[Linux hrtimer分析（一）](http://blog.csdn.net/walkingman321/article/details/6133171)

标签： [linux](http://www.csdn.net/tag/linux)[struct](http://www.csdn.net/tag/struct)[timer](http://www.csdn.net/tag/timer)[算法](http://www.csdn.net/tag/%e7%ae%97%e6%b3%95)[list](http://www.csdn.net/tag/list)

2011-01-12 23:44 16369人阅读 [评论](http://blog.csdn.net/walkingman321/article/details/6133171#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/walkingman321/article/details/6133171#report)

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[(?)[+]](http://blog.csdn.net/walkingman321/article/details/6133171)

本文分析了Linux2.6.29中hrtimer的实现。

Linux2.6中实现了一种新的定时器hrtimer。与传统定时器使用时间轮算法不同，hrtimer使用了红黑树算法。hrtimer本身可以配置成高精度和普通精度两种，在单CPU系统和多CPU系统中的实现也有区别。这里先分析最简单的配置成普通精度、单CPU的情况。配置成高精度的情况见后续文章。

**1.**     **时钟源的定义**

为了实现hrtimer，Linux为系统中每一个CPU定义了一个hrtimer\_cpu\_base，这个结构体的定义如下：

struct hrtimer\_cpu\_base {

       spinlock\_t                     lock;                                          // 自旋锁

       // 时钟源

       struct hrtimer\_clock\_base     clock\_base[HRTIMER\_MAX\_CLOCK\_BASES];

       // 所有到期的且配置为软中断的定时器，见下文

       struct list\_head              cb\_pending;

#ifdef CONFIG\_HIGH\_RES\_TIMERS        // 高精度配置，这里暂不考虑

       ktime\_t                         expires\_next;

       int                         hres\_active;

       unsigned long                nr\_events;

#endif

};

在hrtimer.c中，有为每个CPU具体定义hrtimer\_cpu\_base的代码：

DEFINE\_PER\_CPU(struct hrtimer\_cpu\_base, hrtimer\_bases) =

{

       .clock\_base =

       {

              {

                     .index = CLOCK\_REALTIME,

                     .get\_time = &ktime\_get\_real,

                     .resolution = KTIME\_LOW\_RES,

              },

              {

                     .index = CLOCK\_MONOTONIC,

                     .get\_time = &ktime\_get,

                     .resolution = KTIME\_LOW\_RES,

              },

       }

};

可以看出，每个CPU都必须定义两个时钟源：REAL和MONOTONIC。

REAL代表实时时钟，MONOTONIC代表单调递增时钟。两者的区别在于，当用户更改计算机时间时，REAL时钟会收到影响，但MONOTONIC不受影响。这可以从它们两个的get\_time函数指针看出来，REAL时钟指向的是ktime\_get\_real，MONOTONIC指向的是ktime\_get。

时钟源的结构体定义为struct hrtimer\_clock\_base，其中有两个域struct rb\_node       \*first和struct rb\_root   active，这两个域维护了hrtimer的红黑树。也就是说，每一个hrtimer\_clock\_base都维护了自己的一个红黑树。

hrtimer在初始化时，都需要加入到某一个时钟源的红黑树中，这个时钟源要么是REAL，要么是MONOTONIC，这个关联通过struct hrtimer的base域实现。

**2.**     **hrtimer的基本操作**

Linux的传统定时器通过时间轮算法实现（timer.c），但hrtimer通过红黑树算法实现。在struct hrtimer里面有一个node域，类型为struct rb\_node，这个域代表了hrtimer在红黑树中的位置。

**hrtimer\_start**

hrtimer\_start函数将一个hrtimer加入到一个按照到期时间排序的红黑树中，其主要流程为：

inthrtimer\_start( struct hrtimer **\*timer,** \

ktime\_t **tim**, \

const enum hrtimer\_mode **mode**);

  // 根据time和mode参数的值计算hrtimer的超时时间，并设置到timer->expire域。

       // expire设置的是绝对时间，所以如果参数mode的值为HRTIMER\_MODE\_REL（即参数tim的值为相对时间），那么需要将tim的值修正为绝对时间：

       //     expire = tim + timer->base->get\_time()。（注意本文只研究单CPU的情况）

       //调用enqueue\_hrtimer，将hrtimer加入到红黑树中。

**hrtimer的到期**

hrtimer在hrtimer\_run\_queues函数中判断是否到期执行。hrtimer\_run\_queues的调用链为：Linux的系统时钟函数->update\_process\_times->run\_local\_timers->hrtimer\_run\_queues。

void hrtimer\_run\_queues(void)

  // 判断是否是高精度模式，如果是高精度模式，立即返回。本文暂不考虑这种情况。

       // 对每一个时钟源（REAL和MONOTONIC）的红黑树，按到期先后顺序检查hrtimer，看它们是否到期（将定时器与时钟源的softirq\_time比较）。如果到期，就把这个到期的定时器取出，然后按照定时器的具体模式执行相应的操作：

         如果定时器模式为HRTIMER\_CB\_SOFTIRQ，那么将定时器搬到hrtimer\_cpu\_base的cb\_pending队列

         调用\_\_run\_hrtimer，在\_\_run\_hrtimer中执行定时器的回调函数。

在没有配置高精度模式时，cb\_pending队列中的定时器会在T\_SOFTIRQ软中断中执行。调用链为

       run\_timer\_softirq-> hrtimer\_run\_pending-> run\_hrtimer\_pending-> run\_hrtimer\_pending

**hrtimer\_cancel**

hrtimer\_cancel函数的作用是删除一个正在排队的定时器。这里分三种情况，一种是定时器已到期，并且设置了软中断模式；第二种是没有到期，还在红黑树中；第三种是定时器正在执行。

         第一种情况，定时器被挂在hrtimer\_cpu\_base的cb\_pending队列中，所以需要把它从pending队列中移出。

         第二种情况，定时器还在红黑树中，那么把它从红黑树中移出。由于本文暂时只考虑高精度没有打开的情况，所以先不研究定时器正好排在红黑树第一个时的情况（即代码中调用hrtimer\_force\_reprogram函数的部分）。

         第三种情况删除失败，hrtimer\_cancel函数会循环重试，等到定时器执行完的时候再删除。（这在多CPU系统中可能会发生）

**3.**     **未使能高精度模式时与传统timer的区别**

         传统timer使用时间轮算法，hrtimer使用红黑树算法。

         传统timer在软中断中执行，hrtimer在硬中断中执行（update\_process\_times -> run\_local\_timers -> hrtimer\_run\_queues）。如果hrtimer设置了HRTIMER\_CB\_SOFTIRQ模式，那么timer会被移到pending队列，然后再由软中断执行。

MTK中马达实例

#include <linux/hrtimer.h>

static struct hrtimer vibe\_timer;

hrtimer\_init( &vibe\_timer,CLOCK\_MONOTONIC, HRTIMER\_MODE\_REL);

vibe\_timer.function = vibrator\_timer\_func;

hrtimer\_start( &vibe\_timer,\

ktime\_set(**value / 1000**, **(value % 1000) \*1000000** ),\

HRTIMER\_MODE\_REL);

[高精度定时器使用示例。](http://blog.chinaunix.net/uid-361890-id-257337.html) 2011-04-21 17:13:19

分类： LINUX

1. 1
2. 2 #include <linux/module.h>
3. 3 #include <linux/kernel.h>
4. 4 #include <linux/hrtimer.h>
5. 5 #include <linux/jiffies.h>
6. 6
7. 7
8. 8 static struct hrtimer timer;
9. 9 ktime\_t kt;
10. 10
11. 11 static enum hrtimer\_restart hrtimer\_handler(struct hrtimer \*timer)
12. 12 {
13. 13 //kt = ktime\_set(1, 10);
14. 14 printk(" ------ I am in hrtimer -----\n");
15. 15 hrtimer\_forward(timer, timer->base->get\_time(), kt);
16. 16 return HRTIMER\_RESTART;
17. 17 }
18. 18
19. 19 static int \_\_init test\_init(void)
20. 20 {
21. 21
22. 22 pr\_info("timer resolution: %lu\n", TICK\_NSEC);
23. 23 kt = ktime\_set(1, 10); /\* 1 sec, 10 nsec \*/
24. 24 hrtimer\_init(&timer, CLOCK\_MONOTONIC, HRTIMER\_MODE\_REL);
25. 25 //hrtimer\_set\_expires(&timer, kt);
26. 26 hrtimer\_start(&timer, kt, HRTIMER\_MODE\_REL);
27. 27 timer.function = hrtimer\_handler;
28. 28
29. 29 printk("\n-------- test start ---------\n");
30. 30 return 0;
31. 31 }
32. 32
33. 33 static void \_\_exit test\_exit(void)
34. 34 {
35. 35 hrtimer\_cancel(&timer);
36. 36 printk("-------- test over ----------\n");
37. 37 return;
38. 38 }
39. 39
40. 40 MODULE\_LICENSE("GPL");
41. 41
42. 42 module\_init(test\_init);
43. 43 module\_exit(test\_exit);
44. ~

编译：  
  
  2 name = test.o  
  3 #name ?= jprobe.o  
  4 obj-m := $(name)  
  5  
  6 KERNELDIR := /home/tangyt/tmp/linux-2.6-cloud.p1020/  
  7  
  8 ARCH = powerpc  
  9 CROSS\_COMPILE = /home/tangyt/toolchain/powerpc-linux-gnu-  
 10 export ARCH CROSS\_COMPILE  
 11 #export CC  
 12  
 13 default:  
 14     make -C $(KERNELDIR) M=$(shell pwd) modules  
 15  
 16  
 17 clean:  
 18     make -C $(KERNELDIR) M=$(shell pwd) clean  
  
说明：  
1.  40 MODULE\_LICENSE("GPL");不可少，否则找不到相应的函数  
2. 为了测试需要设置了1秒10纳秒延时，此处完全可以设置纳秒级的延时