实验四 实验报告

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **课程名称** | 操作系统 | **指导教师** | 叶晖 | **实验日期** | 2022年4月25日 |
| **班级** | 20软件02班 | **姓名** | 李佳骏 | **学号** | B20190103224 |
| **实验成绩** |  | | | | |
| **一、实验名称**  P，V原语应用程序 | | | | | |
| **二、实验目的**  (1) 掌握信号量的原理及 P、V 操作。  (2) 了解生产者与消费者问题，并通过信号量解决单缓冲区生产者－消费者问题和 m 个生产者和 n 个消费者共享 k 件产品缓冲区的问题。 | | | | | |
| **三、实验环境**  (1) 硬件环境：Intel Pentium III 以上 CPU，128MB 以上内存，2GB 以上硬盘  (2) 软件环境：Ubuntu 13.04 操作系统。  注：Ubuntu 是 Linux 的一个免费发行的流行版本，系英国"科能软件股份有限公司  (Canonical)"开发 | | | | | |
| **四、实验内容**  (1) 熟悉信号量的操作，及如何实现进程的互斥与同步。  (2) 了解生产者与消费者问题。  (3) 掌握解决生产者与消费者问题的伪代码。  (4) 查看程序演示结果，并分析各个过程及现象。 | | | | | |
| **五、实验原理**  (1) 实验原理：在操作系统中，用信号量表示物理资源的实体，是一个与队列有关的整型变量。信号量仅存在赋值、P、V 三种操作，且 P、V 操作为不可分割的原子性操作。可通过信号量实现进程的互斥、进程的同步。 | | | | | |
| **六、实验步骤**  【**任务 1**】  (1) 思考：在上述单缓冲区生产者－消费者模型中，消费者进程 consumer()的 V(empty)和consume()能否互换位置，并说明理由。  (2) 思考：在 m 个生产者和 n 个消费者共享 k 个缓冲区时，能否将生产者的 P(empty)和P(mutex)互换位置，并说明理由。能否将生产者的 V(mutex)和 V(full)互换位置，并说明理由  【**任务 2**】  （1） 程序设计  用信号量及共享存储实现 m 个生产者 n 个消费者共享 k 个缓冲区的程序  （2）  上机操作  用 gedit 输入源代码。  编译、链接程序，生成可执行文件。  打开 5 个终端，其中 3 个终端执行 muti\_producer，2 个终端执行 muti\_consumer。  按程序提示操作  观察程序的结果 | | | | | |
| **七、实验结果**  任务一: 1.答案：不能互换位置，因为这会引起效率问题。当消费者将产品从缓冲区中取出后（即  P(full）)，即可释放缓冲区(V(empty))，以让生产者可以继续往缓冲区中放入产品，  而不必等到消费者消费(consume())完后，再释放缓冲区，否则，因为消费者消费速度  太慢，将会导致有可用的缓冲区时，生产者仍不能生产  2.答案：不能将生产者的 P(empty)和 P(mutex)互换位置。否则，将会导致生产者将 buffer锁住(P(mutex))后，执行 P(empty)时，因没有足够的 empty 缓冲区，而导致生产者进程阻塞，并且其它生产者与消费者会因缓冲区锁住而无法继续执行发生阻塞，导致死锁。从效率上讲，不能将 V(mutex)和 V(full)互换位置，因为首先执行 V(mutex)解锁操作后，将会唤醒与此缓冲区相关的等待队列进程中的一个。  任务二:  1.程序设计  muti\_producer代码如下：  /\* 文件名: muti\_producer.c  \* 功能: 生产者  \* 模拟多个生产者时，只要将该文件编译后的可执行程序在多个终端分别执行即可  \*/  #include <errno.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/sem.h>  #include <sys/shm.h>  #include <sys/types.h>  #include <unistd.h>  #define SEMNUM 3  #define SHMSIZE 900  typedef union semun {  int val;  struct semid\_ds \*buf;  short int \*array;  } SEMUN;  int GetKey();  int GetShm(int key, int \*shmid, void \*\*shmaddr);  int GetSemCollection(int key, int \*semid);  int Append(void \*AppendFirstAddr, char \*buffer, int pos);  int V(const int semid, char \*PType, int len);  int P(const int semid, char \*PType, int len);  int Produce(char \*buf);  int main() {  int semid = 0;  int shmid = 0;  int key = 0;  int num\_of\_production = 0; /\* Num Of Production \*/  int pos = 0;  char \*AppendFirstAddr = NULL;  int i = 0;  char buffer[SHMSIZE];  char \*shmaddr = NULL;  short int array[SEMNUM];  memset(buffer, 0x00, sizeof(buffer));  memset(array, 0x00, sizeof(array));  printf("<------------------------------------------>\n");  printf("The producer's ID is [%d]\n", getpid());  if ((key = GetKey()) < 0) {  printf("Get Key Failed\n");  return -1;  }  printf("key[%d]\n", key);  if (GetSemCollection(key, &semid) < 0) {  printf("Get Sem Failed[%d]\n", errno);  return -1;  }  printf("semid[%d]\n", semid);  if (GetShm(key, &shmid, &shmaddr) < 0) {  printf("Get Shm Failed[%d]\n", errno);  return -1;  }  printf("shmid[%d], shmaddr[%d]\n", shmid, shmaddr);  printf("<------------------------------------------>\n");  /\* Init the shm's first sizeof(int) unit to 0 \*/  memcpy(&pos, shmaddr, sizeof(int));  AppendFirstAddr = shmaddr + 2 \* sizeof(int);  printf("AppendFirstAddr[%s] pos[%d]\n", AppendFirstAddr, pos);  while (1) {  memset(buffer, 0x00, sizeof(buffer));  printf("\n");  /\* Produce Production \*/  if (semctl(semid, 0, GETALL, array) < 0) {  printf("Get Sem Val Failed\n");  return -1;  }  printf("The resource of the system's list:\n");  for (i = 0; i < SEMNUM; i++) {  if (i == 0) {  printf("empty position=[%d]\n", array[i]);  } else if (i == 1) {  printf("full position=[%d]\n", array[i]);  } else if (i == 2) {  printf("mutex=[%d]\n", array[i]);  }  }  printf("\n");  if (Produce(buffer) < 0) {  printf("Get Shm Failed[%d]\n", errno);  return -1;  }  if (strncmp(buffer, "quit", 4) == 0) {  printf("All producer's Sem and shm will be over\n");  break;  }  num\_of\_production = strlen(buffer);  printf("---->Producer[%d] produce [%d] productions:[%s]<----\n\n",  getpid(), num\_of\_production, buffer);  printf(  "Producer [%d] Requiring [%d] shop's position to put the "  "production \n",  getpid(), num\_of\_production);  /\* Get Empty Shared Memory \*/  if (P(semid, "empty", num\_of\_production) < 0) {  printf("P full Operation Failed\n");  return -1;  }  printf("Producer [%d] Get [%d] position of the shop\n", getpid(),  num\_of\_production);  printf(  "Producer[%d] Requring the right(mutex) to put the production to "  "the position... \n",  getpid());  /\* Mutex \*/  if (P(semid, "mutex", 1) < 0) {  printf("P mutex Operation Failed\n");  return -1;  }  printf(  "Producer[%d] Get the right(mutex) to put the production to right "  "position\n",  getpid());  memcpy(&pos, shmaddr, sizeof(int));  printf("----->pos[%d]<------\n", pos);  if (Append(AppendFirstAddr, buffer, pos) < 0) {  printf("Append Production To Shared Memory Failed\n");  return -1;  }  printf(  "Producer [%d] put the production [%s] to the position of the "  "shop\n",  getpid(), buffer);  printf("(The production of the shop is )Shared Memory Content[%s]\n",  AppendFirstAddr);  pos = (pos + num\_of\_production) % SHMSIZE;  memcpy(shmaddr, &pos, sizeof(int));  V(semid, "mutex", 1);  printf(  "Producer [%d] release the operation right(mutex) of the position "  "of the shop\n",  getpid());  V(semid, "full", num\_of\_production);  printf("Now Consumer can take [%d] more of production[%s] to consume\n",  num\_of\_production, buffer);  }  semctl(semid, 0, IPC\_RMID);  shmdt(shmaddr);  if (shmctl(shmid, IPC\_RMID, NULL) < 0) {  printf("RM Shm Failed[%d]\n", errno);  return -1;  }  return 0;  }  int GetKey() {  char filepath[128];  char cmd[128];  int key = 0;  memset(filepath, 0x00, sizeof(filepath));  memset(cmd, 0x00, sizeof(cmd));  sprintf(filepath, "%s/key.ini", getenv("HOME"));  sprintf(cmd, "touch %s", filepath);  if (system(cmd) < 0) {  printf("Create Key File Failed\n");  return -1;  }  if ((key = ftok(filepath, 20)) < 0) {  printf("Create Key Failed\n");  return -1;  }  return key;  }  int GetSemCollection(int key, int \*semid) {  int lsemid = 0;  SEMUN unsem;  short int array[3];  if ((lsemid = semget(key, SEMNUM, IPC\_CREAT | 0666)) < 0) {  printf("Get Sem ID Failed\n");  return -1;  }  array[0] = SHMSIZE; /\* Empty Buffer Size \*/  array[1] = 0; /\* Full Buffer Size \*/  array[2] = 1; /\* For Mutex \*/  unsem.array = array;  if (semctl(lsemid, 0, SETALL, unsem) < 0) {  printf("Init Sem Failed\n");  return -1;  }  \*semid = lsemid;  return 0;  }  int GetShm(int key, int \*shmid, void \*\*shmaddr) {  int lshmid = 0;  int shmsize = SHMSIZE + 2 \* sizeof(int);  char \*lshmaddr = NULL;  /\* shmsize=SHMSIZE+2\*sizeof(int);\*/  if ((lshmid = shmget(key, shmsize, IPC\_CREAT | 0666)) < 0) {  printf("Get Shared Memory Failed\n");  return -1;  }  if ((lshmaddr = (char \*)shmat(lshmid, 0, 0)) == (char \*)-1) {  printf("Attach Shared Memory Failed, PID[%d]\n", getpid());  return -1;  }  \*shmid = lshmid;  \*shmaddr = lshmaddr;  return 0;  }  int Produce(char \*buf) {  if (buf == NULL) {  printf("Produce Buffer Couldn't Be NULL\n");  return -1;  }  printf("Producer[%d] Produce Production(Please Input):\n");  gets(buf);  return 0;  }  int P(const int semid, char \*PType, int len) {  struct sembuf stSembuf;  memset(&stSembuf, 0x00, sizeof(stSembuf));  if (PType == NULL) {  printf("PType Cann't Be NULL\n");  return -1;  }  if (strcmp(PType, "empty") == 0) {  stSembuf.sem\_num = 0;  } else if (strcmp(PType, "full") == 0) {  stSembuf.sem\_num = 1;  } else if (strcmp(PType, "mutex") == 0) {  stSembuf.sem\_num = 2;  } else {  printf("Sem Type Wrong\n");  return -1;  }  stSembuf.sem\_op = (-1) \* len;  stSembuf.sem\_flg = SEM\_UNDO;  if (semop(semid, &stSembuf, 1) < 0) {  printf("P [%s] Operation Failed\n", PType);  return -1;  }  return 0;  }  int V(const int semid, char \*PType, int len) {  struct sembuf stSembuf;  memset(&stSembuf, 0x00, sizeof(stSembuf));  if (PType == NULL) {  printf("PType Cann't Be NULL\n");  return -1;  }  if (strcmp(PType, "empty") == 0) {  stSembuf.sem\_num = 0;  } else if (strcmp(PType, "full") == 0) {  stSembuf.sem\_num = 1;  } else if (strcmp(PType, "mutex") == 0) {  stSembuf.sem\_num = 2;  } else {  printf("Sem Type Wrong\n");  return -1;  }  stSembuf.sem\_op = len;  stSembuf.sem\_flg = SEM\_UNDO;  if (semop(semid, &stSembuf, 1) < 0) {  printf("P [%s] Operation Failed\n", PType);  return -1;  }  return 0;  }  int Append(void \*AppendFirstAddr, char \*buffer, int pos) {  char \*tmp = (char \*)AppendFirstAddr;  if (AppendFirstAddr == NULL || buffer == NULL) {  printf("Parameter cann't be NULL[%d]\n", \_\_LINE\_\_);  return -1;  }  printf("---%d--[%d]---\n", pos, \_\_LINE\_\_);  sprintf(tmp + pos, "%s", buffer);  return 0;  }  修改后的muti\_consumer代码如下  /\* 文件名: muti\_consumer.c  \* 功能: 消费者  \* 模拟多个消费者时，只要将该文件编译后的可执行程序在多个终端分别执行即可  \*/  #include <errno.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/sem.h>  #include <sys/shm.h>  #include <sys/types.h>  #include <unistd.h>  #define SEMNUM 3  #define SHMSIZE 900  typedef union semun {  int val;  struct semid\_ds \*buf;  short int \*array;  } SEMUN;  int Buy(void \*outbuffer, void \*ConsumeFirstAddr, int pos, int num);  int V(const int semid, char \*PType, int len);  int P(const int semid, char \*PType, int len);  int Produce(char \*buf);  int GetShm(int key, int \*shmid, void \*\*shmaddr);  int GetSemCollection(int key, int \*semid);  int GetKey();  int main() {  int semid = 0;  int shmid = 0;  int key = 0;  int num\_to\_buy = 0; /\* Num Of Production \*/  int pos = 0;  int i = 0;  short int array[SEMNUM];  char buffer[SHMSIZE];  char \*shmaddr = NULL;  char \*ConsumeFirstAddr = NULL;  memset(buffer, 0x00, sizeof(buffer));  memset(array, 0x00, sizeof(array));  printf("Consumer's ID is[%d]\n", getpid());  if ((key = GetKey()) < 0) {  printf("Get Key Failed\n");  return -1;  }  printf("key[%d]\n", key);  if (GetSemCollection(key, &semid) < 0) {  printf("Get Sem Failed[%d]\n", errno);  return -1;  }  printf("semid[%d]\n", semid);  if (GetShm(key, &shmid, &shmaddr) < 0) {  printf("Get Shm Failed[%d]\n", errno);  return -1;  }  printf("shmid[%d] shmaddr[%d]\n", shmid, shmaddr);  memcpy(&pos, shmaddr + sizeof(int), sizeof(int));  ConsumeFirstAddr = shmaddr + 2 \* sizeof(int);  printf("=======pos[%d], ConsumeFirstAddr[%s]===\n", pos, ConsumeFirstAddr);  while (1) {  printf("\n\n");  memset(buffer, 0x00, sizeof(buffer));  if (semctl(semid, 0, GETALL, array) < 0) {  printf("Get Sem Val Failed\n");  return -1;  }  for (i = 0; i < SEMNUM; i++) {  if (i == 0) {  printf("empty=[%d]\n", array[i]);  } else if (i == 1) {  printf("full=[%d]\n", array[i]);  } else if (i == 2) {  printf("mutex=[%d]\n", array[i]);  }  }  printf("Please input the num of production you want to buy:");  scanf("%d", &num\_to\_buy);  if (num\_to\_buy < 0 || num\_to\_buy > 900) {  printf("The num input is wrong,it must between 0 and 100\ n");  continue;  }  /\* Get Empty Shared Memory \*/  printf("Consumer[%d] Now Requiring [%d] production to buy\n", getpid(),  num\_to\_buy);  if (P(semid, "full", num\_to\_buy) < 0) {  printf("P full Operation Failed\n");  return -1;  }  printf("Consumer[%d] can buy [%d] productions \n", num\_to\_buy);  printf("Consumer[%d] waiting for buying the productions\n", getpid());  /\* Mutex \*/  if (P(semid, "mutex", 1) < 0) {  printf("P mutex Operation Failed\n");  return -1;  }  printf("Now it's turn of the consumer[%d] to buy the productions\n",  getpid());  memcpy(&pos, shmaddr + sizeof(int), sizeof(int));  printf("----->pos[%d] num\_to\_buy[%d] shmaddr[%s]\n", pos, num\_to\_buy,  shmaddr + 2 \* sizeof(int));  if (Buy(buffer, ConsumeFirstAddr, pos, num\_to\_buy) < 0) {  printf("Buy Production Failed\n");  return -1;  }  printf("Consumer [%d] Buy Production[%s] shmaddr[%s]\n", getpid(),  buffer, shmaddr + 2 \* sizeof(int));  pos = (pos + num\_to\_buy) % SHMSIZE;  memcpy(shmaddr + sizeof(int), &pos, sizeof(int));  V(semid, "mutex", 1);  printf("Consumer [%d] leave the shop\n", getpid());  V(semid, "empty", num\_to\_buy);  printf("Now the shop has [%d] more position to put the productions\n",  num\_to\_buy);  }  return 0;  }  int GetKey() {  char filepath[128];  char cmd[128];  int key = 0;  memset(filepath, 0x00, sizeof(filepath));  memset(cmd, 0x00, sizeof(cmd));  sprintf(filepath, "%s/key.ini", getenv("HOME"));  /\*  sprintf(cmd, "touch %s", filepath);  if ( system(cmd) < 0 )  {  printf("Create Key File Failed\n");  return -1;  }  \*/  if ((key = ftok(filepath, 20)) < 0) {  printf("Create Key Failed\n");  return -1;  }  return key;  }  int GetSemCollection(int key, int \*semid) {  int lsemid = 0;  SEMUN unsem;  short int array[3];  if ((lsemid = semget(key, SEMNUM, IPC\_CREAT | 0666)) < 0) {  printf("Get Sem ID Failed\n");  return -1;  }  #if 0  array[0] = SHMSIZE; /\* Empty Buffer Size \*/  array[1] = 0; /\* Full Buffer Size \*/  array[2] = 1; /\* For Mutex \*/  unsem.array=array;  if ( semctl(lsemid, 0, SETALL, unsem) < 0 )  {  printf("Init Sem Failed\n");  return -1;  77  }  #endif  \*semid = lsemid;  return 0;  }  int GetShm(int key, int \*shmid, void \*\*shmaddr) {  int lshmid = 0;  char \*lshmaddr = NULL;  if ((lshmid = shmget(key, SHMSIZE + 2 \* sizeof(int), IPC\_CREAT | 0666)) <  0) {  printf("Get Shared Memory Failed\n");  return -1;  }  if ((lshmaddr = (char \*)shmat(lshmid, 0, 0)) == (char \*)-1) {  printf("Attach Shared Memory Failed, PID[%d]\n", getpid());  return -1;  }  \*shmid = lshmid;  \*shmaddr = lshmaddr;  return 0;  }  int Produce(char \*buf) {  if (buf == NULL) {  printf("Produce Buffer Couldn't Be NULL\n");  return -1;  }  printf("Please Produce Production:\n");  gets(buf);  printf("\nProducer PID[%d] Has Produce Something [%s]\n", buf);  return 0;  }  int P(const int semid, char \*PType, int len) {  struct sembuf stSembuf;  memset(&stSembuf, 0x00, sizeof(stSembuf));  if (PType == NULL) {  printf("PType Cann't Be NULL\n");  return -1;  }  if (strcmp(PType, "empty") == 0) {  stSembuf.sem\_num = 0;  } else if (strcmp(PType, "full") == 0) {  stSembuf.sem\_num = 1;  } else if (strcmp(PType, "mutex") == 0) {  stSembuf.sem\_num = 2;  } else {  printf("Sem Type Wrong\n");  return -1;  }  stSembuf.sem\_op = (-1) \* len;  stSembuf.sem\_flg = SEM\_UNDO;  if (semop(semid, &stSembuf, 1) < 0) {  printf("P [%s] Operation Failed\n", PType);  return -1;  }  return 0;  }  int V(const int semid, char \*PType, int len) {  struct sembuf stSembuf;  memset(&stSembuf, 0x00, sizeof(stSembuf));  if (PType == NULL) {  printf("PType Cann't Be NULL\n");  return -1;  }  if (strcmp(PType, "empty") == 0) {  stSembuf.sem\_num = 0;  } else if (strcmp(PType, "full") == 0) {  stSembuf.sem\_num = 1;  } else if (strcmp(PType, "mutex") == 0) {  stSembuf.sem\_num = 2;  } else {  printf("Sem Type Wrong\n");  return -1;  }  stSembuf.sem\_op = len;  stSembuf.sem\_flg = SEM\_UNDO;  if (semop(semid, &stSembuf, 1) < 0) {  printf("P [%s] Operation Failed\n", PType);  return -1;  }  return 0;  }  int Buy(void \*outbuffer, void \*ConsumeFirstAddr, int pos, int num) {  int i = 0;  char \*tmp = (char \*)ConsumeFirstAddr;  if (outbuffer == NULL) {  printf("Parameter 'outbuffer' cann't be NULL\n");  return -1;  }  memcpy(outbuffer, tmp + pos, num);  for (i = 0; i < num; i++) { tmp[pos + i] = '\*'; }  return 0;  }  修改方法:  (1)导入需要但未导入的库，<stdlib.h>和<string.h>  (2)将所有后方函数在main函数前声明  (3)将GetShm函数中的  (char \*)\*shmaddr=(char \*)lshmaddr;  更改为  \*shmaddr=lshmaddr;   1. 实验结果   在如图五个终端中，proc(1, 2, 3)为3个producer，con（1， 2）为2个consumer    生产者提示界面如图    消费者提示界面如图    先让1,2消费者都购买2个商品处于等待状态      再让1,2,3生产者分别生产a，b，cd        现在可以在消费者页面看到1,2两个消费者都成功购买商品      本实验完成了通过pv操作对生产者消费者模型的程序设计 | | | | | |
| **八、实验总结**  1.深入理解了P，V操作的作用和使用方法：  (1)用于高效同步协调进程  (2)生产者要在生产前调用对empty的p操作，检查是否可生产，生产完成后，调用对full的v操作，使产品数加一  (3)消费者先调用对full的p操作，检查是否可购买，再调用对empty的p操作，使得产品能继续生产  2.程序解析  1.GetKey():首先在home下生成key.ini文件，再以此文件作为参数生成唯一的key返回，多个进程共用这一个文件保证了key，semid，shmid的相同，并以此建立通信  2.GetSemCollection():根据key值获取信号量集合semid  3.GetShm():根据key值创建(获取)共享空间用于进程通信，返回共享空间及其首地址  4.produce():生产产品，保存至buf中  5.P()/V():根据传入信息生成sembuf，对semid进行操作，当sembuf中的sem\_op小于0时相当于P操作，大于0时为V操作。通过semop函数，传入semid，sembuf完成操作  6.Buy():提取购物信息，返回购买的物品及其首地址  7.Append():提取生产信息，返回生产的物品及其首地址。  8.main():  (1)生成buffer缓存区和array信号量空间，输出当前生产者(消费者)编号  (2)生成并输出key码，生成(读取)并输出共享区及其首地址  (3)从共享区读取之前的生产消费信息并定位到最新的位置  (4)进入循环生产(消费)  .1生产者:创建缓冲区用于存储产品->读取信号量后将其输出->生产产品->输出此次生产的信息->对empty执行P操作->对mutex执行P操作->添加产品到缓存区->对mutex执行V操作->对full执行V操作  .2消费者:创建缓冲区用于读取产品->读取信号量信息并输出->用户输入购买产品数量->输出购买信息->对full执行P操作->对互斥锁mutex执行P操作->从缓存区取出产品->对mutex执行V操作->对empty执行V操作 | | | | | |
| 1. **教师评阅意见** | | | | | |