Linux 系统编程

传智播客-邢文鹏[1]

2014-09-27

[1]http://blog.csdn.net/itcastcpp

前言

学习目标

成为linux/unix系统程序员

学习态度

* 谦虚
* 严谨
* 勤思
* 善问

学习方法

从本章开始学习各种Linux系统函数，这些函数的用法必须结合Linux内核的工作原理来理解，因为系统函数正是内核提供给应用程序的接口，而要理解内核的工作原理，必须熟练掌握C语言，因为内核也是用C语言写的，我们在描述内核工作原理时必然要用“指针”、“结构体”、“链表”这些名词来组织语言，就像只有掌握了英语才能看懂英文书一样，只有学好了C语言才能看懂我描述的内核工作原理。读者看到这里应该已经熟练掌握了C语言了，所以应该有一个很好的起点了。我们在介绍C标准库时并不试图把所有库函数讲一遍，而是通过介绍一部分常用函数让读者把握库函数的基本用法，在掌握了方法之后，书上没讲的库函数读者应该自己查Man Page学会使用。同样，本书的第三部分也并不试图把所有的系统函数讲一遍，而是通过介绍一部分系统函数让读者理解操作系统各部分的工作原理，在有了这个基础之后就应该能够看懂Man Page学习其它系统函数的用法。

读者可以结合[APUE2e]学习本书的第三部分，该书在讲解系统函数方面更加全面，但对于内核工作原理涉及得不够深入，而且假定读者具有一定的操作系统基础知识，所以并不适合初学者。该书还有一点非常不适合初学者，作者不辞劳苦，在N多种UNIX系统上做了实验，分析了它们的内核代码，把每个系统函数在各种UNIX系统上的不兼容特性总结得非常详细，很多开发者需要编写可移植的应用程序，一定爱死他了，但初学者看了大段大段的这种描述（某某函数在4.2BSD上怎么样，到4.4BSD又改成怎么样了，在SVR4上怎么样，到Solaris又改成怎么样了，现在POSIX标准是怎么统一的，还有哪些系统没有完全遵守POSIX标准）只会一头雾水，不看倒还明白，越看越不明白了。也正因为该书要兼顾各种UNIX系统，所以没法深入讲解内核的工作原理，因为每种UNIX系统的内核都不一样。而本书的侧重点则不同，只讲Linux平台的特性，只讲Linux内核的工作原理，涉及体系结构时只讲x86平台，对于初学者来说，绑定到一个明确的平台上学习就不会觉得太抽象了。当然本书的代码也会尽量兼顾可移植性，避免依赖于Linux平台特有的一些特性。

只听不练肯定学不会Linux，每个知识点都需要去动手实践

目录

第 1 章

文件I/O

1.1 C标准函数与系统函数的区别

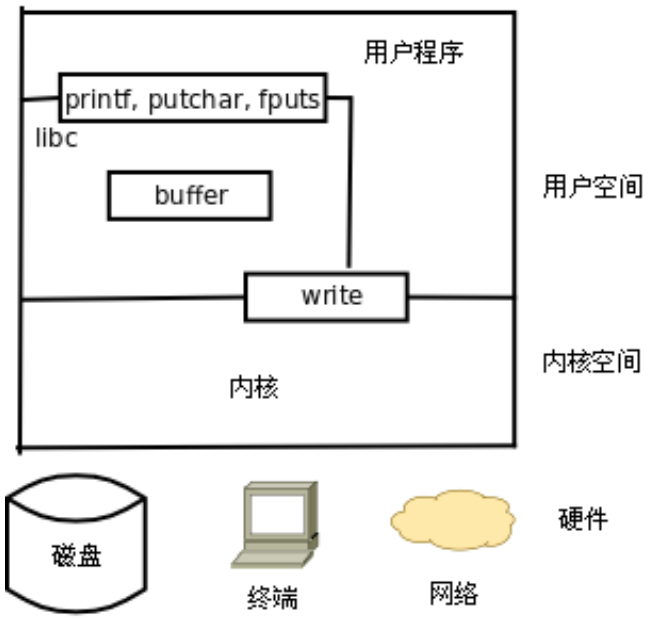

图 1.1: 带缓冲区的c标准函数

1.1.1 I/O缓冲区

每一个FILE文件流都有一个缓冲区buffer，默认大小8192Byte。

1.1.2 效率

1.1.3 程序的跨平台性

事实上Unbuffered I/O这个名词是有些误导的，虽然write系统调用位于C标准库I/O缓冲区的底层，但在write的底层也可以分配一个内核I/O缓冲区，所以write也不一定是直接写到文件的，也可能写到内核I/O缓冲区中，至于究竟写到了文件中还是内核缓冲区中对于进程来说是没有差别的，如果进程A和进程B打开同一文件，进程A写到内核I/O缓冲区中的数据从进程B也能读到，而C标准库的I/O缓冲区则不具有这一特性（想一想为什么）。

1.2 PCB概念

1.2.1 task_struct结构体

/usr/src/linux-headers/include/linux/sched.h

1.2.2 files_struct结构体

1.3 open/close

O_CREAT	创建文件
O_EXCL	创建文件时，如果文件存在则出错返回
O_TRUNC	把文件截断成0
O_RDONLY	只读 （互斥）
O_WRONLY	只写 （互斥）
O_RDWR	读写 （互斥）
O_APPEND	追加，移动文件读写指针位置到文件末尾
O_NONBLOCK	非阻塞标志
O_SYNC	使每次write都等到物理I/O操作完成，包括文件属性的更新

图 1.2: open

1.3.1 文件描述符

一个进程默认打开3个文件描述符

```
STDIN_FILENO 0
STDOUT_FILENO 1
STDERR_FILENO 2
```

新打开文件返回文件描述符表中未使用的最小文件描述符。

open函数可以打开或创建一个文件。

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
返回值：成功返回新分配的文件描述符，出错返回-1并设置errno
```

在Man Page中open函数有两种形式，一种带两个参数，一种带三个参数，其实在C代码中open函数是这样声明的：

```
int open(const char *pathname, int flags, ...);
```

最后的可变参数可以是0个或1个，由flags参数中的标志位决定，见下面的详细说明。

pathname参数是要打开或创建的文件名，和fopen一样，pathname既可以是相对路径也可以是绝对路径。flags参数有一系列常数值可供选择，可以同时选择多个常数用按位或运算符连接起来，所以这些常数的宏定义都以O_开头，表示or。

必选项：以下三个常数中必须指定一个，且仅允许指定一个。

* O_RDONLY 只读打开
* O_WRONLY 只写打开
* O_RDWR 可读可写打开

以下可选项可以同时指定0个或多个，和必选项按位或起来作为flags参数。可选项有很多，这里只介绍一部分，其它选项可参考open(2)的Man Page：

* O_APPEND 表示追加。如果文件已有内容，这次打开文件所写的数据附加到文件的末尾而不覆盖原来的内容。
* O_CREAT 若此文件不存在则创建它。使用此选项时需要提供第三个参数mode，表示该文件的访问权限。
* O_EXCL 如果同时指定了O_CREAT，并且文件已存在，则出错返回。
* O_TRUNC 如果文件已存在，并且以只写或可读可写方式打开，则将其长度截断（Truncate）为0字节。
* O_NONBLOCK 对于设备文件，以O_NONBLOCK方式打开可以做非阻塞I/O（Nonblock I/O），非阻塞I/O在下一节详细讲解。

注意open函数与C标准I/O库的fopen函数有些细微的区别：

以可写的方式fopen一个文件时，如果文件不存在会自动创建，而open一个文件时必须明确指定O_CREAT才会创建文件，否则文件不存在就出错返回。

以w或w+方式fopen一个文件时，如果文件已存在就截断为0字节，而open一个文件时必须明确指定O_TRUNC才会截断文件，否则直接在原来的数据上改写。

第三个参数mode指定文件权限，可以用八进制数表示，比如0644表示-rw-r--r--，也可以用S_IRUSR、S_IWUSR等宏定义按位或起来表示，详见open(2)的Man Page。要注意的是，文件权限由open的mode参数和当前进程的umask掩码共同决定。

补充说明一下Shell的umask命令。Shell进程的umask掩码可以用umask命令查看：

```
$ umask
0002
```

用touch命令创建一个文件时，创建权限是0666，而touch进程继承了Shell进程的umask掩码，所以最终的文件权限是0666&~022=0644。

```
$ touch file123
$ ls -l file123
-rw-rw-r--  1 xingwenpeng xingwenpeng          0  9月 11 23:48 file123
```

同样道理，用gcc编译生成一个可执行文件时，创建权限是0777，而最终的文件权限是0777 & ~022 = 0755。

```
xingwenpeng@ubuntu:~$ umask
0002
xingwenpeng@ubuntu:~$ gcc main.c
xingwenpeng@ubuntu:~$ ls -l a.out
-rwxrwxr-x 1 xingwenpeng xingwenpeng 7158  9月 11 23:51 a.out
```

我们看到的都是被umask掩码修改之后的权限，那么如何证明touch或gcc创建文件的权限本来应该是0666和0777呢？我们可以把Shell进程的umask改成0，再重复上述实验：

```
$ rm file123 a.out
$ umask 0
$ touch file123
$ ls -l file123
-rw-rw-rw-  1 xingwenpeng xingwenpeng          0  9月 11 23:52 file123
$ gcc main.c
$ ls -l a.out
-rwxrwxr-x 1 xingwenpeng xingwenpeng 7158  9月 11 23:52 a.out
```

现在我们自己写一个程序，在其中调用open(“somefile”, O_WRONLY | O_CREAT, 0664);创建文件，然后在Shell中运行并查看结果：

close函数关闭一个已打开的文件：

```
#include <unistd.h>

int close(int fd);
返回值：成功返回0, 出错返回-1并设置errno
```

参数fd是要关闭的文件描述符。需要说明的是，当一个进程终止时，内核对该进程所有尚未关闭的文件描述符调用close关闭，所以即使用户程序不调用close，在终止时内核也会自动关闭它打开的所有文件。但是对于一个长年累月运行的程序（比如网络服务器），打开的文件描述符一定要记得关闭，否则随着打开的文件越来越多，会占用大量文件描述符和系统资源。

由open返回的文件描述符一定是该进程尚未使用的最小描述符。由于程序启动时自动打开文件描述符0、1、2，因此第一次调用open打开文件通常会返回描述符3，再调用open就会返回4。可以利用这一点在标准输入、标准输出或标准错误输出上打开一个新文件，实现重定向的功能。例如，首先调用close关闭文件描述符1，然后调用open打开一个常规文件，则一定会返回文件描述符1，这时候标准输出就不再是终端，而是一个常规文件了，再调用printf就不会打印到屏幕上，而是写到这个文件中了。后面要讲的dup2函数提供了另外一种办法在指定的文件描述符上打开文件。

1.3.2 最大打开文件个数

查看当前系统允许打开最大文件个数

```
cat /proc/sys/fs/file-max
```

当前默认设置最大打开文件个数1024

```
ulimit -a
```

修改默认设置最大打开文件个数为4096

```
ulimit -n 4096
```

1.4 read/write

read函数从打开的设备或文件中读取数据。

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
返回值：成功返回读取的字节数，出错返回-1并设置errno，如果在调read之前已到达文件末尾，则这次read返回0
```

参数count是请求读取的字节数，读上来的数据保存在缓冲区buf中，同时文件的当前读写位置向后移。注意这个读写位置和使用C标准I/O库时的读写位置有可能不同，这个读写位置是记在内核中的，而使用C标准I/O库时的读写位置是用户空间I/O缓冲区中的位置。比如用fgetc读一个字节，fgetc有可能从内核中预读1024个字节到I/O缓冲区中，再返回第一个字节，这时该文件在内核中记录的读写位置是1024，而在FILE结构体中记录的读写位置是1。注意返回值类型是ssize_t，表示有符号的size_t，这样既可以返回正的字节数、0（表示到达文件末尾）也可以返回负值-1（表示出错）。read函数返回时，返回值说明了buf中前多少个字节是刚读上来的。有些情况下，实际读到的字节数（返回值）会小于请求读的字节数count，例如：

读常规文件时，在读到count个字节之前已到达文件末尾。例如，距文件末尾还有30个字节而请求读100个字节，则read返回30，下次read将返回0。

从终端设备读，通常以行为单位，读到换行符就返回了。

从网络读，根据不同的传输层协议和内核缓存机制，返回值可能小于请求的字节数，后面socket编程部分会详细讲解。

write函数向打开的设备或文件中写数据。

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
返回值：成功返回写入的字节数，出错返回-1并设置errno
```

写常规文件时，write的返回值通常等于请求写的字节数count，而向终端设备或网络写则不一定。

1.5 阻塞和非阻塞

读常规文件是不会阻塞的，不管读多少字节，read一定会在有限的时间内返回。从终端设备或网络读则不一定，如果从终端输入的数据没有换行符，调用read读终端设备就会阻塞，如果网络上没有接收到数据包，调用read从网络读就会阻塞，至于会阻塞多长时间也是不确定的，如果一直没有数据到达就一直阻塞在那里。同样，写常规文件是不会阻塞的，而向终端设备或网络写则不一定。

现在明确一下阻塞（Block）这个概念。当进程调用一个阻塞的系统函数时，该进程被置于睡眠（Sleep）状态，这时内核调度其它进程运行，直到该进程等待的事件发生了（比如网络上接收到数据包，或者调用sleep指定的睡眠时间到了）它才有可能继续运行。与睡眠状态相对的是运行（Running）状态，在Linux内核中，处于运行状态的进程分为两种情况：

正在被调度执行。CPU处于该进程的上下文环境中，程序计数器（eip）里保存着该进程的指令地址，通用寄存器里保存着该进程运算过程的中间结果，正在执行该进程的指令，正在读写该进程的地址空间。

就绪状态。该进程不需要等待什么事件发生，随时都可以执行，但CPU暂时还在执行另一个进程，所以该进程在一个就绪队列中等待被内核调度。系统中可能同时有多个就绪的进程，那么该调度谁执行呢？内核的调度算法是基于优先级和时间片的，而且会根据每个进程的运行情况动态调整它的优先级和时间片，让每个进程都能比较公平地得到机会执行，同时要兼顾用户体验，不能让和用户交互的进程响应太慢。

下面这个小程序从终端读数据再写回终端。

1.5.1 阻塞读终端

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

int main(void)
{
    char buf[10];
    int n;
    n = read(STDIN_FILENO, buf, 10);
    if (n < 0) {
        perror("read STDIN_FILENO");
        exit(1);
    }
    write(STDOUT_FILENO, buf, n);
    return 0;
}
```

执行结果如下：

```
$ ./a.out
hello (回车)
hello
$ ./a.out
hello world (回车)
hello worl$ d
bash: d: command not found
```

第一次执行a.out的结果很正常，而第二次执行的过程有点特殊，现在分析一下：

Shell进程创建a.out进程，a.out进程开始执行，而Shell进程睡眠等待a.out进程退出。

a.out调用read时睡眠等待，直到终端设备输入了换行符才从read返回，read只读走10个字符，剩下的字符仍然保存在内核的终端设备输入缓冲区中。

a.out进程打印并退出，这时Shell进程恢复运行，Shell继续从终端读取用户输入的命令，于是读走了终端设备输入缓冲区中剩下的字符d和换行符，把它当成一条命令解释执行，结果发现执行不了，没有d这个命令。

如果在open一个设备时指定了O_NONBLOCK标志，read/write就不会阻塞。以read为例，如果设备暂时没有数据可读就返回-1，同时置errno为EWOULDBLOCK（或者EAGAIN，这两个宏定义的值相同），表示本来应该阻塞在这里（would block，虚拟语气），事实上并没有阻塞而是直接返回错误，调用者应该试着再读一次（again）。这种行为方式称为轮询（Poll），调用者只是查询一下，而不是阻塞在这里死等，这样可以同时监视多个设备：

```
while(1) {
    非阻塞read(设备1);
    if(设备1有数据到达)
        处理数据;
    非阻塞read(设备2);
    if(设备2有数据到达)
        处理数据;
    ...
}
```

如果read(设备1)是阻塞的，那么只要设备1没有数据到达就会一直阻塞在设备1的read调用上，即使设备2有数据到达也不能处理，使用非阻塞I/O就可以避免设备2得不到及时处理。

非阻塞I/O有一个缺点，如果所有设备都一直没有数据到达，调用者需要反复查询做无用功，如果阻塞在那里，操作系统可以调度别的进程执行，就不会做无用功了。在使用非阻塞I/O时，通常不会在一个while循环中一直不停地查询（这称为Tight Loop），而是每延迟等待一会儿来查询一下，以免做太多无用功，在延迟等待的时候可以调度其它进程执行。

```
while(1) {
    非阻塞read(设备1);
    if(设备1有数据到达)
        处理数据;
    非阻塞read(设备2);
    if(设备2有数据到达)
```

```
        处理数据;
    ...
    sleep(n);
}
```

这样做的问题是，设备1有数据到达时可能不能及时处理，最长需延迟n秒才能处理，而且反复查询还是做了很多无用功。以后要学习的select(2)函数可以阻塞地同时监视多个设备，还可以设定阻塞等待的超时时间，从而圆满地解决了这个问题。

以下是一个非阻塞I/O的例子。目前我们学过的可能引起阻塞的设备只有终端，所以我们用终端来做这个实验。程序开始执行时在0、1、2文件描述符上自动打开的文件就是终端，但是没有O_NONBLOCK标志。所以就像例 28.2 “阻塞读终端”一样，读标准输入是阻塞的。我们可以重新打开一遍设备文件/dev/tty（表示当前终端），在打开时指定O_NONBLOCK标志。

1.5.2 非阻塞读终端

```
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

#define MSG_TRY "try again\n"

int main(void)
{
    char buf[10];
    int fd, n;
    fd = open("/dev/tty", O_RDONLY|O_NONBLOCK);
    if(fd<0) {
        perror("open /dev/tty");
        exit(1);
    }
tryagain:
    n = read(fd, buf, 10);
    if (n < 0) {
        if (errno == EAGAIN) {
            sleep(1);
            write(STDOUT_FILENO, MSG_TRY, strlen(MSG_TRY));
            goto tryagain;
        }
        perror("read /dev/tty");
        exit(1);
    }
    write(STDOUT_FILENO, buf, n);
    close(fd);
    return 0;
}
```

以下是用非阻塞I/O实现等待超时的例子。既保证了超时退出的逻辑又保证了有数据到达时处理延迟较小。

1.5.3 非阻塞读终端和等待超时

```
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

#define MSG_TRY "try again\n"
#define MSG_TIMEOUT "timeout\n"

int main(void)
{
    char buf[10];
    int fd, n, i;
    fd = open("/dev/tty", O_RDONLY|O_NONBLOCK);
    if(fd<0) {
        perror("open /dev/tty");
        exit(1);
    }
    for(i=0; i<5; i++) {
        n = read(fd, buf, 10);
        if(n>=0)
            break;
        if(errno!=EAGAIN) {
            perror("read /dev/tty");
            exit(1);
        }
        sleep(1);
        write(STDOUT_FILENO, MSG_TRY, strlen(MSG_TRY));
    }
    if(i==5)
        write(STDOUT_FILENO, MSG_TIMEOUT, strlen(MSG_TIMEOUT));
    else
        write(STDOUT_FILENO, buf, n);
    close(fd);
    return 0;
}
```

1.6 lseek

每个打开的文件都记录着当前读写位置，打开文件时读写位置是0，表示文件开头，通常读写多少个字节就会将读写位置往后移多少个字节。但是有一个例外，如果以O_APPEND方式打开，每次写操作都会在文件末尾追加数据，然后将读写位置移到新的文件末尾。lseek和标准I/O库的fseek函数类似，可以移动当前读写位置（或者叫偏移量）。

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

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

参数offset和whence的含义和fseek函数完全相同。只不过第一个参数换成了文件描述符。和fseek一样，偏移量允许超过文件末尾，这种情况下对该文件的下一次写操作将延长文件，中间空洞的部分读出来都是0。

若lseek成功执行，则返回新的偏移量，因此可用以下方法确定一个打开文件的当前偏移量：

```
off_t currpos;
currpos = lseek(fd, 0, SEEK_CUR);
```

这种方法也可用来确定文件或设备是否可以设置偏移量，常规文件都可以设置偏移量，而设备一般是不可以设置偏移量的。如果设备不支持lseek，则lseek返回-1，并将errno设置为ESPIPE。注意fseek和lseek在返回值上有细微的差别，fseek成功时返回0失败时返回-1，要返回当前偏移量需调用ftell，而lseek成功时返回当前偏移量失败时返回-1。

1.7 fcntl

先前我们以read终端设备为例介绍了非阻塞I/O，为什么我们不直接对STDIN_FILENO做非阻塞read，而要重新open一遍/dev/tty呢？因为STDIN_FILENO在程序启动时已经被自动打开了，而我们需要在调用open时指定O_NONBLOCK标志。这里介绍另外一种办法，可以用fcntl函数改变一个已打开的文件的属性，可以重新设置读、写、追加、非阻塞等标志（这些标志称为File Status Flag），而不必重新open文件。

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

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

这个函数和open一样，也是用可变参数实现的，可变参数的类型和个数取决于前面的cmd参数。下面的例子使用F_GETFL和F_SETFL这两种fcntl命令改变STDIN_FILENO的属性，加上O_NONBLOCK选项，实现和例 28.3 “非阻塞读终端”同样的功能。

1.7.1 用fcntl改变File Status Flag

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

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

#define MSG_TRY "try again\n"

int main(void)
{
    char buf[10];
    int n;
    int flags;
    flags = fcntl(STDIN_FILENO, F_GETFL);
    flags |= O_NONBLOCK;
    if (fcntl(STDIN_FILENO, F_SETFL, flags) == -1) {
        perror("fcntl");
        exit(1);
    }
tryagain:
    n = read(STDIN_FILENO, buf, 10);
    if (n < 0) {
        if (errno == EAGAIN) {
            sleep(1);
            write(STDOUT_FILENO, MSG_TRY, strlen(MSG_TRY));
            goto tryagain;
        }
        perror("read stdin");
        exit(1);
    }
    write(STDOUT_FILENO, buf, n);
    return 0;
}
```

1.8 ioctl

ioctl用于向设备发控制和配置命令，有些命令也需要读写一些数据，但这些数据是不能用read/write读写的，称为Out-of-band数据。也就是说，read/write读写的数据是in-band数据，是I/O操作的主体，而ioctl命令传送的是控制信息，其中的数据是辅助的数据。例如，在串口线上收发数据通过read/write操作，而串口的波特率、校验位、停止位通过ioctl设置，A/D转换的结果通过read读取，而A/D转换的精度和工作频率通过ioctl设置。

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

int ioctl(int d, int request, ...);
```

d是某个设备的文件描述符。request是ioctl的命令，可变参数取决于request，通常是一个指向变量或结构体的指针。若出错则返回-1，若成功则返回其他值，返回值也是取决于request。

以下程序使用TIOCGWINSZ命令获得终端设备的窗口大小。

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

int main(void)
{
    struct winsize size;
    if (isatty(STDOUT_FILENO) == 0)
        exit(1);
    if(ioctl(STDOUT_FILENO, TIOCGWINSZ, &size)<0) {
        perror("ioctl TIOCGWINSZ error");
        exit(1);
    }
    printf("%d rows, %d columns\n", size.ws_row, size.ws_col);
    return 0;
}
```

在图形界面的终端里多次改变终端窗口的大小并运行该程序，观察结果。

1.9 习题

* 在系统头文件中查找flags和mode参数用到的这些宏定义的值是多少。把这些宏定义按位或起来是什么效果？为什么必选项只能选一个而可选项可以选多个？
* 请按照下述要求分别写出相应的open调用。

打开文件/home/xingwenpeng/itcast.txt用于写操作，以追加方式打开

打开文件/home/xingwenpeng/itcast.txt用于写操作，如果该文件不存在则创建它

打开文件/home/xingwenpeng/itcast.txt用于写操作，如果该文件已存在则截断为0字节，如果该文件不存在则创建它

打开文件/home/xingwenpeng/itcast.txt用于写操作，如果该文件已存在则报错退出，如果该文件不存在则创建它

* 创建一个10M的空文件
* 实现输出重定向，当C标准printf打印时，打印到你指定的test文件里
* mycp拷贝命令实现。（思考如何拷贝目录呢？）
* 获取当前图形界面的屏幕分辨率

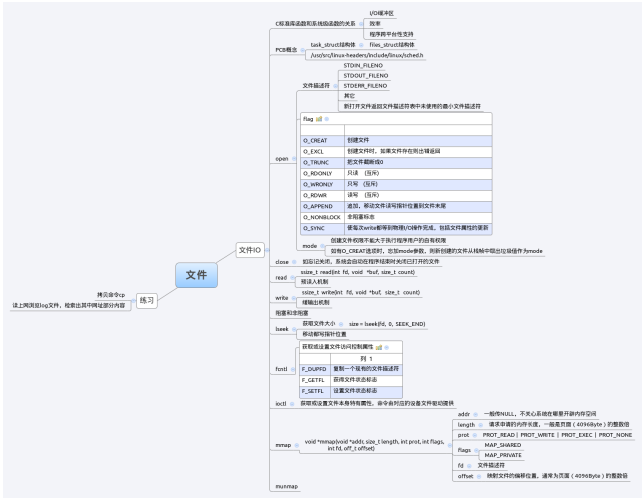
文件
文件IO
I/O缓冲区
C标准库函数和系统底层函数的关系
效率
程序跨平台性支持
PCB概念
task_struct结构体
/usr/src/linux-headers/include/linux/sched.h
文件描述符
STDIN_FILENO
STDOUT_FILENO
STDERR_FILENO
其它
新打开文件返回文件描述符表中未使用的最小文件描述符
open
flag
O_CREAT 创建文件
O_EXCL 创建文件时，如果文件存在则出错返回
O_TRUNC 把文件截断成0
O_RDONLY 只读 （互斥）
O_WRONLY 只写 （互斥）
O_RDWR 读写 （互斥）
O_APPEND 追加，移动文件读写指针位置到文件末尾
O_NONBLOCK 非阻塞标志
O_SYNC 要取决write都等到物理I/O操作完成。包括文件属性的更新
mode
创建文件权限不能大于执行程序用户的自有权限
如有O_CREAT选项时，忘加mode参数，则新创建的文件从栈帧中取出垃圾值作为mode
close
如忘记关闭，系统会自动在程序结束时关闭已打开的文件
read
ssize_t read(int fd, void *buf, size_t count)
输入机制
write
ssize_t write(int fd, void *buf, size_t count)
输输出机制
阻塞和非阻塞
lseek
获取文件大小
size = lseek(fd, 0, SEEK_END)
移动都写指针位置
fcntl
获取或设置文件访问控制属性
到 1
F_DUPFD 复制一个现有的文件描述符
F_GETFL 获得文件状态标志
F_SETFL 设置文件状态标志
ioctl
获取或设置文件本身特有属性。由令自己设的设备文件驱动提供
mmap
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
addr 一般传NULL，不关心系统在哪里开辟内存空间
length 请求映射的内存长度，一般是页面（4096byte）的整数倍
prot PROT_READ | PROT_WRITE | PROT_EXEC | PROT_NONE
flags MAP_SHARED
MAP_PRIVATE
fd 文件描述符
offset 映射文件的偏移位置，通常为页面（4096byte）的整数倍
munmap
练习
拷贝命令cp
读上网浏览log文件，检索出其中网址部分内容

图 1.3: 总结

第 2 章

文件系统

2.1 ext2文件系统

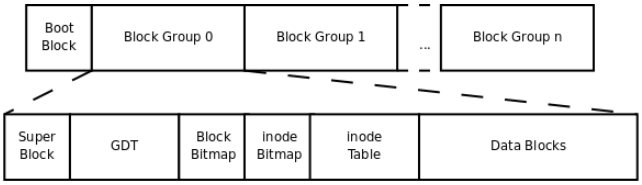

图 2.1: ext2文件系统

我们知道，一个磁盘可以划分成多个分区，每个分区必须先用格式化工具（例如某种mkfs命令）格式化成某种格式的文件系统，然后才能存储文件，格式化的过程会在磁盘上写一些管理存储布局的信息。下图是一个磁盘分区格式化成ext2文件系统后的存储布局。

文件系统中存储的最小单位是块（Block），一个块究竟多大是在格式化时确定的，例如mke2fs的-b选项可以设定块大小为1024、2048或4096字节。而上图中启动块（Boot Block）的大小是确定的，就是1KB，启动块是由PC标准规定的，用来存储磁盘分区信息和启动信息，任何文件系统都不能使用启动块。启动块之后才是ext2文件系统的开始，ext2文件系统将整个分区划成若干个同样大小的块组（Block Group），每个块组都由以下部分组成。

超级块（Super Block） 描述整个分区的文件系统信息，例如块大小、文件系统版本号、上次mount的时间等等。超级块在每个块组的开头都有一份拷贝。

块组描述符表（GDT，Group Descriptor Table） 由很多块组描述符组成，整个分区分成多少个块组就对应有多少个块组描述符。每个块组描述符（Group Descriptor）存储一个块组的描述信息，例如在这个块组中从哪里开始是inode表，从哪里开始是数据块，空闲的inode和数据块还有多少个等等。和超级块类似，块组描述符表在每个块组的开头也都有一份拷贝，这些信息是非常重要的，一旦超级块意外损坏就会丢失整个分区的数据，一旦块组描述符意外损坏就会丢失整个块组的数据，因此它们都有多份拷贝。通常内核只用到第0个块组中的拷贝，当执行e2fsck检查文件系统一致性时，第0个块组中的超级块和块组描述符表就会拷贝到其它块组，这样当第0个块组的开头意外损坏时就可以用其它拷贝来恢复，从而减少损失。

块位图（Block Bitmap） 一个块组中的块是这样利用的：数据块存储所有文件的数据，比如某个分区的块大小是1024字节，某个文件是2049字节，那么就需要三个数据块来存，即使第三个块只存了一个字节也需要占用一个整块；超级块、块组描述符表、块位

图、inode位图、inode表这几部分存储该块组的描述信息。那么如何知道哪些块已经用来存储文件数据或其它描述信息，哪些块仍然空闲可用呢？块位图就是用来描述整个块组中哪些块已用哪些块空闲的，它本身占一个块，其中的每个bit代表本块组中的一个块，这个bit为1表示该块已用，这个bit为0表示该块空闲可用。

为什么用df命令统计整个磁盘的已用空间非常快呢？因为只需要查看每个块组的块位图即可，而不需要搜遍整个分区。相反，用du命令查看一个较大目录的已用空间就非常慢，因为不可避免地要搜遍整个目录的所有文件。

与此相联系的另一个问题是：在格式化一个分区时究竟会划出多少个块组呢？主要的限制在于块位图本身必须只占一个块。用mke2fs格式化时默认块大小是1024字节，可以用-b参数指定块大小，现在设块大小指定为b字节，那么一个块可以有8b个bit，这样大小的一个块位图就可以表示8b个块的占用情况，因此一个块组最多可以有8b个块，如果整个分区有s个块，那么就可以有s/(8b)个块组。格式化时可以用-g参数指定一个块组有多少个块，但是通常不需要手动指定，mke2fs工具会计算出最优的数值。

inode位图（inode Bitmap） 和块位图类似，本身占一个块，其中每个bit表示一个inode是否空闲可用。

inode表（inode Table） 我们知道，一个文件除了数据需要存储之外，一些描述信息也需要存储，例如文件类型（常规、目录、符号链接等），权限，文件大小，创建/修改/访问时间等，也就是ls -l命令看到的那些信息，这些信息存在inode中而不是数据块中。每个文件都有一个inode，一个块组中的所有inode组成了inode表。

inode表占多少个块在格式化时就要决定并写入块组描述符中，mke2fs格式化工具的默认策略是一个块组有多少个8KB就分配多少个inode。由于数据块占了整个块组的绝大部分，也可以近似认为数据块有多少个8KB就分配多少个inode，换句话说，如果平均每个文件的大小是8KB，当分区存满的时候inode表会得到比较充分的利用，数据块也不浪费。如果这个分区存的都是很大的文件（比如电影），则数据块用完的时候inode会有一些浪费，如果这个分区存的都是很小的文件（比如源代码），则有可能数据块还没用完inode就已经用完了，数据块可能有很大的浪费。如果用户在格式化时能够对这个分区以后要存储的文件大小做一个预测，也可以用mke2fs的-i参数手动指定每多少个字节分配一个inode。

数据块（Data Block） 根据不同的文件类型有以下几种情况

对于常规文件，文件的数据存储在数据块中。

对于目录，该目录下的所有文件名和目录名存储在数据块中，注意文件名保存在它所在目录的数据块中，除文件名之外，ls -l命令看到的其它信息都保存在该文件的inode中。注意这个概念：目录也是一种文件，是一种特殊类型的文件。

对于符号链接，如果目标路径名较短则直接保存在inode中以便更快地查找，如果目标路径名较长则分配一个数据块来保存。

设备文件、FIFO和socket等特殊文件没有数据块，设备文件的主设备号和次设备号保存在inode中。

2.1.1 目录中记录项文件类型

```
编码 文件类型
0        Unknown
1        Regular file
2        Directory
3        Character device
4        Block device
```

```
5       Named pipe
6       Socket
7       Symbolic link
```

2.1.2 数据块寻址

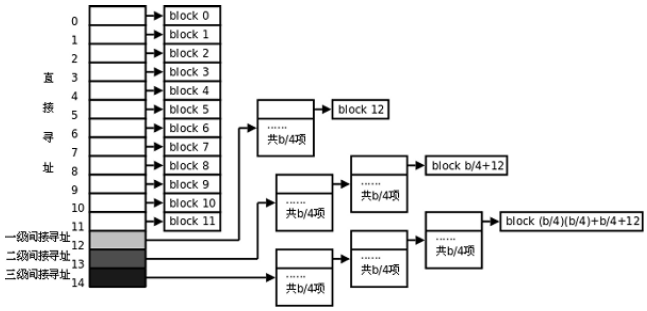

图 2.2: 数据块寻址

从上图可以看出，索引项Blocks[13]指向两级的间接寻址块，最多可表示(b/4)2+b/4+12个数据块，对于1K的块大小最大可表示64.26MB的文件。索引项Blocks[14]指向三级的间接寻址块，最多可表示(b/4)3+(b/4)2+b/4+12个数据块，对于1K的块大小最大可表示16.06GB的文件。

可见，这种寻址方式对于访问不超过12个数据块的小文件是非常快的，访问文件中的任意数据只需要两次读盘操作，一次读inode（也就是读索引项）一次读数据块。而访问大文件中的数据则需要最多五次读盘操作：inode、一级间接寻址块、二级间接寻址块、三级间接寻址块、数据块。实际上，磁盘中的inode和数据块往往已经被内核缓存了，读大文件的效率也不会太低。

2.2 stat

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

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);

struct stat {
               dev_t     st_dev;     /* ID of device containing file */
               ino_t     st_ino;     /* inode number */
               mode_t    st_mode;    /* protection */
               nlink_t   st_nlink;   /* number of hard links */
               uid_t     st_uid;     /* user ID of owner */
               gid_t     st_gid;     /* group ID of owner */
               dev_t     st_rdev;    /* device ID (if special file) */
               off_t     st_size;    /* total size, in bytes */
               blksize_t st_blksize; /* blocksize for file system I/O */
```

```
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

stat既有命令也有同名函数，用来获取文件Inode里主要信息，stat 跟踪符号链接，lstat不跟踪符号链接

stat里面时间辨析

atime(最近访问时间)： mtime(最近更改时间):指最近修改文件内容的时间 ctime(最近改动时间)：指最近改动Inode的时间 ## access

```
#include <unistd.h>

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

按实际用户ID和实际组ID测试,跟踪符号链接

参数mode

```
R_OK 是否有读权限
W_OK 是否有写权限
X_OK 是否有执行权限
F_OK 测试一个文件是否存在
```

实际用户ID： 有效用户ID：sudo执行时，有效用户ID是root，实际用户ID是xingwenpeng

2.3 chmod

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

2.4 chown

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
```

chown使用时必须拥有root权限。

	列 2	列 3
S_ISUID	04000	执行时设置用户ID
S_ISGID	02000	执行时设置组ID
S_ISVTX	01000	黏住位
S_ISUSR	00400	所有者_读
S_IWUSR	00200	所有者_写
S_IXUSR	00100	所有者_执行
S_IRGRP	00040	组_读
S_IWGRP	00020	组_写
S_IXGRP	00010	组_执行
S_IROTH	00004	其他_读
S_IWOTH	00002	其他_写
S_IXOTH	00001	其他_执行

图 2.3: mode标志

2.5 utime

2.6 truncate

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

int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

2.7 link

2.7.1 link

创建一个硬链接

当rm删除文件时，只是删除了目录下的记录项和把inode硬链接计数减1，当硬链接计数减为0时，才会真正的删除文件。

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
```

* 硬链接通常要求位于同一文件系统中,POSIX允许夸文件系统
* 符号链接没有文件系统限制
* 通常不允许创建目录的硬链接，某些unix系统下超级用户可以创建目录的硬链
* 创建目录项以及增加硬链接计数应当是一个原子操作

2.7.2 symlink

```
int symlink(const char *oldpath, const char *newpath)
```

2.7.3 readlink

读符号链接所指向的文件名字，不读文件内容

```
ssize_t readlink(const char *path, char *buf, size_t bufsiz)
```

2.7.4 unlink

```
int unlink(const char *pathname)
1. 如果是符号链接，删除符号链接
2. 如果是硬链接，硬链接数减1，当减为0时，释放数据块和inode
3. 如果文件硬链接数为0，但有进程已打开该文件，并持有文件描述符，则等该进程关闭该文件时，kernel才真正去删除该文件
4. 利用该特性创建临时文件，先open或creat创建一个文件，马上unlink此文件
```

2.8 rename

文件重命名

```
#include <stdio.h>
int rename(const char *oldpath, const char *newpath);
```

2.9 chdir

```
#include <unistd.h>

int chdir(const char *path);
int fchdir(int fd);
```

改变当前进程的工作目录

2.10 getcwd

获取当前进程的工作目录

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

2.11 pathconf

```
#include <unistd.h>

long fpathconf(int fd, int name);
long pathconf(char *path, int name);
```

2.12 目录操作

2.12.1 mkdir

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

int mkdir(const char *pathname, mode_t mode);
```

2.12.2 rmdir

```
#include <unistd.h>
int rmdir(const char *pathname);
```

2.12.3 opendir/fdopendir

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

DIR *opendir(const char *name);
DIR *fdopendir(int fd);
```

2.12.4 readdir

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);

struct dirent {
            ino_t          d_ino;       /* inode number */
            off_t          d_off;       /* offset to the next dirent */
            unsigned short d_reclen;    /* length of this record */
            unsigned char  d_type;      /* type of file; not supported
                                           by all file system types */
            char           d_name[256]; /* filename */
        };
```

readdir每次返回一条记录项，DIR*指针指向下一条记录项

2.12.5 rewinddir

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

#include <dirent.h>

void rewinddir(DIR *dirp);
```

把目录指针恢复到目录的起始位置。

2.12.6 telldir/seekdir

```
#include <dirent.h>
long telldir(DIR *dirp);

#include <dirent.h>
void seekdir(DIR *dirp, long offset);
```

2.12.7 closedir

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

int closedir(DIR *dirp);
```

2.12.8 递归遍历目录

递归列出目录中的文件列表

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

#define MAX_PATH 1024

/* dirwalk:  apply fcn to all files in dir */
void dirwalk(char *dir, void (*fcn)(char *))
{
    char name[MAX_PATH];
    struct dirent *dp;
    DIR *dfd;

    if ((dfd = opendir(dir)) == NULL) {
        fprintf(stderr, "dirwalk: can't open %s\n", dir);
        return;
    }
    while ((dp = readdir(dfd)) != NULL) {
        if (strcmp(dp->d_name, ".") == 0
            || strcmp(dp->d_name, "..") == 0)
            continue;    /* skip self and parent */
        if (strlen(dir)+strlen(dp->d_name)+2 > sizeof(name))
            fprintf(stderr, "dirwalk: name %s %s too long\n",
                dir, dp->d_name);
        else {
            sprintf(name, "%s/%s", dir, dp->d_name);
            (*fcn)(name);
        }
    }
    closedir(dfd);
}

/* fsize:  print the size and name of file "name" */
void fsize(char *name)
{
    struct stat stbuf;

    if (stat(name, &stbuf) == -1) {
        fprintf(stderr, "fsize: can't access %s\n", name);
        return;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        dirwalk(name, fsize);
    printf("%8ld %s\n", stbuf.st_size, name);
}

int main(int argc, char **argv)
{
    if (argc == 1)  /* default: current directory */
        fsize(".");
    else
        while (--argc > 0)
```

```
            fsize(*++argv);
    return 0;
}
```

然而这个程序还是不如ls -R健壮，它有可能死循环，思考一下什么情况会导致死循环。

2.13 VFS虚拟文件系统

Linux支持各种各样的文件系统格式，如ext2、ext3、reiserfs、FAT、NTFS、iso9660等等，不同的磁盘分区、光盘或其它存储设备都有不同的文件系统格式，然而这些文件系统都可以mount到某个目录下，使我们看到一个统一的目录树，各种文件系统上的目录和文件我们用ls命令看起来是一样的，读写操作用起来也都是一样的，这是怎么做到的呢？Linux内核在各种不同的文件系统格式之上做了一个抽象层，使得文件、目录、读写访问等概念成为抽象层的概念，因此各种文件系统看起来用起来都一样，这个抽象层称为虚拟文件系统（VFS，Virtual Filesystem）。这一节我们介绍运行时文件系统在内核中的表示。

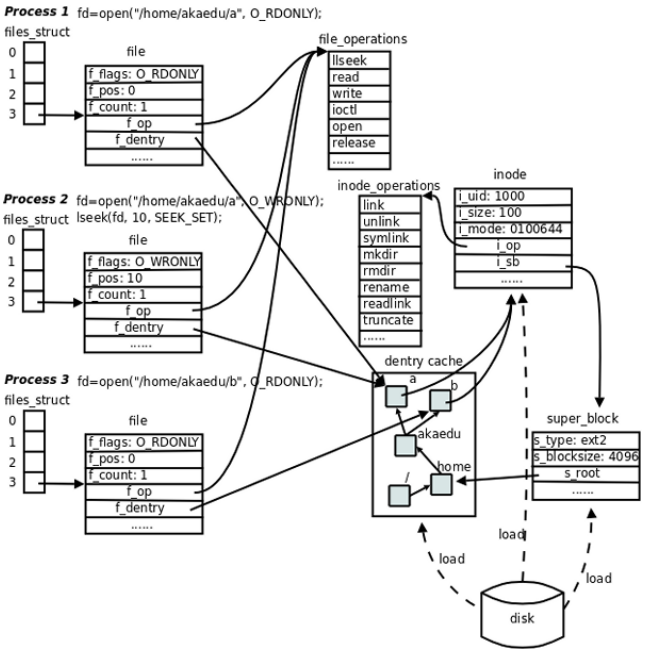

图 2.4: 虚拟文件系统

2.13.1 dup/dup2

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

dup和dup2都可用来复制一个现存的文件描述符，使两个文件描述符指向同一个file结构体。如果两个文件描述符指向同一个file结构体，File Status Flag和读写位置只保存一

份在file结构体中，并且file结构体的引用计数是2。如果两次open同一文件得到两个文件描述符，则每个描述符对应一个不同的file结构体，可以有不同的File Status Flag和读写位置。请注意区分这两种情况。

```
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    int fd, save_fd;
    char msg[] = "This is a test\n";

    fd = open("somefile", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
    if(fd<0) {
        perror("open");
        exit(1);
    }
    save_fd = dup(STDOUT_FILENO);
    dup2(fd, STDOUT_FILENO);
    close(fd);
    write(STDOUT_FILENO, msg, strlen(msg));
    dup2(save_fd, STDOUT_FILENO);
    write(STDOUT_FILENO, msg, strlen(msg));
    close(save_fd);
    return 0;
}
```

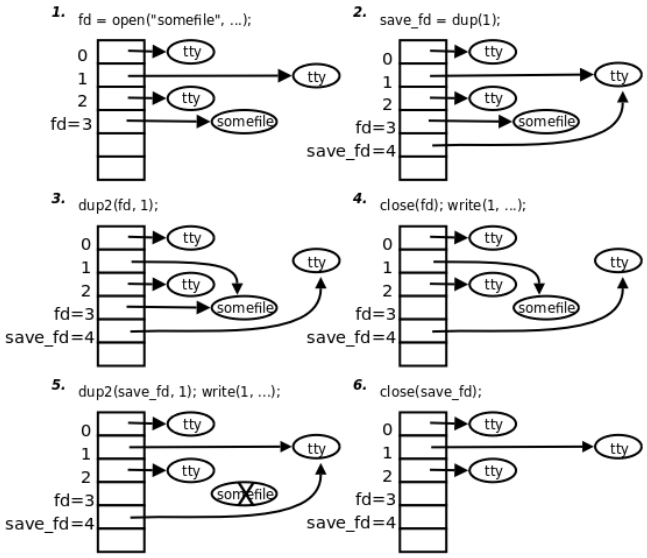

图 2.5: 示例

2.14 练习

1. 实现ls -l功能，可以解析文件权限位。
2. 实现ls -l功能，可以解析出文件所有者和文件所有组。（偏难）
3. 实现rm删除命令，如
   rm file
   rm directory
   *注意，千万不要在你有代码的目录下做测试，防止删除你的有用文件,友情提示（rmdir/unlink和递归遍历目录）
4. 从文件里面读出1000个随机数，进行排序，再写到另一文件中。（考虑使用重定向dup/dup2）

第 3 章

进程

我们知道，每个进程在内核中都有一个进程控制块（PCB）来维护进程相关的信息，Linux内核的进程控制块是task_struct结构体。现在我们全面了解一下其中都有哪些信息。

* 进程id。系统中每个进程有唯一的id，在C语言中用pid_t类型表示，其实就是一个非负整数。
* 进程的状态，有运行、挂起、停止、僵尸等状态。
* 进程切换时需要保存和恢复的一些CPU寄存器。
* 描述虚拟地址空间的信息。
* 描述控制终端的信息。
* 当前工作目录（Current Working Directory）。
* umask掩码。
* 文件描述符表，包含很多指向file结构体的指针。
* 和信号相关的信息。
* 用户id和组id。
* 控制终端、Session和进程组。
* 进程可以使用的资源上限（Resource Limit）。

目前大家并不需要理解这些信息的细节，在随后课程中讲到某一项时会再次提醒大家它是保存在PCB中的。

fork和exec是本章要介绍的两个重要的系统调用。fork的作用是根据一个现有的进程复制出一个新进程，原来的进程称为父进程（Parent Process），新进程称为子进程（Child Process）。系统中同时运行着很多进程，这些进程都是从最初只有一个进程开始一个一个复制出来的。在Shell下输入命令可以运行一个程序，是因为Shell进程在读取用户输入的命令之后会调用fork复制出一个新的Shell进程，然后新的Shell进程调用exec执行新的程序。

我们知道一个程序可以多次加载到内存，成为同时运行的多个进程，例如可以同时开多个终端窗口运行/bin/bash，另一方面，一个进程在调用exec前后也可以分别执行两个不同的程序，例如在Shell提示符下输入命令ls，首先fork创建子进程，这时子进程仍在执行/bin/bash程序，然后子进程调用exec执行新的程序/bin/ls 。

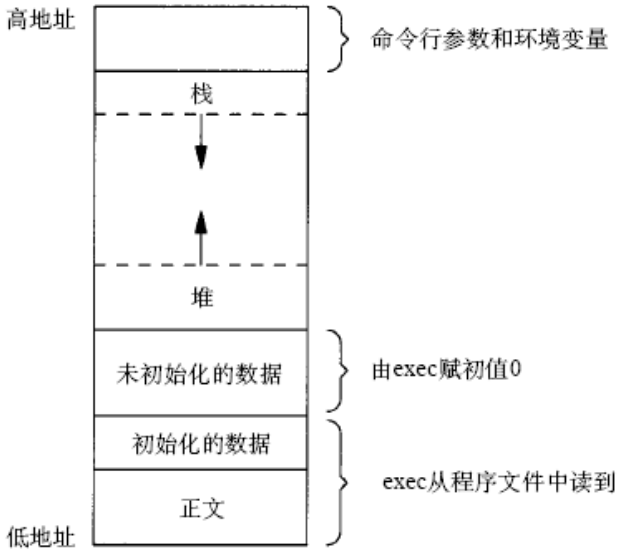

图 3.1

3.1 进程环境

libc中定义的全局变量environ指向环境变量表，environ没有包含在任何头文件中，所以在使用时要用extern声明。例如：

```
#include <stdio.h>

int main(void)
{
    extern char **environ;
    int i;
    for(i=0; environ[i]!=NULL; i++)
        printf("%s\n", environ[i]);
    return 0;
}
```

由于父进程在调用fork创建子进程时会把自己的环境变量表也复制给子进程，所以a.out打印的环境变量和Shell进程的环境变量是相同的。

按照惯例，环境变量字符串都是name=value这样的形式，大多数name由大写字母加下划线组成，一般把name的部分叫做环境变量，value的部分则是环境变量的值。环境变量定义了进程的运行环境，一些比较重要的环境变量的含义如下：

PATH

* 可执行文件的搜索路径。ls命令也是一个程序，执行它不需要提供完整的路径名/bin/ls，然而通常我们执行当前目录下的程序a.out却需要提供完整的路径名./a.out，这是因为PATH环境变量的值里面包含了ls命令所在的目录/bin，却不包含a.out所在的目录。PATH环境变量的值可以包含多个目录，用:号隔开。在Shell中用echo命令可以查看这个环境变量的值：

  $ echo $PATH

SHELL

* 当前Shell，它的值通常是/bin/bash。

TERM

* 当前终端类型，在图形界面终端下它的值通常是xterm，终端类型决定了一些程序的输出显示方式，比如图形界面终端可以显示汉字，而字符终端一般不行。

LANG

* 语言和locale，决定了字符编码以及时间、货币等信息的显示格式。

HOME

* 当前用户主目录的路径，很多程序需要在主目录下保存配置文件，使得每个用户在运行该程序时都有自己的一套配置。

用environ指针可以查看所有环境变量字符串，但是不够方便，如果给出name要在环境变量表中查找它对应的value，可以用getenv函数。

```
#include <stdlib.h>
char *getenv(const char *name);
getenv的返回值是指向value的指针，若未找到则为NULL。
```

修改环境变量可以用以下函数

```
#include <stdlib.h>

int setenv(const char *name, const char *value, int rewrite);
void unsetenv(const char *name);

putenv和setenv函数若成功则返回为0，若出错则返回非0。
```

setenv将环境变量name的值设置为value。如果已存在环境变量name，那么
若rewrite非0，则覆盖原来的定义；
若rewrite为0，则不覆盖原来的定义，也不返回错误。
unsetenv删除name的定义。即使name没有定义也不返回错误。
例 修改环境变量

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

int main(void)
{
    printf("PATH=%s\n", getenv("PATH"));
    setenv("PATH", "hello", 1);
    printf("PATH=%s\n", getenv("PATH"));
    return 0;
}
```

3.2 进程状态

画图演示

修改进程资源限制，软限制可改，最大值不能超过硬限制，硬限制只有root用户可以修改

```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
```

查看进程资源限制

```
cat /proc/self/limits
ulimit -a
```

3.3 进程原语

3.3.1 fork

```
#include <unistd.h>

pid_t fork(void);
```

子进程复制父进程的0到3g空间和父进程内核中的PCB，但id号不同。

fork调用一次返回两次

```
+ 父进程中返回子进程ID
+ 子进程中返回0
+ 读时共享，写时复制
```

fork

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

int main(void)
{
```

```
    pid_t pid;
    char *message;
    int n;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        message = "This is the child\n";
        n = 6;
    } else {
        message = "This is the parent\n";
        n = 3;
    }
    for(; n > 0; n--) {
        printf(message);
        sleep(1);
    }
    return 0;
}
```

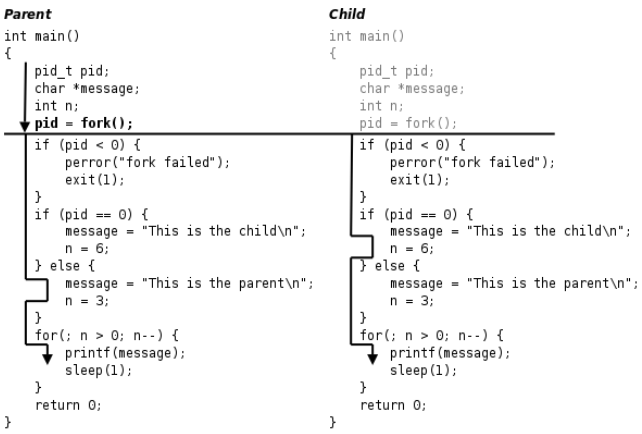

图 3.2: fork

进程相关函数

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

pid_t getpid(void);   //返回调用进程的PID号
pid_t getppid(void);  //返回调用进程父进程的PID号
```

getpid/gteppid

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

uid_t getuid(void);  //返回实际用户ID
uid_t geteuid(void); //返回有效用户ID
```

getuid

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

gid_t getgid(void);  //返回实际用户组ID
gid_t getegid(void); //返回有效用户组ID
```

getgid

vfork

* 用于fork后马上调用exec函数
* 父子进程，共用同一地址空间,子进程如果没有马上exec而是修改了父进程出得到的变量值，此修改会在父进程中生效
* 设计初衷，提高系统效率，减少不必要的开销
* 现在fork已经具备读时共享写时复制机制，vfork逐渐废弃

3.3.2 exec族

用fork创建子进程后执行的是和父进程相同的程序（但有可能执行不同的代码分支），子进程往往要调用一种exec函数以执行另一个程序。当进程调用一种exec函数时，该进程的用户空间代码和数据完全被新程序替换，从新程序的启动例程开始执行。调用exec并不创建新进程，所以调用exec前后该进程的id并未改变。

其实有六种以exec开头的函数，统称exec函数：

```
#include <unistd.h>

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 *path, char *const argv[], char *const envp[]);
```

这些函数如果调用成功则加载新的程序从启动代码开始执行，不再返回，如果调用出错则返回-1，所以exec函数只有出错的返回值而没有成功的返回值。

这些函数原型看起来很容易混，但只要掌握了规律就很好记。不带字母p（表示path）的exec函数第一个参数必须是程序的相对路径或绝对路径，例如“/bin/ls”或“./a.out”，而不能是“ls”或“a.out”。对于带字母p的函数：

如果参数中包含/，则将其视为路径名。

否则视为不带路径的程序名，在PATH环境变量的目录列表中搜索这个程序。

带有字母l（表示list）的exec函数要求将新程序的每个命令行参数都当作一个参数传给它，命令行参数的个数是可变的，因此函数原型中有…，…中的最后一个可变参数应该是NULL，起sentinel的作用。对于带有字母v（表示vector）的函数，则应该先构造一个指向各参数的指针数组，然后将该数组的首地址当作参数传给它，数组中的最后一个指针也应该是NULL，就像main函数的argv参数或者环境变量表一样。

对于以e（表示environment）结尾的exec函数，可以把一份新的环境变量表传给它，其他exec函数仍使用当前的环境变量表执行新程序。

exec调用举例如下：

```
char *const ps_argv[] ={"ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL};
char *const ps_envp[] ={"PATH=/bin:/usr/bin", "TERM=console", NULL};
execl("/bin/ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);
execv("/bin/ps", ps_argv);
execle("/bin/ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL, ps_envp);
execve("/bin/ps", ps_argv, ps_envp);
execlp("ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);
execvp("ps", ps_argv);
```

事实上，只有execve是眞正的系统调用，其它五个函数最终都调用execve，所以execve在man手册第2节，其它函数在man手册第3节。这些函数之间的关系如下图所示。

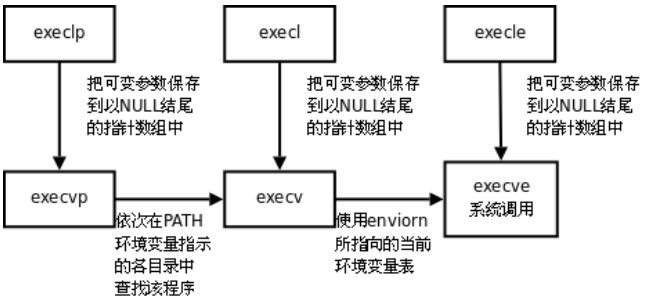

图 3.3: exec函数族

一个完整的例子：

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

int main(void)
{
    execlp("ps", "ps", "-o", "pid,ppid,pgrp,session,tpgid,comm", NULL);
    perror("exec ps");
    exit(1);
}
```

由于exec函数只有错误返回值，只要返回了一定是出错了，所以不需要判断它的返回值，直接在后面调用perror即可。注意在调用execlp时传了两个“ps”参数，第一个“ps”是程序名，execlp函数要在PATH环境变量中找到这个程序并执行它，而第二个“ps”是第一个命令行参数，execlp函数并不关心它的值，只是简单地把它传给ps程序，ps程序可以通过main函数的argv[0]取到这个参数。

调用exec后，原来打开的文件描述符仍然是打开的。利用这一点可以实现I/O重定向。先看一个简单的例子，把标准输入转成大写然后打印到标准输出：

例 upper

```
/* upper.c */
#include <stdio.h>

int main(void)
{
    int ch;
    while((ch = getchar()) != EOF) {
        putchar(toupper(ch));
    }
    return 0;
}
```

例 wrapper

```
/* wrapper.c */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
    int fd;
    if (argc != 2) {
        fputs("usage: wrapper file\n", stderr);
        exit(1);
    }
    fd = open(argv[1], O_RDONLY);
    if(fd<0) {
        perror("open");
        exit(1);
    }
    dup2(fd, STDIN_FILENO);
    close(fd);
    execl("./upper", "upper", NULL);
    perror("exec ./upper");
    exit(1);
}
```

wrapper程序将命令行参数当作文件名打开，将标准输入重定向到这个文件，然后调用exec执行upper程序，这时原来打开的文件描述符仍然是打开的，upper程序只负责从标准输入读入字符转成大写，并不关心标准输入对应的是文件还是终端。运行结果如下：

exec族

```
l  命令行参数列表
p  搜索file时使用path变量
v  使用命令行参数数组
e  使用环境变量数组,不使用进程原有的环境变量，设置新加载程序运行的环境变量
```

3.3.3 wait/waitpid

僵尸进程：子进程退出，父进程没有回收子进程资源（PCB），则子进程变成僵尸进程

孤儿进程：父进程先于子进程结束，则子进程成为孤儿进程,子进程的父进程成为1号进程init进程，称为init进程领养孤儿进程

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

pid_t wait(int *status);

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

< -1 回收指定进程组内的任意子进程
-1   回收任意子进程
0    回收和当前调用waitpid一个组的所有子进程
> 0  回收指定ID的子进程
```

一个进程在终止时会关闭所有文件描述符，释放在用户空间分配的内存，但它的PCB还保留着，内核在其中保存了一些信息：如果是正常终止则保存着退出状态，如果是异常终止则保存着导致该进程终止的信号是哪个。这个进程的父进程可以调用wait或waitpid获取这些信息，然后彻底清除掉这个进程。我们知道一个进程的退出状态可以在Shell中用特殊变量$?查看，因为Shell是它的父进程，当它终止时Shell调用wait或waitpid得到它的退出状态同时彻底清除掉这个进程。

如果一个进程已经终止，但是它的父进程尚未调用wait或waitpid对它进行清理，这时的进程状态称为僵尸（Zombie）进程。任何进程在刚终止时都是僵尸进程，正常情况下，僵尸进程都立刻被父进程清理了，为了观察到僵尸进程，我们自己写一个不正常的程序，父进程fork出子进程，子进程终止，而父进程既不终止也不调用wait清理子进程：

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

int main(void)
{
    pid_t pid=fork();
    if(pid<0) {
        perror("fork");
        exit(1);
    }
    if(pid>0) { /* parent */
```

```
        while(1);
    }
    /* child */
    return 0;
}
```

若调用成功则返回清理掉的子进程id，若调用出错则返回-1。父进程调用wait或waitpid时可能会：

* 阻塞（如果它的所有子进程都还在运行）。
* 带子进程的终止信息立即返回（如果一个子进程已终止，正等待父进程读取其终止信息）。
* 出错立即返回（如果它没有任何子进程）。

这两个函数的区别是：

* 如果父进程的所有子进程都还在运行，调用wait将使父进程阻塞，而调用waitpid时如果在options参数中指定WNOHANG可以使父进程不阻塞而立即返回0。
* wait等待第一个终止的子进程，而waitpid可以通过pid参数指定等待哪一个子进程。

可见，调用wait和waitpid不仅可以获得子进程的终止信息，还可以使父进程阻塞等待子进程终止，起到进程间同步的作用。如果参数status不是空指针，则子进程的终止信息通过这个参数传出，如果只是为了同步而不关心子进程的终止信息，可以将status参数指定为NULL。

例 waitpid

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

int main(void)
{
    pid_t pid;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        int i;
        for (i = 3; i > 0; i--) {
            printf("This is the child\n");
            sleep(1);
        }
        exit(3);
    } else {
        int stat_val;
        waitpid(pid, &stat_val, 0);
```

```
        if (WIFEXITED(stat_val))
            printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
        else if (WIFSIGNALED(stat_val))
            printf("Child terminated abnormally, signal %d\n", WTERMSIG(stat_val));
    }
    return 0;
}
```

wait阻塞函数，阻塞等待子进程结束 waitpid 4种情况 < -1 = -1 = 0 > 0

```
    进程的退出状态
    非阻塞标志, WNOHANG
    获取进程退出状态的函数见manpages
    调用进程若无子进程, 则wait出错返回
```

3.4 练习

1.实现多进程拷贝命令

2.实现多进程打印输出自己的身份

第 4 章

进程间通信

每个进程各自有不同的用户地址空间，任何一个进程的全局变量在另一个进程中都看不到，所以进程之间要交换数据必须通过内核，在内核中开辟一块缓冲区，进程1把数据从用户空间拷到内核缓冲区，进程2再从内核缓冲区把数据读走，内核提供的这种机制称为进程间通信（IPC，InterProcess Communication）。

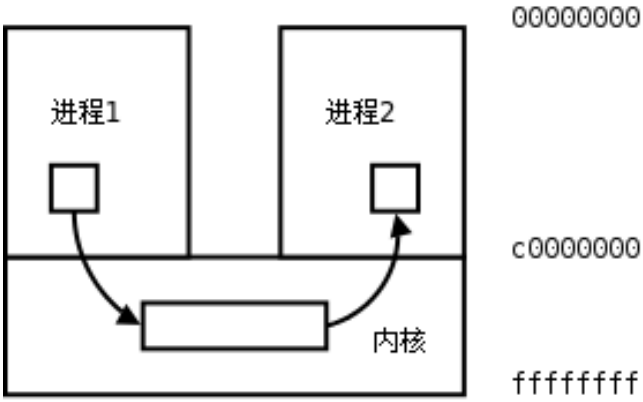

图 4.1: 进程间通信

4.1 pipe管道

管道是一种最基本的IPC机制，由pipe函数创建：

```
#include <unistd.h>

int pipe(int filedes[2]);
```

管道作用于有血缘关系的进程之间,通过fork来传递

调用pipe函数时在内核中开辟一块缓冲区（称为管道）用于通信，它有一个读端一个写端，然后通过filedes参数传出给用户程序两个文件描述符，filedes[0]指向管道的读端，filedes[1]指向管道的写端（很好记，就像0是标准输入1是标准输出一样）。所以管道在用户程序看起来就像一个打开的文件，通过read(filedes[0]);或者write(filedes[1]);

向这个文件读写数据其实是在读写内核缓冲区。pipe函数调用成功返回0，调用失败返回-1。

开辟了管道之后如何实现两个进程间的通信呢？比如可以按下面的步骤通信。

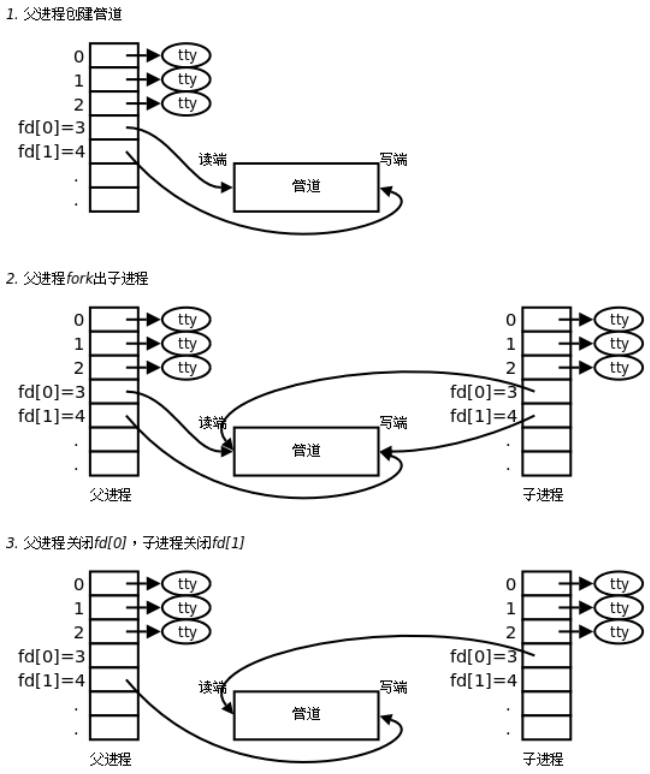

图 4.2: pipe管道

1.父进程调用pipe开辟管道，得到两个文件描述符指向管道的两端。

2.父进程调用fork创建子进程，那么子进程也有两个文件描述符指向同一管道。

3.父进程关闭管道读端，子进程关闭管道写端。父进程可以往管道里写，子进程可以从管道里读，管道是用环形队列实现的，数据从写端流入从读端流出，这样就实现了进程间通信。

例 pipe管道

```
#include <stdlib.h>
#include <unistd.h>
#define MAXLINE 80

int main(void)
{
    int n;
    int fd[2];
    pid_t pid;
    char line[MAXLINE];

    if (pipe(fd) < 0) {
        perror("pipe");
        exit(1);
    }
    if ((pid = fork()) < 0) {
```

```
        perror("fork");
        exit(1);
    }
    if (pid > 0) { /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
        wait(NULL);
    } else {       /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    return 0;
}
```

使用管道有一些限制：

两个进程通过一个管道只能实现单向通信，比如上面的例子，父进程写子进程读，如果有时候也需要子进程写父进程读，就必须另开一个管道。请读者思考，如果只开一个管道，但是父进程不关闭读端，子进程也不关闭写端，双方都有读端和写端，为什么不能实现双向通信？

管道的读写端通过打开的文件描述符来传递，因此要通信的两个进程必须从它们的公共祖先那里继承管道文件描述符。上面的例子是父进程把文件描述符传给子进程之后父子进程之间通信，也可以父进程fork两次，把文件描述符传给两个子进程，然后两个子进程之间通信，总之需要通过fork传递文件描述符使两个进程都能访问同一管道，它们才能通信。

使用管道需要注意以下4种特殊情况（假设都是阻塞I/O操作，没有设置O_NONBLOCK标志）：

1.如果所有指向管道写端的文件描述符都关闭了（管道写端的引用计数等于0），而仍然有进程从管道的读端读数据，那么管道中剩余的数据都被读取后，再次read会返回0，就像读到文件末尾一样。

2.如果有指向管道写端的文件描述符没关闭（管道写端的引用计数大于0），而持有管道写端的进程也没有向管道中写数据，这时有进程从管道读端读数据，那么管道中剩余的数据都被读取后，再次read会阻塞，直到管道中有数据可读了才读取数据并返回。

3.如果所有指向管道读端的文件描述符都关闭了（管道读端的引用计数等于0），这时有进程向管道的写端write，那么该进程会收到信号SIGPIPE，通常会导致进程异常终止。讲信号时会讲到怎样使SIGPIPE信号不终止进程。

4.如果有指向管道读端的文件描述符没关闭（管道读端的引用计数大于0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道写端写数据，那么在管道被写满时再次write会阻塞，直到管道中有空位置了才写入数据并返回。

管道的这四种特殊情况具有普遍意义。

非阻塞管道，fcntl函数设置O_NONBLOCK标志

fpathconf(int fd, int name)测试管道缓冲区大小，_PC_PIPE_BUF

4.2 fifo有名管道

创建一个有名管道，解决无血缘关系的进程通信，fifo:

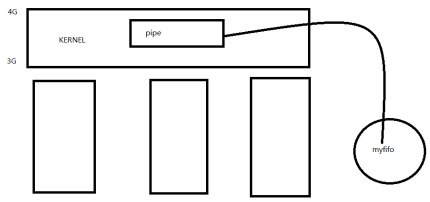

图 4.3: fifo通信

```
xingwenpeng@ubuntu:~$ mkfifo xwp
xingwenpeng@ubuntu:~$ ls -l xwp
prw-rw-r-- 1 xingwenpeng xingwenpeng 0  9月 15 18:34 xwp
```

mkfifo 既有命令也有函数

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

int mkfifo(const char *pathname, mode_t mode);
```

* 当只写打开FIFO管道时，如果没有FIFO没有读端打开，则open写打开会阻塞。
* FIFO内核实现时可以支持双向通信。（pipe单向通信，因为父子进程共享同一个file结构体）
* FIFO可以一个读端，多个写端；也可以一个写端，多个读端。（请测试）

4.3 内存共享映射

4.3.1 mmap/munmap

mmap可以把磁盘文件的一部分直接映射到内存，这样文件中的位置直接就有对应的内存地址，对文件的读写可以直接用指针来做而不需要read/write函数。

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

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

如果addr参数为NULL，内核会自己在进程地址空间中选择合适的地址建立映射。如果addr不是NULL，则给内核一个提示，应该从什么地址开始映射，内核会选择addr之上的某个合适的地址开始映射。建立映射后，真正的映射首地址通过返回值可以得到。len参数是需要映射的那一部分文件的长度。off参数是从文件的什么位置开始映射，必须是页大小的整数倍（在32位体系统结构上通常是4K）。filedes是代表该文件的描述符。

prot参数有四种取值：

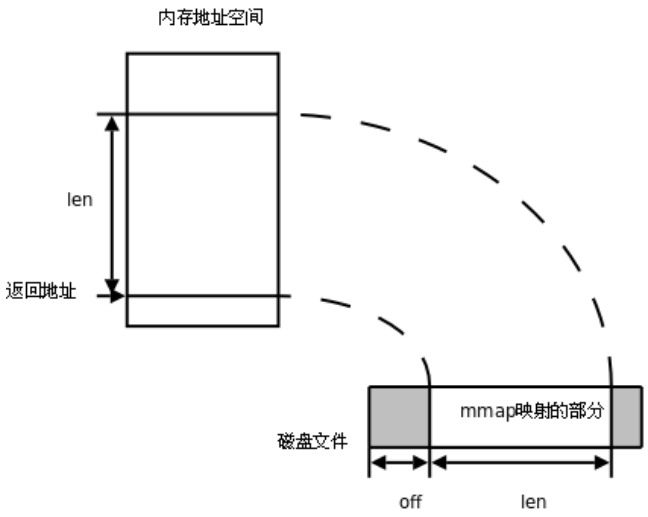

图 4.4: mmap

* PROT_EXEC表示映射的这一段可执行，例如映射共享库
* PROT_READ表示映射的这一段可读
* PROT_WRITE表示映射的这一段可写
* PROT_NONE表示映射的这一段不可访问

flag参数有很多种取值，这里只讲两种，其它取值可查看mmap(2)

* MAP_SHARED多个进程对同一个文件的映射是共享的，一个进程对映射的内存做了修改，另一个进程也会看到这种变化。
* MAP_PRIVATE多个进程对同一个文件的映射不是共享的，一个进程对映射的内存做了修改，另一个进程并不会看到这种变化，也不会真的写到文件中去。

如果mmap成功则返回映射首地址，如果出错则返回常数MAP_FAILED。当进程终止时，该进程的映射内存会自动解除，也可以调用munmap解除映射。munmap成功返回0，出错返回-1。

下面做一个简单的实验。

```
xingwenpeng@ubuntu:~$ vi hello
xingwenpeng@ubuntu:~$ cat hello
helloworld
xingwenpeng@ubuntu:~$ od -tx1 -tc hello
0000000  68  65  6c  6c  6f  77  6f  72  6c  64  0a
          h   e   l   l   o   w   o   r   l   d  \n
0000013
```

使用mmap映射

```
#include <stdlib.h>
#include <sys/mman.h>
#include <fcntl.h>

int main(void)
```

```
{
    int *p;
    int fd = open("hello", O_RDWR);
    if (fd < 0) {
        perror("open hello");
        exit(1);
    }
    p = mmap(NULL, 6, PROT_WRITE, MAP_SHARED, fd, 0);
    if (p == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }
    close(fd);
    p[0] = 0x30313233;
    munmap(p, 6);
    return 0;
}
```

* 用于进程间通信时，一般设计成结构体，来传输通信的数据
* 进程间通信的文件，应该设计成临时文件
* 当报总线错误时，优先查看共享文件是否有存储空间

4.3.2 进程间共享通信

写进程实现

```
/* process_mmap_w.c*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#define MAPLEN  0x1000

struct STU {
    int id;
    char name[20];
    char sex;
};

void sys_err(char *str, int exitno)
{
    perror(str);
    exit(exitno);
}
int main(int argc, char *argv[])
{
    struct STU *mm;
    int fd, i = 0;
    if (argc < 2) {
```

```
        printf("./a.out filename\n");
        exit(1);
    }
    fd = open(argv[1], O_RDWR | O_CREAT, 0777);
    if (fd < 0)
        sys_err("open", 1);

    if (lseek(fd, MAPLEN-1, SEEK_SET) < 0)
        sys_err("lseek", 3);

    if (write(fd, "\0", 1) < 0)
        sys_err("write", 4);

    mm = mmap(NULL, MAPLEN, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (mm == MAP_FAILED)
        sys_err("mmap", 2);

    close(fd);

    while (1) {
        mm->id = i;
        sprintf(mm->name, "zhang-%d", i);
        if (i % 2 == 0)
            mm->sex = 'm';
        else
            mm->sex = 'w';
        i++;
        sleep(1);
    }
    munmap(mm, MAPLEN);
    return 0;
}
```

读进程实现

```
/* process_mmap_r.c*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#define MAPLEN  0x1000

struct STU {
    int id;
    char name[20];
    char sex;
};

void sys_err(char *str, int exitno)
{
    perror(str);
```

```
    exit(exitno);
}
int main(int argc, char *argv[])
{
    struct STU *mm;
    int fd, i = 0;
    if (argc < 2) {
        printf("./a.out filename\n");
        exit(1);
    }
    fd = open(argv[1], O_RDWR);
    if (fd < 0)
        sys_err("open", 1);

    mm = mmap(NULL, MAPLEN, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (mm == MAP_FAILED)
        sys_err("mmap", 2);

    close(fd);
    unlink(argv[1]);

    while (1) {
        printf("%d\n", mm->id);
        printf("%s\n", mm->name);
        printf("%c\n", mm->sex);
        sleep(1);
    }
    munmap(mm, MAPLEN);
    return 0;
}
```

4.4 Unix Domain Socket

讲网络编程socket时再来介绍此方法

```
xingwenpeng@ubuntu:~$ ls -l /var/run/
总用量 72
srw-rw-rw-  1 root       root          0  9月 15 18:18 acpid.socket
...
srw-rw-rw-  1 root       root          0  9月 15 18:18 rpcbind.sock
```

4.5 习题

1.在父进程只用到写端，因而把读端关闭，子进程只用到读端，因而把写端关闭，然后互相通信，不使用的读端或写端必须关闭，请读者想一想如果不关闭会有什么问题。

2.请大家修改pipe管道例题代码，验证我上面所说的管道四种特殊情况。

3.利用fifo实现本地聊天室

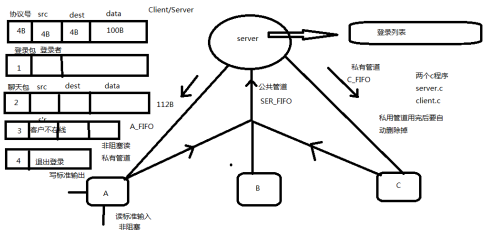
协议号 src dest data Client/Server
4B 4B 4B 100B
登录包 登录者
1
聊天包 src dest data
2
112B
3 客户不在线
A_FIFO
4 退出登录
非阻塞读
私有管道
写标准输出
读标准输入
非阻塞
server
登录列表
私有管道
C_FIFO
公共管道
SER_FIFO
两个c程序
server.c
client.c
私用管道用完后要自
动删除掉
A
B
C

图 4.5: 本地聊天室

第 5 章

信号

5.1 信号的概念

5.1.1 信号编号

kill -l

```
xingwenpeng@ubuntu:~$ 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) SIGIO       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
```

5.1.2 信号机制

man 7 signal

```
Term   Default action is to terminate the process.

Ign    Default action is to ignore the signal.

Core   Default action is to terminate the process and dump core (see core(5)).

Stop   Default action is to stop the process.

Cont   Default action is to continue the process if it is currently stopped.
```

First the signals described in the original POSIX.1-1990 standard.

```
Signal     Value     Action   Comment

SIGHUP        1       Term    Hangup detected on controlling terminal
                              or death of controlling process
SIGINT        2       Term    Interrupt from keyboard
SIGQUIT       3       Core    Quit from keyboard
SIGILL        4       Core    Illegal Instruction
SIGABRT       6       Core    Abort signal from abort(3)
SIGFPE        8       Core    Floating point exception
SIGKILL       9       Term    Kill signal
SIGSEGV      11       Core    Invalid memory reference
SIGPIPE      13       Term    Broken pipe: write to pipe with no
                              readers
SIGALRM      14       Term    Timer signal from alarm(2)
SIGTERM      15       Term    Termination signal
SIGUSR1   30,10,16    Term    User-defined signal 1
SIGUSR2   31,12,17    Term    User-defined signal 2
SIGCHLD   20,17,18    Ign     Child stopped or terminated
SIGCONT   19,18,25    Cont    Continue if stopped
SIGSTOP   17,19,23    Stop    Stop process
SIGTSTP   18,20,24    Stop    Stop typed at tty
SIGTTIN   21,21,26    Stop    tty input for background process
SIGTTOU   22,22,27    Stop    tty output for background process

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.
```

表中第一列是各信号的宏定义名称，第二列是各信号的编号，第三列是默认处理动作：

```
Term表示终止当前进程.
Core表示终止当前进程并且Core Dump（Core Dump 用于gdb调试）.
Ign表示忽略该信号.
Stop表示停止当前进程.
Cont表示继续执行先前停止的进程.
```

表中最后一列是简要介绍，说明什么条件下产生该信号。

5.1.3 信号产生种类

```
    ctl+c SIGINT
    ctl+z SIGTSTP
    ctl+\ SIGQUIT
```

终端特殊按键

硬件异常

* 除0操作
* 访问非法内存

```
int kill(pid_t pid, int sig)
        pid > 0
            sig发送给ID为pid的进程
        pid  ==  0
            sig发送给与发送进程同组的所有进程
        pid  <  0
            sig发送给组ID为|-pid|的进程，并且发送进程具有向其发送信号的权限
        pid  ==  -1
            sig发送给发送进程有权限向他们发送信号的系统上的所有进程
    sig为0时，用于检测，特定为pid进程是否存在，如不存在，返回-1。
```

kill函数或kill命令 不过，kill向调用者返回测试结果时，原来存在的被测试进程可能刚终止

```
                int raise(int sig)
                void abort(void)
```

某种软件条件已发生 定时器alarm到时,每个进程只有一个定时器

```
unsigned int alarm(unsigned int seconds)
```

例：

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

int main(void)
{
    int counter;
    alarm(1);
    for(counter=0; 1; counter++)
        printf("counter=%d ", counter);
    return 0;
}
```

管道读端关闭，写端写数据

5.1.4 信号产生原因

```
1）SIGHUP:当用户退出shell时，由该shell启动的所有进程将收到这个信号，默认动作为终止进程
2）SIGINT：当用户按下了<Ctrl+C>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号。默认动作为终止里程。
```

3）SIGQUIT：当用户按下<ctrl+\>组合键时产生该信号，用户终端向正在运行中的由该终端启动的程序发出些信号。默认动作为终止进程。
4）SIGILL：CPU检测到某进程执行了非法指令。默认动作为终止进程并产生core文件
5）SIGTRAP：该信号由断点指令或其他 trap指令产生。默认动作为终止里程 并产生core文件。
6 ）SIGABRT:调用abort函数时产生该信号。默认动作为终止进程并产生core文件。
7）SIGBUS：非法访问内存地址，包括内存对齐出错，默认动作为终止进程并产生core文件。
8）SIGFPE：在发生致命的运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为0等所有的算法错误。默认动作为终止进程并产生core文件。
9）SIGKILL：无条件终止进程。本信号不能被忽略，处理和阻塞。默认动作为终止进程。它向系统管理员提供了可以杀死任何进程的方法。
10）SIGUSE1：用户定义 的信号。即程序员可以在程序中定义并使用该信号。默认动作为终止进程。
11）SIGSEGV：指示进程进行了无效内存访问。默认动作为终止进程并产生core文件。
12）SIGUSR2：这是另外一个用户自定义信号 ，程序员可以在程序中定义 并使用该信号。默认动作为终止进程。1
13）SIGPIPE：Broken pipe向一个没有读端的管道写数据。默认动作为终止进程。
14）SIGALRM:定时器超时，超时的时间 由系统调用alarm设置。默认动作为终止进程。
15）SIGTERM：程序结束信号，与SIGKILL不同的是，该信号可以被阻塞和终止。通常用来要示程序正常退出。执行shell命令Kill时，缺省产生这个信号。默认动作为终止进程。
16）SIGCHLD：子进程结束时，父进程会收到这个信号。默认动作为忽略这个信号。
17）SIGCONT：停止进程的执行。信号不能被忽略，处理和阻塞。默认动作为终止进程。
18）SIGTTIN：后台进程读终端控制台。默认动作为暂停进程。
19）SIGTSTP：停止进程的运行。按下<ctrl+z>组合键时发出这个信号。默认动作为暂停进程。
21）SIGTTOU:该信号类似于SIGTTIN，在后台进程要向终端输出数据时发生。默认动作为暂停进程。
22）SIGURG：套接字上有紧急数据时，向当前正在运行的进程发出些信号，报告有紧急数据到达。如网络带外数据到达，默认动作为忽略该信号。
23）SIGXFSZ：进程执行时间超过了分配给该进程的CPU时间 ，系统产生该信号并发送给该进程。默认动作为终止进程。
24）SIGXFSZ：超过文件的最大长度设置。默认动作为终止进程。
25）SIGVTALRM：虚拟时钟超时时产生该信号。类似于SIGALRM，但是该信号只计算该进程占用CPU的使用时间。默认动作为终止进程。
26）SGIPROF：类似于SIGVTALRM，它不公包括该进程占用CPU时间还包括执行系统调用时间。默认动作为终止进程。
27）SIGWINCH：窗口变化大小时发出。默认动作为忽略该信号。
28）SIGIO：此信号向进程指示发出了一个异步IO事件。默认动作为忽略。
29）SIGPWR：关机。默认动作为终止进程。
30）SIGSYS：无效的系统调用。默认动作为终止进程并产生core文件。
31）SIGRTMIN～（64）SIGRTMAX：LINUX的实时信号，它们没有固定的含义（可以由用户自定义）。所有的实时信号的默认动作都为终止进程。

5.2　进程处理信号行为

manpage里信号3种处理方式：

```
SIG_IGN
SIG_DFL
a signal handling function
```

进程处理信号的行为：

```
1.默认处理动作
        term
```

```
        core
        gcc -g file.c
                ulimit  -c  1024
                gdb  a.out  core
                进程死之前的内存情况，死后验尸
        ign
        stop
        cont
2.忽略
3.捕捉（用户自定义信号处理函数）
```

5.3 信号集处理函数

```
sigset_t为信号集,可sizeof(sigset_t)察看

int sigemptyset(sigset_t *set)
int sigfillset(sigset_t *set)
int sigaddset(sigset_t *set, int signo)
int sigdelset(sigset_t *set, int signo)
int sigismember(const sigset_t *set, int signo)
```

5.4 PCB的信号集

信号在内核中的表示示意图,画图

如果在进程解除对某信号的阻塞之前这种信号产生过多次，将如何处理？POSIX.1允许系统递送该信号一次或多次。Linux是这样实现的：常规信号在递达之前产生多次只计一次，而实时信号在递达之前产生多次可以依次放在一个队列里。本章不讨论实时信号。从上图来看，每个信号只有一个bit的未决标志，非0即1，不记录该信号产生了多少次，阻塞标志也是这样表示的。因此，未决和阻塞标志可以用相同的数据类型sigset_t来存储，sigset_t称为信号集，这个类型可以表示每个信号的“有效”或“无效”状态，在阻塞信号集中“有效”和“无效”的含义是该信号是否被阻塞，而在未决信号集中“有效”和“无效”的含义是该信号是否处于未决状态。

阻塞信号集也叫做当前进程的信号屏蔽字（Signal Mask），这里的“屏蔽”应该理解为阻塞而不是忽略。

5.4.1 sigprocmask

调用函数sigprocmask可以读取或更改进程的信号屏蔽字。

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
返回值：若成功则为0，若出错则为-1
```

如果oset是非空指针，则读取进程的当前信号屏蔽字通过oset参数传出。如果set是非空指针，则更改进程的信号屏蔽字，参数how指示如何更改。如果oset和set都是非空指针，则先将原来的信号屏蔽字备份到oset里，然后根据set和how参数更改信号屏蔽字。假设当前的信号屏蔽字为mask，下表说明了how参数的可选值。

how参数的含义

```
SIG_BLOCK       set包含了我们希望添加到当前信号屏蔽字的信号，相当于mask=mask|set
SIG_UNBLOCK       set包含了我们希望从当前信号屏蔽字中解除阻塞的信号，相当于mask=mask&~set
SIG_SETMASK       设置当前信号屏蔽字为set所指向的值，相当于mask=set
```

如果调用sigprocmask解除了对当前若干个未决信号的阻塞，则在sigprocmask返回前，至少将其中一个信号递达。

5.4.2 sigpending

```
#include <signal.h>

int sigpending(sigset_t *set);
```

sigpending读取当前进程的未决信号集，通过set参数传出。调用成功则返回0，出错则返回-1。

下面用刚学的几个函数做个实验。程序如下：

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

void printsigset(const sigset_t *set)
{
    int i;
    for (i = 1; i < 32; i++)
        if (sigismember(set, i) == 1)
            putchar('1');
        else
            putchar('0');
    puts("");
}

int main(void)
{
    sigset_t s, p;
    sigemptyset(&s);
    sigaddset(&s, SIGINT);
    sigprocmask(SIG_BLOCK, &s, NULL);
    while (1) {
        sigpending(&p);
        printsigset(&p);
        sleep(1);
```

```
    }
    return 0;
}
```

程序运行时，每秒钟把各信号的未决状态打印一遍，由于我们阻塞了SIGINT信号，按Ctrl-C将会使SIGINT信号处于未决状态，按Ctrl-\仍然可以终止程序，因为SIGQUIT信号没有阻塞。

```
xingwenpeng@ubuntu:~$ ./a.out
00000000000000000000000000000000
00000000000000000000000000000000（这时按Ctrl-C）
01000000000000000000000000000000
01000000000000000000000000000000（这时按Ctrl-\）
Quit (core dumped)
```

5.5 信号捕捉设定

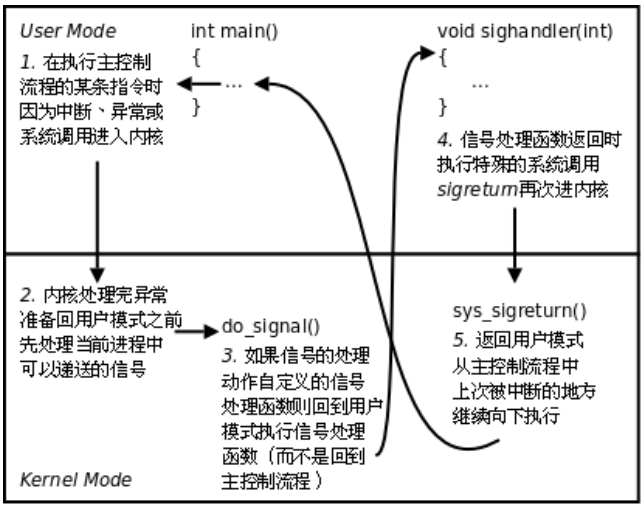

图 5.1: 信号捕捉

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
                struct sigaction *oldact);

struct sigaction 定义:

       struct sigaction {
           void     (*sa_handler)(int);
           void     (*sa_sigaction)(int, siginfo_t *, void *);
           sigset_t   sa_mask;
           int        sa_flags;
           void     (*sa_restorer)(void);
       };

sa_handler : 早期的捕捉函数
```

sa_sigaction ：新添加的捕捉函数，可以传参 ，和sa_handler互斥，两者通过sa_flags选择采用哪种捕捉函数
sa_mask ：在执行捕捉函数时，设置阻塞其它信号，sa_mask | 进程阻塞信号集，退出捕捉函数后，还原回原有的阻塞信号集
sa_flags ： SA_SIGINFO 或者 0
sa_restorer ：保留，已过时

举例SIGINT被捕捉：
当前进程从内核返回用户空间代码前检查是否有信号递达，有则去响应

5.5.1 利用SIGUSR1和SIGUSR2实现父子进程同步输出

注意：子进程继承了父进程的信号屏蔽字和信号处理动作

5.6 C标准库信号处理函数

```
typedef void (*sighandler_t)(int)
sighandler_t signal(int signum, sighandler_t handler)

int system(const char *command)
集合fork, exec, wait一体
```

5.6.1 signal

5.7 可重入函数

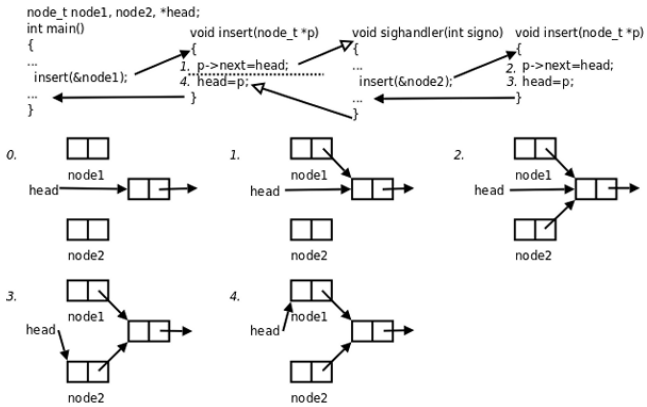

图 5.2: 不可重入函数

* 不含全局变量和静态变量是可重入函数的一个要素
* 可重入函数见man 7 signal
* 在信号捕捉函数里应使用可重入函数

* 在信号捕捉函数里禁止调用不可重入函数

例如：strtok就是一个不可重入函数，因为strtok内部维护了一个内部静态指针，保存上一次切割到的位置，如果信号的捕捉函数中也去调用strtok函数，则会造成切割字符串混乱，应用strtok_r版本，r表示可重入。

5.8 信号引起的竞态和异步I/O

5.8.1 时序竞态

```
int pause(void)
        使调用进程挂起，直到有信号递达，如果递达信号是忽略，则继续挂起
int sigsuspend(const sigset_t *mask)
        1.以通过指定mask来临时解除对某个信号的屏蔽，
2.然后挂起等待，
3.当被信号唤醒sigsuspend返回时，进程的信号屏蔽字恢复为原来的值
```

mysleep实现,这种实现方式是否存在BUG？

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

void sig_alrm(int signo)
{
    /* nothing to do */
}

unsigned int mysleep(unsigned int nsecs)
{
    struct sigaction newact, oldact;
    unsigned int unslept;

    newact.sa_handler = sig_alrm;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    sigaction(SIGALRM, &newact, &oldact);

    alarm(nsecs);
    pause();

    unslept = alarm(0);
    sigaction(SIGALRM, &oldact, NULL);

    return unslept;
}

int main(void)
{
    while(1){
```

```
        mysleep(2);
        printf("Two seconds passed\n");
    }
    return 0;
}
```

mysleep改进版

```
unsigned int mysleep(unsigned int nsecs)
{
    struct sigaction    newact, oldact;
    sigset_t            newmask, oldmask, suspmask;
    unsigned int        unslept;

    /* set our handler, save previous information */
    newact.sa_handler = sig_alrm;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    sigaction(SIGALRM, &newact, &oldact);

    /* block SIGALRM and save current signal mask */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGALRM);
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);

    alarm(nsecs);

    suspmask = oldmask;
    sigdelset(&suspmask, SIGALRM);    /* make sure SIGALRM isn't blocked */
    sigsuspend(&suspmask);            /* wait for any signal to be caught */

    /* some signal has been caught,   SIGALRM is now blocked */

    unslept = alarm(0);
    sigaction(SIGALRM, &oldact, NULL);  /* reset previous action */

    /* reset signal mask, which unblocks SIGALRM */
    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    return(unslept);
}
```

5.8.2 全局变量异步I/O

5.8.3 可重入函数

```
1.不含全局变量和静态变量是可重入函数的一个要素
2.可重入函数见man  7  signal
3.在信号捕捉函数里应使用可重入函数
```

5.8.4 避免异步I/O的类型

```
sig_atomic_t
        平台下的原子类型
volatile
        防止编译器开启优化选项时，优化对内存的读写
```

5.9 SIGCHLD信号处理

5.9.1 SIGCHLD的产生条件

```
子进程终止时
子进程接收到SIGSTOP信号停止时
子进程处在停止态，接受到SIGCONT后唤醒时
```

代码实现

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

void sys_err(char *str)
{
    perror(str);
    exit(1);
}
void do_sig_child(int signo)
{
    int status;
    pid_t pid;
    while ((pid = waitpid(0, &status, WNOHANG)) > 0) {
        if (WIFEXITED(status))
            printf("child %d exit %d\n", pid, WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
            printf("child %d cancel signal %d\n", pid, WTERMSIG(status));
    }

}
int main(void)
{
    pid_t pid;
    int i;
    //阻塞SIGCHLD
    for (i = 0; i < 10; i++) {
```

```
        if ((pid = fork()) == 0)
            break;
        else if (pid < 0)
            sys_err("fork");
    }
    if (pid == 0) {
        int n = 18;
        while (n--) {
            printf("child ID %d\n", getpid());
            sleep(1);
        }
        return i;
    }
    else if (pid > 0) {
        //先设置捕捉
        //再解除对SIGCHLD的阻塞
        struct sigaction act;

        act.sa_handler = do_sig_child;
        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;
        sigaction(SIGCHLD, &act, NULL);
        while (1) {
            printf("Parent ID %d\n", getpid());
            sleep(1);
        }
    }

    return 0;
}
```

5.9.2 status处理方式

```
pid_t waitpid(pid_t pid, int *status, int options)
        options
                WNOHANG
                        没有子进程结束，立即返回
                WUNTRACED
                        如果子进程由于被停止产生的SIGCHLD， waitpid则立即返回
                WCONTINUED
                        如果子进程由于被SIGCONT唤醒而产生的SIGCHLD， waitpid则立即返回
        获取status
                WIFEXITED(status)
                        子进程正常exit终止，返回真
                                WEXITSTATUS(status)返回子进程正常退出值
                WIFSIGNALED(status)
                        子进程被信号终止，返回真
                                WTERMSIG(status)返回终止子进程的信号值
                WIFSTOPPED(status)
                        子进程被停止，返回真
                                WSTOPSIG(status)返回停止子进程的信号值
                WIFCONTINUED(status)
```

```
                    子进程由停止态转为就绪态，返回真
```

见man 2 wait 练习

5.10 向信号捕捉函数传参

5.10.1 sigqueue

```
    int sigqueue(pid_t pid, int sig, const union sigval value)
    union sigval {
       int   sival_int;
       void *sival_ptr;
    };
```

5.10.2 sigaction

```
    void     (*sa_sigaction)(int, siginfo_t *, void *)
    siginfo_t {
                    int      si_int;      /* POSIX.1b signal */
                    void    *si_ptr;      /* POSIX.1b signal */
                    sigval_t si_value;    /* Signal value */
                     ...
}

    sa_flags = SA_SIGINFO
```

实例

* 进程自己收发信号，在同一地址空间
* 不同进程间收发信号，不在同一地址空间,不适合传地址

5.11 信号中断系统调用

read阻塞时，信号中断系统调用:
1.返回部分读到的数据
2.read调用失败，errno设成EINTER

5.12 练习

1.当进程处理SIGINT信号时，临时阻塞SIGQUIT信号。 2.进程间利用信号传参，来控制数据同步，如两个进程交叉报数。

第 6 章

进程间关系

6.1 终端

在UNIX系统中，用户通过终端登录系统后得到一个Shell进程，这个终端成为Shell进程的控制终端（Controlling Terminal），在讲进程时讲过，控制终端是保存在PCB中的信息，而我们知道fork会复制PCB中的信息，因此由Shell进程启动的其它进程的控制终端也是这个终端。默认情况下（没有重定向），每个进程的标准输入、标准输出和标准错误输出都指向控制终端，进程从标准输入读也就是读用户的键盘输入，进程往标准输出或标准错误输出写也就是输出到显示器上。信号中还讲过，在控制终端输入一些特殊的控制键可以给前台进程发信号，例如Ctrl-C表示SIGINT，Ctrl-\表示SIGQUIT。

```
init-->fork-->exec-->getty-->用户输入帐号-->login-->输入密码-->exec-->shell
```

文件与I/O中讲过，每个进程都可以通过一个特殊的设备文件/dev/tty访问它的控制终端。事实上每个终端设备都对应一个不同的设备文件，/dev/tty提供了一个通用的接口，一个进程要访问它的控制终端既可以通过/dev/tty也可以通过该终端设备所对应的设备文件来访问。ttyname函数可以由文件描述符查出对应的文件名，该文件描述符必须指向一个终端设备而不能是任意文件。下面我们通过实验看一下各种不同的终端所对应的设备文件名。

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

int main()
{
    printf("fd 0: %s\n", ttyname(0));
    printf("fd 1: %s\n", ttyname(1));
    printf("fd 2: %s\n", ttyname(2));
    return 0;
}
```

硬件驱动程序负责读写实际的硬件设备，比如从键盘读入字符和把字符输出到显示器，线路规程像一个过滤器，对于某些特殊字符并不是让它直接通过，而是做特殊处理，比如在键盘上按下Ctrl-Z，对应的字符并不会被用户程序的read读到，而是被线路规程截获，解释

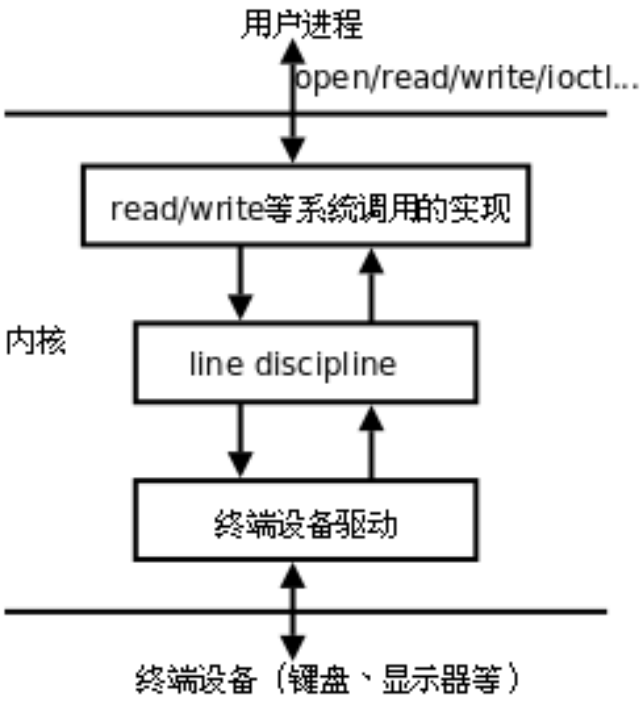

图 6.1: 终端设备模块

成SIGTSTP信号发给前台进程，通常会使该进程停止。线路规程应该过滤哪些字符和做哪些特殊处理是可以配置的。

6.1.1 网络终端

虚拟终端或串口终端的数目是有限的，虚拟终端(字符控制终端)一般就是/dev/tty1～/dev/tty6六个，串口终端的数目也不超过串口的数目。然而网络终端或图形终端窗口的数目却是不受限制的，这是通过伪终端（Pseudo TTY）实现的。一套伪终端由一个主设备（PTY Master）和一个从设备（PTY Slave）组成。主设备在概念上相当于键盘和显示器，只不过它不是真正的硬件而是一个内核模块，操作它的也不是用户而是另外一个进程。从设备和上面介绍的/dev/tty1这样的终端设备模块类似，只不过它的底层驱动程序不是访问硬件而是访问主设备。网络终端或图形终端窗口的Shell进程以及它启动的其它进程都会认为自己的控制终端是伪终端从设备，例如/dev/pts/0、/dev/pts/1等。下面以telnet为例说明网络登录和使用伪终端的过程。

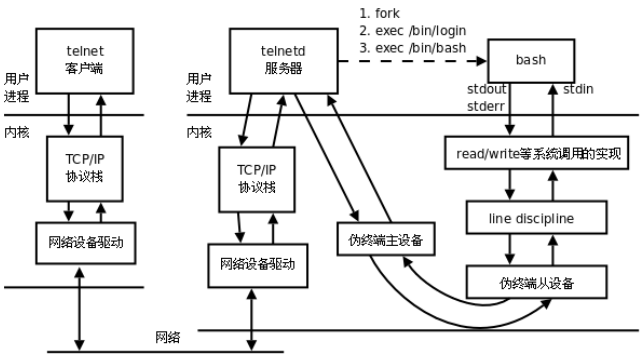

图 6.2: 网络终端

如果telnet客户端和服务器之间的网络延迟较大，我们会观察到按下一个键之后要过几

秒钟才能回显到屏幕上。这说明我们每按一个键telnet客户端都会立刻把该字符发送给服务器，然后这个字符经过伪终端主设备和从设备之后被Shell进程读取，同时回显到伪终端从设备，回显的字符再经过伪终端主设备、telnetd服务器和网络发回给telnet客户端，显示给用户看。也许你会觉得吃惊，但真的是这样：每按一个键都要在网络上走个来回！

6.2 进程组

一个或多个进程的集合,进程组ID是一个正整数。 用来获得当前进程进程组ID的函数

```
pid_t getpgid(pid_t pid)
pid_t getpgrp(void)
```

获得父子进程进程组

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

int main(void)
{
    pid_t pid;

    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }else if (pid == 0) {
        printf("child process PID is %d\n",getpid());
        printf("Group ID is %d\n",getpgrp());
        printf("Group ID is %d\n",getpgid(0));
        printf("Group ID is %d\n",getpgid(getpid()));
        exit(0);
    }

    sleep(3);
    printf("parent process PID is %d\n",getpid());
    printf("Group ID is %d\n",getpgrp());

    return 0;
}
```

组长进程标识:其进程组ID==其进程ID

组长进程可以创建一个进程组，创建该进程组中的进程，然后终止,只要进程组中有一个进程存在，进程组就存在，与组长进程是否终止无关

进程组生存期:进程组创建到最后一个进程离开(终止或转移到另一个进程组)

一个进程可以为自己或子进程设置进程组ID

```
int setpgid(pid_t pid, pid_t pgid)
        如改变子进程为新的组，应在fork后，exec前使用
        非root进程只能改变自己创建的子进程，或有权限操作的进程
```

setpgid()加入一个现有的进程组或创建一个新进程组,如改变父子进程为新的组

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

int main(void)
{
    pid_t pid;

    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    } else if (pid == 0) {
        printf("child process PID is %d\n",getpid());
        printf("Group ID of child is %d\n",getpgid(0)); // 返回组id
        sleep(5);
        printf("Group ID of child is changed to %d\n",getpgid(0));
        exit(0);
    }

    sleep(1);
    setpgid(pid,pid); // 父进程改变子进程的组id为子进程本身

    sleep(5);
    printf("parent process PID is %d\n",getpid());
    printf("parent of parent process PID is %d\n",getppid());
    printf("Group ID of parent is %d\n",getpgid(0));
    setpgid(getpid(),getppid()); // 改变父进程的组id为父进程的父进程
    printf("Group ID of parent is changed to %d\n",getpgid(0));

    return 0;
}
```

6.3 会话

```
pid_t setsid(void)
```

1.调用进程不能是进程组组长,该进程变成新会话首进程(session header) 2.该进程成为一个新进程组的组长进程。 3.需有root权限(ubuntu不需要) 4.新会话丢弃原有的控制终端,该会话没有控制终端 5.该调用进程是组长进程，则出错返回 6.建立新会话时，先调用fork, 父进程终止，子进程调用

```
pid_t getsid(pid_t pid)
```

pid为0表示察看当前进程session ID

ps ajx命令查看系统中的进程。参数a表示不仅列当前用户的进程，也列出所有其他用户的进程，参数x表示不仅列有控制终端的进程，也列出所有无控制终端的进程，参数j表示列出与作业控制相关的信息。

组长进程不能成为新会话首进程，新会话首进程必定会成为组长进程。

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

int main(void)
{
    pid_t pid;

    if ((pid = fork())<0) {
        perror("fork");
        exit(1);
    } else if (pid == 0) {
        printf("child process PID is %d\n", getpid());
        printf("Group ID of child is %d\n", getpgid(0));
        printf("Session ID of child is %d\n", getsid(0));
        sleep(10);
        setsid(); // 子进程非组长进程，故其成为新会话首进程，且成为组长进程。该进程组id即为会话进程
        printf("Changed:\n");
        printf("child process PID is %d\n", getpid());
        printf("Group ID of child is %d\n", getpgid(0));
        printf("Session ID of child is %d\n", getsid(0));
        sleep(20);
        exit(0);
    }

    return 0;
}
```

第 7 章

守护进程

7.1 概念

Daemon(精灵)进程,是Linux中的后台服务进程,生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。

7.2 模型

守护进程编程步骤

```
1. 创建子进程，父进程退出
    所有工作在子进程中进行
    形式上脱离了控制终端
2. 在子进程中创建新会话
    setsid()函数
    使子进程完全独立出来，脱离控制
3. 改变当前目录为根目录
    chdir()函数
    防止占用可卸载的文件系统
    也可以换成其它路径
4. 重设文件权限掩码
    umask()函数
    防止继承的文件创建屏蔽字拒绝某些权限
    增加守护进程灵活性
5. 关闭文件描述符
    继承的打开文件不会用到，浪费系统资源，无法卸载
6. 开始执行守护进程核心工作
7. 守护进程退出处理
```

代码模型

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

void daemonize(void)
```

```
{
    pid_t  pid;

    /*
     * 成为一个新会话的首进程，失去控制终端
     */
    if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    } else if (pid != 0) /* parent */
        exit(0);
    setsid();

    /*
     * 改变当前工作目录到/目录下.
     */
    if (chdir("/") < 0) {
        perror("chdir");
        exit(1);
    }
    /* 设置umask为0 */
    umask(0);
    /*
     * 重定向0，1，2文件描述符到 /dev/null，因为已经失去控制终端，再操作0，1，2没有意义.
     */
    close(0);
    open("/dev/null", O_RDWR);
    dup2(0, 1);
    dup2(0, 2);
}

int main(void)
{
    daemonize();
    while(1);          /* 在此循环中可以实现守护进程的核心工作 */
}
```

运行这个程序，它变成一个守护进程，不再和当前终端关联。用ps命令看不到，必须运行带x参数的ps命令才能看到。另外还可以看到，用户关闭终端窗口或注销也不会影响守护进程的运行。

思考：守护进程为什么要和控制终端脱离？

7.3　习题

1.每隔10s在/tmp/dameon.log中写入当前时间

2.编写监控/home/usr/目录下文件创建与更改的守护进程，日志文件放在/home/usr/filechangelog

第 8 章

线程

8.1 线程概念

8.1.1 什么是线程

8.1.2 线程和进程的关系

1.轻量级进程(light-weight process)，也有PCB,创建线程使用的底层函数和进程一样，都是clone

2.从内核里看进程和线程是一样的，都有各自不同的PCB，但是PCB中指向内存资源的三级页表是相同的

3.进程可以蜕变成线程

4.在美国人眼里，线程就是寄存器和栈

5.在linux下，线程最是小的执行单位；进程是最小的分配资源单位

察看LWP号

```
ps -Lf  pid
ps -eLf
```

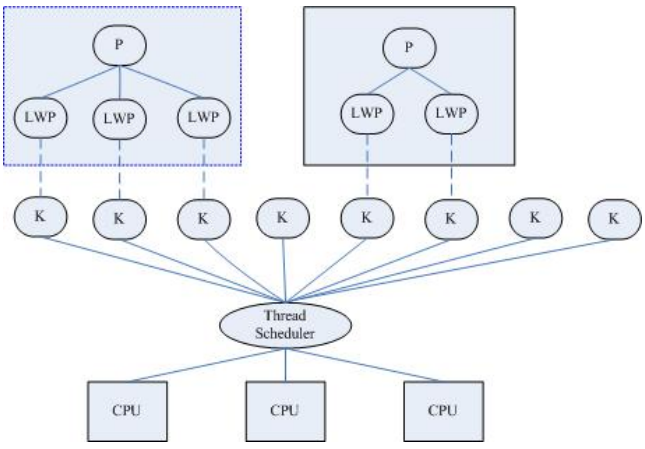

图 8.1: 调度单位为线程

8.1.3 线程间共享资源

1.文件描述符表
2.每种信号的处理方式
3.当前工作目录
4.用户ID和组ID
5.内存地址空间

```
Text
data
bss
堆
共享库
```

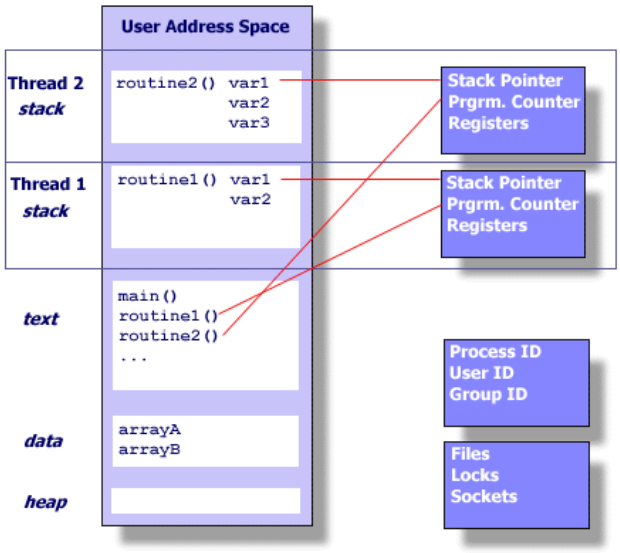

图 8.2: 线程间共享资源

8.1.4 线程间非共享资源

1.线程id
2.处理器现场和栈指针(内核栈)
3.独立的栈空间(用户空间栈)
4.errno变量
5.信号屏蔽字
6.调度优先级

8.1.5 线程优缺点

优点

```
提高程序的并发性
开销小，不用重新分配内存
通信和共享数据方便
```

缺点

```
线程不稳定（库函数实现）
线程调试比较困难（gdb支持不好）
线程无法使用unix经典事件，例如信号
```

8.1.6 pthread manpage

查看manpage关于pthread的函数

```
man -k pthread
```

安装pthread相关manpage

```
sudo apt-get install manpages-posix manpages-posix-dev
```

8.2 线程原语

8.2.1 pthread_create

创建线程

```
#include <pthread.h>

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

pthread_t *thread:传递一个pthread_t变量地址进来，用于保存新线程的tid（线程ID）
const pthread_attr_t *attr:线程属性设置，如使用默认属性，则传NULL
void *(*start_routine) (void *):函数指针，指向新线程应该加载执行的函数模块
void *arg:指定线程将要加载调用的那个函数的参数
返回值:成功返回0，失败返回错误号。以前学过的系统函数都是成功返回0，失败返回-1，而错误号保存在全局变量errno中，而pthread库的函数都是通过返回值返回错误号，虽然每个线程也都有一个errno，但这是为了兼容其它函数接口而提供的，pthread库本身并不使用它，通过返回值返回错误码更加清晰。

Compile and link with -lpthread.

typedef unsigned long int pthread_t;
```

在一个线程中调用pthread_create()创建新的线程后，当前线程从pthread_create()返回继续往下执行，而新的线程所执行的代码由我们传给pthread_create的函数指针start_routine决定。start_routine函数接收一个参数，是通过pthread_create的arg参数传递给它的，该参数的类型为void *，这个指针按什么类型解释由调用者自己定义。start_routine的返回值类型也是void *，这个指针的含义同样由调用者自己定义。start_routine返回时，这个线程就退出了，其它线程可以调用pthread_join得到start_routine的返回值，类似于父进程调用wait(2)得到子进程的退出状态，稍后详细介绍pthread_join。

pthread_create成功返回后，新创建的线程的id被填写到thread参数所指向的内存单元。我们知道进程id的类型是pid_t，每个进程的id在整个系统中是唯一的，调用getpid(2)可以获得当前进程的id，是一个正整数值。线程id的类型是thread_t，它只在当前进程中保证是唯一的，在不同的系统中thread_t这个类型有不同的实现，它可能是一个整数值，也可能是一个结构体，也可能是一个地址，所以不能简单地当成整数用printf打印，调用pthread_self(3)可以获得当前线程的id。

attr参数表示线程属性，本节不深入讨论线程属性，所有代码例子都传NULL给attr参数，表示线程属性取缺省值，感兴趣的读者可以参考[APUE2e]。

8.2.2 pthread_self

获取调用线程tid

```
#include <pthread.h>

pthread_t pthread_self(void);
```

思考：pthread_self获得的tid和pthread_create函数里得到tid是否会出现不一致？

首先看一个简单的例子：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

pthread_t ntid;

void printids(const char *s)
{
    pid_t      pid;
    pthread_t  tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
           (unsigned int)tid, (unsigned int)tid);
}

void *thr_fn(void *arg)
{
    printids(arg);
```

```
    return NULL;
}

int main(void)
{
    int err;

    err = pthread_create(&ntid, NULL, thr_fn, "new thread: ");
    if (err != 0) {
        fprintf(stderr, "can't create thread: %s\n", strerror(err));
        exit(1);
    }
    printids("main thread:");
    sleep(1);

    return 0;
}
```

由于pthread_create的错误码不保存在errno中，因此不能直接用perror(3)打印错误信息，可以先用strerror(3)把错误码转换成错误信息再打印。

如果任意一个线程调用了exit或_exit，则整个进程的所有线程都终止，由于从main函数return也相当于调用exit，为了防止新创建的线程还没有得到执行就终止，我们在main函数return之前延时1秒，这只是一种权宜之计，即使主线程等待1秒，内核也不一定会调度新创建的线程执行，下一节我们会看到更好的办法。

8.2.3 pthread_exit

调用线程退出函数，注意和exit函数的区别，任何线程里exit导致进程退出，其他线程未工作结束，主控线程退出时不能return或exit。

需要注意，pthread_exit或者return返回的指针所指向的内存单元必须是全局的或者是用malloc分配的，不能在线程函数的栈上分配，因为当其它线程得到这个返回指针时线程函数已经退出了。

```
#include <pthread.h>

void pthread_exit(void *retval);
void *retval:线程退出时传递出的参数，可以是退出值或地址，如是地址时，不能是线程内部申请的局部地址。
```

8.2.4 pthread_join

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);

pthread_t thread:回收线程的tid
void **retval:接收退出线程传递出的返回值
返回值：成功返回0，失败返回错误号
```

调用该函数的线程将挂起等待，直到id为thread的线程终止。thread线程以不同的方法终止，通过pthread_join得到的终止状态是不同的，总结如下：

如果thread线程通过return返回，retval所指向的单元里存放的是thread线程函数的返回值。

如果thread线程被别的线程调用pthread_cancel异常终止掉，retval所指向的单元里存放的是常数PTHREAD_CANCELED。

如果thread线程是自己调用pthread_exit终止的，retval所指向的单元存放的是传给pthread_exit的参数。

如果对thread线程的终止状态不感兴趣，可以传NULL给retval参数。

8.2.5 pthread_cancel

在进程内某个线程可以取消另一个线程。

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

被取消的线程，退出值，定义在Linux的pthread库中常数PTHREAD_CANCELED的值是-1。可以在头文件pthread.h中找到它的定义：

```
#define PTHREAD_CANCELED ((void *) -1)
```

例题：

实例：

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

void *thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return (void *)1;
}

void *thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}

void *thr_fn3(void *arg)
{
    while(1) {
        printf("thread 3 writing\n");
```

```
        sleep(1);
    }
}

int main(void)
{
    pthread_t   tid;
    void        *tret;

    pthread_create(&tid, NULL, thr_fn1, NULL);
    pthread_join(tid, &tret);
    printf("thread 1 exit code %d\n", (int)tret);

    pthread_create(&tid, NULL, thr_fn2, NULL);
    pthread_join(tid, &tret);
    printf("thread 2 exit code %d\n", (int)tret);

    pthread_create(&tid, NULL, thr_fn3, NULL);
    sleep(3);
    pthread_cancel(tid);
    pthread_join(tid, &tret);
    printf("thread 3 exit code %d\n", (int)tret);

    return 0;
}
```

8.2.6 pthread_detach

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
pthread_t tid:分离线程tid
返回值：成功返回0，失败返回错误号。
```

一般情况下，线程终止后，其终止状态一直保留到其它线程调用pthread_join获取它的状态为止。但是线程也可以被置为detach状态，这样的线程一旦终止就立刻回收它占用的所有资源，而不保留终止状态。不能对一个已经处于detach状态的线程调用pthread_join，这样的调用将返回EINVAL。如果已经对一个线程调用了pthread_detach就不能再调用pthread_join了。

例题：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>

void *thr_fn(void *arg)
```

```
{
    int n = 3;
    while (n--) {
        printf("thread count %d\n", n);
        sleep(1);
    }
    return (void *)1;
}


int main(void)
{
    pthread_t   tid;
    void        *tret;
    int err;

    pthread_create(&tid, NULL, thr_fn, NULL);
    //第一次运行时注释掉下面这行，第二次再打开，分析两次结果
    pthread_detach(tid);

    while (1) {
        err = pthread_join(tid, &tret);
        if (err != 0)
            fprintf(stderr, "thread %s\n", strerror(err));
        else
            fprintf(stderr, "thread  exit code %d\n", (int)tret);
        sleep(1);
    }

    return 0;
}
```

8.2.7 pthread_equal

比较两个线程是否相等

```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

8.3 线程终止方式

如果需要只终止某个线程而不终止整个进程，可以有三种方法：

1.从线程主函数return。这种方法对主控线程不适用，从main函数return相当于调用exit。

2.一个线程可以调用pthread_cancel终止同一进程中的另一个线程。

3.线程可以调用pthread_exit终止自己。

同一进程的线程间，pthread_cancel向另一线程发终止信号。系统并不会马上关闭被取消线程，只有在被取消线程下次系统调用时，才会真正结束线程。或调用pthread_testcancel，让内核去检测是否需要取消当前线程

8.4 线程属性

本节作为指引性介绍，linux下线程的属性是可以根据实际项目需要，进行设置，之前我们讨论的线程都是采用线程的默认属性，默认属性已经可以解决绝大多数开发时遇到的问题。如我们对程序的性能提出更高的要求那么需要设置线程属性，比如可以通过设置线程栈的大小来降低内存的使用，增加最大线程个数。

```
typedef struct
{
    int                 etachstate;     //线程的分离状态
    int                 schedpolicy;    //线程调度策略
    structsched_param   schedparam;     //线程的调度参数
    int                 inheritsched;   //线程的继承性
    int                 scope;          //线程的作用域
    size_t              guardsize;      //线程栈末尾的警戒缓冲区大小
    int                 stackaddr_set;  //线程的栈设置
    void*               stackaddr;      //线程栈的位置
    size_t              stacksize;      //线程栈的大小
}pthread_attr_t;
```

注：目前线程属性在内核中不是直接这么定义的，抽象太深不宜拿出讲课，为方便大家理解，使用早期的线程属性定义，两者之间定义的主要元素差别不大。

属性值不能直接设置，须使用相关函数进行操作，初始化的函数为pthread_attr_init，这个函数必须在pthread_create函数之前调用。之后须用pthread_attr_destroy函数来释放资源。线程属性主要包括如下属性：作用域（scope）、栈尺寸（stack size）、栈地址（stack address）、优先级（priority）、分离的状态（detached state）、调度策略和参数（scheduling policy and parameters）。默认的属性为非绑定、非分离、缺省M的堆栈、与父进程同样级别的优先级。

8.4.1 线程属性初始化

先初始化线程属性，再pthread_create创建线程

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);       //初始化线程属性
int pthread_attr_destroy(pthread_attr_t *attr);    //销毁线程属性所占用的资源
```

8.4.2 线程的分离状态（detached state）

线程的分离状态决定一个线程以什么样的方式来终止自己。

非分离状态:线程的默认属性是非分离状态，这种情况下，原有的线程等待创建的线程结束。只有当pthread_join()函数返回时，创建的线程才算终止，才能释放自己占用的系统资源。

分离状态:分离线程没有被其他的线程所等待，自己运行结束了，线程也就终止了，马上释放系统资源。应该根据自己的需要，选择适当的分离状态。

线程分离状态的函数：

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);   //设置线程属性，分离or非分离
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);  //获取程属性，分离or非分离

pthread_attr_t *attr:被已初始化的线程属性
int *detachstate:可选为PTHREAD_CREATE_DETACHED（分离线程）和 PTHREAD _CREATE_JOINABLE（非分离线程）
```

这里要注意的一点是，如果设置一个线程为分离线程，而这个线程运行又非常快，它很可能在pthread_create函数返回之前就终止了，它终止以后就可能将线程号和系统资源移交给其他的线程使用，这样调用pthread_create的线程就得到了错误的线程号。要避免这种情况可以采取一定的同步措施，最简单的方法之一是可以在被创建的线程里调用pthread_cond_timedwait函数，让这个线程等待一会儿，留出足够的时间让函数pthread_create返回。设置一段等待时间，是在多线程编程里常用的方法。但是注意不要使用诸如wait()之类的函数，它们是使整个进程睡眠，并不能解决线程同步的问题。

8.4.3　线程的栈地址（stack address）

POSIX.1定义了两个常量_POSIX_THREAD_ATTR_STACKADDR 和_POSIX_THREAD_ATTR_STACKSIZE检测系统是否支持栈属性。也可以给sysconf函数传递_SC_THREAD_ATTR_STACKADDR或 _SC_THREAD_ATTR_STACKSIZE来进行检测。

当进程栈地址空间不够用时，指定新建线程使用由malloc分配的空间作为自己的栈空间。通过pthread_attr_setstackaddr和pthread_attr_getstackaddr两个函数分别设置和获取线程的栈地址。传给pthread_attr_setstackaddr函数的地址是缓冲区的低地址（不一定是栈的开始地址，栈可能从高地址往低地址增长）。

```
#include <pthread.h>

int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **stackaddr);

attr:       指向一个线程属性的指针
stackaddr:  返回获取的栈地址
返回值：若是成功返回0,否则返回错误的编号
说　明：函数已过时，一般用下面讲到的pthread_attr_getstack来代替
```

8.4.4 线程的栈大小（stack size）

当系统中有很多线程时，可能需要减小每个线程栈的默认大小，防止进程的地址空间不够用，当线程调用的函数会分配很大的局部变量或者函数调用层次很深时，可能需要增大线程栈的默认大小。

函数pthread_attr_getstacksize和 pthread_attr_setstacksize提供设置。

```
#include <pthread.h>

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);

attr       指向一个线程属性的指针
stacksize  返回线程的堆栈大小
返回值：若是成功返回0,否则返回错误的编号
```

除上述对栈设置的函数外，还有以下两个函数可以获取和设置线程栈属性

```
#include <pthread.h>

int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr, size_t stacksize);
int pthread_attr_getstack(pthread_attr_t *attr, void **stackaddr, size_t *stacksize);

attr       指向一个线程属性的指针
stackaddr  返回获取的栈地址
stacksize  返回获取的栈大小
返回值：若是成功返回0,否则返回错误的编号
```

8.4.5 线程属性控制实例

```
#include <stdio.h>
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#define SIZE 0x10000
int print_ntimes(char *str)
{
    sleep(1);
    printf("%s\n", str);
    return 0;
}
void *th_fun(void *arg)
{
    int n = 3;
    while (n--)
        print_ntimes("hello xwp\n");
}
int main(void)
```

```
{
    pthread_t tid;
    int err, detachstate, i = 1;
    pthread_attr_t attr;
    size_t stacksize;
    void *stackaddr;

    pthread_attr_init(&attr);

    pthread_attr_getstack(&attr, &stackaddr, &stacksize);
    printf("stackadd=%p\n", stackaddr);
    printf("stacksize=%x\n", (int)stacksize);

    pthread_attr_getdetachstate(&attr, &detachstate);
    if (detachstate == PTHREAD_CREATE_DETACHED)
        printf("thread detached\n");
    else if (detachstate == PTHREAD_CREATE_JOINABLE)
        printf("thread join\n");
    else
        printf("thread un known\n");
    /* 设置线程分离属性 */
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    while (1) {
        /* 在堆上申请内存，指定线程栈的起始地址和大小 */
        stackaddr = malloc(SIZE);
        if (stackaddr == NULL) {
            perror("malloc");
            exit(1);
        }
        stacksize = SIZE;
        pthread_attr_setstack(&attr, stackaddr, stacksize);

        err = pthread_create(&tid, &attr, th_fun, NULL);
        if (err != 0) {
            printf("%s\n", strerror(err));
            exit(1);
        }
        printf("%d\n", i++);
    }

    pthread_attr_destroy(&attr);
    return 0;
}
```

8.5 NPTL

1.察看当前pthread库版本

```
    getconf GNU_LIBPTHREAD_VERSION
```

2.NPTL实现机制(POSIX),Native POSIX Thread Library

3.使用线程库时gcc指定 -lpthread

8.6 细节注意

1.主线程退出其他线程不退出，主线程应调用ptrhed_exit

2.避免僵线程

```
join
pthread_deatch
pthread_create指定分离属性
被join线程可能在join函数返回前就释放完自己的所有内存资源，所以不应当返回被回收线程栈中的值;
```

3.malloc和mmap申请的内存可以被其他线程释放

4.如果线程终止时没有释放加锁的互斥量，则该互斥量不能再被使用

5.应避免在多线程模型中调用fork除非，马上exec，子进程中只有调用fork的线程存在，其他线程在子进程中均pthread_exit

6.信号的复杂语义很难和多线程共存，应避免在多线程引入信号机制

8.7 练习

1.测试当前系统允许创建的最大线程个数

2.多线程拷贝命令,如：./my_cp srcfile destfile N（拷贝线程个数）

```
考察点:
    mmap
    lseek拓展一个文件,write一个字节，使文件真正拓展
    多线程编程模型
    线程控制原语
```

3.多线程检索，改写之前的开房数据查询系统

第 9 章

线程同步

多个线程同时访问共享数据时可能会冲突，这跟前面讲信号时所说的可重入性是同样的问题。比如两个线程都要把某个全局变量增加1，这个操作在某平台需要三条指令完成：

从内存读变量值到寄存器

寄存器的值加1

将寄存器的值写回内存

假设两个线程在多处理器平台上同时执行这三条指令，则可能导致下图所示的结果，最后变量只加了一次而非两次。

CPU1执行 线程A的指令	CPU2执行 线程A的指令	变量i的内存 单元的值
mov 0x8049540, %eax (eax = 5)	其它指令	5
add $0x1, %eax (eax = 6)	mov 0x8049540, %eax (eax = 5)	5
mov %eax, 0x8049540 (eax = 6)	add $0x1, %eax (eax = 6)	6
其它指令	mov %eax, 0x8049540 (eax = 6)	6

图 9.1: 并行访问冲突

实例：

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

#define NLOOP 5000
int counter;                /* incremented by threads */

void *doit(void *);

int main(int argc, char **argv)
{
    pthread_t tidA, tidB;

    pthread_create(&tidA, NULL, &doit, NULL);
```

```
    pthread_create(&tidB, NULL, &doit, NULL);

        /* wait for both threads to terminate */
    pthread_join(tidA, NULL);
    pthread_join(tidB, NULL);

    return 0;
}

void *doit(void *vptr)
{
    int    i, val;

    for (i = 0; i < NLOOP; i++) {
        val = counter;
        printf("%x: %d\n", (unsigned int)pthread_self(), val + 1);
        counter = val + 1;
    }

    return NULL;
}
```

我们创建两个线程，各自把counter增加5000次，正常情况下最后counter应该等于10000，但事实上每次运行该程序的结果都不一样，有时候数到5000多，有时候数到6000多。

9.1　线程为什么要同步

1.共享资源，多个线程都可对共享资源操作
2.线程操作共享资源的先后顺序不确定
3.处理器对存储器的操作一般不是原子操作

9.2　互斥量

mutex操作原语

```
    pthread_mutex_t
    pthread_mutex_init
    pthread_mutex_destroy
    pthread_mutex_lock
    pthread_mutex_trylock
    pthread_mutex_unlock
```

9.2.1　临界区（Critical Section）

保证在某一时刻只有一个线程能访问数据的简便办法。在任意时刻只允许一个线程对共享资源进行访问。如果有多个线程试图同时访问临界区，那么 在有一个线程进入后其他所

有试图访问此临界区的线程将被挂起，并一直持续到进入临界区的线程离开。临界区在被释放后，其他线程可以继续抢占，并以此达到用原子方式操作共享资源的目的。

9.2.2 临界区的选定

临界区的选定因尽可能小，如果选定太大会影响程序的并行处理性能。

9.2.3 互斥量实例

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

实例：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NLOOP 5000

int counter;                /* incremented by threads */
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;

void *doit(void *);

int main(int argc, char **argv)
{
    pthread_t tidA, tidB;

    pthread_create(&tidA, NULL, doit, NULL);
    pthread_create(&tidB, NULL, doit, NULL);

        /* wait for both threads to terminate */
    pthread_join(tidA, NULL);
    pthread_join(tidB, NULL);

    return 0;
}

void *doit(void *vptr)
{
    int     i, val;

    for (i = 0; i < NLOOP; i++) {
        pthread_mutex_lock(&counter_mutex);
```

```
        val = counter;
        printf("%x: %d\n", (unsigned int)pthread_self(), val + 1);
        counter = val + 1;

        pthread_mutex_unlock(&counter_mutex);
    }

    return NULL;
}
```

这样运行结果就正常了，每次运行都能数到10000。

9.3 死锁

1.同一个线程在拥有A锁的情况下再次请求获得A锁
2.线程一拥有A锁，请求获得B锁；线程二拥有B锁，请求获得A锁
死锁导致的结果是什么？

9.4 读写锁

读共享，写独占

```
pthread_rwlock_t
pthread_rwlock_init
pthread_rwlock_destroy
pthread_rwlock_rdlock
pthread_rwlock_wrlock
pthread_rwlock_tryrdlock
pthread_rwlock_trywrlock
pthread_rwlock_unlock
```

实例：

```
#include <stdio.h>
#include <pthread.h>
int counter;
pthread_rwlock_t rwlock;
//3个线程不定时写同一全局资源，5个线程不定时读同一全局资源
void *th_write(void *arg)
{
    int t;
    while (1) {
        pthread_rwlock_wrlock(&rwlock);
        t = counter;
        usleep(100);
        printf("write %x  : counter=%d   ++counter=%d\n", (int)pthread_self(), t, ++counter);
        pthread_rwlock_unlock(&rwlock);
        usleep(100);
```

```
    }
}
void *th_read(void *arg)
{
    while (1) {
        pthread_rwlock_rdlock(&rwlock);
        printf("read %x  : %d\n", (int)pthread_self(), counter);
        pthread_rwlock_unlock(&rwlock);
        usleep(100);
    }
}
int main(void)
{
    int i;
    pthread_t tid[8];
    pthread_rwlock_init(&rwlock, NULL);
    for (i = 0; i < 3; i++)
        pthread_create(&tid[i], NULL, th_write, NULL);
    for (i = 0; i < 5; i++)
        pthread_create(&tid[i+3], NULL, th_read, NULL);
    pthread_rwlock_destroy(&rwlock);
    for (i = 0; i < 8; i++)
        pthread_join(tid[i], NULL);
    return 0;
}
```

9.5 条件变量

条件变量给多个线程提供了一个汇合的场所,条件变量控制原语:

```
        pthread_cond_t
        pthread_cond_init
        pthread_cond_destroy
        pthread_cond_wait
        pthread_cond_timedwait
        pthread_cond_signal
        pthread_cond_broadcast
```

生产者消费者模型:

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

struct msg {
    struct msg *next;
    int num;
};
```

```
struct msg *head;
pthread_cond_t has_product = PTHREAD_COND_INITIALIZER;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *consumer(void *p)
{
    struct msg *mp;

    for (;;) {
        pthread_mutex_lock(&lock);
        while (head == NULL)
            pthread_cond_wait(&has_product, &lock);
        mp = head;
        head = mp->next;
        pthread_mutex_unlock(&lock);
        printf("Consume %d\n", mp->num);
        free(mp);
        sleep(rand() % 5);
    }
}

void *producer(void *p)
{
    struct msg *mp;
    for (;;) {
        mp = malloc(sizeof(struct msg));
        mp->num = rand() % 1000 + 1;
        printf("Produce %d\n", mp->num);
        pthread_mutex_lock(&lock);
        mp->next = head;
        head = mp;
        pthread_mutex_unlock(&lock);
        pthread_cond_signal(&has_product);
        sleep(rand() % 5);
    }
}

int main(int argc, char *argv[])
{
    pthread_t pid, cid;

    srand(time(NULL));
    pthread_create(&pid, NULL, producer, NULL);
    pthread_create(&cid, NULL, consumer, NULL);
    pthread_join(pid, NULL);
    pthread_join(cid, NULL);
    return 0;
}
```

9.6 信号量

信号量控制原语

```
sem_t
sem_init
sem_wait
sem_trywait
sem_timedwait
sem_post
sem_destroy
```

生产者消费者实例：

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

#define NUM 5
int queue[NUM];
sem_t blank_number, product_number;

void *producer(void *arg)
{
    int p = 0;
    while (1) {
        sem_wait(&blank_number);
        queue[p] = rand() % 1000 + 1;
        printf("Produce %d\n", queue[p]);
        sem_post(&product_number);
        p = (p+1)%NUM;
        sleep(rand()%5);
    }
}

void *consumer(void *arg)
{
    int c = 0;
    while (1) {
        sem_wait(&product_number);
        printf("Consume %d\n", queue[c]);
        queue[c] = 0;
        sem_post(&blank_number);
        c = (c+1)%NUM;
        sleep(rand()%5);
    }
}

int main(int argc, char *argv[])
{
    pthread_t pid, cid;

    sem_init(&blank_number, 0, NUM);
    sem_init(&product_number, 0, 0);
    pthread_create(&pid, NULL, producer, NULL);
    pthread_create(&cid, NULL, consumer, NULL);
```

```
    pthread_join(pid, NULL);
    pthread_join(cid, NULL);
    sem_destroy(&blank_number);
    sem_destroy(&product_number);
    return 0;
}
```

9.7 进程间锁

9.7.1 进程间pthread_mutex

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
pshared:
    线程锁：PTHREAD_PROCESS_PRIVATE
    进程锁：PTHREAD_PROCESS_SHARED
    默认情况是线程锁
```

实例：

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <string.h>

struct mt {
    int num;
    pthread_mutex_t mutex;
    pthread_mutexattr_t mutexattr;
};

int main(void)
{
    int fd, i;
    struct mt *mm;
    pid_t pid;
    fd = open("mt_test", O_CREAT | O_RDWR, 0777);
    /* 不需要write,文件里初始值为0 */
    ftruncate(fd, sizeof(*mm));
    mm = mmap(NULL, sizeof(*mm), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);
```

```
    memset(mm, 0, sizeof(*mm));

    /* 初始化互斥对象属性 */
    pthread_mutexattr_init(&mm->mutexattr);

    /* 设置互斥对象为PTHREAD_PROCESS_SHARED共享，即可以在多个进程的线程访问,PTHREAD_PROCESS_PRIVATE
为同一进程的线程共享 */
    pthread_mutexattr_setpshared(&mm->mutexattr,PTHREAD_PROCESS_SHARED);

    pthread_mutex_init(&mm->mutex, &mm->mutexattr);

    pid = fork();
    if (pid == 0){
            /* 加10次。相当于加10 */
        for (i=0;i<10;i++){
            pthread_mutex_lock(&mm->mutex);
            (mm->num)++;
            printf("num++:%d\n",mm->num);
            pthread_mutex_unlock(&mm->mutex);
            sleep(1);
        }
    }
    else if (pid > 0) {
        /* 父进程完成x+2,加10次，相当于加20 */
        for (i=0; i<10; i++){
            pthread_mutex_lock(&mm->mutex);
            mm->num += 2;
            printf("num+=2:%d\n",mm->num);
            pthread_mutex_unlock(&mm->mutex);
            sleep(1);
        }
        wait(NULL);
    }
    pthread_mutex_destroy(&mm->mutex);
    pthread_mutexattr_destroy(&mm->mutexattr);
    /* 父子均需要释放 */
    munmap(mm,sizeof(*mm));
    unlink("mt_test");
    return 0;
}
```

9.7.2 文件锁

使用fcntl提供文件锁

```
struct flock {
              ...
              short l_type;    /* Type of lock: F_RDLCK,
                                  F_WRLCK, F_UNLCK */
              short l_whence;  /* How to interpret l_start:
                                  SEEK_SET, SEEK_CUR, SEEK_END */
              off_t l_start;   /* Starting offset for lock */
```

```
            off_t l_len;     /* Number of bytes to lock */
            pid_t l_pid;     /* PID of process blocking our lock
                                (F_GETLK only) */
            ...
        };
```

实例：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
void sys_err(char *str)
{
    perror(str);
    exit(1);
}
int main(int argc, char *argv[])
{
    int fd;
    struct flock f_lock;
    if (argc < 2) {
        printf("./a.out filename\n");
        exit(1);
    }

    if ((fd = open(argv[1], O_RDWR)) < 0)
        sys_err("open");

    //f_lock.l_type = F_WRLCK;
    f_lock.l_type = F_RDLCK;
    f_lock.l_whence = SEEK_SET;
    f_lock.l_start = 0;
    f_lock.l_len = 0;   //0表示整个文件加锁

    fcntl(fd, F_SETLKW, &f_lock);
    printf("get flock\n");
    sleep(10);
    f_lock.l_type = F_UNLCK;
    fcntl(fd, F_SETLKW, &f_lock);
    printf("un flock\n");

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

9.8　习题

1.请同学们自己编写出死锁程序。

2.哲学家就餐,5个哲学家，但是只有5支筷子,每个哲学家双手各拿起一支筷子时，可以进餐n秒(rand()%5)。分别用互斥量，信号量，条件变量实现实现。

第 10 章

网络基础

10.1 模型

10.1.1 OSI七层模型

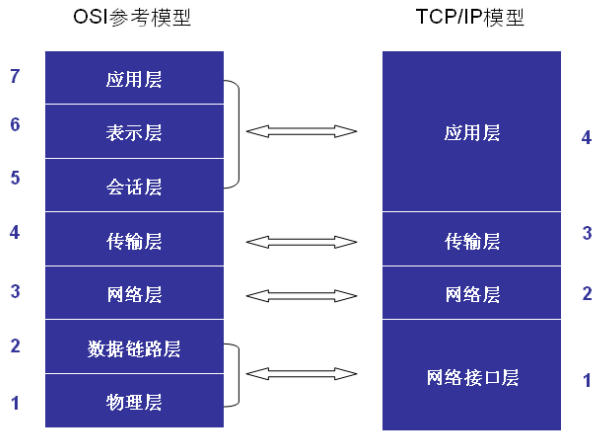

图 10.1: OSI模型

1.物理层：主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流(就是由1、0转化为电流强弱来进行传输，到达目的地后再转化为1、0，也就是我们常说的数模转换与模数转换)。这一层的数据叫做比特。

2.数据链路层：定义了如何让格式化数据以进行传输，以及如何让控制对物理介质的访问。这一层通常还提供错误检测和纠正，以确保数据的可靠传输。

3.网络层：在位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。Internet的发展使得从世界各站点访问信息的用户数大大增加，而网络层正是管理这种连接的层。

4.传输层：定义了一些传输数据的协议和端口号(WWW端口80等)，如：TCP(传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据)，UDP(用户数据报协议，与TCP特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如QQ聊天数据就是通过这种方式传输的)。 主要是将从下层接收的数据进行分段和传输，到达目的地址后再进行重组。常常把这一层数据叫做段。

5.会话层：通过传输层(端口号：传输端口与接收端口)建立数据传输的通路。主要在你

的系统之间发起会话或者接受会话请求(设备之间需要互相认识可以是IP也可以是MAC或者是主机名)。

6.表示层：可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。例如，PC程序与另一台计算机进行通信，其中一台计算机使用扩展二一十进制交换码(EBCDIC)，而另一台则使用美国信息交换标准码（ASCII）来表示相同的字符。如有必要，表示层会通过使用一种通格式来实现多种数据格式之间的转换。

7.应用层：是最靠近用户的OSI层。这一层为用户的应用程序(例如电子邮件、文件传输和终端仿真)提供网络服务。

10.1.2　TCP/IP四层模型

一般开发程序员讨论最多的是TCP/IP模型

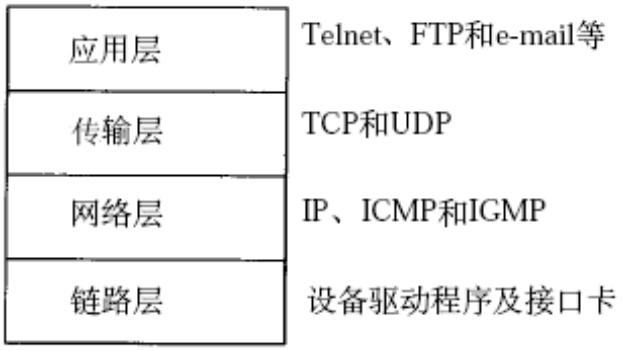

图 10.2: TCP/IP模型

10.2　通信过程

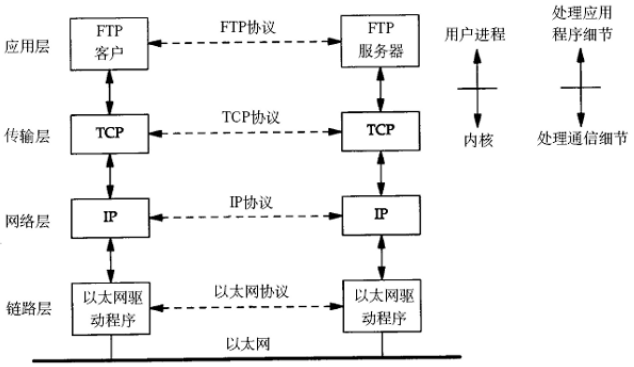

图 10.3: TCP/IP通信过程

上图对应两台计算机在同一网段中的情况，如果两台计算机在不同的网段中，那么数据从一台计算机到另一台计算机传输过程中要经过一个或多个路由器，如下图所示

其实在链路层之下还有物理层，指的是电信号的传递方式，比如现在以太网通用的网线（双绞线）、早期以太网采用的的同轴电缆（现在主要用于有线电视）、光纤等都属于物理层的概念。物理层的能力决定了最大传输速率、传输距离、抗干扰性等。集线器（Hub）是工作在物理层的网络设备，用于双绞线的连接和信号中继（将已衰减的信号再次放大使之传得更远）。

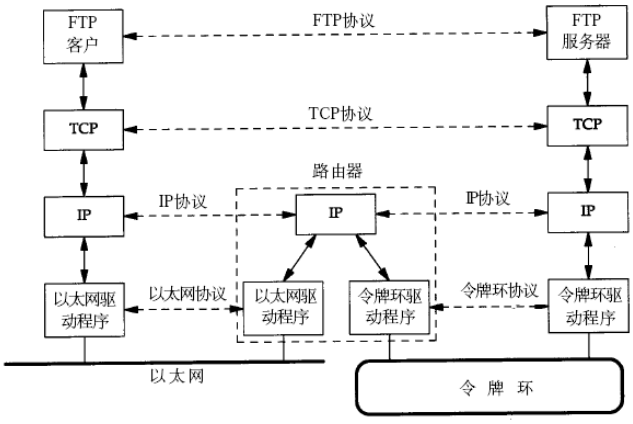

图 10.4: 跨路由通信过程

链路层有以太网、令牌环网等标准，链路层负责网卡设备的驱动、帧同步（就是说从网线上检测到什么信号算作新帧的开始）、冲突检测（如果检测到冲突就自动重发）、数据差错校验等工作。交换机是工作在链路层的网络设备，可以在不同的链路层网络之间转发数据帧（比如十兆以太网和百兆以太网之间、以太网和令牌环网之间），由于不同链路层的帧格式不同，交换机要将进来的数据包拆掉链路层首部重新封装之后再转发。

网络层的IP协议是构成Internet的基础。Internet上的主机通过IP地址来标识，Internet上有大量路由器负责根据IP地址选择合适的路径转发数据包，数据包从Internet上的源主机到目的主机往往要经过十多个路由器。路由器是工作在第三层的网络设备，同时兼有交换机的功能，可以在不同的链路层接口之间转发数据包，因此路由器需要将进来的数据包拆掉网络层和链路层两层首部并重新封装。IP协议不保证传输的可靠性，数据包在传输过程中可能丢失，可靠性可以在上层协议或应用程序中提供支持。

网络层负责点到点（point-to-point）的传输（这里的“点”指主机或路由器），而传输层负责端到端（end-to-end）的传输（这里的“端”指源主机和目的主机）。传输层可选择TCP或UDP协议。TCP是一种面向连接的、可靠的协议，有点像打电话，双方拿起电话互通身份之后就建立了连接，然后说话就行了，这边说的话那边保证听得到，并且是按说话的顺序听到的，说完话挂机断开连接。也就是说TCP传输的双方需要首先建立连接，之后由TCP协议保证数据收发的可靠性，丢失的数据包自动重发，上层应用程序收到的总是可靠的数据流，通讯之后关闭连接。UDP协议不面向连接，也不保证可靠性，有点像寄信，写好信放到邮筒里，既不能保证信件在邮递过程中不会丢失，也不能保证信件是按顺序寄到目的地的。使用UDP协议的应用程序需要自己完成丢包重发、消息排序等工作。

目的主机收到数据包后，如何经过各层协议栈最后到达应用程序呢？整个过程如下图所示

以太网驱动程序首先根据以太网首部中的“上层协议”字段确定该数据帧的有效载荷（payload，指除去协议首部之外实际传输的数据）是IP、ARP还是RARP协议的数据报，然后交给相应的协议处理。假如是IP数据报，IP协议再根据IP首部中的“上层协议”字段确定该数据报的有效载荷是TCP、UDP、ICMP还是IGMP，然后交给相应的协议处理。假如是TCP段或UDP段，TCP或UDP协议再根据TCP首部或UDP首部的“端口号”字段确定应该将应用层数据交给哪个用户进程。IP地址是标识网络中不同主机的地址，而端口号就是同一台主机上标识不同进程的地址，IP地址和端口号合起来标识网络中唯一的进程。

注意，虽然IP、ARP和RARP数据报都需要以太网驱动程序来封装成帧，但是从功能上划分，ARP和RARP属于链路层，IP属于网络层。虽然ICMP、IGMP、TCP、UDP的数据都需要IP协议来封装成数据报，但是从功能上划分，ICMP、IGMP与IP同属于网络层，TCP和UDP属于传输

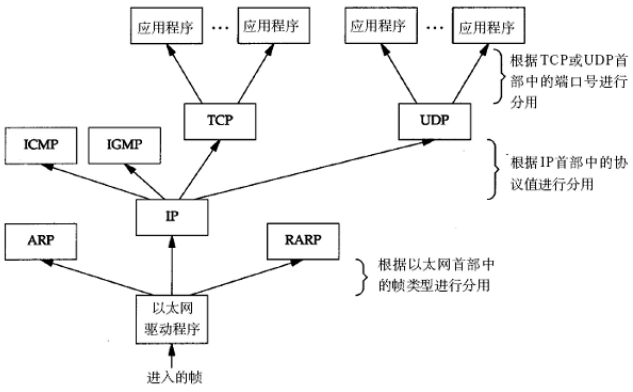

图 10.5: Multiplexing过程

层。本文对RARP、ICMP、IGMP协议不做进一步介绍，有兴趣的读者可以看参考资料。

10.3 协议格式

10.3.1 数据包封装

传输层及其以下的机制由内核提供，应用层由用户进程提供（后面将介绍如何使用socket API编写应用程序），应用程序对通讯数据的含义进行解释，而传输层及其以下处理通讯的细节，将数据从一台计算机通过一定的路径发送到另一台计算机。应用层数据通过协议栈发到网络上时，每层协议都要加上一个数据首部（header），称为封装（Encapsulation），如下图所示

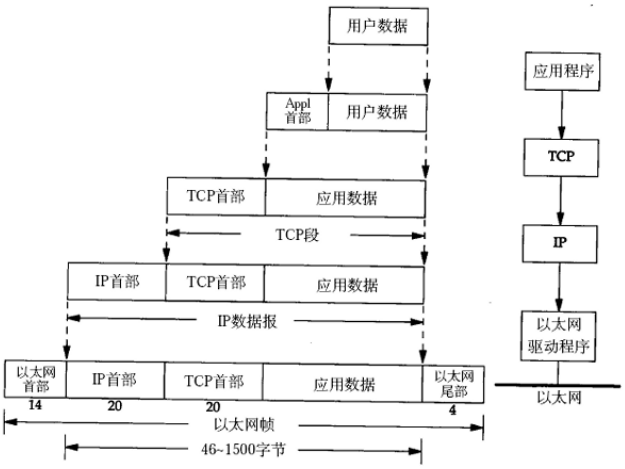

图 10.6: TCP/TP数据包封装

不同的协议层对数据包有不同的称谓，在传输层叫做段（segment），在网络层叫做数据报（datagram），在链路层叫做帧（frame）。数据封装成帧后发到传输介质上，到达目的主机后每层协议再剥掉相应的首部，最后将应用层数据交给应用程序处理。

10.3.2 以太网帧格式

其中的源地址和目的地址是指网卡的硬件地址（也叫MAC地址），长度是48位，是在网卡出厂时固化的。用ifconfig命令看一下，“HWaddr 00:15:F2:14:9E:3F”部分就是硬件地址。协议字段有三种值，分别对应IP、ARP、RARP。帧末尾是CRC校验码。

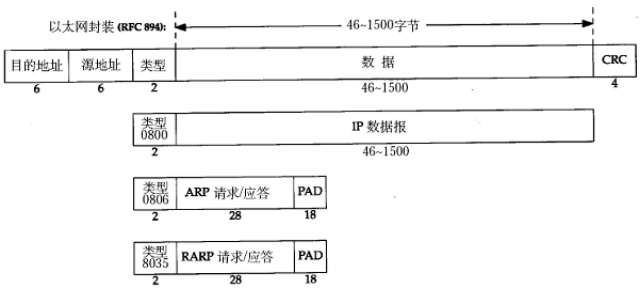

图 10.7: 以太网帧格式

以太网帧中的数据长度规定最小46字节，最大1500字节，ARP和RARP数据包的长度不够46字节，要在后面补填充位。最大值1500称为以太网的最大传输单元（MTU），不同的网络类型有不同的MTU，如果一个数据包从以太网路由到拨号链路上，数据包长度大于拨号链路的MTU了，则需要对数据包进行分片（fragmentation）。ifconfig命令的输出中也有“MTU:1500”。注意，MTU这个概念指数据帧中有效载荷的最大长度，不包括帧首部的长度。

10.3.3 ARP数据报格式

在网络通讯时，源主机的应用程序知道目的主机的IP地址和端口号，却不知道目的主机的硬件地址，而数据包首先是被网卡接收到再去处理上层协议的，如果接收到的数据包的硬件地址与本机不符，则直接丢弃。因此在通讯前必须获得目的主机的硬件地址。ARP协议就起到这个作用。源主机发出ARP请求，询问“IP地址是192.168.0.1的主机的硬件地址是多少”，并将这个请求广播到本地网段（以太网帧首部的硬件地址填FF:FF:FF:FF:FF:FF表示广播），目的主机接收到广播的ARP请求，发现其中的IP地址与本机相符，则发送一个ARP应答数据包给源主机，将自己的硬件地址填写在应答包中。

每台主机都维护一个ARP缓存表，可以用arp -a命令查看。缓存表中的表项有过期时间（一般为20分钟），如果20分钟内没有再次使用某个表项，则该表项失效，下次还要发ARP请求来获得目的主机的硬件地址。想一想，为什么表项要有过期时间而不是一直有效？

ARP数据报的格式如下所示

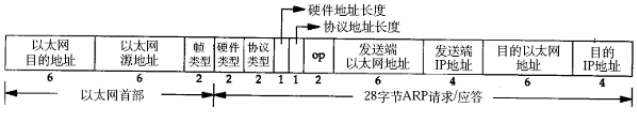

图 10.8: ARP数据报格式

注意到源MAC地址、目的MAC地址在以太网首部和ARP请求中各出现一次，对于链路层为以太网的情况是多余的，但如果链路层是其它类型的网络则有可能是必要的。硬件类型指链路层网络类型，1为以太网，协议类型指要转换的地址类型，0x0800为IP地址，后面两个地址长度对于以太网地址和IP地址分别为6和4（字节），op字段为1表示ARP请求，op字段为2表示ARP应答。

下面举一个具体的例子。

请求帧如下（为了清晰在每行的前面加了字节计数，每行16个字节）：

以太网首部（14字节）

```
0000:  ff ff ff ff ff ff 00 05 5d 61 58 a8 08 06
```

ARP帧（28字节）

```
0000:  00 01
0010:  08 00 06 04 00 01 00 05 5d 61 58 a8 c0 a8 00 37
```

0020: 00 00 00 00 00 00 c0 a8 00 02

填充位（18字节）

0020: 00 77 31 d2 50 10

0030: fd 78 41 d3 00 00 00 00 00 00 00 00

以太网首部：目的主机采用广播地址，源主机的MAC地址是00:05:5d:61:58:a8，上层协议类型0x0806表示ARP。

ARP帧：硬件类型0x0001表示以太网，协议类型0x0800表示IP协议，硬件地址（MAC地址）长度为6，协议地址（IP地址）长度为4，op为0x0001表示请求目的主机的MAC地址，源主机MAC地址为00:05:5d:61:58:a8，源主机IP地址为c0 a8 00 37（192.168.0.55），目的主机MAC地址全0待填写，目的主机IP地址为c0 a8 00 02（192.168.0.2）。

由于以太网规定最小数据长度为46字节，ARP帧长度只有28字节，因此有18字节填充位，填充位的内容没有定义，与具体实现相关。

应答帧如下：

以太网首部

0000: 00 05 5d 61 58 a8 00 05 5d a1 b8 40 08 06

ARP帧

0000: 00 01

0010: 08 00 06 04 00 02 00 05 5d a1 b8 40 c0 a8 00 02

0020: 00 05 5d 61 58 a8 c0 a8 00 37

填充位

0020: 00 77 31 d2 50 10

0030: fd 78 41 d3 00 00 00 00 00 00 00 00

以太网首部：目的主机的MAC地址是00:05:5d:61:58:a8，源主机的MAC地址是00:05:5d:a1:b8:40，上层协议类型0x0806表示ARP。

ARP帧：硬件类型0x0001表示以太网，协议类型0x0800表示IP协议，硬件地址（MAC地址）长度为6，协议地址（IP地址）长度为4，op为0x0002表示应答，源主机MAC地址为00:05:5d:a1:b8:40，源主机IP地址为c0 a8 00 02（192.168.0.2），目的主机MAC地址为00:05:5d:61:58:a8，目的主机IP地址为c0 a8 00 37（192.168.0.55）。

思考题：如果源主机和目的主机不在同一网段，ARP请求的广播帧无法穿过路由器，源主机如何与目的主机通信？

10.3.4 IP段格式

图 10.9: IP数据报格式

IP数据报的首部长度和数据长度都是可变长的，但总是4字节的整数倍。对于IPv4，4位版本字段是4。4位首部长度的数值是以4字节为单位的，最小值为5，也就是说首部长度最小是4x5=20字节，也就是不带任何选项的IP首部，4位能表示的最大值是15，也就是说首部长度最大是60字节。8位TOS字段有3个位用来指定IP数据报的优先级（目前已经废弃不用），还有4个位表示可选的服务类型（最小延迟、最大□吐量、最大可靠性、最小成本），还有一个位总是0。总长度是整个数据报（包括IP首部和IP层payload）的字节数。每传一个IP数据报，16位的标识加1，可用于分片和重新组装数据报。3位标志和13位片偏移用于分片。TTL（Time to live)是这样用的：源主机为数据包设定一个生存时间，比如64，每过一个路由器就把该值减1，如果减到0就表示路由已经太长了仍然找不到目的主机的网络，就丢弃该包，因此这个生存时间的单位不是秒，而是跳（hop）。协议字段指示上层协议是TCP、UDP、ICMP还是IGMP。然后是校验和，只校验IP首部，数据的校验由更高层协议负责。IPv4的IP地址长度为32位。选项字段的解释从略。

想一想，前面讲了以太网帧中的最小数据长度为46字节，不足46字节的要用填充字节补上，那么如何界定这46字节里前多少个字节是IP、ARP或RARP数据报而后面是填充字节？

10.3.5 UDP数据抱格式

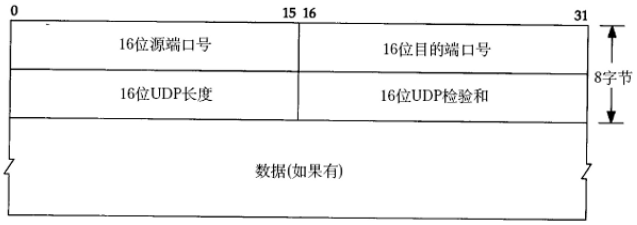

图 10.10: UDP数据段

下面分析一帧基于UDP的TFTP协议帧。

以太网首部

0000: 00 05 5d 67 d0 b1 00 05 5d 61 58 a8 08 00

IP首部

0000: 45 00

0010: 00 53 93 25 00 00 80 11 25 ec c0 a8 00 37 c0 a8

0020: 00 01

UDP首部

0020： 05 d4 00 45 00 3f ac 40

TFTP协议

0020: 00 01 ‘c” : ‘” ’ q’

0030: ‘w’ ‘e’ ‘r’ ‘q” . ‘’ q’ ‘w’ ‘e’ 00 ’ n’ ‘e” t’ ‘a” s’ ‘c’ ‘i’

0040: ‘i’ 00 ’ b’ ‘1’ ‘k” s’ ‘i’ ‘z’ ‘e’ 00 ’ 5’ ‘1’ ‘2’ 00 ’ t’ ‘i’

0050: ’ m’ ‘e’ ‘o’ ‘u” t’ 00 ‘1’ ‘0’ 00 ’ t” s’ ‘i’ ‘z’ ‘e’ 00 ’ 0’

0060: 00

以太网首部：源MAC地址是00:05:5d:61:58:a8，目的MAC地址是00:05:5d:67:d0:b1，上层协议类型0x0800表示IP。

IP首部：每一个字节0x45包含4位版本号和4位首部长度，版本号为4，即IPv4，首部长度为5，说明IP首部不带有选项字段。服务类型为0，没有使用服务。16位总长度字段（包括IP首部和IP层payload的长度）为0x0053，即83字节，加上以太网首部14字节可知整个帧

长度是97字节。IP报标识是0x9325，标志字段和片偏移字段设置为0x0000，就是DF=0允许分片，MF=0此数据报没有更多分片，没有分片偏移。TTL是0x80，也就是128。上层协议0x11表示UDP协议。IP首部校验和为0x25ec，源主机IP是c0 a8 00 37（192.168.0.55），目的主机IP是c0 a8 00 01（192.168.0.1）。

UDP首部：源端口号0x05d4（1492）是客户端的端口号，目的端口号0x0045（69）是TFTP服务的well-known端口号。UDP报长度为0x003f，即63字节，包括UDP首部和UDP层payload的长度。UDP首部和UDP层payload的校验和为0xac40。

TFTP是基于文本的协议，各字段之间用字节0分隔，开头的00 01表示请求读取一个文件，接下来的各字段是：

```
c:\qwerq.qwe
netascii
blksize 512
timeout 10
tsize 0
```

一般的网络通信都是像TFTP协议这样，通信的双方分别是客户端和服务器，客户端主动发起请求（上面的例子就是客户端发起的请求帧），而服务器被动地等待、接收和应答请求。客户端的IP地址和端口号唯一标识了该主机上的TFTP客户端进程，服务器的IP地址和端口号唯一标识了该主机上的TFTP服务进程，由于客户端是主动发起请求的一方，它必须知道服务器的IP地址和TFTP服务进程的端口号，所以，一些常见的网络协议有默认的服务器端口，例如HTTP服务默认TCP协议的80端口，FTP服务默认TCP协议的21端口，TFTP服务默认UDP协议的69端口（如上例所示）。在使用客户端程序时，必须指定服务器的主机名或IP地址，如果不明确指定端口号则采用默认端口，请读者查阅ftp、tftp等程序的man page了解如何指定端口号。/etc/services中列出了所有well-known的服务端口和对应的传输层协议，这是由IANA（Internet Assigned Numbers Authority）规定的，其中有些服务既可以用TCP也可以用UDP，为了清晰，IANA规定这样的服务采用相同的TCP或UDP默认端口号，而另外一些TCP和UDP的相同端口号却对应不同的服务。

很多服务有well-known的端口号，然而客户端程序的端口号却不必是well-known的，往往是每次运行客户端程序时由系统自动分配一个空闲的端口号，用完就释放掉，称为ephemeral的端口号，想想这是为什么。

前面提过，UDP协议不面向连接，也不保证传输的可靠性，例如：

发送端的UDP协议层只管把应用层传来的数据封装成段交给IP协议层就算完成任务了，如果因为网络故障该段无法发到对方，UDP协议层也不会给应用层返回任何错误信息。

接收端的UDP协议层只管把收到的数据根据端口号交给相应的应用程序就算完成任务了，如果发送端发来多个数据包并且在网络上经过不同的路由，到达接收端时顺序已经错乱了，UDP协议层也不保证按发送时的顺序交给应用层。

通常接收端的UDP协议层将收到的数据放在一个固定大小的缓冲区中等待应用程序来提取和处理，如果应用程序提取和处理的速度很慢，而发送端发送的速度很快，就会丢失数据包，UDP协议层并不报告这种错误。

因此，使用UDP协议的应用程序必须考虑到这些可能的问题并实现适当的解决方案，例如等待应答、超时重发、为数据包编号、流量控制等。一般使用UDP协议的应用程序实现都比较简单，只是发送一些对可靠性要求不高的消息，而不发送大量的数据。例如，基于UDP的TFTP协议一般只用于传送小文件（所以才叫trivial的ftp），而基于TCP的FTP协议适用于各种文件的传输。下面看TCP协议如何用面向连接的服务来代替应用程序解决传输的可靠性

问题。

10.3.6 TCP数据报格式

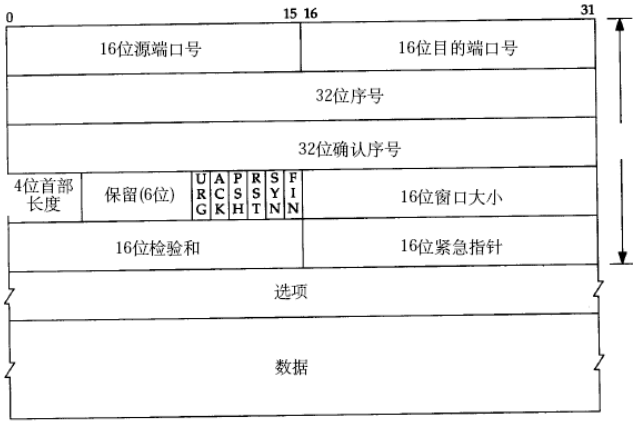

图 10.11: TCP数据段

和UDP协议一样也有源端口号和目的端口号，通讯的双方由IP地址和端口号标识。32位序号、32位确认序号、窗口大小稍后详细解释。4位首部长度和IP协议头类似，表示TCP协议头的长度，以4字节为单位，因此TCP协议头最长可以是4x15=60字节，如果没有选项字段，TCP协议头最短20字节。URG、ACK、PSH、RST、SYN、FIN是六个控制位，本节稍后将解释SYN、ACK、FIN、RST四个位，其它位的解释从略。16位检验和将TCP协议头和数据都计算在内。紧急指针和各种选项的解释从略。

通信时序 下图是一次TCP通讯的时序图。

在这个例子中，首先客户端主动发起连接、发送请求，然后服务器端响应请求，然后客户端主动关闭连接。两条竖线表示通讯的两端，从上到下表示时间的先后顺序，注意，数据从一端传到网络的另一端也需要时间，所以图中的箭头都是斜的。双方发送的段按时间顺序编号为1-10，各段中的主要信息在箭头上标出，例如段2的箭头上标着SYN，8000(0)，ACK 1001，，表示该段中的SYN位置1，32位序号是8000，该段不携带有效载荷（数据字节数为0），ACK位置1，32位确认序号是1001，带有一个mss选项值为1024。

建立连接的过程：

1.客户端发出段1，SYN位表示连接请求。序号是1000，这个序号在网络通讯中用作临时的地址，每发一个数据字节，这个序号要加1，这样在接收端可以根据序号排出数据包的正确顺序，也可以发现丢包的情况，另外，规定SYN位和FIN位也要占一个序号，这次虽然没发数据，但是由于发了SYN位，因此下次再发送应该用序号1001。mss表示最大段尺寸，如果一个段太大，封装成帧后超过了链路层的最大帧长度，就必须在IP层分片，为了避免这种情况，客户端声明自己的最大段尺寸，建议服务器端发来的段不要超过这个长度。

2.服务器发出段2，也带有SYN位，同时置ACK位表示确认，确认序号是1001，表示“我接收到序号1000及其以前所有的段，请你下次发送序号为1001的段”，也就是应答了客户端的连接请求，同时也给客户端发出一个连接请求，同时声明最大尺寸为1024。

3.客户端发出段3，对服务器的连接请求进行应答，确认序号是8001。

在这个过程中，客户端和服务器分别给对方发了连接请求，也应答了对方的连接请求，其中服务器的请求和应答在一个段中发出，因此一共有三个段用于建立连接，称为’‘三方握手（three-way-handshake）”’。在建立连接的同时，双方协商了一些信息，例如双方发送序号的初始值、最大段尺寸等。

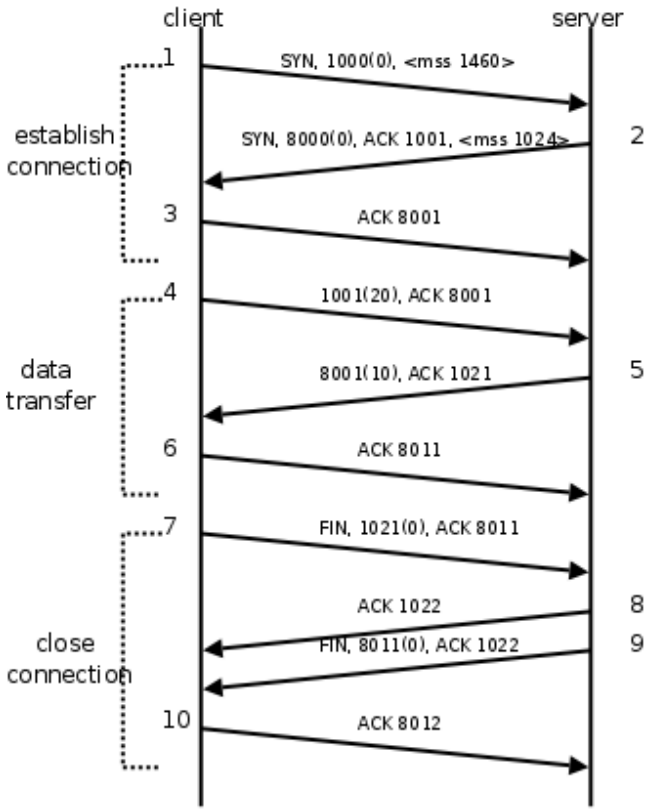

图 10.12: TCP连接建立断开

在TCP通讯中，如果一方收到另一方发来的段，读出其中的目的端口号，发现本机并没有任何进程使用这个端口，就会应答一个包含RST位的段给另一方。

数据传输的过程：

1.客户端发出段4，包含从序号1001开始的20个字节数据。

2.服务器发出段5，确认序号为1021，对序号为1001-1020的数据表示确认收到，同时请求发送序号1021开始的数据，服务器在应答的同时也向客户端发送从序号8001开始的10个字节数据，这称为piggyback。

3.客户端发出段6，对服务器发来的序号为8001-8010的数据表示确认收到，请求发送序号8011开始的数据。

关闭连接的过程：

1.客户端发出段7，FIN位表示关闭连接的请求。

2.服务器发出段8，应答客户端的关闭连接请求。

3.服务器发出段9，其中也包含FIN位，向客户端发送关闭连接请求。

4.客户端发出段10，应答服务器的关闭连接请求。

10.4 再议TCP

10.4.1 tcp状态转换图

这个图N多人都知道，它排除和定位网络或系统故障时大有帮助，但是怎样牢牢地将这张图刻在脑中呢？那么你就一定要对这张图的每一个状态，及转换的过程有深刻 的认识，不能只停留在一知半解之中。下面对这张图的11种状态详细解析一下，以便加强记忆！不过在这之前，先回顾一下TCP建立连接的三次握手过程，以及 关闭连接的四次握手过程。

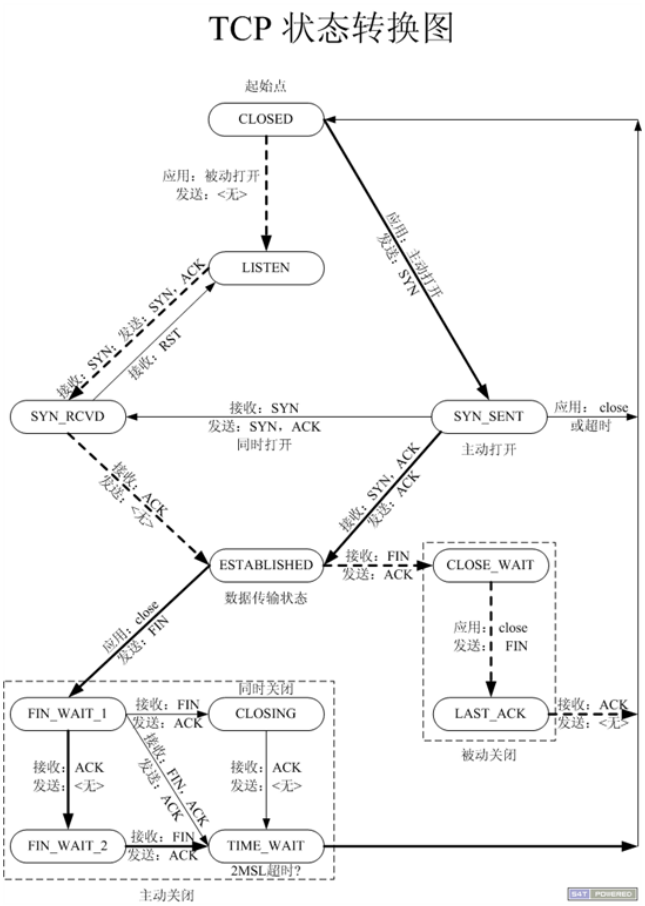

图 10.13: TCP状态转换图

1、建立连接协议（三次握手）

（1）客户端发送一个带SYN标志的TCP报文到服务器。这是三次握手过程中的报文1。

（2） 服务器端回应客户端的，这是三次握手中的第2个报文，这个报文同时带ACK标志和SYN标志。因此它表示对刚才客户端SYN报文的回应；同时又标志SYN给客户端，询问客户端是否准备好进行数据通讯。

（3） 客户必须再次回应服务段一个ACK报文，这是报文段3。

2、连接终止协议（四次握手）

由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

（1） TCP客户端发送一个FIN，用来关闭客户到服务器的数据传送（报文段4）。

（2） 服务器收到这个FIN，它发回一个ACK，确认序号为收到的序号加1（报文段5）。和SYN一样，一个FIN将占用一个序号。

（3） 服务器关闭客户端的连接，发送一个FIN给客户端（报文段6）。

（4） 客户段发回ACK报文确认，并将确认序号设置为收到序号加1（报文段7）。

CLOSED：这个没什么好说的了，表示初始状态。

LISTEN：这个也是非常容易理解的一个状态，表示服务器端的某个SOCKET处于监听状态，可以接受连接了。

SYN_RCVD：这个状态表示接受到了SYN报文，在正常情况下，这个状态是服务器端的SOCKET在建立TCP连接时的三次握手会话过程中的一个中间状态，很短暂，基本 上用netstat你是很难看到这种状态的，除非你特意写了一个客户端测试程序，故意将三次TCP握手过程中最后一个ACK报文不予发送。因此这种状态 时，当收到客户端的ACK报文后，它会进入到ESTABLISHED状态。

SYN_SENT：这个状态与SYN_RCVD遥想呼应，当客户端SOCKET执行CONNECT连接时，它首先发送SYN报文，因此也随即它会进入到了SYN_SENT状 态，并等待服务端的发送三次握手中的第2个报文。SYN_SENT状态表示客户端已发送SYN报文。

ESTABLISHED：这个容易理解了，表示连接已经建立了。

FIN_WAIT_1：这个状态要好好解释一下，其实FIN_WAIT_1和FIN_WAIT_2状态的真正含义都是表示等待对方的FIN报文。而这两种状态的区别 是：FIN_WAIT_1状态实际上是当SOCKET在ESTABLISHED状态时，它想主动关闭连接，向对方发送了FIN报文，此时该SOCKET即 进入到FIN_WAIT_1状态。而当对方回应ACK报文后，则进入到FIN_WAIT_2状态，当然在实际的正常情况下，无论对方何种情况下，都应该马 上回应ACK报文，所以FIN_WAIT_1状态一般是比较难见到的，而FIN_WAIT_2状态还有时常常可以用netstat看到。

FIN_WAIT_2：上面已经详细解释了这种状态，实际上FIN_WAIT_2状态下的SOCKET，表示半连接，也即有一方要求close连接，但另外还告诉对方，我暂时还有点数据需要传送给你，稍后再关闭连接。

TIME_WAIT：表示收到了对方的FIN报文，并发送出了ACK报文，就等2MSL后即可回到CLOSED可用状态了。如果FIN_WAIT_1状态下，收到了对方同时带 FIN标志和ACK标志的报文时，可以直接进入到TIME_WAIT状态，而无须经过FIN_WAIT_2状态。

CLOSING：这种状态比较特殊，实际情况中应该是很少见，属于一种比较罕见的例外状态。正常情况下，当你发送FIN报文后，按理来说是应该先收到（或同时收到）对方的 ACK报文，再收到对方的FIN报文。但是CLOSING状态表示你发送FIN报文后，并没有收到对方的ACK报文，反而却也收到了对方的FIN报文。什 么情况下会出现此种情况呢？其实细想一下，也不难得出结论：那就是如果双方几乎在同时close一个SOCKET的话，那么就出现了双方同时发送FIN报 文的情况，也即会出现CLOSING状态，表示双方都正在关闭SOCKET连接。

CLOSE_WAIT：这种状态的含义其实是表示在等待关闭。怎么理解呢？当对方close一个SOCKET后发送FIN报文给自己，你系统毫无疑问地会回应一个ACK报文给对 方，此时则进入到CLOSE_WAIT状态。接下来呢，实际上你真正需要考虑的事情是察看你是否还有数据发送给对方，如果没有的话，那么你也就可以 close这个SOCKET，发送FIN报文给对方，也即关闭连接。所以你在CLOSE_WAIT状态下，需要完成的事情是等待你去关闭连接。

LAST_ACK：这个状态还是比较容易好理解的，它是被动关闭一方在发送FIN报文后，最后等待对方的ACK报文。当收到ACK报文后，也即可以进入到CLOSED可用状态了。

10.4.2 TCP流量控制(滑动窗口)

介绍UDP时我们描述了这样的问题：如果发送端发送的速度较快，接收端接收到数据后处理的速度较慢，而接收缓冲区的大小是固定的，就会丢失数据。TCP协议通过’滑动窗口（Sliding Window）’机制解决这一问题。看下图的通讯过程。

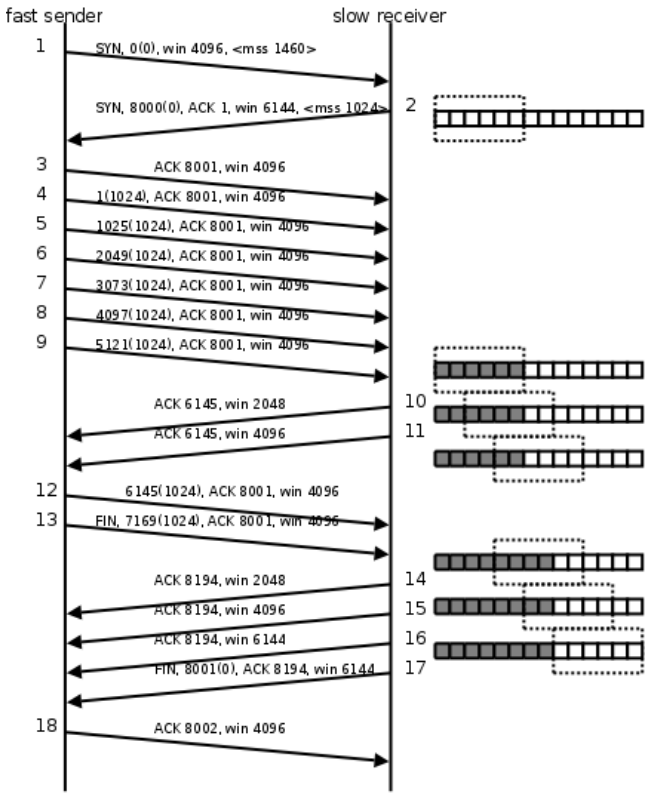

图 10.14: 滑动窗口

1.发送端发起连接，声明最大段尺寸是1460，初始序号是0，窗口大小是4K，表示“我的接收缓冲区还有4K字节空闲，你发的数据不要超过4K”。接收端应答连接请求，声明最大段尺寸是1024，初始序号是8000，窗口大小是6K。发送端应答，三方握手结束。

2.发送端发出段4-9，每个段带1K的数据，发送端根据窗口大小知道接收端的缓冲区满了，因此停止发送数据。

3.接收端的应用程序提走2K数据，接收缓冲区又有了2K空闲，接收端发出段10，在应答已收到6K数据的同时声明窗口大小为2K。

4.接收端的应用程序又提走2K数据，接收缓冲区有4K空闲，接收端发出段11，重新声明窗口大小为4K。

5.发送端发出段12-13，每个段带2K数据，段13同时还包含FIN位。

6.接收端应答接收到的2K数据（6145-8192），再加上FIN位占一个序号8193，因此应答序号是8194，连接处于半关闭状态，接收端同时声明窗口大小为2K。

7.接收端的应用程序提走2K数据，接收端重新声明窗口大小为4K。

8.接收端的应用程序提走剩下的2K数据，接收缓冲区全空，接收端重新声明窗口大小为6K。

9.接收端的应用程序在提走全部数据后，决定关闭连接，发出段17包含FIN位，发送端应答，连接完全关闭。

上图在接收端用小方块表示1K数据，实心的小方块表示已接收到的数据，虚线框表示接收缓冲区，因此套在虚线框中的空心小方块表示窗口大小，从图中可以看出，随着应用程序提走数据，虚线框是向右滑动的，因此称为滑动窗口。

从这个例子还可以看出，发送端是一K一K地发送数据，而接收端的应用程序可以两K两K地提走数据，当然也有可能一次提走3K或6K数据，或者一次只提走几个字节的数据，也就是说，应用程序所看到的数据是一个整体，或说是一个流（stream），在底层通讯中这些数据可能被拆成很多数据包来发送，但是一个数据包有多少字节对应用程序是不可见的，因此TCP协议是面向流的协议。而UDP是面向消息的协议，每个UDP段都是一条消息，应用程序必须以消息为单位提取数据，不能一次提取任意字节的数据，这一点和TCP是很不同的。

10.4.3　TCP半链接状态

当TCP链接中A发送FIN请求关闭，另一段B回应ACK后，B没有立即发送FIN给A时，A方处在半链接状态，此时A可以接收B发送的数据，但是A已不能再向B发送数据。

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

int shutdown(int sockfd, int how);

sockfd: 需要关闭的socket的描述符
how:允许为shutdown操作选择以下几种方式:
    SHUT_RD：关闭连接的读端。也就是该套接字不再接受数据，任何当前在套接字接受缓冲区的数据将被丢弃。
        进程将不能对该套接字发出任何读操作。对 TCP套接字该调用之后接受到的任何数据将被确认然后无声的丢弃掉。
    SHUT_WR:关闭连接的写端，进程不能在对此套接字发出写操作
    SHUT_RDWR:相当于调用shutdown两次：首先是以SHUT_RD,然后以SHUT_WR
```

使用close中止一 个连接，但它只是减少描述符的参考数，并不直接关闭连接，只有当描述符的参考数为0时才关闭连接。 shutdown可直接关闭描述符，不考虑描述 符的参考数，可选择中止一个方向的连接。

注意: 1>. 如果有多个进程共享一个套接字，close每被调用一次，计数减1，直到计数为0时，也就是所用进程都调用了close，套接字将被释放。 2>. 在多进程中如果一个进程中shutdown(sfd, SHUT_RDWR)后其它的进程将无法进行通信. 如果一个进程close(sfd)将不会影响到其它进程. 得自己理解引用计数的用法了

10.4.4　2MSL

2MSL TIME_WAIT状态存在的理由：

TIME_WAIT状态的存在有两个理由：（1）让4次握手关闭流程更加可靠；4次握手的最后一个ACK是是由主动关闭方发送出去的，若这个ACK丢失，被动关闭方会再次发一个FIN

过来。若主动关闭方能够保持一个2MSL的TIME_WAIT状态，则有更大的机会让丢失的ACK被再次发送出去。（2）防止lost duplicate对后续新建正常链接的传输造成破坏。lost duplicate在实际的网络中非常常见，经常是由于路由器产生故障，路径无法收敛，导致一个packet在路由器A，B，C之间做类似死循环的跳转。IP头部有个TTL，限制了一个包在网络中的最大跳数，因此这个包有两种命运，要么最后TTL变为0，在网络中消失；要么TTL在变为0之前路由器路径收敛，它凭借剩余的TTL跳数终于到达目的地。但非常可惜的是TCP通过超时重传机制在早些时候发送了一个跟它一模一样的包，并先于它达到了目的地，因此它的命运也就注定被TCP协议栈抛弃。另外一个概念叫做incarnation connection，指跟上次的socket pair一摸一样的新连接，叫做incarnation of previous connection。lost duplicate加上incarnation connection，则会对我们的传输造成致命的错误。大家都知道TCP是流式的，所有包到达的顺序是不一致的，依靠序列号由TCP协议栈做顺序的拼接；假设一个incarnation connection这时收到的seq=1000，来了一个lost duplicate为seq=1000，len=1000，则tcp认为这个lost duplicate合法，并存放入了receive buffer，导致传输出现错误。通过一个2MSL TIME_WAIT状态，确保所有的lost duplicate都会消失掉，避免对新连接造成错误。

该状态为什么设计在主动关闭这一方：

（1）发最后ack的是主动关闭一方

（2）只要有一方保持TIME_WAIT状态，就能起到避免incarnation connection在2MSL内的重新建立，不需要两方都有

如何正确对待2MSL TIME_WAIT?

RFC要求socket pair在处于TIME_WAIT时，不能再起一个incarnation connection。但绝大部分TCP实现，强加了更为严格的限制。在2MSL等待期间，socket中使用的本地端口在默认情况下不能再被使用。若A 10.234.5.5:1234和B 10.55.55.60:6666建立了连接，A主动关闭，那么在A端只要port为1234，无论对方的port和ip是什么，都不允许再起服务。显而易见这是比RFC更为严格的限制，RFC仅仅是要求socket pair不一致，而实现当中只要这个port处于TIME_WAIT，就不允许起连接。这个限制对主动打开方来说是无所谓的，因为一般用的是临时端口；但对于被动打开方，一般是server，就悲剧了，因为server一般是熟知端口。比如http，一般端口是80，不可能允许这个服务在2MSL内不能起来。解决方案是给服务器的socket设置SO_REUSEADDR选项，这样的话就算熟知端口处于TIME_WAIT状态，在这个端口上依旧可以将服务启动。当然，虽然有了SO_REUSEADDR选项，但sockt pair这个限制依旧存在。比如上面的例子，A通过SO_REUSEADDR选项依旧在1234端口上起了监听，但这时我们若是从B通过6666端口去连它，TCP协议会告诉我们连接失败，原因为Address already in use.

10.5 名词术语解析

10.5.1 什么是路由(route)

1.网络信息从信源到信宿的路径.路由是指路由器从一个接口上收到数据包，根据数据包的目的地址进行定向并转发到另一个接口的过程。

2.路由通常与桥接来对比，在粗心的人看来，它们似乎完成的是同样的事。它们的主要区别在于桥接发生在OSI参考模型的第二层（数据链路层），而路由发生在第三层（网络层）。这一区别使二者在传递信息的过程中使用不同的信息，从而以不同的方式来完成其任务。

3.确定最佳路径,通过网络传输信息

10.5.2　路由器工作原理

路由器（Router）是连接因特网中各局域网、广域网的设备，它会根据信道的情况自动选择和设定路由，以最佳路径，按前后顺序发送信号的设备。

传统地，路由器工作于OSI七层协议中的第三层，其主要任务是接收来自一个网络接口的数据包，根据其中所含的目的地址，决定转发到下一个目的地址。因此，路由器首先得在转发路由表中查找它的目的地址，若找到了目的地址，就在数据包的帧格前添加下一个MAC地址，同时IP数据包头的TTL（Time To Live）域也开始减数， 并重新计算校验和。当数据包被送到输出端口时，它需要按顺序等待，以便被传送到输出链路上。

路由器在工作时能够按照某种路由通信协议查找设备中的路由表。如果到某一特定节点有一条以上的路径，则基本预先确定的路由准则是选择最优（或最经济）的传输路径。由于各种网络段和其相互连接情况可能会因环境变化而变化，因此路由情况的信息一般也按所使用的路由信息协议的规定而定时更新。

网络中，每个路由器的基本功能都是按照一定的规则来动态地更新它所保持的路由表，以便保持路由信息的有效性。为了便于在网络间传送报文，路由器总是先按照预定的规则把较大的数据分解成适当大小的数据包，再将这些数据包分别通过相同或不同路径发送出去。当这些数据包按先后秩序到达目的地后，再把分解的数据包按照一定顺序包装成原有的报文形式。路由器的分层寻址功能是路由器的重要功能之一，该功能可以帮助具有很多节点站的网络来存储寻址信息，同时还能在网络间截获发送到远地网段的报文，起转发作用；选择最合理的路由，引导通信也是路由器基本功能；多协议路由器还可以连接使用不同通信协议的网络段，成为不同通信协议网络段之间的通信平台。

路由和交换之间的主要区别就是交换发生在OSI参考模型第二层（数据链路层），而路由发生在第三层，即网络层。这一区别决定了路由和交换在移动信息的过程 中需使用不同的控制信息，所以两者实现各自功能的方式是不同的。

10.5.3　路由表(Routing Table)

在计算机网络中，路由表或称路由择域信息库（RIB）是一个存储在路由器或者联网计算机中的电子表格（文件）或类数据库。路由表存储着指向特定网络地址的路径

10.5.4　以太网交换机工作原理

以太网交换机是基于以太网传输数据的交换机，以太网采用共享总线型传输媒体方式的局域网。以太网交换机的结构是每个端口都直接与主机相连，并且一般都工作在全双工方式。交换机能同时连通许多对端口，使每一对相互通信的主机都能像独占通信媒体那样，进行无冲突地传输数据。

以太网交换机工作于OSI网络参考模型的第二层（即数据链路层），是一种基于MAC（Media Access Control，介质访问控制）地址识别、完成以太网数据帧转发的网络设备。

10.5.5　hub工作原理

集线器实际上就是中继器的一种，其区别仅在于集线器能够提供更多的端口服务，所以集线器又叫多口中继器。

集线器功能是随机选出某一端口的设备，并让它独占全部带宽，与集线器的上联设备(交换机、路由器或服务器等)进行通信。从Hub的工作方式可以看出，它在网络中只起到信号放大和重发作用，其目的是扩大网络的传输范围，而不具备信号的定向传送能力，是一个

标准的共享式设备。其次是Hub只与它的上联设备(如上层Hub、交换机或服务器)进行通信，同层的各端口之间不会直接进行通信，而是通过上联设备再将信息广播到所有端口上。 由此可见，即使是在同一Hub的不同两个端口之间进行通信，都必须要经过两步操作：第一步是将信息上传到上联设备；第二步是上联设备再将该信息广播到所有端口上.

10.5.6 半双工/全双工

Full-duplex(全双工）全双工是在通道中同时双向数据传输的能力。

Half-duplex(半双工）在通道中同时只能沿着一个方向传输数据。

10.5.7 DNS服务器

DNS 是域名系统（Domain Name System）的缩写，是因特网的一项核心服务，它作为可以将域名和IP地址相互映射的一个分布式数据库，能够使人更方便的访问互联网，而不用去记住能够被机器直接读取的IP地址串。

它是由解析器以及域名服务器组成的。域名服务器是指保存有该网络中所有主机的域名和对应IP地址，并具有将域名转换为IP地址功能的服务器。

10.5.8 局域网(local area network;LAN)

一种覆盖一座或几座大楼、一个校园或者一个厂区等地理区域的小范围的计算机网

（1）覆盖的地理范围较小，只在一个相对独立的局部范围内联，如一座或集中的建筑群内。

（2）使用专门铺设的传输介质进行联网，数据传输速率高（10Mb/s～10Gb/s）

（3）通信延迟时间短，可靠性较高

（4）局域网可以支持多种传输介质

10.5.9 广域网(wide area network;WAN)

一种用来实现不同地区的局域网或城域网的互连，可提供不同地区、城市和国家之间的计算机通信的远程计算机网。

覆盖的范围比局域网（LAN）和城域网（MAN）都广。广域网[1]的通信子网主要使用分组交换技术。

广域网的通信子网可以利用公用分组交换网、卫星通信网和无线分组交换网，它将分布在不同地区的局域网或计算机系统互连起来，达到资源共享的目的。如互联网是世界范围内最大的广域网。

1、适应大容量与突发性通信的要求；

2、适应综合业务服务的要求；

3、开放的设备接口与规范化的协议；

4、完善的通信服务与网络管理。

10.5.10 端口

逻辑意义上的端口，一般是指TCP/IP协议中的端口，端口号的范围从0到65535，比如用于浏览网页服务的80端口，用于FTP服务的21端口等等。

（1）端口号小于256的定义为常用端口，服务器一般都是通过常用端口号来识别的。

（2）客户端只需保证该端口号在本机上是惟一的就可以了。客户端口号因存在时间很短暂又称临时端口号；

（3）大多数TCP/IP实现给临时端口号分配1024—5000之间的端口号。大于5000的端口号是为其他服务器预留的。

10.5.11 MTU

MTU:通信术语 最大传输单元（Maximum Transmission Unit，MTU）

是指一种通信协议的某一层上面所能通过的最大数据包大小（以字节为单位）。最大传输单元这个参数通常与通信接口有关（网络接口卡、串口等）。

以下是一些协议的MTU

```
FDDI协议：4352字节
以太网（Ethernet）协议：1500字节
PPPoE（ADSL）协议：1492字节
X.25协议（Dial Up/Modem）：576字节
Point-to-Point：4470字节
```

10.6 常见网络知识面试题:

1.TCP如何建立链接

2.TCP如何通信

3.TCP如何关闭链接

4.什么是滑动窗口

5.什么是半关闭

6.局域网内两台机器如何利用tcp/ip通信?

7.internet上两台主机如何进行通信

8.如何在internet上识别唯一一个进程

答：通过“IP地址+端口号”来区 分不同的服务的

9.为什么说tcp是可靠的链接，udp不可靠

10.路由器和交换机的区别

11.点到点，端到端

第 11 章

socket编程

socket这个词可以表示很多概念：

在TCP/IP协议中，“IP地址+TCP或UDP端口号”唯一标识网络通讯中的一个进程，“IP地址+端口号”就称为socket。

在TCP协议中，建立连接的两个进程各自有一个socket来标识，那么这两个socket组成的socket pair就唯一标识一个连接。socket本身有“插座”的意思，因此用来描述网络连接的一对一关系。

TCP/IP协议最早在BSD UNIX上实现，为TCP/IP协议设计的应用层编程接口称为socket API。

本章的主要内容是socket API，主要介绍TCP协议的函数接口，最后介绍UDP协议和UNIX Domain Socket的函数接口。

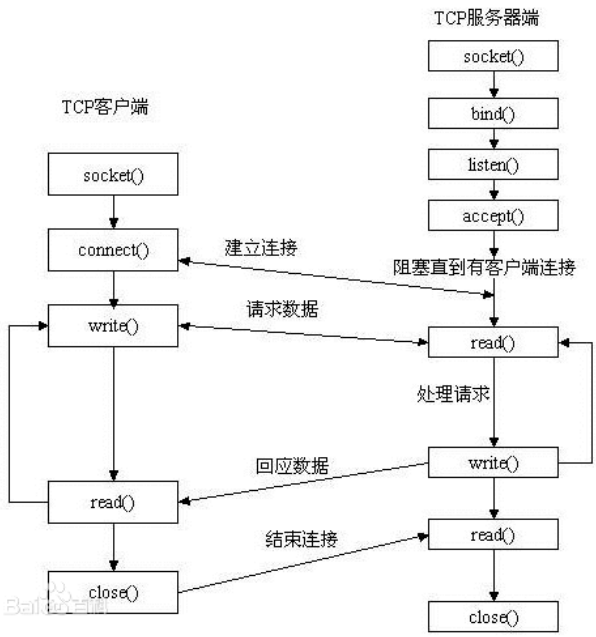

图 11.1: socketAPI

11.1 预备知识

11.1.1　网络字节序

我们已经知道，内存中的多字节数据相对于内存地址有大端和小端之分，磁盘文件中的多字节数据相对于文件中的偏移地址也有大端小端之分。网络数据流同样有大端小端之分，那么如何定义网络数据流的地址呢？发送主机通常将发送缓冲区中的数据按内存地址从低到高的顺序发出，接收主机把从网络上接到的字节依次保存在接收缓冲区中，也是按内存地址从低到高的顺序保存，因此，网络数据流的地址应这样规定：先发出的数据是低地址，后发出的数据是高地址。

TCP/IP协议规定，网络数据流应采用大端字节序，即低地址高字节。例如上一节的UDP段格式，地址0-1是16位的源端口号，如果这个端口号是1000（0x3e8），则地址0是0x03，地址1是0xe8，也就是先发0x03，再发0xe8，这16位在发送主机的缓冲区中也应该是低地址存0x03，高地址存0xe8。但是，如果发送主机是小端字节序的，这16位被解释成0xe803，而不是1000。因此，发送主机把1000填到发送缓冲区之前需要做字节序的转换。同样地，接收主机如果是小端字节序的，接到16位的源端口号也要做字节序的转换。如果主机是大端字节序的，发送和接收都不需要做转换。同理，32位的IP地址也要考虑网络字节序和主机字节序的问题。

为使网络程序具有可移植性，使同样的C代码在大端和小端计算机上编译后都能正常运行，可以调用以下库函数做网络字节序和主机字节序的转换。

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);

h表示host，n表示network，l表示32位长整数，s表示16位短整数。
如果主机是小端字节序，这些函数将参数做相应的大小端转换然后返回，如果主机是大端字节序，这些函数不做转换，将参数原封不动地返回。
```

11.1.2　IP地址转换函数

早期

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);

只能处理IPv4的ip地址
不可重入函数
注意参数是struct in_addr
```

现在

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

int inet_pton(int af, const char *src, void *dst);
const char *inet_ntop(int af, const void *src,  char *dst, socklen_t size);

支持IPv4和IPv6
可重入函数
```

其中inet_pton和inet_ntop不仅可以转换IPv4的in_addr，还可以转换IPv6的in6_addr，因此函数接口是void *addrptr。

11.1.3 sockaddr数据结构

strcut sockaddr 很多网络编程函数诞生早于IPv4协议，那时候都使用的是sockaddr结构体,为了向前兼容，现在sockaddr退化成了（void *）的作用，传递一个地址给函数，至于这个函数是sockaddr_in还是sockaddr_in6，由地址族确定，然后函数内部再强制类型转化为所需的地址类型

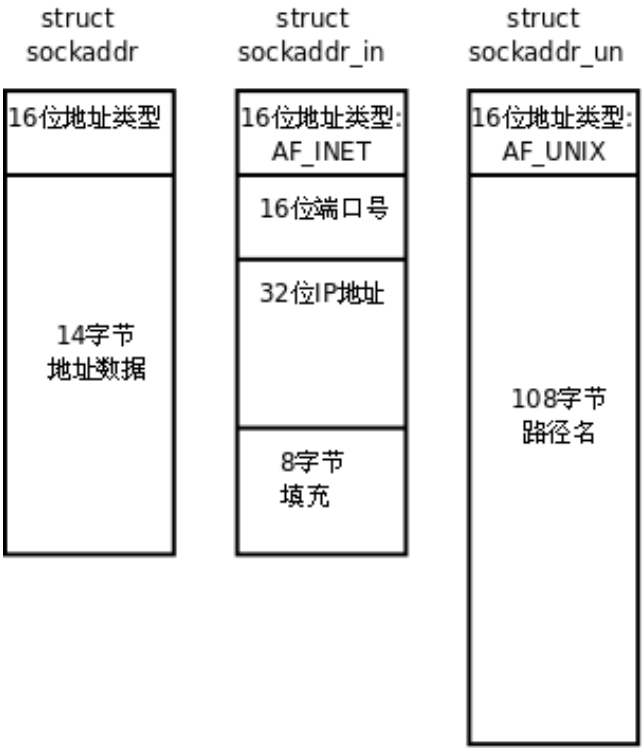

图 11.2: sockaddr数据结构

```
struct sockaddr {
    sa_family_t        sa_family;           /* address family, AF_xxx       */
    char                sa_data[14];       /* 14 bytes of protocol address       */
};

struct sockaddr_in {
  __kernel_sa_family_t        sin_family;       /* Address family               */
```

```
  __be16                sin_port;               /* Port number                          */
  struct in_addr        sin_addr;            /* Internet address                */

/* Pad to size of `struct sockaddr'. */
unsigned char __pad[__SOCK_SIZE__ - sizeof(short int) -
            sizeof(unsigned short int) - sizeof(struct in_addr)];
};

/* Internet address. */
struct in_addr {
    __be32        s_addr;
};

struct sockaddr_in6 {
    unsigned short int        sin6_family;    /* AF_INET6 */
    __be16                        sin6_port;      /* Transport layer port # */
    __be32                        sin6_flowinfo;  /* IPv6 flow information */
    struct in6_addr                sin6_addr;      /* IPv6 address */
    __u32                         sin6_scope_id;  /* scope id (new in RFC2553) */
};

struct in6_addr {
    union {
        __u8                u6_addr8[16];
        __be16                u6_addr16[8];
        __be32                u6_addr32[4];
    } in6_u;
    #define s6_addr                         in6_u.u6_addr8
    #define s6_addr16                in6_u.u6_addr16
    #define s6_addr32                in6_u.u6_addr32
};


#define UNIX_PATH_MAX        108
struct sockaddr_un {
__kernel_sa_family_t sun_family; /* AF_UNIX */
char sun_path[UNIX_PATH_MAX];        /* pathname */
};
```

Pv4和IPv6的地址格式定义在netinet/in.h中，IPv4地址用sockaddr_in结构体表示，包括16位端口号和32位IP地址，IPv6地址用sockaddr_in6结构体表示，包括16位端口号、128位IP地址和一些控制字段。UNIX Domain Socket的地址格式定义在sys/un.h中，用sockaddr_un结构体表示。各种socket地址结构体的开头都是相同的，前16位表示整个结构体的长度（并不是所有UNIX的实现都有长度字段，如Linux就没有），后16位表示地址类型。IPv4、IPv6和Unix Domain Socket的地址类型分别定义为常数AF_INET、AF_INET6、AF_UNIX。这样，只要取得某种sockaddr结构体的首地址，不需要知道具体是哪种类型的sockaddr结构体，就可以根据地址类型字段确定结构体中的内容。因此，socket API可以接受各种类型的sockaddr结构体指针做参数，例如bind、accept、connect等函数，这些函数的参数应该设计成void *类型以便接受各种类型的指针，但是sock API的实现早于ANSI C标准化，那时还没有void *类型，因此这些函数的参数都用struct sockaddr *类型表示，在传递参数之前要强制类型转换一下，例如：

```
struct sockaddr_in servaddr;
/* initialize servaddr */
bind(listen_fd, (struct sockaddr *)&servaddr, sizeof(servaddr));
```

11.2 网络套接字函数

11.2.1 socket

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

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

domain:
    AF_INET 这是大多数用来产生socket的协议，使用TCP或UDP来传输，用IPv4的地址
    AF_INET6 与上面类似，不过是来用IPv6的地址
    AF_UNIX 本地协议，使用在Unix和Linux系统上，一般都是当客户端和服务器在同一台及其上的时候使用
type:
    SOCK_STREAM 这个协议是按照顺序的、可靠的、数据完整的基于字节流的连接。这是一个使用最多的socket类型，这个socket是使用TCP来进行传输。
    SOCK_DGRAM 这个协议是无连接的、固定长度的传输调用。该协议是不可靠的，使用UDP来进行它的连接。
    SOCK_SEQPACKET 这个协议是双线路的、可靠的连接，发送固定长度的数据包进行传输。必须把这个包完整的接受才能进行读取。
    SOCK_RAW 这个socket类型提供单一的网络访问，这个socket类型使用ICMP公共协议。（ping、traceroute使用该协议）
    SOCK_RDM 这个类型是很少使用的，在大部分的操作系统上没有实现，它是提供给数据链路层使用，不保证数据包的顺序
protocol:
    0 默认协议
返回值：
    成功返回一个新的文件描述符，失败返回-1，设置errno
```

socket()打开一个网络通讯端口，如果成功的话，就像open()一样返回一个文件描述符，应用程序可以像读写文件一样用read/write在网络上收发数据，如果socket()调用出错则返回-1。对于IPv4，domain参数指定为AF_INET。对于TCP协议，type参数指定为SOCK_STREAM，表示面向流的传输协议。如果是UDP协议，则type参数指定为SOCK_DGRAM，表示面向数据报的传输协议。protocol参数的介绍从略，指定为0即可。

11.2.2 bind

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

```
sockfd：
    socket文件描述符
addr:
    构造出IP地址加端口号
addrlen:
    sizeof(addr)长度
返回值：
    成功返回0，失败返回-1，设置errno
```

服务器程序所监听的网络地址和端口号通常是固定不变的，客户端程序得知服务器程序的地址和端口号后就可以向服务器发起连接，因此服务器需要调用bind绑定一个固定的网络地址和端口号。

bind()的作用是将参数sockfd和addr绑定在一起，使sockfd这个用于网络通讯的文件描述符监听addr所描述的地址和端口号。前面讲过，struct sockaddr *是一个通用指针类型，addr参数实际上可以接受多种协议的sockaddr结构体，而它们的长度各不相同，所以需要第三个参数addrlen指定结构体的长度。如：

```
struct sockaddr_in servaddr;
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(8000);
```

首先将整个结构体清零，然后设置地址类型为AF_INET，网络地址为INADDR_ANY，这个宏表示本地的任意IP地址，因为服务器可能有多个网卡，每个网卡也可能绑定多个IP地址，这样设置可以在所有的IP地址上监听，直到与某个客户端建立了连接时才确定下来到底用哪个IP地址，端口号为8000。

11.2.3 listen

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int listen(int sockfd, int backlog);
sockfd:
    socket文件描述符
backlog:
    排队建立3次握手队列和刚刚建立3次握手队列的链接数和
```

查看系统默认backlog

```
cat /proc/sys/net/ipv4/tcp_max_syn_backlog
```

典型的服务器程序可以同时服务于多个客户端，当有客户端发起连接时，服务器调用的accept()返回并接受这个连接，如果有大量的客户端发起连接而服务器来不及处理，尚未accept的客户端就处于连接等待状态，listen()声明sockfd处于监听状态，并且最多允许有backlog个客户端处于连接待状态，如果接收到更多的连接请求就忽略。listen()成功返回0，失败返回-1。

11.2.4 accept

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
sockdf:
    socket文件描述符
addr:
    传出参数，返回链接客户端地址信息，含IP地址和端口号
addrlen:
    传入传出参数（值-结果）,传入sizeof(addr)大小，函数返回时返回真正接收到地址结构体的大小
返回值：
    成功返回一个新的socket文件描述符，用于和客户端通信，失败返回-1，设置errno
```

三方握手完成后，服务器调用accept()接受连接，如果服务器调用accept()时还没有客户端的连接请求，就阻塞等待直到有客户端连接上来。addr是一个传出参数，accept()返回时传出客户端的地址和端口号。addrlen参数是一个传入传出参数（value-result argument），传入的是调用者提供的缓冲区addr的长度以避免缓冲区溢出问题，传出的是客户端地址结构体的实际长度（有可能没有占满调用者提供的缓冲区）。如果给addr参数传NULL，表示不关心客户端的地址。

我们的服务器程序结构是这样的：

```
while (1) {
    cliaddr_len = sizeof(cliaddr);
    connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
    n = read(connfd, buf, MAXLINE);
    ......
    close(connfd);
}
```

整个是一个while死循环，每次循环处理一个客户端连接。由于cliaddr_len是传入传出参数，每次调用accept()之前应该重新赋初值。accept()的参数listenfd是先前的监听文件描述符，而accept()的返回值是另外一个文件描述符connfd，之后与客户端之间就通过这个connfd通讯，最后关闭connfd断开连接，而不关闭listenfd，再次回到循环开头listenfd仍然用作accept的参数。accept()成功返回一个文件描述符，出错返回-1。

11.2.5 connect

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
sockdf:
    socket文件描述符
addr:
    传入参数，指定服务器端地址信息，含IP地址和端口号
addrlen:
    传入参数,传入sizeof(addr)大小
返回值：
    成功返回0，失败返回-1，设置errno
```

客户端需要调用connect()连接服务器，connect和bind的参数形式一致，区别在于bind的参数是自己的地址，而connect的参数是对方的地址。connect()成功返回0，出错返回-1。

11.3 C/S模型-TCP

下图是基于TCP协议的客户端/服务器程序的一般流程：

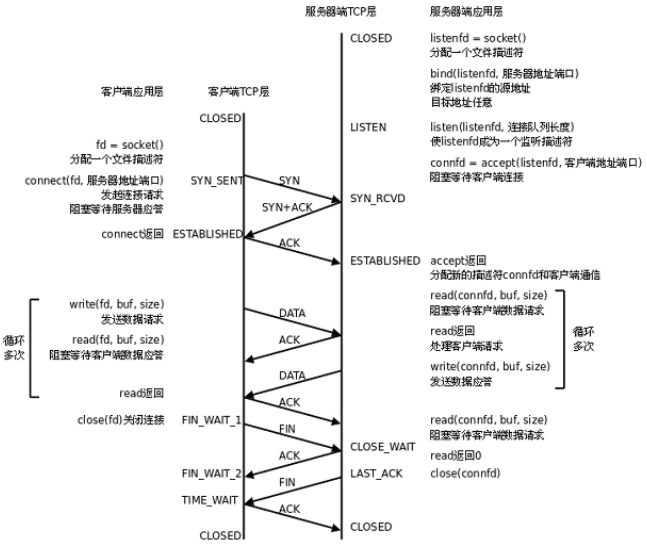

图 11.3: TCP协议通讯流程

服务器调用socket()、bind()、listen()完成初始化后，调用accept()阻塞等待，处于监听端口的状态，客户端调用socket()初始化后，调用connect()发出SYN段并阻塞等待服务器应答，服务器应答一个SYN-ACK段，客户端收到后从connect()返回，同时应答一个ACK段，服务器收到后从accept()返回。

数据传输的过程：

建立连接后，TCP协议提供全双工的通信服务，但是一般的客户端/服务器程序的流程是由客户端主动发起请求，服务器被动处理请求，一问一答的方式。因此，服务器从accept()返回后立刻调用read()，读socket就像读管道一样，如果没有数据到达就阻塞等待，这时客

户端调用write()发送请求给服务器，服务器收到后从read()返回，对客户端的请求进行处理，在此期间客户端调用read()阻塞等待服务器的应答，服务器调用write()将处理结果发回给客户端，再次调用read()阻塞等待下一条请求，客户端收到后从read()返回，发送下一条请求，如此循环下去。

如果客户端没有更多的请求了，就调用close()关闭连接，就像写端关闭的管道一样，服务器的read()返回0，这样服务器就知道客户端关闭了连接，也调用close()关闭连接。注意，任何一方调用close()后，连接的两个传输方向都关闭，不能再发送数据了。如果一方调用shutdown()则连接处于半关闭状态，仍可接收对方发来的数据。

在学习socket API时要注意应用程序和TCP协议层是如何交互的： *应用程序调用某个socket函数时TCP协议层完成什么动作，比如调用connect()会发出SYN段 *应用程序如何知道TCP协议层的状态变化，比如从某个阻塞的socket函数返回就表明TCP协议收到了某些段，再比如read()返回0就表明收到了FIN段

11.3.1 server

下面通过最简单的客户端/服务器程序的实例来学习socket API。

server.c的作用是从客户端读字符，然后将每个字符转换为大写并回送给客户端。

```
/* server.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MAXLINE 80
#define SERV_PORT 8000

int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, connfd;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];
    int i, n;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    listen(listenfd, 20);

    printf("Accepting connections ...\n");
    while (1) {
```

```
        cliaddr_len = sizeof(cliaddr);
        connfd = accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);

        n = read(connfd, buf, MAXLINE);
        printf("received from %s at PORT %d\n",
               inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
               ntohs(cliaddr.sin_port));

        for (i = 0; i < n; i++)
            buf[i] = toupper(buf[i]);
        write(connfd, buf, n);
        close(connfd);
    }
}
```

11.3.2 client

client.c的作用是从命令行参数中获得一个字符串发给服务器，然后接收服务器返回的字符串并打印。

```
/* client.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MAXLINE 80
#define SERV_PORT 8000

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;
    char *str;

    if (argc != 2) {
        fputs("usage: ./client message\n", stderr);
        exit(1);
    }
    str = argv[1];

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
```

```
    write(sockfd, str, strlen(str));

    n = read(sockfd, buf, MAXLINE);
    printf("Response from server:\n");
    write(STDOUT_FILENO, buf, n);

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

由于客户端不需要固定的端口号，因此不必调用bind()，客户端的端口号由内核自动分配。注意，客户端不是不允许调用bind()，只是没有必要调用bind()固定一个端口号，服务器也不是必须调用bind()，但如果服务器不调用bind()，内核会自动给服务器分配监听端口，每次启动服务器时端口号都不一样，客户端要连接服务器就会遇到麻烦。

客户端和服务器启动后可以查看链接情况：

```
netstat -apn|grep 8000
```

11.4 C/S模型-UDP

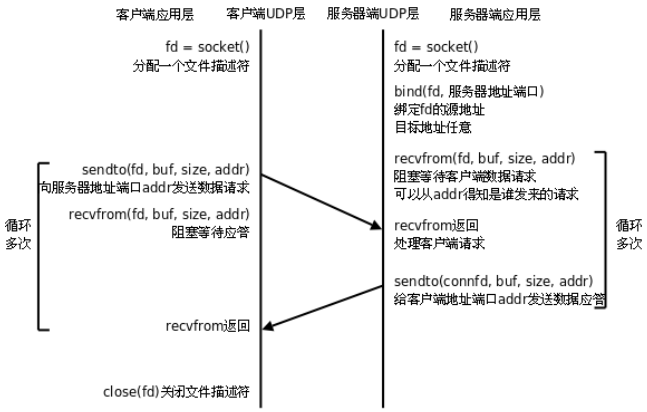

图 11.4: UDP处理模型

由于UDP不需要维护连接，程序逻辑简单了很多，但是UDP协议是不可靠的，实际上有很多保证通讯可靠性的机制需要在应用层实现。

编译运行server，在两个终端里各开一个client与server交互，看看server是否具有并发服务的能力。用Ctrl+C关闭server，然后再运行server，看此时client还能否和server联系上。和前面TCP程序的运行结果相比较，体会无连接的含义。

11.4.1 server

```
/* server.c */
#include <stdio.h>
```

```
#include <string.h>
#include <netinet/in.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 8000

int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int sockfd;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];
    int i, n;

    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    printf("Accepting connections ...\n");
    while (1) {
        cliaddr_len = sizeof(cliaddr);
        n = recvfrom(sockfd, buf, MAXLINE, 0, (struct sockaddr *)&cliaddr, &cliaddr_len);
        if (n == -1)
            perr_exit("recvfrom error");
        printf("received from %s at PORT %d\n",
               inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
               ntohs(cliaddr.sin_port));

        for (i = 0; i < n; i++)
            buf[i] = toupper(buf[i]);
        n = sendto(sockfd, buf, n, 0, (struct sockaddr *)&cliaddr, sizeof(cliaddr));
        if (n == -1)
            perr_exit("sendto error");
    }
}
```

11.4.2 client

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include "wrap.h"

#define MAXLINE 80
```

```
#define SERV_PORT 8000

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    int sockfd, n;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];
    socklen_t servaddr_len;

    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    while (fgets(buf, MAXLINE, stdin) != NULL) {
        n = sendto(sockfd, buf, strlen(buf), 0, (struct sockaddr *)&servaddr, sizeof(servaddr));
        if (n == -1)
            perr_exit("sendto error");

        n = recvfrom(sockfd, buf, MAXLINE, 0, NULL, 0);
        if (n == -1)
            perr_exit("recvfrom error");

        Write(STDOUT_FILENO, buf, n);
    }

    Close(sockfd);
    return 0;
}
```

11.5 出错处理封装函数

上面的例子不仅功能简单，而且简单到几乎没有什么错误处理，我们知道，系统调用不能保证每次都成功，必须进行出错处理，这样一方面可以保证程序逻辑正常，另一方面可以迅速得到故障信息。

为使错误处理的代码不影响主程序的可读性，我们把与socket相关的一些系统函数加上错误处理代码包装成新的函数，做成一个模块wrap.c：

11.5.1 wrap.c

```
/* wrap.c */
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>

void perr_exit(const char *s)
{
```

```
    perror(s);
    exit(1);
}

int Accept(int fd, struct sockaddr *sa, socklen_t *salenptr)
{
    int n;

again:
    if ( (n = accept(fd, sa, salenptr)) < 0) {
        if ((errno == ECONNABORTED) || (errno == EINTR))
            goto again;
        else
            perr_exit("accept error");
    }
    return n;
}

void Bind(int fd, const struct sockaddr *sa, socklen_t salen)
{
    if (bind(fd, sa, salen) < 0)
        perr_exit("bind error");
}

void Connect(int fd, const struct sockaddr *sa, socklen_t salen)
{
    if (connect(fd, sa, salen) < 0)
        perr_exit("connect error");
}

void Listen(int fd, int backlog)
{
    if (listen(fd, backlog) < 0)
        perr_exit("listen error");
}

int Socket(int family, int type, int protocol)
{
    int n;

    if ( (n = socket(family, type, protocol)) < 0)
        perr_exit("socket error");
    return n;
}

ssize_t Read(int fd, void *ptr, size_t nbytes)
{
    ssize_t n;

again:
    if ( (n = read(fd, ptr, nbytes)) == -1) {
        if (errno == EINTR)
            goto again;
        else
            return -1;
    }
```

```
    return n;
}

ssize_t Write(int fd, const void *ptr, size_t nbytes)
{
    ssize_t n;

again:
    if ( (n = write(fd, ptr, nbytes)) == -1) {
        if (errno == EINTR)
            goto again;
        else
            return -1;
    }
    return n;
}

void Close(int fd)
{
    if (close(fd) == -1)
        perr_exit("close error");
}
ssize_t Readn(int fd, void *vptr, size_t n)
{
    size_t  nleft;
    ssize_t nread;
    char   *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0;
            else
                return -1;
        } else if (nread == 0)
            break;

        nleft -= nread;
        ptr += nread;
    }
    return n - nleft;
}

ssize_t Writen(int fd, const void *vptr, size_t n)
{
    size_t nleft;
    ssize_t nwritten;
    const char *ptr;

    ptr = vptr;
    nleft = n;
    while (nleft > 0) {
        if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
            if (nwritten < 0 && errno == EINTR)
```

```
                nwritten = 0;
            else
                return -1;
        }

        nleft -= nwritten;
        ptr += nwritten;
    }
    return n;
}
static ssize_t my_read(int fd, char *ptr)
{
    static int read_cnt;
    static char *read_ptr;
    static char read_buf[100];

    if (read_cnt <= 0) {
again:
        if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
            if (errno == EINTR)
                goto again;
            return -1;
        } else if (read_cnt == 0)
            return 0;
        read_ptr = read_buf;
    }
    read_cnt--;
    *ptr = *read_ptr++;
    return 1;
}

ssize_t Readline(int fd, void *vptr, size_t maxlen)
{
    ssize_t n, rc;
    char    c, *ptr;

    ptr = vptr;
    for (n = 1; n < maxlen; n++) {
        if ( (rc = my_read(fd, &c)) == 1) {
            *ptr++ = c;
            if (c  == '\n')
                break;
        } else if (rc == 0) {
            *ptr = 0;
            return n - 1;
        } else
            return -1;
    }
    *ptr  = 0;
    return n;
}
```

11.5.2 wrap.h

```
/* wrap.h */
#ifndef __WRAP_H_
#define __WRAP_H_

void perr_exit(const char *s);
int Accept(int fd, struct sockaddr *sa, socklen_t *salenptr);
void Bind(int fd, const struct sockaddr *sa, socklen_t salen);
void Connect(int fd, const struct sockaddr *sa, socklen_t salen);
void Listen(int fd, int backlog);
int Socket(int family, int type, int protocol);
ssize_t Read(int fd, void *ptr, size_t nbytes);
ssize_t Write(int fd, const void *ptr, size_t nbytes);
void Close(int fd);
ssize_t Readn(int fd, void *vptr, size_t n);
ssize_t Writen(int fd, const void *vptr, size_t n);
static ssize_t my_read(int fd, char *ptr);
ssize_t Readline(int fd, void *vptr, size_t maxlen);

#endif
```

11.6 练习

1.编写获取网络服务器时钟服务器/客户机程序。

2.实现网络版本聊天室服务器客户机

第 12 章

高并发服务器

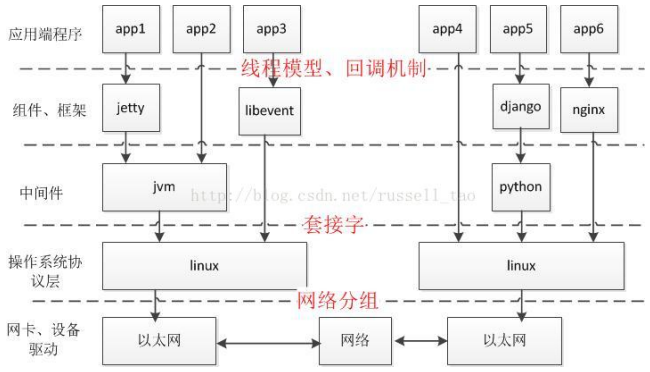

图 12.1: 并发服务器开发

12.1 多进程并发服务器

使用多进程并发服务器时要考虑以下几点：

1.父最大文件描述个数(父进程中需要close关闭accept返回的新文件描述符)

2.系统内创建进程个数(内存大小相关)

3.进程创建过多是否降低整体服务性能(进程调度)

12.1.1 server

```
/* server.c */
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 8000
```

```
void do_sigchild(int num)
{
    waitpid(0, NULL, WNOHANG);
}
int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, connfd;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];
    int i, n;
    pid_t pid;

    struct sigaction newact;
    newact.sa_handler = do_sigchild;
    sigemptyset(&newact.sa_mask);
    newact.sa_flags = 0;
    sigaction(SIGCHLD, &newact, NULL);

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    Listen(listenfd, 20);

    printf("Accepting connections ...\n");
    while (1) {
        cliaddr_len = sizeof(cliaddr);
        connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);

        pid = fork();
        if (pid == 0) {
            Close(listenfd);
            while (1) {
                n = Read(connfd, buf, MAXLINE);
                if (n == 0) {
                    printf("the other side has been closed.\n");
                    break;
                }
                printf("received from %s at PORT %d\n",
                        inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
                        ntohs(cliaddr.sin_port));

                for (i = 0; i < n; i++)
                    buf[i] = toupper(buf[i]);
                Write(connfd, buf, n);
            }
            Close(connfd);
            return 0;
        }
```

```
        else if (pid > 0) {
            Close(connfd);
        }
        else
            perr_exit("fork");
    }
}
```

12.1.2 client

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 8000

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (fgets(buf, MAXLINE, stdin) != NULL) {
        Write(sockfd, buf, strlen(buf));
        n = Read(sockfd, buf, MAXLINE);
        if (n == 0)
            printf("the other side has been closed.\n");
        else
            Write(STDOUT_FILENO, buf, n);
    }

    Close(sockfd);
    return 0;
}
```

12.2 多线程并发服务器

在使用线程模型开发服务器时需考虑以下问题：

1.调整进程内最大文件描述符上限
2.线程如有共享数据，考虑线程同步
3.服务于客户端线程退出时，退出处理。（退出值，分离态）
4.系统负载，随着链接客户端增加，导致其它线程不能及时得到CPU

12.2.1 server

```
/* server.c */
#include <stdio.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 8000
struct s_info {
    struct sockaddr_in cliaddr;
    int connfd;
};
void *do_work(void *arg)
{
    int n,i;
    struct s_info *ts = (struct s_info*)arg;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];

    /* 可以在创建线程前设置线程创建属性,设为分离态,哪种效率高内？ */
    pthread_detach(pthread_self());

    while (1) {
        n = Read(ts->connfd, buf, MAXLINE);
        if (n == 0) {
            printf("the other side has been closed.\n");
            break;
        }
        printf("received from %s at PORT %d\n",
                inet_ntop(AF_INET, &(*ts).cliaddr.sin_addr, str, sizeof(str)),
                ntohs((*ts).cliaddr.sin_port));

        for (i = 0; i < n; i++)
            buf[i] = toupper(buf[i]);
        Write(ts->connfd, buf, n);
    }
    Close(ts->connfd);
}
int main(void)
{
    struct sockaddr_in servaddr, cliaddr;
    socklen_t cliaddr_len;
    int listenfd, connfd;
    int i = 0;
    pthread_t tid;
```

```
    struct s_info ts[383];

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    Listen(listenfd, 20);

    printf("Accepting connections ...\n");
    while (1) {
        cliaddr_len = sizeof(cliaddr);
        connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);
        ts[i].cliaddr = cliaddr;
        ts[i].connfd = connfd;
        /* 达到线程最大数时，pthread_create出错处理，增加服务器稳定性 */
        pthread_create(&tid, NULL, do_work, (void*)&ts[i]);
        i++;
    }
    return 0;
}
```

12.2.2 client

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 8000

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
```

```
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        Write(sockfd, buf, strlen(buf));
        n = Read(sockfd, buf, MAXLINE);
        if (n == 0)
            printf("the other side has been closed.\n");
        else
            Write(STDOUT_FILENO, buf, n);
    }

    Close(sockfd);
    return 0;
}
```

12.3　多路I/O转接服务器

12.3.1　三种模型性能分析

12.3.2　select

1.select能监听的文件描述符个数受限于FD_SETSIZE,一般为1024，单纯改变进程打开的文件描述符个数并不能改变select监听文件个数

2.解决1024以下客户端时使用select是很合适的，但如果链接客户端过多，select采用的是轮询模型，会大大降低服务器响应效率，不应在select上投入更多精力

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

/* According to earlier standards */
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout);

nfds: 监控的文件描述符集里最大文件描述符加1，因为此参数会告诉内核检测前多少个文件描述符的状态
readfds：监控有读数据到达文件描述符集合，传入传出参数
writefds：监控写数据到达文件描述符集合，传入传出参数
exceptfds：监控异常发生达文件描述符集合,如带外数据到达异常，传入传出参数
timeout：定时阻塞监控时间，3种情况
    1.NULL，永远等下去
    2.设置timeval，等待固定时间
    3.设置timeval里时间均为0，检查描述字后立即返回，轮询

struct timeval {
           long    tv_sec;         /* seconds */
           long    tv_usec;        /* microseconds */
       };



void FD_CLR(int fd, fd_set *set);      把文件描述符集合里fd清0
```

```
int  FD_ISSET(int fd, fd_set *set);    测试文件描述符集合里fd是否置1
void FD_SET(int fd, fd_set *set);      把文件描述符集合里fd位置1
void FD_ZERO(fd_set *set);             把文件描述符集合里所有位清0
```

```
/* server.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 8000

int main(int argc, char *argv[])
{
    int i, maxi, maxfd, listenfd, connfd, sockfd;
    int nready, client[FD_SETSIZE];     /* FD_SETSIZE 默认为 1024 */
    ssize_t n;
    fd_set rset, allset;
    char buf[MAXLINE];
    char str[INET_ADDRSTRLEN];          /* #define INET_ADDRSTRLEN 16 */
    socklen_t cliaddr_len;
    struct sockaddr_in        cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(SERV_PORT);

    Bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    Listen(listenfd, 20);   /* 默认最大128 */

    maxfd = listenfd;                /* 初始化 */
    maxi = -1;                        /* client[]的下标 */

    for (i = 0; i < FD_SETSIZE; i++)
        client[i] = -1;         /* 用-1初始化client[] */

    FD_ZERO(&allset);
    FD_SET(listenfd, &allset);  /* 构造select监控文件描述符集 */

    for ( ; ; ) {
        rset = allset;        /* 每次循环时都从新设置select监控信号集 */
        nready = select(maxfd+1, &rset, NULL, NULL, NULL);
        if (nready < 0)
            perr_exit("select error");
```

```
        if (FD_ISSET(listenfd, &rset)) { /* new client connection */
            cliaddr_len = sizeof(cliaddr);
            connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &cliaddr_len);

            printf("received from %s at PORT %d\n",
                    inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
                    ntohs(cliaddr.sin_port));

            for (i = 0; i < FD_SETSIZE; i++)
                if (client[i] < 0) {
                    client[i] = connfd; /* 保存accept返回的文件描述符到client[]里 */
                    break;
                }
            /* 达到select能监控的文件个数上限 1024 */
            if (i == FD_SETSIZE) {
                fputs("too many clients\n", stderr);
                exit(1);
            }

            FD_SET(connfd, &allset);        /* 添加一个新的文件描述符到监控信号集里 */
            if (connfd > maxfd)
                maxfd = connfd; /* select第一个参数需要 */
            if (i > maxi)
                maxi = i;        /* 更新client[]最大下标值 */

            if (--nready == 0)
               continue;        /* 如果没有更多的就绪文件描述符继续回到上面select阻塞监听,负责处理未
处理完的就绪文件描述符 */
        }

        for (i = 0; i <= maxi; i++) {        /* 检测哪个clients 有数据就绪 */
            if ( (sockfd = client[i]) < 0)
                continue;
            if (FD_ISSET(sockfd, &rset)) {
                if ( (n = Read(sockfd, buf, MAXLINE)) == 0) {
                    /* 当client关闭链接时, 服务器端也关闭对应链接 */
                    Close(sockfd);
                    FD_CLR(sockfd, &allset);    /* 解除select监控此文件描述符 */
                    client[i] = -1;
                } else {
                    int j;
                    for (j = 0; j < n; j++)
                        buf[j] = toupper(buf[j]);
                    Write(sockfd, buf, n);
                }

                if (--nready == 0)
                    break;
            }
        }
    }

    close(listenfd);

    return 0;
}
```

server

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 8000

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (fgets(buf, MAXLINE, stdin) != NULL) {
        Write(sockfd, buf, strlen(buf));
        n = Read(sockfd, buf, MAXLINE);
        if (n == 0)
            printf("the other side has been closed.\n");
        else
            Write(STDOUT_FILENO, buf, n);
    }

    Close(sockfd);
    return 0;
}
```

client

pselect 给出pselect原型，此模型用的不多，有需要的同学可参考select模型自行编写C/S

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

int pselect(int nfds, fd_set *readfds, fd_set *writefds,
```

```
            fd_set *exceptfds, const struct timespec *timeout,
            const sigset_t *sigmask);

struct timespec {
               long    tv_sec;         /* seconds */
               long    tv_nsec;        /* nanoseconds */
           };

用sigmask替代当前进程的阻塞信号集，调用返回后还原原有阻塞信号集
```

12.3.3 poll

```
#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

struct pollfd {
  int   fd;         /* 文件描述符 */
  short events;     /* 监控的事件 */
  short revents;    /* 监控事件中满足条件返回的事件 */
};

POLLIN普通或带外优先数据可读,即POLLRDNORM | POLLRDBAND
POLLRDNORM-数据可读
POLLRDBAND-优先级带数据可读
POLLPRI 高优先级可读数据

POLLOUT普通或带外数据可写
POLLWRNORM-数据可写
POLLWRBAND-优先级带数据可写

POLLERR 发生错误
POLLHUP 发生挂起
POLLNVAL 描述字不是一个打开的文件

nfds 监控数组中有多少文件描述符需要被监控

timeout 毫秒级等待
   -1：阻塞等，#define INFTIM   -1 Linux中没有定义此宏
    0：立即返回，不阻塞进程
   >0：等待指定毫秒数，如当前系统时间精度不够毫秒，向上取值
```

如果不再监控某个文件描述符时，可以把pollfd中，fd设置为-1，poll不再监控此pollfd，下次返回时，把revents设置为0。

ppoll GNU定义了ppoll(非POSIX标准)，可以支持设置信号屏蔽字，大家可参考poll模型自行实现C/S

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <poll.h>

int ppoll(struct pollfd *fds, nfds_t nfds,
     const struct timespec *timeout_ts, const sigset_t *sigmask);
```

```
/* server.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <poll.h>
#include <errno.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 8000
#define OPEN_MAX 1024

int main(int argc, char *argv[])
{
    int        i, j, maxi, listenfd, connfd, sockfd;
    int        nready;
    ssize_t        n;
    char buf[MAXLINE], str[INET_ADDRSTRLEN];
    socklen_t clilen;
    struct pollfd client[OPEN_MAX];
    struct sockaddr_in cliaddr, servaddr;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    Listen(listenfd, 20);

    client[0].fd = listenfd;
    client[0].events = POLLRDNORM;  /* listenfd监听普通读事件 */

    for (i = 1; i < OPEN_MAX; i++)
        client[i].fd = -1;              /* 用-1初始化client[]里剩下元素 */
    maxi = 0;                                       /* client[]数组有效元素中最大元素下标  */

    for ( ; ; ) {
        nready = poll(client, maxi+1, -1);      /* 阻塞 */

        if (client[0].revents & POLLRDNORM) {       /* 有客户端链接请求 */
```

```
            clilen = sizeof(cliaddr);
            connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &clilen);
            printf("received from %s at PORT %d\n",
                    inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)),
                    ntohs(cliaddr.sin_port));

            for (i = 1; i < OPEN_MAX; i++)
                if (client[i].fd < 0) {
                  client[i].fd = connfd;       /* 找到client[]中空闲的位置，存放accept返回的connfd */
                    break;
                }
            if (i == OPEN_MAX)
                perr_exit("too many clients");

            client[i].events = POLLRDNORM;  /* 设置刚刚返回的connfd，监控读事件 */
            if (i > maxi)
                maxi = i;                               /* 更新client[]中最大元素下标 */

            if (--nready <= 0)
                continue;                               /* 没有更多就绪事件时,继续回到poll阻塞 */
        }

        for (i = 1; i <= maxi; i++) {        /* 检测client[] */
            if ( (sockfd = client[i].fd) < 0)
                continue;
            if (client[i].revents & (POLLRDNORM | POLLERR)) {
                if ( (n = Read(sockfd, buf, MAXLINE)) < 0) {
                    if (errno == ECONNRESET) {          /* 当收到 RST标志时 */
                        /* connection reset by client */
                        printf("client[%d] aborted connection\n", i);
                        Close(sockfd);
                        client[i].fd = -1;
                    } else
                        perr_exit("read error");
                } else if (n == 0) {
                    /* connection closed by client */
                    printf("client[%d] closed connection\n", i);
                    Close(sockfd);
                    client[i].fd = -1;
                } else {
                    for (j = 0; j < n; j++)
                        buf[j] = toupper(buf[j]);
                    Writen(sockfd, buf, n);
                }

                if (--nready <= 0)
                    break;                               /* no more readable descriptors */
            }
        }
    }

    return 0;
}
```

server

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 8000

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (fgets(buf, MAXLINE, stdin) != NULL) {
        Write(sockfd, buf, strlen(buf));
        n = Read(sockfd, buf, MAXLINE);
        if (n == 0)
            printf("the other side has been closed.\n");
        else
            Write(STDOUT_FILENO, buf, n);
    }

    Close(sockfd);
    return 0;
}
```

client

12.3.4 epoll

epoll是Linux下多路复用IO接口select/poll的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率，因为它会复用文件描述符集合来传递结果而不用迫使开发者每次等待事件之前都必须重新准备要被侦听的文件描述符集合，另一点原因就是获取事件的时候，它无须遍历整个被侦听的描述符集，只要遍历那些被内核IO事件异步唤醒而加入Ready队列的描述符集合就行了。

目前epell是linux大规模并发网络程序中的热门首选模型。

epoll除了提供select/poll那种IO事件的电平触发（Level Triggered）外，还提供了边沿触发（Edge Triggered），这就使得用户空间程序有可能缓存IO状态，减少epoll_wait/epoll_pwait的调用，提高应用程序效率。

一个进程打开大数目的socket描述符

```
cat  /proc/sys/fs/file-max
```

设置最大打开文件描述符限制

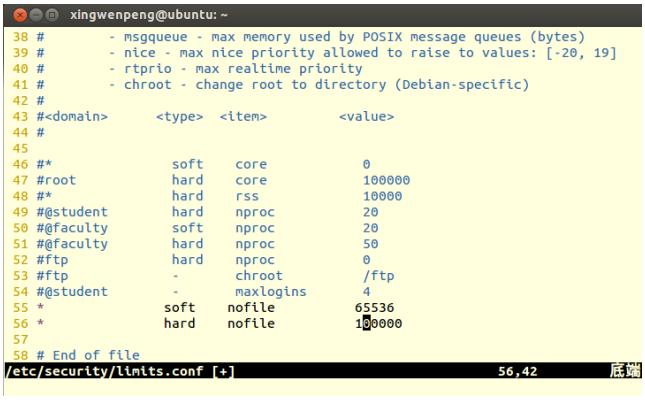

图 12.2: 最大打开文件个数设置

```
sudo vi /etc/security/limits.conf
写入以下配置,soft软限制, hard硬限制
*               soft    nofile          65536
*               hard    nofile          100000
```

epoll API

1.创建一个epoll句柄，参数size用来告诉内核监听的文件描述符个数，跟内存大小有关 #include

```
int epoll_create(int size)
    size：告诉内核监听的数目
```

2.控制某个epoll监控的文件描述符上的事件：注册、修改、删除。

```
#include <sys/epoll.h>
    int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
            epfd：为epoll_creat的句柄
            op：表示动作，用3个宏来表示：
        EPOLL_CTL_ADD(注册新的fd到epfd),
        EPOLL_CTL_MOD(修改已经注册的fd的监听事件),
        EPOLL_CTL_DEL(从epfd删除一个fd)；
    event：告诉内核需要监听的事件
        struct epoll_event {
          __uint32_t events;  /* Epoll events */
          epoll_data_t data;  /* User data variable */
        };
                    EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）
```

```
            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（这里应该表示有带外数据到来）
            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： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的
            EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里
```

3.等待所监控文件描述符上有事件的产生，类似于select()调用。

```
    #include <sys/epoll.h>
        int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)
        events：用来从内核得到事件的集合，
        maxevents：告之内核这个events有多大，这个maxevents的值不能大于创建epoll_create()时的size，
        timeout：是超时时间
            -1：阻塞
             0：立即返回，非阻塞
            >0：指定微秒
        返回值：成功返回有多少文件描述符就绪，时间到时返回0，出错返回-1
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/epoll.h>
#include <errno.h>
#include "wrap.h"


#define MAXLINE 80
#define SERV_PORT 8000
#define OPEN_MAX 1024

int main(int argc, char *argv[])
{
    int i, j, maxi, listenfd, connfd, sockfd;
    int        nready, efd, res;
    ssize_t        n;
    char buf[MAXLINE], str[INET_ADDRSTRLEN];
    socklen_t clilen;
    int        client[OPEN_MAX];
    struct sockaddr_in        cliaddr, servaddr;
    struct epoll_event tep, ep[OPEN_MAX];

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
```

```
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    Bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

    Listen(listenfd, 20);

    for (i = 0; i < OPEN_MAX; i++)
        client[i] = -1;
    maxi = -1;

    efd = epoll_create(OPEN_MAX);

    if (efd == -1)
        perr_exit("epoll_create");

    tep.events = EPOLLIN; tep.data.fd = listenfd;
    res = epoll_ctl(efd, EPOLL_CTL_ADD, listenfd, &tep);
    if (res == -1)
        perr_exit("epoll_ctl");

    for ( ; ; ) {
        nready = epoll_wait(efd, ep, OPEN_MAX, -1);     /* 阻塞监听 */
        if (nready == -1)
            perr_exit("epoll_wait");
        for (i = 0; i < nready; i++) {
            if (!(ep[i].events & EPOLLIN))
                continue;
            if (ep[i].data.fd == listenfd) {
                clilen = sizeof(cliaddr);
                connfd = Accept(listenfd, (struct sockaddr *)&cliaddr, &clilen);
                                                    printf("received   from   %s   at   PORT   %d
\n", inet_ntop(AF_INET, &cliaddr.sin_addr, str, sizeof(str)), ntohs(cliaddr.sin_port));

                for (j = 0; j < OPEN_MAX; j++)
                    if (client[j] < 0) {
                        client[j] = connfd;        /* save descriptor */
                        break;
                    }
                if (j == OPEN_MAX)
                    perr_exit("too many clients");

                if (j > maxi)
                    maxi = j;                                /* max index in client[] array */
                tep.events = EPOLLIN; tep.data.fd = connfd;
                res = epoll_ctl(efd, EPOLL_CTL_ADD, connfd, &tep);
                if (res == -1)
                    perr_exit("epoll_ctl");
            }
            else {
                sockfd = ep[i].data.fd;
                n = Read(sockfd, buf, MAXLINE);
                if (n == 0) {
                    for (j = 0; j <= maxi; j++) {
                        if (client[j] == sockfd) {
```

```
                        client[j] = -1;
                        break;
                    }
                }
                res = epoll_ctl(efd, EPOLL_CTL_DEL, sockfd, NULL);
                if (res == -1)
                    perr_exit("epoll_ctl");
                Close(sockfd);
                printf("client[%d] closed connection\n", j);
            }
            else {
                for (j = 0; j < n; j++)
                    buf[j] = toupper(buf[j]);
                Writen(sockfd, buf, n);
            }
        }
    }
}

    close(listenfd);
    close(efd);
    return 0;
}
```

server

```
/* client.c */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include "wrap.h"

#define MAXLINE 80
#define SERV_PORT 8000

int main(int argc, char *argv[])
{
    struct sockaddr_in servaddr;
    char buf[MAXLINE];
    int sockfd, n;

    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr);
    servaddr.sin_port = htons(SERV_PORT);

    Connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));

    while (fgets(buf, MAXLINE, stdin) != NULL) {
```

```
        Write(sockfd, buf, strlen(buf));
        n = Read(sockfd, buf, MAXLINE);
        if (n == 0)
            printf("the other side has been closed.\n");
        else
            Write(STDOUT_FILENO, buf, n);
    }

    Close(sockfd);
    return 0;
}
```

client

12.4 线程池并发服务器

1.预先创建阻塞于accept多线程，使用互斥锁上锁保护accept
2.预先创建多线程，由主线程调用accept

12.5 UDP局域网服务器

12.6 其它常用函数

12.6.1 名字与地址转换

过时，仅用于IPv4，线程不安全
gethostbyname
gethostbyaddr
getservbyname
getservbyport
趋势，可同时处理IPv4和IPv6，线程安全
getaddrinfo
getnameinfo

第 13 章

shell编程

第 14 章

正则表达式

第 15 章

错误处理机制

15.1 errno

vi /usr/include/asm-generic/errno-base.h

```
#define EPERM        1  /* Operation not permitted */
#define ENOENT       2  /* No such file or directory */
#define ESRCH        3  /* No such process */
#define EINTR        4  /* Interrupted system call */
#define EIO      5  /* I/O error */
#define ENXIO        6  /* No such device or address */
#define E2BIG        7  /* Argument list too long */
#define ENOEXEC      8  /* Exec format error */
#define EBADF        9  /* Bad file number */
#define ECHILD      10  /* No child processes */
#define EAGAIN      11  /* Try again */
#define ENOMEM      12  /* Out of memory */
#define EACCES      13  /* Permission denied */
#define EFAULT      14  /* Bad address */
#define ENOTBLK     15  /* Block device required */
#define EBUSY       16  /* Device or resource busy */
#define EEXIST      17  /* File exists */
#define EXDEV       18  /* Cross-device link */
#define ENODEV      19  /* No such device */
#define ENOTDIR     20  /* Not a directory */
#define EISDIR      21  /* Is a directory */
#define EINVAL      22  /* Invalid argument */
#define ENFILE      23  /* File table overflow */
#define EMFILE      24  /* Too many open files */
#define ENOTTY      25  /* Not a typewriter */
#define ETXTBSY     26  /* Text file busy */
#define EFBIG       27  /* File too large */
#define ENOSPC      28  /* No space left on device */
#define ESPIPE      29  /* Illegal seek */
#define EROFS       30  /* Read-only file system */
#define EMLINK      31  /* Too many links */
#define EPIPE       32  /* Broken pipe */
#define EDOM        33  /* Math argument out of domain of func */
#define ERANGE      34  /* Math result not representable */
```

15.2 perror

```
#include <stdio.h>

void perror(const char *s);

#include <errno.h>

const char *sys_errlist[];
int sys_nerr;
int errno;
```

15.3 strerror

```
#include <string.h>

char *strerror(int errnum);

int strerror_r(int errnum, char *buf, size_t buflen);
          /* XSI-compliant */

char *strerror_r(int errnum, char *buf, size_t buflen);
          /* GNU-specific */
```

第 16 章

syslog机制

第 17 章

命令行参数

第 18 章

时间函数

18.1　文件访问时间

```
#include <sys/types.h>
#include <utime.h>
int utime (const char *name, const struct utimebuf *t);
返回:若成功则为 0,若出错则为- 1
```

如果times是一个空指针,则存取时间和修改时间两者都设置为当前时间;

如果times是非空指针,则存取时间和修改时间被设置为 times所指向的结构中的值。此时,进程的有效用户ID必须等于该文件的所有者 ID,或者进程必须是一个超级用户进程。对文件只具有写许可权是不够的

此函数所使用的结构是:

```
struct utimbuf {
time_t actime;               /*access time*/
time_t modtime;              /*modification time*/
}
```

18.2　cpu使用时间

```
#include <sys/time.h>
#include <sys/resource.h>

int getrusage(int who, struct rusage *usage);

    RUSAGE_SELF
          Return resource usage statistics for the calling process,  which
          is the sum of resources used by all threads in the process.
```

```
RUSAGE_CHILDREN
       Return resource usage statistics for all children of the calling
       process that have terminated and been waited for.  These statis-
       tics  will include the resources used by grandchildren, and fur-
       ther removed descendants, if all of the intervening  descendants
       waited on their terminated children.

RUSAGE_THREAD (since Linux 2.6.26)
       Return resource usage statistics for the calling thread.
```

第 19 章

工具

19.1 网络工具

19.1.1 ifconfig

```
sudo ifconfig eth0 down/up
sudo ifconfig eth0 192.168.102.123
```

19.1.2 ping

19.1.3 netstat

```
-a (all)显示所有选项，默认不显示LISTEN相关
-t (tcp)仅显示tcp相关选项
-u (udp)仅显示udp相关选项
-n 拒绝显示别名，能显示数字的全部转化成数字。
-l 仅列出有在 Listen（监听）的服務状态

-p 显示建立相关链接的程序名
-r 显示路由信息，路由表
-e 显示扩展信息，例如uid等
-s 按各个协议进行统计
-c 每隔一个固定时间，执行该netstat命令。
```

提示：LISTEN和LISTENING的状态只有用-a或者-l才能看到

```
sudo netstat -anp | grep ftp
```

19.1.4 设置IP

* 以DHCP方式配置网卡

编辑文件/etc/network/interfaces：

```
sudo vi /etc/network/interfaces
```

并用下面的行来替换有关eth0的行：

```
# The primary network interface - use DHCP to find our address

auto eth0
iface eth0 inet dhcp
```

用下面的命令使网络设置生效：

```
sudo /etc/init.d/networking restart
```

也可以在命令行下直接输入下面的命令来获取地址

```
sudo dhclient eth0
```

* 为网卡配置静态IP地址

(1)编辑文件/etc/network/interfaces：

```
sudo vi /etc/network/interfaces
```

并用下面的行来替换有关eth0的行：

```
# The primary network interface
auto eth0
iface eth0 inet static
address 192.168.2.1
gateway 192.168.2.254
netmask 255.255.255.0
#network 192.168.2.0
#broadcast 192.168.2.255
```

(2)将上面的ip地址等信息换成你自己就可以了.用下面的命令使网络设置生效：

```
sudo /etc/init.d/networking restart
```

3.设置DNS

要访问DNS 服务器来进行查询,需要设置/etc/resolv.conf文件，

假设DNS服务器的IP地址是192.168.2.2，那么/etc/resolv.conf文件的内容应为：

```
nameserver 192.168.2.2
```

手动重启网络服务：

```
sudo /etc/init.d/networking restart
```

第 20 章

小项目实战

20.1 shell

用讲过的各种C函数实现一个简单的交互式Shell，要求：

1、给出提示符，让用户输入一行命令，识别程序名和参数并调用适当的exec函数执行程序，待执行完成后再次给出提示符。

2、识别和处理以下符号：

简单的标准输入输出重定向：仿照例 “wrapper”，先dup2然后exec。

管道（|）：Shell进程先调用pipe创建一对管道描述符，然后fork出两个子进程，一个子进程关闭读端，调用dup2把写端赋给标准输出，另一个子进程关闭写端，调用dup2把读端赋给标准输入，两个子进程分别调用exec执行程序，而Shell进程把管道的两端都关闭，调用wait等待两个子进程终止。

你的程序应该可以处理以下命令：

```
○ls△-l△-R○>○file1○
○cat○<○file1○|○wc△-c○>○file1○
○表示零个或多个空格，△表示一个或多个空格
```

项目3步走 1.实现加载普通命令 2.实现重定向的功能 3.实现管道 4.实现多重管道支持

20.2 多线程cp

20.3 哲学家就餐

20.4 数字多媒体广告机系统

20.5 高并发即时通信服务器

20.6 web服务器

第 21 章

大项目实战

21.1　研发中