进程间通信(System V IPC)

参考书目和网站: 《Linux/Unix系统编程手册》、《Linux网络编程》、极客时间

前置介绍

这里的IPC是内核进程通信，主要使用共享内存，消息队列，信号量(还可用于线程间)

简单地讲，共享内存指在主机上绑定一块内存，不同的进程通过一些方式去访问这块内存，这是IPC最快捷的方式，因为共享内存方式的通信没有中间过程，而管道、消息队列等方式则是需要将数据通过中间机制进行转换。与此相反，共享内存方式直接将某段内存段进行映射，多个进程共享内存是同一块物理空间，仅仅是地址不同，因此不须要进行复制，可以直接使用这段空间。

消息队列在主机上指定一个或多个队列，通过队列来传递信息。信号量让多个进程不可能同时访问一块区域，做法是设置一个整数，进程可以给这个整数做加法，也可以做减法，如果这个整数将要小于0，就会导致阻塞。

信号量允许多个进程同步它们的动作。一个信号量是一个由内核维护的整数值，它对所有具备相应权限的进程可见。一个进程通过对信号量的值进行相应的修改，来通知其他进程执行某个动作。

共享内存

对于共享内存，可以根据文件描述符来创建一个key，这里使用ftok函数。

```
NAME
       ftok - convert a pathname and a project identifier to a System V IPC key

SYNOPSIS
       #include <sys/types.h>
       #include <sys/ipc.h>

       key_t ftok(const char *pathname, int proj_id);

DESCRIPTION
       The ftok() function uses the identity of the file named by the given pathname (which must refer to an existing, acces-
       sible file) and the least significant 8 bits of proj_id (which must be nonzero) to generate a key_t type System V  IPC
       key, suitable for use with msgget(2), semget(2), or shmget(2).

       The  resulting  value  is  the same for all pathnames that name the same file, when the same value of proj_id is used.
       The value returned should be different when the (simultaneously existing) files or the project IDs differ.

RETURN VALUE
       On success, the generated key_t value is returned.  On failure -1 is returned, with errno indicating the error as  for
       the stat(2) system call.
```

在终端输入ipcs，就可以查看系统上的IPC通信，指定相关参数会有其他效果

```
OPTIONS
       -i, --id id
              Show full details on just the one resource element identified by id.  This option needs to be combined with one
              of the three resource options: -m, -q or -s.

       -h, --help
              Display help text and exit.

       -V, --version
              Display version information and exit.

   Resource options
       -m, --shmems
              Write information about active shared memory segments.

       -q, --queues
              Write information about active message queues.
```

```
hwx@N3ptune:~$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 5472260    hwx        600        524288     2          dest
0x51091e7b 5341195    hwx        600        5242880    5
0x00000000 5439501    hwx        700        1966080    2          dest
0x00000000 5439502    hwx        700        270336     2          dest
0x85dad214 15         hwx        666        136        1
0x00000000 5439504    hwx        700        270336     2          dest
0xab0981d8 19         hwx        666        136        1
0x00000000 5439508    hwx        700        8192       2          dest
0x8d1318d6 26         hwx        666        136        1
0xa065a7df 27         hwx        666        136        1
0x9398ff38 28         hwx        666        136        1
0x85ff1592 30         hwx        666        136        1
0x00000000 5406759    hwx        700        360448     2          dest
0x00000000 5406763    hwx        700        360448     2          dest
0xfff78447 44         hwx        666        136        1
0x4345dbc1 4980782    hwx        666        136        1
0x00000000 5406776    hwx        700        360448     2          dest
0x00000000 5767225    hwx        600        524288     2          dest
```

代码中要使用shmget函数来创建一个新的共享内存段或者访问一个现有的共享内存段，系统以页为单位来分配(操作系统内存分页模型，这里不作赘述)

```
NAME
       shmget - allocates a System V shared memory segment

SYNOPSIS
       #include <sys/ipc.h>
       #include <sys/shm.h>

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

DESCRIPTION
       shmget()  returns  the identifier of the System V shared memory segment associated with the value of the argument key.
       It may be used either to obtain the identifier of a previously created shared memory segment (when shmflg is zero  and
       key does not have the value IPC_PRIVATE), or to create a new set.

       A new shared memory segment, with size equal to the value of size rounded up to a multiple of PAGE_SIZE, is created if
       key has the value IPC_PRIVATE or key isn't IPC_PRIVATE, no shared memory segment  corresponding  to  key  exists,  and
       IPC_CREAT is specified in shmflg.
```

如下代码简单演示了关于共享内存的使用，如下是进程A

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <signal.h>

int* p = NULL;

void hand()
{
    // 卸载共享内存
    shmdt(p);
    printf("bye\n");
    exit(0);
}

int main(void)
{
    signal(2,hand);
    // 创建key
    key_t key = ftok(".",'m');
    if (-1 == key) {
        printf("ftok error:%m\n");
        exit(-1);
    }
    printf("ftok success\n");
    // 获取共享内存
    int shmid = shmget(key,4096,IPC_CREAT);
    if (-1 == shmid) {
        printf("shmid error:%m\n");
        exit(-1);
    }
    // 挂载共享内存
    p = (int*)shmat(shmid,NULL,0);
    if ((int*)-1 == p) {
        printf("shmat error:%m\n");
        exit(-1);
    }
    printf("shmat success\n");
    // 使用共享内存
    int n = 0;
    while(true) {
        *p = n++;
        printf("%d\n",*p);
```

```
        sleep(1);
    }
    return 0;
}
```

如下是进程B

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <signal.h>

int* p = NULL;
void hand() {
    // 卸载共享内存
    shmdt(p);
    printf("bye\n");
    exit(0);
}

int main(void)
{
    signal(2,hand);
    // 创建key
    key_t key = ftok(".",'m');
    if (-1 == key) {
        printf("ftok error:%m\n");
        exit(-1);
    }
    printf("ftok success\n");
    // 获取共享内存
    int shmid = shmget(key,4096,IPC_CREAT);
    if (-1 == shmid) {
        printf("shmget error:%m\n");
        exit(-1);
    }
    printf("shmget success\n");
    // 挂载共享内存
    p = (int*)shmat(shmid,NULL,0);
    if ((int*)-1 == p) {
        printf("shmat error:%m\n");
        exit(-1);
    }
    printf("shmat success\n");
    // 使用共享内存
    while(true) {
        printf("%d\n",*p);
        sleep(1);
    }
```

```
    return 0;
}
```

先运行进程A，向共享内存写入数据并打印

```
hwx@N3ptune:/data/home/hwx/cprogram/testc/linux-process$ sudo ./shmA
ftok success
shmat success
0
1
2
3
4
5
6
7
8
```

运行进程B，从共享内存中获取数据

```
hwx@N3ptune:/data/home/hwx/cprogram/testc/linux-process$ sudo ./shmB
ftok success
shmget success
shmat success
4
5
6
7
8
```

将进程A中断

```
43
44
45
46
47
48
49
50
51
52
53
^Cbye
```

进程B就只能取出重复的数据，因为进程A不再修改共享内存的数据

```
53
53
53
53
53
53
53
53
53
53
```

下面终端运行命令，查看共享内存，可以看到刚刚分配的4096大小的共享内存

```
hwx@N3ptune:/data/home/hwx/cprogram/testc/linux-process$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x51040068 32778      hwx        600        1          1
0xdcf2d382 32779      hwx        666        136        1
0x85dad214 15         hwx        666        136        1
0xab0981d8 20         hwx        666        136        1
0x8d1318d6 27         hwx        666        136        1
0xa065a7df 28         hwx        666        136        1
0x9398ff38 29         hwx        666        136        1
0x85ff1592 31         hwx        666        136        1
0x00000000 65568      hwx        600        1048576    2          dest
0x00000000 65571      hwx        600        524288     2          dest
0x4345dbc1 40         hwx        666        136        1
0x6d0614f7 48         hwx        0          4096       1
```

使用shmdt和shmctl函数可以删除共享内存

```
NAME
       shmctl - System V shared memory control

SYNOPSIS
       #include <sys/ipc.h>
       #include <sys/shm.h>

       int shmctl(int shmid, int cmd, struct shmid_ds *buf);

DESCRIPTION
       shmctl()  performs  the  control  operation specified by cmd on the System V shared memory segment whose identifier is
       given in shmid.
```

消息队列

下面介绍消息队列

其编程模型也不复杂，同样要创建key，使用ftok函数，然后创建消息队列，使用msgget函数。收发消息使用msgrcv函数和msgsnd函数，删除消息队列，使用msgctl函数

```
NAME
       msgget - get a System V message queue identifier

SYNOPSIS
       #include <sys/types.h>
       #include <sys/ipc.h>
       #include <sys/msg.h>

       int msgget(key_t key, int msgflg);

DESCRIPTION
       The  msgget() system call returns the System V message queue identifier associated with the value of the key argument.
       It may be used either to obtain the identifier of a previously created message queue (when msgflg is zero and key does
       not have the value IPC_PRIVATE), or to create a new set.
```

```
NAME
       msgrcv, msgsnd - System V message queue operations

SYNOPSIS
       #include <sys/types.h>
       #include <sys/ipc.h>
       #include <sys/msg.h>

       int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

       ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
                      int msgflg);

DESCRIPTION
       The  msgsnd() and msgrcv() system calls are used, respectively, to send messages to, and receive messages from, a Sys-
       tem V message queue.  The calling process must have write permission on the message queue in order to send a  message,
       and read permission to receive a message.
```

文档中指明所发消息缓冲区的结构，包含了消息类型和数据

```
The msgp argument is a pointer to a caller-defined structure of the following general form:

    struct msgbuf {
        long mtype;       /* message type, must be > 0 */
        char mtext[1];    /* message data */
    };
```

下面指明了所发消息的类型

```
  IPC_NOWAIT
         Return immediately if no message of the requested type is in the queue.  The system call fails with  errno  set
         to ENOMSG.

  MSG_COPY (since Linux 3.8)
         Nondestructively fetch a copy of the message at the ordinal position in the queue specified by msgtyp (messages
         are considered to be numbered starting at 0).

         This flag must be specified in conjunction with IPC_NOWAIT, with the result that, if there is no message avail-
         able  at  the given position, the call fails immediately with the error ENOMSG.  Because they alter the meaning
         of msgtyp in orthogonal ways, MSG_COPY and MSG_EXCEPT may not both be specified in msgflg.

         The MSG_COPY flag was added for the implementation of the kernel checkpoint-restore facility and  is  available
         only if the kernel was built with the CONFIG_CHECKPOINT_RESTORE option.

  MSG_EXCEPT
         Used with msgtyp greater than 0 to read the first message in the queue with message type that differs from msg-
         typ.
```

接收的时候，msgrcv函数指明类型

内核中有一个msgid_ds结构。I这于消息队列而言，它的内部数据结构是msgid_ds结构，对于系统上创建的每个消息队列，内核均为其创建、存储和维护该结构的一个实例。

```
struct msqid_ds {
    struct ipc_perm msg_perm;     /* Ownership and permissions */
    time_t          msg_stime;    /* Time of last msgsnd(2) */
    time_t          msg_rtime;    /* Time of last msgrcv(2) */
    time_t          msg_ctime;    /* Time of last change */
    unsigned long   __msg_cbytes; /* Current number of bytes in
                                     queue (nonstandard) */
    msgqnum_t       msg_qnum;     /* Current number of messages
                                     in queue */
    msglen_t        msg_qbytes;   /* Maximum number of bytes
                                     allowed in queue */
    pid_t           msg_lspid;    /* PID of last msgsnd(2) */
    pid_t           msg_lrpid;    /* PID of last msgrcv(2) */
};
```

下面代码实现两个进程的通信

如下是进程A

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <unistd.h>
#include <time.h>
#include <sys/ipc.h>
#include <fcntl.h>

#define MAX_LEN 64

struct msg_mbuf
{
    int type;
    char text[64];
};
int main(void)
{
    key_t key = ftok(".",'z');
    if (key == -1) {
        printf("ftok error:%m\n");
        exit(-1);
    }
    int msgid = msgget(key,0x666|IPC_CREAT|O_WRONLY);
    if (msgid == -1) {
        printf("msgget error:%m\n");
        exit(-1);
    }
    struct msg_mbuf data;
    strcpy(data.text,"hello message");
    int ret  = msgsnd(msgid,(void*)&data,sizeof(data),0);
    if (ret == -1) {
        printf("msgsnd error:%m\n");
        exit(-1);
    }
    sleep(50);
    return 0;
}
```

如下是进程B

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <unistd.h>
#include <time.h>
#include <sys/ipc.h>
#include <fcntl.h>

struct msg_mbuf
{
    int type;
    char text[64];
};
int main(void)
{
    key_t key = ftok(".",'z');
    if (key == -1) {
        printf("ftok error:%m");
        exit(-1);
    }
    int msgid = msgget(key,O_RDONLY);
    if (msgid == -1) {
        printf("msgget error:%m\n");
        exit(-1);
    }
    struct msg_mbuf data;
    int ret = msgrcv(msgid,&data,sizeof(data),0,0);
    if (ret == -1) {
        printf("msgrcv error:%m\n");
        exit(-1);
    }
    data.text[ret] = 0;
    printf("%s\n",data.text);
    return 0;
}
```

运行结果

```
hwx@N3ptune:/data/home/hwx/cprogram/testc/linux_ipc$ sudo ./msgB
hello message
```

如下代码，打印了消息队列的一系列信息，并且在最后尝试修改了信息

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <unistd.h>
#include <time.h>
#include <sys/ipc.h>

// 显示消息队列属性
void msg_show_attr(int msg_id,struct msqid_ds msg_info)
{
    int ret = -1;
    sleep(1);
    ret = msgctl(msg_id,IPC_STAT,&msg_info);
    if (-1 == ret) {
        printf("获取消息信息失败\n");
        return;
    }
    printf("\n");
    printf("队列中的字节数:%ld\n",msg_info.msg_cbytes);
    printf("队列中的消息数:%ld\n",msg_info.msg_qnum);
    printf("队列中的最大字节数:%ld\n",msg_info.msg_qbytes);
    printf("最后发送消息的进程pid:%d\n",(int)msg_info.msg_lspid);
    printf("最后接收消息的进程pid:%d\n",(int)msg_info.msg_lrpid);
    printf("最后发送消息的时间:%s",ctime(&msg_info.msg_stime));
    printf("最后接收消息的时间:%s",ctime(&msg_info.msg_rtime));
    printf("最后变化的时间:%s",ctime(&msg_info.msg_ctime));
    printf("消息UID:%d\n",msg_info.msg_perm.uid);
    printf("消息GID:%d\n",msg_info.msg_perm.gid);
}

int main(void)
{
    int ret = -1;
    int msg_flags,msg_id;
    // 定义缓冲区
    struct msgmbuf {
        int mtype;
        char mtext[10];
    };
    // 定义了消息结构信息
    struct msqid_ds msg_info;
    struct msgmbuf msg_mbuf;

    int msg_sflags,msg_rflags;
    key_t key = ftok(".",'b'); // 创建key
```

```
    if (key == -1) {
        printf("ftok error:%m\n");
    }
    else
        printf("ftok success\n");
    // 获取消息队列ID
    msg_id = msgget(key,IPC_CREAT|IPC_EXCL|0x666);
    if (msg_id == -1) {
        printf("msgget error:%m\n");
    }
    else
        printf("msgget success\n");

    msg_show_attr(msg_id,msg_info);
    msg_sflags = IPC_NOWAIT;
    msg_mbuf.mtype = 10;
    char text[] = "test message";
    memcpy(msg_mbuf.mtext,text,sizeof(text));
    ret = msgsnd(msg_id,&msg_mbuf,sizeof(text),msg_flags);
    if (-1 == ret) {
        printf("msgsnd error:%m\n");
    }
    else
        printf("msgsnd success\n");

    msg_show_attr(msg_id,msg_info);
    msg_rflags = IPC_NOWAIT|MSG_NOERROR;
    // 接收消息
    ret = msgrcv(msg_id,&msg_mbuf,10,10,msg_rflags);
    if (-1 == ret) {
        printf("msgrcv error:%m\n");
    }
    else
        printf("msgrcv success\n");

    msg_show_attr(msg_id,msg_info);

    msg_info.msg_perm.uid = 8;
    msg_info.msg_perm.gid = 8;
    msg_info.msg_qbytes = 12345;
    ret = msgctl(msg_id,IPC_SET,&msg_info);
    if (-1 == ret) {
        printf("msgctl error:%m\n");
    }
    else
        printf("msgctl success\n");
    msg_show_attr(msg_id,msg_info);
    // 删除消息队列
```

```
    ret = msgctl(msg_id,IPC_RMID,NULL);
    if (-1 == ret) {
        printf("msgctl error:%m\n");
    }
    else
        printf("msgctl success\n");

    return 0;
}
```

```
ftok success
msgget success

队列中的字节数:0
队列中的消息数:0
队列中的最大字节数:16384
最后发送消息的进程pid:0
最后接收消息的进程pid:0
最后发送消息的时间:Thu Jan  1 08:00:00 1970
最后接收消息的时间:Thu Jan  1 08:00:00 1970
最后变化的时间:Sun Apr  3 19:46:21 2022
消息UID:0
消息GID:0
msgsnd success

队列中的字节数:13
队列中的消息数:1
队列中的最大字节数:16384
最后发送消息的进程pid:15502
最后接收消息的进程pid:0
最后发送消息的时间:Sun Apr  3 19:46:22 2022
最后接收消息的时间:Thu Jan  1 08:00:00 1970
最后变化的时间:Sun Apr  3 19:46:21 2022
消息UID:0
消息GID:0
msgrcv error:No message of desired type

队列中的字节数:13
队列中的消息数:1
队列中的最大字节数:16384
最后发送消息的进程pid:15502
最后接收消息的进程pid:0
最后发送消息的时间:Sun Apr  3 19:46:22 2022
最后接收消息的时间:Thu Jan  1 08:00:00 1970
最后变化的时间:Sun Apr  3 19:46:21 2022
```

```
队列中的字节数:13
队列中的消息数:1
队列中的最大字节数:12345
最后发送消息的进程pid:15502
最后接收消息的进程pid:0
最后发送消息的时间:Sun Apr  3 19:46:22 2022
最后接收消息的时间:Thu Jan  1 08:00:00 1970
最后变化的时间:Sun Apr  3 19:46:24 2022
消息UID:8
消息GID:8
msgctl success
```

信号量

事实上，信号量并不是用来在进程间传输数据的。相反，它们用来同步进程的动作。信号量的一个常见用途是同步对一块共享内存的访问，以防止出现一个进程在访问共享内存的同时，另一个进程更新这块内存。

sem信号量，本质上是一个整数，编程模型：

创建key (ftok)，创建信号量(semget)，初始化信号量(semctl)，使用信号量(semop)，删除信号量(semctl)

```
NAME
       semget - get a System V semaphore set identifier

SYNOPSIS
       #include <sys/types.h>
       #include <sys/ipc.h>
       #include <sys/sem.h>

       int semget(key_t key, int nsems, int semflg);

DESCRIPTION
       The  semget()  system  call  returns the System V semaphore set identifier associated with the argument key.  It may be
       used either to obtain the identifier of a previously created semaphore set (when semflg is zero and key does  not  have
       the value IPC_PRIVATE), or to create a new set.
```

semflg是位掩码，和权限相关

semctl系统调用在一个信号量集或集合中单个信号量上执行各种控制操作

```
NAME
       semctl - System V semaphore control operations

SYNOPSIS
       #include <sys/types.h>
       #include <sys/ipc.h>
       #include <sys/sem.h>

       int semctl(int semid, int semnum, int cmd, ...);

DESCRIPTION
       semctl()  performs  the control operation specified by cmd on the System V semaphore set identified by semid, or on the sem
num-th semaphore of that set.  (The
       semaphores in a set are numbered starting at 0.)
```

semid参数是操作所施加的信号量集的标识符。

这里有一个重要的联合体，作为该函数的参数

```
This function has three or four arguments, depending on cmd.  When there are four, the fourth has the type union semun.
The calling program must define this union as follows:

    union semun {
        int              val;    /* Value for SETVAL */
        struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
        unsigned short  *array;  /* Array for GETALL, SETALL */
        struct seminfo  *__buf;  /* Buffer for IPC_INFO
                                    (Linux-specific) */
    };
```

如下是semop函数，负责操作信号量的值

```
NAME
       semop, semtimedop - System V semaphore operations

SYNOPSIS
       #include <sys/types.h>
       #include <sys/ipc.h>
       #include <sys/sem.h>

       int semop(int semid, struct sembuf *sops, size_t nsops);

       int semtimedop(int semid, struct sembuf *sops, size_t nsops,
                      const struct timespec *timeout);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       semtimedop(): _GNU_SOURCE

DESCRIPTION
       Each  semaphore  in  a  System V semaphore set has the following associated
       values:

           unsigned short  semval;   /* semaphore value */
           unsigned short  semzcnt;  /* # waiting for zero */
           unsigned short  semncnt;  /* # waiting for increase */
           pid_t           sempid;   /* PID of process that last
```

这个函数稍稍有点麻烦

semid标识了是信号量集中的一个信号量，sops参数是一个指向数组的指针，数组中包含了要执行的操作，nsops参数给出了数组的大小(至少要包含一个元素)。操作将会按照数组中的顺序以原子的方式执行。

sembuf结构体包含了一下成员

```
unsigned short sem_num;  /* semaphore number */
short          sem_op;   /* semaphore operation */
short          sem_flg;  /* operation flags */
```

sem_num字段标识出了在集合中的哪个信号量上执行操作。sem_op字段指定了要执行的操作。如果sem_op大于0，那么就将sem_op的值加到信号量值上，其结果是等待其他减小信号量值的进程被唤醒，并执行它们的操作，调用进程必须具备在信号量上的写权限。

如果等于0，那么就对信号量的值进行检查以确定它当前是否等于0，如果信号量的值等于0，那么操作立即结束，否则semop就会阻塞直到信号量值变成0为止。调用进程必须要具备在信号量上的读权限。

如果sem_op小于0，那么就将信号量减去sem_op;p。如果信号量的当前值大于或等于sem_op的绝对值，那么操作会立即结束。否则semop会阻塞直到信号量的值增长到在执行操作之后不会导致出现负值的情况，调用进程必须在信号量上具备写权限。

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <signal.h>
#include <stdbool.h>

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

// 创建信号量
int create_sem(key_t key,int value)
{
    union semun sem;
    sem.val = value;
    int semid = semget(key,0,IPC_CREAT|0666);
    if (-1 == semid) {
        printf("semget error:%m\n");
        return -1;
    }
    semctl(semid,0,SETVAL,sem);
    return semid;
}

// 设置信号量值
void set_sem_value(int semid,int value)
{
    union semun sem;
    sem.val = value;
    semctl(semid,0,SETVAL,sem);
}

// 获取信号量值
int get_sem_value(int semid)
{
    union semun sem;
    return semctl(semid,0,GETVAL,sem);
}

// 销毁信号量
void destory_sem(int semid)
```

```
{
    union semun sem;
    sem.val = 0;
    semctl(semid,0,IPC_RMID,sem);
}

// 增加信号量
int add_sem_value(int semid)
{
    struct sembuf sops;
    sops.sem_num = 0;
    sops.sem_op = 1;
    sops.sem_flg = IPC_NOWAIT;
    return (semop(semid,&sops,1));
}

// 减少信号量
int sub_sem_value(int semid)
{
    struct sembuf sops;
    sops.sem_num = 0;
    sops.sem_op = -1;
    sops.sem_flg = IPC_NOWAIT;
    return (semop(semid,&sops,1));
}

int main(void)
{
    key_t key = ftok(".",'m');
    if (key == -1) {
        printf("ftok error:%m\n");
        exit(-1);
    }
    printf("ftok success\n");
    int semid = create_sem(key,100);
    for (int i=0;i<=3;i++)
    {
        add_sem_value(semid);
        sub_sem_value(semid);
    }
    int value = get_sem_value(semid);
    printf("value: %d",value);
    destory_sem(semid);
    return 0;
}
```

写在最后

上述三个进程间通信机制都属于System V IPC，将三者的共性与特性可以进行归纳

接口	消息队列	信号量	共享内存
头文件	sys/msg.h	sys/sem.h	sys/shm.h
关联数据结构	msqid_ds_	semid_ds_	shmid_ds_
创建/打开对象	msgget	semget	shmget+shmat
关闭对象	无	无	shmdt
控制操作	msgctl	semctl	shmctl
执行IPC	msgsnd/msgrcv	semop	访问共享区域内存

无论是哪一种机制，都有一个相关的get系统调用(msgget、semget和shmget)，它与文件上open系统调用相似。给定一个整数key，get系统调用完成下列两种操作：

1. 使用给定的key创建一个新IPC对象并返回一个唯一的标识符来标识对象
2. 返回一个拥有给定的key的已有IPC对象标识符。在这种情况下，get调用所做的事情是将一个数字(key)转换为另一个数字(标识符)

各种System V IPC机制得到ctl系统调用(msgctl()、semctl、shmctl)在对象上执行一组控制操作，其中很多操作是特定于某种IPC机制，但有一些是适用于所有IPC机制的，其中一个就是IPC_RMID控制操作，它可以用来删除一个对象。

对于消息队列和信号量来讲，IPC对象的删除是立即生效的，对象中包含的所有信息都会被销毁，不管是否其他进程仍然在使用该对象。

共享内存对象的删除和操作却不同，shmctl(id,IPC_RMID,NULL)调用之后，只有当所有使用该内存段的进程与该内存段分离之后(使用shmdt)才会删除该共享内存段。

上述代码中无一没有定义一个key，System IPC key是一个整数值，其数据类型为key_t。get调用将其转换为相应的IPC标识符。