北京大学

同步机制

XV6 源码阅读报告

组长： 罗 登 2001210364

组员1： 毕廷竹 2001210186

组员2： 周旭敏 2001210723

组员3： 罗旭坤 2001210368

目录

问题回答

问题 1：什么是临界区？

首先是**临界资源**的概念。多个进程可以共享系统中的各种资源，但其中有些资源，一次只能被一个进程使用。一次只能被一个进程资源称为临界资源。例如：许多物理设备都属于临界资源，如打印机等。内存中的一些变量、数据等也可以被若干进程共享，也属于临界资源。对于临界资源的访问，必须互斥（Mutual Exclusion）地进行，访问临界资源的那段代码称为**临界区**。因此进程的访问过程可以抽象成如下过程：

```
do {
   entry section;    // 进入区
   critical section; // 临界区
   exit section;     // 退出区
   remainder section;// 剩余区
} while(true);
```

- 进入区。为了进入临界区使用临界资源，在进入区要检查是否可以进入临界区，若能进入临界区，则应设置正在访问临界区的标识，以阻止其他进程同时进入临界区。
- 临界区。进程中访问临界资源的那段代码，又称为临界段。
- 退出区。将正在访问临界区的标志清除。
- 剩余区。代码中的剩余部分。

使用图形化表示可以给出如下描述，见图 1，其中 *P1,P2,P3,P4* 四个进程试图访问临界资源，前面的黄色横杆可以视为进入区，对进程进行选择操作，并设置对应的标识为。虚线框出为临界区代码。

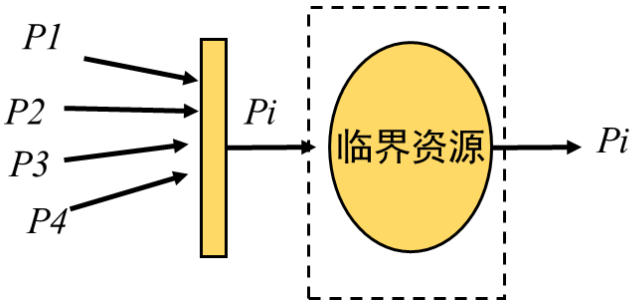


图 1 临界资源访问

问题 2：什么是同步和互斥？

同步 Synchronization。同步也被称为**直接制约关系**，是指为完成某种任务而建立的两个或多个进程，这些进程因为需要在某些位置上协调它们的**工作次序**而等待、传递消息所产生的制约关系。进程间的直接制约关系源于它们之间的互相合作。

例如，输入进程 A 通过单缓冲区向进程 B 提供数据。当该缓冲区为空时，进程 B 不同获得所需数据而阻塞，一旦进程 A 将数据送入缓冲区，进程 B 就被唤醒。反之，当缓冲区满时，进程 A 被阻塞，仅当进程 B 取走缓冲数据时，才

唤醒进程 A。图示如下。

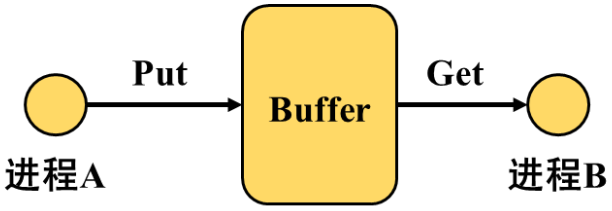
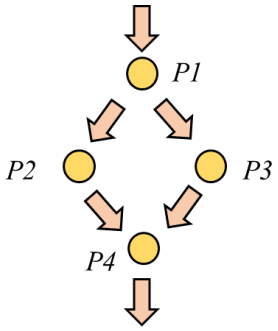


图 2 通过缓冲区实现进程同步



同步关系（进程前驱图）

图 3 一种更广义的同步关系，前驱图

同步关系是进程合作较为高级的一种状态，通常通过信号量机制来实现。而且需要清楚的一点是，要实现进程同步，对临界资源的互斥访问也是必不可少的。例如在上图进程 A/B 通过缓冲区完成同步时，对于 Buffer 的访问应该是不能同时进行的，也就是说，Buffer 是一种临界资源，必须互斥访问。

互斥 Mutual Exclusion。互斥也被称为**间接制约关系**。当一个进程进入临界区使用临界资源的时候，另外一个进程必须等待，当占用临界资源的进程退出临界区后，另外一个进程才被允许去访问该临界资源。通常使用**锁**机制来实现互斥机制，也就是将对临界资源的争用抽象为对锁的争用，只要进程拿到锁，则认为其拿到了临界资源，其他进程也无法访问了。

一般来说互斥是对临界资源来说的，只有讨论的问题是临界资源，才考虑其互斥。但是这里的临界资源种类有很多，不单是硬件如打印机、I/O 设备等，对于内存中的变量的访问，信号量等均可视为一种临界资源。正如之后将看到的，实现信号量机制的时候，也需要在内部使用锁。

对临界区的访问需要满足以下原则：

- 空闲让进。临界区空闲时，可以允许一个请求进入临界区的进程立即进入临界区。
- 忙则等待。当已有进程进入临界区时，其他试图进入临界区的进程必须等待。
- 有限等待。对于请求访问的进程，应保证能在有限时间内进入临界区。
- 让权等待。当进程不能进入临界区时，应立即释放处理器，防止进程忙等待。

关于临界区的互斥，可以使用软件方法和硬件方法，它们的特点和对比如下：

- 软件实现方法
  - ✧ 单标志法。必须交替进入，违背“空闲让进”原则。
  - ✧ 双标志先检查。不必交替进入，可能同时进入，违背“忙着等待”原则。
  - ✧ 双标志后检查。进程间互相谦让，造成都不能进入的饥饿现象。
  - ✧ Peterson’s Algorithm。能够正确实现互斥访问，但是较为复杂。

- 硬件实现方法
  - ✧ 中断屏蔽方法。进入临界区前关中断，访问结束后开中断。效率不高，而且有可能导致系统不安全。
  - ✧ 硬件指令方法。使用 TestAndSet 指令或 Swap 指令等具有原子性操作的指令。在 xv6 中使用 xchg 指令可以理解为这种实现方法。

问题 3：什么是竞争状态？

竞态（Race Condition）。又称为竞争冒险或竞态条件，旨在描述一个系统或进程的输出依赖于**不受控制**的事件出现顺序，或者出现时机。竞争冒险常见于不良设计的电子系统中，以及未合理设计并发控制的多线程程序。

举个例子，两个进程试图同时修改一个共享内存的内容，在没有并发控制的情况下，最后的结果依赖于两个进程的执行顺序与时机，而这是不可控的（异步性）。因此结果可能是不正确的。

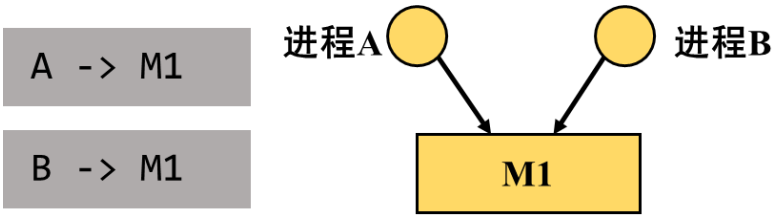


图 4 进程竞争状态实例

问题 4：临界区操作时中断是否应该开启？会有什么影响？

关中断机制的形式化表示如下：

```
...
关中断;
临界区;
开中断;
...
```

如果使用了关中断机制，当一个进程在访问临界区代码的使用，是关闭了整个系统的中断，这样就确保了在临界区内不会发生进程切换，保证了当前运行的进程可以在临界区执行完毕，确保了互斥的正确实现。但是其存在至少以下两点问题：

1. 效率低下。限制了处理机交替执行程序的能力，因此整个系统的执行效率会显著降低。
2. 不安全。对于内核来说，在更新变量或者列表中的几条指令期间，关中断是很方便的，但是将关中断的权利交给用户是很不明智的，如果一个用户进程关中断后不再开启，系统可能会因此终止。

通过对 xv6 源码的阅读，可以发现其在获取锁的 acquire 函数中调用了关闭中断的指令，并且在释放锁 release 函数之前均没有开启中断。因此可以理解为 xv6 中在临界区访问时关闭了中断。

问题 5：xv6 中的自旋锁是如何实现的？有什么操作？

自旋锁。自旋锁是计算机科学用于多线程同步的一种锁，线程反复检查锁变量是否可用。由于线程在这一过程中保持执行，因此是一种忙等待。一旦获取了自旋锁，线程会一直保持该锁，直至显式释放自旋锁。

自旋锁避免了进程上下文的调度开销，因此对于线程只会阻塞很短时间的场合是有效的。

单核单线程的 CPU 不适于使用自旋锁，因为，在同一时间只有一个线程是处在运行状态，假设运行线程 A 发现无法获取锁，只能等待解锁，但因为 A 自身不挂起，所以那个持有锁的线程 B 没有办法进入运行状态，只能等到操作系统分给 A 的时间片用完，才能有机会被调度。这种情况下使用自旋锁的代价很高。

获取、释放自旋锁，是通过读写自旋锁的存储内存或寄存器。因此这种读写操作必须是原子的。通常用 testAndSet 等原子操作来实现。

需要明确的是，自旋锁是对互斥锁的一种实现。在通常的互斥锁中，一个进程如果发现当前资源不可用，则会选择阻塞自身，主动放弃 CPU。而对于自旋锁而言，其会保持 CPU 并不断循环测试，一旦锁可用就占用它，因此是一种忙等的状态。这种方式减少了进程调度的次数，因此能带来性能上提升。但是仅限于在多 CPU 环境中才能使用，而且要求临界区代码较短，其他等待过程能够在较短时间内得到锁，这样才能效率高。

对自旋锁的理解可以类比于上厕所。对厕所这种临界资源的访问是互斥的，因此需要争用锁来实现。当看到厕所中有人是，自旋锁实现的方式是在门外等待并不断测试门内是否有人。而普通互斥锁实现的方案中，进程则是选择离开。

xv6 中自旋锁的实现。xv6 中在 spinlock.h 头文件中定义了 struct spinlock，并且在 spinlock.c 文件中提供了与其相关的操作函数。

```
// spinlock.h
struct spinlock
{
  uint locked; // Is the lock held? 1表示持有，0表示未持有，汇编修改

  // For debugging: 用于调试的信息
  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.
};
```

```
// spinlock.c
void initlock(struct spinlock *lk, char *name);  // 初始化
void acquire(struct spinlock *lk);               // 获取
void release(struct spinlock *lk);               // 释放
int holding(struct spinlock *lock);              // 查看当前cpu释放持有
```

图 5 xv6 中的自旋锁概览

struct spinlock 的核心主要在于维护其中 locked 这个数据成员的值，为 0 或者为 1，分别表示当前锁是否被占有。使用了原子操作（汇编指令）来实现对 locked 这个值的测试与修改。操作该锁核心的两个函数是 acquire 和 release 这两个函数，其中又分别涉及到了 popcli 和 pushcli，以及 readeflags 等函数，下面对其做进一步的分析。

pushcli 函数。该函数实现的主要功能是关闭系统中断，通过在内部调用 cli

函数（该函数使用内联汇编调用 cli 指令）完成关闭系统中断的功能。pushcli 函数首先调用 readeflags 将标志寄存器 eflags 的信息读入到 C 语言变量中，然后调用 cli 函数，系统中断关闭。接下来判断当前 cpu 上的 ncli 值是否为 0，并且在后面对其进行了加 1 操作，该值的定义是当前关闭中断的次数。在小组讨论中我们对其作用进行了探讨，也就是 ncli 具体表达了什么值？通过分析发现，只有在当前进程继续请求 acquire 其他自旋锁的时候，才可能导致 ncli 变量的继续增大，也就是说 ncli 表示了当前线程持有的锁个数或称临界资源个数。其中 mycpu()->intena 保存了在关闭中断之前系统中断开启的状态。

popcli 函数。这个函数的主要作用是使用 sti 函数（内部内联调用 sti 指令）来开启系统中断。使用 readeflags() & FL_IF 来获取到当前系统中中断开启的状态信息。FL_IF 宏定义在 mmu.h 头文件中。上式的主要作用是取出 eflags 中表示系统中断信息对应位的值，在 if 语句中进行判断。在执行 popcli 函数是要求系统中断是关闭状态的，否则进入 panic 状态。然后对 ncli 值进行自减操作，如果当前 ncli 即占有锁的数量是 0 且之前是中断启动的状态，调用 sti 函数开启系统中断。

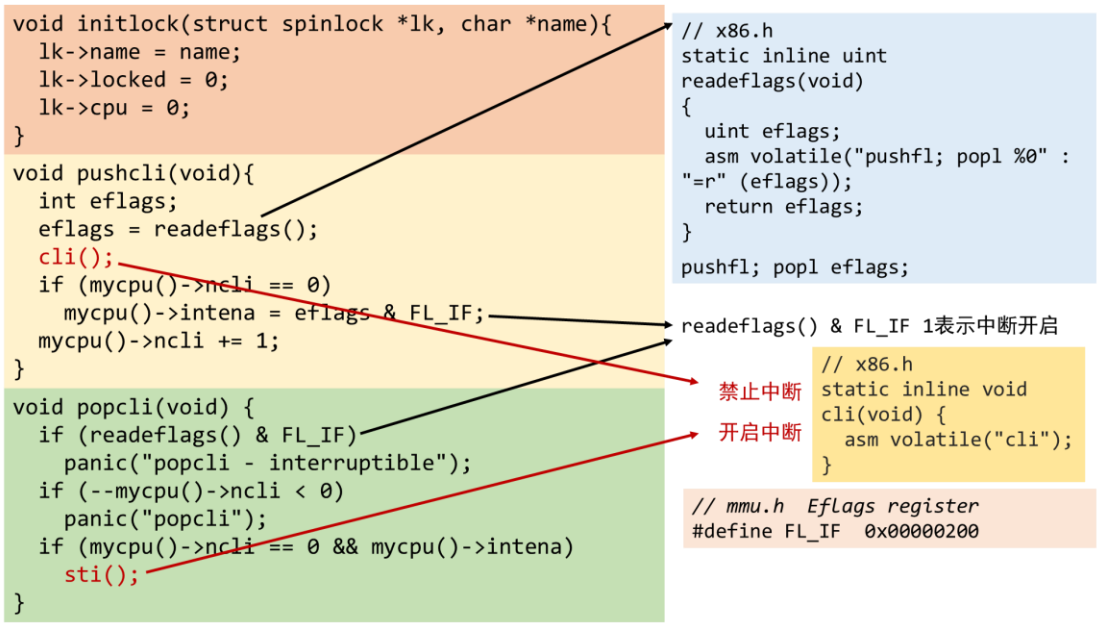


图 6 xv6 中自旋锁相关的函数

综合分析上述两个函数，以及在进程同步与死锁问题中我们有了解到，如果一个进程需要请求多个资源，并且多个进程请求过程中出现了环路，则可能出现死锁现象。xv6 对于这种情况采用了一次性分配的原则，当进程请求到所有资源并且运行结束后，才将系统中断开启。

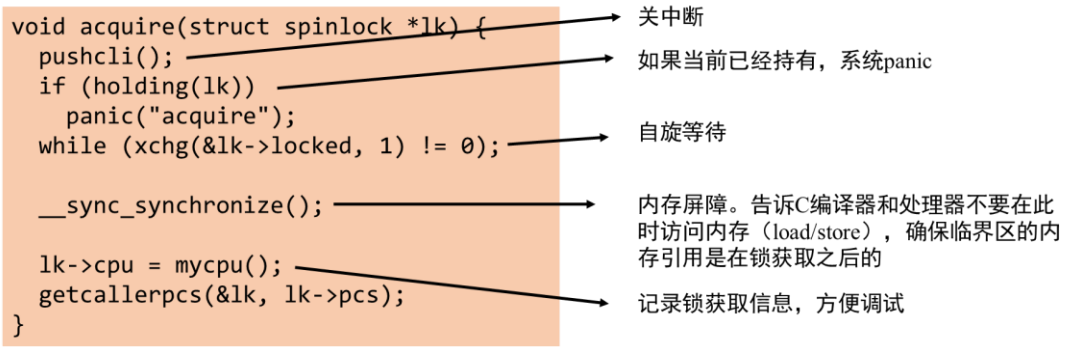


图 7 acquire 函数的分析

acquire 函数。acquire 函数是实现锁获取的函数。在进入该函数前，首先调用 pushcli 关闭中断。之后查看是否已经持有该锁，如果持有，则进入 panic 状态。因此避免了一个进程在获取到当前锁后继续尝试加锁，也就是递归加锁的过程。之后是最重要的过程，也就是自旋操作的实现。在 while 循环中，使用 xchg 函数交换 lk->locked 与 1 的值，而这个函数的返回值是 lk->locked 原来的值，如果为 1，表示之前已经上锁，因此在 while 循环中自旋。如果 lk->locked 的值为 0，则表示该锁可被占有，因此获取该锁，并退出 while 循环，向下执行。

__sync_synchronize()方法是 GCC 提供的内存屏障功能，告诉 C 编译器和处理器不要在此时访问内存（load/store），确保临界区的内存引用是在锁获取之后的，避免内存争用现象。

之后是辅助调试功能，将 mycpu()赋值给 lk->cpu，并且使用 getcallerpcs 函数保存下调用的 pc 路径，在出现系统 panic 后，会将其打印出来，用于分析和调试。

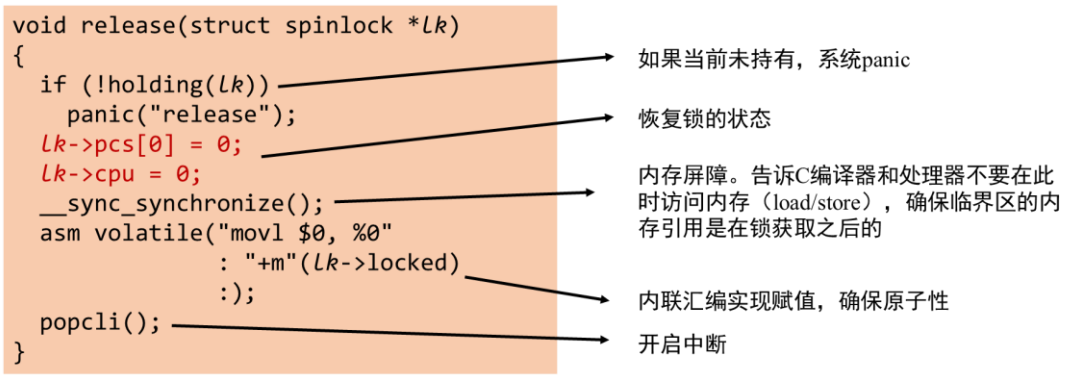


图 8 release 函数的分析

release 函数。release 函数的功能是释放当前锁，首先判断是否持有该锁，若不持有，进入 panic 状态。之后恢复锁的状态，将 lk->pcs[0]和 lk->cpu 赋值为 0，即恢复到初始态。之后是__sync__synchronize()调用，添加内存访问屏障。然后通过一条内联汇编指令，将 lk->locked 的值修改为 0，最后调用 popcli 清理 ncli，降低关闭深度，如果深度为 0，则开启系统中断。

问题 6：xchg 是什么指令，该指令有什么特性？

xchg 指令。xchg 指令是一个双操作数指令，其功能是交换两个操作数的内容。有以下三种形式：

```
XCHG reg1, reg2 ; 寄存器<->寄存器
XCHG reg, mem ; 寄存器<->内存
XCHG mem, reg ; 内存寄<->存器
```

也就是说，该指令可以在两个寄存器间进行交换，也可以在寄存器也内存之间进行交换（xv6 中 xchg 函数使用了内存与寄存器间互相交换实现原子交换操作）。由于在 CPU 中，中断只允许在两条指令执行间发生，因此单条指令的执行可以视为具有原子性的。

使用 xchg 指令，要求两边的操作数长度（类型）相同。下面给出几个使用的例子：

```
xchg ax,bx      ;交换 16 位寄存器内容
xchg ah,al      ;交换 8 位寄存器内容
xchg var1,bx    ;交换 16 位内存操作数与 BX 寄存器内容
xchg eax,ebx    ;交换 32 位寄存器内容
```

xchg 指令使用时对于操作数还要遵循以下要求：

- 不能都为内存操作数
- 任何一个操作数都不能为段寄存器
- 任何一个操作数都不能为立即数
- 两个操作数的长度不能不相等

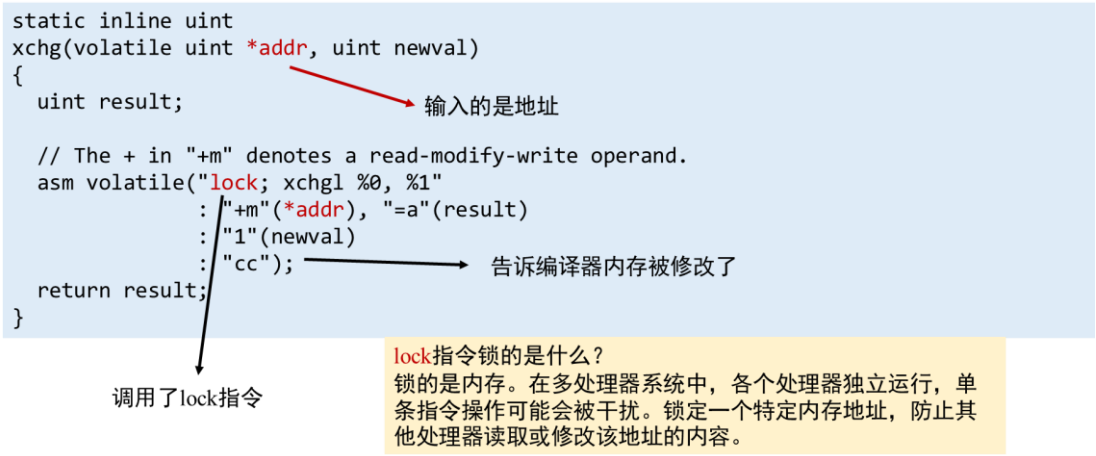


图 9 xchg 指令在 xchg 函数中的使用分析

xchg 函数对 xchg 指令进行了封装，实现了具备原子特性的交换。其内部使用了 gcc 内联汇编语法，关于这个部分的语法，会在扩展部分提到。首先使用 lock 指令，对内存进行加锁，确保了这个区域的内存地址在此时刻只能被一个处理器读取或修改，避免了因为多处理器导致的不一致性。输入参数中第一个是地址，使用了 volatile 关键字修饰，显式告诉编译器，对这个内存地址禁用 cache，避免 cache 与内存的不一致性。

使用自旋锁实现信号量和读写锁的设计方案

信号量实现设计方案

信号量是一种较强的进程同步机制，可以用来解决同步和互斥的问题。考虑

有多种临界资源的情况，信号量的核心是维护一个计数器，表示当前可用的资源数量。提供两个操作 P 操作和 V 操作，如果当前信号量为正，表示资源可用，P 操作将信号量减一。如果进程使用临界资源完毕，使用 V 操作将信号量加一。与自旋等待不同，信号量机制中当资源不够的情况，进程一般选择阻塞自身，因此还要维护一个等待队列。因此可以得到一个简单的信号量结构设计。

```
1  struct semaphore {
2    int value;
3    struct spinlock lock;
4    struct proc *queue[NPROC];
5    int end;
6    int start;
7  };
```

初始化信号量操作。

```
1  void sem_init(struct semaphore *s, int value) {
2    s->value = value;
3    initlock(&s->lock, "semaphore_lock");
4    s->end = s->start = 0;
5  }
```

sem_wait（Wait）操作的定义：

```
 1  void sem_wait(struct semaphore *s) {
 2    acquire(&s->lock);
 3    s->value--;
 4    if (s->value < 0) {
 5      s->queue[s->end] = myproc();
 6      s->end = (s->end + 1) % NPROC;
 7      sleep(myproc(), &s->lock);
 8    }
 9    release(&s->lock);
10  }
```

sem_signal（Signal）操作的定义

```
 1  void sem_signal(struct semaphore *s) {
 2    acquire(&s->lock);
 3    s->value++;
 4    if (s->value <= 0) { // 还有等待
 5      wakeup(s->queue[s->start]);
 6      s->queue[s->start] = 0;
 7      s->start = (s->start + 1) % NPROC;
 8    }
 9    release(&s->lock);
10  }
```

读写锁实现设计方案

在多线程程序中，经常出现读多写少的场景，多个线程进行读数据时，如果都加互斥锁，这降低了性能，显然是不必须的。于是读写锁便应运而生。

读写锁遵循以下规则：

1. 如果没有加写锁时，那么多个线程可以同时加读锁；如果有加写锁时，不可以加读锁；
2. 不管是加了读锁还是写锁，都不能继续加写锁。

基于xv6提供的自旋锁，可以尝试实现读写锁[17]，使用两个自旋锁read_lock和write_lock，分别表示读锁和写锁，并使用一个计数器cnt，标识当前正在进行读操作的进程数量。其具体结构定义如下：

```
1  // rw_lock.h
2  struct rw_lock {
3      int cnt;
4      struct spinlock read_lock;
5      struct spinlock write_lock;
6  };
```

调用spinlock中的initlock函数完成初始化，将cnt设置为0，初始化函数如下：

```
1  void rw_lock_init(struct rw_lock* lock) {
2      initlock(&(lock->read_lock), "rw_read_lock");
3      initlock(&(lock->write_lock), "rw_write_lock");
4      lock->cnt = 0;
5  }
```

当读者需要加锁时，首先获取到读锁，获取成功后，cnt自增，如果当前是读者是第一个，则还需要获取写锁write_lock，读者加锁具体操作如下：

```
1  void readerLock(struct rw_lock* lock) {
2      acquire(&(lock->read_lock));
3      if (++lock->cnt == 1) {
4          acquire(&(lock->write_lock));
5      }
6      release(&(lock->read_lock));
7  }
```

类似的，读者释放锁的过程中也需要加入对计数器的判断，如果当前是最后一个离开的写者，则需要将写锁释放，通知写者可以进行写操作，具体实现如下：

```
1  void readerUnLock(struct rw_lock* lock) {
2      acquire(&(lock->read_lock));
3      if (--lock->cnt == 0) {
4          release(&(lock->write_lock));
5      }
6      release(&(lock->read_lock));
7  }
```

写者操作相对简单，只需要调用对应方法，尝试获取或释放写锁write_lock即可：

```
1  void writerLock(struct rw_lock* lock) {
2      acquire(&(lock->write_lock));
3  }
```

```
1  void writerUnLock(struct rw_lock* lock) {
2      release(&(lock->write_lock));
3  }
```

上述锁实现了基本的读写锁，但是存在一定的不公平现象，即对写线程不公平，如果读线程过多，则写线程一直无法得到写锁，不能将最新的信息写入，这可能导致读者读到的信息都是滞后的，要实现读写公平或写线程优先的锁，还需要更好的设计。

小组讨论问题汇总

1. 在 acquire 锁和 release 锁之间，系统中断是一直关闭的吗？

是。通过代码分析来看，xv6 在获取锁后就一直未开启中断，直到对所有的锁全部释放才开启系统中断。

2. 自旋锁能否递归？如何实现？

xv6 中的自旋锁不支持递归，也就是尝试在本身锁上再次加锁会进入到 panic 状态。递归锁也被称为可重入锁（reentrant mutex），非递归锁又叫不可重入锁(non-reentrant mutex)。二者唯一的区别是，同一个线程可以多次获取同一个递归锁，不会产生死锁。而如果一个线程多次获取同一个非递归锁，则会产生死锁。

Windows 下的 Mutex 和 Critical Section 是可递归的。Linux 下的 pthread_mutex_t 锁默认是非递归的。可以显式地设置 PTHREAD_MUTEX_RECURSIVE 属性，将 pthread_mutex_t 设为递归锁。如果将这两种锁误用，很可能造成程序的死锁。例如：

```
1  MutexLock mutex;
2  void foo() {
3      mutex.lock();
4      // do something
5      mutex.unlock();
6  }
7  void bar() {
8      mutex.lock();
9      // do something
10     foo();
11     mutex.unlock();
12 }
```

foo 函数和 bar 函数都获取了同一个锁，而 bar 函数又会调用 foo 函数。如果 MutexLock 锁是个非递归锁，则这个程序会立即死锁。因此在为一段程序加锁时要格外小心，否则很容易因为这种调用关系而造成死锁。

但是这并不意味着应该用递归锁去代替非递归锁。递归锁用起来固然简单，但往往会隐藏某些代码问题。比如调用函数和被调用函数以为自己拿到了锁，都在修改同一个对象，这时就很容易出现问题。因此在能使用非递归锁的情况下，应该尽量使用非递归锁，因为死锁相对来说，更容易通过调试发现。程序设计如果有问题，应该暴露的越早越好。

考虑读写锁的情况，当读者持有锁的时候，此时为可重入锁，也即递归锁，相同的线程尝试加锁也是不会出现问题的。当写者持有锁时，为不可重入锁，应该禁止相同写者线程加锁。

实现可重入锁可用通过计数器的方法，每当线程尝试持有该锁的时候，将计数器增加，当线程放弃该锁的时候，计数器减少，直到计数器为 0 时，销毁或释放该锁。

3. acquire 关闭系统中断时，只调用了 cli 这个指令，其他 CPU 上中断会受影响吗？

讨论认为，不会。当前关中断指令只在当前 CPU 上起作用，而其他 CPU 不受影响。因为如果其他 CPU 上的中断也被关闭，那么临界区外进程也无法调度到其他 CPU 上执行，这与单 CPU 单线程无异，降低了系统的效率。

4. 进程在临界区内，是否可以被调度？

一般来说，进程在临界区内，是可以被调度的，而且被调度下去是很正常的。如果采用了锁相关的机制，在临界区的进程是可以被调度下 CPU 的，因为其一直持有锁，因此保证了临界区访问的互斥性。

因为在临界区中的进程可能被调度下 CPU，才有了优先级翻转的问题出现。优先级翻转的例子是当一个低优先级的进程 L 在临界区内，没有访问结束的时候，这时它被抢占，下了 CPU，这时系统中进入了优先级进程 M 和高优先级进程 H，假设 H 需要使用临界资源，因此其即使调度上 CPU 后，也无法执行下去，只能被阻塞。由于 M 的优先级更高，因此 M 上 CPU 执行，如果 M 进程很多，则原本优先级更高的 A 进程始终无法得到执行，因此就出现了优先级翻转的现象。

5. CPU 检查指令特权级是在什么时候？

CPU 在指令译码后会查看相应特权位。CPU 跟踪判断当前指令的特权级别主要是通过段选择器来实现的。主要是数据段选择器和代码段选择器。

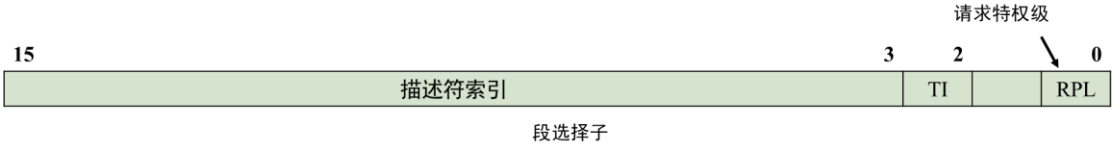


图 10 段选择子与请求特权位

6. xv6 中大量使用了内联汇编，语法是什么？怎样看懂？

内联汇编（Inline Assembly）指的是在 C 语言程序中嵌入汇编程序代码，这样主要有三点好处：1）能够在某些关键的操作中提升效率，减少内存等，因为汇编代码的效率的最高的；2）可以使用操作系统或与平台相关的一些特殊指令，例如原子操作；3）可以用来实现一些系统调用。

GCC 中采用的内联汇编格式遵循了 AT&T 汇编语法，这种语法与 Intel 汇编的语法格式有许多不同，主要包括：

- 源/目的操作数方向
- 寄存器命名
- 立即数
- 操作数大小
- 内存操作数

具体可以参考资料[14]中的表述。

使用内联汇编的基本格式是使用 asm 或__asm__关键字将汇编指令包含在内，但是 GCC 也提供了扩展内联汇编的用法，其功能更加强大，也更加常用。基本格式如下：

```
asm [ volatile ] (
        assembler template
        [ : output operands ]                   /* optional */
        [ : input operands  ]                   /* optional */
        [ : list of clobbered registers ]       /* optional */
        );
```

- asm/__asm__，表示汇编的开始；
- volatile/__volatile__，表示不需要编译器对汇编代码做任何优化；
- 汇编模板；
- 输出操作数；
- 输入操作数；
- 一个寄存器列表（可能被修改的寄存器）。

使用内联汇编，能够在 C 语言中直接使用汇编指令操作寄存器或者内存，实现单条指令的原子操作，并且其提供的扩展语法，使得这个功能非常强大。

7. xv6 中 sleeplock 的实现。

在 xv6-public 中，还引入了 sleeplock，也就是睡眠等待锁，简单理解就是通常意义上的互斥锁。通过分析可以发现，sleeplock 内部使用了一个 spinlock 来实现互斥，其结构如下：

```
1  struct sleeplock {
2    uint locked;       // Is the lock held?
3    struct spinlock lk; // spinlock protecting this sleep lock
4
5    // For debugging:
6    char *name;        // Name of lock.
7    int pid;           // Process holding lock
8  };
```

在尝试获取锁的部分 acquiresleep 函数中，首先尝试获取内部自旋锁 lk，如果当前 lk 已经上锁，则调用 sleep 函数阻塞等待。否则将 lk->locked 设置为 1，并且对 lk->pid 进行赋值，完成后直接释放内部的自旋锁。其具体实现如下：

```
1  void acquiresleep(struct sleeplock *lk) {
2    acquire(&lk->lk);
3    while (lk->locked) {
4      sleep(lk, &lk->lk);
5    }
6    lk->locked = 1;
7    lk->pid = myproc()->pid;
8    release(&lk->lk);
9  }
```

```
1  void releasesleep(struct sleeplock *lk) {
2    acquire(&lk->lk);
3    lk->locked = 0;
4    lk->pid = 0;
```

```
5    wakeup(lk);
6    release(&lk->lk);
7  }
```

可以看到，由于有了内部自旋锁的保护，对 sleeplock 中的 locked 变量的操作不需要再通过汇编指令来完成，只需要在操作前后加锁以及释放锁即可。

参考文献

[1]王道考研《2020 操作系统考研指导》第 2 章进程同步，P77-103.
[2]维基百科，竞争状态(https://en.wikipedia.org/wiki/Race_condition)
[3]维基百科，自旋锁(https://zh.wikipedia.org/wiki/自旋锁/)
[4]线程同步之详解自旋锁(https://www.cnblogs.com/cposture/p/SpinLock.html)
[5]汇编指令 CLI/STI (https://blog.csdn.net/zang141588761/article/details/52325106)
[6] X86 PUSHF/PUSHFD/PUSHFQ 指令详解
(https://blog.csdn.net/ross1206/article/details/72833084)
[7]维基百科，Kernel Panic(https://en.wikipedia.org/wiki/Kernel_panic)
[8]xv6 解析-- 多处理器操作(https://www.cnblogs.com/Dream-Chaser/p/9158917.html)
[9]原子操作与 x86 上的 lock 指令前缀
(https://blog.csdn.net/zacklin/article/details/7445442)
[10]百度百科 XCHG(https://baike.baidu.com/item/XCHG/1589577)
[11] XV6 操作系统代码阅读心得（三）：锁
(https://www.cnblogs.com/hehao98/p/10678493.html)
[12]维基百科，读写锁(https://zh.wikipedia.org/wiki/读写锁/)
[13]x86 常用汇编寄存器
(https://blog.csdn.net/qq_24423085/article/details/103835522)
[14] GCC 内联汇编基础(https://www.jianshu.com/p/1782e14a0766)
[15] 线程同步之利器(1)——可递归锁与非递归锁
(https://blog.csdn.net/zouxinfox/article/details/5838861)
[16] linux: 读写锁(https://blog.csdn.net/dangzhangjing97/article/details/80368822)
[17] 一步一步实现读写锁(https://cloud.tencent.com/developer/article/1021461)