Kernel Time System

--wxc200

看完 CFS(complete fair scheduler)后，一直想研究下 scheduler 与 kernel time 之间的关系。内核的时间系统，与 调度器是紧密联系的。同时，它也是非常有意思的。我很乐意把读代码与思考过程中的心得写下来，与大家分享。

在开始前， 我先向大家推荐土狗兄的一篇帖子：
研究下 hrtimer 及内核 clock/timer 子系统变化
(http://www.linuxforum.net/forum/gshowflat.php?Cat=&Board=linuxK&Number=664730&page=0&view=collapsed&sb=5&o=all&fpart=) 他在文中介绍了 kernel hrtimer 机制，写得很不错。他在文末的一些资料链接，希望大家也看看吧。

代码基于 2.6.29-rc8,一些调试 的信息是在 borqs android(PXA chip)平台上做的，kernel version:2.6.25.

我打算从系统初始化时间系统建立开始分析，后面时间系统建立了，会分析些运行机制。

请先留意这几个与 time 相关的宏：
“
```
CONFIG_TICK_ONESHOT=y
CONFIG_NO_HZ=y
CONFIG_HIGH_RES_TIMERS=y
CONFIG_GENERIC_CLOCKEVENTS_BUILD=y
```
“

我们后面在代码分析过程中，会返回来看这几个宏． HIGH_RES_TIMERS 表示要采取高精时间模式，NO_HZ 使系统有条件运行 tickless system.关于后者，请看下面这段话：

按照 ingo 说的（http://kerneltrap.org/node/6750）：
“The tickless kernel feature (CONFIG_NO_HZ) enables 'on-demand' timer
interrupts: if there is no timer to be expired for say 1.5 seconds when
the system goes idle, then the system will stay totally idle for 1.5
seconds. This should bring cooler CPUs and power savings: on our (x86)
testboxes we have measured the effective IRQ rate to go from HZ to 1-2

timer interrupts per second.
”

省电，cool~ borqs 的 phone 就采用了 NO_HZ mode。旧在 tick mechasim kenel system ,每秒中会触发 HZ 个时间中断，用来维持系统的正常运行。在 NOHZ 模式中，系统会 goes idle，这时不会有时间中断了。Kernle will call tick_do_update_jiffies64() to update Jiffies and wall-time when recover from idle,代码是：

do_timer(++ticks)

这个地方 ticks 值分根据 idle 的时间变化，不像以前，在 tick 中断里把 jiffies 加 1.大家可以搜一下 do_timer()的调用，大多数参数是 1.

先被一段许久前写的 hrtimer 分析：
"
先看几个概念：
1）高精度时间(High Resolution Time) ,这是 Ingo Monar 新实现的时间机制，它将时间的精度由原先的 hz，提高到 ns 级（hardware timer chip support）.
2）tick 中断,系统时间更新调度的周期。在中断处理函数里完成跟时间有关的操作。
代码分析：
看几个结构体：
clocksource:硬件时钟，提供读取 free-running 的 counter register。这儿一定要注意，读出来的 counter 寄存器的数值，是 cycle_t 类型的，由外部时钟频率决定。它与 ns 中间有换算。
clock_event_device:对时钟事件描述的设备。有几个回调函数比较重要，set_next_event(设置下次中断事件），event_handler(事件回调函数，把中断事件传给上层处理)，broad_cast(分发事件).
另外其它的 field，感兴趣的朋友可以自己去查看。"

从系统初始化建立时间系统来说。

1.
Init/main.c 里 start_kernel()与时间有关的第一个函数，应该是 tick_init(),it “ initialize the tick control”,its code:

```
void __init tick_init(void)
{
     clockevents_register_notifier(&tick_notifier);
}
```

很简单，它会为 clockevents framework register notifier,关于 notifier 机制，请参考 Linux Notification(http://blog.chinaunix.net/u3/92077/showart_1835817

.html)

tick_notifier 这个 notifier_block 会在后面注册 clock event 时回调，最终会调到 tick_notify,根据不同的 reson 参数,执行不同的处理。在第一次注册时，会传"CLOCK_EVT_NOTIFY_ADD”参数，继而调用 tick_check_new_device(dev) 建立最终的 clock_event_device.

这个 notifier block 调用 clockevents_register_notifier(struct notifier_block *nb)注册到了 clockevents_chain 链表上。在 kernel/time/clockevents.c 中，这个函数下面就是 clockevents_do_notify(),used for “Notify about a clock event change.”。

之前在读这部分代码时，误进入 notifier 层去了，非常深，看着晕啊~ 在这儿只需要明白，后面的 clockevent 注册时，会回调这 clock_events_chain 上的 notifier call 就成了。

2.接下来, kernel 会调用 setup_arch(&command_line)做一些根具体的 machine 相关的指针初始化工作。

以 arch/arm/kernel/setup.c 为例，它会做许多有意思的事情，我们在这儿不关心它 setup_machine or 对 command line 参数的处理，只需要看下面这一行：
system_timer = mdesc->timer;
system_tiemr 是 kernel timer structure,arch/arm/include/mach/time.h has detail about it.我们只关心它在 init 指针的调用，请先看它对 init 的说明：
“init
 *   Initialise the kernels jiffy timer source, claim interrupt
 *   using setup_irq. This is called early on during initialisation
 * while interrupts are still disabled on the local CPU.”

这儿会读不同 machine 的一些参数，在 struct machine_desc 结构体中，保存着所有信息，诸如中断，machine 的 init 指针，当然还有 这个 system_timer.

下面是一个定义 machine 相关信息的宏：
```
MACHINE_START(BENZINA, "Benzina")
     .phys_io        = 0x40000000,
     .boot_params    = 0xa0000100,
     .io_pg_offst    = (io_p2v(0x40000000) >> 18) & 0xfffc,
     .map_io         = pxa_map_io,
     .init_irq       = pxa3xx_init_irq,
     .timer          = &pxa_timer,
     .init_machine   = benzina_init,
```

MACHINE_END

关于 timer 的初始化工作会在后面的 time_init()函数里去做.

3.init_timers()
这个函数简单些，调用 init_timers_cpu() 它会初始化旧的轮子数组。关于旧的时间机制，请参考 ulk 相关部分。
同时，打开了 softirq:
open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
关于 run_timer_softirq 这个函数：
This function runs timers and the timer-tq in bottom half context.
后面在 tick 中断处理函数会打开并调用它。这部分需要了解 softirq 的知识，ulk 上也有。

4 hrtimers_init()
调用 init_hrtimers_cpu()做些初始化工作：
1) get per-cpu variable:hrtimer_cpu_base
2)initialize cpu-base's clock hrtimer_clock_base array clock_base[HRTIMER_MAX_CLOCK_BASES]
3)hrtimer_init_hres() initialize related parts of cpu-base

如果打开了 HIGH—RESOLUTION，会打开 HRTIMER 相应的 softirq:

open_softirq(HRTIMER_SOFTIRQ, run_hrtimer_softirq);
执行函数最终会调用 hrtimer_peek_ahead_timers 它会"run soft-expired timers now”.

这部分只是初始化一些数据结构，比较简单。

5 timekeeping_init()
Initializes the clocksource and common timekeeping values.
这个也简单，获取相应的 clock source,读取硬件时钟的 cycle,给 xtime 及 wall-to-monotic 初始化，这些变量在后面是经常遇到的。

6 time_init()

最终会调到初始化这个 system tiemr,在 mach-pxa 平台下，这个初始化的函数是:
```
pxa_timer_init(){
...
setup_irq(IRQ_OST0, &pxa_ost0_irq);

     clocksource_register(&cksrc_pxa_oscr0);
     clockevents_register_device(&ckevt_pxa_osmr0);
```

```
...
}
```
这个函数做的具体事情，是跟不同的平台 machine 相关的，大家可以结合自己的平台看。也可以 x86 在这部分的工作。
我们分析这个函数中做的三件事情：

1) 建立时间中断。可以参考 include/linux/interrupt.h 看中断的相关信息。这儿为硬件时钟设定了中断的信息，当产生中断时，会跳到 irqaction->handler,这个 handler 很好玩，它除了做一些硬件时钟的中断处理外，最重要的是如下代码：

```
pxa_ost0_interrupt(int irq, void *dev_id)
{
     struct clock_event_device *c = dev_id;
     c->event_handler(c);
}
```
dev_id 是 clock_event_device(clock event device descriptor)，它有两 个很重要的函数指针:

```
int           (*set_next_event)(unsigned long evt,
                                struct clock_event_device *);

     void          (*event_handler)(struct clock_event_device
*);
```
一个是设置下一次时钟事件，一个是对当前的时钟事件中断做处理。
在 machine 相关的配置中，会对 set_next_event 指针预先设置，它一般是写硬件 timer 寄存器设置下一个中断。而 event_handler，刚会根据系统采用的时间配置信息，在后面注册的过程中，最终建立。按照我目前的调试信息，它与 CONFIG_NO_HZ 和 CONFIG_HIGH_RES_TIMERS 有关.最终是这个样子的：
```
1) CONFIG_NO_HZ=y CONFIG_HIGH_RES_TIMERS=n :
event_handler=tick_nohz_handler  → The nohz low res interrupt
handler
2) CONFIG_NO_HZ=y/n CONFIG_HIGH_RES_TIMERS=y :
event_handler=hrtimer_interrupt   -->High resolution timer
interrupt
1) CONFIG_NO_HZ=n CONFIG_HIGH_RES_TIMERS=n :
event_handler=tick_handle_periodic()  →  Event handler for
periodic ticks
```

关于这部分调试的详细信息，大家可以参照这个帖子的第二楼：
(http://linux.chinaunix.net/bbs/thread-1098007-1-2.html)

2) 第二件事。注册 clock_source。clock_source 结构体有详细关于一个硬件时钟的参数配置，hardware abstraction for a free running counter.

这个注册过程会根据不同 clock 的精度入列，并 call select_clocksource()选择合适的 clock_source.由于只有一个硬件 timer,这部分不过多分析了。

3 ） 第三件事，也是最重要的，就是 register a clock event device.clockevents_register_device()会先把当前的 clock_event_device 挂 在 clockevent_devices 链表上，然后调用 clockevents_do_notify(CLOCK_EVT_NOTIFY_ADD, dev);

去回调之前注册的 notifier.

这样，就会跳到 tick_check_new_device(struct clock_event_device *newdev),”Check, if the new registered device should be used.”

7.这儿有个 cpu 变量 tick_device,它有两个 fields,一个指向 clock_event_device 的针，个 tick_device_mode(PERIODIC & ONESHOT), 即周期性的与单次触发。这个函数会对新的 tick_event_device 进行分析，是否替代当前的 event_device,whether local device,then compare the rating of new&current device,select the bigger rating one.then new event device has a chance to replace the curr device.If curr device is broadcast one,just shut it down,donot give it back to clockevents layer again.Now,it's time to setup tick device:

```
tick_setup_device(td, newdev, cpu, cpumask_of(cpu));
```

it's clear,after the checking new event device routine,we need setup the device now.

这个函数其实就是根据 event_device 的不同属性及系统配置，设置相应的 event handler 指针。会判断是否第一次 setup device,系统刚初始化的时候，执行的是旧的 periodic handler 机制。由于是第一次注册 clock-event-device,所以会调用：

```
tick_setup_periodic(newdev, 0);
```

this function will” Setup the device for a periodic tick”,先通过调用：

```
tick_set_periodic_handler(dev, broadcast);
```

来设置 handler 指针，这个函数如下：

```
/*
 * Set the periodic handler depending on broadcast on/off
 */
void tick_set_periodic_handler(struct clock_event_device *dev,
int broadcast)
{
     if (!broadcast)
          dev->event_handler = tick_handle_periodic;
     else
          dev->event_handler = tick_handle_periodic_broadcast;
}
```

传进来的参数 broadcast=0,所以，系统刚启动时，hanlder 指针即 tick_handle_periodic.
来了时钟中断，会跳到它的执行函数里。下面要做的事情，就是驱动系统跑起来：
  1) set hardware clock timer by clockevents_program_event()
  2) go to clock event handler when interrupt comes

再返回到 tick_check_new_device,如果 new event device register as broadcast device,it will fall into out_bc:

```
out_bc:
     /*
      * Can the new device be used as a broadcast device ?
      */
     if (tick_check_broadcast_device(newdev))
         ret = NOTIFY_STOP;
```

broadcast device will be started in tick_broadcast_start_periodic(),then call tick_setup_periodic(bc, 1);// notice ,parameter = 1 now.

这和上面的函数就一致了，因参数变成 1,故最终的 handler 指针变成：
dev->event_handler = tick_handle_periodic_broadcast;

那么，这个 handler 做什么事呢？ 分发事件到相应的 device，即设置相应 clock event device 的 next event.如果这儿是周期模式的，就没必要设置了。“ Setup the next period for devices, which do not have periodic mode.”

这样，tick_check_new_device 的任务就完成了。剩下的事情很简单，等下一个 HZ 中断到了，跳转到 tick_periodic_handle()这个 event handler 里做事情。start_kernel 关于时间部分的初始化工作 ，这些就够了。在下一个中断到来时，会做什么事情呢？

详细分析下不同的 event handler 处理的事情吧。

1. 旧的周期性时间中断，在初始化的时候仍然使用

```
 /*
 * 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); //做些每个 tick 必做的事情，比如更新 process
运行时间

    if (dev->mode != CLOCK_EVT_MODE_ONESHOT)
        return;
    /*
     * Setup the next period for devices, which do not have
     * periodic mode:
     */
    next = ktime_add(dev->next_event, tick_period);  下一个 tick
中断时间
    for (;;) {
        if (!clockevents_program_event(dev, next,
ktime_get()))
            return;
        tick_periodic(cpu);
        next = ktime_add(next, tick_period);
    }   //这个循环尝试写硬件时钟寄存器为下一个 tick-period
}
```

这种旧的 tick 中断机制比较简单，相当于硬件时钟产生周期中断，在中断处理函数里调用 tick_periodic()去做与进程更新相关的事情。

这儿最有意思的，要属 update_process_times() 中调用的 run_local_timers()这个函数了。

2. 该函数代码

```
void run_local_timers(void)
{
    hrtimer_run_queues();//运行过期的 hrtimer
    raise_softirq(TIMER_SOFTIRQ);
    softlockup_tick();
}
```

这儿一定注意，现在采用的仍然是旧的时间中断机制，那么此时的 hrtimer 怎么运行呢？

看这个函数：

```
void hrtimer_run_queues(void)
{
    struct rb_node *node;
    struct hrtimer_cpu_base *cpu_base =
&__get_cpu_var(hrtimer_bases);
    struct hrtimer_clock_base *base;
```

```
     int index, gettime = 1;

     if (hrtimer_hres_active())  //如果高精时间机制active,就返回。在
hrtimer_init()时这个值是0
          return;

//对每个cpu_base上所有的hrtimer clock base检查已经过期的hrtimer
     for (index = 0; index < HRTIMER_MAX_CLOCK_BASES; index++) {
          base = &cpu_base->clock_base[index];

          if (!base->first)
               continue;

          if (gettime) {
               hrtimer_get_softirq_time(cpu_base);//获取当前的时间
               gettime = 0;
          }

          spin_lock(&cpu_base->lock);

          while ((node = base->first)) {
               struct hrtimer *timer;

               timer = rb_entry(node, struct hrtimer, node);
               if (base->softirq_time.tv64 <=
                         hrtimer_get_expires_tv64(timer))//与
hrtimer的过期时间比较，如果比当前时间大，刚没有过期，退出。。。
                    break;

               __run_hrtimer(timer);//运行过期的hrtimer。这个我们后
面再分析。
          }
          spin_unlock(&cpu_base->lock);
     }
}
```

3.这儿会把前面注册的软中断打开 raise_softirq(TIMER_SOFTIRQ)

我们看下这个软中断做的事情：

```
/*
 * This function runs timers and the timer-tq in bottom half
context.
 */
static void run_timer_softirq(struct softirq_action *h)
```

```
{
     struct tvec_base *base = __get_cpu_var(tvec_bases);

     hrtimer_run_pending();//有机会转到 高精度或者 no-hz 时间机制

     if (time_after_eq(jiffies, base->timer_jiffies))
          __run_timers(base); 根据 jiffies 运行过期的普通 timer
}
```

在软中断里尝试转到之前配置的时间模式（高精或 no-hz）,结束目前采用的周期性中断机制。并且运行过期的普通 timers。关于 run_timers()，比较简单，可以参照 ulk 分析。

4 看看如何转到 hrtimer 或 no-hz mode

```
void hrtimer_run_pending(void)
{
     if (hrtimer_hres_active())  //已经采用高精模式就退出
          return;

     /*
      * This _is_ ugly: We have to check in the softirq context,
      * whether we can switch to highres and / or nohz mode. The
      * clocksource switch happens in the timer interrupt with
      * xtime_lock held. Notification from there only sets the
      * check bit in the tick_oneshot code, otherwise we might
      * deadlock vs. xtime_lock.
      */
     if (tick_check_oneshot_change(!hrtimer_is_hres_enabled()))
          hrtimer_switch_to_hres();
//上面两行代码都很重要
}
```

我们分两种情况来讨论吧，一种是 no-hz,即 CONFIG_NO_HZ=Y,CONFIG_HIGH_RESOLUTION_TIMER=N,这时候 hrtimer_is_hres_enabled() = 0,最终执行:tick_check_oneshot_change(1), 它会调 tick_nohz_switch_to_nohz()转到 nohz 模式。

代码如下：

```
/**
 * tick_nohz_switch_to_nohz - switch to nohz mode
 */
static void tick_nohz_switch_to_nohz(void)
{
```

```
     struct tick_sched *ts = &__get_cpu_var(tick_cpu_sched); //
每个 cpu 的 tick simulation scheduler，包含一个高精定时器，用来模拟旧的
HZ 机制
     ktime_t next;

     if (!tick_nohz_enabled)
          return;

     local_irq_disable();
     if (tick_switch_to_oneshot(tick_nohz_handler)) { //尝试改变
event handler为 tick_nohz_handler.
          local_irq_enable();
          return;
     }

     ts->nohz_mode = NOHZ_MODE_LOWRES;

     /*
      * Recycle the hrtimer in ts, so we can share the
      * hrtimer_forward with the highres code.
      */
     hrtimer_init(&ts->sched_timer, CLOCK_MONOTONIC,
HRTIMER_MODE_ABS);//初始化调度定时器，放在 monotonic base 里
     /* Get the next period */
     next = tick_init_jiffy_update();//设定调度定时器在下一个 tick
period 后过期。。。它一定是周期变化的，用来替代旧的 HZ.

     for (;;) {
          hrtimer_set_expires(&ts->sched_timer, next);
          if (!tick_program_event(next, 0)) //非强制设置硬件时钟下一
个中断的时间点,有可能失败
               break;
          next = ktime_add(next, tick_period);
     }
     local_irq_enable();

     printk(KERN_INFO "Switched to NOHz mode on CPU #%d\n",
            smp_processor_id());
}
```

按照代码中的注释，这个函数比较好理解，两句话：
1）更改 clock event interrupt handler
2）初始化调度定时器，并设定在下一个 tick period 过期，触发中断

这个函数执行完后，过一个 tick_period 就会硬件时钟触发中断，并跳到新的 handler 里处理。
5.回到 hrtimer_run_pending(),当转换成 no-hz 后，
```
if (tick_check_oneshot_change(!hrtimer_is_hres_enabled()))
        hrtimer_switch_to_hres();
```

if 条件不成立，将不再有机会转化到 hrtimer 中断模式了。如果打开了 HIGH_RESOLUTIONS 的模式，在
tick_check_oneshot_change(int allow_nohz)中参数为 0,将执行
```
if (!allow_nohz)
        return 1;
```
后返回 1.那么就会执行
hrtimer_switch_to_hres();

我们分析完 no-hz 模式中断后，再从这个点继续往下分析，看看如何转到高精度时间模式。

到这儿，旧的 hz tick 中断机制就要结束了，在下一个 tick 中断到来时，中断处理函数由 tick_handle_periodic 转变到 tick_no-hz_handler。

6.nohz 中断处理函数
```
/*
 * The nohz low res interrupt handler
 */
static void tick_nohz_handler(struct clock_event_device *dev)
{
    struct tick_sched *ts = &__get_cpu_var(tick_cpu_sched);
    struct pt_regs *regs = get_irq_regs();
    int cpu = smp_processor_id();//current cpu id
    ktime_t now = ktime_get();//get the time now

    dev->next_event.tv64 = KTIME_MAX;

    /*
     * Check if the do_timer duty was dropped. We don't care
about
     * concurrency: This happens only when the cpu in charge
went
     * into a long sleep. If two cpus happen to assign themself
to
     * this duty, then the jiffies update is still serialized
by
     * xtime_lock.
     */
```

```
    if (unlikely(tick_do_timer_cpu == TICK_DO_TIMER_NONE))
        tick_do_timer_cpu = cpu;

    /* Check, if the jiffies need an update */
    if (tick_do_timer_cpu == cpu)
        tick_do_update_jiffies64(now);

    /*
     * When we are idle and the tick is stopped, we have to
touch
     * the watchdog as we might not schedule for a really long
     * time. This happens on complete idle SMP systems while
     * waiting on the login prompt. We also increment the
"start
     * of idle" jiffy stamp so the idle accounting adjustment
we
     * do when we go busy again does not account too much
ticks.
     */
    if (ts->tick_stopped) {
        touch_softlockup_watchdog();
        ts->idle_jiffies++;
    }

    update_process_times(user_mode(regs));//做每个 tick 中断要做的
事情
    profile_tick(CPU_PROFILING);

    while (tick_nohz_reprogram(ts, now)) { 设置下一个时间中断
        now = ktime_get();
        tick_do_update_jiffies64(now);
    }
}
```

这个函数并非是周期性执行的，有可能 cpu 已经 idle 了一段时间，所以这时候要及时对 jiffies 进行补偿：
tick_do_update_jiffies64(now);
在看这个函数的时候，还有一个很重要的变量：last_jiffies_update，它保留着上一次更新 jiffies 时的时间点，这样我们可以计算出 cpu idle 的时间了，继而算出这段时间的 jiffies 丢失值，进而补偿。

这个地方还需要看一下，什么情况下进 idle,让 jiffies 不再更新，什么时候退出，进而补偿等等。Borqs android pxa 平台有个 sleep 机制，在睡眠时只更新 wall time,睡醒时不更新 jiffies,我在调试过程中发现这和 no-hz 机制的 idle 后不报

tick 中断是有区别的。查了些资料，好像只有 x86 支持开头说的比较 cool 的 nohz 优势，也发现 stop_nohz_handler 确实在 x86 平台下才被调用。这个地方我不确定。

下面比较下在 nohz 中断处理函数里面对下一次时间中断设置与刚从 tick_handle_periodic 转换过来时设置硬件时钟的不同。

```
while (tick_nohz_reprogram(ts, now)) { 设置下一个时间中断
          now = ktime_get();
          tick_do_update_jiffies64(now);
     }
```

和

```
for (;;) {
          hrtimer_set_expires(&ts->sched_timer, next);
          if (!tick_program_event(next, 0)) //非强制设置硬件时钟下一个中断的时间点,有可能失败
               break;
          next = ktime_add(next, tick_period);
     }
```

下面这个很明显，把 ts->sched_timer 的过期设置在下一个 tick_period，并尝试修改硬件时钟。而上面的则调用 tick_nohz_reprogram(ts, now)去设置下一个时钟中断。这儿有个非常有意思的地方，在 tick_nohz_reprogram 中，对 ts->sched_timer 过期时间的设置，是能过 hrtimer_forward()来实现的，即在当前过期时间的基础上，再延长一个 tick_period，而非直接 hrtimer_set_expires，这说明如果想进 idle 时，只需要重新修改 ts->sched_timer 的下一个过期时间即可。

关于这部分信息，可以再分析下 tick_nohz_stop_sched_tick()这个函数，它会做这部分事情。在退出中断时，有可能调用它。详细信息在 irq_exit()里。

在 nohz 模式下，这个调度定时器，它的功能并不是特别大，只是借它一用，在设置下一个中断时，可以适当地延长一些。这和旧的时间 tick 机制差别不大，并没有提高定时器的精度，因为过期的定时器的执行，仍然是在 tick 中断里依据 jiffies 处理的。下面将要分析的高精度时间机制，就把这种情况打破了。。。

到现在为止 tick_nohz_handler 的处理就介绍完了。下面看一下如何转到 high resolution timer,并且 high resolution timer works mechasim.

HRTIMER

在了解高精度时间机制前，请先参考这篇文章（http://lwn.net/Articles/162773/）及内核文档目录里关于 hrtimer 的介绍。

接着之前在软中断里由旧的时间中断机制转到新的(nohz 或 hrtimer)时间中断处理函数，

```
     if (tick_check_oneshot_change(!hrtimer_is_hres_enabled()))
          hrtimer_switch_to_hres();
```

在打开 hrtimer 后，if 条件成立，执行 hrtimer_switch_to_hres().
关于这个函数，有了前 2 种 handler 的建立认识，它也不难了。详细看下代码：

```
/*
 * Switch to high resolution mode
 */
static int hrtimer_switch_to_hres(void)
{
     int cpu = smp_processor_id();
     struct hrtimer_cpu_base *base = &per_cpu(hrtimer_bases,
cpu);
     unsigned long flags;

     if (base->hres_active)
          return 1;

     local_irq_save(flags);

     if (tick_init_highres()) {//转到高精时间模式
          local_irq_restore(flags);
          printk(KERN_WARNING "Could not switch to high
resolution "
                         "mode on CPU %d\n", cpu);
          return 0;
     }
     base->hres_active = 1;
     base->clock_base[CLOCK_REALTIME].resolution =
```

```
KTIME_HIGH_RES;
     base->clock_base[CLOCK_MONOTONIC].resolution =
KTIME_HIGH_RES;

     tick_setup_sched_timer();//上面介绍过的调度定时器，保持原有的
tick 架构不变。。。

     /* "Retrigger" the interrupt to get things going */
     retrigger_next_event(NULL); //触发下一次中断
     local_irq_restore(flags);
     printk(KERN_DEBUG "Switched to high resolution mode on CPU
%d\n",
            smp_processor_id());
     return 1;
}
```

下面将分别介绍五部分：转到高精时间模式 ；建立调度定时器;触发下次中断；高精中断处理函数；调度定时器执行函数.

1 转到高精时间模式

```
int tick_init_highres(void)
{
     return tick_switch_to_oneshot(hrtimer_interrupt);
}
```

将 hrtimer_interrupt 作为参数传入，
`int tick_switch_to_oneshot(void (*handler)(struct clock_event_device *))`将把当前的时间处理机制转化到高精模式。

这个函数细节不再分析了，它跟前面转到 nohz handler 一样。这儿只需牢记一点儿，下一个时钟中断来的时候，就跑到 hrtimer_interrupt 里去执行了。

2.建立调度定时器

在上面的代码中，会设置 hrtimer_cpu_base 里两个 hrtimer 数组为高精模式，然后 tick_setup_sched_timer()建立 hz 机制。

Code:

```
/**
 * tick_setup_sched_timer - setup the tick emulation timer
 */
void tick_setup_sched_timer(void)
{
```

```
     struct tick_sched *ts = &__get_cpu_var(tick_cpu_sched);
     ktime_t now = ktime_get();
     u64 offset;

     /*
      * Emulate tick processing via per-CPU hrtimers:
      */   这儿初始化与之前在 nohz 是一样的。
     hrtimer_init(&ts->sched_timer, CLOCK_MONOTONIC,
HRTIMER_MODE_ABS);
     ts->sched_timer.function = tick_sched_timer;
//执行函数

     /* Get the next period (per cpu) */
     hrtimer_set_expires(&ts->sched_timer,
tick_init_jiffy_update());
     offset = ktime_to_ns(tick_period) >> 1;
     do_div(offset, num_possible_cpus());
     offset *= smp_processor_id();
     hrtimer_add_expires_ns(&ts->sched_timer, offset);

     for (;;) {
          hrtimer_forward(&ts->sched_timer, now, tick_period);
          hrtimer_start_expires(&ts->sched_timer,
HRTIMER_MODE_ABS);  //让调度定时器跑起来
          /* Check, if the timer was already in the past */
          if (hrtimer_active(&ts->sched_timer))
               break;
          now = ktime_get();
     }

#ifdef CONFIG_NO_HZ
     if (tick_nohz_enabled)
          ts->nohz_mode = NOHZ_MODE_HIGHRES;
#endif
}
```

之前在未打开 hrtimer 模式时，这个调度定时器是没有执行函数的，因为它也压根没有 start 过，这儿给它设置执行 function,并在下面让它 start，加入到 hrtimer red black 树里，等它过期时取出来执行。这个执行函数，就是做一些 scheduler_tick()之类的中断里做的事情。那么，为什么之前没有执行函数，现在有了呢？

这儿做这样的一个比喻。在没有 hrtimer 之前，一个定时器的精度，最大就是 hz，按照 jiffies 来保证系统时间的更新。这样，如果要提高定时器的精度 ，只能提高 HZ

的数值，但是这个代价太大了，搞得系统会很频繁地进行调度。Borqs android kernel 的 hz 值是 128,这样每个 jiffies 的值大约是 7~8ms,那么执行进程 sleep 或者 delay 的精度就可想而知了。而硬件定时器提供 32.768khz,精度远超过 jiffies.

hrtimer 的时间精度，支持到最高 ns,只要硬件支持，ns 的 sleep 也是可以的。这之间的跨度是非常大的，对时间精度要求高的系统，将会很受用哦~

现在应该有这样的一种印象了，采用高精度时间的内核，支持高达 ns 级别的定时器，假如此时的 HZ=128,硬件时钟频率是 32.768khz,那么系统每隔 7~8 毫秒就会支持一次这个 sched_timer 定时器，用来做每个 tick 中断做的事情，比如更新进程 运行时间 ，完成一些进程的调度等等。而如果你同时设置了放多定时器，精度远比 jiffies 高，那么系统照样准确地执行。所有的高精度定时器都是在红黑树中排列的，调度定时器只是其中一个而已。它只是周期性地执行，执行，并不对其它比它精度高或低的普通定时器产生 影响。 大家看我之前测试不同时间模式的调试信息会发现在 monotoic hrtimer clock base 数组里，是这样的信息：

```
clock 1:
  .index:      1
  .resolution: 1 nsecs
  .get_time:   ktime_get
  .offset:     0 nsecs
active timers:
#0: <c4859e78>, tick_sched_timer, S:01
# expires at 516039062500 nsecs [in 4547120 nsecs]
#1: <c4859e78>, hrtimer_wakeup, S:01
# expires at 516047668457 nsecs [in 13153077 nsecs]
#2: <c4859e78>, hrtimer_wakeup, S:01
# expires at 516077905273 nsecs [in 43389893 nsecs]
#3: <c4859e78>, hrtimer_wakeup, S:01
# expires at 516704315185 nsecs [in 669799805 nsecs]
#4: <c4859e78>, hrtimer_wakeup, S:01
# expires at 517013671875 nsecs [in 979156495 nsecs]
#5: <c4859e78>, hrtimer_wakeup, S:01
# expires at 517016418457 nsecs [in 981903077 nsecs]
#6: <c4859e78>, hrtimer_wakeup, S:01
# expires at 518217690429 nsecs [in 2183175049 nsecs]
#7: <c4859e78>, hrtimer_wakeup, S:01
# expires at 518306128906 nsecs [in 2271613526 nsecs]
#8: <c4859e78>, hrtimer_wakeup, S:01
# expires at 522464243896 nsecs [in 6429728516 nsecs]
#9: <c4859e78>, hrtimer_wakeup, S:01
# expires at 523713317871 nsecs [in 7678802491 nsecs]
#10: <c4859e78>, it_real_fn, S:01
```

```
# expires at 532023681640 nsecs [in 15989166260 nsecs]
#11: <c4859e78>, hrtimer_wakeup, S:01
# expires at 572952708984 nsecs [in 56918193604 nsecs]
#12: <c4859e78>, hrtimer_wakeup, S:01
# expires at 573862792968 nsecs [in 57828277588 nsecs]
#13: <c4859e78>, it_real_fn, S:01
# expires at 616407379150 nsecs [in 100372863770 nsecs]
#14: <c4859e78>, hrtimer_wakeup, S:01
# expires at 1800000745605 nsecs [in 1283966230225 nsecs]
#15: <c4859e78>, hrtimer_wakeup, S:01
# expires at 10381001688964 nsecs [in 9864967173584 nsecs]
#16: <c4859e78>, hrtimer_wakeup, S:01
# expires at 16777227615020751 nsecs [in 16776711580505371
nsecs]
  .expires_next   : 516039062500 nsecs
  .hres_active    : 1
  .nr_events      : 31890
  .nohz_mode      : 2
  .idle_tick      : 516007812500 nsecs
  .tick_stopped   : 0
  .idle_jiffies   : 27649
  .idle_calls     : 60167
  .idle_sleeps    : 34351
  .idle_entrytime : 516009765625 nsecs
  .idle_waketime  : 516015594482 nsecs
  .idle_exittime  : 516015686035 nsecs
  .idle_sleeptime : 438415315475 nsecs
  .last_jiffies   : 27649
  .next_jiffies   : 27650
  .idle_expires   : 516015625000 nsecs
jiffies: 27652
```

3 触发下次中断

retrigger_next_event()将会触发下一次硬件时钟中断，会跳到hrtimer_interrupt()处理函数里执行。修改硬件时钟的函数是：

```
/*
 * Reprogram the event source with checking both queues for the
 * next event
 * Called with interrupts disabled and base->lock held
 */
static void hrtimer_force_reprogram(struct hrtimer_cpu_base
*cpu_base)
{
```

```
    int i;
    struct hrtimer_clock_base *base = cpu_base->clock_base;
    ktime_t expires;

    cpu_base->expires_next.tv64 = KTIME_MAX;

    for (i = 0; i < HRTIMER_MAX_CLOCK_BASES; i++, base++) {
        struct hrtimer *timer;

        if (!base->first)
            continue;
        timer = rb_entry(base->first, struct hrtimer, node);
        expires = ktime_sub(hrtimer_get_expires(timer), base-
>offset);
        /*
         * clock_was_set() has changed base->offset so the
         * result might be negative. Fix it up to prevent a
         * false positive in clockevents_program_event()
         */
        if (expires.tv64 < 0)
            expires.tv64 = 0;
        if (expires.tv64 < cpu_base->expires_next.tv64)
            cpu_base->expires_next = expires;
    }

    if (cpu_base->expires_next.tv64 != KTIME_MAX)
        tick_program_event(cpu_base->expires_next, 1);
}
```

这个函数比较 cpu_base 与红黑树中第一个要过期的 timer 的 expires 值，选择比较近的设置到硬件时钟。

这样，最多在一个 tick_period 时间内，将会触发下一次硬件时钟中断，那么新的硬件中断处理函数将会做什么事情呢？ 它会根据不同的 hrtimer 的发生，选择性的执行，并把最近的要过期的下一个 hrtimer 设置到硬件时钟。这样，高精度时间处理就正常运转起来 了。

高精中断处理函数

```
4.
5. /*
6.  * High resolution timer interrupt
```

```
7. * Called with interrupts disabled
8. */
9.void hrtimer_interrupt(struct clock_event_device *dev)
10.{
11.    struct hrtimer_cpu_base *cpu_base =
  &__get_cpu_var(hrtimer_bases);
12.    struct hrtimer_clock_base *base;
13.    ktime_t expires_next, now;
14.    int nr_retries = 0;
15.    int i;
16.
17.    BUG_ON(!cpu_base->hres_active);
18.    cpu_base->nr_events++;
19.    dev->next_event.tv64 = KTIME_MAX;
20.
21. retry:
22.    /* 5 retries is enough to notice a hang */
23.    if (!(++nr_retries % 5))
24.        hrtimer_interrupt_hanging(dev,
  ktime_sub(ktime_get(), now));
25.
26.    now = ktime_get();
27.
28.    expires_next.tv64 = KTIME_MAX;
29.
30.    base = cpu_base->clock_base;
31.  //处理当前cpu-base里 real+monotonic两个hrtimer红黑树里所有的
  过期hrtimer
32.    for (i = 0; i < HRTIMER_MAX_CLOCK_BASES; i++) {
33.        ktime_t basenow;
34.        struct rb_node *node;
35.
36.        spin_lock(&cpu_base->lock);
37.
38.        basenow = ktime_add(now, base->offset);
39.
40.        while ((node = base->first)) { //循环在此红黑树中所有
  的过期定时器
41.            struct hrtimer *timer;
42.
43.            timer = rb_entry(node, struct hrtimer, node);
44.
45.            /*
46.             * The immediate goal for using the
```

```
  softexpires is
47.              * minimizing wakeups, not running timers at
  the
48.              * earliest interrupt after their soft
  expiration.
49.              * This allows us to avoid using a Priority
  Search
50.              * Tree, which can answer a stabbing querry
  for
51.              * overlapping intervals and instead use the
  simple
52.              * BST we already have.
53.              * We don't add extra wakeups by delaying
  timers that
54.              * are right-of a not yet expired timer,
  because that
55.              * timer will have to trigger a wakeup
  anyway.
56.              */
57.
58.             if (basenow.tv64 <
  hrtimer_get_softexpires_tv64(timer)) {
59.                 ktime_t expires;
60.
61.                 expires =
  ktime_sub(hrtimer_get_expires(timer),
62.                                   base->offset);
63.                 if (expires.tv64 < expires_next.tv64)
64.                     expires_next = expires;
65.                 break;
66.             }
67.             注：最新代码此地方有变动，不再根据 timer cbmod 将过期
  timer 放到链表中
68.             __run_hrtimer(timer); //运行过期的定时器
69.         }
70.         spin_unlock(&cpu_base->lock);
71.         base++;
72.     }
73.
74.     cpu_base->expires_next = expires_next;
75.
76.     /* Reprogramming necessary ? */
77.     if (expires_next.tv64 != KTIME_MAX) {
78.         if (tick_program_event(expires_next,
```

```
    force_clock_reprogram))    //编写硬件时钟，设置最近的过期的
    hrtimer
79.             goto retry;
80.  //2.6.25在此处有如下代码：
81.     /* Raise softirq ? */
82.     if (raise)
83.         raise_softirq(HRTIMER_SOFTIRQ);
84.     }
85.}
86.最新的代码里此中断处理函数已经有所变动。2.6.25 中在真正要执行一个过期
    的 timer 前，会有如下 判断代码：
87./* Move softirq callbacks to the pending list */
88.             if (timer->cb_mode == HRTIMER_CB_SOFTIRQ) {
89.                 __remove_hrtimer(timer, base,
90.                                  HRTIMER_STATE_PENDING, 0);
91.                 list_add_tail(&timer->cb_entry,
92.                                   &base->cpu_base->cb_pending);
93.                 raise = 1;
94.                 continue;
95.             }
```

将最终判断过期的 hrtimer 是否立即执行。这儿请注意，并非所有的 hrtimer 过期后需要立即执行，有一些定时器需要相应的执行环境.请留意 上面红色部分。如果一些代码需要在软中断环境中执行，则需要 raise_softirq,那么在 tick 中断里，会反回来调前面已经注册的 hrtimer 软中断函数：

```
static void run_hrtimer_softirq(struct softirq_action *h)
{
    run_hrtimer_pending(&__get_cpu_var(hrtimer_bases));
}
```

而 run_hrtimer_pending 将 cpu_base->cb_pending 链表上过期的 hrtimer 在软中断环境里执行。

最新的代码在这个地方有了比较大的改动！！！（我也是刚看 2629-rc9 发现的，还以为之前理解错了呢 ^-^）

```
    run_hrtimer()真正执行过期的 hrtimer.
    static void __run_hrtimer(struct hrtimer *timer)
96.{
97.    struct hrtimer_clock_base *base = timer->base;
98.    struct hrtimer_cpu_base *cpu_base = base->cpu_base;
99.    enum hrtimer_restart (*fn)(struct hrtimer *);
100.   int restart;
101.
102.   WARN_ON(!irqs_disabled());
103.
```

```
104.   debug_hrtimer_deactivate(timer);
105.   __remove_hrtimer(timer, base, HRTIMER_STATE_CALLBACK,
  0);
106.   timer_stats_account_hrtimer(timer);
107.   fn = timer->function;
108.
109.   /*
110.    * Because we run timers from hardirq context, there
  is no chance
111.    * they get migrated to another cpu, therefore its
  safe to unlock
112.    * the timer base.
113.    */
114.   spin_unlock(&cpu_base->lock);
115.   restart = fn(timer);
116.   spin_lock(&cpu_base->lock);
117.
118.   /*
119.    * Note: We clear the CALLBACK bit after
  enqueue_hrtimer and
120.    * we do not reprogramm the event hardware. Happens
  either in
121.    * hrtimer_start_range_ns() or in hrtimer_interrupt()
122.    */
123.   if (restart != HRTIMER_NORESTART) {
124.       BUG_ON(timer->state != HRTIMER_STATE_CALLBACK);
125.       enqueue_hrtimer(timer, base);
126.   }
127.   timer->state &= ~HRTIMER_STATE_CALLBACK;
128.}
```

这个函数不难理解，先将过期的 hrtimer 从树上摘下来，运行 它的 funtion,决定是否需要继续 restart，需要入列否等等。 这部分对 hrtimer 操作的细节，我后面看情况再做详细分析。本文的主要目的是介绍时间系统的运行机制。

5 调度定时器执行函数

之前在 setup 调度定时器时，设置 tick_sched_timer 为模拟 tick 的定时器的处理函数。

```
    ts->sched_timer.function = tick_sched_timer;
```

这个函数的代码如下：

```
static enum hrtimer_restart tick_sched_timer(struct hrtimer
*timer)
```

```
{
     struct tick_sched *ts =
         container_of(timer, struct tick_sched, sched_timer);
     struct pt_regs *regs = get_irq_regs();
     ktime_t now = ktime_get();
     int cpu = smp_processor_id();

#ifdef CONFIG_NO_HZ
     /*
      * Check if the do_timer duty was dropped. We don't care
about
      * concurrency: This happens only when the cpu in charge
went
      * into a long sleep. If two cpus happen to assign themself
to
      * this duty, then the jiffies update is still serialized
by
      * xtime_lock.
      */
     if (unlikely(tick_do_timer_cpu == TICK_DO_TIMER_NONE))
         tick_do_timer_cpu = cpu;
#endif

     /* Check, if the jiffies need an update */
     if (tick_do_timer_cpu == cpu)
         tick_do_update_jiffies64(now);

     /*
      * Do not call, when we are not in irq context and have
      * no valid regs pointer
      */
     if (regs) {
         /*
          * When we are idle and the tick is stopped, we have
to touch
          * the watchdog as we might not schedule for a really
long
          * time. This happens on complete idle SMP systems
while
          * waiting on the login prompt. We also increment the
"start of
          * idle" jiffy stamp so the idle accounting adjustment
we do
          * when we go busy again does not account too much
```

```
ticks.
          */
         if (ts->tick_stopped) {
              touch_softlockup_watchdog();
              ts->idle_jiffies++;
         }
         update_process_times(user_mode(regs));
         profile_tick(CPU_PROFILING);
     }

     hrtimer_forward(timer, now, tick_period);

     return HRTIMER_RESTART;
}
```

它做的事情也很简单，完成旧的 tick 中断机制做的事情，延长自己 的 expires 值，设定在下一个 tick_period,返回 restart，那么前面 run_hrtimer()将会重新将此 hrtimer 入列。

这样，用一个 sched_timer 就把旧的 tick 时钟中断机制实现了。

在 2.6.25 代码中，在时钟中断里会把需要软中断执行环境的一些过期高精度定时器从链表上摘下来执行。最新的代码在这部分变化比较大，我会在后面继续学习研究，与大家分享。

这样 ，我们从系统初始化到根据不同的 kernel 配置，选择不同的硬件时钟中断处理函数，最终建立时间系统的分析就靠一段落。我没有在“普通定时器的运行 机制 ，高精度定时器的建立”等方面进行过多的分析。相反，我觉得如果明白了 kernel 时间系统的运行机制，也就了解了不同类型的定时器如何建立，运行，维护等。

我在 Borqs Android 手机上对时间系统进行过一些测试，但由于保密原因，一些东西不能直接贴出来。我把几个不同配置的调试信息贴在这儿，大家可以结合文中的分析做下对应。

由于个人水平和时间原因，分析肯定有许多出错的地方，一些地方也不太详细，或者一些地方显得很罗琐，请大家指正，包涵。

希望有机会把 cfs + hrtimer + mm 结合着总结下，与大家分享。

谢谢！
Apr26 2009 Sunshine

我的联系方式 ：
Email:wxc200@gmail.com

MSN:wxc200@hotmail.com
Company:BORQS BJ

补一段帖子的内容简介

帖子地址：
1. 内核时间系统（http://www.linuxforum.net/forum/showflat.php?Cat=&Board=linuxK&Number=717291&page=0&view=collapsed&sb=5&o=7&fpart=）
2.**hrtimer(高精度时间处理)--欢迎讨论交流**（http://linux.chinaunix.net/bbs/thread-1098007-1-1.html）
内核的时间系统,还是比较有意思的.上次写过 cfs 后,便看了 hrtimer 部分,一直没抽出时间来整理,现在还有一些不太明白的地方,我准备开这个帖子,和大家多聊聊,聊完了,我再做个总结...毕竟,一个人的思维局限性很大,呵呵

我的理解肯定有许多错误的地方,欢迎大家指正,交流....

今晚只简单说说,我会在后面详细总结出内核时间系统的建立和运行机制,并将学习过程中收集的资料,与大家分享.

1） 采用 hrtimer resolution 的 kernel,在硬件支持的情况下,可以提供精度更高的定时器

2） 没有那么高精度的硬件 kernel 怎么办呢？ hrtimer 系统会建立一个 hrtimer,模拟之前的 tick 机制..说白了,就是往硬件的 timer 寄存器写下一个 tick 的值,模拟出 tick 中断来.

1)的精度,就是硬件的时钟精度,,软件支持最高是 1ns,大约 1Ghz 就行了...如果比 1g 还高,我的理解是 kernel 目前不支持.
也就是说,如果你的 hardware support 1ghz,你可以设定一个 1ns

的定时器....不能再低了.

2)的精度就不一样了,它是 hz.在每个 hz 的处理函数里,会处理那些过期的 hrtimer.

1)的处理函数是 hrtimer_interrupt(),
我的 kernel 打开了 CONFIG_NOHZ,这样 2)的处理函数是 tick_nohz_handler().

调试部分

我贴几种不同宏配置的时间调试信息。请大家主要留意 event handler 和 timer resolution:
查看 调试信息 cat /proc/timer_list
1）旧的 hz 模式：

```
cat timer_list
Timer List Version: v0.3
HRTIMER_MAX_CLOCK_BASES: 2
now at 49850860595 nsecs

cpu: 0
clock 0:
  .index:      0
  .resolution: 7812500 nsecs
  .get_time:   ktime_get_real
active timers:
#0: <c5ea1e88>, alarm_timer_triggered, S:01
# expires at 946689840000000000 nsecs [in 946689790149139405
nsecs]
#1: <c5ea1e88>, alarm_timer_triggered, S:01
# expires at 946690079041129395 nsecs [in 946690029190268800
nsecs]
#2: <c5ea1e88>, alarm_timer_triggered, S:01
# expires at 946769048688000000 nsecs [in 946768998837139405
nsecs]
clock 1:
  .index:      1
  .resolution: 7812500 nsecs
  .get_time:   ktime_get
active timers:
#0: <c5ea1e88>, hrtimer_wakeup, S:01
```

```
# expires at 50172006835 nsecs [in 321146240 nsecs]
#1: <c5ea1e88>, it_real_fn, S:01
# expires at 52031829833 nsecs [in 2180969238 nsecs]
#2: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 52184722605 nsecs [in 2333862010 nsecs]
#3: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 52805023193 nsecs [in 2954162598 nsecs]
#4: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 53235778808 nsecs [in 3384918213 nsecs]
#5: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 53385986123 nsecs [in 3535125528 nsecs]
#6: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 86530027587 nsecs [in 36679166992 nsecs]
#7: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 87602142333 nsecs [in 37751281738 nsecs]
#8: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 88824238281 nsecs [in 38973377686 nsecs]
#9: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 94290870117 nsecs [in 44440009522 nsecs]
#10: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 98105346679 nsecs [in 48254486084 nsecs]
#11: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 310324707031 nsecs [in 260473846436 nsecs]
#12: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 1800000342773 nsecs [in 1750149482178 nsecs]
#13: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 3613476715087 nsecs [in 3563625854492 nsecs]
#14: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 10024793676757 nsecs [in 9974942816162 nsecs]
#15: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 86446031921386 nsecs [in 86396181060791 nsecs]
#16: <c5ea1e88>, hrtimer_wakeup, S:01
# expires at 16777220143981933 nsecs [in 16777170293121338
nsecs]
  .nohz_mode      : 0
  .idle_tick      : 0 nsecs
  .tick_stopped   : 0
  .idle_jiffies   : 0
  .idle_calls     : 0
  .idle_sleeps    : 0
  .idle_entrytime : 0 nsecs
  .idle_waketime  : 0 nsecs
  .idle_exittime  : 0 nsecs
  .idle_sleeptime : 0 nsecs
```

```
  .last_jiffies   : 0
  .next_jiffies   : 0
  .idle_expires   : 0 nsecs
jiffies: 4294935277


Tick Device: mode:     0
Clock Event Device: 32k timer
max_delta_ns:   2354769360
min_delta_ns:   488953
mult:           549
shift:          24
mode:           3
next_event:     49859375000 nsecs
set_next_event: pxa_timer_set_next_event
set_mode:       pxa_timer_set_mode
event_handler:  tick_handle_periodic

2) CONFIG_NO_HZ
/proc # cat timer_list
Timer List Version: v0.3
HRTIMER_MAX_CLOCK_BASES: 2
now at 239915557861 nsecs

cpu: 0
clock 0:
  .index:      0
  .resolution: 7812500 nsecs
  .get_time:   ktime_get_real
active timers:
#0: <cdb19e88>, alarm_timer_triggered, S:01
# expires at 9466852800000000000 nsecs [in 9466850400844442139
nsecs]
#1: <cdb19e88>, alarm_timer_triggered, S:01
# expires at 946685373843936036 nsecs [in 9466851339283781 75
nsecs]
#2: <cdb19e88>, alarm_timer_triggered, S:01
# expires at 9466853793800000000 nsecs [in 9466851394644421 39
nsecs]
clock 1:
  .index:      1
  .resolution: 7812500 nsecs
  .get_time:   ktime_get
active timers:
```

```
#0: <cdb19e88>, hrtimer_wakeup, S:01
# expires at 240383627197 nsecs [in 468069336 nsecs]
#1: <cdb19e88>, hrtimer_wakeup, S:01
# expires at 240884918212 nsecs [in 969360351 nsecs]
#2: <cdb19e88>, hrtimer_wakeup, S:01
# expires at 243968872070 nsecs [in 4053314209 nsecs]
#3: <cdb19e88>, it_real_fn, S:01
# expires at 245430206298 nsecs [in 5514648437 nsecs]
#4: <cdb19e88>, hrtimer_wakeup, S:01
# expires at 275922088623 nsecs [in 36006530762 nsecs]
#5: <cdb19e88>, hrtimer_wakeup, S:01
# expires at 284986511230 nsecs [in 45070953369 nsecs]
#6: <cdb19e88>, hrtimer_wakeup, S:01
# expires at 311084014892 nsecs [in 71168457031 nsecs]
#7: <cdb19e88>, hrtimer_wakeup, S:01
# expires at 1800000843749 nsecs [in 1560085285888 nsecs]
#8: <cdb19e88>, hrtimer_wakeup, S:01
# expires at 3614101806640 nsecs [in 3374186248779 nsecs]
#9: <cdb19e88>, hrtimer_wakeup, S:01
# expires at 10163399799804 nsecs [in 9923484241943 nsecs]
#10: <cdb19e88>, hrtimer_wakeup, S:01
# expires at 86639430297851 nsecs [in 86399514739990 nsecs]
#11: <cdb19e88>, hrtimer_wakeup, S:01
# expires at 16777220448211669 nsecs [in 16776980532653808
nsecs]
  .nohz_mode      : 1
  .idle_tick      : 239835937500 nsecs
  .tick_stopped   : 0
  .idle_jiffies   : 4294959594
  .idle_calls     : 38015
  .idle_sleeps    : 24630
  .idle_entrytime : 239891448974 nsecs
  .idle_waketime  : 239891357421 nsecs
  .idle_exittime  : 239891387939 nsecs
  .idle_sleeptime : 161173858885 nsecs
  .last_jiffies   : 4294959602
  .next_jiffies   : 4294959603
  .idle_expires   : 239968750000 nsecs
jiffies: 4294959605


Tick Device: mode:     1
Clock Event Device: 32k timer
max_delta_ns:   2354769360
```

```
min_delta_ns:   488953
mult:           549
shift:          24
mode:           3
next_event:     239921875000 nsecs
set_next_event: pxa_timer_set_next_event
set_mode:       pxa_timer_set_mode
event_handler:  tick_nohz_handler

3)HRTIMER_RESOLUTION
/proc # cat timer_list
Timer List Version: v0.3
HRTIMER_MAX_CLOCK_BASES: 2
now at 516034515380 nsecs

cpu: 0
clock 0:
  .index:      0
  .resolution: 1 nsecs
  .get_time:   ktime_get_real
  .offset:     9466876894999084448 nsecs
active timers:
#0: <c4859e78>, alarm_timer_triggered, S:01
# expires at 9466882200000000000 nsecs [in 9466877039654846620
nsecs]
#1: <c4859e78>, alarm_timer_triggered, S:01
# expires at 9466883931689084448 nsecs [in 9466878771343930688
nsecs]
clock 1:
  .index:      1
  .resolution: 1 nsecs
  .get_time:   ktime_get
  .offset:     0 nsecs
active timers:
#0: <c4859e78>, tick_sched_timer, S:01
# expires at 516039062500 nsecs [in 4547120 nsecs]
#1: <c4859e78>, hrtimer_wakeup, S:01
# expires at 516047668457 nsecs [in 13153077 nsecs]
#2: <c4859e78>, hrtimer_wakeup, S:01
# expires at 516077905273 nsecs [in 43389893 nsecs]
#3: <c4859e78>, hrtimer_wakeup, S:01
# expires at 516704315185 nsecs [in 669799805 nsecs]
#4: <c4859e78>, hrtimer_wakeup, S:01
# expires at 517013671875 nsecs [in 979156495 nsecs]
```

```
#5: <c4859e78>, hrtimer_wakeup, S:01
# expires at 517016418457 nsecs [in 981903077 nsecs]
#6: <c4859e78>, hrtimer_wakeup, S:01
# expires at 518217690429 nsecs [in 2183175049 nsecs]
#7: <c4859e78>, hrtimer_wakeup, S:01
# expires at 518306128906 nsecs [in 2271613526 nsecs]
#8: <c4859e78>, hrtimer_wakeup, S:01
# expires at 522464243896 nsecs [in 6429728516 nsecs]
#9: <c4859e78>, hrtimer_wakeup, S:01
# expires at 523713317871 nsecs [in 7678802491 nsecs]
#10: <c4859e78>, it_real_fn, S:01
# expires at 532023681640 nsecs [in 15989166260 nsecs]
#11: <c4859e78>, hrtimer_wakeup, S:01
# expires at 572952708984 nsecs [in 56918193604 nsecs]
#12: <c4859e78>, hrtimer_wakeup, S:01
# expires at 573862792968 nsecs [in 57828277588 nsecs]
#13: <c4859e78>, it_real_fn, S:01
# expires at 616407379150 nsecs [in 100372863770 nsecs]
#14: <c4859e78>, hrtimer_wakeup, S:01
# expires at 1800000745605 nsecs [in 1283966230225 nsecs]
#15: <c4859e78>, hrtimer_wakeup, S:01
# expires at 10381001688964 nsecs [in 9864967173584 nsecs]
#16: <c4859e78>, hrtimer_wakeup, S:01
# expires at 16777227615020751 nsecs [in 16776711580505371
nsecs]
  .expires_next   : 516039062500 nsecs
  .hres_active    : 1
  .nr_events      : 31890
  .nohz_mode      : 2
  .idle_tick      : 516007812500 nsecs
  .tick_stopped   : 0
  .idle_jiffies   : 27649
  .idle_calls     : 60167
  .idle_sleeps    : 34351
  .idle_entrytime : 516009765625 nsecs
  .idle_waketime  : 516015594482 nsecs
  .idle_exittime  : 516015686035 nsecs
  .idle_sleeptime : 438415315475 nsecs
  .last_jiffies   : 27649
  .next_jiffies   : 27650
  .idle_expires   : 516015625000 nsecs
jiffies: 27652
```

```
Tick Device: mode:     1
Clock Event Device: 32k timer
max_delta_ns:   2354769360
min_delta_ns:   488953
mult:           549
shift:          24
mode:           3
next_event:     516039062500 nsecs
set_next_event: pxa_timer_set_next_event
set_mode:       pxa_timer_set_mode
event_handler:  hrtimer_interrupt
```

4) 双核ubuntu

```
Timer List Version: v0.3
HRTIMER_MAX_CLOCK_BASES: 2
now at 4809339721317 nsecs

cpu: 0
clock 0:
  .index:      0
  .resolution: 1 nsecs
  .get_time:   ktime_get_real
  .offset:     12404513546107541 29 nsecs
active timers:
clock 1:
  .index:      1
  .resolution: 1 nsecs
  .get_time:   ktime_get
  .offset:     0 nsecs
active timers:
#0: <f7473ebc>, tick_sched_timer, S:01,
tick_nohz_restart_sched_tick,
swapper/0
# expires at 4809340000000 nsecs [in 278683 nsecs]
#1: <f7473ebc>, hrtimer_wakeup, S:01, do_nanosleep, cron/5899
# expires at 4847257809661 nsecs [in 37918088344 nsecs]
#2: <f7473ebc>, it_real_fn, S:01, do_setitimer, exim4/5418
# expires at 5419334664794 nsecs [in 609994943477 nsecs]
  .expires_next   : 4809340000000 nsecs
  .hres_active    : 1
  .nr_events      : 1293964
  .nohz_mode      : 2
  .idle_tick      : 4809248000000 nsecs
  .tick_stopped   : 0
```

```
  .idle_jiffies   : 1127311
  .idle_calls     : 1289832
  .idle_sleeps    : 179294
  .idle_entrytime : 4809336601539 nsecs
  .idle_sleeptime : 1272196402879 nsecs
  .last_jiffies   : 1127333
  .next_jiffies   : 1127467
  .idle_expires   : 4809868000000 nsecs
jiffies: 1127334

cpu: 1
clock 0:
  .index:      0
  .resolution: 1 nsecs
  .get_time:   ktime_get_real
  .offset:     12404513546107541 29 nsecs
active timers:
clock 1:
  .index:      1
  .resolution: 1 nsecs
  .get_time:   ktime_get
  .offset:     0 nsecs
active timers:
#0: <f7473ebc>, tick_sched_timer, S:01,
tick_nohz_stop_sched_tick,
swapper/0
# expires at 4809341000000 nsecs [in 1278683 nsecs]
#1: <f7473ebc>, it_real_fn, S:01, do_setitimer, syslogd/4952
# expires at 4817918067967 nsecs [in 8578346650 nsecs]
#2: <f7473ebc>, hrtimer_wakeup, S:01, do_nanosleep, atd/5880
# expires at 7222135751419 nsecs [in 2412796030102 nsecs]
  .expires_next   : 4809341000000 nsecs
  .hres_active    : 1
  .nr_events      : 1245537
  .nohz_mode      : 2
  .idle_tick      : 4809329000000 nsecs
  .tick_stopped   : 0
  .idle_jiffies   : 1127332
  .idle_calls     : 1844345
  .idle_sleeps    : 381801
  .idle_entrytime : 4809335825455 nsecs
  .idle_sleeptime : 1421067516594 nsecs
  .last_jiffies   : 1127333
  .next_jiffies   : 1127334
```

```
  .idle_expires   : 4809332000000 nsecs
jiffies: 1127334


Tick Device: mode:     1
Clock Event Device: hpet
max_delta_ns:   2147483647
min_delta_ns:   1920
mult:           107374182
shift:          32
mode:           3
next_event:     9223372036854775807 nsecs
set_next_event: hpet_legacy_next_event
set_mode:       hpet_legacy_set_mode
event_handler:  tick_handle_oneshot_broadcast
tick_broadcast_mask: 00000000
tick_broadcast_oneshot_mask: 00000000


Tick Device: mode:     1
Clock Event Device: lapic
max_delta_ns:   669736915
min_delta_ns:   1197
mult:           53795441
shift:          32
mode:           3
next_event:     4809340000000 nsecs
set_next_event: lapic_next_event
set_mode:       lapic_timer_setup
event_handler:  hrtimer_interrupt

Tick Device: mode:     1
Clock Event Device: lapic
max_delta_ns:   669736915
min_delta_ns:   1197
mult:           53795441
shift:          32
mode:           3
next_event:     4809341000000 nsecs
set_next_event: lapic_next_event
set_mode:       lapic_timer_setup
event_handler:  hrtimer_interrupt
```