系统中有很多与时间相关的程序（比如定期执行的任务，某一时间执行的任务，推迟一段时间执行的任务）

**主要内容：**

* 系统时间
* 定时器
* 定时器相关概念
* 定时器执行流程
* 实现程序延迟的方法
* 定时器和延迟的例子

### **1. 系统时间**

系统中管理的时间有2种：实际时间和定时器。

#### **1.1  实际时间**

实际时间就是现实中钟表上显示的时间

实际时间的获取是在开机后，内核初始化时从**RTC**读取的。

内核读取这个时间后就将其放入内核中的 **xtime** 变量中，并且在系统的运行中不断更新这个值。

**注：**RTC就是实时时钟的缩写，它是用来存放系统时间的设备。一般和BIOS一样，由主板上的电池供电的，所以即使关机也可将时间保存。

实际时间存放的变量 **xtime** 在文件 kernel/time/timekeeping.c中。

/\* 按照16位对齐，其实就是2个long型的数据 \*/

struct timespec xtime \_\_attribute\_\_ ((aligned (16)));

/\* timespec结构体的定义如下， 参考 <linux/time.h> \*/

struct timespec {

\_\_kernel\_time\_t tv\_sec; /\* seconds \*/

long tv\_nsec; /\* nanoseconds \*/

};

/\* \_kernel\_time\_t 定义如下 \*/

typedef long \_\_kernel\_time\_t;

系统读写 **xtime** 时用的就是顺序锁。

/\* 写入 xtime 参考 do\_sometimeofday 方法 \*/

int do\_settimeofday(struct timespec \*tv)

{/\* 省略 。。。。 \*/

write\_seqlock\_irqsave(&xtime\_lock, flags); /\* 获取写锁 \*/

/\* 更新 xtime \*/

write\_sequnlock\_irqrestore(&xtime\_lock, flags); /\* 释放写锁 \*/

/\* 省略 。。。。 \*/

return 0;

}

/\* 读取 xtime 参考 do\_gettimeofday 方法 \*/

void do\_gettimeofday(struct timeval \*tv)

{

struct timespec now;

getnstimeofday(&now); /\* 就是在这个方法中获取读锁，并读取 xtime \*/

tv->tv\_sec = now.tv\_sec;

tv->tv\_usec = now.tv\_nsec/1000;

}

void getnstimeofday(struct timespec \*ts)

{/\* 省略 。。。。 \*/

/\* 顺序锁中读锁来循环获取 xtime，直至读取过程中 xtime 没有被改变过 \*/

do {

seq = read\_seqbegin(&xtime\_lock);

\*ts = xtime;

nsecs = timekeeping\_get\_ns();

/\* If arch requires, add in gettimeoffset() \*/

nsecs += arch\_gettimeoffset();

} while (read\_seqretry(&xtime\_lock, seq));/\* 省略 。。。。 \*/

}

上述场景中，写锁必须要优先于读锁(因为 **xtime** 必须及时更新)，而且写锁的使用者很少(一般只有系统定期更新**xtime**的线程需要持有这个锁)。

这正是 顺序锁的应用场景。

#### **1.2 定时器**

定时器是内核中主要使用的时间管理方法，通过定时器，可以有效的调度程序的执行。

### **2. 定时器**

内核中的定时器有2种，静态定时器和动态定时器。

静态定时器一般执行了一些周期性的固定工作，如定时更新系统运行时间

动态定时器顾名思义，是在需要时（一般是推迟程序执行）动态创建的定时器，使用后销毁（一般都是只用一次）。

一般我们在内核代码中使用的定时器基本都是动态定时器

### **3. 定时器相关概念**

定时器的使用中，下面3个概念非常重要：

1. HZ
2. jiffies
3. 时间中断处理程序

#### **3.1 HZ**

节拍率(HZ)是时钟中断的频率，表示的一秒内时钟中断的次数。

比如 HZ=100 表示一秒内触发100次时钟中断程序。

HZ的值一般与体系结构有关，x86 体系结构一般定义为 100，参考文件 include/asm-generic/param.h

此外，有一点需要注意，内核中使用的HZ可能和用户空间中定义的HZ值不一致，为了避免用户空间取得错误的时间，

内核中也定义了 USER\_HZ，即用户空间使用的HZ值。

一般来说，USER\_HZ 和 HZ 都是相差整数倍，内核中通过函数 **jiffies\_to\_clock\_t** 来将内核中的 jiffies转为用户空间 jiffies

/\* 参见文件： kernel/time.c \*

/

/\*

\* Convert jiffies/jiffies\_64 to clock\_t and back.

\*/

clock\_t jiffies\_to\_clock\_t(unsigned long x)

{#if (TICK\_NSEC % (NSEC\_PER\_SEC / USER\_HZ)) == 0

# if HZ < USER\_HZ

return x \* (USER\_HZ / HZ);

# else

return x / (HZ / USER\_HZ);

# endif#else

return div\_u64((u64)x \* TICK\_NSEC, NSEC\_PER\_SEC / USER\_HZ);#endif

}

EXPORT\_SYMBOL(jiffies\_to\_clock\_t);

#### **3.2 jiffies**

jiffies用来记录自系统启动以来产生的总节拍数。比如系统启动了 N 秒，那么 jiffies就为 N×HZ

jiffies的相关定义参考头文件 **<linux/jiffies.h>**  include/linux/jiffies.h

/\* 64bit和32bit的jiffies定义如下 \*/

extern u64 \_\_jiffy\_data jiffies\_64;

extern unsigned long volatile \_\_jiffy\_data jiffies;

使用定时器时一般都是以jiffies为单位来延迟程序执行的，比如延迟5个节拍后执行的话，执行时间就是 jiffies+5

32位的jiffies的最大值为 2^32-1，在使用时有可能会出现回绕的问题。

比如下面的代码：

unsigned long timeout = jiffies + HZ/2; /\* 设置超时时间为 0.5秒 \*/

while (timeout < jiffies)

{

/\* 还没有超时，继续执行任务 \*/

}

/\* 执行超时后的任务 \*/

正常情况下，上面的代码没有问题。当jiffies接近最大值的时候，就会出现回绕问题。

由于是**unsinged long**类型，所以jiffies达到最大值后会变成0然后再逐渐变大，这样就会造成死循环

为了回避回扰的问题，可以使用**<linux/jiffies.h>**头文件中提供的 **time\_after**，**time\_before**等宏

#define time\_after(a,b) \

(typecheck(unsigned long, a) && \

typecheck(unsigned long, b) && \

((long)(b) - (long)(a) < 0))#define time\_before(a,b) time\_after(b,a)

#define time\_after\_eq(a,b) \

(typecheck(unsigned long, a) && \

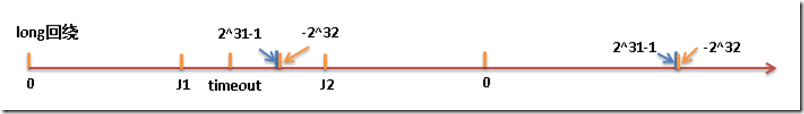
typecheck(unsigned long, b) && \

((long)(a) - (long)(b) >= 0))

#define time\_before\_eq(a,b) time\_after\_eq(b,a)

上述代码的原理其实就是将 **unsigned long** 类型转换为 **long** 类型来避免回扰带来的错误，

**long** 型的数据的回绕会出现在 2^31-1 变为 -2^32 的时候，如下图所示：

[](https://images0.cnblogs.com/blog/83005/201305/10075543-dc3a73ecb64e40e29299d6eba8d04a11.png)

1. 第一次比较时，jiffies = J1，没有超时
2. 第二次比较时，jiffies = J2，一般 J2 是负数   
   理论上 (long)timeout - (long)J2 = 正数 - 负数 = 正数（result）   
   但是，这个正数（result）一般会大于 2^31 - 1，所以long型的result又发生了一次回绕，变成了负数。 所以  
   (long)timeout - (long)J2 = 正数 - 负数 = 负数（result）

利用time\_after宏就可以巧妙的避免回绕带来的超时判断问题，将之前的代码改成如下代码即可：

unsigned long timeout = jiffies + HZ/2; /\* 设置超时时间为 0.5秒 \*/

while (time\_after(jiffies, timeout))

{

/\* 还没有超时，继续执行任务 \*/

}

/\* 执行超时后的任务 \*/

#### **3.3 时钟中断处理程序**

时钟中断处理程序作为系统定时器而注册到内核中，体系结构的不同，可能时钟中断处理程序中处理的内容不同。

但是以下这些基本的工作都会执行：

1. 获得 xtime\_lock 锁，以便对访问 jiffies\_64 和墙上时间 xtime 进行保护
2. 需要时应答或重新设置系统时钟
3. 周期性的使用墙上时间更新实时时钟
4. 调用体系结构无关的时钟例程 tick\_periodic()

tick\_periodic（）执行下面更多的工作:

1. 给jiffies\_64变量增加1 (这个操作即使是在32位体系结构上也是安全的，因为前面已经 获得了 xtime\_lock 锁)。
2. 更新资源消耗的统计值，比如当前进程所消耗的系统时间和用户时间。
3. 执行已经到期的动态定时器(11.6节将讨论)。
4. 执行第4章曾讨论的sheduler\_tick()函数。
5. 更新墙上时间，该时间存放在xtime变量中。
6. 计算平均负载值。

因为上述工作分别都由单独的函数负责完成，所以tick\_periodic例程的代码看起来非常简单。

tick\_periodic函数位于： kernel/time/tick-common.c 中

static void tick\_periodic(int cpu)

{

if (tick\_do\_timer\_cpu == cpu) {

write\_seqlock(&xtime\_lock);

/\* Keep track of the next tick event \*/

tick\_next\_period = ktime\_add(tick\_next\_period, tick\_period);

do\_timer(1);

write\_sequnlock(&xtime\_lock);

}

update\_process\_times(user\_mode(get\_irq\_regs()));

profile\_tick(CPU\_PROFILING);

}

其中最重要的是 do\_timer 和 update\_process\_times 函数。

我了解的步骤进行了简单的注释。

void do\_timer(unsigned long ticks)

{

/\* jiffies\_64 增加指定ticks \*/

jiffies\_64 += ticks;

/\* 更新实际时间 \*/

update\_wall\_time();

/\* 更新系统的平均负载值 \*/

calc\_global\_load();

}

void update\_process\_times(int user\_tick)

{

struct task\_struct \*p = current;

int cpu = smp\_processor\_id();

/\* 更新当前进程占用CPU的时间 \*/

account\_process\_tick(p, user\_tick);

/\* 同时触发软中断，处理所有到期的定时器 \*/

run\_local\_timers();

rcu\_check\_callbacks(cpu, user\_tick);

printk\_tick();

/\* 减少当前进程的时间片数 \*/

scheduler\_tick();

run\_posix\_cpu\_timers(p);

}

### **4. 定时器执行流程**

这里讨论的定时器执行流程是动态定时器的执行流程。

#### **4.1 定时器的定义**

定时器在内核中用一个链表来保存的，链表的每个节点都是一个定时器。

参见头文件 **<linux/timer.h>**

struct timer\_list {

struct list\_head entry;

unsigned long expires;

void (\*function)(unsigned long);

unsigned long data;

struct tvec\_base \*base;

#ifdef CONFIG\_TIMER\_STATS

void \*start\_site;

char start\_comm[16];

int start\_pid;

#endif

#ifdef CONFIG\_LOCKDEP

struct lockdep\_map lockdep\_map;

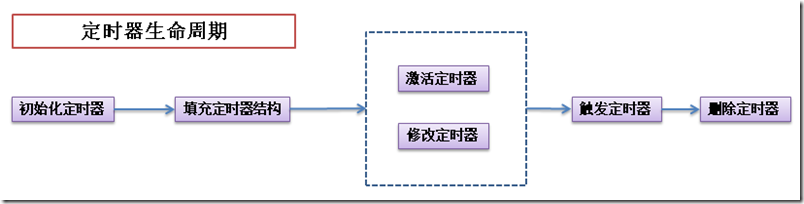
#endif

};

通过加入条件编译的参数，可以追加一些调试信息。

#### **4.2 定时器的生命周期**

一个动态定时器的生命周期中，一般会经过下面的几个步骤：

[](https://images0.cnblogs.com/blog/83005/201305/10075544-278a7b346677402a861343e0fdce0314.png)

1. 初始化定时器：

struct timer\_list my\_timer; /\* 定义定时器 \*/

init\_timer(&my\_timer); /\* 初始化定时器 \*/

2. 填充定时器：

my\_timer.expires = jiffies + delay; /\* 定义超时的节拍数 \*/

my\_timer.data = 0; /\* 给定时器函数传入的参数 \*/

my\_timer.function = my\_function; /\* 定时器超时时，执行的自定义函数 \*/

/\* 从定时器结构体中，我们可以看出这个函数的原型应该如下所示： \*/

void my\_function(unsigned long data);

3. 激活定时器和修改定时器：

激活定时器之后才会被触发，否则定时器不会执行。

修改定时器主要是修改定时器的延迟时间，修改定时器后，不管原先定时器有没有被激活，都会处于激活状态。

填充定时器结构之后，可以只激活定时器，也可以只修改定时器，也可以激活定时器后再修改定时器。

所以填充定时器结构和触发定时器之间的步骤，也就是虚线框中的步骤是不确定的。

add\_timer(&my\_timer); /\* 激活定时器 \*/

mod\_timer(&my\_timer, jiffies + new\_delay); /\* 修改定时器，设置新的延迟时间 \*/

4. 触发定时器：

每次时钟中断处理程序会检查已经激活的定时器是否超时，如果超时就执行定时器结构中的自定义函数。

5. 删除定时器：

激活和未被激活的定时器都可以被删除，已经超时的定时器会自动删除，不用特意去删除。

/\*

\* 删除激活的定时器时，此函数返回1

\* 删除未激活的定时器时，此函数返回0

\*/

del\_timer(&my\_timer);

在多核处理器上用 del\_timer 函数删除定时器时，可能在删除时正好另一个CPU核上的时钟中断处理程序正在执行这个定时器，于是就形成了竞争条件。

为了避免竞争条件，建议使用 del\_timer\_sync 函数来删除定时器。

del\_timer\_sync 函数会等待其他处理器上的定时器处理程序全部结束后，才删除指定的定时器。

/\*

\* 和del\_timer 不同，del\_timer\_sync 不能在中断上下文中执行

\*/

del\_timer\_sync(&my\_timer);

### **5. 实现程序延迟的方法**

内核中有个利用定时器实现延迟的函数 schedule\_timeout

这个函数会将当前的任务睡眠到指定时间后唤醒，所以等待时不会占用CPU时间。

/\* 将任务设置为可中断睡眠状态 \*/

set\_current\_state(TASK\_INTERRUPTIBLE);

/\* 小睡一会儿，“s“秒后唤醒 \*/

schedule\_timeout(s\*HZ);

schedule\_timeout 一般用于延迟时间较长的程序。

这里的延迟时间较长是对于计算机而言的，其实也就是延迟大于 1 个节拍(jiffies)。

对于某些极其短暂的延迟，比如只有1ms，甚至1us，1ns的延迟，必须使用特殊的延迟方法。

1s = 1000ms = 1000000us = 1000000000ns (1秒=1000毫秒=1000000微秒=1000000000纳秒)

假设 HZ=100，那么 1个节拍的时间间隔是 1/100秒，大概10ms左右。

所以对于那些极其短暂的延迟，schedule\_timeout 函数是无法使用的。

好在内核对于这些短暂，精确的延迟要求也提供了相应的宏。

/\* 具体实现参见 include/linux/delay.h

\* 以及 arch/x86/include/asm/delay.h

\*/

#define mdelay(n) ...

#define udelay(n) ...

#define ndelay(n) ...

通过这些宏，可以简单的实现延迟，比如延迟 5ns，只需 ndelay(5); 即可。

这些短延迟的实现原理并不复杂，

首先，内核在启动时就计算出了当前处理器1秒能执行多少次循环，即 loops\_per\_jiffy

(loops\_per\_jiffy 的计算方法参见 init/main.c 文件中的 calibrate\_delay 方法)。

然后算出延迟 5ns 需要循环多少次，执行那么多次空循环即可达到延迟的效果。

loops\_per\_jiffy 的值可以在启动信息中看到：

[root@vbox ~]# dmesg | grep delay

Calibrating delay loop (skipped), value calculated using timer frequency.. 6387.58 BogoMIPS (lpj=3193792)

我的虚拟机中看到 (lpj=3193792)

### **6. 定时器和延迟的例子**

下面的例子测试了短延迟，自定义定时器以及 schedule\_timeout 的使用：

#include <linux/init.h>

#include <linux/slab.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/rbtree.h>

#include <linux/time.h>

#include <linux/string.h>

#include <linux/sched.h>

#include <linux/jiffies.h>

#include <asm/param.h>

#include <linux/delay.h>

MODULE\_LICENSE("Dual BSD/GPL");

static void test\_short\_delay(void);

static void test\_delay(void);

static void test\_schedule\_timeout(void);

static void my\_delay\_function(struct timer\_list \* tlp);

static int testdelay\_init(void)

{

printk(KERN\_ALERT "HZ in current system: %dHz\n", HZ);

    /\* test short delay \*/

test\_short\_delay();

    /\* test delay \*/

test\_delay();

    /\* test schedule timeout \*/

test\_schedule\_timeout();

    return 0;

}

static void testdelay\_exit(void)

{

    printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    printk(KERN\_ALERT "testdelay is exited!\n");

    printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

// 测试短时间延迟

static void test\_short\_delay()

{

    // 输出当前节拍数，jiffies保存系统启动以来经过的节拍数

    printk(KERN\_ALERT "jiffies [b e f o r e] short delay: %lu", jiffies);

    // 延迟 5 纳秒

    ndelay(5);

    printk(KERN\_ALERT "jiffies [a f t e r] short delay: %lu", jiffies);

}

// 测试定时

static void test\_delay()

{

    /\* 初始化定时器 \*/

DEFINE\_TIMER(my\_timer, my\_delay\_function);

    /\* 填充定时器 \*/

my\_timer.expires = jiffies + 1\*HZ; /\* 2秒后超时函数执行 \*/

    /\* 激活定时器 \*/

    add\_timer(&my\_timer);

}

// 定时器回调函数

static void my\_delay\_function(struct timer\_list \* tlp)

{

    printk(KERN\_ALERT "This is my delay function start......\n");

    printk(KERN\_ALERT "The jiffies when timer is running: %lu\n", jiffies);

    printk(KERN\_ALERT "This is my delay function end........\n");

}

// 测试长时间延迟

static void test\_schedule\_timeout()

{

printk(KERN\_ALERT "This sample start at : %lu", jiffies);

    /\* 睡眠2秒 \*/

    set\_current\_state(TASK\_INTERRUPTIBLE);

    printk(KERN\_ALERT "sleep 2s ....\n");

schedule\_timeout(2\*HZ);     // 一个HZ等于1秒

    printk(KERN\_ALERT "This sample end at : %lu", jiffies);

}

module\_init(testdelay\_init);

module\_exit(testdelay\_exit);