File

张海宁

贵州大学

hnzhang1@gzu.edu.cn

December 19, 2018

Overview

文件结构

Linux 中，一切（或几乎一切）都是文件。
这意味着针对串口，打印机等设备的操作可以像文件一样来进行操作。
文件可以分为以下两类：

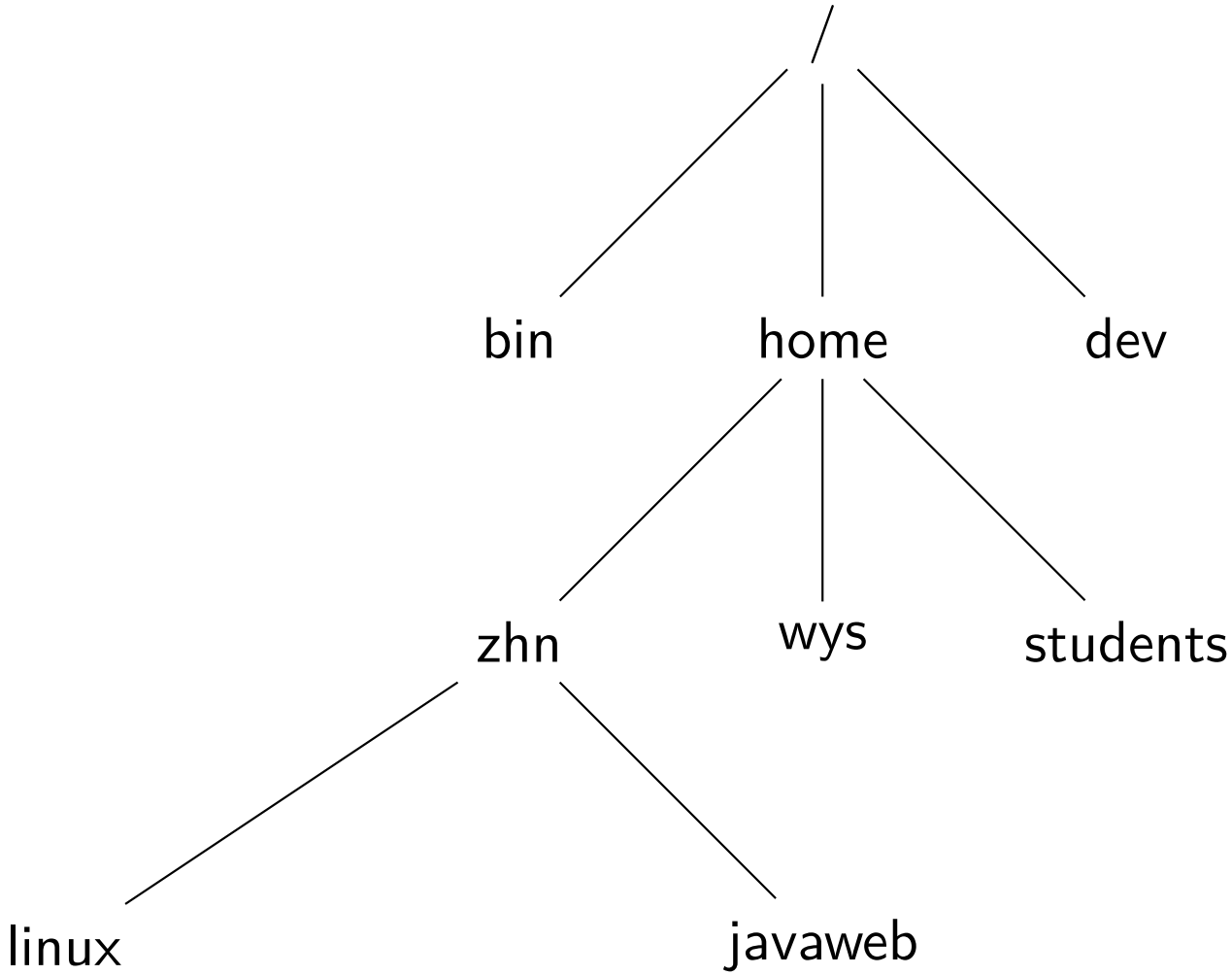
1. 目录
   特殊的文件
2. 文件和设备

Linux 系统是通过 inode 来识别文件的，文件名只是为了用户的使用方便。

inode

inode 是 linux 系统中的一种数据结构。它存储了文件系统对象（包括文件、目录、设备文件、socket、管道，等等）的元信息数据，但不包括数据内容或者文件名。
文件系统中每个“文件系统对象”对应一个“inode”数据，并用一个整数值来辨识。这个整数常被称为 inode 号码（“i-number”或“inode number”）。
Inode 存储了文件系统对象的一些元信息，如所有者、访问权限（读、写、执行）、类型（是文件还是目录）、内容修改时间、inode 修改时间、上次访问时间、对应的文件系统存储块的地址，等等。
一个文件系统对象可以有多个别名，但只能有一个 inode，并用这个 inode 来索引文件系统对象的存储位置。
使用 stat 系统调用可以查询一个文件的 inode 号码及一些元信息。

目录



文件和设备

硬件设备在 Linux 系统中也通常被映射为文件，比如 U 盘，会被 Linux 系统自动挂载到一个目录下面。
Linux 系统中，比较重要的设备文件有：

- /dev/console
  这个设备代表系统控制台，通常是指“活跃的虚拟控制台”，或者是一个特殊的控制台窗口。
- /dev/tty
  如果一个进程有控制终端的话，那么，特殊文件/dev/tty 就是这个控制终端的别名。
- /dev/null
  这是空设备，任何写向这个设备（文件）的输出都将被丢弃。

与底层文件访问有关的几个重要的系统调用

每个运行中的程序被称为进程，它有一些与之相关的文件描述符，可以通过这些文件描述符来访问打开的文件或设备。在运行程序时，一般有这 3 个已经打开的文件描述符：0,1,2.（各是什么意思？）
下面列出几个常用的与文件访问有关的系统调用：

1. write
2. read
3. open
4. close

write

系统调用 write 的作用是把缓冲区 buf 的前 nbyte 写入与文件描述符相关联的文件中去，会返回实际写入的字节数，若未写入任何数据则返回 0, 返回-1 表示出现了错误，对应的错误代码会保存在全局变量 errno 中。

Example (write 系统调用的原型)

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

write

Example (write 系统调用的使用实例)

```
 1  #include<unistd.h>
 2  #include<stdlib.h>
 3  #include<stdio.h>
 4
 5  int main(){
 6    if((write(1,"Here is some data.\n",18)!=18)){
 7      write(2,"A write error has occurred!\n",46);
 8    }
 9   printf("continue?");
10  exit(0);
11 }
```

read

系统调用 read 的作用是从与文件描述符相关联的文件中读出前 nbyte 写入到缓冲区 buf 里，会返回实际读出的字节数，若返回 0 则表示读到了文件的末尾，返回-1 表示出现了错误。

Example (read 系统调用的原型)

```
#include <unistd.h>
size_t read(int fildes, void *buf, size_t nbytes);
```

read

Example (read 系统调用的使用实例)

```
 1  #include<unistd.h>
 2  #include<stdlib.h>
 3
 4  int main(){
 5  char buffer[128];
 6  int nread;
 7 nread = read(0,buffer,128);
 8  if(nread==-1){
 9   write(2,"A read error has occurred.\n",27);
10  }
11  if((write(1,buffer,nread))!=nread){
12    write(2,"A write error has occurred.\n",27);
13  }
14  exit(0);
15  }
```

运行 read 程序的方法

1. 使用管道输入
2. 使用输入重定向进行输入
3. 直接运行程序，从终端输入

```
cat simple_read.c | ./read
./read < simple_read.c
./read
```

open

系统调用 open 的作用是创建文件描述符。这个文件描述符可以提供给 read 或 write 系统调用使用。
两个程序打开同一个文件会得到两个不同的文件描述符，后写入的内容会覆盖之前写入的内容!

Example (open 系统调用的原型)

```
#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);
```

oflags

oflags 是用来指定打开文件方式的参数。open 调用必须指定表??中所示的文件访问模式之一。

模式	说明
O_RDONLY	以只读方式打开
O_WRONLY	以只写方式打开
O_RDWR	以读写方式打开

Table: 主要文件访问模式

oflags-II

oflags 参数还可以包含以下可选模式的组合（按位或）。

模式	说明
O_APPEND	追加数据在文件末尾
O_TRUNC	丢弃文件原有内容
O_CREAT	如果需要，就按照 mode 中给出的访问模式创建文件

Table: 可选文件访问模式

按位或

```
/usr/include/sys/fcntl.h
#define O_CREAT   0x0200   /* create if nonexistant */
#define O_TRUNC   0x0400   /* truncate to zero length */
#define O_EXCL    0x0800   /* error if already exists */
```

O_CREAT

当使用带有 O_CREAT 标志的 open 来创建文件时，必须使用带有三个参数格式的 open 调用。第三个参数 mode 是几个标志按位或得到的。

```
cat /usr/include/sys/_types/_s_ifmt.h
#define S_IRWXU  0000700  /* [XSI] RWX mask for owner
#define S_IRUSR  0000400  /* [XSI] R for owner */
#define S_IWUSR  0000200  /* [XSI] W for owner */
#define S_IXUSR  0000100  /* [XSI] X for owner */
#define S_IRWXG  0000070  /* [XSI] RWX mask for group
#define S_IRGRP  0000040  /* [XSI] R for group */
#define S_IWGRP  0000020  /* [XSI] W for group */
#define S_IXGRP  0000010  /* [XSI] X for group */
#define S_IRWXO  0000007  /* [XSI] RWX mask for other
#define S_IROTH  0000004  /* [XSI] R for other */
#define S_IWOTH  0000002  /* [XSI] W for other */
#define S_IXOTH  0000001  /* [XSI] X for other */
```

open

Example (open 系统调用的使用实例)

```
1 #include <fcntl.h>
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <stdlib.h>
5
6 int main(){
7 open("myfile",O_CREAT,S_IRUSR|S_IWUSR);
8 exit(0);
9 }
```

```
ls -al myfile
-rw------- 1 hnz staff 0 Apr 10 00:34 myfile
```

umask

文件被创建时，会指定访问权限，比如 open 使用带有 O_CREAT 标志的参数来创建文件时，会指定权限。但最终的文件权限还是要结合 umask（也叫用户掩码，是 8 进制数）码来确定。
open 的权限位会与 umask 的反码做 AND 操作。

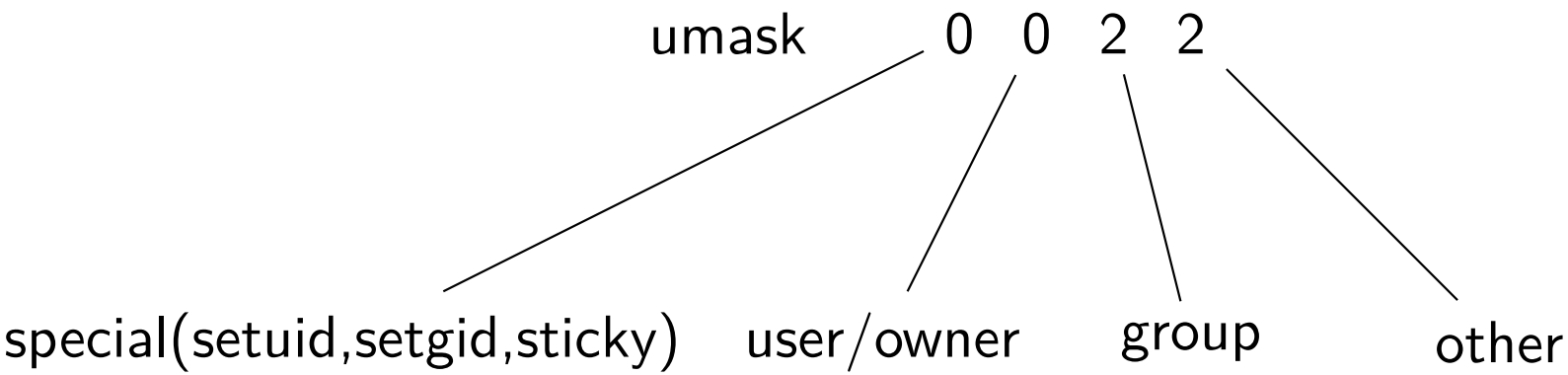
```
$ umask              # display current value (as octal)
0022
$ umask -S           # display current value symbolically
u=rwx,g=rx,o=rx
```

umask 值的说明

umask	Permissions
0	any permission may be set (read, write, execute)
1	setting of execute permission is prohibited (read and write)
2	setting of write permission is prohibited (read and execute)
3	setting of write and execute permission is prohibited (read only)
4	setting of read permission is prohibited (write and execute)
5	setting of read and execute permission is prohibited (write only)
6	setting of read and write permission is prohibited (execute only)
7	all permissions are prohibited from being set (no permissions)

Table: umask 值说明

umask


umask
0 0 2 2
special(setuid,setgid,sticky)
user/owner
group
other

special bit in umask

setuid and setgid (short for "set user ID upon execution" and "set group ID upon execution", respectively) are Unix access rights flags that allow users to run an executable with the permissions of the executable's owner or group respectively and to change behaviour in directories. They are often used to allow users on a computer system to run programs with temporarily elevated privileges in order to perform a specific task. While the assumed user id or group id privileges provided are not always elevated, at a minimum they are specific.

When a directory's sticky bit is set, the filesystem treats the files in such directories in a special way so only the file's owner, the directory's owner, or root user can rename or delete the file. Without the sticky bit set, any user with write and execute permissions for the directory can rename or delete contained files, regardless of the file's owner. Typically this is set on the /tmp directory to prevent ordinary users from deleting or moving other users' files.

close 系统调用

close 系统调用是用来终止一个文件描述符与其对应文件之间关联的，这个文件描述符会被释放并能被重新使用。其返回值为 0 或-1 。

Example (close 系统调用的原型)

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

实验：拷贝文件

使用 open,read,write 这三个系统调用，编写一个程序 simple_copy.c，将一个文件逐个字符 (或多个字符) 地拷贝到另外一个文件中去。
以下的系统调用原型供参考：

```
#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);
size_t read(int fildes, void *buf, size_t nbytes);
size_t write(int fildes,
        const void *buf, size_t nbytes);
```

实验：拷贝文件代码

```
 1 #include <unistd.h>
 2 #include <sys/stat.h>
 3 #include <fcntl.h>
 4 #include <stdlib.h>
 5 int main(){
 6 char c;
 7 int in,out;
 8 in = open("simple_read.c",O_RDONLY);
 9 out = open("copy_example",
          O_WRONLY|O_CREAT,S_IRWXU);
10 while(read(in,&c,1) ==1 ){
11 write(out,&c,1);
12 write(1,&c,1);}
13 close(in); close(out);
14 exit(0);
15 }
```

标准 I/O 库

标准 I/O 库及其头文件 stdio.h 为底层 I/O 系统调用提供了一个通用的接口。标准 I/O 库提供了许多复杂的函数，用于格式化输出和扫描输入，并且具有缓冲功能。
标准 I/O 库与底层文件描述符很类似，在启到一个程序时，也是有三个流是自动打开的，分别是 stdin,stdout,stderr 与 0,1,2 对应。
标准 I/O 库中的常用文件操作库函数：

1. fopen fclose
2. fread fwrite
3. fflush
4. fgetc getc getchar
5. fputc putc putchar
6. fgets gets
7. printf fprintf sprintf
8. scanf fscanf sscanf

参考资料：
https://www.tutorialspoint.com/c_standard_library/index.htm

fopen

fopen 函数类似系统调用 open。主要用于文件和终端的输入输出。

Example (fopen 函数原型)

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

fopen 打开由 filename 参数指定的文件，并将其与一个文件流关联起来，此文件流被实现为指向结构 FILE 的指针。
mode 参数指定文件的打开方式，它是下列字符串中的一个：

模式	说明	模式	说明
"r"	只读	"r+"	打开文件进行读和写）
"w"	创建文件（写，截断文件）	"w+"	创建文件（读写，截断文件）
"a"	写（追加文件）	"a+"	打开文件进行读或追加

fread

fread 函数会从指定的文件流中，读取指定数量的指定长度的数据到指定的位置/缓冲区，其返回值为本次读取记录的个数。
fwrite 函数与 fread 相反（源和目的地反转）。

Example (fread、fwrite 函数原型)

```
#include <stdio.h>
size_t fread(const void *ptr, size_t size, size_t nitems,
                FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nitems,
                FILE *stream);
```

fclose

fclose 函数关闭指定的文件流，并使所有尚未写出的数据（一般是指缓冲区）写出。

Example (fclose 函数原型)

```
#include <stdio.h>
inf fclose(FILE *stream);
```

fflush

fflush 函数把指定的文件流所有尚未写出的数据（一般是指缓冲区）写出。

Example (fflush 函数原型)

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

fseek

fseek 函数为下一次的读写操作指定位置。

Example (fseek 函数原型)

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

offset 参数用来指定位置，whence 参数指定该偏移值的用法。

模式	说明
SEEK_SET	offset 是一个绝对位置
SEEK_CUR	offset 是相对于当前位置的一个相对值
SEEK_END	offset 是一个相对文件尾的一个相对值

fgetc,getc,getchar

fgetc 函数从文件流里取出下一个字节并把它做为一个字符返回，并向前移动位置。当它到达文件末尾或出现错误时，会返回 EOF，这时个需要通过 ferror 或 feof 来区分具体错误。
getc 函数作用基本与 fgetc 一样，但是不能使用 getc 的地址作为一个函数指针。fgetc 的优化比 getc 好。
getchar 函数相当于 getc(stdin)，它从标准输入里读取下一个字符。

Example (fegtc,getc,getchar 函数原型)

```
#include <stdio.h>

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

fputc,putc,putchar

fputc 函数把一个字符写到一个输出流中。它返回写入的值，若失败则返回 EOF。
putc 与 fputc 的关系类似于 getc 与 fgetc 的关系。
putchar 函数相当于 putc(c, stdout)。

Example (fputc,putc,putchar 函数原型)

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);
```

fgets,gets

fgets 函数从输入文件流里读取一个字符串，并写到指定的字符串里面去。结合下面的函数原型，每次最多传输 n-1 个字符，fgets 会向接收字符串里加上一个表示结尾的空字节 0。
gets 函数类似于 fgets，只不过它从标准输入读取数据并丢弃遇到的换行符。

Example (fgets,gets 函数原型)

```
#include <stdio.h>

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

printf,fprintf,sprintf

系统调用 read 的作用是从与文件描述符相关联的文件中读出前 nbyte 写入到缓冲区 buf 里，会返回实际读出的字节数，若返回 0 则表示读到了文件的末尾，返回-1 表示出现了错误。

Example (printf,fprintf,sprintf 函数原型)

这几个函数都会按照 format 格式指定的形式进行输出。
printf 会将自己的输出送到标准输出。
fprintf 把自己的输出送到一个指定的文件流。
sprintf 把自己的输出和一个结尾空字符写到一个字符串中。

```
#include <stdio.h>

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

转换控制符

格式	说明	格式	说明
%d %i	输出十进制数	%o %x	输出八、十六进制数
%c	输出一个字符	%s	输出一个字符串)
%f	输出单精度符点数	%g	输出双精度浮点数

Table: 常用转换控制符表

scanf,fscanf,sscanf

scanf 系列函数的工作方式与 printf 系列函数很相似，只是 scanf 系列的函数是用来把读到的数存入到一个指针参数指向的变量中。

Example (scanf,fscanf,sscanf 函数原型)

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

由于历史原因（有漏洞等），建议使用 fread 或 fgets 来读取输入。

实验：使用标准库函数拷贝文件

本实验通过使用标准 I/O 库里的函数来实现文件的拷贝。以下库函数及其函数原型供参考：

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
size_t fread(const void *ptr, size_t size,
             size_t nitems, FILE *stream);
size_t fwrite(const void *ptr, size_t size,
              size_t nitems, FILE *stream);
```

模式	说明	模式	说明
"r"	只读	"r+"	打开文件进行读和写）
"w"	创建文件（写，截断文件）	"w+"	创建文件（读写，截断文件）
"a"	写（追加文件）	"a+"	打开文件进行读或追加

Table: open 函数 mode 参数

实验：使用标准库函数拷贝文件

使用 getc,putc 函数进行读写。

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

模式	说明	模式	说明
"r"	只读	"r+"	打开文件进行读和写）
"w"	创建文件（写，截断文件）	"w+"	创建文件（读写，截断文件）
"a"	写（追加文件）	"a+"	打开文件进行读或追加

Table: open 函数 mode 参数

实验：使用标准库函数拷贝文件参考代码

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
int c;
FILE *in;
FILE *out;

in=fopen("simple_read.c","r");
out=fopen("io_copy_getcputc","w");
while((c=fgetc(in))!=EOF){
    fputc(c,out);
}
fclose(in);
fclose(out);
exit(0);
}
```

文件和目录的维护

标准库和系统调用为文件和目录的创建和维护提供了全面的支持。

常用的一些系统调用有：

1. chmod
2. chown
3. unlink link symlink
4. mkdir rmdir
5. chdir

常用的一些库函数有：

1. opendir closedir
2. readdir
3. telldir
4. seekdir

chmod

可以通过 chmod 系统调用来改变文件或目录的访问权限。这构成了 shell 程序 chmod 的基础。

Example (chmod 系统调用的原型)

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

这里的 mode 类似于 open 里的 mode。

chown

超级用户可以通过 chown 系统调用来改变文件或目录的属主。

Example (chown 系统调用的原型)

```
#include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
```

这里的 uid,gid 是使用的数字值。

unlink,link,symlink

可以用 unlink 系统调用来删除一个文件。unlink 系统调用删除一个文件的目录项并减少它的链接数，成功时返回 0，失败返回 -1 。前提是要拥有该文件所属目录的写和执行权限。

Example (unlink,link,symlink 系统调用的原型)

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

link 是用来创建文件链接的（硬 ln），symlink 是用来创建符号链接的(软 ln -s)。

mkdir,rmdir

用来创建或删除目录的系统调用。

Example (mkdir,rmdir 系统调用的原型)

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

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

使用 rmdir 时要注意，目录为空时才可以。

chdir, getcwd

系统调用 chdir 可以用来功换目录。getcwd 函数可以用来确定自己当前的工作目录。

Example (chdir, getcwd 的原型)

```
#include <unistd.h>
int chdir(const char *path);


char *getcwd(char *buf, size_t size);
```

getcwd 将当前目录名写到缓冲区 buf 中，若目录名的长度超过了 size，则返回 NULL，否则返回指针 buf。

目录扫描

与文件操作中的文件流（FILE *）结构类似，目录流被封装成了（DIR *）结构。目录项本身被封装在 dirent 结构中。
dirent 结构中包含的目录项内容包括以下部分：

- ino_t d_ino
  文件的 inode 号
- char d_name[]
  文件的名字

opendir 函数

opendir 函数的作用是打开并建立一个目录。

Example (opendir 函数原型)

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

DIR *opendir(const char *name);
```

readdir 函数

readdir 函数将返回一个指针，指针指向的结构里保存着目录流中下一个目录项的有关资料。

Example (readdir 函数原型)

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

telldir 函数

telldir 函数返回值记录着一个目录流里的当前位置。

Example (telldir 函数原型)

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

long int telldir(DIR *dirp);
```

seekdir 函数

seekdir 函数返的作用是设置目录流 dirp 的目录项指针，loc 用来设置指针的位置，loc 应该通过 telldir 函数获得。

Example (seekdir 函数原型)

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

void seekdir(DIR *dirp, long int loc);
```

closedir 函数

closedir 函数关闭一个目录流，并释放与之关联的资源。其执行成功返回0, 否则返回-1.

Example (closedir 函数原型)

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

int closedir(DIR *dirp);
```

实验：实现目录列表功能

使用目录操作相关的函数、系统调用来实现目录列表功能。
要求：

- 每个目录或文件独占一行
- 每个目录名字后面加上一个"/" 符号
- 每个目录下面的文件或子目录比目录缩进 4 个空格

/proc 文件系统

Linux 将一切事物看作文件，硬件设备在文件系统中也有相应的条目。/dev 目录中的文件使用底层系统调用这种特殊方式来访问硬件。要访问硬件必然要通过驱动程序，而为了访问驱动程序，Linux 提供了一个特殊的文件系统 procs，其通常表现为/proc 目录。该目录包含了许多特殊文件，以允许对驱动和内核信息进行高层访问。
比较常用的一些文件有：

1. cat /proc/cpuinfo
   查看 cpu 信息
2. cat /proc/meminfo
   查看内存信息
3. cat /proc/sys/fs/file-max
   查看内核允许程序同时打开多少个文件

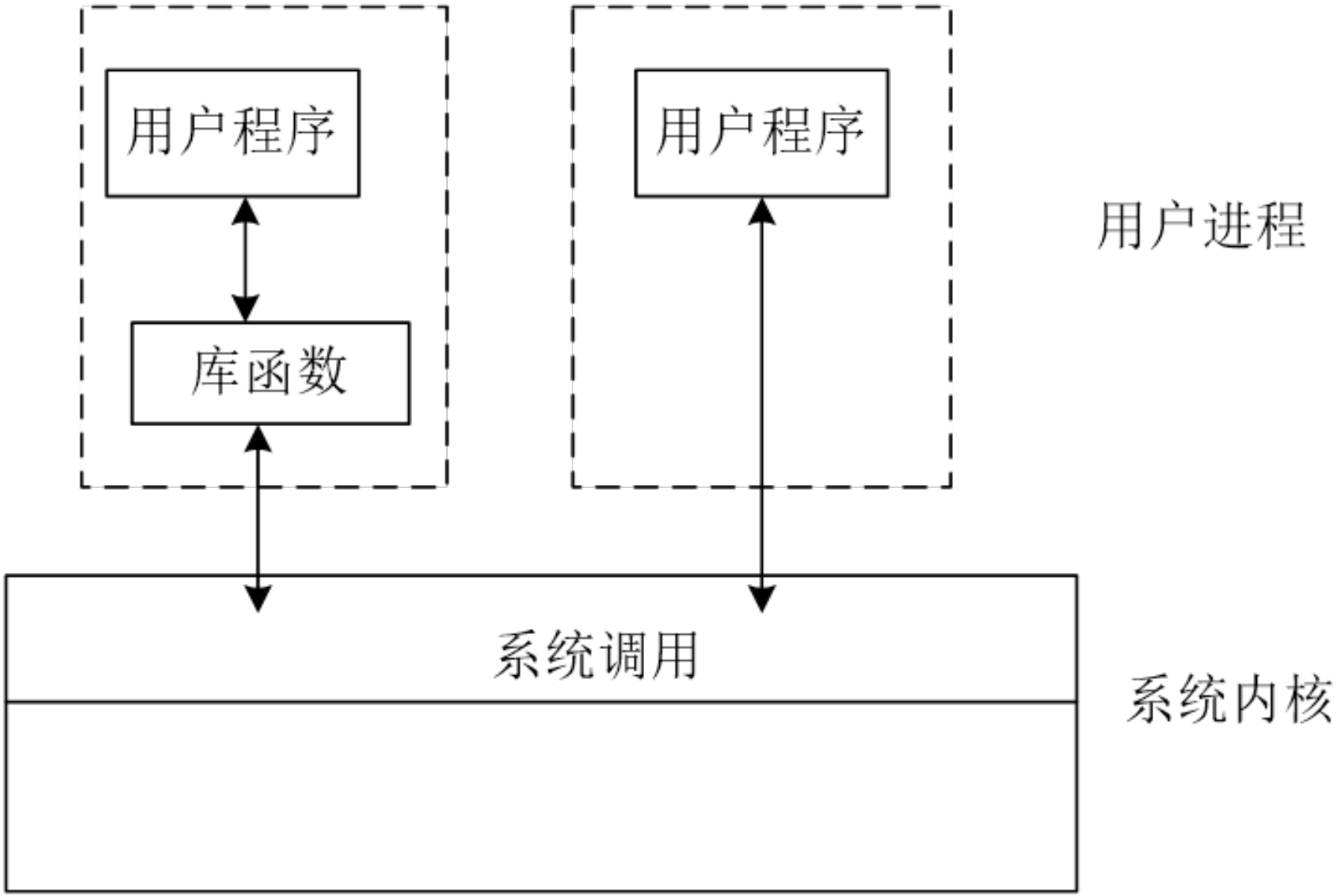
作业

完成以下作业：

1. 文件拷贝
2. 目录列表

The End

用户程序 vs 库函数 vs 系统调用



about man page

The manual is generally split into eight numbered sections, organized as follows (on Research Unix, BSD, macOS and Linux):

section	description
1	General commands
2	System calls
3	Library function(C standard library)
4	Special files(devices) and drivers
5	File formats and conventions
6	Games and screensavers
7	Miscellanea
8	System administration commands and daemons

Table: man page

在终端中运行 man read 与 man 2 read ， 观察其输出的区别。