# 第19章　有关进程通信的编程

本章介绍有关Linux系统进程间通信的内容。

## 19.1　进程间通信简介

Linux 系统中的IPC (InterProcess Communication)函数提供了系统中多个进程之间相互通信的方法。对于Linux系统中的C语言程序员来说，系统中包括如下几种方式的IPC：

- 半双工UNIX 管道。
- FIFO(命名管道)。
- SYSV 风格的消息队列。
- SYSV 风格的信号量设置。
- SYSV 风格的共享内存段。
- 网络套接口 (Berkeley 风格)。
- 全双工管道 (STREAMS 管道)。

以上的这些函数，如果使用得当的话，提供了在 UNIX系统（当然也包括Linux系统）上开发客户机/服务器模式应用程序的有效工具。

## 19.2　半双工UNIX管道

### 19.2.1　基本概念

简单地说，管道就是一种连接一个进程的标准输出到另一个进程的标准输入的方法。管道是最古老的IPC工具，从UNIX系统一开始就存在。它提供了一种进程之间单向的通信方法。管道在系统中的应用很广泛，即使在shell环境中也要经常使用管道技术。

当进程创建一个管道时，系统内核设置了两个管道可以使用的文件描述符。一个用于向管道中输入信息(write)，另一个用于从管道中获取信息 (read)。在这一点上，管道没有多少的实际用途，因为创建管道的进程只能使用管道和进程自己通信。

如果进程通过管道fd0来发送数据，它可以通过fd1来读取此数据。当管道最初和进程连接时，在管道中传送数据是通过内核进行的。在 Linux系统环境下，管道最初在系统内核中都是使用索引节点表示的。当然，此索引节点保存在内核之中，而并不存在于任何的物理文件系统中。这就为我们提供了一种特别方便地使用I/O的方法，这一点我们以后将会看到。

这时，管道基本上是没有任何用的，因为进程没有必要和自己进行通信。通常的做法是进程派生出一个子进程。因为子进程将会继承父进程中所有打开的文件描述符，那么我们就可以在父进程和子进程之间通信了。

一个简单的管道创建完了。下面介绍如何使用管道。如果要直接存取管道，可以使用和低级的文件I/O同样的系统调用，因为在系统内核中管道实际上是由一个有效的索引节点表示的。

如果希望向管道中发送数据，可以使用系统调用 write()，反之，如果希望从管道中读取数据，可以使用系统调用read()。虽然大部分低级的文件I/O系统调用可以使用文件描述符，但一

些系统调用，例如lseek()不能通过文件描述符使用管道。

### 19.2.2 使用C语言创建管道

使用C语言创建管道要比在 shell下使用管道复杂一些。如果要使用 C语言创建一个简单的管道，可以使用系统调用 pipe()。它接受一个参数，也就是一个包括两个整数的数组。如果系统调用成功，此数组将包括管道使用的两个文件描述符。创建一个管道之后，一般情况下进程将产生一个新的进程。

**系统调用**：pipe()；

原型：int pipe( int fd[2] )；

返回值：如果系统调用成功，返回0

如果系统调用失败返回-1：errno = EMFILE (没有空闲的文件描述符)
EMFILE (系统文件表已满)
EFAULT (fd数组无效)

**注意** fd[0] 用于读取管道，fd[1] 用于写入管道。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
 main()
{
        int     fd[2];
        pipe(fd);
        .
        .
    }
```

一旦创建了管道，我们就可以创建一个新的子进程：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
main()
{
        int     fd[2];
        pid_t   childpid;
        pipe(fd);
        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
        .
        .
    }
```

如果父进程希望从子进程中读取数据，那么它应该关闭 fd1，同时子进程关闭 fd0。反之，如果父进程希望向子进程中发送数据，那么它应该关闭 fd0，同时子进程关闭fd1。因为文件描述符是在父进程和子进程之间共享，所以我们要及时地关闭不需要的管道的那一端。单从技术

的角度来说，如果管道的一端没有正确地关闭的话，你将无法得到一个 EOF。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
main()
{
        int     fd[2];
        pid_t   childpid;
pipe(fd);
if((childpid = fork()) == -1)
{
        perror("fork");
        exit(1);
}
if(childpid == 0)
{
        /* Child process closes up input side of pipe */
        close(fd[0]);
}
 else
{
        /* Parent process closes up output side of pipe */
        close(fd[1]);
        }
        .
        .
}
```

正如前面提到的，一但创建了管道之后，管道所使用的文件描述符就和正常文件的文件描述符一样了。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{
    int     fd[2], nbytes;
    pid_t   childpid;
    char    string[] = "Hello, world!\n";
    char    readbuffer[80];
    pipe(fd);
    if((childpid = fork()) == -1)
    {
        perror("fork");
        exit(1);
    }
    if(childpid == 0)
    {
        /* Child process closes up input side of pipe */
        close(fd[0]);
        /* Send "string" through the output side of pipe */
        write(fd[1], string, strlen(string));
        exit(0);
```

```
    }
    else
    {
        /* Parent process closes up output side of pipe */
        close(fd[1]);
        /* Read in a string from the pipe */
        nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
        printf("Received string: %s", readbuffer);
    }
    return(0);
}
```

一般情况下，子进程中的文件描述符将会复制到标准的输入和输出中。这样子进程可以使用exec()执行另一个程序，此程序继承了标准的数据流。

**系统调用：**dup();

原型： int dup( int oldfd );

返回：如果系统调用成功，返回新的文件描述符

如果系统调用失败，返回-1: errno = EBADF (oldfd 不是有效的文件描述符)

EBADF (newfd超出范围)

EMFILE (进程的文件描述符太多)

**注意** 旧文件描述符oldfd没有关闭。

虽然旧文件描述符和新创建的文件描述符可以交换使用，但一般情况下需要首先关闭一个。系统调用dup()使用的是号码最小的空闲的文件描述符。

再看下面的程序：

```
.
.
childpid = fork();
if(childpid == 0)
{
    /* Close up standard input of the child */
    close(0);
    /* Duplicate the input side of pipe to stdin */
    dup(fd[0]);
    execlp("sort", "sort", NULL);
    .
}
```

因为文件描述符0（stdin）被关闭，所以dup()把管道的输入描述符复制到它的标准输入中。这样我们可以调用execlp()，使用sort程序覆盖子进程的正文段。因为新创建的程序从它的父进程中继承了标准输入/输出流，所以它实际上继承了管道的输入端作为它的标准输入端。现在，最初的父进程送往管道的任何数据都将会直接送往sort函数。

**系统调用：** dup2();

原型：int dup2( int oldfd, int newfd );

返回值： 如果调用成功，返回新的文件描述符

如果调用失败，返回-1 ： errno = EBADF (oldfd 不是有效的文件描述符)

EBADF (newfd 超出范围)

EMFILE (进程的文件描述符太多)

**注意** dup2()将关闭旧文件描述符。

使用此系统调用，可以将close操作和文件描述符复制操作集成到一个系统调用中。另外，此系统调用保证了操作的自动进行，也就是说操作不能被其他的信号中断。这个操作将会在返回系统内核之前完成。如果使用前一个系统调用dup()，程序员不得不在此之前执行一个close()操作。

请看下面的程序：

```
.
.
childpid = fork();
if(childpid == 0)
{
    /* Close stdin, duplicate the input side of pipe to stdin */
    dup2(0, fd[0]);
    execlp("sort", "sort", NULL);
    .
    .
}
```

### 19.2.3 创建管道的简单方法

如果你认为上面创建和使用管道的方法过于繁琐的话，你也可以使用下面的简单的方法：

库函数：popen();

原型： FILE *popen ( char *command, char *type);

返回值：如果成功，返回一个新的文件流。

如果无法创建进程或者管道，返回NULL。

此标准的库函数通过在系统内部调用pipe()来创建一个半双工的管道，然后它创建一个子进程，启动shell，最后在shell上执行command参数中的命令。管道中数据流的方向是由第二个参数type控制的。此参数可以是r或者 w，分别代表读或写。但不能同时为读和写。在Linux系统下，管道将会以参数type中第一个字符代表的方式打开。所以，如果你在参数type中写入rw，管道将会以读的方式打开。

虽然此库函数的用法很简单，但也有一些不利的地方。例如它失去了使用系统调用 pipe()时可以有的对系统的控制。尽管这样，因为可以直接地使用shell命令，所以shell中的一些通配符和其他的一些扩展符号都可以在command参数中使用。

使用popen()创建的管道必须使用pclose()关闭。其实， popen/pclose和标准文件输入/输出流中的fopen() / fclose()十分相似。

库函数： pclose();

原型： int pclose( FILE *stream );

返回值： 返回系统调用wait4()的状态 。

如果stream无效，或者系统调用wait4()失败，则返回-1。

**注意** 此库函数等待管道进程运行结束，然后关闭文件流。

库函数pclose()在使用popen()创建的进程上执行wait4()函数。当它返回时，它将破坏管道和文件系统。

在下面的例子中，用sort命令打开了一个管道，然后对一个字符数组排序：

```
#include <stdio.h>
#define MAXSTRS 5
int main(void)
{
        int  cntr;
        FILE *pipe_fp;
        char *strings[MAXSTRS] = { "echo", "bravo", "alpha",
                                "charlie", "delta"};
        /* Create one way pipe line with call to popen() */
        if (( pipe_fp = popen("sort", "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }
        /* Processing loop */
        for(cntr=0; cntr<MAXSTRS; cntr++) {
                fputs(strings[cntr], pipe_fp);
                fputc('\n', pipe_fp);
        }
        /* Close the pipe */
        pclose(pipe_fp);
        return(0);
}
```

因为popen()使用shell执行命令，所以所有的shell扩展符和通配符都可以使用。此外，它还可以和popen()一起使用重定向和输出管道函数。再看下面的例子：

```
popen("ls ~scottb", "r");
popen("sort > /tmp/foo", "w");
popen("sort | uniq | more", "w");
```

下面的程序是另一个使用popen()的例子，它打开两个管道（一个用于ls命令，另一个用于sort命令）：

```
#include <stdio.h>
int main(void)
{
        FILE *pipein_fp, *pipeout_fp;
        char readbuf[80];
        /* Create one way pipe line with call to popen() */
        if (( pipein_fp = popen("ls", "r")) == NULL)
        {
                perror("popen");
                exit(1);
        }
        /* Create one way pipe line with call to popen() */
        if (( pipeout_fp = popen("sort", "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }
        /* Processing loop */
        while(fgets(readbuf, 80, pipein_fp))
```

```
            fputs(readbuf, pipeout_fp);
        /* Close the pipes */
        pclose(pipein_fp);
        pclose(pipeout_fp);
        return(0);
}
```

最后，我们再看一个使用popen()的例子。此程序用于创建一个命令和文件之间的管道：

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        FILE *pipe_fp, *infile;
        char readbuf[80];
        if( argc != 3) {
                fprintf(stderr, "USAGE:  popen3 [command] [filename]\n");
                exit(1);
        }
        /* Open up input file */
        if (( infile = fopen(argv[2], "rt")) == NULL)
        {
                perror("fopen");
                exit(1);
        }
        /* Create one way pipe line with call to popen() */
        if (( pipe_fp = popen(argv[1], "w")) == NULL)
        {
                perror("popen");
                exit(1);
        }
        /* Processing loop */
        do {
                fgets(readbuf, 80, infile);
                if(feof(infile)) break;
                fputs(readbuf, pipe_fp);
        } while(!feof(infile));
        fclose(infile);
        pclose(pipe_fp);
        return(0);
}
```

下面是使用此程序的例子：

```
popen3 sort popen3.c
popen3 cat popen3.c
popen3 more popen3.c
popen3 cat popen3.c | grep main
```

### 19.2.4　使用管道的自动操作

所谓管道的自动操作就是指此操作不能被任何事件中断，整个操作一次发生。按照 POSIX 标准，在/usr/include/posix1_lim.h中定义了管道自动操作的最大缓冲区大小：

```
#define _POSIX_PIPE_BUF        512
```

每一次从自动管道中读取的数据和写入管道的数据可以达到 512个字节。超出这个缓冲区大小的数据将可能被分割，也就是说不能自动地操作。在 Linux系统下，此自动操作大小的限

制是在linux/limits.h中定义的：

```
#define PIPE_BUF        4096
```

当涉及到多个进程时，这种自动操作就显得特别重要。

### 19.2.5 使用半双工管道时的注意事项

可以通过打开两个管道来创建一个双向的管道。但需要在子进程中正确地设置文件描述符。

必须在系统调用fork()中调用pipe()，否则子进程将不会继承文件描述符。

当使用半双工管道时，任何关联的进程都必须共享一个相关的祖先进程。因为管道存在于系统内核之中，所以任何不在创建管道的进程的祖先进程之中的进程都将无法寻址它。而在命名管道中却不是这样。

## 19.3 命名管道

### 19.3.1 基本概念

命名管道和一般的管道基本相同，但也有一些显著的不同：

- 命名管道是在文件系统中作为一个特殊的设备文件而存在的。
- 不同祖先的进程之间可以通过管道共享数据。
- 当共享管道的进程执行完所有的I/O操作以后，命名管道将继续保存在文件系统中以便以后使用。

### 19.3.2 创建FIFO

可以有几种方法创建一个命名管道。头两种方法可以使用 shell。

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

上面的两个命名执行同样的操作，但其中有一点不同。命令 mkfifo提供一个在创建之后直接改变FIFO文件存取权限的途径，而命令mknod需要调用命令chmod。

一个物理文件系统可以通过p指示器十分容易地分辨出一个FIFO文件。

```
$ ls -l MYFIFO
prw-r--r--   1 root     root           0 Dec 14 22:15 MYFIFO|
```

请注意在文件名后面的管道符号“|”。

我们可以使用系统调用mknod()来创建一个FIFO管道：

库函数： mknod();

原型：int mknod( char *pathname, mode_t mode, dev_t dev);

返回值：如果成功,返回0

如果失败，返回 -1：errno = EFAULT (无效路径名)
EACCES (无存取权限)
ENAMETOOLONG (路径名太长)
ENOENT (无效路径名)
ENOTDIR (无效路径名)

下面看一个使用C语言创建FIFO管道的例子：

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

在这个例子中，文件/tmp/MYFIFO是要创建的FIFO文件。它的存取权限是0666。存取权限也可以使用umask 修改：

```
final_umask = requested_permissions & ~original_umask
```

一个常用的使用系统调用umask()的方法就是临时地清除umask的值：

```
umask(0);
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

另外，mknod()中的第三个参数只有在创建一个设备文件时才能用到。它包括设备文件的主设备号和从设备号。

### 19.3.3 FIFO操作

FIFO上的I/O操作和正常管道上的I/O操作基本一样，只有一个主要的不同。系统调用 open 用来在物理上打开一个管道。在半双工的管道中，这是不必要的。因为管道在系统内核中，而不是在一个物理的文件系统中。在我们的例子中，我们将像使用一个文件流一样使用管道，也就是使用fopen()打开管道，使用fclose()关闭它。

请看下面的简单的服务程序进程：

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <linux/stat.h>
#define FIFO_FILE      "MYFIFO"
int main(void)
{
      FILE *fp;
      char readbuf[80];
      /* Create the FIFO if it does not exist */
      umask(0);
      mknod(FIFO_FILE, S_IFIFO|0666, 0);
      while(1)
      {
            fp = fopen(FIFO_FILE, "r");
            fgets(readbuf, 80, fp);
            printf("Received string: %s\n", readbuf);
            fclose(fp);
      }
      return(0);
}
```

因为FIFO管道缺省时有阻塞的函数，所以你可以在后台运行此程序：

```
$ fifoserver&
```

再来看一下下面的简单的客户端程序：

```
#include <stdio.h>
#include <stdlib.h>
#define FIFO_FILE      "MYFIFO"
int main(int argc, char *argv[])
{
      FILE *fp;
      if ( argc != 2 ) {
```

```
            printf("USAGE: fifoclient [string]\n");
            exit(1);
        }
        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
            perror("fopen");
            exit(1);
        }
        fputs(argv[1], fp);
        fclose(fp);
        return(0);
}
```

### 19.3.4 FIFO的阻塞

一般情况下，FIFO管道上将会有阻塞的情况发生。也就是说，如果一个 FIFO管道打开供读取的话，它将一直阻塞，直到其他的进程打开管道写入信息。这种过程反过来也一样。如果你不需要阻塞函数的话，你可以在系统调用 open()中设置O_NONBLOCK标志，这样可以取消缺省的阻塞函数。

### 19.3.5 SIGPIPE信号

最后一点是，一个管道必须既有读取进程，也要有写入进程。如果一个进程试图写入到一个没有读取进程的管道中，那么系统内核将会产生 SIGPIPE信号。当两个以上的进程同时使用管道时，这一点尤其重要。

## 19.4 System V IPC

### 19.4.1 基本概念

在UNIX的System V版本，AT&T引进了三种新形式的IPC功能（消息队列、信号量、以及共享内存）。但BSD版本的UNIX使用套接口作为主要的IPC形式。Linux系统支持这两个版本。我们先看一下System V版本的IPC功能。

1. IPC 标识符

每一个IPC目标都有一个唯一的IPC标识符。这里所指的IPC目标是指一个单独的消息队列、一个信号量集或者一个共享的内存段。系统内核使用此标识符在系统内核中指明 IPC目标。例如，如果希望存取一个共享的内存段，你唯一需要的就是给该内存段的标识符指定值。

我们所说的标识符的唯一性和所谈到的 IPC 目标的类型有关。我们假设一个标识符是12345。虽然两个消息队列不能使用这个相同的IPC标识符，但一个消息队列和一个共享内存段可以同时拥有此相同的标识符。

2. IPC 关键字

想要获得唯一的标识符，则必须使用一个 IPC关键字。客户端进程和服务器端进程必须双方都同意此关键字。这是建立一个客户机/服务器框架的第一步。

在System V IPC机制中，建立两端联系的路由方法是和IPC关键字直接相关的。

通过在应用程序中设置关键字值，每一次使用的关键字都可以是相同的。一般情况下，可以使用ftok()函数为客户端和服务器端产生关键字值。

库函数： ftok();
原型： key_t ftok ( char *pathname, char proj );
返回值： 如果成功，则返回一个新的IPC 关键字值。
如果失败，则返回-1 ，可以使用系统调用stat() 得到错误的状态。

从ftok()中返回的值是结合索引节点值，第一个参数中的文件的从设备号和第2个参数的标识符的一个字符所产生的。这并不能保证它的唯一性，但应用程序可以检测冲突，并重新产生新的关键字。

```
key_t   mykey;
mykey = ftok("/tmp/myapp", 'a');
```

在上面的例子中，目录/tmp/myapp和a一起构成了关键字。另一个例子是用当前目录：

```
key_t   mykey;
mykey = ftok(".", 'a');
```

IPC关键字，无论是如何产生的，都用于在其后的IPC系统调用中创建或者存取IPC目标。

3. ipcs 命令

命令ipcs用于读取System V IPC目标的状态。

ipcs -q: 只显示消息队列。
ipcs -s: 只显示信号量。
ipcs -m: 只显示共享内存。
ipcs --help: 其他的参数。

缺省情况下，此命令将同时显示三种的IPC目标。请看下面的ipcs命令输出的例子：

```
------ Shared Memory Segments --------
shmid    owner    perms    bytes    nattch    status

------ Semaphore Arrays --------
semid    owner    perms    nsems    status

------ Message Queues --------
msqid    owner    perms    used-bytes  messages
0        root     660      5           1
```

这里我们可以看到一个单一的消息队列，它的标识符是0。它的所有者是root，存取权限是660，也就是-rw-rw--。消息队列中有一个消息，消息的大小是5个字节。

命令ipcs是一个十分有用的工具，它提供了一种观察系统内核中存储IPC目标的机制的方法。

4. ipcrm 命令

命令ipcrm用于将IPC目标从系统内核中移走。虽然也可以通过系统调用在程序中移走 IPC目标，但在一些情况下，特别是在开发环境中，经常需要手工移走 IPC目标。它的用法十分的简单：

```
ipcrm <msg | sem | shm>  <IPC ID>
```

只需指出要移走的是消息队列(msg)、信号量集 (sem)或者共享内存段(shm)既可。你可以通过ipcs命令得到IPC ID。

### 19.4.2 消息队列基本概念

消息队列是系统内核地址空间中的一个内部的链表。消息可以按照顺序发送到队列中，也可以以几种不同的方式从队列中读取。每一个消息队列用一个唯一的 IPC标识符表示。

1. 内部和用户数据结构

了解在系统内核中的数据结构是了解System V IPC机制如何工作的最好的方法。对一些数据结构的存取，对于一些最基本的操作是十分必要的，而一些其他的数据结构则存在于一些较为低级的层次中。

2. 消息缓冲区

首先我们看一下数据结构msgbuf。此数据结构可以说是消息数据的模板。虽然此数据结构需要用户自己定义，但了解系统中有这样一个数据结构是十分重要的。在 inux/msg.h中，此数据结构是这样定义的：

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
   long mtype;       /* type of message */
   char mtext[1];    /* message text */
};
```

在数据结构msgbuf中共有两个元素：

mtype 指消息的类型，它由一个整数来代表，并且，它只能是整数。

mtext 是消息数据本身。

这种给一个给定的消息指定类型的能力，使得你能够在一个单个的队列中重复使用消息。例如，你可以指定客户端进程一个多功能的数值，它可以作为服务器端发送过来的消息的类型。服务器端本身可以使用其他的数值，而客户端可以使用此数值向服务器端发送消息。在另一种情况中，一个应用程序可以将错误消息标志为类型 1，需要的消息标志为类型2，以此类推。这样组合的可能性是无穷无尽的。

另一个问题是，不要误解消息数据元素(mtext)的名字。这个字段不但可以存储字符，还可以存储任何其他的数据类型。此字段可以说是完全任意的，因为程序员自己可以重新定义此数据结构。请看下面重新定义的例子：

```
struct my_msgbuf {
      long    mtype;        /* Message type */
      long    request_id;   /* Request identifier */
      struct  client info;  /* Client information structure */
};
```

这里的消息类型字段和前面的一样，但数据结构的其余部分则由其他的两个字段所代替，而其中的一个还是另外一个结构。这就体现了消息队列的灵活之处。内核本身并不对消息结构中的数据做任何翻译。你可以在其中发送任何信息，但确实存在一个内部给定的消息大小的限制。在Linux系统中，这是在linux/msg.h中定义的：

```
#define MSGMAX  4056   /* <= 4056 */   /* max size of message (bytes) */
```

消息的最大的长度是4056个字节，其中包括mtype，它占用4个字节的长度。

3. 内核中msg 数据结构

系统内核将每一个消息保存在数据结构 msg组成的队列中。数据结构 msg在linux/msg.h中是如下定义的：

```
/* one msg structure for each message */
struct msg {
   struct msg *msg_next;   /* next message on queue */
   long  msg_type;
   char *msg_spot;         /* message text address */
   short msg_ts;           /* message text size */
```

```
};
```

各个字段的意义如下：

msg_next 是指向队列中下一个消息的指针。

msg_type 是消息类型，和用户自己在msgbuf结构中指定的消息类型相同。

msg_spot 指向消息体开始的指针。

msg_ts 是消息体的长度。

4. 内核 msqid_ds 数据结构

三种IPC目标都有自己的由系统内核维护的内部数据结构。对于消息队列来说，这个数据结构是msqid_ds。系统内核为系统中创建的每一个消息队列创建、存储和维护一个此数据结构的实例。这是在linux/msg.h中定义的：

```
/* one msqid structure for each queue on the system */
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;  /* first message on queue */
    struct msg *msg_last;   /* last message in queue */
    time_t msg_stime;       /* last msgsnd time */
    time_t msg_rtime;       /* last msgrcv time */
    time_t msg_ctime;       /* last change time */
    struct wait_queue *wwait;
    struct wait_queue *rwait;
    ushort msg_cbytes;
    ushort msg_qnum;
    ushort msg_qbytes;      /* max number of bytes on queue */
    ushort msg_lspid;       /* pid of last msgsnd */
    ushort msg_lrpid;       /* last receive pid */
};
```

虽然你可能很少需要了解此数据结构中的大部分元素，但一个简要的了解还是很有必要的：

msg_perm 是数据结构ipc_perm的一个实例，而数据结构ipc_perm是在linux/ipc.h中定义的。它保存的是消息队列的存取权限的信息，和其他的例如队列的创建者等信息。

msg_first 指向队列中的第一个消息。

msg_last 指向队列中的最后一个消息。

msg_stime 发送到队列中的最后一条消息的时间。

msg_rtime 从队列中读取的最后一条消息的时间。

msg_ctime 是队列最后一次改动的时间。

wwait 和rwait 指向内核中等待队列的指针。当一个在消息队列中的操作认为进程需要进入到睡眠状态时使用。

msg_cbytes 是队列中所有消息的总长度。

msg_qnum 是当前在队列中的消息的数目。

msg_qbytes 是队列中的最大的字节数。

msg_lspid 为发送最后一条消息的进程的PID。

msg_lrpid 为最后一个读取队列中的消息的进程的PID。

5. 内核ipc_perm 数据结构

系统内核将IPC目标的权限的信息存储在数据结构 ipc_perm中。例如，在上面讨论的数据

结构中，元素msg_perm就是此类型的数据结构，它是在linux/ipc.h中定义的：

```
struct ipc_perm
{
  key_t  key;
  ushort uid;   /* owner euid and egid */
  ushort gid;
  ushort cuid;  /* creator euid and egid */
  ushort cgid;
  ushort mode;  /* access modes see mode flags below */
  ushort seq;   /* slot usage sequence number */
};
```

### 19.4.3 系统调用msgget()

如果希望创建一个新的消息队列，或者希望存取一个已经存在的消息队列，你可以使用系统调用msgget()。

系统调用： msgget();

原型： int msgget ( key_t key, int msgflg );

返回值：如果成功，返回消息队列标识符

如果失败，则返回-1：errno =EACCESS (权限不允许)

EEXIST (队列已经存在，无法创建)

EIDRM (队列标志为删除)

ENOENT (队列不存在)

ENOMEM (创建队列时内存不够)

ENOSPC (超出最大队列限制)

系统调用msgget()中的第一个参数是关键字值（通常是由 ftok()返回的）。然后此关键字值将会和其他已经存在于系统内核中的关键字值比较。这时，打开和存取操作是和参数 msgflg中的内容相关的。

IPC_CREAT 如果内核中没有此队列，则创建它。

IPC_EXCL 当和IPC_CREAT一起使用时，如果队列已经存在，则失败。

如果单独使用IPC_CREAT，则msgget()要么返回一个新创建的消息队列的标识符，要么返回具有相同关键字值的队列的标识符。如果 IPC_EXCL 和 IPC_CREAT一起使用，则msgget()要么创建一个新的消息队列，要么如果队列已经存在则返回一个失败值 -1。IPC_EXCL单独使用是没有用处的。

下面看一个打开和创建一个消息队列的例子：

```
int open_queue( key_t keyval )
{
    int     qid;
    if((qid = msgget( keyval, IPC_CREAT | 0660 )) == -1)
    {
        return(-1);
    }
    return(qid);
}
```

### 19.4.4 系统调用msgsnd()

一旦我们得到了队列标识符，我们就可以在队列上执行我们希望的操作了。如果想要往队列中发送一条消息，你可以使用系统调用 msgsnd（）：

系统调用： msgsnd();

原型： int msgsnd ( int msqid, struct msgbuf *msgp, int msgsz, int msgflg );

返回值：如果成功， 0。

如果失败， -1 ：errno = EAGAIN (队列已满，并且使用了IPC_NOWAIT)
EACCES (没有写的权限)
EFAULT (msgp 地址无效)
EIDRM （消息队列已经删除)
EINTR (当等待写操作时，收到一个信号)
EINVAL (无效的消息队列标识符，非正数的消息类型，或者无效的消息长度)
ENOMEM (没有足够的内存复制消息缓冲区)

系统调用msgsnd()的第一个参数是消息队列标识符，它是由系统调用 msgget返回的。第二个参数是msgp，是指向消息缓冲区的指针。参数 msgsz中包含的是消息的字节大小，但不包括消息类型的长度（4个字节）。

参数msgflg可以设置为0（此时为忽略此参数），或者使用 IPC_NOWAIT。

如果消息队列已满，那么此消息则不会写入到消息队列中，控制将返回到调用进程中。如果没有指明，调用进程将会挂起，直到消息可以写入到队列中。

下面是一个发送消息的程序：

```
int send_message( int qid, struct mymsgbuf *qbuf )
{
        int     result, length;
        /* The length is essentially the size of the structure minus sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);
        if((result = msgsnd( qid, qbuf, length, 0)) == -1)
        {
                return(-1);
        }
        return(result);
}
```

这个小程序试图将存储在缓冲区qbuf中的消息发送到消息队列qid中。下面的程序是结合了上面两个程序的一个完整程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/ipc.h>
#include <linux/msg.h>
main()
{
        int    qid;
        key_t  msgkey;
        struct mymsgbuf {
                long    mtype;          /* Message type */
```

```
        int     request;        /* Work request number */
        double  salary;         /* Employee's salary */
    } msg;
    /* Generate our IPC key value */
    msgkey = ftok(".", 'm');
    /* Open/create the queue */
    if(( qid = open_queue( msgkey)) == -1) {
        perror("open_queue");
        exit(1);
    }
    /* Load up the message with arbitrary test data */
    msg.mtype   = 1;        /* Message type must be a positive number! */
    msg.request = 1;        /* Data element #1 */
    msg.salary  = 1000.00;  /* Data element #2 (my yearly salary!) */
    /* Bombs away! */
    if((send_message( qid, &msg )) == -1) {
        perror("send_message");
        exit(1);
    }
}
```

在创建和打开消息队列以后，我们将测试数据装入到消息缓冲区中。最后调用 send_messag 把消息发送到消息队列中。

现在在消息队列中有了一条消息，我们可以使用 ipcs命令来查看队列的状态。下面讨论如何从队列中获取消息。可以使用系统调用 msgrcv()：

系统调用： msgrcv();

原型： int msgrcv ( int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg );

返回值：如果成功，则返回复制到消息缓冲区的字节数。

如果失败，则返回-1： errno = E2BIG (消息的长度大于msgsz,没有MSG_NOERROR)

EACCES (没有读的权限)

EFAULT (msgp 指向的地址是无效的)

EIDRM (队列已经被删除)

EINTR (被信号中断)

EINVAL (msgqid 无效, 或者msgsz 小于0)

ENOMSG (使用IPC_NOWAIT, 同时队列中的消息无法满足要求)

很明显，第一个参数用来指定将要读取消息的队列。第二个参数代表要存储消息的消息缓冲区的地址。第三个参数是消息缓冲区的长度，不包括 mtype的长度，它可以按照如下的方法计算：

```
msgsz = sizeof(struct mymsgbuf) - sizeof(long);
```

第四个参数是要从消息队列中读取的消息的类型。如果此参数的值为 0，那么队列中最长时间的一条消息将返回，而不论其类型是什么。

如果调用中使用了 IPC_NOWAIT作为标志，那么当没有数据可以使用时，调用将把 ENOMSG返回到调用进程中。否则，调用进程将会挂起，直到队列中的一条消息满足 msgrcv() 的参数要求。如果当客户端等待一条消息的时候队列为空，将会返回 EIDRM。如果进程在等

待消息的过程中捕捉到一个信号，则返回EINTR。

下面就是一个从队列中读取消息的程序：

```
int read_message( int qid, long type, struct mymsgbuf *qbuf )
{
        int     result, length;
        /* The length is essentially the size of the structure minus sizeof(mtype) */
        length = sizeof(struct mymsgbuf) - sizeof(long);
        if((result = msgrcv( qid, qbuf, length, type,  0)) == -1)
        {
                return(-1);
        }
        return(result);
}
```

在成功地读取了一条消息以后，队列中的这条消息的入口将被删除。

参数msgflg中的MSG_NOERROR位提供一种额外的用途。如果消息的实际长度大于msgsz，同时使用了MSG_NOERROR，那么消息将会被截断，只有与msgsz长度相等的消息返回。一般情况下，系统调用msgrcv()会返回-1，而这条消息将会继续保存在队列中。我们可以利用这个特点编制一个程序，利用这个程序可以查看消息队列的情况，看看符合我们条件的消息是否已经到来：

```
int peek_message( int qid, long type )
{
        int     result, length;
        if((result = msgrcv( qid, NULL, 0, type,  IPC_NOWAIT)) == -1)
        {
                if(errno == E2BIG)
                        return(TRUE);
        }
        return(FALSE);
}
```

在上面的程序中，我们忽略了缓冲区的地址和长度。这样，系统调用将会失败。尽管如此，我们可以检查返回的E2BIG值，它说明符合条件的消息确实存在。

### 19.4.5 系统调用msgctl()

下面我们继续讨论如何使用一个给定的消息队列的内部数据结构。我们可以使用系统调用msgctl()来控制对消息队列的操作。

系统调用： msgctl();

调用原型： int msgctl ( int msgqid, int cmd, struct msqid_ds *buf );

返回值： 0，如果成功。

-1，如果失败：errno = EACCES (没有读的权限同时cmd 是IPC_STAT)

EFAULT (buf 指向的地址无效)

EIDRM (在读取中队列被删除)

EINVAL (msgqid无效, 或者msgsz 小于0)

EPERM (IPC_SET或者IPC_RMID 命令被使用，但调用程序没有写的权限)

下面我们看一下可以使用的几个命令：

IPC_STAT

读取消息队列的数据结构msqid_ds，并将其存储在buf指定的地址中。

IPC_SET

设置消息队列的数据结构msqid_ds中的ipc_perm元素的值。这个值取自buf参数。

IPC_RMID

从系统内核中移走消息队列。

我们在前面讨论过了消息队列的数据结构 (msqid_ds)。系统内核中为系统中的每一个消息队列保存一个此数据结构的实例。通过使用 IPC_STAT命令，我们可以得到一个此数据结构的副本。下面的程序就是实现此函数的过程：

```
int get_queue_ds( int qid, struct msgqid_ds *qbuf )
{
    if( msgctl( qid, IPC_STAT, qbuf) == -1)
    {
        return(-1);
    }
    return(0);
}
```

如果不能复制内部缓冲区，调用进程将返回 -1。如果调用成功，则返回 0。缓冲区中应该包括消息队列中的数据结构。

消息队列中的数据结构中唯一可以改动的元素就是 ipc_perm。它包括队列的存取权限和关于队列创建者和拥有者的信息。你可以改变用户的 id、用户的组id以及消息队列的存取权限。下面是一个修改队列存取模式的程序：

```
int change_queue_mode( int qid, char *mode )
{
    struct msqid_ds tmpbuf;
    /* Retrieve a current copy of the internal data structure */
    get_queue_ds( qid, &tmpbuf);
    /* Change the permissions using an old trick */
    sscanf(mode, "%ho", &tmpbuf.msg_perm.mode);
    /* Update the internal data structure */
    if( msgctl( qid, IPC_SET, &tmpbuf) == -1)
    {
        return(-1);
    }
    return(0);
}
```

我们通过调用get_queue_ds来读取队列的内部数据结构。然后，我们调用 sscanf()修改数据结构msg_perm中的mode 成员的值。但直到调用msgctl（）时，权限的改变才真正完成。在这里msgctl（）使用的是IPC_SET命令。

最后，我们使用系统调用msgctl()中的IPC_RMID命令删除消息队列：

```
int remove_queue( int qid )
{
    if( msgctl( qid, IPC_RMID, 0) == -1)
    {
        return(-1);
```

```
        }
        return(0);
}
```

### 19.4.6　一个msgtool的实例

程序msgtool使用命令行参数来决定它要执行的操作。它的函数包括创建、发送、读取、改变权限以及删除消息队列。它的用法如下：

1. 发送消息

```
msgtool s (type) "text"
```

2. 读取消息

```
msgtool r (type)
```

3. 改变权限

```
msgtool m (mode)
```

4. 删除队列

```
msgtool d
```

下面是使用的例子：

```
msgtool  s   1 test
msgtool  s   5 test
msgtool  s   1 "This is a test"
msgtool  r   1
msgtool  d
msgtool  m   660
```

下面是msgtool的源代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX_SEND_SIZE 80
struct mymsgbuf {
        long mtype;
        char mtext[MAX_SEND_SIZE];
};
void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);
int main(int argc, char *argv[])
{
        key_t key;
        int   msgqueue_id;
        struct mymsgbuf qbuf;
        if(argc == 1)
                usage();
        /* Create unique key via call to ftok() */
```

```
        key = ftok(".", 'm');
        /* Open the queue - create if necessary */
        if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1) {
                perror("msgget");
                exit(1);
        }
        switch(tolower(argv[1][0]))
        {
                case 's': send_message(msgqueue_id, (struct mymsgbuf *)&qbuf,
                                atol(argv[2]), argv[3]);
                        break;
                case 'r': read_message(msgqueue_id, &qbuf, atol(argv[2]));
                        break;
                case 'd': remove_queue(msgqueue_id);
                        break;
                case 'm': change_queue_mode(msgqueue_id, argv[2]);
                        break;
                 default: usage();
        }

        return(0);
}
void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
        /* Send a message to the queue */
        printf("Sending a message ...\n");
        qbuf->mtype = type;
        strcpy(qbuf->mtext, text);
        if((msgsnd(qid, (struct msgbuf *)qbuf,
                strlen(qbuf->mtext)+1, 0)) ==-1)
        {
                perror("msgsnd");
                exit(1);
        }
}
void read_message(int qid, struct mymsgbuf *qbuf, long type)
{
        /* Read a message from the queue */
        printf("Reading a message ...\n");
        qbuf->mtype = type;
        msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);
        printf("Type: %ld Text: %s\n", qbuf->mtype, qbuf->mtext);
}
void remove_queue(int qid)
{
        /* Remove the queue */
        msgctl(qid, IPC_RMID, 0);
}
void change_queue_mode(int qid, char *mode)
```

```
{
        struct msqid_ds myqueue_ds;
        /* Get current info */
        msgctl(qid, IPC_STAT, &myqueue_ds);
        /* Convert and load the mode */
        sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);
        /* Update the mode */
        msgctl(qid, IPC_SET, &myqueue_ds);
}
void usage(void)
{
        fprintf(stderr, "msgtool - A utility for tinkering with msg queues\n");
        fprintf(stderr, "\nUSAGE: msgtool (s)end <type> <messagetext>\n");
        fprintf(stderr, "              (r)ecv <type>\n");
        fprintf(stderr, "              (d)elete\n");
        fprintf(stderr, "              (m)ode <octal mode>\n");
        exit(1);
}
```

## 19.5　使用信号量编程

### 19.5.1　基本概念

信号量是一个可以用来控制多个进程存取共享资源的计数器。它经常作为一种锁定机制来防止当一个进程正在存取共享资源时，另一个进程也存取同一资源。下面先简要地介绍一下信号量中涉及到的数据结构。

1. 内核中的数据结构semid_ds

和消息队列一样，系统内核为内核地址空间中的每一个信号量集都保存了一个内部的数据结构。数据结构的原型是semid_ds。它是在linux/sem.h中做如下定义的：

```
/* One semid data structure for each set of semaphores in the system. */
struct semid_ds {
        struct ipc_perm sem_perm;       /* permissions .. see ipc.h */
        time_t          sem_otime;      /* last semop time */
        time_t          sem_ctime;      /* last change time */
        struct sem      *sem_base;      /* ptr to first semaphore in array */
        struct wait_queue *eventn;
        struct wait_queue *eventz;
        struct sem_undo  *undo;         /* undo requests on this array */
        ushort          sem_nsems;      /* no. of semaphores in array */
};
```

sem_perm　是在linux/ipc.h定义的数据结构ipc_perm的一个实例。它保存有信号量集的存取权限的信息，以及信号量集创建者的有关信息。

sem_otime　最后一次semop() 操作的时间。

sem_ctime　最后一次改动此数据结构的时间。

sem_base　指向数组中第一个信号量的指针。

sem_undo　数组中没有完成的请求的个数。

sem_nsems 信号量集（数组）中的信号量的个数。

2. 内核中的数据结构sem

在数据结构semid_ds中包含一个指向信号量数组的指针。此数组中的每一个元素都是一个数据结构sem。它也是在linux/sem.h中定义的：

```
/* One semaphore structure for each semaphore in the system. */
struct sem {
        short   sempid;       /* pid of last operation */
        ushort  semval;       /* current value */
        ushort  semncnt;      /* num procs awaiting increase in semval */
        ushort  semzcnt;      /* num procs awaiting semval = 0 */
};
```

sem_pid 最后一个操作的PID（进程ID）。

sem_semval 信号量的当前值。

sem_semncnt 等待资源的进程数目。

sem_semzcnt 等待资源完全空闲的进程数目。

### 19.5.2 系统调用semget()

我们可以使用系统调用 semget()创建一个新的信号量集，或者存取一个已经存在的信号量集：

系统调用：semget();

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

返回值：如果成功，则返回信号量集的IPC 标识符。

如果失败，则返回-1： errno = EACCESS (没有权限)

EEXIST (信号量集已经存在，无法创建)

EIDRM (信号量集已经删除)

ENOENT (信号量集不存在，同时没有使用IPC_CREAT )

ENOMEM (没有足够的内存创建新的信号量集)

ENOSPC (超出限制)

系统调用semget()的第一个参数是关键字值（一般是由系统调用 ftok()返回的）。系统内核将此值和系统中存在的其他的信号量集的关键字值进行比较。打开和存取操作与参数 semflg中的内容相关。

IPC_CREAT 如果信号量集在系统内核中不存在，则创建信号量集。

IPC_EXCL 当和IPC_CREAT 一同使用时，如果信号量集已经存在，则调用失败。

如果单独使用 IPC_CREAT ，则semget()要么返回新创建的信号量集的标识符，要么返回系统中已经存在的同样的关键字值的信号量的标识符。如果 IPC_EXCL和IPC_CREAT一同使用，则要么返回新创建的信号量集的标识符，要么返回 -1。IPC_EXCL 单独使用没有意义。

参数nsems指出了一个新的信号量集中应该创建的信号量的个数。信号量集中最多的信号量的个数是在linux/sem.h中定义的：

```
#define SEMMSL  32      /* <=512 max num of semaphores per id */
```

下面是一个打开和创建信号量集的程序：

```
int open_semaphore_set( key_t keyval, int numsems )
{
```

```
    int     sid;

    if ( ! numsems )
        return(-1);

    if((sid = semget( mykey, numsems, IPC_CREAT | 0660 )) == -1)
    {
        return(-1);
    }

    return(sid);
}
```

### 19.5.3 系统调用semop()

系统调用：semop();
调用原型： int semop ( int semid, struct sembuf *sops, unsigned nsops);
返回值： 0，如果成功。
-1，如果失败： errno = E2BIG (nsops 大于最大的ops 数目)
EACCESS (权限不够)
EAGAIN (使用了IPC_NOWAIT，但操作不能继续进行)
EFAULT (sops 指向的地址无效)
EIDRM (信号量集已经删除)
EINTR (当睡眠时接收到其他信号)
EINVAL (信号量集不存在, 或者semid 无效)
ENOMEM (使用了SEM_UNDO ,但无足够的内存创建所需的数据结构)
ERANGE (信号量值超出范围)

第一个参数是关键字值。第二个参数是指向将要操作的数组的指针。第三个参数是数组中的操作的个数。参数sops指向由sembuf组成的数组。此数组是在linux/sem.h中定义的：

```
/* semop system call takes an array of these */
struct sembuf {
    ushort  sem_num;      /* semaphore index in array */
    short   sem_op;       /* semaphore operation */
    short   sem_flg;      /* operation flags */
};
```

sem_num 将要处理的信号量的个数。
sem_op 要执行的操作。
sem_flg 操作标志。

如果sem_op是负数，那么信号量将减去它的值。这和信号量控制的资源有关。如果没有使用IPC_NOWAIT，那么调用进程将进入睡眠状态，直到信号量控制的资源可以使用为止。

如果sem_op是正数，则信号量加上它的值。这也就是进程释放信号量控制的资源。

最后，如果sem_op是0，那么调用进程将调用sleep()，直到信号量的值为0。这在一个进程等待完全空闲的资源时使用。

### 19.5.4 系统调用semctl()

系统调用：semctl();
原型： int semctl ( int semid, int semnum, int cmd, union semun arg );
返回值：如果成功，则为一个正数。
如果失败，则为-1：errno = EACCESS (权限不够)
EFAULT (arg 指向的地址无效)
EIDRM (信号量集已经删除)
EINVAL (信号量集不存在，或者semid 无效)
EPERM (EUID 没有cmd 的权利)
ERANGE (信号量值超出范围)

系统调用semctl 用来执行在信号量集上的控制操作。这和在消息队列中的系统调用 msgctl 是十分相似的。但这两个系统调用的参数略有不同。因为信号量一般是作为一个信号量集使用的，而不是一个单独的信号量。所以在信号量集的操作中，不但要知道 IPC关键字值，也要知道信号量集中的具体的信号量。

这两个系统调用都使用了参数 cmd，它用来指出要操作的具体命令。两个系统调用中的最后一个参数也不一样。在系统调用 msgctl中，最后一个参数是指向内核中使用的数据结构的指针。我们使用此数据结构来取得有关消息队列的一些信息，以及设置或者改变队列的存取权限和使用者。但在信号量中支持额外的可选的命令，这样就要求有一个更为复杂的数据结构。

系统调用semctl()的第一个参数是关键字值。第二个参数是信号量数目。

参数cmd中可以使用的命令如下：

IPC_STAT 读取一个信号量集的数据结构semid_ds，并将其存储在semun中的buf参数中。

IPC_SET 设置信号量集的数据结构semid_ds中的元素ipc_perm，其值取自semun中的buf参数。

IPC_RMID 将信号量集从内存中删除。

GETALL 用于读取信号量集中的所有信号量的值。

GETNCNT 返回正在等待资源的进程数目。

GETPID 返回最后一个执行semop操作的进程的PID。

GETVAL 返回信号量集中的一个单个的信号量的值。

GETZCNT 返回这在等待完全空闲的资源的进程数目。

SETALL 设置信号量集中的所有的信号量的值。

SETVAL 设置信号量集中的一个单独的信号量的值。

参数arg代表一个semun的实例。semun是在linux/sem.h中定义的：

```
/* arg for semctl system calls. */
union semun {
    int val;              /* value for SETVAL */
    struct semid_ds *buf;   /* buffer for IPC_STAT & IPC_SET */
    ushort *array;        /* array for GETALL & SETALL */
    struct seminfo *__buf;  /* buffer for IPC_INFO */
    void *__pad;
};
```

val 当执行SETVAL命令时使用。

buf 在IPC_STAT/IPC_SET命令中使用。代表了内核中使用的信号量的数据结构。

array 在使用GETALL/SETALL命令时使用的指针。

下面的程序返回信号量的值。当使用GETVAL命令时，调用中的最后一个参数被忽略：

```
int get_sem_val( int sid, int semnum )
{
    return( semctl(sid, semnum, GETVAL, 0));
}
```

下面是一个实际应用的例子：

```
    #define MAX_PRINTERS 5

    printer_usage()
    {
        int x;

        for(x=0; x<MAX_PRINTERS; x++)
            printf("Printer %d: %d\n\r", x, get_sem_val( sid, x ));
    }
```

下面的程序可以用来初始化一个新的信号量值：

```
void init_semaphore( int sid, int semnum, int initval)
{
    union semun semopts;

    semopts.val = initval;
    semctl( sid, semnum, SETVAL, semopts);
}
```

**注意** 系统调用semctl中的最后一个参数是一个联合类型的副本，而不是一个指向联合类型的指针。

### 19.5.5 使用信号量集的实例：semtool

你可以使用semtool程序在shell中创建、使用、控制和删除一个信号量集。

1) 创建一个信号量集

```
semtool c (number of semaphores in set)
```

2) 锁定一个信号量

```
semtool l (semaphore number to lock)
```

3) 解锁一个信号量

```
semtool u (semaphore number to unlock)
```

4) 改变信号量的权限

```
semtool m (mode)
```

5) 删除一个信号量

```
semtool d
```

下面是一些使用的实例：

```
semtool  c 5
semtool  l
semtool  u
semtool  m 660
semtool  d
```

最后是此程序的完整的源代码：

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define SEM_RESOURCE_MAX        1       /* Initial value of all semaphores */
void opensem(int *sid, key_t key);
void createsem(int *sid, key_t key, int members);
void locksem(int sid, int member);
void unlocksem(int sid, int member);
void removesem(int sid);
unsigned short get_member_count(int sid);
int getval(int sid, int member);
void dispval(int sid, int member);
void changemode(int sid, char *mode);
void usage(void);
int main(int argc, char *argv[])
{
        key_t key;
        int   semset_id;
        if(argc == 1)
                usage();
        /* Create unique key via call to ftok() */
        key = ftok(".", 's');
        switch(tolower(argv[1][0]))
        {
                case 'c': if(argc != 3)
                                usage();
                          createsem(&semset_id, key,  atoi(argv[2]));
                          break;
                case 'l': if(argc != 3)
                                usage();
                          opensem(&semset_id, key);
                          locksem(semset_id, atoi(argv[2]));
                          break;
                case 'u': if(argc != 3)
                                usage();
                          opensem(&semset_id, key);
                          unlocksem(semset_id, atoi(argv[2]));
                          break;
                case 'd': opensem(&semset_id, key);
                          removesem(semset_id);
                          break;
                case 'm': opensem(&semset_id, key);
                          changemode(semset_id, argv[2]);
                          break;
                 default: usage();

        }
```

```
        return(0);
}

void opensem(int *sid, key_t key)
{
        /* Open the semaphore set - do not create! */

        if((*sid = semget(key, 0, 0666)) == -1)
        {
                printf("Semaphore set does not exist!\n");
                exit(1);
        }
}
void createsem(int *sid, key_t key, int members)
{
        int cntr;
        union semun semopts;
        if(members > SEMMSL) {
                printf("Sorry, max number of semaphores in a set is %d\n",
                        SEMMSL);
                exit(1);
        }
        printf("Attempting to create new semaphore set with %d members\n",
                        members);
        if((*sid = semget(key, members, IPC_CREAT|IPC_EXCL|0666))
                        == -1)
        {
                fprintf(stderr, "Semaphore set already exists!\n");
                exit(1);
        }
        semopts.val = SEM_RESOURCE_MAX;
        /* Initialize all members (could be done with SETALL) */
        for(cntr=0; cntr<members; cntr++)
                semctl(*sid, cntr, SETVAL, semopts);
}
void locksem(int sid, int member)
{
        struct sembuf sem_lock={ 0, -1, IPC_NOWAIT};
        if( member<0 || member>(get_member_count(sid)-1))
        {
                fprintf(stderr, "semaphore member %d out of range\n", member);
                return;
        }
        /* Attempt to lock the semaphore set */
        if(!getval(sid, member))
        {
                fprintf(stderr, "Semaphore resources exhausted (no lock)!\n");
                exit(1);
        }
        sem_lock.sem_num = member;
        if((semop(sid, &sem_lock, 1)) == -1)
```

```
        {
                fprintf(stderr, "Lock failed\n");
                exit(1);
        }
        else
                printf("Semaphore resources decremented by one (locked)\n");
        dispval(sid, member);
}
void unlocksem(int sid, int member)
{
        struct sembuf sem_unlock={ member, 1, IPC_NOWAIT};
        int semval;
        if( member<0 || member>(get_member_count(sid)-1))
        {
                fprintf(stderr, "semaphore member %d out of range\n", member);
                return;
        }
        /* Is the semaphore set locked? */
        semval = getval(sid, member);
        if(semval == SEM_RESOURCE_MAX) {
                fprintf(stderr, "Semaphore not locked!\n");
                exit(1);
        }
        sem_unlock.sem_num = member;
        /* Attempt to lock the semaphore set */
        if((semop(sid, &sem_unlock, 1)) == -1)
        {
                fprintf(stderr, "Unlock failed\n");
                exit(1);
        }
        else
                printf("Semaphore resources incremented by one (unlocked)\n");
        dispval(sid, member);
}
void removesem(int sid)
{
        semctl(sid, 0, IPC_RMID, 0);
        printf("Semaphore removed\n");
}
unsigned short get_member_count(int sid)
{
        union semun semopts;
        struct semid_ds mysemds;
        semopts.buf = &mysemds;
        /* Return number of members in the semaphore set */
        return(semopts.buf->sem_nsems);
}
int getval(int sid, int member)
{
        int semval;
        semval = semctl(sid, member, GETVAL, 0);
        return(semval);
```

```
}
void changemode(int sid, char *mode)
{
        int rc;
        union semun semopts;
        struct semid_ds mysemds;
        /* Get current values for internal data structure */
        semopts.buf = &mysemds;
        rc = semctl(sid, 0, IPC_STAT, semopts);
        if (rc == -1) {
                perror("semctl");
                exit(1);
        }
        printf("Old permissions were %o\n", semopts.buf->sem_perm.mode);
        /* Change the permissions on the semaphore */
        sscanf(mode, "%ho", &semopts.buf->sem_perm.mode);
        /* Update the internal data structure */
        semctl(sid, 0, IPC_SET, semopts);
        printf("Updated...\n");
}
void dispval(int sid, int member)
{
        int semval;
        semval = semctl(sid, member, GETVAL, 0);
        printf("semval for member %d is %d\n", member, semval);
}
void usage(void)
{
        fprintf(stderr, "semtool - A utility for tinkering with semaphores\n");
        fprintf(stderr, "\nUSAGE:  semtool4 (c)reate <semcount>\n");
        fprintf(stderr, "               (l)ock <sem #>\n");
        fprintf(stderr, "               (u)nlock <sem #>\n");
        fprintf(stderr, "               (d)elete\n");
        fprintf(stderr, "               (m)ode <mode>\n");
        exit(1);
}
```

## 19.6　共享内存

### 19.6.1　基本概念

共享内存就是由几个进程共享一段内存区域。这可以说是最快的 IPC形式，因为它无须任何的中间操作（例如，管道、消息队列等）。它只是把内存段直接映射到调用进程的地址空间中。这样的内存段可以是由一个进程创建的，然后其他的进程可以读写此内存段。

### 19.6.2　系统内部用户数据结构shmid_ds

每个系统的共享内存段在系统内核中也保持着一个内部的数据结构 shmid_ds。此数据结构是在linux/shm.h中定义的，如下所示：

```
        /* One shmid data structure for each shared memory segment in the system. */
```

```
struct shmid_ds {
    struct ipc_perm shm_perm;       /* operation perms */
    int     shm_segsz;              /* size of segment (bytes) */
    time_t  shm_atime;              /* last attach time */
    time_t  shm_dtime;              /* last detach time */
    time_t  shm_ctime;              /* last change time */
    unsigned short  shm_cpid;       /* pid of creator */
    unsigned short  shm_lpid;       /* pid of last operator */
    short   shm_nattch;             /* no. of current attaches */

                                    /* the following are private */

    unsigned short   shm_npages;    /* size of segment (pages) */
    unsigned long   *shm_pages;     /* array of ptrs to frames -> SHMMAX */
    struct vm_area_struct *attaches;    /* descriptors for attaches */
};
```

shm_perm　是数据结构ipc_perm的一个实例。这里保存的是内存段的存取权限，和其他的有关内存段创建者的信息。

shm_segsz　内存段的字节大小。

shm_atime　最后一个进程存取内存段的时间。

shm_dtime　最后一个进程离开内存段的时间。

shm_ctime　内存段最后改动的时间。

shm_cpid　内存段创建进程的PID。

shm_lpid　最后一个使用内存段的进程的PID。

shm_nattch　当前使用内存段的进程总数。

### 19.6.3 系统调用shmget()

系统调用：shmget();

原型：int shmget ( key_t key, int size, int shmflg );

返回值：如果成功，返回共享内存段标识符。

如果失败，则返回-1：errno = EINVAL (无效的内存段大小)
EEXIST (内存段已经存在，无法创建)
EIDRM (内存段已经被删除)
ENOENT (内存段不存在)
EACCES (权限不够)
ENOMEM (没有足够的内存来创建内存段)

系统调用shmget() 中的第一个参数是关键字值（它是用系统调用 ftok()返回的）。其他的操作都要依据shmflg中的命令进行。

IPC_CREAT　如果系统内核中没有共享的内存段，则创建一个共享的内存段。

IPC_EXCL　当和IPC_CREAT一同使用时，如果共享内存段已经存在，则调用失败。

当IPC_CREAT单独使用时，系统调用shmget()要么返回一个新创建的共享内存段的标识符，要么返回一个已经存在的共享内存段的关键字值。如果 IPC_EXCL和IPC_CREAT一同使用，则要么系统调用新创建一个共享的内存段，要么返回一个错误值 -1。IPC_EXCL单独使用没有意义。

下面是一个定位和创建共享内存段的程序：

```
int open_segment( key_t keyval, int segsize )
{
    int    shmid;

    if((shmid = shmget( keyval, segsize, IPC_CREAT | 0660 )) == -1)
    {
        return(-1);
    }

    return(shmid);
}
```

一旦一个进程拥有了一个给定的内存段的有效 IPC标识符，它的下一步就是将共享的内存段映射到自己的地址空间中。

### 19.6.4　系统调用shmat()

系统调用： shmat();
原型：int shmat ( int shmid, char *shmaddr, int shmflg);
返回值：如果成功，则返回共享内存段连接到进程中的地址。
　　如果失败，则返回-1：errno =　EINVAL (无效的IPC ID 值或者无效的地址)
　　ENOMEM (没有足够的内存)
　　EACCES (存取权限不够)

如果参数addr的值为0，那么系统内核则试图找出一个没有映射的内存区域。我们推荐使用这种方法。你可以指定一个地址，但这通常是为了加快对硬件设备的存取，或者解决和其他程序的冲突。

下面的程序中的调用参数是一个内存段的IPC标识符，返回内存段连接的地址：

```
char *attach_segment( int shmid )
{
    return(shmat(shmid, 0, 0));
}
```

一旦内存段正确地连接到进程以后，进程中就有了一个指向该内存段的指针。这样，以后就可以使用指针来读取此内存段了。但一定要注意不能丢失该指针的初值。

### 19.6.5　系统调用shmctl()

系统调用：shmctl();
原型：int shmctl ( int shmqid, int cmd, struct shmid_ds *buf );
返回值： 0，如果成功。
　　-1，如果失败：errno =　EACCES (没有读的权限，同时命令是IPC_STAT)
　　EFAULT (buf指向的地址无效，同时命令是 IPC_SET和IPC_STAT )
　　EIDRM (内存段被移走)
　　EINVAL (shmqid 无效)
　　EPERM (使用IPC_SET 或者 IPC_RMID 命令，但调用进程没有写的权限)

IPC_STAT

读取一个内存段的数据结构shmid_ds，并将它存储在buf参数指向的地址中。

IPC_SET

设置内存段的数据结构shmid_ds中的元素ipc_perm的值。从参数buf中得到要设置的值。

IPC_RMID

标志内存段为移走。

命令IPC_RMID并不真正从系统内核中移走共享的内存段，而是把内存段标记为可移除。

进程调用系统调用shmdt（）脱离一个共享的内存段。

### 19.6.6 系统调用shmdt()

系统调用：shmdt();

调用原型：int shmdt ( char *shmaddr );

返回值：如果失败，则返回 -1：errno = EINVAL (无效的连接地址)

当一个进程不在需要共享的内存段时，它将会把内存段从其地址空间中脱离。但这不等于将共享内存段从系统内核中移走。当进程脱离成功后，数据结构 shmid_ds中元素shm_nattch将减1。当此数值减为0以后，系统内核将物理上把内存段从系统内核中移走。

### 19.6.7 使用共享内存的实例：shmtool

最后一个System V IPC目标的例子是shmtool，它同样也可以在shell下创建、读取、写入和删除共享的内存段。

1) 将字符串写入到内存段中

```
shmtool w "text"
```

2) 从内存段中读取字符串

```
shmtool r
```

3) 改变内存段的权限

```
shmtool m (mode)
```

4) 删除内存段

```
shmtool d
```

下面是应用的实例：

```
shmtool  w   test
shmtool  w   "This is a test"
shmtool  r
shmtool  d
shmtool  m   660
```

最后在这里给出程序的源代码如下：

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

#define SEGSIZE 100
main(int argc, char *argv[])
{       key_t key;
        int   shmid, cntr;
char  *segptr;
```

```
    if(argc == 1)
        usage();        /* Create unique key via call to ftok() */
    key = ftok(".", 'S');
    /* Open the shared memory segment - create if necessary */
    if((shmid = shmget(key, SEGSIZE, IPC_CREAT|IPC_EXCL|0666)) == -1)
    {
        printf("Shared memory segment exists - opening as client\n");
        /* Segment probably already exists - try as a client */
        if((shmid = shmget(key, SEGSIZE, 0)) == -1)
         {
            perror("shmget");
            exit(1);
        }
    }
    else
    {
        printf("Creating new shared memory segment\n");
    }
    /* Attach (map) the shared memory segment into the current process */
    if((segptr = shmat(shmid, 0, 0)) == -1)
    {
        perror("shmat");
        exit(1);
    }
    switch(tolower(argv[1][0]))
    {
        case 'w': writeshm(shmid, segptr, argv[2]);
                  break;
        case 'r': readshm(shmid, segptr);
                  break;
        case 'd': removeshm(shmid);
                  break;
        case 'm': changemode(shmid, argv[2]);
                  break;
         default: usage();
    }
}
writeshm(int shmid, char *segptr, char *text)
{
    strcpy(segptr, text);
    printf("Done...\n");
}
readshm(int shmid, char *segptr)
{
    printf("segptr: %s\n", segptr);
}
removeshm(int shmid)
{
    shmctl(shmid, IPC_RMID, 0);
    printf("Shared memory segment marked for deletion\n");
}
```

```
changemode(int shmid, char *mode)
 {      struct shmid_ds myshmds;
       /* Get current values for internal data structure */
       shmctl(shmid, IPC_STAT, &myshmds);       /* Display old permissions */
       printf("Old permissions were: %o\n", myshmds.shm_perm.mode);
       /* Convert and load the mode */
       sscanf(mode, "%o", &myshmds.shm_perm.mode);       /* Update the mode */
       shmctl(shmid, IPC_SET, &myshmds);
       printf("New permissions are : %o\n", myshmds.shm_perm.mode);
}
usage()
{
       fprintf(stderr, "shmtool - A utility for tinkering with shared memory\n");
       fprintf(stderr, "\nUSAGE:  shmtool (w)rite <text>\n");
       fprintf(stderr, "               (r)ead\n");
       fprintf(stderr, "               (d)elete\n");
       fprintf(stderr, "               (m)ode change <octal mode>\n");
       exit(1);
}
```