Chapter6 Locking

还记得在第一章中我们提过操作系统关注的三大部分吗？是的，关于虚拟化，我们已经讨论了内存虚拟化和一部分CPU虚拟化，CPU虚拟化的另一重要部分——调度，我们将在之后讨论。关于持久性的内容也会在以后介绍。在这一章中，我们将重点关注**并发**。

正如大多数操作系统内核一样，xv6可以交错地同时执行多个活动。硬件方面，我们拥有多个CPU，这些CPU各自独立地执行指令，并且共用RAM（实际上，每个CPU都有cache，但从概念上看多处理器系统还是使用单一的，共享的RAM），因此我们可以在多个CPU之间维护一些**共享数据结构**。这会导致一些问题，例如在一个CPU读取数据时，同时另一个CPU却在更新该数据。如果我们不进行一些干预的话，最后会导致读出错误数据，或者损坏该数据结构。

即使是单CPU，也会在多个进程之间切换执行，使这些进程的执行序列变得交错。特别是，当中断开放时，如果某中断处理程序正在修改某些数据，而进行到一半时又被新的中断介入，该数据就会出现问题。因此，共享数据结构在多CPU或单CPU的情况下，都可能会出现问题。

以上这些都是我们需要考虑的**并发Concurrency**问题，它通常由于**多个执行指令流交错**而产生，更进一步可以归结为以下原因：**多处理器的并行执行**，**线程切换**，**中断**。

内核之中总是遍布着各种共享数据结构。例如，多个CPU可以同时地调用`kalloc`，并发地从free-list上取出第一页。我们在设计内核时，经常允许这种**大量并发**的情况发生，通过并行性可以提高性能和响应速度，然而我们需要花费较多的精力提供**正确性**。

我们希望有一种兼具并发性和正确性的抽象设计，线程应该使用这种设计提供的**互斥Mutual Exclusion**原语，使多个执行线程可以同时进入临界区，避免产生竞争条件，最终能够产生确定的输出。为此，设计者们提供了很多**并发控制Concurrency Control技术**，或者我个人更喜欢这个名字，**同步工具Synchronization Tools**。

常用的同步工具有：**锁Lock**，**条件变量Conditional Variable**，**信号量Semaphore**等。

这些同步工具，保证了临界区中工作的**原子性atomic**，换句话说就是**All-or-Nothing**。

在这一章里，我们将精力集中在**锁**上，在下一章中我们会补充更多同步工具的相关内容。

锁是一种互斥原语，**一次只有一个CPU能够获得锁**。因此，如果我们给一个临界区上锁，那么一次只能有一个CPU在临界区中执行，锁成功地保护了临界区中的共享数据结构。尽管使用锁可以解决很多的并发问题，但如果使用不当的话，会影响性能表现，因为锁本质上使并发操作变得**串行化/序列化Serialized**，当然也会有多个CPU**争夺Content**同一把锁。

我们不妨用三句简单的话（Frans Kaashoek教授在课上总结的）来解释锁的作用。

- 使用锁有助于避免更新的丢失。（**Locks help avoiding lost updates.**）
- 使用锁使多步操作变为原子性的操作。（**Locks make multi-step operations atomic.**）
- 使用锁有助于维持一些规则或特性的不变性。（**Locks help maintain invariant.**）

其中这个invariant是指对于某个数据结构必须要维持的性质，比如链表就要维持list指向头元素，每个元素指向下一个元素，锁需要用来保证数据结构在**不同操作之间**，都保持这个特性。

Race Conditions and Critical Sections

访问共享资源的代码段，我们称之为**临界区Critical Section**，而多个执行线程同时进入临界区并尝试更新共享数据结构的情况，我们称之为**竞争条件Race Condition**。

多个进程并发地修改某个共享数据结构，并因此产生不确定的结果，这就是**竞争条件**。而这些进程并发地修改共享数据结构的代码段，我们就称为**临界区**，如下图所示

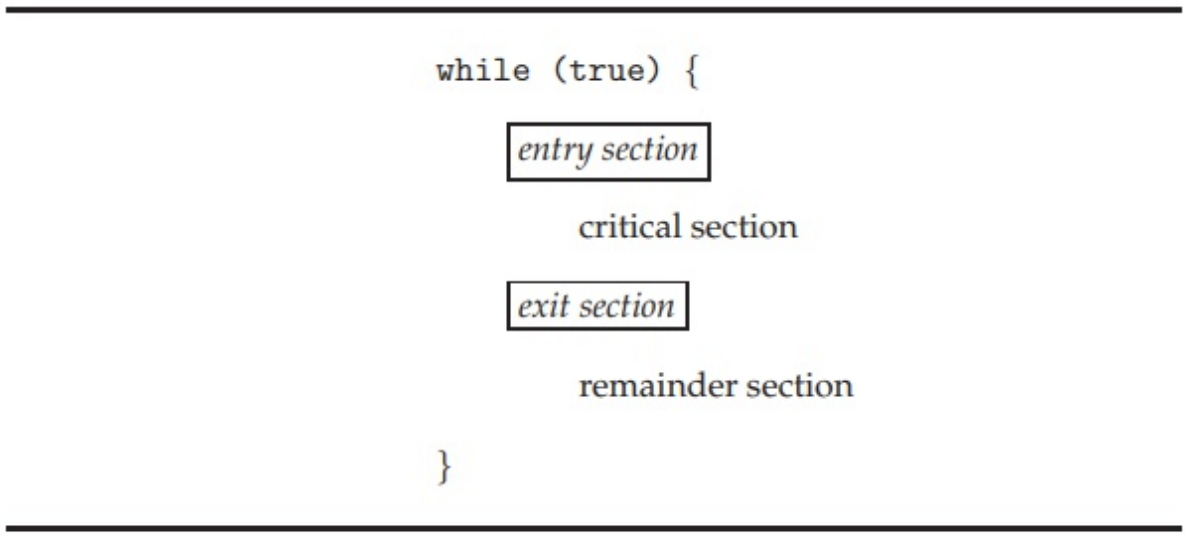

Figure 6.1 General structure of a typical process.

要解决临界区问题，其中一种方法就是使用同步工具里的**互斥锁Mutex Lock**，或者简称为锁。使用锁的解决方案应该要满足以下的要求。

- **互斥**：这是最基本的要求，**一次只能有一个进程进入临界区**中，并在其中执行。
- **进步**：如果当前没有进程在临界区中执行，且有多个进程同时想要进入临界区，应该有某种方式来决定谁先进入，而且这种**决定不能被无限推迟**（**死锁Deadlock**或**活锁Livelock**）。
- **有限等待/无饥饿**：进程从请求进入临界区开始，直到被允许进入临界区，这段时间应该是有限的。换句话说，每个竞争进程都应该有**公平**的机会抢到锁，不能有竞争锁的进程处于**饥饿Starvation**，一直无法获得锁。

有两种用于处理操作系统临界区问题的常用方法：**抢占式内核Preemptive Kernels**和**非抢占式内核Nonpreemptive Kernels**。抢占式内核允许在内核空间下运行的进程被抢占，不断通过时钟中断一个线程，运行其它线程；而非抢占式内核则不允许，进程会一直运行直到退出内核空间、阻塞或自愿放弃CPU。

非抢占式内核基本不会导致竞争条件，因为任一时刻只有一个进程运行；而抢占式内核则需要仔细设计，以防止竞争条件出现，尤其是对称多处理器体系结构。但我们更倾向于使用抢占式内核，因为这种内核响应更快，进程不会在内核中运行任意长的时间，还允许实时进程抢占在内核空间下运行的其它进程，因此我们的后面讨论主要基于抢占式内核。

Hardware Support

解决临界区问题的软件方案不能保证在现代计算机体系结构上正常工作。但是只需要很少的**硬件支持**，实现锁就会容易很多。下面介绍一些硬件方法，其中涉及到的硬件指令都是**原子的**。

Test and set

这个特殊硬件指令叫做test-and-set，逻辑如下

```
int TestAndSet(int *old_ptr, int new){
  int old = *old_ptr;  // 获取旧的值
  *old_ptr = new;      // 存储新的值
  return old;          // 返回旧的值
}
```

它返回old_ptr指向的旧值old，同时更新成新值new，利用这个特性，我们可以按以下方式实现一个简单的锁。如果当前线程不能获取该锁，它就在原地上等待，不断消耗CPU周期，这种锁叫做**自旋锁 Spinning Lock**。这是一个简单的解决方案，但是不适合于采用非抢占式内核的单处理器系统，因为一个自旋的线程永远不会放弃CPU。

```
typedef struct lock_t{
  int flag;
} lock_t;

void init(lock_t *lock){
  lock->flag = 0;  // 0表示锁可用，1表示锁已被占用
}

void lock(lock_t *lock){
  while(TestAndSet(&lock->flag, 1) == 1)  // 第一个执行TestAndSet的线程会得到旧值0而跳出循环
    ;  // 不做任何事情，只是自旋地等待
}

void unlock(lock_t *lock){
  lock->flag = 0;
}

void main()
{
  init(&lock_t);

  do{
    lock(&lock_t);

    // 临界区

    unlock(&lock_t);
  }while(true);

  return;
}
```

自旋锁解决方案满足互斥和进步，但是**不满足有限等待**。一种可能的改进方法如下所示，它满足了临界区问题的三个要求。同样地，这段代码也不够严谨，只是希望你能理解它的设计思想。你可以验证，希望进入临界区的线程，现在至多只需要等待n-1次。

```
typedef struct lock_t{
  int flag;
} lock_t;

int waiting[n];  // waiting[i] = 1 时表示i号线程想要进入临界区

void init(lock_t *lock){
  lock->flag = 0;  // 0表示锁可用，1表示锁已被占用
  for(int i=0;i<n;i++)
    waiting[i] = 0;
}

void lock(lock_t *lock){
  waiting[i] = 1;  // i是当前线程的编号，表示i号线程想要进入临界区，因此先等待
  int key = 1;
  while(waiting[i] && key)  // 只有当waiting[i] = 0 或 key = 0 时i号线程才能进入临界区
    key = TestAndSet(&lock->flag, 1);  // 第一个执行TestAndSet的线程会使key变为0
  waiting[i] = 0;  // 表示i号线程已经进入临界区，不用再自旋等待
}

void unlock(lock_t *lock){
  j = (i + 1) % n;  // 尝试把机会让给下一个线程
  while((j != i) && !waiting[j])
    j = (j + 1) % n;  // 轮询一圈，按编号顺序搜索，找到需要进入临界区的下一个线程
  if(j == i)
    lock->flag = 0;  // 当前没有其它线程需要进入临界区，直接释放锁
  else
    waiting[j] = 0;  // j号线程想要进入临界区，于是使其不再等待
                     // waiting[j] = 0 表示j号线程可以进入
                     // 而lock->flag = 1 仍成立，表示锁仍被持有，只是从i转移到j
}

void main()
{
  init(&lock_t);

  do{
    lock(&lock_t);

    // 临界区

    unlock(&lock_t);
  }while(true);

  return;
}
```

Compare and Swap

这个特殊硬件指令叫做compare-and-swap，逻辑如下：

```
int CompareAndSwap(int *ptr, int expected, int new){
  int actual = *ptr;
  if(actual == expected)
    *ptr = new;
  return actual;
}
```

思路就是检测ptr指向的值是否和expected相等。如果是，就更新ptr的值为new，否则什么也不更新。无论如何，最后都返回该内存地址的实际值actual。

和前面利用test-and-set指令实现互斥锁的思路一样，我们只需要替换lock函数如下。实现有限等待的方法也是类似的，把用test-and-set的地方改为用compare-and-swap来实现即可。

```
void lock(lock_t *lock){
  while(CompareAndSwap(&lock->flag, 0, 1) == 1)
    ;
}
```

Fetch and Add

这个特殊硬件指令叫做fetch-and-add，逻辑如下：

```
int FetchAndAdd(int *ptr){
  int old = *ptr;
  *ptr = old + 1;
  return old;
}
```

指令让特定地址的值自增1，并且返回旧值，整个过程是原子的。这一点和test-and-set很相似，但是我们现在可以用fetch-and-add指令实现一个更有趣的**ticket**锁。这个ticket锁也可以满足临界区问题的三个要求，互斥、不会死锁、不会饥饿。

```
typedef struct lock_t{
  int ticket;
  int turn;
} lock_t;

void lock_init(lock_t *lock){
  lock->ticket = 0;
  lock->turn = 0;
}

void lock(lock_t *lock){
  int myturn = FetchAndAdd(&lock->ticket);
  while(lock->turn != myturn)
    ; // 自旋等待
}

void unlock(lock_t *lock){
```

```
  FetchAndAdd(&lock->turn);
}

void main()
{
  lock_init(&lock_t);

  do{
    lock(&lock_t);

    // 临界区

    unlock(&lock_t);
  }while(true);

  return;
}
```

Load-Linked and Store-Conditional

在MIPS架构中，提供了**链接加载Load-Linked**和**条件式存储Store-Conditional**指令，可以配合使用，以实现很多并发结构。这两条指令的逻辑，用C代码和伪代码结合的形式展示，如下所示。

```
int LoadLinked(int *ptr){
  return *ptr;
}

int StoreConditional(int *ptr, int value){
  if(在上一次加载ptr之后，期间没有对ptr的更新){
    *ptr = value;
    return 1;  // 成功
  }else{
    return 0;  // 失败
  }
}
```

Load-Linked和典型加载指令类似，从内存中取出值存入一个寄存器。而Store-Conditional比较特别，只有上一次加载的地址ptr，在这期间都没有被更新时，它才会返回1表示成功，同时更新经过Load-Linked得到的ptr中的值；如果失败，返回0，且不会更新ptr中的值。

使用Load-Linked和Store-Conditional实现一个锁的方案，如下所示。

```
void lock(lock_t *lock){
  while(1){
    while(LoadLinked(&lock->flag) == 1)
      ;  // 自旋等待
    if(StoreConditional(&lock->flag, 1) == 1)
      return;  // 如果成功把flag置1，则可以进入临界区
               // 否则，反复尝试
  }
}

void unlock(lock_t *lock){
  lock->flag = 0;
}
```

首先，一个线程自旋等待flag被置0，这表明锁不被任何人持有。一旦如此，线程就尝试通过Store-Conditional来获取锁。如果成功，flag也会被更改为1，然后线程可以进入临界区。

Store-Conditional何时会失败？考虑以下情况，一个线程调用lock，执行了Load-Linked并返回0，但在执行Store-Conditional之前，发生了时钟中断，另一个线程也调用lock，同样执行Load-Linked，因为锁还没有被获取，所以同样返回0。两个线程都执行了Load-Linked，而且都将要执行Store-Conditional，但只有一个线程会成功更新flag为1并获得锁，第二个线程会失败，因为在这期间flag已经被更新了，所以它需要重新尝试获取锁。

一种等价的，更为简洁的lock实现如下，你可以验证这和上面的代码是等价的。

```
void lock(lock_t *lock){
  while(LoadLinked(&lock->flag) || !StoreConditional(&lock->flag, 1))
    ;
}
```

Memory Barriers

Peterson算法在现代计算机上不能正确地执行，有一部分原因是代码中的存在一些没有直接关联的语句，**编译器**可能会优化它们，重新组织它们的顺序，这很可能将导致Peterson算法不能正确运行。

计算机体系结构将确定它为应用程序提供什么样的内存保证，这被称为**内存模型Memroy Model**。一般来说，内存模型可以分为两类：

- 严格一致性/强一致性（**Strongly Ordered**）：一个CPU对内存做了某些修改，其它CPU马上就能看到这项更新。
- 松散一致性/弱一致性（**Weakly Ordered**）：一个CPU对内存做了某些修改，但其它CPU不能马上，而可能在稍后才会看见这项更新。

由于内存模型的类型和CPU密切相关，在一个共享RAM的多处理器系统中，我们的内核不能对这些修改的可见性做任何假设。为此，计算机体系结构提供了一种指令，它可以强制内存的更新对所有CPU可见，这种指令就是**内存屏障Memory Barriers**（或Memory Fences）。

当执行内存屏障指令时，系统确保，在内存屏障**之前**所有的load和store，都会在内存屏障**之后**的load或store执行前完成。因此，即使指令被重新排序，内存屏障确保了，位于它之前的存储操作已经在内存中完成，因此在内存屏障之后，这些存储操作对所有处理器是可见的。

可以利用内存屏障来确保Peterson算法的正确运行了。

```
// 现在保证首先加载flag的值，然后再加载x的值
while (!flag)
  memory barrier();
print x;

// 现在保证x首先被赋值，然后再到flag被赋值
x = 100;
memory barrier();
flag = true;
```

Atomic Variables

前面的几种硬件指令，诸如test-and-set、compare-and-swap，它们并不能直接提供互斥机制，我们是利用了这些指令作为基本组件来构建同步工具的，这些指令是原子的。现在，我们将介绍另一种原子的硬件支持，不过这次是变量类型，我们称为**原子变量Atomic Variables**。

原子变量可以提供互斥机制，保证在**更新该变量时不会出现竞争条件**。

很多支持原子变量的操作系统都提供特殊的原子变量数据类型，以及访问和操作它们的函数。这些函数经常用你已经熟知的compare-and-swap来实现，下面是其中一个例子。

```
void increment(atomic int *v)
{
  int temp;
  do {
    temp = *v;
  }
  while (temp != compare and swap(v, temp, temp+1));
}
```

值得一提的是，尽管原子变量提供**原子性的更新**，从而避免了在更新变量时出现竞争条件，但是却无法避免其它竞争条件的出现。还是回到我们一开始举的有界缓冲区的例子，现在我们可以用一个原子变量代替count，count的更新就是原子性的。但是对于生产者和消费者进程的while循环，其条件判断也是取决于count的。考虑一种情况，当前缓冲区为空，因此有两个消费者进程会不断循环，等待count>0。如果现在有生产者往缓冲区中写入一项内容，那么count=1，因此两个消费者进程都有机会跳出while循环，并同时进入临界区中，但我们的缓冲区中只有1项内容可读，因此错误就会发生。

因此，想要能够处理更普遍情况下的竞争条件，我们最好还是使用一些健壮性更强的同步工具。在前面部分中，用硬件指令构造的自旋锁就是一个不错的解决方案。

Spinning Lock

对于锁我们一般提供两个接口，一个是`acquire`用于获取锁，一个是`release`用于释放锁。

```
acquire(){
  while(!available)
    ;  // busy wait (i.e. spinning)
  available = false;
}

release(){
  available = true;
}
```

我们再看看xv6里面自旋锁的实现（kernel/spinlock.c），我们可以看到其中利用了RISC-V的test_and_set指令，利用了RISC-V的内存屏障指令，还结合了中断的控制（马上我们将讨论中断对锁的影响），实现了一个可用的自旋锁。

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

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
};

// Acquire the lock.
// Loops (spins) until the lock is acquired.
void
acquire(struct spinlock *lk)
{
  push_off(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // On RISC-V, sync_lock_test_and_set turns into an atomic swap:
  //   a5 = 1
  //   s1 = &lk->locked
  //   amoswap.w.aq a5, a5, (s1)
  while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
    ;

  // 即 memory barrior
  // it tells the compiler and CPU to not reorder loads or stores across the
barrier
  __sync_synchronize();

  // Record info about lock acquisition for holding() and debugging.
  lk->cpu = mycpu();
}

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

  lk->cpu = 0;

  // On RISC-V, this emits a fence instruction.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code doesn't use a C assignment, since the C standard
  // implies that an assignment might be implemented with
  // multiple store instructions.
  // On RISC-V, sync_lock_release turns into an atomic swap:
```

```
  //   s1 = &lk->locked
  //   amoswap.w zero, zero, (s1)
  __sync_lock_release(&lk->locked);

  pop_off();
}
```

前面我们已经提过，自旋锁**不满足有限等待的要求，可能会导致某些线程饥饿。**然后关注性能，自旋锁的主要缺点是，它会**忙等待**，这种忙等会持续消耗CPU资源，因为它要不停地旋转以等待锁可用。

考虑单CPU的情况，性能开销相当大，假设一个线程持有锁进入临界区，然后因时钟中断被抢占，调度器因此运行其它每一个线程，而其它线程都在竞争该锁，因此每个线程都会在放弃CPU之前，自旋一整个时间片，从而浪费了大量CPU周期。不过，在多CPU上，尤其是当线程数大于CPU数时，自旋锁的性能表现不错。假设线程A在CPU1上，线程B在CPU2上，都竞争同一个锁。A占有锁时，B会在CPU2上自旋，而临界区一般很短，所以B很快就获得锁。因此，自旋等待其它CPU上的锁时，不会浪费很多CPU周期。

另外，线程在等待锁时，没有上下文切换（尤其是上下文切换的开销可能比较大），因此如果临界区较短，使用自旋锁也是一个不错的方案。

Locks and Interrupt

最早的互斥解决方案之一，是为单CPU系统开发的方法，在临界区中关闭中断。假设我们运行在单CPU系统上，进入临界区之前关闭中断，可以保证临界区内的代码原子性地执行，离开临界区之后我们再重新开放中断。

这个方法的优点是简单，但遗憾的是缺点很多，下面列出这个方案的缺点。

- 开关中断是特权指令，这要求我们给予用户线程高特权级别，即要求我们信任这些线程。显然，如果我们必须信任所有的程序，问题就会接踵而至。例如，一个线程可以随意地关闭中断，然后占用CPU；或者线程关闭中断后，不幸陷入死循环，但控制器无法返回到内核中。
- 这种方案不适用于多处理器。如果多个线程运行在不同CPU上，每个线程都试图进入同一个临界区，那么关闭中断也是没有作用的，因为线程可以运行在其它CPU上，同样可以进入同一个临界区。再者多处理器系统已经很普遍，所以我们需要支持多处理器的解决方案。
- 关闭中断可能将导致中断丢失，这可能会导致严重的系统问题。例如磁盘设备完成了读取请求，但CPU错失了这一消息，那么之后内核如何知道区唤醒等待读取的进程？
- 还有一个不太重要的原因是，相比于其它常规指令，开关中断指令执行较慢。

还有一种情况需要注意，这种情况可以在xv6中看到，就是当线程和中断处理程序要获取同一把自旋锁的时候。xv6中的一个例子是，时钟中断处理程序`clockintr`需要增加ticks值，而一个内核线程也可能通过系统调用`sys_sleep`来访问ticks值，因此，我们为ticks维护了一把锁tickslock，`clockintr`和`sys_sleep`都需要获取该自旋锁来访问或修改ticks值。

但是，自旋锁和中断处理程序出现在一起时，可能会导致一些问题。现在假设，一个内核线程调用`sys_sleep`，因此它持有tickslock。此时运行该内核线程的CPU收到时钟中断，因此被导向到中断处理程序`clockintr`中，`clockintr`第一件事就是acquire(&tickslock)，但是因为这把锁已经被内核线程持有，所以`clockintr`会一直在原地自旋等待tickslock被释放。

糟糕的事情发生了，tickslock无法被该内核线程释放！因为中断处理程序`clockintr`还没有返回！因此该CPU发生**死锁Deadlock**，其它试图获取tickslock的CPU也会接二连三地死锁。

你可能会问，为什么这个持有自旋锁的内核线程不能被其它CPU调度运行，从而有释放锁的可能？首先，对于单CPU系统，显然`clockintr`会一直原地自旋占用CPU，如果允许中断嵌套的话，我们只是运行了新的中断处理程序或者新的`clockintr`，最后无论如何都会死锁在`clockintr`上。然后，对于多CPU系统，其它CPU也无法选择该持有锁的内核线程调度运行。为什么呢？这要从内核的调度器的工作方式进行说明。内核调度器选择那些状态为RUNNABLE的线程调度执行，而对于被中断的线程，由于yield在`clockintr`返回之后才被调用，因此该线程的状态还是RUNNING而不是RUNNABLE，自然就被其它的内核调度器无视了。

因此，为了避免在这种情况下发生死锁，我们应该做以下规定：

> 如果中断处理程序需要持有某一把**自旋锁**，那么每个CPU在持有这把自旋锁时，一定要**保持中断关闭**。

xv6的解决方式更加保守一些：只要CPU试图获取任何自旋锁，那么该CPU总是会关闭中断。所以，你在前面xv6的acquire实现中，可以看到进入该函数的第一件事就是关闭中断。中断仍然可以在其它CPU上产生，所以一个中断处理程序的acquire可以等待别的线程释放相应的锁，只是该线程将在不同的CPU上运行。

最后，你应该注意到，**在xv6的实现中，一旦某个线程获取了自旋锁，那么在该CPU上将不会产生中断（因此不会发生线程调度）**，因为在acquire中已经关闭了该CPU的中断。

Sleep Lock

到目前为止，我们讨论的互斥锁都是自旋锁，自旋锁的优缺点我们也在前面已经讨论过。我们希望，如果一把锁在长时间内都被其它进程持有，那么尝试获取该锁的其它线程不必占用大量的CPU周期，而是以一种**睡眠**的方式**主动放弃CPU**，在稍后的某个时间再被重新**唤醒**执行。

遗憾的是，自旋锁并不支持这种特性。首先，自旋锁会浪费大量的CPU周期；其次，**已经持有自旋锁的进程不应该主动放弃CPU，**这可能会导致**死锁**，因为acquire()是原子操作（中断关闭），则当存在第二个线程尝试获取已被第一个进程持有的自旋锁时，会导致无限循环并且无法调度，所以应当遵循进程持有自旋锁时中断应保持关闭这一原则。

为此，我们重新设计另一种类型的锁。这种类型的锁，会在acquire需要自旋等待时让出CPU；同时，在持有这种锁时，允许主动放弃CPU和开放中断。

这种类型的锁，我们称为**睡眠锁Sleep Lock**。

先从一个简单的解决方案开始，一个简单的形式化定义如下所示。

```
void init(){
  flag = 0;
}

void lock(){
  while(TestAndSet(&flag, 1) == 1)
    yield();  // 主动放弃CPU
}

void unlock(){
  flag = 0;
}
```

我们假设内核提供**yield**原语，进程可以调用它主动放弃CPU（事实上xv6真的提供了这个原语，我们将在调度的章节里介绍它）。

进程的运行状态可以按下面的enum来定义，我们关注RUNNING和RUNNABLE两种状态。yield让进程的状态从RUNNING变为RUNNABLE，并让调度器寻找其它状态为RUNNABLE的进程允许，让出CPU的本质就是进程**取消调度deschedule**了自己。

```
enum procstate { UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

但是，这个解决方案还不够完善。回忆一下我们临界区问题的三个要求，对于有限等待条件，该解决方案还不能明确地支持它，原因是该方案还存在很多偶然性，例如调度程序进行调度的合理性。无论是一直自旋，还是立刻主动让出CPU，都可能造成浪费，也不能防止饥饿。

所以，我们需要显式地增加某种机制，决定锁释放时，谁能抢到锁。为此需要操作系统提供更多的支持，使用队列来保存等待锁的进程是常用的解决方案。改进实现方法如下。

```
// 在这里，有两个lock，flag和guard
typedef struct lock_t{
  int flag;
  int guard;
  queue_t *q;
} lock_t;

void lock_init(lock_t *m){
  m->flag = 0;
  m->guard = 0;
  queue_init(m->q);
}

void lock(lock_t *m){
  while(TestAndSet(&m->guard, 1) == 1)
    ;  // 自旋地等待直到获取guard lock
  if(m->flag == 0){
    m->flag = 1;   // flag lock还没有被获取，因此成功获取
    m->guard = 0;  // 释放guard lock
  }else{
    queue_add(m->q, gettid());  // flag lock已经被获取，该线程将进入队列中睡眠
    m->guard = 0;  // 睡眠之前释放guard lock
    park();        // 睡眠
  }
}

void unlock(lock_t *m){
  while(TestAndSet(&m->guard, 1) == 1)
    ;  // 自旋地等待直到获取guard lock
  if(queue_empty(m->q))
    m->flag = 0;  //没有人要获取flag lock，直接释放它
  else
    unpark(queue_remove(m->q));  // 从队列里唤醒一个线程，把flag lock交给它
  m->guard = 0;  // 结束之前释放guard lock
}
```

在这个解决方案里，我们引入了队列，保证了公平性，现在这个睡眠锁的设计满足有限等待条件。以下是我们需要关注的一些点：

- 我们实际上使用了**两把锁**来实现这个睡眠锁，一个是自旋锁guard lock（小锁），另一个锁是真正的临界区锁flag lock（大锁）。因此，这个方法并没有完全避免自旋等待，但是，这个自旋等待的

时间很有限，因为guard lock不是保护真正的临界区，只是保护在lock和unlock中的几条指令。
- 我们在调用park睡眠之前，要先释放这把小的自旋锁guard lock，否则会出现问题。
- 唤醒另一个线程的时候flag仍然保持为1，尽管它并非持有guard lock。从逻辑上看，没有持有guard lock时是不能将flag置1的，但我们直接保持flag为1，并唤醒这个线程，相当于直接把flag lock传递给它。

最后，这个解决方案还有一个潜在的竞争条件。例如，thread1已经持有了flag lock，后续thread2尝试获取flag lock时就会陷入lock( )的else分支中，将自己添加到队列之后，但在调用park( )之前被切换到thread1，thread1执行unlock时，由于队列不为空，将会执行unpark(thread2)，但thread2的park( )并未执行，unpark( )可能会返回thread2并未睡眠的消息，然后thread1退出临界区，thread2再次执行并park( )自身，进入睡眠。由于thread1始终没有将flag lock重置为0，因此后续尝试获取该锁的thread3、thread4...都将与thread2一并调用park( )进入睡眠中，陷入一种死锁状态，这种情况有时也称为**唤醒/等待竞争wakeup/waiting race。**

Solaris增加了第三个系统调用`setpark`来解决这种情况，一个线程通过调用`setpark`表明自己马上要park，此时如果刚好另一个线程被调度，并且调用了unpark，那么后续的park调用就会直接返回，而不是一直睡眠。或者，另外一种方案是把guard传入内核，内核可以采取一些预防措施，保证原子地释放锁，并把运行进程移出队列。

我们来看看xv6的睡眠锁实现（kernel/sleeplock.c），你可以看见，它也采用了自旋锁和真正的睡眠锁相结合的方式。**xv6的睡眠锁实现，允许中断开放，因此中断处理程序不能使用睡眠锁**（同样会导致死锁）。再者，因为睡眠锁会让出CPU，所以**不能在自旋锁保护的临界区中使用睡眠锁**，但是可以在睡眠锁保护的临界区中使用自旋锁。

```
// Long-term locks for processes
struct sleeplock {
  uint locked;       // Is the lock held?
  struct spinlock lk; // spinlock protecting this sleep lock

  // For debugging:
  char *name;        // Name of lock.
  int pid;           // Process holding lock
};

void
acquiresleep(struct sleeplock *lk)
{
  // 第一个进程先抓取小锁,将小锁的locked置1
  // 这时大锁还未被上锁(lk->locked=0),跳过while继续执行
  // 将大锁也抓取,将大锁的locked置1
  // 最后释放小锁,这时大锁在第一个进程手中
  // 后续的进程进来之后可以抓到小锁
  // 但是因为大锁被抓,lk->locked=1,则进到while中
  // 调用sleep(),释放小锁并且挂起进程在大锁上
  // 只有调用releasesleep(),才会释放大锁,同时唤醒挂起在大锁上的进程
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}
```

```
void
releasesleep(struct sleeplock *lk)
{
  // 要释放大锁的进程先获取小锁
  // 将lk->lk->locked置0,同时调用wakeup()唤醒一个挂起在大锁上的进程
  // 选中一个进程改变其状态为RUNNABLE之后,回来释放小锁
  // 被唤醒的进程会从sleep()中sched()后的对应位置继续,获取小锁
  // 然后,从上次acquiresleep()的while中继续,此时lk->locked=0,跳出while
  // 然后将lk->locked置1,最后释放小锁,这样上次挂起的进程就抓到了大锁
  // 这里注意,在sleep()中需要使用acquire()去抓取小锁
  // 因为要保证该进程总能抓到这把锁,即使该进程要空转,这是为了防止死锁
  // 这样唤醒进程发现小锁被抓走时,会在该处自旋,等到小锁被重新放出为止
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

睡眠锁的优点显然是避免使用自旋锁时，带来的大量CPU周期的浪费；其缺点是，来回地切换进程会导致较高的上下文切换开销，例如100个线程反复竞争一把睡眠锁，若一个线程持有锁，在释放锁之前被抢占，其它99个线程分别尝试获取睡眠锁，发现被抢占后，让出CPU，引起上下文切换。如果使用某种轮转调度程序，这99个线程会一直处于运行—让出的模式，直到持有锁的线程再次运行。虽然这比原来使用自旋锁，浪费99个完整的时间片要好，但是99次上下文切换的成本也是惊人的，浪费仍然很大。

根据两种锁的特点，在临界区较短的时候，使用自旋锁比较合适；而在临界区比较长的时候，就应该使用睡眠锁。也可以将两种锁结合，称为**两阶段锁**，第一阶段先自旋一段时间，希望可以获取锁，但如果没有获得，那么在第二阶段调用者会睡眠，直到锁可用。Linux就采用了这种锁，不过只自旋一次，更常见的方式是在循环中自旋固定的次数，然后再睡眠。

在下一章我们将学习sleep和wakeup，那么对一个自旋锁调用sleep，和直接使用一个睡眠锁有何不同？我认为，这两种方式都是为了在某个条件不满足时，将自身挂起并放弃CPU。以xv6为例，不同点在于，对自旋锁调用sleep，可以用在任何条件不满足的情况（例如缓冲区为空、缓冲区已满、UART发送未完成等），且中断处理程序也可以使用这种方法（因为获取自旋锁的同时会关闭中断）；而睡眠锁条件不满足的情况是特定的，即不能获取锁，因此主动让出CPU，但是中断处理程序不能使用睡眠锁，因为**在睡眠锁保护的临界区内，中断并不关闭。**

Contention on the Locks

到此为止，我们已经介绍完了互斥锁，利用互斥锁，我们确实解决了大部分的并发问题，现在我们的临界区总是能正确运行；但是使用锁确实会使并发问题变得串行化，一次只能有一个进程在临界区中运行，从而影响性能。有趣的是，我们因为使用了多CPU和多线程而产生并发问题，而为了解决并发问题，我们使用了锁，这又使得多CPU的性能优势衰减，经常出现多个进程争夺锁的情况，我们称这种现象为锁的**争夺Contention。**

内核设计中，一个重要的部分就是如何避免锁的争夺。xv6对此只做了很简单的工作，复杂的内核会更细致地组织它们的数据结构和相应算法，从而尽最大可能避免锁的争夺。

怎么减少锁的争夺？以xv6为例子，xv6的所有的锁如下图所示。

Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids intermixed output
ftable.lock	Serializes allocation of a struct file in file table
icache.lock	Protects allocation of inode cache entries
vdisk_lock	Serializes access to disk hardware and queue of DMA descriptors
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's pi->lock	Serializes operations on each pipe
pid_lock	Serializes increments of next_pid
proc's p->lock	Serializes changes to process's state
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

Figure 6.3: Locks in xv6

内核分配器的free-list只用了一把锁保护，因此多个CPU想要分配和释放物理页时，都需要在争夺同一把锁。一个可能的解决方案是，我们为每个CPU都维护一个free-list，因此锁的数量从原来的一把，增加到和CPU的数量相等，这样CPU大部分时间下都不用和其它CPU去争夺锁，而只需要持有自己的锁，并访问自己的free-list。只有在自己的free-list用完了之后，才去和其它CPU争夺别人的锁，然后"偷"一些物理页到自己的free-list上。可以验证，这种方式大量减少了锁的争夺。同样地，对于只有一把锁的缓冲区buffer cache，我们也可以用类似的思路，将缓冲区的空间分配到多个哈希桶中，因此每个哈希桶一把锁，并根据块号来索引哈希桶，从而减少了在缓冲区上的锁的争夺。

因此这指示我们在实际设计并发代码时，我们可以先采用**粗粒度Coarse-grained**的上锁方式，对要保护的临界区上一把大锁。例如前面提到的内核分配器就是一个很好的例子。然后我们在保证并发正确性的情况下，逐步地拆解这些数据结构，并且使用**细粒度Fine-grained**的上锁方式。例如xv6对于每个文件都单独维护一把锁，因此修改不同文件的不同进程，不需要在锁的争夺上烦恼。甚至，如果你想再细致一点，你还可以让多个进程同时修改文件，因此使用更细粒度的锁。总之，采用何种粒度的上锁方式，取决于你对性能和代码的复杂性的要求。

Deadlock and Lock Ordering

在前面的讨论中，我们发现某些情况下会出现**死锁**。进程申请资源时，如果没有可用资源，那么该进程会进入等待状态，但如果所申请的资源被其它的等待进程占有，那么该进程有可能再也无法改变状态。有关于死锁的更详细的讨论，我们将在下一章中展开。

我们前面已经提到了两种可能引起死锁的情况：

- 中断处理程序尝试获取线程已经持有的自旋锁。
- 持有自旋锁的进程主动放弃CPU。

现在我们再补充一种，这种死锁的出现和**上锁的顺序Lock Ordering**有关。

一个最简单的例子，进程1获取锁的顺序是先A后B，进程2则是先B后A。如果进程1获得A的同时，进程2也获得了B，那么死锁就发生了。

事实上，这在内核中很常见，因为修改某些数据结构可能会要求你同时持有多个锁，因此上锁的顺序很重要，稍有不慎就会导致死锁。避免这种死锁的解决方案看上去也很简单，只需要规定一个**全局的上锁顺序**即可。规定全局的上锁顺序，意味着锁实际上是每个函数规范的一部分：调用者必须以一种规定的顺序调用函数，以遵循获取锁的顺序。

xv6里面有很多这种上锁顺序的实例，尤其是同时持有两个锁的上锁顺序，几乎处处可见。

例如xv6有许多长度为2的lock-order chain涉及到每个进程的锁，因为`sleep`的执行方式（在第7章会讲）。`cnosoleintr`函数是处理键入字符的中断的一部分，当一行输入到达时，所有等待console输入的进程都应该被wakeup。为了达到这个，`consoleintr`当调用wakeup时会持有cons.lock，wakeup获取等待进程的锁为了唤醒它们。因此，避免死锁的全局上锁顺序中包含了cons.lock必须在任何进程锁之前获得。

xv6的文件系统代码包含最长的上锁序列。例如，在创建一个文件时，我们需要同时获取以下的锁才能继续执行：文件所在目录的锁，新文件的inode的锁，磁盘缓冲块的锁，磁盘驱动器的锁，以及调用进程的锁。按以上的顺序获取这些锁，死锁才不会发生。

遵循一个全局的上锁顺序是十分困难的。例如，在一段代码的逻辑可能是先调用M1然后调用M2，但是上锁顺序要求，先取得M2中的锁，再取得M1中的锁。这种事常发生的原因是，上锁顺序并不总是能事先知道的，可能需要持有一把锁之后，才能发现下一把要获取的锁是哪个。例如，文件系统连续地查找路径名中的目录或文件时，获取前一级结点才能查询下一级；或者在wait和exit中，获取父进程的锁，才能在进程表中查找它的子进程。

Real World

尽管对并发问题和相关原语已经进行了多年的研究，使用互斥锁进行编程仍然十分具有挑战性。通常最好将锁隐藏在更高级的结构中，比如同步队列，xv6并没有这么做。如果你使用锁进行编程，最好使用一些能够识别竞争条件的工具，以帮助你构建正确的并发程序。

现在大多数操作系统都支持**POSIX threads**（Pthreads），用户可以利用它，实现多个线程在多个CPU上并发运行。Pthreads支持用户级别的锁，内存屏障等功能。Pthreads也需要操作系统支持一些工作，来保证其正常运作。例如，如果一个pthread阻塞在系统调用上，同一进程中的另一个pthread应该可以抢占在该CPU上执行；又或者，如果一个pthread改变了进程的地址空间，对于运行同一进程的其它pthreads的其它CPU，内核应该更新这些CPU的页表硬件，以反映进程地址空间的变化。

不使用前面所介绍的原子硬件指令来构建互斥锁也是行得通的，不过开销很大，因此大多数操作系统还是会使用原子硬件指令作为构件。

多个CPU对于锁的争夺是一件很可怕的事，特别是如果把CPU的缓存也考虑进去的时候。如果一个CPU持有一把锁，并且缓存在它的局部缓存中，那么当另一个CPU需要抢占这把锁的时候，需要用原子指令，更新持有该锁的CPU的局部缓存，并将对应这把锁的缓存行Cache Line（Cache Line可以简单的理解为CPU Cache中的最小缓存单位）移动到另一个CPU的局部缓存中，从而使原CPU中的很多缓存行可能都失效。因此，这告诉我们，从另一个CPU的局部缓存中，获取一行Cache Line，可能比从本地缓存中获取一行Cache Line代价要昂贵得多。

为了避免使用锁而带来的种种问题和大量开销，很多操作系统也提供了无锁Lock-Free的数据结构和算法。例如，对于不加锁的普通链表，我们自然不能保证并发访问操作该链表的原子性，然而，现在提供了往链表中插入一项的原子性指令。无锁编程是更为复杂和困难的，例如，必须担心指令和内存的重新排序问题。

Lec10 Multiprocessors and locking

内容和book中差不多。