Linux操作系统分析
Chapter 7 Linux中的 时钟和定时测量

陈香兰（xlanchen@ustc.edu.cn）

计算机应用教研室@计算机学院
嵌入式系统实验室@苏州研究院
中国科学技术大学
Fall 2014

October 28, 2014

Outline

定时测量

- Linux内核提供两种主要的定时测量
  1. 获得当前的时间和日期
     - 系统调用：time(), ftime()以及gettimeofday()
  2. 维持定时器
     - settimer(), alarm()
- 定时测量是由基于固定频率振荡器和计数器的几个硬件电路完成的

Outline

硬时钟概述

- 80x86体系结构上，内核必须显式的与各种时钟打交道
  1. 实时时钟（Real time clock，RTC）
  2. 时间戳计数器（Time stamp counter，TSC）
  3. 可编程间隔定时器（Programmable interval timer, PIT）
  4. CPU本地定时器
  5. 高精度事件定时器
  6. ACPI电源管理定时器

Outline

1、实时时钟RTC

- 基本上所有的PC都包含实时时钟RTC
- RTC独立于CPU与所有其他芯片，依靠一个独立的小电池供电给RTC中的振荡器
  - 即使关闭PC电源，还会继续运转
- RTC与CMOS RAM往往集成在一个芯片内
  - 例如：Motorola 146818
- RTC能在IRQ8上发出周期性的中断，频率在2HZ~8192之间
  - 可以对其编程实现一个闹钟
- 内核通过0x70和0x71两个端口访问RTC
- Linux本身只使用RTC获得时间和日期

1、实时时钟RTC

- 可以通过设备文件/dev/rtc对其编程

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <linux/rtc.h>
#include <sys/ioctl.h>
int main(void){
    int fd;
    struct rtc_time rtc_time;
    if ((fd=open(" /dev/rtc" ,O_RDONLY))>0){
        if(ioctl(fd,RTC_RD_TIME,&rtc_time)!=-1)
            printf(" CURRENT TIME(H:M:S): %d:%d:%d\n" ,
                rtc_time.tm_hour,
                rtc_time.tm_min,
                rtc_time.tm_sec);
        else { perror(" IOCTL error\n" ); exit(-1); }
    } else { perror(" OPEN failed\n" ); exit(-1); }
}
```

- 系统管理员可以通过执行时钟程序设置时钟

Outline

2、时间戳计数器TSC

- 在80x86微处理器中，有一个CLK输入引线
  - 接收外部振荡器的时钟信号
- 从pentium开始，很多80x86微处理器都引入了一个TSC
  - 一个64位的、用作时间戳计数器的寄存器
  - 它在每个时钟信号（CLK）到来时+1，例如时钟频率400MHz的微处理器，TSC每2.5ns就+1
  - rdtsc指令用于读该寄存器
- 与后面介绍的可编程间隔定时器相比，TSC可以获得更精确的时钟
  - 为此，Linux在系统初始化的时候必须确定时钟信号CLK的频率（即CPU的实际频率）
  - tsc_calibrate
    - 根据在一个相对较长的时间间隔内（约5ms）所发生的TSC计数的个数进行计算
    - 那个间隔由可编程间隔定时器给出
    - 由于只在系统初始化的时候运行一次，因此本程序可以执行较长时间，而不会引起问题

Outline

3、可编程间隔定时器PIT

- 经过适当编程后，可以周期性的给出时钟中断
- 通常是8254 CMOS芯片，使用I/O端口0x40~0x43
- Linux将PIT编程为：
  - 100Hz、1000Hz
  - 通过IRQ0发出时钟中断
  - 每若干毫秒（100Hz为10ms）产生一次时钟中断，即一个tick

3、可编程间隔定时器PIT

- Tick的长短
  - 短
    - 优点：分辨率高
    - 缺点：需要较多的CPU时间处理，会导致用户程序运行变慢
    - 适用于非常强大的机器，这种机器能够承担较大的系统开销
  - Tick的设置是一个折中，例如
    - 在大多数惠普的Alpha和Intel的IA-64上约1ms产生一个tick（每秒1024个时钟中断）
    - Rawhide Alpha工作站采用更高（1200tick/秒）

3、可编程间隔定时器PIT

- 在Linux中，下列宏决定时钟中断频率

```
#ifdef __KERNEL__
# define HZ CONFIG_HZ /* Internal kernel timer frequency */
# define USER_HZ 100 /* some user interfaces are */
# define CLOCKS_PER_SEC (USER_HZ) /* in "ticks" like times() */
#endif

#ifndef HZ
#define HZ 100
#endif
```

HZ：每秒钟时钟中断的次数，
即每秒tick（滴答）数。
参见include/asm-x86/param.h

```
# CONFIG_HZ_100 is not set
CONFIG_HZ_250=y
# CONFIG_HZ_300 is not set
# CONFIG_HZ_1000 is not set
CONFIG_HZ=250
```

CONFIG_HZ是HZ的配置情况。
例如arch/x86/configs/i386_defconfig中

3、可编程间隔定时器PIT

- 时钟中断频率在init_pit_timer()中初始化，参见arch/x86/kernel/i8253.c

```
    case CLOCK_EVT_MODE_PERIODIC:
        /* binary, mode 2, LSB/MSB, ch 0 */
        outb_pit(0x34, PIT_MODE);
        outb_pit(LATCH & 0xff , PIT_CH0); /* LSB */
        outb_pit(LATCH >> 8 , PIT_CH0); /* MSB */
        break;
```

LATCH是触发周期性时钟的锁存值，参见include/linux/jiffies.h

```
/* LATCH is used in the interval timer and ftape setup. */
#define LATCH ((CLOCK_TICK_RATE + HZ/2) / HZ) /* For divider */
```

CLOCK_TICK_RATE是时钟的振荡频率，参见include/asm-x86/timex.h

```
#ifdef CONFIG_X86_ELAN
#    define PIT_TICK_RATE 1189200 /* AMD Elan has different frequency! */
#elif   defined(CONFIG_X86_RDC321X)
#    define PIT_TICK_RATE 1041667 /* Underlying HZ for R8610 */
#else
#    define PIT_TICK_RATE 1193182 /* Underlying HZ */
#endif

#define CLOCK_TICK_RATE PIT_TICK_RATE
```

因此，当HZ=100时，大约每10ms产生一次时钟中断。

Outline

Linux的计时体系结构

- Linux的计时体系结构
  - 更新自系统启动以来所经过的时间
  - 更新时间和日期
  - 确定当前进程的执行时间，考虑是否要抢占
  - 更新资源使用统计计数
  - 检查到期的软定时器
- 在单处理器系统中，所有定时活动都由IRQ0上的时钟中断触发，包括
  - 在中断中立即执行的部分，和
  - 作为下半部分延迟执行的部分

计时体系结构中的关键数据结构和变量

1. 滴答产生机制：时钟中断→tick
   - tick_device机制和clockevents机制
2. Jiffies变量
3. 计时时钟源
4. Xtime变量

Outline

tick_device机制

- 每一个逻辑CPU有一个tick_cpu_device，
  参见 kernel/time/tick-common.c

```
/*
 * Tick devices
 */
DEFINE_PER_CPU(struct tick_device, tick_cpu_device);
/*
 * Tick next event: keeps track of the tick time
 */
ktime_t tick_next_period;
ktime_t tick_period;
int tick_do_timer_cpu __read_mostly = -1;
DEFINE_SPINLOCK(tick_device_lock);
```

```
struct tick_device {
    struct clock_event_device *evtdev;
    enum tick_device_mode mode;
};
```

- tick_setup_device()中初始化evtdev为指定的clock_event_device
- tick_shutdown()中evtdev被设置为NULL

tick_device机制

- tick_device机制提供tick_notifier()来接收来自clockevents机制的事件

tick_notify()在某个clockevent发生时，会被调用来处理该事件

```
/*
 * Notification about clock event devices
 */
static int tick_notify(struct notifier_block *nb, unsigned long reason, void *dev) {
    ...
    case CLOCK_EVT_NOTIFY_ADD:
        return tick_check_new_device(dev);
    ...
}
```

- 例如，添加一个clockevents设备时，要调用tick_check_new_device()来确定是否要更换tick_device的evtdev

tick_device机制

- 在start_kernel()中调用tick_init()将tick_notifier注册到clockevents_chain上，以便于其接受各种clockevents并做出相关处理。

```
static struct notifier_block tick_notifier = {
    .notifier_call = tick_notify,
};

/**
 * tick_init - initialize the tick control
 *
 * Register the notifier with the clockevents framework
 */
void __init tick_init(void) {
    clockevents_register_notifier(&tick_notifier);
}
```

tick_device机制提供的周期性滴答处理函数

- 参见 kernel/time/tick-common.c

```
/*
 * Event handler for periodic ticks
 */
void tick_handle_periodic(struct clock_event_device *dev) {
    int cpu = smp_processor_id();
    ktime_t next;

    tick_periodic(cpu);
    ...
}
```

tick_device机制提供的周期性滴答处理函数

- 参见 kernel/time/tick-common.c

```
/*
 * Periodic tick
 */
static void tick_periodic(int cpu) {
    if (tick_do_timer_cpu == cpu) {
        write_seqlock(&xtime_lock);

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

        do_timer(1);
        write_sequnlock(&xtime_lock);
    }

    update_process_times(user_mode(get_irq_regs()));
    profile_tick(CPU_PROFILING);
}
```

clockevents设备

- 全局变量clockevent_devices将注册的所有时钟事件设备组织在一起。

```
/* The registered clock event devices */
static LIST_HEAD(clockevent_devices);
static LIST_HEAD(clockevents_released);
```

- clockevent设备的注册函数是clockevents_register_device
  1. 将一个新的有效的clockevent设备加入到clockevent_devices链表中
  2. 使用clockevent通知函数通知CLOCK_EVT_NOTIFY_ADD事件发生（导致tick_notify被调用以处理该事件：tick_check_new_device()被用来判断是否采用新注册的设备，若是，则tick_setup_device()被调用，它进一步调用tick_setup_periodic()，使得clockevent设备的event_handler被初始化，例如可能是tick_handle_periodic()）
  3. 调用clockevents_notify_released对旧设备进行后续处理

clockevent链

- 全局变量clockevents_chain用来组织对clockevents感兴趣的nofifier，如tick_notifier

```
/* Notification for clock events */
static RAW_NOTIFIER_HEAD(clockevents_chain);
```

- 在include/linux/clockchips.h中，定义了clockevent的种类

```
/* Clock event notification values */
enum clock_event_nofitiers {
    CLOCK_EVT_NOTIFY_ADD,
    CLOCK_EVT_NOTIFY_BROADCAST_ON,
    CLOCK_EVT_NOTIFY_BROADCAST_OFF,
    CLOCK_EVT_NOTIFY_BROADCAST_FORCE,
    CLOCK_EVT_NOTIFY_BROADCAST_ENTER,
    CLOCK_EVT_NOTIFY_BROADCAST_EXIT,
    CLOCK_EVT_NOTIFY_SUSPEND,
    CLOCK_EVT_NOTIFY_RESUME,
    CLOCK_EVT_NOTIFY_CPU_DEAD,
};
```

基于PIT的clockevent设备：pit_clockevent

arch/x86/kernel/i8253.c

```
static struct clock_event_device pit_clockevent = {
    .name = "pit",
    .features = CLOCK_EVT_FEAT_PERIODIC | CLOCK_EVT_FEAT_ONESHOT,
    .set_mode = init_pit_timer,
    .set_next_event = pit_next_event,
    .shift = 32, .irq = 0,
};
```

- pit_clockevent在setup_pit_timer中被注册为clockevents设备，并且使用全局变量global_clock_event指向它。

```
void __init setup_pit_timer(void) {
    ...
    pit_clockevent.cpumask = cpumask_of_cpu(smp_processor_id());
    pit_clockevent.mult = div_sc(CLOCK_TICK_RATE, NSEC_PER_SEC, pit_clockevent.shift);
    pit_clockevent.max_delta_ns = clockevent_delta2ns(0x7FFF, &pit_clockevent);
    pit_clockevent.min_delta_ns = clockevent_delta2ns(0xF, &pit_clockevent);
    clockevents_register_device(&pit_clockevent);
    global_clock_event = &pit_clockevent;
}
```

Outline

Jiffies变量

- Jiffies变量用来记录系统自启动以来系统产生的tick数，每次时钟中断+1。其定义方式如下：

jiffies_64在kernel/timer.c中定义：

```
u64 jiffies_64 __cacheline_aligned_in_smp = INITIAL_JIFFIES;
EXPORT_SYMBOL(jiffies_64);
```

在include/linux/jiffies.h中

```
...
#define __jiffy_data __attribute__((section(" .data" )))
...
extern u64 __jiffy_data jiffies_64;
extern unsigned long volatile __jiffy_data jiffies;
...
/*
 * Have the 32 bit jiffies value wrap 5 minutes after boot
 * so jiffies wrap bugs show up earlier.
 */
#define INITIAL_JIFFIES ((unsigned long)(unsigned int) (-300*HZ))
```

Jiffies变量

- jiffies_64和32位的jiffies的关系：

jiffies在arch/x86/kernel/vmlinux_32.lds.S中定义

```
...
OUTPUT_FORMAT("elf32-i386","elf32-i386","elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(phys_startup_32)
jiffies = jiffies_64;
...
```

在vmlinux的符号表中，可以看到这两个变量在同一个地址上

```
c0599c00 D jiffies
c0599c00 D jiffies_64
```

Jiffies变量

- jiffies变量的更新函数

kernel/timer.c

```
/*
 * The 64-bit jiffies value is not atomic - you MUST NOT read it
 * without sampling the sequence number in xtime_lock.
 * jiffies is defined in the linker script...
 */
void do_timer(unsigned long ticks) {
     jiffies_64 += ticks;
     update_times(ticks);
}
```

Outline

时钟源机制

- 时钟源抽象
  - 是系统时钟源，定义了系统时钟源的接口
  - 参见数据结构include/linux/clocksource.h::clocksource
- 时钟源列表clocksource_list：
  - 按照各自的rating值由高到低排序
  - 时钟源注册/注销函数：
    clocksource_register()/clocksource_unregister()：
    - 将指定的时钟源插入到时钟源列表中，或者从中移除。
- 缺省时钟源：具有最低rating值（=1）的Jiffies时钟源（clocksource_jiffies）
- 当前时钟源指针curr_clocksource指向当前所用的时钟源。最开始使用缺省时钟源。
- 时钟源的更新时机：
  kernel/time/timekeeping.c::update_wall_time结尾处。

时钟源机制

在kernel/time/clocksource.c中：

```
/* XXX - Would like a better way for initializing curr_clocksource */
extern struct clocksource clocksource_jiffies;
/*[Clocksource internal variables]———
 * curr_clocksource: ...
 * next_clocksource:
 * pending next selected clocksource.
 * clocksource_list:
 * linked list with the registered clocksources
 * clocksource_lock: ...
 * override_name:
 * Name of the user-specified clocksource.
 */
static struct clocksource *curr_clocksource = &clocksource_jiffies;
static struct clocksource *next_clocksource;
static struct clocksource *clocksource_override;
static LIST_HEAD(clocksource_list);
static DEFINE_SPINLOCK(clocksource_lock);
static char override_name[32];
static int finished_booting;
```

缺省时钟源：jiffies时钟源

- 参见kernel/time/jiffies.c

```
static cycle_t jiffies_read(void) {
    return (cycle_t) jiffies;
}

struct clocksource clocksource_jiffies = {
    .name = "jiffies",
    .rating = 1, /* lowest valid rating*/
    .read = jiffies_read,
    .mask = 0xffffffff, /*32bits*/
    .mult = NSEC_PER_JIFFY << JIFFIES_SHIFT, /* details above */
    .shift = JIFFIES_SHIFT,
};

static int __init init_jiffies_clocksource(void) {
    return clocksource_register(&clocksource_jiffies);
}
core_initcall(init_jiffies_clocksource);
```

PIT时钟源

```
static struct clocksource clocksource_pit = {
    .name = ” pit” ,
    .rating = 110,
    .read = pit_read,
    .mask = CLOCKSOURCE_MASK(32),
    .mult = 0,
    .shift = 20,
};
...
static int __init init_pit_clocksource(void) {
     /*
      * Several reasons not to register PIT as a clocksource:
      *
      * - On SMP PIT does not scale due to i8253_lock
      * - when HPET is enabled
      * - when local APIC timer is active (PIT is switched off)
      */
    if (num_possible_cpus() > 1 || is_hpet_enabled() ||
        pit_clockevent.mode != CLOCK_EVT_MODE_PERIODIC)
        return 0;
    clocksource_pit.mult = clocksource_hz2mult(CLOCK_TICK_RATE,
                clocksource_pit.shift);
    return clocksource_register(&clocksource_pit);
}
arch_initcall(init_pit_clocksource);
```

Outline

xtime变量

- xtime：存放当前时间和日期，在kernel/time/timekeeping.c中定义

```
/*
 * The current time
 * wall_to_monotonic is what we need to add to xtime (or xtime corrected
 * for sub jiffie times) to get to monotonic time. Monotonic is pegged
 * at zero at system boot time, so wall_to_monotonic will be negative,
 * however, we will ALWAYS keep the tv_nsec part positive so we can use
 * the usual normalization.
 *
 * wall_to_monotonic is moved after resume from suspend for the monotonic
 * time not to jump. We need to add total_sleep_time to wall_to_monotonic
 * to get the real boot based time offset.
 *
 * - wall_to_monotonic is no longer the boot time, getboottime must be
 * used instead.
 */
struct timespec xtime __attribute__ ((aligned (16)));
struct timespec wall_to_monotonic __attribute__ ((aligned (16)));
static unsigned long total_sleep_time; /* seconds */
static struct timespec xtime_cache __attribute__ ((aligned (16)));
```

xtime变量

- xtime使用数据结构timespec

timespec@include/linux/time.h

```
#ifndef _STRUCT_TIMESPEC
#define _STRUCT_TIMESPEC
struct timespec {
    time_t tv_sec; /* seconds */
    long tv_nsec; /* nanoseconds */
};
#endif
```

  - 时间纪元（Epoch）：
    1970-01-01 00:00:00 +0000 午夜（UTC).

xtime变量

- Xtime的更新
  - 基本上每个tick更新一次
  - 参见：update_wall_time@kernel/time/timekeeping.c
    - 根据时钟源来更新xtime的秒数和纳秒数
    - 时钟源

Outline

时钟中断处理函数timer_interrupt

1. timer_interrupt() @ arch/x86/kernel/time_32.c

```
/*
 * This is the same as the above, except we _also_ save the current
 * Time Stamp Counter value at the time of the timer interrupt, so that
 * we later on can estimate the time of day more exactly.
 */
irqreturn_t timer_interrupt(int irq, void *dev_id) {
    /* Keep nmi watchdog up to date */
    per_cpu(irq_stat, smp_processor_id()).irq0_irqs++;
    ...
    do_timer_interrupt_hook();
    ...
    return IRQ_HANDLED;
}
```

时钟中断处理函数timer_interrupt

2. do_timer_interrupt_hook() @ include/asm-x86/mach-default/do_timer.h

```
/**
 * do_timer_interrupt_hook - hook into timer tick
 *
 * Call the pit clock event handler. see asm/i8253.h
 **/
static inline void do_timer_interrupt_hook(void) {
    global_clock_event->event_handler(global_clock_event);
}
```

- global_clock_event??
  在setup_pit_timer中被初始化为pit_clockevent

时钟中断处理函数timer_interrupt

3. setup_pit_timer() @ arch/x86/kernel/i8253.c

```
/*
 * Initialize the conversion factor and the min/max deltas of the clock event
 * structure and register the clock event source with the framework.
 */
void __init setup_pit_timer(void) {
    /*
     * Start pit with the boot cpu mask and make it global after the
     * IO_APIC has been initialized.
     */
    pit_clockevent.cpumask = cpumask_of_cpu(smp_processor_id());
    pit_clockevent.mult = div_sc(CLOCK_TICK_RATE, NSEC_PER_SEC, pit_clockevent.shift);
    pit_clockevent.max_delta_ns = clockevent_delta2ns(0x7FFF, &pit_clockevent);
    pit_clockevent.min_delta_ns = clockevent_delta2ns(0xF, &pit_clockevent);
    clockevents_register_device(&pit_clockevent);
    global_clock_event = &pit_clockevent;
}
```

- event_handler？？(例如tick_handle_periodic)

时钟中断处理函数timer_interrupt

4. do_timer() @ kernel/timer.c
   1. 更新jiffies
   2. 更新xtime

```
/*
 * Called by the timer interrupt. xtime_lock must already be taken
 * by the timer IRQ!
 */
static inline void update_times(unsigned long ticks) {
    update_wall_time();
    calc_load(ticks);
}

/*
 * The 64-bit jiffies value is not atomic - you MUST NOT read it
 * without sampling the sequence number in xtime_lock.
 * jiffies is defined in the linker script...
 */
void do_timer(unsigned long ticks) {
     jiffies_64 += ticks;
     update_times(ticks);
}
```

时钟中断处理函数timer_interrupt

5. update_process_times() @ kernel/timer.c
   1. 更新软定时器
   2. 调用调度器的tick函数：scheduler_tick()

```
/*
 * Called from the timer interrupt handler to charge one tick to the current
 * process.  user_tick is 1 if the tick is user time, 0 for system.
 */
void update_process_times(int user_tick) {
    struct task_struct *p = current;
    int cpu = smp_processor_id();
    /* Note: this timer irq context must be accounted for as well. */
    account_process_tick(p, user_tick);
    run_local_timers();
    if (rcu_pending(cpu))
        rcu_check_callbacks(cpu, user_tick);
    scheduler_tick();
    run_posix_cpu_timers(p);
}
```

时钟中断处理函数timer_interrupt

6. scheduler_tick() @ kernel/sched.c
   - 在scheduler_tick中，当前进程调用所属调度类的task_tick函数

```
/*
 * This function gets called by the timer code, with HZ frequency.
 * We call it with interrupts disabled.
 *
 * It also gets called by the fork code, when changing the parent's
 * timeslices.
 */
void scheduler_tick(void) {
    int cpu = smp_processor_id();
    struct rq *rq = cpu_rq(cpu);
    struct task_struct *curr = rq->curr;

    sched_clock_tick();

    spin_lock(&rq->lock);
    update_rq_clock(rq);
    update_cpu_load(rq);
    curr->sched_class->task_tick(rq, curr, 0);
    spin_unlock(&rq->lock);
    ...
}
```

Outline

软定时器

- 定时器是一种软件功能，它允许在将来的某个时刻调用某个函数
- 大多数设备驱动程序利用定时器完成一些特殊工作
  - 软盘驱动程序在软盘暂时不被访问时就关闭设备的发动机
  - 并行打印机利用定时器检测错误的打印机情况
- Linux中存在两类定时器：
  - 动态定时器：内核使用
  - 间隔定时器：由进程在用户态创建
  - 注意：
    由于软定时器在下半部分处理，内核不能保证定时器正好在时钟到期的时候被执行，会存在延迟，不适用于实时应用

动态定时器

- 动态定时器被动态的创建和撤销，当前活动的动态定时器个数没有限制
- 一个定时器由一个timer_list数据结构来定义，参见include/linux/timer.h

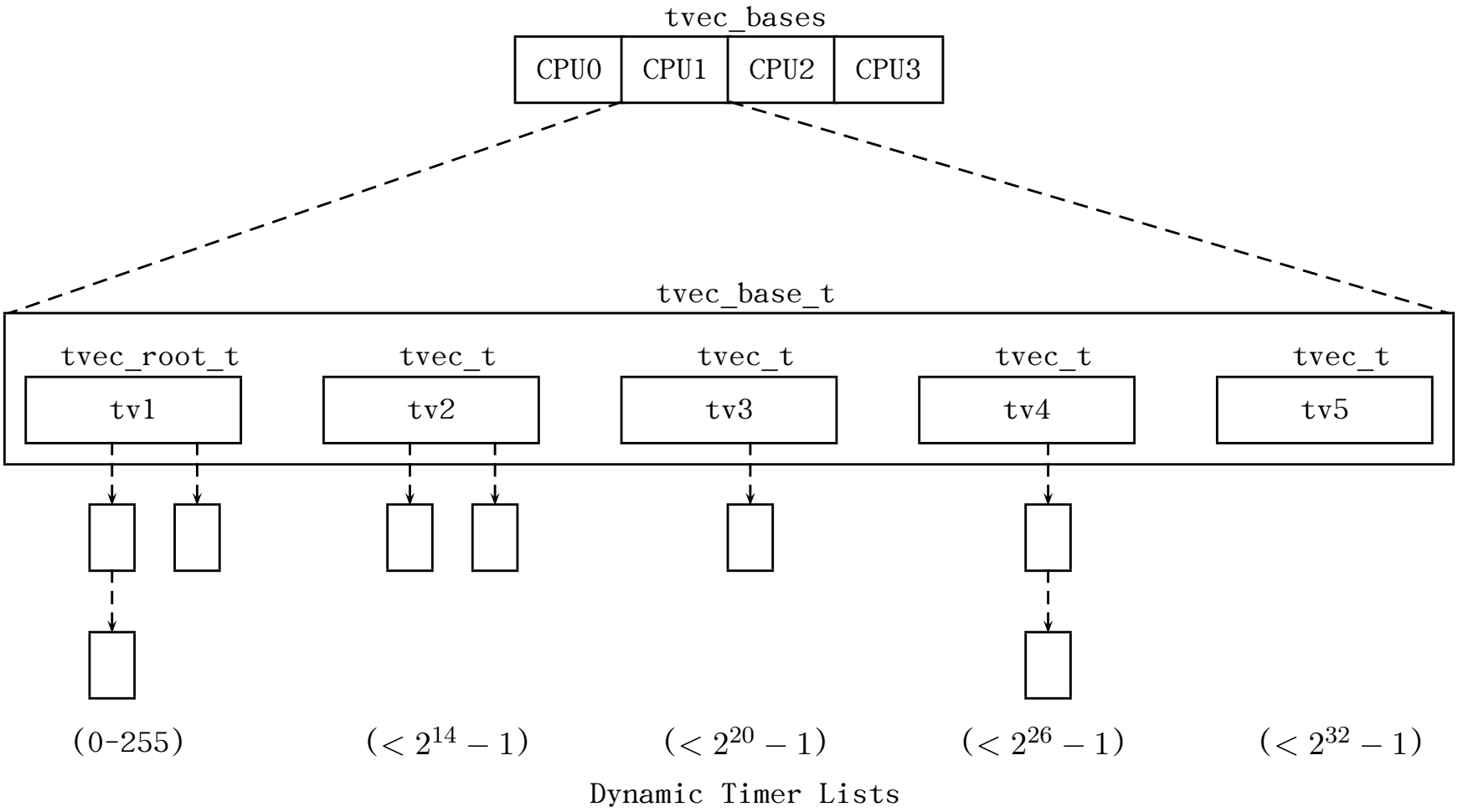
```
struct timer_list {
    struct list_head entry;
    unsigned long expires;

    void (*function)(unsigned long);
    unsigned long data;
    struct tvec_base *base;
    ...
};
```

创建并激活一个动态定时器

1. 创建一个新的timer_list对象
2. 调用init_timer初始化，并设置定时器要处理的函数和参数
3. 设置定时时间
4. 使用add_timer加入到合适的链表中

- 通常定时器只能执行一次，如果要周期性的执行必须再次将其加入链表

动态定时器的维护



Dynamic Timer Lists

维护用的数据结构

kernel/timer.c

```
/* * per-CPU timer vector definitions: */
#define TVN_BITS (CONFIG_BASE_SMALL ? 4 : 6)
#define TVR_BITS (CONFIG_BASE_SMALL ? 6 : 8)
#define TVN_SIZE (1 << TVN_BITS)
#define TVR_SIZE (1 << TVR_BITS)
#define TVN_MASK (TVN_SIZE - 1)
#define TVR_MASK (TVR_SIZE - 1)
```

```
Sstruct tvec {
    struct list_head vec[TVN_SIZE];
};

struct tvec_root {
    struct list_head vec[TVR_SIZE];
};

struct tvec_base {
    spinlock_t lock;
    struct timer_list *running_timer;
    unsigned long timer_jiffies;
    struct tvec_root tv1;
    struct tvec tv2;
    struct tvec tv3;
    struct tvec tv4;
    struct tvec tv5;
} ____cacheline_aligned;
```

```
struct tvec_base boot_tvec_bases;
EXPORT_SYMBOL(boot_tvec_bases); static
DEFINE_PER_CPU(struct tvec_base *, tvec_bases) = &boot_tvec_bases;
```

动态定时器的处理

- run_local_timers() @ kernel/timer.c在时钟中断处理过程中被update_process_times() @ kernel/timer.c调用

```
/*
 * Called by the local, per-CPU timer interrupt on SMP.
 */
void run_local_timers(void) {
    hrtimer_run_queues();
    raise_softirq(TIMER_SOFTIRQ);
    softlockup_tick();
}
```

- 软中断TIMER_SOFTIRQ对应的处理函数？？

init_timers()@kernel/timer.c

```
void __init init_timers(void) {
    ...
    open_softirq(TIMER_SOFTIRQ, run_timer_softirq, NULL);
}
```

- 阅读run_timer_softirq()@kernel/timer.c

动态定时器应用之delayed work

kernel/workqueue.c

```
int queue_delayed_work_on(int cpu, struct workqueue_struct *wq,
                struct delayed_work *dwork, unsigned long delay)
{
    int ret = 0;
    struct timer_list *timer = &dwork->timer;
    struct work_struct *work = &dwork->work;

    if (!test_and_set_bit(WORK_STRUCT_PENDING, work_data_bits(work))) {
        BUG_ON(timer_pending(timer));
        BUG_ON(!list_empty(&work->entry));
        timer_stats_timer_set_start_info(&dwork->timer);
        /* This stores cwq for the moment, for the timer_fn */
        set_wq_data(work, wq_per_cpu(wq, raw_smp_processor_id()));
        timer->expires = jiffies + delay;
        timer->data = (unsigned long)dwork;
        timer->function = delayed_work_timer_fn;
        if (unlikely(cpu >= 0))
            add_timer_on(timer, cpu);
        else
            add_timer(timer);
        ret = 1;
    }
    return ret;
}
```

动态定时器应用之schedule_timeout

kernel/timer.c

```
signed long __sched schedule_timeout(signed long timeout) {
    ...
    expire = timeout + jiffies;

    setup_timer_on_stack(&timer, process_timeout, (unsigned long)current);
    __mod_timer(&timer, expire);
    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;
}
```

Outline

延迟函数

- udelay(n), ndelay(n) @ include/asm-x86/delay.h

```
/* 0x10c7 is 2**32 / 1000000 (rounded up) */
#define udelay(n) (__builtin_constant_p(n) ? \
    ((n) > 20000 ? __bad_udelay() : __const_udelay((n) * 0x10c7ul)) : \
    __udelay(n))

/* 0x5 is 2**32 / 1000000000 (rounded up) */
#define ndelay(n) (__builtin_constant_p(n) ? \
    ((n) > 20000 ? __bad_ndelay() : __const_udelay((n) * 5ul)) : \
    __ndelay(n))
```

- 进一步查看arch/x86/lib/delay_32.c
- delay_fn可以是：（arch/x86/lib/delay_32.c）
  1. delay_loop：simple loop based delay
  2. delay_tsc：TSC based delay

Outline

与时钟和定时测量相关的API

- time() - get time in seconds
  - 返回从1970年1月1日凌晨0点开始的秒数

```
time_t time(time_t *t);
```

- ftime() - return date and time
  - 返回从1970年1月1日凌晨0点开始的秒数以及最后一秒的毫秒数

```
int ftime(struct timeb *tp);
```

```
struct timeb {
    time_t time;
    unsigned short millitm;
    short timezone;
    short dstflag;
};
```

与时钟和定时测量相关的API

- gettimeofday() , settimeofday() - get and set the time
  - 前者返回从1970年1月1日凌晨0点开始的秒数
  - 对应于sys_gettimeofday()

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(const struct timeval *tv, const struct
timezone *tz);
```

```
struct timeval {
    time_t tv_sec; /* seconds */
    suseconds_t tv_usec; /* microseconds */
};
struct timezone {
    int tz_minuteswest; /* minutes west of Greenwich */
    int tz_dsttime; /* type of DST correction */
};
```

与时钟和定时测量相关的API

- getitimer(), setitimer() - get or set value of an interval timer
  - 每个进程有三个间隔定时器：
    1. ITIMER_REAL：real time
    2. ITIMER_VIRTUAL：user space time
    3. ITIMER_PROF：user + kernel space time
  - 频率：周期性的触发定时器（若为0，只触发一次）

```
int getitimer(int which, struct itimerval *curr_value);
int setitimer(int which, const struct itimerval
*new_value, struct itimerval *old_value);
```

```
struct itimerval {
    struct timeval it_interval; /* next value */
    struct timeval it_value; /* current value */
};
struct timeval {
    time_t tv_sec; /* seconds */
    suseconds_t tv_usec; /* microseconds */
};
```

与时钟和定时测量相关的API

- alarm() - set an alarm clock for delivery of a signal
  - 若干秒后引起SIGALARM信号

unsigned int alarm(unsigned int seconds);

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
static int flag=0;
void sig_alarm(int signo){
    flag=1;
}
int main(void){
    if (signal(SIGALRM, sig_alarm)==SIG_ERR){
        perror(" Can' t set new signal action" );
        exit(1);
    }
    alarm(10);
    pause();

    if(flag) printf(" SIGALRM received and flag changed!\n" );
    return 0;
}
```

与时钟和定时测量相关的API

- asctime, ctime, gmtime, localtime, mktime, asctime_r, ctime_r, gmtime_r, localtime_r - transform date and time to broken-down time or ASCII
  - 改变时钟格式

与时钟和定时测量相关的命令

- date - print or set the system date and time
- time - - run programs and summarize system resource usage

Outline

小结

1. 定时的硬件设备
   - 实时时钟RTC
   - 时间戳计数器TSC
   - 可编程间隔定时器PIT
2. Linux的计时体系结构
   - 滴答产生机制
   - Jiffies变量
   - Linux的时钟源
   - xtime变量
   - 时钟中断处理
   - 软定时器
3. 延迟函数
4. 相关API和命令
5. 小结、作业和project

Project

- 具体要求参见课程主页

作业

1. 名词解释
    1. RTC和PIT
    2. 滴答和jiffies变量
    3. 动态定时器

Thanks !

The end.