Linux 内核抢占实现机制分析

Sailor_forever　sailing_9806@163.com 转载请注明

http://blog.csdn.net/sailor_8318/archive/2008/09/03/2870184.aspx

【摘要】本文详解了 Linux 内核抢占实现机制。首先介绍了内核抢占和用户抢占的概念和区别，接着分析了不可抢占内核的特点及实时系统中实现内核抢占的必要性。然后分析了禁止内核抢占的情况和内核抢占的时机，最后介绍了实现抢占内核所做的改动以及何时需要重新调度。

【关键字】内核抢占，用户抢占，中断， 实时性，自旋锁，抢占时机，调度时机，schedule，preempt count

1　内核抢占概述

2.6 新的可抢占式内核是指内核抢占，即当进程位于内核空间时，有一个更高优先级的任务出现时，如果当前内核允许抢占，则可以将当前任务挂起，执行优先级更高的进程。

在 2.5.4 版本之前，Linux 内核是不可抢占的，高优先级的进程不能中止正在内核中运行的低优先级的进程而抢占 CPU 运行。进程一旦处于核心态(例如用户进程执行系统调用)，则除非进程自愿放弃 CPU，否则该进程将一直运行下去，直至完成或退出内核。与此相反，一个可抢占的 Linux 内核可以让 Linux 内核如同用户空间一样允许被抢占。当一个高优先级的进程到达时，不管当前进程处于用户态还是核心态，如果当前允许抢占，可抢占内核的 Linux 都会调度高优先级的进程运行。

2　用户抢占

内核即将返回用户空间的时候，如果 need resched 标志被设置，会导致 schedule()被调用，此时就会发生用户抢占。在内核返回用户空间的时候，它知道自己是安全的。所以，内核无论是在从中断处理程序还是在系统调用后返回，都会检查 need resched 标志。如果它被设置了，那么，内核会选择一个其他(更合适的)进程投入运行。

简而言之，用户抢占在以下情况时产生：

从系统调返回用户空间。

从中断处理程序返回用户空间。

3　不可抢占内核的特点

在不支持内核抢占的内核中，内核代码可以一直执行，到它完成为止。也就是说，调度程序没有办法在一个内核级的任务正在执行的时候重新调度—内核中的各任务是协作方式调度的，不具备抢占性。内核代码一直要执行到完成(返回用户空间)或明显的阻塞为止。

在单 CPU 情况下，这样的设定大大简化了内核的同步和保护机制。可以分两步对此加以分析：

首先，不考虑进程在内核中自愿放弃 CPU 的情况(也即在内核中不发生进程的切换)。一个进程一旦进入内核就将一直运行下去，直到完成或退出内核。在其没有完成或退出内核之前，不会有另外一个进程进入内核，即进程在内核中的执行是串行的，不可能有多个进程同时在内核中运行，这样内核代码设计时就不用考虑多个进程同时执行所带来的并发问题。Linux 的内核开发人员就不用考虑复杂的进程并发执行互斥访问临界资源的问题。当进程在访问、修改内核的数据结构时就不需要加锁来防止多个进程同时进入临界区。这时只需再考虑一下中断的情况，若有中断处理例程也有可能访问进程正在访问的数据结构，那么进程只要在进

入临界区前先进行关中断操作，退出临界区时进行开中断操作就可以了。
再考虑一下进程自愿放弃 CPU 的情况。因为对 CPU 的放弃是自愿的、主动的，也就意味着进程在内核中的切换是预先知道的，不会出现在不知道的情况下发生进程的切换。这样就只需在发生进程切换的地方考虑一下多个进程同时执行所可能带来的并发问题，而不必在整个内核范围内都要考虑进程并发执行问题。

4 为什么需要内核抢占？

实现内核的可抢占对 Linux 具有重要意义。首先，这是将 Linux 应用于实时系统所必需的。实时系统对响应时间有严格的限定，当一个实时进程被实时设备的硬件中断唤醒后，它应在限定的时间内被调度执行。而 Linux 不能满足这一要求，因为 Linux 的内核是不可抢占的，不能确定系统在内核中的停留时间。事实上当内核执行长的系统调用时，实时进程要等到内核中运行的进程退出内核才能被调度，由此产生的响应延迟，在如今的硬件条件下，会长达 100ms 级。

这对于那些要求高实时响应的系统是不能接受的。而可抢占的内核不仅对 Linux 的实时应用至关重要，而且能解决 Linux 对多媒体(video, audio)等要求低延迟的应用支持不够好的缺陷。由于可抢占内核的重要性，在 Linux2.5.4 版本发布时，可抢占被并入内核，同 SMP 一样作为内核的一项标准可选配置。

5 什么情况不允许内核抢占

有几种情况 Linux 内核不应该被抢占，除此之外 Linux 内核在任意一点都可被抢占。这几种情况是：

- 内核正进行中断处理。在 Linux 内核中进程不能抢占中断(中断只能被其他中断中止、抢占，进程不能中止、抢占中断)，在中断例程中不允许进行进程调度。进程调度函数 schedule()会对此作出判断，如果是在中断中调用，会打印出错信息。
- 内核正在进行中断上下文的 Bottom Half(中断的底半部)处理。硬件中断返回前会执行软中断，此时仍然处于中断上下文中。
- 内核的代码段正持有 spinlock 自旋锁、writelock/readlock 读写锁等锁，处干这些锁的保护状态中。内核中的这些锁是为了在 SMP 系统中短时间内保证不同 CPU 上运行的进程并发执行的正确性。当持有这些锁时，内核不应该被抢占，否则由于抢占将导致其他 CPU 长期不能获得锁而死等。
- 内核正在执行调度程序 Scheduler。抢占的原因就是为了进行新的调度，没有理由将调度程序抢占掉再运行调度程序。
- 内核正在对每个 CPU“私有”的数据结构操作(Per-CPU date structures)。在 SMP 中，对于 per-CPU 数据结构未用 spinlocks 保护，因为这些数据结构隐含地被保护了(不同的 CPU 有不一样的 per-CPU 数据，其他 CPU 上运行的进程不会用到另一个 CPU 的 per-CPU 数据)。但是如果允许抢占，但一个进程被抢占后重新调度，有可能调度到其他的 CPU 上去，这时定义的 Per-CPU 变量就会有问题，这时应禁抢占。

为保证 Linux 内核在以上情况下不会被抢占，抢占式内核使用了一个变量 preempt_ count，称为内核抢占锁。这一变量被设置在进程的 PCB 结构 task_struct 中。每当内核要进入以上几种状态时，变量 preempt_ count 就加 1，指示内核不允许抢占。每当内核从以上几种状态退出时，变量 preempt_ count 就减 1，同时进行可抢占的判断与调度。

从中断返回内核空间的时候，内核会检查 need_resched 和 preempt_count 的值。如果 need_ resched 被设置，并且 preempt count 为 0 的话，这说明可能有一个更为重要的任务需要执行并且可以安全地抢占，此时，调度程序就会被调用。如果 preempt-count 不为 0，则说明内核现在处干不可抢占状态，不能进行重新调度。这时，就会像通常那样直接从中断返回当前执行进程。如果当前进程持有的所有的锁都被释放了，那么 preempt_ count 就会重新为 0。此时，释放锁的代码会检查 need_ resched 是否被设置。如果是的话，就会调用调度程序。

6 内核抢占时机

在 2.6 版的内核中，内核引入了抢占能力；现在，只要重新调度是安全的，那么内核就可以在任何时间抢占正在执行的任务。

那么，什么时候重新调度才是安全的呢？只要 premptcount 为 0，内核就可以进行抢占。通常锁和中断是非抢占区域的标志。由于内核是支持 SMP 的，所以，如果没有持有锁，那么正在执行的代码就是可重新导人的，也就是可以抢占的。

如果内核中的进程被阻塞了，或它显式地调用了 schedule()，内核抢占也会显式地发生。这种形式的内核抢占从来都是受支持的(实际上是主动让出 CPU)，因为根本无需额外的逻辑来保证内核可以安全地被抢占。如果代码显式的调用了 schedule()，那么它应该清楚自己是可以安全地被抢占的。

内核抢占可能发生在：

当从中断处理程序正在执行，且返回内核空间之前。

当内核代码再一次具有可抢占性的时候，如解锁及使能软中断等。

如果内核中的任务显式的调用 schedule()

如果内核中的任务阻塞(这同样也会导致调用 schedule())

7 如何支持抢占内核

抢占式 Linux 内核的修改主要有两点：一是对中断的入口代码和返回代码进行修改。在中断的入口内核抢占锁 preempt_count 加 1，以禁止内核抢占；在中断的返回处，内核抢占锁 preempt_count 减 1，使内核有可能被抢占。

我们说可抢占 Linux 内核在内核的任一点可被抢占，主要就是因为在任意一点中断都有可能发生，每当中断发生，Linux 可抢占内核在处理完中断返回时都会进行内核的可抢占判断。若内核当前所处状态允许被抢占，内核都会重新进行调度选取高优先级的进程运行。这一点是与非可抢占的内核不一样的。在非可抢占的 Linux 内核中，从硬件中断返回时，只有当前被中断进程是用户态进程时才会重新调度，若当前被中断进程是核心态进程，则不进行调度，而是恢复被中断的进程继续运行。

另一基本修改是重新定义了自旋锁、读、写锁，在锁操作时增加了对 preempt count 变量的操作。在对这些锁进行加锁操作时 preemptcount 变量加 1，以禁止内核抢占；在释放锁时 preemptcount 变量减 1，并在内核的抢占条件满足且需要重新调度时进行抢占调度。下面以 spin_lock(), spin_unlock()操作为例说明：

//

/linux+v2.6.19/kernel/spinlock.c

320 void __lockfunc _spin_unlock(spinlock_t *lock)

```
321     {
322             spin_release(&lock->dep_map, 1, _RET_IP_);
323             _raw_spin_unlock(lock);
324             preempt_enable();
325}
326     EXPORT_SYMBOL(_spin_unlock);


178     void __lockfunc _spin_lock(spinlock_t *lock)
179     {
180             preempt_disable();
181             spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
182             _raw_spin_lock(lock);
183}
184
185      EXPORT_SYMBOL(_spin_lock);
```

//

```
 29     #define preempt_disable() \
 30     do { \
 31             inc_preempt_count(); \
 32             barrier(); \
 33             } while (0)
```

```
34
35      #define preempt_enable_no_resched() \
36      do { \
37          barrier(); \
38          dec_preempt_count(); \
39          } while (0)
40
41      #define preempt_check_resched() \
42      do { \
43          if (unlikely(test_thread_flag(TIF_NEED_RESCHED))) \
44                      preempt_schedule(); \
45      } while (0)
46
47      #define preempt_enable() \
48      do { \
49          preempt_enable_no_resched(); \
50          barrier(); \
51          preempt_check_resched(); \
52      } while (0)
53
```

另外一种可抢占内核实现方案是在内核代码段中插入抢占点(preemption point)的方案。在这一方案中，首先要找出内核中产生长延迟的代码段，然后在这一内核代码段的适当位置插入

抢占点，使得系统不必等到这段代码执行完就可重新调度。这样对于需要快速响应的事件，系统就可以尽快地将服务进程调度到 CPU 运行。抢占点实际上是对进程调度函数的调用，代码如下:

```
if (current->need_ resched) schedule();
```

通常这样的代码段是一个循环体，插入抢占点的方案就是在这一循环体中不断检测 need_ resched 的值，在必要的时候调用 schedule()令当前进程强行放弃 CPU

8 何时需要重新调度

内核必须知道在什么时候调用 schedule()。如果仅靠用户程序代码显式地调用 schedule()，它们可能就会永远地执行下去。相反，内核提供了一个 need_resched 标志来表明是否需要重新执行一次调度。当某个进程耗尽它的时间片时， scheduler tick()就会设置这个标志；当一个优先级高的进程进入可执行状态的时候，try_to_wake_up 也会设置这个标志。

set_ tsk_need_resched：设置指定进程中的 need_ resched 标志

clear tsk need_resched：清除指定进程中的 need_ resched 标志

need_resched()：检查 need_ resched 标志的值;如果被设置就返回真，否则返回假

信号量、等到队列、completion 等机制唤醒时都是基于 waitqueue 的，而 waitqueue 的唤醒函数为 default_wake_function，其调用 try_to_wake_up 将进程更改为可运行状态并置待调度标志。

在返回用户空间以及从中断返回的时候，内核也会检查 need_resched 标志。如果已被设置，内核会在继续执行之前调用调度程序。

每个进程都包含一个 need_resched 标志，这是因为访问进程描述符内的数值要比访问一个全局变量快(因为 current 宏速度很快并且描述符通常都在高速缓存中)。在 2.2 以前的内核版本中，该标志曾经是一个全局变量。2.2 到 2.4 版内核中它在 task_struct 中。而在 2.6 版中，它被移到 thread_info 结构体里，用一个特别的标志变量中的一位来表示。可见，内核开发者总是在不断改进。

/linux+v2.6.19/include/linux/sched.h

```
1503    static inline void set_tsk_need_resched(struct task_struct *tsk)
1504    {
1505            set_tsk_thread_flag(tsk,TIF_NEED_RESCHED);
```

```
1506        }
1507
1508        static inline void clear_tsk_need_resched(struct task_struct *tsk)
1509        {
1510                clear_tsk_thread_flag(tsk,TIF_NEED_RESCHED);
1511         }
1512
1513         static inline int signal_pending(struct task_struct *p)
1514        {
1515                return unlikely(test_tsk_thread_flag(p,TIF_SIGPENDING));
1516         }
1517
1518         static inline int need_resched(void)
1519         {
1520                return unlikely(test_thread_flag(TIF_NEED_RESCHED));
1521         }
```

//

/linux+v2.6.19/kernel/sched.c

```
 991/*
 992 * resched_task - mark a task 'to be rescheduled now'.
 993 *
```

```
 994 * On UP this means the setting of the need_resched flag, on SMP it
 995 * might also involve a cross-CPU call to trigger the scheduler on
 996 * the target CPU.
 997 */
 998     #ifdef CONFIG_SMP
 999
1000     #ifndef tsk_is_polling
1001     #define tsk_is_polling(t) test_tsk_thread_flag(t, TIF_POLLING_NRFLAG)
1002     #endif
1003
1004     static void resched_task(struct task_struct *p)
1005     {
1006             int cpu;
1007
1008             assert_spin_locked(&task_rq(p)->lock);
1009
1010             if (unlikely(test_tsk_thread_flag(p, TIF_NEED_RESCHED)))
1011                     return;
1012
1013             set_tsk_thread_flag(p, TIF_NEED_RESCHED);
1014
1015             cpu = task_cpu(p);
```

```
1016            if (cpu == smp_processor_id())
1017                    return;
1018
1019            /* NEED_RESCHED must be visible before we test polling */
1020            smp_mb();
1021            if (!tsk_is_polling(p))
1022                    smp_send_reschedule(cpu);
1023    }
1024    #else
1025     static inline void resched_task(struct task_struct *p)
1026     {
1027            assert_spin_locked(&task_rq(p)->lock);
1028            set_tsk_need_resched(p);
1029     }
1030    #endif
```

//

//

```
1366/***
1367 * try_to_wake_up - wake up a thread
1368 * @p: the to-be-woken-up thread
1369 * @state: the mask of task states that can be woken
```

```
1370 * @sync: do a synchronous wakeup?
1371 *
1372 * Put it on the run-queue if it's not already there. The "current"
1373 * thread is always on the run-queue (except when the actual
1374 * re-schedule is in progress), and as such you're allowed to do
1375 * the simpler "current->state = TASK_RUNNING" to mark yourself
1376 * runnable without the overhead of this.
1377 *
1378 * returns failure only if the task is already active.
1379 */
1380     static int try_to_wake_up(struct task_struct *p, unsigned int state, int sync)
```

```
////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////
1538      int fastcall wake_up_process(struct task_struct *p)
1539      {
1540            return try_to_wake_up(p, TASK_STOPPED | TASK_TRACED |
1541                                    TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE, 0);
1542      }
1543      EXPORT_SYMBOL(wake_up_process);

1545      int fastcall wake_up_state(struct task_struct *p, unsigned int state)
```

```
1546    {
1547            return try_to_wake_up(p, state, 0);
1548    }
```

```
1616/*
1617 * wake_up_new_task - wake up a newly created task for the first time.
1618 *
1619 * This function will do some initial scheduler statistics housekeeping
1620 * that must be done for every newly created context, then puts the task
1621 * on the runqueue and wakes it.
1622 */
1623    void fastcall wake_up_new_task(struct task_struct *p, unsigned long clone_flags)
```

```
3571/*
3572 * The core wakeup function.    Non-exclusive wakeups (nr_exclusive == 0) just
3573 * wake everything up.    If it's an exclusive wakeup (nr_exclusive == small +ve
3574 * number) then we wake all the non-exclusive tasks and one exclusive task.
3575 *
3576 * There are circumstances in which we can try to wake a task which has already
3577 * started to run but is not in state TASK_RUNNING.    try_to_wake_up() returns
```

```
3578 * zero in this (rare) case, and we handle it by continuing to scan the queue.
3579 */
3580     static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
3581                                  int nr_exclusive, int sync, void *key)
```

//

//

```
3595/**
3596 * __wake_up - wake up threads blocked on a waitqueue.
3597 * @q: the waitqueue
3598 * @mode: which threads
3599 * @nr_exclusive: how many wake-one or wake-many threads to wake up
3600 * @key: is directly passed to the wakeup function
3601 */
3602     void fastcall __wake_up(wait_queue_head_t *q, unsigned int mode,
3603                             int nr_exclusive, void *key)
3604     {
3605             unsigned long flags;
3606
3607             spin_lock_irqsave(&q->lock, flags);
3608             __wake_up_common(q, mode, nr_exclusive, 0, key);
3609             spin_unlock_irqrestore(&q->lock, flags);
```

```
3610      }
3611      EXPORT_SYMBOL(__wake_up);



3564      int default_wake_function(wait_queue_t *curr, unsigned mode, int sync,
3565                                void *key)
3566      {
3567              return try_to_wake_up(curr->private, mode, sync);
3568      }
3569      EXPORT_SYMBOL(default_wake_function);


3652      void fastcall complete(struct completion *x)
3653      {
3654              unsigned long flags;
3655
3656              spin_lock_irqsave(&x->wait.lock, flags);
3657              x->done++;
3658      __wake_up_common(&x->wait, TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE,
3659                               1, 0, NULL);
3660              spin_unlock_irqrestore(&x->wait.lock, flags);
3661      }
3662      EXPORT_SYMBOL(complete);
```

9 参考资料

请解释抢占式内核与非抢占式内核的区别联系，http://oldlinux.org/oldlinux/viewthread.php?tid=3024

抢占式内核中的锁问题，http://hi.baidu.com/juventus/blog/item/a71c8701960454d2277fb5f0.html

http://www.linuxforum.net/forum/showflat.php?Cat=&Board=linuxK&Number=610932&page=

http://linux.chinaunix.net/bbs/viewthread.php?tid=912039

Linux kernel design and development

Linux 抢占式内核就是由 Robert Love 修改实现的。在他的书中有如下描述：

User Preemption

User preemption occurs when the kernel is about to return to user-space, need_resched is set, and therefore, the scheduler is invoked. If the kernel is returning to user-space, it knows it is in a safe quiescent state. In other words, if it is safe to continue executing the current task, it is also safe to pick a new task to execute. Consequently, whenever the kernel is preparing to return to user-space either on return from an interrupt or after a system call, the value of need_resched is checked. If it is set, the scheduler is invoked to select a new (more fit) process to execute. Both the return paths for return from interrupt and return from system call are architecture dependent and typically implemented in assembly in entry.S (which, aside from kernel entry code, also contains kernel exit code).

In short, user preemption can occur

When returning to user-space from a system call

When returning to user-space from an interrupt handler

Kernel Preemption

The Linux kernel, unlike most other Unix variants and many other operating systems, is a fully

preemptive kernel. In non-preemptive kernels, kernel code runs until completion. That is, the scheduler is not capable of rescheduling a task while it is in the kernel. kernel code is scheduled cooperatively, not preemptively. Kernel code runs until it finishes (returns to user-space) or explicitly blocks. In the 2.6 kernel, however, the Linux kernel became preemptive: It is now possible to preempt a task at any point, so long as the kernel is in a state in which it is safe to reschedule.

So when is it safe to reschedule? The kernel is capable of preempting a task running in the kernel so long as it does not hold a lock. That is, locks are used as markers of regions of non-preemptibility. Because the kernel is SMP-safe, if a lock is not held, the current code is reentrant and capable of being preempted.

The first change in supporting kernel preemption was the addition of a preemption counter, preempt_count, to each process's thread_info. This counter begins at zero and increments once for each lock that is acquired and decrements once for each lock that is released. When the counter is zero, the kernel is preemptible. Upon return from interrupt, if returning to kernel-space, the kernel checks the values of need_resched and preempt_count. If need_resched is set and preempt_count is zero, then a more important task is runnable and it is safe to preempt. Thus, the scheduler is invoked. If preempt_count is nonzero, a lock is held and it is unsafe to reschedule. In that case, the interrupt returns as usual to the currently executing task. When all the locks that the current task is holding are released, preempt_count returns to zero. At that time, the unlock code checks whether need_resched is set. If so, the scheduler is invoked. Enabling and disabling kernel preemption is sometimes required in kernel code and is discussed in Chapter 9

.

Kernel preemption can also occur explicitly, when a task in the kernel blocks or explicitly calls schedule(). This form of kernel preemption has always been supported because no additional logic is required to ensure that the kernel is in a state that is safe to preempt. It is assumed that the code that explicitly calls schedule() knows it is safe to reschedule.

Kernel preemption can occur

When an interrupt handler exits, before returning to kernel-space

When kernel code becomes preemptible again

If a task in the kernel explicitly calls schedule()

If a task in the kernel blocks (which results in a call to schedule())