# 第15章 进程间通信

## 15.1引言

前面章节讨论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方法只能是经由fork或exec传送打开文件，或者通过文件系统。本章将说明进程之间相互通信的其他技术IPC (InterProcess Communication)。

过去，LINUX/UNIX系统IPC是各种进程通信方式的统称，但是，其中极少能在所有LINUX/UNIX系统实现中进行移植。随着POSIX和Open Group (以前是X/Open)标准化的推进和影响的扩大，情况虽已得到改善，但差别仍然存在。一般来说，LINUX/UNIX常见的进程间通信方式有如下这些：

管道、消息队列、信号量、共享内存、套接字等。

## 15.2管道

管道是LINUX/UNIX系统IPC的最古老形式，并且所有LINUX/UNIX系统都提供此种通信机制。管道有下 面两种局限性：

(1) 历史上，它们是半双工的（即数据只能在一个方向上流动）。现在，某些系统提供全双工管道，但是为了最佳的可移植性，我们决不应预先假定系统使用此特性。

(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FIFO 没有第二种局限性，LINUX/UNIX域套接字和命名流管道则没有这两种局限性。

尽管有这两种局限性，半双工管道仍是最常用的IPC形式。每当你在管道线中键入一个由shell执行的命令序列时，shell为每一条命令单独创建一个进程，然后将前一条命令进程的标准输出用管道与后一条命令的标准输人相连接。

管道是由调用pipe函数而创建的：

|  |
| --- |
| #include <unistd.h>  int pipe (int filedes[2]);  返回值：若成功则返回0，若出错则返回-1 |

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

有两种方法来描绘一个半双工管道，见下图。

|  |
| --- |
|  |
| 观察半双工管道的两种方法 |

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。

fstat函数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。

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（或反向）的IPC通道。下图显示了这种情况。

|  |
| --- |
|  |
| 调用fork之后的半双工管道 |

调用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[l]）。下图显示了在此之后描述符的安排。

|  |
| --- |
|  |
| 从父进程到子进程的管道 |

为了构造从子进程到父进程的管道，父进程关闭fd[l]，子进程关闭fd[0]。

当管道的一端被关闭后，下列两条规则起作用：

(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，使得有多个进程对它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。后面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返回-1， errno设置为EPIPE。

在写管道(或FIFO)时，常量PIPE\_BUF规定了内核中管道缓冲区的大小。如果对管道调用write，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），而且有进程要求写的字节数超过PIPE\_BUF字节数时，则写操作的数据可能相互穿插。

下面的程序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。

|  |
| --- |
|  |

在上面的例子中，直接对管道描述符调用read和write。更好的方法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执行另一个程序，该程序或者从标准输入（已创建的管道）读数据，或者将数据写至其标准输出（该管道）。

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LINUX/UNIX系统实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，调用fork产生一个子进程，使子进程的标准输入成为管道的读端，然后调用exec，执行用户喜爱的分页程序。

下面程序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。

|  |
| --- |
|  |

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），应当注意在复制之前该描述符的值并不是所希望的值。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。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试图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。

会议前面的5个函数：TELL\_WAIT、TELL\_PARENT、TELL\_CHILD、WAIT\_PARENT 以及WAIT\_CHILD。前面提供了一个使用信号的实现。下列则是一个使用管道的实现。

|  |
| --- |
|  |

如下图所示所示：

|  |
| --- |
|  |
| 用两个管道实现父子进程同步 |

在fork之前创建了两个管道。父进程在调用TELL\_CHILD时，写一个字符“p”至上一个管道，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读这个字符，并发生阻塞。

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pfdl[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。

## 15.3 popen和pclose函数

常见的操作是创建一个管道连接到另一个进程，然后读其输出或向其输入端发送数据，为此，标准I/O库提供了两个函数popen和pclose。这两个函数实现的操作是：创建一个管道，调用fork产生一个子进程，关闭管道的不使用端，执行一个shell以运行命令，然后等待命令终止。

|  |
| --- |
| #include <stdio.h>  FILE \*popen (const char \* cmdstring, const char \*type);  返回值：若成功则返回文件指计，若出错则返回NULL  int pclose (FILE \*fp)；  返回值：cmdstring的终止状态，若出错则返回-1 |

函数popen先执行fork，然后调用exec以执行并且返回一个标准I/O文件指针。如果type是"r"，则文件指针连接到的标准输出。

|  |
| --- |
|  |
| 执行fp = popen (cmdsirmg, "r")函数的结果 |

如果type是"w"，则文件指针连接到cmdstring的标准输入。

|  |
| --- |
|  |
| 执行fp = popen(cmdstring, "w")函数的结果 |

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。我们曾在前面对终止状态进行过说明，system函数也返回终止状态。如果shell不能被执行，则pclose返回的终止状态与shell已执行exit (127) —样。

cmdstring由shell以下列方式执行； sh -c cmdstring

这表示shell将扩展中的任何特殊字符。例如，可以使用，

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

或者

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

用popen重写前面的第二个程序如下：

|  |
| --- |
|  |

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。

下面程序是一个popen和pclose的实现。

|  |
| --- |
|  |

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poperi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得到文件描述符，然后取得子进程ID，并用其作为参数调用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），其数组长度应当是最大文件描述符数，于是该数组中可以存放与最大文件描述符数相同的子进程。

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.1标准要求子进程关闭在以前调用popen时打开且当前仍旧打开的所有I/O流。为此，在子进程中从头逐个检査childpid数组的各元素，关闭仍旧打开的任何描述符。

若pclose的调用者已经为信号SIGCHLD设置了一个信号处理程序，则pclose中的waitpid调用将返回一个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。

注意，如果应用程序调用waitpid，并且获得popen所创建的子进程的终止状态，则在应用程序调用pclose时，其中将调用waitpid，它发现子进程已不再存在，此时返回-1，errno则被设置为ECHILD。这正是POSIX.1所要求的。

注意，popen决不应由设置用户ID或设置组ID程序调用。当它执行命令时，popen等同于：

execl("/bin/sh"# "sh", "-c", command, NULL);

它在从调用者继承的环境中执行shell,并由shell解释执行一个心怀不轨的用户可以操纵这种环境，使得shell能以设置ID文件模式所授与的提升了的权限以及非预期的方式执行命令。

popen特别适用于构造简单的过滤器程序，它变换运行命令的输入或输出。当命令希望构造它自己的管道时，就是这种情形。

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便对输入进行变换处理。下图显示了为此做的进程安排。

|  |
| --- |
|  |
| 用popen对输入进行变换处理 |

对输入进行的变换可能是路径名扩充，或者是提供一种历史机制（记住以前输入的命令）。

下面是一个简单的过滤程序，它只是将标准输入复制到标准输出，在复制时将所有大写字符变换为小写字符。在写了一行之后，对标准输出进行了冲洗(用fflush)，其理由将在下节介绍协同进程时讨论。

|  |
| --- |
|  |

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，然后在下面程序中用popen调用它们。

|  |
| --- |
|  |

因为标准输出通常是行缓冲的，而提示并不包含换行符，所以在写了提示之后，需要调用fflush。

## 15.4协同进程

LINUX/UNIX系统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。几个过滤程序通常在shell管道命令行中线性地连接。当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）。

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

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。下图显示了这种安排。

|  |
| --- |
|  |
| 写协同进程的标准输入，读它的标准输出 |

下面程序程序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。协同进程通常会做较此更有意义的工作。设计本实例的目的是帮助了解将进程连接起来所需的各种管道设施。

|  |
| --- |
|  |

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。

下面程序是从其标准输入读入两个数之后调用add2协同进程，并将协同进程送来的值写到其标准输出。

|  |
| --- |
|  |

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。两个管道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。子进程调用dup2使管道描述符移至其标准输入和标准输出，然后调用execl。

若编译和运行上述程序，它如所希望的那样进行工作。进而言之，在程序正等待输入时，若先杀死add2协同进程，然后输入两个数，接着程序对管道进行写操作，此时，由于该管道已无读进程，于是调用信号处理程序。

在协同进程add2中，有意地使用了read和writeI/O（LINUX/UNIX系统调用）。如果使用标准I/O改写该协同进程，其后果是什么呢？下面程序就是改写后的版本。

|  |
| --- |
|  |

若前面的程序程序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缓冲机制上。当调用新的程序时，对标准输入的第一个fgets引起标准I/O库分配一个缓冲区，并选择缓冲区的类型。因为标准输入是个管道，所以标准I/O库由系统默认是全缓冲的。对标准输出也作同样的处理。当add2从其标准输入读取而发生阻塞时，前面的程序从管道读时也发生阻塞，于是发生了死锁。

为此，更改将要运行的协同进程的缓冲类型，在程序中的while循环之前加上下面4行：

if (setvbuf(stdin, NULL, 一IOLBF, 0) 1= 0)

err\_sys("setvbuf error");

if (setvbuf(stdout, NULL, \_IOLBF, 0) != 0)

errsys("setvbuf error");

这些代码行使得当有一行可用时，fgets就返回，并使得当输出一换行符时，printf立即执行fflush操作。对setvbuf进行的这些显式调用使得上面程序能正常工作。

如果不能修改这种协同进程程序，则需使用其他技术。例如，如果在程序中使用awk(l)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的缓冲机制问题。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）。我们不能修改awk的可执行代码，于是也就不能更改处理其标准I/O缓冲的方式。

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

## 15.5 FIFO

FIFO有时被称为命名管道。管道只能由相关进程使用，这些相关进程的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。

前面已经提及FIFO是一种文件类型。stat结构成员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/stat.h>  int mkfifo (const char \*pathname, mode\_t mode);  返回值：若成功则返回0，若出错则返回-1 |

mkfifo函数中mode参数的规格说明与open函数中的mode相同。新FIFO的用户和组的所有权规则与文件章节所述的相同。

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

应用程序可以用mknod函数创建FIFO。POSIX.1原先并没有包括mknod函数，它首先提出了mkfifo。mknod现在已包括在LINUX/UNIX。在大多数系统中，mkf ifo调用mknod创建FIFO。

当打开一个FIFO时，非阻塞标志（O\_NONBLOCK）产生下列影响：

在一般情况中（没有指定O\_NONBLOCK），只读open要阻塞到某个其他进程为写而打开此FIFO。类似地，只写open要阻塞到某个其他进程为读而打开它。

如果指定了O\_NONBLOCK，则只读open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open将出错返回-1，其errno是ENXIO。

类似于管道，若用write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有多个写进程是很常见的。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，常量PIPE\_BUF说明了可被原子地写到FIFO的最大数据量。

FIFO有下面两种用途：

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

(2) FIFO用于客户进程-服务器进程应用程序中，以在客户进程和服务器进程之间传递数据。

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。下图表示了这种安排。

|  |
| --- |
|  |
| 对一个经过过滤的输人流进行两次处理 |

使用FIFO以及LINUX/UNIX系统程序tee(l)，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出以及其命令行中包含的命名文件中。）

mkfifo fifol

prog3 < fifol &

progl < infile | tee fifol | prog2

我们创建FIFO，然后在后台启动prog3，它从FIFO读数据。然后启动progl，用tee将其输出发送到FIFO和prog2。下图显示了有关安排。

|  |
| --- |
|  |
| 使用FIFO和tee将一个流发送到两个进程 |

FIFO的另一个应用是在客户进程和服务器进程之间传送数据。如果有一个服务器进程，它与很多客户进程有关，则每个客户进程都可将其请求写到一个该服务器进程创建的众所周知的FIFO中（“众所周知”的意思是：所有需与服务器进程联系的客户进程都知道该FIFO的路径名）。 下图显示了这种安排。

|  |
| --- |
|  |
| 客户进程用FIFO向服务器进程发送请求 |

因为对于该FIFO有多个写进程，客户进程发送给服务器进程的请求其长度要小于PIPE\_BUF字节。这样就能避免客户多fwrite之间的交错。

在这种类型的客户进程—服务器进程通信中使用FIFO的问题是：服务器进程如何将回答送回各个客户进程。不能使用单个FIFO，因为服务器进程会发出对各个客户进程请求的响应，而请求者却不可能知道什么时候去读才能恰如其分地读到对它的响应。一种解决方法是毎个客户 进程都在其请求中包含它的进程ID。然后服务器进程为毎个客户进程创建一个FIFO,所使用的 路径名是以客户进程的进程ID为基础的。例如，服务器进程可以用名字/tmp/servl.XXXXX 创建FIFO，其中XXXXX被替换成客户进程的进程ID。下图显示了这种安排。

|  |
| --- |
|  |
| 客户进程-服务器进程用FIFO进行通信 |

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

按照上图的安排，如果服务器进程以只读方式打开众所周知的FIFO （因为它只需读该FIFO），则毎当客户进程数从1变成0时，服务器进程就将在FIFO中读到一个文件结束标记。为使服务器进程免于处理这种情况，一种常用的技巧是使服务器进程以读-写方式打开其FIFO 。

## 15.6 XSI IPC

有三种IPC我们称作XSI IPC，即消息队列、信号量以及共享存储，它们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，本节先介绍它们相类似的特征。

### 15.6.1标识符和键

毎个内核中的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，只需要知道其队列标识符。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，然后回转到0。

标识符是IPC对象的内部名。为使多个合作进程能够在同一IPC对象上会合，需要提供一个外部名方案。为此使用了键（可以），每个IPC对象都与一个键相关联，于是键就用作为该对象的外部名。

无论何时创建IPC结构（调用semget或shmget），都应指定一个键，键的数据类型是基本系统数据类型key\_t，通常在头文件<sys/types.h>中被定义为长整型。键由内核变换成标识符。

有多种方法使客户进程和服务器进程在同一IPC结构上会合：

(1) 服务器进程可以指定键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）以便客户进程取用。键IPC\_PRIVATE保证服务器进程创建-个新IPC结构。这种技术的缺点是：服务器进程要将整型标识符写到文件中，此后客户进程又要读文件取得此标识符。

IPC\_PRIVATF键也可用于父、子进程关系。父进程指定IPC\_PRIVATE创建一个新IPC结构，所返回的标识符在调用fork后可由子进程使用。接着，子进程又可将此标识符作为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进程和服务器进程都认可的键。然后服务器进程指定此键创建一个新的IPC结构。这种方法的问题是该键可能已与一个IPC结构相结合，在此情况下，get函数（semget或shmget）出错返回。服务器进程必须处理这一错误，删除已存在的IPC结构，然后试着再创建它。

(3) 客户进程和服务器进程认同一个路径名和项目ID（项目ID是0~255之间的字符值），接着调用函数ftok将这两个值变换为一个键。然后在方法(2)中使用此键。ftok提供的唯一服务就是由一个路径名和项目ID产生一个键。

|  |
| --- |
| #include <sys/ipc.h>  key\_t ftok(const char \*path, int id);  返回值：若成功则返回键，若出错则返回(key\_t)-1 |

path参数必须引用一个现存文件。当产生键时，只使用id参数的低8位。

ftok创建的键通常是用下列方式构成的：按给定的路径名取得其stat结构，从该结构中取出部分st\_dev和st\_ino字段，然后再与项目ID组合起来。如果两个路径名引用两个不同的文件，那么，对这两个路径名调用ftok通常返回不同的键。但是，因为i节点号和键通常都存放在长整型中，于是创建键时可能会丢失信息。这意味着，如果使用同一项目ID, 那么对于不间文件的两个路径名可能产生相同的键。

三个get函数(msgget、semget和shmget)都有两个类似的参数：一个key和一个整型flag。如若满足下列两个条件之一，则创建一个新的IPC结构(通常由服务器进程创建)：

* key是 IPC\_PRIVATE,
* key当前未与特定类型的IPC结构相结合，并且flag中指定了IPC\_CREAT位。

为访问现存的队列（通常由客户进程进行），必须等于创建该队列时所指定的键，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作为键。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键创建的现存队列，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和msgrcv）使用该标识符。

如果希望创建一个新的IPC结构，而且要确保不是引用具有间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）。

416 第15章进程间通信

### 15.6.2权限结构

XSI IPC为每一个IPC结构设置了一个ipc\_perm结构。该结构规定了权限和所有者。它至少包括下列成员：

truct ipc\_perm

{

uid\_t uid； /\* owner's effective user id \*/

gid\_t gid； /\* owner,s effective group id \*/

uid\_t cuid； /\* creatorf s effective user id \*/

gid\_t cgid； /\* creator's effective group id \*/

mode\_t mode； /\* access modes \*/

};

每种实现在其ipc\_perin结构中会包括另外一些成员。如想要了解你所用系统中它的完整定义，请参见< sys/ipc.h>。

在创建IPC结构时，对所有字段都赋初值。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字段的值类似于文件模式表所示的值，但是对于任何IPC结构都不存在执行权限。另外消息队列和共享存储使用术语读（read）和写（write），而信号量则用术语读(rend)和更改 (alter)。下表中对每种IPC说明了6种权限。

|  |  |
| --- | --- |
| XSIIPC权限 | |
| 权限 | 位 |
| 用户读 | 0400 |
| 用户写(更改) | 0200 |
| 组读 | 0040 |
| 组写(更改) | 0020 |
| 其他读 | 0004 |
| 其他写(更新) | 0002 |

### 15.6.3结构限制

三种形式的XSI IPC都有内置限制(built-in limit)。这些限制的大多数可以通过重新配置内核而加以更改。当叙说每种IPC时，我们都会指出它的限制。

### 15.6.4优点和缺点

XSI 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，如果进程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会被删除。它们余留在系统中直至出现下述情况：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(l)命令删除消息队列，或由正在再启动的系统删除消息队列。将此与管道相比，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HFO中的数据却在此时全部被删除，于是也就徒有其名了。

XSI IPC的另一个问题是：这些IPC结构在文件系统中没有名字。我们不能用前面章节中所述的函数来访问它们或修改它们的特性。为了支持它们不得不增加了十几条全新的系统调用（msgget、semop、shmat等）。我们不能用ls命令见到IPC对象，不能用rm命令删除它们，也不能用chmod命令更改它们的访问权限。于是，就不得不增加新的命令ipcs(l)和 ipcrm(l)。

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。这就使得难于一次使用多个IPC结构，以及在文件或设备I/O中使用IPC结构。例如，没有某种形式的忙-等待循环，就不能使一个服务器进程等待将要放在两个消息队列任一个中的消息。

下面对三种形式的IPC进行详细的说明。

## 15.7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。在本节中，我们把消息队列简称为队列（queue），其标识符为队列ID（queue ID）。

msgget用于创建一个新队列或打开一个现存的队列。msgsnd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阮数据字节（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结构与其相关联：

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;

…

};

此结构规定了队列的当前状态。结构中所示的各成员是由Single UNIX Specification定义的。具体实现可能包括标准中没有定义的另一些字段。

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/msg.h>  int msgget (key\_t key, int flag);  返回值：若成功则返回消息队列ID,若出错则返回-1 |

前面说明了将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该结构中mode成员按flag中的相应权限位设置。
* 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\_gbytes设置为系统限制值。

若执行成功，msgget返回非负队列ID。此后，该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。它和另外两个与信号量和共享存储有关的函数（semctl和shmctl）是XSI IPC的类似于ioctl的函数（亦即垃圾桶函数）。

|  |
| --- |
| #include <sys/msg.h>  int msgctl (int msqid, int cmd, struct msqid\_ds \*buf);  返回值：若成功则返回0.若出错则返回-1 |

cmd参数说明对由指定的队列要执行的命令：

|  |  |
| --- | --- |
| IPC\_STAT | 取此队列的msgid\_ds结构，并将它存放在如/指向的结构中。 |
| IPC\_SET | 按由buf指向结构中的值，设置与此队列相关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中的所有数据。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中。

|  |
| --- |
| #include <sys/msg.h>  int msgsnd (int msqid, const void \*ptr, size\_t tibytes, int flag);  返回值：若成功则返回0，若出错则返回-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参数指向一个长整型数，它包含了正的整型消息类型，在其后紧跟着消息数据。（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。

参数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。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， 则进程阻塞直到下述情况出现为止：有空间可以容纳要发送的消息，从系统中删除了此队列，或捕捉到一个信号，并从信号处理程序返回。在第二种情况下，返回EIDRM（“标识符被删除”）。 最后一种情况则返回EINTR。

注意，对删除消息队列的处理不是很完善。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），所以删除一个队列会造成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。信号量机制也以同样方式处理其删除。相反，删除一个文件时，要等到使用该文件的最后一个进程关闭了它的文件描述符后，才能删除文件中的内容。

当msgsnd成功返回，与消息队列相关的msgid\_ds结构得到更新，以标明发出该调用的 进程ID（msg.lspid），进行该调用的时间（msg\_stime），并指示队列中增加了一条消息 （msg-qnum）。

msgrcv从队列中取用消息：

|  |
| --- |
| #include <sys/msg.h>  ssize\_t msgrcv(int msqid, void \*ptr, size\_t nbytes, long type, int flag);  返回值：若成功则返回消息的数据部分的长度，若出错则返回-1 |

如同msgsnd中一样，pfr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冲区。说明数据缓冲区的长度。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，消息的截去部分被丢弃。）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绝对值的消息，如果这种消息有若干个，则取类型值最小的消息。

Type值非0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进程和一个服务器进程使用，那么type字段可以用来包含客户进程的进程ID（只要进程ID可以存放在长整型中）。

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返回-1，errno设置为ENOMSG。如果没有指定IPC\_NOWAIT，则进程阻塞直至如下情况出现才终止：有了指定类型的消息，从系统中删除了此队列（出错则返回-1且errno置为EIDRM），或捕捉到一个信号并从信号处理程序返回（msgrcv返回-1，errno设置为 EINTR）。

msgrcv成功执行时，内核更新与该消息队列相关联的msgid\_ds结构，以指示调用者的 进程ID （msg\_lrpid）和调用时间（msg\_rtime），并将队列中的消息数（msg\_qnum）减1。

消息队列原来的实施目的是提供比一般IPC更髙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。考虑到使用消息队列具有的问题，我们得出的结论是，在新的应用程序中不应当再使用它们。

## 15.8信号量

信号量（semaphore）与已经介绍过的IPC机制（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访问。

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，则进程进入休眠状态，直至信号量值大于0。进程被唤醒后，它返回至第(1)步。

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

为了正确地实现信号量，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为二元信号量或双态信号量（binary semaphore）。它控制单个资源，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。

遗憾的是，XSI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) 信号量并非是单个非负值，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，要指定该集合中信号量值的数量。

(2) 创建信号量(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，并且对该集合中的各个信号量值陚初值。

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

内核为每个信号量集合设置了一个semid\_ds结构：

struct semid\_ds {

struct ipc\_perm sem\_perm;

unsigned short sem\_nsems;

time\_t sem\_otime;

time\_t sem ctime;

...

}

Single UNIX Specification定义了上面所示的各字段，但是具体实现可在semid\_ds结构中定义添加的成员。

每个信号量由一个无名结构表示，它至少包含下列成员：

struct {

unsigned short semval;

pid\_t sempid;

unsigned short sem ncnt;

unsigned short semzcnt；

...

}

要获得一个信号量ID,要调用的第一个函数是seraget。

|  |
| --- |
| #include <sys/sem.h>  int semget (key\_t key, int nsems, int flag);  返回值：若成功则返回信号量ID,若出错则返回-1 |

前面节说明了将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。创建一个新集合时，对semid\_ds结构的下列成员陚初值：

对ipc\_perm结构赋初值。该结构中的mode被设置为flag中的相应权限位。

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（一般在服务器进程中），则必须指定nsems。如果引用一个现存的集合（一个客户进程），则将nsems指定为0。

semctl函数包含了多种信号量操作。

|  |
| --- |
| #include <sys/sem.h>  int semctl (int semid, int semnum, int cmd, .../\* union semun arg \*/);  返回值：（见下） |

注意，依赖于所请求的命令，第四个参数是可选的，如果使用该参数，则其类型是semun，它是多个特定命令参数的联合（union）：

union semun {

int val; /\* for SETVAL \*/

struct semid\_ds \*buf; /\* for IPC\_STAT and IPC—SET \*/

unsigned short \*array; /\* for GETALL and SETALL \*/

}

注意，这是一个联合，而非指向联合的指针。

cmd参数指定下列10种命令中的一种，在指定的信号量集合上执行此命令。其中有5条命令是针对一个特定的信号量值的，它们用semnum指定该信号量集合中的一个成员。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#指向的结构中。 |
| IPC\_SET | 按由arg.buf指向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。这种删除是立即发生的。仍在使用此信号量集合的其他进程在它们下次试图对此信号量集合进行操作时，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 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.arrary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函数都返回相应的值。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，这是个原子操作。

|  |
| --- |
| #include <sys/sem.h>  int semop (int semid, struct sembuf semoparray[] , size\_t nops);  返回值：若成功则返回0，若出错则返回-1 |

参数semoparray是一个指针，它指向一个信号量操作数组，信号量操作由sembuf结构表示：

struct sembuf {

unsigned short sem\_num;

short sem\_op;

short sem \_flg;

};

参数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值规定。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。这对应于进程释放占用的资源数。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semop出错返回EAGAIN。

(b) 若未指定IPC\_NOWAIT，则该信号量的semncnt值加1（因为调用进程将进入休眠态），然后调用进程被挂起直至下列事件之一发生：

⑴ 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。

⑵从系统中删除了此信号量。在此情况下，函数出错则返回EIDRM。

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 （因为调用进程不再等待），并且函数出错返回EINTR。

(3) 若sem\_op为0，这表示调用进程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I PC\_NOWAIT，则出错返回EAGAIN。

(b) 若未指定IPC\_NOWAIT，则该信号量的semzcnt值加1 (因为调用进程将进入休眠状态)，然后调用进程被挂起，直至下列事件之一发生为止：

⑴ 此信号量值变成0。此信号量的semzcnt值减1 (因为调用进程已结束等待)。

⑵ 从系统中删除了此信号量。在此情况下，函数出错返回EIDRM。

⑶ 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 (因为调用进程不再等待)，并且函数出错返回EINTR。

semop函数具有原子性，它或者执行数组中的所有操作，或者什么也不做。

**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，分配给调用进程多少资源（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处理。

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。

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条件下进程终止的情况。

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一个锁的属主进程终止时，内核会自动释放该锁。

一般来说记录锁慢于信号量。虽然记录锁慢于信号量锁，但如果只需锁一个资源（例如共享存储段）并且不需要使用信号量的所有花哨的功能，则宁可使用记录锁。理由是使用简易，且进程终止时系统会处理任何遗留下来的锁。

## 15.9共享存储

共享存储允许两个或更多进程共享一给定的存储区。因为数据不需要在客户进程和服务器进程之间复制，所以这是最快的一种IPC。使用共享存储时要掌握的唯一窍门是多个进程之间对一给定存储区的同步访问。若服务器进程正在将数据放入共享存储区，则在它做完这一操作之前，客户进程不应当去取这些数据。通常，信号量被用来实现对共享存储访问的同步。（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 shmidds {

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;

}

(按照支持共享存储段的需要，毎种实现会在shmid\_ds结构中增加其他成员。)

shmatt\_t类型定义为不带符号整型，它至少与unsigned short一样大。

为获得一个共享存储标识符，调用的第一个函数通常是shmget。

|  |
| --- |
| #include <sys/shm.h>  int shmget (key\_t key, size\_z size, int flag);  返回值：若成功则返冋共享存储ID，若出错则返回-1 |

前面说明了将变换成一个标识符的规则，以及是创建一个新共享存储段还是引用一个现 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 该结构中的mode成员按flag中的相应权限位设置。

* shm\_lpid、shm\_nattach、shm\_atime、以及shm\_dtime都设置为0。
* shm\_ctime设置为当前时间。
* shm\_segsz设置为请求的长度（size）。

参数size是该共享存储段的长度（单位：字节）。实现通常将其向上取为系统页长的整数倍。但是，若应用指定的size值并非系统页长的整数倍，那么最后一页的余下部分是不可使用的。如果正在创建一个新段（一般是在服务器进程中），则必须指定其size。如果正在引用一个现存的段（一个客户进程），则将size指定为0。当创建一新段时，段内的内容初始化为0。

shmctl函数对共享存储段执行多种操作。

|  |
| --- |
| #include <sys/shm.h>  int shmctl (int shmid, int cmd, struct shmid\_ds \*buf);  返回值：若成功则返回0，若出错则返回-1 |

cmd参数指定下列5种命令中一种，使其在指定的段上执行。

IPC\_STAT取此段的shmid\_ds结构，并将它存放在由buf指向的结构中。

IPC\_SET 按buf指向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（shmid\_ds结构中的shm\_nattch字段），所以除非使用该段的最后一个进程终止或与该段脱节，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。

Linux提供了下列另外两种命令：

SHM\_LOCK 将共享存储段锁定在内存中。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。

一且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/shm.h>  void \* shmat (int shmid, const void \*addr, int flag);  返回值：若成功则返回指向共享存储的指针，若出错则返回-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。

* 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。这是推荐的使用方式。
* 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。
* 如果addr非0，并且指定了SHM\_RND，则此段连接到addr所表示的地址上。SHM\_RND命令的意思是“取整”。SHM\_RND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），否则不应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。如果shmat成功执行，那么内核将使该共享存储段shmid\_ds结构中的shm\_nattch计数器值加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（一般是服务器进程）调用 shmctl （带命令IPC\_RMID）特地删除它。

|  |
| --- |
| #include <sys/shm.h>  int shmdt (void \*addr);  返回值：若成功则返回0，若出错则返回-1 |

addr参数是以前调用shmat时的返回值。如果成功，shmdt将使相关shmid\_ds结构中的 shm\_nattch计数器值减1。

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。下面的程序打印一些信息，它们与特定系统将各种不同类型的数据放在什么位置有关。

|  |
| --- |
|  |

在一个基于Intel的Linux系统上运行此程序，其输出如下：

|  |
| --- |
|  |

下图描述了这种情况，这与讨论进程时的典型存储区布局类似。注意，共享存储段紧靠在栈之下。

|  |
| --- |
|  |
| 在基干Intel的Linux系统上的存储区布局 |

前面曾说明mmap函数可将一个文件的若干部分映射至进程地址空间。这在概念上类似于用shmat XSI IPC函数连接一共享存储段。两者之间的主要区别是，用mmap映射的存储段是与文件相关联的，而XSI共享存储段则并无这种关联。

共享存储可由不相关的进程使用。但如果进程是相关的，则某些实现提供了一种不同的技术。

在读设备/dev/zero时，该设备是0字节的无限资源。它也接收写向它的任何数据，但又忽略这些数据。我们对此设备作为IPC的兴趣在于，当对其进行存储映軒叶，它具有一些特殊的性质：

* 创建一个未名存储区，其长度是mmap的第二个参数，将其向上取整为系统的最近页长。
* 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。

下面的程序在父、子进程间使用/dev/zero存储映射I/O来进行IPC

|  |
| --- |
|  |

它打开此/dev/zero设备，然后指定长整型的长度调用mmap。注意，一旦存储区映射成功，就关闭此设备。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射区的数据可由另一进程见到。（如果已指定MAP\_PRIVATE，则此示例程序不能工作。）

然后，父、子进程交替运行，使用前面章节中的同步函数各自对共享存储映射区中的长整型数加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，因为增加的是其值，而不是指针，所以必须使用括号。

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：它只在相关进程间起作用。但在相关进程之间使用线程可能更为简单、有效。注意，无论使用哪一种技术，都需对共享数据进行同步访问。

很多实现提供了一种类似干/dev/zero的设施，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，并将文件描述符指定为-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），并且创建一个可与后代进程共享的存储区。

为使上面程序应用这种特征，对它做了三处修改：一是删除了对于/dev/zero的open语句，二是删除了对干fd的close语句，三是将mmap调用修改成

if ((area = mmap(0, SIZE, PROT\_READ |

PROT\_WRITE, MAP\_ANON | MAPSHARED, -1, 0)) == MAP\_FAILED)

的形式。在此调用中，指定了MAP\_ANON标志，并将文件描述符取为-1。程序的其余部分则没有改变。

最后两个例子说明了在多个相关进程之间如何使用共享存储段。如果在无关进程之间使用共享存储段，那么有两种替换的方法。其一是应用程序使用XSI共享存储函数，另一种是使用mmap将同一文件映射至它们的地址空间，为此使用MAP\_SHARED标志。

## 15.10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（通常称为XSI IPC），即消息队列、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（例如共享存储段）的同步访问。对于管道，我们说明了popen函数的实现，说明了协同进程,以及使用标准I/O库缓冲机制时可能遇到的问题。

将消息队列对全双工管道、信号量对记录锁等不同方法的耗时做了比较，然后提出了下列建议：要学会使用管道和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，而应当考虑全双工管道和记录锁，它们使用起来会简单得多。共享存储段有其应用场合，而mmap函数也能提供同样的功能。

下一章将介绍网络IPC，它们使进程能够跨越计算机的边界进行通信。