Linux时间子系统之（四）：timekeeping

作者：linuxer 发布于：2014-12-29 18:03 分类：时间子系统

一、前言

timekeeping模块是一个提供时间服务的基础模块。Linux内核提供各种time line，real time clock，monotonic clock、monotonic raw clock等，timekeeping模块就是负责跟踪、维护这些timeline的，并且向其他模块（timer相关模块、用户空间的时间服务等）提供服务，而timekeeping模块维护timeline的基础是基于clocksource模块和tick模块。通过tick模块的tick事件，可以周期性的更新time line，通过clocksource模块、可以获取tick之间更精准的时间信息。

本文熟悉介绍timekeeping的一些基础概念，接着会介绍该模块初始化的过程，此后会从上至下介绍该模块提供的服务、该模块如何和tick模块交互以及如何和clocksource模块交互，最后介绍电源管理相关的内容。

二、timekeeper核心数据定义

1、struct timekeeper数据结构解析

旧的内核定义了很多零散的全局变量来管理linux kernel中的各种系统clock，现在，内核定义的struct timekeeper数据结构来管理各种系统时钟的跟踪以及控制，定义如下：

```
struct timekeeper {
    struct clocksource    *clock; - - - - - - - - - - - - - - - - - - - - - - - - - (1)

    u32             mult; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - (2)
    u32             shift;


    cycle_t             cycle_interval; - - - - - - - - - - - - - - - - - - - - - - (3)
    cycle_t             cycle_last;
    u64             xtime_interval;
    s64             xtime_remainder;
    u32             raw_interval;

    s64             ntp_error;
    u32             ntp_error_shift;



    u64             xtime_sec; - - - - - - - - - - - - - - - - - - - - - - - - - - (4)
    u64             xtime_nsec;



    struct timespec         wall_to_monotonic; - - - - - - - - - - - - - - - - - - (5)
    ktime_t             offs_real;
    struct timespec         total_sleep_time; - - - - - - 记录系统睡眠时间
    ktime_t             offs_boot; - - - - - - - - - - - - 记录系统boot time

    struct timespec         raw_time; - - - - - - - - - - - - - - - - - - - - - - - (6)
```

```
    s32             tai_offset; - - - - - - - - - - - - - - - - - - - - - - - - - - - - (7)
    ktime_t         offs_tai;

};
```

（1）timekeeper当前使用的clocksource。这个clock应该系统中最优的那个，如果有好过当前clocksource注册入系统，那么clocksource模块会通知timekeeping模块来切换clocksource。

（2）clock source的cycle值和纳秒转换的facotr，概念和clocksource的mult和shift一致。

（3）NTP相关的成员，这里不详述了，实在是对NTP没有兴趣。

（4）CLOCK_REALTIME类型的系统时钟（其实就是墙上时钟）。我们都知道，时间就像是一条直线（line），不知道起点，也不知道终点，因此我们称之time line。time line有很多种，和如何定义0值的时间以及用什么样的刻度来度量时间相关。人类熟悉的墙上时间和linux kernel中定义的CLOCK_REALTIME都是用来描述time line的，只不过时间原点和如何度量time line上两点距离的刻度不一样。对于人类的时间，0值是耶稣诞生的时间点；对于CLOCK_REALTIME，0值是linux epoch，即1970年1月1日...。对于墙上时间，在度量的时候虽然也是基于秒的，但是人类做了grouping，因此使用了年月日时分秒的概念。这里的秒数是相对与当前分钟值内的秒数。对于linux世界中的CLOCK_REALTIME time，直接使用秒以及纳秒在当前秒内的偏移来表示。
因此，这里xtime_sec用秒这个的刻度单位来度量CLOCK_REALTIME time line上，时间原点到当前点的距离值。当然xtime_sec是一个对current time point的取整值，为了更好的精度，还需要一个纳秒表示的offset，也就是xtime_nsec。不过为了内核内部计算精度（内核对时间的计算是基于cycle的），并不是保存了时间的纳秒偏移值，而是保存了一个shift之后的值，因此，用户看来，当前时间点的值应该是距离时间原点xtime_sec + (xtime_nsec << shift)距离的那个时间点值

（5）CLOCK_MONOTONIC类型的系统时钟。这种系统时钟并没有象墙上时钟一样定义一个相对于linux epoch的值，这个成员定义了monotonic clock到real time clock的偏移，也就是说，这里的wall_to_monotonic和offs_real需要加上real time clock的时间值才能得到monotonic clock的时间值。当然，从这里成员的名字就看出来了。wall_to_monotonic和offs_real的意思是一样的，不过时间的格式不一样，用在不同的场合，以便获取性能的提升。

（6）CLOCK_MONOTONIC_RAW类型的系统时钟

（7）CLOCK_TAI类型的系统时钟。TAI（international atomic time）是原子钟，在时间的基本概念文档中，我们说过，UTC就是base TAI的，也就是说用铯133的振荡频率来定义秒的那个时钟，当然UTC还有考虑leap second以便方便广大人民群众。CLOCK_TAI类型的系统时钟就是完完全全使用铯133的振荡频率来定义秒的那个时钟，不向人类妥协。

2、全局变量

```
static struct timekeeper timekeeper;
static DEFINE_RAW_SPINLOCK(timekeeper_lock);
static seqcount_t timekeeper_seq;

static struct timekeeper shadow_timekeeper;
```

timekeeper维护了系统的所有的clock。一个全局变量（共享资源）没有锁保护怎么行，timekeeper_lock和timekeeper_seq都是用来保护timekeeper的，用在不同的场合。

shadow_timekeeper主要用在更新系统时间的过程中。在update_wall_time中，首先将时间调整值设定到shadow_timekeeper中，然后一次性的copy到真正的那个timekeeper中。这样的设计主要是可以减少持有timekeeper_seq

锁的时间（在更新系统时间的过程中），不过需要注意的是：在其他的过程中（非update_wall_time），需要sync shadow timekeeper。

三、timekeeping初始化

timekeeping初始化的代码位于timekeeping_init函数中，在系统初始化的时候（start_kernel）会调用该函数进行timekeeping的初始化。

1、从persistent clock获取当前的时间值

timekeeping模块中支持若干种system clock，这些system clock的数据保存在ram中，一旦断电，数据就丢失了。因此，在系加电启动后，会从persistent clock中中取出当前时间值（例如RTC，RTC有battery供电，因此系统断电也可以保存数据），根据情况初始化各种system clock。具体代码如下：

```
read_persistent_clock(&now); - - - - - - - - - - - - - - - - - - - - - - - （1）
if (!timespec_valid_strict(&now)) { - - - - - - - - - - - - - - - - - - - - - - （2）
    now.tv_sec = 0;
    now.tv_nsec = 0;
} else if (now.tv_sec || now.tv_nsec)
    persistent_clock_exist = true;  - - - - - - - - - - - - - - - - - - - - - （3）

read_boot_clock(&boot); - - - - - - - - - - - 概念同上
if (!timespec_valid_strict(&boot)) {
    boot.tv_sec = 0;
    boot.tv_nsec = 0;
}
```

（1）read_persistent_clock是一个和architecture相关的函数，具体如何支持可以看具体的architecture相关的代码实现。对于ARM，其实现在linux/arch/arm/kernel/time.c文件中。该函数的功能就是从系统中的HW clock（例如RTC）中获取时间信息。

（2）timespec_valid_strict用来校验一个timespec是否是有效。如何判断从RTC获取的值是有效的呢？要满足timespec中的秒数值要大于等于0，小于KTIME_SEC_MAX，纳秒值要小于NSEC_PER_SEC（10^9）。KTIME_SEC_MAX这个宏定义了ktime_t这种类型的数据可以表示的最大的秒数值，从RTC中读出的秒数值当然不能大于它，KTIME_SEC_MAX定义如下：

```
#define KTIME_MAX           ((s64)~((u64)1 << 63))
#if (BITS_PER_LONG == 64)
# define KTIME_SEC_MAX          (KTIME_MAX / NSEC_PER_SEC)
#else
# define KTIME_SEC_MAX          LONG_MAX
#endif
```

ktime_t这种数据类型占据了64 bit的size，对于64 bit的CPU和32 bit CPU上是不一样的，64 bit的CPU上定义为一个signed long long，该值直接表示了纳秒值。对于32bit CPU而言，64 bit的数据分成两个signed int类型，分别表示秒数和纳秒数。

（3）设定persistent_clock_exist flag，说明系统中存在RTC的硬件模块，timekeeping模块会和RTC模块进行交互。例如：在suspend的时候，如果该flag是true的话，RTC driver不能sleep，因为timekeeping模块还需要在resume的时候通过RTC的值恢复其时间值呢。

2、为timekeeping模块设置default的clock source

```
clock = clocksource_default_clock(); - - - - - - - - - - - - - - - - - - - - - （1）
if (clock->enable)
    clock->enable(clock); - - - - - enalbe default clocksource
tk_setup_internals(tk, clock); - - - - - - - - - - - - - - - - - - - - - - - - - （2）
```

（1）在timekeeping初始化的时候，很难选择一个最好的clock source，因为很有可能最好的那个还没有初始化呢。因此，这里的策略就是采用一个在timekeeping初始化时一定是ready的clock source，也就是基于jiffies 的那个clocksource。clocksource_default_clock定义在kernel/time/jiffies.c，是一个weak symble，如果你愿意也可以重新定义clocksource_default_clock这个函数。不过，要保证在timekeeping初始化的时候是ready的。

（2）建立default clocksource和timekeeping伙伴关系。

3、初始化real time clock、monotonic clock和monotonic raw clock

```
tk_set_xtime(tk, &now); - - - - - - - - - - - - - - - - - - - - - - - - - - - （1）
tk->raw_time.tv_sec = 0; - - - - - - - - - - - - - - - - - - - - - - - - - - - （2）
tk->raw_time.tv_nsec = 0;
if (boot.tv_sec == 0 && boot.tv_nsec == 0)
    boot = tk_xtime(tk); - - - 如果没有获取到有效的booting time，那么就选择当前的real time clock

set_normalized_timespec(&tmp, -boot.tv_sec, -boot.tv_nsec); - - - - - - - - - - （3）
tk_set_wall_to_mono(tk, tmp);

tmp.tv_sec = 0;
tmp.tv_nsec = 0;
tk_set_sleep_time(tk, tmp); - - - - - - 初始化sleep time为0
```

（1）根据从RTC中获取的时间值来初始化timekeeping中的real time clock，如果没有获取到正确的RTC时间值，那么缺省的real time（wall time）就是linux epoch。

（2）monotonic raw clock被设定为从0开始。

（3）启动时将monotonic clock设定为负的real time clock，timekeeper并没有直接保存monotonic clock，而是保存了一个wall_to_monotonic的值，这个值类似offset，real time clock加上这个offset就可以得到monotonic clock。因此，初始化的时间点上，monotonic clock实际上等于0（如果没有获取到有效的booting time）。当系统运行之后，real time clock+ wall_to_monotonic是系统的uptime，而real time clock+ wall_to_monotonic + sleep time也就是系统的boot time。

四、获取和设定当前系统时钟的时间值

1、获取monotonic clock的时间值：ktime_get和ktime_get_ts

```
ktime_t ktime_get(void)
{
    struct timekeeper *tk = &timekeeper;
    unsigned int seq;
    s64 secs, nsecs;

    do {
        seq = read_seqcount_begin(&timekeeper_seq);
        secs = tk->xtime_sec + tk->wall_to_monotonic.tv_sec; - - - - - 获取monotonic clock的秒值
        nsecs = timekeeping_get_ns(tk) + tk->wall_to_monotonic.tv_nsec; - - - 获取纳秒值

    } while (read_seqcount_retry(&timekeeper_seq, seq));

    return ktime_add_ns(ktime_set(secs, 0), nsecs); - - - - 返回一个ktime类型的时间值
}
```

一般而言，timekeeping模块是在tick到来的时候更新各种系统时钟的时间值，ktime_get调用很有可能发生在两次tick之间，这时候，仅仅依靠当前系统时钟的值精度就不甚理想了，毕竟那个时间值是per tick更新的。因此，为了获得高精度，ns值的获取是通过timekeeping_get_ns完成的，该函数获取了real time clock的当前时刻的纳秒值，而这是通过上一次的tick时候的real time clock的时间值（xtime_nsec）加上当前时刻到上一次tick之间的delta时间值计算得到的。

ktime_get_ts的概念和ktime_get是一样的，只不过返回的时间值格式不一样而已。

2、获取real time clock的时间值：ktime_get_real和ktime_get_real_ts

这两个函数的具体逻辑动作和获取monotonic clock的时间值函数是完全一样的，大家可以自己看代码分析。这里稍微提一下另外一个函数：current_kernel_time，代码如下：

```
static inline struct timespec tk_xtime(struct timekeeper *tk)
{
    struct timespec ts;

    ts.tv_sec = tk->xtime_sec;
    ts.tv_nsec = (long)(tk->xtime_nsec >> tk->shift);
    return ts;
}

struct timespec current_kernel_time(void)
{
    struct timekeeper *tk = &timekeeper;
    struct timespec now;
    unsigned long seq;

    do {
        seq = read_seqcount_begin(&timekeeper_seq);

        now = tk_xtime(tk);
    } while (read_seqcount_retry(&timekeeper_seq, seq));
```

```
    return now;
}
```

上面的代码并没有调用clocksource的read函数获取tick之间的delta时间值，因此current_kernel_time是一个粗略版本的real time clock，精度低于ktime_get_real，不过性能要好些。类似的，monotonic clock也有一个get_monotonic_coarse函数，概念类似current_kernel_time。

3、获取boot clock的时间值：ktime_get_boottime和get_monotonic_boottime

```
ktime_t ktime_get_boottime(void)
{
    struct timespec ts;

    get_monotonic_boottime(&ts);
    return timespec_to_ktime(ts);
}
```

boot clock这个系统时钟和monotonic clock有什么不同？monotonic clock是从一个固定点开始作为epoch，对于linux，就是启动的时间点，因此，monotonic clock是一个从0开始增加的clock，并且不接受用户的setting，看起来好象适合boot clock是一致的，不过它们之间唯一的差别是对系统进入suspend的处理，对于monotonic clock，它是不记录系统睡眠时间的，因此monotonic clock得到的是一个system uptime。而boot clock计算睡眠时间，直到系统reboot。

ktime_get_boottime返回ktime的时间值，get_monotonic_boottime函数返回timespec格式的时间值。

4、获取TAI clock的时间值：ktime_get_clocktai和timekeeping_clocktai

原子钟和real time clock（UTC）是类似的，只是有一个偏移而已，记录在tai_offset中。代码非常简单，大家自己阅读即可。ktime_get_clocktai返回ktime的时间值，而timekeeping_clocktai返回timespec格式的时间值。

5、设定wall time clock

```
int do_settimeofday(const struct timespec *tv)
{

……

    timekeeping_forward_now(tk); - - - 更新timekeeper至当前时间

    xt = tk_xtime(tk);
    ts_delta.tv_sec = tv->tv_sec - xt.tv_sec;
    ts_delta.tv_nsec = tv->tv_nsec - xt.tv_nsec;  - - - - 计算delta

    tk_set_wall_to_mono(tk, timespec_sub(tk->wall_to_monotonic, ts_delta));  - - 不调mono clock

    tk_set_xtime(tk, tv);  - - - 调整wall time clock

    timekeeping_update(tk, TK_CLEAR_NTP | TK_MIRROR | TK_CLOCK_WAS_SET);  - - 更tk
```

```
……
}
```

五、和clocksource模块的交互

除了直接调用clocksource的read函数之外，timekeeping和clocksource主要的交互就是change clocksource的操作了。当系统中有更高精度的clocksource的时候，会调用timekeeping_notify函数通知timekeeping模块进行clock source的切换，代码如下：

```
int timekeeping_notify(struct clocksource *clock)
{
    struct timekeeper *tk = &timekeeper;

    if (tk->clock == clock) - - - - 新的clocksource和旧的一样，不需要切换
        return 0;
    stop_machine(change_clocksource, clock, NULL);
    tick_clock_notify(); - - - - 通知tick模块，具体在其他文档中描述
    return tk->clock == clock ? 0 : -1;
}
```

stop_machine从字面上就可以知道是停掉了所有cpu上的任务（这个machine都不能对外提供服务了），只是执行一个函数，在这个场景下是change_clocksource。（为何不直接调用change_clocksource而是使用stop_machine这样的大招？现在还在思考中……）。change_clocksource主要执行的步骤包括：

（1）调用timekeeping_forward_now函数。就要更换新的clocksource了，就是旧clocksource最后再发挥一次作用。调用旧的clocksource的read函数，将最后的这段时间间隔（当前到上次read）加到real time system clock以及minitonic raw system clock上去。

（2）调用tk_setup_internals函数设定新的clocksource，disable旧的clocksource。tk_setup_internals函数代码如下：

```
static void tk_setup_internals(struct timekeeper *tk, struct clocksource *clock)
{
    cycle_t interval;
    u64 tmp, ntpinterval;
    struct clocksource *old_clock;

    old_clock = tk->clock;
    tk->clock = clock; - - - 更换为新的clocksource
    tk->cycle_last = clock->cycle_last = clock->read(clock); - - - - 更新last cycle值

    tmp = NTP_INTERVAL_LENGTH; - - - NTP interval设定的纳秒数
    tmp <<= clock->shift;
    ntpinterval = tmp; - - - - 计算remainder的时候会用到
    tmp += clock->mult/2;
    do_div(tmp, clock->mult); - - - - - - 将NTP interval的纳秒值转成新clocksource的cycle值
```

```
    if (tmp == 0)
        tmp = 1;

    interval = (cycle_t) tmp;
    tk->cycle_interval = interval;  - - - 设定新的NTP interval的cycle值

    tk->xtime_interval = (u64) interval * clock->mult; - - - - 将NTP interval的cycle值转成ns
    tk->xtime_remainder = ntpinterval - tk->xtime_interval; - - - 计算remainder
    tk->raw_interval =
        ((u64) interval * clock->mult) >> clock->shift;  - - - - - NTP interval的ns值

     if (old_clock) { - - - - - - xtime_nsec保存的是不是实际的ns值而是一个没有执行shift版本的
        int shift_change = clock->shift - old_clock->shift;
        if (shift_change < 0) - - - - - 如果新旧的shift值不一样，那么当前的xtime_nsec要修正
            tk->xtime_nsec >>= -shift_change;
        else
            tk->xtime_nsec <<= shift_change;
    }
    tk->shift = clock->shift;  - - - - - 更换新的shift factor

    tk->ntp_error = 0;
    tk->ntp_error_shift = NTP_SCALE_SHIFT - clock->shift;

    tk->mult = clock->mult; - - - - - 更换新的mult factor
}
```

由于更换了新的clocksource，一般而言，新旧clocksource的工作参数不一样，就要就导致timekeeper的一些内部的数据成员要进行更新，例如NTP interval、multi和shift facotr数值等。

（3）调用timekeeping_update函数。由于更新了clocksource，因此timekeeping模块要更新其内部数据。TK_CLEAR_NTP控制clear 旧的NTP的状态数据。TK_MIRROR用来更新shadow timekeeper，主要是为了保持和real timekeeper同步。TK_CLOCK_WAS_SET用在paravirtual clock场景中，这里就不详细描述了。

六、和tick device模块的接口

1、periodic tick

当系统采用periodic tick机制的时候，tick device模块会在周期性tick到来的时候，调用tick_periodic来进行下面的动作：

（1）如果是global tick，需要调用do_timer来修改jiffies，计算系统负荷。

（2）如果是global tick，需要调用update_wall_time来更新系统时间。timekeeping模块是按照自己的节奏来更新系统时间的，更新一般是发生在周期性tick到来的时候。如果HZ = 100的话，那么每10ms就会有一个tick事件（clockevent事件），跟的太紧，会浪费CPU，跟的太松会损失一些精度。timekeeper中的cycle_interval成员就是周期性tick的cycle interval，如果距离上次的更新还不到一个tick的时间，那么就不再更新系统时间，直接退出。

（3）调用update_process_times和profile_tick，分别更新进程时间和进行内核剖析相关的操作。

2、dynamic tick

TODO

七、timekeeping模的电源管理

1、初始化

```
static struct syscore_ops timekeeping_syscore_ops = {
    .resume         = timekeeping_resume,
    .suspend    = timekeeping_suspend,
};

static int __init timekeeping_init_ops(void)
{
    register_syscore_ops(&timekeeping_syscore_ops);
    return 0;
}

device_initcall(timekeeping_init_ops);
```

在系统初始化的过程中，会调用 timekeeping_init_ops来注册和timekeeping相关的system core operations。在旧的内核中，这部分的功能是通过sysdev class和sysdev实现的。通过sysdev class和sysdev实现的suspend和resume看起来比较笨重而且效率低，因此新的内核为某些core subsystem设计了新的基于syscore_ops 的接口。而注册的这些callback函数会在系统suspend和resume的时候，在适当的时机执行（在system suspend过程中，syscore suspend的执行非常的靠后，在那些普通的总线设备之后，对应的，system resume过程中，非常早的醒来进入工作状态）。当然，这属于电源管理子系统的内容，这篇文章就不描述了，大家可以参考suspend_enter函数。

2、suspned 回调函数

```
static int timekeeping_suspend(void)
{
    struct timekeeper *tk = &timekeeper;
    unsigned long flags;
    struct timespec         delta, delta_delta;
    static struct timespec     old_delta;

    read_persistent_clock(&timekeeping_suspend_time);  - - - - - - - - - - - - -  (1)
    if (timekeeping_suspend_time.tv_sec || timekeeping_suspend_time.tv_nsec)
        persistent_clock_exist = true;

    raw_spin_lock_irqsave(&timekeeper_lock, flags);
    write_seqcount_begin(&timekeeper_seq);
    timekeeping_forward_now(tk); - - - - - - - - - - - - - - - - - - - - - - - - - (2)
    timekeeping_suspended = 1;  - - - - - - - - - - - - - - - - - - - - - - - - - (3)

    delta = timespec_sub(tk_xtime(tk), timekeeping_suspend_time); - - - - - - - (4)
    delta_delta = timespec_sub(delta, old_delta);
    if (abs(delta_delta.tv_sec)  >= 2) {
        old_delta = delta;
    } else {
        timekeeping_suspend_time =
```

```
        timespec_add(timekeeping_suspend_time, delta_delta);
    }

    timekeeping_update(tk, TK_MIRROR); - - - - 更新shadow timekeeper
    write_seqcount_end(&timekeeper_seq);
    raw_spin_unlock_irqrestore(&timekeeper_lock, flags);

    clockevents_notify(CLOCK_EVT_NOTIFY_SUSPEND, NULL); - - - - - - - - - (5)
    clocksource_suspend(); - - - suspend系统中所有的clocksource设备
    clockevents_suspend(); - - - suspend系统中所有的clockevent设备

    return 0;
}
```

（1）一般而言，在整机suspend之后，clocksource和clockevent所依赖的底层硬件会被推入深度睡眠甚至是断电状态（当然，也有一些例外，有些clocksource会标记CLOCK_SOURCE_SUSPEND_NONSTOP flag），这时候，有些有计时能力的硬件（persistent clock），例如RTC，仍然是running状态。虽然RTC的精度不是很好，但是time keeping的动作在suspend中的时候也要继续，需要记录这一段时间的流逝。因此，这里调用read_persistent_clock将suspend时间点信息记录到timekeeping_suspend_time变量中。persistent_clock_exist变量标识系统中是否有RTC的硬件，按理说应该在timekeeping初始化的时候设定，不过也有可能在那个时刻，系统中RTC驱动还没有初始化，因此，如果这里能得到一个有效的时间值的话，也相应的更新persistent_clock_exist变量。

（2）timekeeping subsystem马上就睡下去了，临睡前，最后一次更新timekeeper的系统时钟的数据，此后，底层的硬件会停掉，硬件counter和硬件timer都会停止工作了。

（3）标记timekeeping subsystem进入suspend过程。在这个过程中的获取时间操作应该被禁止。

（4）persistent clock的精度一般没有那么好，可能只是以秒的精度在计时。因此，一次suspend/resume的过程中，read persistent clock会引入半秒的误差。为了防止连续的suspend/resume引起时间偏移，这里也考虑了real time clock和persistent clock之间的delta值。delta是本次real time clock和persistent clock之间的差值，delta_delta是两次suspend之间delta的差值，如果delta_delta大于2秒，

（5）调用clockevents_notify函数通知clockevent模块系统suspend事件。

3、resume回调函数

```
static void timekeeping_resume(void)
{
    struct timekeeper *tk = &timekeeper;
    struct clocksource *clock = tk->clock;
    unsigned long flags;
    struct timespec ts_new, ts_delta;
    cycle_t cycle_now, cycle_delta;
    bool suspendtime_found = false;

    read_persistent_clock(&ts_new); - - - - - - 通过persistent clock记录醒来的时间点

    clockevents_resume(); - - - - - - - - - - - resume系统中所有的clockevent设备
    clocksource_resume(); - - - - - - - - - - resume系统中所有的clocksource设备


    cycle_now = clock->read(clock);
```

```
    if ((clock->flags & CLOCK_SOURCE_SUSPEND_NONSTOP) &&
        cycle_now > clock->cycle_last) { - - - - - - - - - - - - - - - - - - - - - - - （1）
        u64 num, max = ULLONG_MAX;
        u32 mult = clock->mult;
        u32 shift = clock->shift;
        s64 nsec = 0;

        cycle_delta = (cycle_now - clock->cycle_last) & clock->mask; - - - 本次suspend的时间
        do_div(max, mult);
        if (cycle_delta > max) {
            num = div64_u64(cycle_delta, max);
            nsec = (((u64) max * mult) >> shift) * num;
            cycle_delta -= num * max;
        }
        nsec += ((u64) cycle_delta * mult) >> shift; - - - - 将suspend时间从cycle转换成ns

        ts_delta = ns_to_timespec(nsec); - - - - 将suspend时间从ns转换成timespec
        suspendtime_found = true;
    } else if (timespec_compare(&ts_new, &timekeeping_suspend_time) > 0) { - - - - - （2）
        ts_delta = timespec_sub(ts_new, timekeeping_suspend_time);
        suspendtime_found = true;
    }

    if (suspendtime_found)
        __timekeeping_inject_sleeptime(tk, &ts_delta); - - - - - - - - - - - - - - - - - （3）

    tk->cycle_last = clock->cycle_last = cycle_now; - - - 更新last cycle的值
    tk->ntp_error = 0;
    timekeeping_suspended = 0; - - - - 标记完成了suspend/resume过程
    timekeeping_update(tk, TK_MIRROR | TK_CLOCK_WAS_SET); - - 更新shadow timerkeeper
    write_seqcount_end(&timekeeper_seq);
    raw_spin_unlock_irqrestore(&timekeeper_lock, flags);

    touch_softlockup_watchdog();

    clockevents_notify(CLOCK_EVT_NOTIFY_RESUME, NULL); - - - 通知resume信息到clockevent
    hrtimers_resume(); - - - 高精度timer相关，另文描述
}
```

（1）如果timekeeper当前的clocksource在suspend的时候没有stop，那么有机会使用精度更高的clocksource而不是persistent clock。前提是clocksource没有溢出，因此才有了cycle_now > clock->cycle_last的判断（不过，这里要求clocksource应该有一个很长的overflow的时间）。

（2）如果没有suspend nonstop的clock，也没有关系，可以用persistent clock的时间值。

（3）调用__timekeeping_inject_sleeptime函数，具体如下：

```
static void __timekeeping_inject_sleeptime(struct timekeeper *tk,  struct timespec *delta)
{
    tk_xtime_add(tk, delta); - - - - - - 将suspend的时间加到real time clock上去
    tk_set_wall_to_mono(tk, timespec_sub(tk->wall_to_monotonic, *delta));
```

```
    tk_set_sleep_time(tk, timespec_add(tk->total_sleep_time, *delta));
    tk_debug_account_sleep_time(delta);
}
```

monotonic clock不计sleep时间，因此wall_to_monotonic要减去suspend的时间值。total_sleep_time当然需要加上suspend的时间值。

原创文章，转发请注明出处。蜗窝科技

http://www.wowotech.net/timer_subsystem/timekeeping.html