CS 5264/4224; ECE 5414/4414 (Advanced) Linux Kernel Programming Lecture 12

Time

March 6, 2025

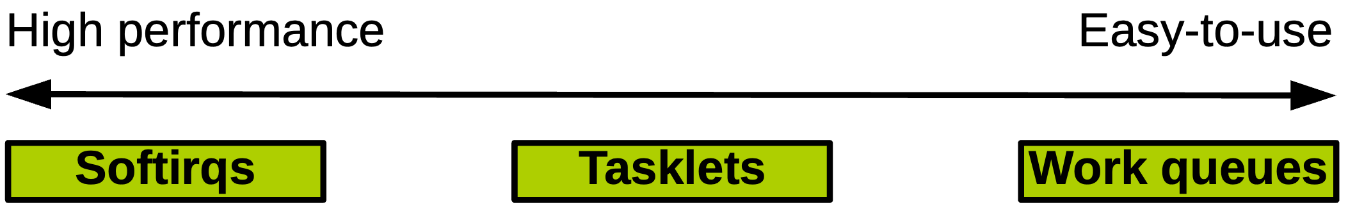
Huaicheng Li

https://people.cs.vt.edu/huaicheng/lkp-sp25/

Acknowledgement: Credits to Dr. Changwoo Min for the original LKP lecture slides.

Today's Bottom Halves in Linux

- All bottom-half mechanisms run with all interrupts enabled
- Softirqs and tasklets run in interrupt context
  - Softirq is rarely used directly
  - Tasklet is a simple and easy-to-use softirq (bult on softirq)
- Work queues run in process context
  - They can block and go to sleep



Choosing the Right Bottom-Half

Bottom half	Context	Inherent serialization
Softirq	Interrupt	None
Tasklet	Interrupt	Against the same tasklet
Workqueue	Process	None

All of these generally run with interrupts enabled

If there is shared data with an interrupt handler (top-half), need to disable interrupts or use locks

Tasklet is depreciated, use workqueue!

Today's Agenda

- Kernel's notion of time
- Ticks and Jiffies
- Hardware clocks and timers
- Timers
- Delayed execution
- Wall clock time

Kernel Notion of Time

- Having the notion of time passing in the kernel is essential in many cases:
  - Perform periodic tasks (e.g., scheduler time accounting)
  - Delay tasks to run later
  - Get time of the day
- Absolute vs. relative time
- Central role of the system timer
  - periodic interrupts, system timer interrupt
  - Update system uptime, time of day, balance runqueues, update statistics, etc
  - Pre-programmed frequency, timer tick rate
  - tick = 1 / (tick rate) seconds
- Dynamic timers to scheduler event to run at a relative time from now in the future

Ticks 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 architecture dependent*

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

Tick Rate: Ideal HZ Value

- High timer frequency → high precision
  - kernel timers (fine resolution)
  - system call with timeout value (e.g., poll) → significant performance improvement for some applications
  - timing measurement
  - process preemption occurs more accurately → low frequency allows processes to potentially get (way) more CPU time after the expiration of their timeslices
- However, higher timer frequency would lead to
  - more timer interrupts → larger overhead
    - » Not significant on modern hardware

Tickless OS

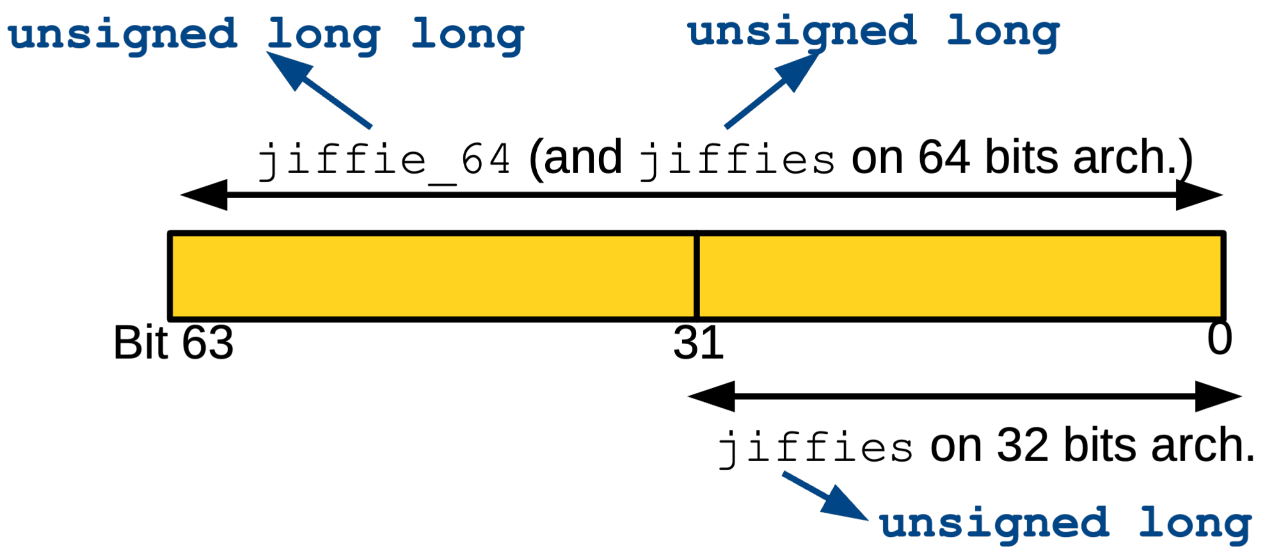
- Option to compile the kernel as a tickless system
  - ***NO_HZ*** family of compilation options
- The kernel dynamically reprogram the system timer according to the current timer status
  - Situation in which there are no events for hundreds of milliseconds
- Overhead reduction, energy savings
  -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

jiffies

- A global variable holds the number of timer ticks since the system boots
  - unsigned long
  - sizeof(jiffies) = 32 on 32-bit architectures and 64 on 64-bit architectures
- On 32-bit architecture with HZ=100, overflows in 497 days
  - If HZ=1000, overflows in 50 days
- No overflow in a practical timeframe
- 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 */
```

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



jiffies Wraparound

- An unsigned integer going over its maximum value wraps around to zero
  - on 32-bit architecture, 0xFFFFFFFF + 0x1 is 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 ... */
}
```

```c
/* 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

- clock(3)
- For conversion between architecture-specific jiffies and userspace clock ticks, Linux kernel provides APIs and macros
- USER_HZ: kernel-internal constant to represent # of ticks per seconds for userspace timekeeping
  - e..g, on x86, fixed at 100 regardless of HZ (kernel's tick rate)
  - sysconf(_SC_CLK_TCK)
  - cat /proc/stat, /proc/uptime, /proc/loadavg
- Conversion between jiffies and user-space clock ticks
  - clock_t jiffies_to_clock_t(unsigned long x);
  - clock_t jiffies_64_to_clock_t(u64 x);

Hardware Clocks and Timers

- Real-Time Clock (RTC)
  - Stores the wall clock time (functional when server is powered off)
  - Backed up by a small battery on the motherboard
  - Linux stores the wall clock time in a data structure (xtime) at boot time
- System timer
  - Provides a mechainsm for driving an interrupt at a periodic rate regardless of architcture
  - System timers in x86
    - » Local APIC: primary timer today
    - » Programmable interrupt timer (PIT), until 2.6.17
- Processor's time stamp counter (TSC)
  - rdtsc, rdtscp
  - most accurate (CPU clock cycle resolution)
  - invariant to clock frequency (x86 architecture)
    - » seconds = clocks / maximum CPU clock Hz

Timer Interrupt Processing

- It consists of two parts
  - top half: arch-dependent
  - bottom half: arch-independent
- Top-half
  - acknowledge the system timer interrupt (reset if needed)
  - save the wall clock time to the RTC
  - call arch-independent bottom-half function (executed as part of the top-half)
- Bottom-half: tick_handle_periodic()
  - call tick_periodic()
  - increment jiffies64
  - update statistics for the currently running process and the entire system (load average)
  - run dynamic timers
  - run scheduler_tick()

```c
/* 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);
}
```

```c
/* 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
- Call scheduler_tick()
  - Call the task_tick() function of the currently running process's scheduler class → update timeslice 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 certain 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);
}
```

- del_timer(struct timer_list *)
  - Deactivate a timer
  - 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
- del_timer_sync(struct timer_list *)
  - Wait for a potential currently running handler to finish 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
- Proctect 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)
  - Softirq handler is run_timer_softirq() and it calls __run_timers()
    - » Grab expired timers through collect_expired_timers()
    - » Execute function handlers with data parameters for expired timers with expire_timers()
  - Timer handlers are executed in interupt (softirq) context

```c
#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);

    /* 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)
  - e.g., drivers communicating with the hardware
  - Needed delay can be quite short, sometimes shorter than the timer tick period
- Several solutions
  - Busy looping: spin on a loop until certain # of ticks has elapsed
    - » Can use jiffies, HZ, or rdtsc
    - » busy looping is good for delaying very short period time but wastes CPU cycles
  - Small delays and BogoMIPS
  - schedule_timeout()

Busy Looping

- Use loop together with 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 held or while interrupts are disabled

```
/* 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 a period of time shorter than one clock tick?
  - If HZ is 100, 1 tick is 10ms
  - If HZ is 1000, 1 tick is 1ms
- Use mdelay(), udelay(), ndelay()
  - Implemented as a busy loop
  - udelay()/ndelay() should only be called for delays <1ms due to potential overflows
  - kernel knows how many loop interations the kernel can be done in a given amount of time: BogoMIPS
    - » Unit: iterations/jiffy
    - » Calibrated at boot time
    - » Can be see in /proc/cpuinfo

```
/* 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() puts 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
- Tasks can be placed on waitqueues to wait for a specific event
  - To wait for such an event without a timeout, call schedule_timeout() instead of schedule()

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

```c
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;
        }
    }

    expire = timeout + jiffies;
    setup_timer_on_stack(&timer, process_timeout, (unsig
    __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() call wake_up_process()

Time of Day

- Linux provides various APIs to get / set the time of the day
- Several data structures to represent a given point in time
  - struct timespec, 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;
```

Example

```c
#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);

    /* 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);


    /* 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;
}
```

- Enjoy the Spring break!
- What's next?
  - kernel synchronization mechanisms
  - mm