操作系统 Lab3 同步与互斥

宁晨然 17307130178

一、实验目的

了解自旋锁的原理，阅读源代码，看懂自旋锁相关函数。
理解自旋锁对于操作系统同步与互斥的作用。
认识临界区的工作原理，熟悉使用锁进入临界区的过程。

二、实验过程

本实验已全部完成，包括选做内容。

（一）、TODO #1：阅读 spin_lock

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

①xchg()

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

xchg()函数是 xchg 指令的具体操作实现的函数，目的是交换数据。
首先 xchg 函数的参数为，一个地址，和一个值。xchgl %0 %1 就是将值传递到地址的内存，并且返回值为该地址内存存放的值。假设 newval 存在 eax，addr 存在 rbx 中，则汇编指令如下：
lock; xchgl (%rbx), %eax

解释 – lock

lock 是指令前缀，保证了指令对总线和缓存的独占权，即该指令执行过程中不会有其他 CPU 或同 CPU 内的指令访问缓存的内存。xchgl 指令就是交换两个寄存器或者内存地址的 4 字节值，不能同时是内存地址。

关键 – 原子性

保证 xchg()运行正确的是，这个指令是原子性的，不会被中断。

②pushcli()和 popcli()

```
void
pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}
```

```
void
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

两个函数的作用同 cli/sti 作用相同，即禁止中断发生和允许中断发生，不同点在于 pushcli 和 popcli 操作是相互对应的。
pushcli 的作用即：禁止中断发生，屏蔽外部中断保证当前运行的代码不被打断，起到保护代码运行的作用。
popcli 的作用即：允许中断发生，外部中断恢复，可以打破被保护代码的运行，允许硬件中断转而处理中断的作用。

pushcli：操作如下

读取 eflags -> 禁止中断 cli -> 如果第一次 pushcli，则设置 intena 为“eflags 的 IF 位”

```
// Eflags register
#define FL_IF          0x00000200      // Interrupt Enable
```

，即中断允许标志位 -> cpu 当前嵌套 pushcli 深度+1

又因为 cli 是禁止中断操作，会将中断标志置 0，IF=0，所以正常情况下，若第一次 pushcli，intena 会置 0。

popcli:操作如下

如果当前允许中断（IF=1）/嵌套深度=0，都将出错。所以保证了当前状况是，禁止中断

状态 IF=0 且嵌套深度不为 0（有过 pushcli，并且这里自减 1）。

上操作已经减去嵌套深度 1 次。

如果当前嵌套深度为 0 且 intena 为 1 时，可以允许中断，sti 指令执行后 IF 置为 1。

③三个函数在自旋锁中的作用：

自旋锁的目的是当某个线程的一条指令访问某个内存的时候，其他的线程的指令无法访问该内存。其实现就是当线程将进入临界区的时候（访问共享变量），需要先获得锁，如果获取到了就可以进入，如果不能获得就忙等（什么都不做直到获得锁）。

```
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

可以从 acquire 锁的时候看他们的功能。因为临界区的代码需要保证原子性，所以不允许期间有并发的竞争冒险，所以要保证没有中断发生，所以第一步就是禁止中断。

下面判断如果 cpu 正在占用锁，则 panic。

while 循环就是自旋操作了：

xchg 指令交换 locked 和 1 的值，该指令是原子性的（本来也不可能被中断），假设 1 的值在 eax，则交换后 eax 存放了 locked 的值作为返回值与 0 比较。如果 locked 值为 1，则不能进入临界区。如果 locked 值为 0，则可以继续进入临界区。__sync_synchronize 同步操作的作用（查资料后得知）：

最后，acquire()函数使用__sync_synchronize 为了避免编译器对这段代码进行指令顺序调整的话和避免 CPU 在这块代码采用乱序执行的优化。

```
int
holding(struct spinlock *lock)
{
  int r;
  pushcli();
  r = lock->locked && lock->cpu == mycpu();
  popcli();
  return r;
}
```

再查看一个 holding 函数可以知道 pushcli 和 popcli 的操作作用。为了保证原子性，holding 操作必须要防止中断，所以每次都需要先确认防止中断、再允许中断。

综上所述，三个函数分别的作用是：

xchg 拥有原子性，用于自旋操作并比对 locked 是否为 0。

pushcli 和 popcli 成对存在，用于保证 acquire/holding 等函数运行过程中不会被中断。（因为如果中断到来可能执行了中断例程，而中断例程访问临界区，就与当前对临界区的操作冲突，原子性被破坏；还有可能中断到来执行了其他进程，其他进程再次请求锁，导致死锁）

（二）、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);
```

该代码用于测试锁功能。

先创建一个自旋锁 test_lk，使用 initlock 函数对 test_lk 初始化，赋予名字、locked 初始为 0、锁占用 cpu 为 0。使用 acquire 函数申请锁，并且会申请成功。第二次使用 acquire 函数申请锁不会成功，因为现在有 cpu 正在占用锁，所以会输出 panic 信息。

“acquire”

```
if(holding(lk))
  panic("acquire");
```

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。
提示：你可以尝试将此代码片段加入系统源码中（自行选择合适的位置）运行。

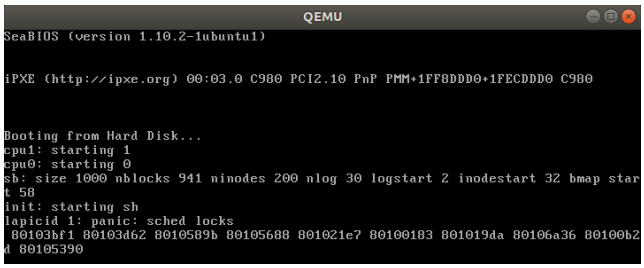
```
static struct proc*
allocproc(void)
{

    cprintf("test lock!\n");
    struct spinlock test_lk;
    initlock(&test_lk,"Test Lock in XV6");
    acquire(&test_lk);
    acquire(&test_lk);

  struct proc *p;
  char *sp;
```

```
Booting from Hard Disk...
cpu1: starting 1
test lock!
lapicid 0: panic: acquire
```

将代码添加到 allocproc 中，每次分配进程时都会调用该代码，可见一开始打开的时候就出了错。panic 输出 acquire。
3）解释运行代码片段后出现的结果。
第一次 acquire 成功了锁，此时锁处于被某个 cpu 进程 holding 状态，所以第二次 acquire 的时候就会出错，访问不了该锁。

（三）、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）。




```
7549     panic("sched locks");
7550 80103be4:   83 ec 0c                sub    $0xc,%esp
7551 80103be7:   68 a2 74 10 80          push   $0x801074a2
7552 80103bec:   e8 9f c7 ff ff          call   80100390 <panic>
7553 80103bf1:   eb 0d                   jmp    80103c00 <exit>
```

```
void
iderw(struct buf *b)
{
  struct buf **pp;

  if(!holdingsleep(&b->lock))
    panic("iderw: buf not locked");
  if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if(b->dev != 0 && !havedisk1)
    panic("iderw: ide disk 1 not present");

  acquire(&idelock);  //DOC:acquire-lock
    sti();
```

输出结果如上，出现了 panic 信息 sched locks，查看 kernel.asm 中对应 80103bf1 前后的代码可知，这个 panic 出现是在 sched()中出现的，而该信息的输出说明条件 mycpu()->ncli!=1 成立。

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

sched()函数是进程列表 ptable 的调度程序，如果要进入调度程序，必须满足：①有且仅有 ptable 一个锁②当前进程状态已经非运行状态③禁止中断状态，才可以进行进程调度上下文切换操作。
由于 iderw 中一开始 acquire 了锁，ncli++，但是 sti 释放了锁，可能会在程序执行过程中被中断，进入调度程序，此时调度程序检查到 ncli 并不符合调度条件所以 panic。
经过测试确实如此，并不是每次 make qemu 都会出现该错误，原因是不是每次都有可能 iderw 中处于允许中断的代码段都会被进程中断，然后进入调度程序。

acquire 和 sti，cli 和 release 看似已经匹配，但是这样有两个问题：

A．acquire 和 release 中间部分的代码并没有上锁。该代码处于允许被中断的状态，可能被调度程序调度到其他进程。

B．acquire 和 sti 并不匹配，因为 sti 和 cli 都不会对 mycpu()->ncli 进行加减操作，即

不会改变嵌套深度的大小，这会导致调度程序检测到异常，因为此时并不是只有 ptable 一个锁。

（四）、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？

```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli();
}
```

```
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};
```

因为 lk->locked 置 0 后再清除 lk->pcs[0]和 lk->cpu 可能会发生：其他请求该锁的进程请求成功，并且改变了 lk->pcs[0]和 lk->cpu，但是此时 release 继续执行就会将 lk->pcs[0]和 lk->cpu 清零，导致锁信息丢失。

之前清除的好处在于：①保证了锁的互斥：只有一个进程访问该锁。②保证了原子性：所有该进程与锁相关的操作均在 lk->locked 等于 1 时进行。③保证了信息稳定：防止锁信息丢失。

（五）、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：

- struct semaphore {
- };
- // 初始化信号量
- void sem_init(struct semaphore *s, int value) {};
- void sem_wait(struct semaphore *s) {} ;
- void sem_signal(struct semaphore *s) {};

```
struct semaphore {
  int value;
  struct proc *list[NPROC];
  struct spinlock lock;
  int start,end;
}

void sem_init(struct semaphore *s, int value){
  s->value = value;
  initlock(&s->lock,"semaphore_lock");
  s->start = s->end = 0;
}
```

```
void sem_wait(struct semaphore *s){
  acquire(&s->lock);
  s->value--;
  if(s->value < 0){
    struct proc* p = myproc();
    s->list[s->end] = p;
    s->end = (s->end+1) % NPROC;
    sleep(p, &s->lock);
  }
  release(&s->lock);
}

void sem_signal(struct semaphore *s){
  acquire(&s->lock);
  s->value++;
  if(s->value <= 0){
    wakeup(s->list[s->start]);
    s->list[s->start] = 0;
    s->start = (s->start+1)%NPROC;
  }
  release(&s->lock);
}
```

实现信号量的代码如图。信号量的关键在于 wait()和 signal()两个函数，而现需要保证 semaphore 结构中有一个锁，因为需要保证信号量改变过程中不能被并发进行。wait 操作中的核心就是 value—后如果<0，则将该进程阻塞（与忙等不同），阻塞在这里的实现就是将该进程睡眠，所以 sleep 操作。

signal 的核心就是 value++后，如果<=0 则可以唤醒信号量中进程等待队列中的首进程。

每次操作队列都要记得修改 start 和 end（指针）。

每次操作前后都需要加上锁保证信号量的原子执行。

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。

```
struct semaphore *chopstick[5];
void eat();
void think();

void thinker(int i)
{
  do
  {
    if (i % 2)    {
      wait(chopstick[i]); //odd
      wait(chopstick[(i + 1) % 5]);
    }
    else    {
      wait(chopstick[(i + 1) % 5]); //even
      wait(chopstick[i]);
    }
    eat();
    if (i % 2)    {
      signal(chopstick[i]); //odd
      signal(chopstick[(i + 1) % 5]);
    }
    else    {
      signal(chopstick[(i + 1) % 5]); //even
      signal(chopstick[i]);
    }
    think();
  }while(1);
}
```

解决死锁的方案，此处使用管程略显复杂，可以简单的使用非对称解决方法：

奇数哲学家先拿起左边的筷子，接着拿起右边的筷子；偶数哲学家先拿起右边的筷子，再拿起左边的筷子。

实现过程就是保证非对称。

每个 chopstick 都作为信号量，如果要 eat 的时候就必须先请求左右信号量，但由于奇数和偶数的拿起顺序不同，所以不会造成死锁。

三、实验感受

本次实验充分体会到了锁的作用。自旋锁需要用来保证临界区的原子性和非并发性，并且保证临界区不被中断。也可以使用信号量来保证该特性。锁赋予了临界区共享变量不被扰乱的特性，在操作系统中经常需要使用到，但是也不能过度依赖，因为会降低并发性。

该实验的难点在于读懂锁的几个函数，acquire/release/holding，都是锁的基本操作，看懂之后对自旋锁的理解也加深了不少。