XV6 锁 阅读报告

本次代码阅读主要对 XV6 中的锁的机制进行调研。主要涉及的代码有：spinlock.h，spinlock.c，x86.h 中的一些辅助函数和 proc.h 中的 CPU 类。下面我将逐个剖析，以阐释 XV6 的进程调度机制。

一、XV6 中的锁机制

在分析完后，我们来分析一下 XV6 中的锁机制。XV6 中的锁主要由 spinlock.h 和 spinlock.c 两份文件实现，下面让我们来逐一分析一下。

1. XV6 中的互斥锁结构——spinlock

XV6 中的互斥锁结构由 spinlock 结构体实现，这个结构体定义于 spinlock.h 中，内容如下：

```
2  struct spinlock {
3    uint locked;
4
5    // For debugging:
6    char *name;
7    struct cpu *cpu;
8    uint pcs[10];
9
10 };
```

locked 这个无符号整数用于标注当前的锁是否处于被锁住的状态（是否有进程获取了这个锁）。它的初始值为 0，表示未被占用。当某个进程（cpu）占用它的时候，会把这个值设置成 1.

name 标识锁的名称，在 debug 的时候使用。

cpu 是一个 cpu 结构的指针，指向持有这个锁的 CPU，以实现 CPU 之间的同步。

pcs 数组中储存了占用这个锁的进程的调用栈信息。

可以说，在整个自旋锁结构中，最关键的是 locked 这一变量，它是锁的互斥控制标记。

2. XV6 中的互斥锁方法

XV6 中的锁的结构相对比较简单，下面我们继续来分析一下 XV6 中为 spinlock 提供的函数方法。这些函数主要被声明在 spinlock.c 中，内容如下：

（1）initlock

这个函数是自旋锁结构的初始化函数，每个自旋锁被声明后都需要调用它。函数的主要功能就是初始化自旋锁的 name、locked、cpu 成员变量。locked 被初始化为 0，代表初始没有进程占用这个锁。

```
12  void
13  initlock(struct spinlock *lk, char *name)
14  {
15    lk->name = name;
16    lk->locked = 0;
17    lk->cpu = 0;
18  }
```

（2）acquire、pushcli、holding、xchg、getcallerpcs

获取锁的函数。函数会在屏蔽中断后，**原子化**地检查当前所的占用情况。如果当前锁的 locked 仍然为 1，那么进行死循环来等待锁被释放（即 locked 重新变为 0）。锁被释放后，对应锁的 cpu 标号被更新，且当前自旋锁还会记录当前进程的栈信息。

```
25  acquire(struct spinlock *lk)
26  {
27    pushcli(); // disable interrupts to avoid deadlock.
28    if(holding(lk))
29      panic("acquire");
30
31    // The xchg is atomic.
32    // It also serializes, so that reads after acquire are not
33    // reordered before it.
34    while(xchg(&lk->locked, 1) != 0)
35      ;
36
37    // Record info about lock acquisition for debugging.
38    lk->cpu = cpu;
39    getcallerpcs(&lk, lk->pcs);
40  }
```

【Step1】

在等待获取锁之前，函数首先会调用 pushcli 函数屏蔽所有中断。XV6 中的锁是自旋锁，根据课堂讲授的内容，我们可以知道，自旋锁非常容易发生死锁现象。屏蔽中断就是为了**避免在抢占式内核中，当前进程占用了锁以后，因为中断进入到内核态，再次想要获取锁时出现的死锁现象**。

XV6 使用 pushcli 函数实现这一目标。pushcli 函数**首先获得标志寄存器 EFLAGS 中的值，然后调用 cli 函数封装的 cli 汇编指令（clear interrupt）来关闭中断**。关闭完成后，如果这是当前 CPU（在多次累加累减后）第一次调用 pushcli，那么函数会更新 CPU 中的 intena 变量（取出 EFLAGS 寄存器原始值中负责记录中断情况的位，用 FL_IF 宏来标记），以记录在（多次屏蔽-复原抵消后的）第一次屏蔽中断前，当前 CPU 是否恢复了中断响应。除此之外，**每次调用完这个函数后，无论是否执行 if 内的语句，当前 CPU 的 ncli 变量都会被递增，以记录这个函数的调用次数（也就是申请获取锁的个数）**。之所以如此处理，是因为如果代码获得了多个锁，那么只有每个锁都释放以后，才能恢复中断，因此这需要我们记录锁的个数。

```
94  static inline uint
95  readeflags(void)
96  {
97    uint eflags;
98    asm volatile("pushfl; popl %0" : "=r" (eflags));
99    return eflags;
100 }
```

```
108 static inline void
109 cli(void)
110 {
111   asm volatile("cli");
112 }
```

```
96  void
97  pushcli(void)
98  {
99    int eflags;
100
101   eflags = readeflags();
102   cli();
103   if(cpu->ncli++ == 0)
104     cpu->intena = eflags & FL_IF;
105 }
```

【Step2】

现在，中断已经被屏蔽，在正式请求获取锁之前，我们还需要对锁的持有情况进行检查。如果当前 cpu 获取了锁（locked 为 1，锁的 cpu 值等于当前 cpu 标号），那么会调用 panic 函数报错。因为这时一个 cpu 两次申请获取同一个锁，这会造成死锁问题。

```
85  int
86  holding(struct spinlock *lock)
87  {
88    return lock->locked && lock->cpu == cpu;
89  }
```

【Step3】

现在所有准备工作均已完成，我们可以开始等待获取当前锁。这一步骤的核心问题是如何实现**等待的原子化**。试想，如果我们用下面的循环来等待锁的释放，当锁被释放时，如果在一个进程执行完 if 语句后，调度到的新进程，新的进程也在执行完 if 语句后调度回原进程，这样就有两个进程同时获取了这个锁，这就违反了锁获取的互斥性。

```
for(;;) {
    if(!lk->locked) {
        lk->locked = 1;
        break;
    }
}
```

因此，我们需要原子化地执行等待循环，以避免这种竞争关系的发生。

XV6 的处理办法是采用 x86（386）架构中的 xchg 指令。xchg 函数中使用 GCC 的内联

汇编特性（asm）封装了 xchgl 的汇编代码。**volatile 用于避免 gcc 进行优化**；第一个冒号后的**`"+m" (*addr), "=a" (result)`表示汇编指令的两个输出值**，**`newval`是汇编指令的输入值**；**lock** 是一个指令前缀，**保证了指令对总线和缓存的独占权**（也就是这条指令的执行过程中不会有其他 CPU 或同 CPU 内的指令访问缓存和内存）；**xchg** 的作用是**交换两个位置的 4 字节值**，但两个位置不能都是内存地址。除此之外，由于这个函数是 inline 的，所以每次调用时它会被直接嵌入调入段的代码。

```
120  static inline uint
121  xchg(volatile uint *addr, uint newval)
122  {
123    uint result;
124
125    // The + in "+m" denotes a read-modify-write operand.
126    asm volatile("lock; xchgl %0, %1" :
127                 "+m" (*addr), "=a" (result) :
128                 "1" (newval) :
129                 "cc");
130    return result;
131  }
```

因此，回到 acquire 函数中，**这个函数使用一条指令便交换了 locked 和 1（或者更准确的说，是存放 1 的寄存器）的值，并且可以同时通过返回值（%eax 寄存器）来判断 locked 是否为 0**。因此，这个内联汇编函数便可以用一句代码（一条指令）原子地实现上述循环体的判断 locked 的原始值+赋值的功能。

注意到这里还有一个指令顺序的细节，即 pushcli 必须在 xchg 之前完成。否则，我们可能会在获得锁的时候仍然接受中断，这样又可能造成中断处理函数申请这个已被挂起的进程占据的锁的死锁局面。

【Step4】

上面我们已经原子化地实现了锁的获取，最后，我们还需要更新和记录一些必需的信息。一方面，我们要更新占用这把锁的 CPU id，另一方面，我们要在锁的 pcs 数组中记录当前进程的栈的部分信息。第二方面的内容由 getcallerpcs 这一函数实现：

```
67  void
68  getcallerpcs(void *v, uint pcs[])
69  {
70    uint *ebp;
71    int i;
72
73    ebp = (uint*)v - 2;
74    for(i = 0; i < 10; i++){
75      if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
76        break;
77      pcs[i] = ebp[1];     // saved %eip
78      ebp = (uint*)ebp[0]; // saved %ebp
79    }
80    for(; i < 10; i++)
81      pcs[i] = 0;
82  }
```

这个函数通过追踪%ebp 寄存器（帧指针）的值，来逐级追踪当前进程中先前调用的函数的帧位置的信息（每次调用后的返回值），并把它们依次（由最近的调用到最远的调用）

依次储存在 pcs 的数组中，最多储存 10 次调用的信息。这些信息可以帮我们追踪过程的执行流程。

（3）release、popcli

XV6 锁操作除了占用锁啊 acquire 函数，对应地还有释放锁的 release 函数。这个函数的流程与 acquire 在思路上很接近。

```
43  void
44  release(struct spinlock *lk)
45  {
46    if(!holding(lk))
47      panic("release");
48
49    lk->pcs[0] = 0;
50    lk->cpu = 0;
51
52    // The xchg serializes, so that reads before release are
53    // not reordered after it.  The 1996 PentiumPro manual (Volume 3,
54    // 7.2) says reads can be carried out speculatively and in
55    // any order, which implies we need to serialize here.
56    // But the 2007 Intel 64 Architecture Memory Ordering White
57    // Paper says that Intel 64 and IA-32 will not move a load
58    // after a store. So lock->locked = 0 would work here.
59    // The xchg being asm volatile ensures gcc emits it after
60    // the above assignments (and after the critical section).
61    xchg(&lk->locked, 0);
62
63    popcli();
64  }
```

【Step1】

函数首先检查当前 cpu 是否真的占据了这个锁，如果没占据，调用 panic 函数以提示异常。

【Step2】

函数把锁中保存的调用栈信息和 CPU 标号清空，以便下一个占据锁的进程进行修改。

【Step3】

函数释放锁，把 locked 变量重置为 0。这里使用 xchg 的原因主要是为了保证指令的序列化，因为某些处理器（1996 PentiumPro）可能会随机安排读取的位置。如果使用复制指令代替 xchg，当乱序处理器把赋值语句放到 popcli 的后面，那么可能在锁释放之前就允许中断了，acquire 会被打断，因此我们需要用 xchg 来序列化指令的执行。但 IA32 和 2007 Intel 64 等处理器不会把加载指令移到存储指令之后，这种情况下直接使用赋值指令也是可行的。

【Step4】

最后，release 会调用 popcli 函数恢复对中断的响应。

```
107 void
108 popcli(void)
109 {
110   if(readeflags()&FL_IF)
111     panic("popcli - interruptible");
112   if(--cpu->ncli < 0)
113     panic("popcli");
114   if(cpu->ncli == 0 && cpu->intena)
115     sti();
116 }
```

这个函数首先检查在恢复之前 CPU 是否已经是可响应中断的了，如果是，则调用 panic 函数并退出。接下来函数会检查之前是否还有调用过的 pushcli 函数未处理，如果没有，也会调用 panic 函数退出。最后，当先前抵消的是最后一个 pushcli 调用，而且 pushcli 调用之前确实是中断可相应的，那么函数调用 sti 函数恢复对中断的响应。sti 函数使用内联汇编，调用 sti 指令，以恢复 EFLAGS 中的中断响应位的值。

```
114  static inline void
115  sti(void)
116  {
117    asm volatile("sti");
118  }
```

可以看到，这个函数与 pushcli 配套使用，记录了当前 CPU 上获取的锁的个数，同时又不会因为释放多把锁中的某一个而提前恢复中断响应，而且还可以利用 CPU 的 intena 字段来恢复至获取锁之前的中断响应状态。

与 acquire 中相似，这里需要先释放锁再 popcli，这个顺序不能改变，否则还是会出现前面讨论过的死锁问题。

三、锁的几个应用

上面我们分析了 XV6 中锁的机制，那么接下来我们来看几个 XV6 中锁的应用实例。

1.IDE 驱动中的锁

XV6 中的磁盘驱动会维护一个未完成磁盘请求的链表，每个进程会对其进行写操作来添加请求因此，处理器可能会并发地向链表中加入新的请求。为了保护链表和驱动中的其他不变量，iderw 函数会申请获取 idelock 锁，并在执行完添加请求操作后释放锁。如果不加入锁机制来保护，很显然，对于多处理器的情况，有可能两个 CPU 上的不同进程同时向链表中的相同位置写入请求，这样一个进程的请求就被覆盖掉了。因此，这个锁需要在链表的循环前就获取，否则便可能发生竞争。

```
125  void
126  iderw(struct buf *b)
127  {
128    struct buf **pp;
129
130    if(!(b->flags & B_BUSY))
131      panic("iderw: buf not busy");
132    if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
133      panic("iderw: nothing to do");
134    if(b->dev != 0 && !havedisk1)
135      panic("iderw: ide disk 1 not present");
136
137    acquire(&idelock);  //DOC:acquire-lock
138
139    // Append b to idequeue.
140    b->qnext = 0;
141    for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  //DOC:insert-queue
142      ;
143    *pp = b;
144
145    // Start disk if necessary.
146    if(idequeue == b)
147      idestart(b);
148
149    // Wait for request to finish.
150    while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
151      sleep(b, &idelock);
152    }
153
154    release(&idelock);
155  }
```

2.中断处理程序中的锁

XV6 用锁来防止中断处理程序与另一个 CPU 上正在运行的进程使用同一个数据。比如，时钟中断会增加 ticks，但是 sys_sleep 的系统调用也要使用该变量。因此，XV6 中使用 tickslock 为这个变量实现同步。

```
50    case T_IRQ0 + IRQ_TIMER:
51      if(cpu->id == 0){
52        acquire(&tickslock);
53        ticks++;
54        wakeup(&ticks);
55        release(&tickslock);
56      }
```

```
66    acquire(&tickslock);
67    ticks0 = ticks;
68    while(ticks - ticks0 < n){
69      if(proc->killed){
70        release(&tickslock);
71        return -1;
72      }
73      sleep(&ticks, &tickslock);
74    }
75    release(&tickslock);
```

除此之外，即便是单处理器，也可能有死锁的问题。比如，如果 iderw 函数已经持有了 idelock，然后中断发生，中断处理程序运行 ideintr 函数，这个中断处理函数也需要获取 idelock 锁，但此时已经被中断前的进程获取了，这样，在这个 CPU 上就出现了死锁。

```
91  void
92  ideintr(void)
93  {
94    struct buf *b;
95
96    // First queued buffer is the active request.
97    acquire(&idelock);
98    if((b = idequeue) == 0){
99      release(&idelock);
100     // cprintf("spurious IDE interrupt\n");
101     return;
102   }
103   idequeue = b->qnext;
104
105   // Read data if needed.
106   if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
107     insl(0x1f0, b->data, 512/4);
108
109   // Wake process waiting for this buf.
110   b->flags |= B_VALID;
111   b->flags &= ~B_DIRTY;
112   wakeup(b);
113
114   // Start disk on next buf in queue.
115   if(idequeue != 0)
116     idestart(idequeue);
117
118   release(&idelock);
119 }
```

为了避免这种情况，当中断处理会使用某个锁时，处理器应该避免中断发生时持有这把锁。XV6 中的处理办法是：**不允许中断时持有任何一把锁**。因此，acquire 和 release 会分别调用 pushcli 和 popcli，既修改了对中断的响应状态，又起到了计数的作用。

四、小结

本次代码阅读中，我调研了 XV6 中锁的机制。XV6 中实现了自旋锁来解决互斥和同步问题。XV6 中使用了自旋锁来进行同步的保护。每次在访问共享区域时，进程（函数）会调用 acquire 函数试图获取锁。acquire 首先屏蔽中断，如果当前锁被占用，那么就进行忙等待; 否则用 xchg 的原子化操作来占用这个锁。而执行完所有操作后, release 函数会释放锁。为了避免乱序执行带来的不正确行为，函数使用 xchg 操作来释放锁变量，这样便可以序列化地执行中断恢复的操作了。

在系统中很多地方，都会用到锁来解决竞争问题。本次调研中我主要研究了磁盘请求和中断发生时的竞争关系以及锁机制的解决方法。除了这两个例子以外，在前面已经调研过的进程机制中，sleep、wake 等函数也使用了锁机制来同步进程表。

虽然 XV6 的机制解决了同步和竞争的问题，但由于 XV6 使用的是自旋锁，因此忙等待会造成 CPU 低效率空转的问题，产生了资源的浪费，这是 XV6 锁机制中需要改进的一个不足。