山东大学 计算机科学与技术 学院

操作系统 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201900161140 | 姓名：张文浩 | | 班级： 19智能 |
| 实验题目：死锁问题实验 | | | |
| 实验学时：2 | | 实验日期： 5.20 | |
| 实验目的：  通过本实验观察死锁产生的现象，考虑解决死锁问题的方法。从而进一步加深对于死锁问题的理解。掌握解决死锁问题的几种算法的编程和调试技术。练习怎样构造管程和条件变量，利用管程机制来避免死锁和饥饿问题的发生。思考示例实验是否真正模拟了哲学家就餐问题？为什么示例程序不会产生死锁？为什么会出现进程死锁和饥饿现象？怎样利用实验造成和表现死锁和饥饿现象？管程能避免死锁和饥饿的机理是什么？为什么在管程中要使用条件变量而不直接使用信号量来达到进程同步的目的？ | | | |
| 硬件环境： 虚拟机 | | | |
| 软件环境：  Ubuntu | | | |
| 实验步骤与内容：   1. 根据指导书，完成实例实验   运行结果如下：    可以看到 5 个哲学家进程在 3 中状态中不断的轮流变换，且连续的 5 个输出中不应第二部分 操作系统算法实验 第 71 页 有多于 2 个的状态为 eating，同一进程号不应有两个连续的输出。  将速度设置为3，重新执行    经过多次测试，发现不会出现死锁现象。   1. 独立实验，单行车道问题   大体的思路：  建立一个管程，里面包括两个阻塞队列和对应的信号量，一个控制进程进入管程的lock锁。管程中包含3个方法，即arrive，cross和qiut，分别对应着车辆的到达、车辆的行驶和车辆行驶结束。每个车辆对应着一个进程，给个车辆先互斥地进入管程完成到达操作，如果不满足行驶条件，就进入管程与之对应的阻塞队列，等别的车辆行驶结束后唤醒自己。然后在被唤醒后就再次申请进入管程进行cross操作，在车道上行驶，最后行驶结束就退出，并尝试唤醒另一条方向上对应的阻塞队列上的车辆。  dp.h文件如下：  #include <iostream>  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <sys/sem.h>  #include <sys/msg.h>  #include <sys/wait.h>  /\*信号灯控制用的共同体\*/  typedef union semuns  {  int val;  } Sem\_uns;  //管程中使用的信号量  class Sema  {  public:  Sema(int id);  ~Sema();  int down(); //信号量加 1  int up(); //信号量减 1  private:  int sem\_id; //信号量标识符  };  //管程中使用的锁  class Lock //管程要使用的锁  {  public:  Lock(Sema \*lock);  ~Lock();  void close\_lock(); //关锁，不让其他进程进入管程  void open\_lock(); //开锁，允许其他进程进入管程  private:  Sema \*sema; //锁使用的信号量  };  class Condition //条件变量  {  public:  Condition(Sema \*sema1, Sema \*sema2);  ~Condition();  //条件变量的两个方法，wait和signal  void Wait(Lock \*conditionLock, int direct); //wait里面对信号量进行down操作，判断是否满足发车条件，不满足则阻塞，及侵入当前条件变量的阻塞队列  int Signal (int direc); //唤醒相反方向条件变量的一个进程  //唤醒相反方向阻塞车辆  private:  Sema\* sema0; // 一个方向阻塞队列  Sema\* sema1; // 另一方向阻塞队列  Lock\* lock; // 进入管程时获取的锁  };  class OneWay //定义管程  {  public:  OneWay (int maxall, int maxcur);  ~OneWay();  //管程中的三种方法：arrive,cross,quit  void Arrive (int direc);  // 车辆准备上单行道,direc 为行车方向  void Cross (int direc);  // 车辆正在单行道上  void Quit (int direc);  // 车辆通过了单行道  int \*eastCount; //当前车道上向东行驶车辆的数量  int \*westCount; //当前车道上向西行驶车辆的数量  int \*eastWait; //当前等待向东行驶车辆的数量  int \*westWait; //当前等待向西行驶车辆的数量  int \*sumPassedCars;//已经通过的车辆总数  private:  //建立或获取 ipc 信号量的一组函数的原型说明  int get\_ipc\_id (char \*proc\_file, key\_t key);  int set\_sem(key\_t sem\_key, int sem\_val, int sem\_flag);  //创建共享内存  char \*set\_shm(key\_t shm\_key, int shm\_num, int shm\_flag);  int rate; //车速  int \*maxCars;//最大同向车数  int \*numCars; //当前正在通过的车辆数  int \*currentDire;//当前通过的车辆的方向  Condition \*condition; //通过单行道的条件变量  Lock \*lock;//单行道管程锁  };  dp.c文件如下：  #include "dp.h"  using namespace std;  Sema::Sema(int id)  {  sem\_id = id;  }  Sema::~Sema()  {  }  /\*  \* 信号灯上的 down/up 操作  \* semid:信号灯数组标识符  \* semnum:信号灯数组下标  \* buf:操作信号灯的结构  \*/  int Sema::down()  {  struct sembuf buf;  buf.sem\_op = -1;  buf.sem\_num = 0;  buf.sem\_flg = SEM\_UNDO;  if ((semop(sem\_id, &buf, 1)) < 0)  {  perror("down error ");  exit(EXIT\_FAILURE);  }  return EXIT\_SUCCESS;  }  int Sema::up()  {  Sem\_uns arg;  struct sembuf buf;  buf.sem\_op = 1;  buf.sem\_num = 0;  buf.sem\_flg = SEM\_UNDO;  if ((semop(sem\_id, &buf, 1)) < 0)  {  perror("up error ");  exit(EXIT\_FAILURE);  }  return EXIT\_SUCCESS;  }  /\*  \* 用于单行道管程的互斥执行  \*/  Lock::Lock(Sema \* s) //保证只有一个进程使用管程的锁  {  sema = s;  }  Lock::~Lock()  {  }  //上锁  void Lock::close\_lock()  {  sema->down();  }  //开锁  void Lock::open\_lock()  {  sema->up();  }  //得到标识符  int OneWay::get\_ipc\_id (char \*proc\_file, key\_t key)  {  #define BUFSZ 256  FILE \*pf;  int i, j;  char line[BUFSZ], colum[BUFSZ];  if ((pf = fopen(proc\_file, "r")) == NULL)  {  perror("Proc file not open");  exit(EXIT\_FAILURE);  }  fgets(line, BUFSZ, pf);  while (!feof(pf))  {  i = j = 0;  fgets(line, BUFSZ, pf);  while (line[i] == ' ')  i++;  while (line[i] != ' ')  colum[j++] = line[i++];  colum[j] = '\0';  if (atoi(colum) != key)  continue;  j = 0;  while (line[i] == ' ')  i++;  while (line[i] != ' ')  colum[j++] = line[i++];  colum[j] = '\0';  i = atoi(colum);  fclose(pf);  return i;  }  fclose(pf);  return -1;  }  /\*  \*  set\_shm 函数建立一个具有 n 个字节 的共享内存区  \*  如果建立成功,返回 一个指向该内存区首地址的指针 shm\_buf  \*  输入参数:  \*  shm\_key 共享内存的键值  \*  shm\_val 共享内存字节的长度  \*  shm\_flag 共享内存的存取权限  \*/  char \* OneWay::set\_shm(key\_t shm\_key, int shm\_num, int shm\_flg) //在管程中建立共享内存  {  int i, shm\_id;  char \* shm\_buf;  //测试由 shm\_key 标识的共享内存区是否已经建立  if ((shm\_id = get\_ipc\_id("/proc/sysvipc/shm", shm\_key)) < 0)  {  //shmget 新建 一个长度为 shm\_num 字节的共享内存  if ((shm\_id = shmget(shm\_key, shm\_num, shm\_flg)) < 0)  {  perror("shareMemory set error");  exit(EXIT\_FAILURE);  }  //shmat 将由 shm\_id 标识的共享内存附加给指针 shm\_buf  if ((shm\_buf = (char \*) shmat (shm\_id, 0, 0)) < (char \*) 0)  {  perror("get shareMemory error");  exit(EXIT\_FAILURE);  }  for (i = 0; i < shm\_num; i++)  shm\_buf[i] = 0; //初始为 0  }  //共享内存区已经建立,将由 shm\_id 标识的共享内存附加给指针 shm\_buf  if ((shm\_buf = (char \*) shmat(shm\_id, 0, 0)) < (char \*) 0)  {  perror("get shareMemory error");  exit(EXIT\_FAILURE);  }  return shm\_buf;  }  /\*  \*  set\_sem 函数建立一个具有 n 个信号灯的信号量  \*  如果建立成功,返回 一个信号量的标识符 sem\_id  \*  输入参数:  \*  sem\_key 信号量的键值  \*  sem\_val 信号量中信号灯的个数  \*  sem\_flag 信号量的存取权限  \*/  int OneWay::set\_sem(key\_t sem\_key, int sem\_val, int sem\_flg) //在管程中建立信号灯  {  int sem\_id;  Sem\_uns sem\_arg;  //测试由 sem\_key 标识的信号量是否已经建立  if ((sem\_id = get\_ipc\_id("/proc/sysvipc/sem", sem\_key)) < 0)  {  //semget 新建一个信号灯,其标号返回到 sem\_id  if ((sem\_id = semget(sem\_key, 1, sem\_flg)) < 0)  {  perror("semaphore create error");  exit(EXIT\_FAILURE);  }  }  //设置信号量的初值  sem\_arg.val = sem\_val;  if (semctl(sem\_id, 0, SETVAL, sem\_arg) < 0)  {  perror("semaphore set error");  exit(EXIT\_FAILURE);  }  return sem\_id;  }  Condition::Condition(Sema \*semax1, Sema \*semax2)  {  sema0 = semax1;  sema1 = semax2;  }  /\*\*  \* 看看是否能通过  \*/  void Condition::Wait(Lock \*lock, int direc) //当前车辆不能上路，利用wait方法进入条件变量的阻塞队列  {  if (direc == 0)  {  cout << getpid() << " 号车辆在等待向东行驶" << "\n";  lock->open\_lock(); //开锁，让其他进程进入管程，等待被其他进程唤醒  sema0->down(); //阻塞，等待被up唤醒  lock->close\_lock(); //关锁，自己要上路了  }  else if (direc == 1)  {  cout << getpid() << " 号车辆在等待向西行驶" << "\n";  lock->open\_lock();  sema1->down();  lock->close\_lock();  }  }  int Condition::Signal (int direc) //条件变量中唤醒相反车道上车辆进程的方法  {  int i;  if (direc == 0)  {  i = sema0->up();  }  else if (direc == 1)  {  i = sema1->up();  }  return i;  }  /\*  \* get\_ipc\_id() 从/proc/sysvipc/文件系统中获取 IPC 的 id 号  \* pfile: 对应/proc/sysvipc/目录中的 IPC 文件分别为  \*  msg-消息队列,sem-信号量,shm-共享内存  \* key: 对应要获取的 IPC 的 id 号的键值  \*/  Condition::~Condition()  {  }  ;  /\*  \*  set\_shm 函数建立一个具有 n 个字节 的共享内存区  \*  如果建立成功,返回 一个指向该内存区首地址的指针 shm\_buf  \*  输入参数:  \*  shm\_key 共享内存的键值  \*  shm\_val 共享内存字节的长度  \*  shm\_flag 共享内存的存取权限  \*/  OneWay::OneWay (int maxall, int maxcur)  {  //管程中的两个阻塞队列  Sema \*sema0;  Sema \*sema1;  //实现进入互斥进入管程的锁  Sema \*semaLock;  int ipc\_flg = IPC\_CREAT | 0644;  //建立共享内存  maxCars = (int \*) set\_shm(100, 1, ipc\_flg); //最大车辆数  numCars = (int \*) set\_shm(200, 1, ipc\_flg); //当前车辆数  currentDire = (int \*) set\_shm(300, 1, ipc\_flg); //当前单行道的行驶方向  eastCount = (int \*) set\_shm(501, 1, ipc\_flg); //当前单行道上向东行驶的车辆数  westCount = (int \*) set\_shm(502, 1, ipc\_flg); //当前单行道上向西行驶的车辆数  sumPassedCars = (int \*) set\_shm(700, 1, ipc\_flg); //一共通过了多少车  eastWait = (int \*) set\_shm(801, 1, ipc\_flg); //等待向东行驶的车辆数  westWait = (int \*) set\_shm(802, 1, ipc\_flg); //等待向西行驶的车辆数  int sema0\_id = set\_sem(401, 0, ipc\_flg); //信号量  int sema1\_id = set\_sem(402, 0, ipc\_flg);  int semaLock\_id = set\_sem(601, maxcur, ipc\_flg);  //初始化管程中的共享内存  \*maxCars = maxcur;  \*numCars = 0;  \*currentDire = 0;  \*eastCount = 0;  \*westCount = 0;  \*sumPassedCars = 0;  \*eastWait = 0;  \*westWait = 0;  sema0 = new Sema(sema0\_id);  sema1 = new Sema(sema1\_id);  semaLock = new Sema(semaLock\_id);  lock = new Lock(semaLock);  condition = new Condition(sema0, sema1);  }  void OneWay::Arrive (int direc)  {  lock->close\_lock(); //有车到达，想进入管程，先关锁  if(direc==0){  \*eastWait += 1;  }else{  \*westWait += 1;  }  if ((\*currentDire != direc || \*numCars >= \*maxCars) & \*sumPassedCars > 0) //如果当前不能通行  {  condition->Wait(lock, direc); //调用条件变量的wait方法，进入阻塞队列  }  //在被唤醒后或者车辆到达的时候满足条件，说明现在可以进入单车道，更新一下各个共享内存的状态  if (direc == 0) //东 +1  {  \*eastWait -= 1;  \*eastCount = \*eastCount + 1;  cout << getpid() << " 号车辆进入单车道，向东行驶\n";  }  else if (direc == 1) //西 +1  {  \*westCount = \*westCount + 1;  \*westWait -= 1;  cout << getpid() << " 号车辆进入单车道，向西行驶\n";  }  \*numCars = \*numCars + 1;  \*currentDire = direc;  \*sumPassedCars += 1;  lock->open\_lock();  }  void OneWay::Cross (int direc)  {  lock->close\_lock(); //当前车辆在路上，三秒走完  sleep(3);  if (direc == 0)  cout << getpid() << " 号车辆向东行驶过车道"<< "\n";  else if (direc == 1)  cout << getpid() << " 号车辆向西行驶过车道"<< "\n";  lock->open\_lock();  }  void OneWay::Quit (int direc)  {  lock->close\_lock();  \*numCars -= 1;  if (direc == 0)  {  cout << getpid() << " 号向东行驶的车辆离开" << "\n";  }  else if (direc == 1)  {  cout << getpid() << " 号向西行驶的车辆离开" << "\n";  }  if (\*numCars == 0)  {  if (direc == 0)  {  if (\*westWait > 0)  {  condition->Signal(1); //如果这辆车离开之后车道上没别的车了，就唤醒另一条车道的阻塞队列（如果有的话）  }  else if (\*eastWait > 0)  {  condition->Signal(0);  }  }  else if (direc == 1)  {  if (\*eastWait > 0)  {  condition->Signal(0);  }  else if (\*westWait > 0)  {  condition->Signal(1);  }  }  }  lock->open\_lock();  }  OneWay::~OneWay()  {  delete condition;  }  int main (int argc, char \*\*argv)  {  int maxCars;  int maxSingelDirect;  cout << "请输入总车辆数:";  cin >> maxCars;  cout << "请输入单方向通过的最大车数:";  cin >> maxSingelDirect;  OneWay \*oneWay = new OneWay(maxCars, maxSingelDirect);  //建立管程,判断可不可进、决定方向,进入单行道  int i;  int pid[maxCars];  for (i = 0; i < maxCars; i++) //对每一辆车都创建一个子进程  {  pid[i] = fork();  if (pid[i] == 0)  {  sleep(1);  srand(getpid());  int direct;  direct = rand() % 2;  //direct=\*oneWay->sumPassedCars%2;  oneWay->Arrive(direct);  oneWay->Cross(direct);  oneWay->Quit(direct);  exit(EXIT\_SUCCESS);  }  }  for (i = 0; i < maxCars; i++)  {  waitpid(pid[i], NULL, 0);  }  cout << \*(oneWay->eastCount) << "辆列车向东" << \*(oneWay->westCount)  << "辆列车向西,正常通行.\n";  delete oneWay;  return EXIT\_SUCCESS;  }  运行结果如下：  设置总车辆数为5，单次可通行的最大车辆数为2。  在本次运行中，有两个向西行驶的车辆到达，进入车道，随后又来了一辆向西行驶的，但当前车道已经有两个车了，所以第三辆向西行驶的车进入阻塞队列。又来了一辆向东行驶的车辆，因为与当前行驶方向不用，也要加入阻塞度列。随后在一开始进入的两个车辆行驶结束之后，为了防止向东行驶的车辆饥饿，先唤醒向东行驶车辆的阻塞队列…… | | | |
| 结论分析与体会：  在哲学家的实验中，出现死锁的情况是所有的哲学家都先获取左边的筷子，都等待着右边的筷子，这样就永远无法得到，就出现了死锁的现象。  在实例实验中，模拟并解决了哲学家就餐问题， 哲学家可以在想要就餐的时候，首先先在其获得筷子前互斥检查是否可以同时获得左右的两只筷子，然后再决定是否可以同时获得两只筷子。而当用完自己的筷子后释放掉自己的筷子并通知身边的其他人这一情况。实现了死锁的避免。  产生饥饿的主要原因是：在一个动态系统中，对于每类系统资源，操作系统需要确定一个分配策略，当多个进程同时申请某类资源时，由分配策略确定资源分配给进程的次序。有时资源分配策略可能是不公平的，即不能保证等待时间界的存在。在这种情况下，即使系统没有发生死锁某些进程也可能会长时间等待．当等待时间给进程推进和响应带来明显影响时， 称发生了进程饥饿，当饥饿到一定程度的进程所赋予的任务即使完成也不再具有实际意义时称该进程被饿死。  管程解决进程同步问题就是将互斥区和进入区互斥锁代码段封装成管程，任意时刻只有一个进程获得管程所有权，解决了死锁问题。信号量是通过在某一进程获得所有权时， 通过某一信号量阻塞其它想要进入临界区的进程，完成临界区之后， 再通过信号量通知其它想要进入临界区的进程。条件变量是用来表示某一资源是否可用，然后再决定是否同时获得竞争资源。信号灯机制则侧重于点灯，即告知资源可用； 没有等待线程的解锁或激发条件都是没有意义的，而没有等待灯亮的线程的点灯操作则有效，且能保持灯亮状态。当然， 这样的操作原语也意味着比条件变量更多的开销。  总之，通过本次实验，我考虑解决死锁问题的方法，进一步加深了对于死锁问题的理解 | | | |

附源代码