

# jianglong000000的ChinaUnix博客

linuxflyer.blog.chinaunix.net

快乐求职中

首页

|

博文目录

|

关于我



jianglong000000

博客访问：28122

博文数量：59

博客积分：629

博客等级：上士

技术积分：405

用户组：普通用户

注册时间：2011-10-15 10:49

加关注

短消息

论坛

加好友

文章分类

全部博文 (59)

Android (9)

电路 (2)

杂乱无章 (6)

Linux系统 (11)

micro2440 (6)

未分配的博文 (25)

文章存档

2012年 (25)

2011年 (34)

我的朋友



bingzhea



瑾年绝恋

最近访客



12918297



s\_waitin



wd\_2014



paul719



草根老师



yx91490

## 对Linux系统休眠的理解

2012-07-11 23:12:37

分类：

原文地址：对Linux系统休眠的理解

作者：tekkamanninja

今天看了一个关于中断例程为什么不能休眠的文章，引发了我的思考。其实这个问题在学习驱动的时候早就应该解决了，但是由于5年前学驱动的时候属于Linux初学者，能力有限，所以对这个问题就知其然，没有能力知其所以然。现在回头看这个问题的时候，感觉应该可以有一个较为清晰的认识了。

首先必须意识到：休眠是一种进程的特殊状态（即task->state= TASK\_UNINTERRUPTIBLE | TASK\_INTERRUPTIBLE）]

### 一、休眠的目的

简单的说，休眠是为在一个当前进程等待暂时无法获得的资源或者一个event的到来时（原因），避免当前进程浪费CPU时间（目的），将自己放入进程等待队列中，同时让出CPU给别的进程（工作）。休眠就是为了更好地利用CPU。

一旦资源可用或event到来，将由内核代码（可能是其他进程通过系统调用）唤醒某个等待队列上的部分或全部进程。从这点来说，休眠也是一种进程间的同步机制。

### 二、休眠的对象

休眠是针对进程，也就是拥有task\_struct的独立个体。

当进程执行某个系统调用的时候，暂时无法获得的某种资源或必须等待某event的到来，在这个系统调用的底层实现代码就可以通过让系统调度的手段让出CPU，让当前进程处于休眠状态。

### 三、进程什么时候会被休眠？

进程进入休眠状态，必然是他自己的代码中调用了某个系统调用，而这个系统调用中存在休眠代码。这个休眠代码在某种条件下会被激活，从而让改变进程状态，说到底就是以各种方式包含了：

- 1、条件判断语句
- 2、进程状态改变语句
- 3、schedule();

### 三、休眠操作做了什么

进程被置为休眠，意味着它被标识为处于一个特殊的状态（TASK\_UNINTERRUPTIBLE 或 TASK\_INTERRUPTIBLE），并且从调度器的运行队列中移走。这个进程将不在任何 CPU 上被调度，即不会被运行。直到发生某些事情改变了那个状态（to TASK\_WAKING）。这时处理器重新开始执行此进程，此时进程会再次检查是否需要继续休眠（资源是否真的可用？），如果不需要就做清理工作，并将自己的状态调整为TASK\_RUNNING。过程如下图所示：

### 四、谁来唤醒休眠进程


进程在休眠后，就不再被调度器执行，就不可能由自己唤醒自己，也就是说进程不可能“睡觉睡到自然醒”。唤醒工作必然是由其他进程或者内核本身来完成的。唤醒需要改变进程的task\_struct中的状态等，代码必然在内核中，所以唤醒必然是在系统调用的实现代码中（如你驱动中的read、write方法）以及各种形式的中断代码（包括软、硬中断）中。

如果在系统调用代码中唤醒，则说明是由其他的某个进程来调用了这个系统调用唤醒了休眠的进程。

如果是中断中唤醒，那么唤醒的任务可以说是内核完成了。

  
木景风

  
gmind31

  
george.jg

微信关注



IT168企业级官微  
微信号：IT168qiye



系统架构师大会  
微信号：SACC2013

订阅

推荐博文

• 机器学习【二】：梯度下降...

• linux sort uniq awk grep 及...

• SAE 平台 Django + mysql 环...

• 交叉编译php5、nginx、squid...

• kernel 3.10内核源码分析--IO...

• Oracle索引合并coalesce操作...

• 使用10046事件查看oracle执行...

• MySQL 主从复制资料汇总...

• database replay基础学习

• “Connection refused” vs ...

热词专题

• Qt 怎样获取一个字符串中空格...

• 硬盘安装CentOS 6.0（超级详...

• Qt

• 配置hadoop2.2.0格式化namen...

• hadoop2.2.0安装手册

• 如何找到需要唤醒的进程：等待队列

上面其实已经提到了：休眠代码的一个工作就是将当前进程信息放入一个等待队列中。它其实是一个包含等待某个特定事件的所有进程相关信息的链表。一个等待队列由一个wait\_queue\_head\_t 结构体来管理，其定义在<linux/wait.h>中。

wait\_queue\_head\_t 类型的数据结构如下：

```
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```

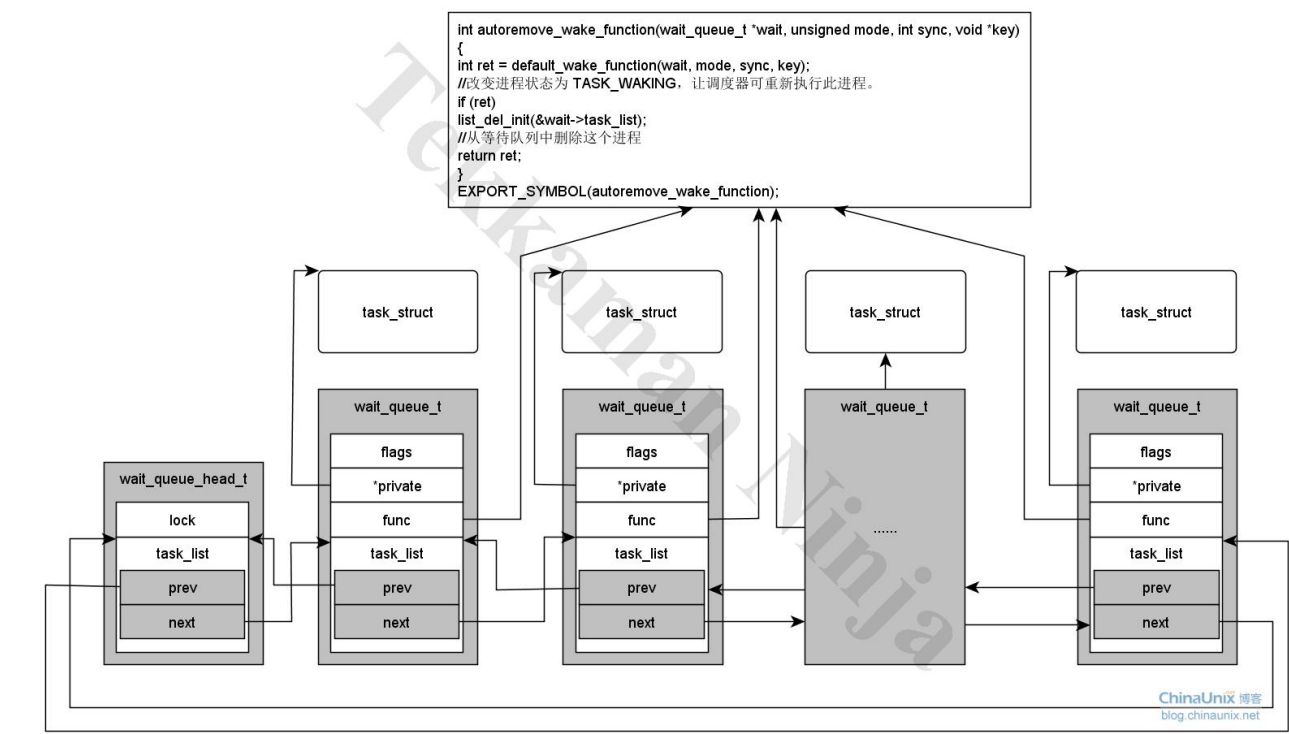
它包含一个自旋锁和一个链表。这是一个等待队列链表头，链表中的元素被声明做wait\_queue\_t。自旋锁用于包含链表操作的原子性。

wait\_queue\_t包含关于睡眠进程的信息和唤醒函数。

```
typedef struct __wait_queue wait_queue_t;
typedef int (*wait_queue_func_t)(wait_queue_t *wait, unsigned mode, int flags, void *key);
int default_wake_function(wait_queue_t *wait, unsigned mode, int flags, void *key);

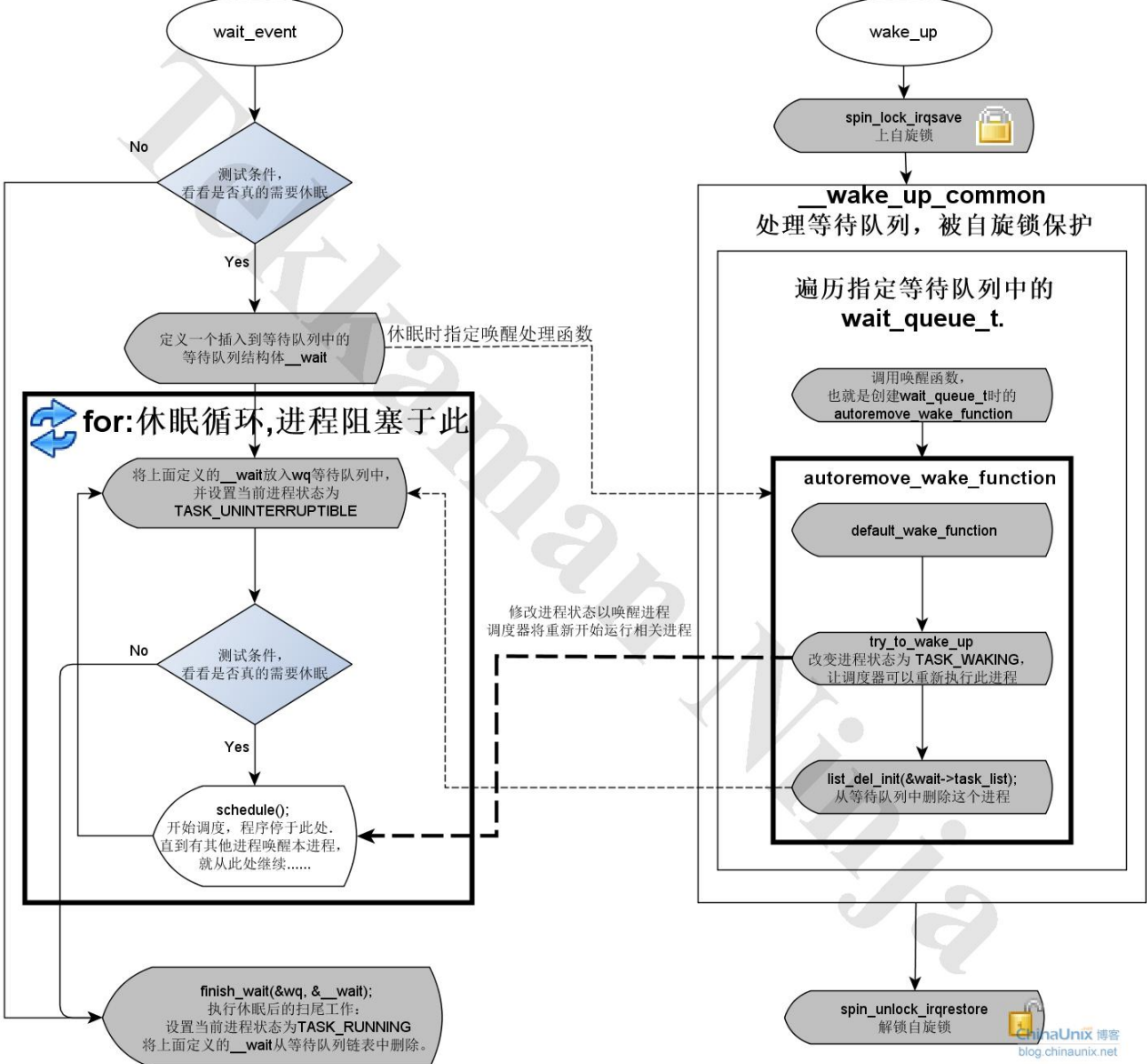
struct __wait_queue {
    unsigned int flags;
#define WQ_FLAG_EXCLUSIVE 0x01
    void *private;
    wait_queue_func_t func;
    /* 表示等待进程想要被独占地唤醒 */
    /* 指向等待进程的task_struct结构图 */
    /* 用于唤醒等待进程的处理例程，在其中实现了进程状态的改变 */
    struct list_head task_list;
    /* 双向链表结构体，用于将wait_queue_t链接到wait_queue_head_t */
    e_head_t *
};
```

他们在内存中的结构大致如下图所示：



等待队列头wait\_queue\_head\_t一般是定义在模块或内核代码中的全局变量，而其中链接的元素 wait\_queue\_t的定义被包含在了休眠宏中。

休眠和唤醒的过程如下图所示：



### 五、休眠和唤醒的代码简要分析

下面我们简单分析一下休眠与唤醒的内核原语。

#### 1、休眠：wait\_event

```
/**
 * wait_event - 休眠，直到 condition 为真
 * @wq: 所休眠的等待队列
 * @condition: 所等待事件的一个C表达式
 *
 * 进程被置为等待状态（TASK_UNINTERRUPTIBLE）直到
 * @condition 评估为真。@condition 在每次等待队列@wq 被唤醒时
 * 都被检查。
 *
 * wake_up() 必须在改变任何可能影响等待条件结果
 * 的变量之后被调用。
 */
#define wait_event(wq, condition) \
do { \
if (condition) \
break; \

先测试条件，看看是否真的需要休眠

__wait_event(wq, condition); \
} while (0)

#define __wait_event(wq, condition) \
do { \
DEFINE_WAIT(__wait); \

定义一个插入到等待队列中的等待队列结构体，注意.private = current,（即当前进程）
#define DEFINE_WAIT_FUNC(name, function) \
wait_queue_t name = { \
.private = current, \
.func = function, \
.task_list = LIST_HEAD_INIT((name).task_list), \
}
#define DEFINE_WAIT(name) DEFINE_WAIT_FUNC(name, autoremove_wake_function)

\
for (;;) { \
prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE); \

将上面定义的结构体__wait放入wq等待队列中，并设置当前进程状态为TASK_UNINTERRUPTIBLE

if (condition) \
break; \

测试条件状态，看看是否真的需要休眠调度

schedule(); \

开始调度，程序停于此处，直到有其他进程唤醒本进程，就从此处继续.....

} \
finish_wait(&wq, &__wait); \

由于测试条件状态为假，跳出以上循环后执行休眠后的扫尾工作：
```

设置当前进程状态为TASK\_RUNNING  
将上面定义的\_\_wait从等待队列链表中删除。

```
} while (0)
```

2、唤醒：wake\_up

```
#define wake_up(x) __wake_up(x, TASK_NORMAL, 1, NULL)
```

```
/**
 * __wake_up - 唤醒阻塞在等待队列上的线程.
 * @q: 等待队列
 * @mode: which threads
 * @nr_exclusive: how many wake-one or wake-many threads to wake up
 * @key: is directly passed to the wakeup function
 *
 * It may be assumed that this function implies a write memory barrier before
 * changing the task state if and only if any tasks are woken up.
 */
void __wake_up(wait_queue_head_t *q, unsigned int mode,
int nr_exclusive, void *key)
{
    unsigned long flags;
    spin_lock_irqsave(&q->lock, flags);
    __wake_up_common(q, mode, nr_exclusive, 0, key);
    spin_unlock_irqrestore(&q->lock, flags);
}
EXPORT_SYMBOL(__wake_up);
```

kernel/sched.c

```
/*
 * 核心唤醒函数.非独占唤醒(nr_exclusive == 0) 只是
 * 唤醒所有进程. If it's an exclusive wakeup (nr_exclusive == small +ve
 * number) then we wake all the non-exclusive tasks and one exclusive task.
 *
 * There are circumstances in which we can try to wake a task which has already
 * started to run but is not in state TASK_RUNNING. try_to_wake_up() returns
 * zero in this (rare) case, and we handle it by continuing to scan the queue.
 */
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
int nr_exclusive, int wake_flags, void *key)
{
    wait_queue_t *curr, *next;
    list_for_each_entry_safe(curr, next, &q->task_list, task_list) {

        遍历指定等待队列中的wait_queue_t.

        unsigned flags = curr->flags;
        if (curr->func(curr, mode, wake_flags, key) &&
            (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
            break;

        调用唤醒函数，也就是创建wait_queue_t时的 autoremove_wake_function

    }
}
```

```
int autoremove_wake_function(wait_queue_t *wait, unsigned mode, int sync, void *key)
{
    int ret = default_wake_function(wait, mode, sync, key);

    if (ret)
        list_del_init(&wait->task_list);
```

从等待队列中删除这个进程

```
return ret;
}
EXPORT_SYMBOL(autoremove_wake_function);

int default_wake_function(wait_queue_t *curr, unsigned mode, int wake_flags,
void *key)
{
    return try_to_wake_up(curr->private, mode, wake_flags);
```

主要是要改变进程状态为 TASK\_WAKING，让调度器可以重新执行此进程。

```
}
EXPORT_SYMBOL(default_wake_function);
```



上面分析的休眠函数是最简单的休眠唤醒函数，其他类似的函数，如后缀为\_timeout、\_interruptible、\_interruptible\_timeout的函数其实都是在唤醒后的条件判断上有些不同，多判断一些唤醒条件而已。这里就不再赘述了。

## 六、使用休眠的注意事项

(1) **永远不要在原子上下文中进入休眠**，即当驱动在持有一个自旋锁、seqlock或者 RCU 锁时不能睡眠；关闭中断也不能睡眠，终端例程中也不可休眠。

**持有一个信号量时休眠是合法的，如果代码在持有一个信号量时睡眠，任何其他的等待这个信号量的线程也会休眠。发生在持有信号量时的休眠必须短暂，而且决不能阻塞那个将最终唤醒你的进程。**

(2) 当进程被唤醒，它并不知道休眠了多长时间以及休眠时发生什么；也不知道是否另有进程也在休眠等待同一事件，且那个进程可能在它之前醒来并获取了所等待的资源。所以**不能对唤醒后的系统状态做任何假设，并必须重新检查等待条件来确保正确的响应。**

(3) 除非**确信其他进程会在其他地方唤醒休眠的进程**，否则也不能睡眠。使进程可被找到意味着：需要维护一个**等待队列**的数据结构。它是一个进程链表，其中包含了等待某个特定事件的所有进程的相关信息。

## 七、不可在中断例程中休眠的原因

如果在某个系统调用中把当前进程休眠，是有明确目标的，这个目标就是过来call这个系统调用的进程（注意这个进程正在running）。

但是中断和进程是异步的，在中断上下文中，当前进程大部分时候和中断代码可能一点关系都没有。要是在这里调用了休眠代码，把当前进程给休眠了，那就极有可能把无关的进程休眠了。再者，如果中断不断到来，会殃及许多无辜的进程。

在中断中休眠某个特定进程是可以实现的，通过内核的task\_struct链表可以找到的，不论是根据PID还是name。但是只要这个进程不是当前进程，休眠它也可能没有必要。可能这个进程本来就在休眠；或者正在执行队列中但是还没执行到，如果执行到他了可能又无须休眠了。

还有一个原因是中断也是所谓的原子上下文，有的中断例程中会禁止所有中断，有的中断例程还会使用自旋锁等机制，在其中使用休眠也是非常危险的。下面会介绍。

## 八、不可在持有自旋锁、seqlock、RCU 锁或关闭中断时休眠的原因

其实自旋锁、seqlock、RCU 锁或关闭中断期间的代码都称为原子上下文，比较有代表性的就是自旋锁spinlock。

对于UP系统，如果A进程在拥有spinlock时休眠，这个进程在拥有自旋锁后主动放弃了处理器。其他的进程就开始使用处理器，只要有一个进程B去获取同一个自旋锁，B必然无法获取，并做所谓的自旋等待。由于自旋锁禁止所有中断和抢占，B的自旋等待是不会被打断的，并且B也永远获得不了锁。因为B在CPU中运行，没有其他进程可以运行并唤醒A并释放锁。系统就此锁死，只能复位了。

对于SMP系统，如果A进程在拥有spinlock时休眠，这个进程在拥有自旋锁后主动放弃了处理器。如果所有处理器都为了获取这个锁而自旋等待，由于自旋锁禁止所有中断和抢占，，就不会有进程可能去唤醒A了，系统也就锁死了。

并不是所一旦系统获得自旋锁休眠就会死，而是有这个可能。但是注意了计算机的运行速度之快，只要有亿分之一的可能，也是很容易发生。

所有的原子上下文都有这样的共性：不可在其中休眠，否则系统极有可能锁死。

如果你对此还有怀疑，眼见为实。我编写了一个故意锁死系统的及其简单的驱动：



spin\_lock\_sleep.rar

只要对其设备节点做两次读写操作，系统必死。我在X86 的SMP系统，ARMv5、ARMv6、ARMv7中都做了如下的实验（**单核(UP)系统必须配置CONFIG\_DEBUG\_SPINLOCK，否则自旋锁是没有实际效果（起码不会有“自旋”），系统可以多次获取自旋锁，没有实验效果。之后博文中有详细描述**）。现象都和上面叙述的死法相同，看了源码就知道（关键在read\write方法）。以下是实验记录：

```
# insmod spin_lock_sleep.ko
spin_lock sleep module loaded!
# cat /proc/devices
Character devices:
1 mem
4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
7 vcs
10 misc
13 input
14 sound
21 sg
29 fb
81 video4linux
89 i2c
90 mtd
116 alsa
```

```
128 ptm
136 pts
252 spin_lock_sleep
253 tty0
254 rtc
Block devices:
1 ramdisk
259 blkext
7 loop
8 sd
11 sr
31 mtddbblock
65 sd
66 sd
67 sd
68 sd
69 sd
70 sd
71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
179 mmc
# mknod spin_lock_sleep c 252 0
# cat spin_lock_sleep
spin_lock_sleep_read:prepare to get spin_lock! PID:1227
spin_lock_sleep_read:have got the spin_lock! PID:1227
spin_lock_sleep_read:prepare to sleep! PID:1227
spin_lock_sleep_write:prepare to get spin_lock! PID:1229
BUG: spinlock cpu recursion on CPU#0, sh/1229
lock: dd511c3c, .magic: dead4ead, .owner: cat/1227, .owner_cpu: 0
Backtrace:
[<c005e9fc>] (dump_backtrace+0x0/0x118) from [<c04335e4>] (dump_stack+0x20/0x24)
r7:00000002 r6:dd511c3c r5:dd511c3c r4:dd7ef000
[<c04335c4>] (dump_stack+0x0/0x24) from [<c02178d0>] (spin_bug+0x94/0xa8)
[<c021783c>] (spin_bug+0x0/0xa8) from [<c0217a58>] (do_raw_spin_lock+0x6c/0x160)
r5:bf04c408 r4:dd75e000
[<c02179ec>] (do_raw_spin_lock+0x0/0x160) from [<c0435ec4>] (_raw_spin_lock+0x18/0x1c)
[<c0435eac>] (_raw_spin_lock+0x0/0x1c) from [<bf04c1a4>] (spin_lock_sleep_write+0xb4/0x19
0 [spin_lock_sleep])
[<bf04c0f0>] (spin_lock_sleep_write+0x0/0x190 [spin_lock_sleep]) from [<c011ee90>] (vfs_w
rite+0xb8/0xe0)
r6:dd75ff70 r5:400d7000 r4:dd43bf00
[<c011edd8>] (vfs_write+0x0/0xe0) from [<c011ef8c>] (sys_write+0x4c/0x78)
r7:00000002 r6:dd43bf00 r5:00000000 r4:00000000
[<c011ef40>] (sys_write+0x0/0x78) from [<c005a380>] (ret_fast_syscall+0x0/0x48)
r8:c005a5a8 r7:00000004 r6:403295e8 r5:400d7000 r4:00000002
```

此时在另一个终端（ssh、telnet等）中执行命令：  
**echo 'l' > spin\_lock\_sleep**

```
BUG: spinlock lockup on CPU#0, sh/1229, dd511c3c
Backtrace:
[<c005e9fc>] (dump_backtrace+0x0/0x118) from [<c04335e4>] (dump_stack+0x20/0x24)
r7:dd75e000 r6:dd511c3c r5:00000000 r4:00000000
[<c04335c4>] (dump_stack+0x0/0x24) from [<c0217b0c>] (do_raw_spin_lock+0x120/0x160)
[<c02179ec>] (do_raw_spin_lock+0x0/0x160) from [<c0435ec4>] (_raw_spin_lock+0x18/0x1c)
[<c0435eac>] (_raw_spin_lock+0x0/0x1c) from [<bf04c1a4>] (spin_lock_sleep_write+0xb4/0x19
0 [spin_lock_sleep])
[<bf04c0f0>] (spin_lock_sleep_write+0x0/0x190 [spin_lock_sleep]) from [<c011ee90>] (vfs_w
rite+0xb8/0xe0)
r6:dd75ff70 r5:400d7000 r4:dd43bf00
[<c011edd8>] (vfs_write+0x0/0xe0) from [<c011ef8c>] (sys_write+0x4c/0x78)
r7:00000002 r6:dd43bf00 r5:00000000 r4:00000000
[<c011ef40>] (sys_write+0x0/0x78) from [<c005a380>] (ret_fast_syscall+0x0/0x48)
r8:c005a5a8 r7:00000004 r6:403295e8 r5:400d7000 r4:00000002
```

而你这样原子环境中休眠调度，内核一旦检测到（主要是检测到关闭了抢占），你可能会看到如下信息，警告你：

```
# cat spin_lock_sleep
spin_lock_sleep_read:prepare to get spin_lock! PID:540
spin_lock_sleep_read:have got the spin_lock! PID:540
spin_lock_sleep_read:prepare to sleep! PID:540
BUG: scheduling while atomic: cat/540/0x00000002
Modules linked in: spin_lock_sleep
[<c0070364>] (unwind_backtrace+0x0/0xe4) from [<c03304a4>] (schedule+0x74/0x36c)
[<c03304a4>] (schedule+0x74/0x36c) from [<bf0062bc>] (spin_lock_sleep_read+0xe8/0x1bc [sp
in_lock_sleep])
[<bf0062bc>] (spin_lock_sleep_read+0xe8/0x1bc [spin_lock_sleep]) from [<c00dd370>] (vfs_r
ead+0xac/0x154)
[<c00dd370>] (vfs_read+0xac/0x154) from [<c00dd454>] (sys_read+0x3c/0x68)
[<c00dd454>] (sys_read+0x3c/0x68) from [<c006ae60>] (ret_fast_syscall+0x0/0x2c)
```

上一篇: [Linux下访问u-boot环境变量简介](#)  
下一篇: [没有了](#)

相关热门文章

- |                                      |   |   |
|--------------------------------------|---|---|
| <a href="#">ZigBee城市道路井盖安全监测系...</a> | <a href="#">A sample .exrc file for vi e...</a> | <a href="#">sed -e "/grep/d" 是什么意思...</a> |
| <a href="#">大学生竞价案例分享：每个月广...</a>    | <a href="#">IBM System p5 服务器 HACMP ...</a>     | <a href="#">谁能够帮我解决LINUX 2.6 10...</a>    |
| <a href="#">Linux入门基础分享[二]</a>       | <a href="#">busybox的httpd使用CGI脚本(Bu...</a>      | <a href="#">现在的博客积分不会更新了吗? ...</a>        |
| <a href="#">Linux入门基础分享[一]</a>       | <a href="#">Solaris PowerTOP 1.0 发布</a>         | <a href="#">shell怎么读取网页内容...</a>          |
| <a href="#">[C++] 关于进程中的内存空间分...</a> | <a href="#">For STKMonitor</a>                  | <a href="#">ssh等待连接的超时问题...</a>           |

给主人留下些什么吧！~~

评论热议

请登录后再评论。  
[登录](#) [注册](#)