**电子科技大学**信息与软件工程**学院**

**标 准 实 验 报 告**

**（实验）课程名称 操作系统基础实验**

**电子科技大学教务处制表**

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名： 成棋伟 学 号：2022130102013**

**指导教师： 刘瑶**

**实验地点：西306 实验时间： 2025年4月28日**

1. **实验室名称：**

信软楼西306

1. **实验项目名称：**

生产者/消费者问题的实现

1. **实验学时：**

4学时

1. **实验原理：**

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

1、Linux信号量机制

操作系统中需要一个地位高于进程的管理者来解决公有资源的使用问题，信号量就是操作系统提供的管理公有资源的有效手段，作为操作系统核心代码执行，不受进程调度的打断，信号量代表可用资源实体的数量。

每个信号量s除一个整数值s.count（计数）外，还有一个进程阻塞队列s.queue，信号量只能通过初始化和两个标准的原语（P操作，V操作）来访问，初始化信号量指定s.count为一个非负整数值，表示空闲资源总数（又称为“资源信号量”），在程序运行中如果s.count若为非负值表示当前的空闲资源数，若为负值其绝对值表示当前等待临界区的进程数。二进制信号量：只允许信号量取0或1值（此时信号量功能和互斥量类似）。

在Linux系统中，使用信号量通常分为以下4个步骤：

   ① 创建信号量或获得在系统中已存在的信号量，此时需要调用 semget() 函数。不同进程通过使用同一个信号量键值来获得同一个信号量。

   ② 初始化信号量，此时使用 semctl() 函数的SETVAL操作。当使用二维信号量时，通常将信号量初始化为1。

   ③ 进行信号量的PV操作，此时，调用 semop()函数。这一步是实现进程间的同步和互斥的核心工作部分。

   ④ 如果不需要信号量，则从系统中删除它，此时使用semctl()函数的 IPC\_RMID操作。需要注意的是，在程序中不应该出现对已经被删除的信号量的操作。

2、Linux共享内存机制

共享内存是内核为进程创建的一个特殊内存段，它将出现在自己的地址空间中，其它进程可以将同一段共享内存连接(attach)到自己的地址空间，它是最快的进程间通信方式，不提供任何同步功能。

对共享内存的主要操作如下：

打开或创建共享内存

int semget(key\_t key, int size, int flag);

将共享内存连接到进程空间

void \*shmat(int shmid, void \*addr, int flag);

断开进程空间和共享内存的连接

int shmdt(void \*addr);

共享内存控制操作

int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);

1. **实验目的：**

* 掌握进程、线程的概念，熟悉相关的控制语；
* 掌握进程、线程间的同步原理和方法；
* 掌握进程、线程间的互斥原理和方法。

1. **实验内容：**

有一群生产者线程在生产产品，并将这些产品提供给消费者进程去消费。为使生产者线程与消费者进程能并发执行，在两者之间设置了一个具有n个缓冲区的缓冲池：生产者线程从文件中读取一个数据，并将它存放到一个缓冲区中； 消费者线程从一个缓冲区中取走数据，并输出此数据。生产者和消费者之间必须保持同步原则：不允许消费者进程到一个空缓冲区去取产品；也不允许生产者线程向一个已装满产品且尚未被取走的缓冲区中投放产品。

创建3线程作为生产者，4个线程作为消费者。创建一个文件作为数据源，文件中事先写入一些内容作为数据。

生产者和消费者线程都具有相同的优先级。

1. **实验器材（设备、元器件）：**

PC计算机，操作系统：Ubuntu

1. **实验步骤：**

1.编写多线程完成生产者/消费者问题程序

3.使用gcc将.c文件编译成可执行文件

4.在虚拟机linux环境下执行程序观察结果

1. **实验数据及结果分析：**

线程程序实现如下：

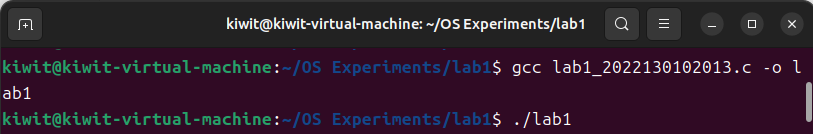
|  |
| --- |
| #include <stdio.h>  #include <unistd.h>  #include <semaphore.h>  #include <pthread.h>  #include <stdlib.h>  // 宏定义  #define PRO\_NUM 3 // 生产者数量为3  #define CON\_NUM 4 // 消费者数量为4  #define FILE\_NAME "input.txt" // 文件名  #define BUF\_NUM 7 // 缓冲区大小  sem\_t full, empty; // 定义两个信号量  pthread\_mutex\_t mutex; // 定义互斥锁  int buf[BUF\_NUM]; // 缓冲区数组  int producer\_index[PRO\_NUM] = {0, 1, 2}; // 生产者编号数组  int consumer\_index[CON\_NUM] = {0, 1, 2, 3}; // 消费者编号数组  int isfinish = PRO\_NUM; // 定义生产者任务完成数量  int in = 0, out = 0; // 缓冲区的指针  // 生产者线程函数  void producer(int\* arg){      int i = \*arg; // 获取当前生产者的编号      FILE \*f = fopen(FILE\_NAME, "r"); // 打开指定的文件      while(1){          sem\_wait(&empty); // 空缓冲区信号量减1，若已经为0则等待          pthread\_mutex\_lock(&mutex); // 上锁，保证互斥访问          int ch = getc(f); // 从文件中读取一个字符          if(ch == EOF){ // 如果是文件末尾，任务结束              printf("producer%d is over\n", i);              pthread\_mutex\_unlock(&mutex); // 解锁              sem\_post(&empty); // 空缓冲区信号量加1              isfinish--; // 任务完成数量减1              break;          }          printf("producer%d:\"%c\" in:%d\n", i, ch, in); // 输出生产者编号和字符          buf[in] = ch; // 将读取的字符存入缓冲区          in = (in + 1) % BUF\_NUM; // 缓冲区指针后移          pthread\_mutex\_unlock(&mutex); // 解锁          sem\_post(&full); // 满缓冲区信号量加1          usleep(3000); // 线程挂起一段时间      }      fclose(f); // 关闭文件  }  // 消费者线程函数  void consumer(int\* arg){      int i = \*arg; // 获取当前消费者的编号      while(1){          if(isfinish == 0){ // 如果所有生产者的任务都完成了，退出循环              printf("consumer%d is over\n", i);              break;          }          if(sem\_trywait(&full) != 0){ // 尝试将满缓冲区信号量减1，若已经为0则立即返回错误              continue; // 若无法减1则跳过本次循环          }          pthread\_mutex\_lock(&mutex); // 上锁          int ch = buf[out]; // 取出缓冲区中的一个字符          printf("consumer%d:\"%c\" out:%d\n", i, ch, out); // 输出消费者编号和字符          out = (out + 1) % BUF\_NUM; // 缓冲区指针后移          pthread\_mutex\_unlock(&mutex); // 解锁          sem\_post(&empty); // 空缓冲区信号量加1          usleep(4000); // 线程挂起一段时间      }  }  // 主函数  int main(){      pthread\_mutex\_init(&mutex, NULL); // 初始化互斥锁      pthread\_t consumers[CON\_NUM]; // 定义消费者线程数组      pthread\_t producers[PRO\_NUM]; // 定义生产者线程数组      sem\_init(&empty, 0, BUF\_NUM); // 初始化空缓冲区信号量为缓冲区大小      sem\_init(&full, 0, 0); // 初始化满缓冲区信号量为0      for(int i = 0; i < PRO\_NUM; i++){ // 创建生产者线程          pthread\_create(&producers[i], NULL, (void \*)producer, &producer\_index[i]);      }      for(int i = 0; i < CON\_NUM; i++){ // 创建消费者线程          pthread\_create(&consumers[i], NULL, (void \*)consumer, &consumer\_index[i]);      }      for(int i = 0; i < PRO\_NUM; i++){ // 等待所有生产者线程结束          pthread\_join(producers[i], NULL);      }      for(int i = 0; i < CON\_NUM; i++){ // 等待所有消费者线程结束          pthread\_join(consumers[i], NULL);      }      pthread\_mutex\_destroy(&mutex); // 销毁互斥锁      sem\_destroy(&empty); // 销毁空缓冲区信号量      sem\_destroy(&full); // 销毁满缓冲区信号量      return 0;  } |

“input.txt”文件内容：

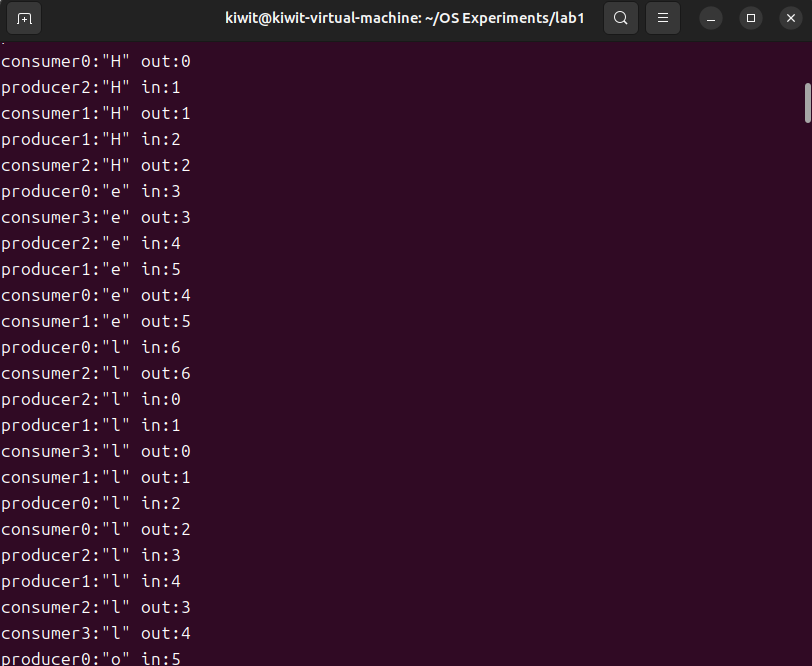
Hello world!

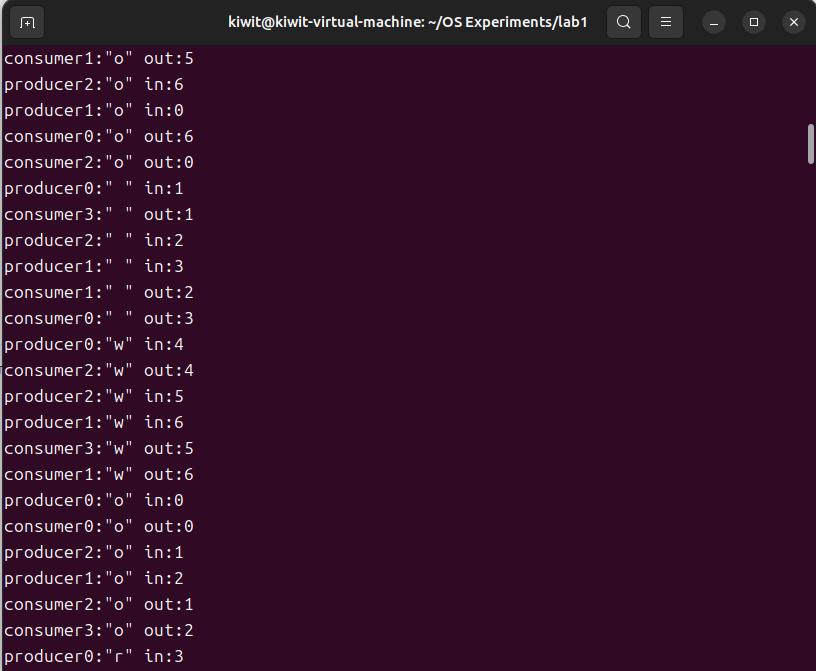
This is a test file.

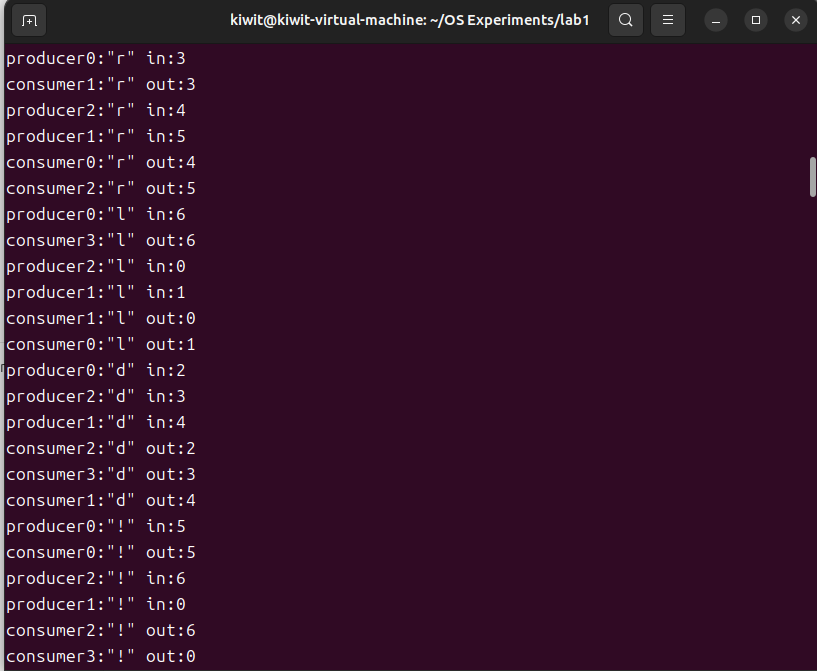
使用gcc进行编译、执行：

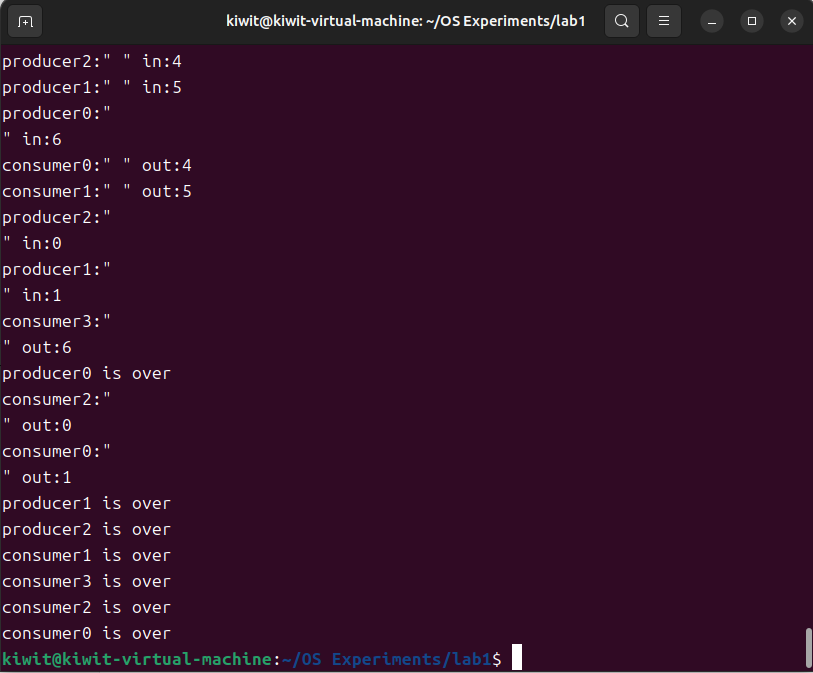


程序结果如下：







由截图可见生产者与消费者成功完成工作且未死锁并正常退出

1. **总结及心得体会：**

本实验中要求实现生产者/消费者问题，相较于哲学家进餐问题，该问题需要考虑如何确定生产者和消费者的任务已经完成后退出。在实现过程中，我采用了一个全局变量isfinish来记录生产者的数量，当该值变为0时，即意味着生产者已经全部完成工作，消费者仅需将剩余缓冲池内的货物消费完即可退出。

同时，在消费者线程中，我将sem\_wait函数改为sem\_trywait函数，这样能够避免消费者一直陷入堵塞的情况，使得消费者能够正常退出。通过这次实验，我深入理解了多线程编程的基本原理和方法，掌握了Linux信号量和共享内存机制的基本使用方法，同时还提高了自己的编程思维和实践能力。

1. **实验改进与建议：**

可以添加更多的功能，例如记录每个线程的执行时间、在缓冲池为空或已满时给出提示等。

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名：成棋伟 学 号：2022130102013**

**指导教师： 刘瑶**

**实验地点：信软楼西306 实验时间： 2025年4月28日**

1. **实验室名称：**

信软楼西306

1. **实验项目名称：**

哲学家就餐问题

1. **实验学时：**

4学时

1. **实验原理：**

哲学家进餐问题描述有五个哲学家，他们的生活方式是交替地进行思考和进餐，哲学家们共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五支筷子，平时哲学家进行思考，饥饿时便试图取其左、右最靠近他的筷子，只有在他拿到两支筷子时才能进餐，进餐完毕，放下筷子又继续思考。

约束条件：

(1)只有拿到两只筷子时，哲学家才能吃饭。

(2)如果筷子已被别人拿走，则必须等别人吃完之后才能拿到筷子。

(3)任一哲学家在自己未拿到两只筷子吃饭前，不会放下手中拿到的筷子。

信号量机制：

筷子是临界资源，一段时间只允许一位哲学家使用。为了表示互斥，用一个信号量表示一只筷子，五个信号量构成信号量数组。本文中算法用类C语言描述伪码算法。算法描述如下：n用五支筷子的信号量构成信号量数组：

Semaphore chopstick[5]={1，l，1，l，1}；

p(stick[i])；

p(stick[(i+1) % 5])；

进餐；

v(stick[i])；

v(stick[(i+1) % 5])；

思考；

当哲学家饥饿时，总是先去拿他左边的筷子，执行wait(chopstick[I])，成功后，再去拿他右边的筷子，执行wait(chopstick[I+1]%5)；成功后便可进餐。进餐毕，先放下他左边的筷子，然后再放下右边的筷子。当五个哲学家同时去取他左边的筷子，每人拿到一只筷子且不释放，即五个哲学家只得无限等待下去，引起死锁。程序应该用相应措施避免死锁或者在程序发生死锁时解除死锁。

确保同一时间里只有一个线程访问共享资源或临界区域。互斥量（mutex）本质上是一把锁，在访问共享资源后临界区域前，对互斥量进行加锁，在访问完成后释放互斥量上的锁，对互斥量进行加锁后，任何其他试图再次对互斥量加锁的线程将会被阻塞，直到锁被释放。

互斥量变量由pthread\_mutex\_t数据类型表示。在我们使用一个mutex变量之前，我们必须首先初始化它，通过把它设置为常量PTHREAD\_MUTEX\_INITIALIZER（只针对于静态分配的互斥量）或调用pthread\_mutex\_init。如果我们动态地分配互斥量（例如通过调用malloc），那么我们需要在释放内存前调用pthread\_mutex\_destroy。

#include <pthread.h>

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr);

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

成功返回0，失败返回错误码。

为了使用默认属性初始化一个互斥量，我们把attr设置为NULL。

为了锁住一个互斥量，我们调用pthread\_mutex\_lock。如果互斥量已经被锁了，那么调用线程将阻塞，直到互斥量被解锁。要解锁一个互斥量，我们调用pthread\_mutex\_unlock。

#include <pthread.h>

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

成功返回0，失败返回错误号。

如果一个线程不能容许阻塞，那么它可以使用pthread\_mutex\_trylock来有条件地锁这个互斥量。如果互斥量在pthread\_mutex\_trylock被调用时已经解锁，那么pthread\_mutex\_trylock会不阻塞地锁住这个互斥量并返回0。否则，pthread\_mutex\_trylock将会失败，返回EBUSY而不锁住这个互斥量。

　 多个任务循环等待它方占有的资源而无限期地僵持下去的局面。即：行进中的任务，等待永远不可能发生的事件。死锁是一种CPU的状态。

产生死锁的原因是系统资源不足（有限）；进程推进顺序不合适（操作不当）

　 同时具备以下四个条件时，死锁就可能发生。

1、互斥条件：

一个资源，在一个时刻只能由一个进程占有；

2、不可抢占条件：

一个资源被占有，在末释放之前，其它进程不得抢占；

3、占有且申请条件：

进程已占有某种资源，又申请新的资源；

4、循环等待条件：

进程已占有某种资源，又要等待其它进程已拥有的资源。

解决死锁的办法有：预防、避免、检测与恢复三种。

死锁预防的基本思想：要求进程申请资源时必须遵守某种协议，从而打破死锁产生的四个必要条件之一。

1、打破互斥条件：

　　允许进程同时访问某些资源，但有些资源不允许同时被访问；

2、打破不可抢占条件：

允许进程强行从占有者处抢占某些资源；

＊以上两个条件的打破都对系统管理不利，所以不易实行，不可取。

3、打破占有且申请条件：

　　采用资源预先分配策略。

　　这个策略是：只有当进程申请的所有资源都得到满足后，才调度其运行。

4、打破循环等待条件：

　　实行资源有序分配策略。

这个策略是将资源按类编号，按序分配。进程申请资源必须按号递增提出。若一个号已被占有，它只能申请它以后的号。

1. **实验目的：**

* 掌握哲学家就餐问题，掌握预防死锁的实现方法；
* 了解Linux系统下进程和线程的实现；
* 掌握通过互斥量、POSIX信号量、XSI信号量集实现多线程/多进程同步控制的方法

1. **实验内容：**

熟悉Ubuntu系统环境和命令；熟悉Ubuntu系统下的多线程/多进程编程；在Ubuntu系统下编程实现哲学家就餐问题。实现教材2.5.2节中所描述的哲学家就餐问题。要求显示出每个哲学家的工作状态，如吃饭，思考。连续运行30次以上都未出现死锁现象。

1. **实验器材（设备、元器件）：**

PC计算机，操作系统：Ubuntu

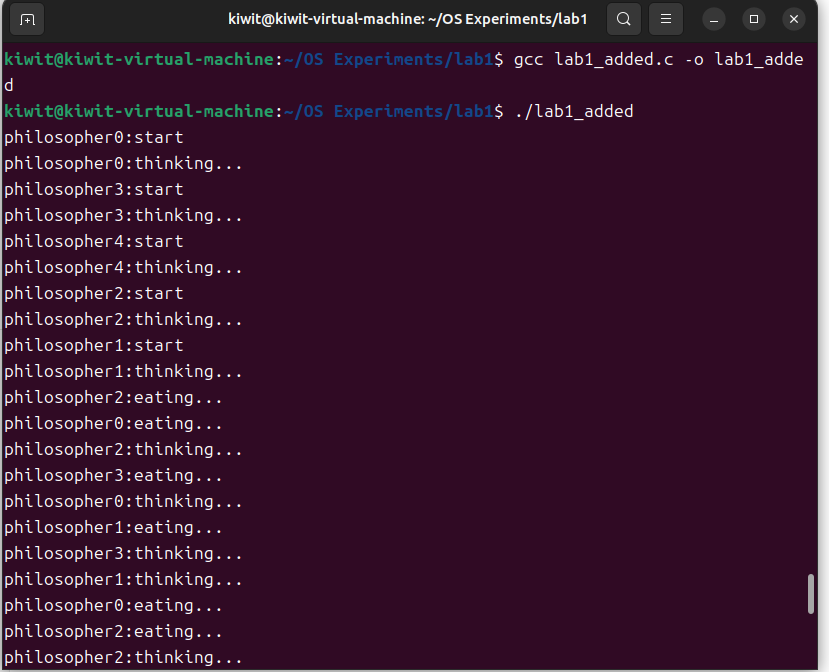
1. **实验步骤：**
2. 编写多线程完成哲学家进餐问题程序
3. 使用gcc将.c文件编译成可执行文件
4. 在虚拟机linux环境下执行程序观察结果
5. **实验数据及结果分析：**

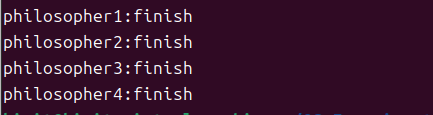
多线程程序代码如下：

|  |
| --- |
| #include<stdio.h>  #include<unistd.h>  #include<semaphore.h>  #include<pthread.h>  #define NUM 5 //定义哲学家数量  sem\_t chopsticks[NUM];//定义五个信号量，代表五个筷子  int number[NUM] = {0,1,2,3,4};//定义一个数组，存储每个哲学家编号  //定义哲学家行为  void philosopher(int\* arg){      int i = \*arg;      printf("philosopher%d:star\n",i);//打印哲学家进入餐厅的消息      int j = 7;      while(j--){          printf("philosopher%d:thinking……\n",i);          sleep(0.5);          if(i%2 == 0){ //偶数号哲学家优先拿左手边的筷子              sem\_wait(&chopsticks[i]);//获取左边的筷子信号量              sem\_wait(&chopsticks[(i+1)%NUM]);//获取右边的筷子信号量              printf("philosopher%d:eating...\n",i); //打印哲学家进餐消息              sleep(0.5);              sem\_post(&chopsticks[i]);//释放左边的筷子信号量              sem\_post(&chopsticks[(i+1)%NUM]);//释放右边的筷子信号量          }          else{ //奇数号哲学家优先拿右手边的筷子              sem\_wait(&chopsticks[(i+1)%NUM]); //获取右边的筷子信号量              sem\_wait(&chopsticks[i]); //获取左边的筷子信号量              printf("philosopher%d:eating...\n",i); //打印哲学家进餐消息              sleep(1);              sem\_post(&chopsticks[i]); //释放左边的筷子信号量              sem\_post(&chopsticks[(i+1)%NUM]); //释放右边的筷子信号量          }      }  }  int main(){      pthread\_t id[5]; //定义五个线程id      for (int i = 0; i < NUM;i ++){          sem\_init(&chopsticks[i],0,1); //初始化五个信号量为1，表示筷子可用      }      for(int i = 0; i < NUM; i ++){          pthread\_create(&id[i],NULL,(void \*)philosopher,&number[i]);//创建五个哲学家线程      }      for(int i = 0; i < NUM; i ++){          pthread\_join(id[i],NULL); //等待每个哲学家线程结束          printf("philosopher%d:finish\n",i); //打印哲学家离开餐厅的消息      }      for(int i = 0; i < NUM; i ++){          sem\_destroy(&chopsticks[i]); //销毁五个信号量      }  } |

该程序限制奇数号哲学家只能先拿左手边筷子，偶数号哲学家只能先拿右手边筷子，这样即可避免死锁情况的产生。

使用gcc编译程序，程序结果如下：





由截图可见程序成功执行了30次以上并且未死锁

1. **总结及心得体会：**

在本次实验中，我通过学习Ubuntu系统环境和命令，熟悉了Ubuntu系统下的多线程编程，并成功实现了哲学家就餐问题。通过此次实验，我深刻认识到死锁问题的严重性，加深了对多线程编程的理解。在编程过程中，我注重代码的规范性和可读性，并多次进行调试和测试，确保程序的正确性和稳定性。通过不断地修改和完善代码，我成功实现了30次以上连续运行未出现死锁问题的目标。整个实验过程让我受益匪浅，提高了我的编程能力和解决问题的能力。

1. **实验改进与建议：**

考虑到哲学家就餐问题本身的随机性，可以运行多次以验证程序的正确性和稳定性。可以使用脚本或者其他自动化工具来运行多次实验，并对结果进行统计和分析。