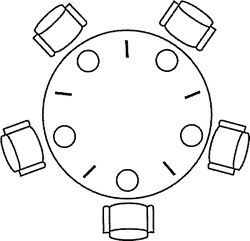
哲学家就餐问题之解

网络工程161 江彦君

引言：哲学家就餐问题是研究操作系统进程同步控制的经典模型。

摘要：如今良好的交互性、并行性、同步性早已是大多数计算机操作系统不可或缺的功能，无论是基于单核还是多核计算机，倘若没有以上几个性能，则毫无疑问是糟糕的操作系统。哲学家就餐问题是操作系统研究领域中一个著名而有趣的问题，是从计算机实现伪同步并行功能研究中演化而来的经典进程间通讯问题，对研究同步性有很大的帮助和启发。该文探究了哲学家问题的原理，并使用C语言对其进行了模拟。   
中国论文网 http://www.xzbu.com/8/view-4349388.htm  
关键词：操作系统同步性；死锁竞争；哲学家就餐问题   
　　中图分类号：TP311 文献标识码：A 文章编号：1009-3044（2013）15-3516-06   
　　1965年，著名的数学家Dijkstra提出了一个他称为“哲学家就餐”的问题，并亲自解决之。自那时起，每个发明同步原语的人都希望通过解决此问题来展示其同步原语的精妙之处。   
　　同步原语这一概念，被用于描述进程的执行动作，在操作系统中的地位举足轻重，其对实现操作系统程序的并行性和同步性有着奠基性的作用。那么，什么是同步原语呢？   
　　在计算机的操作系统中，程序一般被抽象成进程这一概念。在一台计算机中，所有存在的进程要实现计算、访问资源等一系列复杂的操作。   
　　当今计算机能够依靠相当少的CPU（相对于进程庞大的数量来说，一两个或者四个CPU简直少得可怜）实现程序并行的功能正是基于同步原语：计算机内的进程都被分为运行时间相同的原子执行动作，即一个复杂的动作由许多子动作组成，这些子动作不能再分成更小的子动作。同步原语中的原语便是指这些子动作。   
　　对于CPU来说，每次只执行一个原子动作，这个原子动作不一定是来自于相同的进程，一般按照某种规则来运行原子动作，比如在一段时间内，一个进程只能占用CPU运行若干个属于它自已的原子动作。由于CPU的运行速度非常快，当下普通CPU其一秒内的运算速度都可以以亿为单位计量，因此在一段时间内，可以让很多进程都完成一部分原子动作，在人类的角度看来，就好像程序并行运行了。   
　　在这里，就可以看到同步原语的必要性。若无同步原语的话，进程便会胡作非为了，因为计算机没有规定一个进程能执行动作到何种地步，对于进程间的竞争问题和通讯问题，即使有了很好的策略也无法控制进程执行的结果。例如，对于一个缓冲区，一个进程获得了它的修改权，但是操作系统没有确定原子动作，这个进程不知道会执行到哪一步，于是，在操作系统还没有来得及告知其他进程时，就可能会有另一个无知的进程也获得了该缓冲区的修改权，从而与前一个进程一同修改缓冲区的内容，这将会导致世界大乱。   
　　同时，毫无疑问，在这里，存在一个很现实的问题。   
　　正如上所述，计算机中的进程数往往多得惊人，光是操作系统本身就需要并行运行众多进程，再加上用户开启的各种应用进程（比如用户想听MP3歌曲，便开启了一个MP3播放器，这一个播放器可能包含不止一个进程），其数目可能会非常大。然而，计算机的硬件设备却极其有限，例如，目前的普通计算机只有一或者两个中央处理器（CPU），只有有限的内存，只有一个显示器，这么多进程要同时使用这些相对来说少得可怜的硬件设备，若没有一个优秀的访问机制，程序间便会出乱子。例如客户开启了一个文本编辑进程用来编写一篇文章，在某个时候，这篇文章的代码被保存在内存中的一些位置上，过了一段时间，这个客户又开启了一个视频播放器，而原先的文本编辑进程仍没有关闭，播放器要使用的内存位置正好与编辑器所占用的位置有冲突，此时该如何是好？当文本编辑程序不适用CPU而让视屏播放器使用CPU时，该如何保存文本编辑器的记录？文本编辑器修改了内存上的内容，视屏播放器如何知道？如果这些进程不相互沟通各自的信息，那相信该用户不久便会精神崩溃。   
　　因此，进程间必须实现良好的信息共享策略，哲学家就餐问题便是研究这类策略的一个经典抽象数学问题。[[1]](#endnote-1)

**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);

}

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。

**3 求解方法**  
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；

##### **使用信号量解决**

以下代码都是伪代码   
**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) /\*利用信号量的保护机制实现。

原理：通过互斥信号量 mutex 对 eat() 之前取左侧和右侧筷子的操作进行保护，可以防止死锁的出现。\*/[[2]](#endnote-2)

P(fork[i]);

P(fork[(i+1)%N);

V(mutex)

eat();

V(fork[i]);

V(fork[(i+1)%N];

}

* 1

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];

}

因为V操作不会阻塞，所以不需要分两种情况。   
**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);

}

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。

##### **使用管程解决**

**方法4**

#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();

}[[3]](#endnote-3)

#### 3总结

本文对哲学家进餐问题产生死锁的现象进行了分析,提出了3种解决方案, 并从在理论依据、算法设计、编程实现等方面进行了较为详细地阐述。这对学习和理解《操作系统原理》课程中的经典进程同步问题有一定的参考价值。

参考文献

1、Linux经典问题—五哲学家就餐问题

<https://blog.csdn.net/aspenstars/article/details/70149038>

2、ISSN 1009-3044 电脑知识与技术 Vol No.15 May 2013(中山大学，广东 广州 510275, 解晨)

3、经典同步问题（二）---哲学家就餐问题

<https://blog.csdn.net/sunny_ss12/article/details/47423363>

4、java多线程--哲学家就餐问题 <https://blog.csdn.net/qq1004642027/article/details/50055917>

5、从哲学家进餐问题分析死锁

<http://blog.sina.com.cn/s/blog_c33b15000102x6qh.html>

1. 原文地址：<http://www.xzbu.com/8/view-4349388.htm> [↑](#endnote-ref-1)
2. 原文地址：http://blog.sina.com.cn/s/blog\_e33e27b70102w68z.html [↑](#endnote-ref-2)
3. 引自<https://blog.csdn.net/sunny_ss12/article/details/47423363> sunny\_ss的专栏 [↑](#endnote-ref-3)