一、简介

Ingo Molnar 的实时补丁是完全开源的，它采用的实时实现技术完全类似于Timesys Linux，而且中断线程化的代码是基于TimeSys Linux的中断线程化代码的。这些实时实现技术包括：中断线程化（包括IRQ和softirq）、用Mutex取代spinlock、优先级继承和死锁检测、等待队列优先级化、大内核锁（BKL-Big Kernel Lock）可抢占等。

该实时实现包含了以前的VP补丁（在内核邮件列表这么称呼，即Voluntary Preemption），VP补丁由针对2.4内核的低延迟补丁（low latency patch）演进而来，它使用两种方法来实现低延迟：

一种就是锁分解，即把大循环中保持的锁分解为每一轮循环中都获得锁和释放锁，典型的代码结构示例如下： 锁分解前：

```
spin_lock(&x_lock);
for (…) {
        some operations;
        …
}
spin_unlock(&x_lock);
```

锁分解后：

```
for (…) {
        spin_lock(&x_lock);
        some operations;
        …
        spin_unlock(&x_lock);
}
```

另一种是增加抢占点，即自愿被抢占，下面是一个鼠标驱动的例子：

未增加抢占点以前在文件driver/char/tty_io.c中的一段代码：

```
        /* Do the write .. */
        for (;;) {
                size_t size = count;
                if (size > chunk)
                        size = chunk;
                ret = -EFAULT;
                if (copy_from_user(tty->write_buf, buf, size))
                        break;
                lock_kernel();
                ret = write(tty, file, tty->write_buf, size);
                unlock_kernel();
                if (ret <= 0)
                        break;
                written += ret;
                buf += ret;
                count -= ret;
                if (!count)
                        break;
                ret = -ERESTARTSYS;
                if (signal_pending(current))
```

```
                break;
        }
```

增加抢占点之后：

```
                /* Do the write .. */
        for (;;) {
                size_t size = count;
                if (size > chunk)
                        size = chunk;
                ret = -EFAULT;
                if (copy_from_user(tty->write_buf, buf, size))
                        break;
                lock_kernel();
                ret = write(tty, file, tty->write_buf, size);
                unlock_kernel();
                if (ret <= 0)
                        break;
                written += ret;
                buf += ret;
                count -= ret;
                if (!count)
                        break;
                ret = -ERESTARTSYS;
                if (signal_pending(current))
                        break;
                cond_resched();
        }
```

语句cond_resched()将判断是否有进程需要抢占当前进程，如果是将立即发生调度，这就是增加的强占点。

为了能并入主流内核，Ingo Molnar的实时补丁也采用了非常灵活的策略，它支持四种抢占模式：

1. No Forced Preemption (Server)，这种模式等同于没有使能抢占选项的标准内核，主要适用于科学计算等服务器环境。

2. Voluntary Kernel Preemption (Desktop)，这种模式使能了自愿抢占，但仍然失效抢占内核选项，它通过增加抢占点缩减了抢占延迟，因此适用于一些需要较好的响应性的环境，如桌面环境，当然这种好的响应性是以牺牲一些吞吐率为代价的。

3. Preemptible Kernel (Low-Latency Desktop)，这种模式既包含了自愿抢占，又使能了可抢占内核选项，因此有很好的响应延迟，实际上在一定程度上已经达到了软实时性。它主要适用于桌面和一些嵌入式系统，但是吞吐率比模式2更低。

4. Complete Preemption (Real-Time)，这种模式使能了所有实时功能，因此完全能够满足软实时需求，它适用于延迟要求为100微秒或稍低的实时系统。

实现实时是以牺牲系统的吞吐率为代价的，因此实时性越好，系统吞吐率就越低。

在写本文时最新的实时实现补丁是：

http://people.redhat.com/~mingo/realtime-premmpt/realtime-preempt-2.6.12-rc4-

V0.7.47-03

它自2004年10月发布以来一直更新很频繁，几乎每天都有新版本发布，直到最近才比较稳定。它的很多代码部分已经并入到标准的2.6内核源码数，包括IRQ子系统，那为中断线程化提供了很好的基础；自愿抢占；大内核锁可抢占；这些已经包含在2.6.11中。作者预期，其余的代码部分也将很快进入到主流内核，可能是2.6.12或以后的某个版本。

因此，本文专门对这个实时实现进行详细的实现分析将有重要意义。

回页首

二、中断线程化

中断线程化是实现Linux实时性的一个重要步骤，在Linux标准内核中，中断是最高优先级的执行单元，不管内核当时处理什么，只要有中断事件，系统将立即响应该事件并执行相应的中断处理代码，除非当时中断关闭（即使用local_irq_disable失效了IRQ）。因此，如果系统有严重的网络或I/O负载，中断将非常频繁，实时任务将很难有机会运行，也就是说，毫无实时性可言。中断线程化之后，中断将作为内核线程运行而且赋予不同的实时优先级，实时任务可以有比中断线程更高的优先级，这样，实时任务就可以作为最高优先级的执行单元来运行，即使在严重负载下仍有实时性保证。

中断线程化的另一个重要原因是spinlock被mutex取代。中断处理代码中大量地使用了spinlock，当spinlock被mutex取代之后，中断处理代码就有可能因为得不到锁而需要被挂到等待队列上，但是只有可调度的进程才可以这么做，如果中断处理代码仍然使用原来的spinlock，则spinlock取代mutex的努力将大打折扣，因此为了满足这一要求，中断必须被线程化，包括IRQ和softirq。

在Ingo Molnar的实时补丁中，中断线程化的实现方法是：

对于IRQ，在内核初始化阶段init（该函数在内核源码树的文件init/main.c中定义）调用init_hardirqs（该函数在内核源码树的文件kernel/irq/manage.c中定义）来为每一个IRQ创建一个内核线程，IRQ号为0的中断赋予实时优先级49，IRQ号为1的赋予实时优先级48，依次类推直到25，因此任何IRQ线程的最低实时优先级为25。原来的 do_IRQ 被分解成两部分，架构相关的放在类似于arch/*/kernel/irq.c的文件中，名称仍然为do_IRQ，而架构独立的部分被放在IRQ子系统的位置kernel/irq/handle.c中，名称为__do_IRQ。当发生中断时，CPU将执行do_IRQ来处理相应的中断，do_IRQ将做了必要的架构相关的处理后调用__do_IRQ。函数__do_IRQ将判断该中断是否已经被线程化（如果中断描述符的状态字段不包含SA_NODELAY标志说明中断被线程化了），如果是将唤醒相应的处理线程，否则将直接调用handle_IRQ_event（在IRQ子系统位置的kernel/irq/handle.c文件中）来处理。对于已经线程化的情况，中断处理线程被唤醒并开始运行后，将调用do_hardirq(在源码树的IRQ子系统位置的文件kernel/irq/manage.c中定义)来处理相应的中断，该函数将判断是否有中断需要被处理（中断描述符的状态标志IRQ_INPROGRESS），如果有就调用handle_IRQ_event来处理。handle_IRQ_event将直接调用相应的中断处理句柄来完成中断处理。

如果某个中断需要被实时处理，它可以用SA_NODELAY标志来声明自己非线程化，例如：

系统的时钟中断就是，因为它被用来维护系统时间以及定时器等，所以不应当被线程化。

```
static struct irqaction irq0  =
{ timer_interrupt, SA_INTERRUPT | SA_NODELAY, CPU_MASK_NONE, "timer", NULL, NULL};
```

这是在静态声明时指定不要线程化，也可以在调用request_irq时指定，如：

```
request_irq (HIGHWIRE_SMI_IRQ,
highwire_smi_interrupt, SA_NODELAY, "System Management Switch", NULL))
```

对于softirq，标准Linux内核已经使用内核线程的方式来处理，只是Ingo Molnar的实时补丁做了修改使其易于被抢占，改进了实时性，具体的修改包括：

把ksoftirqd的优先级设置为nice值为-10，即它的优先级高于普通的用户态进程和内核态线程，但它不是实时线程，因此这样一来softirq对实时性的影响将显著减小。

在处理软中断期间，抢占是使能的，这使得实时性更进一步地增强。

在处理软中断的函数___do_softirq中，每次处理完一个待处理的软中断后，都将调用cond_resched_all()，这显著地增加了调度点数，提高了整个系统的实时性。

增加了两个函数_do_softirq和___do_softirq，其中___do_softirq就是原来的__do_softirq，只是增加了调度点。__do_softirq则是对___do_softirq的包装，_do_softirq是对do_softirq的替代，但保留do_softirq用于一些特殊需要。

回页首

三、spinlock转换成mutex

spinlock是一个高效的共享资源同步机制，在SMP（对称多处理器Symmetric Multiple Proocessors）的情况下，它用于保护共享资源，如全局的数据结构或一个只能独占的硬件资源。但是spinlock保持期间将使抢占失效，用spinlock保护的区域称为临界区（Critical Section），在内核中大量地使用了spinlock，有大量的临界区存在，因此它们将严重地影响着系统的实时性。Ingo Molnar的实时补丁使用mutex来替换spinlock，它的意图是让spinlock可抢占，但是可抢占后将产生很多后续影响。

Spinlock失效抢占的目的是避免死锁。Spinlock如果可抢占了，一个spinlock的竞争者将可能抢占该spinlock的保持者来运行，但是由于得不到spinlock将自旋在那里，如果竞争者的优先级高于保持者的优先级，将形成一种死锁的局面，因为保持者无法得到运行而永远不能释放spinlock，而竞争者由于不能得到一个不可能释放的spinlock而永远自旋在那里。

由于中断处理函数也可以使用spinlock，如果它使用的spinlock已经被一个进程保持，中断处理函数将无法继续进行，从而形成死锁，这样的spinlock在使用时应当中断失效来避免这种死锁的情况发生。标准linux内核就是这么做的，中断线程化之后，中断失效就没有必要，因为遇到这种状况后，中断线程将挂在等待队列上并放弃CPU让别的线程或进程来运行。

等待队列就是解决这种死锁僵局的方法，在Ingo Molnar的实时补丁中，每个spinlock都有一个等待队列，该等待队列是按进程或线程的优先级排队的。如果一个进程或线程竞争的spinlock已经被另一个线程保持，它将把自己挂在该spinlock的优先级化的等待队列上，然后发生调度把CPU让给别的进程或线程。

需要特别注意，对于非线程化的中断，必须使用原来的spinlock，原因前面已经讲得很清楚。

原来的spinlock结构如下：

```
typedef struct {
        volatile unsigned long lock;
# ifdef CONFIG_DEBUG_SPINLOCK
        unsigned int magic;
# endif
```

```
# ifdef CONFIG_PREEMPT
        unsigned int break_lock;
# endif
} spinlock_t;
```

它非常简洁，替换成mutex之后，它的结构为：

```
typedef struct {
        struct rt_mutex lock;
        unsigned int break_lock;
} spinlock_t;
```

其中struct rt_mutex结构如下：

```
struct rt_mutex {
        raw_spinlock_t          wait_lock;
        struct plist            wait_list;
        struct task_struct      *owner;
        int                     owner_prio;
# ifdef CONFIG_RT_DEADLOCK_DETECT
        int                     save_state;
        struct list_head        held_list;
        unsigned long           acquire_eip;
        char                    *name, *file;
        int                     line;
# endif
};
```

类型raw_spinlock_t就是原来的spinlock_t。在结构struct rt_mutex中的wait_list字段就是优先级化的等待队列。

原来的rwlock_t结构如下：

```
typedef struct { volatile unsigned long lock; # ifdef CONFIG_DEBUG_SPINLOCK unsigned magic; # endif # ifdef CONFIG_PREEMPT unsigned int break_lock; # endif } rwlock_t;
```

被mutex化的rwlock结构如下：

```
typedef struct { struct rw_semaphore lock; unsigned int break_lock; } rwlock_t;
```

其中rw_semaphore结构为：

```
struct rw_semaphore { struct rt_mutex lock; int read_depth; };
```

rwlock_t和spinlock_t没有本质的不同，只是rwlock_t只能有一个写者，但可以有多个读者，因此使用了字段read_depth，其他都等同于spinlock_t。

如果必须使用原来的spinlock，可以把它声明为raw_spinlock_t，如果必须使用原来的rwlock_t，可以把它声明为raw_rwlock_t，但是对其进行锁或解锁操作时仍然使用同样的函数，静态初始化时必须分别使用RAW_SPIN_LOCK_UNLOCKED和RAW_RWLOCK_UNLOCKED。为什么不同的变量类型可以使用同样的函数操作呢？关键在于使用了gcc的内嵌函数

__builtin_types_compatible_p，下面以spin_lock为例来说明其中的奥妙：

```
#define spin_lock(lock)         PICK_OP(raw_spinlock_t, spin, _lock, lock)
```

PICK_OP的定义为：

```
#define PICK_OP(type, optype, op, lock)                         \
do {                                                            \
        if (TYPE_EQUAL((lock), type))                           \
                _raw_##optype##op((type *)(lock));              \
        else if (TYPE_EQUAL(lock, spinlock_t))                  \
                _spin##op((spinlock_t *)(lock));                \
        else __bad_spinlock_type();                             \
} while (0)
```

TYPE_EQUAL的定义为：

```
#define TYPE_EQUAL(lock, type) \
__builtin_types_compatible_p(typeof(lock), type *)
```

gcc内嵌函数__builtin_types_compatible_p用于判断一个变量的类型是否为某指定的类型，如果是就返回1，否则返回0。

因此，如果一个spinlock的类型如果是spinlock_t，宏spin_lock的预处理结果将是：

```
do {
        if (0)
                _raw_spin_lock((raw_spinlock_t *)(lock));
        else if (1)
                _spin_lock((spinlock_t *)(lock));
        else __bad_spinlock_type;
} while (0)
```

如果一个spinlock的类型为raw_spinlock_t，宏spin_lock的预处理结果将是：

```
do {
        if (1)
                _raw_spin_lock((raw_spinlock_t *)(lock));
        else if (0)
                _spin_lock((spinlock_t *)(lock));
        else __bad_spinlock_type;
} while (0)
```

很明显，如果类型为spinlock_t，将运行函数_spin_lock，而如果类型为raw_spinlock_t，将运行函数_raw_spin_lock。

_spin_lock是新的spinlock的锁实现函数，而_raw_spin_lock就是原来的spinlock的锁实现函数。

等待队列优先级化的目的是为了更好地改善实时性，因为优先级化后，每次当spinlock保持者释放锁时总是唤醒排在最前面的优先级最高的进程或线程，而唤醒的时间复杂度为O(1)。

回页首

四、优先级继承和死锁检测

spinlock被mutex化后会产生优先级逆转（Priority Inversion）现象。 所谓优先级逆转，就是优先级高的进程由于优先级低的进程保持了竞争资源被迫等待，而让中间优先级的进程运行，优先级逆转将导致高优先级进程的抢占延迟增大，中间优先级的进程的执行时间的不确定性导致了高优先级进程抢占延迟的不确定性，因此为了保证实时性，必须消除优先级逆转现象。

优先级继承协议（Priority Inheritance Protocol）和优先级顶棚协议（Priority Ceiling Protocol）就是专门针对优先级逆转问题提出的解决办法。

所谓优先级继承，就是spinlock的保持者将继承高优先级的竞争者进程的优先级，从而能先于中间优先级进程运行，尽可能快地释放锁，这样高优先级进程就能很快得到竞争的spinlock，使得抢占延迟更确定，更短。

所谓优先级顶棚，就是根据静态分析确定一个spinlock的可能拥有者的最高优先级，然后把spinlock的优先级顶棚设置为该确定的值，每次当进程获得该spinlock后，就将该进程的优先级设置为spinlock的优先级顶棚值。

Ingo Molnar的实时补丁实现了优先级继承协议，但没有实现优先级顶棚协议。

Spinlock被mutex化后引入的另一个问题就是死锁，典型的死锁有两种：

一种为自锁，即一个spinlock保持者试图获得它已经保持的锁，很显然，这会导致该进程无法运行而死锁。

另一种为非顺序锁而导致的，即进程 P1已经保持了spinlock LOCKA但是要获得进程P2已经保持的spinlock LOCKB，而进程P2要获得进程P1已经保持的spinlock LOCKA，这样进程P1和P2都将因为需要得到对方拥有的但永远不可能释放的spinlock而死锁。

Ingo Molnar的实时补丁对这两种情况进行了检测，一旦发生这种死锁，内核将输出死锁执行路径并panic。

回页首

五、大内核锁可抢占

大内核锁（BKL---Big Kernel Lock）实质上也是spinlock，只是它一般用于保护整个内核，该锁的保持时间比较长，因此它对整个系统的实时性影响是非常大的，在Ingo Molnar的实时补丁中，大内核锁使用了semaphore来实现，如果内核配置为前面三种抢占模式，struct semaphore是架构相关的，如对于x86，结构定义如下：

```
struct semaphore {
        atomic_t count;
        int sleepers;
        wait_queue_head_t wait;
```

```
};
```

但对于第四种抢占模式，其结构为：

```
struct semaphore {
        atomic_t count;
        struct rt_mutex lock;
};
```

注意新的spinlock定义也包含字段struct rt_mutex lock，因此可抢占大内核锁和新的spinlock共用了低层的处理代码。使用semaphore之后，大内核锁就可抢占了。

回页首

六、架构支持和一些移植以及驱动注意事项

Ingo Molnar的实时补丁支持的架构包括i386、x86_64、ppc和mips，基本上含盖了主流的架构，对于其他的架构，移植起来也是非常容易的。

架构移植主要涉及到以下几个方面：

1．中断线程化

中断线程化有两种做法，一种是利用IRQ子系统的代码，另一种是在架构相关的子树实现，前一种方法利用的是已有的中断线程化代码，因此移植时几乎不需要做什么工作，但是对一些架构，这种方法缺乏灵活性，尤其是一些架构中断处理比较特别时，可能会是IRQ子系统的中断线程化代码部分变的越来越丑陋，因此对于这种架构，后一种方法就有明显优势，当然在后一种方法中仍然可以拷贝IRQ子系统内的大部分线程化处理代码。

中断线程化要求一些spinlock或rwlock必须是raw_*类型的，而且一些IRQ必须是非线程化的，如时钟中断、级联中断等。这些是中断线程化的必要前提。

2．一些架构相关的代码

有一些变量定义在架构相关的子树下，如hardirq_preemption等，还有就是需要对entry.S做一些修改，因为增加了一个新的调用preempt_schedule_irq，它要求在调用之前失效中断。还有就是一些调试代码支持，那是完全架构相关的必须重新实现，如mcount。

3．架构相关的semaphore定义必须在第四种抢占模式下失效

前面已经讲过，如果使能第四种抢占模式，将使用新定义的semaphore，它是架构无关的，相应的处理代码也是架构无关的，因此原来的架构相关的定义和处理代码必须失效，这需要修改相应的.h、.c和Makefile。

4．一些spinlock必须声明为raw_*类型的

在架构相关的子树中，一些spinlock必须声明为raw_*类型的，静态初始化也必须修改为RAW_*，一些外部声名也得做相应的改动。

5．在打开第四种抢占模式或中断线程化使能之后，一些编程逻辑要求已经发生了变化。

中断线程化后，在中断处理函数中失效中断不在需要，因为如果中断处理线程在中断失效后想

得到spinlock时，将可能发生上下文切换，新的实时实现认为这种状况不应当发生将输出警告信息。

原来用中断失效保护共享资源，现在完全可以用抢占失效来替代，因此不是万不得已，建议不使用中断失效。 在网卡驱动的发送处理函数中不能失效中断，因此原来显式得失效中断的函数应当被替换，如：

local_irq_save应当变成为local_irq_save_nort
local_irq_restore应当变成为local_irq_restore_nort

网络的核心代码将主动检测这种情况，如果中断失效了，将重新打开中断，但是将输出警告信息。

在保持了raw_spinlock之后不能在试图获得新的spinlock类型的锁，因为raw_spinlock是抢占失效的，但是新的spinlock却能够导致进程睡眠或发生抢占。

对于新的semaphore，必须要求执行down和up操作的是同一个进程，否则优先级继承和死锁检测将无法实现。而且代码本身也将操作失败。

关于作者

杨燚，计算机科学硕士，毕业于中科院计算技术研究所，有 4 年的 Linux 内核编程经验，目前从事嵌入式实时 Linux 的开发与性能测试。您可以通过yang.yi@bmrtech.com或yyang@ch.mvista.com与作者联系。