同步与互斥

概述

目标

1. 理解操作系统的同步互斥的设计要求
2. 掌握在 XV6 中互斥机制的具体实现
3. 了解经典进程同步问题，并能使用同步机制解决进程同步问题
4. 熟悉信号量的使用以及实现

简述

同步和互斥是计算机科学中用于处理多个线程或进程访问共享资源时的重要概念。它们都是为了确保在访问共享资源时不会发生意外的错误。

同步指的是一组操作，在这组操作中，一个进程或线程需要等待另一个进程或线程完成某些任务后才能执行。例如，在生产者-消费者问题中，如果缓冲区已满，则生产者必须等待消费者消耗一些数据后再继续生产。同步可以通过各种机制实现，如信号量、锁和条件变量等。

互斥指的是一种对共享资源进行保护的机制，以防止多个进程或线程同时访问它。当一个进程或线程正在访问共享资源时，其他进程或线程需要等待直到该进程或线程释放资源，才能够获得对资源的访问权限。互斥可以通过加锁来实现，确保任何时候只有一个进程或线程可以访问共享资源。

在并发编程中，同步和互斥通常是不可避免的，并且它们的正确使用对于程序的正确性和性能至关重要。

在本部分中，我么将会介绍 XV6 系统是实现并发控制同步与互斥的机制。在实验部分，会由浅入深的安排若干实验，从学习如何使用 Mutex ，信号量等并发控制工具，并设计实现 XV6 上的信号量。

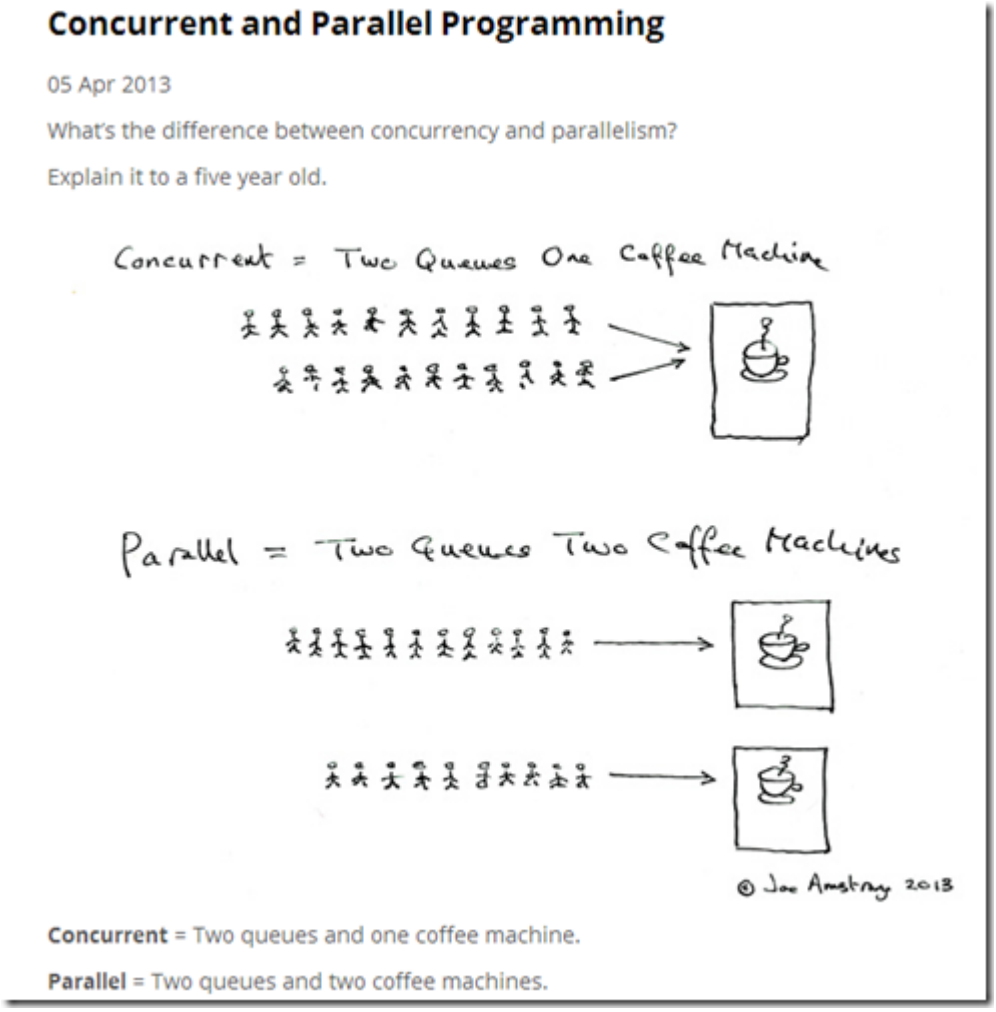
讲解

并行？并发？同步？互斥？

首先，先来说明一下在多核编程时经常出现的一些概念。

- 并发（Concurrency）
- 如果某个系统支持两个或者多个动作的**同时存在**，那么这个系统就是一个并发系统。
    - 并发是指在一定的时间范围内，有多个任务在执行；具体到某个时间节点，可能只有一个任务在执行，其他任务在排队等待。在一定的时间范围内，各个任务轮流运行，“看起来”好像在这个时间段内，这些任务在同时运行。
- 并行（Parallelism）
- 如果某个系统支持两个或者多个动作**同时执行**，那么这个系统就是一个并行系统。
    - 并行是指两个或者多个任务在同一时刻被执行。
    - **“并行”概念是“并发”概念的一个子集**。

`Erlang` 之父 `Joe Armstrong`（伟大的异步编程先驱）用一张 5 岁小孩都能看懂的图片解释了并发与并行的区别：



上图的例子很直观的体现了：

- 并发(Concurrent) 是多个队列使用同一个咖啡机，然后两个队列轮换着使用（未必是 1:1 轮换，也可能是其它轮换规则），最终每个人都能接到咖啡
- 并行(Parallel) 是每个队列都拥有一个咖啡机，最终也是每个人都能接到咖啡，但是效率更高，因为同时可以有两个人在接咖啡

对于并发系统，多个任务“看起来”在同时执行，但有时我们要这些任务的执行具有一定的先后顺序，或者不让它们“同时”执行。

- 同步（Synchronization）
- **同步**指的是进程间的执行需要按照某种先后顺序，即访问是有序的。
- 互斥（Mutual exclusivity）
- **互斥**指的是对于某些共享资源的访问不能同时进行，同一时间只能有一定数量的进程进行。

这两种情况基本构成了我们在多线程执行中遇到的各种问题。这两种情况基本构成了我们在多线程执行中遇到的各种问题。

与同步互斥相关的另一个概念是**临界区**。

- 临界区指的是进程的一段代码，其特征要求了同一时间段只能有一个进程执行，否则就有可能出现问题。

在进入临界区前，进程请求进入的许可，这段代码称为**进入区**；
退出临界区时，进程应该通过协议告知别的进程自己已经使用完临界区，这段代码称为**退出区**；
临界区其他的部分称为**剩余区**。

为什么要多核?

故事要从应用程序想要使用多个 CPU 核开始。使用多个 CPU 核可以带来性能的提升，如果一个应用程序运行在多个 CPU 核上，并且执行了系统调用，那么内核需要能够处理并行的系统调用。如果系统调用并行的运行在多个 CPU 核上，那么它们可能会并行的访问内核中共享的数据结构。

XV6 中有很多共享的数据结构，例如 proc、ticks 和我们之后会看到的 buffer cache 等等。当并行的访问数据结构时，例如一个核在读取数据，另一个核在写入数据，我们需要使用一定的机制来协调对于共享数据的更新，以确保数据的一致性。

但是实际的情况有些令人失望，因为我们想要通过并行来获得高性能，我们想要并行的在不同的 CPU 核上执行系统调用，但是如果这些系统调用使用了共享的数据，我们又需要使用同步机制，而这些同步机制又会使得这些系统调用串行执行，所以最后这些同步机制反过来又限制了性能。

以上是一个大概的介绍，但是回到最开始，为什么操作系统一定要使用多个 CPU 核来提升性能呢？这个实际上与过去几十年技术的发展有关，下面这张非常经典的图可以解释为什么。

这张图有点复杂，X 轴是时间，Y 轴是单位，Y 轴具体意义取决于特定的曲线。

这张图中的核心点是，大概从2000年开始：

- CPU的时钟频率就没有再增加过了（绿线）。
    - 这样的结果是，**CPU 的单线程性能达到了一个极限**并且也没有再增加过（蓝线）。
- 但是另一方面，CPU 中的晶体管数量在持续的增加 （最上方的深红色线）。
    - 所以现在不能通过使用单核来让代码运行的更快，**要想运行的更快，唯一的选择就是使用多个 CPU 核**。所以从2000年开始，处理器上核的数量开始在增加（黑线）。

所以现在如果一个程序想要提升性能，它不能只依赖单核，必须要依赖于多核。这也意味着，如果程序与内核交互的较为紧密，那么操作系统也需要高效的在多个 CPU 核上运行。这就是我们对内核并行的运行在多个 CPU 核上感兴趣的直接原因。

自旋锁 - 互斥

为了实现多线程的互斥，自旋锁是一个常见的选择。

自旋锁是一种基于忙等待的锁机制，在多核 CPU 系统中使用较为广泛。当一个线程请求自旋锁时，如果该锁已被其他线程持有，则该线程将不断循环判断是否可以获得该锁，直到它成功获取到该锁再继续执行。

锁的特性就是只有一个进程可以获取锁，在任何时间点都不能有超过一个锁的持有者。锁的特性就是只有一个进程可以获取锁，在任何时间点都不能有超过一个锁的持有者。

为了达到这一点，我们来看看 XV6 是如何实现的。

XV6 中自旋锁的实现

下图是 XV6 中自旋锁结构体的定义：

```
struct spinlock {       rtm, 17年前 • cvs add spinlo
  uint locked;       // Is the lock held?

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

该结构体的最关键的就是第一行 `locked` 字段。
其核心思想也很简单：在一个死循环里，不断查看 `locked` 字段，如果为 0 那表明当前锁没有持有者，然后设置 `locked` 字段为1，就获取到了锁。

其 C 代码类似于：

```c
void acquire(struct spinlock *lk){
    while(1){
        if(lk->locked == 0){
            lk->locked = 1;
            return;
        }
    }
}
```

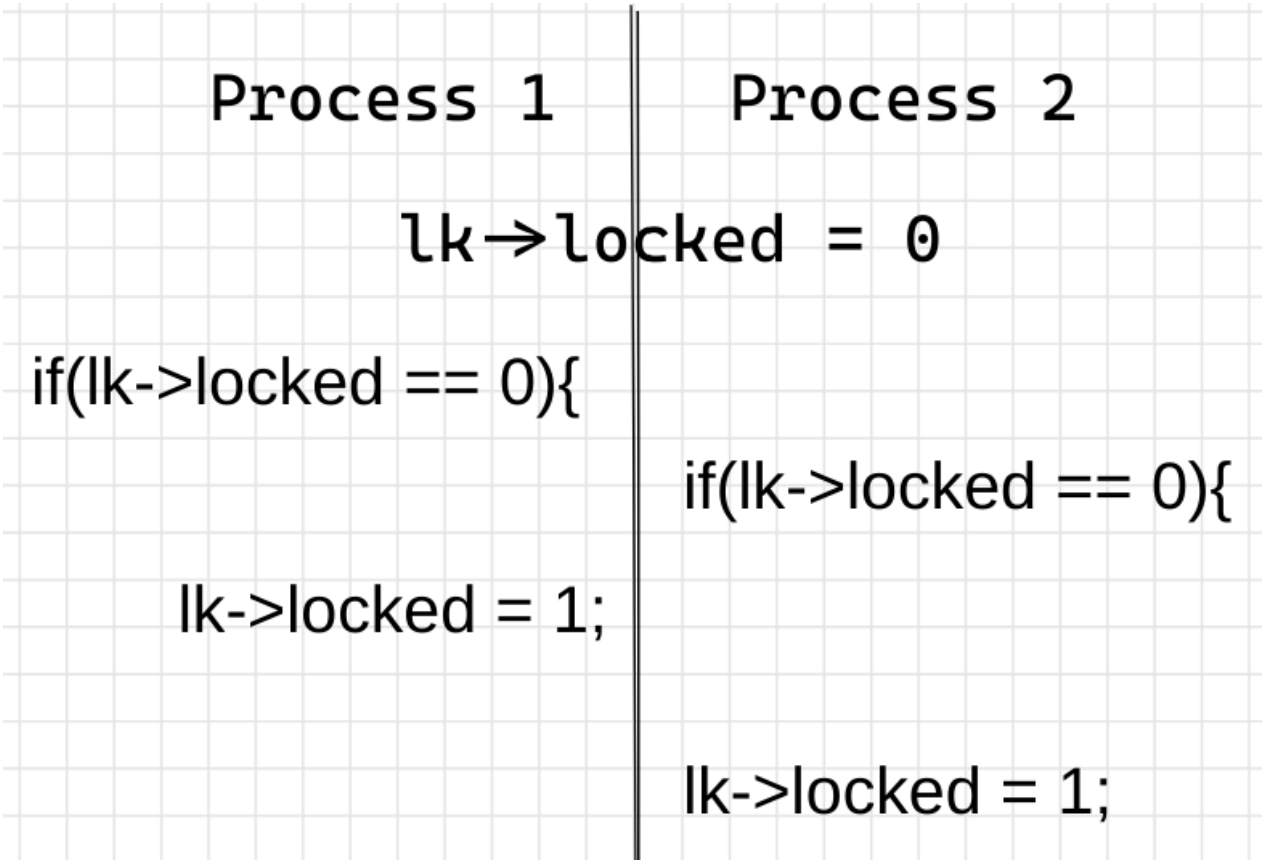
问题 1 - 原子操作

请问以上的 C 代码所实现的自旋锁能正确的运行吗？

答案是不能，因为你查看 `locked` 字段值，然后修改的这一系列操作（**read-modify-write**）不是原子操作。

> 原子操作是指一组操作在执行过程中不会被其他并发操作所干扰或打断的操作。通常，原子操作是一个不可分割的操作单位，针对某个特定的内存地址或变量进行读写操作。原子操作可以确保在多线程或者分布式系统中数据的一致性和正确性。

考虑以下的执行顺序



`locked` 的初始值为 0, 进程 1 与进程 2 同时看到 `locked == 0`，然后都进行修改，也就是说进程 1,2 都获取到了锁。

这违背了锁的特性，所以这样的实现是错误的。

对于存在两个以上进程的系统，不依靠硬件提供的指令支持，仅仅在软件层面，这个问题是无解的。

> Peterson算法 可以控制**两个进程**访问一个共享的单用户资源而不发生访问冲突，仅靠软件算法。

实际上，我们上文中检查 `locked` 字段是否为 0, 若为 0, 则赋值为 1。这一系列操作名为 test and set.

> test-and-set 是一种操作，通常用于多线程编程和并发控制中，用于设置一个变量并返回其旧值。它需要一个参数，即待修改的内存地址，在执行时，先读取该地址处的值返回给调用者，然后将该地址处的值修改为 1。

为了解决这里的问题，即保证 test-and-set 操作的原子性，最常见的方法是依赖于一个特殊的硬件指令。这个特殊的硬件指令会保证一次 test-and-set 操作的原子性。在 RISC-V 上，这个特殊的指令就是 `amoswap (atomic memory swap)`。
这个指令接收3个参数，分别是 address，寄存器 r1，寄存器 r2。这条指令会先锁定住 address，将 address 中的数据保存在一个临时变量中（tmp），之后将 r1中的数据写入到地址中，之后再将保存在临时变量中的数据写入到 r2中，最后再对于地址解锁。

此外，非常方便的是 Gcc 编译器的 GNU 拓展，为我们提供了 C 语言层面的原子 test-and-set 操作 `__sync_lock_test_and_set()`，编译器会自动这个函数转化为对应平台的原子 test-and-set 实现。

有了这一工具，我们的自旋锁的实现就变成了：

```
void acquire(struct spinlock *lk){
    while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
        ;

    // Something else
}
```

问题 2 - 内存序

现实世界中还存在另一个问题：内存序。

因为 Cache 的存在，以及编译器层面的源代码乱序，CPU 层面的指令乱序，实际上代码的执行顺序并不是按照我们所写的，完全顺序执行的。

为了描述这一“乱序”，所以抽象出了内存序这一概念，对此我们不过多介绍。

更多内容可以看下面这篇文章。

> https://zhuanlan.zhihu.com/p/611868395

考虑到这一问题，自旋锁的实现变为以下：

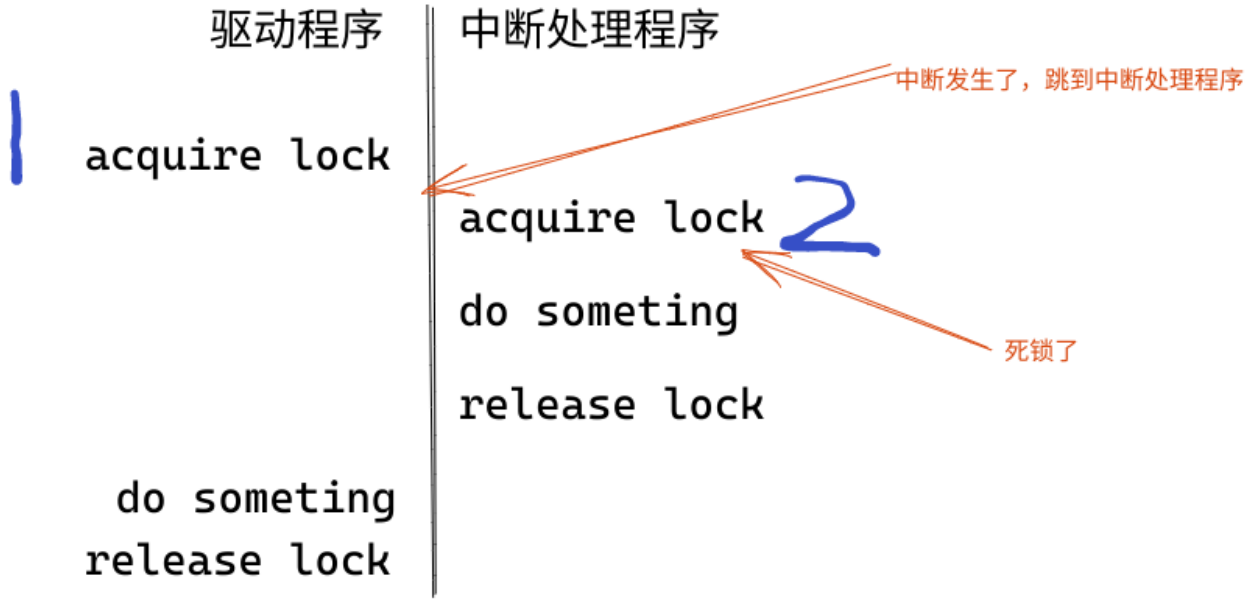
```
void acquire(struct spinlock *lk){
    while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
        ;

    // No-reorder execution and full memory fence
    __sync_synchronize();

    // Something else
}
```

问题 3 - 中断

以上的自旋锁的实现，是一个标准的用户态下的自旋锁的实现。

但对于，操作系统内核态下的自旋锁而言，这个实现还不够正确，因为没有考虑中断。这一点也是操作系统内核代码的复杂所在。

考虑一个单核处理器（或者这两个事件都被同一个 CPU 执行）上的操作系统情景：



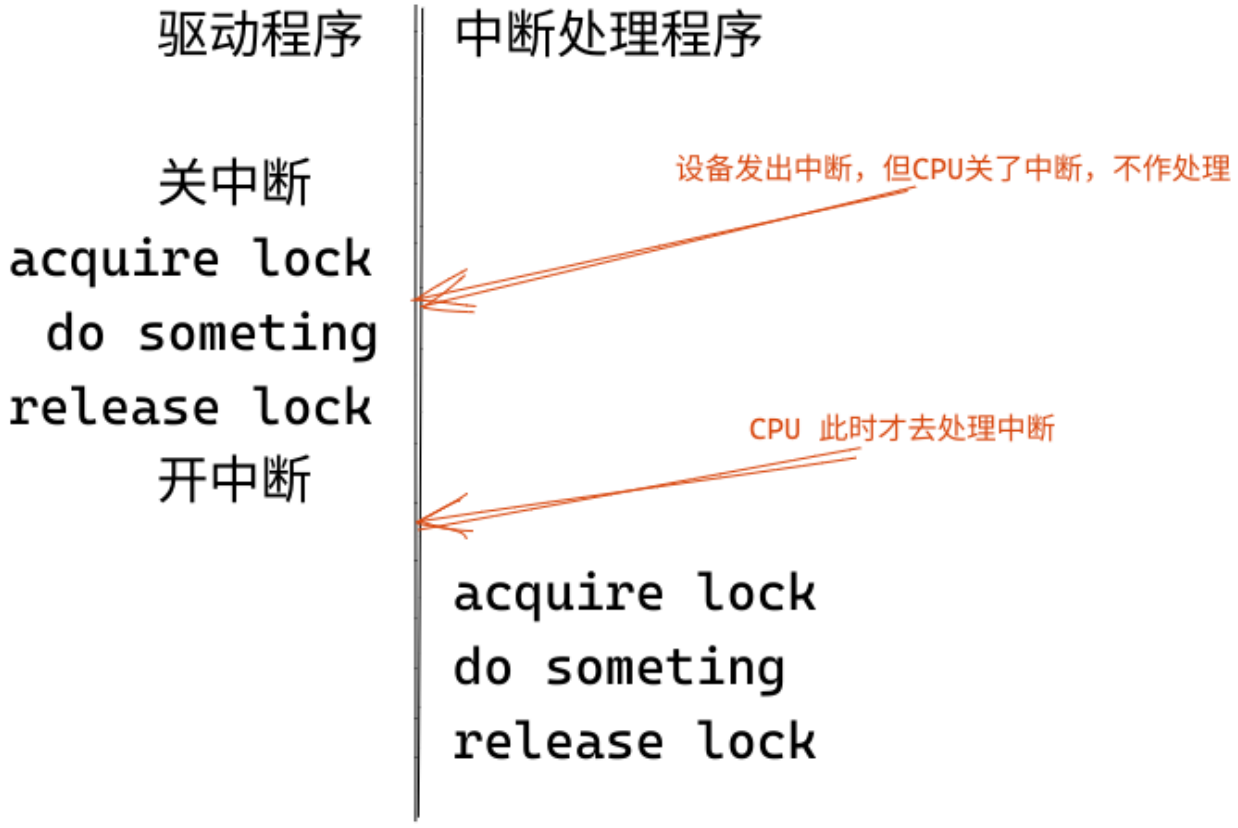
1. CPU 0 正在执行驱动程序代码

    1. 首先，获取到了一个锁
2. 这时，中断发生了，CPU 0 跳到中断处理程序
    1. 首先，获取同一个锁（驱动程序代码里那个锁）
3. 这时就进入了死锁

CPU 0 在驱动程序里获取到了锁，然后在中断处理程序里也要获取同一个锁，但这个锁被 CPU 0 自己的驱动程序所持有，驱动程序正等着中断程序结束，中断程序要获取到锁才能结束。这就死锁了。

> 这一情景在 XV6 的 UART 部分就真实存在，uartputc 函数会 acquire 锁，UART 本质上就是传输字符，当 UART 完成了字符传输它会做什么？是的，它会产生一个中断之后会运行 uartintr 函数，在 uartintr 函数中，会获取同一把锁，但是这把锁正在被 uartputc 持有。如果这里只有一个 CPU 的话，那这里就是死锁。中断处理程序 uartintr 函数会一直等待锁释放，但是 CPU 不出让给 uartputc 执行的话锁又不会释放。在 XV6中，这样的场景会触发 panic，因为同一个 CPU 会再次尝试 acquire 同一个锁。

为了解决这一问题，获取锁先关中断。



自旋锁的代码变成：

```
void acquire(struct spinlock *lk){
    // 关中断
    intr_off();

    while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
        ;

    // No-reorder and full memory fence
    __sync_synchronize();

    // Something else
}
```

问题 4 - 锁的嵌套

以上的实现，在获取锁的时候关中断，在释放锁的时候开中断，看起来已经十分正确了。

但很遗憾，这个实现依旧是错的。

操作系统不止有一把锁，同一个函数里也可能不会仅获取一把锁。这时获取锁关中断，释放锁开中断的策略就有问题了。

考虑一下情景，`foo` 函数依次获得两把锁，然后再释放。

```
void foo(){
    acquire(lock_a); // intr_off()
    acquire(lock_b); // intr_off()

    // Do something

    release(lock_b); // intr_on()
                     //         <--- Bug here
    release(lock_a); // intr_on()
}
```

- 获取锁 a
- 获取锁 b
- 释放锁 b
- 注意，这时我们依旧持有锁 a
    - 注意，这时中断已经被打开了
    - **这就回到了问题 3 的情景，在中断打开的情况下，持有一把锁，这可能会导致死锁。**
- 释放锁 a

我们希望直到持有的**最后一把自旋锁被释放时，才能开中断**，这可以通过一个“计数器”来实现。

`acquire` 与 `release` 的过程比较像压栈，acquire 的时候就会往栈上 push 一把锁，再 acquire 的时候就会再 push 一把锁，release 的时候就会把栈顶的锁弹出，我们用一个计数器来维护栈的大小，往栈里 push 锁的时候计数+1，pop 的时候计数-1，当发现栈里面所有的锁都被 pop 出来之后再打开处理器的中断。

但仅仅这样做是不够的：

- 中断处理程序，可能会使用自旋锁以避免数据竞争。
- 中断处理程序往往不希望被其他中断所打断，因此会在其执行的过程当中关闭中断
- 我们上边的策略是释放最后一把锁的时候，开中断。但假如是在中断处理程序中，释放了所有锁，但中断处理程序还没有处理完，此时开中断将可能会导致错误。

因此我们要扩展我们的实现，不妨在第一次执行 `acquire` 的时候把处理器的中断情况记录下来，最后一次 release 的时候再把中断状态恢复到最开始的状态。因此我们可以每次执行 acquire 的时候把 CPU 的 flags 压入栈中，每次 release 的时候就把栈顶保存的 CPU flags pop 到处理器上。由于第一次执行完 acquire 之后，后面的 acquire 操作在进行时 CPU 都是关中断的状态，CPU flags 都是一样的，所以说只需要记录第一次压入栈中的 CPU flags 和栈的高度即可。

也就是说：

- 关中断，若是第一次获取锁的时候，保存旧的中断状态。
- 释放最后一把锁的时候，恢复到第一次获取锁时所保存的中断状态。

至此，内核态的自旋锁的正确实现为：

```
void acquire(struct spinlock *lk){
    // 第一次获取锁的时候，保存当前中断状态，然后关中断
    push_off();
```

```
    while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
        ;

    // No-reorder and full memory fence
    __sync_synchronize();

    // Something else
}

void release(struct spinlock *lk){

    // No-reorder exec and full memory fence
    __sync_synchronize();

    // equivalent to lk->locked = 0 atomicly.
    __sync_lock_release(&lk->locked);

    // 释放最后一把锁的时候，恢复到第一次获取锁时所保存的中断状态。
    pop_off();
}
```

还有一个问题，刚刚说的那个栈应该保存在哪里呢？我们应该把这个栈保存在 CPU 对应的结构体里，原因如下：在前面的场景里我们先后获取了好多把锁，所以说把栈保存在某一个锁对应的结构体里肯定不合适；保存在线程的结构体里理论上是可以的，但因为线程第一次关闭中断之后它就独占 CPU 运行了，它就和它此时所在的 CPU 绑定了，现代操作系统中同时运行着成千上万的线程与进程，CPU 的数目远少于线程数，所以说没有必要在线程结构体中浪费那么多存储空间，不如把 flags 栈放到 CPU 结构体里。

```
void
push_off(void)
{
  int old = intr_get();

  intr_off();
  if(mycpu()->noff == 0)
    mycpu()->intena = old;
  mycpu()->noff += 1;
}

void
pop_off(void)
{
  struct cpu *c = mycpu();
  c->noff -= 1;
  if(c->noff == 0 && c->intena)
    intr_on();
}
```

给内核使用的自旋锁，因为内核需要相应处理中断，因此设计上有众多需要额外考虑的地方。上文通过 4 个问题，一步步实现了一个自旋锁，也就是 XV6 中自旋锁的实现。

自旋锁的缺点

自旋锁的主要缺点是在高负载情况下会导致 CPU 资源的浪费。因为当一个进程获得自旋锁时，如果其他线程也想获取该锁，它们会一直循环等待，直到该锁被释放。这种循环等待的过程需要消耗 CPU 资源，从而降低了系统的性能。

此外，当自旋锁用于保护某些被频繁访问的数据结构时，如果持有锁的线程执行时间太长，其他线程可能会一直等待，导致系统出现死锁或饥饿等问题。

“自旋”可以理解为“自我旋转”，这里的“旋转”指“循环”，比如 while 循环或者 for 循环。“自旋”就是自己在这里不停地循环，直到目标达成，这一过程也是占用 CPU 时间的，假如获取锁十分耗时，比如外部设备（机械硬盘等）所需要的时间比较长，我们依旧占用 CPU 时间，不断尝试获取锁，这就很浪费。最好能够进入“休眠”，让出 CPU ，等条件允许再被唤醒。

此外，自旋锁只能用于互斥，不能用于同步。也就是说，我们无法掌控获取自旋锁的进程，它们获取到锁的顺序，这是自旋锁的固有限制。

大多数高级语言通常提供 Mutex，可以说自旋锁就是一种最简单的 Mutex 实现.

wakeup & sleep - 同步

当你在写一个线程的代码时，有些场景需要等待一些特定的事件，或者不同的线程之间需要交互。

- 假设我们有一个Pipe，并且我正在从Pipe中读数据。但是Pipe当前又没有数据，所以我需要等待一个Pipe非空的事件。
- 类似的，假设我在读取磁盘，我会告诉磁盘控制器请读取磁盘上的特定块。这或许要花费较长的时间，尤其当磁碟需要旋转时
    - （通常是毫秒级别），磁盘才能完成读取。而执行读磁盘的进程需要等待读磁盘结束的事件。
- 类似的，一个Unix进程可以调用wait函数。这个会使得调用进程等待任何一个子进程退出。所以这里父进程有意的在等待另一个进程产生的事件。

以上就是进程需要等待特定事件的一些例子。特定事件可能来自于 I/O，也可能来自于另一个进程，并且它描述了某件事情已经发生。以上就是进程需要等待特定事件的一些例子。特定事件可能来自于 I/O，也可能来自于另一个进程，并且它描述了某件事情已经发生。

"Sleep-and-wakeup" 机制是指计算机操作系统通过控制进程的运行状态，使其在特定条件下进入休眠状态并在需要时唤醒进程的一种机制。

当进程在等待其他任务完成时，操作系统会将其挂起并放入睡眠状态。这可以防止进程浪费处理器资源，并且可以使其他进程获得更多的资源以提高系统整体性能。进程进入睡眠状态后，它将不再占用任何 CPU 时间，直到满足某些特定的条件，如接收到中断信号或等待的事件发生等。

当操作系统检测到某个进程需要被唤醒时，它会向该进程发送一个中断信号或者通知该进程等待的事件已经发生。此时，进程将从睡眠状态中恢复并继续执行其先前被挂起的任务。通过这种方式，操作系统可以有效地管理系统资源，并确保进程在需要时始终处于活动状态。

连接 `sleep` 与 `wakeup`

这里有件事情需要注意， `sleep` 和 `wakeup` 函数需要通过某种方式链接到一起。

也就是说，如果我们调用 `wakeup` 函数，我们只想唤醒正在等待刚刚发生的特定事件的进程。所以， `sleep` 函数和 `wakeup` 函数都带有一个叫做 `sleep channel` 的参数。我们在调用 `wakeup` 的时候，需要传入与调用 `sleep` 函数相同的 `sleep channel` 。不过 `sleep` 和 `wakeup` 函数只是接收表示了 `sleep channel` 的 64bits 数值，它们并不关心这个数值代表什么。当我们调用 `sleep` 函数时，我们通过一个 `sleep channel` 表明我们等待的特定事件，当调用 `wakeup` 时我们希望能传入相同的数值来表明想唤醒哪个线程。

这两个函数的声明如下：

```
void sleep(void *chan, struct spinlock *lk);
void wakeup(void *chan);
```

对于 sleep 函数，有一个有趣的参数，我们需要将一个锁作为第二个参数传入，这背后是一个大的故事，我后面会介绍背后的原因。

丢失唤醒

在解释 sleep 函数为什么需要一个锁使用作为参数传入之前，我们先来看看假设我们有了一个更简单的不带锁作为参数的 sleep 函数，会有什么样的结果。这里的结果就是丢失唤醒（ lost wakeup）。

假设 sleep 只是接收任意的 sleep channel 作为唯一的参数。它其实不能正常工作，我们称这个 sleep 实现为 broken_sleep：

- 将进程的状态设置为 SLEEPING，表明当前进程不想再运行，而是正在等待一个特定的事件。
- 需要记录特定的 sleep channel 值，这样之后的 `wakeup` 函数才能发现是当前进程正在等待 `wakeup` 对应的事件。
- 最后再调用 `switch` 函数让出 CPU。

之后是 wakeup 函数。我们希望唤醒所有正在等待特定 sleep channel 的线程。

- wakeup 函数中会查询进程表单中的所有进程，如果进程的状态是 SLEEPING 并且进程对应的 channel 是当前 wakeup 的参数，那么将进程的状态设置为 RUNNABLE。

考虑一下情景：

```
void uartwrite(){
    acquire(tx_lock);
    // Do something

    release(tx_lock);
    broken_sleep(chan);
    acquire(tx_lock);
    // Do other tings
    release(tx_lock);
}

void uartintr(){
    acquire(tx_lock);
    // Do something

    wakeup(chan);
    release(tx_lock);
}
```

接下来，我们会探索为什么只接收一个参数的 `broken_sleep` 在这不能工作。为了让锁能正常工作，我们需要在调用 `broken_sleep` 函数之前释放 `uart_tx_lock` ，并在 `broken_sleep` 返回时重新获取锁。

问题在于：

- 在前面的代码中，`broken_sleep` 之前释放了锁，但是在释放锁和 `broken_sleep` 之间可能会发生中断。
- 一旦释放了锁，当前 CPU 的中断会被重新打开。因为这是一个多核机器，所以中断可能发生在任意一个 CPU 核。
- 其他 CPU 核上正在执行 UART 的中断处理程序 `uartintr()`，并且正在 acquire 函数中等待当前锁释放。所以一旦锁被释放了，另一个 CPU 核就会获取锁，最后再调用 wakeup 函数，并传入 tx_chan。
- 目前为止一切都还好，除了一点：现在写进程程 `uartwrite` 还在执行并位于 release 和 broken_sleep 之间，也就是写线程还没有进入 SLEEPING 状态，所以 `uartwrite` 中的 wakeup 并没有唤醒任何进程，因为还没有任何进程在 tx_chan 上睡眠。
- 之后写线程会继续运行，调用 broken_sleep，将进程状态设置为 SLEEPING，保存 sleep channel。
- 但是中断已经发生了，wakeup 也已经被调用了。所以这次的 broken_sleep，没有人会唤醒它，因为 wakeup 已经发生过了。

这就是 lost wakeup 问题。

现在我们的目标是消灭掉 lost wakeup。这可以通过消除释放锁与进入 sleep 之间的时间窗口来达到。

```
void uartwrite(){
    acquire(tx_lock);
    // Do something

    release(tx_lock);
                    // <- intr_on , bug here
    broken_sleep(chan);
    acquire(tx_lock);
    // Do other tings
    release(tx_lock);
}
```

首先我们必须要释放 `tx_lock` 锁，因为中断需要获取这个锁，但是我们又不能在释放锁和进程将自己标记为 SLEEPING 之间留有窗口。
这样中断处理程序中的 `wakeup` 才能看到 SLEEPING 状态的进程，并将其唤醒，进而我们才可以避免 lost wakeup 的问题。所以，我们应该消除这里的窗口。

为了实现这个目的，我们需要将 sleep 函数设计的稍微复杂点。这里的解决方法是，即使 `sleep` 函数不需要知道你在等待什么事件，它还是需要你知道你在等待什么数据，并且传入一个用来保护你在等待数据的锁。

在接口层面，`sleep` 承诺可以原子性的将进程设置成 SLEEPING 状态，同时释放锁。
这样 `wakeup` 就不可能看到这样的场景：锁被释放了但是进程还没有进入到 SLEEPING 状态。
所以 sleep 这里将释放锁和设置进程为 SLEEPING 状态这两个行为合并为一个原子操作。

- 所以我们需要有一个锁来保护 `sleep` 的条件，并且这个锁需要传递给 `sleep` 作为参数。
- 更进一步的是，当调用 `wakeup` 时，锁必须被持有。
- 如果程序员想要写出正确的代码，都必须遵守这些规则来使用 `sleep` 和 `wakeup`。

下面来看，XV6 中的实现代码：

首先我们来看一下 proc.c 中的 `wakeup` 函数。

```
1  // Wake up all processes sleeping on chan.
2  // Must be called without any p->lock.
3  void
4  wakeup(void *chan)
5  {
6    struct proc *p;
7
8    for(p = proc; p < &proc[NPROC]; p++) {
9      if(p != myproc()){
10       acquire(&p->lock);
11       if(p->state == SLEEPING && p->chan == chan) {
12         p->state = RUNNABLE;
13       }
14       release(&p->lock);
15     }
16   }
17 }
```

`wakeup` 函数并不十分出人意料。它查看所有进程，对于每个进程首先加锁，这点很重要。之后查看进程的状态，如果进程当前是 SLEEPING 并且进程的 channel 与 wakeup 传入的 channel 相同，将进程的状态设置为 RUNNABLE。最后再释放进程的锁。

下面是带有锁作为参数的 `sleep` 函数。

```
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  // Must acquire p->lock in order to
  // change p->state and then call sched.
  // Once we hold p->lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup locks p->lock),
  // so it's okay to release lk.

  acquire(&p->lock);  //DOC: sleeplock1
  release(lk);

  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  // Tidy up.
  p->chan = 0;

  // Reacquire original lock.
  release(&p->lock);
  acquire(lk);
}
```

我们已经知道了 `sleep` 函数需要释放作为第二个参数传入的锁，这样中断处理程序才能获取锁。函数中第一件事情就是释放这个锁。当然在释放锁之后，我们会担心在这个时间点相应的 `wakeup` 会被调用并尝试唤醒当前进程，而当前进程还没有进入到 SLEEPING 状态。所以我们不能让 `wakeup` 在 release 锁之后执行。为了让它不在 release 锁之后执行，在 release 锁之前， `sleep` 会获取即将进入 SLEEPING 状态的进程的锁。

而 `wakeup` 在唤醒一个进程前，需要先获取进程的锁。
所以在 `sleep` 函数中调用 `sched` 函数之间，这个线程一直持有了保护 sleep 条件的锁或者 p->lock 。

这里的效果是由之前定义的一些规则确保的，这些规则包括了：

- 调用 `sleep` 时需要持有 `condition lock` ，这样 `sleep` 函数才能知道相应的锁。
- `sleep` 函数只有在获取到进程的锁 `p->lock` 之后，才能释放 `condition lock` 。
- `wakeup` 需要同时持有两个锁才能查看进程。

这样的话，我们就不会再丢失任何一个 wakeup，也就是说我们修复了 lost wakeup 的问题。

生产者消费者问题

通过 Wakeup-and-sleep 机制，进程间的同步问题就得到了解决。

接下来，演示通过使用 wakeup & sleep 来解决生产者消费者问题。

```
typedef int Type;
#define N 10
Type buf[N];

struct spinlock lock;
uint rp = 0, wp = 0;

void producer(Type data) {
  acquire(&lock);

  while ((wp + 1) % N == rp) {
    sleep(&wp, &lock);
  }

  buf[wp] = data;
  wp = (wp + 1) % N;
  wakeup(&rp);

  release(&lock);
}

void consumer(Type *data) {
  acquire(&lock);

  while (rp == wp) {
    sleep(&rp, &lock);
  }

  *data = buf[rp];
  rp = (rp + 1) % N;
  wakeup(&wp);

  release(&lock);
}
```

生产者，消费者共用一个环形队列。当队列满的时候，生产者会进入休眠，让出 CPU ；当队列空的时候，消费者会 sleep 让出 CPU 。当生产者生产了一个数据时，会尝试唤醒可能等待的消费者；当消费者消费了一个数据时，会尝试唤醒等待的生产者。

在一些高级语言模型中，这套 wakeup-and-sleep 机制叫作条件变量。

实验

实验介绍

1. 学习使用 pthread 库, 编程解决生产者消费者问题
2. 学习使用信号量，编程解决生产者消费者问题
3. 编程实现信号量 semaphore

实验内容 1 - pthread

```
> git add .
> git commit -m 'message'    #暂存修改
> git switch lab3-1-pthread    #切换到本实验的分支
```

任务：

- 在本机上进行（不是在 RISC-V XV6 里）
- 2 个生产者，2 个消费者，每个生产者/消费者分配一个线程 pthread
  - 桌子上有一个空盘子，允许存放一只水果。
  - 爸爸（生产者 1）可以向盘中放苹果，妈妈（生产者 2）向盘子中放橘子，女儿（消费者 1）专门吃盘子中的苹果，儿子（消费者 2）专门吃盘子中的橘子。
  - 规定当盘子空的时候一次只能放一只水果。
  - **解决方案应该放在 pthreadt.c**

要求：

- 使用 C 语言，使用 Pthread 库
- 使用 mutex ，pthread_cond 等工具

Tips：

- https://hanbingyan.github.io/2016/03/07/pthread_on_linux/

预期输出：

```
❯ make pthread
Dad put an apple on the plate.
Daughter eat an apple, which is the 1th apple.
Dad put an apple on the plate.
Daughter eat an apple, which is the 2th apple.
Mom put an orange on the plate.
Son eat an orange, which is the 1th orange.
Dad put an apple on the plate.
Daughter eat an apple, which is the 3th apple.
Dad put an apple on the plate.
Daughter eat an apple, which is the 4th apple.
Mom put an orange on the plate.
Son eat an orange, which is the 2th orange.
Mom put an orange on the plate.
Son eat an orange, which is the 3th orange.
Mom put an orange on the plate.
Son eat an orange, which is the 4th orange.
```

实验内容2 - 使用信号量

```
> git add .
> git commit -m 'message'    #暂存修改
> git switch lab3-2-sem    #切换到本实验的分支
```

任务：

- 在本机上进行（不是在 RISC-V XV6 里）
- 使用信号量解决*实验内容 1*里的问题
  - `#include<semaphore.h>`
  - **解决方案应该放在 semt.c**

要求：

- 使用 C 语言，使用 Pthread 库
- 使用 semaphore
- 不可以使用 mutex ，pthread_cond 等工具

Tips：

- https://linux.die.net/man/7/sem_overview

预期输出：

```
❯ make sem
Dad put an apple on the plate.
Daughter eat an apple, which is the 1th apple.
Dad put an apple on the plate.
Daughter eat an apple, which is the 2th apple.
Mom put an orange on the plate.
Son eat an orange, which is the 1th orange.
Dad put an apple on the plate.
Daughter eat an apple, which is the 3th apple.
Mom put an orange on the plate.
Son eat an orange, which is the 2th orange.
Dad put an apple on the plate.
Daughter eat an apple, which is the 4th apple.
Mom put an orange on the plate.
Son eat an orange, which is the 3th orange.
Mom put an orange on the plate.
Son eat an orange, which is the 4th orange.
```

实验内容3 - 实现信号量

```
> git add .
> git commit -m 'message'    #暂存修改
> git switch lab3-3-sem_impl    #切换到本实验的分支
```

任务：

- 在本机上
- 使用 mutex 与条件变量实现信号量
  - **在文件 sem_impl.c 里进行你的实现**

要求：

- 使用 C 语言，使用 Pthread 库
- 使用 mutex ，pthread_cond 等工具

预期输出：

```
❯ make sem_impl
Dad put an apple on the plate.
Daughter eat an apple, which is the 1th apple.
Mom put an orange on the plate.
Son eat an orange, which is the 1th orange.
Dad put an apple on the plate.
Daughter eat an apple, which is the 2th apple.
Mom put an orange on the plate.
Son eat an orange, which is the 2th orange.
Dad put an apple on the plate.
Daughter eat an apple, which is the 3th apple.
Mom put an orange on the plate.
Son eat an orange, which is the 3th orange.
Dad put an apple on the plate.
Daughter eat an apple, which is the 4th apple.
Mom put an orange on the plate.
Son eat an orange, which is the 4th orange.
```

拓展实验内容

> 互斥量、信号量、条件变量这三者可以互相实现，请最多用其余两者实现第三者。

Tips：

- 信号量→互斥量
- 互斥量+信号量→条件变量
- 条件变量+互斥量→信号量

> https://bigcat.ee/mutex-semaphore-condvar/

附录

> https://www.infoq.com/presentations/go-locks/