-leds/*

Appendix A : Threaded Interrupts

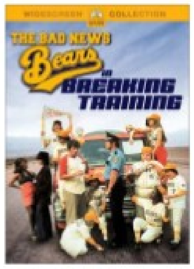
Source – excellent (though a bit old) discussion slides on interrupt theory and handling on Linux. (Only a few relevant slides shown below).





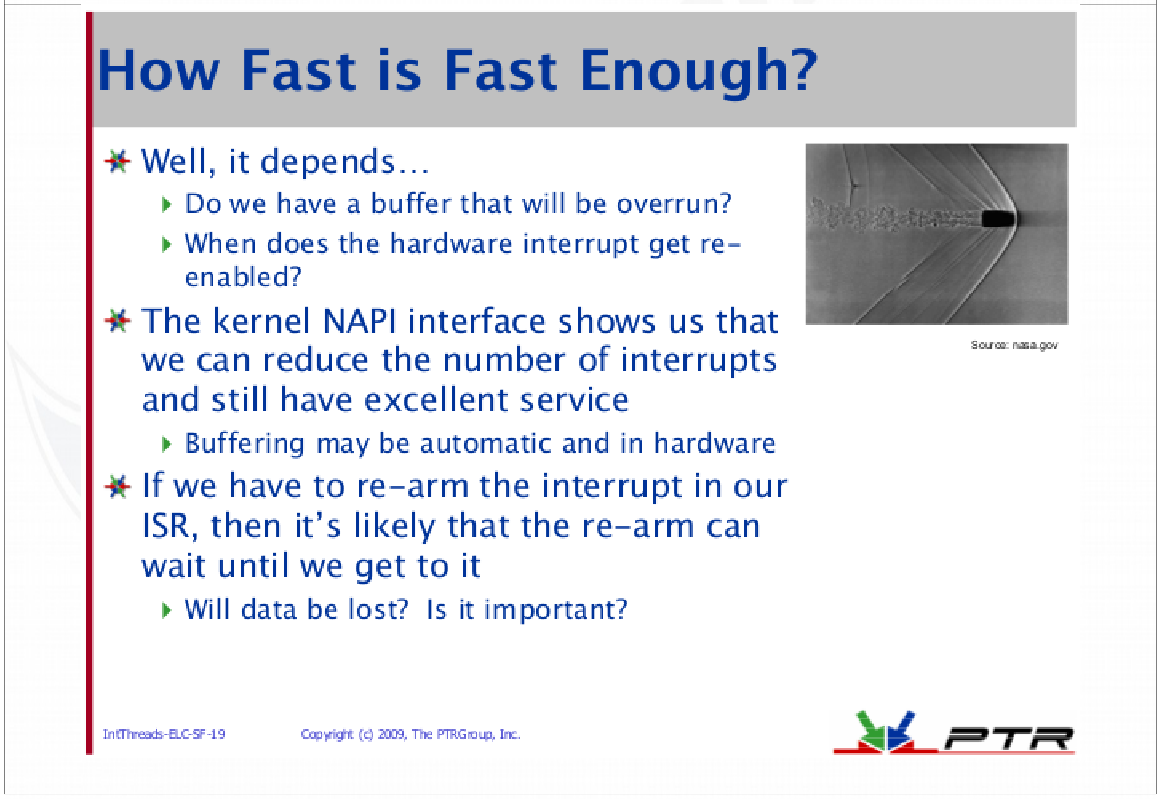
Breaking Training

- We've been trained to think that interrupt code must be:
  - Fast
  - Atomic
  - Run in a special context
- But, what processor instructions *must* be run in interrupt context?
  - Return from interrupt
    - E.g., PPC RFI or x86 IRET
  - That's about it
- OK, what about fast and atomic?



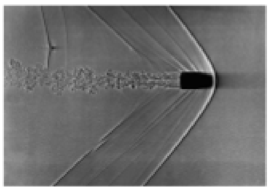
Source: sportonvideo.com

IntThreads-ELC-SF-18 Copyright (c) 2009, The PTRGroup, Inc.



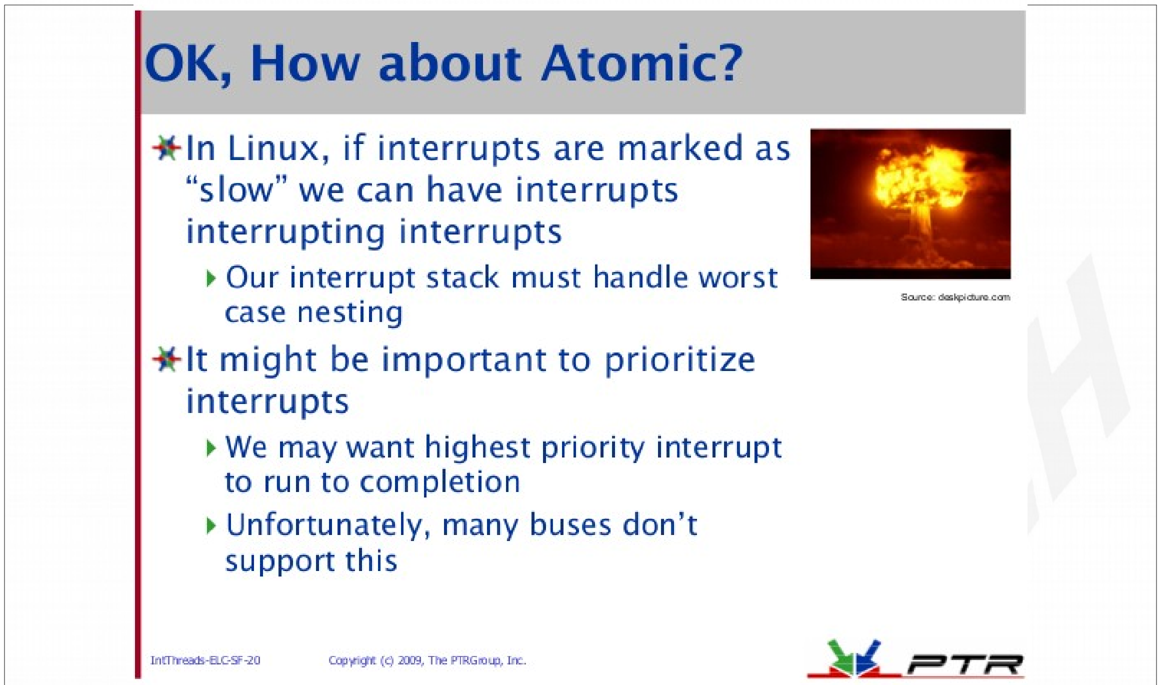
How Fast is Fast Enough?

- Well, it depends…
  - Do we have a buffer that will be overrun?
  - When does the hardware interrupt get re-enabled?
- The kernel NAPI interface shows us that we can reduce the number of interrupts and still have excellent service
  - Buffering may be automatic and in hardware
- If we have to re-arm the interrupt in our ISR, then it's likely that the re-arm can wait until we get to it
  - Will data be lost? Is it important?



Source: nasa.gov

IntThreads-ELC-SF-19 Copyright (c) 2009, The PTRGroup, Inc.

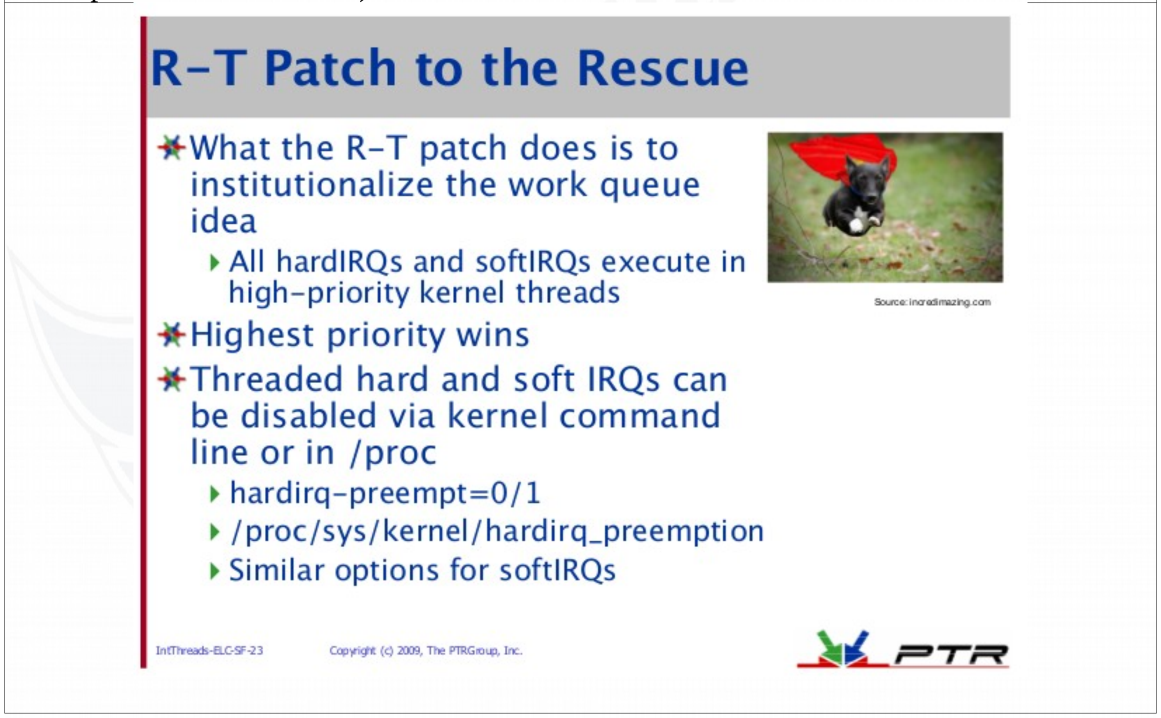


OK, How about Atomic?

- In Linux, if interrupts are marked as "slow" we can have interrupts interrupting interrupts
  - Our interrupt stack must handle worst case nesting
- It might be important to prioritize interrupts
  - We may want highest priority interrupt to run to completion
  - Unfortunately, many buses don't support this

IntThreads-ELC-SF-20 Copyright (c) 2009, The PTRGroup, Inc.

NOTE: Careful! "Slow" interrupts are outdated; recent kernel's think of all interrupts as "fast": this really implies that when a particular IRQn service routine is running, *all* IRQs are masked out (blocked) on the local processor (another way of saying this, is that interrupts are reentrant-safe).

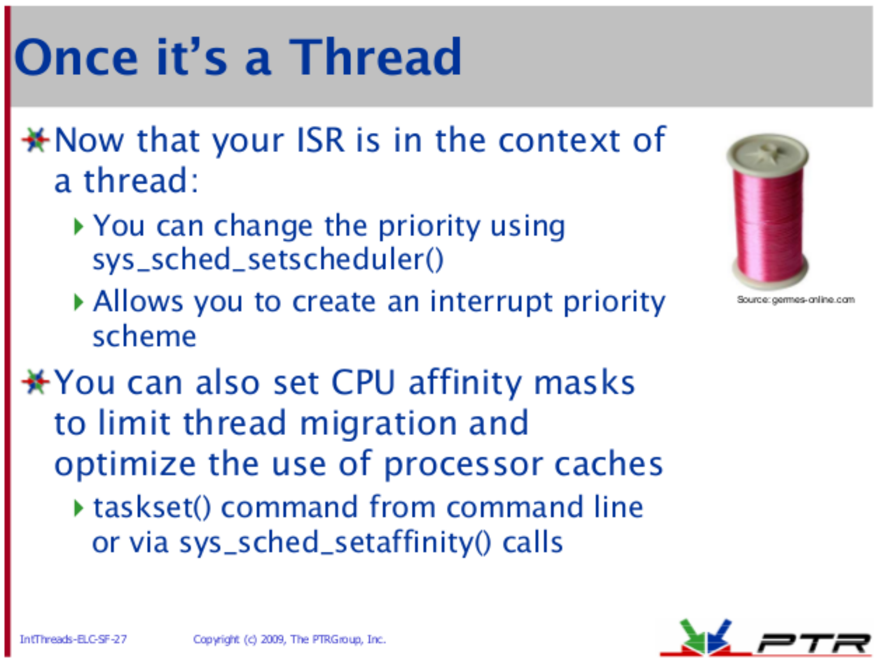


R-T Patch to the Rescue

- What the R-T patch does is to institutionalize the work queue idea
  - All hardIRQs and softIRQs execute in high-priority kernel threads
- Highest priority wins
- Threaded hard and soft IRQs can be disabled via kernel command line or in /proc
  - hardirq-preempt=0/1
  - /proc/sys/kernel/hardirq_preemption
  - Similar options for softIRQs

IntThreads-ELC-SF-23 Copyright (c) 2009, The PTRGroup, Inc.

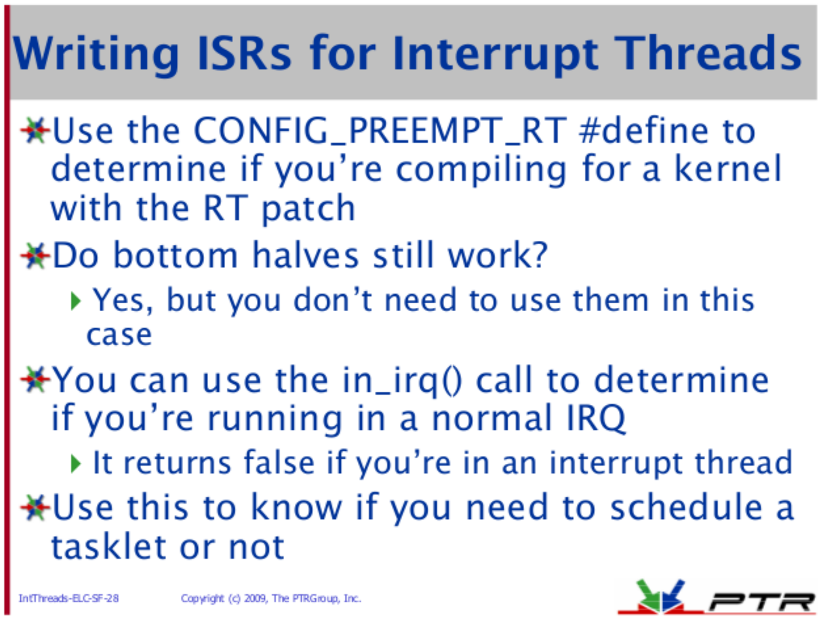
NOTE: Careful! Outdated: the */proc/sys/kernel/hardirq_preemption* entry does not exist on recent kernels.



Once it's a Thread

- Now that your ISR is in the context of a thread:
  - You can change the priority using sys_sched_setscheduler()
  - Allows you to create an interrupt priority scheme
- You can also set CPU affinity masks to limit thread migration and optimize the use of processor caches
  - taskset() command from command line or via sys_sched_setaffinity() calls

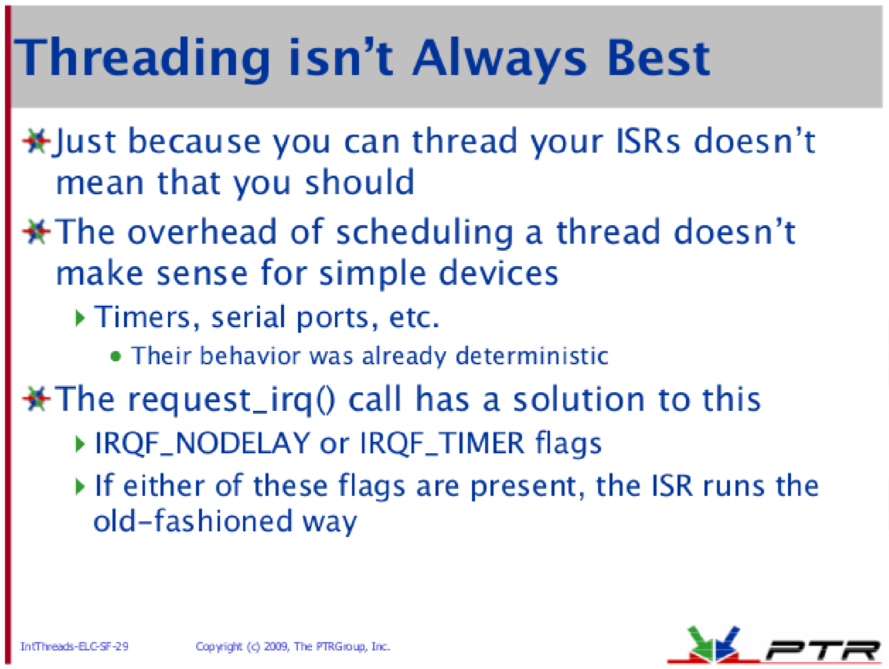
IntThreads-ELC-SF-27 Copyright (c) 2009, The PTRGroup, Inc.



Writing ISRs for Interrupt Threads

- Use the CONFIG_PREEMPT_RT #define to determine if you're compiling for a kernel with the RT patch
- Do bottom halves still work?
  - Yes, but you don't need to use them in this case
- You can use the in_irq() call to determine if you're running in a normal IRQ
  - It returns false if you're in an interrupt thread
- Use this to know if you need to schedule a tasklet or not

IntThreads-ELC-SF-28 Copyright (c) 2009, The PTRGroup, Inc.

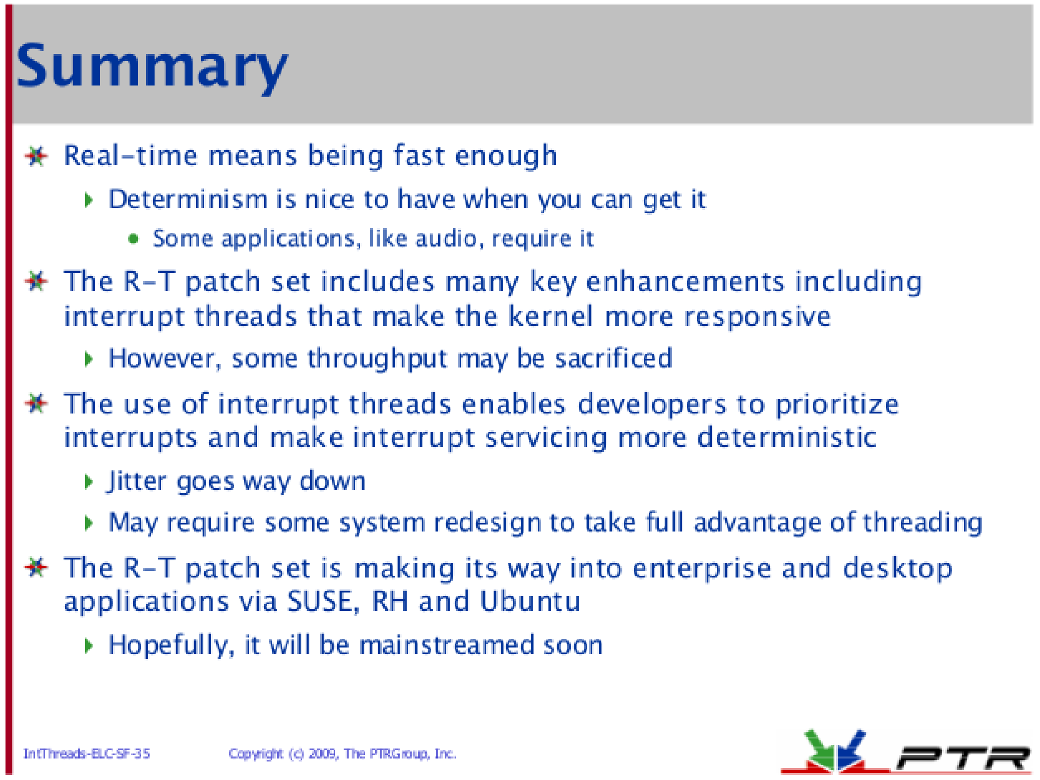


Threading isn't Always Best

- Just because you can thread your ISRs doesn't mean that you should
- The overhead of scheduling a thread doesn't make sense for simple devices
  - Timers, serial ports, etc.
    - Their behavior was already deterministic
- The request_irq() call has a solution to this
  - IRQF_NODELAY or IRQF_TIMER flags
  - If either of these flags are present, the ISR runs the old-fashioned way

IntThreads-ELC-SF-29 Copyright (c) 2009, The PTRGroup, Inc.

Note: Careful! The flag is no longer IRQF_NODELAY ; use IRQF_NO_THREAD instead.



Summary

- Real-time means being fast enough
  - Determinism is nice to have when you can get it
    - Some applications, like audio, require it
- The R-T patch set includes many key enhancements including interrupt threads that make the kernel more responsive
  - However, some throughput may be sacrificed
- The use of interrupt threads enables developers to prioritize interrupts and make interrupt servicing more deterministic
  - Jitter goes way down
  - May require some system redesign to take full advantage of threading
- The R-T patch set is making its way into enterprise and desktop applications via SUSE, RH and Ubuntu
  - Hopefully, it will be mainstreamed soon

IntThreads-ELC-SF-35 Copyright (c) 2009, The PTRGroup, Inc.

Appendix B : Low level Interrupt Handling on x86

<< Sourced from ULK3; hardware-specific to IA-32 architecture >>

[FYI / OPTIONAL : x86 only]

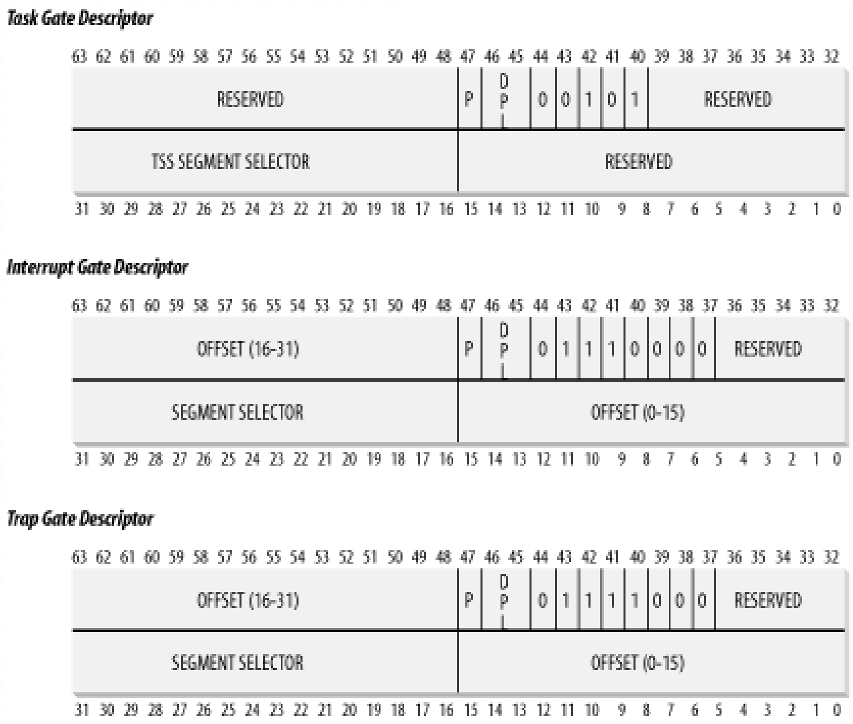
Interrupt Descriptor Table (IDT)

A system table called Interrupt Descriptor Table (IDT) associates each interrupt or exception vector with the address of the corresponding interrupt or exception handler. The IDT must be properly initialized before the kernel enables interrupts.

The IDT format is similar to that of the GDT and the LDTs examined in Chapter 2. Each entry corresponds to an interrupt or an exception vector and consists of a 8-byte descriptor. Thus, a maximum of 256 x 8 = 2048 bytes are required to store the IDT.

The idtr CPU register allows the IDT to be located anywhere in memory:
it specifies both the IDT base physical address and its limit (maximum length). It must be initialized before enabling interrupts by using the lidt assembly language instruction.

The IDT may include three types of descriptors; Figure 4-2 illustrates the meaning of the 64 bits included in each of them. In particular, the value of the Type field encoded in the bits 40-43 identifies the descriptor type.



The descriptors are:

Task gate

Includes the TSS selector of the process that must replace the current one when an interrupt signal occurs. Linux does not use task gates.

Interrupt gate

Includes the Segment Selector and the offset inside the segment of an interrupt handler. While transferring control to the proper segment, the processor clears the IF flag, thus disabling further maskable interrupts.

Trap gate

Similar to an interrupt gate, except that while transferring control to the proper segment, the processor does not modify the IF flag.
<< The int 0x80 syscall is a trap gate entry: type 15, DPL 3. >>

Linux uses interrupt gates to handle interrupts and trap gates to handle exceptions.