# 定时器的使用

Linux内核使用timer\_list结构来表示一个定时器：

struct timer\_list {

struct hlist\_node entry;

unsigned long expires;

void (\*function)(unsigned long);

unsigned long data;

u32 flags;

};

成员含义如下：

**expires**：表示定时器超时时的节拍数，比如设置一个1s的定时器，那么expires=jiffies + 1\*HZ，内核也提供了一系列时间与jiffies进行转换的函数。

/\*毫秒互换\*/

int jiffies\_to\_msecs(const unsigned long j)

long msecs\_to\_jiffies(const unsigned int m)

/\*微秒互换\*/

int jiffies\_to\_usecs(const unsigned long j)

long usecs\_to\_jiffies(const unsigned int u)

/\*纳秒互换\*/

u64 jiffies\_to\_nsecs(const unsigned long j)

unsigned long nsecs\_to\_jiffies(u64 n)

**function**：定时器的超时回调函数，定时器超时时的处理就需要在这个函数中实现。

**data**：function的输入参数

内核已经提供了一系列的函数用于定时器的使用：

init\_timer(timer)

定时器的初始化，使用此函数初始化时还需要对timer成员expires/ function/data进行设置，可以直接使用setup\_timer(timer, fn, data)或者DEFINE\_TIMER(\_name, \_function, \_expires, \_data)对定时器初始化的同时对timer成员进行设置。

void add\_timer(struct timer\_list \*timer)

用于定时器的注册，也就是将定时器添加到内核动态定时器链表中，注册完毕后定时器就会开始运行。

int del\_timer(struct timer\_list \*timer)

定时器的删除，无论定时器是否正在运行，正在运行的定时器会直接停止计时。如果删除的定时器正在运行，则返回1，否则返回0。

现在我们来考虑一种情况，如果我们的timer结构是kmalloc申请的，我们想在del\_timer后kfree掉，考虑SMP系统，那么就存在一种可能性，在timer.function执行的时候我们将timer资源释放了，那么如果在function中访问了timer资源肯定就会出现问题，所以如果出现这种资源竞争情况，我们就需要等待function执行完毕才删除定时器。

int del\_timer\_sync(struct timer\_list \*timer)

等待定时器function执行完毕才删除定时器。但是这个函数使用时有很多注意事项，内核已经对注意事项做了注释。

1. 不能在中断上下文中调用此函数，因为del\_timer\_sync一直在忙等function执行完毕，如果我们在中断中调用此函数，而中断又恰好发生在正在执行function的那个核上，那个中断就会一直等待function执行完毕，而function又在等待中断执行完毕，此时就陷入了一个死锁状态。
2. 调用者不能持有会阻止定时器处理程序完成的锁，执行此函数时不能持有中断上下文持有的锁，内核针对此问题也给出了详细的注释。

\* CPU0 CPU1

\* ---- ----

\* <SOFTIRQ>

\* call\_timer\_fn();

\* base->running\_timer = mytimer;

\* spin\_lock\_irq(somelock);

\* <IRQ>

\* spin\_lock(somelock);

\* del\_timer\_sync(mytimer);

\* while (base->running\_timer == mytimer);

如果cpu0在等待function执行完毕，而cpu1正在执行function，恰好cpu1的中断将function打断，那么就会出现中断在等锁而del\_timer\_sync也在死等的死锁状态。

1. 还有两点没理解的注意事项：
2. Callers must prevent restarting of the timer,otherwise this function is meaningless.
3. The timer's handler must not call add\_timer\_on()

int mod\_timer(struct timer\_list \*timer, unsigned long expires)

修改定时器的超时时间，如果定时器没有运行，则激活此定时器。新的expires立即生效，expires时间到来后才会执行回调函数。

# 定时器的实现

# 高精度定时器的使用

定时器是基于硬件的周期性中断实现的，所以定时器的粒度为1/HZms，比如HZ为100，那么定时器的最小定时时间就为10ms，如果我将定时器的超时时间设置为小于10ms的值，最终的超时时间依然为10ms。高精度定时器hrtimer可以支持微妙级别甚至纳秒级别的定时，这取决于CPU频率。

内核对hrtimer结构定义如下：

struct hrtimer {

struct timerqueue\_node node;

ktime\_t \_softexpires;

enum hrtimer\_restart (\*function)(struct hrtimer \*);

struct hrtimer\_clock\_base \*base;

u8 state;

struct list\_head cb\_entry;

int irqsafe;

u8 is\_rel;

};

成员含义如下：

**\_softexpires** ：定时器到期时间

**function**：定时器回调函数，该函数返回一个枚举值，它决定了该hrtimer是否需要被重新激活，在回调函数返回前要手动设置下一次超时时间。其返回值的枚举值如下：

enum hrtimer\_restart {

HRTIMER\_NORESTART, /\* 不重启定时器 \*/

HRTIMER\_RESTART, /\* 重启定时器 \*/

};

**state**:用于表示hrtimer当前的状态，有以下几种：

#define HRTIMER\_STATE\_INACTIVE 0x00 // 定时器未激活

#define HRTIMER\_STATE\_ENQUEUED 0x01 // 定时器已经被排入红黑树中

#define HRTIMER\_STATE\_CALLBACK 0x02// 定时器的回调函数正在被调用

#define HRTIMER\_STATE\_MIGRATE 0x04 // 定时器正在CPU之间做迁移

内核提供了一系列对于hrtimer的操作函数：

void hrtimer\_init(struct hrtimer \*timer, clockid\_t clock\_id, enum hrtimer\_mode mode)

hrtimer的初始化。

void hrtimer\_start(struct hrtimer \*timer, ktime\_t tim, const enum hrtimer\_mode mode)

hrtimer的启动，tim为超时时间，可以使用ktime\_set(secs，nsecs)函数设置。

hrtimer\_forward\_now(struct hrtimer \*timer, ktime\_t interval)

设置超时时间，在定时器超时后可以利用次函数设置下次超时的时间

int hrtimer\_cancel(struct hrtimer \*timer)

关闭定时器并等待回调函数执行完毕，返回值为0代表定时器未激活，为1代表将激活状态定时器关闭。

# 高精度定时器的实现

# 延迟执行delayed\_work

# 延迟与睡眠

# Posix定时器实现

SYSCALL\_DEFINE3(timer\_create, const clockid\_t, which\_clock,

struct sigevent \_\_user \*, timer\_event\_spec,

timer\_t \_\_user \*, created\_timer\_id)

struct k\_clock \*kc = clockid\_to\_kclock(which\_clock);

new\_timer->it\_sigev\_notify = event.sigev\_notify;

new\_timer->sigq->info.si\_signo = event.sigev\_signo;

new\_timer->sigq->info.si\_value = event.sigev\_value;

new\_timer->sigq->info.si\_tid = new\_timer->it\_id;

new\_timer->sigq->info.si\_code = SI\_TIMER;

error = kc->timer\_create(new\_timer);

common\_timer\_set

posix\_timer\_fn

posix\_timer\_event > 0

task = pid\_task(timr->it\_pid, PIDTYPE\_PID);

ret = send\_sigqueue(timr->sigq, task, shared); > 0

int sig = q->info.si\_signo;

prepare\_signal(sig,t,false) == 0

!(sig\_ignored(p, sig, force); == 1)

!(sig\_task\_ignored(t, sig, force) == 1)

Handler = t->sighand->action[sig - 1].sa.sa\_handler;

return handler == SIG\_IGN ||

(handler == SIG\_DFL && sig\_kernel\_ignore(sig));