Linux Kernel Training.

Timers, Delays, Deferred Works

March 31, 2023

GlobalLogic

Linux Kernel Time Sources

- Real-time clock (RTC)
  - Battery-backed HW clock;
  - Used to set and keep current date and time even when system is off;
- System Timers (Low Resolution): kernel/time/timer.c ;
  - Generate System ticks (100,250,1000 Hz);
  - Support System Time;
  - Tasks and Events scheduling;
- High Resolution Timers: kernel/time/hrtimer.c ;
  - Integrated into kernel mainline from 2.6.21;
  - Can support resolutions higher than 1 ms.
  - While S/W supports 1 ns resolution, normally rounded to the clock resolution of the specific platform.

Jiffies and HZ

- Jiffies
  - Until 2.6.21, jiffies was just a counter that was incremented every clock interrupt
  - Jiffies can wrap around depending on platform
    - 32 bits, 1000 HZ: about 50 days
    - 64 bits, 1000 HZ: about 600 million years
  - Jiffies_64:
    - On 64 bit machines, jiffies == jiffies_64;
    - On 32 bits, jiffies points to low-order 32 bits, jiffies_64 to high-order bits (be careful about atomicity!) =>

```
u64 get_jiffies_64(void);
```

- HZ
  - Determines how frequently the clock interrupt fires
  - Default is 1000 on x86, or 1 millisecond
  - Configurable at compile time or boot time
    Other typical values are 100 (10 ms) or 250 (4 ms)

- What's a good value for HZ?
  - Low values: less overhead
  - High values: better responsiveness

Kernel time structures

From include/uapi/linux/time(64).h (old type)

```
struct timespec(64) {
     __kernel_time_t     tv_sec;    /* seconds */
     long       tv_nsec;        /* nanoseconds */
};

struct timeval {
     __kernel_time_t      tv_sec;  /* seconds */
     __kernel_suseconds_t tv_usec; /* microseconds */
};
```

From include/uapi/linux/rtc.h

```
struct rtc_time {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;      /* unused */
    int tm_yday;      /* unused */
    int tm_isdst;     /* unused */
};
```

Conversion functions (`jiffies.h`)

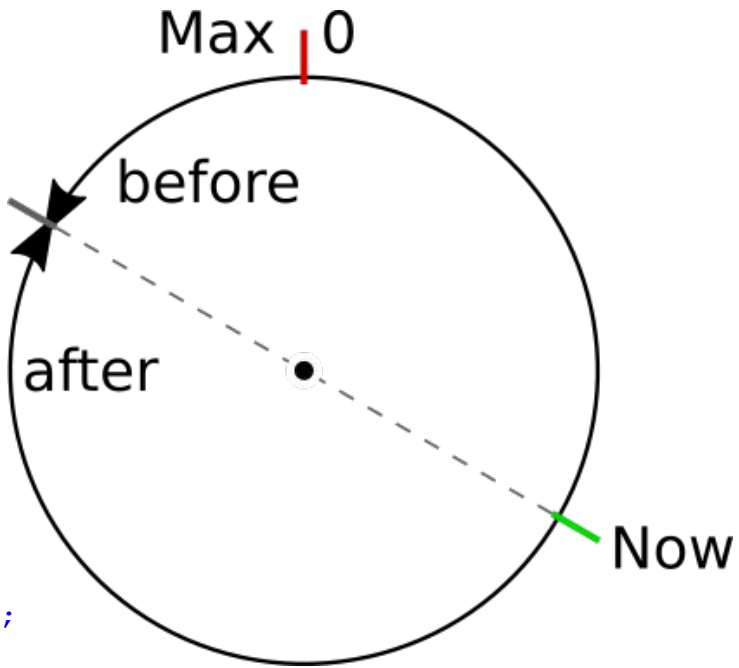
```
unsigned long timespec_to_jiffies(struct timespec *value);
void          jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void          jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
unsigned int  jiffies_to_msecs(const unsigned long j);          /* the same for _usecs */
unsigned long msecs_to_jiffies(const unsigned int m);
```

Measuring Time Lapses

- Using jiffies

```
#include <linux/jiffies.h>
j = jiffies; /* read the current value */
stamp_1 = j + HZ; /* 1 second in the future */
stamp_half = j + HZ/2; /* half a second */
stamp_n = j + n*HZ/1000; /* n milliseconds */

bool time_after(unsigned long a, unsigned long b);
bool time_before(unsigned long a, unsigned long b);
bool time_after_eq(unsigned long a, unsigned long b);
bool time_before_eq(unsigned long a, unsigned long b);
```



- Similar for 64-bit

```
bool time_after64(u64 a, u64 b);
bool time_before64(u64 a, u64 b);
bool time_after_eq64(u64 a, u64 b);
bool time_before_eq64(u64 a, u64 b);
```

Processor Specific Registers

- x86:

  To access the timecounter, include <asm/msr.h> and use the following marcos

```
/* read into two 32-bit variables */
rdtsc(low32,high32);
/* read low half into a 32-bit variable */
rdtscl(low32);
/* read into a 64-bit long long variable */
rdtscll(var64);
```

  1-GHz CPU overflows the low half of the counter every 4.2 seconds

- Linux offers an architecture-independent function to access the architecture-specific cycle counter

```
#include <linux/timex.h>
cycles_t get_cycles(void);
```

  Returns 0 on platforms that have no cycle-counter register

Time Delays

- Busy-waiting

```
while (time_before(jiffies, j1))
        cpu_relax();

while (time_before(jiffies, j1))
        schedule();
```

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

- Non-busy waiting

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
void __sched usleep_range(unsigned long min, unsigned long max);
signed long __sched schedule_timeout(signed long timeout);
     signed long __sched schedule_timeout_interruptible(signed long timeout);
     signed long __sched schedule_timeout_uninterruptible(signed long timeout);
     signed long __sched schedule_timeout_killable(signed long timeout);
```

Delays using WQ

See `linux/wait.h` (really they are macros):

```
DECLARE_WAIT_QUEUE_HEAD(name);
void wait_event(queue, condition);
long wait_event_timeout(wait_queue_head_t q, condition, long timeout);
long wait_event_interruptible_timeout(wait_queue_head_t q, condition, long timeout);
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

Example:

```
#include <linux/wait.h>

wait_queue_head_t wait;

init_waitqueue_head(&wait);
wait_event_interruptible_timeout(wait, 0, delay);
```

Condition = 0 (no condition to wait for). Execution resumes when someone calls wake_up() or timeout expires;

Which function is best for me ?

- Documentation/timers/timers-howto.txt
  - "Is my code in an atomic context?"
  - This should be followed closely by "Does it really need to delay in atomic context?"

```
BUG: scheduling while atomic: swapper/1/0/0xffff0000
Modules linked in: tun libcomposite ipv6
[<c0014e4c>] (unwind_backtrace+0x0/0x11c) from [<c03a0720>] (__schedule_bug+0x48/0x5c)
[<c03a0720>] (__schedule_bug+0x48/0x5c) from [<c03a536c>] (__schedule+0x68/0x6e0)
[<c03a536c>] (__schedule+0x68/0x6e0) from [<c000ef08>] (cpu_idle+0xe4/0xfc)
```

- Are we doing bottom half or hardware interrupt processing? Are we in a softirq context? Interrupt context? See linux/preempt.h

```
#define in_irq()        (hardirq_count())
#define in_softirq()    (softirq_count())
#define in_interrupt()  (irq_count())
...
#define in_atomic()     ((preempt_count() != 0)
```

Contexts

- ATOMIC CONTEXT:

  You **must** use the *delay family of functions.

- NON-ATOMIC CONTEXT:

  You should use the *sleep[_range] family of functions.

  - SLEEPING FOR "A FEW" USECS (< ~10us?):

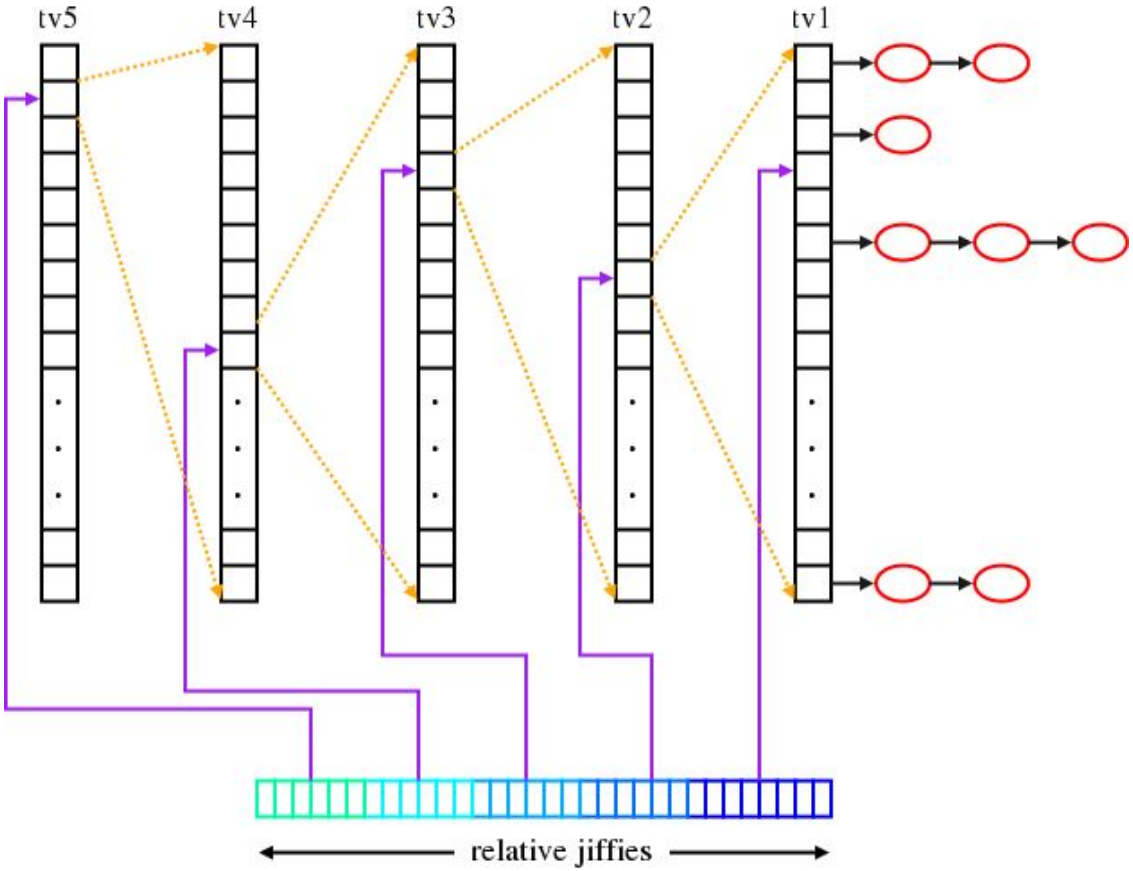
    Use udelay
  - SLEEPING FOR ~USECS OR SMALL MSECS (10us - 20ms):

    Use usleep_range
  - SLEEPING FOR LARGER MSECS (10ms+)

    Use msleep or possibly msleep_interruptible

Kernel Timers

Cascaded Timer Wheel



tv1 - containing a set of 256 (in most configurations) linked lists of upcoming timer events;

tv2 - set of 64 next level timers;

Cascading initiated after all timers of the given level expired;

Timer add and Timer delay complexity O(1)

Timer cascading complexity O(n)

Timer list structure

Up to kernel 4.14

`data` is a pointer to related structure (may be timer itself)

```
struct timer_list {
     struct hlist_node   entry;
     unsigned long       expires;
     void                (*function)(unsigned long);
     unsigned long       data;
     u32                 flags;
};

setup_timer(&mydev.timer, timer_callback, &mydev)

struct mydev *md = t->data;     /* in callback */
```

Since kernel 4.15

Use `container_of`-based macro `from_timer()`

```
struct timer_list {
     struct hlist_node   entry;
     unsigned long       expires;
     void                (*function)(struct timer_list *);
     u32                 flags;
};

timer_setup(&mydev.mytimer, timer_callback, TIMER_FLAGS);

struct mydev *md = from_timer(md, t, mytimer); /* in callback */
```

Kernel Timer API

- **Creation and manipulation**

```
void init_timer(struct timer_list *timer);
void init_timer_deferrable(struct timer_list *timer);
void add_timer(struct timer_list * timer);
int del_timer(struct timer_list * timer);
int del_timer_sync(struct timer_list *timer);
int mod_timer(struct timer_list *timer, unsigned long expires);
```

- **Example (drivers/pci/hotplug/cpqphp_ctrl.c, pre-4.15 style)**

```
init_timer(&p_slot->task_event);
p_slot->task_event.expires = jiffies + 5 * HZ;   /* 5 second delay */
p_slot->task_event.function = pushbutton_helper_thread;
p_slot->task_event.data = (u32) p_slot;
add_timer(&p_slot->task_event);
//
mod_timer(&my_timer, next_timeout);
// ----
del_timer(&p_slot->task_event);
```

High Resolution Timers

- Motivated by the observation of 2 types of timers:
  - Timeout functions, which we don't expect to actually happen (e.g., retransmission timer for packet loss). Have low resolution and are usually removed before expiration.
  - Timer functions, which we do expect to run. Have high resolution requirements and usually expire
- Original timer implementation is based on jiffies and thus depends on HZ.
  - Works well for timeouts, less so for timers.
  - Resolution no better than HZ (e.g., 1 millisecond)
- High resolution timers, introduced in 2.6.16, allow 1 nanosecond resolution

  Implemented in an red-black tree (rbtree)
- Insert, delete, search in O(log n) time

High Resolution Timers API

- #include <linux/ktime.h>

```
union ktime {
     s64 tv64;      // in nanoseconds
};
```

- Initialization of time variable (defined in include/linux/ktime.h)

```
ktime_t kt;
kt = ktime_set(long secs, long nanosecs);

ktime_t ktime_add(ktime_t kt1, ktime_t kt2);
ktime_t ktime_sub(ktime_t kt1, ktime_t kt2);  /* kt1 - kt2 */
ktime_t ktime_add_ns(ktime_t kt, u64 nanoseconds);
ktime_t timespec_to_ktime(struct timespec tspec);
ktime_t timeval_to_ktime(struct timeval tval);
struct timespec ktime_to_timespec(ktime_t kt);
struct timeval ktime_to_timeval(ktime_t kt);
clock_t ktime_to_clock_t(ktime_t kt);
u64 ktime_to_ns(ktime_t kt);
```

High Resolution Timers API

void hrtimer_init(struct hrtimer *timer, clockid_t which_clock);

1. CLOCK_MONOTONIC: a clock which is guaranteed always to move forward in time, but which does not reflect "wall clock time"
2. CLOCK_REALTIME which matches the current real-world time.

void hrtimer_rebase(struct hrtimer *timer, clockid_t new_clock);

- Callback function :

```
int   (*function)(void *);
```

  - HRTIMER_NORESTART
  - HRTIMER_RESTART
- Setting restart time:

```
u64 hrtimer_forward(struct hrtimer *timer, ktime_t now, ktime_t interval);
u64 hrtimer_forward_now(struct hrtimer *timer,ktime_t interval);
static inline u64 hrtimer_forward_now(struct hrtimer *timer, ktime_t interval)
{
     return hrtimer_forward(timer, timer->base->get_time(), interval);
}
```

High Resolution Timers API

- **int hrtimer_start(struct hrtimer *timer, ktime_t time, enum hrtimer_mode mode);**
  - HRTIMER_ABS
  - HRTIMER_REL
- **int hrtimer_cancel(struct hrtimer *timer);**
  - The return value will be zero if the timer was not active (meaning it had already expired, normally), or one if the timer was successfully canceled;
- **int hrtimer_try_to_cancel(struct hrtimer *timer);**
  - Returns -1 if the timer function is running;
- **void hrtimer_restart(struct hrtimer *timer)** - can restart cancelled timer;
- **ktime_t hrtimer_get_remaining(const struct hrtimer *timer);**
- **bool hrtimer_active(const struct hrtimer *timer);**
- **int hrtimer_get_res(clockid_t which_clock, struct timespec *tp);**

HRT Example

```
int init_module(void)
{
     ktime_t ktime;

     ktime = ktime_set(0, MS_TO_NS(delay_in_ms));
     hrtimer_init(&hr_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
     hr_timer.function = &my_hrtimer_callback;
     hrtimer_start( &hr_timer, ktime, HRTIMER_MODE_REL );
     return 0;
}

enum hrtimer_restart my_hrtimer_callback( struct hrtimer *timer)
{
     if (restart--) {
          hrtimer_forward_now(timer, ns_to_ktime(MS_TO_NS(delay_in_ms)));
          return HRTIMER_RESTART;
     }
     return HRTIMER_NORESTART;
}
```

Summary

- Timer sources: System and RTC
- Jiffies - time measurement quant
- HZ - divider value
- Small resolution timer and high resolution timer are stored in different way in kernel

Thanks!