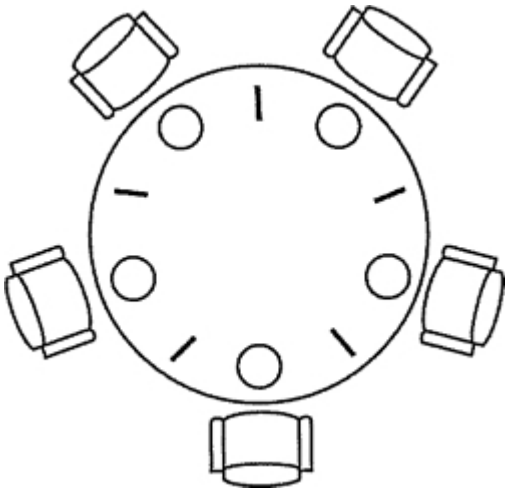
经典同步问题（二）---哲学家就餐问题

1. 问题描述：

一张圆桌上坐着 5 名哲学家，桌子上每两个哲学家之间摆了一根叉子，桌子的中间是一碗米饭，如图所示：



哲学家们倾注毕生精力用于思考和进餐，哲学家在思考时，并不影响他人。只有当哲学家饥饿的时候，才试图拿起左、右两根叉子（一根一根拿起）。如果叉子已在他人手上，则需等待。饥饿的哲学家只有同时拿到了两根叉子才可以开始进餐，当进餐完毕后，放下叉子继续思考。

2. 问题分析：

（1）5名哲学家与左右邻居对其中间叉子的访问是互斥关系。同时当哲学家要使用叉子时需要等待两个邻居都放下叉子才行，他们又是同步关系 （2）解决办法：这里要开5个线程，每个哲学家对应一个线程。最开始想到的办法是：每个哲学家先拿起左叉子，再拿起右叉子。并定义互斥信号量数组chopstick[5] = {1,1,1,1,1}用于对 5 根叉子的互斥访问。实例代码如下：

```
semaphore chopstick[5] = {1,1,1,1,1}  // 信号量数组,信号量初始化为1互斥访问每根叉子
Pi()                                  // i号哲学家的线程
{
    do
    {
        P(chopstick[i]);              // 取左
        P(chopstick[(i+1)%5]);        // 取右边叉子

        eat;                          // 进餐

        V(chopstick[i]);              // 放回左边叉子
        V(chopstick[(i+1)%5]);        // 放回右边叉子
        think;                        // 思考
    }while(1);
}
12345678910111213141516
```

但是这样会存在问题：如果每个哲学家同时拿起左叉子，都在等待右叉子时造成死锁。 为了防止死锁的发生，可以对哲学家线程施加一些限制条件，比如： （1）同时只允许一位哲学家就餐 （2）对哲学家顺序编号，要求奇数号哲学家先抓左边的叉子，然后再抓他右边的叉子，而偶数号哲学家刚好相反。 （3）仅当一个哲学家左右两边的叉子都可用时才允许他抓起叉子；

3. 使用信号量解决

以下代码都是伪代码 **3.1 方法1**

```
#define N 5
semaphore fork[5]={1,1,1,1,1};
semaphore mutex = 1;
void philosopher(int i){
   while(TRUE){
      think();
      P(mutex)
      P(fork[i]);
      P(fork[(i+1)%N);
      V(mutex)
      eat();
      V(fork[i]);
      V(fork[(i+1)%N];
    }1234567891011121314
```

这里把就餐（而不是叉子）看做是互斥访问的临界资源，因此会造成（叉子）资源的浪费。从理论上说，如果有五把叉子，应允许两个不相邻的哲学家同时进餐。 **3.2 方法2**

```
#define N 5
semaphore fork[5]={1,1,1,1,1};
void philosopher(int i){
   while(TRUE){
      think();
      if(i%2==1){
         P(fork[i]);
         P(fork[(i+1)%N);
      }else{
         P(fork[(i+1)%N]);
         P(fork[i]);
      }
      eat();
      V(fork[i]);
      V(fork[(i+1)%N];
    }12345678910111213141516
```

因为V操作不会阻塞，所以不需要分两种情况。 **3.3 方法3** 哲学家要么不拿，要么就拿两把叉子。那么哲学家就有三种状态：思考状态不用叉子、饥饿状态在等待左右叉子；吃饭状态正在使用叉子。

```
#define N 5
#define LEFT (i)
#define RIGHT (i+1)/N
#define EATTING 2
#define HUNGRY  1
#define THINKING 0
int state[N];
semaphore mutex;
semaphore s[N];
void philosopher(i)
{
    think(i);
    take_forks(i);    //吃饭前先等待两只叉子
    eatting();
    put_forks(i);     //放下叉子，查看左右邻居是否两只叉子都空闲，如果空闲提醒邻居拿起叉子
}
void take_forks(i)
{
   P(mutex)
   state[i] = HUNGRY;               //代表当前哲学家正在等待叉子
   test_take_left_right_forks(i);   //尝试是否能拿到叉子

   V(mutex);
   P(s[i]);                        //如果拿不到叉子就阻塞
}
void test_take_left_right_forks(i)
{
   if(state[i] == HUNGRY && state[LEFT] != EATTING && state[RIGTH] != EATTING)
   {
       state[i] = EATTING;    //用EATTING代表当前哲学家能拿到两只叉子
       V(s[i]);               //如果能够拿到两只叉子，唤醒当前线程
   }
}

void putdown(i)
{
  P(mutex)
  state[i] = THINKING;  //代表当前不需要叉子
  test_take_left_right_forks(LEFT);
  test_take_left_right_forks(RIGHT);
  V(mutex);
}

void thinking(i)
{
   P(mutex);
   state[i] = THINKING;
   V(mutex);
}12345678910111213141516171819202122232425262728293031323334353637383940414243444546474849
```

因为每个线程都要访问哲学家状态，这里把哲学家状态看成临界资源。并使用状态标记每个哲学家是否能拿到两只叉子。

3. 使用管程解决

方法**3**

```
#define N 5
#define LEFT (i)
#define RIGHT (i+1)/N
#define EATTING 2
#define HUNGRY  1
#define THINKING 0
int state[N];
lock mutex;
Condition c[N];
void philosopher(i)
{
    think(i);
    take_forks(i);
    eatting();
    put_forks(i);
}
void take_forks(i)
{
   lock.acqure();
   state[i] = HUNGRY;    //代表当前哲学家正在等待筷子，处于阻塞状态
   test_take_left_right_forks(i);   //尝试是否能拿到叉子
   while(state[i] != EATTING)
       c[i].wait(&lock);
   lock.release();

}
void test_take_left_right_forks(i)
{
   if(state[i] == HUNGRY && state[LEFT] != EATTING && state[RIGTH] != EATTING)
   {
       state[i] = EATTING;    //用EATTING代表当前哲学家能拿且会用叉子
       condition[i].signal();
   }

}

void putdown(i)
{
  lock.acquier();
  state[i] = THINKING;  //代表当前不需要筷子
  test_take_left_right_forks(LEFT);
  test_take_left_right_forks(RIGHT);
  lock.release();
}

void thinking(i)
{
   lock.acquier();
   state[i] = THINKING;
   lock.release();
}
```