**实验五：同步互斥实验**

**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？

**开关中断**

根据操作系统原理的知识，如果没有在硬件级保证读内存-修改值-写回内存的原子性，我们只能通过复杂的软件来实现同步互斥操作。但由于有开关中断和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atomic.c文件中实现的test\_and\_set\_bit等原子操作。

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/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);

……

由于目前ucore只实现了对单处理器的支持，所以通过这种方式，就可简单地支撑互斥操作了。在多处理器情况下，这种方法是无法实现互斥的，因为屏蔽了一个CPU的中断，只能阻止本CPU上的进程不会被中断或调度，并不意味着其他CPU上执行的进程不能执行临界区的代码。所以，开关中断只对单处理器下的互斥操作起作用。在本实验中，开关中断机制是实现信号量等高层同步互斥原语的底层支撑基础之一。

**等待队列**

到目前为止，我们的实验中，用户进程或内核线程还没有睡眠的支持机制。在课程中提到用户进程或内核线程可以转入休眠状态以等待某个特定事件，当该事件发生时这些进程能够被再次唤醒。内核实现这一功能的一个底层支撑机制就是等待队列（wait queue），等待队列和每一个事件（睡眠结束、时钟到达、任务完成、资源可用等）联系起来。需要等待事件的进程在转入休眠状态后插入到等待队列中。当事件发生之后，内核遍历相应等待队列，唤醒休眠的用户进程或内核线程，并设置其状态为就绪状态（runnable state），并将该进程从等待队列中清除。ucore在kern/sync/{ wait.h, wait.c }中实现了wait结构和wait queue结构以及相关函数），这是实现ucore中的信号量机制和条件变量机制的基础，进入wait queue的进程会被设为睡眠状态，直到他们被唤醒。

typedef struct {

struct proc\_struct \*proc; //等待进程的指针

uint32\_t wakeup\_flags; //进程被放入等待队列的原因标记

wait\_queue\_t \*wait\_queue; //指向此wait结构所属于的wait\_queue

list\_entry\_t wait\_link; //用来组织wait\_queue中wait节点的连接

} wait\_t;

typedef struct {

list\_entry\_t wait\_head; //wait\_queue的队头

} wait\_queue\_t;

le2wait(le, member) //实现wait\_t中成员的指针向wait\_t 指针的转化

与wait和wait queue相关的函数主要分为两层，底层函数是对wait queue的初始化、插入、删除和查找操作，相关函数如下：

void wait\_init(wait\_t \*wait, struct proc\_struct \*proc); //初始化wait结构

bool wait\_in\_queue(wait\_t \*wait); //wait是否在wait queue中

void wait\_queue\_init(wait\_queue\_t \*queue); //初始化wait\_queue结构

void wait\_queue\_add(wait\_queue\_t \*queue, wait\_t \*wait); //把wait前插到wait queue中

void wait\_queue\_del(wait\_queue\_t \*queue, wait\_t \*wait); //从wait queue中删除wait

wait\_t \*wait\_queue\_next(wait\_queue\_t \*queue, wait\_t \*wait);//取得wait的后一个链接指针

wait\_t \*wait\_queue\_prev(wait\_queue\_t \*queue, wait\_t \*wait);//取得wait的前一个链接指针

wait\_t \*wait\_queue\_first(wait\_queue\_t \*queue); //取得wait queue的第一个wait

wait\_t \*wait\_queue\_last(wait\_queue\_t \*queue); //取得wait queue的最后一个wait

bool wait\_queue\_empty(wait\_queue\_t \*queue); //wait queue是否为空

高层函数基于底层函数实现了让进程进入等待队列，以及从等待队列中唤醒进程，相关函数如下：

//让wait与进程关联，且让当前进程关联的wait进入等待队列queue，当前进程睡眠

void wait\_current\_set(wait\_queue\_t \*queue, wait\_t \*wait, uint32\_t wait\_state);

//把与当前进程关联的wait从等待队列queue中删除

wait\_current\_del(queue, wait);

//唤醒与wait关联的进程

void wakeup\_wait(wait\_queue\_t \*queue, wait\_t \*wait, uint32\_t wakeup\_flags, bool del);

//唤醒等待队列上挂着的第一个wait所关联的进程

void wakeup\_first(wait\_queue\_t \*queue, uint32\_t wakeup\_flags, bool del);

//唤醒等待队列上所有的等待的进程

void wakeup\_queue(wait\_queue\_t \*queue, uint32\_t wakeup\_flags, bool del);

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。

* sem\_init：对信号量进行初始化的函数，根据在原理课上学习到的内容，信号量包括了等待队列和一个整型数值变量，该函数只需要将该变量设置为指定的初始值，并且将等待队列初始化即可；
* \_\_up：对应到了原理课中提及到的 V 操作，表示释放了一个该信号量对应的资源，如果有等待在了这个信号量上的进程，则将其唤醒执行；结合函数的具体实现可以看到其采用了禁用中断的方式来保证操作的原子性，函数中操作的具体流程为：
  + 查询等待队列是否为空，如果是空的话，给整型变量加 1；
  + 如果等待队列非空，取出其中的一个进程唤醒；
* \_\_down：同样对应到了原理课中提及的P操作，表示请求一个该信号量对应的资源，同样采用了禁用中断的方式来保证原子性，具体流程为：
  + 查询整型变量来了解是否存在多余的可分配的资源，是的话取出资源（整型变量减 1），之后当前进程便可以正常进行；
  + 如果没有可用的资源，整型变量不是正数，当前进程的资源需求得不到满足，因此将其状态改为 SLEEPING 态，然后将其挂到对应信号量的等待队列中，调用 schedule 函数来让出 CPU，在资源得到满足，重新被唤醒之后，将自身从等待队列上删除掉；
* up, down：对 \_\_up, \_\_down 函数的简单封装；
* try\_down：不进入等待队列的 P 操作，即时是获取资源失败也不会堵塞当前进程；

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

管程由四部分组成：

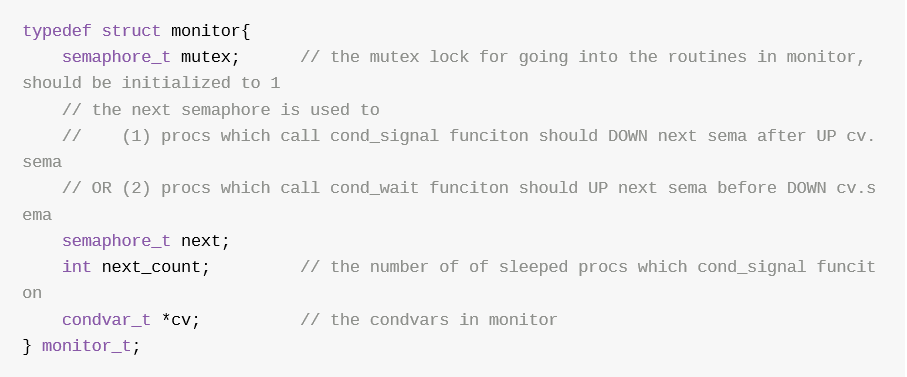
管程内部的共享变量；

管程内部的条件变量；

管程内部并发执行的进程；

对局部于管程内部的共享数据设置初始值的语句。

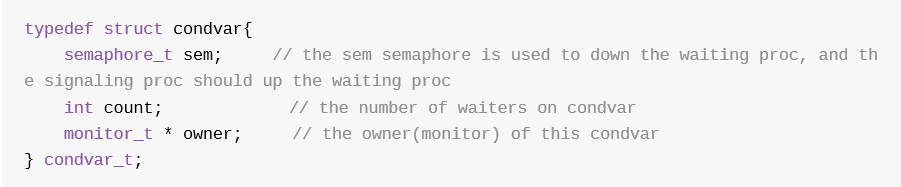
数据结构:



管程中的成员变量mutex是一个二值信号量，是实现每次只允许一个进程进入管程的关键元 素，确保了互斥访问性质。管程中的条件变量cv通过执行 wait\_cv ，会使得等待某个条件 Cond为真的进程能够离开管程并睡眠，且让其他进程进入管程继续执行；而进入管程的某进 程设置条件Cond为真并执行 signal\_cv 时，能够让等待某个条件Cond为真的睡眠进程被唤 醒，从而继续进入管程中执行。

注意：管程中的成员变量信号量next和整型变量next\_count是配合进程对条件变量cv的操作而 设置的，这是由于发出 signal\_cv 的进程A会唤醒由于 wait\_cv 而睡眠的进程B，由于管程中 只允许一个进程运行，所以进程B执行会导致唤醒进程B的进程A睡眠，直到进程B离开管程， 进程A才能继续执行，这个同步过程是通过信号量next完成的；而next\_count表示了由于发 出 singal\_cv 而睡眠的进程个数。

管程中的条件变量的数据结构condvar\_t定义如下：

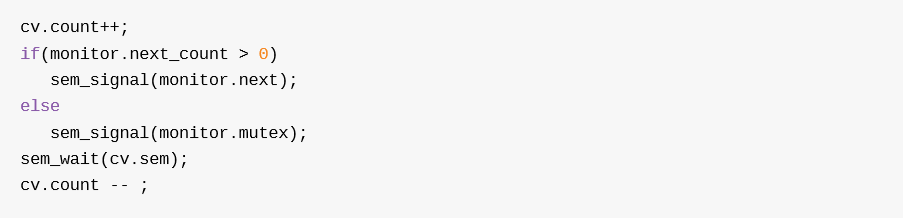


条件变量的定义中也包含了一系列的成员变量，信号量sem用于让发出 wait\_cv 操作的等待某 个条件Cond为真的进程睡眠，而让发出 signal\_cv 操作的进程通过这个sem来唤醒睡眠的进 程。count表示等在这个条件变量上的睡眠进程的个数。owner表示此条件变量的宿主是哪个 管程。

条件变量的signal和wait的设计

理解了数据结构的含义后，我们就可以开始管程的设计实现了。ucore设计实现了条件变 量wait\_cv 操作和 signal\_cv 操作对应的具体函数，即 cond\_wait 函数和 cond\_signal函数， 此外还有 cond\_init 初始化函数.

wait\_cv的原理描述:



对照着可分析出 cond\_wait 函数的具体执行过程。可以看出如果进程A执行了 cond\_wait 函 数，表示此进程等待某个条件Cond不为真，需要睡眠。因此表示等待此条件的睡眠进程个数 cv.count要加一。接下来会出现两种情况。

情况一：如果monitor.next\_count如果大于0，表示有大于等于1个进程执行cond\_signal函数且 睡了，就睡在了monitor.next信号量上（假定这些进程挂在monitor.next信号量相关的等待队列 Ｓ上），因此需要唤醒等待队列Ｓ中的一个进程B；然后进程A睡在cv.sem上。如果进程A醒 了，则让cv.count减一，表示等待此条件变量的睡眠进程个数少了一个，可继续执行了！

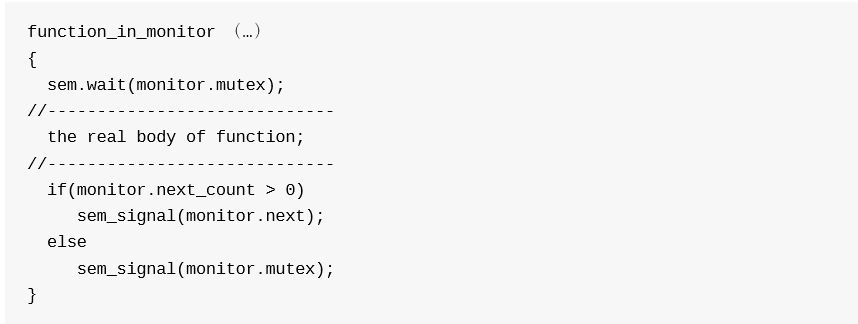
这里隐含这一个现象，即某进程A在时间顺序上先执行了 cond\_signal ，而另一个进程B 后执行了 cond\_wait ，这会导致进程A没有起到唤醒进程B的作用。 问题: 在cond\_wait有sem\_signal(mutex)，但没有看到哪里有sem\_wait(mutex)，这好像 没有成对出现，是否是错误的？ 答案：其实在管程中的每一个函数的入口处会有 wait(mutex)，这样二者就配好对了。

情况二：如果monitor.next\_count如果小于等于0，表示目前没有进程执行cond\_signal函数且 睡着了，那需要唤醒的是由于互斥条件限制而无法进入管程的进程，所以要唤醒睡在 monitor.mutex上的进程。然后进程A睡在cv.sem上，如果睡醒了，则让cv.count减一，表示等 待此条件的睡眠进程个数少了一个，可继续执行了！

对照着可分析出 cond\_signal 函数的具体执行过程。首先进程B判断cv.count，如果不大于0， 则表示当前没有执行cond\_wait而睡眠的进程，因此就没有被唤醒的对象了，直接函数返回即 可；如果大于0，这表示当前有执行cond\_wait而睡眠的进程A，因此需要唤醒等待在cv.sem上 睡眠的进程A。由于只允许一个进程在管程中执行，所以一旦进程B唤醒了别人（进程A）， 那么自己就需要睡眠。故让monitor.next\_count加一，且让自己（进程B）睡在信号量 monitor.next上。如果睡醒了，这让monitor.next\_count减一。

管程中函数的入口出口设计

为了让整个管程正常运行，还需在管程中的每个函数的入口和出口增加相关操作，即：



这样带来的作用有两个，（1）只有一个进程在执行管程中的函数。（2）避免由于执行了 cond\_signal函数而睡眠的进程无法被唤醒。对于第二点，如果进程A由于执行了cond\_signal 函数而睡眠（这会让monitor.next\_count大于0，且执行sem\_wait(monitor.next)），则其他进 程在执行管程中的函数的出口，会判断monitor.next\_count是否大于0，如果大于0，则执行 sem\_signal(monitor.next)，从而执行了cond\_signal函数而睡眠的进程被唤醒。上诉措施将使 得管程正常执行。

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。

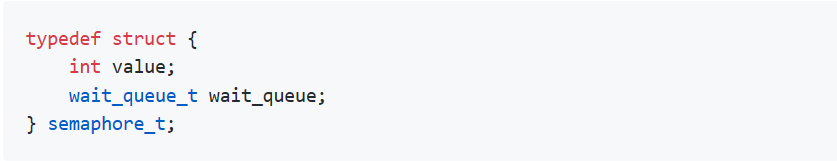
当多个进程可以进行互斥或同步合作时，一个进程会由于无法满足信号量设置的某条件而在某一位置停止，直到它接收到一个特定的信号（表明条件满足了）。为了发信号，需要使用一个称作信号量的特殊变量。为通过信号量 s 传送信号，信号量通过 V、P 操作来修改传送信号量。

count > 0，表示共享资源的空闲数，

count < 0，表示该信号量的等待队列里的进程数，

count = 0，表示等待队列为空

具体的信号量数据结构被定义在（kern/sync/sem.h）中：



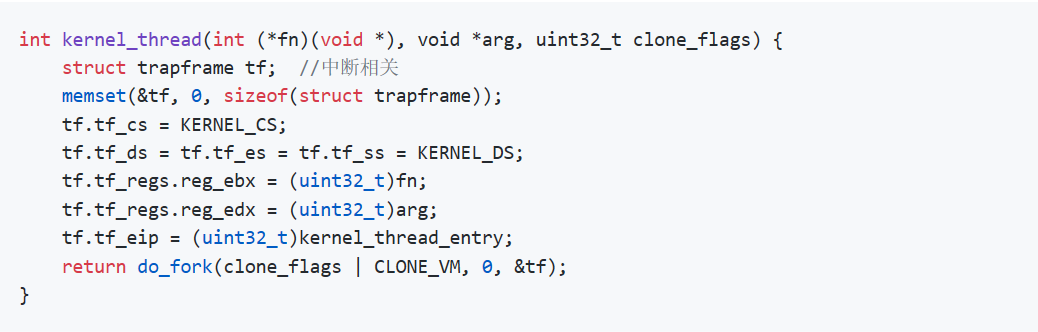
相关函数：



它调用了 check\_sync() 这个函数去执行了哲学家就餐问题。我们分析 check\_sync 函数（kern/sync/check\_sync.c）



这个 check\_sync 函数被分为了两个部分，第一部分使用了信号量来解决哲学家就餐问题，第二部分则是使用管程的方法。利用 kernel\_thread 函数创建了一个哲学家就餐问题的内核线程（kern/process/proc.c）



这个函数需要传入三个参数：

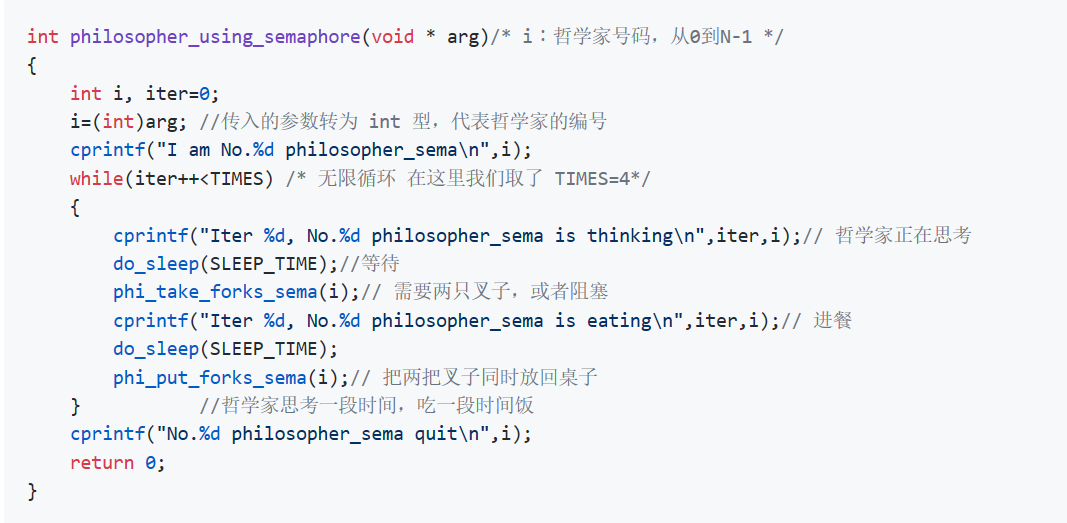
第一个 fn 是一个函数，代表这个创建的内核线程中所需要执行的函数；

第二个 arg 是相关参数，这里传入的是哲学家编号 i；

第三部分是共享内存的标记位，内核线程之间内存是共享的，因此应该设置为 0。

其余地方则是设置一些寄存器的值，保留需要执行的函数开始执行的地址，以便创建了新的内核线程之后，函数能够在内核线程中找到入口地址，执行函数功能。

内核线程去执行的目标函数 philosopher\_using\_semaphore ：



传入参数 \*arg，代表在上一个函数中“参数”部分定义的 (void \*)i，是哲学家的编号。

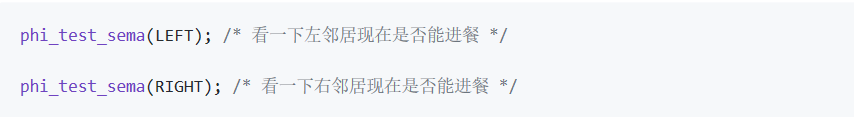
iter++<TIMES，表示循环 4 次，目的在于模拟多次试验情况。

哲学家需要思考一段时间，然后吃一段时间的饭，这里面的“一段时间”就是通过系统调用 sleep 实现的，内核线程调用 sleep，然后这个线程休眠指定的时间，从某种方面模拟了吃饭和思考的过程。

up 函数的作用是：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。

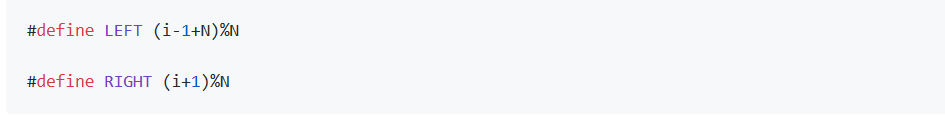
down 函数的作用是：首先关掉中断，然后判断当前信号量的 value 是否大于 0。如果是 >0，则表明可以获得信号量，故让 value 减一，并打开中断返回即可；如果不是 >0，则表明无法获得信号量，故需要将当前的进程加入到等待队列中，并打开中断，然后运行调度器选择另外一个进程执行。如果被 V 操作唤醒，则把自身关联的 wait 从等待队列中删除（此过程需要先关中断，完成后开中断）。

Test函数：



在试图获得筷子的时候，函数的传入参数为 i，即为哲学家编号，此时，他自己为 HUNGRY，而且试图检查旁边两位是否都在吃。如果都不在吃，那么可以获得 EATING 的状态。

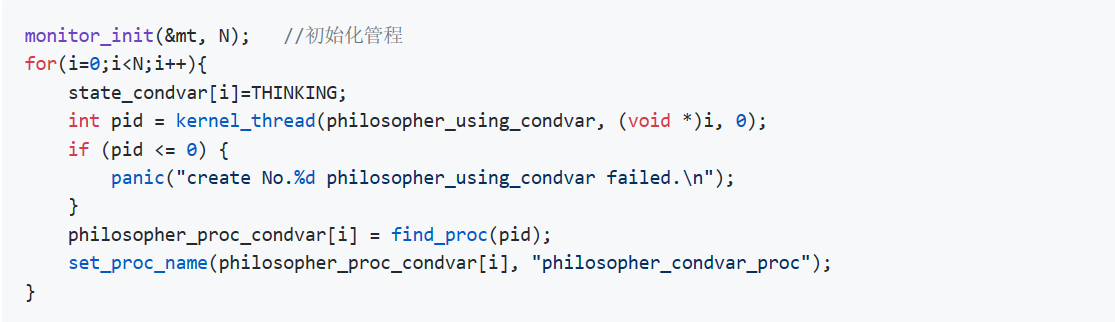
在从吃的状态返回回到思考状态的时候，需要调用两次该函数，传入的参数为当前哲学家左边和右边的哲学家编号，因为他试图唤醒左右邻居，如果左右邻居满足条件，那么就可以将他们设置为 EATING 状态。



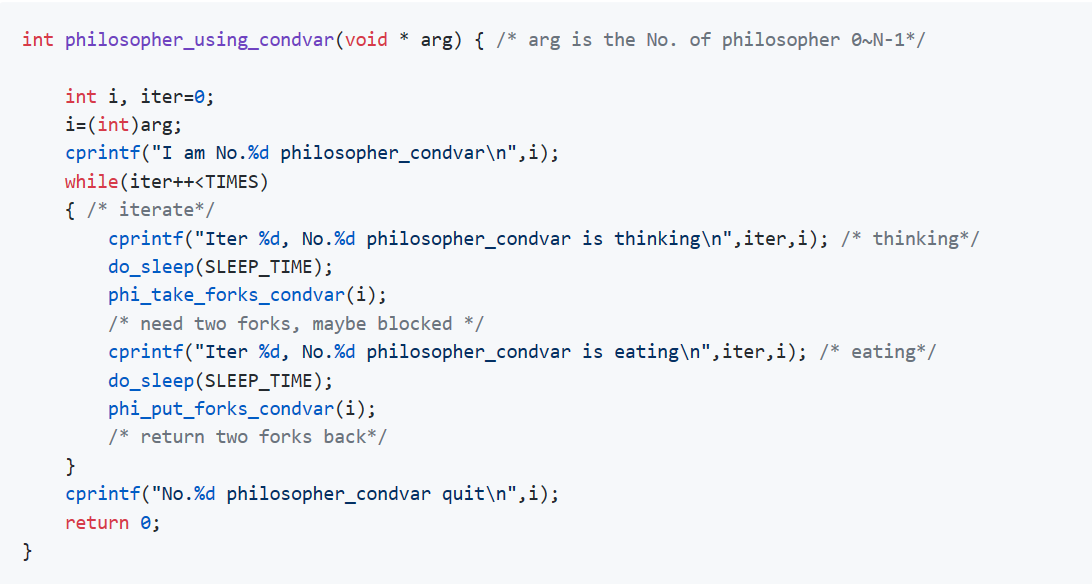
由于哲学家坐圆桌，因此可以使用余数直接获取左右编号。

1. 说明ucore中基于管程的哲学家就餐问题的实现机制。

在（kern/sync/check\_sync）实现：



这个实现过程和使用信号量无差别，不同之处在于，各个线程所执行的函数不同，此处执行的为 philosopher\_using\_condvar 函数：philosopher\_using\_condvar 函数被定义在（kern/sync/check\_sync）



这里和用信号量还是没有本质的差别，不同之处在于，获取筷子和放下都使用了不同的，配套管程使用的函数 phi\_take\_forks\_condvar 和 phi\_put\_forks\_condvar。

phi\_take\_forks\_condvar 和 phi\_put\_forks\_condvar 被定义在（kern/sync/check\_sync，）

其中，mtp 为一个管程， state\_convader 数组记录哲学家的状态。



这个地方的意思是，如果当前管程的等待数量在唤醒了一个线程之后，还有进程在等待，那么就会唤醒控制当前进程的信号量，让其他进程占有它，如果没有等待的了，那么直接释放互斥锁，这样就可以允许新的进程进入管程了。



和信号量的实现差不多，我们在拿起筷子和放下的时候，主要都还要唤醒相邻位置上的哲学家，但是，具体的test操作中，实现有所不同。



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；