Operating Systems 2

Time Management And Timers in Linux

Arkadiusz Chrobot

Department of Information Systems

May 13, 2025

Outline

1. Introduction
2. Time Management
3. Low-Resolution Timers
4. High-Resolution Timers
5. Delaying Execution

Introduction

The information about time is vital to the kernel and user-space applications alike. Some of the kernel functions are activated at specific moments of time, which means that the kernel needs to measure the *relative time* (from a specific moment to another). User applications, such as databases management systems, web applications and so on, need information about *absolute time*, also called *wall time*, *real-world time* or *wall-clock time*. The kernel has to address both of this requirements. In this lecture subsystems of the kernel responsible for time-keeping are described. Also the timers are discussed.

Time Management

The main component that allows the kernel to measure time is a *system timer*, which generates an interrupt at a predefined rate. The interrupt is then handled by a dedicated ISR. The hardware platforms supported by Linux have many devices that can assume the role of the *system timer*. Their operation is usually based on electric impulses generated by a crystal oscillator. For example, in computers based on $x86$ CPUs there are several types of such devices: the oldest one called PIT (*Programmable Interval Timer*), the more modern ▸ HPET (*High Precision Event Timer*), and the (L)APIC timer. Other hardware platforms are equipped with similar devices. Each of these hardware timers offers a different accuracy and requires a different handling. In the 2.6 series of kernels, Linux programmers added an abstraction layer that hides most of these differences and unifies the handling of these devices.

Virtual Clocks

The abstraction layer uses hardware timers to create two types of virtual clocks:

clock sources are monotonically incremented counters with read-only access,

clock event devices generate an interrupt (event) at a given moment in the future; they may be programmed to do so only once, or repetitively.

Each ▸ clock source and clock event device has assigned a natural number that represents its quality. The virtual clock with the highest quality is chosen as the default clock source or clock event device in the system. Moreover, the default clock event device assumes the role of the system timer.

The HZ Constant

The frequency of the system timer is specified by the HZ constant[1]. The timer period is the inverse of this constant. For most hardware platforms supported by Linux, the value of HZ is 100. There are several exceptions, like the computer systems based on the $x86$ CPUs. Initially, the value of HZ for these hardware platforms was also 100 (period of 10 ms), but it had been changed in the 2.4 series of kernels to 1000 (period of 1 ms), to address the needs of user-space multimedia applications. This had resulted in a better resolution of the timer interrupt and better control over the time-driven kernel activities. Unfortunately, the increased frequency of the system timer had caused overloading of the CPU with servicing timer interrupts. Moreover, the change had conflicted with handling the NTP protocol (the *Network Time Protocol*). That's why the value of the HZ constant for computers with $x86$ CPUs has been finally set to 250 (period of 4 ms). The issues of multimedia applications have been resolved with high resolution timers.

[1] The Hz (hertz) is a unit of frequency.

Dynamic Ticks

For hardware platforms that require energy saving support, like embedded systems and laptops, the kernel can be configured to disregard the `HZ` constants and program the system timer to generate time events when they are needed. Servicing the timer interrupt each time it occurs costs some energy. If energy saving is a priority in the computer system, and the kernel and the user-space applications do not have to do anything within, for example, the next two seconds, then the kernel will generate the timer interrupt after 2 seconds and not each 4 ms.

The jiffies Variable

The number of the system timer interrupts, generated since the computer system boot up, is stored in the jiffies variable. The time period when the computer is working (so-called *uptime*) is equal to jiffies/HZ. In the 32-bit computers, jiffies is 32-bit wide, and in the 64-bit computers, it is 64-bit wide. Rising the frequency of the system timer for the 32-bit $x86$ CPU computers, caused a problem with jiffies overflow. With the value of HZ set to 100, the jiffies overflows each 497 days. The interval was reduced to 49.7 days, when the value of HZ had been changed to 1000. The issue has been solved with adding a 64-bit wide variable named jiffies_64. In the 32-bit computers, the jiffies variable overlaps the 4 least significant bytes of the jiffies_64 variable, while in 64-bit computers these names refer to the same variable. In the 32-bit computers, the jiffies_64 should be read with the help of the get_jiffies_64() function, which assures indivisible access to this variable.

The jiffies Variable

Kernel programmers have defined four macros that make comparing time expressed with the use of values of the `jiffies` variable much easier. They take into account the overflows of that variable. Each of these macros takes two arguments, which are the values of the `jiffies` variable. The `time_after` macro evaluates to a nonzero value (true) when the moment referred by its first argument happens after the moment referred by its second argument. The `time_before` macro does the same, if the first argument refers to a moment that happens before the moment referred by the second argument. Two others macros (`time_after_eq` and `time_before_eq`) behave similarly, but they also evaluate to a nonzero value when both arguments refer the same moment in time.

USER_HZ

User-space applications assume that the value of the HZ constant is 100. This is true for most of hardware platforms, with some exceptions, like the computers based on $x86$ processors or the DEC Alpha CPUs. For such computer systems the kernel programmers defined a separate constant named USER_HZ, that is set to the value of the system timer frequency expected by the user-space applications. To convert the actual number of timer interrupts to a number that corresponds to the value of the USER_HZ constant, the jiffies_to_clock_t() or the jiffies_64_to_clock_t() functions are used.

The Real-Time Clock

The Linux kernel also tracks the wall-clock time, because the current date and time are needed by some user-space programs. Usually, this information is provided by a hardware device[2], which is read by the kernel during the system boot up and then only updated by the timer interrupt handler. The kernel registers the number of seconds since the midnight of 1^{st} of January 1970 (the starting point of so-called Unix Epoch) and the number of nanoseconds since the beginning of the last second[3].
The information about current date and time is provided to user-space software via the `gettimeofday()` system call.

[2] A notable exception are the Raspberry Pi computers, up to version 5.

[3] This way of storing information about time may cause in the future a problem, called Y2K38, for 32-bit computers.

The Timer Interrupt Servicing Routine

The code responsible for handling the system timer interrupt is split into two parts: one hardware-dependent and one hardware-independent. The first one performs such operations as:

- servicing the system timer device,
- periodically updating the wall-clock device,
- invoking the `tick_periodic()` function, which is an implementation of the hardware-independent part.

The second one does the following:

- updates the `jiffies_64` variable,
- updates the resource usage by the current process for each of the CPUs,
- updates the variables storing the wall-time,
- calculates the average load of system.

Low-Resolution Timers

The kernel provides *timers* (also known as *kernel timers* or *dynamic timers*) that are a way for deferring some operations to be done, for a given amount of time. These operations are performed in the interrupt context. In other words, timers are another implementations of bottom halves. There are two types of such timers. The first one are the *low-resolution* timers, also known as the *timer wheel*. The resolution of these timers is of the length of the system timer period. They are not suitable for real-time or multimedia applications, but are good enough for any other purposes. The low-resolution timers are also not cyclic, which means that when they expire they are not automatically renewed. A single low-resolution timer is represented by the `stuct timer_list` type variable. After initialization of the structure, the values of two its members have to be set: `expires` and `function`.

Low-Resolution Timers

The first one defines the moment when the timer should expire. This value is expressed in periods of the system timer. The second one stores the address of a function implementing the operations that needs to be performed when the timer expires. The prototype of the function is as follows:

```
void timer_function(struct timer_list *);
```

The function gets as an argument the address of the timer structure. The members of the `struct timer_list` variable may be initialized with the help of the `timer_setup` macro. The timer is activated with the help of the `add_timer()` function.

Low-Resolution Timers

The moment of expiry for an active timer can be changed with the use of the `mod_timer()` function. This is the only safe way of modifying that value for such a timer. If this function is applied to an inactive timer it will activate the timer. In uniprocessor systems an active timer can be removed with the use of the `del_timer()` function. On multiprocessor systems the `del_timer_sync()` function should be used for this purpose. The `timer_pending()` function returns `1` if it is invoked for an active timer or `0` otherwise. The function that implements operations performed by the timer should use synchronization devices protecting the shared resources it accesses. The timer's function is called by the sortirq handler, when the timer expires. Before the version 4.8 of Linux kernel the low-resolution timers were stored in a linked list, that was unsorted, but divided into five parts. The timers were added to those parts, depending on their expiration time, and then they were moved from one group to another until they expired.

Low-Resolution Timers

The main advantage of this ▸ approach was that the average inaccuracy of low-resolution timers was about half of the system timer period. Unfortunately, the cost of moving the timers inside the list was too big. Thomas Gleixner added in the 4.8 version of the kernel a patch, that eradicates the need for such an operation, but increases the inaccuracy of the timers. In case of timers with very long expiry period, it can even be several hours. The internal work of the new low-resolution timers subsystem[4] depends strongly on the value of the `HZ` constant. Here, it is described for the case when this value is 250. The kernel creates a hierarchy of 9 arrays (in case of other values of `HZ` it can be 8). Each of the arrays has 64 elements that are pointers to a timer list.

[4] The description is based on the following article: https://lwn.net/Articles/646950/ and also on the comments in the `timer.c` file: (https://elixir.bootlin.com/linux/v4.8/source/kernel/time/timer.c).

Low-Resolution Timers

The highest array in the hierarchy gathers timers with the expiry period ranging from 0 ms to 255 ms and their resolution (or *granularity*) is 4 ms (2^2 ms). The maximal inaccuracy is also 4 ms. The resolutions offered by other arrays are as follows: 32 ms (2^5), 256 ms (2^8), 2048 ms (2^{11}), 2^{14} ms, 2^{17} ms, 2^{20} ms, 2^{23} ms, 2^{26} ms. The inaccuracy in case of the lowest array in the hierarchy is about 18 h, but the expiry period of timers in that array is ranging from 6 to 49 days. In this case the accuracy of low-resolution timers doesn't matter very much, because most of them is used as *watchdogs* and usually they are canceled before their expiry. The accuracy of timers with short expiry period is the same as in the old implementation of the timer wheel and the operation of moving the timers inside the list is eliminated.

High-Resolution Timers

The *high-resolution timers* offer a nanosecond resolution and are used in applications for which the low-resolution timers are not enough, like multimedia processing. Initially, Linux kernel programmers wanted to replace the low-resolution timers with the high-resolution timers, but it proved to be a difficult task, so they decided to incorporate the new timers into the existing code. These timers are available if the kernel is compiled with their support enabled and the hardware provides at least two types of clocks that can be used by them. If the second requirement is not met, then the API of high-resolution timers will be available, but the timers will offer the same functionality as low-resolution timers. These two types of clocks, expected by the implementation of high-resolution timers, are monotonic and real-time clocks. Both offer a nanosecond resolution, but the first one is always incremented in specific moments of time and the second can be in some cases decremented, so the kernel has to compensate for these changes. In a multiprocessor environment, each CPU usually has one pair of such clocks.

High-Resolution Timers API

A high-resolution timer is represented by a structure of the `struct hrtimer` type. Since 2017, when activated, the timer is added to a red-black tree or to a queue[5]. When the hardware clock associated with high-resolution timers generates an interrupt, the functions of expired timers in the red-black tree are performed by the ISR. Then, the interrupt handler checks, if the first clock in the queue should expire. If so, it raises a softirq, that performs the timer function and also checks the other timers in the queue for expiration. Aside from members that allow the structure of the `struct hrtimer` type to be inserted into a queue or a red-black tree, it also contains the `expires` filed, specifying the length of the time period, after which the timer expires. The unit of this time is a nanosecond. Another field of this structure is the `function` member, which stores the address of a timer function that implements the operations to be performed by the timer.

[5] In that year Anna-Maria Gleixner introduced a patch to the kernel that changed how the high-resolution timers work.

High-Resolution Timers API

The prototype of this function is as follows:

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

The enumeration, that defines the type of values returned by the function, has two elements: `HRTIMER_NORESTART` indicating that the timer won't be automatically renewed and `HRTIMER_RESTART` meaning that the timer will be cyclic. In the latter case, the timer function has to modify the `expires` field of the `struct hrtimer` structure that points to the function. That's why the address of the structure is passed to the function as its argument. Using the `hrtimer_forward()` function is the only safe way of modifying that field. One of the members of the `stuct hrtimer` structure also stores the current state of the timer, expressed by one of the following constants:

`HRTIMER_STATE_INACTIVE` the timer is inactive,

`HRTIMER_STATE_ENQUEUED` the timer is active and awaits to be performed.

High-Resolution Timers API

Functions that handle the high-resolution timers are similar to those that handle the low-resolution timers. The `hrtimer_init()` function initializes the `struct hrtimer` structure. One of its arguments specifies if the value in the `expires` member expresses absolute time (in this case, since the computer has been turned on) or relative time (in this case, since the last activation of the timer). The `hrtimer_start()` function activates the timer. There are two functions (`hrtimer_cancel()` and `hrtimer_try_to_cancel()`) responsible for canceling the timer. Both return 0 if the timer was inactive or 1 if it was active. The latter also returns -1 if the timer has been already performing its function. The timer can be reactivated with the use of the `hrtimer_restart()` function. Often the timers are used for waking up a thread which is sleeping in a waiting queue. The kernel provides a structure of the `struct hrtimer_sleeper` type, that links the timer and the descriptor of the thread and simplifies handling of such a case. For more detailed description of the timers API please refer to the 7^{th} laboratory instruction.

Delaying Execution

The simplest way of delaying an execution of the code in the kernel-space is to use busy-waiting. In Linux kernel it can be implemented by reading in a loop the `jiffies` variable and comparing its current value with the desired number of system timer interrupts. To compare these values the `time_before` macro can be used, for example. The `jiffies` variable is prepared for such an usage. It is declared with the use of the `volatile` keyword, to prevent the compiler from storing its value in a register. This would cause the loop to always read the same value of this variable. The keyword assures that the value is read directly from the variable, or in other words directly from the memory. Since busy-waiting is an anti-pattern it is recommended to invoke the `condition_reached()` function inside the loop, which results in rescheduling the processes. If the delay should be short, one of the following functions can be used: `udelay()`, `mdelay()`, or `ndelay()`.

Delaying Execution

The first one delays the execution for a specific number of microseconds, the second one for a given number of milliseconds and the last one for a specific number of nanoseconds. All these functions apply busy-waiting. The `udelay()` functions performs a loop which number of iterations is determined by its argument and the number of so-called *BogoMIPS*. The last value is set at the boot time and specifies how many times in a given period of time the CPU can perform some instructions. Starting from the 3.6 version of the kernel, for the computer systems based on the ARM CPUs, the `udelay()` function uses a special hardware timer available in these platforms. In case of other computer systems its implementation is unchanged. The `mdelay()` function just invokes the `udelay()` function.

Delaying Execution

If the delay should be long, then using busy-waiting is not a good idea. Instead the schedule_timeout() function can be applied, that puts a thread to sleep and wakes it up after a given period of time. Before the function is invoked the state of the thread should be changed to TASK_UNINTERRUPTIBLE or TASK_INTERRUPTIBLE or TASK_KILLABLE. To simplify the usage of this function the kernel programmers defined three other functions:

schedule_timeout_killable() sets the TASK_KILLABLE state of the thread, and calls the schedule_timeout() function,

schedule_timeout_interruptible() sets the TASK_INTERRUPTIBLE state of the thread, and calls the schedule_timeout() function,

schedule_timeout_uninterruptible() sets the thread state to the TASK_UNINTERRUPTIBLE, and calls the schedule_timeout() function.

Delaying Execution

Finally, the `struct hrtimer_sleeper` structure and high-resolution timers can be used for delaying the execution of a thread.

Questions

?

The End

Thank You for Your attention!