第七章　进程间通信

进程为了能在同一项任务上协调工作，它们彼此之间必须能够进行通信。例如，在一个 shell 管道中，第 1 个进程的输出必须传送到第 2 个进程，这样沿着管道传递下去。因此在需要通信的进程之间，最好使用一种结构较好的通信方式。

Linux 支持许多不同形式的进程间通信机制 CIPCC，在特定的情况下，它们各自有优缺点。这一章将讨论最有用的进程间通信机制，即管道、 System V 的 IPC 机制及信号。至于 Linux 支持的完全网络兼容的进程间通信机制 Sockets，将在第十三章网络部分中介绍。

本章介绍的管道通信中，要讨论匿名管道和命名管道两种方式；而在 System V 的 IPC 机制中，要讨论信号量、消息队列及共享内存 3 种通信方式。对于信号，则主要讨论 Linux 的信号机制。

本章将从内核和系统调用两个角度来讨论这些通信机制，以使你在写协调工作的并发进程时，可以作出明智的选择。

7.1　管道

在进程之间通信的最简单的方法是通过一个文件，其中有一个进程写文件，而另一个进程从文件中读，这种方法比较简单，其优点体现在：

- 只要进程对该文件具有访问权限，那么，两个进程间就可以进行通信；
- 进程之间传递的数据量可以非常大。

尽管如此，使用文件进行进程间通信也有两大缺点。

- 空间的浪费。写进程只有确保把新数据加到文件的尾部，才能使读进程读到数据，对长时间存在的进程来说，这就可能使文件变得非常大。
- 时间的浪费。如果读进程读数据比写进程写数据快，那么，就可能出现读进程不断地读文件尾部，使读进程做很多无用功。

要克服以上缺点而又使进程间的通信相对简单，管道是一种较好的选择。

所谓管道，是指用于连接一个读进程和一个写进程，以实现它们之间通信的共享文件，又称 pipe 文件。向管道（共享文件）提供输入的发送进程（即写进程），以字符流形式将大量的数据送入管道；而接收管道输出的接收进程（即读进程），可从管道中接收数据。由于发送进程和接收进程是利用管道进行通信的，故又称管道通信。这种方式首创于 UNIX 系统，因它能传送大量的数据，且很有效，故很多操作系统都引入了这种通信方式，Linux 也不例外。

为了协调双方的通信，管道通信机制必须提供以下 3 方面的协调能力。

- 互斥。当一个进程正在对 pipe 进行读/写操作时，另一个进程必须等待。

- 同步。当写（输入）进程把一定数量（如 4KB）数据写入 pipe 后，便去睡眠等待，直到读（输出）进程取走数据后，再把它唤醒。当读进程读到一空 pipe 时，也应睡眠等待，直至写进程将数据写入管道后，才将它唤醒。
- 对方是否存在。只有确定对方已存在时，才能进行通信。

7.1.1 Linux 管道的实现机制

在 Linux 中，管道是一种使用非常频繁的通信机制。从本质上说，管道也是一种文件，但它又和一般的文件有所不同，管道可以克服使用文件进行通信的两个问题，具体表现如下所述。

- 限制管道的大小。实际上，管道是一个固定大小的缓冲区。在 Linux 中，该缓冲区的大小为 1 页，即 4KB，使得它的大小不像文件那样不加检验地增长。使用单个固定缓冲区也会带来问题，比如在写管道时可能变满，当这种情况发生时，随后对管道的 write()调用将默认地被阻塞，等待某些数据被读取，以便腾出足够的空间供 write()调用写。
- 读取进程也可能工作得比写进程快。当所有当前进程数据已被读取时，管道变空。当这种情况发生时，一个随后的 read()调用将默认地被阻塞，等待某些数据被写入，这解决了 read()调用返回文件结束的问题。

注意，从管道读数据是一次性操作，数据一旦被读，它就从管道中被抛弃，释放空间以便写更多的数据。

1．管道的结构

在 Linux 中，管道的实现并没有使用专门的数据结构，而是借助了文件系统的 file 结构和 VFS 的索引节点 inode。通过将两个 file 结构指向同一个临时的 VFS 索引节点，而这个 VFS 索引节点又指向一个物理页面而实现的。如图 7.1 所示。

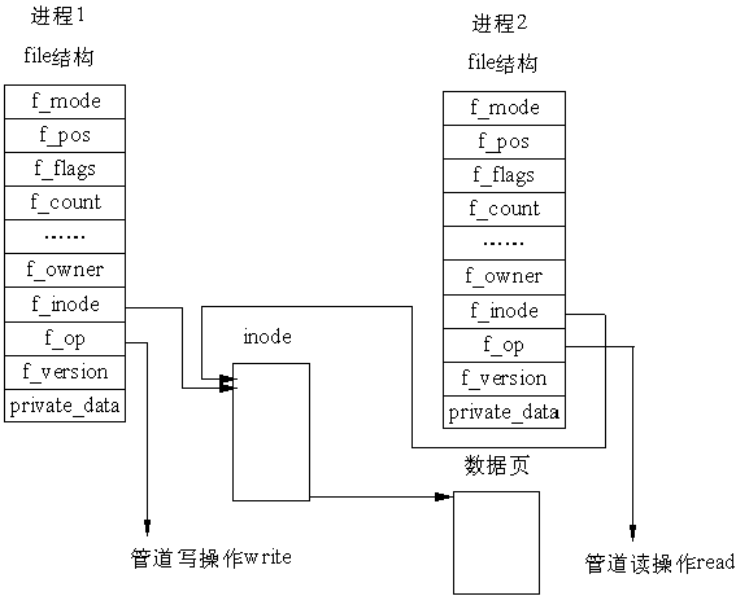

图 7.1　管道结构示意图

图 7.1 中有两个 file 数据结构，但它们定义文件操作例程地址是不同的，其中一个是向管道中写入数据的例程地址，而另一个是从管道中读出数据的例程地址。这样，用户程序的系统调用仍然是通常的文件操作，而内核却利用这种抽象机制实现了管道这一特殊操作。

2．管道的读写

管道实现的源代码在 fs/pipe.c 中，在 pipe.c 中有很多函数，其中有两个函数比较重要，即管道读函数 pipe_read()和管道写函数 pipe_wrtie()。管道写函数通过将字节复制到 VFS 索引节点指向的物理内存而写入数据，而管道读函数则通过复制物理内存中的字节而读出数据。当然，内核必须利用一定的机制同步对管道的访问，为此，内核使用了锁、等待队列和信号。

当写进程向管道中写入时，它利用标准的库函数 write()，系统根据库函数传递的文件描述符，可找到该文件的 file 结构。file 结构中指定了用来进行写操作的函数（即写入函数）地址，于是，内核调用该函数完成写操作。写入函数在向内存中写入数据之前，必须首先检查 VFS 索引节点中的信息，同时满足如下条件时，才能进行实际的内存复制工作：

- 内存中有足够的空间可容纳所有要写入的数据；
- 内存没有被读程序锁定。

如果同时满足上述条件，写入函数首先锁定内存，然后从写进程的地址空间中复制数据到内存。否则，写入进程就休眠在 VFS 索引节点的等待队列中，接下来，内核将调用调度程序，而调度程序会选择其他进程运行。写入进程实际处于可中断的等待状态，当内存中有足够的空间可以容纳写入数据，或内存被解锁时，读取进程会唤醒写入进程，这时，写入进程将接收到信号。当数据写入内存之后，内存被解锁，而所有休眠在索引节点的读取进程会被唤醒。

管道的读取过程和写入过程类似。但是，进程可以在没有数据或内存被锁定时立即返回错误信息，而不是阻塞该进程，这依赖于文件或管道的打开模式。反之，进程可以休眠在索引节点的等待队列中等待写入进程写入数据。当所有的进程完成了管道操作之后，管道的索引节点被丢弃，而共享数据页也被释放。

因为管道的实现涉及很多文件的操作，因此，当读者学完有关文件系统的内容后来读 pipe.c 中的代码，你会觉得并不难理解。

7.1.2　管道的应用

管道是利用 pipe()系统调用而不是利用 open()系统调用建立的。pipe()调用的原型是：

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

我们看到，有两个文件描述符与管道结合在一起，一个文件描述符用于管道的 read()端，一个文件描述符用于管道的 write()端。由于一个函数调用不能返回两个值，pipe()的参数是指向两个元素的整型数组的指针，它将由调用两个所要求的文件描述符填入。

fd[0]元素将含有管道 read()端的文件描述符，而 fd[1]含有管道 write()端的文件描述符。系统可根据 fd[0]和 fd[1]分别找到对应的 file 结构。在第八章我们会描述 pipe()系统

调用的实现机制。

注意，在 pipe 的参数中，没有路径名，这表明，创建管道并不像创建文件一样，要为它创建一个目录连接。这样做的好处是，其他现存的进程无法得到该管道的文件描述符，从而不能访问它。那么，两个进程如何使用一个管道来通信呢？

我们知道，fork()和 exec()系统调用可以保证文件描述符的复制品既可供双亲进程使用,也可供它的子女进程使用。也就是说,一个进程用 pipe()系统调用创建管道,然后用 fork()调用创建一个或多个进程，那么，管道的文件描述符将可供所有这些进程使用。pipe()系统调用的具体实现将在下一章介绍。

这里更明确的含义是：一个普通的管道仅可供具有共同祖先的两个进程之间共享，并且这个祖先必须已经建立了供它们使用的管道。

注意，在管道中的数据始终以和写数据相同的次序来进行读，这表示 lseek()系统调用对管道不起作用。

下面给出在两个进程之间设置和使用管道的简单程序：

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main (void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
       char    readbuffer[80];

       pipe (fd) ;

        if ( (childpid = fork())  == -1)
        {
                printf ("Error:fork") ;
                exit (1) ;
        }

        if (childpid == 0)          /* 子进程是管道的写进程 */
       {
               close (fd[0]) ;      /*关闭管道的读端  */
               write (fd[1], string, strlen (string) ) ;
               exit (0) ;
        }
        else                            /* 父进程是管道的读进程  */
        {
                   close (fd[1]) ;    /*关闭管道的写端 */
               nbytes = read (fd[0], readbuffer, sizeof (readbuffer) ) ;
                printf ("Received string: %s", readbuffer) ;
        }
        return (0) ;
}
```

注意，在这个例子中，为什么这两个进程都关闭它所不需的管道端呢？这是因为写进程完全关闭管道端时，文件结束的条件被正确地传递给读进程。而读进程完全关闭管道端时，

写进程无需等待继续写数据。

阻塞读和写分别成为对空和满管道的默认操作，这些默认操作也可以改变，这就需要调用 fcntl()系统调用，对管道文件描述符设置 O_NONBLOCK 标志可以忽略默认操作：

```
# include  <fcntl.h>

fcntl (fd,F_SETFL,O_NONBlOCK) ;
```

7.1.3　命名管道 CFIFOC

Linux 还支持另外一种管道形式，称为命名管道，或 FIFO，这是因为这种管道的操作方式基于“先进先出”原理。上面讲述的管道类型也被称为“匿名管道”。命名管道中，首先写入管道的数据是首先被读出的数据。匿名管道是临时对象，而 FIFO 则是文件系统的真正实体，如果进程有足够的权限就可以使用 FIFO。FIFO 和匿名管道的数据结构以及操作极其类似，二者的主要区别在于，FIFO 在使用之前就已经存在，用户可打开或关闭 FIFO；而匿名管道只在操作时存在，因而是临时对象。

为了创建先进先出文件，可以从 shell 提示符使用 mknod 命令或可以在程序中使用 mknod()系统调用。

mknod()系统调用的原型为：

```
#include  <sys/type.h>
#inlcude  <sys/state.h>
#include  <fcntl.h>
#include  <unistd.h>

int mknod (char *pathname,node_t mode, dev_t dev) ;
```

其中 pathname 是被创建的文件名称，mode 表示将在该文件上设置的权限位和将被创建的文件类型（在此情况下为 S_IFIFO），dev 是当创建设备特殊文件时使用的一个值。因此，对于先进先出文件它的值为 0。

一旦先进先出文件已经被创建，它可以由任何具有适当权限的进程利用标准的 open()系统调用加以访问。当用 open()调用打开时，一个先进先出文件和一个匿名管道具有同样的基本功能。即当管道是空的时候，read()调用被阻塞。当管道是满的时候，write()等待被阻塞，并且当用 fcntl()设置 O_NONBLOCK 标志时，将引起 read()调用和 write()调用立即返回。在它们已被阻塞的情况下，带有一个 EAGAIN 错误信息。

由于命名管道可以被很多无关系的进程同时访问，那么，在有多个读进程和/或多个写进程的应用中使用 FIFO 是非常有用的。

多个进程写一个管道会出现这样的问题，即多个进程所写的数据混在一起怎么办？幸好系统有这样的规则：一个 write()调用可以写管道能容纳（Linux 为 4KB）的任意个字节，系统将保证这些数据是分开的。这表示多个写操作的数据在 FIFO 文件中并不混合而将被维持分离的信息。

7.2 信号（signal）

尽管大多数进程间通信是计划好的，但同时还需要处理不可预知的通信问题。例如，用户使用文本编辑器要求列出一个大文件的全部内容，但随即他认识到该操作并不重要，这时就需要一种方法来中止编辑器的工作，例如，用户可以通过 DEL 键作到这点，按 DEL 键实际上是向编辑器发送一个信号，编辑器收到此信号即停止打印文件的内容。信号还可用来报告硬件捕获到的特定的陷入，如非法指令或浮点运算溢出，超时也是通过信号来实现的。

实际上，信号机制是在软件层次上对中断机制的模拟。从概念上讲，一个进程接受到一个信号与一个处理器接受到一个中断请求是一样的。一个进程所接收到的信号可以来自其他进程，可以来自外部事件，也可以来自进程自身。最重要的是，信号和中断都是“异步”的。处理器在执行一段程序时并不需要停下来等待中断的发生，也不知道中断会何时发生。信号也一样，一个进程并不需要通过一个什么样的操作来等待信号的到达，也不知道信号会什么时候到达。

7.2.1 信号种类

每一种信号都给予一个符号名，对 32 位的 i386 平台而言，一个字为 32 位，因此信号有 32 种，而对 64 位的 Alpha AXP 平台而言，每个字为 64 位，因此信号最多可有 64 种。Linux 定义了 i386 的 32 个信号，在 include/asm/signal.h 中定义。表 7.1 给出常用的符号名、描述和它们的信号值。

表 7.1 信号和其对应的值

符号名	描述	信号值
SIGHUP	在控制终端上发生的结束信号	1
SIGINT	中断，用户键入 CTRL－C 时发送	2
SIGQUIT	从键盘来的中断（ctrl_c）信号	3
SIGILL	非法指令	4
SIGTRAP	跟踪陷入	5
SIGABRT	非正常结束，程序调用 abort 时发送	6
SIGIOT	IOT 指令	6
SIGBUS	总线超时	7
SIGFPE	浮点异常	8
SIGKILL	杀死进程（不能被捕或忽略）	9
SIGUSR1	用户定义信号#1	10
SIGSEGV	段违法	11
SIGUSR2	用户定义信号#2	12
SIGPIPE	向无人读到的管道写	13
SIGALRM	定时器告警，时间到	14

SIGTERM	Kill 发出的软件结束信号	15
SIGCHLD	子程序结束或停止	17
SIGCONT	如果已停止则续继	18
SIGSTOP	停止信号	19

续表

符号名	描述	信号值
SIGTSTP	交互停止信号	20
SIGTTIN	后台进程想读	21
SIGTTOU	后台进程想写	22
SIGPWR	电源失效	30

每种信号类型都有对应的信号处理程序（也叫信号的操作），就好像每个中断都有一个中断服务例程一样。大多数信号的默认操作是结束接收信号的进程。然而，一个进程通常可以请求系统采取某些代替的操作，各种代替操作如下所述。

（1）忽略信号。随着这一选项的设置，进程将忽略信号的出现。有两个信号不可以被忽略：SIGKILL，它将结束进程；SIGSTOP，它是作业控制机制的一部分，将挂起作业的执行。

（2）恢复信号的默认操作。

（3）执行一个预先安排的信号处理函数。进程可以登记特殊的信号处理函数。当进程收到信号时，信号处理函数将像中断服务例程一样被调用，当从该信号处理函数返回时，控制被返回给主程序，并且继续正常执行。

但是，信号和中断有所不同。中断的响应和处理都发生在内核空间，而信号的响应发生在内核空间，信号处理程序的执行却发生在用户空间。

那么，什么时候检测和响应信号呢？通常发生在以下两种情况下：

（1）当前进程由于系统调用、中断或异常而进入内核空间以后，从内核空间返回到用户空间前夕；

（2）当前进程在内核中进入睡眠以后刚被唤醒的时候，由于检测到信号的存在而提前返回到用户空间。

当有信号要响应时，处理器执行路线的示意图如图 7.2 所示。

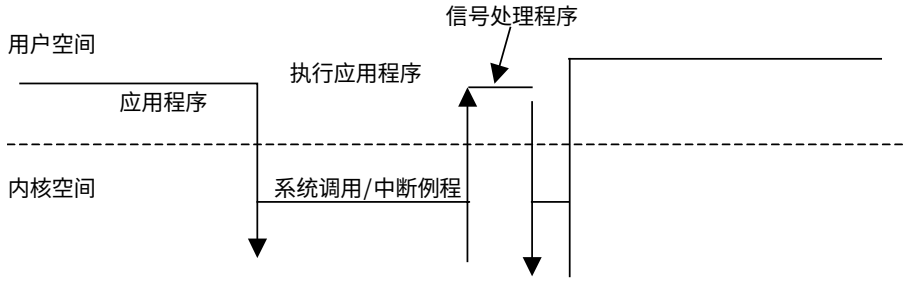

图 7.2　信号的检测及处理流程示意图

从图 7.2 中可以看出，当前进程在用户态执行的过程中，陷入系统调用或中断服务例程，

于是，当前进程从用户态切换到内核态；当处理完系统调用要返回到用户态前夕，发现有信号处理程序需要执行，于是，又从内核态切换到用户态；当执行完信号处理程序后，并不是接着就在用户态执行应用程序，而是还要返回到内核态。为什么还要返回到内核态呢？这是因为此时还没有真正从系统调用返回到用户态，于是从信号处理程序返回到内核态就是为了处理从系统调用到用户态的返回。读者能否想出更好的办法来处理这种状态的来回切换呢。

7.2.2 信号掩码

在 POSIX 下，每个进程有一个信号掩码（Signal Mask）。简单地说，信号掩码是一个“位图”，其中每一位都对应着一种信号。如果位图中的某一位为 1，就表示在执行当前信号的处理程序期间相应的信号暂时被“屏蔽”，使得在执行的过程中不会嵌套地响应那种信号。

为什么对某一信号进行屏蔽呢？我们来看一下对 CTRL+C 的处理。大家知道，当一个程序正在运行时，在键盘上按一下 CTRL+C，内核就会向相应的进程发出一个 SIGINT 信号，而对这个信号的默认操作就是通过 do_exit()结束该进程的运行。但是，有些应用程序可能对 CTRL+C 有自己的处理，所以就要为 SIGINT 另行设置一个处理程序，使它指向应用程序中的一个函数，在那个函数中对 CTRL+C 这个事件作出响应。但是，在实践中却发现，两次 CTRL+C 事件往往过于密集，有时候刚刚进入第 1 个信号的处理程序，第 2 个 SIGINT 信号就到达了，而第 2 个信号的默认操作是杀死进程，这样，第 1 个信号的处理程序根本没有执行完。为了避免这种情况的出现，就在执行一个信号处理程序的过程中将该种信号自动屏蔽掉。所谓“屏蔽”，与将信号忽略是不同的，它只是将信号暂时“遮盖”一下，一旦屏蔽去掉，已到达的信号又继续得到处理。

Linux 内核中有一个专门的函数集合来执行设置和修改信号掩码，它们放在 kernel/signal.c 中，其函数形式和功能如下：

函数形式	功能
int sigemptyset（sigset_t *mask）	清所有信号掩码的阻塞标志
int sigfillset（sigset_t *mask，int signum）	设置所有信号掩码的阻塞标志
int sigdelset（sigset_t *mask，int signum）	删除个别信号阻塞
int sigaddset（sigset_t *mask，int signum）	增加个别信号阻塞
int sigisnumber（sigset_t *mask，int signum）	确定特定的信号是否在掩码中被标志为阻塞

另外，进程也可以利用 sigprocmask()系统调用改变和检查自己的信号掩码的值，其实现代码在 kernel/signal.c 中，原型为：

```
int sys_sigprocmask（int how，sigset_t *set，sigset_t *oset）
```

其中，set 是指向信号掩码的指针，进程的信号掩码是根据参数 how 的取值设置成 set。参数 how 的取值及含义如下：

SIG_BOLCK	set 规定附加的阻塞信号
SIG_UNBOCK	set 规定一组不予阻塞的信号
SIG_SETBLOCK	set 变成新进程的信号掩码

用一段代码来说明这个问题：

```
switch (how) {
case SIG_BLOCK:
       current->blocked |= new_set;
           break;
 case SIG_UNBLOCK:
          current->blocked &= ~new_set;
          break;
 case SIG_SETMASK:
          current->blocked = new_set;
          break;
 default:
     return -EINVAL;
    }
```

其中 current 为指向当前进程 task_struc 结构的指针。

第 3 个参数 oset 也是指向信号掩码的指针，它将包含以前的信号掩码值，使得在必要的时候，可以恢复它。

进程可以用 sigpending()系统调用来检查是否有挂起的阻塞信号。

7.2.3　系统调用

除了 signal()系统调用，Linux 还提供关于信号的系统调用如下：

调用原型	功能
int sigaction (sig,&handler,&oldhandler)	定义对信号的处理操作
int sigreturn (&context)	从信号返回
int sigprocmask (int how, sigset_t *mask, sigset_t *old)	检查或修改信号屏蔽
int sigpending (sigset_t mask)	替换信号掩码并使进程挂起
int kill (pid_t pid, int sig)	发送信号到进程
long alarm (long secs)	设置事件闹钟
int pause (void)	将调用进程挂起直到下一个进程

其中 sigset_t 定义为：

```
typedef unsigned long sigset_t;       /* 至少 32 位*/
```

下面介绍几个典型的系统调用。

1. ill 系统调用

从前面的叙述可以看到，一个进程接收到的信号，或者是由异常的错误产生（如浮点异常），或者是用户在键盘上用中断和退出信号干涉而产生，那么，一个进程能否给另一个进程发送信号？回答是肯定的，但发送者进程必须有适当的权限。Kill()系统调用可以完成此任务：

```
int kill (pid_t pid, int sig)
```

参数 sig 规定发送哪一个信号，参数 pid（进程标识号）规定把信号发送到何处，pid

各种不同值具有下列意义：

pid>0　信号 sig 发送给进程标识号为 pid 的进程；

pid=0　设调用 kill()的进程其组标识号为 p，则把信号 sig 发送给与 p 相等的其他所有进程；

pid=-1　inux 规定把信号 sig 发送给系统中除去 init 进程和调用者以外的所有进程；

pid<-1　信号发送给进程组-pid 中的所有进程。

为了用 kill()发送信号，调用进程的有效用户 ID 必须是 root，或者必须和接收进程的实际或有效用户 ID 相同。

2．ause()和 alarm()系统调用

当一个进程需要等待另一个进程完成某项操作时，它将执行 pause()调用，当这项操作已完成时，另一个进程可以发送一个预约的信号给这一暂停的进程，它将强迫 pause()返回，并且允许收到信号的进程恢复执行，知道它正在等待的事件现在已经出现。

对于许多实际应用，需要在一段指定时间后，中断进程的原有操作，以进行某种其他的处理，例如在不可靠的通信线路上重传一个丢失的包，为了处理此类情况，系统提供了 alarm()系统调用。每个进程都有一个闹钟计时器与之相联，在经过预先设置的时间后，进程可以用它来给自己发送 SIGALARM 信号。Alarm()调用只取一个参数 secs，它是在闹钟关闭之前所经过的秒数。如果传递一个 0 值给 alarm()，这将关闭任何当前正在运行的闹钟计时器。

Alarm()返回值是以前的闹钟计时器值，如果当前没有设置任何闹钟计时器，这将是零，或者是当作出该调用时，闹钟的剩余时间。

某些情况下，进程在信号到达之前不要做任何操作，例如一个测验阅读和理解速度的 CAI 系统，它先在屏幕上显示一些课文，然后调用 alarm()设定在 30s 后向自己发送一个信号，以激活程序进行一些处理。当学生阅读课文时，程序无需执行任何操作，它可以采用的一种方法是执行空操作循环以等待时间到，但假如此时系统中还有其他进程运行，这将浪费 CPU 的时间，好的方法是使用 pause()系统调用，它将挂起调用进程直到信号到来，这段时间里别的进程可以使用 CPU。

7.2.4　典型系统调用的实现

sigaction()系统调用的实现较具代表性，它的主要功能为设置信号处理程序，其原型为：

```
int sys_sigaction (int signum, const struct sigaction * action,
                   struct sigaction * oldaction)
```

其中，sigaction 数据结构在 include/asm/signal.h 中定义，其格式为：

```
struct sigaction {
        __sighandler_t  sa_handler;
    sigset_t       sa_mask;
          unsigned long  sa_flags;
        void  (*sa_restorer)  (void) ;
           };
```

其中__sighandler_t 定义为：

```
typedef void (*__sighandler_t) (int);
```

在这个结构中， sa_handler 为指向处理函数的指针，sa_mask 是信号掩码，当该信号 signum 出现时，这个掩码就被逻辑或到接收进程的信号掩码中。当信号处理程序执行时，这个掩码保持有效。Sa_flags 域是几个位标志的逻辑或（OR）组合，其中两个主要的标志是：

SA_ONESHOT　信号出现时，将信号操作置为默认操作；

SA_NOMASK　忽略 sigaction 结构的 sa_mask 域。

Linux 中定义的信号处理的 3 种类型为：

```
#define SIG_DFL ((__sighandler_t) 0)       * 缺省的信号处理*/
#define SIG_IGN ((__sighandler_t) 1)       *忽略这个信号 */
#define SIG_ERR ((__sighandler_t) -1) *从信号返回错误 */
```

下面是 sigaction()系统调用在内核中实现的代码及解释。

```
  int sys_sigaction (int signum, const struct sigaction * action,
                     struct sigaction * oldaction)
  {
      struct sigaction new_sa, *p;

        if (signum<1 || signum>32)
             return -EINVAL;
        /* 信号的值不在 1～32 之间，则出错 */
        if (signum==SIGKILL || signum==SIGSTOP)
             return -EINVAL;
        /*SIGKILL 和 SIGSTOP 不能设置信号处理程序 */
         p = signum - 1 + current->sig->action;
            /*在当前进程中，指向信号 signum 的 action 的指针 */
         if (action) {
       int err = verify_area (VERIFY_READ, action, sizeof (*action) ) ;
           /* 验证给 action 在用户空间分配的地址的有效性 */
            if (err)
                 return err;
       memcpy_fromfs (&new_sa, action, sizeof (struct sigaction) ) ;
          /* 把 actoin 的内容从用户空间拷贝到内核空间*/
         new_sa.sa_mask |= _S (signum) ;
          /* 把信号 signum 加到掩码中 */
        if (new_sa.sa_flags & SA_NOMASK)
            new_sa.sa_mask &= ~_S (signum) ;
/* 如果标志为 SA_NOMASK，当信号 signum 出现时，将它的操作置为默认操作 */
          new_sa.sa_mask &= _BLOCKABLE;
          /* 不能阻塞 SIGKILL 和 SIGSTOP  */
if (new_sa.sa_handler != SIG_DFL && new_sa.sa_handler !=SIG_IGN) {
             err = verify_area (VERIFY_READ, new_sa.sa_handler, 1) ;
/* 当处理程序不是信号默认的处理操作，并且 signum 信号不能被忽略时，验证给信号处理程序分配空间的有效性 */
          if (err)
                 return err;       }
          }
      if (oldaction) {
```

```
int err = verify_area (VERIFY_WRITE, oldaction,   sizeof (*oldaction) ) ;
    if  (err)
              return err;
    memcpy_tofs (oldaction, p, sizeof (struct sigaction) ) ;
     /* 恢复原来的信号处理程序 */
                  }
    if  (action)  {
       *p = new_sa;
       check_pending (signum) ;
             }
        return 0;
  }
```

Linux 可以将各种信号发送给程序，以表示程序故障、用户请求的中断、其他各种情况等。通过对 sigaction()系统调用源代码的分析，有助于灵活应用信号的系统调用。

7.2.5 进程与信号的关系

Linux 内核中不存在任何机制用来区分不同信号的优先级。也就是说，当同时有多个信号发出时，进程可能会以任意顺序接收到信号并进行处理。另外，如果进程在处理某个信号之前，又有相同的信号发出，则进程只能接收到一个信号。产生上述现象的原因与内核对信号的实现有关。

系统在 task_struct 结构中利用两个域分别记录当前挂起的信号（Signal）以及当前阻塞的信号（Blocked）。挂起的信号指尚未进行处理的信号。阻塞的信号指进程当前不处理的信号，如果产生了某个当前被阻塞的信号，则该信号会一直保持挂起，直到该信号不再被阻塞为止。除了 SIGKILL 和 SIGSTOP 信号外，所有的信号均可以被阻塞，信号的阻塞可通过系统调用 sigprocmask()实现。每个进程的 task_struct 结构中还包含了一个指向 sigaction 结构数组的指针，该结构数组中的信息实际指定了进程处理所有信号的方式。如果某个 sigaction 结构中包含有处理信号的例程地址，则由该处理例程处理该信号；反之，则根据结构中的一个标志或者由内核进行默认处理，或者只是忽略该信号。通过系统调用 sigaction()，进程可以修改 sigaction 结构数组的信息，从而指定进程处理信号的方式。

进程不能向系统中所有的进程发送信号，一般而言，除系统和超级用户外，普通进程只能向具有相同 uid 和 gid 的进程，或者处于同一进程组的进程发送信号。当有信号产生时，内核将进程 task_struct 的 signal 字中的相应位设置为 1。系统不对置位之前该位已经为 1 的情况进行处理，因而进程无法接收到前一次信号。如果进程当前没有阻塞该信号，并且进程正处于可中断的等待状态（INTERRUPTIBLE），则内核将该进程的状态改变为运行（RUNNING），并放置在运行队列中。这样，调度程序在进行调度时，就有可能选择该进程运行，从而可以让进程处理该信号。

发送给某个进程的信号并不会立即得到处理，相反，只有该进程再次运行时，才有机会处理该信号。每次进程从系统调用中退出时，内核会检查它的 signal 和 block 字段，如果有任何一个未被阻塞的信号发出，内核就根据 sigaction 结构数组中的信息进行处理。处理过程如下。

（1）检查对应的 sigaction 结构，如果该信号不是 SIGKILL 或 SIGSTOP 信号，且被忽略，则不处理该信号。

（2）如果该信号利用默认的处理程序处理，则由内核处理该信号，否则转向第（3）步。

（3）该信号由进程自己的处理程序处理，内核将修改当前进程的调用堆栈，并将进程的程序计数寄存器修改为信号处理程序的入口地址。此后，指令将跳转到信号处理程序，当从信号处理程序中返回时，实际就返回了进程的用户模式部分。

Linux 是 与 POSIX 兼容的，因此，进程在处理某个信号时，还可以修改进程的 blocked 掩码。但是，当信号处理程序返回时，blocked 值必须恢复为原有的掩码值，这一任务由内核的 sigaction()函数完成。Linux 在进程的调用堆栈帧中添加了对清理程序的调用，该清理程序可以恢复原有的 blocked 掩码值。当内核在处理信号时，可能同时有多个信号需要由用户处理程序处理，这时，Linux 内核可以将所有的信号处理程序地址推入堆栈中，而当所有的信号处理完毕后，调用清理程序恢复原先的 blocked 值。

7.2.6　信号举例

下面通过 Linux 提供的系统调用 signal()，来说明如何执行一个预先安排好的信号处理函数。Signal()调用的原型是：

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

void  (* signal (int signum, void (*handler)  (int) ) )  (int) ;
```

signal()的返回值是指向一个函数的指针，该函数的参数为一个整数，无返回值，下面是用户级程序的一段代码。

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

int ctrl_c_count=0;
void  (* old_handler)  (INT) ;
void ctrl_c (int) ;

main()
{
   int c;

   old_handler = signal (SIGINT,ctrl_c) ;

       while  ( (c=getchar()) ! = '\n') ;

       printf ("ctrl-c count = %d\n",ctrl_c_count) ;

        (void)  signal (SIGINT,old_handler) ;
     }

   void ctrl_c (int signum)
```

```
    {
       (void) signal (SIGINT,ctrl_c)
      ++ctrl_c;
 }
```

程序说明：这个程序是从键盘获得字符，直到换行符为止，然后进入无限循环。这里，程序安排了捕获 ctrl_c 信号（SIGINT），并且利用 SIGINT 来执行一个 ctrl_c 的处理函数。当在键盘上敲入一个换行符时，SIGINT 原来的操作（很可能是默认操作）才被恢复。Main()函数中的第一个语句完成设置信号处理程序：

```
    old_handler = signal (SIGINT,ctrl_c) ;
```

signal()的两个参数是：信号值，这里是键盘中断信号 SIGINT，以及一个指向函数的指针，这里是 ctrl_c，当这个中断信号出现时，将调用该函数。Signal()调用返回旧的信号处理程序的地址，在此它被赋给变量 older_handler，使得原来的信号处理程序稍后可以被恢复。

一旦信号处理程序放在应放的位置，进程收到任何中断（SIGINT）信号将引起信号处理函数的执行。这个函数增加 ctrl_c_count 变量的值以保持对 SIGINT 事件出现次数的计数。注意信号处理函数也执行另一个 signal()调用，它重新建立 SIGINT 信号和 ctrl_c 函数之间的联系。这是必需的，因为当信号出现时，用 signal()调用设置的信号处理程序被自动恢复为默认操作，使得随后的同一信号将只执行信号的默认操作。

7.3 System V 的 IPC 机制

为了提供与其他系统的兼容性，Linux 也支持 3 种 system V 的进程间通信机制：消息、信号量（semaphores）和共享内存，Linux 对这些机制的实施大同小异。我们把信号量、消息和共享内存统称 System V IPC 的对象，每一个对象都具有同样类型的接口，即系统调用。就像每个文件都有一个打开文件号一样，每个对象也都有唯一的识别号，进程可以通过系统调用传递的识别号来存取这些对象，与文件的存取一样，对这些对象的存取也要验证存取权限，System V IPC 可以通过系统调用对对象的创建者设置这些对象的存取权限。

在 Linux 内核中，System V IPC 的所有对象有一个公共的数据结构 pc_perm 结构，它是 IPC 对象的权限描述，在 linux/ipc.h 中定义如下：

```
struct ipc_perm
{
   key_t  key;     /* 键 */
   ushort uid;     /* 对象拥有者对应进程的有效用户识别号和有效组识别号 */
   ushort gid;
  ushort cuid;     /* 对象创建者对应进程的有效用户识别号和有效组识别号 */
   ushort cgid;
   ushort mode;         /*  存取模式 */
   ushort seq;     /* 序列号 */
};
```

在这个结构中，要进一步说明的是键（key）。键和识别号指的是不同的东西。系统支持两种键：公有和私有。如果键是公有的，则系统中所有的进程通过权限检查后，均可以找到 System V IPC 对象的识别号。如果键是公有的，则键值为 0，说明每个进程都可以用键值

0 建立一个专供其私用的对象。注意，对 System V IPC 对象的引用是通过识别号而不是通过键，从后面的系统调用中可了解这一点。

7.3.1　信号量

信号量及信号量上的操作是 E.W.Dijkstra 在 1965 年提出的一种解决同步、互斥问题的较通用的方法，并在很多操作系统中得以实现，Linux 改进并实现了这种机制。

信号量（semaphore）实际是一个整数，它的值由多个进程进行测试（test）和设置（set）。就每个进程所关心的测试和设置操作而言，这两个操作是不可中断的，或称“原子”操作，即一旦开始直到两个操作全部完成。测试和设置操作的结果是：信号量的当前值和设置值相加，其和或者是正或者为负。根据测试和设置操作的结果，一个进程可能必须睡眠，直到有另一个进程改变信号量的值。

信号量可用来实现所谓的“临界区”的互斥使用，临界区指同一时刻只能有一个进程执行其中代码的代码段。为了进一步理解信号量的使用，下面我们举例说明。

假设你有很多相互协作的进程，它们正在读或写一个数据文件中的记录。你可能希望严格协调对这个文件的存取，于是你使用初始值为 1 的信号量，在这个信号量上实施两个操作，首先测试并且给信号量的值减 1，然后测试并给信号量的值加 1。当第 1 个进程存取文件时，它把信号量的值减 1，并获得成功，信号量的值现在变为 0，这个进程可以继续执行并存取数据文件。但是，如果另外一个进程也希望存取这个文件，那么它也把信号量的值减 1，结果是不能存取这个文件，因为信号量的值变为-1。这个进程将被挂起，直到第一个进程完成对数据文件的存取。当第 1 个进程完成对数据文件的存取，它将增加信号量的值，使它重新变为 1，现在，等待的进程被唤醒，它对信号量的减 1 操作将获得成功。

上述的进程互斥问题，是针对进程之间要共享一个临界资源而言的，信号量的初值为 1。实际上，信号量作为资源计数器，它的初值可以是任何正整数，其初值不一定为 0 或 1。另外，如果一个进程要先获得两个或多个的共享资源后才能执行的话，那么，相应地也需要多个信号量，而多个进程要分别获得多个临界资源后方能运行，这就是信号量集合机制，Linux 讨论的就是信号量集合问题。

1．信号量的数据结构

Linux 中信号量是通过内核提供的一系列数据结构实现的，这些数据结构存在于内核空间，对它们的分析是充分理解信号量及利用信号量实现进程间通信的基础，下面先给出信号量的数据结构（存在于 include/linux/sem.h 中），其他一些数据结构将在相关的系统调用中介绍。

(1) 系统中每个信号量的数据结构（sem）

```
struct sem {
  int  semval;          /* 信号量的当前值  */
           int   sempid;          /*在信号量上最后一次操作的进程识别号 *
        };
```

(2) 系统中表示信号量集合（set）的数据结构（semid_ds）

```
struct semid_ds {
```

```
    struct ipc_perm sem_perm;        /*  IPC 权限 */
    long        sem_otime;               /* 最后一次对信号量操作（semop）的时间 */
  long        sem_ctime;             /* 对这个结构最后一次修改的时间 */
    struct sem      *sem_base;       /* 在信号量数组中指向第一个信号量的指针 */
    struct sem_queue *sem_pending;            /* 待处理的挂起操作*/
    struct sem_queue **sem_pending_last;  /* 最后一个挂起操作 */
    struct sem_undo *undo;                    /* 在这个数组上的 undo 请求 */
  ushort          sem_nsems;                      /* 在信号量数组上的信号量号 */
    };
```

(3) 系统中每一信号量集合的队列结构（sem_queue）

```
struct sem_queue {
    struct sem_queue *  next;     /* 队列中下一个节点 */
    struct sem_queue ** prev;      /* 队列中前一个节点，* (q->prev)  == q */
    struct wait_queue * sleeper;     /* 正在睡眠的进程 */
    struct sem_undo *   undo;      /* undo 结构*/
    int            pid;  /* 请求进程的进程识别号 */
    int            status;    /* 操作的完成状态 */
    struct semid_ds *   sma;  /*有操作的信号量集合数组 */
    struct sembuf *     sops;      /* 挂起操作的数组 */
    int       nsops;     /* 操作的个数 */
};
```

(4) 几个主要数据结构之间的关系

从图 7.3 可以看出，semid_ds 结构的 sem_base 指向一个信号量数组，允许操作这些信号量集合的进程可以利用系统调用执行操作 。注意，信号量与信号量集合的区别，从上面可以看出，信号量用“ sem ” 结构描述，而信号量集合用“ semid_ds ”结构描述，实际上，在后面的讨论中，我们以信号量集合为讨论的主要对象。下面我们给出这几个结构之间的关系，如图 7.3 所示。

Linux 对信号量的这种实现机制，是为了与消息和共享内存的实现机制保持一致，但信号量是这 3 者中最难理解的，因此我们将结合系统调用做进一步的介绍，通过对系统调用的深入分析，我们可以较清楚地了解内核对信号量的实现机制。

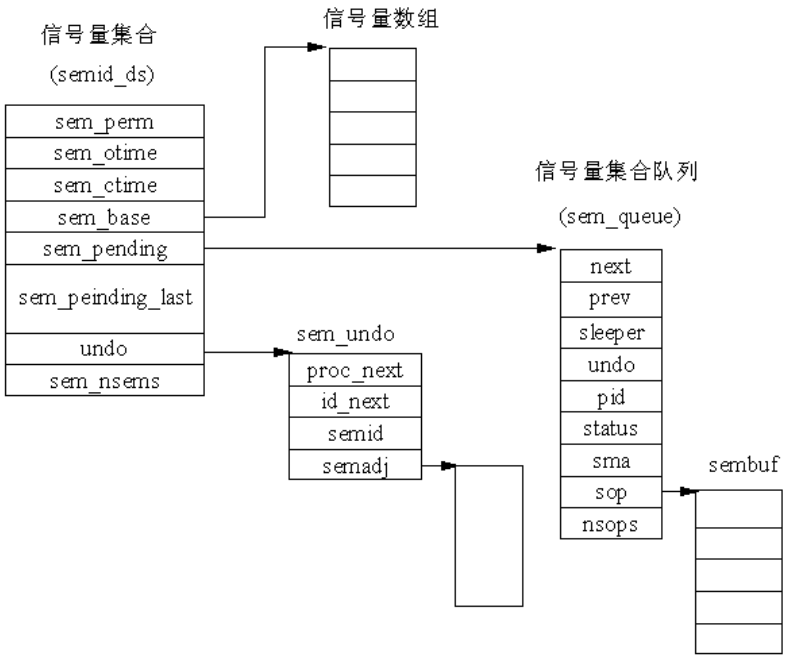

图 7.3　System V IPC 信号量数据结构之间的关系

2．系统调用：semget()

为了创建一个新的信号量集合，或者存取一个已存在的集合，要使用 segget()系统调用，其描述如下：

原型：int semget （ key_t key, int nsems, int semflg ）；

返回值：如果成功，则返回信号量集合的 IPC 识别号；

如果为-1，则出现错误。

semget()中的第 1 个参数是键值，这个键值要与已有的键值进行比较，已有的键值指在内核中已存在的其他信号量集合的键值。对信号量集合的打开或存取操作依赖于 semflg 参数的取值。

- IPC_CREAT ：如果内核中没有新创建的信号量集合，则创建它。
- IPC_EXCL ：当与 IPC_CREAT 一起使用时，如果信号量集合已经存在，则创建失败。

如果 IPC_CREAT 单独使用，semget()为一个新创建的集合返回标识号，或者返回具有相同键值的已存在集合的标识号。如果 IPC_EXCL 与 IPC_CREAT 一起使用，要么创建一个新的集合，要么对已存在的集合返回-1。IPC_EXCL 单独是没有用的，当与 IPC_CREAT 结合起来使用时，可以保证新创建集合的打开和存取。

作为 System V IPC 的其他形式，一种可选项是把一个八进制与掩码或，形成信号量集合的存取权限。

第 3 个参数 nsems 指的是在新创建的集合中信号量的个数。其最大值在“ linux/sem.h ”中定义：

```
#define SEMMSL  250             /* <= 8 000 max num of semaphores per id */
```

注意，如果你是显式地打开一个现有的集合，则 nsems 参数可以忽略。

下面举例说明。

```
int open_semaphore_set ( key_t keyval, int numsems )
{
        int     sid;

        if  ( ! numsems )
                return (-1) ;

        if ( (sid = semget ( keyval, numsems, IPC_CREAT | 0660 ) )  == -1)
        {
                return (-1) ;
        }

        return (sid) ;
}
```

注意，这个例子显式地用了 0660 权限。这个函数要么返回一个集合的标识号，要么返回-1 而出错。键值必须传递给它，信号量的个数也传递给它，这是因为如果创建成功则要分配空间。

3．系统调用：semop()

原型： int semop （ int semid, struct sembuf *sops, unsigned nsops）;

返回：如果所有的操作都执行，则成功返回 0。

如果为-1，则出错。

semop()中的第 1 个参数（semid）是集合的识别号（可以由 semget()系统调用得到）。第 2 个参数（sops）是一个指针，它指向在集合上执行操作的数组。而第 3 个参数（nsops）是在那个数组上操作的个数。

sops 参数指向类型为 sembuf 的一个数组，这个结构在/inclide/linux/sem.h 中声明，是内核中的一个数据结构，描述如下：

```
        struct sembuf {
          ushort  sem_num;        /* 在数组中信号量的索引值 */
          short   sem_op;         /* 信号量操作值（正数、负数或 0） */
          short   sem_flg;        /* 操作标志，为 IPC_NOWAIT 或 SEM_UNDO*/
        };
```

如果 sem_op 为负数，那么就从信号量的值中减去 sem_op 的绝对值，这意味着进程要获取资源，这些资源是由信号量控制或监控来存取的。如果没有指定 IPC_NOWAIT，那么调用进程睡眠到请求的资源数得到满足（其他的进程可能释放一些资源）。

如果 sem_op 是正数，把它的值加到信号量，这意味着把资源归还给应用程序的集合。

最后，如果 sem_op 为 0，那么调用进程将睡眠到信号量的值也为 0，这相当于一个信号量到达了 100%的利用。

综上所述，Linux 按如下的规则判断是否所有的操作都可以成功：操作值和信号量的当前值相加大于 0，或操作值和当前值均为 0，则操作成功。如果系统调用中指定的所有操作中有一个操作不能成功时，则 Linux 会挂起这一进程。但是，如果操作标志指定这种

情况下不能挂起进程的话，系统调用返回并指明信号量上的操作没有成功，而进程可以继续执行。如果进程被挂起，Linux 必须保存信号量的操作状态并将当前进程放入等待队列。为此，Linux 内核在堆栈中建立一个 sem_queue 结构并填充该结构。新的 sem_queue 结构添加到集合的等待队列中（利用 sem_pending 和 sem_pending_last 指针）。当前进程放入 sem_queue 结构的等待队列中（sleeper）后调用调度程序选择其他的进程运行。

为了进一步解释 semop()调用，让我们来看一个例子。假设我们有一台打印机，一次只能打印一个作业。我们创建一个只有一个信号量的集合（仅一个打印机），并且给信号量的初值为 1（因为一次只能有一个作业）。

每当我们希望把一个作业发送给打印机时，首先要确定这个资源是可用的，可以通过从信号量中获得一个单位而达到此目的。让我们装载一个 sembuf 数组来执行这个操作：

```
struct sembuf sem_lock = { 0, -1, IPC_NOWAIT };
```

从这个初始化结构可以看出，0 表示集合中信号量数组的索引，即在集合中只有一个信号量，-1 表示信号量操作（sem_op），操作标志为 IPC_NOWAIT，表示或者调用进程不用等待可立即执行，或者失败（另一个进程正在打印）。下面是用初始化的 sembuf 结构进行 semop()系统调用的例子：

```
if ( (semop (sid, &sem_lock, 1)  == -1)
        fprintf (stderr,"semop\n") ;
```

第 3 个参数（nsops）是说我们仅仅执行了一个操作（在我们的操作数组中只有一个 sembuf 结构），sid 参数是我们集合的 IPC 识别号。

当我们使用完打印机，我们必须把资源返回给集合，以便其他的进程使用。

```
struct sembuf sem_unlock = { 0, 1, IPC_NOWAIT };
```

上面这个初始化结构表示，把 1 加到集合数组的第 0 个元素，换句话说，一个单位资源返回给集合。

4．系统调用 ：semctl()

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

返回值： 成功返回正数，出错返回-1。

注意，semctl()是在集合上执行控制操作。

semctl()的第 1 个参数（semid）是集合的标识号，第 2 个参数（semnum）是将要操作的信号量个数，从本质上说，它是集合的一个索引，对于集合上的第一个信号量，则该值为 0。

- cmd 参数表示在集合上执行的命令，这些命令及解释如表 7.2 所示。
- arg 参数的类型为 semun，这个特殊的联合体在 include/linux/sem.h 中声明，对它的描述如下：

```
/* arg for semctl system calls. */
union semun {
        int val;               /* value for SETVAL */
        struct semid_ds *buf;  /* buffer for IPC_STAT & IPC_SET */
        ushort *array;         /* array for GETALL & SETALL */
        struct seminfo *__buf; /* buffer for IPC_INFO */
        void *__pad;
};
```

这个联合体中，有 3 个成员已经在表 7.1 中提到，剩下的两个成员_buf 和_pad 用在内核中信号量的实现代码，开发者很少用到。事实上，这两个成员是 Linux 操作系统所特有的，在 UINX 中没有。

表 7.2　　cmd 命令及解释

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

这个系统调用比较复杂，我们举例说明。

下面这个程序段返回集合上索引为 semnum 对应信号量的值。当用 GETVAL 命令时，最后的参数（semnum）被忽略。

```
int get_sem_val ( int sid, int semnum )
{
        return ( semctl (sid, semnum, GETVAL, 0) ) ;
}
```

关于信号量的 3 个系统调用，我们进行了详细的介绍。从中可以看出，这几个系统调用的实现和使用都和系统内核密切相关，因此，如果在了解内核的基础上，再理解系统调用，相对要简单得多，也深入地多。

5．死锁

和信号量操作相关的概念还有“死锁”。当某个进程修改了信号量而进入临界区之后，却因为崩溃或被“杀死（kill）”而没有退出临界区，这时，其他被挂起在信号量上的进程永远得不到运行机会,这就是所谓的死锁。Linux 通过维护一个信号量数组的调整列表(semadj)来避免这一问题。其基本思想是，当应用这些“调整”时，让信号量的状态退回到操作实施前的状态。

关于调整的描述是在 sem_undo 数据结构中，在 include/linux/sem.h 描述如下：

```
/*每一个任务都有一系列的恢复（undo）请求，当进程退出时，自动执行 undo 请求*/

struct sem_undo {
    struct sem_undo *  proc_next;  /*在这个进程上的下一个 sem_undo 节点 */
    struct sem_undo *  id_next;    /* 在这个信号量集和上的下一个 sem_undo 节点*/
    int    semid;                 /* 信号量集的标识号*/
```

```
    short *         semadj;   /* 信号量数组的调整，每个进程一个*/
};
```

sem_undo 结构也出现在 task_struct 数据结构中。

每一个单独的信号量操作也许要请求得到一次“调整”，Linux 将为每一个信号量数组的每一个进程维护至少一个 sem_undo 结构。如果请求的进程没有这个结构，当必要时则创建它，新创建的 sem_undo 数据结构既在这个进程的 task_struct 数据结构中排队，也在信号量数组的 semid_ds 结构中排队。当对信号量数组上的一个信号量施加操作时，这个操作值的负数与这个信号量的“调整”相加，因此，如果操作值为 2，则把-2 加到这个信号量的“调整”域。

当进程被删除时，Linux 完成了对 sem_undo 数据结构的设置及对信号量数组的调整。如果一个信号量集合被删除，sem_undo 结构依然留在这个进程的 task_struct 结构中，但信号量集合的识别号变为无效。

7.3.2　消息队列

一个或多个进程可向消息队列写入消息，而一个或多个进程可从消息队列中读取消息，这种进程间通信机制通常使用在客户/服务器模型中，客户向服务器发送请求消息，服务器读取消息并执行相应请求。在许多微内核结构的操作系统中，内核和各组件之间的基本通信方式就是消息队列。例如，在 Minlx 操作系统中，内核、I/O 任务、服务器进程和用户进程之间就是通过消息队列实现通信的。

Linux 中的消息可以被描述成在内核地址空间的一个内部链表，每一个消息队列由一个 IPC 的标识号唯一地标识。Linux 为系统中所有的消息队列维护一个 msgque 链表，该链表中的每个指针指向一个 msgid_ds 结构，该结构完整描述一个消息队列。

1．数据结构

(1) 消息缓冲区（msgbuf）

我们在这里要介绍的第一个数据结构是 msgbuf 结构，可以把这个特殊的数据结构看成一个存放消息数据的模板，它在 include/linux/msg.h 中声明，描述如下：

```
/* msgsnd 和 msgrcv 系统调用使用的消息缓冲区*/
struct msgbuf {
    long mtype;         /* 消息的类型，必须为正数 */
    char mtext[1];      /* 消息正文 */
};
```

注意，对于消息数据元素（mtext），不要受其描述的限制。实际上，这个域（mtext）不仅能保存字符数组，而且能保存任何形式的任何数据。这个域本身是任意的，因为这个结构本身可以由应用程序员重新定义：

```
struct my_msgbuf {
        long    mtype;          /* 消息类型 */
        long    request_id;     /* 请求识别号 */
        struct  client info;    /* 客户消息结构 */
};
```

我们看到，消息的类型还是和前面一样，但是结构的剩余部分由两个其他的元素代替，

而且有一个是结构。这就是消息队列的优美之处，内核根本不管传送的是什么样的数据，任何信息都可以传送。

但是，消息的长度还是有限制的，在 Linux 中，给定消息的最大长度在 include/linux/msg.h 中定义如下：

```
#define MSGMAX  8192    /* max size of message (bytes) */
```

消息总的长度不能超过 8192 字节，包括 mtype 域，它是 4 字节长。

(2) 消息结构（msg）

内核把每一条消息存储在以 msg 结构为框架的队列中，它在 include/ linux/msg.h 中定义如下：

```
struct msg {
    struct msg *msg_next;   /* 队列上的下一条消息 */
    long  msg_type;          /*消息类型*/
    char *msg_spot;         /* 消息正文的地址 */
    short msg_ts;           /* 消息正文的大小 */
};
```

注意，msg_next 是指向下一条消息的指针，它们在内核地址空间形成一个单链表。

(3) 消息队列结构（msgid_ds）

当在系统中创建每一个消息队列时，内核创建、存储及维护这个结构的一个实例。

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

2．系统调用：msgget()

为了创建一个新的消息队列，或存取一个已经存在的队列，要使用 msgget()系统调用。

原型：int msgget （ key_t key，int msgflg ）；

返回：成功，则返回消息队列识别号，失败，则返回-1。

semget()中的第一个参数是键值，这个键值要与现有的键值进行比较，现有的键值指在内核中已存在的其他消息队列的键值。对消息队列的打开或存取操作依赖于 msgflg 参数的取值。

- IPC_CREAT ：如果这个队列在内核中不存在，则创建它。
- IPC_EXCL ：当与 IPC_CREAT 一起使用时，如果这个队列已存在，则创建失败。

如果 IPC_CREAT 单独使用，semget()为一个新创建的消息队列返回标识号，或者返回具有相同键值的已存在队列的标识号。如果 IPC_EXCL 与 IPC_CREAT 一起使用，要么创建一个新

的队列，要么对已存在的队列返回-1。IPC_EXCL 不能单独使用，当与 IPC_CREAT 结合起来使用时，可以保证新创建队列的打开和存取。

与文件系统的存取权限一样，每一个 IPC 对象也具有存取权限，因此，可以把一个 8 进制与掩码或，形成对消息队列的存取权限。

下面我们来创建一个打开或创建消息队列的函数：

```
int open_queue ( key_t keyval )
{
        int     qid;

        if ( (qid = msgget ( keyval, IPC_CREAT | 0660 ) )  == -1)
        {
                return (-1) ;
        }

        return (qid) ;
}
```

注意，这个例子显式地用了 0660 权限。这个函数要么返回一个消息队列的标识号，要么返回-1 而出错。键值作为唯一的参数必须传递给它。

3．系统调用：msgsnd()

一旦我们有了队列识别号，我们就可以在这个队列上执行操作。要把一条消息传递给一个队列，必须用 msgsnd()系统调用。

原型：int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgflg) ;

返回：成功为 0， 失败为-1。

msgsnd()的第 1 个参数是队列识别号，由 msgget()调用返回。第 2 个参数 msgp 是一个指针，指向我们重新声明和装载的消息缓冲区。msgsz 参数包含了消息以字节为单位的长度，其中包括了消息类型的 4 个字节。

msgflg 参数可以设置成 0（忽略），或者设置或 IPC_NOWAIT ：如果消息队列满，消息不写到队列中，并且控制权返回给调用进程（继续执行）；如果不指定 IPC_NOWAIT，调用进程将挂起（阻塞）直到消息被写到队列中。

下面我们来看一个发送消息的简单函数：

```
int send_message ( int qid, struct mymsgbuf *qbuf )
{
        int     result, length;
        /* mymsgbuf 结构的实际长度 */
        length = sizeof (struct )  - sizeof (long) ;

        if ( (result = msgsnd ( qid, qbuf, length, 0) )  == -1)
        {
                return (-1) ;
        }

        return (result) ;
}
```

这个小函数试图把缓冲区 qbuf 中的消息，发送给队列识别号为 qid 的消息队列。

现在，我们在消息队列里有了一条消息，可以用 ipcs 命令来看队列的状态。如何从消息队列检索消息，可以用 msgrcv()系统调用。

4．系统调用：msgrcv()

原型：int msgrcv （ int msqid, struct msgbuf *msgp, int msgsz, long mtype, int msgflg ）；

返回值：成功，则为拷贝到消息缓冲区的字节数，失败为-1。

很明显，第 1 个参数用来指定要检索的队列（必须由 msgget()调用返回），第 2 个参数（msgp）是存放检索到消息的缓冲区的地址，第 3 个参数（msgsz）是消息缓冲区的大小，包括消息类型的长度（4 字节）。

第 4 个参数（mtype）指定了消息的类型。内核将搜索队列中相匹配类型的最早的消息，并且返回这个消息的一个拷贝，返回的消息放在由 msgp 参数指向的地址。这里存在一个特殊的情况，如果传递给 mytype 参数的值为 0，就可以不管类型，只返回队列中最早的消息。

如果传递给参数 msgflg 的值为 IPC_NOWAIT，并且没有可取的消息，那么给调用进程返回 ENOMSG 错误消息，否则，调用进程阻塞，直到一条消息到达队列并且满足 msgrcv()的参数。如果一个客户正在等待消息，而队列被删除，则返回 EIDRM。如果当进程正在阻塞，并且等待一条消息到达但捕获到了一个信号，则返回 EINTR。

下面我们来看一个从我们已建的消息队列中检索消息的例子

```
int read_message ( int qid, long type, struct mymsgbuf *qbuf )
{
        int     result, length;

        /* 计算 mymsgbuf 结构的实际大小*/
        length = sizeof (struct mymsgbuf)  - sizeof (long) ;

        if ( (result = msgrcv ( qid, qbuf, length, type,  0) )  == -1)
        {
                return (-1) ;
        }

        return (result) ;
}
```

当从队列中成功地检索到消息后，这个消息将从队列中删除。

7.3.3 共享内存

共享内存可以被描述成内存一个区域（段）的映射，这个区域可以被更多的进程所共享。这是 IPC 机制中最快的一种形式，因为它不需要中间环节，而是把信息直接从一个内存段映射到调用进程的地址空间。一个段可以直接由一个进程创建，随后，可以有任意多的进程对其读和写。但是，一旦内存被共享之后，对共享内存的访问同步需要由其他 IPC 机制，例如信号量来实现。像所有的 System V IPC 对象一样，Linux 对共享内存的存取是通过对访问键和访问权限的检查来控制的。

1．数据结构

与消息队列和信号量集合类似，内核为每一个共享内存段（存在于它的地址空间）维护着一个特殊的数据结构 shmid_ds，这个结构在 include/linux/shm.h 中定义如下：

```
/* 在系统中 每一个共享内存段都有一个 shmid_ds 数据结构. */
struct shmid_ds {
    struct ipc_perm shm_perm;          /* 操作权限 */
    int     shm_segsz;                      /* 段的大小（以字节为单位） */
    time_t  shm_atime;                      /* 最后一个进程附加到该段的时间 */
    time_t  shm_dtime;                 /* 最后一个进程离开该段的时间 */
    time_t  shm_ctime;                 /* 最后一次修改这个结构的时间 */
    unsigned short  shm_cpid;          /*创建该段进程的 pid */
    unsigned short  shm_lpid;          /* 在该段上操作的最后一个进程的 pid */
    short   shm_nattch;                /*当前附加到该段的进程的个数  */

                                            /* 下面是私有的 */

    unsigned short   shm_npages;       /*段的大小（以页为单位） */
   unsigned long   *shm_pages;         /* 指向 frames -> SHMMAX 的指针数组 */
            struct vm_area_struct *attaches;    /* 对共享段的描述 */
     };
```

我们用图 7.4 来表示共享内存的数据结构 shmid_ds 与其他相关数据结构的关系。

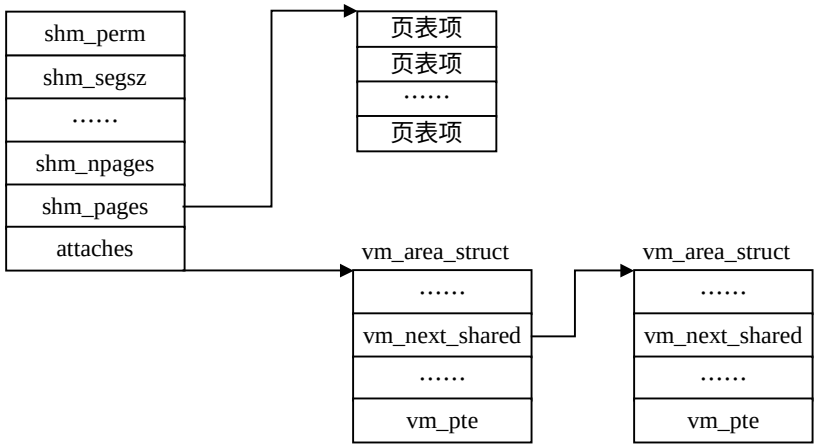

图 7.4　共享内存的数据结构

2．共享内存的处理过程

某个进程第 1 次访问共享虚拟内存时将产生缺页异常。这时，Linux 找出描述该内存的 vm_area_struct 结构，该结构中包含用来处理这种共享虚拟内存段的处理函数地址。共享内存缺页异常处理代码对 shmid_ds 的页表项表进行搜索，以便查看是否存在该共享虚拟内存的页表项。如果没有，系统将分配一个物理页并建立页表项，该页表项加入 shmid_ds 结构的同时也添加到进程的页表中。这就意味着当下一个进程试图访问这页内存时出现缺页异常，共享内存的缺页异常处理代码则把新创建的物理页给这个进程。因此说，第 1 个进程对共享

内存的存取引起创建新的物理页面，而其他进程对共享内存的存取引起把那个页添加到它们的地址空间。

当某个进程不再共享其虚拟内存时，利用系统调用将共享段从自己的虚拟地址区域中移去，并更新进程页表。当最后一个进程释放了共享段之后，系统将释放给共享段所分配的物理页。

当共享的虚拟内存没有被锁定到物理内存时，共享内存也可能会被交换到交换区中。

3．系统调用：shmget()

原型：int shmget （ key_t key, int size, int shmflg ）；
返回：成功，则返回共享内存段的识别号，失败返回-1。
shmget()系统调用类似于信号量和消息队列的系统调用，在此不进一步赘述。

4．系统调用：shmat()

原型：int shmat（int shmid, char *shmaddr, int shmflg）；
返回：成功，则返回附加到进程的那个段的地址，失败返回-1。
其中 shmid 是由 shmget()调用返回的共享内存段识别号，shmaddr 是你希望共享段附加的地址，shmflag 允许你规定希望所附加的段为只读（利用 SHM_RDONLY）以代替读写。通常，并不需要规定你自己的 shmaddr，可以用传递参数值零使得系统为你取得一个地址。

这个调用可能是最简单的，下面看一个例子，把一个有效的识别号传递给一个段，然后返回这个段被附加到内存的内存地址。

```
char *attach_segment ( int shmid )
{
        return (shmat (shmid, 0, 0) ) ;
}
```

一旦一个段适当地被附加，并且一个进程有指向那个段起始地址的一个指针，那么，对那个段的读写就变得相当容易。

5．系统调用：shmctl()

原型： int shmctl （ int shmqid, int cmd, struct shmid_ds *buf ）;
返回：成功返回 0 ，失败返回-1。
这个特殊的调用和 semctl()调用几乎相同，因此，这里不进行详细的讨论。有效命令的值如下所述。

- IPC_STAT：检索一个共享段的 shmid_ds 结构，把它存到 buf 参数的地址中。
- IPC_SET：对一个共享段来说，从 buf 参数中取值设置 shmid_ds 结构的 ipc_perm 域的值。
- IPC_RMID：把一个段标记为删除。
- IPC_RMID 命令实际上不从内核删除一个段，而是仅仅把这个段标记为删除，实际的删除发生在最后一个进程离开这个共享段时。

当一个进程不再需要共享内存段时，它将调用 shmdt()系统调用取消这个段，但是，这并不是从内核真正地删除这个段，而是把相关 shmid_ds 结构的 shm_nattch 域的值减 1，当

这个值为 0 时，内核才从物理上删除这个共享段。