第14章　进程间通信

14.1　引言

第8章说明了进程控制原语并且观察了如何调用多个进程。但是这些进程之间交换信息的唯一方法是经由fork或exec传送打开文件，或通过文件系统。本章将说明进程之间相互通信的其他技术——IPC（InterProcess Communication）。

UNIX IPC已经是而且继续是各种进程通信方式的统称，其中极少能在所有UNIX的实现中进行移植。表14-1列出了不同实现所支持的不同形式的IPC。

表14-1　UNIX IPC

IPC类型	POSIX.1	XPG3	V7	SVR2	SVR3.2	SVR4	4.3BSD	4.3+BSD
管道(半双工)	•	•	•	•	•	•	•	•
FIFOs(命令管道)	•	•		•	•	•		•
流管道(全双工)					•	•	•	•
命令流管道					•	•	•	•
消息队列		•		•	•	•		
信号量		•		•	•	•		
共享存储		•		•	•	•		
套接口						•	•	•
流					•	•		

正如上表所示，不管哪一种UNIX实现，都可依靠的唯一一种IPC是半双工的管道。表中前7种IPC通常限于同一台主机的各个进程间的IPC。最后两种；套接口和流，则支持不同主机上各个进程间IPC（关于网络IPC的详细情况，请参见Stevens〔1990〕）。虽然中间三种形式的IPC（消息队列、信号量以及共享存储器）在表中说明为只受到系统V的支持，但是在大多数制造商所支持的，从伯克利UNIX导出的UNIX系统中（例如，SunOS以及Ultrix），已经添加了这三种形式的IPC。

> 几个POSIX小组正在对IPC进行工作，但是最后结果还不很清楚，可能要到1994年甚至更迟一点与IPC有关的POSIX才能制定出来。

我们将与IPC有关的讨论分成两章。本章将讨论经典的IPC；管道、FIFO、消息队列、信号量以及共享存储器。下一章将观察SVR4和4.3+BSD共同支持的IPC的某些高级特征，包括；流管道和命名流管道，以及用这些更高级形式的IPC可以做的一些事情。

14.2　管道

管道是UNIX IPC的最老形式，并且所有UNIX系统都提供此种通信机制，管道有两种限制；

(1) 它们是半双工的。数据只能在一个方向上流动。

(2) 它们只能在具有公共祖先的进程之间使用。通常，一个管道由一个进程创建，然后该

进程调用fork，此后父、子进程之间就可应用该管道。

我们将会看到流管道（见15.2节）没有第一种限制，FIFO（见14.5节）和命名流管道（见15.5节）则没有第二种限制。尽管有这两种限制，半双工管道仍是最常用的IPC形式。

管道是由调用pipe函数而创建的。

```
#include  <unistd.h>
int pipe(int filedes[2]) ;
```

返回：若成功则为0，若出错则为-1

经由参数*filedes*返回两个文件描述符：*filedes[0]*为读而打开，*filedes[1]*为写而打开。*filedes[1]*的输出是*filedes[0]*的输入。

有两种方法来描绘一个管道，见图14-1。左半图显示了管道的两端在一个进程中相互连接，右半图则说明数据通过内核在管道中流动。

> 在SVR4下，管道是双全工的。两个描述符都可用于读、写。于是，图14-1中的箭头在两端都有。我们称这种全双工管道为“流管道”，下一章将详细讨论这种管道。因为POSIX.1只提供半双工管道，为了可移植性，我们假定pipe函数创建一个单方向的管道。

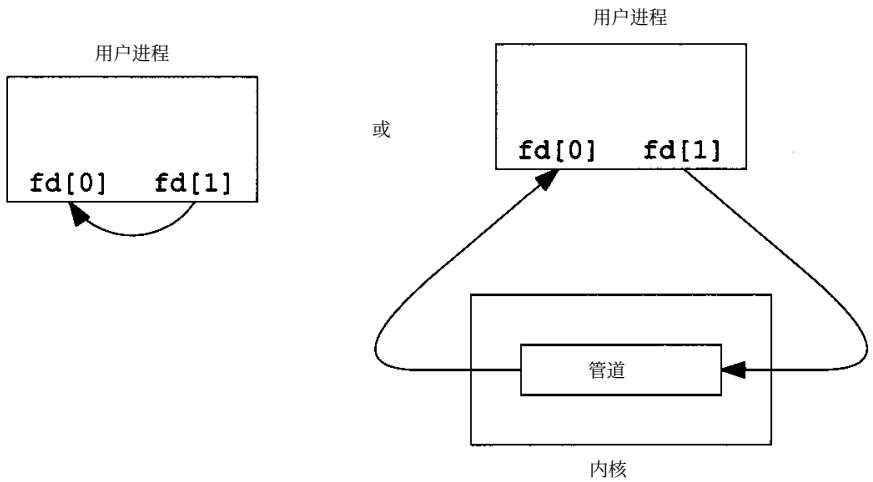

图14-1 观察UNIX管道的两种方法

fstat函数（见4.2节）对管道的每一端都返回一个FIFO类型的文件描述符，可以用S_ISFIFO宏来测试管道。

> POSIX.1规定stat结构的st_size成员对于管道是未定义的。但是当fstat函数应用于管道读端的文件描述符时，很多系统在st_size中存放管道中可用于读的字节数。但是，这是不可移植的。

单个进程中的管道几乎没有任何用处。通常，调用pipe的进程接着调用fork，这样就创建了从父进程到子进程或反之的IPC通道。图14-2显示了这种情况。

fork之后做什么取决于我们想要有的数据流的方向。对于从父进程到子进程的管道，父进程关闭管道的读端（fd[0]），子进程则关闭写端（fd[1]）。图14-3显示了描述符的最后安排。

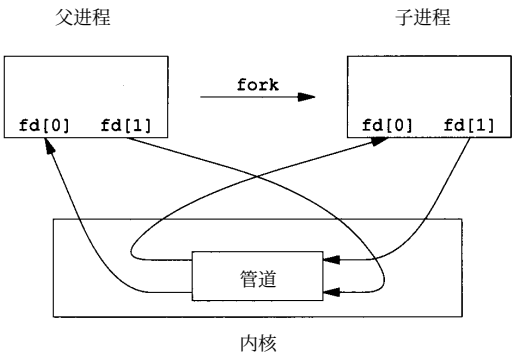

图14-2 fork之后的半双工管道

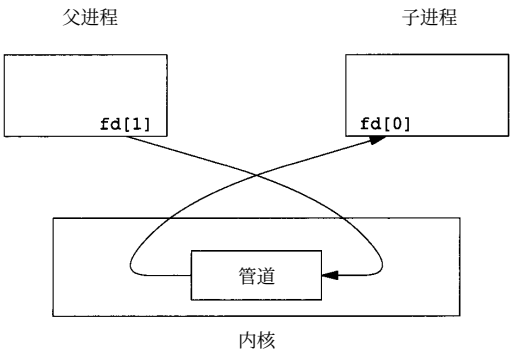

图14-3 从父进程到子进程的管道

对于从子进程到父进程的管道，父进程关闭 fd[1]，子进程关闭fd[0]。

当管道的一端被关闭后，下列规则起作用：

(1) 当读一个写端已被关闭的管道时，在所有数据都被读取后， read返回0，以指示达到了文件结束处（从技术方面考虑，管道的写端还有进程时，就不会产生文件的结束。可以复制一个管道的描述符，使得有多个进程具有写打开文件描述符。但是，通常一个管道只有一个读进程，一个写进程。下一节介绍FIFO时，我们会看到对于一个单一的FIFO常常有多个写进程）。

(2) 如果写一个读端已被关闭的管道，则产生信号 SIGPIPE。如果忽略该信号或者捕捉该信号并从其处理程序返回，则write出错返回， errno设置为EPIPE。

在写管道时，常数PIPE_BUF规定了内核中管道缓存器的大小。如果对管道进行 write调用，而且要求写的字节数小于等于 PIPE_BUF，则此操作不会与其他进程对同一管道（或 FIFO）的write操作穿插进行。但是，若有多个进程同时写一个管道（或 FIFO），而且某个或某些进程要求写的字节数超过PIPE_BUF字节数，则数据可能会与其他写操作的数据相穿插。

实例

程序14-1创建了一个从父进程到子进程的管道，并且父进程经由该管道向子进程传送数据。

程序14-1 经由管道父进程向子进程传送数据

```
#include    "ourhdr.h"

int
main(void)
{
    int     n, fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");

    else if (pid > 0) {     /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);

    } else {                /* child */
```

```
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }

    exit(0);
}
```

在上面的例子中，直接对管道描述符调用read和write。更为有益的是将管道描述符复制为标准输入和标准输出。在此之后通常子进程调用 exec，执行另一个程序，该程序从标准输入（已创建的管道）或将数据写至其标准输出（管道）。

实例

试编写一个程序，其功能是每次一页显示已产生的输出。已经有很多 UNIX公用程序具有分页功能，因此无需再构造一个新的分页程序，而是调用用户最喜爱的分页程序。为了避免先将所有数据写到一个临时文件中，然后再调用系统中的有关程序显示该文件，我们希望将输出通过管道直接送到分页程序。为此，先创建一个管道，一个子进程，使子进程的标准输入成为管道的读端，然后exec用户喜爱的分页程序。程序 14-2显示了如何实现这些操作。（本例要求在命令行中有一个参数说明要显示的文件的名称。通常，这种类型的程序要求在终端上显示的数据已经在存储器中。）

程序14-2 将文件复制到分页程序

```
#include    <sys/wait.h>
#include    "ourhdr.h"

#define DEF_PAGER   "/usr/bin/more"     /* default pager program */

int
main(int argc, char *argv[])

{
    int     n, fd[2];
    pid_t   pid;
    char    line[MAXLINE], *pager, *argv0;
    FILE    *fp;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if ( (fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);

    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid > 0) {                                 /* parent */
        close(fd[0]);       /* close read end */
            /* parent copies argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
            n = strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
            err_sys("fgets error");

       close(fd[1]);    /* close write end of pipe for reader */
```

```
        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
        exit(0);
    } else {                                        /* child */
        close(fd[1]);   /* close write end */
        if (fd[0] != STDIN_FILENO) {
            if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd[0]);   /* don't need this after dup2 */
        }

            /* get arguments for execl() */
        if ( (pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;
        if ( (argv0 = strrchr(pager, '/')) != NULL)
            argv0++;        /* step past rightmost slash */
        else
            argv0 = pager;  /* no slash in pager */

        if (execl(pager, argv0, (char *) 0) < 0)
            err_sys("execl error for %s", pager);
    }
}
```

在调用fork之前先创建一个管道。fork之后父进程关闭其读端，子进程关闭其写端。子进程然后调用dup2，使其标准输入成为管道的读端。当执行分页程序时，其标准输入将是管道的读端。

当我们将一个描述符复制到另一个时（在子进程中，fd[0]复制到标准输入），应当注意该描述符的值并不已经是所希望的值。如果该描述符已经具有所希望的值，并且我们先调用dup2，然后调用close则将关闭此进程中只有该单个描述符所代表的打开文件。（回忆3.12节中所述，当dup2中的两个参数值相等时的操作。）在本程序中，如果shell没有打开标准输入，那么程序开始处的fopen应已使用描述符0，也就是最小未使用的描述符，所以fd[0]决不会等于标准输入。尽管如此，只要先调用dup2，然后调用close以复制一个描述符到另一个，作为一种保护性的编程措施，我们总是先将两个描述符进行比较。

请注意，我们是如何使用环境变量 PAGER获得用户分页程序名称的。如果这种操作没有成功，则使用系统默认值。这是环境变量的常见用法。

实例

回忆 8.8节中的五个函数： TELL_WAIT、TELL_PARENT、TELL_CHILD、WAIT_PARENT以及WAIT_CHILD。程序10-17提供了一个使用信号的实现。程序 14-3则是一个使用管道的实现。

见图14-4，在fork之前创建了两个管道。

父进程在调用 TELL_CHILD时经由上一个管道写一个字符“ P”，子进程在调用TELL_PARENT时，经由下一个管道写一个字符“ C”。相应的WAIT_XXX函数调用read读一个字符，没有读到字符时阻塞（睡眠等待）。

请注意，每一个管道都有一个额外的读取进程，这没有关系。也就是说除了子进程从 pfd1[0]读取，父进程也有上一个管道的读端。因为父进程并没有执行对该管道的读操作，所以这不会产生任何影响。

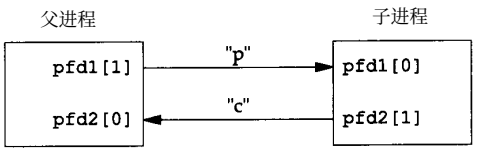

图14-4 用两个管道实现父-子进程的同步

程序14-3 使父、子进程同步的例程

```
#include    "ourhdr.h"

static int  pfd1[2], pfd2[2];

void
TELL_WAIT(void)
{
    if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        err_sys("pipe error");
}

void
TELL_PARENT(pid_t pid)
{
    if (write(pfd2[1], "c", 1) != 1)
        err_sys("write error");
}

void
WAIT_PARENT(void)
{
    char    c;

    if (read(pfd1[0], &c, 1) != 1)
        err_sys("read error");
    if (c != 'p')
        err_quit("WAIT_PARENT: incorrect data");
}

void
TELL_CHILD(pid_t pid)
{
    if (write(pfd1[1], "p", 1) != 1)
        err_sys("write error");
}

void
WAIT_CHILD(void)
{
    char    c;

    if (read(pfd2[0], &c, 1) != 1)
        err_sys("read error");
    if (c != 'c')
        err_quit("WAIT_CHILD: incorrect data");
}
```

14.3 popen和pclose函数

因为常见的操作是创建一个连接到另一个进程的管道，然后读其输出或向其发送输入，所以标准I/O库为实现这些操作提供了两个函数 popen和pclose。这两个函数实现的操作是：创建一个管道，fork一个子进程，关闭管道的不使用端，exec一个shell以执行命令，等待命令终止。

```
#include  <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
```

返回：若成功则为文件指针，若出错则为NULL

```
int pclose(FILE *fp);
```

返回：*cmdstring*的终止状态，若出错则为-1

函数popen 先执行fork，然后调用exec以执行*cmdstring*，并且返回一个标准I/O文件指针。如果*type*是"r"，则文件指针连接到*cmdstring*的标准输出（见图14-5）。

如果*type* 是"w",则文件指针连接到*cmdstring* 的标准输入（见图14-6）。

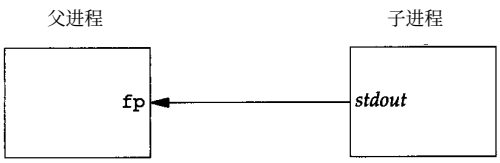

图14-5 fp=popen(*command* , "r")的结果

图14-6 fp=popen(*command*,"w")的结果

有一种方法可以帮助我们记住 popen最后一个参数及其作用，这种方法就是与 fopen进行类比。如果*type*是"r",则返回的文件指针是可读的，如果*type*是"w"，则是可写的。

pclose函数关闭标准I/O流，等待命令执行结束，然后返回 shell的终止状态。（我们曾在8.6节对终止状态进行过说明， system函数（见8.12节）也返回终止状态。）如果shell不能被执行，则pclose返回的终止状态与shell执行exit（127）一样。

cmdstring 由Bourne shell以下列方式执行；

```
sh -c cmdstring
```

这表示shell将扩展*cmdstring*中的任何特殊字符。例如，可以使用；

```
fp = popen("ls *.c" , "r");
```

或者

```
fp = popen("cmd 2>&1" , "r");
```

> POSIX.1没有说明popen、pclose,因为它们与shell有交互作用，而 shell 是由POSIX.2说明的。我们对这两个函数的说明与 POSIX.2的11.2草案相一致。该草案对这两个函数的说明与以前的实现有些区别。

实例

用popen重写程序14-2，其结果是程序14-4。使用popen减少了需要编写的代码量。

shell命令${PAGER:-more}的意思是：如果shell变量PAGER已经定义，且其值非空，则使用其值，否则使用字符串more。

程序14-4 用popen向分页程序传送文件

```
#include    <sys/wait.h>
#include    "ourhdr.h"

#define PAGER   "${PAGER:-more}" /* environment variable, or default */

int
main(int argc, char *argv[])
{
    char    line[MAXLINE];
    FILE    *fpin, *fpout;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");
```

```
    if ( (fpin = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);

    if ( (fpout = popen(PAGER, "w")) == NULL)
        err_sys("popen error");

        /* copy argv[1] to pager */
    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (ferror(fpin))
        err_sys("fgets error");
    if (pclose(fpout) == -1)
        err_sys("pclose error");
    exit(0);
}
```

实例——popen函数

程序14-5是我们编写的popen和pclose版本。虽然popen的核心部分与本章中以前用过的代码类似，但是增加了很多需要考虑的细节。首先每次调用 popen时，应当记住所创建的子进程的进程ID，以及其文件描述符或FILE指针。我们选择在数组childpid中保存子进程ID，并用文件描述符作为其下标。于是，当以 FILE指针作为参数调用 pclose时，我们调用标准 I/O函数fileno以得到文件描述符，然后取得子进程 ID，并用于调用 waitpid。因为一个进程可能调用popen多次，所以在动态分配childpid数组时（第一次调用popen时），其长度可以容纳与文件描述符数相同的进程数。

调用pipe、fork以及为每个进程复制相应的文件描述符，这些操作与本章前面所述的类似。

POSIX.2要求子进程关闭在以前调用 popen时形成，当前仍旧打开的所有 I/O流。为此，在子进程中从头逐个检查childpid数组的各元素，关闭仍旧打开的任一描述符。

若pclose的调用者已经为信号 SIGCHLD设置了一个信号处理程序，则 waitpid将返回一个EINTR。因为允许调用者捕捉此信号（或者任何其他可能中断 waitpid调用的信号），所以当waitpid被一个捕捉到的信号中断时，我们只是再次调用 waitpid。

> 如果一个信号中断了wait， pclose的早期版本返回EINTR。
>
> pclose的早期版本在 wait期间，阻塞或忽略信号 SIGINT、SIGQUIT以及SIGHUP。POSIX.2则不允许这一点。

程序14-5 popen和pclose函数

```
#include    <sys/wait.h>
#include    <errno.h>
#include    <fcntl.h>
#include    "ourhdr.h"

static pid_t    *childpid = NULL;
                        /* ptr to array allocated at run-time */
static int      maxfd;  /* from our open_max(), Program 2.3 */

#define SHELL   "/bin/sh"

FILE *
popen(const char *cmdstring, const char *type)
```

```
{
    int     i, pfd[2];
    pid_t   pid;
    FILE    *fp;

            /* only allow "r" or "w" */
    if ((type[0] != 'r' && type[0] != 'w') || type[1] != 0) {
        errno = EINVAL;     /* required by POSIX.2 */
        return(NULL);
    }

    if (childpid == NULL) {     /* first time through */
                /* allocate zeroed out array for child pids */
        maxfd = open_max();
        if ( (childpid = calloc(maxfd, sizeof(pid_t))) == NULL)
            return(NULL);
    }

    if (pipe(pfd) < 0)
        return(NULL);   /* errno set by pipe() */

    if ( (pid = fork()) < 0)
        return(NULL);   /* errno set by fork() */
    else if (pid == 0) {                            /* child */
        if (*type == 'r') {
            close(pfd[0]);
            if (pfd[1] != STDOUT_FILENO) {
                dup2(pfd[1], STDOUT_FILENO);
                close(pfd[1]);
            }
        } else {
            close(pfd[1]);
            if (pfd[0] != STDIN_FILENO) {
                dup2(pfd[0], STDIN_FILENO);
                close(pfd[0]);
            }
        }
            /* close all descriptors in childpid[] */
        for (i = 0; i < maxfd; i++)
            if (childpid[i] > 0)
                close(i);

        execl(SHELL, "sh", "-c", cmdstring, (char *) 0);
        _exit(127);
    }
                                /* parent */
    if (*type == 'r') {
        close(pfd[1]);
        if ( (fp = fdopen(pfd[0], type)) == NULL)
            return(NULL);
    } else {
         close(pfd[0]);
         if ( (fp = fdopen(pfd[1], type)) == NULL)
             return(NULL);
     }
     childpid[fileno(fp)] = pid; /* remember child pid for this fd */
     return(fp);
}

int
pclose(FILE *fp)
{
    int     fd, stat;
    pid_t   pid;
```

```
    if (childpid == NULL)
        return(-1);     /* popen() has never been called */

    fd = fileno(fp);
    if ( (pid = childpid[fd]) == 0)
        return(-1);     /* fp wasn't opened by popen() */

    childpid[fd] = 0;
    if (fclose(fp) == EOF)
        return(-1);

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* error other than EINTR from waitpid() */

    return(stat);   /* return child's termination status */
}
```

实例

考虑一个应用程序，它向标准输出写一个提示，然后从标准输入读1行。使用popen,可以在应用程序和输入之间插入一个程序以对输入进行变换处理。图14-7显示了进程的安排。

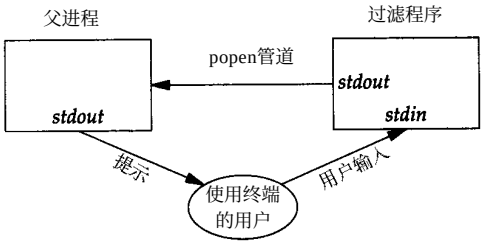

图14-7 用popen变换输入

对输入进行的变换可能是路径名的扩充，或者是提供一种历史机制（记住以前输入的命令）。(本实例取自POSIX.2草案。)

程序14-6是一个简单的过滤程序，它只是将输入复制到输出，在复制时将任一大写字符变换为小写字符。在写了一行之后，对标准输出进行了刷清（用fflush），其理由将在下一节介绍协同进程时讨论。

程序14-6 过滤程序，将大写字符变换成小写字符

```
#include    <ctype.h>
#include    "ourhdr.h"

int
main(void)
{
    int     c;

    while ( (c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("output error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

对该过滤程序进行编译，其可执行目标代码存放在文件 myuclc中，然后在程序 14-7中用popen调用它们。

因为标准输出通常是按行进行缓存的，而提示并不包含新行符，所以在写了提示之后，需要调用fflush。

程序14-7 调用大写/小写过滤程序以读取命令

```
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
{
    char    line[MAXLINE];
    FILE    *fpin;

    if ( (fpin = popen("myuclc", "r")) == NULL)
        err_sys("popen error");

    for ( ; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (pclose(fpin) == -1)
        err_sys("pclose error");
    putchar('\n');
    exit(0);
}
```

14.4 协同进程

UNIX过滤程序从标准输入读取数据，对其进行适当处理后写到标准输出。几个过滤进程通常在shell管道命令中线性地连接。当同一个程序产生某个过滤程序的输入，同时又读取该过滤程序的输出时，则该过滤程序就成为协同进程(coprocess)。

KornShell提供了协同进程。Bourne shell和C shell并没有提供将进程连接起来按协同进程方式工作的方法。协同进程通常在shell的后台运行，其标准输入和标准输出通过管道连接到另一个程序。虽然要求初始化一个协同进程，并将其输入和输出连接到另一个进程的shell语法是十分奇特的（详细情况见Bolsky和Korn〔1989〕中的pp.66～66）,但是协同进程的工作方式在C程序中也是非常有用的。

popen提供连接到另一个进程的标准输入或标准输出的一个单行管道，而对于协同进程，则它有连接到另一个进程的两个单行管道 —— 一个接到其标准输入，另一个则来自标准输出。我们先要将数据写到其标准输入，经过其处理后，再从其标准输出读取数据。

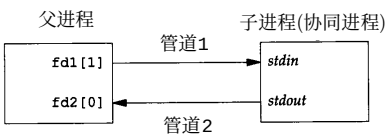

图14-8 驱动一个协同进程 —— 写其标准输入，读其标准输出

实例

让我们通过一个实例来观察协同进程。进程先创建两个管道：一个是协同进程的标准输入，另一个是协同进程的标准输出。图14-8显示了这种安排。

程序14-8是一个简单的协同进程，它从其标准输入读两个数，计算它们的和，然后将结果写至标准输出。

程序14-8 对两个数求和的简单过滤程序

```
#include    "ourhdr.h"

int
```

```
main(void)
{
    int     n, int1, int2;
    char    line[MAXLINE];

    while ( (n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
        line[n] = 0;        /* null terminate */
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            sprintf(line, "%d\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                err_sys("write error");
        }
    }
    exit(0);
}
```

对此程序进行编译，将其可执行目标代码存入名为add2的文件。

程序14-9在从其标准输入读入两个数之后调用add2协同进程。从协同进程送来的值则写到其标准输出。

程序14-9 驱动add2过滤程序的程序

```
#include    <signal.h>
#include    "ourhdr.h"

static void sig_pipe(int);      /* our signal handler */

int
main(void)
{
    int     n, fd1[2], fd2[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
        err_sys("signal error");

    if (pipe(fd1) < 0 || pipe(fd2) < 0)
        err_sys("pipe error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");

 else if (pid > 0) {                        /* parent */
     close(fd1[0]);
     close(fd2[1]);
     while (fgets(line, MAXLINE, stdin) != NULL) {
         n = strlen(line);
         if (write(fd1[1], line, n) != n)
             err_sys("write error to pipe");
         if ( (n = read(fd2[0], line, MAXLINE)) < 0)
             err_sys("read error from pipe");
         if (n == 0) {
             err_msg("child closed pipe");
             break;
         }
         line[n] = 0;    /* null terminate */
         if (fputs(line, stdout) == EOF)
             err_sys("fputs error");
```

```
        }
        if (ferror(stdin))
            err_sys("fgets error on stdin");
        exit(0);

    } else {                                        /* child */
        close(fd1[1]);
        close(fd2[0]);
        if (fd1[0] != STDIN_FILENO) {
            if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd1[0]);
        }
        if (fd2[1] != STDOUT_FILENO) {
            if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
                err_sys("dup2 error to stdout");
            close(fd2[1]);
        }
        if (execl("./add2", "add2", (char *) 0) < 0)
            err_sys("execl error");
    }
}

static void
sig_pipe(int signo)
{
    printf("SIGPIPE caught\n");
    exit(1);
}
```

在程序中创建了两个管道，父、子进程各自关闭它们不需使用的端口。创建两个管道的理由是：一个用做协同进程的标准输入，另一个则用做它的标准输出。然后在调用 execl之前，子进程调用dup2使管道描述符移至其标准输入和输出。

若编译和运行程序14-9，它如所希望的那样进行工作。并且，当程序 14-9正等待输入时，若杀死 add2协同进程，然后输入两个数，当程序14-9对管道进行写操作时，由于该管道无读进程，于是调用信号处理程序（见习题14.4）。

程序15-1将提供这一实例的另一个版本，它使用一个全双工管道而不是两个半双工管道。

实例

在协同进程add2（见程序14-8）中，使用了UNIX I/O:read和write。如果使用标准I/O改写该协同进程，其后果是什么呢？程序14-10即改写后的版本。

程序14-10 对两个数求和的滤波程序，使用标准 I/O

```
#include    "ourhdr.h"

int
main(void)
{
    int     int1, int2;
    char    line[MAXLINE];

    while (fgets(line, MAXLINE, stdin) != NULL) {
        if (sscanf(line, "%d%d", &int1, &int2) == 2) {
            if (printf("%d\n", int1 + int2) == EOF)
                err_sys("printf error");
        } else {
            if (printf("invalid args\n") == EOF)
```

```
                err_sys("printf error");
            }
        }
        exit(0);
    }
```

若程序14-9调用此新的协同进程，则它不再工作。问题出在系统默认的标准 I/O缓存机制上。当程序14-10被调用时，对标准输入的第一个fgets引起标准I/O库分配一个缓存，并选择缓存的类型。因为标准输入是个管道，所以 isatty为假，于是标准I/O库由系统默认是全缓存的。对标准输出也有同样的处理。当add2从其标准输入读取而发生堵塞时，程序14-9从管道读时也发生堵塞，于是产生了死锁。

对将要执行的这样一个协同进程可以加以控制。在程序14-10中的while循环之前加上下面4行：

```
if (setvbuf(stdin, NULL, _IOLBF, 0) != 0)
    err_sys("setvbuf error");
if (setvbuf(stdout, NULL, _IOLBF, 0)!= 0)
    err_sys("setvbuf error");
```

这使得当有一行可用时，fgets即返回，并使得当输出一新行符时，printf即执行fflush操作。对setvbuf进行了这些显式调用，使得程序14-10能正常工作。

如果不能修改程序，则需使用其他技术。例如，如果在程序中使用 awk(1)代替add2作为协同进程，则下列命令行不能工作；

```
#!/bin/awk/ -f
{ print $1 + $2 }
```

不能工作的原因还是标准 I/O的缓存机制问题。但是，在这种情况下不能改变 awk的工作方式（除非有awk的源代码）。

对这种问题的一般解决方法是使被调用（在本例中是 awk）的协同进程认为它的标准输入和输出被连接到一个终端。这使得协同进程中的标准 I/O例程对这两个I/O流进行行缓存，这类似于前面所做的显式setvbuf调用。第19章将用伪终端实现这一点。

14.5 FIFO

FIFO有时被称为命名管道。管道只能由相关进程使用，它们共同的祖先进程创建了管道。但是，通过FIFO，不相关的进程也能交换数据。

第14章已经提及FIFO是一种文件类型。stat结构（见4.2节）成员st_mode的编码指明文件是否是FIFO类型。可以用S_ISFIFO宏对此进行测试。

创建FIFO类似于创建文件。确实，FIFO的路径名存在于文件系统中。

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

int mkfifo(const char *pathname, mode_t mode);
```

返回：若成功则为0，若出错则为-1

mkfifo函数中*mode*参数的规格说明与open函数中的*mode*相同（见3.3节）。新FIFO的用户和组的所有权与4.6节所述的相同。

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

mkfifo是POSIX.1首先提出的。SVR3用mknod（2）系统调用创建FIFO。而在SVR4中，mkfifo调用mknod创建FIFO。

POSIX.2已经建议了一个mkfifo(1)命令。SVR4和4.3+BSD现在支持此命令。于是，用一条shell命令就可以创建一个FIFO，然后用一般的shell I/O重新定向对其进行存取。

当打开一个FIFO时，非阻塞标志（O_NONBLOCK）产生下列影响：

(1) 在一般情况中（没有说明O_NONBLOCK），只读打开要阻塞到某个其他进程为写打开此FIFO。类似，为写而打开一个FIFO要阻塞到某个其他进程为读而打开它。

(2) 如果指定了O_NONBLOCK，则只读打开立即返回。但是，如果没有进程已经为读而打开一个FIFO，那么只写打开将出错返回，其errno是ENXIO。

类似于管道，若写一个尚无进程为读而打开的FIFO，则产生信号SIGPIPE。若某个FIFO的最后一个写进程关闭了该FIFO，则将为该FIFO的读进程产生一个文件结束标志。

一个给定的FIFO有多个写进程是常见的。这就意味着如果不希望多个进程所写的数据互相穿插，则需考虑原子写操作。正如对于管道一样，常数 PIPE_BUF说明了可被原子写到FIFO的最大数据量。

FIFO有两种用途：

(1) FIFO由shell命令使用以便将数据从一条管道线传送到另一条，为此无需创建中间临时文件。

(2) FIFO用于客户机-服务器应用程序中，以在客户机和服务器之间传递数据。

我们各用一个例子来说明这两种用途。

实例 —— 用FIFO复制输出流

FIFO可被用于复制串行管道命令之间的输出流，于是也就不需要写数据到中间磁盘文件中（类似于使用管道以避免中间磁盘文件）。管道只能用于进程间的线性连接，然而，因为FIFO具有名字，所以它可用于非线性连接。

考虑这样一个操作过程，它需要对一个经过过滤的输入流进行两次处理。图14-9表示了这种安排。

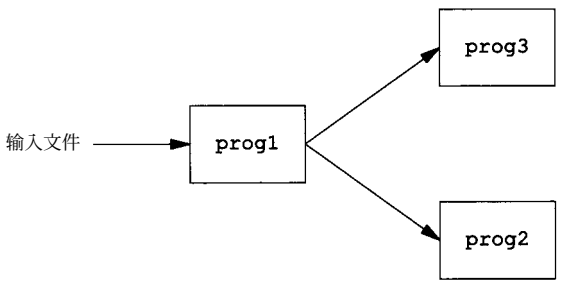

图14-9 对一个经过过滤的输入流进行两次处理

使用FIFO以及UNIX程序tee(1)，就可以实现这样的过程而无需使用临时文件。（tee程序将其标准输入同时复制到其标准输

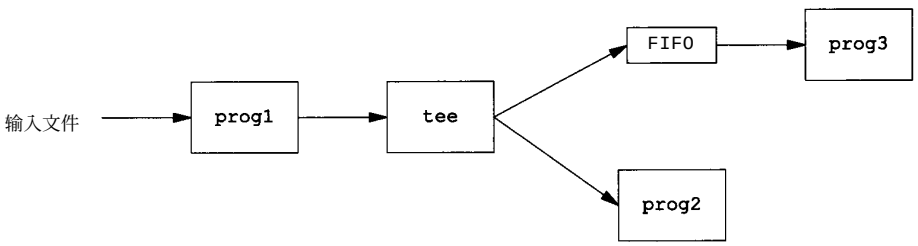

图14-10 使用FIFO和tee将一个流发送到两个不同的进程

出以及其命令行中包含的命名文件中。）

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

创建FIFO，然后在后台起动prog3，它从FIFO读数据。然后起动progl，用tee将其输出发送到FIFO和prog2。图14-10显示了有关安排。

实例 —— 客户 - 服务器使用FIFO进行通信

FIFO的另一个应用是在客户机和服务器之间传送数据。如果有一个服务器，它与很多客户机有关，每个客户机都可将其请求写到一个该服务器创建的众所周知的 FIFO中（“ 众所周知 ”的意思是；所有需与服务器联系的客户机都知道该 FIFO的路径名）。图14-11显示了这种安排。因为对于该FIFO有多个写进程，客户机发送给服务器的请求其长度要小于 PIPE_BUF字节。这样就能避免客户机各次写之间的穿插。

在这种类型的客户机-服务器通信中使用FIFO的问题是：服务器如何将回答送回各个客户机。不能使用单个FIFO，因为服务器会发出对各个客户机请求的响应，而请求者却不可能知道什么时候去读才能恰恰得到对它的响应。一种解决方法是每个客户机都在其请求中发送其进程ID。然后服务器为每个客户机创建一个 FIFO，所使用的路径名是以客户机的进程 ID为基础的。例如，服务器可以用名字/tmp/serv1.XXXXX创建FIFO，其中XXXXX被替换成客户机的进程ID。图14-12显示了这种安排。

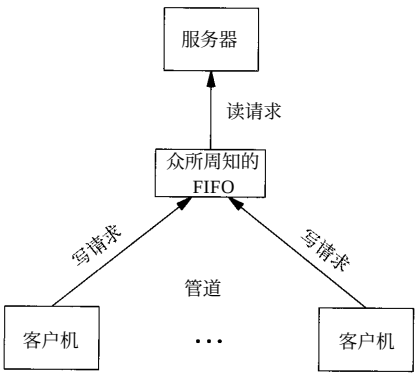

图14-11 客户机用FIFO向服务器发送请求

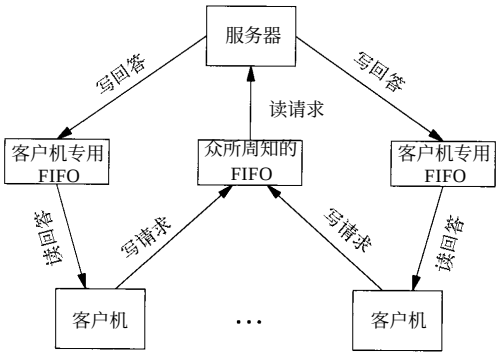

图14-12 客户机-服务器用FIFO进行通信

这种安排可以工作，但也有一些不足之处。其中之一是服务器不能判断一个客户机是否崩溃终止，这就使得客户机专用的FIFO会遗留在文件系统中。另一个是服务器必须捕捉 SIGPIPE信号，因为客户机在发送一个请求后没有读取响应就可能终止，于是留下一个有写进程（服务器）而无读进程的客户机专用FIFO。

按照图14-12中的安排，如果服务器以只读方式打开众所周知的 FIFO（因为它只需读该FIFO），则每次客户机数从1变成0，服务器就将在FIFO中读到一个文件结束标记。为使服务器免于处理这种情况，一种常见的技巧是使服务器以读-写方式打开该FIFO（见习题14.10）。

14.6 系统V IPC

三种系统V IPC：消息队列、信号量以及共享存储器之间有很多相似之处。以下各节将说明这些IPC的各自特殊功能，本节先介绍它们类似的特征。

这三种IPC源自于1970年的一种称为Columbus UNIX的UNIX内部版本。后来它们被加到SV上。

14.6.1 标识符和关键字

每个内核中的IPC结构（消息队列、信号量或共享存储段）都用一个非负整数的标识符(identifier)加以引用。例如，为了对一个消息队列发送或取消息，只需知道其队列标识符。与文件描述符不同，IPC标识符不是小的整数。当一个IPC结构被创建，以后又被删除时，与这种结构相关的标识符连续加1，直至达到一个整型数的最大正值，然后又回转到0。(即使在IPC结构被删除后也记住该值，每次使用此结构时则增 1，该值被称为“槽使用顺序号”。它在ipc_perm结构中，下一节将说明此结构。)

无论何时创建IPC结构（调用msgget、semget或shmget）,都应指定一个关键字（key），关键字的数据类型由系统规定为key_t，通常在头文件<sys/types.h>中被规定为长整型。关键字由内核变换成标识符。

有多种方法使客户机和服务器在同一IPC结构上会合：

(1) 服务器可以指定关键字IPC_PRIVATE创建一个新IPC结构，将返回的标识符存放在某处（例如一个文件）以便客户机取用。关键字 IPC_PRIVATE保证服务器创建一个新IPC结构。这种技术的缺点是：服务器要将整型标识符写到文件中，然后客户机在此后又要读文件取得此标识符。

IPC_PRIVATE关键字也可用于父、子关系进程。父进程指定IPC_PRIVATE创建一个新IPC结构，所返回的标识符在fork后可由子进程使用。子进程可将此标识符作为exec函数的一个参数传给一个新程序。

(2) 在一个公用头文件中定义一个客户机和服务器都认可的关键字。然后服务器指定此关键字创建一个新的IPC结构。这种方法的问题是该关键字可能已与一个 IPC结构相结合，在此情况下，get函数（msgget、semget或shmget）出错返回。服务器必须处理这一错误，删除已存在的IPC结构，然后试着再创建它。

(3) 客户机和服务器认同一个路径名和课题ID（课题ID是0~255之间的字符值），然后调用函数ftok将这两个值变换为一个关键字（函数ftok在手册页stdipc（3）中说明）。然后在方法(2)中使用此关键字。ftok提供的唯一服务就是由一个路径名和课题ID产生一个关键字。因为一般来说，客户机和服务器至少共享一个头文件，所以一个比较简单的方法是避免使用 ftok，而只是在该头文件中存放一个大家都知道的关键字。这样做还避免了使用另一个函数。

三个get函数（msgget、semget和shmget）都有两个类似的参数*key*和一个整型的*flag*。如若满足下列条件，则创建一个新的IPC结构（通常由服务器创建）：

(1) *key*是IPC_PRIVATE，或

(2) *key*当前未与特定类型的IPC结构相结合，*flag*中指定了IPC_CREAT位。为访问现存的队列(通常由客户机进行)，*key*必须等于创建该队列时所指定的关键字，并且不应指定IPC_CREAT。

注意，为了访问一个现存队列，决不能指定IPC_PRIVATE作为关键字。因为这是一个特殊的键值，它总是用于创建一个新队列。为了访问一个用 IPC_PRIVATE关键字创建的现存队列，一定要知道与该队列相结合的标识符，然后在其他IPC调用中（例如msgsnd、msgrcv）使用该标识符。

如果希望创建一个新的IPC结构，保证不是引用具有同一标识符的一个现行IPC结构，那么必须在*flag*中同时指定IPC_CREAT和IPC_EXCL位。这样做了以后，如果IPC结构已经存在就会造成出错，返回EEXIST（这与指定了O_CREAT和O_EXCL标志的open相类似）。

14.6.2 许可权结构

系统V IPC为每一个IPC结构设置了一个ipc_perm结构。该结构规定了许可权和所有者。

```
struct  ipc_perm {
  uid_t   uid;         /* owner's effective user id */
  gid_t   gid;         /* owner's effective group id */
  uid_t   cuid;        /* creator's effective user id */
  gid_t   cgid;        /* creator's effective group id */
  mode_t  mode;        /* access modes */
  ulong   seq;         /* slot usage sequence number */
  key_t   key;         /* key */
}
```

在创建IPC结构时，除seq以外的所有字段都赋初值。以后，可以调用 msgctl、semctl或shmctl修改uid、gid和mode字段。为了改变这些值，调用进程必须是IPC结构的创建者或超级用户。更改这些字段类似于对文件调用chown和chmod。

mode字段的值类似于表4-4中所示的值，但是对于任何IPC结构都不存在执行许可权。另外，消息队列和共享存储使用术语“读”和“写”，而信号量则用术语“读”和“更改”。表14-2中对每种IPC说明了6种许可权。

表14-2 系统V IPC许可权

许可权	消息队列	信号量	共享存储
用户读	MSG_R	SEM_R	SHM_R
用户写(更改)	MSG_W	SEM_A	SHM_W
组读	MSG_R >> 3	SEM_R >> 3	SHM_R >> 3
组写(更改)	MSG_W >> 3	SEM_A >> 3	SHM_W >> 3
其他读	MSG_R >> 6	SEM_R >> 6	SHM_R >> 6
其他写(更改)	MSG_W >> 6	SEM_A >> 6	SHM_W >> 6

14.6.3 结构限制

三种形式的系统V IPC都有我们可能会遇到的内在限制。这些限制的大多数可以通过重新配置而加以更改。当叙说每种IPC时，都会指出它的限制

> 在SVR4中，这些值以及它们的最小、最大值都在文件/etc/conf/cf.d/mtune中。

14.6.4 优点和缺点

系统V IPC的主要问题是；IPC结构是在系统范围内起作用的，没有访问计数。例如，如果创建了一个消息队列，在该队列中放入了几则消息，然后终止，但是该消息队列及其内容并不被删除。它们余留在系统中直至：由某个进程调用 msgrcv或msgctl读消息或删除消息队列，或某个进程执行ipcrm(1)命令删除消息队列；或由正在再起动的系统删除消息队列。将此与管道pipe相比，那么当最后一个访问管道的进程终止时，管道就被完全地删除了。对于 FIFO而言虽

然当最后一个引用FIFO的进程终止时其名字仍保留在系统中，直至显式地删除它，但是留在FIFO中的数据却在此时全部删除。

系统V IPC的另一个问题是；这些IPC结构并不按名字为文件系统所知。我们不能用第3、4章中所述的函数来存取它们或修改它们的特性。为了支持它们不得不增加了十多个全新的系统调用（msgget、semop、shmat等）。我们不能用ls命令见到它们，不能用rm命令删除它们，不能用chmod命令更改它们的存取权。于是，也不得不增加了全新的命令ipcs和ipcrm。

因为这些IPC不使用文件描述符，所以不能对它们使用多路转接I/O函数：select和poll。这就使得一次使用多个IPC结构，以及用文件或设备I/O来使用IPC结构很难做到。例如，没有某种形式的忙-等待循环，就不能使一个服务器等待一个消息放在两个消息队列的任一一个中。

Andrade、Carges以及Kovach〔1989〕对使用系统V IPC的一个实际事务处理系统进行了综述。他们认为系统V IPC使用的名字空间（标识符）是一个优点而不是前面所说的问题，理由是使用标识符使一个进程只要使用单个函数调用（ msgsnd）就能将一个消息发送到一个队列，而其他形式的IPC则通常要求open、write和close。这种论据是不真实的。为了避免使用一个关键字和调用msgget，客户机总要以某种方式获得服务器队列的标识符。分派给特定队列的标识符取决于在创建该队列时，有多少消息队列已经存在，取决于自内核自举以来，内核中将分配给新队列的表项已经使用了多少次。这是一个动态值，不能被猜测或事先存放在一个头文件中。正如14.6.1节所述，至少服务器应将分配给队列的标识符写到一个文件中以便客户机读取。

这些作者列举的消息队列的其他优点是：（a）它们是可靠的，（b）流是受到控制的，（c）面向记录，（d）可以用非先进先出方式处理。正如在12.4节中所见，流也具有所有这些优点，虽然在向一个流发送数据之前，需要一个open，在结束时需要一个close。表14-3对这些不同形式的IPC的某些特征进行了比较。

表14-3 不同形式IPC之间特征的比较

类 型	无连接?	可靠?	流控制?	记录?	消息类型或优先权?
消息队列	否	是	是	是	是
流	否	是	是	是	是
UNIX流套接口	否	是	是	否	否
UNIX数据报套接口	是	是	否	是	否
FIFO	否	是	是	否	否

（第15章将对UNIX流和数据报套接口进行简要说明。）表中的“无连接”指的是无需先调用某种形式的open，就能发送消息的能力。正如前述，因为需要有某种技术以获得队列标识符，所以我们并不认为消息队列具有无连接特性。因为所有这些形式的IPC都限制用在单主机上，所以它们都是可靠的。当消息通过网络传送时，丢失消息的可能性就要加以考虑。流控制的意思是：如果系统资源短缺（缓存）或者如果接收进程不能再接收更多消息，则发送进程就要睡眠。当流控制条件消失时，发送进程应自动地被唤醒。

表14-3中没有表示的一个特征是：IPC设施能否自动地为每个客户机自动地创建一个到服务器的唯一连接。第15章将说明，流以及UNIX流套接口可以提供这种能力。

下面三节顺次对三种形式的系统V IPC进行详细说明。

14.7 消息队列

消息队列是消息的链接表，存放在内核中并由消息队列标识符标识。我们将称消息队列为

“队列”，其标识符为“队列ID”。msgget用于创建一个新队列或打开一个现存的队列。msgsnd用于将新消息添加到队列尾端。每个消息包含一个正长整型类型字段，一个非负长度以及实际数据字节（对应于长度），所有这些都在将消息添加到队列时，传送给msgsnd。msgrcv用于从队列中取消息。我们并不一定要以先进先出次序取消息，也可以按消息的类型字段取消息。

每个队列都有一个msqid_ds结构与其相关。此结构规定了队列的当前状态。

```
struct msqid_ds {
  struct ipc_perm  msg_perm; /* see Section 14.6.2 */
  struct msg  *msg_first;  /* ptr to first message on queue */
  struct msg  *msg_last;   /* ptr to last message on queue */
  ulong        msg_cbytes; /* current # bytes on queue */
  ulong        msg_qnum;   /* # of messages on queue */
  ulong        msg_qbytes; /* max # of bytes on queue */
  pid_t        msg_lspid;  /* pid of last msgsnd() */
  pid_t        msg_lrpid;  /* pid of last msgrcv() */
  time_t       msg_stime;  /* last-msgsnd() time */
  time_t       msg_rtime;  /* last-msgrcv() time */
  time_t       msg_ctime;  /* last-change time */
};
```

两个指针msg-first和msg-last分别指向相应消息在内核中的存放位置，所以它们对用户进程而言是无价值的。结构的其他成员是自定义的。

表14-4列出了影响消息队列的系统限制（见14.6.3节）。

表14-4 影响消息队列的系统限制

名 字	说 明	典型值
MSGMAX	可发送的最长消息的字节长度	2048
MSGMNB	特定队列的最大字节长度（亦即队列中所有消息之和）	4096
MSGMNI	系统中最大消息队列数	50
MSGTOL	系统中最大消息数	50

调用的第一个函数通常是msgget，其功能是打开一个现存队列或创建一个新队列。

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

int msgget(key_t key, int flag);
```

返回：若成功则为消息队列ID，若出错则为-1

14.6节说明了将*key*变换成一个标识符的规则，并且讨论是否创建一个新队列或访问一个现存队列。当创建一个新队列时，初始化msqid-ds结构的下列成员：

- ipc-perm结构按14.6.2节中所述进行初始化。该结构中mode按*flag*中的相应许可权位设置。这些许可权用表14-2中的常数指定。
- msg_qnum,msg_lspid、msg_lrpid、msg_stime和msg_rtime都设置为0。
- msg_ctime设置为当前时间。
- msg_qbytes设置为系统限制值。

若执行成功，则返回非负队列ID。此后，此值就可被用于其他三个消息队列函数。

msgctl函数对队列执行多种操作。它以及另外两个与信号量和共享存储有关的函数 (semctl和shmctl)是系统V IPC的类似于ioctl的函数（亦即垃圾桶函数）。

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

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

返回：若成功则为0，出错则为-1

*cmd*参数指定对于由*msqid*规定的队列要执行的命令：

• IPC_STAT 取此队列的msqid_ds结构，并将其存放在*buf*指向的结构中。

• IPC_SET 按由*buf*指向的结构中的值，设置与此队列相关的结构中的下列四个字段：msg_perm.uid、msg_perm.gid、msg_perm;mode和msg_qbytes。此命令只能由下列两种进程执行：一种是其有效用户ID等于msg_perm.cuid或msg_perm.uid;另一种是具有超级用户特权的进程。只有超级用户才能增加msg_qbytes的值

• IPC_RMID 从系统中删除该消息队列以及仍在该队列上的所有数据。这种删除立即生效。仍在使用这一消息队列的其他进程在它们下一次试图对此队列进行操作时，将出错返回EIDRM。此命令只能由下列两种进程执行：一种是其有效用户 ID等于msg_perm.cuid或msg_perm.uid;另一种是具有超级用户特权的进程。

这三条命令（IPC_STAT、IPC_SET和IPC_RMID）也可用于信号量和共享存储。

调用msgsnd将数据放到消息队列上。

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

int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

返回：若成功则为0，若出错则为-1

正如前面提及的，每个消息都由三部分组成，它们是：正长整型类型字段、非负长度（*nbytes*）以及实际数据字节（对应于长度）。消息总是放在队列尾端。

*ptr*指向一个长整型数，它包含了正整型消息类型，在其后立即跟随了消息数据。（若*nbytes*是0，则无消息数据。）若发送的最长消息是512字节，则可定义下列结构：

```
struct mymesg {
  long mtype;       /* positive message type */
  char mtext[512]; /* message data, of length nbytes */
};
```

于是，*ptr*就是一个指向mymesg结构的指针。接收者可以使用消息类型以非先进先出的次序取消息。

*flag*的值可以指定为IPC_NOWAIT。这类似于文件I/O的非阻塞I/O标志（见12.2节）。若消息队列已满（或者是队列中的消息总数等于系统限制值，或队列中的字节总数等于系统限制值），则指定IPC_NOWAIT使得msgsnd立即出错返回EAGAIN。如果没有指定IPC_NOWAIT，则进程阻塞直到（a）有空间可以容纳要发送的消息，或（b）从系统中删除了此队列，或（c）捕捉

到一个信号，并从信号处理程序返回。在第二种情况下，返回 EIDRM（“标志符被删除”）。最后一种情况则返回EINTR。

注意，对消息队列删除的处理不是很完善。因为对每个消息队列并没有设置一个引用计数器（对打开文件则有这种计数器），所以删除一个队列使得仍在使用这一队列的进程在下次对队列进行操作时出错返回。信号量机构也以同样方式处理其删除。删除一个文件则要等到使用该文件的最后一个进程关闭了它，才能删除文件的内容。

msgrcv从队列中取用消息。

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

int msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

返回：若成功则为消息数据部分的长度，若出错则为 -1

如同msgsnd中一样，*ptr*参数指向一个长整型数（返回的消息类型存放在其中），跟随其后的是存放实际消息数据的缓存。*nbytes*说明数据缓存的长度。若返回的消息大于 *nbytes*，而且在*flag*中设置了MSG_NOERROR，则该消息被截短（在这种情况下，不通知我们消息截短了）。如果没有设置这一标志，而消息又太长，则出错返回 E2BIG（消息仍留在队列中）。

参数*type*使我们可以指定想要哪一种消息：

- *type* == 0　返回队列中的第一个消息。
- *type* > 0　返回队列中消息类型为*type*的第一个消息。
- *type* < 0　返回队列中消息类型值小于或等于*type*绝对值，而且在这种消息中，其类型值又最小的消息。

非0*type*用于以非先进先出次序读消息。例如，若应用程序对消息赋优先权，那么 *type*就可以是优先权值。如果一个消息队列由多个客户机和一个服务器使用，那么 *type*字段可以用来包含客户机进程ID。

可以指定*flag*值为IPC_NOWAIT，使操作不阻塞。这使得如果没有所指定类型的消息，则msgrcv出错返回ENOMSG。如果没有指定IPC_NOWAIT，则进程阻塞直至（a）有了指定类型的消息，或（b）从系统中删除了此队列（出错返回 EIDRM），或（c）捕捉到一个信号并从信号处理程序返回（出错返回EINTR）。

实例——消息队列与流管道的时间比较

如若需要客户机和服务器之间的双向数据流，可以使用消息队列或流管道（ 15.2节将介绍流管道，它与管道类似，但是是全双工的）。

表14-5显示了在两个不同系统上这两种技术在时间方面的比较。测试程序先创建 IPC通道，调用fork，然后从父进程向子进程发送 20M字节数据。数据发送的方式是：对于消息队列，调用10 000次msgsnd，每个消息长度为2000字节；对于流管道，调用10 000次write,每次写2000字节。时间都以秒为单位。

在SPARC上，流管道是用UNIX域套接口实现的。在SVR4之下，pipe函数提供流管道（使用12.4节所述的流机制）。

从这些数字中可见，消息队列原来的实施目的是提供比一般 IPC更高速度的进程通信方法，

但现在与其他形式的IPC相比，在速度方面已经没有什么差别了。（在原来实施消息队列时，唯一的其他形式的IPC是半双工管道。）考虑到使用消息队列具有的问题（见14.6.4节），我们得出的结论是，在新的应用程序中不应当再使用它们。

表14-5 消息队列和流管道的时间比较

操 作	SPARC,SunOS 4.1.1			80386,SVR4		
	用户	系统	时钟	用户	系统	时钟
消息队列	0.8	10.7	11.6	0.7	19.6	20.1
流管道	0.3	10.6	11.0	0.5	21.4	21.9

14.8 信号量

信号量与已经介绍过的IPC机构（管道、FIFO以及消息列队）不同。它是一个计数器，用于多进程对共享数据对象的存取。为了获得共享资源，进程需要执行下列操作：

(1) 测试控制该资源的信号量。

(2) 若此信号量的值为正，则进程可以使用该资源。进程将信号量值减 1，表示它使用了一个资源单位。

(3) 若此信号量的值为 0，则进程进入睡眠状态，直至信号量值大于 0。若进程被唤醒后，它返回至(第(1)步)。

当进程不再使用由一个信息量控制的共享资源时，该信号量值增 1。如果有进程正在睡眠等待此信号量，则唤醒它们。

为了正确地实现信息量，信号量值的测试及减 1操作应当是原子操作。为此，信号量通常是在内核中实现的。

常用的信号量形式被称之为双态信号量(binary semaphore)。它控制单个资源，其初始值为1。但是，一般而言，信号量的初值可以是任一正值，该值说明有多少个共享资源单位可供共享应用。

不幸的是，系统V的信号量与此相比要复杂得多。三种特性造成了这种并非必要的复杂性：

(1) 信号量并非是一个非负值，而必需将信号量定义为含有一个或多个信号量值的集合。当创建一个信号量时，要指定该集合中的各个值

(2) 创建信息量（semget）与对其赋初值（semctl）分开。这是一个致命的弱点，因为不能原子地创建一个信号量集合，并且对该集合中的所有值赋初值。

(3) 即使没有进程正在使用各种形式的系统 V IPC，它们仍然是存在的，所以不得不为这种程序担心，它在终止时并没有释放已经分配给它的信号量。下面将要说明的 undo功能就是假定要处理这种情况的。

内核为每个信号量设置了一个semid_ds结构。

```
struct semid_ds {
  struct ipc_perm  sem_perm;* see Section 14.6.2 */
  struct sem  *sem_base;/* ptr to first semaphore in set */
  ushort      sem_nsems; /* #of semaphores in set */
  time_t      sem_otime; /* last-semop() time */
  time_t      sem_ctime; /* last-change time */
};
```

对用户而言，sem_base指针是没有价值的，它指向内核中的 sem结构数组，该数组中包含了sem_nsems个元素，每个元素各对应于集合中的一个信号量值。

```
struct sem {
  ushort semval;    /* semaphore value, always >= 0 */
  pid_t  sempid;    /* pid for last operarion */
  ushort semncnt;   /* # processes awaiting semval > currval */
  ushort semzcnt;   /* # processes awaiting semval = 0 */
};
```

表14-6列出了影响信号量集合的系统限制（见14.6.3节）。

表14-6 影响信号量的系统限制

名 字	说 明	典型值
SEMVMX	任一信号量的最大值	32 767
SEMAEM	任一信号量的最大终止时调整值	16 384
SEMMNI	系统中信号量集的最大数	10
SEMMNS	系统中信号量集的最大数	60
SEMMSL	每个信号量集中的最大信号量数	25
SEMMNU	系统中undo结构的最大数	30
SEMUME	每个undo结构中的最大undo项数	10
SEMOPM	每个semop调用所包含的最大操作数	10

要调用的第一个函数是semget以获得一个信号量ID。

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

int semget(key_t key, int nsems, int flag);
```

返回：若成功则返回信号量ID，若出错则为-1

14.6.1节说明了将*key*变换为标识符的规则，讨论了是否创建一个新集合，或是引用一个现存的集合。但创建一个新集合时，对semid_ds结构的下列成员赋初值：

- 按14.6.2节中所述，对ipc_perm结构赋初值。该结构中的mode被设置为*flag*中的相应许可权位。这些许可权是用表14-2中的常数设置的。
- sem_otime设置为0。
- sem_ctime设置为当前时间。
- sem_nsems设置为*nsems*。

*nsems*是该集合中的信号量数。如果是创建新集合（一般在服务器中），则必须指定*nsems*。如果引用一个现存的集合（一个客户机），则将*nsems*指定为0。

semctl函数包含了多种信号量操作。

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

int semctl(int semid, int semnum, int cmd, union semun arg);
```

返回：(见下)

注意，最后一个参数是个联合（union），而非指向一个联合的指针。

```
union semun {
  int                 val;       /* for SETVAL */
  struct semid_ds   *buf;        /* for IPC_STAT and IPC_SET */
  ushort             *array;     /* for GETALL and SETALL */
};
```

*cmd*参数指定下列十种命令中的一种，使其在*semid*指定的信号量集合上执行此命令。其中有五条命令是针对一个特定的信号量值的，它们用 *semnum*指定该集合中的一个成员。 *semnum*值在0和*nsems*－1之间（包括0和*nsems*－1）。

• IPC_STAT　对此集合取semid_ds结构，并存放在由*arg.buf*指向的结构中。

• IPC_SET　按由*arg.buf*指向的结构中的值设置与此集合相关结构中的下列三个字段值：sem_perm.uid,sem_perm.gid和sem_perm.mode。此命令只能由下列两种进程执行：一种是其有效用户ID等于sem_perm.cuid或sem_perm.uid的进程;另一种是具有超级用户特权的进程。

• IPC_RMID　从系统中删除该信号量集合。这种删除是立即的。仍在使用此信号量的其他进程在它们下次意图对此信号量进行操作时，将出错返回 EIDRM。此命令只能由下列两种进程执行：一种是具有效用户ID等于sem_perm.cuid或sem_perm.uid的进程；另一种是具有超级用户特权的进程。

• GETVAL　返回成员*semnum*的semval值。

• SETVAL　设置成员*semnum*的semval值。该值由*arg.val*指定。

• GETPID　返回成员*semnum*的sempid值。

• GETNCNT　返回成员*semnum*的semncnt值。

• GETZCNT　返回成员*semnum*的semzcnt值。

• GETALL　取该集合中所有信号量的值，并将它们存放在由*arg.array*指向的数组中。

• SETALL　按*arg.array*指向的数组中的值设置该集合中所有信号量的值。

对于除GETALL以外的所有GET命令，semctl函数都返回相应值。其他命令的返回值为 0。函数semop自动执行信号量集合上的操作数组。

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

int semop(int semid, struct sembuf semoparray[], size_t nops);
```

返回：若成功则为0，若出错则为-1

---

*semoparray*是一个指针，它指向一个信号量操作数组。

```
struct sembuf {
  ushort   sem_num;   /* member # in set (0,1, nsems-1 */
  short    sem_op;    /* operation(negative, 0,or pasitive */)
  short    sem_flg;   /* IPC_NOWAIT, SEM_UNDO */
};
```

*nops*规定该数组中操作的数量（元素数）。

对集合中每个成员的操作由相应的sem_op规定。此值可以是负值、0或正值。（下面的讨论将提到信号量的undo标志。此标志对应于相应sem_flg成员的SEM_UNDO位。）

(1) 最易于处理的情况是sem_op为正。这对应于返回进程占用的资源。 sem_op值加到信号

量的值上。如果指定了undo标志，则也从该进程的此信号量调整值中减去sem_op。

(2) 若sem_op为负，则表示要获取由该信号量控制的资源。

如若该信号量的值大于或等于sem_op的绝对值（具有所需的资源），则从信号量值中减去sem_op的绝对值。这保证信号量的结果值大于或等于0。如果指定了undo标志，则sem_op的绝对值也加到该进程的此信号量调整值上。

如果信号量值小于sem_op的绝对值（资源不能满足要求），则：

(a) 若指定了IPC_NOWAIT，则出错返回EAGAIN;

(b) 若未指定IPC_NOWAIT，则该信号量的semncnt值加1（因为将进入睡眠状态），然后调用进程被挂起直至下列事件之一发生：

i. 此信号量变成大于或等于sem_op的绝对值（即某个进程已释放了某些资源）。此信号量的semncnt值减1（因为已结束等待），并且从信号量值中减去sem_op的绝对值。如果指定了undo标志,则sem_op的绝对值也加到该进程的此信号量调整值上。

ii. 从系统中删除了此信号量。在此情况下，函数出错返回ERMID。

iii. 进程捕捉到一个信号，并从信号处理程序返回，在此情况下，此信号量的semncnt值减1（因为不再等待），并且函数出错返回EINTR.

(3) 若sem_op为0，这表示希望等待到该信号量值变成0。

如果信号量值当前是0，则此函数立即返回。

如果信号量值非0，则：

(a) 若指定了IPC_NOWAIT，则出错返回EAGAIN；

(b) 若未指定IPC_NOWAIT，则该信号量的semncnt值加1（因为将进入睡眠状态），然后调用进程被挂起，直至下列事件之一发生：

i. 此信号量值变成0。此信号量的semzcnt值减1（因为已结束等待）。

ii. 从系统中删除了此信号量。在此情况下，函数出错返回ERMID。

iii. 进程捕捉到一个信号，并从信号处理程序返回。在此情况下，此信号量的semzcnt值减1（因为不再等待），并且函数出错返回EINTR.

semop具有原子性，因为它或者执行数组中的所有操作，或者一个也不做。

exit时的信号量调整

正如前面提到的，如果在进程终止时，它占用了经由信号量分配的资源，那么就会成为一个问题。无论何时只要为信号量操作指定了SEM_UNDO标志，然后分配资源（sem_op值小于0），那么内核就会记住对于该特定信号量，分配给我们多少资源（ sem_op的绝对值）。当该进程终止时，不论自愿或者不自愿，内核都将检验该进程是否还有尚未处理的信号量调整值，如果有，则按调整值对相应量值进行调整。

如果用带SETVAL或SETALL命令的semctl设置一信号量的值，则在所有进程中，对于该信号量的调整值都设置为0。

实例 —— 信号量与记录锁的时间比较

如果多个进程共享一个资源，则可使用信号量或记录锁。对这两种技术在时间上的差别进行比较是有益的。

若使用信号量，则先创建一个包含一个成员的信号量集合，然后对该信号量值赋初值 1。为了分配资源，以sem_op为－1调用semop，为了释放资源，则以sem_op为+1调用semop。对每个操作都指定SEM_UNDO，以处理在未释放资源情况下进程终止的情况。

若使用记录锁，则先创建一个空文件，并且用该文件的第一个字节（无需存在）作为锁字节。为了分配资源，先对该字节获得一个写锁，释放该资源时，则对该字节解锁。记录锁的性质确保了，当有一个锁的进程终止时，内核会自动释放该锁。

表14-7显示了在两个不同系统上，使用这两种不同技术进行锁操作所需的时间。在各种情况中，资源都被分配，然后释放10 000次。这同时由三个不同的进程执行。表14-7中所示的时间是三个进程的总计，单位是秒。

表14-7 信号量锁和记录锁的时间比较

操 作	SPARC,SunOS 4.1.1			80386,SVR4		
	用户	系统	时钟	用户	系统	时钟
带undo的信号量	0.9	13.9	15.0	0.5	13.1	13.7
建议性纪录锁	1.1	15.2	16.5	2.1	20.6	22.9

在SPARC上，记录锁与信号量锁相比，在系统时间方面要多耗用10%。在80386上，多耗用约50%。

虽然记录锁稍慢于信号量锁，但如果只需锁一个资源（例如共享存储段）并且不需要使用系统V信号量的所有花哨的功能，则宁可使用记录锁。理由是：(a)使用简易，(b）进程终止时，会处理任一遗留下的锁。

14.9 共享存储

共享存储允许两个或多个进程共享一给定的存储区。因为数据不需要在客户机和服务器之间复制，所以这是最快的一种IPC。使用共享存储的唯一窍门是多个进程之间对一给定存储区的同步存取。若服务器将数据放入共享存储区，则在服务器做完这一操作之前，客户机不应当去取这些数据。通常，信号量被用来实现对共享存储存取的同步。（不过正如前节最后部分所述，记录锁也可用于这种场合。）

内核为每个共享存储段设置了一个shmid_ds结构。

```
struct shmid_ds {
  struct ipc_perm  shm_perm; /* see Section 14.6.2 */
  struct anon_map *shm_amp;  /* pointer in kernel */
  int      shm_segsz;    /* size of segment in bytes */
  ushort   shm_lkcnt;    /* number of times segment is being locked */
  pid_t    shm_lpid;     /* pid of last shmop() */
  pid_t    shm_cpid;     /* pid of creator */
  ulong    shm_nattch;   /* number of current attaches */
  ulong    shm_cnattch;  /* used only for shminfo */
  time_t   shm_atime;    /* last-attach time */
  time_t   shm_dtime;    /* last-detach time */
  time_t   shm_ctime;    /* last-change time */
};
```

表14-8列出了影响共享存储的系统限制（见14.6.3节）。

调用的第一个函数通常是shmget，它获得一个共享存储标识符。

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

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

int shmget(key_t key, int size, int flag);
```

返回：若成功则为共享内存ID，若出错则为-1

表14-8 影响共享存储的系统限制

名 字	说 明	典型值
SHMMAX	共享存储段的最大字节数	131 072
SHMMIN	共享存储段的最小字节数	1
SHMMNI	系统中共享存储段的最大段数	100
SHMSEG	每个进程，共享存储段的最大段数	6

14.6.1节说明了将*key*变换成一个标识符的规则，以及是创建一个新共享存储段或是存访一个现存的共享存储段。当创建一个新段时，初始化shmid_ds结构的下列成员：

• ipc_perm结构按14.6.2节中所述进行初始化。该结构中的mode按*flag*中的相应许可权位设置。这些许可权用表14-2中的常数指定。

• shm_lpid、shm_nattach、shm_atime、以及shm_dtime都设置为0。

• shm_ctime设置为当前时间。

*size*是该共享存储段的最小值。如果正在创建一个新段（一般在服务器中），则必须指定其*size*。如果正在存访一个现存的段（一个客户机），则将*size*指定为0。

shmctl函数对共享存储段执行多种操作。

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

返回：若成功则为0，若出错则为-1

*cmd*参数指定下列5种命令中一种，使其在*shmid*指定的段上执行。

• IPC_STAT 对此段取shmid_ds结构，并存放在由*buf*指向的结构中。

• IPC_SET 按*buf*指向的结构中的值设置与此段相关结构中的下列三个字段：shm_perm.uid、shm_perm.gid以及shm_perm.mode。此命令只能由下列两种进程执行：一种是其有效用户ID等于shm_perm.cuid或shm_perm.uid的进程；另一种是具有超级用户特权的进程。

• IPC_RMID 从系统中删除该共享存储段。因为每个共享存储段有一个连接计数（shm_nattch在shmid_ds结构中），所以除非使用该段的最后一个进程终止或与该段脱接，否则不会实际上删除该存储段。不管此段是否仍在使用，该段标识符立即被删除，所以不能再用shmat与该段连接。此命令只能由下列两种进程执行：一种是其有效用户ID等于shm_perm.cuid或shm_perm.uid的进程；另一种是具有超级用户特权的进程。

• SHM_LOCK 锁住共享存储段。此命令只能由超级用户执行。

• SHM_UNLOCK 解锁共享存储段。此命令只能由超级用户执行。

一旦创建了一个共享存储段，进程就可调用shmat将其连接到它的地址空间中。

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

void *shmat(int shmid, void *addr, int flag);
```

返回：若成功则为指向共享存储段的指针，若出错则为 -1

共享存储段连接到调用进程的哪个地址上与 *addr*参数以及在*flag*中是否指定SHM_RND位有关。

(1) 如果*addr*为0，则此段连接到由内核选择的第一个可用地址上。

(2) 如果*addr*非0，并且没有指定SHM_RND，则此段连接到*addr*所指定的地址上。

(3) 如果*addr*非0，并且指定了SHM_RND，则此段连接到（*addr*－(*addr* mod SHMLBA)）所表示的地址上。SHM_RND命令的意思是：取整。SHMLBA的意思是：低边界地址倍数，它总是2的乘方。该算式是将地址向下取最近1个SHMLBA的倍数。

除非只计划在一种硬件上运行应用程序（这在当今是不大可能的），否则不用指定共享段所连接到的地址。所以一般应指定*addr*为0，以便由内核选择地址。

如果在*flag*中指定了SHM_RDONLY位，则以只读方式连接此段。否则以读写方式连接此段。

shmat的返回值是该段所连接的实际地址，如果出错则返回－1。

当对共享存储段的操作已经结束时，则调用 shmdt脱接该段。注意，这并不从系统中删除其标识符以及其数据结构。该标识符仍然存在，直至某个进程（一般是服务器）调用 shmctl（带命令IPC_RMID）特地删除它。

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

int shmdt(void *addr);
```

返回：若成功则为0，若出错则为 -1

*addr*参数是以前调用shmat时的返回值。

实例

内核将以地址0连接的共享存储段放在什么位置上与系统密切相关。程序 14-11打印一些信息，它们与指定系统将不同类型的数据放在什么位置有关。在一个特定的系统上运行此程序，其输出如下：

```
$ a.out
array[]  from 18f48 to 22b88
stack around f7fffb2c
malloced from 24c28 to 3d2c8
shared memory attached from f77d0000 to f77e86a0
```

图14-13显示了这种情况，这与图7-3中所示的典型存储区布局类似。注意，共享存储段紧靠在栈之下。实际上，在共享存储段和栈之间有大约8M字节的未用地址空间。

程序14-11 打印不同类型的数据所存放的位置

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

#define ARRAY_SIZE   40000
#define MALLOC_SIZE 100000
#define SHM_SIZE    100000
#define SHM_MODE    (SHM_R | SHM_W) /* user read/write */

char    array[ARRAY_SIZE];  /* uninitialized data = bss */

int
main(void)
{
    int     shmid;
    char    *ptr, *shmptr;

    printf("array[] from %x to %x\n", &array[0], &array[ARRAY_SIZE]);
    printf("stack around %x\n", &shmid);

    if ( (ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("malloc error");
    printf("malloced from %x to %x\n", ptr, ptr+MALLOC_SIZE);

    if ( (shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
        err_sys("shmget error");
    if ( (shmptr = shmat(shmid, 0, 0)) == (void *) -1)
        err_sys("shmat error");
    printf("shared memory attached from %x to %x\n",
                shmptr, shmptr+SHM_SIZE);
    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");

    exit(0);
}
```

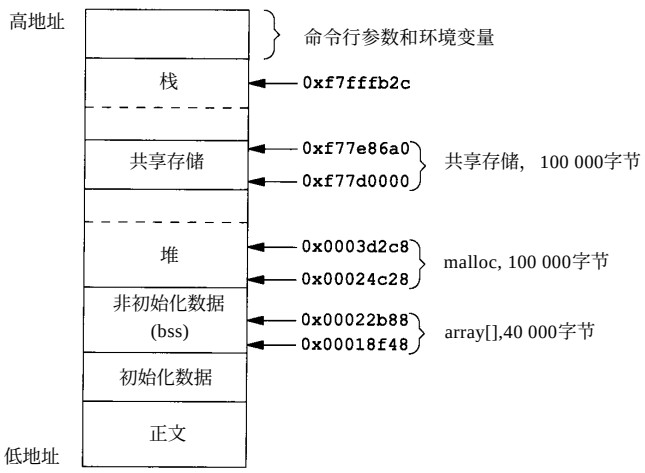

图14-13 特定系统上的存储区布局

实例——/dev/zero的存储映射

共享存储可由不相关的进程使用。但是，如果进程是相关的，则 SVR4提供了一种不同的

技术。

设备/dev/zero在读时，是0字节的无限资源。此设备也接收写向它的任何数据，但忽略此数据。我们对此设备作为IPC的兴趣在于，当对其进行存储映射时，它具有一些特殊性质：

- 创建一个未名存储区，其长度是mmap的第二个参数，将其取整为系统上的最近页长。
- 存储区都初始化为0。
- 如果多个进程的共同祖先进程对mmap指定了MAP_SHARED标志，则这些进程可共享此存储区。

程序14-12是使用此特殊设备的一个例子。它打开此/dev/zero设备，然后指定一个长整型调用mmap。注意，一旦该存储区被映射了，就能关闭此设备。然后，进程创建一个子进程。因为在调用mmap时指定了MAP_SHARED，所以一个进程写到存储映照区的数据可由另一进程见到。（如果已指定MAP_PRIVATE，则此程序不会工作。）

然后，父、子进程交替运行，使用 8.8节中的同步函数各自对共享存储映射区中的一个长整型数加1。存储映射区由mmap初始化为0。父进程先对它进行增1操作，使其成为1，然后子进程对其进行增1操作，使其成为2，然后父进程使其成为3……注意，当在update函数中，对长整型值增1时，必须使用括号，因为增加的是其值，而不是指针。

程序14-12　在父、子进程间使用/dev/zero存储映射I/O的IPC

```
#include    <sys/types.h>
#include    <sys/mman.h>
#include    <fcntl.h>
#include    "ourhdr.h"

#define NLOOPS      1000
#define SIZE        sizeof(long)    /* size of shared memory area */

static int  update(long *);

int
main()
{
    int     fd, i, counter;
    pid_t   pid;
    caddr_t area;

    if ( (fd = open("/dev/zero", O_RDWR)) < 0)
        err_sys("open error");
    if ( (area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                                    MAP_SHARED, fd, 0)) == (caddr_t) -1)
        err_sys("mmap error");
    close(fd);      /* can close /dev/zero now that it's mapped */

    TELL_WAIT();
    if ( (pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {           /* parent */
        for (i = 0; i < NLOOPS; i += 2) {
            if ( (counter = update((long *) area)) != i)
                err_quit("parent: expected %d, got %d", i, counter);
            TELL_CHILD(pid);
            WAIT_CHILD();
        }
    } else {                        /* child */
        for (i = 1; i < NLOOPS + 1; i += 2) {
            WAIT_PARENT();
            if ( (counter = update((long *) area)) != i)
```

```
                err_quit("child: expected %d, got %d", i, counter);
            TELL_PARENT(getppid());
        }
    }
    exit(0);
}
static int
update(long *ptr)
{
    return( (*ptr)++ ); /* return value before increment */
}
```

以上述方式使用/dev/zero的优点是：在调用mmap创建映射区之前，无需存在一个实际文件。映射/dev/zero自动创建一个指定长度的映射区。这种技术的缺点是：它只能由相关进程使用。如果在无关进程之间需要使用共享存储区，则必须使用shmXXX函数。

实例 —— 匿名存储映射

4.3+BSD提供了一种类似于/dev/zero的施设，称为匿名存储映射。为了使用这种功能，在调用mmap时指定MAP_ANON标志，并将描述符指定为－1。结果得到的区域是匿名的（因为它并不通过一个文件描述符与一个路径名相结合），并且创建一个存储区，它可与后代进程共享。

为了使程序14-12应用4.3+BSD的这种特征，做了两个修改：(a）删除/dev/zero的open条语句，(b）将mmap调用修改成下列形式：

```
if ( (area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                MAP_ANON | MAP_SHARED, -1, 0)) == (caddr_t) -1)
```

在此调用中，指定了MAP_ANON标志，并将文件描述符设置为－1。程序14-12的其余部分则不加改变。

14.10 客户机-服务器属性

下面详细说明客户机和服务器的属性，这些属性受到它们之间所使用的 IPC的不同类型的影响。

最简单的关系类型是使客户机fork并执行所希望的服务器。在fork之前先创建两个单向管道以使数据可在两个方向传输。图 14-8是这种形式的一个例子。被执行的服务器可能是设置 -用户-ID的程序，这使它具有了特权。查看客户机的实际用户 ID就可以决定客户机的身份。(回忆8.9节，从中可了解到在exec前后实际用户ID和实际组ID并没有改变。)

在这种安排下，可以构筑一个“开放式服务器”(15.4节即提供了这种客户机和服务器的一种实现)。它为客户机开放文件而不是客户机调用open函数。这样就可以增加在正常的UNIX用户/组/其他许可权之上或之外的附加的许可权检查。假定服务器是设置 -用户-ID程序，这给予了它附加的许可权（很可能是root许可权)。服务器用客户机的实际用户 ID以决定是否给予它对所要求的文件的存取。使用这种方式，可以构筑一个服务器，它允许某种用户通常没有的存取权。

在此例子中，因为服务器是父进程的子进程，所以它能做的一切是将文件内容传送给父进程。这种方式对一般文件工作得很好，同时，也可被用于专用设备文件。我们希望能做的是使服务器打开所要的文件，并将文件描述符送回。父进程可向子进程传送打开文件描述符，而子进程则不能向父进程传回一个描述符（除非使用将在下一章介绍的专门编程技术)。

下一种服务器类型已显示于图 14-12中，服务器是一个精灵进程，客户机则用某种形式的

IPC与其联系。可以将管道用于这种形式的客户机 -服务器关系。要求有一种命名的 IPC，例如FIFO或消息队列。对于 FIFO，如果服务器必需将数据送回客户机，则对每个客户机都要有单独使用的FIFO。如果客户机-服务器应用程序只有客户机向服务器送数据，则只需要一个众所周知的FIFO。（系统V行式打印机假脱机程序使用这种形式的客户机 -服务器。客户机是lp(1)命令，服务器是lpsched进程。因为只有从客户机到服务器的数据流，没有任何数据需送回客户机，所有只需使用一个FIFO。

使用消息队列则存在多种可能性：

(1) 在服务器和客户机之间可以只使用一个队列，使用每个消息的类型字段指明谁是消息的接受者。例如，客户机可以用类型字段为 1发送它们的消息。在要求之中应包括客户机的进程ID。此后，服务器在发送响应消息时，将类型字段设置为客户机的进程 ID。服务器只接受类型字段为1的消息（msgrcv的第四个参数），客户机则只接受类型字段等于它们的进程ID的消息。

(2) 另一种方法是每个客户机使用一个单独的消息队列。在向服务器发送第一个请求之前，每个客户机先创建它自己的消息队列，创建时使用关键字 IPC_PRIVATE。服务器也有它自己的队列，其关键字或标识符是所有客户机知道的。客户机将其第一个请求送到服务器的众所周知的队列上，该请求中应包含其客户机消息队列的队列 ID。服务器将其第一个响应送至客户机队列，此后的所有请求和响应都在此队列上交换。

使用这种技术的一个问题是：每个客户机专用队列通常只有一个消息在其中 —— 或者是对服务器的一个请求，或者是对客户机的响应。这似乎是对有限的系统资源（消息队列）的浪费，可以用一个FIFO来代替。另一个问题是服务器需从多个队列读消息。对于消息队列， select和poll都不起作用

使用消息队列的这两种技术都可以用共享存储段和同步方法（信号量或记录锁）实现。使用共享存储段的问题是，一次只能有一个消息在共享存储段中 —— 类似于队列限制为只能有一个消息。为此，在使用共享存储IPC时，通常每个客户机使用一个共享存储段。

这种类型的客户机 -服务器关系（客户机和服务器是无关系进程）的问题是：服务器如何准确地标识客户机。除非服务器正在执行一种非特权操作，否则服务器知道谁是客户机是很重要的。例如，若服务器是一个设置 -用户-ID程序，就有这种要求。虽然，所有这几种形式的IPC都经由内核，但是它们并未提供任何措施使内核能够标识发送者。

对于消息队列，如果在客户机和服务器之间使用一个专用队列（于是一次只有一个消息在该队列上），那么队列的msg_lspid包含了对方进程的进程ID。但是当客户机将请求发送给服务器时，我们想要的是客户机的有效用户ID，而不是它的进程ID。现在还没有一种可移植的方法，在已知进程ID情况下用其可以得到有效用户 ID。（确实，内核在进程表项中保持有这两种值，但是除非彻底检查内核存储空间，否则已知一个，无法得到另一个。）

我们将在15.5.2中使用下列技术,使服务器可以标识客户机。同样的技术也可用于 FIFO、消息队列、信号量或共享存储。下面的说明具体针对按图 14-12中的方式使用FIFO情况。客户机必须创建它自己的FIFO，并且设置FIFO的文件存取许可权，使得只允许用户 -读，用户-写。假定服务器具有超级用户特权（或者它很可能并不关心客户机的真实标识），所以服务器仍可读、写此FIFO。当服务器在众所周知的 FIFO上接受到客户机的第一个请求时（它应当包含客户机专用FIFO的标识），服务器调用针对客户机专用 FIFO的stat或fstat。服务器所采用的假设是；客户机的有效用户ID是FIFO的所有者（stat结构的st_uid字段）。服务器验证该FIFO只有用户-读、用户-写许可权。服务器还应检查是该 FIFO的三个时间量（ stat结构中的st_atime,st_mtime和st_ctime字段），要检查它们与当前时间是否很接近（例如不早于当前时间 15秒或30秒）。如果

一个有预谋的客户机可以创建一个FIFO，使另一个用户成为其所有者，并且设置该文件的许可权为用户-读和用户-写，那么在系统中就存在了其他基础性的安全问题。

为了在系统V IPC中应用这种技术，回想一下与每个消息队列、信号量、以及共享存储段相关的ipc_perm结构，其中cuid和cgid字段标识IPC结构的创建者。以FIFO为例，服务器应当要求客户机创建该IPC结构，并使客户机将存取权设置为只允许用户-读和用户-写。服务器也应检验与该IPC相关的时间量与当前时间是否很接近（因为这些IPC结构在显式地删除之前一直存在）。

在15.5.1节中，将会看到进行这种身份验证工作的一种更好方法是内核提供客户机的有效用户ID和有效组ID。SVR4在进程之间传送文件描述符时可以做到这一点。

14.11 小结

本章详细说明了进程间通信的多种形式；管道、命名管道（FIFO）以及另外三种IPC形式，通常称之为系统V IPC——消息队列、信号量和共享存储。信号量实际上是同步原语而不是IPC，常用于共享资源的同步存取，例如共享存储段。对于管道，说明了popen的实现，说明了协同进程，以及使用标准I/O库缓存机制时可能遇到的问题。

在时间方面，对消息队列与流管道、信号量与记录锁做了比较后，提出了下列建议：学会使用管道和FIFO，因为在大量应用程序中仍可有效地使用这两种基本技术。在新的应用程序中，要尽可能避免使用消息队列以及信号量，而应当考虑流管道和记录锁，因为它们与UNIX内核的其他部分集成得要好得多。共享存储段有其应用场合，而mmap函数（见12-9节）则可能在以后的版本中起更大作用。

下一章将介绍一些更先进的IPC形式，它们由更加新的系统，例如SVR4和4.3+BSD提供。

习题

14.1 在程序14-2中父进程代码的末尾，如删除waitpid前的close，结果将如何？

14.2 在程序14-2中父进程代码的末尾，如删除waitpid，结果将如何？

14.3 如果popen的参数是一个不存在的命令会有什么结果？编写一段程序测试一下。

14.4 删除程序14-9中的信号量处理程序，执行程序并终止子进程。输入一行后，怎样能说明父进程是由SIGPIPE终止的？

14.5 将程序14-9中进行管道读、写的read和write用标准I/O库代替。

14.6 POSIX.1加入waitpid函数的理由之一是POSIX.1前的大多数系统不能处理下面的代码。

```
if ( (fp = popen("/bin/true", "r")) == NULL )
    ...
if ( (rc = system("sleep 100")) == -1)
    ...
if (pclose (fp) == -1)
    ...
```

若在这段代码中不使用waitpid函数会如何？用wait代替呢？

14.7 当一个管道被写者关闭后，解释select和poll如何处理该管道的输入描述符？当一个管道的读端被关闭时，请重做此习题以查看该管道的输出描述符。编两个测试程序，一个用select一个用poll，并确定答案是否正确。

14.8 如果popen以*type*为r执行*cmdstring*并将结果写到标准出错输出，结果如何？

14.9 popen会使得shell执行它的*cmdstring*参数，当*cmdstring*终止时会产生什么结果？（提

示：画出包含的所有进程。）

14.10 大多数UNIX系统允许读写FIFO，但是POSIX.1特别声明没有定义为读写而打开FIFO。请用非阻塞方法实现为读写而打开FIFO。

14.11 除非文件包含敏感或机密数据，否则允许其他用户读文件不会造成损害。但是，如果一个恶意进程读取了被一个服务器和几个客户机进程使用的消息队列中的一条消息后会有什么结果？恶意进程需要知道哪些信息就可以读消息队列？

14.12 编写一段程序完成下面的工作：循环五次创建一个消息队列，并打印队列的标识符，然后删除队列。接着再循环五次利用关键字IPC_PRIVATE创建消息队列并将一条消息放在队列中，程序终止后用ipcs(1)查看消息队列。解释队列标识符的变化。

14.13 描述如何在共享存储段中建立一个数据对象的连接列表。列表指针如何保存？

14.14 画出程序14-12的变量i在父进程和子进程中随时间变化的值。由update函数返回该值，其类型为长整型，保存在共享存储区。假设fork后子进程先运行。

14.15 使用14.9节的shmXXX函数代替共享存储映射区重写程序14-12。

14.16 使用14.8节系统V提供的信号量函数重写程序14-12实现父进程与子进程间的交替。

14.17 使用建议性记录锁定方法重写程序14-12实现父进程与子进程间的交替。

14.18 解释mmap函数的文件描述符参数如何在4.3+BSD系统匿名存储映射方式下允许不相关进程间的存储共享。