

Advanced Operating Systems and Virtualization

[6] Time Management

DIAG

Department of Computer,
Control and Management
Engineering "A. Ruberti",
Sapienza University of Rome

Outline

1. Introduction
2. Timekeeping Architecture
 1. Low-resolution Timers
 2. Generic Time Subsystem
3. Watchdogs

6.1

6. Time Management

Introduction

Time keeping

A computer would be useless if programs would not have the possibility to keep track of time passing. This fundamental facility is handled by the kernel in two ways:

1. keeping the current **time** and **date** so they can be returned to user programs (e.g. `time()`, `gettimeofday()`, ...)
2. maintaining **timers**, mechanisms that are able to notify the kernel or a user program (e.g. `setitimer()` and `alarm()`) that a certain interval of time has elapsed.

Time measurements are performed by hardware circuits based on oscillators.

Clock and Timer Circuits

In the 8086 architecture the CPU interacts with clock circuits that keep track of the time of day and timer circuits, programmed by the kernel so that they issue interrupts at a desired time. We have:

- **Real Time Clock (RTC)**, which is independent of all the other chips and keeps track of the time of day, ticking even if the PC is off. The RTC is capable of issuing periodic interrupts on IRQ8 and can be programmed and used as an alarm. Linux uses the RTC only for time and date (`/dev/rtc`).
- **Timestamp Counter (TSC)**, it is a counter incremented from an external oscillator connected at the CLK pin of the CPU, it can be read with the asm instruction `rdtsc`.
- **Programmable Interval Timer (PIT)**, its role is similar to an alarm clock of a microwave oven, when timer ends it does not ring a bell but it issues an interrupt. The PIT can be programmed also for ticking at periodic intervals, these ticks are as a beat time for all the kernel activities, as a metronome.
- **CPU Local Timer** (in the LAPIC) is similar to PIT but local to the processor.
- **High Precision Event Timer (HPET)**, provides a number of programmable timers .

6.2

6. Time Management

Timekeeping Architecture

Keeping time in the Linux kernel

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

- updates the time elapsed since the startup
- updates time and date
- determines, for every cpu, how long the current process has been running and preempts it if it exceeded its time slot
- update resource usage statistics
- checks whether the interval of time associated with each software timer has elapsed

The *Linux timekeeping architecture* is a set of data structures and functions related to the flow of time. There are some differences in time keeping between a multi and a single processor architecture:

- in a single-processor system all time-keeping activities are triggered by interrupts raised by a global timer
- in a multi-processor system general activities (e.g. software timers) are triggered by the global timer but CPU-specific activities (e.g. monitoring the execution time of the current process) are triggered by the local APIC timer

Kernel Timers

The entire kernel timing subsystem wraps around timers. Timers can be divided into:

- classic timers available since the first version of the kernel, their resolution is usually 4ms and they are called **low-resolution** timers or *timer wheel* timers
- **high-resolution** timers, they have a precision of nanoseconds, useful especially for media-oriented applications

Independently of the resolution, the kernel distinguishes between:

- **Time-outs** that represent events that are bound to happen after some time, e.g. expect a packet within the next 10 seconds. In general, resolution is not important for this kind of timers;
- **Timers** that are used for implementing temporal sequences, e.g. a sound card driver could issue data in small and periodic time intervals. These timers usually requires high resolution;

Kernel Timers

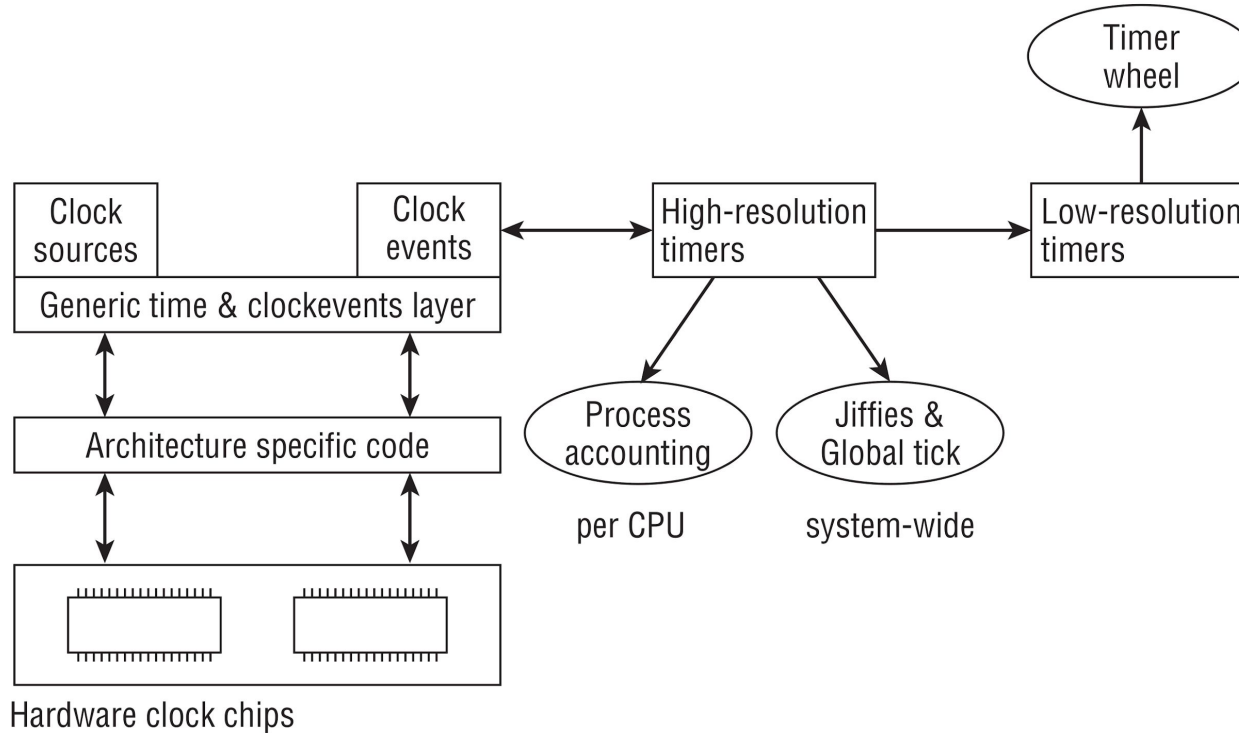


Figure 15-2: Overview of the components that build up the timing subsystem.

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

Clock Events & Ticks

The Clock Events are the foundation of periodic events. High Resolution Timers are based on the Clock Events abstraction, whereas the low-resolution mechanism can come or from a low-resolution clock device or from the high resolution subsystem. Two important tasks for which low-resolution timers assume the responsibility are:

- handling the global jiffies counter
- perform per-process accounting

Configuration Options

In general a periodic tick is enable for the whole life of the kernel but this would not allow the system to go in sleep mode (e.g. laptop), for this reason the kernel allows to configure a dynamic tick which does not require a periodic signal (*tickless* system).

On right the possible configurations.

High-res Dynamic ticks	High-res Periodic ticks
Low-res Dynamic ticks	Low-res Periodic ticks

Figure 15-3: Possible timekeeping configurations that arise because of high- and low-resolution timers and dynamic/periodic ticks.

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

6.2.1

6. Time Management

2. Timekeeping Architecture

Low-Resolution Timers

Low-Resolution Timers

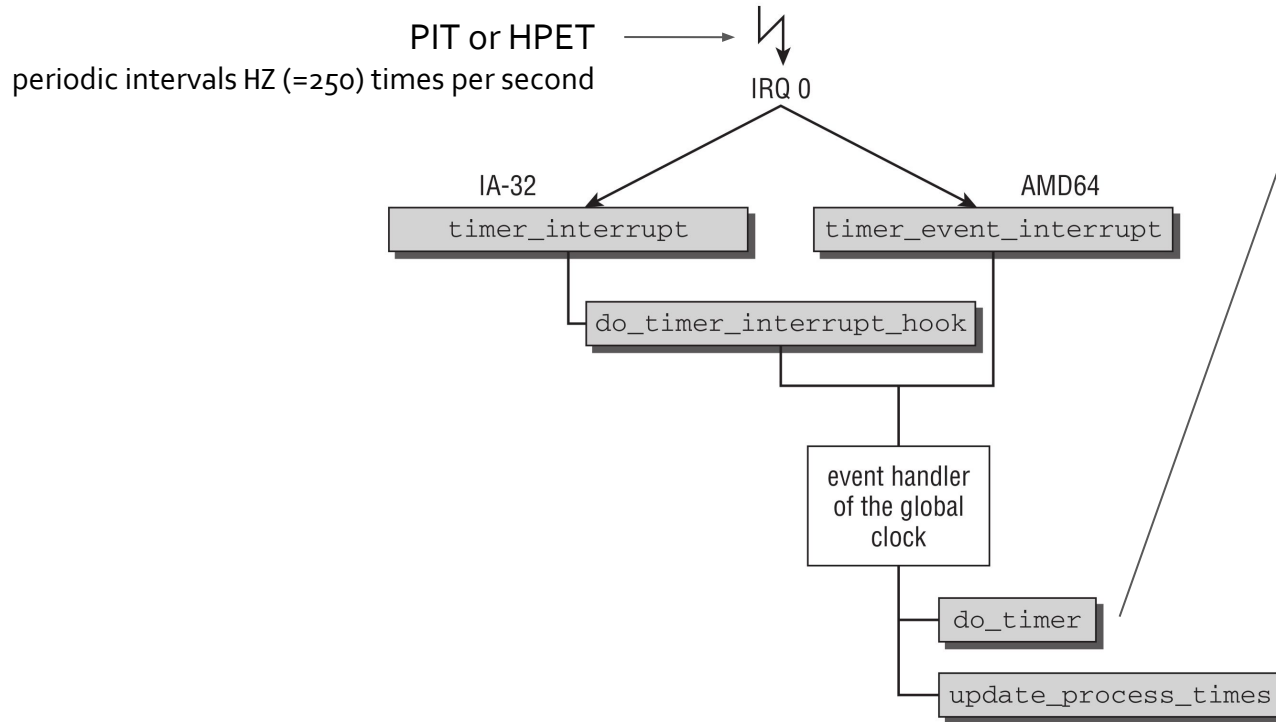


Figure 15-4: Overview of periodic low-resolution timer interrupts on IA-32 and AMD64 machines.

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

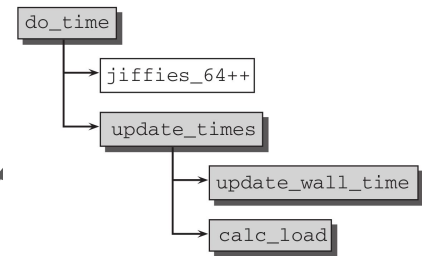


Figure 15-5: Code flow diagram for `do_time`.

System-wide, **global** tasks:
update the jiffies value,
handle process accounting
(statistics), if
multiprocessor **only one
CPU does this**

Performed by every CPU,
besides process
accounting **it activates
and expires all
low-resolution timers**
(the `TIMER_SOFTIRQ` is
raised) and trigger the
process scheduler

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 interrupts occurs.

In the 8086 the jiffies variable is of 32bit in size so it wraps in about 50 days, this because a long long (64bit) could not allow an atomic increment. This is true even today, with kernel 5.11.

```
74  /*  
75   * The 64-bit value is not atomic - you MUST NOT read it  
76   * without sampling the sequence number in jiffies_lock.  
77   * get_jiffies_64() will do this for you as appropriate.  
78   */  
79  extern u64 __cacheline_aligned_in_smp jiffies_64;  
80  extern unsigned long volatile __cacheline_aligned_in_smp __jiffy_arch_data jiffies;
```

jiffies and jiffies_64 matches in their less significant bits and therefore they must point to the same memory location or same register. For achieving this, the two variables are declared separately but the linker merges them, in 64bit machine they are the same but in 32bit jiffies is the upper (or lower depending on endianness) of jiffies_64.

<https://elixir.bootlin.com/linux/v5.11/source/include/linux/jiffies.h#L74>

Dynamic Timers

Timers allow a generic function to be activated at a later time, they can be dynamically created and destroyed and they can be assigned to the kernel itself or to a process. For this reason, we need an efficient way for managing timers in the kernel. Timers are associated with deferrable functions Linux **does not guarantee** that activation takes place at exact time

APIs

- `void init_timer(struct timer_list *timer);`
- `void setup_timer(struct timer_list *timer, void (*function)(unsigned long), unsigned long data);`
- `int mod_timer(struct timer_list *timer, unsigned long expires);`
- `void del_timer(struct timer_list *timer);`
- `int timer_pending(const struct timer_list *timer);`

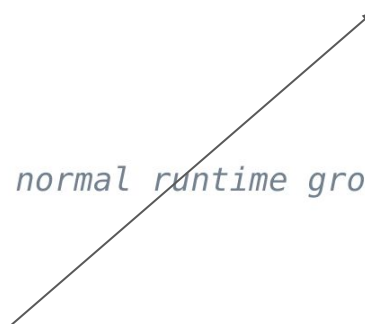
Timers are prone to race conditions (e.g., if resources are released). They should be deleted before releasing the resources.

Dynamic Timers

struct timer_list

A Timer is represented by the struct `timer_list`.

```
11 struct timer_list {
12     /*
13      * All fields that change during normal runtime grouped to the
14      * same cacheline
15      */
16     struct hlist_node    entry;
17     unsigned long        expires;
18     void                 (*function)(struct timer_list *);
19     u32                  flags;
20
21     #ifdef CONFIG_LOCKDEP
22         struct lockdep_map    lockdep_map;
23     #endif
24 };
```



In jiffies

<https://elixir.bootlin.com/linux/v5.11/source/include/linux/timer.h#L11>

Dynamic Timers

The Timer Wheel

Earlier versions of the kernel used a single timer list sorted according to the expiration time. This was significantly unreliable and inefficient. Newer versions of the kernel introduced the so-called **Timer Wheel** a nested structure for efficiently retrieving timers. For understanding the principle of operation we will consider **simplified** example.

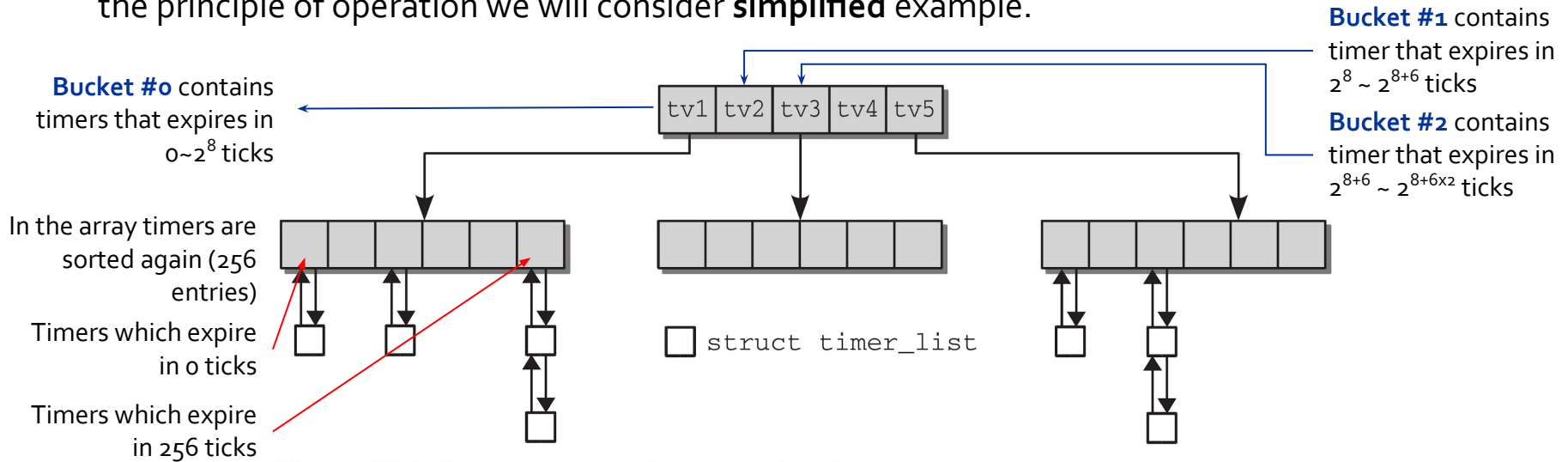


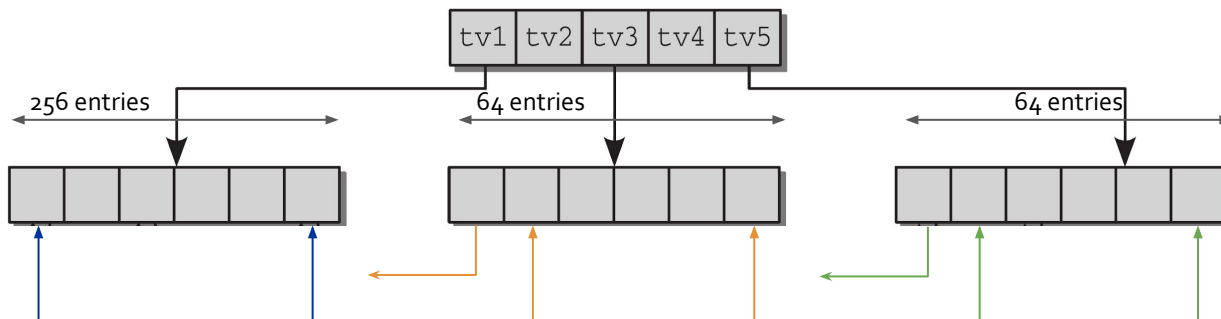
Figure 15-6: Data structures for managing timers.

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

Dynamic Timers

The Timer Wheel

Suppose that each bucket has a counter that stores the number of an array position.



The tv1 counter initially points to 0, then it is incremented at each tick and every timer executed

When 255 is reached the tv1 counter is reset to 0 and the timer of the first position of tv2 bucket replenish the bucket tv1, the counter of tv2 is increased by 1

When 63 is reached the tv2 counter is reset to 0 and the timer of the first position of tv3 bucket replenish the bucket tv2, the counter of tv3 is increased by 1

To determine which timers have expired, the kernel need not scan through an enormous list of timers but can limit itself to checking a *single* array position in the first group

6.2.2

6. Time Management
2. Timekeeping Architecture

Generic Time Subsystem

Components

There are three main abstractions for providing time in the kernel:

- **Clock Sources** (`struct clocksource`) that are the backbox of time management. Essentially each clock provides a monotonically increasing counter with read only access;
- **Clock event devices** (`struct clock_event_device`) add the possibility of equipping clocks with events that occurs in the future.
- **Tick Devices** (`struct tick_device`) extend the clock event sources to provide a continuous stream of tick events that happen periodically. A dynamic tick may stop a tick device if requested.

There are then two kinds of clocks:

- a **global clock** that is responsible for providing the periodic tick for updating jiffies (old PIT)
- **one local clock** for each CPU for performing process accounting, profiling and high-resolution timers

Overview

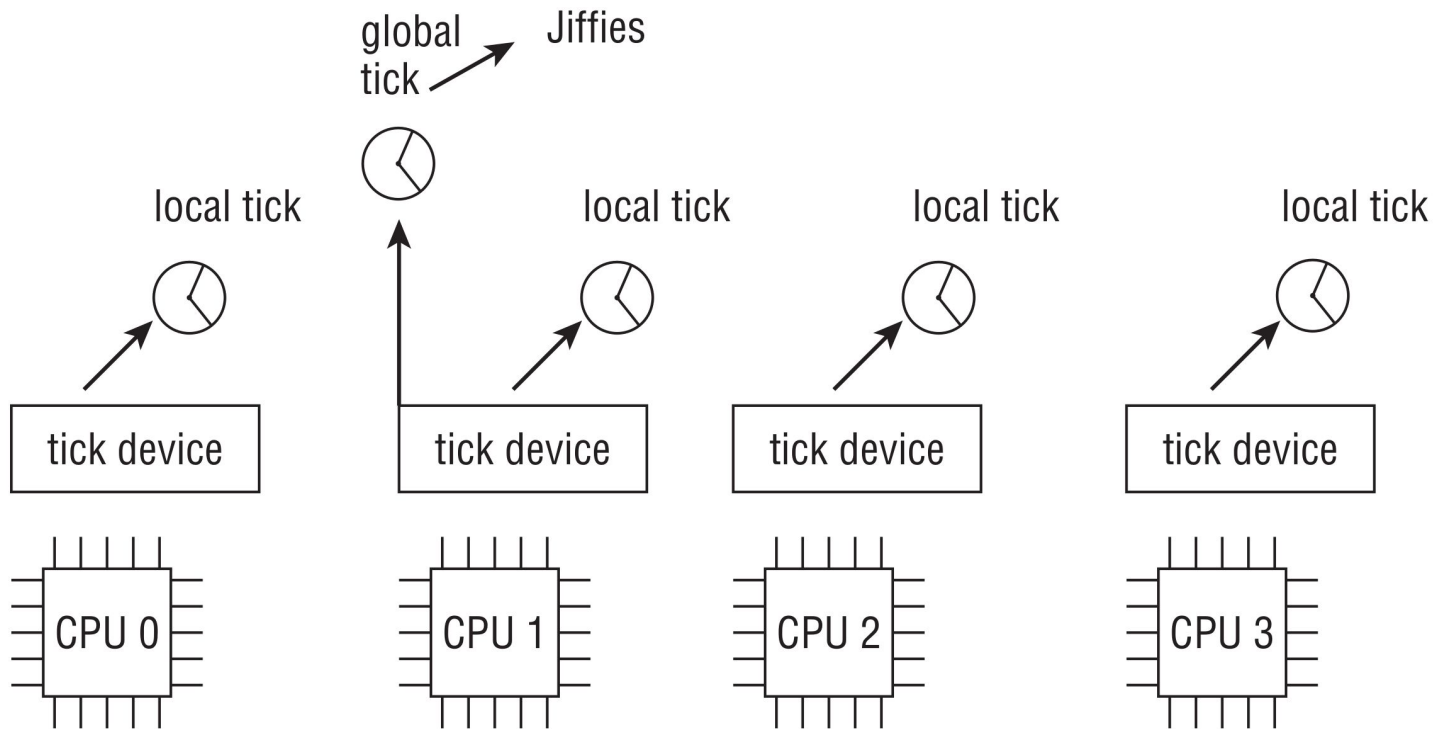


Figure 15-7: Overview of the generic time subsystem.

The clocksource struct represents an abstraction of a source of time. The assigned function read allows to retrieve the current cycle value from the source.

```
33  /**
34   * struct clocksource - hardware abstraction for a free running counter
35   *     Provides mostly state-free accessors to the underlying hardware.
36   *     This is the structure used for system time.
37   *
38   * @read:      Returns a cycle value, passes clocksource as argument
39   * @mask:      Bitmask for two's complement
40   *             subtraction of non 64 bit counters
41   * @mult:      Cycle to nanosecond multiplier
42   * @shift:     Cycle to nanosecond divisor (power of two)
43   * @max_idle_ns: Maximum idle time permitted by the clocksource (nsecs)
44   * @maxadj:    Maximum adjustment value to mult (~11%)
```

```
89  struct clocksource {
90      u64      (*read)(struct clocksource *cs);
91      u64      mask;
92      u32      mult;
93      u32      shift;
94      u64      max_idle_ns;
95      u32      maxadj;
96  #ifdef CONFIG_ARCH_CLOCKSOURCE_DATA
97      struct arch_clocksource_data archdata;
```

Jiffies as Clocksource

V5.11

```
41  * The Jiffies based clocksource is the lowest common
42  * denominator clock source which should function on
43  * all systems. It has the same coarse resolution as
44  * the timer interrupt frequency HZ and it suffers
45  * inaccuracies caused by missed or lost timer
46  * interrupts and the inability for the timer
47  * interrupt hardware to accurately tick at the
48  * requested HZ value. It is also not recommended
49  * for "tick-less" systems.
50  */
51  static struct clocksource clocksource_jiffies = {
52      .name           = "jiffies",
53      .rating         = 1, /* lowest valid rating*/
54      .read           = jiffies_read,
55      .mask           = CLOCKSOURCE_MASK(32),
56      .mult           = TICK_NSEC << JIFFIES_SHIFT, /* details above */
57      .shift          = JIFFIES_SHIFT,
58      .max_cycles     = 10,
59  };
```

<https://elixir.bootlin.com/linux/v5.11/source/kernel/time/jiffies.c#L40>

Timer Interrupt Management

The handling of timer interrupts are handled according to the top/bottom half paradigm (using Task Queues, which have now been removed from the Kernel).

The top half executes the following actions:

- registers the bottom half
- increments jiffies
- checks whether the CPU scheduler needs to be activated, and in the positive case flags `need_resched` (more on this later)

High Resolution Timers

High resolution timers are based on the `ktime_t` type (nanosecond scalar representation) rather than jiffies.

High resolution timers heavily depends on the architecture wrt the low-resolution ones. They:

- are **arranged** in a red-black tree
- they are **independent** of periodic ticks, they are not based on jiffies

Low-resolution timers are based on high-resolution ones.

```
97  /**
98   * struct hrtimer - the basic hrtimer structure
99   * @node:          timerqueue node, which also manages node.expires,
100   *                the absolute expiry time in the hrtimers internal
101   *                representation. The time is related to the clock on
102   *                which the timer is based. Is setup by adding
103   *                slack to the _softexpires value. For non range timers
104   *                identical to _softexpires.
105   * @_softexpires: the absolute earliest expiry time of the hrtimer.
106   *                The time which was given as expiry time when the timer
107   *                was armed.
108   * @function:      timer expiry callback function
109   * @base:          pointer to the timer base (per cpu and per clock)
110   * @state:         state information (See bit values above)
111   * @is_rel:        Set if the timer was armed relative
112   * @is_soft:       Set if hrtimer will be expired in soft interrupt context.
113   * @is_hard:       Set if hrtimer will be expired in hard interrupt context
114   *                even on RT.
115   *
116   * The hrtimer structure must be initialized by hrtimer_init()
117   */
118 struct hrtimer {
119     struct timerqueue_node    node;
120     ktime_t                  _softexpires;
121     enum hrtimer_restart      (*function)(struct hrtimer *);
122     struct hrtimer_clock_base *base;
123     u8                       state;
124     u8                       is_rel;
125     u8                       is_soft;
126     u8                       is_hard;
127 };
```


6.3

6. Time Management

Watchdogs

Watchdogs

A watchdog is a component that monitors a system for “normal” behaviour and if it fails, it performs a system reset to hopefully recover normal operation.

This is a last resort to maintain system availability or to allow sysadmins to remotely log after a restart and check what happened. In Linux, this is implemented in two parts:

- a kernel-level **module** which is able to perform a hard reset
- a **user-space** background **daemon** that refreshes the timer

At kernel level, this is implemented using a Non-Maskable Interrupt (NMI). The userspace daemon will notify the kernel watchdog module via the `/dev/watchdog` special device file that user space is still alive.

```
while (1) {  
    ioctl(fd, WDIOC_KEEPALIVE, 0);  
    sleep(10);  
}
```

Advanced Operating Systems and Virtualization

[6] Time Management

LECTURER

Gabriele **Proietti Mattia**

BASED ON WORK BY

<http://www.ce.uniroma2.it/~pellegrini/>



gpm.name · proiettimattia@diag.uniroma1.it

DIAG