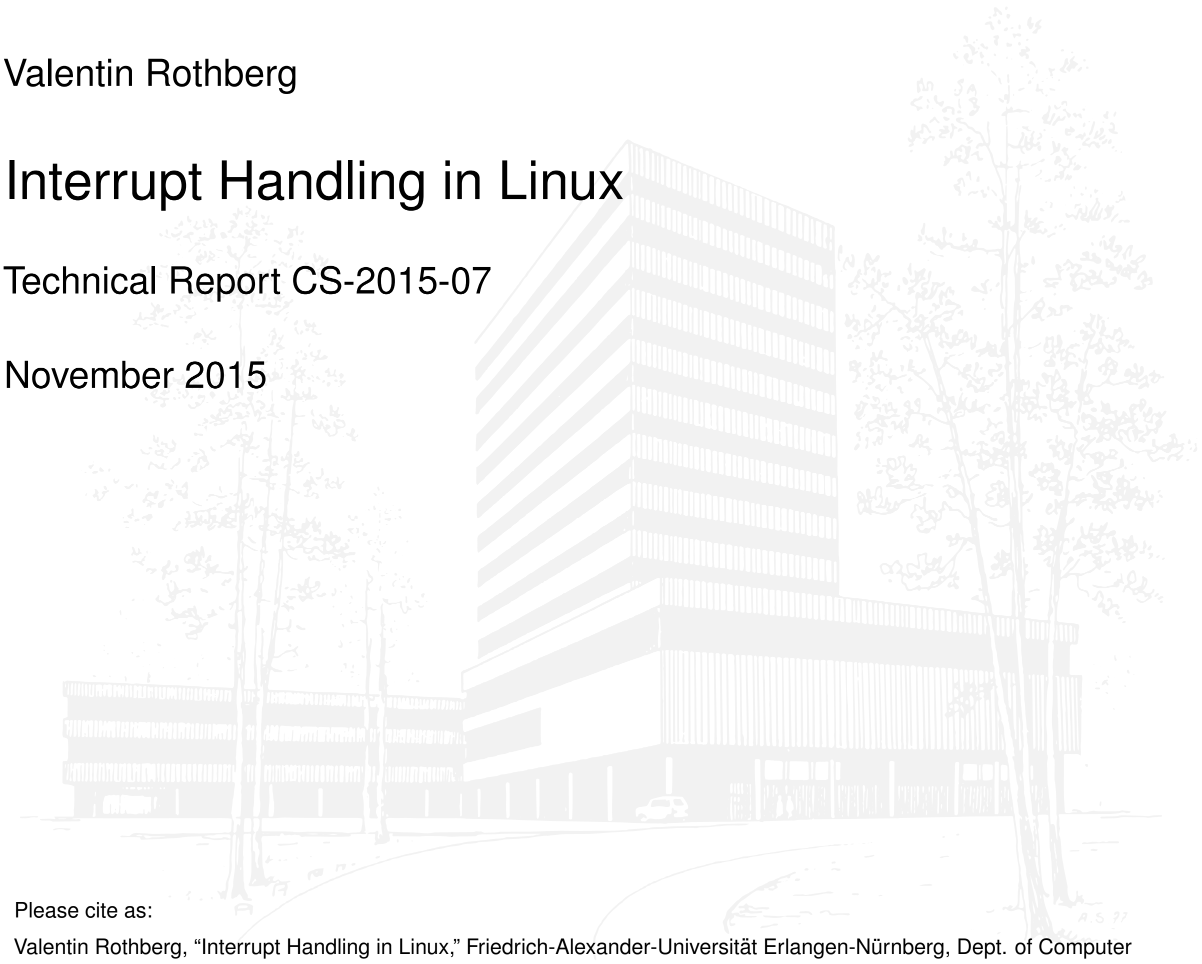
Department Informatik

Technical Reports / ISSN 2191-5008



Valentin Rothberg

Interrupt Handling in Linux

Technical Report CS-2015-07

November 2015

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.cs.fau.de

Interrupt Handling in Linux

Valentin Rothberg

Distributed Systems and Operating Systems
Dept. of Computer Science, University of Erlangen, Germany

rothberg@cs.fau.de

November 8, 2015

An interrupt is an event that alters the sequence of instructions executed by a processor and requires immediate attention. When the processor receives an interrupt signal, it may temporarily switch control to an interrupt service routine (ISR) and the suspended process (i.e., the previously running program) will be resumed as soon as the interrupt is being served. The generic term *interrupt* is oftentimes used synonymously for two terms, *interrupts* and *exceptions* [2]. An exception is a *synchronous* event that occurs when the processor detects an error condition while executing an instruction. Such an error condition may be a devision by zero, a page fault, a protection violation, etc. An interrupt, on the other hand, is an *asynchronous* event that occurs at random times during execution of a program in response to a signal from hardware. A proper and timely handling of interrupts is critical to the performance, but also to the security of a computer system.

In general, interrupts can be emitted by hardware as well as by software. *Software interrupts* (e.g., via the `INT n` instruction of the x86 instruction set architecture (ISA) [5]) are means to change the execution context of a program to a more privileged interrupt context in order to enter the kernel and, in contrast to hardware interrupts, occur synchronously to the currently running program. Consequently, such instructions are gates to the privileged operating-system kernel (e.g., ring 0 on x86) and thus are used to request services from the kernel (i.e., system calls). A *hardware-triggered interrupt* indicates that some device requires attention of the processor, and hence implements a means of communication between devices and the operating system. Interrupts from hardware can greatly improve a system's performance when devices send interrupts (e.g., a keystroke on a keyboard) instead of expensive polling of devices (e.g., periodically polling a keyboard for stroked keys). Furthermore, hardware-emitted interrupts by timers are used for timing and time measurement in general, but also for time sharing as ticks can be used to schedule another task.

After arrival of an interrupt, the processor executes the interrupt service routine that is associated with this interrupt. The ISR is also referred to as the interrupt request (IRQ) handler. But as interrupts are disabled during execution of the IRQ handler, not all kinds of code can or should be executed in this IRQ context. For instance, if such routine goes to sleep with interrupts disabled, the system is likely to freeze. On the other hand, active waiting and blocking should be avoided since other interrupts with potentially urgent needs remain disabled and, hence, cannot be served. Furthermore, acquiring locks is likely to cause deadlocks. As a consequence, the Linux kernel offers various mechanisms and application programming interfaces (APIs) to implement interrupt handling in order to meet certain functional and non-functional requirements.

This report focuses on how the Linux operating-system kernel handles interrupts on the *software* side and aims to give brief background information as well as implementation details. Detailed information about *exceptions* and exception handling in the x86 architecture can be found in the CPU manual "Intel 64 and IA-32 Architectures Software Developer's Manual" [5]. Notice that the report does not aim for completeness, neither does it target to introduce the general concept of interrupts. It rather tries to provide information for developers, researchers and students, familiar with operating systems and operating-system concepts, how Linux handles interrupts on the software side.

1 Basic Interrupt Handling

After arrival of an interrupt, the processor executes an interrupt request handler that is associated with this interrupt and hence with the issuing device or software interrupt.[1] The execution of such IRQ handlers in Linux is constrained by certain conditions. The most important condition is that the execution of an IRQ handler cannot be interrupted. Hence, interrupts on the executing CPU are disabled until the handler returns.

Earlier versions of the Linux kernel knew two types of interrupt handlers. A *fast* handler that runs with interrupts disabled, and a *slow* handler that runs with interrupts enabled. Slow handlers turned out to be dangerous since they could be interrupted by other interrupts. Such nested interruptions lead to undesirable issues, most importantly to overflows of the interrupt stack. Nowadays, interrupt handlers run with interrupts disabled which puts certain limitations on the code to be executed:

Execution Time

Interrupt handlers need to be as fast as possible. The more time it takes to finish execution of an IRQ handler, the longer local interrupts will be disabled. Hence, long running handlers can slow down the system and may also lead to losing interrupts. The faster the handler returns, the lower the interrupt latencies in the kernel, which is especially important for real-time systems.

Execution Context

Interrupt handlers are executed in hard-interrupt context – CPU-local interrupts remain disabled. As a consequence, locking is undesirable and sleeping must be avoided. Handlers that rely on such mechanisms need to defer parts of their code into another, safer context (see Section 2 and Section 3).

Both limitations lead to the fact that most interrupt handlers execute only a small amount of code and defer the rest of the work to a later point in time. This assures a fast re-enabling of CPU-local interrupts and thereby reduces latency, and avoids losing interrupts. Furthermore, code that may block or sleep (e.g., when allocating memory) can be executed in a safer context without slowing down or even stopping the system.

The prototype of a Linux IRQ handler expects two arguments: the interrupt line and a unique device identifier of the peripheral (i.e., the pointer to the device structure of the associated hardware device).

```
typedef irqreturn_t (*irq_handler_t)(int, void *);
```

[1]The process of how the processor or the OS finds the right handler is architecture and implementation specific and will not be covered in this document.

1.1 Registering Interrupt Handlers

Linux provides several functions to register an IRQ handler (see `include/linux/interrupt.h`):

request_irq() expects as arguments an interrupt line (i.e., the interrupt number of the issuing device), a function pointer to the interrupt handler, interrupt flags (see Section 1.2), an ASCII string and a unique device identifier. When an interrupt occurs, the interrupt line and the pointer to the device will be passed to the registered handler which can use this data for further processing.

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

request_threaded_irq() requires a second `irq_handler_t` handler that will be executed in a dedicated kernel thread after the IRQ handler returns. Moving code execution into a kernel thread is mandatory for handlers that may sleep. Section 2 explains in greater detail why and how interrupt handling can be deferred. Note that if the IRQ handler is set to `NULL` it will default to the default primary handler which does nothing more than returning `IRQ_WAKE_THREAD` (see Section 1.3).

```
extern int __must_check
request_threaded_irq(unsigned int irq, irq_handler_t handler,
                     irq_handler_t thread_fn, unsigned long flags,
                     const char *name, void *dev);
```

request_any_context_irq() has the same interface as `request_irq()` but different semantics. Depending on the underlying hardware platform, the function will further call `request_irq()` or `request_threaded_irq()` to execute the requested handler as a threaded interrupt handler. This function is helpful for drivers that register interrupt handlers that may run as a thread, depending on the underlying hardware. By using this function the kernel can decide during execution time how and where the IRQ handler will run. See git commit ae731f8d0785 ("genirq: Introduce request_any_context_irq()") for more information.

request_percpu_irq() has the same interface as request_irq() but will set the handler as a per-CPU interrupt handler. Per-CPU interrupts are used for instance to associate local timers to each core. While these timers are separate devices and have a separate interrupt line to a core, they all use the same IRQ number. To distinguish between the different cores, this function uses CPU-local device identifiers. Hence, the passed device identifier (`void *dev`) must be globally unique; the handler is then invoked with the interrupted CPU's instance of that variable.

1.2 Interrupt Flags

The following IRQ flags can be passed to request an IRQ handler and are defined in (see `include/linux/interrupts.h`):

`IRQF_DISABLED`
: In ancient versions of the Linux kernel, there were two types of main interrupt handlers. Fast handlers, which ran with interrupts disabled (i.e., with `IRQF_DISABLED` set), and slow handlers that remained interruptible. According to commit e58aa3d2d0cc ("genirq: Run irq handlers with interrupts disabled") running IRQ handlers with interrupts enabled can cause stack overflows which is an undesirable situation, especially in an OS kernel. Since then, interrupts remain disabled during execution of an IRQ handler. See Linux Weekly News (LWN) [4] to get more background information.

  Note that this flag has been removed by commit d8bf368d0631 ("genirq: Remove the deprecated '`IRQF_DISABLED`' request_irq() flag entirely") since Linux v4.1 (August 2015).

`IRQF_SHARED`
: Allow an interrupt line to be shared among several devices. Handlers that share an interrupt line need to check first if the passed device identifier belongs to them (i.e., they check if the arriving interrupt was issued by their device) and must return `IRQ_NONE` (see Section 1.3) in case not.

`IRQF_PROBE_SHARED`
: Set by callers to avoid sharing mismatches (i.e., when the line is already occupied by another handler that does not share the line). In case of a mismatch the kernel will print an error message and dump the stack. This flag is barely used since most devices that support sharing can use multiple interrupt lines and just pick another line if a request fails.

`IRQF_PERCPU`
: Interrupt is per CPU.

`IRQF_NOBALANCING`
: Flag to exclude this interrupt from IRQ balancing. The purpose of IRQ balancing is to distribute hardware interrupts across processors on a multiprocessor system in order to increase performance. The load balancing takes the current workload as well as cache locality and other functional and non-functional properties into account. Setting this flag forbids to set any CPU affinity for the requested interrupt handler. Such functionality is needed to provide a flexible setup for clocksources.

`IRQF_IRQPOLL`
: A polled interrupt indicates that one of the attached devices may need attention

but the kernel does not know on which interrupt line. Hence the system needs to poll all attached devices to find out which one has triggered the interrupt. For performance reasons, the kernel only checks the first registered device on shared interrupt lines. Polling interrupts are implemented as timers and are used, for instance, in old network-card drivers to periodically poll for incoming network packets in order to mitigate interrupts under high load.

`IRQF_ONESHOT`
: The interrupt is not reenabled after the IRQ handler finishes. This flag is required for threaded interrupts which need to keep the interrupt line disabled until the threaded handler has run.

  Specifying this flag is mandatory if the primary handler is set to `NULL`. The default primary handler does nothing more than to return `IRQ_WAKE_THREAD` to wake up a kernel thread to execute the thread_fn IRQ handler. If the `IRQF_ONESHOT` flag is not specified, some handlers will end up in stack overflows since the interrupt of the issuing device is still active. Hence the kernel rejects such requests and throws an error message. The Coccinelle[2] script `scripts/coccinelle/misc/irqf_oneshot.cocci` detects such cases automatically.

`IRQF_NO_SUSPEND`
: Do not disable this IRQ during suspension. There are interrupts that can legitimately trigger during the entire system suspend-resume cycle, including the "noirq" phases of suspending and resuming devices, as well as during the time when no-boot CPUs are taken offline and brought back online. That applies to timer interrupts in the first place, but also to inter process interrupts (IPI) and to some other special-purpose interrupts.

  Note that the `IRQF_NO_SUSPEND` flag affects the entire IRQ line and not just one user of it. Thus, if the line is shared, all of the interrupt handlers installed for it will be executed as usual after suspend_device_irqs(). For this reason, using `IRQF_NO_SUSPEND` and `IRQF_SHARED` at the same time should be avoided.

`IRQF_FORCE_RESUME`
: Enable the IRQ on resume even if `IRQF_NO_SUSPEND` is set. Xen needs to reenable interrupts that are marked `IRQF_NO_SUSPEND` in the resume path.

`IRQF_NO_THREAD`
: The interrupt cannot be threaded. Some low level interrupts (e.g., timers) cannot be threaded even when we force to thread all interrupt handlers.

`IRQF_TIMER`
: Flag to mark this interrupt as a timer interrupt. This flag is an alias for "`IRQF_NO_SUSPEND | IRQF_NO_THREAD`".

[2] `http://coccinelle.lip6.fr/`

`IRQF_EARLY_RESUME`
: Resume IRQ early at resume time of system core operations instead of at device resume time. See the commit message of the introducing commit 9bab0b7fbace ("genirq: Add `IRQF_RESUME_EARLY` and resume such IRQs earlier"): "Under Xen we need to resume IRQs associated with IPIs early enough that the resched IPI is unmasked and we can therefore schedule ourselves out of the stop_machine where the suspend/resume takes place."

In some cases it might be useful to know in advance if a specified interrupt line can be requested with specific flags. The kernel provides the function `can_request_irq()` to accomplish this task:

```
int can_request_irq(unsigned int irq, unsigned long flags);
```

The function first searches the line's IRQ descriptor, and then checks if the specified flags conflict with the flags of any already registered IRQ handler of this interrupt line. If no other handler is registered, the line can be requested immediately. Since the peripheral component interconnect (PCI) standard requires devices to share interrupts, can_request_irq() is used in this context to check if the specified interrupt line can be shared. However, most drivers just loop over an array of supported interrupt lines and use the first successfully requested line.

1.3 Return Type of Interrupt Handlers

The return type `irqreturn_t` of interrupt handlers constraints the behavior after return of the handler:

`IRQ_NONE`
: This flag is mostly used for shared IRQ handlers. When the interrupt did not originate from the driver's device, `IRQ_NONE` must be returned to allow the kernel to call the next registered interrupt handler.

`IRQ_HANDLED`
: The interrupt has been handled successfully.

`IRQ_WAKE_THREAD`
: The IRQ handler thereby requests to wake the handler thread. It implies `IRQ_HANDLED` and must only be returned by handlers registered with `request_threaded_interrupt()` (see Section 2.4) in order to wake up the kernel thread that will execute the second handler.

1.4 Unregister Interrupt Handlers

An interrupt handler can be unregistered from an interrupt line by calling `free_irq()`. This function will also disable the interrupt line if it is no longer used by handlers. The function waits for running IRQ handlers to finish, and should consequently never be called from any interrupt context to avoid potential deadlocks.

```
void free_irq(unsigned int irq, void *dev_id);
```

1.5 Managed Device Resources

The basic idea of *managed device resources* is to ease driver development by keeping track of all allocated resources, which need to be freed automatically when the driver detaches from the device. Hence, developers do not need to deal with the entire set of deallocation calls and the kernel is less prone to leak memory. In addition to the aforementioned API, the kernel ships similar functions for managed device resources, which expect a `struct device` as the first argument implemented in `kernel/irq/devres.c`. Note that the interrupt line still needs to be freed. You may read `Documentation/driver-model/devres.txt` for general information about the concept or read a detailed article on LWN from January 2007 [3].

2 Bottom Halves

When the processor receives an interrupt signal, it temporarily switches control to an interrupt-request handler and the suspended process (i.e., the previously running program) will be resumed as soon as the interrupt is being served. Such interrupt handlers in Linux run with disabled interrupts, which are re-enabled when the handler returns. Hence, not all kinds of code can or should be executed in this so-called hard interrupt context. Especially blocking operations, such as acquiring a lock or requesting memory, are likely to cause deadlocks or may even lead to losing interrupts since arriving interrupts cannot be served. Sleeping, on the other hand, is doomed to freeze the currently executing processor.

Linux, among other UNIX-like operating systems, addresses the limitations of code execution in interrupt handlers by moving (parts of) the code outside the hard-interrupt context by splitting up interrupt handlers into two halves: *top halves* (i.e., the common interrupt handlers), and *bottom halves* which run in a so-called soft-interrupt context or even in process context executed by kernel threads. Note that the meaning of top and bottom halves may differ in literature depending on the target operating system and the hardware architecture of interest. In the course of this report, a top half represents the hard interrupt handler, which can defer work to one or multiple bottom halves. In general, a top half should execute as little code as possible, and, if needed, to schedule the corresponding bottom half. In stark contrast to top halves, bottom halves generally remain interruptible and thus are more suitable for long running tasks. Note that the decision of which code should be executed in the top half and which code in the bottom half is not always trivial. If a developer choses to run his interrupt handler only as a top half, the interrupt will be handled as fast as possible since scheduling and executing bottom halves takes time and, hence, entails latency. However, depending on the execution time of the top half, the developer may risk to lose other interrupts, which is generally intolerable.

The following sections cover a broad set of bottom half mechanisms and APIs provided by the Linux kernel, each tailored to specific needs and use cases.

2.1 Softirqs

Softirqs are bottom halves that run in kernel context, and are suitable for long running, non-blocking handlers. Softirqs are executed after return of a hard-interrupt handler (i.e., top half) and before return to user space from a system call. Hence, a softirq is executed as early as possible but remains interruptible so that the execution can be preempted by any top half. All softirqs are defined in an enumerator, which is used internally to map a softirq handler (i.e., a function pointer) to a unique number (i.e., the associated enumerator entry). The unique softirq number is used at run time to raise/schedule a specific softirq. Since a softirq is executed on the processor it has been raised on, each processor has its own softirq bitmap. After return of a top half and before return to user space from a system call, the kernel iterates over the bitmap (least significant bit first) and executes the associated softirq handler of those entries that are set. All in all, there are four major properties of softirqs:

1. Softirqs are *declared at compile-time* in an enumerator. Hence, softirqs are not suitable for drivers that want to register their softirqs when they are loaded at run time.

2. Softirqs are *executed as early as possible* (i.e., after return of a top half and before return to user space from a system call), giving a *high priority* to the executed softirq handlers. In case of Linux, this is mainly networking and block IO. However, softirqs delay any user or kernel thread, which declassifies blocking to avoid monopolization of the CPU and the kernel.

3. Softirqs can *run in parallel.* Each processor has its own softirq bitmap. Hence, one softirq cannot be scheduled twice on the same processor but it may run in parallel on another. This property makes a lot of sense for networking and IO since data can be processed CPU-locally (good for cache optimizations) and the associated softirqs can run in parallel. However, the parallel execution of softirqs may require some locking to avoid race conditions which in turn raises the risk of deadlocks. Ideally, softirq handlers should be written in a non-blocking manner.

4. Softirqs have a *fixed execution hierarchy.* The kernel iterates over the softirq bitmap, least significant bit (LSB) first, and executes the associated softirq handlers. Due to the LSB first execution, softirqs are executed in ascending order as declared in the softirq enumerator.

The softirq enumerator is declared in `include/linux/interrupts.h` and looks as illustrated below. Remember that softirqs are executed in ascending order, such that `HI_SOFITRQ` is to be executed first, `RCU_SOFTIRQ` last.

```
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,
    NR_SOFTIRQS
};
```

2.1.1 Registering Softirq Handlers

The kernel provides the function `open_softirq()` to associate a softirq handler (2nd parameter) with a specified softirq (1st parameter). Note that the specified function (action) has the same signature as top-half handlers. It becomes clear that softirqs are simply a mean to defer work from hard-interrupt handlers to soft-interrupt context and consequently provide the same interface.

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

2.1.2 Scheduling Softirqs

Linux keeps a CPU-local bit mask to indicate if a softirq needs to be executed or not. `raise_softirq()` sets the corresponding bit of the specified softirq, which schedules the execution of the associated interrupt handler; either after return of a top half or before return to user space from a system call. `raise_softirq_irqoff()` must be used to schedule a softirq when interrupts are disabled in order to avoid potential dead locks.

```
void raise_softirq(unsigned int nr)
{
    unsigned long flags;
    local_irq_save(flags);
    raise_softirq_irqoff(nr);
    local_irq_restore(flags);
}
```

2.1.3 Executing Softirqs

The actual execution of softirqs is managed by `do_softirq()` (see `kernel/softirq.c`) which further calls __do_softirq() in case any bit in the local softirq bit mask is set. Depending on the context and the execution path of the caller, a new stack might be set up before to avoid exceeding stack boundaries. __do_softirq() then iterates over the softirq bit mask (least significant bit first) and invokes scheduled softirq handlers.

2.1.4 ksoftirqd - When Softirqs are just too much

Softirqs are executed as long as the processor-local softirq bitmap is set. Since softirqs are bottom halves and thus remain interruptible during execution, the system can find itself in a state where it does nothing else than serving interrupts and softirqs: incoming interrupts may schedule softirqs what leads to another iteration over the bitmap. Such processor-time monopolization by softirqs is acceptable under high workloads (e.g., high IO or network traffic), but it is generally undesirable for a longer period of time since (user) processes cannot be executed.

The Linux kernel addresses the problem of processor monopolization by softirqs with a simple, yet powerful mechanism. After the tenth iteration over the softirq bitmap, the kernel schedules the so-called *ksoftirqd* kernel thread, which takes control over the execution of softirqs. This processor-local kernel thread then executes softirqs as long as any bit in the softirq bitmap is set. The aforementioned processor-monopolization is thus avoided by deferring softirq execution into process context (i.e., kernel thread), so that the ksoftirqd can be preempted by any other (user) process. Another advantage to move execution of softirqs into process context is to let the scheduler decide when to dispatch which task; the scheduler has a global view of the current work load and can also take task priorities into consideration.

In older kernels, the ksoftirqd threads ran at the lowest possible priority. This means that softirqs ran either on high priority (i.e., kernel context) or on the lowest priority. Since Linux v2.6.23 (October 2007), ksoftirqds run at normal priority by default.

2.1.5 Softirqs and NAPI

The new API (NAPI) is a networking API of the Linux kernel, introduced with Linux v2.5 in the year 2001 and targets towards interrupt mitigation. NAPI mixes interrupts with polling and thereby provides higher performance under high traffic load by significantly reducing the load on the CPU [1].

Chrisitan Benvenuti describes the general idea behind NAPI in his book *Understanding Linux Network Internals* [1] as follows: In the old model, a device driver generates an interrupt for each network frame it receives. As a consequence, under high traffic loads, the time and effort spent handling interrupts can lead to a considerable waste of resources, higher latencies and a decreasing throughput. The main idea behind NAPI is simple: instead of using a pure interrupt-driven model, it uses a mix of interrupts and polling. If new frames are received when the kernel has not finished handling the previous ones yet, there is no need for the driver to generate other interrupts: it is just easier to have the kernel keep processing whatever is in the device input queue (with interrupts disabled for the device), and reenable interrupts once the queue is empty. This way, the driver reaps the advantages of both interrupts and polling:

- Asynchronous events, such as the reception of one or more frames, are indicated by interrupts so that the kernel does not have to check continuously if the device's ingress queue is empty.
- If the kernel knows that there is something left in the device's ingress queue, there is no need to wait for the next interrupt and waste (again) time handling following interrupt, but the kernel can switch to polling instead.

By looking at the softirq enumerator, the entries `NET_TX_SOFTIRQ` and `NET_RX_SOFTIRQ` in particular, it becomes clear that Linux developers implemented NAPI by means of softirqs:

```
enum
{   [...],
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ, [...]
};
```

A NAPI driver uses a softirq handler to dequeue frames from the ingress queue(s) of the supported network card. When the limit of maximum dequeued frames per dequeue-iteration is reached, the softirq handler sets the corresponding bit in the softirq bitmap (i.e., `NET_RX_SOFTIRQ`) to make sure that the queue will be dequeued timely – before return to user space. Hence, by making use of softirqs, a network driver cannot monopolize the kernel even under high load: as soon as the kernel dequeued the ingress-queue for the tenth time, the *ksoftirqd* kernel thread (Section 2.1.4) will be scheduled which takes control over softirq handling and hence over dequeuing. Thereby, the ksoftirqd kernel thread completely eliminates the need to implement polling by means of timers, since the scheduler ensures that the network card is periodically polled.

2.2 Tasklets

Tasklets are bottom halves that run in kernel context. They are suitable for long running and non-blocking tasks, and allow – in contrast to softirqs – a dynamic allocation of new handlers. Internally, tasklets are implemented by means of softirqs. In a simplified way: the tasklet softirq-handler is responsible for executing all items (i.e., tasklet handlers) of a linked list. A major difference to softirqs is that a tasklet cannot run simultaneously on multiple CPUs. Tasklets avoid concurrency by design and thereby take the responsibility to synchronize data access from the developer, which further eases driver development. Nonetheless, tasklets are bottom halves, so the principle of a split interrupt service remains the same: the top half performs a small amount of work in hard interrupt context and defers the rest of the work to an interruptible context – by scheduling a tasklet.

2.2.1 The Requirement of Dynamic Bottom-Half Allocation

Softirqs were originally designed as a vector of 32 entries. Each time a new softirq was added to the enumerator, developers were forced to change the hierarchy among softirqs. This approach increased the complexity of softirqs as well as their development, since developers were forced to come up with a new hierarchy for each new softirq. To avoid this manual approach, a new class of bottom halves has been introduced with Linux v2.3.43 (February 2000), tasklets. Tasklets can be dynamically allocated, making them suitable for driver developers who rely on dynamically allocatable resources since many drivers can be loaded into the kernel at runtime.

2.3 Execution as Softirqs

Tasklets are organized in linked lists. Each CPU has two local tasklet lists, one for normal and one for high priority tasklets. The execution of tasklets happens in the softirq action handlers `tasklet_action()` and `tasklet_hi_action()` (see `kernel/softirq.c`). Both handlers iterate over the corresponding list of tasklets and call the associated tasklet handler of each list item. In case a tasklet is already running on another core, it will be re-added to the list and the pending bit in the softirq bitmask will be set to indicate an additional iteration of softirq execution. Data accesses to the tasklet lists are synchronized by disabling local interrupts. The synchronization of one tasklet over multiple CPUs is accomplished by atomary `test_and_set_bit()` operations; if the operation succeeds, the tasklet is not already running on another CPU and, hence, the associated handler can be invoked. Otherwise, the tasklet will be rescheduled by adding it back to the list and setting the associated softirq bit to indicate another round of softirq execution.

Let us have a look again at the softirq enumerator, which contains two types of tasklets: `HI_SOFTIRQ` for high priority tasklets, and `TASKLET_SOFTIRQ` for normal priority tasklets. Remember that softirqs are executed in ascending order. It is interesting to see that high priority tasklets are executed before the timer, networking and block IO, which traditionally have the highest priority in Linux.

```
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,
    NR_SOFTIRQS
};
```

The internal data structure of tasklets, `tasklet_struct`, includes a next pointer, a state flag, an atomic counter to indicate if the tasklet is enabled or disabled , a function pointer to the bottom-half handler, and the argument for this function.

```
struct tasklet_struct
{
    struct tasklet_struct *next;// linked list
    unsigned long state;        // scheduled or running? (for waiting)
    atomic_t count;             // enabled or disabled?
    void (*func)(unsigned long);// function pointer, i.e. bottom half
    unsigned long data;         // argument for function
};
```

The kernel provides two macros to declare tasklets. `DECLARE_TASKLET()` sets the atomic counter to 0, which indicates that the tasklet is ready to be scheduled, whereas `DECLARE_TASKLET_DISABLED()` sets the counter to 1, so that the tasklet cannot be scheduled without explicitly setting the counter to 0.

There is also a function to initialize a specified tasklet that will be marked to be ready for scheduling:

```
void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long),
                  unsigned long data)
{
    t->next = NULL;
    t->state = 0;
    atomic_set(&t->count, 0);
    t->func = func;
    t->data = data;
}
```

To enable a tasklet, we can use `tasklet_enable()`. Tasklets can be disabled with two functions, whereas `tasklet_disable()` waits until running or already scheduled instances of the tasklets have returned; `tasklet_disable_nosync()` is a brute-force disabling of the specified tasklet.

```
static inline void tasklet_enable(struct tasklet_struct *t);
static inline void tasklet_disable(struct tasklet_struct *t);
static inline void tasklet_disable_nosync(struct tasklet_struct *t);
```

2.3.1 Scheduling Tasklets

The following enumerator is used to represent the two states of a tasklet, namely if it is already scheduled for execution or if it is running at the moment. This differentiation is especially important for SMP systems, since one tasklet cannot run simultaneously on multiple cores.

```
enum
{
    TASKLET_STATE_SCHED,/* Tasklet is scheduled for execution */
    TASKLET_STATE_RUN   /* Tasklet is running (SMP only) */
};
```

Since there are two classes of tasklets and hence two different CPU-local linked lists, the kernel provides two functions to schedule (i.e., enqueue) a tasklet at normal and at high priority. If a specified tasklet has been scheduled before (i.e., if the `TASKLET_STATE_SCHED` is set), it will not be reschuled.

```
static inline void tasklet_schedule(struct tasklet_struct *t)
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
        __tasklet_schedule(t);
}
static inline void tasklet_hi_schedule(struct tasklet_struct *t)
{
    if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
        __tasklet_hi_schedule(t);
}
```

Once scheduled, a tasklet can be removed from the ready list by calling `tasklet_kill(struct tasklet_struct *t)`. The function waits for running instances to return and unsets the `TASKLET_STATE_SCHED` bit in the flags of the specified tasklet.

2.4 Threaded Interrupts

Threaded interrupts are bottom halves that run each in their own, separate kernel thread. The concept was introduced by Thomas Gleixner[3] and originates from the real-time kernel tree. As a consequence, threaded interrupts meet many requirements of a real-time system, most importantly reducing interrupt latency in the kernel and allowing a fine-grained priority model.

The general rule behind threaded interrupts is simple: defer as much work to the kernel thread as possible – preferably all work. Only minimal work should be done in hard-interrupt context, for instance, verifying the interrupt on shared interrupt lines. Besides reducing the interrupt latency in the kernel, using threaded interrupts reduces complexity since it avoids many sources of deadlocks in synchronizing top and bottom halves – all code should be executed in the bottom half. Buggy IRQ handlers are also less likely to break the system since they are executed in process context by means of kernel threads.[4] Since each requested threaded interrupt receives its own kernel thread, using them brings along two essential benefits:

1. A threaded interrupt handler cannot ultimately block the execution of another as it can happen by using the workqueue API, where one kernel thread may serve as an execution provider for multiple work items (see Section 3).
2. Real-time systems are all about priorities. Since each handler has its own thread, we can assign individual priorities to the associated threads/interrupts. By default, the threads run at normal real-time priority (i.e., `MAX_USER_RT_PRIO/2`).

2.4.1 Requesting Threaded Interrupts

Requesting a threaded interrupt handler is straight forward and requires the specification of a second interrupt handler that will be executed in process context.

```
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);
```

Note that the first (hard) IRQ handler can be set to `NULL`, so that the default primary handler will be executed. This default handler does nothing more than to return `IRQ_WAKE_THREAD` to wake up the associated kernel thread which will execute the `thread_fn` handler. Hence, it is possible to move the execution of interrupt handlers entirely to process context, which is widely used in the domain of real-time systems. In such case the `IRQF_ONESHOT` flag must be passed, otherwise the request will fail. If you do not specify the `IRQF_ONESHOT` flag in such a situation, the interrupt would be reenabled after return of the top half, resulting in stack overflows for level interrupts since the issuing device has still the interrupt line asserted. Hence, the kernel rejects such requests and throws an error message.

[3] `https://lkml.org/lkml/2009/3/23/346`
[4] `https://lwn.net/Articles/302043/`

3 Workqueues

The workqueue API is the most commonly used interface to defer work into an asynchronous process execution context. The basic concept is to queue so-called *work items* in a workqueue. All work items are dequeued and executed in process context: the so-called *worker*, a kernel thread, picks up the items from the workqueue and executes the functions associated with those items. If there is no work, the worker will idle, but wake up as soon as new items arrive in the workqueue. Since work items are executed in process context, using workqueues is a suitable choice for work that needs to sleep. Workqueues are also suitable for long running and lengthy tasks, since the system scheduler can guarantee forward progress for other threads – also user processes – so that workers can be preempted at any time. Thereby the entire system can benefit as less time is spent in the kernel, and other processes can progress what can greatly improve the overall user experience.

Workqueues can be used inside and outside the context of interrupt handling, and hence receive a separate section in this report. Note that the following text is mainly based on the workqueue documentation shipped with the sources of the Linux kernel (see `Documentation/workqueue.txt`).

3.1 The Workqueue Approach

A **work item** is a simple data structure that holds a pointer to the function that is to be executed asynchronously. Whenever a user (e.g., a driver or subsystem) wants a function to be executed asynchronously by means of workqueues, it has to set up a work item pointing to that function and queue that work item on a workqueue. Special purpose threads, so-called **worker threads**, execute the functions after dequeuing the items, one after the other. If no work is queued, the worker threads become idle.

Worker threads are controlled by so-called **worker pools** which take care of the level of concurrency – the simultaneously running worker threads – and the process management. The design of concurrency managed workqueues differentiates between workqueues (e.g., used by drivers to defer work from top halves to a process context) and the backend mechanism which manages worker pools and processes the queued work items. In general, there are at least two worker pools per CPU: one for normal priority and one for high priority work items. Besides those two pools, there might be some extra worker pools to serve work items queued on unbound workqueues – the number of these backing pools is dynamic. When a work item is queued on a workqueue, the target worker pool is determined according to the queue parameters and its attributes, and the item will be appended on the worklist of this worker pool.

The workqueue management tries to keep concurrency at a minimal but sufficient level. *Minimal* to save resources and *sufficient* such that the system is used at its full capacity. Each worker pool is bound to a CPU and implements concurrency management by hooking the scheduler. The worker pool is notified whenever an active worker wakes up or sleeps and keeps track of the number of currently runnable workers. In general, work items are not expected to hog a CPU and consume many cycles. That means maintaining just enough concurrency to prevent work processing from stalling should be optimal. A worker pool does nothing as long as there is one or more of its workers runnable on the CPU. As soon as the last running worker goes to sleep, the pool schedules a new worker thread immediately to process the next queued item. This allows using a minimal number of workers without losing potential execution bandwidth. For **unbound workqueues** the number of backing pools is dynamic. The attributes of an unbound workqueue can be set individually by using `apply_workqueue_attrs()` and the workqueue will automatically create a worker pool matching these attributes.

Workqueues guarantee forward progress to all qeueued items. This guarantee relies on the fact that there is always a sufficient amount of execution contexts (i.e., worker threads) with the help of so-called **rescue workers**, which are executed under memory pressure to avoid potential dead locks. Consequently, all work items which might be used on code paths that handle memory reclaim are required to be queued on workqueues that have a rescuer thread. Note that users of workqueues do not need to worry about synchronization; the workqueue implementation is synchronized by means of mutexes and spinlocks, depending on the execution context (i.e., worker pool, workqueue, or worker).

3.2 Original Workqueues

The original workqueue implementation included two kinds of workqueues: a multi-threaded (MT) workqueue had one worker thread per CPU, a single-threaded (ST) workqueue had one worker thread system-wide. However, a growing number of CPUs and workqueues lead to situations where some systems saturated the default 32k PID space just booting up. Besides this resource hunger of workqueues, the provided level of concurrency was unsatisfactory as well. The limitation was common to both ST and MT workqueues albeit less severe on MT. Each workqueue maintained its own worker pool, which lacks a global view of concurrency in the system and thereby lead to various problems such as to deadlocks. The tension between the provided level of concurrency and resource usage also forced its users to make unnecessary trade-offs, such as libata choosing to use ST workqueues for polling PIOs and accepting an unnecessary limitation that no two polling PIOs can progress at the same time. As MT workqueues do not provide much better concurrency, users which require a higher or more dynamic level of concurrency, for instance *fscache*[5], a general purpose cache for network filesystems, had to implement its own thread pool, making a global view of concurrency barely possible.

[5]`https://www.kernel.org/doc/Documentation/filesystems/caching/fscache.txt`

3.3 Concurrency-Managed Workqueues

The concurrency-managed workqueue is a reimplementation of workqueues trying to eliminate the aforementioned drawbacks of the original implementation, mainly the unsatisfacory level of concurrency. The main focus of concurrency managed workqueues is as follows:

- Remain *compatible* with the original workqueue API.
- Use per-CPU *worker pools* shared by all workqueues to provide a flexible level of concurrency (i.e., the number of simultaneously running workers) on demand without wasting resources.
- *Automatically regulate* worker pool and level of *concurrency* so that the API users do not need to take care about such details.

3.4 Source and API

3.4.1 Workqueue Flags

There are various flags used in the context of workqueues that should be discussed before describing the API of workqueues:

`WQ_UNBOUND`

Work items queued to an unbound workqueues are served by the special worker pools that host workers that are not bound to any specific CPU. This makes an unbound workqueue behave as a simple execution-context provider without concurrency management. The unbound worker-pools try to start execution of work items as soon as possible. Unbound workqueues sacrifice locality what is useful for the following cases:

1. Wide fluctuation in the concurrency level requirement is expected and using bound workqueues may end up creating a large number of mostly unused workers across different CPUs.
2. Long running, CPU-intensive workloads can be better managed by the system scheduler (e.g., migration to other CPUs).

`WQ_FREEZABLE`

A freezable workqueue participates in the freeze phase of the system suspend operations. Work items on the workqueues are drained and no new work item starts execution until thawed.

This flag is used in the context of power management and file systems, and is especially important for creating the system image in the suspend phase, since non-freezable items could lead to file system corruption.

You can find more information about this topic in `Documentation/power/freezing-of-tasks.txt`.

`WQ_MEM_RECLAIM`

All workqueues which might be used in the memory reclaim paths must have this flag set. The workqueue is guaranteed to have at least one woker, a so-called rescuer thread, regardless of memory pressure. Such rescuers are needed in situations where resources, especially memory, run short. `GFP_KERNEL` allocations then may block and deadlock the entire workqueue. Rescuer threads pick up queued items and execute the associated functions, which may help to solve the queue's deadlock.

Let us consider the following scenario:

Workqueue W has 3 items A, B and C. A does some work and then waits until C has finished some work. Afterwards, B does some `GFP_KERNEL` allocations and blocks as there is not enough memory available. As a result, C cannot run since B still occupies the W's worker; another worker cannot be created because there is not enough memory.

A pre-allocated rescuer thread can solve this problem, by executing C which then wakes up A. B will continue as soon as there is enough available memory to allocate.

`WQ_HIGHPRI`

Work items of a high priority workqueues are queued to the high priority worker pool of the target CPU. High priority worker pools are served by worker threads with elevated nice level.

Note that normal priority and high priority worker pools do not interact with each other. They maintain separate pools and implement concurrency management among its workers.

`WQ_CPU_INTENSIVE`

Work items of a CPU-intensive workqueue do not contribute to the concurrency level. In other words, runnable CPU intensive work items will not prevent other work items in the same worker pool from starting execution. This is useful for bound work items which are expected to hog CPU cycles so that their execution is regulated by the system scheduler.

Although CPU intensive work items do not contribute to the concurrency level, start of their executions is still regulated by the concurrency management so that runnable non-CPU-intensive work items can delay the execution of CPU intensive work items.

This flag is meaningless for unbound workqueues.

`max_active`

`max_acvtive` determines the maximum number of execution contexts (workers) per CPU of the respective workqueue. If `max_active` is set to n, only n work items of the queue can run on the same CPU at the same time.

3.4.2 Scheduling Work Items

Work items can be queued to a specific workqueue with `queue_work()`. The function returns false if the work is already queued somewhere else, and returns true otherwise.

```
static inline bool queue_work(struct workqueue_struct *wq,
                              struct work_struct *work);
```

If you want to queue work on the standard system-wide workqueue `system_wq` (see Section 3.5, then you can use `schedule_work()`. `schedule_work_on()` additionally allows to specify a CPU on which the work item will be scheduled and executed on.

In some situatios we may want to make sure that all queued items on a specific workqueue have run to completion. The kernel provides the function `flush_workqueue()` for this task. This function sleeps until all previously queued non-idle work items finish execution, but it is not livelocked by new incoming ones.

```
void flush_workqueue(struct workqueue_struct *wq);
```

`flush_scheduled_work()` is a wrapper to flush work queued in `system_wq`. Besides the warning that using this function can lead to a deadlock, the source-code comment gives the following advice: *"In most situations flushing the entire workqueue is overkill; you merely need to know that a particular work item isn't queued and isn't running. In such cases you should use cancel_delayed_work_sync() or cancel_work_sync() instead."*.

There is also a number of functions to queue/schedule work after a specified number of jiffies. Those functions include the substring `_delayed_` and have an additional parameter to specify the delay in jiffies. For more information about the workqueue API you may read the workqueue header (i.e., `include/linux/workqueue.h`) and function definition in `kernel/workqueue.c`.

3.4.3 Allocating Workqueues

In case you decide to implement your own workqueue to create an execution environment that fits exactly your needs, you can chose between two macros. `alloc_workqueue()` allocates a workqueue with the specified flags and the concurrency level which is defined by `max_active`. The name of the workqueue can be specified with a printf format string (`fmt`) and the corresponding variable number of arguments. The function returns a pointer to the workqueue on success and `NULL` on failure.

```
#define alloc_workqueue(fmt, flags, max_active, args...) [...]
#define alloc_ordered_workqueue(fmt, flags, args...)     [...]
```

There is also the macro `alloc_ordered_workqueue()` to allocate ordered workqueues, which executes the queued items one by one in the given order (i.e., FIFO order). Such queues are unbound and have `max_active` set to 1. This function replaces create_freezeable_workqueue() and create_singlethread_workqueue() (see git commit 81dcaf6516d8).

3.5 System-Wide Workqeueues

There are 7 system-wide persistent workqueues in Linux v3.17 (see `include/linux/workqueue.h`):

1. `system_wq`
   System-wide multi-threaded workquque used by various variants of `schedule_work()`. Do not queue long-running work items, since users expect a relatively short flush tim.

2. `system_highpri_wq`
   Similar to system_wq but for work items which require `WQ_HIGHPRI`.

3. `system_long_wq`
   Similar to system_wq but may host long running works. Queue flushing is expected to take relatively long.

4. `system_unbound_wq`
   Unbound workqueue. Workers are not bound to any specific CPU, not concurrency managed, and all queued works are executed immediately as long as `max_active` limit is not reached and resources are available.

5. `system_freezable_wq`
   Equivalent to system_wq but with `WQ_FREEZABLE` enabled.

6. `power_efficient_wq`
   Inclined towards saving power and converted into `WQ_UNBOUND` variants if `WQ_POWER_EFFICIENT` is set.

7. `system_freezable_power_efficient_wq`
   Combination of system_freezable_wq and power_efficient_wq.

References

[1] Christian Benvenuti. *Understanding Linux Network Internals.* 2005.

[2] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel.* Oreilly & Associates Inc, 2005. ISBN: 0596005652.

[3] Jonathan Corbet. *Device resource management.* 2007. URL: `https://lwn.net/Articles/215996/` (visited on 11/03/2015).

[4] Jonathan Corbet. *Disabling IRQF_DISABLED.* 2010. URL: `https://lwn.net/Articles/380931/` (visited on 11/03/2015).

[5] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual.* 325462-055US. 2015.