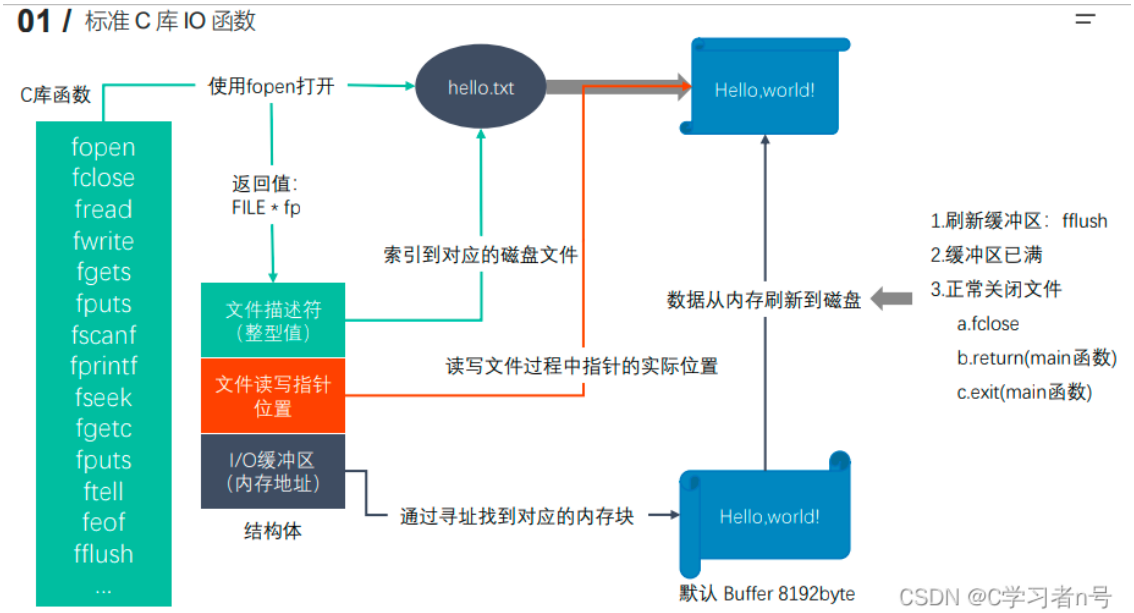
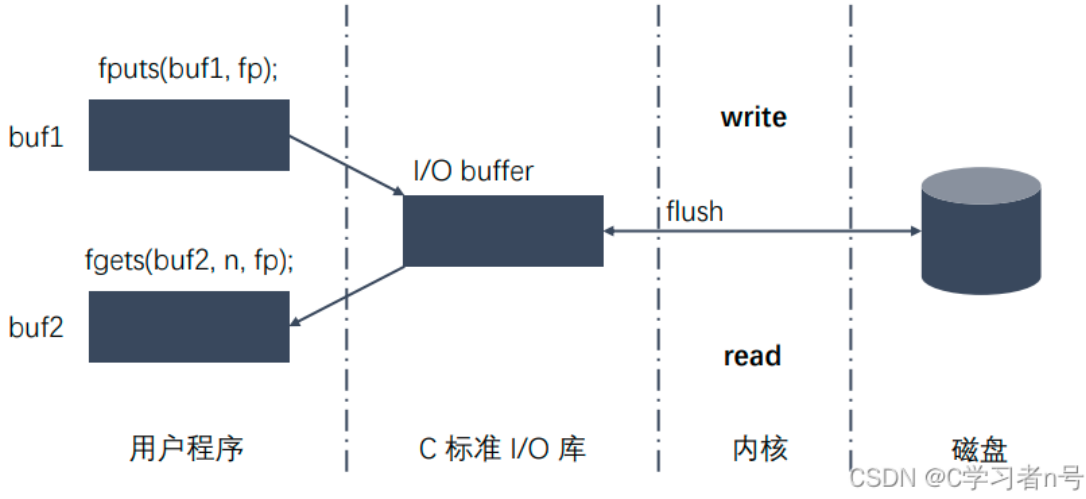
Linux的IO函数

预备知识

1. 标准c库的文件IO



2. 标准c库IO和Linux系统调用IO关系

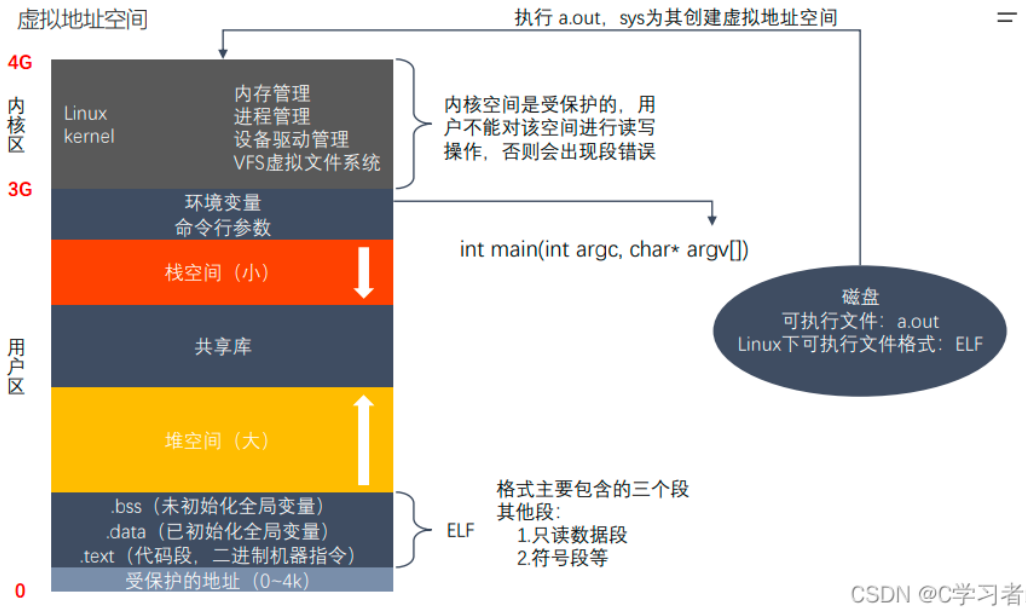


3. 虚拟地址空间

多任务操作系统中，每个进程都运行在属于自己的内存沙盘中，这个沙盘就是虚拟地址空间（virtual address space）。虚拟地址空间由内核空间（kernel space）和用户模式空间（user mode space）两部分组成。

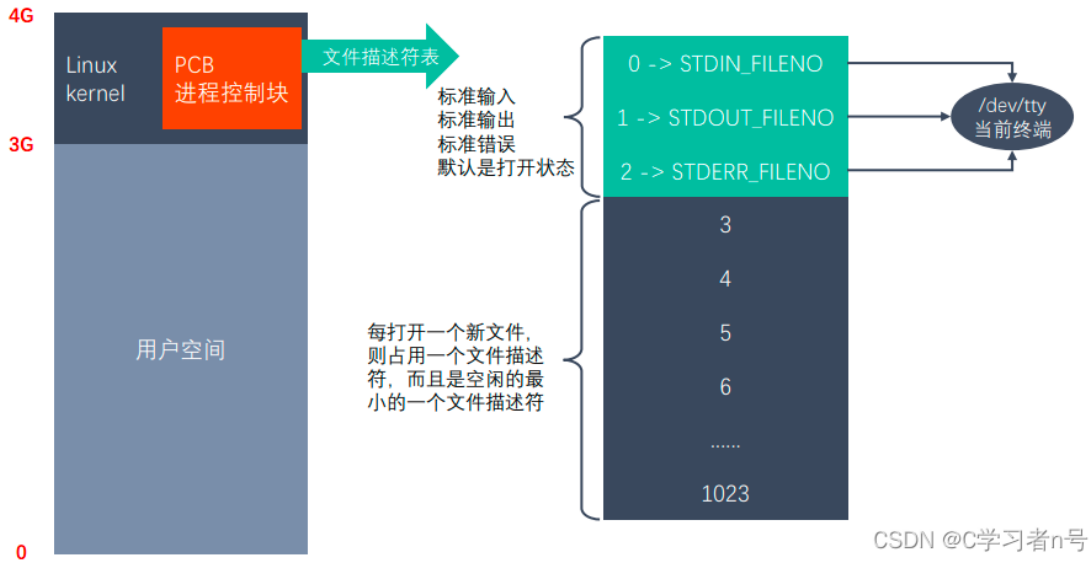
虚拟地址会通过页表（page table）映射到物理内存，页表由操作系统维护并被处理器引用，每个

进程都有自己的页表。



4. 文件描述符

   内存管理块中，有一个进程控制块PCB，每一个进程都通过PCB来管理，他记录了每一个进程的id，每一个进程都有一个文件描述符表，这个表记录了该进程打开的所有文件，默认情况下，每个进程的文件描述符表有三个文件描述符，对应的是0,1，2，分别表示 标准输入、标准输出和标准错误，默认是打开的状态。所以当在自己的程序中打开一个文件，如果成功，那么他的描述符是3.



5. linux命令--> man

   man是一个操作手册命令

> 1、man + 命令 --> 查看系统标准命令
> 2、man + 2 + xxx --> 查看linux的系统调用，比如open函数
> 3、man + 3 + xxx --> 查看库函数，比如标准c库，fopen函数

open函数和close函数

1. 函数原型

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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

```c
#include <unistd.h>

int close(int fd);  //关闭指定的文件描述符指向的文件
```

2. 详解

> 1、pathname --> 要打开的文件名，包括路径
> 2、flags --> 对文件的操作权限，参数是由一个或者多个标志位按位或组合而成。最基本的支持三种访问模式，O_RDONLY，O_WRONLY，O_RDWR，分别表示只读、只写和读写，且三个互斥。还有其他的设置。flags参数是一个int类型的数据，占4个字节，32位。每一位就是一个标志位，代表了一种权限，通过按位或运算可以将其他的权限合并在一起。
> 3、返回值 --> 返回一个新的文件描述符，如果调用失败返回-1.此时要涉及到一个errno错误编号变量。当调用失败时，系统会errno一个值，这个值在系统中对应了错误的解释信息。
> 4、errno错误编号，属于Linux系统函数库，库里面的一个全局变量，记录的是最近的错误号。
> 5、void perror(const char *s); 该函数在`#include <stdio.h>`标准c库中，作用是用于打印errno对应的错误描述。参数s是用户描述，比如s="出现错误"，最终输出的内容是`出现错误:xxxxx`
> 6、mode_t ---> 八进制的数，也定义了对应的宏，方便使用，通过按位或运算指定多个权限。该参数是在需要创建文件的时候使用的，用于指定创建出来的文件对所有者、组用户和所有用户的读写和执行权限的设定。但是，实际的权限并不完全是由输入的mode参数所指定的，具体内部操作是: `mode & (~umask)`，得出的结果就是实际上的创建出来的文件对用户的权限。其中umask是一个进程级别的属性，在shell中输入umask就可以查看这个值，当然也可以自定义设置，一般使用默认的，作用就是抹去输入的部分权限。比如输入的mode参数是0666，而系统中的umask是022，那么最终文件的权限就是0644.

3. open函数使用

两个open函数的功能不一样，第一个函数是在指定的文件存在时调用的，而第二个函数更多的是创建一个新的文件的功能，在flags参数中有一个选项是O_CREAT表示在没有文件的情况下就创建文件，此时需要指定第三个参数mode。下面详细介绍open的两种比较常见的情况：

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

int main()
{
    //第一个open函数原型的使用，v.txt文件已经存在，不需要创建，因此也不需要第三个参数
    int fd = open("v.txt", O_RDONLY);

    if(fd == -1)
    {
        perror("错误");
    }

    close(fd);  //关闭文件描述符指定的文件,并归还文件描述符
```

```
    return 0;
}
```

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

int main()
{
    //第二种函数原型，创建不存在的文件，且指定用户权限
    int fd = open("v.txt", O_WRONLY|O_CREAT|O_TRUNC, 0777);

    if(fd == -1)
    {
        perror("错误");
    }

    close(fd);

    return 0;
}
```

4. open函数flags和mode参数详解

> flags参数，除了只读、只写和读写三种模式，还有以下模式与三种模式搭配使用
> 1、O_APPEND 文本将以追加的模式打开，每次写操作之前，将会更新文件的位置指针，指向文件末尾。
> 2、O_CREAT 当参数pathname指定的文件不存在时，内核自动创建这个文件，如果文件已经存在，那么该标志位就无效。
> 3、O_TRUNC 如果文件存在且是普通文件，并且有写权限，该标志位会把文件长度截断为0，文件中的内容直接删除掉。对于截断操作，需要提供写权限操作。如果文件不存在，且没有指定O_CREAT，那么open函数就会调用失败。

> mode参数，只有当flags指定了O_CREAT时才需要使用。下面四队一组分别表示所有者、组用户、其他用户的读、写和执行以及三个权限合并的情况：
> 1、S_IRWXU 00700 user (file owner) has read, write, and execute permission
> 2、S_IRUSR 00400 user has read permission
> 3、S_IWUSR 00200 user has write permission
> 4、S_IXUSR 00100 user has execute permission
> 5、S_IRWXG 00070 group has read, write, and execute permission
> 6、S_IRGRP 00040 group has read permission
> 7、S_IWGRP 00020 group has write permission
> 8、S_IXGRP 00010 group has execute permission
> 9、S_IRWXO 00007 others have read, write, and execute permission
> 10、S_IROTH 00004 others have read permission
> 11、S_IWOTH 00002 others have write permission
> 12、S_IXOTH 00001 others have execute permission

5. 一个创建文件的函数

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

int creat(const char *pathname, mode_t mode);
```

这个函数的功能是创建文件，本质还是open函数，对应的open函数是：

```
int creat(const char * pathname, mode_t mode)
{
    return open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
}
```

所以，creat函数的功能也是打开文件，且是在文件不存在的时候就创建新的文件；如果存在就打开文件。

read和write函数

1. 函数原型

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

注：返回值类型ssize_t是有符号的size_t。

> read函数的参数和返回值
> 参数：
> - fd：文件描述符，open得到的，通过这个文件描述符操作某个文件
> - buf：需要读取数据存放的地方，数组的地址（传出参数）
> - count：指定的参数数组的大小
>
> 返回值：
> - 成功：
>
> 大于0: 返回实际的读取到的字节数
> 等于0：文件已经读取完了
> - 失败：-1 ，并且设置errno

> 参数：
> - fd：文件描述符，open得到的，通过这个文件描述符操作某个文件
> - buf：要往磁盘写入的数据，数据
> - count：要写的数据的实际的大小
>
> 返回值：
> 成功：实际写入的字节数
> 失败：返回-1，并设置errno

2. 读写实现复制功能

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

```c
int main()
{
    //1、打开一个文件，以读的模式打开
    int fd_src = open("v.txt", O_RDONLY);
    if(fd_src == -1)
    {
        perror("open");
        return -1;
    }

    //2、创建一个新的文件，用于写入读取到的数据
    int fd_target = open("c.txt", O_WRONLY | O_CREAT | O_TRUNC, 0777);
    if(fd_target == -1)
    {
        perror("creat");
        return -1;
    }

    //3、读写复制
    ssize_t read_len = 0;
    char buf[1024] = {0};
    while ((read_len = read(fd_src, buf, sizeof(buf))) > 0) //读取到的长度大于0就拷贝到新的文件中
    {
        write(fd_target, buf, read_len);
    }

    //4、关闭文件
    close(fd_target);
    close(fd_src);

    return 0;
}
```

3. 添加写入文本

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

int main()
{
    int fd_target = open("c.txt", O_WRONLY | O_APPEND);
    if(fd_target == -1)
    {
        perror("open");
        return -1;
    }

    //计算字节数，适用于中文和英文数字等。
    void * app_context = "\n这是一个文件，添加到文件末尾!!!";
    write(fd_target, app_context, strlen(app_context) * sizeof(void));
```

```
    close(fd_target);

    return 0;
}
```

lseek函数

1. 函数原型

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

off_t lseek(int fd, off_t offset, int whence);
```

```
/*
参数：
            - fd：文件描述符，通过open得到的，通过这个fd操作某个文件
            - offset：偏移量
            - whence:
                SEEK_SET
                    设置文件指针的偏移量，即offset指定的偏移大小
                SEEK_CUR
                    设置偏移量：当前位置 + 第二个参数offset的值
                SEEK_END
                    设置偏移量：文件大小 + 第二个参数offset的值
        返回值：返回文件指针的位置，单位是子节


    作用：
        1.移动文件指针到文件头
        lseek(fd, 0, SEEK_SET);

        2.获取当前文件指针的位置
        lseek(fd, 0, SEEK_CUR);

        3.获取文件长度
        lseek(fd, 0, SEEK_END);

        4.拓展文件的长度，当前文件10b，110b，增加了100个字节
        lseek(fd, 100, SEEK_END)
        注意：需要写一次数据，否则磁盘上的文件的实际大小还是原来的大小
*/
```

2. 代码测试

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

int main()
{
```

```
    int fd = open("c.txt", O_WRONLY);
    if(fd == -1)
    {
        perror("open");
        return -1;
    }
    //1、获取文件的长度
    off_t size_file = lseek(fd, 0, SEEK_END);

    //2、获取当前文件指针位置
    lseek(fd, 110, SEEK_SET);   //设置文件指针到110位置，返回值就是110
    off_t pos_cur = lseek(fd, 0, SEEK_CUR);  //获取文件指针位置

    printf("文件大小 = %lu\n", size_file);
    printf("文件指针当前位置 = %lu\n", pos_cur);

    //3、扩展文件长度
    off_t t1 = lseek(fd, 0, SEEK_SET);      //将文件指针设置到文件头处，返回值是0，还原上面的移动操作
    off_t t2 = lseek(fd, 100, SEEK_END);    //扩展文件100子节大小，返回值是(文件大小 + 110)

    //上面通过lseek扩展之后，此时的文件指针就指向了扩展后的最某端，此时需要插入一次数据才能让磁盘上的文件真正的大小改变
    void * buf = "必须要插入的数据";
    write(fd, buf, strlen(buf) * sizeof(buf));

    close(fd);

    return 0;
}
```

stat、lstat和fstat函数

1. 函数原型

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

int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
```

2. stat函数参数说明

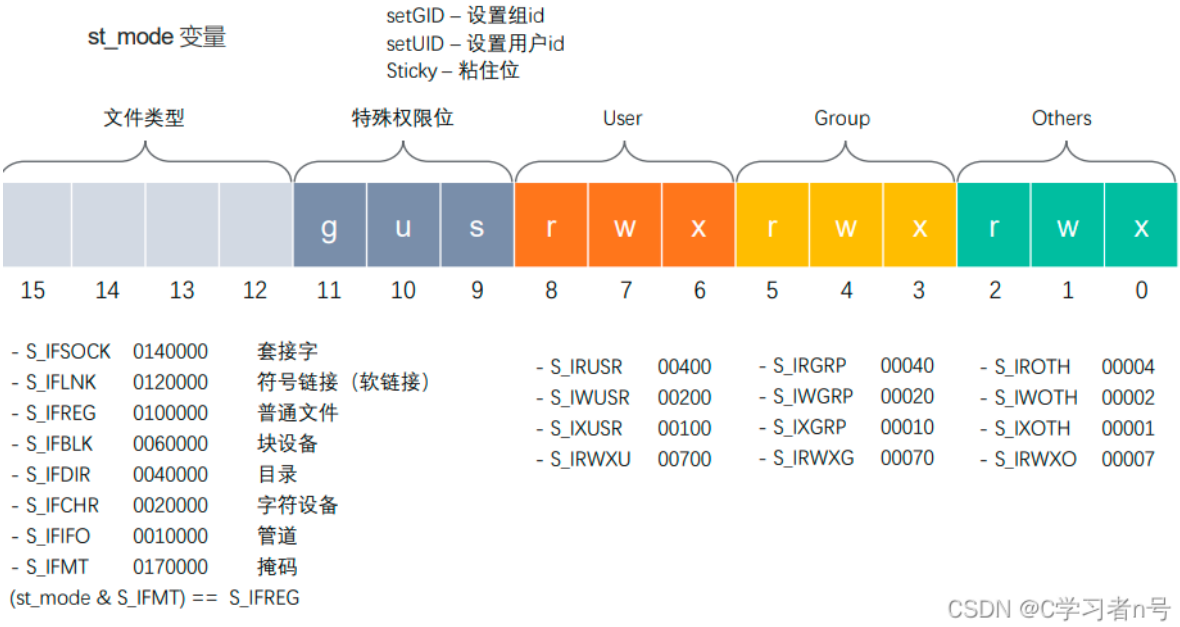
> 作用：获取一个文件相关的一些信息，通过第二个结构体类型参数传出
> 参数:
> - pathname：操作的文件的路径及文件名
> - statbuf：结构体变量，传出参数，用于保存获取到的文件的信息
> 返回值:
> 成功：返回0
> 失败：返回-1 设置errno

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

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

int main()
{
    //stat函数
    struct stat stat_out;
    int ret = stat("./v.txt", &stat_out);

    if(ret == -1)
    {
        perror("stat");
        return -1;
    }

    //根据传出参数获取结构体中的信息
    int size = stat_out.st_size;    //文件大小
    printf("%d\n", size);

    return 0;
}
```

3. struct stat 结构体参数类型

```c
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
    uid_t st_uid;           // 用户ID
    gid_t st_gid;           // 组ID
    dev_t st_rdev;          // 设备文件的设备编号
    off_t st_size;          // 文件字节数(文件大小)
    blksize_t st_blksize;   // 块大小
    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)
};
```



4. lstat函数和fstat函数

第一、相对于stat函数，fstat函数的功能和stat函数一样，只是stat函数是通过文件名来指定第一个参数；而fstat函数则是通过文件描述符来指定第一个参数。
第二、如果一个文件被指定为一个链接文件，那么通过stat函数获取这个文件的信息时，获取的信息是这个文件所指向的文件的信息；如果想要得到这个链接文件本身的信息，就需要lstat函数。

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

int main()
{
    //lstat函数
    //cc.txt是一个软链接，指向的是c.txt，文件大小为624，而cc.txt本身只有5字节
    struct stat stat_out;
    int ret = lstat("./cc.txt", &stat_out);

    if(ret == -1)
    {
        perror("stat");
        return -1;
    }

    //根据传出参数获取结构体中的信息
    int size = stat_out.st_size;    //文件大小
    printf("%d\n", size);           //5，当上面使用stat时，这里的输出是624

    return 0;
}
```

5. 实现ls -l，了解获取文件类型、文件所有者、文件权限等宏的操作

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```c
#include <unistd.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <string.h>

// 模拟实现 ls -l 指令
// -rw-rw-r-- 1 nowcoder nowcoder 12 12月  3 15:48 a.txt
int main(int argc, char * argv[]) {

    // 判断输入的参数是否正确
    if(argc < 2) {
        printf("%s filename\n", argv[0]);
        return -1;
    }

    // 通过stat函数获取用户传入的文件的信息
    struct stat st;
    int ret = stat(argv[1], &st);
    if(ret == -1) {
        perror("stat");
        return -1;
    }

    // 1、获取文件类型和文件权限
    char perms[11] = {0};   // 用于保存文件类型和文件权限的字符串

    switch(st.st_mode & S_IFMT) {   //变量值通过和S_IFMT与操作得到类型
        case S_IFLNK:
            perms[0] = 'l';
            break;
        case S_IFDIR:
            perms[0] = 'd';
            break;
        case S_IFREG:
            perms[0] = '-';
            break;
        case S_IFBLK:
            perms[0] = 'b';
            break;
        case S_IFCHR:
            perms[0] = 'c';
            break;
        case S_IFSOCK:
            perms[0] = 's';
            break;
        case S_IFIFO:
            perms[0] = 'p';
            break;
        default:
            perms[0] = '?';
            break;
    }

    //2、 判断文件的访问权限，和对应的权限宏进行按位与操作，得到是否有这个权限，如果有则为1，否则为0

    // 文件所有者
```

```
    perms[1] = (st.st_mode & S_IRUSR) ? 'r' : '-';
    perms[2] = (st.st_mode & S_IWUSR) ? 'w' : '-';
    perms[3] = (st.st_mode & S_IXUSR) ? 'x' : '-';

    // 文件所在组
    perms[4] = (st.st_mode & S_IRGRP) ? 'r' : '-';
    perms[5] = (st.st_mode & S_IWGRP) ? 'w' : '-';
    perms[6] = (st.st_mode & S_IXGRP) ? 'x' : '-';

    // 其他人
    perms[7] = (st.st_mode & S_IROTH) ? 'r' : '-';
    perms[8] = (st.st_mode & S_IWOTH) ? 'w' : '-';
    perms[9] = (st.st_mode & S_IXOTH) ? 'x' : '-';

    //3、 硬连接数
    int linkNum = st.st_nlink;

    //4、 文件所有者，根据所有者id，通过getpwuid函数的返回值获取所有者的名称
    char * fileUser = getpwuid(st.st_uid)->pw_name;

    //5、 文件所在组，类似于文件所有者的获取，通过组id间接获取名称
    char * fileGrp = getgrgid(st.st_gid)->gr_name;

    //6、 文件大小
    long int fileSize = st.st_size;

    //7、 获取修改的时间
    char * time = ctime(&st.st_mtime);  //将秒数转化为时间,这个st结构体中的时间是1970年至现在的秒数。

    char mtime[512] = {0};
    strncpy(mtime, time, strlen(time) - 1); //删除最后的换行符

    //将上面的结果连接到一个结果数组中
    char buf[1024];
    sprintf(buf, "%s %d %s %s %ld %s %s", perms, linkNum, fileUser, fileGrp, fileSize, mtime, argv[1]);

    printf("%s\n", buf);

    return 0;
}
```

access函数

1. 函数原型

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

2. 参数与返回值

> 作用：判断调用进程是否可以访问文件路径名，主要判断文件是否存在、是否具有某个权限。如果pathname 是符号链接，则取消引用，也就是说，判断的不是符号链接文件，而是链接到的文件。
>
> 参数：
>
> - pathname: 判断的文件路径
> - mode:
>
> R_OK: 判断是否有读权限
>
> W_OK: 判断是否有写权限
>
> X_OK: 判断是否有执行权限
>
> F_OK: 判断文件是否存在
>
> 返回值：成功返回0， 失败返回-1

3. 测试代码

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

int main()
{
    //access函数的使用，如果是那么返回0，否则返回-1
    int isW = access("c.txt", W_OK);    //是否可写
    int isR = access("c.txt", R_OK);    //是否可读
    int isX = access("c.txt", X_OK);    //是否可执行
    int isF = access("c.txt", F_OK);    //是否存在

    printf("%d\n", isW);
    printf("%d\n", isR);
    printf("%d\n", isX);
    printf("%d\n", isF);

    //如果pathname是一个符号链接文件，那么他的指向就是链接到的文件
    int ret = access("cc.txt", X_OK);
    printf("%d\n", ret);        //-1

    return 0;
}
```

chmod函数

1. 函数原型

```
#include <sys/stat.h>

//功能一样，一个通过文件名指定改变的文件，一个通过文件描述符指定
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```

2. 参数与返回值

> 作用: 修改文件的权限，和access一样，如果指定的文件是一个符号链接，那么修改的是符号链接指定的文件。
>
> 参数：
>
> - pathname: 需要修改的文件的路径
> - mode:需要修改的权限值，八进制的数
>
> 返回值：修改成功返回0，失败返回-1

3. 代码测试

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

int main()
{
    //chmod函数
    int ret = chmod("c.txt", 0777);
    printf("%d\n", ret);  //0，修改成功，权限全部为是  ---> 0777

    //修改符号链接文件
    ret = chmod("cc.txt", 0644);
    printf("%d\n", ret);  //0，修改成功，权限根据0644删除了原来的部分权限，且是对链接到的文件修改，即c.txt文件

    return 0;
}
```

truncate函数

1. 函数原型

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

//功能一样的两个函数
 int truncate(const char *path, off_t length);
 int ftruncate(int fd, off_t length);
```

2. 参数返回值

> 作用：缩减或者扩展文件的尺寸至指定的大小，如果指定的文件是符号链接文件，那么它修改的是链接到的文件，而不是符号链接本身的文件。
>
> 参数：
>
> - path: 需要修改的文件的路径
> - length: 需要最终文件变成的大小
>
> 返回值：
>
> 修改成功返回0， 失败返回-1

3. 代码测试

```
int main()
{
    //truncate函数

    int ret = truncate("b.txt", 10);

    if(ret == -1)
    {
        perror("truncate");
        return -1;
    }

    return 0;
}
```

Linux的目录IO函数

mkdir函数

```
/**
     mkdir函数原型:
        #include <sys/stat.h>
        #include <sys/types.h>

        int mkdir(const char *pathname, mode_t mode);
    参数
        pathname ---> 创建的目录路径
        mode ---> 目录的权限，八进制数，可以是宏；和open打开文件的mode原理一样
    返回值
        创建成功返回0，否则返回-1，并且设置对应的错误编号errno
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

int main()
{
    int ret = mkdir("/home/kiko/lesson/myfile/dir1", 0777);

    if(ret == -1)
    {
        perror("mkdir");  //比如，如果路径已经存在，那么会输出---> mkdir: File exists
        return -1;
    }

    return 0;
}
```

rmdir函数

```
/**
     rmdir函数原型:
        #include <unistd.h>

       int rmdir(const char *pathname);
    作用
        rmdir() deletes a directory, which must be empty.
    参数
        pathname ---> 要删除的目录路径
    返回值
        删除成功返回0，否则返回-1，并且设置对应的错误编号errno
 */

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

int main()
{
```

```
    int ret = rmdir("/home/kiko/lesson/myfile/dir1");

    if(ret == -1)
    {
        perror("rmdir");  //比如删除的目录不是一个空目录 --> rmdir: Directory not
empty
        return -1;
    }

    return 0;
}
```

rename函数

```
 /**
     rename函数原型:
       #include <stdio.h>

       int rename(const char *oldpath, const char *newpath);
    作用
        修改文件或者目录的文件名，如果是目录，即使目录不是一个空目录也能修改其目录名

    参数
        oldpath ---> 旧的路径名
        newpath ---> 新的路径名
    返回值
        修改成功返回0，否则返回-1，并且设置对应的错误编号errno
 */

#include <stdio.h>

int main()
{
    int ret = rename("/home/kiko/lesson/myfile/c.txt",
"/home/kiko/lesson/myfile/ccc.txt");

    if(ret == -1)
    {
        perror("rename");
        return -1;
    }

    return 0;
}
```

chdir和getcwd函数

```
/**
    函数原型:
        #include <unistd.h>

        int chdir(const char *path);
    作用：修改进程的工作目录
        比如在/home/nowcoder 启动了一个可执行程序a.out, 进程的工作目录 /home/nowcoder
    参数:
        path : 需要修改到的工作目录
```

```
-------------------------------------------------------------------
    函数原型:
        #include <unistd.h>

        char *getcwd(char *buf, size_t size);
    作用：获取当前工作目录
    参数：
        - buf : 存储的路径，指向的是一个数组（传出参数）
        - size: 数组的大小
    返回值：
        返回的指向的一块内存，这个数据就是第一个参数

综上两个函数：
    当chdir执行改变工作目录之后，那么在此之后的工作都是在新的目录里面执行的，
    比如下面的代码在新的目录中创建了文件，会在修改后的工作目录里面创建出来。
 */

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

int main()
{
    //1、chdir函数
    int ret = chdir("/home/kiko/lesson/chdirv");

    if(ret == -1)
    {
        perror("chdir");
        return -1;
    }

    //此处创建一个文件并写入数据
    int fd = open("chdir.txt", O_WRONLY | O_CREAT | O_TRUNC, 0777);

    if(fd == -1)
    {
        perror("open");
        return -1;
    }

    //向文件写入内容
    void * str = "这是一段内容";
    write(fd, str, sizeof(void) * strlen(str));
    close(fd);  //关闭文件

    //2、getcwd函数
    char buf[128] = {0};
    char * result = getcwd(buf, sizeof(buf));
    printf("%s\n", result);  //直接输出buf也是可以的

    return 0;

}
```

遍历目录函数opendir、readdir和closedir

```
/*
    //1、打开一个目录
    函数原型：
        #include <sys/types.h>
        #include <dirent.h>

        DIR *opendir(const char *name);
    参数：
        - name: 需要打开的目录的名称
    返回值：
        DIR * 类型，理解为目录流
        错误返回NULL

    //2、读取目录中的数据
    函数原型：
        #include <dirent.h>

        struct dirent *readdir(DIR *dirp);
    - 参数：dirp是opendir函数返回的结果
    - 返回值：
        struct dirent，代表读取到的文件（可以是各种文件类型的文件）的信息
        读取到了末尾或者失败了，返回NULL。
        如果到达目录流的末端，返回NULL，errno不改变。 如果发生错误，则返回NULL，errno被适当地设置。
        为了区分目录流的结束和错误，在调用readdir()之前将errno设置为零，然后检查errno的值，如果NULL 被返回时，再检查errno的值。

    //3、关闭目录
    函数原型：
        #include <sys/types.h>
        #include <dirent.h>

        int closedir(DIR *dirp);

*/

#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int getCommonFileCount(const char *pathname); //函数声明

//通过传入的目录来实现查找这个目录中所有的普通文件个数（包括目录内部的目录的普通文件）
int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        printf("请传入一个需要扫描的目录\n");
        return -1;
    }

    int count = getCommonFileCount(argv[1]);
```

```c
    printf("普通文件个数-->%d\n", count);

    return 0;
}

int getCommonFileCount(const char *pathanme)
{
    int total = 0; //普通文件个数计数器，当作返回值返回

    // 1、打开传入的目录
    DIR *curDIR = opendir(pathanme);

    if (curDIR == NULL)
    {
        perror("opendir");
        exit(0);
    }

    // 2、读取打开的目录流
    struct dirent *ptr;
    while ((ptr = readdir(curDIR)) != NULL)
    {
        //获取文件名称
        char *dname = ptr->d_name;

        //根据文件名称剔除掉.和..
        if (strcmp(dname, ".") == 0 || strcmp(dname, "..") == 0)
        {
            continue; //继续读取下一个文件即可
        }

        //判断是目录还是普通文件
        if ((ptr->d_type) == DT_DIR)
        {
            //是目录，则递归继续判断这个目录中的普通文件
            char newpathname[256];
            sprintf(newpathname, "%s/%s", pathanme, dname); //拼接新的目录地址
            total += getCommonFileCount(newpathname);
        }

        //如果是普通文件，则添加计数即可
        if ((ptr->d_type) == DT_REG)
        {
            total++;
        }
    }

    //关闭目录流
    closedir(curDIR);

    //返回结果
    return total;
}
```

readdir函数的返回值结构体

```
struct dirent                                   d_type
{                                                       DT_BLK  - 块设备
    // 此目录进入点的inode                              DT_CHR  - 字符设备
    ino_t d_ino;                                        DT_DIR  - 目录
    // 目录文件开头至此目录进入点的位移                 DT_LNK  - 软连接
    off_t d_off;                                        DT_FIFO - 管道
    // d_name 的长度, 不包含NULL字符                    DT_REG  - 普通文件
    unsigned short int d_reclen;                        DT_SOCK - 套接字
    // d_name 所指的文件类型                            DT_UNKNOWN - 未知
    unsigned char d_type;
    // 文件名
    char d_name[256];
};                                                                  CSDN @C学习者n号
```

Linux的文件描述符复制函数

dup函数

```
/*
    函数原型:
        #include <unistd.h>

        int dup(int oldfd);
    作用：复制一个新的文件描述符
        fd=3, int fd1 = dup(fd),
        fd指向的是a.txt, fd1也是指向a.txt
        从空闲的文件描述符表中找一个最小的，作为新的拷贝的文件描述符
*/

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

int main()
{
    int fd = open("a.txt", O_RDWR | O_CREAT, 0664);

    int fd1 = dup(fd);

    if (fd1 == -1)
    {
        perror("dup");
        return -1;
    }

    printf("fd : %d , fd1 : %d\n", fd, fd1);  //输出3 , 4

    close(fd);

    char *str = "hello,world";
```

```
    int ret = write(fd1, str, strlen(str));
    if (ret == -1)
    {
        perror("write");
        return -1;
    }

    close(fd1);

    return 0;
}
```

dup2函数

```
/*
    函数原型:
        #include <unistd.h>

        int dup2(int oldfd, int newfd);
    作用：重定向文件描述符
        oldfd 指向 a.txt, newfd 指向 b.txt
        调用函数成功后：newfd 和 b.txt 做close, newfd 指向了 a.txt，所以newfd和oldfd指向相同的文件
        oldfd 必须是一个有效的文件描述符
        oldfd和newfd值相同，相当于什么都没有做
    返回值
        On success, these system calls return the new file descriptor.
        On error, -1 is returned, and errno is set appropriately.

    和dup函数的区别：
        dup函数是根据传入的文件描述符，找到一个文件描述符表中可用的最小的文件描述符来指向和参数文件描述符指向一致的文件描述符作为返回值
        dup2函数是在有两个有指向的文件描述符情况下，让newfd文件描述符指向和oldfd指向的文件一致，此时newfd和之前的文件连接关闭；
            文件描述符本身数值不变，只是指向变了，同时他的返回值也是和newfd的值一样的。
*/
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{

    int fd = open("1.txt", O_RDWR | O_CREAT | O_TRUNC, 0664);
    if (fd == -1)
    {
        perror("open");
        return -1;
    }

    int fd1 = open("2.txt", O_RDWR | O_CREAT, 0664);
    if (fd1 == -1)
    {
        perror("open");
```

```
        return -1;
    }

    printf("fd : %d, fd1 : %d\n", fd, fd1);

    int fd2 = dup2(fd, fd1); //此时fd1也指向了fd指向的文件，返回值和fd1一样
    if (fd2 == -1)
    {
        perror("dup2");
        return -1;
    }

    // 通过fd1去写数据，实际操作的是1.txt，而不是2.txt
    char *str = "hello, dup2";
    int len = write(fd1, str, strlen(str));

    if (len == -1)
    {
        perror("write");
        return -1;
    }

    printf("fd : %d, fd1 : %d, fd2 : %d\n", fd, fd1, fd2);

    close(fd);
    close(fd1);

    return 0;
}
```

fctnl函数

```
/*

    #include <unistd.h>
    #include <fcntl.h>

    int fcntl(int fd, int cmd, ...);
    参数:
        fd : 表示需要操作的文件描述符
        cmd: 表示对文件描述符进行如何操作
            - F_DUPFD : 复制文件描述符,复制的是第一个参数fd，得到一个新的文件描述符（返回
值）
                int ret = fcntl(fd, F_DUPFD);

            - F_GETFL : 获取指定的文件描述符文件状态flag
              获取的flag和我们通过open函数传递的flag是一个东西。

            - F_SETFL : 设置文件描述符文件状态flag
              必选项: O_RDONLY, O_WRONLY, O_RDWR 不可以被修改
              可选性: O_APPEND, O_NONBLOCK
                O_APPEND 表示追加数据
                O_NONBLOK 设置成非阻塞

        阻塞和非阻塞：描述的是函数调用的行为。
*/
```

```c
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>

int main()
{

    // 1.复制文件描述符
    // int fd = open("1.txt", O_RDONLY);
    // int ret = fcntl(fd, F_DUPFD);

    // 2.修改或者获取文件状态flag
    int fd = open("1.txt", O_RDWR);
    if (fd == -1)
    {
        perror("open");
        return -1;
    }

    // 获取文件描述符状态flag
    int flag = fcntl(fd, F_GETFL);
    if (flag == -1)
    {
        perror("fcntl");
        return -1;
    }
    flag |= O_APPEND; // flag = flag | O_APPEND

    // 修改文件描述符状态的flag，给flag加入O_APPEND这个标记
    int ret = fcntl(fd, F_SETFL, flag);
    if (ret == -1)
    {
        perror("fcntl");
        return -1;
    }

    char *str = "nihao";
    write(fd, str, strlen(str));

    close(fd);

    return 0;
}
```