优先级继承运用于Linux内核信号量的研究与实现①

王亚军 (中国人民武装警察部队学院 河北 廊坊 065000)

摘 要: 运用优先级继承协议解决Linux内核信号量操作中存在的优先级倒转问题，是将Linux运用于实时系统的重要手段。在具体分析Linux内核信号量操作源代码的基础上，针对优先级倒转问题修改内核实现基本优先级继承协议，并在PCM-3350嵌入式计算机系统上测试通过。

关键词: 优先级倒转; 优先级继承; 信号量; 实时性; 嵌入式系统

Research and Realization of Priority Inheritance Applied in Linux Kernel Semaphore

WANG Ya-Jun

(Chinese People's Armed Police Forces Academy, Langfang 065000, China)

Abstract: It is very important to solve the problem of priority inversion by means of priority inheritance when applying Linux in real-time embedded system. Based on the analysis of the Linux source codes about kernel semaphore operation, this paper realizes the base priority inheritance protocol by modifying the Linux kernel, and passes the test on the embedded system of PCM-3350.

Keywords: priority inversion; priority inheritance; semaphore; real-time; embedded system

1 引言

为了避免多个进程同时进入临界区，Linux 内核采取了严格的同步机制，信号量就是其中之一。如果一个进程试图获得一个已经被占用的内核信号量，该进程就会被推进一个等待队列进入“睡眠”状态，从而释放处理器，使处理器可以执行其它进程的代码。当持有信号量的进程释放信号量以后，在等待队列中睡眠的进程将被唤醒，并再次试图获得该信号量以进入临界区[1]。下面以 Linux-2.6.10 内核为例，在分析内核信号量操作源代码的基础上，基于实时应用实现基本优先级继承协议。

2 Linux-2.6.10内核信号量操作分析

Linux-2.6.10 内核中最常用的内核信号量是在文件 include/asm-i386/semaphore.h 中定义的 struct semaphore 数据结构。内核根据不同的情况，为内核信号量提供了几组不同的 P 和 V 操作函数，其中以 down()和 up()函数最有代表性。down()和 up()函数是在 include/ asm-i386/ semaphore.h 文件中定义的，函数头如下：static inline void down(struct semaphore * sem) static inline void up(struct semaphore * sem)

其中 down()操作先使 sem->count 减 1，减后的结果若为非负数，表明当前进程可以进入临界区，则 down()操作结束；减后的结果若为负数，表明进程不能进入临界区，则进程在 down()操作中调用 __down_failed()操作函数进入深度睡眠状态。函数 __down_failed()的作用是通过调用函数__down()实现的，而函数__down()中调用的函数 add_wait_queue_exclusive_locked()的作用则是把代表当前进程的等待项链入等待队列的尾部。

函数 up()首先递增 sem->count 的值，递增后

① 收稿时间:2010-03-17;收到修改稿时间:2010-04-23

的结果若为非负数就调用__up_wakeup()函数。函数__up_wakeup()的作用是通过间接调用函数wake_up()实现的。函数wake_ up()依次唤醒等待队列中所有WQ_FLAG_EXCLUSIVE标志位为0的进程和第一个WQ_FLAG_EXCLUSIVE标志位为1的进程[2]。

从代码中可以看出，Linux 内核信号量操作中存在两种优先级倒转现象：(1)等待队列中的进程是按照“先进先出”的原则被唤醒进入临界区的，进程的优先级并没有起作用。 (2)信号量所保护的临界区内允许进程调度，临界区内的低优先级进程一旦受阻进入睡眠就很难再得到机会运行，从而阻塞了等待队列中的高优先级进程。这种阻塞可能是有限的，也可能是无限的[3]。

为了将 Linux 内核用于实时系统，应该对 Linux 内核信号量操作中存在的优先级倒转问题加以解决。对于第一种优先级倒转现象，只需要修改等待队列的排队规则，让进程按照优先级排队；而对于第二种优先级倒转现象，则需要实施优先级继承[3]。

3 基本优先级继承协议工作原理

优先级继承协议是 L.Sha 等人于 1990 年提出的用于解决优先级倒转问题的重要方法，分为基本优先级继承和封顶优先级继承两个协议。一般情况下，由于可能进入临界区的进程集合难于预测，所以应该采用基本优先级继承协议，其工作原理是：(1) 一个获得处理器的进程 T，在运行中可能尝试获得互斥信号量 S 以进入临界区 Z。如果 S 已经被别的进程占有，则 T 在 S 上阻塞睡眠；否则 T 执行 P(S)操作进入 Z。当 T 执行 V(S)操作退出 Z 时，若有被 T 阻塞的高优先级进程，则 T 唤醒该进程。(2) T 首先使用自身固有优先级运行，当它进入 Z 并阻塞了更高优先级进程时，T 将继承被它阻塞的所有进程中的最高优先级并继续运行。当 T 退出 Z 时，恢复原来的优先级。(3) 优先级继承具有传递性，当进程嵌套进入几个临界区时，可能会发生嵌套优先级继承现象。(4) 如果有比当前进程优先级更高的进程就绪，则当前进程将被剥夺处理器使用权[4]。

具体说来，修改内核实现优先级继承时需要考虑如下几个方面：(1)高优先级进程将自身的优先级“借给”低优先级进程以后，要求内核安排一次进程调度，使得低优先级进程能够获得 CPU 的使用权。(2)低优先级进程可能需要嵌套进入多个内核临界区，可能因某个里层的临界区被另一个低优先级进程占据而被迫挂入等待队列。为此，需要实现高优先级的可传递性。(3)如果当低优先级进程每退出一个内核临界区都恢复一次优先级，那么将增加代码的复杂度而降低效率。鉴于进程在之后的操作中是否会继承更高的优先级不可预测，并且进程以较高的优先级运行有利于推动它尽快退出所有内核临界区，所以直到进程退出所有内核临界区后，再一次性恢复其原始的优先级。(4)进程在内核临界区中可能创建子进程，也可能修改自身的调度策略和优先级，为此需要修改相关代码。

4 基本优先级继承协议的实现

如前所述，当把 Linux 内核用于实时系统时，需要解决内核信号量操作中存在的两种优先级倒转现象。

4.1 针对第一种优先级倒转现象：

需要对内核中等待项的入队列操作进行修改。具体地说，当那些标志位 WQ_FLAG_EXCLUSIVE 为 1 的等待项入队列时，如果等待项中的进程为普通进程，就直接插入到队尾；如果是实时进程，就按照优先级顺序插入到等待队列中，使得所有实时进程都排在普通进程的前面，高优先级的实时进程排在低优先级的实时进程的前面。为此，需要修改文件include\linux\wait.h 中的函数 __add_wait_queue_tail()如下：

```
static inline void __add_wait_queue_tail(wait_queue_head_t *head,wait_queue_t*new)
{if(new->task->policy)/*实时进程入队列,按优先级排序*/
{wait_queue_t *wqt;
struct list_head *tmp;
list_for_each(tmp, &(head->task_list))
{wqt = list_entry(tmp, wait_queue_t, task_list);
if(wqt-> flags & WQ_FLAG_EXCLUSIVE && new->task-> prio < wqt->task->prio) break;
}
list_add_tail(&new->task_list, tmp);
} else /*普通进程入队列,直接插入队尾*/
list_add_tail(&new->task_list,&head->task_list);
```

}

4.2 针对第二种优先级倒转现象：

首先需要修改内核信号量的数据结构定义 struct semaphore，在该结构体末尾增加一个指针“struct task_struct *tsk;”，指向进入临界区的进程。然后，修改进程控制块的数据结构定义 struct task_struct，增加下列成员：

unsigned long orig_policy,orig_rt_priority; /*记录进程原调度策略和实时优先级*/

int sem_count; /*记录进程已经获得并且尚未释放的内核信号量个数*/

struct semaphore *sem; /*指向进程争取获得的信号量*/

wait_queue_t *wqt; /*指向进程现在所在的等待项*/

其中，成员 orig_policy 和 orig_rt_priority 在函数 copy_process()中设置子进程控制块时和在函数 pre__up_wakeup()中恢复优先级时需要用到。成员 sem 和 wqt 在函数 add_wait_queue_exclusive_locked()中调用该函数自身实施递归优先级继承时需要用到。成员 sem_count 在函数 setscheduler()中设置进程的调度策略和实时优先级时需要用到，用以判断进程是否位于内核临界区中。

由于在进程控制块中增加了指向进程所在的等待项的指针 wqt，所以还要在文件 include/linux/wait.h 中的一个宏定义 DECLARE_WAITQUEUE()后面增加一条语句为该指针赋值：tsk->wqt=&name; /*进程 tsk 被加入等待项 name*/接下来，修改文件 include/asm-i386/semaphore.h 中的函数 down(),将其中一行代码 “js 2f\n” 改为 “jmp 2f\n”。也就是说，修改后，在函数 down()中，无论 sem->count 减 1 后的结果为正为负，都要间接调用文件 arch/i386/kernel/semaphore.c 中的函数 __down()。在函数__down()中新增几行代码如下：

fastcall void __sched __down(struct semaphore * sem)

{if (sem->count>=0) goto SUCCED;/*判断进程是否能够进入临界区*/

struct task_struct *tsk = current;

tsk->sem=sem; /*当前进程争取获得信号量 sem */

……

SUCCED: /*进程即将进入临界区*/

sem->tsk= tsk; /*当前进程 tsk 已经获得信号量 sem */

tsk->sem_count++;

tsk->sem=NULL; /*当前进程没有要争取获得的信号量*/

tsk->wqt=NULL; /*当前进程现在没有位于等待项中*/

}

在函数__down()中，无法进入临界区的进程是通过在函数 add_wait_queue_exclusive_locked()中调用上面修改的函数__add_wait_queue_tail()挂入等待队列的。由于普通进程没有实时性要求，所以普通进程即使被挂入队首的位置，也不必实施优先级继承。而实时进程有实时性要求，如果实时进程被挂入队首且比临界区中的进程至少高两个优先级，那么就实施基本优先级继承。在实施继承之后，如果临界区中的进程已经嵌套进入另一个临界区并被挂入等待队列，则需要调整其在等待队列中的位置并可能需要递归实施优先级继承。为此，修改函数如下：

static inline void add_wait_queue_exclusive_locked (wait_queue_head_t *q, wait_queue_t * wait)

{……

if(wait->task->policy==NORMAL)

return;/*普通进程入队列,不实施优先级继承*/

struct semaphore *sem= wait->task->sem;

If(q-> task_list.next== &(wait->task_list) && wait-> task->prio+1< sem-> tsk-> prio) /*若实时进程插入队首且比临界区中的进程至少高两个优先级，则实施优先级继承*/

{runqueue_t *rq;

prio_array_t *array;

unsigned long flags;

struct task_struct *semtsk= sem->tsk;

read_lock_irq(&tasklist_lock);

rq = task_rq_lock(semtsk, &flags);

if(array=semtsk - >array)deactivate_task (semtsk, rq); /*把临界区中的进程移出可执行队列

```
*/
    __setscheduler(semtsk, SCHED_FIFO, wait->task-> rt_priority); /*实施优先级继承*/
    if (array)
    {__activate_task(semtsk, rq); /*把临界区中的进程移入可执行队列*/
    resched_task(rq->curr);/*要求内核重新调度*/
    }
    task_rq_unlock(rq, &flags);
    read_unlock_irq(&tasklist_lock);
    if(semtsk->wqt)  /*如果临界区中的进程已经嵌套进入另一个临界区并被挂入等待队列*/
    {spin_unlock_irqrestore(&q->lock, flags);
    spin_lock_irqsave(&semtsk->sem->wait.lock, flags);
    remove_wait_queue_locked(&semtsk->sem->wait,semtsk->wqt);/*进程移出等待队列*/
    add_wait_queue_exclusive_locked(&semtsk->sem->wait, semtsk->wqt); /*进程重新挂入等待队列，递归实施优先级继承*/
    spin_unlock_irqrestore(&semtsk->sem->wait.lock, flags);
    spin_lock_irqsave(&q->lock, flags);
    }/*end if*/
    }/*end if*/
    }
```

相应地，当进程即将退出临界区时，如果该进程的优先级“借自”别的进程，那么还要恢复其原来的优先级。为此，需要修改文件 include/asm-i386/semaphore.h 中的函数 up(),在语句 LOCK "incl %0\n\t"执行之前需要调用在文件 arch/i386/kernel/semaphore.c 中新增的函数 pre__up_wakeup(),恢复进程原来的优先级。

```
    fastcall void pre__up_wakeup(struct semaphore *sem)
    {struct task_struct *semtsk= sem->tsk;
    semtsk-> sem_count--;
    If(!semtsk->sem_count&&(semtsk->policy!=semtsk->
orig_policy||semtsk->rt_priority!=semtsk->orig_rt_ priority))  /*如果进程即将退出所有内核临界区,并且发生过优先级继承,那么恢复进程原始的优先级*/
    {runqueue_t *rq;
    unsigned long flags;
    read_lock_irq(&tasklist_lock);
    rq = task_rq_lock(semtsk, &flags);
    deactivate_task(semtsk,rq);/*把临界区中进程移出可执行队列*/
    __setscheduler(semtsk,semtsk->orig_policy,semtsk-> orig_rt_priority); /*恢复进程原调度策略和优先级*/
    __activate_task(semtsk, rq); /*把临界区中的进程移入可执行队列*/
    sem->tsk=NULL;
    task_rq_unlock(rq, &flags);
    read_unlock_irq(&tasklist_lock);
    }/*end if*/
    }
```

当进程无论在内核临界区内还是内核临界区外创建子进程时，子进程的调度策略和实时优先级都应该来自父进程原来的调度策略和实时优先级。为此，在文件 kernel/fork.c 中，需要修改函数 do_fork()中调用的函数 copy_process()，在父进程把进程控制块信息复制给子进程操作 dup_task_struct()完成后，加入如下代码修改子进程的调度策略和实时优先级等信息：

```
    p->policy=p->orig_policy;
    p->rt_priority=p->orig_rt_priority;
    p->sem_count=0;
    p->sem=NULL;
    p->wqt=NULL;
```

另外，还要修改文件 include\linux\Init_task.h 中的宏定义 INIT_TASK()，在语句“.policy=SCHED_NORMAL,\”下面加入下列语句来补充设置初始进程的控制块：

```
    .orig_policy=SCHED_NORMAL,\
    .sem_count=0,\
    .wqt=NULL,\
    .sem=NULL,\
```

当进程重新设置自身或其它进程的调度策略和实时优先级时，相关操作也要修改。具体地说，需要在

文件 kernel\sched.c 中的函数 setscheduler()中，将语句“__setscheduler(p, policy, lp.sched_priority);”修改为：

/*如果进程在内核临界区以外，或者进程实时优先级将被提高，那么修改进程的调度策略和实时优先级，否则只修改进程原来的调度策略和实时优先级(因为进程在退出内核临界区时会据此恢复调度策略和实时优先级)*/

unsigned long flags= policy != SCHED_NORMAL && (p->policy==SCHED_NORMAL||lp.sched_priority>p-> rt_priority); /*判断进程实时优先级是否将被提高*/

If(!p->sem_count || flags) /*如果进程在内核临界区以外，或其实时优先级将被提高*/

__setscheduler(p, policy, lp.sched_priority);

p->orig_policy=policy;

p->orig_rt_priority= lp.sched_priority;

另外，如果进程在等待队列中并且其实时优先级得到提高，那么还需要重新调整进程在等待队列中的位置。如果调整后进程位于等待队列的队首，还可能递归实施优先级继承。为此，需要在函数 setscheduler() 中的语句“read_unlock_irq(&tasklist_lock);”后增加下列语句：

if(p->wqt && flags) /*如果进程在等待队列中，并且其实时优先级已经得到提高*/

{spin_lock_irqsave(&p->sem->wait.lock, flags);

remove_wait_queue_locked(&p->sem->wait, p->wqt);/*把进程移出等待队列*/

add_wait_queue_exclusive_locked(&p->sem->wait,p-> wqt); /*进程重挂入等待队列,可能递归实施优先级继承*/

spin_unlock_irqrestore(&p->sem->wait.lock, flags);

}

除了 down()和 up()函数之外，内核信号量还有另外几组 P 和 V 操作函数，修改方法同上。

5 新内核实时性分析与测试

综上所述，修改内核源代码解决了内核信号量中存在的两种优先级倒转现象。针对第一种优先级倒转现象所做的修改，标志位为 1 的等待项的入队列操作所需时间与队列中已有的标志位为 1 的等待项个数 n 有关，时间复杂度为 O(n)，比原来内核相关代码的时间复杂度 O(1)略有增加。这样修改的结果使得优先级最高的实时进程在队列中的等待时间与队列中已有的标志位为 1 的等待项的个数 n 无关，其时间复杂度和等待时间的不确定性在更高的量级上由 O(n)降为 O(1)，提高了内核的实时性。

针对第二种优先级倒转现象所做的修改，带来的空间/时间复杂度上的增加都是常量级的，对内核的基本架构影响很小。基本优先级继承的具体实现是通过修改函数 add_wait_queue_exclusive_locked()实现的。如果临界区中的进程嵌套进入多个临界区并被挂入等待队列，那么该函数将可能递归调用自身来实现优先级继承的传递性。即使在极端情况下，低优先级进程也能够“借用”高优先级运行并退出临界区，将 CPU 使用权交给等待的高优先级实时进程，不会发生无限制的优先级倒转现象。

为了在 PCM-3350 嵌入式计算机系统(Intel 486 CPU，64MB 内存)上对修改前后的内核分别进行测试，在内核中自定义两个全局的内核信号量 sem_a 和 sem_b，再自定义两个新的系统调用 sys_a 和 sys_b。sys_a 在操作中嵌套获得信号量 sem_a 和 sem_b，sys_b 在操作中获得信号量 sem_b，两个系统调用在临界区中都执行循环语句。接下来，创建一个普通进程 A、一个低优先级实时进程 B、一组中优先级实时进程 C1~C5(都比 B 高两个优先级)、一个高优先级实时进程 D(比 B 高四个优先级)。为了记录进程的执行流程，在内核代码中插入函数 do_gettimeofday()并在进程代码中插入函数 gettimeofday()以获得系统时刻，并将数据通过 printk()函数(在内核中)和 printf()函数(在进程中)打印在屏幕上。对原内核的测试过程如下：A 首先投入运行，执行 sys_b，获得 sem_b，执行循环语句中途被 B 抢占。B 执行 sys_a，获得 sem_a，尝试获得 sem_b 失败而挂入 sem_b 的等待队列。A 第二次投入运行又被 D 抢占。D 执行 sys_a，尝试获得 sem_a 失败而挂入 sem_a 的等待队列。A 第三次投入运行，又被 C1~C5 抢占。C1~C5 依次完成各自的操作后，A 第四次投入运行并且退出临界区释放 sem_b，唤醒 B。B 获得 sem_b，完成操作后依次释放 sem_b 和 sem_a，并唤醒 D。D 依次获得 sem_a

(下转第 82 页)

(上接第 207 页)

和 sem_b，完成操作后依次释放 sem_b 和 sem_a。可见，在原内核中发生了优先级倒转。而对新内核的测试表明，B 挂入 sem_b 的等待队列后对 A 实施了优先级继承。D 挂入 sem_a 的等待队列后对 B 实施了优先级继承，B 又对 A 实施了优先级继承。A 第三次投入运行后没有被 C1~C5 抢占，而是尽快退出临界区并唤醒 B，B 再尽快退出两层临界区并唤醒 D，D 完成两层临界区操作后才将 CPU 的使用权交给 C1~C5，D 的执行时间缩短了 650 毫秒。可见，在新内核中实现了基本优先级继承协议，内核的实时性得到有效提高。

6 结语

在 Linux 内核中，信号量及其操作是内核对访问共享资源的多个进程进行同步的关键，对系统的性能具有重要影响。当把 Linux 内核用于实时系统时，需要运用优先级继承协议解决内核信号量操作中存在的优先级倒转问题，这对于提高内核的实时性具有重要意义。

参考文献

1 LOVE，ROBERT. Linux Kernel Development. New York: MACMILLAN COMPUTER PUB, 2005.

2 毛德操,胡希明. Linux 内核源代码情景分析.杭州:浙江大学出版社, 2002.

3 毛德操,胡希明.嵌入式系统.杭州:浙江大学出版社, 2003.

4 Sha L, RajKumar R, Lehoczky J. Priority inheritance protocols: An approach to real time synchronization. IEEE Transactions on Computers, 1990,39(9):1175—1185.