Multitasking in the Linux Kernel. Interrupts and Tasklets

Written by **Vita Loginova** | 27 January 2015 | Original Source (http://habrahabr.ru/post/244071/)

In the previous article (http://kukuruku.co/hub/opensource/multitasking-management-in-the-operating-system-kernel) I mentioned about multithreading. The article covered such basic notions as types of multitasking, the scheduler, scheduling strategies, the state machine, and other.

This time, I want to look at the problem of scheduling from another perspective. Namely, I'm going to tell you about scheduling not threads, but their "younger brothers". Since the article turned out to be quite long, at the last moment I decided to break it up into several parts:

1. Multitasking in the Linux Kernel. Interrupts and Tasklets
2. Multitasking in the Linux Kernel. Workqueue
3. Protothread and Cooperative Multitasking

In the third part, I will also try to compare all of these seemingly different entities and extract some useful ideas. After a little while, I will tell you about the way we managed to apply these ideas in practice in the Embox (https://code.google.com/p/embox/) project, and about how we started our operating system on a small board with almost full multitasking.

I will try to tell you about all of this in detail, describing the basic API and sometimes going into details of the implementation peculiarities, particularly drawing attention to the problem of scheduling.

Interrupts and Their Processing

Hardware interrupt (Interrupt request — **IRQ**) is an external asynchronous event that comes from the hardware. It suspends the program flow and passes control to the processor to handle the event.

The processing of a hardware interrupt occurs the following way:

1. The current thread of control is suspended; the context information to return to the thread is saved.
2. A handler function (**ISR** – interrupt service routine) is performed in the context of the disabled hardware interrupts. The handler should perform the actions that are necessary for the current interrupt.
3. The equipment is informed that the interrupt is handled. So now it can generate new interrupts.
4. The context is restored for the return from the interrupt.

The handler function can be quite long, which is improper considering the fact that it is performed within the context of disabled hardware interrupts. Therefore, it was decided to divide the handling of interrupts into two parts (they are called top-half and bottom-half in Linux):

- The ISR itself, which is invoked during the interruption, performs only the most minimal work that cannot be postponed for later. It collects information about the interrupt that is necessary for further handling. Also, it interacts with the hardware, for example, blocks or clears the IRQ of the device and schedules the second part.
- The second part, in which the main handling is performed, is run in the different context of the processor, in which hardware interrupts are enabled. The call of this part of the handler will be performed later.

Thus, we have come to the deferred processing of interrupts. In Linux, there are **tasklet** and **workqueue** for this purpose.

Tasklet

In short, a **tasklet** is something like a very small thread that has neither stack, not context of its own. Such "threads" work quickly and completely. The main features of tasklets are the following:

- tasklets are atomic, so we cannot use **sleep()** and such synchronization primitives as mutexes (https://en.wikipedia.org/wiki/Mutual_exclusion), semaphores (https://en.wikipedia.org/wiki/Semaphore_(programming)), etc. from them. But we can

use, say, spinlock (https://en.wikipedia.org/wiki/Spinlock);
- they are called in a "softer" context, than ISR. In this context hardware interrupts are allowed. They displace tasklets for the time of the ISR execution. This context is called **softirq** in the Linux kernel and in addition to the running of tasklets, it is also used by several subsystems;
- a tasklet runs on the same core that schedules it. Or rather, has been the first one to schedule it by calling softirq, the handlers of which are always bound to the calling kernel;
- different tasklets can be running in parallel. But at the same time, a tasklet cannot be called concurrently with itself, as it runs on one kernel only: the kernel that has scheduled its execution;
- tasklets are executed by the principle of non-preemptive scheduling, one by one, in turn. We can schedule them with two different priorities: *normal* and *high*.

It's time to look under the hood and see how they work. First of all, the tasklet structure (defined in <linux/interrupt.h>):

```
struct tasklet_struct
{
    struct tasklet_struct *next;  /* The next tasklet in line for scheduling */
    unsigned long state;          /* TASKLET_STATE_SCHED or TASKLET_STATE_RUN */
    atomic_t count;               /* Responsible for the tasklet being activated or not */
    void (*func)(unsigned long);  /* The main function of the tasklet */
    unsigned long data;           /* The parameter func is started with */
};
```

Before using a tasklet, we should initialize it at first:

```
/* By default, the tasklet is activated */
void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data);
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data); /* the deactivated tasklet */
```

Tasklets are scheduled easily: the tasklet is placed into one queue out of two, depending on the priority. Queues are organized as singly-linked lists. At that, each CPU has its own queues. We can do it with the help of the following functions:

```
void tasklet_schedule(struct tasklet_struct *t);           /* with normal priority */
void tasklet_hi_schedule(struct tasklet_struct *t);        /* with high priority */
void tasklet_hi_schedule_first(struct tasklet_struct *t);  /* out of the queue */
```

When the tasklet is scheduled, its state set to **TASKLET_STATE_SCHED**, and the tasklet is being added to a queue. While it is in this state, we cannot schedule it, as nothing will happen in this case. The tasklet can be in several places in the queue for scheduling that is organized via the next field of the **tasklet_struct** structure. However, it is true for any lists bound via the object field, like <linux/list.h>.

For the time of execution, the tasklet state is set to **TASKLET_STATE_RUN**. By the way, the tasklet gets out the queue before the execution, and the **TASKLET_STATE_SCHED** state is removed. Thus, we can schedule it again during its execution. This can be done by the tasklet itself, as well as by the interruption on another kernel. However, in the latter case it will be called only after it will finish its running on the first kernel.

Interestingly enough, we can enable and disable the tasklet recursively. The following functions are in charge of this:

```
void tasklet_disable_nosync(struct tasklet_struct *t); /* disabling */
void tasklet_disable(struct tasklet_struct *t); /* disabling with the wait for the completion of ta
sklet’s operation */
void tasklet_enable(struct tasklet_struct *t); /* enabling */
```

If the tasklet has been disabled, we can still add it to the queue for scheduling, but it will not be executed on the CPU until it is enabled again. Moreover, if the tasklet has been disabled several times, it should be enabled exactly the same number of times, there is the *count* field in the structure for this purpose.

We can also kill tasklets. Like this:

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

At that, it will be killed after the execution only, in case if it has been scheduled already. If the tasklet schedules itself, we should not forget to forbid it doing this before we call this function. The programmer is responsible for this.

The most interesting are the functions playing the role of the scheduler:

```
static void tasklet_action(struct softirq_action *a);
static void tasklet_hi_action(struct softirq_action *a);
```

Since they are almost the same, there’s no sense in providing the code of both functions. Still, we should take a look at one of them to find out more details:

```
static void tasklet_action(struct softirq_action *a)
{
    struct tasklet_struct *list;
    local_irq_disable();
    list = __this_cpu_read(tasklet_vec.head);
    __this_cpu_write(tasklet_vec.head, NULL);
    __this_cpu_write(tasklet_vec.tail, &__get_cpu_var(tasklet_vec).head);
    local_irq_enable();
    while (list) {
        struct tasklet_struct *t = list;
        list = list->next;
        if (tasklet_trylock(t)) {
            if (!atomic_read(&t->count)) {
                if (!test_and_clear_bit(TASKLET_STATE_SCHED, &t->state))
                    BUG();
                t->func(t->data);
                tasklet_unlock(t);
                continue;
            }
            tasklet_unlock(t);
        }
        local_irq_disable();
        t->next = NULL;
        *__this_cpu_read(tasklet_vec.tail) = t;
        __this_cpu_write(tasklet_vec.tail, &(t->next));
        __raise_softirq_irqoff(TASKLET_SOFTIRQ);
        local_irq_enable();
    }
}
```

Pay attention to the call of **tasklet_trylock()** and **tasklet_lock()** functions. **tasklet_trylock()** sets the tasklet state to **TASKLET_STATE_RUN**, and thereby blocks the tasklet, which prevents the execution of the same tasklet on different CPUs.

In fact, these schedule functions implement cooperative multitasking that I have discussed in detail in this article (http://kukuruku.co/hub/opensource/multitasking-management-in-the-operating-system-kernel). The functions are registered as the handlers of *softirq* that is initiated during the scheduling of tasklets.

You can find the implementation of all the above functions in *include/linux/interrupt.h* and *kernel/softirq.c* files.

To Be Continued

In the next article, I will tell you about a much more powerful mechanism – the workqueue – that is also often used for the deferred processing of interrupts.

#os (http://kukuruku.co/tag/os/) #kernel (http://kukuruku.co/tag/kernel/) #linux (http://kukuruku.co/tag/linux/) #multitasking (http://kukuruku.co/tag/multitasking/) #multithreading (http://kukuruku.co/tag/multithreading/) #irq (http://kukuruku.co/tag/irq/) #tasklet (http://kukuruku.co/tag/tasklet/)

10434 2 (http://kukuruku.co/hub/nix/multitasking-in-the-linux-kernel-interrupts-and-tasklets#comments)

PUBLISHED BY

(http://kukuruku.co/profile/kukuruku/)

Kukuruku Hub (http://kukuruku.co/profile/kukuruku/)

RATING: 15.51

Follow

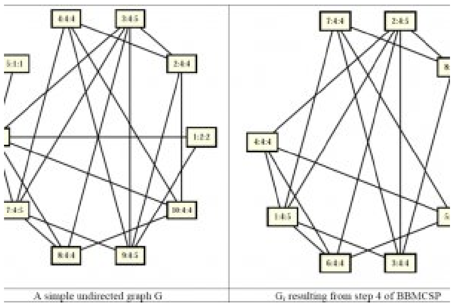
PUBLISHED IN

*nix (http://kukuruku.co/hub/nix/)

The *nix world is all about Unix-like systems, e.g., Linux, BSD, etc

RECOMMENDED

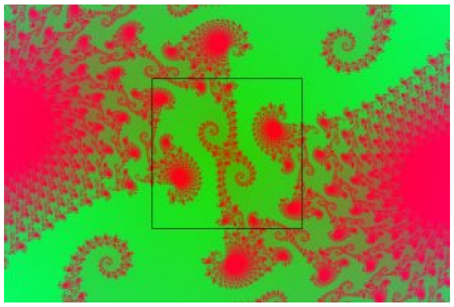
Algorithms (http://kukuruku.co/hub/algorithms/)


A simple undirected graph G
G, resulting from step 4 of BBMCSP

(http://kukuruku.co/hub/algorithms/exact-maximum-clique-for-large-or-massive-real-graphs)

Exact Maximum Clique for Large or Massive Real Graphs (http://kukuruku.co/hub/algorithms/exact-maximum-clique-for-large-or-massive-real-graphs)

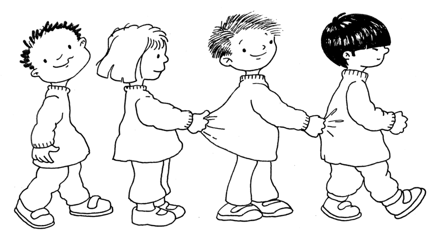
Algorithms (http://kukuruku.co/hub/algorithms/)



(http://kukuruku.co/hub/algorithms/julia-set)

Julia Set (http://kukuruku.co/hub/algorithms/julia-set)

C++ (http://kukuruku.co/hub/cpp/)



(http://kukuruku.co/hub/cpp/lock-free-data-structures-yet-another-treatise)

Lock-Free Data Structures. Yet Another Treatise (http://kukuruku.co/hub/cpp/lock-free-data-structures-yet-another-treatise)

Game Development (http://kukuruku.co/hub/gamedev/)

(http://kukuruku.co/hub/gamedev/how-i-built-my-first-android-game-and-realized-creativity-is-all-about-iterations)

How I built my first Android Game and realized creativity is all about iterations (http://kukuruku.co/hub/gamedev/how-i-built-my-first-android-game-and-realized-creativity-is-all-about-iterations)

DIY (http://kukuruku.co/hub/diy/)

(http://kukuruku.co/hub/diy/usb-killer)

USB Killer (http://kukuruku.co/hub/diy/usb-killer)

Programming (http://kukuruku.co/hub/programming/)

(http://kukuruku.co/hub/programming/i-do-not-know-c)

I Do Not Know C (http://kukuruku.co/hub/programm do-not-know-c)

Subscribe to Kukuruku Hub

Enter your e-mail address

Subscribe

Or subscribe with RSS (http://kukuruku.co/rss/index/)

2 comments

(http://kukuruku.co/profile/ak-linux/)

ak-linux (http://kukuruku.co/profile/ak-linux/) 31 January 2015, 10:14 PM **+1**

I am not sure if my previous comment got submitted, hence submitting again. When you say «tasklet run on same kernel» do you mean on the same CPU?

(http://kukuruku.co/profile/kukuruku/)
Kukuruku Hub (http://kukuruku.co/profile/kukuruku/) 31 January 2015, 10:52 PM ↑

Sorry, definitely, on the same *core*. Fixed that.

B () *I* () ~~S~~ () U () ❞ () </> () () ()

Any thoughts?

Preview Comment

Read Next

*nix (http://kukuruku.co/hub/nix/)

Multitasking in the Linux Kernel. Workqueues

8343

(http://kukuruku.co/hub/nix/multitasking-in-the-linux-kernel-workqueues)

Open Source (http://kukuruku.co/hub/opensource/)

Multitasking Management in the Operating System Kernel

17832

(http://kukuruku.co/hub/opensource/multitasking-management-in-the-operating-system-kernel)

*nix (http://kukuruku.co/hub/nix/)

Teaching the File System to Read

19718

(http://kukuruku.co/hub/nix/teaching-the-file-system-to-read)