# POSIX信号量

**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. **本实验涉及的系统调用**
2. 信号量初始化：int sem\_init(sem\_t \*sem,int pshared,unsigned int value);

参数说明：

sem\_t \*sem： 信号量变量；

int pshared： 指明信号量的类型。不为0时此信号量在**相关进程**间共享，否则只能为当前进程的所有线程共享；

unsigned int value：该参数指定信号量的初始值。

返回值：

成功时返回0；错误时，返回 -1，并把 errno 设置为合适的值。

1. 信号量的减1操作：int sem\_wait(sem\_t \*sem);

函数说明：

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。

1. 信号量的加1操作：int sem\_post(sem\_t \*sem);

函数说明：

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。

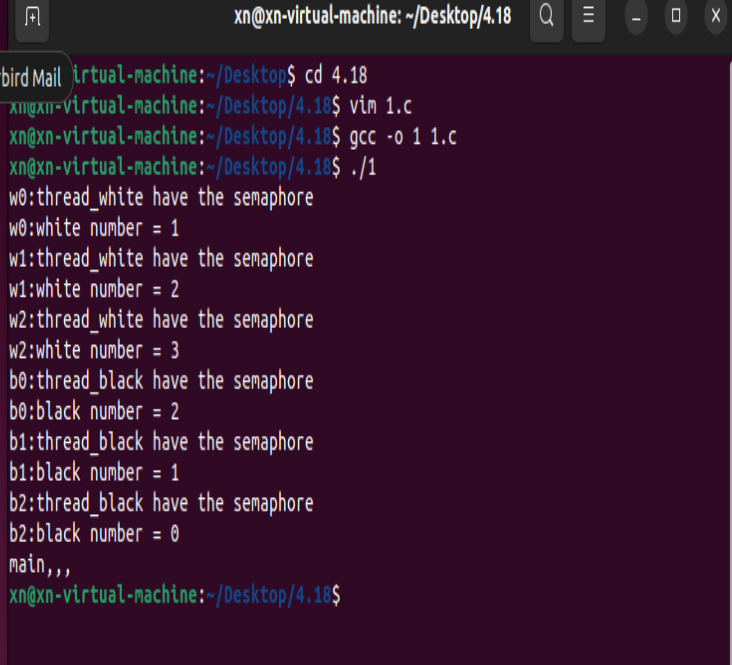
1. 获取信号量的值，一般只用来调试：int sem\_getvalue(sem\_t \*sem);
2. **并发线程的互斥控制**
3. 线程互斥例题：

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；

P操作（等待）sem\_wait(&sem\_id);

V操作（释放）sem\_post(&sem\_id);

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。

通过保留 sleep 语句和信号量操作，即使没有实际修改共享资源，信号量仍然确保每个线程都可以按预定顺序安全地“进入”和“退出”临界区。这演示了信号量的基本功能：防止资源竞争和数据冲突，确保数据一致性和程序稳定性。

1. 线程同步例题：

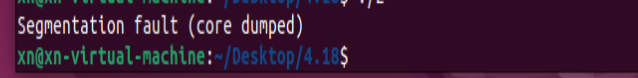
本例有四个线程。两个Read线程从文件中读取数据，两个Handle线程处理读取出来的数据。要求先读取数据，后处理数据，两组线程是同步关系。

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);  } |

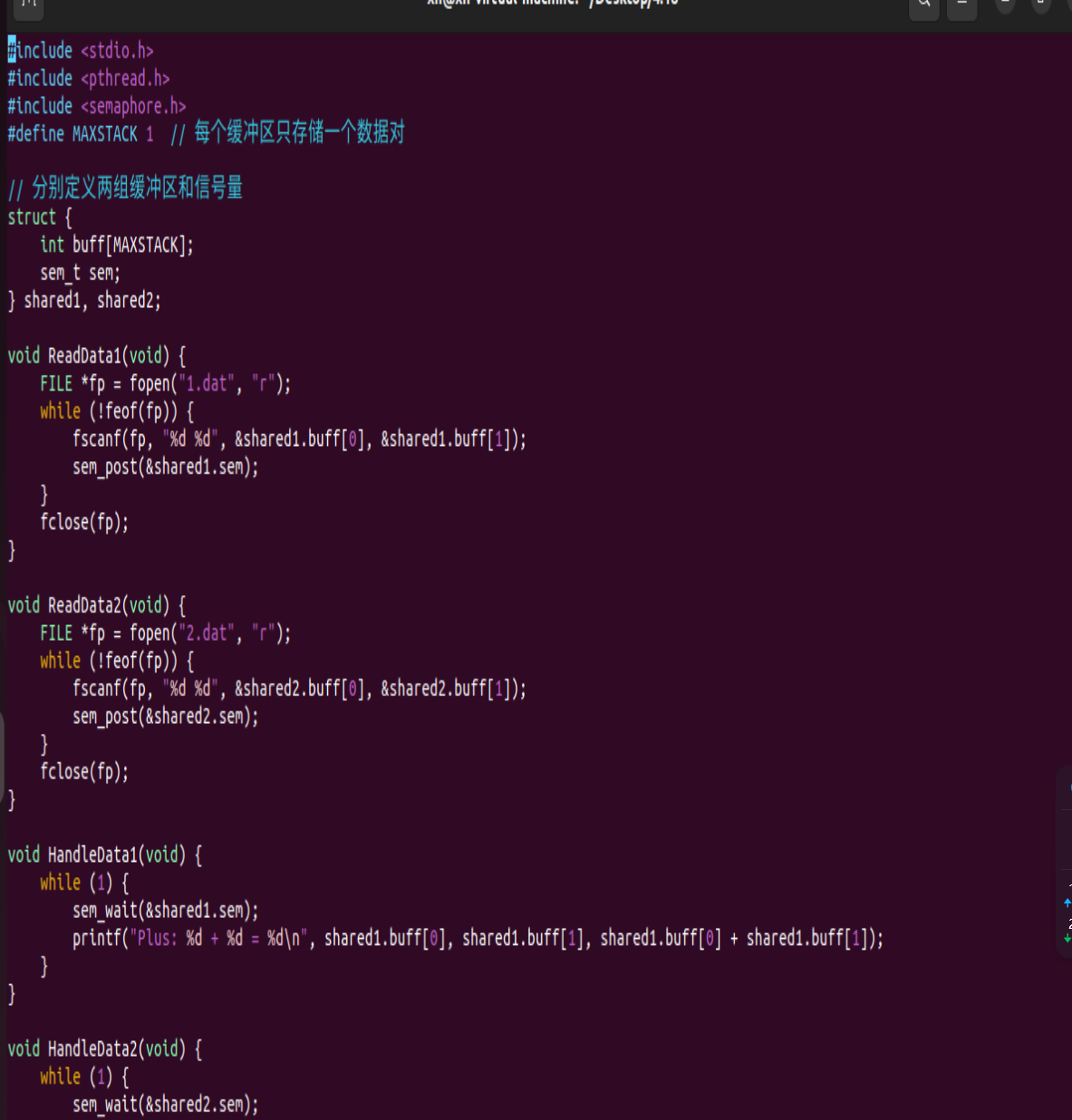
编译链接通过后，多次运行例程，观察进程并发执行结果，并思考下述问题：：

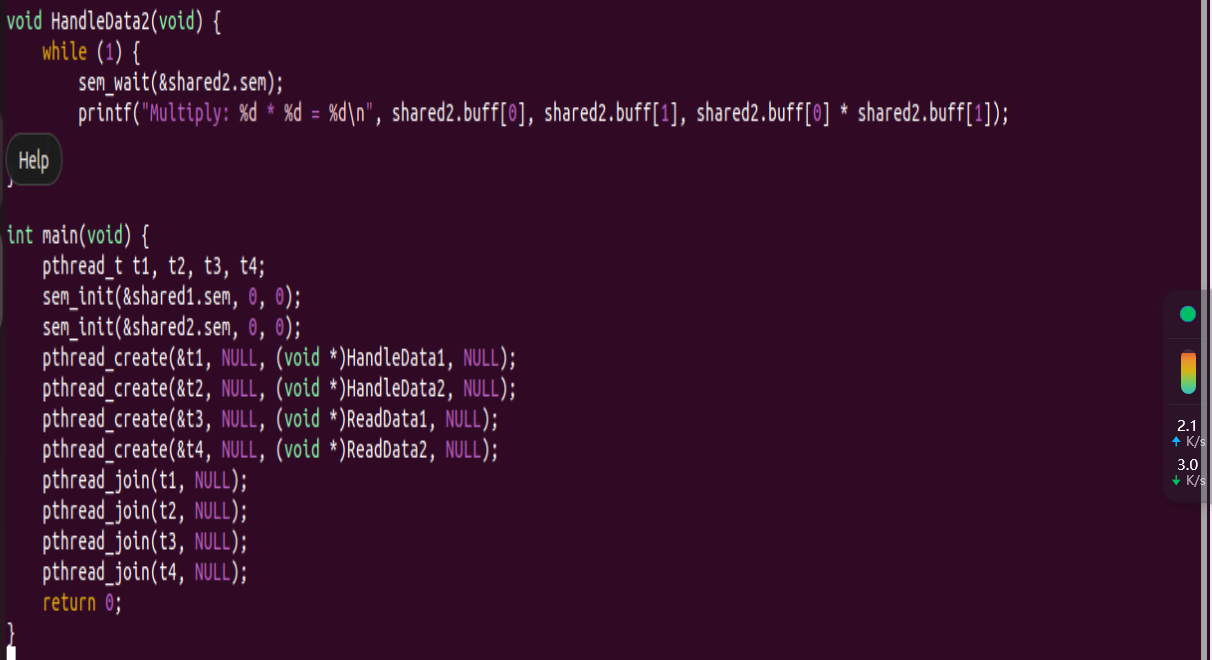


1. 理解信号量的初始化操作、P操作和V操作；
2. 分析出四个线程的前驱后继关系；

* ReadData1 和 ReadData2 是前驱动作，它们读取数据并通过sem\_post操作释放信号量。
* HandleData1 和 HandleData2 是后继动作，它们通过sem\_wait操作等待信号量，然后处理数据。
* 当前的设计允许任何处理线程处理由任何读取线程读取的数据，因为所有线程共享同一个信号量和数据缓冲区

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。

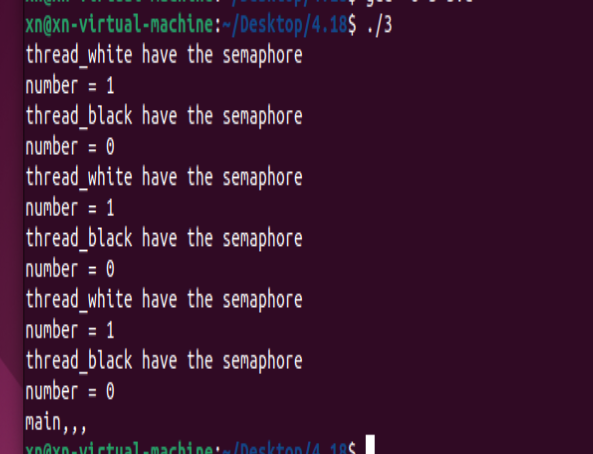




1. 捡棋子题中，如果还要求进程A和进程B交替拣子，则为进程同步问题，修改代码并调试运行。

|  |
| --- |
| #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；

信号量的初始化操作：

sem\_init(&sem\_id1, 0, 1); 初始化信号量sem\_id1，初始值为1。这意味着进程A可以立即执行。

sem\_init(&sem\_id2, 0, 0); 初始化信号量sem\_id2，初始值为0。这意味着进程B必须等待进程A通过sem\_post(&sem\_id2)释放信号量后才能执行。

P操作（等待）：

sem\_wait(&sem\_id1); 进程A等待sem\_id1。由于sem\_id1的初始值为1，进程A可以立即执行。

sem\_wait(&sem\_id2); 进程B等待sem\_id2。因为初始值为0，进程B需要等待进程A释放这个信号量。

V操作（释放）：

sem\_post(&sem\_id2); 进程A执行完毕后，通过调用sem\_post(&sem\_id2)来增加sem\_id2的值，允许进程B执行。

sem\_post(&sem\_id1); 进程B执行完毕后，通过调用sem\_post(&sem\_id1)来增加sem\_id1的值，允许进程A执行。

1. 修改信号量的初始值，观察线程同步的顺序变化。

sem\_init(&sem\_id1, 0, 0); // 忙的

sem\_init(&sem\_id2, 0, 1); // 空闲的

* 进程B（thread\_black\_fun）将首先获得执行的机会，因为sem\_id2现在是空闲的。
* 进程A将必须等待进程B释放sem\_id1后才能开始执行。

1. 用Posix信号量解决“写者优先的读者-写者问题”

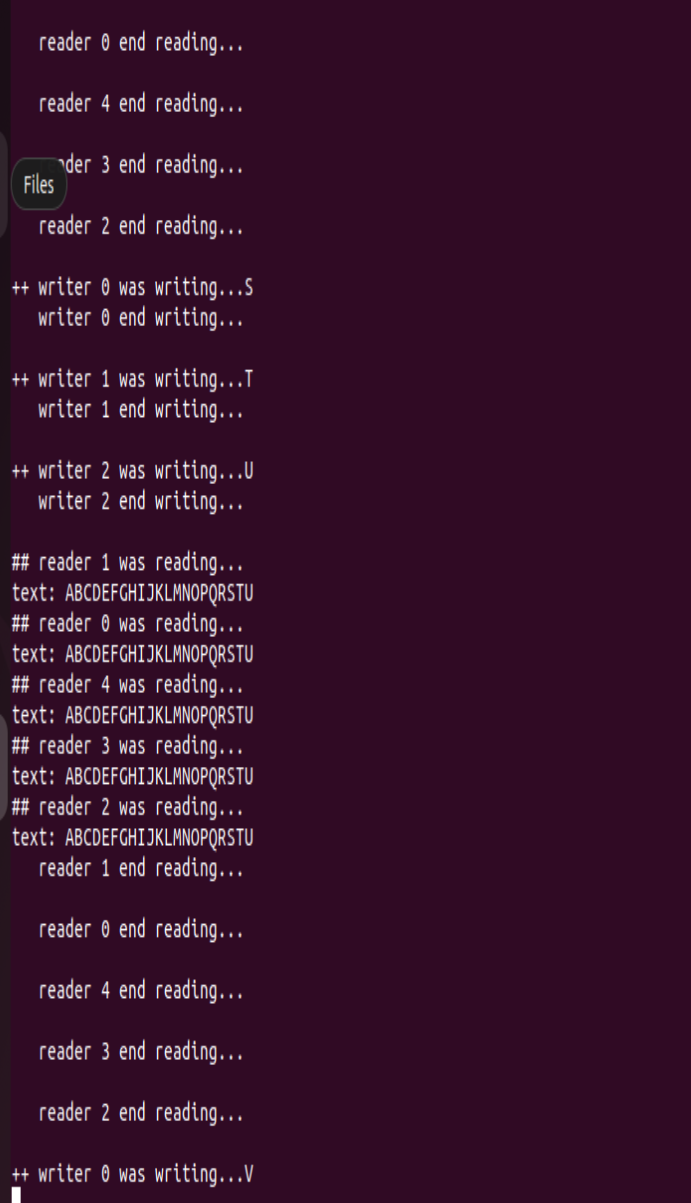
使用信号量解决写者优先的读者-写者问题的方法如下：

|  |  |
| --- | --- |
| 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；

初始化信号量：

sem\_init(&rmutex, 0, 1); 初始化用于读者计数保护的信号量，允许一个读者进入临界区修改读者计数。

sem\_init(&wmutex, 0, 1); 初始化用于写者控制的信号量，确保写者独占访问。

P操作（等待/阻塞）和V操作（释放/信号）：

sem\_wait(&rmutex); 和 sem\_post(&rmutex); 用于保护对 nReader（读者计数）的访问。

sem\_wait(&wmutex); 在读者数量为0时，写者尝试获取写锁，这防止其他写者或读者进入。

sem\_post(&wmutex); 写者完成写操作后释放锁，允许其他写者或读者进入。

1. 观察线程并发，理解读者优先。

读者优先机制：在代码中，只要有读者在读，后续的读者可以继续进入读，即使有写者在等待。写者必须等到所有读者离开（nReader 降至0）才可以开始写。

并发行为：运行程序时，观察到多个读者可能会同时进行读操作，而写者会等待所有读者完成后才开始写。如果读者不断地来读，写者可能会长时间等待，这就是典型的“读者优先”策略。

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；

初始化信号量:

sem\_open(SEM\_MUTEX, O\_CREAT, 0, 1); 创建并初始化互斥信号量MUTEX，初始值为1。这保证了在任一时刻，只有一个线程（生产者或消费者）可以访问缓冲区。

sem\_open(SEM\_NEMPTY, O\_CREAT, 0, NBUFF); 创建并初始化信号量EMPTY，其初始值为缓冲区的大小，表示有多少空槽可用于生产者放置新产品。

sem\_open(SEM\_NSTORED, O\_CREAT, 0, 0); 创建并初始化信号量FULL，初始值为0，表示开始时缓冲区中没有任何产品。

P操作和V操作：

生产者：

sem\_wait(shared.nempty); 生产者在添加新产品之前检查是否有空槽（EMPTY > 0）。如果没有空槽，生产者线程将阻塞。

sem\_wait(shared.mutex); 进入临界区前获取互斥锁。

sem\_post(shared.mutex); 离开临界区后释放互斥锁。

sem\_post(shared.nstored); 产品放置后，增加FULL的计数，表示有产品可供消费。

消费者：

sem\_wait(shared.nstored); 消费者在取产品前检查是否有产品（FULL > 0）。如果没有品，消费者线程将阻塞。

sem\_wait(shared.mutex); 进入临界区前获取互斥锁。

sem\_post(shared.mutex); 离开临界区后释放互斥锁。

sem\_post(shared.nempty); 取出产品后，增加EMPTY的计数，表示空槽增加。

2)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。

信号量的初始化操作、P操作和V操作:

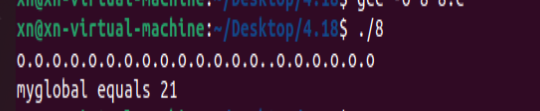
信号量初始化设定了开始时的同步条件。P操作（sem\_wait）用于请求资源，如果所请求的资源不可用，则线程将阻塞。V操作（sem\_post）用于释放资源，允许其他阻塞的线程继续执行。

线程并发和供需平衡:

多次运行程序可以观察到生产者和消费者如何互相协作。生产者不会在缓冲区满时生产新产品，而消费者则不会在缓冲区空时尝试消费。

此设计确保了生产者和消费者不会因为竞争同一资源而导致的数据不一致问题。

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。



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。
2. 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。

