附录C　习 题 答 案

第1章

1.1　利用ls(1)命令中的下面两个选择项：-i —— 显示文件或目录的i节点数目（关于i节点在4.14节中会详细讨论）；-d —— 如果参数是一目录，只列出其名字，而不是目录中的所有文件。

执行命令的结果为：

```
$ ls -ldi /etc/. /etc/..  -i要求打印i节点的数量
3077 drwxr-sr-x  7 bin  2048 Aug 5 20:12 /etc/./
2 drwxr-xr-x 13 root  512 Aug 5 20:11 /etc/../
$ls -ldi /. /..              .和..的i节点数均为2
2 drwxr-xr-x 13 root  512 Aug 5 20:11 /./
2 drwxr-xr-x 13 root  512 Aug 5 20:11 /../
```

1.2　UNIX是多任务系统，所以，在程序1-4运行的同时其他两个进程也在运行。

1.3　假如perror的*ptr*参数是一个指针，则perror就可以改变*ptr*所指串的内容。所以利用限定词const使得perror不能修改*ptr*所指的串。而strerror的参数是错误号，由于其是整数类型并且C传递的是参数值，因此strerror不能修改参数的值，也就没有必要使用const属性。（如果C中函数参数的处理不是很清楚，可参见Kernighan和Ritchie〔1998〕5.2节。）

1.4　调用fflush，fprintf和vprintf函数可修改errno的值。如果它的值变了但没有保存，则最终显示的错误信息是不正确的。

在过去开发的许多程序中，都可以发现不保存 errno的情况，典型的错误信息是“ Not a typewriter（打字机不存在）”。5.4节中标准I/O库根据标准I/O流是否指向终端设备而改变流的缓存器。istty（见11.9节）通常用来判断流是否指向终端设备，如果流不指向终端设备， errno可能置为ENOTTY，从而引起该错误。程序C-1显示了这一特性。

程序C-1　errno和printf的交互作用

```
#include    <stdio.h>

/*
 * The following prints errno=25 (ENOTTY) under 4.3BSD and SVR2,
 * when stdout is redirected to a file.
 * Under SVR4 and 4.3+BSD it works OK.
 */

int
main()
{
    int         fd;
    extern int  errno;

    if ( (fd = open("/no/such/file", 0)) < 0) {
        printf("open error: ");
        printf("errno = %d\n", errno);
    }
    exit(0);
}
```

执行上面的程序，结果为：

```
$ grep BSD /etc/motd
4.3 BSD UNIX #29: Thu Mar 29 11:14:13 MST 1990
$ a.out
open error: error = 2      工作正常，stdout是一个终端
$ a.out > temp.foo
$ cat temp.foo
open error: error = 25     错误
```

1.5 2038年。

1.6 大约248天。

第2章

2.1 下面是4.3+BSD中使用的技术。在<machine/ansi.h> 中，用大写字母定义可在多个头文件中出现的基本数据类型。例如：

```
#ifndef  _ANSI_H_
#define  _ANSI_H_

#define  _CLOCK_T_ unsigned long
#define  _SIZE_T_  unsigned int

...

#endif   /* _ANSI_H_ */
```

以下面的顺序可以在这6个头文件中分别定义size_t。

```
#ifdef  _SIZE_T_
typedef  _SIZE_T_ size_t;
#undef  _SIZE_T_
#endif
```

这样，实际上只执行一次typedef。

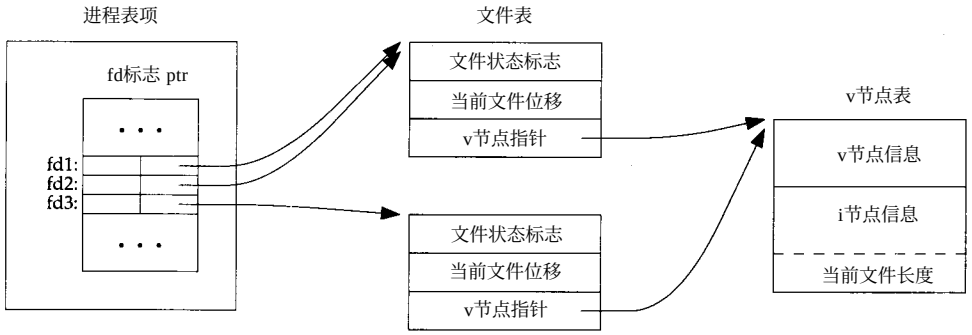
第3章

3.1 所有的磁盘I/O都要经过内核的块缓存器，唯一例外的是对原始磁盘设备的I/O，但是我们不考虑这种情况（Bach〔1986〕的第3章讲述了这种缓存器的操作）。既然read或write的数据都要被内核缓存，那么术语“无缓存装置的I/O”指的是在用户的进程中对这两个函数不会自动缓存，每次read或write就要进行一次系统调用。

3.3 每次调用open函数就分配一个文件表项，如果两次打开的是相同的文件，则两个文件表项指向相同的v节点。调用dup引用已存在的文件表项（此处指fd1的文件表项），见图C-1。当F_SETFD作用于fd1时，只影响fd1的文件描述符标志；F_SETFL作用于fd1时，则影响fd1及fd2的文件描述符标志。

3.4 如果fd是1，执行dup2(fd，1)后返回1，但是没有关闭描述符1（见3.12节）。调用3次dup2后，3个描述符指向相同的文件表项，所以不需要关闭描述符。

如果fd是3，调用3次dup2后，有4个描述符指向相同的文件表项，所以需要关闭描述符3。



图C-1 open和dup的结果

3.5 shell从左到右处理命令行,所以

```
a.out > outfile 2>&1
```

首先设置标准输出到outfile，然后执行dups将标准输出复制到描述符2（标准错误）上，其结果是将标准输出和标准错误设置为相同的文件，即描述符 1和2指向相同的文件表项。而对于命令行

```
a.out 2 >&1 >outfile
```

由于首先执行dups，所以描述符2成为终端（假设命令是交互执行的），标准输出重定向到outfile。结果是描述符1指向outfile的文件表项，描述符2指向终端的文件表项。

3.6 这种情况之下，仍然可以用lseek和read函数读文件中任意一处的内容。但是write函数在写数据之前会自动将文件位移量设置为文件尾，所以写文件时只能从文件尾开始，不能在任一位置。

第4章

4.1 stat函数总是顺一个符号连接向前，所以修改后的程序不会显示文件类型是“符号连接”。例如：/bin是/usr/bin的一个符号连接，但是stat函数的结果只显示/bin是一个目录，而不说明它是一个符号连接。若一个符号连接指向一不存在的文件，则 stat出错返回。

4.2 将下面的几行语句加入<ourhdr.h>

```
#if defined (S_IFLNK)  &&  !defined(S_ISLNK )
#define  S_ISLNK(mode)    (((mode) & S_IFMT ) == S_IFLNK)
#endif
```

这是一个我们编写的头文件如何屏蔽某些系统差别的实例。

4.3 关闭了该文件的所有存取许可权。

```
$ umask 777
$ data > temp.foo
$ ls -l temp.foo
---------- l stevens   29 Jan 14 06:39 temp.foo
```

4.4 下面的命令表示关闭用户读许可权的情况。

```
$ data > foo
$ chmod u-r foo     关闭用户读许可权
$ ls -l foo         验证文件的许可权
--w-rw-r-- l stevens   29 Jul 31 09:00 foo
$ cat foo           读文件
```

```
cat: foo: Permission denied
```

4.5 如果用open或creat创建已经存在的文件，则该文件的存取许可权不变。程序 4-3可以验证这点。

```
$ rm foo bar                          删除文件
$ data > foo                          创建文件
$ data > bar
$ chmod a-r foo bar                   关闭所有的读许可权
$ ls -l foo bar                       验证其许可权
--w--w----  l stevens        29 Jul 31 10:47 bar
--w--w----  l stevens        29 Jul 31 10:47 foo
$ a.out                               运行程序4-3
$ ls -l foo bar                       检查文件的许可权和大小
--w--w----  lstevens          0 Jul 31 10:47 bar
--w--w----  lstevens          0 Jul 31 10:47 foo
```

可以看出存取许可权没有改变，但是文件长度缩短了。

4.6 目录的长度从来不会是0，因为它总是包含 .和 ..两项。符号连接的长度指其路径名包含的字符数，由于路径名中至少有一个字符，所以长度也不为 0。

4.8 当创建新的core文件时，内核对其存取许可权有一个默认设置，在本例中是 rw-r--r--。这一默认值可能会可能不会被 umask的值修改。shell对创建的重定向的新文件也有一个默认的访问许可权，本例中为rw-rw-rw-。这个值总是被当前的umask修改，在本例中umask为02。

4.9 不能使用du的原因是它需要文件名，如：

```
du tempfile
```

或目录名，如：

```
du .
```

只有当unlink函数返回时才释放tempfile的目录项， du.命令没有计算仍然被tempfile占用的空间。本例中只能使用df命令察看文件系统中实际可用的自由空间。

4.10 如果被删除的链接不是该文件的最后一个链接，则该文件不会删除。此时，文件的状态改变时间被更新。如果是最后一个链接被删除，则该文件将被物理删除。这时再去更新文件的状态改变时间就没有意义，因为包含文件所有信息的 i节点将会随着文件的删除而被释放。

4.11 用opendir打开一个目录后，循环调用函数dopath。假设opendir使用一个文件描述符，并且只有处理完目录后调用 closedir才释放描述符，这就意味着每次打开目录就要降一级使用另外一个描述符。所以系统可打开的描述符数就限制了文件系统中树的深度。 SVR4中的ftw允许调用者指定使用的描述符数，这隐含着该实现可以关闭描述符并且重用它们。

4.13 chroot函数用于辅助因特网文件传输程序（FTP）中的安全性。系统中没有帐号的用户（也称为匿名FTP）放在一个单独的目录下，利用 chroot将此目录当作新的根目录就可以阻止用户访问此目录以外的文件。

chroot也用于在另一台机器上构造一文件系统层次结构的一个副本，然后修改此副本，但不更改原来的文件系统。这可用于测试新软件包的安装。 chroot只能由超级用户使用，一旦更改了一个进程的root，该进程及其后代进程就再也不能恢复至原先的 root。

4.14 首先调用stat函数取得文件的三个时间值，然后调用 utime设置期望的值。我们不希望在调用utime时改变的值就是stat中相应的值。

4.15 finger(1)对邮箱调用stat函数，最近一次的修改时间是上一次接收邮件的时间，最近存取时间是上一次读邮件的时间。

4.16 对cpio来说，既可以改变文件的访问时间（st_atime）和修改时间（st_mtime），也可以都不改变。cpio的-a选项可以在读文件后重新设置文件的存取时间，改变文件的存取时间。另一方面，-m将文件的修改时间和存取时间保存为归档时的值。

对tar来说，在抽取文件时，其默认方式是复原归档时的修改时间，但是-m选择项则将修改时间设置为抽取文件时的时间。无论tar在何种情况，文件的存取时间均是抽取文件时的时间。

由于不能修改状态改变时间（utime也只能改变访问时间和修改时间），所以没有将其保存在文档上。

4.17 read改变了文件存取时间，为了消除这一影响，有些版本的 file(1)调用utime恢复文件的存取时间，但是这样做会修改文件的状态改变时间。

4.18 内核对目录的深度没有内在的限制，但是如果路径名的长度超出了 PATH_MAX，则有许多命令会失败。程序 C-2创建了一个深度为 100的目录树，每一级目录名有 45个字符。利用getcwd可以得到第100级目录的绝对路径名（需要多次调用 realloc申请一个足够大的缓存）。

程序C-2 创建深目录树

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

#define DEPTH   100         /* directory depth */
#define MYHOME  "/home/stevens"
#define NAME    "alonglonglonglonglonglonglonglonglongname"

int
main(void)
{
    int     i, size;
    char    *path;

    if (chdir(MYHOME) < 0)
        err_sys("chdir error");

    for (i = 0; i < DEPTH; i++) {
        if (mkdir(NAME, DIR_MODE) < 0)
            err_sys("mkdir failed, i = %d", i);
        if (chdir(NAME) < 0)
            err_sys("chdir failed, i = %d", i);
    }
    if (creat("afile", FILE_MODE) < 0)
        err_sys("creat error");

    /*
     * The deep directory is created, with a file at the leaf.
     * Now let's try and obtain its pathname.
     */

    path = path_alloc(&size);
    for ( ; ; ) {
        if (getcwd(path, size) != NULL)
            break;
        else {
            err_ret("getcwd failed, size = %d", size);
            size += 100;
            if ( (path = realloc(path, size)) == NULL)
```

```
                err_sys("realloc error");
            }
        }
        printf("length = %d\n%s\n", strlen(path), path);

        exit(0);
    }
```

运行后得到：

```
$ a.out
getcwd failed, size = 1025: Result too large
getcwd failed, size = 1125: Result too large
...                          33行
getcwd failed, size = 4525: Result too large
length = 4613
```

显示4613字节的路径名

但是由于文件名太长了，不能用tar或cpio对该目录建立档案文件，而且也不能用rm -r命令删除该目录。（怎样才能删除该目录树？）

4.19 /dev目录关闭了一般用户的写许可权，所以用户不能删除目录中的文件，即unlink失败。

第5章

5.2 fgets函数读入数据，直到行结束或缓存满（当然会留出一个字节存放终止字符）。同样，fputs只负责将缓存的内容输出，而并不考虑缓存中是否包含换行符。所以，如果将MAXLINE设得很小，这两个函数仍然会正常工作，只不过被执行的次数要比MAXLINE值较大的时候多。

如果这些函数删除或添加换行符（如gets和puts），则必需保证缓存足够大。

5.3 当printf没有输出任何字符时，如：printf(""），则返回0。

5.4 这是一个比较常见的错误。getc以及getchar的返回值是整型，而不是字符型。由于EOF经常定义为－1，那么如果系统使用的是有符号的字符类型，程序还可以正常工作。但如果使用的是无符号字符类型，那么返回的EOF被保存到字符c后将不再是－1，所以，程序会进入死循环。

5.5 5个字符长的前缀、4个字符长的进程内唯一标识再加5个字符长的系统内唯一标识（进程ID）刚好组成14位的UNIX传统文件长度限制。

5.6 使用方法为：先调用fflush后调用fsync，fsync所使用的参数由fileno函数获得。如果不调用fflush，所有的数据仍然在内存缓存中，此时调用fsync将没有任何效果。

5.7 当程序交互运行时，标准输入和输出设备均为行缓存方式。每次调用fgets时标准输出设备将自动刷清。

第6章

6.1 在SVR4系统中，用户手册中讲述了存取阴影口令文件的函数。我们不能使用6.2节所述函数返回的pw_passwd变量来比较加密口令。正确的方法是使用阴影口令文件中对应用户的加密口令来进行比较。

在4.3+BSD系统中，口令文件的阴影是自动建立的。仅当调用者的用户ID为0时，getpwnam或getpwuid函数返回的passed结构中的pw_passwd字段才包含有加密口令。

6.2 在SVR4系统中，程序C-3将输出加密口令。当然，除非有超级用户许可权，否则调用getspnam将返回EACCES错误。

程序C-3 在SVR4系统中输出加密口令

```
#include    <sys/types.h>
#include    <shadow.h>
#include    "ourhdr.h"

int
main(void)      /* SVR4 version */
{
    struct spwd *ptr;

    if ( (ptr = getspnam("stevens")) == NULL)
        err_sys("getspnam error");

    printf("sp_pwdp = %s\n",
            ptr->sp_pwdp == NULL || ptr->sp_pwdp[0] == 0 ?
            "(null)" : ptr->sp_pwdp);
    exit(0);
}
```

在4.3+BSD系统中，具有超级用户许可权时，程序C-4将输出加密口令。否则pw_passed的返回值为星号(*)。

程序C-4 在4.3+ BSD系统中输出加密口令

```
#include    <sys/types.h>
#include    <pwd.h>
#include    "ourhdr.h"

int
main(void)      /* 4.3+BSD version */
{
    struct passwd   *ptr;

    if ( (ptr = getpwnam("stevens")) == NULL)
        err_sys("getpwnam error");

    printf("pw_passwd = %s\n",
            ptr->pw_passwd == NULL || ptr->pw_passwd[0] == 0 ?
            "(null)" : ptr->pw_passwd);
    exit(0);
}
```

6.4 程序C-5以date格式输出日期。

程序C-5 以date(1)的格式输出日期和时间

```
#include    <time.h>
#include    "ourhdr.h"

int
main(void)
{
    time_t      caltime;
    struct tm   *tm;
    char        line[MAXLINE];

    if ( (caltime = time(NULL)) == -1)
        err_sys("time error");
    if ( (tm = localtime(&caltime)) == NULL)
        err sys("localtime error");
```

```
    if (strftime(line, MAXLINE, "%a %b %d %X %Z %Y\n", tm) == 0)
        err_sys("strftime error");
    fputs(line, stdout);

    exit(0);
}
```

程序C-5的运行结果如下：

```
$ echo $TZ                          作者使用的默认值
MST7
$ a.out
Wed Jan 15 06:48:57 MST 1992
$ TZ=EST5EDT a.out                  美国东海岸
Wed Jan 15 08:49:06 EST 1992
$ TZ=JST-9 a.out                    日本
Wed Jan 15 22:49:12 JST 1992
```

第7章

7.1 原因在于printf的返回值（输出的字符数）变成了main函数的返回码。当然，并不是所有的系统都会出现该情况。

7.2 当程序处于交互运行方式时，标准输出设备通常处于行缓存方式，所以当输出换行符时，上次的结果才被真正输出。如果标准输出设备被定向到一个文件而处于完全缓存方式，则当标准I/O清理操作执行时，结果才真正被输出。

7.3 由于agrc和argv不像environ一样保存在全局变量中，所以在大多数 UNIX系统中没有其他办法。

7.4 当C程序复引用一个空指针出错时，执行该程序的进程将终止，于是可以利用这种方法终止进程。

7.5 定义如下：

```
typedef void Exitfunc(void) ;

int atexit(Exitfunc *func) ;
```

7.6 calloc将分配的内存空间初始化为0。但是ANSI C并不保证0值与浮点0或空指针的值相同。

7.7 只有通过exec函数执行一个程序时，才会分配堆和堆栈。

7.8 可执行文件包含了用于调试core文件的符号表信息，用strip(1)可以删除这些信息，对两个a.out文件执行这条命令，它们的大小减为98 304和16 384。

7.9 没有使用共享库时，可执行文件的大部分都被标准I/O库所占用。

7.10 这段代码不正确。因为在if语句中定义了自动变量val，所以当if中的复合语句结束时，该变量就不存在了，但是在if语句之外又用指针引用已经不存在的自动变量val。

第8章

8.1 用下面几行代替程序8-2中调用printf的语句：

```
i = printf("pid = %d, glob = %d, var = %d\n",
                                getpid( ), glob, var);
sprintf (buf,"%d\n", i);
```

```
write (STDOUT_FILENO, buf, strlen(buf));
```

注意要定义变量i和buf。

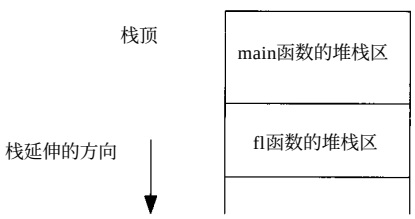
这里假设子进程调用 exit时只关闭标准 I/O流，并不关闭与标准输出相关的文件描述符STDOUT_FILENO。有些版本的标准I/O库会关闭与标准输出相关的文件描述符从而引起写失败，这种情况就调用dup将标准输出复制到另一个描述符， write则使用新复制的文件描述符。

8.2 可以通过程序C-6来说明这个问题。

程序C-6 错误使用vfork的例子

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

static void f1(void), f2(void);

int
main(void)
{
    f1();
    f2();
    _exit(0);
}

static void
f1(void)
{
    pid_t   pid;

    if ( (pid = vfork()) < 0)
        err_sys("vfork error");
    /* child and parent both return */
}

static void
f2(void)
{
    char    buf[1000];      /* automatic variables */
    int     i;

    for (i = 0; i < sizeof(buf); i++)
        buf[i] = 0;
}
```

当函数f1调用vfork时，父进程的堆栈指针就指向f1的栈帧，见图C-2。vfork使得子进程先执行然后从f1返回，接着子进程调用f2并且覆盖了f1的堆栈区间，在f2中子进程将自动变量buf的值置为0，即将堆栈中的1000个字节的值都置为0。从f2返回后父进程继续执行调用_exit，这时堆栈中main以下的内容已经被f2修改了，但是父进程仍然以为调用了vfork后从f1返回。返回信息虽然保存在堆栈中，但是可能已经被子进程修改了。对这个例子，父进程继续执行的结果要依赖于实际的UNIX系统。（如：返回信息保存在堆栈的具体位置，修改动态变量时覆盖了哪些信息等等。）通常的结果是一个core文件。



图C-2 调用vfork时的栈帧

8.3 在程序8-7中我们先让父进程输出，但是当父进程输出完毕子进程要输出时，要让父进程终止。是父进程先终止还是子进程先执行输出要依赖

于内核对两个进程的调度。shell在父进程终止后会开始执行其他程序，这样也许仍会影响子进程。要避免这种情况就是在子进程完成输出后才终止父进程。用下面的语句替换程序中 fork后面的代码。由于只有终止父进程才能开始下一个程序，所以不会出现上面的情况。

```
else if (pid == 0) {
    WAIT_PARENT();          /* parent goes first */
    charatatime("output from child\n");
    TELL_PARENT(getppid()); /* tell parent we're done */
} else {
    charatatime("output from parent\n");
    TELL_CHILD(pid);        /* tell child we're done */
    WAIT_CHILD();           /* wait for child to finish */
}
```

8.4 对argv[2]打印的是相同的值（/home/stevens/bin/testinterp）。原因是execlp在结束时调用了execve，并且与直接调用execl的路径名相同。

8.5 不提供返回保存的设置-用户-ID的函数，我们必须在进程开始时保存有效的用户ID。

8.6 程序C-7创建了一个僵死进程。

程序C-7 创建一个僵死进程并用ps查看其状态

```
#include    "ourhdr.h"

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)      /* child */
        exit(0);

    /* parent */
    sleep(4);

    system("ps");

    exit(0);
}
```

执行程序结果如下（ps（1）用Z表示僵死进程）：

```
$ a.out
  PID TT STATTIME COMMAND
 5940 p3 S   0:00 a.out
 5941 p3 Z   0:00 <defunct>      僵死进程
 5942 p3 S   0:00 sh -c ps
 5943 p3 R   0:00 ps
```

第9章

9.1 因为init是login shell的父进程，当登录shell终止时它收到SIGCHLD信号量，所以init进程知道什么时候终端用户注销。

网络登录没有包含init，相应的注销项是由一个处理登录并监测注销的进程写的（本例中为telnetd）。

第10章

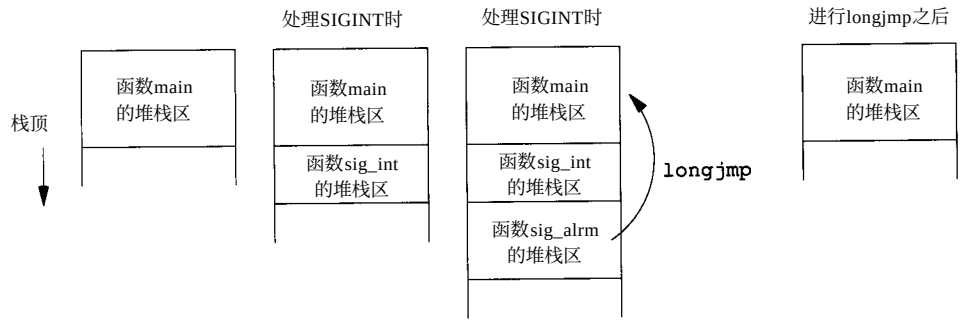
10.1 当程序第一次接收到发送给它的信号量时就终止了。因为一捕捉到信号量 pause函数就返回。

10.2 程序C-8实现了raise函数。

程序C-8 raise函数的实现

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

int
raise(int signo)
{
    return( kill(getpid(), signo) );
}
```

10.3 见图C-3。



图C-3 longjmp前后的堆栈状态

从sig_alrm通过longjmp返回main，有效地避免了继续执行sig_int。

10.4 如果进程在调用alarm和setjmp之间被内核阻塞了，alarm时间走完之后就调用信号量处理程序，然后调用longjmp。但是由于没有调用setjmp，所以没有设置env_alrm缓存区。如果longjmp的跳转缓存没有被setjmp初始化，则说明没有定义longjmp的操作。

10.5 参见Don Libes的“Implementing Software Timers”（*C Users Journal*, Vol. 8, no. 11, Nov. 1990）中的例子。

10.7 如果仅仅调用_exit，则进程终止状态就不能表示该进程是由于SIGABRT信号量而终止的。

10.8 如果信号量是由其他用户的进程发出的，进程必须设置用户的 ID为根或者是接收进程的所有者，否则kill不能执行。所以实际的用户ID为信号量的接收者提供了更多的信息。

10.10 对于本书中所用的系统，大约每 60~90分钟增加一秒，这个误差是因为每次调用sleep都要调度一次将来的时间事件，但是由于CPU调度，有时我们并没有在事件发生时被叫醒。另外一个原因是开始运行进程和调用sleep都需要一定的时间。

BSD中的cron每分钟都要取当前时间，它首先设置一个休眠周期，然后在下一分钟开始时唤醒。大多数调用是sleep(60)，偶尔有一个sleep(59)用于在下一分钟同步。但是若在进程中花费了许多时间执行命令或者系统的负载重调度慢，这时休眠值可能远小于 60。

10.11 在SVR4中，从来没有调用过 SIGXFSZ的信号量处理程序，一旦文件的大小达到1024时，write就返回24。

在4.3+BSD中，文件大小达到1500字节时调用该信号处理程序，write返回－1并且errno设置为EFBIG。

SunOS4.1.2的情况与SVR4类似，但是调用了该信号量处理程序。

系统V在文件大小达到软资源限制时无错返回一个较小的数，而 BSD判断文件超出限制时错误返回，没有写任何数据。

10.12 结果依赖于标准I/O库的实现 —— fwrite如何处理一个被中断的写。

第11章

11.1 注意由于终端是非正规模式，所以要用换行符而不是回车终止 reset命令。

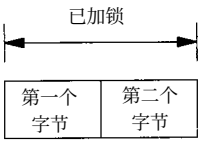
11.2 它为128个字符建了一张表，根据用户的要求设置奇偶校验位。然后使用 8位I/O处理奇偶位的产生。

11.3 在SVR4中运行stty -a，并且将标准输入重定向到运行vi的终端，结果显示vi设置MIN为1，TIME为1。reads等待至少敲入一个字符，但是该字符输入后，只对后继的字符等待十分之一秒即返回。

11.4 在SVR4中使用扩展的通用终端接口。参见 AT&T〔1991〕手册中的termiox(7)。在4.3+BSD中使用c_cflag字段的CCTS_OFLOW和CRTS_IFLOW标志，参见表11-1。

第12章

12.1 程序运行正常，不会发生 ENOLCK的错误。第一次循环调用 writew_lock、write和un_lock。调用un_lock后只保留了第一个字节的锁，第二次循环时，调用 writew_lock使得新锁与第一个字节的锁合并，图C-4是第二次循环的结果。



图C-4 第二次循环后锁的状态

每循环一次就扩展一个字节的锁，内核将这些锁合并后就只保持了一个锁，因此符合锁结构的定义。

12.2 在SVR4和4.3+BSD中,fd_set是只包含一个成员的结构，该成员为一个长整型数组。数组中每一位对应于一个描述符。四个 FD_宏通过开关或测试指定的位来操纵这个数组。将之定义为一个包含数组的结构而不仅仅是一个数组的原因是：通过 C语言的赋值语句，可以使fd_set类型变量相互赋值。

12.3 在SVR4和4.3+BSD中允许用户在头文件<sys/types.h> 前定义常数FD_SETSIZE。例如下面的代码可以使fd_set数据类型包含2048个描述符。

```
#define FD_SETSIZE 2048
#include    <sys/types.h>
```

12.4 下面列出了功能类似的函数。

FD_ZERO	sigemptyset
FD_SET	sigaddset
FD_CLR	sigdelset
FD_ISSET	sigismember

没有与sigfillset对应的FD_xxx函数。对信号量来说，指向信号量集合的指针是第一个参数，信号量数是第二个参数；对于描述符来说，描述符数是第一个参数，指向描述符集合的指针是第二个参数。

12.5 最多五种信息：数据，数据长度，控制信息，控制信息的长度和标志。

12.6 利用select实现的程序见C-9，利用poll实现的程序见C-10。

程序C-9 用select实现sleep_us函数

```
#include    <sys/types.h>
#include    <sys/time.h>
#include    <stddef.h>
#include    "ourhdr.h"

void
sleep_us(unsigned int nusecs)
{
    struct timeval  tval;

    tval.tv_sec = nusecs / 1000000;
    tval.tv_usec = nusecs % 1000000;
    select(0, NULL, NULL, NULL, &tval);
}
```

程序C-10 用poll实现sleep_us函数

```
#include    <sys/types.h>
#include    <poll.h>
#include    <stropts.h>
#include    "ourhdr.h"

void
sleep_us(unsigned int nusecs)
{
    struct pollfd   dummy;
    int             timeout;

    if ( (timeout = nusecs / 1000) <= 0)
        timeout = 1;
    poll(&dummy, 0, timeout);
}
```

BSD中的usleep(3)使用setitimer设置间隔计时器，并且执行8个系统调用。它可以正确地和调用进程设置的其他计时器交互作用，而且即使捕捉到信号量也不会被中断。

12.7 不行。我们可以使TELL_WAIT创建一个临时文件，其中一个字节用作为父进程的锁，另一个字节用作为子进程的锁。WAIT_CHILD使得父进程等待子进程的锁，TELL_PARENT使得子进程释放子进程的锁。但是问题在于调用 fork后，子进程释放了所有的锁导致子进程不能具有任何它自己的锁而开始执行。

12.8 用select的方法见程序C-11，使用poll的情况类似。

程序C-11 用select计算管道的性能

```
#include    <sys/types.h>
#include    <sys/time.h>
#include    "ourhdr.h"

int
```

```
main(void)
{
    int             i, n, fd[2];
    fd_set          writeset;
    struct timeval  tv;

    if (pipe(fd) < 0)
        err_sys("pipe error");
    FD_ZERO(&writeset);

    for (n = 0; ; n++) { /* write 1 byte at a time until pipe is full */
        FD_SET(fd[1], &writeset);
        tv.tv_sec = tv.tv_usec = 0;     /* don't wait at all */
        if ( (i = select(fd[1]+1, NULL, &writeset, NULL, &tv)) < 0)
            err_sys("select error");
        else if (i == 0)
            break;
        if (write(fd[1], "a", 1) != 1)
            err_sys("write error");
    }
    printf("pipe capacity = %d\n", n);
    exit(0);
}
```

在SVR4和SunOS 4.1.1中使用select和poll计算出的结果等于表2-6的值。在4.3+BSD中使用select计算的结果为3073。

12.9 在SVR4、4.3+BSD和SunOS 4.1.2中，程序12-14确实修改了输入文件的最近一次存取时间。

第13章

13.1 如果进程调用chroot就不能打开/dev/log，解决的办法是在chroot之前调用选择项为LOG_NDELAY的openlog。这样即使调用了chroot之后，仍然可以打开特定的设备文件（UNIX与数据报套接口）并生成一个有效的描述符。

13.3 程序C-12是一种解决方案。

程序C-12 调用daemon_init获得注册名

```
#include    "ourhdr.h"

int
main(void)
{
    char    *ptr, buff[MAXLINE];

    daemon_init();

    close(0);
    close(1);
    close(2);

    ptr = getlogin();
    sprintf(buff, "login name: %s\n",
                    (ptr == NULL) ? "(empty)" : ptr);
    write(3, buff, strlen(buff));
    exit(0);
}
```

结果依赖于不同的系统实现和是否关闭文件描述符 1、2和3。关闭描述符影响结果的原因是：当程序开始执行时与控制终端连接，调用 deamon_init后关闭3个描述符就意味着getlogin没

有控制终端，所以不能在utmp文件中看到登录项。

但是在4.3+BSD中，登录名是由进程表维护的，并且可以通过 fork复制。也就是说除非其父进程没有登录名（如系统自引导时调用init），否则进程总能获得其登录名。

第14章

14.1 如果写管道端总是不关闭，则读者就决不会看到文件的结束符。页面调度程序就会一直阻塞在读标准输入。

14.2 父进程向管道写完最后一行以后就终止了，然后读者读到管道的结尾时自动关闭管道。但是由于子进程（页面调度程序）要等待输出的页，所以父进程可能比子进程领先一个管道缓存器。如果在一个交互式命令行shell上运行，当父进程终止时shell会改变终端的模式并提示用户。由于大部分页面调度程序在等待处理下一个页面时将终端设置为非正规模式，所以终止父进程就会影响页面调度程序。

14.3 因为执行了shell，所以popen返回一个文件指针。但是shell不能执行不存在的命令，因此在标准错误上显示下面信息后终止，其退出状态为1，调用pclose就将该退出状态返回。

```
sh: a.out: not found
```

14.4 当父进程终止时，用shell看它的终止状态。对于Bourne shell和KornShell所用的命令是

```
echo $?
```

打印的结果是128加信号量数。

14.5 首先加入下面的定义，

```
FILE  *fpin, *fpout;
```

然后用fdopen关联管道描述符和标准I/O流，并将流设置为行缓存的，在 while循环从标准输入读之前作下面的工作。

```
if ( (fpin = fdopen(fd2[0], "r")) == NULL)
    err_sys("fdopen error");
if ( (fpout = fdopen(fd1[1], "w")) == NULL)
    err_sys("fdopen error");
if (setvbuf(fpin, NULL, _IOLBF, 0) < 0)
    err_sys("setvbuf error");
if (setvbuf(fpout, NULL, _IOLBF, 0) < 0)
    err_sys("setvbuf error");
```

while中的read和write用下面的语句代替：

```
if (fputs(line, fpout) == EOF)
    err_sys("fputs error to pipe");
if (fgets(line, MAXLINE, fpin) == NULL) {
    err_msg("child closed pipe");
    break;
}
```

14.6 虽然system函数调用了wait，但是终止的第一个子进程是由popen产生的，所以它将再次调用wait并一直阻塞到sleep完成，然后system返回。当pclose调用wait时，由于没有子进程可等待所以返回出错，导致pclose也返回出错。

14.7 select表明描述符是可读的。调用read读完所有的数据后返回0就表明到达了文件尾端。对于poll（假设管道是一个流设备）来说，若返回POLLHUP也许仍有数据可以读。但是一旦读完了所有的数据 read就返回0，即表明到达了文件尾端。 poll读完了所有的数据后并不返回POLLIN。

对于被读者关闭的指向管道的输出描述符来说， select表明描述符是可写的，调用 write时产生SIGPIPE信号量，如果忽略该信号量或从信号量处理程序中返回时 write就返回EPIPE错误。而对于poll，如果管道是一个流设备， poll就对该描述符返回POLLHUP。

14.8 子进程向标准出错写的内容同样也在父进程的标准出错中出错。只要在 *cmdstring*中包含重定向2 > &1命令，就可以将标准出错发送给父进程。

14.9 popen生成一个子进程，子进程通过exec执行Bourne shell。然后shell再调用fork，最后由shell的子进程执行命令串。当*cmdstring*终止时shell恰好在等待该事件，然后shell退出，而这一事件又是pclose中waitpid所等待的。

14.10 解决的办法是打开FIFO两次，一次读一次写。我们绝不会使用为写而打开的描述符，但是使该描述符打开就可在客户数从 1变为0时，阻止产生文件终止。打开 FIFO两次需要注意下列操作方式：第一次以非阻塞、只读方式 open，第二次以阻塞、只写方式 open。(如果用非阻塞、只写方式打开将返回错误。）然后关闭读描述符的非阻塞属性。参见程序C-13。

程序C-13 以非阻塞方式打开FIFO进行读写操作

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

#define FIFO    "temp.fifo"

int
main(void)
{
    int     fdread, fdwrite;

    unlink(FIFO);
    if (mkfifo(FIFO, FILE_MODE) < 0)
        err_sys("mkfifo error");

    if ( (fdread = open(FIFO, O_RDONLY | O_NONBLOCK)) < 0)
        err_sys("open error for reading");
    if ( (fdwrite = open(FIFO, O_WRONLY)) < 0)
        err_sys("open error for writing");

    clr_fl(fdread, O_NONBLOCK);

    exit(0);
}
```

14.11 随意读取现行队列中的消息会干扰客户机 -服务器协议，导致丢失客户机请求或者服务器的响应。由于队列允许所有的用户读，所以进程只要知道队列的标识符就可以读队列。

14.13 由于服务器和客户机都可能将段连接到不同的地址，所以在共享存储段中不存放实际物理地址。相反，当在共享存储段中建立链表时，指针的值设置为共享存储段内的位移。位移量为所指目标的实际地址减去共享存储段的起始地址。

14.14 表C-1显示了相关的事件。

表C-1 程序14-12中父子进程间的切换过程

父进程中i的值	子进程中i的值	共享变量的值	update的返回值	备 注
		0		由mmap初始化
	1			子进程先运行，然后阻塞

(续)

父进程中i的值	子进程中i的值	共享变量的值	update的返回值	备　注
0				父进程运行
		1		
			0	父进程阻塞
		2		子进程恢复
			1	
	3			子进程阻塞
2				父进程恢复
		3		
			2	父进程阻塞
		4		
			3	
	5			子进程阻塞
4				父进程恢复

第15章

15.3　说明（declaration）指定了标识符集合的属性（例如数据类型），如果说明的同时分配了存储单元就是定义（definition）。

在头文件open.h中用extern说明了三个全局变量，这时并没有为它们分配存储单元，在文件main.c中定义了三个全局变量，有时会在定义时就初始化全局变量，但通常使用C的默认值。

15.5　select和poll都返回就绪的描述符个数。当将这些就绪描述符都处理完后，操作client数组的循环就可以结束。

第16章

16.1　_db_dodelete中保守的加锁操作是为了避免和db_nextrec发生竞态条件。如果没有使用写锁保护_db_writedat调用，则有可能在_db_nextrec读某个记录时擦去该记录：db_nextrec首先读入一个索引记录，发现该记录非空，则接着读入记录内容，但是在它调用 _db_readidx后_db_readdat前，该记录却给_db_dodelete删除了。

16.2　假定db_nextrec调用了_db_readidx，它将记录的关键字读入索引缓存进行处理。但该处理过程被内核调度进程打断，另一个执行的进程刚好调用db_delete删除了这一条记录，使得索引文件和记录文件中对应的内容都被清空。当第一个进程恢复执行并调用 _db_readdat（在db_nextrec函数体中）时，返回的是空记录。所以db_nextrec中的读锁使得读入记录索引的过程和读入记录内容的过程是一个原子操作（至少对其他操作同一数据库的并发进程中的写操作而言）。

16.3　强制锁对其他的读者和写者产生了影响 —— 其他的读和写操作都被阻塞，直到_db_writeidx和_db_writedat设置的锁被解除。

第17章

17.1　psif必须读取文件的前两个字节并且与 %!进行比较。如果文件是可以随机定位的，则可以rewind文件，并调用lprps或textps。如果文件不可随机定位，则只能将该两个字节重新放回标准输入设备。此时一个可行的办法是：建立一个管道并 fork一个子进程。然后父进程将其标准输入设备设置为管道，并执行textps或lprps。然后子进程将它读到的两个字节写入管道，再接着将文件的其他部分输出。

第18章

18.2 通常getopt只用来处理单个参数列表。在 getopt函数的初始化数据段，全局变量optind被初始化为1。但是在我们的服务器中，调用 getopt来处理多个参数列表 —— 每个客户机一个，所以必须在为每个客户机首次调用getopt前均重新初始化optind。

18.3 我们使用了Client结构维护Systems文件的位移量。如果在保存了位移量后、再次使用之前修改了该文件，则有可能上次保存的位移量不再指向以前指向的行。虽然服务器可以检测到文件是否被修改了（如何检测？），但是我们无法再将位移量恢复到原来的正确位置。当文件修改后,我们唯一的办法是不让有关用户再登录进来。

18.4 只有当client_add被调用时，才能通过 realloc移动client数组。因为client_add是在select后，而不是在使用cliptr的循环中。

18.5 发送到远端系统的不同命令可能会被混淆起来。可以在 take_put_args中加入一些检查功能实现命令的区分。

18.6 一个常用的方法是：要求用户在修改任何文件后通知服务器，以使服务器可以重新读取文件。SIGHUP命令就是经常用来完成这项任务的。

18.9 可以在远端执行stty命令，然后分析其输出结果。但是考虑到不同UNIX平台的stty命令输出结果差别很大，这种方式实现起来较为困难。

第19章

19.1 telnetd和rlogind两个服务器均以超级用户的许可权运行,所以它们都可以成功地调用chown和chmod。

19.3 执行：

```
pty -n stty -a
```

以避免伪终端从设备的termios 结构和winsize 结构初始化。

19.5 很不幸，fcntl 的F_SETFL 命令不允许改变读-写状态。

19.6 有三个进程组：(1) 登录shell，(2) pty 父进程和子进程，(3) cat进程。前两个进程组与登录shell组成了一个对话期，其中，登录shell为对话期首进程。第二个对话期仅包含cat进程。第一个进程组（登录shell）是后台进程组，其他两个进程组是前台进程组。

19.7 当接受到文件终止符时，首先是cat终止，然后是伪终端从设备及伪终端主设备。接着，正从伪终端主设备读取的pty父进程产生一个文件终止符，该父进程将 SIGTERM信号发送给子进程，子进程终止（子进程不捕捉该信号）。最后，父进程调用main 函数尾端的exit(0)。

程序8-17中相关的输出为：

```
cat     e =     270, chars =     274, stat =    0:
pty     e =     262, chars =      40, stat =   15: F    X
pty     e =     288, chars =     188, stat =    0:
```

19.8 这可通过使用shell的echo和date(1)命令实现：

```
#!/bin/sh
(echo "Script started on " `date`;
 pty "${SHELL:-/bin/sh}";
 echo "Script done on " `date`) | tee typescript
```

19.9 伪终端上的行规程能够回应，故pty可以读取其标准输入，并写向伪终端主设备。尽管程序（ttyname）从不读取输入，该回应也可通过伪终端上的行规程模块实现。