PDF-XChange
Click to buy NOW!
www.docu-track.com

深入浅出
Linux工具与编程

- ✧ 这是一部有思想、有内容的书刊。
- ✧ 这是一部回答学什么、怎么学的书刊。
- ✧ 这是一部大学毕业生一次阅读、终生受益的书刊。
- ✧ 这是一部集实用性、典型性、模仿性案例的书刊。
- ✧ 这是一部很通俗易懂的书刊。
- ✧ 这是一部帮你突破技术玻璃纸、一通百通的书刊。
- ✧ 这是一部包含作者从业十年心得经验的书刊。
- ✧ 这是一部零起点Linux专家的速成培训教程。

PDF-XChange Click to buy NOW! www.docu-track.com

时间是人类发展的空间，赢得时间就是赢得个人发展的空间。在个人的职业生涯中，一步领先常常可以做到步步领先。只要读者静下心来一个月对本书进行阅读，按照教材进行练习，读者就可以大大提高Linux工具及其编程的技术水平，从而让读者在从业生涯中做到终生受益，完全掌握本书内容即可达到Linux专家水平。可以说本书是一本通向Linux专家之路的速成教程。

由于作者水平有限，本书有所错漏在所难免，希望读者批评指正。

本书内容：

本书总共六篇，所有内容注重了理论联系实际和实战性。每篇的主要内容如下：

第一篇 Linux命令及其工具

本篇包括Linux操作系统介绍，Linux命令说明，Linux常见实用工具（正则表达式、find、sed、awk）说明及实例练习，Shell编程语法说明及编程实例。

第二篇 Linux C语言程序设计

本篇包括C语言基础、C语言控制结构、C语言函数、C语言数组、结构体及指针、C语言预编译、格式化I/O函数、字符串和内存操作函数、字符类型测试函数、字符串转换函数、Linux C语言开发工具(vi与vim编辑器、gcc、Makefile和gdb)。本篇多次运用堆栈表格对程序运行进行解释，这对于理解计算机语言运行机理非常重要。只有理解的才是最深刻的，理解其运行机理，可以触类旁通、一通百通，移植到理解C++语言和Java等语言。

第三篇 Linux 进程

本篇包括Linux编程基本概念、Linux进程、Linux线程、管道与信号、消息队列、信号量和共享内存。Linux进程章节中守护进程模板和数据仓库多进程处理案例可以应用到实际项目中。本篇Linux进程间通信程序范例是实际项目中精简的demo程序，程序模型和使用方法与实际项目中类似。

第四篇 Linux 文件

本篇包括Linux文件编程，该章节对文件函数进行了分类总结并提供了典型范例。

第五篇 网络编程

本篇包括网络知识基础、socket编程。socket编程章节包括TCP并发服务器案例、TCP迭代服务器案例、文件服务器案例、UDP服务器编程、UDP广播、UDP多播、Unix/Linux域套接字编程等。

第六篇 XML编程

本篇包括XML概念、XML语法、XPath语法、libxml编程、支付宝银行端接口XML项目案例。本篇是目前市面上唯一对Linux下XML编程进行全面总结的书刊，在实际项目开发中有较大的借鉴意义。

PDF-XChange
Click to buy NOW!
www.docu-track.com

第 3 篇 Linux 进程

- 第 12 章 Linux进程编程
- 第 13 章 Linux线程编程
- 第 14 章 Linux进程间通信—管道与信号
- 第 15 章 System V 进程间通讯

学海聆听：

- 三思而后行。
- 善学者，假人之长以补其短。
- 培养一种良好的习惯受益终生。
- 许多事物的外延无法改变，然而内涵修炼能使事物大大增值。
- 人无远虑，必有近忧。
- 目标、态度、方法、意志、爱好。
- 知之者不如乐之者，乐知者不如好之者；兴趣是最好的老师。
- 理想是力量的泉源、智慧的摇篮、冲锋的战旗、斩棘的利剑。
- 知识改变命运，态度影响未来。
- 读书需用心，一字值千金。
- 懵懂而死，与草木同朽；悟道而生，是为永生。
- 行百里者半九十；从量变到质变，需要锲而不舍的努力。

带格式的：项目符号和编号

第1章 Linux进程间通信—管道与信号

管道和信号是进程间通信的两种机制。现实中的管道，水从一端流入，然后从另一端流出；与此类似操作系统中的每一个管道有两个文件描述符，一个文件描述符用来读，另一个用来写。信号是一个软件中断，主要用于进程间异步事件通知与进程控制。

1. 进程间的通信类型

进程间的通信类型有如下六种：

① 管道（pipe）和有名管道（FIFO）。
② 信号（signal）。
③ 共享内存。
④ 消息队列。
⑤ 信号量。
⑥ 套接字（socket）。

2. 进程间通信目的

进程间通信目的有如下五种：

① 数据传输：一个进程需要将它的数据发送给另一个进程。
② 共享数据：多个进程想要共享数据，一个进程对共享数据进行修改后，别的进程可以立刻看到。
③ 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
④ 资源共享：多个进程之间共享同样的资源。为了做到这一点，需要内核提供锁和同步机制。
⑤ 进程控制：有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

带格式的：项目符号和编号

1.2 管道

管道是Linux中最常用的进程间通信IPC机制。使用管道时，一个进程的输出可成为另外一个进程的输入。

当输入/输出的数据量特别大时，管道这种IPC机制非常有用。

在Linux中，通过将两个file结构指向同一个临时的VFS索引节点，而两个VFS索引节点又指向同一个物理页而实现管道。

如图 14-1 所示为管道的内核实现图，每个file数据结构定义不同的文件操作地址，其中一个用来向管道中写入数据，而另外一个用来从管道中读出数据。

当写进程向管道中写入时，它利用标准的库函数进行写操作，系统根据库函数传递的文件描述符可找到该文件的file结构。

file结构中指定了用来进行写操作的地址，于是，内核调用写函数向指定地址完成写操作。

写入函数在向内存中写入数据之前，必须首先检查VFS索引节点信息，同时满足如下条件时，才能进行实际的内存复制工作。

- 内存中有足够的空间可容纳所有要写入的数据。

- 内存没有被读程序锁定。

管道的读取过程和写入过程类似。

管道的打开模式为非阻塞的情况下，进程可以在没有数据或内存被锁定时立即返回错误信息，而不是阻塞该进程。

管道的打开模式为阻塞的情况下，进程可以休眠在索引节点的等待队列中等待写入进程写入数据。

当所有的进程完成了管道操作之后，管道的索引节点被丢弃，而共享数据页也被释放。

上面讲述的管道类型也被称为“匿名管道”。匿名管道包括pipe管道和标准流管道。

Linux还支持另外一种管道形式，称为命名管道（FIFO），这种管道的操作方式基于“先进先出”原理。

命名管道和匿名管道的数据结构以及操作极其类似，二者的主要区别在于：命名管道在使用之前就已经存在，用户可打开或关闭命名管道；而匿名管道只在操作时存在，因而是临时对象。

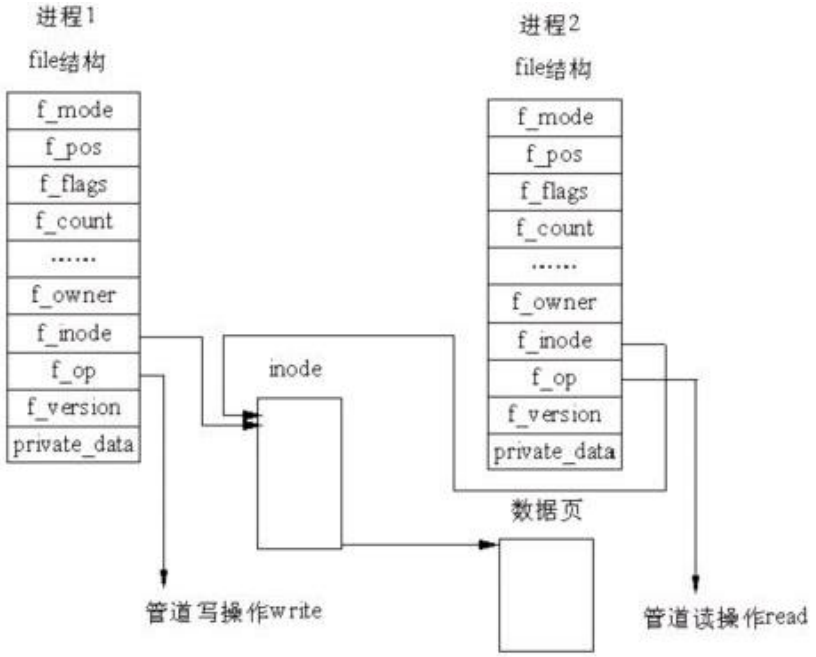

图 14-1 管道的内核实现图

带格式的：项目符号和编号

1.2.1 pipe 管道

1. pipe函数原型

若要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，一个为读端，一个为写端。pipe管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）。

pipe函数原型如下：

```
pipe(建立简单管道)
```

PDF-XChange Click to buy NOW! www.docu-track.com

所需头文件	#include <unistd.h>
函数说明	pipe()会建立管道，并将文件描述符由参数 filedes 数组返回。
函数原型	int pipe(int filedes[2])
函数传入值	filedes：管道文件描述符，filedes[0]为管道的读取端，filedes[1]则为管道的写入端
函数返回值	成功：0
	出错：-1，错误原因存于 errno 中
错误代码	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
附加说明	管道主要用于父子进程间通信。实际上，通常先创建一个管道，再通过 fork 函数创建一个子进程。其实 pipe 的两个文件描述符指向相同的内存空间，只不过 filedes[1]有写权限，filedes[0]有读权限

2. pipe管道原理测试范例

下面程序测试pipe管道的实现原理，pipetest.c源代码如下：

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
void look_into_pipe() {
     int n;
    int fd[2];
    char line[1024];
    struct stat buf;
    if (pipe(fd) < 0) {        /*创建管道*/
        printf("pipe error.\n");
        return;
    }
    fstat(fd[0], &buf);
    if (S_ISFIFO(buf.st_mode)) { /*S_ISFIFO为测试此文件类型是否为管道文件*/
        printf("fd[0]: FIFO file type.\n");
    }
    printf("fd[0]: inode=%d\n", buf.st_ino);
    fstat(fd[1], &buf);
    if (S_ISFIFO(buf.st_mode)) {
        printf("fd[1]: FIFO file type.\n");
    }
    printf("fd[1]: inode=%d\n", buf.st_ino);
    write(fd[1], "hello world.\n", 12);
    n = read(fd[0], line, 512 );
    write(STDOUT_FILENO, line, n);
    n=write(fd[0], "HELLO WORLD.\n", 12);  /*0 端只允许读，1 端才允许写，这样做是为了测试*/
    if (  -1==n  )
    {
        printf("\nwrite error\n") ;
    }
}
int main(){
    look_into_pipe() ;
}
```

编译 gcc pipetest.c –o pipetest。

执行 ./pipetest，执行结果如下：

```
fd[0]: FIFO file type.
fd[0]: inode=24962
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
fd[1]: FIFO file type.
fd[1]: inode=24962
hello world.
write error
```

3. pipe管道典型应用范例

（1）pipe管道典型应用程序

pipe管道的典型应用是使用在父子通信的场合。下面程序是pipe管道典型应用程序范例， pipe.c源代码如下：

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(){
    int pipe_fd[2];
    pid_t pid;
    char buf_r[100];
    char *p_wbuf;
    int r_num;

    memset(buf_r,0,sizeof(buf_r));
    //创建管道
    if(pipe(pipe_fd)<0){
        perror("pipe create error\n");
        return -1;
    }
    if((pid=fork())==0){//表示在子进程中
        //关闭管道写描述符，进行管道读操作
        printf("child pipe1=%d; pipe2=%d\n", pipe_fd[0], pipe_fd[1] ) ;
        close(pipe_fd[1]);
        //管道描述符中读取
        sleep(2) ;
        if((r_num=read(pipe_fd[0],buf_r,100))>0){
            printf("%d numbers read from the pipe, data is %s\n",r_num,buf_r);
        }
        close(pipe_fd[0]);
        exit(0);
    }
    else if(pid>0){//表示在父进程中，父进程写
        //关闭管道读描述符，进行管道写操作
        printf("parent pipe1=%d; pipe2=%d\n", pipe_fd[0], pipe_fd[1] ) ;
        close(pipe_fd[0]);
        if(write(pipe_fd[1],"Hello",5)!=-1)
            printf("parent write1 success!\n");
        if(write(pipe_fd[1]," Pipe",5)!=1)
            printf("parent write2 success!\n");
        close(pipe_fd[1]);
        sleep(3) ;
        waitpid(pid,NULL,0);
        exit(0);
    }else{
        perror("fork error");
        exit(-1);
    }
}
```

编译 gcc pipe.c –o pipe。

执行 ./pipe，执行结果如下：

```
child pipe1=3; pipe2=4
parent pipe1=3; pipe2=4
parent write1 success!
parent write2 success!
10 numbers read from the pipe, data is Hello Pipe
```

（2）pipe管道实现图解

对于pipe.c程序，其管道实现如图 14-2 ~ 图 14-4 所示。

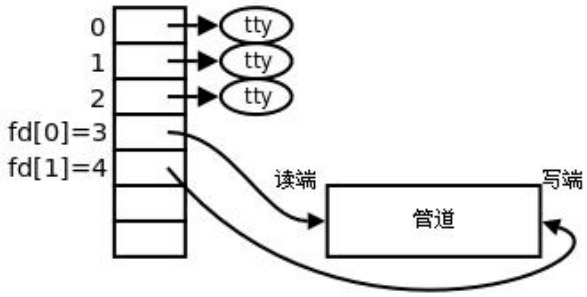

图 14-2 父进程创建的管道

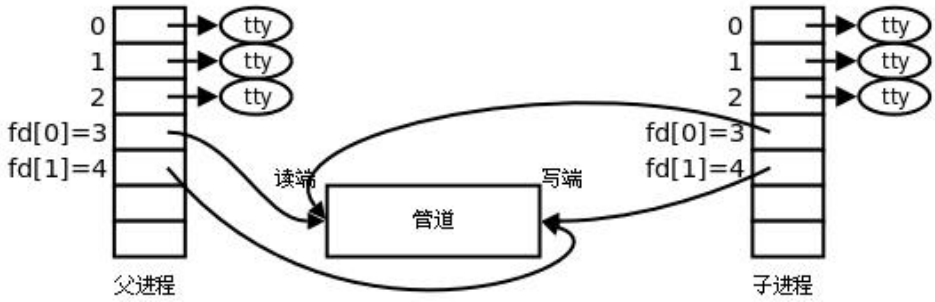

图 14-3 父进程fork出子进程

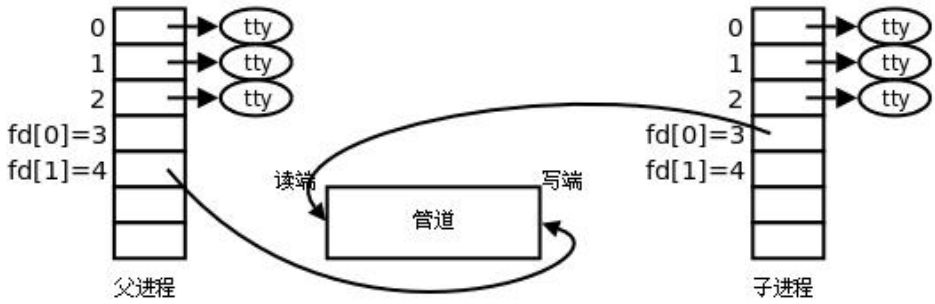

图 14-4 父进程关闭fd[0]，子进程关闭fd[1]

管道的概念是借鉴现实中形象的管道，但读端和写端指向相同的inode地址，图 14-2~图 14-4 这样画出是让大家能形象地理解。由于进程的文件描述符 0 指向标准输入（键盘），1 指向标准输出（屏幕），2 指向标准错误（屏幕），所以进程可用的文件描述符从 3 开始。进程是通过对文件描述符的操作从而实现对文件描述符所指向文件的操作。

（3）pipe.c程序的简要说明

对于pipe.c程序，简要说明如下：

父进程调用fork()产生子进程时，子进程复制了几乎父进程的所有资源，所

以也复制了管道文件描述符。由于两个进程的管道文件描述符都指向同一个inode地址，所以能实现能相互通信。

本程序实现的是父子进程单工通信，一个进程为读端，一个进程为写端，所以写端需要关闭读文件描述符，读端需要关闭写文件描述符，以免误操作。一个进程在一个管道上完成又读又写功能技术上能实现，但造成了程序控制流程的复杂和功能的不清晰；若要实现父子进程的双工通讯，一般是通过建立两个管道来实现，双工通信实现原理与单工通信类似。

带格式的：项目符号和编号

1.2.2 标准流管道

像Linux系统中的文件操作是基于文件流标准I/O一样，管道的操作也支持文件流模式，这种管道称为标准流管道。标准流管道通过popen()创建一个管道，popen()会调用fork()产生一个子进程，执行一个shell以运行命令来开启一个进程，并把执行结果写入管道中，然后返回一个文件指针。程序通过文件指针可读取管道中的内容。使用popen()创建的标准流管道，需要用pclose()进行关闭。

1. 标准流管道函数说明

下面两表列出了popen函数、pclose函数的函数原型及其具体说明。

popen（建立标准流管道）	
所需头文件	#include <stdio.h>
函数说明	popen()会调用 fork()产生子进程，然后从子进程中调用/bin/sh -c 来执行参数 command 的指令。参数 type 可使用“r”代表读取，“w”代表写入。依照此 type 值，popen()会建立管道连到子进程的标准输出设备或标准输入设备，然后返回一个文件指针。随后进程便可利用此文件指针来读取子进程的输出设备或是写入到子进程的标准输入设备中。此外，所有使用文件指针(FILE*)操作的函数也都可以使用，除了 fclose()以外
函数原型	FILE * popen(const char * command,const char * type)
函数传入值	command：执行的指令
	type：“r”代表读取，“w”代表写入
函数返回值	成功：文件流指针
	出错：NULL，错误原因存于 errno 中
错误代码	EINVAL：参数 type 不合法
注意事项	在编写具有 SUID/SGID 权限的程序时请尽量避免使用 popen()，popen()会继承环境变量，通过环境变量可能会造成系统安全的问题
附加说明	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

pclose（关闭标准流管道）	
所需头文件	#include <stdio.h>
函数说明	pclose()用来关闭由 popen 所建立的管道及文件指针。参数 stream 为先前由 popen()所返回的文件指针
函数原型	int pclose(FILE * stream)
函数传入值	stream：文件流指针
函数返回值	成功：子进程的结束状态
	失败：-1，错误原因存于 errno 中
错误代码	ECHILD：pclose()无法取得子进程的结束状态

2. 标准流管道代码举例

popen.c 源代码如下：

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#define BUFSIZE 1024
int main()
{
    FILE *fp;
    char *cmd = "ps -ef";
    char buf[BUFSIZE];
    buf[BUFSIZE] = '\0';
    if((fp=popen(cmd,"r"))==NULL)
        perror("popen");
    while((fgets(buf,BUFSIZE,fp))!=NULL)
        printf("%s",buf);
    pclose(fp);
    exit(0);
}
```

编译 gcc popen.c －o popen。

执行 ./popen，执行结果如下：

```
root      4325     1  0 17:57 tty4     00:00:00 /sbin/getty 38400 tty4
root      4330     1  0 17:57 tty2     00:00:00 /sbin/getty 38400 tty2
    ……
```

带格式的：项目符号和编号

1.2.3 命名管道(FIFO)

1. 命令管道原理

命名管道（FIFO）和一般的管道基本相同，但也有以下一些显著的不同：

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命名管道共享数据。

当共享命名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。

普通管道只能由父子兄弟等相关进程使用，它们共同的祖先进程创建了管道。但是，通过命名管道，不相关的进程也能交换数据。

一旦已经用mkfifo函数创建了一个FIFO，就可用open打开它。实际上，一般的文件I/O函数（close、read、write、unlink等）都可用于FIFO。

2. 命名管道函数原型

命名管道mkfifo函数原型及其说明如下。

mkfifo（创建命名管道）	
所需头文件	#include <sys/types.h> #include <sys/stat.h>
函数说明	mkfifo()会依参数 pathname 建立特殊的 FIFO 文件，该文件必须不存在，而参数 mode 为该文件的权限（mode&~ umask），因此 umask 值也会影响到 FIFO 文件的权限。mkfifo()建立的 FIFO 文件其他进程都可以用读写一般文件的方式存取。当使用 open()来打开 FIFO 文件时，O_NONBLOCK 旗标会有影响： ① 当使用 O_NONBLOCK 旗标时，打开 FIFO 文件来读取的操作会立刻返回，但是若还没有其他进程打开 FIFO 文件来读取，则写入的操作会返回 ENXIO 错误代码 ② 没有使用 O_NONBLOCK 旗标时，打开 FIFO 来读取的操作会等到其他进程打开 FIFO 文件来写入才正常返回。同样，打开 FIFO 文件来写入的操作会等到其他进程打开 FIFO 文件来读取后才正常返回

PDF-XChange Click to buy NOW! www.docu-track.com

函数原型	int mkfifo(const char * pathname,mode_t mode)
函数传入值	pathname：管道文件名
	mode：管道创建方式
函数返回值	成功：0
	失败：-1，错误原因存于 errno 中
错误代码	EACCESS：参数 pathname 所指定的目录路径无可执行的权限 EEXIST：参数 pathname 所指定的文件已存在 ENAMETOOLONG：参数 pathname 的路径名称太长 ENOENT：参数 pathname 包含的目录不存在 ENOSPC：文件系统的剩余空间不足 ENOTDIR：参数 pathname 路径中的目录存在但却非真正的目录 EROFS：参数 pathname 指定的文件存在于只读文件系统内 ENXIO：指定的设备或地址不存在

3. 命名管道代码举例

本例展示不同的程序通过命名管道交换数据。

（1）创建命名管道并写入数据

fifo_snd.c源代码如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#define FIFO  "/tmp/fifo"
int main()
{
    char buffer[80];
    int fd;
    int n ;
    int ret ;
    char info[80] ;
    unlink(FIFO); /*若存在该管道文件，  则进行删除*/
    ret = mkfifo(FIFO, 0600); /*0600 表明只有该用户进程有读写权限*/
    if ( ret )
    {
        perror("mkfifo error") ;
        return -1 ;
    }
    memset(info, 0x00, sizeof(info)) ;
    strcpy(info, "happy new year!") ;
    fd = open (FIFO,O_WRONLY);
    n=write(fd, info, strlen(info) );
    if ( n < 0 )
    {
        perror("write error") ;
        return -1 ;
    }
    close(fd);
    return 0 ;
}
```

（2）从管道中读取数据

fifo_rcv.c源代码如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <stdio.h>
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
#include <fcntl.h>
#define FIFO  "/tmp/fifo"
int main()
{
    char buffer[80];
    int fd;
    int n ;
    char info[80] ;
    fd= open(FIFO,O_RDONLY);
    n = read(fd,buffer,80);
    if ( n < 0 )
    {
        perror("read error") ;
        return -1 ;
    }
    printf("buffer=%s\n", buffer);
    close(fd);
    return 0 ;
}
```

（3）编译与执行

① 编译 gcc fifo_snd.c –o fifo_snd。

② 编译 gcc fifo_rcv.c –o fifo_rcv。

③ 在此用户下执行./fifo_snd，可以看见此进程正在阻塞。

④ 在此用户其他界面下执行./fifo_rcv，执行结果如下：

```
buffer=happy new year!
```

⑤ 查看此管道文件l /tmp/fifo，此管道文件存在。

```
prw------- 1 zjkf db2iadm1 0 2011-01-13 02:31 /tmp/fifo
```

⑥ 在其他用户下执行./fifo_rcv，由于管道文件权限为 0600，执行结果如下：

```
read error: Bad file descriptor
```

带格式的：项目符号和编号

1.3 信号

1.3.1 信号概述

信号是进程间通信机制中唯一的异步通信机制，可以看作是异步通知，通知接收信号的进程有哪些事情发生了。

信号同时又是一种软件中断，当某进程接收到信号时，会中止当前程序的执行，去处理信号的注册函数，然后回到断点程序继续往下执行。

信号事件的发生由两类原因引起，一为是硬件原因引起(按下键盘或者其他硬件故障)，如在终端上按DELETE键通常产生中断信号SIGINT。一为软件原因引起，如kill、raise、alarm、setitimer等系统函数会引起信号的发送，同时除 0 等非法运算也会引起信号的发送。

进程能对每一个信号设置独立的处理方式：它能忽略该信号，也能设置相应的信号处理程序（称为捕捉），或对信号什么也不做，信号发生的时候执行系统的默认动作。

进程还能通过信号集操作函数设置对某些信号的阻塞(block)标志。如果一个信号被设置为阻塞，当信号发生的时候，它会和正常的信号一样被递送(deliver)给进程，但只有进程解除对该信号的阻塞时才会被处理。也就是说信号阻塞只阻止信号被处理，但不阻止信号的产生。

从一个信号被递送给进程到该信号得到处理之间的时间间隔称为信号未决

（或称信号挂起、信号被搁置）。

所有的信号中，有两个信号（SIGSTOP和SIGKILL）是特别的，它们既不能被捕捉、也不能被忽略、也不能被阻塞，这个特性确保了系统管理员在所有时候内都能用暂停信号和杀死信号结束某个进程。

1. 信号分类

信号有两种分类方式：从可靠性上可分为可靠信号与不可靠信号，从与时间的关系上可分为实时信号与非实时信号。

Linux信号机制基本上是从Unix系统中继承过来的。早期Unix系统中的信号机制比较简单和原始，后来在实践中暴露出一些问题。因此，把那些建立在早期机制上的信号叫做“不可靠信号”，信号值小于 32 的信号都是不可靠信号，这就是“不可靠信号”的来源。不可靠信号的主要问题是：进程每次处理某个信号后，就对该信号的响应设置为默认动作。在某些情况下，这将导致对信号的错误处理。因此，用户如果不希望这种结果，那么就要在信号处理函数结尾再一次调用signal重新安装该信号。其次不可靠信号还有可能造成信号丢失。因此，早期Unix下的不可靠信号主要指的是进程可能对信号做出错误的反应以及信号的可能丢失。

随着时间的发展，实践证明了有必要对信号的原始机制加以改进和扩充。所以，后来出现的各种Unix版本分别在这方面进行了研究，力图实现“可靠信号”。由于原来定义的信号已有许多应用，不好再做改动，最终只好又新增加了一些信号，并在一开始就把它们定义为可靠信号，这些信号支持排队，不会丢失。同时，信号的发送和安装也出现了新版本的信号发送函数sigqueue()及信号安装函数sigaction()。POSIX 4 对可靠信号机制做了标准化。但是，POSIX只对可靠信号机制应具有的功能以及信号机制的对外接口做了标准化，对信号机制的实现没有作具体的规定。信号值位于 32 和 64 之间的信号都是可靠信号，可靠信号克服了信号可能丢失的问题。

Linux支持不可靠信号，但是对不可靠信号机制做了改进：在调用完信号处理函数后，不必重新调用该信号的安装函数，Linux信号安装函数sigaction是在可靠机制上的实现。因此，Linux下不可靠信号的问题主要指的是信号可能丢失。

Linux在支持新版本的信号安装函数sigaction以及信号发送函数sigqueue的同时，仍然支持早期的signal信号安装函数和信号发送函数kill。

不要有这样的误解，由sigqueue发送、由sigaction安装的信号就是可靠的。事实上，可靠信号是指后来添加的新信号（信号值位于 32 及 64 之间），不可靠信号是指信号值小于 32 的信号。信号的可靠与不可靠只与信号值有关，与信号的发送及安装函数无关。目前Linux系统中的signal是通过sigaction函数封装实现的；因此，即使通过signal安装的信号，在信号处理函数的结尾也不必再调用一次信号安装函数。同时，由signal安装的实时信号支持排队，同样不会丢失。

对于目前Linux的两个信号安装函数signal及sigaction来说，它们都不能把 32 以前的信号变成可靠信号（1~31 的信号不支持排队，仍有可能丢失，仍然是不可靠信号），而对 32 以后的信号都支持排队。这两个函数的最大区别在于，经过sigaction安装的信号都能传递信息给信号处理函数，而经过signal安装的信号却不能向信号处理函数传递信息。

非实时信号都不支持排队，都是不可靠信号；实时信号都支持排队，都是

PDF-XChange Click to buy NOW! www.docu-track.com

可靠信号。实时信号之所以是可靠的，因为在该进程阻塞等待处理的时间内，发送给该进程的所有实时信号会排队等待；而对于非实时信号则系统只会处理第一个信号，在该进程阻塞的时间内后续信号被简单丢弃。早期的kill函数只能向特定的进程发送一个特定的信号，并且早期的信号处理函数也不能接收附加数据，而sigqueue和sigaction解决了这个问题。

kill -l命令可以查看信号列表。信号可以由数字表示，也可以用SIG开头的宏表示。信号值为 1~ 31 的信号为传统Unix支持的信号，是不可靠信号(非实时信号)；信号值为 32 ~ 63 的信号是后来扩充的，称做可靠信号(实时信号)。现在在Linux系统中，不可靠信号和可靠信号的区别在于前者不支持排队，可能会造成信号的丢失，而后者不会。

2. 信号源分类

内核为进程产生信号，来说明不同的事件，这些事件就是信号源。主要的信号源种类如下 7 种：

① 异常：进程运行过程中出现异常。
② 其他进程：一个进程可以向另外一个或一组进程发送信号。
③ 终端中断：Ctrl-C、Ctrl-\等。
④ 作业控制：前台、后台进程的管理。
⑤ 分配额：CPU超时或文件大小突破限制。
⑥ 通知：通知进程某事件发生，如I/O就绪等。
⑦ 报警：计时器到期。

3. 不可靠信号说明

表 14-1 列出了不可靠信号列表及其说明。在实际应用编程中，用到的主要还是不可靠信号（也就是信号值 1~31 的信号）。

表 14-1 不可靠信号列表

信号值	信号宏名称	默认动作	说明
1	SIGHUP	终止进程	从终端上发出的结束信号
2	SIGINT	终止进程	中断进程
3	SIGQUIT	建立 CORE 文件	终止进程，并且生成 core 文件
4	SIGILL	建立 CORE 文件	非法指令
5	SIGTRAP	建立 CORE 文件	跟踪自陷
6	SIGABRT	建立 CORE 文件	异常终止
7	SIGBUS	建立 CORE 文件	总线错误
8	SIGFPE	建立 CORE 文件	浮点异常
9	SIGKILL	终止进程	杀死进程
10	SIGUSR1	终止进程	用户定义信号 1
11	SIGSEGV	建立 CORE 文件	段非法错误
12	SIGUSR2	终止进程	用户定义信号 2
13	SIGPIPE	终止进程	向一个没有读进程的管道写数据
14	SIGALARM	终止进程	计时器到时
15	SIGTERM	终止进程	软件终止信号
16	SIGSTKFLT	终止进程	Linux 专用，数学协处理器的栈异常
17	SIGCHLD	忽略信号	当子进程停止或退出时通知父进程
18	SIGCONT	忽略信号	继续执行一个停止的进程
19	SIGSTOP	停止进程	非终端来的停止信号
20	SIGTSTP	停止进程	终端来的停止信号
21	SIGTTIN	停止进程	后台进程读终端

22	SIGTTOU	停止进程	后台进程写终端
23	SIGURG	忽略信号	I/O 紧急信号
24	SIGXGPU	终止进程	超过 CPU 时间限制
25	SIGXFSZ	终止进程	进程超过文件长度限制
26	SIGVTALRM	终止进程	虚拟计时器超时
27	SIGPROF	终止进程	统计分布计时器超时
28	SIGWINCH	忽略信号	窗口大小发生变化
29	SIGIO	忽略信号	异步 I/O 事件
30	SIGPWR	忽略信号	电源失效再启动
31	SIGSYS	建立 CORE 文件	非法系统调用

4. 常见信号说明

表 14-2 列出了常见信号列表及其说明。

表 14-2 常见信号列表

信号宏名称	信号说明
SIGHUP	从终端上发出的结束信号
SIGINT	来自键盘的中断信号（Ctrl-C）
SIGQUIT	来自键盘的退出信号（Ctrl-\）
SIGFPE	浮点异常信号（例如浮点运算溢出）
SIGKILL	该信号结束接收信号的进程
SIGALRM	进程的定时器到期时，发送该信号
SIGTERM	kill 命令发出的信号
SIGCHLD	标识子进程停止或结束的信号
SIGSTOP	来自键盘（Ctrl-Z）或调试程序的停止执行信号

5. 信号的三种操作方式

信号具有以下三种操作方式：

① 忽略此信号，SIG_IGN常数表示信号函数的忽略。但SIGKILL和SIGSTOP信号不能忽略。

② 捕捉信号。在某种信号发生时，调用一个用户自定义函数，在用户自定义函数中可执行用户希望对这个事件进行的处理。

③ 执行系统的默认动作，SIG_DFL常数表示信号函数的默认。对大多数信号来说系统的默认动作是终止该进程。

6. 信号的五种默认动作

信号发生时，对接收信号进程的处理有如下五种默认动作：

① 异常终止（abort）：在进程的当前目录下，把进程的地址空间内容、寄存器内容保存到一个叫做core的文件中，而后终止该进程。

② 退出（exit）：不产生core文件，直接终止该进程。

③ 忽略（ignore）：忽略该信号。

④ 停止（stop）： 挂起该进程。

⑤ 继续（continue）：如果进程被挂起，则恢复进程的运行。否则，忽略信号。

7. 阻塞信号和忽略信号两种操作的区别

阻塞信号允许信号被发送给进程，但不进行处理，需要等到阻塞解除后再处理。而忽略信号是进程根本不接收该信号，所有被忽略的信号都被简单丢弃。

用系统调用sigprocmask可设置信号是否被阻塞，而系统调用sigaction和库函数signal则设置进程是否忽略一个信号。

1.3.2 信号的发送和捕捉函数

下面是信号中常用到的信号发送和捕捉函数，其中alarm和kill两个信号发送函数在应用编程中最为常用。

1. alarm函数

(1) alarm函数原型

alarm（设置信号传送闹钟）	
所需头文件	#include <unistd.h> #include <signal.h>
函数说明	alarm()用来设置信号SIGALRM在经过参数seconds指定的秒数后传送给目前的进程。如果参数seconds 为0，则之前设置的闹钟会被取消，并将剩下的时间返回
函数原型	unsigned int alarm(unsigned int seconds)
函数返回值	返回之前闹钟的剩余秒数，如果之前未设闹钟，则返回0

(2) alarm函数说明

当所设置的时间值超过后，产生SIGALRM信号。如果不忽略或不捕捉此信号，则其默认动作是终止该进程。

每个进程只能有一个闹钟时间。如果在调用alarm时，以前已为该进程设置过闹钟时间，而且它还没有超时，则该闹钟时间的剩余时间值作为本次alarm函数调用的值返回，以前登记的闹钟时间则被新值代换。

如果有以前登记的尚未超过的闹钟时间，而新设的闹钟时间值为0，则取消以前的闹钟时间，其剩余时间值仍作为函数的返回值。

(3) alarm函数举例

alarm.c 源代码如下：

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
void handler() {  /*信号处理函数*/
    printf("hello\n");
}
int main()
{
    int i;
    int time ;
    signal(SIGALRM,handler);  /*注册SIGALRM信号处理方式*/
    alarm(3);
    for(i=1;i<5;i++){
        printf("sleep %d ...\n", i);
        sleep(1);
    }

    alarm(3);
    sleep(2);
    time=alarm(0) ; /*取消SIGALRM信号，返回剩余秒数*/
    printf("time=%d\n",time) ;
    for(i=1;i<3;i++){
        printf("sleep %d ...\n", i);
        sleep(1);
    }
```

```
    return 0 ;
}
```

编译 gcc alarm.c –o alarm。

执行./alarm，执行结果如下：

```
sleep 1 ...
sleep 2 ...
sleep 3 ...
hello
sleep 4 ...
time=1
sleep 1 ...
sleep 2 ...
```

对程序结果分析如下：

信号是一种软中断，中断的原理是保留执行现场，跳转到中断函数处执行，执行完后恢复以前的现场在断点处继续往下执行。所以alarm函数经过 3 秒时，for循环已经到了sleep的 3 秒位置；alarm发送闹铃信号，引起中断，程序执行SIGALRM注册函数handler（中断处理函数），即输出hello。信号处理handler函数执行完后，应用程序从中断点恢复，继续执行for循环，此时sleep到了 4 秒的位置。

alarm(0)取消注册的闹铃（SIGALRM）信号，所以，第二次没有执行SIGALRM信号注册函数。

2. kill函数

(1) kill函数原型

kill函数是将信号发送给指定的pid进程。普通用户利用kill函数将信号发送给该用户下任意一个进程，而特权用户(root)可以将信号发送系统中的任意一个进程。

kill函数原型及说明如下：

kill(传送信号给指定的进程)		
所需头文件	#include <sys/types.h> #include <signal.h>	
函数说明	kill()可以用来传送参数 sig 指定的信号给参数 pid 指定的进程	
函数原型	int kill(pid_t pid,int sig)	
函数传入值	pid	pid>0 将信号传给进程识别码为 pid 的进程
		pid=0 将信号传给和目前进程相同进程组的所有进程
		pid=-1 将信号广播传送给系统内所有的进程
		pid<0 将信号传给进程组识别码为 pid 绝对值的所有进程
	sig	信号编号
函数返回值	成功：0	
	出错：-1，错误原因存于 error 中	
错误代码	EINVAL：参数 sig 不合法 ESRCH：参数 pid 所指定的进程或进程组不存在 EPERM：权限不够无法传送信号给指定进程	

(2) kill函数举例

下面代码实现的是父进程发信号给子进程。

kill.c 源代码如下：

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

PDF-XChange Click to buy NOW! www.docu-track.com

```
#include <sys/types.h>
#include <sys/wait.h>
int main()
{
    pid_t pid;
    int status;

    pid= fork() ;
    if( 0==pid ){
        printf("Hi I am child process!\n");
        sleep(10);
    }
    else if ( pid > 0 ){
        printf("send signal to child process (%d) \n",pid);
        sleep(1);
   /*发送SIGABRT信号给子进程，此信号引起接收进程异常终止*/
        kill(pid ,SIGABRT);
   /*等待子进程返回终止信息*/
        wait(&status);
        if(WIFSIGNALED(status))
            printf("chile process receive signal %d\n",WTERMSIG(status));
    }else{
        perror("fork error") ;
        return -1 ;
    }

    return 0 ;
}
```

编译 gcc kill.c –o kill。

执行./kill，执行结果如下：

```
Hi I am child process!
send signal to child process (10498)
chile process receive signal 6
```

3. raise函数

与kill函数不同的是，raise函数运行是向进程自身发送信号。

raise（向自己发送信号）	
所需头文件	#include <sys/types.h> #include <signal.h>
函数说明	向自己发送信号
函数原型	int raise(int sig)
函数传入值	sig：信号
函数返回值	成功：0
	出错：-1，错误原因存于 error 中

4. pause函数

(1) pause函数原型

pause（让进程暂停直到信号出现）	
所需头文件	#include <unistd.h>
函数说明	令目前的进程暂停（进入睡眠状态），直到被信号(signal)所中断
函数原型	int pause(void)
函数返回值	只返回-1，错误原因存于 error 中
错误代码	EINTR：有信号到达中断了此函数

PDF-XChange Click to buy NOW! www.docu-track.com

(2) pause函数举例

下面pause.c源代码简单的实现了sleep函数的功能。由于SIGALRM信号默认的系统动作为终止进程，所以在程序调用pause之后就暂停，当 3 秒后接收到alarm信号时进程就终止。

pause.c源代码如下：

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int ret ;
    ret=alarm(3) ; /*调用alarm定时器函数*/
    pause() ;
    printf("I have been waken up.\n") ;
    return 0 ;
}
```

编译 gcc pause.c －o pause。

执行./pause，执行结果如下：

```
Alarm clock
```

5. sleep和abort

(1) sleep函数原型

sleep（让进程暂停执行一段时间）	
所需头文件	#include <unistd.h>
函数说明	sleep()会令目前的进程暂停，直到达到参数 seconds 所指定的时间，或是被信号所中断
函数原型	unsigned int sleep(unsigned int seconds)
函数传入值	seconds：睡眠的秒数
函数返回值	若进程暂停到参数seconds所指定的时间，则返回 0；若有信号中断则返回剩余秒数

(2) abort函数原型

abort（以异常方式结束进程）	
所需头文件	#include <stdlib.h>
函数说明	此函数将 SIGABRT 信号给调用进程，此信号将引起调用进程的异常终止。此时所有已打开的文件流会自动关闭，所有的缓冲区数据会自动写回。
函数原型	void abort(void)

(3) sleep、abort函数举例

sys_sleep.c源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    system("pwd") ;
    sleep(9) ; /*wait 9 second*/
    printf("Calling abort()\n");
        abort();
    printf("abort() after\n") ;
        return 0; /* This is never reached */
}
```

编译 gcc sys_sleep.c -o sys_sleep。

PDF-XChange Click to buy NOW! www.docu-track.com

执行./sys_sleep，执行结果如下：

```
/home/zjkf/public/signal
Calling abort()
```

6. 信号的发送与捕捉简要总结

对上述信号的发生与捕捉函数简要总结说明如下：

kill函数可以向有用户权限的任何进程发送信号，通常用kill函数来结束进程。

与kill函数不同的是，raise函数只向进程自身发送信号。

使用alarm函数可以设置一个时间值(闹铃时间)，在将来的某个时刻该时间值超过时发送信号。

pause函数使调用进程挂起直至捕捉到一个信号。

带格式的：项目符号和编号

1.3.3 信号的处理

信号是与一定的进程相联系的。也就是说，一个进程可以决定在进程中对哪些信号进行什么样的处理。例如，一个进程可以忽略某些信号而只处理其他一些信号；另外，一个进程还可以选择如何处理信号。总之，这些总与特定的进程相联系的。因此，首先要建立其信号和进程的对应关系，这就是信号的安装登记。

Linux主要有两个函数实现信号的安装登记：signal和sigaction。其中signal在系统调用的基础上实现，是库函数。它只有两个参数，不支持信号传递信息，主要是用于前 32 个非实时信号的安装；而sigaction是较新的函数（由两个系统调用实现：sys_signal以及sys_rt_sigaction），有三个参数，支持信号传递信息，主要用来与sigqueue系统调用配合使用。当然，sigaction同样支持非实时信号的安装，sigaction优于signal主要体现在支持信号带有参数。

对于应用程序自行处理的信号来说，信号的生命周期要经过信号的安装登记、信号集操作、信号的发送和信号的处理四个阶段。信号的安装登记指的是在应用程序中，安装对此信号的处理方法。信号集操作的作用是用于对指定的一个或多个信号进行信号屏蔽，此阶段对有些应用程序来说并不需要。信号的发送指的是发送信号，可以通过硬件（如在终端上按下Ctrl-C）发送的信号和软件（如通过kill函数）发送的信号。信号的处理指的是操作系统对接收信号进程的处理，处理方法是先检查信号集操作函数是否对此信号进行屏蔽，如果没有屏蔽，操作系统将按信号安装函数中登记注册的处理函数完成对此进程的处理。

1. signal函数

（1）函数说明

在signal函数中，有两个形参，分别代表需要处理的信号编号值和处理信号函数的指针。它主要是用于前 32 种非实时信号的处理，不支持信号的传递信息。但是由于使用简单，易于理解，因此在许多场合被程序员使用。

对于Unix系统来说，使用signal函数时，自定义处理信号函数执行一次后失效，对该信号的处理回到默认处理方式。下面以一个例子进行说明，例如一程序中使用signal(SIGQUIT, my_func)函数调用，其中my_func是自定义函数。应用进程收到SIGQUIT信号时，会跳转到自定义处理信号函数my_func处执行，执行后信号注册函数my_func失效，对SIGQUIT信号的处理回到操作系统的默认处理方式，当应用进程再次收到SIGQUIT信号时，会按操作系统默认的处理方式进行处理（即不再执行my_func处理函数）。而在Linux系统中，signal函数已被改写，

由sigaction函数封装实现，则不存在上述问题。

（2）signal函数原型

signal（设置信号处理方式）		
所需头文件	#include <signal.h>	
函数说明	设置信号处理方式。signal()会依参数 signum 指定的信号编号来设置该信号的处理函数。当指定的信号到达时就会跳转到参数 handler 指定的函数执行	
函数原型	void (*signal(int signum,void(* handler)(int)))(int)	
函数传入值	signum	指定信号编号
	handle	SIG_IGN：忽略参数 signum 指定的信号
		SIG_DFL：将参数 signum 指定的信号重设为核心预设的信号处理方式，即采用系统默认方式处理信号
		自定义信号函数处理指针
函数返回值	成功	返回先前的信号处理函数指针
	出错	SIG_ERR(-1)
附加说明	在 Unix 环境中，在信号发生跳转到自定的 handler 处理函数执行后，系统会自动将此处理函数换回原来系统预设的处理方式，如果要改变此情形请改用 sigaction 函数。在 Linux 环境中不存在此问题	

signal函数原型比较复杂，如果使用下面的typedef，则可使其简化。

```
typedef void sign(int);
sign *signal(int, handler *);
```

可见，该函数原型首先整体指向一个无返回值带一个整型参数的函数指针，也就是信号的原始配置函数。接着该原型又带有两个参数，其中的第二个参数可以是用户自定义的信号处理函数的函数指针。对这个函数格式可以不理解，但需要学会模仿使用。

（3） signal函数使用实例

该示例表明了如何使用signal函数进行安装登记信号处理函数。当该信号发生时，登记的信号处理函数会捕捉到相应的信号，并做出给定的处理。这里，my_func就是信号处理的函数指针。读者还可以将my_func改为SIG_IGN或SIG_DFL查看运行结果。

signal.c源代码如下：

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
/*自定义信号处理函数*/
void my_func(int sign_no)
{
    if(sign_no==SIGINT)
        printf("I have get SIGINT\n");
    else if(sign_no==SIGQUIT)
        printf("I have get SIGQUIT\n");
}
int main()
{
    printf("Waiting for signal SIGINT or SIGQUIT \n ");
    /*发出相应的信号，并跳转到信号处理函数处*/
    signal(SIGINT, my_func);
    signal(SIGQUIT, my_func);
    pause();
```

```
    pause();
    exit(0);
}
```

编译 gcc signal.c –o signal。

执行 ./signal，执行结果如下：

```
Waiting for signal SIGINT or SIGQUIT
I have get SIGINT       /*按下Ctrl+C,操作系统就会向进程发送SIGINT信号*/
I have get SIGQUIT      /*按下Ctrl-\（退出）,操作系统就会向进程发送SIGQUIT信号*/
```

2. sigaction函数

（1）sigaction函数原型

sigaction函数用来查询和设置信号处理方式，它是用来替换早期的signal函数。sigaction函数原型及说明如下：

<table>
<tr><th colspan="3">sigaction（查询和设置信号处理方式）</th></tr>
<tr><td>所需头文件</td><td colspan="2">#include <signal.h></td></tr>
<tr><td>函数说明</td><td colspan="2">sigaction()会依参数 signum 指定的信号编号来设置该信号的处理函数</td></tr>
<tr><td>函数原型</td><td colspan="2">int sigaction(int signum,const struct sigaction *act ,struct sigaction *oldact)</td></tr>
<tr><td rowspan="3">函数传入值</td><td>signum</td><td>可以指定 SIGKILL 和 SIGSTOP 以外的所有信号</td></tr>
<tr><td>act</td><td>参数结构 sigaction 定义如下
struct sigaction
{
void (*sa_handler) (int);
void (*sa_sigaction)(int, siginfo_t *, void *);
sigset_t sa_mask;
int sa_flags;
void (*sa_restorer) (void);
}
① sa_handler：此参数和 signal()的参数 handler 相同，此参数主要用来对信号旧的安装函数 signal()处理形式的支持
② sa_sigaction：新的信号安装机制，处理函数被调用的时候，不但可以得到信号编号，而且可以获悉被调用的原因以及产生问题的上下文的相关信息。
③ sa_mask：用来设置在处理该信号时暂时将 sa_mask 指定的信号搁置
④ sa_restorer： 此参数没有使用
⑤ sa_flags：用来设置信号处理的其他相关操作，下列的数值可用。可用 OR 运算（|）组合
• A_NOCLDSTOP:如果参数 signum 为 SIGCHLD，则当子进程暂停时并不会通知父进程
• SA_ONESHOT/SA_RESETHAND:当调用新的信号处理函数前，将此信号处理方式改为系统预设的方式
• SA_RESTART:被信号中断的系统调用会自行重启
• SA_NOMASK/SA_NODEFER:在处理此信号未结束前不理会此信号的再次到来
• SA_SIGINFO：信号处理函数是带有三个参数的 sa_sigaction</td></tr>
<tr><td>oldact</td><td>如果参数 oldact 不是 NULL 指针，则原来的信号处理方式会由此结构 sigaction 返回</td></tr>
<tr><td rowspan="2">函数返回值</td><td colspan="2">成功：0</td></tr>
<tr><td colspan="2">出错：-1，错误原因存于 error 中</td></tr>
</table>

附加说明	信号处理安装的新旧两种机制： ① 使用旧的处理机制：struct sigaction act； act.sa_handler=handler_old; ② 使用新的处理机制：struct sigaction act; act.sa_sigaction=handler_new; 并设置 sa_flags 的 SA_SIGINFO 位
错误代码	EINVAL：参数 signum 不合法，或是企图拦截 SIGKILL/SIGSTOP 信号 EFAULT：参数 act，oldact 指针地址无法存取 EINTR：此调用被中断

（2）sigaction函数使用实例

sigaction.c源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
void new_op(int, siginfo_t *, void *);
int main(int argc,char**argv)
{
    struct sigaction act;
    int sig;
    sig=atoi(argv[1]);
    sigemptyset(&act.sa_mask);
    act.sa_flags=SA_SIGINFO;
    act.sa_sigaction=new_op;
    if(sigaction(sig,&act,NULL) < 0)
    {
        perror("install sigal error");
        return -1 ;
    }
    while(1)
    {
        sleep(2);
        printf("wait for the signal\n");
    }

    return 0 ;
}
void new_op(int signum,siginfo_t *info,void *myact)
{
    printf("receive signal %d\n", signum);
    sleep(5);
}
```

编译 gcc sigaction.c -o sigaction。

执行 ./sigaction 2，执行结果如下：

```
wait for the signal
receive signal 2        /*按下Ctrl+C */
退出                    /*按下Ctrl-\ */
```

3. 信号集操作函数

由于有时需要把多个信号当作一个集合进行处理，这样信号集就产生了，信号集用来描述一类信号的集合，Linux所支持的信号可以全部或部分的出现在信号集中。信号集操作函数最常用的地方就是用于信号屏蔽。比如有时候希望某个进程正确执行，而不想进程受到一些信号的影响，此时就需要用到信号集操作函数完成对这些信号的屏蔽。

信号集操作函数按照功能和使用顺序分为三类，分别为创建信号集函数，设置信号屏蔽位函数和查询被搁置（未决）的信号函数。创建信号集函数只是创建一个信号的集合，设置信号屏蔽位函数对指定信号集中的信号进行屏蔽，查询被搁置的信号函数是用来查询当前“未决”的信号集。信号集函数组并不能完成信号的安装登记工作，信号的安装登记需要通过sigaction函数或signal函数来完成。

查询被搁置的信号是信号处理的后续步骤，但不是必需的。由于有时进程在某时间段内要求阻塞一些信号，程序完成特定工作后解除对该信号阻塞，这个时间段内被阻塞的信号称为“未决”信号。这些信号已经产生，但没有被处理，sigpending函数用来检测进程的这些“未决”信号，并进一步决定对它们做何种处理（包括不处理）。

（1） 创建信号集函数

创建信号集函数有如下 5 个：

① sigemptyset：初始化信号集合为空。
② sigfillset：把所有信号加入到集合中，信号集中将包含Linux支持的 64 种信号。
③ sigaddset：将指定信号加入到信号集合中去。
④ sigdelset：将指定信号从信号集中删去。
⑤ sigismember：查询指定信号是否在信号集合之中。

创建信号集合函数原型	
所需头文件	#include <signal.h>
函数原型	int sigemptyset(sigset_t *set)
	int sigfillset(sigset_t *set)
	int sigaddset(sigset_t *set,int signum)
	int sigdelset(sigset_t *set,int signum)
	int sigismember(sigset_t *set,int signum)
函数传入值	set：信号集
	signum：指定信号值
函数返回值	成功：0（sigismember 函数例外，成功返回 1，失败返回 0）
	出错：-1，错误原因存于 error 中

（2） 设置信号屏蔽位函数

每个进程都有一个用来描述哪些信号递送到进程时将被阻塞的信号集，该信号集中的所有信号在递送到进程后都将被阻塞。调用函数sigprocmask可设定信号集内的信号阻塞或不阻塞。其函数原型及说明如下：

sigprocmask（设置信号屏蔽位）		
所需头文件	#include <signal.h>	
函数原型	int sigprocmask(int how,const sigset_t *set,sigset_t *oset)	
函数传入值	how(决定函数的操作方式)	SIG_BLOCK：增加一个信号集合到当前进程的阻塞集合之中
		SIG_UNBLOCK：从当前的阻塞集合之中删除一个信号集合
		SIG_SETMASK：将当前的信号集合设置为信号阻塞集合
	set：指定信号集	
	oset：信号屏蔽字	
函数返回值	成功：0	
	出错：-1，错误原因存于 error 中	

PDF-XChange Click to buy NOW! www.docu-track.com

（3） 查询被搁置（未决）信号函数

sigpending函数用来查询“未决”信号。其函数原型及说明如下：

sigpending（查询未决信号）	
所需头文件	#include <signal.h>
函数说明	将被搁置的信号集合由参数 set 指针返回
函数原型	int sigpending(sigset_t *set)
函数传入值	set：要检测信号集
函数返回值	成功：0
	出错：-1，错误原因存于 error 中
错误代码	EFAULT：参数 set 指针地址无法存取 EINTR：此调用被中断

（4） 对信号集操作函数的使用方法

对信号集操作函数的使用方法和顺序如下：

① 使用signal或sigaction函数安装和登记信号的处理。
② 使用sigemptyset等定义信号集函数完成对信号集的定义。
③ 使用sigprocmask函数设置信号屏蔽位。
④ 使用sigpending函数检测未决信号，非必需步骤。

（5） 信号集操作函数使用实例

该实例首先使用sigaction函数对SIGINT信号进行安装登记，安装登记使用了新旧两种机制，其中#if 0 进行注释掉的部分为信号安装的新机制。接着程序把SIGQUIT、SIGINT两个信号加入信号集，并把该信号集设为阻塞状态。程序开始睡眠 30 秒，此时用户按下Ctrl+C，程序将测试到此未决信号（SIGINT）；随后程序再睡眠 30 秒后对SIGINT信号解除阻塞，此时将处理SIGINT登记的信号函数my_func。最后可以用SIGQUIT（Ctrl+\）信号结束进程执行。

sigset.c源代码如下：

```
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
/*自定义的信号处理函数*/
#if 0
void my_funcnew(int signum, siginfo_t *info,void *myact)
#endif
void my_func(int signum)
{
    printf("If you want to quit,please try SIGQUIT\n");
}
int main()
{
    sigset_t set, pendset;
    struct sigaction action1,action2;

    /*设置信号处理方式*/
    sigemptyset(&action1.sa_mask);

#if 0 /*信号新的安装机制*/
    action1.sa_flags= SA_SIGINFO;
```

PDF-XChange
Click to buy NOW!
www.docu-track.com

```
    action1.sa_sigaction=my_funcnew;
#endif
    /*信号旧的安装机制*/
    action1.sa_flags= 0;
    action1.sa_handler=my_func;
    sigaction(SIGINT,&action1,NULL);

    /*初始化信号集为空*/
    if(sigemptyset(&set)<0)
    {
        perror("sigemptyset");
        return -1 ;
    }
    /*将相应的信号加入信号集*/
    if(sigaddset(&set,SIGQUIT)<0)
    {
        perror("sigaddset");
        return -1 ;
    }
    if(sigaddset(&set,SIGINT)<0)
    {
        perror("sigaddset");
        return -1 ;
    }

    /*设置信号集屏蔽字*/
    if(sigprocmask(SIG_BLOCK,&set,NULL)<0)
    {
        perror("sigprocmask");
        return -1 ;
    }
    else
    {
        printf("blocked\n");
    }

    /*测试信号是否加入该信号集*/
    if(sigismember(&set,SIGINT)){
            printf("SIGINT in set\n") ;
    }

    sleep( 30 ) ;
    /*测试未决信号*/
    if ( sigpending(&pendset) <0 )
    {
        perror("get pending mask error");
    }
    if(sigismember(&pendset, SIGINT) )
    {
        printf("signal SIGINT is pending\n");
    }

    sleep(30) ;
    if(sigprocmask(SIG_UNBLOCK,&set,NULL)<0)
    {
        perror("sigprocmask");
        return -1 ;
    }
    else
```

```
        printf("unblock\n");

    while(1)
    {
        sleep(1) ;
    }

    return 0 ;
}
```

编译 gcc sigset.c -o sigset。

执行 ./sigset，执行结果如下：

```
blocked
SIGINT in set   /*按下Ctrl+C */
signal SIGINT is pending
If you want to quit,please try SIGQUIT  /*按下Ctrl+C */
退出
```

带格式的：项目符号和编号

第2章 System V 进程间通讯

System V，曾经也被称为 AT&T System V，是Unix操作系统众多版本中的一支。它最初由AT&T开发，在 1983 年第一次发布。一共发行了 4 个System V的主要版本：版本 1、2、3 和 4。System V Release 4，或者称为SVR4，是最成功的版本，成为一些Unix共同特性的源头。System V是AT&T 的第一个商业Unix版本（Unix System III）的加强。System V IPC（Inter-Process Communication，进程间通信）包括三种进程通信方式，即消息队列、信号量和共享内存。这三种方式完全被Linux系统继承和兼容。

带格式的：项目符号和编号

2.1 System V IPC的键值

消息队列、信号灯、共享内存常用在Linux服务端编程的进程间通信环境中。而此三类编程函数在实际项目中都是用System V IPC函数实现的。System V IPC函数名称和说明如下表 15-1 所示。

表 15-1 System V IPC函数

	消息队列	信号灯	共享内存区
头文件	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
创建或打开 IPC 函数	msgget	semget	shmget
控制 IPC 操作的函数	msgctl	semctl	shmctl
IPC 操作函数	msgsnd msgrcv	semop	shmat shmdt

1. key_t键和ftok函数

函数ftok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键值（也称IPC key键值）。ftok函数原型及说明如下：

ftok（把一个已存在的路径名和一个整数标识符转换成 IPC 键值）	
所需头文件	#include <sys/types.h> #include <sys/ipc.h>
函数说明	把从 pathname 导出的信息与 id 的低序 8 位组合成一个整数 IPC 键
函数原型	key_t ftok(const char *pathname, int proj_id)
函数传入值	pathname：指定的文件，此文件必须存在且可存取
	proj_id：计划代号（project ID）

函数返回值	成功：返回 key_t 值（即 IPC 键值）
	出错：-1，错误原因存于 error 中
附加说明	key_t 一般为 32 位的 int 型的重定义

ftok的典型实现是调用stat函数，然后组合以下三个值：

① pathname所在的文件系统的信息（stat结构的st_dev成员）。

② 该文件在本文件系统内的索引节点号(stat结构的st_ino成员)。

③ proj_id的低序 8 位（不能为 0）。

上述三个值的组合产生一个 32 位键。

2. ftok函数代码举例

ftok.c源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/stat.h>
#include <unistd.h>
int main(int argc, char **argv)
{
    struct stat stat1 ;
    if ( argc != 2 )
    {
        printf("usage: ftok < pathname >" ) ;
        exit(1) ;
    }
    stat( argv[1], &stat1 ) ;
    printf("st_dev:%lx, st_ino:%lx, key:%x\n",  \
(unsigned long)stat1.st_dev, (unsigned long)stat1.st_ino , ftok(argv[1],0x579 )) ;
    printf("st_dev:%lx, st_ino:%lx, key:%x\n",  \
(unsigned long)stat1.st_dev, (unsigned long)stat1.st_ino , ftok(argv[1],0x118 )) ;
    printf("st_dev:%lx, st_ino:%lx, key:%x\n",  \
(unsigned long)stat1.st_dev, (unsigned long)stat1.st_ino , ftok(argv[1],0x22 )) ;
    printf("st_dev:%lx, st_ino:%lx, key:%x\n",  \
(unsigned long)stat1.st_dev, (unsigned long)stat1.st_ino , ftok(argv[1],0x33 )) ;
    exit(0) ;
}
```

编译 gcc ftok.c –o ftok

运行 ./ftok /tmp，执行结果如下：

```
st_dev:801, st_ino:4db21, key:7901db21
st_dev:801, st_ino:4db21, key:1801db21
st_dev:801, st_ino:4db21, key:2201db21
st_dev:801, st_ino:4db21, key:3301db21
st_dev:801, st_ino:4db21, key:4401db21
```

从上面程序可以看出，通过ftok返回的是根据文件（pathname）信息和计划编号（proj_id）合成的IPC key键值，从而避免用户使用key值的冲突。proj_id值的意义让一个文件也能生成多个IPC key键值。ftok利用同一文件最多可得到IPC key键值 0xff（即 256）个，因为ftok只取proj_id值二进制的后 8 位，即 16 进制的后两位与文件信息合成IPC key键值。

3. IPC键值与IPC标识符

（1）key值选择方式

对于key值，应用程序有如下三种选择：

① 调用ftok，给它传递pathname和proj_id，操作系统根据两者合成key值。

② 指定key为IPC_PRIVATE，内核保证创建一个新的、唯一的IPC对象，IPC标识符与内存中的标识符不会冲突。IPC_PRIVATE为宏定义，其值等于0。

③ 指定key为大于0的常数，这需要用户自行保证生成的IPC key值不与系统中存在的冲突，而前两种是操作系统保证的。

（2）IPC标识符

给semget、msgget、shmget传入key值，它们返回的都是相应的IPC对象标识符。注意IPC键值和IPC标识符是两个概念，后者是建立在前者之上。图15-1画出了从IPC键值生成IPC标识符图，其中key为IPC键值，由ftok函数生成；ipc_id为IPC标识符，由semget、msgget、shmget函数生成。ipc_id在信号量函数中称为semid，在消息队列函数中称为msgid，在共享内存函数中称为shmid，它们表示的是各自IPC对象标识符。

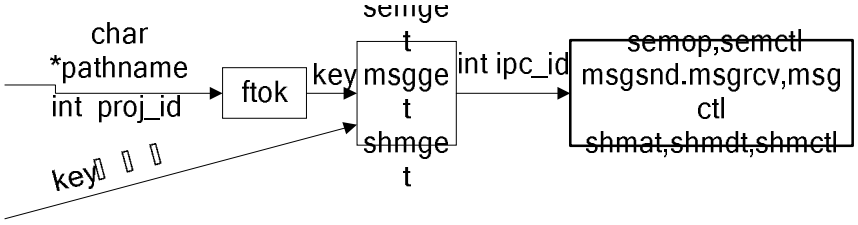

图15-1 从IPC键值生成IPC标识符图

4. ipc_perm结构说明

系统为每一个IPC对象保存一个ipc_perm结构体，该结构说明了IPC对象的权限和所有者，并确定了一个IPC操作是否可以访问该IPC对象。

```
struct ipc_perm {
     key_t    key ;          /* 此IPC对象的key键 */
     uid_t    uid ;          /* 此IPC对象用户ID */
     gid_t    gid ;          /* 此IPC对象组ID */
     uid_t    cuid ;         /* IPC对象创建进程的有效用户ID */
     gid_t    cgid ;         /* IPC对象创建进程的有效组ID */
     mode_t    mode ;        /* 此IPC的读写权限 */
     ulong_t   seq ;         /* IPC对象的序列号 */
} ;
```

表15-2列出了ipc_perm中mode的含义，其含义与文件访问权限相似。当调用IPC对象创建函数（semget、msgget、shmget）时，会对ipc_perm结构变量的每一个成员赋值，其中mode的值来源于IPC对象创建函数最右边的形参flag（msgget中为msgflg、semget中为semflg、shmget中shmflg）。如需修改这几个成员变量则需调用相应的控制函数（msgctl、semctl、shmctl）。

表15-2 IPC对象存取权限表

ipc_perm中mode的含义			
操作者	读	写	可读可写
用户	0400	0200	0600
组	0040	0020	0060
其他	0004	0002	0006

5. IPC对象的创建权限

msgget、semget、shmget函数最右边的形参flag（msgget中为msgflg、semget中为semflg、shmget中shmflg）为IPC对象创建权限，三种xxxget函数中flag的作用基本相同。

IPC对象创建权限(即flag）格式为 0xxxxx，其中 0 表示 8 位制，低三位为用户、属组、其他的读、写、执行权限(执行位不使用)，其含义与ipc_perm的mode相同，具体含义见表15-2。在这里姑且把IPC对象创建权限格式的低三位称为“ IPC对象存取权限”。如 0600 代表只有此用户下的进程才有可读可写权限。IPC对象存取权限常与下面IPC_CREAT、IPC_EXCL两种标志进行或（|）运算完成对IPC对象创建的管理，在这里姑且把IPC_CREAT、IPC_EXCL两种标志称为IPC创建模式标志。下面是两种创建模式标志在<sys/ipc.h>头文件中的宏定义。

```
#define IPC_CREAT    01000    /* Create key if key does not exist. */
#define IPC_EXCL     02000     /* Fail if key exists.  */
```

综上所述，flag标志由两部分组成，一为IPC对象存取权限（含义同ipc_perm中的mode），一为IPC对象创建模式标志（IPC_CREAT、IPC_EXCL），两者进行|运算合成IPC对象创建权限。

6. 创建或打开IPC对象流程图

semget、msgget、shmget函数的作用是创建一个新的IPC对象或者访问一个已存在的IPC对象。其创建或访问的规则如下：

① 指定key为IPC_PRIVATE操作系统保证创建一个唯一的IPC对象。

② 设置flag参数的IPC_CREAT位但不设置它的IPC_EXCL位时，如果所指定key键的IPC对象不存在，那就是创建一个新的对象；否则返回该对象。

③ 同时设置flag的IPC_CREAT和IPC_EXCL位时，如果所指定key键的IPC对象不存在，那就创建一个新的对象；否则返回一个EEXIST错误，因为该对象已存在。

综上所述，flag创建模式标志的作用如下表 15-3 所示。

表 15-3 三种xxxget函数flag的创建模式标志作用表

flag 创建模式标志	不存在	已存在
无特殊标志	出错，errno=ENOENT	成功，引用已存在对象
IPC_CREAT	成功，创建新对象	成功，引用已存在对象
IPC_CREAT|IPC_EXCL	成功，创建新对象	出错，errno=EEXIST

下图 15-2 画出了semget、msgget、shmget创建或打开一个IPC对象的逻辑流程图，它说明了内核创建和访问IPC对象的流程。

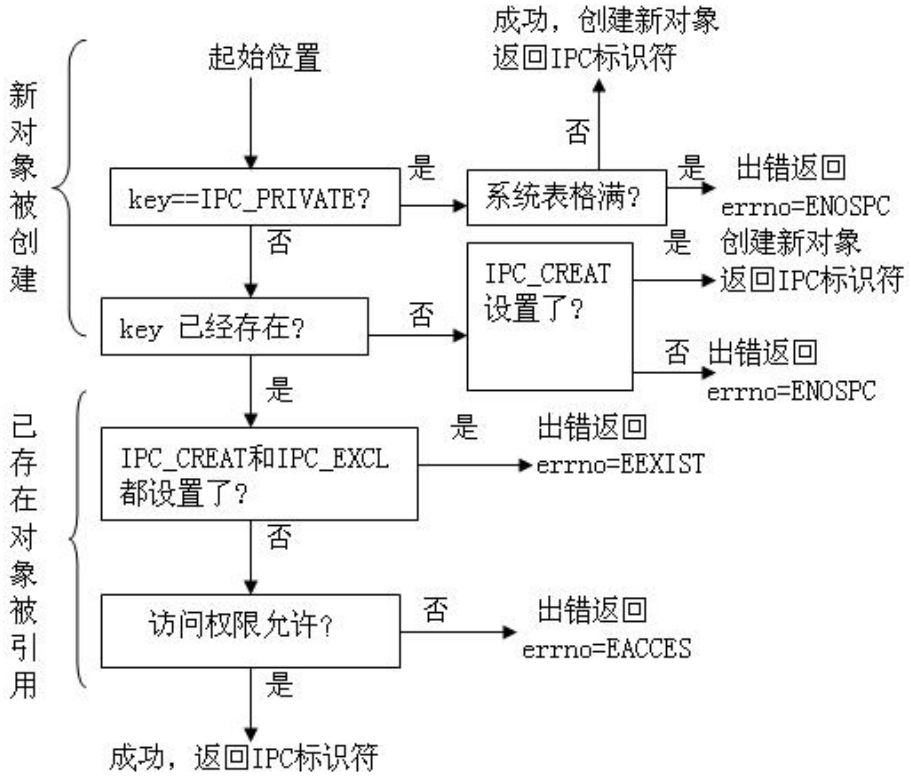

图 15-2 semget、msgget、shmget创建或打开一个IPC对象的逻辑流程图

使用semget、msgget、shmget创建一个IPC对象时，需要指定flag标志，在key不等于IPC_PRIVATE情况下，flag标志决定了创建方式和创建后IPC对象的存取权限。在key等于IPC_PRIVATE情况下，flag标志决定了创建后IPC对象的存取权限。如果只是引用一个已经存在的IPC对象只需把flag标志设为 0 即可。

带格式的：项目符号和编号

2.2 消息队列

一个或多个进程向消息队列写入消息，另外的一个或多个进程从消息队列中读取消息，这种进程间通信机制通常使用在请求/服务模型中，请求进程向服务进程发送请求的消息，服务进程读取消息并执行相应操作。在许多微内核结构的操作系统中，内核和各组件之间的基本通信方式就是消息队列。例如，在Minix操作系统中，内核、I/O任务、服务器进程和用户进程之间就是通过消息队列实现通信的。

Linux中的消息可以被描述成在内核地址空间的一个内部链表，每一个消息队列都有唯一的一个消息队列标识号。Linux为系统中所有的消息队列维护一个msgque链表，该链表中的每个指针指向一个msgid_ds结构，该结构完整地描述了一个个消息队列。

带格式的：项目符号和编号

2.2.1 消息队列简要说明

消息队列常使用在一个进程向消息队列发消息，另一个进程向消息队列接收消息的场合。接收消息分两种情况，即接收最开始的一条消息或特定类型的消息。

1. 消息队列内核数据结构

当在系统中创建每一个消息队列时，内核创建、存储及维护着 msqid_ds 结构的一个实例。msqid_ds 结构的具体说明如下：

```
/* 在系统中的每一个消息队列对应一个msqid_ds 结构 */
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;   /* 队列上第一条消息，即链表头*/
    struct msg *msg_last;    /* 队列中的最后一条消息，即链表尾 */
    time_t msg_stime;        /* 发送给队列的最后一条消息的时间 */
    time_t msg_rtime;        /* 从消息队列接收到的最后一条消息的时间 */
    time_t msg_ctime;        /* 最后修改队列的时间*/
    ushort msg_cbytes;       /*队列上所有消息总的字节数 */
    ushort msg_qnum;         /*在当前队列上消息的个数 */
    ushort msg_qbytes;       /* 队列最大的字节数 */
    ushort msg_lspid;        /* 发送最后一条消息的进程的pid */
    ushort msg_lrpid;        /* 接收最后一条消息的进程的pid */
};
```

2. 消息队列通信原理图

消息队列常使用在两进程收发信息的场合。下图 15-4 为消息队列通信原理图，一进程向消息队列发送消息，而另一进程从消息队列收取消息。

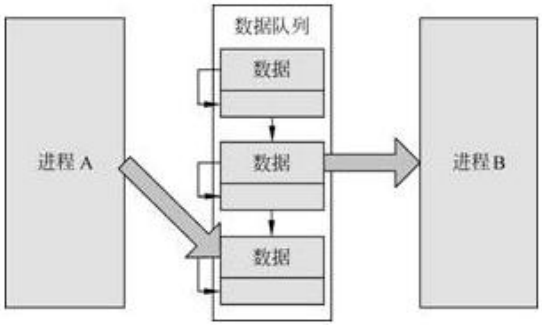

图 15-4 消息队列通信原理图

带格式的：项目符号和编号

2.2.2 消息队列函数

消息队列函数由msgget、msgctl、msgsnd、msgrcv四个函数组成。下面的表格列出了这四个函数的函数原型及其具体说明。

1. msgget函数原型

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

		IPC_CREAT|IPC_EXCL：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则报错
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	
附加说明	上述 msgflg 参数为模式标志参数，使用时需要与 IPC 对象存取权限（如 0600）进行|运算来确定消息队列的存取权限	
错误代码	EACCES：指定的消息队列已存在，但调用进程没有权限访问它 EEXIST：key指定的消息队列已存在，而msgflg中同时指定IPC_CREAT和IPC_EXCL标志 ENOENT：key指定的消息队列不存在同时msgflg中没有指定IPC_CREAT标志 ENOMEM：需要建立消息队列，但内存不足 ENOSPC：需要建立消息队列，但已达到系统的限制	

如果用msgget创建了一个新的消息队列对象时，则msqid_ds结构成员变量的值设置如下：

- msg_qnum、msg_lspid、msg_lrpid、 msg_stime、msg_rtime设置为 0。
- msg_ctime设置为当前时间。
- msg_qbytes设成系统的限制值。
- msgflg的读写权限写入msg_perm.mode中。
- msg_perm结构的uid和cuid成员被设置成当前进程的有效用户ID，gid和cuid成员被设置成当前进程的有效组ID。

2. msgctl函数原型

msgctl (获取和设置消息队列的属性)		
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>	
函数说明	获取和设置消息队列的属性	
函数原型	int msgctl(int msqid, int cmd, struct msqid_ds *buf)	
函数传入值	msqid	消息队列标识符
	cmd	IPC_STAT:获得msgid的消息队列头数据到buf中
		IPC_SET：设置消息队列的属性，要设置的属性需先存储在buf中，可设置的属性包括：msg_perm.uid、msg_perm.gid、msg_perm.mode以及msg_qbytes
		buf：消息队列管理结构体，请参见消息队列内核结构说明部分
函数返回值	成功：0	
	出错：-1，错误原因存于 error 中	
错误代码	EACCESS：参数cmd为IPC_STAT，确无权限读取该消息队列 EFAULT：参数buf指向无效的内存地址 EIDRM：标识符为msqid的消息队列已被删除 EINVAL：无效的参数 cmd 或 msqid EPERM：参数 cmd 为 IPC_SET 或 IPC_RMID，却无足够的权限执行	

3. msgsnd函数原型

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
		IPC_NOERROR：若发送的消息大于size字节，则把该消息截断，截断部分将被丢弃，且不通知发送进程。
函数返回值	成功：0	
	出错：-1，错误原因存于error中	
错误代码	EAGAIN：参数msgflg设为IPC_NOWAIT，而消息队列已满 EIDRM：标识符为msqid的消息队列已被删除 EACCESS：无权限写入消息队列 EFAULT：参数msgp指向无效的内存地址 EINTR：队列已满而处于等待情况下被信号中断 EINVAL：无效的参数msqid、msgsz或参数消息类型type小于0	

msgsnd()为阻塞函数，当消息队列容量满或消息个数满会阻塞。消息队列已被删除，则返回EIDRM错误；被信号中断返回E_INTR错误。

如果设置IPC_NOWAIT消息队列满或个数满时会返回-1，并且置EAGAIN错误。msgsnd()解除阻塞的条件有以下三个条件：

① 不满足消息队列满或个数满两个条件，即消息队列中有容纳该消息的空间。

② msqid代表的消息队列被删除。

③ 调用msgsnd函数的进程被信号中断。

4. msgrcv函数原型

msgrcv (从消息队列读取消息)		
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h>	
函数说明	从标识符为msqid的消息队列读取消息并存于msgp中，读取后把此消息从消息队列中删除	
函数原型	ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);	
函数传入值	msqid	消息队列标识符
	msgp	存放消息的结构体，结构体类型要与msgsnd函数发送的类型相同
	msgsz	要接收消息的大小，不含消息类型占用的4个字节
	msgtyp	0：接收第一个消息
		>0：接收类型等于msgtyp的第一个消息
		<0：接收类型等于或者小于msgtyp绝对值的第一个消息
	msgflg	0：阻塞式接收消息，没有该类型的消息msgrcv函数一直阻塞等待
		IPC_NOWAIT：如果没有返回条件的消息调用立即返回，此时错误码为ENOMSG
		IPC_EXCEPT：与msgtype配合使用返回队列中第一个类型不为msgtype的消息
		IPC_NOERROR：如果队列中满足条件的消息内容大于所请求的size字节，则把该消息截断，截断部分将被丢弃
函数返回值	成功：实际读取到的消息数据长度	

	出错：-1，错误原因存于 error 中
错误代码	E2BIG：消息数据长度大于msgsz而msgflag没有设置IPC_NOERROR EIDRM：标识符为msqid的消息队列已被删除 EACCESS：无权限读取该消息队列 EFAULT：参数msgp指向无效的内存地址 ENOMSG：参数msgflg设为IPC_NOWAIT，而消息队列中无消息可读 EINTR：等待读取队列内的消息情况下被信号中断

msgrcv()解除阻塞的条件有以下三个：

① 消息队列中有了满足条件的消息。

② msqid代表的消息队列被删除。

③ 调用msgrcv()的进程被信号中断。

带格式的：项目符号和编号

2.2.3 消息队列使用程序范例

1. 消息队列控制范例

msgctl.c源代码如下：

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <error.h>
#define TEXT_SIZE  512
struct msgbuf
{
    long mtype ;
    char mtext[TEXT_SIZE] ;
}  ;
int main(int argc, char **argv)
{
    int msqid ;
    struct msqid_ds info ;
    struct msgbuf buf ;
    struct msgbuf buf1 ;
    int flag ;
    int sendlength, recvlength ;

    msqid = msgget( IPC_PRIVATE, 0666 ) ;
    if ( msqid < 0 )
    {
        perror("get ipc_id error") ;
        return -1 ;
    }

    buf.mtype = 1 ;
    strcpy(buf.mtext, "happy new year!") ;
    sendlength = sizeof(struct msgbuf) - sizeof(long) ;
    flag = msgsnd( msqid, &buf, sendlength , 0 ) ;
    if ( flag < 0 )
    {
        perror("send message error") ;
        return -1 ;
    }
    buf.mtype = 3 ;
    strcpy(buf.mtext, "good bye!") ;
    sendlength = sizeof(struct msgbuf) - sizeof(long) ;
```

```
    flag = msgsnd( msqid, &buf, sendlength , 0 ) ;
    if ( flag < 0 )
    {
        perror("send message error") ;
        return -1 ;
    }

    flag = msgctl( msqid, IPC_STAT, &info ) ;
    if ( flag < 0 )
    {
        perror("get message status error") ;
        return -1 ;
    }
    printf("uid:%d, gid = %d, cuid = %d, cgid= %d\n" ,
        info.msg_perm.uid,  info.msg_perm.gid,  info.msg_perm.cuid,  info.msg_perm.cgid  ) ;
    printf("read-write:%03o, cbytes = %lu, qnum = %lu, qbytes= %lu\n" ,
        info.msg_perm.mode&0777, info.msg_cbytes, info.msg_qnum, info.msg_qbytes ) ;
    system("ipcs -q") ;
    recvlength = sizeof(struct msgbuf) - sizeof(long) ;
    memset(&buf1, 0x00, sizeof(struct msgbuf)) ;

    flag = msgrcv( msqid, &buf1, recvlength ,3,0 ) ;
    if ( flag < 0 )
    {
        perror("recv message error") ;
        return -1 ;
    }
    printf("type=%d, message=%s\n", buf1.mtype, buf1.mtext) ;

    flag = msgctl( msqid, IPC_RMID,NULL) ;
    if ( flag < 0 )
    {
        perror("rm message queue error") ;
        return -1 ;
    }
    system("ipcs -q") ;

   return 0 ;
}
```

编译 gcc msgctl.c –o msgctl。

执行 ./msg，执行结果如下:

```
uid:1008, gid = 1003, cuid = 1008, cgid= 1003
read-write:666, cbytes = 1024, qnum = 2, qbytes= 163840

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00000000 65536      zjkf       666        1024         2

type=3, message=good bye!

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

2. 两进程通过消息队列收发消息

（1）发送消息队列程序

msgsnd.c源代码如下：

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

PDF-XChange
Click to buy NOW!
www.docu-track.com

```
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include  <time.h>
#define TEXT_SIZE  512
struct msgbuf
{
    long mtype ;
    int  status ;
    char time[20] ;
    char mtext[TEXT_SIZE] ;
}  ;
char  *getxtsj()
{
    time_t  tv ;
    struct  tm   *tmp ;
    static  char  buf[20] ;
    tv = time( 0 ) ;
    tmp = localtime(&tv) ;
    sprintf(buf,"%02d:%02d:%02d",tmp->tm_hour , tmp->tm_min,tmp->tm_sec);
    return   buf ;
}
int main(int argc, char **argv)
{
    int msqid ;
    struct msqid_ds info ;
    struct msgbuf buf ;
    struct msgbuf buf1 ;
    int flag ;
    int sendlength, recvlength ;
    int key ;

    key = ftok("msg.tmp", 0x01 ) ;
    if ( key < 0 )
    {
        perror("ftok key error") ;
        return -1 ;
    }

    msqid = msgget( key, 0600|IPC_CREAT ) ;
    if ( msqid < 0 )
    {
        perror("create message queue error") ;
        return -1 ;
    }

    buf.mtype = 1 ;
    buf.status = 9 ;
    strcpy(buf.time, getxtsj()) ;
    strcpy(buf.mtext, "happy new year!") ;
    sendlength = sizeof(struct msgbuf) - sizeof(long) ;
    flag = msgsnd( msqid, &buf, sendlength , 0 ) ;
    if ( flag < 0 )
    {
        perror("send message error") ;
        return -1 ;
    }
    buf.mtype = 3 ;
    buf.status = 9 ;
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
    strcpy(buf.time, getxtsj()) ;
    strcpy(buf.mtext, "good bye!") ;
    sendlength = sizeof(struct msgbuf) - sizeof(long) ;
    flag = msgsnd( msqid, &buf, sendlength , 0 ) ;
    if ( flag < 0 )
    {
        perror("send message error") ;
        return -1 ;
    }
    system("ipcs -q") ;
   return 0 ;
}
```

（2）接收消息队列程序

msgrcv.c源代码如下：

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define TEXT_SIZE  512
struct msgbuf
{
    long mtype ;
    int  status ;
    char time[20] ;
    char mtext[TEXT_SIZE] ;
}  ;
int main(int argc, char **argv)
{
    int msqid ;
    struct msqid_ds info ;
    struct msgbuf buf1 ;
    int flag ;
    int  recvlength ;
    int key ;
    int mtype ;

    key = ftok("msg.tmp", 0x01 ) ;
    if ( key < 0 )
    {
        perror("ftok key error") ;
        return -1 ;
    }

    msqid = msgget( key, 0 ) ;
    if ( msqid < 0 )
    {
        perror("get ipc_id error") ;
        return -1 ;
    }

    recvlength = sizeof(struct msgbuf) - sizeof(long) ;
    memset(&buf1, 0x00, sizeof(struct msgbuf)) ;
    mtype = 1 ;
    flag = msgrcv( msqid, &buf1, recvlength ,mtype,0 ) ;
    if ( flag < 0 )
    {
        perror("recv message error\n") ;
```

```
        return -1 ;
    }
    printf("type=%d,time=%s, message=%s\n", buf1.mtype, buf1.time,  buf1.mtext) ;
    system("ipcs -q") ;
   return 0 ;
}
```

（3）编译与执行程序

① 在当前目录下利用>msg.tmp建立空文件msg.tmp。

② 编译发送消息队列程序 gcc msgsnd.c -o msgsnd。

③ 执行./msgsnd，执行结果如下：

```
  ----- Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x0101436d 294912     zjkf       600        1072         2
```

④ 编译接收消息程序 gcc msgrcv.c -o msgrcv

⑤ 执行./msgrcv，执行结果如下：

```
 type=1,time=03:23:16, message=happy new year!

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x0101436d 294912     zjkf       600        536          1
```

⑥ 利用ipcrm -q 294912 删除该消息队列。因为消息队列是随内核持续存在的，在程序中若不利用msgctl函数或在命令行用ipcrm命令显式地删除，该消息队列就一直存在于系统中。另外信号量和共享内存也是随内核持续存在的。

带格式的：项目符号和编号

2.3 信号量

2.3.1 信号量简要说明

信号量（也称信号灯）与其他进程间通信的方式不大相同，它主要提供对进程间共享资源访问控制机制，相当于内存中的标志，进程可以根据它判定是否能够访问某些共享资源，从而实现多个进程对某些共享资源的互斥访问；同时，进程也可以修改该标志。信号量除了用于访问控制外，还可用于进程同步。由于一个信号量标识符指向的是一组信号量，所以在这里把信号量称为信号量集，一个信号量集使用同一个信号量标识符（或称信号量集标识符），管理的是一组信号量。这样实现避免了系统中有过多的信号量对象，而且易于编程。用户使用信号量时是以信号量集中每一个信号量为操作单位，所以操作信号量时要指明信号量标识符和该信号量在信号量集中的编号。后文为了避免理解歧义，将信号量标识符一律称为信号量集标识符。

1. 信号量集内核图

下图 15-5 画出信号量集的实现方法。从图中可以看出，sem_base指向的是一组信号量，所以一个信号量集管理的是一组信号量，该信号量集中信号量的个数用户可根据实际需要进行自行指定。

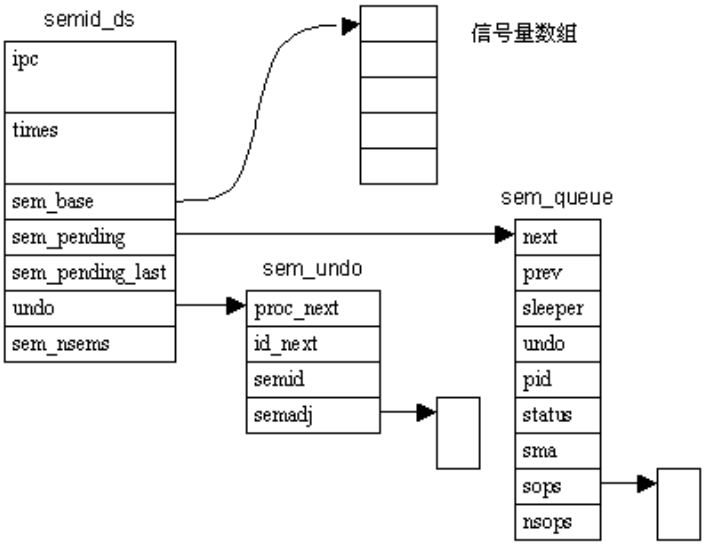

图 15-5 信号量集内核图

2. 信号量集内核结构定义

在Linux系统中，内核对信号量整体的管理是通过semary数组来实现，semary变量在系统中的定义如下：

```
struct semid_ds *semary[SEMMNI];
```

semary数组中的每个元素都是一个指向semid_ds数据结构的指针，而一个semid_ds数据结构则描述了一个信号量集。SEMMNI的值是 128，它限制了系统中同时存在的信号量集的数量，但一个信号量集可以管理一组信号量，所以这个数值远远够用。数据结构semid_ds的定义如下：

```
struct semid_ds {
   struct ipc_perm sem_perm;        /*包含信号量集资源的属主和访问权限*/
   struct sem      *sem_base;       /*指向信号量集合的指针*/
   unsigned short  int sem_nsems;   /*集合中信号量的个数*/
   time_t          sem_otime;       /*最后一次操作的时间*/
   time_t          sem_ctime;       /*最后一次修改的时间*/
  }
struct sem {
  unsigned short semval;            /*当前信号量值*/
  pid_t     sempid;                 /*最后修改信号量的进程*/
  unsignde short semcnt;            /*等待进行P操作的进程数*/
  unsigned short semzcnt;           /*等待semval为 0 的进程数*/
 }
```

带格式的：项目符号和编号

2.3.2 信号量函数

信号量函数由semget、semop、semctl三个函数组成。下面的表格列出了这三个函数的函数原型及具体说明。

1. semget函数原型

semget(得到一个信号量集标识符或创建一个信号量集对象)	
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数说明	得到一个信号量集标识符或创建一个信号量集对象并返回信号量集标识符

函数原型	int semget(key_t key, int nsems, int semflg)	
函数传入值	key	0(IPC_PRIVATE)：会建立新信号量集对象
		大于 0 的 32 位整数:视参数semflg来确定操作,通常要求此值来源于ftok返回的IPC键值
	nsems	创建信号量集中信号量的个数，该参数只在创建信号量集时有效
	msgflg	0：取信号量集标识符，若不存在则函数会报错
		IPC_CREAT：当semflg&IPC_CREAT为真时，如果内核中不存在键值与key相等的信号量集，则新建一个信号量集；如果存在这样的信号量集，返回此信号量集的标识符
		IPC_CREAT|IPC_EXCL：如果内核中不存在键值与key相等的信号量集，则新建一个消息队列；如果存在这样的信号量集则报错
函数返回值	成功：返回信号量集的标识符	
	出错：-1，错误原因存于 error 中	
附加说明	上述 semflg 参数为模式标志参数，使用时需要与 IPC 对象存取权限（如 0600）进行|运算来确定信号量集的存取权限	
错误代码	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semflg没有设置IPC_CREAT标志 ENOMEM：没有足够的内存创建新的信号量集 ENOSPC：超出限制	

如果用semget创建了一个新的信号量集对象时，则semid_ds结构成员变量的值设置如下：

- sem_otime设置为 0。
- sem_ctime设置为当前时间。
- msg_qbytes设成系统的限制值。
- sem_nsems设置为nsems参数的数值。
- semflg的读写权限写入sem_perm.mode中。
- sem_perm结构的uid和cuid成员被设置成当前进程的有效用户ID，gid和cuid成员被设置成当前进程的有效组ID。

2. semop函数原型

semop(完成对信号量的 P 操作或 V 操作)	
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数说明	对信号量集标识符为 semid 中的一个或多个信号量进行 P 操作或 V 操作
函数原型	int semop(int semid, struct sembuf *sops, unsigned nsops)
函数传入值	semid：信号量集标识符

	sops：指向进行操作的信号量集结构体数组的首地址，此结构的具体说明如下：
	struct sembuf {
	short semnum; /*信号量集合中的信号量编号，0 代表第 1 个信号量*/
	short val;/*若val>0 进行V操作信号量值加val，表示进程释放控制的资源 */
	/*若val<0 进行P操作信号量值减val，若(semval-val)<0（semval为该信号量值），则调用进程阻塞，直到资源可用；若设置IPC_NOWAIT不会睡眠，进程直接返回EAGAIN错误*/
	/*若val==0 时阻塞等待信号量为 0，调用进程进入睡眠状态，直到信号值为 0；若设置IPC_NOWAIT，进程不会睡眠，直接返回EAGAIN错误*/
	short flag; /*0 设置信号量的默认操作*/
	/*IPC_NOWAIT设置信号量操作不等待*/
	/*SEM_UNDO 选项会让内核记录一个与调用进程相关的UNDO记录，如果该进程崩溃，则根据这个进程的UNDO记录自动恢复相应信号量的计数值*/
	};

	nsops：进行操作信号量的个数，即sops结构变量的个数，需大于或等于 1。最常见设置此值等于 1，只完成对一个信号量的操作
函数返回值	成功：返回信号量集的标识符
	出错：-1，错误原因存于 error 中
错误代码	E2BIG：一次对信号量个数的操作超过了系统限制 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

sops 为指向 sembuf 数组，定义所要进行的操作序列。下面是信号量操作举例。

```
struct sembuf sem_get={0,-1,IPC_NOWAIT}; /*将信号量对象中序号为 0 的信号量减 1*/
struct sembuf sem_get={0,1,IPC_NOWAIT};  /*将信号量对象中序号为 0 的信号量加 1*/
struct sembuf sem_get={0,0,0};           /*进程被阻塞，直到对应的信号量值为 0*/
```

flag 一般为 0，若 flag 包含 IPC_NOWAIT，则该操作为非阻塞操作。若 flag 包含 SEM_UNDO，则当进程退出的时候会还原该进程的信号量操作，这个标志在某些情况下是很有用的，比如某进程做了 P 操作得到资源，但还没来得及做 V 操作时就异常退出了，此时，其他进程就只能都阻塞在 P 操作上，于是造成了死锁。若采取 SEM_UNDO 标志，就可以避免因为进程异常退出而造成的死锁。

3. semctl函数原型

semctl (得到一个信号量集标识符或创建一个信号量集对象)		
所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>	
函数说明	得到一个信号量集标识符或创建一个信号量集对象并返回信号量集标识符	
函数原型	int semctl(int semid, int semnum, int cmd, union semun arg)	
函数传入值	semid	信号量集标识符
	semnum	信号量集数组上的下标，表示某一个信号量
	cmd	见下文表 15-4

PDF-XChange Click to buy NOW! www.docu-track.com

	arg	union semun { short val; /*SETVAL用的值*/ struct semid_ds* buf; /*IPC_STAT、IPC_SET用的semid_ds结构*/ unsigned short* array; /*SETALL、GETALL用的数组值*/ struct seminfo *buf; /*为控制IPC_INFO提供的缓存*/ } arg;
函数返回值	成功：大于或等于 0，具体说明请参照表 15-4	
	出错：-1，错误原因存于 error 中	
附加说明	semid_ds 结构见上文信号量集内核结构定义	
错误代码	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：进程有效用户没有 cmd 的权限 ERANGE：信号量值超出范围	

表 15-4 semctl 函数 cmd 形参说明表

命令	解　　释
IPC_STAT	从信号量集上检索 semid_ds 结构，并存到 semun 联合体参数的成员 buf 的地址中
IPC_SET	设置一个信号量集合的 semid_ds 结构中 ipc_perm 域的值，并从 semun 的 buf 中取出值
IPC_RMID	从内核中删除信号量集合
GETALL	从信号量集合中获得所有信号量的值，并把其整数值存到 semun 联合体成员的一个指针数组中
GETNCNT	返回当前等待资源的进程个数
GETPID	返回最后一个执行系统调用 semop()进程的 PID
GETVAL	返回信号量集合内单个信号量的值
GETZCNT	返回当前等待 100%资源利用的进程个数
SETALL	与 GETALL 正好相反
SETVAL	用联合体中 val 成员的值设置信号量集合中单个信号量的值

2.3.3 信号量应用程序举例

sem.c源代码如下：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
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 */
        };
/***对信号量数组semnum编号的信号量做P操作***/
int P(int semid, int semnum)
{
        struct sembuf sops={semnum,-1, SEM_UNDO};
        return (semop(semid,&sops,1));
}
/***对信号量数组semnum编号的信号量做V操作***/
int V(int semid, int semnum)
{
```

PDF-XChange
Click to buy NOW!
www.docu-track.com

```
        struct sembuf sops={semnum,+1, SEM_UNDO};
        return (semop(semid,&sops,1));
}

int main(int argc, char **argv)
{
        int key ;
        int semid,ret;
        union semun arg;
        struct sembuf semop;
        int flag ;

        key = ftok("/tmp", 0x66 ) ;
        if ( key < 0 )
        {
            perror("ftok key error") ;
            return -1 ;
        }
        /***本程序创建了三个信号量，实际使用时只用了一个 0 号信号量***/
        semid = semget(key,3,IPC_CREAT|0600);
        if (semid == -1)
        {
                perror("create semget error");
                return ;
        }
        if ( argc == 1 )
        {
            arg.val = 1;
            /***对 0 号信号量设置初始值***/
            ret =semctl(semid,0,SETVAL,arg);
            if (ret < 0 )
            {
                    perror("ctl sem error");
                    semctl(semid,0,IPC_RMID,arg);
                    return -1 ;
            }
        }
        /***取 0 号信号量的值***/
        ret =semctl(semid,0,GETVAL,arg);
        printf("after semctl setval  sem[0].val =[%d]\n",ret);
        system("date") ;
        printf("P operate begin\n") ;
        flag = P(semid,0)  ;
        if ( flag )
        {
            perror("P operate error") ;
            return -1 ;
        }
        printf("P operate end\n") ;
        ret =semctl(semid,0,GETVAL,arg);
        printf("after P sem[0].val=[%d]\n",ret);
        system("date") ;
        if ( argc == 1 )
        {
            sleep(120) ;
        }
        printf("V operate begin\n") ;
    if (V(semid, 0) < 0)
        {
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
            perror("V operate error") ;
            return -1 ;
        }
        printf("V operate end\n") ;
        ret =semctl(semid,0,GETVAL,arg);
        printf("after V sem[0].val=%d\n",ret);
        system("date") ;
        if ( argc >1 )
        {
            semctl(semid,0,IPC_RMID,arg);
        }

        return 0 ;
}
```

① 编译 gcc sem.c －o sem。

② 在一窗口执行./sem，执行结果如下：

```
after semctl setval  sem[0].val =[1]
2011 年 01 月 11 日 星期二 10:08:11 CST
P operate begin
P operate end
after P sem[0].val=[0]
2011 年 01 月 11 日 星期二 10:08:11 CST
V operate begin
V operate end
after V sem[0].val=0
2011 年 01 月 11 日 星期二 10:10:11 CST
```

③ 然后在另一窗口中执行./sem test1，执行结果如下：

```
after semctl setval  sem[0].val =[0]
2011 年 01 月 11 日 星期二 10:08:36 CST
P operate begin
P operate end
after P sem[0].val=[0]
2011 年 01 月 11 日 星期二 10:10:11 CST
V operate begin
V operate end
after V sem[0].val=1
2011 年 01 月 11 日 星期二 10:10:11 CST
```

带格式的：项目符号和编号

2.4 共享内存

2.4.1 共享内存简要说明

共享内存区域是被多个进程共享的一部分物理内存。如果多个进程都把该内存区域映射到自己的虚拟地址空间，则这些进程就都可以直接访问该共享内存区域，从而可以通过该区域进行通信。共享内存是进程间共享数据的一种最快的方法，一个进程向共享内存区域写入了数据，共享这个内存区域的所有进程就可以立刻看到其中的内容。这块共享虚拟内存的页面，出现在每一个共享该页面的进程的页表中。下图 15-6 为共享内存实现原理图。

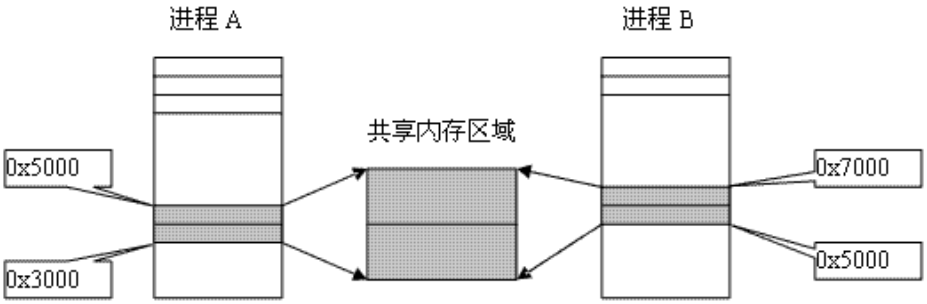

图 15-6 共享内存实现原理图

1. 共享内存内核结构定义

每一个新创建的共享内存对象都可用一个shmid_kernel数据结构来表达。系统中所有的shmid_kernel数据结构都保存在shm_segs数组中，shm_segs数组的每一个元素都是一个指向shmid_kernel数据结构的指针，shm_segs数组管理着系统中所有的共享内存。shmid_kernel结构和shm_segs变量的定义如下：

```
struct shmid_kernel *shm_segs[SHMMNI];
/* SHMMNI为 128，表示系统中最多可以有 128 个共享内存对象*/
/* 数据结构shmid_kernel的定义如下：*/
struct shmid_kernel
{
    struct shmid_ds u;          /* the following are private */
    unsigned long shm_npages;   /* size of segment (pages) */
    unsigned long *shm_pages;   /* array of ptrs to frames->SHMMAX */
    struct vm_area_struct *attaches;  /* descriptors for attaches */
};
```

其中：

- shm_pages：是指向该共享内存对象所占据的内存页面数组，数组里面的每个元素是每个内存页面的起始地址。
- shm_npages：是该共享内存区域的大小，以页为单位。
- shmid_ds：是一个数据结构，它描述了这个共享内存区的访问权限信息，字节大小，最后一次粘附时间、分离时间、改变时间，创建该共享区域的进程ID，最后一次对它操作的进程ID，当前有多少个进程在使用它的信息等。其定义如下：

```
struct shmid_ds {
    struct ipc_perm shm_perm;    /* operation perms */
    int shm_segsz;               /* size of segment (bytes) */
    __kernel_time_t shm_atime;   /* last attach time */
    __kernel_time_t shm_dtime;   /* last detach time */
    __kernel_time_t shm_ctime;   /* last change time */
    __kernel_ipc_pid_t shm_cpid; /* pid of creator */
    __kernel_ipc_pid_t shm_lpid; /* pid of last operator */
    unsigned short shm_nattch;   /* number of current attaches */
    unsigned short shm_unused;   /* compatibility */
    void *shm_unused2;           /* ditto - used by DIPC */
    void *shm_unused3;           /* unused */
};
```

带格式的：项目符号和编号

2.4.2 共享内存函数

共享内存函数由shmget、shmat、shmdt、shmctl四个函数组成。下面的表格列出了这四个函数的函数原型及其具体说明。

1. shmget函数原型

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	
附加说明	上述 shmflg 参数为模式标志参数，使用时需要与 IPC 对象存取权限（如 0600）进行|运算来确定信号量集的存取权限	
错误代码	EINVAL：参数 size 小于 SHMMIN 或大于 SHMMAX EEXIST：预建立 key 所指的共享内存，但已经存在 EIDRM：参数 key 所指的共享内存已经删除 ENOSPC：超过了系统允许建立的共享内存的最大值(SHMALL) ENOENT：参数 key 所指的共享内存不存在，而参数 shmflg 未设 IPC_CREAT 位 EACCES：没有权限 ENOMEM：核心内存不足	

在Linux环境中，对开始申请的共享内存空间进行了初始化，初始值为 0x00。

如果用shmget创建了一个新的消息队列对象时，则shmid_ds结构成员变量的值设置如下：

- shm_lpid、shm_nattach、shm_atime、shm_dtime设置为 0。
- msg_ctime设置为当前时间。
- shm_segsz设成创建共享内存的大小。
- shmflg的读写权限放在shm_perm.mode中。
- shm_perm结构的uid和cuid成员被设置成当前进程的有效用户ID，gid和cuid成员被设置成当前进程的有效组ID。

2. shmat函数原型

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 shmid 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 shmid, const void *shmaddr, int shmflg)	
函数传入值	msqid	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
附加说明	fork 后子进程继承已连接的共享内存地址。exec 后该子进程与已连接的共享内存地址自动脱离(detach)。进程结束后，已连接的共享内存地址会自动脱离(detach)
错误代码	EACCES：无权限以指定方式连接共享内存 EINVAL：无效的参数 shmid 或 shmaddr ENOMEM：核心内存不足

3. shmdt函数原型

shmat(断开共享内存连接)	
所需头文件	#include <sys/types.h> #include <sys/shm.h>
函数说明	与shmat函数相反，是用来断开与共享内存附加点的地址，禁止本进程访问此片共享内存
函数原型	int shmdt(const void *shmaddr)
函数传入值	shmaddr：连接的共享内存的起始地址
函数返回值	成功：0
	出错：-1，错误原因存于error中
附加说明	本函数调用并不删除所指定的共享内存区，而只是将先前用shmat函数连接（attach）好的共享内存脱离（detach）目前的进程
错误代码	EINVAL：无效的参数shmaddr

4. shmctl函数原型

shmctl(共享内存管理)		
所需头文件	#include <sys/types.h> #include <sys/shm.h>	
函数说明	完成对共享内存的控制	
函数原型	int shmctl(int shmid, int cmd, struct shmid_ds *buf)	
函数传入值	msqid	共享内存标识符
	cmd	IPC_STAT：得到共享内存的状态，把共享内存的shmid_ds结构复制到buf中
		IPC_SET：改变共享内存的状态，把buf所指的shmid_ds结构中的uid、gid、mode复制到共享内存的shmid_ds结构内
		IPC_RMID：删除这片共享内存
	buf	共享内存管理结构体。具体说明参见共享内存内核结构定义部分
函数返回值	成功：0	
	出错：-1，错误原因存于 error 中	
错误代码	EACCESS：参数cmd为IPC_STAT，确无权限读取该共享内存 EFAULT：参数buf指向无效的内存地址 EIDRM：标识符为msqid的共享内存已被删除 EINVAL：无效的参数 cmd 或 shmid EPERM：参数 cmd 为 IPC_SET 或 IPC_RMID，却无足够的权限执行	

带格式的：项目符号和编号

2.4.3 共享内存应用范例

1. 父子进程通信范例

父子进程通信范例，shm.c源代码如下：

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <error.h>
```

PDF-XChange Click to buy NOW! www.docu-track.com

```
#define SIZE 1024
int main()
{
    int shmid ;
    char *shmaddr ;
    struct shmid_ds buf ;
    int flag = 0 ;
    int pid ;

    shmid = shmget(IPC_PRIVATE, SIZE, IPC_CREAT|0600 ) ;
    if ( shmid < 0 )
    {
            perror("get shm  ipc_id error") ;
            return -1 ;
    }
    pid = fork() ;
    if ( pid == 0 )
    {
        shmaddr = (char *)shmat( shmid, NULL, 0 ) ;
        if ( (int)shmaddr == -1 )
        {
            perror("shmat addr error") ;
            return -1 ;

        }
        strcpy( shmaddr, "Hi, I am child process!\n") ;
        shmdt( shmaddr ) ;
        return  0;
    } else if ( pid > 0) {
        sleep(3 ) ;
        flag = shmctl( shmid, IPC_STAT, &buf) ;
        if ( flag == -1 )
        {
            perror("shmctl shm error") ;
            return -1 ;
        }

        printf("shm_segsz =%d bytes\n", buf.shm_segsz ) ;
        printf("parent pid=%d, shm_cpid = %d \n", getpid(), buf.shm_cpid ) ;
        printf("chlid pid=%d, shm_lpid = %d \n",pid , buf.shm_lpid ) ;
        shmaddr = (char *) shmat(shmid, NULL, 0 ) ;
        if ( (int)shmaddr == -1 )
        {
            perror("shmat addr error") ;
            return -1 ;

        }
        printf("%s", shmaddr) ;
        shmdt( shmaddr ) ;
        shmctl(shmid, IPC_RMID, NULL) ;
    }else{
        perror("fork error") ;
        shmctl(shmid, IPC_RMID, NULL) ;
    }

    return 0 ;
}
```

编译 gcc shm.c －o shm。

执行 ./shm，执行结果如下：

```
shm_segsz =1024 bytes
shm_cpid = 9503
shm_lpid = 9504
Hi, I am child process!
```

2. 多进程读写范例

多进程读写即一个进程写共享内存，一个或多个进程读共享内存。下面的例子实现的是一个进程写共享内存，一个进程读共享内存。

（1）下面程序实现了创建共享内存，并写入消息。

shmwrite.c源代码如下：

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
typedef struct{
    char name[8];
    int age;
} people;
int main(int argc, char** argv)
{
    int shm_id,i;
    key_t key;
    char temp[8];
    people *p_map;
    char pathname[30] ;

    strcpy(pathname,"/tmp") ;
    key = ftok(pathname,0x03);
    if(key==-1)
    {
        perror("ftok error");
        return -1;
    }
    printf("key=%d\n",key) ;
    shm_id=shmget(key,4096,IPC_CREAT|IPC_EXCL|0600);
    if(shm_id==-1)
    {
        perror("shmget error");
        return -1;
    }
    printf("shm_id=%d\n", shm_id) ;
    p_map=(people*)shmat(shm_id,NULL,0);
    memset(temp, 0x00, sizeof(temp)) ;
    strcpy(temp,"test") ;
    temp[4]='0';
    for(i = 0;i<3;i++)
    {
        temp[4]+=1;
        strncpy((p_map+i)->name,temp,5);
        (p_map+i)->age=0+i;
    }
    shmdt(p_map) ;
    return 0 ;
}
```

（2）下面程序实现从共享内存读消息。

PDF-XChange Click to buy NOW! www.docu-track.com

shmread.c源代码如下：

```
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
typedef struct{
    char name[8];
    int age;
} people;
int main(int argc, char** argv)
{
    int shm_id,i;
    key_t key;
    people *p_map;
    char pathname[30] ;

    strcpy(pathname,"/tmp") ;
    key = ftok(pathname,0x03);
    if(key == -1)
    {
        perror("ftok error");
        return -1;
    }
    printf("key=%d\n", key) ;
    shm_id = shmget(key,0, 0);
    if(shm_id == -1)
    {
        perror("shmget error");
        return -1;
    }
    printf("shm_id=%d\n", shm_id) ;
    p_map = (people*)shmat(shm_id,NULL,0);
    for(i = 0;i<3;i++)
    {
        printf( "name:%s\n",(*(p_map+i)).name );
        printf( "age %d\n",(*(p_map+i)).age );
    }
    if(shmdt(p_map) == -1)
    {
        perror("detach error");
        return -1;
    }
    return 0 ;
}
```

（3）编译与执行

① 编译gcc shmwrite.c -o shmwrite。

② 执行./shmwrite，执行结果如下：

```
key=50453281
shm_id=688137
```

③ 编译gcc shmread.c -o shmread。

④ 执行./shmread，执行结果如下：

```
key=50453281
shm_id=688137
name:test1
age 0
```

PDF-XChange
Click to buy NOW!
www.docu-track.com

```
    name:test2
    age 1
    name:test3
    age 2
```

⑤ 再执行./shmwrite，执行结果如下：

```
    key=50453281
    shmget error: File exists
```

⑥ 使用ipcrm -m 688137 删除此共享内存。

此书由电子工业出版社即将出版，部分文档开源处理。
本书作者：余国平
相关下载地址：
C语言部分下载地址
http://www.docin.com/p-188767635.html
http://download.csdn.net/source/3213421
Shell部分下载地址
http://download.csdn.net/source/3257512
http://www.docin.com/p-197418889.html
Socket网络部分下载地址
http://download.csdn.net/source/3260134