


# LINUX进程通信

SYSTEM V IPC(续)

# 进程通信（三）：

## SYSTEM V IPC:共享内存\消息队列\信号量

1. SYSTEM V IPC简介
  2. SHELL环境控制IPC
  3. 进程间通信关键字
  4. 进程间通信标识符
  5. IPC权限许可结构
- 
- A series of three parallel white diagonal lines are located in the bottom right corner of the slide, extending from the middle of the right edge towards the bottom-left.

# 1. SYSTEM v IPC简介

## SystemV IPC三种类型

- SystemV消息队列 `sys/msg.h`
- SystemV信号灯 `sys/sem.h`
- SystemV共享内存区 `sys/shm.h`

## 创建或打开函数

- `msgget`, `semget`, `shmget`

## 控制操作函数

- `msgctl`, `semctl`, `shmctl`

## 操作函数

- `msgsnd`, `msgrcv`, `semop`, `shmat`, `shmdt`

# SYSTEM V关键字

每一个System V 对象(消息队列, 共享内存和信号量)创建时, 需要的第一个参数是整数的Key值,

- ✓ 头文件<sys/types.h>把key\_t定义为一个整数

System V 创建对象时假设进行IPC通讯双方都取了相同的key值. 这样将双方关联起来

生成key的方法有三种

- ✓ 双方直接设置为一个相同的整数为key值
- ✓ 用IPC\_PRIVATE让系统自动产生一个key值,
- ✓ 用ftok函数将一个路径转换为key值

# FTOK函数

ftok函数把一个已存在的路径名和一个整数标识符转换成一个key\_t值，称为IPC键(IPC key)：

```
#include <sys/ipc.h>
```

```
key_t ftok(const char *pathname, int id);
```

- ✓ 如果pathname不存在，或者对调用进程不可访问，ftok返回-1
- ✓ 不能保证两个不同的路径名与同一个id值的组合产生不同的键。
- ✓ 用于产生键的pathname不能是服务器存活期间由它反复创建并删除的文件，否则会导致ftok多次调用返回不同的值

# SYSTEM V IPC的类型

- **报文**（Message）队列（消息队列）：消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。
- **共享内存**：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。
- **信号量**（semaphore）：主要作为进程间以及同一进程不同线程之间的同步手段。

## 2. SHELL环境控制IPC

内核为IPC对象建立相应的数据结构

- ✓ 通过*ipcs*查看当前系统中的IPC对象情况

----- Shared Memory Segments -----						
key	shmid	owner	perms	bytes	nattch	status
0x74015cce	2883584	root	600	4	0	
----- Semaphore Arrays -----						
key	semid	owner	perms	nsems		
----- Message Queues -----						
key	msqid	owner	perms	used-bytes	messages	

- ✓ 通过*ipcrm*删除一个IPC对象



### 3. 进程通信-共享内存

#### IPC-KEY值

每个IPC对象都具有一个唯一的标识符，保证不同的进程能够获取同一个IPC对象，必须提供一个 IPC关键字(IPC KEY)，内核将IPC关键字转换成IPC标识符。

IPC关键字可以通过系统调用***ftok***函数获得：

***key\_t ftok(const char \*pathname, int proj\_id)***

- ***\*pathname*** 指定的文件名(文件必须存在且可访问)
- ***proj\_id*** 标识号(1-255)

两进程如在参数上达成一致，双方就都能够通过调用***ftok***函数得到同一个IPC键。

函数执行成功则会返回一个KEY\_T键值，失败则返回-1.



## 1). 获取KEY值

```
//get ipc key value  
key=ftok("./shmsvr.c",1);  
if (key==-1)  
{  
    perror("ftok error!");  
    exit(1);  
}
```

进程1

```
//get ipc key value  
key=ftok("./shmsvr.c",1);  
if (key==-1)  
{  
    perror("ftok error!");  
    exit(1);  
}
```

进程2

## 2). 申请一个共享内存空间

```
//get shared m with size of SHM_SIZE
shm_id=shmget(key, SHM_SIZE, IPC_CREAT);
if (shm_id==-1)
{
    perror("shmget error!");
    exit(2);
}
```

shmget(得到一个共享内存标识符或创建一个共享内存对象)

所需  
头文  
件

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

函数  
说明

得到一个共享内存标识符或创建一个共享内存对象并返回共享内存标识符

函数  
原型

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

函数  
传入  
值

key	0(IPC_PRIVATE)：会建立新共享内存对象
	大于0的32位整数：视参数shmflg来确定操作。通常要求此值来源于ftok返回的IPC键值
size	大于0的整数：新建的共享内存大小，以字节为单位
	0：只获取共享内存时指定为0
shmflg	0：取共享内存标识符，若不存在则函数会报错
	IPC_CREAT：当shmflg&IPC_CREAT为真时，如果内核中不存在键值与key相等的共享内存，则新建一个共享内存；如果存在这样的共享内存，返回此共享内存的标识符
	IPC_CREAT IPC_EXCL：如果内核中不存在键值与key相等的共享内存，则新建一个消息队列；如果存在这样的共享内存则报错

函数  
返回  
值

成功：返回共享内存的标识符

出错：-1，错误原因存于error中

### 3). 映射共享区到进程地址空间

```
//map the shm to user space
shm_p=shmat(shm_id, NULL, 0);
if (shm_p==NULL)
{
    perror("shm mapping error!");
    exit(3);
}
```

shmat(把共享内存区对象映射到调用进程的地址空间)

所需头文件	#include <sys/types.h> #include <sys/shm.h>	
函数说明	连接共享内存标识符为shmids的共享内存，连接成功后把共享内存区对象映射到调用进程的地址空间，随后可像本地空间一样访问	
函数原型	void *shmat(int shmids, const void *shmaddr, int shmflg)	
函数传入值	shmids	共享内存标识符
	shmaddr	指定共享内存出现在进程内存地址的什么位置，直接指定为NULL让内核自己决定一个合适的地址位置
	shmflg	SHM_RDONLY：为只读模式，其他为读写模式
函数返回值	成功：附加好的共享内存地址	
	出错：-1，错误原因存于error中	

## 4) 使用共享内存通信

### 发送信息

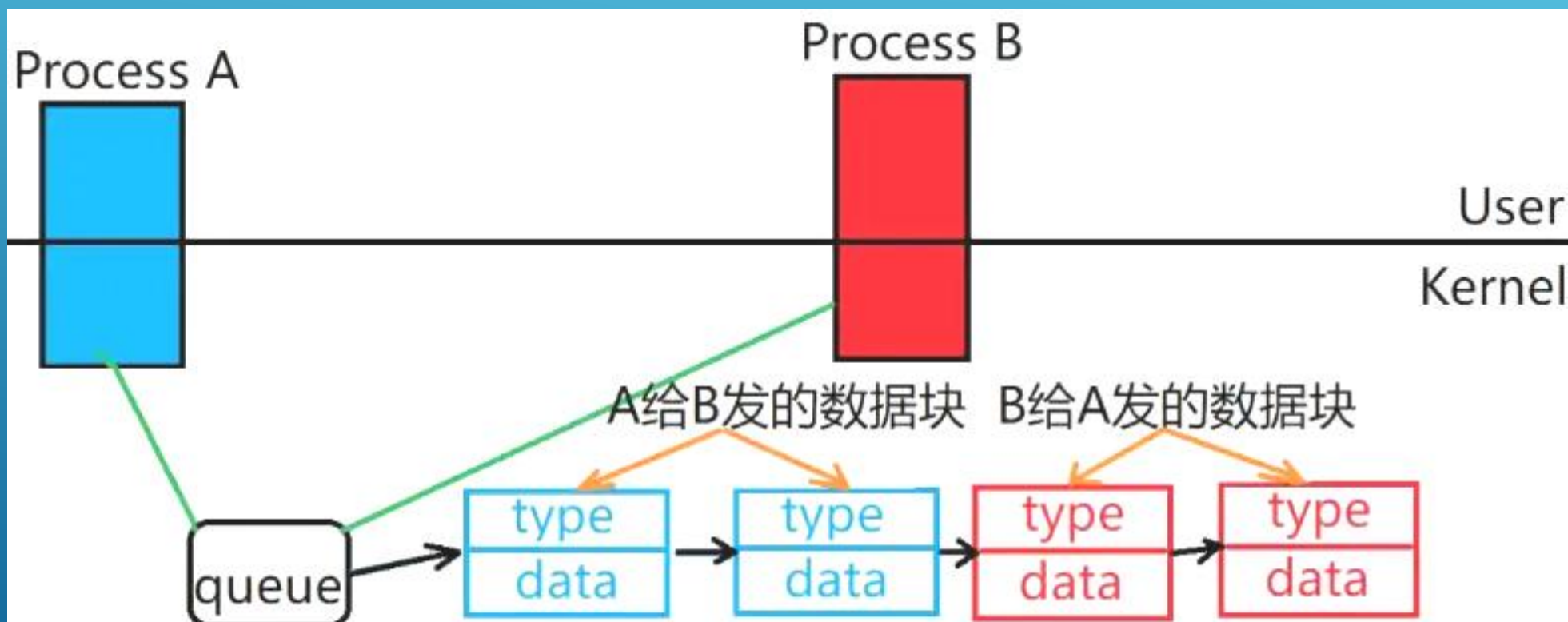
```
//write data to the shared mblock.  
memset(shm_p, 0, SHM_SIZE);  
strcpy(shm_p, "Hi, boys and girls, this msg is form shared memory writed by shmsvr program!\n");
```

### 接收信息

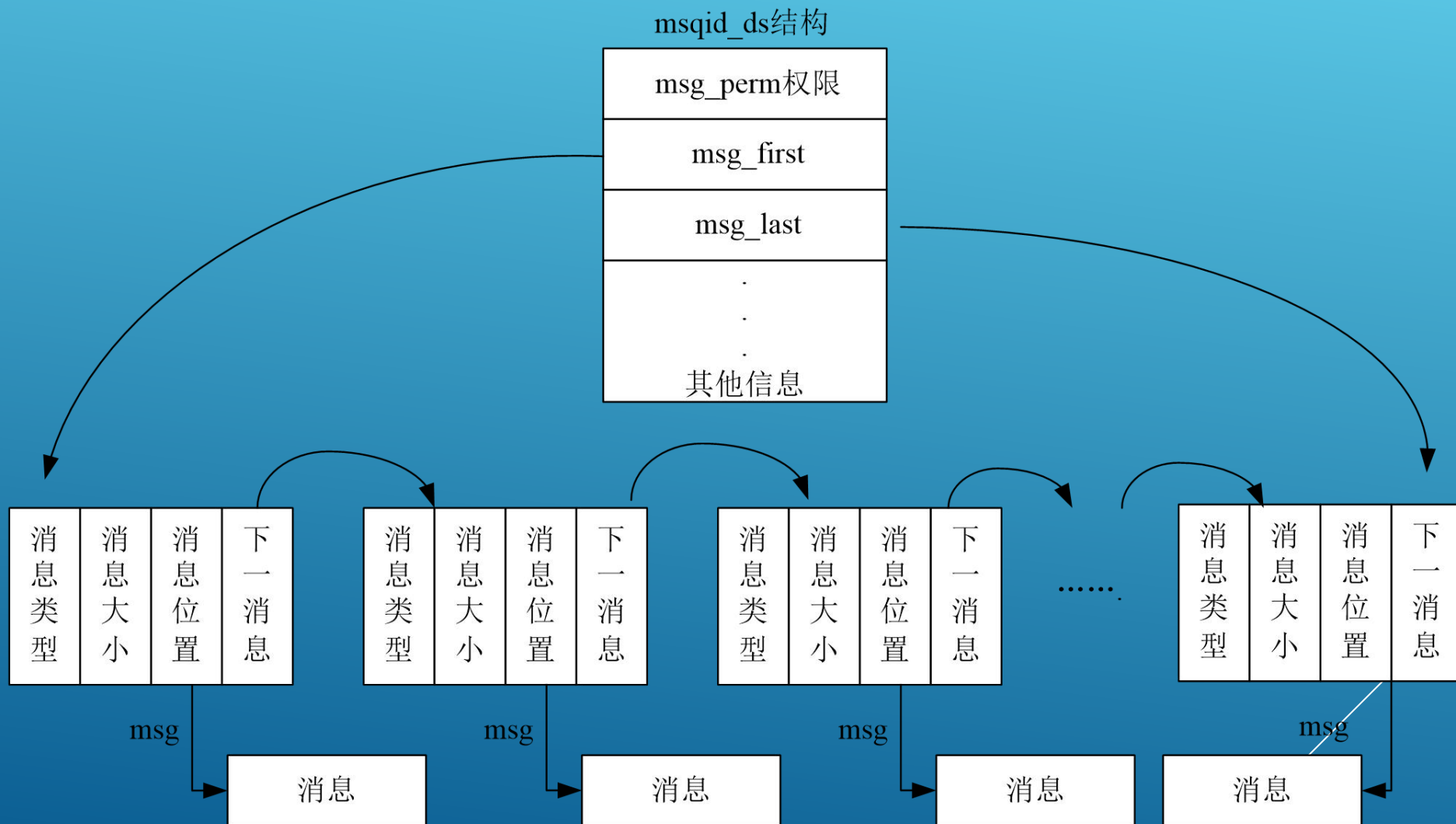
```
//Read data from the shared mblock.  
memset(buf, 0, SHM_SIZE);  
strcpy(buf, shm_p);  
printf("The received data is: %s", buf);
```



## 4. 进程通信——消息队列



## 4. 进程通信——消息队列



# 消息队列结构

```
struct msqid_ds {  
    struct ipc_perm msg_perm;           //权限  
    struct msg *msg_first;               /* first message on queue,unused */ //指向消息头  
    struct msg *msg_last;               /* last message in queue,unused */ //指向消息尾  
    __kernel_time_t msg_stime;          /* last msgsnd time */           //最近发送消息时间  
    __kernel_time_t msg_rtime;          /* last msgrev time */           //最近接收消息时间  
    __kernel_time_t msg_ctime;          /* last change time */           //最近修改时间  
    unsigned long msg_lbytes;           /* Reuse junk fields for 32 bit */  
    unsigned long msg_lqbytes;          /* ditto */  
    unsigned short msg_cbytes;           /* current number of bytes on queue */ //当前队列大小  
    unsigned short msg_qnum;             /* number of messages in queue */ //当前队列消息个数  
    unsigned short msg_qbytes;           /* max number of bytes on queue */ //队列最大值  
    __kernel_ipc_pid_t msg_lspid;        /* pid of last msgsnd */           //最近msgsnd的pid  
    __kernel_ipc_pid_t msg_lrpid;        /* last receive pid */           //最近receive的pid  
};
```



# 消息结构

```
//come from /usr/src/kernels/`uname -r`/include/linux/msg.h
67 /* one msg_msg structure for each message */
68 struct msg_msg {
69     struct list_head m_list;
70     long m_type;                //消息类型
71     int m_ts;                   /* message text size */ //消息大小
72     struct msg_msgseg* next;    //下一个消息位置
73     void *security;             //真正消息位置
74     /* the actual message follows immediately */
75 };
```

# 创建消息队列

```
extern int msgget (key_t __key, int __msgflg);
```

第一个参数 `key` 为由 `ftok` 创建的 `key` 值，关于 `ftok` 函数本章在前一小节已经介绍。

第二个参数 `__msgflg` 的低位用来确定消息队列的访问权限。如 `0770`，为文件的访问权限类型。此外，还可以附加以下参数值。这些值可以与基本权限以或的方式一起使用。

```
//come from /usr/include/bit/ipc.h
/* resource get request flags */
#define IPC_CREAT  00001000    /* create if key is nonexistent */如果key不存在，则创建
                                   //存在，返回ID
#define IPC_EXCL   00002000    /* fail if key exists */    如果key存在，返回失败
#define IPC_NOWAIT 00004000    /* return error on wait */    如果需要等待，直接返回错误
```

# 消息队列属性控制

```
extern int msgctl (int __msqid, int __cmd, struct msqid_ds *__buf);
```

其包括三个参数：

第一个参数\_\_msqid 为消息队列标识符，该值为使用 msgget 函数创建消息队列的返回值。

第二个参数\_\_cmd 为执行的控制命令，即要执行的操作。包括以下选项：

```
//come from /usr/include/linux/ipc.h
```

```
/* Control commands used with semctl, msgctl and shmctl see also specific commands in sem.h, msg.h and  
shm.h */
```

#define IPC_RMID 0	/* remove resource */	//删除
#define IPC_SET 1	/* set ipc_perm options */	//设置ipc_perm参数
#define IPC_STAT 2	/* get ipc_perm options */	//获取ipc_perm参数
#define IPC_INFO 3	/* see ipcs */	//如ipcs

# 发送信息到消息队列

```
extern int msgsnd (int __msqid, __const void *__msgp, size_t __msgsz, int __msgflg);
```

此函数参数说明如下：

第一个参数 `msqid` 为指定的消息队列标识符（由 `msgget` 生成的消息队列标识符），即将消息添加到哪个消息队列中。

第二个参数 `msgp` 为指向的用户定义缓冲区，下面是用户定义缓冲区结构：

```
//come from /usr/include/linux/msg.h
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
    long mtype;           /* type of message */           //消息类型
    char mtext[1];        /* message text */           //消息内容，在使用时自己重新定义此结构
};
```

- `mtype` 是一个正整数，由产生消息的进程生成，用于表示消息的类型，因此，接收进程可以用来进行消息选择（消息队列在存储信息时是按发送的先后顺序放置的）。
- `mtext` 是文本内容，即消息内容。此处大小为 1，显示不够用，在使用时自己重新定义此结构。



## 发送信息到消息队列（2）

第三个参数为接收信息的大小，其数据类型为 `size_t`，即 `unsigned int` 类型。其大小为 0 到系统对消息队列的限制值。

第四个参数用来指定在达到系统为消息队列所定的界限（如达到字数限制）时应采取的操作。

- 如果设置为 `IPC_NOWAIT`，如果需要等待，则不发送消息并且调用进程立即返回错误信息 `EAGAIN`。
- 如果设置为 0，则调用进程挂起执行，直到达到系统所规定的最大值为止，并发送

消息。

成功调用后，此函数将返回 0，否则返回 01，同时将对消息队列 `msqid` 数据结构的成员执行下列操作：

- `msg_qnum` 以 1 为增量递增。
- `msg_lspid` 设置为调用进程的进程 ID。
- `msg_stime` 设置为当前时间。

# 从消息队列接收信息

```
extern int msgrev (int __msqid, void *__msgp, size_t __msgsz, long int __msgtyp, int __msgflg);
```

此函数从与 `msqid` 指定的消息队列标识符相关联的队列中读取消息，并将其放置到由 `msgp` 指向的结构中。

第一个参数为读的对象，即从哪个消息队列获得消息。

第二个参数为一个临时消息数据结构，用来保存读取的信息。其定义如下：

```
//come from /usr/include/linux/msg.h
```

```
/* message buffer for msgsnd and msgrev calls */
```

```
struct msgbuf {
```

```
    long mtype;           /* type of message */    //消息类型
```

```
    char mtext[1];        /* message text */        //存储消息位置，需要重新定义
```

```
};
```

`mtype` 是接收消息的类型（由发送进程指定）。`mtext` 为消息的文本。

第三个参数 `msgsz` 指定 `mtext` 的大小（以字节为单位）。如果收到的消息大于 `msgsz`，并且 `msgflg & MSG_NOERROR` 为真，则将该消息截至 `msgsz` 字节，消息的截断部分将丢失，并且不向调用进程提供截断的提示。

第四个参数 `msgtyp` 指定请求的消息类型：

- `msgtyp = 0` 收到队列中的第一条消息，任意类型。
- `msgtyp > 0` 收到第一条 `msgtyp` 类型的消息。
- `msgtyp < 0` 收到第一条最低类型（小于或等于 `msgtyp` 的绝对值）的消息。

第五个参数 `msgflg` 指定所需类型消息不在队列上时将要采取的操作。

# 消息队列参考代码：发送者

```
struct msg_st
{
    long int msg_type;
    char text[MAX_TEXT];
};
int main()
{
    int running = 1;
    struct msg_st data;
    char buffer[BUFSIZ];
    int msgid = -1;
    //建立消息队列
    msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
    if(msgid == -1)
    {
        fprintf(stderr, "msgget failed with error: %d\n", errno);
        exit(EXIT_FAILURE);
    }
    //向消息队列中写消息，直到写入end
```

```
while(running)
{
    //输入数据
    printf("Enter some text: ");
    fgets(buffer, BUFSIZ, stdin);
    data.msg_type = 1;
    strcpy(data.text, buffer);
    //向队列发送数据
    if(msgsnd(msgid, (void*)&data, MAX_TEXT, 0) == -1)
    {
        fprintf(stderr, "msgsnd failed\n");
        exit(EXIT_FAILURE);
    }
    //输入end结束输入
    if(strncmp(buffer, "end", 3) == 0)
        running = 0;
    sleep(1);
}
exit(EXIT_SUCCESS);
}
```



# 消息队列参考代码：接收者

```
struct msg_st
{
    long int msg_type;
    char text[BUFSIZ];
};

int main()
{
    int running = 1;
    int msgid = -1;
    struct msg_st data;
    long int msgtype = 0;
    //建立消息队列
    msgid = msgget((key_t)1234, 0666 |
IPC_CREAT);
    if(msgid == -1)
```

```
    {
        fprintf(stderr, "msgget failed with error: %d\n",
errno);
        exit(EXIT_FAILURE);
    }
    //从队列中获取消息，直到遇到end消息为止
```

```
while(running)
{
    if(msgrcv(msgid, (void*)&data, BUFSIZ,
msgtype, 0) == -1)
    {
        fprintf(stderr, "msgrcv failed with errno:
%d\n", errno);
        exit(EXIT_FAILURE);
    }
    printf("Received msg: %s\n",data.text);
    //遇到end结束
    if(strncmp(data.text, "end", 3) == 0)
        running = 0;
}
//删除消息队列
if(msgctl(msgid, IPC_RMID, 0) == -1)
{
    fprintf(stderr, "msgctl(IPC_RMID)
failed\n");
    exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
}
```

# 附1：MSGGET函数原型

```
//create a message queue.
msg_id=msgget(key, IPC_CREAT);
if (msg_id==-1)
{
    perror("msgget error!");
    exit(2);
}
```

msgget(得到消息队列标识符或创建一个消息队列对象)		
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>	
函数说明	得到消息队列标识符或创建一个消息队列对象并返回消息队列标识符	
函数原型	int msgget(key_t key, int msgflg)	
函数传入值	key	0(IPC_PRIVATE)：会建立新的消息队列
		大于0的32位整数：视参数msgflg来确定操作。通常要求此值来源于ftok返回的IPC键值
	msgflg	0：取消息队列标识符，若不存在则函数会报错
		IPC_CREAT：当msgflg&IPC_CREAT为真时，如果内核中不存在键值与key相等的消息队列，则新建一个消息队列；如果存在这样的消息队列，返回此消息队列的标识符
函数返回值	成功：返回消息队列的标识符	
	出错：-1，错误原因存于error中	

# 附2：MSGSEND函数原型

```
//send data to message queue.
ret=msgsnd(msg_id, &msg, sizeof(msgbuf),0);
if (ret==-1)
{
    perror("send message error!");
    exit(3);
}
```

msgsnd (将消息写入到消息队列)		
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>	
函数说明	将msgp消息写入到标识符为msqid的消息队列	
函数原型	int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)	
函数传入值	msqid	消息队列标识符
	msgp	发送给队列的消息。msgp可以是任何类型的结构体，但第一个字段必须为long类型，即表明此发送消息的类型，msgrcv根据此接收消息。msgp定义的参照格式如下： struct s_msg{ /*msgp定义的参照格式*/ long type; /* 必须大于0,消息类型 */ char mtext[256]; /*消息正文，可以是其他任何类型*/ } msgp;
	msgsz	要发送消息的大小，不含消息类型占用的4个字节,即mtext的长度
	msgflg	0：当消息队列满时，msgsnd将会阻塞，直到消息能写进消息队列
		IPC_NOWAIT：当消息队列已满的时候，msgsnd函数不等待立即返回
函数返回值	成功：0	
	出错：-1，错误原因存于error中	