蜗窝科技

慢下来，享受技术。

进程切换分析（3）：同步处理

作者：linuxer 发布于：2017-12-11 17:59 分类：进程管理

一、前言

本文主要描述了主调度器（schedule函数）中的同步处理。

二、进程调度简介

进程切换有两种，一种是当进程由于需要等待某种资源而无法继续执行下去，这时候只能是主动将自己挂起（调用schedule函数），引发一次任务调度过程。另外一种是进程被抢占。所谓抢占（preempt）就是在当前进程欢快执行的时候，终止其对CPU资源的占用，切换到另外一个更高优先级的进程执行。进程被抢占往往是由于各种调度事件的发生：

1、 时间片用完

2、 在中断上下文中唤醒其他优先级更高的进程

3、 在其他进程上下文中唤醒其他优先级更高的进程。

4、 在其他进程上下文中修改了其他进程的调度参数

5、 ……

在当前进程被抢占的场景下，调度并不是立刻发生，而是延迟执行，具体的方法是设定当前进程的need_resched等于1，然后静静的等待最近一个调度点的来临，当调度点到来的时候，内核会调用schedule函数，抢占当前task的执行。

此外，我们还需要了解基本的抢占控制的知识。在一个进程的thread info中有一个preempt_count的成员用来控制抢占，当该成员等于0的时候表示允许抢占，在本文中，我们分别用preempt counter、hardirq counter和softirq counter分别表示其中的bit field。更详细的描述可以参考相关文档的描述

三、schedule函数使用了哪些同步机制

schedule函数的代码框架如下：

```
asmlinkage __visible void __sched schedule(void)
{
    do {
        preempt_disable(); - - - - - - - - - - - - - - - - - - - a

                raw_spin_lock_irq(&rq->lock); - - - - - - - - b

                选择next task

                切到next task执行
```

```
            raw_spin_unlock_irq(&rq->lock);  - - - - - - - - c
        sched_preempt_enable_no_resched();  - - - - - - - - - d
    } while (need_resched());  - - - - - - - - - - - - - - - - e
}
```

我们以X进程切换到Y进程为例，描述schedule函数中同步机制的使用情况。在X进程上下文中，a点首先关闭了抢占，X task的preempt counter会加1。然后在b点会持有该CPU runqueue的spinlock，当然在这个过程中会disable CPU中断处理，同时将X task的preempt counter再次加1，这时候X task的preempt counter应该等于2。

打开X task的抢占的时候是在重新调度X在某个CPU上执行的时候，这时候，在上面代码中的c和d点来递减preempt counter，当进入e点的时候，preempt counter已经等于0。

由于在切换过长设计runqueue队列的操作，因此需要spin lock来保护。不过在进程切换过程中，runqueue spin lock是不同进程来协同处理的。我们仍然以X进程切换到Y进程为例。在X进程中，在b点持锁并disable了本地中断，而spin lock的释放是在Y进程中完成的（c点），在释放spin lock的同时，也会打开cpu中断。

四、可不可以禁止抢占的时候调用schedule函数

在进程上下文中，下面的调用序列是否可以呢?

```
preempt_disable

……schedule……

preempt_enable
```

无论什么场景，disable preempt然后调用schedule都是很奇怪的一件事情：本来你已经禁止抢占了，但是又显示的调用schedule函数，你这不是精神分裂吗？schedule函数怎么处理这个精神分裂的task呢？在调用schedule函数之前，它毫无疑问是期待preempt count等于0的，只有当前task的preempt count等于0才说明抢占的合理性。不过在整个进程切换的过程中，首先会在a点禁止抢占，这样可以确保CPU和当前task之间的关系不变（cpu不变、current task不变，runqueue不变）。这样，在a和b之间的对caller的调用检查就比较好开展了，具体如下：

```
static inline void schedule_debug(struct task_struct *prev)
{

    if (unlikely(in_atomic_preempt_off())) {
        __schedule_bug(prev);
        preempt_count_set(PREEMPT_DISABLED);
    }
}
```

in_atomic_preempt_off这个宏就是对当前preempt count进行测试，这时候正确的preempt counter应该是等于1，其他的bit field，例如softirq counter、hardirq count等都是0。具体关于preempt count的位域描述可以参考本站软中断的文档。如果没有设定正确的preempt_count就调用schedule函数，那么说明在atomic上下文中错误的进行了调度，__schedule_bug会打印出相关信息，方便调试。

虽然在错误的场景中调用了schedule函数，但是内核还是要艰难前行啊，因此这里会修改preempt count的值为PREEMPT_DISABLED，而这才是进入schedule函数正确的姿势。

五、可不可以关闭中断调用schedule函数?

在进程上下文中，下面的调用序列是否可以呢?

项目专区(0)
X Project(28)

随机文章

Linux电源管理(12)_Hibernate功能
Unix的历史
process credentials相关的用户空间文件
致驱动工程师的一封信
Linux vm运行参数之（一）：overcommit相关的参数

文章存档

2022年2月(2)
2022年1月(1)
2021年12月(1)
2021年11月(5)
2021年7月(1)
2021年6月(1)
2021年5月(3)
2020年3月(3)
2020年2月(2)
2020年1月(3)
2019年12月(3)
2019年5月(4)
2019年3月(1)
2019年1月(3)
2018年12月(2)
2018年11月(1)
2018年10月(2)
2018年8月(1)
2018年6月(1)
2018年5月(1)
2018年4月(7)
2018年2月(4)
2018年1月(5)
2017年12月(2)
2017年11月(2)
2017年10月(1)
2017年9月(5)
2017年8月(4)
2017年7月(4)
2017年6月(3)
2017年5月(3)
2017年4月(1)
2017年3月(8)
2017年2月(6)
2017年1月(5)
2016年12月(6)
2016年11月(11)
2016年10月(9)
2016年9月(6)
2016年8月(9)
2016年7月(5)
2016年6月(8)
2016年5月(8)
2016年4月(7)
2016年3月(5)
2016年2月(5)
2016年1月(6)
2015年12月(6)
2015年11月(9)
2015年10月(9)
2015年9月(4)
2015年8月(3)
2015年7月(7)
2015年6月(3)
2015年5月(6)

> local_irq_disable
>
> ……schedule……
>
> local_irq_enable

当然这里也许不是直接调用schedule函数，很多内核接口API会隐含调用schedule函数，因此也许你会有意无意的写出上面形态的代码。

首先需要明确一点：从X进程切换到Y进程的时候，如果在X进程中关闭中断，然后切换到Y进程，如果中断不恢复的话，那么Y进程会一直执行，直到Y自己良心发现，让出CPU。这当然是不被允许的。因此，在调用schedule进行进程切换的时候，无论调用者是否关闭中断，在b点都会关闭中断（注意，这时候并没有记录之前的中断状态）。而在切入到Y进程之后，在c点都会显式的打开CPU中断。因此，上面的代码虽然不推荐，但是也不会对调度产生太大的影响。

六、禁止中断是否可以禁止抢占?

禁止了中断的确等于了禁止抢占，但是并不意味着它们两个完全等同，因为在preempt disable - - - preempt enable这个的调用过程中，在打开抢占的时候有一个抢占点，内核控制路径会在这里检查抢占，如果满足抢占条件，那么会立刻调度schedule函数进行进程切换，但是local irq disable - - - local irq enable的调用中，并没有显示的抢占检查点，当然，中断有点特殊，因为一旦打开中断，那么pending的中断会进来，并且在返回中断点的时候会检查抢占，但是也许下面的这个场景就无能为力了。进程上下文中调用如下序列：

（1）local irq disable

（2）wake up high level priority task

（3）local irq enable

当唤醒的高优先级进程被调度到本CPU执行的时候，按理说这个高优先级进程应该立刻抢占当前进程，但是这个场景无法做到。在调用try_to_wake_up的时候会设定need resched flag并检查抢占，但是由于中断disable，因此不会立刻调用schedule，但是在step （3）的时候，由于没有检查抢占，这时候本应立刻抢占的高优先级进程会发生严重的调度延迟.....直到下一个抢占点到来。

原创文章，转发请注明出处。蜗窝科技

标签: schedule

« USB-C(USB Type-C)规范的简单介绍和分析 | 逆向映射的演进»

评论:

大明白
2019-07-18 17:46

由于在切换过长设计runqueue队列的操作，因此需要spin lock来保护。不过在进程切换过程中，runqueue spin lock是不同进程来协同处理的。我们仍然以X进程切换到Y进程为例。在X进程中，在b点持锁并disable了本地中断，而spin lock的释放是在Y进程中完成的（c点），在释放spin lock的同时，也会打开cpu中断。

==》spinlock的释放也是在同一进程中处理的，不会通过其他进程处理。X进程在b点持锁后，如果X会被切出当前cpu，那么在切出之前会unlock；如果X不会被切出当前cpu，那么X进程跑到c点自己去unlock。总之并不会和其他进程协同处理rq->lock。

回复

nzg
2018-10-10 17:35

hi 您好，这里有2个疑问：
当唤醒的高优先级进程被调度到本CPU执行的时候，按理说这个高优先级进程应该立刻抢占当前进程，但是这个场景无法做到。
Q：本CPU 的中断已经关闭，无法响应IPI 中断，如何能把这个唤醒的高优先级进程调度到本CPU 呢?

在调用try_to_wake_up的时候会设定need resched flag并检查抢占，但是由于中断disable，因此不会立刻调用schedule，但是在step （3）的时候，由于没有检查抢占，这时候本应立刻抢占的高优先级进程会发生严重的调度延迟.....直到下一个抢占点到来。
Q：由于中断disable, 为何就不会立刻调用schedule? 是因为不响应IPI吗？如果调用try_to_wake_up之后，这里调用了preempt_disable->premmpt_enable, 这样就会触发schedule吧?

谢谢

回复

man8266
2018-12-16 01:06

@nzg：这里跟IPI有啥关系?

回复

linuxerer
2017-12-18 10:33

首先需要明确一点：从X进程切换到Y进程的时候，如果在X进程中关闭中断，然后切换到Y进程，如果中断不恢复的话，那么Y进程会一直执行，直到Y自己良心发现，让出CPU。这当然是不被允许的。因此，在调用schedule进行进程切换的时候，无论调用者是否关闭中断，在b点都会关闭中断（注意，这时候并没有记录之前的中断状态）。而在切入到Y进程之后，在c点都会显式的打开CPU中断。因此，上面的代码虽然不推荐，但是也不会对调度产生太大的影响。

c点显示打开中断这点,

```
if (likely(prev != next)) {
        rq->nr_switches++;
        rq->curr = next;
        ++*switch_count;

        trace_sched_switch(preempt, prev, next);
        rq = context_switch(rq, prev, next); /* unlocks the rq */
        cpu = cpu_of(rq);
    } else {
        lockdep_unpin_lock(&rq->lock);
        raw_spin_unlock_irq(&rq->lock);
    }
```

c点不是每一次都会跑到吧?

回复

linuxerer
2017-12-18 15:15

@linuxerer：找到了, 是在
context_switch
finish_task_switch
finish_lock_switch
raw_spin_unlock_irq

回复

jack
2017-12-12 10:52

针对case四：
个人认为从2.5.4开始支持抢占，设计者在这里的抢占=”内核抢占“，因为之前的内核也没有“用户态抢占”一说。在用户态发生的中断返回时，不再检查是否支持“抢占”，只有发生在内核态的中断返回才检查preempt_count。而处于内核中的进程上下文显示的schedule()时，我认为是自己让出，并不算抢占吧。进程X不允许被抢占，就是不允许在内核中被其他因素调度出去，但是并不等于不允许自己主动放弃CPU吧。就好比一个进程的preempt_count一直为1，但并不影响它的运行。只是这个进程恢复到了”2.5.4内核“前的动作一样。可以理解为在早期的2.4版本上，为thread_info增加了preempt_count成员，且这个成员初始化为1，但本身内核又完全没加入抢占的feature。
再者，case四中，如果schedule()前preepmt_count非0，即使不去”重新修改设置"preempt_count：preempt_count_set(PREEMPT_DISABLED); 也没有影响吧。
某个进程就是不想2.6内核的抢占，只想在2.4上呢?

回复

linuxer
2017-12-16 09:35

@jack：其实你说得也也一些道理，以前的非抢占式内核时代，一旦进入内核态，也就是禁止了抢占，有一点类似整个内核态设定preempt_count总是等于1.

在ARM平台上，preempt_count是per task的，所以：
preempt_disable
……schedule……

preempt_enable
的调用序列也无可厚非。我们假设schedule函数发生了A--》B任务的切换，那么上面的调用序列在执行B任务的时候，A task的preempt_count保持非0，似乎也没有什么影响，反正A任务下次调度回来会继续。然而，并非所有平台上，preempt_count都是per task的，有些平台，例如X86，preempt_count是per CPU的，你调用preempt_disable影响的是当前CPU的抢占状态，因此，上面的调用序列就存在问题了。因为B进程调度回来之后，CPU上的抢占被A进程disable了。
也正是因为这个原因，在scheduler_debug中，当显示的修正preempt_count，以防止在per cpu preempt count的情况下，disable preempt会从一个task溢出到另外的一个task。

回复

发表评论：

昵称

邮件地址 (选填)

个人主页 (选填)

发表评论