Linux Kernel Programming
Time Management

Pierre Olivier

Systems Software Research Group @ Virginia Tech

March 19, 2017

Outline

Outline

Kernel notion of time

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

Outline

Tick rate and Jiffies

Tick rate: HZ

- 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 (in Hertz)	Period (ms)
x86	100	10
arm	100	10
Alpha	1024	1
...		

Tick rate and Jiffies

Tick rate: the ideal `HZ` value

- **High vs low system timer frequency**
- **High timer frequency pros:**
  - High ***precision*** for:
    - Kernel timers (finer resolution)
    - System call with timeout value (ex: `poll`)
      - Significant performance improvement for some applications
    - Timing measurements
  - ***Process preemption occurs more accurately***
    - Low frequency allows processes to potentially get (way) more CPU time after the expiration of their timeslices
- **Cons:**
  - More interrupts, more overhead
    - Not very significant on modern hardware

Tick rate and Jiffies
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

Tick rate and Jiffies
Jiffies

- **`jiffies`** is a global variable containing the number of timer ticks since the system booted
- **`unsigned long`**
- `include/linux/jiffies.h`:

```
extern unsigned long volatile __jiffy_data jiffies;
```

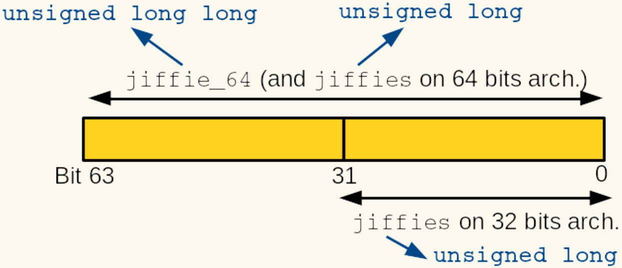
- Conversions:
  - Seconds → `jiffies`: `(seconds * HZ)`
  - `jiffies` → seconds: `(jiffies / HZ)`

```
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 */
```

Tick rate and Jiffies

Jiffies: internal representation

- `unsigned long` size is 32 bits on 32 bits architectures, and 64 bits for 64 bits 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
    - Want access to a 64 bits variable while still maintaining an `unsigned long` on both architectures → linker magic



Tick rate and Jiffies

Jiffies: wraparound

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

```
unsigned long timeout = jiffies + HZ/2; /* timeout in .5 seconds */

/* do some work ... */

/* then check if we timed out */
if (jiffies < timeout) {
  /* we did not time out */
} else {
  /* timeout, error */
}
```

- If `jiffies` wraps around, chances are it will be inferior to `timeout` even in the case of an actual timeout

Tick rate and Jiffies

Jiffies: wraparound (2)

- Macros are available in `include/linux/jiffies.h` to handle `jiffies` wraparound:

```
#define time_after(a,b)
#define time_before(a,b)
#define time_after_eq(a,b)
#define time_before_eq(a,b)
```

```
unsigned long timeout = jiffies + HZ/2; /* timeout in .5 seconds */
/* ... */
if (time_before(jiffies, timeout)) {
  /* we did not time out */
} else {
  /* timeout, error */
}
```

Tick rate and Jiffies

Userspace and HZ

- Values in ticks can be sent to userspace
  - Some applications grew to rely on a hard-coded value of HZ to convert in seconds
    - The fact that HZ can change caused some malfunction
- The kernel defines a constant value for the tick rate viewed from userspace: USER_HZ
  - For example it is 100 for x86
- In order to export a value in ticks (kernel space) to userspace, conversion is needed:

```
clock_t jiffies_to_clock(unsigned long x);
clock_t jiffies_64_to_clock_t(u64 x);
```

Virginia Tech

Outline

hardware clocks and timers

RTC and the system timer

- **System timer**
  - Programmable hardware timer sending an interrupt at regular intervals
    - Programmed at boot time by the kernel to send an interrupt at `HZ` frequency
  - Other time sources on x86:
    - CPU timestamp counter (TSC) incremented every CPU clock cycle (read through `RDTSC`)
    - Local APIC (intrerrupt controller) timer
- **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

hardware clocks and timers

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
  - Generally performs those steps:
    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)
- **Architecture independent part**: `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()`

hardware clocks and timers

Timer interrupt processing: `tick_periodic()`, `do_timer`

- `kernel/time/tick-common.c`:

```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);
}
```

- `kernel//time/timekeeping.c`:

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

hardware clocks and timers

Timer interrupt processing: `update_process_times()`

- **`update_process_times()`** in `kernel/timer/timer.c`

1. 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
2. Call `run_local_timers()` and run expired timers
   - Raise a softirq
3. 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)

Outline

Timers

Presentation

- **Timers** == **dynamic timers** == **kernel timers**
  - Used to **delay the execution of some piece of code *for a given amount of time***
    - Contrary to bottom-halves that are deferring work in a "just not now" fashion

- **`struct timer_list`** in `includes/linux/timer.h`
- `entry`: linked list of timers
- `expires`: timer expiration date in jiffies
- `function`: handler

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

- `data`: handler parameters

- `flags`: `TIMER_IRQSAFE` (executed with interrupts disabled), `TIMER_DEFERRABLE` (does not wake up an idle CPU [1])

Timers
Using timers

- **Declaring**, **initializing** and **activating** a timer:

```c
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:

```c
mod_timer(&my_timer, jiffies + another_delay);
```

Timers

Using timers (2)

- **Deactivate a timer** prior to its expiration:

```
del_timer(&my_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
    - Solution: `del_timer_sync()`

```
del_timer_sync(&my_timer);
```

    - 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

Timers

Race conditions

- Timers run **asynchronously** with the currently running code
  - They run in softirq context
  - Several potential race conditions exist
- Do not directly modify the `expire` field as a substitution for `mod_timer()`:

```
/* unsafe on SMP: */
del_timer(&my_timer);
my_timer->expires = jiffies + new_delay;
add_timer(&my_timer);
```

- Use `del_timer_sync()` rather than `del_timer()`
- Protect data shared by the handler and other entities

Timers

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()`
  - 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**

Timers

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);

  /* 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);
```

Outline

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 small, sometimes inferior to the timer tick period
  - Several solutions:
    1. **Busy looping**
    2. **Small delays**
    3. **`schedule_timeout()`**

Delaying execution
Busy looping

- **Busy looping**: spin on a loop until a given amount of ticks has elapsed

```
unsigned long timeout = jiffies + 10; /* timeout in 10 ticks */

while(time_before(jiffies, timeout)); /* spin until now > timeout */
```

- Can use `HZ` to specify a delay in seconds:

```
unsigned long delay = jiffies + 2*HZ; /* 2 seconds */

while(time_before(jiffies, timeout));
```

- Amount of time to wait must be a multiple of the timer period
- This technique is generally sub-optimal as the waiting process monopolizes the CPU

Delaying execution

Busy looping (2)

- A better solution is to leave the CPU while waiting:

```
unsigned long delay = jiffies + 2*HZ;

while(time_before(jiffies, delay))
  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**

Delaying execution

Small delays and BogoMIPS

- What if one wants to sleep for a **time inferior to the system timer period**?
  - `HZ` is 100 → period is 10ms
  - `HZ` is 1000 → period is 1ms
  - `include/linux/delay.h`:

```
void mdelay(unsigned long msecs);
void udelay(unsigned long usecs);
void ndelay(unsigned long nsecs);
```

- Implemented as a busy loop
  - 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`
- `udelay/ndelay` should only be called for delays <1ms
  - Risk of overflow

Delaying execution

schedule_timeout()

- **schedule_timeout()** put the calling task to sleep for *at least n ticks*
  - Usage:

```
set_current_state(TASK_INTERRUPTIBLE); /* can also use TASK_UNINTERRUPTIBLE */

schedule_timeout(2 * HZ); /* go to sleep for at least 2 seconds */
```

- Calling task must be in TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE otherwise calling schedule_timeout() has no effect
- schedule_timeout() should be called:
  1. From process context
  2. Without any lock held

Delaying execution

schedule_timeout(): implementation

```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;
```

```c
  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()`

Delaying execution

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()`

Outline

Time of day

struct timespec and ktime_t

- Linux provides plenty of function to get / set the time of the day
- Several data structures to represent a given point in time
  - Two important ones are struct timespec and ktime_t

- uapi/linux/time.h:

```
struct timespec {
  __kernel_time tv_sec;  /* seconds */
  long          tv_nsec; /* nanoseconds */
  /* __kernel_time_t is long on x86_64 */
}
```

- include/linux/time64.h:

```
#define timespec64 timespec
```

- include/linux/ktime.h:

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

typedef union ktime ktime_t;
```

Time of day

API usage examples

```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 */
```

Time of day
API usage examples (2)

```c
/* another interface using ktime_t (
    number of nsec since boot) */
  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;
}
```

```c
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");
```

```makefile
obj-m += timeofday.o

all:
  make -C /lib/modules/$(shell uname -r)/
      build M=$(PWD) modules

test: all
  sudo rmmod timeofday.ko &> /dev/null ||
      true
  sudo insmod timeofday.ko
  sudo rmmod timeofday.ko

clean:
  make -C /lib/modules/$(shell uname -r)/
      build M=$(PWD) clean
```

Tech

Bibliography I

[1] CORBET, J.
Deferrable timers.
https://lwn.net/Articles/228143/.
Accessed: 2017-03-14.