进程间的五种通信方式介绍 - moon-zhu - 博客园

进程间通信（IPC，InterProcess Communication）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、共享存储、Socket、Streams等。其中 Socket和Streams支持不同主机上的两个进程IPC。

以Linux中的C语言编程为例。

一、管道

管道，通常指无名管道，是 UNIX 系统IPC最古老的形式。

1、特点：

1. 它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端。
2. 它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）。
3. 它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

一、管道

管道，通常指无名管道，是 UNIX 系统IPC最古老的形式。

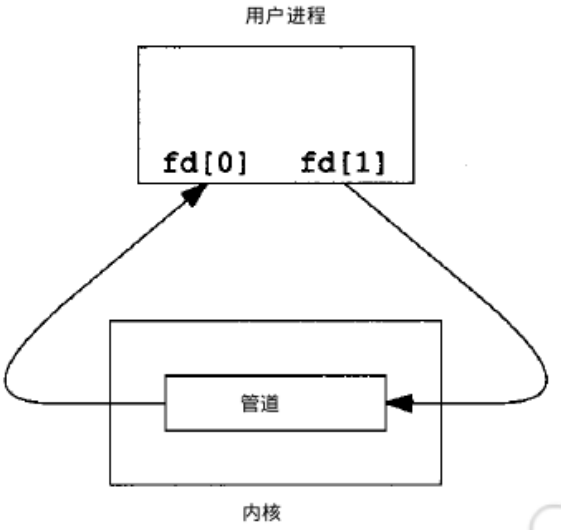
1、特点：

1. 它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端。
2. 它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）。
3. 它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

2、原型：

```
1 #include <unistd.h>
2 int pipe(int fd[2]);    // 返回值：若成功返回0，失败返回-1
```

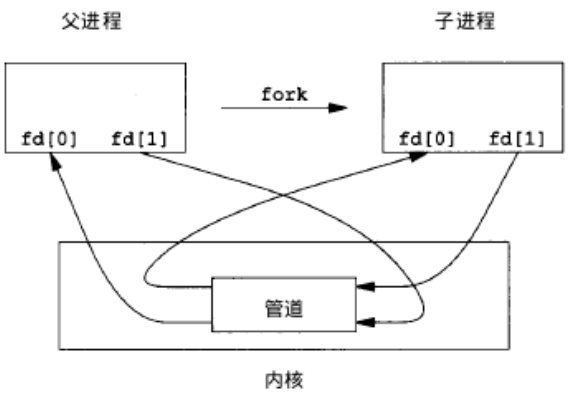
当一个管道建立时，它会创建两个文件描述符：`fd[0]`为读而打开，`fd[1]`为写而打开。如下图：



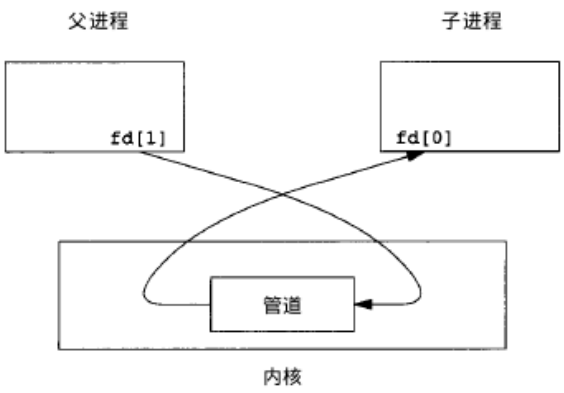
要关闭管道只需将这两个文件描述符关闭即可。

3、例子

单个进程中的管道几乎没有任何用处。所以，通常调用 pipe 的进程接着调用 fork，这样就创建了父进程与子进程之间的 IPC 通道。如下图所示：



左图 - fork之后的半双工管道



右图 - 从父进程到子进程的管道

若要数据流从父进程流向子进程，则关闭父进程的读端（`fd[0]`）与子进程的写端（`fd[1]`）；反之，则可以使数据流从子进程流向父进程。

```
#include<stdio.h>
#include<unistd.h>

int main()
{
    int fd[2];  // 两个文件描述符
    pid_t pid;
    char buff[20];

    if(pipe(fd) < 0)  // 创建管道
        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
    {
```

```
17         close(fd[0]); // 关闭读端
18         write(fd[1], "hello world\n", 12);
19     }
20     else
21     {
22         close(fd[1]); // 关闭写端
23         read(fd[0], buff, 20);
24         printf("%s", buff);
25     }
26
27     return 0;
28 }
```

二、FIFO

FIFO，也称为命名管道，它是一种文件类型。

1、特点

1. FIFO可以在无关的进程之间交换数据，与无名管道不同。

2. FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

2、原型

```
1 #include <sys/stat.h>
2 // 返回值：成功返回0，出错返回-1
3 int mkfifo(const char *pathname, mode_t mode);
```

其中的 mode 参数与open函数中的 mode 相同。一旦创建了一个 FIFO，就可以用一般的文件I/O函数操作它。

当 open 一个FIFO时，是否设置非阻塞标志（O_NONBLOCK）的区别：

- 若没有指定O_NONBLOCK（默认），只读 open 要阻塞到某个其他进程为写而打开此 FIFO。类似的，只写 open 要阻塞到某个其他进程为读而打开它。
- 若指定了O_NONBLOCK，则只读 open 立即返回。而只写 open 将出错返回 -1 如果没有进程已经为读而打开该 FIFO，其errno置ENXIO。

3、例子

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”。下面的例子演示了使用 FIFO 进行 IPC 的过程：

write_fifo.c

```
1 #include<stdio.h>
2 #include<stdlib.h>   // exit
3 #include<fcntl.h>    // O_WRONLY
4 #include<sys/stat.h>
5 #include<time.h>     // time
6
7 int main()
8 {
9     int fd;
```

```
10     int n, i;
11     char buf[1024];
12     time_t tp;
13 
14     printf("I am %d process.\n", getpid()); // 说明进程ID
15 
16     if((fd = open("fifo1", O_WRONLY)) < 0) // 以写打开一个FIFO
17     {
18         perror("Open FIFO Failed");
19         exit(1);
20     }
21 
22     for(i=0; i<10; ++i)
23     {
24         time(&tp);  // 取系统当前时间
25         n=sprintf(buf,"Process %d's time is %s",getpid(),ctime(&tp));
26         printf("Send message: %s", buf); // 打印
27         if(write(fd, buf, n+1) < 0)  // 写入到FIFO中
28         {
29             perror("Write FIFO Failed");
30             close(fd);
31             exit(1);
32         }
33         sleep(1);  // 休眠1秒
34     }
35 
36     close(fd);  // 关闭FIFO文件
37     return 0;
38 }
```

read_fifo.c

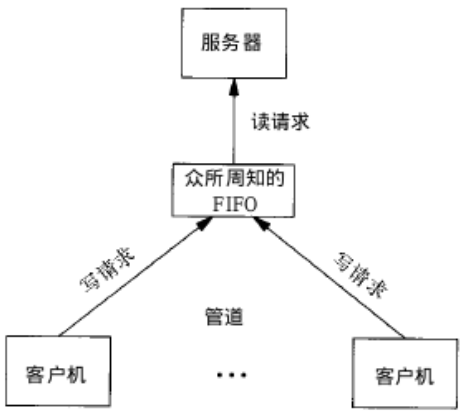
```
 1 #include<stdio.h>
 2 #include<stdlib.h>
 3 #include<errno.h>
 4 #include<fcntl.h>
 5 #include<sys/stat.h>
 6 
 7 int main()
 8 {
 9     int fd;
10     int len;
11     char buf[1024];
12 
13     if(mkfifo("fifo1", 0666) < 0 && errno!=EEXIST) // 创建FIFO管道
14         perror("Create FIFO Failed");
15 
16     if((fd = open("fifo1", O_RDONLY)) < 0)  // 以读打开FIFO
17     {
18         perror("Open FIFO Failed");
19         exit(1);
20     }
21 
22     while((len = read(fd, buf, 1024)) > 0) // 读取FIFO管道
23         printf("Read message: %s", buf);
24 
25     close(fd);  // 关闭FIFO文件
26     return 0;
27 }
```

在两个终端里用 gcc 分别编译运行上面两个文件，可以看到输出结果如下：

```
 1 [cheesezh@localhost]$ ./write_fifo
 2 I am 5954 process.
 3 Send message: Process 5954's time is Mon Apr 20 12:37:28 2015
 4 Send message: Process 5954's time is Mon Apr 20 12:37:29 2015
 5 Send message: Process 5954's time is Mon Apr 20 12:37:30 2015
 6 Send message: Process 5954's time is Mon Apr 20 12:37:31 2015
 7 Send message: Process 5954's time is Mon Apr 20 12:37:32 2015
 8 Send message: Process 5954's time is Mon Apr 20 12:37:33 2015
 9 Send message: Process 5954's time is Mon Apr 20 12:37:34 2015
10 Send message: Process 5954's time is Mon Apr 20 12:37:35 2015
11 Send message: Process 5954's time is Mon Apr 20 12:37:36 2015
12 Send message: Process 5954's time is Mon Apr 20 12:37:37 2015
```

```
 1 [cheesezh@localhost]$ ./read_fifo
 2 Read message: Process 5954's time is Mon Apr 20 12:37:28 2015
 3 Read message: Process 5954's time is Mon Apr 20 12:37:29 2015
 4 Read message: Process 5954's time is Mon Apr 20 12:37:30 2015
 5 Read message: Process 5954's time is Mon Apr 20 12:37:31 2015
 6 Read message: Process 5954's time is Mon Apr 20 12:37:32 2015
 7 Read message: Process 5954's time is Mon Apr 20 12:37:33 2015
 8 Read message: Process 5954's time is Mon Apr 20 12:37:34 2015
 9 Read message: Process 5954's time is Mon Apr 20 12:37:35 2015
10 Read message: Process 5954's time is Mon Apr 20 12:37:36 2015
11 Read message: Process 5954's time is Mon Apr 20 12:37:37 2015
```

上述例子可以扩展成 客户进程—服务器进程 通信的实例，write_fifo的作用类似于客户端，可以打开多个客户端向一个服务器发送请求信息，read_fifo类似于服务器，它适时监控着FIFO的读端，当有数据时，读出并进行处理，但是有一个关键的问题是，每一个客户端必须预先知道服务器提供的FIFO接口，下图显示了这种安排：



三、消息队列

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列ID）来标识。

1、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以及特定的优先级。
2. 消息队列独立于发送与接收进程。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。

2、原型

```
1 #include <sys/msg.h>
2 // 创建或打开消息队列：成功返回队列ID，失败返回-1
3 int msgget(key_t key, int flag);
4 // 添加消息：成功返回0，失败返回-1
5 int msgsnd(int msqid, const void *ptr, size_t size, int flag);
6 // 读取消息：成功返回消息数据的长度，失败返回-1
7 int msgrcv(int msqid, void *ptr, size_t size, long type,int flag);
8 // 控制消息队列：成功返回0，失败返回-1
9 int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

在以下两种情况下，msgget将创建一个新的消息队列：

- 如果没有与键值key相对应的消息队列，并且flag中包含了IPC_CREAT标志位。
- key参数为IPC_PRIVATE。

函数msgrcv在读取消息队列时，type参数有下面几种情况：

- type == 0，返回队列中的第一个消息；
- type > 0，返回队列中消息类型为 type 的第一个消息；
- type < 0，返回队列中消息类型值小于或等于 type 绝对值的消息，如果有多个，则取类型值最小的消息。

可以看出，type值非 0 时用于以非先进先出次序读消息。也可以把 type 看做优先级的权值。（其他的参数解释，请自行Google之）

3、例子

下面写了一个简单的使用消息队列进行IPC的例子，服务端程序一直在等待特定类型的消息，当收到该类型的消息以后，发送另一种特定类型的消息作为反馈，客户端读取该反馈并打印出来。

msg_server.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/msg.h>
 4
 5 // 用于创建一个唯一的key
 6 #define MSG_FILE "/etc/passwd"
 7
 8 // 消息结构
 9 struct msg_form {
10     long mtype;
11     char mtext[256];
12 };
13
14 int main()
15 {
16     int msqid;
17     key_t key;
18     struct msg_form msg;
19
```

```
20     // 获取key值
21     if((key = ftok(MSG_FILE,'z')) < 0)
22     {
23         perror("ftok error");
24         exit(1);
25     }
26
27     // 打印key值
28     printf("Message Queue - Server key is: %d.\n", key);
29
30     // 创建消息队列
31     if ((msqid = msgget(key, IPC_CREAT|0777)) == -1)
32     {
33         perror("msgget error");
34         exit(1);
35     }
36
37     // 打印消息队列ID及进程ID
38     printf("My msqid is: %d.\n", msqid);
39     printf("My pid is: %d.\n", getpid());
40
41     // 循环读取消息
42     for(;;)
43     {
44         msgrcv(msqid, &msg, 256, 888, 0);// 返回类型为888的第一个消息
45         printf("Server: receive msg.mtext is: %s.\n", msg.mtext);
46         printf("Server: receive msg.mtype is: %d.\n", msg.mtype);
47
48         msg.mtype = 999; // 客户端接收的消息类型
49         sprintf(msg.mtext, "hello, I'm server %d", getpid());
50         msgsnd(msqid, &msg, sizeof(msg.mtext), 0);
51     }
52     return 0;
53 }
```

msg_client.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <sys/msg.h>
 4
 5 // 用于创建一个唯一的key
 6 #define MSG_FILE "/etc/passwd"
 7
 8 // 消息结构
 9 struct msg_form {
10     long mtype;
11     char mtext[256];
12 };
13
14 int main()
15 {
16     int msqid;
17     key_t key;
18     struct msg_form msg;
19
20     // 获取key值
21     if ((key = ftok(MSG_FILE, 'z')) < 0)
22     {
23         perror("ftok error");
24         exit(1);
25     }
26
27     // 打印key值
28     printf("Message Queue - Client key is: %d.\n", key);
29
```

```
30     // 打开消息队列
31     if ((msqid = msgget(key, IPC_CREAT|0777)) == -1)
32     {
33         perror("msgget error");
34         exit(1);
35     }
36
37     // 打印消息队列ID及进程ID
38     printf("My msqid is: %d.\n", msqid);
39     printf("My pid is: %d.\n", getpid());
40
41     // 添加消息，类型为888
42     msg.mtype = 888;
43     sprintf(msg.mtext, "hello, I'm client %d", getpid());
44     msgsnd(msqid, &msg, sizeof(msg.mtext), 0);
45
46     // 读取类型为777的消息
47     msgrcv(msqid, &msg, 256, 999, 0);
48     printf("Client: receive msg.mtext is: %s.\n", msg.mtext);
49     printf("Client: receive msg.mtype is: %d.\n", msg.mtype);
50     return 0;
51 }
```

四、信号量

信号量（semaphore）与已经介绍过的 IPC 结构不同，它是一个计数器。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。

1、特点

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。

2、原型

最简单的信号量是只能取 0 和 1 的变量，这也是信号量最常见的一种形式，叫做二值信号量（Binary Semaphore）。而可以取多个正整数的信号量被称为通用信号量。

Linux 下的信号量函数都是在通用的信号量数组上进行操作，而不是在一个单一的二值信号量上进行操作。

```
1 #include <sys/sem.h>
2 // 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
3 int semget(key_t key, int num_sems, int sem_flags);
4 // 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
5 int semop(int semid, struct sembuf semoparray[], size_t numops);
6 // 控制信号量的相关信息
7 int semctl(int semid, int sem_num, int cmd, ...);
```

当semget创建新的信号量集合时，必须指定集合中信号量的个数（即num_sems），通常为1；如果是引用一个现有的集合，则将num_sems指定为 0 。

在semop函数中，sembuf结构的定义如下：

```
1 struct sembuf
2 {
3     short sem_num; // 信号量组中对应的序号，0~sem_nums-1
4     short sem_op;  // 信号量值在一次操作中的改变量
5     short sem_flg; // IPC_NOWAIT, SEM_UNDO
6 }
```

其中 sem_op 是一次操作中的信号量的改变量：

- 若sem_op > 0，表示进程释放相应的资源数，将 sem_op 的值加到信号量的值上。如果有进程正在休眠等待此信号量，则换行它们。
- 若sem_op < 0，请求 sem_op 的绝对值的资源。
    - 如果相应的资源数可以满足请求，则将该信号量的值减去sem_op的绝对值，函数成功返回。
    - 当相应的资源数不能满足请求时，这个操作与sem_flg有关。
        - sem_flg 指定IPC_NOWAIT，则semop函数出错返回EAGAIN。
        - sem_flg 没有指定IPC_NOWAIT，则将该信号量的semncnt值加1，然后进程挂起直到下述情况发生：
            1. 当相应的资源数可以满足请求，此信号量的semncnt值减1，该信号量的值减去sem_op的绝对值。成功返回；
            2. 此信号量被删除，函数smeop出错返回EIDRM；
            3. 进程捕捉到信号，并从信号处理函数返回，此情况下将此信号量的semncnt值减1，函数semop出错返回EINTR
- 若sem_op == 0，进程阻塞直到信号量的相应值为0：
    - 当信号量已经为0，函数立即返回。
    - 如果信号量的值不为0，则依据sem_flg决定函数动作：
        - sem_flg指定IPC_NOWAIT，则出错返回EAGAIN。
        - sem_flg没有指定IPC_NOWAIT，则将该信号量的semncnt值加1，然后进程挂起直到下述情况发生：
            1. 信号量值为0，将信号量的semzcnt的值减1，函数semop成功返回；
            2. 此信号量被删除，函数smeop出错返回EIDRM；
            3. 进程捕捉到信号，并从信号处理函数返回，在此情况将此信号量的semncnt值减1，函数semop出错返回EINTR

在semctl函数中的命令有多种，这里就说两个常用的：

- SETVAL：用于初始化信号量为一个已知的值。所需要的值作为联合semun的val成员来传递。在信号量第一次使用之前需要设置信号量。
- IPC_RMID：删除一个信号量集合。如果不删除信号量，它将继续在系统中存在，即使程序已经退出，它可能在你下次运行此程序时引发问题，而且信号量是一种有限的资源。

3、例子

```
1 #include<stdio.h>
2 #include<stdlib.h>
```

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

// 联合体，用于semctl初始化
union semun
{
    int              val; /*for SETVAL*/
    struct semid_ds *buf;
    unsigned short  *array;
};

// 初始化信号量
int init_sem(int sem_id, int value)
{
    union semun tmp;
    tmp.val = value;
    if(semctl(sem_id, 0, SETVAL, tmp) == -1)
    {
        perror("Init Semaphore Error");
        return -1;
    }
    return 0;
}

// P操作:
//    若信号量值为1，获取资源并将信号量值-1
//    若信号量值为0，进程挂起等待
int sem_p(int sem_id)
{
    struct sembuf sbuf;
    sbuf.sem_num = 0; /*序号*/
    sbuf.sem_op = -1; /*P操作*/
    sbuf.sem_flg = SEM_UNDO;

    if(semop(sem_id, &sbuf, 1) == -1)
    {
        perror("P operation Error");
        return -1;
    }
    return 0;
}

// V操作：
//    释放资源并将信号量值+1
//    如果有进程正在挂起等待，则唤醒它们
int sem_v(int sem_id)
{
    struct sembuf sbuf;
    sbuf.sem_num = 0; /*序号*/
    sbuf.sem_op = 1;  /*V操作*/
    sbuf.sem_flg = SEM_UNDO;

    if(semop(sem_id, &sbuf, 1) == -1)
    {
        perror("V operation Error");
        return -1;
    }
    return 0;
}

// 删除信号量集
int del_sem(int sem_id)
{
    union semun tmp;
    if(semctl(sem_id, 0, IPC_RMID, tmp) == -1)
    {
        perror("Delete Semaphore Error");
        return -1;
    }
    return 0;
}
```

```
 73
 74
 75 int main()
 76 {
 77     int sem_id;  // 信号量集ID
 78     key_t key;
 79     pid_t pid;
 80
 81     // 获取key值
 82     if((key = ftok(".", 'z')) < 0)
 83     {
 84         perror("ftok error");
 85         exit(1);
 86     }
 87
 88     // 创建信号量集，其中只有一个信号量
 89     if((sem_id = semget(key, 1, IPC_CREAT|0666)) == -1)
 90     {
 91         perror("semget error");
 92         exit(1);
 93     }
 94
 95     // 初始化：初值设为0资源被占用
 96     init_sem(sem_id, 0);
 97
 98     if((pid = fork()) == -1)
 99         perror("Fork Error");
100     else if(pid == 0) /*子进程*/
101     {
102         sleep(2);
103         printf("Process child: pid=%d\n", getpid());
104         sem_v(sem_id);  /*释放资源*/
105     }
106     else  /*父进程*/
107     {
108         sem_p(sem_id);   /*等待资源*/
109         printf("Process father: pid=%d\n", getpid());
110         sem_v(sem_id);   /*释放资源*/
111         del_sem(sem_id); /*删除信号量集*/
112     }
113     return 0;
114 }
```

上面的例子如果不加信号量，则父进程会先执行完毕。这里加了信号量让父进程等待子进程执行完以后再执行。

五、共享内存

共享内存（Shared Memory），指两个或多个进程共享一个给定的存储区。

1、特点

1. 共享内存是最快的一种 IPC，因为进程是直接对内存进行存取。
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。

2、原型

```
#include <sys/shm.h>
// 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
int shmget(key_t key, size_t size, 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
void *shmat(int shm_id, const void *addr, int flag);
// 断开与共享内存的连接：成功返回0，失败返回-1
int shmdt(void *addr);
// 控制共享内存的相关信息：成功返回0，失败返回-1
int shmctl(int shm_id, int cmd, struct shmid_ds *buf);
```

当用shmget函数创建一段共享内存时，必须指定其 size；而如果引用一个已存在的共享内存，则将 size 指定为0 。

当一段共享内存被创建以后，它并不能被任何进程访问。必须使用shmat函数连接该共享内存到当前进程的地址空间，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。

shmdt函数是用来断开shmat建立的连接的。注意，这并不是从系统中删除该共享内存，只是当前进程不能再访问该共享内存而已。

shmctl函数可以对共享内存执行多种操作，根据参数 cmd 执行相应的操作。常用的是 IPC_RMID（从系统中删除该共享内存）。

3、例子

下面这个例子，使用了【共享内存+信号量+消息队列】的组合来实现服务器进程与客户进程间的通信。

- 共享内存用来传递数据；
- 信号量用来同步；
- 消息队列用来 在客户端修改了共享内存后 通知服务器读取。

server.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/shm.h>  // shared memory
#include<sys/sem.h>  // semaphore
#include<sys/msg.h>  // message queue
#include<string.h>   // memcpy

// 消息队列结构
struct msg_form {
    long mtype;
    char mtext;
};

// 联合体，用于semctl初始化
union semun
{
    int              val; /*for SETVAL*/
    struct semid_ds *buf;
    unsigned short  *array;
};

// 初始化信号量
int init_sem(int sem_id, int value)
{
    union semun tmp;
```

```
26     tmp.val = value;
27     if(semctl(sem_id, 0, SETVAL, tmp) == -1)
28     {
29         perror("Init Semaphore Error");
30         return -1;
31     }
32     return 0;
33 }
34
35 // P操作:
36 //  若信号量值为1，获取资源并将信号量值-1
37 //  若信号量值为0，进程挂起等待
38 int sem_p(int sem_id)
39 {
40     struct sembuf sbuf;
41     sbuf.sem_num = 0; /*序号*/
42     sbuf.sem_op = -1; /*P操作*/
43     sbuf.sem_flg = SEM_UNDO;
44
45     if(semop(sem_id, &sbuf, 1) == -1)
46     {
47         perror("P operation Error");
48         return -1;
49     }
50     return 0;
51 }
52
53 // V操作：
54 //  释放资源并将信号量值+1
55 //  如果有进程正在挂起等待，则唤醒它们
56 int sem_v(int sem_id)
57 {
58     struct sembuf sbuf;
59     sbuf.sem_num = 0; /*序号*/
60     sbuf.sem_op = 1;  /*V操作*/
61     sbuf.sem_flg = SEM_UNDO;
62
63     if(semop(sem_id, &sbuf, 1) == -1)
64     {
65         perror("V operation Error");
66         return -1;
67     }
68     return 0;
69 }
70
71 // 删除信号量集
72 int del_sem(int sem_id)
73 {
74     union semun tmp;
75     if(semctl(sem_id, 0, IPC_RMID, tmp) == -1)
76     {
77         perror("Delete Semaphore Error");
78         return -1;
79     }
80     return 0;
81 }
82
83 // 创建一个信号量集
84 int creat_sem(key_t key)
85 {
86     int sem_id;
87     if((sem_id = semget(key, 1, IPC_CREAT|0666)) == -1)
88     {
89         perror("semget error");
90         exit(-1);
91     }
92     init_sem(sem_id, 1);  /*初值设为1资源未占用*/
93     return sem_id;
94 }
95
```

```
 96
 97 int main()
 98 {
 99     key_t key;
100     int shmid, semid, msqid;
101     char *shm;
102     char data[] = "this is server";
103     struct shmid_ds buf1;  /*用于删除共享内存*/
104     struct msqid_ds buf2;  /*用于删除消息队列*/
105     struct msg_form msg;  /*消息队列用于通知对方更新了共享内存*/
106
107     // 获取key值
108     if((key = ftok(".", 'z')) < 0)
109     {
110         perror("ftok error");
111         exit(1);
112     }
113
114     // 创建共享内存
115     if((shmid = shmget(key, 1024, IPC_CREAT|0666)) == -1)
116     {
117         perror("Create Shared Memory Error");
118         exit(1);
119     }
120
121     // 连接共享内存
122     shm = (char*)shmat(shmid, 0, 0);
123     if((int)shm == -1)
124     {
125         perror("Attach Shared Memory Error");
126         exit(1);
127     }
128
129
130     // 创建消息队列
131     if ((msqid = msgget(key, IPC_CREAT|0777)) == -1)
132     {
133         perror("msgget error");
134         exit(1);
135     }
136
137     // 创建信号量
138     semid = creat_sem(key);
139
140     // 读数据
141     while(1)
142     {
143         msgrcv(msqid, &msg, 1, 888, 0); /*读取类型为888的消息*/
144         if(msg.mtext == 'q')  /*quit - 跳出循环*/
145             break;
146         if(msg.mtext == 'r')  /*read - 读共享内存*/
147         {
148             sem_p(semid);
149             printf("%s\n",shm);
150             sem_v(semid);
151         }
152     }
153
154     // 断开连接
155     shmdt(shm);
156
157     /*删除共享内存、消息队列、信号量*/
158     shmctl(shmid, IPC_RMID, &buf1);
159     msgctl(msqid, IPC_RMID, &buf2);
160     del_sem(semid);
161     return 0;
162 }
```

client.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/shm.h>  // shared memory
#include<sys/sem.h>  // semaphore
#include<sys/msg.h>  // message queue
#include<string.h>   // memcpy

// 消息队列结构
struct msg_form {
    long mtype;
    char mtext;
};

// 联合体，用于semctl初始化
union semun
{
    int              val; /*for SETVAL*/
    struct semid_ds *buf;
    unsigned short  *array;
};

// P操作:
//  若信号量值为1，获取资源并将信号量值-1
//  若信号量值为0，进程挂起等待
int sem_p(int sem_id)
{
    struct sembuf sbuf;
    sbuf.sem_num = 0; /*序号*/
    sbuf.sem_op = -1; /*P操作*/
    sbuf.sem_flg = SEM_UNDO;

    if(semop(sem_id, &sbuf, 1) == -1)
    {
        perror("P operation Error");
        return -1;
    }
    return 0;
}

// V操作：
//  释放资源并将信号量值+1
//  如果有进程正在挂起等待，则唤醒它们
int sem_v(int sem_id)
{
    struct sembuf sbuf;
    sbuf.sem_num = 0; /*序号*/
    sbuf.sem_op = 1;  /*V操作*/
    sbuf.sem_flg = SEM_UNDO;

    if(semop(sem_id, &sbuf, 1) == -1)
    {
        perror("V operation Error");
        return -1;
    }
    return 0;
}


int main()
{
    key_t key;
    int shmid, semid, msqid;
    char *shm;
    struct msg_form msg;
    int flag = 1; /*while循环条件*/

```

```
67      // 获取key值
68      if((key = ftok(".", 'z')) < 0)
69      {
70          perror("ftok error");
71          exit(1);
72      }
73
74      // 获取共享内存
75      if((shmid = shmget(key, 1024, 0)) == -1)
76      {
77          perror("shmget error");
78          exit(1);
79      }
80
81      // 连接共享内存
82      shm = (char*)shmat(shmid, 0, 0);
83      if((int)shm == -1)
84      {
85          perror("Attach Shared Memory Error");
86          exit(1);
87      }
88
89      // 创建消息队列
90      if ((msqid = msgget(key, 0)) == -1)
91      {
92          perror("msgget error");
93          exit(1);
94      }
95
96      // 获取信号量
97      if((semid = semget(key, 0, 0)) == -1)
98      {
99          perror("semget error");
100         exit(1);
101     }
102
103     // 写数据
104     printf("****************************************\n");
105     printf("*                 IPC                  *\n");
106     printf("*    Input r to send data to server.   *\n");
107     printf("*    Input q to quit.                  *\n");
108     printf("****************************************\n");
109
110     while(flag)
111     {
112         char c;
113         printf("Please input command: ");
114         scanf("%c", &c);
115         switch(c)
116         {
117             case 'r':
118                 printf("Data to send: ");
119                 sem_p(semid);  /*访问资源*/
120                 scanf("%s", shm);
121                 sem_v(semid);  /*释放资源*/
122                 /*清空标准输入缓冲区*/
123                 while((c=getchar())!='\n' && c!=EOF);
124                 msg.mtype = 888;
125                 msg.mtext = 'r';  /*发送消息通知服务器读数据*/
126                 msgsnd(msqid, &msg, sizeof(msg.mtext), 0);
127                 break;
128             case 'q':
129                 msg.mtype = 888;
130                 msg.mtext = 'q';
131                 msgsnd(msqid, &msg, sizeof(msg.mtext), 0);
132                 flag = 0;
133                 break;
134             default:
135                 printf("Wrong input!\n");
136                 /*清空标准输入缓冲区*/
```

```
137                 while((c=getchar())!='\n' && c!=EOF);
138         }
139     }
140 
141     // 断开连接
142     shmdt(shm);
143 
144     return 0;
145 }
```

注意：当scanf()输入字符或字符串时，缓冲区中遗留下了\n，所以每次输入操作后都需要清空标准输入的缓冲区。但是由于 gcc 编译器不支持fflush(stdin)（它只是标准C的扩展），所以我们使用了替代方案：

```
1 while((c=getchar())!='\n' && c!=EOF);
```

五种通讯方式总结

1.管道：速度慢，容量有限，只有父子进程能通讯

2.FIFO：任何进程间都能通讯，但速度慢

3.消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题

4.信号量：不能传递复杂消息，只能用来同步

5.共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间通讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存