实验五 同步互斥

一、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。

二、实验内容

本次实验，主要是熟悉ucore的进程同步机制—信号量（semaphore）机制，以及基于信号量的哲学家就餐问题解决方案。然后掌握管程的概念和原理，并参考信号量机制，实现基于管程的条件变量机制和基于条件变量来解决哲学家就餐问题。

在本次实验中，在kern/sync/check\_sync.c中提供了一个基于信号量的哲学家就餐问题解法。

三、练习

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？

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。kern/check\_sync.c中实现的开关中断的控制函数local\_intr\_save(x)和local\_intr\_restore(x)，它们是基于kern/driver文件下的intr\_enable()、intr\_disable()函数实现的。具体调用关系为：

*关中断：local\_intr\_save* --> *\_\_intr\_save* --> *intr\_disable* --> *cli*

*开中断：local\_intr\_restore*--> *\_\_intr\_restore* --> *intr\_enable* --> *sti*

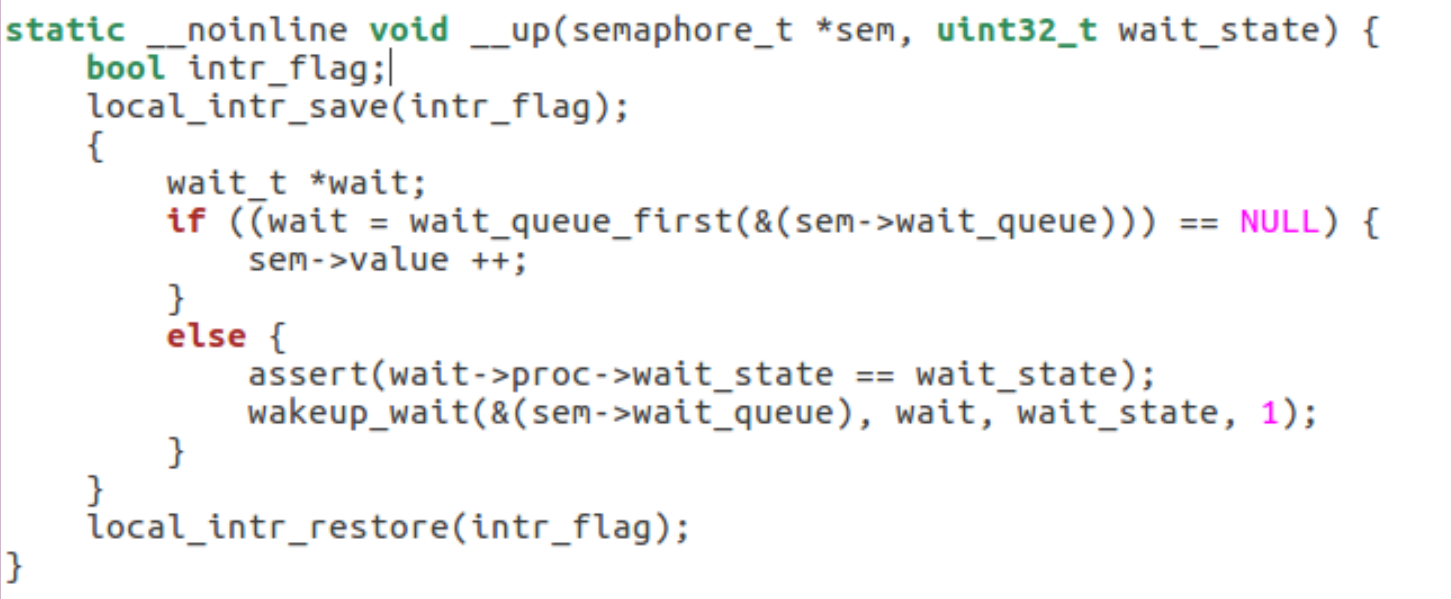
最终的cli和sti是x86的机器指令，最终实现了关中断和开中断，即设置了eflags寄存器中与中断相关的位。通过关闭中断，可以防止对当前执行的控制流被其他中断事件处理所打断。既然不能中断，那也就意味着在内核运行的当前进程无法被打断或被从新调度，即实现了对临界区的互斥操作。所以在单处理器情况下，可以通过开关中断实现对临界区的互斥保护，需要互斥的临界区代码的一般写法为：

local\_intr\_save(intr\_flag); { 临界区代码 }

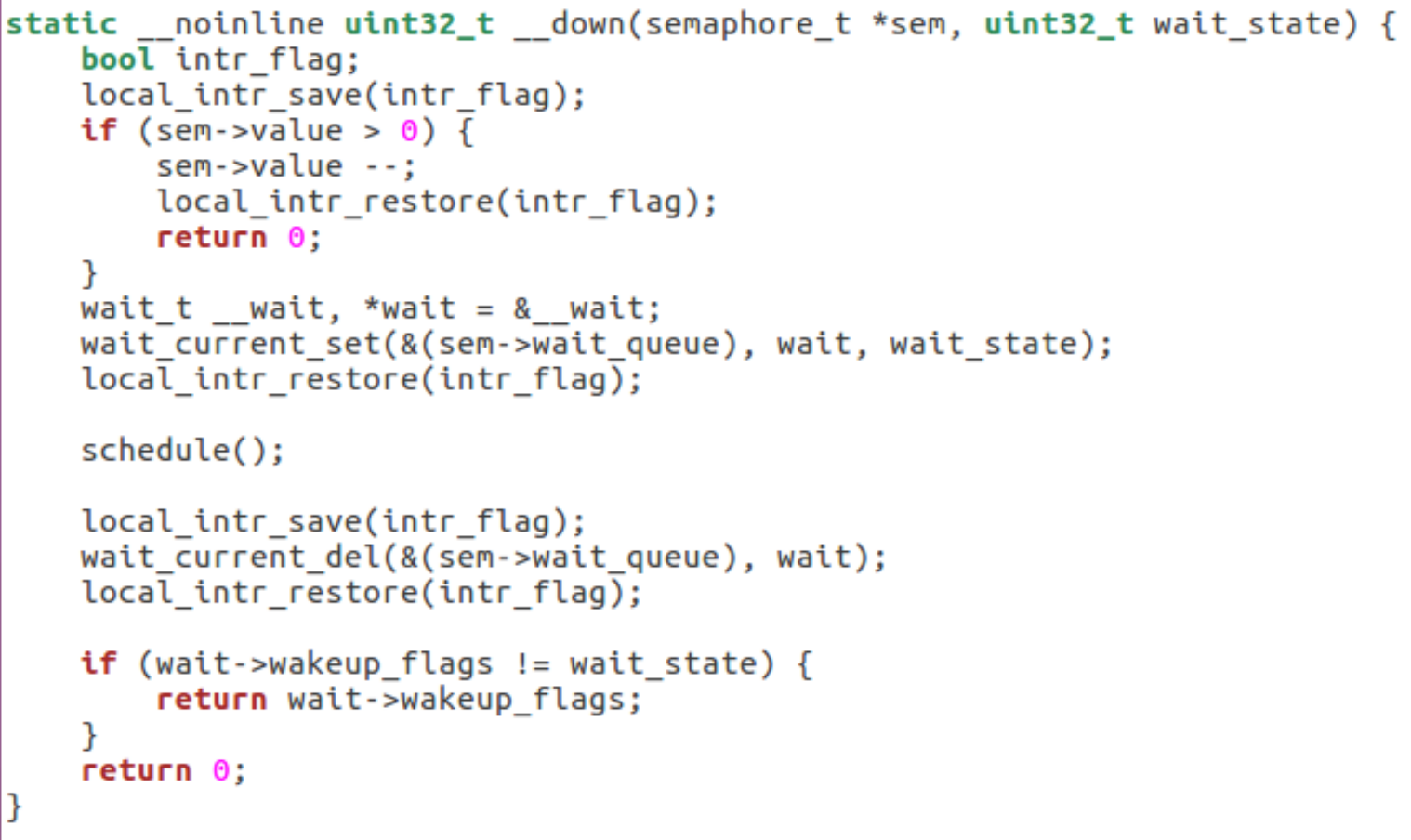
local\_intr\_restore(intr\_flag);

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：



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）。



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。

对照信号量的原理性描述和具体实现，可以发现二者在流程上基本一致，只是具体实现采用了关中断的方式保证了对共享资源的互斥访问，通过等待队列让无法获得信号量的进程睡眠等待。另外，我们可以看出信号量的计数器value具有有如下性质：

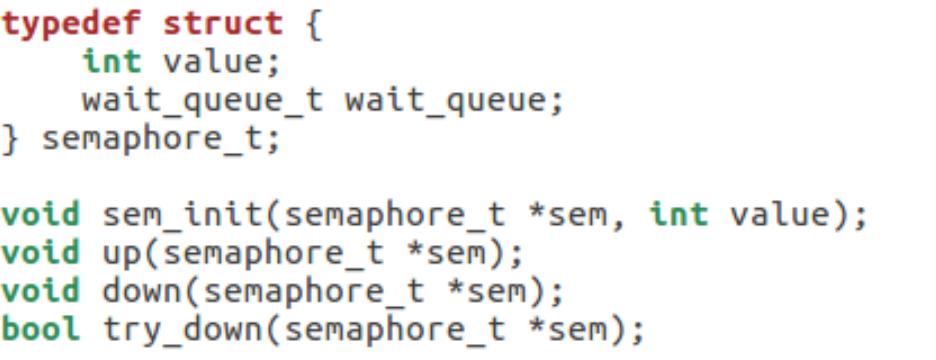
value>0，表示共享资源的空闲数

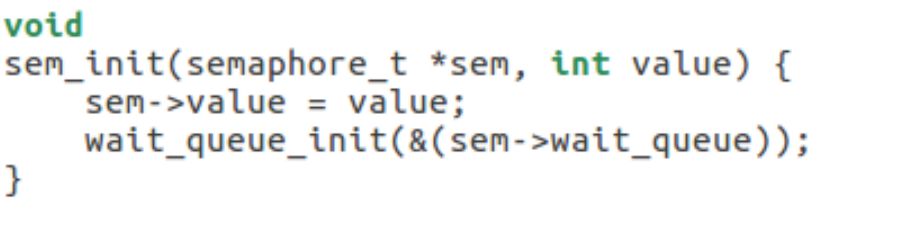
​vlaue<0，表示该信号量的等待队列里的进程数

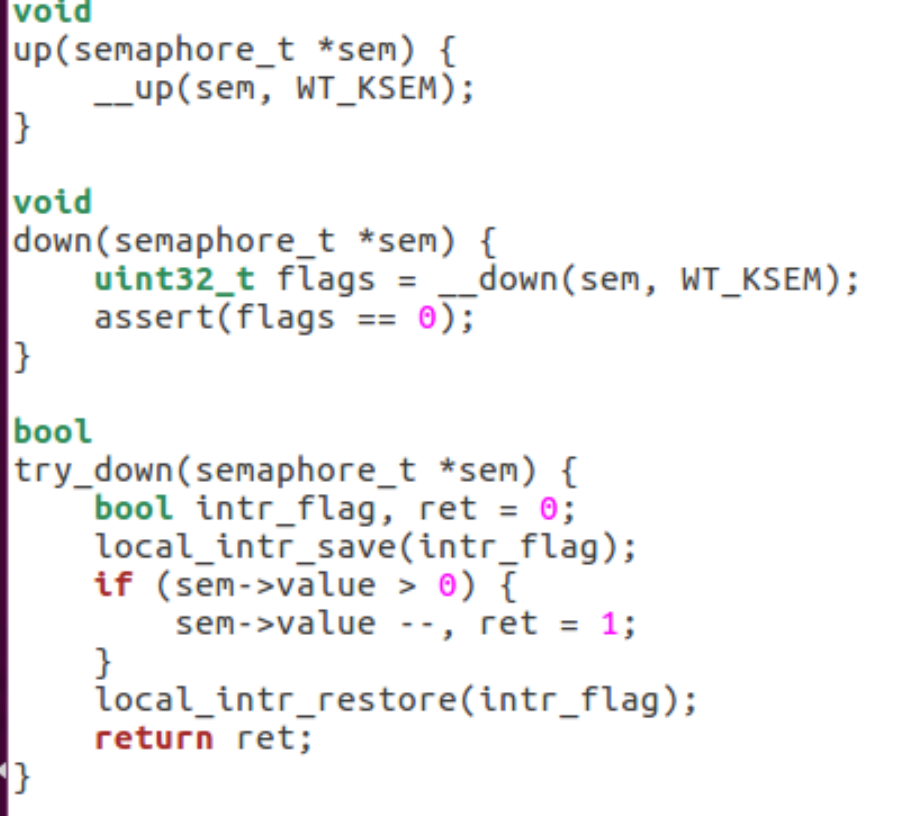
​value=0，表示等待队列为空

​

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。







down()函数：

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）。

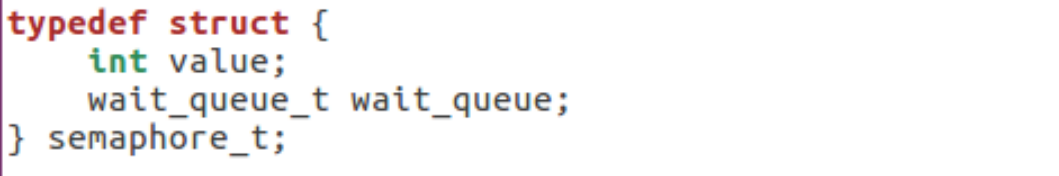
up()函数：

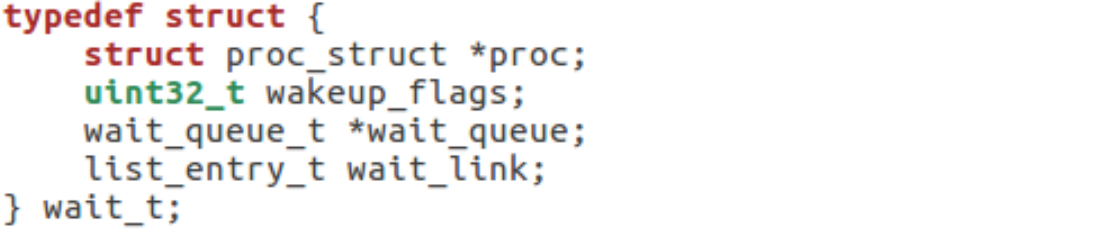
首先关中断，如果信号量对应的wait queue中没有进程在等待，直接把信号量的value加一，然后开中断返回；如果有进程在等待且进程等待的原因是semophore设置的，则调用wakeup\_wait函数将waitqueue中等待的第一个wait删除，且把此wait关联的进程唤醒，最后开中断返回。

练习2: 了解基于信号量和管程的哲学家就餐问题

1. 说明ucore中基于信号量的哲学家就餐问题的实现机制。

首先定义了一些数据结构，包括信号量、等待队列等





信号量的计数器value 含义如下：

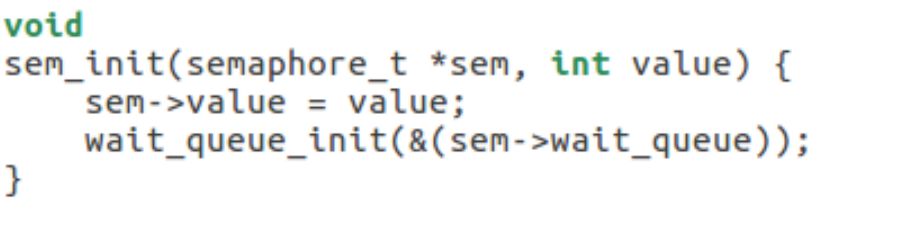
value>0， 表示共享资源的空闲数

vlaue<0， 表示该信号量的等待队列里的进程数

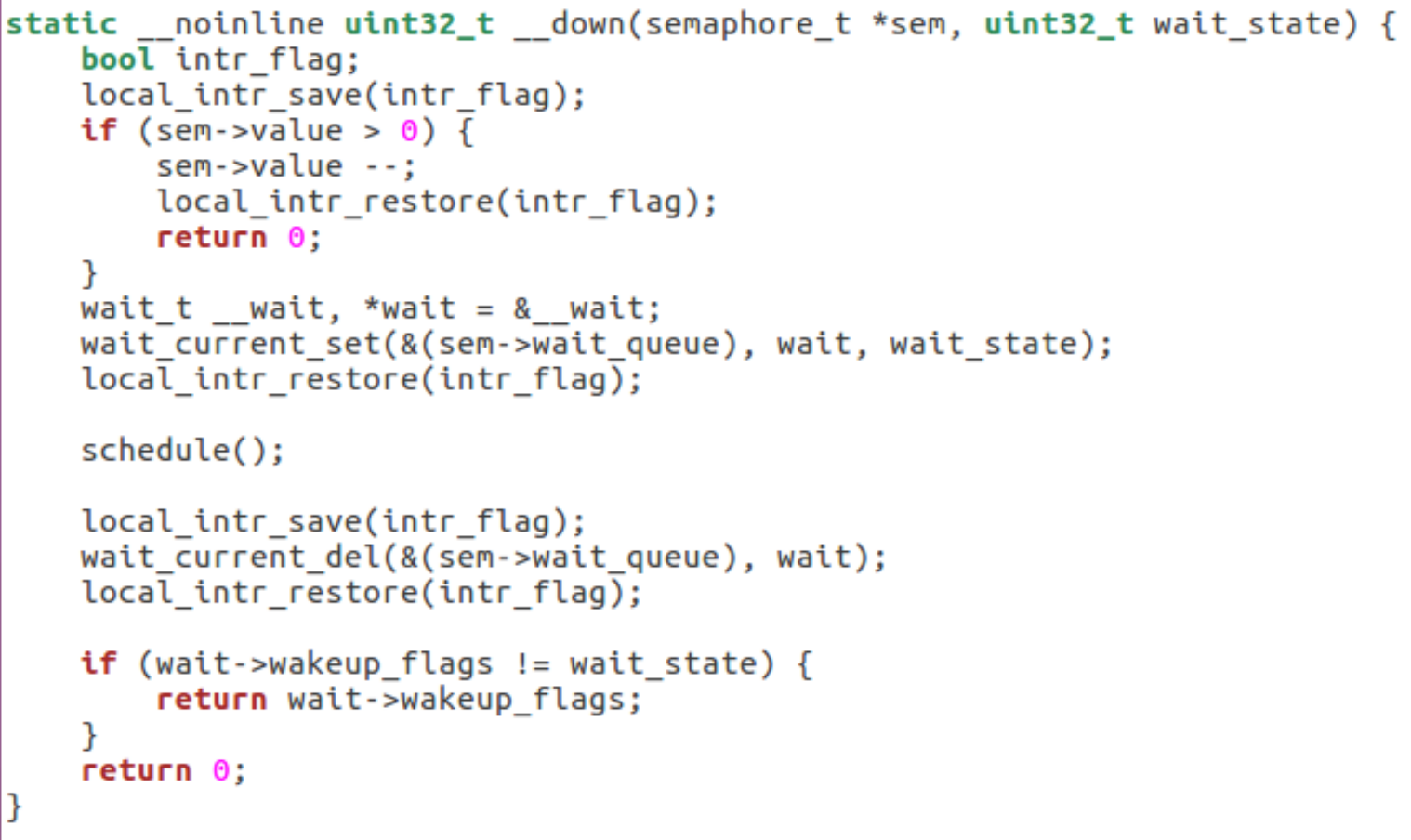
value=0， 表示等待队列为空

之后有相应的有关信号量的操作函数：

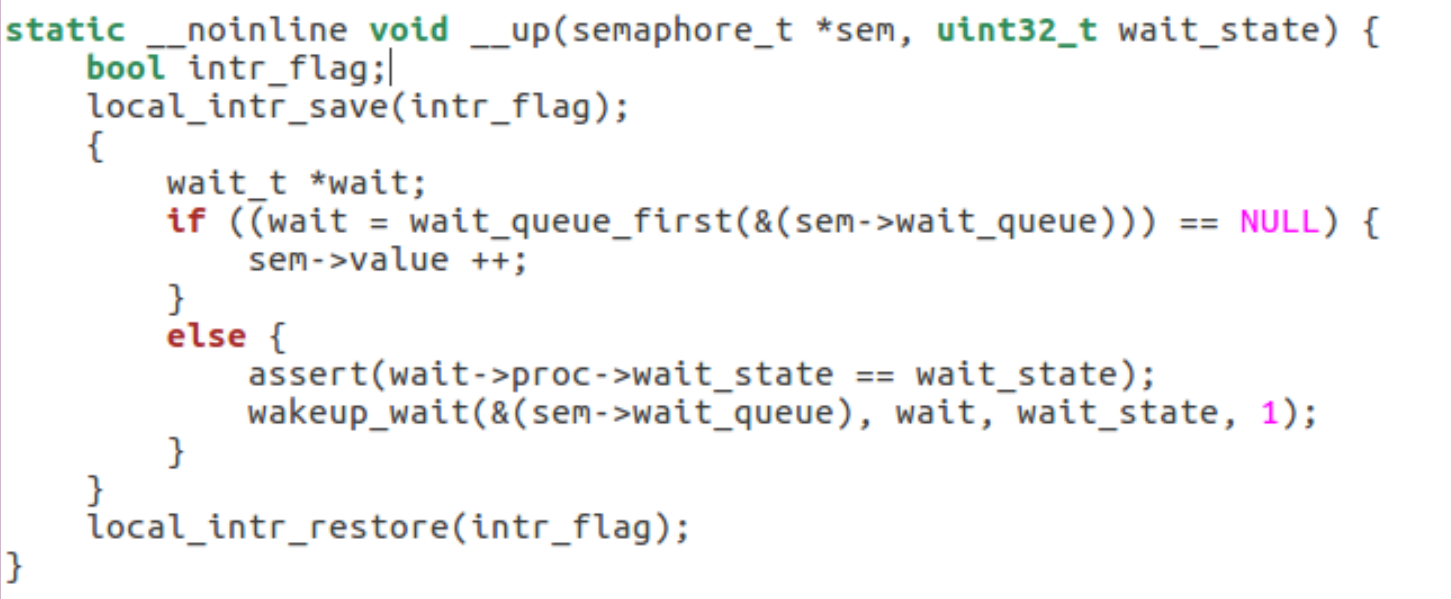
初始化函数 sem\_init:



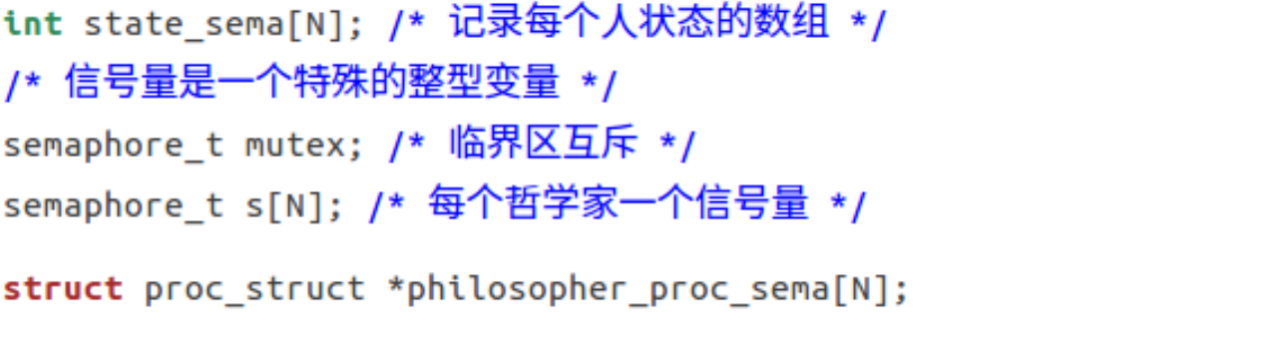
实现P操作，请求一个信号量对应的资源：



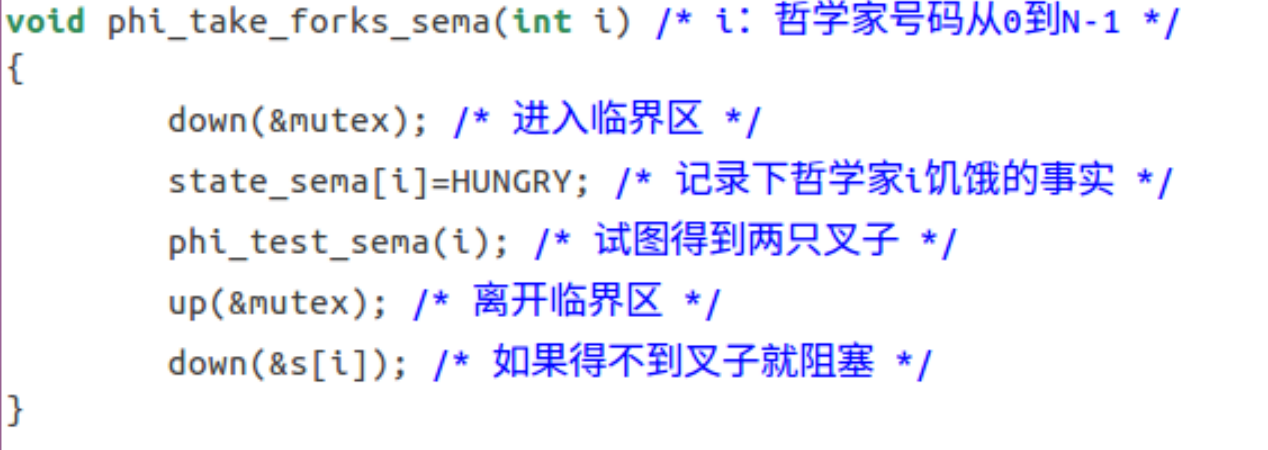
实现V操作，释放一个信号量资源：



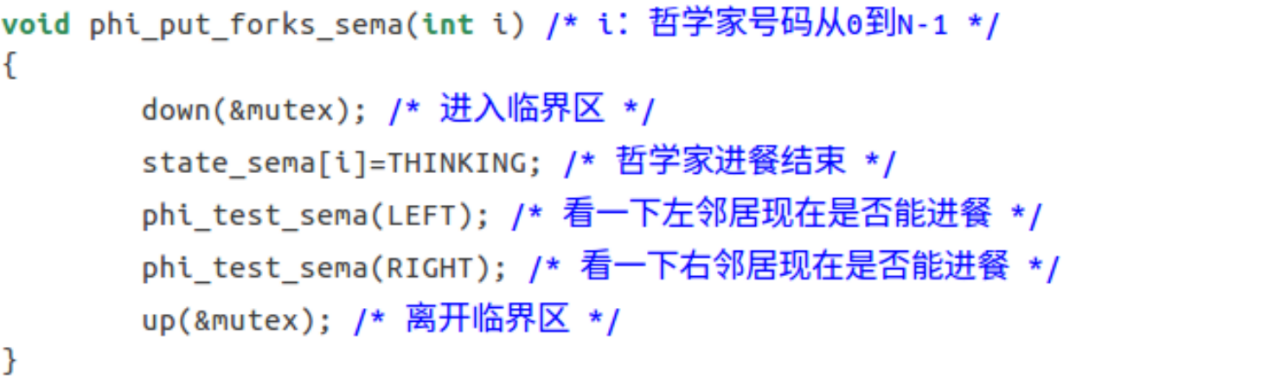
实现哲学家问题主要在check\_sync.c中：



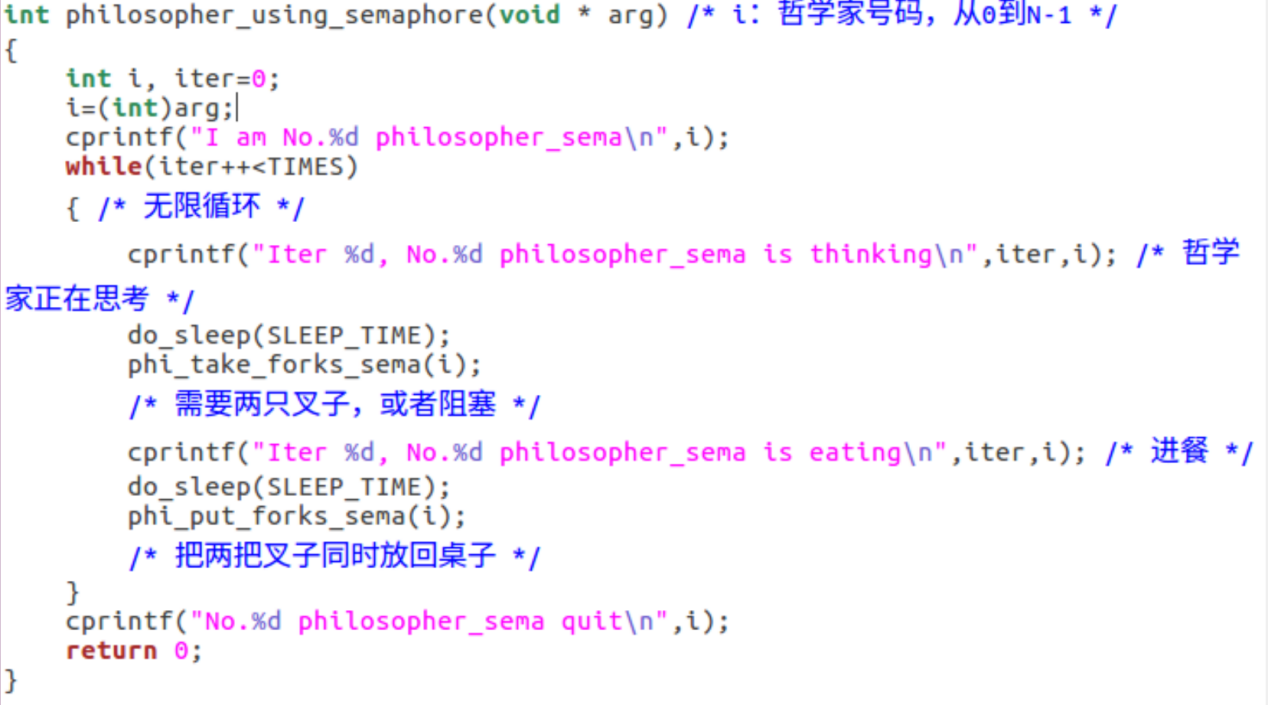
这里定义了哲学家的信号量数组；



哲学家拿起叉子；



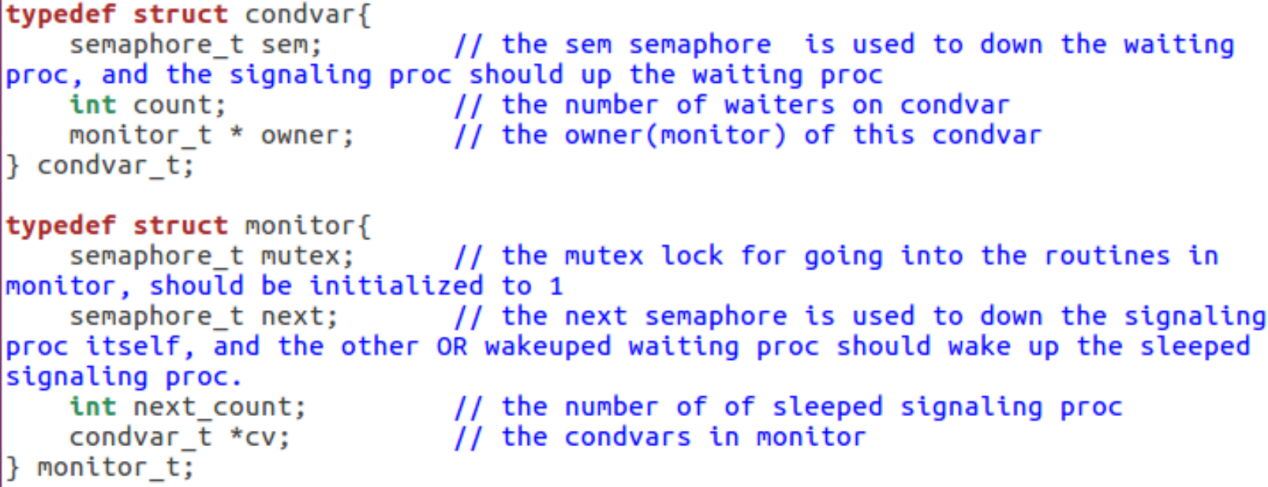
哲学家放下叉子；



主函数，具体分析哲学家是否拿起或放下叉子。

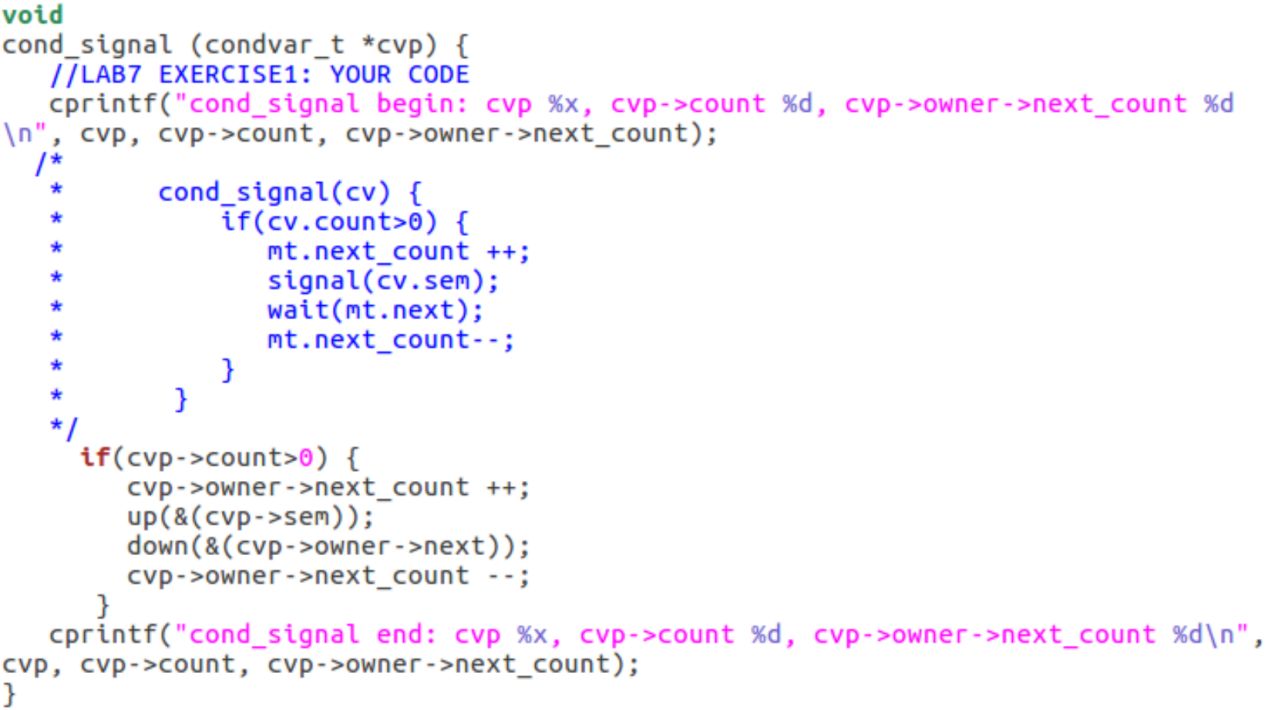
1. 说明ucore中基于管程的哲学家就餐问题的实现机制。

实现管程所构建的数据结构 monitor 以及 condvar



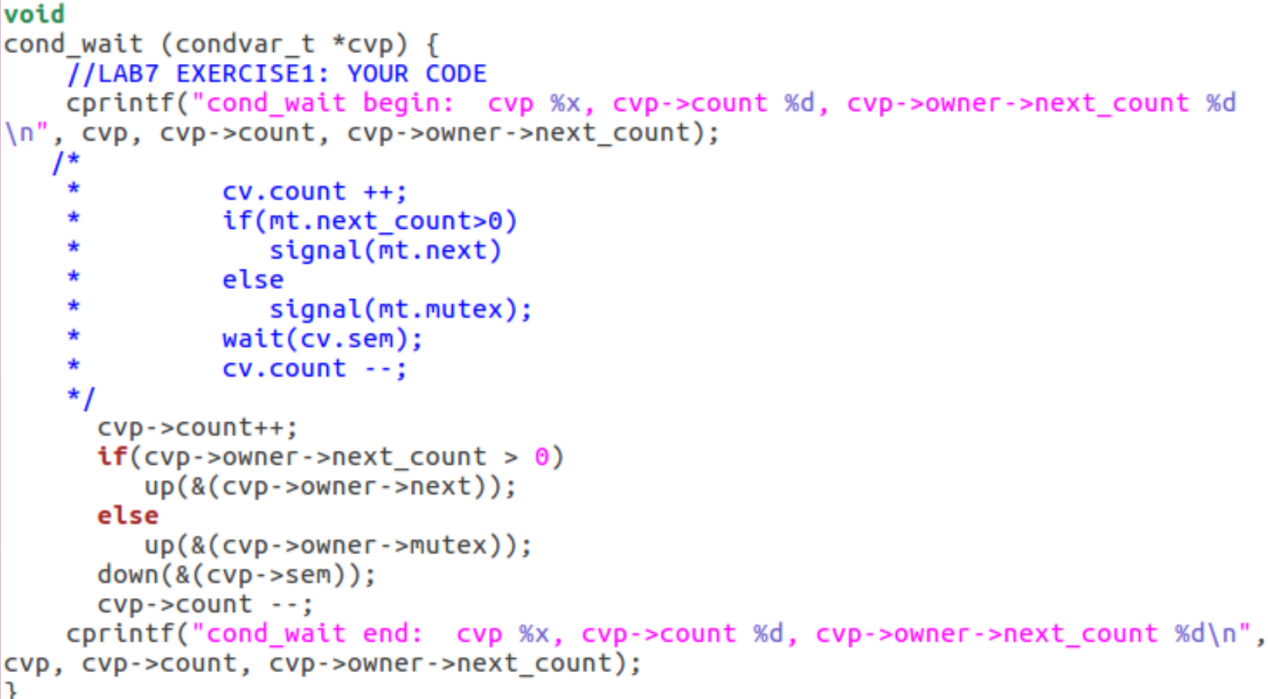
条件变量机制的实现主要是cond\_signal, cond\_wait两个函数中，分别表示提醒等待在这个条件变量上的进程恢复执行，以及等待在这个条件变量上，直到有其他进行将其唤醒为止.

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。该操作实现了对共享变量访问的互斥性；



执行cond\_signal函数时，首先当前进程测试cv.count，如果不大于0，则表示当前没有执行cond\_wait而进入休眠状态的进程，函数直接返回。如果cv.count大于0，这表示当前有执行cond\_wait而进入休眠状态的进程，因此需要唤醒在cv.sem上等待出入休眠状态的进程。由于只允许一个进程在管程中执行，所以一旦当前进程唤醒了其他进程，自身就要进入休眠状态，即增加monitor.next\_count，让当前进程在信号量monitor.next上进入休眠状态，直到被唤醒时，减少monitor.next\_count。

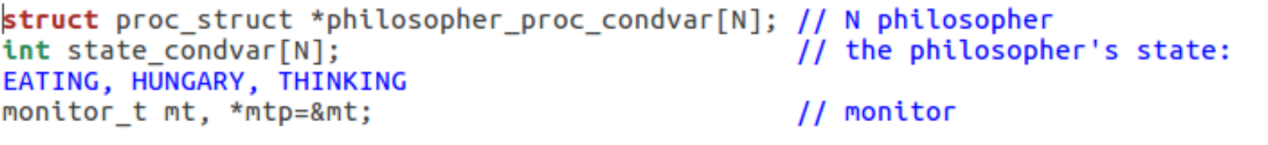
cond\_wait：该函数的功能为将当前进程等待在指定信号量上，其操作过程为将等待队列的计数加1，然后释放管程的锁或者唤醒一个next上的进程来释放锁，然后把自己等在条件变量的等待队列上，直到有signal信号将其唤醒，正常退出函数；



若某个进程执行了cond\_wait函数，表明该进程需因要的某个条件变量不满足需求而进入休眠状态。等待这个条件变量的休眠进程数cv.count增加。此时如果monitor.next\_count大于0，表示至少有1个进程执行cond\_signal函数后进入休眠状态，等待monitor.next信号量时，则需要唤醒等待该条件量的另一个进程，然后该进程在cv.sem上休眠。当进程A被唤醒时，减少cv.count，表示等待此条件变量的睡眠进程个数减少了一个，可继续执行。如果monitor.next\_count小于等于0，表明目前没有进程执行cond\_signal函数进入休眠，则需要唤醒的是由于互斥条件限制而无法进入管程的进程，即唤醒在monitor.mutex上休眠的进程。然后当前进程在cv.sem上休眠，直到被唤醒时，减少cv.count，表示等待此条件的睡眠进程个数减少，可继续执行。

实现上述函数后，管程基本已经实现，接下来分析本实验中基于条件变量和管程的哲学家就餐问题的实现:

实现哲学家问题主要在check\_sync.c中



这里创建了5个线程表示5个哲学家，哲学家尝试4次思考->拿叉子->吃饭->放下叉子。

具体分析实现:主要是phi\_take\_forks\_condvar函数以及phi\_put\_forks\_condvar函数

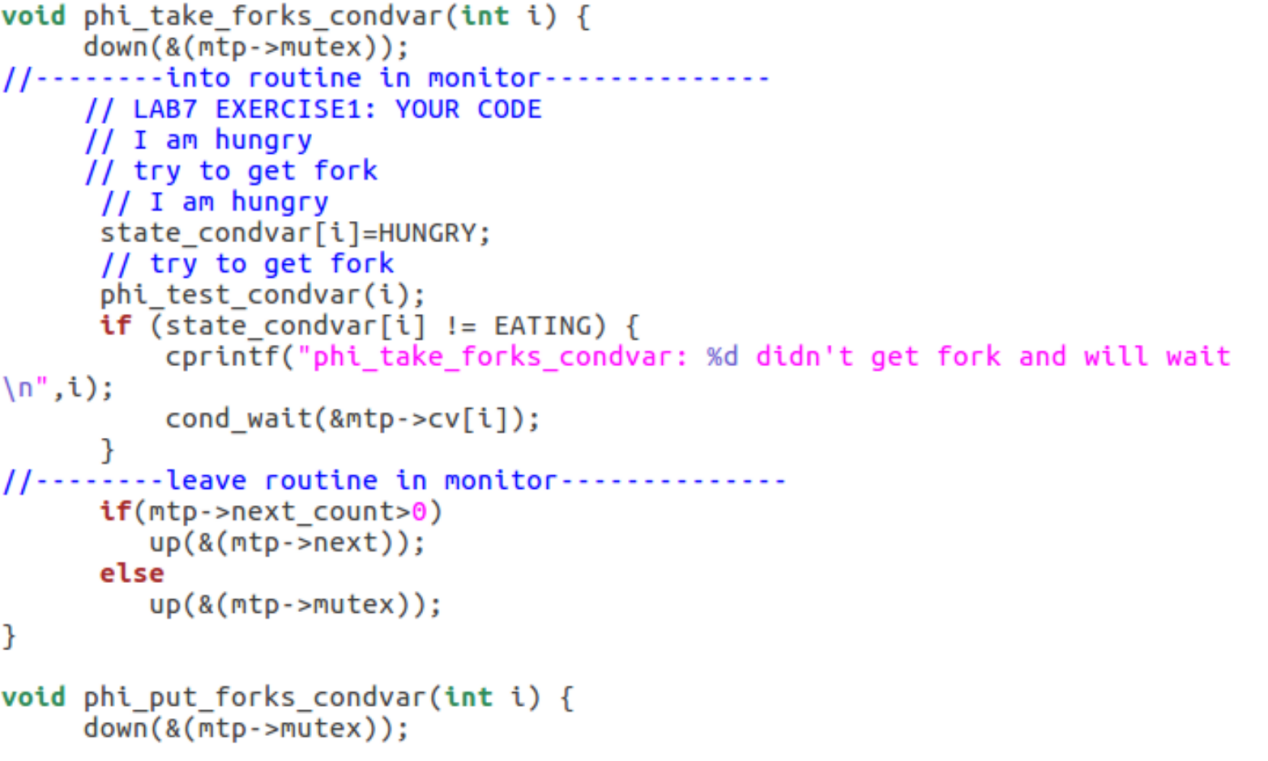
phi\_take\_forks\_condvar函数表示指定的哲学家尝试获得自己所需要进餐的两把叉子，如果不能获得则阻塞，具体实现流程为：

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；

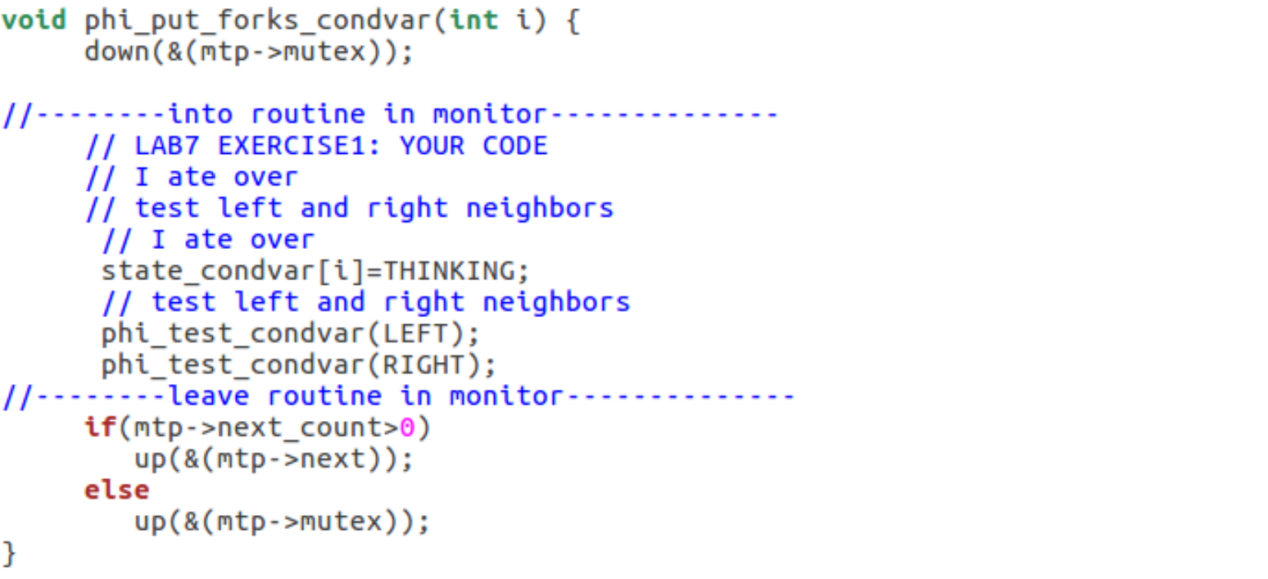


phi\_put\_forks\_condvar函数表示释放当前哲学家占用的叉子，并且唤醒相邻的因为得不到资源而进入等待的哲学家：

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

释放锁，离开管程



由于每个哲学家只可能占有所有需要的资源或者完全不占用资源，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；最终必定所有哲学将都能成功就餐