学习目标

1. 掌握/open/read/write/lseek/close函数的使用
2. 掌握stat/lstat函数的使用
3. 掌握目录遍历相关函数的使用
4. 掌握dup、dup2函数的使用
5. 掌握fcntl函数的使用

文件IO

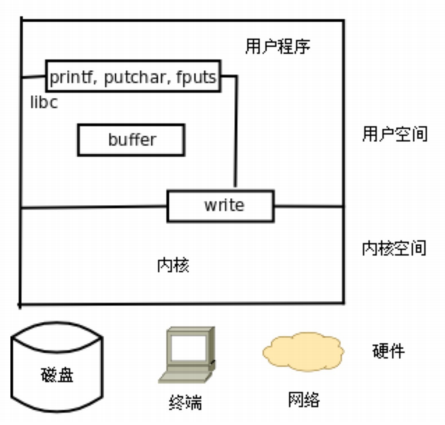
从本章开始学习各种Linux系统函数,这些函数的用法必须结合Linux内核的工作原理来理解, 因为系统函数正是内核提供给应用程序的接口, 而要理解内核的工作原理,必须熟练掌握C语言, 因为内核也是用C语言写的, 我们在描述内核工作原理时必然要用“指针”、“结构体”、“链表”这些名词来组织语言, 就像只有掌握了英语才能看懂英文书一样, 只有学好了C语言才能看懂我描述的内核工作原理。

C标准函数与系统函数的区别

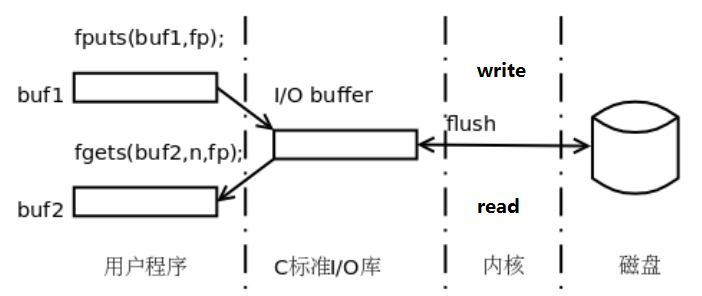
什么是系统调用

由操作系统实现并提供给外部应用程序的编程接口。(Application Programming Interface，API)。是应用程序同系统之间数据交互的桥梁。

一个helloworld如何打印到屏幕。



每一个FILE文件流（标准C库函数）都有一个缓冲区buffer，默认大小8192Byte。Linux系统的IO函数默认是没有缓冲区.



open/close

文件描述符

一个进程启动之后，默认打开三个文件描述符：

#define STDIN\_FILENO 0

#define STDOUT\_FILENO 1

#define 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, 调用open函数可以打开或创建一个文件, 得到一个文件描述符.**

open函数

* 函数描述: 打开或者新建一个文件
* 函数原型:

int open(const char \*pathname, int flags);

int open(const char \*pathname, int flags, mode\_t mode);

* 函数参数：
* 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, 这里只介绍几个常用选项：
    - 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。
      * **文件最终权限：mode & ~umask**
    - 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, 将其长度截断为为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(NonblockI/O),非阻塞I/O。
* **函数返回值:**
* 成功: 返回一个最小且未被占用的文件描述符
* 失败: 返回-1, 并设置errno值.

close函数

* 函数描述: 关闭文件
* 函数原型: int close(int fd);
* 函数参数: fd文件描述符
* 函数返回值:
* 成功返回0
* 失败返回-1, 并设置errno值.

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。

read/write

read函数

* 函数描述: 从打开的设备或文件中读取数据
* 函数原型: ssize\_t read(int fd, void \*buf, size\_t count);
* 函数参数:
* fd: 文件描述符
* buf: 读上来的数据保存在缓冲区buf中
* count: buf缓冲区存放的最大字节数
* 函数返回值:
  + >0：读取到的字节数
  + =0：文件读取完毕
  + -1： 出错，并设置errno

write

* 函数描述: 向打开的设备或文件中写数据
* 函数原型: ssize\_t write(int fd, const void \*buf, size\_t count);
* 函数参数：
  + fd：文件描述符
  + buf：缓冲区，要写入文件或设备的数据
  + count：buf中数据的长度
* 函数返回值:
  + 成功：返回写入的字节数
  + 错误：返回-1并设置errno

lseek

所有打开的文件都有一个当前文件偏移量(current file offset),以下简称为cfo. cfo通常是一个非负整数, 用于表明文件开始处到文件当前位置的字节数. 读写操作通常开始于 cfo, 并且使 cfo 增大, 增量为读写的字节数. 文件被打开时, cfo 会被初始化为 0, 除非使用了 O\_APPEND.

使用 lseek 函数可以改变文件的 cfo.

#include <sys/types.h>

#include <unistd.h>

off\_t lseek(int fd, off\_t offset, int whence);

* 函数描述: 移动文件指针
* 函数原型: off\_t lseek(int fd, off\_t offset, int whence);
* 函数参数：
  + fd：文件描述符
  + 参数 offset 的含义取决于参数 whence：
    - 如果 whence 是 SEEK\_SET，文件偏移量将设置为 offset。
    - 如果 whence 是 SEEK\_CUR，文件偏移量将被设置为 cfo 加上 offset，offset 可以为正也可以为负。
    - 如果 whence 是 SEEK\_END，文件偏移量将被设置为文件长度加上 offset，offset 可以为正也可以为负。
* 函数返回值: 若lseek成功执行, 则返回新的偏移量。
* lseek函数常用操作
* 文件指针移动到头部

lseek(fd, 0, SEEK\_SET);

* 获取文件指针当前位置

int len = lseek(fd, 0, SEEK\_CUR);

* 获取文件长度

int len = lseek(fd, 0, SEEK\_END);

* lseek实现文件拓展

off\_t currpos;

// 从文件尾部开始向后拓展1000个字节

currpos = lseek(fd, 1000, SEEK\_END);

// 额外执行一次写操作，否则文件无法完成拓展

write(fd, “a”, 1); // 数据随便写

练习:

1 编写简单的IO函数读写文件的代码

2 使用lseek函数获取文件大小

3 使用lseek函数实现文件拓展

perror和errno

errno是一个全局变量, 当系统调用后若出错会将errno进行设置, perror可以将errno对应的描述信息打印出来.

如:perror("open"); 如果报错的话打印: open:(空格)错误信息

练习:编写简单的例子, 测试perror和errno.

阻塞和非阻塞:

思考: 阻塞和非阻塞是文件的属性还是read函数的属性?

* 普通文件：hello.c
  + 默认是非阻塞的
* 终端设备：如 /dev/tty
* 默认阻塞
* 管道和套接字
  + 默认阻塞

练习:

1 测试普通文件是阻塞还是非阻塞的?

2 测试终端设备文件/dev/tty是阻塞还是非阻塞的.

得出结论: 阻塞和非阻塞是文件本身的属性, 不是read函数的属性.

文件和目录

文件操作相关函数

stat/lstat函数

* 函数描述: 获取文件属性
* 函数原型: int stat(const char \*pathname, struct stat \*buf);

int lstat(const char \*pathname, struct stat \*buf);

* 函数返回值：
* 成功返回 0
* 失败返回 -1

struct stat {

dev\_t st\_dev; //文件的设备编号

ino\_t st\_ino; //节点

mode\_t st\_mode; //文件的类型和存取的权限

nlink\_t st\_nlink; //连到该文件的硬连接数目，刚建立的文件值为1

uid\_t st\_uid; //用户ID

gid\_t st\_gid; //组ID

dev\_t st\_rdev; //(设备类型)若此文件为设备文件，则为其设备编号

off\_t st\_size; //文件字节数(文件大小)

blksize\_t st\_blksize; //块大小(文件系统的I/O 缓冲区大小)

blkcnt\_t st\_blocks; //块数

time\_t st\_atime; //最后一次访问时间

time\_t st\_mtime; //最后一次修改时间

time\_t st\_ctime; //最后一次改变时间(指属性)

};

- st\_mode -- 16位整数

○ 0-2 bit -- 其他人权限

S\_IROTH 00004 读权限

S\_IWOTH 00002 写权限

S\_IXOTH 00001 执行权限

S\_IRWXO 00007 掩码, 过滤 st\_mode中除其他人权限以外的信息

○ 3-5 bit -- 所属组权限

S\_IRGRP 00040 读权限

S\_IWGRP 00020 写权限

S\_IXGRP 00010 执行权限

S\_IRWXG 00070 掩码, 过滤 st\_mode中除所属组权限以外的信息

○ 6-8 bit -- 文件所有者权限

S\_IRUSR 00400 读权限

S\_IWUSR 00200 写权限

S\_IXUSR 00100 执行权限

S\_IRWXU 00700 掩码, 过滤 st\_mode中除文件所有者权限以外的信息

If (st\_mode & S\_IRUSR) -----为真表明可读

If (st\_mode & S\_IWUSR) ------为真表明可写

If (st\_mode & S\_IXUSR) ------为真表明可执行

○ 12-15 bit -- 文件类型

S\_IFSOCK 0140000 套接字

S\_IFLNK 0120000 符号链接（软链接）

S\_IFREG 0100000 普通文件

S\_IFBLK 0060000 块设备

S\_IFDIR 0040000 目录

S\_IFCHR 0020000 字符设备

S\_IFIFO 0010000 管道

S\_IFMT 0170000 掩码,过滤 st\_mode中除文件类型以外的信息

If ((st\_mode & S\_IFMT)==S\_IFREG) ----为真普通文件

if(S\_ISREG(st\_mode)) ------为真表示普通文件

if(S\_ISDIR(st.st\_mode)) ------为真表示目录文件

stat函数和lstat函数的区别

* 对于普通文件, 这两个函数没有区别, 是一样的.
* 对于连接文件,调用lstat函数获取的是链接文件本身的属性信息;

而stat函数获取的是链接文件指向的文件的属性信息.

练习:

1 stat函数获取文件大小

2 stat函数获取文件类型和文件权限

3 lstat函数获取连接文件的属性(文件大小)

目录操作相关函数

opendir函数

* 函数描述:打开一个目录
* 函数原型: DIR \*opendir(const char \*name);
* 函数返回值: 指向目录的指针
* 函数参数: 要遍历的目录(相对路径或者绝对路径)

readdir函数

* 函数描述: 读取目录内容--目录项
* 函数原型: struct dirent \*readdir(DIR \*dirp);
* 函数返回值: 读取的目录项指针
* 函数参数: opendir函数的返回值

struct dirent

{

ino\_t d\_ino; // 此目录进入点的inode

off\_t d\_off; // 目录文件开头至此目录进入点的位移

signed short int d\_reclen; // d\_name 的长度, 不包含NULL 字符

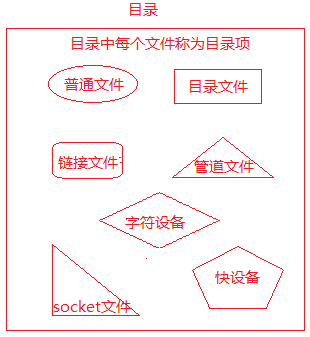
unsigned char d\_type; // d\_name 所指的文件类型

char d\_name[256]; // 文件名

};

d\_type的取值:

* DT\_BLK - 块设备
* DT\_CHR - 字符设备
* DT\_DIR - 目录
* DT\_LNK - 软连接
* DT\_FIFO - 管道
* DT\_REG - 普通文件
* DT\_SOCK - 套接字
* DT\_UNKNOWN - 未知



closedir函数

* 函数描述: 关闭目录
* 函数原型: int closedir(DIR \*dirp);
* 函数返回值: 成功返回0, 失败返回-1
* 函数参数: opendir函数的返回值

读取目录内容的一般步骤

1 DIR \*pDir = opendir(“dir”); //打开目录

2 while((p=readdir(pDir))!=NULL){} //循环读取文件

3 closedir(pDir); //关闭目录

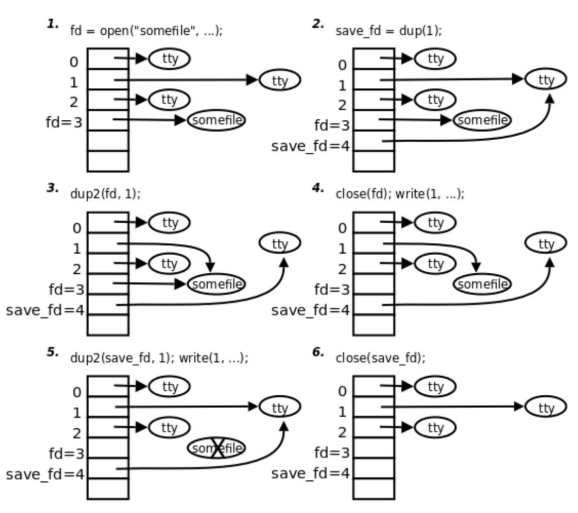
练习

1 遍历指定目录下的所有文件, 并判断文件类型.

2 递归遍历目录下所有的文件, 并判断文件类型.

特别注意: 递归遍历指定目录下的所有文件的时候, 要过滤掉.和..文件, 否则会进入死循环

dup/dup2/fcntl



dup函数

* 函数描述: 复制文件描述符
* 函数原型: int dup(int oldfd);
* 函数参数: oldfd -要复制的文件描述符
* 函数返回值:
* 成功: 返回最小且没被占用的文件描述符
* 失败: 返回-1, 设置errno值

练习: 编写程序, 测试dup函数.

dup2函数

* 函数描述: 复制文件描述符
* 函数原型: int dup2(int oldfd, int newfd);
* 函数参数:
* oldfd-原来的文件描述符
* newfd-复制成的新的文件描述符
* 函数返回值:
  + 成功: 将oldfd复制给newfd, 两个文件描述符指向同一个文件
  + 失败: 返回-1, 设置errno值
* 假设newfd已经指向了一个文件，首先close原来打开的文件，然后newfd指向oldfd指向的文件.

若newfd没有被占用，newfd指向oldfd指向的文件.

练习:

1编写程序, 测试dup2函数实现文件描述符的复制.

2 编写程序, 完成终端标准输出重定向到文件中

fcntl函数

* 函数描述: 改变已经打开的文件的属性
* 函数原型: int fcntl(int fd, int cmd, ... /\* arg \*/ );
* 若cmd为F\_DUPFD, 复制文件描述符, 与dup相同
* 若cmd为F\_GETFL, 获取文件描述符的flag属性值
* 若cmd为 F\_SETFL, 设置文件描述符的flag属性
* 函数返回值:返回值取决于cmd
* 成功
* 若cmd为F\_DUPFD, 返回一个新的文件描述符
* 若cmd为F\_GETFL, 返回文件描述符的flags值
* 若cmd为 F\_SETFL, 返回0
* 失败返回-1, 并设置errno值.
  + fcntl函数常用的操作:

1 复制一个新的文件描述符:

int newfd = fcntl(fd, F\_DUPFD, 0);

2 获取文件的属性标志

int flag = fcntl(fd, F\_GETFL, 0)

3 设置文件状态标志

flag = flag | O\_APPEND;

fcntl(fd, F\_SETFL, flag)

4 常用的属性标志

O\_APPEND-----设置文件打开为末尾添加

O\_NONBLOCK-----设置打开的文件描述符为非阻塞

练习:

1 使用fcntl函数实现复制文件描述符

2 使用fcntl函数设置在打开的文件末尾添加内容.