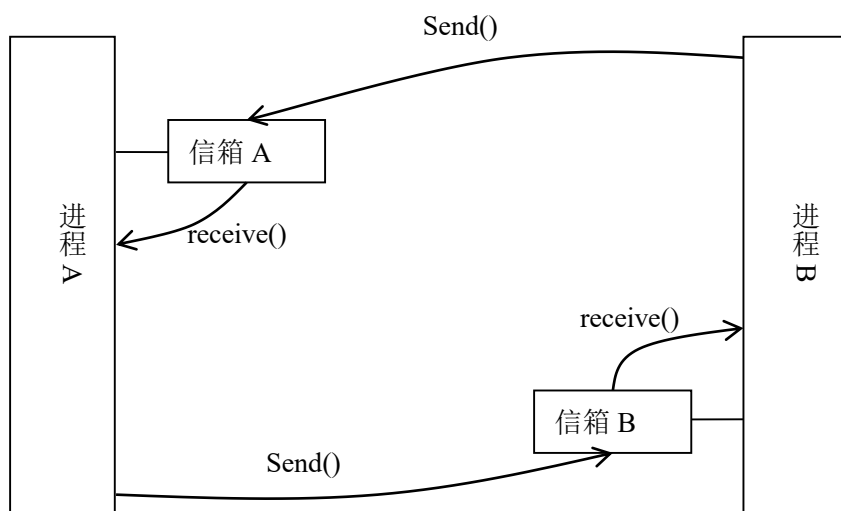


## 进程通信（实验 2）

【实验名称】：进程通信（实验 2）

【实验目的】：掌握用邮箱方式进行进程通信的方法，并通过设计实现简单邮箱理解进程通信中的同步问题以及解决该方法。

【实验原理】：邮箱机制类似于日常使用的信箱。对于用户而言使用起来比较方便，用户只需使用 `send()` 向对方邮箱发邮件 `receive()` 从自己邮箱取邮件，`send()` 和 `receive()` 的内部操作用户无需关心。因为邮箱在内存中实现，其空间有大小限制。其实 `send()` 和 `receive()` 的内部实现主要还是要解决生产者与消费者问题。



【实验内容】：进程通信的邮箱方式由操作系统提供形如 `send()` 和 `receive()` 的系统调用来支持，本实验要求学生首先查找资料了解所选用操作系统平台上用于进程通信的系统调用具体形式，然后使用该系统调用编写程序进行进程间的通信，要求程序运行结果可以直观地体现在界面上。在此基础上查找所选用操作系统平台上支持信号量机制的系统调用具体形式，运用生产者与消费者模型设计实现一个简单的信箱，该信箱需要有创建、发信、收信、撤销等函数，至少能够支持两个进程互相交换信息，比较自己实现的信箱与操作系统本身提供的信箱，分析两者之间存在的异同。

【实验要求】：每个同学必须独立完成本实验、提交实验报告、源程序和可执行程序。实验报告中必须包含背景知识介绍，设计方案，预计的实验结果，关键代码的分析，调试记录，实际的实验结果，实验结果分析等内容。

## (一) 背景知识介绍

### 1) 什么是消息队列

消息队列提供了一种从一个进程向另一个进程发送一个数据块的方法。每个数据块都被认为含有一个类型，接收进程可以独立地接收含有不同类型的数据结构。我们可以通过发送消息来避免命名管道的同步和阻塞问题。但是消息队列与命名管道一样，每个数据块都有一个最大长度的限制。

### 2) 在 Linux 中使用消息队列

Linux 提供了一系列消息队列的函数接口来让我们方便地使用它来实现进程间的通信。它的用法与其他两个 System V PIC 机制，即信号量和共享内存相似。

#### i. msgget()函数

该函数用来创建和访问一个消息队列。它的原型为：

```
int msgget(key_t, key, int msgflg);
```

返回一个以 key 命名的消息队列的标识符（非零整数），失败时返回 -1。

#### ii. msgsnd()函数

该函数用来把消息添加到消息队列中。它的原型为：

```
int msgsend(int msgid, const void *msg_ptr, size_t  
msg_sz, int msgflg);
```

如果调用成功，消息数据的一份副本将被放到消息队列中，并返回 0，失败时返回 -1。

### iii. msgrcv()函数

该函数用来从一个消息队列获取消息，它的原型为：

```
int msgrcv(int msgid, void *msg_ptr, size_t msg_st,  
long int msgtype, int msgflg);
```

调用成功时，该函数返回放到接收缓存区中的字节数，消息被复制到由 msg\_ptr 指向的用户分配的缓存区中，然后删除消息队列中的对应消息。失败时返回-1.

### iv. msgctl()函数

该函数用来控制消息队列，它与共享内存的 shmctl 函数相似，它的原型为：

```
int msgctl(int msgid, int command, struct msgid_ds  
*buf);
```

成功时返回 0，失败时返回-1.

## 3) 什么是信号量

为了防止出现因多个程序同时访问一个共享资源而引发的一系列问题，我们需要一种方法，它可以通过生成并使用令牌来授权，在任一时刻只能有一个执行线程访问代码的临界区域。临界区域是指执行数据更新的代码需要独占式地执行。而信号量就可以提供这样的一种访问机制，让一个临界区同一时间只有一个线程在访问它，也就是说信号量是用来调协进程对共享资源的访问的。

#### 4) 在 Linux 中使用信号量

Linux 提供了一组精心设计的信号量接口来对信号进行操作，它们不只是针对二进制信号量，下面将会对这些函数进行介绍，但请注意，这些函数都是用来对成组的信号量值进行操作的。它们声明在头文件 `sys/sem.h` 中。

##### 1. `semget()`函数

创建一个新信号量或取得一个已有信号量，原型为：

```
int semget(key_t key, int num_sems, int sem_flags);
```

成功返回一个相应信号标识符（非零），失败返回-1。

##### 2. `semop()`函数

改变信号量的值，原型为：

```
int semop(int sem_id, struct sembuf *sem_opa, size_t  
num_sem_ops);
```

失败返回-1。

##### 3. `semctl()`函数

该函数用来直接控制信号量信息，它的原型为：

```
int semctl(int sem_id, int sem_num, int command, ...);
```

#### 5) 什么是共享内存

顾名思义，共享内存就是允许两个不相关的进程访问同一个逻辑内存。共享内存是在两个正在运行的进程之间共享和传递数据的一种非常有效的方式。不同进程之间共享的内存通常安排为同一段物理内存。进程可以将同一段共享内存连接到它们自

己的地址空间中，所有进程都可以访问共享内存中的地址，就好像它们是由用 C 语言函数 `malloc()` 分配的内存一样。而如果某个进程向共享内存写入数据，所做的改动将立即影响到可以访问同一段共享内存的任何其他进程。

特别提醒：共享内存并未提供同步机制，也就是说，在第一个进程结束对共享内存的写操作之前，并无自动机制可以阻止第二个进程开始对它进行读取。所以我们通常需要用其他的机制来同步对共享内存的访问，例如前面说到的信号量。

## 6) 在 Linux 中使用共享内存

### a) `shmget()` 函数

用来创建共享内存，它的原型为：

```
int shmget(key_t key, size_t size, int shmflg);
```

成功时返回一个与 `key` 相关的共享内存标识符（非负整数），用于后续的共享内存函数。调用失败返回 -1。

### b) `shmat()` 函数

第一次创建完共享内存时，它还不能被任何进程访问，

`shmat()` 函数的作用就是用来启动对该共享内存的访问，并

把共享内存连接到当前进程的地址空间。它的原型如下：

```
void *shmat(int shm_id, const void *shm_addr, int shmflg);
```

调用成功时返回一个指向共享内存第一个字节的指针，如

果调用失败返回 -1。

### c) shmdt()函数

该函数用于将共享内存从当前进程中分离。注意，将共享内存分离并不是删除它，只是使该共享内存对当前进程不再可用。它的原型如下：

```
int shmdt(const void *shmaddr);
```

调用成功时返回 0，失败时返回 -1.

### d) shmctl()函数

与信号量的 semctl()函数一样，用来控制共享内存，它的原型如下：

```
int shmctl(int shm_id, int command, struct shmid_ds *buf);
```

## (二) 设计方案

对于实验内容中“首先查找资料了解所选用操作系统平台上用于进程通信的系统调用具体形式，然后使用该系统调用编写程序进行进程间的通信，要求程序运行结果可以直观地体现在界面上。”

查阅资料发现 Ubuntu 通过消息队列来实现进程间的通信。使用系统提供的三个函数创建两个消息队列，实现进程通信。

对于实验内容中另一部分“在此基础上查找所选用操作系统平台上支持信号量机制的系统调用具体形式，运用生产者与消费者模型设计实现一个简单的信箱，该信箱需要有创建、发信、收信、撤销等函数，至少能够支持两个进程互相交换信息。”

通过共享内存建立一个邮箱，并使用信号量机制实现控制，使其发送和接收邮件时单进程占用内存，而不受破坏，当不满足临界条件时，该进程执行发送或接收或撤销操作会被挂起，直至另外一个进程执行发送或者接收或撤销操作。

## (三) 预计的实验结果

一个进程接收到的消息与另一个进程发送的消息完全相同，并且满足生产者与消费者模型。当不满足临界条件时，该进程执行发送或接收或撤销操作会被挂起，直至另外一个进程执行发送或者接收或撤销操作。

## (四) 关键代码的分析

### (1) 消息队列实现进程通信

```

int main()
{
    struct msg data;
    int msgid1 = msgget((key_t)KEY, 0666|IPC_CREAT); //创建消息队列1
    if(msgid1 < 0)
    {
        printf("msg1get failed.\n");
        exit(EXIT_FAILURE);
    }

    int msgid2= msgget((key_t)(KEY+1), 0666|IPC_CREAT); //创建消息队列2
    if(msgid2 < 0)
    {
        printf("msg2get failed.\n");
        exit(EXIT_FAILURE);
    }

    int op;
    while(1)
    {
        printf("input op:0 send, 1 receive, 2 exit:");
        scanf("%d", &op);

        if(op == 0)
        {
            printf("input msg:");
            scanf("%s",data.message);
            msgsnd(msgid1, &data,MESSAGEMAXSIZE,0); //向消息队列1发送消息
        }
        else if(op == 1)
        {
            msgrcv(msgid2, &data,MESSAGEMAXSIZE,RECEIVETYPE,0);
            printf("msg: %s\n",data.message); //从消息队列2接收消息
        }
        else if(op == 2)
        {
            break;
        }
        else
        {
            printf("invalid op! please input again!\n");
        }
    }
    msgctl(msgid1, IPC_RMID, 0);
    msgctl(msgid2, IPC_RMID, 0);
}

```

- (2) 共享内存与信号量实现进程通信  
头文件 sem.h

```

int getNewSem(key_t key) //获取新信号量
{
    return semget(key,1,0666|IPC_CREAT);
}

union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};

int setSemValue(int semid,int n) //设置信号量
{
    union semun semTemp;
    semTemp.val = n;
    if(semctl(semid,0,SETVAL,semTemp)==-1)
    {
        return 0;
    }
    return 1;
}

```

```

int delSem(int semid)//删除信号量
{
    union semun semTemp;
    semctl(semid,0,IPC_RMID,semTemp);
}
/*
struct sembuf
{
    unsigned short int sem_num;    //在信号量集中的序号
    short int sem_op;              // 对信号量的操作, >0, 0, <0
    short int sem_flg;            // 操作标识: 0, IPC_WAIT, SEM_UNDO
};
*/
int P(int semid)//P操作
{
    struct sembuf semTemp;
    semTemp.sem_num = 0;
    semTemp.sem_op = -1;
    semTemp.sem_flg = SEM_UNDO;
    if(semop(semid,&semTemp,1) == -1)
    {
        printf("P() failed\n");
        return 0;
    }
    return 1;
}

int V(int semid)//V操作
{
    struct sembuf semTemp;
    semTemp.sem_num = 0;
    semTemp.sem_op = 1;
    semTemp.sem_flg = SEM_UNDO;
    if(semop(semid,&semTemp,1) == -1)
    {
        printf("V() failed\n");
        return 0;
    }
    return 1;
}

头文件 box.h
struct box
{
    //信箱头
    int bid;//信箱标识符
    int bsize;//信格总数
    /*同步信号量*/
    int mailnum;//与信箱中信件数量相关的信号量
    int freenum;//与信箱中空信格数量相关的信号量
    /*互斥信号量*/
    int rmutex;//接收信件时的互斥信号量
    int wmutex;//存入信件时的互斥信号量
    int out;//当前可读取信件的信格地址
    int in;//当前可存入信件的信格地址
    //信箱体
    int *buf;
    void *shm;
};

```



```

struct box* getNewBox(int len,int n)//获取新信箱,len为信格总数, n为唯一邮箱变量
{
    int shmid;
    struct box* msgbox = (struct box*)malloc(sizeof(struct box));
    shmid = shmget((key_t)(boxId+n),sizeof(int)*len,0666|IPC_CREAT);
    msgbox->shm = shmat(shmid,0,0);
    msgbox->buf = (int *)(msgbox->shm);
    //初始化信格总数
    msgbox->bsize = len;
    //初始化邮箱标号,为共享信号的标号
    msgbox->bid = shmid;
    //初始化信号量
    msgbox->mailnum = getNewSem((key_t)(shmid+1));
    setSemValue(msgbox->mailnum,0);

    msgbox->freenum = getNewSem((key_t)(shmid+2));
    setSemValue(msgbox->freenum,len);

    msgbox->rmutex = getNewSem((key_t)(shmid+3));
    setSemValue(msgbox->rmutex,1);

    msgbox->wmutex = getNewSem((key_t)(shmid+4));
    setSemValue(msgbox->wmutex,1);

    msgbox->out = 0;
    msgbox->in = 0;

    return msgbox;
}

void send(struct box* dest,int msg)//发送
{
    P(dest->freenum);
    P(dest->wmutex);
    dest->buf[dest->in] = msg;
    dest->in = (dest->in+1)%dest->bsize;
    V(dest->wmutex);
    V(dest->mailnum);
};

int receive(struct box* addr)//接收
{
    int msg;
    P(addr->mailnum);
    P(addr->rmutex);
    msg = addr->buf[addr->out];
    addr->out = (addr->out+1)%addr->bsize;
    V(addr->rmutex);
    V(addr->freenum);

    return msg;
};

void recall(struct box* addr)//撤销
{
    P(addr->mailnum);
    P(addr->wmutex);
    P(addr->rmutex);
    addr->in = (addr->in-1)%addr->bsize;
    V(addr->rmutex);
    V(addr->wmutex);
    V(addr->freenum);
};

```

```
void deleteBox(struct box* msgbox)//删除信箱
```

```
{
    delSem(msgbox->freenum);
    delSem(msgbox->mailnum);
    delSem(msgbox->rmutex);
    delSem(msgbox->wmutex);
    shmdt(msgbox->shm);
    shmctl(msgbox->bid,IPC_RMID,0);
};
```

进程

```
int main()
{
```

```
    while(1)
    {
```

```
        int op;
        printf("input op:0 send, 1 receive, 2 recall, 3 exit:");
        scanf("%d",&op);
        if(op == 0)
```

```
        {
            printf("input msg:");
            int msg;
            scanf("%d",&msg);
            send(boxB,msg);
        }
```

```
    else if(op == 1)
    {
        printf("msg:%d\n",receive(boxA));
    }
```

```
    else if(op == 2)
    {
        recall(boxB);
    }
```

```
    else if(op == 3)
    {
        deleteBox(boxA);
        break;
    }
```

```
    else
    {
        printf("invalid op! please input again!\n");
    }
}
```

```
    return 0;
}
```

## (五) 调试记录

打开两个终端，分别运行 p1.exe 和 p2.exe

```
osstudy@osstudy-VirtualBox: ~/ex2
osstudy@osstudy-VirtualBox:~$ cd ex2
osstudy@osstudy-VirtualBox:~/ex2$ ls
b1.c  b1.exe  b2.c  b2.exe  box.h  p1.cpp  p1.exe  p2.cpp  p2.exe  sem.h
osstudy@osstudy-VirtualBox:~/ex2$ ./p1.exe
input op:0 send, 1 receive, 2 recall, 3 exit:

osstudy@osstudy-VirtualBox: ~/ex2
osstudy@osstudy-VirtualBox:~$ cd ex2
osstudy@osstudy-VirtualBox:~/ex2$ ls
b1.c  b1.exe  b2.c  b2.exe  box.h  p1.cpp  p1.exe  p2.cpp  p2.exe  sem.h
osstudy@osstudy-VirtualBox:~/ex2$ ./p2.exe
input op:0 send, 1 receive, 2 recall, 3 exit:
```

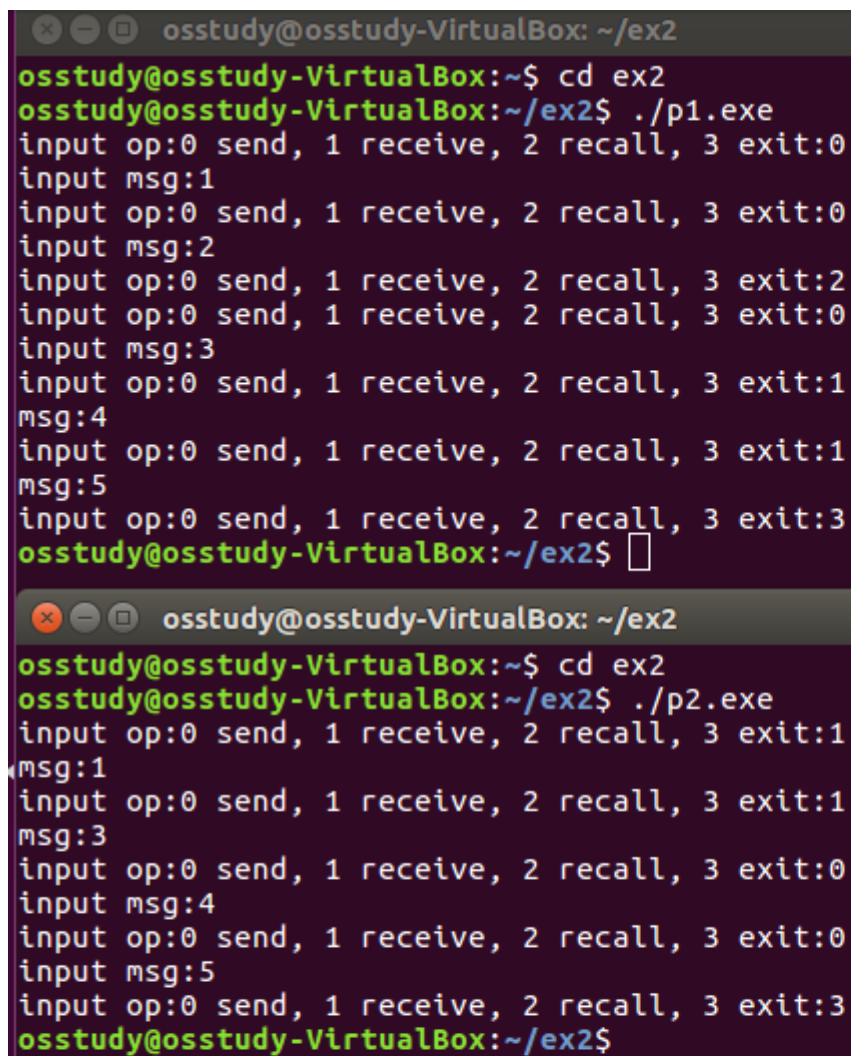
## (六) 实际的实验结果

消息队列实现进程通信

```
osstudy@osstudy-VirtualBox: ~/ex2
osstudy@osstudy-VirtualBox:~$ cd ex2
osstudy@osstudy-VirtualBox:~/ex2$ ls
b1.c  b1.exe  b2.c  b2.exe  box.h  p1.cpp  p1.exe  p2.cpp  p2.exe  sem.h
osstudy@osstudy-VirtualBox:~/ex2$ ./b1.exe
input op:0 send, 1 receive, 2 exit:0
input msg:1
input op:0 send, 1 receive, 2 exit:0
input msg:2
input op:0 send, 1 receive, 2 exit:1
msg: 3
input op:0 send, 1 receive, 2 exit:1
msg: 4
input op:0 send, 1 receive, 2 exit:2
osstudy@osstudy-VirtualBox:~/ex2$

osstudy@osstudy-VirtualBox: ~/ex2
osstudy@osstudy-VirtualBox:~$ cd ex2
osstudy@osstudy-VirtualBox:~/ex2$ ls
b1.c  b1.exe  b2.c  b2.exe  box.h  p1.cpp  p1.exe  p2.cpp  p2.exe  sem.h
osstudy@osstudy-VirtualBox:~/ex2$ ./b2.exe
input op:0 send, 1 receive, 2 exit:1
msg: 1
input op:0 send, 1 receive, 2 exit:1
msg: 2
input op:0 send, 1 receive, 2 exit:0
input msg:3
input op:0 send, 1 receive, 2 exit:0
input msg:4
input op:0 send, 1 receive, 2 exit:2
osstudy@osstudy-VirtualBox:~/ex2$
```

共享内存和信号量实现进程通信



```
osstudy@osstudy-VirtualBox: ~/ex2
osstudy@osstudy-VirtualBox:~$ cd ex2
osstudy@osstudy-VirtualBox:~/ex2$ ./p1.exe
input op:0 send, 1 receive, 2 recall, 3 exit:0
input msg:1
input op:0 send, 1 receive, 2 recall, 3 exit:0
input msg:2
input op:0 send, 1 receive, 2 recall, 3 exit:2
input op:0 send, 1 receive, 2 recall, 3 exit:0
input msg:3
input op:0 send, 1 receive, 2 recall, 3 exit:1
msg:4
input op:0 send, 1 receive, 2 recall, 3 exit:1
msg:5
input op:0 send, 1 receive, 2 recall, 3 exit:3
osstudy@osstudy-VirtualBox:~/ex2$

osstudy@osstudy-VirtualBox: ~/ex2
osstudy@osstudy-VirtualBox:~$ cd ex2
osstudy@osstudy-VirtualBox:~/ex2$ ./p2.exe
input op:0 send, 1 receive, 2 recall, 3 exit:1
msg:1
input op:0 send, 1 receive, 2 recall, 3 exit:1
msg:3
input op:0 send, 1 receive, 2 recall, 3 exit:0
input msg:4
input op:0 send, 1 receive, 2 recall, 3 exit:0
input msg:5
input op:0 send, 1 receive, 2 recall, 3 exit:3
osstudy@osstudy-VirtualBox:~/ex2$
```

## (七) 实验结果分析

预计实验结果的内容都实现了，但是由于截图无法体现挂起等待其他进程操作这个结果。

## (八) 参考资料

[Linux 进程间通信（五）：信号量](#)

[Linux 进程间通信（六）：共享内存](#)

[Linux 进程间通信（七）：消息队列](#)

教材 P94-P95