第 6 章 进程间通信

6.l 简 介

复杂的编程环境通常使用多个相关的进程来执行有关操作。进程互相间必须进行通信，来共享资源和信息。因此要求内核提供必要的机制，这些机制通常被叫做进程间通信，或 IPC。本章我们阐述在大多数种类的 UNIX 中的 IPC 方法。

进程间通信有如下一些目的：

数据传输 进程可能要发送数据到另一个进程。发送的数据量可以在一个字节到几兆字节之间。

并享数据 多个进程想要操作共享数据。一个进程修改了数据，其他共享该数据的进程应该立即看见这个变化。

通知事件 当一些事件发生时，进程也许会向另一个进程或一组进程发消息通知事件的发生。比如进程终止时，它要通知它的父进程。接收者可能是被异步通知的，这时候它的正常处理被中断。由此，接收者可以选择等待通知。

资涯共享 尽管内核为资源分配指定了缺省的语义，但是这些语义并不适合所有的应用。一些有相互操作的进程可能要自行定义一些协议针对它们要访问的特定的资源。这些协议通过使用锁和同步机制来实现的，而锁和同步机制是建立在内核提供的基本原语之上的。

进程控制 有些进程，比如 debugger 希望完全控制另一个进程（目标进程）的执行。控制进程希望能够拦截为目标进程设计的所有的陷入和异常，并且能够及时知道目标进程的状态改变。

UNIX 提供了许多种不同的 IPC 机制。本章首先描述出现在所有版本的 UNIX 中的内核 IPC 集合，也就是信号，管道，和进程跟踪。接下来详细阐述 System 5的 IPC 的原语。最后，我们介绍一下 Mach 中的基于消息的 IPC 机制，它提供了丰富的方法，而不是单独的，统一的框架结构。

6．2 通用 IPC 方法

UNIX 系统最先向外界发行时，它提供了三种进程间通信的方法——信号、管道和进程跟踪①。它们是各种 UNIX 共有的 IPC 机制。信号和管道在本书的其他部分有非常详尽的描述，本章中，我们讨论系统是怎样用它们来完成进程间通信的。

6.2.1 信号

信号主要用来通知进程异步事件的发生。最初信号设计的目的是为了处理错误，它们也用来作为最基本的 IPC 机制。现在的 UNIX 版本可以识别 31 或更多种不同的信号。这些信号中的大部分都有了预先定义好的意义，但是至少有两个，SIGUSR1 和 SIGUSR2 可以由应用程序来定义。进程可以显式地用 kill 或是 killpg 系统调用来向另一个进程或进程组发信号。此外，内核可以响应不同的事件而产生内部信号。例如，当在终端键入 Ctrl＋C 时内核便发送一个 SIGINT 信号到前台的进程。

每一个信号有一个缺省的动作，典型的是终止进程。进程可以通过提供信号处理函数来取代对于任意信号的缺省反应。信号发生时，内核中断当前的进程，进程执行处理函数来响应信号。信号处理完后，进程恢复正常的处理。

以上就是事件通知进程和进程响应异步事件的方式。信号也可以用来同步。进程可以调用 sigpause 等待信号的到来。在早期的 UNIX 版本中，许多应用程序完全在信号的基础上

开发资源共享和锁协议。

信号最初设计目的主要是来处理错误。举个例子，内核把一些硬件异常，如被零除或其他无效指令转换成信号。如果进程没有对这些异常的处理程序，则内核要终止进程。

作为一种 IPC 机制，信号有一些局限性：信号花销太大；发送进程要进行系统调用；内核中断接受进程，而且要管理它的堆栈，同时还要调用处理程序，之后还要恢复执行被中断的进程。更重要的是，信号带宽非常局限，因为只存在 31 种不同的信号（SVR4 或 4.3BSD 中是 31 种；其他一些 UNIX 如 AIX 中多提供一些的信号），而且信号能传送的信息量十分有限，用户产生的信号不可能发送附加信息及各种参量②。信号对于事件通知很有用，但是对于复杂的交互操作，信号是不能胜任的。

信号在第 4 章中有详细的讨论。

6.2.2 管道

在传统的实现中，管道是单向的、先入先出的、无结构的、固定大小的数据流③。写进程在管道的尾端写入数据，读进程从管道的首端读出数据。数据读出后，将从管道移走，其他读进程都不能再读到这些数据。管道提供了简单的流控制机制。进程试图读空管道时，在有数据写入管道前，进程一直阻塞。同样，管道已经满时，进程想写入数据，在其他进程从管道中读走数据（也就是移去数据）之前，这个写进程将发生阻塞。

系统调用一 pipe 生成一个管道并返回两个描述符，一个用来读管道，一个用来写管道。这些描述符为子进程所继承，因此它们可以共享访问文件。通过这种方式，每个管道可以有许多读进程和写进程（见图 6-1）。给定的进程可以是读进程也可以是写进程，或者两者都是。然而，通常管道被两个进程共享，每个进程拥有管道的一端。对管道的 I/O 操作和对文件的 I/O 操作非常相似，通过对管道的描述符调用 read 和 write 系统调用来操作。进程通常不知道它正在读或写的实际上是一个管道。

图 6-1 通过管道的数据流

典型的应用程序，如 shell 控制管道描述符，而管道仅仅有一个读进程和一个写进程，因此使用管道作为单方向的数据流。最普通的管道应用就是让一个程序的输出变为另一个程序的输入。用户通常使用 shell 的管道操作符‘|’来连接两个程序。

从 lPC 的角度看，管道提供了从一个进程向另一个进程传输数据的有效方法。但是，管道有一些固有的局限性：

·因为读数据的同时也将数据从管道移去，因此管道不能用来对多个接受者广播数据。

·管道中的数据被当做是字节流，因此无法识别信息的边界。如果写进程发送不同长度的数据对象通过管道，那么读进程不能确定发送了多少个对象，或是它不能确定对象的边界④。

·如果一个管道有多个读进程，那么写进程不能发送数据到指定的读进程。同样，如果有多个写进程。那么没有方法来判别是它们中的哪一个发送的数据⑤。

实现管道有很多种方法。传统的途径（比如，在 SVR2 中）是利用文件系统机制，把管道同 i 节点（imode）和文件表项（file table entry）联系起来。许多基于 BSD 的 UNIX 变体用套接字（sockets）实现管道。SVR4 提供了双向的、基于流（STREAMS）的管道，在以下章节我们要做描述。

在 System 5 UNIX 和其他许多商用的变体 UNIX 中出现的相关方法，就是 FIFO 文件，也叫做有名管道。它们和管道主要区别是在创建和访问的方式上。用户通过调用 mknod 来创建 FIFO 文件，参数是一个文件名和一个创建模式。创建模式字段包括 S_IFIFO 类型的文件和通常的访问权限。拥有恰当权限的进程才能打开 FIFO 文件并对它进行读或写操作。读

和写 FIFO 文件的语法和对管道的操作非常相似，我们将在 8.4.2 小节做进一步的描述。在对 FIFO 文件进行显式的断开操作（unlink）前，它将一直存在，而不管有没有活动的读或写进程存在。

相比之下，FIFO 文件比管道功能要多一些，它们可以被无关进程访问。它们的存活期更长，这一点对于那些要比活动用户存在时间更长的数据非常重要。FIFO 文件在文件名字空间拥有一个名字。当然，FIFO 文件也有一些不足。当它们不使用时，必须要对它们进行显式的删除。因为任意拥有正确权限的进程都可以访问它们，因此它们没有管道安全。管道要比 FIFO 文件更容易建立而且消耗更少的资源。

6.2.3 SVR4 的管道

SVR4 使用流（STREAMS）（参看第 17 章）作为它的基本框架来进行网络处理和实现管道（pipes）和有名管道（FIFO）。这种机制提供了许多新的和有用的管道特性⑥。本小节将描述这些新功能；17.9 节讨论实现细节。

SVR4 的管道是双向的。系统调用 pipe 像从前一样返回两个描述符，但是，它们都打开，可以进行读操作和写操作。语法（和传统的 UNIX 一样）是：

```
status = pipe(int fildes[2]);
```

在 SVR4 中，这个系统调用创建两个独立的、先入先出的 I/O 通道，用两个描述符来表示。写入 fildes[1]的数据可以从 fildes[0]读出，写入 filde[0]的数据可以从 fildes[1]读出。这是十分有用的，因为许多应用程序都需要双向通信，而在 SVR4 以前的版本中是用两个分开的管道来实现。

SVR4 也允许进程把任意的流（STREAMS）描述符同文件系统中的一个对象联系起来[Pres 90]。应用程序调用 pipe 创建管道，并通过下面的调用把它的每个描述符都连到一个文件名。

```
status = fattach(int fildes, char *path);
```

这里 path 是文件系统中对象的路径，这个路径对调用者是可写的⑦。这里面所说的对象可以是普通文件，目录，或者是其他特殊的文件，但是它不能是一个调用的安装点（不能在其上安装文件系统）或者一个远程文件系统中的对象，同样这个对象不能连接到其他流（STREAMS）文件描述符。可以把这个对象连接到多个路径上，从而有多个名字。

一旦连接以后，所有后继的关于路径的操作就直接作用在流文件上，直到系统调用 fdetach 把文件描述符同路径分离开来。使用这种方法，进程可以创建一个管道，接下来其他无关的进程可以访问这个管道。

最后，用户可以把流（STREAMS）模块加到管道或 FIFO 中。这些模块截获从流中经过的数据并以某种方式来处理。因为，这些模块在内核中运行，它们可以提供用户级应用程序无法完成的功能。没有授权的用户不能往系统内核中添加模块，但是他们可以往他们打开的流中添加。

6.2.4 进程跟踪

系统调用 ptrace 提供了进程跟踪的一组基本的方法。一些调试工具，如 sbd 和 dbx 主要使用系统调用 ptrace。通过使用 ptrace，进程可以控制它的子进程的执行。尽管很少使用，但是使用 ptrace 可以控制多个子进程。ptrace 的语法如下：

```
ptrace(cmd, pid, addr, data);
```

其中 pid 是目标进程的 ID，addr 是目标进程地址空间的一个位置，data 的含义取决于参数 cmd。父进程利用参数 cmd 可以完成以下操作：

- 读或写子进程地址空间中的一个字。
- 读或写子进程 u 区的一个字。

·读或写子进程的通用寄存器。

·截获特殊的信号。当为子进程生成了一个截获的信号，内核会挂起子进程，并把事件通知给父进程。

·设置或删除子进程地址空间中的观测点（watchpoint）。

·恢复停止的子进程执行。

·单步执行子进程——恢复子进程的执行，不过执行完一条指令后再次停止子进程的执行。

·终止子进程。

cmd==0 这个命令为子进程保留。子进程使用这个命令告诉内核它的父进程要跟踪它。内核设置子进程的跟踪标志位（在 proc 结构中），这个标志位反映了子进程是如何响应信号的。如果目标进程产生了一个信号，内核挂起这个进程并通过信号 SIGCHLD 通知它的父进程，而不去调用信号处理程序。通过这种方式，父进程可以截获信号并恰当的处理它。跟踪标志位同样也影响到系统调用 exec 的执行特性。子进程调用一个程序，exec 在返回用户态之前向子进程发送一个 SIGTRAP 信号。同样地，这种方式允许子进程执行之前获得对子进程的控制。

父进程创建子进程，子进程调用 ptrace 让父进程控制它。父进程使用系统调用 wait 等待改变子进程状态的事件的发生。事件发生时，内核通知父进程。Wait 的返回值指出子进程只是挂起而不是终止，同时也提供引起子进程挂起的事件的信息。接下来，父进程通过一个或几个 ptrace 命令控制子进程。

尽管在 ptrace 基础上开发了许多调试器，但是它本身有许多不足和局限的地方：

·进程可以控制它的直接子进程的执行。如果被跟踪的目标进程调用了 fork，调试器没有办法控制新的进程或者它的后代进程。

·ptrace 的效率极低，从子进程传送一个字到父进程需要几个上下文转换。因为调试器没有办法直接访问子进程的地址空间，这些上下文转换是不可避免的。

·调试器不能跟踪已经运行的进程，因为子进程想要被跟踪时，它必须调用 ptrace 来通知内核。

·如果 setuid 程序随后调用了 exec，那么跟踪它将会引起安全问题。狡猾的用户能够使用调试器去修改目标进程的地址空间，这样 exec 调用 shell 而不是请求运行的程序，这样用户就可以获得超级用户的权利。为了避免这个问题，UNIX 或者禁止跟踪 setuid 程序，或是限制在 setuid 和 setgid 后执行 exec 调用。

很长一段时间，prace 是调试程序的唯一工具。现代 UNIX 系统，如 SVR4 和 Solaris。提供了许多非常有效的调试方法，这些方法使用/proc 文件系统[Faul 91]，在 9.11.2 小节中将要详细讨论。它解决了 ptrace 的不足，且增加了一些其他的功能，如进程可以跟踪无关进程，调试器也可以跟踪正在运行的进程。许多调试器都用/proc 重新编写，取代了 ptrace。

6.3 System V 的进程间通信

前面介绍的方法不能满足许多应用程序的 IPC 需求。System 5 UNIX 带来的最大进步就是它提供三种机制——信号量，消息队列和共享内存。这三种机制联合组成了 System 5 的 IPC[Bach 86]，它初设计的目的是用来支持事务式处理的应用需要的。后来，包括那些基于 BSD 的系统开发的大多数 UNIX 供应商都实现了这些机制。本节讨论这三种机制的功能，以及它们是如何在 UNIX 中实现的。

6.3.l 公共元素

这三种机制展现给编程者的接口以及它们的实现方法都非常相似。在讨论它们的公共

特性时，我们使用术语 IPC 资源（或简称为一个资源）来表示单独的信号量集合。消息队列或是共享的内存区域。每一个 IPC 资源具有下列属性：

·键（key）一个由用户提供的整数，用来标志这个资源的实例。

·创建者（creator）创建这个资源的进程的用户 ID（UID）和组 ID（GID）。

·所有者（owner）资源的所有者的 UID 和 GID。资源创建的时候，资源的创建者就是资源的所有者。拥有能改变所有者权力的进程可以给资源指定一个新的所有者。资源的创建者进程、当前的所有者进程和超级用户具有这个权力。

·权限（permissions）文件系统类型的权限，是指资源的所有者进程，同组中的进程和其他用户对资源的读/写/执行权限。

进程通过系统调用 shmget，semget 或 msgget，传递键值、某些标志位（flag）以及依赖于特定机制的其他的参数来获得资源。允许的标志有 IPC_CREAT 和 IPC_EXCL。IP_CREAT 表示如果资源不存在就请求内核创建一个资源。IPC_EXCL 和 IPC_CREAT 联合使用表示如果这个资源已经存在的时候，内核返回一个错误。如果没有指定标志位，那么内核就查找有同样键值的已经存在的资源⑧，如果找到了，并且如果调用者具有访问这个资源的权力，内核将返回一个资源 ID，在以后的操作中可以使用它快速定位这个资源。

每一种机制都有一个用于控制的系统调用（shmctl，semctl 或 msgctl）。它们提供一些命令，这些命令包括：IPC_STAT 和 PC_SET 以获得和设置资源的状态信息（依靠于特定的机制）。IPC_RMID 释放这个资源。信号量机制提供获得和设置信号量集合中的单个信号量值的其他控制命令。

IPC 资源必须使用 IPC_RMID 命令来显式地释放。否则，内核会认为这个资源仍然处在活动状态，甚至所有的使用它的进程都已经终止。然而这种特性十分有用，比如，进程可以向共享内存区或消息队列写数据，然后退出；过一段时间后，另一个进程可以从这个资源获得数据。IPC 资源可以长久的存在，超过访问它的进程的存活期仍然可用。

但是这种机制存在缺点。内核很难判断，资源是为了将来进程访问而故意留下来处于调用状态，还是被无意地抛弃了，或是由于想要释放它的进程在操作前不正常终止了。结果造成内核必须无限地保存这类资源，如果这种情况经常发生，系统就会用光这类资源。至少，占有内存的资源不能被更好地使用。

只有创建者进程、当前的占有者进程或是超级用户进程能够使用 IPC_RMID 命令。释放 IPC 资源将影响当前访问它的所有进程，内核必须保证这些进程能一致地处理好这个事件。这个操作的细节对于不同的 IPC 机制有所不同，在下面将详细讨论。

为了实现这个接口，每类资源都有它自己固定大小的资源表。资源表的大小可以进行设置，它限制了在系统中同时活动的那种 IPC 机制的实例的总数。资源表表项包含一个公共的 ipc_term 结构和那种类型资源的部分信息，结构 ipc_term 包含资源的公共属性（键，创建者和占有者的 ID 及权限），也包含一个顺序号，它是一个计数器，每次表项使用时，它就加“1”。

用户分配到一个 IPC 资源时，内核返回这个资源的 ID。ID 是通过下面的公式来计算的：

```
id = seq * table_size＋index;
```

其中，seq 是资源的顺序号，table_size 是资源表的大小，index 是资源在表中的索引。因为 seq 是递增的，这样就保证了表中的元素重新被使用的时候，生成的是一个新的 ID，从而防止了使用一个过期（stale）的 ID 去访问一个资源。

注意：增加一个变量是指对这个变量的值加 1；减少一个变量是指对这个变量的值减 1，这些术语来自 C 的操作符号加（++）和减（--）。

用户在随后的对那个资源的调用时，以 id 为参数。内核用下面的公式转换 id，在资源表中定位资源。

```
index = id % table_size;
```

6.3.2 信号量

信号量[Dijk 65] 是具有整数值的对象。它支持两种原子操作 P( )和 V( )⑨。P( )操作减少信号重的值，如果新的信号量的值小于 0，则操作阻塞；V( )操作增加信号量的值；如果结果值大于或等于 0，V( )操作就要唤醒一个等待的线程或进程（如果存在的话）。这两个操作是原子的。

信号量可以用来实现一些同步协议。比如，考虑管理一个计数资源，也就是说资源有固定数目的实例。进程想获得资源的一个实例，当它使用完这个资源后释放它，这个资源能用初始化的数值实例的信号量表示。想获得资源时使用 P( )操作，每次请求成功它都要减少信号量的值。信号量的值减至 0 时（无空闲可用资源），下一个 P( )操作将被阻塞。释放资源的时候使用 V( )操作，它增加信号量的值，同时唤醒被阻塞的进程。

许多 UNIX 系统中，内核使用信号量来同步它的操作。向应用程序提供同样的方法也是很有意义的。System 5 提供一个通用版本的信号量。系统调用 semget( )创建或获得一个信号量数组（数组大小有上界），它的语法是:

```
semid = semget(key, count, flag);
```

其中，key 是一个 32 位的用户提供的值。调用 semget( )返回一组和 key 值相关的计数信号量。如果没有信号量集合与 key 值相关，而且用户没有提供 IPC_CREAT 标志位，则调用失败。否则系统创建一个新的信号量集合。如果调用者同时也提供了参数 IPC_EXCL，那么，如果那个键值的信号量集合已经存在，则 semget 返回一个错误（error）。以后的信号量操作使用 semid 值来识别信号量数组。

系统调用 semop( )用来对数组中的单个信号量进行操作。语法如下：

```
status = semop(semid, sops, nsops);
```

其中 sops 是指向 sembuf 结构中 nsops 元素的一个指针。每一个 sembuf，如下面描述的那样，代表对信号量集合中单个信号量的一个操作：

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

sem_num 识别数组中的一个信号，sem_op 指定执行的操作，sem_op 值的含义如下；

sem_op > 0 把 sem_op 的值加到当前的信号量值上，这可能唤起那些等待信号量值增加的进程。

sem_op = 0 信号量值变为 0 以前一直阻塞。

sem_op < 0 在信号量的值大于或等于 sem_op 的绝对值之前，一直阻塞，接着，减去 sem_op 的值。如果信号量的值已经大于 sem_op 的绝对值，调用者就不用阻塞。

因此，一个 semop 调用能指定几个操作，并且内核保证所有的（或者部分）操作的完成。进一步，内核保证在这个操作完成或阻塞之前，这个数组中的其他操作可以开始。如果执行了部分操作后，sem_op 调用必须阻塞，那么内核会返回到操作的开始处（恢复所有的修改），从而保证了调用的原子性。

参数 sem_flg 能向调用提供两个标志。标志 IPC_NOWAIT 要求内核返回一个错误，而不是阻塞。同样地，进程没有释放信号量而永久地退出时，会发生死锁。其他想得到这个信号量的进程将永远地阻塞在 P( )操作中。为了避免这个问题的发生，semop 可以使用标志 SEM_UNDO。使用这个标志，内核会记住这个操作，进程退出时，系统就会撤消操作。

最后．必须在调用 semctl 中显式地使用 IPC_MID 来释放信号量。否则，不管其他进程是否还使用，系统将保存这个信号量。在需要信号量存在的时间超过进程时，这种特性有意义。但是，当应用程序退出时没有释放掉信号量，它们会用光系统资源的。

进程发出一个 IPC_RMID 命令时，内核清除在资源表中的信号量。内核也唤醒阻塞在信号量操作上的所有进程；这些进程从 semop 调用返回一个 EIDRM 状态信息。一旦信号量被释放，进程不能再访问它了（无论是使用键值还是信号量 ID）。

实现的细节

内核通过转换 semid 来获得信号量资源表表项，这个表项用下面的数据结构来描述：

```
struct semid_ds {
    struct ipc_perm sem_perm;          /* see section 6.3.1 */
    struct  sem*  sem_base;            /* pointer to array of semaphorse in set */
    ushort sem_nsems;                  /* number of semaphorse in set */
    time_t sem_otime;                  /* last operation time */
    time_t sem_ctime;                  /* last change time */
};
```

内核在下面的结构中维护集合中的每个信号量的值和同步信息；

```
struct  sem{
    ushort semval;          /*  current  value  */
    pid_t sempid;           /* pid of process tha invoked the last operation */
    ushort semncnt;         /* num of procs waiting for semval to increase */
    ushort semzcnt;         /* num if procs waiting for semval to equal 0 */
    };
```

最后，内核为每一个使用 SEM_UNDO 标志的调用信号量操作的进程维护一个 undo 列表。这个列表包含一组必须回滚的操作。当一个进程退出时，内核要检查它是否有一个 undo 列表。如果有的话，内核就遍历这个 undo 列表，并撤销（reverse）所有的操作。

讨论

信号量机制为互相操作的进程提供了一种复杂的同步方法。早期的 UNIX 系统不支持信号量。需要同步的应用程序使用 UNIX 中的其他原子操作。其中一个就是系统调用 link，如果新的 link 已经存在，那么调用就失败。如果两个进程同时调用相同的 link 操作，那么只能有一个能成功。但是，如果使用 link 这样的文件系统操作只是为了进程的同步，则是非常昂贵而且无太大意义。信号量机制补充了应用程序开发人员的主要需求。

信号量机制存在的主要问题包括竞争和死锁避免。如果进程想获得多个信号量的话，简单的信号量（和信号量数组相对）很容易引起死锁。举个例子，在图 6-2 中，进程 A 占有信号量 S1 并且想获得信号量 S2，而占有信号量 S2 的进程 B 想获得信号量 S1，这两个进程都不能向前运行。尽管这是个简单的情况，很容易探测和避免。但是死锁可以发生在任意复杂的情况下，并且可能包括许多个信号量和大量的进程。

把死锁探测和死锁避免的代码放在内核中是不实际的。进一步地说，没有通用的而且有界的算法能适用所有的死锁情况。因此，内核把死锁检测的任务留给应用程序去做。通过提供有复合原子操作的信号量集合，内核提供了灵活处理多个信号量的机制。应用程序可以从一些著名的死锁避免算法中去选择。其中的一些将在 7.10.1 小节中讨论。

图 6-2 信号量可能引起死锁

System 5 信号量实现中存在一个主要的问题，就是分配和初始化信号量的操作不是原子的。用户使用系统调用 semget( )去分配信号量集合，接下来调用 semctl( )去初始化它。这将

导致竞争，必须在应用程序级上防止这种竞争的发生[Stev 90]。

最后，所有的 IPC 机制都需要显式地使用命令 IPC_RMID 来释放资源、尽管允许资源的存活期超过它的创建者的存活期，但是如果进程在退出时没有释放这些资源，就可能产生一个垃圾收集的问题。

6.3.3 消息队列

消息队列就是消息链表的头部指什。每个消息包含一个 32 位的类型值，接下来是数据区域。进程通过系统调用 msgget( )来创建或获得一个消息队列。语法如下：

```
msgqld = msgget(key, flag);
```

这个调用的语义和调用 semget 相似。键值（key）是用户选择的整数。标志位 lPC_CRAT 用来创建一个新的消息队列。如果对应那个键值已经存在一个队列，并且同时使用了 IPC_EXCL 标志位，调用将失败。其他的调用可以使用 msggid 值访问这个消息队列。

用户通过下面的调用来把消息放入队列中：

```
msgsnd(msgqid, msgp, count, flag);
```

其中 msgp 指向消息缓冲区（包含一个跟随数据区域的类型域），count 是消息中全部字节的数目（包括类型域）。发送消息时发生阻塞（如果队列已满，比如，队列有一个设置好的界限来限制它所能包括的数据的全部数目），可以使用标志位 IPC_NOWAIT 返回一个错误码。

图 6-3 描述了消息队列的操作。每个队列在消息队列资源表中有一个表项，并用下面的结构来表示：

```
struct msqid_ds{
    struct ipc_perm msg_perm;   /* described in Section 6.3.1 */
    struct msg* msg_first;      /* first message on quene */
    struct msg* msg_last;       /* last message on quene */
    struct msg_cbytes;          /* current byte count on quene */
    ushort msg_qbytes;          /* max bytes allowed on quene */
    ushort msg_qunm;            /* number of messages currently on quene */
};
```

消息在队列中按到来的顺序来维护。进程读消息时，这些消息从队列中移去（先入先出，即 FIFO），使用下面的调用：

```
count = msgrcv(msgqid, msgp, maxcnt, msgtype, flag);
```

图 6-3 使用消息队列

在这里，msgp 指向放置到来消息的缓冲区，maxcnt 限制了能够被读到的数据的大小。如果到来的数据大于 maxcnt 规定的字节数，它将被截断、用户必须保证 msgp 所指的缓冲区应该能容纳 maxcnt 个字节。返回值 count 指出成功读到的字节数目。

如果 msgtype 等于 0，msgrcv 返回队列中的第一个消息。如果 msgtype 大于零。msgrcv 返回的是 msgtype 型的第一条消息。如果 msgtype 小于 0，则 msgrcv 返回的是小于或等于 msgtype 绝对值的最低类型的第一个消息。同样，如果合适的消息没有在队列上，且使用了标志 IPC_NOWAIT 则调用立即返回。

一旦读出，消息将从队列中移去，其他的进程不能再读到了。同样地，如果因为接收的缓冲区太小造成消息被截断，截断的部分将永久地丢失而不给接收者任何提示。

进程必须通过带有 IPC_RMID 命令的 msgctl 调用来显式地删除消息队列。删除发生时，内核就释放消息队列，同时删除队列中的所有消息。如果这时有读或是写这个消息队列而阻

塞的进程，内核将唤醒它们，它们将从调用中返回，并返回一个 EIDRM 状态（消息 ID 已经被移去）。

讨论

消息队列和管道提供相似的服务。但是消息队列功能要更加强大并解决了管道中所存在的一些问题。消息队列传递数据时是以一种不连续消息的方式，而不是用一种无格式的字流的方式。这样就可以更加灵活的来处理数据。可以不同的方式来使用消息的类型域。比如，它可以同消息的优先级联系起来。这样接收者就可以在读不紧急消息之前检查紧急消息。在多个进程共享一个消息队列的环境中，类型域可以用来指定接收者。

传输小的数据块时使用消息队列效率很高。但是在传递大量数据时，则花费很高。进程发送一个消息时内核把数据拷贝到内部缓冲区中。另一个进程接收这个消息时，内核把数据再拷贝到接收者的地址空间中。因此，消息的传送包括两次拷贝操作，从而降低了系统的性能。本章的余下部分中，我们将看到 Mach IPC 机制是如何实现高效的大块数据的传送的。

消息队列的另一个局限性，就是它不能指定接收者。任何拥有适当权限的进程都可以从队列中读取消息。尽管如前面提到的那样，互相操作的进程可以通过达成的协议来指定接收者，但是内核并不支持这种操作。最后，消息队列不支持广播机制。因此进程发出的一个单独的消息，不能为许多接收进程所接收。

大多数现代的 UNIX 系统中的流（STREAMS）框架为消息传递提供了一组丰富的方法。流框架比消息队列提供更多的功能，也更有效。相比之下，消息队列很过时、但是消息队列中存在，而流框架中没有的一个特性就是在消息队列中，可以在优先级的基础上有选择地接收消息。这对于某些应用是很有用的，但是，大多数的应用都认为流机制更加有用。现在的 UNIX 系统中保留消息队列主要的原因是为了保持向上的兼容性。第 17 章中我们将详细地讨论流机制。

6.3.4 共享内存

共享内存区域是被多个进程共享的一部分物理内存。进程可以把这些区域映射到它们地址空间中的任一合适的虚拟地址范围、这些地址范围对每一个进程来说可以是不同的（见图 6-4）。映射后，这些区域就可以像其他任何内存位置那样被访问，而不需对它使用读（read）或写（write）调用。因此，共享内存机制提供了进程共享数据的最快的方法。进程向共享内存区域写入了数据，那么共享这个区域的所有进程可以立即看见共享区域中新的内容⑩

图 6-4 映射一片共享内存区

进程首先通过下面的调用来创建和获得一块共享内存区域：

```
shmid = shmget(key, size,flag);
```

其中，size 是要共享的内存区域的大小，其他的参数和标志位同系统调用 semget( )和 msgget( )中的作用相同。接下来，进程用下面的调用把这个共享共存区域同一个虚拟地址范围联系起来。

```
addr = shmat(shmid, shnaddr, shmflag);
```

参数 shmaddr 指出要和共享内存区联系的区域的地址，参数 shmflag 指定 SHM_RND 标志位。使用这个标志位，内核可以用一个合适的对齐因子来舍入（round down）地址 shmaddr。如果 shmaddr 等于 0，内核就可以自由选择任何地址。标志位 SHM_RDONLY 指出映射的内存区域是只读的。Shmat 调用返回映射的共享内存的实际地址。

进程可以使用下面的调用把共享的内存区域和它自己的地址空间分离开来。

```
shmdt(shmaddr);
```

为了完全的释放共享的内存区域，进程必须使用带有 IPC_RMID 命令的 shmctl 调用。

它标志要释放内存区域，当所有的进程都把这个共享内存区域同自己的地址分离后，这个共享内存区域将被释放。内核记录和共享内存联系的进程的数目。一旦共享内存区域被标志为释放，新的进程就不能再和它连接。如果共享内存区域不是显式地删除，内核仍然会保留它，而不管是否还有进程与它连接。某些应用可能需要这些特性。进程可以在共享内存区域放入一些数据，然后终止。过一段时间后，另一个和它有互相操作关系的进程能使用相同的键值连接到共享内存区域，并读取其中的数据。

共享内存的实现和操作系统的虚拟内存体系结构关系十分密切。一些系统使用单个页表来映射共享内存区域，对所有连接到共亭内存区域的进程都共享这些表。其他系统中，每个进程对共享内存区域都有单独的地址转换图。这种模型中，如果进程执行操作改变了一个共享页的地址映射，这个改变必须应用到对那个页的所有映射上。系统 SVR4 中，使用一个 anon_map 结构来定位共享内存区域的页。SVR 中的内存管理在第 14 章中介绍。它的共亭内存资源表中包含的表项用下面的结构来表示：

```
struct shmid_ds{
    struct ipc_perm shm_perm;       /* described in Section 6.3.1 */
    int shm_segsz;                  /* segment size in bytes */
    struct anon_map *shm_amp;       /* pointer to memory info */
    ushort shm_nattch;              /* number or current attaches */
};
```

共享内存提供了一种快速的灵活的机制，允许不用拷贝的方法或是使用系统调用就可以共享大量的数据。它的主要局限性就是它不能提供同步。如果两个进程企图修改相同的共享内存区域，内核不能串行化这些操作，因此写的数据可能任意地互相混合。使用共享内存的进程必须设计它们自己的同步协议。它们通常使用像信号量的原语。这些原语包括一个或多个系统调用，这就对共享内存的性能加上了一些额外的开销。

大多数现代的 UNIX 系统（包括 SVR4）也提供系统调用 mmap，它把一个文件或是文件的一部分映射到调用者的地址空间中。进程可以使用 mmap 来进行进程间通信。把相同的文件映射到它们的地址空间中（以 MAP_SHARED 方式），效果同初始化为文件内容的共享内存区域相似。如果进程修改了一个映射了的文件，那么这个改变对于那些映射了这个文件的所有进程立即可见。内核同时也要更新磁盘上的文件。系统调用 mmap 的一个好处就是，它使用文件系统的名字空间而不是键值。共享内存的页是由交换空间来支持的（见 14.7.6 小节），而 mmap 的页是由它们映射的文件来支持的。14.2 节中将详细讨论系统调用 mmap。

6.3.5 讨论

IPC 机制和文件系统间有一些相似之处。资源 ID 和文件描述符相似，系统调用 get 和系统调用 open 相似，命令 IPC_RMID 和命令 unlink 相似，调用 send 和 rceive 与 read 及 write 相似，系统调用 shmdt 为共享内存提供了一个类似 close 的功能。然而，对于消息队列和信号量来说，没有系统调用同 close 等价。这可能对于完全地释放资源是很必要的。结果，使用消息队列或信号量的进程可能突然发现那个资源已经不再存在。

相比之下，与资源相关的键值形成的名字空间和文件系统的名字空间有着显著的不同[11]。每一种机制都有它自己的名字空间。键值唯一地识别名字空间中的一个资源。因为键值是简单的、用户选择的整数，因此它只能在单独的一个机器中有用，而不适合分布式的环境。同时，对于无关的进程选择一个整数作为键值，并且避免和其他应用使用的键值冲突是很困难的。因此，UNIX 提供了一个库例程叫做 ftok（在手册的页 StdipC（3C）中介绍），它在文件名和整数的基础上生产一个键值。其语法如下：

```
key = ftok(char *pathname, int ndx);
```

ftok 例程通常在 ndx 和文件的 i 节点的基础之上产生一个键值。应用选择唯一的文件名比唯一的整数值要容易很多（比如，它可以使用它的执行文件的路径名），因此它可以减少键冲突的可能性。参数 ndx 允许更大的灵活性，它能用来指定一个为所有互相操作进程已知的项目 ID。

因为资源 ID 实际上是全局资源表中的一个索引，因此，安全是一个摆在人们面前的问题。未被授权的进程可以简单地猜测资源的 ID 来访问这个资源，这样它能够读或者写消息队列或共享内存区域，或者是篡改被其他进程使用的信号量。对每个资源相关的权限提供一些保护。但是许多应用必须和属于不同用户的进程共享这些资源，因此不能使用非常严格的权限。使用顺序号作为资源 ID 的一个组成部分提供了更多的保护。因为有许多 ID 可以从中猜测，所以对于那些与安全有关的应用提出了一个严峻的问题。

System 5 中的 IPC 提供的许多功能都可以用其他文件系统的方法来复制实现。如文件锁和管道。但是，IPC 手段更加灵活和通用，比文件系统提供了更好的性能。

6.4 Mach IPC

本章的其余部分将讨论 Mach 的基于消息的 IPC 机制。在 Mach 中，IPC 是内核中最重要的基本组成部分。与传统操作系统支持 IPC 机制的想法不同，Mach 的 IPC 机制支持大多数的操作系统。以下是 Mach IPC 设计的一些重要目标：

·消息传递必须是最基本的通信机制。

·在一个单独的消息中所携带的数据的大小可以从几个字节到整个地址空间（典型地到 4G）。内核应该保证大量的数据传输而无需那些不必要的拷贝操作。内核应该提供安全的通信。只允许授权的线程能够发送和接收消息。

·通信和内存管理是紧密相连的。IPC 子系统使用内存管理子系统的写拷贝（copy-on-write）机制来有效地传输大量的数据。反过来内存管理子系统使用 IPC 来和用户级的内存管理器通信（外部页面管理器）。

·Mach IPC 支持用户任务之间的通信，也支持用户和内核之间的通信。在 Mach 中，线程通过向内核发送消息来进行系统调用，内核在答复的消息中返回调用的结果。

·IPC 机制应该适合客户-服务器模型的应用。Mach 使用用户的程序来执行许多传统上由操作系统内核完成的服务（如文件系统和内存管理）。这些服务器使用 MackIPC 来处理对这些服务的请求。

·IPC 接口应该能透明地扩展到分布式环境。用户无需知道他发送的消息是给本地节点的接收者还是给远程节点的接收者。

Mach 的 IPC 机制经过许多版本的演变。在 6.4 节和 6.9 节中将要讨论 Mach 2.5 中的 IPC 机制，它是 Mach 版本中最流行的并且是操作系统 OSF/l 和 Digtal UNIX 等的基础。Mach 3.0 的 IPC 在许多方面有了改进和完善，这些特点在 6.10 节中介绍。

本章对 Mach 的任务和线程做了一些说明。3.7.l 小节详细地讨论了这些抽象概念。简言之，任务就是资源的集合，包括一个或多个线程执行的地址空间。线程是一个动态的执行实体，它表示程序中一个独立的程序计数器和堆栈——也是一个逻辑控制序列。UNIX 的进程相当于只包含一个单独线程的任务。任务中的线程共享任务中的所有资源。

6.4.1 基本概念

Mach 中的两个基本的 IPC 抽象概念是消息和端口[Rash 86] 消息是有类型数据的集合，端口是一个保护的消息队列。消息只能发送到端口，而不能发送到线程或是任务本身。Mach 对每一个端口赋予发送权和接收权，这些权限都属于任务。发送权允许任务向端口发送消息、接收权允许任务接收发送到端口的消息。可能有几个任务拥有对一个端口的发送权。但是只

能有一个任务（端口的所有者）拥有对端口的接收权[12]。 因此，端口允许多对一的通信，如图 6-5 中描述的那样。

消息可以简单也可以是复杂的。简单的消息包含普通的数据，不用被内核解释。复杂的消息可以包含普通的数据、Out-of-line 内存（通过引用传递的数据，使用写拷贝语义），以及对不同端口的发送权和接收权。内核对复杂消息中的信息进行解释，并把它们转换到对接收者有意义的形式。

每一个端口都有一个引用计数来监视对它的权限数目。每一个这样的权限（也被认为是能力(capacity)）代表那个端口的一个名字。名字是整数，名字空间对每个任务是局部的。因此两个任务对相同的端口可能有不同的名字（图 6-6）。同样，相同的端口名字在不同的任务中可能是指不同的端口。

图 6-6 端口的本地名

端口也可以表示内核对象。因此每一个对象如任务，线程或是处理器都可以用端口来表示。这些端口的权限代表对对象的引用，并且允许持有这些权限的任务对那个对象执行操作。内核对这些端口具有接收权。

每个端口有一个有限大小的消息队列。这个队列的大小提供一个流控制机制。当队列满的时候，发送者将被发生阻塞。当队列空的时候，接收者将被阻塞。

每一个任务都有一个缺省的端口集合。比如，每个任务对代表它自己的 task_self 端口有发送权（内核对这个端口有接收权），对端口 task_notifv 有接收权（内核对这个端口有发送权）。任务也对那些提供对名字服务器访问的端口 bootstrap 有发送权。每个线程对端口 thread_self 有发送权，对端口 reply 有接收权，来接收系统调用的应答和对其他任务的远端过程调用。每一个任务和线程有一个异常端口和它相联系。每线程端口权力由线程在其中运行的任务所拥有。因此，这些端口能够被任务中的所有线程访问。

任务也从它们的父任务那里继承其他一些端口权力。每个住务都有一些注册的端口。这些端口允许任务访问系统范围的不同的服务，这些端口在新任务创建的过程中被继承。

6.5 消 息

Mach 是基于消息传递的内核。许多系统服务通过消息交换来完成。Mach 的 IPC 机制提供不同用户任务间、用户和内核间以及不同的内核子系统间的通信。使用被称为 netmsgserver 的用户级程序，MgCh IPC 透明地在网络上扩展。这些任务同远程任务交换消息就像图 6-5 通过 Mach 端口通信和本地任务交换消息一样容易。Mach IPC 的最基本的抽象概念是消息和端口。本节将描述实现这些抽象概念的数据结构和函数。

6.5.1 消息的数据结构

消息是有类型数据的集合。它包含三种基本类型的数据：

- 普通数据，不用内核解释，通过物理的拷贝就可以直接传递给接收者。
- 脱机内存，使用写拷贝（cony-on-write）技术来传输大量的数据，详见 6.7.2 小节。
- 对端口的发送权和接收权。

消息包括一个固定大小的头部，接着是可变大小的数据集合部分（见图 6-7）。这些消息的头部包括以下的信息:

- 类型 简单的（只是普通数据）和复杂类型（脱机内存或是端口权力）。
- 大小 包括头部的消息大小。
- 目的端口。
- 应答端口（reply port） 想发送应答的接收者对这个端口有发送权；当发送者需要应

答时才设置这个字段。

·消息 ID 应用程序需要时使用。

图 6-7 Mach 的消息

消息是在发送前在发送任务的地址空间形成的。此时，目的端口和回答端口是这些端口的任务的本地名字。在传递消息前，内核必须把这些端口名字转换为对接收者有意义的值。数据部分包括一个类型描述符和其后的数据本身。描述符包含以下信息:

·名字 识别数据的类型。Mach 2.5 能识别出 16 种不同的名字值，包括内存（internal memory），端口发送权或接收权，以及像字节、16 位整数、32 位整数、字符串或实数等标量。

·大小 数据组成部分的每个数据项的大小。

·数目 数据部分数据项目的数目。

·标志 指出数据是联机类型还是脱机类型，以及内存或端口权力在发送者任务中是否一定要被释放。

6.5.2 消息传递接口

应用程序可以以几种不同的方式来使用消息传递：

发送消息，不需要应答。

等待那些没有请求的消息，并在它们到来时处理它们。

发送消息并要求应答，但不等待应答。应用程序异步地接收应答并在以后的合适的时间对它进行处理。

发送消息并等待应答。

消息传递的编程接口[Baro 90]由三个函数组成，它们联合起来完成上面那些形式的通信：

```
msg_end(msg_header_t *hdr,
                    msg_ptlon_t  option,
                    msg_timeout_t  timeout);
msg_rcv(msg_header_t *hdr,
                    msg_option_t  option,
                  msg_timeout_t  timeout);
msg_rpc(msg_header_t *hdr,
                    msg_size_t  rcv_size,
                    msg_option_t  option
                    msg_timeout_t  send_timeout,
                  msg_timeout_t  receive_timeout);
```

调用 msg_send( )发送消息但是它并不等待应答。如果目的端口的消息队列已经满了，这个调用将被阻塞。同样地，如果调用 msg_rcv( )，那么在接收到消息之前也将阻塞。每一个调用都接受 SEND_TIMEOUT 和 RCV_TIMEOUT 选项。如果指定了这两个选项，调用就阻塞到最大的超时时间。时间超时之后，调用就返回一个 time_out 状态码，而不是一直等下去。使用选项 RCV_NO_SENDERS 时，如果没有任务对这个端口具有发送权用，那么调用 msg_rcv 返回。

调用 msg_rpc 向外发送消息，接着等待应答消息到来。它仅仅是以优化的方式执行调用 msg_send( )并跟随 msg_rcv( )调用。应答消息仍然使用向外发送消息的消息缓冲。调用 msg_rpc 包括所有 msg_send( )和 msg_rcv( )的选项。

消息的头部包含消息的大小。调用 msg_rcv 时，头部包含调用者能接收的最大消息的大小；返回时，头部包含实际接收的消息的大小。调用 msg_rcv( )时，rcv_size 必须单独被指定，因为消息的头部包含有向外发送的消息大小。

6．6 端　口

端口是被保护的消息队列。任务能获得对端口的发送权和接收权或是能力（capabilities）。只有拥有适当权限才能访问端口。尽管许多任务可以对端口拥有发送权，但是只能有一个任务对端口拥有接收权。拥有接收权的任务自动地对那个端口拥有发送权。

也可以用端口代表 Mach 的对象如任务、线程和处理器。内核对这些端口拥有接收权。端口是计数引用的，发送权组成了端口所代表的对象的一个引用。通过这个引用允许所有者管理它代表的对象。比如，任务的 task_self 端口代表这个任务，其他任务可以向这个端口发送消息来请求影响那个任务的内核服务。如果另一个任务，如调试器（debugger）也对那个端口具有发送权，它就可以通过向端口发送消息对这个任务进行操作。如挂起这个任务，允许指定的操作依靠于端口对象以及它输出的接口。

本节描述端口的名字空间以及用来代表端口的数据结构。

6.6.1 端口名字空间

能力（capacity）或端口权力代表端口的一个名字，这些名字就是简单的整数。名字空间对于任务来说是局部的。因此，不同的任务对于相同的端口具有不同的名字；反过来，相同的名字可能在不同的任务中代表不同的端口。在这种意义上，端口名字和 UNIX 中的文件描述符类似。

任务对任何端口最多有一个名字。可以通过消息传递端口权力。因此任务可以多次得到对同一个端口的权力。内核保证每次都使用相同的名字。这样，任务可以比较两个端口名字——如果它们不匹配，那么它们指的不是相同的端口。

端口也用一个全局的内核数据结构来表示。内核必须把本地的端口名字转换为全局的端口名字（对那个端口的全局数据结构相同的地址），反之亦然。在文件描述符的例子中，每个 UNIX 进程在它的 u 区内维护一个描述符表来存放指向相应打开文件对象的指针。Mach 使用另一种不同的转换方法，在 6.6.3 小节我们再进行介绍。

6.6.2 端口数据结构

内核为每个端口维护一个 kern_port_t 数据结构。它包含下列信息：

- 对这个端口的所有名字（权力）的引用数目。
- 指向对端口有接收权的任务的指什。
- 在有接收权的住务中这个端口的局部名字。
- 指向后备（backup）端口的指针。如果这个端口被释放了，那么所有发送到这个端口的消息就发送到这个后备端口中。
- 双向链接的消息。
- 被封锁的发送者队列。
- 被封锁的接收者队列。尽管只有一个任务对端口具有接收权，但是在任务中的许多线程可能都在等待接收一个消息。
- 对这个对象的所有转换的链表。
- 指向端口集合的指针，以及指向端口集合中下一个和前一个端口的指针（如果这个端口属于一个端口集合）（见 6.8.3 小节）。
- 当前在队列中的消息个数。

・允许在队列中的最大消息数目（backlog）

6.6.3 端口变换

Mach 对每一个端口权力维护一个变换项。这个变换项必须能够管理＜任务，端口，局部名字，类型＞四元组的集合。其中任务是指拥有权力的任务，端口是指向端口内核数据结构的指针，局部名字是指在任务中端口的名字，类型是指发送或是接收。Mach 以几种不同的方式使用这个变换。

・msg_send 必须把＜任务，局部名字＞转换为端口。

・msg_rcv 必须把＜任务，端口＞转换为局部名字。

・当任务释放端口时，它必须找到这个端口的所有权力。

・当任务被撤销时，内核必须能够找到对那个任务的所有端口变换，并释放相应的引用。

・当端口被释放时。内核必须能够找到对这个端口的所有变换并修改对它拥有权力的任务。

这样就需要变换策略有效地支持上面所有的操作。图 6.6 描述了 Mach 2.5 中端口变换的数据结构。Mach 使用了两个全局的散列（hash）表来快速地找到变换项——表 TP_table 在＜任务，端口＞的基础上散列变换项，表 TL_table 在＜任务，局部名字＞的基础上散列这些变换项。数据结构 kernel_port_t 和 task 分别保存对端口或任务的交换链表的头指针。

因此在图 6-8 中，变换项 a，b，c 描述了在相同任务中的不同端口的变换，而变换项 c，d，e 是在不同的任务中相同的端口的变换。在哈希表 TP_table 中变换项 b，d，f 散列有相同的索引。而在哈希表 TL_table 中变换项 e 和 g 散列具有相同的索引。每一个变换项用一个 port_hash_t 结构来表示。这个结构包含下列信息：

・任务 拥有这个权力的任务。

・局部名字 在这个任务中权力的名字。

・类型 发送或者接收。

・obj 指向内核中端口对象的指针。

此外，每个变换项在以下的每一个双向链表上。

・TP_chain 在＜任务，端口＞基础上的散列链表。

・TL_chain 以＜任务，局部名字＞为基础的散列链表。

・task_chain 被相同任务拥有的所有变换项列表。

・Obj_chain 这个端口的所有变换项列表。

6.7 消息传递

消息传递需要以下的几个操作：

图 6-8 Mach 的端口转换

1．发送者在自己的地址空间中创建这个消息。

2．发送者调用系统调用 msg_send 来发送这个消息，这个消息的头部包含有目的端口。

3．内核利用例程 msg_copyin( )来把消息拷贝到内部数据结构 kern_msg_t 中。在这个过程中，端口权力转换为指向端口内核对象的指针，同时脱机内存被拷贝到一个保留的映射中。

4．（a）如果有一个线程在等待接收这个消息（这个线程在这个端口的被阻塞的接收者队列中），那么它将被唤醒，同时消息直接就传给了这个线程。（b）否则，如果这个消息

队列已经满了，那么发送者就会被封锁，直到有一个消息从队列中移去。（c）否则，消息就在那个端口排队，直到接收者任务中有一个线程执行了调用 msg_rcv( )。

5．当消息在端口排队或是传给了接收者，内核就从 msg_send 调用返回。

6．当接收者调用 msg_rcv 时，内核就调用 msg_dequeue( )来把一个消息从队列中移去。如果队列是空的，接收者在消息到达前将一直被封锁。

7．内核利用函数 msg_copyout( )把消息拷贝到接收者的地址空间。如果是脱机内存或者是端口权力，那么这个函数要对数据做进一步的变换。

8．通常，发送者需要一个应答。为此，接收者必须对发送者拥有的端口有发送权。在这种情况下，发送者要使用 mach_rpc( )调用来优化这种交换。这个调用在语义上和调用 msg_send 后紧跟着调用 msg_rcv( )相同。

图 6-9 描述了在传输的过程中。消息的不同组成部分的转换。下面让我们详细地看一些有关消息传递的重要问题。

6.7.1 端口权力的传递

有些原因要求在消息中传递端口权力。最普通的情况就是应答端口（见图 6-10）。在任务 T1 中的线程向端口 P2 发送消息，端口 P2 由任务 T2 所属。在这个消息中，T1 把发送权传递给端口 P1。任务 TI 对端口 P1 具有接收权，这样任务 T2 能向端口 P1 发送应答消息，发送线程将等待这个消息。这种情况十分常见，即在消息的头部包括一个保存应答端口权力的字段。

另一个十分普遍的情况就是一个服务器程序、一个客户程序以及一个名字服务器三者的交互操作（见图 6-11）。名字服务器拥有对系统中一些服务器程序的发送权。一般地，当服务器程序开始执行时，它们都要在名字服务器中注册（a）。所有的任务在创建的过程中都继承了对名字服务器的发送权（这个值保存在 task 结构的 bootstrap 端口域中）。

图 6-9 消息传送的两个阶段

图 6-10 消息可以包含应答端口的发送权

客户程序想要访问服务器程序时，它首先必须获得对服务器程序拥有的端口的发送权。为此，客户程序查询服务器程序（b），名字服务器返给客户一个发送权（c）。客户使用这个发送权向服务器程序发请求（d），这个请求包含一个应答端口，服务器程序可以使用它来对客户做应答（e）。这样，进一步的客户和服务器之间的操作就不用包括名字服务器了。

发送者使用端口的局部名的来发送端口权力、消息的类型描述符组成部分告诉内核所要发送的数据是端口权力。因为端口的局部名字对于接收者无任何意义，所以内核必须转换它。为此，内核通过＜任务，局部名字＞散列搜寻变换项并识别那个端口对应的内核对象（全局名字）。

图 6-11 使用名字服务器来启动客户与服务器间的交互

提取消息时，内核必须把端口的全局名字转换为接收任务的局部名字。首先，内核检查接收者是否对这个端口具有权力（通过散列＜任务，端口＞）。如果它有权力，内核把它换为相同的名字。否则内核就在接收者那里分配新的端口名字，并为这个端口的映射名字创建新的变换项。端口名字通常是个小整数，内核使用可能最小的整数来做端口的名字。

因为内核为那个端口创建了一个额外的引用，因此它必须增加对那个端口对象的引用数。当内核把消息拷贝到系统空间时创建新的引用，它对端口对象引用数的增加也是在此时完成的。同样，发送者可以在类型描述符中指定释放标志。这种情况下，内核释放在发送

者任务中的权力，并且不用增加端口引用数。

6.7.2 脱机内存

如果消息包含的是少量的数据，它们可以通过物理的拷贝数据来传输。首先是拷到内核的缓冲区内，然后（数据提取时）再拷到接收者任务的地址空间。然而，这种方法对于大量的数据传输花销是非常大的。因为 Mach 允许消息包含多到整个地址空间（在 32 位体系结构中多达 4G）的数据，因此它必须提供一种十分有效的办法来传输数据。

典型地，大量的数据传输时，大部分的数据可能永远不被发送者和接收者所修改。这种情况下，没有必要对数据再做一份拷贝。两个任务中只有一个任务想修改它时，才有必要对这个页做一份拷贝。在此之前，两个任务共享一个物理页的拷贝。Mach 是通过使用 Mach 的虚拟内存子系统的写拷贝共享机制来实现这个方法。第 15 章中详细地描述了 Mach 的内存管理。在这里，我们的讨论限制在和 IPC 有关的一些方面。

图 6-12 描述了脱机内存的传输、发送者在类型描述符中用一个标志位指定了脱机内存。例程 msg-copyin（由系统调用 msg_send 调用）修改发送者任务对这些页的映射力只读的和写拷贝的。接下来，在内核中创建一个暂时的“ bolding map ”。同时也把这些表项标志为只读和写拷贝的（图 6-12(a)）当接收者调用 msg_rcv 时，函数 msg_copyout( )在接收者的地址空间中分配一块地址范围，并把 holding map 中的表项拷到接收者的地址映射中。它标记新的项为只读的和写拷贝的，同时释放 holding map（图 6-12（b））。

图 6-12 传送脱机内存

此时，发送者和接收者以写拷贝方式共享这些页。两个任务中的任何一个想修改共享页时，它将引起一个页错误。这个页错误的处理程序能识别出这种情况，并通过映射该页的一份新的拷贝，并改变发生错误的任务重新映射这个拷贝来解决问题。它也改变保护这样两个任务都可以写它们的对这个页的拷贝（图 6-12(c)）。

注意到脱机内存的传输包括两个阶段——首先，消息被放置到队列中，同时页在传输中。一段时间后，提取消息时，这些页被发送者和接收者共享。由“ holding map ”来处理传输阶段。它保证在接收者提取消息之前发送者修改了这个页，内核会为发送者创建一份新的拷贝，这样接收者仍然能够继续访问最初的那份拷。

这种方法在发送者和接收者都不对共享的页修改的情况下工作得最好。它适合于许多种应用。即使这些页被修改的时候，这种方法也能保存操作的一份拷贝。联机内存被拷贝两次——一次是从发送者到内核，接着从内核到接收者。脱机内存最多拷贝一次，即在两个任务的一个在第一次想修改它时。

发送者可以在消息的类型描述符中设置释放标志。这种情况，内核不使用写拷贝共享．它只是简单地在 msg_copyin( )时把地址映射项拷贝到 holding map 中。同时把这些映射项从发送者的地址映射中释放。提取消息时，mng_copyout( )把这些项从 holding map 中拷贝到接收者的地址映射中，同时释放 holding map。这样，这些页就从发送者的地址空间移到接收者的地址空间，而不用任何数据拷贝。

6.7.3 控制流

消息传送可以两种方式进行——快速的和慢速的。消息发送的时候，接收者并没有在等待，在这种情况下，慢的方式比较适合。发送者把消息放入端口的队列中就返回了。接收者执行 msg_rcv( )时，内核把这个消息从端口的消息队列中取出，并把它拷贝到接收者的地址空间中。

端口有一个 backlog 参数的可设置的界限，可以用这个参数控制在端口的消息队列能

容纳的消息的最大数目。到达界限时，表示端口已满，新的发送者将会阻塞，等待消息从队列中提取出去。每次一个消息从有阻塞进程的端口提取后，那么内核会唤醒一个发送者。最后一个消息从那个端口中移走后，内核会唤醒所有的阻塞发送进程。

快速方式的情形是，接收者已经在等待这个消息。这种情况下，msg_send 就不把消息放入端口的消息队列中，而是直接唤醒接收者，并把消息交给它。Mach 提供了一种方法叫做 handoff 调度[Drav 91]，就是线程直接地把处理器让给另一个指定的线程。快速方式代码就是使用这种手法把控制切换到接收线程，接收线程调用 msg_rcv，使用 msg_copyout( ) 来把消息拷贝到自己的地址空间例程中。这就减少了把消息放入消息队列和从消息队列中再取出的开销。因为新的线程是直接选择来运行的，同时也加快了了上下文切换的速度。

6.7.4 通知

通知就是内核发出的异步消息，它告诉任务某些事件已经发生。内核发送这个消息到任务的通知端口。Mach IPC 使用三种类型的通知：

NOTIFY_PORT_DESTROYED 当一个端口被释放时这个消息发送到备份端口的所有者。端口的释放和备份端口将在下节讨论。

NOTIFY_PORT_DELETED 当一个端口被释放时，这个消息发送到对这个端口有发送权的任务。

NOTIFY_MSG_ACCEPTED 当向一个消息队列已满的端口发送消息时。发送者可以在一个消息从队列一移走时请求一个通知（使用一个 SEND_NOTIFY 选项）。

最后一种情况需要一些说明，当使用 SEND_NOTIFY 选项时，甚至在队列已满的情况下，消息仍然要被传送，内核返回一个 SEND_WILL_NOTIFY 状态，它要求发送者不再向队列发送更多的消息。直到它接收到一个 NOTIFY_MSG_ACCEPTED 通知。

6.8 端口操作

本节将讨论几个端口操作。

6.8.1 释放一个端口

一般情况下，端口所有者的任务中止时，端口的接收权被释放，这个端口也就被释放了。端口被释放后，一个 NOTIFY_PORT_DELETED 通知发送到对这个端口有发送权的所有住务。这个端口的消息队列中保存的所有消息都要被释放，阻塞的发送者和接收者都要被唤醒，并且分别收到一个 SEND_INVALD_PORT 和 RCV_INVALID_PORT 错误。

因为端口消息队列中的消息可能包含对其他端口的权力，这使得端口的释放变得很复杂。如果消息所包含的权力有接收权，那么相应的端口也要被释放。事实上，不怀好意的用户可在消息中把端口的接收权就发送给那个端口。虽然这种情况很少发生，但是当它们发中时，将引起一些死锁和无限的递归问题。这是 Mach 2.5 所不能很好解决的。

6.8.2 备份端口

系统调用 port_set_backup 为端口分配一个备份端口。当有备份端口的端口被释放时，内核并不释放这个端口；而是把端口的接收权转移给它的备份端口。

图 6-13 解释了这个过程。内核把端口 P2 分配为端口 P1 的备份端口。当端口 P1 释放时（可能是由于拥有它的任务 T1 的中止），内核就会发送一个 NOTIFY_PORT_DESTROYED 消息到端口 P2 的所有者。端口 P1 并不被释放，但是拥有对端口 P2 的发送权。所有发送到端口 P1 的消息自动地转发给端口 P2，并能被端口 P2 的所有者提取。

6.8.3 端口集合

端口集合（图 6-l4）由一组端口组成。这些端口的接收权由一个集宽（set_wide）接收权所取代。因此，单独的任务可以从这个集合接收消息。进一步地讲，它不能有选择地从集合中的单个端口接收消息。与之相对比的是，消息是发送到每个组成端口而不是端口集合。任务拥有对端口集合中特定端口的发送权，接收到消息时，这个消息包含它被发送的端口的信息。

服务器管理几个对象时，端口集合是十分有用的。服务器把每一个对象同一个端口联系起来，并把它们放入到端口集合中。服务器就可以接收到发送到端口集合中任何一个端口的消息了。每一个消息对端口所代表的对象请求一个操作。因为根据消息可以识别出它发往的端口，服务器就可以知道该去管理哪个对象。

端口集合的功能可以比作 UNIX 的系统调用 select 实现的功能，select 允许进程检查在几个描述符上的输入。一个重要的区别在于，从端口集合提取消息或是向端口集合中的端口发送信息所用时间和端口集合中的端口数目无关。

图 6-14 端口集合的实现

端口的内核对象包含指向它所属的端口集合的指针，也包括维护集合中所有端口的双向链表的指什。如果端口不是端口集合的成员，那么这个指针为空（NULL）。端口集合对象包含一个单独的消息队列，成员端口的消息队列没有被使用。但是每个端口保留它自己消息队列中消息的数目和积压参数（backlog），同时也保留阻塞的发送者队列。

线程向端口集合中的端口发送消息时，内核检查端口的消息数目是否超过积压参数（backlog）（如果已经超过，内核把这个发送者放入端口的阻塞发送者的队列中），并且增加这个端口的消息队列中消息的数目。内核在消息中记录端口的标志，并把这个消息放入端口集合的消息队列中。接收者任务中的线程在这个端口集合上调用 msg_rcv 时，内核提取在集合消息队列中的第一个消息，而不考虑这个消息是放在哪一个组成端口的队列中。

系统调用 port_set_allocate 和系统调用 port_set_deallocate 来实现端口集合的创建和撤销。端口集合中单个端口的插人和移出是通过系统调用 port_set_insert 和系统调用 port_set_remove 完成的。

6.8.4 端口的添加

端口插入允许任务用不同端口的权力替换另一个任务的权力，或者是从另一个任务获得权力。这种方法使得调试器（debugger）和模拟器（emulator）能够很好的控制目标任务。

图 6-15 描述了在调试器和目标任务之间一种可能的情形。首先，调试器使用 task_extract_send 调用把发送者的发送权移到它的 task_self 端口，从而使自己得到这个权力。接着它调用 task_insert_send 给端口 P1 插入一个发送权（P1 属于调试器），替换目标任务的 task_self 端口权力。同样地，调试器使用系统调用 task_extract_receive 和 task_insert_receive 把目标任务的接收权从它的 task_notify 端口去掉（consume），同时把接收权转到调试器有发送权的端口 P2。

图 6-15 调试器使用端口添加

完成这些处理以后，调试器就可以截获目标任务发往自己的 task_notify 端口（Mach 的系统调用）的任何消息。调试器处理这个系统调用，保证应答消息最终发往消息中指定的应答端口。调试器可以选择模拟调用，并发送应答消息。另外，它也能使用目标任务最初的

task_notify 端口（调试器对其有发送权）来向内核转发消息。发送的消息到达内核时，它可以直接把应答消息送到目标任务的应答端口或是指定它自己的应答端口，这样就能截获内核的应答。

同样地，调试器截获发往任务的任何通知（notification）并决定是代替目标任务来执行处理还是把它们转发给目标任务。因此，只要调试器（或其他的任务）对任务的 task_self 端口有发送权，它就可以控制目标任务的任何端口。

6.9 扩 展 性

Mach IPC 的设计可以透明地扩展到分布式的环境中。用户级程序网络消息服务器（netmsgserver）能把 Mach IPC 扩展到整个网络。这样用户可以像和本地任务通信那样很容易和在远程机器上的任务进行通信。应用程序并不知道远程连接的存在，只是继续使用那种和本地通信所用的相同的接口和系统调用。应用程序通常并不知道它是和本地任务还是和远程任务进行通信。

下面有两个原因解释了为什么 Mach 能够提供这样透明的扩展性。首先，端口权力提供了与位置无关的名字空间。发送者只是使用端口的本地名字（发送权）来发送消息，它不必知道这个端口代表的是本地对象还是远程对象，内核维护任务的端口本地名字和端口的内核对象之间的映射。

其次，发送者是匿名的。通过消息并不能识别出发送者。发送者可以在消息中传递发送权给一个应答端口。内核对这进行变换，因此接收者看到的只是这个发送权的本地名字。而不知道谁是发送者。进一步的，发送者不必拥有应答端口，它只需对应答端口有发送权就可以。通过指定属于另一个任务的应答端口，发送者就可把应答消息发送给另一个不同的任务，这对于调试器、模拟器等是十分有用的。

网络消息服务器的操作是简单的。图 6-16 给出一个典型的情形。网络上的每一台机器上都运行网络消息服务器程序。如果节点 A 上的一个客户想要和节点 B 的服务器通信时，网络消息服务器在节点 A 上建立一个代理端口，客户能够向其发送消息。接下来，网络消息服务器提取发送到那个端口的消息并把它转发到节点 B 上的网络消息服务器，它将把消息再转发给服务器的端口。

如果客户希望一个应答消息，它就会在消息中指定应答端口。节点 A 上的网络消息服务器保存对应答端口的发送权。节点 B 上的网络消息服务器为应答端口创建一个代理端口，并且把对代理端口的权力发送给服务器。服务器应答代理端口；这个应答消息经由这两个网络消息服务器传送到客户的应答端口。

服务器在本地的网络消息服务器上注册它们自己，同时传递一个发送权到服务器监听的端口。网络消息服务器维护一个分布式数据，记录这种注册的网络端口，并且给这些端口提供与内核给本地端口提供的服务相同（保护，通知等等）。因而，网络消息服务器互相查询来提供一个全局名字查找服务。任务使用这种服务来获得对在远程网络消息服务器注册的端口的发送权。在没有网络的情况下，它将变为简单的本地的名字服务。

网络消息服务器通过使用低层网络协议来互相通信，而不是通过 IPC 消息。

图 6-16 使用 netmsgservers 实现远程通信

6.10 Mach 3.0 的改进

Mach 3.0 对 IPC 子系统引进厂一些改进[Dray 90]，并且解决了在 Mach 2.5 实现中存在的一些严重问题。这些改进对接口、内部数据结构和算法都有影响。本节将深入研究一些重大的改进。

Mach 2.5 的一个主要的问题是关于发送权的。IPC 主要用于客户-服务器交互。典型地，客户发送的消息包含对客户拥有的应答端口的发送权。当服务器发送完应答消息后，这个发送权就不再需要。但是，服务器不能释放这个权力，这是因为，由于另一个独立消息的存在，服务器中的另一个线程也许已经接收到或正在使用对同一个端口的发送权作为分离消息的结果。内核把第二个权力变换为相同的本地名字，因此服务器对这个端口只有一个发送权，如果第一个线程释放了这个权力，第二个线程就不能发送其应答。

可以永远不释放这个发送权，服务器通常工作在这种假定下。这将产生另一个问题。它是在安全方面的冒险——客户可能希望服务器不要永久地持有这个权力。同时，也不需要用尽所有的变换表项，因为这将影响整个系统的性能。最后，客户释放掉端口的时候，将对服务器产生不必要的通知。内核必须发送一个通知给所有对端口持有发送权的服务器，服务器必须接收并处理这些通知，尽管可能有些服务器对端口没有任何意义。

Mach 3.0 有三个单独的改进，它们解决这个普遍的问题——一次发送权，通知请求和对发送权的用户引用。

6.10.l 一次发送权

对端口的一次发送权，就像它的名字所示，这个发送权只能被使用一次。这个权力是由对端口有接收权的任务产生的，但是它可以从一个任务传递到另一个任务。它保证从中产生一个消息。通常地，一些发送权被使用是通过在应答消息中把它当作目的端口；当消息被接收后，一次发送权被释放。如果这个权力是以另外的方式释放的，比如拥有这个权力的任务突然中止，那么内核向这个端口发送一个一次发送通知。

内核对一次发送权和发送权分开来维护。因此如果一个任务对相同的一个端口请求获得发送权和一次发送权时，它将有两个不同的本地名字。一个住务也可以请求对同一个端口的一次发送权。与发送权不同的是每一个一次发送权都有一个唯一的名字。端口内核对象的引用计数只是反映突出的（outstanding）的发送权，它并不受一次发送权的影响。

Mach 3.0 使用一次发送权来指定应答端口。当应答消息接收后，这个权力就会被释放。因此服务器并不是永久地保存它。这样就消除了当客户释放一个端口时而产生的不必要的通知。

6.10.2 Mach 3.0 的通知

Mach 2.5 异步地发送通知来响应不同的系统事件。任务无法控制它们收到哪一个通知。通常地，它们不处理这些事件，而是简单地抛弃它们。大量不必要的通知将降低整个系统的性能。

进一步地说，一个单独的任务范围的通知端口太有限了。线程可以截获和释放一个通知，而在任务中的另一个线程可能正在等待这个通知。用户的库函数增加了这种情况的可能性，因为主程序和库函数独立地使用不同的通知。

Mach 3.0 中，通知只是发向那些显式地使用 mach_port_requet_noification 调用请求它们的那些任务。这些请求对于通知端口指定了一次发送权。Mach 3.0 因而允许在一个任务中有几个通知端口。每一个程序组成部分或每一个线程都能够分配自己的通知端口。从而避免了竞争。

6.10.3 发送权的用户引用记数

如果任务获得对端口的多个发送权，内核就会把它们映射为任务端口名字空间中的单个本地名字，因此把它们合成为单个发送权。如果一个线程释放了这个权力，其他的线程就都不能使用它了．尽管其他线程已经独立地获得了这个权力。

尽管一次发送权消灭了这类问题的根源，但是它不能完全地解决问题。一次发送权主要是给服务器提供应答端口。发送权可以通过不同的方式来获得。如果客户想和服务器通信，客户获得（通过名字服务器）对它的发送权。客户中的每个线程独立地开始和服务器联系，它们中的每一个都将获得一个发送权。这些权力合成为一个名字。如果其中一个线程释放这个名字，这将对所有其他的线程产生影响。

Mach 3.0 通过把每个发送权同一个用户引用联系起来解决这个问题。这样，在以前的例子中，任务在每次获得相同的发送权时内核增加引用计数，而在每次线程释放这个权力时，减少引用计数。当最后一个引用被释放后，内核可以安全地释放这个权力。

6.11 讨 论

Mach 使用 IPC 不仅是来进行进程间的通信，而且把它作为一个基本的内核结构原语。虚拟内存子系统使用 IPC 来实现写拷贝[Youn 87]。内核使用 IPC 来控制任务和线程。Mach 中的最基本抽象，如任务、线程和端口，通过消息传递来交互。

这种体系结构提供了一些吸引人的功能。比如，网络消息服务器透明地把 IPC 扩展到一个分布式系统中。这样，任务可以控制和操作远程节点上的对象。这样就允许 Mach 提供诸如远程调试，分布式共享内存和其他的客户用务器程序等方法。

同样，大量地使用消息传递会使系统的性能下降。有一段时间人们对于构造微内核操作系统有很大的兴趣，其中大部分的方法都由使用 IPC 的用户级服务器任务间的相互通信来提供。尽管许多供应商仍然在这种解决方案上努力工作，在考虑到性能原因时，这种方法已经走向了非主流。

Mach 的支持者强烈认为 IPC 性能在设计微内核操作系统时不是一个重要的因素[Bers92]。原因如下：

- 对 IPC 性能的改善要远远大于对操作系统其他方面的改善。
- 随着硬件高速缓存（cache）可靠性的增加，操作系统服务的代价将由高速缓存的命中模式所支配。因为 IPC 代码是高度本地化的，它可以很容易的转向优化使用高速缓存（cache）。
- 数据传输可以使用如共享内存等机制来完成。
- 把一些内核功能转移到用户级服务器，减少了内核态和用户态切换的次数以及减少了保护边界的越界，而这些操作的开销都是很大的。

研究者已经对改善 IPC 性能给了极大的重视[Bers 90，Bars 91]。但是到目前为止，Mach IPC 对商用系统的影响还是十分有限的。尽管 Digital UNIX 是基于 Mach 的，但是它的许多内核子系统都不使用 Mach 的 IPC。

6.12 小 结

本章描述了一些 IPC 机制。信号，管道和跟踪（ptrace）是十分普遍的方法。这些机制在早期的所有的 UNIX 系统中都存在。Sytem 5 的 IPC 机制由共享内存、信号量和消息队列组成，普遍存在于现代的 UNIX 变体中。在 Mach 的内核中，所有的对象使用 IPC 互相操作。Mach IPC 通过网络消息服务器可扩展，允许开发分布式的客户-服务器程序。

其他一些 IPC 机制在本书的其他部分有所介绍——第 8 章中的文件锁，第 14 章的内存映射文件以及第 17 章的流管道。

6.13 练 习

1．使用工具 prace 来编写调试器有哪些局限？

2．Ptrace 的参数 pid 必须是调用者的子进程的进程 ID。放松这些要求暗示着什么？为

什么进程不能够使用 ptrace 与任意一个进程进行交互？

3．比较管道和消息队列提供的 IPC 功能。它们各自的优缺点都有哪些？在什么情况下其中一个比另一个更合适？

4．大多数 UNIX 系统允许进程将同一共享内存区映射到地址空间中的多处。这是一个错误，还是一个特殊功能？什么时候这个功能有用了它又会引起哪些问题？

5．在选择映射共享内存地址时，程序员必须注意哪些问题？操作系统必须要防止哪些错误？

6．如何在使用信号量时利用 IPC_NOWAIT 来避免死锁的发生？

7．分别使用(a)FIFO 文件，(b)信号量，(c)mkdir 系统调用和(d)flock 或 lockf 系统调用使互相协作的进程可以锁定专用资源。比较它们的性能并解释之。

8．对于上面的程序，程序员必须要注意哪些副作用？

9．是否可以通过(a)只使用信号或(b)共享内存和信号实现资源锁？这种方法的性能如何？

10．写个程序使用（a）管道，（b）FIFO，（c）消息队列和（d）共享内存并用信号量同步机制完成两个进程间的大量数据的传送。比较它们的性能并解释之。

11．与 System V IPC 机制相关的安全问题是什么？一个恶意的程序如何窥听或干扰其他进程间的通信？

12．通过 semget 创建信号量，但要通过 semctl 进行初始化。由于创建和初始化不在一步中原子地完成。试举一例，因为这一情况而导致竞争，并给出一种解决方案。

13．是否可以在 Mach IPC 的基础上实现 System V 的消息队列？这一实现必须解决什么问题。

14．为什么一次发送权非常有用？

15．在 Mach 中，如何利用端口集合辅助开发客户-服务器应用？

6.14 参考文献

[Bach 86] Buch，M.J.，The Design of the UNIX Operating System，PrenticeHall，Englewood Clieffs, NJ, 1986．

[Baro 90] Baron，R.V.，Black，D.，Bolosky，W.，Chew，J.，Draves，R．P.，Golub，D.B.，Rashid，R.F., Tevanian，A.，Jr.，and Young，M.W.，Mach Kernel Interface Manual，Department of Computer Science，Carnegie Mellon University, Jan. 1990．

[Barr 91] Barrera，J.S., III，“A Fast Mach Network IPC Implementation”，Proceedings of the USENIX Mach Symposium，Nov. 1991，PP. 1-12．

[Bers 90] Berahad，B.N.，Anderson，T.E.，Lazowska，E.D.，and Levy,H.M.，“Lightweight Remote Procedure Call”，ACM Transactions on Comuter Systems，Vol．8，No.2 Feb，1990，PP．37-55．

[Bers 92] Bershad，B.N.，“The lncreasing Irrelevance of IPC Perrformance for ImcrokerndBased Operation Systems”，USENIX workhop on Micro Kernels andother Kernel Architectures，Apr．1992，pp．205-212．

[Dijk 65] Dijkstra, E.W.，“Solution of a Problem in Concurrent Programming Control”，Communications of the ACM，Vol．8，Sep．1965，pp．569－578．

[Drav 90] Draves，R．P.，“A Revised IPC Interface，”Proceedings of the First Mach USENIX Workshop，Oct．1990，pp．101－121．

[Dray 91] Draves，R．P．Bershad，B．N，Rashid，R．F，and Dean，R，W，“Using Continuations to Implment Thread Management and Commuication in Operating Systems”，

Technical Report CMU-CS-91-115R．School of Computer Science，Carnegie Mellon University，Oct．1991．

[Faul 91] Faulkner，R．and Gcmes，R，“The Proce的File System and Proce的Model in UNIX System V，”Proceedings of the 1991 WinterUSENIX Conference，Jan，1991，pp，243－252．

[Pres90] Presotto，D，L，and Ritchie，D．M，“Interproe的Communications inthe Ninth Edition UNIX System”，UNIX Research System Papers，Tenth Edition，Vol．11，Saunders College Publishing，1990，pp．523－530．

[Rash 86] Rashid，R．F，“Threads of a New System”UNIX Review，Aug，1986，pp．37－49．

[Salu 94] Salu，P．H，A Quarter Century Of UNIX，Addison-Wesley，Reading，MA，1994．

[Stevens 90] Stevens，R．W，UNIX Network Programming，Prentice-Hall，Englewood CliffS，NJ，1990．

[Thom 98] Thompson，K，“UNIX lmplementation，”The Bell System Technical Journal，Vol，57，No．6，Part 2，Jul-Aug，1978，PP，131－1946．

[Youn 87] Young，M，Tevanian，A，Rashid，R．F，Golub，D，Eppinger J，Chew，J，Boloshy, W．Black，D，and Baron，R，“The Duality of Memery and Communication in the Implementation of a Multiproce的or Operating System” Proceedings of the Eleventh ACM Symposium on Operating Systems Principles，Nov．1987，pp 63－76．

注释

① 最早的贝尔实验室里的 UNIX 系统并没有这些功能。例如，管道是由 Doug Mcllroy 和 Ken Thompson 研制的，并于 1973 年出现在版本 3 的 UMX 系统中［Salu 94］。

②由内核产生的响应负责硬件异常的信号通过传递给 siginfo 结构返回给处理函数一些额外的信息。

③像 SVR2 这样的传统 UNIX 系统在文件系统中实现管道并直接使用 i 节点中直接块地址域来定位管道的数据块。这种实现就限制了管道的尺寸只能达到 10 个块。较新的 UNIX 系统虽然使用了不同的方法实现管道，但依旧保持了这个限制。

④合作的应用可以彼此间达成一个协议，在每个对象中保存分组边界。

⑤合作的应用同样可以彼此间达成一个协议，为每个对象标上数据源。

⑥这一新功能只适用于管道。SVR4 的 FIFO 与传统的 FIFO 很相似。

⑦否则调用者必须是一个特权进程。

⑧如果键值是一个特殊值 IPC_PRIVATE，内核就创建新资源。这个资源不能通过其他 get 调用进行访问（于每次内核都要产生一个新资源），因而调用者对它有独占权。通过 fork 系统调用子进程继承这些资源，属主可以同它的子进程共享资源。

⑨名称 P( )、V( )源自这些操作的荷兰语。

⑩在多处理器中．还要有附加的操作来保证 cache 的一致性。15.13 节将讨论这些问题。

⑾某些操作系统，诸如 Window/NT 和 OS/2。使用路径名来命名共享内存对象。

⑿早期的 Mach 有独立的属主权和接收权。Mach 2.5 和更新的发行用后备端口替代了属主权，这些内容将在 6.8. 2 节中介绍。