Laboratory 7: "Timers" (one week)

Arkadiusz Chrobot, PhD
Karol Tomaszewski, PhD

May 11, 2024

Contents

Introduction

Timers, also called *kernel timers* are one of the mechanisms of the bottom halves. Operations executed within the timers are performed in the context of the interrupt. Two types of these mechanisms are available: *low-resolution timers* that count the time in system clock ticks and *high-resolution timers* that measure the time in nanoseconds (1 nanosecond = 10^{-9} seconds). Chapter 1 of this guide is related to low-resolution timers and Chapter 2 describes the high-resolution timers. The document ends with a list of tasks to be carried out as part of the laboratory.

1. Low-Resolution Timers

The mechanism of low-resolution timers, also known as the „*timers wheel*" has historically appeared earlier in the Linux kernel (before the high-resolution timers), but since version 2.6.16 the operation of low-resolution timers is based on the mechanism of high-resolution timers. The delay for low-resolution timers is counted in the system clock ticks. The duration of such tick depends on the hardware platform on which Linux runs. For example, for computers with x86 processors, a single clock cycle lasts 4 ms, while for a Raspberry Pi 3 computer, its length is 10 ms. For most typical applications, this resolution is sufficient. The low-resolution timer after being scheduled is processed only once. To be re-executed, it must be re-scheduled with a new delay value, which is determined in relation to the beginning of the operating system time being measured. The Linux kernel stores the number of clock cycles since it was started in the `jiffies` variable. Most often new execution time is calculated as the sum of the current value of the `jiffies` variable and the number of system clock ticks after which the timer should be executed. In other words the `jiffies` variable is used as the reference point marking the moment of the current execution.

1.1. API Description

To use the low-resolution timers in the module, it is necessary to include the `linux/timer.h` header file in its source code. Such timers are represented in the system kernel with the variables of type `struct timer_list`. For a programmer using low-resolution timers, only three members of this structure are relevant:

1. `expires` - contains the number of a system clock ticks after which the function associated with the timer is to be called,
2. `function` - is a pointer to the mentioned function,
3. `data` - contains an `unsigned long int` number that is given as the call argument to the function associated with the timer.

The operations to be performed within the low-resolution timer must be programmed in the function with the following prototype:

```
void timer_handler(unsigned long int data);
```

Both the function name and parameter name may be different. The source code of this function must meet the same requirements as for the function implemented within the tasklet. Any number that does not exceed the `unsigned long int` range can be given to this function as its call argument. This value can be used to distinguish timers when the function is called from not one, but several timers. The following functions and macros are used to handle the low-resolution timers:

`init_timer(timer)` - a macro that initializes a `struct timer_list` structure whose address is given to it as a call argument. This initiation does not include the fields described above, hence the programmer must initiate them explicitly.

`void add_timer(struct timer_list *timer)` - a function that schedules (activates) the timer to be performed. As the call argument, it takes an address of the `struct timer_list` structure that represents this timer in the system.

`int mod_timer(struct timer_list *timer, unsigned long expires)` - a function that allows to modify the delay after which the timer will be started. If the timer has not been activated yet, the function will activate it and return `0`. If the timer was activated, the function will modify its delay and return the value `1`. **The delay of timers that have already been activated can only be changed using this function.**

`int del_timer(struct timer_list * timer)` - function deactivating the timer. As a call argument it takes an address of the `struct timer_list` structure that represents the timer to be deactivated. This function returns `0` if the timer was not activated before it was called and it returns `1` if it was activated already.

`int del_timer_sync(struct timer_list *timer)` - a function which, in the case of multiprocessor systems, deactivates the timer and additionally checks whether the function associated with that timer is performed on any of the processors. If that is the case then the function waits with the end of its execution until the timer function has been completed. It returns the same values as `del_timer()`. It is recommended to use it instead of `del_timer()` for both multiprocessor and single processor systems. In the case of the latter, the compiler automatically replaces the call to this function with a macro with the same name, which is then expanded to call the `del_timer()` function.

1.2. Example

Listing 1 contains the source code of the module, which demonstrates the use of a low-resolution timer.

Listing 1: Example module using low-resolution timer

```c
#include<linux/module.h>
#include<linux/timer.h>

static struct timer_list timer;

static void timer_handler(unsigned long int data)
{
        pr_info("Timer nr %lu is active!\n",data);
}

static int __init timer_module_init(void)
{
        init_timer(&timer);
        timer.expires = jiffies + 15*HZ;
        timer.data = 0;
        timer.function = timer_handler;
        add_timer(&timer);
        return 0;
19 }
20
21 static void __exit timer_module_exit(void)
22 {
23         if(del_timer_sync(&timer))
24                 pr_notice("The timer was not active!\n");
25 }
26
27 module_init(timer_module_init);
28 module_exit(timer_module_exit);
29 MODULE_LICENSE("GPL");
30 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
31 MODULE_DESCRIPTION("A module demonstrating the usage of timers.");
32 MODULE_VERSION("1.0");
```

In line 2 of the module source code, the `linux/timer.h` header file is included, the content of which allows using the low-resolution timer mechanism. In line 4 a structure called `timer` of type `struct timer_list` has been defined, which will represent a single low-resolution timer. Lines 6-9 contain the definition of the function that will be performed within this timer. Its operation is simple. It places a message in the kernel buffer containing the timer number provided to it by the `data` parameter. It is assumed in this module that the timers will be numbered starting from 0. In the module constructor (lines 11-19), the timer structure is first initialized by the `init_timer()` function, and then values are assigned to its `expires`, `data` and `function` fields. In the `function` field, the address of the `timer_handler()` function is stored, which will be executed as part of the timer. In the `data` field a number is stored which will be given to the timer function as a call argument (it is `0`). The `expires` field stores the delay value for this timer, which is calculated as the sum of the current value of the `jiffies` variable and the product of `15*HZ`, where `HZ` is a constant that determines the frequency of the system clock (the number of cycles that system clock does in a second). This expression therefore ensures that the timer will be executed after approximately[1] fifteen seconds from the current time. After initializing these fields, the timer is activated using the `add_timer()` function. In the module destructor (lines 21-25), the timer is deactivated before removing the module from the kernel, with the use of `del_timer_sync()` function. If the timer has not been activated by then, the module will put a respective message in the kernel buffer.

2. High-Resolution Timers

High-resolution timers, similar to low-resolution timers are not a precise mechanism, but offer time countdown with nanosecond resolution, which is required for some applications, such as multimedia support. In order for the full capabilities of the high-resolution timers mechanism to be available, there must be an appropriate clock available in the system, which will be the time source for this subsystem. Besides of that the kernel must be compiled with the appropriate option enabled (currently this option is selected by default). If these conditions are not met, high-resolution timers API will be available, but the timers themselves will work with the same resolution as low-resolution timers. Most often, in modern computer systems, two types of clocks are available in the system that meet the requirements of the high-resolution timer mechanism. These types of clocks are identified by two constants: `CLOCK_MONOTONIC` and `CLOCK_REALTIME`. The first type consists of clocks, the indications of which increase at the same rate from the moment the computer is turned on until it is turned off. The second type consists of clocks counting down the „*wall-clock time*”, i.e. the current time, related to a given time zone. Their indications can therefore be corrected e.g. by the system administrator or by the subsystem responsible for handling the NTP protocol, which means that they may increase at an uneven pace or even go backwards. More information, although somehow outdated, about this mechanism can be found in the article available at: `https://lwn.net/Articles/167897/`.

[1]The low-resolution timers are not precise. It should be also taken into account the time that will elapse from the moment the value is assigned to the `exipres` field until the timer is activated.

2.1. API Description

The high-resolution timers interface becomes available after including the `linux/hrtimer.h` header file in the source code, however, to use these timers, the `ktime_t` type from the `linux/ktime.h` header file have to be checked out as well. This file does not need to be included in the module source code, because it is also included in the `linux/hrtimer.h` file. The `k_time` type defines a union that contains a single field for storing time in nanoseconds. Due to compatibility with earlier versions of the kernel, this union cannot be replaced with a simple type variable and must be handled by, among the others with the following macros and functions:

`ktime_t ktime_set(const s64 secs, const unsigned long nsecs)` - an `inline` function that returns a `k_time` union containing a time indication that is calculated based on the values passed to it. The first call argument is the number of seconds, and the second argument is the number of nanoseconds.

`ktime_sub(lhs, rhs)` - a macro that subtracts the value of the `rhs` union of type `k_time` from the value of the `lhs` union. The result returned by this macro is also of type `k_time`.

`ktime_add(lhs, rhs)` - a macro that adds the values of `k_time` unions given as call arguments and returns the result also in the form of a `k_time` union.

`ktime_add_ns(kt, nsval` - a macro that adds the number of nanoseconds given by `nsval` call argument to a value of the `k_time` union given by the `kt` call argument. The result is also returned in the form of a `k_time` union.

`ktime_to_ns(kt)` - a macro that converts the time represented in the `ktime_t` union given as a call argument to the respective number of nanoseconds.

High-resolution timers are represented in the system kernel with the `struct hrtimer` structures. For a programmer using this type of timers, the most important field in this structure is the `function` field, which is a pointer to the function executed within the timer. This function must have the following prototype:

```
enum hrtimer_restart hrtimer_handler(struct hrtimer *hrtimer);
```

The function name given in this prototype and the parameter name can be changed. Because, as in the case of low-resolution timers, this function is performed in the context of an interrupt, it cannot contain any operations that are associated with putting the thread into the waiting state, because this context is not associated with any kernel thread. This function should return one of two values that are contained in the `enum hrtimer_restart` enumeration type. Those are: `HRTIMER_NORESTART` and `HRTIMER_RESTART`. If the function returns the first of these, it will mean that the timer will not be automatically activated and re-executed. If the function returns the second of the described values, the timer will be automatically activated and re-executed. The following functions, are related to handling the high-resolution timers:

`void hrtimer_init(struct hrtimer *timer, clockid_t which_clock, enum hrtimer_mode mode)` - this function initializes a `struct hrtimer` structure whose address is given as its first call argument. As a second argument, one of the two constants described earlier is used to specify the type of clock used to count down the time for the timer. The third call argument should be a value that is an element of the `hrtimer_mode` enumeration type, which determines how the delay should be counted for the timer. Usually, one of the two following elements is used: `HRTIMER_MODE_ABS` - the delay will be calculated relative to the moment the computer is started, or `HRTIMER_MODE_REL` - the delay will be calculated relative to the current moment. This function does not initialize the `function` field, which must be initialized using the normal assignment operator.

`void hrtimer_start(struct hrtimer *timer, ktime_t tim, const enum hrtimer_mode mode)` - an `inline` function that activates a high-resolution timer represented by a `struct hrtimer` structure whose address is given as its first call argument. As the second call argument, a `k_time` union is given containing the number of nanoseconds after which the timer function should be executed. The third call argument is the same as for the `hrtimer_init()` function.

`int hrtimer_cancel(struct hrtimer *timer)` - a function that deactivates the timer represented by the `struct hrtimer` structure whose address is given to it as a call argument. If the timer-related function is already being executed, the `hrtimer_cancel()` function is waiting for its completion.

This function returns `0` if the canceled timer was not yet activated, or it returns `1` if the timer was activated already.

`int hrtimer_try_to_cancel(struct hrtimer *timer)` - this function works similar to `hrtimer_cancel()` function. The difference between them is that if the function associated with the canceled timer is in progress, then the `hrtimer_try_to_cancel()` function does not wait for its completion, but immediately returns the value `-1`.

`u64 hrtimer_forward(struct hrtimer *timer, ktime_t now, ktime_t interval)` - this function is usually called by the function executed within the timer. It is responsible to set a new time to reactivate this timer. It takes three call arguments. The first call argument is the address of a `struct hrtimer` structure representing the timer. The second argument is the time in nanoseconds, stored in a `k_time` union. The new delay will be counted relatively to this time. The third argument is the delay, calculated in nanoseconds and stored in a `ktime_t` union. The function returns a number specifying how many delay periods the timer has waited for execution. If the system is not overloaded and works properly, this number is usually `1`.

`u64 hrtimer_forward_now(struct hrtimer *timer, ktime_t interval)` - an `inline` function, which is a simpler equivalent of the `hrtimer_forward()` function, because it does not require to give the time over which the delay is calculated.

`void hrtimer_restart(struct hrtimer *timer)` - an `inline` function that activates a timer that was previously canceled. It takes one call argument, which is the address of a `struct hrtimer` structure representing this timer.

`ktime_t hrtimer_get_expires(const struct hrtimer *timer)` - an `inline` function that returns a `ktime_t` union representing a delay after which the function associated with the timer will be started. The address of a `struct hrtimer` structure representing this timer is given to the function as a call argument.

`s64 hrtimer_get_expires_ns(const struct hrtimer *timer)` - an `inline` function that works similarly to the `hrtimer_get_expires()` function, but returns the delay in expressed in a number of nanoseconds.

`ktime_t hrtimer_expires_remaining(const struct hrtimer *timer)` - an `inline` function that returns the time remaining to run the timer with which the `struct hrtimer` structure is associated, whose address was given to the function as the call argument. The returned value is stored in a `k_time` union.

`int hrtimer_is_hres_active(struct hrtimer *timer)` - this function takes as the call argument the address of the `struct hrtimer` structure associated with the timer and returns `1` if it works as a high-resolution timer or `0` if it works as a low-resolution timer.

2.2. Example

Listing 2 contains the source code of the module, which demonstrates the use of a high-resolution timer.

Listing 2: Example module using a high-resolution timer

```c
#include<linux/module.h>
#include<linux/hrtimer.h>

static struct hrtimer timer;
static ktime_t delay;

static enum hrtimer_restart hrtimer_function(struct hrtimer *hrtimer)
{
        u64 overruns;
        pr_info("The timer is active!\n");
        overruns = hrtimer_forward_now(hrtimer,delay);
        pr_info("The overruns number since last activation: %llu.\n", overruns);
        return HRTIMER_RESTART;
}

```

```c
16 static int __init hrtimer_module_init(void)
17 {
18         delay = ktime_set(1,0);
19         hrtimer_init(&timer,CLOCK_MONOTONIC,HRTIMER_MODE_REL);
20         timer.function = hrtimer_function;
21         hrtimer_start(&timer,delay,HRTIMER_MODE_REL);
22         return 0;
23 }
24 
25 static void __exit hrtimer_module_exit(void)
26 {
27         if(!hrtimer_cancel(&timer))
28                 pr_alert("The timer was not active!\n");
29 }
30 
31 module_init(hrtimer_module_init);
32 module_exit(hrtimer_module_exit);
33 MODULE_LICENSE("GPL");
34 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
35 MODULE_DESCRIPTION("A module demonstrating the use of high-resolution timers.");
36 MODULE_VERSION("1.0");
```

The `linux/hrtimer.h` header file is included in the module source code in line 2. In line 4, a `struct hrtimer` variable is declared that will represent the timer, and in line 5 a `delay` variable is declared, which is a `ktimer_t` union and will store the number of nanoseconds after which the timer will be activated. Lines 7-14 contain the definition of the function that will be performed within the timer. This function is placing a message in the kernel buffer about the timer starting and the number of delays that have passed since its last activation (the correct value is `1`). It also reactivates the timer with which it is associated. It does this by changing the delay time of this timer using the `hrtimer_forward_now()` function. To call it, the `hrtimer_function()` gives the pointer to the timer structure (which it received by its parameter) as well as the delay value stored in the `delay` variable. Direct references to this variable are allowed in functions, because it is a variable declared with the `static` keyword, so it is local inside the module, while it is not available outside. This is a common practice in C language libraries for user space, so it can also be used in the kernel library, which is the module. The value returned by `hrtimer_forward_now()` function is stored in the local variable and this is the number of delays mentioned earlier. For the timer to be reactivated, the `hrtimer_function()` must also return `HRTIMER_RESTART` value. In the module constructor, the `delay` variable is first initialized (line 18) using the `ktime_set()` function. The number of nanoseconds stored in it is equivalent to one second. In line 19, the `timer` variable is initialized using the `timer_init()` function. The monotonic clock is selected as the time source for the timer represented by this variable, and the delay will be calculated relative to the current time. In line 20, the address of the `hrtimer_function()` function is assigned to the `function` field of the `timer` variable. This function will be executed within the timer. In line 21, the timer is activated with the delay determined by the value of the `delay` variable (1 second), calculated relative to the current moment. This activation is done by calling the `hrtimer_start()` function. In the destructor, the timer is canceled before removing the module from the system kernel, with the use of the `htimer_cancel()` function. It is convenient to observe the operation of this module using the `dmesg -w -d` command.

Exercises

1. `[3 points]` Modify the function related to the timer from module 1 so that it reactivates this timer.
2. `[5 points]` Write a kernel module in which the low resolution timer will be cyclically activated, but its delay with each activation will be doubled. The timer should be stopped when the value of this delay exceeds the threshold you set, e.g. 10 seconds.
3. `[7 points]` Write a module in which two high-resolution timers will be used. The first timer should have a long delay. The second timer should be performed cyclically and using the appropriate

functions described in section 2.1 should output information about the first timer in the file, in the `procfs` file system.

4. `[3 points]` Modify the function related to the timer from module 2 so that this timer is run only once.

5. `[5 points]` Write a module in which two high-resolution timers will run functions sharing one variable. Each of these functions should place the value of this variable in the kernel buffer and then modify it. The function associated with the first timer can e.g. increase it by two, and the second function can increase it by three.

6. `[7 points]` Write a module in which the function performed as part of a cyclically running high-resolution timer will create and append elements to the end of the list. These elements should contain numbers, e.g. pseudo-random numbers. A second timer, also of high-resolution timers, should periodically read the contents of the elements of this list and place it in the kernel buffer. Secure the module so that the first timer does not create more than 100 elements. The list should be deleted before removing the module from the system kernel.