第一章 基本文件 I/O

引言：

学完本章内容以后，你将能够

- 掌握 Linux 文件的概念以及常用的文件操作方法
- 熟悉系统调用和 C 库函数基于流的方法来进行文件操作
- 了解两种文件操作方法的区别

1.1 文件与文件访问基本概念

1.1.1 Linux 文件

对普通计算机用户来说，文件是存储在永久性存储器上的一段数据流，通常是可执行程序或者是某种格式的数据。文件放置于文件夹，文件夹放置于某个磁盘分区中。这是从普通计算机用户眼里看到的文件。

但 Linux 操作系统中文件的概念，却远远不局限于此，文件是 Linux 对大多数系统资源访问的接口。Linux 常见的文件类型：普通文件、目录文件、设备文件、管道文件、套接字和链接文件等等。

(1)普通文件(regular file)，而前面提到的普通计算机用户看到的文件，仅仅是 Linux 文件类型中的一种，我们称之为普通文件，它们通常驻留在磁盘上的某处。普通文件按照信息存储方式来划分，可分为文本文件和二进制文件：

文本文件：这类文件以文本的某种编码(比如 ASCII 码)形式存储在存储器中。它是以“行”为基本结构的一种信息组织和存储方式。

二进制文件：这类文件以文本的二进制形式存储在计算机中，用户一般不能直接读懂它们，只有通过相应的软件才能将其显示出来。二进制文件一般是可执行程序、图形、图像、声音等等。

例如用 ls 命令查看一个普通文件：

```
[alex@alex~]$ls -l minicom.log
-rw-rw-r-- 1 alex alex 2903829 Apr 23 22:41 minicom.log
```

对于普通文件，可以看到有“-rw-rw-r--”的文件属性，第一个符号是“-”，这样的文件在 Linux 中就是普通文件。

(2)目录文件(directory)，主要目的是用于管理和组织系统中的大量文件。它存储一组相关文件的位置、大小等与文件有关的信息。

使用 ls 命令列出目录中的文件时，它打开该目录文件，并打印出他所包含的所有文件的信息：

```
[alex@alex~/spec]$ls -l
total 8784
- rw-r--r-- 1 alex alex 8970378 Dec 13 12:56 2800000 1.pdf
drwxrwxr-x 2 alex alex 4096 Nov 23 09:13 uart
```

目录文件“uart”的属性“drwxrwxr-x”，第一个字符是“d”。

(3)设备文件，Linux 操作系统把每一个 I/O 设备都看成一个文件，与普通文件一样处理，这样可以使文件与设备的操作尽可能统一，对 I/O 设备的使用和一般文件的使用一样，不必了解 I/O 设备的细节。设备文件可以细分为块设备文件(表示文件系统高速缓存的设备，

例如硬盘驱动器)和字符设备文件(表示非高速缓存的设备，例如磁带驱动器、鼠标和系统终端)。前者的存取是以一个个字符块为单位的，后者则是以单个字符为单位的。

下面用 ls 查看两个设备文件：

```
[alex@alex/dev]$ls -la tty
crw-rw-rw- 1 root root  5, 0  Apr 23 19:59 tty
[alex@alex/dev]$ls -la sda1
brw-r----- 1 root disk 8, 1 Apr 23 17:31 sda1
```

设备文件“/dev/tty”的属性是“crw-rw-rw-”，第一个字符是“c”，这表示字符设备文件。

设备文件“/dev/sda1”的属性是“brw-r---”，注意前面的第一个字符是“b”，这表示块设备文件。

(4)管道文件(named pipe)，主要用于在进程间传递数据。管道文件又称先进先出(FIFO)文件。管道只针对两个进程之间的通信而设计，建立管道的时候，实际获得两个文件描述符：一个用于读取而另一个用于写入。任何写入管道写入端的数据可以从读取端读出。

下面的例子，用 mknod 命令创建一个管道文件并用 ls 查看：

```
[alex@alex~]$mknod mypipe p
[alex@alex~]$ls -l mypipe
prw-rw-r-- 1 alex alex 0 Apr 24 09:14 mypipe
```

管道文件的属性是“prw—rw-r”，第一个字母“p”表示管道文件。

(5)套接字文件(socket)，类似于管道文件。管道文件用于本地通信，而套接字允许网络上的通信。

在“/tmp”临时目录下，经常会有一些套接字文件：

```
[alex@alex/tmp]$ls -l
srwxrwxr-x 1 alex alex  0 Apr 21 16:40 mapping-alex
srwxr-xr-x 1 root root  0 Mar 10 16:19 mapping-root
```

套接字文件属性“srwxr-xr-x”的第一个字母是“s”。

(6)符号链接文件(symbolic link)，这个文件包含了另一个文件的路径名。被链接的文件可以是任意文件或目录，可以链接不同文件系统的文件。链接文件甚至可以链接不存在的文件，这就产生一般称之为“断链”的问题，链接文件共至可以循环链接自己。

对于这个新的文件名，我们可以为之指定不同于被链接文件的访问权限，以控制对信息的共享和安全性的问题。

下面的例子创建并查看一个符号链接文件：

```
[alex@alex~]$ls -l minicom.log
-rw--rw-r-- 1 alex alex 2903794 Apr 23 15:01 minicom.log
```

```
[alex@alex~]$ln -sf minicom.log mlink
[alex@alex~]$ls -l mlink
lrwxrwxrwx 1 alex alex 4 Apr 23 22:32 mlink -> minicom.log
```

符号链接文件的属性“lrwxrwxrwx”的第一个字母是“l”。

上述是Linux丰富的文件类型，包括除了普通文件和目录文件之外的几种“特殊文件”，正是由于这些“特殊文件”的存在，Linux程序员可以按照统一的接口来实现基本文件读写、设备访问、硬盘读写、网络通信、系统终端，甚至内核状态信息的访问等等。无论是哪种类型的文件，Linux都把他们看作是无结构的流式文件，把文件的内容看作是一系列有序的字符流。

在Linux操作系统中，和文件操作相关的最基本元素是：目录结构、索引节点和文件的数据本身。

(1)目录结构。系统中的每一个目录都处于一定的目录结构当中。目录结构含有目录中所有目录的列表，每个目录项都含有一个文件名称和一个索引节点。借助于名称，应用程序可以访问目录项的内容，而索引节点提供了文件自身的信息。所以，目录只是将文件的名称和它的索引节点号结合在一起的一张表，目录中每一对文件名称和索引节点号称为一个连接。对于一个文件来说有唯一的索引节点号与之对应，对于一个索引节点号，却可以有多个文件名与之对应。因此，在磁盘上的同一个文件可以通过不同的路径去访问它。

(2)索引节点。目录结构包含文件名称和目录位置等信息，而索引节点本身并不包含这些信息，因为Linux允许使用多个文件名来引用磁盘上的同一块数据，多个文件名都可以访问同一个索引节点。索引节点包含了一个文件的访问权限、文件大小、文件最后更改的时间、文件的所有者和文件相关的特殊标志以及其他细节。

(3)文件的数据，它的存储位置由索引节点指定。有些特殊文件，比如管道及设备文件，在硬盘上不具有数据区域。而普通文件和目录都拥有数据区域。

后面我们将了解如何获取目录结构信息、从索引节点获取文件信息以及访问文件的数据。

1.1.2 文件访问基本概念

上一节我们提到和文件操作相关的一些基本元素：目录结构、索引节点和文件数据。通过opendir()及相关函数，可以获取目录结构信息。通过系统调用stat()可以从索引节点获得文件信息。通过常用的文件操作open()、read()等等可以访问文件的数据。Linux提供了丰富的文件访问接口，首先，我们需要了解Linux文件访问的实质，然后，才能灵活运用各种文件访问函数以及系统调用进行文件操作。

在Linux操作系统中，每一个文件都有一个唯一的索引节点，而每个索引节点可以有一个或者多个指向它的符号名，这些符号名就是文件的路径名。一个文件路径名只能指向一个索引节点(但一个索引节点可以有多个路径名)，因此文件路径名唯一的标识单个文件。比如

“/home/alex/myfile.txt”是一个文件路径名，指向的是“/home/alex”目录下的myfile.txt 文件，通过“/home/alex/myfile.txt”这个路径名只能访问到这个文件。

程序要访问一个文件，首先需要通过一个文件路径名打开文件。当进程打开一个文件的时候，进程将获得一个非负整数标识，即“文件描述符”。通过文件描述符，可以对文件进行 I/O 处理。

在 Linux 操作系统中各种类型的文件都采用统一的 I/O 方法来进行访问。因此，从磁盘中读取一个文件中的程序和从网络中读取数据的程序一样简单，而且，程序员可以使用统一接口来访问硬件设备和系统端口等等。

对文件执行 I/O 操作，有两种基本方式：一种是系统调用的 I/O 方法，另一种是基于流的 I/O 方法。

系统调用的 I/O 方法提供了最基本的文件访问接口，包括 open()、close()、write()、read()和 lseek()等。基于流的 I/O 方法实际上是建立在系统调用的 I/O 方法基础上的 C 函数库，它基于系统调用方法的封装并增加了额外的功能，例如采用缓冲技术来提高程序的效率、输入解析以及格式化输出等。然而在处理设备、管道、网络套接字和其他特殊类型的文件的时候，必须使用系统调用 I/O 方法。

系统调用 I/O 方法和基于流的 I/O 方法的区别是：

(1)基于流的文件操作函数的名字都是以字母“f”开头，而系统调用函数则不同。例如流函数 fopen()对应于系统调用的 open()。

(2)系统调用 I/O 方法是更低一级的接口，通常完成相同的任务是，比使用基于流的 I/O 方法需要更多编码的工作量。

(3)系统调用直接处理文件描述符，而流函数则处理“FILE*”类型的文件句柄。

(4)基于流的 I/O 方法是对系统调用方法的封装，流 I/O 方法使用自动缓冲技术，使程序能减少系统调用，从而提高程序的性能。

(5)基于流的 I/O 方法可以支持格式化输出，比如使用 fprintf()这样的函数。

(6)基于流的 I/O 方法替用户处理有关系统调用的细节，比如系统调用被信号中断的处理等等。

基于流的 I/O 方法显然给程序员提供了极大的方便，但是某些程序却不能使用基于流的 I/O 方法。比如使用缓冲技术使得网络通信陷入困境，因为它将干扰网络通讯所使用的通信协议。考虑到这两种 I/O 方法的不同，在使用终端或者通过文件交换信息时，通常采用基于流的 I/O 方法。而使用网络或者管道通信时，通常采用系统调用的 I/O 方法。

1.2 文件访问的系统调用 API

Linux 最常用的文件操作系统调用包括：打开、创建文件的 open()和 creat()，关闭文件 close()，读取文件 read()，写入文件 write()，移动文件指针 lseek()，文件控制 fcntl()和 access()等。

1.2.1 文件的创建、打开和关闭

通过 open()和 creat()系统调用，都可以创建一个并打开一个文件，系统调用 open()和 creat()成功时，都会返回一个非负数的文件描述符。使用 close()函数可以关闭指定的文件描述符的文件。

先看一个用 open()函数创建并打开一个新文件的例子。

示例 1-1：新建文件并打开

源文件：open_test.c

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

int main( int argc, char **argv )
{
    int fd;

    fd = open( "/tmp/open_test", O_CREAT|O_WRONLY|O_TRUNC, 0640 );
    if( fd == -1 ){
        perror( "create /tmp/open_test error!" );
    }else{
        printf("/tmp/open_test created, fd = [%d]\n", fd);
        close( fd );
    }
    return 0;
}
```

使用任何与文件相关的系统调用之前，程序应该包含<fcntl.h>和<unistd.h>头文件，它们为最普遍的文件例程提供了函数原型和常数。

代码中 open()函数传入了 3 个参数：

第一个参数是一个字符串参数，是要创建和打开的新文件的路径名。

第二个参数指明了 open 操作需要创建一个新文件，参数的具体含义我们稍后了解。

第三个参数指明了新建的文件的访问权限。

整数类型的变量 fd 记录了 open()函数的返回值。如果 fd 等于-1，那么表明 open()函数返回失败，否则 fd 记录了系统返回的所新建并打开的文件描述符。

在程序退出之前，使用 close()来关闭文件。

在上面一段代码中，也可以用 creat()函数来实现。把代码：

```
fd = open( "/tmp/open_test", O_CREAT|O_WRONLY|O_TRUNC, 0640 );
```

替换为：

```
fd = creat("/tmp/open_test ", 0640);
```

也就是说，使用 open()函数就实现创建一个新文件或者打开一个已经存在的文件。

下面我们来看 open()和 creat()的函数原型以及使用方法：

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

当 open()和 creat()调用成功，它会返回一个新的独一无二的文件描述符，这个新的文件描述符为其后对该文件的各种操作函数中使用。文件打开时，文件的读写指针被置于文件头。

open()函数必须指定打开文件的 flags 标志。表 2-1 中列出了各种标志及其作用。其中必须指定标志 O_RDONLY、O_WRONLY 和 O_RDWR 中的一个，其他标志都是可选的，可以将其于前面的三种标志之一进行或运算(OR)以生成最终的标志。

表 1-1 open 系统调用的标志

O_RDONLY	以只读方式打开文件
O_WRONLY	以只写方式打开文件
O_RDWR	以读写方式打开文件
O_APPEND	以追加模式打开文件，在每次写入操作指向之前，自动将文件指针定位到文件末尾。但在网络文件系统进行操作时却没有保证。
O_CREAT	如果指定的文件不存在，则按照 mode 参数指定的文件权限来创建文件。
O_DIRECTORY	假如参数 pathname 不是一个目录，那么 open 将失败。
O_EXCL	如果使用了这个标志，则使用 O_CREAT 标志来打开一个文件时，如果文件已经存在，open 将返回失败。但在网络文件系统进行操作时却没有保证。
O_NOCTTY	打开一个终端特殊设备时，使用这个标志将阻止它成为进程的控制终端。
O_NOFOLLOW	强制参数 pathname 所指的文件不能是符号链接。
O_NONBLOCK	打开文件后，对这个文件描述符的所有的操作都以非阻塞方式进行。
O_NDELAY	和O_NONBLOCK完全一样。
O_SYNC	当把数据写入到这个文件描述符时，强制立即输出到物理设备。
O_TRUNC	如果打开的文件是一个已经存在的普通文件，并且指定了可写标志(O_WRONLY、O_RDWR)，那么在打开时就消除原文件的所有内容。但打开的文件是一个 FIFO 或者终端设备时，这个标志将不起作用。

在文件打开后，可选参数可以通过 fcntl()系统调用来改变。

creat()相当于 open 使用了参数 flags 等于 O_CREAT()|O_WRONLY|O_TRUNC。

当 open()使用了 O_CREAT 标志或者是使用 creat 创建新文件时，参数 mode 指定了文件的访问权限。创建的文件通常也会被 umask 修改。所以一般新建文件的权限为(mode&~umask)。

注意：
文件权限只被应用于将来对这新文件的使用中，open()调用创建一个新的只读文件，但仍将返回一个可读写文件描述符。

open()和 creat()系统调用成功时，都会返回一个新的文件描述符，当返回失败时，将返回-1。通过 errno 以及使用 perror()可以查看错误信息。表 2-2 是常见的错误信息。

表 1-2 open()和 creat()常见错误信息

EACCES	访问请求不允许(权限不够)，在参数 pathname 中有目录不允许搜索，或者文件不存在且对上层目录的写操作又不允许。
EEXIST	使用了 O_CREAT 和 O_EXCL 标志，但文件已经存在。
EFAULT	文中在一个不能访问的地址空间。
EISDIR	参数 pathname 是一个目录，而又涉及到写操作。
ELOOP	在分解 pathname 时，遇到太多符号联接或者指明 O_NOFOLLOW 但是 pathname 是一个符号联接。
EMFILE	程序打开的文件数已经达到最大值了。
ENAMETOOLONG	文件名超长。
ENFILE	打开的总文件数已经达到了上限。
ENODEV	打开的文件是设备专用文件，而相应的设备不存在。
ENOENT	文件不存在或者是一个悬空的符号联接，或者是指定了 O_CREAT 标志但是路径中的目录不存在。
ENOMEM	可获得的核心内存不够。
ENOSPC	文件将要被创建，但是设备储存没有空间了。
ENOTDIR	参数 pathname 不是一个目录。
ENXIO	使用 O_NONBLOCK	O_WRONLY，命名的文件是 FIFO，但还没有进程创建一个 FIFO，或者打开一个设备专用文件而相应的设备不存在。
EROFS	文件是一个只读文件，但有写操作请求。
ETXTBSY	文件是一个正在被执行的可执行文件，又有写操作被请求。

当程序完成文件读写等操作之后，不再使用文件时，应当关闭文件。下面是 close()的函数原型：

```
int close(int fd);
```

close()系统调用关闭并释放一个文件描述符。close()成功时返回值等于 0，错误时返回-1。但是程序一般不需要检查 close 系统调用的返回值，除非发生严重的程序错误。

由于 Linux 内核对物理存储可能会有写延迟，所以就算成功的关闭了一个文件，也不能保证数据都被成功写入物理存储。当文件关闭时，对文件系统来说一般不去刷新缓冲区。

如果你要保证数据写入磁盘等物理存储设备中就使用 fsync()。fsync()函数原型是：

```
int fsync(int fd);
```

函数的参数是文件描述符。当数据成功同步到物理设备时，函数返回 0，否则返回-1。

1.2.2 文件读写

文件打开后，我们可以使用 read()和 write()来进行文件的读写操作。这两个系统调用的函数原型是：

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

三个参数分别是：

fd：要进行读写操作的文件描述符。

buf：要写入文件内容或读出文件内容的内存地址。

count：要读写的字节数。

read()是从文件描述符 fd 所引用的文件中读取 count 字节到 buf 缓冲区中。

注意：

必须提供一个合适的缓冲区，即 buf 的长度要大于或者等于 count。

如果 read()成功读取了数据，就返回所读取的字节数目，否则返回-1。如果 read()读到了文件的结尾或者被一个信号所中断，返回值会小于 count。当文件指针已经位于文件结尾，read()操作将返回 0。

下面的代码是从一个文本文件中凑出 100 个字节的例子,读取的文件名从程序的运行参数传入。

示例 1-2：用 read()函数从文件读取数据

源文件：read_test.c

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

int main( int argc, char **argv )
{
    int fd_read;
    char buf[100]={0};
    int ret;

    /*判断是否传入文件名*/
    if( argc < 2 ){
        printf("Usage: read_test FILENAME\n");
        return -1;
    }

    /*打开要读取的文件*/
    fd_read = open( argv[1], O_RDONLY );
    if( fd_read == -1 ){
        perror( "open error" );
        return -1;
    }

    /*读取数据*/
    ret = read( fd_read, buf, sizeof( buf )-1 );
    printf("read return = [%d]\n", ret );

    /*如果读出数据，则打印数据*/
    if( ret >= 0 ){
        printf("=========== buf ===========\n");
        printf("%s\n", buf );
        printf("==========================\n");
    }else{
        perror( "read error" );
    }
    close( fd_read );
    return 0;
}
```

当 read()返回-1 的时候，errno 以及使用 perror()可以查看读文件的错误信息。表 1-3 是常见的读错误信息。

我们经常需要检查的是 EINTR 错误，产生这个错误是由于 read()系统调用在读取任何数据前被信号所中断。发生这个错误的时候，文件指针并没有移动，我们需要重新读取数据。

表 1-3 read 常见错误信息

EAGAIN	使用 O_NONBLOCK 标志指定了非阻塞式输入输出，但当前没有数据可读。
EBADF	fd 不是一个合法的文件描述符，或者不是为读操作而打开。
EFAULT	buf 超出用户可访问的地址空间。
EINTR	在读取到数据以前调用被信号所中断。
EINVAL	fd 所指向的对象不适合读。或者是文件打开时指定了 O_DIRECT 标志。
EIO	输入输出错误。可能是正处于后台进程组进程试图读取其控制终端，但读操作无效，或者被信号 SIGTTIN 所阻塞，或者其进程组是孤儿进程组，也可能执行的是读磁盘或者磁带机这样的底层输入输出错误。
EISDlR	fd 指向一个目录。

write()从 buf 中写 count 字节到文件描述符 fd 所引用的文件中，成功时返回实际所写的字节数。

在实际的写入过程中，可能会出现写入的字节数少于 count。这时返回的是实际写入的字节数，所以调用 write()后都必须检查返回值是否与要写入的相同，如果不同就要采取相应的措施。

我们稍加修改，把前面读取数据的代码改为复制文件，即把读出的数据再写入新的文件中。也就是拷贝文件 cp 命令的一个简单实现。

示例 1-3：拷贝文件的简单实现

源代码：write_test.c

```
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char **argv )
{
    int fd_read;
    int fd_write;
    char buf[128]={0};
    int bytes_read, bytes_write;
```

```
    /*判断是否传入文件名*/
    if( argc < 3 ){
        printf("Usage: write_test FILEREAD FILEWRITE\n");
        return -1;
}

    /*打开要读取的文件*/
    fd_read = open( argv[1], O_RDONLY );
    if( fd_read == -1 ){
        perror( "open error" );
        return -2;
}
    /*创建一个新文件*/
    fd_write = open( argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0640 );
    if( fd_write == -1 ){
        perror( "create error" );
        close( fd_read );
        return -3;
}

    /*循环读取数据，并把每次读到的数据写入文件*/
    while( bytes_read = read( fd_read, buf, sizeof( buf )-1 ) ) {
        if( bytes_read > 0 ) {
            bytes_write = write( fd_write, buf, bytes_read );
            memset(buf,0,sizeof(buf));//请清空缓冲区
            /*write时候发生错误*/
            if( bytes_write == -1 )  break;
        }else if( ( bytes_read == -1 ) &&( errno != EINTR ) ){
            /*read错误发生了*/
            break;
        }
    }
    close( fd_read );
    close( fd_write );
    return 0;
}
```

这段代码实现了文件复制，但是并不完美。前面我们提到，write()写入数据的时候，应当检查已经写入的数据是否等于要等待写入的数据。那么我们增加一个 write_buffer 函数，来确保所有的数据都能完整写入文件：

```
int write_buffer(int fd, const void *buf, size_t count ) {
   int n = 0, r = count;

   while( count > 0 ) {
       n = write( fd, buf, count );
       if( n == 0 ) {
             r = 0; break;
       }
       if( n > 0 ) {
           count -= n;
           buf += n;
       } else if( n < 0 ) {
           /*由于中断信号造成的写函数返回，重新写入*/
           if( errno == EINTR )
               continue;
           r = -1;
           break;
       }
   }
   return r;
}
```

在 write_test.c 中，只要把

```
bytes_write = write(fd_write,buf,bytes_read);
```

替换为调用 write_buffer()函数

```
bytes_write = write_buffer (fd_write, buf, bytes_read);
```

这样就是一段经典的复制文件的代码了。

当 write()返回-1 的时候，errno 以及使用 perror()可以查看读文件的错误信息。表 2-4 是常见的读错误信息。和 read 一样，EINTR 错误是通常需要检查的。

表 1-4 Write 常见错误信息

EAGAIN	使用 O_NONBLOCK 标志指定了非阻塞式输入输出，但当前没有数据可读。
EBADF	fd 不是一个合法的文件描述符，或者不是为写操作而打开。
EFAULT	buf 超出用户可访问的地址空间。
EINTR	系统调用在写入任何数据之前调用被信号所中断。
EINVAL	fd 所指向的对象不适合写。或者是文件打开时指定了 O_DIRECT 标志。
EIO	当编辑一个节点时发生了底层输入输出错误。
ENOSPC	fd 指向的文件所在的设备无可用空间。
EPIPE	fd 连接到一个管道，或者套接字的读方向一端已关闭。此时写进程将接收到 SIGPIPE 信号， 如果此信号被捕获、阻塞或忽略，那么将返回错误 EPIPE。

1.2.3 文件的随机存取

每一个被打开的文件，都有一个文件指针表明当前的存取位置。一般文件被新建或者打开的时候，文件指针都位于文件头，除非在打开的时候指定了 O_APPEND 标志。文件的读写操作都是从文件指针位置开始的，每次文件的读取和写入，文件指针都会根据读写的字节数向后移动，直到文件结尾。

如果需要从文件的随机位置读写数据，那么就需要先移动文件指针。lseek()系统调用可以使文件指针移动到文件中的指定位置。下面是 lseek()的函数原型。

```
off_t lseek(int fildes,off_t offest,int whence);
```

lseek()的三个参数分别是：
fildes：文件描述符
offset：移动的偏移量，单位为字节数
whence：文件指针移动偏移量的解释，有三个选项，如表 2-5。

表1-5 lseek偏移量解释

SEEK_SET	从文件头开始计算，文件指针移动到 offset 个字节位置。
SEEK_CUR	从文件指针当前位置开始计算，向后移动 offset 个字节的位置。
SEEK_END	文件指针移动到文件结尾

lseek()移动文件指针成功时，将返回文件指针的当前位置。失败时返回-1。

下面的代码，通过把文件指针移动到文件结尾，lseek()将返回文件指针的位置，这是指针偏移量就是文件大小。经常可以通过这种方式来获得一个文件的大小。

示例 1-4：用 lseek()获取文件大小

源代码 lssek_test.c

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

int main( int argc, char **argv )
{
    int fd_lseek;
    char buf[100];
    int file_size;

    /*判断是否传入文件名*/
    if( argc < 2 ){
        printf("Usage: lseek_test FILENAME\n");
        return -1;
    }

    /*打开要读取的文件*/
    fd_lseek = open( argv[1], O_RDONLY );
    if( fd_lseek == -1 ){
        perror( "open error" );
        return -1;
    }

    /*/*移动指针到文件尾*/
    file_size = lseek( fd_lseek, 0, SEEK_END );
    if( file_size >= 0 ){
        printf("size of [%s] = %d\n", argv[1], file_size );
    }else{
        perror( "lseek error" );
    }
    close( fd_lseek );
    return 0;
}
```

编译运行这个小程序，结果如下：

```
[alex@alex-/tutotail]$gcc -o lseek_test lseek_test.c
[alex@alex-/tutotail]$./lseek_test lseek_test.c
size of [lseek_test.c]=622
[alex@alex-/tutotail]$ls lseek_test.c
-rw-rw-r--1 alex alex 622 May 13 14:50 lseek_test.c
```

可见，lseek()返回的指针位置等于文件的大小 622 字节。

1.2.4 文件的访问权限

Linux 有严格的文件权限控制，当我们需要在程序中判断一个文件是否具有读、写等权限的时候，可以使用 access()系统调用。

```
int access(const char *pathname, int mode);
```

参数 pathname 是要判断的文件路径名。参数 mode 可以是以下值或者是他们的组合：

R_OK：判断文件是否有读权限。
W_OK：判断文件是否有写权限。
X_OK：判断文件是否有可执行权限。
F_OK：判断文件是否存在。

当 access()系统调用对文件的测试成功时返回 0,只要有其中一个条件不符,则返回-1。下面是一个简单的例子，检查一个文件是否存在并具有可执行权限并输出检查结果。

示例 1-5：判断文件的访问权限

源代码：access_test.c

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

int main( int argc, char **argv )
{
    int ret;
```

/*判断是否传入文件名*/

```
    if( argc < 2 ){
        printf("Usage: access_test FILENAME\n");
        return -1;
    }
```

/*检查文件是否存在并具有可执行权限*/

```
    ret = access( argv[1], F_OK | X_OK );
```

```
    if( ret == -1 ){
        perror( "access:" );
    }else{
        printf("[%s] exist and executable!\n", argv[1]);
    }
    return 0;
}
```

1.2.5 修改文件属性

当文件被打开之后，进程会获得一个文件描述符，文件描述符包含了文件描述符标志以及当前进程对文件的访问权限等信息文件状态标志。当我们需要获取或者修改文件描述符中包含的标志时，可以使用 fcntl()系统调用。其函数原型时：

```
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

fcntl()是第一个参数 fd 是文件描述符，第二个参数指定了函数的操作，fcntl()函数常用的功能有：

复制一个文件描述符(cmd = F_DUPFD)。

获取/设置文件描述符标志(cmd = F_GETFD 或 cmd = F_SETFD)。

获取/设置文件状态标志(cmd = F_GETFL 或 cmd = F_SETFL)。

获取/设置文件锁(cmd = F_GETLK、cmd = F_SETLK 或 cmd = F_SETLKW)。

第五种功能中，fcntl()用于控制文件锁的时候，第三个函数参数是一个 struct flock 结构体。文件锁的具体使用，我们将在进程间通信的章节中再详细描述。其他 cmd 指定的操作的主要功能是：

F_DUPFD，复制文件描述符，新的文件描述符作为函数返回值返回。新的文件描述符与原文件描述符 fd 共同指向同一个文件，但是它有自己的一套文件描述符标　　志。

F_GETFD，获取 fd 对应文件描述符标志。目前只有一个可用的文件描述符标志 FD_CLOEXEC，当 FD_CLOEXEC 标志位等于 0 的时候，进程派生出的子进程用 exec 运行新的程序是，文件句柄被保留，子进程仍可使用这个文件句柄。FD_CLOEXEC 标志位等于 1 的时候文件描述符会被关闭。默认 FD_CLOEXEC 标志位等于 0。

F_SETFD，设置文件描述符标志，通过第三个参数设置。

F_GETFL，获取文件描述符对应的文件标志。文件标志在说明 open()函数的是已经介绍过了，可以查看表 2-1。

F_SETFL，设置文件描述符对应的文件标志。可以更改的标志包括：O_APPEND、O_ASYNC、O_DIRECT、O_NOATIME 和 O_ NONBLOCK。

fcntl()函数，根据 cmd 的不同，函数执行成功时返回值的含义各不相同，函数执行失败的时候，返回-1。

下面的函数是一个修改文件标志的例子，函数 set_fl 可以对文件描述符的文件标志进行打开或者关闭操作。

示例 1-6：修改文件标志

源代码：fcntl_test.c

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

void   set_fl( int fd, int flags, int on )
{
    int fl;
    if( fl = fcntl( fd, F_GETFL, 0 )  == -1 ) {
        perror( "fcntl" );
    }
    if( on ){
        /* turn flags on */
        fl |= flags;
    }else{
        /* turn flags off */
        fl&=~flags;
    }
    if( fcntl( fd, F_SETFL, fl ) == -1 ){
        perror( "fcntl" );
    }
}

int main( int argc, char **argv )
{
    int fd;

    fd = creat( "/tmp/fcntl_test", 0640 );
    if( fd == -1 ){
        perror( "create /tmp/open_test error!" );
    }
    printf("/tmp/open_test created, fd = [%d]\n", fd);

    set_fl( fd, O_APPEND, 1 );
```

```
    close( fd );
    return 0;
}
```

1.3 文件访问的 C 库函数

前面的章节已经介绍，C 函数库提供了丰富的基于流的文件操作函数。下面将介绍常用的 C 库中的文件操作函数，包折如何创建、打开和关闭文件，如何读写文件等等。

在了解 C 库的文件操作之前，我们先来了解文件句柄。基于流的文件操作都是对文件句柄进行操作。文件句柄指向一个包含文件信息的结构体，这些信息包括：缓冲区的地址、缓冲区中当前字符的位置、文件的访问模式等信息。但是程序员不需要关心这些细节，在头文件<stdio.h>里面定义了包含这些信息的结构体 FILE。在打开文件前声明一个文件句柄就可以了，比如：

```
#include<stdio.h>
FILE *fp;
```

1.3.1 文件创建、打开和关闭

不同于系统调用,C 库函数没有类似于 creat()的创建文件的函数。fopen()函数和 open()系统调用一样，既可以打开一个文件，也可以新建并打开一个文件。fopen 函数原型是：

```
FILE *fopen(const char *path, const char *mode);
```

第一个参数就是文件的路径名。

第二个参数表示文件的打开模式。表 2-6 列出了 fopen()可用的文件打开模式。

表 1-6 fopen 打开文件的模式

模式	模式说明
“r”	以只读方式打开文件，文件打开时指针位于文件首。
“r+”	以读和写方式打开文件，文件打开时指针位于文件首。
“w”	以只写方式打开一个文件，如果文件已经存在，清空文件内容，如果文件不存在， 则创建一个新的文件。文件打开时指针位于文件首。
“w+”	以读和写方式打开一个文件，如果文件已经存在，先清空文件内容，如果文件不 存在，则创建一个新的文件。文件打开时指针位于文件首。

“a”	以追加写的方式打开一个文件，如果文件不存在，则创建一个新文件，文件打开 时指针位于文件尾，而且在每次写操作前，文件指针自动先移动到文件尾。
“a+”	以追加写的方式打开一个文件，如果文件已经存在，先清空文件内容，如果文件 不存在，则创建一个新文件，文件打开时指针位于文件尾，而且在每次写操作前， 文件指针自动先移动到文件尾。

从 fopen()的 mode 参数中可以看到，只要以写的方式打开文件，如果文件不存在，fopen()会先创建一个新文件并打开。和 open 系统调用函数不同的是，fopen()函数不能设置新建的文件的读写权限。被创建的文件的访问权限会被设为组、用户和其他用户都可读写权限“S_RUSRIS_IWUSRIS_IRGRPIS_IWGRPIS_IROTHIS_IWOTH(0666)”，但是还要被用户的 umask 修改。所以一般新建文件的权限为(0666&~umask)。

如果成功的打开一个文件，fopen()函数返回文件句柄，否则返回空指针(NULL)。

当文件操作结束之后，需要关闭文件并释放文件句柄。用 fclose()函数来关闭一个由 fopen()函数打开的文件，其函数原型是：

```
int fclose(FILE *fp);
```

当文件关闭成功时，fclose()返回 0，否则返回一个非零值。

下面的例子，从程序参数传入一个文件名，然后用读写方式打开，如果传入的文件名不存在，则创建一个新文件，最后关闭文件退出程序。

示例 1-7：打开和关闭文件

源代码：fopen_test.c

```
#include <stdio.h>

int main( int argc, char **argv )
{
    FILE *fp;
```

```
    /*判断是否传入文件名*/
    if( argc != 2 ){
        printf("Usage: fopen_test FILENAME\n");
        return -1;
    }
```

```
    fp = fopen( argv[1], "w+" );
    if( fp == NULL ){
        perror( "fopen" );
    }else{
        printf("fopen [%s] success\n", argv[1] );
        if( fclose( fp ) != 0 ){
            perror( "fclose" );
        }else{
            printf("fclose [%s] success\n", argv[1] );
        }
    }
    return 0;
}
```

1.3.2 按字符读写文件

对于文本文件，我们经常需要按字符来进行读写。函数 fgetc()和 fputc()用于从指定文件中读出一个字符或把一个字符写入指定的文件。fgetc()和 fputc()分别还有另外一个别名 getc()和 putc()，它们的使用方法和功能是完全一样的，只是后者是前者的宏定义的别名。

我们先来看读字符操作，fgetc()和 getc()的函数原型是：

```
int fgetc(FILE* stream);
int getc(FILE* stream);
```

函数的参数是文件句柄。

fgetc()用 unsigned char 的格式来读取一个字符并映射为一个 int 值，如果读取正确，返回读取的字符的 int 值；否则，当读取错误或遇到文件结束标志 EOF 时，返回 EOF，EOF 在头文件的定义为-1。

下面的例子打开一个文件，按字符读取文件并打印到屏幕。

示例 1-8：读取文件

源代码：fgetc_test.c

```
#include <stdio.h>

int main( int argc, char** argv )
{
    FILE *fp;
    int c;
```

```

```

/*判断是否传入文件名*/

```
    if( argc != 2 ){
        printf("Usage: fgetc_test FILENAME\n");
        return -1;
    }
```

/*打开要读取的文件*/

```
    fp = fopen( argv[1], "r" );

    if( fp == NULL ){
        printf( "fgetc: can't open %s\n", argv[1] );
        return -2;
    }
```

/*按字符读取文件直到遇到文件结束符*/

```
    while( ( c = fgetc( fp ) ) != EOF ){
        putchar( c );
    }

    fclose( fp );
    return 0;
}
```

按字符写入文件 fputc()和 putc()函数的原型是：

```
int fputc(int c, FILE* stream);
int puct(int c, FILE* stream);
```

函数的第一个参数是要写入文件的字符，第二个参数是文件句柄。

如果字符被成功写入文件，函数返回写入的字符的值，写入失败则返回 EOF。

上一节，我们举过一个拷贝文件的例子。现在把 fputc_test.c 稍加修改，传入两个文件名，从第一个文件中逐一读取字符，同时写入第二个文件，这就是按字符复制版本的 cp 命令的实现。

示例 1-9：按字符复制文件

源代码：fputc_test.c

```
#include <stdio.h>

int main( int argc, char ** argv )
```

```
{
    FILE *fp_read, *fp_write;
    int c;

    /*判断是否传入文件名*/
    if( argc != 3 ){
        printf("Usage: fputc FILEREAD FILEWRITE\n");
        return -1;
    }

    /*打开要读取的文件*/
    fp_read = fopen( argv[1], "r" );

    if( fp_read == NULL ){
        printf( "fputc: can't open %s\n", argv[1] );
        return -2;
    }

    /*创建一个新文件*/
    fp_write = fopen( argv[2], "w" );
    if( fp_write == NULL ){
        printf( "fputc: can't open %s\n", argv[2] );
        fclose( fp_read );
        return -3;
    }

    /*按字符读取文件直到遇到文件结束符*/
    while( ( c = fgetc( fp_read ) ) != EOF ){
        fputc( c, fp_write );
    }

    fclose( fp_read );
    fclose( fp_write );
    return 0;
}
```

1.3.3 按字符串读写文件

有时候按字符来读写文件，程序的效率比较低，C 库提供了按字符串来读写文件的函数 fgets()和 fputs0。

Fgets()函数从文件中读入一行以“\n”或 EOF 结尾的字符串。Fgets 函数原型是：

```
char *fgets(char *s, int size, FILE *stream);
```

第一个参数是存放读出来的字符串的地址，第二个参数指定了一次最多读取 n 个字符，第三个参数指定了文件句柄。fgets()函数将从文件偏移指针的当前位置依次读取字符存入字符串指针 s 中。函数读取字符直至 n-1 个字符或遇到换行符或文件结束标志 EOF 为止。

如果执行成功，返回读取的字符串。当读文件发生错误，或者读到文件尾遇 EOF 并且没有读到任何字符时，则返回空指针。

示例 1-10：按字符串读取文件

源代码：fgets_test.c

```
#include <stdio.h>

int main( int argc, char** argv )
{
    FILE *fp;
    char buf[128];

    /*判断是否传入文件名*/
    if( argc != 2 ){
        printf("Usage: fgetc_test FILENAME\n");
        return -1;
    }

    /*打开要读取的文件*/
    fp = fopen( argv[1], "r" );

    if( fp == NULL ){
        printf( "fgets: can't open %s\n", argv[1] );
        return -2;
    }

    /*读取文件直到遇到文件结束符*/
    while( ( fgets( buf, sizeof( buf ), fp ) ) != NULL ){
        printf( "%s", buf );
```

```
    }

    fclose( fp );
    return 0;
}
```

写入字符串函数 futs()函数的原型是：

```
int fputs(const char *s, FILE *stream);
```

函数的第一个参数是要写入文件的字符串，第二个参数是文件句柄。如果字符串被成功写入文件，函数返回非负整数，写入失败则返回 EOF。

下面的例子又是一个 cp 命令的实现的新版本，按字符复制文件。

示例 1-11：按字符串复制文件

源代码 fputs_test.c

```
#include <stdio.h>

int main( int argc, char** argv )
{
    FILE *fp_read, *fp_write;
    char buf[128];

    /*判断是否传入文件名*/
    if( argc != 3 ){
        printf("Usage: fputc FILEREAD FILEWRITE\n");
        return -1;
    }

    /*打开要读取的文件*/
    fp_read = fopen(argv[1], "r");

    if( fp_read == NULL ){
        printf("fputc: can't open %s\n", argv[1]);
        return -2;
    }

    /*创建一个新文件*/
    fp_write = fopen( argv[2], "w" );
    if(fp_write == NULL){
        printf( "fputc: can't open %s\n", argv[2] );
```

```
        fclose( fp_read );
        return -3;
    }

    /*按字符读取文件直到遇到文件结束符*/
    while( ( fgets( buf, sizeof( buf ), fp_read ) ) != NULL ){
        fputs( buf, fp_write );
    }

    fclose( fp_read );
    fclose( fp_write );
    return 0;
}
```

1.3.4 按数据块读写文件

对于非文本文件，比如二进制文件等，当需要一次读写一组数据的时候，我们可以使用 fread()和 fwrite()函数来读写文件。

这两个函数的原型是：

```
size_t fread(void *ptr, size_t size，size_t nmemb，FILE *stream);
size_t fwrite(const void  *ptr, size_t size，size_t nmemb，FILE *stream);
```

第一个参数分别是要读写的数据的地址，第二个参数是数据块的大小，第三个参数是要读写的数据块的数目，第四个参数是文件句柄。

当读写操作成功时，函数返回成功读写的数据块的数量。如果返回的数量小于第三个参数指定的读写数量时，表明发生了错误或者读文件遇到了文件结束符 EOF。这种情况下，fread()无法区分读数据时是出错还是遇到文件结束符，通常可以使用 feof()函数来判断是否读到了文件尾，或者用 ferror()函数来判断是否是读文件出错。

下面是一个 fread()和 fwrite()读写文件的例子。首先从终端输入 1 0 个浮点数，然后把 10 个浮点数，以一个浮点数为数据块，分为 10 个数据块写入文件。然后再以 10 个浮点数为一个数据块，从文件读出数据并打印到屏幕。

示例 1-12：按数据块渎写文件

源代码：frwite_test.c

```
#include <stdio.h>

int main( int argc, char **argv )
{
    FILE *fp;
    float fa[10];
```

```
    int i;

    /*创建一个新文件*/
    fp = fopen( "test", "w" );
    if( fp == NULL ){
        perror( "fopen" );
        return -3;
    }

    /*终端输入10个浮点数*/
    for( i = 0; i < 10; i++ ){
        scanf( "%f", &fa[i] );
    }

    /*一个浮点数为数据块，分为10个数据块写入文件*/
    i = fwrite( fa, sizeof( float ), 10, fp );
    printf( "fwrite return = [%d]\n", i );

    /*以10个浮点数为一个数据块从文件读出数据*/
    i = fread( fa, sizeof( fa ), 1, fp );

    for( i = 0; i < 10; i++ ){
        printf( "%f\n", fa[i] );
    }
}
```

我们来看测试结果，编译执行这段程序，随意输入 10 个数值：

```
[alex@alex~/work/tutotail]$gcc -o fwrite_test fwrite_test.c
[alex@alex~/work/tutotail]$./fwrite_test
12.1
124
4352.2
3425
243
452
342
54
31
462
```

```
Fwrite return = [10]
12.100000
124.000000
4352.200195
3425.000000
243.000000
452.000000
342.000000
54.000000
31.000000
462.000000
```

从结果可以看到，从文件重新读出的数据和屏幕输入的数据是数值是一样的。

1.3.5 文件的格式化输入输出

前面我们提到过基于流的文件操作比系统调用文件操作增加的一些特性，其中包括格式化输入和输出。fprin()和 fscanf()函数可以实现文件的格式化读写，与最常用的 printf()和 scanf()用法非常相似，只是 scanf()和 printf()于标准输入和标准输出进行读写，而 fprintf()和 scanf()可以对文件进行读写。

fprintf()的函数原型是：

```
int fprintf(FILE *stream, const char *format, ...);
```

函数的第一个参数是文件句柄，后面的参数用法和 prinf()函数完全一样，包含格式化字符串以及要输入的数据参数。

当 fprintf()写入成功是，函数返回实际写入文件的字符数，如果写入错误，将返回一个负数。

下面我们把一个表格写入文件。

Company	Address	Project
Zeffi	BeiJing	DC2000
PuHong	ShenZhen	SC2410

假设表格的每个栏位最多是 10 个字符，且靠右对齐。下面的代码新建一个文件名“tab.dat”的文件，并将数据按照格式化写入文件。

示例 1-13：格式化输入

源代码：fprintf_test.c

```
#include <stdio.h>

int main( int argc, char **argv )
```

```
{
    FILE *fp;
    int i;
    char buf[9][10] = {"Company", "Address", "Project",
                       "Zeffi", "ShangHai", "DC2000",
                       "Puhong", "Shenzhen", "SC2410"};

    fp = fopen( "tab.dat", "w" );
    if( fp == NULL ){
        perror( "fopen" );
        return -1;
    }
    for(i=0; i<9; i+=3){
        fprintf( fp, "%10s%10s%10s\n", buf[i], buf[i+1], buf[i+2]);
    }

    fclose( fp );
    return 0;
}
```

fscanf()的函数原型是：

```
int fscanf(FILE *stream, const char *format, …);
```

第一个参数是文件描述符，后面的参数和 scanf()一样，包含格式化字符串以及存储出的参数。

下面的代码用fscanf()从 fprintf_test.c示例中生成的tab.dat 文件读出表格数据并打印输出。

示例 1-14：格式化输出

源代码：scanf_test.c

```
#include<stdio.h>

int main(int argc, char **argv)
{
        FILE*fp;
        int i;
        char buf[9][10];

        fp=fopen(“tab.dat”,”r”);
        if(fp==NULL){
                perror(“fopen”);
```

```
                return-1
          }
          for(i=0;i<9;i+=3){
                fscanf(fp,”%10s%10s%10s”,buf[i],buf[i+1],buf[i+2]);
                printf(“%10s%10s%10s\n”,buf[i],buf[i+1],buf[i+2]);
          }
          for(i=0;i<9,i+3=){
        }
        fclose(fp);
        return0;
}
```

编译执行这段代码，程序输出：

Company	Address	Project
Zeffi	BeiJing	SC2000
PuHong	ShenZhen	SC6000

用 fprintf()和 fscanf()函数读写文件非常方便，容易理解，但效率不高。其原因是，fscanf()读取的只是字符串，需将其转换成二进制才能存入内存；输出时，fprintf()又需将二进制数转换为字符串才能输出到文件。文件的读、写都需转换，花费时间较多。在要求速度的情况下最好采用 fread()和 fwrite()函数。

1.3.6 文件的随机存取

基于流的文件操作，每个打开的文件，也都有一个文件指针表明当前的存取位置。一般文件被新建或者打开的时候，文件指针都位于文件头，除非在打开的时候指定了“a”或者“a+”的模式，文件指针会在每次写入的时候自动把文件指针先移动到文件尾。文件的读写操作都是从文件指针位置开始的，每次文件的读取和写入，文件指针都会根据读写的字节数向后移动，直到文件结尾。

如果需要从文件的随机位置读写数据，那么就需要先移动文件指针。fseek()函数提供了非常类似与 lseek()系统调用方法，可以使文件指针移动到文件中的指定位置。下面是 fseek()的函数原型。

```
int fseek(FILE *stream, long off’set, int whence);
```

fseek()的三个参数分别是：

stream：文件句柄

offset：指针移动的偏移量，单位为字节数

whence：文件指针移动偏移量的解释，有三个选项，如表 2-7。

表 1-7 fseek 偏移量解释

SEEK_SET	从文件头开始计算，文件指针移动到 offset 个字节的位置。
SEEK_CUR	从文件指针当前位置开始计算，向后移动 offset 个字节的位置。
SEEK_END	文件指针移动到文件结尾.

fseek()移动文件指针成功时，将返回 0。失败时返回-1。

C 库还提供了获得当前指针位置的函数 ftell()：

```
long ftell(FILE *stream);
```

ftell()的参数是文件句柄，函数执行成功时返回文件指针的当前位置。失败时返回-1。

还有两个函数可以对文件指针进行操作：

```
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, fpos_t *pos);
```

这两个函数分别可以获取和设置文件指针的位置。文件指针的设置或返回都通过函数的第二个参数 pos 来传递。fgetpos()函数和 fseek()指定了 SEEK_SET 的参数时的功能时完全一样的。

我们修改一下上一节读取表格的例子，通过 fseek()移动文件指针，忽略第一行表头。

示例 1-15：移动文件指针

源代码：feek_test.c

```
#include<stdio.h>
int      main(int argc,char**argv)
{
        FILE*fp;
        int i;
        char buf[9][10];

        fp=fopen(“tab.dat”,”r”);
        if(fp==NULL){
        perror(“fopen”);
        return-1;
        }
        /*把文件指针移动 30 个字符*/
        fseek(fp,30,SEEK_SET);

        /*输出当前的文件指针位置*/
        printf(“current file position[%|d]\n”,ftell(fp));

       for(i=0;i<6;i+=3){
             fscanf(fp,”%10s%10s%10s”,buf[i],buf[i+1],buf[i+2]);
             printf(”%10s%s10s%10s\n”,buf[i],buf[i+1],buf[i+2]);
```

```
        }
        fclose(fp);
        return0;
}
```

测试题

1.1 标准输入，标准输出，标准错误的文件描述符分别是_____，_____，_____。

1.2 标准库函数 getchar 的返回值地类型_____，EOF 的值_____。

1.3 FILE *fp 的文件描述符为_____。

本章总结

本章主要介绍 Linux 文件和文件访问的基本概念，详细介绍了常用的文件操作相关的系统调用和 C 库函数。

第二章 进程和线程

引言：

学完本章内容以后，你将能够

了解进程与线程的基本概念

编写多进程和多线程程序来运行其他可执行程序

了解信号的使用，以及使用信号异步清理子进程

了解并能够进行多线程同步编程

强化并行性的概念:同步和死锁

了解进程和线程的区别

2.1 进程和线程基本概念

2.1.1 什么是进程

程序是一个包含可执行指令的文件，而进程是一个开始执行但还没有结束的程序的实例。进程的标准定义是：“进程是一个具有独立功能的程序关于某个数据集合的一次可以并发执行的运行活动，是处于活动状态的计算机程序”。

进程是一个在执行过程不断变化的实体。和程序要包含指令和数据一样，进程也包含程序计数器和所有 CPU 寄存器的值，同时它在堆栈中存储着如子程序参数、返回地址以及变量之类的临时数据。Linux 下一个进程在内存里有三部分的数据：“代码段”、“堆栈段”和“数据段”。“代码段”存放程序代码的数据，假如系统中有数个进程运行相同的一个程序，那么他们就可以使用相同的代码段。“堆栈段”存放的就是子程序的返回地址、子程序的参数以及程序的局部变量。而数据段则存放程序的全局变量、常数以及动态数据分配的数据空间。系统如果同时运行数个相同的程序，它们之间就不能使用同一个堆栈段和数据段。

在进程的整个运行期间，它将会用到各种系统资源，会用到 CPU 运行它的指令，需要物理内存保存它的数据，它可能打开和使用各种文件，直接或间接地使用系统中的各种物理设备。系统中最为宝贵的资源是 CPU，因为一般情况下一个系统只有一个 CPU。Linux 是一个多进程的操作系统，Linux 内核为每个进程指派一定的运行时间，这个时间通常很短，短到毫秒为单位，然后依照某种规则，从多个进程中挑选一个运行，其他的进程暂时等待，当正在运行的那个进程时间耗尽，或执行完毕退出。当正在运行的进程等待其他的系统资源时，Linux 内核将取得 CPU 的控制权，并将 CPU 分配给其他正在等待的进程。因为每个进程占用的时间都很短，从使用者的角度来看，就好像多个进程同时运行一样了。Linux 系统内核必须了解进程本身的情况和进程所用到的各种资源，以便在多个进程之间合理地分配系统资源。

Linux 是一个多处理操作系统。进程具有独立的权限与职责。如果系统中某个进程崩溃，它不会影响到其余的进程。每个进程运行在其各自的虚拟地址空间中，进程之间可以通过由内核控制的机制相互通讯。

用 ps 命令，可以看到系统有多少进程正在运行，比如这是“ps -aux“命令看到的进程列表：

PID	Uid	VmSize	Stat	Command
1	root	472	S	init
2	root		SWN	[ksoftirqd/0]
3	root		SW<	[events/0]
4	root		SW<	[khelper]
17	root		SW<	[kblockd/0]
18	root		SW	[khubd]
28	root		SW	[kapmd]

```
  31     root             DW     [charged]
  33     root             SW     [pdflush]
  34     root             SW     [pdflush]
  36     root             SW<    [ai0/0]
  35     root             SW     [kswapd0]
 624     root             SW     [kseriod]
 638     root             DW     [v05-dma-worker]
 644     root             SW     [mtdblockd]
 646     root             SW     [pccardd]
 660     root             SW     [kjournald]
 666     root     1236    S      /mnt/gui/arm/bin/hammerhead
 682     root      772    S      -sh
 690     root      512    S      ./esd -as 0
 691     root    10676    S      ./desktop
 700     root             SWN    [kwm97xx_ts]
 706     root    10676    S      ./desktop
 710     root     4136    S N    quicklauncher
 714     root      708    S      /sbin/cardmgr
 727     root      676    R      ps -ax
```

除了第一行为标题行之外，每一行表示一个进程。在各个 Linux 版本中，看到的列可能不完全一样，常见的列包含了进程的 ID、进程的用户 ID、进程状态和进程的 Command 等等信息。

```
USER    进程的属主；
PID 进程的 ID；
PPID  父进程；
%CPU    进程占用的 CPU 百分比；
%MEM    占用内存的百分比；
NI     进程的 NICE 值，数值大，表示较少占用 CPU 时间；
VSZ   进程虚拟大小；
RSS   驻留中页的数量；
TTY   终端 ID
STAT  进程状态

D    Uninterruptible sleep (usually IO)
R    正在运行可中在队列中可过行的；
```

S 处于休眠状态；
T 停止或被追踪；
Z 僵尸进程；

< 优先级高的进程
N 优先级较低的进程
L 有些页被锁进内存；
s 进程的领导者（在它之下有子进程）；
l is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
+ 位于后台的进程组；

WCHAN 正在等待的进程资源；
START 启动进程的时间；
TIME 进程消耗 CPU 的时间；
COMMAND 命令的名称和参数；

这些信息保存在 Linux 内核为每个进程都创建一个叫做 task_struct(定义在 Linux 源代码 include/Linux/sched.h)的数据结构中，而所有指向这些数据结构的指针组成系统中的一个进程向量数组 task。task 数组是 Linux 操作系统管理线程的基础。

2.1.2 tast struct 简介

前面我们提到，为了让 Linux 来管理系统中的进程，每个进程用一个 task_struct 数据结构来表示。数组 task 包含指向系统中所有 task_struct 结构的指针。创建新进程时，Linux 将从系统内存中分配一个 task_struct 结构并将其加入 task 数组。系统中还有一个当前进程的指针，用来指向当前运行进程的结构。

task_struct 数据结构庞大而复杂，但它可以分成一些功能组成部分：

(1)进程状态

进程在执行过程中会根据环境来改变状态。Linux 进程有以下状态：运行状态、等待状态、停止状态和僵尸状态。下一小节将介绍各种状态及切换。

(2)进程调度信息

调度器需要这些信息以便判定系统中哪个进程最迫切需要运行。

(3)标识符

系统中每个进程都有进程标志。进程标志并不是 task 数组的索引，它仅仅是个数字。每个进程还有一个用户与组标志，它们用来控制进程对系统中文件和设备的存取权限。

(4)进程间通信

Linux 支持经典的 Unix IPC 机制，如信号、管道和信号灯以及系统 V 中 IPC 机制，包括共享内存、信号灯和消息队列。我们将在下一章中详细讨论 Linux 中进程间通信机制。

(5)链接

Linux 系统中所有进程都是相互联系的。除了初始化进程外，所有进程都有一个父进程。新进程不是被创建，而是被复制，或者从以前的进程克隆而来。每个进程对应的 task_struct 结构中包含有指向其父进程和兄弟进程（具有相同父进程的进程）以及子进程的指针。我们可以使用 pstree 命令来观察 Linux 系统中运行进程间的关系，比如：

```
[alex@alex ~]# pstree -p
init(1)-+-acpid(2035)
        |-atd(2266)
        |-auditd(1685)-+-python(1687)
        |              `-{auditd}(1686)
        |-automount(2011)-+-{automount}(2012)
        |                 |-{automount}(2013)
        |                 |-{automount}(2016)
        |                 `-{automount}(2019)
        |-avahi-daemon(2302)---avahi-daemon(2303)
……
……
```

另外，系统中所有进程都有一个双向链表连接起来，而它们的根是 init 进程的 task_struct 数据结构。这个链表被 Linux 核心用来寻找系统中所有进程。

（6）时间和定时器

核心需要记录进程的创建时间以及在其生命期中消耗的 CPU 时间。时钟每跳动一次，核心就要更新保存在 jiffies 变量中，记录进程在系统和用户模式下消耗的时间量。Linux 支持与进程相关的 interval 定时器，进程可以通过系统调用来设定定时器以便在定时器到时后向它发送信号。这些定时器可以是一次性的或者周期性的。

(7)文件系统

进程可以自由地打开或关闭文件，进程的 task_struct 结构中包含一个指向每个打开文件描述符的指针以及两个指向 VFS 索引节点的指针。每个 VFS 索引节点唯一地表示文件中的一个目录或者文件，同时还对底层文件系统提供统一的接口。第一个索引节点是进程的根目录，第二个节点是当前的工作目录。两个 VFS 索引节点都有一个计数字段用来表明指向节点的进程数，当多个进程引用它们时，它的值将增加。

(8)虚拟内存

多数进程都有一些虚拟内存(核心线程和后台进程没有)，Linux 核心必须跟踪虚拟内存与系统物理内存的映射关系。

(9)处理器的内容

进程可以认为是系统当前状态的总和。进程运行时，它将使用处理器的寄存器以及堆栈等等。进程被挂起时，进程的上下文，所有的 CPU 相关的状态必须保存在它的 task_struct 结构中。当调度器重新调度该进程时，所有上下文被重新设定。

2.1.3 进程状态及状态切换

进程在生存周期中的其状态是变化的。下面是 Linux 操作系统的进程的几种常见的状态：

- ✓ 运行状态，此状态下进程正在运行(即系统的当前进程)或者是准备运行状态(即就绪状态)。在ps命令列出的状态列Stat列中，字母R表示运行状态。
- ✓ 等待状态，进程正在等待事件的发生或者等待某种系统资源。Linux操作系统中的等待进程分为可中断等待和不可中断等待。可中断等待进程可以被信号（signal）中断，ps命令看到的中断中，字母S表示程序处于可中断等待状态。不可中断状态进程不受信号干扰，直到硬件状态改变，通常是处于I/O操作过程中，字母D表示进程处于不可中断等待状态。
- ✓ 停止状态，进程被停止，通常是收到了一个控制信号或者正在被跟踪调试。字母T表示进程处于停止状态。
- ✓ 僵尸状态：进程由于某种原因已经终止或结束，但在进程表项中仍有纪录，该纪录可由父进程收集。字母Z表示进程处于僵尸状态。

下面的图 2-1 表明了进程的状态变化关系：

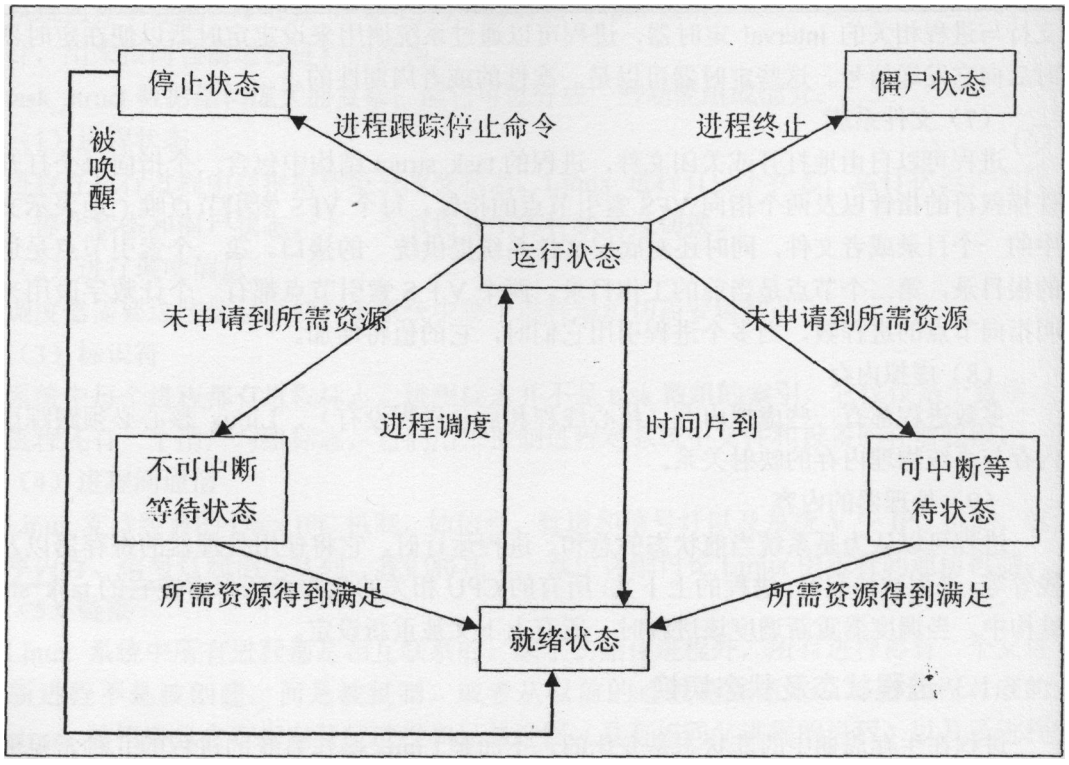


图 2-1 进程状态变化关系

在 Linux 系统中，进程的执行模式划分为用户模式和内核模式。如果当前运行的是用户程序、应用程序或者内核之外的系统程序，那么对应进程就在用户模式下运行；如果在用户程序执行过程中出现系统调用或者发生中断事件，就要运行操作系统核心程序，进程模式就变成内核模式。在内核模式下运行的进程可以执行机器的特权指令，而且此时该进程的运行不受用户的干预，即使是超级用户 root 也不能干预内核模式下进程的运行。

按照进程的功能和运行的程序分类，进程可划分为两大类：一类是系统进程，只运行在内核模式，执行操作系统代码，完成一些管理性的工作，例如内存分配和进程切换；另外一类是用户进程，通常在用户模式中执行，并通过系统调用或在出现中断、异常时进入内核模式。用户进程既可以在用户模式下运行，也可以在内核模式下运行，如图 2-2 所示。

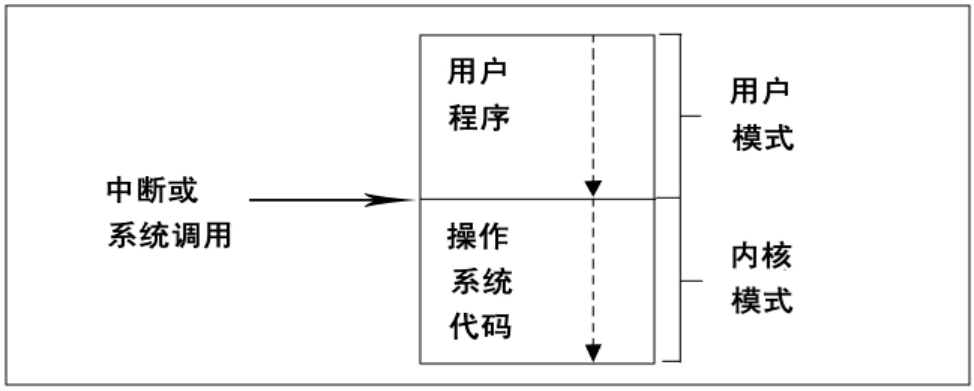


图 2-2 用户进程的两种运行模式

2.1.4 进程控制

现在我们开始了解一下进程的控制，主要介绍内核对 fork()、exec()、wait()、exit()几个和进程控制相关的系统调用的处理过程，为使用这些调用打下概念上的基础，并介绍系统启动的过程以及进程 init 的作用。

在 Linux 系统中，用户创建一个进程的唯一方法就是使用系统调用 fork()。内核为完成系统调用 fork()要进行几步操作:

第一步，为新进程在进程表中分配一个表项。系统对一个用户可以同时运行的进程数是有限制的，对超级用户没有该限制，但也不能超过进程表的最大表项的数目。

第二步，给子进程一个唯一的进程标识号(PID)。该进程标识号其实就是该表项在进程表中的索引号。

第三步，复制一个父进程的进程表项的副本给子进程。内核初始化子进程的进程表项时，是从父进程处拷贝的。所以子进程拥有与父进程一样的 uid、euid、gid、用于计算优先权的 nice 值、当前目录、当前根、用户文件描述符表等。

第四步，把与父进程相连的文件表和索引节点表的引用数加 1。这些文件自动地与该子进程相连。

第五步，内核为子进程创建用户级上下文。内核为子进程的 u 区及辅助页表分配内存，并复制父进程的区内容。这样生成的是进程的静态部分。

第六步，生成进程的动态部分，内核复制父进程的上下文的第一层，即寄存器上下文和内核栈，内核再为子进程虚设一个上下文层，这是为了子进程能“恢复”它的上下文。这时，该调用会对父进程返回子进程的 pid，对子进程返回 0。

Linux 系统的系统调用 exit()是进程用来终止执行的。进程发出该调用，内核就会释放该进程所占的资源，释放进程上下文所占的内存空间，但进程表项还被保留，内核将进程表项中纪录进程状态的字段设为僵尸状态。内核在进程收到不可捕捉的信号时，会从内核内部调用 exit()，使得进程退出。父进程通过 wait()得到其子进程的进程表项中记录的计时数据，并释放进程表项。最后，内核使得进程 1（init 进程)接收终止执行的进程的所有子进程。如果有子进程僵尸，就向 init 进程发出一个 SIGCHLD 的中断信号。

一个进程通过用 wait()来与它的子进程同步，如果发出 wait()调用的进程没有子进程则返回一个错误，如果找到一个僵尸的子进程就取子进程的 PID 及子进程退出时的退出参数。如果有子进程，但没有僵尸的子进程，发出 wait()调用的进程就将进入可中断的睡眠状态，直到收到一个子进程僵尸(SIGCLD)的信号或其他信号。

进程控制的另一个主要内容就是对其他程序引用。该功能是通过系统调用 exec()来实现的，exec()调用将一个可执行的程序文件读入并执行。内核读入程序文件的正文，清除原先进程的数据区，清除原先进程的中断信号处理函数的地址，当 exec()调用返回时，进程执行新的正文。

一个系统启动的过程，也称作是自举的过程，该过程因机器的不同而有所差异。但该过程的目的对所有机器都相同：将操作系统装入内存并开始执行。首先由硬件将引导块的内容读到内存并执行，引导程序将内核文件系统装入内存，并将控制转入内核的入口，内核开始运行。然后，内核首先初始化它的数据结构，并将根文件系统安装到根“/”，为进程 0 形成执行环境。设置好进程 0 的环境后，内核便作为进程 0 开始执行，并调用系统函数 fork()。因为这时进程 0 运行在内核状态，所以新的进程也运行在内核状态。新的进程(进程 1)创建自己的用户级上下文，设置并保存好用户寄存器上下文。这时，进程 1 就从内核状态返回用户状态调用 exec 执行/sbin/init 程序。进程 1 通常称为初始化进程，它负责初始化新的进程。

进程 init 除了产生新的进程外，还负责一些使用户在系统上登陆的进程。例如，进程 init 一般要产生一些 getty 的子进程来监视终端。如果一个终端被打开，getty 子进程就要求在这个终端上执行一个登陆的过程，当成功登陆后，执行一个 shell 程序，来使得用户与系统交互。同时，进程 init 执行系统调用 wait 来监视子进程的死亡，以及由于父进程的退出而产生的孤儿进程的移交。以上是系统启动和进程 init 的一个粗略的模型。

2.1.5 进程调度

所有进程部分时间运行于用户模式，部分时间运行于内核模式。如何支持这些模式，底层硬件的实现各不相同,但是存在一种安全机制可以使它们在用户模式和系统模式之间来回切换。用户模式的权限比系统模式下的小得多。进程通过系统调用切换到系统模式继续执行，此时核心为进程而执行。在 Linux 中，进程不能被抢占。只要能够运行它们就不能被停止。当进程必须等待某个系统事件时，它才决定释放出 CPU。例如进程可能需要从文件中读出字符。一般等待发生在系统调用过程中，此时进程处于内核模式；处于等待状态的进程将被挂起而其他的进程被调度管理器选出来执行。

进程常因为执行系统调用而需要等待。由于处于等待状态的进程还可能占用 CPU 时间，所以 Linux 采用了预加载调度策略。在此策略中，每个进程只允许运行很短的时间：200 毫秒，当这个时间用完之后，系统将选择另一个进程来运行，原来的进程必须等待一段时间以继续运行，这段时间称为时间片。

Linux 使用基于优先级的简单调度算法来选择下一个运行进程。当选定新进程后，系统必须将当前进程的状态，处理器中的寄存器以及上下文状态保存到 task_struct 结构中。同时它将重新设置新进程的状态并将系统控制权交给此进程。为了将 CPU 时间合理的分配给系统中每个可执行进程，调度管理器必须将这些时间信息也保存在 task_struct 中。每个进程的 task_struct 结构中包括下面这些调度信息：

(1)策略。

这时进程将会使用的调度策略。Linux 系统共有两种类型的进程：一般进程和实时进程。实时进程的权限要比其他进程的权限高。如果有一个实时进程等待运行，那么一般情况下都将首先运行。实时进程也有两种策略：轮流策略和先进先出策略。在轮流策略中，每个进程轮流运行，而在先进先出策略中，进程按照运行队列中的顺序执行，并且运行队列的顺序永远不变。

(2)优先权。

这是调度算法给予进程的优先权，也即当进程被允许运行时能够运行的时间的长短(jiffies)。你可以通过系统调用改变此优先权。

(3)实时优先权。

这是给予实时进程之间的一个相对的优先权。可以通过系统调用改变此实时进程的优先权。

(4)计数器。

这是进程允许运行的时间。当进程第一次运行时，它将被设置为进程的优先权，并且在每个时钟周期后减一。

调度算法可以在多种情况下发生。它可以在将当前进程放入等待队列时发生，也可以在一个系统调用后发生。当调度算法发生时，它将执行以下个步骤：

第一步：处理内核中的工作。

第二步：处理当前进程。

在其他进程运行之前，当前进程必须处理好：
如果当前进程使用的是轮流策略，则当前进程被放到运行队列的最后。
如果当前进程是可以被中断的，并且在最后一次调度之后接收到中断信号，则进程的状态被设置为可运行。
如果当前进程的运行时间用完，则进程的状态设置为可运行。
如果当前进程为可运行，则进程状态保持为可运行。
那些既不是可运行也不是可中断的进程将被从运行队列中移走。

第三步：选择进程。

调度算法查找运行队列以找出最需要运行的进程。如果队列中有实时进程，那么实时进程将优先运行。一个普通进程的优先权是其计数器的值，而实时进程的优先权是其计数器的值加上 1000。当前进程由于已经消耗了一些时间片，所以和其他的具有相同优先权的进程相比将处于不利的位置。如果有几个进程具有相同的优先权，则最靠近队列前端的进程将被执行。

第四步：进程交换。

如果最需要执行的进程不是当前进程，那么当前进程就会被挂起，同时一个新的进程将被执行。在结束当前进程时，进程所涉及的一切机器状态，包括程序计数器以及 CPU 寄存器将保存到进程的 task_struct 中，而即将运行的进程的 task_struct 中的状态将装入到机器中。如果当前进程和即将运行的进程使用了虚拟内存的话，系统的内存页面表页会被更新。

2.1.6 什么是线程

进程是资源管理的最小单位，线程是程序的最小单位。在操作系统设计上，从进程演化出线程，最主要的目的就是更好地支持多处理器，并且减小进程上下文切换的开销。在两个普通进程间进行切换时，内核准备从一个进程的上下文切换到另--个进程的上下文要花费很大的开销。线程允许进程在几个正在运行的任务之间进行切换，而不必执行前面提到的上下文切换。

线程是进程的一条执行路径，它包含独立的堆栈和 CPU 寄存器状态，每个线程共享其所附属的进程的所有的资源，包括打开的文件、内存页面、信号标识及动态分配的内存等等。线程和进程的关系是：线程是属于进程的，线程运行在进程空间内，同一进程所产生的线程共享同一物理内存空间，当进程退出时该进程所产生的线程都会被强制退出并清除。一个进程至少需要一个线程作为它的指令执行体，进程管理着资源(比如 cpu、内存、文件等等)，而将线程分配到某个 cpu 上执行。一个进程当然可以拥有多个线程，此时，如果进程运行在多处理器的机器上，它就可以同时使用多个 CPU 来执行各个线程，达到最大程度的并行，以提高效率。同时，即使是在单 cpu 的机器上，采用多线程模型来设计程序，正如采用多进程模型代替单进程模型一样，使设计更简洁、功能更完备，程序的执行效率也更高，例如采用多个线程响应多个输入，而此时多线程模型所实现的功能实际上也可以用多进程模型来实现，而与后者相比，线程的上下文切换开销就比进程要小多了，从语义上来说，同时响应多个输入这样的功能，实际上就是共亨了除 cpu 以外的所有资源的。

线程间的通信机制区别于进程间通信机制。对不同进程，它们具有独立的数据空间，要进行数据的传递只能通过进程间通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其它线程所用，这不仅快捷，而且方便。当然，数据的共享也带来其他一些问题，有的变量不能同时被两个线程所修改，有的子程序中声明为 static 的数据更有可能给多线程程序带来灾难性的打击，这些正是编写多线程程序时最需要注意的地方。

多线程程序作为一种多任务、并发的工作方式，还有以下的优点：

提高应用程序响应。这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作置于一个新的线程，可以避免这种尴尬的情况。

使多 CPU 系统更加有效。操作系统会保证当线程数不大于 CPU 数目时，不同的线程运行于不同的 CPU 上。

- ✓ 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。Linux 有两种线程模型：

一种是 POSIX 标准的线程，即 Pthread。

另一种是 IBM 开发的 NGPT(Next Generation Posix Threads for Linux)，它是基于 GNU Pth(GNU Portable Threads)。

我们主要介绍的线程编程主要是 Pthread，因为 Linux 对它的支持最好。相对进程而言，线程是一个更加接近于执行体的概念，它可以与进程中的其它线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。在串行程序基础上引入线程和进程是为了提高程序的并发度，从而提高程序运行效率和响应时间。

那么在什么场合会使用 Pthread 呢？

在返回前阻塞的 I/O 任务能够使用一个线程处理，同时继续执行其它处理任务。

在有一个或多个任务受不确定性事件影响，比如网络通信的可获得性影响的场合，能够使用线程处理这些异步事件，同时继续执行正常的处理。

如果某些程序功能比其它的功能更重要，可以使用线程以保证所有功能都出现，但那些时间密集型的功能具有更高的优先级。

以上三点可以归纳为在检查程序中潜在的并行性时，也就是说找出能够同时执行任务时使用 Pthread。

2.2 进程编程

2.2.1 获得与进程有关的 ID

在学习如何创建线程之前，需要先了解和线群 ID 相关的一些函数。在介绍线群的结构的是，我们曾经提到 task_struct 包含了一些和进程相关的标识。Linux 进程的标识非常多，但我们经常需要关心的标识包括：

- ✓ 真正用户标识号(UID)：用于标识正在运行进程的用户。
- ✓ 真正用户组标识号(GID)：用于标识运行进程的用户所属的组ID。

- ✓ 进程标识号(PID)：用于标识进程。
- ✓ 进程组标识号(process group ID)：用于进程所属的进程组ID。一个进程可以属于某个进程组。可以发送信号给一组进程。注意，它不同于GID。

下面是获取这些标识号的系统调用函数原型：

```
#include<sys/types.h>
#include<unistd.h>
uid_t getuid(void);
gid_t getgid(void);
pid_t getgid(void);
pid_t getppid(void);
pid_t getpgrp(void);
pid_t getpgid(pid_t pid);
```

getuid()系统调用，获得进程的用户标识号。

getgid()系统调用，捩得进程的用户所属的用户组 ID。

getpid()系统调用，要获得当前进程的 ID。

getppid()系统调用，狭得当前进程的父进程的 ID。

getpgrp()系统调用，获得当前进程所在的进程组的 ID。

getpgid(pid_t)系统调用，获得进程 ID 为 pid 的进程所在的进程组 ID。

注意一下 GID 和进程组 ID 的区别，一般执行该进程的用户的组 ID 就是该进程的 GID。对于进程组 ID，一般来说，一个进程在 shell 下执行，shell 程序就将该进程的 PID 赋给该进程的进程组 ID，从该进程派生的子进程都拥有父进程所属的进程组 ID，除非父进程将子进程的所属组 ID 设置成与该子进程的 PID 一样。

下面我们来编写一个简单的程序来查看进程的标识信息。

示例2-1：获得进程有关的ID

源代码：pid_test.c

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

int main( int argc, char **argv )
{
    printf("Current process's UID = [%d]\n", getuid() );
    printf("Current process's GID = [%d]\n", getgid() );
    printf("Current process's PID = [%d]\n", getpid() );
    printf("Current process's PPID = [%d]\n", getppid() );
    printf("Current process group ID = [%d] = [%d]\n", getpgrp(),
getpgid(getpid()) );
```

```
    return 0;
}
```

程序编译运行的结果如下：

```
  [alex@alex～/]$  gcc -o pid_test pid_test.c
  [alex@alex～/]$  ./pid_test
Current process's UID = [502]
Current process's GID = [502]
Current process's PID = [4022]
Current process's PPID = [3873]
Current process group ID = [4022] = [4022]
  [alex@alex～/]$ps
  PID TTY          TIME CMD
 3873 pts/0    00:00:00 bash
 4023 pts/0    00:00:00 ps
```

由于我们是在 shell 执行了 pid_test 程序，从 ps 命令，可以看到当前 bash shell 的 PID 是 3873。程序 pid_test 运行时的 PID 为 4022，它的用户及用户组 ID 都是 502,它的父进程 ID 等于 bash shell 的 PID 3873，它的进程组用户被设置为等于 PID，为 4022。

2.2.2 派生进程

现看一个简单的派生进程的例子：

示例 2-2 ：派生进程

源代码：fork_test.c

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

int main( int argc, char **argv )
{
     pid_t pid;

     printf("Current process's PID = [%d]\n", getpid() );

     pid = fork();
```

```
if( pid == 0 ){
    printf("This is child process, PID = [%d], my parrent PID = [%d]\n",
getpid(), getppid() );
    }else if( pid != -1 ){
        printf("This is parent process, PID = [%d]\n", getpid() );
    }else{
        printf("There was an error with forking!\n");
    }
    return 0;
}
```

程序非常简单，调用了 fork()函数并把返回值赋值给 pid 变量。分别在 pid 等于 0、pid 不等于-1 的时候输出当前进程的 PID。

编译运行这个程序，结果如下：

```
[alex@alex~/]$./fork_test
Current process's PID = [4144]
This is child process, PID = [4145], my parrent PID = [4144]
This is parent process, PID = [4144]
```

可见，当 pid 等于 0 的时候，表明是当前进程派生出来的子进程。如果返回的 pid 不等于-1，表明派生操作成功并且返回值 pid 就是新的子进程的 PID。如果返回值等于-1，那么操作失败。

fork()系统调用的函数原型是：

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


pid_t fork(void);
```

在语句 pid=fork()之前，只有一个进程在执行这段代码，但在这条语句之后，就变成两个进程在执行了，这两个进程的代码部分完全相同，将要执行的下一条语句都是 if(pid==0)。

父子进程的区别除了进程 PID 不同外，变量 pid 的值也不相同，pid 存放的是 fork 的返回值。for 调用的一个奇妙之处就是它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：

- ✓ 在父进程中，fork返回新创建子进程的进程PID。
- ✓ 在子进程中，fork返回0。
- ✓ 如果出现错误，fork返回-1。fork出错可能有两种原因：
- ✓ 当前的进程数已达到了系统规定的上限，这时errno的值被设置为EAGAIN。
- ✓ 系统内存不足，这时errno的值被设置为ENOMEM。
- ✓ 下面在修改一下示例2-1的代码，来看看两个进程是如何被调度执行的。

示例2-3：研究进程是如何被调度

源代码：fork_test2.c

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

int  main( int argc, char **argv )
{
    pid_t pid;
    int i;
    printf("Current process's PID= [%d]\n",getpid());
    pid=fork();
    if(pid==0){
        for(i=0;i<10;i++){
        printf("This  is  child  process,PID=[%d],my  parrent  PID
=[%d]\n",getpid(),getppid());
                sleep(1);
            }
}else if(pid != -1){
        for(i=0;i<10;i++){
        printf("This   is   parent   process,PID=[%d],child   PID
=[%d]\n",getppid(),pid);
                sleep(1);
        }
}else{
    printf("There was an error with forking!\n");
```

```
}
        return 0;
}
```

我们在父进程和子进程函数中都加入了一个 for 循环，循环输出 10 次，并在每次输出之后睡眠 1 秒。下面是程序的执行结果：

```
 [alex@alex～/] $./fork_test
Current process's PID= [4259]
This is child process,PID=[4260],my parrent PID =[4259]
This is parent process,PID=[4052],child PID =[4260]
This is child process,PID=[4260],my parrent PID =[4259]
This is parent process,PID=[4052],child PID =[4260]
This is child process,PID=[4260],my parrent PID =[4259]
This is parent process,PID=[4052],child PID =[4260]
……
……
This is child process,PID=[4260],my parrent PID =[4259]
This is parent process,PID=[4052],child PID =[4260]
```

从输出的结果，可以看到，子进程的输出和父进程的输出是交递输出的，而不是由某个一个进程执行完 10 次循环之后，才能执行另外一个进程。前面我们介绍过，当进程进入睡眠模式的时候，内核会调度新的进程进入运行模式，这样使得两个进程看上去像是同步执行。

2.2.3 执行其他程序

我们派生一个子进程来完成某项工作的时候，经常需要让另外一个程序来完成。函数 exec()可以用来执行一个可执行文件来代替当前进程的执行映像。需要注意的是，该调用并没有生成新的进程，而是在原有进程的基础上，替换原有进程的正文，调用前后是同一个进程，进程号 PID 不变，但执行的程序变了(执行的指令序列改变了)。在 Linux 它有六种调用的形式，他们的声函数原型如下：

```
#include<unistd.h>



//extern char **environ;

int execl(const char *path,const char *arg,…);
int execlp(const char *file,const char *arg,…);
int execle(const char *path,const char *arg,…,char *const envp[]);
```

```
    int execv(const char *path,char *const argv[]);
    int execvp(const char *file,char *const argv[]);
    int execve(const char *filename,char *const argv[],char *const
envp[]);
```

六个函数中，其中 execve()是系统调用，其他函数都是 C 库函数。下面我们先详细讲述其中的一个，然后再给出它们之间的区别。

先看一个简单的例子：

示例2-4：用execl()执行一个程序

源代码execl_test.c

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

//extern char **environ;

int main( int argc, char **argv )
{
    printf("This is an execl test.\n");
    execl ( "/bin/ls","ls","/",NULL);
    printf("This code is running after the execl() call.\n");
    printf("You should never see this message unless execl() failed.\n");
}
```

这个程序非常简单，首先打印一行信息，然后调用 execl()函数。

在函数 execl()中，参数 file 是将要执行的文件，剩余的参数将作为 argv 传递给函数。

注意：

传给main函数的argv参数，argv[0]通常是程序名，而第二个参数开始才是真正的程序的参数，所以ls会出现两次。

最后一个参数必须是空指针，表明参数列表到次为止，因此 NULL 参数必不可少。

由于 execl()执行了新的程序，代替了当前进程的映像，因此，除非 execl()调用失败，否则后面的代码不会被执行。比如下面的运行结果：

```
    [alex@alex~]$./execl_test
    This is an execl test.
    bin   dev  home  lost+found  misc  net  proc  sbin     srv
tftpboot  usr
```

```
   boot   etc   lib   media       mnt   opt   root   selinux   sys   tmp
var
```

这个结果看得出来，原来的程序的映像已经被程序 ls 的映像代替，后面的信息没有被显示出来。

exec()的六个函数实现的功能是一样的,只是在传递参数和设置环境变量方面提供了不同的方式。六个函数的都是以“exec”四个字母开头的，后面的字母表示了其用法上的区别：

- ✓ 带有字母“l"的函数，表明后面的参数列表是要传递给程序的参数列表，参数列表的第一个参数必须是要执行程序，最后一个参数必须是NULL。
- ✓ 带有字母“p”的函数，第一个参数可以是相对路径或程序名，如果无法立即找到要执行的程序，那么就在环境变量PATH指定的路径中搜索。其他函数的第一个参数则必须是绝对路径名。
- ✓ 带有字母“v”的函数，表明程序的参数列表通过一个字符串数组来传递。这个数组和最后传递给程序的main函数的字符串数组argv完全一样。第一个参数必须是程序名，最后一个参数也必须是NULL。
- ✓ 带有字母“e”的函数，用户可以自己设置程序接收一个设置环境变量的数组。

按照这些规则组合，就是 exec()的六个函数。那么我们也可以来用其他五个函数来实现示例 2-4。下面是用 execvp()函数的例子。

示例2-5：用execvp()执行一个函数

源代码：exevp_test.c

```
#include <stdio.h>
#include <unistd.h>
//extern char **environ;
int    main(int argc,char**argv)
{
    char*arguments[3];
    arguments[0]="ls";
    arguments[1]="/";
    arguments[2]= NULL;
    printf("This is an execl test.\n");
    execvp("ls",arguments);
    printf("This code is running after the execl() call .\n");
    printf("You should never see this message unless execl() faild.\n");
}
```

和示例 2-4 的区别有两点：

- ✓ 第一个参数由“/bin/ls”变为“ls"。
- ✓ 第二个参数是一个字符串数组。

下面是程序的运行结果：

```
    [alex@alex~]$./execvp_test
   This is an execl test.
   bin   dev  home  lost+found  misc  net  proc  sbin     srv
tftpboot  usr
   boot  etc  lib   media       mnt   opt  root  selinux  sys  tmp
var
```

运行结果和示例 2-4 是完全一样的。第一个参数并没有指定 ls 命令的路径，但是 exevp()根据环境变量 PATH 自动找到了 ls 命令并执行。

那么，读者也可以用其他的函数来修改这个例子作为练习。

exec()函数通常和 fork()一起配合使用，由 fork 派生一个子进程，在子进程执行新的程序。下面的例子在子进程里立即执行了 execlp()函数，如果执行失败，则输出错误信息。

示例2-6：fork()和exec()配合使用

源代码：fork_test3.c

```
#include<stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>

//extern char **environ;

int main( int argc, char **argv )
{
    int ret;
    pid_t pid;
    pid_t pid_c;

    pid = fork();

    if( pid == 0 ){
        printf("Child PID [%d] will exit with status 0.\n", getpid() );
        exit(0);
    }else if( pid != -1 ){
        pid_c = wait( &ret );
```

```
        printf("Child process PID [%d] return [%d].\n", pid_c, ret);
    }else{
        printf("There was an error with forking!\n");
    }
    return 0;
}
```

在这个示例中，进程调用了 wait()函数，等待子进程退出，这也是多进程编程的一个重要函数--等待进程。

2.2.4 终止进程及进程返回值

exit()函数的功能是终止发出调用的进程。它的函数原型如下：

```
#include <stdlib.h>
void exit(int status);
```

exit()函数会调用系统调用_exit，立即终止发出调用的进程。所有属于该进程的文件描述符都关闭。该进程的所有子进程由进程 1(进程 init)接收，并对该进程的父进程发出一个 SIGCHLD(子进程僵死)的信号。参数 status 作为退出的状态值返回父进程，该值可以通过系统调用 wait()来收集。如果进程是一个控制终端进程，则 SIGHUP 信号将被送往该控制终端的前台进程。

2.2.5 等待进程

前面我们介绍过，子进程终止的时候，必须由父进程回收其进程表，否则进程将处于僵尸状态知道被回收。如果父进程在子进程终止前已经终止，那么该进程的所有子进程由 init 进程回收。

下面我们来看一个例子，如果子进程终止而没有被回收的状态。

示例2-7：子进程没有被回收

源代码：zombi_test.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#include<stdlib.h>

int main(int argc, char **argv)
```

```
{
        int ret;
        pid_t pid;
        pid_t pid_c;

        pid = fork();
        if(pid==0){
           printf("Child PID [%d] will exit now.\n",getpid());
           exit(0);
        }
        else if(pid != -1)
        {
           sleep(60);
        }
       else{
           printf("There was an error with forking!\n");
        }

}
```

我们在一个终端执行这个程序，屏幕输出：

```
 [alex@alex～/] $./zombie_test
Child PID [4635] will exit now.
```

然后到另外一个终端查看了进程 4635 状态：

```
    [alex@alex～/]$ps 4635
      PID TTY      STAT   TIME COMMAND
       4635 pts/0    Z+     0:00 [g] <defunct>
```

可见这时 zombie_test 处于“Z”的僵尸状态。当 30 秒后，父进程退出，我们再查看子进程时，发现子进程已经被回收了。原来当父进程退出的时候，子进程将被移交给 init，init 进程回收了这个僵尸进程。

系统调用 wait()的功能是发出调用的进程只要有子进程，就睡眠直到它们中的一个终止。它的函数原型是：

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

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

wait()函数将使进程进入睡眠直到它的一个子进程退出时或收到一个不能被忽略的信号被唤醒。如果调用发出时，已经有退出的子进程(这时子进程的状态是僵尸状态)，该调用立即返回。其中调用返回时参数 status 中包含子进程退出时的状态信息，返回值是退出的子进程的 PID。如果参数 status 是 NULL，那么返回值将被忽略。

下面的例子显示了 wait()的使用方法。

示例2-8：wait()函数的使用方法

源代码：wait_test.c

```
#include<stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>

//extern char **environ;

int main( int argc, char **argv )
{
    int ret;
    pid_t pid;
    pid_t pid_c;

    pid = fork();

    if( pid == 0 ){
        printf("Child PID [%d] will exit with status 0.\n", getpid() );
        exit(0);
    }else if( pid != -1 ){
        pid_c = wait( &ret );
        printf("Child process PID [%d] return [%d].\n", pid_c, ret);
    }else{
        printf("There was an error with forking!\n");
    }
    return 0;
}
```

执行这个函数，结果如下：

```
[alex@alex~/]$./wait_test
Child PID [4693] will exit with status 0.
Child process PID [4693] return [0].
```

wait()函数回收了子进程 4693 并获得返回值 0。

waitpid()函数与调用 wait()的区别是 waitpid()等待由参数 pid 指定的子进程退出。其中参数 pid 的含义与取值方法如下：

- ✓ 参数pid<-1时，等待进程组ID等于pid的绝对值的子进程退出。
- ✓ 参数pid=0时，等待进程组ID等于当前进程的进程组ID的子进程退出。
- ✓ 参数pid>0时，等待进程ID等于参数pid的子进程退出。
- ✓ 参数pid=-1时，等待任何子进程退出，相当于调用wait()。

参数 options 的让 wait()的使用更加灵活，下面宏定义是 option 的可选值及含义，option 可以多个宏使用位或运算获得：

- ✓ WNOHANG：该选项要求如果没有子进程退出就立即返回。
- ✓ WUNTRACED：对已经停止但未报告状态的子进程，该调用也从等待中返回和报告状态。如果status不是空，调用将使status指向该信息。

waitpid()将返回退出的子进程的 PID。如果设置了 WNOHANG 选项没有子进程退出就返回 0。如果发生错误时返回-1，发生错误时，可能设置的错误代码如下。

- ✓ ECHILD：该调用指定的子进程pid不存在，或者不是发出调用进程的子进程。
- ✓ ElNVAL：参数options无效。
- ✓ ERESTARTSYS：WNOHANG没有设置并且捕获到SIGCHLD或其它未屏蔽信号。

表 2-1 中的宏定义检查子进程的返回状态 status。前面三个用来判断退出的原因，后面三个是对应不同的原因返回状态值。

表 2-1 检查子程序返回状态的宏定义

宏定义	含义
WIFEXITED(status)	如果进程通过系统调用_exit 或函数调用 exit 正常退出，该宏的值为真。
WIFSIGNALED(status)	如果子进程由于得到的信号(signal)没有被捕捉而导致退出时，该宏的值为真。
WIFSTOPPED(status)	如果子进程没有终止，但停止了并可以重新执行时，该宏返回真。这种情况仅出现在 waitpid 调用中使用了 WUNTRACED 选项。
WEXITSTATUS(status)	如果 WIFEXITED(status)返回真，该宏返回由子进程调用_exit(status)或 exit(status)时设置的调用参数 status 值。
WTERMSIG(status)	如果 WIFSIGNALED(status)返回为真，该宏返回导致了子进程退出的信号(signal)的值。

WSTOPSIG(status)	如果 WIFSTOPPED(status)返叫真，该宏返回导致子进程停止的信号(signal)值。

下面的例子用了 waitpid()来等待子进程，并用宏定义来判断返回值。

示例2-9：waitpid()的使用方法。

源代码：waitpid_test.c

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<errno.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
    int status;
    pid_t pid;
    pid_t pid_c;
    pid=fork();
    if(pid ==0){
             printf("child process will sleep 5 seconds.\n");
             sleep(5);
             printf("child PID[%d]will exit with status
3.\n",getpid());
             exit(3);
    }else if(pid!=-1){
              do{
                pid_c = waitpid(pid,&status,WNOHANG);
        if(pid_c==0){
                printf("Not child process exit\n");
                       sleep(1);
                     }
                }while(pid_c==0);
           if(WIFEXITED(status)){
                /*如果WIFEXITED返回非零值*/
```

```
                printf("the  child process  %d  exit
normally.\n",pid_c);
                printf("the  return  code  is %
d.\n",WEXITSTATUS(status));
                  }else{
                            /*如果WIFEXITED返回零*/
                     printf("the  child process  %d  exit
abnormally.\n",pid_c);
    }
        }else{
                printf("There was an error with forking!\n");
   }
   return 0;
   }
```

在这个例了中，我们让子进程先睡眠 5 秒然后退出。父进程等待函数 waitpid()的 option 使用了 WNOHANG 选项，是的父进程不会被阻塞在 waitpid()函数。用一个 while 循环来等待子进程的退出。通常父进程还要执行其他任务的时候，我们采用类似的方法让父进程得以继续工作但同时又可以及时回收子进程。

下面是程序的执行结果：

```
[alex@alex～/]$./waitpid_test
child process will sleep 5 seconds.
Not child process exit
Not child process exit
Not child process exit
Not child process exit
Not child process exit
child PID[%d]will exit with status 3.
the  child process  %d  exit normally.
the  return  code  is %d.
```

Linux2.6.9 之后的内核版本，还通过了更加精确的进程等待方法 waitid()。在此不做详细介绍了，有兴趣者可以查看 waitid 的 Man page。

2.2.6 进程的同步措施

有时候，两个或者多个进程需要能够保持同步。比如它们可能会同时向一个文件执行写操作，但某个时间段只能允许一个进程对文件写入，以避免文件遭到不可预料的破坏。或者父进程需要等待子进程完成一件任务之后才能继续执行，wait 函数或许能解决一部分这样的问题。但是有更多的方法可以实现进程的同步。

我们可以使用文件锁定、信号、信号量、管道或者套接字等方法来进行进程同步，其中文件锁定和信号量是专门为了实现进程同步而设计的，而其他都是通用的进程间通信的工具，也可以用来完成特定的进程同步处理。

比如进程派生出一个子进程来完成特定任务，然后父进程和子进程各自独立运行。当子进程完成它的任务时要退出，它会自动发送一个可以捕捉的 SIGCHLD 信号给父进程。

下一章，我们将详细介绍进程同步措施文件锁定和信号量的使用方法。

2.3 线程编程

本节我们主要介绍 Linux 中的 pthread 线程模型的编程。pthread 线程通过 libpthread 线程函数库来实现。所以在编译多线程程序的时候，需要链接 libpthread，比如

```
gcc  -o pthread_test -lpthread
```

2.3.1 线程的创建和使用

创建线程用 pthread_create()函数，其函数原型是：

```
#include<pthread.h>

int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restric attr,
                   void* (*start_routine)(void *),void *restrict arg);
```

第一个参数为指向线程标识符的指针，第二个参数用来设置线程属性，大多数情况下，我们不需要设置线程的属性，那么空指针 NULL 就可以了，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数。

当创建线程成功时，函数返回 0，若不为 0 则说明创建线程失败，常见的错误返回代码为：

- ✓ EAGAIN：前者表示系统限制创建新的线程，例如线程数目过多了。
- ✓ EINVAL：后者表示第二个参数代表的线程属性值非法。

创建线程成功后，新创建的线程则运行参数三和参数四确定的函数。

我们先看一个例子，创建一个线程并在线程输出一些信息。

示例2-10：创建线程

源代码：pthread_test.c

```
#include<stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/types.h>
#include <linux/unistd.h>
#include <errno.h>

void thread(void)
{
    int i;
    for( i = 0; i < 3; i++ ){
        printf( "This is a pthread.\n" );
        sleep(1);
    }
}

int main(void)
{
    pthread_t id;
    int i,ret;
    ret = pthread_create( &id, NULL, (void *)thread, NULL );
    if( ret != 0 ){
        printf("Create pthread error!\n");
        exit(1);
    }
    for( i = 0; i < 3; i++ ){
        printf("This is the main process.\n");
        sleep(1);
    }
    pthread_join( id, NULL );
    return (0);
}
```

在这个例了中，第二个参数我们设为空指针，这样将生成默认属性的线程。函数 thread 不需要参数，所以最后一个参数也设为空指针。

在 thread 线程中，用一个 for 循环，循环 3 次输出“This is a pthread.”，每次输出之后睡眠 1 秒。主线程也用 for 循环，循环 3 次输出“This is the main process.”，每次输出后睡眠 1 秒。这个程序有点类似于示例 2-3，我们来看程序运行的结果：

```
[alex@alex～/] $./pthread_test
This is the main process.
This is a pthread.
This is the main process.
This is a pthread.
This is the main process.
This is a pthread.
```

两个线程的输出是交替的，可见主线程和子线程在进入睡眠的时候，另外一个线程就会被调度执行。

在示例 2-10 中还出现了一个函数 pthread_join()，这个函数是主线程用于等待子线程结束的，其函数原型是：

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

第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。

一个线程的结束有两种途径，一种是和示例 2-10 一样，函数结束了，调用它的线程也就结束了；另一种方式是通过函数 pthread_exit()来实现。它的函数原型为：

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

函数的参数是函数的返回代码，只要 pthread_join 中的第二个参数 value_ptr 不是 NULL，这个值将被传递给主线程。

注意：

一个线程不能被多个线程等待，否则第一个接收到信号的线程成功返回，其余调用pthread_join()的线程则返回错误代码ESRCH。

我们修改 2-10，获取线程 thread 的返回值并打印到屏幕。

示例 2-11：获取子线程返回值

源代码：pthread_test2.c

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

```
#include <pthread.h>
#include <sys/types.h>
#include <linux/unistd.h>
#include <errno.h>

void thread(void)
{
    int i;
    for( i = 0; i < 3; i++ ){
        printf( "This is a pthread.\n" );
        sleep(1);
    }
    pthread_exit( (void *)i );
}

int main(void)
{
    pthread_t id;
    int i,ret;
    void *result;
    ret = pthread_create( &id, NULL, (void *)thread, NULL );
    if( ret != 0 ){
        printf("Create pthread error!\n");
        exit(1);
    }
    for( i = 0; i < 3; i++ ){
        printf("This is the main process.\n");
        sleep(1);
    }
    pthread_join( id, &result );
    printf("Child thread return [%d]\n", (int)result);
    return (0);
}
```

程序的执行结果如下：

```
[alex@alex～/] $./pthread_test2t
This is the main process.
This is a pthread.
```

```
This is the main process.
This is a pthread.
This is the main process.
This is a pthread.
Child thread return [3]
```

2.3.2 线程同步-互斥锁

当我们需要控制对共享资源的存取的时候，可以用一种简单的加锁的方法来控制。我们可以创建一个读/写程序，它们共用一个共享缓冲区，使用互斥锁来控制对缓冲区的存取。

我们先看一个例子。这是一个读/写程序，它们公用一个缓冲区，缓冲区只有两个状态：有数据或没有数据，用变量 buffer_has_data 来记录 buffer 的状态。主线程先启动一个子线程来执行 read_ buffer()函数，然后等待用户输入，如果有用户输入，则调用 write_buffer()函数把数据写入 buffer。read_buffer()函数中，每秒 1 个循环等待 buffer 数据。由于用户输入数据的时间是随机的，为了避免读写操作的冲突，在读写数据前对线程进行锁定，读写数据结束后解除锁定。

示例2-12：互斥锁的使用方法

源代码：mutex_test.c

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

char buffer[128];
int buffer_has_data=0;
pthread_mutex_t mutex;

void   write_buffer (char *data)
{
    /* 锁定互斥锁*/
    pthread_mutex_lock (&mutex);
    if (buffer_has_data == 0){
        sprintf( buffer, "%s", data);
        buffer_has_data=1;
    }
    /* 打开互斥锁*/
    pthread_mutex_unlock(&mutex);
```

```
}

void   read_buffer(void)
{
    while(1){
        /* 锁定互斥锁*/
        pthread_mutex_lock(&mutex);
        if(buffer_has_data == 1){
            printf("Read buffer, data = [%s]\n", buffer);
            buffer_has_data=0;
        }
        /* 打开互斥锁*/
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
}

int main ( int argc, char **argv )
{
    char input[128];
    pthread_t reader;
    /* 用默认属性初始化一个互斥锁对象*/
    pthread_mutex_init( &mutex, NULL );
    pthread_create(&reader, NULL, (void *)(read_buffer), NULL);

    while( 1 ){
        scanf( "%s", input );
        write_buffer( input );
    }
    return 0;
}
```

首先声明了互斥锁变量 mutex，pthread_mutex_t 为不公开的数据类型，其中包含一个系统分配的属性对象。

函数 pthread_mutex_init()用来生成一个互斥锁。其函数原型如下：

```
#include<pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
```

第一个参数是互斥变量的地址，第二个参数设置互斥变量的属性，大多数情况下.选择默认属性，则传入空指针 NULL。

Pthread_mutex_lock()函数声明开始用互斥锁上锁，此后的代码直至调用 pthread _mutex_ unlock()为止，均被上锁，即同一时间只能被一个线程调用执行。当一个线程执行到 pthread_mutex_lock()处时，如果该锁此时被另一个线程使用，那么此线程被阻塞，线程一直阻塞知道另一个线程释放此互斥锁。这两个函数原型是：

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

两个函数的参数都是互斥变量的地址。函数执行成功返回 0，否则返回错误号。

互斥锁使用很方便，但是有一个缺点是使用互斥锁的过程中很有可能会出现死锁：两个线程试图同时占有两个资源，并按不同的次序锁定相应的互斥锁，例如两个线程都需要锁定互斥锁 1 和互斥锁 2，A 线程锁定了互斥锁 1，B 线程了锁定互斥锁 2，此时如果 A 线程在解除互斥锁 1 之前又去锁定互斥锁 2，而 B 恰好又需要去锁定互斥锁 1，这时就出现了死锁。看起来像是绕口令，通俗一点的来解释：线程 A 和线程 B 都需要独占使用 2 个资源，但是他们都分别先占据了一个资源,然后有相互等待另外一个资源的释放,这样的形成了一个死锁。

此时我们可以使用函数 pthread_mutex_trylock()，它是函数 pthread_mutex_lock()的非阻塞函数，当它发现死锁不可避免时，它会通过 errno 返回 EBUSY，我们可以针对死锁做出相应的处理。Pthread_mutex_try_lock()的函数原型是：

```
int  pthread_mutex_trylock(pthread_mutex_t *mutex);
```

当互斥锁不再使用的时候，应当释放互斥变量，函数 pthread _mutex.destory()用来释放一个互斥变量。

测试题

2.1 某个进程的 pid 是_____；它的父进程 ID 是_____；Linux 系统中进程 id=1 是_____。

2.2 创建进程的两种方法____和_____。

2.3 说明 execXXX()函数中每个 XXX 字母的代表的意义。

2.4 Linux 进程创建的两个步骤是_____；和_____。

2.5 信号的处理方式有_____,_____,_____,_____。

2.6 存在 sig_atomic_t 类型的意义何在？

2.7 查看帮助，给出 int waitpid()函数参数的意义。

2.8 编译支持线程需要使用的库是_____。

2.9 线程中，类似于进程的等待函数是_____。作用是什么？

2.10 支持阻塞的线程互斥体的函数是 _____，不阻塞的函数是_____。

2.11 如何设定不可取消的线程临界区？

2.12 条件变量和线程信号量使用的时候区别有哪些？

2.13 进程的地址空间分为_____，_____，_____，_____，_____。

2.14 堆和栈的区别有哪些？

本章总结

本章主要介绍进程的基本概念，介绍了进程的状态、控制以及调度的概念，介绍了由进程演化出的线程的基本概念，然后介绍如何进行进程和线程的派生、控制等基本编程方法。

第三章 进程间通信

本章介绍进程间进行通信和同步的方法。首先介绍最古老的信号机制，用于进程同步的文件锁，用于进程间传递数据的管道，还介绍了 Linux 系统支持的进程间通信机制(Interprocess Communication，简称 IPC)，包括信号量、消息队列、共享内存。

引言：

学完本章内容以后，你将能够

了解进程间通信的多种方法

熟练使用常见的 IPC 方式编程

了解进程间的同步机制

3.1 信号

3.1.1 什么是信号机制

信号(signal)机制是 Linux 系统中最为古老的进程之间的通信机制。Linux 信号也可以称为软中断，是在软件层次上对中断机制的一种模拟。在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信号的到达，信号是进程间通信机制中唯一的异步通信机制，可以看作是异步通知，通知接收信号的进程发生了什么。

信号事件的发生有两个来源：

- ✓ 硬件来源，比如我们按下了键盘或者其它硬件故障；
- ✓ 软件来源，最常用发送信号的系统函数是 kill()，rise()，alarm()和 setitimer()等函数，软件来源还包括一些非法运算等操作。

Linux 系统中定义了一系列的信号，可以使用“kill -l”命令列出所有的信号，比如：

```
[alex@alex~]$kill  -l
```

1)SIGHUP	2)SIGINT	3)SIGQUIT	4)SIGILL
5)SIGTRAP	6)SIGABRT	7)SIGBUS	8)SIGFPE
9)SIGKILL	10)SIGUSR1	11)SIGSEGV	12)SIGUSR2
13)SIGPIPE	14)SIGALRM	15)SIGTERM	16)SIGSTKFLT
17)SIGCHLD	18)SIGCONT	19)SIGSTOP	20)SIGTSTP
21)SIGTTIN	22)SIGTTOU	23)SIGURG	24)SIGXCPU
25)SIGXFSZ	26)SIGVTALRM	27)SIGPROF	28)SIGWINCH
29)SIGlO	30)SIGPWR	31)SIGSYS	34)SIGRTMIN
35)SIGRTMIN+1	36)SIGRTMIN+2	37)SIGRTMIN+3	38)SIGRTMIN+4
39)SIGRTMIN+5	40)SIGRTMIN+6	41)SIGRTMIN+7	42)SIGRTMIN+8
43)SIGRTMIN+9	44)SIGRTMIN+10	45)SIGRTMIN+11	46)SIGRTMIN+12
47)SIGRTMIN+13	48)SIGRTMIN+14	49)SIGRTMIN+15	50)SIGRTMAX-14
51)SIGRTMAX-13	52)SIGRTMAX-12	53)SIGRTMAX-11	54)SIGRTMAX-10
55)SIGRTMAX-9	56)SIGRTMAX-8	57)SIGRTMAX-7	58)SIGRTMAX-6
59)SIGRTMAX-5	60)SIGRTMAX-4	61)SIGRTMAX-3	62)SIGRTMAX-2
63)SIGRTMAX-1	64)SIGRTMAX		

Linux 的信号机制是从 Unix 继承下来的，早期 Unix 系统只定义了 32 种信号，现在 Linux 支持了 64 种信号，而且将来可能进一步增加。

前 32 种信号已经有了预定义值，每个信号有了确定的用途及含义，并且每种信号都有各自的缺省动作。如按键盘的“CTRL^C”时，会产生 SIGINT 信号，对该信号的默认反应就

是进程终止。这 32 种信号在实践过程中暴露出一个主要的缺陷是信号有可能丢失。这些信号也称为非实信号。

后 32 个信号是实时信号，是 Linux 经过改进和扩充原始机制之后的，支持信号排队，这保证了发送的多个实时信号都被接收。

Linux 支持的信号列表如下。很多信号是与机器的体系结构相关的，首先列出的是 POSIX.1 中列出的信号：

信号	值	处理动作	发出信号的原因
SIGHUP	1	A	终端挂起或者控制进程终止
SIGINT	2	A	键盘中断（如 break 键被按下）
SIGQUIT	3	C	键盘的退出键被按下
SIGILL	4	C	非法指令
SIGABRT	6	C	由 abort(3)发出的退出指令
SIGFPE	8	C	浮点异常
SIGKILL	9	AEF	Kill 信号
SIGSEGV	11	C	无效的内存引用
SIGPIPE	13	A	管道破裂: 写一个没有读端口的管道
SIGALRM	14	A	由 alarm(2)发出的信号
SIGTERM	15	A	终止信号 网管 bitscn_com
SIGUSR1	30,10,16	A	用户自定义信号 1
SIGUSR2	31,12,17	A	用户自定义信号 2
SIGCHLD	20,17,18	B	子进程结束信号
SIGCONT	19,18,25		进程继续（曾被停止的进程）
SIGSTOP	17,19,23	DEF	终止进程
SIGTSTP	18,20,24	D	控制终端（tty）上按下停止键
SIGTTIN	21,21,26	D	后台进程企图从控制终端读
SIGTTOU	22,22,27	D	后台进程企图从控制终端写

3.1.2 进程对信号的响应和处理

进程可以通过三种方式来响应和处理一个信号：

- ✓ 忽略信号，即对信号不做任何处理，但是有两个信号不能忽略：SlGKILL 及 SIGSTOP。SIGKILL 使得正在运行的进程退出运行，SIGSTOP 用于暂停一个正 在运行的进程。
- ✓ 捕捉信号，当信号发生时，执行用户定义的信号处理函数。
- ✓ 执行缺省操作，Linux 对每种信号都规定了默认操作。

信号之间不存在相对的优先权。信号在产生时也并不马上送给进程，信号必须等待直到进程再一次被调度运行。每当一个进程从系统调用中退出时，系统都将检查进程的信号。

进程对收到的信号默认都会进行缺省操作，如果进程要处理某一信号，那么就要在进程中安装该信号。安装信号主要用来确定要忽略某个信号或者捕捉信号，执行相应的处理函数。

信号的安装主要有两个函数，其函数原型是：

```
#include<signal.h>
signal(int sig, void( *func)(int));
int sigaction(int signum，const struct sigaction *act，struct sigaction *oldact);
```

其中 signal()在可靠信号系统调用的基础上实现，是库函数。它只有两个参数，不支持信号传递信息，主要是用于前 32 种非实时信号的安装。signal()函数的第一个参数是要安装的信号值，第二个参数是对第一个参数指定的信号的处理，可以下面两个宏：

- ✓ SIG_DFL：采用系统默认的方式处理信号，执行缺省操作。
- ✓ SIG_IGN：忽略信号。

另外，第二个参数也可以是一个函数指针，这个函数是用户自定义的对信号的处理函数。

我们来看一个例子，在描述多进程的章节的时候，我们曾经提到过父进程为了回收已经终止了的子进程，需要调用 wait()函数，但是 wait()函数会阻塞主进程，影响程序流程。子进程终止的时候，都会给父进程发送 SIGCHILD 信号，那么我们可以通过信号安装的方式，来捕捉 SIGCHILD，在收到这个信号的时候调用 wait()函数来回收子进程。

示例 3-1：用 signal()捕捉信号

```
#include<stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

//extern char **environ;

/*SIGCHLD 捕捉 SIGCHLD 信号 捕捉到该信号说明有子进程退出，调用 wait 函数回收子进程信息*/
void    handle_sig_child()
{
    int ret;
    pid_t pid_c;

    pid_c = wait(&ret);
    printf("Child process PID [%d] return [%d].\n", pid_c, ret);
}

int main(int argc, char **argv)
```

```
{
    pid_t pid;

    /*安装 SIGCHLD 信号，当一个进程的子进程退出时系统发送一个信号
SIGCHLD 给本进程*/
    signal( SIGCHLD, handle_sig_child );

    pid = fork();

    if( pid == 0 ){
        printf("Child PID [%d] will exit with status 0.\n", getpid() );
        exit(0);
    }else if( pid != -1 ){
        while( 1 ){
            sleep( 1 );
        }
    }else{
        printf("There was an error with forking!\n");
    }
    return 0;
}
```

从这段代码中可以看得出来，父进程不需要被阻塞，因此不必为了回收子进程而改变程序流程，而且程序效率更高。下面是程序运行的结果：

```
[alex@alex-/work/tutorail]$./signal_test
Child PID [13370] will exit with status 0.
Child process PID [13370] return [0].
```

当了进程退出时，父进程立即收到 SIGCHLD 信号并回收了子进程。

3.1.3 信号的发送

除了内核和超级用户，并不是每个进程都可以向其他的进程发送信号。一般的进程只能向具有相同 uid 和 gid 的进程发送信号，或向相同进程组中的其他进程发送信号。常用的发送信号的函数有 kill()、raise ()、alarm()、setitimer()、abort()等，下面我们来逐一介绍。

kill()函数可以给指定的进程发送某一个信号，其函数原型是：

```
#include<sys/types.h>
#include<signal.h>
int kill(pid_t pid, int sig);
```

第一个参数 pid 的各个取值意义如下：

- ✓ pid>0，给 PID 为 pid 的进程发送信号。
- ✓ pid=0，给同一个进程组的所有进程发送信号。
- ✓ pid<0 且 pid!=-1，给进程组 ID 为 pid 的绝对值的所有进程发送信号。
- ✓ pid=-1，给除了自身之外的 PID 大于 1 的进程发送信号。

第二个参数是要发送的信号值。当第二个参数为 0 的时候，实际上不会发送任何信号，通常用于错误检查,因此可以用来检查目标进程是否存在或者进程是否具有向目标进程发送信号的权限。

当 kill()函数成功发送一个信号时，函数返回 0，否则返回-1。通过 errno 以及使用 perror 可以查看错误信息：

EINVAL：所发送的信号无效。

- ✓ EPERM： 没有向目标进程发送信号的权限。
- ✓ ESRCH： 目标进程不存在或者进程已经终止，处于僵尸状态。

我们来看一个用 kill()函数发送信号的例子，请注意主进程中的信号处理函数和示例 3-1 中的信号处理函数的区别：示例 3-2 的信号处理函数多了一个整型参数，通过这个整型参数，信号处理函数可以获得进程收到的信号值，这样我们可以在同一个信号处理函数中对多个信号进行处理。同样，在安装信号的时候，signal()函数的第二个参数都是这个信号处理函数的函数指针。

在这个例子中，我们还定义了一个 pid_printf()函数，在每行输出语句前都自动插入当前进程 ID，这样可以方便的从输出结果中看到每一句是被哪个进程输出的。

示例 3-2：用kill()函数发送信号

源代码：kill_test.c

```
#include <stdarg.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include<stdlib.h>
#include <signal.h>

//extern char **environ;

/*自定义一个pid_printf,在每行输出前插入进程ID*/
void pid_printf( char *format, ... )
{
    va_list ap;
    va_start( ap, format );
    printf("[%d]:", getpid() );
```

```
    vprintf( format, ap );
    va_end(ap);
}

/*信号处理函数*/
void signal_handler( int signo )
{
    int ret;
    pid_t pid_c;

    switch( signo ){
        case SIGCHLD:
            pid_c = wait( &ret );
            pid_printf("Child process PID [%d] return [%d].\n", pid_c,
ret);
            break;
        case SIGUSR2:
            pid_printf("Signal SIGUSR2 received\n");
            break;
        default:
            pid_printf("Signal [%d] received\n", signo);
            break;
    }
}

int main( int argc, char **argv )
{
    pid_t pid;

    /*安装SIGCHLD信号*/
    signal( SIGCHLD, signal_handler );
    /*安装SIGUSR2信号*/
    signal( SIGUSR2, signal_handler );

    pid = fork();

    if( pid == 0 ){
```

```
        pid_printf("Child process send signal SIGUSR2 to parent
process.\n" );
        kill( getppid(), SIGUSR2 );
        pid_printf("Child process will exit with status 0.\n" );
        exit(0);
    }else if( pid != -1 ){
        while( 1 ){
            sleep( 1 );
        }
    }else{
        pid_printf("There was an error with forking!\n");
    }
    return 0;
}
```

运行这个程序，输出结果如下：

```
[15322]:Child process send signal SIGUSR2 to parent process.
[15322]:Child process will exit with status 0.
[15321]:Child process PID [15322] return [0].
[15321]:Signal SIGUSR2 received
```

请注意一个细节，当子进程发出 SIGUSR2 信号的时候，父进程并不是立即捕捉到信号，因为父进程还需要等待调度。当进程被调度的时候，信号的处理也不一定是按照信号发送的顺序被处理的。那么如果连续多次给同一个父进程发送同一个信号，父进程又会接到几个信号呢？读者可以自己尝试一些。

raise()函数用户给进程本身发送一个信号，其函数原型为：

```
#include<signal.h>
int raise(int sig);
```

raise()函数的参数是发送的信号值。

示例 3-3：用 raise()函数发送一个信号

源代码：raise_test.c

```
#include <stdarg.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

//extern char **environ;//引用系统环境变量

/*自定义一个 pid_printf，在每行输出前插入进程 ID*/
```

```
void   pid_printf( char *format, ... )
{
    va_list ap;
    va_start( ap, format );
    printf("[%d]:", getpid() );
    vprintf( format, ap );
    va_end(ap);
}

/*信号处理函数*/
void   signal_handler( int signo )
{
    int ret;
    pid_t pid_c;

    switch( signo ){
        case SIGUSR2:
            pid_printf("Signal SIGUSR2 received\n");
            break;
        case SIGTERM:
            pid_printf("Signal TERM received\n");
            break;
        default:
            pid_printf("Signal [%d] received\n", signo);
            break;
    }
}

int main( int argc, char **argv )
{
    int i;

    /*安装 SIGUSR2 信号*/
    signal( SIGUSR2, signal_handler );
    /*安装 SIGTERM 信号*/
    signal( SIGTERM, signal_handler );

    raise( SIGUSR2 );
```

```
    sleep(1);
    raise( SIGTERM );
    sleep(1);
    pid_printf("raise SIGKILL\n");
    raise( SIGKILL );

    return 0;
}
```

程序运行结果如下：

```
[alex@alex-/work/tutorail]$./raise_test
[13643]:Signal SIGUSR2 received
[13643]:Signal TERM received
[13643]:raise SIGKILL
Killed
```

这个例子中，进程捕捉了 SIGTERM 信号，如果不捕捉这个信号，程序运行结果又会如何呢？读者可以自己修改代码试验一下。

alarm()函数是一个简单定时器，专为 SIGALRM 信号设计，其函数原型是：

```
#include<unistd.h>
unsigned int alarm(unsigned int seconds);
```

函数的参数是定时器的定时时间，单位为秒。当设置了 alarm()之后，在指定的 seconds 秒之后，将给线程本身发送一个 SIGALRM 信号。当参数 seconds 为 0 的时候，将清除当前进程的 alarm 设置。

调用 alarm()函数时，如果进程已经有一个未结束的 alarm，那么旧的 alarm 将被删除，并返回旧的 alarm 的剩余时间。否则 alarm()函数返回 0。

Setitimer()功能强大更强大的定时器函数，支持 3 种类型的定时器，但是从本质上，它是和 alarm 共享同一个进程内的定时器。其函数原型是：

```
int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);
```

setitimer()第一个参数 which 指定定时器类型：

ITIMER_REAL：设定绝对时间，经过指定的时间后，内核将发送 SIGALRM 号给本进程。

ITIMER_VIRTUAL：设定程序执行时间，只有程序被调度执行的时候才记录时间，经过指定的时间后，内核将发送 SIGVTALRM 信号给本进程。

ITIMER_PROF：设定进程执行以及内核因本进程而消耗的时间和，经过指定的时间后，内核将发送 ITIMER_VIRTUAL 信号给本进程。

第二个参数是结构 itimerval 的一个实例，详细结构可以查看 setitimer()函数的 man 手册页。setitimer()调用成功返回 0，否则返回-1。

abort()向进程发送 SIGABORT 信号，默认情况下进程会异常退出，当然可定义自己的信号处理函数，通常可以用于做程序退出前的统一操作和处理。

信号相关还有很多更复杂的操作，本小节只是介绍了信号相关的最基本函数的使用方法，但灵活使用这些基本函数也能满足大多数的需求。

3.2 信号量

在 UNIX 的 System V 版本，AT&T 引进了一种新形式的 IPC 功能(信号量、消息队列以及共享内存)。Linux 继承了 IPC 机制，下面我们将分别介绍信号量、消息队列和共享内存。

在介绍多线程编程的时候，我们已经介绍了用信号量来完成线程同步，用于线程同步的信号量是 POSIX 标准的无名信号量。现在我们来介绍通常用于进程同步的 System V 信号量。两种信号量的基本原理都是一样的，它们为资源建立一个计数器，当信号量的值大于 0 的时候表明有可使用的资源，当资源被使用，信号量减 1，信号量值等于 0 的时候，进程必须等待信号量再大于 0 的时候才能使用资源。但是 System V 信号量要比无名信号量要更复杂一些。

在介绍信号量之前，首先要先了解一下与各种进程间通信相关的 IPC 标识符和关键字。

3.2.1 IPC 标识符和关键字

System IPC 中,对于每一个新建的信号量、消息队列以及共享内存，都有一个在整个系统中唯一的标识符。每个标识也都有唯一对应的关键字，关键字的数据类型由系统定义为 key_t。

在终端输入命令“ipcs”,可以看到目前系统中所有的 IPC 信息：

```
[alex@alex-/work/tutorail]$ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 229376     user       600        393216     2          dest
0x00000000 262145     user       600        393216     2          dest
0x00000000 163842     user       600        393216     2          dest
0x00000000 196611     user       600        393216     2          dest
0x00000000 294916     user       600        393216     2          dest
0x00000000 327685     user       600        393216     2          dest
0x00000000 360454     user       600        393216     2          dest
 ------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x6406f9da 3080201     alex       600          1
------ Message Queues --------
```

```
key          msqid       owner       perms       used-bytes    messages
0x80600001   0           alex        600         16361         226
```

第一列的 key 就是 IPC 的关键字，第二列是 IPC 的标识符。

fotk()函数用于获得一个 IPC 关键字，其函数原型是：

```
#include<sys/types.h>
#include<sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

ftok()函数返回一个以 pathname 指向的文件相对应的文件和 proj_id 来确定一个 IPC 关键字 ket_t。pathname 必须是一个已经存在并具有访问权限的文件，proj_d 只有最低的 8 个字节是有效的，所以通常用一个 ASCII 字符来作为 proj_id。当 pathname 和 proj_id 完全相同时，每次调用 ftok()都会获取一个相同的键值。

IPC 关键字可以由 ftok()函数获得，也可以设为 IPC_PRIVATE。这时操作系统会确保创建一个新的 IPC，其标识符需要由进程自己记录并告诉其他进程。

3.2.2 创建或获取信号量

通过调用 semget()创建信号量集，其函数原型是：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

第一个参数是信号量键值。第二个参数是信号量的数目。第三个参数是一些标志。这些标志包括：

- ✓ IPC_CREAT：如果 key 指定的信号量不存在，创建一个新信号量集。
- ✓ IPC_EXCL：和 IPC_CREAT 标志一起使用，如果信号量已经存在，返回错误。

semget()调用成功时，返回与键值 key 对应的信号量集的标识符，否则返回-1。

3.2.3 信号量操作

当进程需要申请或者释放公共资源的时候，可以调用 semop()来对信号量进行操作，其函数原型是：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

参数 semid 是信号量 ID,第二个参数 sops 指向一个 sembuf 结构的数组,每一个 sembuf 结构都定义了在对信号量的操作，第三个参数 nsops 为 sops 指向数组的大小。

其中 sembuf 结构如下：

```
struct sembuf{
        unsigned short      sem_num;        /*信号量在信号量集中的索
引号*/
        short               sem_op;         /*对信号量的操作*/
        short               sem_flg;        /*操作标志位*/
} ;
```

sem_num 对应信号集中的信号量的索引号，0 对应第一个信号量，1 对应第二个信号量，依此类推。

sem_op 是一个指定了操作类型的整数.如果 sem_op 是一个正整数，则这个值会立刻被加到信号量的值上 .如果 sem_op 为负，则将从信号量值中减去它的绝对值。如果这将使信号量的值小于零，则这个操作会导致进程阻塞，直到信号量的值至少等于操作值的绝对值（由其它进程增加它的值）。 如果 sem_op 为 0，这个操作会导致进程阻塞，直到信号量的值为零才恢复。

sem_flg 是一个符号位。指定 IPC_NOWAIT 以防止操作阻塞；如果该操作本应阻塞，则 semop 调用会失败。如果为 sem_flg 指定 SEM_UNDO，Linux 会在进程退出的时候自动撤销该次操作。semop()调用成功时返回 0，否则返回-1。

semtimedop()的功能和 semop()基本一样，只是增加了一个时间限制，其函数原型如下：

```
int semtimedop(int semid, struct sembuf *sops, unsigned nsops, struct
timespec *timeout);
```

如果信号量操作时进程被阻塞，那么经过 timeout 时间后，还是没有可用资源，那么函数会立即返回。

3.2.4 信号量控制

semctl()系统调用用于对信号量的各种控制操作，其函数原型为：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
int semctl(int semid, int semnum, int cmd, …);
```

第一个参数 semid 指定信号量集，第二个参数 semnum 指定了对信号量集里面的哪一个信号量进行操作，第三个参数 cmd 指定具体的操作类型，表 3-2 列出了各种 cmd 操作，其他参数用于设置或返回信号量信息。

表3-2 semctl各种操作

Cmd	操作

IPC_STAT	获取信号量信息，信息由 arg.buf 返回，记录在 semid_ds 结构体。
IPC_SET	设置信号量信息，待设置信息保存在 arg.buf 中。
IPC_RMID	立即删除信号量集。
IPC_INFO	获取信号量集信息，由 arg.buf 返回，记录在一个 struct seminfo 结构体。
SEM_INFO	返回和 IPC_INFO 一样 seminfo 结构体，只是部分字段的含义有所区别。
SEM_STAT	返回和 IPC_STAT 一样的 semid_ds 结构体，只足部分字段含义有所区别。
GETALL	返回所有信号量的值，结果保存在 arg.array 中，参数 sennum 被忽略。
GETNCNT	返回等待 semnum 所代表信号量的值增加的进程数，相当于目前有多少进程在等待 semnum 代表的信号量所代表的共享资源。
GETPID	返回最后一个对 semnum 所代表信号量执行 semop 操作的进程 ID。
GETVAL	返回 semnum 所代表信号量的值；GETZCNT 返回等待 semnum 所代表信号量的值变成 0 的进程数。
GETZCNT	返回等待 semnum 锁代表的信号量值成为 0 的进程数目。
SETALL	通过 arg.array 更新所有信号量的值；同时，更新与本信号集相关的 semid_ds 结构的 sem_ctime 成员。
SETVAL	设置 semnum 所代表信号量的值为 arg.val。

semctl()调用失败时返回-1，调用成功时，根据 cmd 的不同，返回值也各不相同，具体可以查看 semctl 的 man 手册页。

3.2.5 信号量综合示例

首先来实现两个函数，用于请求和释放信号量，每个进程只能请求一次，所以我们定义了一个全局变量 semheld 来记录请求次数，当 semheld 大于 0 时不再增加信号量，当 semheld 小于 1 时，不再释放信号量。

示例3-9：信号量综合示例

源代码：ipc_sem_test.c

```
/*释放信号量*/
```

```
void sem_release(int id)
{
  struct sembuf sb;

  if(semheld<1){
    pid_printf("I don't have any reources; nothing to release\n");
    return;
  }

  sb.sem_num=0;
  sb.sem_op=1;
  sb.sem_flg = SEM_UNDO;
  if(semop(id,&sb,1) == -1){
    pid_printf("Semop release error: %s\n",strerror(errno));
    exit(-1);
  }
  semheld--;

  pid_printf("Resource released.\n");
}

//请求信号量
void  sem_request(int id)
{
    struct sembuf sb;

    if(semheld>0){
    pid_printf("I already hold the resource; not requesting another
one.\n");
    return;
    }

    sb.sem_num =0;
    sb.sem_op = -1;
    sb.sem_flg=SEM_UNDO;

    pid_printf("Requesting resource ...");
    fflush(stdout);
```

```
    if(semop(id,&sb,1)==-1){
    pid_printf("Semop request error: %s\n",strerror(errno));
    exit(-1);
    }
  semheld++;
  printf("Done.\n");
}
```

sem_delete()函数用于创建信号量的进程退出时，立即删除信号量集：

```
/*删除信号量集*/
void sem_delete(void)
{
  printf("Master exiting; delete semaphore.\n");
  if(semctl(id,0,IPC_RMID,0)== -1){
    pid_printf("Error releasing semaphone.\n");
  }
}
```

下面是主函数，如果程序没有传入参数，那么程序将创建一个新的信号量集。如果程序传入了一个参数，这个参数应该是一个已经存在的信号量集 ID，那么程序直接使用这个信号量集 ID。当进程创建了新的信号量集的时候，调用了 atexit(&sem_delete)函数，使得程序退出时，信号量集被释放。

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<stdarg.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include<errno.h>
#include<sys/shm.h>

#if defined( __GNU_LIBRARY__)&&!defined(_SEM_SEMUN_UNDEFINED)

#else
  union semun{
    int    val;/*Value for SETVAL*/
    struct semid_ds *bur; /*Buffer for IPC_STAT,IPC_SET*/
    unsigned short *array; /*Array for GETALL,SETALL*/
```

```
    struct seminfo *__but; /*Buffer for IPC_INFO(Linux specific)*/
  };
#endif

int semheld = 0;/*此实例中，一个进程申请一次信号量*/

int id = 0;
void pid_printf(char *format,...)
{
  va_list ap;
  va_start(ap,format);
  printf("[%d]:",getpid());
  vprintf(format,ap);
  va_end(ap);
}
int main(int argc, char **argv)
{
  union semun sunion;

  if(argc<2){
    id = semget(IPC_PRIVATE,1,SHM_R | SHM_W);

    if(id !=-1){
      atexit(&sem_delete);
       sunion.val=1;
       if(semctl(id,0,SETVAL,sunion) ==1){
        pid_printf("semctl  failed: %s\n",strerror(errno));
          exit(-1);
      }
    }

  }else{
    id = atoi(argv[1]);
      pid_printf("Using existing semaphore %d.\n",id);
  }

  if(id==-1){
  pid_printf("Semaphore  request  failed: %s.\n", strerror(errno));
```

```
  return 0;
  }

  pid_printf("Successfully allocated semaphore id %d.\n",id);

  while(1){
    int action;
    printf("\nStatus: %d request held by this process.\n", semheld);

    printf("Please select:\n");
    printf("1. Release a resource\n");
    printf("2. Request a resource\n");
    printf("3. Exit this process\n");
    printf("Your choice:");

    scanf("%d",&action);

    switch(action){
      case 1:
        sem_release(id);
        break;
        case 2:
        sem_request(id);
        break;
        case 3:
        exit(0);
        break;
    }
  }
  return 0;
}
```

测试这个程序，需要分别在两个终端进行操作。现在第一个终端运行程序，并选择“2”申请资源。屏幕输出如下：

```
[alex@alex-/work/tutorail]$./ipc_sem_test
[14459]:Successfully allocated semaphore id 98305.

Status: 0 request held by this process.
Please select:
```

```
1. Release a resource
2. Request a resource
3. Exit this process
Your choice:2
Requesting resource ...Done.

Status: 1 request held by this process.
Please select:
1. Release a resource
2. Request a resource
3. Exit this process
Your choice:
```

进程创建了一个 ID 为 98305 的信号量集，并申请了一个资源。我们从另外一个终端通过 ipcs 命令也可以看到这个信号量集：

```
[alex@alex-/work/tutorail]$./ipcs  -s

------ Semaphore Arrays --------
key         semid      owner      perms      nsems
0x00000000 65536      user       600        1
0x00000000 98305      user       600        1
0x00000000 131074     user       600        1
```

在另外一个终端再执行这个程序，将信号量 ID 作为参数传入：

```
[alex@alex-/work/tutorail]$./ipc_sem_test 98305
[14536]:Using existing semaphore 98305.
[14536]:Successfully allocated semaphore id 98305.

Status: 0 request held by this process.
Please select:
1. Release a resource
2. Request a resource
3. Exit this process
Your choice:2
[14536]:Requesting resource ...
```

选择“2”，这时程序会被阻塞，知道另外一个进程把资源释放掉。切换到第一个终端，输入“1”，此时可以看到第二个终端的程序马上申请到了资源。

依次把两个程序都退出，再用 ipcs 命令查看当前系统中的信号量，可以看到信号量集已经被删除了。

3.3 消息队列

3.3.1 消息队列基本概念

消息队列是系统内核地址空间中的一个内部的链表。消息可以按照顺序发送到队列中，也可以以几种不同的方式从队列中读取。每一个消息队列用一个唯一的 IPC 标识符表示。

了解在系统内核中的数据结构是了解 IPC 机制如何工作的最好的方法。

首先我们看一下数据结构 msgbuf。此数据结构可以说是消息数据的模板。虽然此数据结构需要用户自己定义，但了解系统中有这样一个数据结构是十分重要的。在<sys/msg.h>中，此数据结构是这样定义的：

```
struct msgbuf {
    long mtype;          /*type of message, must＞0*/
    char mtext[1];       /*message text*/
} ;
```

在数据结构 msgbuf 中共有两个元素：

- ✓ mtype 指消息的类型，它由一个整数来代表，并且它只能是大于 0 的整数。
- ✓ mtext 是消息数据本身。

mtext 字段不但可以存储字符，还可以存储任何其他的数据类型。此字段可以说是完全任意的，因为程序员自己可以重新定义此数据结构。请看下面重新定义的例子：

```
struct my_msgbuf{
    long mtype;            /*Message type*/
    char request_id;       /*Request identifier*/
    struct client info;    /*Client information structure*/
};
```

这里的消息类型字段和前面的一样，但数据结构的其余部分则由其他的两个字段所代替，而其中的一个还是另外一个结构。这就体现了消息队列的灵活之处。内核本身并不对消息结构中的数据做任何翻译。你可以在其中发送任何信息，但存在一个内部给定的消息大小的限制。在 Linux 系统中，消息的最大的长度是 4056 个字节，其中包括 mtype，它占用 4 个字节的长度。

3.3.2 创建消息队列

系统调用 msgget()用于创建一个新的消息队列，或者存取一个已经存在的消息队列，其函数原型是：

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
```

```
int msgget(key_t key, int msgflg);
```

系统调用 msgget()中的第一个参数是消息队列关键字值,可以由上一节我们介绍过的 ftok()获得。第二个参数 msgflg 是一些标志，包括：

IPC_CREAT：如果内核中没有此队列，则创建它。

IPC_EXCL：当和 IPC_CREAT 一起使用时，如果队列已经存在，则返回错误。

当 msgget()执行成功时，返回消息队列的标识符，否则返回-1，通过 errno 和 perror()函数可以查看错误信息。

下面是一个打开和创建一个消息队列的例子，函数返回消息队列的标识符：

示例3-10：创建一个消息队列

```
int open_ queue(key_ t keyval)
{
  int qid;
  if((qid = msgget(keyval, IPC_CREAT|0660)) == -1){
    perror(”msgget”);
    return(-1);
  }
  return(qid);
}
```

3.3.3 发送和接收消息

当得到了消息队列标识符，就可以在队列上执行发送或者接收消息了。msgsnd()系统调用用于向队列发送一条消息，其函数原型是：

```
int msgsnd(int msqid, struct msgbuf *msgp, sizet msgsz, int msgflg);    I
```

第一个参数是消息队列标识符。第二个参数 msgp,是指向消息缓冲区的指针。参数 msgsz 指定了消息的字节大小，但不包括消息类型的长度(4 个字节)。参数 msgflg 可以设置为：

- ✓ 0：此时为忽略此参数，如果消息队列已满，调用进程将会挂起，直到消息可以写入到队列中。
- ✓ IPC_NOWAIT：如果消息队列已满，那么此消息则不会写入到消息队列中，控制将返回到调用进程中。消息队列写入成功时，函数返回 0，否则返回-1。

下面是一个发送消息的例子：

示例 3-11：发送一条消息

```
int send_message(int qid, struct mymsgbuf *qbuf)
{
```

```
  int result, length;
  /*The length is essentially the size of the structure minus
sizeof(mtype)*/
  length = sizeof(struct mymsgbuf) - sizeof(long);
  if((result = msgsnd(qid, qbuf, length, 0)) == -1){
     perror(”msgsnd”);
     return(-1);
  }
  return(result);
}
```

msgrcv()系统调用用于从消息队列读取一条消息，其函数原型是：

```
ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long
msgtype, int msgflg);
```

第一个参数是消息队列的标识符。第二个参数代表要存储消息的缓冲区的地址。第三参数是消息缓冲区的长度，不包括 mtype 的长度，它可以按照如下的方法计算：

msgsz=sizeof(struct mymsgbuf)-sizeof(long);

第四个参数是要从消息队列中读取的消息的类型。

如果 msgtype=0,接收消息队列的第一个消息。大于 0 接收队列中消息类型等于这个值的第一个消息。小于 0 接收消息队列中小于或者等于 msgtype 绝对值的所有消息中的最小一个消息。 一般为 0。

第五个参数 msgflg 取值为：

- ✓ 0：从队列中取出最长时间的一条消息。
- ✓ IPC_NOWAIT：当队列没有消息时，调用会立即返回 ENOMSG 错误。否则，调用进程将会挂起，直到队列中的一条消息满足 msgrcv()的参数要求。

当函数成功时，返回写入缓冲区的数据大小，否则返回-1。

下面是一个接收消息的例子：

示例 3-12：接收一条消息

```
int read_message(int qid, long type, struct mymsgbuf *qbuf)
{
  int result, length;
  /* The length is essentially the size of the structure minus
sizeof(mtype)*/
  length = sizeof(struct mymsgbuf) - sizeof(long);
  if((result = msgrcv(qid, qbuf, length, type, 0))==-1){
  perror(¨msgrcv¨);
  return(-1);
  }
```

```
  return(result);
}
```

3.3.4 消息队列的控制

消息队列标识符的属性被记录在一个 msgid_ds 结构体：

```
struct msqid_ds{
  struct ipc_perm msg_perm;      /*所有者和权限*/
  time_t          msg_stime;     /*最后一次向队列发送消息的时间*/
  time_t          msg_rtime;     /*最后一次从队列接收消息的时间*/
  time_t          msg_ctime;     /*队列最后一次改动的时间*/
  unsigned long   __msg_cbytes;  /*当前队列所有消息的总长度*/
  msgqnum_t       msg_qnum;      /*当前队列中的消息数量*/
  msglen_t        msg_qbytes;    /*消息队列的最大消息总长度*/
  pid_t           msg_lspid;     /*最一次给队列发送消息的进程PID*/
  pid_t           msg_lrpid;     /*最后一次从队列接收消息的进程PID*/
}
```

通过 msgctl()可以对消息队列进行控制或者一些属性的修改，其函数原型为：

```
int msgctl(int msqid, int cmd, struct msqid ds *buf;
```

第一个参数是消息队列的标识符，第二个参数 cmd 指定了操作，下面是几个常用的操作：

IPC_STAT：读取消息队列的数据结构 msqid_ds，并将其存储在 buf 指定的地址中。

IPC_SET：设置消息队列的数据结构 msqid_ds 中的 ipc_perm、msg_qbytes、msg_ctime 元素的值。这个值取自 buf 参数。

IPC_RMID：从系统内核中移走消息队列。

比如下面是一个删除消息队列的例子。

示例3-13：删除消息队列

```
int remove_queue(int qid){
  if(msgctl(qid, IPC_RMID, 0)==-1){
    perror(”msgctl”);
    return(-1);
  }
  return(0);
}
```

3.3.5 综合示例 msgtool

我们来实现一个简单的消息队列工具，用于创建消息队列、发送、读取消息、改变权限以及删除消息队列。

它的用法如下：

（1） 发送消息

```
msgtool s (type)”text”
```

（2） 读取消息

```
msgtool r (type)
```

（3） 改变权限

```
msgtool l m (mode)
```

（4） 删除队列

```
msgtool d
```

示例 3-14：msgtool

源代码 msgtool.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_SEND_SIZE 80
struct mymsgbuf {
    long mtype;
    char mtext[MAX_SEND_SIZE];
};

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text);
void read_message(int qid, struct mymsgbuf *qbuf, long type);
void remove_queue(int qid);
void change_queue_mode(int qid, char *mode);
void usage(void);

int main(int argc, char *argv[])
{
    key_t key;
```

```
    int msgqueue_id;
    struct mymsgbuf qbuf;
    if(argc == 1)
        usage();
    /* Create unique key via call to ftok() */
    key = ftok(".", 'm');
    /* Open the queue - create if necessary */
    if((msgqueue_id = msgget(key, IPC_CREAT|0660)) == -1) {
        perror("msgget");
        exit(1);
    }

    printf("message queue id = [%d]\n", msgqueue_id );

    switch(tolower(argv[1][0]))
    {
        case 's':
            if( argc < 4 ){
                usage();
                break;
            }
            send_message(msgqueue_id, (struct mymsgbuf *)&qbuf,
atol(argv[2]), argv[3]);
            break;
        case 'r':
            if( argc < 3 ){
                usage();
                break;
            }
            read_message(msgqueue_id, &qbuf, atol(argv[2]));
            break;
        case 'd':
            remove_queue(msgqueue_id);
            break;
        case 'm':
            if( argc < 3 ){
                usage();
                break;
```

```
            }
            change_queue_mode(msgqueue_id, argv[2]);
            break;
        default: usage();
    }
    return(0);
}

void send_message(int qid, struct mymsgbuf *qbuf, long type, char *text)
{
    /* Send a message to the queue */
    printf("Sending a message ...\n");
    qbuf->mtype = type;
    strcpy(qbuf->mtext, text);
    if((msgsnd(qid, (struct msgbuf *)qbuf, strlen(qbuf->mtext)+1, 0))
==-1)
    {
        perror("msgsnd");
        exit(1);
    }
}

void read_message(int qid, struct mymsgbuf *qbuf, long type)
{
    /* Read a message from the queue */
    printf("Reading a message ...\n");
    qbuf->mtype = type;
    msgrcv(qid, (struct msgbuf *)qbuf, MAX_SEND_SIZE, type, 0);
    printf("Type: %ld Text: %s\n", qbuf->mtype, qbuf->mtext);
}

void remove_queue(int qid)
{
    /* Remove the queue */
    msgctl(qid, IPC_RMID, 0);
}

void change_queue_mode(int qid, char *mode)
```

```
{
    struct msqid_ds myqueue_ds;
    /* Get current info */
    msgctl(qid, IPC_STAT, &myqueue_ds);
    /* Convert and load the mode */
    sscanf(mode, "%ho", &myqueue_ds.msg_perm.mode);
    /* Update the mode */
    msgctl(qid, IPC_SET, &myqueue_ds);
}

void usage(void)
{
    fprintf(stderr, "msgtool - A utility for tinkering with msg
queues\n");
    fprintf(stderr, "USAGE: msgtool (s)end <type> <messagetext>\n");
    fprintf(stderr, " (r)ecv <type>\n");
    fprintf(stderr, " (d)elete\n");
    fprintf(stderr, " (m)ode <octal mode>\n");
    exit(1);
}
```

我们先来发送一条消息：

```
[alex@alex ~/work/tutorail]$./msgtool s 1 ”Hello”
message queue id = [32768]
Sending a message...
```

用 ipcs 命令可以看到创建的消息队列：

```
[alex@alex 一/work/tutorail]$  ipcs -q


------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x6d01c86f 32768      user       660        6            1
```

可以看到现在队列里有 1 条消息。

再用 msgtool 读出先读出一条消息，然后再次读取消息：

```
[alex@alex-4work/tutorai l]$./msgtool r 0
message queue id = [32768]
Reading a message ...
Type: 1 Text: Hello
[alex@alex-/work/tutorai l]$./msgtool r 0
message queue id 2[32768]
```

```
Reading a message...
```

此时消息队列已经为空，进程被阻塞等待消息。我们到另外一个终端发送一条消息：

```
[alex@alex-/work/tutorai l]$./msgtool s 2 “Another message”
message queue id=[32768]
Sending a message…
```

这是第一个终端的 msgtool 收到消息：

```
    Type：2 Text：Another message
```

下面我们修改消息队列的访问权限：

```
[alex@alex-/work/tutorai l]$./msgtool m 600
message queue id=[32768]
[alex@alex 一/work/tutorail]$ ipcs  -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x6d01c86f 32768      user       660        6            1
```

从 ipcs 命令看到的消息队列中，perms 列已经被改为 600。最后删除消息队列：

```
[alex@alex-/work/tutorai l]$./msgtool d
message queue id=[32768]
[alex@alex 一/work/tutorail]$ ipcs  -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

3.4 共享内存

共享内存可以说是最有用的进程间通信方式，也是最快的 IPC 形式。两个不同进程 A、B 共享内存的基本原理是，同一块物理内存被映射到进程 A、B 各自的进程地址空间。进程 A 可以即时看到进程 B 对共享内存中数据的更新，反之亦然。由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。

3.4.1 创建和获取共享内存

系统调用 shmget()用于创建共享内存或者获取一个已经存在的共享内存的标识符，其函数原型是：

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

```
int shmget(key_t key, size_t size, int shmflg);
```

系统调用 shmget()中的第一个参数是共享内存关键字值，可以由前面介绍过的 flok()获得。第二个参数 size 是共享内存的大小，第一个参数 shmflg 是一些标志，包括：

IPC_CREAT：如果内核中没有此共享内存，则创建它。

IPC_EXCL：当和 IPC_CREAT 一起使用时，如果共享内存已经存在，则返回错误。

当 shmget()执行成功时，返回共享内存的标识符，否则返回-1，通过 errno 和 perror()函数可以查看错误信息。

下面的函数是获取一个共享内存的例子，使用 IPC_CREAT 参数，如果共享内存不存在，那么创建一个新的共享内存。

示例 3-15：获取共享内存标识符

```
int open_segment(key_t keyval, int segsize){
    int shmid;
    if((shmid = shmget(keyval, segsize, IPC_CREAT|0660)) == -1){
        perror(" shmget ");
        return(-1);
    }
    return(shmid);
}
```

3.4.2 连接共享内存

系统调用 shmat()可以获取一个共享内存的地址，并将其连接到进程中，其函数原型是：

```
void *shmat(int shmid, const void *shnaddr, int shmflg);
```

第一个参数是共享内存标识符。第二个参数 shmaddr 和 shmflg 相关联：

✓ 如果 shmaddr 为 0，则共享内存连接到由内核选择的第一个可用的地址上。

如果 shmaddr 为非 0，并且 shmflg 没有指定 SHM_RND，则共享内存被连接到 shmaddr 指定的地址上。

如果 shmaddr 为非 0，并且 shmflg 指定了 SHM_RND，则共享内存被连接到 shmaddr 指定的地址向下取最近一个 SHMLBA 的地址倍数的地址。

如果 shmflg 指定了 SHM_RDONLY，则以只读方式连接此共享内存，否则以读写方式连接。

shmget()成功地连接地址时，返回该段所连接地实际地址，如果出错则返回-1。

当一个进程不在需要共享的内存时，用 shmdt()系统调用把共享内存从其地址空间中脱接，但这不等于将共享内存段从系统内核中删除。shmdt()的函数原型是：

```
int shmdt(const void *shmaddr);
```

参数 shmaddr 是调用 shmat()函数连接共享内存时获取的地址。当共享内存脱接成功时返回 0，否则返回-1。

3.4.3 共享内存的控制

共享内存标识符的属性被记录在一个 shmid_ds 结构里：

```
struct shmid—ds{
    struct ipc_perm shm_perm;    /*所有者和权限*/
    size_t          shm_segsz;   /*共享内存的的字节大小*/
    time_t          shm_amime;   /*最后一个进程连接共享内存的时间*/
    time_t          shm_dtime;   /*最后一个进程脱连共享内存的时间*/
    time_t          shm_ctime;   /*共享内存最后被修改的时间*/
    pid_t           shm_cpid;    /*创建共享内存的进程PID*/
    pid_t           shm_lpid;    /*最后一个连接或脱连的进程PID*/
    shmatt_t        shm_nattch;  /*当前连接到共享内存的进程总数*/
  …
}
```

通过 shmctl()可以对消息队列进行控制或者一些属性的修改，其函数原型为：

```
int shmct(int shmid, ind cmd, struct shmid_ds *buf);
```

第一个参数是共享内存的标识符，第二个参数 cmd 指定了其操作，下面是几个常用的操作：

- ✓ IPC_STAT：读取一个共享内存的数据结构 shmid_ds，并将其存储在 buf 指定的地址中。
- ✓ IPC_SET：设置消息队列的数据结构 shmid_ds 中各个元素的值。这个值取自 buf 参数。
- ✓ IPC_RMID：把共享内存标记为可删除，当最后一个进程脱连此共享内存的时候，系统将删除该共享内存。

3.4.4 综合示例 shmtool

我们来实现一个简单的共享内存操作工具，用于创建共享内存、写读共享内存、改变权限以及删除共享内存。在读写操作的时候，如果已经共享内心已经存在，直接进行连接并读写，否则先创建一个共享内存

它的用法如下：

（1） 把字符串写入共享内存

```
shmtool w ”text”
```

（2） 从共享内存中读取文件

```
shmtool r
```

（3） 改变权限

```
shmtool m (mode)
```

（4） 删除共享内存

```
shmtool d
```

下面是示例程序的源代码：

示例 3-16：共享内存工具 shmtooi

源代码： shmtool.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SEGSIZE 100

void    writeshm(int shmid, char *segptr, char *text)
{
    strcpy(segptr, text);
    printf("Done...\n");
}

void    readshm(int shmid, char *segptr)
{
    printf("segptr: %s\n", segptr);
}

void    removeshm(int shmid) {
    shmctl(shmid, IPC_RMID, 0);
    printf("Shared memory segment marked for deletion\n");
}

void    changemode(int shmid, char *mode)
{
    struct shmid_ds myshmds;
    /* Get current values for internal data structure */
    shmctl(shmid, IPC_STAT, &myshmds);
    /* Display old permissions */
    printf("Old permissions were: %o\n", myshmds.shm_perm.mode);
    /* Convert and load the mode */
```

```
    sscanf(mode, "%ho", &myshmds.shm_perm.mode);
    /* Update the mode */
    shmctl(shmid, IPC_SET, &myshmds);
    printf("New permissions are : %o\n", myshmds.shm_perm.mode);
}

void    usage() {
    fprintf(stderr, "shmtool - A utility for tinkering with shared
memory\n");
    fprintf(stderr, "USAGE: shmtool (w)rite <text>\n");
    fprintf(stderr, " (r)ead\n");
    fprintf(stderr, " (d)elete\n");
    fprintf(stderr, " (m)ode change <octal mode>\n");
    exit(1);
}


int main(int argc, char *argv[])
{
    key_t key;
    int shmid, cntr;
    char *segptr;

    if(argc == 1)
        usage();

    /* Create unique key via call to ftok() */
    key = ftok(".", 'S');

    /* Open the shared memory segment - create if necessary */
    if((shmid = shmget(key, SEGSIZE, IPC_CREAT|IPC_EXCL|0666)) == -1) {
        printf("Shared memory segment exists - opening as client\n");
        /* Segment probably already exists - try as a client */
        if((shmid = shmget(key, SEGSIZE, 0)) == -1) {
            perror("shmget");
            exit(1);
        }
    } else {
```

```
        printf("Creating new shared memory segment id = %d\n ", shmid);

    }

    /* Attach (map) the shared memory segment into the current process */
    if( ( segptr = ( char * )shmat(shmid, NULL, 0) ) == (void * )-1) {
        perror("shmat");
        exit(1);
    }

    switch(tolower(argv[1][0])){
        case 'w':
            writeshm(shmid, segptr, argv[2]);
            break;
        case 'r':
            readshm(shmid, segptr);
            break;
        case 'd':
            removeshm(shmid);
            break;
        case 'm':
            changemode(shmid, argv[2]);
            break;
        default:
            usage();
    }
}
```

首先我们用 shmtool 写入一个字符串：

```
[alex@alex～/work/tutorail]$./shmtool w ”Hello”
Creating new shared memory segment, id=[ 917516]
Done…
```

此时标识符为 917516 的共享内存被创建，且字符串被写入，用 ipcs 命令查看系统中的共享内存列表，可以看到这个新建的共享内存的信息：

```
[alex@!alex～/work/tutorail]$ipcs  -m

------ Shared Memory Segments --------
```

```
key        shmid      owner      perms      bytes      nattch
status
0x00000000 229376     user       600        393216     2          dest
0x00000000 262145     user       600        393216     2          dest
0x00000000 163842     user       600        393216     2          dest
0x00000000 196611     user       600        393216     2          dest
0x00000000 294916     user       600        393216     2          dest
0x00000000 327685     user       600        393216     2          dest
0x00000000 360454     user       600        393216     2          dest
0x00000000 393223     user       600        393216     2          dest
0x00000000 425992     user       600        393216     2          dest
0x00000000 458761     user       600        393216     2          dest
0x00000000 491530     user       600        393216     2          dest
0x00000000 524299     user       600        393216     2          dest
0x5301c86b 917516     user       666        100        0
```

下面再用 shmtool 读取共享内存中的文本：

```
[alex@alex-/work/tutorail]$./shmtool r
Shared memory segment exists-opening as client
segptr: Hello
```

修改共享内存权限：

```
[alex@alex-/work/tutorail]$./shmtool m 600
 Shared memory segment exists-opening as client
 Old permissions were: 666
 New permissions are: 600
[alex@alex～/work/tutorail]$ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch
status
0x00000000 229376     user       600        393216     2          dest
0x00000000 262145     user       600        393216     2          dest
0x00000000 163842     user       600        393216     2          dest
0x00000000 196611     user       600        393216     2          dest
0x00000000 294916     user       600        393216     2          dest
0x00000000 327685     user       600        393216     2          dest
0x00000000 360454     user       600        393216     2          dest
0x00000000 393223     user       600        393216     2          dest
0x00000000 425992     user       600        393216     2          dest
```

```
0x00000000 458761     user      600        393216     2          dest
0x00000000 491530     user      600        393216     2          dest
0x00000000 524299     user      600        393216     2          dest
0x5301c86b 917516     user      600        100        0
```

标记删除共享内存：

```
[alex@alex～/work/tutorail]$.]shmtool d
Shared memory segment exists - opening as client
Shared memory segment marked for deletion
[alex@alex-/work/tutorail]$ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch
status
0x00000000 229376     user      600        393216     2          dest
0x00000000 262145     user      600        393216     2          dest
0x00000000 163842     user      600        393216     2          dest
0x00000000 196611     user      600        393216     2          dest
0x00000000 294916     user      600        393216     2          dest
0x00000000 327685     user      600        393216     2          dest
0x00000000 360454     user      600        393216     2          dest
0x00000000 393223     user      600        393216     2          dest
0x00000000 425992     user      600        393216     2          dest
0x00000000 458761     user      600        393216     2          dest
0x00000000 491530     user      600        393216     2          dest
0x00000000 524299     user      600        393216     2          dest
```

由于没有其他进程连接到此共享内存，共享内存被立即删除。

在这个例子中，由于每次操作后进程都退出了，因此没有多个进程对该共享内存同时读写的可能。但是当多个进程都在同时访问共享内存的时候，需要通过一些同步机制来保证读写操作的原子性。通常使用信号量作为同步机制，请读者利用前面介绍过的信号量来完善这个共享内存工具。

测试题

3.1 GNU/Linux 提供的进程间通信方式有哪些？列出你所知道的。

3.2 为什么共享内存是最有效率的？他有哪些缺点？

3.3 进程信号量的阻塞操作可以用来实现进程间的　　。

3.4 共享内存和文件映射的本质区别是什么？

本章总结

本章主要介绍了进程间进行通信和同步的方法。首先介绍最古老的信号机制，最后介绍了 Linux 系统支持的进程间通信机制(Interprocess Communication，简称 IPC)：包括信号量、消息队列、共享内存。

第四章 网络编程(上)

引言：

学完本章内容以后，你将能够

了解网络的概念

理解 OSI 七层网络模型

掌握 TCP 套接字编程

4.1 网络基础

4.1.1 引言

Intemet 近年来获得了飞速的发展。20 年前，很少有人接触过网络。现在网络已成为我们社会结构的一个基本组成部分。网络被用于工商业的各个方面，包括广告宣传、生产、运输、计划、报价和会计等。因此，绝大多数公司拥有了多个网络。从小学到研究生教育的各级学校都使用计算机网络为教师和学生提供全球范围的联网图书信息的即时检索。简而言之，计算机网络已遍布各个领域。而这一切要归功于计算机联网协议的发展。

计算机网络中实现通信必须有一些约定，这些约定即被称为通信协议，如对速率、传输代码、代码结构、传输控制步骤、出错控制等的约定。也就是说，为了在两个节点之间成功地进行通信，两个节点之间必须约定使用共同的“语言”。这些被通信各方共同遵循的约定、语言、规则，有时又被称为协议(protocol)。

在 Internet 中，最为通用的网络协议就是 TCP/IP 协议，TCP/IP 协议起源于上个世纪 60 年代末美国政府资助的一个分组交换网络研究项目，到现在已发展成为计算机之间最常应用的组网形式。它是一个真正的开放系统，因为协议的定义及其多种实现免费就可以获得。它成为被称作“全球互联网”或“因特网(Internet)”的基础，该广域网(WAN)已包含超过数以亿计遍布世界各地的计算机，正是诸如 TCP/IP 这样的联网协议支撑着 Internet 发展到今天。

第一套完整的 TCP/IP 代码起源于 1983 年的 4.1 BSD 系统(Berkeley 软件发行)和“BSD 联网版本”。这个源代码是很多其他实现的起点，不论是 Unix 或 Windows 操作系统。尽管最初的源代码由 U.C.Berkeley 发行并被称为“伯克利软件发行”，但 TCP/IP 代码确实是各种研究者的工作的融合，包括加州大学伯克利分校和其他地域的研究人员。

这一节我们将对网络模型和 TCP/IP 协议进行简单的概述，其目的是为本书其余章节提供充分的背景知识。

4.1.2 网络分层模型

TCP/IP 模型

TCP/IP(通常它是指传输控制协议/网际协议，Transmission Control Protocol/Internet Protocol)是发展至今最成功的通信协议，它被用于当今所构筑的最大的开放式网络系统 Internet 之上就是其成功的证明。

TCP/IP 分不同层次进行开发，每一层分别负责不同的通信功能。一个协议族，比如 TCP/IP，是一组不同层次上的多个协议的组合。TCP/IP 通常被认为是一个四层协议系统，如图 4-1 所示。每一层负责不同的功能：

链路层，有时也称作数据链路层或网络接口层，通常包括操作系统中的设备驱动程序和计算机中对应的网络接口卡。它们一起处理与电缆(或其他任何传输媒介)的物理接口细节。

- ✓ 网络层，有时也称作互联网层，处理分组在网络中的活动，例如分组的选路。在 TCP/IP 协议族中，网络层协议包括 IP 协议(网际协议)，ICMP 协议(Internet 互联网控制报文协议)，以及 IGMP 协议(Internet 组管理协议)。
- ✓ 传输层主要为两台主机上的应用程序提供端到端的通信。在 TCP/IP 协议族中，有两个互不相同的传输协议：TCP(传输控制协议)和 UDP(用户数据报协议)。TCP 为两台主机提供高可靠性的数据通信。它所做的工作包括把应用程序交给它的数据分成合适的小块交给下面的网络层，确认接收到的分组，设置发送最后确认分组的超时时钟等。由于运输层提供了高可靠性的端到端的通信，因此应用层可以忽略所有这些细节。而另一方面，UDP 则为应用层提供一种非常简单的服务。它只是把称作数据报的分组从一台主机发送到另一台主机，但并不保证该数据报能到达另一端。任何必需的可靠性必须由应用层来提供。这两种运输层协议分别在不同的应用程序中有不同的用途，我们将在后面的章节中看到这些区别。
- ✓ 应用层负责处理特定的应用程序细节。几乎各种不同的 TCP/IP 实现都会提供下面这些通用的应用程序：

  Telnet 远程登录。

  FTP 文件传输协议。

  SMTP 简单邮件传送协议。

  SNMP 简单网络管理协议。

另外还有许多其他应用，在第 6 章中将介绍其中的一部分。

OSI 模型

1978 年，国际标准化组织(ISO)开发了开放式系统互联(OSI)参考模型，以促进计算机系统的开放互联。开放式互联就是可在多个厂家的环境中支持互联。该模型为计算机间开放式通信所需要定义的功能层次建立了全球标准。OSI 模型将通信会话需要的各种进程划分成 7 个相对独立的功能层次，这些层次的组织是：

OSI 参考模型的 7 层为(从低到高)：

物理层：最底层，它是网络硬件设备之间的接口；

数据链路层：在网络实体之间建立、维持和释放数据链路连接，以及传输数据链路服务数据单元；

网络层：通过网络连接交换网络服务数据单元：

传输层：在系统之间提供可靠的、透明的数据传送，提供端到端的错误恢复和流控制；

会话层：提供两个进程间的连接管理功能；

表示层：处理被传输数据的表示问题，完成数据转换、格式化和文本压缩；

应用层：是直接面对用户的一层，提供 OSI 用户服务；

OSI 模型的 1～3 层提供了网络访问，4～7 层用于支持端到端通信。与 OSI 参考模型相比，TCP/IP 模型更侧重于互联设备间的数据传送，而不是严格的功能层次划分。它通过解释功能层次分布的重要性来做到这一点，但它仍为设计者具体实现协议留下很大的余地。因

此，OSI 参考模型在解释互联网络通信机制上比较适合，但 TCP/IP 成为了互联网络协议的市场标准。

TCP/IP 参考模型比 OSI 模型更灵活，

图 4-1 对此有所描述。

图 4-1 TCP/IP 四层模型和 OSI 参考模型之间的对应关系

4.1.3 数据的封装和拆封

位于 TCP/IP 四层模型各个层的数据通常用一个公共的机制来封装：定义描述元信息和数据报的部分真实信息的报头的协议，这些元信息可以是数据源、目的地和其他的附加属性。来自于高层的协议封装在较低层的数据报中，当信息在不同的层之间传递时，都会在每一层被封装上协议的特有头部。而当我们收到数据时会一层层的剥离头部，直至取出数据。

图 4-2 以一个 HTTP 协议发送的数据包为例描述头部封装，该实例中，最底层基于以太网传输，上层运行基于 HTTP 协议的应用。

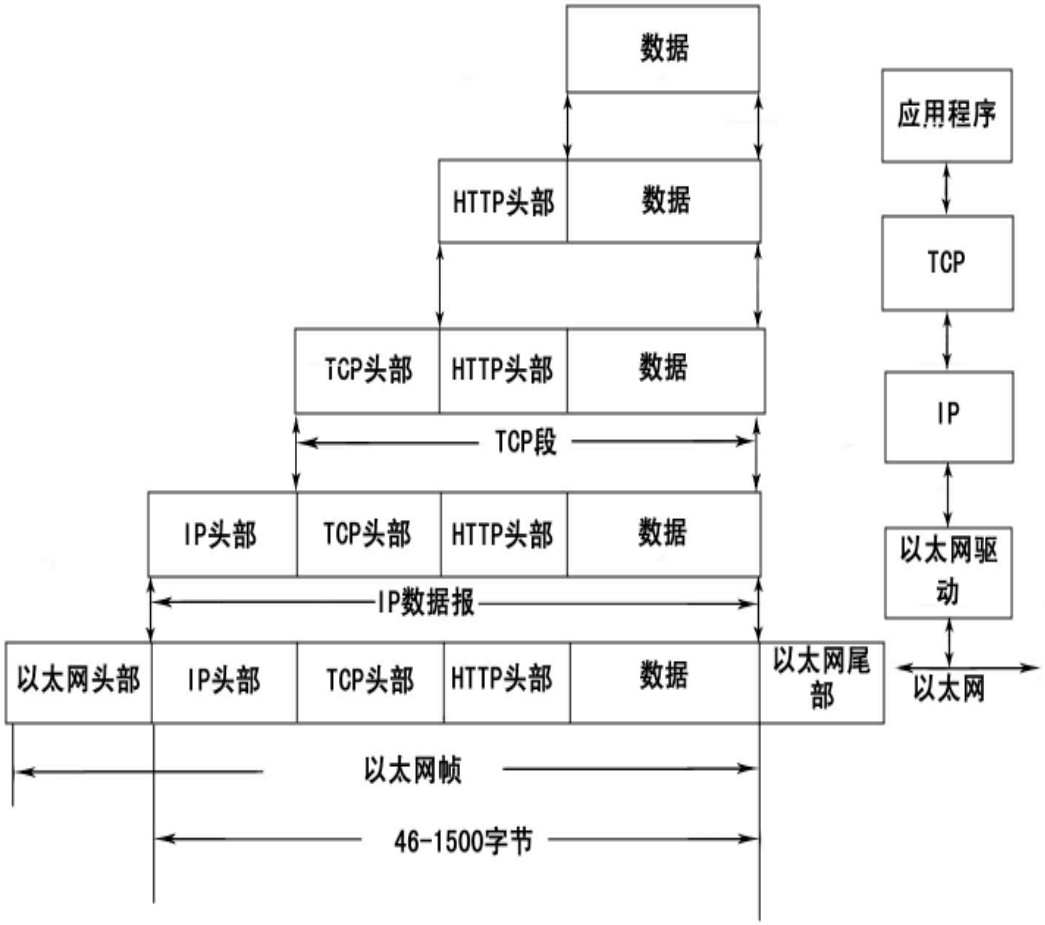


图 4-2 分层头部封装

4.1.4 IP 协议

IP 协议是最为通用的网络层协议。所有的 TCP、UDP 数据都以 IP 数据报文的格式传输。IP 协议是一个无连接、不可靠的协议。不可靠(unreliable)的意思是它不能保证 I P 数据报能成功地到达目的地。IP 仅提供最好的传输服务，如果发生某种错误时，如途中某个路由器故障，IP 有一个简单的错误处理算法：丢弃该数据报，然后发送 ICMP 消息报给信源端。任何可靠性要求必须由传输层来提供(如 TCP)。无连接(connectionless)这个术语的意思是 IP 并不维护任何关于后续数据报的状态信息。每个数据报的处理是相互独立的。这也说明，IP 数据报可以不按发送顺序接收。如果一“信源”向相同的“信宿”发送两个连续的数据报文(先是 A，然后是 B)，每个数据报都是独立地进行路由选择，可能选择不同的路线，因此 B 可能在 A 到达之前先到达。关于 IP 的正式规范文件，可参考 RFC791。

4.1.5 TCP 协议

TCP 提供一种面向连接的、可靠的字节流服务，它位于 TCP/IP 模型的传输层。面向连接意味着两个使用 TCP 的应用(通常是一个客户和一个服务器)在彼此交换数据之前必须先建立一个 TCP 连接。这一过程与打电话很相似。

TCP 通过下列方式来提供可靠性：

应用数据被分割成 TCP 认为最适合发送的数据块。由 TCP 传递给 IP 层的信息单位称为报文段或段(segment)。当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

当 TCP 收到发自 TCP 连接另一端的数据，它将发送一个确认(这个确认不被立即发送，通常将推迟几分之一秒，尽可能的和数据一起发送)。

TCP 通过检验和的形式，提供对 TCP 首部和 TCP 数据的基本校验功能。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到端计算出的检验和，和原始的相比有差错，TCP 将丢弃这个报文段，并不确认收到此报文段(发送端将超时并重发)。

既然 TCP 报文段作为 IP 数据报来传输，而 IP 数据报的到达可能会失序，因此 TCP 报文段的到达也可能会失序。如果有必要，TCP 将对收到的数据进行重新排序，将收到的数据以正确的顺序交给上层。

既然 IP 数据报会发生重复，TCP 的接收端必须丢弃重复的数据。

TCP 还能提供流量控制。TCP 连接的每一方都有固定大小的缓冲空间。TCP 的接收端只允许另一端发送接收端缓冲区所能接纳的数据。这将防止较快主机致使较慢主机的缓冲区溢出。

综上所述，TCP 是一个较为可靠的数据传输协议。但是 TCP 确认的数据不能保证被应用层收到。比如，当 TCP 确认后的数据已放入套接字缓冲区，而此时恰巧应用进程非正常退出，所以编写一个好的网络程序，我们需要注意的细节很多。

4.1.6 IP 地址

我们知道，在 Internet 中，有成千上万的计算机进行通信，那么，计算机和计算机之间进行通信的时候如何找到对方呢？

在网络中，计算机的唯一性是通过 IP 地址来标识的。也就是说，为了实现 Internet 上不同计算机之间的通信，每台计算机都必须有一个不与其他计算机重复的地址，即 IP 地址。

连接到 lnternet 上的每一台主机都有一个或多个 IP 地址，它是在 Internet 的所有计算机中唯一标识该计算机的一个 32 位的无符号整数，这个整数可表示 4294967296 个地址。这样巨大的，看起来没有什么特征的数字是很难记忆的，因此为了反映这些整数作为地址的特征以便于记忆，人们使用一种点分十进制的表示法来书写 IP 地址。

标准点分十进制表示法将 32 位的地址按 8 位一组分成 4 组，每一个 8 位的数均用十进制的数表示，所产生的 4 组数字之间用“.”连接在一起。例如，我们使用的这台主机的 IP 地址是 192.168.2.1，它对应的无符号整数是 0xc0a80201
。图 4-3 说明了这种表示方法和 IP 地址的对照关系：

11000000 (0xc0)	10101000 (0xa8)	00000010 (0x02)	00000001 (0x01)
192 .	168 .	2 .	1

图 4-3 点分十进制 IP 地址表示方法

IP 地址的标准点分十进制表示法一般记为“A.B.C.D”，4 个字母分别代表 4 个字节中的十进制整数的字符串表示。这样表示的地址除了方便记忆外，更重要的是它反映了网络的层次结构。实际上，主机的 IP 地址是根据它所在的 Inter
net 网络位置指定的。这四个字节数据可以分成 3 部分：netID，subnetID，hostID (网络地址，子网地址，主机地址)，网络地址可以由前 1-3 个字节表示，剩余的字节则为子网地址，netID 在网络信息中心(NIC)注册，并分为 A，B，C 三类。各个主机的主机地址在具体的网络管理单位注册。

A 类网络具有一字节的网络地址(最高字节)，其范围为 0～127，因此只有少量的 A 类网络，但它们中的每一个网络都支持巨大的主机数量(允许 24 位作为主机地址)。中等规模的 B 类网络具有两个字节的网络地址，其中第一个字节的范围为 192-255。因此， Internet 地址的第 1，2 或 3 字节指明网络地址，Internet 地址的其余字节指明那个网络内的地址。例如：与 lnternet 相连的某公司的网络号为 15(即，地址的第一个 8 位为二进制的 00001111)，这是一个 A 类网络。该公司将这个私有网络进一步划分为若干的子网，公司中的每一个子网指定一个子网地址，每个子网地址均由 IP 地址的前一个 8 位所标识。例如该公司的某个子网具有地址 15.255.152，属于这个子网的每一台主机的 IP 地址均以 15.255.152 开头。A 类网络 127 保留作为本地环回地址(loopback)，我们总是可用 Inter
net 地址 127.0.0.1 访问主机自身。

4.1.7 服务和端口号

Internet 通信域中套接字地址由主机的 IP 地址加上端口号组成，IP 地址用来标识 Internet 中唯一的主机，端口号则用来区别同一台主机中不同的服务程序。同一台计算机中每一个网络服务程序使用不同端口号，因为可以有很多个服务程序，因此我们需要告诉计算机数据应传送给哪个服务程序。

端口号是一个 16 位无符号整数，每一台计算机可以有 65535 个端口号(端口号 0 被保留)。Internet 中大部分计算机中相同的服务程序均使用相同的端口号，这些端口号是“知名的”，例如 rlogin 使用端口号 513，ftp 文件传输服务程序使用端口号 21，SMPT 邮件服务程序使用端口号 25，而端口号 23 则专门用于 tel

net。这些著名的端口号的使用使得客户程序能够方便的找到任意目标计算机上的适当服务程序。

Linux 中，小于 1024 的端口号保留用于标准的服务程序，它们也称之为特权端口号，因为只有 root 用户执行的服务程序才能使用它们。特权端口号的这种特征可以防止普通用户用任意的服务程序从知名端口号接收数据获取他人重要的信息。

Linux 系统有一个记录“知名”服务的配置文件，这个配置文件通常命名为 /etc/services，它里面记录了 IANA（Internet Assigned Numbers Authority 因特网号码指派管理局）“知名端口号”与服务的对应关系。

表 4-1　常见服务和对应端口号

端口(Port)	服务(Service)
7	Echo
21	FTP
23	Telnet
25	SMTP
79	Finger
80	HTTP

4.1.8 域名

在 Internet 中，对于主机的 IP 地址，除了用点分十进制表示之外，也可以使用域名表示。域名的优点是它更便于记忆。例如，Internet 地址：59.151.21
.100 也可以记为 www.google.cn。

主机名采用层次结构的方法来命名，它由“.”分隔的多个域组成(例如 www.google.cn)。主机名的每一个域按照范围从小到大，从左至右的顺序用“.”连接起来。这些域名中位于最前面的是最具体的，通常第一个域指明具体的计算机，越靠后的域则越不具体，它通常表示一类网络的集合。Internet 中使用应用范围或地域来划分域名最右边的域，例如：edu 代表教育，com 代表商业机构，org 代表非营利性组织，gov 代表政府机构。世界上的其他国家都使用两个 ascii 字符的国家代码作为最右端的域，例如 cn 代表中国，jp 代表日本。

每一台主机可以有多个主机名，它们实际上是同一个主机名的别名。别名有时是很有用的，例如，当一台计算机已有的主机名为 foo.bar.baz，如果同时在这台机器中还提供 ftp 和 Web 服务，那么可能会为这台主机创建另外两个别名 ftp.bar.tom 和 www.bar.com，它们都指向 foo.bar.com。这样做有助于人们记住服务程序所运行的这台计算机的名字。

大部分著名的服务程序既接受命令行或配置文件中给出的主机名，也接收点分十进制形式的 IP 地址。不过在打开连接时，主机名最终要转换为它所代表的大端字节序 IP 地址。

Linux 系统内部都用一个数据库来记住主机名与主机 IP 地址之间的映射，这一数据库要么是/etc/hosts 文件，要么由 DNS 服务系统提供。在互联网中，域名系统(DNS)是由很多层分布在互联网中的域名数据库组成，由它来提供 IP 地址和主机名之间的映射信息，进行域名到 IP 地址之间的转换。

4.2 Socket 编程

4.2.1 套接字(Socket)和 Socket API 简介

套接字是一种通讯机制，是一种使用标准 Unix/Linux 文件描述字与其他主机进程通讯的手段。从程序员的角度来看，套接字很像文件描述字(一个整数)，因为它同文件和管道一样可以使用 read()/write()来读写数据。但是，套接字与普通文件描述字不同，首先，套接字不像文件描述字那么简单，它除了需要地址端口等信息之外，还明确包含有关通信的其他属性(如地址族，协议类型，协议族)。其次，套接字的使用可以是非对称的，它通常明确的区分通信的两个进程为客户进程和服务进程，并且允许不同系统或主机上的多个客户与单个服务进程相连。最后，套接字的创建以及控制套接字的各种操作与文件描述字也有所不同，这些操作的不同是由于建立套接字网络连接比磁盘访问要复杂而带来的。

Socket API(套接字接口)包含完整的调用接口和数据结构的定义，它为应用程序提供对套接字进行操作的手段。进程可以通过调用 Socket API 对套接字进行操作从而实现访问网络并使用各式各样的网络协议进行通讯。

Socket API 最初作为 TCP/IP 协议代码的。部分被 U.C.Berkeley 发布，所以该接口也被称为 Berkeley Socket。Socket API 在其他操作系统中得到广泛的支持，这包括众多的“类 UNIX”操作系统(AIX，solaris，sunOS，FreeBSD，NetBSD，OpenBSD，BSDI，unixware， SCO unix，hp-unix，MAC，Linux 等等)和 Windows 系统。Socket 层的实现在内核之中，如果应用程序想要调用它就要通过系统调用方式。图 4-4 描述了 Socket 层所处的结构位置。

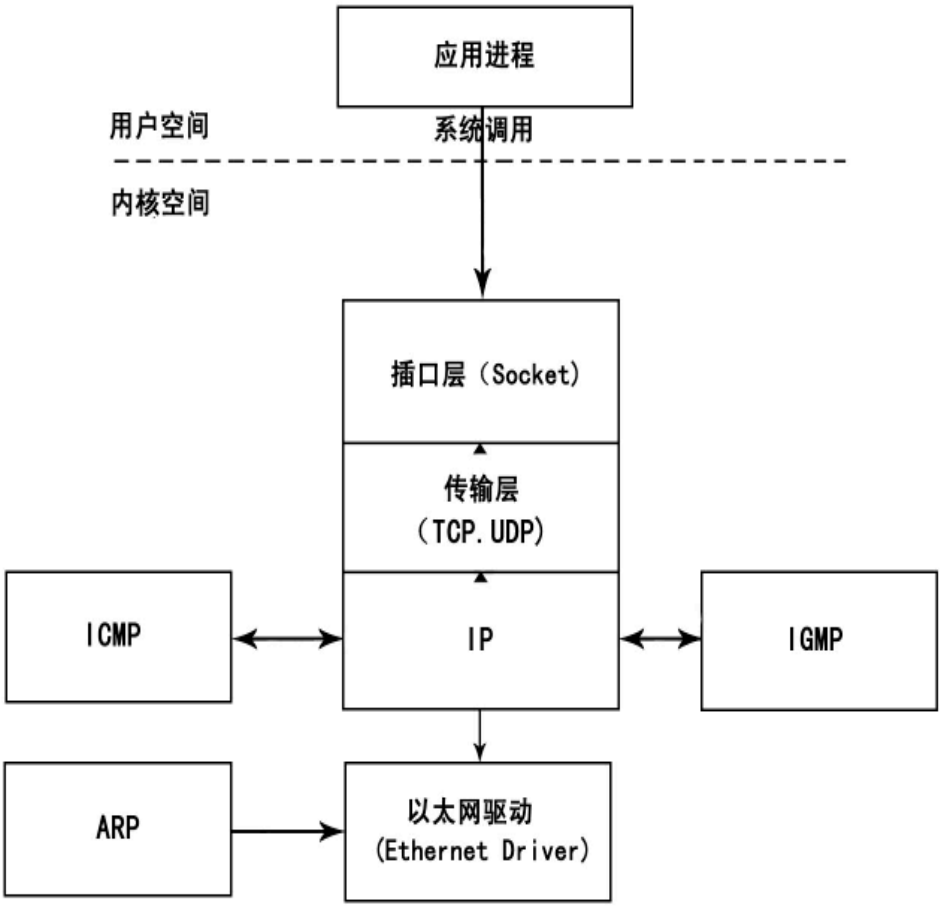


图 4-4 插口层描述

4.2.2 网络编程基础知识

在开始学习如何使用套接字编程之前，我们需要先学习一些重要的相关知识和概念，这其中包括服务器客户端模型，字节序，地址族，地址结构，域名解析等相关内容。

服务器和客户端模型

客户端和服务器端之间使用 TCP 协议通信，这种情形类似于电话通信，相互通信的两个进程之间需显式的调用套接字连接函数建立连接，而且客户和服务进程的角色进一步被使用和建立套接字的方法所增强。这种模式被称为面向连接的客户/服务模型。流套接字(如使用 TCP 的套接字)通讯属于这种模式。

我们先来看 TCP 套接字通信的典型过程，如图 4-5。

服务进程通过系统调用 socket()创建一个套接字，随后，服务进程要给该套接字捆绑一个众所周知的端口，以便其他程序能够向服务进程请求并与之通讯，这个过程通过给套接字调用 bind()来实现。在套接字捆绑端口号之后，服务进程便等待客户进程与该套接字连接。因为服务器必须允许多个客户同时与该套接字连接，所以 listen()函数用于为客户端连接创建一个连接列队。服务进程通过调用 accept()接受这些连接。

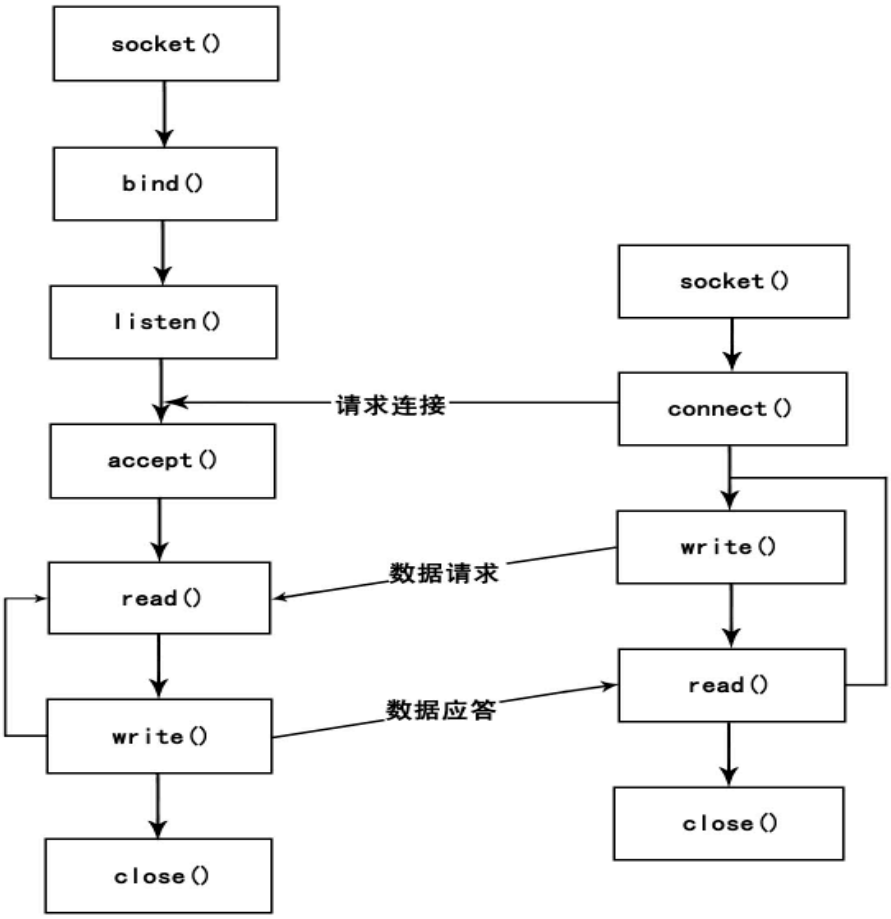


图 4-5 客户端和服务器端编程模型

服务进程每调用一次 accept()便创建一个新的套接字，它不同于前面由 socket()创建的套接字，这个新的套接字完全只用于与特定的客户通信，而 socket()创建的套接字则保留用于与等待其他客户连接的到来。利用这一特征，我们可以使服务进程服务于多个客户。对于简单的服务，后继的客户则要等待在 listen()列队中直到服务进程就绪后再次调用 accept()为止。

客户端的动作比服务器端要直观一些。客户进程通过调用 socket()创建一个套接字，然后通过将服务进程套接字的捆绑的地址和端口作为参数，调用 connect()与服务进程建立连接。一旦连接建立，客户和服务进程两端就可以像对普通文件描述字一样的使用套接字进行通信。客户端通常对它所使用的端口号和地址并不关心，只需保证该端口号在本机上是唯一的就可以了，当客户端调用 connect()时，系统将为客户端套接字分配一个空闲的端口号，所以客户端口号也称作临时端口号。

TCP 的连接建立和关闭过程

为了帮助大家理解 TCP 连接的概念，我们必须了解如何建立和关闭 TCP 连接以及 TCP 的状态变换。图 4-6 展示了 TCP 的连接和关闭过程。

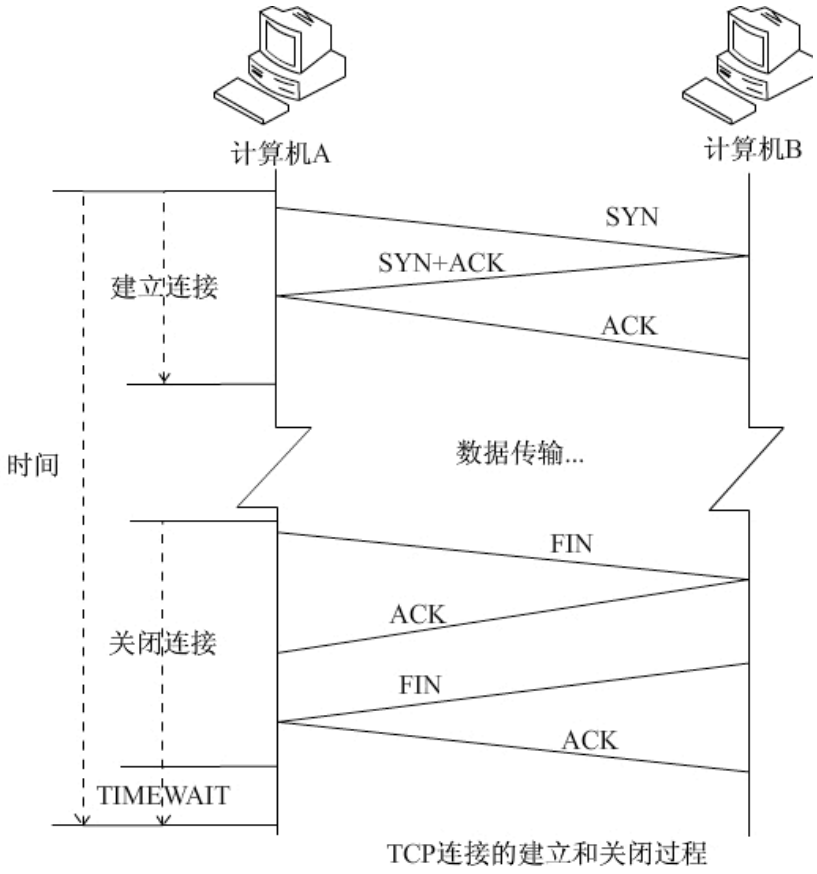


图 4-6 TCP 连接的建立和关闭过程

TCP 使用 SYN、FIN、RST 等控制报文来转换连接状态，ACK 是 TCP 对数据和控制报文进行确认的报文，它有可能单独存在，但绝大多数时候是随着其他报文一同发送(在数据或控制报文中进行确认)，所以并不认为 ACK 为控制报文。

TCP 不同的控制报文的作用如下：

- ✓ SYN 报文：建立连接的请求。
- ✓ FIN 报文：用来关闭连接的请求。
- ✓ RST 报文：用来复位一条连接，使之成为未连接状态。
- ✓ ACK 报文：用来确认数据，不过 ACK 很少作为单独的报文发送。

TCP 建立连接的过程

TCP 建立一次连接需要经历三次握手过程：

- ✓ 服务器端做好监听准备，通常是服务器端通过调用 socket()，bind()和 listen()函数监听服务器的某个已知端口。
- ✓ 客户端应用通过设置服务器端的 IP 地址和端口号并调用 socket()和 connect()函数，导致客户端的 TCP 层发送一个 SYN 报文以请求连接，以及初始序号(例如 I)。
- ✓ 服务器端 TCP 层收到客户端的 SYN 报文后，发送包含服务器的初始序号的 SYN 报文段作为应答(初始序列号例如 J)，同时，该 SYN 报文将头部的确认序号设置为 I+1(客户端的序号加 1)对客户的 SYN 报文段进行确认。

- ✓ 客户端 TCP 层在收到服务器端的 SYN 后对服务器进行确认，发送的 ACK 的确认序号 J+1(服务器端 SYN 的序号加 1)，此时连接建立的过程完成。
- ✓ 三次握手时我们提到 TCP 的序列号，TCP 协议对传输中的每个字节都单独编号（不包含 TCP 头部以及头部选项)，连接时 ACK 确认序号为 SYN 的序号加 1，所以可以看出 SYN 只占用一个字节的序列号空间。

TCP 连接关闭的过程

建立一个连接需要一次握手，而终止一个连接要经过四次握手。这是由 TCP 支持半关闭状态造成的，因为 TCP 连接是全双工的，所以每个方向须单独进行关闭。

我们假设网络上有 A 和 B 两台主机，他们已经通过一次握手建立了连接。

- ✓ 当 A 主机完成它的数据发送任务后就能发送一个 FIN 来终止这个方向连接，序号为 M，发送 FIN 通常是应用程序通过调用 close()或 shutdown()导致的。
- ✓ B 主机收到 FIN 后将响应一个 ACK，ACK 的确认序号为收到的 FIN 的序号 M 加 1，同时它必须通知应用程序对方已经终止了那个方向的数据传送,通常是对 I/O 操作函数返回文件结束标志(应用程序可以通过调用 read()、write()、select()、poll()等系统调用察觉这个文件结束标志)。
- ✓ 如果 B 主机的应用程序在察觉到这个变化时也调用 close()或 shutdown()，这会导致 B 主机发送一个 FIN 给 A 主机，序号为(N)。
- ✓ 主机 A 收到 FIN 后也将响应一个 ACK，序号为 N 加 1。
- ✓ 此时关闭的过程结束,TCP 将进入 TIME WAIT 状态(详见第 6 章套接字选项一节关于 TIME WAIT 的描述)。

字节序

不同种类的计算机在存储“字”数据时，使用不同的字节序协定，一些计算机把最重要字节(Byte)放置在字(Word)的最前面，称之为大端(big-endian)字节序，另外一些体系的计算机把最重要字节(Byte)放在最后，称之为小端(little-endian)字节序。这里所说的最重要字节(Byte)是相对计算机所表示的一个数字而言，指的是用来存放数字“最重要位(MSB)”的字节，如对于整数而言，最重要的位是最高位，而对于浮点数而言，是存放阶码最高位的字节。

我们常见的 x86 构架就采用小端字节序，而有些 RISC 处理器会使用大端字节序如 Sparc，作为嵌入式产品常见的 ARM 处理器在启动时可设定小端字节序或大端字节序。

在采用大端字节序的计算机中，一个字节的顺序由低地址至高地址的地址编号，字节顺序与地址顺序是一致的，而采用小端字节序的计算机系统中，一个字的字节顺序则由高地址至低地址的地址编号，字节顺序与地址顺序相反。图 4-7 说明了这两种字节序的主机对存储在地址 1000 处的整数在存储方式上的不同：

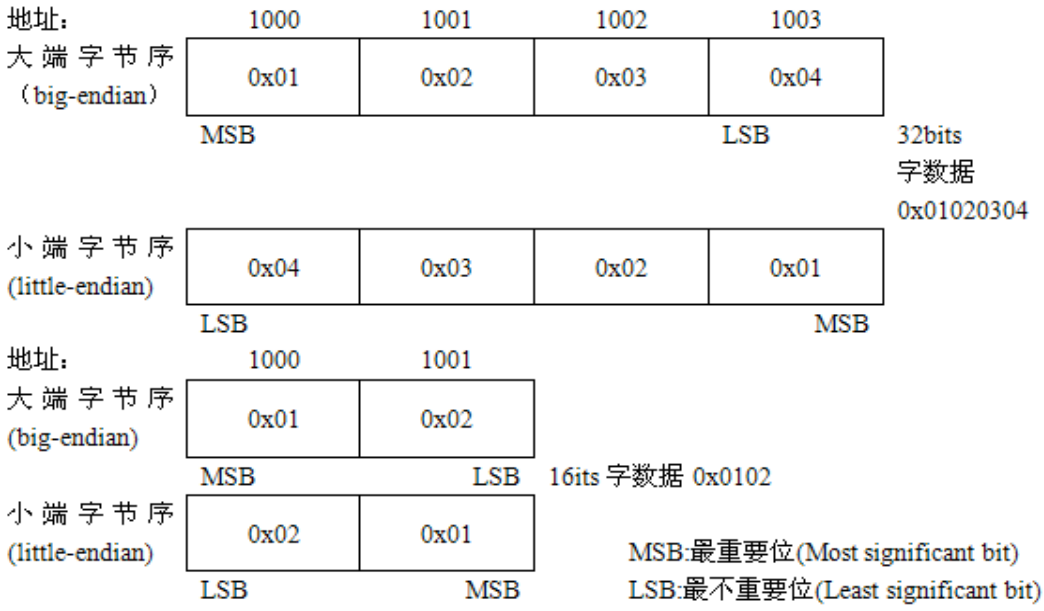


图 4-7 不同字节序的主机保存在内存地址 1000 处整数的存储方式

上图演示了存放在内存地址 1000 处的 32 位整数(0x01020304)和 16 位整数(0x0102)在大端字节序体系和小端字节序体系中的不同。假设存储器的地址 1000-1003 中分别存储的数字为 0x01，0x02，0x03，0x04，在大端字节序的系统中，它表示十六进制的 0x01020304。而在小端字节序的系统中，它表示 0x04030201。

互联网是个复杂的结合体，由各式各样的主机构成，不同体系结构的主机对整数的存储方式是不同的，如果它们各自使用不同的字节序在网络中通讯，将导致严重的问题，为了使得具有不同字节序的机器之间能够通讯，Internet 协议为网络上传输的数据规定了一种规范的字节顺序约定，称之为网络字节序，网络字节序采用大端字节序。

让我们来熟悉两个概念：

1.网络字节序：网络字节序等同于大端字节序(big-endian)。

2.主机字节序：主机体系结构特有的字节序。

示例 4-1 是一个判断主机字节序的小程序，这个程序中，如果是大端字节序则显示 BigEndian，否则显示 Little Endian。

示例 4-1：判断字节序

源代码：byte_order.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
static int isBigEndian()
{
```

```
  uint32_t thisx = 0x01020304;
  uint8_t *thisp = (uint8_t *)&thisx;
  return (*thisp == 1)  ? 1 : 0;
}

int main(int argc, char **argv)
{
   printf("Byte order: %s Endian\n", isBigEndian() ?"Big" :"Little");
   return 0;
}
```

示例 4-1 中，我们对一个 32 位整数赋予值 0X01020304，然后我们使用一个 8 位的指针指向这个整数在内存中的首地址，通过判断旨地址是 0x01 还是 0x04 就能判断主机字节序。

多数情况下，我们不需要判断主机字节序，而直接使用 C 库中的字节序转换函数：

```
#include <sys/socket.h>
#include<arpa/inet.h>

unit32_t htonl(unit32_t hostlong);
unit16_t htons(unit_t hostshort);
unit32_t ntohl(unit32_t netlong);
unit16_t ntohs(unit16_t netshort);
```

其中：

- ✓ htonl 将一个 32 位整数由主机字节序转换成网络字节序。
- ✓ htons 将一个 16 位整数由主机字节序转换成网络字节序。
- ✓ ntohl 将一个 32 位整数由网络字节序转换成主机字节序。
- ✓ ntohs 将一个 16 侮整数由网络字节序转换成主机字节序。

当使用这些函数时，我们并不关心主机字节序的实际值，只是正确的调用适当的函数在主机和网络字节序之间转换数值，在那些字节序与网络字节序相同的系统中，这几个函数实际上是空的宏定义。

4.2.3 地址族

一般而言，在使用套接字通信之前，我们首先要为套接字选定一个地址族，简单的说，为套接字指定地址族的目的是告诉套接字使用哪一种网络层进行通讯（如 IPv4 或 IPv6）。在 Linux 系统中，sys/socket.h 头文件定义了繁多的地址族类型，它们都是“AF_”或“PF_”开头的数字常量宏定义，它们有些是某些厂商专有的，有些则非常通用。例如 AF_INET 适用于地址族（IPv4），AF_INTE6 适用于 IPv6。

4.2.4 地址结构

在开始编写基于套接字的网络程序之前需要先学习如何填充地址结构。大多数SocketAPI 函数都需要一个指向地址结构的指针作为参数。每个地址族都定义了它自己的套接字地址结构。这些结构的名字以“sockaddr_”开头，并以对应每个协议族的唯一后缀结束(如 sockaddr_in 表示 AF_ INET 地址族结构)。

在 sys/socket.h 文件中有各种各样的地址结构定义，分别针对不同的地址族，可是 Socket API 只有少量且固定的几个接口定义，如何才能传递多种地址族结构给 Socket API 函数呢？Berkeley Socket 专门为此声明了一个通用地址结构 sockaddr：

```
typedef unsigned short sa_family_t;
#include<sys/socket.h>
struct sockaddr {
    sa_family_t sa_family;  /*地址族*/
    char    sa _data[14];     /*地址值，实际可能更长*/
    };
    #define SOCK_MAXADDRLEN 255     /*可能的最长的地址长度*/
```

sockaddr 结构的 sa_data 域的定义并不明确。它只是被定义为 14 字节的数组，SOCK_MAXADDRLEN 宏暗示内容可能超过 14 字节。这种不确定性是经过深思熟虑的。Socket API 是个非常强大的接口，它能处理多种多样网络层地址族，这包括我们常常提到的 IPv4、IPv6、IPX。

SocketAPI 为了能够传递繁多的地址族结构，它使用通用地址结构类型 sockaddr 作为地址参数类型，在该类型的前面包含了所有地址结构共有的 16
bits 域来存放地址族类型(它是主机字节序的，用来表明该结构的地址族类型，如 AF_INET，AF_UNIX 等等)，Socket API 会根据这个域判断如何对待结构的后续数据。这也就是说，我们不会真的用到 sockaddr 结构，我们需要填写真正的与地址族相关的地址结构，然后在传递给需要地址结构的函数时，把指向该结构的指针转换成 sockaddr 结构类型的指针传递进去。例如，当我们要使用 IPv4 地址族，此时需要填写的是 sockaddr_in 结构类型，我们把 sockaddr_in 的 sin_family 域设定为 AF_INET,然后把 sockaddr_in 结构的地址传递给需要 sockaddr 结构指针参数的函数。

IPv4 地址族结构

IPv4 是迄今为止最通用的网络层协议，所以它对应的地址结构被称为“网际套接字地址结构”，以 sockaddr_in 命名，它定义在头文件<netinet/in.h>中：

```
//IPv4 地址族结构定义
#include<netinet/in.h>
struct in_addr_t s_addr{
    in_addr_t s_addr;
```

```
};
struct sockaddr_in {
    unit16_t sin_family;
    unit16_t sin_port;
    struct   in_addr sin_addr;
    char     sin_zero[8];
};
```

IPv4 地址族结构的三个重要的域是：

- ✓ sin_family 是一个 16 位的无符号整数，它需要填写主机字节序的地址结构类型(如 AF_INET)。
- ✓ sin_port 为 16 位无符号整数值，这个域用来存放网络字节序的端口号。
- ✓ sin_addr 用来存放 IPv4 地址，地址值为网络字节序。sin_addr 由于历史原因被声明为 in_addr 结构类型，但它实际上只包含一个名为 s_addr 的 32 位无符号整数值，这个类型定义在 netinet/in.h 之中。

图 4-8 描述了未填充前的 sockaddr_in 结构。

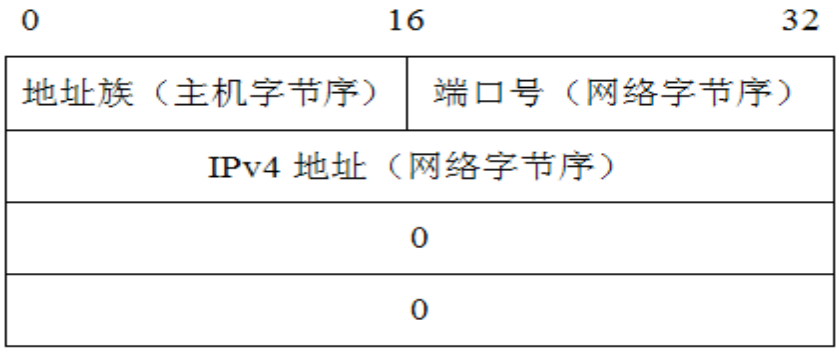


图 4-8 IPv4 地址族结构(sockaddr_in)描述

假设远程的主机地址是 192.168.2.1，端口号为 3001，现在我们尝试填充它：

```
struct sockaddr_in sin;
memset(&sin, 0, sizeof(sin));
sin.sin_port = hons(3001);
sin.sin_addr.s_addr = htonl(0x0a80201);  /*192.168.2.1*/
```

图 4-9 给出了填充后的地址结构：

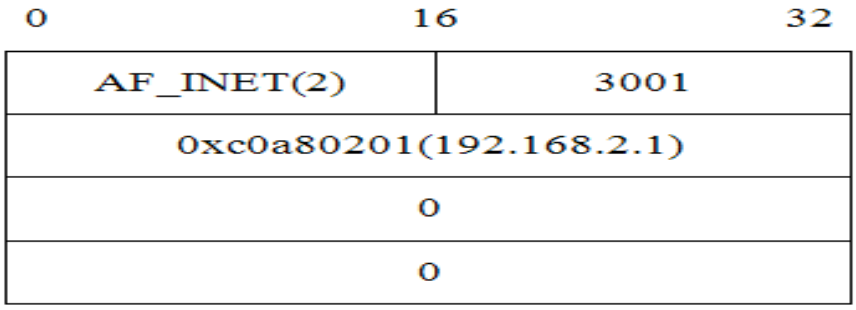


图 4-9 填充后的 sockaddr_in 地址族结构

sin_family 需要填充正确的地址族类型 AF_INET。

0xc0a80201 是地址 192.168.2.1 的 IPv4 地址值，我们需要把它转换成网络字节序，但这么做很麻烦，我们需要自己计算 32 位的地址值。幸好 Socket API 已经定义了转换字符串至 IP 地址值的函数，函数原型如下：

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
int inet_aton(const char *strptr,struct in_addr *addrptr);
in_addr_t inet_addr(const char *strptr);
char *inet_ntoa(struct in_addr in);
const char *inet_ntop(int af, const void *restrict src,char *restrict dst, socklen_t size);
int inet_pton(int af, const char*restrict src, void *restrict dst);
```

第一个函数 inet_aton()将 strptr 指向的字符串转换成网络字节序的 32 位的 IPv4 地址，并通过指针 addrptr 来存储转换后的结果，如果转换成功返回 1，否则返回 0。

inet_addr()执行和 inet_aton()相同的转换，返回值是网络字节序的 IPv4 地址，这个函数存在一个严重的问题，因为 IPv4 地址是 32 位的，也就说明所有 32 位无符号整数能表示的数字都是有效的地址(0x00000000 到 0xffffffff 的 32 位整数对应地址 0.0.0.0 到 255.255.255.255)，在 inet_addr 的相关文档中提到，如果 inet addr()执行出错，它将返回 INADDR_NONE 表示错误,这个值在绝大多数操作系统中被定义为一 1(补码为 0xffffffff)，这也同时说明 inet add()不能处理 255.255.255.255(值为 0xffffffff)这个受限广播地址。

函数 inet_ntoa()将一个网络字节序的 IPv4 地址转换成字符串，并将字符串作为返回值返回。用于保存返回值的字符串驻留在静态内存中，这意味着这个函数是不可重入的，也就是说我们把它用在并发操作(线程或 vfork())函数中要非常小心。

以上三个函数或多或少都存在问题，我们并不推荐使用，下面将着重介绍两个用于同样目地的转换函数，它们不仅可以用于转换 IPv4 地址，还可以用来转换 IPv6 的地址。

inet_ntop()把网络字节序的 ip 地址 src 转换成字符串保存在 dst 中作为返回值返回，参数 size 为 dst 所包含的字节数。

inet_pton()把字符串 src 转换成 ip 地址保存在 dst 中。该函数调用成功返回大于 0 的整数。

inet_ntop()和 inet_pton()有个共有的地址族参数 af，如果我们要转换 IPv4 的地址，这里需要填写 AF_INET，现在可以将上面的例子改写一下：

```
#include <sys/types.h>
 #include <sys/types.h>
 #include <sys/socket.h>
#include <netinet/in.h>
# include <arpa/inet.h>
struct sockaddr_in sin;
char buf[16]={0};
memset(&sin,0,sizeof(sin));
sin.sin_family=AF_INET;
sin.sin_port=htons(3001);
if(inet_pton(AF_INET,“192.168.2.1”,&sin.sin_addr.s_addr)<=0)
{
  //错误处理
}
printf(“%s\n,inet_ntop(AF_INET,&sin.sin_addr.s_addr,buf,sizeof(buf))
);
```

上例中我们首先把端口号 3001 用 htons()变成网络字节序并填充地址族结构的 sin_port 域，然后调用 inet_pton()将字符串表述的 IPv4 地址转换成协议族大端字节序地址。最后，我们调用 inet_ntop()再将它转换成字符串表述的 IPv4 地址，并调用 printf 输出它，在调用 inet_ntop 的时候，我们必须给它传递一个用于保存结果的缓冲区，我们在栈中分配 16 字节的数组 buf,之所以这么做的原因是因为 16 字节足够存储一个字符串表示点分 lPv4 的地址。

域名解析

使用点分十进制字符串表述的 IP 地址仍然不够方便,尽管通过 IP 地址可以标识远程主机，但是人们最喜欢使用的还是主机域名(如 www.google.cn)，同时这样可以使我们的程序变得更加易于使用。

域名不能直接传送给任何 Socket API 函数，解析器库(Resolver library)包含了域名解析函数，它将域名转换成网络字节序的协议地址：

```
   #include<netdb.h>
   struct hostent{
```

```
    char   *h_name;
    char   **h_aliases;
    int     h_addrtype;
    int     h_length;
    char    **h_addr_list;
};
#define h_addr h_addr_list[0]
struct hostent *gethostbyname(const char *hostname);
extern int h_errno;
char *hstrerror(int err);
```

hostent 结构体成员：

- ✓ h_name 被称为主机的正式(canonical)名(例如 www.google.cn 的正式主机名是 cn.l.google.com)。
- ✓ h_aliases 是主机的别名列表，有的主机可能有几个别名(www.google.cn 就是 google 他自己的别名)。
- ✓ h_addrtype 表示的是主机地址族的类型，AF_INET 或 AF_INET6。
- ✓ h_length 表示返回地址的长度，如果返回的是 IPv4 地址，这个长度为 4。
- ✓ h_addr_list 返回主机的地址列表，以网络字节序存储。

Gethostbyname()根据配置文件(/etc/resolv.conf)里的 DNS 主机列表向 DNS 服务器发送 UDP 的查询或通过查找本地静态主机文件/etc/hosts 来实现域名解析。

参数 hostname 是一个包含主机域名的字符串(如“www.google.cn”)或包含 IPv4 地址的字符串(如“192.168.2.1”)。

当 gethostbyname()成功执行将返回非空的 hostent 结构地址，失败则返回空指针，并用错误码设置全局变量 h_errno，可以通过调用 hstrerror()并传递 h_errno 作为参数取得错误描述信息。

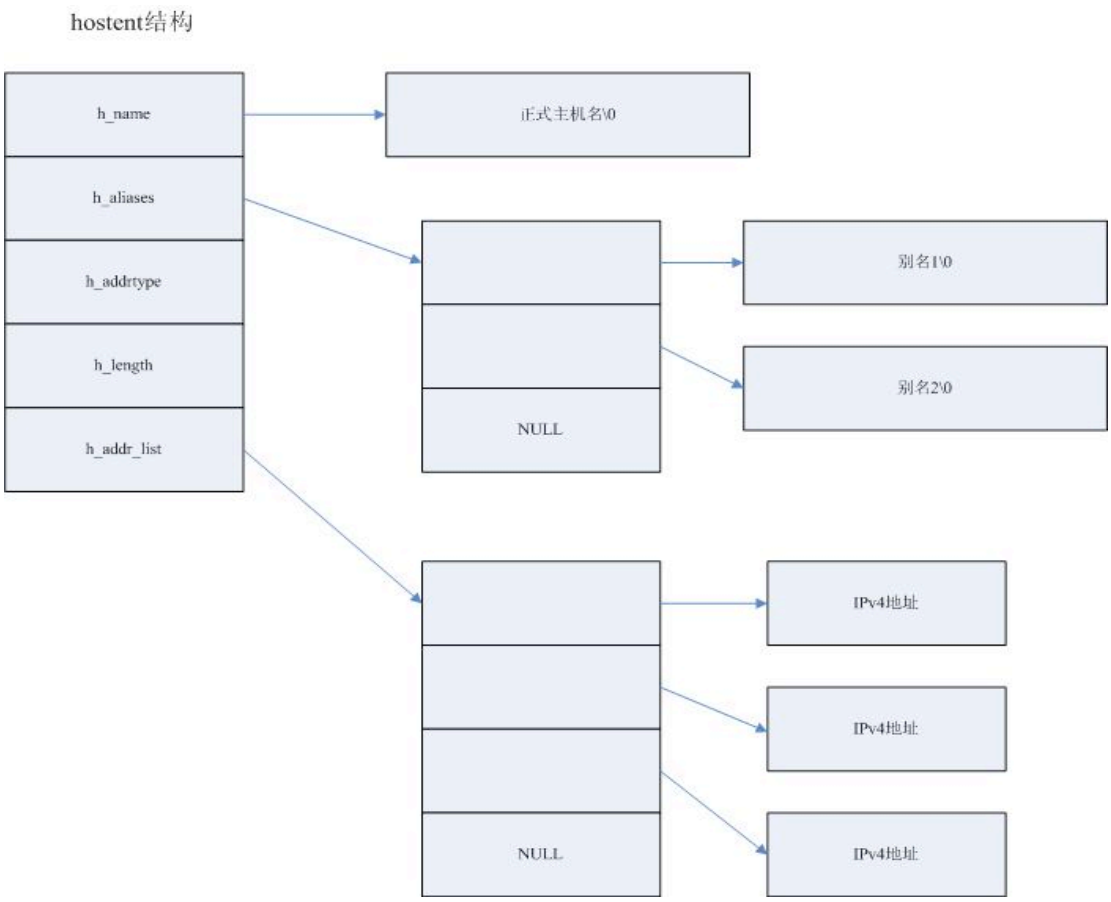


图 4-10 gethostbyname 返回的 hostent 结构

图 4.10 描述了 gethostbyname()返回的 hostent 结构，它假设被查询的域名有两个别名和三个 IPv4 地址，在这些字段中，主机名和所有的别名都是以字节 0 结束的字符串。

下面我们来看一个域名解析程序的例子。

示例 4-2：域名解析程序

源文件：domain.c

```
#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(int argc, char **argv)
{
    char **pptr;
    char  str[46];
    struct hostent *hptr;
    if(argc < 2)
    {
```

```
        fprintf(stderr, "usage: domain <domain>\n");
        return -1;
    }
    if((hptr = gethostbyname(argv[1])) == NULL)
    {
        fprintf(stderr, "gethostbyname call failed. %s\n",
    hstrerror(h_errno));
        return -1;
    }
    printf("official name: %s\n", hptr->h_name);
    for(pptr = hptr->h_aliases; *pptr != NULL; pptr ++)
    {
        printf("\t alias: %s\n", *pptr);
    }
    if(hptr->h_addrtype != AF_INET)
    {
        fprintf(stderr, "Invalid address type %d\n", hptr->h_addrtype);
        return -1;
    }
    pptr = hptr->h_addr_list;
    for(; *pptr != NULL; pptr ++)
    {
        printf("\t address: %s\n", inet_ntop(hptr->h_addrtype, *pptr,
str, sizeof(str)));
    }
    return 0;
}
```

上面的程序要求用户输入一个域名作为参数，程序通过调用 gethostbyname ()获取该域名的协议地址，并将返回的结果输出到标准输出。

程序首先输出别名地址，通过图 4-10 中可以看出 hostent 结构的 h_aliases 是一个指针数组，它以 NULL 指针结尾，每个指针指向一个 0 结尾的字符串，所以我们在循环中调用 printf()输出每一个别名地址。

h_addrtype 域指出地址协议的类型，我们不打算支持其他类型的协议地址(如 IPV6)，所以返回的协议地址不是 AF_INET(IPV4)我们就不输出它。

h_addr_list 域也是一个指针数组，它每个指针指向一个大端字节序的 IPv4 地址(32 位整数)，并以 NULL 指针结束，最后我们在循环中调用 printf()输出每个 IPv4 地址。

图 4-11 是程序的执行结果。

```
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
user@user-desktop:~/share-sb-ubuntu/chaptar_2$ ./test www.google.com
official name: www.l.google.com
        alias: www.google.com
        address: 64.233.189.104
        address: 64.233.189.99
        address: 64.233.189.147
        address: 64.233.189.103
user@user-desktop:~/share-sb-ubuntu/chaptar_2$
```

图 4-11 编译后的 domain.c 运行结果

我们在程序的参数中输入 www.google.cn，它通过查询 dns，并将结果输出，我们可以看出 www.google.com 的正式主机名为 www.l.google.com，而 www.google.cn 是它的别名(alias)，它有四个 IP 地址分别为 64.233.189.104，64.233.189.99，64.233.189.147 和 64.233.189.103。

4.2.5 建立套接字

从这一节开始我们将学习如何使用套接字编程写网络通信程序。

无论是客户端或服务器端，在使用套接字之前都要先创建套接字，这就如同打电话必须先安装电话机一样。在 Socket API 中 socket()函数用来创建一个套接字：

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

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

socket()执行成功将返回非负数的套接字，失败将返回-1。

参数 domain 告诉系统你需要使用什么地址协议族，他们都是用 AF_或 P_ 开头的数字常量宏定义，地址族的声明在 sys/socket.h 中，如 AF_INET 适用于地址族(IPv4)。

参数 type 有五个定义好的值，也在 sys/socket.h 中。这些值都以“SOCK_”开头。其中最通用的是 SOCK_STREAM，它告诉系统你需要一个可靠的流传送服务，当使用 AF_INET 和 SOCK_STREAM 作为参数调用 socket()将指明创建一个使用 TCP 协议的套接字。如果指定 SOCK_DGRAM，则将请求一个无连接报文传送服务(如 UDP)。如果你需要存取原始套接字，你就需要指定 SOCK_RAW。

protocol 参数可以指定一个更精确的协议(如 IPPROTO_TCP 对应与 TCP 协议)，它取决于前两个参数，而且并非总是有意义，多数时候可以不关心它，在大多数情况下可以传递 0，系统会根据前两个参数为我们选择一个合适的协议族。

4.2.6 连接远程主机(connect)

主动请求连接是客户端的动作，TCP/IP 客户端通常要调用 connect()去连接一个服务端，我们先看一下 connect()函数的原理：

```
#include<sys/types.h>
#include<sys/socket.h>
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

调用 connect()函数会引发一次三次握手过程(发送 SYN)。等待连接建立后（三次握手完成)或出错才返回。在连接的起始阶段套接字层首先会为我们找到一个空闲的端口号，并根据服务器地址查询路由表选择一个本地 IP 地址，在连接完成后将使用这个地址和端口与服务器通讯。其中的三个参数意义如下：

- ✓ 参数 sockfd 是成功调用 socket()返回的套接字。
- ✓ 参数 serv_addr 形式上是一个指向 sockaddr 的指针，而实际上需要传递的是特定地址族的地址结构(如 struct sockaddr_in)的地址，关于这个问题前面已经讨论过了。
- ✓ 参数 addrlen 表示第二个参数的字节数(如 sizeof(struct sockaddr_in))。

如果 connect()调用成功，表示连接已经建立(三次握于完成)，返回 0。否则返回-1 并将错误码代码存放于全局变量 errno 中。

有些可能导致 connect()失败的原因。例如，指定 IP 地址的主机可能不存在，没有到指定主机的路由，远程服务器端未启动，或者指定端口在远程服务器上并未启动监听等等，出错原因可以使用 perror()函数来输出错误信息。

4.2.7 关闭套接字

关闭套接字可以简单的调用 close()函数：

```
#include<unistd.h>
int close(int sockfd);
```

其中，参数 sockfd 可以是打开的任何类型的文件描述字，包括套接字。
close()调用成功返回 0，失败返回-1，并用出错代码设置 errno。

通常情况下(指末使用 SO_LINGER 套接字选项修改 close()的行为)对一个已连接套接字成功调用 close()。它表示套接字缓冲区中的数据发送给对方的 TCP 层并成功执行 TCP 关闭连接的 4 个过程，close()要等待关闭过程完成后才返回。

这里有一个例外：当我们对一个已打开的文件描述字或套接字调用 dup()进行复制，或进程调用 fork()时，系统会增加套接字的引用计数器。当套接字的引用计数器大于 1 时，它并不关闭连接，而是简单的递减套接字引用计数器后直接返回。

另外一个专门用来关闭套接字的函数是 shutdown()，shutdown()和 close()有些不同，当调用 shutdown()关闭写操作时(SHUT_WR)，系统将不顾套接字引用计数而直接引发 TCP 连接关闭过程。

```
#include <sys/socket.h>
int shutdown(int sockfd, int howto);
```

参数 sockfd 必须是成功调用了 socket()函数返回的任何套接字。

参数 howto 有三个取值：

- ✓ SHUT_RD 关闭连接的读一半，进程不能接收套接字缓冲区中的未读数据，且留在套接字缓冲区中的数据将作废。进程不能再对套接字调用 read()读取数据。在调用 shutdown()后，TCP 套接字在接收到数据时仍然发送 ACK 进行确认，但数据都被丢弃掉。
- ✓ SHUT_WR 关闭连接的写这一半。我们称之为 TCP 的半关闭(half-close)状态，程序不能再对该套接字进行写操作，当前套接字缓冲区中的已有数据都将被发送出去，在数据发送完毕后，将发送 TCP 的关闭序列(FIN)。前面我们提到过，这个操作不管套接字的引用计数而直接引发 TCP 关闭。
- ✓ SHUT_RDWR 关闭连接读和写，这等效于调用 shutdown()两次，第一次调用时使用 SHUT_RD 参数，第二次使用 SHUT_WR 参数。shutdown()调用成功返回 0，失败返回-1，并用出错代码设置 errno。

4.2.8 Socket I/O

对于套接字的 I/O(读写)有很多 API 可用，如 send()、recv()、readv()、writev()、sendmsg()、recvmsg()、sendto()、recvfrom()、read()、write()等等。这里我们只介绍常用的 I/O 接口 read()和 write()，它们可以对任何打开的文件描述字进行 I/O 而不仅仅是套接字。

read()和 write()的函数原型为：

```
#include<unistd.h>
int read(int sockfd, void *buf, size_t nbytes);
int write(int sockfd, void *buf, size_t nbytes);
```

read()函数用来从已连接套接字读入数据。其中：

- ✓ 参数 sockfd 为已连接套接字。
- ✓ 参数 buf 用于保存数据的缓冲区。
- ✓ 参数 nbytes 指出期望读入的字节数。

而这个 read()函数将返回读入的字节数，返回 0 表示套接字已关闭，出错返回-1 并设置 errno。

write()函数向已连接套接字写入数据。其中：

- ✓ 参数 sockfd 为已连接套接字。
- ✓ 参数 buf 用于保存数据的缓冲区。
- ✓ 参数 nbytes 指出 buf 中期望写入的字节数。

write()函数将成功返回写入的字节数，出错返回-1 并设置 errno，向一个已关闭套接字多次调用 write()将引发 SIGPIPE 信号。

read()/write()调用，并非总是返回符合我们想要的字节数，因为返回的是实际写入和实际读入的字节数，这个数字可能等于或小于我们的期望值，所以对 read()和 write()调用结束后一定要检查它的返回值。

我们曾经学过,一个文件描述字的 I/O 可以是阻塞或非阻塞的方式(通过调用对 fcntl()设置 O_NONBLOCK 选项或调用 ioctl()对文件描述字设置 FIONBIO)，这对于套接字也同样适用，默认时套接字为阻塞方式，此时 read()和 write()会阻塞在 I/O 操作上，直至以下几种情况发生才会返回：

- ✓ 有数据到来或写出的数据被对方 TCP 层确认才返回，I/O 函数返回实际读入或写出的字节数。
- ✓ 套接字出错，如 TCP 超时或收到了 RST，read()和 write()返到-1。
- ✓ 套接字关闭，read()调用返回 0，write()调用返回-1。
- ✓ 阻塞时收到了信号，read()和 write()调用返回-1，并设置 errno 为 INTR。

我们的焦点集中到第二项和第四项。

假设下面一种情况发生，服务器端和客户端已经建立了连接，服务器主机因为某种原因非正常关机后重启(如电源失效)，此时并没有经历 TCP 的关闭过程，在服务器主机启动后，原有的已连接套接字在服务器端已经不存在，此时当客户端调用 write()连续向服务器端发送数据会引发什么？当第一次调用 write()发送数据到服务器端，服务器端 TCP 层发现没有该套接字存在时会返回一个 RST 表示出错，此时客户端的 write()调用将返回-1，客户端的 TCP 层在收到 RST 后也会将该套接字状态由连接状态变为关闭状态，如果客户端无视返回值再次对一个已关闭的套接字继续调用 write()操作将引发 SIGPIPE 信号，在默认情况下，SIGPIPE 信号将导致进程退出。

现在考虑另外一种情况:当进程阻塞在 I/O 调用(如 read()或 write())上时我们收到了一个信号(如用 kill 命令给进程发送信号),此时会发生什么呢？当进程收到信号时,read()和 write()将返回错误，并设置 errno 为 EINTR，这个问题可以通过对该信号设置信号处理函数，并在调用 sigaction()时置上 SA_RES
TART 标志来解决，或者我们判断当 errno 等于 EINTR 时重新调用 read()或 write()来解决。

我们特别为此封装了两个函数：my_read()和 my_write()。封装，仅供参考，可不敲。

```
int my_read(int fd,void *buf,size_t len)
{
  int cc,total=0
```

```
  while(len>0)
  {
    if((cc=read(fd,(char*)buf,len))<0)
    {
      if(errno==EINTR)
      {
        continue;
      }
      return cc;
    }
    if (cc==0)
    {
      break;
    }
    buf=((char*)buf)+cc;
    total+=cc;
    len-=cc;
  }
  return total;
}
```

```
int my_write(int fd,coast void*buf,size_t len)
{
    ssize_t cc;
    ssize_t total;

    total=0;

    while(len>0)
    {
      if((cc=write(fd,(const char*)buf,len))<0)
      {
         if (errno==EINTR)
         {
           continue;
         }
          return cc;
       }
```

```
            total+=cc;
            buf=((const char*)buf)+cc;
            len=cc;
        }
        return total;
}
```

my_read()和 my_write()确保在通讯正常的情况下写入和读取用户期望的字节数，我们根据返回的字节数更新我们期望读写的字节数。在 I/O 出错时判断 errno 是否等于 EINTR，如果相等则重新启动 I/O，如果不是 EINTR 则直接返回错误。

示例 4-3：简单的 ECHO 服务、客户端例子

现在，我们用前面章节所介绍的知识来编写一个完整的 TCP 客户端程序的例子，这个简单的例予是完成下述功能的一个 ECHO 客户端部分：

- ✓ 客户端从标准输入(stdin)读入一行文本，发送到服务器上
- ✓ 服务器端从网络读入此行数据，并返回给客户端
- ✓ 客户端读入服务器返回的数据写入到标准输出

Echo 程序是一个简单而有效的网络应用程序例子，客户和服务器端模型所需要的基本步骤都需要用到，所以选择 Echo 作为我们的应用范例。并且，我们会在以后新的章节中不断扩充它，使之更加完善。

下面是 ECHO 客户端代码(它接收用户输入发送给服务器端的 2029 端口，接收服务器端的响应并将它显示给用户)，服务器端代码我们将在 4.2.12 节给出。

源代码：mynet.h

```
#ifndef __MYNET_H
#define __MYNET_H

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/stat.h>
```

```
#include <errno.h>

#define RET_OK 0
#define RET_ERR -1

#define LISTEN_QUEUE_NUM 5

#define BUFFER_SIZE 256

#define ECHO_PORT 2029



#endif /*__MYNET_H*/
```

源代码：client.c

```
#include "mynet.h"

int main(int argc, char *argv[])
{
    int sockfd, ret = RET_OK;
    struct sockaddr_in servaddr;
    struct hostent *server;

    char buffer[BUFFER_SIZE];

    if (argc < 2) {
        fprintf(stderr,"usage %s hostname\n", argv[0]);
        return RET_ERR;
    }
    if((server = gethostbyname(argv[1])) == NULL)
    {
        herror("gethostbyname. ");
        return RET_ERR;
    }

    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("ERROR opening socket");
        return RET_ERR;
```

```
    }
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = *(uint32_t *)server->h_addr;
    servaddr.sin_port = htons((uint16_t)ECHO_PORT);

    if ((ret = connect(sockfd,(struct sockaddr *)&servaddr,
sizeof(servaddr))) < 0)
    {
        perror("ERROR connecting");
        goto failed;
    }
    while(1)
    {
        printf("Enter the message  : ");
        memset(buffer,0,sizeof(buffer));
        if(fgets(buffer, sizeof(buffer) - 1, stdin) == NULL)
        {
            break;
        }
        if((ret = write(sockfd,buffer,strlen(buffer))) < 0)
        {
            perror("ERROR writing to socket");
            break;
        }
        if((ret = read(sockfd,buffer, sizeof(buffer) - 1)) < 0)
        {
            perror("ERROR reading from socket");
            break;
        }
        if(ret == 0)
        {
            printf("Server disconnect\n");
            break;
        }
        buffer[ret] = 0;
        printf("Server echo message: %s\n",buffer);
    }
```

```
failed:
    close(sockfd);
    return ret < 0 ? RET_ERR : RET_OK;
}
```

我们的源代码很简单，它首先调用 socket()建立一个 TCP 套接字，请注意地址族是 AF_INET，协议为SOCK STREAM，然后用用户输入的参数作为参数调用 gethostbyname()，之所以这么做的原因是用户输入的可能是服务器的 IP 地址
，也可能是域名，所以我们用 gethostbyname()确保能够获取服务器的 IP 地址，然后我们用地址和地址族和端口号填充 IPv4 地址族结构 sockaddr_in。当我们调用 connect()函数时，我们要强制把指向 sockaddr_in 结构的指针转换成 sockaddr 结构类型的指针，关于这点我们前面提到过。最后程序在一个循环中不断调用 fgets()取得用户输入，调用 write()向服务器发送用户的输入信息，并调用 read()取得服务器返回的结果。

当 I/O 调用返回负数，表示出错。当 read()的返回值等于 0 时，表示服务器断开连接。我们的客户端将打印对应的信息后退出。

4.2.9 给本地套接字赋予地址和端口(bind)

在上一节中我们学习了套接字客户端代码的编写,现在我们开始学习如何编写服务器程序，在开始编写程序之前需要学习服务器端几个重要的系统调用，他们分别是 bind()，listen()，accept()。

进行通讯的套接字通常必须有一个名字，这就像一台电话机必须要有号码才能通话，邮递信件必须写明收信地址一样，每一种通信的套接字都必须有一个地址才能被其他进程访问。用 socke()创建的套接字在开始时是没有地址和它相连的，因此其他进程无法访问它，也无法从它接收消息。当给予了套接字一个地址时进程才能真正的使用它通讯，我们称之为“为套接字捆绑(bind 或绑定)地址”，也称为套接字命名。我们前面提到过，当客户端模型调用 connect()的起始阶段 TCP 层首先会为套接字找到一个空闲的端口号，根据服务器地址查询路由表选择并捆绑一个本地 IP 地址，并使用这个地址和端口与服务器通讯。服务器端和客户端不同，它并不调用 connect()主动连接其他主机，而是监听本地端口和地址被动的等待客户端的连接请求。另外，服务器端程序不同于客户端程序，他不能依赖 TCP 为我们选择的空闲端口(客户端也可以调用 bind()为自己指定一个本地端口和本地地址，但很少这么做)，而是需要明确指出要监听的端口号，这样才能让客户端连接服务器。通常需要调用 bind()函数为套接字捆绑本地端口和地址。

bind()函数的原型：

```
#include<sys/types.h>
#include<sys/socket.h>
int bind(int sockfd,const struct sockaddr *addr,socklen_t addelen);
```

其中：

- ✓ sockfd 参数为成功调用 socket 函数返回的套接字。
- ✓ addr 参数用于指定本地端口和地址。
- ✓ addrlen 参数指出 addr 结构的字节数
- ✓ bind()调用成功返回 0。否则返回-1 并将错误码存放于全局变量 errno 之中，失败的原因可能是指定的地址格式不正确，或指定 TCP 端口已经被其他进程的套接字占用，或者已经对该套接字调用过 bind()。

一台主机可以有多个网络接口和多个 IP 地址，如果我们只关心某个地址的连接请求，我们可以指定一个具体的本地 IP 地址，如果要响应所有接口上的连接请求就要使用一个特殊的地址 INADDR_ANY。

```
#define     INADDR_ANY     (u_int32_t)0x00000000
```

系统将 INADDR_ANY 地址解释为程序运行所在主机中任何合法的网络地址。具有这种地址的套接字可以从指定的端口收到消息，而不给套接字限定接收数据的地址或接口。

4.2.10 给连接排队(listen)

当我们给服务器端套接字调用 bind()指定了本地 IP 地址后，这就如同电话总机已经有了具体的号码，而端口号就如同我们的分机号码，此时需要把分机电话的振铃打开，同时还要对同时拨入的多个电话进行排队，下面提到的 listen()函数就完成这一功能：

```
#include<sys/socket.h>
int listen(int sockfd, int backlog);
```

其中：

- ✓ sockfd 参数为成功调用 socket 函数返回的套接字，并已经成功调用 bind()。
- ✓ backlog 参数告诉套接字在忙于处理上一个请求时还可以接受多少个进入的请求，换句话说，这决定了挂起连接的队列的大小。listen()调用成功返回 0，失败返回-1 并将错误码存放于全局变最 errno 之中。

4.2.11 接受网络连接(accept)

accept()调用类似于听见电话铃响后接起电话,此时你已经建立起一个与你的客户的连接，这个连接保持直到你或你的客户挂线。网络服务器端也有类似的过程，它使用 accept()函数接受客户端的连接：

```
#include<sys/socket.h>
int accept(int sockfd, struct sockaddr *addr,socklen_t *addrlen);
```

其中：

- ✓ 参数 sockfd 为成功调用 socket()返回的套接字,并且已经以它为参数成功调用了 bind()和 listen()。
- ✓ addr 参数用来返回客户端的地址和端口号(如果使用 IPv4 地址族，那么它需要我们传递一个指向 sockaddr_in 结构的指针)。
- ✓ addrlen 参数是个“值-结果”参数，他是个指向 32 位整数的指针，调用前，我们将 addrlen 所指向整数值置为由 addr 地址结构所占用的字节，返回时，这个整数的值将被填写成地址结构的准确长度。

返回非负数说明 accept()调用成功，返回值为一个新的套接字，这个套接字代表着与客户的 TCP 连接，服务器端将使用这个新套接字与客户通信。accept()返回的套接字我们称之为已连接套接字，对于服务器端调用 socket()返回的套接字我们称之为监听套接字。

4.2.12 一个单客户服务器例子

下面我们着手编写单客户 Echo 服务器端代码，首先观察图 4-12，它描述了单客户服务器的工作流程：

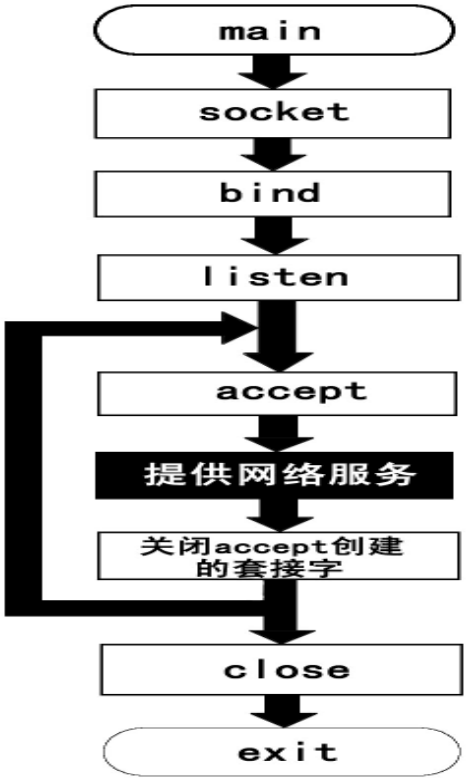


图 4-12 一个单客户端服务器工作流程

上图描述了 Echo 服务器的工作流程，它首先调用 socket()创建一个 TCP 套接字，然后调用 bind()给套接字赋了 INADDR_ANY 地址和 2029 端口，然后调用 listen()，在没有客户连接时，它一直等待在 accept()的调用上。当有连接到来时 accept()返回新的套接字，我

们在另一个循环中调用 my_readline()读入用户请求，然后再调用 write()写回应答，直至连接结束服务器才再次回到对 accept()的等待上。下面我们将给出程序的源代码：

源代码：server.c，仅供参考，可不敲此处代码。

```
static int my_readline(int fd, void *buf, unsigned len)
{
    int n, rc;
    char   c, *ptr;

    ptr = buf;
    for (n = 1; n < len; n++) {
again:
        if ( (rc = read(fd, &c, 1)) == 1) {
            *ptr++ = c;
            if (c == '\n')
                break;
        } else if (rc == 0) {
            if (n == 1)
                return(0);
            else
                break;
        } else {
            if (errno == EINTR)
                goto again;
            return(-1);
        }
    }

    *ptr = 0;
    return(n);
}
```

my_readline()函数封装了 read()调用，每次读入一个字节，并检查该字节是否为换行字符，它确保读取一整行内容。

源代码：server.c

```
static int proc_echo(int sockfd)
{
      int ret;
      char buf[BUFFER_SIZE]={0};
       while(1)
```

```
        {   memset(buf,0,sizeof(buf));//使用之前清空buffer
            if((ret = read (sockfd, buf, sizeof(buf) - 1)) < 0)
            {
                perror("read");
              return -1;
            }
            else
                if(ret == 0)
                {
                    printf("client disconnect.\n");
                    return 0;
                }

            buf[ret] = 0;
            if((ret = write(sockfd , buf, strlen(buf))) < 0)
            {
                perror("write");
                return -1;
            }
        }
        return -1;
}
```

在 proc_echo()函数中我们做了一个循环，在这个循环之中，首先调用 my_readline()读入一行来自客户端的输入，并调用 write()将它返回给客户端。

源代码：server.c

```
#include "mynet.h"

int main(int argc, char *argv[])
{
    int sockfd, nsock, ret = 0;
    uint32_t len;
    struct sockaddr_in servaddr, cliaddr;

    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("ERROR opening socket");
        return RET_ERR;
    }
```

```
    memset(&servaddr, 0, sizeof(servaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(ECHO_PORT);
    if ((ret = bind(sockfd, (struct sockaddr *) &servaddr,
sizeof(servaddr))) < 0)
    {
        perror("ERROR on binding");
        goto failed;
    }
    if((ret = listen(sockfd,LISTEN_QUEUE_NUM)) != 0)
    {
        perror("ERROR on listen");
        goto failed;
    }

    while(1)
    {
        len = sizeof(cliaddr);
        if((nsock = accept(sockfd, (struct sockaddr *) &cliaddr,
(uint32_t *)&len)) < 0)
        {
            perror("accept");
            break;
        }
        proc_echo(nsock);
        close(nsock);
    }
failed:
    close(sockfd);
    return ret < 0 ? RET_ERR : RET_OK;
}
```

main()函数首先调用 socket()创建一个 TCP 套接字，地址族为 AF_INET，然后用 INADDR_ANY 做为地址填充 sockaddr_in 结构，表示我们关注所有本地接口和地址的连接，并将监听端口设置为 2029，当我们调用 bind()时，也和调用 connect()一样需要把指向 sockaddr_in 地址结构的指针强制转换成 sockaddr 结构类型的指针。调用 listen()时把等待连接列队的数量填写成 LISTEN_UEUE

_NUM(5)。然后我们在一个循环中调用 accept()取得客户端的连接。并将 accept()返回的已连接文件描述字作为参数调用 proc_echo()。

我们的 echo 服务器代码只能同时服务于一个客户端,因为每当 accept()返回一个连接,我们就进入另一个循环内部处理客户请求,此时即使有新的连接请求到来,服务器端也没有机会调用 accept()处理新的连接,这个过程一直到客户端关闭连接。在下一节我们将讲述一个并发服务器的模型来改善这种情况。

4.2.13 创建子进程

在我们介绍多客户端服务器的例子之前,我们需要学习一下 fork()系统调用:

```
#include<sys/types.h>
#include<unistd.h>
pid_t fork(void);
```

Fork()函数用来创建一个子进程,成功调用 fork()将返回两次,一次在子进程中返回 0,另一次在父进程中返回子进程的进程 ID(一个大于 0 的整数)。该函数失败返回-1 并设置 errno 为出错代码,我们可以通过 fork()的返回值判断当前该执行是父进程还是子进程的代码。

关于 fork()的执行过程参见图 4-13。

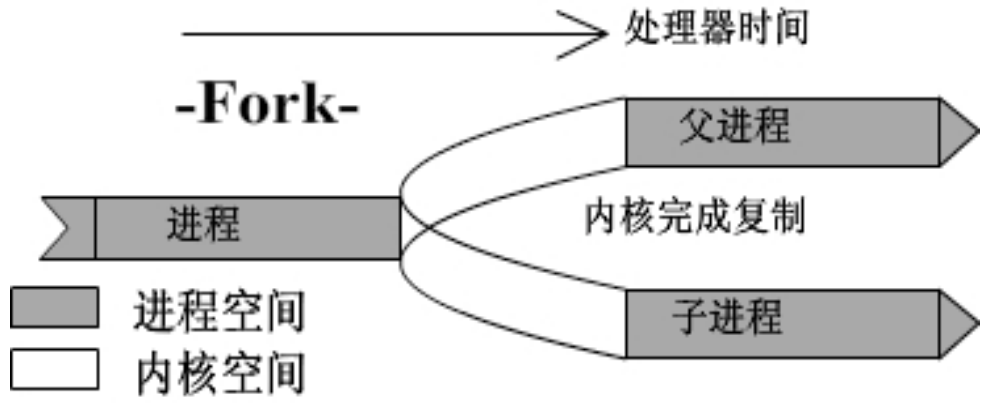


图 4-13 fork()调用过程图解

fork()调用将引发内核对父进程可写段的复制(不是立即复制,写时复制),包括打开的文件描述字和套接字,也就是说在父进程中打开的套接字在子进程中仍然可用,那么子进程中的套接字是新建一条连接呢?还是沿用父进程的连接?

当 fork()调用时,内核对文件描述字和套接字进行简单的处理,对套接字简单的递增引用计数器,也就是说,当 fork()调用前在父进程中已经打开的套接字的引用计数器将被递增,我们在上面已经提到,当调用 close()关闭一个引用计数器大于 1 的套接字时,只是简单的递减引用计数器而不真的关闭连接,那么如果我们想关闭连接就必须要父进程和子进程都对该文件描述字调用 close
(),使它的引用计数器降至为零,才会引发真正的关闭过程。

创建子进程引发一个严重的问题，当子进程先于父进程退出时会向父进程发送 SIGCHLD 信号，同时系统会保留了进程的退出状态，直到父进程调用 waitpid()或 wait()获取子进程退出状态或父进程退出后才消失，这就是 Unix 中经常提到的僵尸(Zombie)进程，它会严重的占用系统资源。对于僵尸进程的处理，Linux 简单的调用 sigaction()函数将 SIGCHLD 信号设置为忽略状态(SIG_IGN)，但是这在其他 Unix 系统上是不允许的(如 FreeBSD)，所以一个更通用的做法是为父进程注册 SIGCHLD 信号处理函数，并在该函数中调用 waitpid()取回子进程的退出状态，下面代码中描述了如何处理 SIGCHLD 信号，同时给出了我们自己封装的 set_signal()函数：

```
#include<sys/types.h>
#include<unistd.h>
#include<signal.h>
#include<sys/wait.h>
#include<stdio.h>

static void sig_routine(int signo)
{
 switch (signo)
 {
   case SIGCHLD:
        printf("in sig routine\n");
        while (waitpid(-1,NULL,WNOHANG)>0);
        break;
  }
  return;
}

void(*set_signal(int signo,void(*func)(int)))(int)
{
   struct sigaction act,oact;
   act.sa_handler=func;
   sigemptyset(&act.sa_mask);
   act.sa_flags=0;
   if(signo!=SIGALRM)
     act.sa_flags|=SA_RESTART;
  if(sigaction(signo,&act,&oact)<0)
return NULL;
  return(oact.sa_handler);
}
```

```
int main()
{
    set_signal(SIGCHILD,sig_routine);
    …
    if(fork()==0)
         return process();
…}
```

set_signal()是我们自己封装的信号注册函数，它内部调用 sigaction()完成信号注册的工作。sig_routine()为信号处理函数，对于 SIGCHLD 信号它调用 waitpid()获取子进程退出状态，waitpid 的第一个参数-1 表示要取回任何已经中止了的子进程退出状态，第二个参数为 NULL 说明我们不关心子进程的退出状态，第三个参数 WNOHANG 表示 waitpid()在没有其他僵尸进程的时候并不等待而是简单的返回 0。综上所述，我们必须在调用 fork()之前调用 set_signal()注册 SIGCHLD 的信号处理函数。

4.2.14 一个多客户端服务器例子

在学过 fork()调用之后，我们可以着手编写我们的并发 Echo 服务器，并发 Echo 服务器同时可以服务于多个客户端，每当有客户端连接到来时，它通过调用 fork()派生子进程为客户端提供服务。

图 4-14 描述了并发服务器的工作流程，并发服务器的父进程一直等待在 accept()调用上，每当一个连接到来，父进程调用 fork()创建一个子进程处理连接请求。

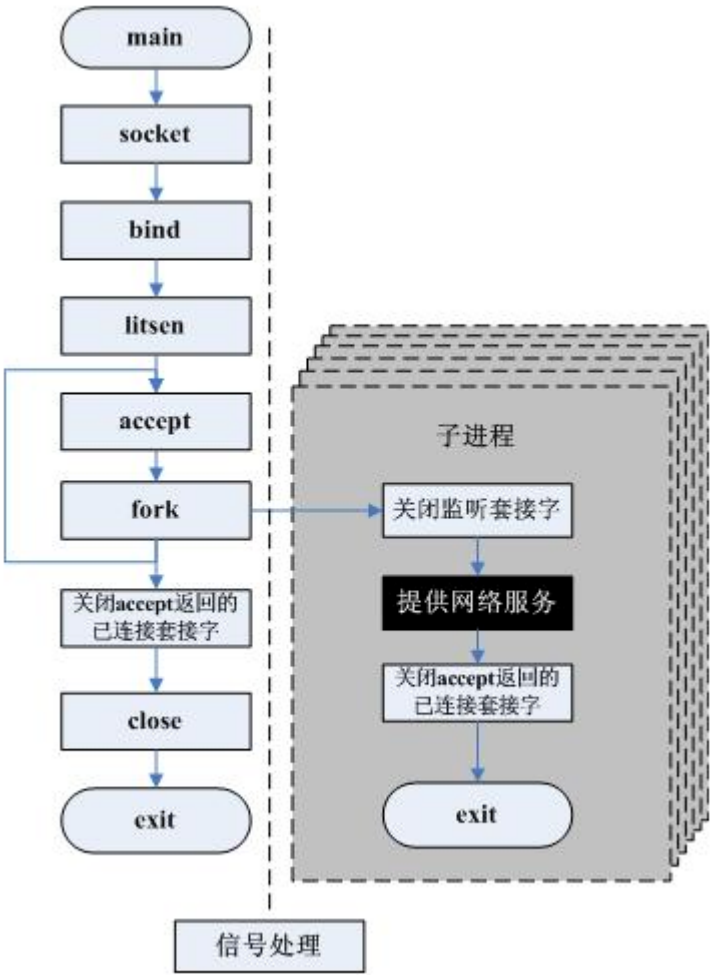


图 4-14 多客户端服务器工作流程

图中灰色的部分为子进程，白色部分为父进程。下面我们分段讲解并发多客户 Echo 服务器的源代码，我们将在下一章学习守护进程(Daemon)后重写它。

示例 4-4：并发多客户 echo 服务器。

源代码：server.c

```
#include "mynet.h"

static void (* set_signal(int signo,void (*func)(int)))(int)
{
    struct sigaction act,oact;
    act.sa_handler=func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if(signo != SIGALRM)
        act.sa_flags|=SA_RESTART;
    if(sigaction(signo,&act,&oact)<0)
```

```
        return NULL;
    return(oact.sa_handler);
}

static  void sig_routine(int signo)
{
    switch(signo)
    {
        case SIGCHLD:
            while(waitpid(-1, NULL, WNOHANG) > 0);
            break;
    }
    return;
}

static int my_readline(int fd, void *buf, unsigned len)
{
    int n, rc;
    char   c, *ptr;

    ptr = buf;
    for (n = 1; n < len; n++) {
again:
        if ( (rc = read(fd, &c, 1)) == 1) {
            *ptr++ = c;
            if (c == '\n')
                break;
        } else if (rc == 0) {
            if (n == 1)
                return(0);
            else
                break;
        } else {
            if (errno == EINTR)
                goto again;
            return(-1);
        }
    }
```

```
    *ptr = 0;
    return(n);
}

int proc_echo(int consock)
{
      int ret = 0;
       char buffer[BUFFER_SIZE]={0};
       while(1)
       {   memset(buffer,0,sizeof(buffer));
           if((ret = read(consock, buffer, sizeof(buffer)-1)) < 0)
           {
               perror("read");
               break;
           }
           else
               if(ret == 0)
               {
                   printf("client disconnect.\n");
                   break;
               }

           if((ret = write(consock, buffer, ret)) < 0)
           {
               perror("write");
               break;
           }
       }
    close(consock);
    return ret;
}

int main(int argc, char *argv[])
{
    int sockfd, nsock;
    uint32_t len;
    pid_t pid;
```

```
    struct sockaddr_in servaddr, cliaddr;
    int ret = RET_OK;

    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("ERROR opening socket");
        return RET_ERR;
    }
    memset(&servaddr, 0, sizeof(servaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(ECHO_PORT);
    if ((ret = bind(sockfd, (struct sockaddr *) &servaddr,
sizeof(servaddr))) < 0)
    {
        perror("ERROR on binding");
        goto failed;
    }
    if((ret = listen(sockfd,LISTEN_QUEUE_NUM)) != 0)
    {
        perror("ERROR on listen");
        goto failed;
    }
    if((set_signal(SIGCHLD, sig_routine)) < 0)
    {
        perror("ERROR on set_signal");
        goto failed;
    }
    while(1)
    {
        len = sizeof(cliaddr);
        if((ret = accept(sockfd, (struct sockaddr *) &cliaddr, &len)) <
0)
        {
            if(errno == EINTR)
                continue;
            perror("accept");
```

```
            break;
        }
        nsock = ret;
        if((pid = fork()) < 0)
        {
            break;
        }
        else
        if(pid == 0)
        {
            close(sockfd);
            return proc_echo(nsock);
        }
        close(nsock);
    }
failed:
    close(sockfd);
    return ret < 0 ? RET_ERR : RET_OK;
}
```

当 accept()返回一条连接后，我们将调用一次 fork()创建一个子进程，我们前面提到 fork()在子进程中将返回 0，而在父进程中返回大于 0 的 pid，所以我们在子进程中调用 proc_request()处理客户的请求。

我们在 fork()之后，父进程已经不需要 accept()返回的已连接套接字，而子进程中不再需要用于监听的套接字。所以我们分别在父进程和子进程中调用 close()关闭它们。

我们在调用 fork()之后，父进程调用 close()关闭 accept()返回的已连接套接字(nsock)，这么做会不会关闭了进程和客户的连接呢？答案是不会的，我们知道 fork()将对父进程中已经打开的每个文件描述字(套接字)增加引用计数器(这如同调用 dup2()一样，而关闭一个文件描述字时，也需要将其引用计数器递减至零才关闭它，而父进程关闭已连接套接字只会将该套接字的引用计数器递减至一，所以不会引起 TCP 连接的关闭过程，直到子进程也调用 close()关闭该套接字时，引用计数器被递减至 0，此时才会真正的引发该连接的 TCP 关闭过程，对于子进程关闭监听套接字也是同样道理。

测试题

4.1 哪一个国际标准组织是联合国的一个机构？

a. IEEE

b. ITU

c. ISO

d. ANSI

4.2 OSI 模型的哪一层提供文件传输服务？

a. 应用层

b. 数据链路层

c. 传输层

d. 表示层

4.3 OSI 模型的哪一层制定了两节点间通信的规则？

a. 传输层

b. 会话层

c. 数据链路层

d. 表示层

4.4 交换机和网桥属于 O S I 模型的哪一层?

a. 数据链路层

b. 传输层

c. 网络层

d. 会话层

4.5 路由器属于 O S I 模型的哪一层?

a. 数据链路层

b. 传输层

c. 网络层

d. 物理层

4.6 网络接口卡属于 O S I 模型的哪一层?

a. 数据链路层

b. 传输层

c. 网络层

d. 物理层

本章总结

在这一章里我们讲解了套接字编程的基本步骤，并着重讲解了 TCP 客户/服务器模型，这个模型非常重要，读者一定要牢记这个模型。

在套接字 I/O 一节我们着重说明了 read()和 write()系统调用的使用，以及它们在套接字不同状态下的返回值，通常我们都是通过 I/O 函数察觉套接字的状态变化，这里有几个要点：

- ✓ read()函数返回读入的字节数，返回 0 表示套接字已关闭，出错返回-1 并设置 errno
- ✓ write()对一个已关闭的套接字进行写操作将返回一 1 并引发 SIGPIPE 信号
- ✓ read()和 write()总是返回实际读写的字节数，它小于或等于我们的期望值，所以对返回值的检查变得非常重要。在创建子进程一节，我讲述了 fork()调用的使用，它有助于我们编写并发服务器程序。在讲述 fork()的同时我们还给出使用它所导致的僵尸进程的处理：
- ✓ 捕获 SIGCHLD 信号并创建信号处理函数，在信号处理函数中调用 waitpid()取回子进程退出状态

在这一章的最后，我们给出了echo并发服务器的例子，它是一个极其典型的Linux/Unix并发服务器模型，希望读者能够熟练掌握这个模型及其编程方法。

第五章 网络编程(下)

引言：

学完本章内容以后，你将能够

学会 I/O 复用,UDP 编程

掌握 select()系统调用的使用方法

学会使用 UDP 传输数据以及广播的使用

5.1 I/O 复用

在上一章看到 TCP 客户端同时处理两个输入：标准输入和 TCP 套接字。我们遇到的问题是客户阻塞于(标准输入上的)fgets()调用，而服务器进程退出。服务器 TCP 虽正确地给客户 TCP 发了一个 FIN，但客户进程正阻塞于从标准输入读操作，它直到对套接字调用 read()时才能察觉到连接关闭(read()返回 0)，这可能已经过了很长时间。我们需要这样的能力：如果一个或多个 I/O 条件满足(例如，输入已“准备好”，或者文件描述符可以承接更多的输出)时，我们会得到通知。这个能力被称为 I/O 复用，是由系统调用 select()/poll()支持的。

I/O 复用典型应用场合

- ✓ 当程序处理多个文件描述符和套接字时，必须使用 I/O 复用。
- ✓ 如果一个 TCP 服务器既要处理监听套接字，又要处理已连接套接字，一般也要用到 I/O 复用。
- ✓ 如果一个服务器既要处理 TCP，又要处理 UDP，一般也要使用 I/O 复用。
- ✓ 如果一个服务器要处理多个服务或者多个协议，一般要使用 I/O 复用。
- ✓ I/O 复用并非只限于网络编程，许多其他类型的应用程序也需要使用这项技术。

5.1.1 I/O 模型

在介绍如何使用 I/O 复用编程之前，让我们先来看看 Linux 下可用的五个 I/O 模型：

阻塞式 I/O
非阻塞式 I/O
I/O 复用（select()和 poll()）
信号驱动 I/O(SIGIO)
异步 I/O(Posix.1 的 aio-系列函数)

5.1.2 select 系统调用

在这一节中我们将详细介绍 I/O 复用的使用方法，I/O 复用中最常见的就是使用 select()系统调用函数，它起源于 BSD，它允许进程阻塞在等待多个事件中，并仅在一个或多个事件发生或经过指定的时间后才唤醒进程。

也就是说使用 select()时，我们只需通知内核我们对哪些文件描述符感兴趣（读、写或异常条件）以及等待多次时间，select()会替我们测试文件描述符是否“准备好”。而且 select()不仅仅适用于套接字，任何文件描述符都可以用 select()来测试，让我们首先看 select()的函数原型及参数定义：

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

```
#include<sys/time.h>
#include<sys/types.h>
#include<unistd.h>
struct timeval{
 long tv_sec;/* 秒数*/
 long tv_usec;/*微妙（百万分之一秒）*/
 };
int select(int maxfd,fd_set *readset,fd_set *writeset,fd_set
*exceptset,timeval
*timeout)
```

我们从此函数的最后一个参数开始介绍,它告诉内核等待一组指定的文件描述符中的任一个“准备好”最多等待多长时间，结构 timeval 指定了秒数和微秒数两个域，要等待的时间为秒数和微秒数之和。如果在超时之前 select()函数成功返回，它将返回一个正整数表示集合中有几个文件描述符已经“准备好”，这个数字为集合中准备好的条件之和，例如当我们关心一个文件描述符的读和写两个状态是否“准备好”，如果返回时该文件描述符读和写都“准备好”则返回 2 而不是 1。select()返回 0 表示超时(没有文件描述符准备好)。出错时 select()返回-1，并设置 errno 指出错误原因(如被信号中断，errno 等 EINTR)。

用不同的 timeout 参数调用 select()可以指定以下三种等待情况：

永远等待下去：仅在有一个文件描述符准备好 I/O 时才返回，为此，我们将参数 timeout 设置为 NULL。

等待固定时间：在有一个文件描述符准备好 I/O 时返回，但不超过由 timeout 参数所指 timeval 结构中指定的秒数和微秒数。

根本不等待：检查文件描述符后直接返回，这称为轮询(polling)。为了实现这一点，参数 timeout 必须指向结构 timevai，且定时器的值(由结构 timeval 指定的秒数和微秒数)必须为 0。

在前两者情况的等待中，如果进程捕获了一个信号并从信号处理程序返回，那么等待可能被中断。为了防止 select()被信号中断，我们必须在安装信号处理程序时设置标志 SA_RESTART 以自动重启被中断的 select()。这意味着，为了更好的可移植性，我们在捕获信号时，必须处理 select()返回 EINTR 错误。

虽然结构 timeval 为我们指定了一个微秒级的分辨率,但内核支持的分辨率可能要粗糙的多。另外还有调度延迟现象,即定时器时间到后内核还需花一点时间调度相应进程的运行。

如果 select()在超时之前返回(可能是因为有文件描述符“准备好”，或者调用进程收到了信号等)，在 Linux 操作系统中，select()的参数 timeout 在它返回时将被修改成剩余的时间，用户可以通过它知道在等待中花去的时间，但是在其他 UNIX 操作系统中 timeout 是不会被 select()修改的，我们不能依赖 Linux 的这一行为，这将导致我们的源代码不可移植。如果我们想知道在等待中花去了多少时间，最好的做法是在调用 select()函数之前得到系统时间，返回后再次取得系统时间，两者相减即可。同样我们也不能依赖 UNIX 的

select()实现不修改 timeout 这一行为，我们必须在每次调用 select()之前对它最新填充时间值。

下面我们介绍 readset，writeset 和 exceptset 参数，它们指定我们要让内核测试读、写、异常条件所需的文件描述符集合，目前只支持两个异常条件：

套接字带外（out of band）数据的到达（带外数据超出了本书的范围）

终端控制状态信息的存在

fd_set 类型是使用一个包含在结构中的数组的位域表示文件描述符集合，所以我们可以用 C 语言中的赋值语句将其赋值成另外一个 fd_set 类型的变量。下面展示的四个接口(多个操作系统中被定义成宏）分别用来初始化 fd_set 类型的数据、设置文件描述符对应的比特位、清除某个文件描述符对应的比特位和测试某个文件描述符是否被置位:

```
#include<sys/select.h>
#include<sys/types.h>
#include<unistd.h>
FD_ZERO(fd_set *fdset);                 /*clear all bits in fdset*/
FD_SET(int fd,fd_set *fdset);          /*turn on the bit for fd in fdset*/
FD_CLR(int fd,fd_set *fdset);          /*turn off the bit for fd in fdset*/
FD_ISSET(int fd,fd_set *fdset);        /*is the bit for fd on in fdset*/
```

其中：

FD_ZERO()把 fdset 指向的文件描述符集合对应的位全部初始化为 0

FD_SET()将 fdset 指向的文件描述符集合中对应文件描述符 fd 的位(bit)置为 1，表示我们关心该文件描述符的状态

FD_CLR()与 FD_SET()相反，他将 fdset 指向的文件描述符集合中对应文件描述符 fd 的位置（bit）置为 0，表示我们不关心该文件描述符的状态

FD_ISSET()用来测试 fdset 指向的文件描述符集合中对应文件描述符 fd 的位（bit）是否被置为 1

对 fd_set 文件描述符集合的初始化工作是极为重要的。如果集合作为一个自动变量分配而未初始化，那将导致不可预测的后果。

函数 select()返回时将修改由指针 readset、writeset 和 exceptset 所指的文件描述符集合。这三个参数均为“值-结果”参数。当我们调用 select()函数时，指定我们所关心的文件描述符集合，当返回时，结果指示哪些文件描述符已“准备好”，我们可以用宏 FD_ISSET 来测试 fd_set 中的文件描述符，文件描述符集合中没有“准备好”的文件描述符相对应的位被清 0（FD_ISSET 测试将返回 0），而集合中“准备好”的文件描述符对应 bit 将置为 1(FD_ISSET()将返回 1)。对此，每次调用 select()，我们都必须重新使用 FD_SET()设置文件描述符集合中关心的文件描述符。

我们知道文件描述符/套接字是一个整数，我们定义一个 fd_set 类型的变量，并打开文件描述符 1（默认为标准输出）、4 和 5 的相应位，调用 select()时我们只关心它们的读状态，并且不设置超时时间，直到等到有一个文件描述符准备好或收到信号，例如如下代码：

```
#include<sys/select.h>
Fd_set rest;
int ret;
FD_ZERO(&rset);                                  /*初始化文件描述符集
合*/
FD_SET(1,&rset)                                  /*打开文件描述符 1*/
```

```
FD_SET(4,&rset);                                 /*打开文件描述符 4*/
FD_SET(5,&rset);                                 /*打开文件描述符 5*/
 if((ret=select(5+1,&rest,NULL,NULL,NULL))< 0)
 {
    perror(“select”);
    return
 }
 if(FD_ISSET(4,&rest))
 {
    /*文件描述符 4 准备好*/
    …
 }
 if(FD_ISSET(5, &rest))
 {
   /*文件描述符 5 准备好*/
   …
 }
```

头文件<sys/select.h>中定义的常值 FD_SETSIZE，是数据类型 fd_set 的文件描述符数量，其值通常是 1024，但很少有程序使用那么多(大)的文件描述符。参数 maxfd 要求我们计算出所关心的最大文件描述符并将此值通知内核。例如，前面给出的例子打开文件描述符 1、4 和 5，其 maxfd 值就应是 6(5+1)。对最大的文件描述符加 1 这点很重要，其原因就在于：我们指定的是文件描述符的个数而不是最大值，而文件描述符是从 0 开始的。maxfd 参数之所以存在，是因为它减少了内核测试和复制整个 fd_set 结构到内核空间给系统带来的负担。

使用 select()的时候，三个需要注意的要点：

1.当使用 select()时，两个最常见的编程错误是：忘了对最大文件描述符加 1 和忘了文件描述符集是值-结果参数，select()返回时会将那些没“准备好”的 bit 置为 0，所以如果要再次调用 select()时，一定要重新用 FD_SET 设置你感兴趣的文件描述符的对应 bit。

2.对于一个套接字，如果该套接字关闭或出错，select()将返回该套接字既可读又可写，所以对于 select()返回可读状态不能只认为它仅仅是可读，我们还要检查 read()或 write()的返回值判断是否该连接已经关闭。

3.对于监听套接字(对其调用了 listen()的文件描述符)，select()将返回可读表示该套接字上有新的连接到来或套接字出错，我们可以调用 accept()判断这两种情况，如果 accept()返回非负数表示有新的连接，并且该返回值为新的已连接套接字，如果 accept()返回-1 表示套接字出错，如已关闭。

下面是我们修改 echo 客户端的例子代码，它调用 select()同时等待标准输入(默认为 0)和套接字的读操作“准备好”：

示例 5-1：使用select()的echo客户端

源代码：client.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <errno.h>
#include <ctype.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/select.h>



#define RET_OK   0
#define RET_ERR -1
#define LISTEN_QUEUE_NUM 5
#define BUFFER_SIZE 256
#define ECHO_PORT 2029
int main(int argc, char **argv)
{
    int sock, maxfd = 0;
    struct sockaddr_in servaddr;
    struct hostent *server;
    fd_set rset, set;
    int nfound, bytesread;
    char buf[BUFFER_SIZE]={0};

    if (argc < 2)
```

```
    {
        fprintf(stderr,"usage %s hostname\n", argv[0]);
        return RET_ERR;
    }
    if((server = gethostbyname(argv[1])) == NULL)
    {
        herror("gethostbyname. ");
        return RET_ERR;
    }
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("socket");
        return -1;
    }
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = *(uint32_t *)server->h_addr;
    servaddr.sin_port = htons((uint16_t)ECHO_PORT);
    if (connect(sock, (struct sockaddr *)&servaddr, sizeof(servaddr)) <
0)
    {
        perror("connect");
        return -1;
    }
    maxfd = fileno(stdin);
    FD_ZERO(&set);
    FD_SET(sock, &set);/*关注网口*/
    FD_SET(maxfd, &set);/*关注键盘*/
    maxfd = (maxfd > sock ? maxfd : sock) + 1;
    while(1)
    {
        rset = set;//请区分出值-结果参数，select 前后会发生变化
        if ((nfound = select(maxfd, &rset, (fd_set *)0, (fd_set *)0,
NULL)) < 0)
        {
            if (errno == EINTR) {
                fprintf(stderr, "interrupted system call\n");
                continue;
```

```
            }
            perror("select");
            exit(1);
        }
        if (FD_ISSET(fileno(stdin), &rset)) {
            if (fgets(buf, sizeof(buf)-1, stdin) == NULL) {
                //if (ferror(stdin)) {
                    //perror("stdin");
                    //return -1;
                //}
                return 0;
            }
            if (write(sock, buf, strlen(buf)) < 0)
            {
                perror("write");
                return -1;
            }
        }
        if (FD_ISSET(sock,&rset)) {
            if((bytesread = read(sock, buf, sizeof(buf)-1)) < 0)
            {
                perror("read");
                exit(1);
            }
            else if(bytesread == 0)
            {
                fprintf(stderr, "server disconnect\n");
                exit(0);
            }
            buf[bytesread] = 0;
            printf("%s\n",buf);
        }
    }
    return 0;
}
```

在示例 5-1 中，首先调用 fileno()取得标准输入的文件描述符， stdin 是缓冲式 I/O 的流指针，它不能用于 select()，所以必须使用 fileno()取得标准输入的文件描述符，它通常为 0，我们把它赋予变量 maxfd。

接着我们使用 FD_ZERO()初始化 fd_set 类型的变量 set，这是不可缺少的步骤。然后调用 FD_SET 将套接字和标准输入(maxfd)加入到 set 集合中，最后我们比较标准输入和 sock 的值，并把较大的值加一后赋予变量 maxfd，对 maxfd 加 1 的原因我们前面已经提到了，这里不再赘述。

我们并不使用 set 作为参数直接调用 select()，而是把它赋予了另外一个变量 rset，并且使用 rset 作为参数调用 select()，其原因是 select()会在返回时修改参数的值以便反映“准备好”的文件描述符，它是值-结果参数，为了避免在循环中每次都调用 FD_SET()重新设置 set 的值，所以使用另外的变量 rset 作为参数调用 select()，我们前面还提到过 fd_set 类型是一个结构，所以我们可以直接用等号赋值。

接着我们调用 select()同时等待在标准输入和套接字上，并且设置 timeout 参数为 NULL，表示如果没有任何文件描述符(套接字)“准备好”我们将永远等待下去。

如果 select()返回大于零的数字，表示有用户输入或套接字上有数据可读。我们调用 FD_ISSET()首选测试标准输入，如果用户输入了数据，FD_ISSET()测试标准输入将返回 1，我们就调用 fgets()读取数据并发送给服务器，接着我们用同样的方法测试套接字是否有数据到达，如果 FD_ISSET()返回 1，就从套接字中读入数据，并将结果显示给用户。

同客户端一样，我们也使用 select()改写服务器端代码，而不是调用 fork()实现多任务并发操作，图 5-6 描述使用 select()的服务器的工作过程：

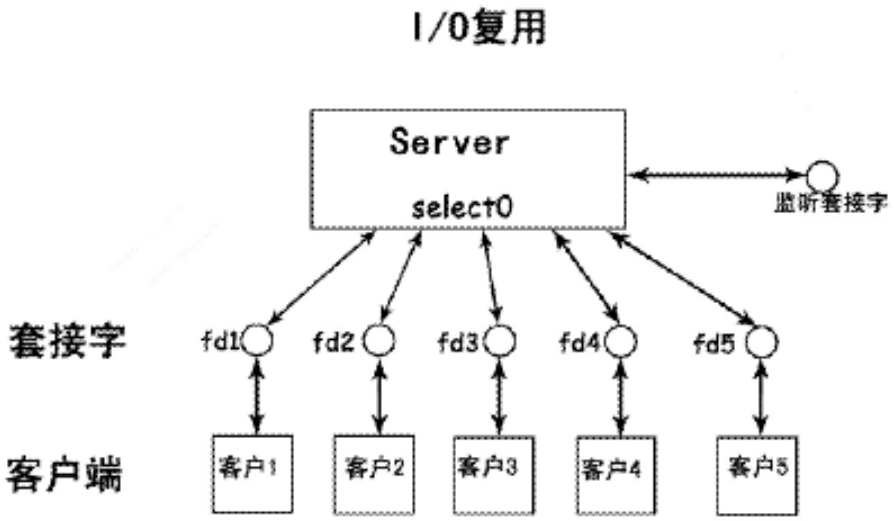


图 5-6 使用 select()多客户服务器的工作原理

使用 I/O 复用的服务器通常是调用 select()等待在监听套接字和已连接套接字上，一旦有连接到来，select()将返回大于 0 的值，并将对应的 bit 置为 1 表示这个套接字可读。如果监听套接字可读表示有连接到来，我们就调用 accept
()创建已连接套接字，并将新的已连接套接字加入到 select()的读等待集合中；如果某个已连接套接字可读，我们就读入客户端送来的数据，并将数据返回给客户端以实现 echo，当做完这一切后，再次调用 select()，下面为服务器程序源代码：

源代码：server.c

```
#include <sys/types.h>
#include <ctype.h>
#include <strings.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <ctype.h>
#include <errno.h>
#include <sys/time.h>
#include <stdio.h>
#include <string.h>
#include <sys/select.h>
#include <stdlib.h>

#define LISTEN_QUEUE_NUM 5
#define BUFFER_SIZE 256
#define ECHO_PORT 2029

int main(int argc, char **argv)
{
    struct sockaddr_in servaddr, remote;
    int request_sock, new_sock;
    int nfound, fd, maxfd, bytesread;
    uint32_t  addrlen;
    fd_set rset, set;
    struct timeval timeout;
    char buf[BUFFER_SIZE];

    if ((request_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
{
        perror("socket");
        return -1;
    }
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = INADDR_ANY;
```

```
    servaddr.sin_port = htons((uint16_t)ECHO_PORT);

    if (bind(request_sock, (struct sockaddr *)&servaddr,
sizeof(servaddr)) < 0)
    {
        perror("bind");
        return -1;
    }
    if (listen(request_sock, LISTEN_QUEUE_NUM) < 0)
    {
        perror("listen");
        return -1;
    }

    FD_ZERO(&set);
    FD_SET(request_sock, &set);
    maxfd = request_sock;
    while(1) {
        rset = set; //请区分出值-结果参数，select前后会发生变化
        timeout.tv_sec = 0;
        timeout.tv_usec = 500000;
    if((nfound = select(maxfd + 1, &rset, (fd_set *)0, (fd_set *)0,
&timeout)) < 0)
        {
            perror("select");
        return -1;
        }
        else
            if (nfound == 0)
            {
                printf("."); fflush(stdout);
                continue;
            }
        if (FD_ISSET(request_sock, &rset))
        {
            addrlen = sizeof(remote);
            if ((new_sock = accept(request_sock, (struct sockaddr
*)&remote, &addrlen)) < 0)
```

```
            {
                perror("accept");
                return -1;
            }
            printf("connection from host %s, port %d, socket %d\r\n",
                    inet_ntoa(remote.sin_addr), ntohs(remote.sin_port),
                    new_sock);
            FD_SET(new_sock, &set);
            if (new_sock > maxfd)
                maxfd = new_sock;
            FD_CLR(request_sock, &rset);//监听套接字只能accept不能read
            nfound --;
        }
        for (fd=0; fd <= maxfd && nfound > 0; fd++) {
            if (FD_ISSET(fd, &rset)) {
                nfound --;
                if ((bytesread = read(fd, buf, sizeof(buf) - 1))<0)
                {
                    perror("read");
                }
                if (bytesread == 0)
                {
                    fprintf(stderr, "server: end of file on %d\r\n",fd);
                    FD_CLR(fd, &set);
                    close(fd);
                    continue;
                }
                buf[bytesread] = 0;
                printf("%s: %d bytes from %d: %s\n", argv[0], bytesread,
fd, buf);
                if (write(fd, buf, bytesread) < 0)
                {
                    perror("echo");
                    FD_CLR(fd, &set);
                    close(fd);
                }
            }
        }
```

```
    }
    return 0;
}
```

示例 5-1 的服务器端代码（server.c 中），我们仍然使用 maxfd 来保存最大的套接字的值。服务器端调用 select()既关心监听套接字（request_sock0）也关心已连接套接字的读操作是否“准备好”，对于监听套接字的读操作“准备好”表示有新的连接到来，我们可以调用 accept()获取连接，如果 accept()返回成功并返回新的已连接套接字，我们就调用 FD_SET()将新的已连接套接字加入到集合 rset 中，如果这个新的已连接套接字的值大于 maxfd 我们将更新 maxfd 的值。而已连接套接字的读操作“准备好”则表示有数据到来，我们可以调用 read()读取数据。对于关闭或出错的连接，我们使用 FD_CLR()将从 rset 中除去。同时我们设置了超时值 500000 微秒，用来表现 select()的超时特性，并在每次超时时输出‘.’字符。

我们的服务器代码有个错误，就是如果最大套接字关闭或出错后，我们使用 FD_CLR()将从集合中除去，但是我们并没有更新 maxfd 的值，按理说，我们应该将它更新成第二大的套接字的值，但是这么做会使我们的代码变得复杂，实际上，maxfd 的值大于集合中的文件描述符的最大值并不会导致 select()出错，不过它会导致 kernel 浪费更多时间去测试更多的位，如果读者有兴趣可以修复这个错误。

5.1.3 epoll 系统调用

Linux 2.6 内核中有提高网络 I/O 性能的新方法，即 epoll。

1、为什么 select 落后

首先，在 Linux 内核中，select 所用到的 FD_SET 是有限的，即内核中有个参数 __FD_SETSIZE 定义了每个 FD_SET 的句柄个数，在 2.6.15-25-386 内核中，该值是 1024，搜索内核源代码得到：

include/linux/posix_types.h:

#define __FD_SETSIZE 1024

也就是说，如果想要同时检测 1025 个句柄的可读状态是不可能用 select 实现的。或者同时检测 1025 个句柄的可写状态也是不可能的。其次，内核中实现 select 是使用轮询方法，即每次检测都会遍历所有 FD_SET 中的句柄，显然，select 函数的执行时间与 FD_SET 中句柄的个数有一个比例关系，即 select 要检测的句柄数越多就会越费时。当然，在前文中并没有提及 poll 方法，事实上用 select 的朋友一定也试过 poll，个人觉得 select 和 poll 大同小异，个人偏好于用 select 而已。

2、内核中提高 I/O 性能的新方法 epoll

epoll 是什么？按照 man 手册的说法：是为处理大批量句柄而作了改进的 poll。要使用 epoll 只需要以下的三个系统函数调用：epoll_create(2)， epoll_ctl(2)， epoll_wait(2)。

3、epoll 的优点

支持一个进程打开大数目的 socket 描述符(FD)

select 最不能忍受的是一个进程所打开的 FD 是有一定限制的，由 FD_SETSIZE 设置，默认值是 1024。对于那些需要支持上万连接数目的 IM 服务器来说显然太少了。这时候你一是可以选择修改这个宏然后重新编译内核，不过资料也同时指出这样会带来网络效率的下降；二是可以选择多进程的解决方案(传统的 Apache 方案)，不过虽然 linux 上面创建进程的代价比较小，但仍旧是不可忽视的，加上进程间数据同步远比不上线程间同步高效，所以这也不是一种完美的方案。不过 epoll 没有这个限制，它所支持的 FD 上限是最大可以打开文件的数目，这个数字一般远大于 select 所支持的 2048。举个例子，在 1GB 内存的机器上大约是 10 万左右，具体数目可以 cat /proc/sys/fs/file-max 察看，一般来说这个数目和系统内存关系很大。

IO 效率不随 FD 数目增加而线性下降

传统 select/poll 的另一个致命弱点就是当你拥有一个很大的 socket 集合，由于网络得延时，使得任一时间只有部分的 socket 是"活跃"的，而 select/poll 每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是 epoll 不存在这个问题，它只会对"活跃"的 socket 进行操作---这是因为在内核实现中 epoll 是根据每个 fd 上面的 callback 函数实现的。于是，只有"活跃"的 socket 才会主动去调用 callback 函数，其他 idle 状态的 socket 则不会，所以 epoll 的效率就远在 select/poll 之上了。

4、epoll 的工作模式

令人高兴的是，linux2.6 内核的 epoll 比其 2.5 开发版本的/dev/epoll 简洁了许多，所以，大部分情况下，强大的东西往往是简单的。唯一有点麻烦的是 epoll 有 2 种工作方式：LT 和 ET。

LT(level triggered)是缺省的工作方式，并且同时支持 block 和 no-block socket。在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的 fd 进行 IO 操作。如果你不作任何操作，内核还是会继续通知你的，所以，这种模式编程出错误可能性要小一点。传统的 select/poll 都是这种模型的代表。

ET (edge-triggered) 是高速工作方式，只支持 no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核就通过 epoll 告诉你，然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知。

epoll 只有 epoll_create,epoll_ctl,epoll_wait 3 个系统调用。

5、epoll 的使用方法

epoll 用到的所有函数都是在头文件 sys/epoll.h 中声明的，下面简要说明所用到的数据结构和函数：

所用到的数据结构：

```
typedef union epoll_data {
void *ptr;
int fd;
```

```
__uint32_t u32;
__uint64_t u64;
} epoll_data_t;

struct epoll_event {
__uint32_t events; /* Epoll events */
epoll_data_t data; /* User data variable */
};
```

结构体 epoll_event 被用于注册所感兴趣的事件和回传所发生待处理的事件，而 epoll_data 联合体用来保存触发事件的某个文件描述符相关的数据。例如一个 client 连接到服务器，服务器通过调用 accept 函数可以得到这个 client 对应的 socket 文件描述符，可以把这文件描述符赋给 epoll_data 的 fd 字段，以便后面的读写操作在这个文件描述符上进行。epoll_event 结构体的 events 字段是表示感兴趣的事件和被触发的事件，可能的取值为：

EPOLLIN：表示对应的文件描述符可以读；

EPOLLOUT：表示对应的文件描述符可以写；

EPOLLPRI：表示对应的文件描述符有紧急的数据可读；

EPOLLERR：表示对应的文件描述符发生错误；

EPOLLHUP：表示对应的文件描述符被挂断；

EPOLLET：表示对应的文件描述符有事件发生；

6. epoll 所用到的系统 API

1)、epoll_create 函数

函数声明：int epoll_create(int size)

该函数生成一个 epoll 专用的 instance 实例文件描述符，其中的参数是指定生成描述符的最大范围。参数 size 的作用已经失效了。

2)、epoll_ctl 函数

函数声明：int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)

该函数用于控制某个文件描述符上的事件，可以注册事件，修改事件，删除事件。

参数：

epfd：由 epoll_create 生成的 epoll 专用的 instance 实例文件描述符；

op：要进行的操作，可能的取值 EPOLL_CTL_ADD 注册、EPOLL_CTL_MOD 修改、EPOLL_CTL_DEL 删除；

fd：关联的文件描述符；

event：指向 epoll_event 的指针；

如果调用成功则返回 0，不成功则返回-1。

3)、epoll_wait 函数

函数声明：int epoll_wait(int epfd,struct epoll_event * events,int maxevents,int timeout)

该函数用于轮询 I/O 事件的发生。
参数：
epfd：由 epoll_create 生成的 epoll 专用的 instance 实例文件描述符；
epoll_event：用于回传待处理事件的数组；
maxevents：每次能处理的事件数；
timeout：等待 I/O 事件发生的超时值；
返回发生事件数。

对 epoll 的操作就这么简单，总共不过 4 个 API：epoll_create, epoll_ctl, epoll_wait 和 close。以下是修改前面 select 例子代码的一个例子，基本上大同小异。

客户端代码：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <errno.h>
#include <sys/epoll.h>
#include <fcntl.h>

int main()
{
   int sockfd=0;
   int ret =0;
   struct sockaddr_in servaddr={0};
   unsigned char buf[1024]={0};

   int i;
   int epoll_instance,nfound;
   /*声明epoll_event结构体的变量，ev用于注册事件，events
   数组用于回传要处理的事件*/
   struct epoll_event ev,events[20];
```

```
    /*生成用于处理accept的epoll专用
    的文件描述符，指定生成描述符的最大范围为256*/
    epoll_instance=epoll_create(256);//size is unused nowadays
    printf("epoll_instance is %d\n",epoll_instance);
    sockfd = socket(AF_INET,SOCK_STREAM,0);
    if(sockfd <0)
    {
       perror("error opening socket");
       goto failed;
    }
    printf("sockfd is %d\n",sockfd);
    ev.data.fd=sockfd;//添加socket
    ev.events=EPOLLIN|EPOLLET;//设置要处理的事件类型
    epoll_ctl(epoll_instance,EPOLL_CTL_ADD,sockfd,&ev);//注册epoll事件

    ev.data.fd=fileno(stdin);//添加键盘
    ev.events=EPOLLIN|EPOLLET;//设置要处理的事件类型
    epoll_ctl(epoll_instance,EPOLL_CTL_ADD,fileno(stdin),&ev);//注册
epoll事件

    memset(&servaddr,0,sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET,"127.0.0.1",&servaddr.sin_addr.s_addr);
    servaddr.sin_port = htons((uint16_t)20000);

    ret = connect(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
    if(ret <0)
    {
       perror("Error connecting");
       goto failed;
    }
    while(1)
    {
        nfound=epoll_wait(epoll_instance,events,20,-1);//等待epoll事
件的发生
      for(i=0;i<nfound;i++)
      {
       if(events[i].events&EPOLLIN &&
```

```
            events[i].data.fd == fileno(stdin))/*监听事件*/
        {
              memset(buf,0,sizeof(buf));
              fgets(buf,sizeof(buf)-1,stdin);
              printf("writing socket\n");
              write(sockfd,buf,strlen(buf));
        }
        else if(events[i].events&EPOLLIN)/*读事件*/
        {
              //sockfd=events[i].data.fd;
              memset(buf,0,sizeof(buf));
              ret=read(sockfd,buf,sizeof(buf)-1);
              if(ret==0)
              {
                ev.data.fd=sockfd;
                ev.events=EPOLLIN|EPOLLET;//
                epoll_ctl(epoll_instance,EPOLL_CTL_DEL,sockfd,&ev);
                close(sockfd);
                printf("del client\n");
                continue;
              }
              printf("echo buf is %s\n",buf);
        }
       }

   }
failed:
    close(sockfd);
}
```

服务端代码：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
```

```
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <errno.h>
#include <sys/epoll.h>
#include <fcntl.h>

/*function prototype*/
/*
void setnonblocking(int sock)
{
    int opts;
    opts=fcntl(sock,F_GETFL);
    if(opts<0)
    {
        perror("fcntl GETFL");
        exit(1);
    }
    opts=opts|O_NONBLOCK;
    if(fcntl(sock,F_SETFL,opts)<0)
    {
        perror("fcntl SETFL");
        exit(1);
    }
}
*/
int main()
{
   int listenfd=0;//listen socket file
   int connfd=0;//connected socket file
   int ret =0;
   char buf[1024]={0};
   uint32_t len = 0;
   struct sockaddr_in servaddr={0};
   struct sockaddr_in cliaddr={0};
   int i;
   int epoll_instance,nfound,sockfd;
   /*声明epoll_event结构体的变量，ev用于注册事件，events
```

```
    数组用于回传要处理的事件*/
    struct epoll_event ev,events[20];
    /*生成用于处理accept的epoll专用
    的文件描述符，指定生成描述符的最大范围为256*/
    epoll_instance=epoll_create(256);//size is unused nowadays

    listenfd = socket(AF_INET,SOCK_STREAM,0);
    if(listenfd <0)
    {
       perror("error opening socket");
       return -1;
    }
    //setnonblocking(listenfd);//把用于监听的socket设置成非阻塞方式
    ev.data.fd=listenfd;//设置与要处理的事件相关的文件描述符
    ev.events=EPOLLIN|EPOLLET;//设置要处理的事件类型
    epoll_ctl(epoll_instance,EPOLL_CTL_ADD,listenfd,&ev);//注册epoll事
件

    memset(&servaddr,0,sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons((uint16_t)20000);

    ret = bind(listenfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
    if(ret <0)
    {
       perror("Error on binding");
       return -1;
    }
    ret = listen(listenfd,5);//backlog
    if(ret !=0)
    {
        perror("Error on listening");
        return -1;
    }
    while(1)
    {
```

```
      nfound=epoll_wait(epoll_instance,events,20,1000);//等待epoll事
件的发生
      if(nfound==0)
      {
        printf(".");
        fflush(stdout);
        continue;
      }
      for(i=0;i<nfound;i++)
      {
        if(events[i].data.fd==listenfd)/*监听事件*/
        {
              len = sizeof(cliaddr);
              connfd=accept(listenfd,(struct sockaddr*)&cliaddr,&len);
              printf("connection from host %s,port %d,sockfd %d\n",

inet_ntoa(cliaddr.sin_addr),ntohs(cliaddr.sin_port),connfd);
              //setnonblocking(connfd);

              ev.data.fd=connfd;//设置用于读操作的文件描述符
              ev.events=EPOLLIN|EPOLLET;//设置用于注册的读操作事件
              epoll_ctl(epoll_instance,EPOLL_CTL_ADD,connfd,&ev);//注
册ev事件
        }
        else if(events[i].events&EPOLLIN)/*读事件*/
        {
              sockfd=events[i].data.fd;
              memset(buf,0,sizeof(buf));
              ret=read(sockfd,buf,sizeof(buf)-1);
              if(ret==0)
              {
                ev.data.fd=sockfd;
                ev.events=EPOLLIN|EPOLLET;//
                epoll_ctl(epoll_instance,EPOLL_CTL_DEL,sockfd,&ev);
                close(sockfd);
                printf("del client\n");
                continue;
              }
```

```
                write(sockfd,buf,ret);
                printf("write back to client.....\n");
            }
        }
    }
}
```

5.2 UDP 编程

5.2.1 概述

TCP 与 UDP 应用程序之间存在本质差异，UDP 是无连接的、不可靠的数据报协议，而 TCP 是面向连接的，提供可靠的字节流。然而，有些情况下更适合用 UDP 而不是 TCP。

在互联网中，有很多流行的应用层协议是用 UDP 实现的，如 DNS(域名系统)、NFS(网络文件系统)和 SNMP(简单网络管理协议)就是这样的例子。另外 UDP 还提供了广播和多播的特性，这是 TCP 无法做到的。

图 5-7 给出了典型的 UDP 客户.服务器程序函数调用。客户端不与服务器建立连接，它调用函数 sendto()给服务器发送数据报，此函数要求目的地址(服务器)作为其参数。类似的，服务器不用调用 accept()接受连接，它只管调用函数 recvfrom()，等待来自某客户的数据到达。Recvfrom()在读入数据的同时返回客户的协议地址，所以服务器可以根据 recvfrom()返回的客户端地址发送应答。图 5-7 所示为典型情况下发生的 UDP 客户-服务器交互的模型，我们可将此图与图 5-5 进行比较。

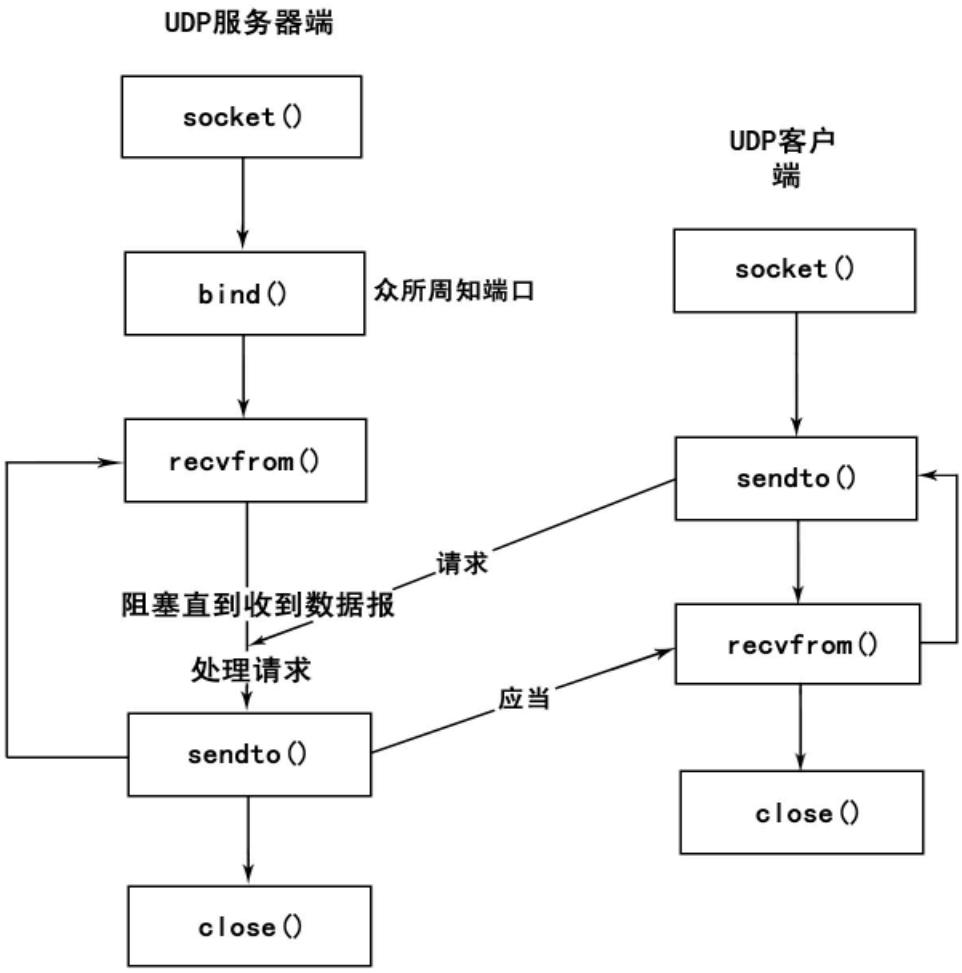


图 5-7 UDP 服务器客户端模型

在这一节中我们将介绍 UDP 套接字的 I/O 函数：recvfrom()和 sendto()并使用 UDP 协议重写我们的 echo 客户-服务器程序。我们也将介绍函数 connect()函数在 UDP 套接字中的应用。

创建一个 UDP 套接字

创建.个 UDP 套接字与创建 TCP 套接字类似，唯一的区别是 socket()函数的第二个参数不能传递 SOCK-STREAM 而必须是 SOCK-DGRAM。示例 5-12 演示如何创建一个基于 IPv4 地址族的 UDP 套接字：

```
sockfd=socket(AF_INET, SOCK_DGRAM, 0);
```

recvfrom 和 sento 调用

recvfrom()和 sendto()这两个函数类似于标准的 read()和 write()函数，通常用于使用 UDP 协议通讯的 I/O，除了和 read()/write()相同的参数外，它们还要求有三个附加参数，让我们先看看函数原型：

```
#include<sys/socket.h>
ssize_t recvform(int sockfd,void *buff,size_t nbytes,int flags,
```

```
                struct sockaddr *from,socklen_t *addrlen);
ssize_t sendto(int sockfd,const void*buff,size_t nbytes,int flags,
                const struct sockaddr  *to,  socklen_t addrlen);
```

两个函数调用成功将返回读/写的字节数(非负数)，返回-1 表示出错。

- ✓ 参数sockfd参数socket()函数返回的套接字。
- ✓ 参数buff指向保存数据的缓冲区的指针。
- ✓ 参数nbytes指出要读写字节数。
- ✓ 参数flags用来修改I/O操作的行为，它支持三个选项：
  - ✧ MSG_OOB如果下层协议为SOCK_STREAM时，该选项用来发送TCP的带外数据(out-of-band)。
  - ✧ MSG_DONTROUTE指明目的主机就在本地网络内，发送数据不必查看路由表。
  - ✧ MSG_DONTWAIT这个标志将单次1/O操作设置为使用非阻塞方式，而不需要使fcntl()或ioctl()将套接字设置成非阻塞式，当该次I/O调用结束，套接字将恢复阻塞方式，如果操作不能马上完成，内核将返回一个EAGAIN(EWoULDBLOCK)错误。

flags 参数很少使用，我们通常简单的给它传递一个 0。

函数 sendto()的参数 to 是一个含目的主机地址和端口号的地址结构，该地址结构的大小串 addrlen 参数来指定。函数 recvfrom()将发送者的协议地址装填到 from 所指的地址结构中,存储在此套接字地址结构中的字节数也以 addrlen 所指向的整数返回给调用者。注意，sendto()的最后一个参数是一整数值,lnnecvnIom()的最后一个参数是一指 l 匈整数值的指针(值一结果参数)。recvfrom()的最后两个参数类似于 accept()的最后两个参数，返回时套接字地址结构的内容告诉我们是谁发送了数据报。sendto()的最后两个参数类似于 connect()的最后两个参数，我们用数据报目的地址和端口号来装填套接字地址族结构。

对于-个 UDP 套接字的写操作一般不会阻塞，其原因是 UDP 并不等待对方的确认，它仅仅是将数据传递给下层的协议后就立即返回，而不关心数据是否真的发送，对方是否收到。

Recvfrom()和 sendto()函数将读写数据的长度作为函数返回值，出错时返回-l，并将全局变量 errno 设置为错误代码。

写一个长度为 0 的数据报是可行的，这将导致发送一个只包含 IP 头部和 8 字节 UDP 头部的 UDP 数据报。这也意味着 recvfrom()返回 0 值也是可行的，它并不表示对方已关闭了连接，这与 TCP 套接字上的 read()返回 0 的情况不同。由于 UDP 是无连接的，所以没有诸如关闭 UDP 连接之类的事情。

如果 recvfrom()的参数 from 是空指针，则相应的长度参数(addrlen)也必须是空指针，这表示我们并不关心发送方的协议地址。

Recvfrom()和 sendt0()也可以用于 TCP 套接字 I/0,，尽管一般来说没有理由这样做。

UDP echo 服务器端程序

现在，我们用 UDP 重写 echo 服务器程序。echo 客户和服务器群序依循图 5-7 中所示的函数调用流程，示例 5-5 给出它的源代码：

示例 5-5：使用 UDP 的 echo 服务器端

源代码：server.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <errno.h>

#define RET_OK   0
#define RET_ERR -1
#define LISTEN_QUEUE_NUM 5
#define BUFFER_SIZE 256
#define ECHO_PORT 2029

int main(int argc, char **argv)
{
    int sockfd, opt = 1;
    uint32_t len;
    struct sockaddr_in cliaddr;
    uint8_t buffer[BUFFER_SIZE];
    int ret = RET_OK;

    if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("ERROR opening socket");
        return RET_ERR;
    }
    if((ret = setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt,
sizeof(opt))) < 0)
    {
        perror("ERROR setsockopt");
        goto failed;
```

```
    }
    memset(&cliaddr, 0, sizeof(cliaddr));
    cliaddr.sin_family = AF_INET;
    cliaddr.sin_addr.s_addr = INADDR_ANY;
    cliaddr.sin_port = htons(ECHO_PORT);
    if ((ret = bind(sockfd, (struct sockaddr *) &cliaddr,
sizeof(cliaddr))) < 0)
    {
        perror("ERROR on binding");
        goto failed;
    }

    do
    {
        len = sizeof(cliaddr);
        if((ret = recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct
sockaddr *)&cliaddr, &len)) > 0)
        {
            printf("Recv from %s\r\n", inet_ntoa(cliaddr.sin_addr));
            ret = sendto(sockfd, buffer, ret, 0, (struct sockaddr
*)&cliaddr, len);
        }
    }while(ret >= 0);
failed:
    close(sockfd);
    return 0;
}
```

程序首先通过指定函数 socket()创建一个 UDP 套接字，在调用 bind()之前我们首先调用 setseckopt()设置 SO_REUSEADDR 套接字选项，该选项避免端口被其他程序占用而导致调用 bind()失败，关于 setsockopt()的调用我们在套接字选项一节讲述。

在调用 bind()之后，程序进入一个简单的循环，它首先调用 recvfrom()读到达的 UDP 数据报，并用 sendt0()将它发送回客户端。尽管这个程序很简单，但也有许多细节问题需要考虑：

首先，除非 I/O 调用出错否则不会终止循环，因为 UDP 是一个无连接协议，所以 recvfrom()返回 0 并不代表连接关闭。

其次，该程序提供一个迭代服务器(iterative server)而不是类似 TCP 的并发服务器，它没有调用 fork()，所以在单一服务进程中处理了所有客户的请求。一般来说，大多数 TCP 服务程序是并发的，而大多数 UDP 服务器是迭代的。

每个 UDP 套接字都有一个接收缓冲区，到达此套接字的每个数据报都进入此套接字接收缓冲区。当进程调用 recvfrom()时，缓冲区中的下一个数据报以先入先出顺序返回给进程。如果进程可以读取数据报之前有多个数据报到达，那么到达的数据将添加到套接字接收缓冲区中。但这个缓冲区的大小有一个限制。我们在套接字选项一节中讨论 SO_RCVBUF 选项并讨论如何增大它。

使用 UDP 改写 echo 客户端程序：

源代码：client.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <errno.h>

#define RET_OK   0
#define RET_ERR -1

#define LISTEN_QUEUE_NUM 5

#define BUFFER_SIZE 256

#define ECHO_PORT 2029


int main(int argc, char *argv[])
{
    int sockfd, ret = RET_OK;
```

```
    struct sockaddr_in servaddr, recvaddr;
    struct hostent *server;

    char buffer[BUFFER_SIZE];

    if (argc < 2) {
        fprintf(stderr,"usage %s hostname\n", argv[0]);
        return RET_ERR;
    }
    if((server = gethostbyname(argv[1])) == NULL)
    {
        herror("gethostbyname. ");
        return RET_ERR;
    }

    if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("ERROR opening socket");
        return RET_ERR;
    }
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = *(uint32_t *)server->h_addr;
    servaddr.sin_port = htons((uint16_t)ECHO_PORT);

    while(1)
    {
        printf("Enter the message  : ");
        if(fgets(buffer, sizeof(buffer) - 1, stdin) == NULL)
        {
            break;
        }
        if((ret = sendto(sockfd, buffer ,strlen(buffer), 0, (struct
sockaddr *)&servaddr, sizeof(servaddr))) < 0)
        {
            perror("ERROR writing to socket");
            break;
        }
```

```
        uint32_t  len = sizeof(recvaddr);
retry:
        if((ret = recvfrom(sockfd, buffer, sizeof(buffer) - 1, 0, (struct
sockaddr *)&recvaddr, &len)) < 0)
        {
            perror("ERROR reading from socket");
            break;
        }
        if(servaddr.sin_family != recvaddr.sin_family ||
           servaddr.sin_addr.s_addr != recvaddr.sin_addr.s_addr||
           servaddr.sin_port != recvaddr.sin_port)
        {
            fprintf(stderr, "Received %s echo\r\n",
inet_ntoa(recvaddr.sin_addr));
            goto retry;
        }
        buffer[ret] = 0;
        printf("Server echo message: %s\n",buffer);
    }
    close(sockfd);
    return ret < 0 ? RET_ERR : RET_OK;
}
```

客户端用服务器的 IP 地址和端口号来装填一个 IPv4 的套接字地址结构。循环发送用户输入的数据，循环中有四个步骤：

用 fgets()从标准输入读一行。

用 sendto()将此行发给服务器。

用 recvfrom()读回服务器的回射。

用 printf()输出回射数据至标准输出。

客户端第一次发送请求时内核将给套接字分配一个空闲的临时端口。(对于 TCP 客户，我们说 connect()调用是分配发生的地方)对于 UDP 套接字，如果进程首次调用 sendto()时还没有捆绑(调用 bind())上一个本地端口，内核就在此时为套接字选择一个临时端口，这个端口在以后的通讯中将不改变，直至关闭这个套接字。客户端也可以显式地调用 bind()为自己指定一个端口和地址，但很少这样做。

在 recvfrom()的调用中，我们查看数据报发送者的地址，通过与服务器的地址和端口号对比，丢弃那些不是来自服务器的应答。但是这不是一个好的解决办法，我们将在下一节中讨论给 UDP 套接字调用 connect()。

UDP 连接

除非为 UDP 套接字调用 connect()，否则异步错误是不会返回到 UDP 套接字的。实际上，我们可以给 UDP 套接字调用 connect()，但这样做的结果与 TCP 连接不相同：没有三次握手过程。内核只是记录对方的 IP 地址和端口号，它们包含在传递给 connect()的套接字地址结构中，connect()函数会立即返回而不像 TCP 套接字要等待连接完成才返回。

对于已调用 connect()的 UDP 套接字，与缺省的 UDP 套接字相比，发生了三个变化：

- ✓ 我们再也不能给I/O操作指定同的IP地地和端口号。也就是说，我们不使用sento()，而使用write()或send()。写到已连接UDP套接字上的任何东西都自动发送到调用connect()时指定的协议地址(IP地址和端口号)。与TCP相似，我们可以给已调用connect()的UDP套接字调用sento()，但不能指定目的地址。sendto() 的第五个参数(指向套接字地址结构的指针)必须为空指针。第六个参数(套接字地址结构的大小)应为0。

不再使用recvfrom()而使用read()或recv()读取数据报。内核只会给recvfrom()返回那些来自调用 connect()时指定协议地址的数据报，也就是说调用 connect()后的套接字只能和调用 connecto()指定的地址和端口进行 UDP 通讯。

异步错误会发送给已调用 connect()的 UDP 套接字，由此推断，未调用 connect()的 UDP 套接字不接受任何异步错误，这些错误信息主要由 IP 层生成，并通过 ICMP 控制报文来通知。

对于已连接 UDP 套接字,进程可给那个套接字再次调用 connect()以达到下面两个目的：

- ✓ 指定新的IP地址和端口号；
- ✓ 断开套接字。

第一种情况即给已连接 UDP 的套接字指定新的对方主机地址，与 TCP 套接字不同：TCP 套接字只允许调用一次 connect()。

为了断开已连接的 UDP 套接字，我们调用 connect()，但设置套接字地址结构的地址族为 AF_UNSPEC。这可能会导致 connect()返回一个 EAFNOSUPPORT 错误(errno 中)。

下.面我们修改客户端代码，给 UDP 套接字调用 connect()，并不再检查数据报的发送者地址和端口：

示例 5-6： 为 UDP 套接字调用 connect()的 echo 客户端

源代码：client.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
```

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

#define RET_OK   0
#define RET_ERR -1

#define LISTEN_QUEUE_NUM 5

#define BUFFER_SIZE 256

#define ECHO_PORT 2029


int main(int argc, char *argv[])
{
    int sockfd, ret = RET_OK;
    struct sockaddr_in servaddr;
    struct hostent *server;

    char buffer[BUFFER_SIZE];

    if (argc < 2) {
        fprintf(stderr,"usage %s hostname\n", argv[0]);
        return RET_ERR;
    }
    if((server = gethostbyname(argv[1])) == NULL)
    {
        herror("gethostbyname. ");
        return RET_ERR;
    }

    if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("ERROR opening socket");
        return RET_ERR;
    }
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
```

```
    servaddr.sin_addr.s_addr = *(uint32_t *)server->h_addr;
    servaddr.sin_port = htons((uint16_t)ECHO_PORT);
    if(connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr))
< 0)
    {
        perror("ERROR connect");
        goto failed;
    }

    while(1)
    {
        printf("Enter the message  : ");
        if(fgets(buffer, sizeof(buffer) - 1, stdin) == NULL)
        {
            break;
        }
        if((ret = write(sockfd, buffer ,strlen(buffer))) < 0)
        {
            perror("ERROR writing to socket");
            break;
        }
        if((ret = read(sockfd, buffer, sizeof(buffer) - 1)) < 0)
        {
            perror("ERROR reading from socket");
            break;
        }
        buffer[ret] = 0;
        printf("Server echo message: %s\n",buffer);
    }
failed:
    close(sockfd);
    return ret < 0 ? RET_ERR : RET_OK;
}
```

源代码：server.c

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

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <errno.h>

#define RET_OK   0
#define RET_ERR -1

#define LISTEN_QUEUE_NUM 5

#define BUFFER_SIZE 256

#define ECHO_PORT 2029

int main(int argc, char **argv)
{
    int sockfd, opt = 1;
    uint32_t len;
    struct sockaddr_in cliaddr;
    uint8_t buffer[BUFFER_SIZE];
    int ret = RET_OK;

    if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("ERROR opening socket");
        return RET_ERR;
    }
    if((ret = setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt,
sizeof(opt))) < 0)
    {
        perror("ERROR setsockopt");
        goto failed;
    }
```

```
    memset(&cliaddr, 0, sizeof(cliaddr));
    cliaddr.sin_family = AF_INET;
    cliaddr.sin_addr.s_addr = INADDR_ANY;
    cliaddr.sin_port = htons(ECHO_PORT);
    if ((ret = bind(sockfd, (struct sockaddr *) &cliaddr,
sizeof(cliaddr))) < 0)
    {
        perror("ERROR on binding");
        goto failed;
    }

    do
    {
        len = sizeof(cliaddr);
        if((ret = recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct
sockaddr *)&cliaddr, &len)) > 0)
        {
            printf("Recv from %s\r\n", inet_ntoa(cliaddr.sin_addr));
            ret = sendto(sockfd, buffer, ret, 0, (struct sockaddr
*)&cliaddr, len);
        }
    }while(ret >= 0);
failed:
    close(sockfd);
    return 0;
}
```

上面的代码中为UDP套接字调用connect()时指定对方的IP地址和端口号，然后使用read()和write()来与对方交换数据。来自其他lP地址和端口的数据报将不递送给已连接套接字，所以我们不必再比较端口号和IP地址。

UDP编程小结

作为小结，仅在进程用UDP套接字与确定的唯一对方进行通信时，UDP客户或服务器才可以调用connect()。一般来说，都是UDP客户调用connect()，但也有UDP服务器与单个客户长时间通信的应用程序(如TFTP)，这种情况下，客户和服务器都可调用connect()。

我们的UDP客户-服务器例子是不可靠的。如果一个客户数据报丢失(譬如说，被客户与服务器间的某路由器丢弃)，客户将永远阻塞在对recvfrom()的调用，等待一个永远不会到

达的服务器应答。与此相似，如果客户数据报到达服务器，但服务器的应答丢失了，客户也将永远阻塞于 recvfrom()的调用。上述问题唯一的解决方法就是给客户端 recvfrom()调用设置一个超时(如信号或套接字选项)。

仅仅是为调用 recvfrom()而设置超时并不是一个完整的办法。例如，如果我们超时了，我们无法辨别超时原因是数据报没有到达服务器，还是服务器的应答没有回到客户。如果客户的请求有点像“从帐户 A 往账户 B 传递固定数目的钱”而不是我们的 echo 例子，则请求的丢失和应答的丢失都是非常严重的事情，所以我们还要实现应用层确认和超时重传机制，最终我们发现我们的 UDP 程序越来越像 TCP 了，那么我们为什么不干脆用 TCP 呢？没错，在很多场合是不适合用 UDP 的，但 UDP 并非一无是处，它提供了广播和多播的能力，另外对于那些不在乎少量数据丢失的视频和音频流服务(如视频会议)，UDP 更适合一些。

我们的 ECHO 客户端程序有个严重的问题，因为 UDP 是不可靠的传输层协议，所以我们不能保证发送的数据被对方收到，所以当我们发送的数据丢失，或来自服务器的应答丢失，ECHO 客户端将永远阻塞在 recvfrom()调用上。我希望读者能够通过某种方法来解决这个问题。

5.3 套接字选项

在学习 UDP 广播之前，有必要先简单学习一下套接字选项，套接字选项用于修饰套接字以及其底层通讯协议的各种行为。函数 setsockopt()和 getsockopt()可以查看和设置套接字的各种选项，让我们先看一下.这两个函数的原型：

```
#include<sys/socket.h>
int getsockopt(int sockfd,int level,int optname,void *optval,socklen_t *optlen);
int sersockopt(int sockfd,int level,int optname,const void *optval,socklen_t optlen);
```

getsockopto 和 setsockopto 调用成功返回 0，失败则返回-1 并设置 errno 错误条件。

参数 sockfd 必须是调用 socket()函数创建的套接字。

参数 level 指明选项所属的层次，它描述选项是作用于 TCP/IP 代码的套接字层还是传输层或是网络层(套接字层是 TCP/IP 代码的一部分，用与处理应用程序的套接字请求)，level 允许的值可以是 SOL_SOCKET，IPPROTO_IP，IPPROTO_TCP 等值。我们这里只涉及 SOL_SOCKET(套接字层)的情况，level 的其他取值超出了木书的范围。

optname 参数指明是哪一个选项，表 5-4 列出了套接字层(level 取值为 SOL_SOCKET)允许的部分选项：

表5-4套接字选项

optname	描述	数据类型	getsockopt	setsockopt
SO_BROADCAST	允许发送广播数据报	int	●	●

SO_REUSEADDR	允许重用本地址	int	●	●
SO_LINGER	如果有未确认的数据，是否延迟关闭套接字	struct linger	●	●
SO_RCVBUF	套接字接收缓冲区大小	Int	●	●
SO_SNDBUF	套接字发送缓冲区大小	Int	●	●

optval 参数是指向一个对象的指针，该对象用来存放由 setsockopto 设置或被 getsockopt()获取的选项值，对象的数据类型视特定的选项而定，大部分选项使用 int 型。

optval 参数是指 l 幻一个对象的指针,该对象用来存放 setsockop 设置或被 getsockop 状取的选项值，对象的数据类型视特定的选项 Inj 定，大部分选项使用 int 型。

对于 getsockopto，optlen 是值-结果参数，他指向一个 socklen_t 类型的变量，该变量给出 optval 所指对象的数据类型包含的字节数并返回实际获得的选项值的字节大小。对于 setsockopto，optlen 指明 optval 所指向的数据类型包含的字节数。

getsockopto()函数获取套接字 sockfd 在层次 level 上的选项 optname 的值。此选项的值存放在 optval 所指的缓冲区中，再调用之前，我们应当提供此缓冲区包含的字节数放入 optlen 指向的整数中。返回时，它包含实际存储在此缓冲区内数据的字节数。

函数 setsockopt 设置套接字在层次 level 上的选项 optname。选项的值传递到 optval 指向的对象内，对象占用的字节数为 optlen。

下面对表 5-4 列出的部分选项作进一步说明，它们均定义在头文件<sys/socket.h>中。我们这里仅列出其中的一部分，目的是使读者对套接字选项有一个大致的了解，如果读者有兴趣深入了解，可以参看 W.Richard Stevens 的《UNIX socket programming》。

SO_BROADCAST

SO_BROADCAST 选项控制着 UDP 套接字是否能够发送广播数据报，选项的类型为 int，非零意味着“是”，注意，只有 UDP 套接字可以使用这个选项，TCP 是不能使用户播的。

另一个重要的限制是：在大多数系统中，要求必须那些具有 root 权限的用户才能发送广播数据报。

下面代码给一个 UDP 套接字启用户播功能：

```
int opt=1;
if((sockfd=socket(AF_INET,SOCK_DGRAM, 0))<0)
{
  / /错误处理
}
if(setsockopt(sockfd, SOL_SOCKET,so_BROADCAST,&opt, sizeof(opt))<
0)
{
  / /错误处理
```

```
}
```

SO_REUSEADDR

服务器端通常要调用 bind()来给套接字赋予一个本地地址和端口，如果这个端口已经被另外一个进程使用将导致 bind()调用失败，但是如果在服务器 bind()操作前对套接字设置 SO_REUSEADDR 选项，这个操作可能成功。之所以用可能而不是肯定的原因是：TCP 不允许两个进程调用 bind()捆绑相同的地址和端口，也就是说如果两个进程使用相同的端口，但是 bind()不同的本地地址(如一个使用地址 INADDR_ANY(0.0.0.0)，而另外一个使用地址 127.0.0.1)这可以通过后调用 bind()的应用程序设置 SO_REUSEADDR 选项来实现。所以 SO_REUSEADDR 对于服务器来说是个非常重要的选项。如果选项的值设置为非零则允许重复 bind()同一端口，否则不允许。

下面代码演示如何设置如何打开 SO_REUSEADDR 选项：

```
int opt=1;
if((sockfd=socket(AF_INET,SOCK_DGRAM, 0))<0)
{
  / /错误处理
}
if(setsockopt(sockfd, SOL_SOCKET,so_BROADCAST,&opt, sizeof(opt))<
0)
{
  / /错误处理
}
```

SO_LINGER

SO_LINGER 选项指定函数 close()对于 TCP 协议如果操作，缺省是如果有数据残留在套接字发送缓冲区内，系统将尝试这些数据发送并得到确认后 close()才返回。SO_LINGER 选项可以修改这一行为。

SO_LINGER 使用 linger 结构，它定义在 sys/socket.h 中：

```
#include<sys/socket. h>
struct linger
{
int l_onoff;    /*0=关闭，非零=打开*/
int l_linger;    /*等待时间(秒)*/
);
```

对于 setsockopto，我们分别根据 linger 结构的两个域的取值讨论以下三种情况：
如果 l_onoff 为 0，则关闭选项。l_linger 的值被忽略。close()使用 TCP 的默认行为。
如果 l_onoff 为非 0，而 l linger 为 0，TCP 将丢弃套接字缓冲区内未发送和未确认的数据，直接发送一个 RST 给对方，而不是从前讨论过的 TCP 关闭序列(四个终止序列)。同时 TCP 也不再经历 TIME_WAIT 状态，这么做有些危险，因为 TIME_WAIT 状态是 TCP 避免残留在网络

中的“旧的连接”上的数据不正确的被立即启动的“新连接”收到而设置的一种延时保护机制。但有时这种方式还是很有用的，比如我们想在老的服务器垮掉的时候立即重新启动它，如果要经历 TIME_WAIT 状态，我们新启动的服务器的 bind()操作将失败，这种情况一直到等到 TIME_WAIT 状态结束。

如果 l_onoff 为非 0 且 l_linger 也非 0，那么 close()关闭套接字时可能要拖延一段时间。也就是说如果套接字发送缓冲|区中仍然残留有数据，close()将等待这些数据被发送完毕并得到对方 TCP 层确认后或超时(时间为 l_linger 中设置的秒数)才返回。

下面代码演示如何设置 SO_LINGER 选项来避免 TCP 的 TIME_WAIT 状态：

```
struct linger so_linger;
if((sockfd=socket(AF_INET,SOCK_STREAM, 0))<0)
{
  //错误处理
}
so—linger.l_onoff = 1;
so_linger.l_linger = 0;
if(setsockopt(sockfd,  SOL_SOCKET,  SO_LINGER,  &so_linger,
sizeof(so_iinger))<0)
{
   //错误处理
}
```

SO_SNDBUF 和 SO_RCVBUF

每一个套接字有一个发送缓冲区和一个接收缓冲区，这两个缓冲区由底层协议使用，接收缓冲区存放由协议接收的数据直到被应用程序读走，发送缓冲区存放应用写出的数据直到被协议发送出去。SO_SNDBUF 和 SO_RCVBUF 选项分别控制发送和接收缓冲区的大小，它们的类型为 int，以字节为单位。

下面的代码演示使用 SO_SNDBUF 选项将套接字发送缓冲区增加 2048 字节：

```
int opt;     .
if((sockfd=socket(AF_IN ET,SOC_STREAM, 0))<0)
{
  //错误处理
}
if(getsockopt(sockfd, SOL_SOCKET,  SO_SNDBUF,&opt,sizeof(opt)< 0)
{
   //错误处理
}
0pt +=2048;
if(setsockopt(sockfd,SOL_SOCKER,SO_SNDBUF,&opt,sizeof(opt0))< 0)
{
```

```
        //错误处理
    }
```

本章总结

在这一章的开始我们简单叙述了 LINUX 编程中常见的 5 种 I/O 模型：

- ✓ 阻塞式I/O
- ✓ 非阻塞式I/O
- ✓ I/O复用(select()和poll())

在这 3 种模型中，阻塞式 I/O 和 I/O 复用最为常用，阻塞式 I/O 上一章我们已经用到，所以我们本章中着重讲解了 I/O 复用的编程方法(select()和 poll()的使用)。I/O 复用使我们的多客户端服务程序变得更简单，但是有几个值得注意的地方，使用 selecto()时三个需要注意的要点：

- ✓ 当使用select()时，两个最常见的编程错误是：忘了对最大文件描述符加l和忘了文件描述符集是值-结果参数，select()返回时会将那些没“准备好”的套接字对应的位置为0，所以如果要再次调用select()时，一定要重新用FD_SET设置你感兴趣的文件描述符的对应位。
- ✓ 对于一个套接字，如果该套接字关闭或出错，select()将返回该套接字既可读又可写，所以对于 select()返回读操作“准备好”状态不能只认为它仅仅是可读，我们还要检查 read()或 write()的返回值判断是否该连接已经关闭。
- ✓ 对于监听套接字(对其调用了 listen()的文件描述符)，select()将返回读操作“准备好”表示该套接字上有新的连接到来或套接字出错，我们可以调用 accept()分这两种情况，如果 accept()返回非负数表示有新的连接，并且该返回值为新已连接套接字，如果 accept()返回-1 表示套接字出错(如套接字已关闭)。

在 UDP 编程一节中，我们介绍了 UDP 特性及其典型应用：

- ✓ UDP 是无连接的、不可靠的数据协议报，而 TCP 是面向连接的，提供可靠的字节流。然而，有些情况下更适合用 UDP 而不是 TCP。有些流行的应用程序是 UDP 实现的：DNS(域名系统)、NFS(网络文件系统)和 SNMP(简单网络管理协议)就是这样的例子。另外 UDP 还提供了广播和多播的特性，这是 TCP 法做到的。

在 UDP 连接一节，我们叙述了为一个 UDP 套接字调用 connect()给我们带来的好处：

- ✓ 我们不用给 I/O 操作的写函数指定同的 IP 地址和端口号。也就是说，我们不使用 sendto()，而使用 write()或 send()。写到已连接 UDP 套接字上的任何东西都自动发送到调用 connect()时指定的协议地址(IP 地址和端口号)。与 TCP 相似，我们可以给已调用 connect()的 UDP 套接字调用 sento()，但不能指定目的地址。Sendto()的第五个参数(指向套接字地址结构的指针)必须为空指针。第六个参数(套接字地址结构的大小)应为 0。
- ✓ 不再使用 recvfrom()而使用 read()或 recv()读取数据报。内核只会给 recvfrom()返回那些来自调用 connect()时指定协议地址的数据报，也就是说调用 connect()后的套接字只能和调用 connecto()时指定的地址和端口进行 UDP 通讯。
- ✓ 异步错误会发送给已调用 connect()的 UDP 套接字，由此推断，未调用 connect()的 UDP

套接字不接受任何异步错误，这些错误信息主要由 IP 层生成，并通 ICMP 控制报文来通知。

UDP 广播一节我们讲述了广播的特点以及编程方法：

- ✓ 在创建 UDP 套接字后我们紧接着调用 setsockopt()给套接字设置 SO_BROADCAST 选项，opt 的值是 1，这将允许该 UDP 文件描述符发送广播数据报。然后向广播地址发送 UDP 数据报就可以了。

由于课时的原因我们这章并没有介绍多播的使用，多播是 UDP 协议一个非常有用的特性，希望读者利用课余时间学习它。

在套接字选项一节中，我们讲解了几个非常有用的套接字选项以及如何使用 getsockopt()和 setsockopt()获取和设置它们的值。

- ✓ SO_BROADCAST　　允许发送广播数据报
- ✓ SO_REUSEADDR　　允许重用本地端口号
- ✓ SO_LINGER　　如果套接字缓冲区中有未确认的数据，是否延迟关闭套接字
- ✓ SO_RCVBUF　　套接字接收缓冲区大小
- ✓ SO_SNDBUF　　套接字发送缓冲区大小

这几个套接字选项非常重要，它能够帮助我们避免一些奇怪的网络编程问题，如在一个服务程序退出后无法立即启动新的服务器应用实例(bind()调用失败，SO_LINGER 选项)，或无法同时启用服务器应用的多个实例(bind()调用失败，SO_REUSEADDR 选项)。

在守护进程(daemon)和超级服务器(inetd)一节中我们详细讲述了守护进程的编写：

- ✓ 守护进程类似于 DOS 下的内存驻留程序，脱离控制终端进入后台执行。比如我们的网络服务程序，可以在完成创建套接字(socket())，捆绑套接字(bind())，设置套接字为监听模式(listen())后，变成守护进程进入后台执行不占用控制终端，这是网络服务程序的常用模式。
- ✓ 因为守护进程没有控制终端，所以它不能打印信息到终端上(也就是说我们不能调用 fprintf()或 printf()函数输出信息)。守护进程为记录事件或错误日志一般会　通过调用 syslog()连接 syslogd 守护进程来实现。

第六章 课程设计

6.1 远程终端管理系统

6.1.1 平台开发和环境简介

Linux 系统 + Gcc + Gdb + makefile

6.1.2 功能描述

远程终端管理系统实现了一个 mini telnet 的功能。该系统采用 C/S 架构，需要分别编写服务端与客户端程序。

其中，服务端采用 I/O 复用机制，实现多客户端连接。通过读取配置文件，服务端将初始化服务端的设置和建立用户信息。在客户认证登陆后，服务端动态维护实时用户的会话信息，接收客户端的命令，将执行结果返回给客户端并显示。

6.1.3 设计实施

一、跟踪调试

1. 加入 4 级日志函数,完成日志函数的定义。

日志：日志记录系统所发生的事件，并将文本信息保存至相应的日志文件。

```
log_info(format...);

log_warning(format...);

log_error(format...);

log_fatal_error(format...);
```

其中，log_info 函数打印普通信息；log_warning 函数打印警告信息；log_error 函数打印普通错误信息；log_fatal_error 函数打印严重错误信息。

打印信息的默认级别为 info，也可通过参数的传入决定打印级别。打印信息写入固定日志文件，或通过参数控制打印到标准输出。

日志文件的基本格式：时间、谁、发生什么事件。

2. 调试跟踪函数

```
debug(format...);
```

trace();

其中，debug 函数是用来调试的函数，可以通过宏定义在编译是去掉。trace 函数是跟踪函数没有参数，打印当前的文件名、函数名、行号、时间、日期。

二、 Sock 编程

1、基于 tcp 协议的客户端服务端通讯。

2、服务端使用 select I/O 复用机制，支持多客户端连接。

3、客户端从终端接受命令，把命令发送给服务端。服务端执行命令，并把执行结果返回给客户端。例如：在客户端终端键入“ls”，服务端将执行结果显示在客户端。

4、服务端使用链表记录当前客户端的会话连接，并动态维护会话。基本信息：客户端的 ip 地址、端口、服务端使用 sock 描述符、客户端连接时间、收发数据包的个数、数据包出现错误的个数、接收客户端的命令条数。

三、 配置文件

服务端的配置从配置文件中获取，不再以启动参数的形式传入。

1、配置内容：服务端的使用的 ip 地址和端口

2、最大的客户连接数

3、用户名和用户密码

4、日志文件保存的路径

5、日志的输出级别

四、 用户认证

服务端和用户端增加用户密码登陆机制

1、在服务端配置文件中对客户端用户密码进行配置，服务启动后从配置文件中读取用户信息形成数据表。

2、在建立连接时，客户端把接受的用户名密码发送至服务端，服务验证后返回认证结果给客户端。认证正确开始接受命令，认证错误重新认证。

五、 协议

根据通信数据的类型，重新设计应用协议。将上述客户端与服务端之间的通信数据，以协议的形式进行封装。

六、心跳机制

客户端与服务端之间使用心跳机制。心跳机制：客户端定时向服务端发送一个数据包（心跳包），证明自己活着。服务端超过一定的时间没有收到服务端的心跳包则说明客户端出现问题，做出相应的处理（一般处理记录状况，并且断开连接）。

七、守护进程

守护进程：一种后台运行独立于所有终端控制之外的进程。此处，编写守护进程，作为“sock 编程”一节中服务端进程的父进程。

1、守护进程中重定向标准输入、输出、错误输出的文件描述符到指定文件。

2、守护进程负责服务进程的关闭。

3、当收到停止服务信号时，守护进程杀掉服务进程，并且以后不再重新启动。

故增加信号的处理：停止服务器，信号值为 SIGUSER1/SIGUSER2。

守护进程采用定时器方式监控服务进程。定时查看服务进程（/proc/pid/status）的状态，并作出相应的处理。另外，如果服务异常退出，则重新启动。

八、 服务端处理机制

1、 改用多进程处理客户端的命令。

2、改用多线程处理客户端的命令。

6.1.4 项目要求

1、采用 C 语言完成代码的编写。

2、编写 makefile 管理整个项目。

3、编写项目设计书。

4、以模块化编写项目代码，按照不同模块组织.h/.c 文件。

5、规范代码格式并添加注释。

6、编写测试报告，包括单模块测试，模块间测试。

7、编写项目总结，包括项目设计说明、项目中采用的知识点列举、项目中遇到的问题及解决方法等。

6.1.5 项目完成参考步骤

可以按一下步骤，以每个步骤作为一个目标，一步一步完成项目。

1、完成服务端与客户端 1 对 1 的 tcp 连接

2、实现服务端与客户端 1 对多的 tcp 连接，并且服务端建立在线用户列表。
3、实现客户端从终端接受命令，并发完服务端，服务端接受数据。
4、服务端处理接受到的数据，执行命令，返回命令的执行结果。

6.2 局域网 OICQ 程序设计

6.2.1 平台开发和环境简介

Linux 系统 + Gcc + Gdb + makefile

6.2.2 功能描述

实现局域网 OICQ 程序设计，包括客户端和服务端。

客户端描述：客户端运行开始出现登陆界面。与服务端进行连接，连接后把账号信息发送给服务端，服务端验证后，把确认结果通知客户端。如果通过验证，客户端从服务端接收其他在线客户端信息并把这些客户信息显示给用户。用户可以选择客户并与之进行信息交流。即发送消息和接受消息。并把结果显示给用户。

服务端功能描述：服务端启动后，等待客户端连接。接受客户端发送过来的账号信息。进行验证。并把验证的结果返回给客户端。如果验证通过，记录客户端的信息。并把该客户端的记录的信息发送给其他的客户端，也把其他的在线用户发送给该用户，实现整个网络内的在线客户信息的同步。接受客户端的断开连接的请求。服务端断开连接，删除记录并把结果发送给其他的客户端。

程序设计提示：

客户端与服务端时间使用 tcp 协议进行通讯，客户端和客户端之间使用 upd 协议进行通讯。

服务端监听一个 tcp 端口，接受客户端的连接。每个客户端使用 udp 协议监听一个端口。接受来自其他的客户端的信息。

服务端启动后，使用 tcp 协议监听 8088 端口。等待客户端的连接。每接受一个连接把创建一个节点，把节点保存到一个链表（在线用户列表）中。包含客户端的 ip、tcp 源端口、用户名、客户端监听的 udp 端口（该端口作为客户端与其他的客户端通讯的端口）、登录时间等信息。把新增加的用户信息发送给当前在线的其他用户。如果有用户请求退出，或者 tcp 连接断开，则从列表中删除该用户并把该信息发送给局域网的其他的用户。做到整个局域网内的在线用户的信息是同步的。

客户端启动后，提示属于账号信息，然后连接服务端。连接成功后，使用 udp 协议监听某端口，该端口作为接受来自其他的客户端消息的端口。监听成功后。把客户端信息发送给服务端，包括：客户端的用户名称，udp 协议监听的端口等信息。

6.2.3 设计实施

一、配置文件读取

服务端程序启动后读取配置文件，配置文件中包含要监听使用的 IP、端口等信息。
配置文件格式可以参考/etc/目录下的各种服务的配置文件。
配置文件一般规则：

1、在配置文件中每一行是作为一条配置；
2、每条配置有两个项：配置项名称（配置变量名）、配置项值；
3、“#”之后的内容为注释；
4、配置文件命名：xxx.conf

二、Sock 编程

根据配置文件信息启动服务端程序，监听端口，等待客户端连接。完成客户端于服务端简单的 tcp 连接。使用 I/O 复用机制完成客户端与服务端之间的一对多的连接。服务端记录每个客户端的基本信息：每个客户端的 IP、端口等基本信息。使用链表记录保存这些信息。

三、数据包协议

客户端使用 TCP 连接服务端后，使用 UDP 协议监听一个端口。并把用户名称信息和监听端口的信息发送给服务端。发送接受数据使用数据封包。封包格式如下：
| 协议版本（1） | 数据包类型（1）| 数据包的长度（2）| 数据包的内容（变长）|

协议版本：发送端填充自己的该数据包的版本信息，服务端接受数据包后，查看该版本是否是自己能识别的版本。是则进行解析，否则作为非法数据包（一般丢弃）。

数据包类型：发送端根据自己发送的数据包里面的数据内容的不同填充不同的类型。

如：如果数据包的数据信息是登录信息，类型为 0x0001。如果是断开连接的信息则类型为 0x0002。这样接收端接受到数据之后就可以根据数据类型的值，对数据做不同的处理。

数据包的长度：TCP 是数据流协议，数据发送者多次发送的数据，但在 tcp 的接受端缓冲区内数据与数据之间没有间隔。无法分开。所以在数据包中增加一项数据长度。

这样接收端可以根据数据长度信息确定本数据包的长度，确定要从 tcp 的缓冲区内每次要读取的长度。

数据包内容：要发送的数据。

数据包发送者：在发送数据前，在数据前增加数据包头。数据包头包含以上的包头信息。封包可以采用数据结构：

```
struct pack_head
{
    unsigned char ver;
    unsigned char type;
    unsigned short len;
    char    buf[0];//buf 指针作为数据包的缓冲的包头。
};
```

数据包接受端：接受到数据后，分成两次读取一个数据包，第一次读取首先读取一个数据包头长度，然后根据数据包头中的数据长度读取整个数据包。这样一个一个的数据包就分别被读取出来。

数据包协议总结：协议是数据的收发端之间一种约定要好的一种规定。发送者按照该格式发送，接受者按照该格式进行解析。

四、数据同步

服务端接受新的客户端的，把其他的所有在线客户的信息（及在线客户表）发送至新登录的客户端。
提示：

1、客户端每隔一定的时间发送请求包，请求同步数据，服务端收到请求包后向客户端发送在线用户列表。客户端接收列表，完成接收后，更新本地的在线用户表。

2、服务端发送这些信息，可以分成多个数据包，每个数据包中存放一定个数的客户端数据信息。最后发送一个结束的数据包。告诉客户端所有客户端信息发送结束。如果是该数据包，数据类型可以为 0x0003。在数据内容的第一项为数据包中所存存放的客户端信息的个数。如果所有的客户端信息发送完成后可以发送一个数据，类型为 0x0003，里面客户端的个数为 0。（如果前期可以把所有的数据放在一个数据包中发送给客户端，不做结束数据包）

1、客户端登陆后，接收到其他的客户端信息后，显示这些客户端信息。并打印用户操作菜单：“1、发送信息。2、接收消息。3、退出”。

如果选择发送信息操作。提示用户输入要发送目标客户的名称。然后提示用户输入要发送数据信息的具体内容。输入回车代表信息数据完毕。根据客户名称在当前在线的客户端的列表（服务端发送过来的）中查找改用户，并根据该用户的信息，即此用户使用 udp 协议监听的端口。把数据封包后使用 udp 协议发送给该客户端。

2、客户端接收到信息后，根据发送者的 IP 和用户信息，提示用户接收到了来自某客户的信息。并把接收到信息显示。并提示是否要回复。
3、守护进程：把服务端改造成守护进程。守护进程定义：运行与后台脱离终端的程序。使用 ps –ef 命令可以查看当前系统中的很多守护进程。

五、心跳机制

客户端每隔一定时间想服务端发送一个数据包，该数据包用途主要使用证明客户端存在，网络没有断开，客户端程序本身也没有出现任何问题。就像证明自己活着一样。如果客户端每隔 3 秒发送一个心跳包给客户端。如果服务端超过 10 秒钟没有接收到客户端的心跳包，则认为该客户端已经死亡（网络断开或者客户端程序出现问题）。

提示：可以在客户端使用 I/O 复用机制，使用 I/O 复用中的超时机制进行定时。

6.2.4 项目要求

1、采用 C 语言完成代码的编写。
2、编写 makefile 管理整个项目。
3、编写项目设计书。
4、以模块化编写项目代码，按照不同模块组织 .h/.c 文件。
5、规范代码格式并添加注释。
6、编写测试报告，包括单模块测试，模块间测试。
7、编写项目总结，包括项目设计说明、项目中采用的知识点列举、项目中遇到的问题及解决方法等。

6.2.5 项目完成参考步骤

可以按一下步骤，以每个步骤作为一个目标，一步一步完成项目。
1、完成服务端与客户端 1 对 1 的 tcp 连接
2、实现服务端与客户端 1 对多的 tcp 连接，并且服务端建立在线用户列表。可以使用数组或者链表保存。服务端每当接受客户端连接时保存客户端的相关信息至在线用户列表。当连接断开后把该客户端从列表中删除。
3、实现在线用户注册、登陆。
4、实现在线用户表的同步：让每个客户端可以得到服务端维护的在线用户表。
5、客户端与客户端使用 udp 协议完成消息收发操作。
6、实现日志、配置文件、心跳、守护进程等功能。

6.3 GPS 车载系统课程设计

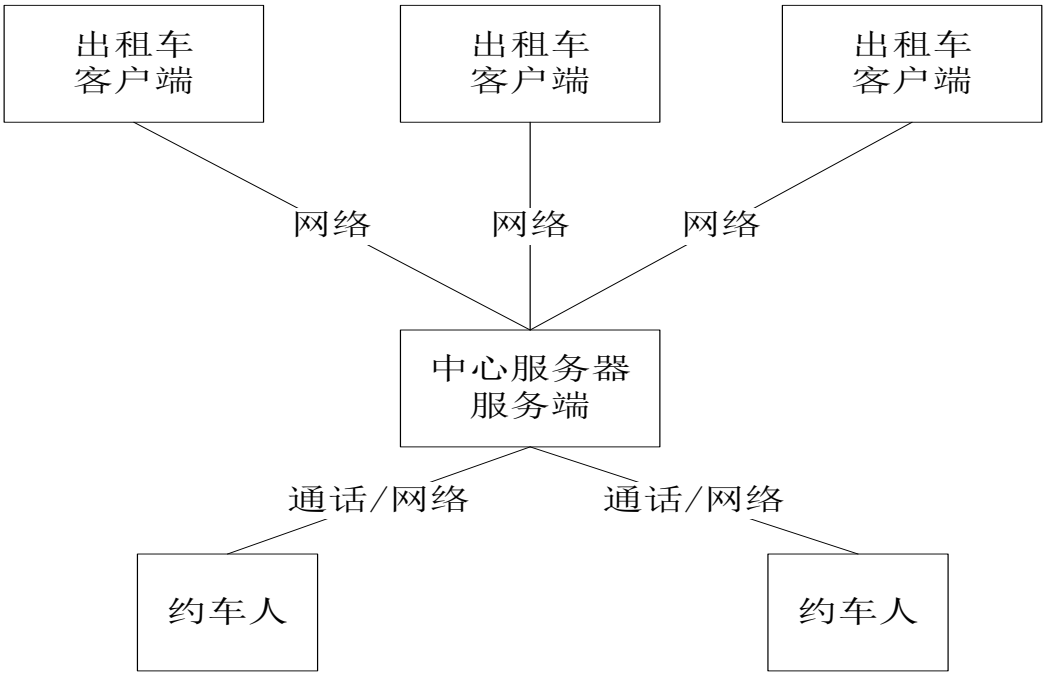
6.3.1 平台开发和环境简介

Linux 系统 + Gcc + Gdb + makefile

6.3.2 应用背景介绍和功能描述

背景：GPS 应用背景介绍

GPS（global positioning system 全球定位系统）车载终端是安装在出租车上的一款集导航，调度，娱乐，远程刷卡，电话等为一体的智能移动终端设备。有了 GPS 车载终端（客户端）和调度中心后台（服务端），可以最优化驾驶员和乘客的出车和求车的资源配置。



功能介绍：

该项目分为客户端、服务端、网络预约端程序。出租车客户端系统嵌入出租车车载系统内，服务端系统运行在中心服务器上。网络预约端程序运行在任意 PC 机上。约车人通过网络跟中心服务系统联系，预约用车。服务中心根据获取的出租车位置信息通知出租车是否接受该业务。

实现客户端：

1、出租车客户端软件开发，基本功能如下：

A、客户端与中心服务端建立 TCP 网络连接

B、客户端登陆服务端，签到

C、客户端上传驾驶员信息给服务端

D、客户端定期发送自己位置信息给服务端

E、接受、上传业务信息。中心通知出租车附近有人叫车。该出租车接受或者拒绝该业务。

F、驾驶员退出。

服务端：

A、监听 TCP 连接

B、对客户端签到信息进行认证

C、接受客户端上传的驾驶员信息。

D、接受客户的预约业务信息，并处理。

E、接受出租车信息注册、并保存到文件长期记录。

F、出租车登陆后维护当前在线连接。

G、接受出租车得位置信息，并保存。

当客户通过网络预约出租车后，服务端根据经纬度寻找里预约客户最近的出租车（没有乘客的），询问是否接受该业务，如果接受该业务则通知客户并告示车牌号等信息。如果不接受则重新寻找新的距离进的出租车。

网络预约端程序：

仿真客户使用地图预约改程序，该程序与服务端建立网络连接，建立后，用户可以提出业务请求，输入用户经纬度，并发送到服务端，服务端。服务端相应后可以想信息会送显示给预约端。

6.3.3 设计实施

1. 模块功能

本项目为模拟 GPS 车载系统调度应用，模拟车载终端和调度中心分别为客户端和服务器端。注意：本项目有些应用无法模拟的地方均作了简化。另外，文中提到的通信格式（协议）通俗讲是一种封装，就像收发信件需要信封一样。每条协议都是有加 header 头的，一是考虑安全，二是考虑你这条协议是干什么的，这样 socket 通信中的网络数据互不干扰，各自处理。客户端和服务器端模型均采用 linux + select IO 复用实现。

2.通讯格式和消息类型

2.1 通讯格式(一切皆协议，请做项目的学生仔细研究协议)

数据包封装格式如下：head 占两个字节；数据包长度占 2 个字节；消息体占 n 个字节。

Head	0
	1
长度[注 1]	2
	3
消息体（body）	4
	..
	n

注 1：长度= 消息内容长度 低位在前 (小端字节)

2.2 车载终端发起消息类型

车载 GPS 终端系统向中心服务端发送的数据的包的 FLAG（即数据包的类型）有如下几种：

head	消息类型	介绍
1000	参数读取	请求参数读取，数据保存在服务器本地
1001	参数设置	受限操作，必须驾驶员登录后才能操作
1002	驾驶员注册	简单模拟，检查数据库信息，建立链表遍历成功后，写数据到数据库
1003	驾驶员登录签到	同注册，遍历比较，登陆成功后返回 IP 和 port
1004	驾驶员登出签退	驾驶员状态为登出
1005	请求驾驶员相片 url 信息	受限操作，必须驾驶员登录后才能操作
1006	url 请求下载驾驶员相片	1005 请求成功后自动发送，不需手动
1007	心跳包	客户端每 5 秒发一次心跳包，服务器子线程每 5 秒扫描一次在线用户链表，超过 60 次（5 分钟），可判断客户

		端已死亡，并从链表中删除节点，服务器主线程收到客户端心跳包后清零此客户端的 timers

客户端本地处理 1 秒的定位信息:本意为客户端每秒检查 GPS 串口和车辆参数，现在简单模拟为客户端线程每秒读取文本数据，接着解析 GPS 数据显示年、月、日、星期、方位，速度等数据。简单模拟不处理车辆参数。

另外客户端有了定位和车辆参数信息，每隔一段时间会向调度中心发送心跳包汇报情况，比如空车重车速度报警等情况，本项目简单模拟处理心跳包：只发送头，不发送消息体。

2.3 调度中心发起的消息类型

调度中心（即中心服务端）向车载系统发送的数据包 head(即数据包类型)有如下几种：

head	类型	附
1008	业务信息处理	服务器端 60 秒定时自动发送调度信息，客户端接受数据解析后显示业务 ID,要车时间，信息内容）

3.消息体定义

3.1 参数读取

车载终端发送(一次读取 3 个)

参数 ID	0
参数 ID	1
参数 ID[注 1]	2

调度中心返回（一次返回 3 个参数内容）[注 2]

参数 ID	0
参数长度（n）	1

参数内容	2~n
参数 ID	n+1
同上[注 3]	...

注 1:当车载终端读取参数时一次读取 3 个参数，index=1,2,3

注 2:不定长度的计算

注 3:有 3 个参数以此类推

可读取的参数列表,

参数 ID 含义：

1=允许拨出电话（MAX16 字节）

2=调度中心刷卡服务 IP（4 字节）

3=调度中心刷卡服务端口号（2 个字节）

3.2 参数设置

车载终端发送:

参数 ID	0
参数长度（n）[注 1]	1
参数数据	2~n

调度中心返回，成功后需要改写数据库

结果码[1-成功, 0-失败]	0

参数 ID：1=允许拨出电话（MAX16 字节），字符串保存即可

2=调度中心刷卡服务 IP（4 字节）,字符串保存即可

3=调度中心刷卡服务端口号（2 个字节），字符串保存即可

[注 1]：不定长度

注意：简单模拟为所有客户端的车辆参数设置都一模一样，数据库无需分别保存。

3.3 驾驶员注册

车载终端发送

驾驶员账号(10)	0
	1
	.
	.
	.
	9
驾驶员密码(10)	10
	.
	.
	.
	19

调度中心返回（成功则写到数据库中（模拟用配置文件））

结果码[1-注册成功，0-注册失败]	0

3.4 驾驶员登录签到

车载终端发送

驾驶员账号(10)	0
	1
	.
	.
	.
	9
驾驶员密码(10)	10
	.

	.
	.
	19

调度中心返回

结果码[1-签到成功，0-签到失败]	0

3.5 驾驶员签退

车载终端发送

消息内容空

调度中心返回，(注意签退后不能下载相片了，故需要记录状态,请做好测试工作)

结果码[1-签退成功，0-签退失败]	0

3.6 请求驾驶员相片 url 信息

车载终端发送

消息内容空

调度中心返回(模拟返回 driverpic.jpg)测试选 1 张相片即可

相片 URL 数据 (n=后面 URL 字节长度) （http://image.baidu.com/driverpic.jpg）	0
	1
	2
	.
	.
	.

3.7 url 请求下载驾驶员相片

（注意：3.5 返回后 3.6 自动请求，不需要手动选择，是个自动请求的过程）

车载终端发送

driverpic.jpg url 数据	0
	1
	2
	.
	.

调度中心返回(模拟下载 driverpic.jpg)

FLAG[注 1]	0
长度[注 2]	1
	2
Driverpic.jpg buffer 数据	3
	4
	5
	.
	.

[注 1]:FLAG=0,后面还有包发送，FLAG=1，发送到最后一包.客户端每接收到的数据 append 保存成到图片文件中。

[注 2]：1000bytes 一包，最后一包可能小于或等于 1000 个字节

做简单模拟：

客户端接收到服务器返回的数据 buffer 后，append 写到一个空的 driverpic.jpg 下。最大的图片文件的大小不超过(1000bytes).

3.8 一秒的定位信息

（模拟来自美国全球卫星 GPS 定位数据）

客户端处理（模拟线程每秒解析数据，处理成北京时间，显示年月日时分秒星期)

参考数据：$GPRMC,100119.999,A,2236.8226,N,11403.7299,E,0.62,120.87,220506,,**（详解见下文）

3.8.1 GPS 解析知识

提取定位数据

GPS 接收机只要处于工作状态就会源源不断地接收 GPS 导航定位信息。把数据放入缓存发送到车载终端进程处理，在没有进一步处理之前缓存中是一长串字节流，这些信息在没有经过分类提取之前是无法加以利用的。因此，必须通过程序将各个字段的信息从缓存字节流中提取出来，将其转化成有实际意义的。同其他通讯协议类似，对 GPS 进行信息提取必须首先明确其帧结构，然后才能根据其结构完成对各定位信息的提取。在本文中，其接受的数据主要由帧头、帧尾和帧内数据组成，根据数据帧的不同，帧头也不相同，主要有"$GPGGA"、"$GPGSA"、"$GPGSV"以及"$GPRMC"等。这些帧头标识了后续帧内数据的组成结构，各帧均以回车符和换行符（0X0D、0X0A）作为帧尾标识一帧的结束。对于通常的情况，我们所关心的定位数据如经纬度、速度、时间等均可以从"$GPRMC"帧中获取得到，该帧的结构及各字段释义如下，

数据丰富的最典型情况，均为 ASCII 字符，可以通过 buf-0x30 转换为数字。

$GPRMC,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,<10>,<11>*hh

<1> 当前位置的格林尼治 UTC 时间，格式为 hhmmss.sss

<2> 状态, A 为有效位置, V 为非有效接收警告，即当前天线视野上方的卫星个数少于 3 颗。A 为车辆已经定位，V 为没有定位

<3> 纬度, 格式为 ddmm.mmmm 格式不定长，例如：3111.4364

<4> 标明南北半球, N 为北半球、S 为南半球

<5> 经度，格式为 dddmm.mmmm 格式不定长，例如：12125.1027

<6> 标明东西半球，E 为东半球、W 为西半球

<7> 地面上的速度，范围为 0.0 到 999.9

<8> 方位角，范围为 000.0 到 359.9 度

<9> 日期, 格式为 ddmmyy 需要通过+2000 转换

<10> 地磁变化，从 000.0 到 180.0 度(不考虑)

<11> 地磁变化方向，为 E 或 W（不考虑）

数据缺失的最典型情况将是：

$GPRMC,<1>,<2>,,,,,,,,*hh

<1> 当前位置的格林尼治时间，格式为 hhmmss.sss

<2> 状态, A 为有效位置, V 为非有效接收警告，即当前天线视野上方的卫星个数少于 3 颗。

数据将会出项缺失，时间回复到出厂时间，定位情况为 V，（不定位警告）。

至于其他几种帧格式，除了特殊用途外，平时并不常用，虽然接收机也在源源不断

地向主机发送各种数据帧，但在处理时一般先通过对帧头的判断而只对"$GPRMC"帧进行数据的提取处理。如果情况特殊，需要从其他帧获取数据，处理方法与之也是完全类似的。由于帧内各数据段由逗号分割，因此在处理缓存数据时一般是通过搜寻 ASCII 码"$"来判断是否是帧头，在对帧头的类别进行识别后再通过对所经历逗号个数的计数来判断出当前正在处理的是哪一种定位导航参数，并作出相应的处理。

例如某数据：

$GPRMC,100119.999,A,2236.8226,N,11403.7299,E,0.62,120.87,220506,,**

'100119.999 --时分秒（格林威治时间）

'2236.8226,N --北纬坐标（2236.8226）

2236.8226,S ----南纬坐标

'11403.7299,E --东经坐标（11403.7299）

11403.7299,W ----西经坐标

'0.62 --gps 的移动速度

'120.87 --地面的方位角

'220506 --日月年 06+2000=2006 年

//参考 GPS 数据结构，存储 ascii 字符串

typedef struct _GPSData

{

char date[15] ; //Gps 数据日期

char time[15] ; //Gps 数据时间

char latitude_type; //纬度类型，北纬，南纬

char latitude[15] ; //纬度值

char longitude_type; //经度类型，东经，西经

char longitude[15] ;//经度值

char speed[6];//速度

char cog[10];//方位角

char IsValid;//是否定位标志

}GPSData;

到此为止，已将时间和经纬度信息提取到GPS结构数组GPSData中的各个变量中去，后续的处理可根据该结构中存储的数据作出相应的处理。

比如：1、请继续判断，是否定位；

2、根据方位角判断目前车辆定位在是正东，正西，正南，正北，东南，东北，西南，西北；

3、请对GPS格林尼治时间（世界时间）进行+8转化为北京时间，提示：请进行平年闰年月份等方面考虑。闰年一年366天，平年365天，闰年在2月份有29天，平年2月份28天。闰年特点：1、能被4整除并且不能被100整除。2、或者是能被400直接整除。

4、显示出星期几，提示：可以以2000年1月1日星期六为参考起点，计算总天数。

3.9 业务处理信息（模拟调度信息发送）[注 1]

调度中心发送:[注 2]

业务 ID[注 3]	0
	1
	2
	3
要车时间[年，月，日，时，分，秒][注 4]	4
	5
	6
	7
	8
	9
内容长度	10
信息内容	11

	…
	n

车载终端不做返回（收到即保存到文本中，显示业务 ID,要车时间，信息内容）

[注 1]：要求服务器端起一个线程，每 60 秒自动广播发送业务给客户端。业务数据根据格式定义。

[注 2]：前提，客户端必须在线而且登陆状态，即服务器获取了登陆的用户名才能发送，否则不发送。

[注 3]：小端保存 4 个字节的方式可以参考，也可以采取移位的方式处理：

```
int a = 0x12345678;

unsigned char b0 = a%(256*256*256);//0x78

unsigned char b8 = (a%(256*256))/256;//0x56

unsigned char b16 = (a/(256*256))%256;//0x34

unsigned char b24 = a/(256*256*256);//0x12
```

[注 4]：year 客户端需要加 2000，服务端-减去 2000

4.Sock 编程

1、基于 tcp 协议的客户端服务端通讯。

2、客户端和服务端均使用 select I/O 复用机制，均关注 socket 文件描述符，支持多客户端连接。

3、客户端和服务端进行数据交互，均采用协议的方式处理。

4、服务端使用链表记录（单链表即可）当前客户端的会话连接，并动态维护会话。

5.配置文件

本文使用配置文件(.ini.dat.cfg.txt 等)作为数据库，存放用户名和密码信息，存放参数信息。

6.用户认证

服务端和客户端需要用户名密码登陆机制。客户端输入账号密码后，服务器从配置文件中读取已认证的合法用户信息，读入链表中进行比较。

7.心跳机制

客户端与服务端之间使用心跳机制。心跳机制：客户端定时向服务端发送一个数据包（心跳包），证明自己活着，也可以汇报状况。服务器超过一定的时间没有收到服务端的心跳包则说明客户端出现问题（可能出现了客户端死机现象），做出相应的处理（一般处理记录状态，并且断开连接）。

8.多线程处理

客户端和服务端可能均可能有多线程处理。

6.3.4 项目要求

1、采用 C 语言完成代码的编写。

2、分别给客户端和服务端编写 makefile 管理整个项目

3、编写项目设计书。

4、以模块化编写项目代码，按照不同模块组织.h/.c 文件

5、规范代码格式并添加注释，善用小函数，函数不宜过长。

6、单元测试，每个函数都必须测试好，尽量减小 bug。

7、项目中遇到的问题和解决方法，项目总结写心得体会。

6.4 微信语音聊天系统

6.4.1 平台开发和环境简介

Linux 系统 + Gcc + Gdb + makefile

6.4.2 功能描述

实现语音微信系统聊天功能。项目分客户端服务端。服务端完成接受注册登录功能，完成在线用户列表动态维护及其同步给网络内所有客户端的功能。其功能与 OICQ 功能相同。客户端与客户端之间通过传输语音数据。完成微信传输。

6.4.3 设计实现

基本功能实现与 OICQ 相同。

语音程序：分为音频录制和音频播放。音频录制和播放可以调用 asound 库函数实现。