Chapter 6. Timing Measurements

We can distinguish two **main kinds of timing measurement** that must be performed by the Linux kernel:

- Keeping the **current time and date** so they can be returned to user programs through the *time( )*, *ftime( )*, and *gettimeofday( )* APIs and used by the kernel itself as timestamps for files and network packets

- Maintaining **timers mechanisms** that are able to notify the kernel or a user program that a certain interval of time has elapsed

6.1. Clock and Timer Circuits

On the 80x86 architecture, the kernel must explicitly interact with **several kinds of clocks and timer circuits**.

•The **clock circuits** are used both to keep track of the current time of day and to make precise time measurements.

•The **timer circuits** are programmed by the kernel, so that they issue interrupts at a fixed, predefined frequency; such periodic interrupts are crucial for implementing the software timers used by the kernel and the user programs.

We'll now briefly describe the clock and hardware circuits that can be found in **IBM-compatible PCs.**

6.1.1. Real Time Clock (RTC)

• All PCs include a clock called Real Time Clock (RTC), which is **independent of the CPU** and all other chips.

• The RTC continues to tick even when the PC is switched off, because it is **energized by a small battery**. The CMOS RAM and RTC are integrated in a single chip (the Motorola 146818 or an equivalent).

• The RTC is capable of issuing **periodic interrupts on IRQ 8** at frequencies ranging between **2 Hz and 8,192 Hz**.

• It can also be programmed to activate the IRQ 8 line when the RTC reaches a specific value, thus **working as an alarm clock**.

• **Linux uses the RTC only to derive the time and date**; however, it allows processes to program the RTC by acting on the /dev/rtc device file.

• **The kernel accesses the RTC through the 0x70 and 0x71 I/O ports**. The system administrator can read and write the RTC by executing the clock Unix system program that acts directly on these two I/O ports.

6.1.2. Time Stamp Counter (TSC)

All 80x86 microprocessors include a **CLK input pin**, which receives the clock signal of an external oscillator.

Starting with the Pentium, 80x86 microprocessors support a **counter that is increased at each clock signal**.

The counter is accessible through the **64-bit Time Stamp Counter** (TSC) register, which can be read by means of the **rdtsc** assembly language instruction.

When using this register, **the kernel has to take into consideration the frequency of the clock signal**:if, for instance, the clock ticks at 1 GHz, the Time Stamp Counter is increased once every nanosecond.

Linux may take advantage of this register to get much more **accurate time measurements**.

The task of figuring out the actual frequency of a CPU is accomplished during the system's boot. The calibrate_tsc( ) function computes the frequency by counting the number of clock signals that occur in a time interval of approximately 5 milliseconds.

6.1.3. Programmable Interval Timer (PIT)

this device issues a special interrupt called **timer interrupt**, which notifies the kernel that one more time interval has elapsed.

the PIT goes on issuing interrupts forever at some fixed frequency established by the kernel.

Each IBM-compatible PC includes at least one PIT, which is usually implemented by an **8254 CMOS** chip using the 0x40-0x43 I/O ports.

The PIT is also used to drive an audio amplifier connected to the computer's internal speaker.

Tick !

As we'll see in detail in the next paragraphs, Linux programs the PIT of IBM-compatible PCs to issue timer interrupts on the IRQ 0 at a (roughly) 1000-Hz frequency that is, once every 1 millisecond.

This time interval is called a **tick**, and its length in nanoseconds is stored in the tick_nsec variable.

On a PC, tick_nsec is initialized to 999,848 nanoseconds (yielding a clock signal frequency of about 1000.15 Hz), but **its value may be automatically adjusted by the kernel if the computer is synchronized with an external clock.**

The ticks beat time for all activities in the system; in some sense, they are like the ticks sounded by a metronome while a musician is rehearsing.

HZ

Generally speaking, **shorter ticks result in higher resolution timers**, which help with smoother multimedia playback and faster response time when performing synchronous I/O multiplexing (poll( ) and select( ) system calls).

This is a trade-off however: **shorter ticks require the CPU to spend a larger fraction of its time in Kernel Mode** that is, a smaller fraction of time in User Mode. As a consequence, user programs run slower.

A few macros in the Linux code yield some constants that determine the frequency of timer interrupts. These are discussed in the following list.

- **HZ** yields the approximate number of timer interrupts per second that is, their frequency. This value is set to 1000 for IBM Pcs.
- **CLOCK_TICK_RATE** yields the value 1,193,182, which is the 8254 chip's internal oscillator frequency.
- **LATCH** yields the ratio between CLOCK_TICK_RATE and HZ, rounded to the nearest integer. It is used to program the PIT.

6.1.4. CPU Local Timer

The local APIC present in recent 80 x 86 microprocessors provides yet another time-measuring device: the CPU local timer .

The CPU local timer is a device similar to the Programmable Interval Timer just described that can issue one-shot or periodic interrupts. There are, however, a few **differences**:

The APIC's timer counter is 32 bits long, while the PIT's timer counter is 16 bits long; therefore, the local timer can be programmed to issue interrupts at very low frequencies (the counter stores the number of ticks that must elapse before the interrupt is issued).

The local APIC timer sends an interrupt only to its processor, while the PIT raises a global interrupt, which may be handled by any CPU in the system.

The APIC's timer is based on the bus clock signal (or the APIC bus signal, in older machines). It can be programmed in such a way to decrease the timer counter every 1, 2, 4, 8, 16, 32, 64, or 128 bus clock signals. Conversely, the PIT, which makes use of its own clock signals, can be programmed in a more flexible way.

6.1.5 High Precision Event Timer

The High Precision Event Timer (HPET) is a new timer chip developed jointly by **Intel and Microsoft**.

Basically, the chip includes **up to eight 32-bit or 64-bit independent counters** .

Each counter is driven by its own clock signal, whose frequency must be at least 10 MHz; therefore, the counter is increased at least once in 100 nanoseconds.

Any counter is associated with at most 32 timers, each of which is composed by a comparator and a match register. The comparator is a circuit that checks the value in the counter against the value in the match register, and raises a hardware interrupt if a match is found. **Some of the timers can be enabled to generate a periodic interrupt.**

The next generation of motherboards will likely sport both the HPET and the 8254 PIT; in some future time, however, the HPET is expected to completely replace the PIT.

6.1.6. ACPI Power Management Timer

The ACPI Power Management Timer (or ACPI PMT) is yet another clock device included in almost all **ACPI-based motherboards**.

Its clock signal has a **fixed frequency of roughly 3.58 MHz**. The device is actually a **simple counter** increased at each clock tick; to read the current value of the counter, the kernel accesses an I/O port whose address is determined by the BIOS during the initialization phase.

The ACPI Power Management Timer is preferable to the TSC if the operating system or the BIOS may dynamically lower the frequency or voltage of the CPU to save battery power. When this happens, the frequency of the TSC changes, thus causing time warps and others unpleasant effects, while the frequency of the ACPI PMT does not. **On the other hand, the high-frequency of the TSC counter is quite handy for measuring very small time intervals.**

However, if an HPET device is present, it should always be preferred to the other circuits because of its richer architecture. Table 6-2 later in this chapter illustrates how Linux takes advantage of the available timing circuits.

6.2. The Linux Timekeeping Architecture

Linux must carry on several time-related activities. For instance, the kernel periodically:

•Updates the **time elapsed since system startup.**

•Updates the **time and date**.

•Determines, for every CPU, **how long the current process has been running**, and preempts it if it has exceeded the time allocated to it.

•Updates **resource usage statistics**.

•Checks whether the interval of time associated with each **software timer** has elapsed.

Uniprocessor and multiprocessor time-keeping architecture

Linux's timekeeping architecture is the **set of kernel data structures and functions related to the flow of time.**

Actually, 80 x 86-based **multiprocessor machines** have a timekeeping architecture that is slightly **different** from the timekeeping architecture of uniprocessor machines:

•In a **uniprocessor** system, **all time-keeping activities** are triggered by interrupts raised by the **global timer** (either the Programmable Interval Timer or the High Precision Event Timer).

•In a **multiprocessor** system, **all general activities** (such as handling of software timers) are triggered by the interrupts raised by the global timer, while **CPU-specific activities** (such as monitoring the execution time of the currently running process) are triggered by the interrupts raised by the **local APIC timer**.

6.2.1. Data Structures of the Timekeeping Architecture

The kernel uses two basic timekeeping functions:

•one to keep the **current time up-to-date** and

•another to count the number of **nanoseconds that have elapsed within the current second.**

There are **different ways to get the last value**. Some methods are more precise and are available if the CPU has a Time Stamp Counter or a HPET; a less-precise method is used in the opposite case.

6.2.1.1. The timer object

In order to handle the possible timer sources in a uniform way, the kernel makes use of a **"timer object"**, which is a descriptor of type timer_opts consisting of the timer name and of **four standard methods** shown in Table 6-1.

Table 6-1. The fields of the timer_opts data structure

Field name	Description
name	A string identifying the timer source
mark_offset	Records the exact time of the last tick; it is invoked by the timer interrupt handler
get_offset	Returns the time elapsed since the last tick
delay	Waits for a given number of "loops"

Time interpolation

The **mark_offset** method is invoked by the timer interrupt handler, and records in a suitable data structure the exact time at which the tick occurred.

Using the saved value, the **get_offset** method computes the time in microseconds elapsed since the last timer interrupt (tick).

Thanks to these two methods, Linux timekeeping architecture achieves a **sub-tick resolution**, that is, the kernel is able to determine the current time with a precision much higher than the tick duration. This operation is **called time interpolation .**

select_time()

The **cur_timer** variable stores the address of the timer object corresponding to the **"best" timer source available in the system**.

Initially, cur_timer points to timer_none, which is the object corresponding to a dummy timer source used when the kernel is being initialized.

During kernel initialization, the **select_timer( )** function sets cur_timer to the address of the appropriate timer object.

Table 6-2 shows the most common timer objects used in the 80x86 architecture, **in order of preference**.

•As you see, select_timer( ) selects the **HPET**, if available; otherwise, it selects the **ACPI Power Management Timer** , if available, or the **TSC**.

•As the last resort, select_timer( ) selects the always-present **PIT**.

•The "**Time interpolation**" column lists the timer sources used by the mark_offset and get_offset methods of the timer object;

•the "**Delay**" column lists the timer sources used by the delay method.

Table 6-2. Typical timer objects of the 80x86 architecture, in order of preference

Timer object name	Description	Time interpolation	Delay
timer_hpet	HPET	HPET	HPET
timer_pmtmr	ACPI PMT	ACPI PMT	TSC
timer_tsc	TSC	TSC	TSC
timer_pit	PIT	PIT	Tight loop
timer_none	dummy	none	Tight loop

Notice that local APIC timers do not have a corresponding timer object. The reason is that **local APIC timers are used only to generate periodic interrupts and are never used to achieve sub-tick resolution.**

6.2.1.2. The jiffies variable

The jiffies variable is a counter that stores the number of **elapsed ticks since the system was started**. It is increased by one when a timer interrupt occurs, that is, on every tick.

In the 80 x 86 architecture, **jiffies is a 32-bit variable**, therefore it wraps around in approximately **50 days**, a relatively short time interval for a Linux server.

In a few cases, however, the kernel needs the real number of system ticks elapsed since the system boot, regardless of the overflows of jiffies. Therefore, in the 80 x 86 architecture the jiffies variable is equated by the linker to the 32 less significant bits of a 64-bit counter called **jiffies_64**.

With a tick of 1 millisecond, the **jiffies_64** variable wraps around in **several hundreds of millions of years**, thus we can safely assume that it never overflows.

You might wonder why jiffies has not been directly declared as a 64-bit unsigned long long integer on the 80 x 86 architecture. The answer is that accesses to 64-bit variables in 32-bit architectures cannot be done atomically. Therefore, every read operation on the whole 64 bits requires some **synchronization technique** to ensure that the counter is not updated while the two 32-bit half-counters are read; as a consequence, every **64-bit read operation is significantly slower than a 32-bit read operation.**

The get_jiffies_64( ) function reads the value of jiffies_64 and returns its value:

```
unsigned long long get_jiffies_64(void)
{
    unsigned long seq;
    unsigned long long ret;
    do {
        seq = read_seqbegin(&xtime_lock);
        ret = jiffies_64;
    } while (read_seqretry(&xtime_lock, seq));
    return ret;
}
```

The 64-bit read operation is protected by the xtime_lock seqlock: the function keeps reading the jiffies_64 variable until it knows for sure that it has not been concurrently updated by another kernel control path.

Conversely, the critical region increasing the jiffies_64 variable must be protected by means of write_seqlock(&xtime_lock) and write_sequnlock(&xtime_lock).

Notice that the ++jiffies_64 instruction also increases the 32-bit jiffies variable, because the latter corresponds to the lower half of jiffies_64.

6.2.1.3. The xtime variable

The xtime variable stores the **current time and date**; it is a structure of type timespec having two fields:

tv_sec
Stores the number of seconds that have elapsed since midnight of January 1, 1970 (UTC)

tv_nsec
Stores the number of nanoseconds that have elapsed within the last second (its value ranges between 0 and 999,999,999)

The xtime variable is usually **updated once in a tick**, that is, roughly 1000 times per second.

user programs get the current time and date from the xtime variable.
The kernel also often refers to it, for instance, when updating inode timestamps.

The **xtime_lock seqlock** avoids the race conditions that could occur due to concurrent accesses to the xtime variable. Remember that xtime_lock also protects the jiffies_64 variable; in general, this seqlock is used to define several critical regions of the timekeeping architecture.

6.2.2. Timekeeping Architecture in Uniprocessor Systems

In a uniprocessor system, **all time-related activities are triggered by the interrupts raised by the Programmable Interval Timer on IRQ line 0**.

As usual, in Linux, some of these activities are executed as soon as possible right after the interrupt is raised, while the remaining activities are carried on by **deferrable functions**

6.2.2.1. Initialization phase

During kernel initialization, the **time_init( ) function** is invoked to set up the timekeeping architecture. It usually performs the following operations:

1. Initializes the xtime variable. **The number of seconds elapsed since the midnight of January 1, 1970 is read from the Real Time Clock** by means of the get_cmos_time( ) function. The tv_nsec field of xtime is set, so that the forthcoming overflow of the jiffies variable will coincide with an increment of the tv_sec field, that is, it will fall on a second boundary.

2. Initializes the **wall_to_monotonic** variable. This variable is of the same type timespec as xtime, and it essentially stores the number of seconds and nanoseconds to be added to xtime in order to get a monotonic (ever increasing) flow of time.

In fact, both leap seconds and synchronization with external clocks might suddenly change the tv_sec and tv_nsec fields of xtime so that they are no longer monotonically increased.

As we'll see in the later section "System Calls for POSIX Timers," sometimes the kernel needs a truly monotonic time source.

initialization...

3. If the kernel supports HPET, it invokes the hpet_enable( ) function to determine whether the ACPI firmware has probed the chip and mapped its registers in the memory address space. In the affirmative case, hpet_enable( ) programs the first timer of the HPET chip so that it **raises the IRQ 0 interrupt 1000 times per second**. **Otherwise, if the HPET chip is not available, the kernel will use the PIT**: the chip has already been programmed by the init_IRQ( ) function to raise 1000 timer interrupts per second, as described in the earlier section "Programmable Interval Timer (PIT)."

4. Invokes **select_timer( )** to select the best timer source available in the system, and sets the **cur_timer** variable to the address of the corresponding timer object.

5. Invokes **setup_irq(0,&irq0)** to set up the interrupt gate corresponding to IRQ0, the line associated with the system timer interrupt source (PIT or HPET).The irq0 variable is statically defined as:

```
struct irqaction irq0 = { timer_interrupt, SA_INTERRUPT, 0,
                          "timer", NULL, NULL };
```

From now on, the **timer_interrupt( ) function will be invoked once every tick** with interrupts disabled, because the status field of IRQ 0's main descriptor has the SA_INTERRUPT flag set.

6.2.2.2. The timer interrupt handler

The **timer_interrupt( )** function is the interrupt service routine **(ISR) of the PIT or of the HPET**; it performs the following steps:

1. Protects the time-related kernel variables by issuing a write_seqlock() on the xtime_lock seqlock.

2. Executes the **mark_offset** method of the cur_timer timer object. As explained in the earlier section "Data Structures of the Timekeeping Architecture," there are **four possible cases:**

Timer interrupt handler

3. Invokes the **do_timer_interrupt( )** function, which in turn performs the following actions:

a. Increases by one the value of jiffies_64. Notice that this can be done safely, because the kernel control path still holds the xtime_lock seqlock for writing.

b. Invokes the update_times( ) function to update the system date and time and to compute the current system load; these activities are discussed later.

c. Invokes the update_process_times( ) function to perform several time-related accounting operations for the local CPU.

d. Invokes the profile_tick( ) function.

e. If the system clock is synchronized with an external clock (an adjtimex( ) system call has been previously issued), invokes the set_rtc_mmss( ) function once every 660 seconds (every 11 minutes) to adjust the Real Time Clock. This feature helps systems on a network synchronize their clocks.

6.2.3. Timekeeping Architecture in Multiprocessor Systems

Multiprocessor systems can rely on two different sources of timer interrupts:
•those raised by the Programmable Interval Timer or the High Precision Event Timer,
•and those raised by the **CPU local timers**.

In Linux 2.6, global timer interrupts raised by the PIT or the HPETsignal activities not related to a specific CPU, such as handling of software timers and keeping the system time up-to-date.

Conversely, a CPU local timer interrupt signals timekeeping activities related to the local CPU, such as monitoring how long the current process has been running and updating the resource usage statistics.

6.3. Updating the Time and Date

User programs get the current time and date from the xtime variable. The kernel must periodically update this variable, so that its value is always reasonably accurate.

The update_times( ) function, which is invoked by the global timer interrupt handler, updates the value of the xtime variable as follows:

```
void update_times(void)
{
    unsigned long ticks;
    ticks = jiffies - wall_jiffies;
    if (ticks) {
        wall_jiffies += ticks;
        update_wall_time(ticks);
    }
    calc_load(ticks);
}
```

The **wall_jiffies** variable stores the time of the last update of the xtime variable. Observe that the value of wall_jiffies can be smaller than jiffies-1, since a few **timer interrupts can be lost**, for instance when interrupts remain disabled for a long period of time; The check for lost timer interrupts is done in the mark_offset method of cur_timer;

The **update_wall_time( )** function invokes the update_wall_time_one_tick( ) function ticks consecutive times; normally, each invocation adds 1,000,000 to the xtime.tv_nsec field. If the value of xtime.tv_nsec becomes greater than 999,999,999, the update_wall_time( ) function also updates the tv_sec field of xtime. If an **adjtimex( )** system call has been issued, the function might tune the value 1,000,000 slightly so the **clock speeds up or slows down a little.**

6.4. Updating System Statistics

The kernel, among the other time-related duties, must periodically collect various data used for:

- Checking the CPU resource limit of the running processes
- Updating statistics about the local CPU workload
- Computing the average system load
- Profiling the kernel code

6.5. Software Timers and Delay Functions

A **timer** is a software facility that allows functions to be invoked at some future moment, after a given time interval has elapsed;

a **time-out** denotes a moment at which the time interval associated with a timer has elapsed.

Timers are widely used both by the kernel and by processes. Most device drivers use timers to detect anomalous conditions

floppy disk drivers, for instance, use timers to switch off the device motor after the floppy has not been accessed for a while, and parallel printer drivers use them to detect erroneous printer conditions.

Timers are also used quite often by programmers to force the execution of specific functions at some future time

timers

Implementing a timer is relatively easy. Each timer contains a field that indicates how far in the future the timer should expire. This field is initially calculated by adding the right number of ticks to the current value of jiffies. The field does not change.

Every time the kernel checks a timer, **it compares the expiration field to the value of jiffies at the current moment**, and the timer expires when jiffies is greater than or equal to the stored value.

Linux considers two types of timers called dynamic timers and interval timers . The first type is used by the kernel, while interval timers may be created by processes in User Mode.

Not real time !

One word of caution about Linux timers: since checking for timer functions is always done by deferrable functions that may be executed a long time after they have been activated, the kernel cannot ensure that timer functions will start right at their expiration times.

It can only ensure that they are executed either at the proper time or after with a delay of up to a few hundreds of milliseconds. For this reason, timers are not appropriate for real-time applications in which expiration times must be strictly enforced.

Besides software timers , the kernel also makes use of delay functions , which execute a tight instruction loop until a given time interval elapses. We will discuss them in the later section "Delay Functions."

6.5.1. Dynamic Timers

Dynamic timers may be dynamically created and destroyed. No limit is placed on the number of currently active dynamic timers.A dynamic timer is stored in the following timer_list structure:

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;
    spinlock_t lock;
    unsigned long magic;
    void (*function)(unsigned long);
    unsigned long data;
    tvec_base_t *base;
};
```

The **function field** contains the address of the function to be executed when the timer expires.

The **data field** specifies a parameter to be passed to this timer function. Thanks to the data field, it is possible to define a single general-purpose function that handles the time-outs of several device drivers; the data field could store the device ID or other meaningful data that could be used by the function to differentiate the device.

The **expires field** specifies when the timer expires; the time is expressed as the number of ticks that have elapsed since the system started up. All timers that have an expires value smaller than or equal to the value of jiffies are considered to be expired or decayed.

The entry field is used to insert the software timer into one of the doubly linked circular lists that group together the timers according to the value of their expires field. The algorithm that uses these lists is described later in this chapter.

Timer softirq

Despite the clever data structures, handling software timers is a time-consuming activity that should not be performed by the timer interrupt handler.

In Linux 2.6 this activity is carried on by a deferrable function, namely the TIMER_SOFTIRQ softirq.

The run_timer_softirq( ) function is the deferrable function associated with the TIMER_SOFTIRQ softirq.

6.5.3. Delay Functions

•Software timers are useless when the kernel must wait for a short time intervallet's say, **less than a few milliseconds**. For instance, often a device driver has to wait for a predefined number of microseconds until the hardware completes some operation.

•Because a dynamic timer has a significant setup overhead and a rather large minimum wait time (1 millisecond), the device driver cannot conveniently use it.

•In these cases, the kernel makes use of the udelay( ) and ndelay( ) functions: the former receives as its parameter a time interval in microseconds and returns after the specified delay has elapsed; the latter is similar, but the argument specifies the delay in nanoseconds.

Essentially, the two functions are defined as follows:

```
void udelay(unsigned long usecs)
    {
        unsigned long loops;
        loops = (usecs*HZ*current_cpu_data.loops_per_jiffy)/1000000;
        cur_timer->delay(loops);
    }
```

```
    void ndelay(unsigned long nsecs)
    {
        unsigned long loops;
        loops = (nsecs*HZ*current_cpu_data.loops_per_jiffy)/1000000000;
        cur_timer->delay(loops);
    }
```