|  |  |  |  |
| --- | --- | --- | --- |
| **《操作系统原理》实验报告** | | | |
| 实验名称 | Linux操作系统研究实例—进程间通信 | 实验序号 | 6 |
| 实验日期 | 2023/5/6 | 实验人 |  |
| 一、实验题目   * 阅读教材第18章（Linux案例），并在互联网上查阅相关资料，对照操作系统课程中所讲的原理（进程管理，存储管理，文件系统，设备管理），了解Linux操作系统实例 * 形成一份专题报告   + 可以是全面综述性报告   + 可以是侧重某一方面的报告（进程调度，进程间通信，存储管理，文件系统，安全） * 阅读教材第19章（可选，Windows案例） | | | |
| 二、相关原理与知识  进程间通信是指在不同进程之间传播或交换信息。IPC的方式通常有管道（包括无名管道和命名管道）、消息队列、信号量、共享内存、Socket等。  1.无名管道  无名管道是UNIX系统最古老的进程间通信形式。它的特点有：  1.它是半双工的，数据只能在一个方向上流动，具有固定的读写端。  2.它只能用于父子进程或兄弟进程之间。  3.它可以看作是一种特殊的文件，对于他的读写也可以使用read、write函数，但它不是普通的文件，不属于任何系统，只存在于内存之中。  原型：   |  | | --- | | #include <unistd.h>  int pipe(int fd[2]); *// 返回值：若成功返回0，失败返回-1；* |   当管道建立时，它会创建两个文件描述符，fd[0]为读而打开，fd[1]为写而打开，相对的，关闭管道只需关闭这两个文件描述符即可，即   |  | | --- | | close(fd[0]); *// 关闭读端*  close(fd[1]); *// 关闭写端* |   2.命名管道（FIFO）  命名管道，又称FIFO，是一种进程间通信形式，同样也是一种特殊的文件类型，其有如下特点：  1.FIFO可以在无关的进程之间交换数据，与无名管道只能用于父子进程或兄弟进程之间不同  2.FIFO有路径名与之关联，它以一种特殊设备文件形式存在于文件系统中。  原型   |  | | --- | | #include <stdio.h>  int mkfifo(const char\* pathname, mode\_t mode); *// 返回值：成功返回0， 出错返回-1* |   其中的mode参数与open函数中的mode相同，规定用户在FIFO中的权限。一旦创造了一个FIFO，就可以用一般的文件I/O函数操作它。  FIFO的通信方式类似于在进程中使用文件来传输数据，只不过FIFO类型文件同时具有管道的特性。在数据读出时，FIFO管道中同时清除数据，并且“先进先出”。  3.消息队列  消息队列，是一种进程间通信机制，用在不相关的进程之间传递消息。它提供了一种可靠的、异步的、基于消息的通信方式，可以用于进程之间的数据交换和协作。一个消息队列由一个标识符（队列ID）来标识，它有如下特点：  1.消息队列是面向记录的，其中的消息具有特定的格式和特定的优先级。  2.消息队列独立于发送和接收进程。进程终止时，消息队列及其内容并不会被删除。  3.消息队列可以实现消息的随机查询，消息不一定要以先进先出的次序读取，也可以按消息的类型读取。  原型   |  | | --- | | #include <stdio.h>    int msgget(key\_t key, int flag); *// 创建或打开消息队列：成功返回队列ID，失败返回-1*    int msgsnd(int msqid, const void ptr, size\_t size, int flag); *// 添加消息：成功返回0，失败返回-1*    int msgrcv(int msqid, void\* ptr, size\_t size, long type, int flag); *// 读取消息：成功返回消息数据的长度，失败返回-1*    int msgctl(int msqid, int cmd, struct msqid\_ds \* buf); *// 控制消息队列：成功返回0， 失败返回-1* |   msgget函数用于创建或打开一个消息队列。它接受两个参数，key是一个唯一的标识符用于标识消息队列，flag用于指定创建或打开消息队列的选项。如果成功，返回一个非负整数作为消息队列的标识符（队列ID），用于后续对该消息队列的操作；如果失败，返回-1。  msgsnd函数用于向指定的消息队列发送消息。它接受四个参数，msqid是消息队列的标识符（队列ID），ptr是指向消息内容的指针，size是消息的长度，flag用于指定发送消息的选项。如果成功，返回0；如果失败，返回-1。  msgrcv函数用于从指定的消息队列中接收消息。它接受五个参数，msqid是消息队列的标识符（队列ID），ptr是接收消息的缓冲区指针，size是缓冲区的大小，type是要接收的消息类型，flag用于指定接收消息的选项。如果成功，返回接收到的消息数据的长度；如果失败，返回-1。  msgctl函数用于对消息队列进行控制操作，例如删除消息队列或获取消息队列的状态信息。它接受三个参数，msqid是消息队列的标识符（队列ID），cmd是控制命令，用于指定要执行的操作，buf是一个指向结构体msqid\_ds的指针，用于传递或接收消息队列的信息。如果成功，返回0；如果失败，返回-1。  4.信号量  信号量是一种进程间通信机制。它允许进程在共享资源的访问和使用方面进行协调和同步。进程可以使用信号量来进行进程间的通信和协作，以保证共享资源的正确访问和使用。信号量与上述介绍的IPC结构不同，它不是文件，而是一个计数器。信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。它的特点如下：  1.信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。  2.信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。  3.每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。  4.支持信号量组。  它的原型如下：   |  | | --- | | #include <semaphore.h>    int semget(key\_t key, int num\_sems, int sem\_flags); *// 创建或获取一个信号量组：若成功返回信号量集ID，失败返回-1*    int semop(int semid, struct sembuf semoparray[], size\_t numops); *// 对信号量组进行操作，改变信号量的值：成功返回0，失败返回-1*    int semctl(int semid, int sem\_num, int cmd, ...); *// 控制信号量的相关信息* |   semget函数用于创建或获取一个信号量组。它接受三个参数，key是一个唯一的标识符用于标识信号量组，num\_sems是信号量的数量，一般为1，sem\_flags是创建或获取信号量组的选项。如果成功，返回一个非负整数作为信号量组的标识符（信号量集ID）；如果失败，返回-1。  semop函数用于对信号量组进行操作，改变信号量的值。它接受三个参数，semid是信号量组的标识符（信号量集ID），semoparray是一个指向sembuf结构体数组的指针，用于指定要执行的操作，numops是操作的数量。如果成功，返回0；如果失败，返回-1。  其中sembuf在库中已经写好，其定义为   |  | | --- | | struct sembuf  {  short sem\_num; *// 信号量组中对应的序号，0～sem\_nums-1*  short sem\_op; *// 信号量值在一次操作中的改变量*  short sem\_flg; *// IPC\_NOWAIT, SEM\_UNDO*  } |   semctl函数用于控制信号量的相关信息，例如获取或设置信号量的值、删除信号量等。它接受四个参数，semid是信号量组的标识符（信号量集ID），sem\_num是信号量的索引，用于指定要操作的具体信号量，cmd是控制命令，用于指定要执行的操作，...是可选参数，用于传递额外的参数。如果成功，返回操作相关的值；如果失败，返回-1。  5.共享内存  Linux中的共享内存是一种高效的进程间通信机制，用于在不相关的进程之间共享大块数据。共享内存允许多个进程访问同一块内存区域，从而实现数据的快速共享和交换。其有如下特点：  1.共享内存是最快的一种IPC，因为这种操作是进程直接对内存进行读取。  2.因为多个进程可以同时操作，所以需要进行同步  3.信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问。  它的原型如下：   |  | | --- | | #include <stdio.h>    int shmget(key\_t key, size\_t size, int flag); *// 创建或获取一个共享内存：成功返回共享内存ID，失败返回-1*    void shmat(int shm\_id, const void \*addr, int flag); *// 连接共享内存到当前进程的地址空间：成功返回指向共享内存的指针，失败返回-1*    int shmdt(void addr); *// 断开与共享内存的连接：成功返回0，失败返回-1*    int shmctl(int shm\_id, int cmd, struct shmid\_ds \*buf); *// 控制共享内存的相关信息：成功返回0，失败返回-1* |   shmget函数用于创建或获取一个共享内存区域。它接受三个参数，key是一个唯一的标识符用于标识共享内存，size是共享内存的大小，flag用于指定创建或获取共享内存的选项。如果成功，返回一个非负整数作为共享内存的标识符（共享内存ID）；如果失败，返回-1。  shmat函数用于将共享内存连接到当前进程的地址空间。它接受三个参数，shm\_id是共享内存的标识符（共享内存ID），addr是指向指定连接地址的指针，flag用于指定连接共享内存的选项。如果成功，返回指向共享内存的指针；如果失败，返回-1。  shmdt函数用于断开与共享内存的连接。它接受一个参数addr，即要断开连接的共享内存的指针。如果成功，返回0；如果失败，返回-1。  shmctl函数用于控制共享内存的相关信息，例如获取或设置共享内存的状态和属性。它接受三个参数，shm\_id是共享内存的标识符（共享内存ID），cmd是控制命令，用于指定要执行的操作，buf是一个指向shmid\_ds结构体的指针，用于传递或接收共享内存的信息。如果成功，返回0；如果失败，返回-1。  6.Socket  Socket是一种用于网络通信的编程接口，用于实现不同计算机之间的数据交换。Socket可以用于不同进程之间的通信，但它的主要目的是实现网络通信而不是进程间通信，因此此处不进行详细介绍。 | | | |
| 三、实验过程与结果  本次实验我选择针对无名管道、命名管道、消息队列、信号量和共享内存进行验证，其中无名管道、命名管道、消息队列分别进行使用代码实现并验证结果；共享内存由于未提供有效的同步机制，因此我们选择使用信号量来进行共享内存进程间同步，因此将信号量和共享内存联合起来进行实验：共4个验证实验。  1.无名管道  本实验创建子进程，在父进程中向无名管道中写入内容，并在子进程中输出。  具体而言，首先创建管道：   |  | | --- | | int fd[2]; // 两个文件描述符  pid\_t pid;  char buff[20];  if (pipe(fd) < 0)  {  printf("Create Pipe Error!\n");  } |   之后创建子进程，父进程对管道写入文本内容，子进程从管道中读取文本内容并输出：   |  | | --- | | if ((pid = fork()) < 0) // 创建子进程  {  printf("Fork Error!\n");  }  else if (pid > 0) // 父进程  {  printf("Parent Process Begin!\n");  close(fd[0]); // 关闭读端  write(fd[1], "This is the information that is piped\n", 100);  printf("Parent Process End!\n");  }  else  {  printf("Child Process Begin!\n");  close(fd[1]); // 关闭写端  read(fd[0], buff, 100);  printf ("%s", buff);  printf("Child Process End!\n");  } |   管道的输出结果为：    可以看到，我们在父进程中传输的信息，在子进程中成功被输出，验证了使用无名管道进行进程间通信的正确性。  2.命名管道  命名管道可以在无关进程之间传递信息，因此我们在系统中开启两个终端，在这两个终端中使用命名通道进行连接，并进行信息的传递。  本实验的验证思路为，在一个终端中运行“读”程序作为一个进程，另一个运行“写”程序作为一个进程，写进程通过命名管道向读进程传递当前的时间信息，总计10次。  具体而言，对于读程序，首先创建管道，并以只读方式打开管道：   |  | | --- | | if (mkfifo("fifo", 0666) < 0) // 创建FIFO管道  {  perror ("Create FIFO Failed\n");  }  if ((fd = open("fifo", O\_RDONLY)) < 0) // 以读方式打开  {  perror ("Open FIFO Failed\n");  exit(1);  } |   之后便对管道不断进行读取，等待内容的写入：   |  | | --- | | while (1) // 读取FIFO管道  {  len = read(fd, buf, 1024);  if(len > 0)  printf ("Read Message: %s", buf);  } |   而对于写程序，首先打开命名管道：   |  | | --- | | if((fd = open("fifo", O\_WRONLY)) < 0) // 以写打开一个FIFO  {  perror("Open FIFO Failed");  exit(1);  } |   之后向管道中写入内容，本实验是写入系统当前时间，每隔1s写入一次，10次为止。   |  | | --- | | for(i=0; i<10; ++i)  {  time(&tp); // 取系统当前时间  n=sprintf(buf,"Process %d's time is %s",getpid(),ctime(&tp));  printf("Send message: %s", buf); // 打印  if(write(fd, buf, n+1) < 0) // 写入到FIFO中  {  perror("Write FIFO Failed");  close(fd);  exit(1);  }  sleep(1); // 休眠1秒  } |   实验结果如下：    其中左边为写入进程，右边为读取进程，可以看到写进程向命名管道中写入的内容全部被读进程读取并输出，这验证了使用命名管道进行进程间通信的正确性。  同时，我也注意到，在运行完上述两个程序以后，在文件夹内出现了管道的文件，且无法打开，这也印证了命名管道是特殊的文件类型。    3.消息队列  针对消息队列的验证，我们使用了server-client程序，即server作为服务器不断接收client传递的信息，而接收信息以后server也向client传递一条信息，表示已经接收到。  同样的，针对消息队列的验证，我们也在两个终端中运行了两个无关的进程，即server进程和client进程，对消息队列的进程间通信效果进行验证。  具体而言，对server程序而言，要想创建消息队列，需要一个唯一的key，因此对key的存储路径做出定义，并获取key值：   |  | | --- | | #define MSG\_FILE "/etc/passwd"  if((key = ftok(MSG\_FILE,'z')) < 0)  {  perror("ftok error");  exit(1);  } |   之后通过key打开消息队列：   |  | | --- | | if ((msqid = msgget(key, IPC\_CREAT|0777)) == -1)  {  perror("msgget error");  exit(1);  } |   最后server进程将循环读取来自client端（类型为888）的信息，并返回信息（类型为999）：   |  | | --- | | for(;;)  {  msgrcv(msqid, &msg, 256, 888, 0);// 返回类型为888的第一个消息  printf("Server: receive msg.mtext is: %s.\n", msg.mtext);  printf("Server: receive msg.mtype is: %d.\n", msg.mtype);  msg.mtype = 999; // 客户端接收的消息类型  sprintf(msg.mtext, "hello, I'm server %d", getpid());  msgsnd(msqid, &msg, sizeof(msg.mtext), 0);  } |   client进程读取key和消息队列的代码与server端一样，此处不做重复说明，不同的是client端在最后会向server端发送消息（类型888）并接收返回的消息（类型999），代码如下：   |  | | --- | | // 添加消息，类型为888  msg.mtype = 888;  sprintf(msg.mtext, "hello, I'm client %d", getpid());  msgsnd(msqid, &msg, sizeof(msg.mtext), 0);  // 读取类型为999的消息  msgrcv(msqid, &msg, 256, 999, 0);  printf("Client: receive msg.mtext is: %s.\n", msg.mtext);  printf("Client: receive msg.mtype is: %d.\n", msg.mtype); |   实验结果如下：    此处我们运行了三次client程序，相当于有三个客户对server进行访问，他们均被正确的输出，这验证了使用消息队列进行进程间通信的正确性。  4.信号量与共享内存  针对信号量和共享内存在进程间通信的作用，由于共享内存中没有进程同步机制，因此我们使用信号量来实现；针对他们两个的作用我们使用一个实验进行验证。  同样的，我们此处使用write-read程序，在两个终端上创建两个不同的进程，来实现write和read，对信号量和共享内存的功能进行验证。  此处我们统一介绍write和read进程中的信号量和共享内存的使用，其中的操作部分分开说。  对于信号量，由于是跨进程，因此需要统一且唯一的key，来保证进程间信号量是同一个信号量，进而保证进程同步，此处给出信号量的定义和创建：   |  | | --- | | #define SHM\_KEY 12345  #define SHM\_SIZE 1024  // P 操作（等待信号量）  void semWait(int semid) {  struct sembuf operation;  operation.sem\_num = 0; // 使用信号量集中的第一个信号量  operation.sem\_op = -1; // 等待信号量  operation.sem\_flg = 0; // 无特殊标志  semop(semid, &operation, 1);  }  // V 操作（释放信号量）  void semSignal(int semid) {  struct sembuf operation;  operation.sem\_num = 0; // 使用信号量集中的第一个信号量  operation.sem\_op = 1; // 释放信号量  operation.sem\_flg = 0; // 无特殊标志  semop(semid, &operation, 1);  }  // 创建信号量  semid = semget(SHM\_KEY, 1, IPC\_CREAT | 0666);  if (semid == -1) {  perror("semget");  exit(1);  }    // 初始化信号量  semctl(semid, 0, SETVAL, 1); |   上述即程序中信号量需要使用到的一些基本定义与操作，不包括信号量的进程同步作用（后面说）。  其次是进程间共享内存的使用，他们的代码也是一样的，即获取共享内存并进行连接：   |  | | --- | | shmid = shmget(SHM\_KEY, sizeof(SharedData), IPC\_CREAT | 0666);  if (shmid == -1) {  perror("shmget");  exit(1);  }    // 连接共享内存  sharedData = (SharedData\*)shmat(shmid, NULL, 0);  if (sharedData == (void\*)-1) {  perror("shmat");  exit(1);  } |   对于write进程，我们等待信号量以后，向其中写入内容：   |  | | --- | | // 在共享内存中进行读写操作  semWait(semid);    printf("Enter a message: ");  fgets(sharedData->message, sizeof(sharedData->message), stdin);  sharedData->counter = strlen(sharedData->message);    semSignal(semid); |   对于read进程，我们等待信号量以后，从其中读取内容：   |  | | --- | | // 读取共享内存中的数据  semWait(semid);  printf("Message: %s", sharedData->message);  printf("Counter: %d\n", sharedData->counter);  semSignal(semid); |   实验结果如下：    可以看到，我们通过共享内存在进程间传输的信息，并使用信号量保证了进程的同步，不会冲突地访问共享内存，这验证了使用信号量和共享内存进行进程间通信的正确性。 | | | |
| 四、源代码   |  | | --- | | 无名管道代码  #include <stdio.h>  #include <stdlib.h>  int main()  {  int fd[2]; // 两个文件描述符  pid\_t pid;  char buff[20];  if (pipe(fd) < 0)  {  printf("Create Pipe Error!\n");  }  if ((pid = fork()) < 0) // 创建子进程  {  printf("Fork Error!\n");  }  else if (pid > 0) // 父进程  {  printf("Parent Process Begin!\n");  close(fd[0]); // 关闭读端  write(fd[1], "This is the information that is piped\n", 100);  printf("Parent Process End!\n");  }  else  {  printf("Child Process Begin!\n");  close(fd[1]); // 关闭写端  read(fd[0], buff, 100);  printf ("%s", buff);  printf("Child Process End!\n");  }  return 0; |  |  | | --- | | 命名管道代码  read.c  #include <stdio.h>  #include<stdlib.h> // exit  #include<fcntl.h> // O\_WRONLY  #include<sys/stat.h>  #include<time.h> // time  #include<errno.h>  int main()  {  int fd;  int len;  char buf[1024];    if (mkfifo("fifo", 0666) < 0) // 创建FIFO管道  {  perror ("Create FIFO Failed\n");  }  if ((fd = open("fifo", O\_RDONLY)) < 0) // 以读方式打开  {  perror ("Open FIFO Failed\n");  exit(1);  }    while (1) // 读取FIFO管道  {  len = read(fd, buf, 1024);  if(len > 0)  printf ("Read Message: %s", buf);  }    close(fd);  return 0;  }  write.c  #include<stdio.h>  #include<stdlib.h> // exit  #include<fcntl.h> // O\_WRONLY  #include<sys/stat.h>  #include<time.h> // time  int main()  {  int fd;  int n, i;  char buf[1024];  time\_t tp;  printf("I am %d process.\n", getpid()); // 说明进程ID  if((fd = open("fifo", O\_WRONLY)) < 0) // 以写打开一个FIFO  {  perror("Open FIFO Failed");  exit(1);  }  for(i=0; i<10; ++i)  {  time(&tp); // 取系统当前时间  n=sprintf(buf,"Process %d's time is %s",getpid(),ctime(&tp));  printf("Send message: %s", buf); // 打印  if(write(fd, buf, n+1) < 0) // 写入到FIFO中  {  perror("Write FIFO Failed");  close(fd);  exit(1);  }  sleep(1); // 休眠1秒  }  close(fd); // 关闭FIFO文件  return 0;  } |  |  | | --- | | 消息队列代码  server.c  #include<stdio.h>  #include<stdlib.h>  #include<sys/types.h>  #include<time.h>  #include<sys/ipc.h>  // 用于创建一个唯一的key  #define MSG\_FILE "/etc/passwd"  // 消息结构  struct msg\_form {  long mtype;  char mtext[256];  };  int main()  {  int msqid;  key\_t key;  struct msg\_form msg;    // 获取key值  if((key = ftok(MSG\_FILE,'z')) < 0)  {  perror("ftok error");  exit(1);  }  // 打印key值  printf("Message Queue - Server key is: %d.\n", key);  // 创建消息队列  if ((msqid = msgget(key, IPC\_CREAT|0777)) == -1)  {  perror("msgget error");  exit(1);  }    // 打印消息队列ID及进程ID  printf("My msqid is: %d.\n", msqid);  printf("My pid is: %d.\n", getpid());  // 循环读取消息  for(;;)  {  msgrcv(msqid, &msg, 256, 888, 0);// 返回类型为888的第一个消息  printf("Server: receive msg.mtext is: %s.\n", msg.mtext);  printf("Server: receive msg.mtype is: %d.\n", msg.mtype);  msg.mtype = 999; // 客户端接收的消息类型  sprintf(msg.mtext, "hello, I'm server %d", getpid());  msgsnd(msqid, &msg, sizeof(msg.mtext), 0);  }  return 0;  }  client.c  #include<stdio.h>  #include<stdlib.h>  #include<sys/types.h>  #include<time.h>  #include<sys/ipc.h>  // 用于创建一个唯一的key  #define MSG\_FILE "/etc/passwd"  // 消息结构  struct msg\_form {  long mtype;  char mtext[256];  };  int main()  {  int msqid;  key\_t key;  struct msg\_form msg;  // 获取key值  if ((key = ftok(MSG\_FILE, 'z')) < 0)  {  perror("ftok error");  exit(1);  }  // 打印key值  printf("Message Queue - Client key is: %d.\n", key);  // 打开消息队列  if ((msqid = msgget(key, IPC\_CREAT|0777)) == -1)  {  perror("msgget error");  exit(1);  }  // 打印消息队列ID及进程ID  printf("My msqid is: %d.\n", msqid);  printf("My pid is: %d.\n", getpid());  // 添加消息，类型为888  msg.mtype = 888;  sprintf(msg.mtext, "hello, I'm client %d", getpid());  msgsnd(msqid, &msg, sizeof(msg.mtext), 0);  // 读取类型为999的消息  msgrcv(msqid, &msg, 256, 999, 0);  printf("Client: receive msg.mtext is: %s.\n", msg.mtext);  printf("Client: receive msg.mtype is: %d.\n", msg.mtype);  return 0;  } |  |  | | --- | | 信号量和共享内存代码  read.c  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <sys/sem.h>  #include <sys/types.h>  #define SHM\_KEY 12345  #define SHM\_SIZE 1024  typedef struct {  int counter;  char message[100];  } SharedData;  // P 操作（等待信号量）  void semWait(int semid) {  struct sembuf operation;  operation.sem\_num = 0; // 使用信号量集中的第一个信号量  operation.sem\_op = -1; // 等待信号量  operation.sem\_flg = 0; // 无特殊标志  semop(semid, &operation, 1);  }  // V 操作（释放信号量）  void semSignal(int semid) {  struct sembuf operation;  operation.sem\_num = 0; // 使用信号量集中的第一个信号量  operation.sem\_op = 1; // 释放信号量  operation.sem\_flg = 0; // 无特殊标志  semop(semid, &operation, 1);  }  int main() {  int shmid, semid;  SharedData \*sharedData;  // 获取共享内存  shmid = shmget(SHM\_KEY, sizeof(SharedData), 0);  if (shmid == -1) {  perror("shmget");  exit(1);  }  // 连接共享内存  sharedData = (SharedData\*)shmat(shmid, NULL, 0);  if (sharedData == (void\*)-1) {  perror("shmat");  exit(1);  }  // 获取信号量  semid = semget(SHM\_KEY, 1, 0);  if (semid == -1) {  perror("semget");  exit(1);  }  // 读取共享内存中的数据  semWait(semid);  printf("Message: %s", sharedData->message);  printf("Counter: %d\n", sharedData->counter);  semSignal(semid);  // 分离共享内存  if (shmdt(sharedData) == -1) {  perror("shmdt");  exit(1);  }  return 0;  }  write.c  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <sys/sem.h>  #include <sys/types.h>  #include <string.h>  #define SHM\_KEY 12345  #define SHM\_SIZE 1024  typedef struct {  int counter;  char message[100];  } SharedData;  // P 操作（等待信号量）  void semWait(int semid) {  struct sembuf operation;  operation.sem\_num = 0; // 使用信号量集中的第一个信号量  operation.sem\_op = -1; // 等待信号量  operation.sem\_flg = 0; // 无特殊标志  semop(semid, &operation, 1);  }  // V 操作（释放信号量）  void semSignal(int semid) {  struct sembuf operation;  operation.sem\_num = 0; // 使用信号量集中的第一个信号量  operation.sem\_op = 1; // 释放信号量  operation.sem\_flg = 0; // 无特殊标志  semop(semid, &operation, 1);  }  int main() {  int shmid, semid;  SharedData \*sharedData;    // 创建共享内存  shmid = shmget(SHM\_KEY, sizeof(SharedData), IPC\_CREAT | 0666);  if (shmid == -1) {  perror("shmget");  exit(1);  }    // 连接共享内存  sharedData = (SharedData\*)shmat(shmid, NULL, 0);  if (sharedData == (void\*)-1) {  perror("shmat");  exit(1);  }    // 创建信号量  semid = semget(SHM\_KEY, 1, IPC\_CREAT | 0666);  if (semid == -1) {  perror("semget");  exit(1);  }    // 初始化信号量  semctl(semid, 0, SETVAL, 1);    // 在共享内存中进行读写操作  semWait(semid);    printf("Enter a message: ");  fgets(sharedData->message, sizeof(sharedData->message), stdin);  sharedData->counter = strlen(sharedData->message);    semSignal(semid);    // 分离共享内存  if (shmdt(sharedData) == -1) {  perror("shmdt");  exit(1);  }    return 0;  } | | | | |