Digital Unix显然不存在semmnu限制。

294 ~ 296

例子

图11-9中的程序确定图11-8中给出的各个限制值。

svsem/limits.c

```
 1 #include    "unpipc.h"

 2          /* following are upper limits of values to try */
 3 #define  MAX_NIDS        4096            /* max # semaphore IDs */
 4 #define  MAX_VALUE       1024*1024       /* max semaphore value */
 5 #define  MAX_MEMBERS     4096            /* max # semaphores per semaphore set */
 6 #define  MAX_NOPS        4096            /* max # operations per semop() */
 7 #define  MAX_NPROC       Sysconf(_SC_CHILD_MAX)

 8 int
 9 main(int argc, char **argv)
10 {
11     int     i, j, semid, sid[MAX_NIDS], pipefd[2];
12     int     semmni, semvmx, semmsl, semmns, semopn, semaem, semume, semmnu;
13     pid_t   *child;
14     union semun arg;
15     struct sembuf   ops[MAX_NOPS];

16         /* see how many sets with one member we can create */
17     for (i = 0; i <= MAX_NIDS; i++) {
18         sid[i] = semget(IPC_PRIVATE, 1, SVSEM_MODE | IPC_CREAT);
19         if (sid[i] == -1) {
20             semmni = i;
21             printf("%d identifiers open at once\n", semmni);
22             break;
23         }
24     }
25         /* before deleting, find maximum value using sid[0] */
26     for (j = 7; j < MAX_VALUE; j += 8) {
27         arg.val = j;
28         if (semctl(sid[0], 0, SETVAL, arg) == -1) {
29             semvmx = j - 8;
30             printf("max semaphore value = %d\n", semvmx);
31             break;
32         }
33     }
34     for (j = 0; j < i; j++)
35         Semctl(sid[j], 0, IPC_RMID);

36         /* determine max # semaphores per semaphore set */
37     for (i = 1; i <= MAX_MEMBERS; i++) {
38         semid = semget(IPC_PRIVATE, i, SVSEM_MODE | IPC_CREAT);
39         if (semid == -1) {
40             semmsl = i - 1;
41             printf("max of %d members per set\n", semmsl);
42             break;
43         }
44         Semctl(semid, 0, IPC_RMID);
45     }

46         /* find max of total # of semaphores we can create */
47     semmns = 0;
```

图11-9 确定System V信号量上的系统限制值

```
48      for (i = 0; i < semmni; i++) {
49          sid[i] = semget(IPC_PRIVATE, semmsl, SVSEM_MODE | IPC_CREAT);
297 50      if (sid[i] == -1) {
51              /*
52               * Up to this point each set has been created with semmsl
53               * members.  But this just failed, so try recreating this
54               * final set with one fewer member per set, until it works.
55               */
56              for (j = semmsl - 1; j > 0; j--) {
57                  sid[i] = semget(IPC_PRIVATE, j, SVSEM_MODE | IPC_CREAT);
58                  if (sid[i] != -1) {
59                      semmns += j;
60                      printf("max of %d semaphores\n", semmns);
61                      Semctl(sid[i], 0, IPC_RMID);
62                      goto done;
63                  }
64              }
65              err_quit("j reached 0, semmns = %d", semmns);
66          }
67          semmns += semmsl;
68      }
69      printf("max of %d semaphores\n", semmns);
70  done:
71      for (j = 0; j < i; j++)
72          Semctl(sid[j], 0, IPC_RMID);
73          /* see how many operations per semop() */
74      semid = Semget(IPC_PRIVATE, semmsl, SVSEM_MODE | IPC_CREAT);
75      for (i = 1; i <= MAX_NOPS; i++) {
76          ops[i - 1].sem_num = i - 1;
77          ops[i - 1].sem_op = 1;
78          ops[i - 1].sem_flg = 0;
79          if (semop(semid, ops, i)  == -1) {
80              if (errno != E2BIG)
81                  err_sys("expected E2BIG from semop");
82              semopn = i-1;
83              printf("max of %d operations per semop()\n", semopn);
84              break;
85          }
86      }
87      Semctl(semid, 0, IPC_RMID);
88          /* determine the max value of semadj */
89          /* create one set with one semaphore */
90      semid = Semget(IPC_PRIVATE, 1, SVSEM_MODE | IPC_CREAT);
91      arg.val = semvmx;
92      Semctl(semid, 0, SETVAL, arg);      /* set value to max */
93      for (i = semvmx - 1; i > 0; i--) {
94          ops[0].sem_num = 0;
95          ops[0].sem_op = -i;
96          ops[0].sem_flg = SEM_UNDO;
97          if (semop(semid, ops, 1)  != -1) {
98              semaem = i;
99              printf("max value of adjust-on-exit = %d\n", semaem);
100             break;
101         }
```

图11-9（续）

```
102     }
103     Semctl(semid, 0, IPC_RMID);

104         /* determine max # undo structures */
105         /* create one set with one semaphore; init to 0 */
106     semid = Semget(IPC_PRIVATE, 1, SVSEM_MODE | IPC_CREAT);
107     arg.val = 0;
108     Semctl(semid, 0, SETVAL, arg);      /* set semaphore value to 0 */
109     Pipe(pipefd);
110     child = Malloc(MAX_NPROC * sizeof(pid_t));
111     for (i = 0; i < MAX_NPROC; i++) {
112         if ( (child[i] = fork()) == -1) {
113             semmnu = i - 1;
114             printf("fork failed, semmnu at least %d\n", semmnu);
115             break;
116         } else if (child[i] == 0) {
117             ops[0].sem_num = 0;     /* child does the semop() */
118             ops[0].sem_op = 1;
119             ops[0].sem_flg = SEM_UNDO;
120             j = semop(semid, ops, 1);  /* 0 if OK, -1 if error */
121             Write(pipefd[1], &j, sizeof(j));
122             sleep(30);              /* wait to be killed by parent */
123             exit(0);                /* just in case */
124         }
125         /* parent reads result of semop() */
126         Read(pipefd[0], &j, sizeof(j));
127         if (j == -1) {
128             semmnu = i;
129             printf("max # undo structures = %d\n", semmnu);
130             break;
131         }
132     }
133     Semctl(semid, 0, IPC_RMID);
134     for (j = 0; j <= i && child[j] > 0; j++)
135         Kill(child[j], SIGINT);

136         /* determine max # adjust entries per process */
137         /* create one set with max # of semaphores */
138     semid = Semget(IPC_PRIVATE, semmsl, SVSEM_MODE | IPC_CREAT);
139     for (i = 0; i < semmsl; i++) {
140         arg.val = 0;
141         Semctl(semid, i, SETVAL, arg);      /* set semaphore value to 0 */

142         ops[i].sem_num = i;
143         ops[i].sem_op = 1;          /* add 1 to the value */
144         ops[i].sem_flg = SEM_UNDO;
145         if (semop(semid, ops, i + 1) == -1) {
146             semume = i;
147             printf("max # undo entries per process = %d\n", semume);
148             break;
149         }
150     }
151     Semctl(semid, 0, IPC_RMID);

152     exit(0);
153 }
```

svsem/limits.c

图11-9（续）

11.8　小结

从Posix信号量到System V信号量发生了如下变动。

(1) System V信号量由一组值构成。当指定应用到某个信号量集的一组信号量操作时，要么所有操作都执行，要么一个操作都不执行。

(2) 可应用到一个信号量集的每个成员的操作有三种：测试其值是否为0、往其值加一个整数以及从其值中减掉一个整数（假设结果值仍然非负）。Posix信号量所允许的操作只是将其值加1或减1（假设结果值仍然非负）。

(3) 创建一个System V信号量集需要技巧，因为创建该集合并随后初始化其各个值需要两个操作，从而可能导致竞争状态。

(4) System V信号量提供“复旧”特性，该特性保证在进程终止时逆转某个信号量操作。

习题

11.1　图6-8是对图6-6的修改，它接受用于指定消息队列的标识符而不是路径名。从中看出访问一个System V消息队列只需知道其标识符（前提是有足够的权限）。对图11-6进行类似的修改，以展示同样的特性也适用于System V信号量。

300

11.2　如果LOCK_PATH文件不存在，那么图11-7中的函数会发生什么？

第四部分

共享内存区

第12章 共享内存区介绍

12.1 概述

共享内存区是可用IPC形式中最快的。一旦这样的内存区映射到共享它的进程的地址空间，这些进程间数据的传递就不再涉及内核。然而往该共享内存区存放信息或从中取走信息的进程间通常需要某种形式的同步。我们在第3部分中已经讨论了各种形式的同步：互斥锁、条件变量、读写锁、记录锁、信号量。

> 这里说的“不再涉及内核”的含义是：进程不再通过执行任何进入内核的系统调用来彼此传递数据。显然，内核必须建立允许各个进程共享该内存区的内存映射关系，然后一直管理该内存区（处理页面故障等）。

考虑用来传递各种类型消息的一个示例客户-服务器文件复制程序中涉及的通常步骤。

- 服务器从输入文件读。该文件的数据由内核读入自己的内存空间，然后从内核复制到服务器进程。
- 服务器往一个管道、FIFO或消息队列以一条消息的形式写入这些数据。这些IPC形式通常需要把这些数据从进程复制到内核。

> 这里使用限定词“通常”是因为Posix消息队列可使用内存映射I/O（本章将描述的mmap函数）实现，如5.8节和习题12.2的解答中所示。在图12-1中，我们假设Posix消息队列是在内核中实现的，这是另外一种可能实现。然而管道、FIFO和System V消息队列的write或msgsnd都涉及从进程到内核的数据复制，它们的read或msgrcv都涉及从内核到进程的数据复制。

- 客户从该IPC通道读出这些数据，这通常需要把这些数据从内核复制到进程。
- 最后，将这些数据从由write函数的第二个参数指定的客户缓冲区复制到输出文件。

这里通常需要总共四次数据复制。而且这四次复制是在内核和某个进程间进行的，往往开销很大（比纯粹在内核中或单个进程内复制数据的开销大）。图12-1展示了客户与服务器之间通过内核桥接的数据转移。

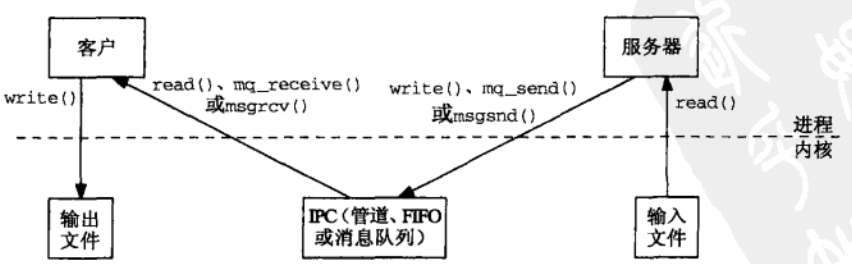


图12-1 从服务器到客户的文件数据流

这些IPC形式（管道、FIFO和消息队列）的问题在于，两个进程要交换信息时，这些信息必须经由内核传递。

通过让两个或多个进程共享一个内存区，共享内存区这种IPC形式提供了绕过上述问题的办法。当然，这些进程必须协调或同步对该共享内存区的使用。（共享一个公共的内存区跟共享一个硬盘文件类似，例如本书所有文件上锁例子中都使用的那个序列号文件。）第三部分讲述的任何技巧都可用于这样的同步目的。

前面的客户-服务器例子现在涉及的步骤如下。

- 服务器使用（譬如说）一个信号量取得访问某个共享内存区对象的权力。
- 服务器将数据从输入文件读入到该共享内存区对象。read函数的第二个参数所指定的数据缓冲区地址指向这个共享内存区对象。
- 服务器读入完毕时，使用一个信号量通知客户。
- 客户将这些数据从该共享内存区对象写出到输出文件中。

304

图12-2展示了这个情形。

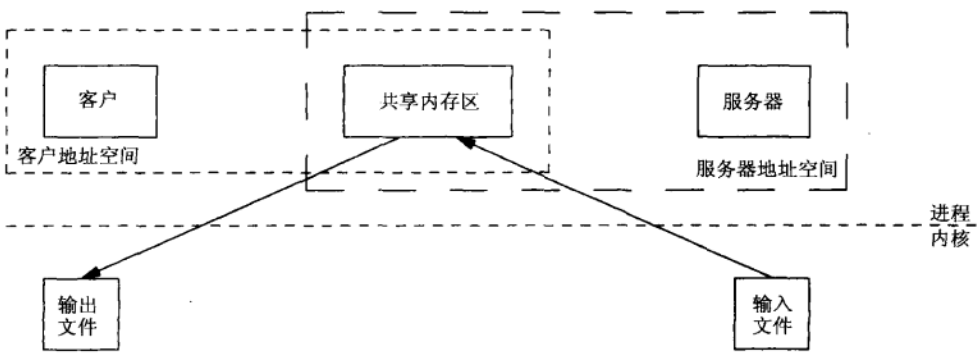


图12-2 使用共享内存区将文件数据从服务器复制到客户

本图中数据只复制两次：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。我们画了一个包围客户和该共享内存区对象的虚框，又画了另一个包围服务器和该共享内存区对象的虚框，目的是强调该共享内存区对象同时出现在客户和服务器的地址空间中。

使用共享内存区所涉及的概念对于Posix接口和System V接口都类似。我们将在第13章中讲述前者，在第14章中讲述后者。

本章中我们返回到第9章中开始介绍的序列号加1的例子。不过我们现在把序列号存放在内存中而不是某个文件里。

我们首先再次强调，默认情况下通过fork派生的子进程并不与其父进程共享内存区。图12-3中的程序让父子进程都给一个名为count的全局整数加1。

创建并初始化信号量

12~14 创建并初始化一个信号量，它保护我们认为其为一个共享变量的对象（全局变量count）。由于这样的假设不正确，该信号量实际上并非必要。注意，我们通过调用sem_unlink从系统中删除了该信号量的名字，但是尽管这么一来删除了它的路径名，对于已经打开的信号量却没有影响。这样做后即使本程序中止了，该路径名也已从系统中删除。

shm/incr1.c

```
 1 #include     "unpipc.h"
 2 #define SEM_NAME     "mysem"
 3 int      count = 0;
 4 int
 5 main(int argc, char **argv)
 6 {
 7     int      i, nloop;
 8     sem_t    *mutex;
 9     if (argc != 2)
10         err_quit("usage: incr1 <#loops>");
11     nloop = atoi(argv[1]);
12         /* create, initialize, and unlink semaphore */
13     mutex = Sem_open(Px_ipc_name(SEM_NAME), O_CREAT | O_EXCL, FILE_MODE, 1);
14     Sem_unlink(Px_ipc_name(SEM_NAME));
15     setbuf(stdout, NULL);        /* stdout is unbuffered */
16     if (Fork() == 0) {           /* child */
17         for (i = 0; i < nloop; i++) {
18             Sem_wait(mutex);
19             printf("child: %d\n", count++);
20             Sem_post(mutex);
21         }
22         exit(0);
23     }
24         /* parent */
25     for (i = 0; i < nloop; i++) {
26         Sem_wait(mutex);
27         printf("parent: %d\n", count++);
28         Sem_post(mutex);
29     }
30     exit(0);
31 }
```

shm/incr1.c

图12-3　父子进程都给同一个全局变量加1

把标准输出设置为非缓冲，然后fork

15　把标准输出设置为非缓冲模式，因为父子进程都要往它写出结果。这样可以防止这两个进程的输出不恰当地交插。①

16~29　父子进程都执行一个循环，该循环对计数器执行指定次数的加1，并小心地保证只在持有保护它的信号量时才给该变量加1。

运行该程序，只查看系统在父子进程间切换时的输出，我们得到如下结果：

```
child: 0                    子进程首先运行，计数器从0开始计数
child: 1
. . .
child: 678
child: 679
parent: 0                   子进程被阻止，父进程运行，计数器从0开始计数
parent: 1
. . .
```

① 确切地说，从观察程序运行的用户看来，缓冲模式的标准输出妨碍了父子进程动态输出的及时反映。——译者注

305
2
306

```
parent: 1220
parent: 1221
child: 680                    父进程被阻止，子进程接着运行
child: 681
. . .
child: 2078
child: 2079
parent: 1222                  子进程被阻止，父进程接着运行
parent: 1223
                              如此等等
```

可以看出这两个进程都有各自的全局变量count的副本。每个进程都从该变量为0的初始值开始，而且每次加1的对象是各自的变量的副本。图12-4展示了调用fork之前的父进程。

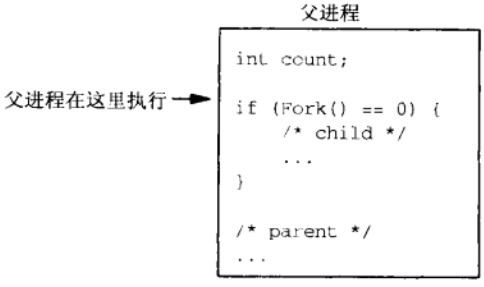


图12-4 调用fork之前的父过程

调用fork后，子进程从其父进程数据空间的映射副本开始运行。图12-5展示了fork返回后的两个进程。

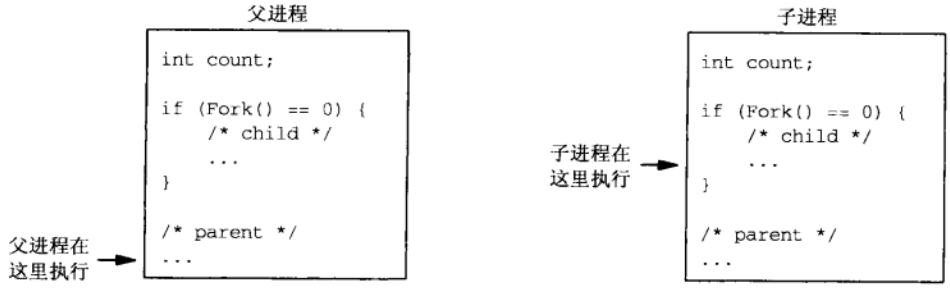


图12-5 fork返回之后的父子进程

我们看到父子进程都有各自的变量count的副本。

12.2 mmap、munmap 和 msync 函数

307

mmap函数把一个文件或一个Posix共享内存区对象映射到调用进程的地址空间。使用该函数有三个目的：

(1) 使用普通文件以提供内存映射I/O（12.3节）；

(2) 使用特殊文件以提供匿名内存映射（12.4节和12.5节）；

(3) 使用shm_open以提供无亲缘关系进程间的Posix共享内存区（第13章）。

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
                                返回：若成功则为被映射区的起始地址，若出错则为MAP_FAILED
```

其中*addr*可以指定描述符*fd*应被映射到的进程内空间的起始地址。它通常被指定为一个空指针，这样告诉内核自己去选择起始地址。无论哪种情况下，该函数的返回值都是描述符*fd*所映射到内存区的起始地址。

*len*是映射到调用进程地址空间中的字节数，它从被映射文件开头起第*offset*个字节处开始算。*offset*通常设置为0。图12-6展示了这个映射关系。

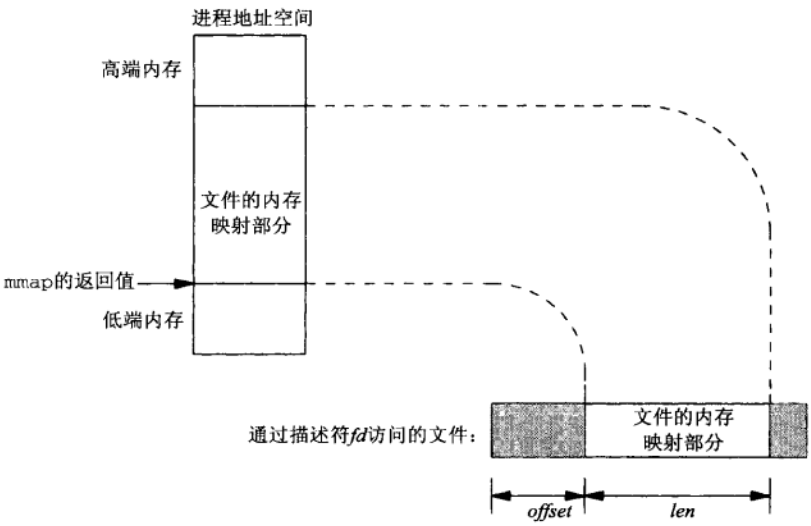


图12-6　内存映射文件的例子

内存映射区的保护由*prot*参数指定，它使用图12-7中的常值。该参数的常见值是代表读写访问的PROT_READ | PROT_WRITE。

308

prot	说　明
PROT_READ	数据可读
PROT_WRITE	数据可写
PROT_EXEC	数据可执行
PROT_NONE	数据不可访问

图12-7　mmap的*prot*参数

*flags*使用图12-8中的常值指定。MAP_SHARED或MAP_PRIVATE这两个标志必须指定一个，并可有选择地或上MAP_FIXED。如果指定了MAP_PRIVATE，那么调用进程对被映射数据所作的修改只对该进程可见，而不改变其底层支撑对象（或者是一个文件对象，或者是一个共享内存区对象）。如果指定了MAP_SHARED，那么调用进程对被映射数据所作的修改对于共享该对象的所有进程都可见，而且确实改变了其底层支撑对象。

Flags	说 明
MAP_SHARED	变动是共享的
MAP_PRIVATE	变动是私自的
MAP_FIXED	准确地解释*addr*参数

图12-8 mmap的*flags*参数

从移植性上考虑，MAP_FIXED不应该指定。如果没有指定该标志，但是*addr*不是一个空指针，那么*addr*如何处置取决于实现。不为空的*addr*值通常被当作有关该内存区应如何具体定位的线索。可移植的代码应把*addr*指定成一个空指针，并且不指定MAP_FIXED。

父子进程之间共享内存区的方法之一是，父进程在调用fork前先指定MAP_SHARED调用mmap。Posix.1保证父进程中的内存映射关系存留到子进程中。而且父进程所作的修改子进程能看到，反过来也一样。我们稍后将给出这样的一个例子。

mmap成功返回后，*fd*参数可以关闭。该操作对于由mmap建立的映射关系没有影响。

为从某个进程的地址空间删除一个映射关系，我们调用munmap。

```
#include <sys/mman.h>
int munmap(void *addr, size_t len);
                                    返回：若成功则为0，若出错则为-1
```

其中*addr*参数是由mmap返回的地址，*len*是映射区的大小。再次访问这些地址将导致向调用进程产生一个SIGSEGV信号（当然这里假设以后的mmap调用并不重用这部分地址空间）。 309

如果被映射区是使用MAP_PRIVATE标志映射的，那么调用进程对它所作的变动都会被丢弃掉。

在图12-6中，内核的虚拟内存算法保持内存映射文件（一般在硬盘上）与内存映射区（在内存中）的同步，前提是它是一个MAP_SHARED内存区。这就是说，如果我们修改了处于内存映射到某个文件的内存区中某个位置的内容，那么内核将在稍后某个时刻相应地更新文件。然而有时候我们希望确信硬盘上的文件内容与内存映射区中的内容一致，于是调用msync来执行这种同步。

```
#include <sys/mman.h>
int msync(void *addr, size_t len, int flags);
                                    返回：若成功则为0，若出错则为-1
```

其中*addr*和*len*参数通常指代内存中的整个内存映射区，不过也可以指定该内存区的一个子集。*flags*参数是图12-9中所示各常值的组合。

常 值	说 明
MS_ASYNC	执行异步写
MS_SYNC	执行同步写
MS_INVALIDATE	使高速缓存的数据失效

图12-9 msync函数的*flags*参数

MS_ASYNC和MS_SYNC这两个常值中必须指定一个，但不能都指定。它们的差别是，一旦写操作已由内核排入队列，MS_ASYNC即返回，而MS_SYNC则要等到写操作完成后才返回。如果还

指定了MS_INVALIDATE，那么与其最终副本不一致的文件数据的所有内存中副本都失效。后续的引用将从文件中取得数据。

为何使用 mmap

到此为止就mmap的描述间接说明了内存映射文件：我们open它之后调用mmap把它映射到调用进程地址空间的某个文件。使用内存映射文件所得到的奇妙特性是，所有的I/O都在内核的掩盖下完成，我们只需编写存取内存映射区中各个值的代码。[①]我们决不调用read、write或lseek。这么一来往往可以简化我们的代码。

回想使用mmap完成的Posix消息队列的实现，其中图5-30有往内存映射区中某个msg_hdr
310 结构存入值的代码，图5-32有从内存映射区中某个msg_hdr结构取出值的代码。

然而需要了解以防误解的说明是，不是所有文件都能进行内存映射。例如，试图把一个访问终端或套接字的描述符映射到内存将导致mmap返回一个错误。这些类型的描述符必须使用read和write（或者它们的变体）来访问。

mmap的另一个用途是在无亲缘关系的进程间提供共享的内存区。这种情形下，所映射文件的实际内容成了被共享内存区的初始内容，而且这些进程对该共享内存区所作的任何变动都复制回所映射的文件（以提供随文件系统的持续性）。这里假设指定了MAP_SHARED标志，它是进程间共享内存所需求的。

有关mmap的实现以及它与内核虚拟内存算法之间的关系具体参见［McKusick et al.1996］（适用于4.4BSD）以及［Vahalia 1996］和［Goodheart and Cox 1994］（适用于SVR4）。

12.3　在内存映射文件中给计数器持续加 1

现在对（不工作的）图12-3进行修改，以使父子进程共享存放着计数器的一个内存区。为此目的，我们使用一个内存映射文件：我们open它之后调用mmap把它映射到调用进程地址空间的某个文件。图12-10给出了这个新程序。

新的命令行参数

11~14　新增的命令行参数是有待内存映射的一个文件的名字。我们打开该文件用于读和写，若不存在则创建之，然后写一个值为0的整数到该文件中。

mmap后关闭描述符

15~16　调用mmap把刚才打开的文件映射到本进程的内存空间。第一个参数是一个空指针，因而由系统来选择起始地址。长度参数是一个int的大小，保护模式参数指定读写访问。通过把第四个参数指定为MAP_SHARED，父进程所作的任何变动子进程都能看到，反过来也一样。函数返回值是待共享内存区的起始地址，我们把它保存在ptr中。

fork

20~34　把标准输出设置成非缓冲模式后调用fork。父子进程都要对由ptr指向的整数计数器执行加1操作。

① 意思是不像调用read和write执行I/O时那样由内核直接参与I/O的完成，而是由内核在背后通过操纵页表等方法间接参与，这样就用户进程看来，I/O不再涉及系统调用。——译者注

shm/incr2.c

```
1 #include    "unpipc.h"
2 #define  SEM_NAME    "mysem"
3 int
4 main(int argc, char **argv)
5 {
6     int     fd, i, nloop, zero = 0;
7     int     *ptr;
8     sem_t   *mutex;
9     if (argc != 3)
10        err_quit("usage: incr2 <pathname> <#loops>");
11    nloop = atoi(argv[2]);
12        /* open file, initialize to 0, map into memory */
13    fd = Open(argv[1], O_RDWR | O_CREAT, FILE_MODE);
14    Write(fd, &zero, sizeof(int));
15    ptr = Mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
16    Close(fd);
17        /* create, initialize, and unlink semaphore */
18    mutex = Sem_open(Px_ipc_name(SEM_NAME), O_CREAT | O_EXCL, FILE_MODE, 1);
19    Sem_unlink(Px_ipc_name(SEM_NAME));
20    setbuf(stdout, NULL);       /* stdout is unbuffered */
21    if (Fork() == 0) {          /* child */
22        for (i = 0; i < nloop; i++) {
23            Sem_wait(mutex);
24            printf("child: %d\n", (*ptr)++);
25            Sem_post(mutex);
26        }
27        exit(0);
28    }
29        /* parent */
30    for (i = 0; i < nloop; i++) {
31        Sem_wait(mutex);
32        printf("parent: %d\n", (*ptr)++);
33        Sem_post(mutex);
34    }
35    exit(0);
36 }
```

shm/incr2.c

图12-10 父子进程给共享内存区中的一个计数器加1

fork对内存映射文件进行特殊处理，也就是父进程在调用fork之前创建的内存映射关系由子进程共享。因此，我们刚才在打开文件后以MAP_SHARED标志调用mmap的操作实际上提供了一个由父子进程共享的内存区。而且，既然该共享内存区是一个内存映射文件，因而对它（由ptr指向的大小为sizeof(int)的内存区）所作的任何变动还会反映到真正的文件中（该文件的名字由命令行参数指定）。

执行这个程序，我们发现由ptr指向的内存区确实在父子进程间共享。下面只给出内核在这两个进程间来回切换上下文时输出的值。

```
solaris % incr2 /tmp/temp.1 10000
child: 0                    子进程首先运行
child: 1
. . .
child: 128
```

311 ~ 312

```
child: 129
parent: 130                          子进程被阻止，父进程启动
parent: 131
. . .
parent: 636
parent: 637
child: 638                           父进程被阻止，子进程接着运行
child: 639
. . .
child: 1517
child: 1518
parent: 1519                         子进程被阻止，父进程接着运行
parent: 1520
. . .
parent: 19999                        最后一行输出
solaris % od -D /tmp/temp.1
0000000 0000020000
0000004
```

既然文件/tmp/temp.1已被内存映射，incr2程序运行终止后我们可以用od程序查看该文件的内容，发现其中确实存放着计数器的最终值（20000）。

图12-11是对图12-5的修改，它画出了共享内存区，并表示出信号量也是共享的。这里的信号量画成是在内核中，然而正如我们讲述Posix信号量时提到的那样，这并不是必须的。不论使用什么来实现，信号量必须至少具有随内核的持续性。如10.15节所展示的那样，该信号量也可作为另一个内存映射文件存放。

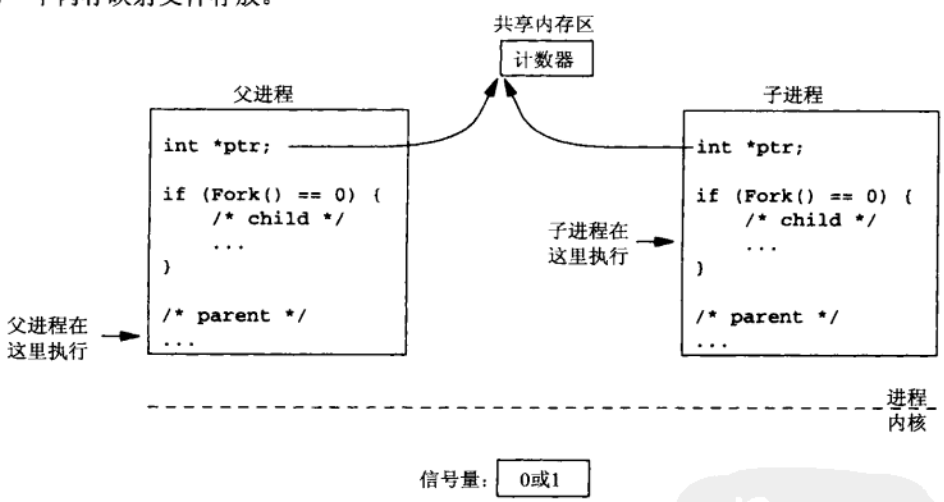


图12-11　共享一个内存区和一个信号量的父子进程

图中画出父子进程都各自有属于自己的指针ptr的副本，但是每个副本都指向共享内存区中的同一个整数：这两个进程都对它执行加1操作的计数器。

313

现在把图12-10中的程序改为使用一个Posix基于内存的信号量，而不是一个Posix有名信号量，并把该信号量存放在共享内存区中。图12-12给出了这个新程序。

定义将存放在共享内存区中的结构

2~5　定义一个结构，其中含有整数计数器以及保护它的信号量。该结构将存放到共享内存区对象中。

shm/incr3.c

```
1 #include    "unpipc.h"

2 struct shared {
3   sem_t mutex;            /* the mutex: a Posix memory-based semaphore */
4   int   count;            /* and the counter */
5 } shared;

6 int
7 main(int argc, char **argv)
8 {
9     int     fd, i, nloop;
10    struct shared   *ptr;

11    if (argc != 3)
12        err_quit("usage: incr3 <pathname> <#loops>");
13    nloop = atoi(argv[2]);

14        /* open file, initialize to 0, map into memory */
15    fd = Open(argv[1], O_RDWR | O_CREAT, FILE_MODE);
16    Write(fd, &shared, sizeof(struct shared));
17    ptr = Mmap(NULL, sizeof(struct shared), PROT_READ | PROT_WRITE,
18               MAP_SHARED, fd, 0);
19    Close(fd);

20        /* initialize semaphore that is shared between processes */
21    Sem_init(&ptr->mutex, 1, 1);

22    setbuf(stdout, NULL);       /* stdout is unbuffered */
23    if (Fork() == 0) {          /* child */
24        for (i = 0; i < nloop; i++) {
25            Sem_wait(&ptr->mutex);
26            printf("child: %d\n", ptr->count++);
27            Sem_post(&ptr->mutex);
28        }
29        exit(0);
30    }
31        /* parent */
32    for (i = 0; i < nloop; i++) {
33        Sem_wait(&ptr->mutex);
34        printf("parent: %d\n", ptr->count++);
35        Sem_post(&ptr->mutex);
36    }
37    exit(0);
38 }
```

shm/incr3.c

314

图12-12 计数器和信号量都在共享内存区中

映射到内存

14~19 创建一个将被映射到内存的文件，将一个值为0的上述结构写到该文件中。我们所做的只是初始化其中的计数器，因为信号量的值是通过调用sem_init初始化的。然而把整个结构写成0要比试图只写一个值为0的整数容易。

初始化信号量

20~21 现在是用一个基于内存的信号量代替一个有名信号量，因此我们调用sem_init把它的值初始化为1。第二个参数必须不为0，以指示该信号量将在进程间共享。

图12-13是对图12-11的修改，注意其中的信号量已从内核挪到了共享内存区中。

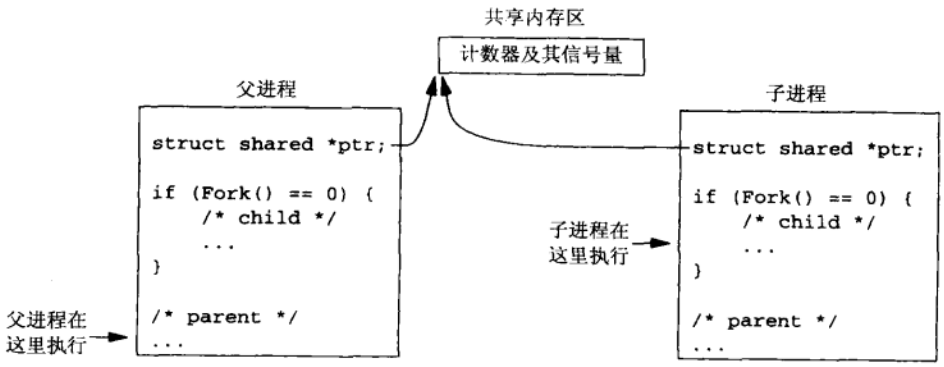


图12-13　现在计数器和信号量都在共享内存区中

12.4　4.4BSD匿名内存映射

图12-10和图12-12中的例子程序工作正确，然而我们不得不在文件系统中创建一个文件（其名字由命令行参数给出），调用open，然后往该文件中write一些0以初始化它。如果调用mmap的目的是提供一个将穿越fork由父子进程共享的映射内存区，那么我们可以简化上述情形，具体方法取决于实现。

(1) 4.4BSD提供匿名内存映射（anonymous memory mapping），它彻底避免了文件的创建和打开。其办法是把mmap的*flags*参数指定成MAP_SHARED | MAP_ANON，把*fd*参数指定为−1。*offset*参数则被忽略。这样的内存区初始化为0。我们将在图12-14中给出这种内存映射的一个例子。

(2) SVR4提供/dev/zero设备文件，我们open它之后可在mmap调用中使用得到的描述符。从该设备读时返回的字节全为0，写往该设备的任何字节则被丢弃。我们将在图12-15中给出使用该设备进行内存映射的一个例子。（许多源自Berkeley的实现也支持/dev/zero，例如SunOS 4.1.x和BSD/OS 3.1。）

315

图12-14给出了改用4.4BSD匿名内存映射后，与图12-10中程序相比唯一有变动的部分。

shm/incr_map_anon.c

```
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int      i, nloop;
 7     int     *ptr;
 8     sem_t   *mutex;

 9     if (argc != 2)
10         err_quit("usage: incr_map_anon <#loops>");
11     nloop = atoi(argv[1]);

12         /* map into memory */
13     ptr = Mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
14                MAP_SHARED | MAP_ANON, -1, 0);
```

shm/incr_map_anon.c

图12-14　4.4BSD匿名内存映射

6~11　自动变量fd和zero，以及指定待创建路径名的命令行参数都被去掉了。

12~14　我们不再open一个文件。在调用mmap时指定了MAP_ANON标志，并置第五个参数（描述符）为−1。

12.5 SVR4 /dev/zero 内存映射

图12-15给出了改为映射/dev/zero后与图12-10中程序相比唯一有变动部分。

shm/incr_dev_zero.c

```
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int      fd, i, nloop;
 7     int     *ptr;
 8     sem_t   *mutex;

 9     if (argc != 2)
10         err_quit("usage: incr_dev_zero <#loops>");
11     nloop = atoi(argv[1]);

12         /* open /dev/zero, map into memory */
13     fd = Open("/dev/zero", O_RDWR);
14     ptr = Mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
15     Close(fd);
```

shm/incr_dev_zero.c

316

图12-15 SVR4 /dev/zero内存映射

6~11 自动变量zero以及指定待创建路径名的命令行参数都被去掉了。

12~15 open文件/dev/zero后把得到的描述符用于mmap调用中。这样的映射保证内存映射区被初始化为0。

12.6 访问内存映射的对象

内存映射一个普通文件时，内存中映射区的大小（mmap的第二个参数）通常等于该文件的大小。例如图12-12中，文件大小由write设置成我们的shared结构的大小，它同时也是内存映射区的大小。然而文件大小和内存映射区大小可以不同。

我们将使用图12-16给出的程序更为细致地探讨mmap函数。

命令行参数

8~11 命令行参数有三个，分别指定即将创建并映射到内存的文件的路径名、该文件将被设置成的大小以及内存映射区的大小。

创建、打开并截短文件；设置文件大小

12~15 待打开的文件若不存在则创建之，若已存在则把它的大小截短成0。接着把该文件的大小设置成由命令行参数指定的大小，办法是把文件读写指针移动到这个大小减去1的字节位置，然后写1个字节。

内存映射文件

16~17 使用作为最后一个命令行参数指定的大小对该文件进行内存映射。其描述符随后被关闭。

输出页面大小

18~19 使用sysconf获取系统实现的页面大小并将其输出。

读出和存入内存映射区

20~26 读出内存映射区中每个页面的首字节和尾字节，并输出它们的值。我们预期这些值全为0。同时把每个页面的这两个字节设置为1。我们预期某个引用会最终引发一个信号，

它将终止程序。当for循环结束时，我们输出下一页的首字节，并预期这会失败（假设此前程序还没有失败）。

shm/test1.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     int      fd, i;
6     char    *ptr;
7     size_t   filesize, mmapsize, pagesize;
8     if (argc != 4)
9         err_quit("usage: test1 <pathname> <filesize> <mmapsize>");
10    filesize = atoi(argv[2]);
11    mmapsize = atoi(argv[3]);
12        /* open file: create or truncate; set file size */
13    fd = Open(argv[1], O_RDWR | O_CREAT | O_TRUNC, FILE_MODE);
14    Lseek(fd, filesize-1, SEEK_SET);
15    Write(fd, "", 1);
16    ptr = Mmap(NULL, mmapsize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
17    Close(fd);
18    pagesize = Sysconf(_SC_PAGESIZE);
19    printf("PAGESIZE = %ld\n", (long) pagesize);
20    for (i = 0; i < max(filesize, mmapsize); i += pagesize) {
21        printf("ptr[%d] = %d\n", i, ptr[i]);
22        ptr[i] = 1;
23        printf("ptr[%d] = %d\n", i + pagesize - 1, ptr[i + pagesize - 1]);
24        ptr[i + pagesize - 1] = 1;
25    }
26    printf("ptr[%d] = %d\n", i, ptr[i]);
27    exit(0);
28 }
```

shm/test1.c

图12-16　访问其大小可能不同于文件大小的内存映射区

我们要展示的第一种情形的前提是：文件大小等于内存映射区大小，但这个大小不是页面大小的倍数。

```
solaris % ls -l foo
foo: No such file or directory
solaris % test1 foo 5000 5000
PAGESIZE = 4096
ptr[0] = 0
ptr[4095] = 0
ptr[4096] = 0
ptr[8191] = 0
Segmentation Fault(coredump)
solaris % ls -l foo
-rw-r--r--   1 rstevens other1     5000 Mar 20 17:18 foo
solaris % od -b -A d foo
0000000 001 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0000016 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
*
0004080 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 001
0004096 001 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
```

```
0004112 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
*
0005000
```

317 ~ 318

页面大小为4096字节，我们能够读完整的第2页（下标为4096~8191），但是访问第3页时（下标为8192）引发SIGSEGV信号，shell将它输出成“Segmentation Fault（分段故障）”。尽管我们把ptr[8191]设置成1，它也不写到foo文件中，因而该文件的大小仍然是5000。内核允许我们读写最后一页中映射区以远部分（内核的内存保护是以页面为单位的）。但是我们写向这部分扩展区的任何内容都不会写到foo文件中。设置成1的其他3个字节（下标分别为0、4905和4906）复制回foo文件，这一点可使用od命令来验证。（-b选项指定以八进制数输出各个字节，-A d选项指定以十进制数输出地址。）图12-17展示了这个例子。

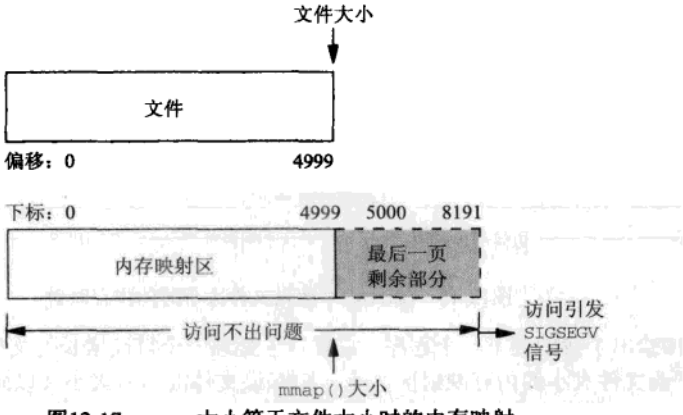


图12-17 mmap大小等于文件大小时的内存映射

在Digital Unix下运行同样的例子，得到的结果类似，不过页面大小现在是8192字节。

```
alpha % ls -l foo
foo not found
alpha % test1 foo 5000 5000
PAGESIZE = 8192
ptr[0] = 0
ptr[8191] = 0
Memory fault(coredump)
alpha % ls -l foo
-rw-r--r--   1 rstevens operator     5000 Mar 21 08:40 foo
```

我们仍然能访问内存映射区以远部分，不过只能在边界所在的那个内存页面内（下标为5000~8191）。访问ptr[8192]将引发SIGSEGV信号，这是我们预期的。

在执行图12-16所示程序的下一个例子中，我们把内存映射区大小（15000字节）指定成大于文件大小（5000字节）。

```
solaris % rm foo
solaris % test1 foo 5000 15000
PAGESIZE = 4096
ptr[0] = 0
ptr[4095] = 0
ptr[4096] = 0
ptr[8191] = 0
Bus Error(coredump)
solaris % ls -l foo
-rw-r--r--   1 rstevens other1     5000 Mar 20 17:37 foo
```

319

其结果与先前那个文件大小等于内存映射区大小（都是5000字节）的例子类似。本例子引发SIGBUS信号（其shell输出为“Bus Error（总线出错）”），前一个例子则引发SIGSEGV信号。两者的差别是，SIGBUS意味着我们是在内存映射区内访问，但是已超出了底层支撑对象的大小。上一个例子中的SIGSEGV则意味着我们已在内存映射区以远访问。可以看出，内核知道被映射的底层支撑对象（本例子中为文件foo）的大小，即使该对象的描述符已经关闭也一样。内核允许我们给mmap指定一个大于该对象大小的大小参数，但是我们访问不了该对象以远的部分（最后一页上该对象以远的那些字节除外，它们的下标为5000~8191）。图12-18展示了这个例子。

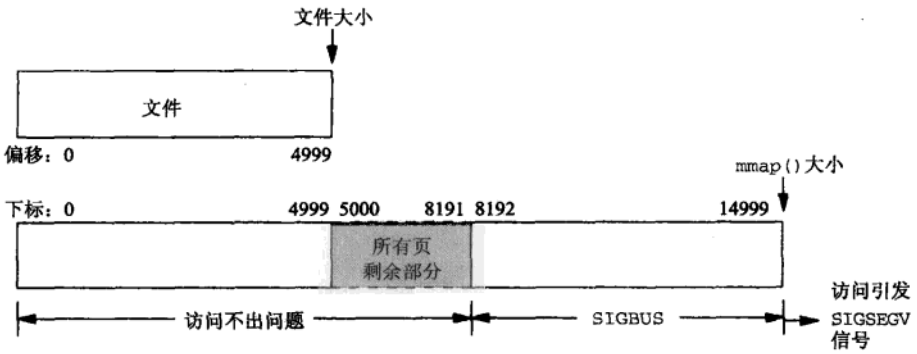


图12-18　mmap大小超过文件大小时的内存映射

图12-19给出了我们的下一个程序。它展示了处理一个持续增长的文件的一种常用技巧：指定一个大于该文件大小的内存映射区大小，跟踪该文件的当前大小（以确保不访问当前文件尾以远的部分），然后就让该文件的大小随着往其中每次写入数据而增长。

shm/test2.c

```
1 #include    "unpipc.h"
2 #define FILE    "test.data"
3 #define SIZE    32768
4 int
5 main(int argc, char **argv)
6 {
7     int     fd, i;
8     char    *ptr;
9         /* open: create or truncate; then mmap file */
10    fd = Open(FILE, O_RDWR | O_CREAT | O_TRUNC, FILE_MODE);
11    ptr = Mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
12    for (i = 4096; i <= SIZE; i += 4096) {
13        printf("setting file size to %d\n", i);
14        Ftruncate(fd, i);
15        printf("ptr[%d] = %d\n", i-1, ptr[i-1]);
16    }
17    exit(0);
18 }
```

shm/test2.c

图12-19　允许文件大小增长的内存映射区例子

打开文件

9~11　打开一个文件，若不存在则创建之，若已存在则把它截短成大小为0。以32768字节的

大小对该文件进行内存映射，尽管它当前的大小为0。

增长文件大小

12~16 通过调用ftruncate（13.3节）把该文件的大小每次增长4096字节，然后取出现在是该文件最后一个字节的那个字节。

现在运行这个程序，我们看到随着文件大小的增长，我们能通过所建立的内存映射区访问新的数据。

```
alpha % ls -l test.data
test.data: No such file or directory
alpha % test2
setting file size to 4096
ptr[4095] = 0
setting file size to 8192
ptr[8191] = 0
setting file size to 12288
ptr[12287] = 0
setting file size to 16384
ptr[16383] = 0
setting file size to 20480
ptr[20479] = 0
setting file size to 24576
ptr[24575] = 0
setting file size to 28672
ptr[28671] = 0
setting file size to 32768
ptr[32767] = 0
alpha % ls -l test.data
-rw-r--r--   1 rstevens other1     32768 Mar 20 17:53 test.data
```

320 ~ 321

本例子表明，内核跟踪着被内存映射的底层支撑对象（本例子中为文件test.data）的大小，而且我们总是能访问在当前文件大小以内又在内存映射区以内的那些字节。在Sloaris 2.6下我们取得了同样的结果。

本节处理的是内存映射文件和mmap。习题13.1中我们要求把这两个程序改为处理Posix共享内存区，将看到相同的结果。

12.7 小结

共享内存区是可用IPC形式中最快的，因为共享内存区中的单个数据副本对于共享该内存区的所有线程或进程都是可用的。然而为协调共享该内存区的各个线程或进程，通常需要某种形式的同步。

本章集中于mmap函数以及普通文件的内存映射，因为这是有亲缘关系的进程间共享内存空间的一种方法。一旦内存映射了一个文件，我们就不再使用read、write和lseek来访问该文件，而只是存取已由mmap映射到该文件的内存位置。把显式的文件I/O操作变换成存取内存单元往往能够简化我们的程序，有时候还能改善性能。

如果设置共享内存区的目的是为了穿越某个后续的fork在父子进程间共享它，那么通过使用匿名内存映射可简化其步骤，这样就不需要创建一个待映射的普通文件。这里或者涉及MAP_ANON这个新标志（适用于源自Berkeley的内核），或者涉及/dev/zero设备文件的映射（适用于源自SVR4的内核）。

我们如此详尽地讨论mmap的理由有两个：一是文件的内存映射是一种很有用的技巧，二是

Posix共享内存区也使用mmap，它是下一章的主题。

Posix还定义了（我们没有讨论过的）处理内存管理的4个额外函数。

- mlockall会使调用进程的整个内存空间常驻内存。munlockall则撤销这种锁定。
- mlock会使调用进程地址空间的某个指定范围常驻内存，该函数的参数指定了这个范围的起始地址以及从该地址算起的字节数。munlock则撤销某个指定内存区的锁定。

习题

12.1　在图12-19中，如果多执行一次for循环内的那段代码，那么会发生什么？

12.2　假设有两个进程，一个是发送者，一个是接收者，前者只是向后者发送消息。再假设它们采用System V消息队列发送消息，请画出消息从发送者去往接收者的示意图。现在假设这两个进程采用我们在
322　5.8节提供的Posix消息队列的实现来发送消息，请画出新的消息传递示意图。

12.3　在讨论mmap的MAP_SHARED标志时，我们说过内核虚拟内存算法将把对内存映像的任何变动更新到实际的文件中。查看/dev/zero的手册页面，判定在内核把对内存映像的变动写回该文件时，发生了什么。

12.4　把图12-10改为指定MAP_PRIVATE标志而不是MAP_SHARED标志，并验证其结果与图12-3的类似。被映射到内存的文件的内容是什么？

12.5　在6.9节中我们提到过，对System V消息队列select读写条件的方法之一是：创建一个匿名共享内存区，派生一个子进程，让该子进程阻塞在msgrcv调用中，以将消息读入到该匿名共享内存区中。父进程还创建两个管道，其中一个管道由子进程用来向父进程通知已在共享内存区中准备好一个消息，另一个管道则由父进程用来向子进程通知共享内存区已可用。这就允许父进程对前一个管道的读出端select可读条件，同时对它想要选择的其他描述符select读写条件。请把上述办法编写成代码。其中匿名共享内存对象的分配调用我们的my_shm函数（图A-46）完成。创建消息队列使用我们在6.6节提供的msgcreate和msgsnd程序，然后把记录放到该队列中。父进程应该只输出由子进
323　程读入的每个消息的大小和类型。

第13章 Posix 共享内存区

13.1 概述

上一章较为笼统地讨论了共享内存区以及mmap函数，并给出了使用mmap提供父子进程间的共享内存区的例子：

- 使用内存映射文件（图12-10）；
- 使用4.4BSD匿名内存映射（图12-14）；
- 使用/dev/zero匿名内存映射（图12-15）。

我们现在把共享内存区的概念扩展到将无亲缘关系进程间共享的内存区包括在内。Posix.1提供了两种在无亲缘关系进程间共享内存区的方法。

(1) 内存映射文件（memory-mapped file）：由open函数打开，由mmap函数把得到的描述符映射到当前进程地址空间中的一个文件。我们在第12章中讲述了这种技术，并给出了它在父子进程间共享内存区时的用法。内存映射文件也可以在无亲缘关系的进程间共享。

(2) 共享内存区对象（shared-memory object）：由shm_open打开一个Posix.1 IPC名字（也许是在文件系统中的一个路径名），所返回的描述符由mmap函数映射到当前进程的地址空间。我们将在本章讲述这种技术。

这两种技术都需要调用mmap，差别在于作为mmap的参数之一的描述符的获取手段：通过open或通过shm_open。图13-1展示了这个差别。Posix把两者合称为内存区对象（memory object）。

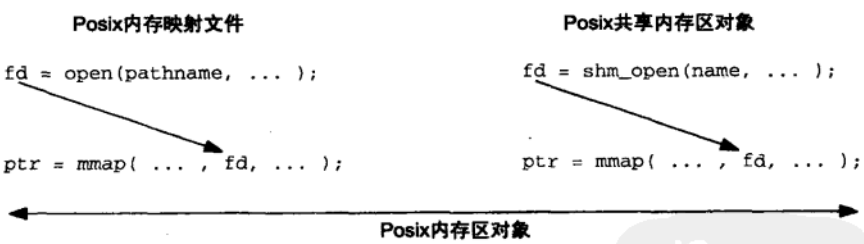


图13-1 Posix内存区对象：内存映射文件和共享内存区对象

13.2 shm_open 和 shm_unlink 函数

Posix共享内存区涉及以下两个步骤要求。

(1) 指定一个名字参数调用shm_open，以创建一个新的共享内存区对象或打开一个已存在的共享内存区对象。

(2) 调用mmap把这个共享内存区映射到调用进程的地址空间。

传递给shm_open的名字参数随后由希望共享该内存区的任何其他进程使用。

> Posix共享内存区采用这样的两步过程，而不是只用单个步骤，即取得一个名字后直接返回调用进程内存空间中的某个地址，其原因在于当Posix发明自己的共享内存区形式时，mmap已经存在。显然，单个函数完全可以做那两步工作。shm_open返回一个描述符（回想一下，mq_open返回一个mqd_t值，sem_open返回一个指向某个sem_t值的指针）的原因是：mmap用于把一个内存区对象映射到调用进程地址空间的是该对象的一个已打开描述符。

```
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
                                        返回：若成功则为非负描述符，若出错则为-1
int shm_unlink(const char *name);
                                        返回：若成功则为0，若出错则为-1
```

326 我们已在2.2节描述过有关*name*参数的规则。

*oflag*参数必须或者含有O_RDONLY（只读）标志，或者含有O_RDWR（读写）标志，还可以指定如下标志：O_CREAT、O_EXCL或O_TRUNC。O_CREAT和O_EXCL标志已在2.3节描述过。如果随O_RDWR指定O_TRUNC标志，而且所需的共享内存区对象已经存在，那么它将被截短成0长度。

*mode*参数指定权限位（图2-4），它在指定了O_CREAT标志的前提下使用。注意，与mq_open和sem_open函数不同，shm_open的*mode*参数总是必须指定。如果没有指定O_CREAT标志，那么该参数可以指定为0。

shm_open的返回值是一个整数描述符，它随后用作mmap的第五个参数。

shm_unlink函数删除一个共享内存区对象的名字。跟所有其他unlink函数（删除文件系统中一个路径名的unlink，删除一个Posix消息队列的mq_unlink，以及删除一个Posix有名信号量的sem_unlink）一样，删除一个名字不会影响对于其底层支撑对象的现有引用，直到对于该对象的引用全部关闭为止。删除一个名字仅仅防止后续的open、mq_open或sem_open调用取得成功。

13.3　ftruncate和fstat函数

处理mmap的时候，普通文件或共享内存区对象的大小都可以通过调用ftruncate修改。

```
#include <unistd.h>
int ftruncate(int fd, off_t length);
                                        返回：若成功则为0，若出错则为-1
```

Posix就该函数对普通文件和共享内存区对象的处理的定义稍有不同。

- 对于一个普通文件：如果该文件的大小大于*length*参数，额外的数据就被丢弃掉。如果该文件的大小小于*length*，那么该文件是否修改以及其大小是否增长是未加说明的。实际上对于一个普通文件，把它的大小扩展到*length*字节的可移植方法是：先lseek到偏移为*length*−1处，然后write 1个字节的数据。所幸的是几乎所有Unix实现都支持使用ftruncate扩展一个文件。
- 对于一个共享内存区对象：ftruncate把该对象的大小设置成*length*字节。

我们调用ftruncate来指定新创建的共享内存区对象的大小，或者修改已存在的对象的大小。当打开一个已存在的共享内存区对象时，我们可调用fstat来获取有关该对象的信息。 327

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

int fstat(int fd, struct stat *buf);
```

返回：若成功则为0，若出错则为-1

stat结构有12个或以上的成员（APUE第4章详细讨论它的所有成员），然而当*fd*指代一个共享内存区对象时，只有四个成员含有信息。

```
struct stat {
  ...
  mode_t    st_mode;    /* mode: S_I{RW}{USR,GRP,OTH} */
  uid_t     st_uid;     /* user ID of owner */
  gid_t     st_gid;     /* group ID of owner */
  off_t     st_size;    /* size in bytes */
  ...
};
```

我们将在下一节给出使用这两个函数的例子。

不幸的是，Posix.1并没有指定一个新创建的共享内存区对象的初始内容。关于shm_open函数的说明只说："（新创建的）共享内存区对象的大小应该为0"。关于ftruncate函数的说明指定，对于一个普通文件（不是共享内存区），"如果其大小被扩展，那么扩展部分应显得好像已用0填写过"。然而同样在关于ftruncate的说明中，却没有任何有关被扩展了的一个共享内存区对象新内容的陈述。Posix.1基本原理声称："如果一个内存区对象被扩展，那么扩展部分内容全为0。"然而这只是基本原理，而不是正式标准。当作者在comp.std.unix新闻组上就此细节提出疑问时，得到的观点是有些厂家反对初始化为0的要求，因为这么做的开销很大。如果一个新扩展的共享内存区未被初始化为某个值（也就是说其内容在扩展前后没有改动），那么有可能成为一个安全漏洞。

13.4 简单的程序

现在开发一些简单的程序来操作Posix共享内存区。

13.4.1 shmcreate 程序

图13-2给出的shmcreate程序以某个指定的名字和长度创建一个共享内存区对象。

pxshm/shmcreate.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5       int     c, fd, flags;
6       char    *ptr;
7       off_t   length;

8       flags = O_RDWR | O_CREAT;
```

图13-2 创建一个具有所指定大小的Posix共享内存区对象

```
 9      while ( (c = Getopt(argc, argv, "e")) != -1) {
10          switch (c) {
11          case 'e':
12              flags |= O_EXCL;
13              break;
14          }
15      }
16      if (optind != argc - 2)
17          err_quit("usage: shmcreate [ -e ] <name> <length>");
18      length = atoi(argv[optind + 1]);

19      fd = Shm_open(argv[optind], flags, FILE_MODE);
20      Ftruncate(fd, length);

21      ptr = Mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

22      exit(0);
23  }
```

pxshm/shmcreate.c

图13-2（续）

19~22　shm_create创建所指定的共享内存区对象。如果指定了-e选项，那么若该对象已经存在则将出错。ftruncate设置该对象的长度，mmap则把它映射到调用进程的地址空间。本程序随后终止。既然Posix共享内存区至少具有随内核的持续性，因此本程序的终止不会删除该共享内存区对象。

13.4.2　shmunlink 程序

图13-3给出的简单程序只是调用shm_unlink从系统中删除一个共享内存区对象的名字。

pxshm/shmunlink.c

```
1 #include     "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     if (argc != 2)
6         err_quit("usage: shmunlink <name>");

7     Shm_unlink(argv[1]);

8     exit(0);
9 }
```

328 ~ 329

pxshm/shmunlink.c

图13-3　删除一个共享内存区对象的名字

13.4.3　shmwrite 程序

图13-4给出了shmwrite程序，它往一个共享内存区对象中写入一个模式：0，1，2，…，254，255，0，1，等等。

10~15　shmopen打开所指定的共享内存区对象，fstat获取其大小信息。使用mmap映射它之后close它的描述符。

16~18　把模式写入该共享内存区。

pxshm/shmwrite.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     int     i, fd;
6     struct stat stat;
7     unsigned char   *ptr;

8     if (argc != 2)
9         err_quit("usage: shmwrite <name>");

10        /* open, get size, map */
11    fd = Shm_open(argv[1], O_RDWR, FILE_MODE);
12    Fstat(fd, &stat);
13    ptr = Mmap(NULL, stat.st_size, PROT_READ | PROT_WRITE,
14               MAP_SHARED, fd, 0);
15    Close(fd);

16        /* set: ptr[0] = 0, ptr[1] = 1, etc. */
17    for (i = 0; i < stat.st_size; i++)
18        *ptr++ = i % 256;

19    exit(0);
20 }
```

pxshm/shmwrite.c

图13-4 打开一个共享内存区对象，填写一个数据模式

13.4.4 shmread 程序

图13-5给出的shmread程序验证由shmwrite写入的模式。

pxshm/shmread.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     int     i, fd;
6     struct stat stat;
7     unsigned char c, *ptr;

8     if (argc != 2)
9         err_quit("usage: shmread <name>");

10        /* open, get size, map */
11    fd = Shm_open(argv[1], O_RDONLY, FILE_MODE);
12    Fstat(fd, &stat);
13    ptr = Mmap(NULL, stat.st_size, PROT_READ,
14               MAP_SHARED, fd, 0);
15    Close(fd);

16        /* check that ptr[0] = 0, ptr[1] = 1, etc. */
17    for (i = 0; i < stat.st_size; i++)
18        if ( (c = *ptr++) != (i % 256))
19            err_ret("ptr[%d] = %d", i, c);

20    exit(0);
21 }
```

pxshm/shmread.c

图13-5 打开一个共享内存区对象，验证其数据模式

10~15　打开所指定的共享内存区对象用于只读，使用fstat获取其大小信息，使用mmap把它映射到内存（用于只读目的），随后关闭其描述符。

16~19　验证由shmwrite写入的模式。

13.4.5　例子

在Digital Unix 4.0B下创建一个长度为123 456字节、名为/tmp/myshm的共享内存区对象。

```
alpha % shmcreate /tmp/myshm 123456
alpha % ls -l /tmp/myshm
-rw-r--r--   1 rstevens system     123456 Dec 10 14:33 /tmp/myshm
alpha % od -c /tmp/myshm
0000000   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0361100
```

我们看到在文件系统中创建了一个同名文件。使用od程序可验证该对象的初始内容为0。（刚刚超过该文件最后一个字节位置的八进制字节偏移0361100，等于十进制的123 456。）

接着运行我们的shmwrite程序，然后使用od验证初始内容与预期的一致。

```
alpha % shmwrite /tmp/myshm
alpha % od -x /tmp/myshm | head -4
0000000  0100 0302 0504 0706 0908 0b0a 0d0c 0f0e
0000020  1110 1312 1514 1716 1918 1b1a 1d1c 1f1e
0000040  2120 2322 2524 2726 2928 2b2a 2d2c 2f2e
0000060  3130 3332 3534 3736 3938 3b3a 3d3c 3f3e
alpha % shmread /tmp/myshm
alpha % shmunlink /tmp/myshm
```

我们使用shmread验证该共享内存区对象的内容后删除其名字。

如果在Solaris 2.6下运行我们的shmcreate程序，我们看到在/tmp目录下创建了一个具有所
331 指定大小的文件。

```
solaris % shmcreate -e /testshm 123
solaris % ls -l/tmp /.*testshm*
-rw-r--r--   1  rstevens other1   123 Dec 10 14:40   /tmp/.SHMtestshm
```

13.4.6　例子

我们现在在图13-6中提供一个简单的例子程序，以展示同一共享内存区对象内存映射到不同进程的地址空间时，起始地址可以不一样。

10~14　创建一个其名字为命令行参数的共享内存区，把它的大小设置为一个整数的大小，然后打开文件/etc/motd。

15~30　fork后父子进程都调用mmap两次，但顺序不一样。父子进程分别输出每个内存映射区的起始地址。子进程接着睡眠5秒，父进程则在共享内存区中写入值777，子进程醒来后输出该值。

运行这个程序，我们发现所指定的共享内存区对象在父子进程中被内存映射到不同的起始地址。

```
solaris % test3 test3.data
parent: shm ptr = eee30000, motd ptr = eee20000
child: shm ptr = eee20000, motd ptr = eee30000
shared memory integer = 777
```

父进程把值777存入0xeee30000位置，子进程却从0xeee20000位置读出该值。父子进程中指针ptr1都指向同一共享内存区，即使每个指针在各自进程内有不同的值也不受影响。

pxshm/test3.c

```
1 #include     "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     fd1, fd2, *ptr1, *ptr2;
6     pid_t   childpid;
7     struct stat stat;
8     if (argc != 2)
9         err_quit("usage: test3 <name>");
10    shm_unlink(Px_ipc_name(argv[1]));
11    fd1 = Shm_open(Px_ipc_name(argv[1]), O_RDWR | O_CREAT | O_EXCL, FILE_MODE);
12    Ftruncate(fd1, sizeof(int));
13    fd2 = Open("/etc/motd", O_RDONLY);
14    Fstat(fd2, &stat);
15    if ( (childpid = Fork()) == 0) {
16            /* child */
17        ptr2 = Mmap(NULL, stat.st_size, PROT_READ, MAP_SHARED, fd2, 0);
18        ptr1 = Mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
19                    MAP_SHARED, fd1, 0);
20        printf("child: shm ptr = %p, motd ptr = %p\n", ptr1, ptr2);
21        sleep(5);
22        printf("shared memory integer = %d\n", *ptr1);
23        exit(0);
24    }
25        /* parent: mmap in reverse order from child */
26    ptr1 = Mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd1, 0);
27    ptr2 = Mmap(NULL, stat.st_size, PROT_READ, MAP_SHARED, fd2, 0);
28    printf("parent: shm ptr = %p, motd ptr = %p\n", ptr1, ptr2);
29    *ptr1 = 777;
30    Waitpid(childpid, NULL, 0);
31    exit(0);
32 }
```

pxshm/test3.c

图13-6 共享内存区在不同进程中可以出现在不同的地址

13.5 给一个共享的计数器持续加 1

现在开发一个类似于12.3节中给出的例子，它由多个进程给存放在共享内存区中的某个计数器持续加1。我们把该计数器存放在一个共享内存区中，并用一个有名信号量来同步，不过不再需要父子进程关系了。既然Posix共享内存区对象和Posix有名信号量都是以名字来访问的，因此将给共享的计数器持续加1的各个进程间可以没有亲缘关系，不过它们都得知道该共享内存区和该信号量的IPC名字，并有访问这两个IPC对象的足够权限。

图13-7给出的服务器程序创建所指定的共享内存区对象，创建并初始化所指定的信号量，然后终止。

创建共享内存区对象

13~19 我们调用shm_unlink以提防所需共享内存区对象已经存在的情况，然后调用shm_open创建该对象。ftruncate将该对象的大小设置成我们的shmstruct结构的大小，该对象本身则随后由mmap映射到调用进程的地址空间。接着关闭该对象的描述符。

pxshm/server1.c

```
 1 #include     "unpipc.h"

 2 struct shmstruct {             /* struct stored in shared memory */
 3     int      count;
 4 };
 5 sem_t    *mutex;               /* pointer to named semaphore */

 6 int
 7 main(int argc, char **argv)
 8 {
 9     int      fd;
10     struct shmstruct*ptr;

11     if (argc != 3)
12         err_quit("usage: server1 <shmname> <semname>");

13     shm_unlink(Px_ipc_name(argv[1]));       /* OK if this fails */
14         /* create shm, set its size, map it, close descriptor */
15     fd = Shm_open(Px_ipc_name(argv[1]), O_RDWR | O_CREAT | O_EXCL, FILE_MODE);
16     Ftruncate(fd, sizeof(struct shmstruct));
17     ptr = Mmap(NULL, sizeof(struct shmstruct), PROT_READ | PROT_WRITE,
18                MAP_SHARED, fd, 0);
19     Close(fd);

20     sem_unlink(Px_ipc_name(argv[2]));       /* OK if this fails */
21     mutex = Sem_open(Px_ipc_name(argv[2]), O_CREAT | O_EXCL, FILE_MODE, 1);
22     Sem_close(mutex);

23     exit(0);
24 }
```

pxshm/server1.c

图13-7　创建并初始化共享内存区和信号量的程序

创建并初始化信号量

20~22　我们调用sem_unlink以提防所需信号量已经存在的情况，然后调用sem_open创建该有名信号量，并把它初始化为1。将给存放在所创建的共享内存区对象中的计数器加1的任何进程都会把该信号量用作一个互斥锁。接着关闭该信号量。

终止

23　进程终止。既然Posix共享内存区至少具有随内核的持续性，因此所创建的该对象将继续存在，直到它的所有打开着的引用都关闭（当本进程终止时，该对象不再有打开着的引用）并且该对象被显式地删除为止。

我们的程序必须给共享内存区和信号量使用不同的名字。操作系统并不保证给IPC名字加上点什么以区分消息队列、信号量和共享内存区。我们已看到Solaris给这三种IPC类型的名字分别加上.MQ、.SEM和.SHM的前缀，但是Digital Unix却没有这样做。

图13-8给出了我们的客户程序，它对存放在共享内存区中的计数器执行一定次数的加1操作，每次给该计数器加1时都事先获取保护它的信号量。

打开共享内存区

15~18　shm_open打开所指定的共享内存区对象，该对象必须存在（因为没有指定O_CREAT标志）。使用mmap把该内存区映射到调用进程的地址空间，然后关闭它的描述符。

打开信号量

19　打开所指定的有名信号量。

pxshm/client1.c

```
 1 #include    "unpipc.h"
 2 struct shmstruct {              /* struct stored in shared memory */
 3     int     count;
 4 };
 5 sem_t   *mutex;                 /* pointer to named semaphore */
 6 int
 7 main(int argc, char **argv)
 8 {
 9     int     fd, i, nloop;
10     pid_t   pid;
11     struct shmstruct*ptr;
12     if (argc != 4)
13         err_quit("usage: client1 <shmname> <semname> <#loops>");
14     nloop = atoi(argv[3]);
15     fd = Shm_open(Px_ipc_name(argv[1]), O_RDWR, FILE_MODE);
16     ptr = Mmap(NULL, sizeof(struct shmstruct), PROT_READ | PROT_WRITE,
17                MAP_SHARED, fd, 0);
18     Close(fd);
19     mutex = Sem_open(Px_ipc_name(argv[2]), 0);
20     pid = getpid();
21     for (i = 0; i < nloop; i++) {
22         Sem_wait(mutex);
23         printf("pid %ld: %d\n", (long) pid, ptr->count++);
24         Sem_post(mutex);
25     }
26     exit(0);
27 }
```

pxshm/client1.c

图13-8 给存放在共享内存区中的一个计数器加1的程序

获取信号量并给计数器持续加1

20~26 给存放在所打开共享内存区中的计数器执行由命令行参数指定次数的加1操作。每次加1前输出该计数器原来的值以及当前的进程ID，输出进程ID是因为我们将同时运行本程序的多个副本。

我们首先启动服务器，然后在后台运行客户程序的三个副本。

```
solaris % server1 shm1 sem1                    创建并初始化共享内存区和信号量

solaris % client1 shm1 sem1 10000 &    client1 shm1 sem1 10000 & \
client1 shm1 sem1 10000 &
[2] 17976                                      由shell输出的各个进程ID
[3] 17977
[4] 17978
pid 17977: 0                                   进程17977首先运行
pid 17977: 1
. . .                                          进程17977继续运行
pid 17977: 32
pid 17976: 33                                  内核切换上下文到进程17976
. . .                                          进程17976继续运行
pid 17976: 707
pid 17978: 708                                 内核切换上下文到进程17978
. . .                                          进程17978继续运行
pid 17978: 852
pid 17977: 853                                 内核切换上下文到进程17977
```

332~335

```
. . .                                   如此等等
pid 17977: 29998
pid 17977: 29999                        最终值输出，它是正确的
```

13.6　向一个服务器发送消息

现在对我们的生产者-消费者例子作如下修改。服务器启动后创建一个共享内存区对象，各个客户进程就在其中放置消息。我们的服务器只是输出这些消息，不过可以一般化为做类似于syslog守护进程所做之事，该守护进程在UNPv1第13章中讲述。我们把其他进程称为客户，因为它们相对于我们的服务器呈现为客户，但是它们也可以是某种处理其他客户的服务器。举例来说，Telnet服务器在向syslog守护进程发送登记消息时就是后者的一个客户。

我们没有使用第2部分中讲述的某种消息传递技术，而是使用共享内存区来容纳消息。当然，这使得我们有必要在存入消息的各个客户和取走并输出消息的服务器之间采取某种形式的同步。图13-9展示了总体设计。

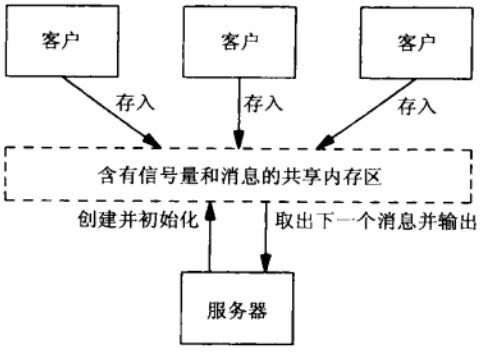


图13-9　多个客户通过共享内存区向一个服务器发送消息

这儿有多个生产者（客户）和单个消费者（服务器）。共享内存区既出现在服务器的地址空间，也出现在各个客户的地址空间。

图13-10是我们的cliserv2.h头文件，它定义了一个给出共享内存区对象布局的结构。

pxshm/cliserv2.h

```
 1 #include    "unpipc.h"

 2 #define MESGSIZE    256         /* max #bytes per message, incl. null at end */
 3 #define NMESG        16         /* max #messages */

 4 struct shmstruct {              /* struct stored in shared memory */
 5     sem_t   mutex;             /* three Posix memory-based semaphores */
 6     sem_t   nempty;
 7     sem_t   nstored;
 8     int     nput;              /* index into msgoff[] for next put */
 9     long    noverflow;         /* #overflows by senders */
10     sem_t   noverflowmutex;    /* mutex for noverflow counter */
11     long    msgoff[NMESG];     /* offset in shared memory of each message */
12     char    msgdata[NMESG * MESGSIZE];      /* the actual messages */
13 };
```

pxshm/cliserv2.h

图13-10　定义共享内存区布局的头文件

基本的信号量和变量

5~8 mutex、nempty和nstored这三个Posix基于内存的信号量与10.6节里生产者-消费者例子中的同名信号量作用相同。变量nput是用于存放一个消息的下一个位置的下标（0、1、…、NMESG-1）。既然我们有多个生产者，该变量就必须存放在共享内存区中，并且只能在持有mutex期间访问。

溢出计数器

9~10 某个客户想发送一个消息，但是所有的消息槽位都被占用了，发生这种情况的可能性是存在的。但是如果该客户实际上同时又是某种类型的一个服务器（譬如说是一个FTP服务器或HTTP服务器），那么它可能不愿意等待服务器释放出一个槽位。因此，我们将把客户程序编写成发生这种情况时并不阻塞，而是给noverflow计数器加1。由于该溢出计数器也是在所有客户和服务器之间共享的，因此它也需要一个互斥锁，以免其值遭受破坏。

消息偏移和数据

11~12 数组msgoff含有针对msgdata数组的各个偏移，指出了每个消息的起始位置。这就是说，msgoff[0]为0，msgoff[1]为256（MESGSIZE的值），msgoff[2]为512，等等。必须搞清楚在处理共享内存区时，我们只能使用像这样子的偏移（offset），因为共享内存区对象可映射到映射它的各个进程的不同物理地址。也就是说，对于同一个共享内存区对象，调用mmap的每个进程所得到的mmap返回值可能不同。由于这个原因，我们不能在共享内存区对象中使用指针（pointer），因为它们含有存放在这些对象内各变量的实际地址。

图13-11是我们的服务器程序，它等待某个客户往所指定的共享内存区中放置一个消息，然后输出这个消息。

创建共享内存区对象

10~16 首先调用shm_unlink删除可能仍然存在的共享内存区对象。接着使用shm_open创建这个对象，再使用mmap把它映射到调用进程的地址空间。然后关闭它的描述符。

初始化偏移量数组

17~19 把偏移量数组msgoff初始化为含有每个消息的位置偏移。

336 ~ 337

初始化信号量

20~24 初始化存放在共享内存区对象中的四个基于内存的信号量。每个sem_init调用的第二个参数都不为0，因为这些信号量将在进程间共享。

等待消息，然后输出

25~36 for循环的前半部分是标准的消费者算法：等待nstored变为大于0，等待mutex，处理数据，释放mutex，然后给nempty加1。

处理溢出

37~43 每次经由这个循环，我们还检查是否溢出。我们测试计数器noverflows的值是否不同于上一次的值，若是则输出并保存这个新值。注意，我们是在持有noverflowmutex信号量期间获取该计数器的值的，但在比较并输出它之前先释放了这个信号量。这么一来展示了如下的一般规则：我们应该把持有某个互斥锁期间执行的代码编写得操作总数尽量地少。

pxshm/server2.c

```
 1 #include    "cliserv2.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     fd, index, lastnoverflow, temp;
 6     long    offset;
 7     struct shmstruct*ptr;
 8     if (argc != 2)
 9         err_quit("usage: server2 <name>");
10         /* create shm, set its size, map it, close descriptor */
11     shm_unlink(Px_ipc_name(argv[1]));       /* OK if this fails */
12     fd = Shm_open(Px_ipc_name(argv[1]), O_RDWR | O_CREAT | O_EXCL, FILE_MODE);
13     ptr = Mmap(NULL, sizeof(struct shmstruct), PROT_READ | PROT_WRITE,
14                MAP_SHARED, fd, 0);
15     Ftruncate(fd, sizeof(struct shmstruct));
16     Close(fd);
17         /* initialize the array of offsets */
18     for (index = 0; index < NMESG; index++)
19         ptr->msgoff[index] = index * MESGSIZE;
20         /* initialize the semaphores in shared memory */
21     Sem_init(&ptr->mutex, 1, 1);
22     Sem_init(&ptr->nempty, 1, NMESG);
23     Sem_init(&ptr->nstored, 1, 0);
24     Sem_init(&ptr->noverflowmutex, 1, 1);
25         /* this program is the consumer */
26     index = 0;
27     lastnoverflow = 0;
28     for ( ; ; ) {
29         Sem_wait(&ptr->nstored);
30         Sem_wait(&ptr->mutex);
31         offset = ptr->msgoff[index];
32         printf("index = %d: %s\n", index, &ptr->msgdata[offset]);
33         if (++index >= NMESG)
34             index = 0;              /* circular buffer */
35         Sem_post(&ptr->mutex);
36         Sem_post(&ptr->nempty);
37         Sem_wait(&ptr->noverflowmutex);
38         temp = ptr->noverflow;          /* don't printf while mutex held */
39         Sem_post(&ptr->noverflowmutex);
40         if (temp != lastnoverflow) {
41             printf("noverflow = %d\n", temp);
42             lastnoverflow = temp;
43         }
44     }
45     exit(0);
46 }
```

pxshm/server2.c

图13-11　从共享内存区中取得并输出消息的服务器程序

图13-12给出了我们的客户程序。

pxshm/client2.c

```
 1 #include    "cliserv2.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     fd, i, nloop, nusec;
 6     pid_t   pid;
 7     char    mesg[MESGSIZE];
 8     long    offset;
 9     struct shmstruct*ptr;
10     if (argc != 4)
11         err_quit("usage: client2 <name> <#loops> <#usec>");
12     nloop = atoi(argv[2]);
13     nusec = atoi(argv[3]);
14         /* open and map shared memory that server must create */
15     fd = Shm_open(Px_ipc_name(argv[1]), O_RDWR, FILE_MODE);
16     ptr = Mmap(NULL, sizeof(struct shmstruct), PROT_READ | PROT_WRITE,
17                MAP_SHARED, fd, 0);
18     Close(fd);
19     pid = getpid();
20     for (i = 0; i < nloop; i++) {
21         Sleep_us(nusec);
22         snprintf(mesg, MESGSIZE, "pid %ld: message %d", (long) pid, i);
23         if (sem_trywait(&ptr->nempty) == -1) {
24             if (errno == EAGAIN) {
25                 Sem_wait(&ptr->noverflowmutex);
26                 ptr->noverflow++;
27                 Sem_post(&ptr->noverflowmutex);
28                 continue;
29             } else
30                 err_sys("sem_trywait error");
31         }
32         Sem_wait(&ptr->mutex);
33         offset = ptr->msgoff[ptr->nput];
34         if (++(ptr->nput) >= NMESG)
35             ptr->nput = 0;          /* circular buffer */
36         Sem_post(&ptr->mutex);
37         strcpy(&ptr->msgdata[offset], mesg);
38         Sem_post(&ptr->nstored);
39     }
40     exit(0);
41 }
```

pxshm/client2.c

图13-12 在共享内存区中给服务器存放消息的客户程序

命令行参数

10~13 第一个命令行参数是共享内存区对象的名字，下一个是给服务器存放的消息数，最后一个是每次存放消息之间停顿的微秒数。通过启动本客户程序的多个副本并指定一个较小的停顿值，我们可以强行造成溢出，然后验证服务器能正确地处理它。

打开并映射共享内存区

14~18 在假设所指定的共享内存区对象已经存在的前提下，我们打开该对象，然后把它映射到当前进程的地址空间。随后关闭它的描述符。

存放消息

19~31　客户程序接着依循基本的生产者算法工作，不过我们把缓冲区中没有存放新消息的空间时生产者阻塞在其中的sem_wait(nempty)调用换成了不会阻塞的sem_trywait调用。如果nempty信号量的值为0，该函数就返回一个EAGAIN错误。我们检测出该错误后给溢出计数器加1。

sleep_us是来自APUE图C.9和图C.10的一个函数。它睡眠指定数目的微秒数，是通过调用select和poll来实现的。

32~37　我们在持有mutex信号量期间取得offset的值并给nput加1，但在接下去把新消息复制到共享内存区之前却释放了mutex。在持有该信号量期间，我们只应该执行那些必须被保护起来的操作。

我们首先在后台启动服务器，然后运行一个客户，给它指定50个待存放消息，每个消息的存放没有彼此间的停顿。

```
solaris % server2 serv2 &
[2]       27223
solaris % client2 serv2 50 0
index = 0: pid 27224: message 0
index = 1: pid 27224: message 1
index = 2: pid 27224: message 2
. . .                                    如此继续
index = 15: pid 27224: message 47
index = 0: pid 27224: message 48
index = 1: pid 27224: message 49         没有消息丢失
```

但是当我们再次运行一个客户时，却看到了一些溢出现象：

```
solaris %  client2 serv2 50 0
index = 2: pid 27228: message 0
index = 3: pid 27228: message 1
. . .                                    仍然正常
index = 10: pid 27228: message 8
index = 11: pid 27228: message 9
noverflow = 25                           服务器检测到有25个消息丢失
index = 12: pid 27228: message 10
index = 13: pid 27228: message 11
. . .                                    消息12~22仍然正常
index = 9: pid 27228: message 23
index = 10: pid 27228: message 24
```

这一次该客户呈现为存放了消息0~9，它们由服务器取走并输出。该客户继续运行，准备存放消息10~49，但是共享内存区中只有存放其中前15个消息的空间，于是剩余的25个消息（编号为25~49）因溢出而未被存放。

很明显，在本例子中通过让客户尽可能快地产生消息，而且每次存放消息之间没有停顿来达到溢出效果，不过这并不是一种典型的现实情形。然而本例子的目的只是展示客户产生的消息没有存放空间可用，但是客户又不想为此而阻塞的情况应如何处理。这种情况并不是只有共享内存区才有的，消息队列、管道和FIFO都可能发生同样情况。

不断提供数据，造成接收者忙不过来的情形并非只有本例子出现。UNPv1的8.13节就UDP数据报和UDP套接字接收缓冲区讨论了同样情形。TCPv3的18.2节讲述了接收者的缓冲区发生溢出时，Unix域数据报套接字是如何向发送者返回一个ENOBUFS错误的。在图13-12中，我们的客户（发送者）知道什么时候服务器的缓冲区溢出了，因此要是把这段代码放到某个供其

他程序调用的通用函数中，那么当服务器的缓冲区溢出时，该函数有可能向调用者返回一个错误。 338 ~ 341

13.7 小结

Posix共享内存区构筑在上一章讲述的mmap函数之上。我们首先指定待打开共享内存区的Posix IPC名字来调用shm_open，取得一个描述符后使用mmap把它映射到内存。其结果类似于内存映射文件，不过共享内存区对象不必作为一个文件来实现。

既然共享内存区对象是由描述符来表示的，它们的大小就使用ftruncate来设置，有关某个已存在对象的信息（保护位、用户ID、组ID及大小）由fstat返回。

在讨论Posix消息队列和Posix信号量时，我们分别在5.8节和10.15节提供了它们基于内存映射I/O的实现。我们不对Posix共享内存区提供这样的实现，因为实在太简单。如果愿意以内存映射一个文件的方式（这在Solaris和Digital Unix上都有实现）来实现Posix共享内存区这种IPC形式，那么shm_open是通过调用open实现的，shm_unlink是通过调用unlink实现的。

习题

13.1 把图12-16和图12-19修改成访问Posix共享内存区而不是内存映射文件，并验证它们的运行结果与原来访问内存映射文件的程序相同。

13.2 在图13-4和图13-5的for循环中，用于步进访问数组元素的C表达式为*ptr++。改用ptr[i]更为可取吗？ 342

第14章 System V 共享内存区

14.1 概述

System V共享内存区在概念上类似于Posix共享内存区。代之以调用shm_open后调用mmap的是，先调用shmget，再调用shmat。

对于每个共享内存区，内核维护如下的信息结构，它定义在<sys/shm.h>头文件中：

```
struct shmid_ds {
  struct ipc_perm  shm_perm;    /* operation permission struct */
  size_t           shm_segsz;   /* segment size */
  pid_t            shm_lpid;    /* pid of last operation */
  pid_t            shm_cpid;    /* creator pid */
  shmatt_t         shm_nattch;  /* current # attached */
  shmat_t          shm_cnattch; /* in-core # attached */
  time_t           shm_atime;   /* last attach time */
  time_t           shm_dtime;   /* last detach time */
  time_t           shm_ctime;   /* last change time of this structure */
};
```

我们已在3.3节描述过其中的ipc_perm结构，它含有本共享内存区的访问权限。

14.2 shmget 函数

343

shmget函数创建一个新的共享内存区，或者访问一个已存在的共享内存区。

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

int shmget(key_t key, size_t size, int oflag);
                                        返回：若成功则为共享内存区对象，若出错则为-1
```

返回值是一个称为共享内存区标识符（shared memory identifier）的整数，其他三个shmXXX函数就用它来指代这个内存区。

*key*既可以是ftok的返回值，也可以是IPC_PRIVATE，我们已在3.2节讨论过。

*size*以字节为单位指定内存区的大小。当实际操作为创建一个新的共享内存区时，必须指定一个不为0的*size*值。如果实际操作为访问一个已存在的共享内存区，那么*size*应为0。

*oflag*是图3-6中所示读写权限值的组合。它还可以与IPC_CREAT或IPC_CREAT | IPC_EXCL按位或，如随图3-4所作的讨论。

当实际操作为创建一个新的共享内存区时，该内存区被初始化为*size*字节的0。

注意，shmget创建或打开一个共享内存区，但并没有给调用进程提供访问该内存区的手段。这是shmat函数的目的，我们将接下去讲述它。

14.3 shmat 函数

由shmget创建或打开一个共享内存区后，通过调用shmat把它附接到调用进程的地址空间。

```
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int flag);
                                    返回：若成功则为映射区的起始地址，若出错则为-1
```

其中*shmid*是由shmget返回的标识符。shmat的返回值是所指定的共享内存区在调用进程内的起始地址。确定这个地址的规则如下。

- 如果*shmaddr*是一个空指针，那么系统替调用者选择地址。这是推荐的（也是可移植性最好的）方法。
- 如果*shmaddr*一是一个非空指针，那么返回地址取决于调用者是否给*flag*参数指定了SHM_RND值：
  - 如果没有指定SHM_RND，那么相应的共享内存区附接到由*shmaddr*参数指定的地址；
  - 如果指定了SHM_RND，那么相应的共享内存区附接到由*shmaddr*参数指定的地址向下舍入一个SHMLBA常值。LBA代表“低端边界地址（lower boundary address）”。 344

默认情况下，只要调用进程具有某个共享内存区的读写权限，它附接该内存区后就能够同时读写该内存区。*flag*参数中也可以指定SHM_RDONLY值，它限定只读访问。

14.4 shmdt 函数

当一个进程完成某个共享内存区的使用时，它可调用shmdt断接这个内存区。

```
#include <sys/shm.h>
int shmdt(const void *shmaddr);
                                    返回：若成功则为0，若出错则为-1
```

当一个进程终止时，它当前附接着的所有共享内存区都自动断接掉。

注意本函数调用并不删除所指定的共享内存区。这个删除工作通过以IPC_RMID命令调用shmctl完成，我们将在下一节讲述它。

14.5 shmctl 函数

shmctl提供了对一个共享内存区的多种操作。

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buff);
                                    返回：若成功则为0，若出错则为-1
```

该函数提供了三个命令。

IPC_RMID　从系统中删除由*shmid*标识的共享内存区并拆除它。[①]

IPC_SET　给所指定的共享内存区设置其shmid_ds结构的以下三个成员：shm_perm.uid、shm_perm.gid和shm_perm.mode，它们的值来自*buff*参数指向的结构中的相应成员。shm_ctime的值也用当前时间替换。

345 IPC_STAT　（通过*buff*参数）向调用者返回所指定共享内存区当前的shmid_ds结构。

14.6　简单的程序

我们现在开发一些对System V共享内存区进行操作的简单程序。

14.6.1　shmget 程序

图14-1给出的shmget程序使用指定的路径名和长度创建一个共享内存区。

svshm/shmget.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     int      c, id, oflag;
6     char    *ptr;
7     size_t   length;

8     oflag = SVSHM_MODE | IPC_CREAT;
9     while ( (c = Getopt(argc, argv, "e")) != -1) {
10        switch (c) {
11        case 'e':
12            oflag |= IPC_EXCL;
13            break;
14        }
15    }
16    if (optind != argc - 2)
17        err_quit("usage: shmget [ -e ] <pathname> <length>");
18    length = atoi(argv[optind + 1]);

19    id = Shmget(Ftok(argv[optind], 0), length, oflag);
20    ptr = Shmat(id, NULL, 0);

21    exit(0);
22 }
```

svshm/shmget.c

图14-1　创建一个指定大小的System V共享内存区

19　shmget创建由用户指定其名字和大小的共享内存区。作为命令行参数传递进来的路径名由ftok映射成一个System V IPC键。如果指定了-e选项，那么一旦该内存区已存在就会出错。如果我们知道该内存区已存在，那么在命令行上的长度参数必须指定为0。

20　shmat把该内存区附接到当前进程的地址空间。本程序然后终止，不过既然System V共享内存区至少具有随内核的持续性，那么这不会删除该共享内存区。

① 删除一个共享内存区指的是使其标识符失效，这样以后针对该标识符的shmat、shmdt和shmctl函数调用必定失败。拆除一个共享内存区指的是释放或回收与它对应的数据结构，包括删除存放在其上的数据。拆除操作要到该共享内存区的引用计数变为0时才进行。另外，当某个shmdt调用发现所指定的共享内存区的引用计数变为0时也顺便拆除它，这就是shmctl的IPC_RMID命令先于最后一个shmdt调用发出时会发生的情形。——译者注

14.6.2 shmrmid 程序

图14-2给出的简单程序只是以一个IPC_RMID命令调用shmctl，以便从系统中删除一个共享内存区。

346

svshm/shmrmid.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     id;

 6     if (argc != 2)
 7         err_quit("usage: shmrmid <pathname>");

 8     id = Shmget(Ftok(argv[1], 0), 0, SVSHM_MODE);
 9     Shmctl(id, IPC_RMID, NULL);

10     exit(0);
11 }
```

svshm/shmrmid.c

图14-2 删除一个System V共享内存区

14.6.3 shmwrite 程序

图14-3给出了shmwrite程序，它往一个共享内存区中写入一个模式：0、1、2、…、254、255、0、1，等等。

svshm/shmwrite.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, id;
 6     struct shmid_ds buff;
 7     unsigned char *ptr;

 8     if (argc != 2)
 9         err_quit("usage: shmwrite <pathname>");

10     id = Shmget(Ftok(argv[1], 0), 0, SVSHM_MODE);
11     ptr = Shmat(id, NULL, 0);
12     Shmctl(id, IPC_STAT, &buff);

13         /* set: ptr[0] = 0, ptr[1] = 1, etc. */
14     for (i = 0; i < buff.shm_segsz; i++)
15         *ptr++ = i % 256;

16     exit(0);
17 }
```

svshm/shmwrite.c

图14-3 打开一个共享内存区，填入一个数据模式

10~12 使用shmget打开所指定的共享内存区后由shmat把它附接到当前进程的地址空间。其大小通过以一个IPC_STAT命令调用shmctl取得。

13~15 往该共享内存区中写入给定的模式。

347

14.6.4　shmread程序

图14-4给出的shmread程序会验证由shmwrite写入的模式。

svshm/shmread.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     i, id;
 6     struct shmid_ds buff;
 7     unsigned char   c, *ptr;

 8     if (argc != 2)
 9         err_quit("usage: shmread <pathname>");

10     id = Shmget(Ftok(argv[1], 0), 0, SVSHM_MODE);
11     ptr = Shmat(id, NULL, 0);
12     Shmctl(id, IPC_STAT, &buff);

13         /* check that ptr[0] = 0, ptr[1] = 1, etc. */
14     for (i = 0; i < buff.shm_segsz; i++)
15         if ( (c = *ptr++) != (i % 256))
16             err_ret("ptr[%d] = %d", i, c);

17     exit(0);
18 }
```

svshm/shmread.c

图14-4　打开一个共享内存区，验证其数据模式

10~12　打开并附接所指定的共享内存区。其大小通过以一个IPC_STAT命令调用shmctl获取。

13~16　验证由shmwrite写入的模式。

14.6.5　例子

在Solaris 2.6下创建一个大小为1234字节的共享内存区。用于标识该内存区的路径名（也就是传递给ftok的路径名）是我们的shmget可执行文件的路径名。对于一个给定的应用，使用服务器的可执行文件路径名往往能够提供一个唯一的标识符。

```
solaris % shmget shmget 1234
solaris % ipcs -bmo
IPC status from <running system> as of Thu Jan  8 13:17:06 1998
T         ID      KEY        MODE        OWNER    GROUP NATTCH      SEGSZ
Shared Memory:
m          1   0x0000f12a --rw-r--r-- rstevens   other1       0       1234
```

我们运行ipcs程序以验证相应的共享内存区已经创建出来。注意它的附接数（存放在该内存区的shmid_ds结构的shm_nattch成员中）为0，跟我们预期的一致。

348

接着运行我们的shmwrite程序，以把该共享内存区的内容设置成给定的模式。然后用shmread验证该共享内存区的内容，并删除其标识符。

```
solaris % shmwrite shmget
solaris % shmread shmget
solaris % shmrmid shmget
solaris % ipcs -bmo
IPC status from <running system> as of Thu Jan  8 13:17:06 1998
T         ID      KEY        MODE        OWNER    GROUP NATTCH      SEGSZ
Shared Memory:
```

我们运行ipcs来验证该共享内存区确实已被删除。

当把服务器可执行文件的名字用作ftok的参数来标识某种形式的System V IPC时，通常应使用绝对路径名（例如/usr/bin/myserverd），而不是像本例子那样使用一个相对路径名（shmget）。我们能在本节的例子中使用相对路径名的原因是，所有程序都是从含有服务器可执行文件的目录中运行的。我们知道ftok使用文件的索引节点来构成IPC标识符（见图3-2），而一个给定文件是使用一个绝对路径名还是一个相对路径名来引用对于其索引节点并没有影响。

14.7 共享内存区限制

跟System V消息队列和System V信号量一样，System V共享内存区也存在特定的系统限制（3.8节）。图14-5给出了本书所用两种不同实现的这些限制值。其中第一栏是含有当前限制值的内核变量的传统System V名字。

名　字	说　明	DUnix 4.0B	Solaris 2.6
shmmax	一个共享内存区的最大字节数	4 194 304	1 048 576
shmmnb	一个共享内存区的最小字节数	1	1
shmmni	系统范围最大共享内存区标识符数	128	100
shmseg	每个进程附接的最大共享内存区数	32	6

图14-5 System V共享内存区的典型系统限制

例子

图14-6中的程序可确定图14-5中给出的四个限制值。

svshm/limits.c

```
1 #include     "unpipc.h"
2 #define MAX_NIDS     4096
3 int
4 main(int argc, char **argv)
5 {
6       int     i, j, shmid[MAX_NIDS];
7       void    *addr[MAX_NIDS];
8       unsigned long   size;
9           /* see how many identifiers we can "open" */
10      for (i = 0; i <= MAX_NIDS; i++) {
11          shmid[i] = shmget(IPC_PRIVATE, 1024, SVSHM_MODE | IPC_CREAT);
12          if (shmid[i] == -1) {
13              printf("%d identifiers open at once\n", i);
14              break;
15          }
16      }
17      for (j = 0; j < i; j++)
18          Shmctl(shmid[j], IPC_RMID, NULL);
19          /* now see how many we can "attach" */
20      for (i = 0; i <= MAX_NIDS; i++) {
```

图14-6 确定共享内存区的系统限制

```
21          shmid[i] = Shmget(IPC_PRIVATE, 1024, SVSHM_MODE | IPC_CREAT);
22          addr[i] = shmat(shmid[i], NULL, 0);
23          if (addr[i] == (void *) -1) {
24              printf("%d shared memory segments attached at once\n", i);
25              Shmctl(shmid[i], IPC_RMID, NULL);    /* the one that failed */
26              break;
27          }
28      }
29      for (j = 0; j < i; j++) {
30          Shmdt(addr[j]);
31          Shmctl(shmid[j], IPC_RMID, NULL);
32      }

33          /* see how small a shared memory segment we can create */
34      for (size = 1; ; size++) {
35          shmid[0] = shmget(IPC_PRIVATE, size, SVSHM_MODE | IPC_CREAT);
36          if (shmid[0] != -1) {      /* stop on first success */
37              printf("minimum size of shared memory segment = %lu\n", size);
38              Shmctl(shmid[0], IPC_RMID, NULL);
39              break;
40          }
41      }

42          /* see how large a shared memory segment we can create */
43      for (size = 65536; ; size += 4096) {
44          shmid[0] = shmget(IPC_PRIVATE, size, SVSHM_MODE | IPC_CREAT);
45          if (shmid[0] == -1) {      /* stop on first failure */
46              printf("maximum size of shared memory segment = %lu\n", size-4096);
47              break;
48          }
49          Shmctl(shmid[0], IPC_RMID, NULL);
50      }

51      exit(0);
52  }
```

svshm/limits.c

350

图14-6（续）

在Digital Unix 4.0B下运行这个程序的结果为：

```
alpha % limits
127 identifiers open at once
32 shared memory segments attached at once
minimum size of shared memory segment = 1
maximum size of shared memory segment = 4194304
```

图14-5指出128个标识符的限制，但我们的程序只能创建127个，其原因在于有一个系统守护进程早已创建了一个共享内存区。

14.8　小结

System V共享内存区在概念上与Posix共享内存区类似。最常用的函数调用有以下几个。

- shmget：获取一个标识符。
- shmat：把一个共享内存区附接到调用进程的地址空间。
- 以一个IPC_STAT命令调用shmctl：获取一个已存在共享内存区的大小。
- 以一个IPC_RMID命令调用shmctl：删除一个共享内存区对象。

两者的差别之一是，Posix共享内存区对象的大小可在任何时刻通过调用ftruncate修改（如习题13.1中所展示的那样），而System V共享内存区对象的大小是在调用shmget创建时固定下来的。

习题

14.1 图6-8是对图6-6的修改，它接受的用于指定队列的是标识符而不是路径名。我们已展示有了这种标识符，就足以访问一个System V消息队列（假设我们有足够权限）。对图14-4作类似的修改，以展示同样的特性也适用于System V共享内存区。

351

第五部分

远程过程调用

第15章 门

15.1 概述

当讨论客户-服务器情形和过程调用时，存在着三种不同类型的过程调用，如图15-1所示。

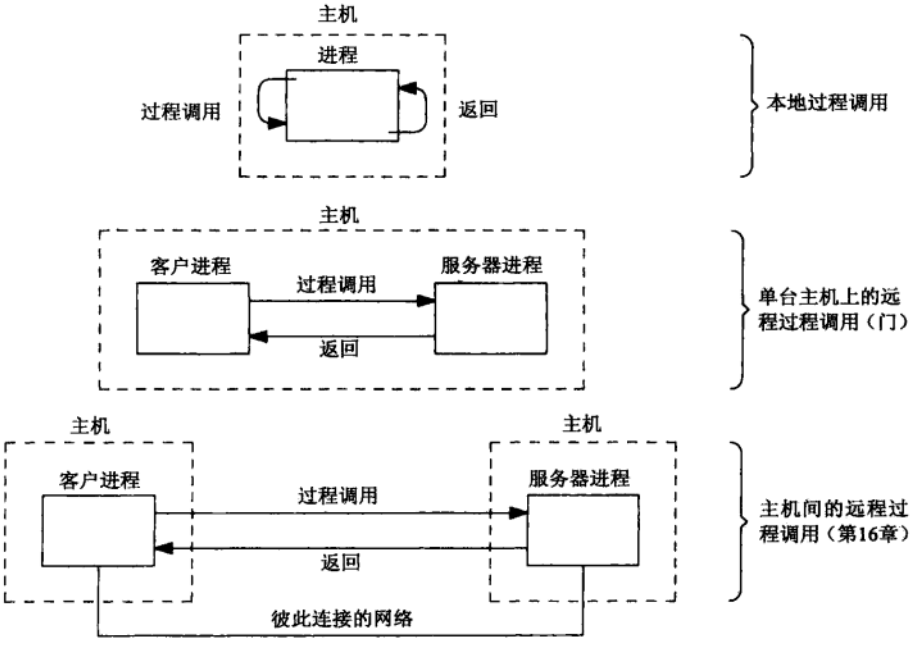


图15-1 三种不同类型的过程调用

(1) 本地过程调用（local procedure call）：这是我们从日常的C编程中早就熟知的过程调用，也就是被调用的过程（函数）与调用过程处于同一个进程中。典型情况是，调用者通过执行某条机器指令把控制传给新过程，被调用过程保存机器寄存器的值，并在栈顶分配存放其本地变量的空间。

(2) 远程过程调用（remote procedure call，简称RPC）：被调用过程和调用过程处于不同的进程中。我们通常称调用者为客户，称被调用的过程为服务器。在图15-1中间的情形中，我们展示客户和服务器是在同一台主机上执行的。它是该图底部的情形经常会发生的一种特殊情况，也是门（door）所提供的能力：一个进程调用同一台主机上另一个进程中的某个过程（函数）。通过给本进程内的某个过程创建一个门，一个进程（服务器）就使得该过程能为其他进程（客户）所调用。我们也可以认为门是一种特殊类型的IPC，因为客户和服务器之间以函数参数和返

回值形式交换信息。

(3) RPC通常允许一台主机上的某个客户调用另一台主机上的某个服务器过程，只要这两台主机以某种形式的网络连接着（图15-1底部的情形）。我们将在第16章讲述这样的RPC。

> 从历史上说，门是为Spring分布式操作系统开发的，具体细节可以从http://www.sun.com/tech/projects/spring上得到。[Hamilton and Kougiouris 1993]中有关于该操作系统中门IPC机制的一个说明。
>
> 后来门添加到了Solaris 2.5中，不过有关它的唯一手册页面中只含有一个警告，说门是仅由某些Sun应用程序使用的一个试验性接口。到了Solaris 2.6后，该接口的文档出现在8个手册页面中，不过这些手册页面把该接口的稳定性列为“进展中（evolving）”。预期我们在本章讲述的API可能会随Solaris将来版本的出现而发生变化。Linux上门接口的初级版本也在开发中，可访问http://www.cs.brown.edu/~tor/doors。[①]
>
> Solaris 2.6中门的实现涉及一个函数库（它含有我们将在本章中描述的door_*XXX*函数）和一个内核文件系统（/kernel/sys/doorfs），用户的应用程序需链接这个函数库（使用-ldoor命令行选项链接程序）。
>
> 尽管门是一个Solaris特有的特性，我们还是详细地讲述它，因为它们提供了很不错的远程过程调用的入门知识，又不必对付任何连网支持细节。我们还将在附录A中看到，与所有其他形式的消息传递机制相比较，门不是更快也至少一样快。

本地过程调用是同步的：调用者直到被调用过程返回后才重新获得控制。线程可认为是提供了某种形式的异步过程调用：有一个函数被调用（pthread_create的第三个参数），该函数和调用者看起来在同时执行。调用者可通过调用pthread_join等待这个新线程的完成。远程过程调用可能是同步的，也可能是异步的，不过我们将看到门调用是同步的。

在进程（客户或服务器）内部，门是用描述符标识的。在进程以外，门可能是用文件系统中的路径名标识的。一个服务器通过调用door_create创建一个门，传递给该函数的参数是将与该门关联的过程的一个指针，该函数的返回值是新创建的门的一个描述符。该服务器然后通过调用fattach给这个门描述符关联一个路径名。一个客户通过调用open来打开一个门，传递给该函数的参数是该门的服务器关联在其上的路径名，该函数的返回值是本客户访问该门的描述符。该客户然后通过调用door_call调用服务器过程。自然，某个门的服务器可以是另一个门的客户。

我们说过门调用是同步的：当客户调用door_call时，该函数直到服务器过程返回（或发生某个错误）时才返回。Solaris的门实现还跟线程相联系。每当有一个客户调用某个服务器过程时，服务器进程中的一个线程就处理该客户的调用。线程管理通常是由门函数库自动进行的，该函数库根据需要创建新的线程，然而我们将看到，如果需要服务器进程亲自管理这些线程的话，它应该怎么去做。这还意味着一个给定的服务器可以同时为多个客户调用同一个服务器过程提供服务，每个客户一个线程。这是一个并发（concurrent）服务器。既然一个给定服务器过程可能有多个实例在同时执行（每个实例作为一个线程），因此服务器过程必须是线程安全的。

调用一个服务器过程时，可以同时从客户向服务器传递数据和描述符。也可以同时从服务器向客户传递回数据和描述符。描述符传递对于门来说是内在的。而且，既然门是用描述符标识的，因此描述符传递允许一个进程把一个门传递给另外某个进程。我们将在15.8节详细讨论描述符传递。

① Linux上门接口实现的URL已挪到了http://www.rampant.org/doors/中。

例子

我们以一个简单的例子开始关于门的讨论：客户向服务器传递一个长整数，服务器以长整数结果返回该值的平方。图15-2给出了客户程序。（我们在本例子中掩盖了许多细节，它们在本章以后再讨论。）

doors/client1.c

```
 1 #include    "unpipc.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     fd;
 6     long    ival, oval;
 7     door_arg_t  arg;
 8     if (argc != 3)
 9         err_quit("usage: client1 <server-pathname> <integer-value>");
10     fd = Open(argv[1], O_RDWR);    /* open the door */
11         /* set up the arguments and pointer to result */
12     ival = atol(argv[2]);
13     arg.data_ptr = (char *) &ival;      /* data arguments */
14     arg.data_size = sizeof(long);       /* size of data arguments */
15     arg.desc_ptr = NULL;
16     arg.desc_num = 0;
17     arg.rbuf = (char *) &oval;     /* data results */
18     arg.rsize = sizeof(long);      /* size of data results */
19         /* call server procedure and print result */
20     Door_call(fd, &arg);
21     printf("result: %ld\n", oval);
22     exit(0);
23 }
```

doors/client1.c

图15-2　向服务器发送一个长整数以求取其平方值的客户

打开门

8~10　调用open打开由命令行上的路径名指定的门。所返回的描述符称为门描述符（door descriptor），不过我们有时候就称它为门（door）。

设置参数和指向结果的指针

11~18　arg参数含有一个指向参数的指针和一个指向结果的指针。data_ptr指向参数的第一个字节，data_size指定参数的字节数。desc_ptr和desc_num处理描述符的传递，我们将在15.8节中讲述。rbuf指向结果缓冲区的第一个字节，rsize是它的大小。

调用服务器过程并输出结果

19~21　通过调用door_call来调用服务器过程，作为参数指定的是所打开的门描述符和指向所设置参数结构的指针。调用返回后输出结果。

图15-3给出了服务器程序。它由一个名为servproc的服务器过程和一个main函数构成。

服务器过程

2~10　调用服务器过程需有5个参数，但是我们真正使用的是dataptr，它指向参数的第一个字节。通过该指针取出长整数参数后求它的平方。然后控制随结果由door_return传递回客户。该函数的第一个参数是指向结果的指针，第二个参数是结果的大小，其余两个参数处理描述符的返回。

doors/server1.c

```
1 #include     "unpipc.h"
2 void
3 servproc(void *cookie, char *dataptr, size_t datasize,
4           door_desc_t *descptr, size_t ndesc)
5 {
6       long    arg, result;
7       arg = *((long *) dataptr);
8       result = arg * arg;
9       Door_return((char *) &result, sizeof(result), NULL, 0);
10 }
11 int
12 main(int argc, char **argv)
13 {
14      int     fd;
15      if (argc != 2)
16          err_quit("usage: server1 <server-pathname>");
17          /* create a door descriptor and attach to pathname */
18      fd = Door_create(servproc, NULL, 0);
19      unlink(argv[1]);
20      Close(Open(argv[1], O_CREAT | O_RDWR, FILE_MODE));
21      Fattach(fd, argv[1]);
22          /* servproc() handles all client requests */
23      for ( ; ; )
24          pause();
25 }
```

doors/server1.c

图15-3 返回一个长整数的平方值的服务器程序

创建一个门描述符并附接到路径名

17~21 调用door_create创建一个门描述符。该函数的第一个参数是指向与该门关联的服务器函数（servproc）的指针。取得门描述符后，必须将它与文件系统中的一个路径名关联，因为该路径名是客户标识这个门的手段。这种关联通过在文件系统中创建一个普通文件（我们首先调用unlink以防该文件已存在，同时忽略它的任何出错返回）后调用fattach完成，其中fattach是把一个描述符与一个路径名相关联的SVR4函数。

主服务器线程不干活

22~24 主服务器线程然后阻塞在一个pause调用中。所有工作全由servproc函数去做，每次有一个客户请求到达时，该函数就作为服务器进程中的另一个线程来执行。

为运行这个客户和服务器程序，我们首先在一个窗口中启动服务器：

```
solaris % server1 /tmp/server1
```

然后在另一个窗口中启动客户，所指定的路径名参数与我们传递给服务器的相同：

```
solaris % client1 /tmp/server1 9
result: 81
solaris % ls -l /tmp/server1
Drw-r--r--   1 rstevens other1         0 Apr  9 10:09 /tmp/server1
```

结果与预期的一致，当执行ls时，我们看到其输出中第一个字符为D，表明路径名/tmp/server1是某个门的路径名。

图15-4展示了本例子表现的行为过程。它看起来是door_call调用了服务器过程，服务器过程然后返回。

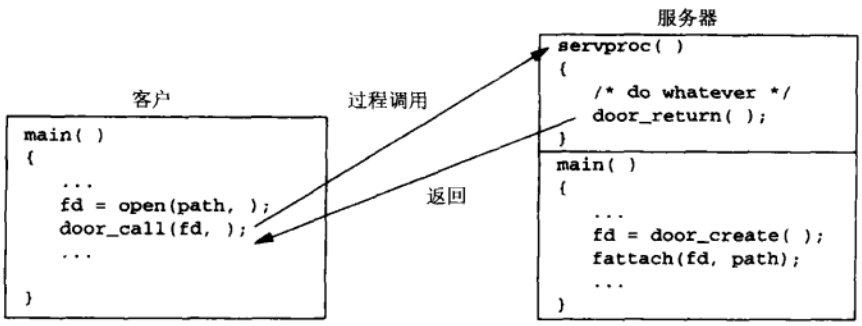
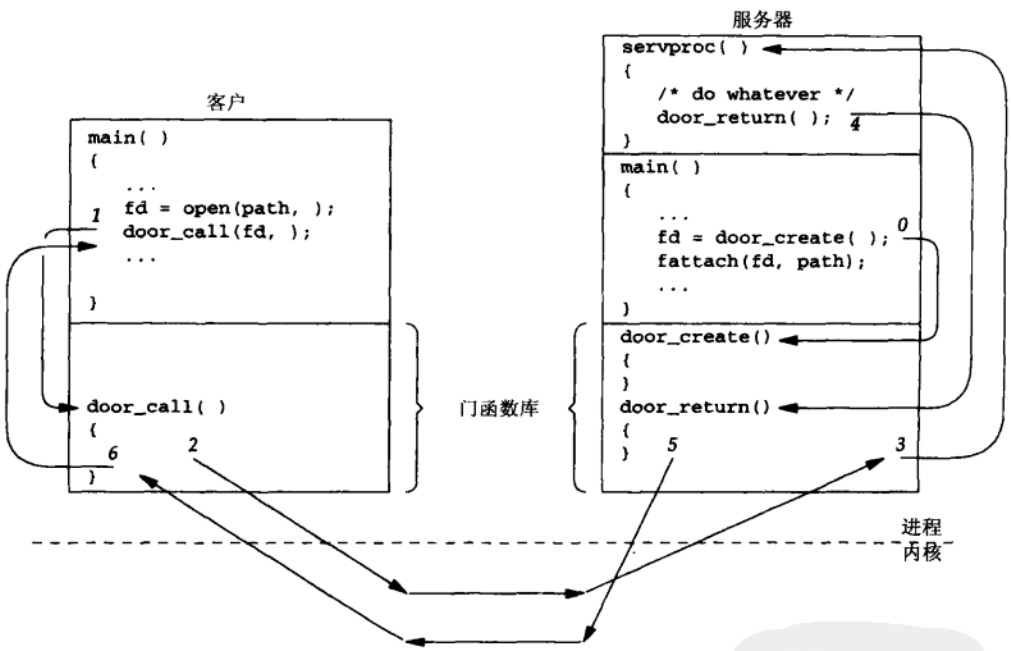


图15-4　从一个进程到另一个进程的表象过程调用

图15-5展示了调用同一台主机上另一个进程中的某个过程时真正发生的行为。



360

图15-5　从一个进程到另一个进程的过程调用的真正控制流

图15-5中发生了以下几个编了号的步骤。

(0) 服务器进程首先启动，它调用door_create创建一个指代函数servproc的门描述符，然后把该描述符附接到文件系统中的某个路径名。

(1) 客户进程启动，它调用door_call。这实际上是门函数库中的一个函数。

(2) door_call库函数执行一个进入内核的系统调用。标识出目标过程后，控制被传递到目标进程中的某个门库函数。

(3) 真正的服务器过程（本例子中名为servproc）被调用。

(4) 服务器过程执行处理客户请求所需的任意操作，执行完后调用door_return。

(5) door_return实际上是门函数库中的一个函数，它执行一个进入内核的系统调用。

(6) 标识出客户后把控制传递回该客户。

本章其余各节更为详尽地讲述门API，同时查看许多例子。在附录A中我们将看到，就延迟而言，门提供了最快的IPC形式。

15.2 door_call 函数

door_call函数由客户调用，它会调用在服务器进程的地址空间中执行的一个服务器过程。

```
#include <door.h>

int door_call(int fd, door_arg_t *argp);
```

返回：若成功则为0，若出错则为-1

其中描述符*fd*通常是由open返回的（例如图15-2）。由客户打开并将所返回的描述符作为第一个参数传递给door_call的路径名标识了该函数所调用的服务器过程。

第二个参数*argp*指向一个结构，该结构描述了调用参数和用于容纳返回值的缓冲区。 361

```
typedef struct door_arg {
  char          *data_ptr;      /* call: ptr to data arguments;
                                   return: ptr to data results */
  size_t         data_size;     /* call: #bytes of data arguments;
                                   return: actual #bytes of data results */
  door_desc_t   *desc_ptr;      /* call: ptr to descriptor arguments;
                                   return: ptr to descriptor results */
  size_t         desc_num;      /* call: number of descriptor arguments;
                                   return: number of descriptor results */
  char          *rbuf;          /* ptr to result buffer */
  size_t         rsize;         /* #bytes of result buffer */
} door_arg_t;
```

返回时也由该结构描述返回值。该结构的所有6个成员在返回时都可能有变化，我们马上就讲到。

> 给其中的两个指针成员使用char*数据类型有些奇怪，为避免出现编译警告，我们不得不在代码中显式地对它们进行类型强制转换。我们倒希望它们是void*数据类型的指针。door_return的第一个参数也同样使用了char*。Solaris 2.7也许会把desc_num的数据类型改为unsigned int，door_return的最后一个参数也将相应地作修改。

无论是参数还是结果都存在两种数据类型：数据和描述符。

- 数据参数（data arguments）是由data_ptr指向的一系列总共data_size个字节。客户和服务器必须以某种方式“知道”这些参数（以及结果）的格式。举例来说，无法通过特殊的编码告诉服务器参数的数据类型。在图15-2和图15-3中，客户程序和服务器程序编写成知道参数是一个长整数，结果也是一个长整数。封装数据类型信息的办法之一是（为了若干年后有人仍能读懂代码）：把所有的参数放进一个结构中，把所有的结果放进另一个结构中，把这两个结构都定义在一个头文件中，客户程序和服务器程序都包括进这个头文件。我们将随图15-11和图15-12给出一个这样的例子。如果没有数据参数，我们就把data_ptr指定成一个空指针，把data_size指定为0。

由于客户和服务器处理的是封装到一个参数缓冲区和一个结果缓冲区中的二进制参数和结果，因而隐含着客户程序和服务器程序必须以同样的编译器编译的要求。有时候在同样的系统上，不同的编译器也会以不同的方式封装结构。

- 描述符参数（descriptor arguments）是一个door_desc_t结构的数组，每个元素含有一个从客户往服务器过程传递的描述符。所传递的door_desc_t结构数为desc_num。（我们将在15.8节描述这个结构以及“传递描述符”的含义。）如果没有描述符参数，我们就把desc_ptr指定成一个空指针，把desc_num指定为0。
- 返回时data_ptr指向数据结果（data results），data_size则指定这些结果的大小。如果没有数据结果，data_size就为0，这时我们应该忽略data_ptr。
- 返回时也可能有描述符结果（descriptor results）：desc_ptr指向一个door_desc_t结构的数组，每个元素含有一个由服务器过程传递回客户的描述符。所返回的door_desc_t结构数存放在desc_num中。如果没有描述符结果，desc_num就为0，这时我们应该忽略desc_ptr。

给参数和结果使用同样的缓冲区是可行的。这就是说调用door_call时，data_ptr和desc_ptr都可指向由rbuf指定的缓冲区中。

在调用door_call前，客户把rbuf设置成指向存放结果的缓冲区，把rsize设置成该缓冲区的大小。返回时data_ptr和desc_ptr通常都指向这个结果缓冲区中。如果该缓冲区太小而容纳不了服务器的结果，门函数库就会在调用者的地址空间中使用mmap（12.2节）自动分配一个新的缓冲区，然后相应地更新rbuf和rsize。data_ptr和desc_ptr将指向这个新分配的缓冲区中。留意rbuf的变化并在以后某个时刻以rbuf和rsize为参数调用munmap把门函数库分配的缓冲区返还给系统，这些是调用者的责任。我们将随图15-7给出这样的一个例子。

15.3 door_create 函数

服务器进程通过调用door_create建立一个服务器过程。

```
#include <door.h>

typedef void Door_server_proc(void *cookie, char *dataptr, size_t datasize,
                              door_desc_t *descptr, size_t ndesc);

int door_create(Door_server_proc *proc, void *cookie, u_int attr);
```

返回：若成功则为非负描述符，若出错则为-1

在上述的声明中，我们加上了自己的typedef，这样简化了服务器过程的函数原型。这个typedef语句说，门服务器过程（例如图15-3中的servproc）是以5个参数调用的，不返回任何值。

当一个服务器调用door_create时，传递给该函数的第一个参数*proc*是一个服务器过程的地址，该服务器过程将与由该函数返回的门描述符相关联。当调用这个服务器过程时，它的第一个参数*cookie*是作为door_create的第二个参数传递进去的值。这么一来给服务器提供了向服务器过程传递某个指针的一种方式，该指针在每次有一个客户调用该过程时传递。服务器过程的以后4个参数（*dataptr*、*datasize*、*descptr*、*ndesc*）描述来自服务器的数据参数和描述符参数，也就是上一节中描述的door_arg_t结构的前4个成员所描述的信息。

door_create的最后一个参数*attr*描述所创建服务器过程的特殊属性，它或为0，或为以下两个常值的按位或。

362 ~ 363

DOOR_PRIVATE　随着客户请求的到达，门函数库在服务器进程中自动创建必要的新线程以调用服务器过程。按照默认设置，这些线程放在进程范围的线程池中，可用于给服务器进程中的任何门提供客户请求的服务。
指定DOOR_PRIVATE属性就是告诉门函数库该门得有自己的服务器线程池，与进程范围的池分开。

DOOR_UNREF　当指代该门的描述符数从2降为1时，将第二个参数（*dataptr*）指定为DOOR_UNREF_DATA再次调用该服务器过程。这次调用的*descptr*参数是一个空指针，datasize和ndesc则均为0。我们将从图15-16的讨论开始给出这个属性的一些例子。

服务器过程的返回值声明为void，因为它们从不通过调用return或掉出函数尾返回。它们会调用我们将在下一节描述的door_return。

我们从图15-3中已看到，调用door_create获取一个门描述符之后，服务器通常调用fattach给该描述符关联以一个文件系统中的路径名。客户open该路径名就获得了调用door_call所需的门描述符。

> fattach不是一个Posix.1函数，但是Unix 98却需要它。另外，有一个名为fdetach的函数撤销这种关联，名为fdetach的命令则简单地激活该同名函数。

由door_create创建的门描述符在其文件描述符标志中设置了FD_CLOEXEC位。这意味着创建一个门描述符的进程如果调用了任意一个exec函数，该描述符就被内核关闭。至于fork，尽管父进程中已打开的所有描述符随后为子进程所共享，却只有父进程会收到来自客户的门激活请求，这些请求不会递交给子进程，即使由door_create返回的描述符在子进程中也是打开的。

> 如果我们把一个门考虑成是由一个进程ID和待调用的相应服务器过程的地址标识的（我们将从在15.6节介绍的door_info_t结构中看到这两个标识信息），那么关于fork和exec的这两条规则就有意义了。子进程永远得不到门激活请求是因为与该门关联的进程ID是调用door_create的父进程的进程ID。遇到exec调用时门描述符必须关闭的原因则在于，即使进程ID没有改变，与该门关联的服务器过程的地址在exec之后新激活的程序中已失去意义。

15.4 door_return 函数

服务器过程完成工作时通过调用door_return返回。这会使客户中相关联的door_call调用返回。

364

```
#include <door.h>
int door_return(char *dataptr, size_t datasize, door_desc_t *descptr, size_t *ndesc);
                                                    返回：若成功则不返回到调用者，若出错则为-1
```

数据结果由*dataptr*和*datasize*指定，描述符结果由*descptr*和*ndesc*指定。

15.5 door_cred函数

门有一个很不错的特性：服务器过程能够获取每个调用对应的客户凭证。这是由door_cred函数完成的。

```
#include <door.h>

int door_cred(door_cred_t *cred);
```

返回：若成功则为0，若出错则为-1

其中由*cred*指向的door_cred_t结构将在返回时填入客户的凭证。

```
typedef struct door_cred {
  uid_tdc_euid;     /* effective user ID of client */
  gid_tdc_egid;     /* effective group ID of client */
  uid_tdc_ruid;     /* real user ID of client */
  gid_tdc_rgid;     /* real group ID of client */
  pid_tdc_pid;      /* process ID of client */
} door _cred_t;
```

APUE的4.4节讨论了有效ID和实际ID之间的差别，我们将随图15-8给出体现这种差异的一个例子。

注意本函数中没有描述符参数。本函数返回发起当前门激活请求的客户的有关信息，因而必须在服务器过程或由该过程调用的某个函数中调用它。

15.6 door_info函数

我们刚才描述的door_cred函数给服务器提供关于客户的信息。客户也可通过调用door_info函数找出有关服务器的信息。

```
#include <door.h>

int door_info(int fd, door_info_t *info);
```

365

返回：若成功则为0，若出错则为-1

其中*fd*指定一个已打开的门。由*info*指向的door_info_t结构将在返回时填入关于服务器的信息。

```
typedef struct door_info {
  pid_t         di_target;      /* server process ID */
  door_ptr_t    di_proc;        /* server procedure */
  door_ptr_t    di_data;        /* cookie for server procedure */
  door_attr_t   di_attributes;  /* attributes associated with door */
  door_id_t     di_uniquifier;  /* unique number */
} door_info_t;
```

其中di_target是服务器的进程ID，di_proc是所指定的服务器过程在服务器进程中的地址（对于客户来说，这个地址信息也许没什么用）。作为第一个参数传递给该服务器过程的cookie指针由di_data返回。

该门的当前属性存放在di_attributes中，我们已在15.3节中描述过两个属性：DOOR_PRIVATE和DOOR_UNREF。两个新属性是DOOR_LOCAL（该过程局部于本进程）和DOOR_REVOKE（服务器已通过调用door_revoke函数撤销了与该门关联的过程）。

每个门刚被创建时都被赋予一个系统范围内唯一的数值，它作为di_uniquifier返回。

本函数通常由客户调用，以获取关于服务器的信息。然而它也可以由一个服务器过程调用，这时第一个参数指定为DOOR_QUERY，所返回的信息是关于调用线程的。这种情形下，服务器过程的地址（di_proc）和cookie（di_data）也许有用。

15.7 例子

现在给出使用之前所述五个函数的一些例子。

15.7.1 door_info 函数

图15-6给出的程序打开一个门，调用door_info，然后输出关于该门的信息。

doors/doorinfo.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     fd;
 6     struct stat stat;
 7     struct door_infoinfo;

 8     if (argc != 2)
 9         err_quit("usage: doorinfo <pathname>");

10     fd = Open(argv[1], O_RDONLY);
11     Fstat(fd, &stat);
12     if (S_ISDOOR(stat.st_mode) == 0)
13         err_quit("pathname is not a door");

14     Door_info(fd, &info);
15     printf("server PID = %ld, uniquifier = %ld",
16            (long) info.di_target, (long) info.di_uniquifier);
17     if (info.di_attributes & DOOR_LOCAL)
18         printf(", DOOR_LOCAL");
19     if (info.di_attributes & DOOR_PRIVATE)
20         printf(", DOOR_PRIVATE");
21     if (info.di_attributes & DOOR_REVOKED)
22         printf(", DOOR_REVOKED");
23     if (info.di_attributes & DOOR_UNREF)
24         printf(", DOOR_UNREF");
25     printf("\n");

26     exit(0);
27 }
```

doors/doorinfo.c

图15-6 输出关于一个门的信息

我们open所指定的路径名并首先验证它是一个门。对应一个门的stat结构的st_mode成员含有一个值，该值使得S_ISDOOR宏求值为真。我们接着调用door_info。

我们首先指定一个不是门的路径名运行该程序，然后针对Solaris 2.6所用的两个门运行它。

```
solaris % doorinfo /etc/passwd
pathname is not a door

solaris % doorinfo /etc/.name_service_door
server PID = 308, uniquifier = 18, DOOR_UNREF
```

366

```
solaris % doorinfo /etc/.syslog_door
server PID = 282, uniquifier = 1635

solaris % ps -f -p 308
    root   308     1  0   Apr 01 ?        0:34 /usr/sbin/nscd
solaris % ps -f -p 282
    root   282     1  0   Apr 01 ?        0:10 /usr/sbin/syslogd -n -z 14
```

我们使用ps命令来查看以由door_info返回的进程ID运行的是什么程序。

15.7.2　结果缓冲区大小

在描述door_call函数时我们提到过，如果结果缓冲区太小而容纳不了服务器的结果，门函数库就会自动分配一个新的缓冲区。我们现在给出展示这一特性的一个例子。图15-7给出了新的客户程序，它是对图15-2的简单修改。

doors/client2.c

```
 1 #include     "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     fd;
 6     long    ival, oval;
 7     door_arg_t  arg;

 8     if (argc != 3)
 9         err_quit("usage: client2 <server-pathname> <integer-value>");

10     fd = Open(argv[1], O_RDWR);    /* open the door */

11         /* set up the arguments and pointer to result */
12     ival = atol(argv[2]);
13     arg.data_ptr = (char *) &ival;/* data arguments */
14     arg.data_size = sizeof(long); /* size of data arguments */
15     arg.desc_ptr = NULL;
16     arg.desc_num = 0;
17     arg.rbuf = (char *) &oval;     /* data results */
18     arg.rsize = sizeof(long);      /* size of data results */

19         /* call server procedure and print result */
20     Door_call(fd, &arg);
21     printf("&oval = %p, data_ptr = %p, rbuf = %p, rsize  = %d\n",
22            &oval, arg.data_ptr, arg.rbuf, arg.rsize);
23     printf("result: %ld\n", *((long *) arg.data_ptr));

24     exit(0);
25 }
```

doors/client2.c

图15-7　输出结果的地址

19~23　在这个版本的程序中，我们输出oval变量的地址、data_ptr的内容（它指向从door_call返回的结果）以及结果缓冲区的地址和大小（rbuf和rsize）。

这时运行该程序，我们还没有修改来自图15-2的结果缓冲区的大小，因此预期data_ptr和rbuf都指向oval变量，rsize则为4字节。确实，这跟我们实际看到的一致：

```
solaris % client2 /tmp/server2 22
&oval = effff740, data_ptr =effff740, rbuf = effff740, rsize = 4
result: 484
```

现在只改动图15-7中一行，把客户的结果缓冲区大小减少1字节。新版本中将图15-7的第18

行改为如下：

```
arg.rsize = sizeof(long) - 1;    /* size of data results */
```

执行这个新的客户程序，我们看到分配了一个新缓冲区，data_ptr就指向这个新缓冲区。

```
solaris % client3 /tmp/server3 33
&oval = effff740, data_ptr =ef620000, rbuf = ef620000, rsize = 4096
result: 1089
```

所分配缓冲区的大小4096是该系统上的页面大小，我们已在12.6节中看到过这个大小。从这个例子可以看出，我们应该总是通过data_ptr指针访问服务器的结果，而不能通过其地址在rbuf中传递给服务器的变量。这就是说在我们的例子中，我们应该以*((long*)arg.data_ptr)访问长整数结果，而不是以oval来访问（图15-2中是这么做的）。

367 ~ 368

通过调用mmap分配的这个新缓冲区可使用munmap返还给系统。客户也可以给后续的door_call调用一直使用该缓冲区。

15.7.3 door_cred 函数和客户凭证

这一次我们对图15-3中的servproc函数做了修改：调用door_cred函数来获取客户的凭证。图15-8给出了这个新的服务器过程，客户和服务器main函数不变，仍然是图15-2和图15-3。

doors/server4.c

```
1 #include     "unpipc.h"

2 void
3 servproc(void *cookie, char *dataptr, size_t datasize,
4          door_desc_t *descptr, size_t ndesc)
5 {
6       long    arg, result;
7       door_cred_t info;

8           /* obtain and print client credentials */
9       Door_cred(&info);
10      printf("euid = %ld, ruid = %ld, pid = %ld\n",
11             (long) info.dc_euid, (long) info.dc_ruid, (long) info.dc_pid);

12      arg = *((long *) dataptr);
13      result = arg * arg;
14      Door_return((char *) &result, sizeof(result), NULL, 0);
15 }
```

doors/server4.c

图15-8 获取并输出客户凭证的服务器过程

首先运行客户程序，正如所料，我们将看到有效用户ID等于实际用户ID。接着变换为超级用户，把客户程序可执行文件的属主改为root，并启用该文件的SUID位，然后再次运行客户程序。①

```
solaris % client4 /tmp/server4 77          第一次运行客户程序
result: 5929
```

① SUID是可执行文件的一个权限位。设置了该位的可执行文件运行时，相应进程的有效用户ID就设置成该文件的属主（本例子中就是root），从而可达到获取原本没有的特权之目的。典型的SUID程序是修改口令用的passwd（路径名通常为/bin/passwd或/usr/bin/passwd，注意与/etc/passwd区分开），该文件的属主为root。当普通用户执行passwd程序时，实际上暂时取得了超级用户的特权，能够修改原本只能读的/etc/passwd文件，不过程序代码控制着具体的操作，使得有恶意的用户不能胡作非为。——译者注

```
solaris % su                                    变为超级用户
Password:
Sun Microsystems Inc.    SunOS 5.6        Generic August 1997
solaris # cd 含有客户程序可执行文件的目录
solaris # ls -l client4
-rwxrwxr-x   1 rstevens other1    139328 Apr 13 06:02 client4
solaris # chown root client4                把属主改为root
solaris # chmod u+s client4                 并打开SUID位
solaris # ls -l client4                     检查文件权限与属主
-rwsrwxr-x   1 root     other1    139328 Apr 13 06:02 client4
solaris # exit
solaris % ls -l client4
-rwsrwxr-x   1 root     other1    139328 Apr 13 06:02 client4
solaris % client4 /tmp/server4 77           然后再次运行客户程序
result: 5929
```

369

查看服务器的输出，我们看到第二次运行客户程序时，有效用户ID发生了变化。

```
solaris % server4 /tmp/server4
euid = 224, ruid = 224, pid = 3168
euid = 0, ruid = 224, pid =3176
```

其中有效用户ID为0表示超级用户。

15.7.4　服务器的自动线程管理

为了查看由服务器执行的线程管理，我们让服务器过程在一开始执行时输出自己所在线程的线程ID，然后让它睡眠5秒钟，以此模拟一个运行时间较长的服务器过程。这段睡眠使我们可以在已有一个客户正在接受服务期间启动多个客户。图15-9给出了新的服务器过程。

doors/server5.c

```
 1 #include    "unpipc.h"

 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4           door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;

 7     arg = *((long *) dataptr);
 8     printf("thread id %ld, arg = %ld\n", pr_thread_id(NULL), arg);
 9     sleep(5);

10     result = arg * arg;
11     Door_return((char *) &result, sizeof(result), NULL, 0);
12 }
```

doors/server5.c

图15-9　输出线程ID后睡眠的服务器过程

其中引入了来自我们自己的函数库的一个新函数pr_thread_id。它需要一个参数（指向一个线程ID的指针或表示使用调用线程之线程ID的空指针），返回的是该线程的一个long类型的整数标识符（往往是一个小整数）。一个进程总是可以由一个整数值来标识，这个整数就是它的进程ID。即使我们不清楚进程ID是int类型还是long类型，也只需把getpid的返回值类型强制转换成long类型后输出其值（图9-2）。但是线程的标识符是一个pthread_t数据类型的值（称为线程ID），它不必是一个整数。事实上，Solaris 2.6使用小整数作为线程ID，Digital Unix则使用指针。然而我们往往希望只给线程输出一个小整数标识符（本例子就是这样），以用于调试目的。图15-10给出的pr_thread_id库函数可处理这个问题。

370

lib/wrappthread.c

```
245 long
246 pr_thread_id(pthread_t *ptr)
247 {
248 #if defined(sun)
249     return((ptr == NULL) ? pthread_self() : *ptr);    /* Solaris */
250 #elif defined(__osf__) && defined(__alpha)
251     pthread_t   tid;
252     tid = (ptr == NULL) ? pthread_self() : *ptr;      /* Digital Unix */
253     return(pthread_getsequence_np(tid));
254 #else
255         /* everything else */
256     return((ptr == NULL) ? pthread_self() : *ptr);
257 #endif
258 }
```

lib/wrappthread.c

图15-10 pr_thread_id函数：给调用线程返回小整数标识符

如果实现没有给线程提供小整数标识符，这个函数就可能复杂得多，需要把pthread_t值映射成小整数，并记住这种映射关系（存放在一个数组或链表中），供以后的调用使用。[Lewis and Berg 1998] 中的thread_name函数完成了这个工作。

返回图15-9，我们接连运行客户程序三次。由于我们在启动下一个客户之前等待shell提示符，因此知道每次在服务器端的5秒钟等待都是完整的。

```
solaris % client5 /tmp/server5 55
result: 3025
solaris % client5 /tmp/server5 66
result: 4356
solaris % client5 /tmp/server5 77
result: 5929
```

查看服务器的输出，我们看到同一个服务器线程为每个客户提供服务：

```
solaris % server5 /tmp/server5
thread id 4, arg = 55
thread id 4, arg = 66
thread id 4, arg = 77
```

现在同时启动三个客户：

```
solaris % client5 /tmp/server5 11 & client5 /tmp/server5 22 & \
client5 /tmp/server5 33 &
[2]         3812
[3]         3813
[4]         3814
solaris% result: 484
result: 121
result: 1089
```

服务器的输出表明，服务器进程创建了两个新线程来处理同一个服务器过程的第二次和第三次激活请求。

```
thread id 4, arg = 22
thread id 5, arg = 11
thread id 6, arg = 33
```

接着同时启动另外两个客户：

```
solaris % client5 /tmp/server5 11 & client5 /tmp/server5 22 &
[2]         3830
```

```
[3]        3831
solaris% result: 484
result: 121
```

我们看到服务器使用的是以前创建的线程：

```
thread id 6, arg = 22
hread id 5, arg = 11
```

从本例子可以看出，服务器进程（实际上是跟我们的服务器代码相链接的门函数库）根据需要自动创建服务器线程。如果一个应用程序希望亲自处理线程的管理，那么它可以使用我们将在15.9节中描述的函数去这么做。

我们还验证了服务器过程是一个并发（concurrent）服务器：同一个服务器过程同时可有多个实例在运行，它们作为彼此独立的线程给不同的客户提供服务。认定服务器并发的第一个办法是，当我们同时运行三个客户时，所有三个结果都在5秒后输出。要是服务器是迭代的（iterative），那么第一个结果在所有三个客户都启动后过5秒输出，下一个结果再过5秒输出，最后一个结果又过5秒后才输出。

15.7.5　服务器的自动线程管理：多个服务器过程

前一个例子的服务器进程中只有一个服务器过程。我们的下一个问题是，同一服务器进程中的多个服务器过程是否可以使用同一个线程池。为测试这一点，我们给服务器进程增加了另一个服务器过程，同时重新编写了前一个例子的代码，以表现出在不同进程间处理参数和结果的一种更好的风格。

我们的第一个文件是名为squareproc.h的头文件，它给我们的求平方函数定义了一个输入参数的数据类型和一个输出参数的数据类型。它还给该过程定义了路径名。图15-11给出了该文件。

doors/squareproc.h

```
1 #define PATH_SQUARE_DOOR       "/tmp/squareproc_door"

2 typedef struct {                    /* input to squareproc() */
3     long    arg1;
4 } squareproc_in_t;

5 typedef struct {                    /* output from squareproc() */
6     long    res1;
7 } squareproc_out_t;
```

doors/squareproc.h

图15-11　squareproc.h头件

我们的新服务器过程接受一个长整数输入值，返回一个double类型的值，该值是输入值的平方根。我们在sqrtproc.h头文件中定义了该过程的路径名、输入结构和输出结构，图15-12给出了该文件。

doors/sqrtproc.h

```
1 #define PATH_SQRT_DOOR   "/tmp/sqrtproc_door"

2 typedef struct {                /* input to sqrtproc() */
3     long    arg1;
4 } sqrtproc_in_t;

5 typedef struct {                /* output from sqrtproc() */
6     double  res1;
7 } sqrtproc_out_t;
```

doors/sqrtproc.h

图15-12　sqrtproc.h头文件

我们的客户程序在图15-13中给出。它只是一先一后调用那两个服务器过程，然后输出结果。该程序与本章中已给出的其他客户程序类似。

doors/client7.c

```
 1 #include    "unpipc.h"
 2 #include    "squareproc.h"
 3 #include    "sqrtproc.h"

 4 int
 5 main(int argc, char **argv)
 6 {
 7     int     fdsquare, fdsqrt;
 8     door_arg_t  arg;
 9     squareproc_in_t square_in;
10     squareproc_out_t square_out;
11     sqrtproc_in_t   sqrt_in;
12     sqrtproc_out_t  sqrt_out;

13     if (argc != 2)
14         err_quit("usage: client7 <integer-value>");

15     fdsquare = Open(PATH_SQUARE_DOOR, O_RDWR);
16     fdsqrt = Open(PATH_SQRT_DOOR, O_RDWR);

17         /* set up the arguments and call squareproc() */
18     square_in.arg1 = atol(argv[1]);
19     arg.data_ptr = (char *) &square_in;
20     arg.data_size = sizeof(square_in);
21     arg.desc_ptr = NULL;
22     arg.desc_num = 0;
23     arg.rbuf = (char *) &square_out;
24     arg.rsize = sizeof(square_out);
25     Door_call(fdsquare, &arg);

26         /* set up the arguments and call sqrtproc() */
27     sqrt_in.arg1 = atol(argv[1]);
28     arg.data_ptr = (char *) &sqrt_in;
29     arg.data_size = sizeof(sqrt_in);
30     arg.desc_ptr = NULL;
31     arg.desc_num = 0;
32     arg.rbuf = (char *) &sqrt_out;
33     arg.rsize = sizeof(sqrt_out);
34     Door_call(fdsqrt, &arg);

35     printf("result: %ld %g\n", square_out.res1, sqrt_out.res1);

36     exit(0);
37 }
```

doors/client7.c

图15-13 调用求平方过程和求平方根过程的客户程序

图15-14给出了我们的两个服务器过程。每个过程输出其所在线程的线程ID和输入参数，睡眠5秒钟，计算结果，然后返回。

图15-15给出的服务器程序main函数打开两个门描述符，然后给每个门描述符关联一个服务器过程。

doors/server7.c

```
 1 #include    "unpipc.h"
 2 #include    <math.h>
 3 #include    "squareproc.h"
 4 #include    "sqrtproc.h"
 5 void
 6 squareproc(void *cookie, char *dataptr, size_t datasize,
 7           door_desc_t *descptr, size_t ndesc)
 8 {
 9     squareproc_in_t in;
10     squareproc_out_tout;
11     memcpy(&in, dataptr, min(sizeof(in), datasize));
12     printf("squareproc: thread id %ld, arg = %ld\n",
13            pr_thread_id(NULL), in.arg1);
14     sleep(5);
15     out.res1 = in.arg1 * in.arg1;
16     Door_return((char *) &out, sizeof(out), NULL, 0);
17 }
18 void
19 sqrtproc(void *cookie, char *dataptr, size_t datasize,
20          door_desc_t *descptr, size_t ndesc)
21 {
22     sqrtproc_in_t   in;
23     sqrtproc_out_t  out;
24     memcpy(&in, dataptr, min(sizeof(in), datasize));
25     printf("sqrtproc: thread id %ld, arg = %ld\n",
26            pr_thread_id(NULL), in.arg1);
27     sleep(5);
28     out.res1 = sqrt((double) in.arg1);
29     Door_return((char *) &out, sizeof(out), NULL, 0);
30 }
```

doors/server7.c

图15-14　两个服务器过程

doors/server7.c

```
31 int
32 main(int argc, char **argv)
33 {
34     int     fd;
35     if (argc != 1)
36         err_quit("usage: server7");
37     fd = Door_create(squareproc, NULL, 0);
38     unlink(PATH_SQUARE_DOOR);
39     Close(Open(PATH_SQUARE_DOOR, O_CREAT | O_RDWR, FILE_MODE));
40     Fattach(fd, PATH_SQUARE_DOOR);
41     fd = Door_create(sqrtproc, NULL, 0);
42     unlink(PATH_SQRT_DOOR);
43     Close(Open(PATH_SQRT_DOOR, O_CREAT | O_RDWR, FILE_MODE));
44     Fattach(fd, PATH_SQRT_DOOR);
45     for ( ; ; )
46         pause();
47 }
```

doors/server7.c

图15-15　服务器程序main函数

我们运行客户程序，它需花10秒钟才输出结果（正如我们的预料）。

```
solaris % client7 77
result: 5929 8.77496
```

查看服务器的输出，我们看到服务器进程中同一线程处理了该客户的先后两个请求。

```
solaris % server7
squareproc: thread id 4, arg = 77
sqrtproc: thread id 4, arg = 77
```

这个例子告诉我们，对于一个给定进程，其服务器线程池中的任意线程都能够处理针对任意服务器过程的客户请求。

15.7.6 服务器的 DOOR_UNREF 属性

我们在15.3节中提到过，DOOR_UNREF可作为一个新创建的门的属性之一指定给door_create函数。该函数的手册页面中说，当指代某个具备该属性的门的描述符数降为1（即该门的引用计数从2变为1）时，该门的服务器过程将有一次特殊的激活。特殊之处在于，传递给该服务器过程的第二个参数（指向数据参数的指针）是常值DOOR_UNREF_DATA。下面列出了引用该门的三种方法。

(1) 服务器中由door_create返回的描述符算作一个引用。事实上，激活某个不再引用过程的触发条件是其引用计数从2变为1而不是从1变为0的原因在于，服务器进程通常在其整个存活期内一直保持该描述符打开。

(2) 附接到该门上的文件系统中的路径名也算作一个引用。我们可以删除这个引用，办法有：调用fdetach函数，运行fdetach程序，或者从文件系统中删除该路径名（既可调用unlink函数，也可运行rm命令）。

(3) 客户中由open返回的描述符算作一个打开的引用，直到该描述符关闭为止，这种关闭既可以显式地调用close完成，也可以隐式地由客户进程的终止完成。本章中已给出的所有客户进程都是隐式地关闭门描述符的。

我们的第一个例子展示，如果服务器在调用fattch之后关闭所创建的门描述符，那么其服务器过程的不再引用激活（unreferenced invocation）将立即发生。图15-16给出了我们的服务器过程和服务器main函数。

7~10 服务器过程认出特殊的不再引用激活后输出一个消息。当前线程通过以两个空指针和两个值为0的大小调用door_return从这个特殊调用中返回。

28 fattach返回后close所创建的门描述符。fattach之后该描述符对于服务器的唯一用途是供调用door_bind、door_info或door_revoke之用。

启动该服务器，我们注意到不再引用激活立即发生：

```
solaris % serverunref1 /tmp/door1
door unreferenced
```

追踪该门的引用计数变化情况，它在door_create返回后变为1，在fattach返回后变为2。服务器调用close使它从2变为1，于是触发了不再引用激活。该门现在所剩的唯一引用是它在文件系统中的路径名，它是客户指代该门所需的手段。这就是说，客户仍能正常工作。

```
solaris % clientunref1 /tmp/door1 11
result: 121
solaris % clientunref1 /tmp/door1 22
result: 484
```

doors/serverunref1.c

```
 1 #include    "unpipc.h"
 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4          door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;
 7     if (dataptr == DOOR_UNREF_DATA) {
 8         printf("door unreferenced\n");
 9         Door_return(NULL, 0, NULL, 0);
10     }
11     arg = *((long *) dataptr);
12     printf("thread id %ld, arg = %ld\n", pr_thread_id(NULL), arg);
13     sleep(6);
14     result = arg * arg;
15     Door_return((char *) &result, sizeof(result), NULL, 0);
16 }
17 int
18 main(int argc, char **argv)
19 {
20     int     fd;
21     if (argc != 2)
22         err_quit("usage: serverunref1 <server-pathname>");
23         /* create a door descriptor and attach to pathname */
24     fd = Door_create(servproc, NULL, DOOR_UNREF);
25     unlink(argv[1]);
26     Close(Open(argv[1], O_CREAT | O_RDWR, FILE_MODE));
27     Fattach(fd, argv[1]);
28     Close(fd);
29         /* servproc() handles all client requests */
30     for ( ; ; )
31         pause();
32 }
```

doors/serverunref1.c

图15-16　处理不再引用激活的服务器过程

而且，该服务器过程没有进一步的不再引用激活发生。事实上对于一个给定门，只会递交一次不再引用激活。

现在把我们的服务器程序改回平常的情形，也就是并不close所创建的门描述符。图15-17给出了服务器过程和服务器main函数。我们设置了6秒钟的睡眠，并在服务器过程返回之前输出信息。在一个窗口中启动服务器，在另一个窗口中验证服务器所创建的门的关联路径名存在于文件系统中，然后用rm命令删除该路径名：

```
solaris % ls -l /tmp/door2
Drw-r--r--   1 rstevens other1         0 Apr 16 08:58 /tmp/door2
solaris % rm /tmp/door2
```

我们一删除该路径名，相应服务器过程的不再引用激活就发生：

```
solaris % serverunref2 /tmp/door2
door unreferenced                          一旦从系统中删除该路径名
```

追踪该门的引用计数变化情况，它在door_create返回后变为1，在fattach返回后变为2。

当我们rm该门的路径名时，这个命令把它的引用计数从2降为1，于是触发了不再引用激活。

在下面的有关该属性的最后一个例子中，我们仍然从文件系统中删除路径名，不过是在启动门的三次客户激活之后。我们要展示的是，每次客户激活给引用计数加1，只有所有三个客户都终止后，不再引用激活才发生。我们使用先前的在图15-17中给出的服务器程序，客户程序没有变动，如图15-2所示。

doors/serverunref2.c

```
 1 #include    "unpipc.h"
 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4         door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;
 7     if (dataptr == DOOR_UNREF_DATA) {
 8         printf("door unreferenced\n");
 9         Door_return(NULL, 0, NULL, 0);
10     }
11     arg = *((long *) dataptr);
12     printf("thread id %ld, arg = %ld\n", pr_thread_id(NULL), arg);
13     sleep(6);
14     result = arg * arg;
15     printf("thread id %ld returning\n", pr_thread_id(NULL));
16     Door_return((char *) &result, sizeof(result), NULL, 0);
17 }
18 int
19 main(int argc, char **argv)
20 {
21     int  fd;
22     if (argc != 2)
23         err_quit("usage: serverunref2 <server-pathname>");
24         /* create a door descriptor and attach to pathname */
25     fd = Door_create(servproc, NULL, DOOR_UNREF);
26     unlink(argv[1]);
27     Close(Open(argv[1], O_CREAT | O_RDWR, FILE_MODE));
28     Fattach(fd, argv[1]);
29         /* servproc() handles all client requests */
30     for ( ; ; )
31         pause();
32 }
```

doors/serverunref2.c

图15-17 不关闭所创建门描述符的服务器程序

```
solaris % clientunref2 /tmp/door2 44 & clientunref2 /tmp/door2 55 & \
clientunref2 /tmp/door2 66 &
[2]     13552
[3]     13553
[4]     13554
solaris % rm /tmp/door2                    在三个客户运行期间
solaris % result: 1936
result: 3025
result: 4356
```

下面是服务器的输出：

```
solaris % serverunref2 /tmp/door2
thread id 4, arg = 44
thread id 5, arg = 55
thread id 6, arg = 66
thread id 4 returning
thread id 5 returning
thread id 6 returning
door unreferenced
```

追踪该门的引用计数变化情况，它在door_create返回后变为1，在fattach返回后变为2。当每个客户调用open时，引用计数逐次加1，从2变为3，从3变为4，最后从4变为5。当我们rm该门的路径名时，引用计数从5变为4。然后随着每个客户的终止，引用计数器从4变为3，从3变为2，最后从2变为1，最后这次减1触发了不再引用激活。

以上这些例子表明，尽管DOOR_UNREF属性的说明很简单（“当引用计数从2变为1时，不再引用激活发生”），但是要使用这一特性，必须首先理解引用计数。

15.8　描述符传递

当考虑把一个打开着的描述符从一个进程传递到另一个进程时，我们通常想到以下两点：

- 调用fork之后，子进程与父进程共享所有打开着的描述符；
- 调用exec之后，所有描述符通常仍保持打开。

前一个例子中，父进程打开一个描述符，调用fork派生出子进程，然后父进程关闭该描述符，由子进程来处理它。这样就把一个打开着的描述符从父进程传递给了子进程。

现今的Unix系统扩充了这种描述符传递概念，提供了把任何打开着的描述符从一个进程传递给任何其他进程的能力，这两个进程间有无亲缘关系皆可。门提供了从客户到服务器以及从服务器到客户的一个描述符传递API。

> 我们在UNPv1的14.7节[①]讲述过使用Unix域套接字的描述符传递。源自Berkely的内核使用这些套接字传递描述符，TCPv3第18章中提供了全部细节。SVR4内核使用另一种技术来传递描述符，即I_SENDFD和I_RECVFD这两个ioctl命令，APUE的15.5.1节讲述了它们。但是SVR4进程仍可以使用Unix域套接字来访问这一内核特性。

注意理解传递描述符的含义。图4-7中，服务器打开文件后把整个文件内容复制到底部的管道中。如果该文件的大小为1MB，那么通过底部的管道从服务器往客户流动了1MB的数据。然
379 而如果是服务器往客户传递回一个描述符的话，通过图4-7中底部的管道传递的仅仅是这个描述符（我们假设它是某些特定于内核的小量信息），而不是文件本身。客户取得该描述符后读出文件内容，并把它写往标准输出。所有的文件读出操作都发生在客户中，服务器只是打开该文件。

注意，服务器不能只是通过图4-7中底部的管道写出描述符号，就像以下代码那样：

```
int    fd;

fd = Open( ... );
Write(pipefd, &fd, sizeof(int));
```

这种方法并不奏效。描述符号是特定于进程的属性。假设fd的值在服务器中为4。即使该描述符在客户中是打开的，也几乎可以肯定它指代的文件不同于服务器进程中描述符4所指代的文件（从一个进程到另一个进程描述符号意义不变的唯一时刻是穿越fork前后和穿越exec前

① 此处为UNPv1第2版英文原版书节号，第3版为15.7节。——编者注

后)。如果服务器中的最低未用描述符为4，那么服务器中一个成功的open将返回4。如果服务器把描述符4“传递”给客户，而客户中的最低未用描述符为7，那么我们希望客户中的描述符7与服务器中的描述符4指代同一个文件。APUE的图15-4和TCPv3的图18-4展示了从内核角度看来必然发生的情况：这两个描述符（这儿的例子是服务器中的4和客户中的7）必须同时指向内核中的同一文件表项。描述符传递涉及一定的内核神秘技巧，然而像门和Unix域套接字这样的API隐藏了这些内部细节，从而允许进程们很容易从一个进程向另一个进程传递描述符。

通过一个门从客户向服务器传递描述符的手段是，把door_arg_t结构的desc_ptr成员设置成指向一个door_desc_t结构的数组，door_num成员则设置成这些door_desc_t结构的数目。从服务器向客户传递回描述符的手段是，把door_return的第三个参数设置成指向一个door_desc_t结构的数组，把该函数的第四个参数设置成待传递描述符的数目。

```
typedef struct door_desc {
  door_attr_t  d_attributes;          /* tag for union */
  union {
    struct {                          /* valid if tag = DOOR_DESCRIPTOR */
      int          d_descriptor;      /* descriptor number */
      door_id_t    d_id;              /* unique id */
    } d_desc;
  } d_data;
} door_desc_t;
```

该结构含有一个联合，其第一个成员是一个标识该联合中含有哪个成员的标记。不过该联合当前只定义了一个成员（描述一个描述符的d_desc结构），标记（d_attributes）于是必须设置为DOOR_DESCRIPTOR。 380

例子

我们修改以前的文件服务器例子（回想图1-9），使服务器打开文件并把打开着的描述符传递给客户，然后由客户把文件的内容复制到标准输出。图15-18展示了这样的编排。

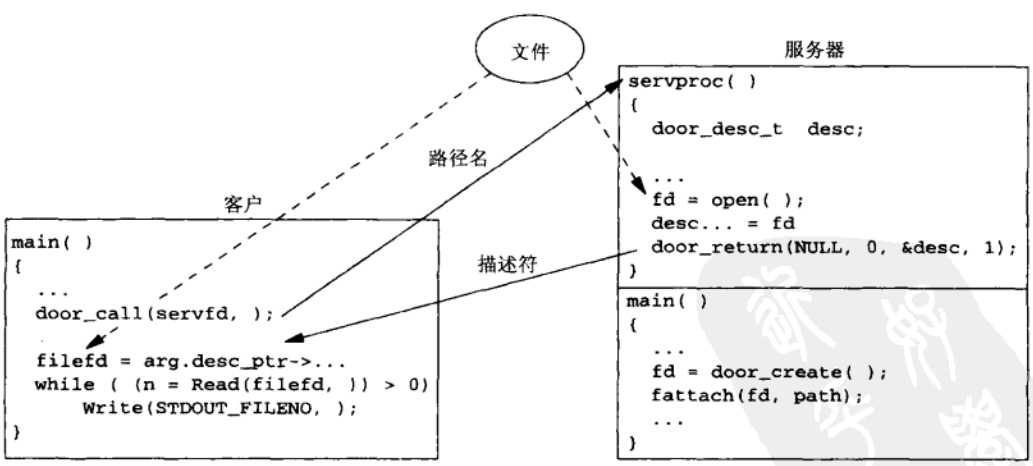


图15-18 服务器传递回打开着描述符的文件服务器例子

图15-19给出了客户程序。

doors/clientfd1.c

```
1 #include    "unpipc.h"

2 int
3 main(int argc, char **argv)
4 {
5     int     door, fd;
6     char    argbuf[BUFFSIZE], resbuf[BUFFSIZE], buff[BUFFSIZE];
7     size_t  len, n;
8     door_arg_t  arg;

9     if (argc != 2)
10        err_quit("usage: clientfd1 <server-pathname>");

11    door = Open(argv[1], O_RDWR);      /* open the door */

12    Fgets(argbuf, BUFFSIZE, stdin);         /* read pathname of file to open */
13    len = strlen(argbuf);
14    if (argbuf[len-1] == '\n')
15        len--;                  /* delete newline from fgets() */

16        /* set up the arguments and pointer to result */
17    arg.data_ptr = argbuf;      /* data argument */
18    arg.data_size = len + 1;    /* size of data argument */
19    arg.desc_ptr = NULL;
20    arg.desc_num = 0;
21    arg.rbuf = resbuf;          /* data results */
22    arg.rsize = BUFFSIZE;       /* size of data results */

23    Door_call(door, &arg);      /* call server procedure */

24    if (arg.data_size != 0)
25        err_quit("%.*s", arg.data_size, arg.data_ptr);
26    else if (arg.desc_ptr == NULL)
27        err_quit("desc_ptr is NULL");
28    else if (arg.desc_num != 1)
29        err_quit("desc_num = %d", arg.desc_num);
30    else if (arg.desc_ptr->d_attributes != DOOR_DESCRIPTOR)
31        err_quit("d_attributes = %d", arg.desc_ptr->d_attributes);

32    fd = arg.desc_ptr->d_data.d_desc.d_descriptor;
33    while ( (n = Read(fd, buff, BUFFSIZE)) > 0)
34        Write(STDOUT_FILENO, buff, n);

35    exit(0);
36 }
```

doors/clientfd1.c

图15-19　描述符传递文件服务器例子的客户程序

打开门，从标准输入读入路径名

9~15　与待打开的门关联的路径名是一个命令行参数。打开该门后，从标准输入读入客户希望打开的文件名，并删掉结尾的换行符。

设置参数和指向结果的指针

16~22　设置door_arg_t结构。我们给路径名的大小多加了1，以允许服务器用空字符终止该路径名。

调用服务器过程并检查结果

23~31　调用服务器过程，然后检查其结果是我们预期的：没有数据，有一个描述符。我们不久将看到，服务器只在打不开客户指定的文件时才返回数据（含有一个出错消息），这种情况下，我们调用err_quit输出这个错误。

获取描述符，把文件复制到标准输出

32~34 从door_desc_t结构取得描述符，然后把相应的文件复制到标准输出。

图15-20给出了服务器过程。服务器main函数没有变化，如图15-3所示。

doors/serverfd1.c

```
 1 #include    "unpipc.h"

 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4         door_desc_t *descptr, size_t ndesc)
 5 {
 6     int     fd;
 7     char    resbuf[BUFFSIZE];
 8     door_desc_t desc;

 9     dataptr[datasize-1] = 0;       /* null terminate */
10     if ( (fd = open(dataptr, O_RDONLY)) == -1) {
11             /* error: must tell client */
12         snprintf(resbuf, BUFFSIZE, "%s: can't open, %s",
13                 dataptr, strerror(errno));
14         Door_return(resbuf, strlen(resbuf), NULL, 0);

15     } else {
16             /* open succeeded: return descriptor */
17         desc.d_data.d_desc.d_descriptor = fd;
18         desc.d_attributes = DOOR_DESCRIPTOR;
19         Door_return(NULL, 0, &desc, 1);
20     }
21 }
```

doors/serverfd1.c

图15-20 打开一个文件并传递回其描述符的服务器过程

为客户打开文件

9~14 用空字符终止由客户提供的路径名，然后尝试open该文件。如果发生错误，数据结果就是含有相应出错消息的字符串。 381~382

成功

15~20 如果open成功，那就返回所得到的描述符，这时没有数据结果。

我们启动服务器，指定其待创建的门的路径名为/tmp/fd1，然后运行客户程序：

```
solaris % clientfd1 /tmp/fd1
/etc/shadow
/etc/shadow: can't open, Permission denied
solaris % clientfd1 /tmp/fd1
/no/such/file
/no/such/file: can't open, No such file or directory
solaris % clientfd1 /tmp/df1
/etc/ntp.conf                          一个只有2行文本的文件
multicastclient 224.0.1.1
driftfile /etc/ntp.drift
```

前面两次我们指定了导致出错返回的路径名，最后一次服务器返回了一个只含2行文本的文件的描述符。

通过门来传递描述符存在一个问题。在我们的例子中要看到这个问题，只需在服务器过程中成功的open调用之后加一个printf语句。你将看到每次打开的描述符值比上一个描述符值大1。问题在于服务器把这些描述符传递给客户后没有关闭它们。然而没有简单的办法做到这一点。从逻辑上讲，执行close的合适位置是在door_return返回之后，因为那时所打 383

开的描述符已发送给客户，然而door_return并不返回!要是我们通过一个Unix域套接字使用sendmsg来传递该描述符，或者通过一个SVR4管道使用ioctl来传递它，那么可以在sendmsg或ioctl返回后close它。然而门的描述符传递规范不同于这两种技术，因为从传递描述符的函数上不会有返回发生。绕过这个问题的唯一办法是，让服务器过程以某种方式记着它已打开了一个描述符，并在以后某个时候关闭它，这么一来程序会变得非常杂乱。

这个问题到Solaris 2.7中应得到纠正，办法是增加一个新的DOOR_RELEASE属性。发送者把d_attributes设置成DOOR_DESCRIPTOR | DOOR_RELEASE，这样告诉系统在把相应的描述符传递给接收者之后要将其关闭。

15.9 door_sever_create 函数

我们随图15-9展示出，当客户请求到达时，门函数库会按照处理它们的需要自动地创建新线程。这些线程由该函数库作为脱离的线程创建，具有默认的线程栈大小，禁止了线程取消功能，并具有从调用door_create的线程初始继承来的信号掩码和调度类。如果我们想要改变上述任何特性，或者希望亲自管理服务器线程池，那么可以调用door_server_create以指定我们自己的服务器创建过程（sever creation procedure）。

```
#include <door.h>

typedef void Door_create_proc(door_info_t *);

Door_create_proc *door_server_create(Door_create_proc *proc);

                                        返回：指向先前的服务器创建过程的指针
```

跟15.3节中door_create的声明一样，我们使用C的typedef语句来简化库函数的函数原型。我们的新数据类型把服务器创建过程定义为接受单个参数（指向一个door_info_t结构的指针），不返回任何值（void）。当我们调用door_server_create时，其参数是指向我们的服务器创建过程的指针，返回值是指向上一个服务器创建过程的指针。

每当需要一个新线程来给某个客户请求提供服务时，我们的服务器创建过程就被调用。至于哪个服务器过程需要这个新线程的信息则存放在其地址作为参数传递进本创建过程的door_info_t结构中。该结构的di_proc成员含有这个服务器过程的地址，di_data成员则含有该服务器过程每次被调用时所传递进去的cookie指针。

查看所发生的活动的最简单办法是使用例子。我们的客户程序没有变化，如图15-2所示。我们的服务器程序中除原来的服务器过程函数和main函数外，增加了两个新函数。图15-21给出服务器进程中四个函数的关系概貌，当时一些函数已注册而且所有函数都已调用。

384 图15-22给出了服务器程序main函数。

与图15-3相比，有四个变动：（1）去掉了门描述符fd的声明（它现在是一个我们将在图15-23中给出并描述的全局变量）；（2）以一个互斥锁保护door_create调用（也在图15-23中说明）；（3）在创建门之前调用door_server_create，将其参数指定为我们的服务器创建过程（my_create，它将在下面给出）；（4）door_create调用中，最后一个参数（属性）现在是DOOR_PRIVATE而不是0。DOOR_PRIVATE告诉门函数库本门将有它自己的称为私用服务器池（private server pool）的线程池。

使用DOOR_PRIVATE指定一个私用服务器池和使用door_server_create指定一个服务器创建过程是互相独立的。总共有以下四种可能情形。

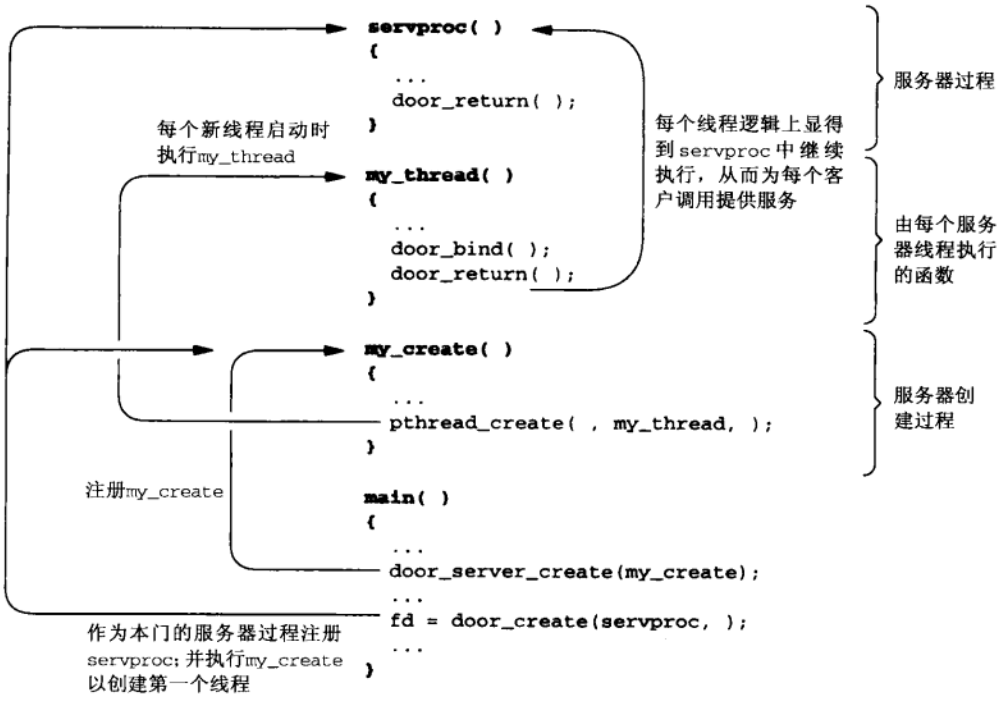


图15-21 我们的服务器进程中四个函数的概貌

doors/server6.c

```
42 int
43 main(int argc, char **argv)
44 {
45     if (argc != 2)
46         err_quit("usage: server6 <server-pathname>");
47     Door_server_create(my_create);
48         /* create a door descriptor and attach to pathname */
49     Pthread_mutex_lock(&fdlock);
50     fd = Door_create(servproc, NULL, DOOR_PRIVATE);
51     Pthread_mutex_unlock(&fdlock);
52     unlink(argv[1]);
53     Close(Open(argv[1], O_CREAT | O_RDWR, FILE_MODE));
54     Fattach(fd, argv[1]);
55         /* servproc() handles all client requests */
56     for ( ; ; )
57         pause();
58 }
```

doors/server6.c

图15-22 线程池管理例子程序的main函数

(1) 默认情形：没有私用服务器池，也没有服务器创建过程。系统根据需要创建线程，它们都进入进程范围的线程池。

(2) 指定DOOR_PRIVATER，不过没有服务器创建过程。系统根据需要创建线程，对于创建时没有指定DOOR_PRIVATE属性的门，新创建的线程将进入进程范围的池，对于创建时指定了

DOOR_PRIVATE属性的门，新创建的线程将进入该门的私用服务器池。

(3) 没有私用服务器，但是指定了一个服务器创建过程。每当需要一个新线程时，该服务器创建过程就被调用，所创建的线程都进入进程范围的线程池。

(4) 指定DOOR_PRIVATE，同时指定了一个服务器创建过程。每当需要一个新线程时，该服务器创建过程就被调用。一个线程创建出来后，必须调用door_bind把自己赋给合适的私用服务器池，否则它将被赋给进程范围的线程池。

图15-23给出了我们的两个新函数：my_create和my_thread。my_create是我们的服务器创建过程，它把my_thread指定为它所创建的每个线程都要执行的函数。

doors/server6.c

```
13 pthread_mutex_t fdlock = PTHREAD_MUTEX_INITIALIZER;
14 static int  fd = -1;            /* door descriptor */

15 void *
16 my_thread(void *arg)
17 {
18     int     oldstate;
19     door_info_t *iptr = arg;

20     if ((Door_server_proc *) iptr->di_proc == servproc) {
21         Pthread_mutex_lock(&fdlock);
22         Pthread_mutex_unlock(&fdlock);

23         Pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
24         Door_bind(fd);
25         Door_return(NULL, 0, NULL, 0);
26     } else
27         err_quit("my_thread: unknown function: %p", "(Door_Server_proc*)iptr->di_proc");
28     return(NULL);             /* never executed */
29 }

30 void
31 my_create(door_info_t *iptr)
32 {
33     pthread_t tid;
34     pthread_attr_t attr;

35     Pthread_attr_init(&attr);
36     Pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
37     Pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
38     Pthread_create(&tid, &attr, my_thread, (void *) iptr);
39     Pthread_attr_destroy(&attr);
40     printf("my_create: created server thread %ld\n", pr_thread_id(&tid));
41 }
```

doors/server6.c

图15-23　我们自己的线程管理函数

服务器创建过程

30~41　每当my_create被调用时，我们就创建一个新线程。但是在调用pthread_create前，我们先初始化该线程的属性，设置其竞用范围（contention scope）为PTHREAD_SCOPE_SYSTEM，将它指定为脱离的线程。接着调用pthread_create创建该线程，它启动执行的是my_thread函数。服务器创建过程以及新线程启动函数的参数都是指向待激活之门的door_info_t结构的指针。如果有一个带多个门的服务器，而且指定了一个服务器创建过程，该服务器创建过程就在其中任何一个门需要一个新线程时被调用。该服务器创建过程以及由它指定给pthread_create的线程启动函数区分这些不同的服

务器过程的唯一办法是：查看该door_info_t结构中的di_proc指针。

把竞用范围设置成PTHREAD_SCOPE_SYSTEM意味着本线程将跟其他进程中的线程竞争处理器资源的使用。与之相对的PTHREAD_SCOPE_PROCESS意味着本线程只跟本进程内的其他线程竞争处理器资源的使用。后者对于门不起作用，因为门函数库要求执行door_return的内核轻权进程跟引发这个激活请求的轻权进程相同。未绑定的线程（PTHREAD_SCOPE_PROCESS）可能在执行服务器过程期间改变轻权进程。①

要求作为脱离的线程来创建新线程是为了防止系统在该线程终止时保存有关它的任何信息，因为不会有其他线程对它调用pthread_join。

线程启动函数

15~20 my_thread是通过调用pthread_create指定的线程启动函数。传递给它的参数是早先传递进my_create函数的指向待激活之门的door_info_t结构的指针。本进程中唯一存在的服务器过程是servproc，因此我们只是验证该参数引用了这个过程。 385~387

等待描述符变为有效

21~22 服务器创建过程在door_create首次被调用时调用，目的是为了创建一个初始的服务器线程。门函数库中该调用是在door_create返回前发出的。然而变量fd要到door_create返回后才含有有效的门描述符。（这是一个鸡与蛋的问题。）既然知道my_thread是作为独立于调用door_create的主线程的另一个线程运行的，我们解决这个定时问题的办法于是就是按下述方式使用互斥锁fdlock：主线程在调用door_create前给该互斥锁上锁，在door_create返回并往fd中存入一个值（图15-22）后给该互斥锁解锁。我们的my_thread函数先是给该互斥锁上锁（也许要阻塞到主线程解开该互斥锁为止），然后给它解锁。我们也许可以增设一个由主线程向它发送信号的条件变量，不过这儿没有这个必要，因为我们知道将发生的调用的顺序。

禁止线程取消功能

23 当使用pthread_create创建一个新的Posix线程时，线程取消功能默认是启用的。这种情况下，当某个客户中止一个进展中的door_call调用时（我们将在图15-31中展示这个操作），线程取消处理程序（如果有的话）将被调用，相应的服务器过程所在线程随后终止。在取消功能禁止的情况下（如这儿将做的那样），当某个客户中止一个进展

① 这儿涉及有关线程的不少概念。首先，我们必须区分并发（concurrency）和并行（parallelism）。一个多处理机应用程序运行时的并行度是实际达到的并行执行程度，因此受限于其进程可用的物理处理器数。该应用程序的并发度则是在处理器数无限的理想前提下所能达到的最大并行度。其次，并发既可在系统级提供，也可在用户级提供。内核通过认知一个进程内的多个线程（也称为热线程）并独立地调度它们来提供系统级并发。内核然后把这些线程复用到可用的处理器上。用户级并发则由应用程序通过用户级线程库提供。内核不认得这样的用户线程（也称为冷线程），因而必须由用户级线程库管理和调度。许多系统（包括提供门API的Solaris 2.6）实现了把两者结合起来的双并发模型：内核认得一个进程内的多个线程，线程库则支持不为内核所见的用户线程。再次，Solaris 2.x支持内核线程（kernelthread）、轻权进程（lightweight process）和用户线程（userthread）。内核线程是能够被独立地调度和派遣到某个系统处理器上运行的基本轻权对象。它不必与一个用户进程相关联，是由内核根据内部需要创建、运行或毁除以执行指定的功能的。内核线程对于应用程序不可见。轻权进程是内核支持的用户可见线程，属于热线程。它基于内核线程，实际上每个轻权进程绑定在各自的内核线程上。用户线程是由线程库实现的内核不可见的更高级对象，属于冷线程。这是用户直接看到的线程。最后，一个进程内有两类用户线程：一类是绑定在某个轻权进程上的线程，一类是共享公共的轻权进程池的未绑定线程。竞用范围为PTHREAD_SCOPE_SYSTEM的线程属于绑定的线程，为PTHREAD_SCOPE_PROCESS的线程属于未绑定的线程。——译者注

中的door_call调用时，相应的服务器过程仍然完成（其所在的线程未被终止），不过来自door_return的结果被简单地丢弃。既然取消功能启用时服务器线程有可能被终止，而且服务器过程当时可能处于为其客户执行的某个操作中（它可能持有某些锁或信号量），因此门函数库（默认）禁止了由它创建的所有线程的线程取消功能。如果一个服务器过程希望在某个客户过早终止时被取消，那么它所在的线程必须启用取消功能，并准备好处理这种事件。

> 注意，PTHREAD_SCOPE_SYSTEM竞用范围和脱离状态时在创建线程时作为属性指定的。然而取消模式只能由当事线程本身在开始运行后设置。事实上，即使我们禁止了取消功能，线程仍能在任何时候随心所欲地启用和禁止它。

把本线程捆绑到一个门

24　调用door_bind把调用线程捆绑到与某个门关联的私用服务器池，其中该门的描述符为该函数的参数。既然该调用需要这个门描述符，于是我们让fd成为这个版本的服务器程序的一个全局变量。

使得本线程对于客户调用可用

25　通过以两个空指针和两个0长度为参数调用door_return，本线程使其自身对于外来的门激活请求可用。

图15-24给出了服务器过程。它与图15-9中的版本相同。

doors/server6.c

```
 1 #include    "unpipc.h"

 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4         door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;

 7     arg = *((long *) dataptr);
 8     printf("thread id %ld, arg = %ld\n", pr_thread_id(NULL), arg);
 9     sleep(5);

10     result = arg * arg;
11     Door_return((char *) &result, sizeof(result), NULL, 0);
12 }
```

doors/server6.c

图15-24　服务器过程

为展示所发生的情况，我们首先启动服务器：

```
solaris % server6 /tmp/door6
my_create: created server thread 4
```

服务器启动后一调用door_create，我们的服务器创建过程就被首次调用，尽管当时我们还没有启动客户。这就创建了第一个线程，它将等待第一个客户调用请求。我们然后接连运行客户程序三次。

```
solaris % client6 /tmp/door6 11
result: 121
solaris % client6 /tmp/door6 22
result: 484
solaris % client6 /tmp/door6 33
result: 1089
```

查看服务器的相应输出，我们看到第一个调用发生时创建了另一个线程（其线程ID为5），然后ID为4的线程给所有三个客户请求提供了服务。门函数库看来总是保留一个额外的线程备用。

```
my_create: created server thread 5
thread id 4, arg = 11
thread id 4, arg = 22
thread id 4, arg = 33
```

接着在后台几乎同时执行客户程序三次：

```
solaris % client6 /tmp/door6 44 & client6 /tmp/door6 55 & \
client6 /tmp/door6 66 &
[2]         4919
[3]         4920
[4]         4921
solaris % result: 1936
result: 4356
result: 3025
```

查看服务器的相应输出，我们看到创建了两个新线程（线程ID分别为6和7），给三个客户请求提供服务的分别是线程4、5和6。

388 ~ 389

```
thread id 4, arg = 44
my_create: created server thread 6
thread id 5, arg = 66
my_create: created server thread 7
thread id 6, arg = 55
```

15.10 door_bind、door_unbind和door_revoke函数

加上以下三个额外函数后，门API就完整了。

```
#include <door.h>

int door_bind(int fd);

int door_unbind(void);

int door_revoke(int fd);
```

均返回：若成功则为0，若出错则为-1

我们在图15-23中引入了door_bind函数。它把调用线程捆绑到与描述符为*fd*的门关联的私用服务器池中。如果调用线程已绑定在另外某个门上，那就执行一个隐式的松绑操作。

door_unbind显式地把调用线程从其已绑定的门上松绑。

door_revoke撤销对于由*fd*标识的门的访问。一个门描述符只能由创建它的进程撤销。调用该函数时已在进展中的任何门激活实例仍允许正常地完成。

15.11 客户或服务器的过早终止

到此为止的所有例子都假设客户和服务器上都没有异常之事发生。我们现在考虑客户和服务器任何一方出错时发生什么情况。我们知道，当客户和服务器处于同一个进程中时（图15-1中的本地过程调用），客户不必担心服务器的崩溃，反之亦然，因为如果任何一方崩溃，那么整个进程崩溃。然而当客户和服务器分散到两个进程上时，我们必须考虑任何一方崩溃时会发生

什么情况，以及如何通知对方这种失败。客户和服务器无论是在同一台主机上还是在不同的主机上，我们都得考虑这些问题。

15.11.1　服务器的过早终止

390 客户阻塞在door_call调用中等待结果期间，必须知道服务器线程是否因某种原因而终止了。为了观察所发生的情况，我们让服务器过程调用pthread_exit以终止所在的线程。这样仅仅终止该线程本身，而不是终止整个服务器进程。图15-25给出了服务器过程。

doors/serverintr1.c

```
 1 #include    "unpipc.h"
 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4           door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;
 7     pthread_exit(NULL);          /* and see what happens at client */
 8     arg = *((long *) dataptr);
 9     result = arg * arg;
10     Door_return((char *) &result, sizeof(result), NULL, 0);
11 }
```

doors/serverintr1.c

图15-25　被激活后终止其自身的服务器过程

服务器程序的其余部分没有变化，如图15-3所示，客户程序没有变化，如图15-2所示。

运行客户程序，我们看到如果服务器过程在返回前终止了，那么客户的door_call将返回一个EINTR错误。

```
solaris % clientintr1 /tmp/door1 11
door_call error: Interrupted system call
```

15.11.2　door_call 系统调用的不可中断性

door_call的手册页面警告说，该函数不是一个可重新启动的系统调用。（门函数库中door_call函数激活一个同名的系统调用。）通过把服务器程序修改成其中的服务器过程在返回前睡眠6秒钟，我们就可以看到这种特性。图15-26给出了这个服务器过程。

doors/serverintr2.c

```
 1 #include    "unpipc.h"
 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4           door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;
 7     sleep(6);                    /* let client catch SIGCHLD */
 8     arg = *((long *) dataptr);
 9     result = arg * arg;
10     Door_return((char *) &result, sizeof(result), NULL, 0);
11 }
```

doors/serverintr2.c

图15-26　睡眠6秒钟的服务器过程

391 我们然后对图15-2给出的客户程序进行修改，即建立一个SIGCHLD信号处理程序，fork一

个子进程，让子进程睡眠2秒钟后终止。这么一来，客户父进程调用door_call后约2秒时，该父进程捕获SIGCHLD信号，接着其信号处理程序返回，从而中断door_call系统调用。图15-27给出了这个客户程序。

doors/clientintr2.c

```
 1 #include     "unpipc.h"

 2 void
 3 sig_chld(int signo)
 4 {
 5     return;                           /* just interrupt door_call() */
 6 }

 7 int
 8 main(int argc, char **argv)
 9 {
10     int     fd;
11     long    ival, oval;
12     door_arg_t  arg;

13     if (argc != 3)
14         err_quit("usage: clientintr2 <server-pathname> <integer-value>");

15     fd = Open(argv[1], O_RDWR);    /* open the door */

16         /* set up the arguments and pointer to result */
17     ival = atol(argv[2]);
18     arg.data_ptr = (char *) &ival;      /* data arguments */
19     arg.data_size = sizeof(long);       /* size of data arguments */
20     arg.desc_ptr = NULL;
21     arg.desc_num = 0;
22     arg.rbuf = (char *) &oval;      /* data results */
23     arg.rsize = sizeof(long);       /* size of data results */

24     Signal(SIGCHLD, sig_chld);
25     if (Fork() == 0) {
26         sleep(2);                   /* child */
27         exit(0);                    /* generates SIGCHLD */
28     }

29         /* parent: call server procedure and print result */
30     Door_call(fd, &arg);
31     printf("result: %ld\n", oval);

32     exit(0);
33 }
```

doors/clientintr2.c

图15-27 2秒钟后捕获SIGCHLD信号的客户程序

客户看到的错误就像服务器过程过早地终止一样，都是EINTR。

```
solaris % clientintr2 /tmp/door2 22
door_call error: Interrupted system call
```

这意味着我们必须阻塞调用door_call期间可能产生的任何信号，防止它们被递交给进程，因为这些信号会中断door_call。

392

15.11.3 等势过程与非等势过程

要是我们知道刚刚捕获了一个信号，当检测到由door_call返回的EINTR错误后接着再次调用同一个服务器过程，情况会怎么样呢？这么做时我们知道错误来自被捕获的信号，而不是

来自服务器过程的过早终止。然而这么做会导致我们将马上看到的问题。

首先把服务器过程修改为：（1）当被调用时输出当前线程ID；（2）睡眠6秒钟；（3）在返回之前输出当前线程ID。图15-28给出了这个版本的服务器过程。

doors/serverintr3.c

```
 1 #include    "unpipc.h"

 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4           door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;

 7     printf("thread id %ld called\n", pr_thread_id(NULL));
 8     sleep(6);                    /* let client catch SIGCHLD */
 9     arg = *((long *) dataptr);
10     result = arg * arg;
11     printf("thread id %ld returning\n", pr_thread_id(NULL));
12     Door_return((char *) &result, sizeof(result), NULL, 0);
13 }
```

doors/serverintr3.c

图15-28　当被调用时和准备返回时输出当前线程ID的服务器过程

图15-29给出了我们的客户程序。

2~8　声明全局变量caught_sigchld，它在SIGCHLD信号被捕获时由其信号处理程序设置为1。

31~42　只要返回的错误是EINTR，而且这是由我们的信号处理程序导致的，我们就在一个循环中调用door_call。

如果只看客户的输出，那么似乎没有问题：

```
solaris % clientintr3 /tmp/door3 33
calling door_call
calling door_call
result: 1089
```

第一次调用door_call后约2秒时，我们的信号处理程序激活，把caught_sigchld设置为1，该信号处理程序的返回导致第一次door_call调用返回EINTR错误，我们于是再次调用door_call。第二次调用时服务器过程运行到完毕，从而返回预期的结果。

但是查看服务器的输出，我们发现服务器过程调用了两次：

```
solaris % serverintr3 /tmp/door3
thread id 4 called
thread id 4 returning
thread id 5 called
393 thread id 5 returning
```

客户的第一次door_call调用被所捕获的信号中断后，它的第二次door_call调用启动了再次调用服务器过程的另一个线程。如果该服务器过程是等势的（idempotent），那是没有问题。但是如果该服务器过程是非等势的，那就有问题了。

等势（idempotent）一词在用于描述一个过程时，意思是该过程可调用任意多次而不出问题。我们那个计算平方值的服务器过程是等势的：不论调用一次还是两次，我们都得到正确的结果。另外一个等势过程的例子是返回当前时间和日期的过程。尽管该过程每次可能返回不同的信息（譬如说它被调用了两次，彼此相差1秒，于是导致返回时间也相差1秒），不过仍然是正确的。非等势过程的经典例子是从某个银行账户减去一笔费用的过程：除非该过程只调用了一次，否则最终结果是错误的。

doors/clientintr3.c

```
1 #include    "unpipc.h"
2 volatile sig_atomic_t caught_sigchld;
3 void
4 sig_chld(int signo)
5 {
6     caught_sigchld = 1;
7     return;                   /* just interrupt door_call() */
8 }
9 int
10 main(int argc, char **argv)
11 {
12     int     fd, rc;
13     long    ival, oval;
14     door_arg_t  arg;
15     if (argc != 3)
16         err_quit("usage: clientintr3 <server-pathname> <integer-value>");
17     fd = Open(argv[1], O_RDWR);    /* open the door */
18         /* set up the arguments and pointer to result */
19     ival = atol(argv[2]);
20     arg.data_ptr = (char *) &ival;      /* data arguments */
21     arg.data_size = sizeof(long);       /* size of data arguments */
22     arg.desc_ptr = NULL;
23     arg.desc_num = 0;
24     arg.rbuf = (char *) &oval;      /* data results */
25     arg.rsize = sizeof(long);       /* size of data results */
26     Signal(SIGCHLD, sig_chld);
27     if (Fork() == 0) {
28         sleep(2);                   /* child */
29         exit(0);                    /* generates SIGCHLD */
30     }
31         /* parent: call server procedure and print result */
32     for ( ; ; ) {
33         printf("calling door_call\n");
34         if ( (rc = door_call(fd, &arg)) == 0)
35             break;                  /* success */
36         if (errno == EINTR && caught_sigchld) {
37             caught_sigchld = 0;
38             continue;               /* call door_call() again */
39         }
40         err_sys("door_call error");
41     }
42     printf("result: %ld\n", oval);
43     exit(0);
44 }
```

doors/clientintr3.c

图15-29 接收到EINTR错误后再次调用door_call的客户程序

15.11.4 客户的过早终止

现在看一看客户在调用door_call之后但在服务器返回之前终止时，服务器过程是如何得到通知的。图15-30给出了我们的客户程序。

doors/clientintr4.c

```
 1 #include    "unpipc.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     fd;
 6     long    ival, oval;
 7     door_arg_t arg;

 8     if (argc != 3)
 9         err_quit("usage: clientintr4 <server-pathname> <integer-value>");

10     fd = Open(argv[1], O_RDWR);    /* open the door */

11         /* set up the arguments and pointer to result */
12     ival = atol(argv[2]);
13     arg.data_ptr = (char *) &ival;      /* data arguments */
14     arg.data_size = sizeof(long);       /* size of data arguments */
15     arg.desc_ptr = NULL;
16     arg.desc_num = 0;
17     arg.rbuf = (char *) &oval;      /* data results */
18     arg.rsize = sizeof(long);       /* size of data results */

19         /* call server procedure and print result */
20     alarm(3);
21     Door_call(fd, &arg);
22     printf("result: %ld\n", oval);

23     exit(0);
24 }
```

394 ~ 395

doors/clientintr4.c

图15-30　调用door_call后过早终止的客户程序

20　与图15-2相比唯一的变化是，在调用door_call之紧前调用alarm(3)。该函数调度了一个3秒后发出的SIGALAM信号，但是由于我们没有捕获这个信号，因此它的默认行为是终止客户进程。这将导致客户在door_call返回前终止，因为我们将在服务器过程中放置一个6秒钟的睡眠。

图15-31给出了我们的服务器过程以及它的线程取消处理程序。

回想8.5节中就线程取消功能的讨论以及随图15-23进行的相关讨论。当系统检测到客户在其door_call调用仍然进展期间即将终止时，就向处理该调用的服务器线程发送一个取消请求。

- 如果该服务器线程禁止了取消功能，那就什么事情都不发生，该线程继续执行到完毕（调用door_return之时），结果则被丢弃。
- 如果该服务器线程启用了取消功能，那就调用所设置的任何清理处理程序，该线程随后终止。

在图15-31给出的服务器过程中，我们首先调用pthread_setcancelstate以启用线程取消功能，因为门函数库创建新线程时禁止该功能。该函数还在变量oldstate中保存当前的取消状态，以便在本函数尾恢复状态。我们然后调用pthread_cleanup_push以把函数servproc_cleanup注册为取消处理程序。该函数只是输出本线程已被取消的消息，不过这儿正是其服务器过程在客户过早终止后做必要的清理工作（譬如说释放互斥锁、写一个日志文件记录，等等）的地方。当清理处理程序返回时，本线程即终止。

我们还在服务器过程中放置了一个6秒钟的睡眠，以允许客户在其door_call调用仍在进展期间中止。

doors/serverintr4.c

```
 1 #include     "unpipc.h"
 2 void
 3 servproc_cleanup(void *arg)
 4 {
 5     printf("servproc cancelled, thread id %ld\n", pr_thread_id(NULL));
 6 }
 7 void
 8 servproc(void *cookie, char *dataptr, size_t datasize,
 9           door_desc_t *descptr, size_t ndesc)
10 {
11     int     oldstate, junk;
12     long    arg, result;
13     Pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
14     pthread_cleanup_push(servproc_cleanup, NULL);
15     sleep(6);
16     arg = *((long *) dataptr);
17     result = arg * arg;
18     pthread_cleanup_pop(0);
19     Pthread_setcancelstate(oldstate, &junk);
20     Door_return((char *) &result, sizeof(result), NULL, 0);
21 }
```

doors/serverintr4.c

图15-31 检测客户过早终止的服务器过程

运行客户程序两次，我们看到当它们的进程被SIGALRM信号所杀灭时，shell输出了“Alarm Clock（报警时钟）”消息。

```
solaris % clientintr4 /tmp/door4 44
Alarm Clock
solaris % clientintr4 /tmp/door4 44
Alarm Clock
```

查看相应的服务器输出，我们看到每次有客户过早终止时，服务器线程确实被取消，清理处理程序也被调用。

```
solaris % serverintr4 /tmp/door4
servproc canceled, thread id 4
servproc canceled, thread id 5
```

我们运行客户程序两次是为了展示，当ID为4的线程被取消后，门函数库创建了一个新线程来处理客户的第二个服务器过程激活请求。

15.12 小结

门提供了调用同一台主机上另一个进程中某个过程的能力。下一章中我们将对这种远程过程调用概念加以扩展，讲述如何调用另一台主机上另一个进程中的某个过程。

基本的API函数比较简单。服务器调用door_create创建一个门，并给它关联一个服务器过程，然后调用fattach给该门附接一个文件系统中的路径名。客户对该路径名调用open，然后调用door_call以调用服务器进程中的服务器过程。该服务器过程通过调用door_return返回。

通常情况下，对一个门所执行的唯一权限测试是由open函数在打开该门时进行的，这种测

试基于客户的用户ID和组ID以及该门的路径名的权限位和属主/属组ID。门具有本书介绍的其他IPC形式所不具备的一个精妙特性，即服务器具有确定客户的凭证的能力，这些凭证包括客户的有效用户ID和实际用户ID以及有效组ID和实际组ID。服务器可使用这些信息来确定自己是否想给相应客户的请求提供服务。

396～397

门允许从客户向服务器以及从服务器向客户传递描述符。这是一个非常有用的技巧，因为Unix中描述符代表着许多访问手段：访问文件以进行文件或设备I/O，访问套接字或XTI以进行网络通信（UNPv1），访问门以进行远程过程调用。

调用另一个进程中的过程时，我们必须考虑对端的过早终止，这是本地过程调用所不必担心的问题。如果门服务器线程过早终止，其客户通过由door_call返回一个EINTR错误得以通知。如果门客户在其door_call调用仍在进展期间终止，其服务器线程就通过接收一个线程取消请求得到通知。该服务器线程必须确定是否处理这个取消请求。

习题

15.1　由door_call作为参数从客户传递给服务器的信息有多少字节？

15.2　在图15-6中，有必要首先调用fstat以验证所打开的描述符是一个门吗？去掉这个调用，看一看发生了什么。

15.3　Solaris 2.6中sleep(3C)的手册页面这么陈述：“The current process is suspended from exection.（当前进程从执行状态挂起。）”在图15-9中，既然这句话意味着一旦有一个线程调用sleep，整个服务器进程即阻塞，那么为什么在第一个线程（ID为4）开始运行后，门函数库仍能创建第二个和第三个线程（ID为5和6）呢？

15.4　在15.3节中我们说过，对于使用door_create创建的描述符，它们的FD_CLOEXEC位是自动设置的。然而我们可以在door_create返回后，调用fcntl把该位关掉。如果我们这么做后调用exec，再从某个客户中激活服务器过程，将发生什么？

15.5　在图15-28和图15-29中，将客户程序和服务器程序各自的两个printf调用改为输出当前时间。运行这两个程序。为什么第一次激活服务器过程在2秒后返回？

15.6　把图15-22和图15-23中保护fd的互斥锁去掉，验证程序不再正确工作。你看到了什么错误？

15.7　如果我们想要改变的唯一的服务器线程特性是启用取消，那么需要建立一个服务器创建过程吗？

15.8　验证door_revoke允许已在进行的客户调用继续完成，并确定服务器过程被取消后door_call的执行情况。

15.9　在上一道习题的解答以及图15-22中我们说过，当服务器过程或服务器创建过程需使用门描述符时，它必须是一个全局变量。这种说法并不正确。重新编写上一道习题解答的程序，让fd作为main函数中的一个自动变量。

398

15.10　在图15-23中，我们每次创建一个线程都得调用pthread_attr_init和pthread_attr_destroy。这样做是最佳的吗？

第16章 Sun RPC

16.1 概述

构筑一个应用程序时，我们首先在以下两者之间作出选择：

(1) 构筑一个庞大的单一程序，完成全部工作；

(2) 把整个应用程序散布到彼此通信的多个进程中。

如果我们选择后者，接下去的择决是：

2a) 假设所有进程运行在同一台主机上（允许IPC用于这些进程间的通信）；

2b) 假设某些进程会运行在其他主机上（要求使用进程间某种形式的网络通信）。

在图15-1中，顶部的情形是选择1，中部的情形是选择2a，底部的情形是选择2b。本书的大部分关注的是2a这种情况，也就是使用消息传递、共享内存区，并可能使用某种形式的同步来进行同一台主机上的进程间IPC。同一进程内不同线程间的IPC以及不同进程内各个线程间的IPC只是这种情形的特殊情况。

不同部分之间需要网络通信的应用程序大多数是使用*显式网络编程*（explicit network programming）方式编写的，也就是如UNPv1中讲述的那样直接调用套接字API或XTI API。使用套接字API时，客户调用socket、connect、read和write，服务器则调用socket、bind、listen、accept、read和write。我们熟悉的大多数应用程序（Web浏览器、Web服务器、Telnet客户、Telnet服务器等程序）就是以这种方式编写的。

编写分布式应用程序的另一种方式是使用*隐式网络编程*（implicit network programming）。远程过程调用（RPC）提供了这样的一个工具。我们使用早已熟悉的过程调用来编写应用程序，但是调用进程（客户）和含有被调用过程的进程（服务器）可在不同的主机上执行。客户和服务器运行在不同的主机上而且过程调用中涉及网络I/O，这样的事实对于程序员基本上是透明的。事实上衡量一个RPC软件包的测度之一就是它能使作为底层支撑的网络I/O对程序员的透明度有多大。

16.1.1 例子

作为RPC的一个例子，我们把图15-2和图15-3重新编写成改用Sun RPC代替门。客户以一个长整数调用服务器的过程，返回值则是该值的平方。图16-1给出了我们的第一个文件square.x。

其名字以.x结尾的文件称为RPC说明书文件（RPC specification file），它们定义了服务器过程以及这些过程的参数和结果。

定义参数和返回值

1~6 定义两个结构，一个用于参数（其成员为单个long变量），另一个用于结果（其成员为单个long变量）。

sunrpc/square1/square.x

```
 1 struct square_in {                    /* input (argument) */
 2     long    arg1;
 3 };
 4 struct square_out {                   /* output (result) */
 5     long    res1;
 6 };
 7 program SQUARE_PROG {
 8     version SQUARE_VERS {
 9         square_out  SQUAREPROC(square_in) = 1;    /* procedure number = 1 */
10     } =     1;                        /* version number */
11 } =     0x31230000;                   /* program number */
```

sunrpc/square1/square.x

图16-1　RPC说明书文件

定义程序、版本和过程

7~11　定义一个名为SQUARE_PROG的RPC程序，它由一个版本（SQUARE_VERS）构成，该版本中又定义了单个名为SQUAREPROC的过程。该过程的参数是一个square_in结构，其返回值则是一个square_out结构。我们还给该过程赋了一个值为1的过程号，给版本赋的值为1，给程序号赋的是一个32位十六进制值（我们将在图16-9中详细讨论程序号）。

我们使用一个随Sun RPC软件包提供的程序来编译这个说明书文件，该程序就是rpc_gen。

我们编写的下一个程序是调用我们的远程过程的客户程序main函数。图16-2给出了该函数。

400

sunrpc/square1/client.c

```
 1 #include    "unpipc.h"        /* our header */
 2 #include    "square.h"        /* generated by rpcgen */
 3 int
 4 main(int argc, char **argv)
 5 {
 6     CLIENT *cl;
 7     square_in in;
 8     square_out *outp;
 9     if (argc != 3)
10         err_quit("usage: client <hostname> <integer-value>");
11     cl = Clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, "tcp");
12     in.arg1 = atol(argv[2]);
13     if ( (outp = squareproc_1(&in, cl)) == NULL)
14         err_quit("%s", clnt_sperror(cl, argv[1]));
15     printf("result: %ld\n", outp->res1);
16     exit(0);
17 }
```

sunrpc/square1/client.c

图16-2　调用远程过程的客户程序main函数

包括进由rpcgen生成的头文件

2　#include由rpcgen产生的square.h头文件。

声明客户句柄

6　我们声明一个名为cl的客户句柄（client handle）。客户句柄意图看起来像标准I/O的FILE指针（因而有全为大写的名字CLIENT）。

获取客户句柄

11 我们调用clnt_create，它运行成功时返回一个客户句柄。

```
#include <rpc/rpc.h>

CLIENT *clnt_create(const char *host, unsigned long program,
                    unsigned long versnum, const char *protocol);
```

返回：若成功则为非空客户句柄，若出错则为NULL

与标准I/O的FILE指针一样，我们并不关心客户句柄指向什么内容。它可能是由RPC运行时系统维护的某个信息结构。clnt_create分配一个这样的结构，并返回指向它的指针，以后每次调用一个远程过程时，我们就把该指针传递给RPC运行时系统。

clnt_create的第一个参数既可以是运行我们的服务器的主机的主机名，也可以是它的IP地址。第二个参数是程序名，第三个参数是版本号，这两者都来自我们的square.x文件（图16-1）。最后一个参数是我们的协议选择，通常指定为TCP或UDP。

401

调用远程过程并输出结果

12~15 调用我们的远程过程，其中第一个参数是指向输入结构的指针（&in），第二个参数是所获取的客户句柄。（在大多数标准I/O调用中，FILE句柄是最后一个参数。类似地，RPC函数的最后一个参数通常为CLIENT句柄。）返回值是指向结果结构的指针。注意，我们给输入结构分配了空间，但是结果结构是由RPC运行时系统分配的。

在square.x说明书文件中，我们称远程过程为SQUAREPROC，但是在客户程序中，我们称它为squareproc_1。这儿的约定是：.x文件中的名字转换成小写字母形式，添上一个底划线后跟以版本号。

在服务器方，我们只编写服务器过程，如图16-3所示。服务器程序的main函数由rpc_gen程序自动生成。

sunrpc/square1/server.c

```
1 #include     "unpipc.h"
2 #include     "square.h"

3 square_out *
4 squareproc_1_svc(square_in *inp, struct svc_req *rqstp)
5 {
6     static square_out   out;

7     out.res1 = inp->arg1 * inp->arg1;
8     return(&out);
9 }
```

sunrpc/square1/server.c

图16-3 使用Sun RPC调用的服务器过程

过程参数

3~4 我们首先注意到服务器过程的名字在版本号后添加了_svc。这样允许square.h头文件中有两个ANSI C函数原型，一个是图16-2中由客户调用的函数（它的一个参数是客户句柄），另一个是实际的服务器函数（它使用的参数与由客户调用的函数不一样）。

当我们的服务器过程被调用时，传递给它的第一个参数是指向输入结构的指针，第二个参数是指向由RPC运行时系统传递的一个结构的指针，该结构含有关于这次激活的信息（我们这个简单的过程忽略了这些信息）。

执行并返回

6~8　取出输入参数并计算其平方值。该结果存放在一个结构中，而该结构的地址则作为本函数的返回值。由于我们是在从函数中返回一个变量的地址，因为该变量不能是一个自动变量。我们把它声明为static变量。

机敏的读者将注意到，这样做妨碍服务器函数成为线程安全函数。我们将在16.2节中讨论这一点，并给出一个线程安全的版本。

402

现在在Solaris下编译我们的客户程序，在BSD/OS下编译我们的服务器程序，启动服务器，然后运行客户程序：

```
solaris % client bsdi 11
result: 121
solaris % client 209.75.135.35 22
result: 484
```

第一次运行时我们指定服务器主机的主机名，第二次运行时指定它的IP地址。这表明客户程序调用的clnt_create函数以及RPC运行时函数都是既允许使用主机名，也允许使用IP地址。

接着展示服务器主机不存在或者尽管存在但没有运行我们的服务器程序时，由clnt_create返回的一些错误。

```
solaris % client nosuchhost 11
nosuchhost: RPC: Unknown host          出自RPC运行时系统
clnt_create error                      出自我们的包裹函数
solaris % client localhost 11
localhost: RPC: Program not registered
clnt_create error
```

我们已编写了一个客户程序和一个服务器程序，并展示了其中没有使用任何显式的网络编程方式。我们的客户程序只是调用两个函数（clnt_create和squareproc_1），而在服务器方，我们只是编写squareproc_1_svc函数。涉及Solaris下的XTI、BSD/OS下的套接字以及网络I/O的所有细节都由RPC运行时系统来处理。这就是RPC的目的：不需要显式的网络编程知识就允许编写分布式应用程序。

本例子的另一个重要之处在于，所用的两个系统（运行Solaris的Sparc系统和运行BSD/OS的Intel x86系统）具有不同的字节序（byte order）。其中Sparc系统是大端（big endian）字节序，Intel系统是小端（little endian）字节序（我们在UNPv1的3.4节中展示了这两种字节序）。这些字节排序上的差异也是由运行时函数库自动处理的，其中使用了一个称为*XDR*（external data representation，外部数据表示）的标准，我们将在16.8节中讨论。

本例子中客户程序和服务器程序的构建所涉及的步骤比本书中其他程序的构建都要多。下面是构建客户程序可执行文件所涉及的步骤：

```
solaris % rpcgen -C square.x
solaris % cc -c client.c -o client.o
solaris % cc -c square_clnt.c -o square_clnt.o
solaris % cc -c square_xdr.c -o square_xdr.o
solaris % cc -o client client.o square_clnt.o square_xdr.o libunpipc.a -lnsl
```

其中rpcgen的-C选项告诉它在square.h头文件中生成ANSI C原型。rpcgen还产生一个称为客户程序存根（client stub）的源文件square_clnt.c和一个名为square_xdr.c的用来处理XDR数据转换的文件。libunpipc.a是我们的函数库（存放本书中所用的函数），-lnsl选项则指定Solaris下存放网络支撑函数（包括RPC和XDR运行时系统）的系统函数库。

构建服务器程序时我们会看到类似的命令，不过rpcgen不必再运行。文件square_svc.c中含有服务器程序main函数，另外，构建客户程序时生成的含有XDR函数的square_xdr.o文件在服务器程序的构建中也需要。 403

```
solaris % cc -c server.c -o server.o
solaris % cc -c square_svc.c -o square_svc.o
solaris % cc -o server server.o square_svc.o square_xdr.o libunpipc.a -lnsl
```

这样会生成都运行在Solaris下的客户程序和服务器程序。

当客户程序和服务器程序是为不同的系统构建时（例如在我们早先的例子中，客户程序运行在Solaris下，服务器程序运行在BSD/OS下），可能需要额外的步骤。举例来说，某些文件必须共享（例如通过NFS）或者在两个系统之间复制，另外，客户程序和服务器程序都使用的文件（square_xdr.o）必须在每个系统上分别编译。

图16-4汇总了构建我们的客户-服务器例子程序所需的文件和步骤。其中三个带阴影的方框是我们必须编写的文件。短划线指出了需要C伪指令#include square.h的那些文件。

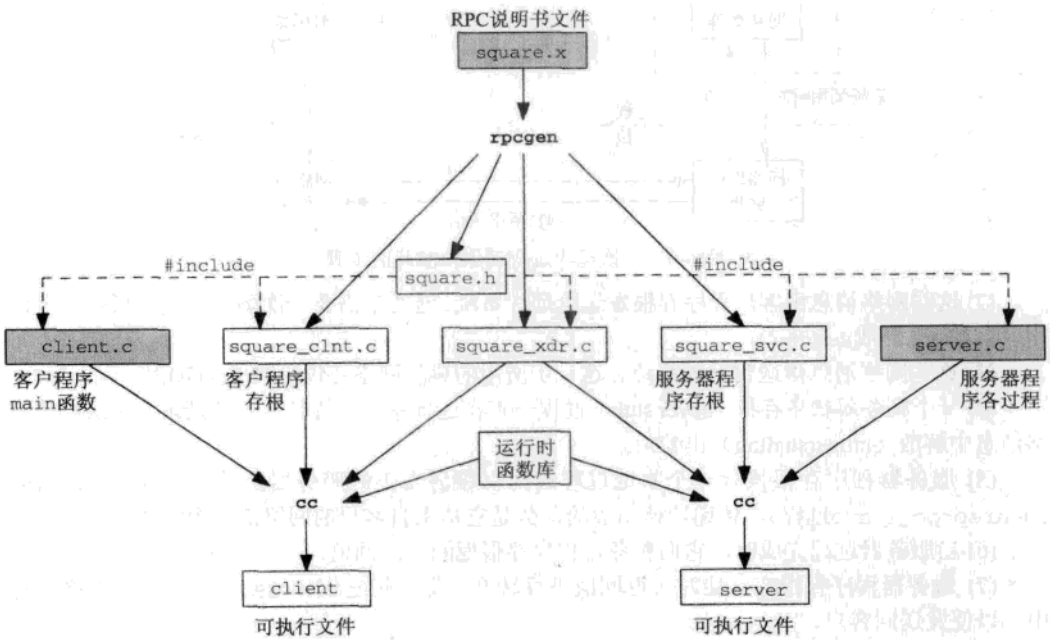


图16-4 构建一个RPC客户-服务器程序所需的步骤汇总

图16-5汇总了一次远程过程调用中通常发生的步骤。编了号的步骤是按顺序执行的。

(0) 服务器启动，它向所在主机上的端口映射器（port mapper）注册自身。然后客户启动，它调用clnt_create，该函数则与服务器主机上的端口映射器联系，以找到服务器的临时端口。clnt_create函数还建立一个与服务器的TCP连接（因为我们在图16-2中指定的协议为TCP）。 404
我们在本图中没有展示这些步骤，留待16.3节中详细地讲述。

(1) 客户调用一个称为客户程序存根（client stub）的本地过程。在图16-2中，该过程名为squareproc_1，而含有这个客户程序存根的文件是由rpcgen产生的，名为square_clnt.c。对于客户来说，客户程序存根看起来像是它想要调用的真正的服务器过程。存根的目的在于把

有待传递给远程过程的参数打成包，可能的话把它们转换成某种标准格式，然后构造一个或多个网络消息。把客户提供的参数打包成一个网络消息的过程称为集结（marshaling）。客户程序的各个例程和存根通常调用RPC运行时函数库中的函数（例如我们早先的例子中的clnt_create）。在Solaris下链接时，这些运行时库函数是从_lnsl函数库中加载的，而BSD/OS下它们是在标准C函数库中。

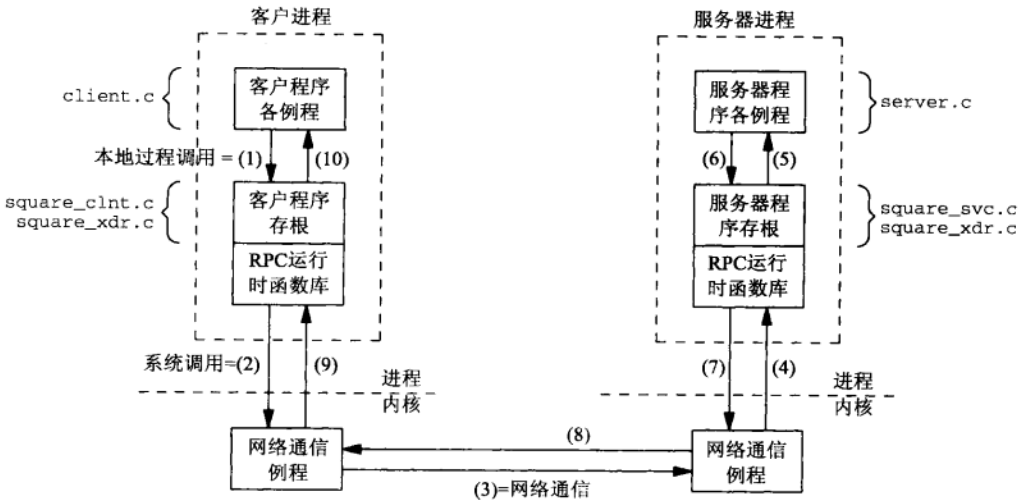


图16-5　一次远程过程调用中涉及的步骤

(2) 这些网络消息由客户程序存根发送给远程系统。这通常需要一次陷入本地内核的系统调用（例如write或sendto）。

(3) 这些网络消息传送到远程系统。这一步所用的典型网络协议为TCP或UDP。

(4) 一个服务器程序存根（sever stub）过程一直在远程系统上等待客户的请求。它从这些网络消息中解散（unmarshaling）出参数。

(5) 服务器程序存根执行一个本地过程调用以激活真正的服务器函数（图16-3中我们的
405 squareproc_1_svc过程），传递给该函数的参数是它从来自客户的网络消息中解散出来的。

(6) 当服务器过程完成时，它向服务器程序存根返回其返回值。

(7) 服务器程序存根在必要时对返回值进行转换，然后把它们集结到一个或多个网络消息中，以便发送回客户。

(8) 这些消息通过网络传送回客户。

(9) 客户程序存根从本地内核中读出这些网络消息（例如read或recvfrom）。

(10) 对返回值进行可能的转换后，客户程序存根最终返回客户函数。这一步看起来像是一个普通的过程返回客户。

16.1.2　历史

关于RPC的最早期论文之一也许是［White 1975］。按照［Corbin 1991］的陈述，White当时转到了Xerox（施乐）公司，那儿于是开发了若干个RPC系统。其中之一的Courier是于1981年面世的一个产品。关于RPC的经典论文是［Birrell and Nelson 1984］，它讲述了20世纪80年代早期在Xerox公司的Dorado单用户工作站上运行的Cedar项目的RPC机制。Xerox在大多数人还不

知道工作站是什么的时候就打算在工作站上实现RPC了!Courier的一个Unix实现版本随4.x BSD各个发行版本传播了许多年，不过现今Courier只具有历史意义了。

Sun公司于1985年发行它的RPC软件包的第一个版本。它是由Bob Lyon开发的，Bob于1983年离开Xerox加入Sun。它的正式名字是ONC/RPC：Open Network Computing Remote Procedure Call（开放的网络计算远程过程调用），不过往往被称为“Sun RPC”。技术上讲，它类似于Courier。Sun RPC的初期版本是用套接字API编写的，既能用于TCP，也能用于UDP。公开可得的源代码版本称为RPCSRC。20世纪90年代早期，Sun RPC改用TLI API重新编写，能用于内核支持的任何网络协议，其中TLI是XTI（在UNPv1第四部分讲述）的前身。套接字版本和TLI版本的公开可得源代码实现都可从ftp://playground.sun.com/pub/rpc中获得，前者的名字为rpcsrc，后者的名字为tirpcsrc（称为TI-RPC，其中“TI”代表“传输独立（transport independent）”）。

RFC 1831［Srinivasan 1995a］提供了Sun RPC的一个概貌，并描述了通过网络发送的RPC消息的格式。RFC 1832［Srinivasan 1995b］讲述了XDR，既包括所支持的数据类型，又包括它们的“在线上（on the wire）”格式。RFC 1833［Srinivasan 1995c］讲述了捆绑协议：RPCBIND及其前身端口映射器。

使用Sun RPC的最广泛流行的应用也许是NFS，即Sun的网络文件系统。正常情况下，NFS不是使用本章讲述的标准RPC工具、rpcgen以及RPC运行时函数库构造的。相反，它的大多数库例程是手工优化过的，并且因性能原因而驻留在内核中。然而支持NFS的大多数系统也支持Sun RPC。

20世纪80年代中期，Appollo公司与Sun在工作站市场展开竞争，并自行设计了称为NCA（Network Computing Architecture，网络计算体系结构）的RPC软件包与Sun RPC一较高下，其实现称为NCS（Network Computing System，网络计算系统）。NCA/RPC是RPC协议，NDR（Network Data Representation，网络数据表示）类似于Sun的XDR，NIDL（Network Interface Definition Language，网络接口定义语言）则定义了客户和服务器之间的接口（例如类似于图16-1中的.x文件）。运行时函数库称为NCK（Network Computing Kernel，网络计算内核）。 406

Apollo于1989年被Hewlett Packard（惠普）公司收购，NCA于是发展成为开放软件基金会（OSF）的分布式计算环境（Distributed Computing Environment，简称DCE），其中RPC是一个基本元素，大多数部件就构建在其上。有关DCE的详细信息可从http://www.camb.opengroup.org/tech/dce中得到。DCE RPC软件包有一个实现是公开可得的，其URL为ftp://gatekeeper.dec.com/pub/DEC/DCE。该目录还含有一个171页的文档，讲述DCE RPC软件包的内部工作原理。许多平台上有DCE可用。

Sun RPC比DCE RPC要流行得多，其原因也许是前者有免费可得的实现，而且大多数版本的Unix把Sun RPC软件包作为基本系统的一部分提供。DCE RPC通常是作为一个增值（也就意味着单独计价）特性提供的。它的公开可得实现并没有得到广泛的移植，尽管往Linux上的移植正在进展中。本书中我们只讨论Sun RPC。Courier、Sun RPC和DCE RPC这三个RPC软件包都极其相似，因为基本的RPC概念是一样的。

大多数Unix厂家提供关于Sun RPC的除手册页面外的详细文档。举例来说，Sun的文档可从http://docs.sun.com获取，在“Developer Collection（开发人员资料汇编）”第1卷中，它是名为“ONC+Developer'S Guide（ONC+开发人员指南）”的共280页的一个部分。Digital Unix的文档可从http://www.u_nix.digital.com/faqs/publications/pub_page/V40D_DOCS.HTM获取，包括一本题目为“Programming with ONC RPC（使用ONC RPC编程）”的116页手册。

RPC本身是一个让人争议的主题。http://www.kohala.com/~rstevens/papers.others/rpc.comments.txt中收集了关于这个主题的8篇文章。

本章中我们假设给大多数例子使用TI-RPC（早先提及的传输独立版本的RPC），其中TCP和UDP作为TI-RPC支持的协议讨论，不过TI-RPC能够支持主机所能支持的任何协议。

16.2　多线程化

回想图15-9，我们从中展示了由门服务器执行的自动线程管理，由此默认提供了一个并发服务器。我们现在展示Sun RPC默认提供的是一个迭代服务器（iterative server）。我们从上一节中的例子程序着手，并只修改其中的服务器过程。图16-6给出了这个新函数，它输出所在线程的线程ID，睡眠5秒钟，再输出自己的线程ID，然后返回。

sunrpc/square2/server.c

```
 1 #include    "unpipc.h"
 2 #include    "square.h"

 3 square_out *
 4 squareproc_1_svc(square_in *inp, struct svc_req *rqstp)
 5 {
 6     static square_out out;

 7     printf("thread %ld started, arg = %ld\n",
 8           pr_thread_id(NULL), inp->arg1);
 9     sleep(5);
10     out.res1 = inp->arg1 * inp->arg1;
11     printf("thread %ld done\n", pr_thread_id(NULL));

12     return(&out);
13 }
```

sunrpc/square2/server.c

图16-6　睡眠5秒钟的服务器过程

我们启动服务器，然后运行客户程序三次：

```
solaris % client localhost 22 & client localhost 33 & \
client localhost 44 &
[3]     25179
[4]     25180
[5]     25181
solaris % result: 484              shell提示符输出后约5秒
result: 1936                       另一个5秒后
result: 1089                       另一个5秒后
```

尽管光看这些输出，我们不能识别每个客户输出各自的结果时彼此间有5秒钟的等待发生。然而要是查看服务器的输出，我们就会看到各个客户请求是迭代地处理的：处理完第一个客户的请求后，接着处理第二个客户的请求直到处理完毕，最后是处理第三个客户的请求直到处理完毕。

```
solaris % server
thread 1 started, arg = 22
thread 1 done
thread 1 started, arg = 44
thread 1 done
thread 1 started, arg = 33
thread 1 done
```

可看出单个线程在处理所有的客户请求：默认情况下服务器并不多线程化。

第15章中我们的门服务器程序从shell启动时都运行在前台，例如：

```
solaris % server
```

这样允许我们在服务器过程中放置调试用的printf调用。但是Sun RPC服务器默认作为守护进程运行，也就是执行了UNPv1的12.4节[①]中概括出的若干步骤。这就要求从服务器过程中调用syslog来输出任何诊断信息。然而我们的做法是在编译服务器程序时指定C编译器标志-DDEBUG，这跟在服务器程序存根（由rpcgen产生的square_svc.c文件）中放置如下行是等效的：

```
#define DEBUG
```

这么一来就阻止了服务器程序main函数将自身变为一个守护程序，服务器于是继续连接到启动它的终端上。这就是我们可以从服务器过程中调用printf的原因。

Sun RPC是随Solaris 2.4提供多线程化的服务器的，它通过向rpcgen指定一个-M命令行选项启用。这使由rpcgen产生的服务器代码变得线程安全。另一个选项-A是让服务器根据处理新客户请求的需要自动创建线程。我们运行rpcgen时，同时使能这两个选项。 407 ~ 408

客户程序和服务器程序的源代码都需要修改，这是我们应该预期到的，因为我们在图16-3中使用了static类型变量。对square.x文件的唯一改动是把版本号从1改为2。服务器过程的参数结构和结果结构的声明都不变。

图16-7给出了新的客户程序。

sunrpc/square3/client.c

```
 1 #include    "unpipc.h"
 2 #include    "square.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     CLIENT *cl;
 7     square_in in;
 8     square_out out;

 9     if (argc != 3)
10         err_quit("usage: client <hostname> <integer-value>");

11     cl = Clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, "tcp");

12     in.arg1 = atol(argv[2]);
13     if (squareproc_2(&in, &out, cl) != RPC_SUCCESS)
14         err_quit("%s", clnt_sperror(cl, argv[1]));

15     printf("result: %ld\n", out.res1);
16     exit(0);
17 }
```

sunrpc/square3/client.c

图16-7 用于多线程化服务器的客户程序main函数

声明存放结果的变量

8 声明一个square_out类型的变量，而不是指向该类型的一个指针。

过程调用的新参数

12~14 指向我们的out变量的指针成为squareproc_2的第二个参数，客户句柄则是最后一个参数。该函数返回的不再是指向结果的指针（如图16-2中所示），而是返回RPC_SUCCESS，或者返回表示发生错误的某个其他值。<rpc/clnt_stat.h>头文件中的clnt_stat枚举列出了所有可能的出错返回值（enum）。

① 此处为UNPv1第2版英文原版书节号，第3版为12.4节。——编者注

图16-8给出了新的服务器过程。与图16-6一样，它输出自己所在线程的线程ID，睡眠5秒钟，输出另一个消息，然后返回。

sunrpc/square3/server.c

```
 1 #include    "unpipc.h"
 2 #include    "square.h"

 3 bool_t
 4 squareproc_2_svc(square_in *inp, square_out *outp, struct svc_req *rqstp)
 5 {
 6     printf("thread %ld started, arg = %ld\n",
 7            pr_thread_id(NULL), inp->arg1);
 8     sleep(5);
 9     outp->res1 = inp->arg1 * inp->arg1;
10     printf("thread %ld done\n", pr_thread_id(NULL));
11     return(TRUE);
12 }

13 int
14 square_prog_2_freeresult(SVCXPRT *transp, xdrproc_t xdr_result,
15                           caddr_t result)
16 {
17     xdr_free(xdr_result, result);
18     return(1);
19 }
```

sunrpc/square3/server.c

图16-8　多线程化的服务器过程

新的参数和返回值

3~12　多线程化所需的变动涉及函数参数和返回值。我们不再返回一个指向结果结构的指针（如图16-3中所示），指向该结构的指针现在成了本函数的第二个参数。指向svc_req结构的指针现在变为第三个参数。本函数的返回值在成功时为TRUE，在发生错误时为FALSE。

释放XDR内存空间的新函数

13~19　我们必须完成的另一个源代码变动是提供一个函数来释放自动分配的内存空间。该函数是在服务器过程返回并且其结果已发送给客户后，从服务器程序存根中调用的。在图16-8中，我们只是调用普通的xdr_free函数。（我们将随图16-19和习题16.10详细讨论该函数。）要是我们的服务器过程曾经分配了任何必要的内存空间以容纳结果（譬如说一个链表），这个新函数就可以释放这部分内存空间。

构造出新的客户程序和服务器程序后，我们再次同时运行客户程序的三个副本：

```
solaris % client localhost 55 & client localhost 66 & \
client localhost 77 &
[3]     25427
[4]     25428
[5]     25429
solaris % result: 4356
result: 3025
result: 5929
```

这一次我们能够辨别出那三个结果是一个紧接一个地输出的。查看服务器的输出，我们看到服务器使用了三个线程，它们是同时运行的。

409
~
410

```
solaris % server
thread 1 started, arg = 55
thread 4 started, arg = 77
```

```
thread 6 started, arg = 66
thread 6 done
thread 1 done
thread 4 done
```

按照多线程化的要求修改源代码有一个不幸的副作用，那就是并非所有系统都支持这个特性。举例来说，Digital Unix 4.0B和BSD/OS 3.1提供的都是不支持多线程化的较老的RPC系统。这意味着如果我们想在这两种类型的系统上编译和运行一个程序，那么在其客户程序和服务器程序中必须加入一些#ifdef伪指令以处理在调用序列上存在的差异。当然举例来说，BSD/OS上未线程化的一个客户仍能调用运行在Solaris上的一个多线程化的服务器过程，但是如果我们有一个希望在这两种类型的系统上都能编译的RPC客户程序（或服务器程序），那就有必要修改源代码以处理这些差异。

16.3 服务器捆绑

在描述图16-5时，我们掩饰了第0步中的细节：服务器如何向它的本地端口映射器（port mapper）注册自身，客户如何发现服务器的端口值。首先应注意的是，运行RPC服务器的任何主机必须在运行端口映射器。赋给端口映射器的是TCP端口111和UDP端口111，它们是赋给Sun RPC的唯一的因特网固定端口。RPC服务器总是先捆绑一个临时端口，再向本地端口映射器注册自己的临时端口。当一个客户启动时，它必须首先跟服务器主机上的端口映射器联系，询问服务器的临时端口号，然后跟这个临时端口上的服务器联系。端口映射器提供了一个范围局限于所在系统的名字服务。

有些读者会声称NFS也有一个固定的端口号2049。尽管许多实现默认使用这个端口，而且某些较早的实现还把这个端口号硬编码到客户程序和服务器程序中，但是大多数当今的实现允许使用其他端口号。大多数NFS客户也是通过与服务器主机上的端口映射器联系来获取NFS服务器的端口号的。

随着Solaris 2.x的出现，Sun公司把端口映射器改名为RPCBIND。换名的原因在于，“端口”一词隐指因特网端口，而TI-RPC软件包能够工作在任何网络协议上，不只是工作在TCP和UDP协议上。我们还是使用传统的名字：端口映射器。另外在下面的讨论中，我们假设服务器主机只支持TCP和UDP协议。

服务器和客户是按如下的步骤执行的。

(1) 当系统进入多用户模式时，端口映射器启动。其可执行文件名一般为portmap或rpcbind。

(2) 当我们的服务器启动时，它的main函数（该函数属于由rpcgen产生的服务器程序存根的一部分）调用库函数svc_create。svc_create确定本主机所支持的网络协议，并为每个协议创建一个传输端点（例如套接字），给TCP和UDP端点各捆绑一个临时端口。该函数然后与本地的端口映射器联系，向它注册（TCP和UDP）这两个临时端口号以及调用程序的RPC程序号和版本号。

端口映射器本身是一个RPC程序，服务器就是使用RPC调用向端口映射器注册自身的（不过所用端口为已知的111）。RFC 1833［Srinivasan 1995c］中有端口映射器所支持过程的相关说明。这个RPC程序存在三个版本：版本2是历史久远的端口映射器，只处理TCP和UDP端口，版本3和版本4则采用较新的RPCBIND协议。

通过执行rpcinfo程序，我们可以看到已向端口映射器注册了的所有RPC程序。我们可执

行该程序来验证端口映射器本身使用端口号111。

```
solaris % rpcinfo -p
   program vers proto   port  service
    100000    4   tcp    111  rpcbind
    100000    3   tcp    111  rpcbind
    100000    2   tcp    111  rpcbind
    100000    4   udp    111  rpcbind
    100000    3   udp    111  rpcbind
    100000    2   udp    111  rpcbind
```

（我们已省略掉了输出中的许多额外的行。）我们看到Solaris 2.6支持所有三个版本的协议，全部在端口111上，并且或者使用TCP，或者使用UDP。从RPC程序号到服务号的映射关系通常存放在文件/etc/rpc中。在BSD/OS 3.1下执行同样的命令，结果表明它只支持版本2的端口映射器RPC程序。

```
bsdi % rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
```

Digital Unix 4.0B也只支持版本2：

```
alpha % rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
```

然后我们的服务器程序进入睡眠，等待客户请求的到达。这种请求可以是在其TCP端口上的一个新连接，也可以是在其UDP端口上的一个UDP数据报的到达。启动图16-3给出的服务器后执行rpcinfo，我们看到：

```
solaris % rpcinfo -p
   program vers proto   port  service
 824377344    1   udp  47972
 824377344    1   tcp  40849
```

其中824377344等于0x31230000，它是我们在图16-1所赋的程序号。我们还在该图中赋了一个值为1的版本号。注意，服务器准备好或者使用TCP或者使用UDP接受客户请求，客户则在创建客户句柄时选择使用其中哪个协议（图16-2中clnt_cre_ate的最后一个参数）。

412

(3) 客户启动并调用clnt_create。该函数的参数（图16-2）包括：服务器主机的主机名或IP地址、程序号、版本号及指定所用协议的字符串。客户向服务器主机的端口映射器发送一个RPC请求（这个RPC消息通常使用UDP作为所用协议），询问关于所指定的程序、版本和协议的信息。假设成功的话，作为答复的服务器端口号就保存到客户句柄中，供将来使用该句柄的所有RPC调用参考。

在图16-1中，我们给例子程序使用了值为0x31230000的程序号。这个32位的程序号是划分成组的，如图16-9所示。

程　序　号	说　明
0x00000000~0x1fffffff	由Sun定义
0x20000000~0x3fffffff	由用户定义
0x40000000~0x5fffffff	临时（供客户编写的应用程序用）
0x60000000~0xffffffff	保留

图16-9　Sun RPC的程序号范围

rpcinfo程序显示本系统上当前已注册的程序。一个给定系统上所支持RPC程序的相关信息的另一个来源是目录/usr/inelude/rpcsvc中的.x文件。

inetd和RPC服务器

默认情况下，由rpcgen创建的服务器可由inetd超级服务器激活。（UNPv1的12.5节[①]详细讨论了inetd。）查看由rpcgen产生的服务器程序存根，可看到服务器程序main函数启动时，会检查标准输入是不是一个XTI端点，若是则假定自身是由inetd启动的。

为了支持这一特性，在创建了一个将由inetd激活的一个RPC服务器之后，必须以该服务器的信息更新/etc/inetd.conf配置文件，这些信息包括：RPC程序名、所支持的程序号、所支持的协议、服务器程序可执行文件的路径名。作为一个例子，下面是出自Solaris配置文件中的一行：

```
rstatd/2-4  tli  rpc/datagram_v  wait  root
    /usr/lib/netsvc/rstat/rpc.rstatd rpc.rstatd
```

其中第一栏是程序名（它将被映射成相应的程序号，这种映射关系存放在/etc/rpc文件中），所支持的版本为2、3和4。下一栏指定一个XTI端点（与套接字端点相对立），第三栏指定所有可见的数据报协议都受支持。查看文件/etc/neteonfig，看到这样的协议有两个：UDP和/dev/clts。（UNPv1第29章讲述该文件和XTI地址。）第四栏wait告诉inetd在监听发往相应XTI端点的下一个客户请求前，先等待本服务器当前这次激活终止。/etc/inetd.conf中的所有RPC服务器都指定wait属性。

再下一栏root指定本程序将在这个用户ID下运行，最后两栏是本程序可执行文件的路径名以及程序名，外带传递给该程序的任何命令行参数（本程序没有命令行参数）。 413

inetd将给所指定的程序及版本创建XTI端点，并向端口映射器登记这些端点。我们可使用rpcinfo程序验证这一点：

```
solaris % rpcinfo | grep statd
100001    2    udp       0.0.0.0.128.11       rstatd     superuser
100001    3    udp       0.0.0.0.128.11       rstatd     superuser
100001    4    udp       0.0.0.0.128.11       rstatd     superuser
100001    2    ticlts    \000\000\020,        rstatd     superuser
100001    3    ticlts    \000\000\020,        rstatd     superuser
100001    4    ticlts    \000\000\020,        rstatd     superuser
```

其中第四栏是XTI地址的可显示格式（它是逐个字节输出的），128×256+11=32779是分配给该XTI端点的UDP临时端口号。

当一个UDP数据报到达端口32779时，inetd将检测到有一个数据报已准备好被读入，于是它fork并exec程序/usr/lib/netsvc/rstat/rpc.rstatd。在fork和exec之间，该服务器的XTI端点被复制到描述符0、1和2上，inetd的所有其他描述符则都被关闭（UNPv1的图12-7[②]）。inetd还将停止监听发往该XTI端点的新的客户请求，直到当前这次服务器激活（它是inetd的一个子进程）终止为止，这是因为该服务器在配置文件中指定了wait属性的缘故。

假设该程序是由rpcgen产生的，它将检测到标准输入是一个XTI端点，于是相应地把该端点初始化为一个RPC服务器端点。这是通过调用RPC函数svc_tli_create和svc_reg完成的，我们不再讨论它们。第二个函数并不向端口映射器登记本服务器，这步工作只由inetd在启动

① 此处为UNPv1第2版英文原版书节号，第3版为13.5节。——编者注
② 此处为UNPv1第2版英文原版书图号，第3版为图13-7。——编者注

时做一次。该RPC服务器接着进入循环，由名为svc_run的函数读入待处理的数据报，并调用相应的服务器过程来处理这个客户请求。

通常情况下，由inetd激活的服务器处理完一个客户请求后就终止，以此允许inetd等待下一个客户请求。作为一种优化手段，由rpcgen产生的RPC服务器会等待一小段时间（默认值为2分钟），以防另一个客户请求在这段时间内到达。如果真是这样，这个已经在运行的现有服务器将读入新的数据报并处理其请求。这就避免了给短时间内相继到达的多个客户请求分别执行一次fork和一次exec的开销。过了这段小的等待期后，服务器将终止。这将给inetd产生一个SIGCHLD信号，从而导致它再次开始查看该XTI端点上有无数据报到达。

16.4　认证

默认情况下，RPC请求中没有标识客户的信息。服务器回答客户的请求时并不关心客户是谁。这称为空认证（null authentication）或AUTH_NONE。

下一个认证级别称为Unix认证（Unix authentication）或AUTH_SYS。客户必须告诉RPC运行时系统随每个请求携带其身份信息（主机名、有效用户ID、有效组ID和可能多个辅助组ID）。我们把16.2节中的客户-服务器程序修改成包括Unix认证。图16-10给出了其中的客户程序。

414

sunrpc/square4/client.c

```
 1 #include    "unpipc.h"
 2 #include    "square.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     CLIENT  *cl;
 7     square_in in;
 8     square_out out;

 9     if (argc != 3)
10         err_quit("usage: client <hostname> <integer-value>");

11     cl = Clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, "tcp");

12     auth_destroy(cl->cl_auth);
13     cl->cl_auth = authsys_create_default();

14     in.arg1 = atol(argv[2]);
15     if (squareproc_2(&in, &out, cl) != RPC_SUCCESS)
16         err_quit("%s", clnt_sperror(cl, argv[1]));

17     printf("result: %ld\n", out.res1);
18     exit(0);
19 }
```

sunrpc/square4/client.c

图16-10　提供Unix认证的客户程序

12~13　这两行是新的。我们首先调用auth_destroy销毁与本客户句柄关联的先前的认证，也就是默认创建的空认证。然后调用函数authsys_create_default创建相应的Unix认证结构，并把该结构存入客户句柄CLIENT结构的cl_auth成员中。客户程序的其余部分与图16-7的一样。

图16-11给出了新的服务器过程，它从图16-8修改而来。我们没有给出square_prog_2_freeresult函数，它没有变动。

sunrpc/square4/server.c

```
1 #include    "unpipc.h"
2 #include    "square.h"

3 bool_t
4 squareproc_2_svc(square_in *inp, square_out *outp, struct svc_req *rqstp)
5 {
6     printf("thread %ld started, arg = %ld, auth = %d\n",
7           pr_thread_id(NULL), inp->arg1, rqstp->rq_cred.oa_flavor);
8     if (rqstp->rq_cred.oa_flavor == AUTH_SYS) {
9         struct authsys_parms  *au;

10        au = (struct authsys_parms *)rqstp->rq_clntcred;
11        printf("AUTH_SYS: host %s, uid %ld, gid %ld\n",
12               au->aup_machname, (long) au->aup_uid, (long) au->aup_gid);
13    }

14    sleep(5);
15    outp->res1 = inp->arg1 * inp->arg1;
16    printf("thread %ld done\n", pr_thread_id(NULL));
17    return(TRUE);
18 }
```

sunrpc/square4/server.c

图16-11 寻找Unix认证的服务器过程

6~8 现在使用一个指向svc_req结构的指针，它总是作为一个参数传递给服务器过程。

```
struct svc_req {
  u_long                rq_prog;      /* program number */
  u_long                rq_vers;      /* version number */
  u_long                rq_proc;      /* procedure number */
  struct opaque_auth    rq_cred;      /* raw credentials */
  caddr_t               rq_clntcred;  /* cooked credentials (read-only) */
  SVCXPRT               *rq_xprt;     /* transport handle */
};

struct opaque_auth {
  enum_t    oa_flavor;      /* flavor: AUTH_xxx constant */
  caddr_t   oa_base;        /* address of more auth stuff */
  u_int     oa_length;      /* not to exceed MAX_AUTH_BYTES */
};
```

其中rq_cred成员含有原始认证信息，它的oa_flavor成员是一个标识认证类型的整数。“原始（raw）”一词意味着RPC运行时系统没有处理由oa_base指向的信息。然而，如果是运行时系统支持的认证类型的话，那么由rq_clntcred指向的成熟（cooked）凭证已被运行时系统处理成某个适合那种认证类型的结构。我们输出认证类型，并检查它是否等于AUTH_SYS。

9~12 对于Unix认证，指向成熟凭证的指针（rq_clntcred）所指的是含有客户身份的一个authsys_parma结构：

```
struct authsys_parms {
  u_long  aup_time;     /* credentials creation time */
  char   *aup_machname;/* hostname where client is located */
  uid_t   aup_uid;      /* effective user ID */
  gid_t   aup_gid;      /* effective group ID */
  u_int   aup_len;      /* #elements in aup_gids[] */
  gid_t  *aup_gids;     /* supplementary group IDs */
};
```

我们取得指向这个结构的指针后，输出客户的主机名、有效用户ID和有效组ID。

启动我们的服务器，然后运行客户程序一次，我们从服务器得到如下的输出：

415
~
416

```
solaris % server
thread 1 started, arg = 44, auth = 1
AUTH_SYS: host solaris.kohala.com, uid 765, gid 870
thread 1 done
```

Unix认证很少使用，因为它极易被攻破。我们可以很容易地自行构造含有Unix认证信息的RPC分组，把其中的用户ID和组ID设置成我们想要的任何值，然后发送给服务器。服务器没有办法验证我们就是所声称的客户。

> 实际上NFS默认使用Unix认证，然而NFS请求通常是由NFS客户主机的内核发出的，而且通常使用一个保留端口（UNPv1的2.7节）。有些NFS服务器配置成只响应从一个保留端口到达的客户请求。如果你信任想要往其上安装你的文件系统的客户主机，那么你也在信任该客户主机的内核能正确地标识自己的用户。要是服务器不要求客户使用一个保留端口，黑客们就能够自行编写出向NFS服务器发送NFS请求的程序，其中的Unix认证ID可设置成任意想要的值。即使服务器要求客户使用一个保留端口，如果你拥有一个自己具有超级用户特权的系统，而且你能够把自己的系统接入网络中，那么你仍可以向服务器发送自己的NFS请求。

不论是请求还是应答，一个RPC分组中实际都含有两个与认证相关的字段：凭证（credential）和验证器（verifier）（图16-30和图16-32）。一个常用的类比是带相片的身份证件（护照、驾驶执照，等等）。凭证是印制的信息（姓名、住址、出生日期，等等），验证器则是相片。验证器还有其他的形式，不过相片的效果要比列出身高、体重、性别等更好。如果我们有一个没有任何形式的识别信息的身份证件（图书馆借书卡往往是这样的例子），那么我们是光有凭证而没有验证器，因而任何人都可以使用它并声称是它的主人。

在空认证的情况下，凭证和验证器都是空的。使用Unix认证时，凭证中含有主机名、用户ID和组ID，但是验证器是空的。RPC还支持其他形式的认证，它们的凭证和验证器含有的信息也不一样。

AUTH_SHORT　另一种形式的Unix认证，在从服务器返回客户的RPC应答的验证器字段中发送。它的信息量比完整的Unix认证少，而且客户可以在后续的请求中把它作为凭证发送回服务器。这种认证类型的意图在于节省网络带宽和服务器主机的CPU时间。

AUTH_DES　DES是数据加密标准（Data Encryption Standard）的首字母缩写，这种认证形式基于私钥和公钥加密机制。这种方案也称为安全的*RPC*（secure RPC），当用作NFS的基础时，这样的NFS称为安全的*NFS*（secure NFS）。

AUTH_KERB　这种方案基于MIT的Kerberos认证系统。

[Garfinkel and Spafford 1996] 第19章详细讨论了后两种形式的认证，包括它们的设置和使用。

16.5　超时和重传

417

现在查看Sun RPC使用的超时和重传策略。Sun RPC使用了两个超时值。

(1) 总超时（total timeout）：一个客户等待其服务器的应答的总时间量。TCP和UDP都使用该值。

(2) 重试超时（retry timeout）：只用于UDP，是一个客户在等待其服务器的应答期间每次重

传请求的间隔时间。

首先应注意使用TCP不需要重试超时，因为TCP是一个可靠的协议。如果服务器主机没有接收到客户的请求，客户主机的TCP就会超时并重传该请求。当服务器主机接收到客户的请求时，服务器主机的TCP会向客户主机的TCP确认这次收到。如果服务器的确认是数据丢失，导致客户主机的TCP重传该请求，那么当服务器主机的TCP接收到这个重复的数据时，它将丢弃该数据，并再次发出一个确认。有了可靠的协议后，可靠性（超时、重传、对重复数据或重复确认的处理）就由传输层提供，RPC运行时系统不再关心它。由客户主机RPC层发出的一个请求将由服务器主机RPC层作为一个请求接收（如果这个请求从未得到过确认，那么客户主机RPC层将得到一个出错指示），而不管在网络层和传输层上发生什么。

创建一个客户句柄后，我们可以调用clnt_control查询或设置影响该句柄的选项。这类似于给一个描述符调用fcntl，或者给一个套接字调用getsockopt和setsockopt。

```
#include <rpc/rpc.h>

bool_t clnt_control(CLIENT *cl, unsigned int request, char *ptr);
                                                    返回：若成功则为TRUE，若出错则为FALSE
```

其中*cl*是客户句柄，由*ptr*指向的内容则取决于*request*。

我们把图16-2给出的客户程序修改为调用该函数并输出RPC的两个超时值。图16-12给出了这个新的客户程序。

sunrpc/square5/client.c

```
 1 #include    "unpipc.h"
 2 #include    "square.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     CLIENT *cl;
 7     square_in in;
 8     square_out *outp;
 9     struct timeval tv;

10     if (argc != 4)
11         err_quit("usage: client <hostname> <integer-value> <protocol>");

12     cl = Clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, argv[3]);

13     Clnt_control(cl, CLGET_TIMEOUT, (char *) &tv);
14     printf("timeout = %ld sec, %ld usec\n", tv.tv_sec, tv.tv_usec);
15     if (clnt_control(cl, CLGET_RETRY_TIMEOUT, (char *) &tv) == TRUE)
16         printf("retry timeout = %ld sec, %ld usec\n", tv.tv_sec, tv.tv_usec);

17     in.arg1 = atol(argv[2]);
18     if ( (outp = squareproc_1(&in, cl)) == NULL)
19         err_quit("%s", clnt_sperror(cl, argv[1]));

20     printf("result: %ld\n", outp->res1);
21     exit(0);
22 }
```

sunrpc/square5/client.c

图16-12 查询并输出两个RPC超时值的客户程序

协议是一个命令行选项

10~12 现在作为另一个命令行选项来指定协议，并把它用作clnt_create的最后一个参数。

取得总超时

13~14　clnt_control的第一个参数是客户句柄，第二个参数是请求，第三个参数则通常是指向一个缓冲区的指针。我们的第一个请求是CLGET_TIMEOUT，它在其地址为第三个参数的timeval结构中返回总超时值。该请求对所有协议都有效。

尝试取得重试超时

15~16　我们的下一个请求是获取重试超时的CLGET_RETRY_TIMEOUT，不过这个超时只对UDP有效。因此，如果返回值为FALSE，我们就什么都不输出。

我们还把图16-6给出的服务器程序修改为睡眠1000秒而不是5秒，以保证客户的请求超时。在主机bsdi上启动服务器后运行客户程序两次，一次指定TCP协议，一次指定UDP协议，然而结果并不是我们所预期的：

```
solaris % date ; client bsdi 44 tcp ; date
Web Apr 22 14:46:57 MST 1998
timeout = 30 sec, 0 usec                    超时值说是30秒
bsdi: RPC: Timed out
Web Apr 22 14:47:22 MST 1998                但这是在25秒后输出的

solaris % date ; client bsdi 55 udp ; date
Web Apr 22 14:48:05 MST 1998
timeout = -1 sec, -1 usec                   奇怪
retry timeout = 15 sec, 0 usec              这倒是正确的
bsdi: RPC: Timed out
Web Apr 22 14:48:31 MST 1998                大约25秒之后
```

在使用TCP的情况下，由cntl_control返回的总超时值为30秒，但是我们的测量给出一个25秒的超时值。至于使用UDP的情况，所返回的总超时值为-1。

418~419　为查看究竟发生了什么，我们分析由rpcgen产生的客户程序存根文件square_clnt.c中的squareproc_1函数。该函数调用clnt_call，传递给它的最后一个参数是名为TIMEOUT的一个timeval结构，这个变量在该文件中声明，它的初始值为25秒。传递给clnt_call的这个参数覆盖了用于TCP的30秒默认超时值和用于UDP的默认值-1。这个参数一直沿用到客户以一个CLSET_TIMEOUT请求调用clnt_control显式地设置总超时值为止。如果我们想改变总超时值，那就应该调用clnt_control，而不应该修改客户程序存根中的timeval结构变量TIMEOUT。

> 验证UDP重试超时的唯一办法是使用tcpdump观察分组。这种观察表明，第一个数据报是在客户一启动后就发送的，下一个数据报的发送则在约15秒之后。

16.5.1　TCP连接管理

使用tcpdump观察刚才描述的客户-服务器程序的运行情况，我们首先看到TCP的三路握手，然后是客户发送其请求，服务器确认这个请求。大约25秒后，客户发送一个FIN分节，它是由客户进程即将终止引起的，接着是TCP连接终止序列的其余三个分节。UNPv1的2.5节详细讲述了这些分节。

我们想要展示Sun RPC在使用TCP连接上的以下特征：客户通过调用clnt_create建立一个新的TCP连接，这个连接由与所指定的程序和版本相关联的所有过程调用来使用。一个客户的TCP连接或者通过调用clnt_destroy显式地终止，或者由客户进程的终止隐式地终止。

```
#include <rpc/rpc.h>

void clnt_destroy(CLIENT *cl);
```

我们从图16-2的客户程序着手，把它修改成调用服务器过程两次，然后调用clnt_destroy，接着pause。图16-13给出了这个新的客户程序。

sunrpc/square9/client.c

```
 1 #include    "unpipc.h"              /* our header */
 2 #include    "square.h"              /* generated by rpcgen */

 3 int
 4 main(int argc, char **argv)
 5 {
 6     CLIENT  *cl;
 7     square_in in;
 8     square_out *outp;

 9     if (argc != 3)
10         err_quit("usage: client <hostname> <integer-value>");

11     cl = Clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, "tcp");

12     in.arg1 = atol(argv[2]);
13     if ( (outp = squareproc_1(&in, cl)) == NULL)
14         err_quit("%s", clnt_sperror(cl, argv[1]));
15     printf("result: %ld\n", outp->res1);

16     in.arg1 *= 2;
17     if ( (outp = squareproc_1(&in, cl)) == NULL)
18         err_quit("%s", clnt_sperror(cl, argv[1]));
19     printf("result: %ld\n", outp->res1);

20     clnt_destroy(cl);

21     pause();

22     exit(0);
23 }
```

sunrpc/square9/client.c

图16-13 检查TCP连接使用情况的客户程序

运行这个程序得到了预期的输出：

```
solaris % client kalae 5
result: 25
result: 100
                                程序只是等待，直到我们杀死它为止
```

不过验证我们早先给出的声明只能通过观察tcpdump的输出。这个输出表明，有一个TCP连接被创建了（通过调用clnt_create），而且两个客户请求都使用这个连接。该连接然后由clnt_destroy调用终止，尽管当时客户进程还没有终止。

16.5.2 事务ID

超时和重传策略的另一部分是使用事务ID（transaction ID）即XID来标识客户请求和服务器应答的。当一个客户发出一个RPC调用时，RPC运行时系统给这个调用赋一个32位整数XID，该值伴随RPC消息发送。服务器必须伴随其应答返回这个XID。RPC运行时系统重传一个请求时，XID并不改变。使用XID的目的有两个。

(1) 客户验证应答的XID等于早先随请求发送的XID，否则的话客户忽略这个应答。如果使用的是TCP协议，那么客户收到XID不正确的应答的机会非常罕见，然而如果使用的是UDP协议，而且存在重传请求的可能，网络也易于丢失分组，那么接收到XID不正确的应答是绝对可能的。

(2) 服务器允许维护一个存放已发送应答的高速缓存（cache），而用于确定一个请求是否为一个重复请求的条目之一是XID。我们稍后讨论这个高速缓存。

TI_RPC软件包使用以下算法来给一个新请求选择一个XID，其中^运算符是C的按位异或：

```
struct timeval now;

gettimeofday(&now, NULL);
xid = getpid() ^ now.tv_sec ^ now.tv_usec;
```

420 ~ 421

16.5.3 服务器重复请求高速缓存

为使能RPC运行时系统维护一个重复请求高速缓存，服务器必须调用svc_dg_enablecache。一旦启用了这个高速缓存，就没有办法关掉它（除非服务器进程终止）。

```
#include <rpc/rpc.h>

int svc_dg_enablecache(SVCXPRT *xprt, unsigned long size);
```

返回：若成功则为0，若出错则为-1

其中*xprt*是一个传输句柄，该指针是svc_req结构的一个成员（16.4节）。而该结构的地址是作为一个参数传递给服务器过程的。*size*是需为之分配内存空间的高速缓存项数。

启用该高速缓存后，服务器便为它所发送的全部应答维护一个FIFO（先进先出）高速缓存。每个应答是由如下信息唯一标识的：

- 程序号；
- 版本号；
- 过程号；
- XID；
- 客户地址（IP地址和UDP端口号）。

每当服务器中的RPC运行时系统接收到一个客户请求时，首先会搜索重复请求高速缓存，看其中是否已有该请求的一个应答。如果有的话，这个高速缓存的应答就返回给客户，而不再调用相应的服务器过程。

重复请求高速缓存的目的是：当接收到对某个服务器过程的多个重复请求时，避免多次调用该服务器过程，因为该过程也许不是等势的。在网络中接收到重复请求的可能原因是应答丢失或者客户重传请求超前于应答的接收。注意，这种重复请求高速缓存只适用于像UDP这样的数据报协议，因为使用TCP协议时应用绝对看不到重复的请求，请求的重复问题是由TCP处理的（见习题16.6）。

16.6 调用语义

在图15-29给出的门客户程序中，当客户的door_call调用被一个捕获的信号所中断时，客户向服务器重传其请求。然而我们接下去展示出这将导致相应的服务器过程被调用两次，而不是一次。我们随后把服务器过程划分成等势（能够任意多次无差错地调用）和非等势（例如从某个银行账户上减去一笔费用）两大类。

过程调用可划分成以下三个类别。

(1) 正好一次（exactly once）：过程只能不多不少地执行一次。这类操作难于实现，因为服务器存在崩溃的可能。

422

(2) 最多一次（at most once）：过程根本不执行或只执行一次。如果正常地返回到调用者，我们就知道该过程执行了一次。然而如果是出错返回，我们就不能肯定该过程执行了一次还是根本没有执行。

(3) 最少一次（at least once）：过程至少执行一次，不过有可能是多次。这对于等势过程没有问题，客户只需一直传送其请求，直到接收到一个有效的响应为止。然而如果客户非得不止一次地发送其请求以接收一个有效的响应，那么该过程执行一次以上的可能是存在的。

对于一个本地过程调用，如果它返回，我们就知道它正好执行了一次，但是如果当前进程在调用该过程后崩溃，我们就不知道它到底执行了一次还是根本没有执行。对于一个远程过程调用，我们必须考虑如下各种情形。

- 如果使用的是TCP协议，而且接收到了一个应答，我们就知道该远程过程正好被调用了一次。但是如果没有接收到应答（譬如说服务器主机崩溃了），我们就不知道该服务器过程已在其主机崩溃之前执行完毕，还是尚未被调用（最多一次的语义）。在服务器主机可能崩溃，而且网络存在停止运作可能的前提下，提供正好一次的语义需要一个事务处理系统，它已超出了RPC软件包的能力。
- 如果使用的是UDP协议，而且服务器主机没有崩溃，应答也接收到了，我们就知道该服务器过程至少被调用了一次，不过也可能是多次（最少一次语义）。
- 如果使用的是UDP协议，而且启用了一个服务器高速缓存，应答也接收到了，我们就知道该服务器过程正好被调用了一次。然而如果没有接收到应答，那就具有最多一次的语义，这跟TCP情形类似。

给定如下三种选择：

(1) TCP；

(2) UDP，带有一个服务器高速缓存；

(3) UDP，没有任何服务器高速缓存。

我们的建议如下所述。

- 总是使用TCP，除非TCP连接的开销对于应用来说过分昂贵。
- 给正确执行的意义相当重大的非等势过程（例如银行账户、机票预订，等等）使用一个事务处理系统。
- 对于非等势过程，使用TCP要比使用带有一个服务器高速缓存的UDP更为可取。TCP一开始就设计成可靠的，而往一个UDP应用中添加可靠性很少能达到使用TCP的效果（例如UNPv1的20.5节）。 423
- 对等势过程使用不带服务器高速缓存的UDP不成问题。
- 对非等势过程使用不带服务器高速缓存的UDP则是危险的。

我们将在下一节讨论使用TCP的其他优势。

16.7 客户或服务器的过早终止

现在考虑客户或服务器之一过早终止，而且使用TCP作为传输协议时会发生什么情况。既然UDP是无连接的，因而打开着某个UDP端点的一个进程终止时，不会有任何信息发送给对方。在使用UDP的情形下，当有一方崩溃时所发生的全部情况为：对方将超时，可能会重传，最终放弃，这是上一节中讨论过的。然而，当具有某个打开着的TCP连接的一个进程终止时，该连接也终止，从而向对方发送一个FIN分节（UNPv1第36~37页），我们就想看一看当RPC运行时系

统在接收到来自对方的这个出乎意料的FIN时会做些什么。

16.7.1　服务器的过早终止

我们首先在服务器仍在处理一个客户请求时过早地终止它。对客户程序所作的唯一变动是：把图16-2中clnt_create调用的“tcp”参数挪走，改成作为一个命令行参数指定所用的传输协议，就像图16-12中的那样。在服务器过程中，我们增加一个abort函数调用。该调用会终止服务器进程，导致服务器主机的TCP向客户主机的TCP发送一个FIN，这一点可使用tcpdump验证。

我们首先对BSD/OS系统上的服务器运行Solaris系统上的客户程序：

```
solaris % client bsdi 22 tcp
bsdi: RPC: Unable to receive; An event requires attention
```

当客户主机接收到服务器主机的FIN时，其RPC运行时系统正在等待服务器的应答。它检测到这个非预期的应答后，从我们的squareproc_1调用返回一个错误。该错误（RPC_CANTRECV）由运行时系统保存在客户句柄中，（从我们的clnt_create包裹函数中）调用clnt_sperror将把该错误输出成“Unable to receive（无法接收）”。该出错消息的其余部分“Anevent requires attention（有一个事件需要留意）”对应于由运行时系统保存的XTI错误，它也由clnt_sperror输出。一个客户调用一个远程过程时能够返回大约30个不同的RPC_xxx错误，它们列在<rpc/clnt_star.h>头文件中。

对换客户和服务器的运行主机，我们看到同样的情形，由RPC运行时系统返回同样的错误（RPC_CANTRECV），不过最后输出的消息不一样。

```
bsdi % client solaris 11 tcp
solaris: RPC: Unable to receive; errno = Connection reset by peer
```

上面我们中止的Solaris服务器不是作为一个多线程化的服务器程序编译的，因此当我们调
424 用abort时，整个进程被终止。如果我们运行一个多线程化的服务器程序，事情就有变化，也就是说只有为客户的调用提供服务的线程才终止。为迫使这种情形出现，我们把abort调用替换成pthread_exit调用，就像我们在图15-25中对使用门的例子所做的那样。然后在BSD/OS系统上运行客户程序，在Solaris系统上运行多线程化的服务器程序。

```
bsdi % client solaris 33 tcp
solaris: RPC: Timed out
```

当服务器线程终止时，与客户的TCP连接并未关闭，也就是说它仍然在服务器进程中保持打开。因此，服务器主机没有给客户发送FIN，于是客户仅仅超时。在客户请求已发送给服务器，并且服务器主机的TCP已确认该请求后，如果服务器主机崩溃，那么我们将看到同样的情况。

16.7.2　客户的过早终止

当一个RPC客户在其使用TCP的某个RPC过程调用仍在进展期间终止时，客户主机的TCP将向服务器主机的TCP发送一个FIN。我们的问题是：服务器的RPC运行时系统是否检测到了这个条件，从而可能向服务器过程发出通知（回想15.11节，当客户过早终止时，门服务器线程被取消）。

为产生这个条件，我们的客户程序在调用服务器过程的紧前调用alarm(3)，我们的服务器过程则调用sleep(6)。（图15-30和图15-31中使用门的例子就是这么做的。由于客户没有捕获SIGALRM，其进程将在服务器的应答返回前约3秒钟时由内核终止。）我们在BSD/OS系统上运行客户程序，在Solaris系统上运行服务器程序。

```
bsdi % client solaris 44 tcp
Alarm call
```

在客户方发生的情况是我们预期的，但在服务器方没有发生任何特殊之事。服务器过程结束其6秒钟的睡眠后返回。用tcpdump查看所发生的情况，我们看到以下情况。

- 当客户终止时（在启动后约3秒钟时），客户主机的TCP向服务器主机的TCP发送一个FIN，服务器主机的TCP对它作了确认。按照TCP的术语，这个过程称为半关闭（half_close，TCPv1的18.5节）。
- 客户和服务器启动后约6秒钟时，服务器发送其应答，该应答由服务器主机的TCP发送给客户。（正如UNPv1第130~132页中讲述的那样，接收到一个FIN后通过同一TCP连接发送数据没有问题，因为TCP连接是全双工的。）客户主机的TCP响应以一个RST分节（复位），因为客户进程已经终止。服务器下一次在这个连接上读或写时将认识到这个现状，然而目前什么都不发生。

我们汇总一下本节探讨的几个关键点。

- 使用UDP的RPC客户和服务器永远不知道对方是否过早终止。当接收不到响应时，它们可能超时，不过无法分辨错误类型：进程过早终止、对方主机崩溃、网络不可达，等等。 425
- 使用TCP的客户和服务器检测出对方所存在问题的机会要大得多，因为对方进程的过早终止自动导致对方主机的TCP关闭其所在端的连接。但是如果对方是一个线程化的服务器，这一点就不起作用，因为对方线程的终止并不会关闭其所在端的连接。另外这一点也无助于检测对方主机的崩溃，因为发生这种情况时，对方主机的TCP并没关闭它的打开着的连接。为处理所有这些情形，超时机制仍然是必需的。

16.8 XDR：外部数据表示

使用前一章中讲述的门从一个进程调用另一个进程中的某个过程时，这两个进程处于同一台主机上，因而没有数据转换问题。但是对于不同主机间的RPC，各种各样的主机可能使用不同的数据格式。首先，各个基本的C数据类型可能有不同的大小（例如某些系统上long数据类型占据32位，其他系统上却占据64位）。其次，各个位真正的先后顺序可能不一样（也就是大端字节序和小端字节序的差异，我们在UNPv1第66~69页和第137~140页[①]中讨论过）。我们已随图16-3碰到过这个问题，那时我们在小端字节序的x86系统上运行服务器程序，在大端字节序的Sparc系统上运行客户程序，然而这样的两台主机之间仍能正确地交换长整数。

Sun RPC使用XDR即外部数据表示（External Data Representation）标准来描述和编码数据（RFC 1832 [Srinivasan 1995b]）。XDR既是一种用于描述数据的语言，又是一组用于编码数据的规则。XDR使用隐式类型指定（implicit typing）方式，它意味着发送者和接收者都得知道数据的类型和字节序：例如两个32位整数值后跟一个单精度浮点数值，再跟一个字符串。

> 作为比较，在OSI领域中ASN.1（抽象语法表示1，Abstract Syntax Notation one）是描述数据的通常方式，BER（基本编码规则，Basic Encoding Rules）是一种编码数据的常用方式。这种方案还使用显式类型指定（explicit typing）方式，它意味着每个数据值之前冠以描述所跟数据之类型的某个值（称为“指定符（specifier）”）。对应刚才这个例子的字节流按顺序含有以下各个字段：说下一个值是一个整数的指定符、整数值、说下一个值是一个整数的指定符、整数值、说下一个值是一个浮点数的指定符、浮点数值、说下一个值是一个字符串的指定符、字符串。

① 此处为UNPv1第2版英文原版书页码，第3版为第77~80页和第147~150页。——编者注

所有数据类型的XDR表示都需要4的倍数的字节数，这些字节总是以大端字节序传送的。带符号整数值使用二进制补码（two's complement）记法存放，浮点数值则使用IEEE格式存放。可变长度字段总是在其末端含有最多3个字节的填充，这样下一个条目总是落在某个4字节的边界。例如一个5字节的ASCII字符串将作为12个字节来传送：

- 一个4字节的整数计数，其值为5；
- 5字节的字符串本身；
- 3个字节的值为0的填充。

426 在讲述XDR和它支持的数据类型时，我们考虑以下三个问题。

(1) 如何在RPC说明书文件（.x文件）中给rpcgen声明各种类型的变量？到此为止的唯一一个例子（图16-1）只使用一个长整数。

(2) rpcgen把定义在.x文件中的变量转换成自己产生的.h头文件中的哪一种C数据类型？

(3) 所传送数据的真正格式是什么？

图16-14回答了前两个问题。为产生这张表格，我们创建了一个RPC说明书文件，它用到了所有受支持的XDR数据类型。该文件通过rpcgen运行后，我们查看所产生的C头文件从而构造出该表格。

	RPC说明书文件（.x）	C头文件（.h）
1	`const name = value;`	`#define name value`
2	`typedef declaration;`	`typedef declaration;`
3	`char var;` `short var;` `int var;` `long var;` `hyper var;`	`char var;` `short var;` `int var;` `long var;` `longlong_t var;`
4	`unsigned char var;` `unsigned short var;` `unsigned int var;` `unsigned long var;` `unsigned hyper var;`	`u_char var;` `u_short var;` `u_int var;` `u_long var;` `u_longlong_t var;`
5	`float var;` `double var;` `quadruple var;`	`float var;` `double var;` `quadruple var;`
6	`bool var;`	`bool_t var;`
7	`enum var { name = const, ... };`	`enum var { name = const, ... };` `typedef enum var var;`
8	`opaque var[n];`	`char var[n];`
9	`opaque var<m>;`	`struct {` `u_int var_len;` `char *var_val;` `} var;`
10	`string var<m>;`	`char *var;`
11	`datatype var[n];`	`datatype var[n];`
12	`datatype var<m>;`	`struct {` `u_int var_len;` `datatype *var_val;` `} var;`
13	`struct var { members ... };`	`struct var { members ... };` `typedef struct var var;`
14	`union var switch (int disc) {` `case discvalueA: armdeclA;` `case discvalueB: armdeclB;` `...` `default: defaultdecl;` `};`	`struct var {` `int disc;` `union {` `armdeclA;` `armdeclB;` `...` `defaultdecl;` `} var_u;` `};` `typedef struct var var;`
15	`datatype *name;`	`datatype *name;`

图16-14　XDR和rpcgen支持的数据类型的汇总

我们现在详细描述各个表项，并以第一栏中给出的顺序号（1~15）指称它们。

(1) const声明转换成C的#define。

(2) typedef声明转换成C的typedef。

(3) 这些是总共5个的带符号整数数据类型。其中前4个是由XDR作为32位值传送的，最后一个是由XDR作为64位值传送的。

> 对于许多C编译器来说，64位整数的类型为long long int或long long。然而不是所有的编译器和操作系统都支持它们。既然所生成的.h文件声明这样的C变量的类型为longlong_t，因此某个头文件中必须有如下的定义：
>
> ```
> typedef  long long  longlong_t;
> ```
>
> XDR的long类型占据32位，但是64位Unix系统上的C语言long类型占据64位（例如UNPv1第27页[①]描述的LP64模型）。这些10年前的XDR名字在当今的世界中确实是不幸的。更好的名字可能是int8_t、int16_t、int32_t、int64_t等等。

(4) 这些是总共5个的无符号整数数据类型。其中前4个是由XDR作为32位值传送的，最后一个是由XDR作为64位值传送的。

(5) 这些是总共3个的浮点数数据类型。其中第一个作为32位值传送，第二个作为64位值传送，第三个作为128位值传送。

> 四精度浮点数在C语言中的类型为long double。然而不是所有的编译器和操作系统都支持它们。（你的编译器也许允许long double，但只是把它作为double类型处理。）由于所生成的.h文件声明这样的C变量的类型为quadruple，因此某个头文件中必须有如下的定义：
>
> ```
> typedef  long double  quadruple;
> ```
>
> 举例来说，Solaris 2.6下我们必须在.x文件的开始处包含如下的行：
>
> ```
> %#include <floatingpoint.h>
> ```
>
> 因为该头文件包含了所需的定义。该行开头的百分号告诉rpcgen把本行剩余部分原封不动地放入所产生的.h头文件中。

427 ~ 428

(6) 布尔（boolean）数据类型与一个带符号整数等效。RPC头文件同时定义常值TRUE为1，常值FALSE为0。

(7) 一个枚举（enumeration）数据类型与一个带符号整数等效，且跟C语言的enum数据类型一样。rpcgen还给所指定的变量名产生一个typedef定义。

(8) 固定长度不透明数据（fixed-length opaque data）是作为8位值传送的确定数目（*n*）的字节，运行时函数库不解释它。

(9) 可变长度不透明数据（variable-length opaque data）也是作为8位值传送的不作解释的一个字节序列，不过真正的字节数是作为一个无符号整数先于数据传送的。发送这种类型的数据时（例如先于某个RPC调用填写传递给它的参数时），应在发出调用之前设置长度。接收这种类型的数据时，必须检查其长度以确定后跟多少数据。

声明中的最大长度*m*可被忽略。但是如果编译时指定了长度，那么运行时函数库将检查真正的长度（我们作为相应C结构的*var*_len成员给出的内容）没有超过*m*的值。

(10) 一个字符串（string）是一个ASCII字符序列。字符串在内存中是作为一个普通的以空

① 此处为UNPv1第2版英文原版书页码，第3版为第28~29页。——编者注

字符结尾的C字符串存放的，但在传输中却冠以一个指定后跟字符实际数目（不包括结尾的空字符）的无符号整数。发送这种类型的数据时，运行时系统通过调用strlen确定字符数。接收这种类型的数据时，它是作为一个以空字符结尾的C字符串存放的。

声明中的最大长度*m*可被忽略。但是如果编译时指定了长度，那么运行时函数库将检查真正的长度没有超过*m*的值。

(11) 一个任意数据类型的固定长度数组（fixed-length array）是作为该数据类型的一个*n*个元素的序列传送的。

(12) 一个任意数据类型的可变长度数组（variable-length array）作为指定该数组中实际元素数目的一个无符号整数以及后跟的各个数组元素传送。

声明中的最大长度*m*可被忽略。但是如果编译时指定了长度，那么运行时函数库将检查真正的长度没有超过*m*的值。

(13) 一个结构（structure）是通过轮流传送其各个成员来传送的。rpcgen还给所指定的变量名产生一个typedef定义。

(14) 一个带判别式的联合（discriminated union）由一个整数判别式后跟基于该判别式的值的一组数据类型（称为分支（arm））构成。在图16-14中给出的判别式是一个int，但它也可以是一个unsigned int、一个enum或一个bool（所有这些判别式都作为一个32位整数值传送）。传送一个带判别式的联合时，其判别式的32位值首先传送，然后传送对应于该判别式的值的唯一一个分支值。这种联合的default声明往往是void，它的意思是在判别式的32位值之后不跟任何数据。我们稍后给出这样的一个例子。

(15) 可选数据（optional data）是一种特殊类型的联合，我们将随图16-24中给出的一个例子描述它。这种数据类型的XDR声明看着像是一个C指针声明，它就是所生成的.h文件所包含的内容。

图16-15汇总了XDR给它的各种数据类型采用的编码格式。

16.8.1　例子：不涉及 RPC 使用 XDR

现在给出一个不涉及RPC使用XDR的例子。也就是说，我们将使用XDR把一个二进制数据的结构编码成一种可在其他系统上加以处理的机器无关表示。这种技巧可用于以一种与机器无关的格式书写文件，或者以一种与机器无关的格式通过网络向另一台计算机发送数据。图16-16给出了我们的RPC说明书文件data.x，它实际上只是一个XDR说明书文件，因为我们没有声明任何RPC过程。

429～431

文件名后缀.x来自“XDR说明书文件”一词。RPC规范（RFC 1831）中说，有时称为RPCL的RPC语言与XDR语言（在RFC 1832中定义）基本相同，差别只是后者增加了程序定义（用于描述程序、版本和过程）。

声明枚举和带判别式的联合

1~11　声明一个共有两个值的枚举数据类型，后跟一个把该枚举类型作为判别式的带判别式联合。如果该判别式的值为RESULT_INT，那么在该判别式的值之后传送的是一个整数值。如果该判别式的值为RESULT_DOUBLE，那么在该判别式的值之后传送的是一个双精度浮点数值。否则的话，在该判别式的值之后不传送任何数据。

声明结构

12~21　声明一个包含多个XDR数据类型的结构。

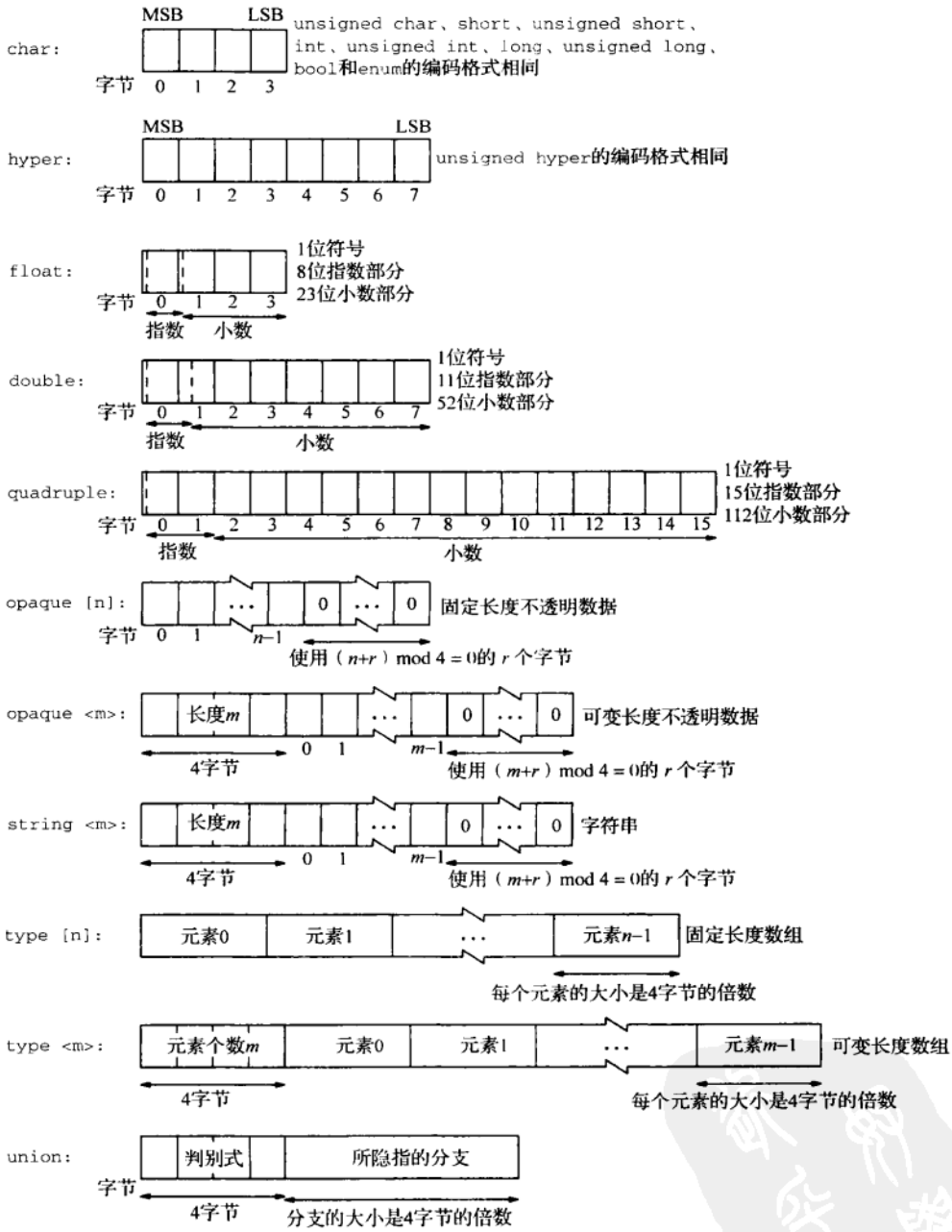


图16-15 XDR给它的各种数据类型采用的编码格式

sunrpc/xdr1/data.x

```
1 enum result_t {
2   RESULT_INT = 1, RESULT_DOUBLE = 2
3 };

4 union union_arg switch (result_t result) {
5 case RESULT_INT:
6     int     intval;
7 case RESULT_DOUBLE:
8     double  doubleval;
9 default:
10    void;
11 };

12 struct data {
13   short     short_arg;
14   long      long_arg;

15   string    vstring_arg<128>;      /* variable-length string */
16   opaque    fopaque_arg[3];    /* fixed-length opaque */
17   opaque    vopaque_arg<>;     /* variable-length opaque */
18   short     fshort_arg[4];     /* fixed-length array */
19   long      vlong_arg<>;       /* variable-length array */
20   union_arg uarg;
21 };
```

sunrpc/xdr1/data.x

图16-16　XDR说明书文件

既然data.x没有声明任何RPC过程，当查看图16-4中由rpcgen产生的所有文件时，我们看到rpcgen并没有产生客户程序存根和服务器程序存根。不过它仍然产生了data.h头文件和data_xdr.c文件，其中data_xdr.c含有用于编码或解码在我们的data.x文件中所声明数据条目的XDR函数。

图16-17给出了所产生的data.h头文件。按照图16-14中给出的转换规则，该头文件的内容正是我们所预期的。

sunrpc/xdr1/data.h

```
1 /*
2  * Please do not edit this file.  It was generated using rpcgen.
3  */

4 #ifndef _DATA_H_RPCGEN
5 #define _DATA_H_RPCGEN

6 enum result_t {
7     RESULT_INT = 1,
8     RESULT_DOUBLE = 2
9 };
10 typedef enum result_t result_t;

11 struct union_arg {
12     result_t result;
13     union {
14         int    intval;
15         double doubleval;
16     } union_arg_u;
17 };
18 typedef struct union_arg union_arg;

19 struct data {
```

图16-17　由rpcgen从图16-16产生的头文件

```
20      short short_arg;
21      long  long_arg;
22      char *vstring_arg;
23      char fopaque_arg[3];
24      struct {
25          u_int vopaque_arg_len;
26          char *vopaque_arg_val;
27      } vopaque_arg;
28      short   fshort_arg[4];
29      struct {
30          u_int   vlong_arg_len;
31          long   *vlong_arg_val;
32      } vlong_arg;
33      union_arg uarg;
34  };
35  typedef struct data data;

36          /* the xdr functions */
37  extern  bool_t xdr_result_t(XDR *, result_t*);
38  extern  bool_t xdr_union_arg(XDR *, union_arg*);
39  extern  bool_t xdr_data(XDR *, data*);

40  #endif /* !_DATA_H_RPCGEN */
```

sunrpc/xdr1/data.h

图16-17（续）

在data_xdr.c文件中定义了一个名为xdr_data的函数，我们可调用它来编码或解码已定义的data结构的内容。（函数名后缀_data来自图16-16中所定义结构的名字。）我们编写的第一个程序的文件名为write.c，它设置data结构中所有变量的值，调用xdr_data函数把所有字段编码成XDR格式，然后把结果写往标准输出。

图16-18给出了这个程序。

把结构成员设置成某个非零值

12~32 首先把data结构的所有成员设置成某个非零值。对于可变长度成员，我们必须设置一个计数以及这个数目的值。对于带判别式的联合，我们将判别式的值设置为RESULT_INT，对应的结果整数值为123。

分配适当地对齐的缓冲区

33 调用malloc分配XDR例程将往其中存入数据的缓冲区空间。由于该缓冲区必须在某个4字节的边界上对齐，因此简单地静态分配一个char数组不能保证这种对齐要求。

创建XDR内存流

34 运行时库函数xdrmem_create把由buff指向的缓冲区初始化成供XDR用作一个内存流。我们分配一个名为xhandle的XDR类型的变量，并把该变量的地址作为第一个参数传递给该函数。XDR运行时系统在这个变量中维护相关信息（缓冲区指针、缓冲区中的当前位置，等等）。最后一个参数是XDR_ENCODE，它告诉XDR我们需从主机格式（我们的out结构）转换成XDR格式。

编码结构

35~36 调用由rpcgen在data_xdr.c文件中生成的xdr_data函数，它把out结构编码成XDR格式。返回值为TRUE表示成功。

sunrpc/xdr1/write.c

```
 1 #include    "unpipc.h"
 2 #include    "data.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     XDR     xhandle;
 7     data    out;                /* the structure whose values we store */
 8     char   *buff;               /* the result of the XDR encoding */
 9     char    vop[2];
10     long    vlong[3];
11     u_int   size;

12     out.short_arg = 1;
13     out.long_arg = 2;
14     out.vstring_arg = "hello, world";  /* pointer assignment */

15     out.fopaque_arg[0] = 99;     /* fixed-length opaque */
16     out.fopaque_arg[1] = 88;
17     out.fopaque_arg[2] = 77;

18     vop[0] = 33;                 /* variable-length opaque */
19     vop[1] = 44;
20     out.vopaque_arg.vopaque_arg_len = 2;
21     out.vopaque_arg.vopaque_arg_val = vop;

22     out.fshort_arg[0] = 9999;    /* fixed-length array */
23     out.fshort_arg[1] = 8888;
24     out.fshort_arg[2] = 7777;
25     out.fshort_arg[3] = 6666;

26     vlong[0] = 123456;           /* variable-length array */
27     vlong[1] = 234567;
28     vlong[2] = 345678;
29     out.vlong_arg.vlong_arg_len = 3;
30     out.vlong_arg.vlong_arg_val = vlong;

31     out.uarg.result = RESULT_INT; /* discriminated union */
32     out.uarg.union_arg_u.intval = 123;

33     buff = Malloc(BUFFSIZE);    /* must be aligned on 4-byte boundary */
34     xdrmem_create(&xhandle, buff, BUFFSIZE, XDR_ENCODE);

35     if (xdr_data(&xhandle, &out) != TRUE)
36         err_quit("xdr_data error");
37     size = xdr_getpos(&xhandle);
38     Write(STDOUT_FILENO, buff, size);

39     exit(0);
40 }
```

sunrpc/xdr1/write.c

图16-18　初始化data结构并以XDR格式将它写出

获取编码后数据的大小并write

37~38　函数xdr_getpos返回XDR运行时系统在输出缓冲区中的当前位置(也就是待存入的下一个字节的字节偏移)，我们用它作为write调用的长度参数。

图16-19给出了我们的read程序，它读入由前一个程序写出的文件，输出data结构所有成员的值。

sunrpc/xdr1/read.c

```
 1 #include    "unpipc.h"
 2 #include    "data.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     XDR     xhandle;
 7     int     i;
 8     char   *buff;
 9     data    in;
10     ssize_t n;

11     buff = Malloc(BUFFSIZE);     /* must be aligned on 4-byte boundary */
12     n = Read(STDIN_FILENO, buff, BUFFSIZE);
13     printf("read %ld bytes\n", (long) n);

14     xdrmem_create(&xhandle, buff, n, XDR_DECODE);
15     memset(&in, 0, sizeof(in));
16     if (xdr_data(&xhandle, &in) != TRUE)
17         err_quit("xdr_data error");

18     printf("short_arg = %d, long_arg = %ld, vstring_arg = '%s'\n",
19            in.short_arg, in.long_arg, in.vstring_arg);

20     printf("fopaque[] = %d, %d, %d\n",
21            in.fopaque_arg[0], in.fopaque_arg[1], in.fopaque_arg[2]);

22     printf("vopaque<> =");
23     for (i = 0; i < in.vopaque_arg.vopaque_arg_len; i++)
24         printf(" %d", in.vopaque_arg.vopaque_arg_val[i]);
25     printf("\n");

26     printf("fshort_arg[] = %d, %d, %d, %d\n", in.fshort_arg[0],
27            in.fshort_arg[1], in.fshort_arg[2], in.fshort_arg[3]);

28     printf("vlong<> =");
29     for (i = 0; i < in.vlong_arg.vlong_arg_len; i++)
30         printf(" %ld", in.vlong_arg.vlong_arg_val[i]);
31     printf("\n");

32     switch (in.uarg.result) {
33     case RESULT_INT:
34         printf("uarg (int) = %d\n", in.uarg.union_arg_u.intval);
35         break;
36     case RESULT_DOUBLE:
37         printf("uarg (double) = %g\n", in.uarg.union_arg_u.doubleval);
38         break;
39     default:
40         printf("uarg (void)\n");
41         break;
42     }

43     xdr_free(xdr_data, (char *) &in);

44     exit(0);
45 }
```

sunrpc/xdr1/read.c

图16-19 读入XDR格式的data结构并输出其值

分配适当对齐过的缓冲区

11~13 调用malloc分配一个适当对齐过的缓冲区，把由前一个程序生成的文件读入该缓冲区。

创建XDR内存流，初始化缓冲区，然后解码

14~17　初始化一个XDR内存流，这次指定XDR_DECODE以指示我们希望从XDR格式转换成主机格式。把我们的in结构初始化为0后调用xdr_data，从而把缓冲区buff中的数据解码到in结构中。我们必须把XDR目的地（in结构）初始化为0，因为有些XDR例程（例如xdr_string）需要这样做。xdr_data与我们从图16-18中调用的同名函数是一样的，有变化的是xdrmem_create的最后一个参数：前一个程序中指定的是XDR_ENCODE，本程序中指定的是XDR_DECODE。该值由xdrmem_create保存在XDR句柄（xhandle）中，XDR运行时系统就用它来确定是编码数据还是解码数据。

输出结构的值

18~42　输出我们的data结构的所有成员的值。

释放由XDR分配的任何内存空间

43　调用xdr_free释放XDR运行时系统可能已动态分配的内存空间（参见习题16.10）。

我们在一台Sparc主机上运行write程序，并把标准输出重新定向到一个名为data的文件：

```
solaris % write > data
solaris % ls -l data
-rw-rw-r--   1 rstevens other1        76 Apr 23 12:32 data
```

我们看到文件大小为76字节，这跟图16-20是对应的，该图详细展示了数据的存放情况（19个4字节值）。

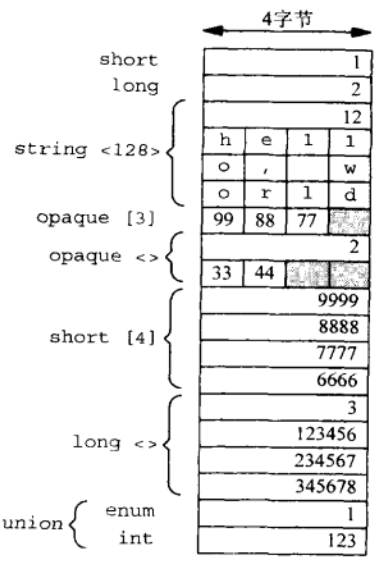


图16-20　由图16-18写出的XDR流的格式

在BSD/OS或Digital Unix下读这个二进制文件，结果跟我们预料的一致：

```
bsdi % read < data
read 76 bytes
short_arg = 1, long_arg = 2, vstring_arg = 'hello, world'
fopaque[] = 99, 88, 77
vopaque<> = 33 44
fshort_arg[] = 9999, 8888, 7777, 6666
```

```
vlong<> = 123456 234567 345678
uarg (int) = 123

alpha % read < data
read 76 bytes
short_arg = 1, long_arg = 2, vstring_arg = 'hello, world'
fopaque[] = 99, 88, 77
vopaque<> = 33 44
fshort_arg[] = 9999, 8888, 7777, 6666
vlong<> = 123456 234567 345678
uarg (int) = 123
```

16.8.2 例子：计算缓冲区的大小

在前一个例子中我们分配了一个长度为BUFFSIZE（该常值在图C-1给出的unpipc.h头文件中定义为8192）的缓冲区，它的大小足够了。不幸的是，没有一种简单方法来计算XDR为一个给定的结构编码所需的总大小。只计算该结构的sizeof值是不对的，因为该结构的每个成员由XDR分别编码。我们必须逐个成员地遍历这个结构，把XDR将用于编码各个成员的大小加在一起。举例来说，图16-21给出了一个有3个成员的结构。

434 / 437

sunrpc/xdr1/example.x

```
1 const   MAXC = 4;

2 struct example {
3     short   a;
4     double  b;
5     short   c[MAXC];
6 };
```

sunrpc/xdr1/example.x

图16-21 一个简单结构的XDR说明书文件

图16-22给出的程序计算出XDR编码这个结构所需的字节数为28。

sunrpc/xdr1/example.c

```
 1 #include    "unpipc.h"
 2 #include    "example.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     size;
 7     example foo;

 8     size = RNDUP(sizeof(foo.a)) + RNDUP(sizeof(foo.b)) +
 9         RNDUP(sizeof(foo.c[0])) * MAXC;
10     printf("size = %d\n", size);
11     exit(0);
12 }
```

sunrpc/xdr1/example.c

图16-22 计算XDR编码所需字节数的程序

8~9 宏RNDUP定义在<rpc/xdr.h>头文件中，它把它的参数向上舍入到下一个BYTES_PER_XDR_UNIT（4）的倍数。对于一个固定长度的数组，使用该宏计算出每个元素的大小后乘以元素数即可。

这种技巧的问题出在可变长度数据类型上。如果我们声明string d<10>，那么所需的最大字节数为RNDUP(sizeof(int))（用于存放长度）加上RNDUP(sizeof(char)*10)（用于存放字符）。但是我们无法计算不带最大值的可变长度数据类型的大小，例如float e<>。最简单的办

法是分配一个应该足够大的缓冲区，并检查XDR例程的失败情况（参见习题16.5）。

16.8.3　例子：可选数据

438 XDR说明书文件中有三种指定可选数据的方式，图16-23给出了所有这三种方式。

sunrpc/xdr1/opt1.x

```
1 union optlong switch (bool flag) {
2   case TRUE:
3       long   val;
4   case FALSE:
5       void;
6 };
7 struct args {
8     optlong arg1;              /* union with boolean discriminant */
9     long    arg2<1>;           /* variable-length array with one element */
10    long    *arg3;             /* pointer */
11 };
```

sunrpc/xdr1/opt1.x

图16-23　展示三种指定可选数据的方式的XDR说明书文件

声明一个带布尔型判别式的联合

1~8　定义一个带TRUE和FALSE两个分支的联合，并把某个结构成员定义为该类型。当判别式flag为TRUE时，后跟的是一个long类型的值，否则什么都不跟。XDR运行时系统编码该参数时，它将被编码成以下两种格式之一：

- 值为1（TRUE）的4字节标志后跟一个4字节的值；
- 值为0（FALSE）的4字节标志。

声明可变长度数组

9　指定一个最多一个元素的可变长度数组时，它将被编码成以下两种格式之一：

- 值为1的4字节长度后跟一个4字节的值；
- 值为0的4字节长度。

声明XDR指针

10　成员arg3展示了指定可选数据的一种新方式（它对应于图16-14中的最后一行）。该参数将被编码成以下两种格式之一：

- 值为1的4字节标志后跟一个4字节值；
- 值为0的4字节标志。

这具体取决于编码数据时相应的C指针的值。如果该指针非空，那就使用第一种编码格式（8个字节），否则使用第二种编码格式（4个字节的0）。当一个可选数据在代码中是通过指针来访问时，这是编码该数据的较为便利的方式。

使得前两个声明产生同样的编码格式的一个实现上的细节是TRUE的值为1，它恰好是只有一个元素的可变长度数组的长度。

图16-24给出了由rpcgen为这个说明书文件产生的.h文件。

439 14~21　尽管所有三个参数都将由XDR运行时系统编码成同样的格式，在C代码中设置和取出它们的值的方法却各不相同。

图16-25是一个简单的程序，它设置上述所有三个参数的值，使得其编码中没有一个long类型的值出现。

sunrpc/xdr1/opt1.h

```
 7 struct optlong {
 8     int     flag;
 9     union {
10         long    val;
11     } optlong_u;
12 };
13 typedef struct optlong optlong;

14 struct args {
15     optlong arg1;
16     struct {
17         u_int   arg2_len;
18         long   *arg2_val;
19     } arg2;
20     long   *arg3;
21 };
22 typedef struct args args;
```

sunrpc/xdr1/opt1.h

图16-24 由rpcgen给图16-23产生的C头文件

sunrpc/xdr1/opt1z.c

```
 1 #include    "unpipc.h"
 2 #include    "opt1.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int      i;
 7     XDR      xhandle;
 8     char    *buff;
 9     long    *lptr;
10     args     out;
11     size_t   size;

12     out.arg1.flag = FALSE;
13     out.arg2.arg2_len = 0;
14     out.arg3 = NULL;

15     buff = Malloc(BUFFSIZE);    /* must be aligned on 4-byte boundary */
16     xdrmem_create(&xhandle, buff, BUFFSIZE, XDR_ENCODE);

17     if (xdr_args(&xhandle, &out) != TRUE)
18         err_quit("xdr_args error");
19     size = xdr_getpos(&xhandle);

20     lptr = (long *) buff;
21     for (i = 0; i < size; i += 4)
22         printf("%ld\n", (long) ntohl(*lptr++));

23     exit(0);
24 }
```

sunrpc/xdr1/opt1z.c

图16-25 使三个参数都不编码long类型值的程序

设置各个值

12~14 把对应第一个参数的联合中的判别式设置为FALSE，把对应第二个参数的可变长度数组的长度设置为0，把对应第三个参数的指针设置为NULL。

分配适当对齐过的缓冲区并编码

15~19 分配一个缓冲区，把我们的out结构编码到一个XDR内存流中。

输出XDR缓冲区

20~22　使用ntohl函数（网络到主机长整数转换函数）把对应于该内存流的缓冲区中的数据从XDR使用的大端字节序转换成当前主机的字节序，然后每个4字节值一次地输出。该输出准确地展示了由XDR运行时系统编码到该缓冲区中的数据。

```
solaris % opt1z
0
0
0
```

正如我们预期的那样，每个参数作为值全为0的4个字节编码，指示后面不跟任何值。

图16-26是对前一个程序的修改，它给所有三个参数赋值，把它们编码到一个XDR内存流中，然后输出这个流。

sunrpc/xdr1/opt1.c

```
 1 #include    "unpipc.h"
 2 #include    "opt1.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int       i;
 7     XDR       xhandle;
 8     char     *buff;
 9     long      lval2, lval3, *lptr;
10     args      out;
11     size_t    size;

12     out.arg1.flag = TRUE;
13     out.arg1.optlong_u.val = 5;

14     lval2 = 9876;
15     out.arg2.arg2_len = 1;
16     out.arg2.arg2_val = &lval2;

17     lval3 = 123;
18     out.arg3 = &lval3;

19     buff = Malloc(BUFFSIZE);     /* must be aligned on 4-byte boundary */
20     xdrmem_create(&xhandle, buff, BUFFSIZE, XDR_ENCODE);

21     if (xdr_args(&xhandle, &out) != TRUE)
22         err_quit("xdr_args error");
23     size = xdr_getpos(&xhandle);

24     lptr = (long *) buff;
25     for (i = 0; i < size; i += 4)
26         printf("%ld\n", (long) ntohl(*lptr++));

27     exit(0);
28 }
```

sunrpc/xdr1/opt1.c

图16-26　给来自图16-23的所有三个参数赋值

设置各个值

12~18　为给对应第一个参数的联合赋一个值，我们把它的判别式设置成TRUE，然后设置它的值。为给对应第二个参数的可变长度数据赋一个值，我们把它的长度设置为1，并把与它关联的指针设置成指向它的值。为给对应第三个参数的指针赋一个值，我们把它设置成存放其值的变量的地址。

运行这个程序，输出的是预期的6个4字节值：

```
solaris % opt1
1                    判别式值为TRUE
5
1                    可变长度数组的长度
9876
1                    非空指针变量的标志
123
```

16.8.4 例子：链表处理

有了前面的例子所介绍的编码可选数据的能力后，我们就可以对XDR的指针表示进行扩充，用它来编码和解码含有可变数目元素的链表。我们的例子是一个名-值对（name-value pair）链表，图16-27给出了它的XDR说明书文件。

sunrpc/xdr1/opt2.x

```
1 struct mylist {
2     string  name<>;
3     long    value;
4     mylist *next;
5 };

6 struct args {
7     mylist *list;
8 };
```

sunrpc/xdr1/opt2.x

图16-27 名-值对链表的XDR说明书文件

1~5 我们的mylist结构含有一个名-值对和一个指向下一个结构的指针。该链表中的最后一个结构将有一个值为null的next指针。

图16-28给出了由rpcgen根据图16-27产生的.h文件。

sunrpc/xdr1/opt2.h

```
 7 struct mylist {
 8     char   *name;
 9     long    value;
10     struct mylist *next;
11 };
12 typedef struct mylist mylist;

13 struct args {
14     mylist *list;
15 };
16 typedef struct args args;
```

sunrpc/xdr1/opt2.h

图16-28 对应于图16-27的C声明

图16-29给出的程序先初始化一个含有3个名-值对的链表，然后调用XDR运行时系统对它进行编码。

初始化链表

11~22 分配4个链表项的空间，但只初始化其中3个。第一项为nameval[2]，第二项为nameval[1]，第三项为nameval[0]。链表的头（out.list）设置成&nameval[2]。以这样的顺序初始化该链表是为了展示XDR运行时系统依循指针规则，而且所编码的链表项顺序跟所用的数组元素没有关系。我们还把各个链表项的值初始化成十六进制值，因为长整数值是以十六进制输出的，这使得查看每个字节中的各个ASCII值更为容易。

sunrpc/xdr1/opt2.c

```
 1 #include    "unpipc.h"
 2 #include    "opt2.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     i;
 7     XDR     xhandle;
 8     long   *lptr;
 9     args    out;                    /* the structure that we fill */
10     char   *buff;                   /* the XDR encoded result */
11     mylist  nameval[4];             /* up to 4 list entries */
12     size_t  size;

13     out.list = &nameval[2];         /* [2] -> [1] -> [0] */
14     nameval[2].name = "name1";
15     nameval[2].value = 0x1111;
16     nameval[2].next = &nameval[1];
17     nameval[1].name = "namee2";
18     nameval[1].value = 0x2222;
19     nameval[1].next = &nameval[0];
20     nameval[0].name = "nameee3";
21     nameval[0].value = 0x3333;
22     nameval[0].next = NULL;

23     buff = Malloc(BUFFSIZE);     /* must be aligned on 4-byte boundary */
24     xdrmem_create(&xhandle, buff, BUFFSIZE, XDR_ENCODE);

25     if (xdr_args(&xhandle, &out) != TRUE)
26         err_quit("xdr_args error");
27     size = xdr_getpos(&xhandle);

28     lptr = (long *) buff;
29     for (i = 0; i < size; i += 4)
30         printf("%8lx\n", (long) ntohl(*lptr++));

31     exit(0);
32 }
```

sunrpc/xdr1/opt2.c

图16-29　初始化一个链表，对它编码后输出结果

程序的输出表明，前三个链表项之前有一个为1的4字节值（我们既可以认为它是一个可变长度数组值为1的长度，也可以认为它是布尔值TRUE），第四个表项仅由一个为0的4字节值构成，指示链表的结尾。

```
solaris % opt2
       1                    后跟一个元素
       5                    字符串长度
6e616d65                    n a m e
31000000                    1，3个填充字节
    1111                    相应的值
       1                    后跟一个元素
       6                    字符串长度
6e616d65                    n a m e
65320000                    e 2，2个填充字节
    2222                    相应的值
       1                    后跟一个元素
       7                    字符串长度
6e616d65                    n a m e
65653300                    e e 3，1个填充字节
```

```
3333                    相应的值
  0                     后面不跟元素：链表尾
```

当XDR给这种格式的链表解码时，它将给链表项和指针动态分配内存空间，并把各个指针链接起来，以允许我们在C中方便地遍历整个链表。

16.9 RPC 分组格式

图16-30展示了封装在一个TCP分节中的一个RPC请求的格式。

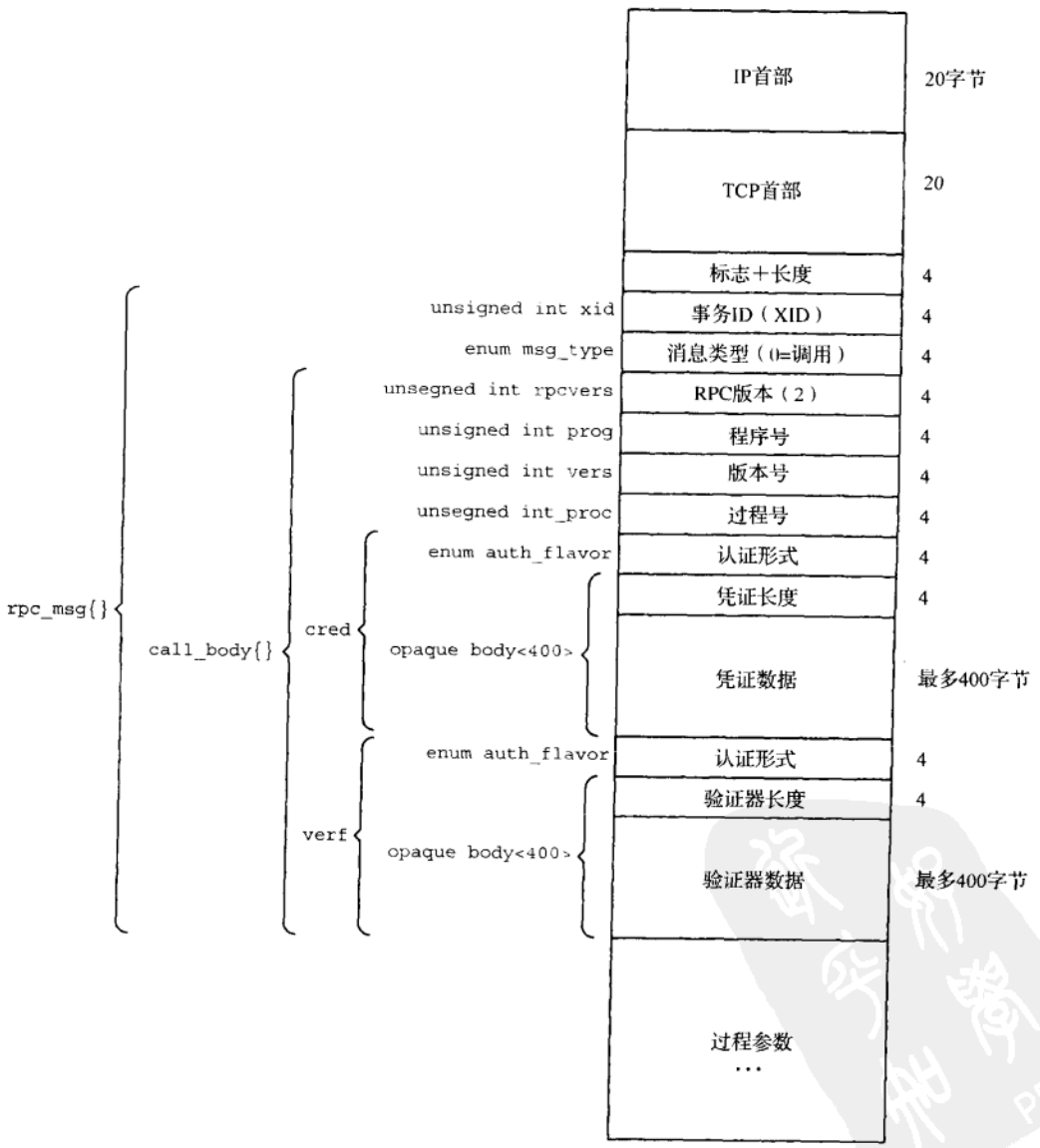


图16-30 封装在一个TCP分节中的RPC请求

既然TCP是一个字节流，不提供消息边界，因此应用程序必须提供界定各个消息的某种方法。Sun RPC定义了既可作为请求也可作为应答的记录（record），每个记录由一个或多个片段（fragment）构成。每个片段以一个4字节值开头：其中最高位是最终片段的标志，低序31位是计数。如果最终片段标志位为0，那么构成当前记录的还有别的片段。

这个4字节值跟所有的4字节XDR整数一样，是以大端字节序传送的，但是本字段却不在标准的XDR格式中，因为XDR并不传送位字段。

如果所用的是UDP而不是TCP，那么紧跟在UDP首部之后的第一个字段是XID，如图16-32（见后页）所示。

440
~
445

使用TCP时，RPC请求和应答的大小几乎不存在限制，因为可使用任意数目的片段，而每个片段又有一个31位的长度字段。然而使用UDP时，请求和应答都必须适合单个UDP数据报，而一个数据报能容纳的最大数据量是65507字节（假设网络协议为IPv4）。先于TI-RPC软件包的许多实现还把请求或应答的大小进一步限制到8192字节左右，因此如果请求或应答需要多于约8000字节的话，那就得改用TCP。

我们现在给出取自RFC 1831的一个RPC请求的真正XDR说明书。图16-30中给出的名字就出自该说明书。

```
enum auth_flavor {
  AUTH_NONE  = 0;
  AUTH_SYS   = 1;
  AUTH_SHORT = 2
    /* and more to be defined */
};

struct opaque_auth {
   auth_flavor  flavor;
   opaque       body<400>;
};

enum msg_type {
   CALL  = 0;
   RELAY = 1
};
struct call_body {
   unsigned intrpcvers;     /* RPC version: must be 2 */
   unsigned intprog;        /* program number */
   unsigned intvers;        /* version number */
   unsigned intproc;        /* procedure number */
   opaque_auth cred;        /* caller's credentials */
   opaque_auth verf;        /* caller's verifier */
     /* procedure-specific parameters start here */
};

struct rpc_msg {
   unsigned int  xid;
   union switch (msg_type  mtype) {
   case CALL:
      call_body   cbody;
   case REPLY:
      reply_body  rbody;
   } body;
};
```

含有凭证和验证器的可变长度不透明数据的内容取决于认证形式。对于空认证（默认形式）来说，该不透明数据的长度应为0。对于Unix认证来说，该不透明数据含有如下信息：

```
struct authsys_parms {
  unsigned int  stamp;
  string        machinename<255>;
  unsigned int  uid;
  unsigned int  gid;
  unsigned int  gids<16>;
};
```

446

当凭证的认证形式为AUTH_SYS时，验证器的认证形式应为AUTH_NONE。

RPC应答的格式比请求的格式复杂，因为请求中可能发生错误。图16-31展示了RPC应答的各种可能。

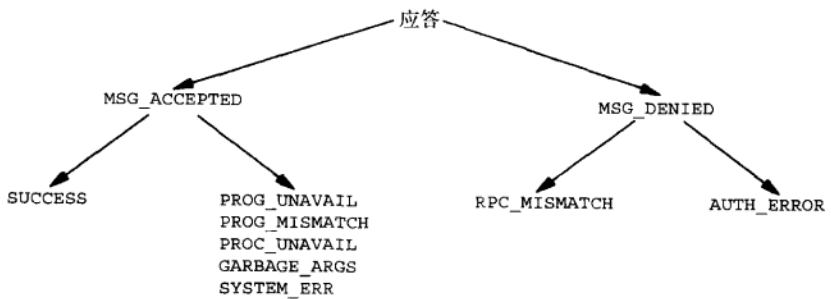


图16-31 可能的RPC应答

图16-32展示了一个成功的RPC应答的格式，不过这一次封装在一个UDP数据报中。

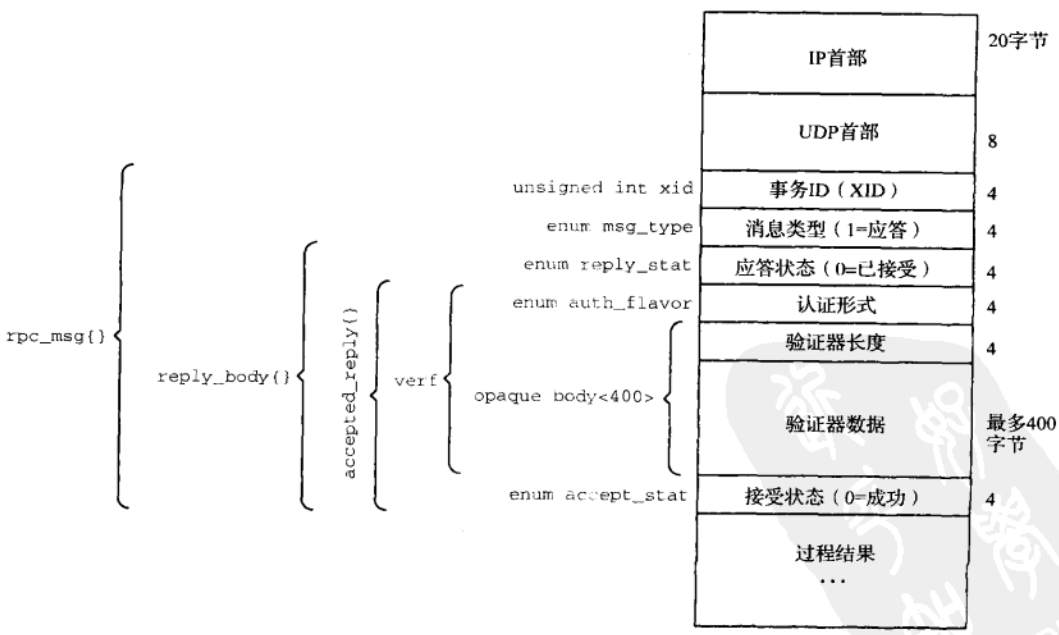


图16-32 封装为一个UDP数据报的成功的RPC应答

我们现在给出取自RFC 1831的一个RPC应答的真正XDR说明书。

```
enum reply_stat {
  MSG_ACCEPTED = 0;
  MSG_DENIED   = 0
};

enum accept_stat {
  SUCCESS       = 0,    /* RPC executed successfully */
  PROG_UNAVAIL  = 1,    /* program # unavailable */
  PROG_MISMATCH = 2,    /* version # unavailable */
  PROC_UNAVAIL  = 3,    /* procedure # unavailable */
  GARBAGE_ARGS  = 4,    /* cannot decode arguments */
  SYSTEM_ERR    = 5     /* memory allocation failure, etc. */
};

struct accepted_reply {
  opaque_auth  verf;
  union switch (accept_stat  stat) {
  case SUCCESS:
    opaque  results[0];     /* procedure-specific results start here */
  case PROG_MISMATCH:
    struct {
      unsigned int  low;    /* lowest version # supported */
      unsigned int  high;   /* highest version # supported */
    } mismatch_info;
  default: /* PROG_UNAVAIL, PROC_UNAVAIL, GARBAGE_ARGS, SYSTEM_ERR */
    void;
  } reply_data;
};

union reply_body switch (reply_stat  stat) {
  case MSG_ACCEPTED:
    accepted_reply  areply;
  case MSG_DENIED:
    rejected_reply  rreply;
} reply;
```

如果RPC版本号有误，或者发生认证错误，服务器就可能拒绝调用请求。

```
enum reject_stat {
  RPC_MISMATCH = 0,     /* RPC version number not 2 */
  AUTH_ERROR   = 1      /* authentication error */
};

enum auth_stat {
  AUTH_OK           = 0,    /* success */
       /* following are failures at server end */
  AUTH_BADCRED      = 1,    /* bad credential (seal broken) */
  AUTH_REJECTEDCRED = 2,    /* client must begin new session */
  AUTH_BADVERF      = 3,    /* bad verifier (seal broken) */
  AUTH_REJECTEDVERF = 4,    /* verifier expired or replayed */
  AUTH_TOOWEAK      = 5,    /* rejected for security reasons */
       /* following are failures at client end */
  AUTH_INVALIDRESP  = 6,    /* bogus response verifier */
  AUTH_FAILED       = 7     /* reason unknown */
};

union rejected_reply switch (reject_stat  stat) {
  case RPC_MISMATCH:
```

```
        struct {
          unsigned int  low;    /* lowest RPC version # supported */
          unsigned int  high;   /* highest RPC version # supported */
        } mismatch_info;
      case AUTH_ERROR:
        auth_stat  stat;
    };
```

16.10 小结

Sun RPC允许我们编写分布式应用程序，让客户运行在一台主机上，服务器运行在另一台主机上。我们首先定义了客户能够调用的服务器过程，然后编写了一个描述这些过程的参数和返回值的RPC说明书文件。我们接着编写了调用服务器过程的客户程序main函数以及服务器过程本身。客户程序的代码看起来只是简单地调用服务器过程，但在其背后，各种各样的RPC运行时例程隐藏了网络通信正在发生的事实。

rpcgen程序是构建使用RPC的应用程序的一个基本工具。它读入我们的说明书文件，产生客户程序存根和服务器程序存根，同时产生调用所需XDR运行时例程以处理所有数据转换的函数。XDR运行时系统也是构建使用RPC的应用程序过程中的一个基本部件。XDR定义了在不同的系统间交换各种数据格式的一种标准方法，这些系统可能具有不同的整数大小、不同的字节序、不同的浮点数格式，等等。正如我们所示的那样，XDR可独立于RPC软件包单独使用，其目的纯粹是为了以一种标准的格式交换数据，而数据的交换可以使用任意形式的真正传送数据的通信手段（例如使用套接字或XTI编写的程序、软盘、CD-ROM，等等）。

Sun RPC提供了自己的命名形式，这种命名使用32位程序号、32位版本号和32位过程号。运行一个RPC服务器的每台主机必须运行一个名为端口映射器（现在称为RPCBIND）的程序。RPC服务器捆绑临时的TCP和UDP端口后向端口映射器注册，从而把这些临时端口与由服务器提供的程序号和版本号关联起来。当一个RPC客户启动时，它首先跟服务器主机上的端口映射器联系以获取所需的端口号，然后跟服务器本身联系，通常情况下要么使用TCP，要么使用UDP。

默认情况下，RPC客户不提供任何认证信息，RPC服务器则处理所收到的任何客户请求。这跟我们使用套接字或XTI编写自己的客户-服务器程序一样。Sun RPC提供了另外三种认证形式：Unix认证（提供客户的主机名、用户ID和组ID）、DES认证（基于私钥和公钥加密技术）和Kerberos认证。

理解作为底层支撑的RPC软件包中的超时和重传策略对于使用RPC（或进行任何形式的网络编程）至关重要。当使用诸如TCP这样的可靠传输层时，RPC客户只需要一个总超时，因为任何丢失或重复的分组都是由传输层完全处理的。然而当使用诸如UDP这样的不可靠传输层时，除总超时外，RPC软件包还有一个重试超时。RPC客户使用事务ID来验证某个接收到的应答是所期望的应答。

任何过程调用可划归为具有正好一次语义、最多一次语义或最少一次语义。对于本地过程调用，我们通常忽略这些问题，但是对于RPC，我们必须清楚这几种语义的差异，并理解等势过程（能够不出问题地调用任意多次的过程）和非等势过程（必须只调用一次的过程）之间的差别。

Sun RPC是一个庞大的软件包，而我们只是触及了它的皮毛而已。不过有了本章中讨论的基本知识后，就能编写出完整的应用程序。rpcgen的使用隐藏了许多细节，并简化了代码编写

447 ~ 449

工作。Sun的手册中把使用RPC的代码编写工作划分成多个级别——简化的接口、顶级、中间级、专家级和底级，不过这样的划分毫无意义。RPC运行时系统总共提供164个函数，划分成以下六类：

- 11个auth_函数（认证）；
- 26个clnt_函数（客户方）；
- 5个pmap_函数（端口映射器访问）；
- 24个rpc_函数（一般性）；
- 44个svc_函数（服务器方）；
- 54个xdr函数（XDR转换）。

比较一下，套接字API和XTI API分别有约25个函数，门API、Posix和System V各自的消息队列API、信号量API和共享内存区API分别有少于10个的函数。处理Posix线程的函数有15个，处理Posix条件变量的函数有10个，处理Posix读写锁的函数有11个，处理fcntl记录上锁的函数只有1个。

450

习题

16.1　当启动某个RPC服务器时，它向端口映射器注册其自身。如果终止该服务器（譬如说使用终端中断键），那么它的注册会有什么变化？如果发往该服务器的某个客户请求在此后某个时刻到达，那会发生什么？

16.2　假设有一个基于UDP使用RPC的客户-服务器系统，它没有服务器应答高速缓存。客户发送一个请求给服务器，但是服务器将其应答发回要花20秒。客户在15秒后超时，导致服务器过程被再次调用。该服务器的第二个应答将发生什么？

16.3　XDR的string数据类型总是编码成在一个长度之后跟以其各个字符。如果我们想要定长的字符串，譬如说以char c[10]代替string s<10>，那么需做哪些变动？

16.4　把图16-16中string的最大大小由128改为10，然后运行write程序，发生了什么？现在把最大长度指定符从string声明中去掉，也就是说写成string vstring_arg<>，比较修改前后分别产生的data_xdr.c文件，有什么变化？

16.5　把图16-18中xdrmem_create的第三个参数（缓冲区大小）改为50，看发生了什么。

16.6　在16.5节中我们讲述了当使用UDP时可被启用的重复请求高速缓存。我们可以说TCP维护着自己的重复请求高速缓存。这具体指什么，另外这个TCP重复请求高速缓存有多大？（提示：TCP是如何检测收到重复数据的？）

16.7　给定唯一标识服务器重复请求高速缓存中每一项的那5个元素，当比较某个新的请求和高速缓存中各个项时，这5个值以怎样的顺序进行比较所需比较次数最少？

16.8　观察16.5节中使用TCP的客户-服务器系统的真正分组传递，可看到请求分节的大小为48字节，应答分节的大小为32字节（忽略IPv4首部和TCP首部）。分析这两个大小（例如如图16-30和图16-32那样）。如果改用UDP代替TCP，那么这两个大小是多少？

16.9　在不支持线程的系统上的RPC客户能调用编译成支持线程的服务器过程吗？我们在16.2节中叙述的调用参数上的差异情况怎么样？

16.10　在图16-19的read程序中，我们分配读入文件用的缓冲区空间，而该缓冲区中含有指针vstring_arg。请问由vstring_arg所指的字符串存放在哪儿？修改该程序以验证你的假设。

16.11　Sun RPC把空过程（null procedure）定义为过程号为0的过程（这就是我们总是以1开始过程编号的原因，如图16-1所示）。另外，由rpcgen生成的每个服务器存根自动定义该过程（这一点只需查看由本章中的例子生成的任何服务器存根就能轻而易举地验证）。空过程不需要参数，也不返回东西，往往用于验证给定服务器正在运行，或者测量到服务器的往返时间。然而要是我们查看客户存根，

那么会发现rpcgen并没有给空过程产生存根。查看clnt_call函数的手册页面，并用它来对本章中给出的任意一个服务器调用空过程。

451

16.12 图A-2中对于使用UDP的Sun RPC来说，为什么不存在消息大小为65536的项？图A-4中对于使用UDP的Sun RPC来说，为什么不存在消息大小为16384和32768的项？

16.13 验证在图16-19中省略xdr_free调用将引入内存空间泄漏（memory leak）。在调用xdrmem_create的紧前插入以下语句：

```
for ( ; ; ) {
```

再把配对的右花括弧放在调用xdr_free的紧前。运行该程序，使用ps观察它的内存空间大小。然后把配对的右花括弧改放到调用xdr_free之后，再运行该程序，观察它的内存空间大小。

452

后　记

本书详细讲述了用于进程间通信（IPC）的四种不同技术：

(1) 消息传递（管道、FIFO、Posix和System V消息队列）；

(2) 同步（互斥锁、条件变量、读写锁、文件和记录锁、Posix和System V信号量）；

(3) 共享内存区（匿名共享内存区、有名Posix共享内存区、有名System V共享内存区）；

(4) 过程调用（Solaris门、Sun RPC）。

消息队列和过程调用往往单独使用，也就是说它们通常提供了自己的同步机制。相反，共享内存区通常需要某种由应用程序提供的同步形式才能正确工作。同步技术有时候单独使用，也就是说不涉及其他形式的IPC。

讨论了共16章的细节后，很显然的一个问题是：解决某个特定问题应使用哪种形式的IPC？遗憾的是不存在关于IPC的简单判定。Unix提供的类型如此之多的IPC表明，不存在解决全部（或者甚至于大部分）问题的单一办法。你能做的仅仅是：逐渐熟悉各种IPC形式提供的机制，然后根据特定应用的要求比较它们的特性。

我们首先列出必须考虑的四个前提，因为它们对于你的应用程序相当重要。

(1) 连网的（networked）还是非连网的（nonnetworked）。我们假设已作出这个决定，IPC就是用于单台主机上的进程或线程间的。如果应用程序有可能散布到多台主机上，那就考虑使
453 用套接字代替IPC，从而简化以后向连网的应用程序转移的工作。

(2) 可移植性（portability）。回想图1-5，几乎所有Unix系统都支持Posix管道、Posix FIFO和Posix记录上锁。到了1998年，大多数Unix系统支持System V IPC（消息、信号量和共享内存区），而支持Posix IPC（消息、信号量和共享内存区）的仅有几个系统。Posix IPC应该出现更多实现，然而（遗憾的是）它在Unix 98中只是个选项。许多Unix系统支持Posix线程（包括互斥锁和条件变量在内），或者应在不久的将来支持它们。有些支持Posix线程的系统不支持互斥锁和条件变量的进程间共享属性。Unix 98需要的读写锁应被Posix所采纳，而且许多版本的Unix已在支持某种类型的读写锁。内存映射I/O得到广泛支持，大多数Unix系统还提供匿名内存映射（或者使用`/dev/zero`，或者使用`MAP_ANON`）。几乎所有Unix系统都可使用Sun RPC，门则是Solaris特有的特性（到目前为止是这样）。

(3) 性能（performance）。如果性能是应用程序设计中的一个关键前提，那就在你自己的系统上运行附录A中开发的程序。更好的做法是，把这些程序修改成模拟特定应用的实际环境，再在这样的环境中测量它们的性能。

(4) 实时调度（realtime scheduling）。如果你的应用需要这一特性，而且你的系统支持Posix实时调度选项，那就考虑使用Posix的消息传递和同步函数（消息队列、信号量、互斥锁、条件变量）。举例来说，当某个线程挂出一个有多个线程阻塞在其上的信号量时，待解阻塞的线程是以一种适合于所阻塞线程的调度策略和参数的方式选择的。相反，System V信号量不能保证实时调度。

为帮助理解各种类型IPC的一些特性和局限，我们汇总了它们的一些主要差异。

- 管道和FIFO是字节流，没有消息边界。Posix消息和System V消息则有从发送者向接收者维护的记录边界。（考虑到UNPv1中讲述的网际协议族，TCP是没有记录边界的字节流，UDP则提供具有记录边界的消息。）
- 当有一个消息放置到一个空队列中时，Posix消息队列可向一个进程发送一个信号，或者启动一个新的线程。System V消息队列不提供类似的通知形式。这两种消息队列都不能直接跟`select`或`poll`（UNPv1的第6章）一起使用，不过我们分别在图5-14和6.9节中提供了间接的方法。
- 管道或FIFO中的数据字节是先进先出的。Posix消息和System V消息具备由发送者赋予的优先级。从一个Posix消息队列读出消息时，首先返回的总是具有最高优先级的消息。从一个System V消息队列读出时，读出者可以请求所想要的任意优先级的消息。 454
- 当有一个消息放置到一个Posix或System V消息队列，或者写到一个管道或FIFO时，只有一个副本递交给刚好一个线程。这些IPC形式不存在窥探能力（即类似于套接字API的`MSG_PEEK`标志，UNPv1的13.7节[①]），它们的消息也不能广播或多播到多个接收者（这对于使用UDP协议的套接字程序和XTI程序是可能的，UNPv1第18章和第19章）。
- 互斥锁、条件变量和读写锁都是无名的，也就是说它们是基于内存的。它们能够很容易地在单个进程内的不同线程间共享。然而只有当它们存放在不同进程间共享的内存区中时，它们才可能为这些进程所共享。而Posix信号量就有两种形式：有名的和基于内存的。有名信号量总能在不同进程间共享（因为它们是用Posix IPC名字标识的），基于内存的信号量也能在不同进程间共享，条件是必须存放在这些进程间共享的内存区中。System V信号量也是有名的，不过所用的是`key_t`数据类型，它往往是从某个文件的路径名获取的。这些信号量能够很容易地在不同进程间共享。
- 如果持有某个锁的进程没有释放它就终止，内核就自动释放`fcntl`记录锁。System V信号量将这一特性作为一个选项提供。互斥锁、条件变量、读写锁和Posix信号量不具备该特性。
- 每个`fcntl`锁都与通过其相应描述符访问的文件中的某个字节范围（我们称之为一个“记录”）相关联。读写锁则不与任何类型的记录关联。
- Posix共享内存区和System V共享内存区都具有随内核的持续性。它们一直存在到被显式地删除为止，即使当前没有任何进程在使用它们也这样。
- Posix共享内存区对象的大小可在其使用期间扩张。System V共享内存区的大小则是在创建时固定下来的。
- System V IPC所存在的三种内核限制往往需要系统管理员对它们进行调整，因为它们的默认值通常不能满足现实应用的需要（3.8节）。Posix IPC所存在的三种内核限制则通常根本不需要调整。
- 有关System V IPC对象的信息（当前大小、属主ID、最后修改时间，等等）可使用三个*XXX*`ctl`函数的`IPC_STAT`命令获取，也可执行`ipcs`命令获取。有关Posix IPC对象的信息则不存在标准的获取方式。如果这些对象是用文件系统中的文件实现的，而且我们知道从Posix IPC名字到路径名的映射关系，那么这些对象的信息可使用`stat`函数或`ls`命令获取。但是如果这些对象不是使用文件实现的，那么可能获取不了这样的信息。 455
- 在众多的同步技术——互斥锁、条件变量、读写锁、记录锁、Posix信号量和System V信

① 此处为UNPv1第2版英文原版书节号，第3版为14.3节。——编者注

号量——中，可从信号处理程序中调用的函数（图5-10）只有sem_post和fcntl。

- 在众多的消息传递技术——管道、FIFO、Posix消息队列和System V消息队列——中，可从一个信号处理程序中调用的函数只有read和write（适用于管道和FIFO）。
- 在所有的消息传递技术中，只有门向服务器准确地提供了客户的标识（15.5节）。在5.4节中我们提到过，另外两种消息传递类型也标识客户：BSD/OS在使用Unix域套接字时提供这种标识（UNPv1的14.8节[①]），SVR4则在通过某个管道传递一个描述符时通过同一个管道传递发送者的标识（APUE的15.3.1节）。

456

① 此处为UNPv1第2版英文原版书节号，第3版为15.2节。——编者注

附录 A

性 能 测 量

A.1 概述

本书讨论了六种类型的消息传递：

- 管道；
- FIFO；
- Posix消息队列；
- System V消息队列；
- 门；
- Sun RPC。

和五种类型的同步：

- 互斥锁和条件变量；
- 读写锁；
- fcntl记录上锁；
- Posix信号量；
- System V信号量。

我们现在开发一些简单的程序来测量这些IPC类型的性能，这样有助于我们就何时该使用某种特定形式的IPC做出明智的决策。

比较不同形式的消息传递时，我们感兴趣的是以下两种测量尺度。

(1) 带宽（bandwidth）：数据通过IPC通道转移的速度。为测量该值，我们从一个进程向另一个进程发送大量数据（几百万字节）。我们还给不同大小的I/O操作（例如管道和FIFO的write和read操作）测量该值，期待发现带宽随每个I/O操作数据量的增长而增长的规律。

(2) 延迟（latency）：一个小的IPC消息从一个进程到另一个进程再返回所花的时间。我们测量的是只有1个字节的消息从一个进程到另一个进程再回来的时间（往返时间）。

在现实世界中，带宽告诉我们大块数据通过一个IPC通道发送出去需花多长时间，然而IPC也用于传送小的控制消息，系统处理这些小消息所需的时间就由延迟提供。这两个数都很重要。

为测量各种同步形式的性能，我们来修改将处于共享内存区中的一个计数器持续加1的程序，该程序或者由多个线程给该计数器每次加1，或者由多个进程给该计数器每次加1。既然加1是个简单的操作，因此它所需的时间差不多由同步原语操作的时间决定。

本附录中用于测量各种形式IPC之性能的简单程序大体上基于［McVoy and Staelin 1996］中描述的lmbench标准测量程序（benchmark）套件。这是一套复杂精致的标准测量程序，用于测量一个Unix系统的许多特征（上下文切换时间、I/O吞吐量，等等），而不光是IPC。它们的源代码是公开可得的：http://www.bitmover.com/lmbench。

本附录中给出的各个数值是为让我们比较本书中讲述的各种技术而提供的。所暗含的一

个动机是展示测量这些值是多么地简单。在从各种技术中作出选择之前，你必须测量自己的系统的这些性能数值。不幸的是，就像测量数值之易的程度一样，当检测到反常现象时，在没法访问出问题的内核或函数库的源代码的情况下,解释起来往往非常困难。

A.2　结果

现在汇总出自本附录的所有结果，目的是在逐个查看我们给出的各个程序的过程中提供方便的指引。

用于所有测量的两个系统是运行Solaris 2.6的一台SparcStation 4/110和运行Digital Unix 4.0B的一台Digital Alpha（DEC 3000型号300，Pelican）。往该Solaris系统的/etc/system文件中添加了以下各行：

```
set msgsys:msginfo_msgmax = 16384
set msgsys:msginfo_msgmnb = 32768
set msgsys:msginfo_msgseg = 4096
```

这样允许在System V消息队列中出现大小为16384字节的消息（图A-2）。通过作为sysconfig程序的输入指定以下各行，对该Digital Unix系统进行同样的修改：

```
ipc:
        msg-max = 16384
        msg-mnb = 32768
```

458

A.2.1　消息传递带宽结果

图A-2列出了在运行Solaris 2.6的一台Sparc主机上测得的带宽结果，图A-3图示了这些值。图A-4列出了在运行Digital Unix 4.0B的一台Alpha主机上测得的带宽结果，图A-5图示了这些值。

正如我们可能预期的那样，带宽通常随消息大小的增长而增长。由于System V消息队列的实现具有较小的内核限制值（3.8节），因此最大的消息是16384字节，而且即使对于这个大小的消息，内核默认值也不得不增加。Solaris系统的带宽在4096字节以上时减小的可能原因在于内部消息队列限制的配置。为跟UNPv1比较，我们还给出了TCP套接字和Unix域套接字的值。这两个值是使用lmbench软件包中的程序测得的，只使用了大小为65536字节的一种消息。对于TCP套接字的测量，它的两个进程处于同一台主机中。

A.2.2　消息传递延迟结果

图A-1列出了在Solaris 2.6和Digital Unix 4.0B下测得的延迟结果。

	延迟（μs）								
	管道	Posix消息队列	System V消息队列	门	Sun RPC TCP	Sun RPC UDP	TCP套接字	UDP套接字	Unix域套接字
Solaris 2.6	324	584	260	121	1891	1677	798	755	465
DUnix 4.0B	574	995	625		1648	1373	848	639	289

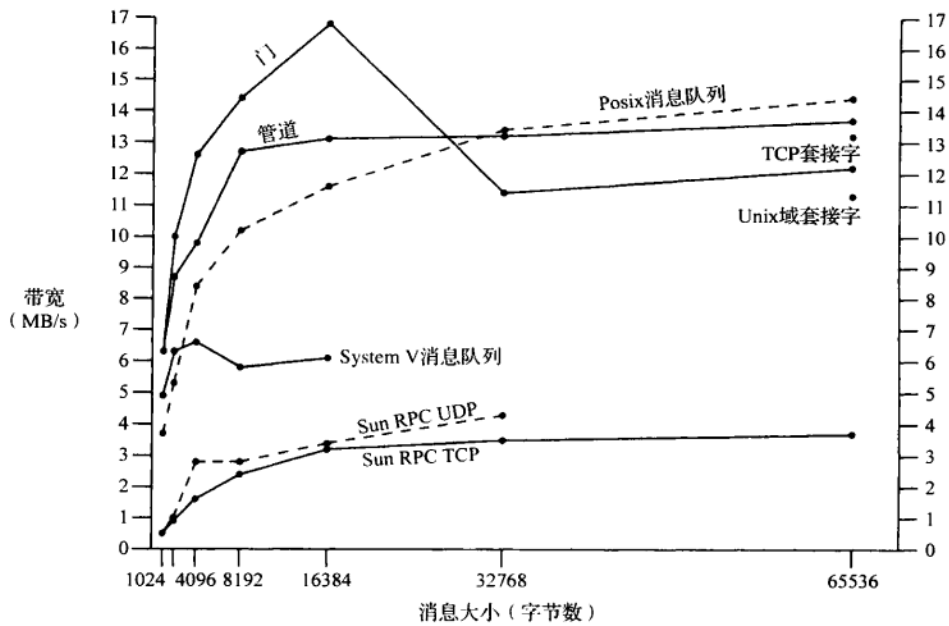
图A-1　使用各种形式的IPC交换一个1字节消息的延迟

在A.4节中我们将给出测量其中前6个值的程序，剩下3个值则出自lmbench套件。对于TCP和UDP的测量，它们的两个进程都处于同一台主机中。

459

消息大小	带宽（MB/s）							
	管道	Posix 消息队列	System V 消息队列	门	Sun RPC TCP	Sun RPC UDP	TCP 套接字	Unix域 套接字
1024	6.3	3.7	4.9	6.3	0.5	0.5		
2048	8.7	5.3	6.3	10.0	0.9	1.0		
4096	9.8	8.4	6.6	12.6	1.6	2.8		
8192	12.7	10.2	5.8	14.4	2.4	2.8		
16384	13.1	11.6	6.1	16.8	3.2	3.4		
32768	13.2	13.4		11.4	3.5	4.3		
65536	13.7	14.4		12.2	3.7		13.2	11.3

图A-2 各种类型消息传递的带宽（Solaris 2.6）

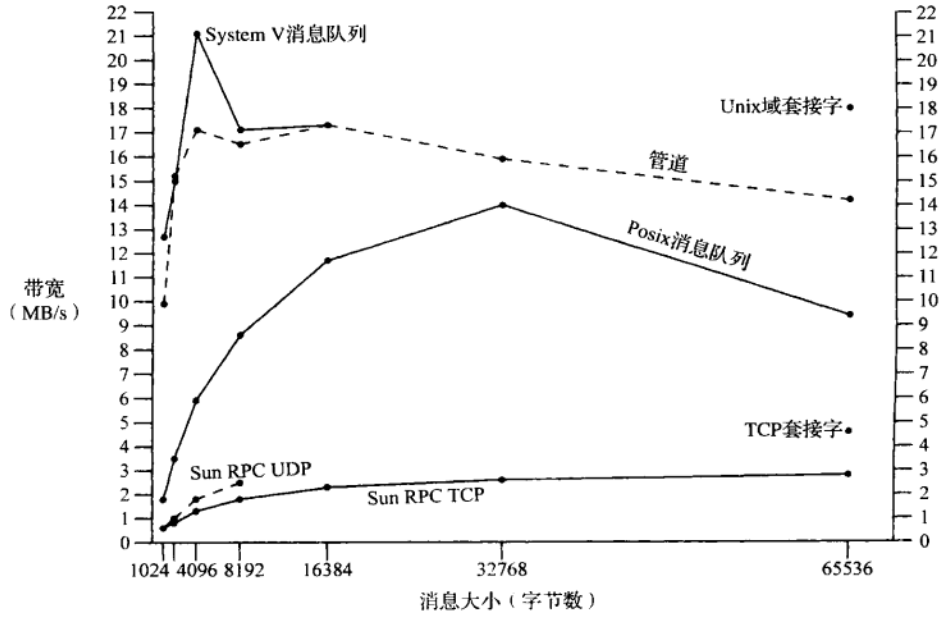


图A-3 各种类型消息传递的带宽（Solaris 2.6）

460

消息大小	带宽（MBit/s）						
	管道	Posix 消息队列	System V 消息队列	Sun RPC TCP	Sun RPC UDP	TCP 套接字	Unix域 套接字
1024	9.9	1.8	12.7	0.6	0.6		
2048	15.2	3.5	15.0	0.8	1.0		
4096	17.1	5.9	21.1	1.3	1.8		
8192	16.5	8.6	17.1	1.8	2.5		
16384	17.3	11.7	17.3	2.3			
32768	15.9	14.0		2.6			
65536	14.2	9.4		2.8		4.6	18.0

图A-4 各种类型消息传递的带宽（Digital Unix 4.0B）



461

图A-5　各种类型消息传递的带宽（Digital Unix 4.0B）

A.2.3　线程同步结果

图A-6列出了Solaris 2.6下使用各种形式的同步，由一个或多个线程给处于共享内存区中的一个计数器持续加1所需的时间，图A-7图示了这些值。每个线程给该计数器加1共1 000 000次，这样的线程数则从1到5变化。图A-8列出了Digital Unix 4.0B下的这些值，图A-9图示了这些值。

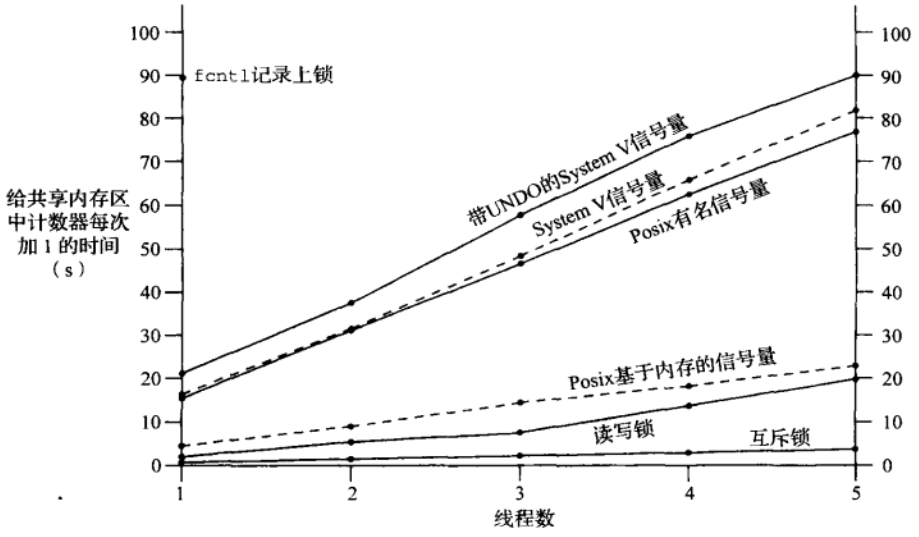
增加线程数的原因在于验证使用同步技术的代码是正确的，并查看时间是否随线程数的增加而开始非线性地增长。对于fcntl记录上锁我们只能测量单个线程，因为这种同步形式工作于进程间，而不是单个进程内的多个线程间。

Digital Unix下，线程数多于一个时两种Posix信号量类型的时间变得非常大，表明存在某种类型的反常。我们没有图示这些值。

出现这些比预期值大的数值的可能原因之一是，本程序是一个病态的同步测试程序。也就是说，各个线程除同步外什么都不干，因而其中的锁基本上所有时间都为某个线程所锁住。既然默认情况下线程是以进程竞用范围属性创建的，因此每当一个线程失去它的时间片时，它可能仍持有该锁，于是切换来运行的新线程可能立即阻塞。

线程数	给共享内存区中的一个计数器持续加1所需时间（s）						
	Posix 互斥锁	读写锁	Posix 基于内存信号量	Posix 有名信号量	System V 信号量	带UNDO的 System V信号量	fcntl 记录上锁
1	0.7	2.0	4.5	15.4	16.3	21.1	89.4
2	1.5	5.4	9.0	31.1	31.5	37.5	
3	2.2	7.5	14.4	46.5	48.3	57.7	
4	2.9	13.7	18.2	62.5	65.8	75.8	
5	3.7	19.7	22.8	76.8	81.8	90.0	

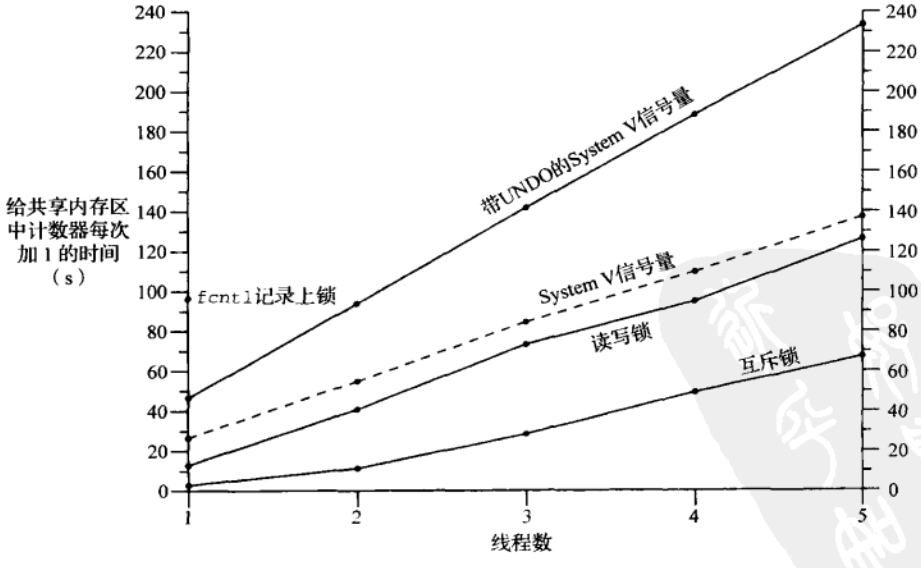
图A-6　给处于共享内存区中的一个计数器持续加1所需的时间（Solaris 2.6）



图A-7 给处于共享内存区中的一个计数器持续加1所需的时间（Solaris 2.6）

线程数	给共享内存区中的一个计数器持续加1所需时间（s）						
	Posix互斥锁	读写锁	Posix基于内存信号量	Posix有名信号量	System V信号量	带UNDO的System V信号量	fcntl记录上锁
1	2.9	12.9	13.2	14.2	26.6	46.6	96.4
2	11.4	40.8	742.5	771.6	54.9	93.9	
3	28.4	73.2	1080.5	1074.7	84.5	141.9	
4	49.3	95.0	1534.1	1502.2	109.9	188.4	
5	67.3	126.3	1923.3	1764.1	137.3	233.6	

图A-8 给处于共享内存区中的一个计数器持续加1所需的时间（Digital Unix 4.0B）



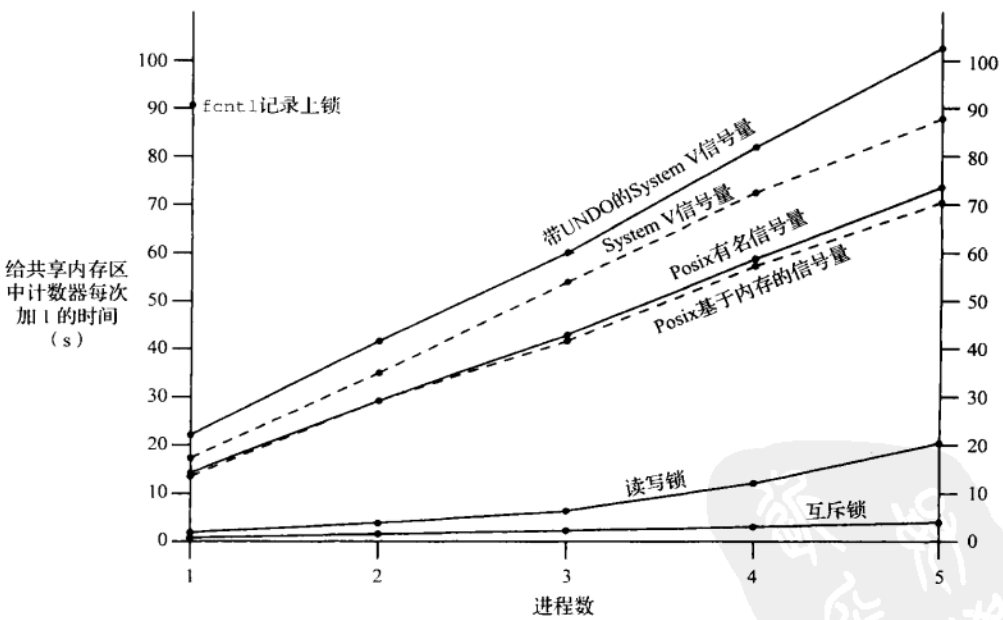
图A-9 给处于共享内存区中的一个计数器持续加1所需的时间（Digital Unix 4.0B）

A.2.4　进程同步结果

图A-6、图A-7、图A-8和图A-9展示了各种同步技术用于同步单个进程内的各个线程时测得的结果。图A-10和图A-11给出了Solaris 2.6下在不同进程间共享计数器时这些同步技术的性能。图A-12和图A-13给出了Digital Unix 4.0B下的进程同步结果。这些结果与对应的线程同步结果类似，不过两种Posix信号量形式的值现在类似于Solaris的结果。我们只画出了fcntl记录上锁的第一个值，因为其余各值太大了。正如我们在7.2节中注出的那样，Digital Unix 4.0B不支持PTHREAD_PROCESS_SHARED特性，因此我们无法测量不同进程间的互斥锁值。在Digital Unix下当涉及多个进程时，我们再次看到了Posix信号量的某种类型的反常。

进程数	给共享内存区中的一个计数器持续加1所需时间（s）						
	Posix互斥锁	读写锁	Posix基于内存信号量	Posix有名信号量	System V信号量	带UNDO的System V信号量	fcntl记录上锁
1	0.8	1.9	13.6	14.3	17.3	22.1	90.7
2	1.6	3.9	29.2	29.2	34.9	41.6	244.5
3	2.3	6.4	41.6	42.9	54.0	60.1	376.4
4	3.1	12.2	57.3	58.8	72.4	81.9	558.0
5	4.0	20.4	70.4	73.5	87.8	102.6	764.0

图A-10　给处于共享内存区中的一个计数器持续加1所需的时间（Solaris 2.6）

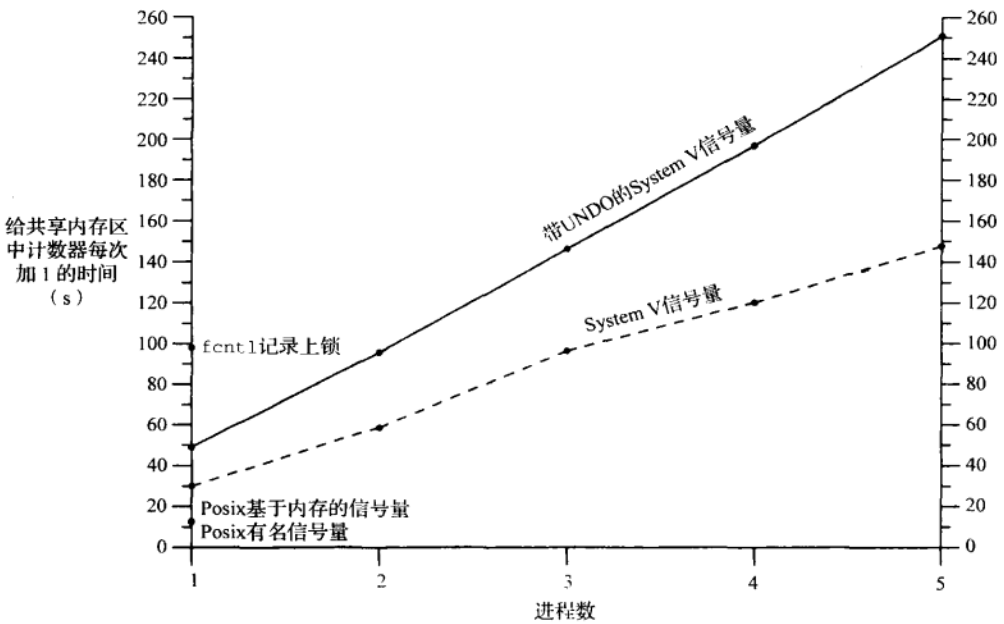


462 ~ 465

图A-11　给处于共享内存区中的一个计数器持续加1所需的时间（Solaris 2.6）

进程数	给共享内存区中的一个计数器持续加1所需时间（s）				
	Posix基于内存信号量	Posix有名信号量	System V信号量	带UNDO的System V信号量	fcntl记录上锁
1	12.8	12.5	30.1	49.0	98.1
2	664.8	659.2	58.6	95.7	477.1
3	1236.1	1269.8	96.4	146.2	1785.2
4	1772.9	1804.1	120.3	197.0	2582.8
5	2179.9	2196.8	147.7	250.9	3419.2

图A-12 给处于共享内存区中的一个计数器持续加1所需的时间（Digital Unix 4.0B）



图A-13 给处于共享内存区中的一个计数器持续加1所需的时间（Digital Unix 4.0B） 466

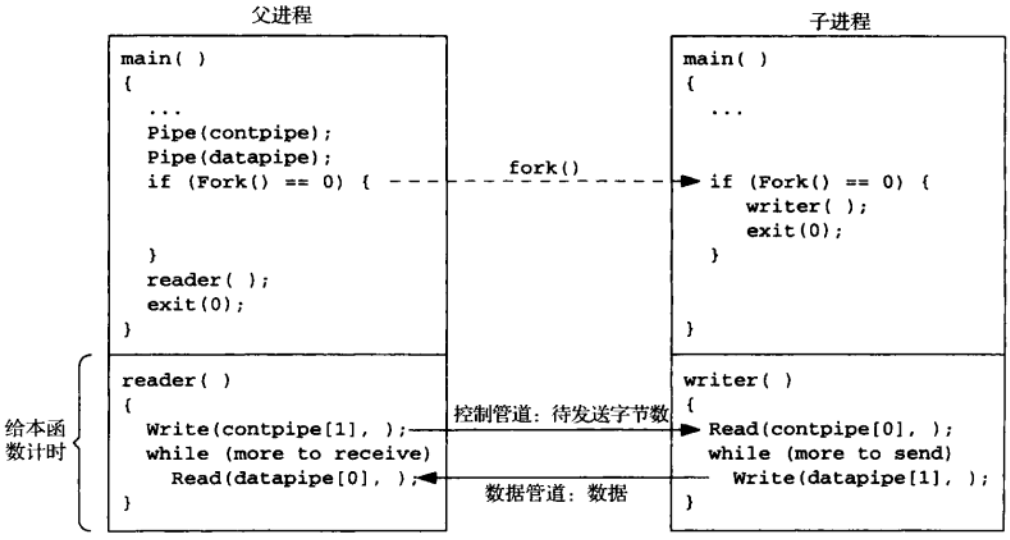
A.3 消息传递带宽程序

本节给出测量管道、Posix消息队列、System V消息队列、门和Sun RPC的带宽的各个程序。我们已在图A-2和图A-3中给出了这些程序的结果。

A.3.1 管道带宽程序

图A-14展示了我们将描述的程序的概貌。

图A-15给出了我们的bw_pipe程序的前半部分，它测量一个管道的带宽。



图A-14　测量一个管道的带宽的程序的概貌

bench/bw_pipe.c

```
 1 #include    "unpipc.h"
 2 void     reader(int, int, int);
 3 void     writer(int, int);
 4 void    *buf;
 5 int      totalnbytes, xfersize;
 6 int
 7 main(int argc, char **argv)
 8 {
 9     int     i, nloop, contpipe[2], datapipe[2];
10     pid_t   childpid;
11     if (argc != 4)
12         err_quit("usage: bw_pipe <#loops> <#mbytes> <#bytes/write>");
13     nloop = atoi(argv[1]);
14     totalnbytes = atoi(argv[2]) * 1024 * 1024;
15     xfersize = atoi(argv[3]);
16     buf = Valloc(xfersize);
17     Touch(buf, xfersize);
18     Pipe(contpipe);
19     Pipe(datapipe);
20     if ( (childpid = Fork()) == 0) {
21         writer(contpipe[0], datapipe[1]);   /* child */
22         exit(0);
23     }
24         /* parent */
25     Start_time();
26     for (i = 0; i < nloop; i++)
27         reader(contpipe[1], datapipe[0], totalnbytes);
28     printf("bandwidth: %.3f MB/sec\n",
29            totalnbytes / Stop_time() * nloop);
30     kill(childpid, SIGTERM);
31     exit(0);
32 }
```

bench/bw_pipe.c

图A-15　一个管道带宽测量程序的main函数

命令行参数

11~15 命令行参数指定待执行的循环数（以下测量中的典型值为5）、待传送的M字节[1]数（值为10的参数导致传送10×1024×1024字节）以及每次write和read的字节数（我们给出的测量结果中该值在1024和65536之间变化）。

分配缓冲区并触及它的各个页面

16~17 valloc是malloc的版本之一，它从某个页面边界开始分配所请求数量的内存空间。我们的函数touch（图A-17）在该缓冲区的每个页面中存入1字节的数据，从而迫使内核把构成该缓冲区的每个页面置换进内存。我们在进行计时之前完成这些工作。

valloc不属于Posix.1函数，Unix 98把它列为一个“代传（legacy）”接口：早期版本的X/Open规范需要它，但它现在是可选的。如果valloc不受支持，我们的Valloc包裹函数就调用malloc来实现它的功能。

创建两个管道

18~19 创建两个管道，其中contpipe[0]和contpipe[1]用于在开始每次传送之前同步一读一写两个进程，datapipe[0]和datapipe[1]用于真正的数据传送。

调用fork派生子进程

20~31 行派生一个子进程，该子进程（它的fork返回值为0）调用writer函数，父进程则调用reader函数。父进程中reader函数调用nloop次。我们的start_time函数在该循环开始前即刻调用，我们的stop_time函数在该循环终止后马上调用。图A-17给出了这两个函数。所输出的带宽为每次循环传送的总字节数除以传送数据所花的时间（stop_time返回的是自调用start_time以来流逝的微秒数），再乘以循环次数。子进程随后被父进程发送的SIGTERM信号杀死，程序随后终止。

467 ~ 468

图A-16给出了bw_pipe程序的后半部分，它包含两个函数writer和reader。

bench/bw_pipe.c

```
33 void
34 writer(int contfd, int datafd)
35 {
36     int     ntowrite;
37     for ( ; ; ) {
38         Read(contfd, &ntowrite, sizeof(ntowrite));
39         while (ntowrite > 0) {
40             Write(datafd, buf, xfersize);
41             ntowrite -= xfersize;
42         }
43     }
44 }
45 void
46 reader(int contfd, int datafd, int nbytes)
47 {
48     ssize_t n;
49     Write(contfd, &nbytes, sizeof(nbytes));
50     while ((nbytes > 0) &&
51            ( (n = Read(datafd, buf, xfersize)) > 0)) {
52         nbytes -= n;
53     }
54 }
```

bench/bw_pipe.c

图A-16 测量一个管道的带宽的writer和reader函数

① megabyte一词存在歧义。我们在它代表2^{20}字节时译为M字节，在它代表10^6字节时译为兆字节。kilobyte和gigabyte两词也有类似译法。——译者注

writer函数

33~44　本函数是由子进程调用的一个无限循环。子进程通过在控制管道读出一个整数（该整数指定子进程应写入数据管道的字节数），来等待父进程表明自身已准备好接收数据。接收到这个通知后，子进程通过管道向父进程写入数据，每次write调用写xfersize个字节。

reader函数

45~54　本函数是由父进程在一个循环中调用的。它每次被调用时往控制管道写入一个整数，告诉子进程应往数据管道中写入多少字节的数据。本函数随后在一个循环中调用read，直到全部数据都接收完为止。

469

图A-17　给出了我们的start_time、stop_time和touch函数。

lib/timing.c

```
 1 #include    "unpipc.h"
 2 static struct timeval   tv_start, tv_stop;
 3 int
 4 start_time(void)
 5 {
 6     return(gettimeofday(&tv_start, NULL));
 7 }
 8 double
 9 stop_time(void)
10 {
11     double  clockus;
12     if (gettimeofday(&tv_stop, NULL) == -1)
13         return(0.0);
14     tv_sub(&tv_stop, &tv_start);
15     clockus = tv_stop.tv_sec * 1000000.0 + tv_stop.tv_usec;
16     return(clockus);
17 }
18 int
19 touch(void *vptr, int nbytes)
20 {
21     char    *cptr;
22     static int  pagesize = 0;
23     if (pagesize == 0) {
24         errno = 0;
25 #ifdef  _SC_PAGESIZE
26         if ( (pagesize = sysconf(_SC_PAGESIZE)) == -1)
27             return(-1);
28 #else
29         pagesize = getpagesize();       /* BSD */
30 #endif
31     }
32     cptr = vptr;
33     while (nbytes > 0) {
34         *cptr = 1;
35         cptr += pagesize;
36         nbytes -= pagesize;
37     }
38     return(0);
39 }
```

lib/timing.c

470

图A-17　计时函数：start_time、stop_time和touch

图A-18给出了tv_sub函数：它在两个timeval结构间做减法运算，并把结果存回第一个结构中。

lib/tv_sub.c

```
 1 #include    "unpipc.h"

 2 void
 3 tv_sub(struct timeval *out, struct timeval *in)
 4 {
 5     if ( (out->tv_usec -= in->tv_usec) < 0) {    /* out -= in */
 6         --out->tv_sec;
 7         out->tv_usec += 1000000;
 8     }
 9     out->tv_sec -= in->tv_sec;
10 }
```

lib/tv_sub.c

图A-18 tv_sub函数：两个timeval结构相减

在运行Solaris 2.6的一台Sparc主机上接连运行我们的程序5次，得到如下结果：

```
solaris % bw_pipe 5 10 65536
bandwidth: 13.722 MB/sec
solaris % bw_pipe 5 10 65536
bandwidth: 13.781 MB/sec
solaris % bw_pipe 5 10 65536
bandwidth: 13.685 MB/sec
solaris % bw_pipe 5 10 65536
bandwidth: 13.665 MB/sec
solaris % bw_pipe 5 10 65536
bandwidth: 13.584 MB/sec
```

每次我们指定5轮循环，每轮循环传送10 485 760个字节，每次write和read调用收发65536个字节。这5次程序运行的平均值为图A-2中所示的13.7MB/s。

A.3.2 Posix消息队列带宽程序

图A-19是一个Posix消息队列带宽测量程序的main函数。图A-20给出了其中的writer和reader函数。该程序与前面那个管道带宽测量程序类似。

bench/bw_pxmsg.c

```
 1 #include    "unpipc.h"
 2 #define NAME    "bw_pxmsg"

 3 void     reader(int, mqd_t, int);
 4 void     writer(int, mqd_t);

 5 void    *buf;
 6 int      totalnbytes, xfersize;

 7 int
 8 main(int argc, char **argv)
 9 {
10     int     i, nloop, contpipe[2];
11     mqd_t   mq;
12     pid_t   childpid;
13     struct mq_attr  attr;

14     if (argc != 4)
15         err_quit("usage: bw_pxmsg <#loops> <#mbytes> <#bytes/write>");
16     nloop = atoi(argv[1]);
```

图A-19 Posix消息队列带宽测量程序的main函数

```
17      totalnbytes = atoi(argv[2]) * 1024 * 1024;
18      xfersize = atoi(argv[3]);

19      buf = Valloc(xfersize);
20      Touch(buf, xfersize);

21      Pipe(contpipe);
22      mq_unlink(Px_ipc_name(NAME));  /* error OK */
23      attr.mq_maxmsg = 4;
24      attr.mq_msgsize = xfersize;
25      mq = Mq_open(Px_ipc_name(NAME), O_RDWR | O_CREAT, FILE_MODE, &attr);

26      if ( (childpid = Fork()) == 0) {
27          writer(contpipe[0], mq);        /* child */
28          exit(0);
29      }
30          /* parent */
31      Start_time();
32      for (i = 0; i < nloop; i++)
33          reader(contpipe[1], mq, totalnbytes);
34      printf("bandwidth: %.3f MB/sec\n",
35             totalnbytes / Stop_time() * nloop);

36      kill(childpid, SIGTERM);
37      Mq_close(mq);
38      Mq_unlink(Px_ipc_name(NAME));
39      exit(0);
40  }
```

bench/bw_pxmsg.c

图A-19（续）

bench/bw_pxmsg.c

```
41  void
42  writer(int contfd, mqd_t mqsend)
43  {
44      int     ntowrite;

45      for ( ; ; ) {
46          Read(contfd, &ntowrite, sizeof(ntowrite));

47          while (ntowrite > 0) {
48              Mq_send(mqsend, buf, xfersize, 0);
49              ntowrite -= xfersize;
50          }
51      }
52  }

53  void
54  reader(int contfd, mqd_t mqrecv, int nbytes)
55  {
56      ssize_t n;

57      Write(contfd, &nbytes, sizeof(nbytes));

58      while ((nbytes > 0) &&
59             ( (n = Mq_receive(mqrecv, buf, xfersize, NULL)) > 0)) {
60          nbytes -= n;
61      }
62  }
```

bench/bw_pxmsg.c

图A-20　测量一个Posix消息队列的带宽的wirter和reader函数

注意，在我们创建消息队列时，必须指定该程序上能存在的最大消息数，我们把它指定为4。IPC通道的容量能够影响性能，因为写进程在阻塞于某个mqsend调用之前能够发送这么多的消息，然后由内核将上下文切换到读进程。因此本程序的性能依赖于这个魔数。Solaris 2.6下把该数从4改为8并不影响图A-2中的各个数值，然而Digital Unix 4.0B下的同样变动却使性能下降了12%。我们原本会猜想消息数较多时性能将增长，因为这可以让上下文切换数降低一半。然而如果用到了内存映射文件，那么这样一来将使该文件的大小增加一倍，所映射的内存区大小也增长一倍。

A.3.3 System V消息队列带宽程序

图A-21是一个System V消息队列带宽测量程序的main函数，图A-22给出了其中的writer和reader函数。

bench/bw_svmsg.c

```
 1 #include    "unpipc.h"

 2 void    reader(int, int, int);
 3 void    writer(int, int);

 4 struct msgbuf *buf;
 5 int     totalnbytes, xfersize;

 6 int
 7 main(int argc, char **argv)
 8 {
 9     int     i, nloop, contpipe[2], msqid;
10     pid_t   childpid;

11     if (argc != 4)
12         err_quit("usage: bw_svmsg <#loops> <#mbytes> <#bytes/write>");
13     nloop = atoi(argv[1]);
14     totalnbytes = atoi(argv[2]) * 1024 * 1024;
15     xfersize = atoi(argv[3]);

16     buf = Valloc(xfersize);
17     Touch(buf, xfersize);
18     buf->mtype = 1;

19     Pipe(contpipe);
20     msqid = Msgget(IPC_PRIVATE, IPC_CREAT | SVMSG_MODE);

21     if ( (childpid = Fork()) == 0) {
22         writer(contpipe[0], msqid);         /* child */
23         exit(0);
24     }
25     Start_time();
26     for (i = 0; i < nloop; i++)
27         reader(contpipe[1], msqid, totalnbytes);
28     printf("bandwidth: %.3f MB/sec\n",
29            totalnbytes / Stop_time() * nloop);

30     kill(childpid, SIGTERM);
31     Msgctl(msqid, IPC_RMID, NULL);
32     exit(0);
33 }
```

bench/bw_svmsg.c

图A-21 一个System V消息队列带宽测量程序的main函数

bench/bw_svmsg.c

```
34 void
35 writer(int contfd, int msqid)
36 {
37      int     ntowrite;

38      for ( ; ; ) {
39          Read(contfd, &ntowrite, sizeof(ntowrite));

40          while (ntowrite > 0) {
41              Msgsnd(msqid, buf, xfersize - sizeof(long), 0);
42              ntowrite -= xfersize;
43          }
44      }
45 }

46 void
47 reader(int contfd, int msqid, int nbytes)
48 {
49      ssize_t n;

50      Write(contfd, &nbytes, sizeof(nbytes));

51      while ((nbytes > 0) &&
52             ( (n = Msgrcv(msqid, buf, xfersize - sizeof(long), 0, 0)) > 0)) {
53          nbytes -= n + sizeof(long);
54      }
55 }
```

bench/bw_svmsg.c

474

图A-22　测量一个System V消息队列的带宽的writer和reader函数

A.3.4　门带宽程序

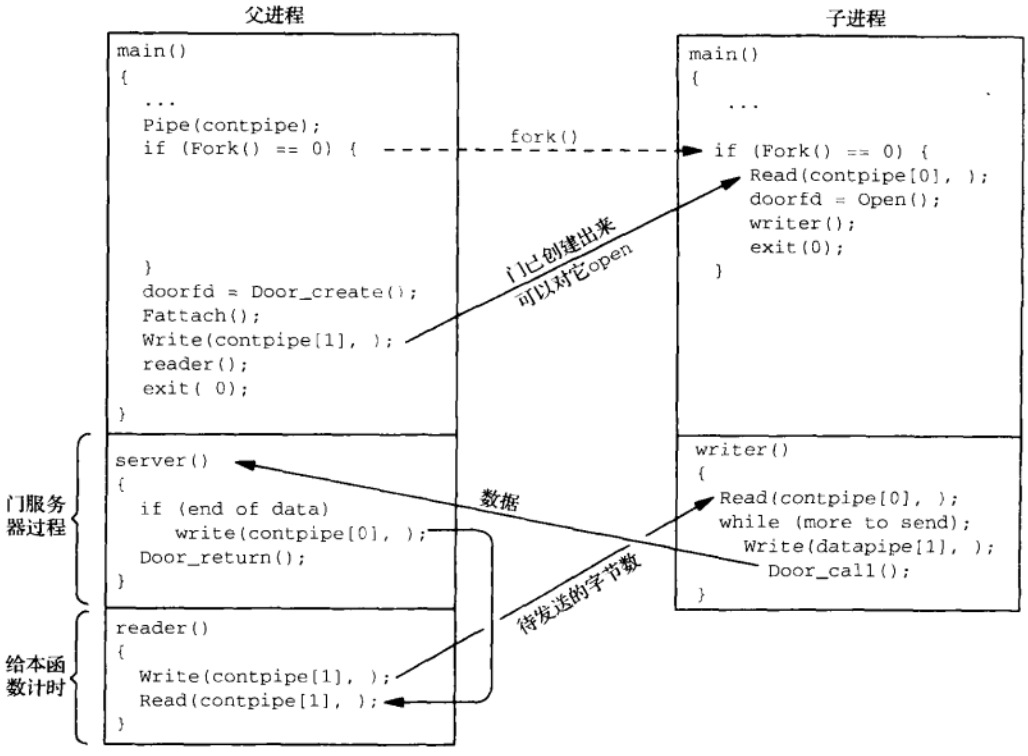
门API带宽测量程序比本节中先前给出的程序要复杂，因为在创建其中的门之前必须fork出子进程。父进程接着创建该门，并通过写一个管道来通知子进程该门能被打开。

不像图A-14的另一个变化是，reader函数不接收数据。相反，数据是由名为server的函数接收的，它是对应门的服务器过程。图A-23给出了该程序的概貌。

门只在Solaris下受支持，于是我们通过假设采用全双工管道（4.4节）来简化程序本身。

与先前的程序相比，另一个变化在于消息传递和过程调用间的基本差异。例如在我们的Posix消息队列程序中，写入者只是在一个循环中往一个队列写入消息，这种写操作是异步的。到某个时刻队列满了，或者写进程丧失了自己的处理器时间片，读出者就开始运行，读出这些消息。举例来说，如果该队列可容纳8个消息，而且写入者每次运行时写入8个消息，读出者每次运行时读出所有8个消息，那么发送N个消息将涉及N/4次上下文切换（其中N/8次是从写入者到读出者，另外N/8次是从读出者到写入者）。然而门API是同步的：调用者每次调用door_call时阻塞，直到服务器过程返回才能恢复。交换N个消息现在涉及N×2次上下文切换。测量RPC调用的带宽时，我们将碰到同样的问题。尽管上下文切换数增加了，从图A-3可以看出，当消息大小在约25000字节或以上时，门却提供了最快的IPC带宽。

图A-24给出了该程序的main函数。writer、server和reader函数则在图A-25中给出。



图A-23 门API带宽测量程序的概貌

bench/bw_door.c

```
1 #include     "unpipc.h"

2 void       reader(int, int);
3 void       writer(int);
4 void       server(void *, char *, size_t, door_desc_t *, size_t);

5 void      *buf;
6 int        totalnbytes, xfersize, contpipe[2];

7 int
8 main(int argc, char **argv)
9 {
10      int     i, nloop, doorfd;
11      char    c;
12      pid_t   childpid;
13      ssize_t n;

14      if (argc != 5)
15      err_quit("usage: bw_door <pathname> <#loops> <#mbytes> <#bytes/write>");
16      nloop = atoi(argv[2]);
17      totalnbytes = atoi(argv[3]) * 1024 * 1024;
18      xfersize = atoi(argv[4]);

19      buf = Valloc(xfersize);
20      Touch(buf, xfersize);
```

图A-24 门API带宽测量程序的main函数

```
21      unlink(argv[1]);
22      Close(Open(argv[1], O_CREAT | O_EXCL | O_RDWR, FILE_MODE));
23      Pipe(contpipe);             /* assumes full-duplex SVR4 pipe */

24      if ( (childpid = Fork()) == 0) {
25              /* child = client = writer */
26          if ( (n = Read(contpipe[0], &c, 1)) != 1)
27              err_quit("child: pipe read returned %d", n);
28          doorfd = Open(argv[1], O_RDWR);

29          writer(doorfd);
30          exit(0);
31      }
32          /* parent = server = reader */
33      doorfd = Door_create(server, NULL, 0);
34      Fattach(doorfd, argv[1]);
35      Write(contpipe[1], &c, 1);      /* tell child door is ready */

36      Start_time();
37      for (i = 0; i < nloop; i++)
38          reader(doorfd, totalnbytes);
39      printf("bandwidth: %.3f MB/sec\n",
40             totalnbytes / Stop_time() * nloop);
41      kill(childpid, SIGTERM);
42      unlink(argv[1]);
43      exit(0);
44  }
```

bench/bw_door.c

图A-24（续）

bench/bw_door.c

```
45  void
46  writer(int doorfd)
47  {
48      int     ntowrite;
49      door_arg_t  arg;

50      arg.desc_ptr = NULL;        /* no descriptors to pass */
51      arg.desc_num = 0;
52      arg.rbuf = NULL;            /* no return values expected */
53      arg.rsize = 0;

54      for ( ; ; ) {
55          Read(contpipe[0], &ntowrite, sizeof(ntowrite));

56          while (ntowrite > 0) {
57              arg.data_ptr = buf;
58              arg.data_size = xfersize;
59              Door_call(doorfd, &arg);
60              ntowrite -= xfersize;
61          }
62      }
63  }

64  static int  ntoread, nread;

65  void
66  server(void *cookie, char *argp, size_t arg_size,
67         door_desc_t *dp, size_t n_descriptors)
68  {
```

图A-25　测量门API的带宽的writer、server和reader函数

```
69      char    c;
70      nread += arg_size;
71      if (nread >= ntoread)
72          Write(contpipe[0], &c, 1);      /* tell reader() we are all done */
73      Door_return(NULL, 0, NULL, 0);
74  }
75  void
76  reader(int doorfd, int nbytes)
77  {
78      char    c;
79      ssize_t n;
80      ntoread = nbytes;           /* globals for server() procedure */
81      nread = 0;
82      Write(contpipe[1], &nbytes, sizeof(nbytes));
83      if ( (n = Read(contpipe[1], &c, 1)) != 1)
84          err_quit("reader: pipe read returned %d", n);
85  }
```

bench/bw_door.c

图A-25（续）

A.3.5 Sun RPC带宽程序

既然Sun RPC中的过程调用是同步的，那么我们有已随前面的门程序提到过的同样限制。使用RPC时生成一个客户和一个服务器两个程序更为容易，因为它们是由rpcgen生成的。图A-26给出了本程序的RPC说明书文件。我们声明了单个过程，它以一个可变长度不透明数据作为输入，不返回任何东西。

bench/bw_sunrpc.x

```
1 %#define     DEBUG       /* so server runs in foreground */
2 struct data_in {
3   opaque      data<>;     /* variable-length opaque data */
4 };
5 program BW_SUNRPC_PROG {
6     version BW_SUNRPC_VERS {
7         void BW_SUNRPC(data_in) = 1;
8     } = 1;
9 } = 0x31230001;
```

bench/bw_sunrpc.x

图A-26 Sun RPC带宽测量程序的RPC说明书文件

图A-27给出了我们的客户程序，图A-28给出了我们的服务器过程。我们把协议（TCP或UDP）指定为客户程序的一个命令行参数，以允许分别测量这两种协议。

bench/bw_sunrpc_client.c

```
1 #include    "unpipc.h"
2 #include    "bw_sunrpc.h"
3 void    *buf;
4 int      totalnbytes, xfersize;
5 int
```

图A-27 测量Sun RPC的带宽的RPC客户程序

```
 6 main(int argc, char **argv)
 7 {
 8     int     i, nloop, ntowrite;
 9     CLIENT  *cl;
10     data_in in;

11     if (argc != 6)
12         err_quit("usage: bw_sunrpc_client <hostname> <#loops>"
13                  " <#mbytes> <#bytes/write> <protocol>");
14     nloop = atoi(argv[2]);
15     totalnbytes = atoi(argv[3]) * 1024 * 1024;
16     xfersize = atoi(argv[4]);

17     buf = Valloc(xfersize);
18     Touch(buf, xfersize);

19     cl = Clnt_create(argv[1], BW_SUNRPC_PROG, BW_SUNRPC_VERS, argv[5]);

20     Start_time();
21     for (i = 0; i < nloop; i++) {
22         ntowrite = totalnbytes;
23         while (ntowrite > 0) {
24             in.data.data_len = xfersize;
25             in.data.data_val = buf;
26             if (bw_sunrpc_1(&in, cl) == NULL)
27                 err_quit("%s", clnt_sperror(cl, argv[1]));
28             ntowrite -= xfersize;
29         }
30     }
31     printf("bandwidth: %.3f MB/sec\n",
32            totalnbytes / Stop_time() * nloop);
33     exit(0);
34 }
```

bench/bw_sunrpc_client.c

475 ~ 479

图A-27（续）

bench/bw_sunrpc_server.c

```
 1 #include    "unpipc.h"
 2 #include    "bw_sunrpc.h"

 3 #ifndef RPCGEN_ANSIC
 4 #define bw_sunrpc_1_svc bw_sunrpc_1
 5 #endif

 6 void *
 7 bw_sunrpc_1_svc(data_in *inp, struct svc_req *rqstp)
 8 {
 9     static int  nbytes;

10     nbytes = inp->data.data_len;
11     return(&nbytes);            /* must be nonnull, but xdr_void() will ignore */
12 }
```

bench/bw_sunrpc_server.c

图A-28　测量Sun RPC的带宽的RPC服务器过程

A.4　消息传递延迟程序

本节给出测量管道、Posix消息队列、System V消息队列、门和Sun RPC的延迟的各个程序。

图A-1给出了它们的性能值。

A.4.1 管道延迟程序

图A-29给出了测量一个管道的延迟的程序。

bench/lat_pipe.c

```
 1 #include    "unpipc.h"
 2 void
 3 doit(int readfd, int writefd)
 4 {
 5     char    c;
 6     Write(writefd, &c, 1);
 7     if (Read(readfd, &c, 1) != 1)
 8         err_quit("read error");
 9 }
10 int
11 main(int argc, char **argv)
12 {
13     int     i, nloop, pipe1[2], pipe2[2];
14     char    c;
15     pid_t   childpid;
16     if (argc != 2)
17         err_quit("usage: lat_pipe <#loops>");
18     nloop = atoi(argv[1]);
19     Pipe(pipe1);
20     Pipe(pipe2);
21     if ( (childpid = Fork()) == 0) {
22         for ( ; ; ) {    /* child */
23             if (Read(pipe1[0], &c, 1) != 1)
24                 err_quit("read error");
25             Write(pipe2[1], &c, 1);
26         }
27         exit(0);
28     }
29         /* parent */
30     doit(pipe2[0], pipe1[1]);
31     Start_time();
32     for (i = 0; i < nloop; i++)
33         doit(pipe2[0], pipe1[1]);
34     printf("latency: %.3f usec\n", Stop_time() / nloop);
35     Kill(childpid, SIGTERM);
36     exit(0);
37 }
```

bench/lat_pipe.c

图A-29 测量一个管道的延迟的程序

doit函数

2~9 本函数在父进程中运行，它所花的时钟时间将被测量出来。它往一个管道写入1个字节（该字节将由子进程读出）后从另一个管道读出1个字节（该字节由子进程写入）。这就是我们所描述的延迟：从发送一个小消息到接收作为应答的一个小消息所花的时间。

创建管道

19~20　创建两个管道，fork一个子进程，形成图4-6所示的布局（不过没有关闭每个管道的未用端，这是不会有问题的）。本测试程序确实需要两个管道，因为管道是半双工的，而我们希望父子进程间有双向通信。

子进程回射只有1个字节的消息

22~27　子进程执行一个无限循环，每次读出一个只有1个字节的消息后就把它发射回来。

测量父进程

29~34　父进程首先调用doit给子进程发送一个只有1个字节的消息，并读出它的也是只有1个字节的应答。这使得两个进程都处于运行状态。父进程然后在一个循环中调用doit函数，同时测量所花的时钟时间。

480 ~ 481

在运行Solaris 2.6的一台Sparc主机上接连运行该程序5次，得到如下结果：

```
solaris % lat_pipe 10000
latency: 278.633 usec
solaris % lat_pipe 10000
latency: 397.810 usec
solaris % lat_pipe 10000
latency: 392.567 usec
solaris % lat_pipe 10000
latency: 266.572 usec
solaris % lat_pipe 10000
latency: 284.559 usec
```

这5次程序运行的平均值为324微秒，它就是图A-1中给出的值。这些时间包括两次上下文切换（从父进程到子进程，然后是从子进程到父进程）、四个系统调用（父进程的write、子进程的read、子进程的write、父进程的read）以及每个方向1字节数据的管道开销。

A.4.2　Posix消息队列延迟程序

图A-30给出了一个Posix消息队列延迟测量程序。

bench/lat_pxmsg.c

```
 1 #include    "unpipc.h"
 2 #define NAME1   "lat_pxmsg1"
 3 #define NAME2   "lat_pxmsg2"
 4 #define MAXMSG     4            /* room for 4096 bytes on queue */
 5 #define MSGSIZE 1024

 6 void
 7 doit(mqd_t mqsend, mqd_t mqrecv)
 8 {
 9     char    buff[MSGSIZE];

10     Mq_send(mqsend, buff, 1, 0);
11     if (Mq_receive(mqrecv, buff, MSGSIZE, NULL) != 1)
12         err_quit("mq_receive error");
13 }

14 int
15 main(int argc, char **argv)
16 {
17     int     i, nloop;
18     mqd_t   mq1, mq2;
19     char    buff[MSGSIZE];
```

图A-30　一个Posix消息队列延迟测量程序

```
20      pid_t   childpid;
21      struct mq_attr  attr;

22      if (argc != 2)
23          err_quit("usage: lat_pxmsg <#loops>");
24      nloop = atoi(argv[1]);

25      attr.mq_maxmsg = MAXMSG;
26      attr.mq_msgsize = MSGSIZE;
27      mq1 = Mq_open(Px_ipc_name(NAME1), O_RDWR | O_CREAT, FILE_MODE, &attr);
28      mq2 = Mq_open(Px_ipc_name(NAME2), O_RDWR | O_CREAT, FILE_MODE, &attr);

29      if ( (childpid = Fork()) == 0) {
30          for ( ; ; ) {           /* child */
31              if (Mq_receive(mq1, buff, MSGSIZE, NULL) != 1)
32                  err_quit("mq_receive error");
33              Mq_send(mq2, buff, 1, 0);
34          }
35          exit(0);
36      }
37          /* parent */
38      doit(mq1, mq2);

39      Start_time();
40      for (i = 0; i < nloop; i++)
41          doit(mq1, mq2);
42      printf("latency: %.3f usec\n", Stop_time() / nloop);

43      Kill(childpid, SIGTERM);
44      Mq_close(mq1);
45      Mq_close(mq2);
46      Mq_unlink(Px_ipc_name(NAME1));
47      Mq_unlink(Px_ipc_name(NAME2));
48      exit(0);
49  }
```

bench/lat_pxmsg.c

图A-30（续）

25~28 创建两个消息队列，一个用于从父进程到子进程的消息传递，另一个用于从子进程到父进程的消息传递。尽管Posix消息具有优先级，从而允许我们给两个不同方向的消息赋不同的优先级，mq_receive却总是返回队列中的下一个消息。因此，我们不能仅给本测试程序使用一个队列。

A.4.3 System V消息队列延迟程序

图A-31给出了一个System V消息队列延迟测量程序。

本程序只创建一个消息队列，它含有两个不同方向的消息：从父进程到子进程和从子进程到父进程。前者的类型字段为1，后者的类型字段为2。doit中msgrcv的第四个参数为2，子进程中msgrcv的第四个参数为1，它们都只读出所指定类型的消息。

> 在9.3节和11.3节中我们提到过，许多由内核定义的结构不能静态地初始化，因为Posix.1和Unix 98只保证在这样的结构中存在特定的成员。这两个标准不保证这些成员的顺序，更何况这样的结构还可以含有其他的非标准成员。然而在本程序中我们还是静态地初始化msgbuf结构，因为System V消息列保证该结构含有一个long类型的消息类型成员，后跟真正的数据。

bench/lat_svmsg.c

```
 1 #include    "unpipc.h"
 2 struct msgbuf   p2child = { 1, { 0 } };          /* type = 1 */
 3 struct msgbuf   child2p = { 2, { 0 } };          /* type = 2 */
 4 struct msgbuf   inbuf;

 5 void
 6 doit(int msgid)
 7 {
 8     Msgsnd(msgid, &p2child, 0, 0);
 9     if (Msgrcv(msgid, &inbuf, sizeof(inbuf.mtext), 2, 0) != 0)
10         err_quit("msgrcv error");
11 }

12 int
13 main(int argc, char **argv)
14 {
15     int     i, nloop, msgid;
16     pid_t   childpid;

17     if (argc != 2)
18         err_quit("usage: lat_svmsg <#loops>");
19     nloop = atoi(argv[1]);

20     msgid = Msgget(IPC_PRIVATE, IPC_CREAT | SVMSG_MODE);

21     if ( (childpid = Fork()) == 0) {
22         for ( ; ; ) {    /* child */
23             if (Msgrcv(msgid, &inbuf, sizeof(inbuf.mtext), 1, 0) != 0)
24                 err_quit("msgrcv error");
25             Msgsnd(msgid, &child2p, 0, 0);
26         }
27         exit(0);
28     }
29         /* parent */
30     doit(msgid);

31     Start_time();
32     for (i = 0; i < nloop; i++)
33         doit(msgid);
34     printf("latency: %.3f usec\n", Stop_time() / nloop);

35     Kill(childpid, SIGTERM);
36     Msgctl(msgid, IPC_RMID, NULL);
37     exit(0);
38 }
```

bench/lat_svmsg.c

图A-31　一个System V消息队列延迟测量程序

A.4.4　门延迟程序

482～484

图A-32给出了门API的延迟测量程序。子进程创建其中的门，然后给该门关联以server函数。父进程接着打开该门，并在一个循环中激活door_call。作为参数传递的是1个字节的数据，返回值则不存在。

bench/lat_door.c

```
 1 #include    "unpipc.h"
 2 void
 3 server(void *cookie, char *argp, size_t arg_size,
 4        door_desc_t *dp, size_t n_descriptors)
 5 {
 6     char    c;
 7     Door_return(&c, sizeof(char), NULL, 0);
 8 }
 9 int
10 main(int argc, char **argv)
11 {
12     int     i, nloop, doorfd, contpipe[2];
13     char    c;
14     pid_t   childpid;
15     door_arg_t  arg;
16     if (argc != 3)
17         err_quit("usage: lat_door <pathname> <#loops>");
18     nloop = atoi(argv[2]);
19     unlink(argv[1]);
20     Close(Open(argv[1], O_CREAT | O_EXCL | O_RDWR, FILE_MODE));
21     Pipe(contpipe);
22     if ( (childpid = Fork()) == 0) {
23         doorfd = Door_create(server, NULL, 0);
24         Fattach(doorfd, argv[1]);
25         Write(contpipe[1], &c, 1);
26         for ( ; ; )                 /* child = server */
27             pause();
28         exit(0);
29     }
30     arg.data_ptr = &c;              /* parent = client */
31     arg.data_size = sizeof(char);
32     arg.desc_ptr = NULL;
33     arg.desc_num = 0;
34     arg.rbuf = &c;
35     arg.rsize = sizeof(char);
36     if (Read(contpipe[0], &c, 1) != 1) /* wait for child to create */
37         err_quit("pipe read error");
38     doorfd = Open(argv[1], O_RDWR);
39     Door_call(doorfd, &arg);  /* once to start everything */
40     Start_time();
41     for (i = 0; i < nloop; i++)
42         Door_call(doorfd, &arg);
43     printf("latency: %.3f usec\n", Stop_time() / nloop);
44     Kill(childpid, SIGTERM);
45     unlink(argv[1]);
46     exit(0);
47 }
```

bench/lat_door.c

图A-32 门API的延迟测量程序

A.4.5　Sun RPC延迟程序

为测量Sun RPC API的延迟，我们编写一个客户和一个服务器两个程序（类似于测量Sun RPC带宽的做法）。我们使用同样的RPC说明书文件（图A-26），不过这次我们的客户程序调用空过程。回想习题16.11，我们知道该过程没有参数，也没有返回值，而这正好是测量延迟所需的。图A-33给出了客户程序。跟习题16.11的解答一样，我们必须通过直接调用clnt_call来调用空过程，在客户程序存根中不提供它的存根函数。

bench/lat_sunrpc_client.c

```
 1 #include    "unpipc.h"
 2 #include    "lat_sunrpc.h"

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     i, nloop;
 7     CLIENT  *cl;
 8     struct timeval  tv;

 9     if (argc != 4)
10         err_quit("usage: lat_sunrpc_client <hostname> <#loops> <protocol>");
11     nloop = atoi(argv[2]);

12     cl = Clnt_create(argv[1], BW_SUNRPC_PROG, BW_SUNRPC_VERS, argv[3]);

13     tv.tv_sec = 10;
14     tv.tv_usec = 0;
15     Start_time();
16     for (i = 0; i < nloop; i++) {
17         if (clnt_call(cl, NULLPROC, xdr_void, NULL,
18                       xdr_void, NULL, tv) != RPC_SUCCESS)
19             err_quit("%s", clnt_sperror(cl, argv[1]));
20     }
21     printf("latency: %.3f usec\n", Stop_time() / nloop);
22     exit(0);
23 }
```

bench/lat_sunrpc_client.c

图A-33　测量Sun RPC延迟的Sun RPC客户程序

我们使用图A-28中的服务器函数编译出服务器程序，不过该函数从来不被调用。既然是使用rpcgen来构造客户程序和服务器程序的，那么我们需要定义至少一个服务器过程，但是可不调用它。使用rpcgen的原因在于它自动产生带空过程的服务器程序的main函数，而我们正需要这些。

A.5　线程同步程序

为测量各种同步技术所需的时间，我们创建一定数目的线程（图A-6和图A-8给出的测量结果使用1个到5个线程），每个线程给处于共享内存区中的一个计数器加1很多次（一个很大的次数），各线程使用相应于当前测量的同步形式来协调对于该共享计数器的访问。

486

A.5.1　Posix互斥锁程序

图A-34给出了Posix互斥锁测量程序的全局变量和main函数。

bench/incr_pxmutex1.c

```
 1 #include    "unpipc.h"
 2 #define MAXNTHREADS 100
 3 int     nloop;
 4 struct {
 5   pthread_mutex_t   mutex;
 6   long  counter;
 7 } shared = {
 8   PTHREAD_MUTEX_INITIALIZER
 9 };
10 void    *incr(void *);
11 int
12 main(int argc, char **argv)
13 {
14     int     i, nthreads;
15     pthread_t   tid[MAXNTHREADS];
16     if (argc != 3)
17         err_quit("usage: incr_pxmutex1 <#loops> <#threads>");
18     nloop = atoi(argv[1]);
19     nthreads = min(atoi(argv[2]), MAXNTHREADS);
20         /* lock the mutex */
21     Pthread_mutex_lock(&shared.mutex);
22         /* create all the threads */
23     Set_concurrency(nthreads);
24     for (i = 0; i < nthreads; i++) {
25         Pthread_create(&tid[i], NULL, incr, NULL);
26     }
27         /* start the timer and unlock the mutex */
28     Start_time();
29     Pthread_mutex_unlock(&shared.mutex);
30         /* wait for all the threads */
31     for (i = 0; i < nthreads; i++) {
32         Pthread_join(tid[i], NULL);
33     }
34     printf("microseconds: %.0f usec\n", Stop_time());
35     if (shared.counter != nloop * nthreads)
36         printf("error: counter = %ld\n", shared.counter);
37     exit(0);
38 }
```

bench/incr_pxmutex1.c

图A-34 测量Posix互斥锁同步的全局变量和main函数

共享的数据

4~9 各线程间共享的数据由互斥锁本身和计数器构成。该互斥锁是静态初始化的。

给互斥锁上锁并创建线程

20~26 主线程在创建其他线程前给共享数据的互斥锁上锁，这样在所有线程都创建出来并且主线程释放该互斥锁之前，没有一个线程能够获取该互斥锁。主线程接着调用我们的set_concurrency函数，并创建出各个线程。每个线程执行接下来给出的incr函数。

启动定时器并释放互斥锁

27~36　一旦所有线程都已创建，主线程就启动定时器并释放互斥锁。它接下去等待所有线程结束，到时候就停止定时器，输出所计的总微秒数。

图A-35给出了每个线程执行的incr函数。

bench/incr_pxmutex1.c

```
39 void *
40 incr(void *arg)
41 {
42      int     i;

43      for (i = 0; i < nloop; i++) {
44          Pthread_mutex_lock(&shared.mutex);
45          shared.counter++;
46          Pthread_mutex_unlock(&shared.mutex);
47      }
48      return(NULL);
49 }
```

bench/incr_pxmutex1.c

图A-35　使用一个Posix互斥锁给一个共享的计数器加1

在临界区中给计数器加1

44~46　获取共享数据的互斥锁后给共享的计数器加1。接着释放该互斥锁。

A.5.2　读写锁程序

使用读写锁的程序由刚才使用Posix互斥锁的程序稍加修改而成。每个线程在给共享的计数器加1前必须获取该计数器读写锁中的写入锁。

实现第8章中讲述的Posix读写锁的系统并不多，Posix读写锁是Unix 98的一部分内容，Posix.1j工作组目前正在考虑它的标准化。本附录中给出的读写锁测量结果是在Solaris 2.6下做出的，它使用在rwlock(3T)手册页面中描述的Solaris读写锁。该实现提供了与提议中的读写锁同样的功能，以我们在第8章中给出的函数为接口来使用这些函数所需要的包裹函数非常简单。

Digital Unix 4.0B下的测量结果是使用Digital的线程无关服务读写锁做出的，这种读写锁在tis_rwlock手册页面中描述。我们不再给出为这些读写锁对图A-36和图A-37进行的简单修改。

488

图A-36给出了main函数，图A-37给出了incr函数。

bench/incr_rwlock1.c

```
 1 #include    "unpipc.h"
 2 #include    <synch.h>          /* Solaris header */

 3 void    Rw_wrlock(rwlock_t *rwptr);
 4 void    Rw_unlock(rwlock_t *rwptr);

 5 #define MAXNTHREADS 100

 6 int     nloop;

 7 struct {
 8   rwlock_t  rwlock;            /* the Solaris datatype */
 9   long  counter;
10 } shared;                      /* init to 0 -> USYNC_THREAD */
```

图A-36　读写锁同步测量程序的main函数

```
11 void    *incr(void *);

12 int
13 main(int argc, char **argv)
14 {
15      int     i, nthreads;
16      pthread_t   tid[MAXNTHREADS];

17      if (argc != 3)
18          err_quit("usage: incr_rwlock1 <#loops> <#threads>");
19      nloop = atoi(argv[1]);
20      nthreads = min(atoi(argv[2]), MAXNTHREADS);

21          /* obtain write lock */
22      Rw_wrlock(&shared.rwlock);

23          /* create all the threads */
24      Set_concurrency(nthreads);
25      for (i = 0; i < nthreads; i++) {
26          Pthread_create(&tid[i], NULL, incr, NULL);
27      }
28          /* start the timer and release the write lock */
29      Start_time();
30      Rw_unlock(&shared.rwlock);

31          /* wait for all the threads */
32      for (i = 0; i < nthreads; i++) {
33          Pthread_join(tid[i], NULL);
34      }
35      printf("microseconds: %.0f usec\n", Stop_time());
36      if (shared.counter != nloop * nthreads)
37          printf("error: counter = %ld\n", shared.counter);

38      exit(0);
39 }
```

bench/incr_rwlock1.c

图A-36（续）

```
40 void *
41 incr(void *arg)
42 {
43      int     i;

44      for (i = 0; i < nloop; i++) {
45          Rw_wrlock(&shared.rwlock);
46          shared.counter++;
47          Rw_unlock(&shared.rwlock);
48      }
49      return(NULL);
50 }
```

bench/incr_rwlock1.c

图A-37 使用一个读写锁给一个共享的计数器加1

A.5.3 Posix基于内存的信号量程序

我们既测量Posix基于内存的信号量，也测量Posix有名信号量。图A-39给出了基于内存的信号量的测量程序的main函数，图A-38给出了它的incr函数。

bench/incr_pxsem1.c

```
37 void *
38 incr(void *arg)
39 {
40     int     i;

41     for (i = 0; i < nloop; i++) {
42         Sem_wait(&shared.mutex);
43         shared.counter++;
44         Sem_post(&shared.mutex);
45     }
46     return(NULL);
47 }
```

bench/incr_pxsem1.c

图A-38　使用一个Posix基于内存的信号量给一个共享的计数器加1

bench/incr_pxsem1.c

```
 1 #include    "unpipc.h"
 2 #define MAXNTHREADS 100
 3 int     nloop;
 4 struct {
 5   sem_t     mutex;            /* the memory-based semaphore */
 6   long      counter;
 7 } shared;
 8 void    *incr(void *);
 9 int
10 main(int argc, char **argv)
11 {
12     int     i, nthreads;
13     pthread_t   tid[MAXNTHREADS];

14     if (argc != 3)
15         err_quit("usage: incr_pxsem1 <#loops> <#threads>");
16     nloop = atoi(argv[1]);
17     nthreads = min(atoi(argv[2]), MAXNTHREADS);

18         /* initialize memory-based semaphore to 0 */
19     Sem_init(&shared.mutex, 0, 0);

20         /* create all the threads */
21     Set_concurrency(nthreads);
22     for (i = 0; i < nthreads; i++) {
23         Pthread_create(&tid[i], NULL, incr, NULL);
24     }
25         /* start the timer and release the semaphore */
26     Start_time();
27     Sem_post(&shared.mutex);

28         /* wait for all the threads */
29     for (i = 0; i < nthreads; i++) {
30         Pthread_join(tid[i], NULL);
31     }
32     printf("microseconds: %.0f usec\n", Stop_time());
33     if (shared.counter != nloop * nthreads)
34         printf("error: counter = %ld\n", shared.counter);

35     exit(0);
36 }
```

bench/incr_pxsem1.c

图A-39　Posix基于内存信号量的同步的测量程序的main函数

18~19 创建一个值为0的信号量，而把给sem_init的第二个参数指定为0表明所创建的信号量将在调用进程的各个线程间共享。

20~27 创建出所有的线程后，主线程启动定时器并调用sem_post一次。

A.5.4 Posix有名信号量程序

图A-41给出了Posix有名信号量测量程序的main函数，图A-40给出了它的incr函数。

bench/incr_pxsem2.c

```
40 void *
41 incr(void *arg)
42 {
43      int     i;

44      for (i = 0; i < nloop; i++) {
45          Sem_wait(shared.mutex);
46          shared.counter++;
47          Sem_post(shared.mutex);
48      }
49      return(NULL);
50 }
```

bench/incr_pxsem2.c

图A-40 Posix有名信号量的同步的测量程序的main函数

bench/incr_pxsem2.c

```
 1 #include    "unpipc.h"

 2 #define MAXNTHREADS 100
 3 #define NAME    "incr_pxsem2"

 4 int     nloop;

 5 struct {
 6   sem_t *mutex;             /* pointer to the named semaphore */
 7   long  counter;
 8 } shared;

 9 void    *incr(void *);

10 int
11 main(int argc, char **argv)
12 {
13      int     i, nthreads;
14      pthread_t   tid[MAXNTHREADS];

15      if (argc != 3)
16          err_quit("usage: incr_pxsem2 <#loops> <#threads>");
17      nloop = atoi(argv[1]);
18      nthreads = min(atoi(argv[2]), MAXNTHREADS);

19          /* initialize named semaphore to 0 */
20      sem_unlink(Px_ipc_name(NAME));      /* error OK */
21      shared.mutex = Sem_open(Px_ipc_name(NAME), O_CREAT | O_EXCL, FILE_MODE,0);

22          /* create all the threads */
23      Set_concurrency(nthreads);
24      for (i = 0; i < nthreads; i++) {
25          Pthread_create(&tid[i], NULL, incr, NULL);
26      }
27          /* start the timer and release the semaphore */
```

图A-41 使用一个Posix有名信号量给一个共享的计数器加1

```
28      Start_time();
29      Sem_post(shared.mutex);

30          /* wait for all the threads */
31      for (i = 0; i < nthreads; i++) {
32          Pthread_join(tid[i], NULL);
33      }
34      printf("microseconds: %.0f usec\n", Stop_time());
35      if (shared.counter != nloop * nthreads)
36          printf("error: counter = %ld\n", shared.counter);
37      Sem_unlink(Px_ipc_name(NAME));

38      exit(0);
39  }
```

bench/incr_pxsem2.c

图A-41（续）

A.5.5　System V信号量程序

图A-42给出了System V信号量测量程序的main函数，图A-43给出它的incr函数。

bench/incr_svsem1.c

```
 1 #include    "unpipc.h"

 2 #define MAXNTHREADS 100

 3 int     nloop;

 4 struct {
 5   int   semid;
 6   long  counter;
 7 } shared;

 8 struct sembuf   postop, waitop;

 9 void    *incr(void *);

10 int
11 main(int argc, char **argv)
12 {
13      int     i, nthreads;
14      pthread_t   tid[MAXNTHREADS];
15      union semun arg;

16      if (argc != 3)
17          err_quit("usage: incr_svsem1 <#loops> <#threads>");
18      nloop = atoi(argv[1]);
19      nthreads = min(atoi(argv[2]), MAXNTHREADS);

20          /* create semaphore and initialize to 0 */
21      shared.semid = Semget(IPC_PRIVATE, 1, IPC_CREAT | SVSEM_MODE);
22      arg.val = 0;
23      Semctl(shared.semid, 0, SETVAL, arg);
24      postop.sem_num = 0;         /* and init the two semop() structures */
25      postop.sem_op  = 1;
26      postop.sem_flg = 0;
27      waitop.sem_num = 0;
28      waitop.sem_op  = -1;
29      waitop.sem_flg = 0;
```

图A-42　测量System V信号量的同步的程序的main函数

```
30          /* create all the threads */
31      Set_concurrency(nthreads);
32      for (i = 0; i < nthreads; i++) {
33          Pthread_create(&tid[i], NULL, incr, NULL);
34      }
35          /* start the timer and release the semaphore */
36      Start_time();
37      Semop(shared.semid, &postop, 1);        /* up by 1 */
38          /* wait for all the threads */
39      for (i = 0; i < nthreads; i++) {
40          Pthread_join(tid[i], NULL);
41      }
42      printf("microseconds: %.0f usec\n", Stop_time());
43      if (shared.counter != nloop * nthreads)
44          printf("error: counter = %ld\n", shared.counter);
45      Semctl(shared.semid, 0, IPC_RMID);
46      exit(0);
47  }
```

bench/incr_svsem1.c

图A-42（续）

bench/incr_svsem1.c

```
48  void *
49  incr(void *arg)
50  {
51      int     i;
52      for (i = 0; i < nloop; i++) {
53          Semop(shared.semid, &waitop, 1);
54          shared.counter++;
55          Semop(shared.semid, &postop, 1);
56      }
57      return(NULL);
58  }
```

bench/incr_svsem1.c

图A-43 使用一个System V信号量给一个共享的计数器加1

20~23 创建一个仅有一个成员的信号量集，并把它的值初始化为0。

24~29 初始化两个semop结构，一个用于挂出该信号量，另一个用于等待该信号量。注意这两个结构的sem_flg成员均为0，也就是说没有指定SEM_UNDO标志。

A.5.6 带 SEM_UNDO 特性的 System V 信号量程序

测量具有SEM_UNDO特性的System V信号量的程序与图A-42相比的唯一差别是：它把两个semop结构的sem_flg成员设置成SEM_UNDO而不是0。我们不再给出这个简单修改后的版本。

A.5.7 fcntl 记录上锁程序

最后一个程序使用fcntl记录上锁提供同步。图A-45给出了它的main函数。该程序只在指定单个线程时才会成功地运行，因为fcntl锁是在不同进程间而不是单个进程的不同线程间共享的。当指定多个线程时，每个线程总能获取所请求的锁（也就是说writew_lock调用从不阻塞，因为调用进程早已拥有该锁），因而计数器的最终值是错误的。

18~22　待创建并随后用于上锁的文件的路径名是作为一个命令行参数指定的。这样允许我们就驻留在不同文件系统上的文件测量这个程序。我们预期当该文件处在某个通过NFS安装的文件系统上时，该程序的运行变慢，这种情况要求两个系统（NFS客户主机和NFS服务器主机）都支持NFS记录上锁。

图A-44给出了使用记录上锁的incr函数。

bench/incr_fcntl1.c

```
44 void *
45 incr(void *arg)
46 {
47      int     i;

48      for (i = 0; i < nloop; i++) {
49          Writew_lock(shared.fd, 0, SEEK_SET, 0);
50          shared.counter++;
51          Un_lock(shared.fd, 0, SEEK_SET, 0);
52      }
53      return(NULL);
54 }
```

bench/incr_fcntl1.c

490 ~ 495

图A-44　使用fcntl记录上锁给一个共享的计数器加1

bench/incr_fcntl1.c

```
 4 #include    "unpipc.h"

 5 #define MAXNTHREADS 100

 6 int     nloop;

 7 struct {
 8   int       fd;
 9   long  counter;
10 } shared;

11 void    *incr(void *);

12 int
13 main(int argc, char **argv)
14 {
15      int     i, nthreads;
16      char    *pathname;
17      pthread_t   tid[MAXNTHREADS];

18      if (argc != 4)
19          err_quit("usage: incr_fcntl1 <pathname> <#loops> <#threads>");
20      pathname = argv[1];
21      nloop = atoi(argv[2]);
22      nthreads = min(atoi(argv[3]), MAXNTHREADS);

23          /* create the file and obtain write lock */
24      shared.fd = Open(pathname, O_RDWR | O_CREAT | O_TRUNC, FILE_MODE);
25      Writew_lock(shared.fd, 0, SEEK_SET, 0);

26          /* create all the threads */
27      Set_concurrency(nthreads);
28      for (i = 0; i < nthreads; i++) {
29          Pthread_create(&tid[i], NULL, incr, NULL);
30      }
31          /* start the timer and release the write lock */
32      Start_time();
```

图A-45　fcntl记录上锁测量程序的main函数

```
33      Un_lock(shared.fd, 0, SEEK_SET, 0);

34          /* wait for all the threads */
35      for (i = 0; i < nthreads; i++) {
36          Pthread_join(tid[i], NULL);
37      }
38      printf("microseconds: %.0f usec\n", Stop_time());
39      if (shared.counter != nloop * nthreads)
40          printf("error: counter = %ld\n", shared.counter);
41      Unlink(pathname);

42      exit(0);
43  }
```

bench/incr_fcntl1.c

图A-45（续）

496

A.6 进程同步程序

上一节给出的程序中，多个线程间共享一个计数器比较简单：我们只需把该计数器作为一个全局变量存放。我们现在修改这些程序以提供不同进程间的同步。

为在一个父进程和它的各个子进程间共享该计数器，我们把它存放在由图A-46给出的my_shm函数分配的共享内存区中。

lib/my_shm.c

```
 1 #include    "unpipc.h"

 2 void *
 3 my_shm(size_t nbytes)
 4 {
 5     void    *shared;

 6 #if     defined(MAP_ANON)
 7     shared = mmap(NULL, nbytes, PROT_READ | PROT_WRITE,
 8                   MAP_ANON | MAP_SHARED, -1, 0);

 9 #elif   defined(HAVE_DEV_ZERO)
10     int     fd;

11         /* memory map /dev/zero */
12     if ( (fd = open("/dev/zero", O_RDWR)) == -1)
13         return(MAP_FAILED);
14     shared = mmap(NULL, nbytes, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
15     close(fd);

16 # else
17 # error cannot determine what type of anonymous shared memory to use
18 # endif
19     return(shared);                 /* MAP_FAILED on error */
20 }
```

lib/my_shm.c

图A-46 给一个父进程和它的各个子进程创建一个共享内存区

如果系统支持MAP_ANON标志（12.4节），我们就使用它，否则我们把/dev/zero（12.5节）映射到内存。

进一步的修改依赖于同步类型以及调用fork时底层支撑数据类型发生的变化。我们已在10.12节讲述过其中一些细节。

- Posix互斥锁：互斥锁必须存放在共享内存区中（跟共享的计数器在一起），而且初始化互斥锁时必须设置PTHREAD_PROCESS_SHARED属性。我们稍后给出这个程序的代码。
- Posix读写锁：读写锁必须存放在共享内存区中（跟共享的计数器在一起），而且初始化读写锁时必须设置PTHREAD_PROCESS_SHARED属性。

497

- Posix基于内存的信号量：信号量必须存放在共享内存区中（跟共享的计数器在一起），而且sem_init的第二个参数必须为1，从而指定该信号量是在进程间共享的。
- Posix有名信号量：我们既可以让父进程和每个子进程都调用sem_open，也可以只让父进程调用sem_open，因为我们知道该信号量将通过fork由子进程共享。
- System V信号量：不必编写任何特殊的代码，因为这种信号量总是能够在进程间共享。子进程只需知道父进程所创建信号量的标识符。
- fcntl记录上锁：不必编写任何特殊的代码，因为描述符可通过fork由子进程共享。

我们只给出Posix互斥锁程序的代码。

Posix互斥锁程序

Posix互斥锁程序的main函数使用一个Posix互斥锁来提供进程间的同步形式，如图A-48所示。图A-47给出了它的incr函数。

bench/incr_pxmutex5.c

```
46 void *
47 incr(void *arg)
48 {
49     int     i;

50     for (i = 0; i < nloop; i++) {
51         Pthread_mutex_lock(&shared->mutex);
52         shared->counter++;
53         Pthread_mutex_unlock(&shared->mutex);
54     }
55     return(NULL);
56 }
```

bench/incr_pxmutex5.c

图A-47　测量进程间Posix互斥锁上锁的incr函数

bench/incr_pxmutex5.c

```
 1 #include    "unpipc.h"

 2 #define MAXNPROC    100

 3 int     nloop;

 4 struct shared {
 5   pthread_mutex_t   mutex;
 6   long  counter;
 7 } *shared;              /* pointer; actual structure in shared memory */

 8 void    *incr(void *);

 9 int
10 main(int argc, char **argv)
11 {
12     int     i, nprocs;
13     pid_t   childpid[MAXNPROC];
14     pthread_mutexattr_t mattr;
```

图A-48　进程间Posix互斥锁上锁测量程序的main函数

```
15      if (argc != 3)
16          err_quit("usage: incr_pxmutex5 <#loops> <#processes>");
17      nloop = atoi(argv[1]);
18      nprocs = min(atoi(argv[2]), MAXNPROC);

19          /* get shared memory for parent and children */
20      shared = My_shm(sizeof(struct shared));

21          /* initialize the mutex and lock it */
22      Pthread_mutexattr_init(&mattr);
23      Pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
24      Pthread_mutex_init(&shared->mutex, &mattr);
25      Pthread_mutexattr_destroy(&mattr);
26      Pthread_mutex_lock(&shared->mutex);

27          /* create all the children */
28      for (i = 0; i < nprocs; i++) {
29          if ( (childpid[i] = Fork()) == 0) {
30              incr(NULL);
31              exit(0);
32          }
33      }
34          /* parent: start the timer and unlock the mutex */
35      Start_time();
36      Pthread_mutex_unlock(&shared->mutex);

37          /* wait for all the children */
38      for (i = 0; i < nprocs; i++) {
39          Waitpid(childpid[i], NULL, 0);
40      }
41      printf("microseconds: %.0f usec\n", Stop_time());
42      if (shared->counter != nloop * nprocs)
43          printf("error: counter = %ld\n", shared->counter);

44      exit(0);
45  }
```

bench/incr_pxmutex5.c

图A-48（续）

19~20 既然使用多个进程（一个父进程的多个子进程），我们就必须把shared结构置于共享内存区中。我们调用图A-46给出的my_shm函数做到这一点。

21~26 既然互斥锁处于共享内存区中，我们就不能静态地对它初始化，而必须在设置一个PTHREAD_PROCESS_SHARED属性对象后调用pthread_mutex_init初始化它。该互斥锁一开始是锁着的。

27~36 创建出所有子进程后，父进程启动定时器，并给互斥锁解锁。

37~43 父进程等待所有子进程都结束，然后停止定时器。

498 ~ 499

附录 B

线 程 入 门

B.1 概述

本附录汇总基本的Posix线程函数。在传统的Unix模型中，当一个进程需要由另一个实体来执行某件事时，它就fork一个子进程，让子进程去进行处理。举例来说，Unix下的大多数网络服务器程序就是这么编写的。

尽管这种模式已成功地使用了很多年，但是fork仍然暴露出了以下问题。

- fork的开销很大。内存映像要从父进程复制到子进程，所有描述符要在子进程中复制一份，等等。当前的系统实现使用一种称为写时复制（copy-on-write）的技术，可避免父进程数据空间一开始就向子进程复制，直到子进程确实需要自己的副本为止。尽管有这种优化技术，fork的开销仍然很大。
- fork子进程后，需要用进程间通信（IPC）在父子进程之间传递信息。fork之前由父进程准备好的信息容易传递，因为子进程是从父进程的数据空间及所有描述符的一个副本开始运行的。但是从子进程返回信息给父进程却颇费周折。

线程有助于解决这两个问题。线程有时称为轻权进程（lightweight process），因为线程比进
501 程“权轻”。[1]也就是说，创建线程可能比创建进程快10～100倍。

一个进程内的所有线程共享同一个全局内存空间。这使得线程间很容易共享信息，但是这种容易性也带来了同步（synchronization）问题。一个进程内的所有线程不光共享全局变量，以下信息也是它们所共享的：

- 进程指令；
- 大多数数据；
- 打开的文件（例如描述符）；
- 信号处理程序和信号处置；
- 当前工作目录；
- 用户ID和组ID。

但是下列信息却是特定于每个线程的：

- 线程ID；
- 寄存器集合，包括程序计数器和栈指针；
- 栈（用于存放局部变量和返回地址）；
- errno；
- 信号掩码；
- 优先级。

① 线程和轻权进程实际上是不同的概念。Solaris 2.x下实际存在内核线程、轻权进程和用户线程三个概念。——译者注

B.2 基本线程函数：创建和终止

本节讨论5个基本线程函数。

B.2.1 pthread_create 函数

当一个程序由exec启动执行时，系统将创建一个称为*初始线程*（initial thread）或*主线程*（main thread）的单个线程。其余线程则由pthread_create函数创建。

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*func)(void *), void *arg);
```

返回：若成功则为0，若出错则为正的Exxx值

一个进程内的各个线程是由线程*ID*（thread ID）标识的，这些线程的数据类型为pthread_t。如果新的线程创建成功，它的ID就通过*tid*指针返回。

每个线程有多个属性（attribute）：优先级、初始栈大小、是否应该是一个守护线程，等等。当创建线程时，我们可通过初始化一个pthread_attr_t变量来指定这些属性以覆盖默认值。我们通常采用默认值，这种情况下，我们只需把*attr*参数指定为一个空指针。

最后，当创建一个线程时，我们应指定一个它将执行的函数，称为它的*线程启动函数*（thread start function）。这个线程以调用该函数开始，以后或者显式地终止（调用pthread_exit），或者隐式地终止（让该函数返回）。该函数的地址作为*func*参数指定，该函数唯一的调用参数则是一个指针*arg*。如果需要给该函数传递多个参数，我们就得把它们打包成一个结构，然后将其地址作为这个唯一的参数，传递给线程启动函数。 502

注意*func*和*arg*的声明。func函数接受一个通用指针参数（void *），返回一个通用指针（void *）。这就使得我们可以给线程传递一个指针（指向任何我们想要指向的东西），再由线程返回一个指针（同样地指向任何我们想要指向的东西）。

Pthread函数的返回值有两种：成功时返回0，出错时返回非零。与出错时返回-1，并置errno为某个正值的大多数系统函数不同，Pthread函数的返回值是正值的出错指示。例如，如果pthread_create函数因为超过了系统线程数目的限制而不能创建新线程，那么它的返回值将是EAGAIN。Pthread函数并不设置errno。成功时返回0，出错时返回非零的约定不成问题，因为在<sys/errno.h>头文件中的所有Exxx值都大于0。0值永远不会赋给任何一个Exxx常值。

B.2.2 pthread_join 函数

我们可以调用pthread_join等待一个线程终止。把线程和Unix进程相比，pthread_create类似于fork，pthread_join则类似于waitpid。

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **status);
```

返回：若成功则为0，若出错则为正的Exxx值

我们必须指定要等待的线程的*tid*。遗憾的是，不是任意一个线程的终止都能等待（类似于给waitpid的进程ID参数传递-1值的情况）。

如果*status*指针非空，那么所等待线程的返回值（指向某个对象的指针）将存放在*status*指向的位置。

B.2.3 pthread_self 函数

每个线程都有在某个给定的进程内标识自身的一个ID。这个线程ID由pthread_create函数返回，我们已看到pthread_join函数用到它。一个线程使用pthread_self取得自己的线程ID。

```
#include <pthread.h>
pthread_t pthread_self(void);
                                                    返回：调用线程的线程ID
```

503

把线程和Unix进程相比较，pthread_self类似于getpid。

B.2.4 pthread_detach 函数

线程或者是可汇合的（joinable），或者是脱离的（detached）。当可汇合的线程终止时，其线程ID和退出状态将保留，直到另外一个线程调用pthread_join。脱离的线程则像守护进程：当它终止时，所有的资源都将释放，因此我们不能等待它终止。如果一个线程需要知道另一个线程的终止时间，那就最好保留第二个线程的可汇合性。

pthread_detach函数将指定的线程变为脱离的。

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
                                    返回：若成功则为0，若出错则为正的Exxx值
```

该函数通常由想让自己脱离的线程使用，例如：

```
pthread_detach(pthread_self());
```

B.2.5 pthread_exit 函数

终止一个线程的方法之一是调用pthread_exit。

```
#include <pthread.h>
void pthread_exit(void *status);
                                                    不返回到调用者
```

如果该线程未脱离，那么其线程ID和退出状态将一直保留到调用进程内的另外某个线程调用pthread_join为止。

指针*status*不能指向局部于调用线程的对象（例如该线程的启动函数中的某个自动变量），因为该线程终止时那个对象也消失了。

使一个线程终止还有其他两种方法。

- 启动该线程的函数（pthread_create的第三个参数）可以调用return。既然该函数必须声明成返回一个void指针，该返回值便是该线程的终止状态。
- 如果本进程的main函数返回，或者某个线程调用了exit或_exit，那么该进程将立即终止，它的仍在运行的任意线程也都将终止。

504

附录 C

杂凑的源代码

C.1 unpipc.h 头文件

本书正文中几乎每个程序都包含了我们的unpipc.h头文件，它如图C-1所示。该头文件包含了大多数网络程序需要的所有标准系统头文件以及一些普通的系统头文件。它还定义了诸如MAXLINE等常值和我们已在正文中定义过的函数（例如px_ipc_name），以及所用到的所有包裹函数的ANSI C函数原型。不过这儿没有给出这些原型。

lib/unpipc.h

```
 1 /* Our own header.  Tabs are set for 4 spaces, not 8 */

 2 #ifndef    __unpipc_h
 3 #define    __unpipc_h

 4 #include   "../config.h"       /* configuration options for current OS */
 5                                /* "../config.h" is generated by configure */

 6 /* If anything changes in the following list of #includes, must change
 7    ../aclocal.m4 and ../configure.in also, for configure's tests. */

 8 #include   <sys/types.h>        /* basic system data types */
 9 #include   <sys/time.h>         /* timeval{} for select() */
10 #include   <time.h>             /* timespec{} for pselect() */
11 #include   <errno.h>
12 #include   <fcntl.h>            /* for nonblocking */
13 #include   <limits.h>           /* PIPE_BUF */
14 #include   <signal.h>
15 #include   <stdio.h>
16 #include   <stdlib.h>
17 #include   <string.h>
18 #include   <sys/stat.h>         /* for S_xxx file mode constants */
19 #include   <unistd.h>
20 #include   <sys/wait.h>

21 #ifdef HAVE_MQUEUE_H
22 # include  <mqueue.h>           /* Posix message queues */
23 #endif

24 #ifdef HAVE_SEMAPHORE_H
25 # include  <semaphore.h>        /* Posix semaphores */

26 #ifndef SEM_FAILED
27 #define SEM_FAILED ((sem_t *)(-1))
28 #endif

29 #endif

30 #ifdef  HAVE_SYS_MMAN_H
```

图C-1 我们的unpipc.h头文件

```
31 # include  <sys/mman.h>      /* Posix shared memory */
32 #endif

33 #ifndef MAP_FAILED
34 #define MAP_FAILED ((void *)(-1))
35 #endif

36 #ifdef HAVE_SYS_IPC_H
37 # include  <sys/ipc.h>       /* System V IPC */
38 #endif

39 #ifdef HAVE_SYS_MSG_H
40 # include  <sys/msg.h>       /* System V message queues */
41 #endif

42 #ifdef HAVE_SYS_SEM_H
43 #ifdef __bsdi__
44 #undef HAVE_SYS_SEM_H        /* hack: BSDI's semctl() prototype is wrong */
45 #else
46 # include  <sys/sem.h>       /* System V semaphores */
47 #endif

48 #ifndef HAVE_SEMUN_UNION
49 union semun {                /* define union for semctl() */
50     int      val;
51     struct semid_ds *buf;
52     unsigned short  *array;
53 };
54 #endif
55 #endif /* HAVE_SYS_SEM_H */

56 #ifdef HAVE_SYS_SHM_H
57 # include  <sys/shm.h>       /* System V shared memory */
58 #endif

59 #ifdef HAVE_SYS_SELECT_H
60 # include  <sys/select.h>    /* for convenience */
61 #endif

62 #ifdef HAVE_POLL_H
63 # include  <poll.h>          /* for convenience */
64 #endif

65 #ifdef HAVE_STROPTS_H
66 # include  <stropts.h>       /* for convenience */
67 #endif

68 #ifdef HAVE_STRINGS_H
69 # include  <strings.h>       /* for convenience */
70 #endif

71 /* Next three headers are normally needed for socket/file ioctl's:
72  * <sys/ioctl.h>, <sys/filio.h>, and <sys/sockio.h>.
73  */
74 #ifdef HAVE_SYS_IOCTL_H
75 # include  <sys/ioctl.h>
76 #endif
77 #ifdef HAVE_SYS_FILIO_H
78 # include  <sys/filio.h>
79 #endif
80 #ifdef HAVE_PTHREAD_H
```

506

图C-1（续）

```
 81 # include  <pthread.h>
 82 #endif

 83 #ifdef HAVE_DOOR_H
 84 # include  <door.h>              /* Solaris doors API */
 85 #endif

 86 #ifdef HAVE_RPC_RPC_H
 87 #ifdef _PSX4_NSPACE_H_TS         /* Digital Unix 4.0b hack, hack, hack */
 88 #undef SUCCESS
 89 #endif
 90 # include  <rpc/rpc.h>           /* Sun RPC */
 91 #endif

 92 /* Define bzero() as a macro if it's not in standard C library. */
 93 #ifndef HAVE_BZERO
 94 #define bzero(ptr,n)         memset(ptr, 0, n)
 95 #endif

 96 /* Posix.1g requires that an #include of <poll.h> DefinE INFTIM, but many
 97    systems still DefinE it in <sys/stropts.h>.  We don't want to include
 98    all the streams stuff if it's not needed, so we just DefinE INFTIM here.
 99    This is the standard value, but there's no guarantee it is -1. */
100 #ifndef INFTIM
101 #define INFTIM          (-1)     /* infinite poll timeout */
102 #ifdef  HAVE_POLL_H
103 #define INFTIM_UNPH              /* tell unpxti.h we defined it */
104 #endif
105 #endif

106 /* Miscellaneous constants */
107 #ifndef PATH_MAX                 /* should be in <limits.h> */
108 #define PATH_MAX   1024          /* max # of characters in a pathname */
109 #endif

110 #define MAX_PATH   1024
111 #define MAXLINE    4096      /* max text line length */
112 #define BUFFSIZE   8192      /* buffer size for reads and writes */

113 #define FILE_MODE  (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
114                    /* default permissions for new files */
115 #define DIR_MODE   (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)
116                    /* default permissions for new directories */

117 #define SVMSG_MODE (MSG_R | MSG_W | MSG_R>>3 | MSG_R>>6)
118                    /* default permissions for new SV message queues */
119 #define SVSEM_MODE (SEM_R | SEM_A | SEM_R>>3 | SEM_R>>6)
120                    /* default permissions for new SV semaphores */
121 #define SVSHM_MODE (SHM_R | SHM_W | SHM_R>>3 | SHM_R>>6)
122                    /* default permissions for new SV shared memory */

123 typedef void Sigfunc(int);     /* for signal handlers */

124 #ifdef HAVE_SIGINFO_T_STRUCT
125 typedef void Sigfunc_rt(int, siginfo_t *, void *);
126 #endif

127 #define min(a,b)    ((a) < (b) ? (a) : (b))
128 #define max(a,b)    ((a) > (b) ? (a) : (b))

129 #ifndef HAVE_TIMESPEC_STRUCT
```

507

图C-1（续）

```
130 struct timespec {
131   time_t    tv_sec;     /* seconds */
132   long      tv_nsec;    /* and nanoseconds */
133 };
134 #endif

135 /*
136  * In our wrappers for open(), mq_open(), and sem_open() we handle the
137  * optional arguments using the va_XXX() macros.  But one of the optional
138  * arguments is of type "mode_t" and this breaks under BSD/OS because it
139  * uses a 16-bit integer for this datatype.  But when our wrapper function
140  * is called, the compiler expands the 16-bit short integer to a 32-bit
141  * integer.  This breaks our call to va_arg().  All we can do is the
142  * following hack.  Other systems in addition to BSD/OS might have this
143  * problem too ...
144  */

145 #ifdef __bsdi__
146 #define va_mode_t  int
147 #else
148 #define va_mode_t  mode_t
149 #endif

150         /* our record locking macros */
151 #define read_lock(fd, offset, whence, len) \
152             lock_reg(fd, F_SETLK, F_RDLCK, offset, whence, len)
153 #define readw_lock(fd, offset, whence, len) \
154             lock_reg(fd, F_SETLKW, F_RDLCK, offset, whence, len)
155 #define write_lock(fd, offset, whence, len) \
156             lock_reg(fd, F_SETLK, F_WRLCK, offset, whence, len)
157 #define writew_lock(fd, offset, whence, len) \
158             lock_reg(fd, F_SETLKW, F_WRLCK, offset, whence, len)
159 #define un_lock(fd, offset, whence, len) \
160             lock_reg(fd, F_SETLK, F_UNLCK, offset, whence, len)
508
161 #define is_read_lockable(fd, offset, whence, len) \
162             lock_test(fd, F_RDLCK, offset, whence, len)
163 #define is_write_lockable(fd, offset, whence, len) \
164             lock_test(fd, F_WRLCK, offset, whence, len)
```

lib/unpipc.h

图C-1（续）

C.2　config.h头文件

本书中使用了GNU autoconf工具以辅助所有源代码的移植。它可以从ftp://prep.ai.mit.edu/pub/gnu获取。这个工具生成一个名为configure的shell脚本，在把软件下载到你的系统后你必须运行该脚本。这个脚本确定你的Unix系统所提供的特性：支持System V消息队列吗？定义unit8_t数据类型了吗？提供gethostname函数了吗？等等，最终生成一个名为config.h的头文件。它是上节介绍的unpipc.h头文件中包含的第一个头文件。图C-2给出了在使用gcc编译器的前提下，在Solaris 2.6系统上生成的config.h头文件。

其中从第1列开始以#define开头的行代表系统提供了的特性。注释掉并且含有#undef的行代表系统没有提供的特性。

sparc-sun-solaris2.6/config.h

```
 1 /* config.h.  Generated automatically by configure.  */
 2 /* Define the following if you have the corresponding header */
 3 #define CPU_VENDOR_OS "sparc-sun-solaris2.6"
 4 #define HAVE_DOOR_H 1                 /* <door.h> */
 5 #define HAVE_MQUEUE_H 1               /* <mqueue.h> */
 6 #define HAVE_POLL_H 1                 /* <poll.h> */
 7 #define HAVE_PTHREAD_H 1              /* <pthread.h> */
 8 #define HAVE_RPC_RPC_H 1              /* <rpc/rpc.h> */
 9 #define HAVE_SEMAPHORE_H 1            /* <semaphore.h> */
10 #define HAVE_STRINGS_H 1              /* <strings.h> */
11 #define HAVE_SYS_FILIO_H 1            /* <sys/filio.h> */
12 #define HAVE_SYS_IOCTL_H 1            /* <sys/ioctl.h> */
13 #define HAVE_SYS_IPC_H 1              /* <sys/ipc.h> */
14 #define HAVE_SYS_MMAN_H 1             /* <sys/mman.h> */
15 #define HAVE_SYS_MSG_H 1              /* <sys/msg.h> */
16 #define HAVE_SYS_SEM_H 1              /* <sys/sem.h> */
17 #define HAVE_SYS_SHM_H 1              /* <sys/shm.h> */
18 #define HAVE_SYS_SELECT_H 1           /* <sys/select.h> */
19 /* #undef   HAVE_SYS_SYSCTL_H */     /* <sys/sysctl.h> */
20 #define HAVE_SYS_TIME_H 1             /* <sys/time.h> */
21 /* Define if we can include <time.h> with <sys/time.h> */
22 #define TIME_WITH_SYS_TIME 1
23 /* Define the following if the function is provided */
24 #define HAVE_BZERO 1
25 #define HAVE_FATTACH 1
26 #define HAVE_POLL 1
27 /* #undef   HAVE_PSELECT */
28 #define HAVE_SIGWAIT 1
29 #define HAVE_VALLOC 1
30 #define HAVE_VSNPRINTF 1
31 /* Define the following if the function prototype is in a header */
32 #define HAVE_GETHOSTNAME_PROTO 1     /* <unistd.h> */
33 #define HAVE_GETRUSAGE_PROTO 1       /* <sys/resource.h> */
34 /* #undef   HAVE_PSELECT_PROTO */    /* <sys/select.h> */
35 #define HAVE_SHM_OPEN_PROTO 1        /* <sys/mman.h> */
36 #define HAVE_SNPRINTF_PROTO 1        /* <stdio.h> */
37 #define HAVE_THR_SETCONCURRENCY_PROTO 1      /* <thread.h> */
38 /* Define the following if the structure is defined. */
39 #define HAVE_SIGINFO_T_STRUCT 1      /* <signal.h> */
40 #define HAVE_TIMESPEC_STRUCT 1       /* <time.h> */
41 /* #undef   HAVE_SEMUN_UNION */      /* <sys/sem.h> */
42 /* Devices */
43 #define HAVE_DEV_ZERO 1
44 /* Define the following to the appropriate datatype, if necessary */
45 /* #undef   int8_t */                /* <sys/types.h> */
46 /* #undef   int16_t */               /* <sys/types.h> */
47 /* #undef   int32_t */               /* <sys/types.h> */
48 /* #undef   uint8_t */               /* <sys/types.h> */
49 /* #undef   uint16_t */              /* <sys/types.h> */
50 /* #undef   uint32_t */              /* <sys/types.h> */
51 /* #undef   size_t */                /* <sys/types.h> */
52 /* #undef   ssize_t */               /* <sys/types.h> */
53 #define POSIX_IPC_PREFIX "/"
54 #define RPCGEN_ANSIC 1        /* defined if rpcgen groks -C option */
```

sparc-sun-solaris2.6/config.h

图C-2 Solaris 2.6上的config.h头文件

509

C.3　标准错误处理函数

我们定义了一组自己的错误处理函数，它们用在整本书中以处理出错情况。定义一组自己的错误处理函数的原因在于，我们可以用一行简单的C代码编写错误处理过程，就像如下所示：

```
if (出错条件)
    err_sys(带任意数目的参数的printf格式串);
```

而不是如下所示用多行：

```
if (出错条件)
    char   buff[200];
    snprintf(buff, sizeof(buff), 带任意数目的参数的printf格式串);
    perror(buff);
    exit(1);
}
```

510

我们的错误处理函数使用来自ANSI C的可变长度参数表机制。具体细节参见［Kernighan and Ritchie 1988］的7.3节。

图C-3列出了各个错误处理函数之间的差异。如果全局整数daemon_proc非零，那么当前出错消息将按指定的级别传递给syslog（关于syslog的细节参见UNPv1第12章）；否则，当前出错消息输出到标准错误输出。

函　数	strerror(errno)?	结束语句	syslog级别
err_dump	是	abort();	LOG_ERR
err_msg	否	return;	LOG_INFO
err_quit	否	exit(1);	LOG_ERR
err_ret	是	return;	LOG_INFO
err_sys	是	exit(1);	LOG_ERR

图C-3　标准错误处理函数汇总

图C-4给出了图C-3所示的5个函数。

lib/error.c

```
 1 #include    "unpipc.h"

 2 #include    <stdarg.h>        /* ANSI C header file */
 3 #include    <syslog.h>        /* for syslog() */

 4 int     daemon_proc;          /* set nonzero by daemon_init() */

 5 static void err_doit(int, int, const char *, va_list);

 6 /* Nonfatal error related to a system call.
 7  * Print a message and return. */

 8 void
 9 err_ret(const char *fmt, ...)
10 {
11     va_list  ap;

12     va_start(ap, fmt);
13     err_doit(1, LOG_INFO, fmt, ap);
14     va_end(ap);
15     return;
```

图C-4　我们的标准错误处理函数

```
16 }
17 /* Fatal error related to a system call.
18  * Print a message and terminate. */
19 void
20 err_sys(const char *fmt, ...)
21 {
22     va_list  ap;
23     va_start(ap, fmt);
24     err_doit(1, LOG_ERR, fmt, ap);
25     va_end(ap);
26     exit(1);
27 }
28 /* Fatal error related to a system call.
29  * Print a message, dump core, and terminate. */
30 void
31 err_dump(const char *fmt, ...)
32 {
33     va_list  ap;
34     va_start(ap, fmt);
35     err_doit(1, LOG_ERR, fmt, ap);
36     va_end(ap);
37     abort();                    /* dump core and terminate */
38     exit(1);                    /* shouldn't get here */
39 }
40 /* Nonfatal error unrelated to a system call.
41  * Print a message and return. */
42 void
43 err_msg(const char *fmt, ...)
44 {
45     va_list     ap;
46     va_start(ap, fmt);
47     err_doit(0, LOG_INFO, fmt, ap);
48     va_end(ap);
49     return;
50 }
51 /* Fatal error unrelated to a system call.
52  * Print a message and terminate. */
53 void
54 err_quit(const char *fmt, ...)
55 {
56     va_list  ap;
57     va_start(ap, fmt);
58     err_doit(0, LOG_ERR, fmt, ap);
59     va_end(ap);
60     exit(1);
61 }
62 /* Print a message and return to caller.
63  * Caller specifies "errnoflag" and "level". */
```

511

图C-4（续）

```
64 static void
65 err_doit(int errnoflag, int level, const char *fmt, va_list ap)
66 {
67     int     errno_save, n;
68     char    buf[MAXLINE];
69     errno_save = errno;  /* value caller might want printed */
70 #ifdef  HAVE_VSNPRINTF
71     vsnprintf(buf, sizeof(buf), fmt, ap);       /* this is safe */
72 #else
73     vsprintf(buf, fmt, ap);                     /* this is not safe */
512 74 #endif
75     n = strlen(buf);
76     if (errnoflag)
77         snprintf(buf+n, sizeof(buf)-n, ": %s", strerror(errno_save));
78     strcat(buf, "\n");
79     if (daemon_proc) {
80         syslog(level, buf);
81     } else {
82         fflush(stdout);           /* in case stdout and stderr are the same */
83         fputs(buf, stderr);
84         fflush(stderr);
85     }
86     return;
513 87 }
```

lib/error.c

图C-4（续）

附录 D

精选习题解答

第 1 章

1.1 这两个进程只需给open函数指定O_APPEND标志，或者给fopen函数指定添加模式。内核保证每次write都将新的数据添加到该文件的末尾。这是可指定的最容易的文件同步形式（APUE第60~61页[①]对此有具体的讨论）。当更新文件中已有的数据时，同步问题会变得复杂起来，数据库系统中的情况就是这样。

1.2 典型的定义类似如下：

```
#ifdef  _REENTRANT
#define errno   (*_errno())
#else
extern int  errno;
#endif
```

如果_REENTRANT已定义，引用errno时就调用一个名为_errno的函数，该函数返回调用线程的errno变量的地址。该变量可能是作为线程特定数据（UNPv1的23.5节[②]）存储的。如果_REENTRANT未定义，errno就是一个全局int变量。

第 2 章

2.1 这两位能改变待运行程序的有效用户ID和/或有效组ID。2.4节中用到了这两个有效ID。

2.2 首先同时指定O_CREAT和O_EXCL标志，如果成功返回，那么已创建了一个新对象。然而如果调用失败并返回EEXIST错误，那么对象已经存在，程序于是得再次调用打开函数，不过不再同时指定O_CREAT和O_EXCL标志。第二次调用应该成功，但是调用失败并返回ENOENT错误的机会仍然存在（尽管很小），它表明在这两次调用之间，另外某个线程或进程已将该对象删除了。

第 3 章

3.1 我们的程序如图D-1所示。

3.2 第二个程序运行时，第一次调用msgget使用的是第一个可用的消息队列，其槽位使用序列号在运行图3-7中的程序两次之后变为20，因而返回的标识符值为1000。假设下一个可用的消息队列从未使用过，其槽位使用序列号于是为0，因而返回的标识符值为1。

3.3 我们的简单程序如图D-2所示。

---

① 此处为APUE第1版英文原版书页码，第2版为第77~78页，中文版为第61~62页。——编者注

② 此处为UNPv1第2版英文原版书节号，第3版为26.5节。——编者注

svmsg/slotseq.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     i, msqid;
6     struct msqid_ds info;

7     for (i = 0; i < 10; i++) {
8         msqid = Msgget(IPC_PRIVATE, SVMSG_MODE | IPC_CREAT);
9         Msgctl(msqid, IPC_STAT, &info);
10        printf("msqid = %d, seq = %lu\n", msqid, info.msg_perm.seq);

11        Msgctl(msqid, IPC_RMID, NULL);
12    }
13    exit(0);
14 }
```

svmsg/slotseq.c

图D-1　输出标识符和槽位使用序列号

svmsg/testumask.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     Msgget(IPC_PRIVATE, 0666 | IPC_CREAT | IPC_EXCL);
6     unlink("/tmp/fifo.1");
7     Mkfifo("/tmp/fifo.1", 0666);

8     exit(0);
9 }
```

svmsg/testumask.c

516

图D-2　测试msgget是否使用文件模式创建掩码

从下面的程序运行情况可以看出，文件模式创建掩码是2（关掉其他用户写位），该位在FIFO中确实关掉，在消息队列中却并未关掉。

```
solaris % umask
02
solaris % testumask
solaris % ls -l /tmp/fifo.1
prw-rw-r--   1 rstevens other1         0 Mar 25 16:05 /tmp/fifo.1
solaris % ipcs -q
IPC status from <running system> as of Wed Mar 25 16:06:03 1998
T         ID      KEY         MODE        OWNER    GROUP
Message Queues:
q        200   00000000 --rw-rw-rw- rstevens    other1
```

3.4　使用ftok的话，系统中另外某个路径名所形成的键与我们的服务器所用的键相同的可能性总是存在。使用IPC_PRIVATE的话，服务器尽管知道它是在创建新的消息队列，但它必须接着把所创建消息队列的标识符写到某个文件中，供客户读取。

3.5　下面是检测冲突的方法之一：

```
solaris % find / -links 1 -not -type l -print |
xargs -n1 ftok1 > temp.1
solaris % wc -l temp.1
  109351 temp.1

solaris % sort +0 -1 temp.1 |
```

```
nawk '{ if (lastkey == $1)
            print lastline, $0
        lastline = $0
        lastkey = $1
}' > temp.2
solaris % wc -l temp.2
  82188 temp.2
```

在find程序中，我们忽略链接数多于一个的文件（因为每个链接都有相同的索引节点），符号链接也忽略（因为stat函数沿循符号链接，也就是说解释并替换符号链接，直到不再有新的符号链接）。很高的冲突比率（75.2%）是由于Solaris 2.x只使用了索引节点号中的12位。这意味着在文件数多于4096的任何文件系统中，有许多冲突会发生。例如索引节点号分别为4096、8192、12288和16384的4个文件都有相同的IPC键（假设它们在同一个文件系统中）。

下一个例子运行在同样的文件系统上，但使用的是来自BSD/OS的ftok函数。由于该函数把整个索引节点号加到键中，因此冲突数只有849（少于1%）。

第4章

4.1 当父进程终止时，如果子进程中fd[1]处于打开状态，那么子进程对fd[0]的read不会返回文件结束符，因为fd[1]在子进程中仍然打开。在子进程中关闭fd[1]保证一旦父进程终止，它的所有描述符即关闭，从而使得子进程对fd[0]的read返回0。 517

4.2 如果调用关系反转了，另外某个进程就有可能在本进程的open和mkfifo两个调用之间创建本进程想要创建的FIFO，结果导致本进程的mkfifo调用失败。

4.3 如果执行如下命令：

```
solaris % mainpopen 2>temp.stderr
/etc/ntp.conf > /myfile
solaris % cat temp.stderr
sh: /myfile: cannot create
```

那么我们看到popen返回成功，但是我们用fgets读到的只是一个文件结束符。该shell出错消息是写到标准错误输出的。

4.5 把第一个open调用改为指定非阻塞标志：

```
readfifo = Open(SERV_FIFO, O_RDONLY | O_NONBLOCK, 0);
```

该调用将立即返回，接下去的open调用（用于只写）也立即返回，因为它要打开的FIFO已经由第一个open调用打开用于读。但是为了避免从readline返回错误，描述符readfifo的O_NONBLOCK标志必须在调用readline之前关掉。

4.6 如果客户在打开服务器的众所周知FIFO（用于只写）之前先打开它的客户特定FIFO（用于只读），那么会发生死锁。避免这种死锁的唯一办法是如图4-24中所示的顺序open这两个FIFO，也可以使用非阻塞标志。

4.7 写进程关闭管道或FIFO的信息通过文件结束符传递给读进程。

4.8 图D-3给出了我们的程序。

4.9 select返回说该描述符是可写的，但调用write却引发SIGPIPE信号。这个概念在UNPv1第153~155页[①]说明过，当发生读（或写）错误时，select返回说相应描述符是可读的（或可写的），真正的错误则由read（或write）返回。图D-4给出了我们的程序。

① 此处为UNPv1第2版英文原版书页码，第3版为第160~163页。——编者注

pipe/test1.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     fd[2];
6     char    buff[7];
7     struct stat info;
8     if (argc != 2)
9         err_quit("usage: test1 <pathname>");
10    Mkfifo(argv[1], FILE_MODE);
11    fd[0] = Open(argv[1], O_RDONLY | O_NONBLOCK);
12    fd[1] = Open(argv[1], O_WRONLY | O_NONBLOCK);
13        /* check sizes when FIFO is empty */
14    Fstat(fd[0], &info);
15    printf("fd[0]: st_size = %ld\n", (long) info.st_size);
16    Fstat(fd[1], &info);
17    printf("fd[1]: st_size = %ld\n", (long) info.st_size);
18    Write(fd[1], buff, sizeof(buff));
19        /* check sizes when FIFO contains 7 bytes */
20    Fstat(fd[0], &info);
21    printf("fd[0]: st_size = %ld\n", (long) info.st_size);
22    Fstat(fd[1], &info);
23    printf("fd[1]: st_size = %ld\n", (long) info.st_size);
24    exit(0);
25 }
```

pipe/test1.c

图D-3　判定fstat是否返回在某个FIFO中的字节数

pipe/test2.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     fd[2], n;
6     pid_t   childpid;
7     fd_set  wset;
8     Pipe(fd);
9     if ( (childpid = Fork()) == 0) {        /* child */
10        printf("child closing pipe read descriptor\n");
11        Close(fd[0]);
12        sleep(6);
13        exit(0);
14    }
15        /* parent */
16    Close(fd[0]);                   /* in case of a full-duplex pipe */
17    sleep(3);
18    FD_ZERO(&wset);
19    FD_SET(fd[1], &wset);
20    n = select(fd[1] + 1, NULL, &wset, NULL, NULL);
21    printf("select returned %d\n", n);
22    if (FD_ISSET(fd[1], &wset)) {
23        printf("fd[1] writable\n");
24        Write(fd[1], "hello", 5);
25    }
26    exit(0);
27 }
```

pipe/test2.c

图D-4　当一个管道的读出端关闭时，判定select为可写性返回的是什么

518 ~ 519

第 5 章

5.1 先不指定任何属性创建该队列，紧接着调用mq_getattr取得默认属性。随后删除该队列并重新创建，对未指定的那个属性使用其默认值。

5.2 对应第二个消息的信号没有产生是因为注册在每次通知发生之后即撤销。

5.3 对应第二个消息的信号没有产生是因为接收该消息时队列不空。

5.4 Solaris 2.6把这两个常值定义成调用sysconf，其上的GNU C编译器将产生如下出错消息：

```
test1.c:13: warning: int format, long int arg (arg 2)
test1.c:13: warning: int format, long int arg (arg 3)
```

5.5 在Solaris 2.6下，我们指定1 000 000个消息，每个消息10字节。这使文件大小为20 000 536字节，它与我们运行图5-5中程序所得的结果是一致的：每个消息占据10字节数据、8字节开销（也许是为存放指针）以及另外2字节开销（也许因4字节对齐之需），每个文件再占据536字节开销。在调用mq_open之前，由ps所报告的该程序大小为1052KB，该消息队列创建之后，大小变为20MB。这使得我们认为Posix消息队列是使用内存映射文件实现的，mq_open把该文件映射到调用进程的地址空间中。在Digital Unix 4.0B下我们也取得了类似的结果。

5.6 对于ANSI C mem*XXX*函数来说，大小参数为0不成问题。最初的1989 ANSI C标准X3.159-1989（也称为ISO/IEC 9899:1990）并没有这么说，作者能找到的手册页面也没有一个提及这一点，然而“Technical Corrigendum Number 1（1号技术勘误）”却明确陈述大小为0可行（不过指针参数仍必须有效）。要参阅有关C语言的信息，http://www.lysator.liu.se/c/是个颇值访问的地方。

5.7 两进程之间的双向通信需2个消息队列（图A-30是这样的一个例子）。事实上，要是我们把图4-14中程序改为使用Posix消息队列而不是管道，就会看到父进程读回它写到队列中的东西。

5.8 互斥锁和条件变量包含在内存映射文件中，而该文件是由打开了相应队列的所有进程共享的。其他进程也许打开着该队列，因此即将关闭该队列本地句柄的一个进程不能摧毁该互斥锁和条件变量。

5.9 C语言中数组不能通过等号赋值，结构却可以。

5.10 main函数几乎把所有时间都花在select调用的阻塞之中，等待管道变为可读。每次提交相应信号时，其信号处理程序的返回会中断这个select调用，使得它返回一个EINTR错误。为处理这种情形，我们的Select包裹函数检查这个错误，并重新调用select，如图D-5所示。 520

lib/wrapunix.c

```
313 int
314 Select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
315         struct timeval *timeout)
316 {
317     int     n;

318 again:
319     if ( (n = select(nfds, readfds, writefds, exceptfds, timeout)) < 0){
320         if (errno == EINTR)
```

图D-5 处理EINTR的Select包裹函数

```
321             goto again;
322         else
323             err_sys("select error");
324     } else if (n == 0 && timeout == NULL)
325         err_quit("select returned 0 with no timeout");
326     return(n);                  /* can return 0 on timeout */
327 }
```

lib/wrapunix.c

图D-5（续）

UNPv1第124页[①]有关于被中断系统调用的详细讨论。

第6章

6.1　其余程序必须接受数值形式的消息队列标识符，而不是路径名（回想一下图6-3中程序的输出）。这些程序上的如此变动既可通过增设--个新的命令行选项做到，也可假设完全为数值的路径名参数是标识符而不是真的路径名。既然传递给ftok的多数路径名是绝对路径名而不是相对路径名（也就是说它们至少包含一个斜杠符），这样的假设也许可行。

6.2　类型为0的消息是不被允许的，而客户是决不可能有1这个进程ID的，因为它通常是init进程的进程ID。

6.3　当如图6-14所示只使用一个队列时，这个恶意的客户影响所有其他客户。当给每个客户准备一个返送队列时（图6-19），这个客户只能影响它自己的队列。

第7章

7.2　进程将终止，而且可能是在消费者线程完成之前，因为调用exit将终止任何仍在运行中的线程。

7.3　Solaris 2.6下，省略destroy函数的调用导致内存泄漏，暗示init函数是在执行动态内存分配。在Digital Unix 4.0B下，我们没有看到这种现象，这意味着实现上存在差异。不过调用匹配的destroy函数仍是需要的。从实现的角度看，Digital Unix像是把attr_t变量用作属性对象本身，Solaris则把该变量用作指向动态分配对象的指针。这两种实现都是可行的。

521

第9章

9.1　你可能需要把原来为20的循环计数加大才能看到这些错误，这取决于你的系统。

9.2　要使标准I/O流不缓冲，我们在main函数的for循环之前插入如下行：

```
setvbuf(stdout, NULL, _IONBF, 0);
```

这么修改不应该有任何效果，因为printf调用只有一个，而且所输出字符串是以换行符结尾的。通常情况下，标准输出是行缓冲的，因此不论哪种缓冲方式（行缓冲或不缓冲），这个单独的printf调用最终变为对内核的单个write调用。

9.3　我们把printf调用改为：

```
snprintf(line, sizeof(line), "%s: pid = %ld, seq# = %d\n",
         argv[0], (long)pid, seqno);
for (ptr = line; (c = *ptr++) != 0; )
     putchar(c);
```

① 此处为UNPv1第2版英文原版书页码，第3版为第134~135页。——编者注

并声明c是一个整数，ptr的类型为char *。保留上一道习题所加的setvbuf调用不变，从而使得标准输出变为不缓冲，于是标准I/O函数库给所输出的每个字符调用一次write，而不是每行调用一次。这么一来需要更多的CPU时间，内核在两个进程之间来回切换的机会也增多。我们应该从这个程序的运行中看到更多的错误。

9.4 既然对于一个文件的同一区段允许多个进程有读出锁，那么就我们的例子而言，这与没有任何锁是一样的。

9.5 没有任何变化，因为一个描述符的非阻塞标志对于fcntl劝告性上锁没有影响。决定fcntl调用是否阻塞的是其命令：F_SETLKW表明总是阻塞，F_SETLK则表明永不阻塞。

9.6 loopfcntlnonb程序运行如常，因为我们已在上一道习题展示，非阻塞标志对于执行fcntl上锁的程序没有影响。然而非阻塞标志确实影响不执行上锁的loopnonenonb程序。我们在9.5节说过，如果对启用了强制性上锁的文件所进行的read或write非阻塞调用与已有的锁发生冲突，那么会返回一个EAGAIN错误。我们看到的这个错误或者是

```
read error: Resource temporarily unavailable
```

或者是

```
write error: Resource temporarily unavailable
```

通过执行如下命令就能验证这个错误是EAGAIN：

```
solaris % grep Resource /usr/include/sys/errno.h
#define EAGAIN  11      /* Resource temporarily unavailable */
```

522

9.7 Solaris 2.6下，强制性上锁增加了约16%的时钟时间和约20%的系统CPU时间。用户CPU时间保持不变，正如我们所预期的那样，这是因为额外的时间花在了内核对每个read和write调用的检查上，而不是在用户进程上。

9.8 锁是以每个进程为基而不是以每个线程为基授予的。要看到上锁请求的竞争现象，我们必须让不同的进程来尝试获取锁。

9.9 如果本守护进程的另一个副本正在运行，当使用O_TRUNC标志open时，由本守护进程的第一个副本存放的进程ID就会被冲掉。我们只有获悉自己是唯一在运行的副本后，才能截掉文件内容。

9.10 SEEK_SET总是最可取的。SEEK_CUR的问题是它取决于文件中的当前偏移量，而该值是由lseek指定的。但是如果在调用lseek之后调用fcntl，那么我们是在使用两个函数调用完成单个操作的任务，而这两个函数调用之间存在由另外一个线程通过调用lseek修改当前偏移量的机会。（回想一下所有线程共享相同的描述符。另外回想一下fcntl记录锁用于不同进程之间的上锁，而不是单个进程内的不同线程之间的上锁。）同样，如果我们指定SEEK_END，那么在基于所认定的文件尾获得一个锁之前，另外一个线程有可能已往该文件添加数据。

第 10 章

10.1 以下是在Solaris 2.6下的结果输出：

```
solaris % deadlock 100
prod: calling sem_wait(nempty)              生产者i=0时的循环
prod: got sem_wait(nempty)
prod: calling sem_wait(mutex)
prod: got sem_wait(mutex), storing 0

prod: calling sem_wait(nempty)              生产者i=1时的循环
```

```
prod: got sem_wait(nempty)
prod: calling sem_wait(mutex)
prod: got sem_wait(mutex), storing 1

prod: calling sem_wait(nempty)              开始下一轮循环，但没有空槽位
                                            上下文从生产者切换到消费者
cons: calling sem_wait(mutex)               消费者i=0时的循环
cons: got sem_wait(mutex)
cons: calling sem_wait(nstored)
cons: got sem_wait(nstored)
cons: fetched 0

cons: calling sem_wait(mutex)               消费者i=1时的循环
cons: got sem_wait(mutex)
cons: calling sem_wait(nstored)
cons: got sem_wait(nstored)
cons: fetched 1

cons: calling sem_wait(mutex)
cons: got sem_wait(mutex)
cons: calling sem_wait(nstored)             消费者永远阻塞于此
                                            上下文从消费者切换到生产者
prod: got sem_wait(nempty)
prod: calling sem_wait(mutex)               生产者永远阻塞于此
```

523

10.2　在我们描述sem_open时指定了信号量初始化规则的前提下，这是可行的。该规则说，如果信号量已经存在，它就不被初始化。因此这4个进程中，只有第一个调用sem_open的进程才真正把信号量的值初始化为1。当其余3个进程以O_CREAT标志调用sem_open时，所需信号量已经存在，其值于是不再被初始化。

10.3　这确实是个问题。信号量在该进程终止时被自动关闭，但是其值并没有改变。这将妨碍其他3个进程中的任何一个取得所需的锁，从而导致另外一种类型的死锁。

10.4　要是我们没有把这两个描述符初始化为-1，那么它们的初始值是不确定的，因为malloc并不对它分配的内存进行初始化。这么一来，如果有一个open调用失败，error标号处的close调用就可能关闭该进程正在使用的某个描述符。把描述符初始化为-1后，我们可知如果描述符尚未被打开则close调用不会有什么后果（除返回一个我们忽略掉的错误外）。

10.5　尽管很小，却存在这样的机会：close调用虽然针对一个有效的描述符，但仍可能返回某个错误，从而把errno由我们想返回的值改为其他值。既然我们想要保存errno值以返回给调用者，显式地去做总比依赖某些副作用（例如当关闭的是一个有效的描述符时，close调用不会返回错误）来得好。

10.6　这个函数中不存在竞争状态，因为如果所需的FIFO已经存在，mkfifo函数将返回一个错误。如果有两个进程几乎同时调用这个函数，那么相应的FIFO只创建一次。调用mkfifo的第二个进程将收到一个EEXIST错误，导致O_CREAT标志被关掉，从而防止再次初始化同一个FIFO。

10.7　图10-37没有我们随图10-43描述的竞争状态，因为其信号量的初始化是通过写数据到相应的FIFO完成的。如果创建该FIFO的进程在调用mkfifo之后但在向该FIFO write数据字节之前被内核挂起，那么第二个进程只是打开该FIFO，随后在首次调用sem_wait处阻塞，因为这个新创建的FIFO在第一个进程（即创建该FIFO的进程）往它写数据字节前一直为空。

10.8　图D-6给出了该测试程序。Solaris 2.6和Digital Unix 4.0B上的实现都检测被某个捕获的

信号中断的情况，并返回EINTR错误。 524

pxsem/testeintr.c

```
1 #include    "unpipc.h"
2 #define NAME    "testeintr"
3 static void sig_alrm(int);
4 int
5 main(int argc, char **argv)
6 {
7     sem_t   *sem1, sem2;
8         /* first test a named semaphore */
9     sem_unlink(Px_ipc_name(NAME));
10    sem1 = Sem_open(Px_ipc_name(NAME), O_RDWR | O_CREAT | O_EXCL,
11                    FILE_MODE, 0);
12    Signal(SIGALRM, sig_alrm);
13    alarm(2);
14    if (sem_wait(sem1) == 0)
15        printf("sem_wait returned 0?\n");
16    else
17        err_ret("sem_wait error");
18    Sem_close(sem1);
19        /* now a memory-based semaphore with process scope */
20    Sem_init(&sem2, 1, 0);
21    alarm(2);
22    if (sem_wait(&sem2) == 0)
23        printf("sem_wait returned 0?\n");
24    else
25        err_ret("sem_wait error");
26    Sem_destroy(&sem2);
27    exit(0);
28 }
29 static void
30 sig_alrm(int signo)
31 {
32    printf("SIGALRM caught\n");
33    return;
34 }
```

pxsem/testeintr.c

图D-6 测试sem_wait是否返回EINTR

我们使用FIFO的实现会返回EINTR，因为sem_wait阻塞在对于一个FIFO的某个read调用中，而read调用必须返回EINTR错误。我们使用内存映射I/O的实现不返回任何错误，因为sem_wait阻塞在某个pthread_cond_wait调用中，而该函数被一个捕获的信号中断时并不返回EINTR。（我们在图5-29中看到过另外一个例子。）我们使用System V信号量的实现返回EINTR，因为sem_wait阻塞在某个semop调用中，而semop调用返回EINTR错误。

10.9 使用FIFO的实现（图10-40）是异步信号安全的，因为write是异步信号安全的。使用内存映射文件的实现（图10-47）则不是，因为没有一个pthread_*XXX*函数是异步信号安全的。使用System V信号量的实现（图10-56）也不是，因为Unix 98没有把semop列为异步信号安全函数。 525

第11章

11.1　只需修改其中一行：

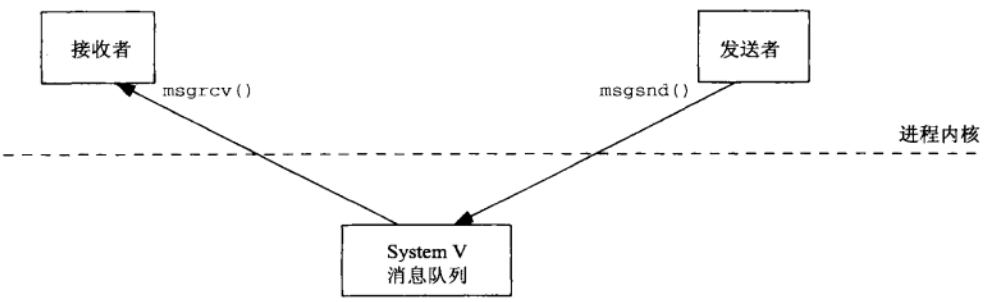
```
<      semid = Semget(Ftok(argv[optind], 0), 0, 0);
---
>      semid = atol(argv[optind]);
```

11.2　ftok调用将失败，从而导致我们的Ftok包裹函数的终止。my_lock函数可在调用semget之前调用ftok，检查是否返回ENOENT错误，若LOCK_PATH文件不存在则创建它。

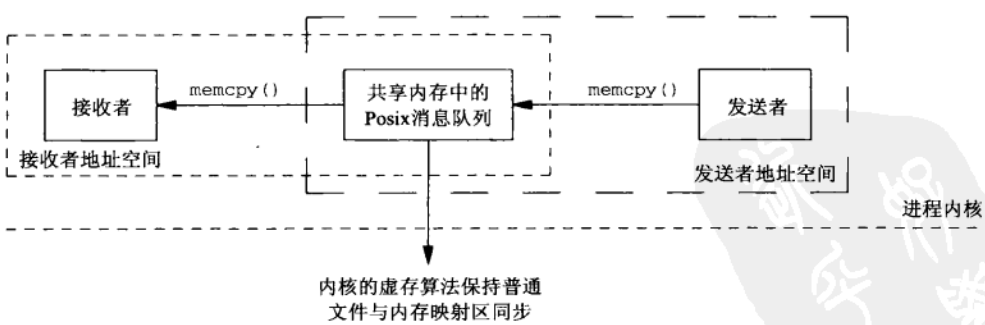
第12章

12.1　文件大小将再增长4 096字节（达到36 864字节），最后一个printf对新文件结束符（ptr字符数组的对应下标为36863）所作的引用可能引发SIGSEGV信号，因为内存映射区的大小为32 768字节。我们说“可能”而不是“将”的原因在于，该信号产生与否取决于页面大小。

12.2　图D-7是使用System V消息队列发送消息的示意图，图D-8是使用通过mmap实现的Posix消息队列发送消息的示意图。图D-8中发送者的memcpy发生在调用mq_send期间（图5-30），接收者的memcpy则发生在调用mq_receive期间（图5-32）。



图D-7　使用System V消息队列发送消息



图D-8　使用通过mmap实现的Posix消息队列发送消息

12.3　对/dev/zero设备文件的任何read，所返回的是所请求数目的全为0的字节。write到该设备的任何数据被直接丢弃掉，就像write到/dev/null设备一样。

12.4 该文件的最终内容为4字节的0（假设int类型为32位）。

12.5 图D-9给出了我们的程序。

shm/svmsgread.c

```
 1 #include    "unpipc.h"
 2 #define MAXMSG  (8192 + sizeof(long))
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     pipe1[2], pipe2[2], mqid;
 7     char    c;
 8     pid_t   childpid;
 9     fd_set  rset;
10     ssize_t n, nread;
11     struct msgbuf   *buff;
12     if (argc != 2)
13         err_quit("usage: svmsgread <pathname>");
14     Pipe(pipe1);              /* 2-way communication with child */
15     Pipe(pipe2);
16     buff = My_shm(MAXMSG);    /* anonymous shared memory with child */
17     if ( (childpid = Fork()) == 0) {
18         Close(pipe1[1]);      /* child */
19         Close(pipe2[0]);
20         mqid = Msgget(Ftok(argv[1], 0), MSG_R);
21         for ( ; ; ) {
22                 /* block, waiting for message, then tell parent */
23             nread = Msgrcv(mqid, buff, MAXMSG, 0, 0);
24             Write(pipe2[1], &nread, sizeof(ssize_t));
25                 /* wait for parent to say shm is available */
26             if ( (n = Read(pipe1[0], &c, 1)) != 1)
27                 err_quit("child: read on pipe returned %d", n);
28         }
29         exit(0);
30     }
31         /* parent */
32     Close(pipe1[0]);
33     Close(pipe2[1]);
34     FD_ZERO(&rset);
35     FD_SET(pipe2[0], &rset);
36     for ( ; ; ) {
37         if ( (n = select(pipe2[0] + 1, &rset, NULL, NULL, NULL)) != 1)
38             err_sys("select returned %d", n);
39         if (FD_ISSET(pipe2[0], &rset)) {
40             n = Read(pipe2[0], &nread, sizeof(ssize_t));
41             if (n != sizeof(ssize_t))
42                 err_quit("parent: read on pipe returned %d", n);
43             printf("read %d bytes, type = %ld\n", nread, buff->mtype);
44             Write(pipe1[1], &c, 1);
45         } else
46             err_quit("pipe2[0] not ready");
47     }
48     Kill(childpid, SIGTERM);
49     exit(0);
50 }
```

shm/svmsgread.c

图D-9 父子进程设置成对System V消息使用select的例子

526 ~ 527

第13章

13.1　图D-10给出了图12-16的修改后版本，图D-11给出了图12-19的修改后版本。注意在第一个程序中，我们必须使用ftruncate设置共享内存对象的大小，而不能使用lseek和write。

pxshm/test1.c

```
1 #include    "unpipc.h"
2 int
3 main(int argc, char **argv)
4 {
5     int     fd, i;
6     char    *ptr;
7     size_t  shmsize, mmapsize, pagesize;
8     if (argc != 4)
9         err_quit("usage: test1 <name> <shmsize> <mmapsize>");
10    shmsize = atoi(argv[2]);
11    mmapsize = atoi(argv[3]);
12        /* open shm: create or truncate; set shm size */
13    fd = Shm_open(Px_ipc_name(argv[1]), O_RDWR | O_CREAT | O_TRUNC,
14                  FILE_MODE);
15    Ftruncate(fd, shmsize);
16    ptr=Mmap(NULL, mmapsize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
17    Close(fd);
18    pagesize = Sysconf(_SC_PAGESIZE);
19    printf("PAGESIZE = %ld\n", (long) pagesize);
20    for (i = 0; i < max(shmsize, mmapsize); i += pagesize) {
21        printf("ptr[%d] = %d\n", i, ptr[i]);
22        ptr[i] = 1;
23        printf("ptr[%d] = %d\n", i + pagesize - 1,ptr[i + pagesize - 1]);
24        ptr[i + pagesize - 1] = 1;
25    }
26    printf("ptr[%d] = %d\n", i, ptr[i]);
27    exit(0);
28 }
```

pxshm/test1.c

528

图D-10　访问其大小可能不同于共享内存区大小的mmap

pxshm/test2.c

```
1 #include    "unpipc.h"
2 #define FILE     "test.data"
3 #define SIZE     32768
4 int
5 main(int argc, char **argv)
6 {
7     int     fd, i;
8     char    *ptr;
9         /* open shm: create or truncate; then mmap shm */
10    fd=Shm_open(Px_ipc_name(FILE),O_RDWR|O_CREAT|O_TRUNC, FILE_MODE);
11    ptr = Mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

图D-11　允许共享内存区大小增长的内存映射例子

```
12      for (i = 4096; i <= SIZE; i += 4096) {
13          printf("setting shm size to %d\n", i);
14          Ftruncate(fd, i);
15          printf("ptr[%d] = %d\n", i-1, ptr[i-1]);
16      }
17      exit(0);
18  }
```

pxshm/test2.c

图D-11（续）

13.2 *ptr++可能存在的问题之一是由mmap返回的指针被改掉，从而妨碍以后调用munmap。如果以后要用到该指针，我们就得把它保存起来，或者不作修改。 529

第 14 章

14.1 只有一行需要改动：

```
13c13
<      id = Shmget(Ftok(argv[1], 0), 0, SVSHM_MODE);
---
>      id = atol(argv[1]);
```

第 15 章

15.1 这样的参数的字节数为data_size+(desc_num*sizeof(door_desc_t))。

15.2 没有调用fstat的必要。如果所打开的描述符指代的不是一个门，那么door_info将返回一个EBADF错误：

```
solaris % doorinfo /etc/passwd
door_info error: Bad file number
```

15.3 该手册页面是错误的。Posix.1对此的正确陈述为“The *sleep()* function shall cause the current thread to be suspended from execution.”（sleep函数会导致当前线程从执行状态挂起）。

15.4 结果难以预料（尽管核心转储是一个相当安全的猜测），因为与其中的门相关联的服务器过程的地址会导致在新执行的程序中，某段随机的代码会被作为一个函数来调用。

15.5 当客户的door_call被所捕获的信号中断时，服务器进程必须被通知到，因为它需随后向处理该客户的服务器线程（它的ID在我们的输出例子中为4）发送一个取消请求，然而我们随图15-23说过，对于由门函数库自动创建的所有服务器线程来说，取消操作是被禁止的，我们正讨论的线程于是不会因取消而终止。相反，当客户的door_call被终止时，服务器过程阻塞在其上的sleep(6)调用看来在该过程被调用后约2秒钟就过早地返回了。但是执行该过程的服务器线程仍持续运行到完成为止。

15.6 我们看到的错误如下：

```
solaris % server6 /tmp/door6
my_thread: created server thread 4
door_bind error: Bad file number
```

当我们连续启动该服务器20次时，该错误出现了5次。该错误是不确定的。

15.7 不需要。我们需做的只是如图15-31中那样，每次服务器过程被调用时就启用取消。这种技术尽管在服务器过程每次被激活时都要调用pthread_setcancelstate，而不

是仅在执行该过程的线程启动时调用一次，但其开销却可能很小。

530 15.8 为验证这一点，我们将某个服务器程序（譬如说图15-9）改为从服务器过程中调用door_revoke。由于门描述符是door_revoke的参数，因此我们还得把fd改成一个全局变量。我们随后执行相应的客户程序（譬如说图15-2）两次：

```
solaris % client8 /tmp/door8 88
result: 7744
solaris % client8 /tmp/door8 99
door_call error: Bad file number
```

第一次激活服务器过程成功返回，从而证实door_revoke不影响已在进行的调用。第二次激活告知从door_call返回的错误是EBADF。

15.9 为避免使fd成为一个全局变量，我们使用传递给door_create的cookie指针，该指针随后在服务器过程每次被调用时传递给它。图D-12给出了服务器程序。

doors/server9.c

```
 1 #include    "unpipc.h"
 2 void
 3 servproc(void *cookie, char *dataptr, size_t datasize,
 4           door_desc_t *descptr, size_t ndesc)
 5 {
 6     long    arg, result;
 7     Door_revoke(*((int *) cookie));
 8     arg = *((long *) dataptr);
 9     printf("thread id %ld, arg = %ld\n", pr_thread_id(NULL), arg);
10     result = arg * arg;
11     Door_return((char *) &result, sizeof(result), NULL, 0);
12 }
13 int
14 main(int argc, char **argv)
15 {
16     int     fd;
17     if (argc != 2)
18         err_quit("usage: server9 <server-pathname>");
19         /* create a door descriptor and attach to pathname */
20     fd = Door_create(servproc, &fd, 0);
21     unlink(argv[1]);
22     Close(Open(argv[1], O_CREAT | O_RDWR, FILE_MODE));
23     Fattach(fd, argv[1]);
24         /* servproc() handles all client requests */
25     for ( ; ; )
26         pause();
27 }
```

doors/server9.c

图D-12　使用cookie指针以避免使fd成为一个全局变量

531 我们可以很容易地对图15-22和图15-23作同样的修改，因为cookie指针对我们的my_create函数而言是可得的（该指针在door_info_t结构中），而该函数又把指向该结构的指针传递给新创建的线程（它需要对应door_bind调用的描述符）。

15.10 本例中线程属性从不改变，因此我们可以只初始化一次线程属性（在main函数中完成）。

第16章

16.1 端口映射器并不监视已向它注册的各个服务器，因而无法检测它们是否崩溃。终止其

中某个服务器后，它在端口映射器中注册的映射关系并不注销，这一点可使用rpcinfo程序来验证。这么一来，该服务器终止后，与端口映射器联系以获取该服务器端口号的某个客户将得到肯定的答复，由端口映射器返回该服务器在终止之前使用的端口号。假设该服务器是一个TCP服务器，当该客户试图与它联系时，客户方RPC运行时环境将收到RST（复位）分节作为对SYN分节的响应（前提是自该服务器终止以来，服务器主机上没有其他进程被赋予同样的端口），从而导致从clnt_create返回一个错误。UDP客户调用clnt_create将成功（因为没有连接需要建立）。但是当它向以前的服务器端口发送UDP数据报时，什么应答都不会返回（同样假设自该服务器终止以来，服务器主机上没有其他进程被赋予同样的端口），该客户的远程过程调用最终将超时。

16.2 当收到服务器的第一个应答后，RPC运行时环境把它返回给客户，这发生在客户发出远程过程调用后约20秒。因超时重传导致的服务器的下一个应答将一直存留在客户方端点的网络缓冲区中，直到该端点被关闭，或者直到RPC运行时环境下一次读该缓冲区为止。现在假设客户在收到第一个应答后立即向服务器再次发出调用[①]，再假设没有网络分组丢失现象，下一个到达客户方端点的数据报将是服务器对于客户的超时重传的应答。然而RPC运行时环境将忽略这个应答，因为它的XID对应于客户的第一次远程过程调用，它不可能与客户的第二次过程调用所用的XID相同。

16.3 构造一个成员为char c[10]的C结构，不过XDR将把它编码成10个4字节整数。要是你确实需要定长的字符串，那就使用定长的不透明数据类型。

16.4 xdr_data调用返回FALSE，因为它的xdr_string调用返回FALSE（参见data_xdr.c文件）。

当指定一个最大长度时，它将作为xdr_string的最后一个参数编码。当省略这个最大长度时，最后一个参数是0的反码（在32位整数前提下，其值为$2^{32}-1$）。

16.5 所有的XDR例程都检查缓冲区中是否有足够的空间以存放将编码到其中的数据，当缓冲区满时返回FALSE错误。不幸的是，没有办法区别来自XDR函数的各种不同的可能错误。

16.6 我们可以说TCP使用序列号检测重复数据在效果上等同于重复请求高速缓存，因为对于作为含有TCP已确认过的重复数据而到达的任何旧分节，这些序列号都能将其标识出来。对于一个给定的连接（例如一个给定客户的IP地址和端口），该高速缓存的大小是TCP的32位序列号空间的一半，也就是2^{31}或约2G字节。 532

16.7 由于对一个给定的请求来说，它的所有5个值必须等于某个高速缓存项中对应的5个值，因此第一个作比较的值应该是最可能不等的那个值，而最后一个作比较的值应该是最可能相等的那个值。在TI-RPC软件包中，真正的比较顺序是：（1）XID，（2）过程号，（3）版本号，（4）程序号，（5）客户地址。在XID随每个请求变化的前提下，首先对它进行比较是明智的。

16.8 在图16-30中，从标志/长度字段开始，包括4字节的长整数过程参数在内，共有12个4字节字段，合计48字节。按照默认的无认证配置，凭证数据和验证器数据均为空。也就是说，凭证和验证器均占用8字节：4字节指定认证方式（AUTH_NONE），另4字节指定认证长度（其值为0）。

在应答分节中（看图16-32，但要意识到由于在使用TCP，因此在XID之前有一个4字

① 注意区别这个由客户主动发出的再次调用和因超时重传导致的由RPC运行时环境完成的再次调用，后者对客户不可见。——译者注

节的标志/长度字段），共有8个4字节字段，从标志/长度字段开始，到4字节的长整数过程结果为止，共计32字节。

当使用UDP时，请求和应答的唯一变动是不存在4字节的标志/长度字段。因此请求的大小为44字节，应答的大小为28字节，这一点可使用tcpdump验证。

16.9　可以。不论在客户端还是在服务器端，参数处理上的差异都局限于主机，而与穿越网络的分组无关。客户的main函数调用客户存根中的某个函数以产生一个网络记录，服务器的main函数则调用服务器存根中的某个函数处理这个网络记录。跨越网络传送的RPC记录是由RPC协议定义的，而RPC协议不论客户端或服务器端是否支持线程都不变。

16.10　XDR运行时环境给这些字符串动态分配字间。给read程序增加下面一行就能验证这个事实：

```
printf("sbrk() = %p, buff = %p, in.vstring_arg = %p\n",
       sbrk(NULL), buff, in.vstring_arg);
```

其中sbrk函数返回处于程序数据段顶部的当前地址，而在此以下的内存空间通常就是malloc从中分配内存的区段。运行该程序产生如下输出：

```
sbrk() = 29638, buff = 25e48, in.vstring_arg = 27e58
```

它表明指针vstring_arg指向malloc使用的内存区段内。8192字节的buff地址为
533 0x25e48~0x27e47，字符串就存放在该缓冲区之后。

16.11　图D-13给出了客户程序。注意clnt_call的最后一个参数是一个真正的timeval结构，而不是指向某个这种结构的指针。还要注意clnt_call的第三个和第五个参数必须是指向XDR例程的非空函数指针，因此我们指定的是不做任何工作的XDR函数xdr_void。（编写一个很小的RPC规范文件，其中定义一个既没有参数也没有返回值的函数，运行rpcgen，然后检查所生成的客户存根，这样你就能验证图D-13中给出的确实是调用一个既没有参数又没有返回值的函数的方法。）

sunrpc/square10/client.c

```
1 #include     "unpipc.h"          /* our header */
2 #include     "square.h"          /* generated by rpcgen */
3 int
4 main(int argc, char **argv)
5 {
6       CLIENT      *cl;
7       struct timeval tv;
8       if (argc != 3)
9           err_quit("usage: client <hostname> <protocol>");
10      cl = Clnt_create(argv[1], SQUARE_PROG, SQUARE_VERS, argv[2]);
11      tv.tv_sec = 10;
12      tv.tv_usec = 0;
13      if (clnt_call(cl, NULLPROC, xdr_void, NULL,
14                    xdr_void, NULL, tv) != RPC_SUCCESS)
15          err_quit("%s", clnt_sperror(cl, argv[1]));
16      exit(0);
17 }
```

sunrpc/square10/client.c

图D-13　调用服务器空过程的客户程序

16.12　所产生的UDP数据报大小（65536+20+RPC开销）超过了IPv4数据报的最大大小65535。图A-4中对于使用UDP的RPC来说，不存在消息大小为16384和32768的项的原因是，这是一个较早的RPCSRC 4.0实现，它把UDP数据报的大小限制在约9000字
534 节左右。

参 考 文 献

要是能找到本参考文献所引用的论文或报告的电子文档的话，我们就给出它们的URL。需留意的是，这些URL可能随时间而变动，因此读者应经常访问作者在http://www.kohala.com/~rstevens的WWW主页，检查本书的最新勘误表。

Bach, M.J. 1986. *The Design of the UNIX Operating System*. Prentice Hall, Englewood Cliffs, N.J.

Birrell, A.D., and Nelson, B.J. 1984. "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol.2, no.1, pp.39-59 (Feb.).

Butenhof, D.R. 1997. *Programming with POSIX Threads*. Addison-Wesley, Reading, Mass.

Corbin, J.R. 1991. *The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls*. Springer-Verlag, New York.

Garfinkel, S.L., and Spafford, E.H. 1996. *Practical UNIX and Internet Security, Second Edition*. O'Reilly & Associates, Sebastopol, Calif.

Goodheart, B., and Cox, J. 1994. *The Magic Garden Explained: The Internals of UNIX System V Release 4, An Open Systems Design*. Prentice Hall, Englewood Cliffs, N.J.

Hamilton, G., and Kougiouris, P. 1993. "The Spring Nucleus: A Microkernel for Objects," *Proceedings of the 1993 Summer USENIX Conference*, pp.147-159, Cincinnati, Oh.

http://www.kohala.com/~rstevens/papers.others/springnucleus.1993.ps

IEEE. 1996. "Information Technology—Portable Operating System Interface (POSIX) —Part 1: System Application Program Interface(API) [C Language]," IEEE Std 1003.1, 1996 Edition, Institute of Electrical and Electronics Engineers, Piscataway, N.J. (July).

这个版本的Posix.1含有1990年版基本API、1003.1b实时扩展（1993年）、1003.1c Pthreads（1995年）以及1003-1i技术性更正（1995年）。它同时也是国际标准ISO/IEC 9945-1: 1996 (E)。IEEE正式标准和草案标准的定购信息可从http://www.ieee.org获取。遗憾的是，因特网上IEEE标准不是免费可得的。

Josey, A., ed. 1997. *Go Solo 2: The Authorized Guide to Version 2 of the Single UNIX Specification*. Prentice Hall, Upper Saddle River, N.J.

另请注意，许多Unix 98的规范（例如所有的手册页面）可从http://www.UNIX-systems.org/online.html在线获取。

Kernighan, B.W., and Pike, R. 1984. *The UNIX Programming Environment*. Prentice Hall, Englewood Cliffs, N.J.

Kernighan, B.W., and Ritchie, D.M. 1988. *The C Programming Language, Second Edition*. Prentice

Hall, Englewood Cliffs, N.J.

Kleiman, S., Shah, D., and Smaalders, B. 1996. *Programming with Threads*. Prentice Hall, Upper Saddle River, N.J.

Lewis, B., and Berg, D.J. 1998. *Multithreaded Programming with Pthreads*. Prentice Hall, Upper Saddle River, N.J.

McKusick, M.K., Bostic, K., Karels, M.J., and Quarterman, J.S. 1996. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, Reading, Mass.

McVoy, L., and Staelin, C. 1996. "Imbench: Portable Tools for Performance Analysis," *Proceedings of the 1996 Winter Technical Conference*, pp.279-294, San Diego, Calif.

这套标准性能测试工具以及本论文可从http://www.bitmover.com/lmbench获取。

Rochkind, M.J. 1985. *Advanced UNIX Programming*. Prentice Hall, Englewood Cliffs, N.J.

Salus, P.H. 1994. *A Quarter Century of Unix*. Addison-Wesley, Reading, Mass.

Srinivasan, R. 1995a. "RPC: Remote Procedure Call Protocol Specification Version 2," RFC 1831, 18 pages (Aug.).

Srinivasan, R. 1995b. "XDR: External Data Representation Standard," RFC 1832, 24 pages (Aug.).

Srinivasan, R. 1995c. "Binding Protocols for ONC RPC Version 2," RFC 1833, 14 pages (Aug.).

Stevens, W.R. 1992. *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, Mass.

全部Unix编程细节。本书称之为APUE。

Stevens, W.R. 1994. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, Mass.

对于网际协议的完整介绍。本书称之为TCPv1。

Stevens, W.R. 1996. *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*. Addison-Wesley, Reading, Mass.

本书称之为TCPv3。

Stevens, W.R. 1998. *UNIX Network Programming, Volume 1, Second Edition, Networking APIs: Sockets and XTI*. Prentice Hall, Upper Saddle River, N.J.

本书称之为UNPv1。

Vahalia, U. 1996. *UNIX Internals: The New Frontiers*. Prentice Hall, Upper Saddle River, N.J.

White, J.E. 1975. "A High-Level Framework for Network-Based Resource Sharing," RFC 707, 27 pages (Dec.).

http://www.kohala.com/~rstevens/papers.others/rfc707.txt

Wright, G.R., and Stevens, W.R. 1995. *TCP/IP Illustrated, Volume 2: The Implementation*. Addison-Wesley, Reading, Mass.

网际协议在4.4BSD-Lite操作系统上的实现。本书称之为TCPv2。

索　引

我们不提供一个单独的词汇表（其中大多数条目将是首字母缩写词），不过本索引也可用作本书所用所有首字母缩写词的词汇表。可以首字母缩写的词条其主条目编排在缩写词之下。举例来说，所有对Remote Procedure Call（远程过程调用）的引用出现在RPC之下；在完整词条“Remote Procedure Call”之下的条目只是引用回RPC之下的主条目。

每个C函数的“definition of（定义）”条目给出该函数带方框的函数原型即基本描述的所在页。每个结构的“definition of（定义）”条目给出该结构的基本定义的所在页。那些在本书中有源代码实现的函数还有“source code（源代码）”条目。

索引中的页码为英文原书页码，与书中页边标注的页码一致。

4.2BSD, 198
4.3BSD, 98
4.4BSD, 311, 315-316
4.4BSD-Lite, 537
64-bit architectures（64位体系结构），85, 427

abort function（abort函数），90, 424-425
absolute time（绝对时间），171
Abstract Syntax Notation One（抽象语法表示1），见ASN.1
accept function（accept函数），399
accept_stat member（accept_stat成员），447
accepted_reply structure, definition of（accepted_reply结构定义），447
access function（access函数），91
ACE (Adaptive Communications Environment)（适应性通信环境），180
address, IP（IP地址），245, 401, 403, 413, 422, 533
advisory locking（劝告性上锁），203-204, 217, 522
aio_return function（aio_return函数），91
aio_suspend function（aio_suspend函数），91
AIX, 151
alarm function（alarm函数），91, 396, 425
American National Standards Institute（美国国家标准局），见ANSI
American Standard Code for Information Interchange（美国标准信息互换代码），见ASCII
anonymous memory mapping（匿名内存映射），315-317
ANSI (American National Standards Institute), 21, 402-403, 505, 511, 520
API (application program interface), 13-14, 356, 379-380, 450, 536
  sockets（套接字API），8, 14, 151, 398-399, 403, 406, 449-450, 454-455
  TLI，406
  XTI，14, 151, 398-399, 403, 406, 413-414, 424, 449-450, 455
Apollo, 406
APUE (Advanced Programming in the UNIX Environment), 536
areply member（areply成员），447
arm（分支），429
array datatype, XDR（XDR数组数据类型），429
array member（array成员），288
ASCII (American Standard Code for Information Interchange), 193, 426, 429, 444
ASN.1 (Abstract Syntax Notation One), 426
Aspen Group, 178
asynchronous（异步的）
  event notification（异步事件通知），87
  I/O（异步I/O），14, 101
  procedure call（异步过程调用），356
async-signal-safe（异步信号安全），90-91, 95, 98, 102, 279, 525-526
at-least-once RPC call semantics（最少一次RPC调用语义），423, 450
at-most-once RPC call semantics（最多一次RPC调用语义），423, 450
atomic（原子的），24, 59, 197, 214, 220, 286
atomicity of pipe and FIFO writes（管道和FIFO写操作的原子性），65-66
attributes（属性）
  condition variable（条件变量属性），113, 172-174, 521
  doors（门属性），363, 366, 375, 384
  message queue（消息队列属性），79-82, 520
  mutex（互斥锁属性），172-174
  process-shared（进程间共享的属性），9-10, 113, 128, 173, 175, 265, 454

read-write lock（读写锁属性），179
thread（线程属性），98, 113, 502, 521, 532
aup_gid member（aup_gid成员），416
aup_gids member（aup_gids成员），416
aup_len member（aup_len成员），416
aup_machname member（aup_machname成员），416
aup_time member（aup_time成员），416
aup_uid member（aup_uid成员），416
AUTH_BADCRED constant（AUTH_BADCRED常值），449
AUTH_BADVERF constant（AUTH_BADVERF常值），449
AUTH_DES constant（AUTH_DES常值），417
AUTH_ERROR constant（AUTH_ERROR常值），448-449
AUTH_FAILED constant（AUTH_FAILED常值），449
AUTH_INVALIDRESP constant（AUTH_FAILED常值），449
AUTH_KERB constant（AUTH_KERB常值），417
AUTH_NONE constant（AUTH_NONE常值），414, 446-447, 533
AUTH_OK constant（AUTH_NONE常值），449
AUTH_REJECTEDCRED constant（AUTH_REJECTEDCRED常值），449
AUTH_REJECTEDVERF constant（AUTH_REJECTEDVERF常值），449
AUTH_SHORT constant（AUTH_SHORT常值），417, 446
AUTH_SYS constant（AUTH_SYS常值），414, 416, 446-447
AUTH_TOOWEAK constant（AUTH_TOOWEAK常值），449
auth_destroy function（auth_destroy函数），415
auth_flavor member（auth_flavor成员），446
auth_stat member（auth_stat成员），449
authentication（认证）
null（空认证），414
RPC（RPC认证），414-417
Unix（Unix认证），414
authsys_create_default function（authsys_create_default函数），415
authsys_parms structure（authsys_parms结构），416
definition of（authsys_parms结构定义），416, 446
autoconf program（autoconf程序），509
awk program（awk程序），13

Bach, M. J., 36, 535
bandwidth（带宽），457
performance, message passing（消息传递带宽性能），467-480
basename program（basename程序），13
Basic Encoding Rules（基本编码规则），见BER
Bass, J., 198
BER (Basic Encoding Rules), 426
Berg, D. J., 371, 536
bibliography（参考文献），535-537
big-endian byte order（大端字节序），403, 426, 444
binary semaphore（二值信号量），219, 281
bind function（bind函数），399
Birrell, A. D., 406, 535
black magic（神秘技巧），380
body member（body成员），446
bool datatype, XDR（XDR bool数据类型），429
Bostic, K., 311, 536
bounded buffer problem（有界缓冲区问题），161
Bourne shell, 13, 52, 72
BSD/OS, 53, 59, 66, 84, 111, 209-210, 213, 316, 403-405, 411-412, 425, 437, 456, 517
buf member（buf成员），288
buffers, multiple（多个缓冲区），249-256
BUFFSIZE constant, definition of（BUFFSIZE常值定义），507
bullet, silver（简单判定），453
Butenhof, D. R., 9, 95, 160, 163, 180, 192, 535
byte（字节）
order, big-endian（大端字节序），403, 426, 444
order, little-endian（小端字节序），403, 426
range（字节范围），197
stream（字节流），67, 74, 76, 444, 454
BYTES_PER_XDR_UNIT constant（BYTES_PER_XDR-VNIT常值），438

C function prototype（C函数原型），21, 105, 363, 384, 402-403, 505
C shell, 72
C standard（C标准），21, 90, 511, 520
C9X（C9X C标准），21
Technical Corrigendum（C标准技术性补遗），520
CALL constant（CALL常值），446
call semantics（调用语义）
at-least-once RPC（最少一次RPC调用语义），423, 450
at-most-once RPC（最多一次RPC调用语义），423, 450
exactly-once RPC（准确地一次RPC调用语义），422-423, 450
RPC（RPC调用语义），422-424
call_body structure, definition of（call_body结构定义），446
calloc function（calloc函数），84, 136
cancellation, thread（线程取消），174, 180, 183, 187-192, 384, 388, 396-398, 530
carriage return（回车符），见CR
cat program（cat程序），52-53, 64-66
cbody member（cbody成员），446
CDE (Common Desktop Environment), 15
Cedar, 406
cfgetispeed function（cfgetispeed函数），91
cfgetospeed function（cfgetospeed函数），91
cfsetispeed function（cfsetispeed函数），91
cfsetospeed function（cfsetospeed函数），91
cgid member（cgid成员），33-34, 131, 283
char datatype, XDR（XDR char数据类型），427
chdir function（chdir函数），91
chmod function（chmod函数），91
chmod program（chmod程序），205
chown function（chown函数），91
chown program（chown程序），33
cl_auth member（cl_auth成员），415
CLGET_RETRY_TIMEOUT constant（CLGET_RETRY_TIMEOUT常值），418
CLGET_TIMEOUT constant（CLGET_TIMEOUT常值），418
client（客户）

handle, definition of（客户句柄定义），401
identity（客户标识），83-84, 365, 369, 397, 415-417, 456
stub（客户程序存根），403, 405
client function(client函数), 48, 54-55, 72, 142, 144, 147, 149
CLIENT structure（CLIENT结构），401-402, 415
clnt_call function（clnt_call函数），419-420, 424, 451, 486, 534
clnt_control function（clnt_control函数），418-420
definition of（clnt_control函数定义），418
clnt_create function(clnt_create函数)，401, 403-405, 412-413, 418, 420, 532
definition of（clnt_create函数定义），401
clnt_destroy function（clnt_destroy函数），420
definition of（clnt_destroy函数定义），420
clnt_sperror function（clnt_sperror函数），424
clnt_stat structure（clnt_stat结构），409
clock_gettime function（clock_gettime函数），91
close function(close函数), 12, 61, 63, 65, 73, 77, 91, 114, 214, 260, 265, 279, 330, 376-378, 383-384, 524
CLSET_TIMEOUT constant（CLSET_TIMEOUT常值），420
coding style（编程风格），12, 90
Columbus Unix, 28
Common Desktop Environment（公共桌面环境），见CDE
concurrency, thread（线程并发度），163, 165-166, 488
concurrent server（并发服务器），66-67, 147, 357, 372, 407
condition variables（条件变量），159-175
attributes（条件变量属性），113, 172-174, 521
config.h header（config.h头文件），509-510
configure program（configure程序），509
connect function（connect函数），399
const datatype, XDR（XDR const数据类型），427
contention scope（竞用范围），386, 388, 462
conventions, source code（源代码约定），11
cooperating processes（协作进程），203
cooperative locks（协作锁），161
Coordinated Universal Time（国际标准时间），见UTC
copy-on-write（写时复制），501
Corbin, J. R., 406, 535
counting semaphore（计数信号量），221, 281
Courier, 406
Cox, J., 36, 311, 535
cpio program（cpio程序），13
CR (carriage return), 67
creat function（creat函数），91
creator ID（创建者ID），33
cred member（cred成员），446
credentials（凭证），417, 446, 449, 533
critical region（临界区），159, 177, 197
cuid member（cuid成员），33-34, 131, 283

d_attributes member（d_attributes成员），380, 384
d_data member（d_data成员），380
d_desc structure（d_desc结构），380
definition of（d_desc结构定义），380
d_descriptor member（d_descriptor成员），380
d_id member（d_id成员），380
daemon(守护程序; 守护进程), 60, 174, 203, 408, 502, 504, 511, 523
starting one copy（启动一个守护进程的唯一副本），213-214
daemon_proc variable（daemon_proc变量），511
Data Encryption Standard（数据加密标准），见DES
data_ptr member(data_ptr成员), 357, 362-363, 367-369
data_size member（data_size成员），357, 362, 530
datatypes, XDR（XDR数据类型），427-430
dc_egid member（dc_egid成员），365
dc_euid member（dc_euid成员），365
dc_pid member（dc_pid成员），365
dc_rgid member（dc_rgid成员），365
dc_ruid member（dc_ruid成员），365
DCE (Distributed Computing Environment), 407
deadlock（死锁），56, 143, 238, 279, 518, 523-524
DEBUG constant（DEBUG常值），408
delta time（相对时间），171
denial-of-service（拒绝服务型），见DoS
DES (Data Encryption Standard), 417
desc_num member（desc_num成员），357, 362-363, 530
desc_ptr member（desc_ptr成员），357, 362-363, 380
descriptor passing（描述符传递），84, 379-384
detached thread（已脱离线程），98, 384, 386-388, 504
/dev/clts device（/dev/clts设备），413
/dev/null device（/dev/null设备），526
/dev/zero device（/dev/zero设备），315-317, 322-323, 325, 454, 497, 526
/dev/zero memory mapping（/dev/zero内存映射），316-317
dg_echo function（dg_echo函数），256
di_attributes member（di_attributes成员），366
di_data member（di_data成员），366, 384
di_proc member（di_proc成员），366, 384, 386
di_target member（di_target成员），366
di_uniquifier member（di_uniquifier成员），366
Digital Unix, 15, 20-21, 37, 51, 73, 77, 79, 82, 98, 100, 104, 109, 154, 163, 209-210, 213, 225, 231-232, 238, 296, 319, 331, 333, 342, 351, 370, 407, 411-412, 437, 458-459, 461-462, 464, 466, 471, 489, 520-522, 524
Dijkstra, E. W., 220
DIR_MODE constant, definition of(DIR_MODE常值定义), 508
discriminant（判别式），429
discriminated union（带判别式的联合），429
Distributed Computing Environment（分布式计算环境），见DCE
Door_create_proc datatype（Door_create_proc数据类型），384
DOOR_DESCRIPTOR constant(DOOR_DESCRIPTOR常值), 380, 384
DOOR_LOCAL constant（DOOR_LOCAL常值），366
DOOR_PRIVATE constant（DOOR_PRIVATE常值），364, 366 386
DOOR_QUERY constant（DOOR_QUERY常值），366
DOOR_RELEASE constant（DOOR_RELEASE常值），384

DOOR_REVOKE constant（DOOR_REVOKE常值），366
Door_server_proc datatype（Door_server_proc数据类型），363
DOOR_UNREF constant（DOOR_UNREF常值），364, 366, 375-379
DOOR_UNREF_DATA constant（DOOR_UNREF_DATA常值），364, 375
door_arg_t structure（door_arg_t结构），363, 380-381
  definition of（door_arg_t结构定义），362
door_bind function（door_bind函数），377, 385-386, 388, 390, 532
  definition of（door_bind函数定义），390
door_call function（door_call函数），357-358, 360-364, 367, 369, 388, 390-393, 395-398, 422, 476, 484, 530-531
  definition of（door_call函数定义），361
door_create function（door_create函数），357-358, 361, 363-364, 375, 377, 379, 384-386, 388-389, 397-398, 531
  definition of（door_create函数定义），363
door_cred function（door_cred函数），365, 369
  definition of（door_cred函数定义），365
door_cred_t structure（door_cred_t结构），365
  definition of（door_cred_t结构定义），365
door_desc_t structure（door_desc_t结构），362-363, 380-381, 530
  definition of（door_desc_t结构定义），380
door_info function（door_info函数），365-367, 377, 530
  definition of（door_info函数定义），365
door_info_t structure（door_info_t结构），364, 366, 384, 386-387, 531
  definition of（door_info_t结构定义），366
door_return function（door_return函数），358, 361-362, 364-365, 377, 380, 383, 385, 387-388, 396-397
  definition of（door_return函数定义），365
door_revoke function（door_revoke函数），366, 377, 390, 398, 530-531
  definition of（door_revoke函数定义），390
door_server_create function（door_server_create函数），384-390
  definition of（door_server_create函数定义），384
door_unbind function（door_unbind函数），390
  definition of（door_unbind函数定义），390
doors（门），355-398
  attributes（门属性），363, 366, 375, 384
  premature termination of client（门客户过早终止），390-397
  premature termination of server（门服务器过早终止），390-397
  thread management（门线程管理），370-375
Dorado, 406
DoS (denial-of-service), 65-67
double buffering（双缓冲），251
double datatype, XDR（XDR double数据类型），427
dup function（dup函数），91
dup2 function（dup2函数），91
duplicate data（重复数据），418, 421, 451, 532
duplicate request cache, RPC server（RPC server重复请求高速缓存），421-424, 451, 532-533

E2BIG error（E2BIG错误），83, 133
EACCES error（EACCES错误），24, 32, 199, 216, 225
EAGAIN error（EAGAIN错误），12, 59-60, 93, 121, 124, 132, 199, 205, 227, 260, 269, 276, 286, 293, 339, 503, 522
EBADF error（EBADF错误），52, 530-531
EBUSY error（EBUSY错误），90, 121, 160, 178, 184, 192
echo program（echo程序），64
EDEADLK error（EDEADLK错误），238
EEXIST error（EEXIST错误），23-24, 31-32, 54, 74, 111, 214-215, 235, 260, 284, 294, 516, 524
effective（有效的）
  group ID（有效组ID），23, 25, 33-34, 131, 283, 365, 414, 416, 515
  user ID（有效用户ID），23, 25, 33-34, 84, 131, 283, 365, 369-370, 414, 416, 515
EIDRM error（EIDRM错误），132-133, 286
EINTR error（EINTR错误），90, 121, 124, 132-133, 149, 227, 279, 286, 391-394, 398, 521, 524-525
EMSGSIZE error（EMSGSIZE错误），13, 83
ENOBUFS error（ENOBUFS错误），341
ENOENT error（ENOENT错误），24, 32, 115, 516, 526
ENOMSG error（ENOMSG错误），133, 139
ENOSPC error（ENOSPC错误），24, 32
enum datatype, XDR（XDR enum数据类型），429
environment variable（环境变量）
  PATH（PATH环境变量），52
  PX_IPC_NAME（PATH环境变量），21
ENXIO error（ENXIO错误），59
ephemeral port（临时端口），404, 411, 414, 450
EPIPE error（EPIPE错误），60
err_doit function, source code（err_doit函数源代码），512
err_dump function（err_dump函数），511
  source code（err_dump函数源代码），512
err_msg function（err_msg函数），511
  source code（err_msg函数源代码），512
err_quit function（err_quit函数），381, 511
  source code（err_quit函数源代码），512
err_ret function（err_ret函数），511
  source code（err_ret函数源代码），511
err_sys function（err_sys函数），11-12, 511
  source code（err_sys函数源代码），511
errata availability（勘误表可获性），xvi
_errno function（_errno函数），515
errno variable（errno变量），11-13, 18, 49, 116, 267, 269, 274, 279, 502-503, 511, 515, 524
<errno.h> header（<errno.h>头文件），13, 18
error functions（错误处理函数），510-513
ESPIPE error（ESPIPE错误），54
ESRCH error（ESRCH错误），121
/etc/inetd.conf file（/etc/inetd.conf文件），413
/etc/netconfig file（/etc/netconfig文件），413
/etc/rpc file（/etc/rpc文件），412-413
/etc/sysconfigtab file（/etc/sysconfigtab文件），38

/etc/system file（/etc/system文件），37, 458
ETIMEDOUT error（ETIMEDOUT错误），171
exactly-once RPC call semantics（正好一次RPC调用语义），422-423, 450
examples road map（示例导读图），15-16
exec function（exec函数），9-10, 13, 58, 73, 364, 379-380, 398, 414, 502, 530
execle function（execle函数），91
execve function（execve函数），91
exercises, solutions to（习题解答），515-534
exit function（exit函数），9, 48, 90, 226, 504, 511, 521
_exit function（_exit函数），9-10, 91, 226, 504
explicit（显式的）
  file I/O（显式文件I/O），322
  network programming（显式网络编程），4, 399, 403
  synchronization（显式同步），161
  thread termination（显式线程终止），502
  typing（显式类型指定），426
external data representation（外部数据表示），见XDR

F_GETFL constant（F_GETFL常值），58
F_GETLK constant（F_GETLK常值），199-200
F_RDLCK constant（F_RDLCK常值），199
F_SETFL constant（F_SETFL常值），58-59
F_SETLK constant（F_SETLK常值），199-200, 522
F_SETLKW constant（F_SETLKW常值），199, 201, 522
F_UNLCK constant（F_UNLCK常值），199
F_WRLCK constant（F_WRLCK常值），199
FALSE constant（FALSE常值），409, 418, 429, 439, 441, 532
fattach function（fattach函数），357, 359, 364, 376-377, 379, 397
fcntl function（fcntl函数），58, 91, 174, 193-194, 198-200, 202, 205, 207, 214-217, 398, 418, 450, 455-456, 462, 495, 522-523
  definition of（fcntl函数定义），199
FD_CLOEXEC constant（FD_CLOEXEC常值），10, 364, 398
fdatasync function（fdatasync函数），91
fdetach function（fdetach函数），364, 376
fdetach program（fdetach程序），364
fdopen function（fdetach函数），68
fgets function（fgets函数），48, 53, 71, 249, 518
FIFO (first in, first out), 54-60
  limits（FIFO限制），72-73
  NFS and（FIFO和NFS），66
  order, lock requests（上锁请求FIFO顺序），210
  order, message queue（消息队列FIFO顺序），133, 138, 143
  order, queued signals（入队信号FIFO顺序），100, 102, 104-105
  order, RPC server reply cache（RPC服务器应答高速缓存FIFO顺序），422
  permissions（FIFO权限），54
  used for implementation of Posix semaphores（FIFO用于实现Posix信号量），257-262
  writes, atomicity of pipe and（管道和FIFO写操作的原子性），65-66
fifo.h header（fifo.h头文件），56
file I/O, explicit（显式文件I/O），322
file locking（文件上锁）
  using Posix semaphores（使用Posix信号量进行文件上锁），238
  using System V semaphores（使用System V信号量进行文件上锁），294-296
  versus record locking（对比文件上锁与记录上锁），197-198
file mode creation mask（文件模式创建掩码），23, 33, 55
file permissions（文件权限），203, 205, 216, 397
FILE structure（FILE结构），52, 401-402
File Transfer Protocol（文件传送协议），见FTP
FILE_MODE constant（FILE_MODE常值），55, 79
  definition of（FILE_MODE常值定义），508
filesystem persistence（随文件系统的持续性），6-7, 78, 311
FIN (finish flag, TCP header), 420, 424-425
find program（find程序），39, 517
finish flag, TCP header（TCP首部完成标志），见FIN
first in, first out（先进先出），见FIFO
flavor member（flavor成员），446
float datatype, XDR（XDR float数据类型），427
floating point format, IEEE（IEEE浮点格式），426
flock function（flock函数），198
flock structure（flock结构），199-201
  definition of（flock结构定义），199
fopen function（fopen函数），54, 68, 71, 149, 515
fork function（fork函数），4, 9-10, 13, 44-47, 51, 55, 58, 66-67, 73, 91, 102, 147, 149, 151, 174, 200, 207, 217, 240, 256, 267, 305, 307, 309, 311, 315, 322, 332, 364, 379-380, 391, 414, 475, 480, 497-498, 501, 503
fpathconf function（fpathconf函数），72-73, 91
fputs function（fputs函数），249
fragment, XDR（XDR片段），444
free function（free函数），21, 260, 275
FreeBSD, 29, 288
FSETLKW constant（FSETLKW常值），215
fstat function（fstat函数），21, 44, 74, 91, 115, 262, 327-328, 330-331, 342, 398, 519, 530
  definition of（fstat函数定义），328
fsync function（fsync函数），91
ftok function（ftok函数），28-31, 38-39, 130, 135, 138, 273, 275, 293, 344, 346, 348-349, 517, 521, 526
  definition of（ftok函数定义），28
FTP (File Transfer Protocol), 67, 337
ftruncate function（ftruncate函数），113, 217, 263, 320, 327-328, 333, 342, 351, 528
  definition of（ftruncate函数定义），327
full-duplex pipe（全双工管道），44, 50-52, 127, 475

GARBAGE_ARGS constant（GARBAGE_ARGS常值），447-448
Garfinkel, S. L., 417, 535
GETALL constant（GETALL常值），288, 290
getconf program（getconf程序），73
getegid function（getegid函数），91
geteuid function（geteuid函数），91

getgid function（getgid函数），91
getgroups function（getgroups函数），91
gethostbyaddr function（gethostbyaddr函数），245
gethostname function（gethostname函数），509
GETNCNT constant（GETNCNT常值），288
getopt function（getopt函数），78, 82
Getopt wrapper function（Getopt包裹函数），78
getpgrp function（getpgrp函数），91
GETPID constant（GETPID常值），288
getpid function（getpid函数），91, 370, 503
getppid function（getppid函数），91
getsockopt function（getsockopt函数），418
getuid function（getuid函数），91
GETVAL constant（GETVAL常值），277, 288
GETZCNT constant（GETZCNT常值），288
gf_time function（gf_time函数），207
gid member（gid成员），33-34, 131, 134, 283, 288, 345, 446
gids member（gids成员），446
GNU (GNU's Not Unix), 509, 520
Goodheart, B., 36, 311, 535
granularity, locking（上锁粒度），198
grep program（grep程序），161
group ID（组ID），328, 397, 417, 502
  effective（有效组ID），23, 25, 33-34, 131, 283, 365, 414, 416, 515
  real（实际组ID），365
  supplementary（辅助组ID），25, 414, 416

half-close（半关闭），425
Hamilton, C., 356, 535
Hewlett Packard, 407
high member（high成员），447, 449
hostname（主机名），245, 401, 403, 413-414, 416-417, 450
HTTP (Hypertext Transfer Protocol), 67, 337
hyper datatype, XDR（XDR hyper数据类型），427
Hypertext Transfer Protocol（超文本传送协议），见HTTP

I_RECVFD constant（I_RECVFD常值），379
I_SENDFD constant（I_SENDFD常值），379
idempotent（等势的），393-395, 422-423
identifier reuse, System V IPC（System V IPC标识符重用），34-36
identity, client（客户标识），83-84, 365, 369, 397, 415-417, 456
IEC (International Electrotechnical Commission), 13-14, 520, 536
IEEE (Institute of Electrical and Electronics Engineers), 13-14, 121, 180, 262, 536
  floating point format（IEEE浮点格式），426
IEEEIX, 13
implementation（实现）
  of Posix message queues using memory-mapped I/O（使用内存映射的I/O实现Posix消息队列），106-126
  of Posix read-write lock using mutexes and condition variables（使用互斥锁和条件变量实现Posix读写锁），179-187
  of Posix semaphores using FIFOs（使用FIFO实现Posix信号量），257-262
  of Posix semaphores using memory-mapped I/O（使用内存映射的I/O实现Posix信号量），262-270
  of Posix semaphores using System V semaphores（使用System V信号量实现Posix信号量），271-278
implicit（隐式的）
  synchronization（隐式同步），161
  thread termination（隐式线程终止），502
  typing（隐式类型指定），426
inetd program（inetd程序），413-414
  RPC and（RPC和inetd程序），413-414
init program（init程序），4, 48, 521
initial thread（初始线程），见main thread
i-node, 28-29, 349, 517
Institute of Electrical and Electronics Engineers（电气与电子工程师协会），见IEEE
int datatype, XDR（XDR int数据类型），427
int16_t datatype（int16_t数据类型），427
int32_t datatype（int32_t数据类型），427
int64_t datatype（int64_t数据类型），427
int8_t datatype（int8_t数据类型），427
International Electrotechnical Commission（国际电工委员会），见IEC
International Organization for Standardization（国际标准化组织），见ISO
Internet Protocol（网际网协议），见IP
Internet Protocol version 4（网际网协议版本4），见IPv4
interprocess communication（进程间通信），见IPC
ioctl function（ioctl函数），379, 384
IP (Internet Protocol), address（IP地址），245, 401, 403, 413, 422, 533
IPC (interprocess communication)
  identifier reuse, System V（System V IPC标识符重用），34-36
  kernel limits, System V（System V IPC内核限制），36-38
  key（IPC键），28
  name space（IPC名字空间），7-9
  names, Posix（Posix IPC名字），19-22
  networked（连网的IPC），453
  nonnetworked（非连网的IPC），453
  permissions, Posix（Posix IPC权限），23, 25-26, 84, 115, 225, 232, 267, 327
  permissions, System V（System V IPC权限），31-35, 39, 130-131, 282-283, 343-345
  persistence（IPC持续性），6-7
  Posix（Posix IPC），19-26
  System V（System V IPC），27-39
IPC_CREAT constant（IPC_CREAT常值），31-32, 38, 130, 283-284, 294, 344
IPC_EXCL constant（IPC_EXCL常值），31-32, 38, 130, 135, 141, 273, 283-284, 289, 294, 344
IPC_NOWAIT constant（IPC_NOWAIT常值），87, 132-133, 139, 143, 276, 286-287, 290
IPC_PRIVATE constant（IPC_NOWAIT常值），29-31, 38-39, 130, 134, 147, 155, 344, 517

IPC_RMID constant（IPC_RMID常值），35, 134, 137, 275, 288-289, 345-346, 351
IPC_SET constant（IPC_SET常值），33, 134, 288, 345
IPC_STAT constant（IPC_STAT常值），38, 134, 274, 285, 289-290, 294, 345, 347-348, 351, 455
ipc_perm structure（IPC_STAT结构），30-35, 38, 129-130, 282-283, 343
  definition of（IPC_STAT结构定义），30
ipcrm program（ipcrm程序），36
ipcs program（ipcs程序），36, 134, 138-140, 348-349, 455
IPv4 (Internet Protocol version 4), 446, 451, 534
is_read_lockable function, definition of（is_read_lockable函数定义），202
is_write_lockable function, definition of（is_write_lockable函数定义），202
ISO (International Organization for Standardization), 13-14, 520, 536
iterative, server（迭代服务器），66-67, 144, 372, 407-408

joinable thread（可联接线程），387, 504
Josey, A., 15, 536
justice, poetic, 517 （正文中未见本词条）

Karels, M. J., 311, 536
Kerberos, 417
kernel limits, System V IPC（System V IPC内核限制），36-38
kernel persistence（随内核的持续性），6, 75, 77, 226
Kernighan, B. W., 12, 511, 536
key, IPC（IPC键），28
key_t datatype（key_t数据类型），8, 28-30, 455
kill function（kill函数），91, 101
Kleiman, S., 180, 536
KornShell, 72-73
Kougiouris, P., 356, 535

l_len member（l_len成员），199-200
l_pid member（l_pid成员），199
l_start member（l_start成员），199-200
l_type member（l_type成员），199
l_whence member（l_whence成员），199-200
last in, first out（后进先出），见LIFO
latency（延迟），361, 458
  performance, message passing（消息传递性能），480-486
leak, memory（内存空间泄漏），114, 175, 452, 521
Leisner, M., xvi
Lewis, B., 371, 536
LF (linefeed), 67
LIFO (last in, first out), 104
lightweight process（轻权进程），501
limit program（limit程序），72
limits（限制）
  FIFO（FIFO限制），72-73
  pipe（管道限制），72-73
  Posix message queue（Posix消息队列限制），86-87
  Posix semaphore（Posix信号量限制），257
  System V IPC kernel（System V IPC内核限制），36-38
  System V message queue（System V消息队列限制），152-154
  System V semaphore（System V信号量限制），296-300
  System V shared memory（System V共享内存区限制），349-351
<limits.h> header（<limits.h>头文件），72
linefeed（换行符），见LF
link function（link函数），91, 215-216
Linux, 288, 356, 407
listen function（listen函数），399
little-endian byte order（小端字节序），403, 426
lmbench program（lmbench程序），458-459
local procedure call（本地过程调用），355
lock priority（锁优先级），180, 207-213
lock_reg function（lock_reg函数），202
lock_test function（lock_test函数），202
lockd program（lockd程序），216
lockf function（lockf函数），198
lockfcntl program（lockfcntl程序），203-204
locking（上锁）
  advisory（劝告性上锁），203-204, 217, 522
  conflicts（上锁冲突），170-171
  file locking versus record（文件上锁与记录上锁对比），197-198
  granularity（上锁粒度），198
  lock files（作锁用文件），214-216
  mandatory（强制性上锁），204-207, 217
  NFS（NFS上锁），216
  priorities of readers and writers（读出者和写入者的优先级），207-213
  record（记录上锁），193-217
  shared-exclusive（共享-独占上锁），177
  versus waiting（上锁与等待对比），165-167
locking function（locking函数），198
locknone program（locknone程序），203-204, 207, 217
LOG_ERR constant（LOG_ERR常值），511
LOG_INFO constant（LOG_INFO常值），511
long datatype, XDR（XDR long数据类型），427
long double datatype（long double数据类型），427
long long datatype（long long数据类型），427
longjmp function（longjmp函数），90
longlong_t datatype（longlong_t数据类型），427
loopfcntl program（loopfcntl程序），205-206, 217
loopfcntlnonb program（loopfcntlnonb程序），217, 522
loopnone program（loopnone程序），205-206
loopnonenonb program（loopnonenonb程序），217, 522
low member（low成员），447, 449
lp program（lp程序），193
LP64, 427
lpr program（lpr程序），193
ls program（ls程序），36, 81, 205, 360, 455
lseek function（lseek函数），5, 54, 91, 113, 115, 200, 202, 310, 322, 327, 523, 528
lstat function（lstat函数），21, 44
Lyon, B., 406

machinename member (machinename成员), 446
magic number (魔数), 109, 117, 181, 258, 262, 271
main thread (主线程), 93, 190, 235, 388, 488, 490, 502
malloc function(malloc函数), 117, 160, 432, 435, 467-468, 524, 533
mandatory locking (强制性上锁), 204-207, 217
many-to-few thread implementation(多对少线程实现), 163
MAP_ANON constant (MAP_ANON常值), 315-316, 322, 454, 497
MAP_FIXED constant (MAP_FIXED常值), 309
MAP_PRIVATE constant (MAP_PRIVATE常值), 309-310, 323
MAP_SHARED constant (MAP_SHARED常值), 309-311, 315, 323
marshaling (集结), 405
MAX_PATH constant, definition of (MAX_PATH常值定义), 507
MAXLINE constant (MAXLINE常值), 49, 505
  definition of (MAXLINE常值定义), 507
McKusick, M. K., 311, 536
McVoy, L., 458, 536
memcpy function (memcpy函数), 127, 526
memory (内存)
  leak (内存空间泄漏), 114, 175, 452, 521
  mapping, anonymous (匿名内存映射), 315-317
  mapping, /dev/zero (/dev/zero内存映射), 316-317
  object (内存区对象), 326
memory-mapped (内存映射的)
  file (内存映射文件), 78, 107, 111, 127, 308, 310-311, 313, 322, 325-326, 471, 520, 525
  I/O (内存映射I/O), 303, 525
  I/O, used for implementation of Posix message queues (内存映射I/O用于实现Posix消息队列), 106-126
  I/O, used for implementation of Posix semaphores (内存映射I/O用于实现Posix信号量), 262-270
mesg structure (mesg结构), 149
mesg_recv function(mesg_recv函数), 69-71, 141-142, 144, 149
mesg_send function(mesg_send函数), 69-70, 141-142, 144
mesg.h header (mesg.h头文件), 68
Mesg_recv function (Mesg_recv函数), 149
message (消息)
  boundary (消息边界), 67, 76, 444, 454
  queue attributes (消息队列属性), 79-82, 520
  queue descriptor, definition of (消息队列描述符定义), 77
  queue ID(消息队列ID), 129-130, 139-140, 142, 147, 149, 151, 154
  queue limits, Posix (Posix消息队列限制), 86-87
  queue limits, System V (System V消息队列限制), 152-154
  queue priority (消息队列优先级), 82-83, 85-86, 109, 123-124, 126, 143, 482
  queues, implementation using memory-mapped I/O, Posix (使用内存映射I/O实现Posix消息队列), 106-126
  queues, Posix (Posix消息队列), 75-128
  queues, System V (System V消息队列), 129-155
  queues with poll function, System V (System V带poll函数的消息队列), 151-152
  queues with select function, Posix (Posix带select函数的消息队列), 95-98
  queues with select function, System V (System V带select函数的消息队列), 151-152
messages (消息)
  multiplexing (复用消息), 142-151
  streams versus (对比流和消息), 67-72
mismatch_info structure, definition of (mismatch_info结构定义), 447, 449
mkdir function (mkdir函数), 91
mkfifo function (mkfifo函数), 39, 54-58, 74, 91, 518, 524
  definition of (mkfifo函数定义), 54
mkfifo program (mkfifo程序), 54
mlock function (mlock函数), 322
mlockall function (mlockall函数), 322
mmap function(mmap函数), 14, 109, 113, 115, 263, 265, 303, 307-311, 315-320, 322-323, 325-328, 330-334, 337, 342-343, 363, 369, 471, 527, 529
  definition of (mmap函数定义), 308
mode member (mode成员), 31-34, 134, 283, 289, 345
mode_t datatype (mode_t数据类型), 110-111
MQ_OPEN_MAX constant (MQ_OPEN_MAX常值), 86
MQ_PRIO_MAX constant (MQ_PRIO_MAX常值), 82-83, 86
mq_attr structure (mq_attr结构), 80, 83
  definition of (mq_attr结构定义), 80
mq_close function (mq_close函数), 76-79, 109, 116-117, 126-127
  definition of (mq_close函数定义), 77
  source code (mq_close函数源代码), 116
mq_curmsgs member (mq_curmsgs成员), 80, 123-124
mq_flags member (mq_flags成员), 80, 108, 118
mq_getattr function (mq_getattr函数), 79-83, 85, 117, 126, 520
  definition of (mq_getattr函数定义), 79
  source code (mq_getattr函数源代码), 118
mq_hdr structure (mq_hdr结构), 109, 113, 117, 119
mq_info structure (mq_info结构), 106, 108-109, 113, 115-118
mq_maxmsg member(mq_maxmsg成员), 76, 80, 86, 112, 123, 127
mq_msgsize member (mq_msgsize成员), 76, 80, 83, 86, 112, 127
mq_notify function (mq_notify函数), 87-99, 117, 119, 126-127
  definition of (mq_notify函数定义), 87
  source code (mq_notify函数源代码), 120
mq_open function(mq_open函数), 19-20, 22, 25, 76-80, 82, 106, 109, 111-114, 116, 126-127, 326-327, 520
  definition of (mq_open函数定义), 76
  source code (mq_open函数源代码), 109
mq_receive function(mq_receive函数), 24, 76, 82-86, 88, 90, 93, 115, 121, 124, 126, 482, 526
  definition of (mq_receive函数定义), 83
  source code (mq_receive函数源代码), 125

mq_send function (mq_send函数), 13, 24, 82-86, 109, 121, 124, 126-127, 471, 526
  definition of (mq_send函数定义), 83
  source code (mq_send函数源代码), 122
mq_setattr function (mq_setattr函数), 79-82, 118, 126
  definition of (mq_setattr函数定义), 79
  source code (mq_setattr函数源代码), 119
mq_unlink function (mq_unlink函数), 76-79, 117, 126, 327
  definition of (mq_unlink函数定义), 77
  source code (mq_unlink函数源代码), 117
mqd_t datatype (mqd_t数据类型), 8, 77, 95, 109, 326
mqh_attr structure (mqh_attr结构), 108
mqh_event structure (mqh_event结构), 119
mqh_free member (mqh_free成员), 108-109, 113
mqh_head member (mqh_head成员), 108-109, 113, 124
mqh_nwait member (mqh_nwait成员), 121, 124
mqh_pid member (mqh_pid成员), 119
mqh_wait member (mqh_wait成员), 121
MQI_MAGIC constant (MQI_MAGIC常值), 109
mqi_flags member (mqi_flags成员), 109
mqi_hdr member (mqi_hdr成员), 109
mqi_magic member (mqi_magic成员), 109
mqueue.h header (mqueue.h头文件), 106
MS_ASYNC constant (MS_ASYNC常值), 310
MS_INVALIDATE constant (MS_INVALIDATE常值), 310
MS_SYNC constant (MS_SYNC常值), 310
MSG_ACCEPTED constant (MSG_ACCEPTED常值), 447-448
MSG_DENIED constant (MSG_DENIED常值), 447-448
MSG_NOERROR constant (MSG_NOERROR常值), 83, 133
MSG_PEEK constant (MSG_PEEK常值), 152, 455
MSG_R constant (MSG_R常值), 33
MSG_TRUNC constant (MSG_TRUNC常值), 83
MSG_W constant (MSG_W常值), 33
msg_cbytes member (msg_cbytes成员), 129, 134
msg_ctime member (msg_ctime成员), 129, 131
msg_first member (msg_first成员), 129
msg_hdr structure (msg_hdr结构), 109, 113, 123, 126, 310
msg_last member (msg_last成员), 129
msg_len member (msg_len成员), 109
msg_lrpid member (msg_lrpid成员), 129, 131
msg_lspid member (msg_lspid成员), 129, 131
msg_next member (msg_next成员), 108-109, 124
msg_perm structure (msg_perm结构), 131, 134
  definition of (msg_perm结构定义), 129
msg_prio member (msg_prio成员), 109
msg_qbytes member (msg_qbytes成员), 129, 131-132, 134
msg_qnum member (msg_qbytes成员), 129, 131
msg_rtime member (msg_rtime成员), 129, 131
msg_stime member (msg_stime成员), 129, 131
msg_type member (msg_type成员), 446
msgbuf structure (msgbuf结构), 131, 134, 136, 482
  definition of (msgbuf结构定义), 131
msgctl function (msgctl函数), 35, 38, 134-135, 137
  definition of (msgctl函数定义), 134
msgget function (msgget函数), 33-35, 38, 130-131, 135, 139, 154, 516-517
  definition of (msgget函数), 130
msghdr structure (msghdr结构), 126
msgmap variable (msgmap变量), 37
msgmax variable (msgmax变量), 37-38, 152, 458
msgmnb variable (msgmnb变量), 37-38, 152, 458
msgmni variable (msgmni变量), 37-38, 152
msgrcv function (msgrcv函数), 83, 87, 131-134, 137-139, 143, 149, 151-152, 304, 323, 482
  definition of (msgrcv函数定义), 132
msgseg variable (msgseg变量), 37, 152, 458
msgsnd function (msgsnd函数), 34, 131-132, 135, 143, 154, 304
  definition of (msgsnd函数定义), 131
msgssz variable (msgssz变量), 37, 152
msgtql variable (msgtql变量), 37-38, 152
msqid_ds structure (msqid_ds结构), 130, 132, 134
  definition of (msqid_ds结构定义), 129
msync function (msync函数), 307-311
  definition of (msync函数定义), 310
mtext member (mtext成员), 131
M-to-N thread implementation (M对N线程实现), 163
mtype member (mtype成员), 131
multiple buffers (多个缓冲区), 249-256
multiplexing messages (复用消息), 142-151
multithreading, RPC (RPC多线程化), 407-411
munlock function (munlock函数), 322
munlockall function (munlockall函数), 322
munmap function (munmap函数), 117, 267, 307-311, 363, 369, 529
  definition of (munmap函数定义), 309
mutex (互斥锁), 159-175
  and condition variables, used for implementation of Posix read-write lock (互斥锁和条件变量用于实现Posix读写锁), 179-187
  attributes (互斥锁属性), 172-174
mutual exclusion (互斥), 159, 194, 221
my_create function (my_create函数), 386-387
my_lock function (my_lock函数), 194, 196-197, 200-202, 214, 217, 238, 279, 294, 296, 526
my_shm function (my_shm函数), 323, 497-498
my_thread function (my_thread函数), 386-388, 531
my_unlock function (my_unlock函数), 194, 196-197, 200, 202, 238, 279, 294
mymesg structure (mymesg结构), 68

name space, IPC (IPC名字空间), 7-9
named pipe (有名管道), 43, 54
names, Posix IPC (Posix IPC名字), 19-22
National Optical Astronomy Observatories (国家光学天文台), 见NOAO
NCA (Network Computing Architecture), 406
NCK (Network Computing Kernel), 407
NCS (Network Computing System), 406
NDR (Network Data Representation), 406

Nelson, B. J., 406, 535
network programming, explicit（显式网络编程），4, 399, 403
Network Computing Architecture（网络计算体系结构），见NCA
Network Computing Kernel（网络计算内核），见NCK
Network Computing System（网络计算系统），见NCS
Network Data Representation（网络数据表示），见NDR
Network File System（网络文件系统），见NFS
Network Interface Definition Language（网络接口定义语言），见NIDL
Network News Transfer Protocol（网络新闻传送协议），见NNTP
networked IPC（连网的IPC），453
NFS (Network File System), 404, 406, 411, 417, 495
  and FIFO（NFS与FIFO），66
  locking（NFS上锁），216
  secure（安全NFS），417
NIDL (Network Interface Definition Language), 406
NNTP (Network News Transfer Protocol), 67
nonblocking（非阻塞的），24, 58-59, 80, 85, 87, 93, 109, 132, 143, 160, 184, 205, 217, 260, 262, 269, 276, 286, 293, 518, 522
noncooperating processes（非协作进程），203-204
nondeterministic（非确定的），197, 217, 530
nonnetworked IPC（非连网的IPC），453
ntohl function, 441
null（空）
  authentication（空认证），414
  procedure（空过程），451, 486, 534
  signal（空信号），121

O_APPEND constant（O_APPEND常值），515
O_CREAT constant（O_CREAT常值），22-25, 31, 54, 77, 110-111, 115, 214-216, 225, 228-229, 239, 258, 260, 263, 265, 273-274, 279, 285, 327, 334, 516, 524
O_EXCL constant（O_EXCL常值），22-25, 31, 54, 77, 111, 214-215, 225, 235, 260, 273, 327, 516
O_NONBLOCK constant(O_NONBLOCK常值)，22, 24, 58-60, 77, 93, 121, 124, 217, 260, 518
O_RDONLY constant(O_RDONLY)，22, 25-26, 61, 63, 77, 115, 225, 327
O_RDWR constant(O_RDWR常值)，22, 25-26, 77, 115, 225, 327
O_TRUNC constant（O_TRUNC常值），22, 24, 216-217, 327, 523
O_WRONLY constant(O_WRONLY常值)，22, 25-26, 61, 77, 115, 216, 225
oa_base member（oa_base成员），416
oa_flavor member（oa_flavor成员），416
oa_length member（oa_length成员），416
od program（od程序），313, 319, 331
ONC (Open Network Computing), 406
opaque data（不透明数据），429
opaque datatype, XDR（XDR opaque数据类型），429
opaque_auth structure, definition of（opaque_auth结构定义），416, 446
open systems interconnection（开放系统互连），见OSI
open function（open函数），22-23, 26, 31, 49, 54, 56, 58, 61, 63, 65-66, 71, 74, 91, 111, 115, 214-217, 260, 265, 273, 310-311, 315-317, 325-327, 342, 357, 361, 364, 367, 376, 379-380, 382-383, 397, 515, 518, 523-524
Open Group, The（开放团体），14-15
Open Network Computing（开放网络计算），见ONC
Open Software Foundation（开放软件基金会），见OSF
OPEN_MAX constant（OPEN_MAX常值），72-73
Operation Support Systems（操作支持系统），28
optarg variable（optarg变量），82
optind variable（optind变量），78
OSF (Open Software Foundation), 14
OSI (open systems interconnection), 426
owner ID（属主ID），25, 33, 38, 397

packet formats, RPC（RPC分组格式），444-449
PATH environment variable（PATH环境变量），52
PATH_MAX constant（PATH_MAX常值），19, 22
pathconf function（pathconf函数），72-73, 91
pause function（pause函数），90-91, 230, 359, 420
pclose function（pclose函数），52-53, 73
  definition of（pclose函数定义），52
_PC_PIPE_BUF constant（_PC_PIPE_BUF常值），72
PDP-11, 37
performance（性能），457-499
  message passing bandwidth（消息传递带宽性能），467-480
  message passing latency（消息传递延迟性能），480-486
  process synchronization（进程同步性能），497-499
  thread synchronization（线程同步性能），486-496
permissions（权限）
  FIFO（FIFO权限），54
  file（文件权限），203, 205, 216, 397
  Posix IPC（Posix IPC权限），23, 25-26, 84, 115, 225, 232, 267, 327
  System V IPC（System V IPC权限），31-35, 39, 130-131, 282-283, 343-345
persistence（持续性），6
  filesystem（随文件系统的持续性），6-7, 78, 311
  IPC（随IPC的持续性），6-7
  kernel（随内核的持续性），6, 75, 77, 226
  process（随进程的持续性），6
pid_t datatype（pid_t数据类型），194
Pike, R., 12, 536
pipe（管道），44-53
  and FIFO writes, atomicity of（管道和FIFO写操作的原子性），65-66
  full-duplex（全双工管道），44, 50-52, 127, 475
  limits（管道限制），72-73
  named（有名管道），43, 54
pipe function（pipe函数），44, 50, 56, 58, 68, 73, 91
  definition of（pipe函数定义），44
PIPE_BUF constant（PIPE_BUF常值），59-60, 65, 72-73, 260
poll function（poll函数），95, 151, 155, 171, 339, 454
  System V message queues with（带poll函数的System V

消息队列），151-152
polling（轮询），87, 167, 214
popen function（popen函数），52-53, 73-74, 518
definition of（popen函数定义），52
port（端口）
ephemeral（临时端口），404, 411, 414, 450
mapper（端口影射器），404, 406, 411-414, 450-451, 532
reserved（保留端口），417
Portable Operating System Interface（可移植操作系统接口），见Posix
portmap program（portmap程序），411
Posix (Portable Operating System Interface), 13-14
IPC（Posix IPC），19-26
IPC names（Posix IPC名字），19-22
IPC permissions(Posix IPC权限)，23, 25-26, 84, 115, 225, 232, 267, 327
message queue limits（Posix消息队列限制），86-87
message queues（Posix消息队列），75-128
message queues, implementation using memory-mapped I/O（使用内存映射I/O实现Posix消息队列），106-126
message queues with select function（带select函数的Posix消息队列），95-98
read-write lock, implementation using mutexes and condition variables(使用互斥锁和条件变量实现Posix读写锁），179-187
realtime signals（Posix实时信号），98-106
semaphore limits（Posix信号量限制），257
semaphores（Posix信号量），219-279
semaphores between processes（进程间共享Posix信号量），256-257
semaphores, file locking using（使用Posix信号量进行文件上锁），238
semaphores, implementation using FIFOs(使用FIFO实现Posix信号量），257-262
semaphores, implementation using memory-mapped I/O（使用内存映射I/O实现Posix信号量），262-270
semaphores, implementation using System V semaphores（使用System V信号量实现Posix信号量），271-278
shared memory（Posix共享内存区），325-342
Posix.1, 8, 14-16, 19, 44, 59, 73, 83, 87, 98, 101, 159, 173, 178, 198, 205, 214, 225, 240, 256, 266, 279, 309, 325, 328, 364, 468, 482, 530, 536
definition of（Posix.1定义），14
Rationale（Posix.1原理），14, 223, 240, 262, 328
Posix.1b, 14, 99, 536
Posix.1c, 14, 536
Posix.1g, 8
Posix.1i, 14, 536
Posix.1j, 178, 488
Posix.2, 14-16
definition of（Posix.2定义），13
Posix.4, 99
POSIX_IPC_PREFIX constant（POSIX_IPC_PREFIX常值），22
_POSIX_C_SOURCE constant（_POSIX_C_SOURCE常值），13
_POSIX_MAPPED_FILES constant（_POSIX_MAPPED_FILES常值），9
_POSIX_MESSAGE_PASSING constant（_POSIX_MESSAGE_PASSING常值），9
_POSIX_REALTIME_SIGNALS constant（_POSIX_REALTIME_SIGNALS常值），9
_POSIX_SEMAPHORES constant（_POSIX_SEMAPHORES常值），9
_POSIX_SHARED_MEMORY_OBJECTS constant（_POSIX_SHARED_MEMORY_OBJECTS常值），9
_POSIX_THREAD_PROCESS_SHARED constant（_POSIX_THREAD_PROCESS_SHARED常值），9, 173
_POSIX_THREADS constant（_POSIX_THREADS常值），8-9
pr_thread_id function（pr_thread_id函数），370-371
source code（pr_thread_id函数源代码），371
printf function(printf函数)，90, 102, 127, 205, 217, 279, 383, 398, 408, 522
priority（优先级）
lock（锁优先级），180, 207-213
message queue（消息队列优先级），82-83, 85-86, 109, 123-124, 126, 143, 482
thread（线程优先级），160, 502
private server pool（私用服务器池），386, 388, 390
proc member（proc成员），446
PROC_UNAVAIL constant（PROC_UNAVAIL常值），447-448
procedure call（过程调用）
asynchronous（异步过程调用），356
local（本地过程调用），355
synchronous（同步过程调用），356-357, 476
procedure, null（空过程），451, 486, 534
process（进程）
lightweight（轻权进程），501
persistence（随进程的持续性），6
processes, cooperating（协作进程），203
process-shared attribute（进程间共享的属性），9-10, 113, 128, 173, 175, 265, 454
producer-consumer problem（生产者-消费者问题），161-165, 233-238, 242-249
prog member（prog成员），446
PROG_MISMATCH constant（PROG_MISMATCH常值），447-448
PROG_UNAVAIL constant（PROG_UNAVAIL常值），447-448
PROT_EXEC constant（PROT_EXEC常值），309
PROT_NONE constant（PROT_NONE常值），309
PROT_READ constant（PROT_READ常值），308-309
PROT_WRITE constant（PROT_WRITE常值），308-309
ps program（ps程序），127, 175, 367, 452, 520
pselect function（pselect函数），171
PTHREAD_CANCEL constant（PTHREAD_CANCEL常值），188
PTHREAD_COND_INITIALIZER constant（PTHREAD_COND_INITIALIZER常值），167, 172
PTHREAD_MUTEX_INITIALIZER constant（PTHREAD_MUTEX_INITIALIZER常值），160, 172
Pthread_mutex_lock wrapper function, source code（Pthread_mutex_lock包裹函数源代码），12
PTHREAD_PROCESS_PRIVATE constant（PTHREAD_PROCESS_PRIVATE常值），173, 179
PTHREAD_PROCESS_SHARED constant（PTHREAD_PROCESS_

SHARED常值), 113, 128, 173, 179, 193, 239, 256, 265, 462, 497-498
PTHREAD_RWLOCK_INITIALIZER constant (PTHREAD_RWLOCK_INITIALIZER常值), 178-179
PTHREAD_SCOPE_PROCESS constant (PTHREAD_SCOPE_PROCESS常值), 387
PTHREAD_SCOPE_SYSTEM constant (PTHREAD_SCOPE_SYSTEM常值), 386, 388
pthread_attr_destroy function (pthread_attr_destroy函数), 398
pthread_attr_init function (pthread_attr_init函数), 398
pthread_attr_t datatype (pthread_attr_t数据类型), 502
pthread_cancel function (pthread_cancel函数), 187, 190
  definition of (pthread_cancel函数定义), 187
pthread_cleanup_pop function (pthread_cleanup_pop函数), 187, 191
  definition of (pthread_cleanup_pop函数定义), 187
pthread_cleanup_push function (pthread_cleanup_push函数), 187, 396
  definition of (pthread_cleanup_push函数定义), 187
pthread_condattr_destroy function (pthread_condattr_destroy函数), 175
  definition of (pthread_condattr_destroy函数定义), 172
pthread_condattr_getpshared function, definition of (pthread_condattr_getpshared函数定义), 173
pthread_condattr_init function (pthread_condattr_init函数), 114, 175
  definition of (pthread_condattr_init函数定义), 172
pthread_condattr_setpshared function, definition of (pthread_condattr_setpshared函数定义), 173
pthread_condattr_t datatype (pthread_condattr_t数据类型), 172
pthread_cond_broadcast function (pthread_cond_broadcast函数), 171, 175, 186
  definition of (pthread_cond_broadcast函数定义), 171
pthread_cond_destroy function, definition of (pthread_cond_destroy函数定义), 172
pthread_cond_init function, definition of (pthread_cond_init函数定义), 172
pthread_cond_signal function (pthread_cond_signal函数), 124, 126, 167-171, 175, 186-187, 227, 268-269
  definition of (pthread_cond_signal函数定义), 167
pthread_cond_t datatype (pthread_cond_t数据类型), 8, 167, 256
pthread_cond_timedwait function (pthread_cond_timedwait函数), 171
  definition of (pthread_cond_timedwait函数定义), 171
pthread_cond_wait function (pthread_cond_wait函数), 121, 167-171, 175, 183-184, 187, 190-192, 227, 269, 525
  definition of (pthread_cond_wait函数定义), 167
pthread_create function (pthread_create函数), 163, 217, 356, 385-388, 502-504
  definition of (pthread_create函数定义), 502
pthread_detach function (pthread_detach函数), 502-504
  definition of (pthread_detach函数定义), 504
pthread_exit function (pthread_exit函数), 174, 187, 391, 425, 502-504
  definition of (pthread_exit函数定义), 504
pthread_join function (pthread_join函数), 357, 387, 502-504
  definition of (pthread_join函数定义), 503
pthread_mutexattr_destroy function (pthread_mutexattr_destroy函数), 175
  definition of (pthread_mutexattr_destroy定义), 172
pthread_mutexattr_getpshared function, definition of (pthread_mutexattr_getpshared函数定义), 173
pthread_mutexattr_init function (pthread_mutexattr_init函数), 113-114, 175, 265
  definition of (pthread_mutexattr_init函数定义), 172
pthread_mutexattr_setpshared function (pthread_mutexattr_setpshared函数), 113, 265
  definition of (pthread_mutexattr_setpshared函数定义), 173
pthread_mutexattr_t datatype (pthread_mutexattr_t数据类型), 172-173
pthread_mutex_destroy function, definition of (pthread_mutex_destroy函数定义), 172
pthread_mutex_init function (pthread_mutex_init函数), 113, 160, 172-173, 265, 498
  definition of (pthread_mutex_init函数定义), 172
pthread_mutex_lock function (pthread_mutex_lock函数), 12, 160, 190, 221
  definition of (pthread_mutex_lock函数定义), 160
pthread_mutex_t datatype (pthread_mutex_t数据类型), 8, 160, 172, 256, 279
pthread_mutex_trylock function (pthread_mutex_trylock函数), 160
  definition of (pthread_mutex_trylock函数定义), 160
pthread_mutex_unlock function (pthread_mutex_unlock函数), 221
  definition of (pthread_mutex_unlock函数定义), 160
pthread_rwlockattr_destroy function, definition of (pthread_rwlockattr_destroy函数定义), 179
pthread_rwlockattr_getpshared function, definition of (pthread_rwlockattr_getpshared函数定义), 179
pthread_rwlockattr_init function, definition of (pthread_rwlockattr_init函数定义), 179
pthread_rwlockattr_setpshared function, definition of (pthread_rwlockattr_setpshared函数定义), 179
pthread_rwlockattr_t datatype (pthread_rwlockattr_setpshared数据类型), 179
pthread_rwlock_destroy function (pthread_rwlock_destroy函数), 179, 181, 192
  definition of (pthread_rwlock_destroy函数定义), 179
  source code (pthread_rwlock_destroy函数源代码), 182
pthread_rwlock.h header (pthread_rwlock.h头文件),

180
pthread_rwlock_init function (pthread_rwlock_init函数), 179, 181
definition of (pthread_rwlock_init函数定义), 179
source code (pthread_rwlock_init函数源代码), 182
pthread_rwlock_rdlock function (pthread_rwlock_rdlock函数), 178-179, 183, 190-191
definition of (pthread_rwlock_rdlock函数定义), 178
source code (pthread_rwlock_rdlock函数源代码), 183
pthread_rwlock_t datatype (pthread_rwlock_t数据类型), 8, 178, 180-181, 183, 188, 193, 256
pthread_rwlock_tryrdlock function (pthread_rwlock_tryrdlock函数), 184
definition of (pthread_rwlock_tryrdlock函数定义), 178
source code (pthread_rwlock_tryrdlock函数源代码), 184
pthread_rwlock_trywrlock function (pthread_rwlock_trywrlock函数), 184
definition of (pthread_rwlock_trywrlock函数定义), 178
source code (pthread_rwlock_trywrlock函数源代码), 185
pthread_rwlock_unlock function (pthread_rwlock_unlock函数), 178-179, 186, 190, 192
definition of (pthread_rwlock_unlock函数定义), 178
source code (pthread_rwlock_unlock函数源代码), 186
pthread_rwlock_wrlock function (pthread_rwlock_wrlock函数), 178-179, 183-184, 190-191
definition of (pthread_rwlock_wrlock函数定义), 178
source code (pthread_rwlock_wrlock函数源代码), 185
pthread_self function (pthread_self函数), 502-504
definition of (pthread_self函数定义), 503
pthread_setcancelstate function (pthread_setcancelstate函数), 396, 530
pthread_setconcurrency function (pthread_setconcurrency函数), 163
pthread_sigmask function (pthread_sigmask函数), 95
pthread_t datatype (pthread_sigmask数据类型), 370-371, 502
<pthread.h> header (<pthread.h>头文件), 180
Pthreads, 15
putchar function (putchar函数), 217
PX_IPC_NAME environment variable (PX_IPC_NAME环境变量), 21
px_ipc_name function (px_ipc_name函数), 21-22, 26, 78, 235, 505
definition of (px_ipc_name函数定义), 21
source code (px_ipc_name函数源代码), 22

quadruple datatype, XDR (XDR quadruple数据类型), 427
Quarterman, J. S., 311, 536
queued signals (排队的信号), 100, 102
FIFO order (排队的信号的FIFO顺序), 100, 102, 104-105

raise function (raise函数), 91
rbody member (rbody成员), 446
rbuf member (rbuf成员), 357, 362-363, 367-369
read ahead (超前读), 251
read function(read函数), 5-6, 43, 49-52, 54, 59, 61, 63, 70, 83, 90-91, 142, 161, 200, 204-207, 249, 254, 260, 262-263, 278, 304, 310-311, 322, 399, 406, 435, 451, 456-457, 467, 469, 471, 482, 517-519, 522-523, 525-526, 533
read_lock function (read_lock函数), 207
definition of (read_lock函数定义), 202
readers-and-writers (读出者与写入者)
locks (读出者与写入者锁), 178
problem (读出者与写入者问题), 177
readline function (readline函数), 61, 63, 74, 518
readw_lock function (readw_lock函数), 207-208
definition of (readw_lock函数定义), 202
read-write lock (读写锁), 177-192
attributes (读写锁属性), 179
implementation using mutexes and condition variables, Posix (使用互斥锁和条件变量实现Posix读写锁), 179-187
real (实际的)
group ID (实际组ID), 365
user ID (实际用户ID), 365, 369
realtime (实时的)
scheduling (实时调度), 14, 160, 171, 454
signals, Posix (Posix实时信号), 98-106
record (记录), 75
locking (记录上锁), 193-217
locking, file locking versus(对比文件上锁与记录上锁), 197-198
recv function (recv函数), 152
recvfrom function (recvfrom函数), 152, 406
recvmsg function (recvmsg函数), 83, 152
_REENTRANT constant (_REENTRANT常值), 13, 515
reject_stat member (reject_stat成员), 449
rejected_reply structure, definition of (reject_stat结构定义), 449
remote procedure call (远程过程调用), 见RPC
remote procedure call language (远程过程调用语言), 见RPCL
remote procedure call source code (远程过程调用源代码), 见RPCSRC
remote terminal protocol (远程终端协议), 见Telnet
rename function (rename函数), 91
REPLY constant (REPLY常值), 446
reply_body structure, definition of (reply_body结构定义), 447
reply_stat member (reply_stat成员), 447
Request for Comments (请求评注), 见RFC
reserved port (保留端口), 417
reset flag, TCP header (TCP首部复位标志), 见RST
results member (results成员), 447
retransmission (重传), 424, 532

RPC timeout and（RPC超时与重传），417-422
RFC (Request for Comments)
1831（RFC 1831），406, 430, 446-447
1832（RFC 1832），406, 426, 430
1833（RFC 1833），406, 412
Ritchie, D. M., 511, 536
rm program（rm程序），36, 376-377, 379
rmdir function（rmdir函数），91
RNDUP function（RNDUP函数），438
road map, examples（示例导读图），15-16
Rochkind, M. J., 27, 536
round-trip time（往返时间），451, 458
RPC (remote procedure call), 355, 399-452
and inetd program（RPC与RNDUP程序），413-414
authentication（RPC认证），414-417
call semantics（RPC调用语义），422-424
call semantics, at-least-once（最少一次RPC调用语义），423, 450
call semantics, at-most-once（最多一次RPC调用语义），423, 450
call semantics, exactly-once（正好一次RPC调用语义），422-423, 450
multithreading（RPC多线程化），407-411
packet formats（RPC分组格式），444-449
premature termination of client（RPC客户过早终止），424-426
premature termination of server（RPC服务器过早终止），424-426
secure（安全RPC），417
server binding（RPC服务器捆绑），411-414
server duplicate request cache(RPC服务器重复请求高速缓存），421-424, 451, 532-533
TCP connection management（RPC TCP连接管理），420
timeout and retransmission（RPC超时与重传），417-422
transaction ID（RPC事务ID），420-422
RPC_CANTRECV constant（RPC_CANTRECV常值），424
RPC_MISMATCH constant（RPC_MISMATCH常值），448-449
RPC_SUCCESS constant（RPC_SUCCESS常值），409
rpc_msg structure, definition of（rpc_msg结构定义），446
rpcbind program（rpcbind程序），406, 411-412, 450
rpcgen program（rpcgen程序），400-406, 408-409, 411, 413-414, 419, 427-429, 432-433, 435, 439-440, 442, 449-451, 476, 486, 534
rpcinfo program（rpcinfo程序），412-414, 532
RPCL (remote procedure call language), 430
RPCSRC (remote procedure call source code), 406, 534
rpcvers member（rpcvers成员），446
rq_clntcred member（rq_clntcred成员），415
rq_cred member（rq_cred成员），415-416
rq_proc member（rq_proc），415
rq_prog member（rq_prog成员），415
rq_vers member（rq_vers成员），415
rq_xprt member（rq_xprt成员），415
rreply member（rreply成员），447
rsize member（rsize成员），357, 362-363, 367-368
RST (reset flag, TCP header), 425, 532
RTSIG_MAX constant（RTSIG_MAX常值），100
RW_MAGIC constant（RW_MAGIC常值），181
rw_condreaders member（rw_condreaders成员），183, 186
rw_condwriters member（rw_condwriters成员），184, 186
rw_magic member（rw_magic成员），181
rw_mutex member（rw_mutex成员），181, 183
rw_nwaitreaders member(rw_nwaitreaders成员），183, 191
rw_nwaitwriters member（rw_nwaitwriters成员），183-184, 190-191
rw_refcount member（rw_refcount成员），181, 183-184, 186
rwlock_cancelrdwait function（rwlock_cancelrdwait函数），191
rwlock_cancelwrwait function（rwlock_cancelwrwait函数），191

S_IRGRP constant（S_IRGRP常值），23
S_IROTH constant（S_IROTH常值），23
S_IRUSR constant（S_IRUSR常值），23
S_ISDOOR constant（S_ISDOOR常值），367
S_ISFIFO macro（S_ISFIFO宏），44
S_IWGRP constant（S_IWGRP常值），23
S_IWOTH constant（S_IWOTH常值），23
S_IWUSR constant（S_IWUSR常值），23
S_IXUSR constant（S_IXUSR常值），111, 263
S_TYPEISMQ macro（S_TYPEISMQ宏），21
S_TYPEISSEM macro（S_TYPEISSEM宏），21
S_TYPEISSHM macro（S_TYPEISSHM宏），21
SA_RESTART constant（SA_RESTART常值），106
SA_SIGINFO constant（SA_SIGINFO常值），100-102, 105-106, 127
sa_flags member（sa_flags成员），106
sa_handler member（sa_handler成员），106
sa_mask member（sa_mask成员），106
sa_sigaction member（sa_sigaction成员），105-106
Salus, P. H., 43, 536
sar program（sar程序），39
sbrk function（sbrk函数），533
_SC_CHILD_MAX constant（_SC_CHILD_MAX常值），297
scheduling, realtime（实时调度），14, 160, 171, 454
Schmidt, D. C., 180
_SC_MQ_OPEN_MAX constant（_SC_MQ_OPEN_MAX常值），87
_SC_MQ_PRIO_MAX constant（_SC_MQ_PRIO_MAX常值），87
scope, contention（竞用范围），386, 388, 462
_SC_OPEN_MAX constant（_SC_OPEN_MAX常值），72
_SC_PAGESIZE constant（_SC_PAGESIZE常值），317, 470, 529
_SC_RTSIG_MAX constant（_SC_RTSIG_MAX常值），102
_SC_SEM_NSEMS_MAX constant（_SC_SEM_NSEMS_MAX常值），257
_SC_SEM_VALUE_MAX constant（_SC_SEM_VALUE_MAX常值），257, 265
secure（安全的）

NFS（安全NFS），417
RPC（安全RPC），417
security, hole（安全漏洞），328
SEEK_CUR constant（SEEK_CUR常值），200, 217, 523
SEEK_END constant（SEEK_END常值），200, 217, 523
SEEK_SET constant（SEEK_SET常值），200, 217, 523
select function（select函数），74, 95, 98, 151-152, 155, 171,
323, 339, 454, 519-521, 528
Posix message queues with（带select函数的Posix消息队列），95-98
System V message queues with（带select函数的System V消息队列），151-152
Select wrapper function, source code（Select包裹函数源代码），521
sem structure（sem结构），273, 282-283
definition of（sem结构定义），282
SEM_A constant（SEM_A常值），33, 283
SEM_FAILED constant（SEM_FAILED常值），225
SEM_MAGIC constant（SEM_MAGIC常值），258, 262
SEM_NSEMS_MAX constant（SEM_NSEMS_MAX常值），257
Sem_post wrapper function, source code（Sem_post包裹函数源代码），11
SEM_R constant（SEM_R常值），33, 283
SEM_UNDO constant（SEM_UNDO常值），174, 286-287, 290, 294, 296, 492
SEM_VALUE_MAX constant（SEM_VALUE_MAX常值），225, 257
sem_base member（sem_base成员），282-283
sem_close function（sem_close函数），224-226, 228, 235, 260, 267, 275
definition of（sem_close函数定义），226
source code（sem_close函数源代码），261, 267, 275
sem_ctime member（sem_ctime成员），282-283, 289
sem_destroy function（sem_destroy函数），224, 238-242
definition of（sem_destroy函数定义），239
sem_flg member（sem_flg成员），276, 285-286, 492
sem_getvalue function（sem_getvalue函数），224-225, 227, 262, 269, 277
definition of（sem_getvalue函数定义），227
source code（sem_getvalue函数源代码），270, 278
sem_init function（sem_init函数），224, 238-242, 256, 315, 339, 490, 498
definition of（sem_init函数定义），239
sem_magic member（sem_magic成员），258, 262
sem_nsems member（sem_nsems成员），282-283, 290
sem_num member（sem_num成员），285-286
sem_op member（sem_op成员），285-287
sem_open function（sem_open函数），19, 22, 25-26, 224-226, 228-229, 232, 235, 239-240, 242, 256, 258, 260, 263, 265-267, 271, 273-274, 279, 285, 326-327, 333, 498, 524
definition of（sem_open函数定义），225
source code（sem_open函数源代码），258, 264, 271
sem_otime member（sem_otime成员），273-274, 282-285, 296
sem_perm structure（sem_perm结构），283, 288-289
definition of（sem_perm结构定义），282
sem_post function（sem_post函数），11, 90-91, 221-225, 227, 238, 242, 256-257, 260, 267, 275, 279, 287, 456, 490
definition of（sem_post函数定义），227
source code（sem_post函数定义），261, 268, 276
sem_t datatype（sem_t数据类型），8, 225, 238-240, 242, 256, 258, 260, 262-263, 265-266, 271, 275, 326
sem_trywait function（sem_trywait函数），224-227, 262, 269, 276, 339
definition of（sem_trywait函数定义），226
source code（sem_trywait函数源代码），270, 277
sem_unlink function（sem_unlink函数），224-226, 235, 242, 260, 267, 275, 305, 327, 333
definition of（sem_unlink函数定义），226
source code（sem_unlink函数源代码），261, 268, 276
sem_wait function（sem_wait函数），221-227, 230, 232, 236, 238, 242, 256, 258, 262, 268-269, 275-276, 279, 287, 339, 524-525
definition of（sem_wait函数定义），226
source code（sem_wait函数源代码），262, 269, 277
semadj member（semadj成员），10, 286-287, 294
semaem variable（semaem变量），37-38, 296
semaphore.h header（semaphore.h头文件），258, 262, 271
semaphores（信号量）
between processes, Posix（进程间共享Posix信号量），256-257
binary（二值信号量），219, 281
counting（计数信号量），221, 281
file locking using Posix（使用Posix信号量执行文件上锁），238
file locking using System V（使用System V信号量执行文件上锁），294-296
ID（信号量ID），271, 283, 290, 300
implementation using FIFOs, Posix（使用FIFO实现Posix信号量），257-262
implementation using memory-mapped I/O, Posix（使用内存映射I/O实现Posix信号量），262-270
implementation using System V semaphores, Posix（使用System V信号量实现Posix信号量），271-278
limits, Posix（Posix信号量限制），257
limits, System V（System V信号量限制），296-300
Posix（Posix信号量），219-279
System V（System V信号量），281-300
sembuf structure（sembuf结构），285-286, 290, 296
definition of（sembuf结构定义），285
semctl function（semctl函数），273-275, 277, 283-284, 287-290, 294
definition of（semctl函数定义），287
semget function（semget函数），34, 38, 257, 273-275, 282-285, 290, 294, 526
definition of（semget函数定义），282
semid_ds structure（semid_ds结构），282-284, 288-290
definition of（semid_ds结构定义），282
semmap variable（semmap变量），37
semmni variable（semmni变量），37-38, 296
semmns variable（semmns变量），37, 296
semmnu variable（semmnu变量），37, 296

semmsl variable（semmsl变量），37-38, 296
semncnt member（semncnt成员），282-283, 286-288
semop function（semop函数），273, 275-276, 283-287, 290, 294, 296, 492, 525-526
  definition of（semop函数定义），285
semopm variable（semopm变量），37-38, 296
sempid member（sempid成员），282-283, 288
semume variable（semume变量），37-38, 296
semun structure（semun变量），506
  definition of（semun变量定义），288
semval member（semval成员），282-283, 286-288
SEMVMX constant（SEMVMX常值），273
semvmx variable（semvmx变量），37-38, 296
semzcnt member（semzcnt成员），282-283, 286-288
sendmsg function（sendmsg函数），384
sendto function（sendto函数），405
seq member（seq成员），34-35, 38
sequence number, slot usage（槽位使用序列号），34
server（服务器）
  binding, RPC（RPC服务器捆绑），411-414
  concurrent（并发服务器），66-67, 147, 357, 372, 407
  creation procedure（服务器创建过程），384
  duplicate request cache, RPC（RPC服务器重复请求高速缓存），421-424, 451, 532-533
  iterative（迭代服务器），66-67, 144, 372, 407-408
  stub（服务器程序存根），405
server function（server函数），48-49, 54-55, 63, 72, 141-142, 144, 149
session（会话），4
set_concurrency function(set_concurrency函数)，163, 165, 488
SETALL constant（SETALL常值），283-284, 288, 290
setgid function（setgid函数），91
set-group-ID（SGID），26, 198, 205
setpgid function（setpgid函数），91
setrlimit function（setrlimit函数），72
setsid function（setsid函数），91
setsockopt function（setsockopt函数），418
setuid function（setuid函数），91
set-user-ID（SUID），26, 205, 369
SETVAL constant（SETVAL常值），273, 283-284, 288
setvbuf function（setvbuf函数），522
sh program（sh程序），52
Shar, D., 180, 536
shared memory（共享内存区），303-351
  ID（共享内存区ID），344, 351
  limits, System V（System V共享内存区限制），349-351
  object（共享内存区对象），325
  Posix（Posix共享内存区），325-342
  System V（System V共享内存区），343-351
shared-exclusive locking（共享-独占上锁），177
SHM_R constant（SHM_R常值），33
SHM_RDONLY constant（SHM_RDONLY常值），345
SHM_RND constant（SHM_RND常值），344
SHM_W constant（SHM_W常值），33
shm_atime member（shm_atime成员），343
shm_cnattch member（shm_cnattch成员），343
shm_cpid member（shm_cpid成员），343
shm_ctime member（shm_ctime成员），343, 345
shm_dtime member（shm_dtime成员），343
shm_lpid member（shm_lpid成员），343
shm_nattch member（shm_nattch成员），343, 348
shm_open function（shm_open函数），19, 22, 25, 308, 325-328, 330, 333-334, 337, 342-343
  definition of（shm_open函数定义），326
shm_perm structure（shm_perm结构），345
  definition of（shm_perm结构定义），343
shm_segsz member（shm_segsz成员），343
shm_unlink function（shm_unlink函数），326-327, 329, 333, 337, 342
  definition of（shm_unlink函数定义），326
shmat function（shmat函数），343-347, 351
  definition of（shmat函数定义），344
shmctl function（shmctl函数），345-348, 351
  definition of（shmctl函数定义），345
shmdt function（shmdt函数），345
  definition of（shmdt函数定义），345
shmget function（shmget函数），34, 38, 343-344, 346-349, 351
  definition of（shmget函数定义），344
shmid_ds structure（shmid_ds结构），345, 348
  definition of（shmid_ds结构定义），343
SHMLBA constant（SHMLBA常值），344
shmmax variable（shmmax变量），37-38, 349
shmmin variable（shmmin变量），37-38
shmmnb variable（shmmnb变量），349
shmmni variable（shmmni变量），37-38, 349
shmseg variable（shmseg变量），37-38, 349
short datatype, XDR（XDR short数据类型），427
SI_ASYNCIO constant（SI_ASYNCIO常值），101
SI_MESGQ constant（SI_MESGQ常值），101, 121
SI_QUEUE constant（SI_QUEUE常值），101, 104, 121
SI_TIMER constant（SI_TIMER常值），101
SI_USER constant（SI_USER常值），101
si_code member（si_code成员），101, 104, 121
si_signo member（si_signo成员），101
si_value member（si_value成员），101
SIG_DFL constant（SIG_DFL常值），106
SIG_IGN constant（SIG_IGN常值），60, 106
sigaction function（sigaction函数），91, 100, 105
sigaction structure, definition of（sigaction结构定义），106
sigaddset function（sigaddset函数），91
SIGALRM signal（SIGALRM信号），100, 106, 396-397, 425
SIGBUS signal（SIGBUS信号），320
SIGCHLD signal（SIGCHLD信号），48, 149, 391-393, 414
sigdelset function（sigdelset信号），91
sigemptyset function（sigemptyset函数），91
sigev structure（sigev结构），98
SIGEV_NONE constant（SIGEV_NONE常值），98
SIGEV_SIGNAL constant（SIGEV_SIGNAL常值），89, 98, 121
SIGEV_THREAD constant（SIGEV_THREAD常值），98, 128

sigev_notify member (sigev_notify成员), 88-89, 98
sigev_notify_attributes member (sigev_notify_attributes成员), 88, 98
sigev_notify_function member (sigev_notify_function成员), 88, 98
sigev_signo member (sigev_signo成员), 88, 90
sigev_value member (sigev_value成员), 88, 98
sigevent structure (sigevent结构), 87, 89, 91, 100, 119, 121
  definition of (sigevent结构定义), 88
sigfillset function (sigfillset函数), 91
Sigfunc_rt datatype (Sigfunc_rt数据类型), 105
siginfo_t structure (siginfo_t结构), 95, 101, 121
  definition of (siginfo_t结构定义), 101
SIGINT signal (SIGINT信号), 100
SIGIO signal (SIGIO信号), 256
sigismember function (sigismember函数), 91
SIGKILL signal (SIGKILL信号), 100
signal (信号)
  disposition (信号处置), 60, 502
  handler (信号处理程序), 60, 88-91, 93, 95, 98, 100-102, 105-106, 121, 149, 227, 256, 286, 391, 393, 456, 502, 520
  mask (信号掩码), 93, 95, 384, 502
  null (空信号), 121
  Posix realtime (Posix实时信号), 98-106
  synchronous (同步信号), 60
signal function (signal函数), 88, 90-91, 105
signal_rt function (signal_rt函数), 102, 105-106
  source code (signal_rt函数源代码), 105
sigpause function (sigpause函数), 91
sigpending function (sigpending函数), 91
SIGPIPE signal (SIGPIPE信号), 59-60, 519
sigprocmask function(sigprocmask函数), 91, 93, 95, 102
sigqueue function (sigqueue函数), 91, 101, 121
SIGRTMAX signal (SIGRTMAX信号), 100, 102, 127
SIGRTMIN signal (SIGRTMIN信号), 100, 127
SIGSEGV signal (SIGSEGV信号), 174, 267, 309, 318-320, 526
sigset function (sigset函数), 91
sigsuspend function (sigsuspend函数), 91, 93
SIGTERM signal (SIGTERM信号), 469
sigtimedwait function (sigtimedwait函数), 95
SIGUSR1 signal (SIGUSR1信号), 88-91, 93, 95
sigval structure (sigval结构), 100-101
  definition of (sigval结构定义), 88
sigwait function (sigwait函数), 93-95
  definition of (sigwait函数定义), 95
sigwaitinfo function (sigwaitinfo函数), 95
silver bullet (简单判定), 453
Simple Mail Transfer Protocol (简单的邮件传送协议), 见SMTP
Single Unix Specification (单一Unix规范), 15
sival_int member (sival_int成员), 88, 102
sival_ptr member (sival_ptr成员), 88
sleep function (sleep函数), 91, 93, 127, 190, 215, 296, 398, 425, 530
sleep_us function (sleep_us函数), 339
slot usage sequence number (槽位使用序列号), 34
Smaalders, B., 180, 536
SMTP (Simple Mail Transfer Protocol), 67
snprintf function (snprintf函数), 21
socket, Unix domain (Unix域套接字), 84, 341, 379-380, 384, 456, 459
socket function (socket函数), 399
socketpair function (socketpair函数), 44, 50
sockets API (套接字API), 8, 14, 151, 398-399, 403, 406, 449-450, 454-455
Solaris, 15, 20-21, 29, 37, 51, 53, 59, 73, 77-78, 82, 98, 100, 104, 109, 154, 163, 165, 209-210, 213, 225, 232, 238, 322, 331, 333, 342, 348, 356-357, 362, 367, 370, 384, 398, 403-405, 408, 411-413, 424-425, 427, 454, 458-460, 462-463, 465, 471, 475, 482, 488, 509-510, 517, 520-524
solutions to exercises (习题解答), 515-534
source code (源代码)
  conventions (源代码约定), 11
Spafford, E. H., 417, 535
Spec 1170,15
spinning (轮转), 167
sprintf function (sprintf函数), 21
spurious wakeup (虚假唤醒), 121, 170
squareproc_1 function (squareproc_1函数), 402-403, 405, 419, 424
Srinivasan, R., 406, 412, 426, 536
st_dev member (st_dev成员), 28-30
st_gid member (st_gid成员), 328
st_ino member (st_ino成员), 28-30
st_mode member (st_mode成员), 21, 44, 115, 267, 328, 367
st_size member (st_size成员), 74, 262, 328
st_uid member (st_uid成员), 328
Staelin, C., 458, 536
Stallman, R. M., 13
stamp member (stamp成员), 446
standards, Unix (Unix标准), 13-15
start_time function (start_time函数), 469-470
  source code (start_time函数源代码), 470
stat function (stat函数), 21, 28-29, 44, 91, 115, 262, 267, 455, 517
stat member (stat成员), 449
stat structure (stat), 21, 28-29, 44, 74, 115, 262, 267, 328, 367
  definition of (stat结构定义), 328
statd program (statd程序), 216
Stevens, W. R., 536-537
stop_time function (stop_time函数), 469-470
  source code (stop_time函数源代码), 470
strchr function (stop_time函数), 63
streams versus messages (对比流与消息), 67-72
strerror function (strerror函数), 49, 511
string datatype, XDR (XDR string数据类型), 429, 438, 451
strlen function (strlen函数), 429

struct datatype, XDR（XDR struct数据类型），429
stub（存根）
  client（客户程序存根），403, 405
  server（服务器程序存根），405
SUCCESS constant（SUCCESS常值），447-448
Sun Microsystems, 406
SunOS 4, 316
superuser（超级用户），25, 33-34, 216, 369-370, 414, 417
supplementary group ID（辅助组ID），25, 414, 416
svc_create function（svc_create函数），411
svc_dg_enablecache function（svc_dg_enablecache函数），422
  definition of（svc_dg_enablecache函数定义），422
svc_reg function（svc_reg函数），414
svc_req structure（svc_reg结构），409, 415, 422
  definition of（svc_reg结构定义），415
svc_run function（svc_run函数），414
svc_tli_create function（svc_tli_create函数），414
SVCXPRT structure（SVCXPRT结构），415
SVMSG_MODE constant（SVMSG_MODE常值），35
  definition of（SVMSG_MODE常值定义），508
svmsg.h header（svmsg.h头文件），140, 144
SVR2 (System V Release 2), 198
SVR3 (System V Release 3), 98, 198, 205
SVR4 (System V Release 4), 34, 44, 50-51, 84, 152, 311, 315-317, 322, 359, 379, 384, 456
SVSEM_MODE constant（SVSEM_MODE常值），274
  definition of（SVSEM_MODE常值定义），508
SVSHM_MODE constant, definition of（SVSHM_MODE常值定义），508
SYN (synchronize sequence numbers flag, TCP header), 532
synchronization（同步）
explicit（显式同步），161
implicit（隐式同步），161
synchronize sequence numbers flag, TCP header（TCP首部同步序列号标志），见SYN
synchronous（同步的）
  procedure call（同步过程调用），356-357, 476
  signal（同步信号），60
sysconf function（sysconf函数），72-73, 86, 91, 100, 102, 257, 265, 318, 520
sysconfig program（sysconfig程序），37, 458
sysconfigdb program（sysconfigdb程序），38
sysdef program（sysdef程序），37
<sys/errno.h> header（<sys/errno.h>头文件），13, 503
<sys/ipc.h> header（<sys/ipc.h>头文件），30
syslog function（syslog函数），336, 408, 511
<sys/msg.h> header（<sys/msg.h>头文件），33, 129, 131, 134
<sys/sem.h> header（<sys/sem.h>头文件），33, 282, 288
<sys/shm.h> header（<sys/shm.h>头文件），33, 343
<sys/stat.h> header（<sys/stat.h>头文件），23, 54
system call（系统调用），5, 198, 205, 220, 303, 361, 391, 405, 482
  interrupted（被中断的系统调用），121, 124, 132-133, 149, 151, 227, 279, 286, 391-392, 395, 521, 524-525
  slow（慢系统调用），286
system function（system函数），134
System V
  IPC（System V IPC），27-39
  IPC identifier reuse（System V IPC标识符重用），34-36
  IPC kernel limits（System V IPC内核限制），36-38
  IPC permissions（System V IPC权限），31-35, 39, 130-131, 282-283, 343-345
  message queue limits（System V消息队列限制），152-154
  message queues（System V消息队列），129-155
  message queues with poll function（带poll函数的System V消息队列），151-152
  message queues with select function（带select函数的System V消息队列），151-152
  Release 2（System V Release 2），见SVR2
  Release 3（System V Release 3），见SVR3
  Release 4（System V Release 4），见SVR4
  semaphore limits（System V信号量限制），296-300
  semaphores（System V信号量），281-300
  semaphores, file locking using（使用System V信号量进行文件上锁），294-296
  semaphores, used for implementation of Posix semaphores（System V信号量用于实现Posix信号量），271-278
  shared memory（System V共享内存区），343-351
  shared memory limits（System V共享内存区限制），349-351
SYSTEM_ERR constant（SYSTEM_ERR常值），447-448
<sys/types.h> header（<sys/types.h>），28

tar program（tar程序），13
tcdrain function（tcdrain函数），91
tcflow function（tcflow函数），91
tcflush function（tcflush函数），91
tcgetattr function（tcgetattr函数），91
tcgetpgrp function（tcgetpgrp函数），91
TCP (Transmission Control Protocol), 67, 74, 401, 404-407, 411-412, 418-426, 444-446, 450-451, 454, 459, 476, 532-533
  connection management, RPC（RPC TCP连接管理），420
  for Transactions（事务目的TCP），见T/TCP
  three-way handshake（TCP三路握手），420
tcpdump program（tcpdump程序），420, 424-425, 533
TCPv1 (TCP/IP Illustrated, Volume 1), 536
TCPv2 (TCP/IP Illustrated, Volume 2), 537
TCPv3 (TCP/IP Illustrated, Volume 3), 537
tcsendbreak function（tcsendbreak函数），91
tcsetattr function（tcsetattr函数），91
tcsetpgrp function（tcsetpgrp函数），91
Telnet (remote terminal protocol), 336, 399
termination of client（客户终止）
  doors, premature（门客户过早制止），390-397
  RPC, premature（RPC客户过早制止），424-426
termination of server（服务器终止）
  doors, premature（门服务器过早终止），390-397
  RPC, premature（RPC服务器过早终止），424-426
thr_setconcurrency function（thr_setconcurrency函

数），163
threads（线程），5-6, 501-504
attributes（线程属性），98, 113, 502, 521, 532
cancellation（线程取消），174, 180, 183, 187-192, 384, 388, 396-398, 530
concurrency（线程并发度），163, 165-166, 488
detached（已脱离线程），98, 384, 386-388, 504
ID（线程ID），502
ID, printing（显示线程ID），371
implementation, many-to-few（多对少线程实现），163
implementation, M-to-N（M对N线程实现），163
implementation, two-level（二级线程实现），163
joinable（可联接线程），387, 504
main（主线程），93, 190, 235, 388, 488, 490, 502
management, doors（门线程管理），370-375
priority（线程优先级），160, 502
start function（线程启动函数），98, 187, 386-387, 502
termination, explicit（显式线程终止），502
termination, implicit（隐式线程终止），502
three-way handshake, TCP（TCP三路握手），420
time（时间）
absolute（绝对时间），171
delta（相对时间），171
round-trip（往返时间），451, 458
time function（time函数），91
timeout（超时），67, 171, 424, 426
and retransmission, RPC（RPC超时与重传），417-422
TIMEOUT constant（TIMEOUT常值），420
timer_getoverrun function（timer_getoverrun函数），91
timer_gettime function（timer_gettime函数），91
timer_settime function（timer_settime函数），91, 101
times function（times函数），91
timespec structure（timespec函数），171, 508
definition of（timespec函数定义），171
timeval structure（timeval结构），418-419, 471, 534
TI-RPC (transport independent RPC), 406-407, 411, 421, 446, 533
TLI (Transport Layer Interface), API(传输层接口API), 406
touch function（touch函数），467, 470
source code（touch函数源代码），470
transaction ID（事务ID），见XID
Transmission Control Protocol（传输控制协议），见TCP
TRUE constant（TRUE常值），409, 418, 429, 435, 439, 441, 444
T/TCP (TCP for Transactions), 537
tv_nsec member（tv_nsec成员），171, 508
tv_sec member（tv_sec成员），171, 508
tv_sub function（tv_sub函数），471
source code（tv_sub函数源代码），471
two-level thread implementation（二级线程实现），163
typedef datatype, XDR（XDR typedef数据类型），427
typing（类型指定）
explicit（显式类型指定），426
implicit（隐式类型指定），426
UDP (User Datagram Protocol), 68, 74, 83, 246, 341, 401, 405-407, 411-414, 418-425, 445-447, 450-452, 454-455, 459, 476, 532-534
uid member(uid成员), 33-34, 131, 134, 283, 288, 345, 446
uint8_t datatype（uint8_t数据类型），509
ulimit program（ulimit程序），72-73
umask function（umask函数），23, 91
umask program（umask程序），23, 39
un_lock function, definition of（un_lock函数定义），202
uname function（uname函数），91
uniform resource locator（统一资源定位符），见URL
union datatype, XDR（XDR union数据类型），429
<unistd.h> header（<unistd.h>头文件），8, 86, 173, 257
Unix
95（Unix 95），15
98（Unix 98），8, 16, 33-34, 36, 44, 84, 90, 129, 159, 163, 173, 178, 192, 205, 282, 284, 288, 364, 454, 468, 482, 488, 526, 536
98, definition of（Unix 98定义），15
authentication（Unix认证），414
Columbus（Columbus Unix），28
domain socket（Unix域套接字），84, 341, 379-380, 384, 456, 459
Specification, Single（单一Unix规范），15
standards（Unix标准），13-15
System III（System III Unix），43, 198
Version 7（Version 7 Unix），98, 198
versions and portability（Unix版本与兼容性），15
unlink function（unlink函数），56, 58, 77, 91, 115, 117, 214-216, 226, 260, 267, 275, 327, 342, 359, 376
unpipc.h header(unpipc.h头文件), 21, 55, 105, 111, 274, 288, 505-509
source code（unpipc.h头文件源代码），505
UNPv1 (UNIX Network Programming, Volume 1), 537
unsigned char datatype, XDR（XDR unsigned char数据类型），427
unsigned hyper datatype, XDR（XDR unsigned hyper数据类型），427
unsigned int datatype, XDR（XDR unsigned int数据类型），427
unsigned long datatype, XDR（XDR unsigned long数据类型），427
unsigned short datatype, XDR（XDR unsigned short数据类型），427
URL (uniform resource locator), 535
User Datagram Protocol（用户数据报协议），见UDP
user ID（用户ID），328, 397, 413, 417, 502
effective(有效用户ID), 23, 25, 33-34, 84, 131, 283, 365, 369-370, 414, 416, 515
real（实际用户ID），365, 369
UTC (Coordinated Universal Time), 171
utime function（utime函数），91
UUCP, 198

va_arg function（va_arg函数），111, 260
va_mode_t datatype（va_mode_t数据类型），111, 260, 263,

273
definition of (va_mode_t数据类型定义), 508
va_start function (va_start函数), 260
Vahalia, U., 311, 537
val member (val成员), 288
valloc function (valloc函数), 467-468
verf member (verf成员), 446-447
verifier (验证器), 417, 446, 449, 533
vers member (vers成员), 446
vi program (vi程序), 13
void datatype (void数据类型), 503-504

wait function (wait函数), 91, 413-414
waiting, locking versus (对比上锁与等待), 165-167
waitpid function (waitpid函数), 48, 73, 91, 149, 503
wakeup, spurious (虚假唤醒), 121, 170
wc program (wc程序), 161
well-known (众所周知的)
key (众所周知的键), 147
pathname (众所周知的路径名), 60, 215
White, J. E., 406, 537
wrapper function (包裹函数), 11-13
source code, Pthread_mutex_lock (Pthread_mutex_lock包裹函数源代码), 12
source code, Select (Select包裹函数源代码), 521
source code, Sem_post (Sem_post包裹函数源代码), 11
Wright, G. R., 537
write function (write函数), 5, 43, 52, 54, 59-60, 65, 83, 90-91, 95, 98, 142, 161, 200, 204-205, 207, 249, 260, 263, 278, 304, 310-311, 315, 317, 322, 327, 399, 405, 435, 451, 456-457, 467, 469, 471, 482, 515, 519, 522-526, 528
write_lock function, definition of (write_lock函数定义), 202
writew_lock function (writew_lock函数), 495
definition of (writew_lock函数定义), 202

XDR (external data representation), 403, 406, 426-444, 450, 532-534
datatypes (XDR数据类型), 427-430
fragment (XDR片段), 444
XDR datatype (XDR数据类型), 432
XDR_DECODE constant (XDR_DECODE常值), 435
XDR_ENCODE constant (XDR_ENCODE常值), 432, 435
xdr_data function (xdr_data函数), 432, 435, 532
xdr_free function (xdr_free函数), 410, 435, 452
xdr_getpos function (xdr_getpos函数), 435
xdr_string function (xdr_string函数), 435, 532
xdr_void function (xdr_void函数), 534
xdrmem_create function (xdrmem_create函数), 432, 435, 451-452
Xenix, 198
Xerox, 406
XID (transaction ID), 420-422, 532-533
xid member (xid成员), 446
X/Open, 14, 198
Portablity Guide (X/Open可移植性指南), 见XPG
Transport Interface (X/Open传输接口), 见XTI
_XOPEN_REALTIME constant (_XOPEN_REALTIME常值), 9
XPG (X/Open Portablity Guide), 15, 198, 284, 468
XTI (X/Open Transport Interface), API, 14, 151, 398-399, 403, 406, 413-414, 424, 449-450, 455

yacc program (yacc程序), 13

zombie (僵尸), 48, 14

函数原型索引表

（下面所列页码均指页边栏中标注的页码）

函数原型	页码
bool_t **clnt_control**(CLIENT *cl, unsigned int *request*, char *ptr);	418
CLIENT *clnt_create(const char *host, unsigned long *prognum*, unsigned long *versnum*, const char *protocol);	401
void **clnt_destroy**(CLIENT *cl);	420
int **door_bind**(int *fd*);	390
int **door_call**(int *fd*, door_arg_t *argp);	361
int **door_create**(Door_server_proc *proc, void *cookie, u_int *attr*);	363
int **door_cred**(door_cred_t *cred);	365
int **door_info**(int *fd*, door_info_t *info);	365
int **door_return**(char *dataptr, size_t *datasize*, door_desc_t *descptr, size_t *ndesc*);	365
int **door_revoke**(int *fd*);	390
Door_create_proc *door_server_create(Door_create_proc *proc);	384
int **door_unbind**(void);	390
void **err_dump**(const char *fmt, ...);	512
void **err_msg**(const char *fmt, ...);	512
void **err_quit**(const char *fmt, ...);	512
void **err_ret**(const char *fmt, ...);	511
void **err_sys**(const char *fmt, ...);	511
int **fcntl**(int *fd*, int *cmd*, ... /* struct flock *arg */);	199
int **fstat**(int *fd*, struct stat *buf);	328
key_t **ftok**(const char *pathname, int *id*);	28
int **ftruncate**(int *fd*, off_t *length*);	327
int **mkfifo**(const char *pathname, mode_t *mode*);	54
void *mmap(void *addr, size_t *len*, int *prot*, int *flags*, int *fd*, off_t *offset*);	308
int **msync**(void *addr, size_t *len*, int *flags*);	310
int **munmap**(void *addr, size_t *len*);	309
int **mq_close**(mqd_t *mqdes*);	77
int **mq_getattr**(mqd_t *mqdes*, struct mq_attr *attr);	79
int **mq_notify**(mqd_t *mqdes*, const struct sigevent *notification*);	87
mqd_t **mq_open**(const char *name, int *oflag*, ... /* mode_t *mode*, struct mq_attr *attr */);	76
ssize_t **mq_receive**(mqd_t *mqdes*, char *ptr, size_t *len*, unsigned int *priop*);	83
int **mq_send**(mqd_t *mqdes*, const char *ptr, size_t *len*, unsigned int *prio*);	83
int **mq_setattr**(mqd_t *mqdes*, const struct mq_attr *attr, struct mq_attr *oattr*);	79
int **mq_unlink**(const char *name);	77
int **msgctl**(int *msqid*, int *cmd*, struct msqid_ds *buff*);	134
int **msgget**(key_t *key*, int *oflag*);	130
ssize_t **msgrcv**(int *msqid*, void *ptr, size_t *length*, long *type*, int *flag*);	132
int **msgsnd**(int *msqid*, const void *ptr, size_t *length*, int *flag*);	131
int **pclose**(FILE *stream);	52
int **pipe**(int *fd[2]*);	44
FILE *popen(const char *command, const char *type);	52

（续）

函数原型	页码
int **pthread_cancel**(pthread_t *tid*);	187
void **pthread_cleanup_pop**(int *execute*);	187
void **pthread_cleanup_push**(void (**function*)(void *), void **arg*);	187
int **pthread_create**(pthread_t **tid*, const pthread_attr_t **attr*, void *(**func*)(void *), void **arg*);	502
int **pthread_detach**(pthread_t *tid*);	504
void **pthread_exit**(void **status*);	504
int **pthread_join**(pthread_t *tid*, void ***status*);	503
pthread_t **pthread_self**(void);	503
int **pthread_condattr_destroy**(pthread_condattr_t **attr*);	172
int **pthread_condattr_getpshared**(const pthread_condattr_t **attr*, int **valptr*);	173
int **pthread_condattr_init**(pthread_condattr_t **attr*);	172
int **pthread_condattr_setpshared**(pthread_condattr_t **attr*, int *value*);	173
int **pthread_cond_broadcast**(pthread_cond_t **cptr*);	171
int **pthread_cond_destroy**(pthread_cond_t **cptr*);	172
int **pthread_cond_init**(pthread_cond_t **cptr*, const pthread_condattr_t **attr*);	172
int **pthread_cond_signal**(pthread_cond_t **cptr*);	167
int **pthread_cond_timedwait**(pthread_cond_t **cptr*, pthread_mutex_t **mptr*, const struct timespec **abstime*);	171
int **pthread_cond_wait**(pthread_cond_t **cptr*, pthread_mutex_t **mptr*);	167
int **pthread_mutexattr_destroy**(pthread_mutexattr_t **attr*);	172
int **pthread_mutexattr_getpshared**(const pthread_mutexattr_t **attr*, int **valptr*);	173
int **pthread_mutexattr_init**(pthread_mutexattr_t **attr*);	172
int **pthread_mutexattr_setpshared**(pthread_mutexattr_t **attr*, int *value*);	173
int **pthread_mutex_destroy**(pthread_mutex_t **mptr*);	172
int **pthread_mutex_init**(pthread_mutex_t **mptr*, const pthread_mutexattr_t **attr*);	172
int **pthread_mutex_lock**(pthread_mutex_t **mptr*);	160
int **pthread_mutex_trylock**(pthread_mutex_t **mptr*);	160
int **pthread_mutex_unlock**(pthread_mutex_t **mptr*);	160
int **pthread_rwlockattr_destroy**(pthread_rwlockattr_t **attr*);	179
int **pthread_rwlockattr_getpshared**(const pthread_rwlockattr_t **attr*, int **valptr*);	179
int **pthread_rwlockattr_init**(pthread_rwlockattr_t **attr*);	179
int **pthread_rwlockattr_setpshared**(pthread_rwlockattr_t **attr*, int *value*);	179
int **pthread_rwlock_destroy**(pthread_rwlock_t **rwptr*);	179
int **pthread_rwlock_init**(pthread_rwlock_t **rwptr*, const pthread_rwlockattr_t **attr*);	179
int **pthread_rwlock_rdlock**(pthread_rwlock_t **rwptr*);	178
int **pthread_rwlock_tryrdlock**(pthread_rwlock_t **rwptr*);	178
int **pthread_rwlock_trywrlock**(pthread_rwlock_t **rwptr*);	178
int **pthread_rwlock_unlock**(pthread_rwlock_t **rwptr*);	178
int **pthread_rwlock_wrlock**(pthread_rwlock_t **rwptr*);	178

（续）

函数原型	页码
long **pr_thread_id**(pthread_t *ptr);	371
char ***px_ipc_name**(const char *name);	21
int **sem_close**(sem_t *sem);	226
int **sem_destroy**(sem_t *sem);	239
int **sem_getvalue**(sem_t *sem, int *valp);	227
int **sem_init**(sem_t *sem, int shared, unsigned int value);	239
sem_t ***sem_open**(const char *name, int oflag, ... /* mode_t mode, unsigned int value */);	225
int **sem_post**(sem_t *sem);	227
int **sem_trywait**(sem_t *sem);	226
int **sem_unlink**(const char *name);	226
int **sem_wait**(sem_t *sem);	226
int **semctl**(int semid, int semnum, int cmd, ... /* union semun arg */);	287
int **semget**(key_t key, int nsems, int oflag);	282
int **semop**(int semid, struct sembuf *opsptr, size_t nops);	285
int **shm_open**(const char *name, int oflag, mode_t mode);	326
int **shm_unlink**(const char *name);	326
void ***shmat**(int shmid, const void *shmaddr, int flag);	344
int **shmctl**(int shmid, int cmd, struct shmid_ds *buff);	345
int **shmdt**(const void *shmaddr);	345
int **shmget**(key_t key, size_t size, int oflag);	344
Sigfunc_rt ***signal_rt**(int signo, Sigfunc_rt *func);	105
int **sigwait**(const sigset_t *set, int *sig);	95
int **start_time**(void);	470
double **stop_time**(void);	470
int **svc_dg_enablecache**(SVCXPRT *xprt, unsigned long size);	422
int **touch**(void *vptr, int nbytes);	470
void **tv_sub**(struct timeval *out, struct timeval *in);	471

结构定义索引表

accepted_reply	447	opaque_auth	416
authsys_parms	416		
		rejected_reply	449
call_body	446	reply_body	447
		rpc_msg	446
d_desc	380		
door_arg_t	362	sem	282
door_cred_t	365	sembuf	285
door_desc_t	380	semid_ds	282
door_info_t	366	sem_perm	282
		semun	288
flock	199	shmid_ds	343
		shm_perm	343
ipc_perm	30	sigaction	106
		sigevent	88
mismatch_info	447	siginfo_t	101
mq_attr	80	sigval	88
msgbuf	131	stat	328
msg_perm	129	svc_req	415
msqid_ds	129		
		timespec	171

TURING 图灵教育 站在巨人的肩上

读者积分赠书卡

手机号码：_____（此为会员编号）

姓　　名：_____　性别：□男 □女　出生年月：____年__月

通信地址：_____

邮政编码：_____

电子邮件：_____

您购买的图书是：23028 /《UNIX网络编程　卷2：进程间通信（第2版）》（89.00元）

您获得的会员积分是：8.9 分

欢迎参加“**有奖DEBUG**”活动。提交本书勘误，每确认一处即可获赠积分5分。详情见图灵网站。

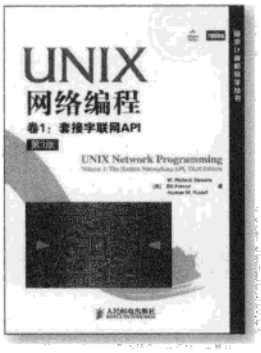
请沿虚线剪下此页，寄回图灵公司，即可成为图灵读者俱乐部的一员（复印无效）。**积分累计，可获赠书**（赠书清单见图灵网站）。

邮政编码：100107

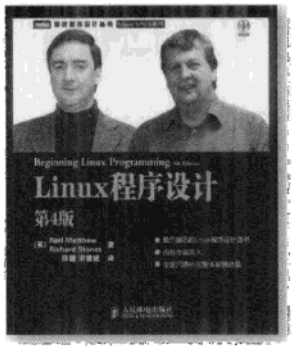
通信地址：北京市朝阳区北苑路 13 号院 1 号楼 C603

北京图灵文化发展有限公司　图灵读者俱乐部

图灵六月重点图书



书名：UNIX网络编程 卷1：套接字联网API（第3版）
书号：978-7-115-22840-6
- 世界顶级网络专家传世之作
- 网络编程权威参考



书名：Linux程序设计(第4版)
书号：978-7-115-22821-5
- 最负盛名的Linux程序设计图书
- 内容全面深入
- 全球开源社区集体智慧结晶



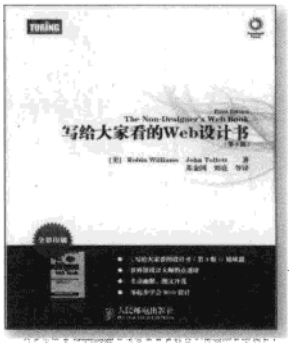
书名：深入Linux内核架构
书号：978-7-115-22743-0
- 内容全面深入
- 全球开源社区集体智慧结晶
- 领略Linuix内核的绝美风光



书名：精通Linux设备驱动程序开发
书号：978-7-115-22167-4
- 全面深入的Linux设备驱程序名著
- 世界级Linux技术大师力作
- 大量底层技术内幕首次公开



书名：Oracle优化日记：一个金牌DBA的故事
书号：978-7-115-23071-3
- 金牌DBA精彩纷呈的经历
- 86个优化技巧活学活用
- 培养多种能力，保障职场成功



书名：写给大家看的Web设计书（第3版）
书号：978-7-115-22536-8
- 《写给大家看的设计书（第3版）》姐妹篇
- 世界级设计大师指点迷津
- 生动幽默，图文并茂
- 零起步学会Web设计

[General Information]
书名=UNIX网络编程　卷2　进程间通信　第2版
作者=（美）史蒂文斯著
页数=460
出版社=人民邮电出版社
出版日期=2010.07
SS号=12636556
DX号=000006916353
URL=http://book.szdnet.org.cn/bookDetail.jsp?dxNumber=00006916353&d=6B42105D6DF81C5808C30F5F19FF9981D

封面
书名
版权
前言
目录
第一部分　简介
第1章　简介
1.1　概述
1.2　进程、线程与信息共享
1.3　IPC对象的持续性
1.4　名字空间
1.5　fork、exec和exit对IPC对象的影响
1.6　出错处理：包裹函数
1.7　Unix标准
1.8　书中IPC例子索引表
1.9　小结
习题
第2章　Posix IPC
2.1　概述
2.2　IPC名字
2.3　创建与打开IPC通道
2.4　IPC权限
2.5　小结
习题
第3章　System V IPC
3.1　概述
3.2　key_t键和ftok函数
3.3　ipc_perm结构
3.4　创建与打开IPC通道
3.5　IPC权限
3.6　标识符重用
3.7　ipcs和ipcrm程序
3.8　内核限制
3.9　小结
习题
第二部分　消息传递
第4章　管道和FIFO
4.1　概述
4.2　一个简单的客户-服务器例子
4.3　管道
4.4　全双工管道
4.5　popen和pclose函数
4.6　FIFO
4.7　管道和FIFO的额外属性
4.8　单个服务器，多个客户
4.9　对比迭代服务器与并发服务器
4.10　字节流与消息
4.11　管道和FIFO限制
4.12　小结
习题
第5章　Posix消息队列
5.1　概述
5.2　mq_open、mq_close和mq_unlink函数
5.3　mq_getattr和mq_setattr函数
5.4　mq_send和mq_receive函数
5.5　消息队列限制
5.6　mq_notify函数
5.7　Posix实时信号
5.8　使用内存映射I/O实现Posix消息队列
5.9　小结
习题
第6章　System V消息队列
6.1　概述
6.2　msgget函数
6.3　msgsnd函数
6.4　msgrcv函数
6.5　msgctl函数
6.6　简单的程序
6.7　客户-服务器例子
6.8　复用消息
6.9　消息队列上使用select和poll
6.10　消息队列限制
6.11　小结
习题

第三部分　同步
第7章　互斥锁和条件变量
7.1　概述
7.2　互斥锁：上锁与解锁
7.3　生产者-消费者问题
7.4　对比上锁与等待
7.5　条件变量：等待与信号发送
7.6　条件变量：定时等待和广播
7.7　互斥锁和条件变量的属性
7.8　小结
习题
第8章　读写锁
8.1　概述
8.2　获取与释放读写锁
8.3　读写锁属性
8.4　使用互斥锁和条件变量实现读写锁
8.5　线程取消
8.6　小结
习题
第9章　记录上锁
9.1　概述
9.2　对比记录上锁与文件上锁
9.3　Posix　fcntl记录上锁
9.4　劝告性上锁
9.5　强制性上锁
9.6　读出者和写入者的优先级
9.7　启动一个守护进程的唯一副本
9.8　文件作锁用
9.9　NFS上锁
9.10　小结
习题
第10章　Posix信号量
10.1　概述
10.2　sem_open、sem_close和sem_unlink函数
10.3　sem_wait和sem_trywait函数
10.4　sem_post和sem_getvalue函数
10.5　简单的程序
10.6　生产者-消费者问题
10.7　文件上锁
10.8　sem_init和sem_destroy函数
10.9　多个生产者，单个消费者
10.10　多个生产者，多个消费者
10.11　多个缓冲区
10.12　进程间共享信号量
10.13　信号量限制
10.14　使用FIFO实现信号量
10.15　使用内存映射I/O实现信号量
10.16　使用System　V信号量实现Posix信号量
10.17　小结
习题
第11章　System　V信号量
11.1　概述
11.2　semget函数
11.3　semop函数
11.4　semctl函数
11.5　简单的程序
11.6　文件上锁
11.7　信号量限制
11.8　小结
习题
第四部分　共享内存区
第12章　共享内存区介绍
12.1　概述
12.2　mmap、munmap和msync函数
12.3　在内存映射文件中给计数器持续加1
12.4　4.4BSD匿名内存映射
12.5　SVR4/dev/zero内存映射
12.6　访问内存映射的对象
12.7　小结
习题
第13章　Posix共享内存区
13.1　概述
13.2　shm_open和shm_unlink函数
13.3　ftruncate和fstat函数

13.4 简单的程序
13.5 给一个共享的计数器持续加1
13.6 向一个服务器发送消息
13.7 小结
习题
第14章 System V共享内存区
14.1 概述
14.2 shmget函数
14.3 shmat函数
14.4 shmdt函数
14.5 shmctl函数
14.6 简单的程序
14.7 共享内存区限制
14.8 小结
习题
第五部分 远程过程调用
第15章 门
15.1 概述
15.2 door_call函数
15.3 door_create函数
15.4 door_return函数
15.5 door_cred函数
15.6 door_info函数
15.7 例子
15.8 描述符传递
15.9 door_sever_create函数
15.10 door_bind、door_unbind和door_revoke函数
15.11 客户或服务器的过早终止
15.12 小结
习题
第16章 Sun RPC
16.1 概述
16.2 多线程化
16.3 服务器捆绑
16.4 认证
16.5 超时和重传
16.6 调用语义
16.7 客户或服务器的过早终止
16.8 XDR：外部数据表示
16.9 RPC分组格式
16.10 小结
习题
后记
附录A 性能测量
附录B 线程入门
附录C 杂凑的源代码
附录D 精选习题解答
参考文献
索引