

(46469)

Linux时间子系统之六：i

(44235)

Linux ALSA声卡驱动之

(43530)

Linux ALSA声卡驱动之

(41137)

Android Audio System

(37842)

Linux ALSA声卡驱动之

(36875)

评论排行

Android Audio System

(57)

Linux ALSA声卡驱动之

(42)

Linux ALSA声卡驱动之

(35)

Linux时间子系统之六：i

(26)

Linux中断（interrupt）

(24)

Linux ALSA声卡驱动之

(22)

Android SurfaceFlinger

(21)

Linux ALSA声卡驱动之

(19)

Android Audio System

(18)

Linux中断（interrupt）

(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 so

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

张鸷: 我还提不出问题，怎么办呢！
- Android Audio System 之一：A

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 so

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

qianxuedequshi: " 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

data 该字段用于上述回调函数的参数。

slack 对有些对到期时间精度不太敏感的定时器，到期时刻允许适当地延迟一小段时间，该字段用于计算每次延迟的HZ数。

要定义一个timer_list，我们可以使用静态和动态两种办法，静态方法使用DEFINE_TIMER宏：

```
#define DEFINE_TIMER(_name, _function, _expires, _data)
```

该宏将得到一个名字为_name，并分别用_function、_expires、_data参数填充timer_list的相关字段。

如果要使用动态的方法，则可以自己声明一个timer_list结构，然后手动初始化它的各个字段：

```
[cpp]
01. struct timer_list timer;
02. ....
03. init_timer(&timer);
04. timer.function = _function;
05. timer.expires = _expires;
06. timer.data = _data;
```

要激活一个定时器，我们只要调用add_timer即可：

```
[cpp]
01. add_timer(&timer);
```

要修改定时器的到期时间，我们只要调用mod_timer即可：

```
[cpp]
01. mod_timer(&timer, jiffies+50);
```

要移除一个定时器，我们只要调用del_timer即可：

```
[cpp]
01. del_timer(&timer);
```

定时器系统还提供了以下这些API供我们使用：

- void add_timer_on(struct timer_list *timer, int cpu); // 在指定的cpu上添加定时器
- int mod_timer_pending(struct timer_list *timer, unsigned long expires); // 只有当timer已经处在激活状态时，才修改timer的到期时刻
- int mod_timer_pinned(struct timer_list *timer, unsigned long expires); // 当
- void set_timer_slack(struct timer_list *time, int slack_hz); // 设定timer允许的到期时刻的最大延迟，用于对精度不敏感的定时器
- int del_timer_sync(struct timer_list *timer); // 如果该timer正在被处理中，则等待timer处理完成才移除该timer

2. 定时器的软件架构

低分辨率定时器是基于HZ来实现的，也就是说，每个tick周期，都有可能有时器到期，关于tick如何产生，请参考：Linux时间子系统之四：定时器的引擎：clock_event_device。系统中有可能有成百上千个定时器，难道在每个tick中断中遍历一下所有的定时器，检查它们是否到期？内核当然不会使用这么笨的办法，它使用了一个更聪明的办法：按定时器的到期时间对定时器进行分组。因为目前的多核处理器使用越来越广泛，连智能手机的处理器动不动就是4核心，内核对多核处理器有较好的支持，低分辨率定时器在实现时也充分地考虑了多核处理器的支持和优化。为了较好地利用cache line，也为了避免cpu之间的互锁，内核为多核处理器中的每个cpu单独分配了管理定时器的相关数据结构和资源，每个cpu独立地管理属于自己的定时器。

2.1 定时器的分组

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...

首先, 内核为每个cpu定义了一个tvec_base结构指针:

```
[cpp]
01. static DEFINE_PER_CPU(struct tvec_base *, tvec_bases) = &boot_tvec_bases;
```

tvec_base结构的定义如下:

```
[cpp]
01. struct tvec_base {
02.     spinlock_t lock;
03.     struct timer_list *running_timer;
04.     unsigned long timer_jiffies;
05.     unsigned long next_timer;
06.     struct tvec_root tv1;
07.     struct tvec tv2;
08.     struct tvec tv3;
09.     struct tvec tv4;
10.     struct tvec tv5;
11. } ____cacheline_aligned;
```

running_timer 该字段指向当前cpu正在处理的定时器所对应的timer_list结构。

timer_jiffies 该字段表示当前cpu定时器所经历过的jiffies数, 大多数情况下, 该值和jiffies计数值相等, 当cpu的idle状态连续持续了多个jiffies时间时, 当退出idle状态时, jiffies计数值就会大于该字段, 在接下来的tick中断后, 定时器系统会让该字段的值追赶上jiffies值。

next_timer 该字段指向该cpu下一个即将到期的定时器。

tv1--tv5 这5个字段用于对定时器进行分组, 实际上, tv1--tv5都是一个链表数组, 其中tv1的数组大小为TVR_SIZE, tv2 tv3 tv4 tv5的数组大小为TVN_SIZE, 根据CONFIG_BASE_SMALL配置项的不同, 它们有不同的大小:

```
[cpp]
01. #define TVN_BITS (CONFIG_BASE_SMALL ? 4 : 6)
02. #define TVR_BITS (CONFIG_BASE_SMALL ? 6 : 8)
03. #define TVN_SIZE (1 << TVN_BITS)
04. #define TVR_SIZE (1 << TVR_BITS)
05. #define TVN_MASK (TVN_SIZE - 1)
06. #define TVR_MASK (TVR_SIZE - 1)
07.
08. struct tvec {
09.     struct list_head vec[TVN_SIZE];
10. };
11.
12. struct tvec_root {
13.     struct list_head vec[TVR_SIZE];
14. };
```

默认情况下, 没有使能CONFIG_BASE_SMALL, TVR_SIZE的大小是256, TVN_SIZE的大小则是64, 当需要节省内存空间时, 也可以使能CONFIG_BASE_SMALL, 这时TVR_SIZE的大小是64, TVN_SIZE的大小则是16, 以下的讨论我都是基于没有使能CONFIG_BASE_SMALL的情况。当有一个新的定时器要加入时, 系统根据定时器到期的jiffies值和timer_jiffies字段的差值来决定该定时器被放入tv1至tv5中的哪一个数组中, 最终, 系统中所有的定时器的组织结构如下图所示:

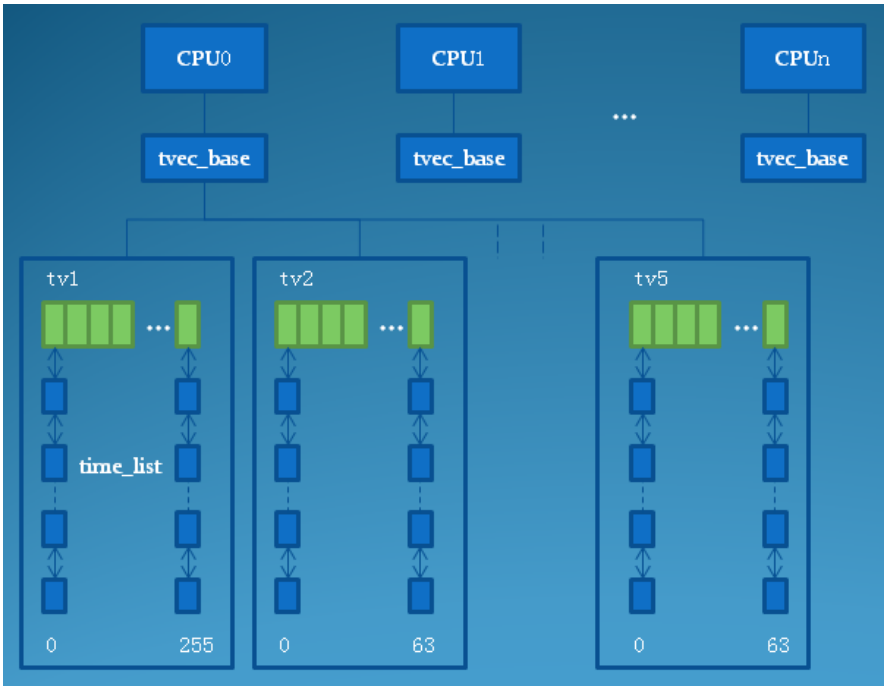


图 2.1.1 定时器在系统中的组织结构

2.2 定时器的添加

要加入一个新的定时器，我们可以通过api函数add_timer或mod_timer来完成，最终的工作会交由internal_add_timer函数来处理。该函数按以下步骤进行处理：

- 计算定时器到期时间和所属cpu的tvec_base结构中的timer_jiffies字段的差值，记为idx；
- 根据idx的值，选择该定时器应该被放到tv1--tv5中的哪一个链表数组中，可以认为tv1-tv5分别占据一个32位数的不同比特位，tv1占据最低的8位，tv2占据紧接着的6位，然后tv3再占位，以此类推，最高的6位分配给tv5。最终的选择规则如下表所示：

链表数组	idx范围
tv1	0-255(2^8)
tv2	256--16383(2^{14})
tv3	16384--1048575(2^{20})
tv4	1048576--67108863(2^{26})
tv5	67108864--4294967295(2^{32})

确定链表数组后，接着要确定把该定时器放入数组中的哪一个链表中，如果时间差idx小于256，按规则要放入tv1中，因为tv1包含了256个链表，所以可以简单地使用timer_list.expires的低8位作为数组的索引下标，把定时器链接到tv1中相应的链表中即可。如果时间差idx的值在256--18383之间，则需要把定时器放入tv2中，同样的，使用timer_list.expires的8--14位作为数组的索引下标，把定时器链接到tv2中相应的链表中。定时器要加入tv3 tv4 tv5使用同样的原理。经过这样分组后的定时器，在后续的tick事件中，系统可以很方便地定位并取出相应的到期定时器进行处理。以上的讨论都体现在internal_add_timer的代码中：

```
[cpp]
01. static void internal_add_timer(struct tvec_base *base, struct timer_list *timer)
02. {
03.     unsigned long expires = timer->expires;
04.     unsigned long idx = expires - base->timer_jiffies;
05.     struct list_head *vec;
06.
07.     if (idx < TVR_SIZE) {
08.         int i = expires & TVR_MASK;
09.         vec = base->tv1.vec + i;
10.     } else if (idx < 1 << (TVR_BITS + TVN_BITS)) {
11.         int i = (expires >> TVR_BITS) & TVN_MASK;
12.         vec = base->tv2.vec + i;
13.     } else if (idx < 1 << (TVR_BITS + 2 * TVN_BITS)) {
14.         int i = (expires >> (TVR_BITS + TVN_BITS)) & TVN_MASK;
15.         vec = base->tv3.vec + i;
```

```

16.     } else if (idx < 1 << (TVR_BITS + 3 * TVN_BITS)) {
17.         int i = (expires >> (TVR_BITS + 2 * TVN_BITS)) & TVN_MASK;
18.         vec = base->tv4.vec + i;
19.     } else if ((signed long) idx < 0) {
20.         .....
21.     } else {
22.         .....
23.         i = (expires >> (TVR_BITS + 3 * TVN_BITS)) & TVN_MASK;
24.         vec = base->tv5.vec + i;
25.     }
26.     list_add_tail(&timer->entry, vec);
27. }

```

2.2 定时器的到期处理

经过2.1节的处理后，系统中的定时器按到期时间有规律地放置在tv1--tv5各个链表数组中，其中tv1中放置着在接下来的256个jiffies即将到期的定时器列表，需要注意的是，并不是tv1.vec[0]中放置着马上到期的定时器列表，tv1.vec[1]中放置着将在jiffies+1到期的定时器列表。因为base.timer_jiffies的值一直在随着系统的运行而动态地增加，原则上是每个tick事件会加1，base.timer_jiffies代表者该cpu定时器系统当前时刻，定时器也是：256个链表tv1中，按2.1节的讨论，定时器加入tv1中使用的下标索引是定时器到期时间expires的低8当前的base.timer_jiffies值是0x34567826，则马上到期的定时器是在tv1.vec[0x26]中，如果这时候系统加入一个在jiffies值0x34567828到期的定时器，他将会加入到tv1.vec[0x28]中，运行两个tick后，base.timer_jiffies的值会变为0x34567828，很显然，在每次tick事件中，定时器系统只要以base.timer_jiffies的低8位作为索引，取出tv1中相应的链表，里面正好包含了所有在该jiffies值到期的定时器列表。

那什么时候处理tv2--tv5中的定时器？每当base.timer_jiffies的低8位为0值时，这表明base.timer_jiffies的第8-13位有进位发生，这6位正好代表着tv2，这时只要按base.timer_jiffies的第8-13位的值作为下标，移出tv2中对应的定时器链表，然后用internal_add_timer把它们从新加入到定时器系统中来，因为这些定时器一定会在接下来的256个tick期间到期，所以它们肯定会被加入到tv1数组中，这样就完成了tv2往tv1迁移的过程。同样地，当base.timer_jiffies的第8-13位为0时，这表明base.timer_jiffies的第14-19位有进位发生，这6位正好代表着tv3，按base.timer_jiffies的第14-19位的值作为下标，移出tv3中对应的定时器链表，然后用internal_add_timer把它们从新加入到定时器系统中来，显然它们会被加入到tv2中，从而完成tv3到tv2的迁移，tv4，tv5的处理可以以此类推。具体迁移的代码如下，参数index为事先计算好的高一级tv的需要迁移的数组索引：

```

[cpp]
01. static int cascade(struct tvec_base *base, struct tvec *tv, int index)
02. {
03.     /* cascade all the timers from tv up one level */
04.     struct timer_list *timer, *tmp;
05.     struct list_head tv_list;
06.
07.     list_replace_init(tv->vec + index, &tv_list); // 移除需要迁移的链表
08.
09.     /*
10.      * We are removing _all_ timers from the list, so we
11.      * don't have to detach them individually.
12.      */
13.     list_for_each_entry_safe(timer, tmp, &tv_list, entry) {
14.         BUG_ON(tbase_get_base(timer->base) != base);
15.         // 重新加入到定时器系统中, 实际上将会迁移到下一级的tv数组中
16.         internal_add_timer(base, timer);
17.     }
18.
19.     return index;
20. }

```

每个tick事件到来时，内核会在tick定时中断处理期间激活定时器软中断：TIMER_SOFTIRQ，关于软件中断，请参考另一篇博文：[Linux中断（interrupt）子系统之五：软件中断（softIRQ）](#)。TIMER_SOFTIRQ的执行函数是__run_timers，它实现了本节讨论的逻辑，取出tv1中到期的定时器，执行定时器的回调函数，由此可见，低分辨率定时器的回调函数是执行在软件中断上下文中的，这点在写定时器的回调函数时需要注意。__run_timers的代码如下：

```

[cpp]
01. static inline void __run_timers(struct tvec_base *base)
02. {
03.     struct timer_list *timer;

```

```

04.
05.     spin_lock_irq(&base->lock);
06.     /* 同步jiffies, 在NO_HZ情况下, base->timer_jiffies可能落后不止一个tick */
07.     while (time_after_eq(jiffies, base->timer_jiffies)) {
08.         struct list_head work_list;
09.         struct list_head *head = &work_list;
10.         /* 计算到期定时器链表在tv1中的索引 */
11.         int index = base->timer_jiffies & TVR_MASK;
12.
13.         /*
14.          * /* tv2--tv5定时器列表迁移处理 */
15.          */
16.         if (!index &&
17.             (!cascade(base, &base->tv2, INDEX(0))) &&
18.             (!cascade(base, &base->tv3, INDEX(1))) &&
19.             !cascade(base, &base->tv4, INDEX(2)))
20.             cascade(base, &base->tv5, INDEX(3));
21.         /* 该cpu定时器系统运行时间递增一个tick */
22.         ++base->timer_jiffies;
23.         /* 取出到期的定时器链表 */
24.         list_replace_init(base->tv1.vec + index, &work_list);
25.         /* 遍历所有的到期定时器 */
26.         while (!list_empty(head)) {
27.             void (*fn)(unsigned long);
28.             unsigned long data;
29.
30.             timer = list_first_entry(head, struct timer_list, entry);
31.             fn = timer->function;
32.             data = timer->data;
33.
34.             timer_stats_account_timer(timer);
35.
36.             base->running_timer = timer; /* 标记正在处理的定时器 */
37.             detach_timer(timer, 1);
38.
39.             spin_unlock_irq(&base->lock);
40.             call_timer_fn(timer, fn, data); /* 调用定时器的回调函数 */
41.             spin_lock_irq(&base->lock);
42.         }
43.     }
44.     base->running_timer = NULL;
45.     spin_unlock_irq(&base->lock);
46. }

```

通过上面的讨论，我们可以发现，内核的低分辨率定时器的实现非常精妙，既实现了大量定时器的管理，又实现了快速的O(1)查找到期定时器的能力，利用巧妙的数组结构，使得只需在间隔256个tick时间才处理一次迁移操作，5个数组就好比是5个齿轮，它们随着base->timer_jiffies的增长而不停地转动，每次只需处理第一个齿轮的某一个齿，低一级的齿轮转动一圈，高一级的齿轮转动一个齿，同时自动把即将到期的定时器迁移到上一个齿轮中，所以低分辨率定时器通常又被叫做时间轮：time wheel。事实上，它的实现是一个很好的空间换时间软件算法。

3. 定时器软件中断

系统初始化时，start_kernel会调用定时器系统的初始化函数init_timers：

```

[cpp]
01. void __init init_timers(void)
02. {
03.     int err = timer_cpu_notify(&timers_nb, (unsigned long)CPU_UP_PREPARE,
04.                               (void *) (long)smp_processor_id());
05.
06.     init_timer_stats();
07.
08.     BUG_ON(err != NOTIFY_OK);
09.     register_cpu_notifier(&timers_nb); /* 注册cpu notify, 以便在hotplug时在cpu之间进行定时器的迁
移 */
10.     open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
11. }

```

可见，open_softirq把run_timer_softirq注册为TIMER_SOFTIRQ的处理函数，另外，当cpu的每个tick事件到来时，在事件处理中断中，update_process_times会被调用，该函数会进一步调用run_local_timers，run_local_timers会触发TIMER_SOFTIRQ软中断：

```
[cpp]
01. void run_local_timers(void)
02. {
03.     hrtimer_run_queues();
04.     raise_softirq(TIMER_SOFTIRQ);
05. }
```

TIMER_SOFTIRQ的处理函数是run_timer_softirq：

```
[cpp]
01. static void run_timer_softirq(struct softirq_action *h)
02. {
03.     struct tvec_base *base = __this_cpu_read(tvec_bases);
04.
05.     hrtimer_run_pending();
06.
07.     if (time_after_eq(jiffies, base->timer_jiffies))
08.         __run_timers(base);
09. }
```

好啦，终于看到__run_timers函数了，2.2节已经介绍过，正是这个函数完成了对到期定时器的处理工作，也完成了时间轮的不停转动。

顶

2

踩

0

上一篇

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

下一篇

Linux时间子系统之六：高精度定时器（HRTIMER）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）
<div>Linux时间子系统之八：动态... 2012-10-27 阅读 17738</div> <div>Linux时间子系统之六：高精... 2012-10-19 阅读 44210</div> <div>Linux时间子系统之三：时间... 2012-09-19 阅读 22224</div> <div>Linux时间子系统之一：cloc... 2012-09-13 阅读 23621</div>	<div>Linux时间子系统之七：定时... 2012-10-23 阅读 17451</div> <div>Linux时间子系统之四：定时... 2012-09-28 阅读 16802</div> <div>Linux时间子系统之二：表示... 2012-09-14 阅读 20857</div>

参考知识库



.NET知识库
3774 关注 | 833 收录



Linux知识库
11722 关注 | 3939 收录



人工智能规划与决策知识库
403 关注 | 8 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

时间子系统3_低分辨率定时框架初始化

Linux时间子系统之六高精度定时器HRTIMER的原理和实


Linux时间子系统之六高精度定时器HRTIMER的原理和实

Linux时间子系统之六高精度定时器HRTIMER的原理和实

Linux时间子系统之六高精度定时器HRTIMER的原理和实

查看评论

2楼 [super程序猿](#) 2015-06-18 15:53发表




在timer handle里面在设置下次触发的时间，这样就是一个multi-times的timer。

```
static void timer_handle(unsigned long arg)
{
    ...
    do something
    ...

    /* set next trig */
    mod_timer(&g_timer, jiffies + (HZ * ms) / 1000);
}
```

1楼 [happy08god](#) 2014-04-14 21:32发表



写得太好了！！

站在应用的角度，我有个问题：

BTW，不知道是否有 运行一次和多次之分？（时间到了，callback被执行）有时候应用层可能需要一个timer多次执行。

当然，应用层可以在timer时间到的处理流程里面重新启动timer。

不知道创建timer的时候是否有指定RUN_ONCE或MULTI-TIMES 这样的flag？

按照你“2.2 定时器的到期处理”的描述，是按照计算与当前时间base->timer_jiffies的差值来选择tv1还是其他。

关于MULTI-TIMES的实现，我想问的是：

是否是通过当timer指定的时间到了后，重新计算index，放入到对应的链表数组里面？重新启动timer也是这么做，对吧？

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](#)