第三章 UNIX 文件系统

1、文件的分类

在 UNIX 中，文件被用来实现统一的设备接口。除了把磁盘数据组织成文件系统之外，UNIX 还把其他外部设备（如 CD-ROM、显示器、键盘、打印机、鼠标和网络接口等）也映射成为一个文件。这种统一接口的 UNIX 设计哲学，既方便了用户（用户只需像访问磁盘文件一样去访问各种设备即可），又成就了 UNIX 系统（系统内核简洁、安全、易移植、易扩展）。

根据文件所映射的具体设备，一般把 UNIX 文件分为三类：普通文件指磁盘上的数据文件；块文件表示特性与磁盘类似的设备，这种设备允许以块或组块的形式传送信息，并具有从设备的任何地方检索块的能力，称为块设备；字符文件表示特性与终端类似的设备，这种设备允许以字节为单位传送信息，并且只能顺序操作。

对所有类型的文件，POSIX 主要提供了 5 个系统调用来访问：open、close、read、write、ioctl，这些函数通过调用被称为设备驱动程序的操作系统模块来完成对具体设备的操作。

同样的，ANSI C 的标准库函数中也提供了对文件的访问函数，它们主要包括 fopen、fclose、fscanf、fprintf、fread、fwrite 等。

上述二者的区别还是有一比的，请随我来。

2、系统调用

2.1 打开和关闭文件

open 系统调用将一个整数值与一个文件或物理设备关联起来，这个整数值被称为文件描述符。如果成功，open 返回一个非负整数来表示打开的文件描述符。如果不成功，open 返回-1 并设置 errno，具体 errno 值请查阅联机帮助手册。

文件描述符表示了打开的文件或设备，可以将文件描述符想象成进程文件描述符表的索引，文件描述符表在进程的用户区，提供了对相关文件或设备的系统信息的访问。每个进程都自动伴随三个打开的文件，它们的文件描述符分别为 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO，其值分别为 0、1 和 2，分别表示标准输入、标准输出和标准错误，缺省情况下分别代表键盘和显示器。

```
SYNOPSIS
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
int open(const char *path, int oflags);
int open(const char *path, int oflags, mode_t mode);   POSIX
```

参数 path 指向要打开的文件或设备的路径名。

参数 oflag 为打开的文件指定了状态标志符和访问模式，该参数通过访问模式标识符和附加标志符按位或（|）运算来构建。访问模式标识符的 POSIX 值为 O_RDONLY、O_WRONLY、O_RDWR，必须指定其中一个来说明只读、只写或读写。附加标志符包括 O_APPEND、O_CREAT、O_EXECL、O_NOCTTY、O_NONBLACK 和 O_TRUNC。其中 O_APPEND 在写操作之前使文件偏移转移到文件的末端，这样就可以向一个已存在的文件添加内容了，与之相反 O_TRUNC 将为写操作打开的正常文件的长度删减为 0。O_NOCTTY 防止一个已打开的设备变成一个终端。O_NONBLOCK 负责控制 open 是立即返回还是一直阻塞到设备准备好为止。O_CREAT 创建一个新文件（必须向 open 传递第三个参数 mode），如果想避免重写一个已存在的文件，可以使用 O_CREAT|O_EXECL 组合，如果文件已经存在，这个组合就返回一个错误。

在 UNIX 中，每个文件都有三种用户：所有者、同组的人和其他人，每种用户的访问权限也有三种：读、写和执行，POSIX 定义的操作权限符号名如下所示：

符号	权限	符号	权限
S_IRUSR	所有者读	S_IWUSR	所有者写
S_IXUSR	所有者执行	S_IRWXU	所有者读、写、执行
S_IRGRP	同组的人读	S_IWGRP	同组的人写
S_IXGRP	同组的人执行	S_IRWXG	同组的人读、谢、执行
S_IROTH	其他的人读	S_IWOTH	其他的人写
S_IXOTH	其他的人执行	S_IRWXO	其他的人读、写、执行
S_ISUID	执行时设置用户 ID	S_ISGID	执行时设置组 ID

例如，下面代码在当前目录中创建文件 info.dat，如果文件已存在，它就被重写，所有者可以对文件进行读和写，但所有其他人只能该读文件（注意没有包含头文件）：

```
int fd;
mode_t  fdmode = (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
if ((fd = open("info.dat", O_RDWR | O_CREAT, fdmode)) == -1)
   perror("Failed to open info.dat");
```

close 系统调用只有一个参数 fildes，用来表示待关闭的文件（需要释放资源）。

```
SYNOPSIS
#include <unistd.h>
int close(int fildes);                                    POSIX
```

如果成功，close 返回 0。如果不成功，close 返回-1 并设置 errno。

2.2 读和写文件

read 系统调用试图从用文件描述符 fildes 表示的文件或设备中取出 nbyte 字节，将其放入用户变量 buf 中，必须提供足够大的缓冲区来装载 nbyte 字节的数据（常见的错误是提供了未初始化的指针 buf，而没有提供实际的缓冲区）。如果成功，read 返回实际读出的字节数，如果不成功，read 返回-1 并设置 errno。如果返回 0 表示文件正常结束。

```
SYNOPSIS
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbyte);    POSIX
```

ssize_t 数据类型是用来表示读入字节数的一个带符号的整数数据类型，size_t 是一个表示要读入的字节数的无符号整数数据类型。

write 系统调用试图从用户缓冲区 buf 中向文件描述符 fildes 表示的文件中输出 nbyte 字节。如果成功，write 返回实际写入的字节数，如果不成功，write 返回-1 并设置 errno。

```
SYNOPSIS
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);POSIX
```

下面的程序例子利用 read 和 write 函数实现了一个简单的文件拷贝功能。它假定输入文件已经存在，输出文件不存在，并且每次从输入文件读一个字节，向输出文件写一个字节，最后也没有显式的关闭打开的文件。程序中的文件名要根据实际情况来修改。假定该程序的可执行文件名为 a.out，找一个比较大一些的源文件，用命令 time ./a.out 执行该文件，试解释其输出。你有什么办法可以减少该程序的执行时间？

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

int main()
{
   char c;
   int in,out;

   in = open("file.in", O_RDONLY);
   out = open("file.out", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR);
   while(read(in, &c, 1) == 1)
      write(out, &c, 1);
   return 0;
}
```

下面的程序例子比较完整的实现了拷贝文件的功能。前面的程序包含一个实用函数 copyfile，后面的主程序通过命令行参数输入源文件和目标文件，并调用 copyfile 来完成文件拷贝工作。这里，先将 copyfile 函数申明在 myinclude.h 中，然后将 copyfile.c 用-c 参数编译成.o 文件，最后用命令：

ar –r libmy.a copyfile.o

将其添加到库文件 libmy.a 中。myinclude.h 和 libmy.a 都在当前目录下。

```
#include <unistd.h>
#include <errno.h>
#define BLKSIZE 1024

int copyfile(int fromfd, int tofd)
{
   char *bp;
```

```
    char buf[BLKSIZE];
    int bytesread,byteswritten;
    int totalbytes = 0;

    for (;;){
        while (((bytesread = read(fromfd, buf, BLKSIZE)) == -1) &&
            (errno == EINTR));
        if (bytesread <= 0)
            break;
        bp = buf;
        while (bytesread > 0){
            while (((byteswritten = write(tofd, bp, bytesread))
             == -1) && (errno == EINTR));
            if (byteswritten <= 0)
                break;
            totalbytes += byteswritten;
            bytesread -= byteswritten;
            bp += byteswritten;
        }
        if (byteswritten == -1)
            break;
    }
    return totalbytes;
}
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include "myinclude.h"
```

```
#define READ_FLAGS O_RDONLY
#define WRITE_FLAGS (O_WRONLY | O_CREAT | O_EXCL)
#define WRITE_PERMS (S_IRUSR | S_IWUSR)

int main(int argc, char *argv[])
{
    int bytes;
    int fromfd, tofd;

    if (argc != 3){
        fprintf(stderr, "Usage: %s  source  target\n", argv[0]);
        return 1;
    }
    if ((fromfd = open(argv[1], READ_FLAGS)) == -1){
        perror("Failed to open source");
        return 1;
    }
    if ((tofd = open(argv[2], WRITE_FLAGS, WRITE_PERMS)) == -1){
        perror("Failed to open target");
        return 1;
    }
    bytes = copyfile(fromfd, tofd);
    printf("%d bytes copied from %s to %s\n",bytes, argv[1],
argv[2]);
    return 0;
}
```

练习 3-1：在上述程序的基础上，编写一个能实现文件合并功能的程序。

2.3 更多关于文件的系统调用

ioctl 系统调用主要针对设备文件，是设备驱动程序中对设备的 I/O 通道进行管理的函数。对它的讨论超出了本教材的范围，有兴趣的话可以看一看有关驱动程序设计的资料。

lseek 系统调用对文件描述符 fildes 的读写指针进行设置，也就是说，可以用它来设置文件的下一个读写位置。读写指针既可以是文件中的绝对位置，也可以是相对位置。

```
SYNOPSIS
#include <unistd.h>
#include <sys/types.h>
off_t lseek(int fildes, off_t offset, int whence);   POSIX
```

offset 参数用来指定位置，而 whence 则定义该参数值的用法：SEEK_SET：offset 是一个绝对位置；SEEK_CUR：offset 是相对于当前位置的相对位置；SEEK_END：offset 是相对于文件尾的相对位置。lseek 返回从文件头到文件指针被设置处的字节偏移值，失败时返回-1。参数 offset 的类型 off_t 定义在 sys/types.h 中。请编写程序学习 lseek。

fstat、stat 和 lstat 系统调用返回文件的状态信息。fstat 用打开的文件描述符（fildes）来访问文件，而 stat 和 lstat 则通过文件名字（*path）来访问。参数*buf 返回了该文件的一个 stat 结构的信息。

```
SYNOPSIS
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
int fstat(int fildes, struct stat *buf);
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);       POSIX
```

如果成功，这些函数就返回 0，否则，就返回-1 并设置 errno。在 sys/stat.h 中定义的 struct stat 结构至少包含下列成员：

mode_t	st_mode	文件访问权限和文件类型
ino_t	st_ino	文件节点号
dev_t	st_dev	包含文件的设备 ID
uid_t	st_uid	文件的用户 ID
gid_t	st_gid	文件的组 ID

time_t	st_atime	最后一次访问的时间
time_t	st_ctime	最后一次文件状态改变的时间
time_t	st_mtime	最后一次数据修改的时间
off_t	st_size	以字节为单位的文件长度（正常文件）
nlink_t	st_nlink	硬链接的数量

下面的程序显示了文件 ex_stat.c 的相关信息：

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

int main(void)
{
   int fd;
   struct stat statbuf;
   char filename[] = "ex_stat.c";

   if (stat(filename, &statbuf) == -1){
      perror("Failed to get file status");
      return 1;
   }
   printf("device ID is: %u\n", statbuf.st_dev);
   printf("Inode is: %u\n", statbuf.st_ino);
   printf("access mode is: %o\n", statbuf.st_mode);
   printf("hard link is: %lu\n", statbuf.st_nlink);
   printf("user ID is: %u\n", statbuf.st_uid);
   printf("group ID is: %u\n", statbuf.st_gid);
   printf("size is: %u\n", statbuf.st_size);
   printf("last accessed at %s",  ctime(&statbuf.st_atime));
```

```
    printf("last modify at %s", ctime(&statbuf.st_mtime));
    printf("last change at %s",  ctime(&statbuf.st_ctime));
    return 0;
}
```

dup 和 dup2 系统调用提供了一种在文件描述符表中复制文件描述符的方法，使我们能够通过两个或更多个不同的文件描述符来访问同一个文件，这可用于在文件的不同位置对文件进行读写。dup 复制文件描述符 fildes 到文件描述符表（见下）中一个空的条目中，然后返回这个空条目的索引；dup2 通过明确指定目标文件描述符来把一个文件描述符复制为另外一个，即把 oldfildes 放到文件描述符表中 newfildes 所在的条目上，这样，以后对 newfildes 的操作就变成了对 oldfildes 的操作。在 UNIX 中，利用 dup2 系统调用可以很方便的实现重定向功能。

下面的程序将标准输出重定向到 myfile.dat 中，然后向那个文件附加一条短消息。

```
SYNOPSIS
#include <unistd.h>
int dup(int oldfildes);
int dup2(int oldfildes， int newfildes)；            POSIX
```

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#define CREATE_FLAGS (O_WRONLY | O_CREAT | O_APPEND)
#define CREATE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

int main(void)
{
    int fd,x;

    while (((fd = open("myfile.dat", CREATE_FLAGS, CREATE_MODE))
```

```
== -1) && (errno == EINTR));
       if (fd == -1){
          perror("Failed to open file");
          return 1;
       }
       if (dup2(fd, STDOUT_FILENO) == -1){
          perror("Failed to redirect standard output");
          return 1;
       }
       if (close(fd) == -1){
          perror("Failed to close file");
          return 1;
       }
       if (write(STDOUT_FILENO, "OK", 2) == -1){
          perror("Failed to writing to file");
          return 1;
       }
       return 0;
   }
```

select 系统调用提供了一种在单个进程中监视文件描述符的方法。在实际应用中，经常需要监视多个文件描述符，例如一个程序期待着来自连个不同源端的输入，但不知道那个源端的输入先到。监视多个文件描述符的一种方法是为每个描述符分别使用一个独立的进程，但这需要建立某种形式的进程间通信机制。select 可以对三种可能的操作文件描述符的状况进行监视——无阻塞的进行读操作、无阻塞的进行写操作及有挂起的错误情况。

```
SYNOPSIS
#include <sys/select.h>
int select(int nfds,  fd_set *restrict readfds,
                      fd_set *restrict writefds,
```

```
                        fd_set *restrict errorfds,
                struct timeval *restrict timeout);
   void FD_CLR(int fd, fd_set *fdset);
   int FD_ISSET(int fd, fd_set *fdset);
   void FD_SET(int fd, fd_set *fdset);
   void FD_ZERO(fd_set, *fdset);                          POSIX
```

参数 nfds 给出了要监视的文件描述符的范围，nfds 的值必须至少比要监视的最大的文件描述符大 1。参数 readfds 指定了为读操作监视的文件描述符集。writefds 指定了为写操作监视的文件描述符集。errorfds 指定了为错误情况监视的文件描述符集。文件描述符集的类型为 fd_set，这些参数中的任何一个都可能为 NULL，在这种情况下，select 不为相应的事件监视文件描述符。最后一个参数是超时值，经过一段特定的时长之后，即使没有准备就绪的文件描述符，它也会迫使 select 返回，如果 timeout 为 NULL，select 可能会无期限的阻塞。

成功返回时，除了那些已经准备就绪的文件描述符之外，select 清空 resdfds、writefds 和 errorfds 中其他描述符，最后 select 返回已准备就绪的文件描述符的数目。如果不成功，select 返回-1 并设置 errno。

宏 FD_SET 设置了*fdset 对应于文件描述符 fd 的那个比特位，宏 FD_CLR 清除了相应的比特位，宏 FD_ZERO 清除了*fdset 中所有的比特位。在调用 select 之前，用这三个宏来设置文件描述符掩码，在 select 返回之后用宏 FD_ISSET 来测试掩码中对应于文件描述符 fd 的那个比特位是否被设置了。

fcntl 系统调用可用来检索和修改与打开的文件描述符相关联的标志符。fildes 指定了描述符，参数 cmd 指定了操作，fcntl 可根据 cmd 的值使用额外的参数。对 fcntl 返回值的解释取决于参数 cmd 的值。但是，如果不成功，fcntl 就返回-1 并设置 errno。

使用 fcntl 的一个重要的例子就是修改对打开的文件描述符的操作，使其使用无阻塞 I/O。将文件描述符设置为无阻塞 I/O 后，如果试图进行阻塞型 I/O 操作，read 和 write 就会返回-1 并将 errno 设置为 EAGAIN，以此来报告进程可能会被延迟。在需要一边做别的工作一边监视多个文件描述符的情况下，无阻塞 I/O 是很有用的。前面所述 select 函数允许进程一直阻塞到描述符集中的任何一个描述符可用为止，但是其在等待其间不能做其它工作。要执行无阻塞 I/O，程序可以用 O_NONBLOCK 标志符集来调用 open，程序也可以用 fcntl 设置 O_NONBLOCK 标志符，将打开的标识符改成无阻塞的。

练习 3-2：

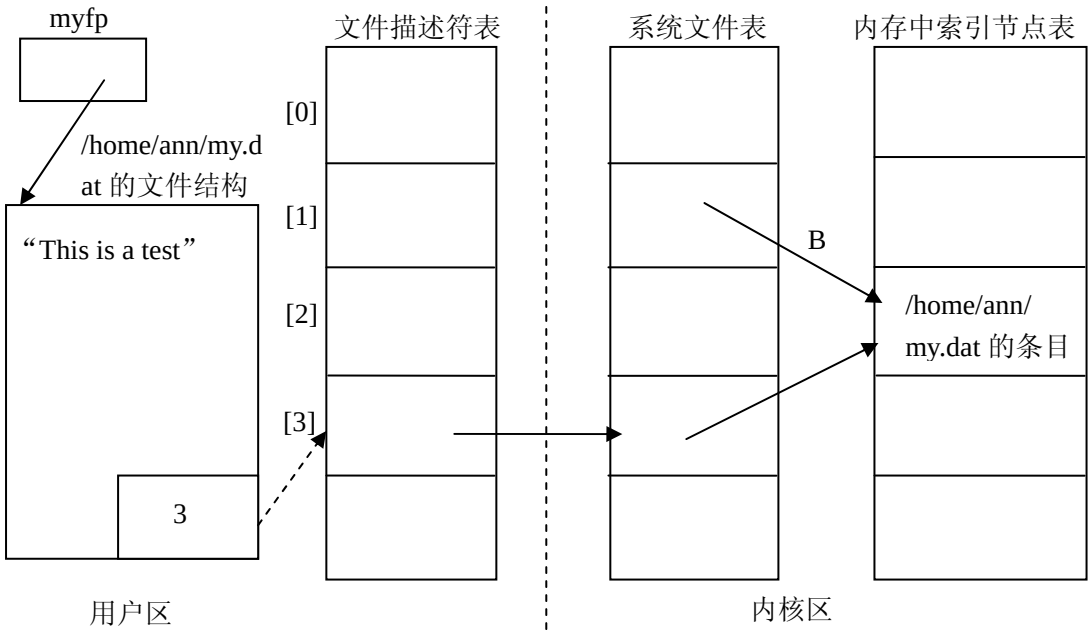
通过联机帮助页面学习 cat 命令的功能和使用方法，编程尽量实现 cat 的功能（注意出错处理）。

3、标准 I/O 库

上述 UNIX 的 I/O 系统调用（open、read、write、ioctl、close）使用文件描述符，而 ANSI C 的标准 I/O 库函数（fopen、fscanf、fprintf、fread、fwrite、fclose）则使用文件指针。文件指针和文件描述符提供了用来执行独立于设备的输入和输出的逻辑标识，这些逻辑标识被称为句柄（handle）。代表标准输入、标准输出和标准错误的文件标识符的符号名分别为 STDIN_FILENO、STDOUT_FILENO 和 STDERR_FILENO，它们定义在 unistd.h 中。代表标准输入、标准输出和标准错误的文件指针的符号名分别为 stdin、stdout 和 stderr，它们定义在 stdio.h 中。

open 系统调用将文件或物理设备与程序中使用的逻辑句柄相关联，用字符串（例如 /home/ann/my.dat 或/dev/tty）来指定文件或物理设备。句柄是一个整数，可以将其理解为进程特定的文件描述符表（file descriptor table）的索引，对进程中每个打开的文件，文件描述符表都包含一个相应条目。文件描述符表是进程用户区的一部分，但是除非通过使用文件描述符的函数，否则程序对其无法访问。语句：

```
myfd = open("/home/ann/my.dat", O_RDONLY);
```



在文件描述符表中创建了一个条目，它指向系统文件表中的一个条目。open 系统调用返回值为 3，说明文件描述符条目在进程文件描述符表的位置 3 上（位置 0、1、2 上分别是缺省打开的标准输入、标准输出和标准错误）。

系统文件表（system file table）为系统中所有的进程共享，对每个活动的 open，它都包含一个条目，每个系统文件表条目都包括文件偏移量、访问模式指示（读、写、读-写）以及指向它的文件描述符表条目的数目计数。

几个系统文件表条目可能对应于一个物理文件，这些条目中的每一个都指向内存中索引节点表（in-memory inode table）中的同一个条目。对系统中（不是进程中）的每个活动文件，内存中索引节点表都包含一个条目。当程序打开一个当前没有打开的特定的物理文件时，调用在这个索引节点表中为那个文件创建一个条目，否则只建立一个连接。请思考：为什么要有系统文件表？文件描述符表直接指向内存中索引节点表不行吗？

而 fopen 函数关联的文件指针（file pointer）则是一个被称为 FILE 结构的数据结构，它主要包括一个缓冲区和一个文件描述符值，这个文件描述符值又是通过 fopen 函数调用 open 系统调用来返回的。也即标准库函数是系统调用的外套。但是标准库函数不同于系统调用的一是它使用文件指针而非文件描述符，二是它有缓冲区，读时只有数据量等于缓冲区大小才调用 read，写时只有先写满缓冲区才调用 write。缓冲区功能可以避免频繁的使用系统调用而提高程序效率，因为系统调用在内核。下面程序段结合上图就能理解文件在操作系统级的组织以及如何面对用户的接口。

```
FILE *myfd;

if ((myfd = fopen(“/home/ann/my.dat”, “w”)) == NULL)
   perror(“Failed to open file”);
else
   fprintf(myfd, “This is a test”);
```

那么程序调用 fprintf 时会发生什么情况呢？这取决于被打开文件的类型。磁盘文件通常都是完全缓冲（fully buffered）的。这就意味着上例中 fprintf 实际上没有将消息 This is a test 写入磁盘，而是将这些字节写入了 FILE 结构的一个缓冲区里，等到缓冲区填满时才调用 write 将其写入磁盘。这种 fprintf 的时刻和实际进行写操作的时刻之间的时延会造成很有趣的结果。另外与终端相关的文件是行缓冲（line buffered）的，而不是完全缓冲的（默认情

况下标准错误是不缓冲的）。对输出来说，行缓冲意味着在缓冲区被填满之前或遇到一个新行符号之前，行不会被写出。

可以通过 ANSI C 标准库函数 fflush 强制写出 FILE 结构中缓冲的任何内容，也可以通过另一个标准库函数 setvbuf 来禁止缓冲。一般程序结束时也相当于调用了 fflush：

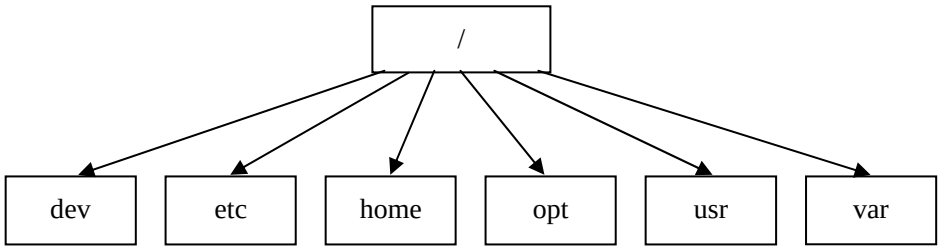
```
#include <stdio.h>
int fflush(FILE * stream);
int setvbuf(FILE * stream);
```

4、目录操作

文件系统（file system）是文件组织结构及其属性的集合。UNIX 文件系统的组织采用倒树型结构，最顶端的节点被称为文件系统的根目录（root directory），用“/”表示。根目录下面包含文件或其他子目录，子目录下面又可以包含文件或其他子目录，以此类推。

物理磁盘与文件系统的关系是：磁盘格式化将物理磁盘分隔成被称为分区（partition）的区域，每个分区都可以有自己的文件系统与之相关联，一个特定的文件系统可以加载到另一个文件系统树的任意一个节点上。这样，UNIX 就可以在保证一个完整的倒树型目录结构的同时，还可以支持不同的文件系统。

下图是一个典型的文件系统，包含了一些标准的 UNIX 子目录：

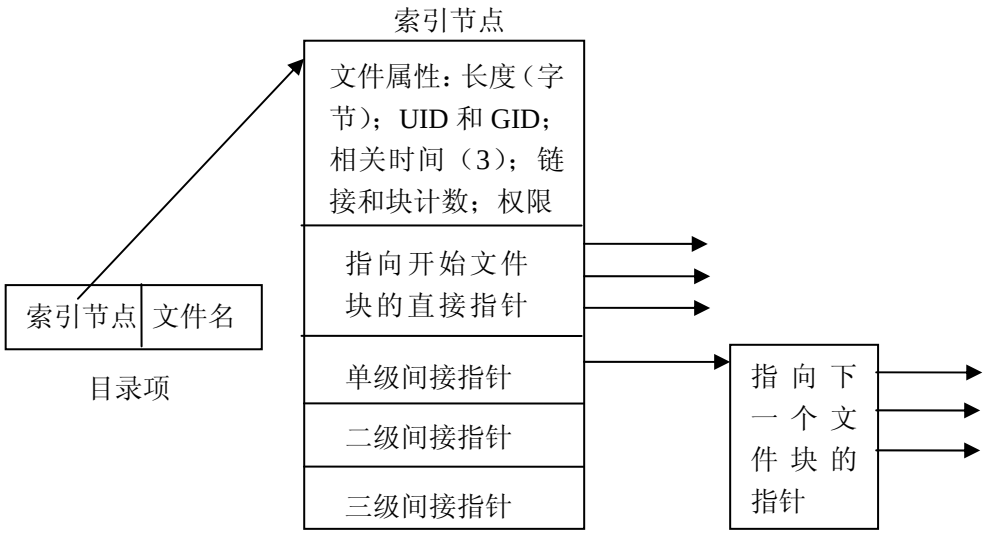


与设备有关的特殊文件	系统相关的特性	用 户目录	应用程序包	可共享文件	可变的文件（日志）

目录是一个包含了目录项（directory entry）的文件，目录项用来将文件名与文件在磁盘上的物理位置关联起来。每个目录项包含一个文件名及一个定长结构的引用，这个定长结构被称为索引节点（inode），包括了与文件长度、文件位置、文件所有者、创建时间、最后访问时间、最后修改时间以及权限等有关的信息。

索引节点中还包含了指向保存文件具体内容的文件数据块的指针。如果文件比较大，那么间接指针就可以派上用场，间接指针（indirect pointer）是一个指向指针块的指针，而指针块中的指针才指向具体的数据块。还有二级间接指针（double indirect pointer）和三级间接指针（triple indirect pointer）。这是一种非常优秀的分配方案！

因此，在 UNIX 支持的文件系统中文件的名字、文件的属性和文件的内容分别放在三个不同的逻辑位置，下图简单说明之：



只包含文件名和索引节点的目录项实现有下列优点：

（1）修改文件名时只需修改目录项即可。移动文件时只要始终在同一个分区，那么仅移动目录项就可以将文件从一个目录转移到另一个目录中（mv 命令就是利用这种技术）；

（2）在磁盘上只需保存文件的一份物理拷贝，但是文件可以有几个名字或在不同目录下使用同一个名字；但前提是必须在同一个分区中（这样 cp 才有用）；

（3）因为文件名是变长的，所以目录项的长度是可变的，由于有关文件的大部分信息都存放在它的索引节点中，因此目录项很小，对可变长的小结构的操作可以是很高效的，比较大的索引节点结构则是定长的。效率就是这样一点一滴来提高的。

假设一个索引节点有 128 个字节，指针为 4 个字节长，而状态信息占用了 68 个字节。假定一个数据块大小为 8K 字节。则直接指针可以表示一个多大的文件？间接指针、二级间接、三级间接指针呢？指针大小与分区大小有什么关系？块大小会影响什么？

任何时候，每个进程都有一个用来做路径名解析的相关目录，这个目录被称为当前目录（current working directory）。环境变量 PWD 指定了进程的当前工作目录。不要直接修改这

个变量，应该用 getcwd 函数检索出当前的工作目录，然后用 chdir 函数来修改进程中的当前工作目录。

```
SYNOPSIS
#include <unistd.h>
int chdir(const char *path);
char *getcwd(char *buf， size_t size);                    POSIX
```

chdir 函数使 path 指定的目录称为调用进程的当前工作目录，如果成功，chdir 返回 0，否则，返回-1 并设置 errno。

getcwd 的参数 buf 表示一个由用户提供的缓冲区，这个缓冲区用来存放当前工作目录的路径名，参数 size 指定了 buf 可以存储的路径名的最大长度，这个长度包含了结尾处的字符串终止符。如果成功，getcwd 返回一个指向 buf 的指针，否则，返回 NULL 并设置 errno。

如果 buf 不为 NULL，getcwd 就将路径名拷贝到 buf 中去，如果 buf 为 NULL，POSIX 声明 getcwd 的行为是未定义的。在某些实现中，getcwd 用 malloc 创建了一个缓冲区来装载路径名，但不能依赖于这种行为。

PATH_MAX 是一个可选的 POSIX 常量，用来指定实现中路径名的最大长度（包括终止的空字节），limit.h 中可能定义也可能没有定义常量 PATH_MAX。一种更灵活的方法是用 pathconf 函数来确定运行时最大路径长度的实际值。pathconf 是允许程序以一种与平台无关的方式来确定系统合运行期极限的函数族中的一个，例如 sysconf 用来计算程序的一些极限值。sysconf 只有一个参数，这个参数是一个可配置的、在全系统范围内都起作用的极限的名字，比如每秒钟的时钟计数次数（_SC_CLK_TCK）或者每个用户允许使用的最大进程数（_SC_CHILD_MAX）。pathconf 和 fpathconf 负责报告与一个特定的文件或目录相关的极限。

```
SYNOPSIS
#include <unistd.h>
long fpathconf(int fildes, int name);
long pathconf(const char *path, int name);
long sysconf(int name);                                  POSIX
```

下面的程序例子显示当前目录：

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

```
#include <unistd.h>

int main(void)
{
long maxpath;
char *mycwdp;

if ((maxpath = pathconf(".", _PC_PATH_MAX)) == -1){
    perror("Failed to determine the pathname length");
    return 1;
}
if ((mycwdp = (char *) malloc(maxpath)) == NULL){
    perror("Failed to allocate space for pathname");
    return 1;
}
if (getcwd(mycwdp, maxpath) == NULL){
    perror("Failed to get current working directory");
    return 1;
}
printf("Current working directory: %s\n",mycwdp);
}
```

目录不能用普通的 open、read 和 close 函数来访问。相反，访问目录需要使用特定的函数，函数名以“dir”结束：opendir、readdir 和 closedir。

opendir 为一个目录流提供了 DIR*类型的句柄，该流的当前位置就在目录的第一项上。

```
SYNOPSIS
#include <dirent.h>
DIR *opendir(const char *dirname);                    POSIX
```

如果成功，opendir 返回一个指向目录对象的指针，否则 opendir 返回 NULL 并设置 errno。

定义在 dirent.h 中的 DIR 类型表示的是一个目录流（directory stream），目录流是一个特

定目录中所有目录项组成的有序序列，目录流中的条目不一定按文件名的字母顺序排列。

readdir 通过返回 dirp 所指向的目录流中的连续条目来读取目录的，readdir 返回一个指向 struct dirent 结构的指针，这个结构中包含了与下一个目录项有关的信息，readdir 在每次调用之后都将流转移到下一个位置上去。

```
SYNOPSIS
#include <dirent.h>
struct dirent *readdir(DIR *dirp);                    POSIX
```

如果成功 readdir 返回一个指向 struct dirent 结构的指针，否则，返回 NULL 并设置 errno。readdir 的实现必须检测的错误只有一种，就是要返回的结构中的值无法正确表达，其错误码是 EOVERFLOW。readdir 返回 NULL 时表示到目录尾，但此时它并不该变 errno。

closedir 关闭一个目录流，而 rewinddir 把目录流重定位在起始处。每个函数都有一个参数 dirp，表示打开的目录流。

```
SYNOPSIS
#include <dirent.h>
int closedir(DIR *dirp);
void rewinddir（DIR *dirp)；                          POSIX
```

如果成功 closedir 返回 0，否则，返回-1 并设置 errno。rewinddir 没有返回值。

下面程序列出了目录中的文件：

```
#include <dirent.h>
#include <errno.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
struct dirent *direntp;
DIR *dirp;

if (argc != 2){
    fprintf(stderr, "Usage: %s directory_name\n",argv[0]);
```

```
    return 1;
}
if ((dirp = opendir(argv[1])) == NULL){
    perror("Failed to open directrory");
    return 1;
}
while ((direntp = readdir(dirp)) != NULL)
    printf("%s\n", (*direntp).d_name);
while ((closedir(dirp) == -1) && (errno == EINTR));
return 0;
}
```

注意最后的 printf 语句中对结构成员的引用，它等价于 direntp->d_name。

下面的程序模仿命令“ls -l”：

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

int printfile(struct stat statbuf)
{
    char fileattr[] = "----------";
    char *mtime;

    if (S_ISBLK(statbuf.st_mode)) fileattr[0] = 'b';
    if (S_ISCHR(statbuf.st_mode)) fileattr[0] = 'c';
    if (S_ISDIR(statbuf.st_mode)) fileattr[0] = 'd';
    if (S_ISFIFO(statbuf.st_mode)) fileattr[0] = 'p';
    if (S_ISLNK(statbuf.st_mode)) fileattr[0] = 'l';
```

```
    if (S_ISSOCK(statbuf.st_mode)) fileattr[0] = 's';
    if (statbuf.st_mode & S_IRUSR) fileattr[1] = 'r';
    if (statbuf.st_mode & S_IWUSR) fileattr[2] = 'w';
    if (statbuf.st_mode & S_IXUSR) fileattr[3] = 'x';
    if (statbuf.st_mode & S_IRGRP) fileattr[4] = 'r';
   if (statbuf.st_mode & S_IWGRP) fileattr[5] = 'w';
   if (statbuf.st_mode & S_IXGRP) fileattr[6] = 'x';
    if (statbuf.st_mode & S_IROTH) fileattr[7] = 'r';
   if (statbuf.st_mode & S_IWOTH) fileattr[8] = 'w';
   if (statbuf.st_mode & S_IXOTH) fileattr[9] = 'x';
    fprintf(stdout, "%10s", fileattr);
    fprintf(stdout, "%5lu", statbuf.st_nlink);
    fprintf(stdout, "%5u", statbuf.st_uid);
    fprintf(stdout, "%5u", statbuf.st_gid);
    fprintf(stdout, "%10u", statbuf.st_size);
    mtime = ctime(&statbuf.st_mtime);
    mtime[24] = '\0';
    fprintf(stdout, "%30s", mtime);
    return 0;
}

int main(int argc, char *argv[])
{
    struct stat statbuf;
    struct dirent *direntp;
    DIR *dirp;

    if (argc == 1)
       argv[1] = ".";
```

```
    if ((dirp = opendir(argv[1])) == NULL){
        perror("Failed to open directory");
        return 1;
    }
    chdir(argv[1]);
    while ((direntp = readdir(dirp)) != NULL){
        if (stat(direntp->d_name, &statbuf) == -1){
            perror("Failed to get file status");
            return 1;
        }
        printfile(statbuf);
        fprintf(stdout, "   %s\n", direntp->d_name);
    }
}
```

练习 3-3：参考上面程序，编写一个遍历目录的程序，要求：（1）只显示目录名和文件名，某目录下的文件名要缩进；（2）带选项，若为“-d”，则按深度优先搜索，否则，则按广度优先搜索，考虑选项的位置（可在参数前，也可在参数后）；（3）带参数，若有，从该目录开始，否则，从当前目录开始。

5、链接文件

UNIX 操作系统提供文件的链接操作，链接是指把一个新的文件名链接到一个已存在的文件上，这样两个不同位置的文件就指向同一个物理文件。链接分为硬链接和软链接，在实际的使用中，硬链接和软链接的行为差异通常不是很明显。对于链接，我们要慎用。

可以用 ln 命令或 link 系统调用为文件创建一个额外的链接，创建新链接会分配一个新的目录项，并增加相应索引节点的链接计数（没有被链接的文件，其索引节点的链接计数为 1）。链接不再使用其它额外的磁盘空间。

当你通过执行命令 rm 或从程序中调用 unlink 系统调用来删除一个文件时，操作系统会删除相应的目录项，并减少索引节点的链接计数。除非这个操作使链接计数减少为 0，否则，操作系统不会释放索引节点和相应的数据块。

```
SYNOPSIS
#include <unistd.h>
int link(const char *path1， const char *path2);
unlink（const char *path）;                                    POSIX
```

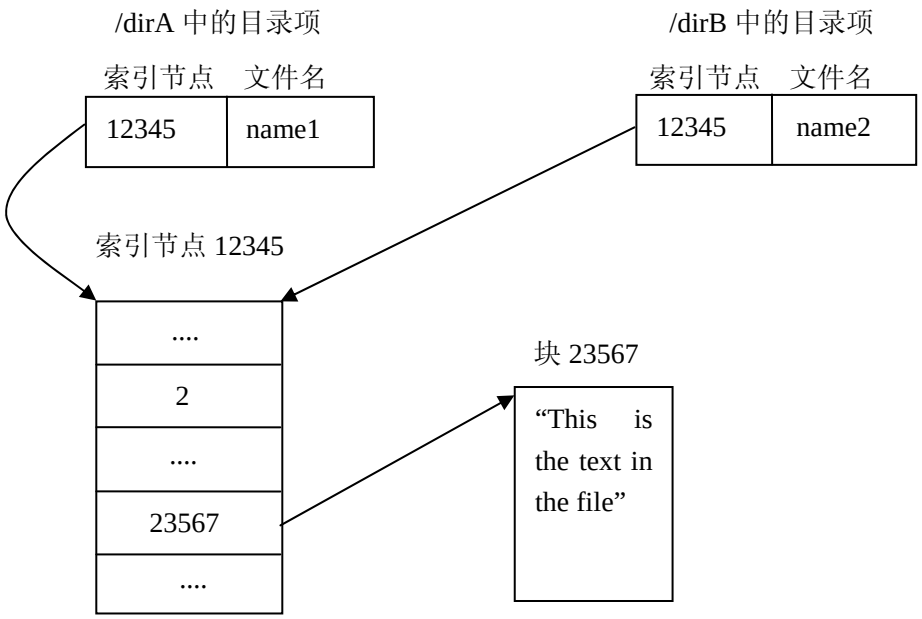
link 函数为 path1 指定的已存在文件创建一个新的目录项，这个文件位于 path2 指定的目录中。如果成功，link 返回 0，否则返回-1 并设置 errno。unlink 删除了 path 指定的目录项，如果文件的链接计数为 0，而且没有进程打开这个文件，unlink 就释放文件占据的空间。如果成功，unlink 返回 0，否则返回-1 并设置 errno。

下面代码执行的动作与 ln /dirA/name1 /dirB/name2 命令执行的动作相同：

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

if (link(“/dirA/name1”, ”/dirB/name2”) == -1)
   perror(“Failed to make a new link in /dirB”);
```

具体结果如下图所示：



符号链接是一个包含了另一个文件或目录名字的文件。引用符号链接的名字会使操作系统去定位对应于那个链接的索引节点，操作系统假设相应的索引节点的数据块中包含另一个路径名，然后操作系统对那个路径的目录项进行定位，并继续跟踪这个链接，直到最终遇到一个硬链接和一个真正的文件为止。如果系统经过一段时间还没有找到真正的文件，它就放

弃并返回 ELOOP 错误。

```
SYNOPSIS
#include <unistd.h>
int symlink(const char *path1， const char *path2);   POSIX
```

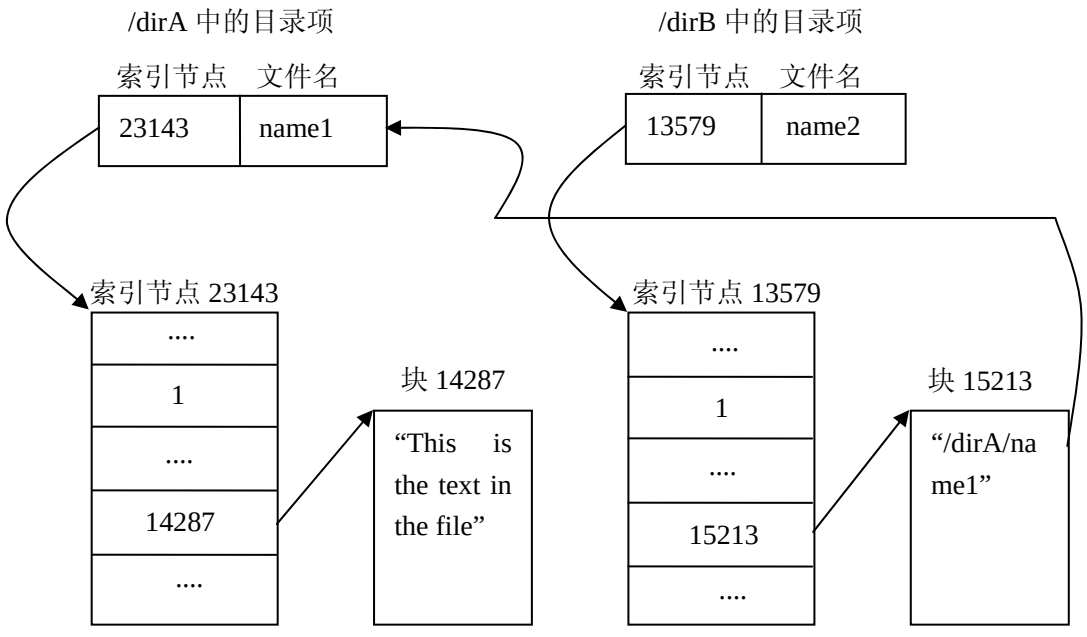
使用带有选项-s 的 ln 命令或在程序中调用 symlink 函数，就可以创建一个符号链接。symlink 的参数 path1 包含了将要成为链接内容的字符串，path2 给出了链接的路径名，也即，path2 是新创建的链接，而新链接指向 path1。如果成功，symlink 返回 0，否则返回-1 并设置 errno。

下面代码执行的动作与 ln -s /dirA/name1 /dirB/name2 命令执行的动作相同：

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

if (symlink(“/dirA/name1”, ”/dirB/name2”) == -1)
    perror(“Failed to create symbolic link in /dirB”);
```

具体结果如下图所示：



为一个符号链接建立一个硬链接是不可靠的。