# POSIX信号量

课程名称 操作系统原理实验

实验项目名称 实验五 POSIX信号量

学生姓名 司晨旭 专业班级 计算2002 学号 32001019

**实验目的和要求:**

1. 理解进程与线程的区别；
2. 使用POSIX信号量，实现线程的互斥和同步控制。

**实验内容:**

1. 使用POSIX信号量，实现线程的互斥和同步控制；
2. 实现读者优先的读者-写者问题；
3. 实现有限缓冲的生产者-消费者问题。

**实验步骤:**

1. **基本概念**

Linux提供两种信号量：内核信号量（由内核控制路径使用）和用户态进程使用的信号量。用户态信号量又分为POSIX 信号量和SYSTEM V信号量（将在实验九介绍）。[POSIX信号量](http://www.cnblogs.com/Anker/archive/2013/01/13/2858765.html)（semaphore）是一种用于提供不同进程间或一个给定进程的不同线程间同步手段的原语。POSIX信号量又分为有名信号量和无名信号量。有名信号量，其值保存在文件中，用于线程，也可以用于相关或无关进程间的同步；无名信号量，其值保存在内存中，用于线程或相关进程间的同步。

本次实验主要介绍**POSIX无名信号量**的使用。**无名信号量必须是多个进程（线程）的共享变量，无名信号量要保护的变量也必须是多个进程（线程）的共享变量。**

1. 线程互斥例题：

生产围棋的工人不小心把相等数量的黑子和白子混装于一个箱子里，现要用自动分拣系统把黑子和白子分开，该系统由两个并发执行的进程组成，功能如下：(1)进程A专门拣黑子，进程B专门拣白子；(2)每个进程每次只拣一个子，当一个进程在拣子时不允许另一个进程去拣子。试用PV操作实现两者的同步。

用PV操作解决互斥的方法：

1. 确定临界资源，找到临界区；
2. 设置互斥信号量，初始值为1；
3. 在临界区前P操作，临界区后V操作。

|  |
| --- |
| #include <pthread.h>  **#include <semaphore.h>**#include <sys/types.h>#include <stdio.h>#include <unistd.h>  int number; // 被保护的全局变量sem\_t sem\_id;  void\* thread\_white\_fun(void \*arg)  { int i; for(i = 0;i < 3;i++) {  **sem\_wait(&sem\_id);** printf("w%d:thread\_white have the semaphore\n",i); number++;usleep(100); printf("w%d:white number = %d\n",i,number); **sem\_post(&sem\_id);**  }}  void\* thread\_black\_fun(void \*arg){ int i; for(i = 0;i < 3;i++) {  **sem\_wait(&sem\_id);** printf("b%d:thread\_black have the semaphore \n",i); number--;usleep(100); printf("b%d:black number = %d\n",i,number); **sem\_post(&sem\_id);**  }}  int main(int argc,char \*argv[]){ number = 0; pthread\_t id1, id2;  **sem\_init(&sem\_id, 0, 1);** pthread\_create(&id1,NULL,thread\_white\_fun, NULL); pthread\_create(&id2,NULL,thread\_black\_fun, NULL); pthread\_join(id1,NULL); pthread\_join(id2,NULL); printf("main,,,\n"); return 0;  } |

编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：：

1. 了解信号量的初始化操作、P操作和V操作；

|  |
| --- |
| **信号量初始化**：int sem\_init(sem\_t \*sem,int pshared,unsigned int value);  **P操作（信号量的减1操作）**：int sem\_wait(sem\_t \*sem);  等待信号量，如果信号量的值大于0，将信号量的值减1，立即返回。如果信号量的值为0，则线程阻塞。相当于P操作。  返回值：  成功返回0，失败返回-1。  **V操作（信号量的加1操作）**：int sem\_post(sem\_t \*sem);  释放信号量，让信号量的值加1。若此时有sem\_wait正在阻塞则唤醒。相当于V操作。 |

**运行结果如下**

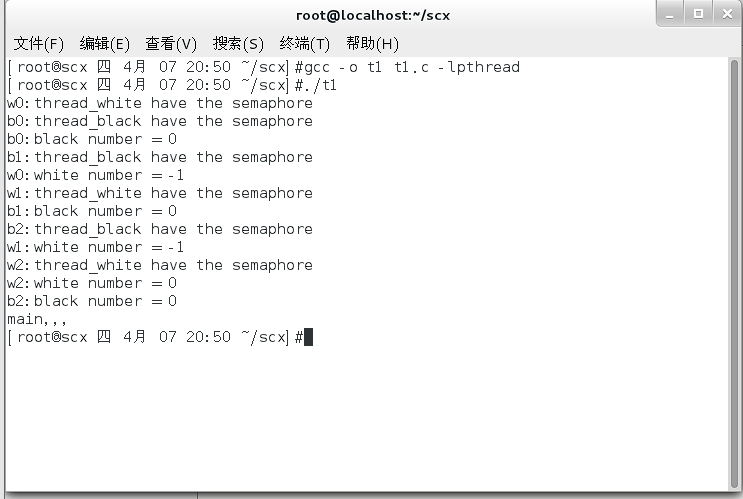
****

**解释**

|  |
| --- |
| 通过信号量为1的sem\_wait()和sem\_post()操作。保护了临界资源number，从而达到了互斥 |

1. 注销红色代码，保留sleep语句，体会如何使用信号量对共享资源的互斥访问进行管理。

**运行结果如下**



|  |
| --- |
| 可以发现，white和black的内容交替出现，并且number的值也十分的混乱。  这是因为临界区被破坏掉了，导致公共资源number可以被同时访问，且每个线程在运行时，在运行到usleep()函数时，转换到另一个进程，导致结果更加混乱。  信号量在这里可以简单的理解为通行证。只有用于通行证的线程才可以使用和访问临界资源。因此当有一个线程使用临界资源，通行证被拿走了，然后其他线程在进行P操作，就被阻塞掉了。等当前使用临界资源的线程进行v操作后，被p操作阻塞的线程继续向下执行，从而保护了临界资源。 |

1. 线程同步例题：

本例有四个线程。两个Read线程从文件中读取数据，两个Handle线程处理读取出来的数据。要求先读取数据，后处理数据，两组线程是同步关系。

用PV操作解决同步的方法：

1. 确定先后动作；
2. 设置同步信号量，初始值为0；
3. 先动作后V操作，后动作前P操作。

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  #include <semaphore.h>  #define MAXSTACK 100  int stack[MAXSTACK];  sem\_t sem;  //从文件1.dat读取数据，每读一次，信号量加一  void ReadData1(void){  int val;  FILE \*fp=fopen("1.dat","r");  while(!feof(fp)){  fscanf(fp,"%d %d",&stack[0],&stack[1]);  sem\_getvalue(&sem,&val);  printf("ReadData1 stack[0] =%d stack[1]=%d sem.val = %d\n", stack[0], stack[1], val);  **sem\_post(&sem);**  }  fclose(fp);  }  //从文件2.dat读取数据  void ReadData2(void){  int val;  FILE \*fp=fopen("2.dat","r");  while(!feof(fp)){  fscanf(fp,"%d %d",&stack[0],&stack[1]);  sem\_getvalue(&sem,&val);  printf("ReadData2 stack[0] =%d stack[1]=%d sem.val = %d\n", stack[0], stack[1], val);  **sem\_post(&sem);**  }  fclose(fp);  }  //阻塞等待缓冲区有数据，读取数据后，释放空间，继续等待  void HandleData1(void){  int val;  while(1){  **sem\_wait(&sem);**  sem\_getvalue(&sem,&val);  printf("Plus:%d+%d=%d sem.val = %d\n", stack[0], stack[1], stack[0] + stack[1],val);  }  }  void HandleData2(void){  int val;  while(1){  **sem\_wait(&sem);**  sem\_getvalue(&sem,&val);  printf("Multiply:%d\*%d=%d sem.val = %d\n",stack[0],stack[1], stack[0]\*stack[1], val);  }  }  int main(void){  pthread\_t t1,t2,t3,t4;  **sem\_init(&sem,0,0);**  pthread\_create(&t1,NULL,(void \*)HandleData1,NULL);  pthread\_create(&t2,NULL,(void \*)HandleData2,NULL);  pthread\_create(&t3,NULL,(void \*)ReadData1,NULL);  pthread\_create(&t4,NULL,(void \*)ReadData2,NULL);  pthread\_join(t1,NULL);  } |

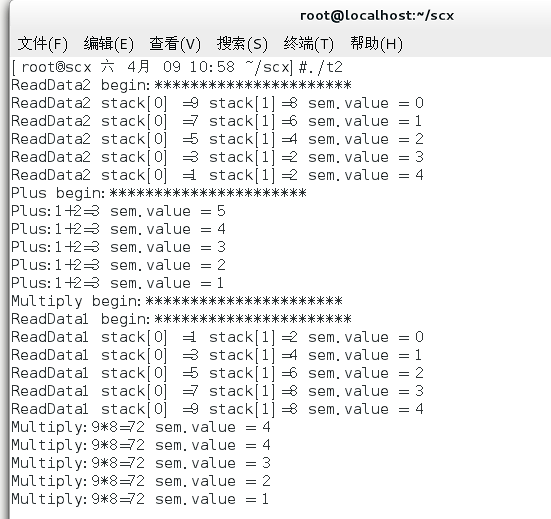
编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：：

1. 理解信号量的初始化操作、P操作和V操作；

|  |
| --- |
| **信号量初始化**：int sem\_init(sem\_t \*sem,int pshared,unsigned int value);  **P操作（信号量的减1操作）**：int sem\_wait(sem\_t \*sem);  等待信号量，如果信号量的值大于0，将信号量的值减1，立即返回。如果信号量的值为0，则线程阻塞。相当于P操作。  返回值：  成功返回0，失败返回-1。  **V操作（信号量的加1操作）**：int sem\_post(sem\_t \*sem);  释放信号量，让信号量的值加1。若此时有sem\_wait正在阻塞则唤醒。相当于V操作。 |

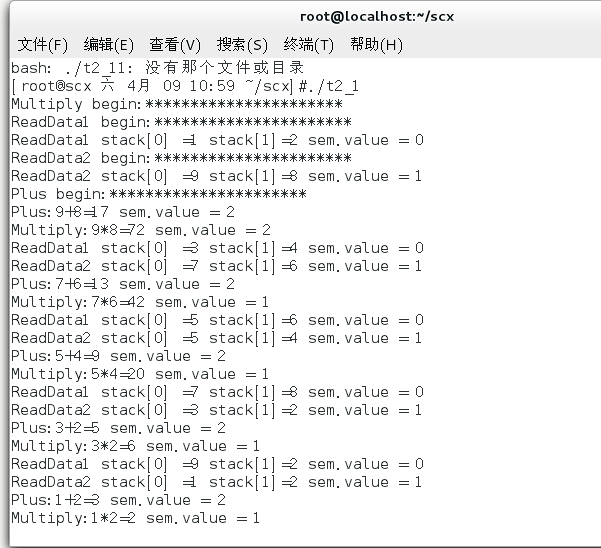
1. 分析出四个线程的前驱后继关系；

**首先给出源代码的运行结果**



**下面给出分析**

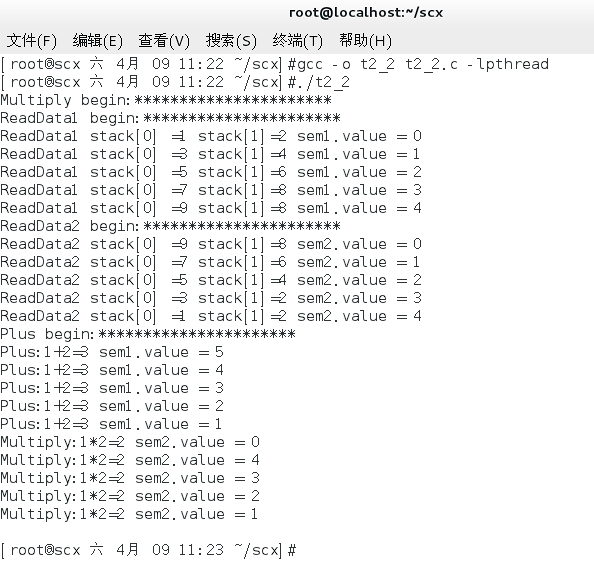
|  |
| --- |
| 可以看到  Readdata1,和readdata2线程每次运行，通行证都会加一，因此再拥有通行证之后，plus和Multiply两个进程，在拥有通行证之后，才可以运行下去  在有时候运行的时候，multiply或者plus线程没有收到通行证，就无法继续运行，阻塞在那里。  **因此，readdata1和readadata2是前驱，然后Plus和Multiply是后继** |

****

附上一个加上sleep(1)的运行结果，可以看到，乘法线程和加法线程更均匀的出现了

1. 若要求ReadData1读出的数据仅由HandleData1处理，ReadData2读出的数据仅由HandleData2处理，请修改例程。

**首先先给上运行结果**



**然后附上我是用的代码，实际上就是设计了两个信号量**

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  #include <semaphore.h>  #define MAXSTACK 100  int stack[MAXSTACK];  int size = 0;  sem\_t sem1;  sem\_t sem2;  */\* 从文件1.dat读取数据，每读一次，信号量加一\*/*  void ReadData1(void)  {      printf("ReadData1 begin:\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");      FILE \*fp = fopen("1.dat", "r");      while (!feof(fp))      {          fscanf(fp, "%d %d", &stack[0], &stack[1]);          sem\_getvalue(&sem1, &size);          printf("ReadData1 stack[0] =%d stack[1]=%d sem1.value = %d\n", stack[0], stack[1], size);          sem\_post(&sem1);  *// sleep(1);*      }      fclose(fp);  }  */\*从文件2.dat读取数据\*/*  void ReadData2(void)  {      printf("ReadData2 begin:\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");      FILE \*fp = fopen("2.dat", "r");      while (!feof(fp))      {          fscanf(fp, "%d %d", &stack[0], &stack[1]);          sem\_getvalue(&sem2, &size);          printf("ReadData2 stack[0] =%d stack[1]=%d sem2.value = %d\n", stack[0], stack[1], size);          sem\_post(&sem2);  *// sleep(1);*      }      fclose(fp);  }  */\*阻塞等待缓冲区有数据，读取数据后，释放空间，继续等待\*/*  void HandleData1(void)  {      printf("Plus begin:\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");      while (1)      {          sem\_getvalue(&sem1, &size);          sem\_wait(&sem1);          printf("Plus:%d+%d=%d sem1.value = %d \n", stack[0], stack[1], stack[0] + stack[1], size);  *// sleep(1);*      }  }  void HandleData2(void)  {      printf("Multiply begin:\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");      while (1)      {          sem\_getvalue(&sem2, &size);          sem\_wait(&sem2);          printf("Multiply:%d\*%d=%d sem2.value = %d\n", stack[0], stack[1], stack[0] \* stack[1], size);  *// sleep(1);*      }  }  int main(void)  {      pthread\_t t1, t2, t3, t4;      sem\_init(&sem1, 0, 0);      sem\_init(&sem2, 0, 0);      pthread\_create(&t1, NULL, (void \*)HandleData1, NULL);      pthread\_create(&t2, NULL, (void \*)HandleData2, NULL);      pthread\_create(&t3, NULL, (void \*)ReadData1, NULL);      pthread\_create(&t4, NULL, (void \*)ReadData2, NULL);  */\* 防止程序过早退出，让它在此无限期等待\*/*      getchar();  } |

1. 捡棋子题中，如果还要求进程A和进程B交替拣子，则为进程同步问题，修改代码并调试运行。

|  |
| --- |
| #include <pthread.h>  #include <semaphore.h>  #include <sys/types.h>  #include <stdio.h>  #include <unistd.h>  int number; // 被保护的全局变量  **sem\_t sem\_id1, sem\_id2;**  void\* thread\_white\_fun(void \*arg)  {  int i;    for(i = 0;i < 3;i++)  {  **sem\_wait(&sem\_id1);**  printf("thread\_white have the semaphore\n");  number++;  printf("number = %d\n",number);  **sem\_post(&sem\_id2);**  //sleep(1);  }  }  void\* thread\_black\_fun(void \*arg)  {  int i;    for(i = 0;i < 3;i++)  {  **sem\_wait(&sem\_id2);**  printf("thread\_black have the semaphore \n");  number--;  printf("number = %d\n",number);  **sem\_post(&sem\_id1);**  //sleep(1);  }  }  int main(int argc,char \*argv[])  {  number = 0;  pthread\_t id1, id2;  **sem\_init(&sem\_id1, 0, 1); // 空闲的**  **sem\_init(&sem\_id2, 0, 0); // 忙的**  pthread\_create(&id1,NULL,thread\_white\_fun, NULL);  pthread\_create(&id2,NULL,thread\_black\_fun, NULL);  pthread\_join(id1,NULL);  pthread\_join(id2,NULL);  printf("main,,,\n");  return 0;  } |

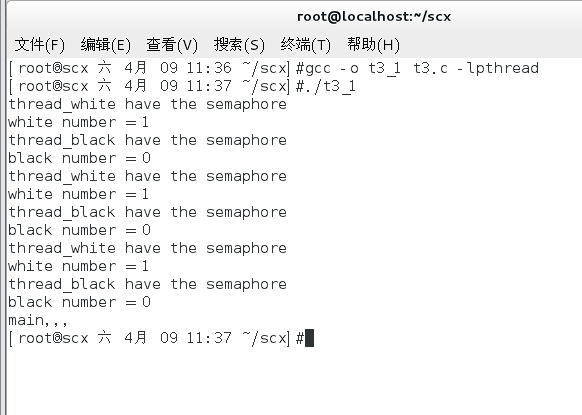
编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：：

1. 理解信号量的初始化操作、P操作和V操作；

|  |
| --- |
| **信号量的初始化**  **sem\_init(&sem\_id1, 0, 1); // 空闲的**  **sem\_init(&sem\_id2, 0, 0); // 忙的**  **P操作**  **sem\_wait(&sem\_id1);**  **sem\_wait(&sem\_id2);**  **V操作**  **sem\_post(&sem\_id2);**  **sem\_post(&sem\_id1);** |

1. 修改信号量的初始值，观察线程同步的顺序变化。

**首先先给出不修改前的运行结果**

****

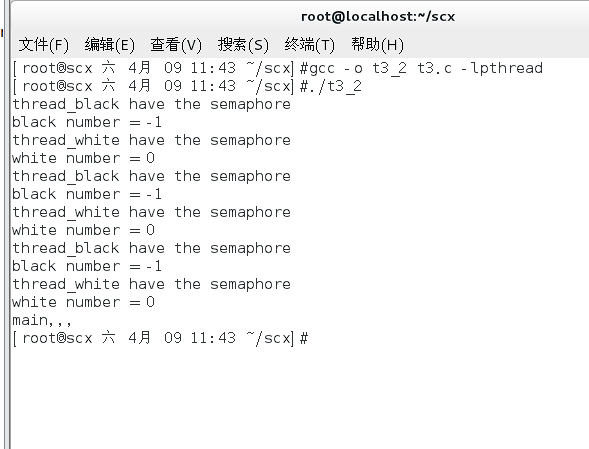
|  |
| --- |
| 可以清楚的看到，白棋优先进行，然后黑棋，再然后白棋。实现了线程的同步运行。  首先是白棋有通行证1，然后白棋的线程先运行消耗掉通行证1，在运行结束之后，给黑棋通行证2，然后黑棋再继续运行消耗掉通行证2，运行结束之后，再给白棋通行证1.如此循环往复，同步运行。 |

**下面给出修改后的运行结果**

**修改的代码为**

|  |
| --- |
| sem\_init(&sem\_id1, 0, 0); *//忙的*      sem\_init(&sem\_id2, 0, 1); *//空闲的* |

**运行结果为**

****

|  |
| --- |
| 与先前的运行结果不同的是，这里黑棋先运行，然后白棋，再然后黑棋。  首先是黑棋有通行证2，然后黑棋的线程先运行消耗掉通行证2，在运行结束之后，给白棋通行证1，然后白棋再继续运行消耗掉通行证1，运行结束之后，再给黑棋通行证2.如此循环往复，同步运行。 |

1. 用Posix信号量解决“写者优先的读者-写者问题”

使用信号量解决写者优先的读者-写者问题的方法如下：

|  |  |
| --- | --- |
| 互斥信号量：wsem =1，x = 1；  共享变量：readcount = 0； | |
| Reader process | Writer process |
| while (true) {  P(x);  readcount = readcount + 1;  if (readcount==1) P(wsem);  V(x);  READ file;  P(x);  readcount = readcount - 1;  if (readcount==0) V(wsem);  V(x); } | while (true) {  P(wsem);  WRITE file;  V(wsem);  } |

代码如下：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #include <semaphore.h>  #include <sys/types.h>  #define READERCOUNT 5 //读者数  #define WRITERCOUNT 3 //写者数  #define PAPERSIZE 26 //文件长度  static char paper[PAPERSIZE]; //文件  unsigned short int write\_index = 0; //写者需要写入的位置  char ch = 'A'; //写者需要写入的字母  sem\_t rmutex,wmutex;  int nReader = 0;  void \*reader(void \*args) //读者线程  {  int number =\*((int \*)args);  int i;  for(i = 0;i <10;i++){  **sem\_wait(&rmutex);**  if (nReader == 0) //如果是第一个读者, 则锁定wmutex  **sem\_wait(&wmutex);**  ++ nReader;  **sem\_post(&rmutex);**    //开始读  printf("## reader %d was reading...\n", number);  printf("text: %s\n", paper);sleep(1);  printf(" reader %d end reading...\n\n", number);    **sem\_wait(&rmutex);**  -- nReader;  if (nReader == 0) //如果是最后一个读者, 则解锁wmutex  **sem\_post(&wmutex);**  **sem\_post(&rmutex);**  sleep(1);  }  pthread\_exit(NULL);  }  void \*writer(void \*args) //写者线程  {  int number =\*((int \*)args);  int i;  for(i = 0;i <9;i++){  **sem\_wait(&wmutex);**  //获取写锁  printf("++ writer %d was writing...%c\n", number,ch); //start writing  paper[write\_index] = ch; sleep(1);  write\_index = (write\_index+1)%26;  ch = ch+1;  if (ch > 'Z') ch = 'A';  printf(" writer %d end writing...\n\n", number); //end writing  **sem\_post(&wmutex);** //释放写锁  sleep(1);  }  pthread\_exit(NULL);  }    int main()  {  int i;  int rThdNum[READERCOUNT], wThdNum[WRITERCOUNT];  pthread\_t wthread[WRITERCOUNT], rthread[READERCOUNT];    **sem\_init(&rmutex, 0, 1);**  **sem\_init(&wmutex, 0, 1);**  for (i = 0; i < READERCOUNT; i++)  rThdNum[i] = i;  for (i = 0; i < WRITERCOUNT; i++)  wThdNum[i] = i;  for ( i = 0; i < READERCOUNT; ++i)  pthread\_create(&rthread[i], NULL, reader,(void \*)&rThdNum[i]);  for (i = 0; i <WRITERCOUNT; ++i)  pthread\_create(&wthread[i], NULL, writer,(void \*)&wThdNum[i]);  for (i = 0; i < READERCOUNT; ++i)  pthread\_join(rthread[i], NULL);  for (i = 0; i < WRITERCOUNT; ++i)  pthread\_join(wthread[i], NULL);  } |

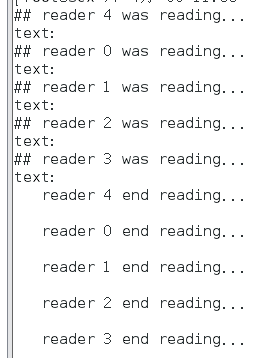
编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：：

1. 理解信号量的初始化操作、P操作和V操作；

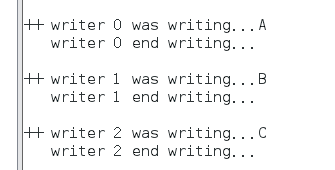
|  |
| --- |
| **这道题的P操作V操作和先前的有些不同。下面给出分析**  **首**先是信号量的初始化代码      sem\_init(&rmutex, 0, 1);      sem\_init(&wmutex, 0, 1);  第一个通行证用来保证读者数量增加时，不会造成混乱。  第二个通行证用来保护读者和写者不能同时运行，同时写者不能有多个同时运行  **然后进行分析分析P操作中，如何实现读者优先（写着线程的PV操作和之前的几道题的类似，再次不做过多分析，主要讲解读者优先的实现）**          sem\_wait(&rmutex);  *//如果是第一个读者, 则锁定wmutex*          if (nReader == 0)              sem\_wait(&wmutex);          ++ nReader;          sem\_post(&rmutex);  **因为读者数量也是一个公共资源，所以首先要给这个公共资源上锁（rmutex），然后呢。在读者线程运行的时候，希望读者线程可以多个同时使用临界资源（要读取的文件paper）**  **因此，我们只需要让第一个进入的读者，上锁（wmutex）即可**  **读者的V操作和P操作类似**          sem\_wait(&rmutex);          -- nReader;  *//如果是最后一个读者, 则解锁wmutex*          if (nReader == 0)              sem\_post(&wmutex);          sem\_post(&rmutex);          sleep(1);  **首先还是，要确定读者数量，因此nReader这个公共资源在使用的时候要上锁，然后使用完之后，用sem\_post()这个V操作进行下锁。然后只有最后一个读者的时候，我们再进行wmutex的下锁。** |

1. 观察线程并发，理解读者优先。

**下面给出我的运行结果，运行结果很长，进行分块解释**

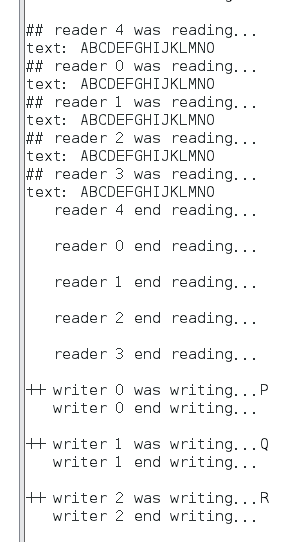
****

刚开始读者先运行了，因此多个读者线程同时访问了公共资源paper，并且此时没有写着往paper里面写入数据，因此读者什么都没读到

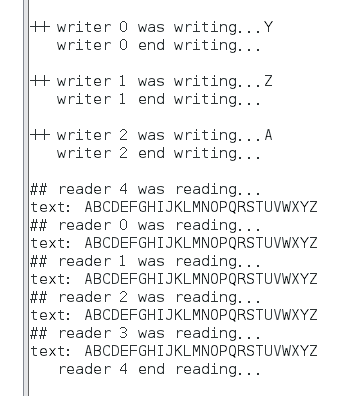


有三个读者往里面写入数据了，并且可以观察到，读者0写完之后读者1才能继续往里面写入数据。三个读者不能同时访问临界资源paper

接下来的运行结果都和前面分析的一样，读者共同读，写者只能一个写完，然后另一个写



**到最后的时候，代码有一点点不一样**



**写者们在写完二十六个英文字母之后，又从头开始写A，且文件长度固定，写的位置又回到了最开头**

1. 用POSIX信号量解决生产者/消费者问题

使用信号量解决有限缓冲区生产者/消费者问题的方法如下：

|  |  |
| --- | --- |
| 同步信号量：EMPTY=k；FULL = 0；  互斥信号量：MUTEX =1； | |
| 生产者 | 消费者 |
| while (true)  {  produce(x);  P（EMPTY）；  P（MUTEX）；  append(x,Q);  V（MUTEX）；  V（FULL）；  } | while (true)  {  P（FULL）；  P（MUTEX）；  take(x,Q);  V（MUTEX）；  V（EMPTY）；  consume(x);  } |

代码如下：

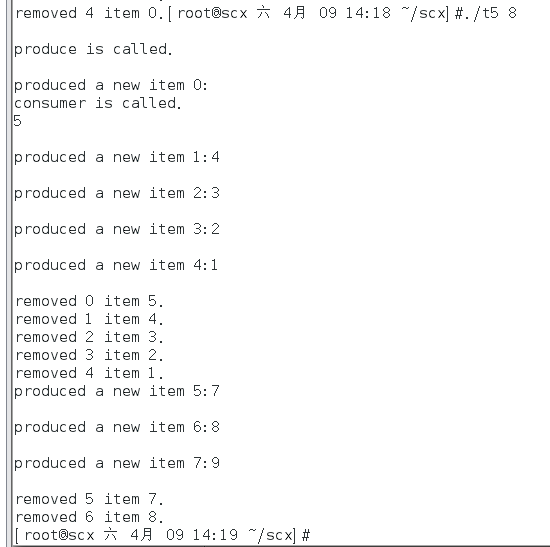
|  |
| --- |
| #include <stdlib.h>  #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  #include <semaphore.h>  #include <errno.h>  #include <fcntl.h>  #define NBUFF 5 //空槽的个数  #define SEM\_MUTEX "mutex1"  #define SEM\_NEMPTY "nemtpy1"  #define SEM\_NSTORED "nstored1"    int nitems; //产品的个数  struct  {  int buff[NBUFF];  sem\_t \*mutex,\*nempty,\*nstored; //信号量  }shared; //缓冲区结构    char \*px\_ipc\_name(const char \*name);  void \*produce(void \*arg);  void \*consume(void \*arg);    int main(int argc,char \*argv[])  {  pthread\_t tid\_produce,tid\_consume;  if(argc != 2)  {  printf("usage: prodcons <#itmes>");  exit(0);  }  nitems = atoi(argv[1]); //获取产品数目  //创建互斥信号量  if((shared.mutex=**sem\_open**(SEM\_MUTEX,O\_CREAT,0,1)) == SEM\_FAILED )  {  perror("sem\_open() error");  exit(-1);  }  //创建nempty信号量  if((shared.nempty=**sem\_open**(SEM\_NEMPTY,O\_CREAT,0,NBUFF))==SEM\_FAILED)  {  perror("sem\_open() error");  exit(-1);  }  //创建nstored信号量  if((shared.nstored=**sem\_open**(SEM\_NSTORED,O\_CREAT,0,0))==SEM\_FAILED)  {  perror("sem\_open() error");  exit(-1);  }  pthread\_setconcurrency(2);  pthread\_create(&tid\_produce,NULL,produce,NULL); //生产者线程  pthread\_create(&tid\_consume,NULL,consume,NULL); //消费者线程  pthread\_join(tid\_produce,NULL);  pthread\_join(tid\_consume,NULL);  sem\_unlink(SEM\_MUTEX);  sem\_unlink(SEM\_NEMPTY);  sem\_unlink(SEM\_NSTORED);  exit(0);  }  void \*produce(void \*arg)  {  int i;  printf("\nproduce is called.");  for(i=0;i<nitems;i++)  {  **sem\_wait(shared.nempty);** //判断是否有空槽，有的将其减少1  **sem\_wait(shared.mutex);**  //锁住缓冲区  printf("\nproduced a new item %d:",i);  scanf("%d",&shared.buff[i%NBUFF]);  sleep(1);  **sem\_post(shared.mutex);**  //释放缓冲区  **sem\_post(shared.nstored);**  //缓冲区中条目数加1  }  return NULL;  }  void \*consume(void \*arg)  {  int i;  printf("\nconsumer is called.");  for(i=0;i<nitems;i++)  {  **sem\_wait(shared.nstored); /**/判断缓冲区中是否有条目，若有产品数减1  **sem\_wait(shared.mutex);** //锁住缓冲区  if(shared.buff[i % NBUFF] != i)  printf("\nremoved %d item %d.",i,shared.buff[i% NBUFF]);  sleep(1);  **sem\_post(shared.mutex);**  //释放缓冲区  **sem\_post(shared.nempty);** //将缓冲区中的空槽数加1  }  return NULL;  } |

编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：：

1. 理解信号量的初始化操作、P操作和V操作；

|  |
| --- |
| **本题的信号量初始化如下**      sem\_init(&mutex,0,1);      sem\_init(&empty,0,NBUFF);      sem\_init(&full,0,0);  mutex是为了消费者不会同时使用临界资源，生产者不会同时使用临界资源  P操作如下          sem\_wait(&empty); *//判断是否有空槽，有的将其减少1*  生产者有空槽才能生产          sem\_wait(&mutex);  *//锁住槽位，对于多个生产者的时候有必要，单个生产者没有必要*  锁住槽位，这样生产者不会同时使用槽位          sem\_wait(&full);  消费者有商品才能消费          sem\_wait(&mutex);  所租槽位，消费者不会同时进行消费  V操作如下          sem\_post(&mutex);  释放槽位          sem\_post(&full);  产品加一          sem\_post(&empty);  空槽加一 |

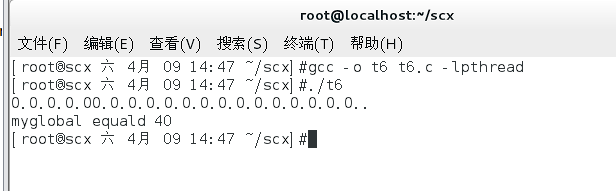
1. 观察线程并发，理解生产者消费者在信号量控制下，实现供需平衡。

****

|  |
| --- |
| **首先输入的参数为8，意思是生产者和消费者各进行八次。**  **然后输入物显示生成了五件商品5 4 3 2 1**  **当生产五件商品之后，Empty为0，生产者无法继续生产，因此消费者进行消费。当消费者消费了五件商品之后，产品数量Full为0，因此消费者无法进行消费，生产者继续生产三件商品7 8 9然后消费者再消费三件商品** |

1. **编程题：理解前述例程后，按要求完成程序编写。**
2. 运行课后习题4.9，可发现输出结果有误。请使用POSIX无名信号量，修改代码，保证程序能输出正确结果。

先给出我的运行结果

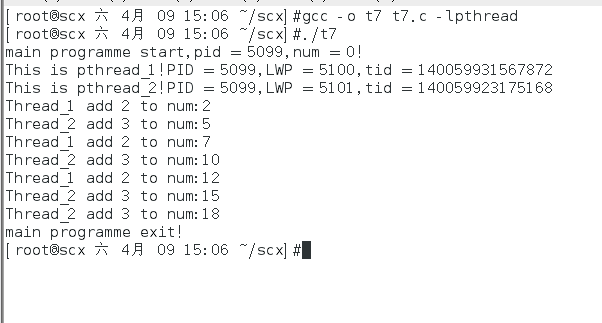


代码如下

|  |
| --- |
| #include <stdlib.h>  #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  #include <semaphore.h>  #include <errno.h>  #include <fcntl.h>  sem\_t sem;  int myglobal=0;  void \*thread\_function(void \*arg)  {  int i,j;  for(i = 0; i <20; i++){  sem\_wait(&sem);  //pthread\_mutex\_lock(&mutex);  j = myglobal;  j++;  printf(".");  fflush(stdout);  myglobal = j;  //pthread\_mutex\_unlock(&mutex);  sem\_post(&sem);  usleep(2000);    }  return NULL;  }  int main(){  pthread\_t mythread;  //pthread\_mutex\_init (&mutex, NULL);  sem\_init(&sem,0,1);  int i;  if( pthread\_create(&mythread,NULL,thread\_function,NULL))  {  printf("error creating thread!\n");  abort();  }  for(i = 0; i <20; i++)  {  sem\_wait(&sem);  //pthread\_mutex\_lock(&mutex);  myglobal++;  printf("O");  fflush(stdout);  //pthread\_mutex\_unlock(&mutex);  sem\_post(&sem);  usleep(2000);  }  if( pthread\_join(mythread,NULL))  {  printf("error joining thread!\n");  abort();  }  printf("\nmyglobal equald %d\n",myglobal);  exit(0);  } |

1. 理解课后习题5.4，编写完整的程序，使用POSIX线程实现并发，使用POSIX信号量实现并发线程互斥运行。

**首先给出我的运行结果**

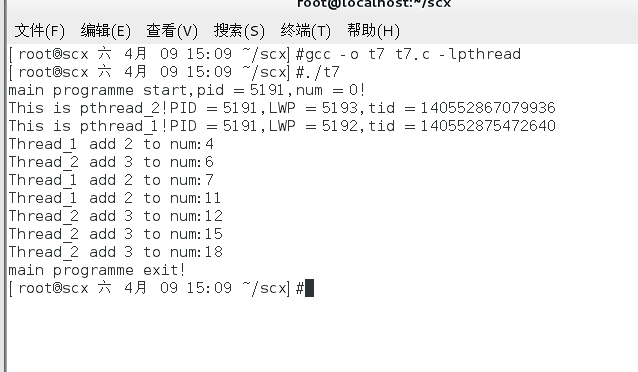
****

|  |
| --- |
| **#include <stdlib.h>**  **#include <pthread.h>**  **#include <stdio.h>**  **#include <unistd.h>**  **#include <semaphore.h>**  **#include <errno.h>**  **#include <fcntl.h>**  **#include <sys/types.h>**  **#include <sys/syscall.h>**  **sem\_t sem;**  **int gnum = 0;**  **static void pthread\_add2(void);**  **static void pthread\_add3(void);**  **pid\_t gettid() {**  **return syscall(SYS\_gettid);**  **}**  **int main(void)**  **{**  **/\*线程的标识符\*/**  **pthread\_t pt\_1 = 0;**  **pthread\_t pt\_2 = 0;**  **int ret = 0;**  **printf("main programme start,pid = %d,num = %d!\n", getpid(), gnum);**  **/\*互斥初始化\*/**  **sem\_init(&sem, 0, 1);**  **/\*分别创建线程1、2\*/**  **ret = pthread\_create(&pt\_1, NULL, (void \*)pthread\_add2, NULL);**  **ret = pthread\_create(&pt\_2, NULL, (void \*)pthread\_add3, NULL);**  **/\*等待线程1、2的结束\*/**  **pthread\_join(pt\_1, NULL);**  **pthread\_join(pt\_2, NULL);**  **printf("main programme exit!\n");**  **return 0;**  **}**  **/\*线程1的服务程序\*/**  **static void pthread\_add2(void)**  **{**  **int i = 0;**  **printf("This is pthread\_1!PID = %d,LWP = %d,tid = %lu\n", getpid(), gettid(), pthread\_self());**  **for (i = 0; i < 3; i++)**  **{**  **sem\_wait(&sem);**  **gnum++;**  **sleep(1);**  **gnum++; /\*临界资源\*/**  **printf("Thread\_1 add 2 to num:%d\n", gnum);**  **sem\_post(&sem);**  **usleep(1000);**  **}**  **pthread\_exit(NULL);**  **}**  **/\*线程2的服务程序\*/**  **static void pthread\_add3(void)**  **{**  **int i = 0;**  **printf("This is pthread\_2!PID = %d,LWP = %d,tid = %lu\n", getpid(), gettid(), pthread\_self());**  **for (i = 0; i < 4; i++)**  **{**  **sem\_wait(&sem);**  **gnum++;**  **sleep(1);**  **gnum++;**  **sleep(1);**  **gnum++; /\*临界资源\*/**  **printf("Thread\_2 add 3 to num:%d\n", gnum);**  **sem\_post(&sem);**  **usleep(1000);**  **}**  **pthread\_exit(NULL);**  **}**  **// sem\_wait(&sem);**  **// sem\_post(&sem);** |

**解释**

**如果删掉信号量有关的函数，运行结果会非常混乱。**

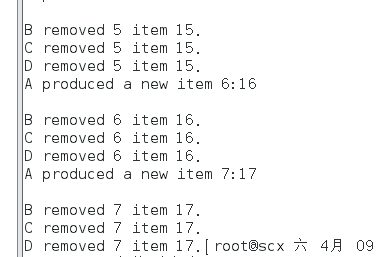
**运行结果如下（删除之后）**

****

1. 请结合生产者-消费者例程，修改代码，实现下述变形的生产者/消费者问题的：有四个线程A、B、C、D。线程A通过一个缓冲区不断地向线程B、C、D发送信息，A每向缓冲区送入一个信息后，必须等线程B、C、D都取走后才可以发送下一个信息，B、C、D对A送入的每一信息各取一次，试用P、V操作实现它们之间的正确通信。

**首先给出我的运行结果**





**代码附上**

|  |
| --- |
| **#include <stdlib.h>**  **#include <pthread.h>**  **#include <stdio.h>**  **#include <unistd.h>**  **#include <semaphore.h>**  **#include <errno.h>**  **#include <fcntl.h>**  **#define NBUFF 5 //槽位的个数**  **int nitems; //条目的个数**  **int buff[NBUFF];**  **sem\_t mutex, fullB, fullC, fullD;**  **sem\_t emptyB, emptyC, emptyD;**  **sem\_t B, C, D;**  **void \*produceA(void \*arg);**  **void \*consumeB(void \*arg);**  **void \*consumeC(void \*arg);**  **void \*consumeD(void \*arg);**  **int main(int argc, char \*argv[])**  **{**  **pthread\_t tid\_produceA, tid\_consumeB, tid\_consumeC, tid\_consumeD;**  **if (argc != 2)**  **{**  **printf("usage: prodcons <#itmes>");**  **exit(0);**  **}**  **nitems = atoi(argv[1]); //获取条目数目**  **//创建信号量**  **sem\_init(&mutex, 0, 1);**  **sem\_init(&emptyB, 0, NBUFF);**  **sem\_init(&emptyC, 0, NBUFF);**  **sem\_init(&emptyD, 0, NBUFF);**  **sem\_init(&fullB, 0, 0);**  **sem\_init(&fullC, 0, 0);**  **sem\_init(&fullD, 0, 0);**  **sem\_init(&B, 0, 1);**  **sem\_init(&C, 0, 1);**  **sem\_init(&D, 0, 1);**  **pthread\_setconcurrency(2); //设置并发级别**  **pthread\_create(&tid\_produceA, NULL, produceA, NULL); //生产者A线程**  **pthread\_create(&tid\_consumeB, NULL, consumeB, NULL); //消费者B线程**  **pthread\_create(&tid\_consumeC, NULL, consumeC, NULL); //消费者B线程**  **pthread\_create(&tid\_consumeD, NULL, consumeD, NULL); //消费者B线程**  **pthread\_join(tid\_produceA, NULL);**  **pthread\_join(tid\_consumeB, NULL);**  **pthread\_join(tid\_consumeC, NULL);**  **pthread\_join(tid\_consumeD, NULL);**  **exit(0);**  **}**  **void \*produceA(void \*arg)**  **{**  **int i;**  **printf("\nproduceA is called.\n");**  **for (i = 0; i < nitems; i++)**  **{**  **//判断是否有空槽，有的将其减少1**  **sem\_wait(&emptyB);**  **sem\_wait(&emptyC);**  **sem\_wait(&emptyD);**  **sem\_wait(&B);**  **sem\_wait(&C);**  **sem\_wait(&D);**  **sem\_wait(&mutex); //锁住槽位，对于多个生产者的时候有必要，单个生产者没有必要**  **printf("\nA produced a new item %d:", i);**  **scanf("%d", &buff[i % NBUFF]);**  **sleep(1);**  **sem\_post(&mutex); //释放锁**  **sem\_post(&fullB); //缓冲区中条目数加1**  **sem\_post(&fullC); //缓冲区中条目数加1**  **sem\_post(&fullD); //缓冲区中条目数加1**  **}**  **return NULL;**  **}**  **void \*consumeB(void \*arg)**  **{**  **int i;**  **printf("\nconsumerB is called.\n");**  **for (i = 0; i < nitems; i++)**  **{**  **sem\_wait(&fullB); //判断缓冲区中是否有条目，有的话将条目数减少1**  **sem\_wait(&mutex); //锁住缓冲区，对多个消费者有必要，对单个消费者没必要**  **if (buff[i % NBUFF] != i)**  **printf("\nB removed %d item %d.", i, buff[i % NBUFF]);**  **sleep(1);**  **sem\_post(&mutex); //释放锁**  **sem\_post(&B); //将缓冲区中的空槽数目加1**  **sem\_post(&emptyB);**  **}**  **return NULL;**  **}**  **void \*consumeC(void \*arg)**  **{**  **int i;**  **printf("\nconsumerC is called.\n");**  **for (i = 0; i < nitems; i++)**  **{**  **sem\_wait(&fullC); //判断缓冲区中是否有条目，有的话将条目数减少1**  **sem\_wait(&mutex); //锁住缓冲区，对多个消费者有必要，对单个消费者没必要**  **if (buff[i % NBUFF] != i)**  **printf("\nC removed %d item %d.", i, buff[i % NBUFF]);**  **sleep(1);**  **sem\_post(&mutex); //释放锁**  **sem\_post(&C); //将缓冲区中的空槽数目加1**  **sem\_post(&emptyC);**  **}**  **return NULL;**  **}**  **void \*consumeD(void \*arg)**  **{**  **int i;**  **printf("\nconsumerD is called.\n");**  **for (i = 0; i < nitems; i++)**  **{**  **sem\_wait(&fullD); //判断缓冲区中是否有条目，有的话将条目数减少1**  **sem\_wait(&mutex); //锁住缓冲区，对多个消费者有必要，对单个消费者没必要**  **if (buff[i % NBUFF] != i)**  **printf("\nD removed %d item %d.", i, buff[i % NBUFF]);**  **sleep(1);**  **sem\_post(&mutex); //释放锁**  **sem\_post(&D); //将缓冲区中的空槽数目加1**  **sem\_post(&emptyD);**  **}**  **return NULL;**  **}** |