

第 14 章 进程间通信

进程间通信 IPC（Interprocess Communication）是在 Linux/UNIX 下编程经常会碰到的问题，它的实际意义在于怎么样让多个进程可以互相的访问数据。在 Linux/UNIX 环境下可以由多种方式来实现上述的问题，接下来将详细为读者介绍这几种方式的具体操作以及相关内容。

14.1 进程间通信概述

进一个大型的应用系统，往往需要众多进程协作，进程间通信的重要性显而易见。本章阐述了 Linux 环境下的几种主要进程间通信手段，并针对每个通信手段的关键技术环节给出详细实例。为达到阐明问题的目的，本章还对某些通信手段的内部实现机制进行了分析。

14.1.1 进程间通信简介

在早期，UNIX 系统 IPC 就是进程间通信方式的统称，进程间通信就是可以让多个进程可以互相之间访问。这种访问包括程序运行的适时数据，也包括对方的代码段，这是在实际应用中及其常见的问题，进程间通信示意图如图 14-1 所示。

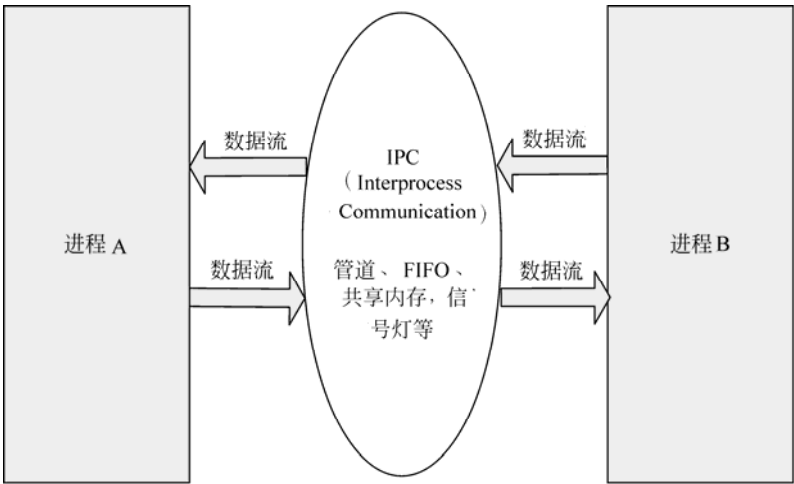


图 14-1 进程间通信简介

上图所示进程间通信的模式,进程 A 和进程 B 在运行的过程中会需要一些外部的数据,IPC 为两个进程提供了一种数据传输的通道。

14.1.2 进程间通信的难点

由于现在应用程序的体积逐渐增大,用户对软件的功能要求也越来越多,所以多进程设计已经是应用技术中不可缺少的一部分,这同时也对进程间通信提出了挑战。

进程运行期间,其地址空间对于其他进程是不可见的(这只是传统上的进程概念,在本章后续章节中 IPC 内存共享机制打破了这个概念),在系统中它们是相对独立的,并不能互相访问对方,如图 14-2 所示。

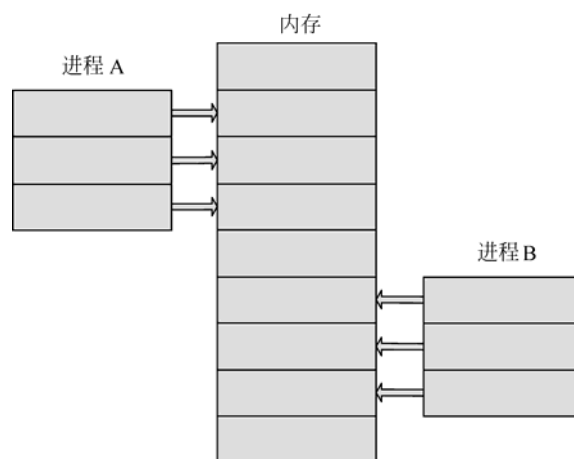


图 14-2 进程在内存中的地址

Linux/UNIX 系统提供一种中间转发的机制,为多个进程建立起互相通信的数据通道。在上述问题中,当进程 A 与进程 B 通信时,通过中间的 IPC 方式来转发数据到目标进程。

14.1.3 IPC 的多种方式

Linux/UNIX 系统下,进程间通信方式有着比较多的应用技术。进程间通信方式从最简单的使用文件系统实现多进程共享文件数据、以及父子进程共享数据段,到高级应用的管道以及共享内存、信号灯。

IPC 是所有 UNIX 系统中各种进程间通信的统称,如表 14-1 所示,列出了所有进程间通信的方式。

表 14-1 IPC 类型

IPC 类型	详细说明
半双工管道	半双工管道: 匿名半双工管道 FIFO(First In First Out 命名半双工管道) 全双工管道: 匿名全双工管道 命名全双工管道
System V IPC / POSIX IPC	消息队列 信号量 共享存储
网络进程间通信	SOCKET STREAMS

上表内容中，全双工管道是在最近才出现的一种技术，它是在半双工管道上的扩充，在有的系统中是不被支持的，在实际应用的过程中请参阅相应的系统手册。

🔔 **注意：**其余的技术中，在不同的系统中会有不同的限制，以及不同的特点。本书只描述它们共有的特性。如需特别需求，请参阅相应的系统手册。

14.2 管道

管道通信是最常见的通信方式之一，其是在两个进程之间实现一个数据流通的管道，该管道可以是双向或单向的。管道是一种很经典的进程之间的通信方式，其优点在于简单易用，其缺点在于功能简单，有很多限制。本小节主要介绍管道操作。

14.2.1 管道的概念

管道是 Linux/UNIX 系统中比较原始的进程间通信形式，它实现数据以一种数据流的方式，在多进程间流动。在系统中其相当于文件系统上的一个文件，来缓存所要传输的数据。在某些特性上又不同于文件，例如，当数据读出后，则管道中就没有数据了，但文件没有这个特性。

匿名半双工管道在系统中是没有实名的，并不可以在文件系统中以任何方式看到该管道。它只是进程的一种资源，会随着进程的结束而被系统清除。管道通信是在 UNIX 系统中应用比较频繁的一种方式，例如使用 `grep` 查找。

```
# ls | grep ipc
```

上述命令中使用的是半双工管道，即 `grep` 命令的输入是 `ls` 命令的输出。管道从数据流动方向上又分全双工管道以及半双工管道，当然全双工管道现在某些系统还不支持，其在具体的实现过程中也只是在文件打开的方式上有一点区别（在操作规则上也有一些不同，全双工管道要相比半双工复杂的多）。

14.2.2 匿名半双工管道

匿名管道没有名字，对于管道中使用的文件描述符没有路径名，也就是不存在任何意义上的文件，它们只是在内存中跟某一个索引节点相关联的两个文件描述符。匿名半双工管道的主要特性如下：

- ❑ 数据只能在一个方向上移动。
- ❑ 只能在具有公共祖先的进程间通信，即或是父子关系进程间、或是在兄弟关系进程间通信。

尽管有如此限制，半双工管道还是最常用的通信方式。Linux 环境下使用 `pipe` 函数创建一个匿名半双工管道，其函数原型如下：

```
#include <unistd.h>

int pipe ( int fd[2] ) ;
```

参数 `int fd[2]` 为一个长度为 2 的文件描述符数组，`fd[0]` 是读出端，`fd[1]` 是写入端，函数的返回值为 0 表示成功，-1 表示失败。当函数成功返回，则自动维护了一个从 `fd[1]` 到 `fd[0]` 的数据通道。

下面实例演示了如何使用 `pipe` 函数创建管道以及关闭管道。程序中先使用函数 `pipe` 建立管道，并使用管道传输数据，在程序的结束部分，释放掉管道占用的文件资源（两个文件描述符），具体实现如下。

(1) 在 vi 编辑器中编辑以下程序：

程序清单 14-1 opro_pipe.c 管道的打开以及关闭操作

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

int main( void )
{
    int fd[2];                /* 管道的文件描述符数组 */
    char str[256];

    if ( (pipe(fd)) < 0 ){
        perror("pipe");
        exit(1);
    }

    write(fd[1], "create the pipe successfully !\n", 31 );
    /*向管道写入端写入数据*/
    read(fd[0], str, sizeof(str) );    /*从管道读出端读出数据*/
    printf ("%s", str );

    printf ( " pipe file descriptors are %d, %d \n", fd[0], fd[1]) ;

    close (fd[0]);             /* 关闭管道的读入文件描述符*/
    close (fd[1]);             /* 关闭管道的读出文件描述符*/

    return 0;
}
```

(2) 在 shell 中编译该程序如下：

```
$gcc opro_pipe.c-o opro_pipe
```

(3) 在 shell 中运行该程序如下：

```
$. ./ opro_pipe
create the pipe successfully !
```

```
pipe file descriptors are 4, 5
```

程序中使用 `pipe` 函数建立了一个匿名管道 `fd`。

注意：文件描述符数组 `fd` 并没有和任何有名文件相关联，之后向管道一端写入数据并从读出端读出数据，将数据输出到标准输出。在程序的最后使用 `close` 函数关闭管道的两端。

14.2.3 匿名半双工管道的读写操作

当对管道进行读写操作时，使用 `read` 和 `write` 函数对管道进行操作。当对一个读端已经关闭的管道进行写操作时，会产生信号 `SIGPIPE`，说明管道读端已经关闭，并且 `write` 操作返回为 -1，`errno` 的值为 `EPIPE`，对于 `SIGPIPE` 信号可以进行捕捉处理。如果写入进程不能捕捉或者干脆忽略 `SIGPIPE` 信号，则写入进程会中断。

注意：在进行读写管道时，对一个管道进行读操作后，`read` 函数返回为 0，有两种意义，一种是管道中无数据并且写入端已经关闭。另一种是管道中无数据，写入端依然存活。这两种情况要根据需要分别处理。

从程序实例 14.1 中可以发现，单独一个进程操作管道是没有任何意义的，管道的应用一般体现在父子进程或者兄弟进程的通信。

如果要建立一个父进程到子进程的数据通道，可以先调用 `pipe` 函数紧接着调用 `fork` 函数，由于子进程自动继承父进程的数据段，则父子进程同时拥有管道的操作权，此时管道的方向取决于用户怎么维护该管道，管道示意图如图 14-3 所示。

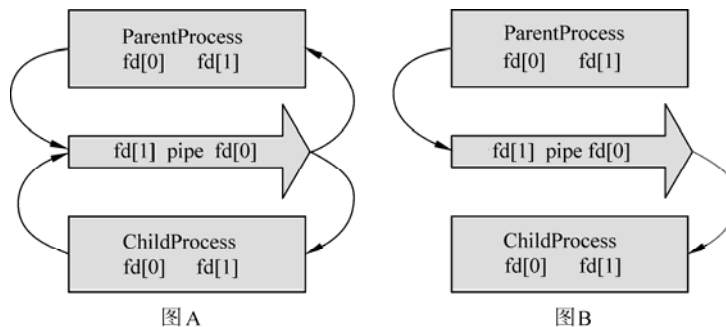


图 14-3 管道示意图

当用户想要一个父进程到子进程的数据通道时，可以在父进程中关闭管道的读出端，相应的在子进程中关闭管道的输出端，如图 14-3 中图 B 所示。相反的，当维护子进程到父进程的数据通道时，在父进程中关闭输出，子进程中关闭读入即可。总之，使用 `pipe` 及 `fork` 组合，可以构造出所有的父进程与子进程，或子进程到兄弟进程的管道。

下面实例演示了使用 `pipe` 以及 `fork` 组合实现父子进程通信。程序中先使用 `pipe` 函数建立管道，使用 `fork` 函数创建子进程。在父子进程中维护管道的数据方向，并在父进程中

向子进程发送消息，在子进程中接收消息并输出到标准输出。

(1) 在 vi 编辑器中编辑该程序如下：

程序清单 14-2 fath_chil.c 管道在父子进程中的应用

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>

#define BUFES PIPE_BUF /* PIPE_BUF 管道默认一次性读写的数据长度*/

int main ( void )
{
    int fd[2];
    char buf[BUFSZ];
    pid_t pid;
    int len;

    if ( (pipe(fd)) < 0 ){ /*创建管道*/
        perror ( "failed to pipe" );
        exit( 1 );
    }

    if ( (pid = fork()) < 0 ){ /* 创建一个子进程 */
        perror ( "failed to fork " );
        exit( 1 );
    }
    else if ( pid > 0 ){
        close ( fd[0] ); /*父进程中关闭管道的读出端*/
        write (fd[1], "hello my son!\n", 14 ); /*父进程向管道写入数据*/
        exit ( 0 );
    }
    else {
        close ( fd[1] ); /*子进程关闭管道的写入端*/
        len = read (fd[0], buf, BUFES ); /*子进程从管道中读出数据*/

        if ( len < 0 ){
            perror ( "process failed when read a pipe " );
            exit( 1 );
        }
        else
            write(STDOUT_FILENO, buf, len); /*输出到标准输出*/

        exit(0);
    }
}
```

(2) 在 shell 中编译该程序如下：


```
$gcc fath_chil.c-o fath_chil
```

(3) 在 shell 中运行该程序。

```
$. / fath_chil
hello my son!
```

程序中使用 `pipe` 函数加 `fork` 组合，实现父进程到子进程的通信。程序在父进程段中关闭了管道的读出端，并相应地在子进程中关闭了管道的输入端，从而实现数据从父进程流向子进程。

管道在兄弟进程间应用时，应该先在父进程中建立管道，然后调用 `fork` 函数创建子进程，在父子进程中维护管道的数据方向。

 **注意：**这里的问题是维护管道的顺序，当父进程创建了管道，只有子进程已经继承了管道后，父进程才可以执行关闭管道的操作。如果在 `fork` 之前已经关闭管道，子进程将不能继承到可以使用的管道的。

下面实例演示了管道在兄弟进程间通信。下例中在父进程中创建管道，并使用 `fork` 函数创建 2 个子进程。在第 1 个子进程中发送消息到第 2 个子进程，第 2 个子进程中读出消息并处理。在父进程中，由于并不使用管道通信，所以什么都不做，直接关闭了管道的两端并退出。

(1) 在 vi 编辑器中编辑该程序。

程序清单 14-3 bro_bro.c 管道在兄弟进程间的应用

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

#define BUFES PIPE_BUF

void err_quit(char * msg){
    perror( msg );
    exit(1);
}

int main ( void )
{
    int fd[2];
    char buf[BUFSZ];          /* 缓冲区 */
    pid_t pid;
    int len;

    if ( (pipe(fd)) < 0 ) /*创建管道*/
        err_quit( "pipe" );
```

```

if ( (pid = fork()) < 0 )                /*创建第一个子进程*/
    err_quit("fork");
else if ( pid == 0 ){                  /*子进程中*/
    close ( fd[0] );                  /*关闭不使用的文件描述符*/
    write(fd[1], "hello brother!", 14 ); /*发送消息*/
    exit(0);
}

if ( (pid = fork()) < 0 )                /*创建第二个子进程*/
    err_quit("fork");
else if ( pid > 0 ){                  /*父进程中*/
    close ( fd[0] );
    close ( fd[1] );
    exit ( 0 );
}
else {                                /*子进程中*/
    close ( fd[1] );                  /*关闭不使用的文件描述符*/
    len = read (fd[0], buf, BUFS );   /*读取消息*/
    write(STDOUT_FILENO, buf, len);
    exit(0);
}
}

```

(2) 在 shell 中编译该程序如下:

```
$gcc bro_bro.c-o bro_bro
```


(3) 在 shell 中运行该程序如下:

```

$./ bro_bro
hello brother!

```

上述程序中父进程分别建立了 2 个子进程, 在子进程 1 中关闭了管道的读出端, 在子进程 2 中关闭了管道的输入端, 并在父进程中关闭了管道的两端。

 **注意:** 程序中父进程在创建第 1 个子进程时并没有关闭管道两端, 而是在创建第 2 个进程时才关闭管道。这是为了在创建第 2 个进程时, 子进程可以继承存活的管道, 而不是一个两端已经关闭的管道。

14.2.4 创建管道的标准库函数

从程序 14-2 和程序 14-3 中可以总结出管道操作的一个流程。父进程中先使用 `pipe` 函数创建管道, 在调用 `fork` 函数创建子进程, 在父子进程中维护管道的数据流向。程序退出时及时关闭管道的两端, 具体流程如图 14-4 所示。

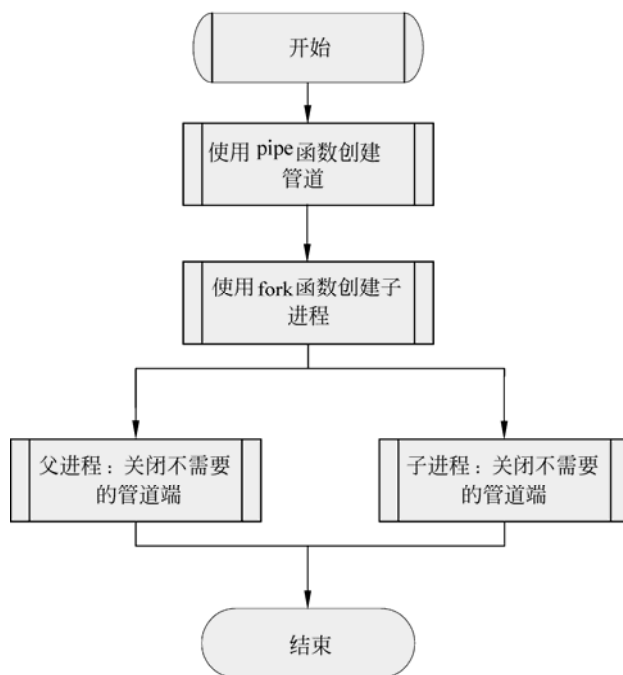


图 14-4 匿名管道的创建流程

管道操作的基本流程为：先创建一个管道，使用 `fork` 创建子进程，在父子进程中关闭不需要的文件描述符使用管道通信，程序结束。由于这是一个比较规范也是比较常用的管道使用模式，所以在 ANSI/ISO C 中将以上操作定义在两个标准的库函数 `popen` 和 `pclose` 中，它们的函数原型是：

```
#include <stdio.h>

FILE *popen( const char * command, const char *mode );
int pclose ( FILE *stream );
```

函数 `popen` 的参数 `command` 是一个在 shell 中可运行的命令字符串的指针，参数 `mode` 是一个字符指针，这个参数只有两种值可以使用，`r` 或者 `w`，分别表示 `popen` 函数的返回值是一个读打开文件指针，还是写打开文件指针。当函数失败时返回值为 `NULL`，并设置出错变量 `errno`。

`popen` 函数先执行创建一个管道，然后调用 `fork` 函数创建子进程，紧接着执行一个 `exec` 函数调用，调用 `/bin/sh -c` 来执行参数 `command` 中的命令字符串，然后函数返回一个标准的 I/O 文件指针。返回的文件指针类型与参数 `mode` 有关，如果参数 `mode` 是 `r` 则文件指针连接到 `command` 命令的标准输出，如果是 `w` 则文件指针连接到 `command` 命令的标准输入。为了关闭 `popen` 函数返回的文件指针，可以调用 `pclose` 函数。`pclose` 函数的参数 `stream` 是一个 `popen` 打开的文件描述符，当函数失败返回 -1。

下面实例演示了使用 `popen` 和 `pclose` 函数实现调用 shell 命令 `cat` 来打印一个文件到显示器的程序。程序中先使用 `popen` 函数为 `cat` 命令创建一条数据管道，并指定数据管道从 `cat` 命令的输出读出数据。在后续的代码中使用 `fgets` 函数读出数据，并将数据显示到标准

输出中。

(1) 在 vi 编辑器中编辑该程序如下：

程序清单 14-4 recat.c 使用 popen 和 pclose 函数创建管道

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>

#define BUFES PIPE_BUF

int main ( void )
{
    FILE *fp;
    char * cmd = "cat file1";           /*shell 命令*/
    char * buf[BUFSZ];

    if ((fp = popen( cmd , "r"))==NULL )    /*创建子进程到父进程的数据管道*/
    {
        perror ( " failed to popen " ) ;
        exit ( 1 ) ;
    }

    while ((fgets(buf, BUFSZ, fp))!= NULL ) /*读出管道的数据*/
        printf ( "%s", buf );


    pclose ( fp ) ;                      /*关闭管道*/
    exit ( 0 ) ;
}
```

(2) 在 shell 中编译该程序如下：

```
$gcc recat.c-o recat
```

(3) 在 shell 中运行该程序如下：

```
$. / recat
Used the popen and pclose function to create a pipe !!!
```

 **说明：**在程序 14-4 recat.c 中，使用 popen 和 pclose 函数创建管道并关闭管道，使用 gets 函数从管道输出端读取数据并打印到标准输出中，使用 popen 和 pclose 可以更简洁地控制管道，而无需那些繁杂的代码。当然这样做的结果是降低了程序员对管道的控制能力。

例如，popen 函数返回的是文件指针，所以，在管道读写时就不能使用低级的 read 和 write I/O 调用了，只能使用基于文件指针的 I/O 函数，并且在 popen 函数中调用 exec 函数来复写子进程，这也是要花费一段运行时间的。

14.3 FIFO 管道

FIFO 也称为有名管道，它是一种文件类型，在文件系统中可以看到。程序中可以查看文件 `stat` 结构中 `st_mode` 成员的值来判断文件是否是 FIFO 文件。创建一个 FIFO 文件类似于创建文件，FIFO 文件就像普通文件一样。本小节将介绍 FIFO 管道。

14.3.1 FIFO 的概念

在本章 14.2.2 小节详细说明了匿名管道的缺点以及限制条件，在 FIFO 中可以很好地解决在无关进程间数据交换的要求，并且由于它们是存在于文件系统上的，这也提供了一种比匿名管道更持久稳定的通信办法。

FIFO 的通信方式类似于在进程中使用文件来传输数据，只不过 FIFO 类型文件同时具有管道的特性。在数据读出时，FIFO 管道中同时清除数据。在 shell 中 `mkfifo` 命令可以建立有名管道，下面通过一个实例来帮助读者理解 FIFO。`mkfifo` 命令的帮助手册如下所示：

```
mkfifo [option] name...
```

其中 `option` 选项中可以选要创建 FIFO 的模式，使用形式为 `-m mode`，这里 `mode` 指出将要创建 FIFO 的八进制模式，注意，这里新创建的 FIFO 会像普通文件一样受到创建进程的 `umask` 修正。在 shell 中输入命令如下：

```
$mkfifo -m 600 fifocat
$cat < fifocat
$./recat >fifocat
$./recat >fifocat

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <limits.h>

#define BUFES PIPE_BUF

int main ( void )
{
    FILE *fp;
    char * cmd = "cat file1"; /*shell 命令*/
    char * buf[BUFSZ];
    ...
    ...
}
```

```

...
    pclose ( fp ) ;      /*关闭管道*/
    exit (0) ;
}
$ _

```

以上实例使用系统命令 `mkfifo` 创建 FIFO 类型文件 `fifocat`, 并通过 14.2.4 节的程序 `recat` 来读取文件 `recat.c`, 将程序的标准输出从定向到 `fifocat` 中, 再使用命令 `cat` 从 `fifocat` 读出数据。

14.3.2 创建 FIFO

创建一个 FIFO 文件类似于创建文件, FIFO 文件就像普通文件一样, 也是可以经过路径名来访问的。相应文件 `stat` 结构的域 `st_mode` 的编码指明了文件是否是 FIFO 类型。FIFO 管道通过函数 `mkfifo` 创建, 函数原型如下:

```

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

int mkfifo( const char * filename, mode_t mode );

```

`mkfifo` 函数中参数 `mode` 指定 FIFO 的读写权限, 新创建 FIFO 的用户 ID 和组 ID 规则域 `open` 函数相同。参数 `filename` 指定新创建 FIFO 的文件名称。函数如果成功返回 0, 出错返回 -1, 并更改 `errno` 的值。`errno` 有可能出现的值为: `EACCESS`、`EEXIST`、`ENAMETOOLONG`、`ENOENT`、`ENOSPE`、`ENOTDIR` 和 `EROFS`。

下面实例演示了如何使用 `mkfifo` 函数来创建一个 FIFO。程序中从程序的命令行参数中得到一个文件名, 然后使用 `mkfifo` 函数创建 FIFO 文件。新创建的 FIFO 只具有读写权限。由于 FIFO 文件的特性, 所以它被隐性地规定不具有执行权限。

(1) 在 vi 编辑器中编辑该程序如下:

程序清单 14-5 create_fifo.c 使用 `mkfifo` 函数创建 FIFO 管道

```

#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
    mode_t mode = 0666;    /*新创建的 FIFO 模式*/

    if ( argc != 2 ){
        /*向用户提示程序使用帮助*/
        printf("USEMSG: create_fifo {fifoname}\n");
        exit (1);
    }
}

```

```

/* 使用 mkfifo 函数创建一个 FIFO 管道*/
if ( ( mkfifo (argv[1], mode )) < 0 ) {
    perror ( "failed to mkfifo" );
    exit ( 1 );
}
else
    printf ("you successfully create a FIFO name is : %s\n", argv[1]);
/* 输出 FIFO 文件的名称 */

exit (0);
}

```

(2) 在 shell 中编译该程序如下:

```
$gcc create_fifo.c-o create_fifo
```

(3) 在 shell 中运行该程序如下:

```

$./ create_fifo
USEMSG: create_fifo {fifoname}

```

输入正确的命令符。

```

$./ create_fifo fifol
you successfully create a FIFO name is :fifol
$./ create_fifo fifol
mkfifo: File exists

```

上述程序使用 `mkfifo` 函数创建一个 FIFO，名字是基于用户的输入文件名，可以看到当要创建一个已经存在的 FIFO 时，程序会产生一个 `EEXIST` 的异常，相对应该异常，`perror` 函数打印了相应的帮助信息为 `mkfifo: File exists`。


14.3.3 FIFO 的读写操作

一般的 I/O (`open close read write unlink`) 函数都可以用于 FIFO 文件，需要注意的是，在使用 `open` 函数打开一个 FIFO 文件时，`open` 函数参数 `flag` 标志位的 `O_NONBLOCK` 标志，它关系到函数的返回状态。详细说明如表 14-2 所示。

表 14-2 `open` 函数的 `flag(O_NONBLOCK)` 详细说明

O_NONBLOCK 标志	详 细 说 明
置位	只读 <code>open</code> 立即返回。当只写 <code>open</code> 时，如果没有进程为读打开 FIFO，则返回 -1，并置 <code>errno</code> 值为 <code>ENXIO</code>
不置位	<code>open</code> 视情况阻塞。只读 <code>open</code> 要阻塞到有进程为写打开 FIFO，只写 <code>open</code> 要阻塞到有进程为读打开 FIFO

FIFO 的写操作规则类似于匿名管道的写操作规则，当没有进程为读打开 FIFO，调用 `write` 函数来进行写操作会产生信号 `SIGPIPE`，则信号可以被捕捉或者完全忽略。

 **注意：**当 FIFO 的所有写进程都已经关闭，则为 FIFO 的读进程产生一个文件结束符。

FIFO 的出现，极好地解决了系统在实际应用中产生的大量的中间临时文件的问题。FIFO 可以被 shell 调用使数据从一个进程到另一个进程，系统不必为该中间通道去烦恼清理不必要的垃圾，或者去释放该通道的资源，它可以被留做后来的进程使用。并且规避了匿名管道在作用域的限制，可应用于不相关的进程之间。

下面实例演示了使用 FIFO 来进行两个进程间通信的例子。在程序 `write_fifo.c` 中打开一个名为 `fifo1` 的 FIFO 文件，并分 10 次向这个 FIFO 中写入数据。在程序 `read_fifo.c` 中先打开 `fifo1` 文件，读取里面的数据并输出到标准输出中。

在 vi 编辑器中编辑该程序如下：

程序清单 14-6 `write_fifo.c` 使用 FIFO 进行通信

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <limits.h>

#define BUFES PIPE_BUF

int main(void)
{
    int fd ;
    int n, i ;
    char buf[BUFES];
    time_t tp;

    printf("I am %d\n",getpid());          /*说明进程的 ID*/

    if((fd=open("fifo1",O_WRONLY))<0){ /*以写打开一个 FIFO1*/
        perror("open");
        exit(1);
    }

    for ( i=0 ; i<10; i++){                /*循环 10 次向 FIFO 中写入数据*/
        time(&tp);                          /*取系统当前时间*/

        /*使用 sprintf 函数向 buf 中格式化写入进程 ID 和时间值*/
        n=sprintf(buf,"write_fifo %d sends %s",getpid(),ctime(&tp));
        printf("Send msg:%s\n",buf);
    }
}
```

```

        if((write(fd, buf, n+1))<0) { /*写入到 FIFO 中*/
            perror("write");
            close(fd);                /* 关闭 FIFO 文件 */
            exit(1);
        }
        sleep(3);                    /*进程睡眠 3 秒*/
    }

    close(fd);                      /* 关闭 FIFO 文件 */
    exit(0);
}

```

程序中使用 `open` 函数打开一个名为 `fifo1` 的 FIFO 管道，并分 10 次向 `fifo1` 中写入字符串，其中的数据有当前进程 ID 以及写入时的系统时间。并把这个数据串输出到标准输出，然后程序自动睡眠 3 秒。

(1) 在 vi 编辑器中编辑该程序如下：

程序清单 14-7 read_fifo.c 使用 FIFO 进行通信

```

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <fcntl.h>
#include <unistd.h>

#define BUFES PIPE_BUF

int main(void)
{
    int fd;
    int len;
    char buf[BUFES];
    mode_t mode = 0666;                /* FIFO 文件的权限 */

    if((fd=open("fifo1",O_RDONLY))<0) /* 打开 FIFO 文件 */
    {
        perror("open");
        exit(1);
    }

    while((len=read(fd,buf, BUFES))>0) /* 开始进行通信 */
        printf("read_fifo read: %s",buf);
}

```

```
close(fd); /* 关闭 FIFO 文件 */
    exit(0);
}
```

程序中使用 `open` 函数以读方式打开一个名为 `fifo1` 的 FIFO 管道，并循环读出管道的数据，这里使用 `while` 循环的作用就是确保数据可以全部读出，因为在读 FIFO 管道数据时，默认的是一次性读取 `PIPE_BUF` 个字节，当管道中数据多于 `PIPE_BUF` 个字节时，一次性读出 `PIPE_BUF-1` 个字节，然后 `read` 函数返回，再打印数据到标准输出。

(2) 在 shell 中分别编译上述两个程序如下：

```
$gcc write_fifo.c-o write_fifo
$gcc read_fifo.c-o read_fifo
```

(3) 在 shell 中使用 `mkfifo` 创建程序中将要用到的 FIFO 管道。

```
$mkfifo -m 666 fifo1
```

(4) 打开两个 shell 分别运行程序 `write_fifo` 和程序 `read_fifo`。一个 shell 中输入如下：

```
./write_fifo
i am 3708
Send msg:write_fifo 3708 sends Thu Apr 17 18:26:01 2008
Send msg:write_fifo 3708 sends Thu Apr 17 18:26:04 2008
Send msg:write_fifo 3708 sends Thu Apr 17 18:26:07 2008
Send msg:write_fifo 3708 sends Thu Apr 17 18:26:10 2008
Send msg:write_fifo 3708 sends Thu Apr 17 18:26:13 2008
Send msg:write_fifo 3708 sends Thu Apr 17 18:26:16 2008
Send msg:write_fifo 3708 sends Thu Apr 17 18:26:19 2008
Send msg:write_fifo 3708 sends Thu Apr 17 18:26:22 2008
Send msg:write_fifo 3708 sends Thu Apr 17 18:26:25 2008
Send msg:write_fifo 3708 sends Thu Apr 17 18:26:28 2008
```

另一个 shell 中输入如下：

```
./read_fifo
read_fifo read: write_fifo 3708 sends Thu Apr 17 18:26:01 2008
read_fifo read: write_fifo 3708 sends Thu Apr 17 18:26:04 2008
read_fifo read: write_fifo 3708 sends Thu Apr 17 18:26:07 2008
read_fifo read: write_fifo 3708 sends Thu Apr 17 18:26:10 2008
read_fifo read: write_fifo 3708 sends Thu Apr 17 18:26:13 2008
read_fifo read: write_fifo 3708 sends Thu Apr 17 18:26:16 2008
read_fifo read: write_fifo 3708 sends Thu Apr 17 18:26:19 2008
read_fifo read: write_fifo 3708 sends Thu Apr 17 18:26:22 2008
read_fifo read: write_fifo 3708 sends Thu Apr 17 18:26:25 2008
read_fifo read: write_fifo 3708 sends Thu Apr 17 18:26:28 2008
```

上述例子可以扩展成客户端与服务器通信的实例，`write_fifo` 的作用类似于客户端，可以打开多个客户端向一个服务器发送请求信息，`read_fifo` 类似于服务器，它适时监控着 FIFO 的读出端，当有数据时，读出并进行处理，但是有一个关键的问题是，每一个客户端

必须预先知道服务器提供的 FIFO 接口，如图 14-5 所示。

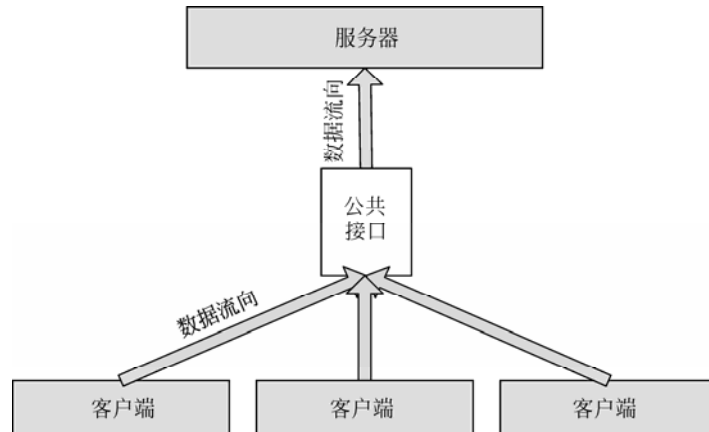



图 14-5 FIFO 在客户端与服务器通信的应用 1

14.3.4 FIFO 的缺点

当然 FIFO 也有它的局限性，如图 14-6 所示。客户端可以发请求到服务器，但前提是要知道一个公共的 FIFO 通道，对于实现服务器回传应答到客户端的问题，可以通过为每一个客户端创建一个专用的 FIFO，来实现回传应答。但也有不足，服务器会同时应答成千上万个客户端，创建如此多的 FIFO 是否会系统负载过大，相应的如何判断客户端是否因意外而崩溃成为难题，或者客户端不读取应答直接退出，所以服务器必须处理 SIGPIPE 信号，并做相应处理。

说明：在服务器端打开公共 FIFO 的时候，如果仅以读打开，则当所有的客户端都退出时，服务器端会读取到文件结束符。这个问题的解决办法是服务器以读写打开公共 FIFO，如图 14-6 所示。服务器与客户端如何实现互相通信。

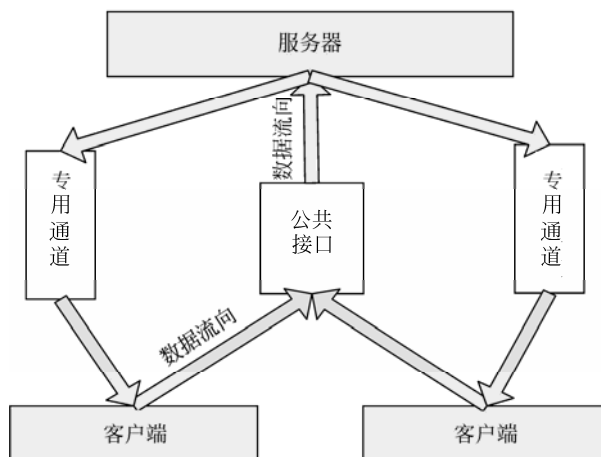


图 14-6 FIFO 在客户端与服务器通信的应用 2

14.4 System V IPC/POSIX IPC

System V IPC 包括三种进程通信方式，即消息队列、信号量以及共享存储器，这是一种比较古老的方式，在最近的版本中已逐渐地被 POSIX IPC 而取代。当然，两者之间还是有着密切关系的，实现的道理也还是一样。

消息队列、信号量以及共享存储器这三种 IPC 的几种结构有时又称 IPC 对象，它不同于前面提到的管道和 FIFO。管道和 FIFO 是基于文件系统的，例如，可在文件系统中看到某一个 FIFO 类型文件，而 System V IPC 是基于系统内核的，可以使用命令 `ipcs` 来查看系统当前的 IPC 对象的状态，本节将详细介绍它们各自的特点以及使用实例。

14.4.1 IPC 对象的概念

由于消息队列、信号量以及共享存储器三种 IPC 有很多相类似的特性，所以本节将对它们的特性，以及相关知识做一些详细的介绍，IPC 对象示意图如图 14-7 所示。

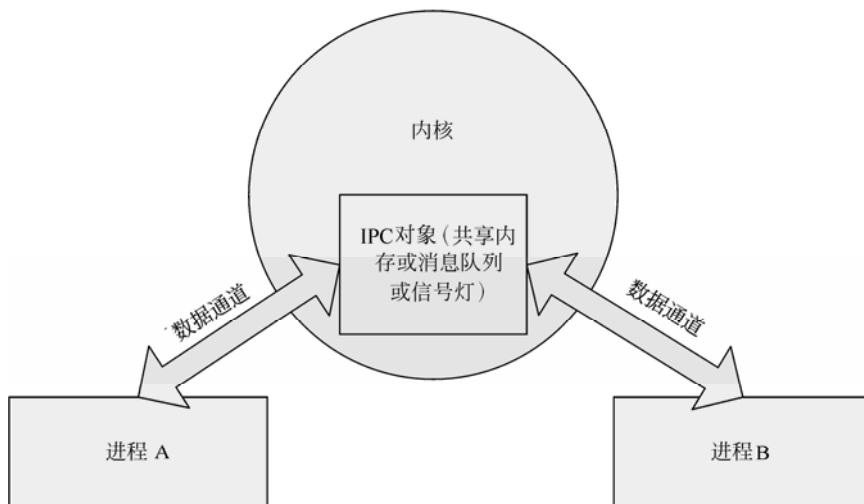


图 14-7 IPC 对象的应用示意图

如图 14-7 所示，IPC 对象是活动在内核级别的一种进程间通信的工具。存在的 IPC 对象通过它的标识符来引用和访问，这个标识符是一个非负整数，它唯一的标识了一个 IPC 对象，这个 IPC 对象可以是消息队列或信号量或共享存储器中的任意一种类型。

在 Linux 系统中标识符被声明成整数，所以可能存在的最大标识符为 65535。这里标识符与文件描述符有所不同，使用 `open` 函数打开一个文件时，返回的文件描述符的值为当前进程最小可用的文件描述符数组的下标。IPC 对象删除或创建时相应的标识符的值会不断增加到最大的值，归零循环分配使用。

IPC 的标识符只解决了内部访问一个 IPC 对象的问题，如何让多个进程都访问某一个

特定的 IPC 对象还需要一个外部键（key），每一个 IPC 对象都与一个键相关联。这样就解决了多进程在一个 IPC 对象上汇合的问题。

创建一个 IPC 对象时需要指定一个键值，类型为 `key_t`，在 `<sys/types.h>` 中定义为一个长整型。键值到标识符的转换是由系统内核来维护的。当有了一个 IPC 对象的键值，如何让多个进程知道这个键，可以有多种实现的办法。

- ❑ 可以使用文件来做中间的通道，创建 IPC 对象进程，使用键 `IPC_PRIVATE` 成功建立 IPC 对象之后，将返回的标识符存储在一个文件中。其他进程通过读取这个标识符来引用 IPC 对象通信。
- ❑ 定义一个多个进程都认可的键，每个进程使用这个键来引用 IPC 对象，值得注意的是，创建 IPC 对象的进程中，创建 IPC 对象时如果该键值已经与一个 IPC 对象结合，则应该删除该 IPC 对象，再创建一个新的 IPC 对象。
- ❑ 多进程通信中，对于指定键引用一个 IPC 对象而言，可能不具有拓展性，并且在该键值已经被一个 IPC 对象结合的情况下。所以必须删除这个存在对象之后再建立一个新的。这有可能影响到其他正在使用这个对象的进程。函数 `ftok` 可以在一定程度上解决这个问题，

函数 `ftok` 可以使用两个参数生成一个键值，函数原型如下：

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

key_t ftok( const char *path, int id );
```

函数中参数 `path` 是一个文件名。函数中进行的操作是，取该文件的 `stat` 结构的 `st_dev` 成员和 `st_ino` 成员的部分值，然后与参数 `ID` 的第八位结合起来生成一个键值。由于只是使用 `st_dev` 和 `st_ino` 的部分值，所以会丢失信息，不排除两个不同文件使用同一个 `ID`，得到同样键值的情况。

系统为每一个 IPC 对象保存一个 `ipc_perm` 结构体，该结构说明了 IPC 对象的权限和所有者，每一个版本的内核各有不同的 `ipc_perm` 结构成员。若要查看详细的定义请参阅文件 `<sys/ipc.h>`。

```
struct ipc_perm {
    key_t key;
    uid_t uid;
    gid_t gid;
    uid_t cuid;
    gid_t cgid;
    unsigned short mode;
    unsigned short seq;
};
```

每一种版本的 `ipc_perm` 结构体定义至少要包含上述几个域。当调用 IPC 对象的创建函数（`semget` `msgget` `shmget`）时，会对 `ipc_perm` 结构的每一个域赋值。在后续的操作中如需修改这几个域则调用相应的控制函数（`msgctl` `semctl` `shmctl`）。


 **注意：**只有超级用户或者创建 IPC 对象的进程有权改变 ipc_perm 结构的值。结构中的 mode 域类似于文件的 stat 结构的 mode 域，但是不可以有执行权限。mode 值描述如表 14-3 所示。

表 14-3 ipc_perm 的 mode 详解表

操作者	读	写（更改 更新）	操作者	读	写（更改 更新）
用户	0400	0200	其他	0004	0002
组	0040	0020			

14.4.2 IPC 对象的问题

IPC 对象所存在的问题主要集中在以下几点：

过于繁杂的编程接口，比起使用其他通信方式，IPC 所要求的代码量要明显增多。

- ❑ IPC 不使用通用的文件系统，这也是饱受指责的原因。所以不能使用标准 I/O 操作函数来读写 IPC 对象。为此不得不新增加一些函数来支持必要的一些操作（例如 msgget msgrev msgctl 等）并且对于不同类型的 IPC 对象都有一系列特定的操作函数。由于 IPC 不使用文件描述符，所以不能使用多路 I/O 监控函数 select 及 poll 函数来操作 IPC 对象。
- ❑ 缺少的资源回收机制。由于 IPC 对象在使用过程中并不保存引用计数，所以当出现一个进程创建了 IPC 对象然后退出时，则这个对象只有在出现后面几种情况才会被释放或者删除，即由某一个进程读出消息，或者 IPC 的所有者或超级用户删除了这个对象。这也是 IPC 相对于管道或 FIFO 所欠缺的资源回收机制。

14.4.3 IPC 对象系统命令

在 shell 下可以使用一些命令来操作 IPC 对象，下面通过几个实际的例子来帮助理解 IPC 对象，使用 ipcs 可以显示 IPC 的状态。在 shell 中输入：

```
$ipcs -a

----- Shared Memory Segments -----
key          shmid      owner      perms      bytes      nattch     status
0x00000000  65536      root       600        393216     2          dest
0x00000000  2654209    root       666        4096       0
0x00000000  2752516    root       666        4096       0

----- Semaphore Arrays -----
key          semid      owner      perms      nsems
0x00000000  294911     root       666        1

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

注意 `ipcs` 输出的信息中的 `key` 以及 `shmid`, `key` 标识的是 IPC 对象的外键, `shmid` 标识的 IPC 对象的标识符。owner 标识的是 IPC 所属的用户, `perms` 标识权限。可以使用 `ipcrm` 命令来删除一个 IPC 对象, 使用实例如下。在 shell 中输入:

```
$ipcrm -m 2752516
$ipcs -a
```

```
----- Shared Memory Segments -----
key          shmid      owner      perms      bytes      nattch     status
0x00000000   65536       root       600        393216     2          dest
0x00000000   2654209     root       666        4096       0
----- Semaphore Arrays -----
key          semid       owner      perms      nsems
0x00000000   294911     root       666        1
----- Message Queues -----
key          msqid       owner      perms      used-bytes  messages
```

在应用中, 如果使用 `kill` 命令删除程序后, 发现系统资源例如内存的使用量仍然很高, 则应检查系统 IPC 状态, 并使用 `ipcrm` 命令删除不使用的 IPC。

14.5 共享内存

共享内存从字面意义解释就是多个进程可以把一段内存映射到自己的进程空间, 以此来实现数据的共享以及传输, 这也是所有进程间通信方式中最快的一种。共享内存是存在于内核级别的一种资源, 在 shell 中可以使用 `ipcs` 命令来查看当前系统 IPC 中的状态, 在文件系统中 `/proc` 目录下有对其描述的相应文件。

14.5.1 共享内存的概念

在系统内核为一个进程分配内存地址时, 通过分页机制可以让一个进程的物理地址不连续, 同时也可以让一段内存同时分配给不同的进程。共享内存机制就是通过该原理来实现的, 共享内存机制只是提供数据的传送, 如何控制服务器端和客户端的读写操作互斥, 这就需要一些其他的辅助工具, 例如, 记录锁概念, 如图 14-8 所示, 描述了多进程如何使用共享内存通信。

如图 14-8 所示。箭头方向描述了进程地址空间映射到系统内存地址的位置。对于每一个共享存储段, 内核会为其维护一个 `shmid_ds` 类型的结构体 (`shmid_ds` 结构体定义在头文件 `<sys/shm.h>` 中)。`shmid_ds` 结构体定义如下:

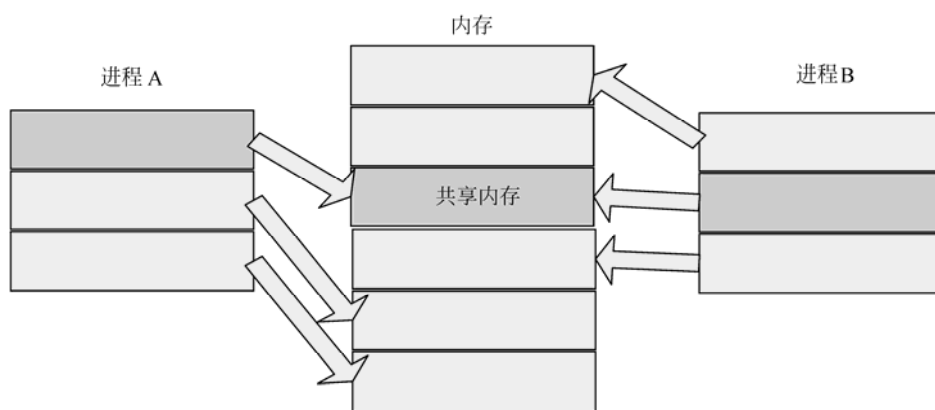


图 14-8 共享内存示意图

```

struct shmid_ds{
    struct ipc_perm shm_perm;    /
    size_t      shm_segsz;
    pid_t      shm_lpid;
    pid_t      shm_cpid;
    shmatt_t    shm_nattch;
    time_t      shm_atime;
    time_t      shm_dtime;
    time_t      shm_ctime;
    .....
    .....
    .....
};

```

结构体 `shmid_ds` 会根据不同的系统内核版本而略有不同，并且在不同的系统中会对共享存储段的大小有限制，在应用时请查询相应的系统手册。

14.5.2 共享内存的创建

共享内存是存在于内核级别的一种资源，在 `shell` 中可以使用 `ipcs` 命令来查看当前系统 `IPC` 中的状态，在文件系统 `/proc` 目录下有对其描述的相应文件。函数 `shmget` 可以创建或打开一块共享内存区。函数原型如下：

```

#include <sys/shm.h>

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

```

函数中参数 `key` 用来变换成一个标识符，而且每一个 `IPC` 对象与一个 `key` 相对应。当新建一个共享内存段时，`size` 参数为要请求的内存长度（以字节为单位）。

注意：内核是以页为单位分配内存，当 `size` 参数的值不是系统内存页长的整数倍时，系统会分配给进程最小的可以满足 `size` 长的页数，但是最后一页的剩余部分内存是不可用的。

当打开一个内存段时，参数 `size` 的值为 0。参数 `flag` 中的相应权限位初始化 `ipc_perm` 结构体中的 `mode` 域。同时参数 `flag` 是函数行为参数，它指定一些当函数遇到阻塞或其他情况时应做出的反应。`shmid_ds` 结构初始化如表 14-4 所示。

表 14-4 shmid_ds 的初始化

shmid_ds 结构数据	初 值	shmid_ds 结构数据	初 值
shm_lpid	0	shm_dtime	0
shm_nattach	0	shm_ctime	系统当前值
shm_atime	0	shm_segsz	参数 <code>size</code>

下面实例演示了使用 `shmget` 函数创建一块共享内存。程序中在调用 `shmget` 函数时指定 `key` 参数值为 `IPC_PRIVATE`，这个参数的意义是创建一个新的共享内存区，当创建成功后使用 `shell` 命令 `ipcs` 来显示目前系统下共享内存的状态。命令参数 `-m` 为只显示共享内存的状态。

(1) 在 `vi` 编辑器中编辑该程序如下：

程序清单 14-8 create_shm.c 使用 `shmget` 函数创建共享内存

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

#define BUFSZ 4096

int main ( void )
{
    int shm_id;    /*共享内存标识符*/

    shm_id=shmget(IPC_PRIVATE, BUFSZ, 0666 ) ;
    if (shm_id < 0 ) { /*创建共享内存*/
        perror( "shmget" ) ;
        exit ( 1 );
    }

    printf ( "successfully created segment : %d \n", shm_id ) ;
    system( "ipcs -m");    /*调用 ipcs 命令查看 IPC*/

    exit( 0 );
}
```

(2) 在 `shell` 中编译该程序如下：

```
$gcc create_shm.c-o create_shm
```

(3) 在 `shell` 中运行该程序如下：

```
$/ create_shm
```

```

successfully created segment : 2752516

----- Shared Memory Segments -----
key          shmid      owner    perms    bytes    nattch   status
0x00000000  65536      root     600      393216   2        dest
0x00000000  2654209   root     666      4096     0
0x0056a4d5  2686978   root     600      488      1
0x0056a4d6  2719747   root     600      131072   1
0x00000000  2752516   root     666      4096     0

```

上述程序中使用 `shmget` 函数来创建一段共享内存，并在结束前调用了系统 `shell` 命令 `ipcs -m` 来查看当前系统 IPC 状态。

14.5.3 共享内存的操作

由于共享内存这一特殊的资源类型，使它不同于普通的文件，因此，系统需要为其提供专有的操作函数，而这无疑增加了程序员开发的难度（需要记忆额外的专有函数）。使用函数 `shmctl` 可以对共享内存段进行多种操作，其函数原型如下：

```

#include <sys/shm.h>

int shmctl( int shm_id, int cmd, struct shmid_ds *buf );

```

函数中参数 `sh_mid` 为所要操作的共享内存段的标识符，`struct shmid_ds` 型指针参数 `buf` 的作用与参数 `cmd` 的值相关，参数 `cmd` 指明了所要进行的操作，其解释如表 14-5 所示。

表 14-5 shmctl函数中参数 cmd详解

cmd 的值	意 义
IPC_STAT	取 <code>shm_id</code> 所指向内存共享段的 <code>shmid_ds</code> 结构，对参数 <code>buf</code> 指向的结构赋值
IPC_SET	使用 <code>buf</code> 指向的结构对 <code>sh_mid</code> 段的相关结构赋值，只对以下几个域有作用， <code>shm_perm</code> . <code>uid shm_perm.gid</code> 以及 <code>shm_perm.mode</code> 注意此命令只有具备以下条件的进程才可以请求： 1. 进程的用户 ID 等于 <code>shm_perm.cuid</code> 或者等于 <code>shm_perm.uid</code> 2. 超级用户特权进程
IPC_RMID	删除 <code>shm_id</code> 所指向的共享内存段，只有当 <code>shmid_ds</code> 结构的 <code>shm_nattch</code> 域为零时，才会真正执行删除命令，否则不会删除该段 注意此命令的请求规则与 <code>IPC_SET</code> 命令相同
SHM_LOCK	锁定共享内存段在内存，此命令只能由超级用户请求
SHM_UNLOCK	对共享内存段解锁，此命令只能由超级用户请求

使用函数 `shmat` 将一个存在的共享内存段连接到本进程空间，其函数原型如下：

```


#include <sys/shm.h>

```



```
void *shmat( int shm_id, const void *addr, int flag );
```

函数中参数 `shm_id` 指定要引入的共享内存，参数 `addr` 与 `flag` 组合说明要引入的地址值，通常只有 2 种用法，`addr` 为 0，表明让内核来决定第 1 个可以引入的位置。`addr` 非零，并且 `flag` 中指定 `SHM_RND`，则此段引入到 `addr` 所指向的位置（此操作不推荐使用，因为不会只对一种硬件上运行应用程序，为了程序的通用性推荐使用第 1 种方法），在 `flag` 参数中可以指定要引入的方式（读写方式指定）。

 **说明：**函数成功执行返回值为实际引入的地址，失败返回-1。`shmat` 函数成功执行会将 `shm_id` 段的 `shmid_ds` 结构的 `shm_nattch` 计数器的值加 1。

当对共享内存段操作结束时，应调用 `shmdt` 函数，作用是将指定的共享内存段从当前进程空间中脱离出去。函数原型如下：

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

int shmdt( void *addr);
```

参数 `addr` 是调用 `shmat` 函数的返回值，函数执行成功返回 0，并将该共享内存的 `shmid_ds` 结构的 `shm_nattch` 计数器减 1，失败返回-1。

下面实例演示了操作共享内存段的流程。程序的开始部分先检测用户是否有输入，如出错则打印该命令的使用帮助。接下来从命令行读取将要引入的共享内存 ID，使用 `shmat` 函数引入该共享内存，并在分离该内存之前睡眠 3 秒以方便查看系统 IPC 状态。

(1) 在 vi 编辑器中编辑该程序如下：

程序清单 14-9 opr_shm.c 操作共享内存段

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

int main ( int argc, char *argv[] )
{
    int shm_id ;
    char * shm_buf;

    if ( argc != 2 ){ /* 命令行参数错误 */
        printf ( "USAGE: atshm <identifier>" ); /*打印帮助消息*/
        exit (1 );
    }

    shm_id = atoi(argv[1]); /*得到要引入的共享内存段*/

    /*引入共享内存段，由内核选择要引入的位置*/
    if ( (shm_buf = shmat( shm_id, 0, 0)) < (char *) 0 ){
        perror ( "shmat" );
    }
}
```

```

        exit (1);
    }

    printf ( " segment attached at %p\n", shm_buf );    /*输出导入的位置*/
    system("ipcs -m");

    sleep(3);                                          /* 休眠 */

    if ( (shmdt(shm_buf)) < 0 ) {    /*与导入的共享内存段分离*/
        perror ( "shmdt");
        exit(1);
    }

    printf ( "segment detached \n" );
    system ( "ipcs -m " );                /*再次查看系统 IPC 状态*/

    exit ( 0 );
}

```

(2) 在 shell 中编译该程序如下:

```
$gcc opr_shm.c-o opr_shm
```

(3) 在 shell 中运行该程序如下:

```

$./ opr_shm 2752516

segment attached at 0xb7f29000

----- Shared Memory Segments -----
key          shmid      owner      perms      bytes      nattch     status
0x00000000   65536       root       600        393216     2          dest
0x00000000   2654209      root       666        4096       0
0x0056a4d5   2686978      root       600        488        1
0x0056a4d6   2719747      root       600        131072     1
0x00000000   2752516      root       666        4096       1

segment detached

----- Shared Memory Segments -----
key          shmid      owner      perms      bytes      nattch     status
0x00000000   65536       root       600        393216     2          dest
0x00000000   2654209      root       666        4096       0
0x0056a4d5   2686978      root       600        488        1
0x0056a4d6   2719747      root       600        131072     1
0x00000000   2752516      root       666        4096       0

```

上述程序中从命令行中读取所要引入的共享内存 ID，并使用 `shmat` 函数引入该内存到当前的进程空间中。注意在使用 `shmat` 函数时，将参数 `addr` 的值设为 0，所表达的意义是由内核来决定该共享内存存在当前进程中的位置。由于在编程的过程中，很少会针对某一个特定的硬件或系统编程，所以由内核决定引入位置也就是 `shmat` 推荐的使用方式。在导入

后使用 shell 命令 `ipcs -m` 来显示当前的系统 IPC 的状态，可以看出输出信息中 `nattch` 字段为该共享内存时的引用值，最后使用 `shmdt` 函数分离该共享内存并打印系统 IPC 的状态。

14.5.4 共享内存使用注意事项

共享内存相比其他方式有着更方便的数据控制能力，数据在读写过程中会更透明。当成功导入一块共享内存后，它只是相当于一个字符串指针来指向一块内存，在当前进程下用户可以随意的访问。缺点是，数据写入进程或数据读出进程中，需要附加的数据结构控制，共享内存通信数据结构示意如图 14-9 所示。

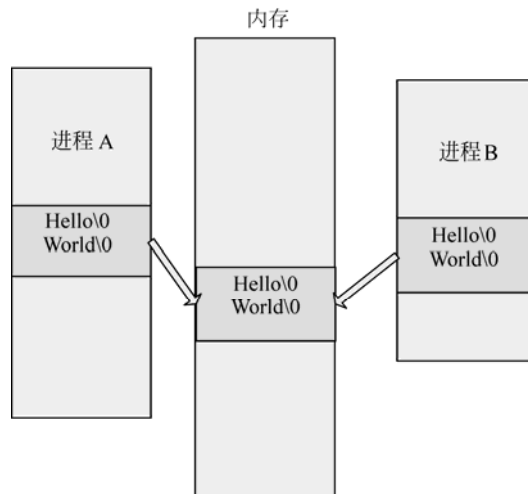



图 14-9 共享内存通信数据结构示意

说明：图中两个进程同时遵循一定的规则来读写该内存。同时，在多进程同步或互斥上也需要附加的代码来辅助共享内存机制。

在共享内存段中都是以字符串的默认结束符为一条信息的结尾。每个进程在读写时都遵守这个规则，就不会破坏数据的完整性。

14.6 信号量

信号量的原理是一种数据操作锁的概念，它本身不具备数据交换的功能，而是通过控制其他的通信资源（文件，外部设备等）来实现进程间通信。信号量本身不具备数据传输的功能，其只是一种外部资源的标识。本小节将深入介绍信号量的操作。

14.6.1 信号量的概念

信号量本身不具备数据传输的功能，它只是一种外部资源的标识，通过该标识可以判

断外部资源是否可用，信号量在此过程中负责数据操作的互斥、同步等功能。

当请求一个使用信号量来表示的资源时，进程需要先读取信号量的值，以判断相应的资源是否可用。当信号量的值大于 0 时，表明有资源可以请求。等于 0 时，说明现在无可用资源，所以进程会进入睡眠状态直至有可用资源时。

当进程不再使用一个信号量控制的共享资源时，此信号量的值增 1，对信号量的值进行增减操作均为原子操作，这是由于信号量主要的作用是维护资源的互斥或多进程的同步访问。而在信号量的创建以及初始化时，不能保证操作均为原子。

同其他的 IPC 对象一样，内核对每一个信号量集都会设置一个 `shmid_ds` 结构（详细介绍见 14.3.4），同时用一个无名结构来标识一个信号量。简要定义如下：

```
struct {
    unsigned short semval;
    pid_t          sempid;
    unsigned short semncnt;
    unsigned short semzcnt;
    ...
    ...
}
```

14.6.2 信号量的创建

同共享内存一样，系统中同样需要为信号量集定制一系列专有的操作函数（`semget`，`semctl` 等）。系统命令 `ipcs` 可查看当前的系统 IPC 的状态，在命令后使用 `-s` 参数。使用函数 `semget` 可以创建或者获得一个信号量集 ID，函数原型如下：

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

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

函数中参数 `key` 用来变换成一个标识符，每一个 IPC 对象与一个 `key` 相对应。当新建一个共享内存段时，使用参数 `flag` 的相应权限位对 `ipc_perm` 结构中的 `mode` 域赋值，对相应信号量集的 `shmid_ds` 初始化的值如表 14-6 所示。

表 14-6 `shmid_ds` 结构初始化值表

ipc_perm 结构数据	初 值	ipc_perm 结构数据	初 值
<code>Sem_otime</code>	0	<code>Sem_nsems</code>	<code>Nsems</code>
<code>Sem_ctime</code>	系统当前值		

参数 `nsems` 是一个大于等于 0 的值，用于指明该信号量集中可用资源数（在创建一个信号量时）。当打开一个已存在的信号量集时该参数值为 0。函数执行成功，则返回信号量集的标识符（一个大于等于 0 的整数），失败，则返回 -1。函数 `semop` 用以操作一个信号量集，函数原型如下：

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

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

函数中参数 `semid` 是一个通过 `semget` 函数返回的一个信号量标识符，参数 `nops` 标明了参数 `semoparray` 所指向数组中的元素个数。参数 `semoparray` 是一个 `struct sembuf` 结构类型的数组指针，结构 `sembuf` 来说明所要执行的操作，其定义如下：

```
struct sembuf{
    unsigned short sem_num;
    short          sem_op;
    short          sem_flg;
}
```

在 `sembuf` 结构中，`sem_num` 是相对应的信号量集中的某一个资源，所以其值是一个从 0 到相应的信号量集的资源总数（`ipc_perm.sem_nsems`）之间的整数。`sem_op` 指明所要执行的操作，`sem_flg` 说明函数 `semop` 的行为。`sem_op` 的值是一个整数，如表 14-7 所示，列出了详细 `sem_op` 的值及所对应的操作。

表 14-7 sem_op 值详解

Sem_op	操 作
正数	释放相应的资源数，将 <code>sem_op</code> 的值加到信号量的值上
0	进程阻塞直到信号量的相应值为 0，当信号量已经为 0，函数立即返回。如果信号量的值不为 0，则依据 <code>sem_flg</code> 的 <code>IPC_NOWAIT</code> 位决定函数动作。 <code>sem_flg</code> 指定 <code>IPC_NOWAIT</code> ，则 <code>semop</code> 函数出错返回 <code>EAGAIN</code> 。 <code>sem_flg</code> 没有指定 <code>IPC_NOWAIT</code> ，则将该信号量的 <code>semncnt</code> 值加 1，然后进程挂起直到下述情况发生。信号量值为 0，将信号量的 <code>semzcnt</code> 的值减 1，函数 <code>semop</code> 成功返回；此信号量被删除（只有超级用户或创建用户进程拥有此权限），函数 <code>semop</code> 出错返回 <code>EIDRM</code> ；进程捕捉到信号，并从信号处理函数返回，在此情况将此信号量的 <code>semncnt</code> 值减 1，函数 <code>semop</code> 出错返回 <code>EINTR</code>
负数	请求 <code>sem_op</code> 的绝对值的资源。如果相应的资源数可以满足请求，则将该信号量的值减去 <code>sem_op</code> 的绝对值，函数成功返回。当相应的资源数不能满足请求时，这个操作与 <code>sem_flg</code> 有关。 <code>sem_flg</code> 指定 <code>IPC_NOWAIT</code> ，则 <code>semop</code> 函数出错返回 <code>EAGAIN</code> 。 <code>sem_flg</code> 没有指定 <code>IPC_NOWAIT</code> ，则将该信号量的 <code>semncnt</code> 值加 1，然后进程挂起直到下述情况发生：当相应的资源数可以满足请求，该信号量的值减去 <code>sem_op</code> 的绝对值。成功返回；此信号量被删除（只有超级用户或创建用户进程拥有此权限），函数 <code>semop</code> 出错返回 <code>EIDRM</code> ；进程捕捉到信号，并从信号处理函数返回，在此情况将此信号量的 <code>semncnt</code> 值减 1，函数 <code>semop</code> 出错返回 <code>EINTR</code>

下面实例演示了关于信号量操作的基本流程。程序中使用 `semget` 函数创建一个信号量集，并使用 `semop` 函数在这个信号集上执行了一次资源释放操作。并在 `shell` 中使用命令查看系统 `IPC` 的状态。

(1) 在 `vi` 编辑器中编辑该程序。

程序清单 14-10 create_sem.c 使用 `semget` 函数创建一个信号量

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

```

int main( void )
{
    int sem_id;
    int nsems = 1;
    int flags = 0666;
    struct sembuf buf;

    sem_id = semget(IPC_PRIVATE, nsems, flags);    /*创建一个新的信号量集*/

    if ( sem_id < 0 ){
        perror( "semget " );
        exit (1 );
    }

    /*输出相应的信号量集标识符*/
    printf ( "successfully created a semaphore : %d\n", sem_id );

    buf.sem_num = 0;                                /*定义一个信号量操作*/
    buf.sem_op = 1;                                  /*执行释放资源操作*/
    buf.sem_flg = IPC_NOWAIT;                        /*定义 semop 函数的行为*/

    if ( (semop( sem_id, &buf, nsems) ) < 0) {      /*执行操作*/
        perror ( "semop");
        exit (1 );
    }

    system ( "ipcs -s " );                          /*查看系统 IPC 状态*/
    exit ( 0 );
}

```

(2) 在 shell 中编译该程序如下:

```
$gcc create_sem.c-o create_sem
```

(3) 在 shell 中运行该程序如下:

```


$./ create_sem

----- Semaphore Arrays -----
key      semid    owner      perms      nsems
0x00000000 294911   root       666        1

successfully created a semaphore : 294911

```

在上面程序中, 用 `semget` 函数创建了一个信号量集, 定义信号量集的资源数为 1, 接下来使用 `semop` 函数进行资源释放操作。在程序的最后使用 shell 命令 `ipcs` 来查看系统 IPC 的状态。

 **注意:** 命令 `ipcs` 参数 `-s` 标识查看系统 IPC 的信号量集状态。

14.6.3 信号量集的操作

三个 IPC 对象类型中，信号量集的操作函数相对于其他两个类型的操作函数要复杂得多，当然信号量的应用也比其他两个更广泛些。像共享内存的操作一样，信号量也有自己的专属操作函数 `semctl`，函数原型如下：

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

int semctl( int sem_id, int semnu, int cmd [, union semun arg]);
```

函数中参数 `sem_id` 是一个信号量标识符，`semnum` 指定 `sem_id` 的信号集中的某一个信号灯，其类似于在信号量集资源数组中的下标，用来对指定资源进行操作。参数 `cmd` 定义函数所要进行的操作。其取值以及表达的意义如表 14-8 所示。

表 14-8 cmd值详解

cmd 的取值	操 作
GETVAL	返回成员 <code>semnum</code> 的 <code>semval</code> 值
SETVAL	使用 <code>arg.val</code> 对该信号量的 <code>semnum.sempid</code> 赋值（需要参数 <code>arg</code> ）
GETPID	返回成员 <code>semnum</code> 的 <code>sempid</code> 值
GETNCNT	返回成员 <code>semnum</code> 的 <code>semncnt</code> 值
GETZCNT	返回成员 <code>semnum</code> 的 <code>semzcnt</code> 值
GETALL	将该信号量集的值赋值到 <code>arg.array</code> （需要参数 <code>arg</code> ）
SETALL	使用 <code>arg.array</code> 数组中的值对信号量集赋值（需要参数 <code>arg</code> ）
IPC_RMID	删除信号量集。此操作只能由具有超级用户的进程或信号量集拥有者的进程执行，这个操作会影响到正在使用该信号量集的进程
IPC_SET	设置此信号量集的 <code>sem_perm.uid</code> 、 <code>sem_perm.gid</code> 以及 <code>sem_perm.mode</code> 的值。此操作只能由具有超级用户的进程或信号量集拥有者的进程执行
SPC_STAT	（需要参数 <code>arg</code> ）

函数中参数 `arg` 为可选参数，根据参数 `cmd` 的相关操作来选择使用，其定义如下：

```
union semun{
    int          val;
    struct semid_ds *buf ;
    unsigned short *array;
};
```

函数成功返回值大于等于 0（当 `semctl` 的操作为 GET 操作时返回相应的值，其余返回 0），失败返回 -1 并设置错误变量 `errno`。

下面实例演示了如何使用 `semctl` 函数。程序中先使用 `semget` 函数创建了一个新的信号量集，然后通过 `shell` 命令查看系统 IPC 的状态，再调用一次 `semctl` 函数做删除操作，并查看系统 IPC 的状态

（1）在 vi 编辑器中编辑该程序如下：

程序清单 14-11 ctl_sem.c 使用 `semctl` 删除信号量

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

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    int sem_id ;
    int nsems = 1;
    int flags = 0666;

    semid = semget ( IPC_PRIVATE, nsems, flags ); /*创建一个信号量集*/

    if ( sem_id < 0 ){                                /* 创建信号两失败 */
        perror ( "semget" );
        exit ( 1 );
    }

    printf ( "successfully created a semaphore: %d \n", sem_id );
                                                /*输出创建的信号量的 ID */

    system ( "ipcs -s" );                                /*查看系统 IPC 状态*/

    if ( (semctl (semid, 0, IPC_RMID)) < 0 ) {          /* 删除指定信号量集*/
        perror ( "semctl" );
        exit (1 );
    }
    else {
        printf ( "semaphore removed \n");
        system ( "ipcs -s " );                                /*查看系统 IPC 状态*/
    }

    exit ( 0 );
}

```

(2) 在 shell 中编译该程序如下:

```
$gcc ctl_sem.c-o ctl_sem
```

(3) 在 shell 中运行该程序如下:

```

$./ ctl_sem

----- Semaphore Arrays-----
key          semid      owner      perms      nsems
0x00000000  294911      root       666        1
0x0056a4d5  327681      root       600        1
0x00000000  360450      root       666        1

successfully created a semaphore: 360450

```



```

----- Semaphore Arrays -----
key          semid      owner      perms      nsems
0x00000000  294911      root       666        1
0x0056a4d5  327681      root       600        1

semaphore removed


```

在 shell 中可以调用如下命令来删除已存在的信号量：

```

$ ./ ipcrm -s <semaphore semid>

```

说明：上述程序中，使用 `semget` 函数创建一个信号量时，有可能系统中已经有了一个跟 `IPC_PRIVATE` 键关联的信号量，此时应在 shell 中先删除该 IPC，然后再运行程序。

14.7 消息队列

消息队列是一种以链表式结构组织的一组数据，存放在内核中，是由各进程通过消息队列标识符来引用的一种数据传送方式。像其他两种 IPC 对象一样，也是由内核来维护。消息队列是三个 IPC 对象类型中最具有数据操作性的数据传送方式，在消息队列中可以随意根据特定的数据类型值来检索消息。

14.7.1 消息队列的概念

消息队列是一个消息的链接表，该表由内核进行维护及存储。消息队列相比其他的通信方式对数据进行更细致的组织。可以通过一个消息的类型来索引指定的数据，它在数据流的概念上扩展了数据传送的概念，可以根据需要只读取指定数据，该点是管道和 FIFO 所不能比拟的。本节中使用术语队列和队列 ID 分别指消息队列和消息队列描述符。消息队列在多进程间通信示意如图 14-10 所示。

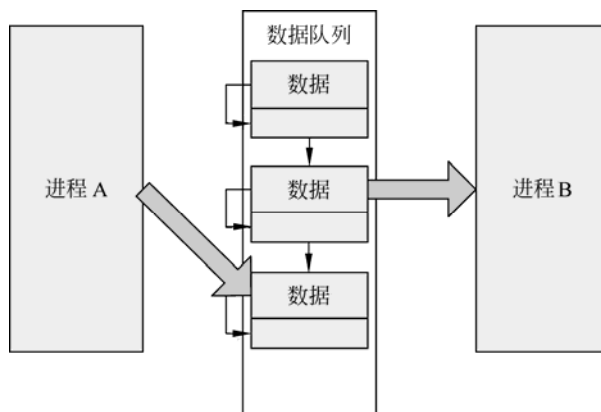



图 14-10 消息队列示意图

对于每个队列都有一个 `msqid_ds` 结构体来描述队列当前的状态。该结构体定义如下：

```
struct msqid_ds{
    struct ipc_perm    msg_perm;
    msgqnum_t    msg_qnum;
    msglen_t    msg_qbytes;
    pid_t    msg_lspid;
    pid_t    msg_lrpid;
    time_t    msg_stime;
    time_t    msg_rtime;
    time_t    msg_ctime;
    ...
    ...
};
```

说明：在不同的系统中，此结构会有不同的新成员，这里只列出最少拥有的关键成员。其中，`msg_qbytes` 成员以及 `msg_qnum` 成员在不同的系统也会有不同的上限值，这里就不逐一介绍了，详细内容请参阅相关系统手册。

14.7.2 创建消息队列

消息队列是三个 IPC 对象类型中最具有数据操作性的数据传送方式，在消息队列中可以随意根据特定的数据类型值来检索消息。当然，其缺点也显而易见，为了维护该数据链表，就需要更多的内存资源，而且在数据读写上比起共享内存也更复杂一些，时间开销也更大一些。

函数 `msgget` 可以创建或打开一个队列，函数原型如下：

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

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

函数中参数 `key` 用来转换成一个标识符，每一个 IPC 对象与一个 `key` 相对应。参数 `flags` 标明函数的行为。下面实例演示了使用 `msgget` 函数创建一个队列，函数中参数 `flags` 指定为 `IPC_CREAT|0666`，说明新建一个权限为 0666 的消息队列，其中组用户、当前用户以及其他用户拥有读写的权限。并在程序的最后使用 `shell` 命令 `ipcs -q` 来查看系统 IPC 的状态。

(1) 在 `vi` 编辑器中编辑该程序如下：

程序清单 14-12 create_msg.c msgget 函数

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

int main ( void )
{
    int    qid;
```

```

key_t      key;

key = 113;
qid=msgget( key, IPC_CREAT | 0666 );           /*创建一个消息队列*/

if ( qid < 0 ) {                               /* 创建一个消息队列失败 */
    perror ( "msgget" );
    exit (1) ;
}

printf ("created queue id : %d \n", qid ); /* 输出消息队列的 ID */

system( "ipcs -q" );                           /*查看系统 IPC 的状态*/
exit ( 0 );
}

```

(2) 在 shell 中编译该程序如下:

```
$gcc create_msg.c-o create_msg
```

(3) 在 shell 中运行该程序如下:

```

$./ create_msg

created queue id : 0

----- Message Queues -----
key      msqid      owner      perms      used-bytes      messages
0x0000af40 623430    root       666        0                0
0x0000007b 0             root       666        0                0

```

在程序中使用了系统命令 `ipcs`，命令参数 `-q` 说明只查看消息队列的状态。注意在输出消息中，key 段标明的是 IPC 的 key 值，msqid 为该队列的 ID 值，perms 为执行权限。同样，队列的执行权限像其他 IPC 对象一样没有执行权限。函数 `msgctl` 可以在队列上做多种操作，函数原型如下：

```

#include <sys/msg.h>

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

```

参数 `msqid` 为指定的要操作的队列，`cmd` 参数指定所要进行的操作，其中有些操作需要 `buf` 参数。`cmd` 参数的详细取值及操作如表 14-9 所示。

表 14-9 cmd参数详解

cmd	操 作
IPC_STAT	取队列的 <code>msqid_ds</code> 结构，将它存放在 <code>buf</code> 所指向的结构中（需要 <code>buf</code> 参数）
IPC_SET	使用 <code>buf</code> 所指向结构中的值对当前队列的相关结构成员赋值，其中包括： <code>msg_perm.uid</code> 、 <code>msg_perm.gid</code> 、 <code>msg_perm.mode</code> 以及 <code>msg_perm.cuid</code> 。该命令只能由具有以下条件的进程执行：进程有效用户 ID 等于 <code>msg_perm.cuid</code> 或 <code>msg_perm.uid</code> 超级用户进程。其中只有超级用户才可以增加队列的 <code>msg_qbytes</code> 的值
IPC_RMID	删除队列，并清除队列中的所有消息。此操作会影响后续进程对这个队列的相关操作。该命令只能由具有以下条件的进程执行。进程有效用户 ID 等于 <code>msg_perm.cuid</code> 或 <code>msg_perm.uid</code> ，超级用户进程

下面实例演示了调用 `msgctl` 函数操作队列，程序中先读取命令行参数，如没有，则打印命令提示信息，在调用 `msgctl` 函数执行删除操作的前后分别调用了一次 `shell` 命令 `ipcs -q` 来查看系统 IPC 的状态。

(1) 在 vi 编辑器中编辑该程序如下：

程序清单 14-13 del_msg.c 调用 `msgctl` 删除指定队列

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

int main ( int argc ,char *argv[] )
{
    int qid ;

    if ( argc != 2 ){ /* 命令行参数出错 */
        puts ( "USAGE: del_msgq.c <queue ID >" );
        exit ( 1 );
    }

    qid = atoi ( argv[1] ); /* 通过命令行参数得到组 ID */
    system( "ipcs -q");

    if ( ( msgctl( qid, IPC_RMID, NULL ) ) < 0 ){ /* 删除指定的消息队列 */
        perror ( "msgctl" );
        exit ( 1 );
    }

    system( "ipcs -q");
    printf ( "successfully removed %d queue\n", qid ); /* 删除队列成功 */

    exit( 0 );
}
```

(2) 在 shell 中编译该程序如下：

```
$gcc del_msg.c-o del_msg
```

(3) 在 shell 中运行该程序如下：

```
$. / del_msg

----- Message Queues -----
key      msqid      owner      perms      used-bytes  messages
0x0000007b 0          root       666        0           0

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

```
successfully removed 0 queue
```

14.7.3 读写消息队列

由于消息队列的特殊性，系统为这个数据类型提供了两个接口（`msgsnd` 函数，`msgrcv` 函数），分别对应写消息队列及读消息队列。将一个新的消息写入队列，使用函数 `msgsnd`，函数原型如下：

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

int msgsnd ( int msqid, const void *prt, size_t nbytes, int flags);
```

对于写入队列的每一个消息，都含有三个值，正长整型的类型字段、数据长度字段和实际数据字节。新的消息总是放在队列的尾部，函数中参数 `msqid` 指定要操作的队列，`prt` 指针指向一个 `msgbuf` 的结构，定义如下：

```
struct msgbuf{
    long mtype;
    char mbuf[];
};
```

这是一个模板的消息结构，其中成员 `mbuf` 是一个字符数组，长度是根据具体的消息来决定的，切忌消息不能以 `NULL` 结尾。成员 `mtype` 是消息的类型字段。

函数参数 `nbytes` 指定了消息的长度，参数 `flags` 指明函数的行为。函数成功返回 0，失败返回 -1 并设置错误变量 `errno`。`errno` 可能出现的值有：`EAGAIN`、`EACCES`、`EFAULT`、`EIDRM`、`EINTR`、`EINVAL` 和 `ENOMEM`。当函数成功返回后会更新相应队列的 `msqid_ds` 结构。

使用函数 `msgrcv` 可以从队列中读取消息，函数原型如下：

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

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

函数中参数 `msqid` 为指定要读的队列，参数 `ptr` 为要接收数据的缓冲区，`nbytes` 为要接收数据的长度，当队列中满足条件的消息长度大于 `nbytes` 的值时，则会参照行为参数 `flag` 的值决定如何操作：当 `flag` 中设置了 `MSG_NOERROR` 位时，则将消息截短到 `nbytes` 指定的长度后返回。如没有 `MSG_NOERROR` 位，则函数出错返回，并设置错误变量 `errno`。设置 `type` 参数指定 `msgrcv` 函数所要读取的消息，`tyre` 的取值及相应操作如表 14-10 所示。

表 14-10 type 值详解

type	操 作
等于 0	返回队列最上面的消息（根据先进先出规则）
大于 0	返回消息类型与 <code>type</code> 相等的第 1 条消息
小于 0	返回消息类型小于等于 <code>type</code> 绝对值的最小值的第 1 条消息

参数 `flag` 定义函数的行为，如设置了 `IPC_NOWAIT` 位，则当队列中无符合条件的消

息时，函数出错返回，`errno` 的值为 `ENMSG`。如没有设置 `IPC_NOWAIT` 位，则进程阻塞直到出现满足条件的消息出现为止，然后函数读取消息返回。

下面实例演示了消息队列在进程间的通信。程序中创建了一个消息的模板结构体，并对声明变量做初始化。使用 `msgget` 函数创建了一个消息队列，使用 `msgsnd` 函数向该队列中发送了一条消息。

(1) 在 vi 编辑器中编辑该程序如下：

程序清单 14-14 `snd_msg.c` 调用 `msgsnd` 函数向队列中发送消息

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

struct msg{
    long msg_types;          /*声明消息结构体*/
    char msg_buf[511];       /*消息类型成员*/
};

int main( void ) {
    int    qid;
    int    pid;
    int    len;
    struct msg pmsg;         /*一个消息的结构体变量*/

    pmsg.msg_types = getpid(); /*消息类型为当前进程的 ID*/
    sprintf (pmsg.msg_buf,"hello!this is :%d\n", getpid() ); /*初始化消息*/
    len = strlen ( pmsg.msg_buf ); /*取得消息长度*/

    if ( (qid=msgget(IPC_PRIVATE, IPC_CREAT | 0666)) < 0 ) { /*创建一个消息队列*/
        perror ( "msgget" );
        exit (1) ;
    }

    if ( (msgsnd(qid, &pmsg, len, 0 )) < 0 ){ /*向消息队列中发送消息*/
        perror ( "msgsn" );
        exit ( 1 );
    }

    printf ("successfully send a message to the queue: %d \n", qid);
    exit ( 0 ) ;
}
```

(2) 在 shell 中编译该程序如下：

```
$gcc snd_msg.c -o snd_msg
```

(3) 在 shell 中运行该程序如下：

```

$./ snd_msg

successfully send a message to the queue 0

```

上述程序中，先定义了一个消息的结构体。该结构体中包含两个成员，`long` 类型成员 `msg_types` 是消息的类型，注意，在消息队列中是以消息类型做索引值来进行检索的。`char` 类型数组存放消息。在程序中先声明了一个消息的结构体变量，并做相应初始化，然后使用了 `msgget` 函数创建一个消息队列，并将该消息发送到此消息队列中。以下是一个使用消息队列发送消息的程序。

下面实例演示了如何使用队列读取消息。在程序的开始部分，判断用户是否输入了目标消息队列 ID，如果没有，则打印命令的帮助信息；如果用户输入了队列的 ID，则从队列中取出该消息，并输出到标准输出。

(1) 在 vi 编辑器中编辑该程序。

程序清单 14-15 rcv_msg.c 使用 `msgrcv` 函数从指定队列中读出消息

```

#include <sys/msg.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFSZ 4096

struct msg{
    /*声明消息结构体*/
    long msg_types; /*消息类型成员*/
    char msg_buf[511]; /*消息*/
};

int main( int argc, char * argv[] ) {
    int qid;
    int len;
    struct msg pmsg;

    if ( argc != 2 ){ /**/
        perror ( "USAGE: read_msg <queue ID>" );
        exit ( 1 );
    }

    qid = atoi ( argv[1] ); /*从命令行中获得消息队列的 ID*/

    /*从指定队列读取消息 */
    len = msgrcv ( qid, &pmsg, BUFSZ, 0, 0 );

    if ( len > 0 ){
        pmsg.msg_buf[len] = '\0'; /*为消息添加结束符*/
        printf ( "reading queue id :%05ld\n", qid ); /*输出队列 ID*/
        /*该消息类型就是发送消息的进程 ID*/
    }
}

```

```

    printf ("message type : %05ld\n", pmsg.msg_types );
    printf ("message length : %d bytes\n", len ); /*消息长度*/
    printf ("message text: %s\n", pmsg.msg_buf);    /*消息内容*/
}
else if ( len == 0 )
    printf ("have no message from queue %d\n", qid );
else {
    perror ( "msgrcv");
    exit (1);
}
system("ipcs -q")
exit ( 0 ) ;
}

```

(2) 在 shell 中编译该程序如下:

```
$gcc rcv_msg.c-o rcv_msg
```

(3) 在 shell 中运行该程序如下:

```

$./ rcv_msg 0

reading queue id :0
message type : 03662
message length : 20 bytes
message text: hello!this is :3662

----- Message Queues -----
key          msqid      owner      perms      used-bytes   messages
0x00000000   0          root      666        0             0

```

该程序中声明了一个消息的结构体类型变量，并从命令行中得到所要操作的消息队列，然后使用函数 `msgrcv` 从指定消息队列中读取队列中最上面的一条消息（函数的第 4 个参数等于 0，说明根据先进先出规则，应从队列的最上面读取一条消息），并将该消息输出到标准输出。在发送消息的程序中，消息类型字段指定的是发送消息进程的 ID，可以使用该内容来判断信息的来源。