File

张海宁

贵州大学

hnzhang1@gzu.edu.cn

April 10, 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：

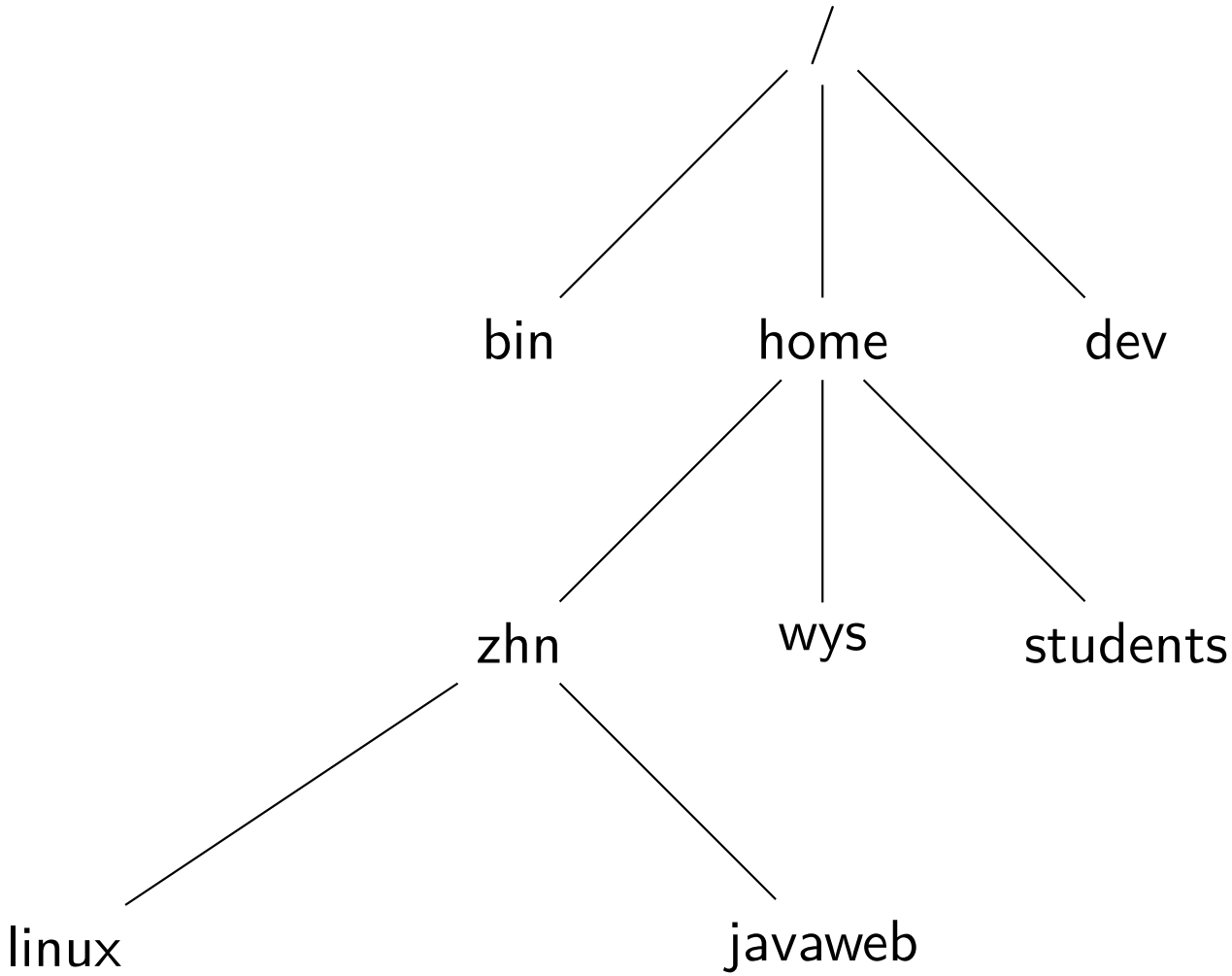
1. 目录
   特殊的文件
2. 文件和设备

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，一般有很 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
5. ioctl

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1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。

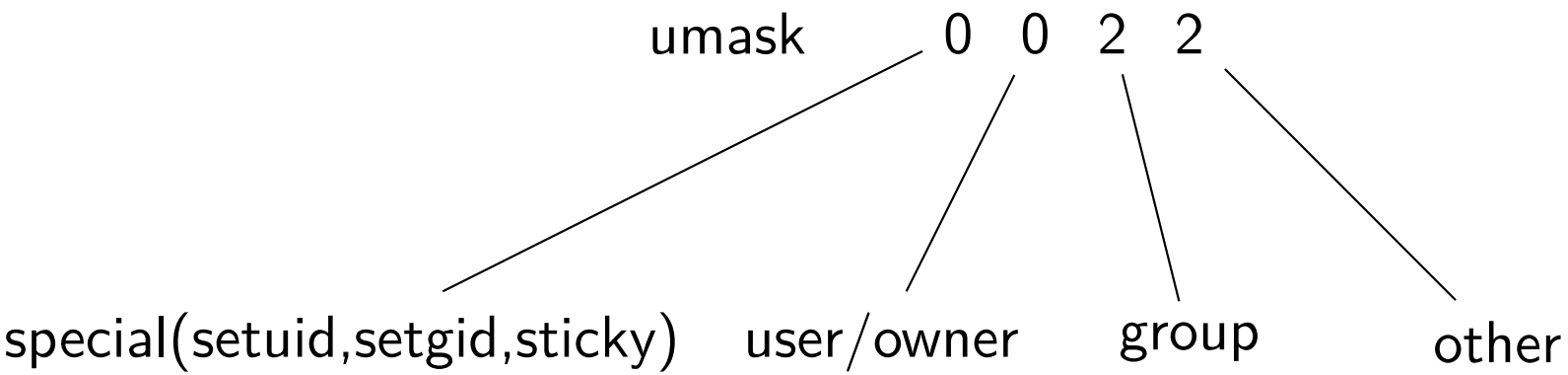
```
$ umask             # display current value (as octal)
0022
$ umask -S          # display current value symbolically
u=rwx,g=rx,o=rx
```

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 /textcolorredrun 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。

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

fopen

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);
```

fread

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);
```

fwrite

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);
```

fclose

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);
```

fflush

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);
```

fseek

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);
```

fgetc,getc,getchar

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);
```

fputc,putc,putchar

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);
```

fgets,gets

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);
```

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 (read 系统调用的原型)

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

scanf,fscanf,sscanf

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);
```

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。

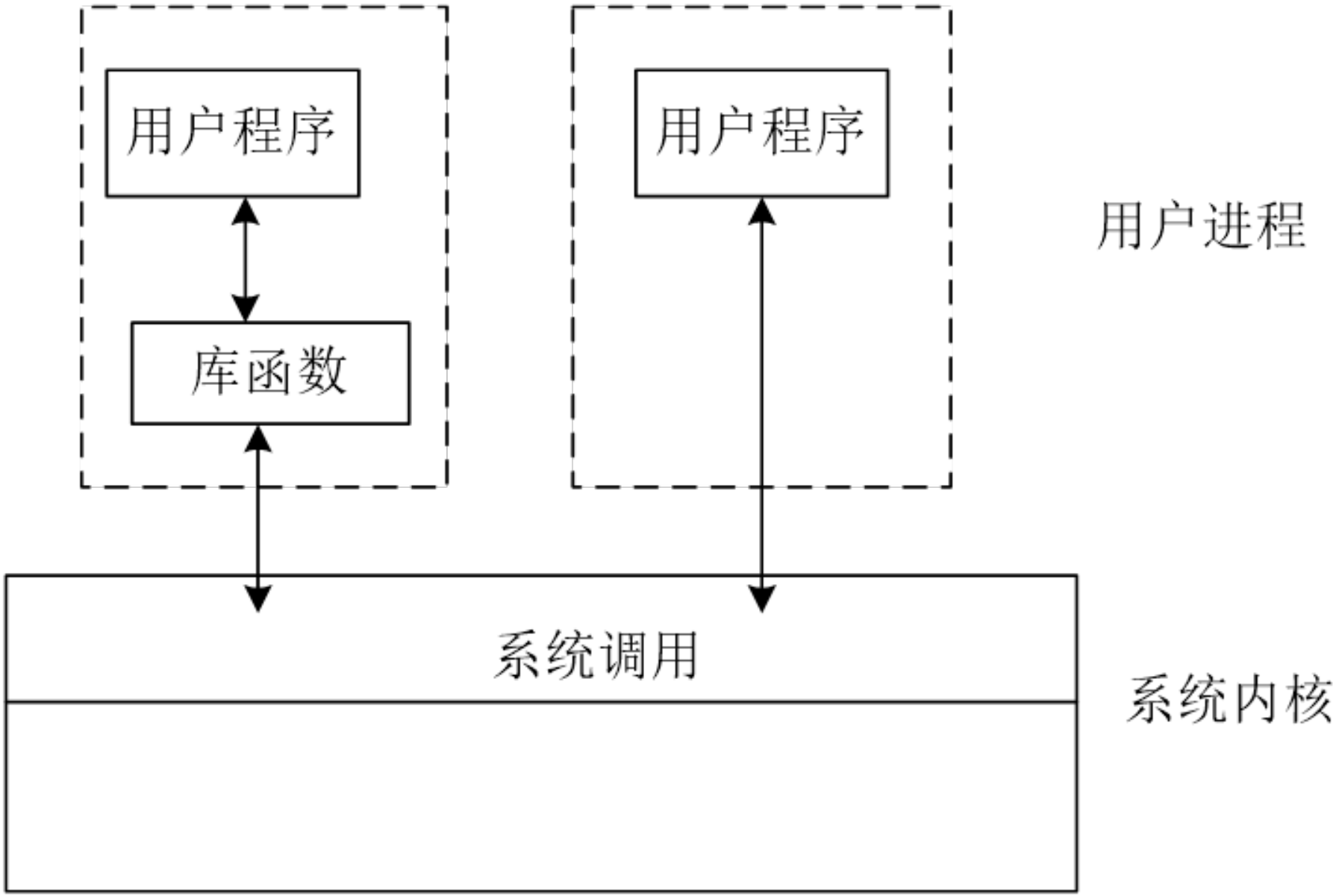
常用的一些系统调用有：

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

用户程序 vs 库函数 vs 系统调用



作业

按以下要求进行编程练习：

1. 2 个 cpp 文件：prog.cpp，aux.cpp
2. 1 个 h 文件：aux.h
3. aux.h 头文件定义函数 Max 和 Min，它们分别计算四个数（参数）的最大值和最小值；aux.cpp 实现这两个函数
4. prog.c 中定义主函数，循环输入 4 个随机数，输出他们的最大和最小值
5. 编译该项目，调试、跟踪程序执行过程；并在控制台界面运行编译的可执行文件。
6. 编写类似功能的 c 语言程序也可。

The End