

网络程序设计系列丛书

# Unix网络编程

## 实用技术与实例分析

张 炯 编著  
潇湘工作室 审校

清 华 大 学 出 版 社

**(京)新登字 158 号**

## 内 容 简 介

本书详细介绍了在 Unix 环境下网络编程的方法,全书分为四部分:第一部分“网络基础”主要讲述 TCP/IP 协议簇,尤其是与编程相关的部分,并说明了网络编程环境;第二部分“套接字”是网络编程的核心,在此通过讲解套接字库函数、TCP 套接字、UDP 套接字及相应的实例,使读者能够编写基本的网络程序;第三部分“Unix 网络编程实用技术”是本书的重点,讲述 Unix 网络开发过程中常用的技术,如并发服务器技术、名字和 IP 地址转换、同步及进程间通信技术、异常处理技术、实用套接字类库的创建,说明如何提高软件的性能、可靠性和可扩充性,并配有大量实例予以说明;第四部分“高级网络编程”主要涉及底层 IP 编程技术,可用于路由器、网络监视器及专用协议的开发,介绍了守护进程、原始套接字、数据链路访问、多接口捆绑及路由套接字技术。

本书涉及的内容包括 Unix 系统、网络协议及编程技术,并由浅入深地讲述了网络编程核心技术、实用技术和高级网络编程。本书既是从事网络开发人员的参考资料,也可以作为学习 Unix 网络编程知识的教材。

**版权所有,翻印必究。**

**本书封面贴有清华大学出版社激光防伪标签,无标签者不得销售。**

## 图书在版编目(CIP)数据

Unix 网络编程实用技术与实例分析/张炯编著. —北京:清华大学出版社,2002.11

(网络程序设计系列丛书)

ISBN 7-302-05891-1

I. U... II. 张... III. Unix 操作系统 — 程序设计 IV. TP316.81

中国版本图书馆 CIP 数据核字(2002)第 070811 号

**出 版 者:**清华大学出版社(北京清华大学学研大厦,邮编:100084)

<http://www.tup.tsinghua.edu.cn>

**责任编辑:**龙啟铭

**印 刷 者:**印刷厂

**发 行 者:**新华书店总店北京发行所

**开 本:**787×1092 1/16 **印张:**22.5 **字数:**547 千字

**版 次:**2002 年 11 月第 1 版 2002 年 11 月第 1 次印刷

**书 号:**ISBN 7-302-05891-1/TP·3497

**印 数:**0001~4000

**定 价:**32.00 元

# 前 言

网络编程是指编写网络通信程序，以完成网络上不同程序间的通信。网络程序分为服务器和客户端两部分。客户端发送请求给服务器，服务器处理客户请求并将结果发回客户端。网络编程是通过调用网络 API 实现的。目前有多种网络 API，其中套接字最为流行。本书讲述的网络编程就是以套接字 API 为基础的。

套接字的发展与 Unix 操作系统的发展紧密相关。Unix 是 20 世纪 70 年代由 AT&T 的 Bell 实验室开发的。从 Unix 的发展历史来看，主要有两大流派：AT&T 的 Unix 系统 V 版本和加州大学伯克利分校的 BSD 版本。套接字 API 最初是随 4.2 版的 BSD 系统于 1983 年发布的，最终在多数 Unix 系统上获得了支持。

网络编程涉及的范围极广，例如所支持的通信协议(TCP/IP、NetBEUI、IPX/SPX 等)、运行的操作系统(Unix、Windows 等)、网络 API(套接字 API、XTI 等)和所用的编程语言(C/C++、Java 等)。国际互联网无疑是网络应用的主体，其采用的通信协议 TCP/IP 协议簇被几乎所有的操作系统所支持并最为广泛地使用，因此，本书的网络协议主要是 TCP/IP 协议簇。对于操作系统，Unix 和 Windows 都非常流行，但由于 Unix 系统的可靠性、安全性及开放性，大量的网络应用是在 Unix 系统下开发的，尤其是服务器部分。例如，在浏览网页时，所用的浏览器多数是在 Windows 系统下运行的，而 Web 服务器则主要在 Unix 下运行。考虑到服务器是决定网络应用系统性能的关键，另外随着 Unix 系统微机版的推广，越来越多的程序员转向 Unix 系统开发，因而本书选择 Unix 系统。Windows 也支持套接字 API（Windows 套接字），所以在 Unix 下开发的网络程序仅需作少量的修改便可在 Windows 下运行。Unix 操作系统是用 C 语言研制开发的。套接字 API 也是以 C 函数形式提供的，因此本书的实例都是用 C/C++编写的。

## 本书的特点

### 大量的实例和代码分析

本书以实用为出发点介绍网络编程，并配有大量的实例和代码分析。这些代码均为作者长期工作经验的积累，非常具有实用价值。

为了方便读者，在介绍各种技术的同时提供了大量实例并进行了详细的分析，不仅分析源代码，还针对程序的运行结果进行说明。

### 体贴读者，详细讲解网络开发难点

本书涉及的技术都是在网络开发过程中常用的技术，由于网络应用的性能及可靠性是网络开发中的主要难点，本书重点对这方面的技术进行了深入的讲解说明。

为了增强实用性，本书介绍了大量实用性强的实例，并且部分实例采用面向对象方法进行封装，便于读者使用、扩充和移植。

在叙述方面，对于仅需了解的知识讲解力求简明，而对于关键技术则深入透彻。尤其是通过实例对比的方式，使读者能体会到每种技术所起到的作用和产生的问题。另外，本

书介绍了所涉及技术的应用背景，使得读者在实际网络编程中能够知道采用哪种技术解决相应问题。

### 作者的体会与经验之谈

网络编程是非常复杂的，需要长期的学习和积累，关于这一点本人深有体会。在开始学习网络编程时，觉得非常简单，找个例子简单修改一下就可以运行了，甚至连网络方面的书籍也不必看。

然而，在开始开发实际项目时，才发现问题层出不穷，每当费了九牛二虎之力解决一个问题后，又产生了新的问题，更糟糕的是经常出现“无故故障”。这时才开始较系统地学习网络协议及相关编程技术，利用相关的网络知识找出这些“无故故障”的原因，并通过采用相应技术提高网络应用系统的性能和可靠性。当看到自己开发的系统高效可靠地运行时，真是有苦尽甘来的感觉。相信读者在学习本书的过程中也会有相同的体会。

在学完本书第二部分时，读者就可以编写网络程序了。然而，由于实际网络应用的高并发性，这些程序会变得毫无用处。当采用并发技术后，又带来同步等新问题。当学完全书后，相信读者将在 Unix 网络编程方面游刃有余。

### 本书结构

在结构安排上，本书遵循由浅入深的原则，非常适于没有网络开发经验的人员学习。本书共分四部分：

- 第一部分“网络基础”主要讲述 TCP/IP 协议簇，尤其是与编程相关的部分，并且对网络编程环境进行了介绍。这部分是网络编程所应具备的基本知识。
- 第二部分“套接字”是网络编程的核心。本部分通过讲解套接字库函数、TCP 套接字、UDP 套接字及相应实例，使读者能够编写基本的网络程序。
- 第三部分“Unix 网络编程实用技术”是本书的重点。讲述 Unix 网络开发过程常用的技术，以提高软件的性能、可靠性和可扩充性，并配有大量实例予以说明。该部分共分 5 章：并发服务器技术、名字和 IP 地址转换、同步及进程间通信技术、异常处理技术、实用套接字类库的创建。
- 第四部分“高级网络编程”主要涉及底层 IP 编程技术，可用于路由器，网络监视器及专用协议的开发，包括守护进程、原始套接字、数据链路访问、多接口捆绑及路由套接字技术。

### 读者对象

本书的目标是使读者学完本书后能够开发实用的网络程序，本书有大量的实例，因此读者应具备 C/C++ 编程及 Unix 系统的使用经验。另外，本书系统地讲述了 Unix 网络编程的主要实用技术，也适用于从事网络开发人员参考之用。

### 联系我们

本书由潇湘工作室策划和组织编写，张炯主笔。该作者自 1991 年计算机系毕业后，一直从事网络开发及研究工作，对 Unix 系统下的网络应用级及系统级的开发具有丰富的经验，并对网络协议和网络互连有深入的研究。

在本书的编写过程中，得到了许多人的热情指导及多方面的帮助，在此一并表示诚挚

的感谢！

读者在学习本书的过程中，若发现本书有疏漏之处和错误，或者，如果您有好的想法和建议，亦或是您需要本书程序的源代码，均请与我们联系，我们将竭尽所能提供帮助，并不断改进工作，为读者奉献高品质的好书。

我们的电子邮件地址为：xiaoxiang-007@sohu.com

# 目 录

## 第一部分 网络基础

|       |                    |    |
|-------|--------------------|----|
| 第 1 章 | Unix 系统基础 .....    | 1  |
| 1.1   | Unix 系统概述 .....    | 1  |
| 1.1.1 | Unix 系统的历史 .....   | 1  |
| 1.1.2 | Unix 系统的特点 .....   | 1  |
| 1.1.3 | Unix 系统的体系结构 ..... | 1  |
| 1.1.4 | Unix 系统的地址空间 ..... | 2  |
| 1.1.5 | POSIX 标准 .....     | 2  |
| 1.2   | 常用 Unix 网络命令 ..... | 2  |
| 1.2.1 | ping .....         | 2  |
| 1.2.2 | netstat .....      | 3  |
| 1.2.3 | ifconfig .....     | 3  |
| 1.2.4 | route .....        | 4  |
| 1.2.5 | tcpdump .....      | 4  |
| 1.3   | 网络基本配置文件 .....     | 5  |
| 1.4   | 软件开发环境 .....       | 5  |
| 1.4.1 | vi 编辑器 .....       | 5  |
| 1.4.2 | gcc 编译器 .....      | 7  |
| 1.4.3 | gdb 调试器 .....      | 7  |
| 1.5   | 简单实例 .....         | 8  |
| 1.5.1 | 源程序分析 .....        | 8  |
| 1.5.2 | 实现过程 .....         | 10 |
| 1.6   | 小结 .....           | 12 |
| 第 2 章 | TCP/IP .....       | 13 |
| 2.1   | TCP/IP 体系 .....    | 13 |
| 2.2   | IP 协议 .....        | 14 |
| 2.2.1 | IP 包的结构 .....      | 14 |
| 2.2.2 | IP 地址组成 .....      | 15 |
| 2.2.3 | IP 地址表示 .....      | 15 |
| 2.2.4 | IP 地址类型 .....      | 15 |
| 2.2.5 | 子网掩码 .....         | 16 |
| 2.3   | TCP 协议 .....       | 16 |

|       |                 |    |
|-------|-----------------|----|
| 2.3.1 | 建立 TCP 连接 ..... | 16 |
| 2.3.2 | 关闭 TCP 连接 ..... | 16 |
| 2.3.3 | TCP 数据包结构 ..... | 17 |
| 2.4   | UDP 协议 .....    | 17 |
| 2.5   | ICMP 协议 .....   | 18 |
| 2.6   | 端口号分配 .....     | 19 |
| 2.6.1 | 端口分类 .....      | 19 |
| 2.6.2 | 常用端口号 .....     | 19 |
| 2.7   | IP 路由 .....     | 20 |
| 2.7.1 | 路由表分类 .....     | 20 |
| 2.7.2 | IP 路由过程 .....   | 20 |
| 2.8   | 小结 .....        | 21 |

## 第二部分 套 接 字

|       |                                |    |
|-------|--------------------------------|----|
| 第 3 章 | 套接字基础 .....                    | 22 |
| 3.1   | 套接字概述 .....                    | 22 |
| 3.2   | 套接字类型 .....                    | 23 |
| 3.3   | 套接字地址结构 .....                  | 23 |
| 3.3.1 | INET 协议簇地址结构 sockaddr_in ..... | 23 |
| 3.3.2 | 存储地址和端口信息的 sockaddr .....      | 24 |
| 3.3.3 | 32 位 IPv4 地址结构 in_addr .....   | 24 |
| 3.4   | 端口 .....                       | 25 |
| 3.5   | 带外数据 .....                     | 26 |
| 3.6   | 连接类型 .....                     | 26 |
| 3.7   | 小结 .....                       | 27 |
| 第 4 章 | TCP 套接字 .....                  | 28 |
| 4.1   | 基本方法 .....                     | 28 |
| 4.1.1 | TCP 套接字实现过程 .....              | 28 |
| 4.1.2 | TCP 服务器模板 .....                | 29 |
| 4.1.3 | TCP 客户模板 .....                 | 30 |
| 4.2   | 实现 TCP 套接字 .....               | 31 |
| 4.2.1 | 产生 TCP 套接字 .....               | 31 |
| 4.2.2 | 绑定 .....                       | 32 |
| 4.2.3 | 监听 .....                       | 34 |
| 4.2.4 | 接受请求 .....                     | 35 |
| 4.2.5 | 连接建立 .....                     | 36 |
| 4.2.6 | 数据传输 .....                     | 38 |

|  |           |
|--|-----------|
| 4.2.7 终止连接 .....                       | 39        |
| 4.3 TCP 套接字编程实例 .....                  | 40        |
| 4.3.1 实例说明 .....                       | 40        |
| 4.3.2 TCP 服务器 .....                    | 40        |
| 4.3.3 TCP 客户 .....                     | 42        |
| 4.3.4 运行程序 .....                       | 44        |
| 4.4 小结 .....                           | 45        |
| <b>第 5 章 UDP 套接字 .....</b>             | <b>46</b> |
| 5.1 基本方法 .....                         | 46        |
| 5.1.1 UDP 套接字实现过程 .....                | 46        |
| 5.1.2 UDP 服务器模板 .....                  | 47        |
| 5.1.3 UDP 客户模板 .....                   | 48        |
| 5.2 函数说明 .....                         | 49        |
| 5.2.1 UDP 套接字的数据发送——sendto()函数 .....   | 49        |
| 5.2.2 UDP 套接字的数据接收——recvfrom()函数 ..... | 49        |
| 5.3 UDP 套接字编程实例 .....                  | 51        |
| 5.3.1 UDP 服务器 .....                    | 51        |
| 5.3.2 UDP 客户 .....                     | 53        |
| 5.3.3 运行程序 .....                       | 55        |
| 5.4 小结 .....                           | 55        |

### 第三部分 Unix 网络编程实用技术

|                           |           |
|---------------------------|-----------|
| <b>第 6 章 并发服务器 .....</b>  | <b>57</b> |
| 6.1 并发服务器基础 .....         | 57        |
| 6.1.1 服务器分类 .....         | 57        |
| 6.1.2 重复性服务器实例 .....      | 58        |
| 6.1.3 并发技术 .....          | 64        |
| 6.1.4 并发服务器算法 .....       | 64        |
| 6.2 多进程服务器 .....          | 68        |
| 6.2.1 进程概念 .....          | 68        |
| 6.2.2 创建进程 .....          | 68        |
| 6.2.3 终止进程 .....          | 70        |
| 6.2.4 多进程并发服务器 .....      | 73        |
| 6.2.5 多进程并发服务器实例 .....    | 77        |
| 6.3 多线程服务器 .....          | 83        |
| 6.3.1 线程基础 .....          | 84        |
| 6.3.2 线程函数调用(POSIX) ..... | 84        |



|              |  |            |
|--------------|--|------------|
| 6.3.3        | 多线程并发服务器 .....                         | 87         |
| 6.3.4        | 给新线程传递参数 .....                         | 88         |
| 6.3.5        | 多线程并发服务器实例.....                        | 92         |
| 6.3.6        | 线程安全 (MT-safe) 实例.....                 | 96         |
| 6.4          | I/O 多路复用服务器 .....                      | 112        |
| 6.4.1        | I/O 模式 .....                           | 113        |
| 6.4.2        | select() 函数.....                       | 114        |
| 6.4.3        | 单线程并发服务器实例.....                        | 116        |
| 6.5          | 套接字终止处理.....                           | 123        |
| 6.6          | 小结 .....                               | 124        |
| <b>第 7 章</b> | <b>名字和 IP 地址转换 .....</b>               | <b>126</b> |
| 7.1          | 名字解析 .....                             | 126        |
| 7.2          | 套接字地址.....                             | 126        |
| 7.2.1        | 地址结构 .....                             | 126        |
| 7.2.2        | 字节顺序 .....                             | 127        |
| 7.2.3        | IP 地址转换函数.....                         | 128        |
| 7.2.4        | 套接字地址信息函数 .....                        | 129        |
| 7.3          | 套接字信息函数.....                           | 130        |
| 7.3.1        | 主机名转换为 IP 地址: gethostbyname() 函数 ..... | 130        |
| 7.3.2        | IP 地址转换为主机名: gethostbyaddr() 函数.....   | 132        |
| 7.3.3        | 获得服务的端口号: getservbyname() 函数.....      | 134        |
| 7.3.4        | 端口号转换为服务名: getservbyport() 函数 .....    | 135        |
| 7.4          | 小结 .....                               | 135        |
| <b>第 8 章</b> | <b>同步及进程间通信 .....</b>                  | <b>136</b> |
| 8.1          | 线程同步 .....                             | 136        |
| 8.1.1        | 线程同步基础 .....                           | 136        |
| 8.1.2        | 互斥锁基础 .....                            | 136        |
| 8.1.3        | 加锁和解锁互斥锁 .....                         | 138        |
| 8.1.4        | 条件变量 .....                             | 142        |
| 8.1.5        | 同步线程退出 .....                           | 151        |
| 8.1.6        | 死锁 .....                               | 158        |
| 8.2          | 进程同步 .....                             | 165        |
| 8.2.1        | 进程关系 .....                             | 165        |
| 8.2.2        | 信号处理 .....                             | 167        |
| 8.2.3        | 处理僵死进程 .....                           | 171        |
| 8.3          | 进程间通信.....                             | 175        |
| 8.3.1        | 管道 .....                               | 176        |

|        |                                      |     |
|--------|--------------------------------------|-----|
| 8.3.2  | FIFO.....                            | 176 |
| 8.3.3  | 消息队列 .....                           | 180 |
| 8.3.4  | 共享内存 .....                           | 180 |
| 8.3.5  | 信号量 .....                            | 181 |
| 8.4    | 小结 .....                             | 182 |
| 第 9 章  | 异常处理 .....                           | 183 |
| 9.1    | 异常处理基础.....                          | 183 |
| 9.2    | 函数调用的错误处理.....                       | 183 |
| 9.2.1  | 显示错误信息 .....                         | 184 |
| 9.2.2  | 定义错误处理函数 .....                       | 187 |
| 9.3    | I/O 超时处理 .....                       | 188 |
| 9.3.1  | 使用 alarm()函数.....                    | 188 |
| 9.3.2  | 使用 select 函数 .....                   | 189 |
| 9.4    | 服务器异常处理.....                         | 190 |
| 9.4.1  | 异常处理的系统调用 .....                      | 190 |
| 9.4.2  | 服务器异常处理实例 .....                      | 191 |
| 9.5    | 客户异常处理.....                          | 196 |
| 9.6    | 小结 .....                             | 196 |
| 第 10 章 | 创建实用套接字类库 .....                      | 197 |
| 10.1   | 创建静态链接库.....                         | 197 |
| 10.1.1 | 创建库文件 .....                          | 197 |
| 10.1.2 | 建立库文件索引 .....                        | 197 |
| 10.1.3 | 连接库文件 .....                          | 197 |
| 10.2   | 创建动态链接库.....                         | 198 |
| 10.2.1 | 创建库文件 .....                          | 198 |
| 10.2.2 | 使用动态链接库 .....                        | 198 |
| 10.2.3 | 相互引用的库文件 .....                       | 199 |
| 10.2.4 | 动态库与静态库并存 .....                      | 199 |
| 10.3   | 创建自定义的套接字类库.....                     | 199 |
| 10.3.1 | 设计套接字类库 .....                        | 199 |
| 10.3.2 | 套接字系统调用: MySocket 类.....             | 201 |
| 10.3.3 | 多线程实现: MyThread 类 .....              | 202 |
| 10.3.4 | 加锁/解锁: MyMutex 类和 MyCondition 类..... | 205 |
| 10.3.5 | 基于 TCP 的多线程并发服务器: TcpServThr 类.....  | 209 |
| 10.3.6 | TCP 多线程客户类: TcpCliThr 类 .....        | 214 |
| 10.4   | 实例分析.....                            | 218 |
| 10.4.1 | 实现聊天室服务器 .....                       | 218 |

|        |               |     |
|--------|---------------|-----|
| 10.4.2 | 实现聊天室客户 ..... | 225 |
| 10.4.3 | 运行程序 .....    | 227 |
| 10.5   | 小结 .....      | 230 |

## 第四部分 高级网络编程技术

|        |                    |     |
|--------|--------------------|-----|
| 第 11 章 | 守护进程 .....         | 231 |
| 11.1   | 输出守护进程消息 .....     | 231 |
| 11.1.1 | syslogd 进程 .....   | 231 |
| 11.1.2 | syslog()函数 .....   | 232 |
| 11.1.3 | closelog()函数 ..... | 234 |
| 11.2   | 创建守护进程 .....       | 234 |
| 11.2.1 | 守护进程的创建过程 .....    | 234 |
| 11.2.2 | 创建守护进程的代码 .....    | 234 |
| 11.3   | 配置守护进程 .....       | 235 |
| 11.4   | 守护进程实例 .....       | 236 |
| 11.5   | 小结 .....           | 241 |
| 第 12 章 | 原始套接字 .....        | 243 |
| 12.1   | 产生原始套接字 .....      | 243 |
| 12.2   | 写原始套接字 .....       | 244 |
| 12.3   | 读原始套接字 .....       | 244 |
| 12.4   | 原始套接字实例 .....      | 245 |
| 12.5   | 小结 .....           | 253 |
| 第 13 章 | 数据链路访问 .....       | 254 |
| 13.1   | 数据链路访问方法 .....     | 254 |
| 13.1.1 | BSD 包过滤器 .....     | 254 |
| 13.1.2 | DLPI .....         | 254 |
| 13.1.3 | SOCK_PACKET .....  | 254 |
| 13.1.4 | libpcap .....      | 255 |
| 13.2   | libpcap 应用 .....   | 255 |
| 13.2.1 | libpcap 库函数 .....  | 255 |
| 13.2.2 | libpcap 数据结构 ..... | 258 |
| 13.2.3 | 过滤程序 .....         | 258 |
| 13.3   | 数据链路访问实例 .....     | 259 |
| 13.4   | 小结 .....           | 264 |
| 第 14 章 | 多接口设计 .....        | 265 |
| 14.1   | 单个服务器绑定到多个接口 ..... | 265 |

---

|               |   |            |
|---------------|---|------------|
| 14.2          | 多个服务器绑定到多个接口 .....                          | 269        |
| 14.3          | 小结 .....                                    | 274        |
| <b>第 15 章</b> | <b>路由套接字 .....</b>                          | <b>275</b> |
| 15.1          | 创建路由套接字 .....                               | 275        |
| 15.2          | 读写路由套接字 .....                               | 275        |
| 15.3          | 读取路由信息 .....                                | 277        |
| 15.4          | 路由套接字实例 .....                               | 279        |
| 15.5          | 小结 .....                                    | 282        |
| <b>第 16 章</b> | <b>简单路由器实例分析 .....</b>                      | <b>283</b> |
| 16.1          | 设计专用路由器 .....                               | 283        |
| 16.2          | 实现专用路由器 .....                               | 286        |
| 16.2.1        | 捕获数据包: myCap 类和 myCapIP 类 .....             | 286        |
| 16.2.2        | 查询系统路由: myRoute 类 .....                     | 293        |
| 16.2.3        | 发送 IP 包: myRaw 类 .....                      | 297        |
| 16.2.4        | 封装串口通信: SerialComm 类 .....                  | 300        |
| 16.2.5        | 处理专用数据传输网络协议: myDevice 类 .....              | 300        |
| 16.2.6        | 同时发送和接收: sendThr、recvThr 和 myRouter 类 ..... | 302        |
| 16.3          | 小结 .....                                    | 310        |
| <b>附录 A</b>   | <b>套接字 Wrapper 类源程序 .....</b>               | <b>311</b> |
| <b>附录 B</b>   | <b>串口通信类源程序 .....</b>                       | <b>337</b> |

# 第一部分 网络基础

## 第 1 章 Unix 系统基础

本章介绍 Unix 系统、常用 Unix 网络命令及软件开发环境，并通过简单的实例说明网络程序开发的过程。

### 1.1 Unix 系统概述

#### 1.1.1 Unix 系统的历史

Unix 系统是 20 世纪 70 年代由 AT&T 的 Bell 实验室开发的。到目前为止，Unix 有两大流派：那就是 AT&T 的 System V 与 BSD (Berkeley Software Distribution)。SVR4 是两大流派融合后的产物。1991 年底，与 System V 针锋相对的 Open Software Foundation 推出了 OSF/1。其中 SVR4 融合了 System V、BSD、SunOS，是各种 Unix 中的主流。

#### 1.1.2 Unix 系统的特点

Unix 问世以来，经过几十年的发展，现已成为功能最为强大和稳定的网络操作系统。它具有如下特点：

- 真正的多任务、多用户操作系统
- 可移植性强
- 完整的网络功能(TCP/IP 网络支持)
- 虚拟内存
- 共享库

#### 1.1.3 Unix 系统的体系结构

Unix 系统分为 3 个层次：用户、核心以及硬件。其中系统调用是用户程序与核心间的接口，通过系统调用进程可由用户模式转入核心模式，在核心模式下完成一定的服务请求后再返回用户模式。系统调用接口通过库把函数调用映射成进入操作系统所需要的原语。

系统调用是在特权方式下运行的，可以存取核心数据结构和它所支持的用户级数据。系统调用的主要功能是用户可以使用操作系统提供的有关设备管理、文件系统、进程控制进程通信以及存储管理方面的功能，而不必了解操作系统的内部结构和有关硬件的细节，从而减轻用户负担和保护系统，并提高资源利用率。

1.1.4 Unix 系统的地址空间

Unix 虚拟地址空间分为两个部分：用户空间和系统空间。在用户模式下只能访问用户空间，而在核心模式下可以访问系统空间和用户空间。系统空间在每个进程的虚拟地址空间中都是固定的。典型的 Unix 系统地址空间分配见图 1.1。

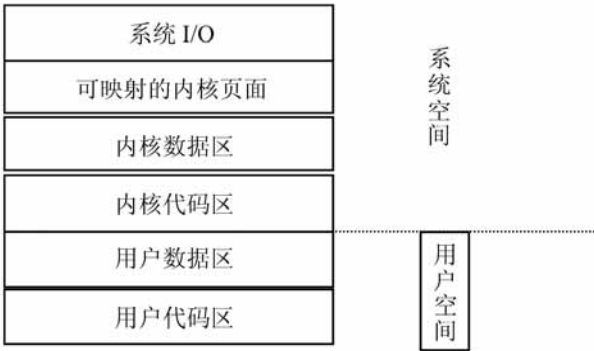


图 1.1 典型 Unix 系统地址空间分配

每个进程有自己的虚拟地址空间，对虚拟地址的引用通过地址转换机制转换为物理地址的引用。进程只能访问自己的地址空间所对应的页面，而不能访问或修改其他进程的地址空间对应的页面。Unix 通过虚存管理机制实现了这种保护。

1.1.5 POSIX 标准

POSIX 表示可移植操作系统接口 (Portable Operating System Interface)。电气和电子工程师协会 (IEEE) 开发 POSIX 标准，是为了提高 Unix 环境下应用程序的可移植性。

目前，大多数 Unix 系统都支持该标准。

1.2 常用 Unix 网络命令

在网络编程中，为了调试程序，经常会用到一些网络命令，以查看或修改网络配置、监视网络运行。以下介绍几种常用 Unix 网络命令。

1.2.1 ping

ping 检测主机连接状况。

1. 命令格式

```
ping [hostname | IP address]
```

## 2. 例子

```
ping 192.9.200.1
```

该命令用于检测本机是否与主机“192.9.200.1”连接。如果收到远程主机（IP 地址为 192.9.200.1）的回应，则说明本机与远程主机通信状态良好，否则存在网络故障。如果存在网络故障，应检查网络设备的连接及配置。

### 1.2.2 netstat

netstat 命令显示与网络有关的各种数据结构。如显示网络连接、路由表和网络接口信息。

#### 1. 命令格式

```
netstat -[r|i] [n]
```

-r: 显示路由表，同 route -e。

-i: 显示所有网络接口的信息，格式同 ifconfig -e。

-n: 以网络 IP 地址代替主机名称，显示出网络连接情形。

#### 2. 例子

显示所有网络接口的信息:

```
$ netstat -i -n
Name Mtu Net/Dest Address Ipkts Ierrs Opkts Oerrs Collis Queue
lo0 8232 127.0.0.0 127.0.0.1 380 0 380 0 0 0
eth0 1500 192.9.200.0 192.9.200.1 408 0 504 0 0 0
```

显示路由表:


```
$ netstat -r -n
Routing Table: IPv4
Destination Gateway Flags Ref Use Interface
-----
224.0.0.0 127.0.0.1 U 1 0 lo0
127.0.0.1 127.0.0.1 UH 20 440 lo0
192.9.201.0 192.9.200.2 UG 1 500 eth0
```

### 1.2.3 ifconfig

ifconfig 显示当前有效网络接口的状态。

#### 1. 命令格式

```
ifconfig [接口]
```

 注意：如以单个接口作为参数，它只显示给出的那个接口的状态；如果给出一个 -a 参数，它会显示所有接口的状态，包括那些停用的接口。

## 2. 例子

```
ifconfig -a
```

显示所有网络接口。

### 1.2.4 route

route 对内核的 IP 路由表进行操作。它主要用于给那些已经用 ifconfig 配置过的接口指定主机或网络设置静态路由。

#### 1. 命令格式

```
route [add|del] [-net|-host] target [gw Gw] [netmask Nm] [metric N]
      [[dev] If]
route [-CFvnee]
```

🔊 注意：当使用了 add 或 del 选项时，route 将修改路由表。如果没有这些选项，route 显示当前路由表的内容。

## 2. 例子

```
route add -net 192.9.200.0 netmask 255.255.255.0 eth0
```

给路由表添加一条指向网络 192.9.200.\* 的路由，其通过接口为 eth0。

```
route -e
```

显示当前路由表的内容。

### 1.2.5 tcpdump

tcpdump 用于转储网络上的数据流。它可显示出在某个网络接口上，匹配布尔表达式 expression 的报文。

#### 1. 命令格式

```
tcpdump [-adeflnNOPqStvx] [-c count] [-F file] [-i interface]
        [-r file] [-s snaplen] [-T type] [-w file] [expression]
```

-a：把网络和广播地址转换成名字。

-c：当收到数目为 count 个报文后退出。

-i：监听 interface。

-r：从 file 中读入数据报(文件是用 -w 选项创建的)。如果 file 是 "-", 就读取标准输入。

-s：从每个报文中截取 snaplen 字节的数据。

-x：以十六进制数形式显示每一个报文。

expression：用来选择要转储的数据报。如果没有指定 expression，就转储网络的全部报文。否则，只转储相对 expression 为真的数据报。



## 2. 例子

```
tcpdump ip host myhost
```

显示所有进出主机 myhost 的 IP 包。

```
tcpdump ip proto tcp
```

显示所有 TCP 包。

```
tcpdump tcp port 21
```

显示所有端口为 21 的 TCP 包。

## 1.3 网络基本配置文件

Unix 系统中包含几个网络基本配置文件，它们的主要作用是用于配置网络。

- /etc/hosts：主机名解析
- /etc/hostname.leo：网络接口名
- /etc/netmasks：网络掩码
- /etc/inetd.conf：服务项目定义

## 1.4 软件开发环境

Unix 中最常用的编辑工具是 vi，虽然它不是图形编辑器，功能却十分强大，也易于使用。由于它不需图形界面的支持，可以在任何终端上运行。

Unix 中最常用的软件开发工具是 GNU 的编译器和调试器。GCC 是 GNU 的 C 和 C++ 编译器。GCC 可同时用来编译 C 程序和 C++ 程序。编译器通过源文件的后缀名来判断是 C 程序还是 C++ 程序。但是，gcc 命令在连接时以 C 的连接方式处理。因而只能编译 C 源文件，而不能自动和 C++ 程序使用的库连接。因此，通常使用 g++ 命令来完成 C++ 程序的编译和连接，该程序会自动调用 gcc 执行编译。

### 1.4.1 vi 编辑器

vi 是一种文本编辑器，它占据了整个屏幕，在屏幕上显示文件的一部分，直到最后一行为止。用户使用屏幕的最后一行给 vi 发出命令，或让 vi 显示信息。

vi 具有如下两种模式。

- 输入模式：输入的内容作为文本显示在屏幕上。
- 命令模式：所输入的内容作为命令来执行。

开始进入 vi 时是命令模式。必须输入相关命令来切换至输入模式。使用 Esc 键可从输入模式切换至命令模式。

## 1. 用法

常用的用法是 vi 后跟随所要编辑的文件。例如：

vi 文件名

## 2. vi 的常用命令

存盘命令如下。

- :w 保存到原文件中，也就是 vi 命令列中第一个参数。
- :w filename 另存到文件 filename 中。

退出命令如下。

- :q 停止编辑并退出（若已经修改内容，但尚未存储这些改变，vi 会拒绝退出）。
- :wq 存储并退出。
- :ZZ 同:wq。
- :q! 不存盘退出。

移动光标命令如下。

- h 将光标向左移动一个字符。
- j 将光标向下移动一个字符。
- k 将光标向上移动一个字符。
- l 将光标向右移动一个字符。

模式转换命令如下。

- Esc 使用 a、i、O 或 o 指令进入输入模式后，可使用 Esc 键从输入模式返回到命令模式。

添加命令如下。

- a 将输入加到光标后，并且进入输入模式。
- i 将输入加到光标前，并且进入输入模式。

产生新行命令如下。

- o 于光标的下方产生新的一行，并且进入输入模式。
- O 于光标的上方产生新的一行，并且进入输入模式。

删除命令如下。

- x 删除光标位置上的字符。
- dd 删除光标位置上整行内容。

拷贝命令如下。

- yy 拷贝光标所在行。
- p 将拷贝粘贴到光标下一位置。

查找命令如下。

- /text 寻找 text，并移动光标至第一个字符。
- :n n 代表行号，将光标移到该行。

### 1.4.2 gcc 编译器

#### 1. 用法

```
gcc [option | filename]...  
g++ [option | filename]...
```

#### 2. 选项

- -ansi：只支持 ANSI 标准的 C 语法。
- -c：只编译并生成目标文件。
- -E：只运行 C 预编译器。
- -g：生成调试信息。GNU 调试器可利用该信息。
- -IDIRECTORY：指定额外的头文件搜索路径 DIRECTORY。
- -LDIRECTORY：指定额外的函数库搜索路径 DIRECTORY。
- -llibrary：连接时搜索指定的函数库 LIBRARY。
- -o FILE：生成指定的输出文件。用在生成可执行文件时。
- -shared：生成共享目标文件。通常用于建立共享库。
- -static：禁止使用共享连接。
- -UMACRO：取消对 MACRO 宏的定义。
- -w：不生成任何警告信息。
- -Wall：生成所有警告信息。

### 1.4.3 gdb 调试器

#### 1. gdb 的功能

GNU 的调试器为 gdb，该程序是一个交互式工具，工作在字符模式。gdb 是功能强大的调试程序，主要功能如下。

- 设置断点；
- 监视程序变量的值；
- 程序的单步执行；
- 修改变量的值。

#### 2. 用法

在使用 gdb 调试程序之前，必须使用 -g 选项编译源文件。运行 gdb 调试程序时使用如下命令：

```
gdb myprogram
```

#### 3. gdb 的常用命令

- break NUM：在指定的行上设置断点。
- bt：显示所有的调用栈帧。该命令可用来显示函数的调用顺序。
- clear：删除设置在特定源文件、特定行上的断点。其用法为：clear

FILENAME:NUM。

- continue：继续执行正在调试的程序。该命令用在程序由于处理信号或断点而导致停止运行时。
- file FILE：装载指定的可执行文件进行调试。
- help NAME：显示指定命令的帮助信息。
- kill：终止正被调试的程序。
- list：显示源代码段。
- next：向前执行下一行源代码。
- step：单步执行。
- print EXPR：显示表达式 EXPR 的值。

## 1.5 简单实例

下面是一个简单的网络编程实例。该实例是一个客户端程序，它首先连接到一个标准时间服务器上，从服务器读取当前时间，然后显示时间。多数 Unix 主机有一个标准时间服务器，它随系统的启动而启动，并在 TCP 端口 13 等待。当它收到客户连接请求后，则建立连接，然后将时间以字符串形式发送给客户，最后关闭连接。

### 1.5.1 源程序分析

#### 1. 源程序（程序 1.1）

实例源程序如下：

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <netinet/in.h>
4 #include <sys/socket.h>
5 #include <netdb.h>
6 #include <unistd.h>
7 #include <string.h>

8 #define HOSTNAMELEN 40
9 #define BUFLLEN 1024
10 #define PORT 13 /* port of daytime server */

11 int main(int argc, char *argv[])
12 {
13     int rc;
14     int sockfd; /* socket descriptor */
15     char buf[BUFLLEN+1];
16     char* pc;
17     struct sockaddr_in sa;
```

```
18     struct hostent*     hen;

19     if (argc < 2) {
20         fprintf(stderr, "Missing host name\n");
21         exit (1);
22     }

23     /* Address resolution */
24     hen = gethostbyname(argv[1]);
25     if (!hen) {
26         perror("couldn't resolve host name");
27     }

28     memset(&sa, 0, sizeof(sa));
29     sa.sin_family = AF_INET;
30     sa.sin_port = htons(PORT);
31     memcpy(&sa.sin_addr.s_addr, hen->h_addr_list[0], hen->h_length);

32     sockfd = socket(AF_INET, SOCK_STREAM, 0);
33     if (sockfd < 0) {
34         perror("socket() failed");
35     }

36     rc = connect(sockfd, (struct sockaddr *)&sa, sizeof(sa));
37     if (rc) {
38         perror("connect() failed");
39     }

40     /* reading the socket */
41     pc = buf;
42     while (rc = read(sockfd, pc, BUFLen - (pc-buf))) {
43         pc += rc;
44     }

45     /* close the socket */
46     close(sockfd);
47     /* pad a null character to the end of the result */
48     *pc = '\0';
49     /* print the result */
50     printf("Time: %s", buf);
51     /* and terminate */
52     return 0;
53 }
```

## 2. 源程序分析

1~7：所需要的头文件。

8：定义主机名长度。

9：定义接收缓冲区长度。

10：定义时间服务器端口号。

19~22：提示用户从命令行中输入服务器主机名。

23~27：地址解析。

28~31：产生套接字地址，用于连接操作，它指明所连接服务器的地址及端口号。

32~35：产生套接字。

36~39：建立与服务器的连接。

40~44：接收服务器发回的字串形式时间。

46：关闭套接字。

48：将收到的字符串后一字节设为“\0”，表示字符串结束，以便于显示。

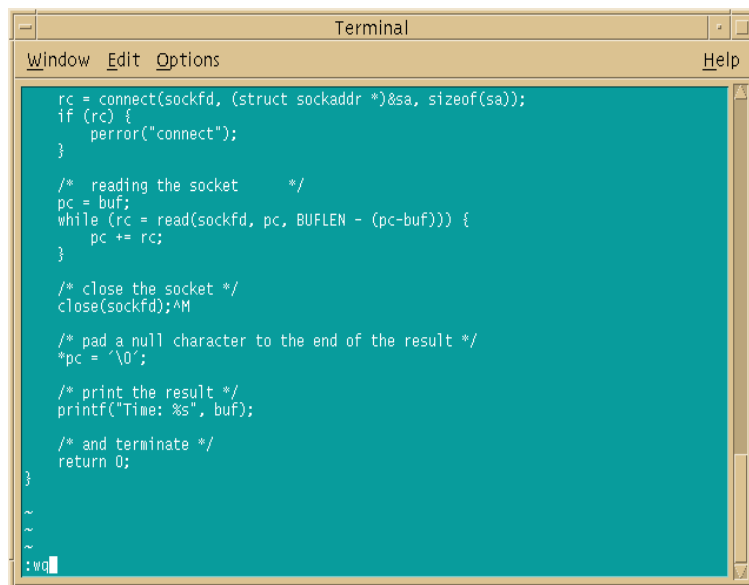
50：显示时间。

## 1.5.2 实现过程

下面以 Sun Solaris 系统为例介绍本实例的实现过程。

### 1. 编写代码

程序的文件名为 simpleexample.c，我们可以通过 vi 编辑器来完成。输入命令 vi simpleexample.c 打开 vi 编辑器。键入 a 后，输入程序。完成后，键入 wq 存盘并退出。此过程见图 1.2。



```
rc = connect(sockfd, (struct sockaddr *)&sa, sizeof(sa));
if (rc) {
    perror("connect");
}

/* reading the socket */
pc = buf;
while (rc = read(sockfd, pc, BUflen - (pc-buf))) {
    pc += rc;
}

/* close the socket */
close(sockfd);

/* pad a null character to the end of the result */
*pc = '\0';

/* print the result */
printf("Time: %s", buf);

/* and terminate */
return 0;
}

~
~
~
:wq
```

图 1.2 编辑程序

## 2. 编译和运行

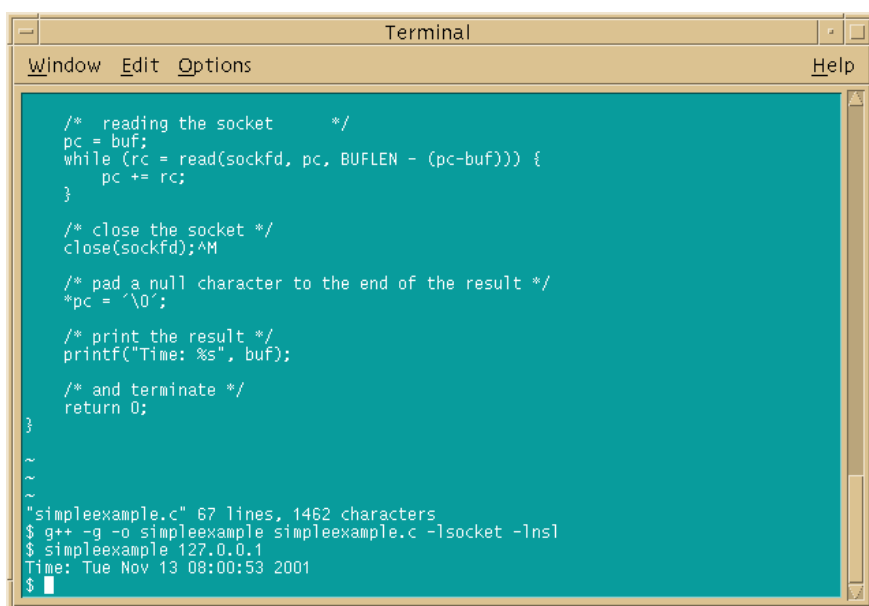
完成程序的编辑后, 进行编译。我们使用 `g++`。为了使用调试工具, 应选用 `-g` 参数以产生符号表; 为生成可执行文件 `simpleexample` 应选用 `-o` 参数; 为完成网络功能要连接 `socket` 和 `ns1` 库, 应选用 `-l` 参数。完整的命令如下:

```
g++ -g -o simpleexample simpleexample.c -lsocket -lnsl
```

如果产生编译错误, 则利用 `vi` 进行修改。编译连接完成后, 就可运行程序。我们以本机作为服务器, 运行该程序。前面提到 Unix 主机有一个标准时间服务器, 它随系统的启动而启动并等候在 TCP 端口 13 上。因此, 该程序通过连接到本机地址(其 IP 地址为 127.0.0.1)就可获得标准时间。地址是通过命令行参数输入的, 我们键入如下命令运行该程序:

```
simpleexample 127.0.0.1
```

编译和运行过程见图 1.3。



```
/* reading the socket */
pc = buf;
while (rc = read(sockfd, pc, BUFLen - (pc-buf))) {
    pc += rc;
}

/* close the socket */
close(sockfd); *M

/* pad a null character to the end of the result */
*pc = '\0';

/* print the result */
printf("Time: %s", buf);

/* and terminate */
return 0;
}

~
~
~
"simpleexample.c" 67 lines, 1462 characters
$ g++ -g -o simpleexample simpleexample.c -lsocket -lnsl
$ simpleexample 127.0.0.1
Time: Tue Nov 13 08:00:53 2001
$
```

图 1.3 编译和运行过程

## 3. 调试

如果运行出现错误, 就要进行调试。我们利用 `gdb` 来调试程序。

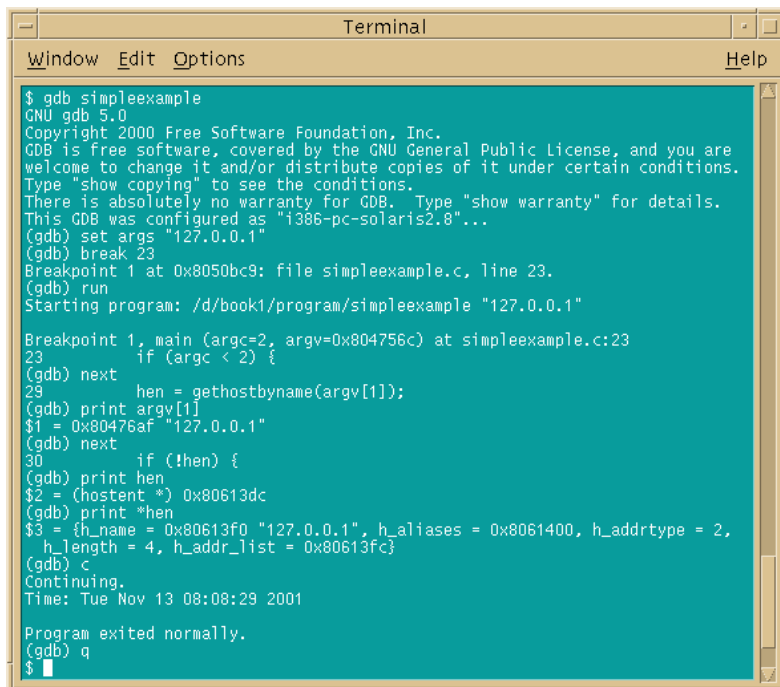
- (1) 首先, 键入命令 `gdb simpleexample` 启动调试器。
- (2) 然后通过参数输入被调试的程序。为设置程序的参数 127.0.0.1, 用命令:

```
set args "127.0.0.1"
```

- (3) 为在 23 行设置断点, 用命令:

```
break 23
```

- (4) 用命令 `run` 运行程序直到断点处。
- (5) 用命令 `next` 执行下一条语句。
- (6) 然后，用命令 `print argv[1]` 检查输入的参数是否正确。
- (7) 再用命令 `next` 执行下一条语句并用命令 `print hen` 检查变量 `hen`。
- (8) 用命令 `c` 完成程序的运行。
- (9) 用命令 `q` 退出调试器。调试过程见图 1.4。



```
$ gdb simpleexample
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-pc-solaris2.8"...
(gdb) set args "127.0.0.1"
(gdb) break 23
Breakpoint 1 at 0x8050bc9: file simpleexample.c, line 23.
(gdb) run
Starting program: /d/book1/program/simpleexample "127.0.0.1"

Breakpoint 1, main (argc=2, argv=0x804756c) at simpleexample.c:23
23      if (argc < 2) {
(gdb) next
29      hen = gethostbyname(argv[1]);
(gdb) print argv[1]
$1 = 0x80476af "127.0.0.1"
(gdb) next
30      if (!hen) {
(gdb) print hen
$2 = (hostent *) 0x80613dc
(gdb) print *hen
$3 = {h_name = 0x80613f0 "127.0.0.1", h_aliases = 0x8061400, h_addrtype = 2,
      h_length = 4, h_addr_list = 0x80613fc}
(gdb) c
Continuing.
Time: Tue Nov 13 08:08:29 2001

Program exited normally.
(gdb) q
$
```

图 1.4 调试过程

## 1.6 小 结

Unix 系统具有完整的网络功能，并提供了相应的网络命令，用于配置、监视网络，例如 `pingnetstat` 等。这些命令对网络程序的调试非常有用。

`vi` 是一种功能极为强大的文本编辑器，由于它可在多种终端上使用，已成为 Unix 系统中最常用的编辑器。`gcc` 和 `gdb` 则是 Unix 系统中最常用的 C/C++ 编译器和调试器。



## 第2章 TCP/IP

TCP/IP 协议是网络编程的基础。本章讲述其体系、端口号分配、IP 路由及其常用协议 (IP、TCP、UDP 和 ICMP)。

### 2.1 TCP/IP 体系

TCP/IP 协议(Transmission Control Protocol/Internet Protocol)意为传输控制/网际协议,是一种网络通信协议,它是 INTERNET 国际互联网络的基础。由于 INTERNET 的广泛使用,使得 TCP/IP 成了事实上的工业标准。

TCP/IP 协议族体系由四个层次组成:网络接口层、网间网层、传输层、应用层。其体系结构见图 2.1。

网络接口层是 TCP/IP 的最低层,负责从网络接收 IP 数据报及通过网络发送 IP 数据报。

网间网层负责相邻计算机之间的通信。主要包括的协议有:IP、ICMP、RIP、OSPF、IGMP 等。其功能包括如下三方面。

- 处理来自传输层的分组发送请求,收到请求后,将分组装入 IP 数据报,填充报头,选择去往信宿机的路径,然后将数据报发往适当的网络接口。
- 处理输入数据报:检查其合法性,去掉 IP 报头,将剩下部分交给适当的传输协议。
- 处理路由、流控、拥塞等。

传输层负责网络主机与主机的通信。主要包括的协议有:TCP 和 UDP。其功能如下。

- 格式化信息流;
- 提供连接/非连接端对端的传输。

应用层向用户提供一组常用的应用程序,比如电子邮件收发、文件传输访问、远程登录等。

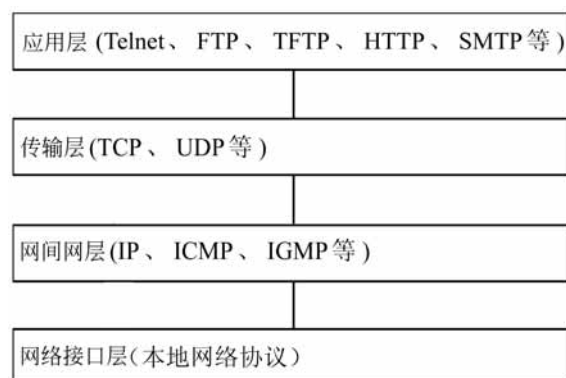


图 2.1 TCP/IP 体系结构

## 2.2 IP 协议

IP(Internet Protocol)协议是 TCP/IP 的核心协议，它提供无连接的传输服务。它在执行数据包传输时并不首先建立连接，没有反馈重发等纠错机制，因而不能保证包的可靠传输。

IP 协议定义了 TCP/IP 网络的基本传输单元。IP 执行 IP 路由、打包/拆包以及包的丢弃等功能。

### 2.2.1 IP 包的结构

IP 包的结构见图 2.2。

|        |     |      |       |      |    |    |
|--------|-----|------|-------|------|----|----|
| 0      | 4   | 8    | 16    | 19   | 24 | 31 |
| 版本     | 头长度 | 服务类型 | 数据包总长 |      |    |    |
| 标识     |     |      | 碎片标志  | 碎片偏移 |    |    |
| 生存时间   |     | 协议   | 包头校验和 |      |    |    |
| 源IP地址  |     |      |       |      |    |    |
| 目的IP地址 |     |      |       |      |    |    |
| IP选项   |     |      |       |      | 填充 |    |
| 数据     |     |      |       |      |    |    |

图 2.2 IP 包结构

版本：IP 协议的版本号，如 IPV4。

头长度：即包头长度，这个值以 4 字节为单位。如果 IP 包没有选项，IP 协议包头的固定长度为 20 个字节，那么这个值为 5。

服务类型：说明提供服务的优先权。

数据包总长：说明 IP 数据包的长度，包括包头和数据。以字节为单位。

标识：标识 IP 数据包，以便于组装碎片。

碎片标志：占 3 个位，分别为 0、DF 和 MF。DF 为 0 表示该包可被拆分成多个碎片，否则不能。MF 为 0 表示最后的碎片，为 1 表示还有碎片。

碎片偏移：表示碎片的偏移位置，第一个必须为 0。

生存时间：经过一个路由时减 1，直到为 0 时被丢弃。

协议：表示创建这个 IP 数据包所包含的高层协议，如 TCP、UDP 协议。

包头校验和：提供对包头的校验。

源 IP 地址：发送者的 IP 地址。

目的 IP 地址：接收者的 IP 地址。

IP 选项：其长度可变，如果不是 32 位的倍数，则填充 0。

### 2.2.2 IP 地址组成

IP 地址标识着网络中一个主机的位置。每个 IP 地址都是由两部分组成的：

- 网络号
- 主机号

其中网络号标识一个网络，同一个网络上所有主机具有相同的网络号，该号在互联网中是惟一的；而主机号确定网络中的一个工作站、服务器、路由器等。对于同一个网络号来说，主机号是惟一的。

### 2.2.3 IP 地址表示

IP 地址有两种表示形式。

- 二进制表示：计算机内部表示方法。
- 点分十进制表示：便于人们使用的表示方法。

例如：

192.9.200.1

为点分十进制表示的 IP 地址，其对应的二进制表示为：

11000000000010011100100000000001

每个 IP 地址的长度为 4 字节。

### 2.2.4 IP 地址类型

为适应不同大小的网络，Internet 定义了 5 种 IP 地址类型。可以通过 IP 地址的前 8 位来确定地址的类型，如图 2.3 所示。

| 类型  | IP 形式   | 网络号   | 主机号   |
|-----|---------|-------|-------|
| A 类 | w.x.y.z | W     | x.y.z |
| B 类 | w.x.y.z | w.x   | y.z   |
| C 类 | w.x.y.z | w.x.y | z     |

图 2.3 IP 地址类型

#### 1. A 类地址

可以拥有很大数量的主机，最高位为 0，紧跟的 7 位表示网络号，余下 24 位表示主机号，总共允许有 126 个网络。其范围为 0.1.0.0 ~ 126.0.0.0。

#### 2. B 类地址

被分配到中等规模和大规模的网络中，最高两位总被置于二进制的 10，允许有 16384 个网络。其范围为 128.0.0.0 ~ 191.255.0.0。

#### 3. C 类地址

用于局域网。高三位被置为二进制的 110，允许大约 200 万个网络。其范围为 192.0.1.0

~ 223.255.255.0。

#### 4. D 类地址

用于多路广播组用户，高四位总被置为 1110，余下的位用于标明客户所属的组。其范围为 224.0.0.0 ~ 239.255.255.255。

#### 5. E 类地址

这是一种仅供试验的地址。其范围为 240.0.0.0 ~ 247.255.255.255。

### 2.2.5 子网掩码

TCP/IP 上的每台主机都需要一个子网掩码。它是一个 4 字节的地址，用来屏蔽 IP 地址的一部分，以区分网络号和主机号。

子网掩码用于确定 IP 包的目的网络：把目的地址和子网掩码做“与”运算以获得网络号，如果网络号与本地网相同，则 IP 包属于本地网上的某台主机，否则 IP 包将通过路由器送到其他网络。

例如，目的地址为 192.9.201.3，子网掩码为 255.255.254.0，两者做“与”运算结果为 192.9.200.0。

如果本地网网络号为 192.9.200.0，则说明该目的地址在本网内，否则要将该包发送给路由器以发送到其他网络。

## 2.3 TCP 协议

TCP（传输控制协议）是一种可靠的面向连接的传送服务。主机交换数据必须首先建立连接，传输完毕后断开连接。它用位流通信，即数据被作为无结构的字节流。它提供反馈重发机制，从而保证数据的可靠传输。

### 2.3.1 建立 TCP 连接

TCP 连接建立是一个三次握手过程，三次握手的目的是使数据的发送和接收同步。其过程如下。

（1）主叫主机发送一个具有同步标志置位的 TCP 数据包(SYN)给被叫主机，即发出连接请求。

（2）被叫主机收到连接请求后，通过发回具有以下项目的数据包(SYN&ACK)表示接受请求：同步标志置位、数据包的序列号、确认号（其值为已收到的请求包的下一个序列号）。

（3）主叫主机再回送确认包(ACK)。

### 2.3.2 关闭 TCP 连接

TCP 连接关闭过程如下。

（1）请求主机发送一个关闭连接请求的 TCP 数据包(FIN)。

(2) 回应主机收到关闭连接请求后, 发回接受关闭连接请求的数据包(ACK)。然后, 再发回一个关闭连接请求的 TCP 数据包(FIN)。

(3) 请求主机再回送确认包(ACK)。

### 2.3.3 TCP 数据包结构

TCP 数据包的结构如图 2.4 所示。

|     |    |     |      |    |    |
|-----|----|-----|------|----|----|
| 0   | 4  | 10  | 16   | 24 | 31 |
| 源端口 |    |     | 目的端口 |    |    |
| 序列号 |    |     |      |    |    |
| 确认号 |    |     |      |    |    |
| 头长度 | 保留 | 控制位 | 窗口   |    |    |
| 校验和 |    |     | 紧急指针 |    |    |
| 选项  |    |     |      | 填充 |    |
| 数据  |    |     |      |    |    |

图 2.4 TCP 包结构

源端口: 发送 TCP 数据的源端口。

目的端口: 接收 TCP 数据的目的端口。

序列号: 标识该 TCP 包的序列号。

确认号: 确认序列号, 表示下一次接收的数据序列号。

窗口: 表明所能接收的最大字符数。

紧急指针: 紧急数据的位置。

控制位: 占 6 个位, 分别表示 TCP 包的如下 6 种类型。

- URG: 如果设置紧急数据指针, 则该位为 1。
- ACK: 为 1 的时候, 表示确认已收到的包。
- PSH: 如果设置为 1, 那么接收方收到数据后, 立即交给上一层程序。
- RST: 为 1 的时候, 表示请求重新连接。
- SYN: 为 1 的时候, 表示请求建立连接。
- FIN: 为 1 的时候, 表示请求关闭连接。

关于 TCP 协议的详细情况, 请查看 RFC793。

## 2.4 UDP 协议

UDP 协议(用户数据报协议)是建立在 IP 协议基础之上的, 用在传输层的协议。UDP 提供了无连接的数据报服务。UDP 和 IP 协议一样, 是不可靠的数据报服务。

UDP 的包格式如图 2.5 所示。

|          |          |    |
|----------|----------|----|
| 0        | 16       | 32 |
| UDP源端口   | UDP目的端口  |    |
| UDP数据报长度 | UDP数据报校验 |    |
| 数据       |          |    |

图 2.5 UDP 包结构

🔊 注意：关于 UDP 协议的详细情况，请参考 RFC768。

## 2.5 ICMP 协议

ICMP(Internet Control Message Protocol)是当 IP 传递错误时 ,用来报告传递错误信息的。例如，由于目的主机故障、IP 包由于生命期结束或中间路由器拥塞而造成 IP 包的丢失，此时必须用 ICMP 协议来告诉发送主机所发生的错误。

ICMP 包与 TCP、UDP 包一样，是封装在 IP 包中被传递的。其包结构如图 2.6 所示。

|    |    |     |    |
|----|----|-----|----|
| 0  | 8  | 16  | 32 |
| 类型 | 代码 | 校验和 |    |
| 数据 |    |     |    |

图 2.6 ICMP 包结构

类型：表明 ICMP 包的类型。ICMP 包括多种类型，如响应请求、响应应答、目的地无法抵达等。

代码：表明相应的错误。随不同类型的 ICMP 包而不同。

数据：数据部分的格式，也是随不同类型的 ICMP 包而不同。对于 ping 命令使用的响应请求和响应应答的 ICMP 包格式如图 2.7 所示。

|         |       |     |    |
|---------|-------|-----|----|
| 0       | 8     | 16  | 32 |
| 类型（8/0） | 代码（0） | 校验和 |    |
| 标识      |       | 序列号 |    |
| 可选数据    |       |     |    |

图 2.7 响应请求和响应应答的 ICMP 包格式

类型：响应请求为类型 8，响应应答为类型 0。

代码：代码为 0。

标识和序列号：用于发送方来匹配应答和响应包。

可选数据：附带的数据。

🔊 注意：关于 ICMP 协议的详细情况可以查看 RFC792。

## 2.6 端口号分配

TCP 和 UDP 使用 16 个位的端口号。不同的进程可以绑定到同一 IP 地址上的不同端口。端口可以被看作是网络连接的接触点。如果一个应用程序想要提供一个特定的服务，它就会将自己连接到一个端口上并等待客户。因此，在同一个 IP 地址上可提供多个服务。

### 2.6.1 端口分类

为了防止端口使用的混乱，应对端口号进行统一分配。可分为以下 3 个范围。

- “众所周知”的端口：0 ~ 1023。这些端口由 IANA 统一控制和分配。
- 注册的端口：1024 ~ 49151。这些端口虽不由 IANA 控制，但 IANA 登记这些端口的使用。
- 动态或私有的端口：49152 ~ 65535。这些端口通常只是暂时使用。

### 2.6.2 常用端口号

常用端口号如下：

- 5 远程作业登录
- 7 回显 (Echo)
- 9 丢弃 (Discard)
- 13 时间 (Daytime)
- 21 文件传输协议(控制) (FTP)
- 23 终端仿真协议 (Telnet)
- 25 简单邮件发送协议 (SMTP)
- 38 路由访问协议 (RAP)
- 42 WINS 主机名服务
- 43 “绰号” who is 服务
- 53 域名服务器 (DNS)
- 66 Oracle SQL\*NET
- 67 引导程序协议服务端 (BOOTPS)
- 68 引导程序协议客户端
- 69 小型文件传输协议 (TFTP)

- 70 信息检索协议 (GOPHER)
- 79 查询远程主机在线用户等信息 (FINGER)
- 80 超文本传输协议 (HTTP)
- 92 网络打印协议
- 93 设备控制协议

## 2.7 IP 路由

IP 路由就是为 IP 包选择一条数据传输路径的过程。发生在 TCP/IP 主机发送 IP 数据包或 IP 路由器进行 IP 包转发的时候。

IP 路由器是连接多个 TCP/IP 网络的设备，其主要工作就是为经过 IP 路由器的每个 IP 包寻找一条最佳传输路径。由此可见，选择最佳路径的路由算法是路由器的关键所在。为了完成这项工作，在 IP 路由器中保存着各种传输路径的相关数据——路由表，供路由选择时使用。

### 2.7.1 路由表分类

路由表分为两类。

#### 1. 静态路由表

由系统管理员事先设置好的固定的路由表称为静态路由表。一般是在系统安装时就根据网络的配置情况预先设置好的，它不会随未来网络结构的改变而自动改变。

#### 2. 动态路由表

动态路由表是路由器根据网络系统的运行情况和一定的路由选择算法自动调整的路由表。路由器根据路由协议提供的功能，自动计算数据传输的最佳路径。

### 2.7.2 IP 路由过程

IP 路由的基本过程如下。

(1) 当一个主机试图与另一个主机通信时，IP 根据子网掩码判断目的主机是一个本地网还是远程网。如果是本地网则直接将 IP 包发给目的主机。

(2) 如果目的主机是远程网，IP 通过查询路由表来为 IP 包选择一个路由器。若未找到相应的路由，则将 IP 包发给默认的网关 (IP 路由器)。

(3) IP 路由器收到 IP 包后，根据路由表将 IP 包路由到相应网络。重复以上过程并最终将 IP 包发送至目的主机。若未发现任何一个路由，源主机将收到一个出错信息。



## 2.8 小 结

TCP/IP 协议族分为网络接口层、网间网层、传输层、应用层。在每一层中都包含了多种协议。

IP 协议是 TCP/IP 的核心协议，其位于网间网层。TCP 协议和 UDP 协议位于传输层，提供端到端的通信，其中 TCP 是面向连接的，而 UDP 是无连接的。ICMP 用来报告传输中的错误信息，它位于网间网层。

# 第二部分 套 接 字

## 第 3 章 套接字基础

本章介绍套接字的概念、类型、套接字地址及端口等内容，为利用套接字进行网络编程打下基础。

### 3.1 套接字概述

追溯历史,Unix 网络编程接口有两个发展方向。一个是套接字接口,是由 Berkeley Unix 系统的开发人员在 80 年代初期开发出来的;另一个是传输层接口(TLI),是 System V 的开发人员于 1986 年开发的。尽管两个接口都符合 XTI,但由于套接字以其简单实用而得到更广泛的应用。

套接字是一种网络 API(应用程序编程接口)。它定义了许多函数和例程,程序员可以用它开发网络应用程序。套接字接口本意在于提供一种进程间通信的方法,使得在相同或不同主机上的进程能以相同的规范进行双向信息传送。

进程通过调用套接字接口 API 来实现相互之间的通信。套接字接口又利用下层的网络通信协议功能和系统调用实现实际的通信工作。它们之间的关系如图 3.1 所示。

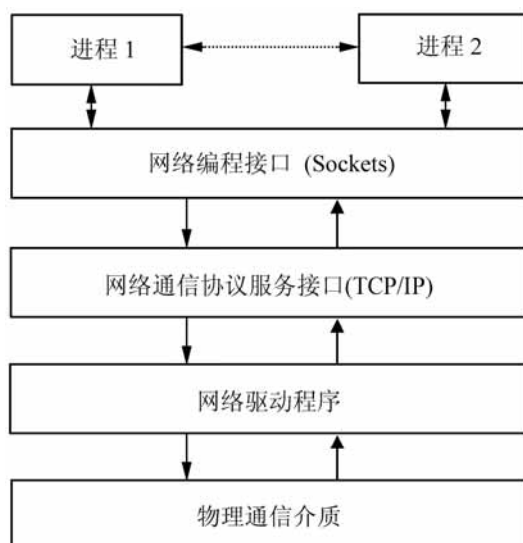


图 3.1 进程通信与套接字接口的关系图

## 3.2 套接字类型

套接字支持各种通信域，即多种不同的通信协议。目前 Unix 系统主要支持以下几种协议。

- Unix：Unix 系统内部协议
- INET：IP 版本 4
- INET6：IP 版本 6

Unix 系统支持多种套接字类型。套接字类型，是指创建套接字的应用程序所希望的通信服务类型。目前，Unix 系统主要定义了以下几种类型。

- SOCK\_STREAM：提供可靠的面向连接传输的数据流，保证数据在传输过程中无丢失、无损坏和无冗余。INET 地址簇中的 TCP 协议支持该套接字。
- SOCK\_DGRAM：提供数据的双向传输，但不保证消息的准确到达，即使消息能够到达，也无法保证其顺序性，并可能有冗余或损坏。INET 地址簇中的 UDP 协议支持该套接字。
- SOCK\_RAW：是低于传输层的低级协议或物理网络提供的套接字类型。它可访问内部网络接口。例如，可以接收和发送 ICMP 报。
- SOCK\_SEQPACKET：提供可靠的、双向的、顺序化的以及面向连接的数据通信。类似于 STREAM 方式，但它的报文大小可变（最大报文长度固定）。
- SOCK\_RDM：类似于 SOCK\_DGRAM，但它可保证数据的正确到达。

## 3.3 套接字地址结构

多数套接字函数需要一个指向地址结构的参数。对应于不同的协议簇，有不同的地址结构。这些结构的名称以 sockaddr\_ 开始，并以协议簇为后缀。例如，对于 INET，其地址结构为 sockaddr\_in。

目前，对于 TCP/IP 协议簇，IP 版本 4 最为常见，以下仅介绍其地址结构。

### 3.3.1 INET 协议簇地址结构 sockaddr\_in

sockaddr\_in 结构定义如下：

```
typedef uint16_t    in_port_t;
typedef unsigned short sa_family_t;

/*
 * IPv4 Socket address.
 */
struct sockaddr_in {
```

```
sa_family_t  sin_family;
in_port_t    sin_port;
struct in_addr sin_addr;
};
```

其中：

- sin\_family 为 Internet 地址簇，即 AF\_INET
- sin\_port 为端口号
- sin\_addr 为 IP 地址

### 3.3.2 存储地址和端口信息的 sockaddr

sockaddr 结构用于存储地址和端口信息，其定义在头文件<sys/socket.h>中。具体的结构定义如下：

```
/*
 * Structure used by kernel to store most
 * addresses.
 */
struct sockaddr {
    sa_family_t  sa_family; /* address family */
    char sa_data[14]; /* up to 14 bytes of direct address */
};
```

该地址结构由内核使用，是一个“通用地址”，因为它可用于在同一 Unix 主机上的 IPC，或不同主机进程间的通信。许多套接字函数包括该结构参数。例如：

```
int accept(int, struct sockaddr, size_t *addr-len)。
```

### 3.3.3 32 位 IPv4 地址结构 in\_addr

用于存储 32 位 IPv4 地址。其定义如下：

```
typedef uint32_t    in_addr_t;

struct in_addr {
    union {
        struct { uint8_t s_b1, s_b2, s_b3, s_b4; } _S_un_b;
        struct { uint16_t s_w1, s_w2; } _S_un_w;
        in_addr_t _S_addr;
    } _S_un;
#define s_addr  _S_un._S_addr      /* should be used for all code */
#define s_host  _S_un._S_un_b.s_b2 /* OBSOLETE: host on imp */
#define s_net   _S_un._S_un_b.s_b1 /* OBSOLETE: network */
#define s_imp   _S_un._S_un_w.s_w2 /* OBSOLETE: imp */
#define s_impno _S_un._S_un_b.s_b4 /* OBSOLETE: imp # */
#define s_lh    _S_un._S_un_b.s_b3 /* OBSOLETE: logical host */
```

```
};
```

其中包括一个联合，可有 3 种方法使用它。最常用的两种如下。

- s\_addr：无符号的 32 位整数，对应 32 位的 IP 地址。
- s\_b1、s\_b2、s\_b3、s\_b4：4 个无符号 8 位整数。对应 4 个数字表示的 IP 地址，例如 127.0.0.1。

以下两段代码都是将 IP 地址 127.0.0.1 赋给 addr。

#### 1. 代码 1

```
in_addr addr;

addr._S_un._S_un_b.s_b1 = 127;
addr._S_un._S_un_b.s_b2 = 0;
addr._S_un._S_un_b.s_b3 = 0;
addr._S_un._S_un_b.s_b4 = 1;
```

#### 2. 代码 2

```
in_addr addr;
addr.s_addr = 0x0100007f;
```

## 3.4 端 口

当客户要与另一主机上的服务进程进行通信时，除了要知道对方主机的地址，还要知道服务进程所守候的端口号。对于服务进程，它监听相应的端口，以接收客户进程的请求。

不同的服务对应不同的端口号，其对应关系在文件/etc/services 中定义。Unix 系统中常用的服务和端口号对应关系如下：

```
/*
 * Port/socket numbers: network standard functions
 */
#define IPPORT_ECHO          7
#define IPPORT_DISCARD      9
#define IPPORT_SYSTAT       11
#define IPPORT_DAYTIME      13
#define IPPORT_NETSTAT      15
#define IPPORT_FTP           21
#define IPPORT_TELNET        23
#define IPPORT_SMTP          25
#define IPPORT_TIMESERVER    37
#define IPPORT_NAMESERVER    42
#define IPPORT_WHOIS         43
#define IPPORT_MTP           57
```

```
/*
 * Port/socket numbers: host specific functions
 */
#define IPPORT_BOOTPS      67
#define IPPORT_BOOTPC      68
#define IPPORT_TFTP        69
#define IPPORT_RJE         77
#define IPPORT_FINGER      79
#define IPPORT_TTYLINK     87
#define IPPORT_SUPDUP      95

/*
 * Unix TCP sockets
 */
#define IPPORT_EXECSERVER  512
#define IPPORT_LOGINSERVER 513
#define IPPORT_CMDSERVER  514
#define IPPORT_EFSSERVER  520

/*
 * Unix UDP sockets
 */
#define IPPORT_BIFFUDP     512
#define IPPORT_WHOSERVER  513
#define IPPORT_ROUTESERVER 520 /* 520+1 also used */
```

一般而言,小于 1024 的端口由 Unix 系统保留,而大于 1024 的端口则由用户进程使用。

## 3.5 带外数据

带外数据也称为 TCP 紧急数据。在流套接字的抽象中包括了带外数据这一概念。带外数据是每一对相连流套接字间逻辑上独立的传输通道。带外数据是独立于普通数据传送给用户的。这要求带外数据设备必须支持每一时刻至少一个带外数据消息被可靠地传送。带外数据消息至少包含一个字节。在任何时刻仅有一个带外数据信息等候发送。

对于仅支持带内数据的通信协议来说(例如紧急数据是与普通数据在同一序列中发送的),系统通常把紧急数据从普通数据中分离出来单独存放。这就允许用户可以在顺序接收紧急数据和非顺序接收紧急数据之间作出选择。

## 3.6 连接类型

进程经网络进行通信时,有两种方法可以选择。

- 面向连接的方式，即虚电路方式。这种方式是在两个连接端点之间建立一条虚电路，两个端点之间的链路可以看作是直接的点到点的连接。两个端点只有在建立连接后才能传输数据。一旦连接建立，双方均可向对方发送非格式化的、可靠的字符流。例如远程登录就是采用这种方式。
- 无连接方式，即数据报方式。在传输报文前，不用建立连接。无连接协议的每个报文包含一个完整的传送地址。数据按数据包形式传输。因此，某一进程可发送信息给某一网络地址，然后再发信息给另一网络地址。

对应于进程的两种通信方式，套接字编程也有两种模式。

- 面向连接的模式。
- 面向无连接的编程模式。

### 3.7 小 结

套接字是一种网络 API，用于不同进程的通信。它支持多种通信协议，并定义了多种类型，如 `SOCK_STREAM`、`SOCK_DGRAM`、`SOCK_RAW`、`SOCK_SEQPACKET` 等。其中最常用的两种是 `SOCK_STREAM`、`SOCK_DGRAM`。`SOCK_STREAM` 用于提供面向连接的传输，而 `SOCK_DGRAM` 用于提供面向无连接的传输。

套接字包括的地址结构有 `SOCKADDR_IN`、`SOCKADDR`、`IN_ADDR` 等。其中 `SOCKADDR_IN` 包括地址簇、端口号和 IP 地址信息，`SOCKADDR` 是“通用的地址”，`IN_ADDR` 对应于 32 位 IP 地址。编程时可根据不同的需要选择相应的地址结构。

## 第 4 章 TCP 套接字

本章描述 TCP 套接字的功能，并实现一个简单却完整的 TCP 客户/服务器程序。

采用 TCP 套接字可实现基于 TCP/IP 协议、面向连接的通信模式。例如 ftp、rlogin 等均采用这种方式。

### 4.1 基本方法

#### 4.1.1 TCP 套接字实现过程

实现 TCP 套接字基本步骤分为服务器端和客户端两部分。

##### 1. 服务器端步骤

- (1) 创建套接字；
- (2) 绑定套接字；
- (3) 设置套接字为监听模式，进入被动接受连接请求状态；
- (4) 接受请求，建立连接；
- (5) 读/写数据；
- (6) 终止连接。

##### 2. 客户端步骤

- (1) 创建套接字；
- (2) 与远程服务程序连接；
- (3) 读/写数据；
- (4) 终止连接。

图 4.1 说明了 TCP 套接字在服务器和客户之间进行通信的过程。

虽然整个过程较为复杂，但模式却相对固定。因此，给出以下用于 TCP 服务器和客户编程的模板。



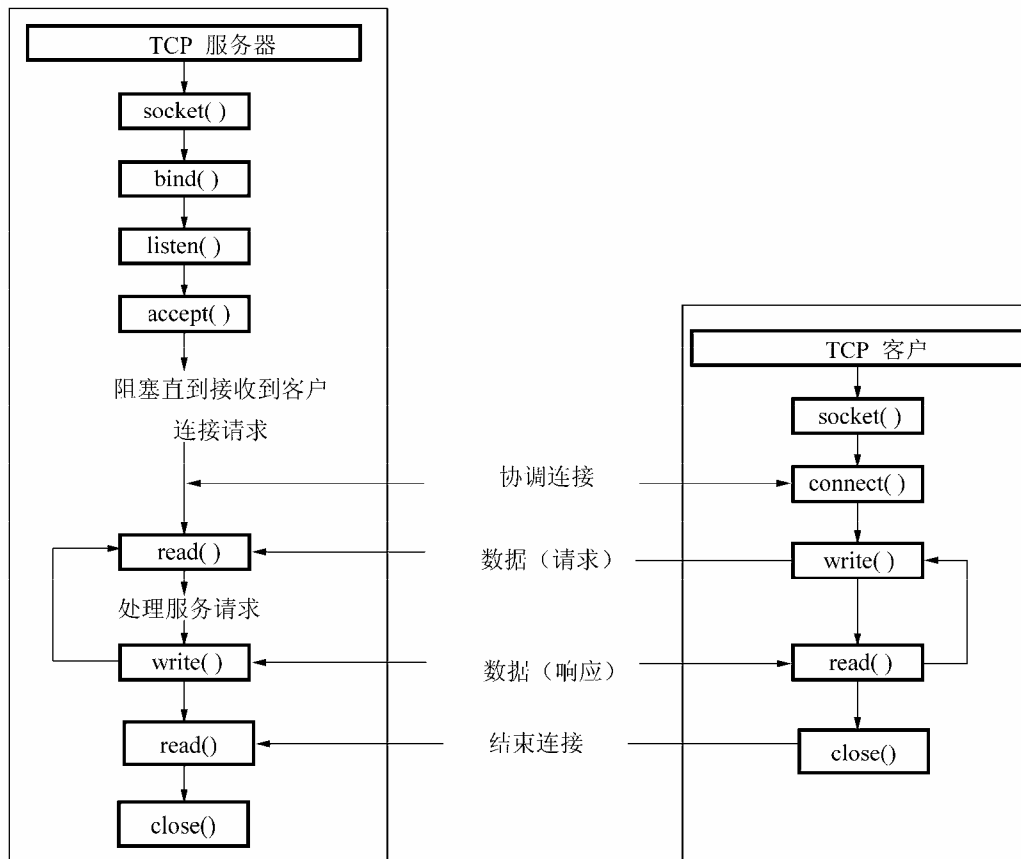


图 4.1 TCP 套接字的使用

### 4.1.2 TCP 服务器模板

```

1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>

4 main()
5 {
6     int sockfd, connect_sock;
7     /* Create TCP socket */
8     if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
9     {
10         perror("Creating socket failed.");
11         exit(1);
12     }
13     /* Bind socket to address */
14     ...
  
```

```
15  /* listen client's requirement */
16  ...
17  loop
18  {
19      /* accept connection */
20      if ((connect_sock = accept(sockfd, NULL, NULL)) == -1)
21      {
22          perror("Acception failed.");
23          exit(1);
24      }
25      /* Create child process or thread to service clients */
26      ...
27  }
28 }
```

程序说明如下。

1~3：所需头文件。

8~12：产生 TCP 套接字。

14：绑定套接字到指定地址。

16：监听客户请求。

17~28：反复接收客户连接请求，一旦接收到请求则产生子进程/线程并进行处理。

#### 4.1.3 TCP 客户模板

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>

4 main()
5 {
6     int sockfd;
7     /* Create TCP socket */
8     if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
9     {
10         perror("Creating socket failed.");
11         exit(1);
12     }
13     /* Connect to server and receive data from the server. */
14     ...
15 }
```

程序说明如下。

1~3：所需头文件。

8~12：产生套接字。

13~14：与服务器建立连接。

## 4.2 实现 TCP 套接字

### 4.2.1 产生 TCP 套接字

对于基于 TCP 的通信,无论是服务器还是客户,都必须首先产生其 TCP 通信传输端点,即 TCP 套接字。应用程序通过调用 `socket()` 产生套接字。该函数调用必须给出所使用的地址簇、套接字类型和协议标志。该函数返回一个套接字描述符。由于在 Unix 系统中套接字也是一种文件,因而套接字描述符是一种文件描述符。之后的任何 I/O 操作都是作用于该套接字描述符。其数据结构包括一个网络连接的 5 种信息:通信协议、本地协议地址、本机主机端口、远程主机地址和远程协议端口。

#### 1. socket 函数原型

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

int socket(int domain, int type, int protocol);
```

#### 2. 函数描述

该函数用于创建一个套接字,并为套接字数据结构分配存储空间。

`domain`: 协议簇和地址簇,确定 `socket` 使用的一组协议。与以下三个符号常数有关。

- `PF_UNIX`: Unix 系统内部协议
- `PF_INET`: IP 版本 4
- `PF_INET6`: IP 版本 6

`type`: 通信类型,相关常数如下。

- `SOCK_STREAM`: 以字节流形式通信,面向连接的协议使用这种形式。
- `SOCK_DGRAM`: 数据以独立的数据包形式流动,无连接协议使用这种形式。
- `SOCK_RAW`: 是低于传输层的低级协议或物理网络提供的套接字类型。它可访问内部网络接口。

`protocol`: 指明此 `socket` 请求所使用的协议,可以使用如下相关符号常数来表示。

- `IPPROTO_TCP`: 表示 TCP 协议。
- `IPPROTO_UDP`: 表示 UDP 协议。

#### 3. 返回值

如调用成功,返回 `socket` 描述符。否则返回 -1。

并不是所有的协议簇和通信类型的组合都是合法的。其合法的组合见表 4.1。

表 4.1 协议簇和通信类型的组合

|               |               |               |
|---------------|---------------|---------------|
| ● PF_INET     | ● RF_INET6    | ● PF_UNIX     |
| ● SOCK_STREAM | ● SOCK_STREAM | ● SOCK_STREAM |
| ● SOCK_DGRAM  | ● SOCK_DGRAM  | ● SOCK_DGRAM  |
| ● SOCK_RAW    | ● SOCK_RAW    | ●             |

4. 用法

为了产生 TCP 套接字 ,domain 取值为 PF\_INET ,type 取值为 SOCK\_STREAM ,protocol 取值为 0。当 protocol 值为 0 时，系统将自动为其选择相应值。由于 domain 值为 PF\_INET , type 为 SOCK\_STREAM，系统自动选择 protocol 值为 IPPROTO\_TCP。相应代码如下：

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 int sockfd;
4 /* create socket */
5 if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
6 {
7     /* handle exception */
8     ...
9     exit(1);
10 }
11 .
12 .
13 .
```

程序说明如下。

- 1~2：所需头文件。
- 5：产生套接字。
- 7~9：处理异常。

4.2.2 绑定

绑定就是将套接字和地址信息相关联，建立地址与套接字的对应关系。对于 TCP 套接字，地址信息包括 IP 地址及端口号。对于绑定操作，地址信息必须惟一，即将要绑定的地址不能用于其他连接通信。在实际应用中，通过绑定端口号来保证地址信息的惟一性。

1. bind 函数原型

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int s, const struct sockaddr *name, int namelen);
```

2. 描述

该函数将套接字描述符与相应的套接字地址对应起来，以指明套接字将使用本地的哪一个协议端口进行数据传送。

s 是套接字描述符，套接字函数返回的套接字描述符。

Name 是本地套接字地址，是指向特定套接字地址结构的指针。指定用于通信的本地协议端口。

Namelen 是本地套接字地址结构的长度。

### 3. 返回值

调用成功返回 0，不成功返回-1，并将 errno 置为相应的错误号。

🔊 注意：通常我们采用地址结构 sockaddr\_in 来提供地址信息，在进行 bind 函数调用时，再将 sockaddr\_in 转换成 sockaddr 结构。

🔊 注意：最常见的错误是所绑定的地址已被其他进程使用，此时 errno 的值为 EADDRINUSE。

### 4. 用法

对于 TCP 服务器，绑定意味着该服务器只接收来自被绑定的协议端口（IP 地址和 TCP 端口号）的数据。而对于 TCP 客户，绑定意味着该客户将通过被绑定的协议端口发送数据给服务器。但通常 TCP 客户并不需要绑定操作，而由系统决定由哪一个协议端口进行发送。在调用 bind 函数时，一般不要将端口号置为小于 1024 的值，因为 1 ~ 1024 是保留端口号，你可以选择大于 1024 的任何一个没有被占用的端口号。

典型代码如下（TCP 服务器）：

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>

4 int sockfd; /*sockfd:listen socket */
5 sockaddr_in my_addr; /* local address */
6 .
7 .
8 .
9 bzero(&my_addr, sizeof(my_addr));
10 my_addr.sin_family=AF_INET;
11 my_addr.sin_port=htons(SERVER_PORT);
12 my_addr.sin_addr.s_addr = htonl (INADDR_ANY);
13 if (bind(sockfd, (struct sockaddr *)&my_addr,
14         sizeof(struct sockaddr)) == -1) {
15     /* handle exception */
16     ...
17     exit(1);
18 }
19 .
20 .
```

程序说明如下。

1~3：所需头文件。

9~12：设置套接字地址。

13~17：绑定套接字到指定地址。

sockaddr\_in 结构更方便使用。并可以用 bzero()或 memset()函数将其置为零。指向 sockaddr\_in 的指针和指向 sockaddr 的指针可以相互转换，这意味着如果一个函数所需参数类型是 sockaddr 时，你可以在函数调用的时候将一个指向 sockaddr\_in 的指针转换为指向 sockaddr 的指针；反之亦然。

使用 bind 函数时，可以用下面的赋值实现自动获得本机 IP 地址和随机获取一个没有被占用的端口号：

```
my_addr.sin_port = 0; /* 系统随机选择一个未被使用的端口号 */
my_addr.sin_addr.s_addr = INADDR_ANY; /* 填入本机 IP 地址 */
```

通过将 my\_addr.sin\_port 置为 0，函数会自动为你选择一个未占用的端口。同样，通过将 my\_addr.sin\_addr.s\_addr 置为 INADDR\_ANY，系统会自动填入本机 IP 地址。

### 4.2.3 监听

对于 TCP 服务器而言，在绑定操作后，必须要进行监听操作，才能使客户端连接到该服务器。监听操作由 listen 函数实现，它将套接字处于被动的监听模式，因此，对于客户端不进行此操作。

#### 1. listen 函数原型

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

```
int listen(int s, int backlog);
```

#### 2. 描述

listen()为申请进入的连接建立输入数据队列，将到达本地的客户服务连接请求保存在此队列上，直到程序处理它们为止。

s 是套接字系统调用返回的套接字描述符；

backlog 指定在请求队列中允许的最大请求数。进入的连接请求将在队列中等待 accept()函数处理它们。backlog 对队列中等待服务请求的数目进行了限制。如果一个服务请求到来时，输入队列已满，该套接字将拒绝连接请求，客户将收到一个出错信息。

#### 3. 返回值

调用成功返回 0，不成功返回-1，并将 errno 置为相应的错误号。

#### 4. 用法

listen 函数只用于 TCP 服务器。请求队列中所允许的最大请求数的确定取决于实际的应用，最大请求数越大则同时所能服务的客户越多，但占用系统的资源就越多，系统的性能

会下降。

典型代码如下（TCP 服务器）：

```
1 #include <sys/types.h>
2 #include <sys/socket.h>

3 int sockfd; /*sockfd:listening socket */
4 int BACKLOG = 5;
5 .
6 .
7 .
8 if (listen(sockfd, BACKLOG) == -1) {
9     /* handle exception */
10     ...
11     exit(1);
12 }
13 .
14 .
15 .
```

程序说明如下。

1~2：所需头文件。

4：设置最大连接数为 5。

8~12：监听操作。

#### 4.2.4 接受请求

当服务器收到一个来自客户端的连接请求时，它要产生一个新的套接字实体。此时，服务器和客户端就建立了 TCP 连接，因此称之为“连接套接字”。以后服务器和客户端间的通信就建立在“连接套接字”之上。前面提到的套接字称为“监听套接字”。通常，一个服务器只产生一个“监听套接字”，而且在整个服务器生命期一直保留。而服务器可产生多个“连接套接字”以服务多个客户，当完成某一客户服务时，只关闭与该连接相应的“连接套接字”。TCP 服务器是通过调用的 `accept` 函数实现接受请求的。

##### 1. `accept` 函数原型

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

int accept(int s, struct sockaddr *addr,
           socklen_t *addrlen);
```

##### 2. 描述

`accept()` 函数让服务器接收客户端的连接请求。在建立好输入队列后，服务器就调用 `accept()` 函数，然后睡眠并等待客户的连接请求。当 `accept()` 函数监视的套接字收到连接请求

时，系统核将建立一个新的套接字，系统核将这个新套接字和请求连接进程的地址联系起来，收到服务请求的初始套接字仍可以继续以前的套接字上监听，同时可以在新的套接字描述符上进行数据传输操作。

s 是被监听的套接字描述符。

addr 是一个指向 sockaddr 变量的指针，该变量用来存放提出连接请求服务的主机的信息（某台主机从某个端口发出该请求）。

addrlen 通常为一个指向值为 sizeof(struct sockaddr\_in)的整型指针变量。

### 3. 返回值

调用成功返回非负整数代表新的套接字，不成功返回-1，并将 errno 置为相应的错误号。

addr 存放提出连接请求服务的主机的信息。

### 4. 用法

accept()函数只用于 TCP 服务器。典型代码如下：

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 int listenfd,connfd; //listenfd:listening socket
                        //connfd: connection socket
4 sockaddr_in client_addr; /* client's address */
5 socklen_t len;
6 .
7 .
8 .
9 connfd = accept(listenfd, (struct sockaddr *)&client_addr, &len);
10 if (connfd == -1) {
11     /* handle exception */
12     ...
13     exit(1);
14 }
15 .
16 .
17 .
```

程序说明如下。

1~2：所需头文件。

9：接受连接请求。

11~13：处理连接请求异常。

## 4.2.5 连接建立

面向连接的客户程序可以使用 connect()函数来配置套接字并与远程服务器建立一个 TCP 连接。



### 1. connect 函数原型

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

int connect(int s, const struct sockaddr *name, int namelen);
```

### 2. 描述

connect()函数用于启动和远程主机的直接连接。

s 是 socket()函数返回的套接字描述符。

name 是包含远程主机 IP 地址和端口号的指针。

namelen 是远程地址结构的长度。

### 3. 返回值

调用成功返回 0，不成功返回-1，并将 errno 置为相应的错误号。

### 4. 用法

只有面向连接的客户端程序使用套接字时才需要将此套接字与远程主机相连。面向连接的服务器也从不启动一个连接，它只是被动地在协议端口监听客户的请求。典型代码如下：

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>

4 int sockfd;
5 struct sockaddr_in server_addr; /* server's address */
6 .
7 .
8 .
9 bzero(&server_addr, sizeof(server_addr));
10 server_addr.sin_family=AF_INET;
11 server_addr.sin_port=htons(SERVER_PORT);
12 server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
13 if (connect(sockfd, (struct sockaddr *)&server_addr,
14             sizeof(struct sockaddr)) == -1) {
15     /* handle exception */
16     ...
17     exit(1);
18 }
19 .
20 .
```

程序说明如下。

1~3：所需头文件。

9~12：设置套接字地址。

13：请求与服务器建立连接。

14~16：处理连接请求异常。

#### 4.2.6 数据传输

一旦连接建立，服务器和客户就可进行双向的数据传输。服务器和客户各自产生相应的套接字描述符用于读写操作。由于套接字描述符也是一种文件描述符，因此可用文件读写函数（read()和 write()函数）进行发送和接收操作。然而对于一些特殊的发送和接收，套接字提供了 recv()和 send()函数。

##### 1. send 函数原型

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

ssize_t send(int s, const void *msg, size_t len, int flags);
```

##### 2. 描述

send 函数用于 TCP 套接字的数据发送。

s 是套接字描述符。对于服务器是 accept()函数返回的套接字描述符。对于客户是 socket()函数返回的套接字描述符。

Msg 是指向一个包含传输信息的数据缓冲区。

len 指明传送数据缓冲区的大小。

Flags 是传输控制标志。其值定义如下。

- 0：常规操作。如同 write()函数。
- MSG\_OOB：发送带外数据。
- MSG\_DONTROUTE：通过最直接的路径发送数据，而忽略底层协议的路由设置。

##### 3. 返回值

调用成功返回发送数据的长度（以字节为单位），否则返回-1。

##### 4. recv 函数原型

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

ssize_t recv(int s, void *buf, size_t len, int flags);
```

##### 5. 描述

recv 函数用于 TCP 套接字的数据接收。

s 是套接字描述符。对于服务器是 accept()函数返回的套接字描述符。对于客户是 socket()函数返回的套接字描述符。

buf 是指向一个包含接收信息的数据缓冲区。

len 指明接收数据缓冲区的大小。

Flags 是传输控制标志。其值定义如下。

- 0：常规操作。如同 read()函数。
- MSG\_PEEK：只查看数据而不读出数据。此意味着后续读操作仍然读出所查看的数据。
- MSG\_OOB：忽略常规数据，而只读带外数据。
- MSG\_WAITALL：recv()函数只有在将接收缓冲区填满后才返回。

#### 6. 返回值

调用成功返回接收数据的长度（以字节为单位），否则返回-1并置相应的 errno 值。当对方关闭连接时，返回的接收数据长度为 0。通常以此来判断对方是否关闭。

#### 7. 用法

recv()函数用于在面向连接的套接字上进行数据接收。典型代码如下：

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 int sockfd;
4 #define MAXDATASIZE 100
5 int recvbytes;
6 char buf[MAXDATASIZE];
7 .
8 .
9 .
10 while ((recvbytes=recv(sockfd, buf, MAXDATASIZE, 0)) > 0) {
11     /*handle received data */
12     ...
13 }
14 .
15 .
16 .
```

程序说明如下。

1~2：所需头文件。

10：接收数据。

11：处理收到的数据。

### 4.2.7 终止连接

当完成服务器和客户间的数据传输后，应终止连接以释放系统资源。有两种方法来关闭连接。一种是用 close()函数来完成，方法同关闭一个文件描述符一样。另一种方法则是调用 shutdown()函数，这种方法很灵活，允许你只停止某个方向上的数据传输，而另一方向可继续传输数据。

#### 1. shutdown 函数原型

```
int shutdown(int s, int how);
```

## 2. 描述

shutdown()函数用于关闭 TCP 套接字。

s 是套接字描述符。对于服务器是 accept 函数返回的套接字描述符。对于客户是 socket() 函数返回的套接字描述符。

how 允许为 shutdown 操作选择以下几种方式。

- 0：不允许继续接收数据。
- 1：不允许继续发送数据。
- 2：不允许继续发送和接收数据。
- 全部为允许则调用 close ()函数。

## 3. 返回值

调用成功返回 0，否则返回-1 并置相应的 errno 值。

# 4.3 TCP 套接字编程实例

## 4.3.1 实例说明

下面虽然是一个简单的 TCP 套接字编程实例，但涉及 TCP 套接字编程的主要方面，是一个完整的 TCP 客户/服务器实例。分为服务器和客户两部分。为了简单明了地说明 TCP 套接字编程的方法，此例采用一服务器对应一客户的模式。其完成的功能如下。

- 客户根据用户提供的 IP 地址，连接相应的服务器。
- 服务器等待客户的连接，一旦连接，则显示客户 IP 地址，并发信息回客户。
- 客户接收服务器机发送的信息并显示。

## 4.3.2 TCP 服务器

由于采用一服务器对应一客户的方法，服务器不需要采用任何并发技术。执行 TCP 套接字服务端的基本过程是创建套接字、绑定套接字、监听、接受请求、读/写数据和终止连接。

### 1. 服务器源程序（程序 4.1）

服务器源程序清单如下：

```
1  #include <stdio.h>          /* These are the usual header files */
2  #include <strings.h>        /* for bzero() */
3  #include <unistd.h>         /* for close() */
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>

8  #define PORT 1234          /* Port that will be opened */
```

```
9  #define BACKLOG 1  /* Number of allowed connections */

10 main()
11 {
12 int listenfd, connectfd; /* socket descriptors */
13 struct sockaddr_in server; /* server's address information */
14 struct sockaddr_in client; /* client's address information */
15 int sin_size;

16 /* Create TCP socket */
17 if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
18 {
19 /* handle exception */
20 perror("Creating socket failed.");
21 exit(1);
22 }

23 /* set socket can be reused */
24 int opt = SO_REUSEADDR;
25 setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

26 bzero(&server, sizeof(server)); /* fill server with 0s */
27 server.sin_family=AF_INET;
28 server.sin_port=htons(PORT);
29 server.sin_addr.s_addr = htonl (INADDR_ANY);
30 if (bind(listenfd, (struct sockaddr *)&server,
31         sizeof(struct sockaddr)) == -1) {
32 /* handle exception */
33 perror("Bind error.");
34 exit(1);
35 }

36 if(listen(listenfd, BACKLOG) == -1){ /* calls listen() */
37 perror("listen() error\n");
38 exit(1);
39 }

40 sin_size=sizeof(struct sockaddr_in);
41 if ((connectfd = accept(listenfd,
42         (struct sockaddr *)&client, &sin_size)) == -1) {
43 /* calls accept() */
44 perror("accept() error\n");
45 exit(1);
46 }
47 /* prints client's IP */
```

```

45  printf("You got a connection from %s\n",inet_ntoa(client.sin_addr) );
/* send to the client welcome message */
46  send(connectfd,"Welcome to my server.\n",22,0);

47  close(connectfd); /* close connectfd */
48  close(listenfd); /* close listenfd */
49  }

```

## 2. 程序分析

1~7：所需的头文件。

8~9：定义端口号和最大允许连接的数量。为简单起见，本例采用宏来定义端口号和最大允许连接的数量，读者可根据需要采用更灵活的方法。另外，由于本例不是并发服务器，所以 BACKLOG 定义为 1。

17~22：产生 TCP 套接字。

24~25：设置套接字选项为 SO\_REUSEADDR，即地址可重用方式。由于系统默认方法是只允许一个套接字绑定在一个特定的地址上（IP 地址和端口号），并且当该套接字被关闭后，系统仍不允许在该地址上绑定其他套接字。当然，这在实际应用中并不存在问题，因为服务器一旦启动则长期运行。但对于程序调试和学习，却带来极大的不便。每当需要重新运行程序时，则必须重新启动操作系统或等待相当长时间。而设置为地址可重用方式后，可马上重新运行程序以便于调试。如没有去掉这两行代码，当第二次运行该程序时，则发生绑定错误，其错误信息为“Bind error.: Address already in use”。

26~34：绑定套接字到相应地址。本例 IP 地址设为 INADDR\_ANY，则可接收来自本机任何 IP 地址的客户连接。注意，端口号和 IP 地址都应转换成网络顺序。

35~38：监听网络连接。

39~44：接受客户连接，客户地址信息存放在变量 client 中。

45~46：显示客户 IP 地址，通过函数 inet\_ntoa 将客户地址转换成可显示的 IP 地址。然后，发送信息给客户。

47~48：关闭套接字。先关闭连接套接字，再关闭监听套接字。

### 4.3.3 TCP 客户

下面程序完成客户端 TCP 套接字的基本过程是：创建套接字、与远程服务程序连接、读/写数据和终止连接。

#### 1. 客户端源程序（程序 4.2）

```

1  #include <stdio.h>
2  #include <unistd.h>
3  #include <strings.h>
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <netdb.h>          /* netdb.h is needed for struct hostent */

```

```
8  #define PORT 1234    /* Open Port on Remote Host */
9  #define MAXDATASIZE 100 /* Max number of bytes of data */

10 int main(int argc, char *argv[])
11 {
12     int fd, numbytes; /* files descriptors */
13     char buf[MAXDATASIZE]; /* buf will store received text */
14     struct hostent *he; /* structure that will get information
                           // about remote host
15     struct sockaddr_in server; /* server's address information */

16     if (argc !=2) { /* this is used because our program will need
                           // one argument (IP)
17         printf("Usage: %s <IP Address>\n",argv[0]);
18         exit(1);
19     }

20     if ((he=gethostbyname(argv[1]))==NULL){ /* calls gethostbyname() */
21         printf("gethostbyname() error\n");
22         exit(1);
23     }

24     if ((fd=socket(AF_INET, SOCK_STREAM, 0))==-1){ /* calls socket() */
25         printf("socket() error\n");
26         exit(1);
27     }

28     bzero(&server,sizeof(server));
29     server.sin_family = AF_INET;
30     server.sin_port = htons(PORT); /* htons() is needed again */
31     server.sin_addr = *((in_addr *)he->h_addr); /* he->h_addr passes "*he"'s
                                                // info to "h_addr" */

32     if(connect(fd, (struct sockaddr *)&server,
33         sizeof(struct sockaddr))==-1){ /* calls connect() */
34         printf("connect() error\n");
35         exit(1);
36     }

37     if ((numbytes=recv(fd,buf,MAXDATASIZE,0)) == -1){ /* calls recv() */
38         printf("recv() error\n");
39         exit(1);
40     }
```

```
40  buf[numbytes]='\0';
41  printf("Server Message: %s\n",buf); /* it prints server's welcome message
    */

42  close(fd); /* close fd */
}
```

## 2. 程序分析

1~7：所需的头文件。

8~9：定义端口号和接收缓冲区的大小。注意端口号要与服务器定义的一致。本例的缓冲区是采用静态方式分配的，为提高内存的使用效率可采用动态分配方式。

16~19：检查用户输入。如果用户输入不正确，提示用户正确的输入方法。

20~23：通过字符串形式的 IP 地址获得服务器的地址信息。gethostbyname()函数将在后面章节介绍。

24~27：产生 TCP 套接字。

28~35：连接到服务器。

36~39：接收服务器发来的信息，并存放在 buf 缓冲区内。

40~41：显示来自服务器的信息。numbytes 是收到的字节数，把 buf[numbytes]赋值为 0，以此标志字符串结束。

42：关闭套接字。

### 4.3.4 运行程序

下面在同一主机上运行上述程序。

(1) 首先启动服务器程序：

```
tcpserver
```

(2) 运行客户：

```
tcpclient 127.0.0.1
```

#### 1. 服务器的运行结果

```
$ tcpserver
You got a connection from 127.0.0.1
$
```

#### 2. 客户运行结果

```
$ tcpclient 127.0.0.1
Server Message: Welcome to my server.
$
```



### 3. 运行结果说明

运行结果说明本机客户能与本机服务器建立 TCP 连接并可相互发送信息。

## 4.4 小 结

TCP 套接字的实现分为 TCP 服务器和 TCP 客户。其模式相对固定,读者仅需理解其基本过程,要编程可套用相应模板。TCP 服务器的基本过程是:创建套接字、绑定套接字、监听并接受连接请求、读/写数据和终止连接。TCP 客户的基本过程是:创建套接字、连接服务器、读写数据和终止连接。

系统提供了相应的函数以实现 TCP 套接字,主要包括 `socket()`、`bind()`、`listen()`、`accept()`、`connect()`、`send()`、`recv()`、`shutdown()` 函数等。

本章的 TCP 套接字实例很好地说明了 TCP 套接字实现的基本方法,但却并不实用,无法满足复杂的网络应用。后面的章节将会讲述更实用的技术。

## 第 5 章 UDP 套接字

上一章讲述了 TCP 套接字，本章则介绍套接字另一主要的用途——UDP 套接字。

采用 UDP 套接字可实现基于 TCP/IP 协议面向无连接的通信模式。与 TCP 套接字相比，UDP 套接字不能提供可靠的数据传输，但 UDP 套接字的实现更加简单，并且具有更高的传输效率。因而同样也得到广泛的应用，如 DNS（域名系统）、NFS（网络文件系统）、SNMP（简单网络管理协议）等的应用。

### 5.1 基本方法

与 TCP 套接字的实现相似，UDP 套接字也分为服务器端和客户端两部分。

#### 5.1.1 UDP 套接字实现过程

##### 1. 服务器端步骤

- (1) 建立 UDP 套接字；
- (2) 绑定套接字到特定地址；
- (3) 等待并接收客户端信息；
- (4) 处理客户端请求；
- (5) 发信息回客户端；
- (6) 关闭套接字。

##### 2. 客户端步骤

- (1) 建立 UDP 套接字；
- (2) 发送信息给服务器；
- (3) 接收来自服务器的信息；
- (4) 关闭套接字。

图 5.1 说明了 UDP 套接字的服务器和客户端之间进行通信的过程。

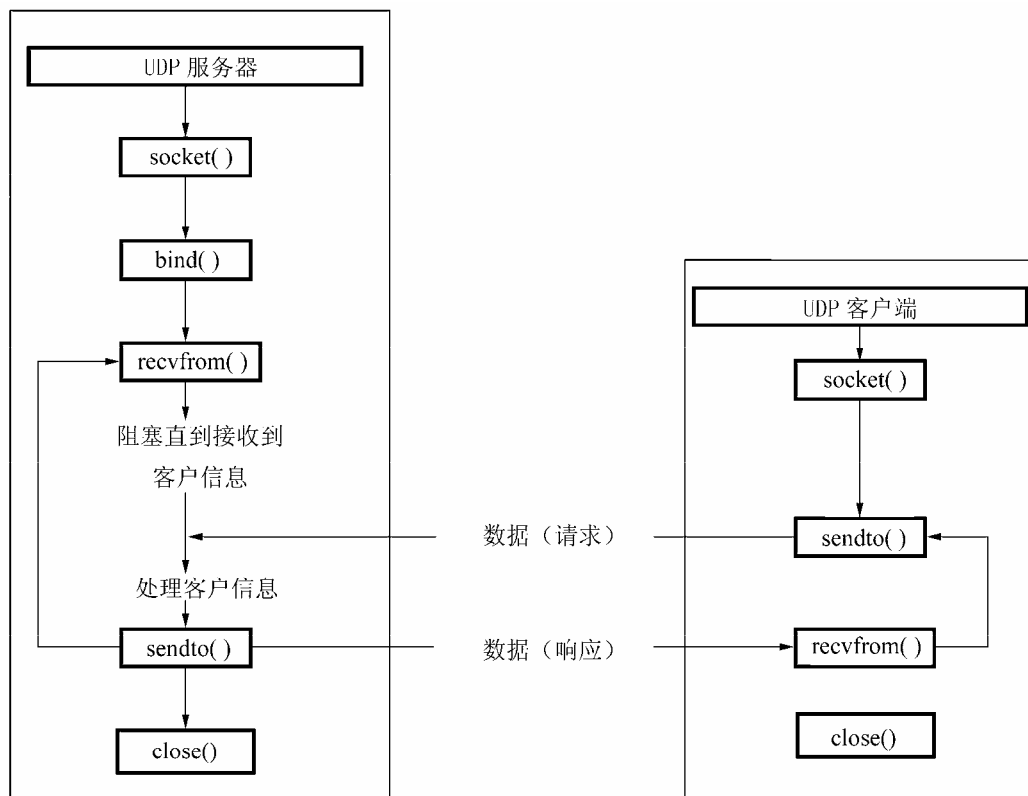


图 5.1 UDP 套接字的使用

### 5.1.2 UDP 服务器模板

UDP 服务器模板如下：

```

1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>

4 main()
5 {
6     int sockfd;
7     /* Create UDP socket */
8     if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
9     {
10         perror("Creating socket failed.");
11         exit(1);
12     }
13     /* Bind socket to address */
14     ...

```

```
15  loop {
16      /* Receive client's data */
17      ...
18      /* handle client's information*/
19      ...
20      /* send information to client */
21      ...
22  }
23  /* close socket */
24  close(sockfd);
25 }
```

程序说明如下。

1~3：所需头文件。

8~12：产生 UDP 套接字。

13~14：绑定套接字到指定地址。

15~22：反复接收客户端数据并处理，然后发数据回客户端。

### 5.1.3 UDP 客户模板

UDP 客户模板如下：

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>

4 main()
5 {
6     int sockfd;
7     /* Create UDP socket */
8     if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
9     {
10         perror("Creating socket failed.");
11         exit(1);
12     }
13     /* send data to the server */
14     ...
15     /* receive data from the server */
16     ...
17     /* close UDP socket*/
18     close(sockfd);
19 }
```

程序说明如下。

1~3：所需头文件。

8~12：产生 UDP 套接字。

13~14：发数据给服务器。  
15~16：接收服务器发回的数据。  
17~18：关闭套接字。

## 5.2 函数说明

UDP 套接字使用的函数 `socket()`、`bind()`已在上一章中介绍了，在此仅说明 UDP 独特的读/写函数（`sendto()`和 `recvfrom()`）。

### 5.2.1 UDP 套接字的数据发送——`sendto()`函数

由于 UDP 套接字使用无连接的协议，因此使用 `sendto()`函数时，应在函数调用中指明目的地址。

#### 1. `Sendto()`函数原型

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

ssize_t sendto(int s, const void *msg, size_t len, int flags,
               const struct sockaddr *to, int tolen);
```

#### 2. 描述

`sendto()`函数用于 UDP 套接字的数据发送。

`s` 是套接字描述符。

`msg` 是指向一个包含传输信息的数据缓冲区。

`len` 指明传送数据缓冲区的大小。

`flags` 是传输控制标志。其值定义如下。

- 0：常规操作。如同 `write()`函数。
- `MSG_OOB`：发送带外数据。
- `MSG_DONTROUTE`：通过最直接的路径发送数据，而忽略底层协议的路由设置。

`to` 是远程套接字地址。

`tolen` 是远程套接字地址长度。

#### 3. 返回值

调用成功返回发送数据的长度（以字节为单位），否则返回-1。

### 5.2.2 UDP 套接字的数据接收——`recvfrom()`函数

无连接的协议使用 `recvfrom()`函数，它与 `sendto()`函数相对应，需要在参数中指明地址。

## 1. recv 函数原型

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

ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, int *fromlen);
```

## 2. 描述

recv 函数用于 UDP 套接字的数据接收。

s 是套接字描述符。

buf 是指向一个包含接收信息的数据缓冲区。

len 指明接收数据缓冲区的大小。

flags 是传输控制标志。其值定义如下。

- 0：常规操作。如同 read() 函数。
- MSG\_PEEK：只查看数据而不读出数据。此意味着后续读操作仍然读出所查看的数据。
- MSG\_OOB：忽略常规数据，而只读带外数据。

from 是远程套接字地址。如果不为 NULL，则系统将信息的发送方套接字地址填入该结构。

fromlen 是远程套接字地址长度。

## 3. 返回值

调用成功返回接收数据的长度（以字节为单位），否则返回-1 并置相应的 errno 值。

## 4. 用法

recvfrom() 函数用于面向无连接的套接字上进行数据接收。典型代码如下：

```
1 #include <sys/types.h>
2 #include <sys/socket.h>

3 int sockfd;
4 struct sockaddr_in remote_addr;
5 #define MAXDATASIZE 1000
6 int recvbytes;
7 char buf[MAXDATASIZE];
8 .
9 .
10 .
11 while ((recvbytes=recvfrom(sockfd, buf, MAXDATASIZE, 0,
12                          (struct sockaddr *)&remote_addr, sizeof(struct sockaddr)))>0) {
13     /* handle received data*/
14     ...
15 }
```

16 .

17 .

程序说明如下。

1~2：所需头文件。

11：接收数据。

12~14：处理接收的数据。

## 5.3 UDP 套接字编程实例

本节介绍一个简单的 UDP 套接字编程实例，涉及了 UDP 套接字编程的主要方面。它分为服务器和客户两部分。完成的功能如下。

- 服务器守候在特定的套接字地址上，循环接收客户发来的信息，并显示客户 IP 地址及相应信息。
- 如果服务收到的客户信息为 “quit”，则退出循环，并关闭套接字。
- 客户向服务器发信息，然后等待服务器回应。一旦接收到服务发来的信息，则显示该信息，并关闭套接字。

### 5.3.1 UDP 服务器

UDP 服务器采用 UDP 套接字实现，由于是无连接的通信方式，因此不需要专门的并发技术就能服务多个不同的客户端，只须通过套接字地址区分不同的客户端。该程序执行 UDP 套接字服务端的基本过程为：创建套接字、绑定套接字、读/写数据和关闭套接字。

#### 1. 程序清单（程序 5.1）

服务器源程序清单如下：

```

1  #include <stdio.h>           /* These are the usual header files */
2  #include <strings.h>         /* for bzero() */
3  #include <unistd.h>          /* for close() */
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>

8  #define PORT 1234           /* Port that will be opened */
9  #define MAXDATASIZE 100     /* Max number of bytes of data */

10 main()
11 {
12     int sockfd; /* socket descriptors */
13     struct sockaddr_in server; /* server's address information */
14     struct sockaddr_in client; /* client's address information */

```

```
15 int sin_size;
16 int num;
17 char msg[MAXDATASIZE]; /* buffer for message */

18 /* Creating UDP socket */
19 if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
20 /* handle exception */
21 perror("Creating socket failed.");
22 exit(1);
23 }

24 bzero(&server, sizeof(server));
25 server.sin_family=AF_INET;
26 server.sin_port=htons(PORT);
27 server.sin_addr.s_addr = htonl (INADDR_ANY);
28 if (bind(sockfd, (struct sockaddr *)&server,
29         sizeof(struct sockaddr)) == -1) {
30 /* handle exception */
31 perror("Bind error.");
32 exit(1);
33 }

34 sin_size=sizeof(struct sockaddr_in);
35 while (1) {
36 num = recvfrom(sockfd, msg, MAXDATASIZE, 0,
37               (struct sockaddr *)&client, &sin_size);
38 if (num < 0){
39 perror("recvfrom error\n");
40 exit(1);
41 }

42 msg[num] = '\0';
43 printf("You got a message (%s) from %s\n", msg,
44        inet_ntoa(client.sin_addr)); /* prints client's IP */
45 sendto(sockfd, "Welcome to my server.\n", 22, 0, (struct sockaddr *)
46        &client, sin_size); /* send to the client welcome message */
47 if (!strcmp(msg, "quit")) break;
48 }

49 close(sockfd); /* close listenfd */
50 }
```

## 2. 程序分析

1~7：所需的头文件。

8~9：定义端口号和接收缓冲区的长度。



19~23：创建 UDP 套接字。

24~32：绑定套接字到本机所有 IP 地址，并且端口号为 1234。

35~39：接收来自客户的信息。收到的信息存放在 msg 中，长度为 num，客户地址存放在 client 变量中。

40~41：显示客户信息，inet\_ntoa 将客户 IP 地址转换成可显示的 IP 地址。

42：发信息回客户。由于 client 中存放的地址是刚被接收的客户，因而回的信息是给该客户。由此可见，该服务器可服务多个客户而不会混乱。

43：如果收到的客户信息是“quit”字符串，则退出循环。

45：关闭套接字。

### 5.3.2 UDP 客户

该程序通过命令行参数输入服务器 IP 地址和发给服务器的信息，然后执行 UDP 套接字客户端的基本过程：创建套接字、读/写数据和关闭套接字。

#### 1. 程序清单（程序 5.2）

```

1  #include <stdio.h>
2  #include <unistd.h>
3  #include <strings.h>
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <netdb.h>          /* netdb.h is needed for struct hostent =) */

8  #define PORT 1234          /* Open Port on Remote Host */
9  #define MAXDATASIZE 100    /* Max number of bytes of data */

10 int main(int argc, char *argv[])
11 {
12     int fd, numbytes;        /* files descriptors */
13     char buf[MAXDATASIZE];   /* buf will store received text */

14     struct hostent *he;      /* structure that will get information
                               //about remote host
15     struct sockaddr_in server,reply; /* server's address information */

16     if (argc !=3) {          // this is used because our program will need
                               // two argument (IP address and a message */
17         printf("Usage: %s <IP Address> <message>\n",argv[0]);
18         exit(1);
19     }

20     if ((he=gethostbyname(argv[1]))==NULL){ /* calls gethostbyname() */
21         printf("gethostbyname() error\n");

```

```
22  exit(1);
23  }

24  if ((fd=socket(AF_INET, SOCK_DGRAM, 0))==-1){ /* calls socket() */
25  printf("socket() error\n");
26  exit(1);
27  }

28  bzero(&server,sizeof(server));
29  server.sin_family = AF_INET;
30  server.sin_port = htons(PORT); /* htons() is needed again */
    /*he->h_addr passes "he"'s info to "h_addr" */
31  server.sin_addr = *((struct in_addr *)he->h_addr);
32  sendto(fd, argv[2], strlen(argv[2]),0,
        (struct sockaddr *)&server,sizeof(struct sockaddr));

33  while (1) {
34  int len;
35  if ((numbytes=recvfrom(fd,buf,MAXDATASIZE,0,
        (struct sockaddr *)&reply,&len)) == -1){ /* calls recvfrom() */
36  printf("recvfrom() error\n");
37  exit(1);
38  }

39  if (len != sizeof(struct sockaddr) || memcmp((const void *)&server,
        (const void *)&reply,len) != 0) {
40  printf("Receive message from other server.\n");
41  continue;
42  }

43  buf[numbytes]='\0';
    /* print server's welcome message */
44  printf("Server Message: %s\n",buf);
45  break;
46  }

47  close(fd); /* close fd */
48  }
```

## 2. 程序分析

1~7：所需头文件。

8~9：端口号和缓冲区长度。

16~19：检查用户输入，如果用户输入不正确，则提示用户正确的输入方法。本例采用命令行方式进行用户输入。

20~23：将字符串形式的 IP 地址转换成相应服务器的地址信息。

24~27：创建套接字。

28~31：用相应信息填充服务地址结构 `server`。

32：发送信息到服务器。必须提取 `server` 指针，并显示转换成 `sockaddr` 指针以满足 `sendto` 参数的定义，其实现的表达式为 `(struct sockaddr *)server`。

35~38：接收服务器发来的信息。`buf` 存放接收的信息，`reply` 中存放发送信息的主机地址。`numbytes` 是收到信息的字节数。

39~42：由于 UDP 套接字是无连接的，它可能接收到其他主机发来的信息，所以应判断信息是否来自相应的服务器。首先，比较地址长度 `len` 是否等于结构 `sockaddr` 长度。如果不是，则说明信息是来自其他服务器。然后判断 `server` 和 `reply` 变量内容是否一致。如果一致，则说明收到的信息是来自相应的服务器。注意，`server` 和 `reply` 首先应转换成常量指针才能使用 `memcmp` 函数进行比较。

43~44：显示来自服务器的信息。

47：关闭套接字。

### 5.3.3 运行程序

下面在同一主机上运行上述程序。

(1) 首先启动服务器程序：

```
udpserver
```

(2) 运行客户：

```
udpclient 127.0.0.1 hello
```

#### 1. 服务器的运行结果

```
$ udpserver
You got a message (hello) from 127.0.0.1
$
```

#### 2. 客户端运行结果

```
$ udpclient 127.0.0.1 hello
Server Message: Welcome to my server.
$
```

#### 3. 运行结果说明

运行结果说明本机客户能与本机 UDP 服务器通过 UDP 套接字相互发送信息。

## 5.4 小 结

由于 UDP 套接字是面向无连接的通信模式，其实现的基本过程更为简单，并且具有更

高的传输效率。通常用于提供短信息服务。

与 TCP 套接字相同，其实现也分为 UDP 服务器和 UDP 客户端。UDP 服务器的基本过程包括：创建套接字、绑定套接字、读/写数据和关闭套接字。UDP 客户端的基本过程包括：创建套接字、读/写数据和关闭套接字。

与 TCP 套接字不同，UDP 套接字读/写函数，`recvfrom()`和 `sendto()`函数都有与套接字地址相关的参数。对于 UDP 服务器而言，用该参数来区分不同的客户端。而对于客户端，则用此确定相应的服务器。这些在本章的实例中予以了说明。

# 第三部分 Unix 网络编程实用技术

## 第 6 章 并发服务器

前面的章节已经讲述了套接字实现的基本方法。然而，在实际应用中，一个软件仅能完成相应的功能是不够的，还必须有足够的处理能力及响应速度。对于服务器/客户体系的软件而言，服务器的性能则占据主导地位。而通过并发技术，能极大地提高服务器的处理能力和响应速度，因此本章将详细地讲述实现并发服务器的主要相关技术。

### 6.1 并发服务器基础

#### 6.1.1 服务器分类

##### 1. 按连接类型分类

服务器按连接类型划分，可分为：

- 面向连接的服务器
- 面向无连接的服务器

面向连接的服务器采用连接型的通信协议（如 TCP 协议），由于协议本身保证了数据的正确传输，在服务器实现时，不必考虑这方面的问题。服务器只需管理和使用连接。它接受客户端连接，并通过连接与客户进行信息交换，当与客户交互完成后，关闭连接。当连接建立后，TCP 提供所有机制来保证数据的可靠传输，如重传丢失的数据；进行数据校验并按顺序组装数据等。同时，TCP 也提供基本的方法，来建立和关闭连接。

面向连接的服务器也有一些不足。如每个连接都需要有相应的套接字，因而服务器需要管理这些套接字，并且这些套接字占用了系统资源。另外，每个连接都需要有建立和关闭过程，这将影响传输的效率和响应时间，并且增加了服务器的负担。

面向无连接的服务器采用无连接的通信协议，如 UDP 协议，协议本身并不保证数据的可靠传输，所以服务器应根据需求来提供相应的保证。由于不存在连接，具有较高的传输效率，占用较少的系统资源，并且服务不必管理连接套接字（一个服务器只需一个套接字）。

##### 2. 按处理方式分类

服务器按处理方式分类，划分为重复性服务器和并发服务器。重复性服务器每次只处理一个客户请求。当上一个客户请求处理完成后，才处理下一个客户请求。其效率较低但实现简单。并发服务器每次可处理多个客户请求。其效率很高却实现复杂。

在实际应用中，服务器要处理大量的客户请求，所有这些客户将访问绑定在某一特定套接字地址上的服务器。因此，服务器必须满足并发的需求。并发服务器的应用见图 6.1。

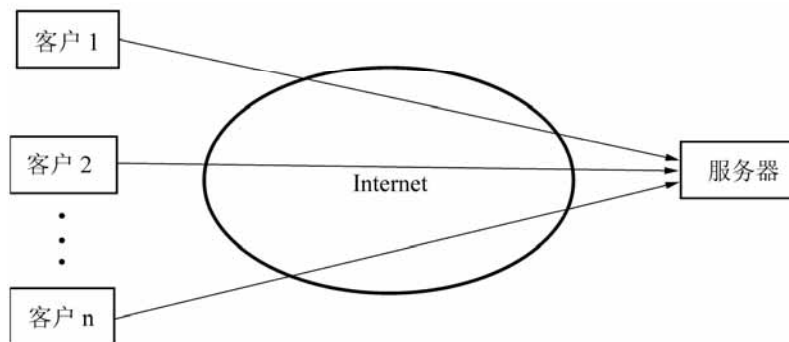


图 6.1 并发服务器的应用

如果不采用并发技术，当服务器处理一个客户请求时，会拒绝其他客户的请求，造成其他客户要不断地请求并长期等待。

### 6.1.2 重复性服务器实例

以下的实例，是采用重复性服务器算法实现的。通过该实例可了解重复性服务器是如何处理多客户请求的

#### 1. 实例功能

该实例包括服务器程序和客户程序，完成的功能如下。

- 服务器等候客户连接，一旦有连接则显示客户的地址，然后接收来自该客户的信息（字符串）。每当收到一个字符串，则显示该字符串，并将字符串反转，再将反转的字符发回客户。之后，继续等待接收该客户的信息直至该客户关闭连接。完成与该客户交互后，服务器开始等待下一客户请求，并重复上述过程。
- 客户首先与相应服务器连接。然后接收用户输入的字符串，再将字符串发送给服务器，接收服务器发回的信息并显示。之后，继续等待用户输入直至用户输入 Ctrl+D。当收到用户输入 Ctrl+D 后，客户关闭连接并退出。

#### 2. 服务器源程序清单（程序 6.1）

```

1  #include <stdio.h>           /* These are the usual header files */
2  #include <strings.h>         /* for bzero() */
3  #include <unistd.h>          /* for close() */
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>

8  #define PORT 1234           /* Port that will be opened */
9  #define BACKLOG 5           /* Number of allowed connections */
10 #define MAXDATASIZE 1000

11 void process_cli(int connectfd, sockaddr_in client);

```

```
12 main()
13 {
14     int listenfd, connectfd; /* socket descriptors */
15     struct sockaddr_in server; /* server's address information */
16     struct sockaddr_in client; /* client's address information */
17     int sin_size;

18     /* Create TCP socket */
19     if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
20         /* handle exception */
21         perror("Creating socket failed.");
22         exit(1);
23     }

24     int opt = SO_REUSEADDR;
25     setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

26     bzero(&server, sizeof(server));
27     server.sin_family=AF_INET;
28     server.sin_port=htons(PORT);
29     server.sin_addr.s_addr = htonl (INADDR_ANY);
30     if (bind(listenfd, (struct sockaddr *)&server,
31             sizeof(struct sockaddr)) == -1) {
32         /* handle exception */
33         perror("Bind error.");
34         exit(1);
35     }

36     if(listen(listenfd, BACKLOG) == -1){ /* calls listen() */
37         perror("listen() error\n");
38         exit(1);
39     }

40     sin_size=sizeof(struct sockaddr_in);
41     while (1) {
42         if ((connectfd = accept(listenfd,
43                                 (struct sockaddr *)&client, &sin_size)) == -1) {
44             perror("accept() error\n");
45             exit(1);
46         }
47         process_cli(connectfd, client);
48     }

49     close(listenfd); /* close listenfd */
```

```
48 }

49 void process_cli(int connectfd, sockaddr_in client)
50 {
51     int num;
52     char recvbuf[MAXDATASIZE], sendbuf[MAXDATASIZE];

    /* prints client's IP */
53     printf("You got a connection from %s\n",inet_ntoa(client.sin_addr) );

54     while (num = recv(connectfd, recvbuf, MAXDATASIZE,0)) {
55         recvbuf[num] = '\0';
56         printf("Received client message: %s",recvbuf);

57         for (int i = 0; i < num - 1; i++) {
58             sendbuf[i] = recvbuf[num - i -2];
59         }
60         sendbuf[num - 1] = '\0';

        /* send to the client welcome message */
61         send(connectfd,sendbuf,strlen(sendbuf),0);
62     }

63     close(connectfd); /* close connectfd */
64 }
```

### 3. 程序分析

1~7：所需的头文件。

8~10：定义端口号、最大允许连接的数量及缓冲区的大小。

11：声明“客户处理”process\_cli()函数。用于处理客户请求。

19~23：产生 TCP 套接字。

24~25：设置套接字选项为 SO\_REUSEADDR。

26~34：绑定套接字到相应地址。本例 IP 地址设为 INADDR\_ANY，因而可接收来自本机任何 IP 地址的客户连接。

35~38：监听网络连接。

39~46：重复接受客户连接。一旦连接成功，则调用“客户处理”process\_cli()函数。

53：显示客户 IP 地址，通过函数 inet\_ntoa 将客户地址转换成可显示的 IP 地址。

54~62：重复接收客户数据。一旦收到，则显示并反转字符串，然后发送反转的字符串回客户端。由于客户端字符串中含回车符，反转时应去掉。

61：关闭连接套接字。

### 4. 客户端源程序清单（程序 6.2）

```
1  #include <stdio.h>
```



```
2  #include <unistd.h>
3  #include <strings.h>
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <netdb.h>          /* netdb.h is needed for struct hostent =) */

8  #define PORT 1234    /* Open Port on Remote Host */
9  #define MAXDATASIZE 1000 /* Max number of bytes of data */

10 void process(FILE *fp, int sockfd);

11 int main(int argc, char *argv[])
12 {
13     int fd; /* files descriptors */
14     /* structure that will get information about remote host */
15     struct hostent *he;
16     struct sockaddr_in server; /* server's address information */

17     if (argc !=2) {
18         printf("Usage: %s <IP Address>\n",argv[0]);
19         exit(1);
20     }

21     if ((he=gethostbyname(argv[1]))==NULL){ /* calls gethostbyname() */
22         printf("gethostbyname() error\n");
23         exit(1);
24     }

25     if ((fd=socket(AF_INET, SOCK_STREAM, 0))==-1){ /* calls socket() */
26         printf("socket() error\n");
27         exit(1);
28     }

29     bzero(&server,sizeof(server));
30     server.sin_family = AF_INET;
31     server.sin_port = htons(PORT); /* htons() is needed again */
32     server.sin_addr = *((struct in_addr *)he->h_addr);

33     if(connect(fd, (struct sockaddr *)&server,
34               sizeof(struct sockaddr))==-1){ /* calls connect() */
35         printf("connect() error\n");
36         exit(1);
37     }

38     process(stdin,fd);
```

```
37 close(fd); /* close fd */
38 }

39 void process(FILE *fp, int sockfd)
40 {
41     char    sendline[MAXDATASIZE], recvline[MAXDATASIZE];
42     int numbytes;

43     printf("Connected to server. \n");
44     while (fgets(sendline, MAXDATASIZE, fp) != NULL) {
45         send(sockfd, sendline, strlen(sendline), 0);

46         if ((numbytes = recv(sockfd, recvline, MAXDATASIZE, 0)) == 0) {
47             printf("Server terminated.\n");
48             return;
49         }
50         recvline[numbytes] = '\0';
51         /* it prints server's welcome message */
52         printf("Server Message: %s\n", recvline);
53     }
```

## 5. 程序分析

1~7：所需的头文件。

8~9：定义了端口号和接收缓冲区的大小。注意端口号与服务器定义的要一致。本例的缓冲区是采用静态方式分配的。为了提高内存的使用效率可采用动态分配方式。

10：声明 process() 函数，用于连接后与服务器的交互。

16~19：检查用户输入。如果用户输入不正确，提示用户正确的输入方法。

20~23：通过字符串形式的 IP 地址获得服务器的地址信息。

24~27：产生 TCP 套接字。

28~35：连接到服务器。

36：调用 process() 函数，将标准输入和套接字作为参数。

37：关闭套接字。

44：重复获得用户输入的字符串，直到用户输入 CTRL+D。CTRL+D 表示文件结束，fgets() 函数读到该字符时返回 NULL。

45：发送用户输入的字符串到服务器。

46~52：接收并显示来自服务器的信息。如果接收的字节数为 0，说明服务器关闭了连接。

## 6. 运行程序

下面在同一主机上运行上述程序。

(1) 首先启动服务器程序：

```
itrserver
```

(2) 运行客户端 1:

```
itrclient 127.0.0.1
```

(3) 输入字符串“1234”，得到反转的显示为“4321”。

(4) 键入 Ctrl+D 退出客户端 1。通过这个试验，可看出服务器与客户端交互的整个过程。

(5) 下面再启动客户端 2 和客户端 3:

```
itrclient 127.0.0.1
```

```
itrclient 127.0.0.1
```

(6) 客户端 2 输入“5678”得到反转结果“8765”，再键入 Ctrl+D 退出。

(7) 客户端 3 输入“abcd”，则程序阻塞直到客户端 2 退出，才正常运行。由此可以看出重复性服务器在处理多个客户端时的不足。

## 7. 服务器的运行结果

```
$ itrserver
You got a connection from 127.0.0.1
Received client message: 1234
You got a connection from 127.0.0.1
Received client message: 5678
You got a connection from 127.0.0.1
Received client message: abcd
Received client message: efg
```

## 8. 客户 1 运行结果

```
$ itrclient 127.0.0.1
Connected to server.
1234
Server Message: 4321
$
```

## 9. 客户 2 运行结果

```
$ itrclient 127.0.0.1
Connected to server.
5678.
Server Message: 8765
$
```

## 10. 客户 3 运行结果

```
$ itrclient 127.0.0.1
Connected to server.
```

```
abcd.  
Server Message: dcba  
efg  
Server Message: gfe  
$
```

### 11. 运行结果说明

运行结果说明重复性服务器可处理多个客户，但效率极低。

## 6.1.3 并发技术

并发技术的执行取决于操作系统提供的基本设施。Unix 系统主要提供三种方式以支持并发：进程、线程及 I/O 多路复用。

### 1. 进程

进程是执行中的计算机程序，可以认为是一个程序的一次运行。它是一个动态实体，是独立的任务。每个单独的进程运行在自己的虚拟地址空间，并且只能通过安全的内核管理机制和其他进程交互。若一个进程崩溃并不会引起系统中另一个进程崩溃。

在 Unix 系统中，多个进程可以同时执行相同的代码，从而支持并发。

对于单 CPU 系统而言，CPU 一次只能执行一个进程，但操作系统可通过分时处理，每个进程在每个时间段中执行，因此对于用户而言，这些进程是在同时执行。

### 2. 线程

线程与进程类似，也支持并发执行。与进程不同，在同一进程中所有线程共享相同的全程变量以及系统分配给进程的资源。因此，线程占用较少的系统资源，并且线程之间的切换更快。

### 3. I/O 多路复用

另一种支持并发的方法，是 I/O 多路复用。select() 函数是系统提供的，它可在多个描述符中选择被激活的描述符进行操作。

例如，一个进程同时有多个客户连接，即存在多个 TCP 套接字描述符。select() 函数阻塞直到任何一个描述符被激活，即有数据传输。从而避免了进程为等待一个已连接上的数据而无法处理其他连接。因而，这是一种时分复用的方法，从用户角度而言，它实现了在一个进程或线程中的并发处理。

## 6.1.4 并发服务器算法

并发服务器通常通过多进程/线程实现，但也可以在单个进程/线程中实现。进程/线程可分为主服务器进程/线程和子服务器进程/线程。主进程/线程创建套接字并等待客户请求，当收到客户请求则产生子进程/线程，进行处理。主进程/线程并不直

接与客户通信，而是交给子进程/线程来完成。当子进程/线程完成与客户的交互后，则退出执行。

以下介绍三种主要的并发服务器算法。

### 1. 并发无连接服务器算法

其算法见图 6.2 所示。

由于是无连接服务器，不需要等待每个已连接上的数据，因而不必采用并发技术也能有效地处理多个用户，另外，多进程/线程需要大量的系统开销，因此，无链接器服务器一般不采用并发技术。

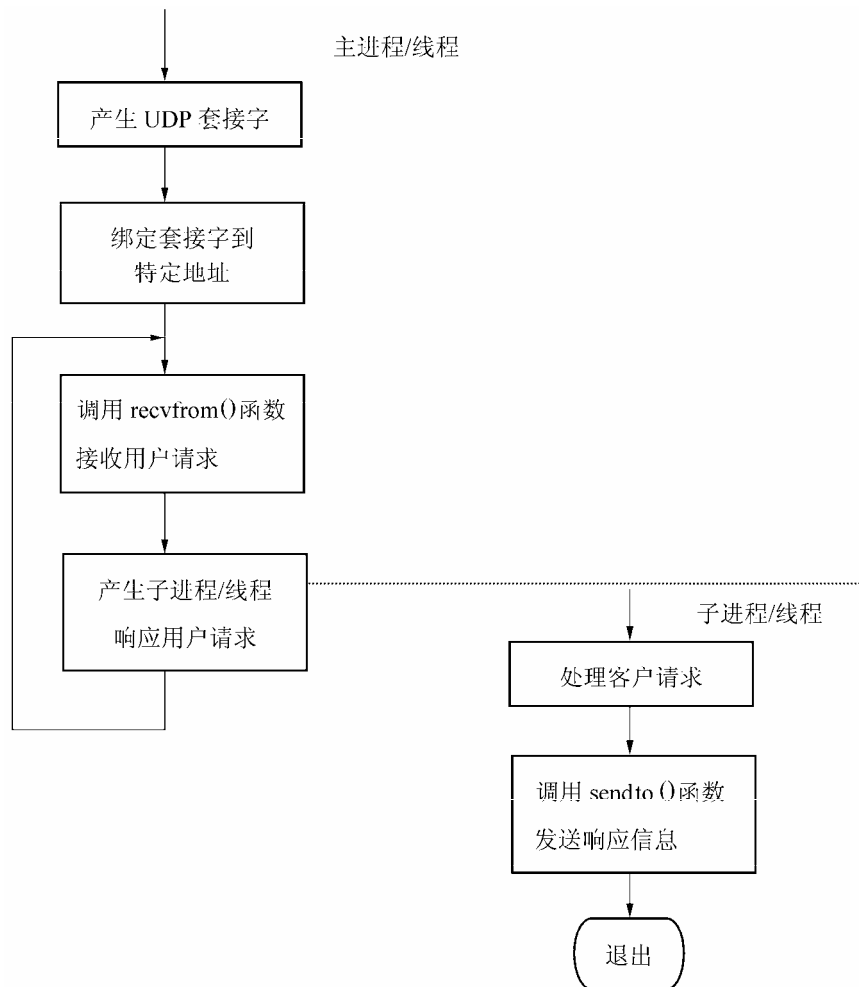


图 6.2 并发无连接服务器算法

### 2. 并发连接服务器算法

面向连接的服务器与客户的通信是建立在连接的基础上。客户首先与服务器建立连接，

然后双方在连接的基础上进行交互，交互结束后断开连接。由于连接服务器需要守候在每个连接上，因此，并发是非常重要的。并发连接服务器算法见图 6.3 所示。

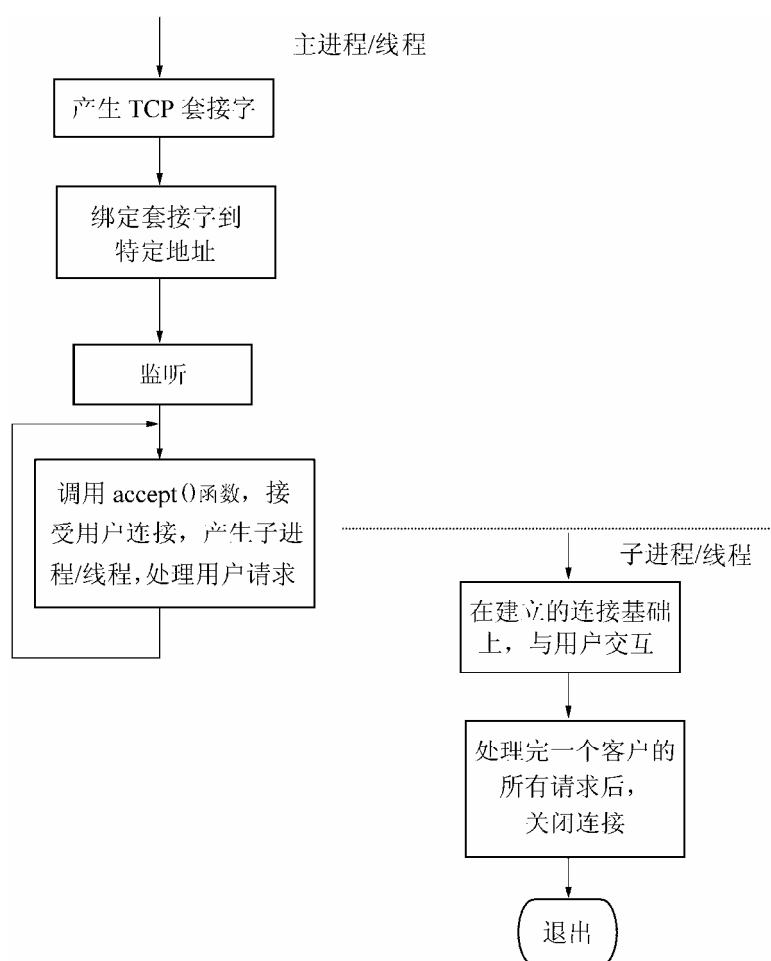


图 6.3 并发连接服务器算法

### 3. 单进程/线程的并发服务器算法

由于进程/线程需要较多的系统开销，有时可以采用多路复用 I/O 来实现处理多个客户连接，而不必产生多个进程/线程，其算法见图 6.4 所示。

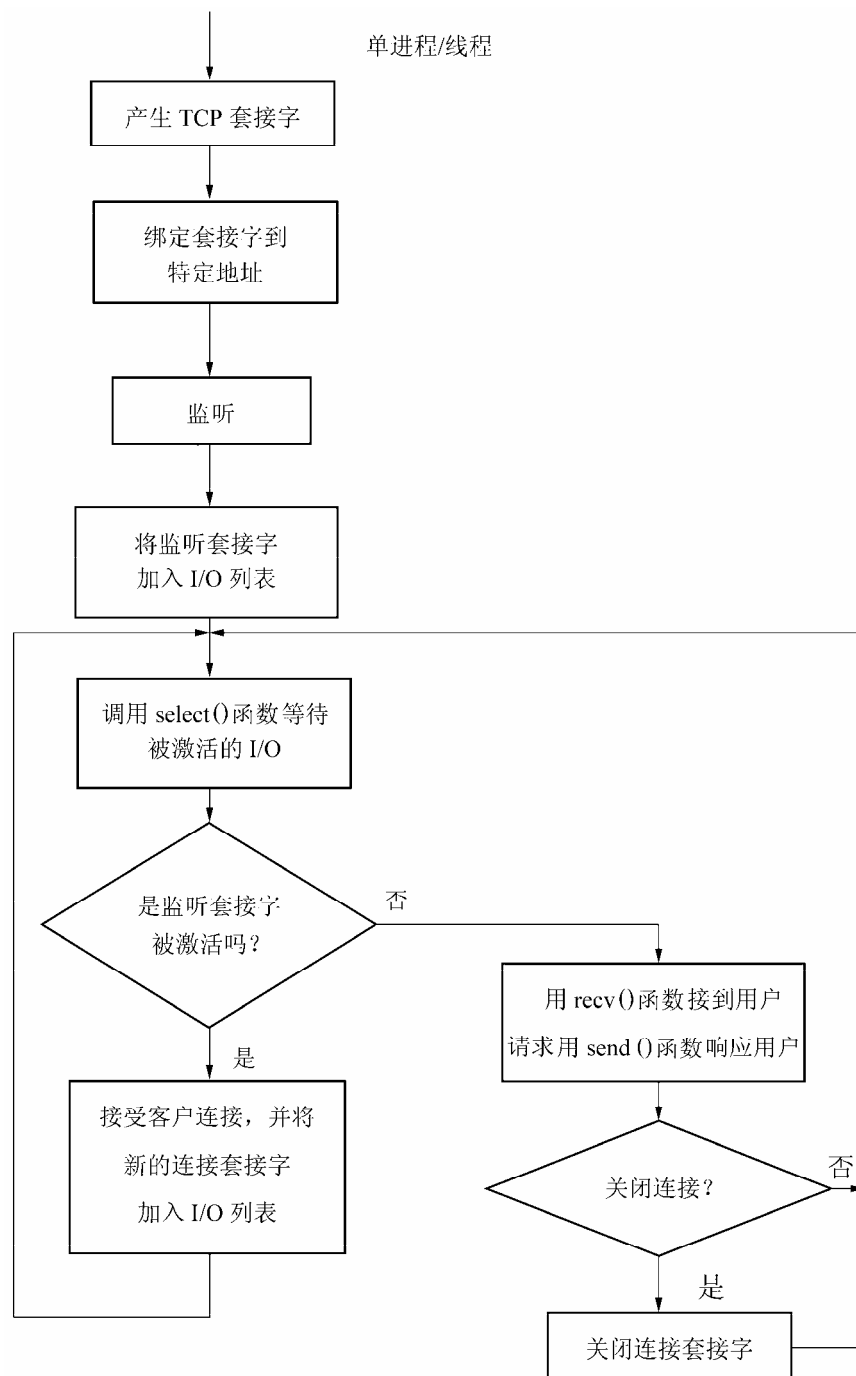


图 6.4 单进程/线程的并发服务器算法

## 6.2 多进程服务器

### 6.2.1 进程概念

进程定义了一个计算的基本单元，它是一个执行某一特定程序的实体。它拥有自己的地址空间、执行堆栈、文件描述符表等。

一个进程不是一个程序。几个进程可能为一个用户或多个用户同时执行相同的计算和程序。可以用 PS 命令查看当前系统运行的进程。

每个进程拥有独立的地址空间，相互并不影响，一个进程崩溃并不会造成其他进程的崩溃。但不同的进程可能维护相同的系统资源（如文件），因而对进程而言，存在可再入性问题。可再入性是指一段代码或一相同功能同时被多次执行。例如，某一段代码是打开一文件并写入数据。当多个进程同时执行该段代码时，是以无法预计的方法向该文件写入数据，可能会造成该文件内容的损坏。因此，我们称这段代码是不可再入的。在多进程环境中应注意这种可再入性。

每个进程都有一个非负整型的惟一进程 ID。因为进程 ID 标识符总是惟一的，常将其用作其他标识符的一部分以保证其惟一性。

### 6.2.2 创建进程

现有进程调用 fork() 函数是创建一个新进程的惟一方法。

#### 1. 函数原型

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

pid_t fork(void);
```

#### 2. 返回值

子进程中为 0，父进程中为子进程 ID，出错为 -1。

#### 3. 子进程与父进程

由 fork() 函数创建的新进程被称为子进程。该函数被调用一次，但返回两次。两次返回的区别是子进程的返回值是 0，而父进程的返回值则是新子进程的进程 ID。将子进程 ID 返回给父进程的理由是：因为一个进程的子进程可以多于一个，但没有一个函数使一个进程可以获得其所有子进程的进程 ID。Fork() 函数使子进程得到返回值 0 是由于：一个进程只有一个父进程，所以子进程总是可以调用 getpid() 函数以获得其父进程的进程 ID。

子进程和父进程继续执行 fork() 函数之后的指令。子进程是父进程的副本。例如，子进程获得父进程的数据空间、堆栈。注意，这是子进程所拥有的副本。父、子进程并不共享这些存储空间部分。如果代码段是只读的，则父、子进程共享代码段。

一般情况下，在调用 fork() 函数之后是父进程先执行，还是子进程先执行是不确定的。这



取决于系统内核所使用的调度算法。要使父、子进程同步，则需要某种形式的进程间通信。

Fork()函数的一个特性是所有由父进程打开的描述符都复制到子进程中。父、子进程每个相同的描述符共享一个文件表项。如果父、子进程写到同一描述符文件，但又没有任何形式的同步(例如使父进程等待子进程)，那么它们的输出就会相互混合(假定所用的描述符是在 fork()函数之前打开的)。

有两种常见的在 fork()函数之后处理文件描述符的情况。

- 父进程等待子进程完成。在这种情况下，父进程无需对其描述符作任何处理。当子进程终止后，被它进行过读、写操作的任一共享描述符的文件位移量也做了相应更新。
- 父、子进程各自执行不同的程序段。在这种情况下，在 fork()函数之后，父、子进程各自关闭它们不需要使用的文件描述符，并且不干扰对方使用的文件描述符。这种方法是网络服务进程常常使用的。

子进程继承父进程的大部分属性，主要包括：

- 实际 UID、GID 和有效 UID、ID
- 环境变量
- 附加 GID
- 调用 exec()函数时的关闭标志
- UID 设置模式位
- GID 设置模式位
- 进程组号
- 会话 ID
- 控制终端
- 当前工作目录
- 根目录
- 文件创建掩码 UMASK
- 文件长度限制 ULIMIT
- 预定值，如优先级和任何其他进程预定参数，根据种类不同而不同
- 决定是否可以继承
- 其他一些属性

但子进程也有与父进程不同的属性：

- 进程号，子进程号不同于任何一个活动的进程组号
- 父进程号
- 子进程继承父进程的文件描述符或流时，具有自己的一个拷贝并且与父进程和其他子进程共享该资源
- 子进程的用户时间和系统时间被初始化为 0
- 子进程的超时时钟设置为 0
- 子进程的信号处理函数指针组置为空
- 子进程不继承父进程的记录锁

#### 4. Fork()函数调用失败的原因

使 fork()函数调用失败的两个主要原因是：

- 系统中已经有了太多的进程；
- 该实际用户 ID 的进程总数超过了系统限制。

#### 5. Fork()函数的用法

fork()函数有两种用法。

- 一个父进程通过复制自己，使父、子进程同时执行不同的代码段。这对网络服务进程是常见的：父进程等待客户的服务请求。当这种请求到达时，父进程调用 fork()函数，使子进程处理此请求。父进程则继续等待下一个客户服务请求。
- 每个进程要执行不同的程序。这对 Shell 是常见的情况。在这种情况下，子进程在从 fork()函数返回后立即调用 exec()函数执行其他程序。

用法如下：

```
1 pid_t pid;
2 if ((pid=fork())>0) {
3     /*parent process*/
4 }
5 else if (pid==0) {
6     /*child process*/
7     exit(0);    /*child must exit with exit()*/
8 }
9 else {
10    printf("fork error\n");
11    exit(0);
12 }
```

代码说明如下。

2：产生子进程。

3：父进程处理过程。

5~7：子进程处理过程。

9~11：出错处理。

### 6.2.3 终止进程

进程的终止存在两种可能：

- 父进程先于子进程终止
- 子进程先于父进程终止

如果父进程在子进程之前终止，则所有子进程的父进程改变为 init 进程。我们称这些进程由 init 进程领养。其操作过程大致是：在一个进程终止时，系统核逐个检查所有活动进程，以判断这些进程是否是正要终止的进程的子进程。如果是，则该进程的父进程 ID 就更改为 1(init 进程的 ID)。这种处理方法保证了每个进程有一个父进程。

如果子进程在父进程之前终止，系统内核为每个终止子进程保存了一定量的信息，这

样当终止进程的父进程调用 `wait()` 函数或 `waitpid()` 函数时, 就可以得到有关信息。这些信息包括进程 ID、该进程的终止状态、以及该进程使用的 CPU 时间总量。系统内核可以释放终止进程所使用的所有存储器, 关闭其所有打开文件。在 Unix 术语中, 一个已经终止、但是其父进程尚未对其进行善后处理(例如获取终止子进程的有关信息、释放它仍占用的资源)的进程被称为僵死进程。

当一个进程正常或异常终止时, 系统内核就向其父进程发送 `SIGCHLD` 信号。因为子进程终止是异步的(即可以在父进程运行的任何时候发生), 所以这种信号也是系统内核向父进程发的异步通知。父进程可以忽略该信号, 或者提供一个该信号发生时即被调用的函数(信号处理程序)。对于这种信号的系统默认动作是忽略它。

调用 `wait` 或 `waitpid()` 函数时, 可能会有以下几种情况。

- 阻塞 (如果其所有子进程都还在运行)。
- 获得子进程的终止状态并立即返回 (如果一个子进程已终止, 正等待父进程存取其终止状态)。
- 出错立即返回 (如果它没有任何子进程)。

以下介绍进程终止所调用的函数。

### 1. `wait()` 函数

等待一个子进程返回并修改状态。

原型:

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

pid_t wait(int *stat_loc);
```

描述:

允许调用进程取得子进程的状态信息。调用进程将会挂起直到其一个子进程终止。

返回值:

等待直到一个子进程终止时, 返回值为该子进程进程号, 否则返回值为 -1。同时 `stat_loc` 返回子进程的返回值。

用法:

```
1 pid_t pid;
2 if ((pid=fork())>0) {
3     /*parent process*/
4     int child_status;
5     wait(&child_status);
6 }
7 else if (pid==0) {
8     /*child process*/
9     exit(0);
10 }
11 else {
```

```
12  printf("fork error\n");
13  exit(0);
14  }
```

代码说明如下。

2：产生子进程。

3：父进程处理过程。

4~5：等待子进程退出。

7~9：子进程处理过程。

12~13：出错处理。

## 2. waitpid()函数

等待指定进程的子进程返回，并修改状态。

原型：

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

```
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

描述：

当 pid 等于-1,options 等于 0 时，该函数等同于 wait()函数。否则该函数的行为由参数 pid 和 options 决定。

pid 指定了父进程要求知道哪些子进程的状态，其中：

- -1 要求知道任何一个子进程的返回状态；
- > 0 要求知道进程号为 pid 的子进程的状态；
- < -1 要求知道进程组号为 pid 的绝对值的子进程状态。

options 参数为以位方式表示的标志，它是各个标志以“或”运算组成的位图，每个标志以字节中某个位置 1 表示：

- WUNTRACED 报告任何未知但已停止运行的指定进程的子进程状态，该子进程的状态自停止运行时起就没有被报告过。
- WCONTINUED 报告任何继续运行的指定进程的子进程状态，该子进程的状态自继续运行起就没有被报告过。
- WNOHANG 若调用本函数时,指定进程的子进程状态，目前并不是立即有效的(即可被立即读取的)，调用进程不被挂起执行。
- WNOWAIT 保持那些将其状态设置在 stat\_loc()函数的进程处于可等待状态。该进程将等待直到下次被要求其返回状态值。

返回值：

当一个子进程返回时,返回值为该子进程号,否则返回值为-1。同时 stat\_loc()函数返回子进程的返回值。

用法：

```
1 pid_t pid;
```

```
2 if ((pid=fork())>0) {
3     /* parent process */
4     int child_status;
5     pid_t child_pid;
6     waitpid(child_pid,&child_status,0); /* parent waits for child to return */
7 }
8 else if (pid==0) {
9     /* child process */
10    exit(0);
11 }
12 else {
13    printf("fork error\n");
14    exit(0);
15 }
```

代码说明如下。

2：产生子进程。

3：父进程处理过程。

4~6：等待子进程退出。

8~10：子进程处理过程。

12~14：出错处理。

### 3. exit()函数

终止进程，并返回状态。

原型：

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

```
void exit(int status);
```

描述：

本函数终止调用进程。exit()函数会关闭所有子进程打开的描述符，向其父进程发送SIGCHLD信号，并返回状态，父进程可通过调用wait()或waitpid()函数获得其状态。

返回值：

无。

## 6.2.4 多进程并发服务器

采用多进程实现并发服务器是一种常见而且有效的方法。当连接建立后，服务器调用fork()函数产生新的子进程，子进程处理客户请求，同时父进程关闭连接套接字，然后等待另一客户的连接。子进程首先关闭监听套接字，然后处理客户请求，最后关闭连接套接字并退出进程。

典型的模板如下：

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

```
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <unistd.h>
5 #include <stdlib.h>

6 main()
7 {
8     int listenfd, connfd;
9     pid_t pid;
10    int BACKLOG = 5;

11    /* Create TCP socket */
12    if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
13        perror("Creating socket failed");
14        exit(1);
15    }
16    /* Bind socket to address */
17    bind(listenfd, ...);
18    /* listen client's requirement */
19    listen(listenfd, BACKLOG);
20    while(1)
21    {
22        /* accept connection*/
23        if ((connfd = accept(listenfd, NULL, NULL)) == -1) {
24            perror("Acceptation failed.");
25            exit(1);
26        }
27        /* create child process to service the client. */
28        if ((pid=fork())>0) {
29            /*parent process*/
30            close(connfd);
31            ...
32            continue;
33        }
34        else if (pid==0) {
35            /*child process*/
36            close(listenfd);
37            ...
38            exit(0);
39        }
40        else {
41            printf("fork error\n");
42            exit(0);
43        }
44    }
```

```
45 }
```

代码说明如下。

1~5：所需的头文件。

10：定义允许连接的最大数量。

11~15：产生 TCP 套接字。

17：把套接字绑定到相应地址。

19：监听网络连接。

23~26：接受客户连接。

28~32：父进程处理过程。首先关闭连接套接字，再进行客户处理。

34~39：子进程处理过程。首先关闭监听套接字。

41~42：如果子进程产生失败，显示出错信息并退出程序。

从以上模板可看出，产生新的子进程后，父进程要关闭连接套接字，而子进程关闭监听套接字。这样做主要基于以下两点原因。

- 由于父进程只负责接收客户请求，而子进程只负责处理客户请求，关闭不需要的套接字可节省系统资源。并且，父进程和子进程共享这些套接字，如果它们都对这些套接字进行操作，会产生不可预计的后果。
- 另一个原因，也是更重要的原因，是为了正确地关闭连接。和文件描述符一样，每个套接字描述符都有一个“引用计数”，“引用计数”是指同时打开一描述符的次数。例如一个文件被同时打开两次，则“引用计数”为 2。根据以上模板，当 `socket()` 函数返回后，`listenfd` 的“引用计数”变为 2，而系统只有在某描述符的“引用计数”为 0 时，才真正关闭该描述符，对于连接套接字而言，系统此时才真正关闭该连接。当父进程关闭 `connfd` 时，`connfd` 的“引用计数”减 1，当子进程关闭 `connfd` 时，`connfd` 的“引用计数”减为 0，此时系统才会关闭该连接。

以下用图例说明上述过程。当服务器调用 `accept()` 函数，阻塞并等待客户连接时，其状态如图 6.5 所示。

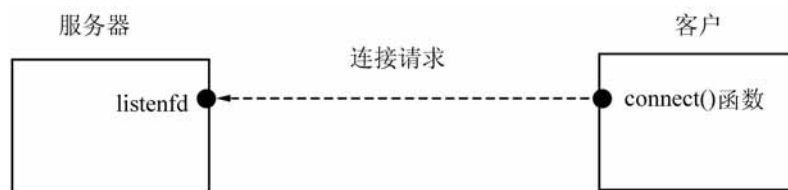
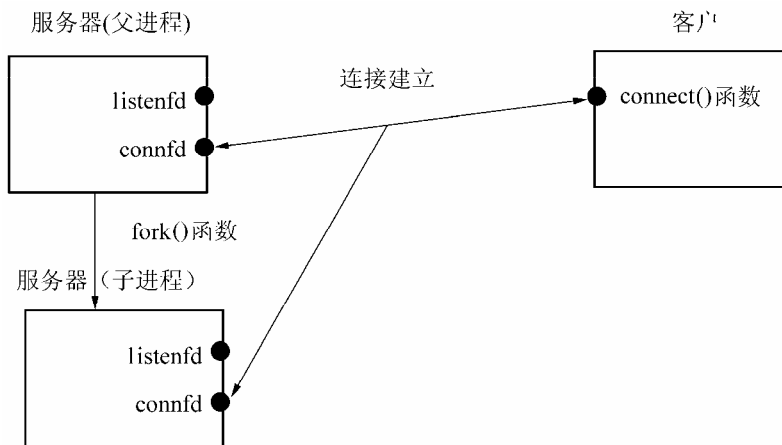


图 6.5 客户/服务器状态图（调用 `accept()` 函数时）

当 `accept()` 函数返回后，服务器接受了客户连接并产生新的连接套接字 `connfd`，此时状态见图 6.6 所示。

图 6.6 客户/服务器状态图 ( `accept()` 函数返回后 )

然后服务器调用 `fork()` 函数，产生新的子进程，此时，父进程和子进程共享 `listenfd` 和 `connfd`，其状态见图 6.7 所示。

图 6.7 客户/服务器状态图 ( `fork()` 函数返回后 )

然后，父进程关闭连接套接字，子进程关闭监听套接字，其状态见图 6.8 所示。

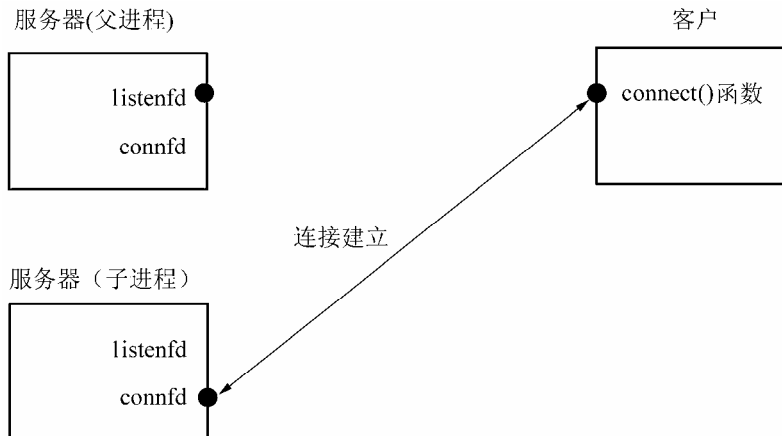


图 6.8 客户/服务器状态图 ( 父进程关闭连接套接字，子进程关闭监听套接字后 )

此时子进程可通过该连接处理客户请求，而父进程继续监听新的客户请求。



### 6.2.5 多进程并发服务器实例

以下的实例，是采用多进程并发服务器算法来实现的。通过该实例可了解多进程并发服务器是如何处理多客户的，以及其性能情况。

该实例包括服务器程序和客户程序。完成的功能与重复性服务器实例类似，具体功能如下。

- 服务器等候客户连接请求，一旦连接成功则显示客户的地址，接着接收该客户的名字并显示。然后接收来自该客户的信息（字符串）。每当收到一个字符串，则显示该字符串，并将字符串反转，再将反转的字符发回客户端。之后，继续等待接收该客户的信息直至该客户关闭连接。服务器具有同时处理多个客户的能力。
- 客户首先与相应服务器连接。接着接收用户输入的客户名字，再将名字发送给服务器。然后接收用户输入的字符串，再将字符串发送给服务器，接收服务器发回的信息并显示。之后，继续等待用户输入直至用户输入 Ctrl+D。当收到用户输入 Ctrl+D 后，客户关闭连接并退出。

#### 1. 服务器源程序清单（程序 6.3）

```
1  #include <stdio.h>           /* These are the usual header files */
2  #include <strings.h>         /* for bzero() */
3  #include <unistd.h>          /* for close() */
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>

8  #define PORT 1234           /* Port that will be opened */
9  #define BACKLOG 2           /* Number of allowed connections */
10 #define MAXDATASIZE 1000
11 void process_cli(int connectfd, struct sockaddr_in client);

12 main()
13 {
14     int listenfd, connectfd; /* socket descriptors */
15     pid_t pid;
16     struct sockaddr_in server; /* server's address information */
17     struct sockaddr_in client; /* client's address information */
18     int sin_size;

19     /* Create TCP socket */
20     if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
21         /* handle exception */
22         perror("Creating socket failed.");
23         exit(1);
```

```
24  }

25  int opt = SO_REUSEADDR;
26  setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

27  bzero(&server, sizeof(server));
28  server.sin_family=AF_INET;
29  server.sin_port=htons(PORT);
30  server.sin_addr.s_addr = htonl (INADDR_ANY);
31  if (bind(listenfd, (struct sockaddr *)&server,
           sizeof(struct sockaddr)) == -1) {
32      /* handle exception */
33      perror("Bind error.");
34      exit(1);
35  }

36  if(listen(listenfd, BACKLOG) == -1){ /* calls listen() */
37      perror("listen() error\n");
38      exit(1);
39  }

40  sin_size=sizeof(struct sockaddr_in);

41  while(1)
42  {
43      /*accept connection */
44      if ((connectfd = accept(listenfd, (struct sockaddr *)&client,
                             &sin_size))== -1) {
45          perror("accept() error\n");
46          exit(1);
47      }
48      /* Create child process to service client */
49      if ((pid=fork())>0) {
50          /* parent process */
51          close(connectfd);
52          continue;
53      }
54      else if (pid==0) {
55          /*child process*/
56          close(listenfd);
57          process_cli(connectfd, client);
58          exit(0);
59      }
60      else {
61          printf("fork error\n");
```

```
62     exit(0);
63 }
64 }
65 close(listenfd); /* close listenfd */
66 }

67 void process_cli(int connectfd, sockaddr_in client)
68 {
69     int num;
70     char recvbuf[MAXDATASIZE], sendbuf[MAXDATASIZE],
        cli_name[MAXDATASIZE];
71     /* prints client's IP */
72     printf("You got a connection from %s. ",inet_ntoa(client.sin_addr) );
73     /* Get client's name from client */
74     num = recv(connectfd, cli_name, MAXDATASIZE,0);
75     if (num == 0) {
76         close(connectfd);
77         printf("Client disconnected.\n");
78         return;
79     }
80     cli_name[num - 1] = '\0';
81     printf("Client's name is %s.\n",cli_name);

82     while (num = recv(connectfd, recvbuf, MAXDATASIZE,0)) {
83         recvbuf[num] = '\0';
84         printf("Received client( %s ) message: %s",cli_name, recvbuf);
85         for (int i = 0; i < num - 1; i++) {
86             sendbuf[i] = recvbuf[num - i -2];
87         }
88         sendbuf[num - 1] = '\0';

89         /* send to the client welcome message */
90         send(connectfd,sendbuf,strlen(sendbuf),0);
91     }
92     close(connectfd); /* close connectfd */
93 }
```

## 2. 程序分析

1~7：所需的头文件。

8~10：定义端口号、最大允许连接的数量及缓冲区的大小。

11：声明“客户处理”process\_cli()函数。用于处理客户请求。

19~23：产生TCP套接字。

25~26：设置套接字选项为SO\_REUSEADDR。

27~35：绑定套接字到相应地址。本例IP地址设为INADDR\_ANY，则可接收来自本机

任何 IP 地址的客户连接。

36~39：监听网络连接。

41~66：接受客户连接请求。一旦连接成功，产生子进程服务客户。然后关闭连接套接字并继续接受下一客户连接。

54~59：子进程首先关闭监听套接字，再调用客户处理函数 `process_cli()` 处理客户请求。完成后退出子进程。`exit()` 函数会关闭所有由子进程打开的描述符。

60~63：如果子进程产生失败，显示出错信息并退出程序。

71：显示客户 IP 地址，通过函数 `inet_ntoa()` 函数将客户地址转换成可显示的 IP 地址。

73~80：接收客户名字并显示。

81~89：重复接收客户数据。一旦收到，则显示并反转字符串，然后发送反转的字符串回客户。由于客户字符串中含回车符，反转时应去掉。

90：关闭连接套接字。

### 3. 客户源程序清单（程序 6.4）

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <strings.h>
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <netdb.h>          /* netdb.h is needed for struct hostent =) */

8  #define PORT 1234    /* Open Port on Remote Host */
9  #define MAXDATASIZE 100 /* Max number of bytes of data */
10 void process(FILE *fp, int sockfd);
11 char* getMessage(char* sendline,int len, FILE* fp);

12 int main(int argc, char *argv[])
13 {
14     int fd; /* files descriptors */
15     /* structure that will get information about remote host */
16     struct hostent *he;
17     struct sockaddr_in server; /* server's address information */

18     if (argc !=2) {
19         printf("Usage: %s <IP Address>\n",argv[0]);
20         exit(1);
21     }

22     if ((he=gethostbyname(argv[1]))==NULL){ /* calls gethostbyname() */
23         printf("gethostbyname() error\n");
24         exit(1);
25     }
```

```
25  if ((fd=socket(AF_INET, SOCK_STREAM, 0))==-1){ /* calls socket() */
26      printf("socket() error\n");
27      exit(1);
28      }

29  bzero(&server,sizeof(server));
30  server.sin_family = AF_INET;
31  server.sin_port = htons(PORT); /* htons() is needed again */
32  server.sin_addr = *((struct in_addr *)he->h_addr);

33  if(connect(fd, (struct sockaddr *)&server,
34      sizeof(struct sockaddr))==-1){ /* calls connect() */
35      printf("connect() error\n");
36      exit(1);
37      }

38  process(stdin,fd);

39  close(fd); /* close fd */
40  }

41  void process(FILE *fp, int sockfd)
42  {
43      char    sendline[MAXDATASIZE], recvline[MAXDATASIZE];
44      int numbytes;

45      printf("Connected to server. \n");
46      /* send name to server */
47      printf("Input name : ");
48      if ( fgets(sendline, MAXDATASIZE, fp) == NULL) {
49          printf("\nExit.\n");
50          return;
51      }
52      send(sockfd, sendline, strlen(sendline),0);

53      /* send message to server */
54      while (getMessage(sendline, MAXDATASIZE, fp) != NULL) {
55          send(sockfd, sendline, strlen(sendline),0);

56          if ((numbytes = recv(sockfd, recvline, MAXDATASIZE,0)) == 0) {
57              printf("Server terminated.\n");
58              return;
59          }
60      }
```

```
59     recvline[numbytes]='\0';
/* it prints server's welcome message */
60     printf("Server Message: %s\n",recvline);

61     }
62     printf("\nExit.\n");
63     }

64     char* getMessage(char* sendline,int len, FILE* fp)
65     {
66         printf("Input string to server:");
67         return(fgets(sendline, MAXDATASIZE, fp));
68     }
```

#### 4. 程序分析

1~7：所需的头文件

8~9：定义了端口号和接收缓冲区的大小。注意端口号与服务器定义的要一致。本例的缓冲区是采用静态方式分配的，为提高内存的使用效率可采用动态分配方式。

10：声明 process()函数，用于连接后与服务器的交互。

11：声明 getMessage()函数，用于提示并获得客户输入字符串。

17~20：检查用户输入。如果用户输入不正确，提示用户正确的输入方法。

21~24：通过字符串形式的 IP 地址获得服务器的地址信息。

25~28：产生 TCP 套接字

29~36：连接到服务器

37：调用 process()函数，将标准输入和套接字作为参数。

38：关闭套接字。

46~51：提示客户输入客户的名字，并发送给服务器。如果客户输入 CTRL+D，则退出程序。

53：重复从标准输入获得用户输入的字符串直到用户输入 CTRL+D。

54：发送用户输入的字符串到服务器。

55~61：接收并显示来自服务器的信息。如果接收的字节数为 0，说明服务器关闭了连接。

66~67：提示并获得客户输入字符串。

#### 5. 运行过程

下面在同一主机上运行上述程序。

(1) 首先启动服务器程序 (procserver)。

(2) 然后同时运行客户 1 (procclient 127.0.0.1) 和客户 2 (procclient 127.0.0.1)。

#### 6. 服务器运行结果

```
$ procserver
You got a connection from 127.0.0.1. Client's name is client1.
```

```
Received client( client1 ) message: 1234
You got a connection from 127.0.0.1. Client's name is client2.
Received client( client2 ) message: abcd
Received client( client1 ) message: 5678
Received client( client2 ) message: efgh
^C$
```

### 7. 客户 1 运行结果

```
$ procclient 127.0.0.1
Connected to server.
Input name : client1
Input string to server:1234
Server Message: 4321
Input string to server:5678
Server Message: 8765
Input string to server: ^D
Exit.
$
```

### 8. 客户 2 运行结果

```
$ procclient 127.0.0.1
Connected to server.
Input name : client2
Input string to server:abcd
Server Message: dcba
Input string to server:efgh
Server Message: hgfe
Input string to server: ^D
Exit.
$
```

### 9. 运行结果说明

从运行结果可看出，两客户均能及时地获得服务。与重复性服务器相比，在处理多个客户时性能有很大的提高。

## 6.3 多线程服务器

本节介绍采用 POSIX 线程库的多线程编程方法，并详细讲述如何采用多线程实现并发服务器。

### 6.3.1 线程基础

线程是提高代码响应和性能的有力手段。线程类似于进程。与进程一样，线程由系统内核按时间分片进行管理。在单处理器系统中，系统内核使用时间分片来模拟线程的并发执行，这种方式与进程相同。而在多处理器系统中，如同多个进程，线程同样可以并发执行。

同一进程可包括多个线程，这些线程共享相同的内存空间。不同的线程可以存取相同的全局变量、相同的堆数据和文件描述符等。所以，程序中的所有线程都可以读或写已声明的全局变量。而进程都有各自独立的内存空间，进程之间的通信需要专门的机制（IPC），这增加了某种形式的额外系统内核开销，从而降低性能。线程带来的开销很小。系统内核无需单独复制进程的内存空间或文件描述符等等。这就节省了大量的 CPU 时间，使得线程的创建比进程的创建快 10~100 倍。因为这一点，你可以大量使用线程而无需太过于担心带来的 CPU 或内存不足。

另一方面，由于线程共享相同的内存空间，如果一个线程崩溃，它将影响同一进程中的其他线程。线程共享所有的全局变量，虽然使得线程间交换信息极为容易，但同时需要考虑同步的问题。

同一进程中的线程共享如下内容：

- 全局变量
- 堆数据
- 打开的文件描述符
- 当前工作目录
- 用户及用户组 ID

但每个线程具有独立的：

- 线程 ID
- 堆栈
- errno 变量
- 优先级

### 6.3.2 线程函数调用(POSIX)

#### 1. pthread\_create()函数

当程序开始运行时，系统产生一个主线程。如果要产生其他线程，需要调用 pthread\_create()函数。其函数原型为：

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine)( void*), void *arg);
```

thread：指向线程 ID 的指针，其类型为 pthread\_t。该 ID 用于惟一标志该线程。

attr：指向线程属性的指针。线程的属性包括：线程优先级，堆栈大小等。如果 attr 的



值为 NULL，则表示使用系统默认的属性。由 attr 定义的属性，如果在调用 pthread\_create() 后修改，则不会影响线程的属性。

start\_routine：指向线程执行的函数。该函数必须是一个静态函数，它只有一个通用指针作为参数，并返回一个通用指针。

arg：是一通用指针，用于往 start\_routine() 传递参数。由于只有一个参数传递给 start\_routine() 函数，因此，当需要传递多个参数时，可将这些参数封装在一个结构中，然后把该 arg 指向该结构。

返回值：如果成功调用则返回 0，否则返回如下错误码。

- ENOMEM：没有足够内存产生另一线程。
- EINVAL：无效的 attr 值。
- EPERM：调用者无权设置调度参数或调度策略。

用法：

```
1 #include <pthread.h>      /* pthread functions and data structures */
2 pthread_t  thread;
3 type arg;
4 /* function to be executed by the new thread */
5 void* start_routine(void* arg);

6 if (pthread_create(&thread, NULL, start_routine, (void*)&arg)) {
7     /* handle exception */
8     ...
9     exit(1);
10 }
11 ...
```

代码说明如下。

1：所需头文件。

4~5：定义线程所执行的函数。

6：产生线程。

7~9：出错处理。

## 2. pthread\_join() 函数

该函数挂起当前线程直到所等待的线程结束。与进程的 waitpid() 函数功能类似。其函数原型如下：

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **value_ptr);
```

thread：所等待线程的 ID。该线程必须是当前进程的成员，并且不是分离的线程或守护线程。几个线程不能等待同一线程完成。如果其中一个成功调用 pthread\_join() 函数，则其他线程将返回 ESRCH 错误。另外，如果所等待的线程已经终止，则该函数将立即返回。

value\_ptr：如果该值非空，则指向终止线程的退出状态值。

返回值：如果成功调用则返回 0，否则返回如下错误码。

- EINVAL：线程 ID 无效。
- ESRCH：无法找到相应 ID 的线程。

### 3. pthread\_detach()函数

线程分为“可联合的”(默认)或“可分离的”。可联合的线程终止后，系统将保留其线程 ID 和退出状态直到另一线程调用 pthread\_join()函数。与之相反，可分离的线程退出后，系统将释放其所有资源，其他线程也无法等待其终止。pthread\_detach()函数将一个线程设置为“可分离”的，其原型如下：

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

thread：被设置为“可分离的”线程的 ID。

返回值：如果成功返回 0，否则返回错误码。

### 4. pthread\_exit()函数

用于终止当前线程，并返回状态值。如果当前线程是“可联合的”，则其退出状态将保留。其函数原型如下：

```
#include <pthread.h>
void pthread_exit(void *value_ptr);
```

value\_ptr：指向线程的退出状态。注意，不要指向一个局部变量，因为当前线程终止后，其所有的局部变量将被撤销。

返回值：无。该函数无法返回其调用者，类似 exit()函数。另外，如下两种方法也能终止线程。

- 线程执行的函数返回。其返回值即为退出状态。
- 当前进程中，任一线程调用 exit()函数，将终止该进程中所有线程。

### 5. pthread\_self()函数

用于返回当前线程的 ID，类似进程的 getpid()函数。原型如下：

```
#include <pthread.h>
pthread_t pthread_self(void);
```

返回值：当前线程 ID。

### 6. pthread\_cancel()函数

用于终止指定线程的执行。其原型如下：

```
#include <pthread.h>
int pthread_cancel(pthread_t target_thread);
```

target\_thread：被终止的线程 ID。

返回值：如果成功返回 0，否则返回错误码。

### 6.3.3 多线程并发服务器

与多进程相同，采用多线程也可实现并发服务器，并且由于线程的系统开销小、切换时间短，对于需处理大量客户的服务器而言具有更大的优势。实现多线程并发服务器的基本过程是：当连接建立后，服务器调用 `pthread_create()` 函数产生新的线程，由新线程处理客户请求，同时主线程等待另一客户的连接请求。其典型模板如下：

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6 #include <pthread.h>

7 /* function to be executed by the new thread */
8 void* start_routine(void* arg);

9 main()
10 {
11     int listenfd, connfd;
12     pthread_t thread;
13     type arg;
14     int BACKLOG = 5;

15     /* Create TCP socket */
16     if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
17         perror("Creating socket failed.");
18         exit(1);
19     }
20     /* Bind socket to address */
21     bind(listenfd, ...);
22     /* listen client's connecting requirement */
23     listen(listenfd, BACKLOG);
24     while(1)
25     {
26         /* accept connection */
27         if ((connfd = accept(listenfd, NULL, NULL)) == -1) {
28             perror("Acceptation failed.");
29             exit(1);
30         }
31         /* Create child to service client */
32         ... /* put parameters to arg */
33         if (pthread_create(&thread, NULL, start_routine, (void*)&arg)) {
34             /* handle exception */
```

```
35     ...
36     exit(1);
37 }
38 }
39 }
```

代码说明如下。

1~6：所需的头文件。

8：声明线程所执行的函数。

14：定义最大允许连接的数量。

15~19：产生 TCP 套接字。

20~21：绑定套接字到相应地址。

22~23：监听网络连接。

24~38：接受客户连接。一旦连接，产生新线程服务客户。然后关闭连接套接字并继续接受下一客户连接。

由以上模板可看出，其结构比多进程服务器更简单。因为所有线程共享打开的描述符，主线程不能关闭连接套接字，新线程也不能关闭监听套接字。任何一个线程关闭连接套接字，将导致系统关闭该连接。

### 6.3.4 给新线程传递参数

传递参数给一新线程并不只是函数的参数传递问题，需要考虑线程的特点。

#### 1. 传递参数的普通方法

首先，用于线程产生的 `pthread_create()` 函数只能允许传递给执行函数一个参数。所以，当需要传递多个数据时，应将这些数据封装在一个结构中，再将该结构传递给执行函数。例子如下：

```
1 void* start_routine(void* arg);
2 struct ARG {
3     int connfd;
4     int other; /* other data */
5 };
6 main()
7 {
8     int listenfd, connfd;
9     pthread_t thread;
10    ARG arg;

11    ...
12    while(1)
13    {
14        /* Accept connection */
15        if ((connfd = accept(sockfd, NULL, NULL)) == -1) {
```

```
16     perrpr("Acception failed.");
17     exit(1);
18 }
19 /* put parameters to arg */
20 arg.connfd = connfd;
21 if (pthread_create(&thread, NULL, start_routine, (void*)&arg)) {
22     printf("Create thread error.\n");
23     exit(1);
24 }
25 }
26 }
27 void* start_routine(void* arg)
28 {
29     ARG info;
30     info.connfd = ((ARG *)arg)->connfd;
31     info.other = ((ARG *)arg)->other;
32     ... /* handle client's requirement */
33     close(info.connfd);
34     pthread_exit(NULL);
35 }
```

代码说明如下。

- 1：声明线程所执行的函数。
- 2~5：定义结构 ARG。
- 15~18：接受客户连接。
- 19~20：设置参数。
- 21~24：产生线程。
- 27~35：实现线程所执行的函数。
- 30~31：取出相应参数。
- 32：处理客户请求。
- 33：关闭连接套接字。
- 34：退出线程。

以上的代码处理一个客户是可以正常工作的，但同时处理多个客户，则无法工作。其原因在于，传递给执行函数的参数是以指针形式传递的。即变量 arg 是所有线程共用的。假设新线程 A 正在处理客户 A 请求，而主线程又接受了另一客户 B 的连接，主线程将修改 arg 内容，这时，线程 A 从 arg 所获得的信息实际上是客户 B 的。由此看出，问题的关键在于，每个新线程如何在主线程修改 arg 之前获得一份 arg 的拷贝。

## 2. 通过指针传递参数

在 C 语言中，传递给函数的参数是被拷贝到函数的执行堆栈中。可利用这一特点使新线程获得参数拷贝。主线程将要传递的数据转换成通用指针类型，然后传递给新线程，新线程再将接收的参数转换成原数据类型。例子如下：

```
1 void* start_routine(void* arg);
2 main()
3 {
4     int listenfd, connfd;
5     pthread_t  thread;
6     ...
7     while(1)
8     {
9         /* Accept a connection */
10         if ((connfd = accept(sockfd, NULL, NULL)) == -1) {
11             perrpr("Acception failed.");
12             exit(1);
13         }
14         if (pthread_create(&thread, NULL, start_routine, (void*)connfd)) {
15             printf("Create thread error.\n");
16             exit(1);
17         }
18     }
19 }
20 void* start_routine(void* arg)
21 {
22     int connfd;
23     connfd = (int ) arg;
24     ... /* handle client requirement */
25     close(info.connfd);
26     pthread_exit(NULL);
27 }
```

代码说明如下。

1：声明线程所执行的函数。

10~13：接受客户连接。

14~17：产生线程。

20~27：实现线程所执行的函数。

23：取出相应参数。

24：处理客户请求。

25：关闭连接套接字。

26：退出线程。

由于执行函数的参数类型被限定为指针类型，因此，arg 的类型必须要能正确地转换成通用指针类型。为保证这一点，arg 的字节长度必须小于或等于通用指针类型的长度。这样可传递的数据很少，而且取决于系统的实现（不同系统有不同的类型长度）。由此可见，这种方法虽然简单，但却有很大的局限性，而且容易出错。

### 3. 通过分配 arg 的空间传递参数

另一种方法，是主线程首先为每个新线程分配存储 arg 的空间，再将 arg 传递给新线程，

新线程使用完后释放 arg 空间。例子如下：

```
1 void* start_routine(void* arg);
2 struct ARG {
3     int connfd;
4     int other;
5 };
6 main()
7 {
8     int listenfd, connfd;
9     pthread_t thread;
10    ARG* arg;

11    ...
12    while(1)
13    {
14        /* Accept connection */
15        if ((connfd = accept(sockfd, NULL, NULL)) == -1) {
16            perrpr("Acception failed.");
17            exit(1);
18        }
19        /* put the parameters to arg */
20        arg = new ARG;
21        arg->connfd = connfd;

22        if (pthread_create(&thread, NULL, start_routine, (void*)arg)) {
23            printf("Create thread error.\n");
24            exit(1);
25        }
26    }
27 }

28 void* start_routine(void* arg)
29 {
30     ARG info;
31     info.connfd = ((ARG *)arg)->connfd;
32     info.other = ((ARG *)arg)->other;
33     ... /* handle client */
34     close(info.connfd);
35     delete arg;
36     pthread_exit(NULL);
37 }
```

代码说明如下。

1：声明线程所执行的函数。

- 2~5：定义结构 ARG。
- 15~18：接受客户连接。
- 19~21：设置参数。
- 22~25：产生线程。
- 28~37：实现线程所执行的函数。
- 30~32：取出相应参数。
- 33：处理客户请求。
- 34：关闭连接套接字。
- 35：退出线程。

还可以让新线程拷贝 arg，但必须保证主线程在新进程拷贝完成后，主线程才修改 arg 内容。这要采用线程同步技术，较为复杂且不实用。

### 6.3.5 多线程并发服务器实例

以下实例是采用多线程并发服务器算法实现的。通过该实例可了解多线程并发服务器是如何处理多客户的。

该实例包括服务器程序，客户采用多进程并发服务器实例的客户程序。完成的功能与多进程并发服务器实例相同。

#### 1. 服务器源程序清单（程序 6.5）

```
1  #include <stdio.h>           /* These are the usual header files */
2  #include <strings.h>         /* for bzero() */
3  #include <unistd.h>          /* for close() */
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>
8  #include <pthread.h>

9  #define PORT 1234           /* Port that will be opened */
10 #define BACKLOG 5           /* Number of allowed connections */
11 #define MAXDATASIZE 1000

12 void process_cli(int connectfd, sockaddr_in client);
13 /* function to be executed by the new thread */
14 void* start_routine(void* arg);
15 struct ARG {
16     int connfd;
17     sockaddr_in client;
18 };

19 main()
20 {
```



```
21 int listenfd, connectfd; /* socket descriptors */
22 pthread_t thread;
23 ARG *arg;
24 struct sockaddr_in server; /* server's address information */
25 struct sockaddr_in client; /* client's address information */
26 int sin_size;

27 /* Create TCP socket */
28 if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
29     /* handle exception */
30     perror("Creating socket failed.");
31     exit(1);
32 }

33 int opt = SO_REUSEADDR;
34 setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

35 bzero(&server, sizeof(server));
36 server.sin_family = AF_INET;
37 server.sin_port = htons(PORT);
38 server.sin_addr.s_addr = htonl(INADDR_ANY);
39 if (bind(listenfd, (struct sockaddr *)&server,
40         sizeof(struct sockaddr)) == -1) {
41     /* handle exception */
42     perror("Bind error.");
43     exit(1);
44 }

44 if (listen(listenfd, BACKLOG) == -1) { /* calls listen() */
45     perror("listen() error\n");
46     exit(1);
47 }

48 sin_size = sizeof(struct sockaddr_in);
49 while(1)
50 {
51     /* Accept connection */
52     if ((connectfd = accept(listenfd,
53         (struct sockaddr *)&client, &sin_size)) == -1) {
54         perror("accept() error\n");
55         exit(1);
56     }
57     /* Create thread */

57     arg = new ARG;
```

```
58     arg->connfd = connectfd;
59     memcpy((void *)&arg->client, &client, sizeof(client));

60     if (pthread_create(&thread, NULL, start_routine, (void*)arg)) {
61         /* handle exception */
62         perror("Pthread_create() error");
63         exit(1);
64     }
65 }
66 close(listenfd); /* close listenfd */
67 }

68 void process_cli(int connectfd, sockaddr_in client)
69 {
70     int num;
71     char recvbuf[MAXDATASIZE], sendbuf[MAXDATASIZE],
        cli_name[MAXDATASIZE];

72     printf("You got a connection from %s. ",inet_ntoa(client.sin_addr) );
73     /* Get client's name from client */
74     num = recv(connectfd, cli_name, MAXDATASIZE,0);
75     if (num == 0) {
76         close(connectfd);
77         printf("Client disconnected.\n");
78         return;
79     }
80     cli_name[num - 1] = '\0';
81     printf("Client's name is %s.\n",cli_name);

82     while (num = recv(connectfd, recvbuf, MAXDATASIZE,0)) {
83         recvbuf[num] = '\0';
84         printf("Received client( %s ) message: %s",cli_name, recvbuf);
85         for (int i = 0; i < num - 1; i++) {
86             sendbuf[i] = recvbuf[num - i -2];
87         }
88         sendbuf[num - 1] = '\0';
89         send(connectfd,sendbuf,strlen(sendbuf),0);
90     }
91     close(connectfd); /* close connectfd */
92 }

93 void* start_routine(void* arg)
94 {
95     ARG *info;
96     info = (ARG *)arg;
```

```

97  /* handle client's requirement */
98  process_cli(info->connfd, info->client);

99  delete arg;
100 pthread_exit(NULL);
101}

```

## 2. 程序分析

1~8：所需的头文件。

9~11：定义端口号、最大允许连接的数量及缓冲区的大小。

12：声明“客户处理”process\_cli()函数。用于处理客户请求。

14：声明线程所执行的函数。

15~18：定义 ARG 结构。用于主线程传递参数给新线程。

27~32：产生 TCP 套接字。

33~34：设置套接字选项为 SO\_REUSEADDR。

35~43：绑定套接字到相应地址。本例 IP 地址设为 INADDR\_ANY，则可接收来自本机任何 IP 地址的客户连接。

44~47：监听网络连接。

48~67：接受客户连接。一旦连接，产生新线程服务客户。然后关闭连接套接字并继续接受下一客户连接。

57~59：分配空间给 arg，然后把连接套接字(connectfd)和客户地址信息(client)赋给 arg。

60~64：产生新线程，如果子进程产生失败，显示出错信息并退出程序。

71~92：实现服务功能。

93~100：实现 start\_routine()函数。

98：调用 process\_cli()函数。

99：释放 arg 空间。

100：退出线程。

## 3. 运行程序

下面在同一主机上运行上述程序。

(1) 首先启动服务器程序：

```
thrserver
```

(2) 然后同时运行客户 1 和客户 2：

```

procclient 127.0.0.1
procclient 127.0.0.1

```

## 4. 服务器运行结果

```

$ thrserver
You got a connection from 127.0.0.1. Client's name is client1.

```

```
Received client( client1 ) message: 1234
You got a connection from 127.0.0.1. Client's name is client2.
Received client( client2 ) message: abcd
Received client( client2 ) message: efgh
Received client( client1 ) message: 5678
```

### 5. 客户 1 运行结果

```
$ procclient 127.0.0.1
Connected to server.
Input name : client1
Input string to server:1234
Server Message: 4321
Input string to server:5678
Server Message: 8765
Input string to server: ^D
Exit.
$
```

### 6. 客户 2 运行结果

```
$ procclient 127.0.0.1
Connected to server.
Input name : client2
Input string to server:abcd
Server Message: dcba
Input string to server:efgh
Server Message: hgfe
Input string to server: ^D
Exit.
$
```

### 7. 运行结果说明

从运行结果可看出，两客户均能及时地获得服务。说明多线程服务器能够实现并发处理。

## 6.3.6 线程安全 (MT-safe) 实例

由于同一进程中的所有线程共享相同的内存空间，如果多个线程修改相同的内存区就会造成意想不到的后果。这就是线程安全问题，在多线程环境中应十分注意。

在 Unix 系统中，由于静态变量的使用，造成许多 API 函数不是线程安全的。对于大多数非安全接口的函数，都存在一个 MT-safe 的版本。新的 MT-safe 函数一般是旧的非安全函数加上 `_r` 后缀。例如 Solaris 系统提供以下 `_r` 函数：

|                              |                              |                              |
|------------------------------|------------------------------|------------------------------|
| <code>asctime_r(3C)</code>   | <code>ctermid_r(3S)</code>   | <code>ctime_r(3C)</code>     |
| <code>fgetgrent_r(3C)</code> | <code>fgetpwent_r(3C)</code> | <code>fgetspent_r(3C)</code> |

|                     |                      |                      |
|---------------------|----------------------|----------------------|
| Gamma_r(3M)         | getgrgid_r(3C)       | getgrnam_r(3C)       |
| getlogin_r(3C)      | getpwnam_r(3C)       | getpwuid_r(3C)       |
| getgrent_r(3C)      | gethostbyaddr_r(3N)  | gethostbyname_r(3N)  |
| gethostent_r(3N)    | getnetbyaddr_r(3N)   | getnetbyname_r(3N)   |
| getnetent_r(3N)     | Getprotobyname_r(3N) | getprotobyname_r(3N) |
| getprotoent_r(3N)   | getpwent_r(3C)       | getrpcbyname_r(3N)   |
| getrpcbyname_r(3N)  | getrpcent_r(3N)      | getservbyname_r(3N)  |
| getservbyport_r(3N) | getservent_r(3N)     | getspent_r(3C)       |
| getspnam_r(3C)      | gmtime_r(3C)         | lgamma_r(3M)         |
| localtime_r(3C)     | nis_sperror_r(3N)    | rand_r(3C)           |
| readdir_r(3C)       | strtok_r(3C)         | tmpnam_r(3C)         |
| ttynam_r(3C)        |                      |                      |

### 6.3.6.1 实例 1

以下通过例子说明线程安全问题。该例子实现的功能与多线程并发服务器实例的类似，但增加了一个客户数据存储功能，即服务器将存储每个连接客户所发来的所有数据，当连接终止后，服务将显示客户的名字及相应的数据。

#### 1. 程序清单（程序 6.6）

源程序清单如下：

```

1  #include <stdio.h>           /* These are the usual header files */
2  #include <strings.h>         /* for bzero() */
3  #include <unistd.h>          /* for close() */
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>
8  #include <pthread.h>

9  #define PORT 1234           /* Port that will be opened */
10 #define BACKLOG 5           /* Number of allowed connections */
11 #define MAXDATASIZE 1000

12 void process_cli(int connectfd, sockaddr_in client);
13 void savedata(char* recvbuf, int len, char* cli_data);
14 /* function to be executed by the new thread */
15 void* start_routine(void* arg);
16 struct ARG {
17     int connfd;
18     sockaddr_in client;
19 };

20 main()
21 {
22     int listenfd, connectfd; /* socket descriptors */
23     pthread_t thread;
```

```
24 ARG *arg;
25 struct sockaddr_in server; /* server's address information */
26 struct sockaddr_in client; /* client's address information */
27 int sin_size;

28 /* Create TCP socket */
29 if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
30     /* handle exception */
31     perror("Creating socket failed.");
32     exit(1);
33 }

34 int opt = SO_REUSEADDR;
35 setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

36 bzero(&server, sizeof(server));
37 server.sin_family=AF_INET;
38 server.sin_port=htons(PORT);
39 server.sin_addr.s_addr = htonl (INADDR_ANY);
40 if (bind(listenfd, (struct sockaddr *)&server,
41         sizeof(struct sockaddr)) == -1) {
42     /* handle exception */
43     perror("Bind error.");
44     exit(1);
45 }

46 if(listen(listenfd, BACKLOG) == -1){ /* calls listen() */
47     perror("listen() error\n");
48     exit(1);
49 }

49 sin_size=sizeof(struct sockaddr_in);
50 while(1)
51 {
52     /* accept connection */
53     if ((connectfd = accept(listenfd,
54         (struct sockaddr *)&client, &sin_size)) == -1) {
55         perror("accept() error\n");
56         exit(1);
57     }
58     /* create child thread */
59     arg = new ARG;
60     arg->connfd = connectfd;
61     memcpy((void *)&arg->client, &client, sizeof(client));
```

```
61     if (pthread_create(&thread, NULL, start_routine, (void*)arg)) {
62         /* handle exception */
63         perror("Pthread_create() error");
64         exit(1);
65     }
66 }
67 close(listenfd); /* close listenfd */
68 }

69 void process_cli(int connectfd, sockaddr_in client)
70 {
71     int num;
72     char cli_data[5000];
73     char recvbuf[MAXDATASIZE], sendbuf[MAXDATASIZE],
        cli_name[MAXDATASIZE];

    /* prints client's IP */
74     printf("You got a connection from %s. ",inet_ntoa(client.sin_addr) );
75     /* Get client's name from client */
76     num = recv(connectfd, cli_name, MAXDATASIZE,0);
77     if (num == 0) {
78         close(connectfd);
79         printf("Client disconnected.\n");
80         return;
81     }
82     cli_name[num - 1] = '\0';
83     printf("Client's name is %s.\n",cli_name);

84     while (num = recv(connectfd, recvbuf, MAXDATASIZE,0)) {
85         recvbuf[num] = '\0';
86         printf("Received client( %s ) message: %s",cli_name, recvbuf);
87         /* save user's data */
88         savedata(recvbuf,num,cli_data);
89         /* reverse usr's data */
90         for (int i = 0; i < num - 1; i++) {
91             sendbuf[i] = recvbuf[num - i -2];
92         }
93         sendbuf[num - 1] = '\0';

        /* send to the client welcome message */
94         send(connectfd,sendbuf,strlen(sendbuf),0);
95     }

96     close(connectfd); /* close connectfd */
97     printf("Client( %s ) closed connection.
```

```
        User's data: %s\n", cli_name, cli_data);
98  }

99  void* start_routine(void* arg)
100 {
101 ARG *info;
102 info = (ARG *)arg;

103 /* Handle client */
104 process_cli(info->connfd, info->client);

105 delete info;
106 pthread_exit(NULL);
107 }

108 void savedata(char* recvbuf, int len, char* cli_data)
109 {
110 static int index = 0;
111 for (int i = 0; i < len - 1; i++) {
112     cli_data[index++] = recvbuf[i];
113 }
114 cli_data[index] = '\0';
115 }
```

代码说明如下。

15：声明线程执行的函数。

16~19：定义 ARG 结构。用于主线程传递参数给新线程。

29~33：产生 TCP 套接字。

34~35：设置套接字选项为 SO\_REUSEADDR。

36~44：绑定套接字到相应地址。本例 IP 地址设为 INADDR\_ANY，则可接收来自本机任何 IP 地址的客户连接。

45~48：监听网络连接。

58~60：分配空间给 arg，然后把连接套接字(connectfd)和客户地址信息(client)赋给 arg。

61~65：产生新线程，如果子进程产生失败，显示出错信息并退出程序。

88：调用 savedata()函数（非线程安全的函数）

97：显示客户关闭连接及接收到的该客户所有数据

110：定义静态变量 index，用于表明当前数据的存储位置

111~113：把接收的客户数据按先后次序存放在客户数据缓冲区（cli\_data）

114：将最后一个字节赋为“\0”，以便于显示。

## 2. 运行程序

下面在同一主机上运行上述程序。

（1）首先启动服务器程序：



```
thrserver1
```

(2) 然后同时运行客户 1 和客户 2 :

```
procclient 127.0.0.1  
procclient 127.0.0.1
```

### 3. 运行结果

服务器运行结果如下 :

```
$ thrserver1  
You got a connection from 127.0.0.1. Client's name is client1.  
Received client( client1 ) message: 1234  
You got a connection from 127.0.0.1. Client's name is client2.  
Received client( client2 ) message: abc  
Received client( client1 ) message: 5678  
Client( client1 ) closed connection. User's data: 1234  
Received client( client2 ) message: efg  
Client( client2 ) closed connection. User's data:
```

客户 1 运行结果如下 :

```
$ procclient 127.0.0.1  
Connected to server.  
Input name : client1  
Input string to server:1234  
Server Message: 4321  
Input string to server:5678  
Server Message: 8765  
Input string to server: ^D  
Exit.  
$
```

客户 2 运行结果如下 :

```
$ procclient 127.0.0.1  
Connected to server.  
Input name : client2  
Input string to server:abc  
Server Message: cba  
Input string to server:efg  
Server Message: gfe  
Input string to server: ^D  
Exit.  
$
```

### 4. 结论

由服务器的运行结果可以看出, 当客户 1 关闭时, 显示的用户数据为 “ 1234 ”, 客户 2 关闭时, 显示的用户数据为 “ ”。而正确的结果应为 “ 12345678 ” 和 “ abcefg ”。为什么会

产生这种结果呢？其关键在于调用了非线性安全的函数（`savedata()`），该函数包含了一个静态变量 `index`。

### 6.3.6.2 实例 2

从上述例子可以看出，在多线程环境中，应避免使用静态变量。在 Unix 系统中，可采用线程专用数据（TSD）取代静态变量。它类似于全局数据，只不过它是线程私有的。TSD 是以线程为界限的。TSD 是定义线程私有数据的惟一方法。每个线程私有数据项都由一个进程内惟一的關鍵字（KEY）来标志。用这个关键字，线程可以存取线程私有的数据。

通常，通过以下四个函数使用 TSD。

#### （1）`pthread_key_create()`函数

`pthread_key_create()`函数在进程内部分配一个标志 TSD 的关键字。关键字是进程内部惟一的，所有线程在创建时的关键字值是 `NULL`。一旦关键字被建立，每一个线程可以为关键字绑定一个值。这个值对于绑定的线程来说是惟一的，由每个线程独立维护。其函数原型为：

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *key,
void (*destructor)(void *value));
```

`key`：指向创建的关键字。

`destructor`：一个可选的析构函数，可以和每个关键字联系起来。如果一个关键字的 `destructor` 不空，而且线程赋给该关键字一个非空值，在线程退出时该析构函数被调用，并使用当前的绑定值。对于所有关键字的析构函数，执行的顺序是不能指定的。

返回值：在正常执行后返回 0，否则返回如下错误码。

- `EAGAIN`：关键字的名字空间用尽。
- `ENOMEM`：内存不够。

#### （2）`pthread_setspecific()`函数

`pthread_setspecific()`函数为 TSD 关键字绑定一个与本线程相关的值。其函数原型为：

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void *value);
```

`key`：TSD 关键字。

`value`：本线程相关的值。

返回值：在正常执行后返回 0，否则返回如下错误码。

- `ENOMEM`：内存不够。
- `EINVAL`：关键字非法。

#### （3）`pthread_getspecific()`函数

`pthread_getspecific()`函数获得与调用线程相关的关键字所绑定的值。其函数原型为：

```
#include <pthread.h>
void *pthread_getspecific(pthread_key_t key);
```

`key`：TSD 关键字。

返回值：在正常执行后返回与调用线程相关的关键字所绑定的值，否则返回 NULL。

#### (4) pthread\_once()函数

pthread\_once()函数用于初始化动态包。在线程专用数据的使用中，该函数被调用多次，但当设置为 PTHREAD\_ONCE\_INIT 时，可保证所指向的函数在同一进程中只被调用一次。其函数原型为：

```
#include <pthread.h>
pthread_once_t once_control = PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *once_control, void
    (*init_routine) (void));
```

once\_control：在线程专用数据的使用中，该值设置为 PTHREAD\_ONCE\_INIT。

init\_routine：所指向的函数。当 once\_control 的值设置为 PTHREAD\_ONCE\_INIT 时，该函数在同一进程中只被调用一次。

返回值：在正常执行后返回 0，否则返回如下错误码。

- EINVAL：关键字非法。

以下实例就是使用了 TSD 取代静态变量来实现相同的功能。

#### 1. 程序清单（程序 6.7）

其源程序如下：

```
1  #include <stdio.h>          /* These are the usual header files */
2  #include <strings.h>        /* for bzero() */
3  #include <unistd.h>         /* for close() */
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>
8  #include <pthread.h>
9  #include <stdlib.h>

10 #define PORT 1234          /* Port that will be opened */
11 #define BACKLOG 5          /* Number of allowed connections */
12 #define MAXDATASIZE 1000

13 void process_cli(int connectfd, sockaddr_in client);
14 void savedata_r(char* recvbuf, int len, char* cli_data);
15 /* function to be executed by the new thread */
16 void* start_routine(void* arg);
17 struct ARG {
18     int connfd;
19     sockaddr_in client;
20 };

21 /* Thread saft */
```

```
22 static pthread_key_t    key;
23 static pthread_once_t    once = PTHREAD_ONCE_INIT;
24 static void destructor(void *ptr)
25 {
26     free(ptr);
27 }
28 static void getkey_once(void)
29 {
30     pthread_key_create(&key, destructor);
31 }
32 typedef struct DATA_THR{
33     int    index;
34 };

35 main()
36 {
37     int listenfd, connectfd; /* socket descriptors */
38     pthread_t  thread;
39     ARG *arg;
40     struct sockaddr_in server; /* server's address information */
41     struct sockaddr_in client; /* client's address information */
42     int sin_size;

43     /* Create TCP socket */
44     if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
45         /* handle exception */
46         perror("Creating socket failed.");
47         exit(1);
48     }

49     int opt = SO_REUSEADDR;
50     setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

51     bzero(&server, sizeof(server));
52     server.sin_family=AF_INET;
53     server.sin_port=htons(PORT);
54     server.sin_addr.s_addr = htonl (INADDR_ANY);
55     if (bind(listenfd, (struct sockaddr *)&server,
56             sizeof(struct sockaddr)) == -1) {
57         /* handle exception */
58         perror("Bind error.");
59         exit(1);
60     }

60     if(listen(listenfd, BACKLOG) == -1){ /* calls listen() */
```

```
61     perror("listen() error\n");
62     exit(1);
63 }

64 sin_size=sizeof(struct sockaddr_in);
65 while(1)
66 {
67     /* accept connection */
68     if ((connectfd = accept(listenfd,
69         (struct sockaddr *)&client,&sin_size))== -1) {
69         perror("accept() error\n");
70         exit(1);
71     }
72     /* create thread */
73     arg = new ARG;
74     arg->connfd = connectfd;
75     memcpy((void *)&arg->client, &client, sizeof(client));
76     if (pthread_create(&thread, NULL, start_routine, (void*)arg)) {
77         /* handle exception */
78         perror("Pthread_create() error");
79         exit(1);
80     }
81 }
82 close(listenfd); /* close listenfd */
83 }

84 void process_cli(int connectfd, sockaddr_in client)
85 {
86     int num;
87     char cli_data[5000];
88     char recvbuf[MAXDATASIZE], sendbuf[MAXDATASIZE],
89         cli_name[MAXDATASIZE];

89     printf("You got a connection from %s. ",inet_ntoa(client.sin_addr) );
90     /* Get client's name from client */
91     num = recv(connectfd, cli_name, MAXDATASIZE,0);
92     if (num == 0) {
93         close(connectfd);
94         printf("Client disconnected.\n");
95         return;
96     }
97     cli_name[num - 1] = '\0';
98     printf("Client's name is %s.\n",cli_name);

99     while (num = recv(connectfd, recvbuf, MAXDATASIZE,0)) {
```

```
100     recvbuf[num] = '\0';
101     printf("Received client( %s ) message: %s",cli_name, recvbuf);

102     /* save user's data */
103     savedata_r(recvbuf,num,cli_data);

104     /* reverse usr's data */
105     for (int i = 0; i < num - 1; i++) {
106         sendbuf[i] = recvbuf[num - i - 2];
107     }
108     sendbuf[num - 1] = '\0';

/* send to the client welcome message */
109     send(connectfd,sendbuf,strlen(sendbuf),0);
110 }

111 close(connectfd); /* close connectfd */
112 printf("Client( %s ) closed connection.
        User's data: %s\n",cli_name,cli_data);
113 }

114 void* start_routine(void* arg)
115 {
116     ARG *info;
117     info = (ARG *)arg;

118     /* 处理客户请求 */
119     process_cli(info->connfd, info->client);

120     delete info;
121     pthread_exit(NULL);
122 }

123 void savedata_r(char* recvbuf, int len, char* cli_data)
124 {
125     DATA_THR* data;

126     /* thread_special data */
127     pthread_once(&once, getkey_once);
128     if ( (data = (DATA_THR *)pthread_getspecific(key)) == NULL) {
129         data = (DATA_THR *)calloc(1, sizeof(DATA_THR));
130         pthread_setspecific(key, data);
131         data->index = 0; }

132     for (int i = 0; i < len - 1; i++) {
```

```

133     cli_data[data->index++] = recvbuf[i];
134     }
135     cli_data[data->index] = '\0';
136 }

```

## 2. 程序分析

14：声明 savedata\_r()函数。用于存储客户数据。

22：定义 TSD 关键字变量 key。

23：定义变量 once，初值为 PTHREAD\_ONCE\_INIT。

24~27：定义析构函数 destructor，在线程退出时该析构函数将被调用，以释放为 TSD 分配的空间。

28~31：定义函数 getkey\_once()，以产生 TSD 关键字。

32~34：定义 DATA\_THR 结构，用于存储 TSD。

103：调用 savedata\_r()函数（线程安全的函数）。

127：调用 getkey\_once()函数产生 TSD 关键字。由于 once 值为 PTHREAD\_ONCE\_INIT，getkey\_once()函数只是在第一个新线程中执行。

128~131：如果当前线程未分配空间给 TSD，则分配空间并与 TSD 关键字绑定。

132~135：把接收的客户数据按先后次序存放在客户数据缓冲区 (cli\_data)。

## 3. 运行程序

(1) 下面在同一主机上运行上述程序。首先启动服务器程序 (thrserver2)。

(2) 然后同时运行客户 1 (procclient) 和客户 2。

## 4. 运行结果

服务器运行结果如下：

```

$ thrserver2
You got a connection from 127.0.0.1. Client's name is client1.
Received client( client1 ) message: 1234
You got a connection from 127.0.0.1. Client's name is client2.
Received client( client2 ) message: abc
Received client( client1 ) message: 5678
Client( client1 ) closed connection. User's data: 12345678
Received client( client2 ) message: def
Client( client2 ) closed connection. User's data: abcdef

```

客户 1 运行结果如下：

```

$ procclient 127.0.0.1
Connected to server.
Input name : client1
Input string to server:1234
Server Message: 4321
Input string to server:5678
Server Message: 8765

```

```
Input string to server: ^D
Exit.
$
```

客户 2 运行结果如下：

```
$ procclient 127.0.0.1
Connected to server.
Input name : client2
Input string to server: abc
Server Message: cba
Input string to server: def
Server Message: fed
Input string to server: ^D
Exit.
$
```

## 5. 结论

由运行结果可以看出，因为使用了 TSD，运行结果完全正确。TSD 的实现略有些繁琐，却是将一个非线程安全函数转换为线程安全函数的常用方法。由 TSD 实例可以看出，为了将 savedata() 函数转换成线程安全的函数(savedata\_r)，只需修改该函数内部代码以及增加几个 TSD 相关的函数，对于程序的其他部分不需做任何变动。

### 6.3.6.3 实例 3

为实现线程安全函数，另一种常用的方法是：通过使用函数的参变量来取代静态变量。这种方法非常简单，但需要改变函数的原型以增加相应的参变量。另外，该函数的调用函数必须为这些变量分配相应的空间并进行初始化。

#### 1. 程序清单（程序 6.8）

该实例实现与 TSD 相同功能，其源程序如下：

```
1  #include <stdio.h>           /* These are the usual header files */
2  #include <strings.h>         /* for bzero() */
3  #include <unistd.h>          /* for close() */
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>
8  #include <pthread.h>
9  #include <stdlib.h>

10 #define PORT 1234           /* Port that will be opened */
11 #define BACKLOG 5           /* Number of allowed connections */
12 #define MAXDATASIZE 1000

13 typedef struct DATA_THR{
```



```
14     int    index;
15     };
16 void process_cli(int connectfd, sockaddr_in client);
17 void savedata(char* recvbuf, int len, char* cli_data, DATA_THR* data);
18 /* function to be executed by the new thread */
19 void* start_routine(void* arg);
20 struct ARG {
21     int connfd;
22     sockaddr_in client;
23 };

24 main()
25 {
26 int listenfd, connectfd; /* socket descriptors */
27 pthread_t thread;
28 ARG *arg;

29 struct sockaddr_in server; /* server's address information */
30 struct sockaddr_in client; /* client's address information */

31 int sin_size;

32 /* Create TCP socket */
33 if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
34     /* handle exception */
35     perror("Creating socket failed.");
36     exit(1);
37 }

38 int opt = SO_REUSEADDR;
39 setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

40 bzero(&server, sizeof(server));
41 server.sin_family=AF_INET;
42 server.sin_port=htons(PORT);
43 server.sin_addr.s_addr = htonl (INADDR_ANY);
44 if (bind(listenfd, (struct sockaddr *)&server,
45         sizeof(struct sockaddr)) == -1) {
46     /* handle exception */
47     perror("Bind error.");
48     exit(1);
49 }

49 if(listen(listenfd, BACKLOG) == -1){ /* calls listen() */
50     perror("listen() error\n");
```

```
51     exit(1);
52 }

53 sin_size=sizeof(struct sockaddr_in);
54 while(1)
55 {
56     /* Accept connection */
57     if ((connectfd = accept(listenfd,
58         (struct sockaddr *)&client,&sin_size))== -1) {
59         perror("accept() error\n");
60         exit(1);
61     }
62     /* create thread */
63     arg = new ARG;
64     arg->connfd = connectfd;
65     memcpy((void *)&arg->client, &client, sizeof(client));
66
67     if (pthread_create(&thread, NULL, start_routine, (void*)arg)) {
68         /* handle exception */
69         perror("Pthread_create() error");
70         exit(1);
71     }
72 }

73 void process_cli(int connectfd, sockaddr_in client)
74 {
75     int num;
76     DATA_THR data;
77     char cli_data[5000];
78     char recvbuf[MAXDATASIZE], sendbuf[MAXDATASIZE],
79         cli_name[MAXDATASIZE];

80     data.index = 0;
81     /* prints client's IP */
82     printf("You got a connection from %s. ",inet_ntoa(client.sin_addr) );
83     /* Get client's name from client */
84     num = recv(connectfd, cli_name, MAXDATASIZE,0);
85     if (num == 0) {
86         close(connectfd);
87         printf("Client disconnected.\n");
88         return;
89     }
90     cli_name[num - 1] = '\0';
```

```
89  printf("Client's name is %s.\n",cli_name);

90  while (num = recv(connectfd, recvbuf, MAXDATASIZE,0)) {
91      recvbuf[num] = '\0';
92      printf("Received client( %s ) message: %s",cli_name, recvbuf);

93      /* save user's data */
94      savedata(recvbuf,num,cli_data, &data);

95      /* reverse usr's data */
96      for (int i = 0; i < num - 1; i++) {
97          sendbuf[i] = recvbuf[num - i -2];
98      }
99      sendbuf[num - 1] = '\0';

/* send to the client welcome message */
100  send(connectfd,sendbuf,strlen(sendbuf),0);
101  }

102 close(connectfd); /* close connectfd */
103 printf("Client( %s ) closed connection.
        User's data: %s\n",cli_name,cli_data);
104 }

105 void* start_routine(void* arg)
106 {
107 ARG *info;
108 info = (ARG *)arg;

109 /* handle client */
110 process_cli(info->connfd, info->client);

111 delete info;
112 pthread_exit(NULL);
113 }

114 void savedata(char* recvbuf, int len, char* cli_data, DATA_THR* data)
115 {
116 for (int i = 0; i < len - 1; i++) {
117     cli_data[data->index++] = recvbuf[i];
118 }
119 cli_data[data->index] = '\0';
120 }
```

## 2. 程序分析

13~14：定义数据结构 DATA\_THR，用于为线程分配数据空间。

19：声明线程所执行的函数。

20~23：定义 ARG 结构。用于主线程传递参数给新线程。

32~37：产生 TCP 套接字。

38~39：设置套接字选项为 SO\_REUSEADDR。

40~48：绑定套接字到相应地址。本例 IP 地址设为 INADDR\_ANY，则可接收来自本机任何 IP 地址的客户连接。

49~52：监听网络连接。

62~64：分配空间给 arg，然后把连接套接字(connectfd)和客户地址信息(client)赋给 arg。

65~69：产生新线程，如果子进程产生失败，显示出错信息并退出程序。

76：定义变量 data。

79：初始化 data。

94：调用 savedata()函数（这是线程安全的函数）。

96~101：将用户数据反转，并发回客户。

103：显示客户关闭连接及接收到的该客户所有数据。

116~118：把接收的客户数据按先后次序存放在客户数据缓冲区（cli\_data）。

119：将最后一个字节赋为“\0”，以便于显示。

## 3. 运行结果说明

与 TSD 实例采用相同方法运行可得到相同的结果。

# 6.4 I/O 多路复用服务器

除了可以采用多进程和多线程方法实现并发服务器之外，还可以采用 I/O 多路复用技术。通过该技术，系统内核缓冲 I/O 数据，当某 I/O 准备好后，系统将通知应用程序该 I/O 可读或可写，这样应用程序可马上完成相应的 I/O 操作，而不再需要等待系统完成相应 I/O 操作，从而应用程序不必因等待 I/O 操作而阻塞。

与多进程和多线程技术相比，I/O 多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。

在网络编程中，I/O 多路复用技术主要应用于以下几方面：

- 客户程序需要同时处理交互式的输入以及与服务端之间的网络连接；
- 客户端需要同时对多个网络连接作出反应；
- TCP 服务器需要同时处理处于监听状态和多个连接状态的套接字；
- 服务器需要处理多个网络协议的套接字；
- 服务器需要同时处理不同的网络服务和协议。

### 6.4.1 I/O 模式

在 Unix 系统中，主要包括以下 5 种 I/O 模式：

- 阻塞 I/O
- 非阻塞 I/O
- I/O 多路复用(select 和 poll 函数)
- 信号驱动 I/O(SIGIO)
- 异步 I/O

#### 1. 阻塞 I/O 模式

进程调用 `recvfrom()` 函数，此函数直到数据报到达且已拷贝到应用缓冲区或者出错才返回。最常见的错误是函数调用被信号中断。进程阻塞的时间是指从调用 `recvfrom()` 函数开始到它返回的这段时间。当进程返回成功指示时，应用进程开始处理数据报。

#### 2. 非阻塞方式

当请求的 I/O 操作不能完成时，不让进程睡眠，而是立即返回一个错误。例如，调用 `recvfrom()` 函数时无数据返回，立即返回一个错误。如果数据报已准备好，数据被拷贝到应用缓冲区，`recvfrom()` 函数返回成功指示，接着处理数据。

#### 3. I/O 多路复用模式

在 I/O 多路复用中，调用 `select` 或 `poll` 函数后，应用程序将阻塞直到 `select/poll` 函数返回，通过这种方法避免因实际的 I/O 系统调用而引起的阻塞。图 6.9 说明了 I/O 多路复用的工作过程。

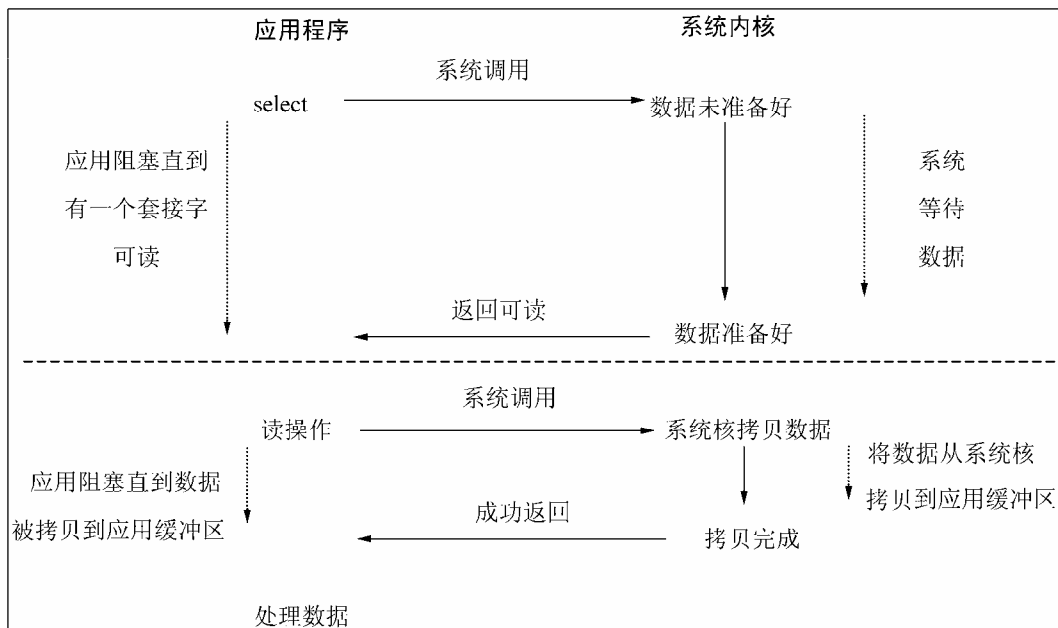


图 6.9 I/O 多路复用的工作过程

如上图所示,当调用 `select()` 函数时,过程阻塞,等待某个 I/O 准备好(如有数据可读)。 `select()` 函数返回后,可调用相应的读操作,系统将数据拷贝到应用程序的缓冲区中,读操作能够立即完成。

#### 4. 信号驱动 I/O 模式

套接字启动信号驱动 I/O, 并通过系统调用安装一个信号处理程序。此系统调用立即返回, 进程继续工作, 它是非阻塞的。当数据报准备好被读时, 就为该进程生成一个 SIGIO 信号。随即可以在信号处理程序中调用 `recvfrom()` 函数来读数据报, 并通知主循环数据已准备好。也可以通知主循环, 让它来读数据报。

#### 5. 异步 I/O 模式

让系统内核启动操作, 并在整个操作完成后(包括将数据从内核拷贝到用户自己的缓冲区)通知用户。

### 6.4.2 `select()` 函数

#### 1. `Select()` 函数的作用

`select()` 函数用于实现 I/O 多路复用, 它允许进程指示系统内核等待多个事件中的任一个发生, 并仅在一个或多个事件发生或经过某指定的时间后才唤醒进程。例如, 可以调用 `select()` 函数并通知系统内核仅在下列情况发生时才返回:

- 集合 {1, 2, 3, 4} 中的任何描述符准备好读;
- 集合 {2, 3, 4} 的任何描述符准备好写;
- 集合 {1, 4} 中的任何描述符有异常条件待处理;
- 已经过了 10.5 秒。

描述符不受限于套接字: 任何描述符(例如文件描述符)都可用 `select()` 函数来测试。

#### 2. `Select()` 函数原型

```
#include <sys/time.h>
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *errorfds, struct timeval *timeout);
```

`nfds`: `select()` 函数监视描述符数的最大值。根据进程中打开的描述符数而定, 一般设为要监视描述符的最大数加 1。当 `select()` 函数刚开始设计时, 操作系统常对每个进程可用的最大描述符数上限作出限制(4.2BSD 的限制为 31), `select()` 函数也就用相同的限制值。目前, Unix 版本对每个进程的描述符数根本不作限制(仅受内存量和管理性限制), 定义在 `sys/types.h` 中:

```
#DEFINE FD_SETSIZE 256.
```

`readfds`: `select()` 函数监视的可读描述符集合。有以下任何一种情况均表示可读。

- 套接字接收缓冲区中的数据字节数大于等于套接字接收缓冲区限度的当前值, 对于 TCP/UDP 默认值为 1。
- 面向连接的一个方向的读操作关闭(接收了 FIN 的 TCP 连接)。

- 套接字是一个监听套接字，其已完成的连接数为非 0。
- 有一个套接字错误待处理。

writelfds: select()函数监视的可写描述符集合。有以下任何一种情况均表示可写。

- 套接字发送缓冲区中的可用空间字节数大于等于套接字发送缓冲区限度的当前值，且套接字已连接（如 TCP），或套接字不要求连接（UDP）；
- 面向连接的一个方向的写操作关闭。对这样的套接字的写操作将产生信号 SIGPIPE。
- 有一个套接字错误待处理。对这样的套接字的写操作将不阻塞且返回一个错误（-1）。

errorfds: select()函数监视的异常描述符集合。如果一个套接字存在带外数据或者仍处于带外标记，那它有异常条件待处理。

timeout: select()函数的超时结束时间。

### 3. 返回值

如果成功，返回总的位数，这些位对应已准备好的描述符。否则返回-1，并在 errno 中设置相应错误码。

readset、writerset 和 errorfds 参数让系统内核测试读、写和异常条件所需的描述符。这三个参数的每一个都指定一个或多个描述符值。fd\_set 的数据结构，是一个整数数组。数组的第一个元素对应于描述符 0~31，数组的第二个元素对应于描述符 32~63，以此类推。每一个数组元素都能与一已打开的描述符建立联系。当调用 select()时，由系统内核根据 I/O 状态修改 fd\_set 的内容，其实现是透明的，可以通过以下的宏来完成相应操作。

- FD\_ZERO(fd\_set \*fdset): 清空 fdset 与所有描述符的联系。
- FD\_SET(int fd, fd\_set \*fdset): 建立描述符 fd 与 fdset 的联系。
- FD\_CLR(int fd, fd\_set \*fdset): 撤销描述符 fd 与 fdset 的联系。
- FD\_ISSET(int fd, fdset \*fdset): 检查与 fdset 联系的描述符 fd 是否可读写，返回非 0 表示可读写。

### 4. Select()函数实现 I/O 多路复用的步骤

采用 select()函数实现 I/O 多路复用的基本步骤如下。

- (1) 清空描述符集合；
- (2) 建立需要监视的描述符与描述符集合的联系；
- (3) 调用 select()函数；
- (4) 检查所有需要监视的描述符，利用 FD\_ISSET 宏判断是否已准备好；
- (5) 对已准备好的描述符进行 I/O 操作。

模板如下：

```
1 ...
2 int fd, allfd[MAX];
3 fd_set fdRSet, fdWSet;
4 struct timeval timeout = ...;
5 int maxfd = MAXSOCKET + 1;
```

```
6 ...
7 while (1) {
8     FD_ZERO(&fdRSet);
9     FD_SET(fd, &fdRSet);
10    FD_ZERO(&fdWSet);
11    FD_SET(fd, &fdWSet);
12    switch (select(maxfd, &fdRSet, &fdWSet, &timeout)) {
13        case -1: handle exception;
14        case 0:  handle timeout;
15        default:
16            for (All of socket) {
17                if ( FD_ISSET ( allfd[i], &fdRSet)) {
18                    Read from the socket.
19                    ...
20                }
21                if ( FD_ISSET ( allfd[i], &fdWSet)) {
22                    Write to the socket.
23                    ...
24                }
25            }
26        }
27    }
28 }
```

代码说明如下。

4：定义超时值。

5：定义所监视的描述符上限 (maxfd)。

8~9：清空读描述符集。

10~11：清空写描述符集。

12：调用 select()函数。

13：处理异常。

14：处理超时。

17~20：处理可读的描述符。

21~24：处理可写的描述符。

### 6.4.3 单线程并发服务器实例

前面讨论了多进程和多线程服务器，但有时需要使用一个线程处理并发的客户请求。比如，有些系统的线程开销较大，服务器无法为每个客户连接产生一个线程。还有些系统不支持线程，但每个连接之间又需共享一些数据。

I/O 多路复用是实现单线程并发服务器的主要技术，图 6.4 已经说明了单线程并发服务器算法。

以下的实例是采用单线程并发服务器算法实现的。通过该实例可了解如何采用 I/O 多路



复用技术实现单线程并发服务器。该实例包括服务器程序，客户为采用多进程并发服务器实例的客户程序。完成的功能与线程安全的多线程并发服务器实例相同。

### 1. 程序清单（程序 6.9）

其源程序如下：

```
1  #include <stdio.h>           /* These are the usual header files */
2  #include <strings.h>         /* for bzero() */
3  #include <unistd.h>          /* for close() */
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>
8  #include <sys/time.h>
9  #include <stdlib.h>

10 #define PORT 1234           /* Port that will be opened */
11 #define BACKLOG 5           /* Number of allowed connections */
12 #define MAXDATASIZE 1000
13 typedef struct CLIENT{
14     int    fd;
15     char*  name;
16     struct sockaddr_in addr; /* client's address information */
17     char*  data;
18 };
19 void process_cli(CLIENT *client, char* recvbuf, int len);
20 void savedata(char* recvbuf, int len, char* data);

21 main()
22 {
23     int i, maxi, maxfd, sockfd;
24     int nready;
25     ssize_t n;
26     fd_set rset, allset;
27     int listenfd, connectfd; /* socket descriptors */
28     struct sockaddr_in server; /* server's address information */
29     /* client's information */
30     CLIENT client[FD_SETSIZE];
31     char recvbuf[MAXDATASIZE];
32     int sin_size;

33     /* Create TCP socket */
34     if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
35         /* handle exception */
36         perror("Creating socket failed.");
37         exit(1);
```

```
38     }

39     int opt = SO_REUSEADDR;
40     setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

41     bzero(&server, sizeof(server));
42     server.sin_family=AF_INET;
43     server.sin_port=htons(PORT);
44     server.sin_addr.s_addr = htonl (INADDR_ANY);
45     if (bind(listenfd, (struct sockaddr *)&server,
46             sizeof(struct sockaddr)) == -1) {
47         /* handle exception */
48         perror("Bind error.");
49         exit(1);
50     }

51     if(listen(listenfd, BACKLOG) == -1){ /* calls listen() */
52         perror("listen() error\n");
53         exit(1);
54     }

55     sin_size=sizeof(struct sockaddr_in);
56     /*initialize for select */
57     maxfd = listenfd;
58     maxi = -1;
59     for (i = 0; i < FD_SETSIZE; i++) {
60         client[i].fd = -1;
61     }
62     FD_ZERO(&allset);
63     FD_SET(listenfd, &allset);

64     while(1)
65     {
66         struct sockaddr_in addr;
67         rset = allset;
68         nready = select(maxfd+1, &rset, NULL, NULL, NULL);

69         if (FD_ISSET(listenfd, &rset)) { /* new client connection */
70             /* Accept connection */
71             if ((connectfd = accept(listenfd,
72                                     (struct sockaddr *)&addr,&sin_size))== -1) {
73                 perror("accept() error\n");
74                 continue;
75             }
76             /* Put new fd to client */
```

```

75     for (i = 0; i < FD_SETSIZE; i++)
76     {
77         client[i].fd = connectfd;    /* save descriptor */
78         client[i].name = new char[MAXDATASIZE];
79         client[i].addr = addr;
80         client[i].data = new char[MAXDATASIZE];
81         client[i].name[0] = '\0';
82         client[i].data[0] = '\0';
83         printf("You got a connection from %s. ",
84             inet_ntoa(client[i].addr.sin_addr) );
85         break;
86     }
87     if (i == FD_SETSIZE)    printf("too many clients\n");
88     FD_SET(connectfd, &allset);    /* add new descriptor to set */
89     if (connectfd > maxfd)    maxfd = connectfd;
90     if (i > maxi)    maxi = i;
91     if (--nready <= 0) continue;    /* no more readable descriptors */
92 }

93 for (i = 0; i <= maxi; i++) {    /* check all clients for data */
94     if ( (sockfd = client[i].fd) < 0) continue;
95     if (FD_ISSET(sockfd, &rset)) {
96         if ( (n = recv(sockfd, recvbuf, MAXDATASIZE, 0)) == 0) {
97             /*connection closed by client */
98             close(sockfd);
99             printf("Client( %s ) closed connection. User's data: %s\n",
100                 client[i].name, client[i].data);
101             FD_CLR(sockfd, &allset);
102             client[i].fd = -1;
103             delete client[i].name;
104             delete client[i].data;
105         } else
106             process_cli(&client[i], recvbuf, n);
107     }
108 }
109 close(listenfd);    /* close listenfd */
110 }

111 void process_cli(CLIENT *client, char* recvbuf, int len)
112 {
113     char sendbuf[MAXDATASIZE];

```

```
114 recvbuf[len-1] = '\0';
115 if (strlen(client->name) == 0) {
116     /* Got client's name from client */
117     memcpy(client->name,recvbuf, len);
118     printf("Client's name is %s.\n",client->name);
119     return;
120 }

121 /* save client's data */
122 printf("Received client( %s ) message: %s\n",client->name, recvbuf);
123 /* save user's data */
124 savedata(recvbuf,len, client->data);
125 /* reverse usr's data */
126 for (int i1 = 0; i1 < len - 1; i1++) {
127     sendbuf[i1] = recvbuf[len - i1 - 2];
128 }
129 sendbuf[len - 1] = '\0';

130 send(client->fd,sendbuf,strlen(sendbuf),0);
131 }

132 void savedata(char* recvbuf, int len, char* data)
133 {
134     int start = strlen(data);
135     for (int i = 0; i < len; i++) {
136         data[start + i] = recvbuf[i];
137     }
138 }
```

## 2. 程序分析

1~9：所需的头文件。

9~12：定义端口号、最大允许连接的数量及缓冲区的大小。

13~18：定义结构 CLIENT，包括客户相关信息：连接套接字（fd），客户名（name），客户地址（addr）及客户数据（data）。

19：声明“客户处理”process\_cli()函数，用于处理客户请求。

20：声明 savedata()函数，用于实现存储客户数据。

34~38：产生 TCP 套接字。

39~40：设置套接字选项为 SO\_REUSEADDR。

41~49：绑定套接字到相应地址。本例 IP 地址设为 INADDR\_ANY，可接收来自本机任何 IP 地址的客户连接。

50~53：监听网络连接。

56：将所监视的描述符上限（maxfd）设置为监听套接字（listenfd），因为当前 listenfd 是惟一被监视的描述符。

57：将最大连接的客户数（maxi）设置为-1，表示没有连接的客户。

58~60：将所有客户的连接套接字（fd）设置为-1，表示这些客户还未连接。

61：清空描述符集（allset）。allset 用于存放所有被监视的描述符。

62：将 listenfd 放入 allset。

66：将 allset 的数据复制给 rset。rset 是可读描述符集合。由于调用 select()函数时，系统内核会根据 I/O 状态修改 rset 的内容，所以必须用 allset 来保存所有被监视的描述符，并且在每次调用 select()函数前，赋值给 rset。

67：调用 select()函数。由于只监视可读的描述符，可写描述符集合和异常描述符集合被置为 NULL。在本例中没有超时限制，将时间参数也置为 NULL，nready 存放 select()函数返回的可读描述符的数量。

68：判断监听套接字是否可读，即是否有新的客户连接。

70~73：接受客户连接。

75~85：在数组 client 寻找一空项，然后初始化新客户信息。

86：如果客户数达到最大限制，显示错误信息。

87：将新产生的连接套接字放入 allset。

88：如果新产生的连接套接字(connectfd)大于 maxfd，则将 maxfd 置为 connectfd。

90：如果没有其他可读的描述符（连接套接字），则重新调用 select()函数，等待新的可读描述符。

92~93：检查所有已连接的客户。

94：判断相应客户的连接套接字是否可读。

95：读客户发来的数据。如果字节数为 0，表示客户关闭了连接。

97~102：处理关闭了连接的客户。关闭套接字；显示客户发来的所有数据；将数据描述符从 allset 撤销并将相应客户信息从 client 中撤销。

104：调用 process\_cli()函数处理客户请求。

114：将接收缓冲区的最后一字节置为“\0”，以便于显示。

115：判断是否是新连接的客户。

117~120：将接收的客户名存放在客户信息中，并显示客户名。

122：显示收到的数据。

124：存储数据。

126~128：将收到的数据反转。

129~130：发送数据回客户。

132~138：将收到的数据存放在客户信息中。

### 3. 运行程序

下面在同一主机上运行上述程序。

（1）首先启动服务器程序：

```
selectserver
```

（2）然后同时运行客户 1 和客户 2：

```
procclient 127.0.0.1
procclient 127.0.0.1
```

#### 4. 服务器运行结果

```
$ selectserver
You got a connection from 127.0.0.1. Client's name is client1.
Received client( client1 ) message: abcd
You got a connection from 127.0.0.1. Client's name is 1234.
Received client( 1234 ) message: 5678
Client( 1234 ) closed connection. User's data: 5678
Received client( client1 ) message: efgh
Client( client1 ) closed connection. User's data: abcdefgh
```

#### 5. 客户 1 运行结果

```
$ procclient 127.0.0.1
Connected to server.
Input name : client1
Input string to server:abcd
Server Message: dcba
Input string to server:efgh
Server Message: hgfe
Input string to server: ^D
Exit.
$
```

#### 6. 客户 2 运行结果

```
$ procclient 127.0.0.1
Connected to server.
Input name : 1234
Input string to server:5678
Server Message: 8765
Input string to server: ^D
Exit.
$
```

#### 7. 结论

从上述例子可以看出,用 I/O 多路复用方法实现的并发服务器程序,其程序结构远不如多进程/线程并发服务器清晰,因而对于复杂的应用,其设计和调试更为困难,并且可读性较差。

## 6.5 套接字终止处理

通常使用 `close()` 函数终止一个连接，但该函数有如下两点限制。

- `Close()` 函数只是减少描述符的参考数，并不直接关闭连接。只有当描述符的参考数为零时，系统才进行关闭连接的操作。但在某些情况下，需要直接关闭连接而不考虑描述符的参考数。
- `Close()` 函数将终止双向的数据传输，即终止读和写操作。由于 TCP 连接是全双工的，在某些情况下，只需关闭一个方向的数据传输。

在默认情况下，`close()` 函数将 TCP 套接字标志为关闭并且立即返回。这时，套接字描述符不能再使用，即无法进行读/写操作。但系统仍试图将套接字发送缓冲区的内容发往对方。如果描述符参考为 0，系统将发送 TCP 终止序列 (FIN)。而套接字接收缓冲区的内容将被丢弃。在某些情况下，会造成数据的丢失。例如，当一方 A 用 `close()` 函数关闭套接字，同时另一方 B 正在发送数据。由于 A 无法再接收数据，即使 B 的数据到达 A，也将被系统丢弃。其过程如图 6.10 所示。

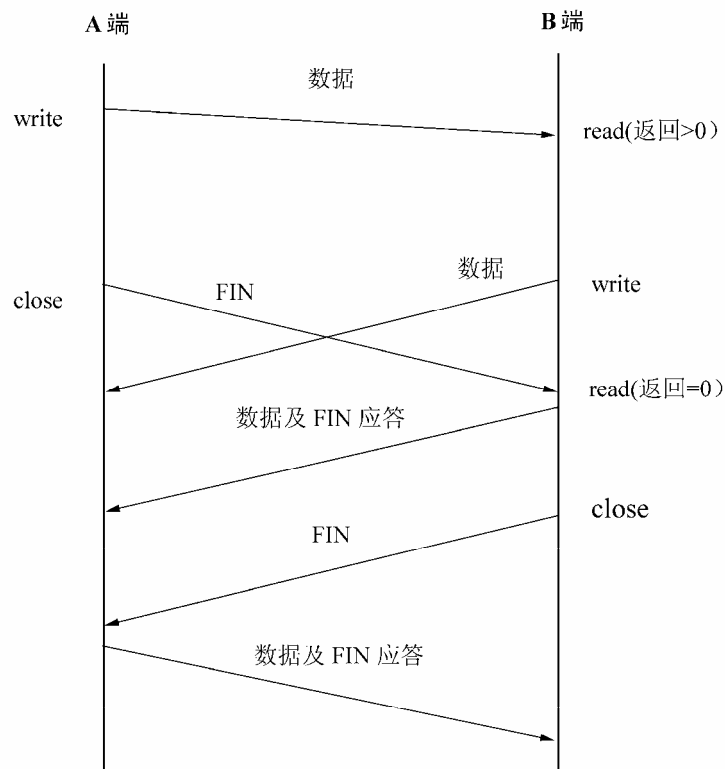


图 6.10 调用 `close()` 函数关闭 TCP 连接

为避免上述问题，可采用 `shutdown()` 函数来关闭连接。`Shutdown()` 函数可直接关闭套接

字描述符,也不考虑其参考数是否为 0,并且它可选择仅终止一个方向的连接,另一方面的连接能继续工作。其典型工作过程如图 6.11 所示。

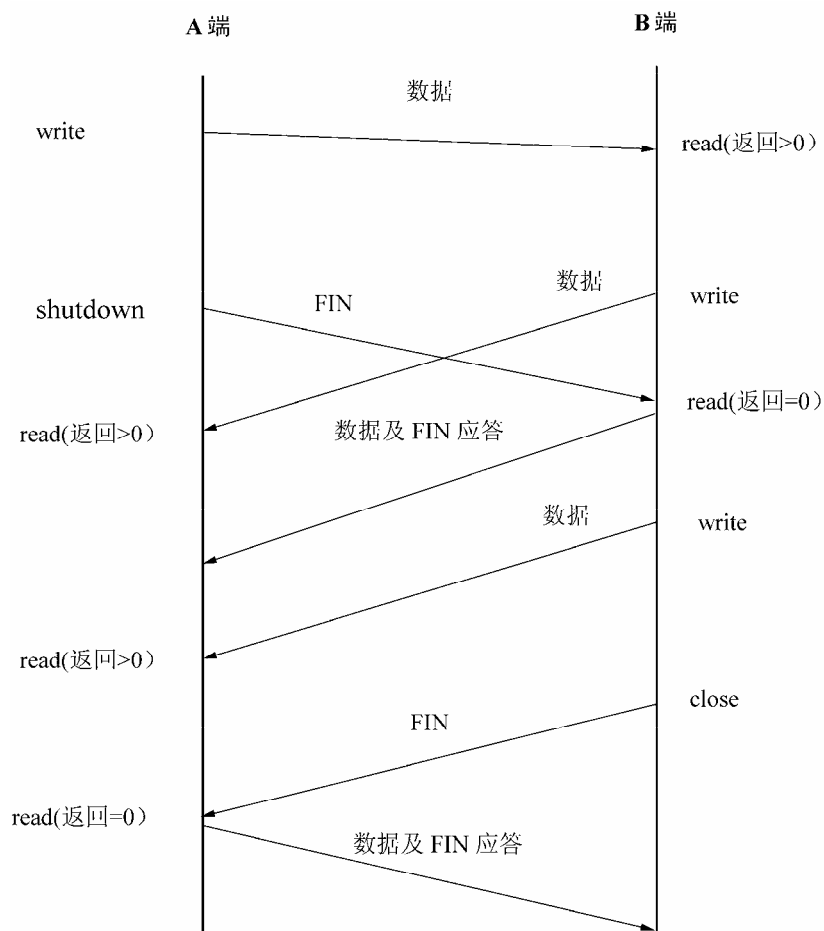


图 6.11 调用 `shutdown()` 函数关闭 TCP 连接

如图 6.11 所示,当 A 调用 `shutdown()` 函数关闭写的连接后,它仍可对套接字描述符进行读操作,从而保证了数据不会丢失。当然,A 可以在确保所有数据都收到后,才调用 `close()` 函数,这样也能保证数据不会丢失,但应用程序需要做更多的工作。

## 6.6 小 结

Unix 系统主要提供三种方式以实现并发服务器:进程、线程及 I/O 多路复用。

采用多进程实现并发服务器是一种最传统的方法。由于各进程具有独立的地址空间,具有可靠性高的特点。但却导致系统开销大,切换时间长,并且进程间共享数据较为复杂。

虽然多线程具有系统开销小、切换速度快等特点,但由于多个线程共用相同的内存区,



从而造成多线程比多进程更严重的可重入问题。尤其是并发服务器的编程，如果不注意这个问题，会产生很多难以调试的 bug。其根本解决方法是：为线程专有的数据分配线程专用的数据区。可采用系统提供的方法或主线程为每个新产生的线程预分配数据区。

I/O 多路复用具有极小的系统开销，因此其性能很高，但其程序结构较为复杂。

由此看来，每种实现方法各有其优缺点，要根据实际应用系统的特点来进行选择。

## 第 7 章 名字和 IP 地址转换

本章介绍名字和 IP 地址的转换方法。在实际应用中，是以域名进行通信的，所以必须将域名转换为 IP 地址。

### 7.1 名字解析

虽然与套接字相关的函数都是使用 IP 地址来进行与远程主机通信的。但实际应用中，人们通常只知道服务器的名字，而非 IP 地址。从而需要将名字转换成相应的 IP 地址，这就是名字解析。

在 Unix 系统中，进程通常采用以下几种方式进行名字解析。

- 查询本机文件/etc/hosts。该文件是一个主机名与 IP 地址的对应表。由于它只在本机有效，因而，必须对每台主机单独维护，对于较大的网络，其维护量极大。
- 网络信息系统（NIS）。可在一个本地/内部网络内提供名字解析。通常 Sun 的系统采用这种方式。
- 域名系统（DNS）。可在 Internet 中提供名字解析。因此，它得到了最广泛的应用。

不同的系统会采用不同的方式进行名字解析，其具体实施由网络管理员负责。对于网络编程而言，并不必关心系统采用何种方式，只需通过相应的系统调用，得到解析结果。

### 7.2 套接字地址

#### 7.2.1 地址结构

在套接字编程中，主要涉及以下两个地址结构。

- sockaddr\_in: IPv4 套接字地址结构。
- sockaddr: 通用套接字地址结构。

IPv4 套接字地址结构提供 IPv4 套接字地址的相关信息（地址簇、端口号和 IP 地址）。而通用套接字地址结构由系统内核使用，适用于多种协议的地址。对于不同网络协议，它可映射到不同的地址结构中，反之亦然。例如，对于广泛使用的 IPv4 协议，sockaddr\_in 与 sockaddr 可相应转换。前面章节已讲述，为了使用套接字函数（如 bind() 函数），首先将相关地址信息填入 sockaddr\_in，再将其转换成 sockaddr。

## 7.2.2 字节顺序

### 1. 字节顺序的概念

在同一网络中，有不同的主机，这些主机为了相互通信，必须定义一个表示数据的标准。其中字节顺序是非常重要的。字节顺序包括两类：

- 高字节在前，低字节在后
- 低字节在前，高字节在后

IP 协议的网络字节顺序为高字节在前。而不同的主机系统却可能采用不同的字节顺序。从而造成主机字节顺序与网络字节顺序的不一致。在 IP 包中，有些信息，如目的地址和包长度，必须被发送方和接收方正确地理解。

TCP/IP 协议规定其网络字节顺序，所有主机或路由器在发送 IP 包前首先要将相应信息转换成网络字节顺序。反之，在接收 IP 包后，要将网络字节顺序转换成主机字节顺序。例如，Sun 工作站采用高字节在前的字节顺序，而 VAX 则采用低字节在前的字节顺序。

如果不进行字节顺序的转换，套接字程序可以在 Sun 工作站正确运行，而在 VAX 上却不行。通常，采用调用相应的字节顺序转换函数解决上述问题。Unix 系统提供了相应的函数，并且，如果相应的主机系统的字节顺序和网络字节顺序一致，这些函数被定义为空的宏。

因此，无论主机系统采用何种字节顺序，均要调用相应的字节顺序转换函数，这样可保证程序能在不同的主机系统运行，并且不影响效率。

### 2. 字节顺序的转换函数

主机字节顺序与网络字节顺序的转换函数原型如下：

```
#include <sys/types.h>
#include <netinet/in.h>
#include <inttypes.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

在这 4 个转换函数中，h 代表主机 (host)，n 代表网络 (network)，s 代表短整数 (short)，l 代表长整数 (long)。

htonl：将本机的长整数转换成网络上的长整数。由于历史原因称之为长整数，其实是 32 位的整数。

htons：将本机的短整数转换成网络上的短整数。

ntohl：将网络上的长整数转换成本机的长整数。

ntohs：将网络上的短整数转换成本机的短整数。

### 7.2.3 IP 地址转换函数

为便于表达, IP 地址都是数字加点(127.0.0.1)构成的, 而在 in\_addr 结构中用的是 32 位的 IP。为方便字符串的 IP 和 32 位的 IP 相互转换, Unix 系统提供了相应的函数。常用的函数包括:

```
#include <arpa/inet.h>
in_addr_t inet_addr(const char *cp);
int inet_aton(const char *cp, struct in_addr *addrp)
char *inet_ntoa(struct in_addr inaddr)
```

函数中的 a 代表 ascii, n 代表网络 (network)。

inet\_addr: 将字符串形式的 IP 地址转换成 32 位的 IP 地址。cp 指向字符串形式的 IP 地址。函数返回 32 位的内部存储格式的 IP 地址 (网络字节顺序)。如果是无效的 IP 地址则返回-1。

inet\_aton: 将字符串形式的 IP 地址转换成 32 位的 IP 地址。cp 指向字符串形式的 IP 地址, addrp 指向 32 位的 IP 地址 (网络字节顺序)。成功返回 1, 否则返回 0。

inet\_ntoa: 将 32 位形式的 IP 地址转换成字符串形式的 IP 地址。inaddr 指向 32 位形式的 IP 地址 (网络字节顺序)。返回指向字符串形式的 IP 地址的指针。该指针指向静态的内存区, 因此, 此函数不是线程安全的。

以下实例说明了这些函数的使用:

```
1 #include <sys/socket.h>
2 #include <netinet/in.h>
3 #include <arpa/inet.h>
4 struct sockaddr_in my_addr;
5 char *a1;
6 ...
7 my_addr.sin_family = AF_INET;           // host byte order
8 my_addr.sin_port = htons(MYPORT);      // short, network byte order
9 inet_aton("192.9.200.10", &(my_addr.sin_addr));
10 memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct
11 ...
12 a1 = inet_ntoa(my_addr.sin_addr); // this is 192.9.200.10
13 printf("address : %s\n", a1);
```

代码说明如下。

7~8: 设置套接字地址。

9: 将字符串形式的 IP 地址 “ 192.9.200.10 ” 转换成 32 位的 IP 地址。

12: 将 32 位形式的 IP 地址转换成字符串形式的 IP 地址。

13: 输出字符串形式的 IP 地址。输出结果如下:

```
address : 192.9.200.10
```

### 7.2.4 套接字地址信息函数

在网络编程中，经常需要获得本地或远程的套接字地址信息（端口号、IP 地址等）。Unix 提供了 `getsockname()` 和 `getpeername()` 函数以提供套接字地址信息。

通常，我们知道本地/远程的套接字地址信息，不必调用专门的函数来获得，但以下情况却是例外。

- TCP 客户不调用 `bind()` 而直接调用 `connect()` 函数。此时，应调用 `getsockname()` 函数才能获得由系统自动分配给该连接的 IP 地址和端口号。
- 调用 `bind()` 函数，且端口参数设为 0（让核选择相应端口）时，则调用 `getsockname()` 函数来获得本地端口号。
- TCP 服务器调用 `bind()` 函数，且 IP 地址参数为 `INADDR_ANY`，则调用 `getsockname()` 函数来获得由核分配的本地 IP 地址。
- 在子进程中，由于执行了 `exec()` 函数而丢失了原内存的数据，则调用 `getpeername()` 函数获得远程的套接字地址信息。

#### 1. Getsockname()函数

`getsockname()` 函数用于获得本地的套接字地址信息。函数原型如下：

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

int getsockname(int s, struct sockaddr *name, socklen_t *namelen);
```

s：套接字描述符。

name：函数成功返回后，指向本地套接字地址信息。

namelen：函数成功返回后，指向套接字地址信息的长度。

返回值：成功返回 0，否则返回 -1，并设置 `errno` 值。

#### 2. Getpeername()函数

`getpeername()` 函数用于获得远端套接字地址信息。函数原型如下：

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

int getpeername(int s, struct sockaddr *name, socklen_t *namelen);
```

s：套接字描述符。

name：函数成功返回后，指向远程套接字地址信息。

namelen：函数成功返回后，指向套接字地址信息的长度。

返回值：成功返回 0，否则返回 -1，并设置 `errno` 值。

以下通过实例说明其使用：

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
```

```
3  int fd, len;
4  struct sockaddr_in server, local;
5  ...
6  if(connect(fd, (struct sockaddr *)&server,
             sizeof(struct sockaddr)==-1){ /* calls connect() */
7      printf("connect() error\n");
8      exit(1);
9  }
10 if (getsockname(fd, (sockaddr *)&local, &len) == -1) return;
11 if (getpeername(fd, (sockaddr *)&server, &len) == -1) return;
12 printf("Local IP addr is %s, local port is %s\n",
        inet_ntoa(local.sin_addr) , ntohs(local.sin_port));
13 printf("Server's IP addr is %s, server's port is %s\n",
        inet_ntoa(server.sin_addr), ntohs(server.sin_port));
14 ...
```

代码说明如下。

1~2：所需头文件。

6~9：与服务器连接。

10：获得本地的套接字地址信息。存放于 local 变量。

11：获得远端套接字地址信息。存放于 server 变量。

12：显示本地主机 IP 地址和端口号。

13：显示服务器 IP 地址和端口号。

## 7.3 套接字信息函数

在实际的网络编程中，经常需要知道一些与套接字相关的信息。Unix 中提供了相应的套接字信息函数。以下介绍常用的几个函数。

### 7.3.1 主机名转换为 IP 地址：gethostbyname()函数

gethostbyname()函数提供名字解析的功能，将主机名转换成相应的 IP 地址。

#### 1. 函数原型

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
```

name：指向主机名。

返回值：成功返回 hostent 指针，否则返回 NULL 并设置 errno 值。

hostent 结构用于描述一个主机实体，包括主机的名字与地址信息。其结构定义如下：

```
struct hostent {
    char    *h_name;           /* canonical name of host */
```

```

char    **h_aliases;    /* alias list */
int     h_addrtype;    /* host address type */
int     h_length;      /* length of address */
char    **h_addr_list; /* list of addresses */
};
#define h_addr h_addr_list[0]

```

\*h\_name：主机的正式名称。

\*\*h\_aliases：主机的别名列表。

h\_addrtype：主机的地址类型。

h\_length：主机的地址长度，对于 IP4 是 4 字节 32 位的地址。

\*\*h\_addr\_list：主机的 IP 地址列表。

h\_addr：主机的第一个 IP 地址。

其中别名列表和 IP 地址列表是一个以 NULL 结尾的列表。

由于返回的 hostent 指针指向一静态存储区，所以该函数不是线程安全的函数。其相应的线程安全的函数为 gethostbyname\_r()，该函数原型如下：

```

struct hostent *gethostbyname_r(const char *name, struct hostent *result,
                                char *buffer, intbuflen, int *h_errnop);

```

name：指向主机名。

result：指向 hostent 结构，其内存空间由调用函数提供。当函数成功返回后，存放主机实体。

buffer：指向一内存区，该内存区应能存放所有的信息。由于实体信息包括两个列表，因此大小是不确定，要考虑最大的可能。

h\_errnop：存放错误信息。

## 2. 实例（程序 7.1）

以下例子说明了 gethostbyname()函数的使用。它通过用户输入的主机名获得主机实体，然后显示主机的正式名字、别名和 IP 地址。

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <arpa/inet.h>
7 #include <netdb.h>

8 main(int argc, const char **argv)
9 {
10     ulong_t addr;
11     struct hostent *hp;
12     char **p;
13     if (argc != 2) {

```

```

14     (void) printf("usage: %s host_name\n", argv[0]);
15     exit (1);
16 }
17 hp = gethostbyname(argv[1]);
18 if (hp == NULL) {
19     (void) printf("host information for %s not found\n", argv[1]);
20     exit (2);
21 }
22 for (p = hp->h_addr_list; *p != 0; p++) {
23     struct in_addr in;
24     char **q;
25     (void) memcpy(&in.s_addr, *p, sizeof (in.s_addr));
26     (void) printf("%s\t%s", inet_ntoa(in), hp->h_name);
27     for (q = hp->h_aliases; *q != 0; q++)
28         (void) printf(" %s", *q);
29     (void) putchar('\n');
30 }
31 exit (0);
32 }

```

代码说明如下。

1~7：所需头文件。

13~16：如果有多个命令参数，则显示命令的用法。

17~21：通过用户输入的主机名获得主机实体（hp）。

22~30：逐项显示主机的 IP 地址、主机名和别名。

### 3. 运行程序

```
$ gethost_n zj
```

如果例子保存名为 gethost\_n，要查找主机名为 zj 的 IP 地址，输入如下命令。

### 4. 运行结果

```
127.0.0.1      localhost loghost zj
$
```

### 5. 运行结果说明

运行结果说明了 gethostbyname()函数的使用。

## 7.3.2 IP 地址转换为主机名：gethostbyaddr()函数

该函数提供反向地址解析功能，即将 IP 地址转换成主机名。

### 1. 函数原型

```

#include <netdb.h>
struct hostent *gethostbyaddr(const char *addr, int len, int type);

```



addr：指向 IP 地址。它是一个指向地址结构为 in\_addr 的指针。

len：地址长度。

type：地址簇（AF\_INET）。

返回值：成功返回主机的实体结构指针，否则返回 NULL 并设置 errno 值。

相应的线程安全函数为 gethostbyaddr\_r()，原型如下：

```
struct hostent *gethostbyaddr_r(const char *addr, int length, int type,
                                struct hostent *result, char *buffer, int buflen, int *h_errnop);
```

addr：指向 IP 地址。

length：地址长度。

type：地址簇（AF\_INET）。

result：指向 hostent 的指针，内存已由调用函数分配。

buffer：用于存放实体信息的缓冲区。

buflen：缓冲区大小。

h\_errnop：存放错误信息。

## 2. 实例（程序 7.2）

以下例子说明了 gethostbyaddr()函数的使用。通过用户所输入的字符串形式的 IP 地址来获得主机实体，然后显示主机的正式名字、别名和 IP 地址。

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <netinet/in.h>
6 #include <arpa/inet.h>
7 #include <netdb.h>

8 main(int argc, const char **argv)
9 {
10     ulong_t addr;
11     struct hostent *hp;
12     char **p;
13     if (argc != 2) {
14         (void) printf("usage: %s IP-address\n", argv[0]);
15         exit (1);
16     }
17     if ((int)(addr = inet_addr(argv[1])) == -1) {
18         (void) printf("IP-address must be of the form a.b.c.d\n");
19         exit (2);
20     }
21     hp = gethostbyaddr((char *)&addr, sizeof (addr), AF_INET);
22     if (hp == NULL) {
23         (void) printf("host information for %s not found\n", argv[1]);
```

```
24     exit (3);
25     }
26     for (p = hp->h_addr_list; *p != 0; p++) {
27         struct in_addr in;
28         char **q;
29         (void) memcpy(&in.s_addr, *p, sizeof (in.s_addr));
30         (void) printf("%s\t%s", inet_ntoa(in), hp->h_name);
31         for (q = hp->h_aliases; *q != 0; q++)
32             (void) printf(" %s", *q);
33         (void) putchar('\n');
34     }
35     exit (0);
36 }
```

代码说明如下。

1~7：所需头文件。

13~16：如果有多个命令参数，则显示命令的用法。

17~20：通过用户输入的字符串形式的主机 IP 地址可获得地址结构为 in\_addr 形式的 IP 地址。

21~25：通过 IP 地址获得主机实体 (hp)。

26~30：逐项显示主机的 IP 地址、主机名和别名。

### 3. 运行程序

要查找地址为 “127.0.0.1” 的主机，可输入命令 “gethost\_a 127.0.0.1”。

### 4. 运行结果

```
$ gethost_a 127.0.0.1
127.0.0.1      localhost
$
```

### 5. 运行结果说明

运行结果说明了 gethostbyaddr() 函数的使用。

## 7.3.3 获得服务的端口号：getservbyname() 函数

不同的服务对应着不同的端口号。与主机相同，通常用名字来标识服务。Getservbyname() 函数可通过服务名来获得相应的端口号。函数原型如下：

```
#include <netdb.h>
struct servent *getservbyname(const char *name, const char *proto);
```

name：指向服务名。

proto：指向协议名称，如 TCP。

返回值：成功返回一 servent 结构指针，否则返回 NULL。

servent 结构用于存储服务实体的信息，如端口号、别名、使用的协议等。其定义如下：

```
struct servent {
    char *s_name;           /* official name of service */
    char **s_aliases;       /* alias list */
    int s_port;             /* port service resides at */
    char *s_proto;          /* protocol to use */
};
```

\*s\_name：正式服务名。

\*\*s\_aliases：别名列表，以 NULL 结尾。

s\_port：端口号。

\*s\_proto：使用的协议。

getservbyname 函数不是线程安全的，其线程安全的函数为 getservbyname\_r()，原型如下：

```
struct servent *getservbyname_r(const char *name, const char *proto,
                                struct servent *result, char *buffer, int buflen);
```

### 7.3.4 端口号转换为服务名：getservbyport()函数

getservbyport()函数用于将端口号转换成相应的服务名。其函数原型如下：

```
struct servent *getservbyport(int port, const char *proto);
```

port：端口号。

proto：指向协议名称。

返回值：成功返回一 servent 结构指针，否则返回 NULL。

其线程安全的函数为 getservbyport\_r()，原型如下：

```
struct servent *getservbyport_r(int port, const char *proto,
                                struct servent *result, char *buffer, int buflen);
```

## 7.4 小 结

由于主机字节顺序与网络字节顺序的不同，需要进行字节顺序的转换，否则会造成程序错误。相关的函数包括：htonl()、htons()、ntohl()和 ntohs()。

同样，在网络编程中，名字和 IP 地址转换函数也是常用的，包括函数 inet\_addr()、inet\_aton()和 inet\_ntoa()。

为获得本地或远程套接字信息，可调用 getsockname()和 getpeername()函数。另外，Unix 还提供了大量函数以获得与套接字相关的信息，主要有 gethostbyname()、gethostbyaddr()、getservbyname() 及 getservbyport()等。需要注意的是，这些函数不是线程安全的，在多线程环境应使用它们的线程安全版本。

## 第 8 章 同步及进程间通信

在多进程/多线程环境中，线程及进程同步是十分重要的。该技术解决了由并发而产生的同步问题，使得并发服务器能可靠地运行。

本章主要对线程同步、进程同步及进程间通信技术进行讲解，并通过例子予以说明。

### 8.1 线程同步

#### 8.1.1 线程同步基础

多线程并发服务器虽然有很多优点，但是存在一些问题，需要十分注意。除了前面讲到的线程安全性问题外，另一重要的方面就是线程同步。线程在处理共享数据和进程资源是必须使用同步机制。

例如，两个线程同时修改一个结构（employee），其定义如下：

```
struct employee
{
    int id;
    char name[10]
}
```

由于两个线程（a 和 b）同时运行，就有可能线程 a 修改了变量 id 后，线程 b 开始运行并修改了变量 id 和 name，然后线程 a 又修改了变量 name。此时，employee 中的 id 变量值是线程 b 的，而 name 是线程 a 的，从而造成 id 和 name 的不一致。这就是由两个进程不同步造成的。

#### 8.1.2 互斥锁基础

##### 1. 互斥锁的概念

在 Unix 系统中，提供一种基本的线程同步机制——互斥锁。它是一种锁，可以用来保护线程代码中共享数据结构的完整性。它具有以下 3 个特点。

- 对互斥锁的操作（加锁/解锁）是原子操作，这就意味着操作系统将保证同时只有一个线程能成功完成对一个互斥锁的加锁操作。
- 如果一个线程已经对某一互斥锁进行了加锁，其他线程只有等待该线程完成对这一互斥锁解锁后，才能完成加锁操作。
- 如果一个线程已经对某一加锁的互斥锁进行了加锁操作，该线程将被挂起。只有当该互斥锁被解锁后，该线程才被唤醒并完成加锁操作。

利用互斥锁可以保证在某一时间内，只有一个线程能执行“关键”代码。通常，这些“关键”代码是用于修改共享数据的，从而保证其数据的完整性。具体的实现方法是：在“关键”代码前加锁某一互斥锁，在“关键”代码结束时解锁该互斥锁。对于上一节的例子，可用以下伪代码实现数据的完整性。

```
加锁互斥锁 x1;
修改变量 id;
修改变量 name;
解锁互斥锁 x1。
```

与上一节相同，线程 a 首先运行，当修改了变量 id 后，线程 b 运行上述代码，而此时线程 a 已锁定了同一互斥锁 X1，线程 b 被挂起来直至线程 a 完成所有修改工作并解锁该互斥锁。最后运行结果 id 和 name 都是由线程 b 修改的，从而保证了数据的完整性。

在互斥锁的应用中，只有使用同一互斥锁才能实现“互斥”。在上例中，如果存在线程 c，它采用同样“关键”代码来修改 employee 中的变量 id 和 name，但采用了不同的互斥锁 X2，此时，employee 的数据完整性仍无法保证。

如果设置了过多的互斥锁，代码就没有什么并发性可言，运行起来也比单线程解决方案慢。如果设置了过少的互斥锁，无法保证数据的完整性。因此，如果要使用共享数据，那么在读、写共享数据时都应使用互斥锁，不要对非共享数据使用互斥锁。并且，如果程序逻辑确保任何时候都只有一个线程能存取特定数据结构，那么也不要使用互斥锁。

另外，对于“关键”代码都应采用互斥锁来进行保证。即使“关键”代码只有一条语句，因为一条语句可能会被编译成多条机器指令，也就有可能在完成该语句前插入其他线程的运行。

## 2. 互斥锁的初始化和消除

在使用互斥锁前，必须对其进行初始化。初始化互斥锁可采用静态或动态的初始化方法。

对于静态初始化方法，需要声明一个 `pthread_mutex_t` 变量，并赋值为常数 `PTHREAD_MUTEX_INITIALIZER`：

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

当代码使用 `malloc()` 函数分配一个新的互斥锁时，应当使用动态初始化方法。要动态地创建互斥锁可使用 `pthread_mutex_init()` 函数，其函数原型如下：

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
```

mutex：指向一个将被初始化的互斥锁。

attr：一个可选的 `pthread_mutexattr_t` 指针。这个结构可用来设置互斥锁的各种属性。但是通常并不需要这些属性，所以一般把它指定为 `NULL`。

返回值：成功时都返回 0，否则返回错误码。

一旦使用 `pthread_mutex_init()` 函数初始化了互斥锁，当不用它时，就应使用 `pthread_mutex_destroy()` 函数撤销它。`pthread_mutex_destroy()` 函数的原型如下：

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

`mutex`：指向一个将被撤销的互斥锁。

返回值：成功时都返回 0，否则返回错误码。

`pthread_mutex_destroy()` 函数释放创建互斥锁时分配给它的任何资源。但不会释放用来存储 `pthread_mutex_t` 的内存。另外，不要撤销已被加锁的互斥锁。

### 8.1.3 加锁和解锁互斥锁

#### 1. 加锁函数 `pthread_mutex_lock()`

`pthread_mutex_lock()` 函数接受一个指向互斥锁的指针作为参数并将其锁定。如果互斥锁已经被锁定，调用者将进入睡眠状态。函数返回时，将唤醒调用者。该函数原型如下：

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

`mutex`：指向互斥锁的指针。

返回值：成功时都返回 0，否则返回错误码。

#### 2. 加锁函数 `pthread_mutex_trylock()`

`pthread_mutex_trylock()` 函数也可用于互斥锁的锁定，并且如果线程正在做其他事情（由于互斥锁当前是锁定的），而希望锁定该互斥锁，这个调用就相当方便。调用 `pthread_mutex_trylock()` 函数时将尝试锁定互斥锁。如果互斥锁当前处于解锁状态，那么完成锁定并且函数将返回零。然而，如果互斥锁已锁定，这个调用也不会阻塞。它会返回非零的 `EBUSY` 错误值。然后可以继续做其他事情，稍后再尝试锁定。该函数原型如下：

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

#### 3. 解锁函数 `pthread_mutex_unlock()`

`pthread_mutex_unlock()` 函数用于互斥锁的解锁操作。该函数原型如下：

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

`mutex`：指向互斥锁的指针。

返回值：成功时都返回 0，否则返回错误码。

使用互斥锁时，应该尽快对已加锁的互斥锁进行解锁（以提高性能）。并且绝对不要对未加锁的互斥锁进行解锁操作（否则，`pthread_mutex_unlock()` 函数调用将失败并返回一个

非零的 EPERM 返回值)。

#### 4. 加锁/解锁函数的使用方法

对互斥锁进行加锁/解锁的典型使用方法如下：

```
1 int rc = pthread_mutex_lock(&a_mutex);
2 if (rc) { /* an error has occurred */
3     perror("pthread_mutex_lock");
4     pthread_exit(NULL);
5 }
6 /* critical code */
7 .
8 .
9 .
10 rc = pthread_mutex_unlock(&a_mutex);
11 if (rc) {
12     perror("pthread_mutex_unlock");
13     pthread_exit(NULL);
14 }
```

代码说明如下。

1~5：加锁操作。

6~9：关键代码。

10~14：解锁操作。

#### 5. 互斥锁的例子 (程序 8.1)

以下例子说明如何使用互斥锁。该例子包括三个线程，第 1 个线程反复将雇员 1 的记录 (employee[1]) 拷入 winner 记录中；第 2 个线程反复将雇员 2 的记录 (employee[2]) 拷入 winner 记录中；第 3 个线程 (主线程) 则检查 winner 记录中的数据是否完整。源程序清单如下：

```
1  #include <stdio.h>
2  #include <pthread.h>    /* pthread functions and data structures */
3
4  #define NUM_EMPLOYEES 2 /* size of each array */
5
6  /* global mutex for our program. assignment initializes it */
7  pthread_mutex_t a_mutex = PTHREAD_MUTEX_INITIALIZER;
8
9  struct employee {
10     int number;
11     int id;
12     char first_name[20];
13     char last_name[30];
14     char department[30];
15     int room_number;
```

```
13  };

14  /* global variable - our employees array, with 2 employees */
15  struct employee employees[] = {
16      { 1, 12345678, "three", "zhang", "Accounting", 101},
17      { 2, 87654321, "four", "li", "Programmers", 202}
18  };

19  /* global variable - employee of the day. */
20  struct employee winner;

21  /* function to copy one employee struct into another */
22  void copy_employee(struct employee* from, struct employee* to)
23  {
24      int rc; /* contain mutex lock/unlock results */

25      /* lock the mutex, to assure exclusive access to 'a' and 'b'. */
26      rc = pthread_mutex_lock(&a_mutex);

27      to->number = from->number;
28      to->id = from->id;
29      strcpy(to->first_name, from->first_name);
30      strcpy(to->last_name, from->last_name);
31      strcpy(to->department, from->department);
32      to->room_number = from->room_number;

33      /* unlock mutex */
34      rc = pthread_mutex_unlock(&a_mutex);
35  }

36  /* function to be executed by the variable setting threads thread */
37  void* do_loop(void* data)
38  {
39      int my_num = *((int*)data); /* thread identifying number */

40      while (1) {
41          /* set employee of the day to be the one with number 'my_num'. */
42          copy_employee(&employees[my_num-1], &winner);
43      }
44  }

45  /* main thread*/
46  int main(int argc, char* argv[])
47  {
48      int i; /* loop counter */
```



```
49 int thr_id1;          /* thread ID for the first new thread */
50 int thr_id2;          /* thread ID for the second new thread */
51 pthread_t p_thread1;  /* first thread's structure */
52 pthread_t p_thread2;  /* second thread's structure */
53 int num1 = 1;         /* thread 1 employee number */
54 int num2 = 2;         /* thread 2 employee number */
55 struct employee eotd; /* local copy of 'winner'. */
56 struct employee* worker; /* pointer to currently checked employee */

57 /* initialize winner to first 1. */
58 copy_employee(&employees[0], &winner);

59 /* create a new thread that will execute 'do_loop()' with '1' */
60 thr_id1 = pthread_create(&p_thread1, NULL, do_loop, (void*)&num1);
61 /* create a second thread that will execute 'do_loop()' with '2' */
62 thr_id2 = pthread_create(&p_thread2, NULL, do_loop, (void*)&num2);

63 /* run a loop that verifies integrity of 'winner' many */
64 for (i=0; i<10000; i++) {
65     /* save contents of 'winner' to local 'worker'. */
66     copy_employee(&winner, &eotd);
67     worker = &employees[eotd.number-1];

68     /* compare employees */
69     if (eotd.id != worker->id) {
70         printf("mismatching 'id' , %d != %d (loop '%d')\n", eotd.id,
71             worker->id, i);
72         exit(0);
73     }
74     if (strcmp(eotd.first_name, worker->first_name) != 0) {
75         printf("mismatching 'first_name' , %s != %s (loop '%d')\n",
76             eotd.first_name, worker->first_name, i);
77         exit(0);
78     }
79     if (strcmp(eotd.last_name, worker->last_name) != 0) {
80         printf("mismatching 'last_name' , %s != %s (loop '%d')\n",
81             eotd.last_name, worker->last_name, i);
82         exit(0);
83     }
84     if (strcmp(eotd.department, worker->department) != 0) {
85         printf("mismatching 'department' , %s != %s (loop '%d')\n",
86             eotd.department, worker->department, i);
87         exit(0);
88     }
89     if (eotd.room_number != worker->room_number) {
90         printf("mismatching 'room_number' , %d != %d (loop '%d')\n",
91             eotd.room_number, worker->room_number, i);
92         exit(0);
93     }
94 }
```

```
86     printf("mismatching 'room_number' , %d != %d (loop '%d')\n",
           eotd.room_number, worker->room_number, i);
87     exit(0);
88 }
89 }

90 printf("Employees contents was always consistent\n");

91 return 0;
92 }
```

代码说明如下。

6~13：定义雇员记录。

15~18：初始化雇员记录。

22~35：拷贝雇员记录，受互斥锁保护。

36~44：线程所执行的函数。反复拷贝指定的雇员记录到 winner 记录中。

60~62：产生两个线程，执行拷贝雇员记录。

63~89：反复检查 winner 的数据完整性。如果不完整则显示相应信息并退出程序。

读者可去掉与互斥锁相关代码（第 26、34 行），然后与上述例子的运行结果进行比较。

#### 6．运行程序

输入命令 mutex。

#### 7．运行结果

```
$ mutex
Employees contents was always consistent
$
```

#### 8．运行结果说明

运行结果说明，在多线程环境下，利用互斥锁可保护共享数据的完整性。

### 8.1.4 条件变量

#### 1．条件变量的概念

在网络编程中，为提高效率或模拟真实的工作过程，常常采用如下工作模式：一个线程用于接收客户请求，并将请求插入到一个队列中。而其他线程检查该队列，如果有客户请求，则进行处理。显然，该队列是一个共享的数据区，在实现时必须保证其数据的完整性。

如果只用互斥锁来实现其数据对象的完整性，那么代码需要反复对互斥锁进行锁定和解锁，以检查“请求队列”是否有新的客户请求。同时，还要快速将互斥锁解锁，以便其他线程能够进行任何必需的更改。这是一种效率极低的方法，因为线程需要在合理的时间范围内频繁地循环检测变化。

在每次检查之间，可以让调用线程短暂地进入睡眠，但是这样线程代码就无法快速做出响应。真正需要的是这样一种方法，当线程在等待满足某些条件时使线程进入睡眠状态。一旦条件满足，还需要一种方法以唤醒因等待满足特定条件而睡眠的线程。如果能够做到这一点，线程代码将是非常高效的，并且不会占用宝贵的互斥锁。这正是 POSIX 条件变量能做的事。

什么是条件变量呢？条件变量是一种机制，它允许线程等待某些事件的发生。几个线程可以等待同一个条件变量直到其他线程激活该条件变量为止，这类似于发送一个通知。这时，可以是一个线程被唤醒以响应这个事件，也可以是所有等待条件变量的线程。

注意，条件变量本身并不提供“锁”，因此互斥锁常伴随条件变量的使用，提供相应的锁以安全地访问条件变量。

## 2. 条件变量的初始化和撤销

在使用条件变量前，必须对其进行初始化。初始化条件变量可采用静态或动态初始化方法。

对于静态初始化方法，需要声明一个 `pthread_cond_t` 变量，并把它赋值为常数 `PTHREAD_COND_INITIALIZER`：

```
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
```

对于动态创建的条件变量，应当使用 `pthread_cond_init()` 函数，其函数原型如下：

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
```

`cond`：指向一个条件变量。

`attr`：一个可选的 `pthread_condattr_t` 指针。这个结构可用来设置各种条件变量属性。但是通常并不需要这些属性，所以通常做法是把它指定为 `NULL`。

返回值：成功时都返回 0，否则返回错误码。

一旦使用 `pthread_cond_init()` 函数初始化了条件变量，当不用它时，就应使用 `pthread_cond_destroy()` 撤销它。`pthread_cond_destroy()` 函数的原型如下：

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

`cond`：指向一个将被撤销的条件变量。

返回值：成功时都返回 0，否则返回错误码。

`pthread_cond_destroy()` 函数释放创建条件变量时分配给它的任何资源，但不会释放用来存储 `pthread_cond_t` 的内存。另外，如果还有线程等待该条件变量，不要撤销它，否则会返回错误码 `EBUSY`。

如果有多个线程等待同一个条件变量，唤醒线程的次序由系统的调度策略决定。同时，被唤醒的线程将拥有其调用 `pthread_cond_wait()` 或 `pthread_cond_timedwait()` 函数时所使用的互斥锁。如果多个线程被唤醒，谁拥有互斥锁由系统的调度策略决定。

### 3. 发送信号和广播

如果线程更改某些共享数据后，要唤醒所有正在等待的线程，则应调用 `pthread_cond_broadcast()` 函数。该函数原型如下：

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
```

`cond`：指向一个将被激活的条件变量。

返回值：成功时都返回 0，否则返回错误码。

在某些情况下，活动线程只需要唤醒一个正在睡眠的线程。假设只对队列添加了一个工作作业，那么只需要唤醒其中一个线程。`pthread_cond_signal()` 函数就只唤醒一个线程。其函数原型如下：

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

`cond`：指向一个将被激活的条件变量。

返回值：成功时都返回 0，否则返回错误码。

`pthread_cond_signal()` 和 `pthread_cond_broadcast()` 函数可以由任一个线程调用，而不管其是否拥有相应的互斥锁。

🔔 注意：条件变量应是已被初始化的，否则调用 `pthread_cond_signal()` 或 `pthread_cond_broadcast()` 函数会返回错误码 `EINVAL`。

### 4. 等待条件变量

一旦初始化了互斥锁和条件变量，就可以等待某个条件。`pthread_cond_wait()` 和 `pthread_cond_timedwait()` 函数实现等待条件变量的功能。它们对拥有的互斥锁进行解锁，然后挂起线程直至该条件变量被激活。当线程被唤醒后，它们对互斥锁进行加锁并返回。`pthread_cond_wait()` 函数原型如下：

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

`cond`：指向条件变量。

`mutex`：指向互斥锁。

返回值：成功时返回 0，否则返回错误码。

与 `pthread_cond_wait()` 函数不同，`pthread_cond_timedwait()` 函数指定一个等待的时间，如果超时，即使等待的条件变量还未被激活，它也将返回。其函数原形如下：

```
#include <pthread.h>
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

`cond`：指向条件变量。

`mutex`：指向互斥锁。

`abstime`：指向一个绝对时间。

返回值：成功时返回 0，否则返回错误码。如果超时，则返回 ETIMEDOUT。

🔊 注意：一个互斥锁可以用于许多条件变量，但每个条件变量只能有一个互斥锁。调用 `pthread_cond_wait()` 和 `pthread_cond_timedwait()` 函数之前应把互斥锁锁定。其典型用法如下。

```
1 ...
2 /* first, lock the mutex */
3 int rc = pthread_mutex_lock(&a_mutex);
4 if (rc) { /* an error has occurred */
5     perror("pthread_mutex_lock");
6     pthread_exit(NULL);
7 }
8 rc = pthread_cond_wait(&a_cond, &a_mutex);

/* we were awakened due to the cond. variable being signaled */
9 if (rc == 0) {
10     /* The mutex is now locked again by pthread_cond_wait() */
11     ...
12 }
13 /* finally, unlock the mutex */
14 pthread_mutex_unlock(&a_mutex);
```

代码说明如下。

2~7：加锁操作。

8：线程被挂起直到条件变量（`a_cond`）被激活。

9~12：条件变量（`a_cond`）被激活后的操作。

10~14：解锁操作。

#### 5. 条件变量实例（程序 8.2）

以下实例是一个多线程的无连接并发服务器。它分为服务器和客户两部分。完成的功能如下。

- 服务器在特定的套接字地址上监听，循环接收客户发来的信息，并显示客户 IP 地址及请求信息，同时显示处理客户请求的线程号。
- 如果服务收到的客户信息为“quit”，则退出循环，并关闭套接字。
- 客户向服务器发信息，然后等待服务器回应。一旦接收到服务发来的信息，则显示该信息，并关闭套接字。

服务器程序采用多线程方式，主线程接收客户请求，并将客户请求插入“请求队列”，而其他 3 个线程则从“请求队列”中取出相应请求，并发信息回相应客户。为实现线程间的同步，采用条件变量的方法。当主线程将客户请求插入“请求队列”时，利用条件变量发信号给其他线程，唤醒其中一个来处理客户请求。

客户程序是第 5.3.3 节中的 UDP 客户程序。该程序通过命令行参数输入服务器 IP 地址和发给服务器的信息。

## 6. 源程序

服务器源程序如下：

```
1  #include <stdio.h>          /* standard I/O routines */
2  #include <pthread.h>        /* pthread functions and data structures */
3  #include <stdlib.h>         /* These are the usual header files */
4  #include <strings.h>        /* for bzero() */
5  #include <unistd.h>         /* for close() */
6  #include <sys/types.h>
7  #include <sys/socket.h>
8  #include <netinet/in.h>
9  #include <arpa/inet.h>

/* number of threads used to service requests */
10 #define NUM_HANDLER_THREADS 3
11 #define PORT 1234          /* Port that will be opened */
12 #define MAXDATASIZE 100    /* Max number of bytes of data */

13 pthread_mutex_t request_mutex = PTHREAD_MUTEX_INITIALIZER;
14 pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER;
15 pthread_cond_t got_request = PTHREAD_COND_INITIALIZER;
16 int num_requests = 0; /* number of pending requests, initially none */

17 int sockfd; /* socket descriptors */
18 struct sockaddr_in server; /* server's address information */
19 struct sockaddr_in client; /* client's address information */
20 int sin_size;

21 /* format of a single request. */
22 struct request {
23     char info[MAXDATASIZE];          /* client's data */
24     struct request* next; /* pointer to next request, NULL if none. */
25 };
26 struct request* requests = NULL; /* head of linked list of requests. */
27 struct request* last_request = NULL; /* pointer to last request. */

28 void add_request(char* info, pthread_mutex_t* p_mutex,
                  pthread_cond_t* p_cond_var)
29 {
30     int rc; /* return code of pthreads functions. */
31     struct request* a_request; /* pointer to newly added request. */

32     /* create structure with new request */
33     a_request = (struct request*)malloc(sizeof(struct request));
34     if (!a_request) { /* malloc failed? */
```

```
35     fprintf(stderr, "add_request: out of memory\n");
36     exit(1);
37 }
38 memcpy(a_request->info, info, MAXDATASIZE);
39 a_request->next = NULL;

40 /* lock the mutex, to assure exclusive access to the list */
41 rc = pthread_mutex_lock(p_mutex);

42 /* add new request to the end of the list, updating list */
43 /* pointers as required */
44 if (num_requests == 0) { /* special case - list is empty */
45     requests = a_request;
46     last_request = a_request;
47 }
48 else {
49     last_request->next = a_request;
50     last_request = a_request;
51 }

52 /* increase total number of pending requests by one. */
53 num_requests++;

54 /* unlock mutex */
55 rc = pthread_mutex_unlock(p_mutex);

56 /* signal the condition variable - there's a new request to handle */
57 rc = pthread_cond_signal(p_cond_var);
58 }

59 struct request* get_request(pthread_mutex_t* p_mutex)
60 {
61     int rc;                                /* return code of pthreads functions. */
62     struct request* a_request;             /* pointer to request. */

63     /* lock the mutex, to assure exclusive access to the list */
64     rc = pthread_mutex_lock(p_mutex);

65     if (num_requests > 0) {
66         a_request = requests;
67         requests = a_request->next;
68         if (requests == NULL) { /* this was the last request on the list */
69             last_request = NULL;
70         }
71     }
72     /* decrease the total number of pending requests */
```

```
72     num_requests--;
73     }
74     else { /* requests list is empty */
75         a_request = NULL;
76     }

77     /* unlock mutex */
78     rc = pthread_mutex_unlock(p_mutex);

79     /* return the request to the caller. */
80     return a_request;
81 }

82 void handle_request(struct request* a_request, int thread_id)
83 {
84     char msg[MAXDATASIZE+40];

85     if (a_request) {
86         printf("Thread '%d' handled request '%s'\n",
87             thread_id, a_request->info);
87         fflush(stdout);
88         sprintf(msg, "Thread '%d' handled your request '%s'\n",
89             thread_id, a_request->info);
89         endto(sockfd, msg, strlen(msg), 0,
90             (struct sockaddr *)&client, sin_size);
90     }
91 }

92 void* handle_requests_loop(void* data)
93 {
94     int rc; /* return code of pthreads functions. */
95     struct request* a_request; /* pointer to a request. */
96     int thread_id = *((int*)data); /* thread identifying number */

97     /* lock the mutex, to access the requests list exclusively. */
98     rc = pthread_mutex_lock(&request_mutex);

99     while (1) {
100     if (num_requests > 0) { /* a request is pending */
101         a_request = get_request(&list_mutex);
102         if (a_request) { /* got a request - handle it and free it */
103             rc = pthread_mutex_unlock(&list_mutex);
104             free(a_request);
105         }
106     }
```



```
107 else {
108     rc = pthread_cond_wait(&got_request, &request_mutex);
109 }
110 }
111}

112 int main(int argc, char* argv[])
113 {

114 int thr_id[NUM_HANDLER_THREADS];    /* thread IDs */
115 pthread_t p_threads[NUM_HANDLER_THREADS]; /* thread's structures */
116 int num;
117 char msg[MAXDATASIZE];

118 /* create the request-handling threads */
119 for (int i=0; i<NUM_HANDLER_THREADS; i++) {
120     thr_id[i] = i;
121     pthread_create(&p_threads[i], NULL, handle_requests_loop,
122                   (void*)&thr_id[i]);
123 }

124 /* Create UDP socket */
125 if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
126 {
127     /* handle exception */
128     perror("Creating socket failed.");
129     exit(1);
130 }

131 int opt = SO_REUSEADDR;
132 setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

133 bzero(&server, sizeof(server));
134 server.sin_family=AF_INET;
135 server.sin_port=htons(PORT);
136 server.sin_addr.s_addr = htonl (INADDR_ANY);
137 if (bind(sockfd, (struct sockaddr *)&server,
138         sizeof(struct sockaddr)) == -1) {
139     /* handle exception */
140     perror("Bind error.");
141     exit(1);
142 }

143 sin_size=sizeof(struct sockaddr_in);
144 while (1)
```

```
143 {
144     num = recvfrom(sockfd,msg,MAXDATASIZE,0,
                    (struct sockaddr *)&client,&sin_size);
145     if (num < 0){
146         perror("recvfrom error\n");
147         exit(1);
148     }

149     msg[num] = '\0';
150     printf("You got a message (%s) from %s\n",
            msg,inet_ntoa(client.sin_addr) ); /* prints client's IP */

151     add_request(msg, &list_mutex, &got_request);

152     if (!strcmp(msg,"quit")) break;
153 }

154 close(sockfd); /* close listenfd */
155 return 0;
156 }
```

## 7. 程序分析

10：定义进行处理客户请求的线程数量。

13：初始化并定义用于保护条件变量的互斥锁。

14：初始化并定义用于保证“请求队列”完整性的互斥锁。

15：定义队列长度，初始化为0，表示无客户请求。

21~27：定义并初始化“请求队列”。

28~51：定义 add\_request()函数，用于向队列中添加一个客户请求。

32~39：创建一个“客户请求”结构

55：解锁互斥锁。

57：发信号给等待的线程。

59~81：定义 get\_request()函数，该函数用于从“请求队列”中取出“客户请求”。利用互斥锁保护“关键代码”，从而保证“请求队列”的数据完整性。

82~91：定义 handle\_request()函数，该函数用于显示当前线程的 ID 及被处理的客户请求，并且将该信息发回相应客户。

92~111：定义 handle\_requests\_loop()函数，该函数是处理客户请求的线程所执行的函数。首先锁定用于保护条件变量的互斥锁（request\_mutex），然后，循环检测“请求队列”的长度是否大于0，如果大于0，则直接处理客户请求。否则，等待条件变量被激活。注意，一定要检测队列长度，否则会造成某些客户请求未被处理。例如，如果所有处理客户的线程都忙。此时，又有新的请求被加入队列。由于没有线程处于等待状态，所以没有一个线程被唤醒来处理该请求。

119~122：产生新线程，用于处理客户请求。

123~129：创建 UDP 套接字

132~140：绑定套接字

142~153：反复接收客户发来的请求，一旦接收则显示客户 IP 地址及请求信息，并调用 `add_request()` 添加客户请求到“请求队列”。如果客户请求为“quit”，则退出。

## 8. 运行程序

下面在同一主机上运行服务器程序和客户端程序。首先启动服务器程序（`thrsyncserv`）。然后运行客户端程序（`udpclient`）5 次。服务器运行结果如下：

```
$ thrsyncserv
You got a message (request1) from 127.0.0.1
Thread '0' handled request 'request1'
You got a message (request2) from 127.0.0.1
Thread '1' handled request 'request2'
You got a message (request3) from 127.0.0.1
Thread '2' handled request 'request3'
You got a message (request4) from 127.0.0.1
Thread '0' handled request 'request4'
You got a message (quit) from 127.0.0.1
$
```

## 9. 客户端运行结果

```
$ udpclient 127.0.0.1 request1
Server Message: Thread '0' handled your request 'request1'

$ udpclient 127.0.0.1 request2
Server Message: Thread '1' handled your request 'request2'

$ udpclient 127.0.0.1 request3
Server Message: Thread '2' handled your request 'request3'

$ udpclient 127.0.0.1 request4
Server Message: Thread '0' handled your request 'request4'

$ udpclient 127.0.0.1 quit
^C$
```

## 10. 运行结果说明

在最后一次运行客户端程序时，该程序无法退出，可以用 `CTRL+C` 强行退出。这是由于线程退出时的不同步造成的。在下一节中，将介绍线程退出时的同步问题。

### 8.1.5 同步线程退出

在上一节的实例中，服务器收到客户的“quit”请求后，主线程返回，从而整个进程退

出。但此时，处理客户请求的线程还未完成客户请求的处理，就随着进程的退出而终止。而客户端正等待服务器的信息，处于等待状态。为解决上述问题，主线程应等待处理线程完成客户请求的处理后再退出。

利用 `pthread_join()` 函数完成线程退出的同步。该函数在第 6.3.2 节中已进行了描述。

为保证线程退出时的同步，对上一节实例中的服务器程序进行修改：主线程退出前，首先利用条件变量通知所有客户处理线程。客户处理线程被唤醒后，检测“请求队列”的长度是否为 0 及是否主线程将要退出。如果是，则终止线程。主线程利用 `pthread_join()` 函数等待所有客户处理线程终止，然后再退出主线程。

### 1. 源程序

修改的服务器服务源代码如下：

```
1  #include <stdio.h>          /* standard I/O routines */
2  #include <pthread.h>        /* pthread functions and data structures */
3  #include <stdlib.h>         /* These are the usual header files */
4  #include <strings.h>        /* for bzero() */
5  #include <unistd.h>         /* for close() */
6  #include <sys/types.h>
7  #include <sys/socket.h>
8  #include <netinet/in.h>
9  #include <arpa/inet.h>
10 /* number of threads used to service requests */
11 #define NUM_HANDLER_THREADS 3
12 #define PORT 1234           /* Port that will be opened */
13 #define MAXDATASIZE 100     /* Max number of bytes of data */
14 pthread_mutex_t request_mutex = PTHREAD_MUTEX_INITIALIZER;
15 pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER;
16 pthread_cond_t got_request = PTHREAD_COND_INITIALIZER;
17 int quit;                   /* 1 means all thread will quit */
18 int num_requests = 0; /* number of pending requests, initially none */
19 int sockfd; /* socket descriptors */
20 struct sockaddr_in server; /* server's address information */
21 struct sockaddr_in client; /* client's address information */
22 int sin_size;
23 /* format of a single request. */
24 struct request {
25     char info[MAXDATASIZE]; /* client' data */
26     struct request* next; /* pointer to next request, NULL if none.*/
27 };
28 struct request* requests = NULL; /* head of linked list of requests. */
29 struct request* last_request = NULL; /* pointer to last request. */
```

```
30 void add_request(char* info, pthread_mutex_t* p_mutex,
    pthread_cond_t* p_cond_var)
31 {
32     int rc; /* return code of pthreads functions. */
33     struct request* a_request; /* pointer to newly added request. */

34     /* create structure with new request */
35     a_request = (struct request*)malloc(sizeof(struct request));
36     if (!a_request) { /* malloc failed?? */
37         fprintf(stderr, "add_request: out of memory\n");
38         exit(1);
39     }
40     memcpy(a_request->info, info, MAXDATASIZE);
41     a_request->next = NULL;
42     /* lock the mutex, to assure exclusive access to the list */
43     rc = pthread_mutex_lock(p_mutex);

44     /* add new request to the end of the list, updating list */
45     /* pointers as required */
46     if (num_requests == 0) { /* special case - list is empty */
47         requests = a_request;
48         last_request = a_request;
49     }
50     else {
51         last_request->next = a_request;
52         last_request = a_request;
53     }

54     /* increase total number of pending requests by one. */
55     num_requests++;

56     /* unlock mutex */
57     rc = pthread_mutex_unlock(p_mutex);

58     /* signal the condition variable - there's a new request to handle */
59     rc = pthread_cond_signal(p_cond_var);
60 }

61 struct request* get_request(pthread_mutex_t* p_mutex)
62 {
63     int rc; /* return code of pthreads functions. */
64     struct request* a_request; /* pointer to request. */

65     /* lock the mutex, to assure exclusive access to the list */
```

```
66 rc = pthread_mutex_lock(p_mutex);

67 if (num_requests > 0) {
68     a_request = requests;
69     requests = a_request->next;
70     if (requests == NULL) { /* this was the last request on the list */
71         last_request = NULL;
72     }
73     /* decrease the total number of pending requests */
74     num_requests--;
75 }
76 else { /* requests list is empty */
77     a_request = NULL;
78 }

79 /* unlock mutex */
80 rc = pthread_mutex_unlock(p_mutex);

81 /* return the request to the caller. */
82 return a_request;
83 }

84 void handle_request(struct request* a_request, int thread_id)
85 {
86     char msg[MAXDATASIZE+40];

87     if (a_request) {
88         printf("Thread '%d' handled request '%s'\n",
89             thread_id, a_request->info);
89         fflush(stdout);
90         sprintf(msg, "Thread '%d' handled your request '%s'\n", thread_id,
91             a_request->info);
91         sendto(sockfd, msg, strlen(msg), 0,
92             (struct sockaddr *)&client, sin_size);
92     }
93 }

94 void* handle_requests_loop(void* data)
95 {
96     int rc; /* return code of pthreads functions. */
97     struct request* a_request; /* pointer to a request. */
98     int thread_id = *((int*)data); /* thread identifying number */

99     /* lock the mutex, to access the requests list exclusively. */
100    rc = pthread_mutex_lock(&request_mutex);
```

```
101 while (1) {
102     if (num_requests > 0) { /* a request is pending */
103         a_request = get_request(&list_mutex);
104         /* got a request */
105         if (a_request) {
106             /*handle request */
107             handle_request(a_request, thread_id);
108             /* free it */
109             free(a_request);
110         }
111     }
112     else {
113         if (quit) {
114             // unlock mutex before exit
115             pthread_mutex_unlock(&request_mutex);
116             pthread_exit(NULL);
117         }
118         rc = pthread_cond_wait(&got_request, &request_mutex);
119     }
120 }

121 int main(int argc, char* argv[])
122 {

123     thr_id[NUM_HANDLER_THREADS]; /* thread IDs */
124     pthread_t p_threads[NUM_HANDLER_THREADS]; /* thread's structures */

125     int num;
126     char msg[MAXDATASIZE];

127     quit = 0;
128     /* create the request-handling threads */
129     for (int i=0; i<NUM_HANDLER_THREADS; i++) {
130         thr_id[i] = i;
131         pthread_create(&p_threads[i], NULL,
132             handle_requests_loop, (void*)&thr_id[i]);
133     }

134     /* Create UDP socket */
135     if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
136     {
137         /* handle exception */
138         perror("Creating socket failed.");
139     }
```

```
138     exit(1);
139 }

140 int opt = SO_REUSEADDR;
141 setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

142 bzero(&server, sizeof(server));
143 server.sin_family=AF_INET;
144 server.sin_port=htons(PORT);
145 server.sin_addr.s_addr = htonl (INADDR_ANY);
146 if (bind(sockfd, (struct sockaddr *)&server,
           sizeof(struct sockaddr)) == -1) {
147     /* handle exception */
148     perror("Bind error.");
149     exit(1);
150 }

151 sin_size=sizeof(struct sockaddr_in);
152 while (1)
153 {

154     num = recvfrom(sockfd, msg, MAXDATASIZE, 0,
                    (struct sockaddr *)&client, &sin_size);
155     if (num < 0) {
156         perror("recvfrom error\n");
157         exit(1);
158     }

159     msg[num] = '\0';
160     printf("You got a message (%s) from %s\n",
            msg, inet_ntoa(client.sin_addr) );
161     add_request(msg, &list_mutex, &got_request);
162     if (!strcmp(msg, "quit")) {
163         /* notify our threads we're done . */
164         int rc;
165         rc = pthread_mutex_lock(&request_mutex);
166         quit = 1;
167         rc = pthread_cond_broadcast(&got_request);
168         rc = pthread_mutex_unlock(&request_mutex);

169         /* wait until other thread quit */
170         for (int i=0; i<NUM_HANDLER_THREADS; i++) {
171             pthread_join(p_threads[i], NULL);
172         }
173         break;
```



```

174     }
175 }
176 close(sockfd); /* close listenfd */
177 return 0;
178 }

```

## 2. 程序分析

11：定义进行处理客户请求的线程数量。

14：初始化并定义用于保护条件变量的互斥锁。

15：初始化并定义用于保证“请求队列”完整性的互斥锁。

16：初始化并定义条件变量。

17：定义 quit 变量，其值为 1 表示主线程将退出。

18：定义队列长度，初始化为 0，表示无客户请求。

23~29：定义并初始化“请求队列”。

30~53：定义 add\_request()函数，用于向队列中添加一个客户请求。

57：解锁互斥锁。

59：发信号给等待的线程。

61~83：定义 get\_request()函数，该函数用于从“请求队列”中取出“客户请求”。利用互斥锁保护“关键代码”，从而保证“请求队列”的数据完整性。

84~93：定义 handle\_request()函数，该函数用于显示当前线程的 ID 及被处理的客户请求，并且将该信息发回相应客户。

94~120：定义 handle\_requests\_loop()函数，该函数是处理客户请求的线程所执行的函数。

113~116：如果“请求队列”为空，并且 quit 值为 1，把互斥锁解锁，然后终止线程。

127：将 quit 值初始化为 0，表示主线程正在运行。

129~132：产生新线程，用于处理客户请求。

133~139：创建 UDP 套接字。

140~150：绑定套接字。

164~168：首先锁定互斥锁，将 quit 设为 1，然后通过条件变量通知所有客户处理线程，之后再互斥锁解锁。

170~172：等待所有客户处理线程退出。

## 3. 运行程序（程序 8.3）

同样在同一主机上运行服务器程序和客户程序。首先启动服务器程序（thrsyncserv1）。然后运行客户程序（udpclient）5 次。服务器运行结果如下：

```

$ thrsyncserv
You got a message (request1) from 127.0.0.1
Thread '0' handled request 'request1'
You got a message (request2) from 127.0.0.1
Thread '1' handled request 'request2'
You got a message (request3) from 127.0.0.1

```

```
Thread '2' handled request 'request3'
You got a message (request4) from 127.0.0.1
Thread '0' handled request 'request4'
You got a message (quit) from 127.0.0.1
$
```

#### 4. 客户程序运行结果

```
$ udpclient 127.0.0.1 request1
Server Message: Thread '0' handled your request 'request1'

$ udpclient 127.0.0.1 request2
Server Message: Thread '1' handled your request 'request2'

$ udpclient 127.0.0.1 request3
Server Message: Thread '2' handled your request 'request3'

$ udpclient 127.0.0.1 request4
Server Message: Thread '0' handled your request 'request4'

$ udpclient 127.0.0.1 quit
Server Message: Thread '1' handled your request 'quit'

$
```

#### 5. 运行结果说明

从运行结果可以看出，最后一次的客户程序的运行完全正常。

### 8.1.6 死锁

死锁是一系列线程竞争一系列资源产生的永久阻塞。导致死锁的最常见的错误是自死锁（self deadlock）：一个线程在拥有一个锁的情况下试图再次获得该锁。例如，在受某一互斥锁保护的“关键代码”中，如果对该锁再次进行锁定操作，将导致死锁。解决这种死锁的办法是：避免在“关键代码”中调用其他对该锁进行操作的函数。

死锁的另外一种情况是，线程 1 和线程 2 分别等待对方为自己的运行提供条件。例如，线程 1 和线程 2 分别获得互斥锁 A 和互斥锁 B。这时线程 1 想获得互斥锁 B，而同时线程 2 想获得互斥锁 A。结果，线程 1 阻塞等待 B，而线程 2 阻塞等待 A，从而造成死锁。

#### 1. 源程序

下面通过例子予以说明。在上一节的实例中，定义了两个互斥锁（request\_mutex 和 list\_mutex）分别用于保护条件变量及“请求队列”。如果仅用一个互斥锁，情况如何？以下程序把上一节实例修改为采用单个互斥锁，源程序如下：

```
1  #include <stdio.h>          /* standard I/O routines */
2  #include <pthread.h>        /* pthread functions and data structures */
```

```
3  #include <stdlib.h>      /* These are the usual header files */
4  #include <strings.h>     /* for bzero() */
5  #include <unistd.h>      /* for close() */
6  #include <sys/types.h>
7  #include <sys/socket.h>
8  #include <netinet/in.h>
9  #include <arpa/inet.h>

10 /* number of threads used to service requests */
11 #define NUM_HANDLER_THREADS 3
12 #define PORT 1234        /* Port that will be opened */
13 #define MAXDATASIZE 100  /* Max number of bytes of data */

14 // pthread_mutex_t request_mutex = PTHREAD_MUTEX_INITIALIZER;
15 pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER;

16 pthread_cond_t got_request = PTHREAD_COND_INITIALIZER;
17 int quit; /* 1 means all thread will quit */

18 int num_requests = 0; /* number of pending requests, initially none */
19 int sockfd; /* socket descriptors */

20 struct sockaddr_in server; /* server's address information */
21 struct sockaddr_in client; /* client's address information */
22 int sin_size;

23 /* format of a single request. */
24 struct request {
25     char info[MAXDATASIZE]; /* client's data */
26     struct request* next; /* pointer to next request, NULL if none. */
27 };
28 struct request* requests = NULL; /* head of linked list of requests. */
29 struct request* last_request = NULL; /* pointer to last request. */

30 void add_request(char* info, pthread_mutex_t* p_mutex,
31                 pthread_cond_t* p_cond_var)
32 {
33     int rc; /* return code of pthreads functions. */
34     struct request* a_request; /* pointer to newly added request. */

35     /* create structure with new request */
36     a_request = (struct request*)malloc(sizeof(struct request));
37     if (!a_request) {
38         fprintf(stderr, "add_request: out of memory\n");
39         exit(1);
40     }
41 }
```

```
39     }
40     memcpy(a_request->info, info, MAXDATASIZE);
41     a_request->next = NULL;

42     /* lock the mutex, to assure exclusive access to the list */
43     printf("Before:add-lock\n");
44     rc = pthread_mutex_lock(p_mutex);
45     printf("After:add-lock\n");
46     /* add new request to the end of the list, updating list */
47     /* pointers as required */
48     if (num_requests == 0) { /* special case - list is empty */
49         requests = a_request;
50         last_request = a_request;
51     }
52     else {
53         last_request->next = a_request;
54         last_request = a_request;
55     }

56     /* increase total number of pending requests by one. */
57     num_requests++;

58     /* unlock mutex */
59     rc = pthread_mutex_unlock(p_mutex);

60     /* signal the condition variable - there's a new request to handle */
61     rc = pthread_cond_signal(p_cond_var);
62 }

63 struct request* get_request(pthread_mutex_t* p_mutex)
64 {
65     int rc;                                /* return code of pthreads functions. */
66     struct request* a_request;             /* pointer to request. */

67     /* lock the mutex, to assure exclusive access to the list */
68     printf("Before:get-lock(%s)\n", requests->info);
69     rc = pthread_mutex_lock(p_mutex);
70     printf("After:get-lock(%s)\n", requests->info);
71     if (num_requests > 0) {
72         a_request = requests;
73         requests = a_request->next;
74         if (requests == NULL) { /* this was the last request on the list */
75             last_request = NULL;
76         }
77         /* decrease the total number of pending requests */
```

```
78     num_requests--;
79     }
80     else { /* requests list is empty */
81         a_request = NULL;
82     }

83     /* unlock mutex */
84     rc = pthread_mutex_unlock(p_mutex);

85     /* return the request to the caller. */
86     return a_request;
87 }

88 void handle_request(struct request* a_request, int thread_id)
89 {
90     char msg[MAXDATASIZE+40];

91     if (a_request) {
92         printf("Thread '%d' handled request '%s'\n",
93             thread_id, a_request->info);
94         fflush(stdout);
95         sprintf(msg, "Thread '%d' handled your request '%s'\n",
96             thread_id, a_request->info);
97         sendto(sockfd, msg, strlen(msg), 0,
98             (struct sockaddr *)&client, sin_size);
99     }

100 void* handle_requests_loop(void* data)
101 {
102     int rc;          /* return code of pthreads functions. */
103     struct request* a_request; /* pointer to a request. */
104     int thread_id = *(int*)data; /* thread identifying number */

105     /* lock the mutex, to access the requests list exclusively. */
106     printf("Before: Thread[%d] locks\n", thread_id);
107     rc = pthread_mutex_lock(&list_mutex);
108     printf("After: Thread[%d] locks\n", thread_id);

109     while (1) {
110         if (num_requests > 0) { /* a request is pending */
111             a_request = get_request(&list_mutex);
112             if (a_request) { /* got a request - handle it and free it */
113                 handle_request(a_request, thread_id);
114                 free(a_request);
115             }
116         }
117     }
118 }
```

```
113     }
114     }
115     else {
116         if (quit) {
117             // unlock mutex before exit
118             pthread_mutex_unlock(&list_mutex);
119             pthread_exit(NULL);
120         }
121         rc = pthread_cond_wait(&got_request, &list_mutex);
122     }
123 }

124 int main(int argc, char* argv[])
125 {

126     int      thr_id[NUM_HANDLER_THREADS]; /* thread IDs */
127     pthread_t p_threads[NUM_HANDLER_THREADS];
128     int num;
129     char msg[MAXDATASIZE];

130     quit = 0;
131     /* create the request-handling threads */
132     for (int i=0; i<NUM_HANDLER_THREADS; i++) {
133         thr_id[i] = i;
134         pthread_create(&p_threads[i], NULL, handle_requests_loop,
135                       (void*)&thr_id[i]);
136     }

137     /* create TCP socket */
138     if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
139     {
140         /* handle exception */
141         perror("Creating socket failed.");
142         exit(1);
143     }

144     int opt = SO_REUSEADDR;
145     setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

146     bzero(&server, sizeof(server));
147     server.sin_family=AF_INET;
148     server.sin_port=htons(PORT);
149     server.sin_addr.s_addr = htonl (INADDR_ANY);
```

```
149 if (bind(sockfd, (struct sockaddr *)&server,
        sizeof(struct sockaddr)) == -1) {
150     /* handle exception */
151     perror("Bind error.");
152     exit(1);
153 }

154 sin_size=sizeof(struct sockaddr_in);
155 while (1)
156 {
157     num = recvfrom(sockfd,msg,MAXDATASIZE,0,
                    (struct sockaddr *)&client,&sin_size);
158 if (num < 0){
159     perror("recvfrom error\n");
160     exit(1);
161 }

162 msg[num] = '\0';
163 printf("You got a message (%s) from %s\n",
        msg,inet_ntoa(client.sin_addr) ); /* prints client's IP */

164 add_request(msg, &list_mutex, &got_request);

165 if (!strcmp(msg,"quit")) {
166     /* notify our threads we're done . */
167     int rc;
168     rc = pthread_mutex_lock(&list_mutex);
169     quit = 1;
170     rc = pthread_cond_broadcast(&got_request);
171     rc = pthread_mutex_unlock(&list_mutex);

172     /* wait until other thread quit */
173     for (int i=0; i<NUM_HANDLER_THREADS; i++) {
174         pthread_join(p_threads[i], NULL);
175     }
176 break;
177 }
178 }
179 close(sockfd); /* close listenfd */
180 return 0;
181 }
```

## 2. 程序分析

14：注释掉互斥锁（request\_mutex）的定义，仅用单个互斥锁。

43~45：在函数 `add_request()` 加锁前显示 “ Before:add-lock\n ”，加锁后显示 “ After:add-lock\n ” (表示加锁成功，线程未阻塞)。

68~70：在函数 `get_request()` 加锁前显示 “ Before:get-lock(%s)\n ”，加锁后显示 “ After:get-lock(%s)\n ” 表示加锁成功，线程未阻塞)。

104~106：在函数 `handle_requests_loop()` 加锁前显示 “ Before: Thread[%d] locks\n ”，锁后显示 “ After: Thread[%d] locks\n ” (表示加锁成功，线程未阻塞)。

### 3. 运行程序 (程序 8.4)

同样,在同一主机上运行服务器程序和客户程序。首先启动服务器程序( `thrsyncserv2` )。然后运行客户程序 ( `udpclient` )。

### 4. 服务器运行结果

```
$ thrsyncserv2
Before: Thread[0] locks
After: Thread[0] locks
Before: Thread[1] locks
Before: Thread[2] locks
You got a message (request) from 127.0.0.1
Before:add-lock
^C$
```

### 5. 客户程序运行结果

```
$ udpclient 127.0.0.1 request
^C$
```

### 6. 运行结果说明

从运行结果可以看出，服务器处于死锁状态。图 8.1 是与互斥锁操作相关的函数的简略流程图 (仅保留与互斥锁操作相关的部分)。下面借助该图进行分析。服务器程序运行后，首先线程 0 加锁互斥锁 ( `list_mutex` )，并加锁成功。此时 `list_mutex` 处于锁定状态，因此线程 1、2 对其加锁时被挂起直到线程 0 对其解锁。同样，主线程接收到客户请求后，调用函数 `add_request()`，该函数首先加锁 `list_mutex` 而造成主线程被挂起。由于主线程被挂起，“请求队列”的长度和 `quit` 始终为 0，线程 0 处于“死循环”状态，无法对 `list_mutex` 解锁。

如果首先主线程加锁互斥锁 ( `list_mutex` )，其他 3 个线程被挂起直到主线程完成将客户请求加入“请求队列”并解锁。此时，某个线程被唤醒，完成锁定 `list_mutex`，然后调用函数 `get_request()`，函数 `get_request()` 也对 `list_mutex` 进行加锁，从而造成同一线程对同一互斥锁进行两次加锁 (即自死锁)。

以上仅分析了两种情况。由于多线程运行的不确定性，情况异常复杂。所以在处理死



锁时要十分注意。

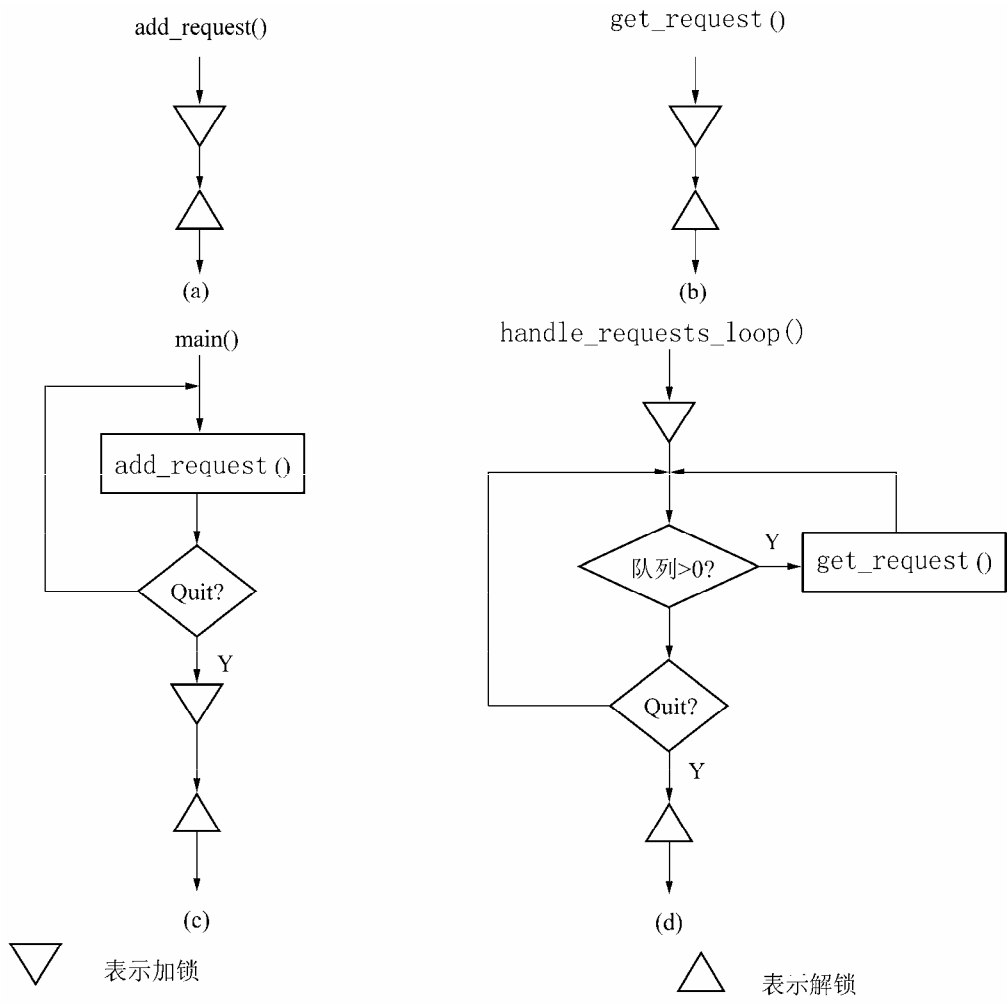


图 8.1 与互斥锁操作相关的函数的简略流程图

## 8.2 进 程 同 步

### 8.2.1 进程关系

#### 1. 父子进程关系

进程之间具有关系。首先，每个进程有一个父进程。父进程通过调用 `fork()` 函数产生子进程。当子进程终止时，父进程会得到通知并能取得子进程退出状态。

## 2. 进程组

每个进程除了有一进程 ID 之外，还属于一个进程组。一个进程组是一个或多个进程的集合。它分为前台进程组和后台进程组。每个进程组有一个惟一的进程组 ID。进程组 ID 类似于进程 ID，是一个正整数，并可存放在 pid\_t 数据类型中。函数 getpgrp() 返回调用进程的进程组 ID。函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp(void);
```

返回值：调用进程的进程组 ID。

每个进程组有一个组长进程。组长进程是，进程组 ID 就等于它的进程 ID。进程组组长可以创建一个进程组，创建该组中的进程，然后终止。只要在某个进程组中有一个进程存在，则该进程组就存在，这与其组长进程是否终止无关。从进程组创建开始到其中最后一个进程离开为止的时间段称为进程组的生命期。某个进程组中的最后一个进程可以终止，或者加入到另一个进程组。

一个进程调用 setpgid() 函数可以加入到一个现存的组或者创建一个新进程组。

```
#include<sys/types.h>
#include<unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

pid：进程 ID。

pgid：进程组 ID。

返回值：若成功为 0，出错为-1。

将 pid 进程的进程组 ID 设置为 pgid。如果这两个参数相等，则由 pid 指定的进程变成进程组组长。进程只能将自身和其子进程设置为进程组 ID。某个子进程调用 exec() 函数之后，就不能再将该子进程的 ID 作为进程组 ID。

如果 pgid 是 0，则把由 pid 指定的进程 ID 作为进程组 ID。

## 3. 会话期

一个会话期是一个或多个进程组的集合，但只能有一个前台进程组。一次登录就形成一个会话。

一个进程调用 setsid() 函数就可建立一个新会话期。Setsid() 函数原型如下：

```
#include<sys/types.h>
#include<unistd.h>
pid_t setsid(void);
```

返回值：若成功为进程组 ID，出错为-1。

只有调用进程不是进程组的组长进程，该函数才能建立新的会话。调用 setsid() 函数后，进程成为新会话的组长进程和新进程组的组长进程，同时进程失去控制终端。

## 4. 控制终端

会话的组长进程打开一个终端之后，该终端就成为该会话的控制终端。与控制终端建

立连接的组长进程称为控制进程。

一个会话只能有一个控制终端，产生在控制终端上的输入和信号将发送给会话的前台进程组中的所有进程。

### 8.2.2 信号处理

#### 1. 信号的概念

信号是软件中断，是异步事件。信号有自己的名称和对应的编号。这些名字都以三个字符 SIG 开头。编号为正整数。POSIX 定义的信号如下。

| 信号      | 动作  | 注释  |
|---------|-----|---|
| SIGHUP  | A   | Hangup detected on controlling terminal or death of controlling process |
| SIGINT  | A   | Interrupt from keyboard   |
| SIGQUIT | A   | Quit from keyboard  |
| SIGILL  | A   | Illegal Instruction   |
| SIGABRT | C   | Abort signal from abort(3)  |
| SIGFPE  | C   | Floating point exception  |
| SIGKILL | AEF | Kill signal   |
| SIGSEGV | C   | Invalid memory reference  |
| SIGPIPE | A   | Broken pipe: write to pipe with no readers                              |
| SIGALRM | A   | Timer signal from alarm(2)  |
| SIGTERM | A   | Termination signal  |
| SIGUSR1 | A   | User-defined signal 1   |
| SIGUSR2 | A   | User-defined signal 2   |
| SIGCHLD | B   | Child stopped or terminated   |
| SIGCONT |     | Continue if stopped   |
| SIGSTOP | DEF | Stop process  |
| SIGTSTP | D   | Stop typed at tty   |
| SIGTTIN | D   | tty input for background process  |
| SIGTTOU | D   | tty output for background process                                       |

其中：

- A 表示默认动作是终止进程；
- B 表示默认动作是忽略信号；
- C 表示默认动作是系统内核转储；
- D 表示默认动作是停止进程；
- E 表示信号不能被捕获；
- F 表示信号不能被忽略。

很多条件可以产生一个信号，例如：

- 用户在按下特定的键之后，将向该终端上的前台进程组发送信号。比如，Ctrl+C。
- 硬件异常会产生信号。比如被 0 除，无效内存引用等。

- kill(2) 函数可允许进程向其他进程或进程组发送任意信号。
- kill(1) 函数允许用户向进程发送任意信号。
- 软件设置的条件，比如 SIGALARM。

信号的处理是由系统完成的，进程通过相应的函数调用要求系统在某个信号出现时按照下列三种方式中的一种进行操作。

- 忽略信号。有两个信号永远不能忽略: SIGKILL 和 SIGSTOP，它们为超级用户提供了杀死和停止进程的必要方法。
- 捕获信号。告诉内核在出现信号时调用自己定义的处理函数。比如，可以在处理 SIGCHLD 信号时利用 waitpid()函数获得子进程的退出状态，以避免生成僵尸进程。
- 执行系统默认动作。每个信号有其默认动作。

在信号处理过程中，如果信号处理函数调用了可能会修改 errno 变量的函数，则应该保存并恢复 errno 的值。另外，对某些系统，当进程调用“慢”系统调用时，如果发生信号，系统内核会终止系统调用，以便让进程有机会处理信号。发生中断系统调用时，被中断的系统调用返回错误值，而 errno 被设置为 EINTR。慢系统调用的如下。

- 终端设备、管道和网络设备上的文件读取/写入。
- 某些设备上的文件打开。
- pause()和 wait()系统调用。
- 某些 ioctl 操作。
- 某些进程间通信函数。

## 2. 信号的产生

产生信号的最常用系统函数是 kill()、raise()、alarm()、setitimer()和 abort()函数。

kill()函数向其他进程发送信号。raise()函数向当前进程发送信号。函数原型如下：

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int sig);
int raise (int sig);
```

kill()函数的 pid 参数的四种条件如下。

- pid > 0: 信号发送到进程 ID 为 pid 的进程。
- pid == 0: 信号发送到与发送进程处于同一进程组的进程。
- pid < -1: 信号发送到进程组 ID 等于 -pid 的所有进程。
- pid == -1: POSIX 未指定。

sig 为 0 时，不会发送任何信号，但仍将执行错误检查，因此可用来检查是否有向指定进程发送信号的许可。

raise()函数等价于 kill (getpid (), sig)。

alarm()函数可设置定时器。定时器到期时，将产生 SIGALRM 信号。函数原型如下：

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds);
```

seconds：定时器设置的秒数。

返回值：为尚未过期的定时器所剩余的时间。

setitimer()是另一个设置间隔定时器的函数，它使用更加灵活。函数原型如下：

```
#include <sys/time.h>
int getitimer (int which, struct itimerval *value);
int setitimer (int which, const struct itimerval *value,
               struct itimerval *ovalue);
```

setitimer()函数定时精度要比 alarm()函数高，同时可以按照进程的执行时间（用户态时间，核心态时间，或两者）进行计时。

abort()函数向进程发送 SIGABRT 信号，从而可导致程序非正常终止。函数原型如下：

```
#include <stdlib.h>
void abort (void);
```

如果捕获 SIGABRT 信号，且信号处理器不返回，则 abort()函数不能终止进程。当 abort()函数终止进程时，所有打开的流均被刷新并关闭。如果 SIGABRT 被阻塞或忽略，abort()函数将覆盖这种设置。

### 3. 信号集

每个进程都有一个信号屏蔽字，它规定了当前要阻塞传送到该进程的信号集。对每种可能的信号在该屏蔽字中都有一位与之对应。对于某种信号，其对应位已设置，则它当前是被阻塞的。信号屏蔽要用到下面的几个函数。

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(sigset_t *set, int signo);
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

sigemptyset()函数初始化信号集合 set，将 set 设置为空。Sigfillset()函数也初始化信号集合，但它是将信号集合设置为包含所有信号的集合。Sigaddset()函数将信号 signo 加入到信号集合之中，sigdelset()函数将信号从信号集合中删除。Sigismember()函数查询信号是否在信号集合之中。

Sigprocmask()函数是最为关键的一个函数。在使用之前要先设置好信号集合 set。这个函数的作用是将指定的信号集合 set 加入到进程的信号阻塞集合之中。如果提供了 oset 函数，那么当前的进程信号阻塞集合将会保存在 oset 里面。参数 how 决定函数的如下操作方式。

- SIG\_BLOCK: 增加一个信号集合到当前进程的阻塞集合之中。
- SIG\_UNBLOCK: 从当前的阻塞集合之中删除一个信号集合。
- SIG\_SETMASK: 将当前的信号集合设置为信号阻塞集合。

### 4. 信号的处理

进程通过 sigaction()函数调用要求系统在某个信号出现时所进行的操作。函数原型

如下：

```
#include <signal.h>
int sigaction (int signum, const struct sigaction *act,
               struct sigaction *oldact);
```

signum：信号。

act：指向用于描述相应操作的结构。

oldact：非空时，可返回先前的设置。

Sigaction：用于描述相应的操作：

```
struct sigaction {
    void (*sa_handler) (int);
    void (*sa_sigaction) (int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer) (void);
}
```

sa\_handler：可指定除 SIGKILL 和 SIGSTOP 之外的信号动作，即可为 SIG\_DFL、SIG\_IGN 和用户定义函数。

sa\_mask：指定在处理 signum 信号时，应该阻塞的信号集。

sa\_flags：指定信号处理过程中的行为，其值为下述标志“或”的结果。

- SA\_NOCLDSTOP: 子进程停止时不接收 SIGCHLD。
- SA\_ONESHOT 或 SA\_RESETHAND: 重置信号动作为默认值。
- SA\_RESTART: 如果该信号中断慢系统调用，则重新启动系统调用。
- SA\_NOMASK 或 SA\_NODEFER: 不要避免在信号处理函数中接收同一信号。
- SA\_SIGINFO: 信号处理函数接受三个参数。这时，必须设置 actsa\_sigaction，其中 siginfo\_t 是内核传递到信号处理函数中的发生信号时进程的状态，利用 SA\_SIGINFO 和 siginfo\_t 实现有效的存储管理。

## 5. 其他信号函数

pause()函数暂停调用进程并等待信号，函数原型如下：

```
#include <unistd.h>
int pause (void);
```

sleep()函数使进程休眠指定的秒数，函数原型如下：

```
#include <unistd.h>
unsigned int sleep (unsigned int seconds);
```

经过指定的时间之后，sleep()函数返回。在这之前，如果接收到信号，则 sleep()函数返回剩余的秒数。

### 8.2.3 处理僵死进程

前面提到，如果子进程在父进程之前终止，系统内核为每个终止子进程保存了一定量的信息。所以终止进程的父进程应调用 `wait()` 或 `waitpid()` 函数，以获取被终止子进程的有关信息并释放它仍占用的资源。否则，终止的子进程成为僵死进程。在多线程并发服务器的编程中应格外注意。由于服务器长期运行并服务大量客户，而且每个客户都会产生多次连接，如果不处理僵死进程或处理的不妥当，必将耗尽系统资源而导致系统崩溃。

在多线程并发服务器实例中，我们首先启动服务器程序（`procserver`），然后同时运行客户 1 和客户 2。

结果似乎很正常，但用命令 “`ps -el`” 查看过程的运行情况，就会发现由于服务器未进行子进程的终止处理而导致了僵死进程（ID 为 444 和 447）的产生。“`ps -el`” 命令的运行结果如下：

```
$ ps -el
 F  S   UID   PID  PPID   TTY      TIME    CMD
...
  8  S   1002   441   402    pts/4      0:00    procserv
...
  8  Z   1002   444   441             0:00    <defunct>
  8  Z   1002   447   441             0:00    <defunct>
```

#### 1. 实例（程序 8.5）

为了处理僵死进程，父进程必须调用 `wait()` 或 `waitid()` 函数。以下对多线程并发服务器实例进行修改。由于当一个进程终止时，父进程将收到 `SIGCHLD` 信号，因此我们增加一个信号处理器，每当捕获到该信号时，信号处理便调用 `waitid()` 函数。修改后的服务器源程序如下：

```
1  #include <stdio.h>           /* These are the usual header files */
2  #include <strings.h>         /* for bzero() */
3  #include <unistd.h>          /* for close() */
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>
8  #include <signal.h>
9  #include <wait.h>
10 #include <errno.h>

11 #define PORT 1234           /* Port that will be opened */
12 #define BACKLOG 2           /* Number of allowed connections */
13 #define MAXDATASIZE 1000

14 void process_cli(int connectfd, sockaddr_in client);
```

```
15 void sig_handler(int s);

16 main()
17 {
18     int listenfd, connectfd; /* socket descriptors */
19     pid_t pid;
20     struct sockaddr_in server; /* server's address information */
21     struct sockaddr_in client; /* client's address information */
22     int sin_size;

23     struct sigaction act, oact;
24     act.sa_handler = sig_handler;
25     sigemptyset(&act.sa_mask);
26     act.sa_flags = 0;
27     if (sigaction(SIGCHLD, &act, &oact) < 0) {
28         perror("Sigaction failed!");
29         exit(1);
30     }

31     /* Create TCP socket */
32     if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
33     {
34         /* handle exception */
35         perror("Creating socket failed.");
36         exit(1);
37     }

38     int opt = SO_REUSEADDR;
39     setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

40     bzero(&server, sizeof(server));
41     server.sin_family=AF_INET;
42     server.sin_port=htons(PORT);
43     server.sin_addr.s_addr = htonl (INADDR_ANY);
44     if (bind(listenfd, (struct sockaddr *)&server,
45             sizeof(struct sockaddr)) == -1) {
46         /* handle exception */
47         perror("Bind error.");
48         exit(1);
49     }

49     if(listen(listenfd, BACKLOG) == -1){ /* calls listen() */
50         perror("listen() error\n");
51         exit(1);
52     }
```



```
53 sin_size=sizeof(struct sockaddr_in);
54 while(1)
55 {
56     /* accept connection */
57     if ((connectfd = accept(listenfd,
58         (struct sockaddr *)&client,&sin_size))== -1) {
59         if (errno == EINTR) continue;
60         perror("accept() error\n");
61         exit(1);
62     }
63     /* create child process */
64     if ((pid=fork())>0) {
65         /* parent process */
66         close(connectfd);
67         continue;
68     }
69     else if (pid==0) {
70         /* child process */
71         close(listenfd);
72         process_cli(connectfd, client);
73         exit(0);
74     }
75     else {
76         printf("fork error\n");
77         exit(0);
78     }
79 }
80 close(listenfd); /* close listenfd */

81 void process_cli(int connectfd, struct sockaddr_in client)
82 {
83     int num;
84     char recvbuf[MAXDATASIZE], sendbuf[MAXDATASIZE],
85         cli_name[MAXDATASIZE];

86     printf("You got a connection from %s. ",inet_ntoa(client.sin_addr) );
87     /* Get client's name from client */
88     num = recv(connectfd, cli_name, MAXDATASIZE,0);
89     if (num == 0) {
90         close(connectfd);
91         printf("Client disconnected.\n");
92         return;
93     }
```

```

93  cli_name[num - 1] = '\0';
94  printf("Client's name is %s.\n",cli_name);

95  while (num = recv(connectfd, recvbuf, MAXDATASIZE,0)) {
96      recvbuf[num] = '\0';
97      printf("Received client( %s ) message: %s",cli_name, recvbuf);
98      for (int i = 0; i < num - 1; i++) {
99          sendbuf[i] = recvbuf[num - i - 2];
100     }
101     sendbuf[num - 1] = '\0';
102     send(connectfd,sendbuf,strlen(sendbuf),0);
103 }
104 close(connectfd); /* close connectfd */
105 }

106 void sig_handler(int s)
107 {
108     pid_t    pid;
109     int      stat;

110     while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
111         printf("child %d terminated\n", pid);
112     return;
113 }

```

代码说明如下。

15：声明信号处理函数

23~30：设置信号处理器。

57~61：接收客户连接请求。注意，由于 `accept()` 函数是慢系统调用，在信号产生时会中断其调用并将 `errno` 变量设置为 `EINTR`，此时应重新调用 `accept()` 函数。否则，程序将显示 “`accept() error: Interrupted system call`” 并退出。

106~113：信号处理函数。反复调用 `waitid()` 函数，直到没有要终止的子进程。

## 2. 运行程序

下面运行修改后的实例，首先启动服务器程序（`procserver1`）。然后同时运行客户 1 和客户 2。

## 3. 运行结果

服务器运行结果如下：

```

$ procserver1
You got a connection from 127.0.0.1. Client's name is c1.
Received client( c1 ) message: 1234
child 616 terminated
You got a connection from 127.0.0.1. Client's name is c2.

```

```
Received client( c2 ) message: abcd  
child 618 terminated
```

客户 1 运行结果如下：

```
$ procclient 127.0.0.1  
Connected to server.  
Input name : c1  
Input string to server:1234  
Server Message: 4321  
Input string to server: ^D  
Exit.
```

客户 2 运行结果如下：

```
$ procclient 127.0.0.1  
Connected to server.  
Input name : c2  
Input string to server:abcd  
Server Message: dcba  
Input string to server: ^D  
Exit.
```

用 “ps -el” 查看的结果如下：

```
$ ps -el  
F S  UID  PID  PPID  TTY      TIME    CMD  
...  
8 S  1002   614   402    pts/4    0:00    procserv1  
. . .
```

#### 4. 运行结果说明

从运行结果可以看出，不再有僵死进程存在了。

## 8.3 进程间通信

在多线程并发服务器中，通常要解决的一个问题是：如何控制和处理不同进程之间的通信和数据交换。Unix 系统主要提供了以下几种手段：

- 管道
- FIFO
- 消息队列
- 共享内存
- 信号量
- 套接字

### 8.3.1 管道

管道是最常见的 IPC 机制，是单工的。若两个进程要做双向传输则需要两个管道。管道生成时即有两端，一端为读，一端为写。两个进程要协调好，一个进程从读方读，另一个进程向写方写。并且只能在关系进程之间进行，比如父子进程之间，通过系统调用 `pipe()` 实现。`pipe()` 函数将创建一个管道和两个文件描述符：`fd[0]` 和 `fd[1]`。原型如下：

```
#include <unistd.h>
int pipe(int filedes[2]);
```

`fd[0]`：文件描述符，将用于读操作。

`fd[1]`：文件描述符，将用于写操作。

返回值：成功返回 0，如果创建失败将返回 -1 并将错误码记录在 `errno` 中。

使用 `int pipe(int fd[2])` 创建管道的标准编程模式如下：

- (1) 创建管道；
- (2) 使用 `fork()` 函数创建两个（或多个）相关联的进程；
- (3) 使用 `read()` 和 `write()` 函数操作管道；
- (4) 管道使用完毕后用 `close(int fd)` 函数关闭管道。

### 8.3.2 FIFO

FIFO 又称做命名管道，为系统特殊文件方式。通过 FIFO 的通信可发生在任何两个进程之间，且只需要对 FIFO 有适当的访问权限。对 FIFO 的读写操作与普通文件类似。管道的传输方式为 FIFO 流方式。

命名管道通过 `mkfifo()` 函数创建。`mkfifo()` 函数原型如下：

```
#include <sys/types.h>
int mkfifo(const char *filename, mode_t mode);
```

`filename`：命名管道的文件名。

`mode`：访问权限。

返回值：不成功则返回 NULL，成功则返回管道的文件指针。

以下通过例子予以说明。该例子包括一个服务器和一个客户程序。

#### 1. FIFO 服务器实例（程序 8.6）

服务器产生一个管道，用于接收客户发来的信息。将收到的字符串转换为大写字母，再通过客户创建的管道发回到相应的客户。其源程序如下：

```
1  #include <ctype.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <string.h>
6  #include <fcntl.h>
```

```
7  #include <limits.h>
8  #include <sys/types.h>
9  #include <sys/stat.h>

10 #define SERVER_FIFO_NAME "./serv_fifo"
11 #define CLIENT_FIFO_NAME "./cli_%d_fifo"

12 #define BUFFER_SIZE 20

13 struct data_to_pass_st {
14     pid_t client_pid;
15     char some_data[BUFFER_SIZE - 1];
16 };

17 int main()
18 {
19     int server_fifo_fd, client_fifo_fd;
20     struct data_to_pass_st my_data;
21     int read_res;
22     char client_fifo[256];
23     char *tmp_char_ptr;

24     mkfifo(SERVER_FIFO_NAME, 0777);
25     server_fifo_fd = open(SERVER_FIFO_NAME, O_RDONLY);
26     if (server_fifo_fd == -1) {
27         fprintf(stderr, "Server fifo failure\n");
28         exit(EXIT_FAILURE);
29     }

30     sleep(10); /* lets clients queue for demo purposes */

31     do {
32         read_res = read(server_fifo_fd, &my_data, sizeof(my_data));
33         if (read_res > 0) {
34             tmp_char_ptr = my_data.some_data;
35             while (*tmp_char_ptr) {
36                 *tmp_char_ptr = toupper(*tmp_char_ptr);
37                 tmp_char_ptr++;
38             }
39             sprintf(client_fifo, CLIENT_FIFO_NAME, my_data.client_pid);
40             client_fifo_fd = open(client_fifo, O_WRONLY);
41             if (client_fifo_fd != -1) {
42                 write(client_fifo_fd, &my_data, sizeof(my_data));
43                 close(client_fifo_fd);
44             }

```

```
45     }  
46     } while (read_res > 0);  
47     close(server_fifo_fd);  
48     unlink(SERVER_FIFO_NAME);  
49     exit(EXIT_SUCCESS);  
50 }
```

代码说明如下。

10：定义服务器管道名。

11：定义客户管道名，由客户进程 ID 和固定字符组成。例如，客户进程 ID 为 456，则管道名为“cli-456-fifo”。

13~16：定义 data\_to\_pass\_st 结构，是客户向服务发送信息的格式。

24~29：创建并打开服务器管道，服务器通过此管道读取客户信息。

31~46：反复从服务器管道读入客户信息。首先，将收到的字符串转换成大写字母，然后通过客户 ID 形成客户管道名，并将转换后的信息通过该管道发回客户。

## 2. FIFO 客户实例（程序 8.7）

客户程序首先打开服务器管道，然后创建自己的管道，通过服务器管道发送信息给服务器，并且通过自己创建的管道接收服务器发回的信息。源程序如下：

```
1  #include <ctype.h>  
2  #include <ctype.h>  
3  #include <unistd.h>  
4  #include <stdlib.h>  
5  #include <stdio.h>  
6  #include <string.h>  
7  #include <fcntl.h>  
8  #include <limits.h>  
9  #include <sys/types.h>  
10 #include <sys/stat.h>  
  
11 #define SERVER_FIFO_NAME "./serv_fifo"  
12 #define CLIENT_FIFO_NAME "./cli_%d_fifo"  
  
13 #define BUFFER_SIZE 20  
  
14 struct data_to_pass_st {  
15     pid_t  client_pid;  
16     char   some_data[BUFFER_SIZE - 1];  
17 };  
  
18 int main()  
19 {  
20     int server_fifo_fd, client_fifo_fd;  
21     struct data_to_pass_st my_data;
```

```
22  int times_to_send;
23  char client_fifo[256];

24  server_fifo_fd = open(SERVER_FIFO_NAME, O_WRONLY);
25  if (server_fifo_fd == -1) {
26      fprintf(stderr, "Sorry, no server\n");
27      exit(EXIT_FAILURE);
28  }

29  my_data.client_pid = getpid();
30  sprintf(client_fifo, CLIENT_FIFO_NAME, my_data.client_pid);
31  if (mkfifo(client_fifo, 0777) == -1) {
32      fprintf(stderr, "Sorry, can't make %s\n", client_fifo);
33      exit(EXIT_FAILURE);
34  }

35  for (times_to_send = 0; times_to_send < 5; times_to_send++) {
36      sprintf(my_data.some_data, "Hello from %d", my_data.client_pid);
37      printf("%d sent %s, ", my_data.client_pid, my_data.some_data);
38      write(server_fifo_fd, &my_data, sizeof(my_data));
39      client_fifo_fd = open(client_fifo, O_RDONLY);
40      if (client_fifo_fd != -1) {
41          if (read(client_fifo_fd, &my_data, sizeof(my_data)) > 0) {
42              printf("received: %s\n", my_data.some_data);
43          }
44          close(client_fifo_fd);
45      }
46  }

47  close(server_fifo_fd);
48  unlink(client_fifo);
49  exit(EXIT_SUCCESS);
50 }
```

代码说明如下。

24~28：打开服务器管道。

29~34：创建客户自己的管道。

35~46：通过服务器管道向服务器发送 5 次信息，然后接收从服务器发回的处理后的信息，并显示结果。

### 3. 运行程序

首先启动服务器程序（prg8\_6）。然后运行客户（prg8\_7）。

### 4. 运行结果

服务器运行结果如下：

```
$ prg8_6
```

```
$
```

客户运行结果如下：

```
$ prg8_7
463 sent Hello from 463, received: HELLO FROM 463
463 sent Hello from 463, received: HELLO FROM 463
463 sent Hello from 463, received: HELLO FROM 463
463 sent Hello from 463, received: HELLO FROM 463
463 sent Hello from 463, received: HELLO FROM 463
$
```

## 5. 结果说明

从结果可以看出进程可通过 FIFO 管道进行通信。

### 8.3.3 消息队列

进程间通过往消息队列发送消息或接收消息来完成通信。消息队列具有如下特点。

- 通过消息队列 key 值来定义和生成消息队列。
- 任何进程只要有访问权限并知道 key 即可访问消息队列。
- 消息队列为内存块方式数据段。
- 消息队列中的消息元素长度可为系统参数限制内的任何长度。
- 消息元素由消息类型分类，其访问方式为按类型访问。
- 在一次读写操作前都必须取得消息队列标识符，即访问权。访问后即脱离访问关系。
- 消息队列中的某条消息被读后即从队列中删除。
- 消息队列的访问具备加锁机制处理，即一个进程在访问时另一个进程不能访问。
- 在权限允许时，消息队列的信息传递是双向的。

### 8.3.4 共享内存

共享内存是效率最高的 IPC 机制，它允许任何两个进程访问相同的逻辑内存区，从而实现进程间通信。它具有如下特点。

- 通过共享内存来 key 值定义和生成共享内存。
- 任何进程只要有访问权限并知道 key 即可访问共享内存。
- 共享内存为内存块方式的数据段。
- 共享内存中的数据长度可为系统参数限制内的任何长度。
- 共享内存的访问与数组的访问方式相同。
- 在取得共享内存标识符将共享内存与进程数据段联接后即可开始对之进行读写操作，在所有操作完成之后再做共享内存和进程数据段脱离操作，才完成全部共享内存访问过程。
- 共享内存中的数据不会因数据被进程读取后消失。
- 共享内存的访问不具备锁机制处理，即多个进程可能同时访问同一个共享内存的



同一个数据单元。因此，共享内存的使用最好和信号量一起操作,以具备锁机制,保证数据的一致。

- 在权限允许时，共享内存的信息传递是双向的。

### 8.3.5 信号量

信号量实际上是一种同步机制，主要用途是保护临界资源(在一个时刻只被一个进程所拥有)，通常与共享内存一起使用。信号量由系统内核提供，它可在进程间共享数据，可执行原子操作。

下面介绍有关的系统调用。

#### 1. semget()函数

获取信号量集合的标识符。

原型：

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

```
int semget(key_t key, int num_sems, int sem_flags);
```

描述：

key：信号量集合的键。

num\_sems：信号量集合中元素个数。

sem\_flags：任选参数。

返回值：

返回信号量集合标识符，若出错则返回-1。

#### 2. semop()函数

信号量操作(P/V)。

原型：

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

```
int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);
```

描述：

sem\_id：信号量集合标识符。

sem\_ops：信号量操作结构的指针。

num\_sem\_ops：信号量操作结构的个数。

返回值：

运算完成前该组信号量中最后一个被运算的信号量的值，若出错则返回-1。

#### 3. semctl()函数

控制信号量，用于信号量的初始化、撤销等操作。

原型：

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

```
int semctl(int sem_id, int sem_num, int command,...);
```

描述：

sem\_id ：信号量集合标识符。

sem\_num ：信号量元素编号。

command ：控制命令。

...：命令参数列表。

返回值：

根据命令返回相应的值，若出错则返回-1。

## 8.4 小 结

开发并发服务器应十分注意同步问题。对于复杂的网络服务器，同步可能会异常复杂，处理不好就会导致系统的可靠性下降。

尤其是多线程并发服务器，多个线程使用同一地址空间。虽然，线程间通信简单，但由于共享数据，则必须处理同步。在 Unix 系统中，提供互斥锁和条件变量来实现线程的同步。互斥锁用于保护共享数据的完整性，条件变量则用于协调多个线程的运行以提高运行效率。同时，线程退出的同步亦不能忽视，通常利用 pthread\_join()函数来实现。

对于多进程并发服务器，多个进程的运行主要由信号机制来协调，另外，利用函数 wait()或 waitid()来处理进程退出的同步，以避免僵死进程的产生。

在处理同步时，要注意防止死锁。并且在长时间的操作（例如 I/O）上尽量不要加锁，这样会对性能造成负影响。

由于各个进程采用不同的地址空间，需要专门的通信机制，以提供进程间通信。Unix 系统提供管道、FIFO、消息队列、共享内存、信号量、套接字等方法，应根据实际应用来选择。

## 第9章 异常处理

异常处理是网络编程中最复杂的部分。本章通过介绍异常产生的主要原因及处理的方法，并且对超时异常、服务器异常、客户异常的处理进行详细分析，从而帮助读者能够编写可靠的网络程序。

### 9.1 异常处理基础

在实际的网络开发中，对异常的处理占有相当大的部分。这是由网络的特点决定的。在网络上，各主机系统的运行是独立的，甚至同一网络应用的服务器和客户也是由不同开发商开发的。网络本身也非常复杂，它包括多种路由、信道、防火墙等等。网络的使用情况也随时变化，造成网络拥挤程度的不同。由于这些因素，经常造成通信中断、IP 包传输延迟过长或丢失包等异常情况。如果处理不好，会直接影响系统的性能。这些异常，在编程中，通常引起一些 I/O 操作的超时。

另一类异常是由本地系统引起的，通常是由于系统资源分配造成的。如 `socket()` 和 `bind()` 函数的调用异常。这些异常影响系统的稳定性，处理不好，可能导致系统的崩溃。

### 9.2 函数调用的错误处理

在网络编程中，应检查每个系统调用的错误返回，尤其是与 I/O 相关的调用。

对于 Unix 系统，大部分系统调用在非正常返回时，其返回值为 -1，并设置全局变量 `errno`。如 `socket()`、`bind()`、`accept()`、`listen()` 函数等。

变量 `errno` 存放一个正整数来表明上一个系统调用的错误值。仅当系统调用发生错误时才设置它。如果系统调用正常返回，它的值是不确定的。因此，当一个系统调用发生错误时应立即检查 `errno` 的值，以避免下一个调用修改了 `errno` 的值。

对于线程而言，每个线程都有专用的 `errno` 变量，它不是一个共享的变量，因此不必考虑多线程同步问题。

错误值定义在头文件 `<sys/errno.h>` 中，为常量。通常用函数 `perror()` 来显示错误信息。其原型如下：

```
#include <stdio.h>
void perror(const char *s);
```

s：指向一字符串

如果 `s` 为 `NULL`，则 `perror()` 函数直接将相应错误信息显示到标准错误输出上。如果 `s` 不为 `NULL`，则首先显示 `s` 所指向的字符串，再显示 “:”，然后显示 `errno` 所代表的错误信息。

### 9.2.1 显示错误信息

下面这个小程序用来显示所有的错误信息。源程序如下：

```
1  #include <errno.h>
2  #include <stdio.h>

3  main()
4  {
5      for (int i =1; i < 150; i++) {
6          errno = i;
7          perror("error is ");
8      }
9  }
```

代码说明如下。

6：设置 `errno` 的值

7：调用 `perror()` 函数显示相应错误信息。

运行该程序的结果如下：

```
error is : Not owner
error is : No such file or directory
error is : No such process
error is : Interrupted system call
error is : I/O error
error is : No such device or address
error is : Arg list too long
error is : Exec format error
error is : Bad file number
error is : No child processes
error is : Resource temporarily unavailable
error is : Not enough space
error is : Permission denied
error is : Bad address
error is : Block device required
error is : Device busy
error is : File exists
error is : Cross-device link
error is : No such device
error is : Not a directory
error is : Is a directory
error is : Invalid argument
```

```
error is : File table overflow
error is : Too many open files
error is : Inappropriate ioctl for device
error is : Text file busy
error is : File too large
error is : No space left on device
error is : Illegal seek
error is : Read-only file system
error is : Too many links
error is : Broken pipe
error is : Argument out of domain
error is : Result too large
error is : No message of desired type
error is : Identifier removed
error is : Channel number out of range
error is : Level 2 not synchronized
error is : Level 3 halted
error is : Level 3 reset
error is : Link number out of range
error is : Protocol driver not attached
error is : No CSI structure available
error is : Level 2 halted
error is : Deadlock situation detected/avoided
error is : No record locks available
error is : Operation canceled
error is : Operation not supported
error is : Disc quota exceeded
error is : Bad exchange descriptor
error is : Bad request descriptor
error is : Message tables full
error is : Anode table overflow
error is : Bad request code
error is : Invalid slot
error is : File locking deadlock
error is : Bad font file format
error is : Owner of the lock died
error is : Lock is not recoverable
error is : Not a stream device
error is : No data available
error is : Timer expired
error is : Out of stream resources
error is : Machine is not on the network
error is : Package not installed
error is : Object is remote
error is : Link has been severed
```

```
error is : Advertise error
error is : Srmount error
error is : Communication error on send
error is : Protocol error
error is : Locked lock was unmapped
error is : Facility is not active
error is : Multihop attempted
error is : Error 75
error is : Error 76
error is : Not a data message
error is : File name too long
error is : Value too large for defined data type
error is : Name not unique on network
error is : File descriptor in bad state
error is : Remote address changed
error is : Can not access a needed shared library
error is : Accessing a corrupted shared library
error is : .lib section in a.out corrupted
error is : Attempting to link in more shared libraries than system limit
error is : Can not exec a shared library directly
error is : Illegal byte sequence
error is : Operation not applicable
error is : Number of symbolic links encountered during path name traversal
        exceeds MAXSYMLINKS
error is : Error 91
error is : Error 92
error is : Directory not empty
error is : Too many users
error is : Socket operation on non-socket
error is : Destination address required
error is : Message too long
error is : Protocol wrong type for socket
error is : Option not supported by protocol
error is : Error 100
error is : Error 101
error is : Error 102
error is : Error 103
error is : Error 104
error is : Error 105
error is : Error 106
error is : Error 107
error is : Error 108
error is : Error 109
error is : Error 110
error is : Error 111
```

```
error is : Error 112
error is : Error 113
error is : Error 114
error is : Error 115
error is : Error 116
error is : Error 117
error is : Error 118
error is : Error 119
error is : Protocol not supported
error is : Socket type not supported
error is : Operation not supported on transport endpoint
error is : Protocol family not supported
error is : Address family not supported by protocol family
error is : Address already in use
error is : Cannot assign requested address
error is : Network is down
error is : Network is unreachable
error is : Network dropped connection because of reset
error is : Software caused connection abort
error is : Connection reset by peer
error is : No buffer space available
error is : Transport endpoint is already connected
error is : Transport endpoint is not connected
error is : Structure needs cleaning
error is : Error 136
error is : Not a name file
error is : Not available
error is : Is a name file
error is : Remote I/O error
error is : Reserved for future use
error is : Error 142
error is : Cannot send after socket shutdown
error is : Too many references: cannot splice
error is : Connection timed out
error is : Connection refused
error is : Host is down
error is : No route to host
error is : Operation already in progress
```

### 9.2.2 定义错误处理函数

对于错误的处理往往会导致程序结构的复杂化，通常我们将定义一个新的函数，该函数不仅包含了相应的功能调用，也包括了错误的处理。这样就可以提高程序的可读性，并且把错误处理独立出来便于修改和调度。例子如下。

```
1 int myaccept(int listenfd, struct sockaddr* client, int* sin_size ) {
2 int connectfd;
3 while ((connectfd = accept(listenfd, client, sin_size))!=-1) {
4     if (errno == EINTR) continue;
5     /* handle exception */
6     ...
7     perror("accept() error\n");
8     break;
9 }
10 return connectfd;
11 }
```

代码说明如下。

3~4: 错误码 EINTR, 表示系统中断。因为 accept()函数是慢系统调用, 当信号产生时, 系统会中断该调用。这并不是一个真正的错误, 通常的作法是重新执行该函数。

5~8: 错误处理, 例如显示错误信息, 关闭相应资源, 设置标志等等。

## 9.3 I/O 超时处理

前面提到由网络引起的异常, 主要表现为 I/O 操作超时。目前, 主要处理的方法有:

- 使用 alarm()函数
- 使用 select()函数

### 9.3.1 使用 alarm()函数

该函数将产生定时信号, 然后通过信号处理器来处理超时。这种方法较为简单, 却不够灵活。典型代码如下:

```
1 static int sTimeout = 0;

2 static void AlarmHandler(int sig)
3 {
4     sTimeout = 1;
5 }

6 .
7 .
8 .
9     signal(SIGALRM, AlarmHandler);
10     sTimeout = 0;
11     alarm(CONNECT_TIMEOUT);

12     if ( connect(sock, (struct sockaddr *) &server, sizeof(server)) == -1)
```



```
13  {
14      if ( sTimeout )
15          perror("timeout connecting stream socket");
16      else
17          perror("connecting failed");
18          close(sock);
19          exit(1);
20  }

21  sTimeout = 0;
22  alarm(0);
23  .
24  .
25  .
```

代码说明如下。

2~5：超时处理函数。将超时状态置为 1。

9：设置报警信号的处理函数为 AlarmHandler()。

10：超时状态置为 0。

11：设置连接超时的时间值。

12：连接服务器。

14~15：处理连接超时。

17~19：处理连接错误。

21：超时状态置为 0。

22：取消连接超时设置。

### 9.3.2 使用 select 函数

select()函数有一个时间选项，当 I/O 操作超时后，函数返回 0。典型代码如下：

```
1  ...
2  int      fd;
3  fd_set  fdRSet, fdWSet;
4  struct  timeval timeout;
5  int maxfd = MAXDESCRIPTOR + 1;
6  ...
7  timeout.tv_sec = ...;
8  timeout.tv_usec = 0;
9  FD_ZERO(&fdRSet);
10 FD_SET(fd, &fdRSet);
11 FD_ZERO(&fdWSet);
12 FD_SET(fd, &fdWSet);
13 switch (select(maxfd, &fdRSet, &fdWSet, &timeout)) {
14     case -1:  handle exception;
15     case 0:   handle timeout;
```

```
16         default:  I/O;
17     }
18 ...
```

代码说明如下。

5：设置 maxfd 为打开的最大描述符加 1。

7~8：设置超时值。

9~10：将读集合置为 0。

11~12：将写集合置为 0。

13：调用 select()函数。

14：处理异常。

15：处理超时。

16：读写操作。

## 9.4 服务器异常处理

无论是网络还是本地系统引起的异常，操作系统都将错误信息通过系统调用的返回值提供给应用程序，因此，异常处理的关键是对返回的错误码进行处理。

### 9.4.1 异常处理的系统调用

对于 TCP 服务器，主要涉及的系统调用包括：socket()、bind()、listen()、accept()、send() 和 recv()函数。以下分别讨论对错误的处理。

#### 1. socket()函数错误处理

socket()函数调用失败，则返回-1，并在 errno 变量中设置如下的错误码。

- EMFILE：描述符表已满。通常的处理方法是提示用户，并退出运行。
- ENOMEM：没有足够的用户内存。通常的处理方法是提示用户，并退出运行。

#### 2. bind()函数错误处理

bind()函数调用失败返回-1，并在 errno 变量中设置如下的错误码。

- EADDRINUSE：地址已被使用。通常的处理方法是选用其他套接字地址或提示用户，并退出运行。

#### 3. listen()函数错误处理

listen()函数调用失败是由程序的逻辑引起的，而不是系统或网络异常造成的。如果调用失败，则显示错误并退出运行。

#### 4. accept()函数错误处理

accept()函数调用失败返回-1，并在 errno 变量中设置如下的错误码。

- EINTR：由于信号的传递而引起系统中断该调用。accept()函数是慢系统调用。通常的处理方法是重新执行该函数调用。

- EMFILE：描述符表已满。通常的处理方法是提示用户，并退出运行。
- ENOMEM：没有足够的用户内存。通常的处理方法是提示用户，并退出运行。

#### 5. recv()函数错误处理

recv()函数调用失败返回-1，并在 errno 变量中设置如下的错误码。

- EINTR：由于信号的传递而引起系统中断该调用。通常的处理方法是重新执行该函数调用。
- EIO：I/O 错误。通常的处理方法是重新执行函数调用。
- ENOMEM：没有足够的用户内存。通常的处理方法是提示用户，并退出运行。

#### 6. send()函数错误处理

send()函数调用失败返回-1，并在 errno 变量中设置如下的错误码。

- EINTR：由于信号的传递而引起系统中断该调用。通常的处理方法是重新执行该函数调用。
- EMSGSIZE：发送的消息太长，通常的处理方法是将消息分段再进行发送。
- ENOMEM：没有足够的用户内存。通常处理方法是提示用户，并退出运行。

从以上的处理方法可以看出，错误处理分为两类：一类是需要终止程序运行，另一类是需要重要调用该函数。另外，还有一类情况需要考虑，当服务器和客户建立连接后，由于网络故障或客户异常而导致服务器长时间收不到客户发来的信息，此时 recv()函数处于阻塞状态，为了减少不必要的系统开销，服务器必须对这种超时进行处理。当发生超时后，服务器应终止连接，释放相应资源以及对客户信息的终止处理。

### 9.4.2 服务器异常处理实例

以下通过实例予以说明。该实例是一个多进程并发服务器，它对 socket()函数的调用异常采用终止程序的方法处理。对于 accept() 和 send()函数的调用异常，如果是 EINTR 错误则重新调用该函数，否则终止程序。对于 recv()函数的调用异常，如果是 EINTR 错误则重新调用该函数，如果是超时错误则终止连接，并显示超时信息并结束该进程。

#### 1. 源程序（程序 9.1）

```
1  #include <stdio.h>           /* These are the usual header files */
2  #include <strings.h>         /* for bzero() */
3  #include <unistd.h>          /* for close() */
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>
8  #include <signal.h>
9  #include <wait.h>
10 #include <errno.h>

11 #define PORT 1234           /* Port that will be opened */
12 #define BACKLOG 2           /* Number of allowed connections */
```

```
13 #define MAXDATASIZE 1000

14 void process_cli(int connectfd, sockaddr_in client),
        void sig_handler(int s);
15 int myrecv(int, char*, int,int), mysend(int,char*,int,int);

16 main()
17 {
18     int listenfd, connectfd; /* socket descriptors */
19     pid_t pid;
20     struct sockaddr_in server; /* server's address information */
21     struct sockaddr_in client; /* client's address information */
22     int sin_size;

23     struct sigaction act, oact;
24     act.sa_handler = sig_handler;
25     sigemptyset(&act.sa_mask);
26     act.sa_flags = 0;
27     if (sigaction(SIGCHLD, &act, &oact) < 0) {
28         perror("Sigaction failed!");
29         exit(1);
30     }

31     /* create TCP socket */
32     if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
33     {
34         /* handle exception */
35         perror("Creating socket failed.");
36         exit(1);
37     }

38     int opt = SO_REUSEADDR;
39     setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

40     bzero(&server,sizeof(server));
41     server.sin_family=AF_INET;
42     server.sin_port=htons(PORT);
43     server.sin_addr.s_addr = htonl (INADDR_ANY);
44     if (bind(listenfd, (struct sockaddr *)&server,
        sizeof(struct sockaddr)) == -1) {
45         /* handle exception */
46         perror("Bind error.");
47         exit(1);
48     }
```

```
49  if(listen(listenfd,BACKLOG) == -1){ /* calls listen() */
50      perror("listen() error\n");
51      exit(1);
52  }

53  sin_size=sizeof(struct sockaddr_in);
54  while(1)
55  {
56      /* Accept connection */
57      if ((connectfd = accept(listenfd,
58          (struct sockaddr *)&client,&sin_size))== -1) {
59          if (errno == EINTR) continue;
60          perror("accept() error\n");
61          exit(1);
62      }
63      /* create child */
64      if ((pid=fork())>0) {
65          /* parent process */
66          close(connectfd);
67          continue;
68      }
69      else if (pid==0) {
70          /* child process */
71          close(listenfd);
72          process_cli(connectfd, client);
73          exit(0);
74      }
75      else {
76          printf("fork error\n");
77          exit(0);
78      }
79  }
80  close(listenfd); /* close listenfd */

81  void process_cli(int connectfd, struct sockaddr_in client)
82  {
83      int num;
84      char recvbuf[MAXDATASIZE], sendbuf[MAXDATASIZE],
85          li_name[MAXDATASIZE];

86      printf("You got a connection from %s. ",inet_ntoa(client.sin_addr) );
87      /* Get client's name from client */
88      num = myrecv(connectfd, cli_name, MAXDATASIZE,0);
89      if (num == 0) {
```

```
89     close(connectfd);
90     printf("Client disconnected.\n");
91     return;
92 }
93 cli_name[num - 1] = '\0';
94 printf("Client's name is %s.\n",cli_name);

95 while (num = myrecv(connectfd, recvbuf, MAXDATASIZE,0)) {
96     recvbuf[num] = '\0';
97     printf("Received client( %s ) message: %s",cli_name, recvbuf);
98     for (int i = 0; i < num - 1; i++) {
99         sendbuf[i] = recvbuf[num - i - 2];
100     }
101     sendbuf[num - 1] = '\0';
102     mysend(connectfd,sendbuf,strlen(sendbuf),0);
103 }
104 close(connectfd); /* close connectfd */
105 }

106 void sig_handler(int s)
107 {
108     pid_t    pid;
109     int      stat;

110     while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
111         printf("child %d terminated\n", pid);
112     return;
113 }

114 #define TIMEOUT 120
115 int myrecv(int connfd, char* buf, int len ,int flag)
116 {
117     int num;
118     fd_set fdRSet;
119     struct timeval timeout;
120     int maxfd = connfd + 1;

121     timeout.tv_sec = TIMEOUT;
122     timeout.tv_usec = 0;
123     FD_ZERO(&fdRSet);
124     FD_SET(connfd, &fdRSet);
125     switch (select(maxfd, &fdRSet,NULL, &timeout)) {
126         case -1: perror();
127                 close(connfd);
128                 exit(1);
```

```
129     case 0: printf("Receiving timeout.\n");
130             close(connfd);
131             exit(1);
132     default: while ((num = recv(connfd, buf, len, flag)) == -1) {
133             if (errno == EINTR) continue;
134             perror();
135             close(connfd);
136             exit(1);
137         }
138     break;
139 }
140 return num;
141 }

142 int mysend(int connfd, char* buf, int len, int flag)
143 {
144     int num;
145     while ((num = send(connfd, buf, len, flag)) == -1) {
146         if (errno == EINTR) continue;
147         perror();
148         close(connfd);
149         exit(1);
150     }
151     return num;
152 }
```

## 2. 程序分析

15: 声明新的接收和发送函数(myrecv()和 mysend)，它们具有异常处理能力。

33~37: socket()函数错误处理。

45~48: bind()函数错误处理。

50~51: listen()函数错误处理。

58~60: 如果错误是 EINTR，则重新调用该函数。

87: 调用具有异常处理能力的 myrecv()函数。

95: 调用具有异常处理能力的 myrecv()函数。

102: 调用具有异常处理能力的 mysend()函数。

114: 定义接收超时为 120 秒。

121~131: 超时处理。如果超时，则显示错误，关闭连接并退出该子进程。

132~136: 接收客户数据。如果错误是 EINTR，则重新接收。否则，显示错误，关闭连接并退出该子进程。

145~146: 如果发送错误是 EINTR，则重新发送。否则，显示错误，关闭连接并退出该子进程。

## 9.5 客户异常处理

与服务器相同, 客户异常处理的关键是对返回的错误码进行处理。对于 TCP 客户, 主要涉及的系统调用包括: `socket()`、`connect()`、`send()`和 `recv()`。其中只有 `connect()`函数与服务器不同。`connect()`调用失败, 则返回-1, 并在 `errno` 变量中设置如下的错误码。

- `EADDRNOTAVAIL`: 远程地址无效。通常处理方法是选择新的地址, 并重新连接或提示用户, 关闭套接字并终止程序。
- `ECONNREFUSED`: 连接被拒绝。当客户调用 `connect()`函数, 初始一个 TCP 连接请求, 而此时服务器端没有进程等待在相应的端口上(例如, 服务器还未启动), 远程系统发回一个连接复位信号(`RST`), 这时 `connect()`函数将返回 `ECONNREFUSED` 错误码。通常的处理方法是等待一段时间后, 仍无法连接则退出。
- `EINTR`: 由于信号的传递而引起系统中断该调用。通常的处理方法是重新执行该函数调用。
- `ENETUNREACH`: 网络无法抵达。由于路由器故障, 导致网络无法返回 ICMP 的响应包, 此时系统将进行多次尝试直到超时。通常处理方法是提示用户, 并退出运行。
- `ENXIO`: 服务器在建立连接成功之前退出。通常处理方法是提示用户, 并退出运行。
- `ETIMEDOUT`: 连接超时。当客户发出连接请求信号(`SYN`)后, 服务器在指定时间段内(75 秒)没有响应, 则产生连接超时错误。通常处理方法是提示用户, 并退出运行。
- `ENOMEM`: 没有足够的用户内存。通常处理方法是提示用户, 并退出运行。

## 9.6 小 结

在网络编程中, 异常处理是非常重要的。由于网络的复杂性, 异常出现的频率较高, 处理不好, 不仅影响系统的性能, 有时也会导致系统的崩溃。

无论是服务器还是客户, 处理异常的关键是处理与网络 I/O 操作相关的系统调用。通常有如下三类处理方法。

- 对于致命错误, 需要终止程序。
  - 对于一般性错误, 处理完异常后, 重新执行调用。
  - 对于超时异常, 根据网络应用情况进行处理。
- I/O 操作超时的处理主要用 `alarm()`或 `select()`函数来实现。



## 第 10 章 创建实用套接字类库

在网络应用开发中，随着需求的改变及性能的提高，程序变得越来越复杂，越来越大。从而导致编译和连接时间长。为更有效地重用代码，应该建立相应的类库。库是包括不同对象的文件。它可作为一个实体进行连接，因而可以极大地提高连接速度。Unix 系统可以创建两种库：静态链接库和动态链接库。

本章讲述创建 Socket 类库的方法，并且通过实例进行说明。该实例采用面向对象方法封装了 TCP Socket 的主要功能，使读者极易实现基于 TCP 的网络程序并且易于扩充和移植。通过该技术，读者可建立自己的 Socket 类库，可极大地提高网络软件开发的效率，而且可开发出具有商业价值的类库。

### 10.1 创建静态链接库

#### 10.1.1 创建库文件

静态链接库包括一些编译后的文件，它们在链接时连接到程序中，而与运行无关。所以，在运行程序时，不需要提供相应的库文件。

在 Unix 中，系统提供的 ar 命令可用于创建静态链接库，例如：

```
ar rc libmyutil.a object1.o object2.o object3.o
```

此命令创建一个名为 libmyutil.a 的库文件，其中包括 3 个已编译的文件(object1.o、object2.o 和 object3.o)。如果库文件已创建，则将这 3 个文件加入到库中，如果有重复则替换相应文件。

#### 10.1.2 建立库文件索引

创建库文件后，需要为库文件建立索引才能被编译器所用。创建和修改索引的命令是 ranlib，例如：

```
ranlib libmyutil.a
```

此命令为库文件 libmyutil.a 建立索引。

#### 10.1.3 连接库文件

在连接时只需用 -l 参数，将库文件连接到可执行文件中，例如：

```
cc main.o -lmyutil -o programname
```

此命令将目标文件（main.o）、库文件（libmyutil.a）连接到可执行文件 programname 中。使用-l 参数时，前缀 lib 与后缀.a 由编译程序自动加上。参数-o 后跟随可执行文件名。

## 10.2 创建动态链接库

动态链接库又称共享库。在连接时，链接器只是检查程序所需要的符号，并不将代码连接到可执行的文件中。只有在程序运行时，由系统将相应的库导入到内存中，并将所需代码与程序代码相连接。这样，多个程序可共享相同的库，从而有效地节省内存空间。

### 10.2.1 创建库文件

为了创建动态链接库，首先要产生与位置无关的代码，例如：

```
cc -fPIC -c object1.c
```

此命令将产生与位置无关代码的目标文件（object1.o）。

然后，再创建库文件，例如：

```
cc -shared libmyutil.so object1.o object2.o object3.o
```

此命令创建动态链接库 libmyutil.so。其中-shared 表示创建动态链接库，libmyutil.so 是库名。object1.o、object2.o 和 object3.o 是库中所包含的目标文件。

### 10.2.2 使用动态链接库

动态链接库的使用包括两个步骤。

（1）编译时，链接器需要检查动态链接库，确保其包括生成可执行文件所需的所有符号，这时并不将代码插入到执行文件中。

（2）在程序运行时，需要告诉系统的动态加载程序动态链接库所在的位置。系统的动态加载程序是一个系统进程，负责自动装载和连接动态链接库到执行进程中。

编译时，动态链接库的用法与静态链接库的方法相同，例如：

```
cc main.o -lmyutil -o programname
```

此命令将目标文件（main.o）、库文件（libmyutil.so）连接到可执行文件 programname 中。使用-l 参数时，前缀 lib 与后缀.so 由编译程序自动加上。参数-o 后跟随可执行文件名。链接器将自动查找动态链接库文件 libmyutil.so。

在运行时，系统的动态加载程序将在系统指定的目录下查找相应的动态链接库，例如目录 /lib、/usr/lib 和 /usr/X11/lib 等。如果创建的动态链接库不在系统目录下，必须设置 LD\_LIBRARY\_PATH 环境变量，可以用命令 ldd 来检查系统是否能正确定位程序所需的动态链接库。例如：

```
ldd programname
```

此命令将列出程序 `programname` 所需的动态链接库的名称及其所在的目录。如果系统找不到相应的库，则显示 `library not found`。

另外，在进行连接要注意次序。链接器在连接时依次检查每个文件。如果是对象文件（.o 文件）则将其插入到执行文件中。如果是库文件，则仅检查前面的文件中未定义的符号是否在该文件中，如果在该库文件中，则将包含相应符号的对象文件插入到执行文件中。这意味着，如果后面的文件中未定义的符号在此库文件中，由于包含相应符号的对象文件并未插入到执行文件中并且该库文件不会再被处理，因此链接器将无法找到相应的符号而产生连接错误。

### 10.2.3 相互引用的库文件

有一种情况需要特别注意，如库文件 A 和库文件 B 相互引用对方定义的符号，此时有一个库必须使用两次。例如：

```
cc main.o -lA -lB -lA
```

当然这种方法的连接效率低，最好是重新设计库文件，从而避免相互引用。

### 10.2.4 动态库与静态库并存

如果系统中同时存在静态库和动态库，链接器首先查找动态库，例如：

```
cc main.o -lmyutil -o programname
```

链接器首先查找文件 `libmyutil.so`，如果找不到该文件，则找文件 `libmyutil.a`。因此在使用静态库时，应避免使用同名的动态链接库。

## 10.3 创建自定义的套接字类库

### 10.3.1 设计套接字类库

为了有效地重用代码，我们采用面向对象的编程方法来实现套接字类库。

由于 Unix 系统提供的网络 API 均是 C 库形式，因此，第一次必须将 C 形式的系统调用封装到相应的类中，即 `Wrapper` 类。在此基础上建立一些共用类。然后，可以根据实际应用的情况，创建一些应用类。另外，还可以创建一些基类用于类的管理或特殊的用途。

#### 1. `Wrapper` 类

根据以上思想，我们建立套接字的 `Wrapper` 类，主要包括以下一些类。

- `MySocket` 类：封装基础的套接字系统调用，并且实现异常处理。为了满足不同的需要，提供了多种形式的函数调用。
- `MyThread` 类：封装了 POSIX 线程的系统调用以实现多线程，并且提供了简单明了的方法来产生线程。
- `MyMutex` 类：提供实现互斥锁的基本方法。

- MyCondition 类：提供实现条件变量的基本方法。

## 2. 共用类

共用类主要包括以下类。

- TcpServThr 类：提供实现基于 TCP 多线程服务器的方法。
- TcpCliThr 类：提供实现基于 TCP 多线程客户的方法。
- MessageQue 类：提供实现带互斥锁的队列的方法。

## 3. 应用类

应用类包括如下。

- ChatServer 类：一个简单的聊天室服务器类，是一个 TCP 多线程并发服务器。
- ChatClient 类：一个聊天室客户类，是一个基于 TCP 多线程客户。

另外，还实现了一个基类 Thread\_interface，用于更灵活地产生线程。

## 4. 各类的关系

它们的关系图见图 10.1。

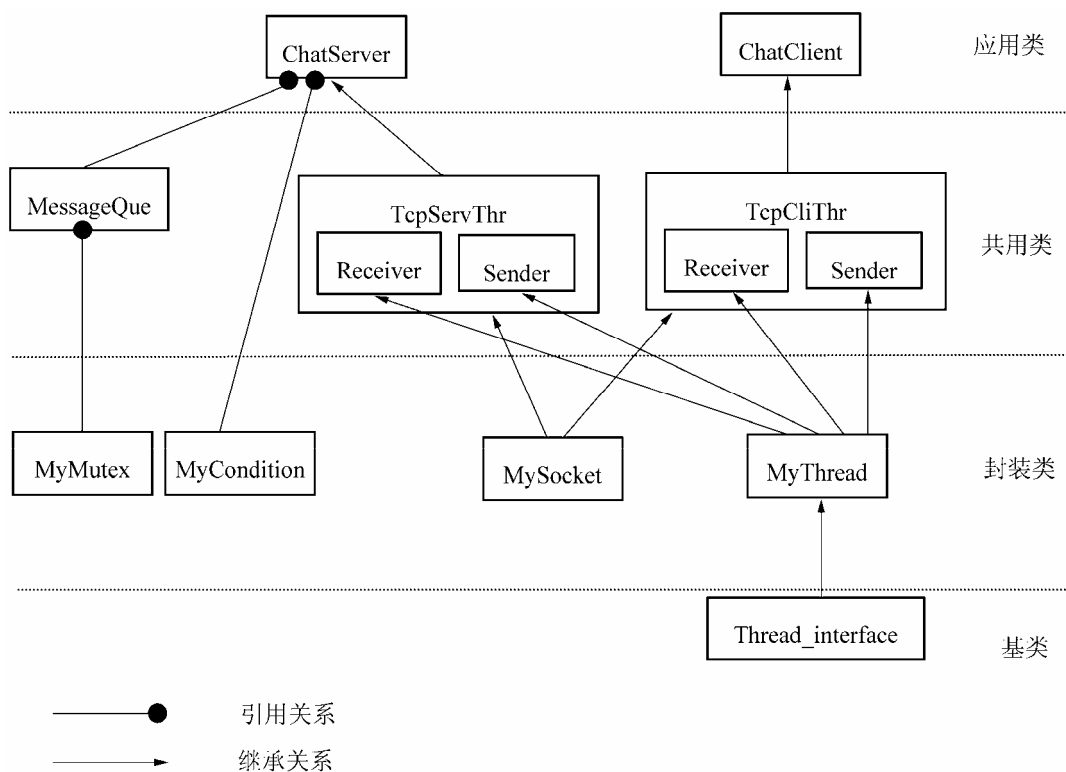


图 10.1 套接字类库对象关系图

从图中可以看出，TcpServThr 类和 TcpCliThr 类从 MySocket 继承而来，可以方便地实现网络操作。它们包括以下两个内部类。

- Receiver 类：用于产生接收客户数据的线程。
- Sender 类：用于产生向客户发送数据的线程。

这两个内部类从 `MyThread` 继承而来，用于实现多线程。由于这两个内部类仅为了方便地使用 `TcpServThr` 类和 `TcpCliThr` 类，并不具有普遍意义，因此将它们定义为内部的。在 `MessageQue` 类中引用了 `MyMutex` 类，以实现互斥锁来保护队列数据的完整性。

读者可以通过引用关系或继承关系生成新的类，例如继承 `MySocket` 类以生成多进程并发服务器。也可以对已有的类进行修改从而增加新的功能，使其更加完善。由此看出，建立套接字类库能迅速地产生新的应用，也能方便地对原系统进行升级，程序也具有良好的结构，便于阅读和调试，从而极大地提高开发效率。

### 10.3.2 套接字系统调用：MySocket 类

`MySocket` 类是该套接字类库的核心，它封装了与套接字相关的主要系统调用。其源代码见附录 A。

#### 1. MySocket 类的成员变量

在一个类中，成员变量代表了该类的特性及状态。`MySocket` 类的成员变量存放与套接字相关的信息，其内容如下。

- `address_family`: 存放地址簇，如 `AF_INET`。
- `protocol_family`: 存放协议簇，如 `IPPROTO_TCP`。
- `socket_type`: 套接字类型，如 `SOCK_STREAM`、`SOCK_DGRAM`。
- `port_number`: 端口号。
- `Mysocket`: 套接字的描述符。对于 `SOCK_STREAM` 类型的套接字为侦听套接字描述符。
- `conn_socket`: 连接套接字描述符。对于 `SOCK_DGRAM` 类型的套接字无效。
- `socket_error`: 自定义的套接字错误码。
- `bytes_read`: 读入的字节数。
- `bytes_move`: 写出的字节数。
- `is_connected`: 是否已连接。0 为未连接，否则已连接。
- `is_bound`: 套接字是否已绑定。
- `sin`: 本地套接字地址。
- `remote_sin`: 远程套接字地址。

#### 2. MySocket 类实现的功能

`MySocket` 类所实现的功能分为以下几类。

- 参数设置。用于设置与套接字相关的参数，如 `SetAddressFamily()`、`SetProtocolFamily()`、`SetPortNumber()`等。
- 套接字功能。用于初始化/关闭套接字，以及绑定连接等操作，如 `InitSocket()`、`Close()`、`Bind()`、`Connect()`、`Accept()`、`Listen()`、`ShutDown()`等。
- 基于流的 I/O 功能。用于 TCP 的读/写操作，如 `Recv()`、`Send()`、`RemoteRecv()`、`RemoteSend()`等。
- 套接字信息数据库功能。用于获取与套接字相关的参数，如 `GetPeerName()`、`GetSockName()`、`GetIPAddress()`等。

- 套接字状态功能。用于获得当前套接字状态，如 `IsConnected()`、`IsBound()`、`SetSocket()`等。
- 基于数据报 I/O 功能。用于 UDP 的读写操作，如 `RecvFrom()`、`SendTo()`等。
- 异常处理功能。用于获取或设置错误码，如 `GetSocketError()`、`SetSocketError()`等。

### 3. 利用 OOP 的重载功能

为了适应不同的用途，同一系统调用被封装到不同的成员函数中，这些函数使用同一名称却有不同参数列表，这是利用 OOP 的重载来实现的。例如，基于 TCP 的读功能有四个成员函数：

- `int Recv(void *buf, int bytes, int flags = 0)`：只有当 `MySocket` 类为客户所使用时，才调用该函数。它从 `MySocket` 套接字中读数据。
- `int Recv(void *buf, int bytes, int seconds, int useconds, int flags = 0)`：与上一函数一样，但增加了超时处理功能。
- `int Recv(int s, void *buf, int bytes, int flags = 0)`：从一指定的套接字中读数据。
- `int Recv(int s, void *buf, int bytes, int seconds, int useconds, int flags = 0)`：与上一函数一样，但增加了超时处理功能。

## 10.3.3 多线程实现：MyThread 类

### 1. MyThread 类的头文件(mythread.h)

`MyThread` 类用于实现多线程。其头文件 `mythread.h` 如下：

```

1  /*****
2  class name : Thread_interface
3  function: It is base class of all threads. used by class Mythread.
4  *****/

5  class Thread_interface
6  {
7  public:
8  virtual    void run() {}
9  };

10 /*****
11 class name : MyThread
12 function: Create thread and run it.
13 *****/

14 class MyThread : public Thread_interface
15 {
16 Thread_interface* worker;
17 int error;
18 pthread_t id;
```

```

19 static void* run(void*);

20 public:
21 MyThread(Thread_interface& w);
22 MyThread() {worker = this;}

23 int Start();
24 void Exit(void *value_ptr) { ::pthread_exit(value_ptr);}
25 int Join(pthread_t thread, void **value_ptr)
    { error = ::pthread_join(thread, value_ptr); return error;}
26 int Cancel(pthread_t target_thread)
    { error = ::pthread_cancel(target_thread); return error;}
27 int Detach() {error = ::pthread_detach(id); return error;}
28 int Detach(pthread_t thread)
    {error = ::pthread_detach(thread); return error;}
29 pthread_t getId() {return id;}
30 int Error(){return error;}
31 };

```

代码说明如下。

- 1~9: 定义 MyThread 的基类 Thread\_interface，它仅包含一个虚函数 run()。
- 16: 定义指向 Thread\_interface 的指针。所指的对象是线程将要执行的实体。
- 18: 定义 id，用于存放线程 ID。
- 19: 声明静态的函数 run()，用于产生线程。
- 21~22: 声明 MyThread 类构造函数。
- 23~30: 声明成员函数。

## 2. MyThread 类的实现代码(mythread.cpp)

MyThread 类的实现代码如下:

```

1 /* File : mythread.cpp */
2 #include <iostream.h>
3 #include <pthread.h>
4 #include <unistd.h>
5 #include <stdio.h>
6 #include <errno.h>
7 #include "mythread.h"

8 MyThread::MyThread(Thread_interface& w)
9 {
10 worker = &w;
11 error =0;
12 }

```

```
13 void* MyThread::run(void* w)
14 {
15     ((Thread_interface *)w)->run();
16 }

17 int MyThread::Start()
18 {
19     error = pthread_create(&id, NULL, run, (void *)worker);
20     return error;
21 }
```

代码说明如下。

8~12: 实现构造函数。将一个 Thread\_interface 的对象赋给变量 worker。默认时将自身 (this) 赋给变量 worker, 因为 MyThread 自身也是一个 Thread\_interface 对象。

13~16: 执行 worker 对象的 run() 函数。

17~21: 通过系统调用 pthread\_create() 产生线程。

### 3. 产生线程

实现的代码并不多, 但略显得复杂。这样做的原因是为了方便使用。我们可以采用两种方法产生线程。一种方法是, 首先生成一个从 MyThread 继承而来的类, 并在该类中执行 run() 函数。最后执行函数 Start() 产生并执行线程。例子如下:

```
class Thread1: public MyThread
{
    ...
public:
    void run() {
        // 线程所执行的功能
        ...
    }
    ...
};

main()
{
    ...
    Thread1 t;
    ...
    t.Start();
    ...
}
```

另一种方法是, 首先生成一个从 Thread\_interface 继承而来的类, 并在类中执行 run() 函数。然后产生一个 MyThread 对象, 其初始参数为新生成的对象, 最后执行 Start() 函数产生并执行线程。例子如下:

```
class Thread2: public Thread_interface
```



```

{
    ...
public:
    void run() {
        //thread implement
        ...
    }
    ...
};
main()
{
    ...
    Thread2 t2;
    ...
    (new MyThread(t2)).Start();
    ...
}

```

这两种方法与 Java 线程的使用方法类似，非常简明直观，并且完全是以对象为核心的方法来产生线程。在实际应用中，根据需要，可以只实现一个类。为了产生线程，只需使该类从 `MyThread` 或 `Thread_interface` 派生，然后实现该类的 `run()` 函数，该函数将覆盖父类的 `run()` 函数，从而被产生的线程所执行。

📢 注意：在实现 `MyThread` 类时，一定要定义静态的成员函数 `run()`，并且由它来执行将被子类覆盖的 `run()` 函数（在 `Thread_interface` 中定义）。这是因为 `pthread_create()` 函数只能接受静态的函数为参数，而静态的函数又不能被子类所覆盖。

### 10.3.4 加锁/解锁：MyMutex 类和 MyCondition 类

`MyMutex` 类用于加锁/解锁功能，以保护共享数据对象的完整性。`MyCondition` 类用于实现条件变量，以解决线程中的同步问题。

#### 1. 头文件(MySync.h)

头文件如下：

```

1  /*****
2  class name : MyMutex
3  Function: support mutex
4  *****/
5  class MyMutex
6  {
7  pthread_mutex_t a_mutex;
8  int error;

9  public:
10 MyMutex();

```

```

11 ~MyMutex();

12 int Lock() { error = pthread_mutex_lock(&a_mutex);return error;}
13 int Trylock() { error = pthread_mutex_trylock(&a_mutex); return error;}
14 int Unlock() {error = pthread_mutex_unlock(&a_mutex); return error;}
15 int Error() {return error;}
16 };

17 /*****
18 class name : MyCondition
19 Function: support MyCondition variable
20 *****/

21 class MyCondition
22 {
23 pthread_mutex_t a_mutex;
24 pthread_cond_t got_request;
25 int error;

26 public:
27 MyCondition();
28 ~MyCondition();

29 int wait(int second = 0);    // wait until signal by other thread
30 int wake();                 // wake a thread waiting for this condition
31 int wakeAll();              // wake all threads waiting for this conditoin
32 int Error() {return error;}
33 };

```

代码说明如下：

7~8：定义 MyMutex 成员变量 a\_mutex 和 error。a\_mutex 用于存放互斥锁。error 用于存放错误码。

10~11：声明 MyMutex 的构造函数及析构函数。

12~15：声明并实现 MyMutex 的成员函数，包括加/锁操作。

23~25：定义 MyCondition 类的成员变量，其中包括互斥锁 a\_mutex 和条件变量 got\_request。

29~30：声明 MyCondition 的成员函数。

## 2. 实现代码(mysync.cpp)

实现代码如下：

```

1 /* File : mysync.cpp */
2 #include <iostream.h>
3 #include <pthread.h>
4 #include <unistd.h>

```

```
5 #include <stdio.h>
6 #include <errno.h>
7 #include "MySync.h"

8 MyMutex::MyMutex()
9 {
10 error = pthread_mutex_init(&a_mutex, NULL);
11 }

12 MyMutex::~MyMutex()
13 {
14 error = pthread_mutex_destroy(&a_mutex);
15 }

16 MyCondition::MyCondition()
17 {
18 error = pthread_cond_init(&got_request, NULL);
19 error += pthread_mutex_init(&a_mutex, NULL);
20 }

21 MyCondition::~MyCondition()
22 {
23 error = pthread_mutex_destroy(&a_mutex);
24 int rc = pthread_cond_destroy(&got_request);
25 /* some thread is still waiting on this condition variable */
26 while (rc == EBUSY) {
27     sleep(1);
28 }
29 error = error + rc;
30 }

30 int MyCondition::wait(int second = 0)
31 {
32 struct timeval now;      /* time when we started waiting */
33 struct timespec timeout; /* timeout value for the wait function */
34 int done;                /* are we done waiting? */

35 /* first, lock the mutex */
36 int rc = pthread_mutex_lock(&a_mutex);
37 if (rc) { /* an error has occurred */
38     perror("pthread_mutex_lock");
39     pthread_exit(NULL);
40 }

41 if (second) {
```

```
42  /* get current time */
43  gettimeofday(&now,NULL);
44  /* prepare timeout value */
45  timeout.tv_sec = now.tv_sec + second;
46  timeout.tv_nsec = now.tv_usec * 1000; /* timeval uses microseconds.*/
47  /* timespec uses nanoseconds. */
48  /* 1 nanosecond = 1000 micro seconds. */

49  error = pthread_cond_timedwait(&got_request, &a_mutex, &timeout);
50 }else error = pthread_cond_wait(&got_request, &a_mutex);

51 /* finally, unlock the mutex */
52 pthread_mutex_unlock(&a_mutex);

53 switch(error) {
/* we were awakened due to the cond. variable being signaled */
54 case 0:
55     /* the mutex was now locked again by pthread_cond_timedwait. */
56     /* do your stuff here... */
57     return 0;

58 case ETIMEDOUT: /* our time is up */
59     return 1;

60 default:        /* some error occurred (e.g. we got a Unix signal) */
61     return 2;    /* break this switch, but re-do the while loop. */
62 }
63 }

64 int MyCondition::wake()
65 {
66 error = pthread_cond_signal(&got_request);
67 return error;
68 }

69 int MyCondition::wakeAll()
70 {
71 error = pthread_cond_broadcast(&got_request);
72 return error;
73 }
```

代码说明如下。

2~7：所需头文件。

8~11：实现 MyMutex 的构造函数，初始化一个互斥锁。

12~15：实现 MyMutex 的析构函数，撤销互斥锁。

16~20：实现 MyCondition 的构造函数，初始化一个互斥锁及条件变量。

21~29：实现 MyCondition 的析构函数，撤销互斥锁及条件变量，如果还有线程守候该条件变量，则等待直至该线程退出。

30~63：实现 wait()函数，它让线程守候在 MyCondition 对象的条件变量 got\_request 上直至被唤醒或超时。

36~40：加锁。

41~49：设置定时，并守候。

52：解锁。

53~61：错误处理。

64~68：唤醒一个线程。

69~73：实现 wakeAll()函数，用于唤醒所有守候线程。

### 10.3.5 基于 TCP 的多线程并发服务器：TcpServThr 类

TcpServThr 类是基于 TCP 的多线程并发服务器类，它完成的功能如下。

- TCP 套接字的创建、绑定，并且负责建立/关闭连接套接字，以及套接字读/写操作。TcpServThr 类通过继承 MySocket 类来实现上述功能。
- 并发功能。为每次连接的读/写操作分别产生不同的线程以完成操作，并且管理这些线程的产生和撤销。TcpServThr 类通过定义两个内部类 Receiver 和 Sender 来产生线程，分别完成发送和接收功能，并且建立一个线程队列以完成线程的管理。

#### 1. 头文件(TcpServThr.h)

头文件 TcpServThr.h 定义如下：

```
1 /*****
2 class name : TcpServThr
3 Function: support TCP Server with multithread
4 *****/
5 class TcpServThr : public MySocket
6 {
7     int max_connections;
8     vector<MyThread*>* ThrSet;

9 public:
10    TcpServThr();
11    TcpServThr(int port, char *hostname = NULL);
12    TcpServThr(int port, int maxconn, char *hostname = NULL);
13    virtual ~TcpServThr();

14    void SetMaxConn(int num) {max_connections = num;}
15    int GetMaxConn() {return max_connections;}
16    int Init();
```

```
17 int Run();

18 virtual void DealRecv(MyThread* thread);
19 virtual void DealSend(MyThread* thread);

20 protected:
21 int CreateThr(MyThread** Rthread, MyThread** Wthread);
22 void AddThread(MyThread* thread);
23 void DelThread(MyThread* thread);
24 int WaitAllThr();

25 class Receiver : public MyThread
26 {
27 public:
28 int socket;
29 TcpServThr* server;

30 Receiver(int connsocket, TcpServThr* serv) {
31     socket = connsocket;
32     server = serv;
33 }
34 void run() { server->DealRecv(this); }
35 };

36 class Sender : public MyThread
37 {
38 public:
39 int socket;
40 TcpServThr* server;

41 Sender(int connsocket, TcpServThr* serv) {
42     socket = connsocket;
43     server = serv;
44 }
45 void run() { server->DealSend(this); }
46 };
```

头文件的说明如下。

7~8：定义成员变量。max\_connections 存放最大的连接数量。ThrSet 是一个指向线程队列的指针，该队列存放当前所有的线程。

10~13：定义构造函数及析构函数。

14：定义 SetMaxConn()函数，用于设置最大连接数。

15：定义 GetMaxConn()函数，用于获得最大连接数。

16：定义 Init()函数，用于初始化 TCP 多线程服务器，产生监听套接字并绑定。

17：定义 Run()函数，用于运行 TCP，建立连接。

18~19：定义 DealRecv()和 DealSend()函数，用于处理接收和发送过程。把这两个函数定义为虚函数，使子类能重载它们以满足不同的应用需要。

21：定义 CreateThr()函数，用于产生收/发线程。

22：定义 AddThread()函数，用于将线程加入线程队列。

23：定义 DelThread()函数，用于将线程从线程队列中删除。

24：定义 WaitAllThr()函数，用于线程结束时的同步，即主线程退出前，应等待所有其他线程退出。

25~35：定义 Receiver 类，用于产生接收线程。

36~46：定义 Sender 类，用于产生发送线程。

## 2. 实现代码(tcpservthr.cpp)

TcpServThr 类实现的代码如下：

```
1 /* File : tcpservthr.cpp */
2 #include <stdio.h>
3 #include <vector.h>
4 #include "pthread.h"
5 #include "Mysocket.h"
6 #include "MyThread.h"
7 #include "TcpServThr.h"

8 TcpServThr::TcpServThr()
9 {
10 max_connections = MAXCONN;
11 ThrSet = new vector<MyThread*>();
12 }

13 TcpServThr::TcpServThr(int port, char *hostname):
14 MySocket(AF_INET, SOCK_STREAM, 0, port, hostname)
15 {
16 max_connections = MAXCONN;
17 ThrSet = new vector<MyThread*>();
18 }

19 TcpServThr::TcpServThr(int port, int maxconn, char *hostname):
20 MySocket(AF_INET, SOCK_STREAM, 0, port, hostname)
21 {
22 max_connections = maxconn;
23 ThrSet = new vector<MyThread*>();
24 }

25 TcpServThr::~TcpServThr()
26 {
```

```
27 /* Wait until all threads exits */
28 WaitAllThr();
29 /* free all memory of threads */
30 vector<MyThread*>::iterator it = ThrSet->begin();
31 while (it != ThrSet->end()) {
32     MyThread* thr = (MyThread*)(*it);
33     delete thr;
34     it++;
35 }

36 delete ThrSet;
37 }

38 int TcpServThr::Init()
39 {
40     int opt = SO_REUSEADDR;
41     setsockopt(Mysocket, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

42     if (Bind() == -1) return socket_error;
43     if (Listen(max_connections) == -1) return socket_error;
44     return 0;
45 }

46 int TcpServThr::Run()
47 {
48     while (Accept() != -1) {
49         /* Create new thread */
50         MyThread *Rthread, *Wthread;
51         if(CreateThr(&Rthread,&Wthread) == -1) return -1;
52         AddThread(Rthread);
53         AddThread(Wthread);
54         if (Rthread->Start()) return -1;
55         if (Wthread->Start()) return -1;
56     }
57     return -1;
58 }

59 int TcpServThr::CreateThr(MyThread** Rthread, MyThread** Wthread)
60 {
61     printf("Accept Conn\n");
62     *Rthread = new Receiver(conn_socket, this);
63     *Wthread = new Sender(conn_socket, this);
64     return 0;
65 }
```



```
66 void TcpServThr::DealRecv(MyThread* thread)
67 {
68     printf("Receiver is running!\n");
69 }

70 void TcpServThr::DealSend(MyThread* thread)
71 {
72     printf("Sender is running!\n");
73 }

74 void TcpServThr::AddThread(MyThread* thread)
75 {
76     if (thread == NULL) return;
77     ThrSet->push_back(thread);
78 }

79 void TcpServThr::DelThread(MyThread* thread)
80 {
81     vector<MyThread*>::iterator it = ThrSet->begin();
82     while (it != ThrSet->end()) {
83         if ((MyThread *)*it == thread) {
84             ThrSet->erase(it);
85             break;
86         }
87         it++;
88     }
89 }

90 int TcpServThr::WaitAllThr()
91 {
92     vector<MyThread*>::iterator it = ThrSet->begin();
93     while (it != ThrSet->end()) {
94         MyThread* thr = (MyThread*)(*it);
95         pthread_join(thr->getId(), NULL);
96         it++;
97     }
98     return 0;
99 }
```

代码说明如下。

8~24：实现构造函数，用于设置与 TCP 套接字相关参数及产生线程队列。

25~35：实现析构函数。首先等待所有线程退出，然后扫描线程队列，删除所有线程对象，最后删除线程队列。

38~45：实现 Init()函数，包括 TCP 套接字的绑定和监听。

46~58：实现 Run()函数，反复接收客户连接，一旦连接建立，则产生收/发线程，并将

它们加入线程队列，再运行它们。

59~65：实现 CreateThr()函数，产生收/发线程对象。

66~69：默认接收处理。它没有实际的意义，在具体的应用中将被重载。

70~73：默认发送处理。它没有实际的意义，在具体的应用中将被重载。

74~78：将线程对象加入线程队列。

79~89：将指定的线程对象从线程队列中删除。

90~99：实现 WaitAllThr()函数，以等待线程队列中所有线程退出。扫描线程队列，等待每个线程退出。

### 10.3.6 TCP 多线程客户类：TcpCliThr 类

TcpCliThr 类是基于 TCP 的多线程客户类，它完成的功能如下。

- TCP 套接字的创建、关闭以及与服务器的连接及读/写操作，它通过继承 MySocket 类来实现上述功能。
- 并发功能。与 TcpServThr 完全相同。

#### 1. 头文件(TcpCliThr.h)

头文件 TcpCliThr.h 定义如下：

```

1  /*****
2  class name : TcpCliThr
3  Function: support TCP client with multithread
4  *****/
5  class TcpCliThr : public MySocket
6  {
7  vector<MyThread*>* ThrSet;

8  public:
9  TcpCliThr();
10 TcpCliThr(int port, char *server);
11 virtual ~TcpCliThr();

12 int ConnectServ();
13 virtual void DealRecv(MyThread* thread);
14 virtual void DealSend(MyThread* thread);

15 protected:
16 int CreateThr(MyThread** Rthread, MyThread** Wthread);
17 void AddThread(MyThread* thread);
18 void DelThread(MyThread* thread);
19 int WaitAllThr();

20 class Receiver : public MyThread
21 {

```

```

22 public:
23 int socket;
24 TcpCliThr* server;

25 Receiver(int connsocket, TcpCliThr* serv) {
26     socket = connsocket;
27     server = serv;
28 }
29 void run() { server->DealRecv(this); }
30 };

31 class Sender : public MyThread
32 {
33 public:
34 int socket;
35 TcpCliThr* server;

36 Sender(int connsocket, TcpCliThr* serv) {
37     socket = connsocket;
38     server = serv;
39 }
40 void run() { server->DealSend(this); }
41 };
42 };

```

头文件的说明如下。

7：定义成员变量 ThrSet，存放线程队列。

9~10：定义构造函数。

11：定义析构函数。必须把它设置为虚函数以便子类被撤销时，能自动调用该函数，完成相应的清理工作。

12：定义 ConnectServ()函数，用于与服务器相连接。

13~14：定义 DealRecv()和 DealSend()函数，用于处理接收和发送过程。把它定义为虚函数，使子类能覆盖它们以满足不同的应用需要。

16：定义 CreateThr()函数，用于产生收/发线程。

17：定义 AddThread()函数，用于将线程加入线程队列。

18：定义 DelThread()函数，用于将线程从线程队列中删除。

19：定义 WaitAllThr()函数，用于线程结束时的同步。即主线程退出前，应等待所有其他线程退出。

20~30：定义 Receiver 类，用于产生接收线程。

31~42：定义 Sender 类，用于产生发送线程。

## 2. 实现代码(tcpclithr.cpp)

TcpCliThr 类实现的代码如下：

```
1 /* File : tcpclithr.cpp */
2 #include <stdio.h>
3 #include <vector.h>
4 #include "pthread.h"
5 #include "Mysocket.h"
6 #include "MyThread.h"
7 #include "TcpCliThr.h"

8 TcpCliThr::TcpCliThr()
9 {
10 ThrSet = new vector<MyThread*>();
11 }

12 TcpCliThr::TcpCliThr(int port, char *server):
13 MySocket(AF_INET, SOCK_STREAM, 0, port, server)
14 {
15 ThrSet = new vector<MyThread*>();
16 }

17 TcpCliThr::~~TcpCliThr()
18 {
19 /* Wait until all threads exits */
20 WaitAllThr();
21 /* free all memory of threads */
22 vector<MyThread*>::iterator it = ThrSet->begin();
23 while (it != ThrSet->end()) {
24     MyThread* thr = (MyThread*)(*it);
25     delete thr;
26     it++;
27 }

28 delete ThrSet;
29 }

30 int TcpCliThr::ConnectServ()
31 {
32     if (Connect() == -1) return -1;
33     /* Create new thread */
34     MyThread *Rthread, *Wthread;
35     if (CreateThr(&Rthread, &Wthread) == -1) return -1;
36     AddThread(Rthread);
37     AddThread(Wthread);
38     if (Rthread->Start()) return -1;
39     if (Wthread->Start()) return -1;
40     return 0;
```

```
41 }

42 int TcpCliThr::CreateThr(MyThread** Rthread, MyThread** Wthread)
43 {
44     printf("Accept Conn\n");
45     *Rthread = new Receiver(Mysocket, this);
46     *Wthread = new Sender(Mysocket, this);
47     return 0;
48 }

49 void TcpCliThr::DealRecv(MyThread* thread)
50 {
51     printf("Receiver is running!\n");
52 }

53 void TcpCliThr::DealSend(MyThread* thread)
54 {
55     printf("Sender is running!\n");
56 }

57 void TcpCliThr::AddThread(MyThread* thread)
58 {
59     if (thread == NULL) return;
60     ThrSet->push_back(thread);
61 }

62 void TcpCliThr::DelThread(MyThread* thread)
63 {
64     vector<MyThread*>::iterator it = ThrSet->begin();
65     while (it != ThrSet->end()) {
66         if (((MyThread *)*it) == thread) {
67             ThrSet->erase(it);
68             break;
69         }
70         it++;
71     }
72 }

73 int TcpCliThr::WaitAllThr()
74 {
75     vector<MyThread*>::iterator it = ThrSet->begin();
76     while (it != ThrSet->end()) {
77         MyThread* thr = (MyThread*)(*it);
78         pthread_join(thr->getId(), NULL);
79         it++;
80     }
81 }
```

```
80 }  
81 return 0;  
82 }
```

代码说明如下。

8~16：实现构造函数，用于设置与 TCP 套接字相关参数并产生线程队列。

17~29：实现析构函数。首先等待所有线程退出，然后扫描线程队列，删除所有线程对象，最后删除线程队列。

30~41：实现 ConnectServ()函数。一旦连接建立，则产生收/发线程，将它们加入线程队列并运行。

42~48：实现 CreateThr()函数，产生收/发线程对象。

49~52：默认接收处理。它没有实际的意义，在具体的应用中将重载。

53~56：默认发送处理。它没有实际的意义，在具体的应用中将重载。

57~61：将线程对象加入线程队列。

62~72：将指定的线程对象从线程队列中删除。

73~82：实现 WaitAllThr()函数，以等待线程队列中所有线程退出。扫描线程队列，等待每个线程退出。

## 10.4 实例分析

建立了自己的套接字类库后，就可以根据实际应用，设计应用程序。以下通过实例说明如何使用建立的套接字类库。

本实例是基于 TCP 的聊天室应用系统，分为服务器和客户两部分。其基本原理是：客户首先与服务器建立 TCP 连接，并发送客户名称给服务器，然后就可以通过服务器发信息给其他已连接的客户，同时通过服务器接收其他客户的信息。

### 10.4.1 实现聊天室服务器

聊天室服务器完成的功能如下。

- 建立和管理与客户的 TCP 连接。通过继承 TcpServThr 类实现。
- 实现并发功能。通过继承 TcpServThr 类实现。
- 实现信息队列的管理，以保存客户发来的信息并决定向客户发什么消息。通过引用 MessageQue 类来实现。
- 实现与客户的收/发消息的功能。通过重载 TcpServThr 类的 DealRecv()和 DealSend()函数来实现，并通过引用 MyCondition 类来解决收/发线程间的同步问题。只有当收到一个客户消息时才唤醒发送线程。

1. 头文件(Chat.h)

其头文件如下：

```
1  /* File : chat.h */
```

```

2  /* Define constant */
3  const int MAX_USRS = 10;
4  const int QUE_LEN = 10;
5  const int MAX_PACKET_LEN = 1000;
6  const int MAX_NAME_LEN = 100;

7  struct Message
8  {
9      long sn;
10     int connection;
11     char* message;
12 };

13 /*****
14 class name : MessageQue
15 Function: A queue for message
16 *****/
17 class MessageQue
18 {
19     long lastSN;
20     int queLen; // The length of the queue
21     vector<Message*> *queue;
22     MyMutex* lock;
23 public:
24     MessageQue(int len);
25     ~MessageQue();

26     int Add(int conn, char* m);
27     int Get(int conn, long* maxsn, char* m);
28     long GetSN() {return lastSN;}
29     void SetSN(int sn) { lastSN = sn;}
30     int GetQueLen { return queLen;}
31     void SetQueLen(int l) { queLen = l;}
32 };

33 /*****
34 class name : ChatServer
35 Function: A server used to chat with multiple users
36 *****/
37 class ChatServer : public TcpServThr
38 {

39     int max_usrs;
40     int queLen;
41     MessageQue *msg;

```

```
42 MyCondition *con;
43 public:
44 ChatServer();
45 ChatServer(int port, char *hostname = NULL);
46 ChatServer(int port, int max_conn, int maxusers, int len,
            char *hostname = NULL);
47 virtual ~ChatServer();

48 void SetMaxUser(int num) {max_usrs = num;}
49 int GetMaxUser() { return max_usrs;}
50 void DealRecv(MyThread* thread);
51 void DealSend(MyThread* thread);
52 void SetQueLen(int l) { queLen = l;}
53 int GetQueLen() { return queLen;}
54 };
```

头文件的说明如下。

2~6：定义常量，包括最大用户数、信息队列长度、最大包长度和名字最大长度。

7~12：定义结构 Message，它包括变量 sn，用于存放该信息的序列号，服务器为每个收到的客户信息连续编号；变量 connection 用于存放连接套接字。变量 message 存放客户发送的字符串。

13~25：定义 MessageQue 类。该类包括一个信息队列，并且该队列由互斥变量进行保护，以避免多线程不同步而造成的队列信息的不一致。变量 last SN 用于存放当前最新的序列号；变量 queLen 存放当前队列的长度；queue 用于指向 Message 队列；变量 lock 用于存放互斥变量。构造函数 MessageQue()用于初始化队列。析构函数~MessageQue()用于队列对象的撤销处理。

该类的成员函数如下。

- Add(): 往队列中增加信息。
- Get(): 从队列中取出信息。
- GetSN(): 取出当前序列号。
- SetSN(): 设置当前序列号。
- GetQueLen(): 获得队列长度。
- SetQueLen(): 设置队列长度。

34~52：定义 ChatServer 类。该类是一个基于 TCP 的多线程服务器，用于交换不同客户输入的信息，以完成对话功能。它继承于 TcpServThr 类，从而可以方便地实现多线程 TCP 服务器。成员变量如下。

- max\_usrs: 最大用户数。
- queLen: 队列长度。
- msg: 指向队列的指针。
- con: 指向条件变量，以完成多线程间的同步。

其构造函数包括 3 种形式，以满足不同初始化要求。



该类的成员函数如下。

- DealRecv(): 用于处理接收数据。
- DealSend(): 用于处理发送数据。
- SetMaxUser(): 设置最大用户数。
- GetMaxUser(): 获得最大用户数。
- SetQueLen(): 设置队列长度。
- GetQueLen(): 获得队列长度。

## 2. 实现代码(chat.cpp)

实现代码如下：

```
1  /* File : chat.cpp */
2  #include <stdio.h>
3  #include <string.h>
4  #include <vector.h>
5  #include "pthread.h"

6  #include "Mysocket.h"
7  #include "MyThread.h"
8  #include "MySync.h"
9  #include "TcpServThr.h"

10 #include "Chat.h"

11 MessageQueue::MessageQueue(int len)
12 {
13     lastSN = 1;
14     queLen = len;
15     queue = new vector<Message*>;
16     lock = new MyMutex;
17 }

18 MessageQueue::~MessageQueue()
19 {
20     /* delete all messages */
21     vector<Message*>::iterator it = queue->begin();
22     while (it != queue->end()) {
23         Message* m = (Message*)*it;
24         delete m->message;
25         delete m;
26         it++;
27     }
28     delete queue;
29     delete lock;
30 }
```

```
31 int MessageQue::Add(int conn, char* m)
32 {
33     if (m == NULL) return -1;
34     Message *mes;
35     mes = new Message;
36     mes->connection = conn;
37     mes->message = new char[strlen(m)];
38     strcpy(mes->message, (const char *) m);
39     lock->Lock();
40     mes->sn = lastSN;
41     lastSN++;
42     if (queue->size() == queLen) {
43         vector<Message*>::iterator it = queue->begin();
44         queue->erase(it);
45     }
46     queue->push_back(mes);
47     lock->Unlock();
48     return 0;
49 }

50 int MessageQue::Get(int conn, long* maxsn, char* m)
51 {
52     int err;
53     err = 1;
54     lock->Lock();
55     vector<Message*>::iterator it = queue->begin();
56     while (it != queue->end()) {
57         Message* mes = (Message*) *it;
58         if ((mes->sn > *maxsn) && (mes->connection != conn)) {
59             strcpy(m, mes->message);
60             *maxsn = mes->sn;
61             m = m + strlen(mes->message) - 1;
62             err = 0;
63         }
64         it++;
65     }
66     lock->Unlock();
67     return err;
68 }

69 ChatServer::ChatServer()
70 {
71     SetMaxConn(MAXCONN);
72     max_usrs = MAX_USRS;
```

```
73  msg = new MessageQue(QUE_LEN);
74  con = new MyCondition;
75  }

76  ChatServer::ChatServer(int port, char *hostname):
77  TcpServThr(port,hostname)
78  {
79  SetMaxConn(MAXCONN);
80  max_usrs = MAX_USRS;
81  msg = new MessageQue(QUE_LEN);
82  con = new MyCondition;
83  }

84  ChatServer::ChatServer(int port, int max_conn, int maxusr,
                        int len, char *hostname):
85  TcpServThr(port, max_conn,hostname)
86  {
87  max_usrs = maxusr;
88  queLen = len;
89  msg = new MessageQue(queLen);
90  con = new MyCondition;
91  }

92  ChatServer::~ChatServer()
93  {
94  /* Wait until all threads exits */
95  WaitAllThr();    // must execute it before ~TcpServThr
96  delete msg;
97  delete con;
98  }

99  void ChatServer::DealRecv(MyThread* thread)
100 {
101 printf("Receiver is running!\n");
102 char buf[MAX_PACKET_LEN];

103 int socket = ((Receiver*) thread)->socket;
104 int len = recv(socket,buf,MAX_NAME_LEN,0);
105 buf[len - 1] = ':';
106 buf[len] = '\0';
107 printf("%s %d\n",buf,len);

108 while (1) {
109     int len1 = recv(socket,buf + len,MAX_PACKET_LEN,0);
110     if (len1 < 1) break;
```

```

111     buf[len1 + len] = '\0';
112     printf("Recv:%s",buf);
113     msg->Add(socket, buf);
114     con->wakeAll();
115 }
116 DelThread(thread);
117 }

118 void ChatServer::DealSend(MyThread* thread)
119 {
120     char buf[MAX_PACKET_LEN];
121     long maxsn;

122     maxsn = msg->GetSN() - 1;
123     int socket = ((Receiver*) thread)->socket;
124     printf("Sender is running!\n");
125     while(1) {
126         con->wait();
127         int err = msg->Get(socket,&maxsn,buf);
128         if (err) continue;
129         printf("Send:%s",buf);
130         Send(socket,buf,strlen(buf),0);
131     }
132     DelThread(thread);
133 }

134 /*****
135 main()
136 {
137     ChatServer chat(1234);
138     chat.Init();
139     chat.Run();
140 }

```

代码说明如下。

2~10：所需头文件。

11~17：实现 MessageQue 类的构造函数 MessageQue()。初始的序列号为 1，队列长度由参数 len 决定。初始化队列 MessageQue 和互斥对象 MyCondition。

18~30：实现 MessageQue 类的析构函数~MessageQue()。首先扫描队列，删除所有字符串信息所占内存。然后删除队列和互斥对象。

31~49：实现 MessageQue 类的成员函数 Add()。首先，产生一个消息结构，并填充相应信息（连接套接字、字符串），然后加锁以修改关键的数据。将序列号加 1，如果队列已满，则删除第 1 个信息。再将新的信息加入队列。最后，进行解锁。

50~68：实现 MessageQue 类的成员函数 Get()。首先加锁，扫描队列。如果消息的序列

号大于当前客户所收到的最大序列号，并且该消息不是当前客户所发送的，则取出当前消息，并置错误码为 0。最后，解锁并返回错误码。

69~91：实现 ChatServer 类的构造函数 ChatServer()，包括初始化最大连接数量、服务器监听的端口号和地址、最大用户数、队列长度、队列及条件变量。由于变量 max\_connections 在 TcpServThr 类中是私有的，其子类不能直接修改它，必须通过公有的成员函数 SetMaxConn()来设置。

92~98：实现 ChatServer 类的析构函数~ChatServer()。首先，主线程等待所有其他线程终止，然后删除信息队列和条件变量。

99~117：实现 ChatServer 类的成员函数 DealRecv()。首先接收客户名称，然后反复接收客户发来的字符串消息。一旦收到消息则加入到消息队列中，然后唤醒其他线程，以便将消息广播到其他客户。当用户终止连接后，调用 DelThread()将该线程从线程队列中删除。

118~133：实现 ChatServer 类的成员函数 DealSend()。首先，进入睡眠状态等待被唤醒。唤醒后从消息队列中取出需要发送的消息，然后发送给客户。连接终止后，调用 DelThread()将该线程从线程队列中删除。

135~140：主程序。产生 ChatServer 对象 chat，其守候的端口号为 1234。然后初始化 chat，并运行它。

### 10.4.2 实现聊天室客户

聊天室客户完成的功能如下。

- 建立和管理与服务器的 TCP 连接，通过继承 TcpCliThr 类来实现。
- 实现并发功能。通过继承 TcpCliThr 类来实现。
- 实现与服务器的收/发消息的功能。通过重载 TcpCliThr 类的 DealRecv()和 DealSend()函数来实现。

#### 1. 实现代码(chatcli.cpp)

其实现如下：

```
1 /* File : chatcli.cpp */
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <vector.h>
5 #include "pthread.h"
6 #include "Mysocket.h"
7 #include "MyThread.h"
8 #include "TcpCliThr.h"

9 const int MAX_MESSAGE_LEN = 1000;
10 const int DEFAULT_PORT = 1234;

11 /*****
12 class name : ChatClient
13 Function: A client used to chat
```

```
14 *****/
15 class ChatClient : public TcpCliThr
16 {
17 public:
18 ChatClient(int port,char* server) : TcpCliThr(port,server) {}
19 ~ChatClient(){WaitAllThr();}

20 void DealRecv(MyThread* thread);
21 void DealSend(MyThread* thread);
22 protected:
23 char* getMessage(char* buf,int len, FILE* fp);
24 };

25 void ChatClient::DealRecv(MyThread* thread)
26 {
27 char buf[MAX_MESSAGE_LEN + 1];
28 int socket = ((Receiver *)thread)->socket;

29 while(1) {
30 int len = recv(socket,buf,MAX_MESSAGE_LEN,0);
31 if (len < 1) break;
32 buf[len] = '\0';
33 printf("\n%s",buf);
34 }
35 close(socket);
36 }

37 void ChatClient::DealSend(MyThread* thread)
38 {
39 char buf[MAX_MESSAGE_LEN + 1];

40 int socket = ((Sender *)thread)->socket;
41 printf("Connected to server. \n");
42 /* send name to server */
43 printf("Input name : ");
44 if ( fgets(buf, MAX_MESSAGE_LEN, stdin) == NULL) {
45 printf("\nExit.\n");
46 close(socket);
47 return;
48 }
49 Send(buf,strlen(buf),0);

50 /* send message to server */
51 while (getMessage(buf, MAX_MESSAGE_LEN, stdin) != NULL) {
```

```

52     Send(buf, strlen(buf), 0);
53 }
54 printf("\nExit.\n");
55 close(socket);
56 }

57 char* ChatClient::getMessage(char* buf, int len, FILE* fp)
58 {
59     printf("Talk:");
60     fflush(stdout);
61     return(fgets(buf, MAX_MESSAGE_LEN, fp));
62 }

63 /*****
64 int main(int argc, char *argv[])
65 {
66     if (argc < 2) {
67         printf("Usage: %s <IP Address> <port>\n", argv[0]);
68         exit(1);
69     }
70     if (argc == 2) {
71         ChatClient chat(DEFAULT_PORT, argv[1]);
72         chat.ConnectServ();
73     }
74     if (argc == 3) {
75         ChatClient chat(atoi(argv[2]), argv[1]);
76         chat.ConnectServ();
77     }
78 }

```

## 2. 代码分析

9：定义常量 MAX\_MESSAGE\_LEN，最大的消息长度。

10：定义默认端口号。

25~36：实现 DealRecv()函数。反复从服务器接收数据，一旦收到则显示，如果出错就退出并关闭套接字。

37~56：实现 DealSend()函数。首先让用户输入名字并将其发送给服务器。然后，反复接收用户输入并将它们发往服务器。

57~62：实现 GetMessage()函数，用于获得用户输入。

64~78：主程序。根据命令行参数，生成 ChatClient 对象，然后执行与服务器的连接操作。

### 10.4.3 运行程序

下面在同一主机上运行聊天室服务器和客户程序。

(1) 首先启动服务器程序 (chat)。

(2) 然后同时运行客户 1、客户 2 和客户 3，名字分别为 client1、client2 和 client3。

#### 1. 服务器运行结果

```
$ chat
Accept Conn
Receiver is running!
Sender is running!
client1: 8
Accept Conn
Receiver is running!
Sender is running!
client2: 8
Accept Conn
Receiver is running!
Sender is running!
client3: 8
Recv:client1:I am clinet1.
Send:client1:I am clinet1.
Send:client1:I am clinet1.
Recv:client2:I am client2.
Send:client2:I am client2.
Send:client2:I am client2.
Recv:client3:I am client3.
Send:client3:I am client3.
Send:client3:I am client3.
Recv:client1:client1 talking.
Send:client1:client1 talking.
Send:client1:client1 talking.
Recv:client2:client2 talking.
Send:client2:client2 talking.
Send:client2:client2 talking.
Recv:client3:client3 talking.
Send:client3:client3 talking.
Send:client3:client3 talking.
Recv:client1:Bye.
Send:client1:Bye.
Send:client1:Bye.
Recv:client2:See you.
Send:client2:See you.
Send:client2:See you.
^C$
```



## 2. 客户 1 运行结果

```
$ chatcli 127.0.0.1
Accept Conn
Connected to server.
Input name : client1
Talk:I am clinet1.
Talk:
client2:I am client2.

client3:I am client3.
client1 talking.
Talk:
client2:client2 talking.

client3:client3 talking.
Bye.
Talk:^D
Exit.
$
```

## 3. 客户 2 运行结果

```
$ chatcli 127.0.0.1
Accept Conn
Connected to server.
Input name : client2
Talk:
client1:I am clinet1.
I am client2.
Talk:
client3:I am client3.

client1:client1 talking.
client2 talking.
Talk:
client3:client3 talking.

client1:Bye.
See you.
Talk:^D
Exit.
$
```

#### 4. 客户 3 运行结果

```
$ chatcli 127.0.0.1
Accept Conn
Connected to server.
Input name : client3
Talk:
client1:I am client1.

client2:I am client2.
I am client3.
Talk:
client1:client1 talking.

client2:client2 talking.
client3 talking.
Talk:
client1:Bye.

client2:See you.
^D
Exit.
$
```

#### 5. 运行结果说明

该实例是简单的聊天室应用系统，读者可对其进行扩充使其功能更加完备。

## 10.5 小 结

创建套接字类库能极大地提高开发效率，易于扩展和使用。库分为静态链接库和动态链接库。静态链接库使用简单，它在连接时已将运行程序所需代码插入到程序中，不存在系统配置问题。而动态链接库是在程序运行时由操作系统找到相应库文件，并将所需代码与运行程序相链接。当多个运行程序可共享相同的代码时，动态链接库能有效地利用内存空间。但动态链接库存在系统配置问题，以便系统能找到相应的库文件。

要建立套接字类库，首先应对相应的系统调用进行封装形成 Wrapper 类。然后根据实际的应用领域抽象出一些共用类。在此基础上，根据具体的项目形成应用类。最后通过应用类构成相应的应用系统。

目前，有大量的商用套接字类库销售，这样能大大缩短开发周期，同时可靠性和效率也得到有效的保证。

# 第四部分 高级网络编程技术

## 第 11 章 守护进程

守护进程是一种在后台运行的进程，它独立于所有的终端。在 Unix 系统中，有许多服务器是采用这种方式运行的，如 Web 服务器、邮件服务器等。

对于普通进程，由于它们与终端相连，当终端被关闭后，它们将被迫退出。而网络服务器通常是长期运行的，服务于网络客户，不能随终端的关闭而退出。因此，网络服务器都是以守护进程方式运行的。

由于守护进程独立于终端，无法从标准输入中获得用户输入，也无法将消息输出到标准输出以及标准错误输出上，因此需要特别的机制解决上述问题。

本章将讲述守护进程的创建、消息输出以及配置方法。

### 11.1 输出守护进程消息

#### 11.1.1 syslogd 进程

Unix 系统提供了 syslogd 来解决守护进程的消息输出问题。syslogd 是一种守护进程，它随着系统的启动而启动，并一直运行，直到系统退出。

syslogd 通过读取/etc/syslog.conf 文件来配置，以决定如何处理收到的消息，通常有以下几种方式。

- 将消息添加到某一文件尾部。
- 将消息输出到控制台。
- 将消息发送给某一登录的用户，如管理员。
- 将消息发送给其他主机上的 syslogd。

以下是一个 syslog.conf 的例子：

```
# syslog configuration file.
#
# This file is processed by m4 so be careful to quote ( ' ' ) names
# that match m4 reserved words. Also, within ifdef's, arguments
# containing commas must be quoted.
#
*.err;kern.notice;auth.notice          /dev/sysmsg
*.err;kern.debug;daemon.notice;mail.crit /var/adm/messages
```

```
# My configure
local0.notice          /usr/prog/chatd.log
*.notice                /var/log/t.log

*.emerg                *
# if a non-loghost machine chooses to have authentication messages
# sent to the loghost machine, un-comment out the following line:
#auth.notice           ifdef('LOGHOST', /var/log/authlog, @loghost)

mail.debug              ifdef('LOGHOST', /var/log/syslog, @loghost)

#
# non-loghost machines will use the following lines to cause "user"
# log messages to be logged locally.
#
ifdef('LOGHOST', ,
user.err                /dev/sysmsg
user.err                /var/adm/messages
user.alert              `root, operator'
user.emerg              *
)
```

syslogd 在启动时将产生 3 个描述符：

- Unix 域套接字并绑定到路径/var/run/log 目录下。
- UDP 套接字并绑定到端口 514 上。
- 打开/dev/klog，该设备接收来自系统内核的错误消息。

syslogd 守候这 3 个描述符，直到任何一个可读。从可读的描述符读出消息后，根据配置文件决定如何发送这些消息。因此，守护进程为了输出消息，可以向 syslogd 产生的 Unix 域套接字或 UDP 套接字发送消息。再由 syslogd 将消息输出到特定位置。

### 11.1.2 syslog()函数

Unix 系统提供了 syslog()函数来解决守护进程的消息输出。它将消息发送给 syslogd。其函数原型如下：

```
#include <syslog.h>
void syslog(int priority, const char *message, .../* arguments */);
```

priority：消息的优先级。syslogd 根据它来决定将该消息发往何处。

message：指向消息。它是一个格式化的字符串，如 printf 中的格式化字符串，但它增加了一个%m 的参数，该参数由变量 error 代表的错误消息来替换。

消息包括两部分。一类分为如下 7 种。

- LOG\_EMERG；紧急情况。通常广播给所有用户。
- LOG\_ALERT：报警情况。应该被立即解决。

- LOG\_CRIT：致命错误。如硬件设备错误。
- LOG\_ERR：一般错误。
- LOG\_WARNING：警告。
- LOG\_NOTICE：通知。一般需要特别的处理。
- LOG\_INFO：普通消息。
- LOG\_DEBUG：调试消息。

另一类包括如下。

- LOG\_KERN：由系统内核产生的消息。不能由用户进程产生。
- LOG\_USER：由用户进程产生的消息。默认的设置。
- LOG\_MAIL：由邮件系统产生的消息。
- LOG\_DAEMON：由系统守护进程产生的消息。如 in.ftpd(1M)。
- LOG\_AUTH：由验证系统产生的消息。如 login(1)、su(1M)、getty(1M)。
- LOG\_LPR：由打印机产生的消息。如 lpr(1B)、lpc(1B)。
- LOG\_NEWS：由新闻系统产生的消息。
- LOG\_UUCP：由 UUCP 产生的消息。
- LOG\_CRON：由 cron/at 产生的消息。如 crontab(1)、at(1)、cron(1M)。
- LOG\_LOCAL0：由本地用户产生的消息。
- LOG\_LOCAL1：由本地用户产生的消息。
- LOG\_LOCAL2：由本地用户产生的消息。
- LOG\_LOCAL3：由本地用户产生的消息。
- LOG\_LOCAL4：由本地用户产生的消息。
- LOG\_LOCAL5：由本地用户产生的消息。
- LOG\_LOCAL6：由本地用户产生的消息。
- LOG\_LOCAL7：由本地用户产生的消息。

这两类是混合使用的，优先级是它们“或”的结果。例如：

```
syslog(LOG_NOTICE | LOG_LOCAL0, "LOG MESSAGE");
```

将以 LOG\_LOCAL0.LOG\_NOTICE 为条件来匹配配置文件的条目。如果配置文件 /etc/syslog.conf 存在以下条目：

```
local0.notice      /var/log/test.log
```

则消息将被添加到文件 /var/log/test.log 中。

进程通过调用 openlog() 函数连接到 syslogd 创建的套接字上。openlog() 函数原型如下：

```
#include <syslog.h>
void openlog(const char *ident, int logopt, int facility);
```

ident：指向消息的前缀，该前缀将由 syslogd 自动添加到消息前，通常为程序的名称。

options：消息处理的选项，包括如下。

- LOG\_CONS：如果不能将消息发送给 syslogd，则发给控制台。
- LOG\_NDELAY：立即打开套接字。

- LOG-PERROR：发消息给标准错误输出，同时发给 syslogd 守护进程。
- LOG-PID：将当前进程 ID 添加到消息中。

facility: syslog()函数默认的设置。

### 11.1.3 closelog()函数

进程可通过函数 closelog()来关闭与 syslogd 的连接。函数原型如下：

```
#include <syslog.h>
void closelog();
```

## 11.2 创建守护进程

### 11.2.1 守护进程的创建过程

守护进程的基本条件是在后台运行并与终端无关，它的创建步骤如下。

(1) 调用 fork()函数产生后台子进程。首先调用 fork()函数产生子进程，然后父进程退出。这时子进程将在后台运行。同时，子进程是属于父进程的进程组，从而保证了子进程不是所属组的组长进程。因此，子进程可以调用 setid()函数产生新的会话期。

(2) 调用 setid()函数产生新的会话期并失去控制终端。产生新的会话期后，进程成为新会话期的组长进程，以及新进程组的组长进程，同时，进程失去控制终端。

(3) 忽略 SIGHUP 信号并再次调用 fork()函数产生新子进程。子进程再次调用 fork()函数产生新的子进程后，退出运行，这就保证了新的进程不可能重新打开控制终端。由于会话期组长进程终止时，会向会话期中的其他进程发 SIGHUP 信号而造成其他进程终止，所以必须忽略该信号。

(4) 改变工作目录。通常将目录改变到根目录下，这样，它启动的目录也可以被卸掉。

(5) 关闭已打开的文件描述符，并打开一个空设备，把它复制到标准输出、标准错误上。这是由于守护进程的程序可能包括一些库或重用一些其他程序代码，库中可能存在向标准输出或标准错误输出的函数调用。因为关闭了所有描述符，这些函数将返回错误而可能造成程序终止。如果将标准输出和标准错误映射到空设置上，将不会产生任何影响。

(6) 调用 openlog()函数，建立与 syslogd 的连接。

### 11.2.2 创建守护进程的代码

创建守护进程的典型代码如下：

```
1 switch (fork()) {
2     case 0: break;
3     case -1: return -1;
4     default: _exit(0);           // exit the original process
```

```
5    }

6    if (setsid() < 0) return -1;

7    signal(SIGHUP,SIG_IGN); // ignore SIGHUP
8    switch (fork()) {
9        case 0: break;
10       case -1: return -1;
11       default: _exit(0);
12    }

13    chdir("/");

14    int fdlimit = sysconf(_SC_OPEN_MAX);
15    while (fd < fdlimit) {
16        close(fd);
17        fd++;
18    }
19    open("/dev/null",O_RDWR);
20    dup(0);
21    dup(0);
22    openlog(...);
```

代码说明如下。

1~5: 调用 fork()函数产生后台子进程。

6: 产生新的会话期并失去控制终端。

7: 忽略 SIGNUP 信号。

8~12: 产生新子进程。这保证新的进程不可能重新打开控制终端。

13: 改变工作目录。

14~ 18: 关闭已打开的文件描述符。

19~21: 打开一个空设备，复制它到标准输出、标准错误上。

22: 建立与 syslogd 的连接。

## 11.3 配置守护进程

由于守护进程没控制终端，因而无法通过用户输入来进行相应配置，在编程时通常采用以下几种方法配置守护进程。

- 配置文件：将所有的配置参数存入一个文件，守护进程可以自动从文件中读取配置信息。
- 环境变量：守护进程通过读取环境变量而获得配置信息。典型代码如下：

```
char *ptr;
```

```

while (所有相关的环境变量) {
    if ((ptr = getenv("PARAMETERn")) != NULL) {
        PAR_CONF1 = atoi(ptr);
    }
    ...
}

```

- 程序在创建守护进程之前,通过用户输入进行配置,但这只能是在程序启动时进行。对于长期运行的守护进程有很大的局限性。

## 11.4 守护进程实例

本实例是将上一章中的聊天室服务器改成采用守护进程方式运行的服务器。我们生成一个新的类 ChatdServer, 它从 ChatServer 派生而来, 并添加相应代码以产生守护进程。

### 1. 源程序(prg11\_1.cpp, 程序 11.1)

其源代码如下:

```

1 /* File : prg11_1.cpp */
2 #include <stdio.h>
3 #include <string.h>
4 #include <vector.h>
5 #include "pthread.h"
6 #include <syslog.h>
7 #include <sys/types.h>
8 #include <unistd.h>
9 #include <signal.h>
10 #include <sys/stat.h>
11 #include <errno.h>
12 #include <fcntl.h>

13 #include "Mysocket.h"
14 #include "MyThread.h"
15 #include "MySync.h"
16 #include "TcpServThr.h"
17 #include "Chat.h"

18 class ChatServerd : public ChatServer
19 {
20     int MsgPriority;
21     void closeall(int fd);

22 public:
23     ChatServerd(int port, char *hostname = NULL);

```



```
24 ChatServerd(int port, int max_conn, int maxusr, int len, char *hostname);

25 int daemon(char* name,int facility = 0);
26 void DealRecv(MyThread* thread);
27 void DealSend(MyThread* thread);
28 void SetPriority(int p) {MsgPriority = p;}
29 int GetPriority() {return MsgPriority;}
30 };

31 ChatServerd::ChatServerd(int port, char *hostname):
32 ChatServer(port,hostname)
33 {
34 MsgPriority = LOG_NOTICE|LOG_LOCAL0;
35 }

36 ChatServerd::ChatServerd(int port, int max_conn, int maxusr,
    int len, char *hostname):
37 ChatServer(port,max_conn, maxusr, len, hostname)
38 {
39 MsgPriority = LOG_NOTICE|LOG_LOCAL0;
40 }

41 void ChatServerd::DealRecv(MyThread* thread)
42 {
43 syslog(MsgPriority,"Receiver is running!\n");
44 char buf[MAX_PACKET_LEN];

45 int socket = ((Receiver*) thread)->socket;
46 int len = recv(socket,buf,MAX_NAME_LEN,0);

47 buf[len - 1] = ':';
48 buf[len] = '\0';
49 syslog(MsgPriority,"%s %d\n",buf,len);

50 while (1) {
51 int len1 = recv(socket,buf + len,MAX_PACKET_LEN,0);
52 if (len1 < 1) break;
53 buf[len1 + len] = '\0';
54 syslog(MsgPriority,"Recv:%s",buf);
55 msg->Add(socket, buf);
56 con->wakeAll();
57 }
58 DelThread(thread);
59 }
```

```
60 void ChatServerd::DealSend(MyThread* thread)
61 {
62     char buf[MAX_PACKET_LEN];
63     long maxsn;

64     maxsn = msg->GetSN() - 1;
65     int socket = ((Receiver*) thread)->socket;
66     syslog(MsgPriority,"Sender is running!\n");

67     while(1) {
68         con->wait();
69         int err = msg->Get(socket,&maxsn,buf);
70         if (err) continue;
71         syslog(MsgPriority,"Send:%s",buf);
72         Send(socket,buf,strlen(buf),0);
73     }
74     DelThread(thread);
75 }

76 int ChatServerd::daemon(char* name,int facility)
77 {
78     switch (fork())
79     {
80     case 0: break;
81     case -1: return -1;
82     default: _exit(0);          // exit the original process
83     }

84     if (setsid() < 0)
85         return -1;

86     signal(SIGHUP,SIG_IGN);    // ignore SIGHUP

87     switch (fork())
88     {
89     case 0: break;
90     case -1: return -1;
91     default: _exit(0);
92     }

93     chdir("/");

94     closeall(0);
95     open("/dev/null",O_RDWR);
96     dup(0);
```

```
97 dup(0);

98 openlog(name, LOG_PID, facility);

99 return 0;
100 }

101 void ChatServerd::closeall(int fd)
102 {
103     int fdlimit = sysconf(_SC_OPEN_MAX);
104     while (fd < fdlimit) {
105         if (fd != Mysocket) close(fd);
106         fd++;
107     }
108 }

109 /*****
110 int main(int argc, char** argv)
111 {
112     FILE* file = fopen("chatd.conf", "r");
113     if (file == NULL) {
114         perror("chatd.conf:");
115         exit(1);
116     }

117     char input[100];
118     int port = 1234;
119     int max_conn = 20;
120     int maxusr = 20;
121     int len = 20;
122     char *hostname = "127.0.0.1";

123     while(1) {
124         char* s = fgets(input, 100, file);
125         if (s == NULL) break;

126         if (!strncasecmp(input, "PORT", 4)) port = atoi(strrchr(input, ' '));
127         if (!strncasecmp(input, "MAX_USRS", 8)) maxusr =
            atoi(strrchr(input, ' '));
128         if (!strncasecmp(input, "QUELEN", 6)) len = atoi(strrchr(input, ' '));
129         if (!strncasecmp(input, "MAX_CONN", 8)) max_conn =
            atoi(strrchr(input, ' '));
130         if (!strncasecmp(input, "HOSTNAME", 8)) hostname =
            strrchr(input, ' ');
131     }
```

```
132 ChatServerd chatd(port, max_conn,maxusr, len, hostname);

133 chatd.Init();

134 if (chatd.daemon(argv[0]) < 0) {
135 perror("daemon");
136 exit(2);
137 }

138 chatd.Run();

139 return 0;
140 }
```

程序代码分析如下。

2~17：头文件。

20：ChatServerd 成员变量 MsgPriority，代表消息的优先级。

23~24：声明构造函数。

25：声明成员函数 daemon()，用于产生守护进程。

28：设置消息优先级。

29：获得消息优先级。

31~40：实现构造函数，首先调用父类的构造函数，然后初始化消息优先级。

41~59：实现接收处理函数。所有的显示信息均由 syslog()函数传送给 syslogd 处理，从而取代 printf()函数直接在标准输出上显示。

60~75：实现发送处理函数。所有的显示信息均由 syslog()函数传送给 syslogd 处理，从而取代 printf()函数直接在标准输出上显示。

78~83：产生后台子进程，父进程退出运行。

84~85：产生新会话期，子进程失去控制终端。

86：忽略 SIGHUP 信号。

87~92：产生新子进程，原子进程退出。

93：将工作目录改为根目录。

94：关闭文件描述符。

95：打开空设备。

96：复制空设备到标准输出。

97：复制空设备到标准错误。

98：建立与 syslogd 的连接。

101~108：关闭所有描述符，注意应保留套接字描述符。

112~116：以只读方式打开配置文件 chatd.conf。

117~122：初始化配置参数。

123~131：从文件中读取配置信息并赋值给配置参数。

132：产生 ChatServerd 对象。

133：初始化聊天室服务器。

134~137：产生守护进程。

138：运行聊天室服务器。

## 2. 运行程序

首先创建配置文件，配置文件 chatd.conf 的内容如下：

```
PORT          1234
MAX_USRS      10
QUELEN        10
MAX_CONN      10
HOSTNAME      zj
```

然后，运行服务器（输入 chatd）及客户 1（输入 chatcli 127.0.0.1）和客户 2（输入 chatcli 127.0.0.1）。运行完毕后，记录文件的内容如下：

```
Feb 15 16:49:38 zj chatd[454]: [ID 437348 local0.notice] Receiver is running!
Feb 15 16:49:38 zj chatd[454]: [ID 156853 local0.notice] Sender is running!
Feb 15 16:49:43 zj chatd[454]: [ID 748610 local0.notice] client1: 8
Feb 15 16:51:46 zj chatd[454]: [ID 437348 local0.notice] Receiver is running!
Feb 15 16:51:46 zj chatd[454]: [ID 156853 local0.notice] Sender is running!
Feb 15 16:51:51 zj chatd[454]: [ID 748610 local0.notice] client2: 8
Feb 15 16:51:55 zj chatd[454]: [ID 663271 local0.notice] Recv:client2:Hi!
Feb 15 16:51:55 zj chatd[454]: [ID 148997 local0.notice] Send:client2:Hi!
Feb 15 16:52:00 zj chatd[454]: [ID 663271 local0.notice] Recv:client1:
Feb 15 16:52:00 zj chatd[454]: [ID 148997 local0.notice] Send:client1:
Feb 15 16:52:14 zj chatd[454]: [ID 663271 local0.notice] Recv:client1:How
are you?
Feb 15 16:52:14 zj chatd[454]: [ID 148997 local0.notice] Send:client1:How
are you?
```

## 3. 运行结果说明

记录文件记录了所有的来自聊天室服务器发来的消息。由此看出，利用记录文件不仅能记录错误消息，还可以利用记录文件进行调试、诊断程序，也可以保存一些客户消息。

# 11.5 小 结

网络服务程序，可以在完成创建套接字、绑定套接字、设置套接字为监听模式后，变成守护进程，进入后台执行而不占用控制终端，这是网络服务程序的常用模式。

守护进程的实现是非常简单的，首先调用 fork() 函数，为避免挂起控制终端，将守护进程放入后台执行。然后调用 setsid() 函数脱离控制终端，登录会话和进程组，使该进程成为会话组长，与原来的登录会话和进程组脱离，进程同时与控制终端脱离。进程已经成为无

终端的会话组长。但它可以重新申请打开一个控制终端。可以通过使进程不再成为会话组长来禁止进程重新打开控制终端，这就需要第二次调用 `fork()` 函数，父进程（会话组长）退出，子进程继续执行，并不再拥有打开控制终端的能力。

守护进程通过 `syslog()` 函数，发消息给守护进程 `syslogd`，它根据消息优先级来决定将消息发往何处。

## 第 12 章 原始套接字

原始套接字是除 TCP 套接字与 UDP 套接字之外的另一较常用的套接字。常用于与路由相关的应用。与 TCP/UDP 套接字相比，它有如下 3 个主要特点。

- 原始套接字可以读/写 ICMP、IGMP 包。
- 原始套接字可以读/写 IP 包，只要这些 IP 包的协议域不是由系统内核处理。通常系统内核只处理 ICMP、IGMP、TCP 和 UDP 几种协议。其他协议则由用户进程通过读/写原始套接字来实现。
- 通过原始套接字，可以构造自己的 IP 包头。这样可以发送具有特殊 IP 头的 UDP 和 TCP 包。

本章讲述原始套接字的产生和读写方法，并通过实例予以说明。

### 12.1 产生原始套接字

原始套接字的产生与 TCP/UDP 套接字一样，使用 `socket()` 函数。但套接字类型为 `SOCK_RAW`，并且协议通常为非 0 值。典型代码如下：

```
int sockfd;  
sockfd = socket(AF_INET, SOCK_RAW, protocol);
```


其中，`protocol` 是一个常量定义，在 `<netinet/in.h>` 中定义如下：

```
#define IPPROTO_IP          0      /* dummy for IP */  
#define IPPROTO_HOPOPTS    0      /* Hop by hop header for IPv6 */  
#define IPPROTO_ICMP       1      /* control message protocol */  
#define IPPROTO_IGMP       2      /* group control protocol */  
#define IPPROTO_GGP        3      /* gateway^2 (deprecated) */  
#define IPPROTO_ENCAP      4      /* IP in IP encapsulation */  
#define IPPROTO_TCP        6      /* tcp */  
#define IPPROTO_EGP        8      /* exterior gateway protocol */  
#define IPPROTO_PUP        12     /* pup */  
#define IPPROTO_UDP        17     /* user datagram protocol */  
#define IPPROTO_IDP        22     /* xns idp */  
#define IPPROTO_IPV6       41     /* IPv6 encapsulated in IP */  
#define IPPROTO_ROUTING    43     /* Routing header for IPv6 */  
#define IPPROTO_FRAGMENT   44     /* Fragment header for IPv6 */  
#define IPPROTO_RSVP       46     /* rsvp */  
#define IPPROTO_ESP        50     /* IPsec Encap. Sec. Payload */
```

```

#define IPPROTO_AH          51      /* IPsec Authentication Hdr. */
#define IPPROTO_ICMPV6      58      /* ICMP for IPv6 */
#define IPPROTO_NONE        59      /* No next header for IPv6 */
#define IPPROTO_DSTOPTS     60      /* Destination options */
#define IPPROTO_HELLO       63      /* "hello" routing protocol */
#define IPPROTO_ND           77      /* UNOFFICIAL net disk proto */
#define IPPROTO_EON          80      /* ISO clnp */
#define IPPROTO_PIM          103     /* PIM routing protocol */
#define IPPROTO_RAW          255     /* raw IP packet */
#define IPPROTO_MAX          256

```

 **注意** :只有超级用户才有权创建原始套接字,否则函数将返回-1,并置 `errno` 为 `EACCES`。

在默认情况下,不能构造自己的 IP 头,除非设置套接字选项,典型代码如下:

```

int sockfd;
const int on = 1;
...
if (setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
    // 错误处理
    ...
}
...

```

## 12.2 写原始套接字

原始套接字的写操作通常使用 `sendto()`和 `sendmsg()`函数。

在默认情况下,写入的数据将由系统内核填入 IP 包的数据域,而 IP 头由系统内核自动产生。如果设置了 IP 套接字选项,则写入的数据将从 IP 头的第 1 个字节开始填充。系统内核只负责填充 IP 头的校验和。如果 IP 的标识设置为 0,则该项由系统来填充。

当包的长度大于 MTU (Maximum Transmission Unit, 最大传输单位) 时,系统会自动进行拆包,这对用户进程是透明的。

## 12.3 读原始套接字

原始套接字的读操作通常使用 `recvfrom()` 和 `recvmsg()`函数。原始套接字无法接收 TCP/UDP 数据包,它只能接收如下数据包。

- ICMP 包。
- IGMP 包。
- 协议域不被系统内核理解的 IP 包。对于这些 IP 包,系统内核仅校验其 IP 版本、



头校验、头长度及目的地址等。

当系统内核收到需要传递给原始套接字的 IP 包后,要检查所有进程产生的原始套接字,然后将 IP 包拷贝给所有匹配的原始套接字。系统采用以下原则进行匹配。

- 如果 IP 包的协议域是非 0 值,则只有原始套接字的协议域与之完全匹配,才将 IP 包传递给该原始套接字。
- 如果原始套接字绑定到本地 IP 地址上,则 IP 包的目的地址必须与套接字绑定的地址匹配,否则,不传递 IP 包给该原始套接字。
- 如果原始套接字通过 connect()函数与远程 IP 地址连接,则 IP 包的源地址必须与该远程 IP 地址匹配,否则系统不会将 IP 包传递给该原始套接字。
- 如果原始套接字的协议域为 0,并且没有绑定或连接到任何 IP 地址上,则该原始套接字将接收所有发送给原始套接字的 IP 包。

## 12.4 原始套接字实例

以下通过一个简单的 ping 程序(程序 12.1)来说明原始套接字的使用。ping 程序原理非常简单:ping 程序发 ICMP 响应请求给某一主机,该主机返回一个 ICMP 响应应答,程序收到应答则显示结果。为完成这个功能,首先应创建协议为 ICMP 的原始套接字以接收/发送 ICMP 包,然后需要构造 ICMP 包,通过原始套接字发送给对方主机。ICMP 包的结构在第 2 章中已讲述了,它所对应的数据结构定义如下:

```
struct icmp {
    uchar_t icmp_type;        /* type of message, see below */
    uchar_t icmp_code;        /* type sub code */
    ushort_t icmp_cksum;      /* ones complement cksum of struct */
    union {
        uchar_t ih_pptr;      /* ICMP_PARAMPROB */
        struct in_addr ih_gwaddr; /* ICMP_REDIRECT */
        struct ih_idseq {
            n_short icd_id;
            n_short icd_seq;
        } ih_idseq;
        int ih_void;

        /* ICMP_UNREACH_NEEDFRAG -- Path MTU Discovery (RFC1191) */
        struct ih_pmtu {
            n_short ipm_void;
            n_short ipm_nextmtu;
        } ih_pmtu;

        struct ih_rtradv {
            uchar_t irt_num_addrs;
        } ih_rtradv;
    }
};
```

```

        uchar_t irt_wpa;
        n_short irt_lifetime;
    } ih_rtradv;
} icmp_hun;
#define icmp_pptr      icmp_hun.ih_pptr
#define icmp_gwaddr    icmp_hun.ih_gwaddr
#define icmp_id        icmp_hun.ih_idseq.icd_id
#define icmp_seq       icmp_hun.ih_idseq.icd_seq
#define icmp_void      icmp_hun.ih_void
#define icmp_pmvoid     icmp_hun.ih_pmtu.ipm_void
#define icmp_nextmtu   icmp_hun.ih_pmtu.ipm_nextmtu
union {
    struct id_ts {
        n_time its_otime;
        n_time its_rtime;
        n_time its_ttime;
    } id_ts;
    struct id_ip {
        struct ip idi_ip;
        /* options and then 64 bits of data */
    } id_ip;
    ulong_t id_mask;
    char    id_data[1];
} icmp_dun;
#define icmp_otime    icmp_dun.id_ts.its_otime
#define icmp_rtime    icmp_dun.id_ts.its_rtime
#define icmp_ttime    icmp_dun.id_ts.its_ttime
#define icmp_ip       icmp_dun.id_ip.idi_ip
#define icmp_mask     icmp_dun.id_mask
#define icmp_data     icmp_dun.id_data
};

```

其内部结构虽然较为复杂，但其实质是对应 ICMP 包的如下七个域：

- 类型域(icmp\_type) :ICMP\_ECHO( ICMP 响应请求 )或 ICMP\_ECHOREPLY( ICMP 响应应答 )
- 代码域(icmp\_code)
- 校验和域(icmp\_cksum)
- 标识域(icmp\_id)
- 序列号域(icmp\_seq)
- 可选数据域(icmp\_data)

为产生 ICMP 响应请求包，ping 程序构造的 ICMP 包的类型域为 ICMP\_ECHO，其校验由专门的算法产生，标识设置为当前的进程 ID，序列号从 0 开始，每发一个包后加 1，可选数据为字符串数据。

对方主机收到 ICMP 响应请求包后，仅将 ICMP 包的类型改为 ICMP\_ECHOREPLY 后

发回。

程序完成发送,就开始接收 ICMP 响应应答包。首先应判断类型,因为所创建的 ICMP 原始套接字不仅接收类型为 ICMP\_ECHOREPLY 的包也同时接收类型为 ICMP\_ECHO 的 ICMP 包。再判断标识是否为本进程的进程 ID,如果是则说明该包为相应的响应包。

### 1. 源程序及代码分析

程序的源程序如下:

```
1 #include <iostream.h>
2 #include <string.h>
3 #include <errno.h>
4 #include <signal.h>
5 #include <stdlib.h>
6 #include <arpa/inet.h> /* inet_ntoa */
7 #include <unistd.h>    /* close */
8 #include <stdio.h>     /* Basic I/O routines */
9 #include <sys/types.h> /* standard system types */
10 #include <netinet/in.h> /* Internet address structures */
11 #include <sys/socket.h> /* socket interface functions */
12 #include <netdb.h>      /* host to IP resolution */
13 #include <netinet/in_systm.h>
14 #include <netinet/ip.h> /* ICMP */
15 #include <netinet/ip_icmp.h> /* ICMP */

16 #define ICMPHEAD 8 // ICMP packet header's length
17 #define MAXICMPLEN 200

18 /*****
19 class: RawSock
20 *****/
21 class RawSock
22 {
23 public:
24 int sock;
25 int error;
26 RawSock(int protocol =0);
27 virtual ~RawSock();
28 int send(const void* msg, int msglen,sockaddr* addr,unsigned int len);
29 int send(const void* msg, int msglen,char* addr);
30 int receive(void* buf,int buflen,sockaddr* from,int* len);
31 int Error() {return error;}
32 };

33 class ICMP: public RawSock
34 {
```

```
35 public:
36 struct icmp *packet;
37 int max_len;
38 int length;

39 ushort_t checksum(ushort_t *addr,int len);
40 ICMP();
41 ICMP(int len);
42 ~ICMP();
43 int send_icmp(char *to, void* buf,int len);
44 int recv_icmp(sockaddr* from);
45 void setCode(int c) { packet->icmp_code = c;}
46 void setId(int i) { packet->icmp_id = i; }
47 void setSeq(int s) { packet->icmp_seq = s;}
48 void setType(int t) { packet->icmp_type = t;}
49 };

50 RawSock::RawSock(int protocol = 0) {
51 sock = socket(AF_INET, SOCK_RAW,protocol);
52 setuid(getuid());
53 if (sock == -1) error= 1;
54 else error = 0;
55 }

56 RawSock::~RawSock(){
57 close(sock);
58 }

59 int RawSock::send(const void* msg,int msglen,sockaddr* to,
        unsigned int len) {
60 if (error) return -1;
61 int length = sendto(sock,msg,msglen,0,(const sockaddr*)to,len);
62 if (length == -1) {
63 error = 2;
64 return -1;
65 }
66 return length;
67 }

68 int RawSock::send(const void* msg,int msglen,char* hostname) {
69 sockaddr_in sin;          // Sock Internet address
70 if (error) return -1;
71 if(hostname) {
72 hostent *hostnm = gethostbyname(hostname);
73 if(hostnm == (struct hostent *) 0) {
```

```
74     return -1;
75 }
76 sin.sin_addr.s_addr = *((unsigned long *)hostnm->h_addr);
77 }
78 else
79     return -1;
80 sin.sin_family = AF_INET;

81 return send(msg,msglen,(sockaddr *)&sin, sizeof(sin));
82 }

83 int RawSock::receive(void* buf,int buflen,sockaddr* from,int* len)
84 {
85     if (error) return -1;
86     while (1) {
87         int length = recvfrom(sock,buf,buflen,0,from,len);
88         if (length == -1)
89             if (errno == EINTR) continue;
90         else {
91             error = 3;
92             return -1;
93         }
94         return length;
95     }
96 }

97 /*****
98 class: ICMP
99 *****/
100 ICMP::ICMP(): RawSock(IPPROTO_ICMP)
101 {
102     max_len = MAXICMPLEN;
103     packet = (icmp*) new char[max_len];

104     packet->icmp_code = 0;
105     packet->icmp_id = 0;
106     packet->icmp_seq = 0;
107     packet->icmp_type = ICMP_ECHO;
108 }

109 ICMP::ICMP(int len): RawSock(IPPROTO_ICMP)
110 {
111     max_len = len;
112     packet = (icmp*) new char[max_len];
```

```
113 packet->icmp_code = 0;
114 packet->icmp_id = 0;
115 packet->icmp_seq = 0;
116 packet->icmp_type = ICMP_ECHO;
117 }

118 ICMP::~ICMP()
119 {
120     delete[] (char*) packet;
121 }

122 ushort_t ICMP::checksum(ushort_t *addr,int len)
123 {
124     int nleft = len;
125     int sum = 0;
126     unsigned short *w = addr;
127     unsigned short answer = 0;

128     while (nleft > 1) {
129         sum+=*w++;
130         nleft -= 2;
131     }

132     if (nleft == 1) {
133         *(unsigned char*) (&answer) = *(unsigned char*) w;
134         sum += answer;
135     }

136     sum = (sum >> 16) + (sum & 0xffff);
137     sum += (sum>>16);
138     answer = ~sum;
139     return (answer);
140 }

141 int ICMP::send_icmp(char *host, void* buf,int len)
142 {
143     memcpy(packet->icmp_data,buf,len);
144     packet->icmp_cksum = 0;
145     packet->icmp_cksum = checksum((unsigned short *)packet, ICMPHEAD + 6);

146     int err = send(packet,MAXICMPLEN,host);
147     return err;
148 }

149 int ICMP::recv_icmp(sockaddr* from)
```

```
150 {
151 char buf[MAXICMPLEN + 100];
152 int hlen1, icmplen;
153 struct ip *ip;
154 struct icmp *icmp;

155 if (Error()) return -1;
156 int addrlen = 0;
157 int len = receive(buf,MAXICMPLEN+100,from,&addrlen);

158 if (len == -1) {
159     cout << "Receiving Failed!\n" ;
160     return -1;
161 }

162 ip = (struct ip *) buf;    /* start of IP header */
163 hlen1 = ip->ip_hl << 2;    /* length of IP header */

164 icmp = (struct icmp *) (buf + hlen1); /* start of ICMP header */
165 if ( (icmplen = len - hlen1) < 8){
166     cout << "Receiving Failed!\n" ;
167     return -1;
168 }

169 length = len - hlen1;
170 memcpy(packet,icmp,length);
171 return 0;
172 }

173 main(int argc, char *argv[])
174 {
175 ICMP icmp;
176 sockaddr from;
177 char *host;
178 int count;

179 if (argc < 2) {
180     printf("Usage: %s <IP Address> <try_number>\n",argv[0]);
181     exit(1);
182 }
183 if (argc == 2) {
184     host = argv[1];
185     count = 5;
186 }
187 if (argc == 3) {
```

```

188  host = argv[1];
189  count = atoi(argv[2]);
190  }

191  for (int i = 0; i <= count; i++) {
192      icmp.setId(getpid());
193      icmp.setSeq(i);
194      char* test_data= "abcde";
195      icmp.send_icmp(host,test_data,strlen(test_data));
196  }
197  int num = 1;
198  while(1) {
199      if (icmp.recv_icmp(&from) < 0) continue;
200      if (icmp.packet->icmp_type == ICMP_ECHOREPLY) {
201          if (icmp.packet->icmp_id == getpid()) {
202              printf("%d bytes from %s: seq=%u, data=%s\n",
203                  icmp.length, host,icmp.packet->icmp_seq,
204                  icmp.packet->icmp_data);
205              num ++;
206              if (num > count) break;
207          }
208      }
209  }

```

代码说明如下。

16：定义 ICMP 头的长度

17：定义最大 ICMP 包的长度

18~32：定义 RawSock 类。成员变量 sock 存放原始套接字，error 存放错误代码。成员函数包括：

- send() 通过原始套接字发数据；
- receive() 通过原始套接字收数据；
- Error() 返回错误码。

33~49：定义 ICMP 类。成员变量 packet 指向 ICMP 包，length 表示接收到 ICMP 包长度。定义的成员函数包括：

- send\_icmp() 发送 ICMP 响应请求包；
- recv\_icmp() 接收 ICMP 响应应答包；
- setCode() 设置代码；
- setSeq() 设置序列号；
- setId() 设置标识；
- setType() 设置 ICMP 类型；
- check\_sum() 计算 ICMP 校验和。

50~55：实现 RawSock 类构造函数。创建原始套接字，并将超级用户还原成当前用户。



这是出于安全方面的考虑，尽量不要在超级用户权限下运行程序。

59~82：实现 RawSock 类的 send()函数。

83~96：实现 RawSock 类的 receive()函数。receive()是慢系统调用，应防止被信号所中断。

100~117：实现 ICMP 类构造函数。

122~140：实现 ICMP 类的 check\_sum()函数。

141~148：实现 ICMP 类的 send\_icmp()函数。

149~172：实现 ICMP 类的 recv\_icmp()函数。

155~161：从原始套接字接收 IP 包。

162~163：获得 IP 包头长度。

164~168：从 IP 包中抽取 ICMP 包，变量 icmp 指向 ICMP 包头。

189~170：拷贝 ICMP 包给 packet。

191~196：循环发送 ICMP\_ECHO 包。

197~208：循环接收 ICMP\_ECHOREPLY 包，并检查其标识，如果是本进程的进程 ID，则显示 ICMP 包的信息。

## 2. 运行程序

在超级用户权限下运行上述程序查看本机 IP 连接情况。输入“ping1 127.0.0.1”。

## 3. 服务器运行结果

```
# ping1 127.0.0.1
200 bytes from 127.0.0.1: seq=0, data=abcde
200 bytes from 127.0.0.1: seq=1, data=abcde
200 bytes from 127.0.0.1: seq=2, data=abcde
200 bytes from 127.0.0.1: seq=3, data=abcde
200 bytes from 127.0.0.1: seq=4, data=abcde
#
```

## 4. 运行结果说明

ping1 利用原始套接字可收到相应的 ICMP\_ECHOREPLY 包。

# 12.5 小 结

原始套接字的使用非常简单，首先使用 socket()函数创建原始套接字，然后就可以利用 sendto()、sendmsg()、recvfrom()、recvmsg()等函数进行读写。

由于系统内核对从原始套接字收/发的 IP 包做很少的协议处理，因此，应用程序需要考虑协议方面的问题，尤其是相应的包结构。

另外，原始套接字必须在超级用户权限下才能创建。它只能读取 ICMP、IGMP 以及协议域不被系统理解的 IP 包。如果需要发送具有特殊 IP 头的包，应设置相应的套接字选项。

## 第 13 章 数据链路访问

前面的章节讲述了利用套接字来读/写网络数据，但是，并不是所有网络数据都能通过上述方法获取，即使是原始套接字所接收的 IP 包也是有限制的。对于某些网络应用，尤其是网络安全及路由方面的应用，必须获取所有网络数据，至少是所有 IP 数据包。另外，这些特殊的应用不影响其他的网络应用，即既不影响其他网络应用的 IP 包收/发，同时又能够获得这些包。为解决上述问题，必须访问数据链路层，直接从数据链路层获取数据。

目前主要的数据链路访问方法包括：

- BSD 包过滤器（BPF）
- SVR4 数据链路供应接口（DLPI）
- Linux 套接字包接口（SOCK\_PACKET）
- libpcap（数据包捕获函数库）

本章介绍这几种数据链路访问方法，并重点讲述 libpcap 的使用方法及实例。

### 13.1 数据链路访问方法

#### 13.1.1 BSD 包过滤器

BSD 系统提供 BPF 来访问数据链路。BPF 具有很强的包过滤能力。每个应用可以打开 BPF 设备，并导入自己的过滤器，再由 BPF 将这些过滤器应用于每个被捕获的包，过滤后的包再通过缓冲区传递给应用程序。

BPF 过滤器在系统内核内，只有过滤后的包才被拷贝给用户程序，这样能大大减少被拷贝的包，从而提高效率。程序可根据需要仅拷贝包的一部分。另外，BPF 仅当其缓冲区满或超时才将缓冲区内拷贝给应用。

#### 13.1.2 DLPI

SVR4 系统提供 DLPI 来访问数据链路，它是一个协议独立的接口。其方法与 BPF 类似，但它的过滤机制与 BPF 不同，与 BPF 相比，速度较慢。

#### 13.1.3 SOCK\_PACKET

Linux 系统提供 SOCK\_PACKET 来访问数据链路。与套接字创建相同，也使用了 socket() 函数。套接字类型为 SOCK\_PACKET，套接字协议用来指明所要捕获包的类型，如以太网帧或 IPv4 包：

```
Sockfd = socket(AF_INET, SOCK_PACKET, htons(ETH_P_ALL));  
Sockfd = socket(AF_INET, SOCK_PACKET, htons(ETH_P_IP));
```

SOCK\_PACKET 没有系统内核缓冲区及系统内核过滤器，因此效率较低，对于监视高速网络存在问题。

### 13.1.4 libpcap

libpcap 是一个独立于操作系统执行的访问数据链路的方法。它是一个数据包捕获函数库，该库提供的 C 函数接口可用于捕获所有经过网络接口的数据包。

由于它独立于操作系统执行，具有良好的兼容性，目前被广泛应用。

## 13.2 libpcap 应用

### 13.2.1 libpcap 库函数

libpcap 的使用是通过调用相应的库函数来实现的。其主要的库函数如下。

#### 1. pcap\_open\_live()函数

pcap\_open\_live()函数获得用于捕获网络数据包的数据包捕获描述符。其函数原型如下：

```
#include <pcap.h>  
pcap_t *pcap_open_live(char *device, int snaplen, int promisc,  
                        int to_ms, char *ebuf);
```

device: 指定打开的网络设备名。

snaplen: 定义捕获数据的最大字节数。

promisc: 指定是否将网络接口置于混杂模式。

to\_ms: 指定超时时间（毫秒）。

返回值: 出错返回 NULL，同时在 errbuf 中存放相关的错误消息。

#### 2. pcap\_lookupdev()函数

pcap\_lookupdev()函数用于返回可被 pcap\_open\_live()或 pcap\_lookupnet()函数调用的网络设备名指针。其函数原型如下：

```
#include <pcap.h>  
char *pcap_lookupdev(char *errbuf)
```

返回值: 出错返回 NULL，同时在 errbuf 中存放相关的错误消息。

#### 3. pcap\_lookupnet()函数

pcap\_lookupnet()函数获得指定网络设备的网络号和掩码。其函数原型如下：

```
#include <pcap.h>  
int pcap_lookupnet(char *device, bpf_u_int32 *netp, bpf_u_int32 *maskp,  
                  char *errbuf)
```

device: 使用的网络设备名。

netp: 网络设备的网络号。

maskp: 网络设备的掩码。

返回值: 出错返回-1, 同时在 errbuf 中存放相关的错误消息。

#### 4. pcap\_dispatch()函数

pcap\_dispatch()函数用于捕获并处理数据包。其函数原型如下:

```
#include <pcap.h>
int pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
```

p: 打开的网络设备描述符。

cnt: 指定函数返回前所处理数据包的最大值。cnt=-1 表示在一个缓冲区中处理所有的数据包。cnt=0 表示处理所有数据包, 直到产生以下错误之一。

- 读取到 EOF;
- 超时读取。

callback: 指定一个带有三个参数的回调函数。这三个参数分别是一个从 pcap\_dispatch() 函数传递过来的 u\_char 指针, 一个 pcap\_pkthdr 结构的指针, 以及一个数据包大小的 u\_char 指针。

user: 传递给回调函数的参数。

返回值: 如果成功则返回读取到的字节数。读取到 EOF 时则返回零值。出错时则返回 -1, 此时可调用 pcap\_perror()或 pcap\_geterr()函数获取错误消息。

#### 5. pcap\_loop()函数

pcap\_loop()函数的功能基本与 pcap\_dispatch()函数相同, 只不过此函数在 cnt 个数据包被处理或出现错误时才返回, 但读取超时不会返回。但是, 如果为 pcap\_open\_live()函数指定了一个非零值的超时设置 然后调用 pcap\_dispatch()函数 则当超时发生时 pcap\_dispatch()函数会返回。cnt 参数为负时 pcap\_loop()函数将始终循环运行, 直至出现错误。其函数原型如下:

```
#include <pcap.h>
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
```

#### 6. pcap\_compile()函数

pcap\_compile()函数用于将指定的字符串编译到过滤程序中。其函数原型如下:

```
#include <pcap.h>
int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize,
                 bpf_u_int32 netmask)
```

p: 打开的网络设备描述符。

fp: 一个 bpf\_program 结构的指针, 在 pcap\_compile()函数中被赋值。

str: 指定编译到过滤程序中的字符串。

optimize: 控制结果代码的优化。

netmask: 指定本地网络的网络掩码。

返回值: 出错时返回-1; 成功时返回 0。

### 7 . pcap\_setfilter()函数

pcap\_setfilter()函数用于指定一个过滤程序。其函数原型如下:

```
#include <pcap.h>
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

p: 打开的网络设备描述符。

fp: bpf\_program 结构指针, 通常在 pcap\_compile()函数调用中被赋值。

返回值: 出错时返回-1; 成功时返回 0。

### 8 . pcap\_next()函数

pcap\_next()函数用于返回指向下一个数据包的指针。其函数原型如下:

```
#include <pcap.h>
u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
```

p: 打开的网络设备描述符。

h: 一个指向 pcap\_pkthdr 结构的指针。

返回值: 指向下一个数据包的指针。出错时返回 NULL。

### 9 . pcap\_datalink()函数

pcap\_datalink()函数用于返回数据链路层类型。其函数原型如下:

```
#include <pcap.h>
int pcap_datalink(pcap_t *p)
```

p: 打开的网络设备描述符。

返回值: 数据链路层类型, 例如 DLT\_EN10MB 或 DLT\_PPP。

### 10 . pcap\_close()函数

pcap\_close()函数用于关闭打开的网络设备描述符, 并释放资源。其函数原型如下:

```
#include <pcap.h>
void pcap_close(pcap_t *p)
```

p: 打开的网络设备描述符。

### 11 . pcap\_geterr()函数

pcap\_geterr()函数用于返回最后一个 pcap 库错误消息。其函数原型如下:

```
#include <pcap.h>
char *pcap_geterr(pcap_t *p)
```

p: 打开的网络设备描述符。

返回值: 最后一个 pcap 库错误消息。

### 13.2.2 libpcap 数据结构

上述函数调用涉及到如下两个重要的数据结构。

#### 1. pcap\_t 结构

网络设备描述符是一个 pcap\_t 类型的结构。其结构定义如下：

```
typedef struct pcap pcap_t;
struct pcap
{
    int          fd;
    int          snapshot;
    int          linktype;
    int          tzoff;    /* timezone offset */
    int          offset;   /* offset for proper alignment*/
    struct pcap_sf sf;
    struct pcap_md md;
    int          bufsize; /* Read buffer */
    u_char *     buffer;
    u_char *     bp;
    int          cc;
    /* Place holder for pcap_next() */
    u_char *     pkt;
    /* Placeholder for filter code if bpf not in kernel. */
    struct bpf_program fcode;
    char          errbuf[PCAP_ERRBUF_SIZE];
};
```

#### 2. pcap\_pkthdr 结构

pcap\_pkthdr 结构位于真正的物理帧前面，用于消除不同链路层支持的差异。常用于获得捕获包的时间戳及长度。其结构定义如下：

```
struct pcap_pkthdr
{
    struct timeval ts;    /* time stamp */
    bpf_u_int32  caplen; /* length of portion present */
    bpf_u_int32  len;    /* length this packet (off wire) */
};
```

### 13.2.3 过滤程序

在实际的应用中，不需要读取所有的包，通常只关心某一类型的包。例如，在某一子网内传输的包或传输到某一地址的包。当然，应用程序可以读取所有的包，然后再进行筛选，但这样将消耗大量的资源。所幸的是，libpcap 提供了内部过滤机制，不但简化了应用

程序，还提高了效率。

libpcap 提供了过滤程序，但应用需要提供相应的过滤规则，以决定哪些包才能被捕获。过滤规则通过 pcap\_compiler()函数编译到过滤程序中，再通过 pcap\_setfilter()函数将其设置为过滤器，从而对包进行内部过滤。当然，也可以编制自己的过滤程序，而不必通过 pcap\_compiler()函数进行编译。

过滤规则是以字符串形式表达的，可以包括如下三种基本规则的一个或多个。

- 类型限定：类型包括 host、net 和 port。例如，host myhost 表示捕捉带有主机地址 myhost 的包；net 201.9.200 表示捕捉子网 201.9.200 内的包；port 20 表示捕捉端口号为 20 的包。
- 方向限定：指明何种方向的包被捕获。方向包括 src、dst、src or dst 和 src and dst。例如，src myhost 表示捕捉源地址为 myhost 的包。
- 协议限定：指明捕捉何种协议的包。协议包括 ether、fddi、ip、arp、rarp、tcp、udp 等。例如，tcp port 80 表示捕捉端口号为 80 的 TCP 包。

过滤规则非常灵活，也很复杂，详细的介绍可查阅 tcpdump 参考手册。

## 13.3 数据链路访问实例

### 1. 实例说明

以下实例是通过 libpcap 函数库访问数据链路层，从而捕获所需的包。它完成的功能如下。

- 打开网络设备。
- 根据用户从命令行输入的过滤规则产生过滤器。
- 捕获所需的包并显示。

其基本流程见图 13.1。

### 2. 源程序（程序 13.1）及代码分析

源程序代码如下：

```
1 #include <pcap.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <errno.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7 #include <arpa/inet.h>

8 #define BUFSIZ 1000
9 #define PCAP_ERRBUF_SIZE 200
10 void Display(const u_char * packet, const size_t length);

11 /* list all packets be filtered */
```

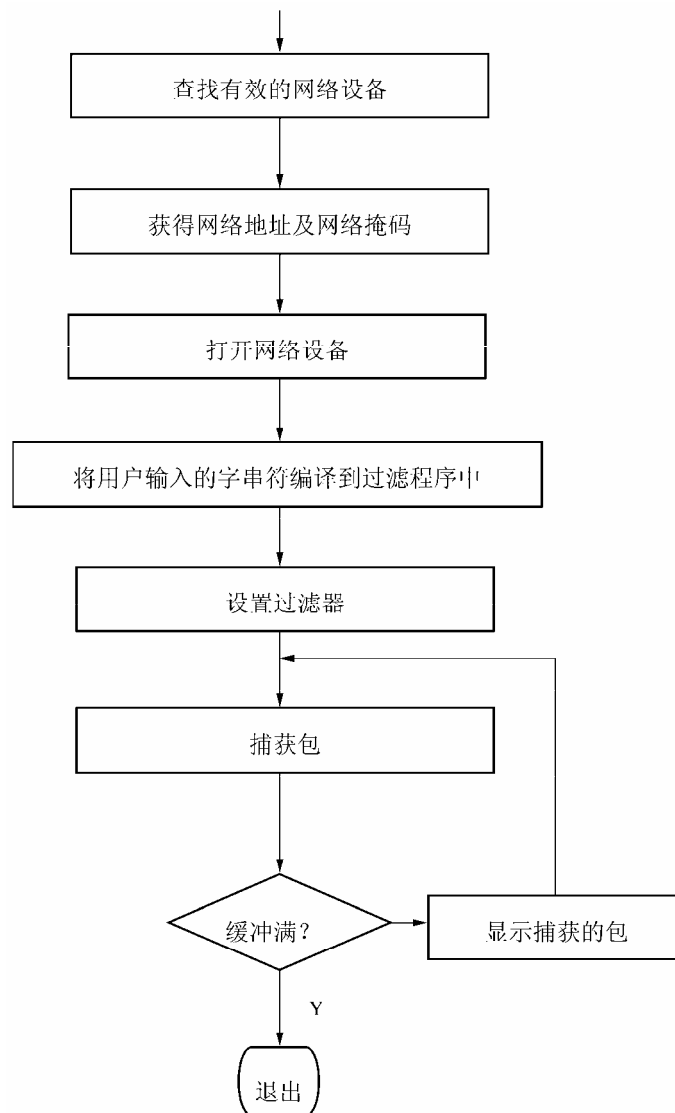


图 13.1 libpcap 数据链路的基本流程

```
12 void my_callback (u_char * none, const struct pcap_pkthdr * pkthdr,  
    const u_char * packet)  
13 {  
14     Display((u_char * )packet, (size_t)(pkthdr->caplen));  
15     return;  
16 }  
  
17 int main(int argc, char **argv)  
18 {  
19     int i;  
20     char *dev;
```



```
21 char errbuf[PCAP_ERRBUF_SIZE];
22 pcap_t* descr;
23 const u_char *packet;
24 struct pcap_pkthdr hdr;
25 struct ether_header *eptr;
26 struct bpf_program fp;
27 bpf_u_int32 maskp;
28 bpf_u_int32 netp;

29 if(argc != 2){
30     fprintf(stdout,"Usage: %s \"filter program\"\\n" ,argv[0]);
31     return 0;
32 }

33 /* grab a device */
34 dev = pcap_lookupdev(errbuf);
35 if(dev == NULL) {
36     fprintf(stderr,"%s\\n",errbuf);
37     exit(1);
38 }

39 /* ask pcap for the network address and mask of the device */
40 pcap_lookupnet(dev,&netp,&maskp,errbuf);

41 /* open device for reading this time lets set it in promiscuous
42 * mode so we can monitor traffic to another machine */
43 descr = pcap_open_live(dev,BUFSIZ,1,-1,errbuf);
44 if(descr == NULL) {
45     printf("pcap_open_live(): %s\\n",errbuf);
46     exit(1);
47 }

48 /* Lets try and compile the program.. non-optimized */
49 if(pcap_compile(descr,&fp,argv[1],0,netp) == -1) {
50     fprintf(stderr,"Error calling pcap_compile\\n");
51     exit(1);
52 }

53 /* set the compiled program as the filter */
54 if(pcap_setfilter(descr,&fp) == -1) {
55     fprintf(stderr,"Error setting filter\\n");
56     exit(1);
57 }

58 /* capture packets */
```

```
59 pcap_loop(descr, -1, my_callback, NULL);

60 return 0;
61 }

62 void Display (const u_char * packet, const size_t length)
63 {
64     u_long offset;
65     int i, j, k;

66     printf("packet [%lu bytes]: \n", (long unsigned int)length);
67     if (length <= 0) {
68         return;
69     }
70     i = 0;
71     offset = 0;
72     for (k = length / 16; k > 0; k--, offset += 16){
73         printf("%08X ", (unsigned int)offset);
74         for (j = 0; j < 16; j++, i++) {
75             if (j == 8){
76                 printf("-%02X", packet[i]);
77             }
78             else printf(" %02X", packet[i]);
79         }
80         printf(" ");
81         i -= 16;
82         for (j = 0; j < 16; j++, i++) {
83             /* if (isprint( (int)packet[i])) */
84             if ((packet[i] >= ' ') && (packet[i] <= 255)) {
85                 printf("%c", packet[i]);
86             }
87             else printf(".");
88         }
89         printf("\n");
90     }
91     k = length - i;
92     if (k <= 0){
93         return;
94     }
95     printf("%08X ", (unsigned int)offset);
96     for (j = 0 ; j < k; j++, i++){
97         if (j == 8){
98             printf("-%02X", packet[i]);
99         }
100        else printf(" %02X", packet[i]);
```

```

101  }
102  i -= k;
103  for (j = 16 - k; j > 0; j--) {
104      printf("  ");
105  }
106  printf("    ");
107  for (j = 0; j < k; j++, i++){
108      if ((packet[i] >= ' ') && (packet[i] <= 255)) {
109          printf("%c", packet[i]);
110      }
111      else {
112          printf(".");
113      }
114  }
115  printf("\n");
116  return;
117  }

```

代码说明如下。

9：定义错误缓冲区大小。

10：声明函数 Display()，用于显示所捕获的包。

12~16：实现回调函数。调用了 Display()函数。

34~38：查找有效的网络设备，例如 eth0。

39~40：获得网络设备对应的网络地址及网络掩码。

41~47：打开网络设备。它指定每个包的最大长度为 BUFSIZ。为了保证包捕捉的效率，该值应适中，以恰能存放所捕捉的包为准，一般取决于所捕捉包的类型。设置网络设备处于混杂模式，这样可以捕捉所有经过该网络设备的数据包，对于以太网而言，即为局域网上的所有包，不论地址是否为本网络设备。超时值设为-1，表示不超时。

49~52：将用户通过命令行输入的字串符 argv[1]编译到过滤程序 fp 中。

53~57：将过滤程序 fp 设置为过滤器。

58~59：捕捉所需的包直到缓冲区满。每捕获一个包则调用一次回调函数 my\_callback()。

62~117：以二进制方式显示所捕获的包。

运行程序 (capture ip) 结果如下：

```

packet [ 54 bytes ] :
00000000  20 53 52 43 00 00 44 45-53 54 00 00 08 00 45 00  SRC..DEST....E.
00000010  00 28 20 03 40 00 80 06-8B 99 9A 14 4A E1 D0 B8  .(.@...J...
00000020  99 85 04 25 00 50 00 04-8E BF 2E 73 B9 93 50 10  ..%.P....s..P.
00000030  01 F8 E3 69 00 00                                .  i..
packet [ 54 bytes ] :
00000000  20 53 52 43 00 00 44 45-53 54 00 00 08 00 45 00  SRC..DEST....E.
00000010  00 28 21 03 40 00 80 06-8A 99 9A 14 4A E1 D0 B8  .(!.@...?J...
00000020  99 85 04 25 00 50 00 04-8E BF 2E 73 B9 93 50 04  ...%.P....s..P.
00000030  00 00 E5 6D 00 00                                ...m..

```

```
packet [18 bytes] :
00000000 20 53 45 4E 44 00 20 53-45 4E 44 00 C0 21 05 05    SEND. SEND.?..
00000010 00 04                                              ..
packet [18 bytes] :
00000000 20 52 45 43 56 00 20 52-45 43 56 00 C0 21 06 05    RECV. RECV.?..
00000010 00 04                                              ..
```

### 3. 运行结果说明

程序 (capture) 每捕获 1 个包, 则显示包的长度并以十六个字符为单位逐行显示偏移量、十六进制码和 ASCII 码。

## 13.4 小 结

为了直接捕获网络数据, 必须访问数据链路层。目前, 访问数据链路层的主要方法有: BPF、DLPI、SOCK\_PACKET 和 libpcap。

由于 libpcap 独立于操作系统的执行, 被广泛地运用。它提供用于捕捉数据的函数。其捕捉数据的基本过程为: 查找并打开网络设备, 设置过滤器, 捕捉数据包。

# 第 14 章 多接口设计

通常，服务器所在的主机存在多个网络接口，即包含多个网络地址。某个特定服务一般都采用固定的端口号，而网络地址却由所运行主机的不同而不同，因此，如果服务器所在主机存在多个网络接口，应考虑多地址绑定问题。

本章介绍在端口固定时，服务器多地址绑定的主要两种情况：

- 单个服务器绑定到多个接口
- 多个服务器绑定到不同接口

## 14.1 单个服务器绑定到多个接口

服务器的绑定是通过 `bind()` 函数实现的，为了绑定到多个接口上，必须使用常量 `INADDR_ANY` 来代表本地主机所有有效的 IP 地址。当服务器绑定到所有的 IP 地址后，客户可以从其中任一地址获得服务器的服务。

以下通过例子说明整个过程。假设服务器所在主机有 2 个 IP 地址(192.9.200.10 和 192.9.200.20)，而 TCP 服务器绑定到所有地址上并且端口号为 80，此时服务器状态如图 14.1 所示。

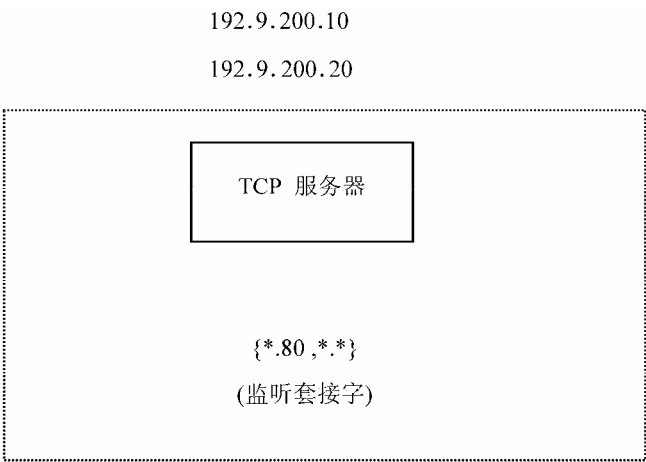


图 14.1 TCP 服务器监听套接字状态

用 `{*.80, *. *}` 表示本地套接字绑定到所有地址上并且端口为 80，远程套接字为任何地址和任何端口。

当远程客户进行连接请求，建立连接并产生连接套接字后，连接套接字的地址则为客

户的请求地址，如图 14.2 所示。

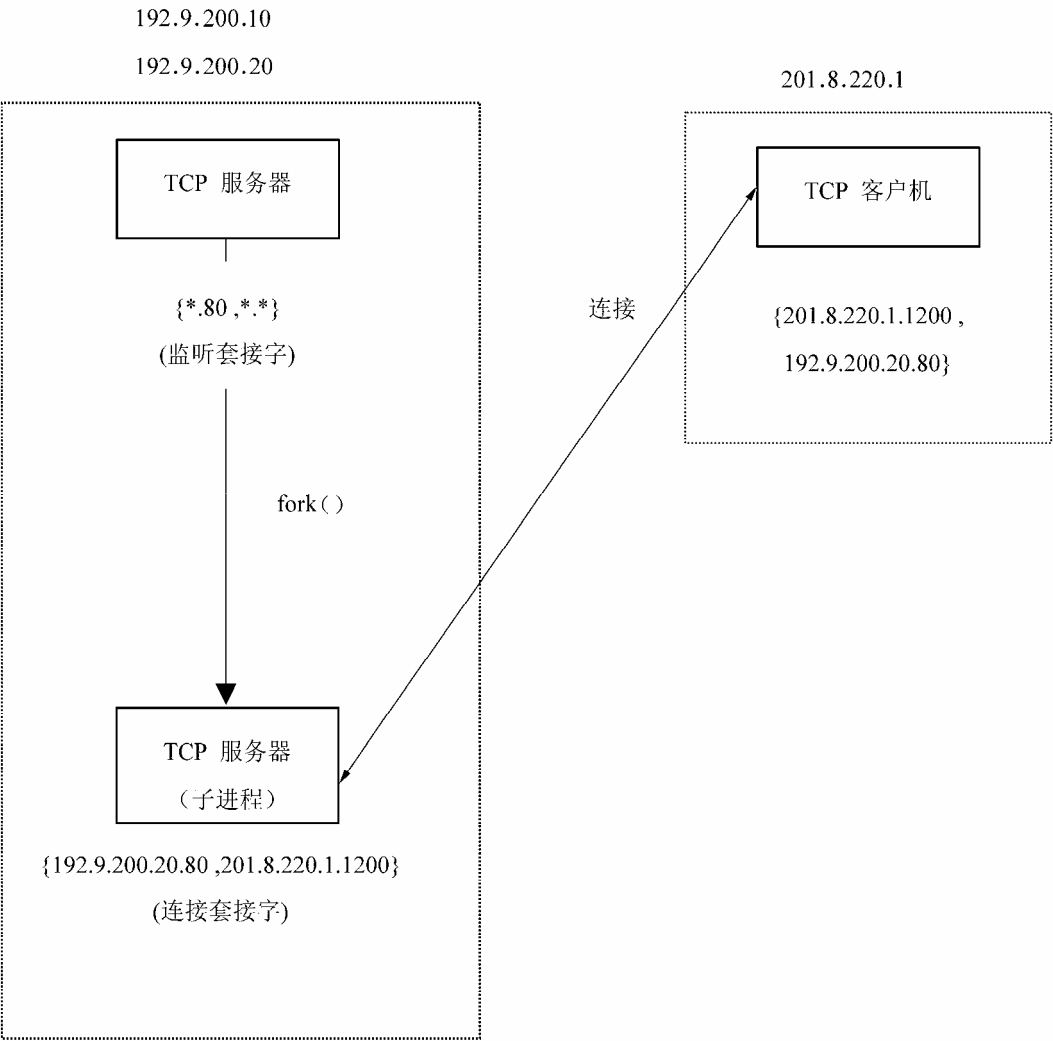


图 14.2 TCP 服务器连接套接字状态

实例（程序 14.1）

为了获得被连接的 IP 地址，应调用 getsockname()函数。以下的例子是一个多进程并发服务器，它绑定于所有 IP 地址上且端口号为 1234，当与客户建立连接后显示被连接的 IP 地址及端口号。源程序如下：

```
1 #include <stdio.h>          /* These are the usual header files */
2 #include <strings.h>        /* for bzero() */
3 #include <unistd.h>         /* for close() */
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
```

```
7 #include <arpa/inet.h>

8 #define PORT 1234 /* Port that will be opened */
9 #define BACKLOG 2 /* Number of allowed connections */
10 void process_cli(int connectfd, sockaddr_in client);

11 main()
12 {
13     int listenfd, connectfd; /* socket descriptors */
14     pid_t pid;
15     struct sockaddr_in server; /* server's address information */
16     struct sockaddr_in client; /* client's address information */
17     int sin_size;

18     /* creaet TCP socket */
19     if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
20     {
21         /* handle exception */
22         perror("Creating socket failed.");
23         exit(1);
24     }

25     bzero(&server, sizeof(server));
26     server.sin_family=AF_INET;
27     server.sin_port=htons(PORT);
28     server.sin_addr.s_addr = htonl (INADDR_ANY);

29     if (bind(listenfd, (struct sockaddr *)&server,
30             sizeof(struct sockaddr)) == -1) {
31         /* handle exception */
32         perror("Bind error.");
33         exit(1);
34     }

35     if(listen(listenfd,BACKLOG) == -1){ /* calls listen() */
36         perror("listen() error\n");
37         exit(1);
38     }

39     sin_size=sizeof(struct sockaddr_in);
40     while(1)
41     {
42         /* accept connection */
43         if ((connectfd = accept(listenfd,
44                                 (struct sockaddr *)&client,&sin_size))== -1) {
```

```
43     perror("accept() error\n");
44     exit(1);
45 }

46 /* create child process */
47 if ((pid=fork())>0) {
48     /* parent process */
49     close(connectfd);
50     continue;
51 }
52 else if (pid==0) {
53     /* child process */
54     close(listenfd);
55     process_cli(connectfd, client);
56     exit(0);
57 }
58 else {
59     printf("fork error\n");
60     exit(0);
61 }
62 }

63 close(listenfd); /* close listenfd */
64 }

65 void process_cli(int connectfd, sockaddr_in client)
66 {
67     sockaddr_in address;
68     int namelen;

69     getsockname(connectfd, (sockaddr *)&address, &namelen);
70     printf("Connected to address ( %s )\n ",inet_ntoa(address.sin_addr) );
71     close(connectfd); /* close connectfd */
72 }
```

代码说明如下。

8~9：定义端口号、最大允许连接的数量。

10：声明“客户处理”process\_cli()函数，用于处理客户请求。

19~23：产生 TCP 套接字。

25~33：绑定套接字到相应地址。本例的 IP 地址设为 INADDR\_ANY，则可接收来自本机任何 IP 地址的客户连接。

34~37：监听网络连接。

41~62：接受客户连接。一旦连接，产生子进程服务客户。然后关闭连接套接字并继续接受下一客户连接。



54~57：子进程首先关闭监听套接字，再调用客户处理函数 `process_cli()` 处理客户请求。完成后退出子进程。`exit()` 会关闭所有子进程打开的描述符。

59~60：如果子进程产生失败，显示出错信息并退出程序。

69：获得连接套接字地址，存放于 `address`。

70：显示连接套接字地址。

71：关闭连接套接字。

## 14.2 多个服务器绑定到多个接口

一个主机系统上经常运行多个服务器，有时它们被绑定到不同的接口上以服务不同的客户。例如，有两个部门分别通过两个接口（192.9.200.10 和 192.9.200.20）访问服务器。这时需要启动 2 个服务器，一个绑定在 192.9.200.10 上，另一个绑定在 192.9.200.20 上。

实例（程序 14.2）

以下通过一个简单程序模拟这一过程。用户通过命令行参数输入该服务器所绑定的地址，服务器完成地址绑定后监听客户连接并为客户服务，其源程序如下：

```
1 #include <stdio.h>           /* These are the usual header files */
2 #include <strings.h>         /* for bzero() */
3 #include <unistd.h>          /* for close() */
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7 #include <arpa/inet.h>

8 #define PORT 1234           /* Port that will be opened */
9 #define BACKLOG 2           /* Number of allowed connections */
10 void process_cli(int connectfd, struct sockaddr_in client);

11 main(int argc, char* argv[])
12 {
13     int listenfd, connectfd; /* socket descriptors */
14     pid_t pid;

15     struct sockaddr_in server; /* server's address information */
16     struct sockaddr_in client; /* client's address information */
17     int sin_size;

18     if (argc != 2) {
19         printf("Usage: %s <IP Address>\n", argv[0]);
20         exit(1);
21     }
```

```
22 if ((he=gethostbyname(argv[1]))==NULL){ /* calls gethostbyname() */
23     printf("gethostbyname() error\n");
24     exit(1);
25 }

26 /* Create TCP socket */
27 if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
28 {
29     /* handle exception */
30     perror("Creating socket failed.");
31     exit(1);
32 }

33 bzero(&server,sizeof(server));
34 server.sin_family=AF_INET;
35 server.sin_port=htons(PORT);
36 server.sin_addr = *((struct in_addr *)he->h_addr);
37 if (bind(listenfd, (struct sockaddr *)&server,
           sizeof(struct sockaddr)) == -1) {
38     /* handle exception */
39     perror("Bind error.");
40     exit(1);
41 }

42 if(listen(listenfd,BACKLOG) == -1){ /* calls listen() */
43     perror("listen() error\n");
44     exit(1);
45 }

46 sin_size=sizeof(struct sockaddr_in);
47 while(1)
48 {
49     /* accept connection */
50     if ((connectfd = accept(listenfd,
                             (struct sockaddr *)&client,&sin_size))== -1) {
51         perror("accept() error\n");
52         exit(1);
53     }

54     /* create child process */
55     if ((pid=fork())>0) {
56         /* parent process */
57         close(connectfd);
58         continue;
```

```

59  }
60  else if (pid==0) {
61      /* child process */
62      close(listenfd);
63      process_cli(connectfd, client);
64      exit(0);
65  }
66  else {
67      printf("fork error\n");
68      exit(0);
69  }
70  }
71  close(listenfd); /* close listenfd */
72  }

73  void process_cli(int connectfd, sockaddr_in client)
74  {
75      sockaddr_in address;
76      int namelen;

77      getsockname(connectfd, (sockaddr *)&address, &namelen);
78      printf("Connected to address ( %s )\n ",inet_ntoa(address.sin_addr));
79      close(connectfd); /* close connectfd */
80  }

```

代码说明如下。

22~25：获得用户从命令行输入的地址信息

36：绑定套接字到由用户输入的地址上。

我们在具有两个接口（192.7.200.10 和 192.9.200.20）的主机上运行上述程序。例如同时运行“server 192.9.200.10”和“server 192.9.200.20”，将该服务器分别绑定在 2 个地址上。但我们会发现第二个命令会发生错误，显示“Bind error.: Address already in use”。这是由于在默认情况下，bind()函数不能同时绑定 2 个以上的服务器到相同的端口上，否则会产生上述错误。解决办法就是利用套接字选项 SO\_REUSEADDR，它允许多个服务器绑定到相同端口但不同的地址上。因此将上述程序进行修改，增加设置 SO\_REUSEADDR 套接字选项，修改后的源程序如下：

```

1  #include <stdio.h> /* These are the usual header files */
2  #include <strings.h> /* for bzero() */
3  #include <unistd.h> /* for close() */
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>

```

```
8 #define PORT 1234 /* Port that will be opened */
9 #define BACKLOG 2 /* Number of allowed connections */

10 void process_cli(int connectfd, sockaddr_in client);

11 main(int argc, char* argv[])
12 {
13     int listenfd, connectfd; /* socket descriptors */
14     pid_t pid;
15     struct sockaddr_in server; /* server's address information */
16     struct sockaddr_in client; /* client's address information */
17     int sin_size;

18     if (argc !=2) {
19         printf("Usage: %s <IP Address>\n",argv[0]);
20         exit(1);
21     }

22     if ((he=gethostbyname(argv[1]))==NULL){ /* calls gethostbyname() */
23         printf("gethostbyname() error\n");
24         exit(1);
25     }

26     /* Create TCP socket */
27     if ((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
28     {
29         /* handle exception */
30         perror("Creating socket failed.");
31         exit(1);
32     }

33     bzero(&server,sizeof(server));
34     server.sin_family=AF_INET;
35     server.sin_port=htons(PORT);
36     server.sin_addr = *((struct in_addr *)he->h_addr);

37     /* set socket option*/
38     int opt = SO_REUSEADDR;
39     setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

40     if (bind(listenfd, (struct sockaddr *)&server,
41             sizeof(struct sockaddr)) == -1) {
42         /* handle exception */
43         perror("Bind error.");
44         exit(1);
45     }
```

```
44  }

45  if(listen(listenfd, BACKLOG) == -1){ /* calls listen() */
46  perror("listen() error\n");
47  exit(1);
48  }

49  sin_size=sizeof(struct sockaddr_in);
50  while(1)
51  {
52  /* accept connection */
53  if ((connectfd = accept(listenfd,
        (struct sockaddr *)&client, &sin_size)) == -1) {
54  perror("accept() error\n");
55  exit(1);
56  }

57  /* create child process */
58  if ((pid=fork())>0) {
59  /* parent process */
60  close(connectfd);
61  continue;
62  }
63  else if (pid==0) {
64  /* child process */
65  close(listenfd);
66  process_cli(connectfd, client);
67  exit(0);
68  }
69  else {
70  printf("fork error\n");
71  exit(0);
72  }
73  }
74  close(listenfd); /* close listenfd */
75  }

76  void process_cli(int connectfd, struct sockaddr_in client)
77  {
78  struct sockaddr_in address;
79  int namelen;

80  getsockname(connectfd, (struct sockaddr *)&address, &namelen);
81  printf("Connected to address ( %s )\n ", inet_ntoa(address.sin_addr));
82  close(connectfd); /* close connectfd */
```

```
83 }
```

37~39 :设置 `SO_REUSEADDR` 套接字选项。注意,该选项必须在调用 `bind()`函数设置。

我们再次运行上述程序,发现虽然该程序两个绑定到不同地址上的运行实体同时运行,但都能正常为客户服务。

## 14.3 小 结

多接口设计主要存在两种情况:单个服务器绑定到多个接口和多个服务器绑定到不同接口。

单个服务器绑定到多个接口的实现是采用绑定套接字到 `INADDR_ANY` 地址上,但此时应注意不能再有其他的服务绑定在相同的端口上。

多个服务器绑定到不同接口的实现是采用设置套接字选项 `SO_REUSEADDR`,以防止第 2 个服务器绑定时发生错误。

## 第 15 章 路由套接字

Unix 系统集成了路由的功能，它包含相应的路由数据库可提供路由信息，同时可对路由数据库进行修改以增加或删除路由。以上的操作可以通过系统命令来完成，例如，使用 `netstat` 命令来获得路由信息，使用 `route` 命令来修改路由表。另外，应用程序也可以利用路由套接字实现对系统路由数据库的读/写操作。

路由套接字是应用程序与系统路由子系统的接口。与其他套接字一样，一旦应用程序产生了路由套接字，就可对其进行读/写操作。应用程序通过写操作把消息发送给路由子系统以增加或删除路由；通过读操作接收从路由子系统发送来的路由信息。

在实际应用中，路由的修改通常由网络管理员通过系统命令来完成。而应用程序往往只需要系统提供路由信息，因此在本章中着重讲述应用程序如何通过路由套接字获取路由信息。

### 15.1 创建路由套接字

路由套接字的创建是调用 `socket()` 函数实现的。其通信域为 `AF_ROUTE`，它只能支持一种套接字类型——原始套接字，典型的代码如下：

```
int sockfd;  
sockfd = socket(AF_ROUTE, SOCK_RAW, 0);
```

路由套接字只有在超级用户的权限下才能创建，以防止普通用户获取或修改路由。

### 15.2 读写路由套接字

创建路由套接字后，与其他套接字一样，可调用 `read()` 或 `write()` 函数对其进行读写。但由于系统内核是根据应用写入的消息来完成对路由信息的提供与修改，因此，与其他套接字不同的是，写入/读取的数据是具有固定格式的消息。例如，为增加路由，消息的类型必须为 `RTM_ADD`；为获取路由，消息的类型必须为 `RTM_GET`。

路由套接字在 `<net/route.h>` 头文件中定义了相应的消息类型以完成不同的功能，主要包括：

- `RTM_ADD` 增加路由；
- `RTM_DELETE` 删除路由；
- `RTM_CHANGE` 改变路由；

- RTM\_GET 获取路由；
- RTM\_MISS 查询路由失败。

消息的发送和接收是通过写入或读取结构实现的。路由套接字涉及3个结构( rt\_msghdr、 if\_msghdr 和 ifam\_msghdr )，它们用于不同类型的消息，具体定义如下：

```
/*
 * Structures for routing messages.
 */
struct rt_msghdr {
    u_short rmt_msglen;    /* to skip over non-understood messages */
    u_char  rtm_version;   /* future binary compatibility */
    u_char  rtm_type;      /* message type */
    u_short rmt_index;     /* index for associated ifp */
    pid_t   rmt_pid;       /* identify sender */
    __uint32_t rtm_addrs; /* bitmask identifying sockaddrs in msg */
    int     rtm_seq;       /* for sender to identify action */
    int     rtm_errno;     /* why failed */
    int     rtm_flags;     /* flags, incl kern & message, e.g. DONE */
    int     rtm_use;       /* from rtenry */
    u_long  rtm_inits;     /* which values we are initializing */
    struct  rt_metrics rtm_rmx; /* metrics themselves */
};

struct rt_metrics {
    u_long rmx_locks;      /* Kernel must leave these values alone */
    u_long rmx_mtu;        /* MTU for this path */
    u_long rmx_hopcount;    /* max hops expected */
    u_long rmx_expire;     /* lifetime for route, e.g. redirect */
    u_long rmx_recvpipe;   /* inbound delay-bandwidth product */
    u_long rmx_sendpipe;   /* outbound delay-bandwidth product */
    u_long rmx_ssthresh;   /* outbound gateway buffer limit */
    u_long rmx_rtt;        /* estimated round trip time */
    u_long rmx_rttvar;     /* estimated rtt variance */
};

/*
 * Message format for use in obtaining information about interfaces
 * from the routing socket
 */
typedef struct if_msghdr {
    ushort_t ifm_msglen;    /* to skip over non-understood messages */
    uchar_t  ifm_version;   /* future binary compatibility */
    uchar_t  ifm_type;      /* message type */
    int      ifm_addrs;     /* like rtm_addrs */
    int      ifm_flags;     /* value of if_flags */
    ushort_t ifm_index;     /* index for associated ifp */
    struct    if_data ifm_data; /* statistics and other data about if */
};
```



```

} if_msghdr_t;
/*
 * Message format for use in obtaining information about interface addresses
 * from the routing socket
 */
typedef struct ifa_msghdr {
    ushort_t ifam_msglen; /* to skip over non-understood messages */
    uchar_t ifam_version; /* future binary compatibility */
    uchar_t ifam_type; /* message type */
    int ifam_addrs; /* like rtm_addrs */
    int ifam_flags; /* route flags */
    ushort_t ifam_index; /* index for associated ifp */
    int ifam_metric; /* value of ipif_metric */
} ifa_msghdr_t;

```

其中 rtm\_type、ifm\_type、ifam\_type 表示消息的类型。

rtm\_addrs、ifm\_addrs、ifam\_addrs 是位屏蔽字，表示跟随在消息后的套接字地址结构 (sockaddr) 的地址类型。类型包括：

- RTA\_DST(0x1) 目的地址；
- RTA\_GATEWAY(0x2) 网关地址；
- RTA\_NETMASK(0x4) 网络掩码；
- RTA\_GENMASK(0x8) 克隆掩码；
- RTA\_IFP(0x10) 接口名；
- RTA\_IFA(0x20) 接口地址；
- RTA\_AUTHOR(0x40) 重定向；
- RTA\_BRD(0x80) 广播地址；
- RTA\_NUMBITS 最大位数，定义为 8。

这些类型是可以组合的，但先后次序是固定的，以便通过位屏蔽字决定哪些类型跟在消息后以及它们相应的位置。例如，位屏蔽字为 0x7，则说明消息后跟了 3 个套接字地址结构，分别表示目的地址 (0x1)、网关地址 (0x2) 和网络掩码 (0x4)。

## 15.3 读取路由信息

应用程序为获取路由信息，可通过所创建的路由套接字向系统内核发 RTM\_GET 消息，然后再接收系统内核发回的路由信息。

应用程序给出目的地址，然后通过路由套接字查询其对应的路由，这是最常见的情况。我们就这种情况来说明路由信息的读取过程。

为了完成发 RTM\_GET 消息，首先要构造该消息。它分为消息头和消息内容两部分 (见图 15.1)。



图 15.1 RTM\_GET 发送消息结构

消息头是由 `rt_msghdr` 定义的，消息类型设为 `RTM_GET`，位屏蔽字设置为 `RTA_DST`，表示消息头后将跟一个目的地址。消息内容由套接字地址结构 `sockaddr` 定义的。由于消息头仅设了 `RTA_DST`，因此消息内容只有一项，用于存放目的地址。

另外，在消息头中还要设置版本号、进程 ID、序号等，这些信息将会由系统内核原封不动地返回，因此可作为消息的标志。

构造完 `RTM_GET` 消息后，就可调用 `write()` 函数将它以数据形式写入路由套接字。

第二步就是调用 `read()` 函数来接收系统内核返回的路由消息。接收的消息也分为消息头和消息内容（见图 15.2）。

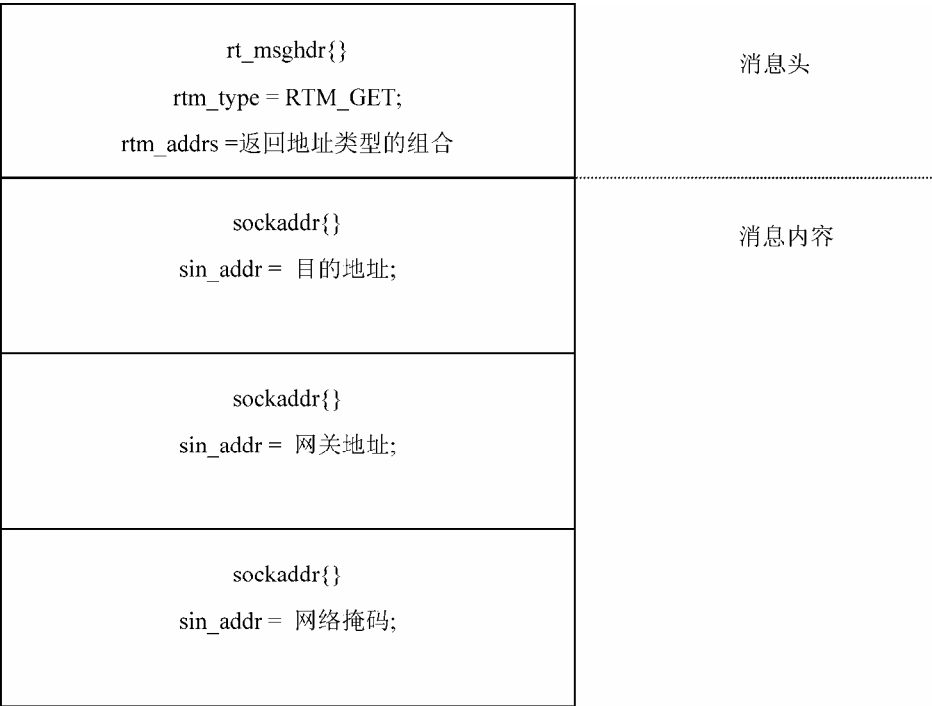


图 15.2 RTM\_GET 接收消息结构

由于存在同时打开多个路由套接字的可能，而系统内核将返回消息发给每一个打开的

原始套接字，因此常常要检查消息的类型、序号、进程 ID 等，以确保所收到的消息是自己所需要的。

## 15.4 路由套接字实例

以下是一个读取路由表项的程序。它从命令行参数读入目的地址，创建路由套接字，发送 RTM\_GET 消息给系统核，然后通过路由套接字读取并显示返回的路由信息。

### 1. 源程序（程序 15.1）

源程序代码如下：

```
1 #include <netinet/in.h>
2 #include <arpa/inet.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <net/route.h>
6 #include <sys/param.h>
7 #include <stdlib.h>
8 #include <stdio.h>
9 #include <unistd.h>

10 void cp_rtaddrs(int, sockaddr_in *, sockaddr_in **);

11 #define BUFLen (sizeof(rt_msghdr) + 512)
12 #define SEQ 1234

13 int main(int argc, char **argv)
14 {
15     int      sockfd;
16     char      *buf;
17     pid_t      pid;
18     ssize_t      n;
19     struct rt_msghdr *rtm;
20     struct sockaddr_in  *sa, *rt_info[RTA_NUMBITS];
21     struct sockaddr_in  *sin;

22     if (argc != 2) {
23         printf("usage: %s <IPaddress>\n", argv[0]);
24         exit(0);
25     }

26     sockfd = socket(AF_ROUTE, SOCK_RAW, 0); /* need superuser privileges */
27     if (sockfd == -1) {
```

```
28 perror("socket: ");
29 exit(1);
30 }

31 buf = (char * ) calloc(1, BUFLen); /* and initialized to 0 */

32 rtm = (rt_msghdr *) buf;
33 rtm->rtm_msglen = sizeof(rt_msghdr) + sizeof(sockaddr_in);
34 rtm->rtm_version = RTM_VERSION;
35 rtm->rtm_type = RTM_GET;
36 rtm->rtm_addrs = RTA_DST;
37 rtm->rtm_pid = pid = getpid();
38 rtm->rtm_seq = SEQ;

39 sin = (sockaddr_in *) (rtm + 1);
40 sin->sin_family = AF_INET;
41 inet_pton(AF_INET, argv[1], &sin->sin_addr);

42 write(sockfd, rtm, rtm->rtm_msglen);

43 do {
44     n = read(sockfd, rtm, BUFLen);
45 } while (rtm->rtm_type != RTM_GET || rtm->rtm_seq !=
          SEQ || rtm->rtm_pid != pid);

46 sa = (sockaddr_in *) (rtm + 1);
47 cp_rtaddrs(rtm->rtm_addrs, sa, rt_info);

48 if ((sa = rt_info[RTA_DST]) != NULL)
49     printf("dest: %s\n", inet_ntoa(sa->sin_addr));

50 if ((sa = rt_info[RTA_GATEWAY]) != NULL)
51     printf("gateway: %s\n", inet_ntoa(sa->sin_addr));

52 if ((sa = rt_info[RTA_NETMASK]) != NULL)
53     printf("netmask: %s\n", inet_ntoa(sa->sin_addr));

54 exit(0);
55 }

56 void cp_rtaddrs(int addrs, struct sockaddr_in *sa,
                  struct sockaddr_in **rt_info)
57 {
58     int     i;
```

```

59 for (i = 0; i < RTA_NUMBITS; i++) {
60     if (addrs & (1 << i)) {
61         rt_info[i] = sa;
62         sa++;
63     } else
64         rt_info[i] = NULL;
65 }
66 }

```

## 2. 代码分析

1~9：所需头文件。

10：声明 `cp_rtaddrs()` 函数，用于将收到的套接字地址拷入套接字地址数组 `rt_info` 中。

11：定义缓冲区长度。

12：定义消息序号。

22~25：显示程序用法。

26~30：创建路由套接字。

31：分配缓冲区空间。

32~38：设置消息的长度、版本号、类型（`RTM_GET`）、位屏蔽字（`RTA_DST`）、进程 ID 和序号。

39~41：将用户输入的目的地址填入跟在消息头后的套接字地址项中。

42：发送 `RTM_GET` 消息给路由套接字。

43~45：反复从路由套接字中读取消息，直到收到类型为 `RTM_GET`、序号为 `SEQ`、进程 ID 为当前进程 ID 的消息，表明该消息是所发消息的回应。

46~47：将读取的套接字地址项拷入数组 `rt_info` 中，该数组可依次存放所有类型的套接字地址项。将读入的套接字地址项拷入相应数组中。如果某类型的套接字地址项没有被读入，则相应的数组项为空。

48~49：如果存在目的地址项，则显示目的地址。

50~51：如果存在网关地址项，则显示网关地址。

52~53：如果存在网络掩码项，则显示网络掩码。

60~64：依次检查位屏蔽字，如果某项存在，则拷入数组对应项中，否则填入 `NULL`。

## 3. 运行程序

在超级用户权限下运行上述程序（`prg15_1`）以查看相应路由。输入“`prg15_1 192.9.200.10`”。

## 4. 服务器运行结果

```

# prg15_1 192.9.200.10
dest: 192.9.200.10
gateway: 192.9.201.1
#

```

### 5. 运行结果说明

192.9.200.10 的路由为 192.9.201.1 ,即目的地址为 192.9.200.10 的 IP 包要经 192.9.201.1 到达目的地。

## 15.5 小 结

应用程序可以通过 `socket()` 函数创建路由套接字, 以及调用 `read()` 和 `write()` 函数来读/写路由套接字, 从而读取或修改系统的路由信息。路由套接字是应用程序与系统内核间的接口。

可以将系统内核看作服务器, 应用程序看作客户, 应用程序通过向原始套接字写入消息而向系统内核提出请求, 并通过路由套接字读取来自系统内核的回应消息。消息可分为消息头和消息内容。头由结构 (`rtm_type`、`ifm_type` 或 `ifam_type`) 定义, 它表明了消息的类型等信息。消息内容由若干套接字地址结构组成。它的类型和数量由位屏蔽字 (`rtm_addrs`、`ifm_addr` 或 `ifam_addrs`) 决定。

## 第 16 章 简单路由器实例分析

在基于 TCP/IP 的网络中，IP 路由器是网络互连的关键，它决定了 IP 包传输的路径。Unix 系统集成了 IP 路由的功能，即具有多个网络接口的 Unix 主机能够通过路由配置而作为路由器使用。但当增加新传输设备时，由于它不属于系统所识别的网络接口，无法将新路径加入到系统的路由子系统中。在这种情况下可采用 2 种办法解决：

- 开发新设备的驱动程序，使之成为系统所识别的网络接口；
- 开发新的路由器。

开发新设备的驱动程序是一种较好的解决办法，它涉及驱动程序的开发，在此不予讨论。在一些特殊的应用中，也可考虑开发新的路由器。例如，对于采用专用传输设备的网络应用，由于其设备应用范围窄，没有必要开发其驱动程序。同时，这些设备所采用的通信协议一般较为简单，因而在原系统路由功能的基础上，开发专用的 IP 路由器并不复杂。本章的实例就是针对这种情况，介绍如何利用 Unix 网络编程技术开发专用的 IP 路由器。

### 16.1 设计专用路由器

本例的路由器是一个专用的 IP 路由器，即它可以完成 IP 包的路由功能，并且通过专用的传输设备/网络传输 IP 包，使得两个基于 TCP/IP 的网络可以通过专用传输设备/网络实现互连。系统体系图见图 16.1。



图 16.1 专用路由器的应用体系

专用路由器的接口包括：

- 网络接口——与基于 TCP 的网络相连；
- 串行口——与专用传输设备相连。

专用传输网络的传输协议非常简单，它可以全双工传输专用格式的数据包。其数据包格式如图 16.2。

|    |     |      |      |    |
|----|-----|------|------|----|
| 7E | 源地址 | 目的地址 | 数据长度 | 数据 |
|----|-----|------|------|----|

图 16.2 专用传输网络的数据包格式

数据包包括：

- 头标志 7E
- 2 个字节的源地址
- 2 个字节的地址
- 2 个字节的数据长度和数据。

专用路由器完成的功能如下。

- 从 TCP/IP 网络捕获 IP 包，查询系统路由表，如果路由表不包含相应的路由，则通过串口发给专用传输网络。
- 通过串口从专用传输网络接收数据包，然后解出 IP 包，并将其发给 TCP/IP 网络。

我们采用 libpcap 库来捕获 IP 包，利用路由套接字完成查询系统路由表；利用串口通信完成从专用传输网络收/发数据包并采用原始套接字来发送 IP 包给 TCP/IP 网络。专用路由器的信息流程图如图 16.3 所示。

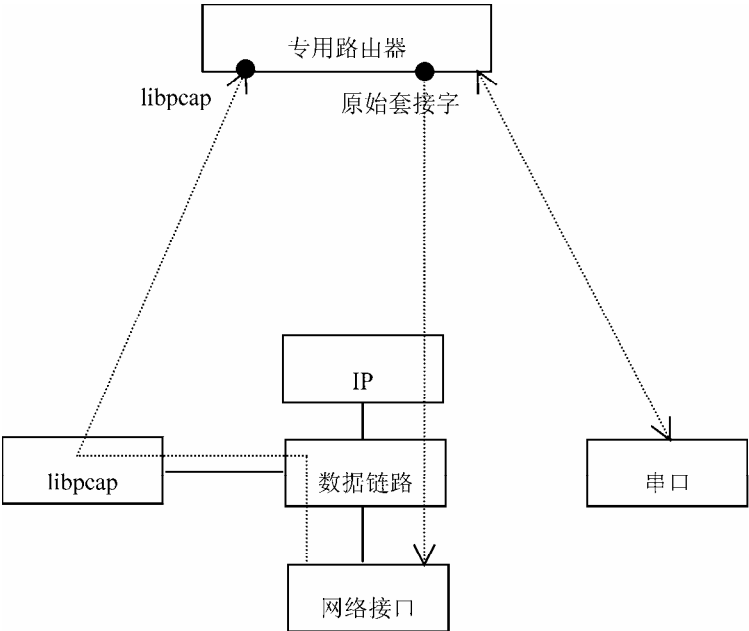


图 16.3 专用路由器信息流程图

为了使专用路由器有效地工作，我们设计如下两个线程。

- 发送线程：负责捕捉网络中 IP 包，取出目的 IP 地址，并查询系统路由表，如果路由表不含相应路由，则将 IP 封装在专用传输网络的数据包内，然后通过串口发给专用传输设备。其流程如图 16.4 所示。
- 接收线程：负责通过串口从专用传输网络接收数据包，并解出 IP 包，然后将 IP



包发给网络。其流程如图 16.5 所示。

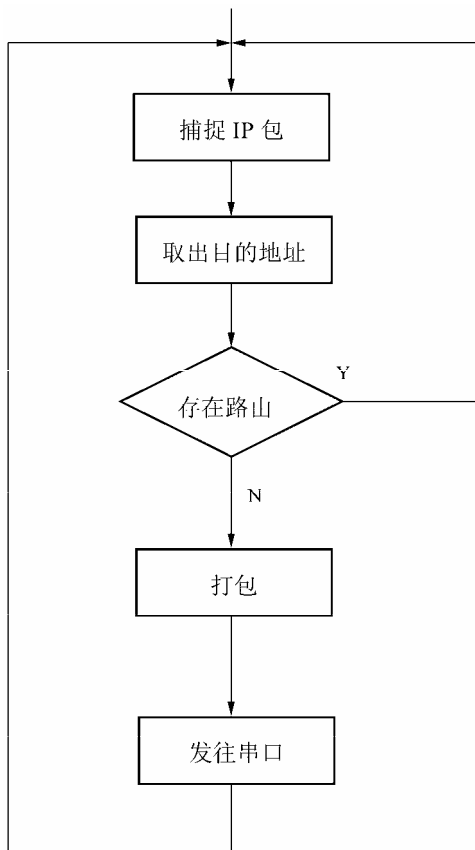


图 16.4 专用路由器发送线程流程

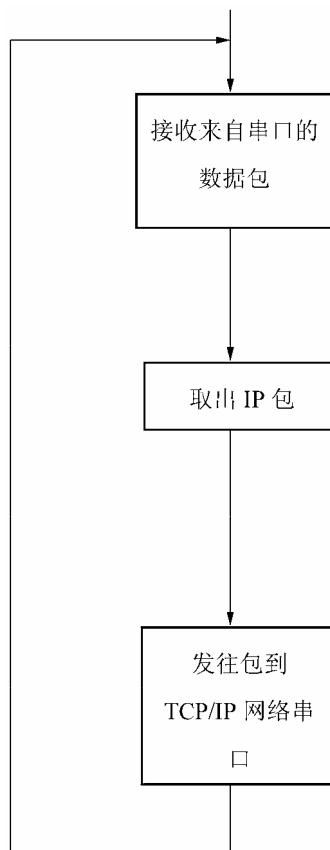


图 16.5 专用路由器发送接收线程流程

为实现专用路由器，我们设计以下的类。

- myCap：封装 libpcap 库函数，用于从数据链路层捕获数据包。
- myRoute：封装路由套接字的功能，用于路由表的查询和修改。
- myThread：封装 POSIX 线程的系统调用，用于产生线程。
- myRaw：封装原始套接字功能。
- SerialComm：封装串口通信功能。
- myDevice：用于收/发专用传输网络数据。
- recvThr：专用路由器接收线程。
- sendThr：专用路由器发送线程。
- myRouter：完成专用路由器初始化，线程控制等功能。
- myCapIP：用于捕获 IP 包。

它们之间的关系如图 16.6 所示。myRouter 引用对象 sendThr 和 recvThr 来完成路由功能并实现多线程以提高效率。sendThr 从 myThread 类派生而来，生成线程并引用对象 myCapIP、myRoute 和 myDevice 以捕获 IP 包、查询系统路由表，发送数据包到专用传输网络。recvThr 从 myThread 类派生而来，生成线程并引用 myRaw 和 myDevice 以完成 IP 包发

送，接收从专用传输网络来的数据包。myCapIP 类从 myCap 类派生而来，以捕获数据链路层数据包。myDevice 类从 SerialComm 类派生而来，以完成串口通信。

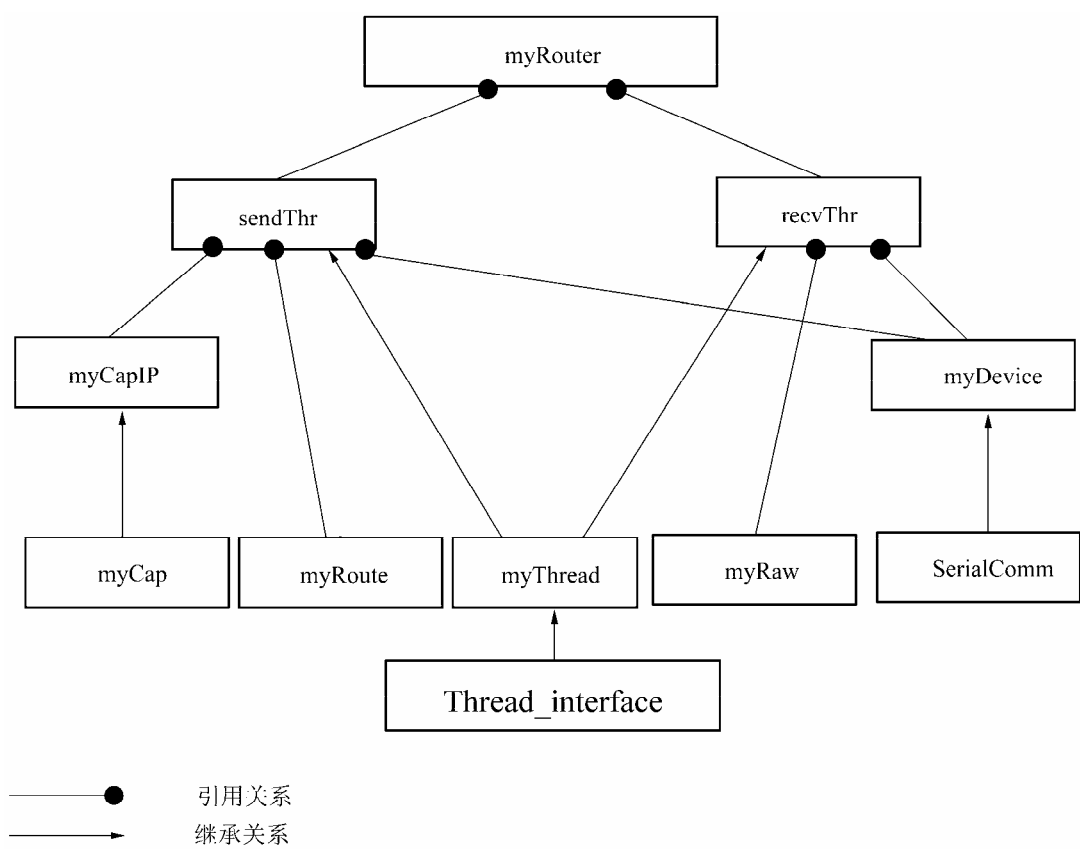


图 16.6 专用专用路由器对象关系图

## 16.2 实现专用路由器

### 16.2.1 捕获数据包：myCap 类和 myCapIP 类

myCap 类是 libpcap 函数库的封装类，利用它可以捕获数据链路层的数据包。myCapIP 为是 myCap 类的子类，它通过设置捕获规则实现只捕获 IP 包，再通过 IP 包的结构解析出 IP 地址等信息。IP 结构的定义如下：

```
struct ip {
#ifdef _BIT_FIELDS_LTOH
    uchar_t ip_hl:4,          /* header length */
            ip_v:4;          /* version */
#else
    uchar_t ip_v:4,          /* version */
            ip_hl:4;         /* header length */
#endif
};
```

```

#endif
    uchar_t ip_tos;          /* type of service */
    short   ip_len;          /* total length */
    ushort_t ip_id;         /* identification */
    short   ip_off;         /* fragment offset field */
#define IP_DF 0x4000        /* dont fragment flag */
#define IP_MF 0x2000        /* more fragments flag */
    uchar_t ip_ttl;         /* time to live */
    uchar_t ip_p;           /* protocol */
    ushort_t ip_sum;        /* checksum */
    struct in_addr ip_src, ip_dst; /* source and dest address */
};

```

myCap 与 myCapIP 类的定义在头文件 myCap.h 中，代码如下：

```

1 #define ERRORLEN 100
2 class myCap
3 {
4 protected:
5     char *device;
6     int snaplen;
7     int promisc;
8     int to_ms;
9     int linktype;
10    char ebuf[ERRORLEN];
11    bpf_u_int32 netp;
12    bpf_u_int32 maskp;
13    pcap_t *p;
14    bpf_program fp;
15 public:
16    myCap();
17    myCap(char *device, int snaplen, int promisc, int to_ms);
18    virtual ~myCap();

19    int init();
20    virtual int init(char* rule);
21    virtual int capture(char* buf);

22    pcap_t *open(){
23        return pcap_open_live(device, snaplen, promisc, to_ms, ebuf);
24    }
25    char *lookupdev(){
26        return pcap_lookupdev(ebuf) ;
27    }
28    int lookupnet( bpf_u_int32 *netp,bpf_u_int32 *maskp){
29        return pcap_lookupnet(device, &netp,&maskp, ebuf);

```

```
30 }
31 int dispatch( int cnt, pcap_handler callback, u_char *user){
32     return pcap_dispatch(p, cnt, callback, user);
33 }
34 int loop(int cnt, pcap_handler callback, u_char *user){
35     return pcap_loop(p, cnt, callback, user);
36 }
37 int compile( char *str, int optimize) {
38     return pcap_compile(p, &fp, str, optimize, maskp);
39 }
40 int setfilter(){
41     return pcap_setfilter(p, &fp);
42 }
43 u_char *next(pcap_pkthdr *h){
44     return pcap_next(p, h);
45 }
46 int datalink(){
47     return pcap_datalink(p);
48 }
49 void close() {
50     pcap_close(p);
51 }
52 char *geterr() {
53     char str = pcap_geterr(p);
54     strcpy(ebuf, (const char *)str);
55     return ebuf;
56 }

57 void setDevice(char *d) {device = d;}
58 void setSnaplen(int s) {snaplen = s;}
59 void setPromisc(int p) {promisc = p;}
60 void setTimeout(int t) {to_ms = t;}
61 char* getDevice() {return device;}
62 int getSnaplen() {return snaplen;}
63 int getMode() {return promisc;}
64 int getTimeout() {return to_ms;}
65 };

66 class myCapIP: public myCap
67 {
68 public:
69     myCapIP(){}
70     myCapIP(char *device, int snaplen, int promisc, int to_ms);

71 int capture(char* buf);
```

```
72 int myCapIP::init();
73 int getProto(char* buf);
74 in_addr getSaddr(char* buf);
75 in_addr getDaddr(char* buf);
76 };
```

代码说明如下。

1：定义错误缓冲区的长度。

5~14：定义类 myCap 的成员变量如下。

- device：网络设备名字，如 leo。
- snaplen：捕获缓冲区长度。
- promisc：捕获模式。
- to\_ms：超时。
- linktype：类路类型，如以太网、PPP、slip 等。
- ebuf：错误缓冲区。
- netp：网络设备的网络号。
- maskp：网络设备的网络掩码。
- p：网络设备描述符。
- fp：过滤程序。

16~18：定义 myCap 的构造函数和析构函数。

19~20：定义 myCap 的成员函数 init()，用于初始化 myCap 对象，使其处于捕获状态。

21：定义成员函数 capture()，用于捕获包。

22~56：封装主要的 libpcap 库函数。

69~70：定义 myCapIP 的构造函数和析构函数。

71~75：定义 myCapIP 类的成员函数如下。

- capture()：捕捉 IP 包。
- init()：初始化对象，使其能够捕捉 IP 包。
- getProto()：返回 IP 包的协议。
- getSaddr()：返回 IP 包源地址。
- getDaddr()：返回 IP 包目的地址。

myCap 与 myCapIP 类的实现在文件 mycap.cpp 中，代码如下：

```
1 #include <pcap.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <errno.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7 #include <arpa/inet.h>
8 #include <netinet/if_ether.h>
9 #include "mycap.h"
```

```
10  /*****
11   class name : myCap
12   *****/
13  myCap::myCap()
14  {
15   device = lookupdev();
16   snaplen = 1000;
17   promisc = 0;
18   to_ms = -1;
19  }

20  myCap::myCap(char *device1, int snaplen1, int promisc1, int to_ms1)
21  {
22   device = device1;
23   snaplen = snaplen1;
24   promisc = promisc1;
25   to_ms = to_ms1;
26  }

27  myCap::~myCap()
28  {
29   close();
30  }

31  int myCap::init()
32  {
33   /* ask pcap for the network address and mask of the device */
34   int err = lookupnet(&netp, &maskp);
35   if (err == -1) return -1;

36   /* open device for reading */
37   p = open();
38   if(p == NULL) {
39     return -1;
40   }

41   linktype = datalink();
42  }

43  int myCap::init(char *rule)
44  {
45   /* ask pcap for the network address and mask of the device */
46   int err = lookupnet(&netp, &maskp);
47   if (err == -1) return -1;
```

```
48  /* open device for reading */
49  p = open();
50  if(p == NULL) {
51      return -1;
52  }

53  linktype = datalink();

54  /* Lets try and compile the program.. non-optimized */
55  if(compile(rule,0) == -1) {
56      return -1;
57  }

58  /* set the compiled program as the filter */
59  if(pcap_setfilter(descr,&fp) == -1) {
60      return -1;
61  }
62 }

63 int myCap::capture(char* buf)
64 {
65     pcap_pkthdr h;
66     char* b = next(&h);
67     if (b == NULL) return -1;
68     memcpy(buf,b,h.caplen);
69     return h.caplen;
70 }

71 /*****
72     class name : myCapIP
73     *****/
74 myCapIP::myCapIP(char *device, int snaplen, int promisc, int to_ms):
75     myCap(device, snaplen, promisc, to_ms)
76 {
77 }

78 int myCapIP::init()
79 {
80     return myCap::init("ip");
81 }

82 int myCapIP::capture(char* buf)
83 {
84     int len;
85     char    *ptr;
```

```
86 struct ether_header *eptr;
87 pcap_pkthdr h;

88 char* ptr = next(&h);
89 if (ptr == NULL) return -1;

90 len = h.caplen;
91 switch (linktype) {
92     case DLT_NULL: /* loopback header = 4 bytes */
93         ptr += 4;
94         len -= 4;
95         break;
96     case DLT_EN10MB:
97         eptr = (struct ether_header *) ptr;
98         if (ntohs(eptr->ether_type) != ETHERTYPE_IP)
99             return -1;
100        ptr += 14;
101        len -= 14;
102        break;
103    case DLT_SLIP: /* SLIP header = 24 bytes */
104        ptr += 24;
105        len -= 24;
106        break;
107    case DLT_PPP: /* PPP header = 24 bytes */
108        ptr += 24;
109        len -= 24;
110        break;
111    default:
112        return -1;
113 }
114 }
115 memcpy(buf, ptr, len);
116 return 0;
117 }

118 int myCapIP::getProto(char* buf)
119 {
120     ip *packet;

121     packet = (ip *)buf;
122     return ip->ip_p;
123 }

124 in_addr myCapIP::getSaddr(char* buf)
125 {
```



```

126 ip  *packet;

127 packet = (ip *)buf;
128 return ip->ip_src;
129 }

130 in_addr myCapIP::getDaddr(char* buf)
131 {
132 ip  *packet;

133 packet = (ip *)buf;
134 return ip->ip_dst;
135 }

```

代码说明如下。

13~26：实现 myCap 类的构造函数，用于给成员变量赋初值。

27~30：实现 myCap 类的析构函数，用于关闭网络设备描述符。

31~62：实现 myCap 类的成员函数 init()。

46~47：获取网络设备的网络号和网络掩码。

49~52：打开网络设备。

53：获得链路类型。

54~57：编译捕获规则。

58~61：设置过滤程序。

63~70：实现 myCap 的成员函数 capture()。它返回数据包长度。

74~77：实现 myCapIP 的构造函数，它调用了其父类构造函数。

78~81：实现 myCapIP 的成员函数 init()。它设置捕获规则为 ip，表示仅捕捉协议为 IP 的数据包。

88~89：捕获协议为 IP 的数据包。

91~114：根据链路类型解析 IP 包。由于每种类型的数据链路有不同的包头，因而所封装的 IP 包有不同的起始位置。

115：拷贝 IP 包到缓冲区。

118~135：返回 IP 包相关信息。首先，将 IP 包映射给 ip 结构，再返回相应信息。

## 16.2.2 查询系统路由：myRoute 类

myRoute 类是通过路由套接字来查询系统路由。它向路由套接字发送 GET 类型消息，然后接收路由子系统返回的路由信息。其头文件 myroute.h 代码如下：

```

1 class myRoute
2 {
3     int sockfd;
4     int buflen;
5     int seq;

```

```

6  pid_t   pid;
7  char*   buf;
8  rt_msghdr *rtm;
9  sockadd_in *rt_info[RTA_NUMBITS];
10 int error;

11 int cp_rtaddrs(int, sockadd_in *, sockadd_in **);
12 public:
13 myRoute();
14 myRoute(int buflen, int seq);
15 ~myRoute();

16 int Get(char *ipaddr);
17 sockadd_in getDst();
18 sockadd_in getGateway();
19 sockadd_in getNetmask();

20 int Error(){return error;}
21 };

```

代码说明如下。

3~10：定义 myRoute 类的成员变量如下。

- sockfd：路由套接字描述符；
- buflen：消息缓冲区长度，是最大的消息长度；
- seq：消息序列号；
- pid：进程 ID；
- buf：消息缓冲区；
- rtm：消息头；
- rt\_info：路由信息，是一个套接字地址类型的数组，数组长度为 RTA\_NUMBITS，在<sys/route.h>中定义；
- error：错误码。

13~15：声明 myRoute 的构造函数和析构函数。

16~20：声明的成员函数如下。

- Get()：通过 IP 地址查询系统路由，没有相应路由则返回 0；
- getDst()：返回路由的目的地址；
- getGateway()：返回网关地址；
- getNetmask()：返回网络掩码；
- Error()：返回错误码。

myRoute 类的实现在文件 myroute.cpp 中，代码如下：

```

1 #include <netinet/in.h>
2 #include <arpa/inet.h>
3 #include <sys/types.h>

```

```
4 #include <sys/socket.h>
5 #include <net/route.h>
6 #include <sys/param.h>
7 #include <stdlib.h>
8 #include <stdio.h>
9 #include <unistd.h>
10 #include "myroute.h"

11 myRoute::myRoute()
12 {
13     buflen = 200;
14     seq = 1234;
15     buf = (char * ) calloc(1, buflen); /* and initialized to 0 */
16     pid = getpid();

17     sockfd = socket(AF_ROUTE, SOCK_RAW, 0); /* need superuser privileges */
18     if (sockfd == -1) {
19         error = 1;
20     }
21 }

22 myRoute::myRoute(int buflen1, int seq1)
23 {
24     buflen = buflen1;
25     seq = seq1;
26     buf = (char * ) calloc(1, buflen); /* and initialized to 0 */
27     pid = getpid();

28     sockfd = socket(AF_ROUTE, SOCK_RAW, 0); /* need superuser privileges */
29     if (sockfd == -1) {
30         error = 1;
31     }
32 }

33 myRoute::~myRoute()
34 {
35     close(sockfd);
36     free(buf);
37 }

38 int myRoute::Get(char *ipaddr)
39 {
40     sockaddr_in *sa;
41     sockaddr_in *sin;
```

```
42 rtm = (rt_msghdr *) buf;
43 rtm->rtm_msglen = sizeof(rt_msghdr) + sizeof(sockaddr_in);
44 rtm->rtm_version = RTM_VERSION;
45 rtm->rtm_type = RTM_GET;
46 rtm->rtm_addrs = RTA_DST;
47 rtm->rtm_pid = pid;
48 rtm->rtm_seq = SEQ;

49 sin = (sockaddr_in *) (rtm + 1);
50 sin->sin_family = AF_INET;
51 inet_pton(AF_INET, ipaddr, &sin->sin_addr);

52 write(sockfd, rtm, rtm->rtm_msglen);

53 do {
54     n = read(sockfd, rtm, buflen);
55 } while (rtm->rtm_type != RTM_GET || rtm->rtm_seq !=
          SEQ || rtm->rtm_pid != pid);

56 sa = (sockaddr_in *) (rtm + 1);
57 return cp_rtaddrs(rtm->rtm_addrs, sa, rt_info);
58 }

59 sockadd_in myRoute::getDst()
60 {
61     if ((sa = rt_info[RTA_DST]) != NULL)
62         return sa->sin_addr;
63     return NULL;
64 }

65 sockadd_in myRoute::getGateway()
66 {
67     if ((sa = rt_info[RTA_GATEWAY]) != NULL)
68         return sa->sin_addr;
69     return NULL;
70 }

71 sockadd_in myRoute::getNetmask()
72 {
73     if ((sa = rt_info[RTA_NETMASK]) != NULL)
74         return sa->sin_addr;
75     return NULL;
76 }

77 int myRoute::cp_rtaddrs(int addrs, struct sockaddr_in *sa,
```

```

        struct sockaddr_in **rt)
78 {
79     int      i, num;

80     num = 0;
81     for (i = 0; i < RTA_NUMBITS; i++) {
82         if (addrs & (1 << i)) {
83             rt[i] = sa;
84             sa++;
85             num++;
86         } else
87             rt[i] = NULL;
88     }
89     return num;
90 }

```

代码说明如下。

11~32：实现 myRoute 构造函数，初始化 buflen、seq 和 pid，分配消息缓冲区并产生路由套接字。

33~37：实现 myRoute 析构函数，关闭路由套接字并释放消息缓冲区空间。

42~48：填充 GET 类型消息的消息头。

49~51：填充消息内容。它是一个套接字地址，其协议簇为 AF\_INET，地址为参数 ipaddr。由于 ipaddr 是字符串形式的 IP 地址，必须用 inet\_pton() 函数转换成 32 位的 IP 地址。

52：发送消息给路由套接字。

53~54：读取相应的返回消息。

56~57：将路由信息拷贝到数组 rt\_info 中。

59~64：查询路由数组 rt\_info，返回目的地址。

65~70：查询路由数组，返回网关地址。

71~76：查询路由数组，返回网络掩码。

77~90：实现 cp\_rtaddrs() 函数，其将返回的消息内容拷贝到路由数组的相应项中，并返回有效信息的数量。

### 16.2.3 发送 IP 包：myRaw 类

myRaw 类是原始套接字的封装类，由于设置了套接字选项 IP\_HDRINCL 的原始套接字可以构造 IP 头，因此可利用它进行 IP 包的发送。myRaw 类的头文件为 myraw.h，代码如下：

```

1 class myRaw
2 {
3 public:
4     int sock;
5     int error;

```

```

6   myRaw(int protocol =0);
7   virtual ~myRaw();
8   int send(const void* msg, int msglen,sockaddr* addr,unsigned int len);
9   int send(const void* msg, int msglen,char* addr);
10  int sendIP(const void* msg,int msglen,sockaddr* to, unsigned int len);
11  int receive(void* buf,int buflen,sockaddr* from,int* len);
12  int Error() {return error;}
13 };

```

代码说明如下。

4~5：定义 myRaw 成员变量。

6~7：声明 myRaw 的构造函数和析构函数。

8~12：声明的 myRaw 成员函数如下。

- send()：向原始套接字发送数据。
- sendIP()：发送 IP 包。
- receive()：从原始套接字接收数据。

myRaw 类实现在文件 myraw.cpp 中，代码如下：

```

1 #include <string.h>
2 #include <errno.h>
3 #include <stdlib.h>
4 #include <arpa/inet.h>    /* inet_ntoa */
5 #include <unistd.h>       /* close */
6 #include <stdio.h>        /* Basic I/O routines */
7 #include <sys/types.h>    /* standard system types */
8 #include <netinet/in.h>   /* Internet address structures */
9 #include <sys/socket.h>   /* socket interface functions */
10 #include <netdb.h>        /* host to IP resolution */
11 #include <netinet/in_sysm.h>
12 #include <netinet/ip.h>   /* ICMP */
13 #include "myraw.h"

14 /*****
15 class: myRaw
16 *****/
17 myRaw::myRaw(int protocol = 0)
18 {
19     sock = socket(AF_INET, SOCK_RAW,protocol);
20     // back to normal user
21     setuid(getuid());
22     if (sock == -1) error= 1;
23     else error = 0;
24 }

25 myRaw::~myRaw()

```

```
26 {
27     close(sock);
28 }

29 int myRaw::send(const void* msg,int msglen,sockaddr* to,
                unsigned int len)
30 {
31     if (error) return -1;
32     int length = sendto(sock,msg,msglen,0,(const sockaddr*)to,len);
33     if (length == -1) {
34         error = 2;
35         return -1;
36     }
37     return length;
38 }

39 int myRaw::send(const void* msg,int msglen,char* hostname)
40 {
41     sockaddr_in sin;          // Sock Internet address

42     if (error) return -1;
43     if(hostname) {
44         hostent *hostnm = gethostbyname(hostname);
45         if(hostnm == (struct hostent *) 0) {
46             return -1;
47         }
48         sin.sin_addr.s_addr = *((unsigned long *)hostnm->h_addr);
49     }
50     else
51         return -1;
52     sin.sin_family = AF_INET;

53     return send(msg,msglen,(sockaddr *)&sin, sizeof(sin));
54 }

55 int myRaw::sendIP(const void* msg,int msglen,sockaddr* to,
                unsigned int len)
56 {
57     int on = 1;
58     setsockopt(sock,IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));

59     return send(msg, msglen, to, len);
60 }

61 int myRaw::receive(void* buf,int buflen,sockaddr* from,int* len)
```

```
62 {  
63     if (error) return -1;  
64     while (1) {  
65         int length = recvfrom(sock,buf,buflen,0,from,len);  
66         if (length == -1)  
67             if (errno == EINTR) continue;  
68         else {  
69             error = 3;  
70             return -1;  
71         }  
72         return length;  
73     }  
74 }
```

代码说明如下。

17~24：实现 myRaw 的构造函数。

25~28：实现 myRaw 的析构函数。

29~54：实现 myRaw 的成员函数 send()。

57~58：设置套接字选项 IP\_HDRINCL。

61~74：实现 myRaw 的成员函数 receive()。

#### 16.2.4 封装串口通信：SerialComm 类

SerialComm 类封装了串口通信功能，实现的源程序见附录 B。它的主要成员函数如下。

- OpenSerialPort()：打开串口。
- InitSerialPort()：初始化串口。
- Close()：关闭串口。
- RawRead()：从串口读数据。
- Recv()：从串口读入指定长度的数据。
- RawWrite()：向串口发送数据。
- Send()：向串口发送指定长度的数据。
- SetBaudRate()：设置波特率。
- SetCharacterSize()：设置字符大小。
- SetParity()：设置校验位。
- SetStopBits()：设置停止位。
- SetFlowControl()：设置流控。
- BinaryMode()：设置为二进制模式。
- CharacterMode()：设置为字符模式。

#### 16.2.5 处理专用数据传输网络协议：myDevice 类

myDevice 类是 SerialComm 的子类，它能进行串口通信并且能处理专用数据传输网络



协议。其头文件 mydevice.h 的代码如下：

```
1 struct head
2 {
3     char flag;
4     short src;
5     short dst;
6     short len;
7 };

8 class myDevice : public SerialComm
9 {
10     head h;
11     int error;
12 public:
13     myDevice(char* dev);

14     int read(void* buf);
15     int write(void* buf,int buflen);
16     int Error() {return error;}
17 };
```

代码说明如下。

1~7：定义专用数据传输网络数据包的头结构 head。

10~11：定义 myDevice 的成员变量。

13：声明 myDevice 的构造函数。

14~16：声明 myDevice 的成员函数如下。

- read()：从专用数据传输网络读数据包。
- write()：发送数据到专用数据传输网络。

myDevice 类实现在文件 mydevice.cpp 中，代码如下：

```
1 #include <unistd.h> // Unix standard function definitions
2 #include <termios.h> // POSIX terminal control definitions
3 #include <fcntl.h> // File control definitions
4 #include <string.h>
5 #include "SerialComm.h"
6 #include "myDevice.h"

7 myDevice::myDevice(char* dev): SerialComm()
8 {
9     h.flag = 0x7e;
10    OpenSerialPort(dev);
11    InitSerialPort();
12 }
```

```

13 int myDevice::read(void* buf)
14 {
15     char buffer[2000];
16     int len;

17     len = RawRead(void *buffer, 2000);
18     if (len < 0 ) {
19         error = 1;
20         return len;
21     }
22     memcpy(buf, buffer + sizeof(h), len - sizeof(h);
23     return len - sizeof(h);
24 }

25 int myDevice::write(void* buf, int buflen, short src, short dst)
26 {
27     char buffer[2000];

28     h.src = src;
29     h.dst = dst;
30     h.len = buflen + sizeof(h);
31     memcpy(buffer, h, sizeof(h));
32     memcpy(buffer + sizeof(h), buf, buflen);

33     int l = RawWrite(buffer, h.len);
34     if ( l < 0 ) {
35         error = 2;
36     }
37     return l;
38 }

```

代码说明如下。

7~12：实现 myDevice 构造函数，初始化数据包头标志，打开并初始化串口。

17~21：从专用数据传输网络读入数据包。

22~23：取出封装在专用数据传输网络数据包中的 IP 包。

28~32：封装 IP 包到专用数据传输网络数据包中。

33~38：发送数据包。

### 16.2.6 同时发送和接收：sendThr、recvThr 和 myRouter 类

sendThr 和 recvThr 为 myRouter 的子类，使发送和接收在不同的线程中处理，即可同时运行。myThread 类已在第 10.3.3 一节中讲述。myRouter 类实现了守护进程功能，使路由器以守护进程方式运行。它们的实现代码如下：

```

1 #include <stdio.h>

```

```
2 #include <pthread.h>
3 #include <string.h>
4 #include <syslog.h>
5 #include <sys/types.h>
6 #include <unistd.h>
7 #include <signal.h>
8 #include <sys/stat.h>
9 #include <errno.h>
10 #include <fcntl.h>
11 #include <arpa/inet.h>
12 #include "MyThread.h"
13 #include "mycap.h"
14 #include "myroute.h"
15 #include "myraw.h"
16 #include "mydevice.h"

17 /*****
18 class name : sendThr
19 *****/
20 class sendThr : public MyThread
21 {
22     int error;
23     short src, dst;
24     char* buf;
25     myCapIP *capIP;
26     myRoute *route;
27     myDevice *device;
28 public:
29     sendThr(char*);
30     sendThr(char*, short , short);
31     ~sendThr();
32     void run();
33     int Error() {return error;}
34     void setSrc(short s) {src = s;}
35     short getSrc() {return src;}
36     void setDst(short d) {dst = d;}
37     short getDst() { return dst;}
38 };

39 sendThr::sendThr(char* dev)
40 {
41     src = 0;
42     dst = 1;
43     capIP = new myCapIP;
44     capIP->init();
```

```
45  route = new myRoute;
46  device = new myDevice(dev);
47  buf = new char[2000];
48  }

49  sendThr::sendThr(char* dev, short src1, short dst1)
50  {
51  src = src1;
52  dst = dst1;
53  capIP = new myCapIP;
54  capIP->init();
55  route = new myRoute;
56  device = new myDevice(dev);
57  buf = new char[2000];
58  }

59  sendThr::~~sendThr()
60  {
61  delete capIP;
62  delete device;
63  delete route;
64  delete[] buf;
65  }

66  void run()
67  {
68  int len;

69  while (1) {
70  // capture IP packet
71  len = capIP->capture(buf);
72  if (len < 0) {
73  error = 1;
74  continue;
75  }

76  // get IP address
77  in_addr addr = capIP->getSaddr(buf);

78  // check routing table
79  char* ipaddr;
80  ipaddr = inet_ntoa(addr); //thread_unsafe
81  int exist = route->Get(ipaddr);
82  if (exist < 1) continue; // no routing .
```

```
83         // send to serial com
84         error = device->write(buf, len, src, dst);
85     }
86 }

87 /*****
88 class name : recvThr
89 *****/
90 class recvThr : public MyThread
91 {
92     int error;
93     short src, dst;
94     char* buf;
95     myRaw *ip;
96     myDevice *device;
97 public:
98     recvThr(char*);
99     ~recvThr();
100    void run();
101    int Error() {return error;}
102 };

103 recvThr::recvThr(char* dev)
104 {
105     ip = new myRaw;
106     device = new myDevice(dev);
107     buf = new char[2000];
108 }

109 recvThr::~~recvThr()
110 {
111     delete ip;
112     delete device;
113     delete[] buf;
114 }

115 void run()
116 {
117     int len;

118     while (1) {
119         // receive IP packet from com
120         len = device->read(buf);
121         if (len < 0) {
122             error = 1;
```

```
123         continue;
124     }

125     // Get IP addr
126     ip *packet;
127     packet = (ip *)buf;
128     in_addr addr = ip->ip_src;
129     // Build dst's addr
130     sockaddr_in to;
131     to.sin_addr = addr;
132     to.sin_family = AF_INET;

133     // send IP packet to network
134     error = ip->sendIP((const void*)buf, len,
                       (sockaddr*) &to, sizeof(to));
135 }
136 }

137 /*****
138 class name : myRouter
139 *****/
140 class myRouter
141 {
142     char* dev;
143     int src;
144     int dst;
145     int error;
146     sendThr *sender;
147     recvThr *receiver;
148     int MsgPriority;
149 public:
150     myRouter(char* dev);
151     ~myRouter;

152     int run();
153     int daemon(char* name, int facility = 0);
154     void SetPriority(int p) {MsgPriority = p;}
155     int GetPriority() {return MsgPriority;}
156     void setSrc(int s) {src = s;}
157     int getSrc() {return src;}
158     void setDst(int d) {dst = d;}
159     int getDst() { return dst;}
160     int Error() { return error;}
161 };
```

```
162 myRouter::myRouter(char* d)
163 {
164     dev = d;
165     MsgPriority = LOG_NOTICE|LOG_LOCAL0;
166     sender = new sendThr(dev);
167     receiver = new recvThr(dev);
168 }

169 myRouter::~~myRouter()
170 {
171     delete sender;
172     delete receiver;
173 }

174 int myRouter::run()
175 {
176     sender.setSrc((short)src);
177     sender.setDst((short)dst);
178     error = sender.Start();
179     if (error) {
180         syslog(MsgPriority,"Thread to send failed.\n");
181         return;
182     }
183     error = receiver.Start();
184     if (error) {
185         syslog(MsgPriority,"Thread to receive failed.\n");
186         return;
187     }

188     // wait for the threads
189     pthread_join(sender->getId(),NULL);
190     pthread_join(receiver->getId(),NULL);
191 }

192 int myRouter::daemon(char* name,int facility)
193 {
194     switch (fork())
195     {
196         case 0: break;
197         case -1: return -1;
198         default: _exit(0);          // exit the original process
199     }

200     if (setsid() < 0)
201         return -1;
```

```
202     signal(SIGHUP,SIG_IGN); // ignore SIGHUP

203     switch (fork())
204     {
205         case 0: break;
206         case -1: return -1;
207         default: _exit(0);
208     }

209     chdir("/");

210     closeall(0);
211     open("/dev/null",O_RDWR);
212     dup(0);
213     dup(0);

214     openlog(name,LOG_PID,facility);
215     return 0;
216 }

217 /***** main program *****/
218 int main(int argc, char* argv[])
219 {
220     if (argc == 3) {
221         myRouter router(argv[1]);
222         router.setSrc(atoi(argv[2]));
223         router.setDst(atoi(argv[3]));
224         if (router.daemon(argv[0]) < 0) {
225             perror("daemon");
226             exit(1);
227         }
228         router.run();
229     }
230     else {
231         printf("Usage: %s <device> <src> <dst>\n",argv[0]);
232         exit(1);
233     }
234 }
```

代码说明如下。

22~27：定义 sendThr 类的成员变量如下。

- src：专用数据传输网络源地址。
- dst：专用数据传输网络目的地址。
- buf：发送缓冲区。



- capIP：myCapIP 对象指针，用于捕捉 IP 包。
- route：myRoute 对象指针，用于查询系统路由。
- device：myDevice 对象指针，用于发送数据到专用数据传输网络。

32：声明 sendThr 的成员函数 run()，用于实现从网络中捕捉 IP 并把它路由到专用数据传输网络。

39~58：实现 sendThr 类的构造函数，包括初始化参数，产生 myCapIP、myRoute 和 myDevice 对象，分配缓冲区空间。

59~65：实现 sendThr 的析构函数。

69~75：捕捉 IP 包。

77：取出 IP 包的目的地址。

78~82：通过字符串形式的目的地址查询路由，如果存在相应路由则重新捕捉 IP 包。

84：将 IP 包写入专用数据传输网络。

92~96：定义 recvThr 类的成员变量如下。

- src：专用数据传输网络源地址；
- dst：专用数据传输网络目的地址；
- buf：接收缓冲区；
- ip：myRaw 对象指针，用于发送 IP 包到网络；
- device：myDevice 对象指针，用于从专用数据传输网络接收数据包。

100：声明类的成员函数，用于从专用数据传输网络接收数据包并发送到网络上。

103~108：实现 recvThr 的构造函数。

109~114：实现 recvThr 的析构函数。

115~124：从专用数据传输网络接收 IP 包。

125~128：取出 IP 地址。

130~132：产生套接字地址。

134：通过原始套接字将 IP 包发往相应的套接字地址。

142~148：定义 myRouter 的成员变量如下。

- dev：串口名字，该串口与专用数据传输网络设备相连。
- src：专用数据传输网络源地址。
- dst：专用数据传输网络目的地址。
- sender：sendThr 对象指针。
- receiver：recvThr 对象指针。
- MsgPriority：消息优先级，用于守护进程的消息输出。

152~160：声明 myRouter 类的成员函数如下。

- run()：用于运行专用路由器；
- daemon()：用于产生守护进程。

176~177：设置专用数据传输网络源地址和目的地址。

178~182：启动发送线程。

183~187：启动接收线程。

189~190：等待发送线程和接收线程结束，以防主线程退出而导致所有线程退出。

192~216：产生守护进程。

221：产生 myRoute 对象，argv[1]是串口名字。

222：设置专用数据传输网络源地址，argv[2]是字符串形式的源地址，用 atoi()函数转换成整型数。

223：设置专用数据传输网络目的地址，argv[3]是字符串形式的目的地址，用 atoi()函数转换成整型数。

224~227：转换成守护进程。

228：运行路由器。

## 16.3 小 结

本章的实例是一个专用的 IP 路由器，它利用系统原有的路由功能实现两个基于 TCP/IP 的网络通过专用数据传输网络进行互连。

它是利用链路层访问技术获取 IP 包；利用原始套接字发送 IP 包；利用路由套接字查询系统路由；并且实现了以守护进程方式运行。

## 附录 A 套接字 Wrapper 类源程序

套接字 Wrapper 类是对 Unix 套接字相关的系统调用进行封装，形成 MySocket 类以便于使用。

MySocket 类封装了基础的套接字系统调用，并且实现异常处理。为了满足不同的需要，提供了多种形式的函数调用。

其头文件 mysocket.h 代码如下：

```
1 /* File : mysocket.h */
2 #include <unistd.h>          // Unix standard function definitions
3 #include <sys/types.h>       // Type definitions
4 #include <arpa/inet.h>       // Inet functions
5 #include <netinet/in.h>      // Structures defined by the internet system
6 #include <sys/socket.h>      // Definitions related to sockets
7 #include <sys/time.h>        // Time value functions

8 /* Define constant */
9 const unsigned MysBUF_SIZE = 1024;          // Fixed string buffer length
10 const unsigned MysMAX_NAME_LEN = 256;       // Maximum string name length
11 const unsigned MysRX_BUF_SIZE = 4096;       // Default receive buffer size
12 const unsigned MysTX_BUF_SIZE = 4096;       // Default transmit buffer size
13 const unsigned MAXCONN = 5;                 // Max connection
14 const unsigned MySOCKET_DEFAULT_PORT = 1234; // default port number

15 /* Define Error */
16 enum MySocketError {    // MySocket exception codes
17     MySOCKET_NO_ERROR = 0,          // No socket errors reported
18     MySOCKET_INVALID_ERROR_CODE,    // Invalid socket error code

19     // Socket error codes
20     MySOCKET_ACCEPT_ERROR,          // Error accepting remote socket
21     MySOCKET_BIND_ERROR,            // Could not bind socket
22     MySOCKET_BUF_OVERFLOW_ERROR,     // Buffer overflow
23     MySOCKET_CONNECT_ERROR,         // Could not connect socket
24     MySOCKET_FILESYSTEM_ERROR,      // A file system error occurred
25     MySOCKET_GETOPTION_ERROR,       // Error getting socket option
26     MySOCKET_HOSTNAME_ERROR,        // Could not resolve hostname
27     MySOCKET_INIT_ERROR,            // Initialization error
28     MySOCKET_LISTEN_ERROR,          // Listen error
29     MySOCKET_PEERNAME_ERROR,        // Get peer name error
```

```

30  MySOCKET_PROTOCOL_ERROR,          // Unknown protocol requested
31  MySOCKET_RECEIVE_ERROR,           // Receive error
32  MySOCKET_REQUEST_TIMEOUT,         // Request timed out
33  MySOCKET_SERVICE_ERROR,           // Unknown service requested
34  MySOCKET_SETOPTION_ERROR,         // Error setting socket option
35  MySOCKET_SOCKNAME_ERROR,          // Get socket name error
36  MySOCKET_SOCKETTYPE_ERROR,        // Unknown socket type requested
37  MySOCKET_TRANSMIT_ERROR,          // Transmit error
38  };

39  /*****
40  class name : MySocket
41  Function: A wrapper class of basic socket function.
42  *****/
43  class MySocket
44  {
45  protected: // Socket variables
46      sa_family_t address_family;     // Object's address family
47      int protocol_family;            // Object's protocol family
48      int socket_type;                // Object's socket type
49      int port_number;                // Object's port number
50      int Mysocket;                   // Socket this object is bound to
51      int conn_socket;                // Socket used for remote connections
52      MySocketError socket_error;     // The last reported socket error

53  protected: // Process control variables
54      int bytes_read;                 // Number of bytes read following a read operation
55      int bytes_moved;                // Number of bytes written following a write operation
56      int is_connected;               // True if the socket is connected
57      int is_bound;                   // True if the socket is bound

58  public: // Data structures used to set the internet domain and addresses
59      sockaddr_in sin;                // Sock Internet address
60      sockaddr_in remote_sin;         // Remote socket Internet address

61  public:
62      MySocket();
63      MySocket(int st, int port, char *hostname = 0);
64      MySocket(sa_family_t af, int st, int pf,
65              int port, char *hostname = 0);
66      virtual ~MySocket();

67  public: // Functions used to set the socket parameters
68      void SetAddressFamily(sa_family_t af) { address_family = af; }
69      void SetProtocolFamily(int pf) { protocol_family = pf; }

```

```
70  int GetProtocolFamily() {return protocol_family;}
71  void SetSocketType(int st) {socket_type = st;}
72  int GetSocketType() {return socket_type;}
73  void SetPortNumber(int p) {port_number = p;}
74  int GetBoundSocket() {return Mysocket;}
75  int GetSocket() {return Mysocket;}
76  int GetRemoteSocket() {return conn_socket;}

77 public: // Socket functions
78  int Socket();
79  int InitSocket(int st, int port, char *hostname = 0);
80  int InitSocket(sa_family_t af, int st,
81                int pf, int port, char *hostname = 0);
82  void Close();
83  void Close(int &s);
84  void CloseSocket();
85  void CloseRemoteSocket();
86  int Bind();
87  int Connect();
88  int Accept();
89  int Listen(int max_connections = MAXCONN);
90  void ShutDown(int how = 0);
91  void ShutDown(int &s, int how = 0);
92  void ShutDownSocket(int how = 0);
93  void ShutDownRemoteSocket(int how = 0);
94  int GetSockOpt(int s, int level, int optName,
95                void *optVal, unsigned *optLen);
96  int GetSockOpt(int level, int optName, void *optVal, unsigned *optLen);
97  int SetSockOpt(int s, int level, int optName,
98                const void *optVal, unsigned optLen);
99  int SetSockOpt(int level, int optName, const void *optVal,
                unsigned optLen);

100 //Stream function
101 int Recv(void *buf, int bytes, int flags = 0); //just for client
102 int Recv(int s, void *buf, int bytes, int flags = 0);
103 int Recv(void *buf, int bytes, int seconds, int useconds,
          int flags = 0); //just for client
104 int Recv(int s, void *buf, int bytes, int seconds, int useconds,
          int flags = 0);
105 int Send(const void *buf, int bytes, int flags = 0); //just for client
106 int Send(int s, const void *buf, int bytes, int flags = 0);
107 int RemoteRecv(void *buf, int bytes, int seconds, int useconds,
                int flags = 0); //just for server
```

```
108 int RemoteRecv(void *buf, int bytes, int flags = 0); //just for server
109 int RemoteSend(const void *buf, int bytes, int flags = 0);
110 void ResetRead() { bytes_read = 0; }
111 void ResetWrite() { bytes_moved = 0; }

112 // information database
113 int GetPeerName(int s, sockaddr_in *sa);
114 int GetPeerName();
115 int GetSockName(int s, sockaddr_in *sa);
116 int GetSockName();
117 int GetServByName(char *name, char *protocol = 0);
118 int GetServByPort(int port, char *protocol = 0);
119 servent *GetServiceInformation(char *name, char *protocol = 0);
120 servent *GetServiceInformation(int port, char *protocol = 0);
121 int GetPortNumber();
122 int GetRemotePortNumber();
123 int GetHostName(char *sbuf);
124 int GetIPAddress(char *sbuf);
125 int GetDomainName(char *sbuf);
126 int GetBoundIPAddress(char *sbuf);
127 int GetRemoteHostName(char *sbuf);
128 hostent *GetHostInformation(char *hostname);
129 void GetClientInfo(char *client_name, int &r_port);
130 sa_family_t GetAddressFamily();
131 sa_family_t GetRemoteAddressFamily();

132 // Process control functions
133 int ReadSelect(int s, int seconds, int useconds);
134 int BytesRead() { return bytes_read; }
135 int BytesMoved() { return bytes_moved; }
136 int SetBytesRead(int bytes = 0) { return bytes_read = bytes; }
137 int SetBytesMoved(int bytes = 0) { return bytes_moved = bytes; }
138 int *GetBytesRead() { return &bytes_read; }
139 int *GetBytesMoved() { return &bytes_moved; }
140 int IsConnected() { return is_connected == 1; }
141 int IsBound() { return is_bound == 1; }
142 int SetSocket(int s) { return Mysocket = s; }
143 int SetRemoteSocket(int s) { return conn_socket = s; }
144 void ReleaseSocket() { Mysocket = (int)-1; }
145 void ReleaseRemoteSocket() { conn_socket = (int)-1; }

146 // Datagram functions
147 int RecvFrom(int s, sockaddr_in *sa, void *buf,
148             int bytes, int seconds, int useconds, int flags = 0);
149 int RecvFrom(int s, sockaddr_in *sa, void *buf,
```

```

150         int bytes, int flags = 0);
151 int SendTo(int s, sockaddr_in *sa, void *buf,
152         int bytes, int flags = 0);
153 int RecvFrom(void *buf, int bytes, int flags = 0);
154 int RecvFrom(void *buf, int bytes, int seconds, int useconds,
155         int flags = 0);
156 int SendTo(void *buf, int bytes, int flags = 0);

156 // Exception handling functions
157 MySocketError GetSocketError() { return socket_error; }
158 MySocketError GetSocketError() const { return socket_error; }
159 MySocketError SetSocketError(MySocketError err) {
160     return socket_error = err;
161 }
162 MySocketError ResetSocketError() {
163     return socket_error = MySOCKET_NO_ERROR;
164 }
165 MySocketError ResetError() {
166     return socket_error = MySOCKET_NO_ERROR;
167 }

168 };

```

8~14：常量定义。

- MysBUF\_SIZE：固定字符串缓冲区长度；
- MysMAX\_NAME\_LEN：最大名字长度；
- MysRX\_BUF\_SIZE：默认接收缓冲区大小；
- MysTX\_BUF\_SIZE：默认发送缓冲区大小；
- MAXCONN：默认最大连接数（TCP 套接字）；
- MySOCKET\_DEFAULT\_PORT：默认端口号。

15~38：定义枚举类型 MySocketError，包括所有自定义错误码。

43~168：定义 MySocket 类，其成员函数包括以下几类。

- 构造/析构函数类；
- 参数设置/读取函数类。
- 套接字函数类。
- 读/写函数类。
- 信息函数类。
- 过程控制函数类。
- 数据报读/写函数类。
- 异常处理函数类。

MySocket 类的实现代码如下：

```
1 /* File : mysocket.cpp */
2 #include<string.h>
3 #include <fcntl.h>
4 #include "mysocket.h"

5 MySocket::MySocket()
6 // Socket constructor that performs no initialization other than
7 // setting default values for the socket data members.
8 {
9     address_family = AF_INET;           // Default address family
10    socket_type = SOCK_STREAM;           // Default socket type
11    protocol_family = IPPROTO_TCP; // Default protocol family
12    port_number = MySOCKET_DEFAULT_PORT; // Default port number
13    Mysocket = -1;
14    conn_socket = -1;
15    bytes_read = bytes_moved = 0;
16    is_connected = 0;
17    is_bound = 0;
18    socket_error = MySOCKET_NO_ERROR;
19 }

20 MySocket::MySocket(sa_family_t af, int st, int pf,
21                    int port, char *hostname)
22 // Socket constructor used to initialize the socket according to the
23 // address family, socket type, and protocol family. A hostname name should
24 // only be specified for client sockets.
25 {
26     Mysocket = -1;
27     conn_socket = -1;
28     bytes_read = bytes_moved = 0;
29     is_connected = 0;
30     is_bound = 0;
31     socket_error = MySOCKET_NO_ERROR;

32     // Initialize the socket. NOTE: Any errors detected during initialization
33     // will be recorded in the socket_error member.
34     InitSocket(af, st, pf, port, hostname);
35 }

36 MySocket::MySocket(int st, int port, char *hostname)
37 // Socket constructor used to initialize the socket according to the
38 // socket type. A hostname name should only be specified for client
39 // sockets.
40 {
41     Mysocket = -1;
```



```
42  conn_socket = -1;
43  bytes_read = bytes_moved = 0;
44  is_connected = 0;
45  is_bound = 0;
46  socket_error = MySOCKET_NO_ERROR;

47  // Initialize the socket. NOTE: Any errors detected during initialization
48  // will be recorded in the socket_error member.
49  InitSocket(st, port, hostname);
50 }

51 MySocket::~MySocket()
52 {
53     Close();
54 }

55 int MySocket::Socket()
56 // Create a socket. Returns a valid socket descriptor or
57 // -1 if the socket cannot be initialized.
58 {
59     Mysocket = socket(address_family, socket_type, protocol_family);
60     if(Mysocket < 0)
61     {
62         socket_error = MySOCKET_INIT_ERROR;
63         return -1;
64     }

65     return Mysocket;
66 }

67 int MySocket::InitSocket(sa_family_t af,
68                          int st,
69                          int pf,
70                          int port, char *hostname)
71 // Create and initialize a socket according to the address family,
72 // socket type, and protocol family. The "hostname" variable is an
73 // optional parameter that allows clients to specify a server name.
74 // Returns a valid socket descriptor or -1 if the socket cannot be
75 // initialized.
76 {
77     address_family = af;
78     socket_type = st;
79     protocol_family = pf;
80     port_number = port;
```

```
81 // Put the server information into the server structure.
82 sin.sin_family = address_family;

83 if(hostname) {
84     // Get the server's Internet address
85     hostent *hostnm = gethostbyname(hostname);
86     if(hostnm == (struct hostent *) 0) {
87         socket_error = MySOCKET_HOSTNAME_ERROR;
88         return -1;
89     }
90     // Put the server information into the client structure.
91     sin.sin_addr.s_addr = *((unsigned long *)hostnm->h_addr);
92 }
93 else
94     sin.sin_addr.s_addr = INADDR_ANY; // Use my IP address
95 sin.sin_port = htons(port_number);

96 // Create a TCP/IP
97 if(Socket() < 0) {
98     socket_error = MySOCKET_INIT_ERROR;
99     return -1;
100 }

101 return Mysocket;
102 }

103 int MySocket::InitSocket(int st, int port,
104     char *hostname)
105 // Create and initialize a socket according to the socket type. This
106 // cross-platform function will only accept SOCK_STREAM and SOCK_DGRAM
107 // socket types. The "hostname" variable is an optional parameter that
108 // allows clients to specify a server name. Returns a valid socket
109 // descriptor or -1 if the socket cannot be initialized.
110 {
111     address_family = AF_INET;
112     port_number = port;

113     if(st == SOCK_STREAM) {
114         socket_type = SOCK_STREAM;
115         protocol_family = IPPROTO_TCP;
116     }
117     else if(st == SOCK_DGRAM) {
118         socket_type = SOCK_DGRAM;
119         protocol_family = IPPROTO_UDP;
```

```
120     }
121     else {
122         socket_error = MySOCKET_SOCKETTYPE_ERROR;
123         return -1;
124     }

125     // Put the server information into the server structure.
126     sin.sin_family = address_family;

127     if(hostname) {
128         // Get the server's Internet address
129         hostent *hostnm = gethostbyname(hostname);
130         if(hostnm == (struct hostent *) 0) {
131             socket_error = MySOCKET_HOSTNAME_ERROR;
132             return -1;
133         }

134         // Put the server information into the client structure.
135         sin.sin_addr.s_addr = *((unsigned long *)hostnm->h_addr);
136     }
137     else
138         sin.sin_addr.s_addr = INADDR_ANY; // Use my IP address

139     sin.sin_port = htons(port_number);

140     // Create a TCP/IP socket
141     if(Socket() < 0) {
142         socket_error = MySOCKET_INIT_ERROR;
143         return -1;
144     }

145     return Mysocket;
146 }

147 int MySocket::Bind()
148 // Bind the socket to a name so that other processes can
149 // reference it and allow this socket to receive messages.
150 // Returns -1 if an error occurs.
151 {
152     int rv = bind(Mysocket, (struct sockaddr *)&sin, sizeof(sin));
153     if(rv >= 0) {
154         is_bound = 1;
155     }
156     else {
157         socket_error = MySOCKET_BIND_ERROR;
```

```
158     is_bound = 0;
159 }
160 return rv;
161 }

162 int MySocket::Connect()
163 // Connect the socket to a client or server. On the client side
164 // a connect call is used to initiate a connection.
165 // Returns -1 if an error occurs.
166 {
167     int rv = connect(Mysocket, (struct sockaddr *)&sin, sizeof(sin));
168     if(rv >= 0) {
169         is_connected = 1;
170     }
171     else {
172         socket_error = MySOCKET_CONNECT_ERROR;
173         is_connected = 0;
174     }
175     return rv;
176 }

177 int MySocket::ReadSelect(int s, int seconds, int useconds)
178 // Function used to multiplex reads without polling. Returns false if
179 // a reply time is longer then the timeout values.
180 {
181     struct timeval timeout;
182     fd_set fds;
183     FD_ZERO(&fds);
184     FD_SET(s, &fds);

185     timeout.tv_sec = seconds;
186     timeout.tv_usec = useconds;

187     // This function calls select() giving it the file descriptor of
188     // the socket. The kernel reports back to this function when the file
189     // descriptor has woken it up.
190     return select(s+1, &fds, 0, 0, &timeout);
191 }

192 int MySocket::Recv(void *buf, int bytes, int flags)
193 // Receive a block of data from the bound socket and do not return
194 // until all the bytes have been read. Returns the total number of
195 // bytes received or -1 if an error occurs.
196 {
197     return Recv(Mysocket, buf, bytes, flags);
198 }
```

```
198 }

199 int MySocket::Recv(void *buf, int bytes, int seconds, int useconds,
    int flags)
200 // Receive a block of data from the bound socket and do not return
201 // until all the bytes have been read or the timeout value has been
202 // exceeded. Returns the total number of bytes received or -1 if an
203 // error occurs.
204 {
205     return Recv(Mysocket, buf, bytes, seconds, useconds, flags);
206 }

207 int MySocket::Send(const void *buf, int bytes, int flags)
208 // Send a block of data to the bound socket and do not return
209 // until all the bytes have been written. Returns the total number
210 // of bytes sent or -1 if an error occurs.
211 {
212     return Send(Mysocket, buf, bytes, flags);
213 }

214 int MySocket::Recv(int s, void *buf, int bytes, int flags)
215 // Receive a block of data from a specified socket and do not return
216 // until all the bytes have been read. Returns the total number of
217 // bytes received or -1 if an error occurs.
218 {
219     bytes_read = 0;           // Reset the byte counter
220     int num_read = 0;         // Actual number of bytes read
221     int num_req = (int)bytes; // Number of bytes requested
222     char *p = (char *)buf;    // Pointer to the buffer

223     while(bytes_read < bytes) { // Loop until the buffer is full
224         if((num_read = recv(s, p, num_req-bytes_read, flags)) > 0) {
225             bytes_read += num_read; // Increment the byte counter
226             p += num_read;          // Move the buffer pointer for the next read
227         }
228         if(num_read < 0) {
229             socket_error = MySOCKET_RECEIVE_ERROR;
230             return -1; // An error occurred during the read
231         }
232     }

233     return bytes_read;
234 }

235 int MySocket::Recv(int s, void *buf, int bytes,
```

```
236     int seconds, int useconds, int flags)
237 // Receive a block of data from a specified socket and do not return
238 // until all the bytes have been read or the timeout value has been
239 // exceeded. Returns the total number of bytes received or -1 if an
240 // error occurs.
241 {
242     bytes_read = 0;           // Reset the byte counter
243     int num_read = 0;         // Actual number of bytes read
244     int num_req = (int)bytes; // Number of bytes requested
245     char *p = (char *)buf;    // Pointer to the buffer

246     while(bytes_read < bytes) { // Loop until the buffer is full
247         if(!ReadSelect(s, seconds, useconds)) {
248             socket_error = MySOCKET_REQUEST_TIMEOUT;
249             return -1; // Exceeded the timeout value
250         }
251         if((num_read = recv(s, p, num_req-bytes_read, flags)) > 0) {
252             bytes_read += num_read; // Increment the byte counter
253             p += num_read;          // Move the buffer pointer for the next read
254         }
255         if(num_read < 0) {
256             socket_error = MySOCKET_RECEIVE_ERROR;
257             return -1; // An error occurred during the read
258         }
259     }

260     return bytes_read;
261 }

262 int MySocket::Send(int s, const void *buf, int bytes, int flags)
263 // Send a block of data to a specified socket and do not return
264 // until all the bytes have been written. Returns the total number of
265 // bytes sent or -1 if an error occurs.
266 {
267     bytes_moved = 0;           // Reset the byte counter
268     int num_moved = 0;         // Actual number of bytes written
269     int num_req = (int)bytes; // Number of bytes requested
270     char *p = (char *)buf;    // Pointer to the buffer

271     while(bytes_moved < bytes) { // Loop until the buffer is empty
272         if((num_moved = send(s, p, num_req-bytes_moved, flags)) > 0) {
273             bytes_moved += num_moved; // Increment the byte counter
274             p += num_moved;          // Move the buffer pointer for the next read
275         }
276         if(num_moved < 0) {
```

```
277     socket_error = MySOCKET_TRANSMIT_ERROR;
278     return -1; // An error occurred during the read
279 }
280 }

281 return bytes_moved;
282 }

283 int MySocket::RemoteRecv(void *buf, int bytes, int seconds, int useconds,
284     int flags)
285 // Receive a block of data from a remote socket in blocking mode with
286 // a specified timeout value. Returns the total number of bytes received
287 // or -1 if an error occurs.
288 {
289     return Recv(conn_socket, buf, bytes, seconds, useconds, flags);
290 }

291 int MySocket::RemoteRecv(void *buf, int bytes, int flags)
292 // Receive a block of data from a remote socket in blocking mode.
293 // Returns the total number of bytes received or -1 if an error occurs.
294 {
295     return Recv(conn_socket, buf, bytes, flags);
296 }

297 int MySocket::RemoteSend(const void *buf, int bytes, int flags)
298 // Send a block of data to a remote socket and do not return
299 // until all the bytes have been written. Returns the total
300 // number of bytes received or -1 if an error occurs.
301 {
302     return Send(conn_socket, buf, bytes, flags);
303 }

304 void MySocket::ShutDown(int how)
305 // Used to close and un-initialize a full-duplex socket.
306 {
307     bytes_moved = 0;
308     bytes_read = 0;
309     is_connected = 0;
310     is_bound = 0;

311     if(Mysocket != -1) shutdown(Mysocket, how);
312     if(conn_socket != -1) shutdown(conn_socket, how);

313     Mysocket = -1;
314     conn_socket = -1;
```

```
315 }

316 void MySocket::ShutDown(int &s, int how)
317 // Used to close and un-initialize the specified full-duplex socket.
318 {
319     if(s != -1) shutdown(s, how);
320     s = -1;
321 }

322 void MySocket::ShutDownSocket(int how)
323 // Used to close a full-duplex server side socket.
324 {
325     if(Mysocket != -1) shutdown(Mysocket, how);
326     Mysocket = -1;
327 }

328 void MySocket::ShutDownRemoteSocket(int how)
329 // Used to close a full-duplex client side socket.
330 {
331     if(conn_socket != -1) shutdown(conn_socket, how);
332     conn_socket = -1;
333 }

334 void MySocket::Close()
335 // Close any and un-initialize any bound sockets.
336 {
337     bytes_moved = 0;
338     bytes_read = 0;
339     is_connected = 0;
340     is_bound = 0;

341     if(Mysocket != -1) close(Mysocket);
342     if(conn_socket != -1) close(conn_socket);

343 }

344 void MySocket::Close(int &s)
345 // Close the specified socket
346 {
347     if(s != -1) close(s);
348     s = -1;
349 }

350 void MySocket::CloseSocket()
```



```
351 // Close the server side socket
352 {
353     if(Mysocket != -1) close(Mysocket);
354     Mysocket = -1;
355 }

356 void MySocket::CloseRemoteSocket()
357 // Close the client socket
358 {

359     if(conn_socket != -1) close(conn_socket);

360     conn_socket = -1;
361 }

362 int MySocket::Listen(int max_connections)
363 // Listen for connections if configured as a server.
364 // The "max_connections" variable determines how many
365 // pending connections the queue will hold. Returns -1
366 // if an error occurs.
367 {
368     int rv = listen(Mysocket,          // Bound socket
369                     max_connections); // Number of connection request queue
370     if(rv < 0) socket_error = MySOCKET_LISTEN_ERROR;
371     return rv;
372 }

373 int MySocket::Accept()
374 // Accept a connect from a remote socket. An Accept()
375 // call blocks the server until the a client requests
376 // service. Returns a valid socket descriptor or -1
377 // if an error occurs.
378 {
379     // Length of client address
380     int addr_size = (int)sizeof(remote_sin);

381     conn_socket = accept(Mysocket, (struct sockaddr *)&remote_sin,
382                           &addr_size);

382     if(conn_socket < 0)
383     {
384         socket_error = MySOCKET_ACCEPT_ERROR;
385         return -1;
386     }
```

```
387     return conn_socket;
388 }

389 int MySocket::GetSockName(int s, sockaddr_in *sa)
390 // Retrieves the current name for the specified socket descriptor.
391 // It is used on a bound and/or connected socket and returns the
392 // local association. This function is especially useful when a
393 // connect call has been made without doing a bind first in which
394 // case this function provides the only means by which you can
395 // determine the local association which has been set by the system.
396 // Returns -1 if an error occurs.
397 {
398     int namelen = (int)sizeof(sockaddr_in);
399     int rv = getsockname(s, (struct sockaddr *)sa, &namelen);
400     if(rv < 0) socket_error = MySOCKET_SOCKNAME_ERROR;
401     return rv;
402 }

403 int MySocket::GetSockName()
404 // Retrieves the current name for this objects socket descriptor.
405 // Returns -1 if an error occurs.
406 {
407     return GetSockName(Mysocket, &sin);
408 }

409 int MySocket::GetPeerName(int s, sockaddr_in *sa)
410 // Retrieves the current name of the specified socket descriptor.
411 // Returns -1 if an error occurs.
412 {
413     int namelen = (int)sizeof(sockaddr_in);
414     int rv = getpeername(s, (struct sockaddr *)&sa, &namelen);
415     if(rv < 0) socket_error = MySOCKET_PEERNAME_ERROR;
416     return rv;
417 }

418 int MySocket::GetPeerName()
419 // Retrieves the current name for the remote socket descriptor.
420 // Returns -1 if an error occurs.
421 {
422     return GetPeerName(conn_socket, &remote_sin);
423 }

424 int MySocket::GetSockOpt(int s, int level, int optName,
425                          void *optVal, unsigned *optLen)
426 // Gets the current socket option for the specified option level or name.
```

```
427 // Returns -1 if an error occurs.
428 {

429     int rv = getsockopt(s, level, optName, optVal, (int *)optLen);
430     if(rv < 0) socket_error = MySOCKET_SETOPTION_ERROR;
431     return rv;
432 }

433 int MySocket::GetSockOpt(int level, int optName, void *optVal,
434                          unsigned *optLen)
435 // Gets the current socket option for the specified option level or name.
436 // Returns -1 if an error occurs.
437 {
438     return GetSockOpt(Mysocket, level, optName, optVal, optLen);
439 }

440 int MySocket::SetSockOpt(int s, int level, int optName,
441                          const void *optVal, unsigned optLen)
442 // Sets the current socket option for the specified option level or name.
443 // Returns -1 if an error occurs.
444 {

445     int rv = setsockopt(s, level, optName, optVal, (int)optLen);
446     if(rv < 0) socket_error = MySOCKET_SETOPTION_ERROR;
447     return rv;
448 }

449 int MySocket::SetSockOpt(int level, int optName, const void *optVal,
450                          unsigned optLen)
451 // Sets the current socket option for the specified option level or name.
452 // Returns -1 if an error occurs.
453 {
454     return SetSockOpt(Mysocket, level, optName, optVal, optLen);
455 }

456 servent *MySocket::GetServiceInformation(char *name, char *protocol)
457 // Function used to obtain service information about a specified name.
458 // The source of this information is dependent on the calling function's
459 // platform configuration which should be a local services file or NIS
460 // database. Returns a pointer to a servent data structure if service
461 // information is available or a null value if the service cannot be
462 // found. NOTE: The calling function must free the memory allocated
463 // for servent data structure upon each successful return.
464 {
465     // If the "protocol" pointer is NULL, getservbyname returns
```

```
466 // the first service entry for which the name matches the s_name
467 // or one of the s_aliases. Otherwise getservbyname matches both
468 // the name and the proto.
469 servent *sp = getservbyname(name, protocol);
470 if(sp == 0) return 0;

471 servent *buf = new servent;
472 if(!buf) return 0; // Memory allocation error
473 memmove(buf, sp, sizeof(servent));
474 return buf;
475 }

476 servent *MySocket::GetServiceInformation(int port, char *protocol)
477 // Function used to obtain service information about a specified port.
478 // The source of this information is dependent on the calling function's
479 // platform configuration which should be a local services file or NIS
480 // database. Returns a pointer to a servent data structure if service
481 // information is available or a null value if the service cannot be
482 // found. NOTE: The calling function must free the memory allocated
483 // for servent data structure upon each successful return.
484 {
485 // If the "protocol" pointer is NULL, getservbyport returns the
486 // first service entry for which the port matches the s_port.
487 // Otherwise getservbyport matches both the port and the proto.
488 servent *sp = getservbyport(port, protocol);
489 if(sp == 0) return 0;

490 servent *buf = new servent;
491 if(!buf) return 0; // Memory allocation error
492 memmove(buf, sp, sizeof(servent));
493 return buf;
494 }

495 int MySocket::GetServByName(char *name, char *protocol)
496 // Set service information corresponding to a service name and protocol.
497 // Returns -1 if an unknown service or protocol is requested. NOTE: This
498 // information is obtained from this machines local services file or
499 // from a NIS database.
500 {
501 // If the "protocol" pointer is NULL, getservbyname returns
502 // the first service entry for which the name matches the s_name
503 // or one of the s_aliases. Otherwise getservbyname matches both
504 // the name and the proto.
505 servent *sp = getservbyname(name, protocol);
506 if(sp == 0) {
```

```
507     socket_error = MySOCKET_PROTOCOL_ERROR;
508     return -1;
509 }
510 sin.sin_port = sp->s_port;
511 return 0;
512 }

513 int MySocket::GetServByPort(int port, char *protocol)
514 // Set service information corresponding to a port number and protocol.
515 // Returns -1 if an unknown service or protocol is requested. NOTE: This
516 // information is obtained from this machines local services file or
517 // from a NIS database.
518 {
519     // If the "protocol" pointer is NULL, getservbyport returns the
520     // first service entry for which the port matches the s_port.
521     // Otherwise getservbyport matches both the port and the proto.
522     servent *sp = getservbyport(port, protocol);
523     if(sp == 0) {
524         socket_error = MySOCKET_PROTOCOL_ERROR;
525         return -1;
526     }
527     sin.sin_port = sp->s_port;
528     return 0;
529 }

530 int MySocket::GetPortNumber()
531 // Return the port number actually set by the system. Use this function
532 // after a call to MySocket::GetSockName();
533 {
534     return ntohs(sin.sin_port);
535 }

536 int MySocket::GetRemotePortNumber()
537 // Return the port number of the client socket.
538 {
539     return ntohs(remote_sin.sin_port);
540 }

541 sa_family_t MySocket::GetAddressFamily()
542 // Returns the address family of this socket
543 {
544     return sin.sin_family;
545 }

546 sa_family_t MySocket::GetRemoteAddressFamily()
```

```
547 // Returns the address family of the remote socket.
548 {
549     return remote_sin.sin_family;
550 }

551 hostent *MySocket::GetHostInformation(char *hostname)
552 // Function used to obtain hostname information about a specified host.
553 // The source of this information is dependent on the calling function's
554 // platform configuration which should be a DNS, local host table, and/or
555 // NIS database. Returns a pointer to a hostent data structure
556 // if information is available or a null value if the hostname cannot
557 // be found. NOTE: The calling function must free the memory allocated
558 // for hostent data structure upon each successful return.
559 {
560     in_addr hostia;
561     hostent *hostinfo;
562     hostia.s_addr = inet_addr(hostname);

563     if(hostia.s_addr == NULL) { // Look up host by name
564         hostinfo = gethostbyname(hostname);
565     }
566     else { // Look up host by IP address
567         hostinfo = gethostbyaddr((const char *)&hostia,
568             sizeof(in_addr), AF_INET);
569     }
570     if(hostinfo == (hostent *) 0) { // No host name info available
571         return 0;
572     }

573     hostent *buf = new hostent;
574     if(!buf) return 0; // Memory allocation error
575     memmove (buf, hostinfo, sizeof(hostent));
576     return buf;
577 }

578 int MySocket::GetHostName(char *sbuf)
579 // Pass back the host name of this machine in the "sbuf" variable.
580 // A memory buffer equal to "MysMAX_NAME_LEN" must be pre-allocated
581 // prior to using this function. Return -1 if an error occurs.
582 {
583     // Prevent crashes if memory has not been allocated
584     if(!sbuf) sbuf = new char[MysMAX_NAME_LEN];
585     int rv = gethostname(sbuf, MysMAX_NAME_LEN);
586     if(rv < 0) socket_error = MySOCKET_HOSTNAME_ERROR;
587     return rv;
```

```
588 }

589 int MySocket::GetIPAddress(char *sbuf)
590 // Pass back the IP Address of this machine in the "sbuf" variable.
591 // A memory buffer equal to "MysMAX_NAME_LEN" must be pre-allocated
592 // prior to using this function. Return -1 if an error occurs.
593 {
594     char hostname[MysMAX_NAME_LEN];
595     int rv = GetHostName(hostname);
596     if(rv < 0) return rv;

597     in_addr *ialist;
598     hostent *hostinfo = GetHostInformation(hostname);
599     if(!hostinfo) {
600         socket_error = MySOCKET_HOSTNAME_ERROR;
601         return -1;
602     }
603     ialist = (in_addr *)hostinfo->h_addr_list[0];

604     // Prevent crashes if memory has not been allocated
605     if(!sbuf) sbuf = new char[MysMAX_NAME_LEN];

606     strcpy(sbuf, inet_ntoa(*ialist));
607     delete hostinfo;
608     return 0;
609 }

610 int MySocket::GetDomainName(char *sbuf)
611 // Pass back the domain name of this machine in the "sbuf" variable.
612 // A memory buffer equal to "MysMAX_NAME_LEN" must be pre-allocated
613 // prior to using this function. Return -1 if an error occurs.
614 {
615     char hostname[MysMAX_NAME_LEN];
616     int rv = GetHostName(hostname);
617     if(rv < 0) return rv;

618     hostent *hostinfo = GetHostInformation(hostname);
619     if(!hostinfo) {
620         socket_error = MySOCKET_HOSTNAME_ERROR;
621         return -1;
622     }
623     // Prevent crashes if memory has not been allocated
624     if(!sbuf) sbuf = new char[MysMAX_NAME_LEN];

625     strcpy(sbuf, hostinfo->h_name);
```

```
626 int i; int len = strlen(sbuf);
627 for(i = 0; i < len; i++) {
628     if(sbuf[i] == '.') break;
629 }
630 if(++i < len) {
631     len -= i;
632     memmove (sbuf, sbuf+i, len);
633     sbuf[len] = 0; // Null terminate the string
634 }
635 delete hostinfo;
636 return 0;
637 }

638 int MySocket::GetBoundIPAddress(char *sbuf)
639 // Pass back the local or server IP address in the "sbuf" variable.
640 // A memory buffer equal to "MysMAX_NAME_LEN" must be pre-allocated
641 // prior to using this function. Return -1 if an error occurs.
642 {
643     char *s = inet_ntoa(sin.sin_addr);
644     if(s == 0) {
645         socket_error = MySOCKET_HOSTNAME_ERROR;
646         return -1;
647     }

648     // Prevent crashes if memory has not been allocated
649     if(!sbuf) sbuf = new char[MysMAX_NAME_LEN];

650     strcpy(sbuf, s);
651     return 0;
652 }

653 int MySocket::GetRemoteHostName(char *sbuf)
654 // Pass back the client host name client in the "sbuf" variable.
655 // A memory buffer equal to "MysMAX_NAME_LEN" must be pre-allocated
656 // prior to using this function. Return -1 if an error occurs.
657 {
658     char *s = inet_ntoa(remote_sin.sin_addr);
659     if(s == 0) {
660         socket_error = MySOCKET_HOSTNAME_ERROR;
661         return -1;
662     }

663     // Prevent crashes if memory has not been allocated
664     if(!sbuf) sbuf = new char[MysMAX_NAME_LEN];
```



```
665 strcpy(sbuf, s);
666 return 0;
667 }

668 void MySocket::GetClientInfo(char *client_name, int &r_port)
669 // Get the client's host name and port number. NOTE: This
670 // function assumes that a block of memory equal to the
671 // MysMAX_NAME_LEN constant has already been allocated.
672 {
673     int rv = GetRemoteHostName(client_name);
674     if(rv < 0) {
675         char *unc = "UNKNOWN";
676         for(unsigned i = 0; i < MysMAX_NAME_LEN; i++) client_name[i] = '\0';
677         strcpy(client_name, unc);
678     }
679     r_port = GetRemotePortNumber();
680 }

681 int MySocket::RecvFrom(void *buf, int bytes, int seconds, int useconds,
682                        int flags)
683 // Receive a block of data from a remote datagram socket
684 // and do not return until all the bytes have been read
685 // or the timeout value has been exceeded. Returns the total
686 // number of bytes received or -1 if an error occurs.
687 {
688     return RecvFrom(Mysocket, &remote_sin, buf, bytes,
689                     seconds, useconds, flags);
689 }

690 int MySocket::RecvFrom(void *buf, int bytes, int flags)
691 // Receive a block of data from a remote datagram socket
692 // and do not return until all the bytes have been read.
693 // Returns the total number of bytes received or -1 if
694 // an error occurs.
695 {
696     return RecvFrom(Mysocket, &remote_sin, buf, bytes, flags);
697 }

698 int MySocket::SendTo(void *buf, int bytes, int flags)
699 // Send a block of data to a datagram socket and do not return
700 // until all the bytes have been written. Returns the total number
701 // of bytes sent or -1 if an error occurs.
702 {
703     return SendTo(Mysocket, &sin, buf, bytes, flags);
704 }
```

```
705 int MySocket::RecvFrom(int s, sockaddr_in *sa, void *buf,
706     int bytes, int seconds, int useconds, int flags)
707 // Receive a block of data from a remote datagram socket
708 // and do not return until all the bytes have been read
709 // or the timeout value has been exceeded. Returns the total
710 // number of bytes received or -1 if an error occurs.
711 {
712     // Length of client address
713     int addr_size = (int)sizeof(sockaddr_in);
714     bytes_read = 0;           // Reset the byte counter
715     int num_read = 0;         // Actual number of bytes read
716     int num_req = (int)bytes; // Number of bytes requested
717     char *p = (char *)buf;    // Pointer to the buffer

718     while(bytes_read < bytes) { // Loop until the buffer is full
719         if(!ReadSelect(s, seconds, useconds)) {
720             socket_error = MySOCKET_REQUEST_TIMEOUT;
721             return -1; // Exceeded the timeout value
722         }
723         if((num_read = recvfrom(s, p, num_req-bytes_read, flags,
724             (struct sockaddr *)sa, &addr_size)) > 0) {
725             bytes_read += num_read; // Increment the byte counter
726             p += num_read;          // Move the buffer pointer for the next read
727         }
728         if(num_read < 0) {
729             socket_error = MySOCKET_RECEIVE_ERROR;
730             return -1; // An error occurred during the read
731         }
732     }
733     return bytes_read;
734 }

734 int MySocket::RecvFrom(int s, sockaddr_in *sa, void *buf,
735     int bytes, int flags)
736 // Receive a block of data from a remote datagram socket
737 // and do not return until all the bytes have been read.
738 // Returns the total number of bytes received or -1 if
739 // an error occurs.
740 {
741     // Length of client address
742     int addr_size = (int)sizeof(sockaddr_in);
743     bytes_read = 0;           // Reset the byte counter
744     int num_read = 0;         // Actual number of bytes read
745     int num_req = (int)bytes; // Number of bytes requested
```

```
746 char *p = (char *)buf;    // Pointer to the buffer

747 while(bytes_read < bytes) { // Loop until the buffer is full
748     if((num_read = recvfrom(s, p, num_req-bytes_read, flags,
749         (struct sockaddr *)sa, &addr_size)) > 0) {
749         bytes_read += num_read; // Increment the byte counter
750         p += num_read;          // Move the buffer pointer for the next read
751     }
752     if(num_read < 0) {
753         socket_error = MySOCKET_RECEIVE_ERROR;
754         return -1; // An error occurred during the read
755     }
756 }
757 return bytes_read;
758 }

759 int MySocket::SendTo(int s, sockaddr_in *sa, void *buf,
760     int bytes, int flags)
761 // Send a block of data to a datagram socket and do not return
762 // until all the bytes have been written. Returns the total number
763 // of bytes sent or -1 if an error occurs.
764 {
765     // Length of address
766     int addr_size = (int)sizeof(sockaddr_in);
767     bytes_moved = 0;          // Reset the byte counter
768     int num_moved = 0;        // Actual number of bytes written
769     int num_req = (int)bytes; // Number of bytes requested
770     char *p = (char *)buf;    // Pointer to the buffer

771     while(bytes_moved < bytes) { // Loop until the buffer is full
772         if((num_moved = sendto(s, p, num_req-bytes_moved, flags,
773             (const struct sockaddr *)sa, addr_size)) > 0) {
773             bytes_moved += num_moved; // Increment the byte counter
774             p += num_moved;          // Move the buffer pointer for the next read
775         }
776         if(num_moved < 0) {
777             socket_error = MySOCKET_TRANSMIT_ERROR;
778             return -1; // An error occurred during the read
779         }
780     }
781     return bytes_moved;
782 }
```

代码说明如下。

5~50：实现 MySocket 的构造函数。

51~54：实现 MySocket 的析构函数。  
55~66：产生套接字。  
67~146：初始化套接字。  
147~161：实现套接字绑定功能。  
162~176：实现套接字连接操作。  
177~191：实现套接字读取操作的超时设定。  
192~206：实现套接字读取操作。  
207~213：实现套接字的发送操作。  
214~261：实现套接字接收操作，用于接收指定数目的字符。  
262~282：发送指定数目的字符。  
283~296：接收指定数目的字符，用于客户端操作。  
297~303：客户端数据发送。  
304~361：实现套接字关闭操作。  
362~372：实现套接字连接操作。  
373~388：实现套接字接收连接操作。  
389~408：获得套接字名字。  
409~423：获得远程套接字名字。  
424~439：获得套接字选项。  
440~455：设置套接字选项。  
456~494：获得服务信息。  
495~512：通过名字获取服务信息。  
513~529：通过端口号获得服务信息。  
530~535：获得本地端口号。  
536~540：获得远程端口号。  
541~545：获得本地地址簇。  
546~550：获得远程地址簇。  
551~577：通过主机名获得主机信息。  
578~588：获得主机名。  
589~609：获得 IP 地址。  
610~637：获得域名。  
638~652：获得绑定的 IP 地址。  
653~667：获得远程主机名。  
668~680：获得客户信息。  
681~697：数据报接收操作。  
698~704：数据报发送操作。  
705~733：数据报接收操作并具有超时返回功能。  
734~758：接收指定数目的字符。  
759~782：发送指定数目的字符。

## 附录 B 串口通信类源程序

SerialComm 类封装了串口通信功能，其头文件代码如下：

```
1 /* File : SerialComm.h */
2 typedef int scommDeviceHandle;

3 // Serial communications base class
4 class SerialComm
5 {
6 public: // Enumerations
7     enum {
8         // Internal error codes used to report the last error
9         scomm_NO_ERROR = 0,          // No errors reported
10        scomm_INVALID_ERROR_CODE, // Invalid error code

11        scomm_BAUDRATE_ERROR,        // Invalid baud rate
12        scomm_CS_ERROR,              // Invalid character size
13        scomm_FLOWCONTROL_ERROR,     // Invalid flow control
14        scomm_INIT_ERROR,            // Cannot initialize serial device
15        scomm_INVALIDPARM,           // Invalid initialization parameter
16        scomm_OPEN_ERROR,            // Cannot open serial device
17        scomm_PARITY_ERROR,          // Invalid parity
18        scomm_RECEIVE_ERROR,         // Serial device receive error
19        scomm_STOPBIT_ERROR,         // Invalid stop bit
20        scomm_TRANSMIT_ERROR,        // Serial device transmit error
21    };

22    enum {
23        // Flow control constants
24        scommHARD_FLOW,
25        scommSOFT_FLOW,
26        scommXON_XOFF,
27        scommNO_FLOW_CONTROL,

28        // Device access constants
29        scommREAD_ONLY,
30        scommWRITE_ONLY,
31        scommREAD_WRITE
32    };
```

```
33 public:
34     SerialComm();
35     virtual ~SerialComm();

36 public:
37     int OpenSerialPort(char *device_name);
38     int InitSerialPort();
39     int InitSerialPort(char *device_name, int sp = 9600, char pr = 'N',
40         int cs = 8, int sb = 1,
41         int flow = SerialComm::scommNO_FLOW_CONTROL,
42         int bin_mode = 1);
43     void Close();
44     int RawRead(void *buf, int bytes);
45     int RawWrite(const void *buf, int bytes);
46     int Recv(void *buf, int bytes);
47     int Send(const void *buf, int bytes);
48     void SetBaudRate(int br) { baud_rate = br; }
49     void SetCharacterSize(int cs) { character_size = cs; }
50     void SetParity(char p) { parity = p; }
51     void SetStopBits(int sb) { stop_bits = sb; }
52     void SetFlowControl(int f) { flow_control = f; }
53     scommDeviceHandle DeviceHandle() { return device_handle; }

54     void BinaryMode() { binary_mode = 1; }
55     void CharacterMode() { binary_mode = 0; }

56     termios *GetTermIOS() { return &options; }
57     int BytesRead() { return bytes_read; }
58     int BytesMoved() { return bytes_moved; }

59 protected:
60     scommDeviceHandle device_handle; // Device handle for the port
61     int baud_rate; // Baud rate
62     int character_size; // Character size
63     char parity; // Parity
64     int stop_bits; // Stop bits
65     int flow_control; // Flow control
66     int binary_mode; // True to enable raw reads and raw writes
67     int scomm_error; // The last reported serial port error

68 protected: // POSIX terminal (serial) interface extensions
69     termios options; // Terminal control structure
70     int bytes_read; // Number of bytes read following a Read() call
71     int bytes_moved; // Number of bytes written following a Write() call
```

```
72 };
```

代码说明如下。

7~21：定义错误码。

22~32：定义流控及读/写控制常量。

33~58：定义成员函数，包括以下几类。

- 构造/析构函数。
- 串口初始化/关闭函数。
- 串口读/写函数。
- 串口控制函数。

SerialComm 类实现代码如下：

```
1 /* File : SerialComm.cpp */
2 #include <unistd.h> // Unix standard function definitions
3 #include <termios.h> // POSIX terminal control definitions
4 #include <fcntl.h> // File control definitions
5 #include <string.h>
6 #include "SerialComm.h"

7 SerialComm::SerialComm()
8 {
9     // Default setting
10    baud_rate = 9600;
11    parity = 'N';
12    character_size = 8;
13    stop_bits = 1;
14    flow_control = SerialComm::scommNO_FLOW_CONTROL;
15    binary_mode = 1;
16    scomm_error = SerialComm::scomm_NO_ERROR;

17    memset(&options, 0, sizeof(options));
18 }

19 SerialComm::~SerialComm()
20 {
21    Close();
22 }

23 int SerialComm::OpenSerialPort(char *device_name)
24 // Open a serial device for read/write operations.
25 {
26    device_handle = open(device_name, O_RDWR | O_NOCTTY | O_NDELAY);

27    if(device_handle < 0)
```

```
28     device_handle = open(device_name, O_RDONLY | O_NOCTTY | O_NDELAY);
29     else
30         return scommREAD_WRITE;

31     if(device_handle < 0)
32         device_handle = open(device_name, O_WRONLY | O_NOCTTY | O_NDELAY);
33     else
34         return scommREAD_ONLY;

35     if(device_handle < 0) {
36         scomm_error = SerialComm::scomm_OPEN_ERROR;
37         return -1;
38     }
39     else
40         return scommWRITE_ONLY;

41 }

42 int SerialComm::InitSerialPort()
43 {
44     // Reset the port options
45     memset(&options, 0, sizeof(options));

46     // Set the baud rates
47     switch(baud_rate) {
48         case 0: // 0 baud (drop DTR)
49             cfsetispeed(&options, B0);
50             cfsetospeed(&options, B0);
51             break;
52         case 50:
53             cfsetispeed(&options, B50);
54             cfsetospeed(&options, B50);
55             break;
56         case 75:
57             cfsetispeed(&options, B75);
58             cfsetospeed(&options, B75);
59             break;
60         case 110:
61             cfsetispeed(&options, B110);
62             cfsetospeed(&options, B110);
63             break;
64         case 134:
65             cfsetispeed(&options, B134);
66             cfsetospeed(&options, B134);
67             break;
```



```
68     case 150:
69         cfsetispeed(&options, B150);
70         cfsetospeed(&options, B150);
71         break;
72     case 200:
73         cfsetispeed(&options, B200);
74         cfsetospeed(&options, B200);
75         break;
76     case 300:
77         cfsetispeed(&options, B300);
78         cfsetospeed(&options, B300);
79         break;
80     case 600:
81         cfsetispeed(&options, B600);
82         cfsetospeed(&options, B600);
83         break;
84     case 1200:
85         cfsetispeed(&options, B1200);
86         cfsetospeed(&options, B1200);
87         break;
88     case 1800:
89         cfsetispeed(&options, B1800);
90         cfsetospeed(&options, B1800);
91         break;
92     case 2400:
93         cfsetispeed(&options, B2400);
94         cfsetospeed(&options, B2400);
95         break;
96     case 4800:
97         cfsetispeed(&options, B4800);
98         cfsetospeed(&options, B4800);
99         break;
100    case 9600:
101        cfsetispeed(&options, B9600);
102        cfsetospeed(&options, B9600);
103        break;
104    case 19200:
105        cfsetispeed(&options, B19200);
106        cfsetospeed(&options, B19200);
107        break;
108    case 57600:
109        cfsetispeed(&options, B57600);
110        cfsetospeed(&options, B57600);
111        break;
112    case 115200:
```

```
113     cfsetispeed(&options, B115200);
114     cfsetospeed(&options, B115200);
115     break;
116     default:
117         scomm_error = SerialComm::scomm_BAUDRATE_ERROR;
118         return -1; // Invalid baud rate
119     }

120 // Set the character size, parity and stop bits
121 if((character_size == 8) && (parity == 'N') && (stop_bits == 1)) {
122     // No parity (8N1)
123     options.c_cflag &= ~PARENB;
124     options.c_cflag &= ~CSTOPB;
125     options.c_cflag &= ~CSIZE;
126     options.c_cflag |= CS8;
127     options.c_iflag = IGNPAR;
128 }
129 else if((character_size==7)&&(parity=='E')&&(stop_bits==1)){
130     // Even parity (7E1)
131     options.c_cflag |= PARENB;
132     options.c_cflag &= ~PARODD;
133     options.c_cflag &= ~CSTOPB;
134     options.c_cflag &= ~CSIZE;
135     options.c_cflag |= CS7;
136     options.c_iflag |= (INPCK | ISTRIP);
137 }
138 else if((character_size==7)&&(parity=='O')&&(stop_bits==1)){
139     // Odd parity (7O1)
140     options.c_cflag |= PARENB;
141     options.c_cflag |= PARODD;
142     options.c_cflag &= ~CSTOPB;
143     options.c_cflag &= ~CSIZE;
144     options.c_cflag |= CS7;
145     options.c_iflag |= (INPCK | ISTRIP);
146 }
147 else if((character_size==7)&&(parity=='M')&&(stop_bits==1)) {
148     // Mark parity is simulated by using 2 stop bits (7M1)
149     options.c_cflag &= ~PARENB;
150     options.c_cflag |= CSTOPB;
151     options.c_cflag &= ~CSIZE;
152     options.c_cflag |= CS7;
153     options.c_iflag |= (INPCK | ISTRIP);
154 }
155 else if((character_size==7)&&(parity=='S')&&(stop_bits==1)){
156     // Space parity is setup the same as no parity (7S1)
```

```
157     options.c_cflag &= ~PARENB;
158     options.c_cflag &= ~CSTOPB;
159     options.c_cflag &= ~CSIZE;
160     options.c_cflag |= CS7;
161     options.c_iflag |= (INPCK | ISTRIP);
162 }
163 else {
164     scomm_error = SerialComm::scomm_INVALIDPARM;
165     return -1; // Invalid character size, parity and stop bits combination
166 }

167     switch(flow_control) {
168         case scommHARD_FLOW :
169 #if defined(CRTSCTS)
170             options.c_cflag |= CRTSCTS;
171             break;
172 #else
173             break; // Hard flow control is not supported
174 #endif

175         case scommXON_XOFF :
176             options.c_iflag |= (IXON | IXOFF | IXANY);
177 #if defined(CRTSCTS)
178             options.c_cflag &= ~CRTSCTS;
179 #endif
180             break;

181         case scommSOFT_FLOW :
182 #if defined(CRTSCTS)
183             options.c_cflag &= ~CRTSCTS;
184             break;
185 #else
186             break; // Hard flow control is not supported
187 #endif

188         case scommNO_FLOW_CONTROL :
189 #if defined(CRTSCTS)
190             options.c_cflag &= ~CRTSCTS;
191             break;
192 #else
193             break; // Hard flow control is not supported
194 #endif

195         default:
196             scomm_error = SerialComm::scomm_FLOWCONTROL_ERROR;
```

```
197     return -1; // Invalid flow control
198 }

199 if(!binary_mode) { // Set the port for canonical input (line-oriented)
200     // Input characters are put into a buffer until a CR (carriage return)
201     // or LF (line feed) character is received.
202     options.c_lflag |= (ICANON | ECHOE);

203     // Postprocess the output.
204     // The ONLCR flag will map NL (linefeeds) to CR-NL on output.
205     // The OLCUC flag will map characters to upper case for tty output.
206     options.c_oflag = OPOST | ONLCR | OLCUC;
207 }
208 else { // Use raw input/output
209     // Input characters are passed through exactly as they are received,
210     // when they are received.
211     options.c_lflag = 0;
212     options.c_oflag = 0;
213 }

214 // Enable the receiver and set local mode
215 options.c_cflag |= (CLOCAL | CREAD);

216 // Initialize control characters if needed.
217 // Default values can be found in /usr/include/termios.h, and are given
218 // in the comments.

219 options.c_cc[VTIME] = 0; // inter-character timer unused
220 options.c_cc[VMIN] = 1; // blocking read until 1 character arrives

221 // Set the new options for the port. The TCSANOW constant specifies
222 // that all changes should occur immediately without waiting for
223 // output data to finish sending or input data to finish receiving.
224 tcflush(device_handle, TCIFLUSH); // Clean the serial line
225 tcsetattr(device_handle, TCSANOW, &options);

226 return 1; // No errors reported
227 }

228 void SerialComm::Close()
229 {

230     close(device_handle);

231 }
```

```
232 int SerialComm::RawRead(void *buf, int bytes)
233 // Read a specified number of bytes from the serial port
234 // and return whether or not the read was completed.
235 // Returns the number of bytes received or -1 if an
236 // error occurred.
237 {

238     bytes_read = read(device_handle, (char *)buf, bytes);
239     if(bytes_read < 0) {
240         scomm_error = SerialComm::scomm_RECEIVE_ERROR;
241         return -1;
242     }
243     return (int)bytes_read;
244 }

245 int SerialComm::RawWrite(const void *buf, int bytes)
246 // Write a specified number of bytes to a serial port
247 // and return whether or not the write was complete.
248 // Returns the total number of bytes moved or -1 if
249 // an error occurred.
250 {
251     bytes_moved = write(device_handle, (char *)buf, bytes);
252     if(bytes_moved < 0) {
253         scomm_error = SerialComm::scomm_TRANSMIT_ERROR;
254         return -1;
255     }
256     return (int)bytes_moved;
257 }

258 int SerialComm::Recv(void *buf, int bytes)
259 // Receive a specified number of bytes from a serial port
260 // and do not return until all the byte have been read.
261 // Returns the total number of bytes read or -1 if an
262 // error occurred.
263 {
264     int br = 0;                // Byte counter
265     int num_read = 0;          // Actual number of bytes read
266     int num_req = (int)bytes;  // Number of bytes requested
267     char *p = (char *)buf;    // Pointer to the buffer

268     while(br < bytes) { // Loop until the buffer is full
269         if((num_read = RawRead(p, num_req-br)) > 0) {
270             br += num_read; // Increment the byte counter
271             p += num_read;  // Move the buffer pointer for the next read
```

```
272     }
273     if(num_read < 0) {
274         scomm_error = SerialComm::scomm_RECEIVE_ERROR;
275         return -1; // An error occurred during the read
276     }
277 }

278 bytes_read = br; // Update the object's byte counter
279 return bytes_read;
280 }

281 int SerialComm::Send(const void *buf, int bytes)
282 // Write a specified number of bytes to a serial port and do
283 // not return until all the bytes have been written. Returns
284 // the total number of bytes written or -1 if an error occurred.
285 {
286     int bm = 0; // Byte counter
287     int num_moved = 0; // Actual number of bytes written
288     int num_req = (int)bytes; // Number of bytes requested
289     char *p = (char *)buf; // Pointer to the buffer

290     while(bm < bytes) { // Loop until the buffer is empty
291         if((num_moved = RawWrite(p, num_req-bm)) > 0) {
292             bm += num_moved; // Increment the byte counter
293             p += num_moved; // Move the buffer pointer for the next read
294         }
295         if(num_moved < 0) {
296             scomm_error = SerialComm::scomm_TRANSMIT_ERROR;
297             return -1; // An error occurred during the read
298         }
299     }

300     bytes_moved = bm; // Update the object's byte counter
301     return bytes_moved;
302 }

303 int SerialComm::InitSerialPort(char *device_name, int sp,
304     char pr, int cs, int sb, int flow, int bin_mode)
305 // Initialize a serial device using the specified parameters. Returns
306 // -1 if an error occurred during initialization or the current access
307 // mode of the port (scommREAD_WRITE, scommREAD_ONLY, scommREAD_WRITE).
308 {
309     int status = OpenSerialPort(device_name);
310     if(status < 0) return -1;
```

```
311  SetBaudRate(sp);
312  SetCharacterSize(cs);
313  SetParity(pr);
314  SetStopBits(sb);
315  SetFlowControl(flow);
316  if(bin_mode) BinaryMode(); else CharacterMode();

317  if(InitSerialPort() < 0) return -1;

318  return status;
319 }
```

代码说明如下。

7~22：实现构造/析构造函数。

23~41：打开串口。

42~227：初始化串口。

228~231：关闭串口。

232~244：串口读。

245~257：串口写。

258~280：读入指定数目字符。

281~302：写指定数目字符。

303~319：设置串口参数。