嵌入式与移动开发系列

NITE 国家信息技术紧缺人才培养工程
National Information Technology Education Project
国家信息技术紧缺人才培养工程系列丛书

众多专家、厂商联合推荐 • 业界权威培训机构的经验总结

嵌入式Linux应用程序开发标准教程（第2版）

华清远见嵌入式培训中心 编著

提供36小时嵌入式专家讲座视频和教学课件

Embeded Linux Application Development

人民邮电出版社
POSTS & TELECOM PRESS

第 8 章　进程间通信

本章目标

在上一章中，读者已经学会了如何创建进程以及如何对进程进行基本的控制，而这些都只是停留在父子进程之间的控制，本章将要学习不同的进程间进行通信的方法，通过本章的学习，读者将会掌握如下内容。

- 掌握 Linux 中管道的基本概念
- 掌握 Linux 中管道的创建
- 掌握 Linux 中管道的读写
- 掌握 Linux 中有名管道的创建读写方法
- 掌握 Linux 中消息队列的处理
- 掌握 Linux 共享内存的处理

8.1 Linux 下进程间通信概述

在上一章中，读者已经知道了进程是一个程序的一次执行。这里所说的进程一般是指运行在用户态的进程，而由于处于用户态的不同进程之间是彼此隔离的，就像处于不同城市的人们，它们必须通过某种方式来进行通信，例如人们现在广泛使用的手机等方式。本章就是讲述如何建立这些不同的通话方式，就像人们有多种通信方式一样。

Linux 下的进程通信手段基本上是从 UNIX 平台上的进程通信手段继承而来的。而对 UNIX 发展做出重大贡献的两大主力 AT&T 的贝尔实验室及 BSD（加州大学伯克利分校的伯克利软件发布中心）在进程间的通信方面的侧重点有所不同。前者是对 UNIX 早期的进程间通信手段进行了系统的改进和扩充，形成了“system V IPC”，其通信进程主要局限在单个计算机内；后者则跳过了该限制，形成了基于套接口（socket）的进程间通信机制。而 Linux 则把两者的优势都继承了下来，如图 8.1 所示。

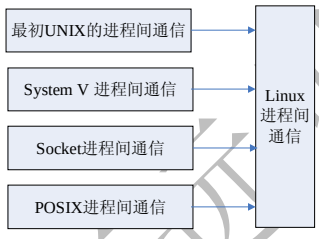

图 8.1　进程间通信发展历程

- UNIX 进程间通信（IPC）方式包括管道、FIFO 以及信号。
- System V 进程间通信（IPC）包括 System V 消息队列、System V 信号量以及 System V 共享内存区。
- Posix 进程间通信（IPC）包括 Posix 消息队列、Posix 信号量以及 Posix 共享内存区。

现在在 Linux 中使用较多的进程间通信方式主要有以下几种。

（1）管道（Pipe）及有名管道（named pipe）：管道可用于具有亲缘关系进程间的通信，有名管道，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。

（2）信号（Signal）：信号是在软件层次上对中断机制的一种模拟，它是比较复杂的通信方式，用于通知进程有某事件发生，一个进程收到一个信号与处理器收到一个中断请求效果上可以说是一样的。

（3）消息队列（Messge Queue）：消息队列是消息的链接表，包括 Posix 消息队列 SystemV 消息队列。它克服了前两种通信方式中信息量有限的缺点，具有写权限的进程可以按照一定的规则向消息队列中添加新消息；对消息队列有读权限的进程则可以从消息队列中读取消息。

（4）共享内存（Shared memory）：可以说这是最有用的进程间通信方式。它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。这种通信方式需要依靠某种同步机制，如互斥锁和信号量等。

（5）信号量（Semaphore）：主要作为进程之间以及同一进程的不同线程之间的同步和互斥手段。

（6）套接字（Socket）：这是一种更为一般的进程间通信机制，它可用于网络中不同机器之间的进程间通信，应用非常广泛。

本章会详细介绍前 5 种进程通信方式，对第 6 种通信方式将会在第 10 章中单独介绍。

8.2 管道

8.2.1 管道概述

本书在第 2 章中介绍“ps”的命令时提到过管道，当时指出了管道是 Linux 中一种很重要的通信方式，它是把一个程序的输出直接连接到另一个程序的输入，这里仍以第 2 章中的“ps –ef | grep ntp”为例，描述管道的通信过程，如图 8.2 所示。

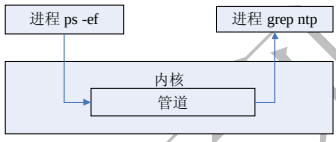

图 8.2 管道的通信过程

管道是 Linux 中进程间通信的一种方式。这里所说的管道主要指无名管道，它具有如下特点。

- 它只能用于具有亲缘关系的进程之间的通信（也就是父子进程或者兄弟进程之间）。
- 它是一个半双工的通信模式，具有固定的读端和写端。
- 管道也可以看成是一种特殊的文件，对于它的读写也可以使用普通的 read()和 write()等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内核的内存空间中。

8.2.2 管道系统调用

1. 管道创建与关闭说明

管道是基于文件描述符的通信方式，当一个管道建立时，它会创建两个文件描述符 fds[0]和 fds[1]，其中 fds[0]固定用于读管道，而 fd[1]固定用于写管道，如图 8.3 所示，这样就构成了一个半双工的通道。

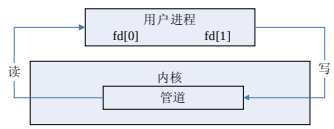

图 8.3 Linux 中管道与文件描述符的关系

管道关闭时只需将这两个文件描述符关闭即可，可使用普通的 close()函数逐个关闭各个文件描述符。

注意　当一个管道共享多对文件描述符时，若将其中的一对读写文件描述符都删除，则该管道就失效。

2．管道创建函数

创建管道可以通过调用 pipe()来实现，表 8.1 列出了 pipe()函数的语法要点。

表 8.1　　pipe()函数语法要点

所需头文件	#include <unistd.h>
函数原型	int pipe(int fd[2])
函数传入值	fd[2]：管道的两个文件描述符，之后就可以直接操作这两个文件描述符
函数返回值	成功：0
	出错：−1

3．管道读写说明

用 pipe()函数创建的管道两端处于一个进程中，由于管道是主要用于在不同进程间通信的，因此这在实际应用中没有太大意义。实际上，通常先是创建一个管道，再通过 fork()函数创建一子进程，该子进程会继承父进程所创建的管道，这时，父子进程管道的文件描述符对应关系如图 8.4 所示。

此时的关系看似非常复杂，实际上却已经给不同进程之间的读写创造了很好的条件。父子进程分别拥有自己的读写通道，为了实现父子进程之间的读写，只需把无关的读端或写端的文件描述符关闭即可。例如在图 8.5 中将父进程的写端 fd[1]和子进程的读端 fd[0]关闭。此时，父子进程之间就建立起了一条“子进程写入父进程读取”的通道。

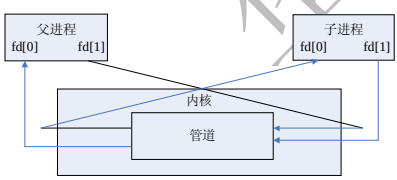

图 8.4　父子进程管道的文件描述符对应关系

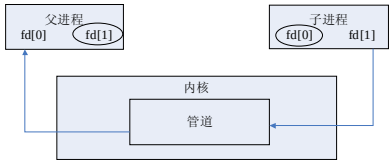

图 8.5　关闭父进程 fd[1]和子进程 fd[0]

同样，也可以关闭父进程的 fd[0]和子进程的 fd[1]，这样就可以建立一条“父进程写入子进程读取”的通道。另外，父进程还可以创建多个子进程，各个子进程都继承了相应的 fd[0]和 fd[1]，这时，只需要关闭相应端口就可以建立其各子进程之间的通道。

想一想　为什么无名管道只能在具有亲缘关系的进程之间建立？

4．管道使用实例

在本例中，首先创建管道，之后父进程使用 fork()函数创建子进程，之后通过关闭父进程的读描述符和子进程的写描述符，建立起它们之间的管道通信。

```
    /* pipe.c */
    #include <unistd.h>
    #include <sys/types.h>
    #include <errno.h>
    #include <stdio.h>
    #include <stdlib.h>

    #define MAX_DATA_LEN   256
    #define DELAY_TIME 1

    int main()
    {
        pid_t pid;
        int pipe_fd[2];
        char buf[MAX_DATA_LEN];
        const char data[] = "Pipe Test Program";
        int real_read, real_write;

        memset((void*)buf, 0, sizeof(buf));
        /* 创建管道 */
        if (pipe(pipe_fd) < 0)
        {
            printf("pipe create error\n");
            exit(1);
        }

        /* 创建一子进程 */
        if ((pid = fork()) == 0)
        {
            /* 子进程关闭写描述符,并通过使子进程暂停1s等待父进程已关闭相应的读描述符 */
            close(pipe_fd[1]);
            sleep(DELAY_TIME * 3);

            /* 子进程读取管道内容 */
            if ((real_read = read(pipe_fd[0], buf, MAX_DATA_LEN)) > 0)
            {
                printf("%d bytes read from the pipe is '%s'\n", real_read,
buf);
            }
```

```
        /* 关闭子进程读描述符 */
        close(pipe_fd[0]);
        exit(0);
    }
    else if (pid > 0)
    {
        /* 父进程关闭读描述符,并通过使父进程暂停1s等待子进程已关闭相应的写描述符 */
        close(pipe_fd[0]);
        sleep(DELAY_TIME);

        if((real_write = write(pipe_fd[1], data, strlen(data))) != -1)
        {
            printf("Parent wrote %d bytes : '%s'\n", real_write, data);
        }

        /*关闭父进程写描述符*/
        close(pipe_fd[1]);

        /*收集子进程退出信息*/
        waitpid(pid, NULL, 0);
        exit(0);
    }
}
```

将该程序交叉编译，下载到开发板上的运行结果如下所示：

```
$ ./pipe
Parent wrote 17 bytes : 'Pipe Test Program'
17 bytes read from the pipe is 'Pipe Test Program'
```

5. 管道读写注意点

- 只有在管道的读端存在时，向管道写入数据才有意义。否则，向管道写入数据的进程将收到内核传来的 SIGPIPE 信号（通常为 Broken pipe 错误）。
- 向管道写入数据时，Linux 将不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读取管道缓冲区中的数据，那么写操作将会一直阻塞。
- 父子进程在运行时，它们的先后次序并不能保证，因此，在这里为了保证父子进程已经关闭了相应的文件描述符，可在两个进程中调用 sleep()函数，当然这种调用不是很好的解决方法，在后面学到进程之间的同步与互斥机制之后，请读者自行修改本小节的实例程序。

8.2.4 标准流管道

1．标准流管道函数说明

与 Linux 的文件操作中有基于文件流的标准 I/O 操作一样，管道的操作也支持基于文件流的模式。这种基于文件流的管道主要是用来创建一个连接到另一个进程的管道，这里的“另一个进程”也就是一个可以进行一定操作的可执行文件，例如，用户执行“ls -l”或者自己编写的程序“./pipe”等。由于这一类操作很常用，因此标准流管道就将一系列的创建过程合并到一个函数 popen()中完成。它所完成的工作有以下几步。

- 创建一个管道。
- fork()一个子进程。
- 在父子进程中关闭不需要的文件描述符。
- 执行 exec 函数族调用。
- 执行函数中所指定的命令。

这个函数的使用可以大大减少代码的编写量，但同时也有一些不利之处，例如，它不如前面管道创建的函数那样灵活多样，并且用 popen()创建的管道必须使用标准 I/O 函数进行操作，但不能使用前面的 read()、write()一类不带缓冲的 I/O 函数。

与之相对应，关闭用 popen()创建的流管道必须使用函数 pclose()来关闭该管道流。该函数关闭标准 I/O 流，并等待命令执行结束。

2．函数格式

popen()和 pclose()函数格式如表 8.2 和表 8.3 所示。

表 8.2　　popen()函数语法要点

所需头文件	#include <stdio.h>	
函数原型	FILE *popen(const char *command, const char *type)	
函数传入值	command：指向的是一个以 null 结束符结尾的字符串，这个字符串包含一个 shell 命令，并被送到/bin/sh 以-c 参数执行，即由 shell 来执行	
	type：	“r”：文件指针连接到 command 的标准输出，即该命令的结果产生输出 “w”：文件指针连接到 command 的标准输入，即该命令的结果产生输入
函数返回值	成功：文件流指针	
	出错：–1	

表 8.3　　pclose()函数语法要点

所需头文件	#include <stdio.h>
函数原型	int pclose(FILE *stream)
函数传入值	stream：要关闭的文件流
函数返回值	成功：返回由 popen()所执行的进程的退出码
	出错：–1

3．函数使用实例

在该实例中，使用 popen()来执行“ps -ef”命令。可以看出，popen()函数的使用能够使程序变得短小精悍。

```
/* standard_pipe.c */
#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];

    /*调用 popen()函数执行相应的命令*/
    if ((fp = popen(cmd, "r")) == NULL)
        {
        printf("Popen error\n");
        exit(1);
    }

    while ((fgets(buf, BUFSIZE, fp)) != NULL)
    {
        printf("%s",buf);
    }
    pclose(fp);
    exit(0);
}
```

下面是该程序在目标板上的执行结果。

```
$ ./standard_pipe
  PID TTY          Uid       Size State Command
    1           root      1832   S   init
    2           root         0   S   [keventd]
    3           root         0   S   [ksoftirqd_CPU0]
    ……
   74           root      1284   S   ./standard_pipe
```

```
75          root      1836   S   sh -c ps -ef
76          root      2020   R   ps –ef
```

8.2.5 FIFO

1．有名管道说明

前面介绍的管道是无名管道，它只能用于具有亲缘关系的进程之间，这就大大地限制了管道的使用。有名管道的出现突破了这种限制，它可以使互不相关的两个进程实现彼此通信。该管道可以通过路径名来指出，并且在文件系统中是可见的。在建立了管道之后，两个进程就可以把它当作普通文件一样进行读写操作，使用非常方便。不过值得注意的是，FIFO 是严格地遵循先进先出规则的，对管道及 FIFO 的读总是从开始处返回数据，对它们的写则把数据添加到末尾，它们不支持如 lseek()等文件定位操作。

有名管道的创建可以使用函数 mkfifo()，该函数类似文件中的 open()操作，可以指定管道的路径和打开的模式。

小知识　用户还可以在命令行使用“mknod 管道名 p”来创建有名管道。

在创建管道成功之后，就可以使用 open()、read()和 write()这些函数了。与普通文件的开发设置一样，对于为读而打开的管道可在 open()中设置 O_RDONLY，对于为写而打开的管道可在 open()中设置 O_WRONLY，在这里与普通文件不同的是阻塞问题。由于普通文件的读写时不会出现阻塞问题，而在管道的读写中却有阻塞的可能，这里的非阻塞标志可以在 open()函数中设定为 O_NONBLOCK。下面分别对阻塞打开和非阻塞打开的读写进行讨论。

（1）对于读进程。

- 若该管道是阻塞打开，且当前 FIFO 内没有数据，则对读进程而言将一直阻塞到有数据写入。
- 若该管道是非阻塞打开，则不论 FIFO 内是否有数据，读进程都会立即执行读操作。即如果 FIFO 内没有数据，则读函数将立刻返回 0。

（2）对于写进程。

- 若该管道是阻塞打开，则写操作将一直阻塞到数据可以被写入。
- 若该管道是非阻塞打开而不能写入全部数据，则读操作进行部分写入或者调用失败。

2．mkfifo()函数格式

表 8.4 列出了 mkfifo()函数的语法要点。

表 8.4　　mkfifo()函数语法要点

所需头文件	#include <sys/types.h> #include <sys/state.h>
函数原型	int mkfifo(const char *filename,mode_t mode)
函数传入值	filename：要创建的管道

函数传入值	mode：	O_RDONLY：读管道
		O_WRONLY：写管道
		O_RDWR：读写管道
		O_NONBLOCK：非阻塞
函数传入值	mode：	O_CREAT：如果该文件不存在，那么就创建一个新的文件，并用第三个参数为其设置权限
		O_EXCL：如果使用 O_CREAT 时文件存在，那么可返回错误消息。这一参数可测试文件是否存在
函数返回值	成功：0	
	出错：–1	

表 8.5 再对 FIFO 相关的出错信息做一归纳，以方便用户查错。

表 8.5　　FIFO 相关的出错信息

EACCESS	参数 filename 所指定的目录路径无可执行的权限
EEXIST	参数 filename 所指定的文件已存在
ENAMETOOLONG	参数 filename 的路径名称太长
ENOENT	参数 filename 包含的目录不存在
ENOSPC	文件系统的剩余空间不足
ENOTDIR	参数 filename 路径中的目录存在但却非真正的目录
EROFS	参数 filename 指定的文件存在于只读文件系统内

3．使用实例

下面的实例包含了两个程序，一个用于读管道，另一个用于写管道。其中在读管道的程序里创建管道，并且作为 main()函数里的参数由用户输入要写入的内容。读管道的程序会读出用户写入到管道的内容，这两个程序采用的是阻塞式读写管道模式。

以下是写管道的程序：

```
/* fifo_write.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MYFIFO            "/tmp/myfifo"      /* 有名管道文件名*/
#define MAX_BUFFER_SIZE       PIPE_BUF       /*定义在于 limits.h 中*/

int main(int argc, char * argv[]) /*参数为即将写入的字符串*/
{
   int fd;
```

```
    char buff[MAX_BUFFER_SIZE];
    int nwrite;

    if(argc <= 1)
    {
        printf("Usage: ./fifo_write string\n");
        exit(1);
    }
    sscanf(argv[1], "%s", buff);

    /* 以只写阻塞方式打开 FIFO 管道 */
    fd = open(MYFIFO, O_WRONLY);
    if (fd == -1)
    {
        printf("Open fifo file error\n");
        exit(1);
    }

    /*向管道中写入字符串*/
    if ((nwrite = write(fd, buff, MAX_BUFFER_SIZE)) > 0)
    {
        printf("Write '%s' to FIFO\n", buff);
    }
    close(fd);
    exit(0);
}
```

以下是读管道程序：

```
/*fifo_read.c*/
(头文件和宏定义同 fifo_write.c)
int main()
{
    char buff[MAX_BUFFER_SIZE];
    int  fd;
    int  nread;

    /* 判断有名管道是否已存在，若尚未创建，则以相应的权限创建*/
    if (access(MYFIFO, F_OK) == -1)
    {
        if ((mkfifo(MYFIFO, 0666) < 0) && (errno != EEXIST))
        {
            printf("Cannot create fifo file\n");
            exit(1);
        }
    }
    /* 以只读阻塞方式打开有名管道 */
    fd = open(MYFIFO, O_RDONLY);
```

```
    if (fd == -1)
    {
        printf("Open fifo file error\n");
        exit(1);
    }

    while (1)
    {
        memset(buff, 0, sizeof(buff));
        if ((nread = read(fd, buff, MAX_BUFFER_SIZE)) > 0)
        {
            printf("Read '%s' from FIFO\n", buff);
        }
    }
    close(fd);
    exit(0);
}
```

为了能够较好地观察运行结果，需要把这两个程序分别在两个终端里运行，在这里首先启动读管道程序。读管道进程在建立管道之后就开始循环地从管道里读出内容，如果没有数据可读，则一直阻塞到写管道进程向管道写入数据。在启动了写管道程序后，读进程能够从管道里读出用户的输入内容，程序运行结果如下所示。

终端一：

```
$ ./fifo_read
Read 'FIFO' from FIFO
Read 'Test' from FIFO
Read 'Program' from FIFO
……
```

终端二：

```
$ ./fifo_write FIFO
Write 'FIFO' to FIFO
$ ./fifo_write Test
Write 'Test' to FIFO
$ ./fifo_write Program
Write 'Program' to FIFO
……
```

8.3 信号

8.3.1 信号概述

信号是 UNIX 中所使用的进程通信的一种最古老的方法。它是在软件层次上对中断机制的一种模拟，是一种异步通信方式。信号可以直接进行用户空间进程和内核进程之间的交互，内核进程也可以利用它来通知用户空间进程发生了哪些系统事件。它可以在任何时候发给某一进程，而无需知道该进程的状态。如果该进程当前并未处于执行态，则该信号就由内核保存起来，直到该进程恢复执行再传递给它为止；如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消时才被传递给进程。

在第 2 章 kill 命令中曾讲解到“-l”选项，这个选项可以列出该系统所支持的所有信号的列表。在笔者的系统中，信号值在 32 之前的则有不同的名称，而信号值在 32 以后的都是用“SIGRTMIN”或“SIGRTMAX”开头的，这就是两类典型的信号。前者是从 UNIX 系统中继承下来的信号，为不可靠信号（也称为非实时信号）；后者是为了解决前面“不可靠信号”的问题而进行了更改和扩充的信号，称为“可靠信号”（也称为实时信号）。那么为什么之前的信号不可靠呢？这里首先要介绍一下信号的生命周期。

一个完整的信号生命周期可以分为 3 个重要阶段，这 3 个阶段由 4 个重要事件来刻画的：信号产生、信号在进程中注册、信号在进程中注销、执行信号处理函数，如图 8.6 所示。相邻两个事件的时间间隔构成信号生命周期的一个阶段。要注意这里的信号处理有多种方式，一般是由内核完成的，当然也可以由用户进程来完成，故在此没有明确画出。

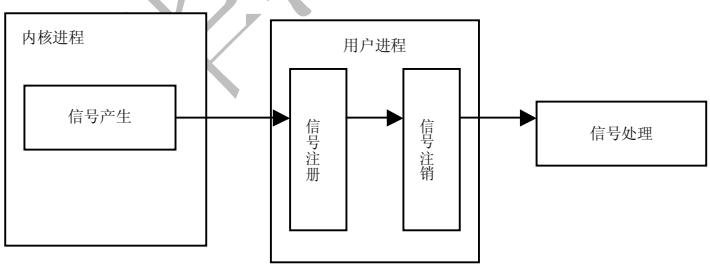

图 8.6 信号生命周期

一个不可靠信号的处理过程是这样的：如果发现该信号已经在进程中注册，那么就忽略该信号。因此，若前一个信号还未注销又产生了相同的信号就会产生信号丢失。而当可靠信号发送给一个进程时，不管该信号是否已经在进程中注册，都会被再注册一次，因此信号就不会丢失。所有可靠信号都支持排队，而所有不可靠信号都不支持排队。

注意 这里信号的产生、注册和注销等是指信号的内部实现机制，而不是调用信号的函数实现。因此，信号注册与否，与本节后面讲到的发送信号函数（如 kill()等）以及信号安装函数（如 signal()等）无关，只与信号值有关。

用户进程对信号的响应可以有 3 种方式。

- 忽略信号，即对信号不做任何处理，但是有两个信号不能忽略，即 SIGKILL 及 SIGSTOP。
- 捕捉信号，定义信号处理函数，当信号发生时，执行相应的自定义处理函数。
- 执行缺省操作，Linux 对每种信号都规定了默认操作。

Linux 中的大多数信号是提供给内核的，表 8.6 列出了 Linux 中最为常见信号的含义及其默认操作。

表 8.6　　常见信号的含义及其默认操作

信 号 名	含　义	默认操作
SIGHUP	该信号在用户终端连接（正常或非正常）结束时发出，通常是在终端的控制进程结束时，通知同一会话内的各个作业与控制终端不再关联	终止
SIGINT	该信号在用户键入 INTR 字符（通常是 Ctrl-C）时发出，终端驱动程序发送此信号并送到前台进程中的每一个进程	终止
SIGQUIT	该信号和 SIGINT 类似，但由 QUIT 字符（通常是 Ctrl-\）来控制	终止
SIGILL	该信号在一个进程企图执行一条非法指令时（可执行文件本身出现错误，或者试图执行数据段、堆栈溢出时）发出	终止
SIGFPE	该信号在发生致命的算术运算错误时发出。这里不仅包括浮点运算错误，还包括溢出及除数为 0 等其他所有的算术错误	终止
SIGKILL	该信号用来立即结束程序的运行，并且不能被阻塞、处理或忽略	终止
SIGALRM	该信号当一个定时器到时的时候发出	终止
SIGSTOP	该信号用于暂停一个进程，且不能被阻塞、处理或忽略	暂停进程
SIGTSTP	该信号用于交互停止进程，用户键入 SUSP 字符时（通常是 Ctrl+Z）发出这个信号	停止进程
SIGCHLD	子进程改变状态时，父进程会收到这个信号	忽略
SIGABORT	进程异常终止时发出	

8.3.2　信号发送与捕捉

发送信号的函数主要有 kill()、raise()、alarm()以及 pause()，下面就依次对其进行介绍。

1．kill()和 raise()

（1）函数说明。

kill()函数同读者熟知的 kill 系统命令一样，可以发送信号给进程或进程组（实际上，kill 系统命令只是 kill()函数的一个用户接口）。这里需要注意的是，它不仅可以中止进程（实际上发出 SIGKILL 信号），也可以向进程发送其他信号。

与 kill()函数所不同的是，raise()函数允许进程向自身发送信号。

（2）函数格式。

表 8.7 列出了 kill()函数的语法要点。

表 8.7　　kill()函数语法要点

所需头文件	#include <signal.h> #include <sys/types.h>	
函数原型	int kill(pid_t pid, int sig)	
函数传入值	pid：	正数：要发送信号的进程号
		0：信号被发送到所有和当前进程在同一个进程组的进程
		–1：信号发给所有的进程表中的进程（除了进程号最大的进程外
		<-1：信号发送给进程组号为-pid 的每一个进程
	sig：信号	
函数返回值	成功：0	
	出错：–1	

表 8.8 列出了 raise()函数的语法要点。

表 8.8　　raise()函数语法要点

所需头文件	#include <signal.h> #include <sys/types.h>
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

（3）函数实例。

下面这个示例首先使用 fork()创建了一个子进程，接着为了保证子进程不在父进程调用 kill()之前退出，在子进程中使用 raise()函数向自身发送 SIGSTOP 信号，使子进程暂停。接下来再在父进程中调用 kill()向子进程发送信号，在该示例中使用的是 SIGKILL，读者可以使用其他信号进行练习。

```
/* kill_raise.c */
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid;
    int ret;

    /* 创建一子进程 */
    if ((pid = fork()) < 0)
    {
        printf("Fork error\n");
        exit(1);
```

```
    }

    if (pid == 0)
    {
        /* 在子进程中使用 raise()函数发出 SIGSTOP 信号,使子进程暂停 */
        printf("Child(pid : %d) is waiting for any signal\n", getpid());
        raise(SIGSTOP);
        exit(0);
    }
    else
    {
        /* 在父进程中收集子进程发出的信号，并调用 kill()函数进行相应的操作 */
        if ((waitpid(pid, NULL, WNOHANG)) == 0)
        {
            if ((ret = kill(pid, SIGKILL)) == 0)
            {
                printf("Parent kill %d\n",pid);
            }
        }

        waitpid(pid, NULL, 0);
        exit(0);
    }
}
```

该程序运行结果如下所示：

```
$ ./kill_raise
Child(pid : 4877) is waiting for any signal
Parent kill 4877
```

2．alarm()和 pause()

（1）函数说明。

alarm()也称为闹钟函数，它可以在进程中设置一个定时器，当定时器指定的时间到时，它就向进程发送 SIGALARM 信号。要注意的是，一个进程只能有一个闹钟时间，如果在调用 alarm()之前已设置过闹钟时间，则任何以前的闹钟时间都被新值所代替。

pause()函数是用于将调用进程挂起直至捕捉到信号为止。这个函数很常用，通常可以用于判断信号是否已到。

（2）函数格式。

表 8.9 列出了 alarm()函数的语法要点。

表 8.9　　alarm()函数语法要点

所需头文件	#include <unistd.h>
函数原型	unsigned int alarm(unsigned int seconds)
函数传入值	seconds：指定秒数，系统经过 seconds 秒之后向该进程发送 SIGALRM 信号
函数返回值	成功：如果调用此 alarm()前，进程中已经设置了闹钟时间，则返回上一个闹钟时间的剩余时间，否则返回 0
	出错：–1

表 8.10 列出了 pause()函数的语法要点。

表 8.10　　pause()函数语法要点

所需头文件	#include <unistd.h>
函数原型	int pause(void)
函数返回值	–1，并且把 error 值设为 EINTR

（3）函数实例。

该实例实际上已完成了一个简单的 sleep()函数的功能，由于 SIGALARM 默认的系统动作为终止该进程，因此程序在打印信息之前，就会被结束了。代码如下所示：

```
/* alarm_pause.c */
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    /*调用 alarm 定时器函数*/
    int ret = alarm(5);
    pause();
    printf("I have been waken up.\n",ret); /* 此语句不会被执行 */
}
$./alarm_pause
Alarm clock
```

想一想　用这种形式实现的 sleep()功能有什么问题？

8.3.3　信号的处理

在了解了信号的产生与捕获之后，接下来就要对信号进行具体的操作了。从前面的信号概述中读者也可以看到，特定的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该进程中需要对哪些信号进行什么样的处理。例如，一个进程可以选择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，这些都是与特定的进程相联系的。因此，首先就要建立进程与其信号之间的对应关系，这就是信号的处理。

注意 请读者注意信号的注册与信号的处理之间的区别，前者信号是主动方，而后者进程是主动方。信号的注册是在进程选择了特定信号处理之后特定信号的主动行为。

信号处理的主要方法有两种，一种是使用简单的 signal()函数，另一种是使用信号集函数组。下面分别介绍这两种处理方式。

1．信号处理函数

（1）函数说明。

使用 signal()函数处理时，只需要指出要处理的信号和处理函数即可。它主要是用于前 32 种非实时信号的处理，不支持信号传递信息，但是由于使用简单、易于理解，因此也受到很多程序员的欢迎。

Linux 还支持一个更健壮、更新的信号处理函数 sigaction()，推荐使用该函数。

（2）函数格式。

signal()函数的语法要点如表 8.11 所示。

表 8.11 signal()函数语法要点

所需头文件	#include <signal.h>	
函数原型	void (*signal(int signum, void (*handler)(int)))(int)	
函数传入值	signum：指定信号代码	
	handler：	SIG_IGN：忽略该信号
		SIG_DFL：采用系统默认方式处理信号
		自定义的信号处理函数指针
函数返回值	成功：以前的信号处理配置	
	出错：−1	

这里需要对这个函数原型进行说明。这个函数原型有点复杂。可先用如下的 typedef 进行替换说明：

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

可见，首先该函数原型整体指向一个无返回值并且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。

表 8.12 列举了 sigaction()的语法要点。

表 8.12 sigaction()函数语法要点

所需头文件	#include <signal.h>
函数原型	int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)
函数传入值	signum：信号代码，可以为除 SIGKILL 及 SIGSTOP 外的任何一个特定有效的信号

	act：指向结构 sigaction 的一个实例的指针，指定对特定信号的处理
	oldact：保存原来对相应信号的处理
函数返回值	成功：0
	出错：–1

这里要说明的是 sigaction()函数中第 2 个和第 3 个参数用到的 sigaction 结构。这是一个看似非常复杂的结构，希望读者能够慢慢阅读此段内容。

首先给出了 sigaction 的定义，如下所示：

```
struct sigaction
{
    void (*sa_handler)(int signo);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restore)(void);
}
```

sa_handler 是一个函数指针，指定信号处理函数，这里除可以是用户自定义的处理函数外，还可以为 SIG_DFL（采用缺省的处理方式）或 SIG_IGN（忽略信号）。它的处理函数只有一个参数，即信号值。

sa_mask 是一个信号集，它可以指定在信号处理程序执行过程中哪些信号应当被屏蔽，在调用信号捕获函数之前，该信号集要加入到信号的信号屏蔽字中。

sa_flags 中包含了许多标志位，是对信号进行处理的各个选择项。它的常见可选值如表 8.13 所示。

表 8.13　　常见信号的含义及其默认操作

选　　项	含　　义
SA_NODEFER \ SA_NOMASK	当捕捉到此信号时，在执行其信号捕捉函数时，系统不会自动屏蔽此信号
SA_NOCLDSTOP	进程忽略子进程产生的任何 SIGSTOP、SIGTSTP、SIGTTIN 和 SIGTTOU 信号
SA_RESTART	令重启的系统调用起作用
SA_ONESHOT \ SA_RESETHAND	自定义信号只执行一次，在执行完毕后恢复信号的系统默认动作

（3）使用实例。

第一个实例表明了如何使用 signal()函数捕捉相应信号，并做出给定的处理。这里，my_func 就是信号处理的函数指针。读者还可以将其改为 SIG_IGN 或 SIG_DFL 查看运行结果。第二个实例是用 sigaction()函数实现同样的功能。

以下是使用 signal()函数的示例：

```
/* signal.c */
#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();
    exit(0);
}
```

运行结果如下所示。

```
$ ./signal
Waiting for signal SIGINT or SIGQUIT...
I have get SIGINT （按 ctrl-c 组合键）
$ ./signal
Waiting for signal SIGINT or SIGQUIT...
I have get SIGQUIT （按 ctrl-\ 组合键）
```

以下是用 sigaction()函数实现同样的功能，下面只列出更新的 main()函数部分。

```
/* sigaction.c */
/* 前部分省略 */
int main()
{
    struct sigaction action;
    printf("Waiting for signal SIGINT or SIGQUIT...\n");
```

```
    /* sigaction 结构初始化 */
    action.sa_handler = my_func;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;

    /* 发出相应的信号，并跳转到信号处理函数处 */
    sigaction(SIGINT, &action, 0);
    sigaction(SIGQUIT, &action, 0);
    pause();
    exit(0);
}
```

2．信号集函数组

（1）函数说明。

使用信号集函数组处理信号时涉及一系列的函数，这些函数按照调用的先后次序可分为以下几大功能模块：创建信号集合、注册信号处理函数以及检测信号。

其中，创建信号集合主要用于处理用户感兴趣的一些信号，其函数包括以下几个。

- sigemptyset()：将信号集合初始化为空。
- sigfillset()：将信号集合初始化为包含所有已定义的信号的集合。
- sigaddset()：将指定信号加入到信号集合中去。
- sigdelset()：将指定信号从信号集合中删除。
- sigismember()：查询指定信号是否在信号集合之中。

注册信号处理函数主要用于决定进程如何处理信号。这里要注意的是，信号集里的信号并不是真正可以处理的信号，只有当信号的状态处于非阻塞状态时才会真正起作用。因此，首先使用 sigprocmask()函数检测并更改信号屏蔽字（信号屏蔽字是用来指定当前被阻塞的一组信号，它们不会被进程接收），然后使用 sigaction()函数来定义进程接收到特定信号之后的行为。检测信号是信号处理的后续步骤，因为被阻塞的信号不会传递给进程，所以这些信号就处于“未处理”状态（也就是进程不清楚它的存在）。sigpending()函数允许进程检测“未处理”信号，并进一步决定对它们作何处理。

（2）函数格式。

首先介绍创建信号集合的函数格式，表 8.14 列举了这一组函数的语法要点。

表 8.14　　创建信号集合函数语法要点

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：指定信号代码	
函数返回值	成功：0（sigismember 成功返回 1，失败返回 0）	
	出错：–1	

表 8.15 列举了 sigprocmask 的语法要点。

表 8.15　　sigprocmask 函数语法要点

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	

此处，若 set 是一个非空指针，则参数 how 表示函数的操作方式；若 how 为空，则表示忽略此操作。

最后，表 8.16 列举了 sigpending 函数的语法要点。

表 8.16　　sigpending 函数语法要点

所需头文件	#include <signal.h>
函数原型	int sigpending(sigset_t *set)
函数传入值	set：要检测的信号集
函数返回值	成功：0
	出错：–1

总之，在处理信号时，一般遵循如图 8.7 所示的操作流程。

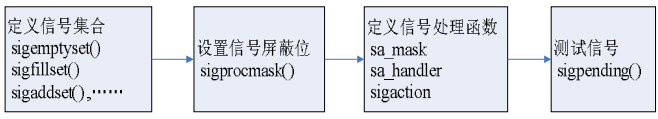

图 8.7　一般的信号操作处理流程

（3）使用实例。

该实例首先把 SIGQUIT、SIGINT 两个信号加入信号集，然后将该信号集合设为阻塞状态，并进入用户输入状态。用户只需按任意键，就可以立刻将信号集合设置为

非阻塞状态，再对这两个信号分别操作，其中 SIGQUIT 执行默认操作，而 SIGINT 执行用户自定义函数的操作。源代码如下所示：

```
/* sigset.c */
#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*/
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;

    /* 初始化信号集为空 */
    if (sigemptyset(&set) < 0)
    {
        perror("sigemptyset");
        exit(1);
    }

    /* 将相应的信号加入信号集 */
    if (sigaddset(&set, SIGQUIT) < 0)
    {
        perror("sigaddset");
        exit(1);
    }

    if (sigaddset(&set, SIGINT) < 0)
    {
        perror("sigaddset");
        exit(1);
    }
```

```
    if (sigismember(&set, SIGINT))
    {
        sigemptyset(&action1.sa_mask);
        action1.sa_handler = my_func;
        action1.sa_flags = 0;
        sigaction(SIGINT, &action1, NULL);
    }

    if (sigismember(&set, SIGQUIT))
    {
        sigemptyset(&action2.sa_mask);
        action2.sa_handler = SIG_DFL;
        action2.sa_flags = 0;
        sigaction(SIGQUIT, &action2,NULL);
    }

    /* 设置信号集屏蔽字，此时 set 中的信号不会被传递给进程，暂时进入待处理状态 */
    if (sigprocmask(SIG_BLOCK, &set, NULL) < 0)
    {
        perror("sigprocmask");
        exit(1);
    }
    else
    {
        printf("Signal set was blocked, Press any key!");
        getchar();
    }

    /* 在信号屏蔽字中删除 set 中的信号 */
    if (sigprocmask(SIG_UNBLOCK, &set, NULL) < 0)
    {
        perror("sigprocmask");
        exit(1);
    }
    else
    {
        printf("Signal set is in unblock state\n");
    }

    while(1);
```

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

该程序的运行结果如下所示，可以看见，在信号处于阻塞状态时，所发出的信号对进程不起作用，并且该信号进入待处理状态。读者输入任意键，并且信号脱离了阻塞状态之后，用户发出的信号才能正常运行。这里 SIGINT 已按照用户自定义的函数运行，请读者注意阻塞状态下 SIGINT 的处理和非阻塞状态下 SIGINT 的处理有何不同。

```
    $ ./sigset
    Signal set was blocked, Press any key!     /* 此时按任何键可以解除阻塞屏蔽字 */
    If you want to quit,please try SIGQUIT   /* 阻塞状态下 SIGINT 的处理*/
    Signal set is in unblock state              /* 从信号屏蔽字中删除 set 中的信
号 */
    If you want to quit,please try SIGQUIT   /* 非阻塞状态下 SIGINT 的处理 */
    If you want to quit,please try SIGQUIT
    Quit                                        /* 非阻塞状态下 SIGQUIT 处理 */
```

8.4 信号量

8.4.1 信号量概述

在多任务操作系统环境下，多个进程会同时运行，并且一些进程之间可能存在一定的关联。多个进程可能为了完成同一个任务会相互协作，这样形成进程之间的同步关系。而且在不同进程之间，为了争夺有限的系统资源（硬件或软件资源）会进入竞争状态，这就是进程之间的互斥关系。

进程之间的互斥与同步关系存在的根源在于临界资源。临界资源是在同一个时刻只允许有限个（通常只有一个）进程可以访问（读）或修改（写）的资源，通常包括硬件资源（处理器、内存、存储器以及其他外围设备等）和软件资源（共享代码段，共享结构和变量等）。访问临界资源的代码叫做临界区，临界区本身也会成为临界资源。

信号量是用来解决进程之间的同步与互斥问题的一种进程之间通信机制，包括一个称为信号量的变量和在该信号量下等待资源的进程等待队列，以及对信号量进行的两个原子操作（PV 操作）。其中信号量对应于某一种资源，取一个非负的整型值。信号量值指的是当前可用的该资源的数量，若它等于 0 则意味着目前没有可用的资源。PV 原子操作的具体定义如下：

P 操作：如果有可用的资源（信号量值>0），则占用一个资源（给信号量值减去一，进入临界区代码）;如果没有可用的资源（信号量值等于 0），则被阻塞到，直到系统将资源分配给该进程（进入等待队列，一直等到资源轮到该进程）。

V 操作：如果在该信号量的等待队列中有进程在等待资源，则唤醒一个阻塞进程。如果没有进程等待它，则释放一个资源（给信号量值加一）。

使用信号量访问临界区的伪代码所下所示：

```
{
    /* 设 R 为某种资源，S 为资源 R 的信号量*/
    INIT_VAL(S);            /* 对信号量 S 进行初始化 */
    非临界区;
    P(S);                   /* 进行 P 操作 */
    临界区（使用资源 R）;    /* 只有有限个（通常只有一个）进程被允许进入该区*/
    V(S);                   /* 进行 V 操作 */
    非临界区;
}
```

最简单的信号量是只能取 0 和 1 两种值，这种信号量被叫做二维信号量。在本节中，主要讨论二维信号量。二维信号量的应用比较容易地扩展到使用多维信号量的情况。

8.4.2 信号量的应用

1．函数说明

在 Linux 系统中，使用信号量通常分为以下几个步骤。

（1）创建信号量或获得在系统已存在的信号量，此时需要调用 semget()函数。不同进程通过使用同一个信号量键值来获得同一个信号量。

（2）初始化信号量，此时使用 semctl()函数的 SETVAL 操作。当使用二维信号量时，通常将信号量初始化为 1。

（3）进行信号量的 PV 操作，此时调用 semop()函数。这一步是实现进程之间的同步和互斥的核心工作部分。

（4）如果不需要信号量，则从系统中删除它，此时使用 semclt()函数的 IPC_RMID 操作。此时需要注意，在程序中不应该出现对已经被删除的信号量的操作。

2．函数格式

表 8.17 列举了 semget()函数的语法要点。

表 8.17　　semget()函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semget(key_t key, int nsems, int semflg)
函数传入值	key：信号量的键值，多个进程可以通过它访问同一个信号量，其中有个特殊值 IPC_PRIVATE。它用于创建当前进程的私有信号量
	nsems：需要创建的信号量数目，通常取值为 1

	semflg：同 open()函数的权限位，也可以用八进制表示法，其中使用 IPC_CREAT 标志创建新的信号量，即使该信号量已经存在（具有同一个键值的信号量已在系统中存在），也不会出错。如果同时使用 IPC_EXCL 标志可以创建一个新的唯一的信号量，此时如果该信号量已经存在，该函数会返回出错
函数返回值	成功：信号量标识符，在信号量的其他函数中都会使用该值
	出错：-1

表 8.18 列举了 semctl()函数的语法要点。

表 8.18　　semctl()函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semctl(int semid, int semnum, int cmd, union semun arg)
函数传入值	semid：semget()函数返回的信号量标识符
	semnum：信号量编号，当使用信号量集时才会被用到。通常取值为 0，就是使用单个信号量（也是第一个信号量）
	cmd：指定对信号量的各种操作，当使用单个信号量（而不是信号量集）时，常用的有以下几种： IPC_STAT：获得该信号量（或者信号量集合）的 semid_ds 结构，并存放在由第 4 个参数 arg 的 buf 指向的 semid_ds 结构中。semid_ds 是在系统中描述信号量的数据结构。 IPC_SETVAL：将信号量值设置为 arg 的 val 值 IPC_GETVAL：返回信号量的当前值 IPC_RMID：从系统中，删除信号量（或者信号量集）

	arg：是 union semnn 结构，该结构可能在某些系统中并不给出定义，此时必须由程序员自己定义
	union semun
	{
	int val;
	struct semid_ds *buf;
	unsigned short *array;
	}

函数返回值	成功：根据 cmd 值的不同而返回不同的值 IPC_STAT、IPC_SETVAL、IPC_RMID：返回 0 IPC_GETVAL：返回信号量的当前值
	出错：-1

表 8.19 列举了 semop()函数的语法要点。

表 8.19　　semop()函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h>
函数原型	int semop(int semid, struct sembuf *sops, size_t nsops)
函数传入值	semid：semget()函数返回的信号量标识符

	sops：指向信号量操作数组，一个数组包括以下成员：
	struct sembuf
	{
	short sem_num;　/* 信号量编号，使用单个信号量时，通常取值为 0 */
	short sem_op;
	/* 信号量操作：取值为-1 则表示 P 操作，取值为+1 则表示 V 操作*/
	short sem_flg;
	/* 通常设置为 SEM_UNDO。这样在进程没释放信号量而退出时，系统自动
	释放该进程中未释放的信号量 */
	}

	nsops：操作数组 sops 中的操作个数（元素数目），通常取值为 1（一个操作）
函数返回值	成功：信号量标识符，在信号量的其他函数中都会使用该值
	出错：-1

3．使用实例

本实例说明信号量的概念以及基本用法。在实例程序中，首先创建一个子进程，接下来使用信号量来控制两个进程（父子进程）之间的执行顺序。

因为信号量相关的函数调用接口比较复杂，我们可以将它们封装成二维单个信号量的几个基本函数。它们分别为信号量初始化函数（或者信号量赋值函数）init_sem()、P 操作函数 sem_p()、V 操作函数 sem_v()以及删除信号量的函数 del_sem()等，具体实现如下所示：

```
/* sem_com.c */
#include "sem_com.h"
/* 信号量初始化（赋值）函数*/
int init_sem(int sem_id, int init_value)
{
    union semun sem_union;
    sem_union.val = init_value;     /* init_value 为初始值 */
    if (semctl(sem_id, 0, SETVAL, sem_union) == -1)
    {
        perror("Initialize semaphore");
        return -1;
    }
    return 0;
}
/* 从系统中删除信号量的函数 */
int del_sem(int sem_id)
{
```

```
    union semun sem_union;
    if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
    {
        perror("Delete semaphore");
        return -1;
    }
}
/* P 操作函数 */
int sem_p(int sem_id)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0;  /* 单个信号量的编号应该为 0 */
    sem_b.sem_op = -1;  /* 表示 P 操作 */
    sem_b.sem_flg = SEM_UNDO; /* 系统自动释放将会在系统中残留的信号量*/
    if (semop(sem_id, &sem_b, 1) == -1)
    {
        perror("P operation");
        return -1;
    }
    return 0;
}
/* V 操作函数*/
int sem_v(int sem_id)
{
    struct sembuf sem_b;
    sem_b.sem_num = 0; /* 单个信号量的编号应该为 0 */
    sem_b.sem_op = 1; /* 表示 V 操作 */
    sem_b.sem_flg = SEM_UNDO; /* 系统自动释放将会在系统中残留的信号量*/
    if (semop(sem_id, &sem_b, 1) == -1)
    {
        perror("V operation");
        return -1;
    }
    return 0;
}
```

现在我们调用这些简单易用的接口，可以轻松解决控制两个进程之间的执行顺序的同步问题。实现代码如下所示：

```
/* fork.c */
#include <sys/types.h>
```

```
    #include <unistd.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/types.h>
    #include <sys/ipc.h>
    #include <sys/shm.h>
    #define DELAY_TIME    3       /* 为了突出演示效果，等待几秒钟，*/

    int main(void)
    {
       pid_t result;
       int sem_id;

       sem_id = semget(ftok(".", 'a'),  1, 0666|IPC_CREAT); /* 创建一个信号量
*/
       init_sem(sem_id, 0);

       /*调用 fork()函数*/
       result = fork();
       if(result ==  -1)
       {
          perror("Fork\n");
       }
       else if (result == 0) /*返回值为 0 代表子进程*/
       {
          printf("Child process will wait for some seconds...\n");
          sleep(DELAY_TIME);
          printf("The returned value is %d in the child process(PID =
%d)\n",
     result, getpid());
          sem_v(sem_id);
       }
       else /*返回值大于 0 代表父进程*/
       {
          sem_p(sem_id);
          printf("The returned value is %d in the father process(PID =
%d)\n",
     result, getpid());
          sem_v(sem_id);
          del_sem(sem_id);
```

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

读者可以先从该程序中删除掉信号量相关的代码部分并观察运行结果。

```
    $ ./simple_fork
    Child process will wait for some seconds… /*子进程在运行中*/
    The returned value is 4185 in the father process(PID = 4184)/*父进程先结束*/
    […]$ The returned value is 0 in the child process(PID = 4185) /* 子进程后结束了*/
```

再添加信号量的控制部分并运行结果。

```
    $ ./sem_fork
    Child process will wait for some seconds…
                                        /*子进程在运行中，父进程在等待子进程结束*/
    The returned value is 0 in the child process(PID = 4185) /* 子进程结束了*/
    The returned value is 4185 in the father process(PID = 4184) /* 父进程结束*/
```

本实例说明使用信号量怎么解决多进程之间存在的同步问题。我们将在后面讲述的共享内存和消息队列的实例中，看到使用信号量实现多进程之间的互斥。

8.5 共享内存

8.5.1 共享内存概述

可以说，共享内存是一种最为高效的进程间通信方式。因为进程可以直接读写内存，不需要任何数据的复制。为了在多个进程间交换信息，内核专门留出了一块内存区。这段内存区可以由需要访问的进程将其映射到自己的私有地址空间。因此，进程就可以直接读写这一内存区而不需要进行数据的复制，从而大大提高了效率。当然，由于多个进程共享一段内存，因此也需要依靠某种同步机制，如互斥锁和信号量等（请参考本章的共享内存实验）。其原理示意图如图 8.8 所示。

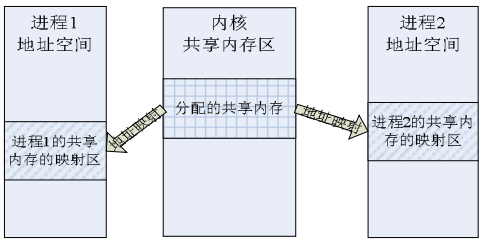

图 8.8　共享内存原理示意图

8.5.2 共享内存的应用

1. 函数说明

共享内存的实现分为两个步骤，第一步是创建共享内存，这里用到的函数是 shmget()，也就是从内存中获得一段共享内存区域，第二步映射共享内存，也就是把这段创建的共享内存映射到具体的进程空间中，这里使用的函数是 shmat()。到这里，就可以使用这段共享内存了，也就是可以使用不带缓冲的 I/O 读写命令对其进行操作。除此之外，当然还有撤销映射的操作，其函数为 shmdt()。这里就主要介绍这 3 个函数。

2. 函数格式

表 8.20 列举了 shmget()函数的语法要点。

表 8.20　　shmget()函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int shmget(key_t key, int size, int shmflg)
函数传入值	key：共享内存的键值，多个进程可以通过它访问同一个共享内存，其中有个特殊值 IPC_PRIVATE。它用于创建当前进程的私有共享内存
	size：共享内存区大小
	shmflg：同 open()函数的权限位，也可以用八进制表示法
函数返回值	成功：共享内存段标识符
	出错：–1

表 8.21 列举了 shmat()函数的语法要点。

表 8.21　　shmat()函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	char *shmat(int shmid, const void *shmaddr, int shmflg)	
函数传入值	shmid：要映射的共享内存区标识符	
	shmaddr：将共享内存映射到指定地址（若为 0 则表示系统自动分配地址并把段共享内存映射到调用进程的地址空间）	
	shmflg	SHM_RDONLY：共享内存只读
		默认 0：共享内存可读写
函数返回值	成功：被映射的段地址	
	出错：–1	

表 8.22 列举了 shmdt()函数的语法要点。

表 8.22　　shmdt()函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int shmdt(const void *shmaddr)
函数传入值	shmaddr：被映射的共享内存段地址
函数返回值	成功：0
	出错：−1

3．使用实例

该实例说明如何使用基本的共享内存函数。首先是创建一个共享内存区（采用的共享内存的键值为 IPC_PRIVATE，是因为本实例中创建的共享内存是父子进程之间的共用部分），之后创建子进程，在父子两个进程中将共享内存分别映射到各自的进程地址空间之中。

父进程先等待用户输入，然后将用户输入的字符串写入到共享内存，之后往共享内存的头部写入“WROTE”字符串表示父进程已成功写入数据。子进程一直等到共享内存的头部字符串为“WROTE”，然后将共享内存的有效数据（在父进程中用户输入的字符串）在屏幕上打印。父子两个进程在完成以上工作之后，分别解除与共享内存的映射关系。

最后在子进程中删除共享内存。因为共享内存自身并不提供同步机制，所以应该额外实现不同进程之间的同步（例如：信号量）。为了简单起见，在本实例中用标志字符串来实现非常简单的父子进程之间的同步。

这里要介绍的一个命令是 ipcs，这是用于报告进程间通信机制状态的命令。它可以查看共享内存、消息队列等各种进程间通信机制的情况，这里使用了 system()函数用于调用 shell 命令“ipcs”。程序源代码如下所示：

```
/* shmem.c */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFFER_SIZE 2048

int main()
{
    pid_t pid;
    int shmid;
    char *shm_addr;
    char flag[] = "WROTE";
```

```
    char *buff;

    /* 创建共享内存 */
    if ((shmid = shmget(IPC_PRIVATE, BUFFER_SIZE, 0666)) < 0)
    {
        perror("shmget");
        exit(1);
    }
    else
    {
        printf("Create shared-memory: %d\n",shmid);
    }

    /* 显示共享内存情况 */
    system("ipcs -m");

    pid = fork();
    if (pid == -1)
    {
        perror("fork");
        exit(1);
    }
    else if (pid == 0) /* 子进程处理 */
    {
        /*映射共享内存*/
        if ((shm_addr = shmat(shmid, 0, 0)) == (void*)-1)
        {
            perror("Child: shmat");
            exit(1);
        }
        else
        {
            printf("Child: Attach shared-memory: %p\n", shm_addr);
        }
        system("ipcs -m");

        /* 通过检查在共享内存的头部是否标志字符串"WROTE"来确认
父进程已经向共享内存写入有效数据 */
        while (strncmp(shm_addr, flag, strlen(flag)))
        {
```

```
            printf("Child: Wait for enable data...\n");
            sleep(5);
        }

        /* 获取共享内存的有效数据并显示 */
        strcpy(buff, shm_addr + strlen(flag));
        printf("Child: Shared-memory :%s\n", buff);

        /* 解除共享内存映射 */
        if ((shmdt(shm_addr)) < 0)
        {
            perror("shmdt");
            exit(1);
        }
        else
        {
            printf("Child: Deattach shared-memory\n");
        }
        system("ipcs -m");

        /* 删除共享内存 */
        if (shmctl(shmid, IPC_RMID, NULL) == -1)
        {
            perror("Child: shmctl(IPC_RMID)\n");
            exit(1);
        }
        else
        {
            printf("Delete shared-memory\n");
        }

        system("ipcs -m");
    }
    else /* 父进程处理 */
    {
        /*映射共享内存*/
        if ((shm_addr = shmat(shmid, 0, 0)) == (void*)-1)
        {
            perror("Parent: shmat");
            exit(1);
```

```
        }
        else
        {
            printf("Parent: Attach shared-memory: %p\n", shm_addr);
        }

        sleep(1);
        printf("\nInput some string:\n");
        fgets(buff, BUFFER_SIZE, stdin);
        strncpy(shm_addr + strlen(flag), buff, strlen(buff));
        strncpy(shm_addr, flag, strlen(flag));

        /* 解除共享内存映射 */
        if ((shmdt(shm_addr)) < 0)
        {
            perror("Parent: shmdt");
            exit(1);
        }
        else
        {
            printf("Parent: Deattach shared-memory\n");
        }
        system("ipcs -m");

        waitpid(pid, NULL, 0);
        printf("Finished\n");
    }

    exit(0);
}
```

下面是运行结果。从该结果可以看出，nattch 的值随着共享内存状态的变化而变化，共享内存的值根据不同的系统会有所不同。

```
$ ./shmem
Create shared-memory: 753665
/* 在刚创建共享内存时（尚未有任何地址映射）共享内存的情况 */
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 753665     david      666        2048       0
```

```
Child: Attach shared-memory: 0xb7f59000 /* 共享内存的映射地址 */
Parent: Attach shared-memory: 0xb7f59000
/* 在父子进程中进行共享内存的地址映射之后共享内存的情况*/
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 753665     david      666        2048       2

Child: Wait for enable data...

Input some string:
Hello /* 用户输入字符串“Hello” */
Parent: Deattach shared-memory
/* 在父进程中解除共享内存的映射关系之后共享内存的情况 */
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 753665     david      666        2048       1
/*在子进程中读取共享内存的有效数据并打印*/
Child: Shared-memory :hello

Child: Deattach shared-memory
/* 在子进程中解除共享内存的映射关系之后共享内存的情况 */
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 753665     david      666        2048       0

Delete shared-memory
/* 在删除共享内存之后共享内存的情况 */
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

Finished
```

8.6 消息队列

8.6.1 消息队列概述

顾名思义，消息队列就是一些消息的列表。用户可以从消息队列中添加消息和读取消息等。从这点上看，消息队列具有一定的 FIFO 特性，但是它可以实现消息的随机查询，比 FIFO 具有更大的优势。同时，这些消息又是存在于内核中的，由“队列 ID”来标识。

8.6.2 消息队列的应用

1. 函数说明

消息队列的实现包括创建或打开消息队列、添加消息、读取消息和控制消息队列这 4 种操作。其中创建或打开消息队列使用的函数是 msgget()，这里创建的消息队列的数量会受到系统消息队列数量的限制；添加消息使用的函数是 msgsnd()函数，它把消息添加到已打开的消息队列末尾；读取消息使用的函数是 msgrcv()，它把消息从消息队列中取走，与 FIFO 不同的是，这里可以指定取走某一条消息；最后控制消息队列使用的函数是 msgctl()，它可以完成多项功能。

2. 函数格式

表 8.23 列举了 msgget()函数的语法要点。

表 8.23 msgget()函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int msgget(key_t key, int msgflg)
函数传入值	key：消息队列的键值，多个进程可以通过它访问同一个消息队列，其中有个特殊值 IPC_PRIVATE。它用于创建当前进程的私有消息队列
	msgflg：权限标志位
函数返回值	成功：消息队列 ID
	出错：−1

表 8.24 列举了 msgsnd()函数的语法要点。

表 8.24 msgsnd()函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)	
函数传入值	msqid：消息队列的队列 ID	

	msgp：指向消息结构的指针。该消息结构 msgbuf 通常为：
	struct msgbuf
	{
	long mtype; /* 消息类型，该结构必须从这个域开始 */
	char mtext[1]; /* 消息正文 */
	}	

	msgsz：消息正文的字节数（不包括消息类型指针变量）	
	msgflg：	IPC_NOWAIT 若消息无法立即发送（比如：当前消息队列已满），函数会立即返回
		0：msgsnd 调阻塞直到发送成功为止
函数返回值	成功：0	
	出错：−1	

表 8.25 列举了 msgrcv()函数的语法要点。

表 8.25　　msgrcv()函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgrcv(int msgid, void *msgp, size_t msgsz, long int msgtyp, int msgflg)	
函数传入值	msqid：消息队列的队列 ID	
	msgp：消息缓冲区，同于 msgsnd()函数的 msgp	
	msgsz：消息正文的字节数（不包括消息类型指针变量）	
	msgtyp：	0：接收消息队列中第一个消息
		大于 0：接收消息队列中第一个类型为 msgtyp 的消息
		小于 0：接收消息队列中第一个类型值不小于 msgtyp 绝对值且类型值又最小的消息
函数传入值	msgflg：	MSG_NOERROR：若返回的消息比 msgsz 字节多，则消息就会截短到 msgsz 字节，且不通知消息发送进程
		IPC_NOWAIT 若在消息队列中并没有相应类型的消息可以接收，则函数立即返回
		0：msgsnd()调用阻塞直到接收一条相应类型的消息为止
函数返回值	成功：0	
	出错：–1	

表 8.26 列举了 msgctl()函数的语法要点。

表 8.26　　msgctl()函数语法要点

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>	
函数原型	int msgctl (int msgqid, int cmd, struct msqid_ds *buf)	
函数传入值	msqid：消息队列的队列 ID	
	cmd： 命令参数	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 域（IPC 操作权限描述结构）值。这个值取自 buf 参数
		IPC_RMID：从系统内核中删除消息队列
	buf：描述消息队列的 msgqid_ds 结构类型变量	
函数返回值	成功：0	
	出错：–1	

3．使用实例

这个实例体现了如何使用消息队列进行两个进程（发送端和接收端）之间的通信，

包括消息队列的创建、消息发送与读取、消息队列的撤消和删除等多种操作。

消息发送端进程和消息接收端进程之间不需要额外实现进程之间的同步。在该实例中，发送端发送的消息类型设置为该进程的进程号（可以取其他值），因此接收端根据消息类型确定消息发送者的进程号。注意这里使用了函数 fotk()，它可以根据不同的路径和关键字产生标准的 key。以下是消息队列发送端的代码：

```
/* msgsnd.c */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define  BUFFER_SIZE       512

struct message
{
    long msg_type;
    char msg_text[BUFFER_SIZE];
};

int main()
{
    int qid;
    key_t key;
    struct message msg;

    /*根据不同的路径和关键字产生标准的 key*/
    if ((key = ftok(".", 'a')) == -1)
    {
       perror("ftok");
       exit(1);
    }

    /*创建消息队列*/
    if ((qid = msgget(key, IPC_CREAT|0666)) == -1)
    {
       perror("msgget");
       exit(1);
```

```
    }
    printf("Open queue %d\n",qid);

    while(1)
    {
       printf("Enter some message to the queue:");
       if ((fgets(msg.msg_text, BUFFER_SIZE, stdin)) == NULL)
       {
          puts("no message");
          exit(1);
       }

       msg.msg_type = getpid();

       /*添加消息到消息队列*/
       if ((msgsnd(qid, &msg, strlen(msg.msg_text), 0)) < 0)
       {
          perror("message posted");
          exit(1);
       }

       if (strncmp(msg.msg_text, "quit", 4) == 0)
       {
          break;
       }
    }
    exit(0);
}
```

以下是消息队列接收端的代码：

```
/* msgrcv.c */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#define  BUFFER_SIZE       512
```

```
    struct message
    {
        long msg_type;
        char msg_text[BUFFER_SIZE];
    };

    int main()
    {
        int qid;
        key_t key;
        struct message msg;

        /*根据不同的路径和关键字产生标准的 key*/
        if ((key = ftok(".", 'a')) == -1)
        {
           perror("ftok");
           exit(1);
        }

        /*创建消息队列*/
        if ((qid = msgget(key, IPC_CREAT|0666)) == -1)
        {
           perror("msgget");
           exit(1);
        }
        printf("Open queue %d\n", qid);

        do
        {
           /*读取消息队列*/
           memset(msg.msg_text, 0, BUFFER_SIZE);
           if (msgrcv(qid, (void*)&msg, BUFFER_SIZE, 0, 0) < 0)
           {
              perror("msgrcv");
              exit(1);
           }
           printf("The message from process %d : %s", msg.msg_type,
msg.msg_text);

        } while(strncmp(msg.msg_text, "quit", 4));
```

```
    /*从系统内核中移走消息队列 */
    if ((msgctl(qid, IPC_RMID, NULL)) < 0)
    {
       perror("msgctl");
       exit(1);
    }

    exit(0);
}
```

以下是程序的运行结果。输入“quit”则两个进程都将结束。

```
$ ./msgsnd
Open queue 327680
Enter some message to the queue:first message
Enter some message to the queue:second message
Enter some message to the queue:quit
$ ./msgrcv
Open queue 327680
The message from process 6072 : first message
The message from process 6072 : second message
The message from process 6072 : quit
```

8.7 实验内容

8.7.1 管道通信实验

1. 实验目的

通过编写有名管道多路通信实验，读者可进一步掌握管道的创建、读写等操作，同时，也复习使用 select()函数实现管道的通信。

2. 实验内容

读者还记得在 6.3.3 小节中，通过 mknod 命令创建两个管道的实例吗？本实例只是在它的基础上添加有名管道的创建，而不用再输入 mknod 命令。

3. 实验步骤

（1）画出流程图。

该实验流程图如图 8.9 所示。

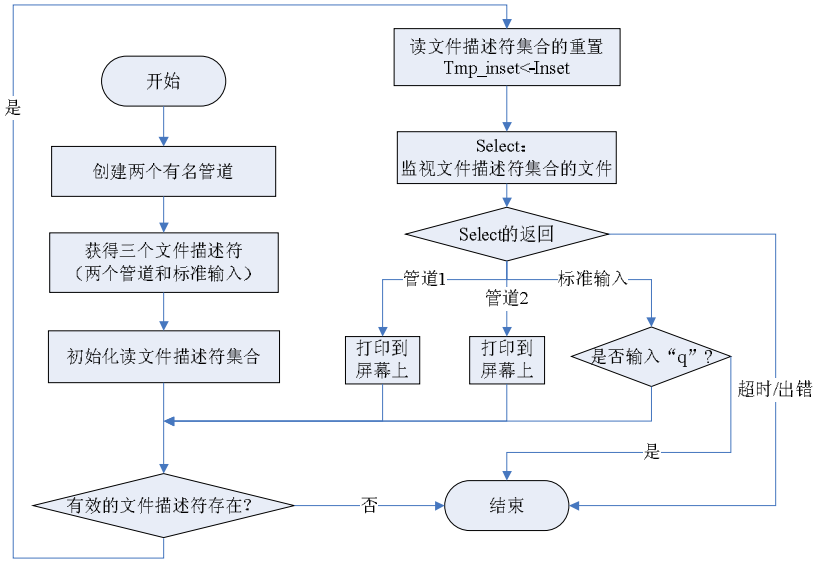

图 8.9　8.6.1 实验流程图

（2）编写代码。

该实验源代码如下所示。

```
/* pipe_select.c*/
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <errno.h>

#define FIFO1              "in1"
#define FIFO2              "in2"
#define MAX_BUFFER_SIZE       1024          /* 缓冲区大小*/
#define IN_FILES           3                /* 多路复用输入文件数目*/
#define TIME_DELAY         60             /* 超时值秒数 */
#define MAX(a, b)          ((a > b)?(a):(b))

int main(void)
{
    int fds[IN_FILES];
    char buf[MAX_BUFFER_SIZE];
    int i, res, real_read, maxfd;
    struct timeval tv;
```

```
    fd_set inset,tmp_inset;

    fds[0] = 0;

    /* 创建两个有名管道 */
    if (access(FIFO1, F_OK) == -1)
    {
       if ((mkfifo(FIFO1, 0666) < 0) && (errno != EEXIST))
       {
          printf("Cannot create fifo file\n");
          exit(1);
       }
    }
    if (access(FIFO2, F_OK) == -1)
    {
       if ((mkfifo(FIFO2, 0666) < 0) && (errno != EEXIST))
       {
          printf("Cannot create fifo file\n");
          exit(1);
       }
    }

    /* 以只读非阻塞方式打开两个管道文件 */
    if((fds[1] = open (FIFO1, O_RDONLY|O_NONBLOCK)) < 0)
    {
       printf("Open in1 error\n");
       return 1;
    }
   if((fds[2] = open (FIFO2, O_RDONLY|O_NONBLOCK)) < 0)
   {
       printf("Open in2 error\n");
       return 1;
    }

    /*取出两个文件描述符中的较大者*/
   maxfd = MAX(MAX(fds[0], fds[1]), fds[2]);
   /*初始化读集合 inset，并在读文件描述符集合中加入相应的描述集*/
   FD_ZERO(&inset);
   for (i = 0; i < IN_FILES; i++)
   {
```

```
            FD_SET(fds[i], &inset);
        }
        FD_SET(0, &inset);

        tv.tv_sec = TIME_DELAY;
        tv.tv_usec = 0;
        /*循环测试该文件描述符是否准备就绪，并调用 select()函数对相关文件描述符做相应操作*/
        while(FD_ISSET(fds[0],&inset)
                || FD_ISSET(fds[1],&inset) || FD_ISSET(fds[2], &inset))
        {
            /* 文件描述符集合的备份， 免得每次进行初始化 */
            tmp_inset = inset;
            res = select(maxfd + 1, &tmp_inset, NULL, NULL, &tv);
            switch(res)
            {
                case -1:
                {
                     printf("Select error\n");
                     return 1;
                  }
                  break;
                  case 0: /* Timeout */
                  {
                     printf("Time out\n");
                     return 1;
               }
                break;
                default:
                {
                     for (i = 0; i < IN_FILES; i++)
                     {
                        if (FD_ISSET(fds[i], &tmp_inset))
                        {
                           memset(buf, 0, MAX_BUFFER_SIZE);
                           real_read = read(fds[i], buf, MAX_BUFFER_SIZE);
                           if (real_read < 0)
                           {
                              if (errno != EAGAIN)
                              {
```

```
                        return 1;
                    }
                  }
                 else if (!real_read)
                 {
                    close(fds[i]);
                    FD_CLR(fds[i], &inset);
                }
                else
                {
                    if (i == 0)
                    {/* 主程序终端控制 */
                       if ((buf[0] == 'q') || (buf[0] == 'Q'))
                        {
                              return 1;
                    }
                    }
                    else
                    {/* 显示管道输入字符串 */
                        buf[real_read] = '\0';
                        printf("%s", buf);
                    }
                 }
              } /* end of if */
           } /* end of for */
        }
        break;
     } /* end of switch */
  } /*end of while */
  return 0;
}
```

（3）编译并运行该程序。

（4）另外打开两个虚拟终端，分别键入“cat > in1”和“cat > in2”，接着在该管道中键入相关内容，并观察实验结果。

4. 实验结果

实验运行结果与第 6 章的例子完全相同。

```
$ ./pipe_select （必须先运行主程序）
SELECT CALL
select call
```

```
TEST PROGRAMME
test programme
END
end
q /* 在终端上输入'q'或'Q'立刻结束程序运行 */

$ cat > in1
SELECT CALL
TEST PROGRAMME
END

$ cat > in2
select call
test programme
end
```

8.7.2 共享内存实验

1. 实验目的

通过编写共享内存实验，读者可以进一步了解使用共享内存的具体步骤，同时也进一步加深对共享内存的理解。在本实验中，采用信号量作为同步机制完善两个进程（“生产者”和“消费者”）之间的通信。其功能类似于“消息队列”中的实例，详见 8.5.2 小节。在实例中使用的与信号量相关的函数，详见 8.3.3 小节。

2. 实验内容

该实现要求利用共享内存实现文件的打开和读写操作。

3. 实验步骤

（1）画出流程图。

该实验流程图如图 8.10 所示。

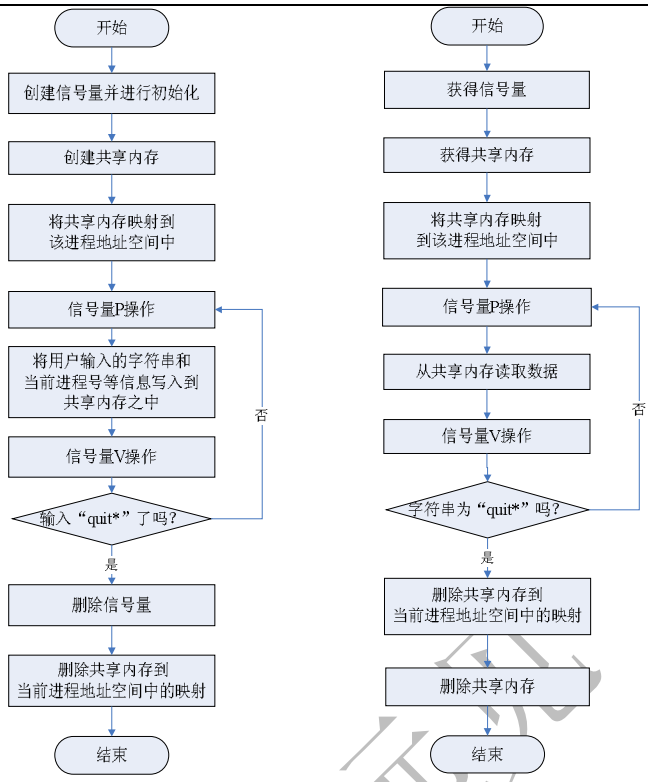

图 8.10　实验 8.6.2 流程图

（2）编写代码。

下面是共享内存缓冲区的数据结构的定义。

```
/* shm_com.h */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHM_BUFF_SZ 2048
struct shm_buff
{
    int pid;
    char buffer[SHM_BUFF_SZ];
};
```

以下是“生产者”程序部分。

```
/* sem_com.h 和 sem_com.c 与“信号量”小节示例中的同名程序相同 */
/* producer.c */
```

```
    #include "shm_com.h"
    #include "sem_com.h"
    #include <signal.h>
    int ignore_signal(void)
    { /* 忽略一些信号，免得非法退出程序 */
        signal(SIGINT, SIG_IGN);
        signal(SIGSTOP, SIG_IGN);
        signal(SIGQUIT, SIG_IGN);
        return 0;
    }

    int main()
    {
        void *shared_memory = NULL;
        struct shm_buff *shm_buff_inst;
        char buffer[BUFSIZ];
        int shmid, semid;
        /* 定义信号量，用于实现访问共享内存的进程之间的互斥*/
          ignore_signal(); /* 防止程序非正常退出 */
        semid = semget(ftok(".", 'a'), 1, 0666|IPC_CREAT); /* 创建一个信号量
*/
        init_sem(semid);/* 初始值为 1 */

        /* 创建共享内存 */
        shmid   =   shmget(ftok(".",   'b'),   sizeof(struct   shm_buff),
0666|IPC_CREAT);
        if (shmid == -1)
        {
            perror("shmget failed");
            del_sem(semid);
            exit(1);
        }

        /* 将共享内存地址映射到当前进程地址空间 */
        shared_memory = shmat(shmid, (void*)0, 0);
        if (shared_memory == (void*)-1)
        {
            perror("shmat");
            del_sem(semid);
            exit(1);
```

```
        }
        printf("Memory attached at %X\n", (int)shared_memory);
        /* 获得共享内存的映射地址 */
        shm_buff_inst = (struct shared_use_st *)shared_memory;
        do
        {
            sem_p(semid);
            printf("Enter some text to the shared memory(enter 'quit' to
exit):");
            /* 向共享内存写入数据 */
                if (fgets(shm_buff_inst->buffer, SHM_BUFF_SZ, stdin) ==
NULL)
            {
                perror("fgets");
                sem_v(semid);
                break;
            }
            shm_buff_inst->pid = getpid();
            sem_v(semid);
        } while(strncmp(shm_buff_inst->buffer, "quit", 4) != 0);

        /* 删除信号量 */
        del_sem(semid);
        /* 删除共享内存到当前进程地址空间中的映射 */
        if (shmdt(shared_memory) == 1)
        {
            perror("shmdt");
            exit(1);
        }
        exit(0);
    }
```

以下是“消费者”程序部分。

```
    /* customer.c */
    #include "shm_com.h"
    #include "sem_com.h"

    int main()
    {
        void *shared_memory = NULL;
```

```
        struct shm_buff *shm_buff_inst;
        int shmid, semid;
        /* 获得信号量 */
        semid = semget(ftok(".", 'a'),  1, 0666);
        if (semid == -1)
        {
            perror("Producer is'nt exist");
            exit(1);
        }
        /* 获得共享内存 */
        shmid   =   shmget(ftok(".",   'b'),   sizeof(struct   shm_buff),
0666|IPC_CREAT);
        if (shmid == -1)
        {
            perror("shmget");
            exit(1);
        }
        /* 将共享内存地址映射到当前进程地址空间 */
        shared_memory = shmat(shmid, (void*)0, 0);
        if (shared_memory == (void*)-1)
        {
            perror("shmat");
            exit(1);
        }
        printf("Memory attached at %X\n", (int)shared_memory);
        /* 获得共享内存的映射地址 */
        shm_buff_inst = (struct shm_buff *)shared_memory;
        do
        {
            sem_p(semid);
            printf("Shared memory was written by process %d :%s"
    ,                   shm_buff_inst->pid, shm_buff_inst->buffer);
            if (strncmp(shm_buff_inst->buffer, "quit", 4) == 0)
            {
               break;
            }
            shm_buff_inst->pid = 0;
            memset(shm_buff_inst->buffer, 0, SHM_BUFF_SZ);
            sem_v(semid);
        } while(1);
```

```
    /* 删除共享内存到当前进程地址空间中的映射 */
    if (shmdt(shared_memory) == -1)
    {
        perror("shmdt");
        exit(1);
    }
    /* 删除共享内存 */
    if (shmctl(shmid, IPC_RMID, NULL) == -1)
    {
        perror("shmctl(IPC_RMID)");
        exit(1);
    }
    exit(0);
}
```

4. 实验结果

```
$./producer
Memory attached at B7F90000
Enter some text to the shared memory(enter 'quit' to exit):First message
Enter some text to the shared memory(enter 'quit' to exit):Second message
Enter some text to the shared memory(enter 'quit' to exit):quit
$./customer
Memory attached at B7FAF000
Shared memory was written by process 3815 :First message
Shared memory was written by process 3815 :Second message
Shared memory was written by process 3815 :quit
```

8.8 本章小结

本章详细讲解了 Linux 中进程间通信的几种机制，包括管道通信、信号通信、消息队列、信号量以及共享内存机制等，并且讲解了进程间通信的演进。

接下来对管道通信、信号通信、消息队列和共享内存机制进行了详细讲解。其中，管道通信又分为有名管道和无名管道。信号通信中要着重掌握如何对信号进行适当的处理，如采用信号集等方式。信号量是用于实现进程之间的同步和互斥的进程间通信机制。

消息队列和共享内存也是很好的进程间通信的手段，其中共享内存具有很高的效率，并经常以信号量作为同步机制。

本章的最后安排了管道通信实验和共享内存的实验，具体的实验数据根据系统的不同可能会有所区别，希望读者认真分析。

8.9 思考与练习

1. 通过自定义信号完成进程间的通信。
2. 编写一个简单的管道程序实现文件传输。