

DroidPhone的专栏

欢迎各位大虾交流，本人联系方式：droid.phx@gmail.com

目录视图

摘要视图

RSS 订阅

个人资料



DroidPhone

访问：1251851次

积分：9517

等级：

ELUC6

排名：第1606名

原创：51篇

转载：0篇

译文：4篇

评论：555条

文章搜索

文章分类

移动开发之Android (11)

Linux内核架构 (15)

Linux设备驱动 (20)

Linux电源管理 (3)

Linux音频子系统 (15)

Linux中断子系统 (5)

Linux时间管理系统 (8)

Linux输入子系统 (4)

文章存档

2014年07月 (1)

2014年04月 (4)

2013年11月 (4)

2013年10月 (3)

2013年07月 (3)

展开

阅读排行

Linux ALSA声卡驱动之-

Android Audio System

(87756)

Linux ALSA声卡驱动之-

Android Audio System

(61776)

Linux ALSA声卡驱动之-

Android Audio System

(52760)

Linux ALSA声卡驱动之-

Android Audio System

(46894)

【活动】Python创意编程活动开始啦!!!

CSDN日报20170425 ——《私活，是对技术达人最好的点赞》

【CSDN日报】| 3.18-4.18 上榜作者排行出炉

Linux时间子系统之四：定时器的引擎：clock_event_device

2012-09-28 16:0216811人阅读评论4个

分类：Linux时间管理系统 (7)Linux内核架构 (14)

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录(?)

[+]

早期的内核版本中，进程的调度基于一个称之为tick的时钟滴答，通常使用时钟中断来定时地产生tick信号，每次tick定时中断都会进行进程的统计和调度，并对tick进行计数，记录在一个jiffies变量中，定时器的设计也是基于jiffies。这时候的内核代码中，几乎所有关于时钟的操作都是在machine级的代码中实现，很多公共的代码要在每个平台上重复实现。随后，随着通用时钟框架的引入，内核需要支持高精度的定时器，为此，通用时间框架为定时器硬件定义了一个标准的接口：clock_event_device，machine级的代码只要按这个标准接口实现相应的硬件控制功能，剩下的与平台无关的特性则统一由通用时间框架层来实现。

/*

声明：本博内容均由http://blog.csdn.net/droidphone原创，转载请注明出处，谢谢！

*/

1. 时钟事件软件架构

本系列文章的第一节中，我们曾经讨论了时钟源设备：clocksource，现在又来一个时钟事件设备：clock_event_device，它们有何区别？看名字，好像都是给系统提供时钟的设备，实际上，clocksource不能被编程，没有产生事件的能力，它主要被用于timekeeper来实现对真实时间进行精确的统计，而clock_event_device则是可编程的，它可以工作在周期触发或单次触发模式，系统可以对它进行编程，以确定下一次事件触发的时间，clock_event_device主要用于实现普通定时器和高精度定时器，同时也用于产生tick事件，供给进程调度子系统使用。时钟事件设备与通用时间框架中的其他模块的关系如下图所示：

进程调度sched

通用时间框架

普通定时器timer

tick_device

高精度定时器hrtimer

clock_event_device

硬件定时器一

硬件定时器二

硬件定时器N

时钟源1

时钟源2

Machine层

图1.1 clock_event_device软件架构

http://blog.csdn.net/droidphone/article/details/8017604

1/9

Linux时间子系统之六：irq	(46469)
Linux ALSA声卡驱动之六	(44235)
Linux ALSA声卡驱动之七	(43530)
Android Audio System 之六	(41137)
Linux ALSA声卡驱动之八	(37842)
	(36875)

评论排行	
Android Audio System 之六	(57)
Linux ALSA声卡驱动之七	(42)
Linux ALSA声卡驱动之八	(35)
Linux时间子系统之六：irq	(26)
Linux中断（interrupt）之六	(24)
Linux ALSA声卡驱动之七	(22)
Android SurfaceFlinger	(21)
Linux ALSA声卡驱动之八	(19)
Android Audio System 之七	(18)
Linux中断（interrupt）之七	(18)

推荐文章	
* 探索通用可编程数据平面	
* 这是一份很有诚意的 Protocol Buffer 语法详解	
* CSDN日报20170420 ——《开发和产品之间的恩怨从何来?》	
* Android图片加载框架最全解析——从源码的角度理解Glide的执行流程	
* 如果两个程序员差不多，选写作能力更好的那个	
* 从构造函数看线程安全	

最新评论	
Linux时间子系统之一：clock source	huanguansong: 我同意4楼的说法是正确的。另外我不理解那个10分钟以及idle状态下时间就不准的意思。
Linux ALSA声卡驱动之五：移动张鸢	我还提不出问题，怎么办呢！
Android Audio System 之一：AmazingDLC	楼主你好，我目前在做音频相关的工作，经常会接触到音频方面的代码，但是现在自己一点基础都没有（应届生）...
Linux时间子系统之一：clock source	Cowincent: 请问咱们的程序流向图使用什么软件画的
Linux ALSA声卡驱动之七：ASo ganye88	活捉一只
Linux ALSA声卡驱动之一：ALS qianxuedegushi	:" drivers 放置一些与CPU、BUS架构无关的公用代码" 是“有关”还...
Linux ALSA声卡驱动之一：ALS chenzhen1080	牛牛牛 谢谢楼主分享
Linux ALSA声卡驱动之六：ASo qq_21836933	看了很多遍，越看觉得思路越清晰，博主确实厉害，我服
Linux ALSA声卡驱动之五：移动qq_21836933	建议楼主可以在网上开课，花钱都愿意听楼主讲课！
Linux SPI总线和设备驱动架构之	

- 与clocksource一样，系统中可以存在多个clock_event_device，系统会根据它们的精度和能力，选择合适的clock_event_device对系统提供时钟事件服务。在smp系统中，为了减少处理器间的通信开销，基本上每个cpu都会具备一个属于自己的本地clock_event_device，独立地为该cpu提供时钟事件服务，smp中的每个cpu基于本地的clock_event_device，建立自己的tick_device，普通定时器和高精度定时器。
- 在软件架构上看，clock_event_device被分为了两层，与硬件相关的被放在了machine层，而与硬件无关的通用代码则被集中到了通用时间框架层，这符合内核对软件的设计需求，平台的开发者只需实现平台相关的接口即可，无需关注复杂的上层时间框架。
- tick_device是基于clock_event_device的进一步封装，用于代替原有的时钟滴答中断，给内核提供tick事件，以完成进程的调度和进程信息统计，负载均衡和时间更新等操作。

2. 时钟事件设备相关数据结构

2.1 struct clock_event_device

时钟事件设备的核心**数据结构**是clock_event_device结构，它代表着一个时钟硬件设备，该设备就好像是一个具有事件触发能力（通常就是指中断）的clocksource，它不停地计数，当计数值达到预先编程设定的数值那一刻，会引发一个时钟事件中断，继而触发该设备的事件处理回调函数，以完成对时钟事件的处理。clock_event_device的定义如下：

```
[cpp]
01. struct clock_event_device {
02.     void (*event_handler)(struct clock_event_device *);
03.     int (*set_next_event)(unsigned long evt,
04.                          struct clock_event_device *);
05.     int (*set_next_ktime)(ktime_t expires,
06.                          struct clock_event_device *);
07.     ktime_t next_event;
08.     u64 max_delta_ns;
09.     u64 min_delta_ns;
10.     u32 mult;
11.     u32 shift;
12.     enum clock_event_mode mode;
13.     unsigned int features;
14.     unsigned long retries;
15.
16.     void (*broadcast)(const struct cpumask *mask);
17.     void (*set_mode)(enum clock_event_mode mode,
18.                    struct clock_event_device *);
19.     unsigned long min_delta_ticks;
20.     unsigned long max_delta_ticks;
21.
22.     const char *name;
23.     int rating;
24.     int irq;
25.     const struct cpumask *cpumask;
26.     struct list_head list;
27. } ____cacheline_aligned;
```

event_handler 该字段是一个回调函数指针，通常由通用框架层设置，在时间中断到来时，machine底层的的中断服务程序会调用该回调，框架层利用该回调实现对时钟事件的处理。

set_next_event 设置下一次时间触发的时间，使用类似于clocksource的cycle计数值（离现在的cycle差值）作为参数。

set_next_ktime 设置下一次时间触发的时间，直接使用ktime时间作为参数。

max_delta_ns 可设置的最大时间差，单位是纳秒。

min_delta_ns 可设置的最小时间差，单位是纳秒。

mult shift 与clocksource中的类似，只不过是用于把纳秒转换为cycle。

mode 该时钟事件设备的工作模式，两种主要的工作模式分别是：

- CLOCK_EVT_MODE_PERIODIC 周期触发模式，设置后按给定的周期不停地触发事件；
- CLOCK_EVT_MODE_ONESHOT 单次触发模式，只在设置好的触发时刻触发一次；

gyfkyu: 写的很好，怒赞！！不过假如楼主能写写具体spi从设备驱动的实现就更好了。这样的话你就把控制器驱动、中间...

set_mode 函数指针，用于设置时钟事件设备的工作模式。

rating 表示该设备的精度等级。

list 系统中注册的时钟事件设备用该字段挂在全局链表变量clockevent_devices上。

2.2 全局变量clockevent_devices

系统中所有注册的clock_event_device都会挂在该链表下面，它在kernel/time/clockevents.c中定义：

```
[cpp]
01. static LIST_HEAD(clockevent_devices);
```

2.3 全局变量clockevents_chain

通用时间框架初始化时会注册一个通知链（NOTIFIER），当系统中的时钟时间设备的状态发生变化时，利用该通知链通知系统的其它模块。

```
[cpp]
01. /* Notification for clock events */
02. static RAW_NOTIFIER_HEAD(clockevents_chain);
```

3. clock_event_device的初始化和注册

每一个machine，都要定义一个自己的machine_desc结构，该结构定义了该machine的一些最基本的特性，其中需要设定一个sys_timer结构指针，machine级的代码负责定义sys_timer结构，sys_timer的声明很简单：

```
[cpp]
01. struct sys_timer {
02.     void (*init)(void);
03.     void (*suspend)(void);
04.     void (*resume)(void);
05. #ifdef CONFIG_ARCH_USES_GETTIMEOFFSET
06.     unsigned long (*offset)(void);
07. #endif
08. };
```

通常，我们至少要定义它的init字段，系统初始化阶段，该init回调会被调用，该init回调函数的主要作用就是完成系统中的clocksource和clock_event_device的硬件初始化工作，以samsung的exynos4为例，在V3.4内核的代码树中，machine_desc的定义如下：

```
[cpp]
01. MACHINE_START(SMDK4412, "SMDK4412")
02.     /* Maintainer: Kukjin Kim <kgene.kim@samsung.com> */
03.     /* Maintainer: Changhwan Youn <chaos.youn@samsung.com> */
04.     .atag_offset = 0x100,
05.     .init_irq = exynos4_init_irq,
06.     .map_io = smdk4x12_map_io,
07.     .handle_irq = gic_handle_irq,
08.     .init_machine = smdk4x12_machine_init,
09.     .timer = &exynos4_timer,
10.     .restart = exynos4_restart,
11. MACHINE_END
```

定义的sys_timer是exynos4_timer，它的定义和init回调定义如下：

```
[cpp]
01. static void __init exynos4_timer_init(void)
02. {
03.     if (soc_is_exynos4210())
04.         mct_int_type = MCT_INT_SPI;
05.     else
06.         mct_int_type = MCT_INT_PPI;
07.
08.     exynos4_timer_resources();
09.     exynos4_clocksource_init();
10.     exynos4_clokevent_init();
11. }
```

```

12.
13.     struct sys_timer exynos4_timer = {
14.         .init      = exynos4_timer_init,
15.     };

```

exynos4_clockevent_init函数显然是初始化和注册clock_event_device的合适时机，在这里，它注册了一个rating为250的clock_event_device，并把它指定给cpu0：

```

[cpp]
01. static struct clock_event_device mct_comp_device = {
02.     .name      = "mct-comp",
03.     .features   = CLOCK_EVT_FEAT_PERIODIC | CLOCK_EVT_FEAT_ONESHOT,
04.     .rating     = 250,
05.     .set_next_event = exynos4_comp_set_next_event,
06.     .set_mode    = exynos4_comp_set_mode,
07. };
08. ....
09. static void exynos4_clockevent_init(void)
10. {
11.     clockevents_calc_mult_shift(&mct_comp_device, clk_rate, 5);
12.     ....
13.     mct_comp_device.cpumask = cpumask_of(0);
14.     clockevents_register_device(&mct_comp_device);
15.
16.     setup_irq(EXYNOS4_IRQ_MCT_G0, &mct_comp_event_irq);
17. }

```

因为这个阶段其它cpu核尚未开始工作，所以该clock_event_device也只是在启动阶段给系统提供服务，实际上，因为exynos4是一个smp系统，具备2-4个cpu核心，前面说过，smp系统中，通常会使用各个cpu的本地定时器来为每个cpu单独提供时钟事件服务，继续翻阅代码，在系统初始化的后段，kernel_init会被调用，它会调用smp_prepare_cpus，其中会调用percpu_timer_setup函数，在arch/arm/kernel/smp.c中，为每个cpu定义了一个clock_event_device：

```

[cpp]
01. /*
02.  * Timer (local or broadcast) support
03.  */
04. static DEFINE_PER_CPU(struct clock_event_device, percpu_clockevent);

```

percpu_timer_setup最终会调用exynos4_local_timer_setup函数完成对本地clock_event_device的初始化工作：

```

[cpp]
01. static int __cpuinit exynos4_local_timer_setup(struct clock_event_device *evt)
02. {
03.     ....
04.     evt->name = mevt->name;
05.     evt->cpumask = cpumask_of(cpu);
06.     evt->set_next_event = exynos4_tick_set_next_event;
07.     evt->set_mode = exynos4_tick_set_mode;
08.     evt->features = CLOCK_EVT_FEAT_PERIODIC | CLOCK_EVT_FEAT_ONESHOT;
09.     evt->rating = 450;
10.
11.     clockevents_calc_mult_shift(evt, clk_rate / (TICK_BASE_CNT + 1), 5);
12.     ....
13.     clockevents_register_device(evt);
14.     ....
15.     enable_percpu_irq(EXYNOS_IRQ_MCT_LOCALTIMER, 0);
16.     ....
17.     return 0;
18. }

```

由此可见，每个cpu的本地clock_event_device的rating是450，比启动阶段的250要高，显然，之前注册给cpu0的精度要高，系统会用本地clock_event_device替换掉原来分配给cpu0的clock_event_device，至于怎么替换？我们先停一停，到这里我们一直在讨论machine级别的初始化和注册，让我们回过头来，看看框架层的初始化。在继续之前，让我们看看整个clock_event_device的初始化的调用序列图：

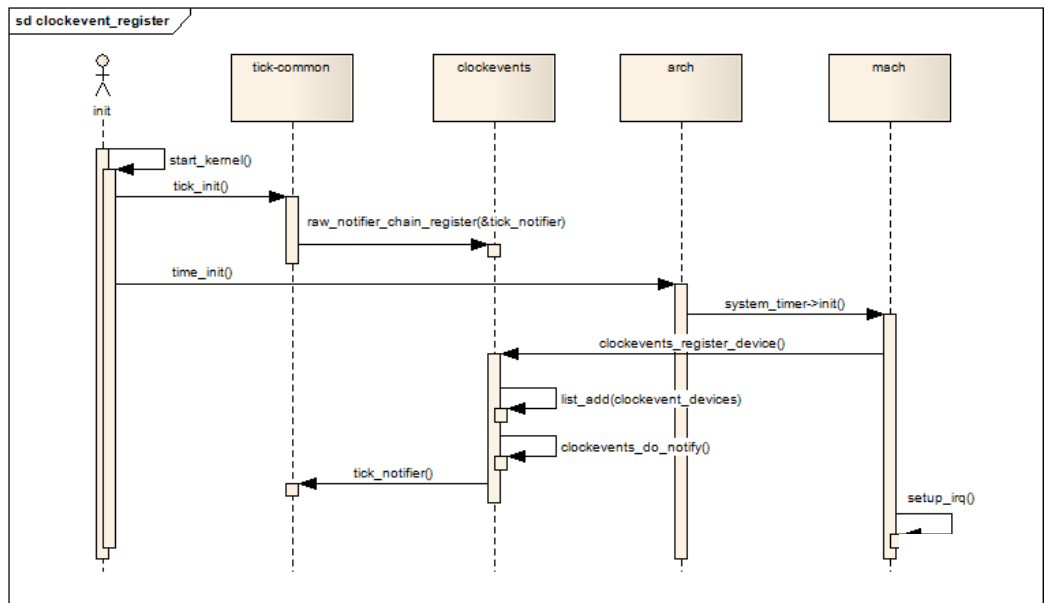


图3.1 clock_event_device的系统初始化

由上面的图示可以看出，框架层的初始化步骤很简单，又start_kernel开始，调用tick_init，它位于kernel/time/tick-common.c中，也只是简单地调用clockevents_register_notifier，同时把类型为notifier_block的tick_notifier作为参数传入，回看2.3节，clockevents_register_notifier注册了一个通知链，这样，当系统中的clock_event_device状态发生变化时（新增，删除，挂起，唤醒等等），tick_notifier中的notifier_call字段中设定的回调函数tick_notify就会被调用。接下来start_kernel调用了time_init函数，该函数通常定义在体系相关的代码中，正如前面所讨论的一样，它主要完成machine级别对时钟系统的初始化工作，最终通过clockevents_register_device注册系统中的时钟事件设备，把每个时钟时间设备挂在clockevent_device全局链表上，最后通过clockevent_do_notify触发框架层事先注册好的通知链，其实就是调用了tick_notify函数，我们主要关注CLOCK_EVT_NOTIFY_ADD通知，其它通知请自行参考代码，下面是tick_notify的简化版本：

```

[cpp]
01. static int tick_notify(struct notifier_block *nb, unsigned long reason,
02.                        void *dev)
03. {
04.     switch (reason) {
05.
06.     case CLOCK_EVT_NOTIFY_ADD:
07.         return tick_check_new_device(dev);
08.
09.     case CLOCK_EVT_NOTIFY_BROADCAST_ON:
10.     case CLOCK_EVT_NOTIFY_BROADCAST_OFF:
11.     case CLOCK_EVT_NOTIFY_BROADCAST_FORCE:
12.         .....
13.     case CLOCK_EVT_NOTIFY_BROADCAST_ENTER:
14.     case CLOCK_EVT_NOTIFY_BROADCAST_EXIT:
15.         .....
16.     case CLOCK_EVT_NOTIFY_CPU_DYING:
17.         .....
18.     case CLOCK_EVT_NOTIFY_CPU_DEAD:
19.         .....
20.     case CLOCK_EVT_NOTIFY_SUSPEND:
21.         .....
22.     case CLOCK_EVT_NOTIFY_RESUME:
23.         .....
24.     }
25.
26.     return NOTIFY_OK;
27. }
  
```

可见，对于新注册的clock_event_device，会发出CLOCK_EVT_NOTIFY_ADD通知，最终会进入函数：

tick_check_new_device，这个函数比对当前cpu所使用的与新注册的clock_event_device之间的特性，如果认为新的clock_event_device更好，则会进行切换工作。下一节将会详细的讨论该函数。到这里，每个cpu已经有了自己的clock_event_device，在这以后，框架层的代码会根据内核的配置项（CONFIG_NO_HZ、CONFIG_HIGH_RES_TIMERS），对注册的clock_event_device进行不同的设置，从而为系统的tick和高精度定时器提供服务，这些内容我们留在本系列的后续文章进行讨论。

4. tick_device

当内核没有配置成支持高精度定时器时，系统的tick由tick_device产生，tick_device其实是clock_event_device的简单封装，它内嵌了一个clock_event_device指针和它的工作模式：

```
[cpp]
01. struct tick_device {
02.     struct clock_event_device *evtdev;
03.     enum tick_device_mode mode;
04. };
```

在kernel/time/tick-common.c中，定义了一个per-cpu的tick_device全局变量，tick_cpu_device：

```
[cpp]
01. /*
02.  * Tick devices
03.  */
04. DEFINE_PER_CPU(struct tick_device, tick_cpu_device);
```

前面曾经说过，当machine的代码为每个cpu注册clock_event_device时，通知回调函数tick_notify会被调用，进而进入tick_check_new_device函数，下面让我们看看该函数如何工作，首先，该函数先判断注册的clock_event_device是否可用于本cpu，然后从per-cpu变量中取出本cpu的tick_device：

```
[cpp]
01. static int tick_check_new_device(struct clock_event_device *newdev)
02. {
03.     .....
04.     cpu = smp_processor_id();
05.     if (!cpumask_test_cpu(cpu, newdev->cpumask))
06.         goto out_bc;
07.
08.     td = &per_cpu(tick_cpu_device, cpu);
09.     curdev = td->evtdev;
```

如果不是本地clock_event_device，会做进一步的判断：如果不能把irq绑定到本cpu，则放弃处理，如果本cpu已经有了一个本地clock_event_device，也放弃处理：

```
[cpp]
01. if (!cpumask_equal(newdev->cpumask, cpumask_of(cpu))) {
02.     .....
03.     if (!irq_can_set_affinity(newdev->irq))
04.         goto out_bc;
05.     .....
06.     if (curdev && cpumask_equal(curdev->cpumask, cpumask_of(cpu)))
07.         goto out_bc;
08. }
```

反之，如果本cpu已经有了一个clock_event_device，则根据是否支持单触发模式和它的rating值，决定是否替换原来旧的clock_event_device：

```
[cpp]
01. if (curdev) {
02.     if ((curdev->features & CLOCK_EVT_FEAT_ONESHOT) &&
03.         !(newdev->features & CLOCK_EVT_FEAT_ONESHOT))
04.         goto out_bc; // 新的不支持单触发，但旧的支持，所以不能替换
05.     if (curdev->rating >= newdev->rating)
06.         goto out_bc; // 旧的比新的精度高，不能替换
07. }
```

在这些判断都通过之后，说明或者来cpu还没有绑定tick_device，或者是新的更好，需要替换：

```
[cpp]
01. if (tick_is_broadcast_device(curdev)) {
02.     clockevents_shutdown(curdev);
03.     curdev = NULL;
```

```

04.     }
05.     clockevents_exchange_device(curdev, newdev);
06.     tick_setup_device(td, newdev, cpu, cpumask_of(cpu));

```

上面的tick_setup_device函数负责重新绑定当前cpu的tick_device和新注册的clock_event_device，如果发现是当前cpu第一次注册tick_device，就把它设置为TICKDEV_MODE_PERIODIC模式，如果是替换旧的tick_device，则根据新的tick_device的特性，设置为TICKDEV_MODE_PERIODIC或TICKDEV_MODE_ONESHOT模式。可见，在系统的启动阶段，tick_device是工作在周期触发模式的，直到框架层在合适的时机，才会开启单触发模式，以便支持NO_HZ和HRTIMER。

5. tick事件的处理--最简单的情况

clock_event_device最基本的应用就是实现tick_device，然后给系统定期地产生tick事件，通用时间框架对clock_event_device和tick_device的处理相当复杂，因为涉及配置项：CONFIG_NO_HZ和CONFIG_HIGH_RES_TIMERS的组合，两个配置项就有4种组合，这四种组合的处理都有所不同，所以这里我先只讨论最简单的情况：

- CONFIG_NO_HZ == 0;
- CONFIG_HIGH_RES_TIMERS == 0;

在这种配置模式下，我们回到上一节的tick_setup_device函数的最后：

```

[cpp]
01. if (td->mode == TICKDEV_MODE_PERIODIC)
02.     tick_setup_periodic(newdev, 0);
03. else
04.     tick_setup_oneshot(newdev, handler, next_event);

```

因为启动期间，第一个注册的tick_device必然工作在TICKDEV_MODE_PERIODIC模式，所以tick_setup_periodic会设置clock_event_device的事件回调字段event_handler为tick_handle_periodic，工作一段时间后，就算有新的支持TICKDEV_MODE_ONESHOT模式的clock_event_device需要替换，再次进入tick_setup_device函数，tick_setup_oneshot的handler参数也是之前设置的tick_handle_periodic函数，所以我们考察tick_handle_periodic即可：

```

[cpp]
01. void tick_handle_periodic(struct clock_event_device *dev)
02. {
03.     int cpu = smp_processor_id();
04.     ktime_t next;
05.
06.     tick_periodic(cpu);
07.
08.     if (dev->mode != CLOCK_EVT_MODE_ONESHOT)
09.         return;
10.
11.     next = ktime_add(dev->next_event, tick_period);
12.     for (;;) {
13.         if (!clockevents_program_event(dev, next, false))
14.             return;
15.         if (timekeeping_valid_for_hres())
16.             tick_periodic(cpu);
17.         next = ktime_add(next, tick_period);
18.     }
19. }

```

该函数首先调用tick_periodic函数，完成tick事件的所有处理，如果是周期触发模式，处理结束，如果工作在单触发模式，则计算并设置下一次的触发时刻，这里用了一个循环，是为了防止当该函数被调用时，clock_event_device中的计时实际上已经经过了不止一个tick周期，这时候，tick_periodic可能被多次调用，使得jiffies和时间可以被正确地更新。tick_periodic的代码如下：

```

[cpp]
01. static void tick_periodic(int cpu)
02. {
03.     if (tick_do_timer_cpu == cpu) {
04.         write_seqlock(&xtime_lock);
05.

```



```
06.         /* Keep track of the next tick event */
07.         tick_next_period = ktime_add(tick_next_period, tick_period);
08.
09.         do_timer(1);
10.         write_sequnlock(&xtime_lock);
11.     }
12.
13.     update_process_times(user_mode(get_irq_regs()));
14.     profile_tick(CPU_PROFILING);
15. }
```

如果当前cpu负责更新时间，则通过do_timer进行以下操作：

- 更新jiffies_64变量；
- 更新墙上时钟；
- 每10个tick，更新一次cpu的负载信息；

调用update_peocess_times，完成以下事情：

- 更新进程的时间统计信息；
- 触发TIMER_SOFTIRQ软件中断，以便系统处理传统的低分辨率定时器；
- 检查rcu的callback；
- 通过scheduler_tick触发调度系统进行进程统计和调度工作；

顶

1

踩

0

上一篇

Linux时间子系统之三：时间的维护者：timekeeper

下一篇

Linux时间子系统之五：低分辨率定时器的原理和实现

我的同类文章

Linux时间管理系统（7）	Linux内核架构（14）
<ul style="list-style-type: none">• Linux时间子系统之八：动态... 2012-10-27 阅读 17738• Linux时间子系统之六：高精... 2012-10-19 阅读 44210• Linux时间子系统之三：时间... 2012-09-19 阅读 22224• Linux时间子系统之一：cloc... 2012-09-13 阅读 23621	<ul style="list-style-type: none">• Linux时间子系统之七：定时... 2012-10-23 阅读 17451• Linux时间子系统之五：低分... 2012-10-13 阅读 13904• Linux时间子系统之二：表示... 2012-09-14 阅读 20857

参考知识库



.NET知识库
3774 关注 | 833 收录



算法与数据结构知识库
15817 关注 | 2320 收录



大型网站架构知识库
8578 关注 | 708 收录

猜你在找

话说linux内核-uboot和系统移植第14部分

Linux时间子系统之七定时器的应用一

嵌入式软件调试技术专题(3)：Linux内核日志与信息打印

从零写Bootloader及移植uboot、linux内核、文件系统

Linux时间子系统之六高精度定时器HRTIMER的原理和实

嵌入式Linux内核移植

Linux时间子系统之五低分辨率定时器的原理和实现


嵌入式Linux系统移植入门

Linux时间子系统之六高精度定时器HRTIMER的原理和实

Linux时间子系统之七定时器的应用一


查看评论

4楼 [puppypyb](#) 2014-11-19 16:35发表




我发现虽然clock_event_device可以有好多，但tick_device每个CPU只能有一个

3楼 [hiupo](#) 2013-06-08 16:38发表




第4节，“如果发现是当前cpu第一次注册tick_device，就把它设置为TICKDEV_MODE_PERIODIC模式，如果是替换旧的tick_device，则根据新的tick_device的特性，设置为TICKDEV_MODE_PERIODIC或TICKDEV_MODE_ONESHOT模式”，但看tick_setup_device函数发现td->mode只在第一次注册时被置为TICKDEV_MODE_PERIODIC，后续不随newdev改变，所以上说法是否不对？

Re: [puppypyb](#) 2014-11-19 16:33发表




回复hiupo：只有启动阶段cpu0的clock_event_device才是PERIODIC模式，后续调用percpu_timer_setup为每个cpu定义的lock_event_device就都是ONESHOT模式了（也可以设置为PERIODIC，但要满足一定

2楼 [屌丝男007](#) 2013-05-15 10:55发表



1、“clocksource不能被编程，没有产生事件的能力，它主要被用于timekeeper来实现对真实时间进行精确的统计”2、“时钟事件设备的核心数据结构是clock_event_device结构，它代表着一个时钟硬件设备，该设备就好像是一个具有事件触发能力（通常就是指中断）的clocksource，它不停地计数，当计数值达到预先编程设定的数值那一刻，会引发一个时钟事件中断”1和2句话是不是前后矛盾了

1楼 [异乡客](#) 2013-04-21 21:29发表



不错的文章，这段时间正学习。多谢。

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题

Hadoop

AWS

移动游戏

Java

Android

iOS

Swift

智能硬件

Docker

OpenStack

VPN

Spark

ERP

IE10

Eclipse

CRM

JavaScript

数据库

Ubuntu

NFC

WAP

jQuery

BI

HTML5

Spring

Apache

.NET

API

HTML

SDK

IIS

Fedora

XML

LBS

Unity

Splashtop

UML

components

Windows Mobile

Rails

QEMU

KDE

Cassandra

CloudStack

FTC

coremail

OPhone

CouchBase

云计算

iOS6

Rackspace

Web App

SpringSide

Maemo

Compuware

大数据

aptech

Perl

Tornado

Ruby

Hibernate

ThinkPHP

HBase

Pure

Solr

Angular

Cloud Foundry

Redis

Scala

Django

Bootstrap

公司简介 | 招贤纳士 | 广告服务 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2016, CSDN.NET, All Rights Reserved 