Linux内核部件分析

原子性操作atomic_t

[日期：2011-10-06] 来源：Linux社区 作者：qb_2008 [字体：大 中 小]

在任何处理器平台下，都会有一些原子性操作，供操作系统使用，我们这里只讲x86下面的。在单处理器情况下，每条指令的执行都是原子性的，但在多处理器情况下，只有那些单独的读操作或写操作才是原子性的。为了弥补这一缺点，x86提供了附加的lock前缀，使带lock前缀的读修改写指令也能原子性执行。带lock前缀的指令在操作时会锁住总线，使自身的执行即使在多处理器间也是原子性执行的。xchg指令不带lock前缀也是原子性执行，也就是说xchg执行时默认会锁内存总线。原子性操作是线程间同步的基础，linux专门定义了一种只进行原子操作的类型atomic_t，并提供相关的原子读写调用API。本节就来分析这些原子操作在x86下的实现。

```
1. typedef struct {
2.     volatile int counter;
3. } atomic_t;
```

原子类型其实是int类型，只是禁止寄存器对其暂存。

```
1. #define ATOMIC_INIT(i)  { (i) }
```

原子类型的初始化。32位x86平台下atomic API在arch/x86/include/asm/atomic_32.h中实现。

```
1. static inline int atomic_read(const atomic_t *v)
2. {
3.     return v->counter;
4. }
5.
6. static inline void atomic_set(atomic_t *v, int i)
7. {
8.     v->counter = i;
9. }
```

单独的读操作或者写操作，在x86下都是原子性的。

```
1. static inline void atomic_add(int i, atomic_t *v)
2. {
3.     asm volatile(LOCK_PREFIX "addl %1,%0"
4.             : "+m" (v->counter)
5.             : "ir" (i));
6. }
7.
8. static inline void atomic_sub(int i, atomic_t *v)
9. {
10.     asm volatile(LOCK_PREFIX "subl %1,%0"
11.             : "+m" (v->counter)
12.             : "ir" (i));
13. }
```

atomic_add和atomic_sub属于读修改写操作，实现时需要加lock前缀。

```
1. static inline int atomic_sub_and_test(int i, atomic_t *v)
2. {
3.     unsigned char c;
4.
5.     asm volatile(LOCK_PREFIX "subl %2,%0; sete %1"
6.             : "+m" (v->counter), "=qm" (c)
7.             : "ir" (i) : "memory");
8.     return c;
9. }
```

atomic_sub_and_test执行完减操作后检查结果是否为0。

```
1. static inline void atomic_inc(atomic_t *v)
2. {
3.     asm volatile(LOCK_PREFIX "incl %0"
```

```
4.              : "+m" (v->counter));
5. }
6.
7. static inline void atomic_dec(atomic_t *v)
8. {
9.     asm volatile(LOCK_PREFIX "decl %0"
10.             : "+m" (v->counter));
11. }
```

atomic_inc和atomic_dec是递增递减操作。

```
1. static inline int atomic_dec_and_test(atomic_t *v)
2. {
3.     unsigned char c;
4.
5.     asm volatile(LOCK_PREFIX "decl %0; sete %1"
6.             : "+m" (v->counter), "=qm" (c)
7.             : : "memory");
8.     return c != 0;
9. }
```

atomic_dec_and_test在递减后检查结果是否为0。

```
1. static inline int atomic_inc_and_test(atomic_t *v)
2. {
3.     unsigned char c;
4.
5.     asm volatile(LOCK_PREFIX "incl %0; sete %1"
6.             : "+m" (v->counter), "=qm" (c)
7.             : : "memory");
8.     return c != 0;
9. }
```

atomic_inc_and_test在递增后检查结果是否为0。

```
1. static inline int atomic_add_negative(int i, atomic_t *v)
2. {
3.     unsigned char c;
4.
5.     asm volatile(LOCK_PREFIX "addl %2,%0; sets %1"
6.             : "+m" (v->counter), "=qm" (c)
7.             : "ir" (i) : "memory");
8.     return c;
9. }
```

atomic_add_negative在加操作后检查结果是否为负数。

```
1. static inline int atomic_add_return(int i, atomic_t *v)
2. {
3.     int __i;
4. #ifdef CONFIG_M386
5.     unsigned long flags;
6.     if (unlikely(boot_cpu_data.x86 <= 3))
7.         goto no_xadd;
8. #endif
9.     /* Modern 486+ processor */
10.    __i = i;
11.    asm volatile(LOCK_PREFIX "xaddl %0, %1"
12.            : "+r" (i), "+m" (v->counter)
13.            : : "memory");
14.    return i + __i;
15.
16. #ifdef CONFIG_M386
17. no_xadd: /* Legacy 386 processor */
18.    local_irq_save(flags);
19.    __i = atomic_read(v);
20.    atomic_set(v, i + __i);
21.    local_irq_restore(flags);
22.    return i + __i;
23. #endif
24. }
```

atomic_add_return 不仅执行加操作，而且把相加的结果返回。它是通过xadd这一指令实现的。

```
1. static inline int atomic_sub_return(int i, atomic_t *v)
2. {
3.     return atomic_add_return(-i, v);
4. }
```

atomic_sub_return 不仅执行减操作，而且把相减的结果返回。它是通过atomic_add_return实现的。

```
1. static inline int atomic_cmpxchg(atomic_t *v, int old, int new)
2. {
3.     return cmpxchg(&v->counter, old, new);
4. }
5.
6. #define cmpxchg(ptr, o, n)                    \
7.     ((__typeof__(*(ptr)))__cmpxchg((ptr), (unsigned long)(o),   \
8.                     (unsigned long)(n),      \
9.                     sizeof(*(ptr))))
10.
11. static inline unsigned long __cmpxchg(volatile void *ptr, unsigned long old,
12.                     unsigned long new, int size)
13. {
14.     unsigned long prev;
15.     switch (size) {
16.     case 1:
17.         asm volatile(LOCK_PREFIX "cmpxchgb %b1,%2"
18.                 : "=a"(prev)
19.                 : "q"(new), "m"(*__xg(ptr)), "0"(old)
20.                 : "memory");
21.         return prev;
22.     case 2:
23.         asm volatile(LOCK_PREFIX "cmpxchgw %w1,%2"
24.                 : "=a"(prev)
25.                 : "r"(new), "m"(*__xg(ptr)), "0"(old)
26.                 : "memory");
27.         return prev;
28.     case 4:
29.         asm volatile(LOCK_PREFIX "cmpxchgl %k1,%2"
30.                 : "=a"(prev)
31.                 : "r"(new), "m"(*__xg(ptr)), "0"(old)
32.                 : "memory");
33.         return prev;
34.     case 8:
35.         asm volatile(LOCK_PREFIX "cmpxchgq %1,%2"
36.                 : "=a"(prev)
37.                 : "r"(new), "m"(*__xg(ptr)), "0"(old)
38.                 : "memory");
39.         return prev;
40.     }
41.     return old;
42. }
```

atomic_cmpxchg是由cmpxchg指令完成的。它把旧值同atomic_t类型的值相比较，如果相同，就把新值存入atomic_t类型的值中，返回atomic_t类型变量中原有的值。

```
1. static inline int atomic_xchg(atomic_t *v, int new)
2. {
3.     return xchg(&v->counter, new);
4. }
5.
6. #define xchg(ptr, v)                        \
7.     ((__typeof__(*(ptr)))__xchg((unsigned long)(v), (ptr), sizeof(*(ptr))))
8.
9. static inline unsigned long __xchg(unsigned long x, volatile void *ptr,
10.                 int size)
11. {
12.     switch (size) {
13.     case 1:
14.         asm volatile("xchgb %b0,%1"
15.                 : "=q" (x)
16.                 : "m" (*__xg(ptr)), "0" (x)
17.                 : "memory");
18.         break;
```

```
19.     case 2:
20.         asm volatile("xchgw %w0,%1"
21.                 : "=r" (x)
22.                 : "m" (*__xg(ptr)), "0" (x)
23.                 : "memory");
24.         break;
25.     case 4:
26.         asm volatile("xchgl %k0,%1"
27.                 : "=r" (x)
28.                 : "m" (*__xg(ptr)), "0" (x)
29.                 : "memory");
30.         break;
31.     case 8:
32.         asm volatile("xchgq %0,%1"
33.                 : "=r" (x)
34.                 : "m" (*__xg(ptr)), "0" (x)
35.                 : "memory");
36.         break;
37.     }
38.     return x;
39. }
```

atomic_xchg则是将新值存入atomic_t类型的变量，并将变量的旧值返回。它使用xchg指令实现。

```
1. /**
2.  * atomic_add_unless - add unless the number is already a given value
3.  * @v: pointer of type atomic_t
4.  * @a: the amount to add to v...
5.  * @u: ...unless v is equal to u.
6.  *
7.  * Atomically adds @a to @v, so long as @v was not already @u.
8.  * Returns non-zero if @v was not @u, and zero otherwise.
9.  */
10. static inline int atomic_add_unless(atomic_t *v, int a, int u)
11. {
12.     int c, old;
13.     c = atomic_read(v);
14.     for (;;) {
15.         if (unlikely(c == (u)))
16.             break;
17.         old = atomic_cmpxchg((v), c, c + (a));
18.         if (likely(old == c))
19.             break;
20.         c = old;
21.     }
22.     return c != (u);
23. }
```

atomic_add_unless的功能比较特殊。它检查v是否等于u，如果不是则把v的值加上a，返回值表示相加前v是否等于u。因为在atomic_read和atomic_cmpxchg中间可能有其它的写操作，所以要循环检查自己的值是否被写进去。

```
1. #define atomic_inc_not_zero(v) atomic_add_unless((v), 1, 0)
2.
3. #define atomic_inc_return(v)  (atomic_add_return(1, v))
4. #define atomic_dec_return(v)  (atomic_sub_return(1, v))
```

atomic_inc_not_zero在v值不是0时加1。

atomic_inc_return对v值加1，并返回相加结果。

atomic_dec_return对v值减1，并返回相减结果。

```
1. #define atomic_clear_mask(mask, addr)               \
2.     asm volatile(LOCK_PREFIX "andl %0,%1"          \
3.             : : "r" (~(mask)), "m" (*(addr)) : "memory")
```

atomic_clear_mask清除变量某些位。

```
1. #define atomic_set_mask(mask, addr)             \
2.     asm volatile(LOCK_PREFIX "orl %0,%1"               \
3.             : : "r" (mask), "m" (*(addr)) : "memory")
```

atomic_set_mask将变量的某些位置位。

```
1. /* Atomic operations are already serializing on x86 */
2. #define smp_mb__before_atomic_dec() barrier()
3. #define smp_mb__after_atomic_dec()  barrier()
4. #define smp_mb__before_atomic_inc() barrier()
5. #define smp_mb__after_atomic_inc()  barrier()
```

因为x86的atomic操作大多使用原子指令或者带lock前缀的指令。带lock前缀的指令执行前会完成之前的读写操作，对于原子操作来说不会受之前对同一位置的读写操作，所以这里只是用空操作barrier()代替。barrier()的作用相当于告诉编译器这里有一个内存屏障，放弃在寄存器中的暂存值，重新从内存中读入。

本节的atomic_t类型操作是最基础的，为了介绍下面的内容，必须先介绍它。如果可以使用atomic_t类型代替临界区操作，也可以加快不少速度。

上一页 1 2 3 4 5 6 7 8 9 下一页 3

【内容导航】

第1页：连通世界的list
第2页：原子性操作atomic_t
第3页：记录生命周期的kref
第4页：更强的链表klist
第5页：设备驱动模型的基石kobject
第6页：设备驱动模型之device
第7页：设备驱动模型之driver
第8页：设备驱动模型之bus
第9页：设备驱动模型之device-driver

Linux内核的学习方法
Linux根目录下主要目录功能说明及常用分区方案

相关资讯 Linux内核

Linux内核Git源码树中的代码已达 (今 20:48)
Linux 5.4.7 / 4.19.92 / 4.14.161 (01月01日)
Linux内核将用Rust编程语言编写？ (09/03/2019 12:06:17)
Linux内核将很快默认情况启用"- (05/11/2019 13:43:07)
Linux内核正在努力实现快速高效的I (02/15/2019 14:51:33)
Linux内核的冷热缓存 (01/27/2019 19:10:52)

本文评论 查看全部评论 (5)

表情： 姓名： 匿名 匿名 字数 0

同意评论声明 请登录

评论声明

- 尊重网上道德，遵守中华人民共和国的各项有关法律法规
- 承担一切因您的行为而直接或间接导致的民事或刑事法律责任
- 本站管理人员有权保留或删除其管辖留言中的任意内容
- 本站有权在网站内转载或引用您的评论
- 参与本评论即表明您已经阅读并接受上述条款

AlexXue 发表于 2018/6/22 11:29:47 第 5 楼

好吧，没有问题，当我没说

回复 支持 (0) 反对 (0)

AlexXue 发表于 2018/6/22 9:05:51 第 4 楼

作者，我认为你的__list_add函数有问题。麻烦画图分析一下。

回复 支持 (0) 反对 (0)

AlexXue 发表于 2018/6/20 10:07:20 第 3 楼

作者你好关于__list_add这个函数，我画图分析之后发现存在问题，烦请您贴图分析一下，感谢。

回复 支持 (0) 反对 (0)

lzxname 发表于 2014/11/4 9:48:24 第 2 楼

好东西。留一笔。

回复 支持 (14) 反对 (11)

lzxname 发表于 2014/11/4 9:43:13 第 1 楼

好东西啊啊。。。

回复 支持 (8) 反对 (13)