Softirqs and Tasklets

We mentioned earlier in the section "Interrupt Handling" that several tasks among those executed by the kernel are not critical: they can be deferred for a long period of time, if necessary. Remember that the interrupt service routines of an interrupt handler are serialized, and often there should be no occurrence of an interrupt until the corresponding interrupt handler has terminated. Conversely, the deferrable tasks can execute with all interrupts enabled. Taking them out of the interrupt handler helps keep kernel response time small. This is a very important property for many time-critical applications that expect their interrupt requests to be serviced in a few milliseconds.

Linux 2.6 answers such a challenge by using two kinds of non-urgent interruptible kernel functions: the so-called *deferrable functions* [*] (*softirqs* and *tasklets*), and those executed by means of some work queues (we will describe them in the section "Work Queues" later in this chapter).

Softirqs and tasklets are strictly correlated, because tasklets are implemented on top of softirqs. As a matter of fact, the term "softirq," which appears in the kernel source code, often denotes both kinds of deferrable functions. Another widely used term is *interrupt context* : it specifies that the kernel is currently executing either an interrupt handler or a deferrable function.

Softirqs are statically allocated (i.e., defined at compile time), while tasklets can also be allocated and initialized at runtime (for instance, when loading a kernel module). Softirqs can run concurrently on several CPUs, even if they are of the same type. Thus, softirqs are reentrant functions and must explicitly protect their data structures with spin locks. Tasklets do not have to worry about this, because their

execution is controlled more strictly by the kernel. Tasklets of the same type are always serialized: in other words, the same type of tasklet cannot be executed by two CPUs at the same time. However, tasklets of different types can be executed concurrently on several CPUs. Serializing the tasklet simplifies the life of device driver developers, because the tasklet function needs not be reentrant.

Generally speaking, four kinds of operations can be performed on deferrable functions:

Initialization

Defines a new deferrable function; this operation is usually done when the kernel initializes itself or a module is loaded.

Activation

Marks a deferrable function as "pending" — to be run the next time the kernel schedules a round of executions of deferrable functions. Activation can be done at any time (even while handling interrupts).

Masking

Selectively disables a deferrable function so that it will not be executed by the kernel even if activated. We'll see in the section "Disabling and Enabling Deferrable Functions" in Chapter 5 that disabling deferrable functions is sometimes essential.

Execution

Executes a pending deferrable function together with all other pending deferrable functions of the same type; execution is performed at well-specified times, explained later in the section "Softirqs."

Activation and execution are bound together: a deferrable function that has been activated by a given CPU must be executed on the same CPU. There is no self-evident reason suggesting that this rule is beneficial for system performance. Binding the deferrable function to the activating CPU could in theory make better use of the CPU hardware cache. After all, it is conceivable that the activating kernel

thread accesses some data structures that will also be used by the deferrable function. However, the relevant lines could easily be no longer in the cache when the deferrable function is run because its execution can be delayed a long time. Moreover, binding a function to a CPU is always a potentially "dangerous" operation, because one CPU might end up very busy while the others are mostly idle.

Softirqs

Linux 2.6 uses a limited number of softirqs . For most purposes, tasklets are good enough and are much easier to write because they do not need to be reentrant.

As a matter of fact, only the six kinds of softirqs listed in Table 4-9 are currently defined.

Table 4-9. Softirqs used in Linux 2.6

Softirq	Index (priority)	Description
HI_SOFTIRQ	0	Handles high priority tasklets
TIMER_SOFTIRQ	1	Tasklets related to timer interrupts
NET_TX_SOFTIRQ	2	Transmits packets to network cards
NET_RX_SOFTIRQ	3	Receives packets from network cards
SCSI_SOFTIRQ	4	Post-interrupt processing of SCSI commands
TASKLET_SOFTIRQ	5	Handles regular tasklets

The index of a sofirq determines its priority: a lower index means higher priority because softirq functions will be executed starting from index 0.

Data structures used for softirqs

The main data structure used to represent softirqs is the `softirq_vec` array, which includes 32 elements of type `softirq_action`. The priority of a softirq is the index of the corresponding `softirq_action` element inside the array. As shown in Table 4-9, only the first six entries of the array are effectively used. The `softirq_action` data structure consists of two fields: an `action` pointer to the softirq function and a `data` pointer to a generic data structure that may be needed by the softirq function.

Another critical field used to keep track both of kernel preemption and of nesting of kernel control paths is the 32-bit `preempt_count` field stored in the `thread_info` field of each process descriptor (see the section "Identifying a Process" in Chapter 3). This field encodes three distinct counters plus a flag, as shown in Table 4-10.

Table 4-10. Subfields of the preempt_count field (continues)

Bits	Description
0–7	Preemption counter (max value = 255)
8–15	Softirq counter (max value = 255).
16–27	Hardirq counter (max value = 4096)
28	`PREEMPT_ACTIVE` flag

The first counter keeps track of how many times kernel preemption has been explicitly disabled on the local CPU; the value zero means that kernel preemption has not been explicitly disabled at all. The second counter specifies how many levels deep the disabling of deferrable functions is (level 0 means that deferrable functions are enabled). The third counter specifies the number of nested interrupt handlers on the local CPU (the value is increased by `irq_enter( )` and decreased by `irq_exit( )`; see the section "I/O Interrupt Handling" earlier in this chapter).

There is a good reason for the name of the `preempt_count` field: kernel preemptability has to be disabled either when it has been explicitly disabled by the kernel code (preemption counter not zero) or when the kernel is running in interrupt context. Thus, to determine whether the current process can be preempted, the kernel quickly checks for a zero value in the `preempt_count` field. Kernel preemption will be discussed in depth in the section "Kernel Preemption" in Chapter 5.

The `in_interrupt( )` macro checks the hardirq and softirq counters in the `current_thread_info( )->preempt_count` field. If either one of these two counters is positive, the macro yields a nonzero value, otherwise it yields the value zero. If the kernel does not make use of multiple Kernel Mode stacks, the macro always looks at the `preempt_count` field of the `thread_info` descriptor of the current process. If, however, the kernel makes use of multiple Kernel Mode stacks, the macro might look at the `preempt_count` field in the `thread_info` descriptor contained in a `irq_ctx` union associated with the local CPU. In this case, the macro returns a nonzero value because the field is always set to a positive value.

The last crucial data structure for implementing the softirqs is a per-CPU 32-bit mask describing the pending softirqs; it is stored in the `_ _softirq_pending` field of the `irq_cpustat_t` data structure (recall that there is one such structure per each CPU in the system; see Table 4-8). To get and set the value of the bit mask, the kernel makes use of the `local_softirq_pending( )` macro that selects the softirq bit mask of the local CPU.

Handling softirqs

The `open_softirq( )` function takes care of softirq initialization. It uses three parameters: the softirq index, a pointer to the softirq function to be executed, and a second pointer to a data structure that may be required by the softirq function. `open_softirq( )` limits itself to initializing the proper entry of the `softirq_vec` array.

Softirqs are activated by means of the `raise_softirq( )` function. This function, which receives as its parameter the softirq index `nr`, performs the following actions:

1. Executes the `local_irq_save` macro to save the state of the `IF` flag of the `eflags` register and to disable interrupts on the local CPU.

2. Marks the softirq as pending by setting the bit corresponding to the index `nr` in the softirq bit mask of the local CPU.

3. If `in_interrupt()` yields the value 1, it jumps to step 5. This situation indicates either that `raise_softirq( )` has been invoked in interrupt context, or that the softirqs are currently disabled.

4. Otherwise, invokes `wakeup_softirqd()` to wake up, if necessary, the

ksoftirqd kernel thread of the local CPU (see later).

5. Executes the `local_irq_restore` macro to restore the state of the `IF` flag saved in step 1.

Checks for active (pending) softirqs should be perfomed periodically, but without inducing too much overhead. They are performed in a few points of the kernel code. Here is a list of the most significant points (be warned that number and position of the softirq checkpoints change both with the kernel version and with the supported hardware architecture):

- When the kernel invokes the `local_bh_enable( )` function[*] to enable softirqs on the local CPU
- When the `do_IRQ( )` function finishes handling an I/O interrupt and invokes the `irq_exit( )` macro
- If the system uses an I/O APIC, when the `smp_apic_timer_interrupt( )` function finishes handling a local timer interrupt (see the section "Timekeeping Architecture in Multiprocessor Systems" in Chapter 6)
- In multiprocessor systems, when a CPU finishes handling a function triggered by a `CALL_FUNCTION_VECTOR` interprocessor interrupt
- When one of the special *ksoftirqd/n* kernel threads is awakened (see later)

The do_softirq( ) function

If pending softirqs are detected at one such checkpoint (`local_softirq_pending()` is not zero), the kernel invokes `do_softirq( )` to take care of them. This function performs the following actions:

1. If `in_interrupt( )` yields the value one, this function returns. This situation indicates either that `do_softirq( )` has been invoked in interrupt context or that the softirqs are currently disabled.
2. Executes `local_irq_save` to save the state of the `IF` flag and to disable the interrupts on the local CPU.

3. If the size of the `thread_union` structure is 4 KB, it switches to the soft IRQ stack, if necessary. This step is very similar to step 2 of `do_IRQ( )` in the earlier section "I/O Interrupt Handling;" of course, the `softirq_ctx` array is used instead of `hardirq_ctx`.

4. Invokes the `_ _do_softirq( )` function (see the following section).

5. If the soft IRQ stack has been effectively switched in step 3 above, it restores the original stack pointer into the `esp` register, thus switching back to the exception stack that was in use before.

6. Executes `local_irq_restore` to restore the state of the `IF` flag (local interrupts enabled or disabled) saved in step 2 and returns.

The _ _do_softirq( ) function

The `_ _do_softirq( )` function reads the softirq bit mask of the local CPU and executes the deferrable functions corresponding to every set bit. While executing a softirq function, new pending softirqs might pop up; in order to ensure a low latency time for the deferrable funtions, `_ _do_softirq( )` keeps running until all pending softirqs have been executed. This mechanism, however, could force `_ _do_softirq( )` to run for long periods of time, thus considerably delaying User Mode processes. For that reason, `_ _do_softirq( )` performs a fixed number of iterations and then returns. The remaining pending softirqs, if any, will be handled in due time by the *ksoftirqd* kernel thread described in the next section. Here is a short description of the actions performed by the function:

1. Initializes the iteration counter to 10.

2. Copies the softirq bit mask of the local CPU (selected by `local_softirq_pending( )`) in the `pending` local variable.

3. Invokes `local_bh_disable( )` to increase the softirq counter. It is somewhat counterintuitive that deferrable functions should be disabled before starting to execute them, but it really makes a lot of sense. Because the deferrable functions mostly run with interrupts enabled, an interrupt can be raised in the middle of the `_ _do_softirq( )` function. When `do_IRQ( )` executes the `irq_exit( )` macro, another instance of the `_ _do_softirq(`

`)` function could be started. This has to be avoided, because deferrable functions must execute serially on the CPU. Thus, the first instance of `_ _do_softirq( )` disables deferrable functions, so that every new instance of the function will exit at step 1 of `do_softirq( )`.

4. Clears the softirq bitmap of the local CPU, so that new softirqs can be activated (the value of the bit mask has already been saved in the `pending` local variable in step 2).

5. Executes `local_irq_enable( )` to enable local interrupts.

6. For each bit set in the `pending` local variable, it executes the corresponding softirq function; recall that the function address for the softirq with index `n` is stored in `softirq_vec[n]->action`.

7. Executes `local_irq_disable()` to disable local interrupts.

8. Copies the softirq bit mask of the local CPU into the `pending` local variable and decreases the iteration counter one more time.

9. If `pending` is not zero—at least one softirq has been activated since the start of the last iteration—and the iteration counter is still positive, it jumps back to step 4.

10. If there are more pending softirqs, it invokes `wakeup_softirqd( )` to wake up the kernel thread that takes care of the softirqs for the local CPU (see next section).

11. Subtracts 1 from the softirq counter, thus reenabling the deferrable functions.

The ksoftirqd kernel threads

In recent kernel versions, each CPU has its own *ksoftirqd/n* kernel thread (where *n* is the logical number of the CPU). Each *ksoftirqd/n* kernel thread runs the `ksoftirqd( )` function, which essentially executes the following loop:

```
for(;;) {
    set_current_state(TASK_INTERRUPTIBLE );
    schedule( );
```

```
        /* now in TASK_RUNNING state */
        while (local_softirq_pending( )) {
            preempt_disable();
            do_softirq( );
            preempt_enable();
            cond_resched( );
        }
    }
```

When awakened, the kernel thread checks the `local_softirq_pending()` softirq bit mask and invokes, if necessary, `do_softirq( )`. If there are no softirqs pending, the function puts the current process in the `TASK_INTERRUPTIBLE` state and invokes then the `cond_resched()` function to perform a process switch if required by the current process (flag `TIF_NEED_RESCHED` of the current `thread_info` set).

The *ksoftirqd/n* kernel threads represent a solution for a critical trade-off problem.

Softirq functions may reactivate themselves; in fact, both the networking softirqs and the tasklet softirqs do this. Moreover, external events, such as packet flooding on a network card, may activate softirqs at very high frequency.

The potential for a continuous high-volume flow of softirqs creates a problem that is solved by introducing kernel threads. Without them, developers are essentially faced with two alternative strategies.

The first strategy consists of ignoring new softirqs that occur while `do_softirq( )` is running. In other words, the `do_softirq( )` function could determine what softirqs are pending when the function is started and then execute their functions. Next, it would terminate without rechecking the pending softirqs. This solution is not good enough. Suppose that a softirq function is reactivated during the execution of `do_softirq( )`. In the worst case, the softirq is not executed again until the next timer interrupt, even if the machine is idle. As a result, softirq latency time is unacceptable for networking developers.

The second strategy consists of continuously rechecking for pending softirqs. The `do_softirq( )` function could keep checking the pending softirqs and would terminate only when none of them is pending. While this solution might satisfy networking developers, it can certainly upset normal users of the system: if a high-frequency flow of packets is received by a network card or a softirq function keeps

activating itself, the `do_softirq( )` function never returns, and the User Mode programs are virtually stopped.

The *ksoftirqd/n* kernel threads try to solve this difficult trade-off problem. The `do_softirq( )` function determines what softirqs are pending and executes their functions. After a few iterations, if the flow of softirqs does not stop, the function wakes up the kernel thread and terminates (step 10 of `_ _do_softirq( )`). The kernel thread has low priority, so user programs have a chance to run; but if the machine is idle, the pending softirqs are executed quickly.

Tasklets

Tasklets are the preferred way to implement deferrable functions in I/O drivers. As already explained, tasklets are built on top of two softirqs named `HI_SOFTIRQ` and `TASKLET_SOFTIRQ`. Several tasklets may be associated with the same softirq, each tasklet carrying its own function. There is no real difference between the two softirqs, except that `do_softirq( )` executes `HI_SOFTIRQ`'s tasklets before `TASKLET_SOFTIRQ`'s tasklets.

Tasklets and high-priority tasklets are stored in the `tasklet_vec` and `tasklet_hi_vec` arrays, respectively. Both of them include `NR_CPUS` elements of type `tasklet_head`, and each element consists of a pointer to a list of *tasklet descriptors*. The tasklet descriptor is a data structure of type `tasklet_struct`, whose fields are shown in Table 4-11.

Table 4-11. The fields of the tasklet descriptor

Field name	Description
`next`	Pointer to next descriptor in the list
`state`	Status of the tasklet
`count`	Lock counter
`func`	Pointer to the tasklet function
`data`	An unsigned long integer that may be used by the tasklet function

The `state` field of the tasklet descriptor includes two flags:

`TASKLET_STATE_SCHED`

When set, this indicates that the tasklet is pending (has been scheduled for execution); it also means that the tasklet descriptor is inserted in one of the lists of the `tasklet_vec` and `tasklet_hi_vec` arrays.

`TASKLET_STATE_RUN`

When set, this indicates that the tasklet is being executed; on a uniprocessor system this flag is not used because there is no need to check whether a specific tasklet is running.

Let's suppose you're writing a device driver and you want to use a tasklet: what has

to be done? First of all, you should allocate a new `tasklet_struct` data structure and initialize it by invoking `tasklet_init( )`; this function receives as its parameters the address of the tasklet descriptor, the address of your tasklet function, and its optional integer argument.

The tasklet may be selectively disabled by invoking either `tasklet_disable_nosync( )` or `tasklet_disable( )`. Both functions increase the `count` field of the tasklet descriptor, but the latter function does not return until an already running instance of the tasklet function has terminated. To reenable the tasklet, use `tasklet_enable( )`.

To activate the tasklet, you should invoke either the `tasklet_schedule( )` function or the `tasklet_hi_schedule( )` function, according to the priority that you require for the tasklet. The two functions are very similar; each of them performs the following actions:

1. Checks the `TASKLET_STATE_SCHED` flag; if it is set, returns (the tasklet has already been scheduled).

2. Invokes `local_irq_save` to save the state of the `IF` flag and to disable local interrupts.

3. Adds the tasklet descriptor at the beginning of the list pointed to by `tasklet_vec[n]` or `tasklet_hi_vec[n]`, where `n` denotes the logical number of the local CPU.

4. Invokes `raise_softirq_irqoff( )` to activate either the `TASKLET_SOFTIRQ` or the `HI_SOFTIRQ` softirq (this function is similar to `raise_softirq( )`, except that it assumes that local interrupts are already disabled).

5. Invokes `local_irq_restore` to restore the state of the `IF` flag.

Finally, let's see how the tasklet is executed. We know from the previous section

PREV 4.6. Interrupt Handling
NEXT 4.8. Work Queues

2. Gets the logical number n of the local CPU.

3. Stores the address of the list pointed to by `tasklet_vec[n]` or `tasklet_hi_vec[n]` in the `list` local variable.

4. Puts a `NULL` address in `tasklet_vec[n]` or `tasklet_hi_vec[n]`, thus emptying the list of scheduled tasklet descriptors.

5. Enables local interrupts.

6. For each tasklet descriptor in the list pointed to by `list`:

    1. In multiprocessor systems, checks the `TASKLET_STATE_RUN` flag of the tasklet.

        1. If it is set, a tasklet of the same type is already running on another CPU, so the function reinserts the task descriptor in the list pointed to by `tasklet_vec[n]` or `tasklet_hi_vec[n]` and activates the `TASKLET_SOFTIRQ` or `HI_SOFTIRQ` softirq again. In this way, execution of the tasklet is deferred until no other tasklets of the same type are running on other CPUs.

        2. Otherwise, the tasklet is not running on another CPU: sets the flag so that the tasklet function cannot be executed on other CPUs.

    2. Checks whether the tasklet is disabled by looking at the `count` field of the tasklet descriptor. If the tasklet is disabled, it clears its `TASKLET_STATE_RUN` flag and reinserts the task descriptor in the list pointed to by `tasklet_vec[n]` or `tasklet_hi_vec[n]`; then the function activates the `TASKLET_SOFTIRQ` or `HI_SOFTIRQ` softirq again.

    3. If the tasklet is enabled, it clears the `TASKLET_STATE_SCHED` flag and executes the tasklet function.

Notice that, unless the tasklet function reactivates itself, every tasklet activation triggers at most one execution of the tasklet function.

[*] These are also called *software interrupts*, but we denote them as "deferrable functions" to avoid confusion with programmed exceptions, which are referred to as "software interrupts " in Intel manuals.

PREV
4.6. Interrupt Handling

NEXT
4.8. Work Queues