

lab3 report

马逸君 17300180070

TODO#1 阅读spin_lock文件

阅读spin_lock文件，分析自旋锁的运作原理并回答：xchg(), pushcli(), popcli()分别做了什么，为什么执行这些操作？

自旋锁的运作原理：如果有进程正占有(hold)一个自旋锁，则其他请求该锁的进程都必须无限循环（自旋）等待该锁被释放，直到获得锁退出循环。

源码分析：（spinlock.c）

```
1 void
2 acquire(struct spinlock *lk)
3 {
4     pushcli(); // 关闭中断，使得进程请求锁的操作都是原子化的，避免死锁
5     if(holding(lk))
6         panic("acquire"); // 如果在当前CPU已经获得锁的情况下又再次请求锁，报错退出
7
8     // “自旋”
9     while(xchg(&lk->locked, 1) != 0)
10         ; // 注意这里的xchg函数是原子化的
11
12     __sync_synchronize(); // 告知C编译器和CPU不要将数据存储到该语句对应的地址以后，从而确保临界区的内存引用都是在获得锁之后才发生的
13
14     // 记录本次请求的信息，供调试使用
15     lk->cpu = mycpu();
16     getcallerpcs(&lk, lk->pcs);
17 }
```

xchg()函数的功能及目的：将一个内存地址的原值换成用户给定的新值，并返回原值。这个过程是原子化(atomic)的。该函数给予了各进程一种不会被其他进程打断（否则可能死锁）的试图获得锁的操作途径——xchg(lock, 1)（lock为进程试图请求的锁），如果换出的值为1说明该锁已被其他进程锁上，换出的值为0说明进程成功获得了该锁。

xchg()函数的实现是通过将用户传入的参数拿去调用汇编指令xchg来实现的；因为xchg是一条单指令，所以不可能被打断，这样就实现了原子性。

源码分析：

```

1 static inline uint
2 xchg(volatile uint *addr, uint newval) // 该操作是原子化的
3 {
4     uint result;
5
6     // "+m"中的加号表示读取-修改-写入操作数。
7     asm volatile("lock; xchgl %0, %1" : // 调用引号内的汇编语句
8                 "+m" (*addr), "=a" (result) : // 指定输出: 将上面的占位符%0替换成
内存地址*addr, 将%1替换成变量result且指定寄存器eax
9                 "1" (newval) : // 输入为%1, 也就是上面已经指定过的第二个寄存器"a"
10                "cc"); // 该行表示该指令会更改条件寄存器(cc: condition code), 即指
令执行可能会导致flags寄存器中的值变得无效
11     return result;
12 }

```

pushcli()/popcli()函数的功能及目的：这对函数可以视为带有计数机制的cli/sti（这对汇编指令的作用是关闭/打开中断）。亦即，它们的作用类似cli/sti但是成对出现，两个pushcli()的影响必须调用两次popcli()才能消除；另外，如果中断本来就是关闭的，它们也会保持中断的关闭。

pushcli()/popcli()的实现方法：维护mycpu中的一个计数器nccli来统计mycpu当前被嵌套pushcli的层数，维护mycpu中的一个变量intena来表示截至当前嵌套中的最高层pushcli生效前中断是否打开，再在if语句中判断各种可能的情况。

源码分析：

```

1 void
2 pushcli(void)
3 {
4     int eflags;
5
6     eflags = readeflags(); // 暂存指令执行前EFLAGS寄存器的值
7     cli(); // 执行一次cli指令（因为如果中断已经关闭，执行cli指令也就自然地无作用，所以执
行之前不需要判断中断是否打开）
8     if(mycpu()->nccli == 0) // 如果指令执行前mycpu被嵌套pushcli的层数为0层
9         mycpu()->intena = eflags & FL_IF; // 则需要更新intena（intena是mycpu中暂存
调用pushcli前中断是否打开的缓存变量）
10    mycpu()->nccli += 1; // 嵌套pushcli的层数增加1
11 }

```

```

1 void
2 popcli(void)
3 {
4     if(readeflags() & FL_IF) // 如果CPU当前是打开了中断的
5         panic("popcli - interruptible"); // 则不应该调用popcli指令，报错并终止xv6运行
6     if(--mycpu()->nccli < 0) // 给mycpu被嵌套pushcli的层数减去1，若减后小于0
7         panic("popcli"); // 说明代码中有错误，报错并终止xv6运行
8     if(mycpu()->nccli == 0 && mycpu()->intena) // 若减后为0（说明自从上次统计intena
以来调用过的所有pushcli都被撤销），而且intena中的状态是中断开启
9         sti(); // 执行一次sti指令，打开中断
10 }

```

值得一提的是上面反复出现的eflags & FL_IF这个表达式。FL_IF是一个常量，值为0x00000200，写成二进制的形式就是只有从低位起第9位为1，其他位均为0；因而，eflags & FL_IF的运算结果就是将EFLAGS寄存器值的第9位提取出来，其他二进制位均置0。所以就可以用eflags & FL_IF是否为0来表示是否开启了中断。

TODO#2 xv6系统中的互斥锁

1) 请理解以下代码片段，说明此代码的含义：

```
1 struct spinlock test_lk;
2 initlock(&test_lk, "Test Lock in xv6");
3 acquire(&test_lk);
4 acquire(&test_lk);
```

该代码段的功能是测试当有进程试图重复申请同一个锁的时候系统的反应。

源码分析：

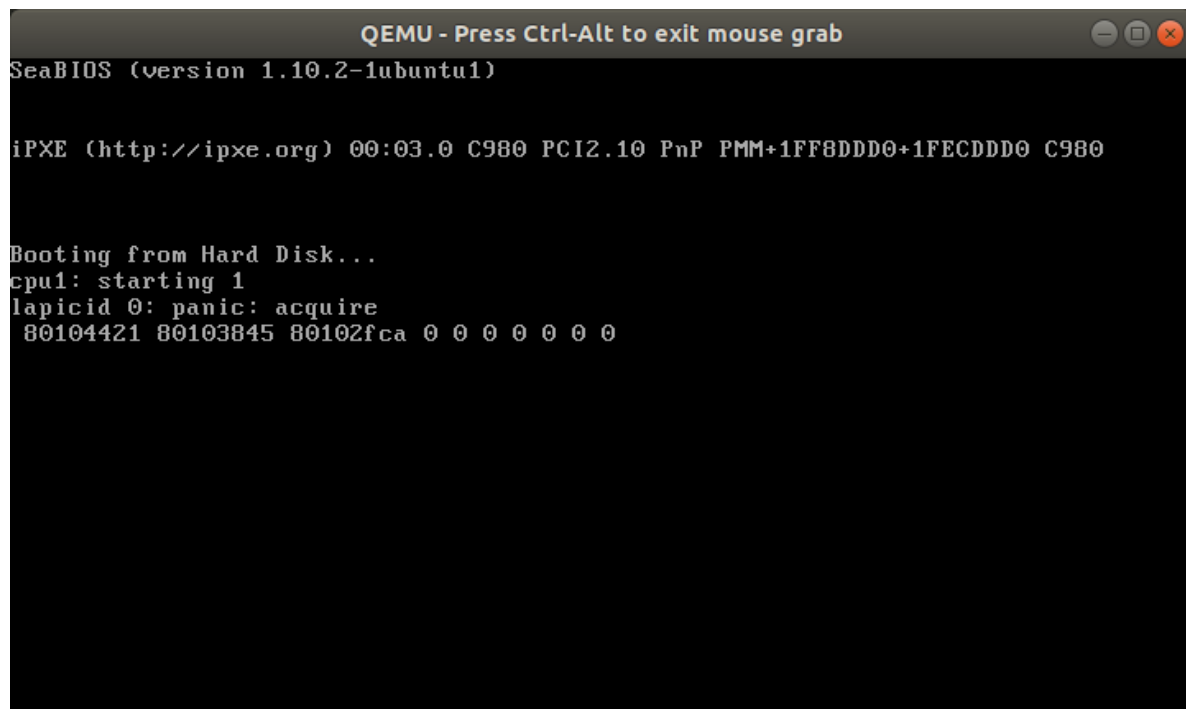
```
1 struct spinlock test_lk; // 声明一个自旋锁test_lk
2 initlock(&test_lk, "Test Lock in xv6"); // 初始化test_lk, 其name赋值为"Test
   Lock in xv6"
3 acquire(&test_lk); // 立即请求test_lk
4 acquire(&test_lk); // 又立即再次请求test_lk
```

2) 尝试在xv6系统中运行上述代码片段，同时请在屏幕上输出运行结果写入实验报告。

只有内核代码有权调用initlock()和acquire()。故打开proc.c，将代码段加到xv6启动时必调用的userinit()函数的最前面：

```
1 void
2 userinit(void)
3 {
4     struct spinlock test_lk;
5     initlock(&test_lk, "Test Lock in xv6");
6     acquire(&test_lk);
7     acquire(&test_lk);
8     ...
9 }
```

并在qemu中运行，结果如下：



```
QEMU - Press Ctrl-Alt to exit mouse grab
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF8DDD0+1FECDDD0 C980

Booting from Hard Disk...
cpu1: starting 1
lapicid 0: panic: acquire
80104421 80103845 80102fca 0 0 0 0 0 0
```

-----以上为答案，以下为本题评注-----

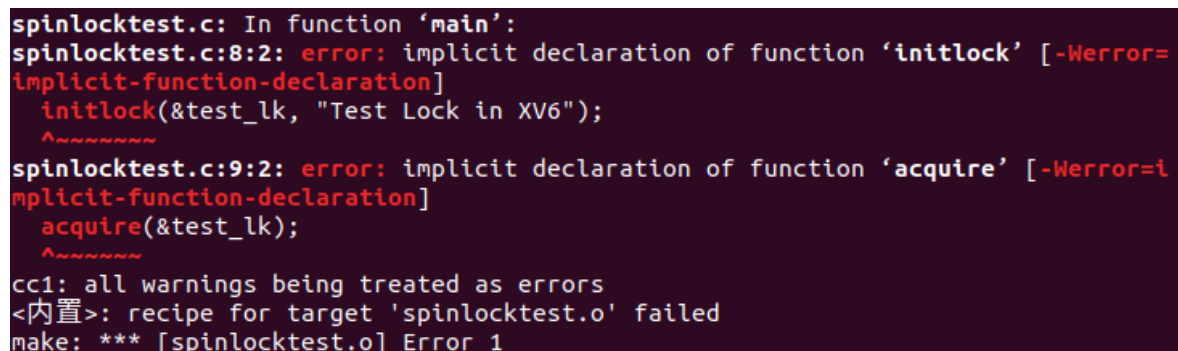
值得一提的是，一开始我试图采用与上一次lab的选做题一样的方法，即通过添加用户代码来测试本题的性能。具体来说，在xv6目录下新建源码文件spinlocktest.c如下：

```
1  #include "types.h"
2  #include "user.h"
3  #include "spinlock.h"
4
5  int main()
6  {
7      struct spinlock test_lk;
8      initlock(&test_lk, "Test Lock in xv6");
9      acquire(&test_lk);
10     acquire(&test_lk);
11     exit();
12 }
```

并修改Makefile，在UPROGS字段下添加spinlocktest：

```
1  UPROGS=\
2      _cat\
3      _echo\
4      _forktest\
5      _grep\
6      _init\
7      _kill\
8      _ln\
9      _ls\
10     _mkdir\
11     _rm\
12     _sh\
13     _stressfs\
14     _usertests\
15     _wc\
16     _zombie\
17     _getfpid\
18     _spinlocktest\
```

最后在qemu中无法编译，如下图所示。



```
spinlocktest.c: In function 'main':
spinlocktest.c:8:2: error: implicit declaration of function 'initlock' [-Werror=implicit-function-declaration]
    initlock(&test_lk, "Test Lock in xv6");
    ^~~~~~
spinlocktest.c:9:2: error: implicit declaration of function 'acquire' [-Werror=implicit-function-declaration]
    acquire(&test_lk);
    ^~~~~~
cc1: all warnings being treated as errors
<内置>: recipe for target 'spinlocktest.o' failed
make: *** [spinlocktest.o] Error 1
```

这是因为acquire()属于系统函数而不属于用户函数。

读者可能会问，如果强行在用户代码里包含内核头文件会如何呢？本人尝试在spinlocktest.c中又添加了包含"defs.h"，下面是运行结果：

```
mayijun@ubuntu: ~/xv6
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

~~~~~
In file included from spinlocktest.c:2:0:
user.h:25:5: note: previous declaration of 'sleep' was here
int sleep(int);
~~~~~
In file included from spinlocktest.c:4:0:
defs.h:144:17: error: conflicting types for 'memmove'
void* memmove(void*, const void*, uint);
~~~~~
In file included from spinlocktest.c:2:0:
user.h:31:7: note: previous declaration of 'memmove' was here
void *memmove(void*, const void*, int);
~~~~~
In file included from spinlocktest.c:4:0:
defs.h:147:17: error: conflicting types for 'strlen'
int strlen(const char*);
~~~~~
In file included from spinlocktest.c:2:0:
user.h:36:6: note: previous declaration of 'strlen' was here
uint strlen(const char*);
~~~~~
<内置>: recipe for target 'spinlocktest.o' failed
make: *** [spinlocktest.o] Error 1
mayijun@ubuntu:~/xv6$
```

xv6的结构决定了用户态的代码是无法直接接触到acquire()这些系统函数的。用户代码必须包含"user.h"以调用用户态函数，而如果在包含"user.h"的程序里强行包含"defs.h"这样的内核头文件，是无法通过编译的。

这一插曲告诉我们，对内核态和用户态的区别要提高认识。

3) 解释运行代码片段后出现的结果。

当执行到后一个acquire(&test_lk);语句时，因为当前CPU已经占有(hold)该锁，所以在acquire()函数中会执行panic("acquire");操作，报错退出。

panic()函数定义在console.c中，它会输出报错信息并打印出引发panic()的调用的栈帧链，然后开始无限循环（象征着xv6停止运行）。

源码分析：

```
1 void
2 acquire(struct spinlock *lk)
3 {
4     pushcli();
5     if(holding(lk)) // 在执行完第一个acquire(&test_lk);语句后，第二次执行时返回值为真
6         panic("acquire"); // 如果在当前CPU已经获得锁的情况下又再次请求锁，报错退出
7
8     ...
9 }
```

```
1 void
2 panic(char *s) // 报错并退出xv6
3 {
4     int i; // 循环变量
5     uint pcs[10]; // 栈帧链
6
7     cli(); // 关闭中断，从而程序的执行会永远停留在panic()中
8     cons.locking = 0; // cons.locking变量的含义是cprintf()向命令行打印信息时是否需要
                        // 互斥（参见cprintf()）。因为panic()的报错信息需要立即被输出，所以取消互斥。
```

```

9 // use lapiccpunum so that we can call panic from mycpu()
10 cprintf("lapicid %d: panic: ", lapicid()); // lapic:local APIC, 它管理内部中
    断
11 cprintf(s); // 输出报错信息
12 cprintf("\n"); // 换行
13 getcallerpcs(&s, pcs); // 通过getcallerpcs()获得（引发panic()的）调用栈帧链
14 for(i=0; i<10; i++)
15     cprintf(" %p", pcs[i]); // 打印栈帧链
16 panicked = 1; // 挂起其他CPU
17 for(;;)
18     ; // 无限循环、永不返回、也不会被打断（已经关闭了中断），xv6停止正常运行
19 }

```

TODO#3 ide.c中的中断

xv6中提供了acquire和release两个API来获取锁和释放锁。在ide.c的iderw函数中，请修改代码，实现在acquire()（获取锁）后调用一次sti()，在release()（释放锁）前调用一次cli()。然后重新编译并使用QEMU打开系统。你将会看到这会导致内核产生panic。请将运行结果放入实验报告中，同时解释为什么这样修改代码后会导致内核panic。

提示：你可以查看在kernel.asm 栈的运行变化（panic输出的%eip的值）。

运行结果如下图所示：

```

$ mayijun@ubuntu:~/xv6$ make qemu
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=r
aw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start
58
lapicid 1: panic: sched locks
80103c01 80103d72 8010587b 8010566f 80100183 80102b55 80103720 80105672 0 0

```

如图，最下面一行十六进制数即是由panic输出的栈帧链%eip值。

首先分析栈帧链：

第一层（位于调用链起始端的，亦即图中最右边的一个非0的eip值对应的）调用，返回地址为0x80105672，从kernel.asm中得知该地址位于trapret中；类似可知第二层调用的返回地址0x80103720位于forkret，第三层0x80102b55位于initlog，第四层0x80100183位于bread；

我们已知的引起错误的iderw()函数就是由这个bread函数调用的，但是栈帧链中记录的下一个返回地址0x8010566f却不在iderw中，而是在alltraps中；（由此我们可以判断，**在iderw中执行的时候发生了中断**，从而将控制权交给了alltraps，这样才有了接下来的调用过程。）

接下来第六层0x8010587b是trap，第七层0x80103d72是yield，第八层0x80103c01是sched，最终到达panic。

接下来找出触发panic()的原因：

最顶层的调用的返回eip地址为0x80103c01，从kernel.asm中得知，其对应sched()函数中的如下语句：（定义于proc.c）

```

1 if(mycpu()->nccli != 1)
2     panic("sched locks");

```

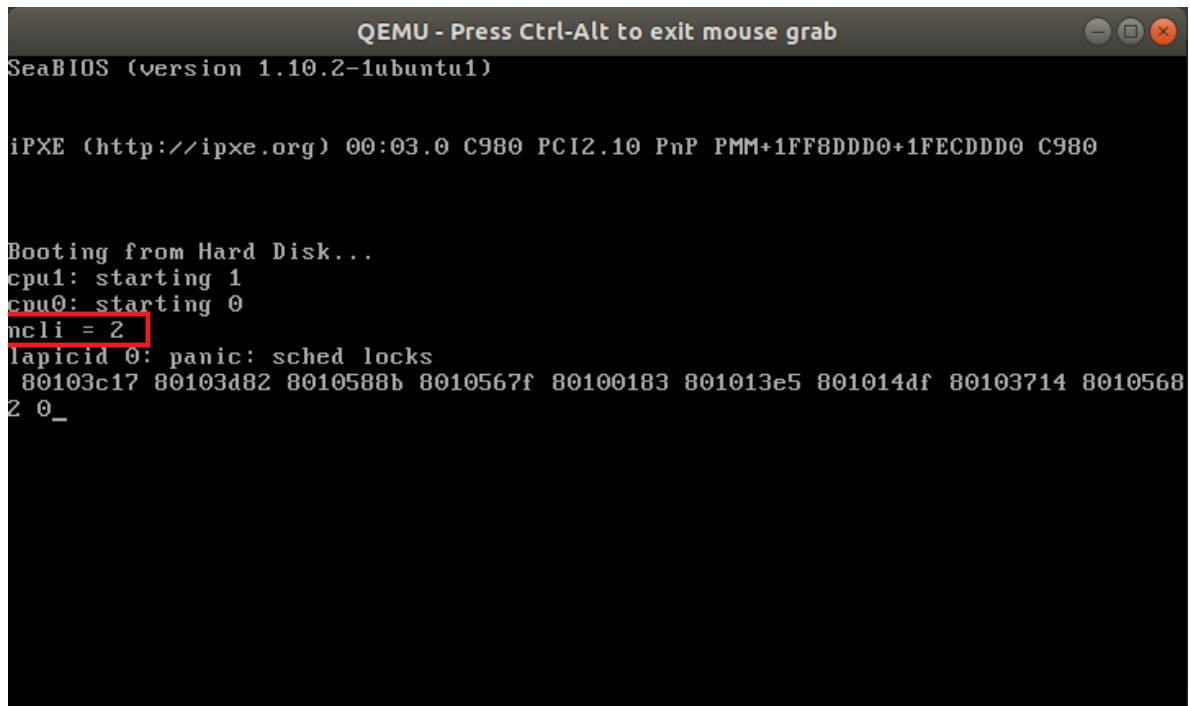
由此可以判断，**中断处理过程中ncli（嵌套调用pushcli的层数）不等于1**，才会发出这样的错误信息。

最后分析pushcli()的调用情况：

在sched()函数中的被触发的panic()语句处增加一个语句，向命令行打印ncli：

```
1 | if(mycpu()->ncli != 1)
2 |     {cprintf("ncli = %d\n", mycpu()->ncli); panic("sched locks");}
```

运行结果如下：



结合对于iderw()之后调用的trap()、yield()、sched()、panic()这几个函数代码的阅读，我们发现，能够增加ncli的只可能是yield()函数中申请加锁的这一行代码：

```
1 | void
2 | yield(void)
3 | {
4 |     acquire(&ptable.lock);
5 |     ...
6 | }
```

由此可以得出结论，**使得ncli等于2的两处调用分别来自：中断处理过程、iderw()。**

另外我们也知道yield()函数的作用是时间片切换，综合以上所有信息，可以作出如下判断：

iderw()中在申请到锁idelock之后，又执行了我们人为添加的sti指令，打开了中断；

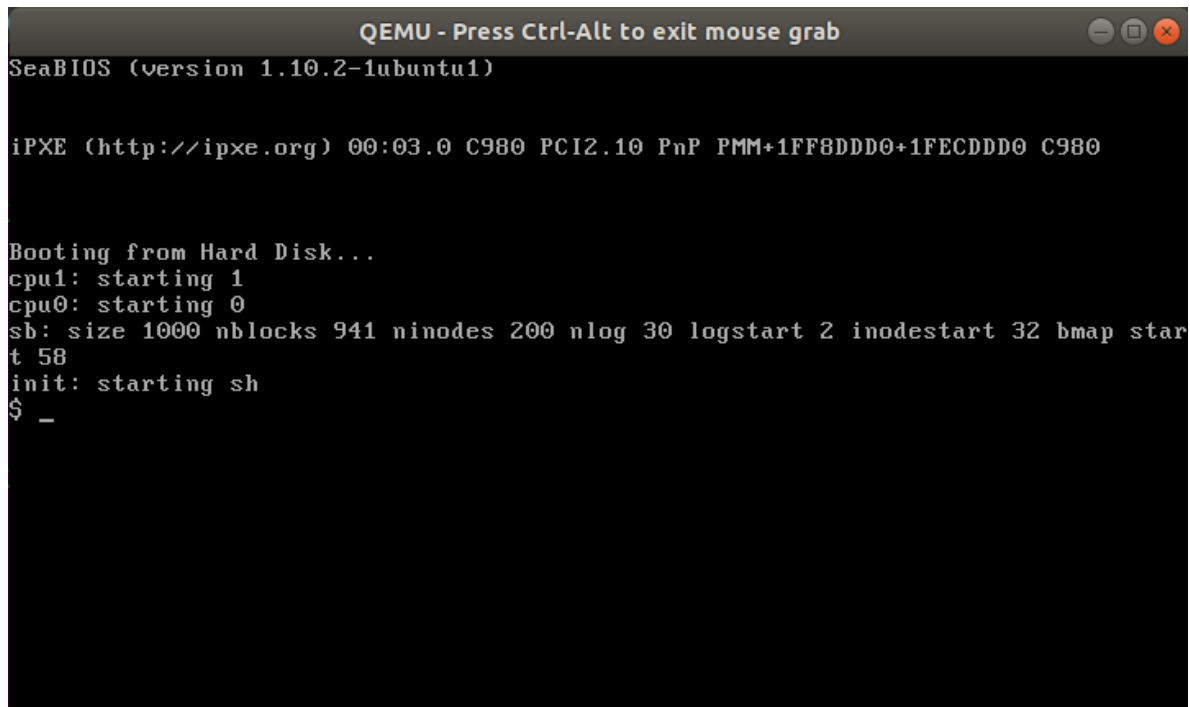
之后在iderw()执行完之前，时间片耗尽，发生时钟中断；

中断处理过程中，发现ncli（嵌套调用pushcli的层数）不等于1，遂触发了panic。

（至于为什么要判断ncli != 1，可能的原因是，在ncli != 1的情况下切换进程是不安全的，有引发死锁的风险[1]。另外也可以结合yield()函数前的注释看出这一点："... must hold only ptable.lock"。）

这样一来，另一个相关的现象也得到了解释：

我在进行这一题实验的时候发现，大多数的运行并不会出现panic：



需要多次尝试（平均5次左右），才会有一次panic出现。

结合前面的结论，可以对此尝试做出解释：只有在iderw()的执行过程中恰好有中断（例如上述的时间片中断）发生才可能导致panic；如果到iderw()执行完毕，时间片都没有耗尽，（也没有发生其他会导致panic的中断，）就不会进入上述的调用过程，也就不会出现panic了。

TODO#4 xv6中互斥锁的实现

请仔细阅读release()代码部分，并解释为什么release()在清除lk->locked之前清除lk->pcs[0]和lk->cpu？为什么不选择在清除lk->locked之后清除lk->pcs[0]和lk->cpu？

这是因为release()并不是原子性的。它的执行过程可能被打断。

如果采用了先清除lk->locked之后清除lk->pcs[0]和lk->cpu的写法，可能会因而发生错误：假设某次执行过程在释放lk->locked之后、清除lk->pcs[0]和lk->cpu之前被打断，则在release()得以继续执行之前，该锁将处于没有锁上(lk->locked == 0)但却含有调试信息(lk->pcs[0]和lk->cpu)的状态，这是有潜在风险的。调试者可能会由此混淆，而且各个进程将可以随意读取上一个上锁者的信息。

而按照实际上的先清除lk->pcs[0]和lk->cpu的写法，如果在清除它们之后、释放lk->locked之前被打断，则锁仍然是锁上的状态，从而不会发生上面的问题。

接下来在对应的源码中解释这个问题：

```
1 void
2 release(struct spinlock *lk)
3 {
4     if(!holding(lk))
5         panic("release");
6
7     lk->pcs[0] = 0;
8     lk->cpu = 0;
9
10    //如果采取了上面讨论的另一种写法，且在执行到这一区域（9行的语句执行之后、17行的语句执行之前）的时候被打断，将造成上面所述的该锁处于没有锁上(lk->locked == 0)但却含有调试信息(lk->pcs[0]和lk->cpu)的状态。
```



```

11  __sync_synchronize();
12
13
14  asm volatile("movl $0, %0" : "+m" (&lk->locked) : );
15
16  popcli();
17 }

```

TODO#5 xv6中信号量的设计

1) xv6系统中没有实现信号量，请设计在xv6中实现基于等待队列的信号量（给出实现代码），可参考如下结构：

```

1 struct semaphore {
2     ...
3 };
4 // 初始化信号量
5 void sem_init(struct semaphore *s, int value) {...}
6 void sem_wait(struct semaphore *s) {...}
7 void sem_signal(struct semaphore *s) {...}

```

代码如下：（该代码为原创）

```

1 struct semaphore
2 {
3     int cnt; // 计数信号量的数值
4     struct proc* q[NPROC]; // 计数信号量的队列；NPROC是xv6中定义的常量，含义是最大
    进程数目
5     int front, rear; // 队列头和尾在数组中的下标
6     struct spinlock mutex; // 用锁保证信号量操作的原子性
7     // 初始化信号量（构造函数）
8     semaphore(int val): cnt(val), front(0), rear(0)
9     {
10         mutex.locked = 0;
11         // queue q is empty by default; nothing to do.
12     }
13 };
14 void wait(struct semaphore *s)
15 {
16     acquire(&(s->mutex)); // 申请互斥锁
17     struct proc* curproc = myproc(); // 当前进程
18     s->cnt -= 1;
19     if (s->cnt < 0)
20     {
21         s->q[s->front] = curproc; s->front = (s->front + 1) % NPROC; // 入队
22         block(); // 阻塞
23     }
24     release(&(s->mutex)); // 解除互斥锁
25 }
26 void signal(struct semaphore *s)
27 {
28     acquire(&(s->mutex)); // 申请互斥锁
29     s->cnt += 1;
30     if (s->cnt <= 0)
31     {
32         struct proc* topproc = s->q[s->rear]; s->rear = (s->rear + 1) %
        NPROC; // 出队

```

```

33     topproc->state = RUNNABLE; // 将出队的进程改为就绪态
34 }
35     release(&(s->mutex)); // 解除互斥锁
36 }

```

2) 选做：使用你实现的信号量设计方案实现哲学家就餐问题，并给出一种解决死锁的方案。

第一种方案，最多允许5位哲学家中的4位同时进入房间：

```

1  semaphore fork[5] = {1, 1, 1, 1, 1}; // “叉子”信号量
2  semaphore room = 4; // “房间”信号量
3  void philosopher(int i) // 哲学家
4  {
5      think(); // 思考
6      wait(room); // 进入房间
7      wait(fork + i); wait(fork + (i+1) % 5); // 取得叉子
8      eat(); // 用餐
9      signal(fork + (i+1) % 5); signal(fork + i); // 释放叉子
10     signal(room); // 退出房间
11 }
12 void main()
13 {
14     parbegin(philosopher(0), philosopher(1), philosopher(2),
15             philosopher(3), philosopher(4));
16 }

```

第二种方案，改变用餐策略，让5位哲学家中的一位改变顺序（从先拿左手叉子再拿右手叉子改为：先拿右手叉子再拿左手叉子）：

```

1  semaphore fork[5] = {1, 1, 1, 1, 1}; // “叉子”信号量
2  void philosopher(int i) // 哲学家
3  {
4      if (i == 0) // 0号哲学家改变顺序
5      {
6          think(); // 思考
7          wait(fork + 1); wait(fork); // 依次取得右手和左手叉子
8          eat(); // 用餐
9          signal(fork); signal(fork + 1); // 依次释放左手和右手叉子
10     }
11     else { // 其他哲学家
12         think(); // 思考
13         wait(fork + i); wait(fork + (i+1) % 5); // 依次取得左手和右手叉子
14         eat(); // 用餐
15         signal(fork + (i+1) % 5); signal(fork + i); // 依次释放右手和左手叉子
16     }
17 }
18 void main()
19 {
20     parbegin(philosopher(0), philosopher(1), philosopher(2),
21             philosopher(3), philosopher(4));
22 }

```

参考资料：

[1] <https://samwho.dev/blog/2013/06/01/context-switching-on-x86/>

Context Switching on x86

