同步互斥

练习1: 了解信号量和管程的实现机制

1. **同步互斥的底层支持是如何实现的？**

有开关中断和test\_and\_set\_bit等原子操作机器指令的存在，使得我们在实现同步互斥原语上可以 大大简化。在ucore中提供的底层机制包括中断开关控制和test\_and\_set相关原子操作机器指令。最终的cli和sti是x86的机器指令，最终实现了关中断和开中断，即设置了eflags寄存器中与中断相关的位。通过关闭中断，可 以防止对当前执行的控制流被其他中断事件处理所打断。既然不能中断，那也就意味着在内核运行的当前进程无法被打断或 被从新调度，即实现了对临界区的互斥操作。

1. **对比原理课上学到的信号量和p，v操作，说明Ucore中信号量机制的实现。**

在ucore中最重要的信号量操作是P操作函数down(semaphore\_t \*sem)和V操作函数 up(semaphore\_t \*sem)。但这两个函数 的具体实现是\_\_down(semaphore\_t \*sem, uint32\_t wait\_state) 函数和\_\_up(semaphore\_t \*sem, uint32\_t wait\_state)函数， 二者的具体实现描述如下：

\_\_down(semaphore\_t \*sem, uint32\_t wait\_state, timer\_t \*timer)：具体实现信号量的P操作，首先关掉中断，然后判断当前 信号量的value是否大于0。如果是>0，则表明可以获得信号量，故让value减一，并打开中断返回即可；如果不是>0，则表 明无法获得信号量，故需要将当前的进程加入到等待队列中，并打开中断，然后运行调度器选择另外一个进程执行。如果被 V操作唤醒，则把自身关联的wait从等待队列中删除（此过程需要先关中断，完成后开中断）。

\_\_up(semaphore\_t \*sem, uint32\_t wait\_state)：具体实现信号量的V操作，首先关中断，如果信号量对应的wait queue中没 有进程在等待，直接把信号量的value加一，然后开中断返回；如果有进程在等待且进程等待的原因是semophore设置的，则 调用wakeup\_wait函数将waitqueue中等待的第一个wait删除，且把此wait关联的进程唤醒，最后开中断返回。具体实现如下 所示：

1. **Ucore中的信号量是基于信号量和条件变量实现的，请说明其中的数据结构和函数方法的设计。**

typedef struct { int value; //信号量的当前值 wait\_queue\_t wait\_queue; //信号量对应的等待队列 } semaphore\_t;

**练习2: 了解基于信号量和管程的哲学家就餐问题**

**说明ucore中基于信号量的哲学家就餐问题的实现机制。**

**信号量的实现：**

semphore\_t mutex 临界区互斥信号量

semphore\_t s[N] 每个哲学家一个信号量

**说明ucore中基于管程的哲学家就餐问题的实现机制。**

关于使用条件变量来完成哲学家就餐问题的实现中，总共有两个关键函数，以及使用到了N（哲学家数量）个条件变量，在管程中，还包括了一个限制管程访问的锁还有N个用于描述哲学家状态的变量（总共有EATING, THINKING, HUNGER）三种状态；

首先分析phi\_take\_forks\_condvar函数的实现，该函数表示指定的哲学家尝试获得自己所需要进餐的两把叉子，如果不能获得则阻塞，具体实现流程为：

给管程上锁；

将哲学家的状态修改为HUNGER；

判断当前哲学家是否有足够的资源进行就餐（相邻的哲学家是否正在进餐）；

如果能够进餐，将自己的状态修改成EATING，然后释放锁，离开管程即可；

如果不能进餐，等待在自己对应的条件变量上，等待相邻的哲学家释放资源的时候将自己唤醒；

而phi\_put\_forks\_condvar函数则是释放当前哲学家占用的叉子，并且唤醒相邻的因为得不到资源而进入等待的哲学家：

首先获取管程的锁；

将自己的状态修改成THINKING；

检查相邻的哲学家是否在自己释放了叉子的占用之后满足了进餐的条件，如果满足，将其从等待中唤醒（使用cond\_signal）；

释放锁，离开管程；

 由于限制了管程中在访问共享变量的时候处于RUNNABLE的进程只有一个，因此对进程的访问是互斥的；并且由于每个哲学家只可能占有所有需要的资源（叉子）或者干脆不占用资源，因此不会出现部分占有资源的现象，从而避免了死锁的产生；

 根据上述分析，可知最终必定所有哲学将都能成功就餐；