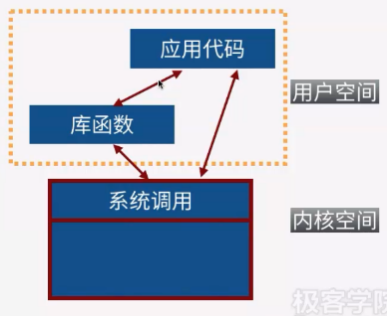
# 文件描述符

## 简介



注：**库函数即标准IO，系统调用即低级IO**。

**在库函数层面表示的标准I/O用流来表示，在系统调用层面用文件描述字/符表示**。不论是流还是文件描述字，表示的**都是进程中与硬盘中文件的连接关系**。

**低级IO又叫文件IO。**

文件描述字：

另一种连接机制（程序与文件）

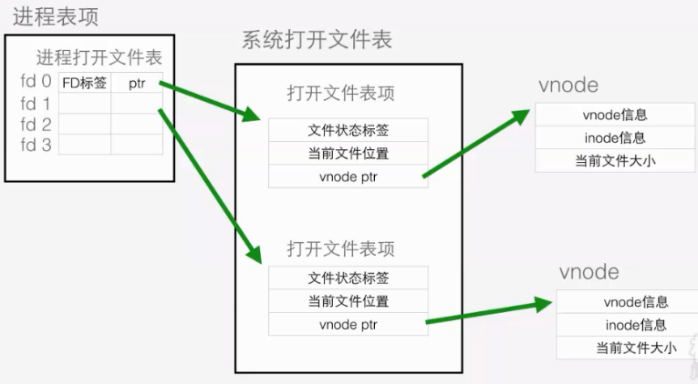
系统调用（底层基本的IO操作）

特定设备（流不行）

特殊方式进行IO（流不行）

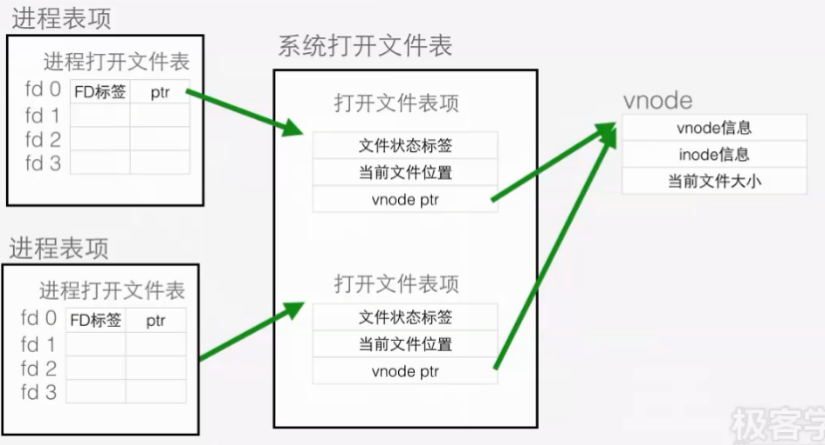
## 内核数据结构

一个进程打开多个文件：



进程表项是用户空间伴随着open创建的，系统打开文件表对应系统内核管理的数据结构。这样可以理解：一个进程中打开不同文件，进程里的文件不存在相互干扰。

多个进程打开同一文件：



## 操作

### 创建文件

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

int main(int argc,char \*\*argv)

{

int ret = -1;

//创建新文件b.c，权限读写执行都有

ret = creat("b.c",0777);

if( -1 == ret )

{

perror("create");

return -1;

}

return 0;

}

### 打开文件

原型：

int open(const char \*pathname, int flags);//只是执行打开

int open(const char \*pathname, int flags, mode\_t mode);//还可以创建（集成了create功能）

参数：flag：O\_RDONLY/O\_WRONLY/O\_RDWR

返回值：int（范围有限制）

预先打开的文件描述字：0,1,2

示例（打开文件）：

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

int main(int argc,char \*\*argv)

{

int ret = -1;

//打开文件b.c

ret = open("b.c",O\_RDWR);//流 r+

//ret = open("b.c",O\_RDWR|O\_TRUNC);//流 w+

//ret = open("b.c",O\_RDWR|O\_APPEND);//流 a+

if( -1 == ret )

{

perror("create");

return -1;

}

close(ret);

return 0;

}

示例（打开并创建）：

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

int main(int argc,char \*\*argv)

{

int ret = -1;

//创建新文件b.c，权限读写执行都有,并且打开文件

ret = open("b.c",O\_RDWR|O\_CREAT,0777);

if( -1 == ret )

{

perror("create");

return -1;

}

close(ret);

return 0;

}

示例（文件存在正常打开，不存在则创建）：

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

#include <errno.h>

int main(int argc,char \*\*argv)

{

int ret = -1;

//文件存在打开,不存在就创建打开

//创建新文件b.c，权限读写执行都有,并且打开文件

ret = open("b.c",O\_RDWR|O\_CREAT|O\_EXCL,0777);

if( -1 == ret )

{

if ( EEXIST == errno )//是否存在

{

ret = open("b.c",O\_RDWR);

if ( -1 == ret )

{

perror("open");

return -1;

}

}

return -1;

}

close(ret);

return 0;

}

示例（文件描述字）：

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

int main(int argc,char \*\*argv)

{

int ret = -1;

//打开文件b.c

while(1)

{

ret = open("b.c",O\_RDWR);//流 r+

if( -1 == ret )

{

perror("create");

return -1;

}

//显示文件描述字

printf("%d\n",ret);

}

close(ret);

return 0;

}

执行发现只能到1023，这个是可以突破上限的，通过ulimit –a查看open files，可以通过ulimit –n num修改

### 关闭文件

### 读写

#### read

示例：

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

#include <stdlib.h>

#include <string.h>

#define MAX (256\*2\*2\*2\*2\*2\*2\*2\*2\*2\*2\*2\*2\*2\*2)

//内存开的越大，耗时先减少后增加，这说明代码中使用用户分配的内存时，不是越大越好，也不是越小越好，应该根据系统的特性检测出一个比较正确的值

int main(int argc,char \*\*argv)

{

//拷贝test.mp4 到 new.mp4中

//打开文件，源 目的

int fd\_s = -1;//源文件 描述字

int fd\_d = -1;//目标文件 描述字

int ret = -1;

fd\_s = open("test.mp4",O\_RDONLY);//流 r 打开源文件

if (-1 == fd\_s )

{

perror("open");

return -1;

}

fd\_d = open("new.mp4",O\_WRONLY|O\_TRUNC|O\_CREAT,0777);//流 w 打开目标文件

if (-1 == fd\_d )

{

perror("open");

return -1;

}

//分配内存

char \*mem = NULL;

mem = malloc(MAX);

if ( NULL == mem )

{

perror("malloc");

return -1;

}

memset(mem,'\0',MAX);

//(读取－》写入)

while(1)

{

ret = read(fd\_s,mem,MAX);

if( -1 == ret )

{

perror("read");

return -1;

}

if( 0 == ret )

{

break;

}

ret = write(fd\_d,mem,ret);

if( -1 == ret )

{

perror("write");

return -1;

}

}

//释放内存

free(mem);

//关闭文件

close(fd\_s);

close(fd\_d);

return 0;

}

可以使用time ./a.out查看可执行文件的实际耗时，系统时间和用户时间。

#### write

#### 文件结束

#### IO效率

### 定位

示例：

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

#include <stdlib.h>

#include <string.h>

#define MAX (256\*2\*2\*2\*2\*2\*2\*2\*2\*2\*2\*2\*2\*2\*2)

int main(int argc,char \*\*argv)

{

//拷贝test.mp4 到 new.mp4中

//打开文件，源 目的

int fd\_s = -1;//源文件 描述字

int fd\_d = -1;//目标文件 描述字

int ret = -1;

fd\_s = open("test.mp4",O\_RDONLY);//流 r 打开源文件

if (-1 == fd\_s )

{

perror("open");

return -1;

}

fd\_d = open("new.mp4",O\_WRONLY|O\_TRUNC|O\_CREAT,0777);//流 w 打开目标文件

if (-1 == fd\_d )

{

perror("open");

return -1;

}

//计算出源文件大小

// 源文件定位到尾部， 计算出大小

long file\_size = -1;

file\_size = lseek(fd\_s,0,SEEK\_END); //不是fseek

if( -1 == file\_size )

{

perror("lseek");

return -1;

}

// 源文件定位到头部

ret = lseek(fd\_s,0,SEEK\_SET);

if( -1 == ret )

{

perror("lseek");

return -1;

}

//分配内存

char \*mem = NULL;

mem = malloc(file\_size);

if ( NULL == mem )

{

perror("malloc");

return -1;

}

memset(mem,'\0',file\_size);

//(读取－》写入)

//while(1)

// {

ret = read(fd\_s,mem,file\_size);

if( -1 == ret )

{

perror("read");

return -1;

}

if( 0 == ret )

{

// break;

}

ret = write(fd\_d,mem,ret);

if( -1 == ret )

{

perror("write");

return -1;

}

// }

//释放内存

free(mem);

//关闭文件

close(fd\_s);

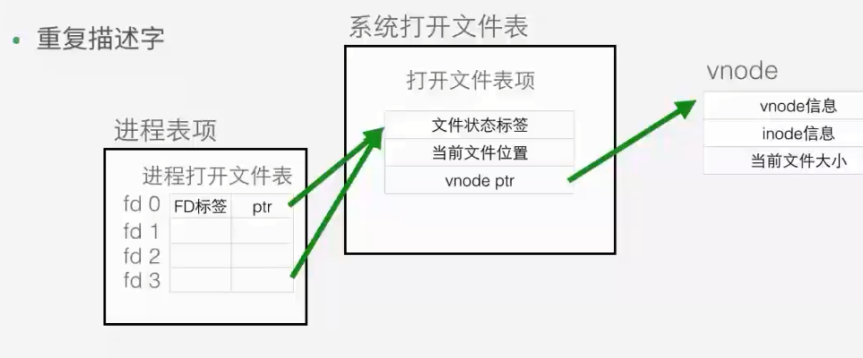
close(fd\_d);

return 0;

}

### 复制

#### 复制描述字



其实复制的是文件描述字进程表项中的fd和fd标签，在系统层面的表项和属性全部都是一样的。

#### dup系列函数

原型：

int dup(int oldfd);

int dup2(int oldfd, int newfd);

int dup3(int oldfd, int newfd, int flags);

注：dup是复制进程信息，系统的信息没有复制，它与两次打开文件是不一样的。

示例：

#include <stdio.h>

int main(int argc,char \*\*argv)

{

int ret = -1;

int fd = -1;

fd = dup(1);//1 --- >标准输出（最后的在fd上的输出其实就是标准输出上）

if ( -1 == fd )

{

perror("dup");

return -1;

}

//写入fd 🡪1（标准输出）🡪显示

ret = write(fd,"hello",5);

if ( -1 == ret )

{

perror("write");

return -1;

}

close(fd);

return 0;

}

示例：

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/types.h>

int main(int argc,char \*\*argv)

{

//重定位标准输出🡪文件

//

int ret = -1;

int fd = -1;

#if 1

fd = open("log",O\_CREAT|O\_TRUNC|O\_RDWR);

if ( -1 == fd )

{

perror("open");

return -1;

}

ret = dup2(fd,1);//标准输出重定位到文件

if ( -1 == ret )

{

perror("dup");

return -1;

}

#endif

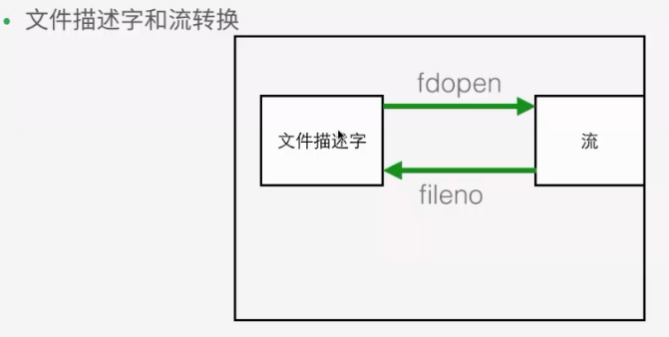
printf("hellworld\n");

close(fd);

return 0;

}

### 与流互换



注：流是由库函数产生的，文件描述符是由系统调用产生的。

示例：

#include <stdio.h>

int main(int argc,char \*\*argv)

{

//测试标准输入流 stdin(man stdin查看帮助文档) 对应的文件描述字

int fd = -1;

fd = fileno(stdin);

if( -1 == fd )

{

perror("fileno");

return -1;

}

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

return 0;

}

示例：

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <fcntl.h>

int main(int argc,char \*\*argv)

{

int ret = -1;

int fd = -1;

//打开文件b.c

fd = open("b.c",O\_RDWR);//流 r+

//ret = open("b.c",O\_RDWR|O\_TRUNC);//流 w+

//ret = open("b.c",O\_RDWR|O\_APPEND);//流 a+

if( -1 == fd )

{

perror("create");

return -1;

}

FILE \*fp;

//文件描述字转化为流

fp = fdopen(fd,"r+");

if( NULL == fp )

{

perror("fdopen");

return -1;

}

//用流写入数据

ret = fwrite("hello",1,5,fp);

if ( -1 == ret )

{

perror("fwrite");

return -1;

}

fflush(fp);

ret = write(fd," world",7);

if ( -1 == ret )

{

perror("fwrite");

return -1;

}

close(ret);

return 0;

}

### 文件控制

#### fcntl函数

fcntl（file description control）提供了**管理低级文件描述符**的方法，可以控制重复文件描述字，文件描述字标签和文件状态标签。

原型：int fcntl(int filedes, int cmd, … struct flock \*flockptr);

参数：flock结构体

Struct flock{

short l\_type; //F\_RDLCK,F\_WRLCK,F\_UNLCK

off\_t l\_start; //offset in bytes,relative to l\_whence

short l\_whence; //SEEK\_SET,SEEK\_CUR,or SEEK\_END

off\_t l\_len; //length,in bytes; o means lock to EOF

pid\_t l\_pid; //return with F\_GETLK

}

示例：

#include <stdio.h>

#include <fcntl.h>

#include <sys/types.h>

#include <unistd.h>

int main(int argc,char \*\*argv)

{

int ret = -1;

int fd = -1;

fd = fcntl(1,F\_DUPFD,0);//等价于下一样，完成文件描述字的复制

// fd = dup(1);//1 --- > 标准输出

if ( -1 == fd )

{

perror("dup");

return -1;

}

//写入fd🡪1（标准输出）🡪显示

ret = write(fd,"hello",5);

if ( -1 == ret )

{

perror("write");

return -1;

}

close(fd);

return 0;

}

示例：

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <sys/types.h>

int main(int argc,char \*\*argv)

{

//重定位标准输出🡪文件

//

int ret = -1;

int fd = -1;

#if 1

fd = open("log",O\_CREAT|O\_TRUNC|O\_RDWR);

if ( -1 == fd )

{

perror("open");

return -1;

}

close(1);

ret = fcntl(fd,F\_DUPFD,1);//标准输出重定位到文件

//ret = dup2(fd,1);//标准输出重定位到文件

if ( -1 == ret )

{

perror("dup");

return -1;

}

#endif

printf("hellworld\n");

close(fd);

return 0;

}

#### 重复文件描述字

#### 文件描述字标签

#### 文件状态标签

#### 文件上锁

当多个用户共同使用，操作一个文件时候，Linux通常采用的方法是给文件上锁，来避免共享资源产生竞争的状态。

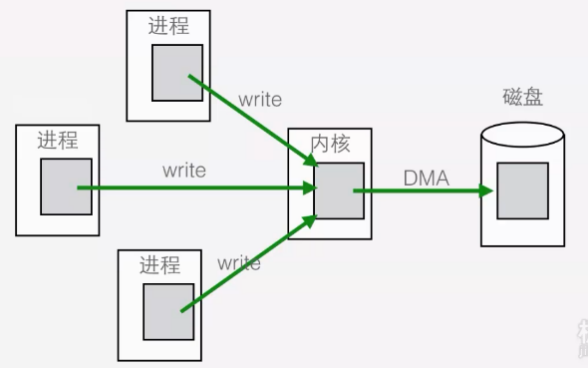
记录锁类型：

1. 共享读锁：描述符必须读打开
2. 共享写锁：描述符必须写打开

fcntl不仅可以实施建议性锁而且可以实施强制性锁。

### 缓冲

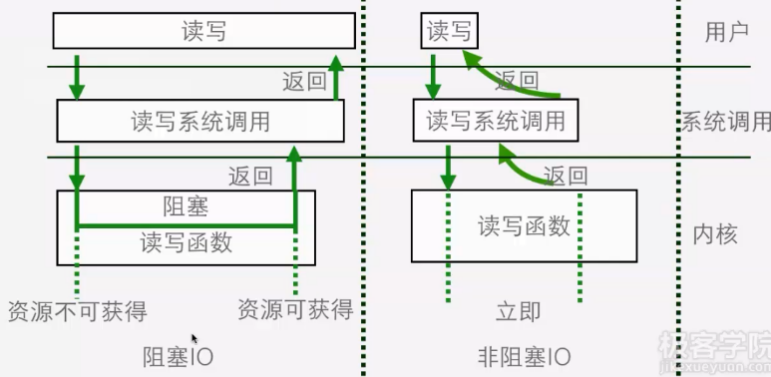
#### 内核的文件同步原理



#### fsync函数

### 非阻塞IO

#### 阻塞/非阻塞



#### 文件状态标志改变