Operating Systems 2

Time Management and Timers in Linux

Arkadiusz Chrobot

Department of Computer Science

April 26, 2020

Outline

Introduction

The information about time is important to the kernel and to user-space applications. Some of the kernel function are activated at specified moments of time, so the kernel measures *relative time* (from one such a moment to another). User applications, such as databases management systems, needs information about *absolute time*, also called *wall time*, *real-world time* or *wall-clock time*. The kernel also addresses this requirement. In this lecture the subsystems of kernel responsible for time-keeping are described. Also the timers are discussed.

Time Management

The main element that allows the kernel to measure the time passing 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 work is usually based on electric impulses generated by a crystal oscillator. For example the in the computers based on x86 processor family there are several types of such devices, the oldest one called PIT (an abbreviation for: *Programmable Interval Timer*), the more modern HPET (*High Precision Event Timer*), the APIC timer and *time stamp counter* (TSC). Similar devices are present in other hardware platform. Each of them offers a different accuracy and requires a different handling. Starting from the 2.6 series of kernels Linux programmers added an abstraction layer that hides most of those differences and unifies the handling of those devices.

Virtual Clocks

The abstraction layer used the hardware timers to create a three types of virtual clocks:

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

clock event devices can generate a signal (event) at a given time in the future,

tick devices generate repeatedly an event that signals passing of a period of time.

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

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

Dynamic Ticks

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

The `jiffies` Variable

The number of interrupts that has been generated since the computer system started is stored in the 64-bits `jiffies` variable. The period of time when the computer is working (so-called *uptime*) can be calculated as `jiffies/HZ`. In the older kernel versions the `jiffies` variable was 32-bits wide for the 32-bit CPUs and 64-bit wide for the 64-bit CPUs. Rising the frequency of system timer caused a problem with `jiffies` variable for the 32-bit CPUs of the x86 family. For the value of `HZ` equal 100, the `jiffies` overflows each 497 days. The period was reduced to 49.7 days, when the value of `HZ` had been changed to 1000. To solve the problem, kernel programmers added a 64-bit variable named `jiffies_64`. In the 32-bit computers the `jiffies` variable overlaps the 4 less significant bytes of the `jiffies_64` variable, while in 64-bit computers those names refer to the same variable. In the 32-bit computers the `jiffies_64` variable should be read with the help of `get_jiffies_64()` function, which assures indivisible access to the variable.

The `jiffies` Variable

Kernel programmers have defined four macros that make the time measurement with the values of `jiffies` variable much easier. They take into account the overflows of that variable. Each of those macros takes two arguments, which are the values of the `jiffies` variable. The `time_after` macro evaluates to a nonzero value (true) if the moment referred by its first argument happen after the moment referred by its second argument. The `time_before` macro does the same, if the first argument refers to a moment that happen before the moment referred by the second argument. Two others macros (`time_after_eq` and `time_before_eq`) behave similarly, but the also evaluate to a nonzero value when both argument reference 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 base on x86 CPU family or the DEC Alpha CPUs. For such computer systems the kernel programmers defined a separate constant named USER_HZ, which is set to the value of the system timer frequency expected by the user-space applications. To convert the number of timer interrupts to a value that corresponds to the value of the USER_HZ constant, the `jiffies_to_clock_t()` or the `jiffies_64_to_clock_t()` function is used.

The Real-Time Clock

The Linux kernel tracks also the passing of 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 process and then only updated by the timer interrupt handler. In older kernels the date and time were stored in a structure called `xtime`, which had two members one for storing the number of seconds that have passed since the 1^{st} of January 1970 (the starting point of so-called Unix Epoch) and another for storing the number of nanoseconds that have passed since the start of the current second[3]. Nowadays kernels store the information in a two separate variables. One is of the `struct tk_read_base` type and stores the number of nanoseconds, and the second one is of the `struct timekeeper` type and stores the number of seconds.

[2] A notable exception are the Raspberry Pi computers.

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

The Real-Time Clock

The information about current date and time is provided to user-space software via the `gettimeofday()` system call.

The Timer Interrupt Servicing Routine

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

- handle the system timer,
- periodically update the wall-clock,
- invoke the `tick_periodic()` function, which is an implementation of the hardware-independent part.

The second one does the following:

- update the `jiffies_64` variable,
- update the resource usages for the current process,
- activate the expired timers,
- update the variables storing the wall-time,
- calculate the average load of system.

Low-Resolution Timers

The kernel provides *timers* (also known as *kernel timers* or *dynamic timers*) which are a way for deferring some work to be done for a given amount of time. The work is performed in the interrupt context. In other words timers are another implementations of bottom haves. There are two types of such timers. The first one are the *low-resolution* of *timer wheel* timers. The resolution of those timers is of the length of the system timer period. The average accuracy is of half of that time. They are not suitable for real-time or multimedia applications, but are good enough for any other usages. The low-resolution timer are also not cyclic, which means that when they expired they are not automatically renewed. A single low-resolution timer is represented by a `stuct timer_list` type variable. Such structure has to be initialized with the use of the `init_timer` macro. After that the values of the following members of the structure have to be set: `expires`, `function` and `data`.

Low-Resolution Timers

The first one defines the amount of time after which the timer expires. The value is expressed in periods of the system timer. The second one stores the address of a function which implements the work that needs to be done when the timer expires. The last one, of `unsigned long` type, stores the data for the function. The prototype of the function is as follows:

```
void timer_function(unsigned long data)
```

The name of the function can be different that the one used in the prototype. Because this subroutine can be associated with more than one timer, the argument passed by the `data` parameter allows it to recognize for which timer it is invoked. The members of the `struct timer_list` variable may be initialized wit the help of the `setup_timer` macro. Since the kernel version 4.15 the `init_timer` and `setup_timer` macros has been replaced by a single function named `timer_setup()`. 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 the work activated by the timer should use appropriate synchronization mechanisms to protect the shared resources it accesses. The low-resolution timers are stored in a linked list, which is unsorted, but divided into five parts. The timers are added to one of the group, depending on their expiration time, and then they are moved from one group to another until they expire. The timer functions are invoked by a softirq handler when the timer expires and after handling the system timer interrupt.

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 those new timers into the existing code. The timers are available if the kernel was compiled with their support enabled and the hardware provides at least two types of clock that can be used by them. If the second requirement is not fulfilled than the API of high-resolution timers will be available but the timers will offer the same functionality as low-resolution timers. Those two types of clock expected by the implementation of high-resolution timers are monotonic and real-time clock. Both offer a nanosecond resolution, but the first one is always incremented in a specified moments of time and the second can be in some cases decremented, so the kernel have to compensate those changes. In 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. When activated the timer is added simultaneously to a red-black tree and a list. The list is traversed sequentially and is sorted according to the timers expiry time with the help of the red-black tree. When the hardware clock associated with high-resolution timers generates an interrupt, the functions of expired timers are performed by a bottom half of the interrupt handling code, which is a softirq. Aside from the members that allows the structure of the `struct hrtime` type to be inserted into a linked list and a red-black tree the structure also contains the `expires` filed, which stores the length of the time period after which the timer expires. The unit of this time is a nanosecond. Another filed of this structure is the `function` which stores the address of a timer function that implements the work that has to be done by the timer.

High-Resolution Timers API

The prototype of this function is as follows:

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

The name of the function doesn't have to be the same as in the prototype. The enumerated type that defines the values returned by the function has two elements: the `HRTIMER_NORESTART` indicates that the timer won't be automatically renewed and the `HRTIMER_RESTART` that means 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. Thats why the structure is passed to the function by a pointer. The field can be safely modified only with the help of `hrtimer_forward()` function. One of the members of the `stuct hrtimer` structure also stores the current state of the timer. As a value it can take 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 ones that handle 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 (measured since the computer has been switch on) or relative time (measured since the 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 and 0 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 was sleeps in a waiting queue. The kernel provides a structure of the `struct hrtimer_sleeper`, that links the timer and the descriptor of the thread and simplifies handling of such a case. For more detailed description of 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-looping. 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 those values the `time_before` macro can be used, for example. The `jiffies` variable is prepared for such a 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()`, and `ndelay()`.

Delaying Execution

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

Delaying Execution

If the delay should be long, then using the busy-waiting is not a good idea. Instead the `schedule_timeout()` function can be applied, that puts the 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()` set the `TASK_INTERRUPTIBLE` state of the thread and calls the `schedule_timeout()` function,

`schedule_timeout_uninterruptible()` set 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!