# Timers and Time Management

*Dongyoon Lee*

# Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
  - syscall, module, kernel data structures
- Process management & scheduling
- Interrupt & interrupt handler
- Kernel synchronization

# Today: timers and time management

- Kernel notion of time
- Tick rate and Jiffies
- hardware clocks and timers
- Timers
- Delaying execution
- Time of day

# Kernel notion of time

- Having the notion of time passing in the kernel is essential in multiple cases:
  - Perform periodic tasks (e.g., CFS time accounting)
  - Delay some processing at a relative time in the future
  - Give the time of the day
- **Absolute** vs **relative** time

# Kernel notion of time

- Central role of the system timer
  - Periodic interrupt, system timer interrupt
  - Update system uptime, time of day, balance runqueues, record statistics, etc.
  - Pre-programmed frequency, timer tick rate
  - tick = 1/(tick rate) seconds
- Dynamic timers to schedule event a relative time from now in the future

# Tick rate and jiffies

- The tick rate (system timer frequency) is defined in the `HZ` variable
- Set to `CONFIG_HZ` in include/asm-generic/param.h
    - Kernel compile-time configuration option
- Default value is per-architecture:

| Architecture | Frequency (HZ) | Period (ms) |
|---|---|---|
| x86 | 1000 | 1 |
| ARM | 100 | 10 |
| PowerPC | 100 | 10 |

# Tick rate: the ideal `HZ` value

- High timer frequency → high precision
  - Kernel timers (finer resolution)
  - System call with timeout value (e.g., `poll`) → significant performance improvement for some applications
  - Timing measurements

# Tick rate: the ideal `HZ` value

- High timer frequency → high precision
  - Process preemption occurs more accurately → low frequency allows processes to potentially get (way) more CPU time after the expiration of their timeslices
- High timer frequency → more timer interrupt → larger overhead
  - Not very significant on modern hardware

# Tickless OS

- Option to compile the kernel as a tickless system
    - `NO_HZ` family of compilation options
- The kernel dynamically reprogram the system timer according to the current timer status
    - Situation in which there are no events for hundreds of milliseconds
- Overhead reduction, Energy savings
    - CPUs spend more time in low power idle states

# `jiffies`

- A global variable holds the number of timer ticks since the system booted (`unsigned long`)
- Conversion between `jiffies` and seconds
  - `jiffies` = seconds * `HZ`
  - seconds = `jiffies` / `HZ`

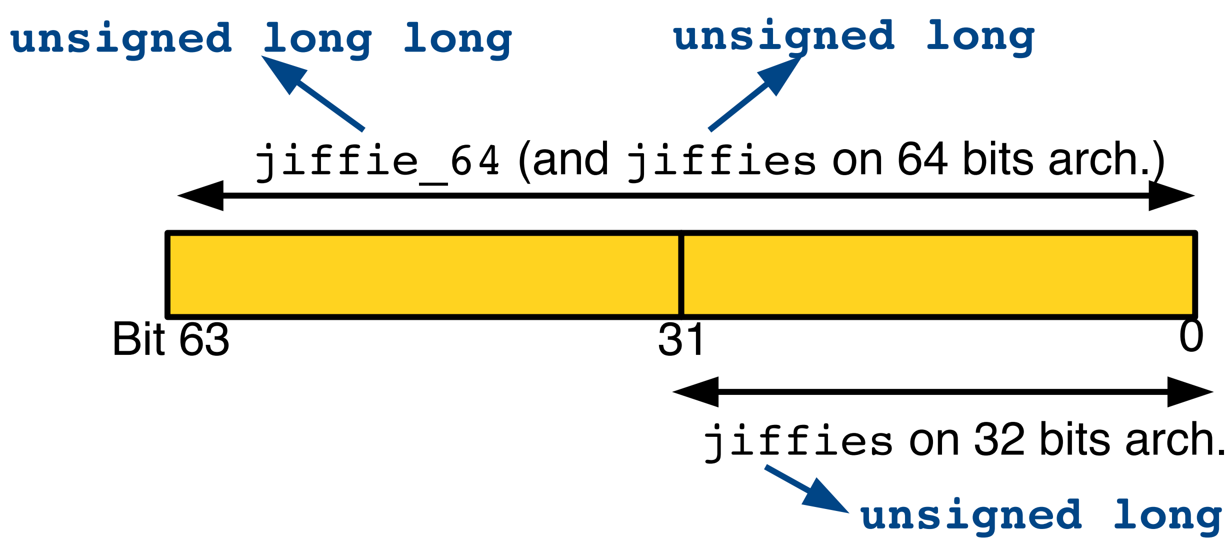
```
unsigned long time_stamp = jiffies;         /* Now */
unsigned long next_tick = jiffies + 1;      /* One tick from now */
unsigned long later = jiffies + 5*HZ;       /* 5 seconds from now */
unsigned long fraction = jiffies + HZ/10;   /* 100 ms from now */
```

# Internal representation of `jiffies`

- `sizeof(jiffies)` is 32 bits on 32-bit architectures and 64 bits for 64-bit architectures
- On a 32 bits variable with `HZ` == 100, overflows in 497 days
  - Still on 32 bits with `HZ` == 1000, overflows in 50 days
- But on a 64 bits variable, no overflow for a very long time

# Internal representation of `jiffies`

- We want access to a 64 bits variable while still maintaining an unsigned long on both architectures → linker magic



# `jiffies` wraparound

- An unsigned integer going over its maximum value wraps around to zero
  - On 32 bits, `0xFFFFFFFF + 0x1 == 0x0`

```
/* WARNING: THIS CODE IS BUGGY */
unsigned long timeout = jiffies + HZ/2; /* timeout in 0.5s */

/* do some work ... */

/* then see whether we took too long */
if (timeout > jiffies) { /* What happen if jiffies wrapped back to zero? */
    /* we did not time out, good ... */
} else {
    /* we timed out, error ... */
}
```

# `jiffies` wraparound

```
/* linux/include/linux/jiffies.h */
#define time_after(a,b)
#define time_before(a,b)
#define time_after_eq(a,b)
#define time_before_eq(a,b)

/* ---------------------------------- */
/* An example of using a time_*() macro */
unsigned long timeout = jiffies + HZ/2; /* timeout in 0.5s */

/* do some work ... */

/* then see whether we took too long */
if (time_before(jiffies, timeout)) { /* Use time_*() macros */
    /* we did not time out, good ... */
} else {
    /* we timed out, error ... */
}
```

# Userspace and `HZ`

- For conversion between architecture-specific `jiffies` and user-space clock tick, Linux kernel provides APIs and macros
- `USER_HZ`: user-space clock tick (100 in x86)
- Conversion between `jiffies` and user-space clock tick
  - `clock_t jiffies_to_clock(unsigned long x);`
  - `clock_t jiffies_64_to_clock_t(u64 x);`
- [clock(3)]()

# Hardware clocks and timers

- Real-Time Clock (RTC)
  - Stores the wall-clock time (still incremented when the computer is powered off)
  - Backed-up by a small battery on the motherboard
  - Linux stores the wall-clock time in a data structure at boot time (`xtime`)

# Hardware clocks and timers

- System timer
  - Provide a mechanism for driving an interrupt at a periodic rate regardless of architecture
  - System timers in x86
    - Local APIC timer: primary timer today
    - Programmable interrupt timer (PIT): was a primary timer until 2.6.17

# Hardware clocks and timers

- Processor's time stamp counter (TSC)
  - `rdtsc`, `rdtscp`
  - most accurate (CPU clock resolution)
  - invariant to clock frequency (x86 architecture)
    - seconds = clocks / maximum CPU clock Hz

# Timer interrupt processing

- Constituted of two parts: (1) architecture-dependent and (2) architecture-independent
- Architecture-dependent part is registered as the handler (top-half) for the timer interrupt
    1. Acknowledge the system timer interrupt (reset if needed)
    2. Save the wall clock time to the RTC
    3. Call the architecture independent function (still executed as part of the top-half)

# Timer interrupt processing

- Architecture independent part: `tick_handle_periodic()`
    1. Call `tick_periodic()`
    2. Increment `jiffies64`
    3. Update statistics for the currently running process and the entire system (load average)
    4. Run dynamic timers
    5. Run `scheduler_tick()`

# Timer interrupt processing

```
/* linux/kernel/time/tick-common.c */

static void tick_periodic(int cpu)
{
    if (tick_do_timer_cpu == cpu) {
        write_seqlock(&jiffies_lock);

        /* Keep track of the next tick event */
        tick_next_period =
            ktime_add(tick_next_period, tick_period);

        do_timer(1); /* ! */
        write_sequnlock(&jiffies_lock);
        update_wall_time(); /* ! */
    }

    update_process_times(
        user_mode(get_irq_regs())); /* ! */
    profile_tick(CPU_PROFILING);
}
```

# do_timer()

```
/* linux/kernel/time/timekeeping.c */

void do_timer(unsigned long ticks)
{
    jiffies_64 += ticks;
    calc_global_load(ticks);
}
```

# `update_process_times()`

- Call account_process_tick() to add one tick to the time passed:
  - In a process in user space
  - In a process in kernel space
  - In the idle task
- Call `run_local_timers()` and run expired timers
  - Raise the TIMER_SOFTIRQ softirq

# `update_process_times()`

- Call `scheduler_tick()`
  - Call the `task_tick()` function of the currently running process's scheduler class → Update timeslices information → Set `need_resched` if needed
  - Perform CPU runqueues load balancing (raise the `SCHED_SOFTIRQ` softirq)

# Timer

- Timers == dynamic timers == kernel timers
  - Used to delay the execution of some piece of code for a given amount of time

```
/* linux/include/linux/timer.h */

struct timer_list {
    struct hlist_node entry;   /* linked list of timers */
    unsigned long     expires; /* expiration time in jiffies */
    void (*function)(unsigned long); /* handler */
    unsigned long     data;    /* argument of the handler */
    u32               flags;   /*
        TIMER_IRQSAFE: executed with interrupts disabled
        TIMER_DEFERRABLE: does not wake up an idle CPU */
    /* ... */
}
```

# Using timers

```
/* Declaring, initializing and activating a timer */
void handler_name(unsigned long data)
{
    /* executed when the timer expires */
    /* ... */
}

void another_function(void)
{
    struct timer_list my_timer;

    init_time(&my_timer);                   /* initialize internal fields */
    my_timer.expires = jiffies + 2*HZ;      /* expires in 2 secs */
    my_timer.data = 42;                     /* 42 passed as parameter to the handler */
    my_timer.function = handler_name;

    /* activate the timer: */
    add_timer(&my_timer);
}

/*********************************************************/
/* Modify the expiration date of an already running timer */
mod_timer(&my_timer, jiffies + another_delay);
```

# Using timers

- `del_timer(struct timer_list *)`
  - Deactivate a timer prior
  - Returns 0 if the timer is already inactive, and 1 if the timer was active
  - Potential race condition on SMP when the handler is currently running on another core

# Using timers

- `del_timer_sync(struct timer_list *)`
  - Waits for a potential currently running handler to finishes before removing the timer
  - Can be called from interrupt context only if the timer is irqsafe (declared with TIMER_IRQSAFE)
  - Interrupt handler interrupting the timer handler and calling `del_timer_sync()` → deadlock

# Timer race conditions

- Timers run in softirq context → Several potential race conditions exist
- Protect data shared by the handler and other entities
- Use `del_timer_sync()` rather than `del_timer()`
- Do not directly modify the `expire` field; use `mod_timer()`

```
/* THIS CODE IS BUGGY! DO NOT USE! */
del_timer(&my_timer);
my_timer->expires = jiffies + new_delay;
add_timer(&my_timer);
```

# Timer implementation

- In the system timer interrupt handler, `update_process_times()` is called
  - Calls `run_local_timers()`
  - Raises a softirq (`TIMER_SOFTIRQ`)

# Timer implementation

- Softirq handler is `run_timer_softirq()` and it calls `__run_timers()`
  - Grab expired timers through `collect_expired_timers()`
  - Executes `function` handlers with `data` parameters for expired timers with `expire_timers()`
- **Timer handlers are executed in interrupt (softirq) context**

# Timer example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/timer.h>

#define PRINT_PREF  "[TIMER_TEST] "

struct timer_list my_timer;

static void my_handler(unsigned long data)
{
    printk(PRINT_PREF "handler executed!\n");
}

static int __init my_mod_init(void)
{
    printk(PRINT_PREF "Entering module.\n");

    /* initialize the timer data structure internal values: */
    init_timer(&my_timer);
```

# Timer example

```
    /* fill out the interesting fields: */
    my_timer.data = 0;
    my_timer.function = my_handler;
    my_timer.expires = jiffies + 2*HZ; /* timeout == 2secs */

    /* start the timer */
    add_timer(&my_timer);
    printk(PRINT_PREF "Timer started\n");

    return 0;
}

static void __exit my_mod_exit(void)
{
    del_timer(&my_timer);
    printk(PRINT_PREF "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);
```

# Delaying execution

- Sometimes the kernel needs to wait for some time without using timers (bottom-halves)
  - For example drivers communicating with the hardware
  - Needed delay can be quite short, sometimes shorter than the timer tick period
- Several solutions
  - Busy looping
  - Small delays and BogoMIPS
  - `schedule_timeout()`

# Busy looping

- Spin on a loop until a given amount of ticks has elapsed
  - Can use `jiffies`, `HZ`, or `rdtsc`
  - Busy looping is good for delaying very short period time but in general it is sub-optimal as wasting CPU cycles.

# Busy looping

- A better solution is to leave the CPU while waiting using `cond_resched()`
  - `cond_resched()` invokes the scheduler only if the `need_resched` flag is set
  - Cannot be used from interrupt context (not a schedulable entity)
  - Pure busy looping is probably also not a good idea from interrupt handlers as they should be fast
  - Busy looping can severely impact performance while a lock is help or while interrupts are disabled

# Busy looping

```
/* Example 1: wait for 10 time ticks */
unsigned long timeout = jiffies + 10;   /* timeout in 10 ticks */
while(time_before(jiffies, timeout));   /* spin until now > timeout */

/* Example 2: wait for 2 seconds */
unsigned long timeout = jiffies + 2*HZ; /* 2 seconds */
while(time_before(jiffies, timeout));

/* Example 3: wait for 1000 CPU clock cycles */
unsinged long long timeout = rdtsc() + 1000;
while(rdtsc() > timeout);

/* Example 4: wait for 2 seconds using cond_resched()*/
unsigned long delay = jiffies + 2*HZ;
while(time_before(jiffies, delay))
    cond_resched(); /* WARNING: cannot use in interrupt context */
```

# Small delays and BogoMIPS

- What if we want to delay for time shorter than one clock tick?
  - If `HZ` is 100, one tick is 10 ms
  - If `HZ` is 1000, one tick is 1 ms
- Use `mdelay()`, `udelay()`, or `ndelay()`
  - Implemented as a busy loop
  - `udelay/ndelay` should only be called for delays <1ms due to risk of overflow

# Small delays and BogoMIPS

- Kernel knows how many loop iterations the kernel can be done in a given amount of time: **BogoMIPS**
  - Unit: iterations/jiffy
  - Calibrated at boot time
  - Can be seen in /proc/cpuinfo

# Small delays and BogoMIPS

```
/* linux/include/linux/delay.h */

void mdelay(unsigned long msecs);
void udelay(unsigned long usecs); /* only for delay <1ms due to overflow */
void ndelay(unsigned long nsecs); /* only for delay <1ms due to overflow */
```

# `schedule_timeout()`

- `schedule_timeout()` put the calling task to sleep for at least `n` ticks
  - Must change task status to TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE
  - Should be called from process context without holding any lock

```
set_current_state(TASK_INTERRUPTIBLE); /* can also use TASK_UNINTERRUPTIBLE */
schedule_timeout(2 * HZ); /* go to sleep for at least 2 seconds */
```

# Sleeping on a waitqueue with a timeout

- Tasks can be placed on wait queues to wait for a specific event
- To wait for such an event with a timeout:
  - Call `schedule_timeout()` instead of `schedule()`

# `schedule_timeout()` implementation

```
signed long __sched schedule_timeout(signed long timeout)
{
    struct timer_list timer;
    unsigned long expire;

    switch (timeout)
    {
    case MAX_SCHEDULE_TIMEOUT:
        schedule();
        goto out;
    default:
        if (timeout < 0) {
            printk(KERN_ERR "schedule_timeout: wrong timeout "
                "value %lx\n", timeout);
            dump_stack();
            current->state = TASK_RUNNING;
            goto out;
        }
    }
```

# `schedule_timeout()` implementation

```
    expire = timeout + jiffies;
    setup_timer_on_stack(&timer, process_timeout, (unsigned long)current);
    __mod_timer(&timer, expire, false);
    schedule();
    del_singleshot_timer_sync(&timer);

    /* Remove the timer from the object tracker */
    destroy_timer_on_stack(&timer);

    timeout = expire - jiffies;

 out:
    return timeout < 0 ? 0 : timeout;
}
```

- When the timer expires, `process_timeout()` calls `wake_up_process()`

# The time of day

- Linux provides plenty of function to get / set the time of the day
- Several data structures to represent a given point in time
  - `struct timespec` and `union ktime`

```
/* linux/include/uapi/linux/time.h */
struct timespec {
    __kernel_time tv_sec;  /* seconds */
    long          tv_nsec; /* nanoseconds */
    /* __kernel_time_t is long on x86_64 */
}

/* linux/include/linux/time64.h */
#define timespec64 timespec

/* linux/include/linux/ktime.h */
union ktime {
    s64 tv64; /* nanoseconds */
};
typedef union ktime ktime_t;
```

# The time of day: example

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/timekeeping.h>
#include <linux/ktime.h>
#include <asm-generic/delay.h>

#define PRINT_PREF  "[TIMEOFDAY]: "

extern void getboottime64(struct timespec64 *ts);

static int __init my_mod_init(void)
{
    unsigned long seconds;
    struct timespec64 ts, start, stop;
    ktime_t kt, start_kt, stop_kt;

    printk(PRINT_PREF "Entering module.\n");

    /* Number of seconds since the epoch (01/01/1970) */
    seconds = get_seconds();
    printk("get_seconds() returns %lu\n", seconds);
```

# The time of day: example

```
/* Same thing with seconds + nanoseconds using struct timespec */
ts = current_kernel_time64();
printk(PRINT_PREF "current_kernel_time64() returns: %lu (sec),"
    "i %lu (nsec)\n", ts.tv_sec, ts.tv_nsec);

/* Get the boot time offset */
getboottime64(&ts);
printk(PRINT_PREF "getboottime64() returns: %lu (sec),"
    "i %lu (nsec)\n", ts.tv_sec, ts.tv_nsec);

/* The correct way to print a struct timespec as a single value: */
printk(PRINT_PREF "Boot time offset: %lu.%09lu secs\n",
    ts.tv_sec, ts.tv_nsec);
/* Otherwise, just using %lu.%lu transforms this:
 * ts.tv_sec  == 10
 * ts.tv_nsec == 42
 * into: 10.42 rather than 10.000000042
 */

/* another interface using ktime_t */
kt = ktime_get();
printk(PRINT_PREF "ktime_get() returns %llu\n", kt.tv64);
```

# The time of day: examples

```
    /* Subtract two struct timespec */
    getboottime64(&start);
    stop = current_kernel_time64();
    ts = timespec64_sub(stop, start);
    printk(PRINT_PREF "Uptime: %lu.%09lu secs\n", ts.tv_sec, ts.tv_nsec);

    /* measure the execution time of a piece of code */
    start_kt = ktime_get();
    udelay(100);
    stop_kt = ktime_get();

    kt = ktime_sub(stop_kt, start_kt);
    printk(PRINT_PREF "Measured execution time: %llu usecs\n", (kt.tv64)/1000);

    return 0;
}
```

# The time of day: examples

```
static void __exit my_mod_exit(void)
{
    printk(PRINT_PREF "Exiting module.\n");
}

module_init(my_mod_init);
module_exit(my_mod_exit);

MODULE_LICENSE("GPL");
```

# Next lecture

- Memory management