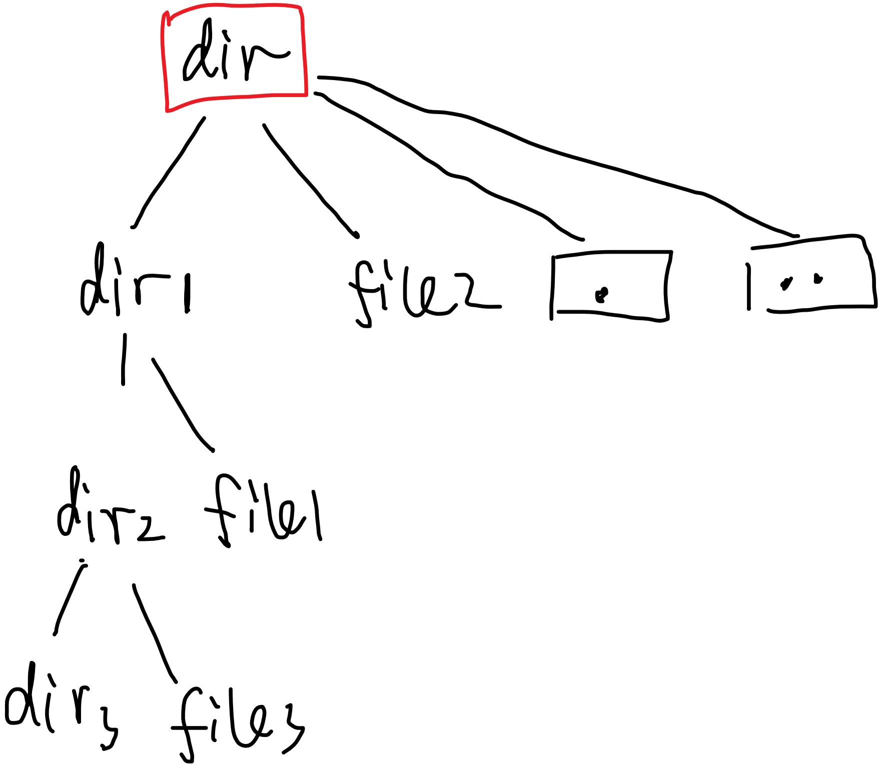
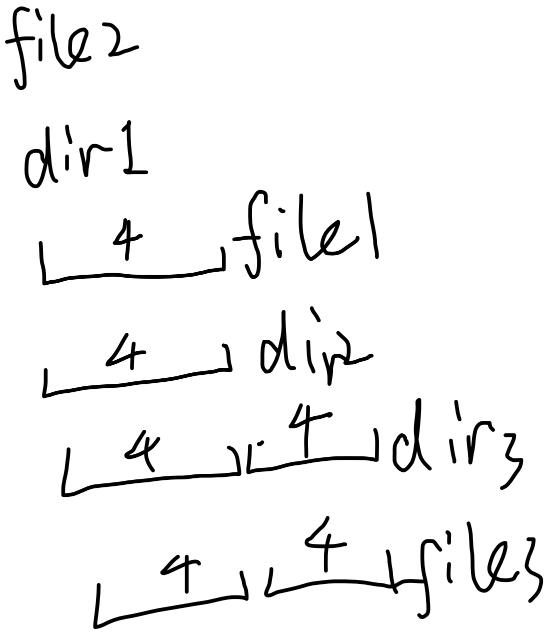
tree命令实现





1 递归.

2. 访问根结点,循环递归处理根的孩子

3、访问完,退出递归.

深度优先遍历.

```
DFS(root)
{
    access(root)
    for( ){
        DFS(root->child)
    }
}
```

printf

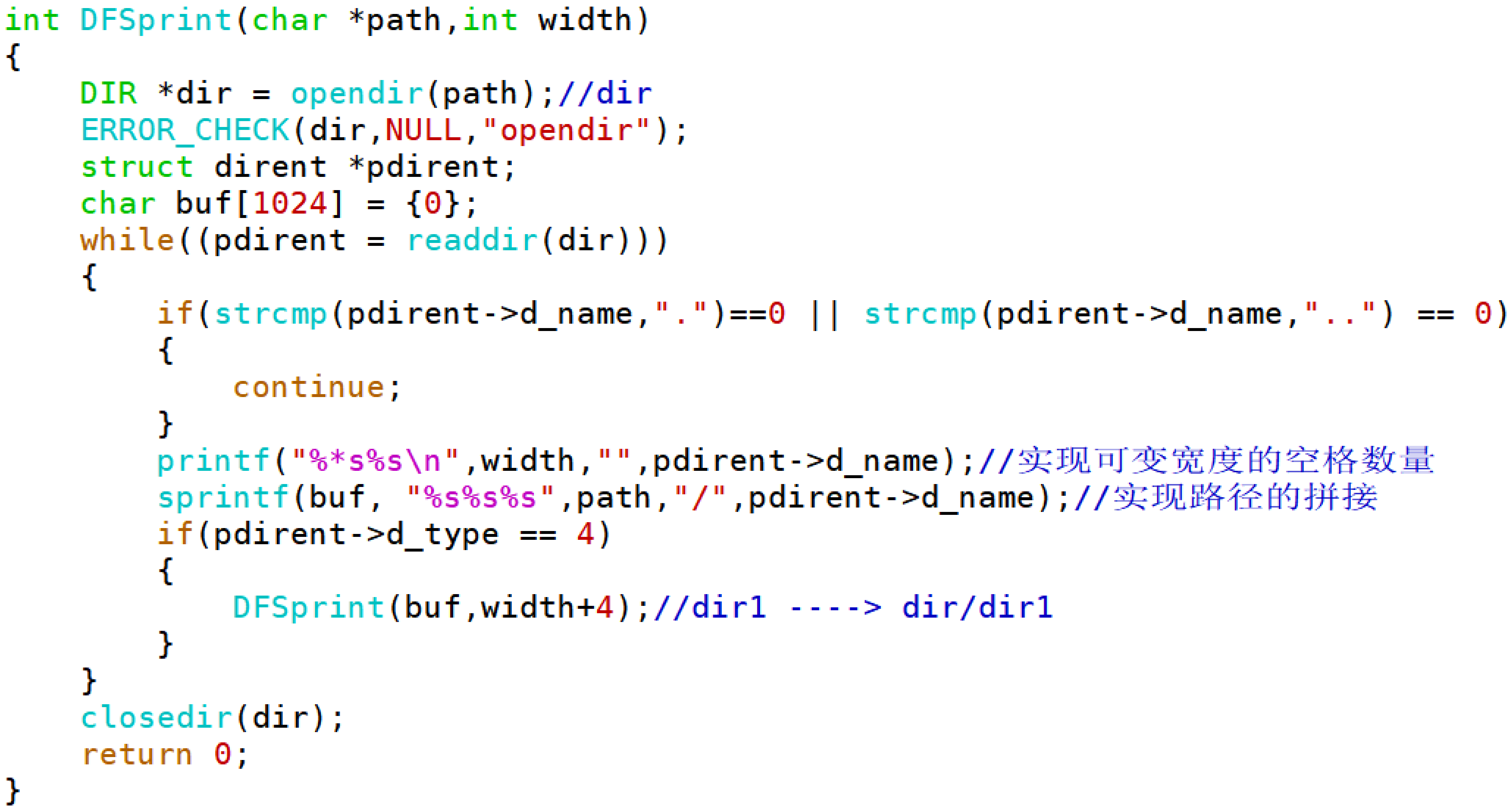
```
printf("%3d", 2);
```

```
__2
```

等价于 ⟹

```
printf("%*d",3,2);
```

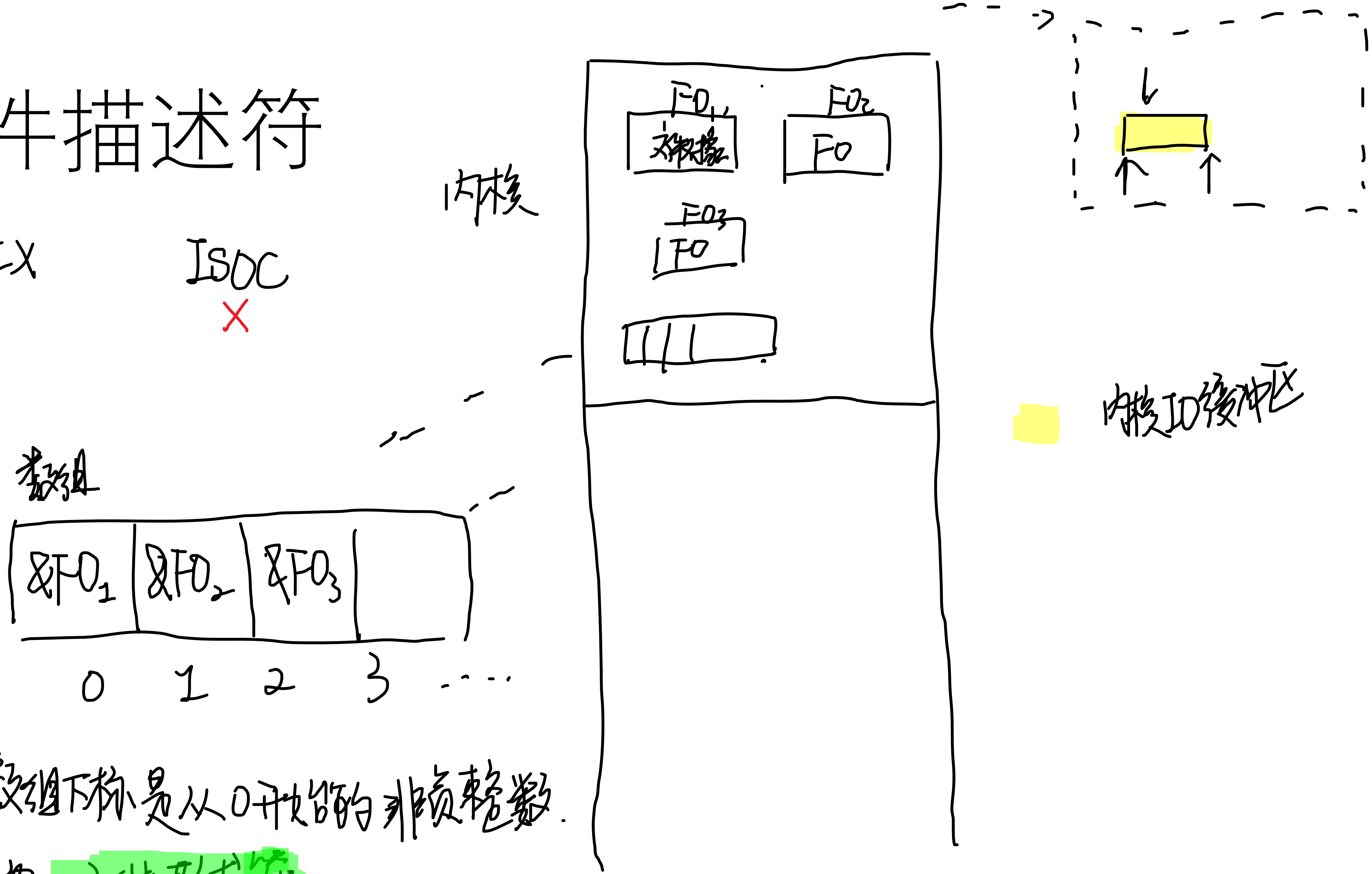
tree命令的实现



```c
int DFSprint(char *path,int width)
{
    DIR *dir = opendir(path);//dir
    ERROR_CHECK(dir,NULL,"opendir");
    struct dirent *pdirent;
    char buf[1024] = {0};
    while((pdirent = readdir(dir)))
    {
        if(strcmp(pdirent->d_name,".")==0 || strcmp(pdirent->d_name,"..") == 0)
        {
            continue;
        }
        printf("%*s%s\n",width,"",pdirent->d_name);//实现可变宽度的空格数量
        sprintf(buf, "%s%s%s",path,"/",pdirent->d_name);//实现路径的拼接
        if(pdirent->d_type == 4)
        {
            DFSprint(buf,width+4);//dir1 ----> dir/dir1
        }
    }
    closedir(dir);
    return 0;
}
```

文件描述符

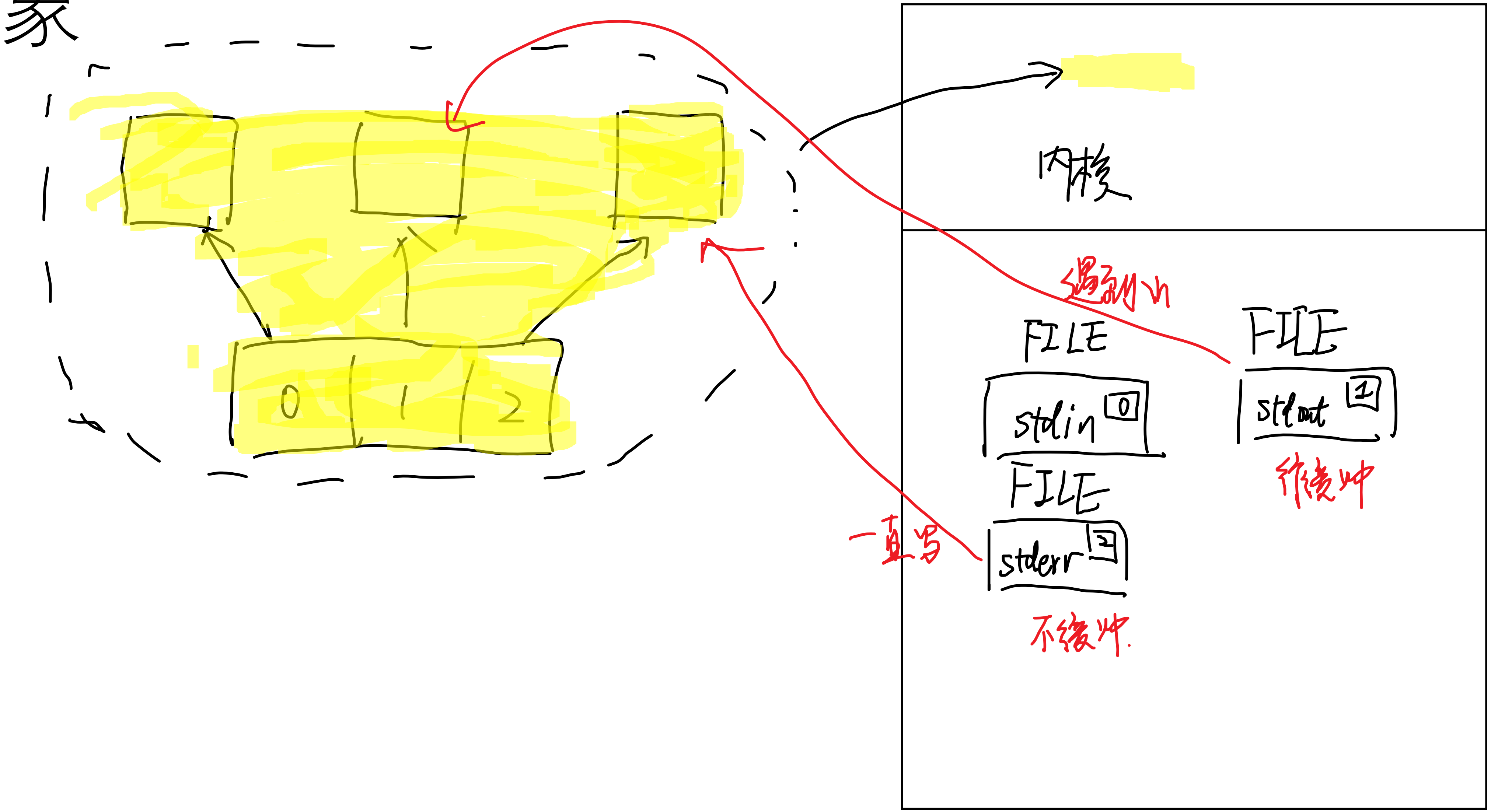
POSIX ✓　　ISOC ✗



数组下标是从0开始的非负整数.

称为 文件描述符

进程刚刚创建时，已经打开了三个文件对象

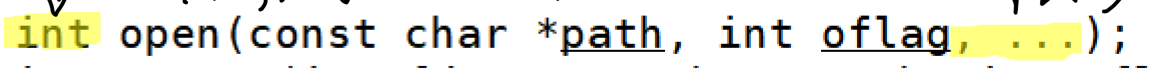


不带缓冲的IO

没有用户态缓冲区（FILE）

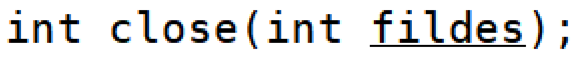
↓ 文件描述符（失败时返回 -1）　　可变参数

打开：



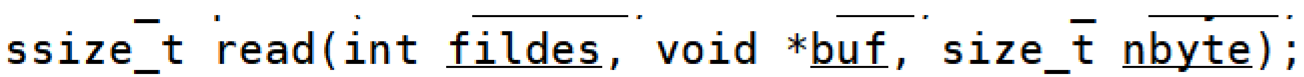
```
int open(const char *path, int oflag, ...);
```

关闭



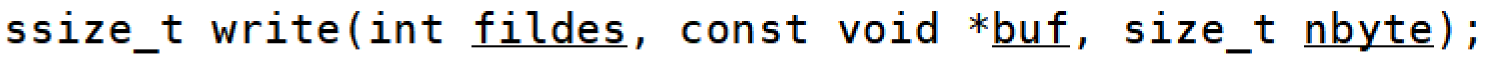
```
int close(int fildes);
```

读



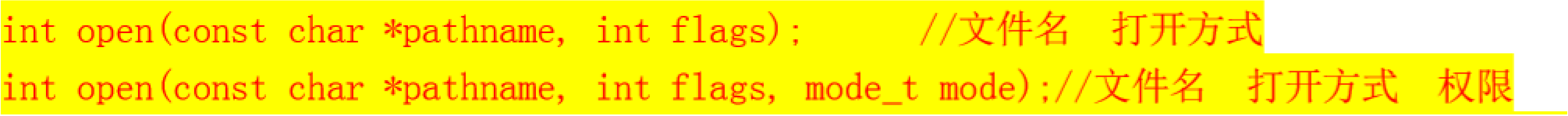
```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

写

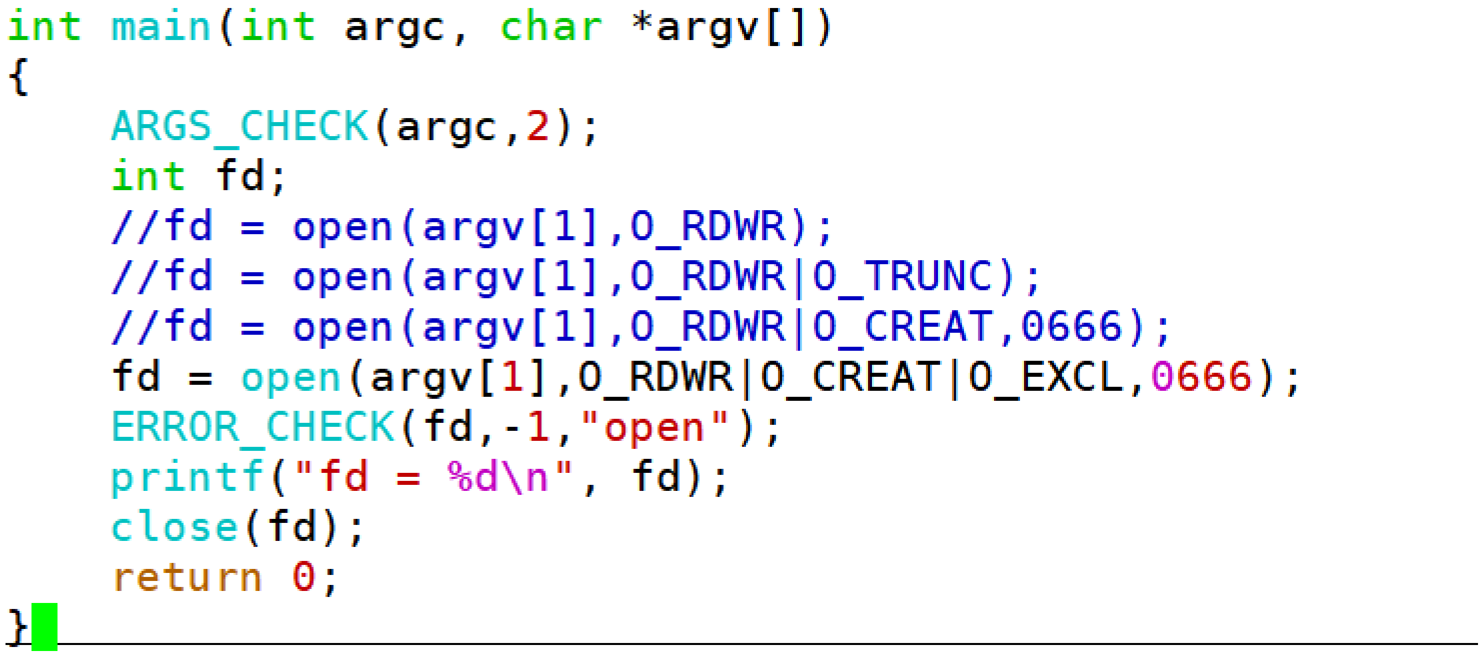


```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

open



```
int open(const char *pathname, int flags);      //文件名  打开方式
int open(const char *pathname, int flags, mode_t mode);//文件名  打开方式  权限
```

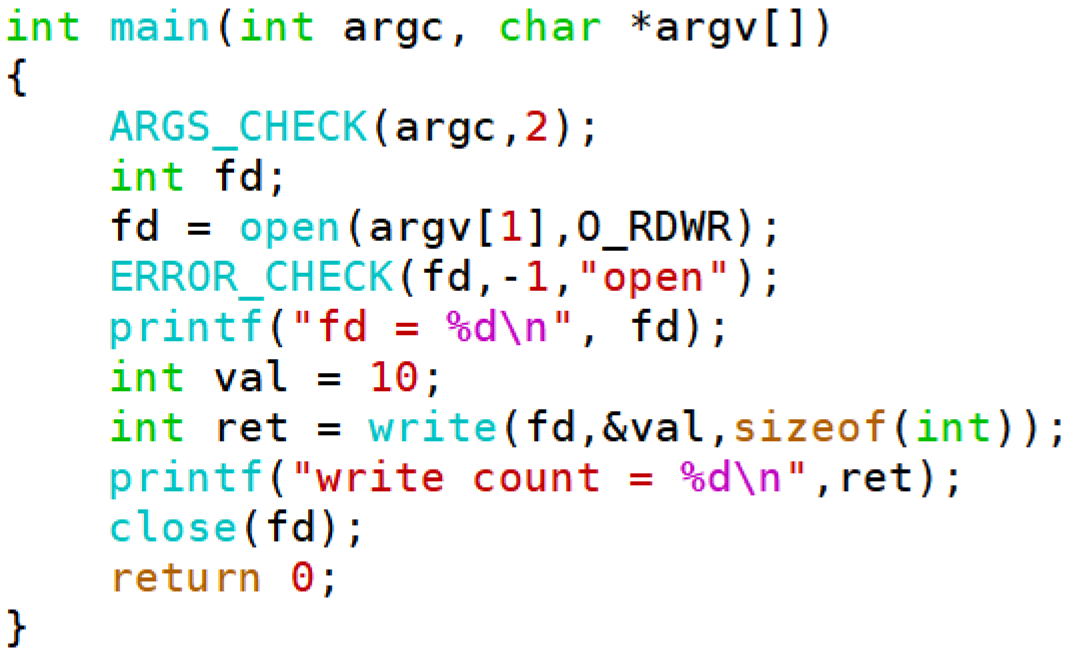


```c
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd;
    //fd = open(argv[1],O_RDWR);
    //fd = open(argv[1],O_RDWR|O_TRUNC);
    //fd = open(argv[1],O_RDWR|O_CREAT,0666);
    fd = open(argv[1],O_RDWR|O_CREAT|O_EXCL,0666);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n", fd);
    close(fd);
    return 0;
}
```

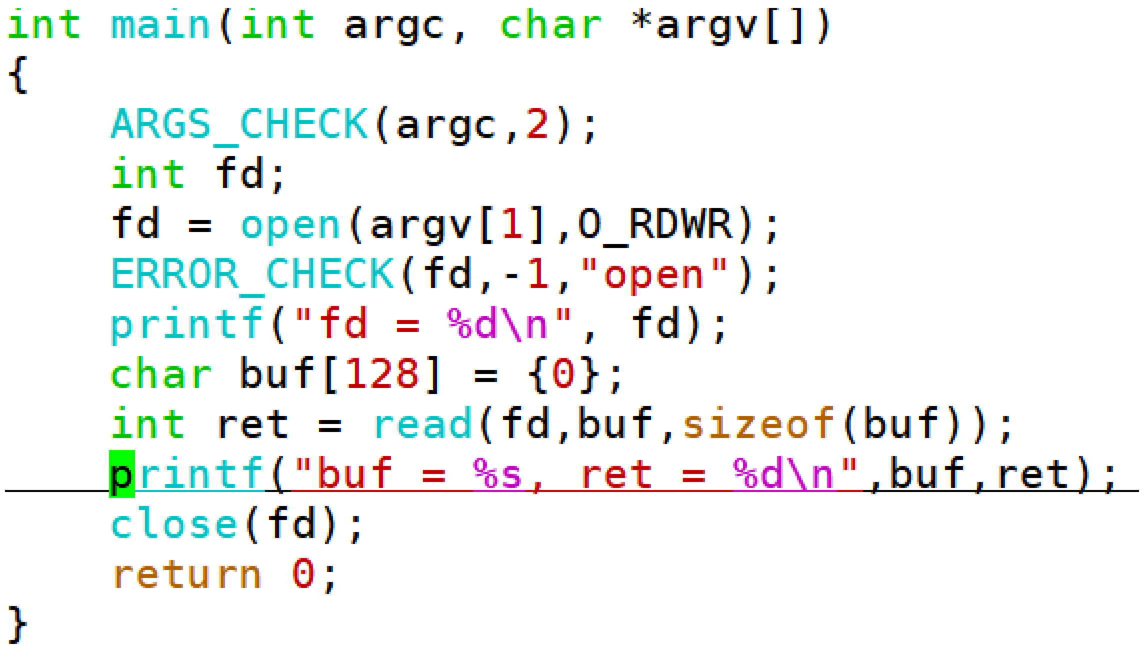
read&write

ssize_t read(int fd, void *buf, size_t count);//文件描述词 缓冲区 长度

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



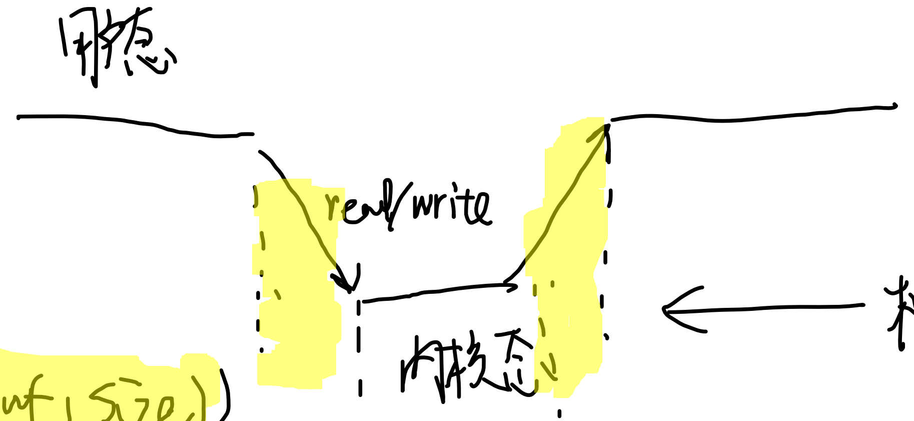
```c
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd;
    fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n", fd);
    int val = 10;
    int ret = write(fd,&val,sizeof(int));
    printf("write count = %d\n",ret);
    close(fd);
    return 0;
}
```



```c
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd;
    fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n", fd);
    char buf[128] = {0};
    int ret = read(fd,buf,sizeof(buf));
    printf("buf = %s, ret = %d\n",buf,ret);
    close(fd);
    return 0;
}
```

效率问题

系统调用 .



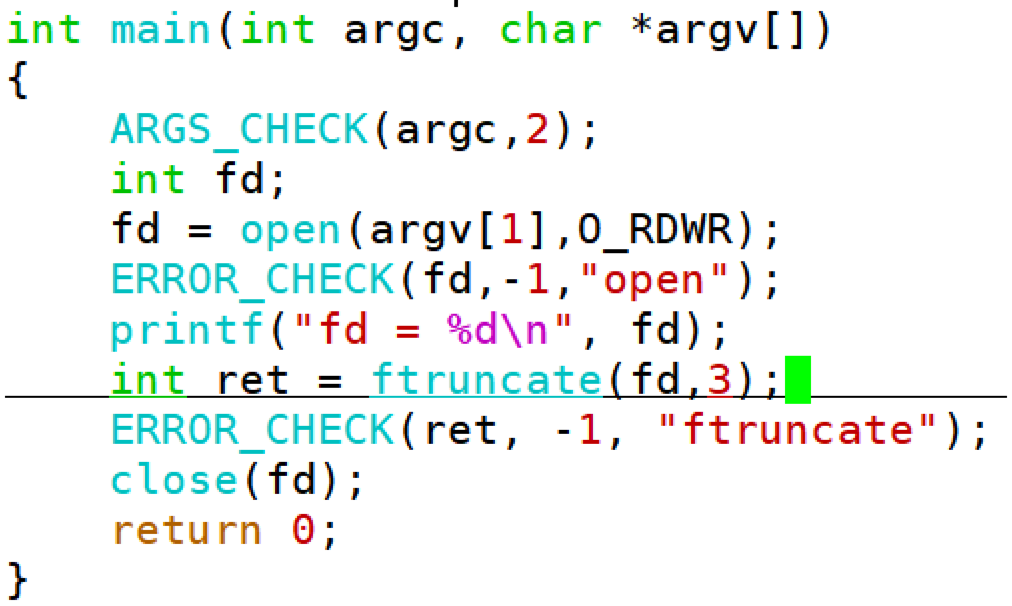
← 根据 read/write ... (系统调用) 的次数

```
while (read(fd, buf, size))
{

}
```

buf 缓冲区不要弄得太小.

ftruncate   规定文件大小

原文件大，截断．原文件小，补0．

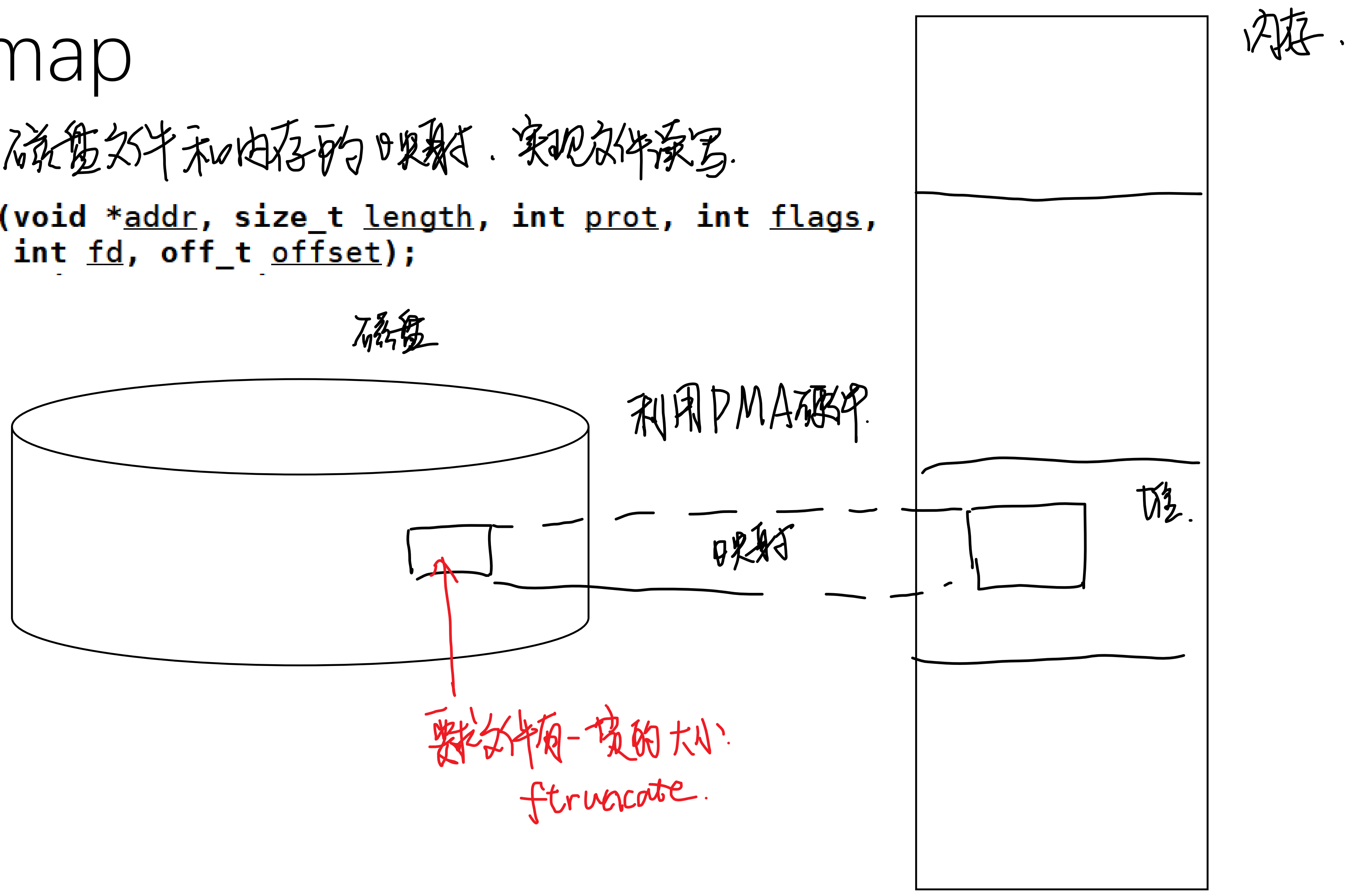


```
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd;
    fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n", fd);
    int ret = ftruncate(fd,3);
    ERROR_CHECK(ret, -1, "ftruncate");
    close(fd);
    return 0;
}
```

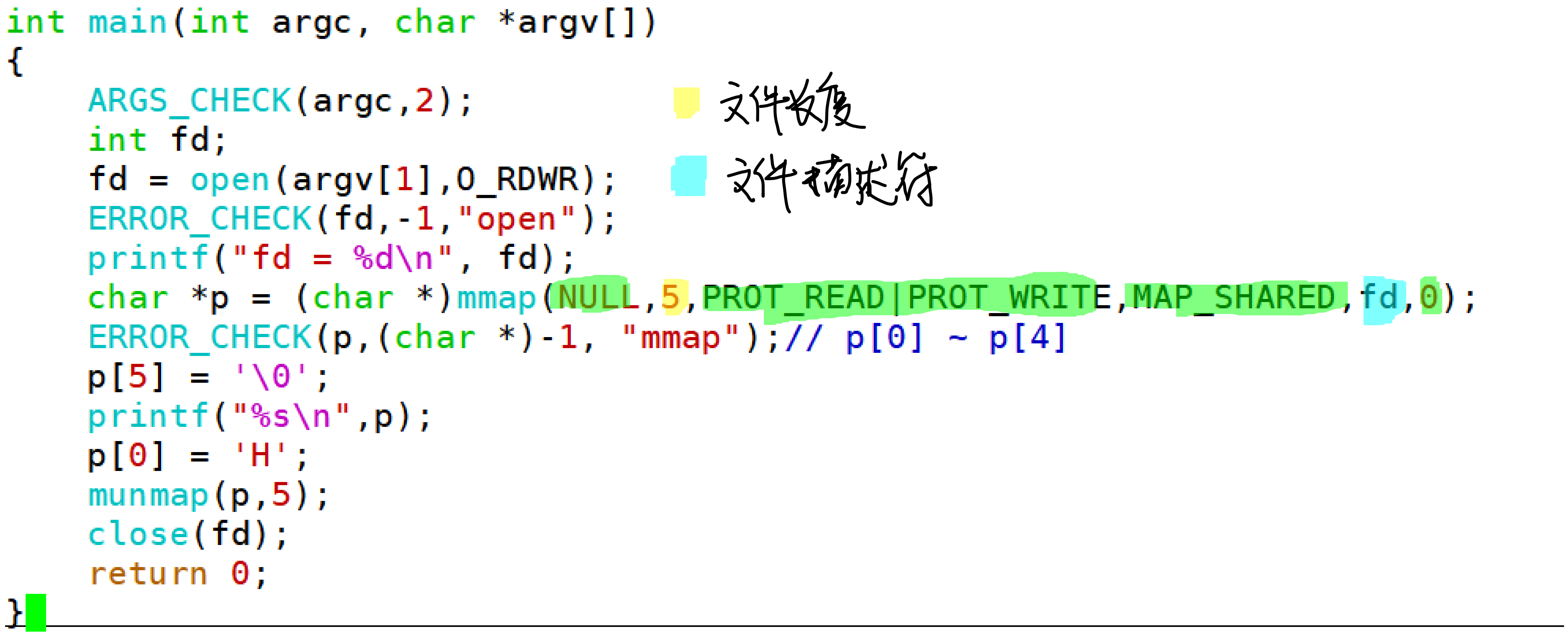
mmap

建立磁盘文件和内存的映射. 实现文件读写.

```
void *mmap(void *addr, size_t length, int prot, int flags,
             int fd, off_t offset);
```



要求文件有一定的大小:
ftruncate.



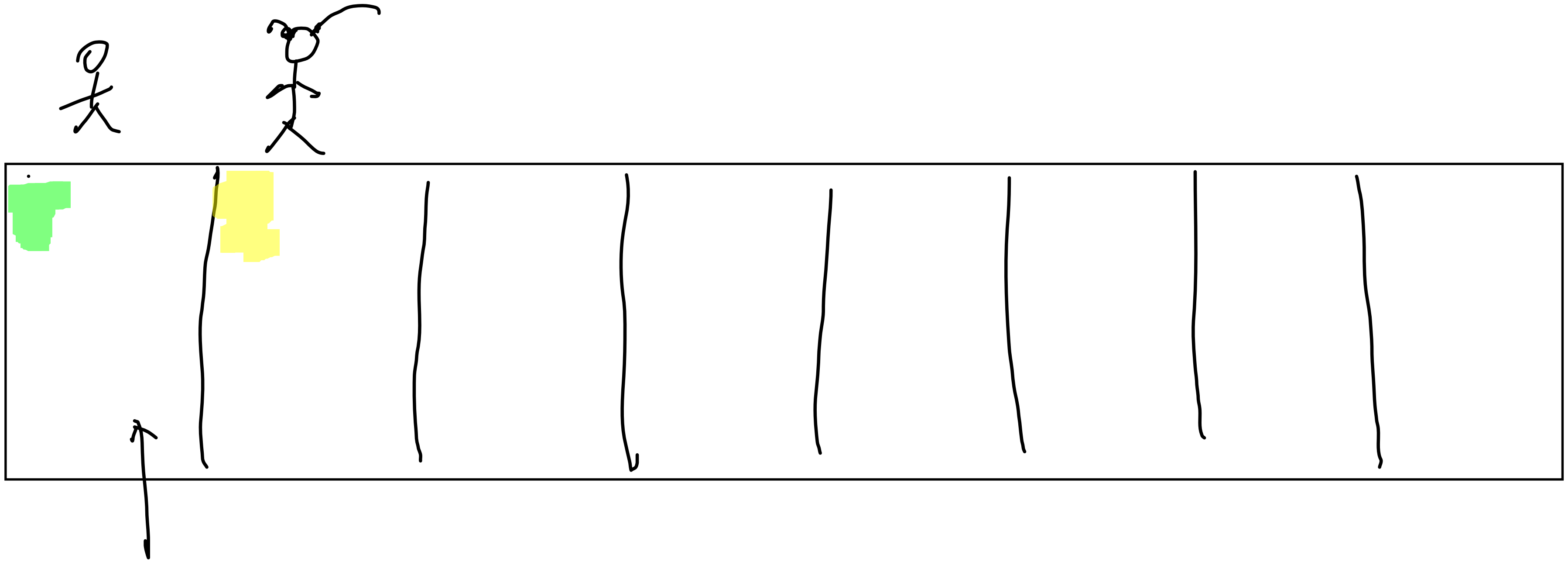
```c
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd;
    fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n", fd);
    char *p = (char *)mmap(NULL,5,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
    ERROR_CHECK(p,(char *)-1, "mmap");// p[0] ~ p[4]
    p[5] = '\0';
    printf("%s\n",p);
    p[0] = 'H';
    munmap(p,5);
    close(fd);
    return 0;
}
```

🟨 文件长度

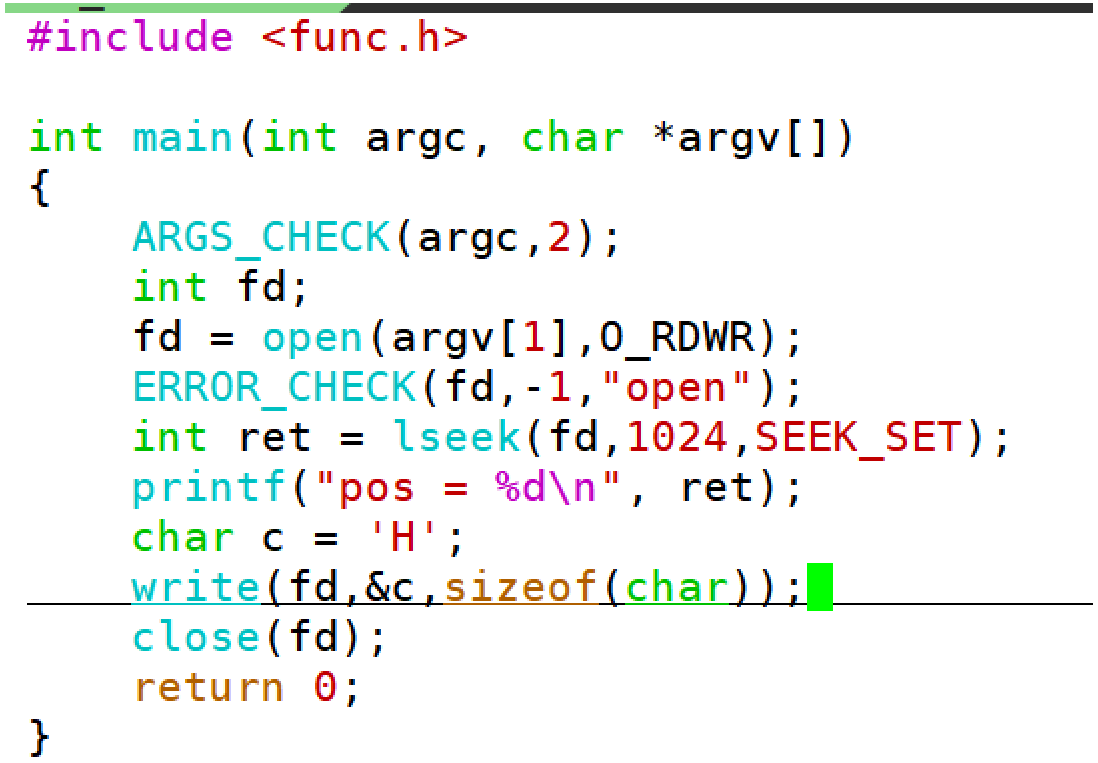
🟦 文件描述符

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



  SEEK_SET
         The file offset is set to offset bytes.

  SEEK_CUR
         The file offset is set to its current location plus offset bytes.

  SEEK_END
         The file offset is set to the size of the file plus offset bytes.
```

文件空洞

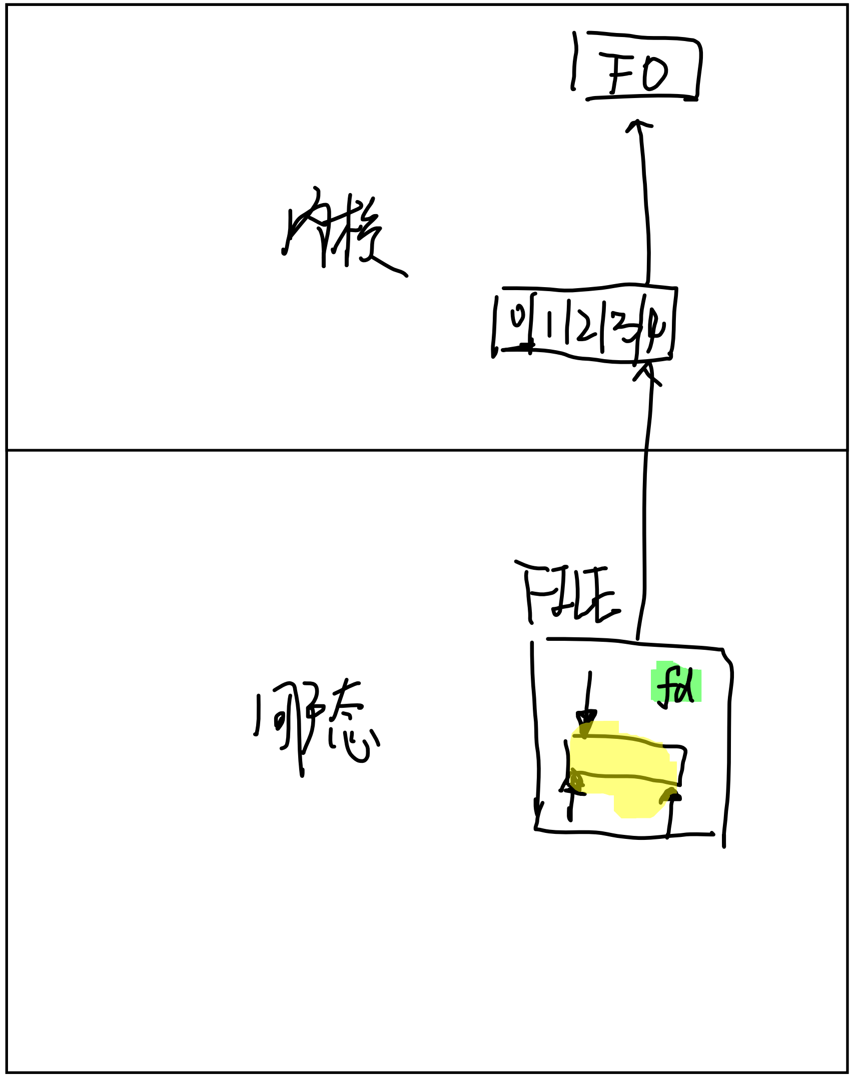


空洞 ： 多个用户同时写入.互不影响

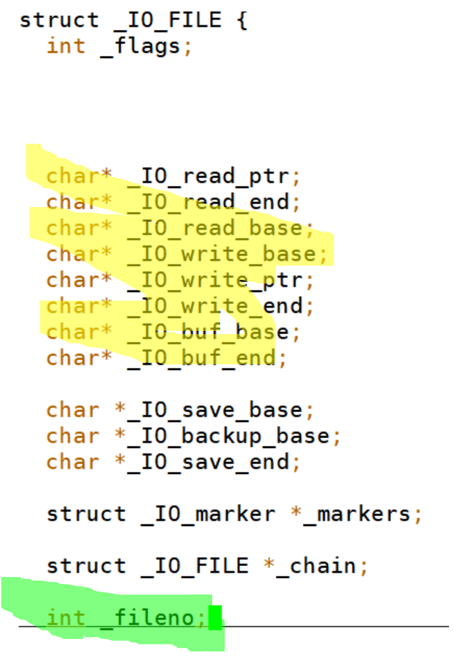


```c
#include <func.h>

int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd;
    fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    int ret = lseek(fd,1024,SEEK_SET);
    printf("pos = %d\n", ret);
    char c = 'H';
    write(fd,&c,sizeof(char));
    close(fd);
    return 0;
}
```

文件描述符和文件指针的关系

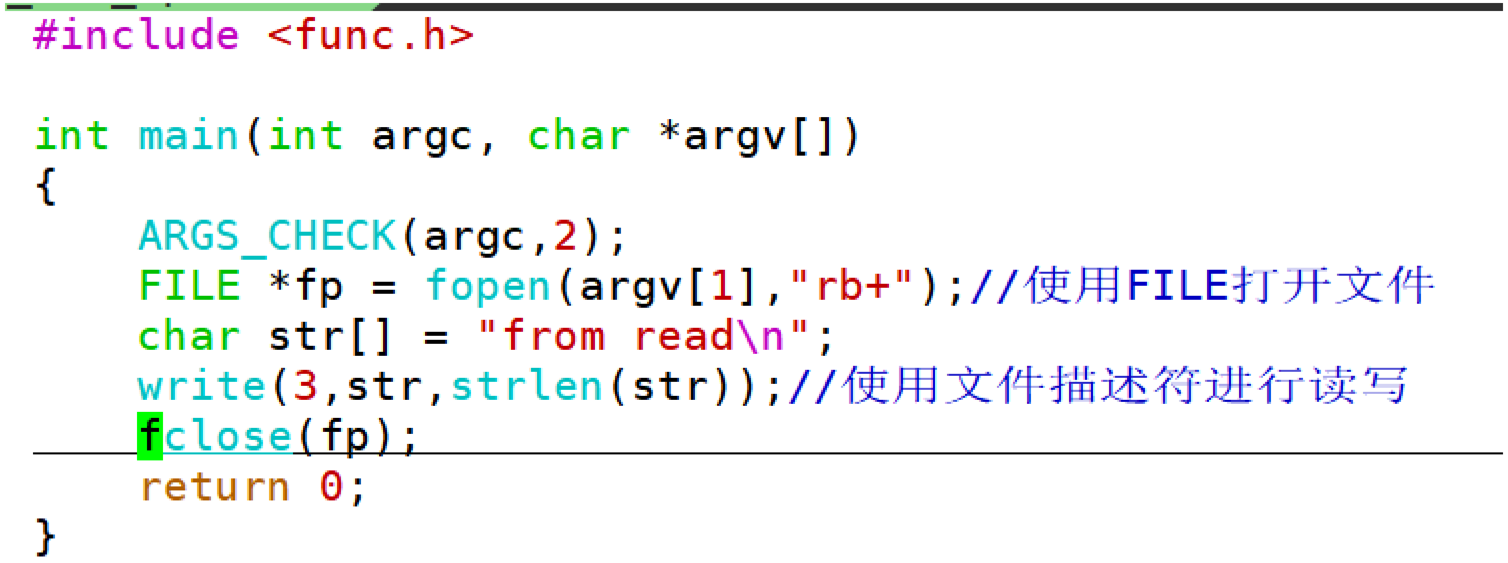


使用FILE 间接使用了fd.



```
struct _IO_FILE {
  int _flags;



  char* _IO_read_ptr;
  char* _IO_read_end;
  char* _IO_read_base;
  char* _IO_write_base;
  char* _IO_write_ptr;
  char* _IO_write_end;
  char* _IO_buf_base;
  char* _IO_buf_end;

  char *_IO_save_base;
  char *_IO_backup_base;
  char *_IO_save_end;

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
```

使用文件指针打开的文件，用描述符也能访问



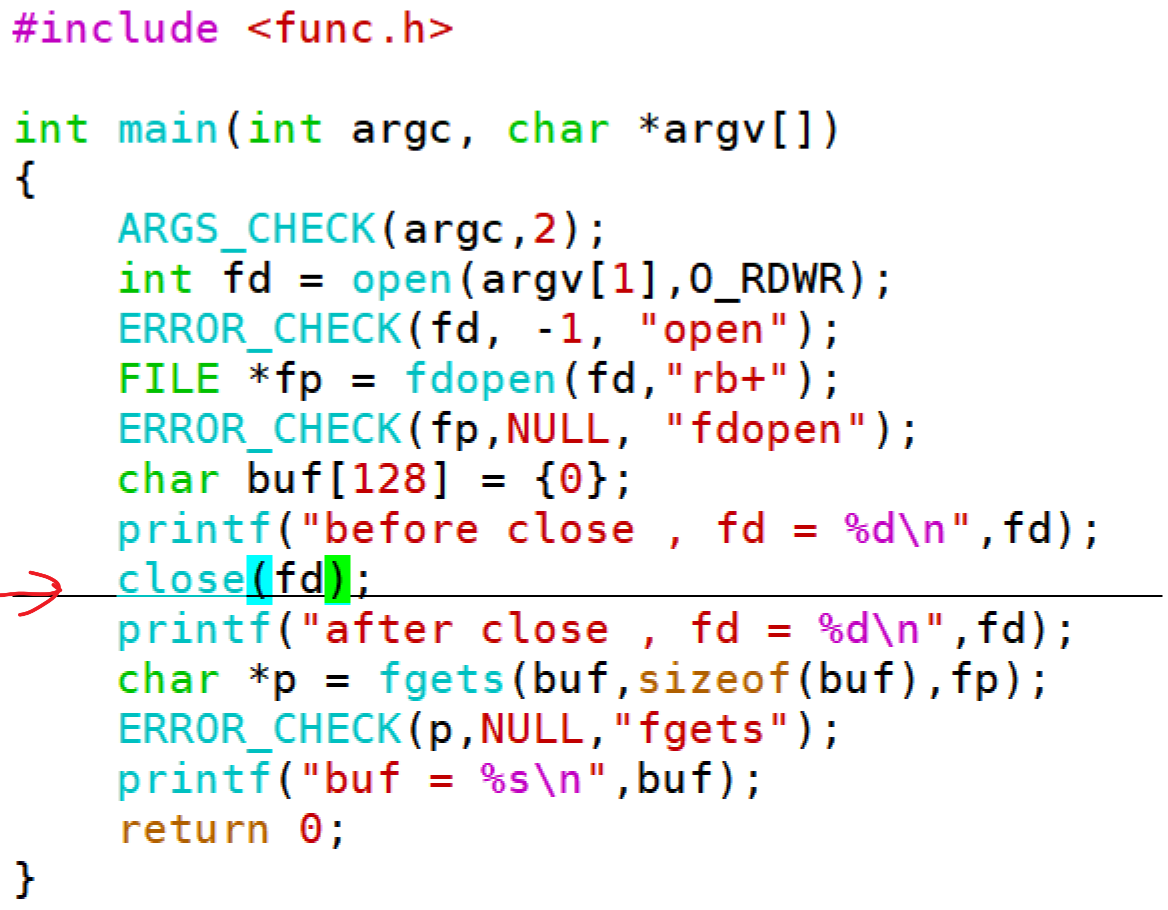
```
#include <func.h>

int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    FILE *fp = fopen(argv[1],"rb+");//使用FILE打开文件
    char str[] = "from read\n";
    write(3,str,strlen(str));//使用文件描述符进行读写
    fclose(fp);
    return 0;
}
```

说明FILE也使用了文件描述符.

文件描述符和文件指针的转换

```
FILE *fdopen(int fd, const char *mode);
```

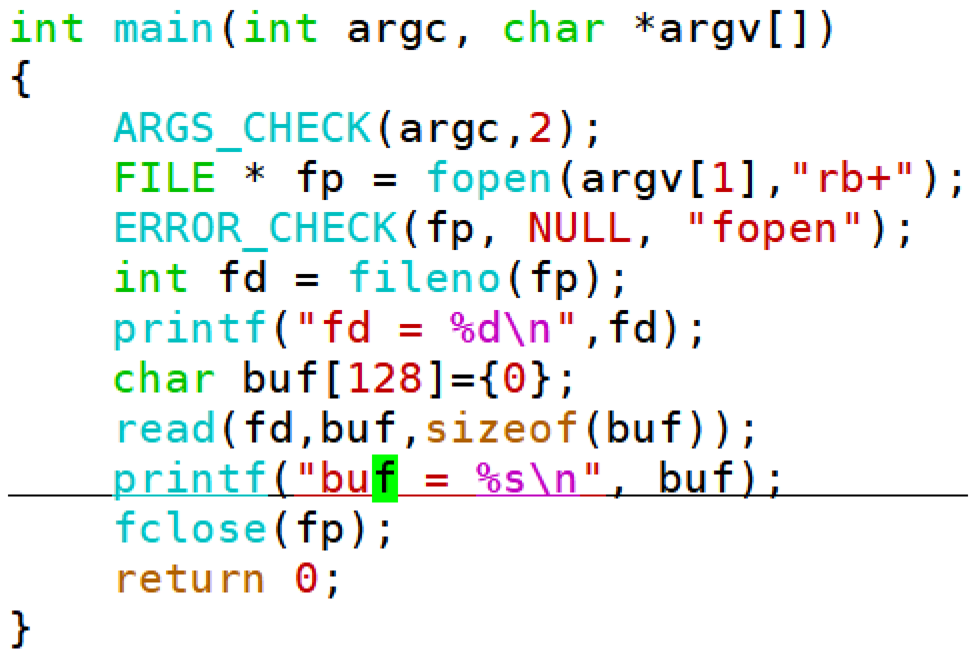
根据文件描述符，创建文件缓冲区



```
#include <func.h>

int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd, -1, "open");
    FILE *fp = fdopen(fd,"rb+");
    ERROR_CHECK(fp,NULL, "fdopen");
    char buf[128] = {0};
    printf("before close , fd = %d\n",fd);
    close(fd);
    printf("after close , fd = %d\n",fd);
    char *p = fgets(buf,sizeof(buf),fp);
    ERROR_CHECK(p,NULL,"fgets");
    printf("buf = %s\n",buf);
    return 0;
}
```

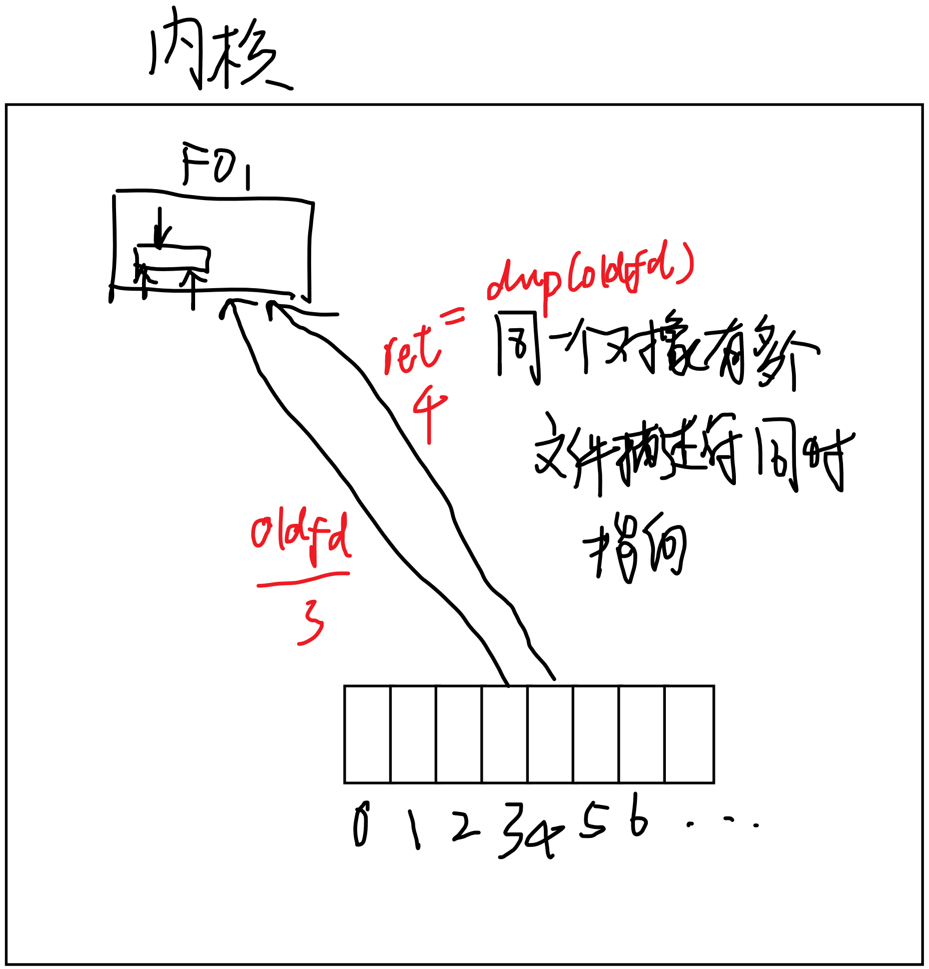
fd内容不变

```
int fileno(FILE *stream);
```



```c
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    FILE * fp = fopen(argv[1],"rb+");
    ERROR_CHECK(fp, NULL, "fopen");
    int fd = fileno(fp);
    printf("fd = %d\n",fd);
    char buf[128]={0};
    read(fd,buf,sizeof(buf));
    printf("buf = %s\n", buf);
    fclose(fp);
    return 0;
}
```

文件描述符的复制

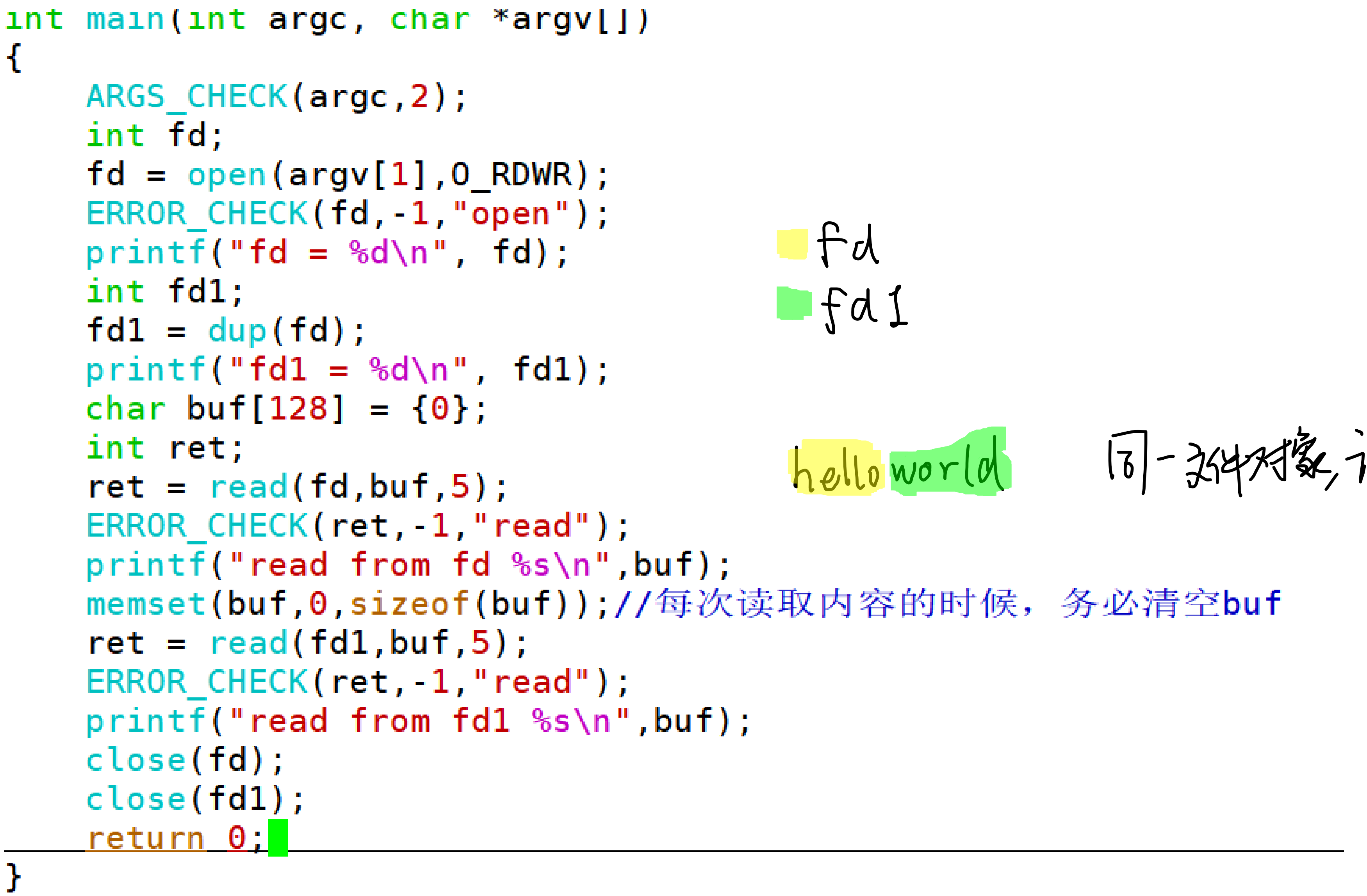


同一个对象有多个文件描述符同时指向

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

返回最小可用的fd.

用户指定newfd.



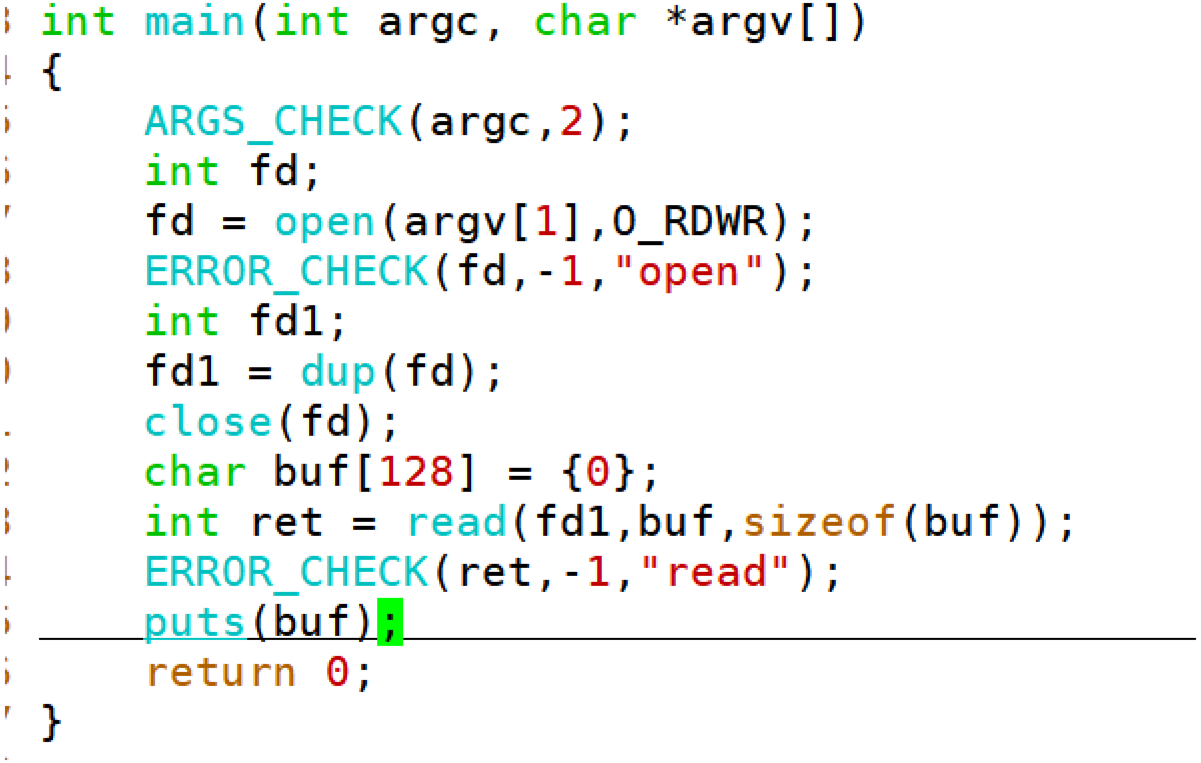
```c
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd;
    fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("fd = %d\n", fd);
    int fd1;
    fd1 = dup(fd);
    printf("fd1 = %d\n", fd1);
    char buf[128] = {0};
    int ret;
    ret = read(fd,buf,5);
    ERROR_CHECK(ret,-1,"read");
    printf("read from fd %s\n",buf);
    memset(buf,0,sizeof(buf));//每次读取内容的时候，务必清空buf
    ret = read(fd1,buf,5);
    ERROR_CHECK(ret,-1,"read");
    printf("read from fd1 %s\n",buf);
    close(fd);
    close(fd1);
    return 0;
}
```

🟨 fd

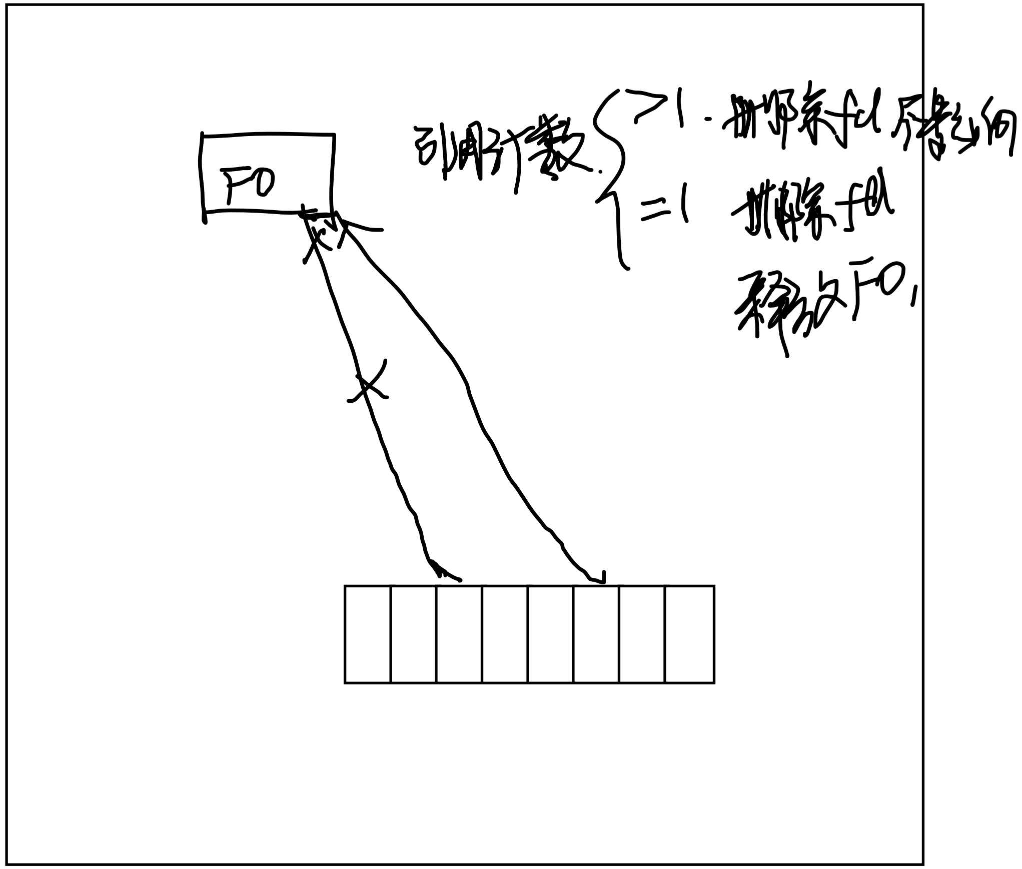
🟩 fd1

helloworld

同一文件对象，读写位置是共享的.



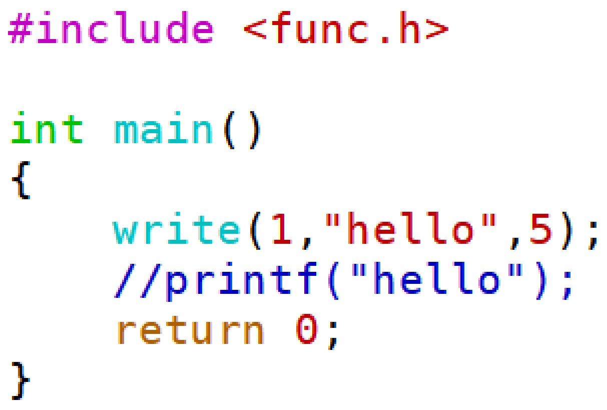
```
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd;
    fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    int fd1;
    fd1 = dup(fd);
    close(fd);
    char buf[128] = {0};
    int ret = read(fd1,buf,sizeof(buf));
    ERROR_CHECK(ret,-1,"read");
    puts(buf);
    return 0;
}
```



标准输出

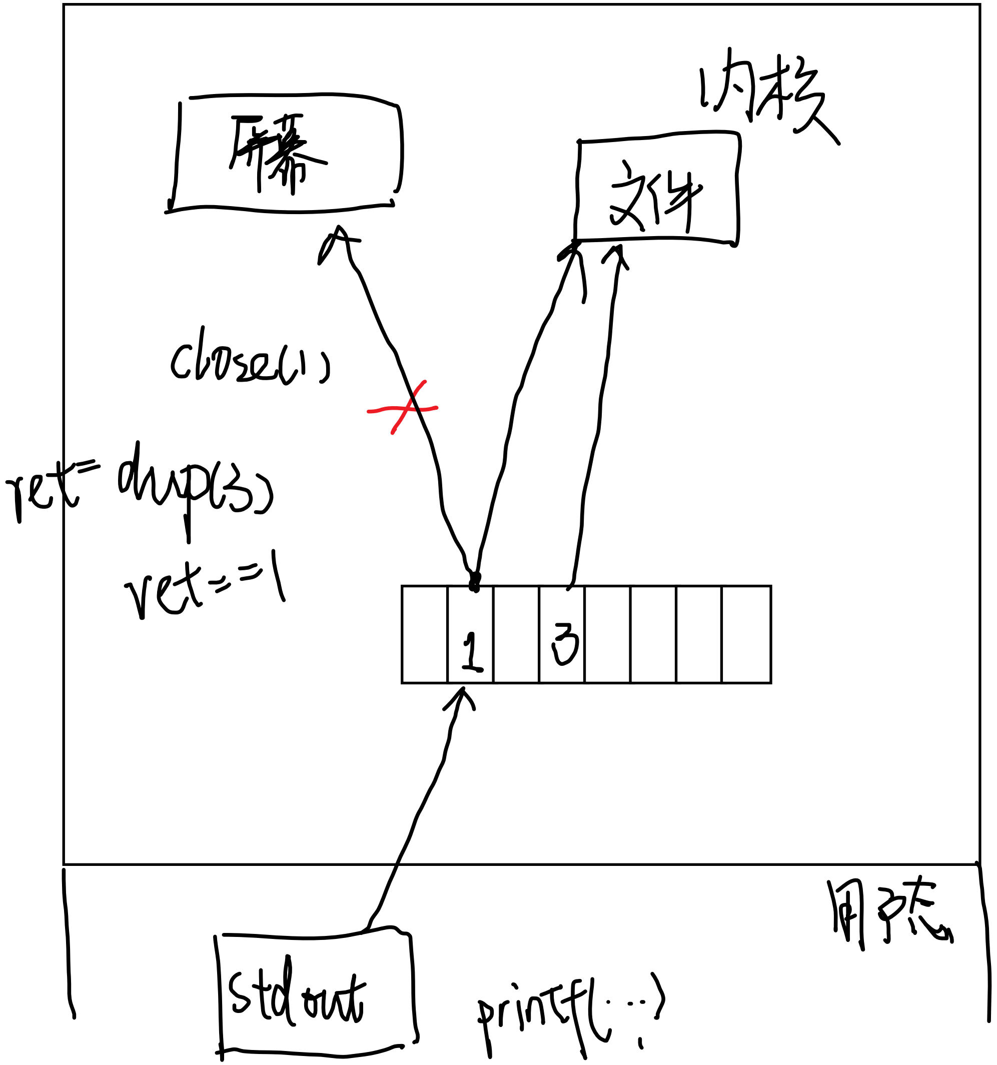
stdout FILE*

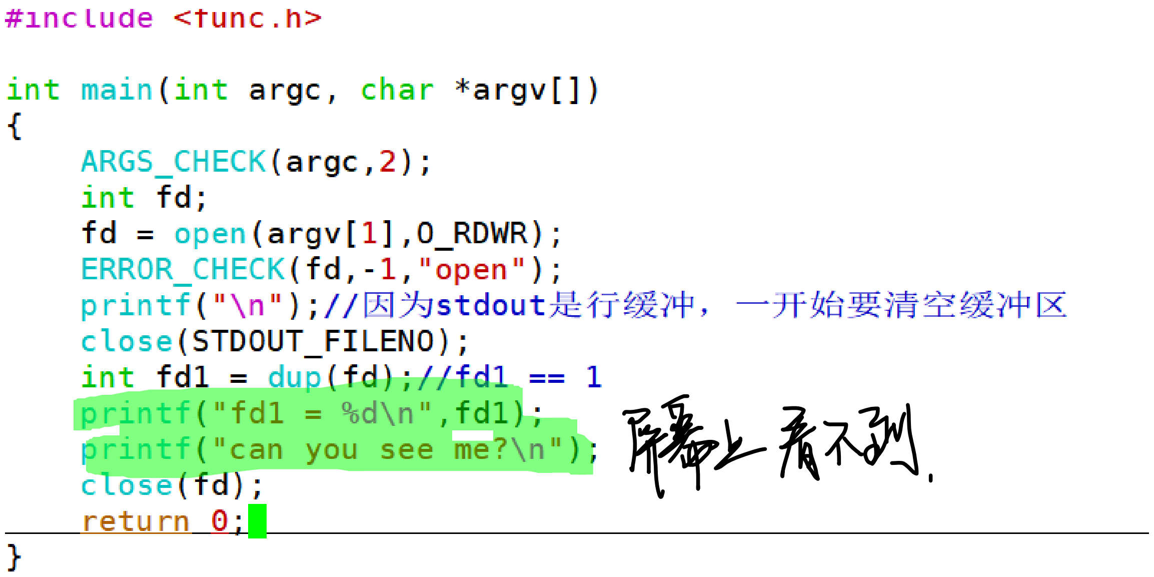
fileno(1).



```
#include <func.h>

int main()
{
    write(1,"hello",5);
    //printf("hello");
    return 0;
}
```

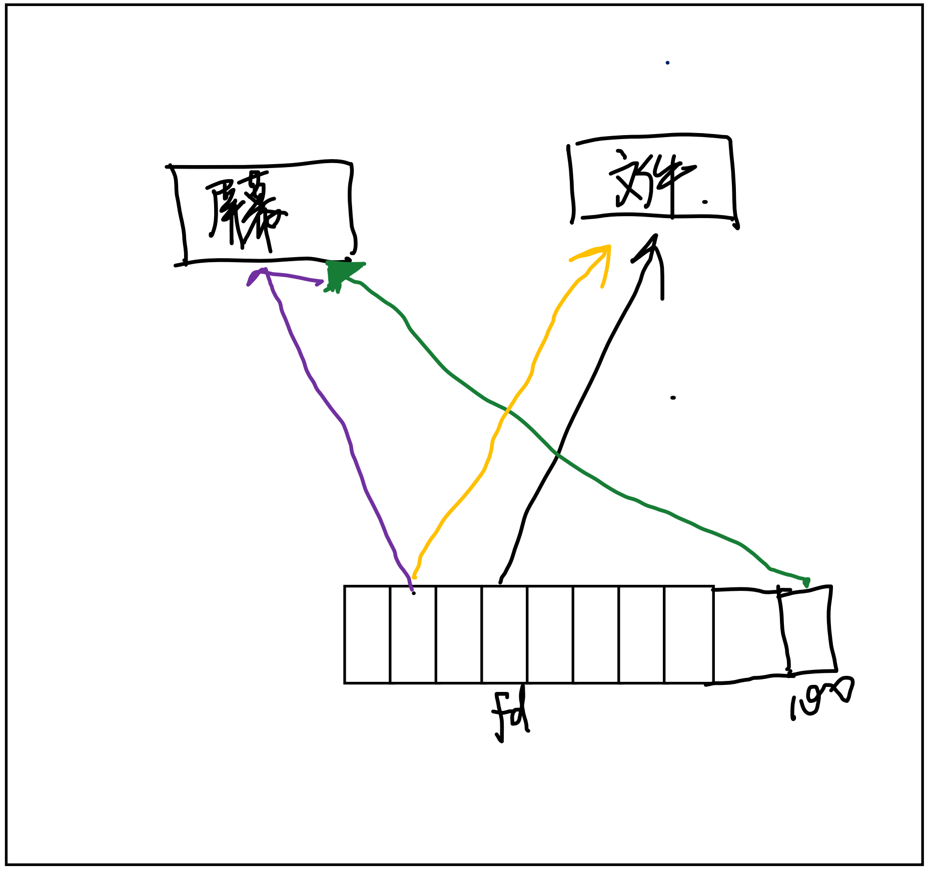
重定向

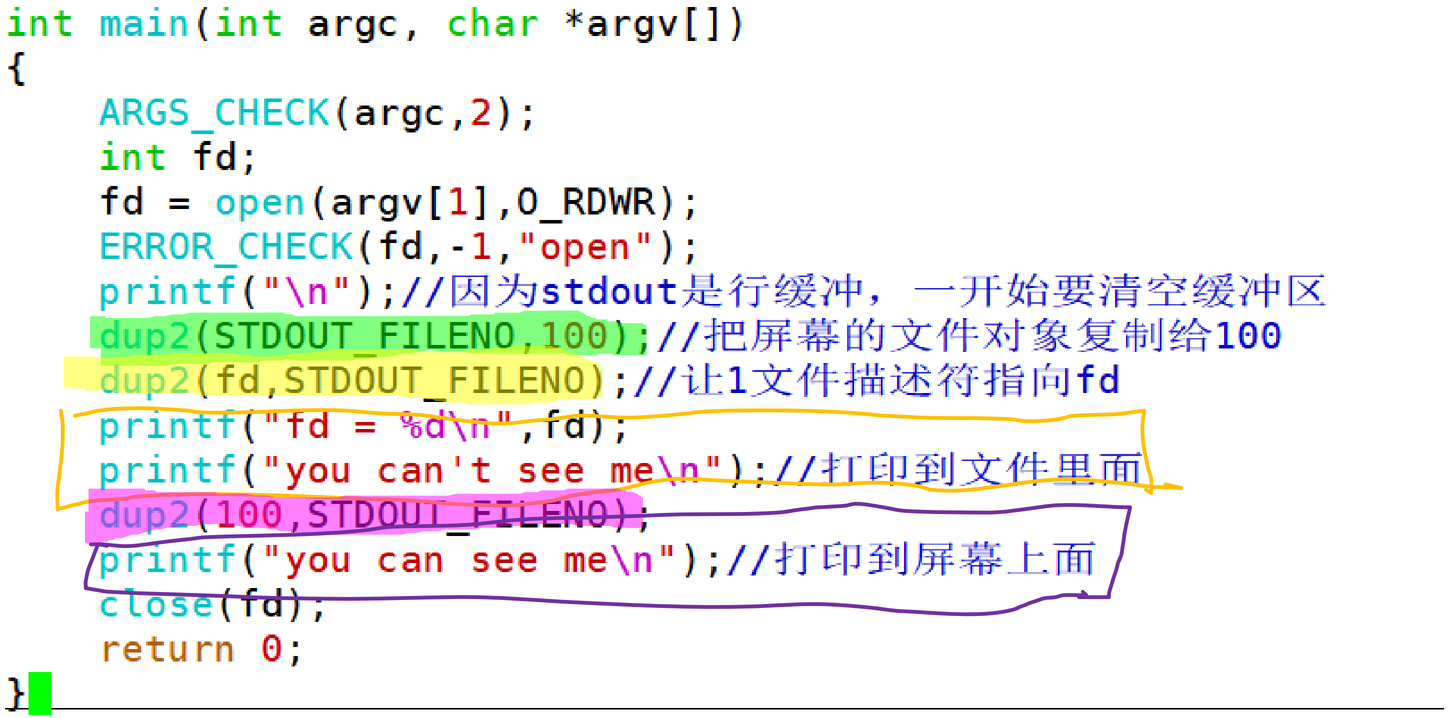




```
#include <func.h>

int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd;
    fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("\n");//因为stdout是行缓冲，一开始要清空缓冲区
    close(STDOUT_FILENO);
    int fd1 = dup(fd);//fd1 == 1
    printf("fd1 = %d\n",fd1);
    printf("can you see me?\n");
    close(fd);
    return 0;
}
```

屏幕上看不到.





```
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fd;
    fd = open(argv[1],O_RDWR);
    ERROR_CHECK(fd,-1,"open");
    printf("\n");//因为stdout是行缓冲，一开始要清空缓冲区
    dup2(STDOUT_FILENO,100);//把屏幕的文件对象复制给100
    dup2(fd,STDOUT_FILENO);//让1文件描述符指向fd
    printf("fd = %d\n",fd);
    printf("you can't see me\n");//打印到文件里面
    dup2(100,STDOUT_FILENO);
    printf("you can see me\n");//打印到屏幕上面
    close(fd);
    return 0;
}
```

dup：选择最小未使用的fd

dup2：选择传入的newfd，newfd原来的指向会关闭

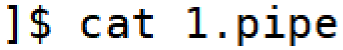
有名管道（named pipe / FIFO)

传输方式	含义
全双工	双方可以同时向另一方发送数据
半双工	双方可以向另一方发送数据，不能同时
单工	永远只能一方向另一方发送数据

ls -l 显示p

管道.

$mkfifo

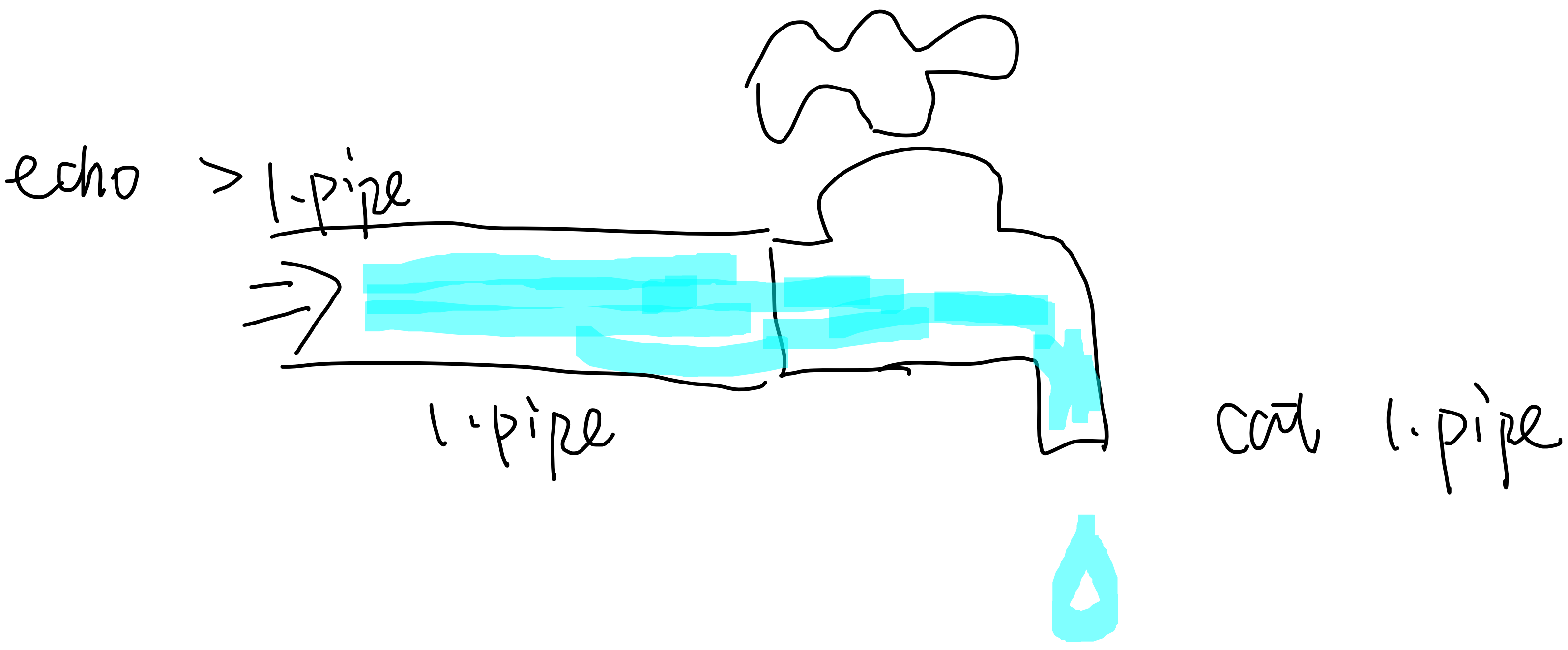


```
]$ cat 1.pipe
```

打开管道读端

① 管道不能存储数据.

② 管道不能打开. 不能cp.

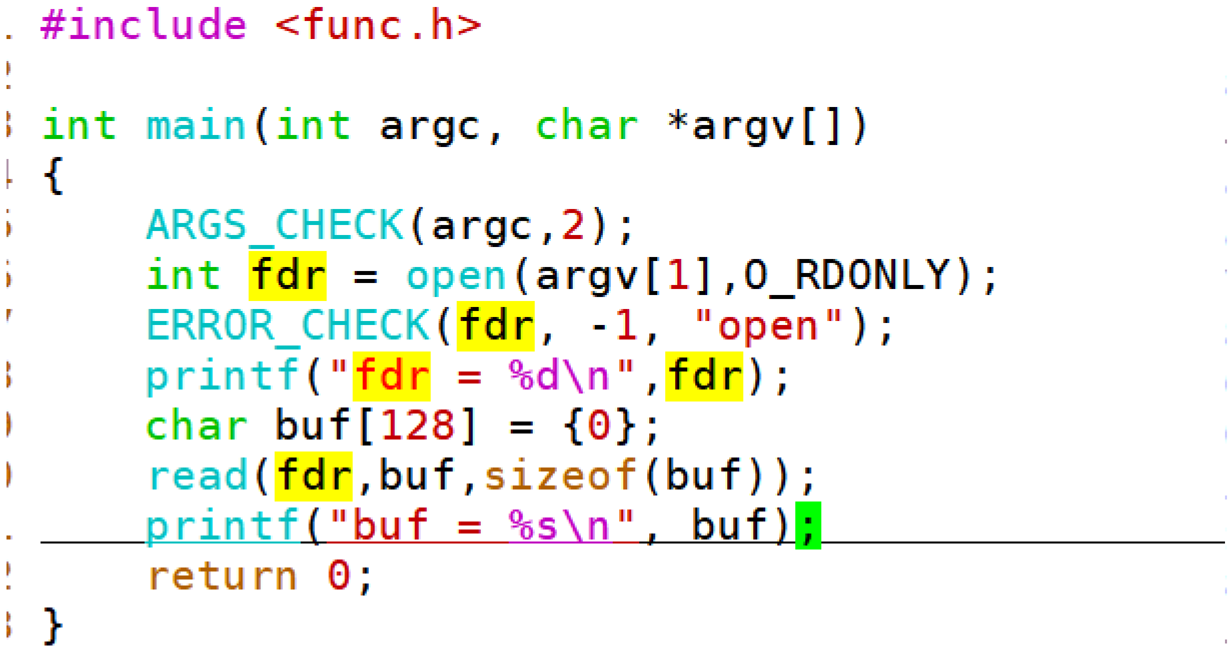


echo >1.pipe
1.pipe
cat 1.pipe

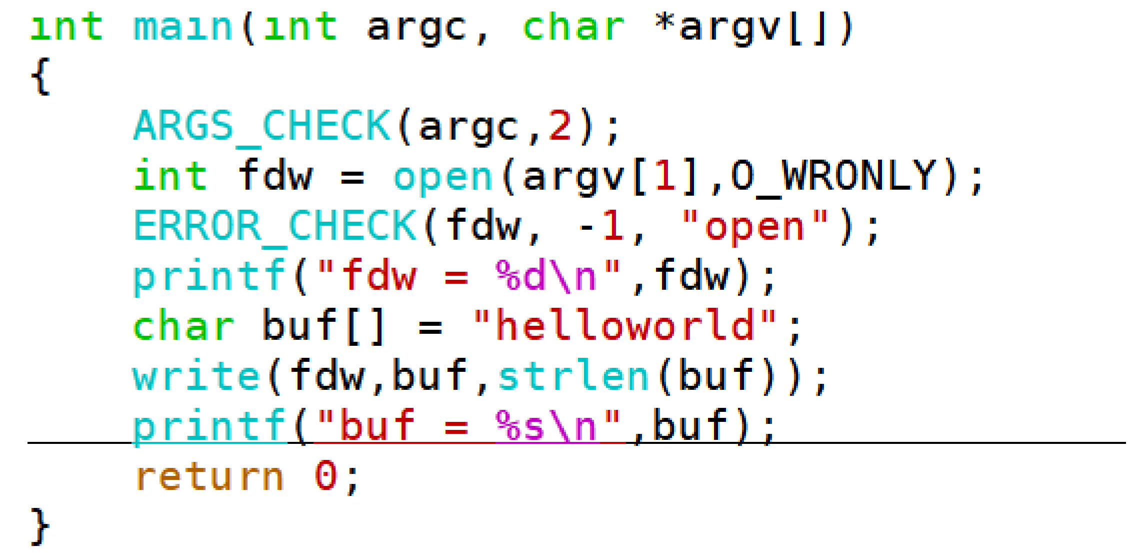
open("1.pipe", O_RDONLY); ⇒ 打开管道的读端

open("1.pipe", O_WRONLY); ⇒ 打开管道的写端

如果只打开一端，程序运行就会阻塞

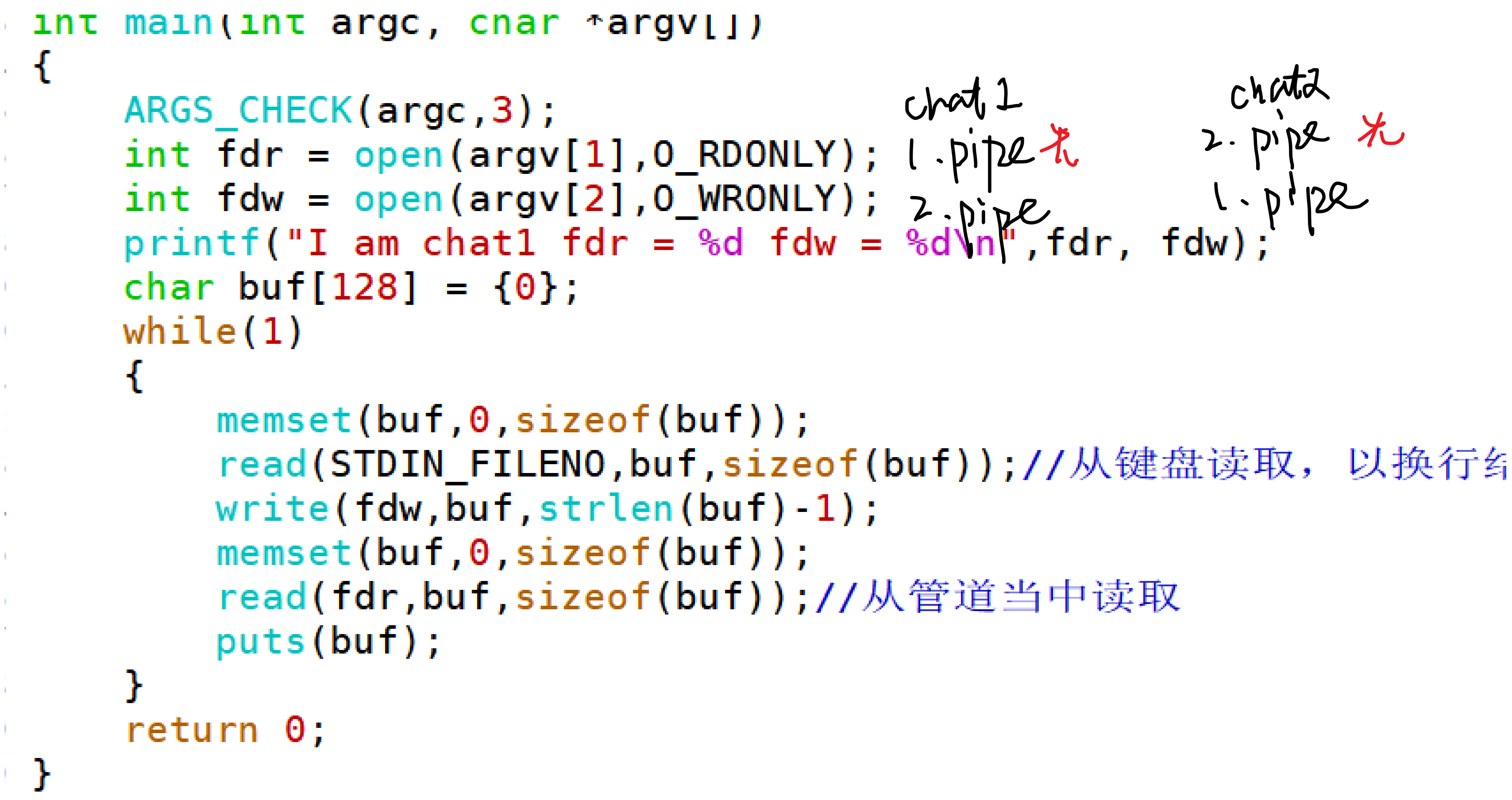


```c
#include <func.h>

int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fdr = open(argv[1],O_RDONLY);
    ERROR_CHECK(fdr, -1, "open");
    printf("fdr = %d\n",fdr);
    char buf[128] = {0};
    read(fdr,buf,sizeof(buf));
    printf("buf = %s\n", buf);
    return 0;
}
```



```c
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,2);
    int fdw = open(argv[1],O_WRONLY);
    ERROR_CHECK(fdw, -1, "open");
    printf("fdw = %d\n",fdw);
    char buf[] = "helloworld";
    write(fdw,buf,strlen(buf));
    printf("buf = %s\n",buf);
    return 0;
}
```

两根管道实现全双工

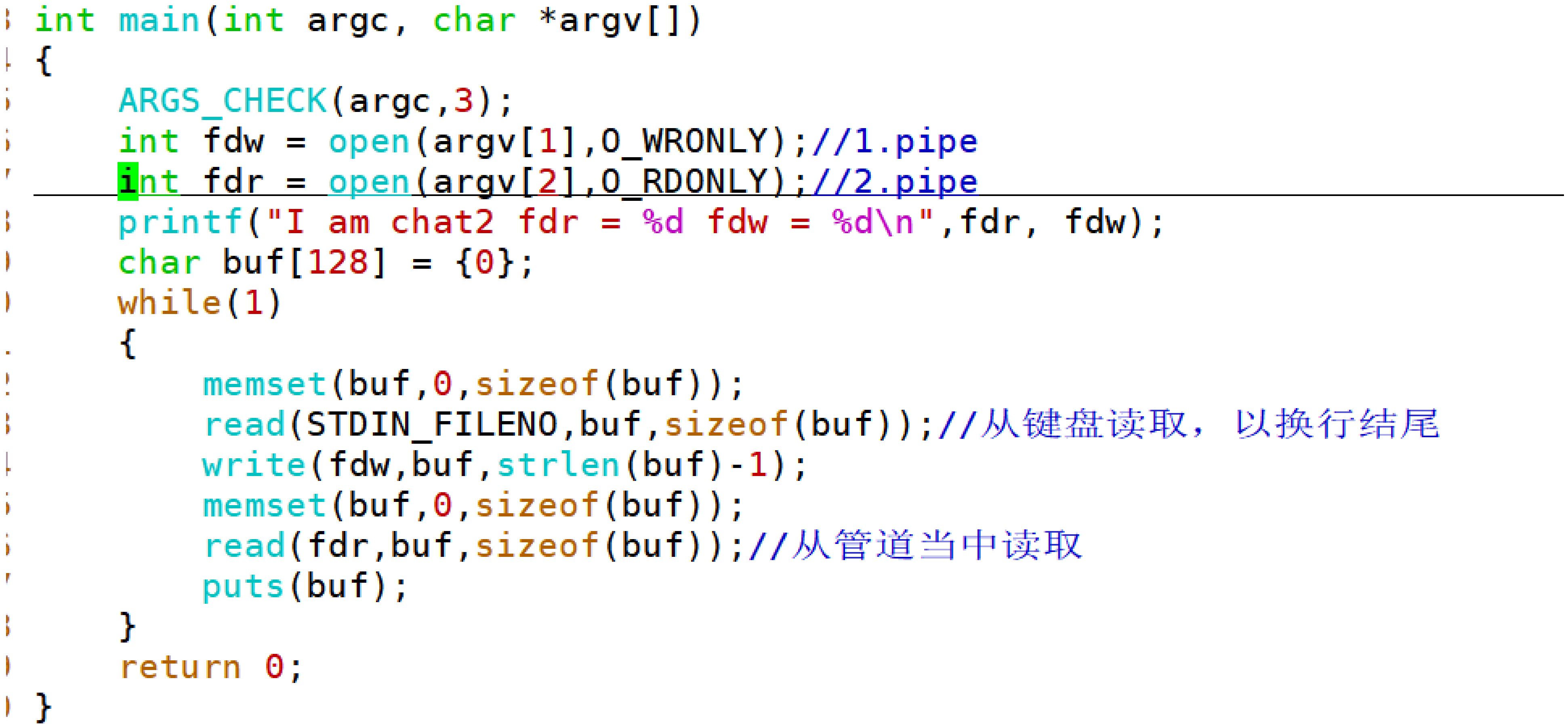


```
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,3);
    int fdr = open(argv[1],O_RDONLY);
    int fdw = open(argv[2],O_WRONLY);
    printf("I am chat1 fdr = %d fdw = %d\n",fdr, fdw);
    char buf[128] = {0};
    while(1)
    {
        memset(buf,0,sizeof(buf));
        read(STDIN_FILENO,buf,sizeof(buf));//从键盘读取， 以换行纟
        write(fdw,buf,strlen(buf)-1);
        memset(buf,0,sizeof(buf));
        read(fdr,buf,sizeof(buf));//从管道当中读取
        puts(buf);
    }
    return 0;
}
```

chat1
1.pipe 先
2.pipe

chat2
2.pipe 先
1.pipe

chat2的代码



```
int main(int argc, char *argv[])
{
    ARGS_CHECK(argc,3);
    int fdw = open(argv[1],O_WRONLY);//1.pipe
    int fdr = open(argv[2],O_RDONLY);//2.pipe
    printf("I am chat2 fdr = %d fdw = %d\n",fdr, fdw);
    char buf[128] = {0};
    while(1)
    {
        memset(buf,0,sizeof(buf));
        read(STDIN_FILENO,buf,sizeof(buf));//从键盘读取，以换行结尾
        write(fdw,buf,strlen(buf)-1);
        memset(buf,0,sizeof(buf));
        read(fdr,buf,sizeof(buf));//从管道当中读取
        puts(buf);
    }
    return 0;
}
```