操作系统实验四

1. 实验目的

1.理解操作系统的同步互斥的设计实现；

2.理解底层支撑技术：禁用中断、定时器、等待队列；

3.在ucore中理解信号量（semaphore）机制的具体实现；

4.理解管程机制，在ucore内核中增加基于管程（monitor）的条件变量（condition variable）的支持；

5.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。

1. 实验内容

练习1: 理解内核级信号量的实现和基于内核级信号量的哲学家就 餐问题（不需要编码）

请在实验报告中给出内核级信号量的设计描述，并说其大致执行流流程。

答：将内核信号量机制迁移到用户态的最大麻烦在于，用于保证操作原子性的禁用中断机制、以及CPU提供的Test and Set指令机制都只能在用户态下运行，而使用软件方法的同步互斥又相当复杂，这就使得没法在用户态下直接实现信号量机制；于是，为了方便起见，可以将信号量机制的实现放在OS中来提供，然后使用系统调用的方法统一提供出若干个管理信号量的系统调用，分别如下所示：

1.申请创建一个信号量的系统调用，可以指定初始值，返回一个信号量描述符(类似文件描述符)；

2.将指定信号量执行P操作；

3.将指定信号量执行V操作；

4.将指定信号量释放掉；

请在实验报告中给出给用户态进程/线程提供信号量机制的设计方案，并比较说明给内核级提供信号量机制的异同。

答：相同点：提供信号量机制的代码实现逻辑是相同的；

不同点：1.由于实现原子操作的中断禁用、Test and Set指令等均需要在内核态下运行，因此提供给用户态进程的信号量机制是通过系统调用来实现的，而内核级线程只需要直接调用相应的函数就可以了；

练习2:完成内核级条件变量和基于内核级条件变量的哲学家就餐 问题（需要编码）

首先掌握管程机制，然后基于信号量实现完成条件变量实现，然后用管程机制实现哲学家就 餐问题的解决方案（基于条件变量）。

执行：make grade 。如果所显示的应用程序检测都输出ok，则基本正确。如果只是某程序过不去，比如matrix.c，则可执行makerun-matrix命令来单独调试它。大致执行结果可看附录。

请在实验报告中给出内核级条件变量的设计描述，并说其大致执行流流程。

答：关于条件变量机制的实现主要位于monitor.c文件中的cond\_signal,cond\_wait两个函数中，这两个函数的含义分别表示提醒等待在这个条件变量上的进程恢复执行，以及等待在这个条件变量上，直到有其他进行将其唤醒位置；

cond\_signal: 将指定条件变量上等待队列中的一个线程进行唤醒，并且将控制权转交给这个进程；具体执行流程为：

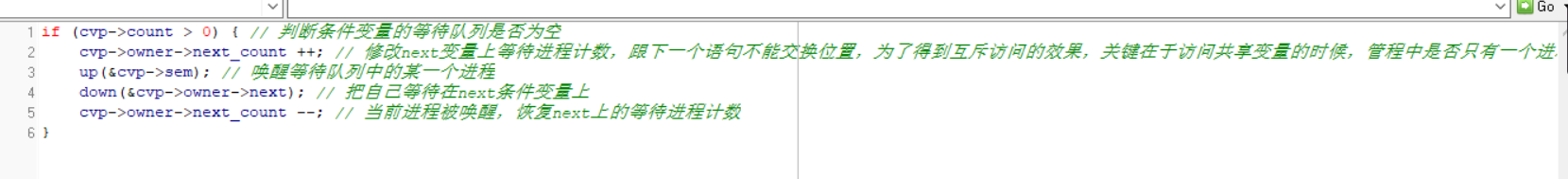
1.判断当前的条件变量的等待队列上是否有正在等待的进程，如果没有则不需要进行任何操作；

2.如果由正在等待的进程，则将其中的一个唤醒，这里的等待队列是使用了一个信号量来进行实现的，由于信号量中已经包括了对等待队列的操作，因此要进行唤醒只需要对信号量执行up操作即可；

3.接下来当前进程为了将控制权转交给被唤醒的进程，将自己等待到了这个条件变量所述的管程的next信号量上，这样的话就可以切换到被唤醒的进程了；由于next信号量的实现，就带来了两个困惑：

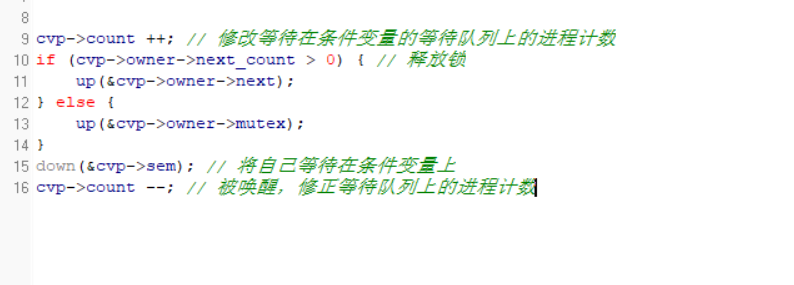
1.等待在next信号量上的进程是否能够被唤醒？由于每一个next信号量上等待的进程的产生必定是因为存在了某个它需要唤醒的进程，而这个进程在结束cond\_wait函数之后返回到管程的函数，还会检查next信号量上是否存在等待着的进程，有的话将其唤醒，因此每一个next信号量上等待的进程最终必定会被唤醒；

2.在等待在next信号量上的时候，管程的mutex锁并没有被释放，是否可能存在该锁永远都被释放不了的情况？不会的。根据前一个问题得知所有next信号量上的等待进程一定会被唤醒，那么最后一个被唤醒的next进程就会将锁释放掉；

代码实现：

3. 接下来，当前进程被从next信号量上被唤醒的时候，首先将next count减一，然后离开cond\_signal函数，回到管程中的函数，检查是否应该释放管程的锁（取决于现在是否还有next信号量上等待的进程，有的话将其唤醒，完成其在函数中的操作，并且将释放锁的操作延迟给这个进程来进行），根据上述描述，我们可以知道在管程中能够运行的进程之间不会有互相有意料外的打断的过程（由于进程的切换时机都是固定好的，由当前的进程来唤醒另外某一个进程），因此实现了对共享变量访问的互斥性；

cond\_wait: 该函数的功能为将当前进程等待在指定信号量上，其操作过程为将等待队列的计数加1，然后释放管程的锁或者唤醒一个next上的进程来释放锁（否则会造成管程被锁死无法继续访问，同时这个操作不能和前面的等待队列计数加1的操作互换顺序，要不不能保证共享变量访问的互斥性），然后把自己等在条件变量的等待队列上，直到有signal信号将其唤醒，正常退出函数；

代码实现： 

本实验中基于条件变量和管程的哲学家就餐问题的实现:

1.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：

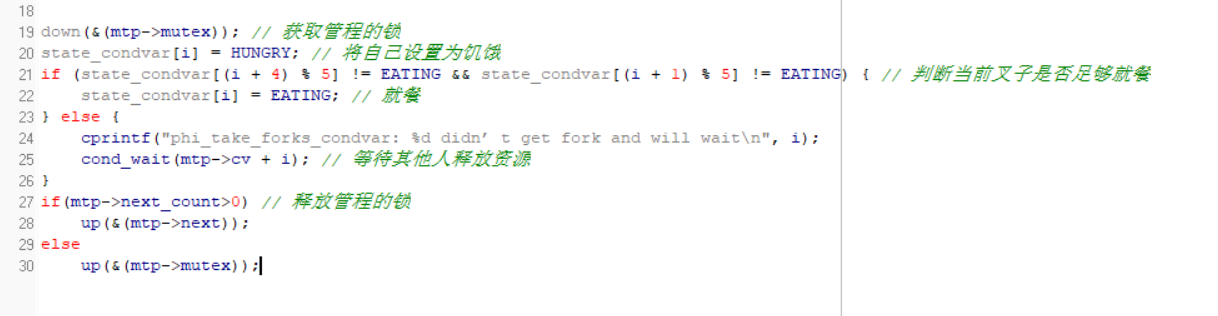
1.给管程上锁；

2.将哲学家的状态修改为HUNGER；

3.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）；

4.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；

5.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；

代码实现：

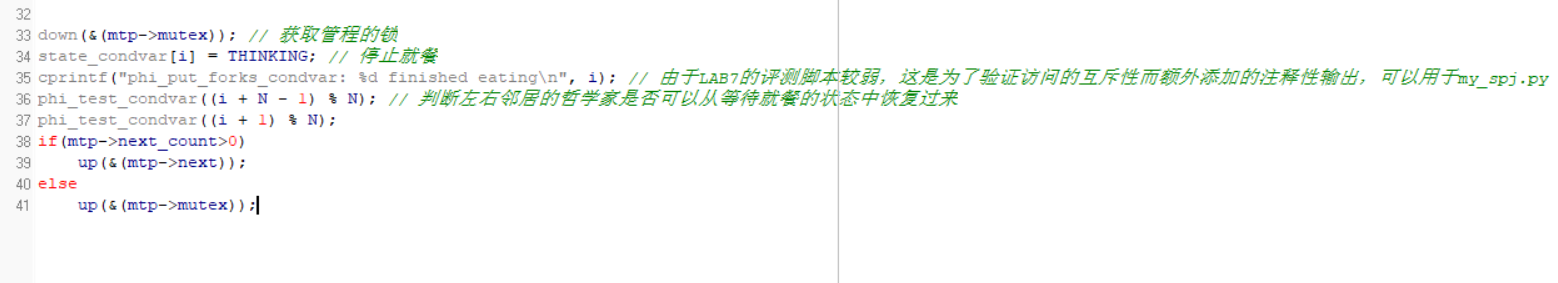
而phi\_put\_forks\_condvar函数则是释放当前哲学家占用的叉子，并且唤醒相邻的因为得不到资源而进入等待的哲学家：

1.首先获取管程的锁；

2.将自己的状态修改成THINKING；

3.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）；

4.释放锁，离开管程；

代码实现：

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。

问题：

请在实验报告中给出给用户态进程/线程提供条件变量机制的设计方案，并比较说明给内核级 提供条件变量机制的异同。

答：发现在本实验中管程的实现中互斥访问的保证是完全基于信号量的，也就是如果按照上文中的说明使用syscall实现了用户态的信号量的实现机制，那么就完全可以按照相同的逻辑在用户态实现管程机制和条件变量机制；

 当然也可以仿照用户态实现条件变量的方式，将对访问管程的操作封装成系统调用；

异同点为： 相同点：基本的实现逻辑相同；

不同点：最终在用户态下实现管程和条件变量机制，需要使用到操作系统使用系统调用提供一定的支持; 而在内核态下实现条件变量是不需要的；

请在实验报告中回答：能否不用基于信号量机制来完成条件变量？如果不能，请给出理由，如果能，请给出设计说明和具体实现。

答：1.能够基于信号量来完成条件变量机制；

2.事实上在本实验中就是这么完成的，只需要将使用信号量来实现条件变量和管程中使用的锁和等待队列即可