一、源代码阅读

spinlock.c(.h) 这两个文件中定义个自旋锁以及自旋锁的一些操作。

自旋锁类中最主要的是标识锁是否可用的无符号整形变量 locked。当 locked=1 的时候表示锁不可用，locked=0 的时候表示锁是可用的。锁的操作最主要的是两个，一个是获得锁，一个是释放锁。获得锁的方法中，先关闭中断，然后利用 xchg 函数的原子性，循环检查锁是否可用，可用则获得锁，否则循环等待。释放锁的方法中，同样是利用 xchg 函数，将 locked 置为 0，表示释放锁。

二、问题

1、什么是临界区?

临界区是指程序中访问互斥资源的那段代码。涉及两个或两个以上的程序是临界区才有意义。

2、什么是同步和互斥?

同步是指进程间的执行顺序存在某种制约关系，一些进程的执行需要以其他一些进程先执行完为条件。互斥是指不同的进程都需要访问某种一次只能有一个进程访问的资源时出现的关系。

3、什么是竞争状态?

当多个进程需要某种资源的时候就需要展开竞争。只有得到资源的进程可以执行，其他的进程只能等待。等到该进程释放资源的时候，其他进程再去竞争该资源。

4、临界区操作时中断是否应该开启?

所谓临界区是指程序中访问互斥资源的那段代码。临界区操作时必须要关中断。例如当一个进程位于临界区的时候，此时发生中断，处理器转去处理中断程序，若该中断程序也需要上个进程所持有的互斥共享资源，因为资源已经被分配，故会产生死锁。

5、中断会有什么影响?

中断发生的时候，处理器要去运行中断处理程序，这就意味着进程的切换，需要切换上下文等。若频繁的产生中断，则会发生上下文频繁切换的情况。

6、XV6 的锁是如何实现的，有什么操作?

XV6 中实现了自旋锁。在定义过锁之后，当有进程想要获得锁的时候，回去进行判断锁的状态。如果未被占据，则修改锁的状态并进入临界区。否则就保持循环测试锁的状态直到锁可用。XV6 中定义的自旋锁主要有三种操作：锁初始化、请求锁、释放锁。

7、xchg 是什么指令，该指令有何特性?

xchg 指令的作用是测试返回当前锁的状态并将锁的状态设置为一个特定的值。当锁可用的时候，返回 0 值并将锁置为 1。当锁不可用的时候，返回 1 并将锁置为 1。该指令的特性是被封装为原子操作，不可分割。

三、功能拓展

1、基于 spinlock 实现信号量机制

定义信号量类:

```
class Semaphore {
    信号量名字 name;
    信号量值 value;              //记录信号量的值
    进程等待队列 queue;          //记录等待的进程
    spinlock* lock;                //因为要基于自旋锁实现信号量机制，需要定义一个自旋锁
    P();
    V();
}
```

信号量构造函数：

```
Semaphore (名字 name，初始值 value){
    初始值赋值给信号量值；
    名字赋值给信号量名字；
    新定义一个自旋锁；
}
```

信号量的成员方法：

```
void
Semaphore :: P ( ) {
    acquire (lock)；//获得锁，得不到的话进入自旋，此时处于关中断的状态,确保不会有多个进程同时对 value 操作；
    while（value == 0）{                    //说明此时没有资源，需要使用 while 循环检查
        将当前进程放入到等待队列；          //等待唤醒
        进程切换；                          //寻找别的可执行的进程
    }
    value--；                               //退出循环时说明有资源，则资源数减一
    release (lock);                             //完成对 value 的操作，释放锁
}

void
Semaphore :: V(){
    acquire (lock);                             //获得锁，得不到的话进入自旋等待。
    if （等待队列不为空）{                  //说明此时存在等待此资源的进程，可将其唤醒
        将等待队列中的一个进程唤醒；        //此时只是唤醒，还不是直接运行，因此上面 P 操作中采用 while
                                            //循环检查。
    }
    value++；                               //资源数加一
    release(lock);                              //释放锁
}
```

2、基于 spinlock 实现读写锁：

实现读者优先的自旋锁：当有写者在的时候，读者与写者均无法入内；当有读者存在的时候，读者可以直接入内，写者无法入内。

读写锁类的定义：

```
class ReadWriteLock{
    spinlock* lock1;        //保护 count，即每次只允许一个读者访问 count；
    spinlock* lock2;        //保护临界区，不允许读写同时或者多写同时；
    int count;              //记录读者人数；
    char[]* name;           //读写锁的名字；
    以读者身份加锁；
    以读者身份解锁；
    以写者什么加锁；
```

```
    以写者身份解锁;
}
读写锁的构造方法:
ReadWriteLock (名字){
    new lock1;
    new lock2;
    count = 0;
}

void 以读者身份加锁（）{
    acquire（lock1）;     //保护读者数量;
    count++;              //读者数量加 1;
    if （读者数量 == 1）  //说明之前没有读者在临界区
        acquire(lock2);    //尝试关闭临界区入口，当有读者在此自旋等待的时候，其他的读者进程无法进
                           //入获得 lock1，因此不会直接跳过 if 判断导致出错。
    release（lock1）;       /解除对读者数量的保护
}

void 以读者身份解锁（）{
    acquire（lock1）;     //保护读者数量;
    count--;              //读者数减一
    if （读者数 == 0）
        release（lock2）;    //开启临界区入口
    release（lock1）;       /解除对读者数量的保护
}

void 以写者身份加锁（）{
    acquire（lock2）;       //尝试对临界区加锁
}

void 以写者身份解锁（）{
    release（lock2）;         //对临界区解锁
}
```

四、感兴趣的问题

对于 XV6 中所实现的自旋锁，我对于获得锁与释放锁的方法中的关中断 pushcli 函数与开中断 popcli 函数比较疑惑。然后经过查阅资料有了自己的理解。

在 cpu 的定义中有个成员变量叫做 intena，标志着在关中断之前中断是否可用。调用 pushcli 函数的时候，其中有个 cli 函数，这才是真正的关中断的函数。XV6 中对关中断的次数有个计数，虽然关闭一次中断以后再次关闭中断不会有什么实质性的操作，但是中断计数会增加。仅当第一次中断的时候 intena 才为真。这样，当调用 popcli 开中断的时候，只有中断计数为 0 且关闭中断前中断可用的时候，才会真正调用关闭中断的函数 sti 去关闭终端。即中断可以多次关闭，但是关闭几次就要相应的打开几次。