Linux的原子操作与同步机制

2014-04-09 18:45 by Florian, 2942 阅读, 3 评论, 收藏, 编辑

Linux的原子操作与同步机制

并发问题

现代操作系统支持多任务的并发，并发在提高计算资源利用率的同时也带来了资源竞争的问题。例如C语言语句“count++;”在未经编译器优化时生成的汇编代码为。

count++;	mov eax, [count]
	inc eax
	mov [count], eax

当操作系统内存在多个进程同时执行这段代码时，就可能带来并发问题。

进程1	进程2
mov eax, [count]	等待
等待	mov eax, [count]
等待	inc eax
等待	mov [count], eax
inc eax	等待
mov [count], eax	等待

假设count变量初始值为0。进程1执行完“mov eax, [count]”后，寄存器eax内保存了count的值0。此时，进程2被调度执行，抢占了进程1的CPU的控制权。进程2执行“count++;”的汇编代码，将累加后的count值1写回到内存。然后，进程1再次被调度执行，CPU控制权回到进程1。进程1接着执行，计算count的累加值仍为1，写回到内存。虽然进程1和进程2执行了两次“count++;”操作，但是count实际的内存值为1，而不是2！

单处理器原子操作

解决这个问题的方法是，将“count++;”语句翻译为单指令操作。

count++;	inc [count]

Intel x86指令集支持内存操作数的inc操作，这样“count++;”操作可以在一条指令内完成。因为进程的上下文切换是在总是在一条指令执行完成后，所以不会出现上述的并发问题。对于

单处理器来说，一条处理器指令就是一个原子操作。

多处理器原子操作

但是在多处理器的环境下，例如SMP架构，这个结论不再成立。我们知道“inc [count]”指令的执行过程分为三步：

1）从内存将count的数据读取到cpu。

2）累加读取的值。

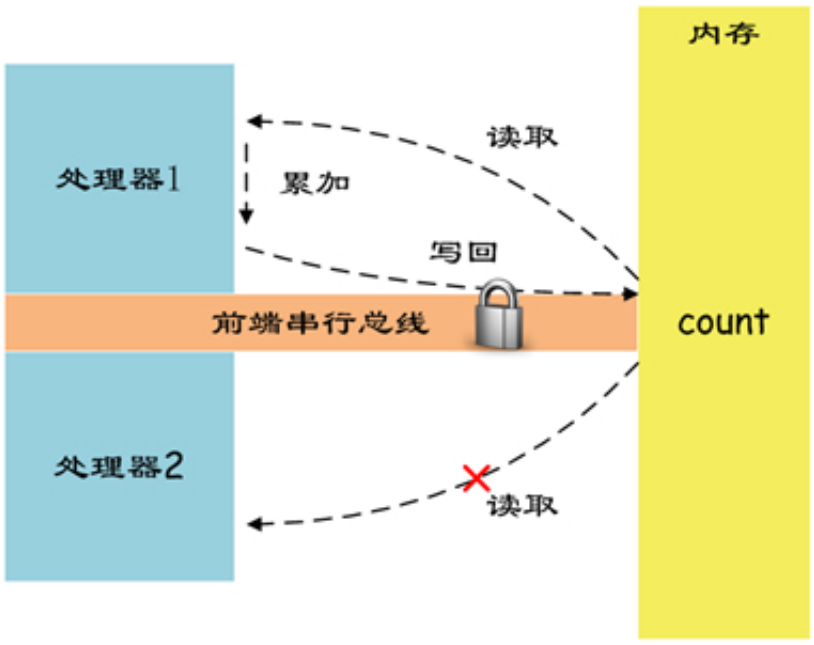
3）将修改的值写回count内存。

这又回到前面并发问题类似的情况，只不过此时并发的主题不再是进程，而是处理器。

Intel x86指令集提供了指令前缀lock用于锁定前端串行总线（FSB），保证了指令执行时不会受到其他处理器的干扰。

count++;	lock inc [count]

使用lock指令前缀后，处理器间对count内存的并发访问（读/写）被禁止，从而保证了指令的原子性。



x86原子操作实现

Linux的源码中x86体系结构原子操作的定义文件为。

linux2.6/include/asm-i386/atomic.h

文件内定义了原子类型atomic_t，其仅有一个字段counter，用于保存32位的数据。

```
typedef struct { volatile int counter; } atomic_t;
```

其中原子操作函数atomic_inc完成自加原子操作。

```
/**
 * atomic_inc - increment atomic variable
 * @v: pointer of type atomic_t
 *
 * Atomically increments @v by 1.
```

```
 */
static __inline__ void atomic_inc(atomic_t *v)
{
    __asm__ __volatile__(
       LOCK "incl %0"
       :"=m" (v->counter)
       :"m" (v->counter));
}
```

其中LOCK宏的定义为。

```
#ifdef CONFIG_SMP
    #define LOCK "lock ; "
#else
    #define LOCK ""
#endif
```

可见，在对称多处理器架构的情况下，LOCK被解释为指令前缀lock。而对于单处理器架构，LOCK不包含任何内容。

arm原子操作实现

在arm的指令集中，不存在指令前缀lock，那如何完成原子操作呢？

Linux的源码中arm体系结构原子操作的定义文件为。

linux2.6/include/asm-arm/atomic.h

其中自加原子操作由函数atomic_add_return实现。

```
static inline int atomic_add_return(int i, atomic_t *v)
{
    unsigned long tmp;
    int result;
    __asm__ __volatile__("@ atomic_add_return\n"
       "1:     ldrex   %0, [%2]\n"
       "       add     %0, %0, %3\n"
       "       strex   %1, %0, [%2]\n"
       "       teq     %1, #0\n"
       "       bne     1b"
       : "=&r" (result), "=&r" (tmp)
       : "r" (&v->counter), "Ir" (i)
       : "cc");
    return result;
}
```

上述嵌入式汇编的实际形式为。

```
1:
ldrex   [result], [v->counter]
add     [result], [result], [i]
strex   [temp], [result], [v->counter]
teq     [temp], #0
bne     1b
```

ldrex指令将v->counter的值传送到result，并设置全局标记“Exclusive”。

add指令完成“result+i”的操作，并将加法结果保存到result。

strex指令首先检测全局标记“Exclusive”是否存在，如果存在，则将result的值写回counter->v，并将temp置为0，清除“Exclusive”标记，否则直接将temp置为1结束。

teq指令测试temp值是否为0。

bne指令temp不等于0时跳转到标号1，其中字符b表示向后跳转。

整体看来，上述汇编代码一直尝试完成“v->counter+=i”的操作，直到temp为0时结束。

使用ldrex和strex指令对是否可以保证add指令的原子性呢？假设两个进程并发执行“ldrex+add+strex”操作，当进程1执行ldrex后设定了全局标记“Exclusive”。此时切换到进程2，执行ldrex前全局标记“Exclusive”已经设定，ldrex执行后重复设定了该标记。然后执行add和strex指令，完成累加操作。再次切换回进程1，接着执行add指令，当执行strex指令时，由于“Exclusive”标记被进程2清除，因此不执行传送操作，将temp设置为1。后继teq指令测定temp不等于0，则跳转到起始位置重新执行，最终完成累加操作！可见ldrex和strex指令对可以保证进程间的同步。多处理器的情况与此相同，因为arm的原子操作只关心“Exclusive”标记，而不在乎前端串行总线是否加锁。

在ARMv6之前，swp指令就是通过锁定总线的方式完成原子的数据交换，但是影响系统性能。ARMv6之后，一般使用ldrex和strex指令对代替swp指令的功能。

自旋锁中的原子操作

Linux的源码中x86体系结构自旋锁的定义文件为。

linux2.6/include/asm-i386/spinlock.h

其中__raw_spin_lock完成自旋锁的加锁功能

```
#define __raw_spin_lock_string \
    "\n1:\t" \
    "lock ; decb %0\n\t" \
    "jns 3f\n" \
    "2:\t" \
    "rep;nop\n\t" \
    "cmpb $0,%0\n\t" \
    "jle 2b\n\t" \
    "jmp 1b\n" \
    "3:\n\t"
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    __asm__ __volatile__(
        __raw_spin_lock_string
        :"=m" (lock->slock) : : "memory");
}
```

上述代码的实际汇编形式为。

```
1:
lock    decb [lock->slock]
jns     3
2:
rep     nop
cmpb    $0, [lock->slock]
jle     2
jmp     1
3:
```

其中lock->slock字段初始值为1，执行原子操作decb后值为0。符号位为0，执行jns指令跳转到3，完成自旋锁的加锁。

当再次申请自旋锁时，执行原子操作decb后lock->slock值为-1。符号位为1，不执行jns指令。进入标签2，执行一组nop指令后比较lock->slock是否小于等于0，如果小于等于0回到标签2进行循环（自旋）。否则跳转到标签1重新申请自旋锁，直到申请成功。

自旋锁释放时会将lock->slock设置为1，这样保证了其他进程可以获得自旋锁。

信号量中的原子操作

Linux的源码中x86体系结构自旋锁的定义文件为。

linux2.6/include/asm-i386/semaphore.h

信号量的申请操作由函数down实现。

```
/*
 * This is ugly, but we want the default case to fall through.
 * "__down_failed" is a special asm handler that calls the C
 * routine that actually waits. See arch/i386/kernel/semaphore.c
 */
static inline void down(struct semaphore * sem)
{
    might_sleep();
    __asm__ __volatile__(
        "# atomic down operation\n\t"
        LOCK "decl %0\n\t"     /* --sem->count */
        "js 2f\n"
        "1:\n"
        LOCK_SECTION_START("")
        "2:\tlea %0,%%eax\n\t"
        "call __down_failed\n\t"
        "jmp 1b\n"
        LOCK_SECTION_END
        :"=m" (sem->count)
        :
        :"memory","ax");
}
```

实际的汇编代码形式为。

```
lock    decl [sem->count]
js 2
1:
<========== another section ==========>
2:
lea     [sem->count], eax
call    __down_failed
jmp 1
```

信号量的sem->count一般初始化为一个正整数，申请信号量时执行原子操作decl，将sem->count减1。如果该值减为负数（符号位为1）则跳转到另一个段内的标签2，否则申请信号量成功。

标签2被编译到另一个段内，进入标签2后，执行lea指令取出sem->count的地址，放到eax寄存器作为参数，然后调用函数__down_failed表示信号量申请失败，进程加入等待队列。最后跳回

标签1结束信号量申请。

信号量的释放操作由函数up实现。

```
/*
 * Note! This is subtle. We jump to wake people up only if
 * the semaphore was negative (== somebody was waiting on it).
 * The default case (no contention) will result in NO
 * jumps for both down() and up().
 */
static inline void up(struct semaphore * sem)
{
    __asm__ __volatile__(
        "# atomic up operation\n\t"
        LOCK "incl %0\n\t"     /* ++sem->count */
        "jle 2f\n"
        "1:\n"
        LOCK_SECTION_START("")
        "2:\tlea %0,%%eax\n\t"
        "call __up_wakeup\n\t"
        "jmp 1b\n"
        LOCK_SECTION_END
        ".subsection 0\n"
        :"=m" (sem->count)
        :
        :"memory","ax");
}
```

实际的汇编代码形式为。

```
lock   incl sem->count
jle     2
1:
<========== another section ==========>
2:
lea    [sem->count], eax
call   __up_wakeup
jmp    1
```

释放信号量时执行原子操作incl将sem->count加1，如果该值小于等于0，则说明等待队列有阻塞的进程需要唤醒，跳转到标签2，否则信号量释放成功。

标签2被编译到另一个段内，进入标签2后，执行lea指令取出sem->count的地址，放到eax寄存器作为参数，然后调用函数__u

p_wakeup唤醒等待队列的进程。最后跳回标签1结束信号量释放。

总结

本文通过对操作系统并发问题的讨论研究操作系统内的原子操作的实现原理，并讨论了不同体系结构下Linux原子操作的实现，最后描述了Linux操作系统如何利用原子操作实现常见的进程同步机制，希望对你有所帮助。

« 上一篇：ARM的常数表达式

» 下一篇：源文件移动后gdb不显示代码的原因

分类: Linux

标签: Linux, 原子操作, 同步机制

最新IT新闻:
· Facebook公开总部超大新建筑，网友神回复
· 大脑如同编程，bug如何修复?
· 体重12克的小鸟三天连续飞行2700公里！
· 大批基金公司暂停与支付宝合作 因不愿被宰
· 索尼变卖一半奥林巴斯股权 套现4亿美元
» 更多新闻...

最新知识库文章:
· 我们为什么要思考算法
· 事件流如何提高应用程序的扩展性、可靠性和可维护性
· Web动效研究与实践
· 关于工作效率的心得分享
· 编码之道：取个好名字很重要
» 更多知识库文章...