Operating Systems 2

Bottom Halves

Arkadiusz Chrobot

Department of Information Systems

April 15, 2024

Outline

Introduction

It was stated in the previous lecture that modern operating systems split the hardware interrupt handling code into two parts: the top and bottom haves. The top half is the ISR. It must work as quickly as possible, because it is performed when the IRQ line associated with it or even all interrupts are disabled. Usually it does only the most important and necessary work related to interrupt handling and the rest is postponed and performed in a bottom half. Linux has several kinds of bottom halves. One of them was already introduced: the threaded interrupts. The top half verifies that the hardware interrupt was caused by the I/O device with which it is associated and returns a value that causes the kernel to wake up a thread that does the rest of interrupt handling. The kernel thread is performed in the process context, so it can sleep and be rescheduled. That way of servicing interrupts allows the kernel to perform other high–priority work first. In this lecture three other bottom halves are discussed. The rest, namely the kernel timers, will be introduced in a future lecture.

Introduction

There is no general rule about which activities related to interrupt handling should be performed in the top half and which in the bottom half. There are only some recommendations:

1. if the activities are time bounded, then they should be in top half,
2. if the activities require accessing the I/O device, then they should be in the top half,
3. if the activities cannot be interrupted by another or the same interrupt, then they should be in the top half,
4. any other activities may be performed in the bottom half.

To verify if the decision of putting some activities in the top or the bottom half was right the programmer must check the performance of the kernel after the solution is implemented.

Softirqs

Softirqs together with *tasklets* have replaced a bottom half infrastructure that had been used in the kernels predating the 2.4 series. However, the softirqs closely resemble this old mechanism. They are statically declared, which means that they cannot be used in kernel modules. The total number of softirqs is limited to 32, but this is more than enough. For example in the 5.10 version of the kernel only 10 is used (see the result of `cat /proc/softirqs`). All softirqs are performed in the interrupt context and in a multiprocessor computer system they can be performed in parallel. Each softirq is represented by a structure of the `struct softirq_action` type. The definition of the type is given in the listing 1.

```
1 struct softirq_action
2 {
3         void (*action)(struct softirq_action *);
4 };
```

Listing 1: Definition of the `struct softirq_action` type

Softirqs

In earlier versions of the kernel there was another member of this structure type, a field of the `void *` type, called `data`. It was not used, so it was removed. The member that is left is a pointer to a function called a *softirq handler*. The prototype of this function is as follows:

```
void softirq_handler(struct sortirq_action *);
```

The name of the function is usually different from what is used in this prototype. In earlier versions of the kernel the parameter was of the `void *` type. When the `data` field was removed it had to be replaced with some other pointer and it is the pointer to the structure of `struct softirq_action` type. It means that the function takes as an argument an address of the structure that encloses the pointer to it. An array named `softirq_vec` is defined in the `kernel/sofirq.c` file. It has 32 elements of the `struct softriq_action` type. In other words it is the array of softirq descriptors. The indices of the array also define the priorities of softirqs.

Softirqs

The softirq with the index 0 has the highest priority. There is also an enumerated type defined in the `linux/interrupt.h` header file, which contains convenient names to each of the indices. The `softirq_vec` array is used by the `__do_softirq()` function, which is responsible for invoking the softirq handlers. Before a softirq can be performed it has to be marked for execution by a top half. This is called *raising the softirq* and is performed by the `raise_softirq()` function which takes as an argument one element from the aforementioned enumerated type, which corresponds to the softirq that has to be raised. The function disables interrupts and sets one bit in a bitmap called `pending`, which has 32 bits. The position of the bit corresponds to the priority of the softirq and in consequence to its index in the `softirq_vec` array. After that the function re–enables the interrupts. If the interrupts are already disabled the `raise_softirq_off()` function can be used instead. Disabling interrupts is necessary, to prevent possible race conditions.

Softirqs

Usually the softirqs are preformed just after the ISR exits, but in some cases, when the kernel has some more important work to do, they can be deferred, for some unspecified, but usually short time. The kernel checks if there are any pending softirqs in the code performed after the top half finishes, or in the `ksoftirqd` kernel thread or in any other kernel code that explicitly verifies if there are any softirqs to be executed. If it is necessary to run the softirqs, then the `__do_softirq()` function is called. The function first disables the interrupts, copies the `pending` bitmap to a local variable, zeros out the bitmap and re–enables the interrupts. After that, in a loop it checks the value of the most significant bit in the bitmap local copy. If it is set, then the function invokes the first softirq handler from the `softirq_vec` array. Then, it shifts right by one bit the bitmap copy and advances to the next element of the array. Actually, it uses a pointer to the array, so it only increments the value of this pointer. The loop finishes when there are no more softirqs to run.

Softirqs

New softirqs can be registered in the kernel with the use of the `open_softirq()` function, which takes two arguments. The first one is the element of the enumerated type, which corresponds to the new softirq (it has to be first added to the type) and the second one is the pointer to the softirq handler (the name of it). The softirqs run when the interrupts are enabled and cannot sleep (they are performed in the interrupt context). Only one sortifq can run on a single processor at the same time, but in multiprocessor environment several softirqs can run simultaneously or even several instances of the same softirq can be performed at the same time (each on a single processor). Moreover the softirq handlers tend to use only data which are local for a given processor. It means that the softirqs are very scalable.

Tasklets

If a bottom half, that runs in the interrupt context, is needed in a kernel module, then a *tasklet* can be used. *Tasklets*[1] are closely related to the softirqs, but they are less scalable. In a multiprocessor computer system only one instance of the tasklet may be performed at the same time, but different tasklets can run simultaneously. However, tasklets may be used for works which are performed with high frequency. Each tasklet is represented by a structure of the `struct tasklet_struct` type. The structure has five fields. The first one is a pointer to the next structure of the same type (these structures are linked into a list). The second one is a field that stores the state of the tasklet. This member can have only one value from the following three: 0 — the tasklet is inactive, `TASKLET_STATE_RUN` — used in multiprocessor environments to indicate that the tasklet is already running on one of the processors, `TASKLET_STATE_SCHED` — the tasklet is scheduled for running.

[1] Not to be confused with a *task*, which in Linux terminology is the same as a process, but generally it is a process, which is not interactive.

Tasklets

The third field is a reference counter. If its value is greater than zero, then the tasklet is disabled, and when it is zero, it is enabled. The fourth is a pointer to the tasklet handler. It is a function of the following prototype:

```
void tasklet_handler(unsigned long);
```

The name of the actual handler can be different. The function takes only one argument — data for the tasklet. The fifth field is of `unsigned long` type and it stores data for the tasklet. There are two types of tasklets in the Linux kernel: high–priority and regular. The high–priority tasklets are grouped in a list which is traversed by the highest priority softirq handler. This function is responsible for performing these tasklets. The regular tasklets are also linked into a list, which is traversed by the softirq handler of the 6^{th} priority (counting from zero). The function performs the tasklets too.

Tasklets

The list of the regular tasklets is named `tasklet_vec` while the list of high–priority tasklets is called `tasklet_hi_vec`. Both types of tasklets are managed by the same kernel functions, with one exception. The high–priority taskles are scheduled (added to the list) with the use of the `tasklet_hi_schedule()` function and the regular ones are scheduled with the use of the `tasklet_schedule()` function. If a tasklet is already added to one of the lists then it cannot be added to the same list until it is performed. A tasklet, regardless if high–priority or regular, can be declared with the use of the `DECLARE_TASKLET` macro. If the tasklet has to be disabled after creating the `DECLARE_TASKLET_DISABLED` macro can be used. The tasklet structure can be initialized with the use of the `tasklet_init()` function. If a tasklet that is scheduled needs to be disabled the `tasklet_disable()` function can be applied. If the tasklet is already running then the function waits until it finishes and only then it exits.

Tasklets

A less safe version of the `tasklet_disable()` function is called `tasklet_disable_nosync()`. The latter doesn't wait even if the tasklet is already running. A scheduled tasklet can be removed from the list with the use of the `tasklet_kill()` function. It cannot be used in the interrupt context, because it may wait for the tasklet to exit. To enable a tasklet the `tasklet_enable()` function is used. The tasklets are performed in the interrupt context. They may be scheduled by an ISR or by other code. Just like in the case of the softirqs, there is no way of telling when exactly tasklets will be performed. To learn the details of the tasklet API please refer to the sixth laboratory instruction.

The `ksoftirqd` Kernel Thread

Softirqs and tasklets that are repeated with a high frequency or reactivate themselves can pose a problem to the system. They may create too much load for the CPU or CPUs. To mitigate the problem handling of such softirqs and tasklets is delegated to the `ksoftirqd` kernel thread. Each CPU has one instance of this thread, which runs with the lowest possible priority. When the thread is woken up, it checks if there are any softirqs (and thus tasklets) pending. If so it calls the `__do_softirq()` function. After the function exits the thread changes it state to `TASK_INTERRUPTIBLE` and goes to sleep.

Work Queues

Work queues[2] provide a way of performing a bottom half in the process context. Each work, that has to be performed by a work queue is represented as a work item — an element of a queue (list). The work item points to a function which implements the work. The work queue is traversed by a special kernel thread called a *worker thread* that performs the `worker_thread()` function and also invokes functions pointed by work items.

Worker threads are grouped in *thread–pools*. In a multiprocessor environment each CPU has two thread pools, one for high–priority work queues and one for regular work queues. The kernel also has so–called unbound work queues, that are not permanently associated with any specific CPU. The number of threads in each pool is self–regulated.

[2]The work queues or workqueues have replaced a bottom half implementation that existed in the kernel before the 2.6 series was released and was called *task queues*. Those tasks were unrelated to the processes in any way aside from the name.

Work Queues

The work queue is represented in the kernel by a structure of the `struct workqueue_struct` type (in earlier kernel versions it represented a worker thread). Linux has a default work queue which is handled by a worker thread called `kworker`, but new work queues can be created with the use of the `alloc_workqueue()` function which takes three arguments. The first one is a string that represents the name of the queue and also the name of so–called rescue worker thread. Those threads are used for servicing work queues which work items are involved in memory reclaim. The next argument is a set of following flags:

`WQ_NON_REENTRANT` by default there can be many instances of a work function run in a multiprocessor computer system; if the flag is given then only one instance of the function can be performed system–wide,

Work Queues

`WQ_UNBOUND` the queue is not associated with a specific CPU,

`WQ_FREEZABLE` the queue will participate in hibernating the system,

`WQ_MEM_RECLAIM` the queue will participate in memory reclaim,

`WQ_HIGHPRI` the queue will handle high–priority work items,

`WQ_CPU_INTENSIVE` CPU intensive work items in the queue will not prevent other work items handled by the same thread–pool from starting execution. This flag has no effect for unbound queues.

The last argument of the `alloc_workqueue()` function is a number that specifies how many work items can be performed concurrently at most.

Work Queues

There are also available two macros that create new work queues. The first one is named `create_workqueue` and it creates a work queue which is serviced by as many worker threads as the computer has CPUs. The other one is called `create_singlethread_workqueue` and it creates a work queue that is handled by only one worker thread. Nowadays implementation of the work queue doesn't create a fixed number of threads for each of the queues. The kernel monitors the load of the CPUs and the number of work items in a work queue and dynamically adds worker threads if they are needed or removes them if they are redundant. Any work queue, except for the default one, may be removed with the use of the `destroy_workqueue()` function.

Work Queues

The work item is represented by a structure of either the `struct work_struct` type or `struct delayed_work` type. The first one is for works that are postponed for unspecified time and the second one for works with a specified time period before they start. The work queue guarantees only that the delayed works won't start before the specified delay, but it doesn't assure that these works will be performed immediately after. Each work item points to a function called a *work handler*, which has the following prototype:

```
void work_handler(struct work_struct *work);
```

The actual work function, or work handler doesn't have to be named like that. The code in the function may cause the worker thread to go to sleep, but it cannot access the user–space. The work item structures can be created with the use of the `DECLARE_WORK` and `DECLARE_DELAYED_WORK` macros. The first one creates a work item of the `struct work_struct` type and the second one a work item of the `struct delayed_work` type.

Work Queues

Structures of the first type can be initialized with the `INIT_WORK` macro and the structures of the second type can be initialized with the `INIT_DELAYED_WORK` macro. To add an initialized work item to the default work queue the `schedule_work()` function may be applied. If the work has to be performed on a particular CPU the `schedule_work_on()` function can be used. A work item represented by the structure of the `struct delayed_work` type can be added with the use of the `schedule_delayed_work()` function to the default work queue. If the work has to be done on a specific CPU, then the `schedule_delayed_work_on()` function may be used. The `flush_scheduled_work()` function forces execution of all work handlers scheduled in the default work queue.

Work Queues

The work queues API has functions that can be applied to any work queue. The `queue_work()` and `queue_delayed_work()` functions add to a specified work queue a work item represented by a structure of the `struct work_struct` type or the `struct delayed_work` type respectively. If the work has to be performed on a specific CPU then the `queue_work_on()` or the `queue_delayed_work_on()` function may be used instead. If delayed work is already scheduled then the period after which it can be performed may be changed with the use of the `mod_delayed_work()` or the `mod_delayed_work_on()` function. The function `cancel_work_sync()` cancels scheduled work. If the work handler is already running then the function will wait until it exits. Likewise the `cancel_delayed_work_sync()` function cancels the delayed work. The `cancel_delayed_work()` also cancels delayed work, but in less safe fashion — it doesn't check if the work handler is already running. The `flush_work()` function waits until a specific work is performed. It returns immediately if the work is not scheduled for execution.

Work Queues

The `flush_delayed_work()` does the same for specified delayed work. Finally, the `flush_workqueue()` function forces the execution of all work items added to a specified queue and waits until the last work handler exits.
For more details on the work queue API please refer to the sixth laboratory instruction.

Questions

?

The End

Thank You for Your attention!