

bob2004

(addict)
07-09-21
10:50

📖 关于谁来调用schedule () 函数的问题?

✎ 编辑 ✎ 回复

我现在在分析2.6 kernel 里面的进程调度和抢占的代码，

我发现scheduler_tick()的主要作用就是每一个tick 进程陷入内核后，他的时间片就递减1，当变为0的时候，会设置TIF_NEED_RESCHED为1。

而schedule () 函数的作用就是 选择一个合适的进程，完成进程切换，

那么调用schedule () 函数的时机又是怎么样呢？

我知道最简单的情况就是进程主动调用schedule () 函数，让kernel来调度。

但是被动调度又是怎么样呢？

看文章是这样写的：

“被动调度

..通过在当前进程的need_resched设为1来请求调度

..设置need_resched的时机：

..当前进程用完了它的CPU时间片，update_process_times()重新进行计算

..当一个进程被唤醒，而且它的优先级比当前进程高

..当sched_setscheduler()或sched_yield()系统调用被调用时

..每次在进入一个用户态进程之前，都会检查need_resched的值，决定是否调用函数schedule()”

>> 通过在当前进程的need_resched设为1来请求调度

谁来检查这个标志呢？没有在代码里面找到啊？

>> 当一个进程被唤醒，而且它的优先级比当前进程高

这个应该就是抢占相关的部分了，也没有找到代码实现，

>> 每次在进入一个用户态进程之前，都会检查need_resched的值，决定是否调用函数schedule()”

这句话肯定是对的，但是也没有找到代码实现，

关键是进程由kernel空间返回到用户空间，这部分代码是哪里做的呢？

有认真分析过代码的朋友研究下啊。

<http://KernelChina.cublog.cn>

文章选项： 🗑 📧 📁 📄

bob2004

(addict)
07-09-21
11:30

📖 Re: 关于谁来调用schedule () 函数的问题? [re: bob2004]

✎ 编辑 ✎ 回复

看kernel代码的时候，看到了，Documentations/sched-arch.txt
看到了这么一段：

CPU idle ===== Your cpu_idle routines need to obey the following rules:

1. Preempt should now disabled over idle routines. Should only be enabled to call schedule() then disabled again.
2. need_resched/TIF_NEED_RESCHED is only ever set, and will never be cleared until the running task has called schedule(). Idle threads need only ever query need_resched, and may never set or clear it.

3. When cpu_idle finds (need_resched() == 'true'), it should call schedule(). It should not call schedule() otherwise.

应该就是这里了。看来，是cpu_idle()这个线程一直在监视着 TIF_NEED_RESCHED 这个标志，一旦某个进程的 这个标志置1，cpu_idle () 就会调用schedule () 函数了。

查了一下cpu_idle()函数，里面

```
/*
 * The idle thread. We try to conserve power, while trying to keep
 * overall latency low. The architecture specific idle is passed
 * a value to indicate the level of "idleness" of the system. //表明系统空转的等级
 */
void cpu_idle(void)
{
    local_fiq_enable();

    /* endless idle loop with no priority at all */ //没有任何的优先级
    while (1) {
        void (*idle)(void) = pm_idle;

#ifdef CONFIG_HOTPLUG_CPU
        if (cpu_is_offline(smp_processor_id())) {
            leds_event(led_idle_start);
            cpu_die();
        }
#endif

        if (!idle)
            idle = default_idle;
        leds_event(led_idle_start);
        hrtimer_stop_sched_tick();
        while (!need_resched() && !need_resched_delayed())
            idle();
        leds_event(led_idle_end);
        hrtimer_restart_sched_tick();
        local_irq_disable();
        __preempt_enable_no_resched();
        __schedule();
        preempt_disable();
        local_irq_enable();
    }
}
```

4. The only time interrupts need to be disabled when checking need_resched is if we are about to sleep the processor until the next interrupt (this doesn't provide any protection of need_resched, it prevents losing an interrupt).

4a. Common problem with this type of sleep appears to be: local_irq_disable();
if (!need_resched()) { local_irq_enable();


*** resched interrupt arrives here ***
 __asm__("sleep until next interrupt"); }
 5. TIF_POLLING_NRFLAG can be set by idle routines that do not need an interrupt to wake them up when need_resched goes high.
 In other words, they must be periodically polling need_resched, although it may be reasonable to do some background work or enter a low CPU priority.
 5a. If TIF_POLLING_NRFLAG is set, and we do decide to enter an interrupt sleep, it needs to be cleared then a memory barrier issued (followed by a test of need_resched with interrupts disabled, as explained in 3).
 arch/i386/kernel/process.c has examples of both polling and sleeping idle functions.

<http://KernelChina.cublog.cn>

编辑者: bob2004 (07-09-21 11:36)

文章选项:    

leviathan
 (old hand)
 07-09-21
 11:34

 **Re: 关于谁来调用schedule () 函数的问题?** [re: bob2004]

 回复


1和3在entry.S里头, ret_from_intr和resume_userspace。

第2个应该是default_wake_function > try_to_wake_up

 天運苟如此, 且進杯中物!

文章选项:    

bob2004
 (addict)
 07-09-21
 11:43

 **Re: 关于谁来调用schedule () 函数的问题?** [re: leviathan]


 编辑  回复

谢谢了。我再仔细分析下。

<http://KernelChina.cublog.cn>

文章选项:    

wheelz
 (Pooh-Bah)
 07-09-21
 11:55

 **Re: 关于谁来调用schedule () 函数的问题?** [re: bob2004]

 回复


当内核在某些不能进行调度的context下, 觉得应该尽快进行调度的时候, 就会设置TIF_NEED_RESCHED。你没有看过《情景分析》? 这个问题我记得在《情景分析》中讲的很清楚。

除了正常的调度点(比如返回用户态, 中断返回等等), 当内核觉得自己占用的时间太长的时候, 大多会调用cond_resched(), 去检查TIF_NEED_RESCHED, 这里如果需要, 会调用schedule()

<http://www.kernelchina.org/>

文章选项:    

bob2004
 (addict)
 07-09-21

 **Re: 关于谁来调用schedule () 函数的问题?** [re: leviathan]

 编辑  回复

16:43

>> >> 当一个进程被唤醒，而且它的优先级比当前进程高
>> 这个应该就是要抢占相关的部分了，也没有找到代码实现，

呵呵，找到了，在这里：

try_to_wake_up(struct task_struct *p, unsigned int state, int sync, int mutex)

```
/*
 * Sync wakeups (i.e. those types of wakeups where the waker
 * has indicated that it will leave the CPU in short order)
 * don't trigger a preemption, if the woken up task will run on
 * this cpu. (in this case the 'I will reschedule' promise of
 * the waker guarantees that the freshly woken up task is going
 * to be considered on this CPU.)
 */
if (!sync || cpu != this_cpu) {
    /*
     * Mutex wakeups cause no boosting:
     */
    if (mutex)
        __activate_task(p, rq);
    else
        activate_task(p, rq, cpu == this_cpu);
    if (TASK_PREEMPTS_CURR(p, rq)) {
        resched_task(rq->curr);
        trace_start_sched_wakeup(p, rq);
    }
} else {
    activate_task(p, rq, cpu == this_cpu);
    if (TASK_PREEMPTS_CURR(p, rq))
        set_tsk_need_resched_delayed(rq->curr);
}
```

TASK_PREEMPTS_CURR()

就是用来被唤醒的进程与当前CPU的运行队列上的当前进程的优先级，如果当前进程的优先级低，自然就设置 TIF_NEED_RESCHED为1

to Wheel，

正好手头有情景分析，重新翻了一下，不过2.4实现的确实比较简单啊。

这样看起来，


2.4kernel的调度时机就是进程在从系统空间返回到用户空间的前夕，执行schedule () 函数的

。

但是对于 2.6 可抢占和实时内核来说 这是不够的，必须增加多个调度点，现在的主要任务，就是搞清楚另外新增加的这些调度点。而且，2.6kernel 抢占就是抢占系统调用的（抢占的定义），马上就执行schedule () 函数了，这部分代码有人分析过吗？哦对了，我看得是打了实时patch的2.6.18.8 还有MontaVista的kernel

<http://KernelChina.cublog.cn>

文章选项：   

 Re: 关于谁来调用schedule () 函数的问题? [re: bob2004]

 编辑  回复

bob2004

(addict)
07-09-21
17:36

"可抢占内核"

如果内核不是在一个中断处理程序中，并且不在互斥的保护代码中，就认为可以进行“安全”的抢占
在释放spinlock时，或者当中断返回时，如果当前执行进程的need_resched被标记，则进行抢占式调度"

现在主要关注的就是这个地方， 现在没有看明白这个地方是怎么实现的，
上面说 “当中断返回时 ”这个好理解， 就是ret_from_intr，


那释放spinlock时候， 就检查 need_resched 标记，是怎么实现的呢？

还有就是2.6 增加了哪些调度点呢？ 除了 返回用户空间前夕和 中断返回， 还有上面说的释放spinlock时， 还有其他的时机吗？

<http://KernelChina.cublog.cn>

文章选项：   

wheelz
(Pooh-Bah)
07-09-21
18:55

 **Re: 关于谁来调用schedule () 函数的问题？** [re: bob2004]

 回复

spin_lock的时候，会preempt_disable()，
spin_unlock的时候，会preempt_enable()，而preempt_enable()会去检查TIF_NEED_RESCHED

<http://www.kernelchina.org/>

文章选项：   

bob2004
(addict)
07-09-24
14:14

 **Re: 关于谁来调用schedule () 函数的问题？** [re: wheelz]

 编辑  回复

具体代码是 include/linux/preempt.h

```
#ifdef CONFIG_PREEMPT

asmlinkage void preempt_schedule(void);

#define preempt_disable() \
do { \
    inc_preempt_count(); \
    barrier(); \
} while (0)

#define preempt_enable_no_resched() \
do { \
    barrier(); \
    dec_preempt_count(); \
} while (0)

#define preempt_check_resched() \
do { \
    if (unlikely(test_thread_flag(TIF_NEED_RESCHED))) \
        preempt_schedule(); \
} while (0)

#define preempt_enable() \
do { \
```

```
preempt_enable_no_resched(); \
barrier(); \
preempt_check_resched(); \
} while (0)


#else

#define preempt_disable()          do { } while (0)
#define preempt_enable_no_resched() do { } while (0)
#define preempt_enable()          do { } while (0)
#define preempt_check_resched()    do { } while (0)
```

<http://KernelChina.cublog.cn>

文章选项：   

bob2004
(addict)
07-09-24
15:22

 **Re: 关于谁来调用schedule () 函数的问题?** [re: bob2004]

 编辑  回复

不好意思，其实讨论了这么多，我还是有点疑问？

对于实时和抢占，肯定是为了提高效率，上面关于设置当前进程的TIF_NEED_RESCHED 标志的问题已经讨论清楚了。问题的关键在于，什么时候kernel来调度（代码就是执行schedule（）），现在看起来，就是三条路，

1> 中断返回

2> 返回到user space

3> 显式调用 preempt_enable()函数。

很明显，头两个就是2.4 或者标准2.6 的做法，

第三个，倒是可以很快的调用schedule() 函数，但是 这看起来，还是当前进程的自愿行为啊，


当一个优先级高的进程出现的时候，应该能尽快的能被调度到，否则，不就太那个了。

我再继续仔细看看代码，应该可能有些地方没有看到吧。

<http://KernelChina.cublog.cn>

文章选项：   

bob2004
(addict)
07-09-24
17:40

 **Re: 关于谁来调用schedule () 函数的问题?** [re: bob2004]

 编辑  回复

找了篇文档

href=<http://www-128.ibm.com/developerworks/cn/linux/kernel/l-kn26sch/index.html>><http://www-128.ibm.com/developerworks/cn/linux/kernel/l-kn26sch/index.html>

这里面说的比较准确，基本上回答了我的疑问，

其实我的疑问也恰恰是“内核实时补丁要解决的问题”当进程设置了TIF_NEED_RESCHED 标志，

可是什么时候被调度走呢，

怎么保证实时进程（优先级高的进程），尽快的被调度到，如果不能在有限的时间内被调度到，就不是实时了”，

ibm的这篇文章是这样解释的，其中包含了我们上面讨论的结果：

9. 调度器对内核抢占运行的支持

bob注：解释了为什么2.4kernel是不可抢占的内核？

在2.4 系统中，在核心态运行的任何进程，只有当它调用 schedule() 主动放弃控制时，调度器才有机会选择其他进程运行，因此我们说 Linux 2.4 的内核是不可抢占运行的。缺乏这一支持，核心就无法保证实时任务的及时响应，因此也就满足不了实时系统（即使是软实时）的要求。

2.6

内核实现了抢占运行，没有锁保护的代码段都有可能被中断，它的实现，对于调度技术来说，主要就是增加了调度器运行的时机。我们知道，在 2.4 内核里，调度器有两种启动方式：主动式和被动式，其中被动方式启动调度器只能是在控制从核心态返回用户态的时候，因此才有内核不可抢占的特点。

2.6 中，调度器的启动方式仍然可分为主动式和被动式两种，color=blue>所不同的是被动启动调度器的条件放宽了很多。它的修改主要在 entry.S 中：color=blue>

```
.....
ret_from_exception:                #从异常中返回的入口
    preempt_stop                    #解释为 cli, 关中断, 即从异常中返回过程中不允许抢占
ret_from_intr:                     #从中断返回的入口
    GET_THREAD_INFO(%ebp)          #取task_struct的thread_info信息
    movl EFLAGS(%esp), %eax
    movb CS(%esp), %al
    testl $(VM_MASK | 3), %eax
    jz resume_kernel               #"返回用户态"和"在核心态中返回"的分路口
ENTRY(resume_userspace)
    cli
    movl TI_FLAGS(%ebp), %ecx
    andl $_TIF_WORK_MASK, %ecx      #
    ( _TIF_NOTIFY_RESUME |
    _TIF_SIGPENDING
    # | _TIF_NEED_RESCHED )
    jne work_pending
    jmp restore_all
ENTRY(resume_kernel)
    cmpl $0, TI_PRE_COUNT(%ebp)
    jnz restore_all
    #如果preempt_count非0, 则不允许抢占
need_resched:
    movl TI_FLAGS(%ebp), %ecx
    testb $_TIF_NEED_RESCHED, %cl
    jz restore_all
    #如果没有置NEED_RESCHED位, 则不需要调度
    testl $IF_MASK, EFLAGS(%esp)
    jz restore_all                #如果关中断了, 则不允许调度
    movl $PREEMPT_ACTIVE, TI_PRE_COUNT(%ebp)
    #preempt_count 设为 PREEMPT_ACTIVE,
    通知调度器目前这次调度正处在一次抢
    #占调度中
    sti
    call schedule
    movl $0, TI_PRE_COUNT(%ebp)    #preempt_count清0
    cli
    jmp need_resched
.....
work_pending:                      #这也是从系统调用中返回时的resched入口
    testb $_TIF_NEED_RESCHED, %cl
    jz work_notifysig
    #不需要调度, 那么肯定是因为有信号需要处理才进入work_pending的
```

```

work_resched:
    call schedule
    cli
    movl TI_FLAGS(%ebp), %ecx
    andl $_TIF_WORK_MASK, %ecx
    jz restore_all                #没有work要做了，也不需要resched
    testb $_TIF_NEED_RESCHED, %cl
    jnz work_resched              #或者是需要调度，或者是有信号要处理
work_notifysig:
.....

```

现在，无论是返回用户态还是返回核心态，都有可能检查 `NEED_RESCHED` 的状态；返回核心态时，只要 `preempt_count` 为 0，即当前进程目前允许抢占，就会根据 `NEED_RESCHED` 状态选择调用 `schedule()`。在核心中，因为至少时钟中断是不断发生的，因此，只要有进程设置了当前进程的 `NEED_RESCHED` 标志，当前进程马上就有可能被抢占，而无论它是否愿意放弃 `cpu`，即使是核心进程也是如此。


所以这样看起来，2.6 kernel 实现实时的关键就在于在返回用户态之前（也就是在kernel空间），是可以发生进程调度的，比如 a 进程正在运行，B进程的优先级更高，a在某个时机（也就是上面讨论的`NEED_RESCHED` 被1的条件发生了）`TIF_NEED_RESCHED`被置1了，等下一个时钟中断发生的时候，必然就重新调度了，?br>度•飧鯁进程就有可能被调度到了，这样的时间也就控制在一个tick之内（只要b进程的priority是最高的，马上就能被scheduler调度到），这样实时才有实现的可能。

<http://KernelChina.cublog.cn>

编辑者: bob2004 (07-09-25 09:56)

文章选项:    

wheelz
(Pooh-Bah)
07-09-24
17:42

 **Re: 关于谁来调用schedule () 函数的问题?** [re: bob2004]

 回复

实时和抢占，是为了减少latency

在非抢占的内核中（比如2.4），中断返回时（包括时钟中断？），如果发现CPU是在内核态，是不进行调度的。因此，在内核态中的某些代码，如果运行时间太长的话，是会增加schedule latency的。此时，主要依靠内核本身主动进行调度。

对于可以抢占的内核，中断返回时，即使CPU是内核态，也可以调度，因此可以减少latency。此时，即使是内核态，也可以被动的调度。

但是，即使在可以抢占的内核，如果在某个锁中的运行时间太长，也会增加schedule latency，因为锁中是不能抢占的。因此，内核的很多代码，当觉得占用了较长时间，就会主动调用`cond_resched()`。

也就是说，在可抢占的内核中，没有占用锁的时候，可以抢占（中断返回的时候），这是被动的调度。占用锁的时候，主要靠内核本身主动调度。

<http://www.kernelchina.org/>

文章选项:    **littletiger**
(enthusiast)
07-09-24
18:19 **Re: 关于谁来调用schedule () 函数的问题?** [re: wheelz] 回复

占用锁, 是否就意味着preempt为disable, 这时也可以调用schedule吗?

kernel && app

<http://hi.baidu.com/littertiger>

文章选项:    **bob2004**
(addict)
07-09-24
18:26 **Re: 关于谁来调用schedule () 函数的问题?** [re: wheelz] 编辑  回复

wheelz这段结论比较经典, 真是越分析我越清楚明白了, 多谢wheeze了。

>> 因此, 在内核态中的某些代码, 如果运行时间太长的话, 是会增加schedule latency的。

>> 此时, 主要依靠内核本身主动进行调度。
能否举个例子?

>>
对于可以抢占的内核, 中断返回时, 即使CPU是内核态, 也可以调度, 因此可以减少latency。
>> 此时, 即使是内核态, 也可以被动的调度。
[这是2.6 实时抢占的前提和基础。](#)

>> 但是, 即使在可以抢占的内核, 如果在某个锁中的运行时间太长, 也会增加schedule latency, 因为锁>> 中是不能抢占的。

因此, 内核的很多代码, 当觉得占用了较长时间, 就会主动调用cond_resched

>> ()。

wheelz这里的锁肯定指的就是自旋锁了吧, 因为自旋锁是禁止了抢占的。

但是Montavista的改进是 自旋锁是可以睡眠的了, 并不是自旋的。

他们的实现是:

```
// #define spin_lock(lock)      _spin_lock(lock)
#define spin_lock(lock)        PICK_OP(_lock, lock)
// #define spin_lock_irq(lock)   PICK_OP(_lock_irq, lock) //bob test
#define PICK_OP(op, lock)
do {
    if (TYPE_EQUAL((lock), raw_spinlock_t))
        __spin##op((raw_spinlock_t *) (lock));
    else if (TYPE_EQUAL(lock, spinlock_t))
        __spin##op((spinlock_t *) (lock));
    else __bad_spinlock_type();
} while (0)
```

```
#ifdef CONFIG_PREEMPT_RT
# define _spin_lock(l)          rt_spin_lock(l)
# define _spin_lock_nested(l, s) rt_spin_lock_nested(l, s)
# define _spin_lock_bh(l)       rt_spin_lock(l)
# define _spin_lock_irq(l)      rt_spin_lock(l)
# define _spin_unlock(l)        rt_spin_unlock(l)
# define _spin_unlock_no_resched(l) rt_spin_unlock(l)
# define _spin_unlock_bh(l)     rt_spin_unlock(l)
# define _spin_unlock_irq(l)    rt_spin_unlock(l)
# define _spin_unlock_irqrestore(l, f) rt_spin_unlock(l)
```

//实时自旋锁,
void __lockfunc rt_spin_lock(spinlock_t *lock)

```
{  
    rt_spin_lock_fastlock(&lock->lock, rt_spin_lock_slowlock);  
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);  
}
```


MV的新的实现并没有禁止抢占，也就是说 mv5下的kernel spinlock是可以被抢占的，也就是说在MV5的kernel环境，还用spin_lock(spin_lock_t *) 是没有禁止抢占的，而且用信号量实现的，拿不到锁就睡眠。这样就不会导致高优先级的进程死锁了。

<http://KernelChina.cublog.cn>

编辑者: bob2004 (07-09-24 18:43)

文章选项:    

wheelz
(Pooh-Bah)
07-09-24
18:34

 **Re: 关于谁来调用schedule () 函数的问题?** [re: littletiger]


 回复

占用锁时，要先释放锁，再调度，回来后，再获得锁。

<http://www.kernelchina.org/>

文章选项:    

wheelz
(Pooh-Bah)
07-09-24
18:36

 **Re: 关于谁来调用schedule () 函数的问题?** [re: bob2004]

 回复

>> 此时，主要依靠内核本身主动进行调度。
能否举个例子？

比如，主动检查TIF_NEED_RESCHED的cond_resched()之类的东西。

<http://www.kernelchina.org/>

文章选项:    

bob2004
(addict)
07-09-25
14:26

 **Re: 关于谁来调用schedule () 函数的问题?** [re: wheelz]

 编辑  回复

这里主要是研究一下cond_resched() 函数。

查看自己最开始的学习笔记，居然看到了 cond_resched() 函数了，后来居然给忘记了。

2.6 kernel的抢占就4个级别，
CONFIG_PREEMPT_NONE=y
CONFIG_PREEMPT_VOLUNTARY is not set
CONFIG_PREEMPT_DESKTOP is not set
CONFIG_PREEMPT_RT is not set

2.6 kernel的抢占，增加了调度点的功能（也就是自愿抢占），就是利用这个函数来实现的

kernel/sched.c ==> int __sched cond_resched(void)
其实这个就是用来实现自愿抢占的（自己主动call cond_resched()函数），对应 #

CONFIG_PREEMPT_VOLUNTARY

与这个自愿抢占相关的一个函数就是 `might_sleep()` ; kernel倒是大量用到了这个函数。

```
#ifdef CONFIG_PREEMPT_VOLUNTARY
extern int cond_resched(void);
# define might_resched() cond_resched()
#else
# define might_resched() do { } while (0)
#endif
```

在 `include/linux/buffer_head.h` 中 , 就调用到了 `might_sleep()` 函数

```
/*
 * Calling wait_on_buffer() for a zero-ref buffer is illegal, so we call into
 * __wait_on_buffer() just to trip a debug check.  Because debug code in inline
 * functions is bloaty.
 */
static inline void wait_on_buffer(struct buffer_head *bh)
{
    might_sleep(); //看内核选项 , 如果CONFIG_PREEMPT_VOLUNTARY =y
    , 那这里就主动的调度一下 。
    if (buffer_locked(bh) || atomic_read(&bh->b_count) == 0)
        __wait_on_buffer(bh);
}
```

当然需要一些特定的场合来调用`cond_resched()`函数。

我在source insight里面search了一下, call这个函数还挺多的。

看起来在文件系统的代码里面调这个函数的比较多, 大概是因为读磁盘比较费时间, 而且有可能睡眠。

一般的情况都是 在`cond_resched()` 的下一行, 可能会睡眠, 要不就是耗费时间较多。比如下面的这个例子: `fs/hpfs/buffer.c`

```
/* Map a sector into a buffer and return pointers to it and to the buffer. */
void *hpfs_map_sector(struct super_block *s, unsigned secno, struct buffer_head
**bhp,
                    int ahead)
{
    struct buffer_head *bh;

    cond_resched();

    *bhp = bh = sb_bread(s, secno);
    if (bh != NULL)
        return bh->b_data;
    else {
        printk("HPFS: hpfs_map_sector: read error\n");
        return NULL;
    }
}
```


当然这种抢占和 `CONFIG_PREEMPT_RT` 肯定是不能比了。

<http://KernelChina.cublog.cn>

文章选项:    

bob2004

(addict)
 07-09-25
 16:32

 **Re: 关于谁来调用schedule () 函数的问题?** [re: bob2004]

 编辑  回复

我现在现在普通的pc上玩玩RT。

本来的干净的2.6.18.8 ,到这里下载了

href=<http://www.kernel.org/pub/linux/kernel/projects/rt/older/patch-2.6.18-rt7>>[http://](http://www.kernel.org/pub/linux/kernel/projects/rt/older/patch-2.6.18-rt7)

www.kernel.org/pub/linux/kernel/projects/rt/older/patch-2.6.18-rt7

打上后, 正常编译, 发现出现了这样的错误:

```
[root@bobzhanglinux linux-2.6.18.8-rt]# make bzImage
CHK      include/linux/version.h
CHK      include/linux/utsrelease.h
CHK      include/linux/compile.h
dnsdomainname: Host name lookup failure
CC      kernel/softirq.o
kernel/softirq.c: In function `cpu_callback':
kernel/softirq.c:838: incompatible types in assignment
kernel/softirq.c:841: warning: passing arg 1 of `wake_up_process' from incompatible
pointer type
make[1]: *** [kernel/softirq.o] Error 1
make: *** [kernel] Error 2
```

原来是softirq.c 838行出错, 看了一眼:

```
static int __cpuinit cpu_callback(struct notifier_block *nfb,
                                unsigned long action,
                                void *hcpu)
{
    int hotcpu = (unsigned long)hcpu;
    struct task_struct *p;

    switch (action) {
    case CPU_UP_PREPARE:
        p = kthread_create(ksoftirqd, hcpu, "ksoftirqd/%d", hotcpu);
        if (IS_ERR(p)) {
            printk("ksoftirqd for %i failed\n", hotcpu);
            return NOTIFY_BAD;
        }
        kthread_bind(p, hotcpu);
        per_cpu(ksoftirqd, hotcpu) = p;
        break;
    case CPU_ONLINE:
        wake_up_process(per_cpu(ksoftirqd, hotcpu));
        break;
#ifdef CONFIG_HOTPLUG_CPU
```

仔细查了一下, 原来是 per_cpu()宏的问题, 没有太看明白这个宏:

我没有选SMP, 那就该是这个:



```
#define per_cpu(var, cpu) (*(void)(cpu), &per_cpu_##var))
```

我没看明白，这个宏展开后，会和 `per_cpu(ksoftirqd, hotcpu) = p;` 一致吗？感觉不对，但是我查看标准的2.6.18.8(没有打过RT patch的)的 `cpu_callback` 也有 `color=blue>per_cpu(ksoftirqd, hotcpu) = p;` 这就奇怪了，大侠指导一下啊。

<http://KernelChina.cublog.cn>

编辑者: bob2004 (07-09-25 16:36)

文章选项:    

 加到“个人收藏夹” |  打印

[◀ 上一篇](#) [▲ 索引](#) [▶ 下一篇](#) [≡ 平坦模式](#) [📁 树状模式](#)

[前往讨论区](#)

[Contact Us](#)

[LINUXFORUM.NET](#)