1017文件描述符

```c
int main()
{
    //C语言
    printf("hello printf\n");
    fprintf(stdout,"hello fprintf\n");
    const char* s = "hello fputs\n";
    fputs(s,stdout);

    //系统
    const char *ss = "hello write\n";
    write(1,ss,strlen(ss));

    //在最后调用的fork -- 上面的函数已经被执行完了
    fork(); //创建子进程

    return 0;
}
```

这个是上节课的遗留问题

关于缓冲区的认识：

一般而言，行缓冲的设备文件 -- 显示器
一般而言，全换从的设备文件 -- 磁盘文件

所有设备，永远都倾向于全缓冲！
--- 缓冲区满了才刷新 --- 需要更少的IO操作 --- 更少次的外设访问 --- 提高效率！

当和外部设备进行IO的时候，数据量的大小不是主要矛盾，和外设预备IO的过程才是最耗费时间的

其他刷新策略是，结合具体情况做的妥协！

极端情况下，是可以自定义规则的！

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
yufc@VM-12-12-centos:~/bit/1017$ make
gcc -o myfile myfile.c
yufc@VM-12-12-centos:~/bit/1017$ ./myfile > log.txt
yufc@VM-12-12-centos:~/bit/1017$ cat log.txt
hello write
hello printf
hello fprintf
hello fputs
hello printf
hello fprintf
hello fputs
yufc@VM-12-12-centos:~/bit/1017$
```

为什么上节课代码是这个现象？

同样的一个程序，向显示器打印
输出4行文本

向普通文件（磁盘）打印的时候，变成了7行

```
//在最后调用的fork -- 上面的函数已经被执行完了
fork(); //创建子进程
```

走到这里的时候
函数已经执行完了
不代表数据已经刷新了！

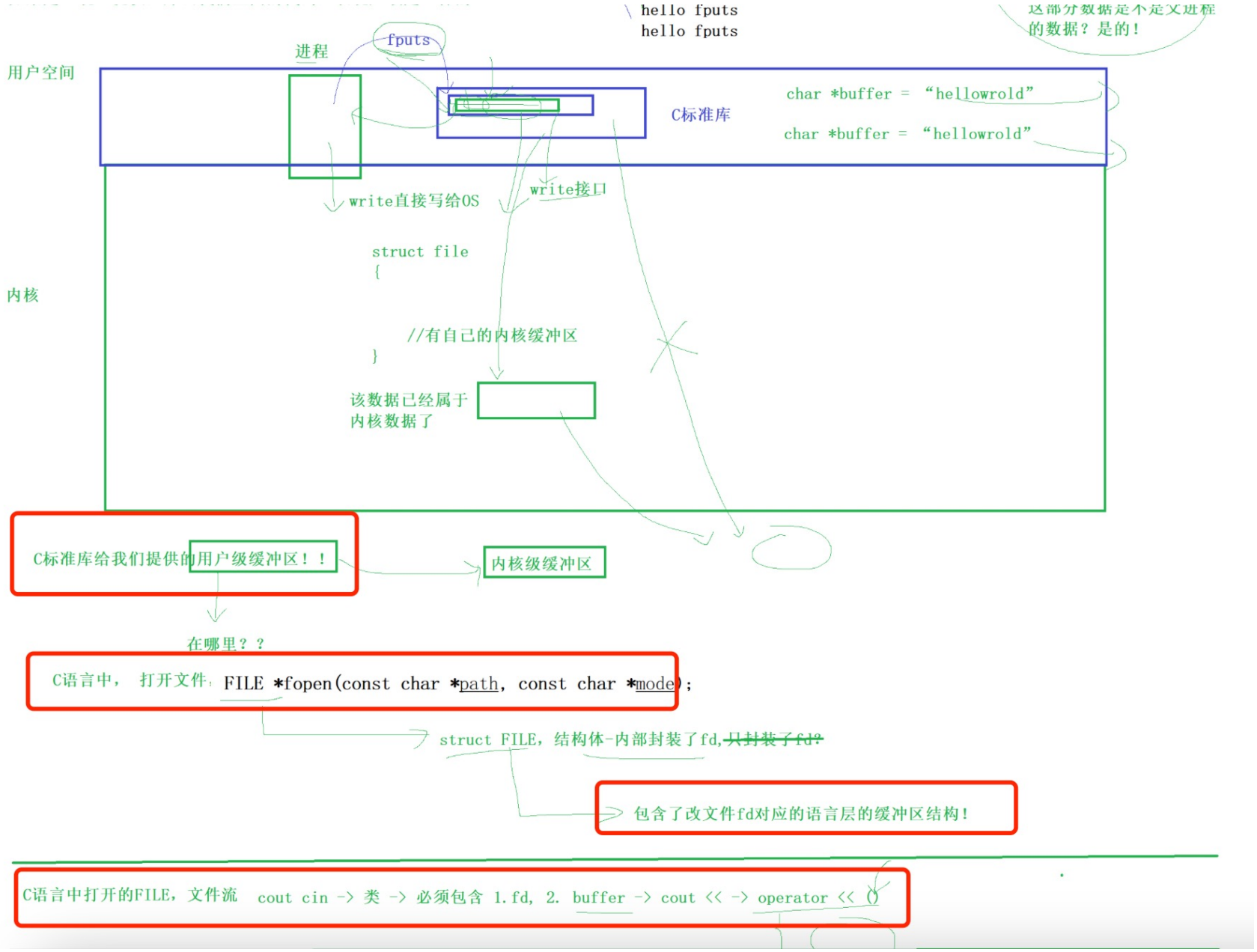
说到这里，
我们谈的缓冲区
都是C标准库提供的
用户级缓冲区

fork()什么时候是会拷贝一份？
写时拷贝

我们曾经所谈的“缓冲区”，绝对不是OS提供的！
因为如果是OS统一提供
那么上面的代码的结果应该是一样的。

1. **如果向显示器打印，刷新策略是行刷新**，那么最后执行fork的时候，一定一定是函数执行完了 && 数据已经刷新了
2. 如果对应的程序做了重定向，本质是向磁盘文件打印 --- **隐性的刷新策略变成了全缓冲！**此时代码的\n已经没有意义了，所以fork的时候，**函数执行完了，但是数据没有刷新！**现在数据在，**当前进程的C标准库中！**
3. 这部分数据，属不属于父进程的数据？肯定是的！fork之后，父子各自执行自己的退出。**进程退出是需要刷新缓冲区的！**
4. 那么现在的一个问题，**刷新这个动作，算不算“写”？**算的，从缓冲区刷新出去，相当于写到显示器里
5. **此时会有写时拷贝！**
6. 所以！C的接口会出现两份的数据！

这个用户级缓冲区在哪里？



1. 我们自己设计一下用户层缓冲区
2. Minishell支持重定向

```c
//我们自己设计一下用户层缓冲区
#define NUM 1024

struct MyFILE_
{
    int fd;
    char buffer[NUM];
    int end; //当前缓冲区的结尾
};
```

```c
void fflush_(MyFILE* fp)
{
    assert(fp);
    if(fp->end!=0)
    {
        //暂且认为刷新了 -- 其实是把数据写到内核里面了
        write(fp->fd,fp->buffer,fp->end);
        syncfs(fp->fd); //讲数据写入到文件中（磁盘...）
        fp->end = 0;
    }
}
```

```c
MyFILE* fopen_(const char *pathname, const char* mode)
{
    assert(pathname);
    assert(mode);
    MyFILE* fp = NULL;//如果传入的mode有问题 -- 返回NULL

    if(strcmp(mode,"r") == 0){}
    else if(strcmp(mode,"r+") == 0){}
    else if(strcmp(mode,"w") == 0)
    {

        int fd = open(pathname,O_WRONLY|O_TRUNC|O_CREAT,0666); //封装系统调用
        if(fd >= 0) //成功打开
        {
            fp = (MyFILE*)malloc(sizeof(MyFILE));
            memset(fp,0,sizeof(MyFILE));
            fp->fd = fd;
        }
    }
    else if(strcmp(mode,"w+") == 0){}
    else if(strcmp(mode,"a") == 0){}
    else if(strcmp(mode,"a+") == 0){}
    else
    {
        //什么都不做
        ;
    }
    return fp;
}

void fputs_(const char* message, MyFILE*fp)
{
    assert(message);
    assert(fp);

    //向缓冲区中写入
    strcpy(fp->buffer+fp->end,message);
    fp->end += strlen(message);//把end往后移

    printf("%s\n",fp->buffer);//for debug

    //现在暂时没有刷新
}
```

我们看到下面的结果，
确实字符串长度在不断变长，
因为没有刷新

```c
void fclose_(MyFILE* fp)
{
    assert(fp);
    fflush_(fp);
    close(fp->fd);
    free(fp);
}

int main()
{
    MyFILE* fp = fopen_("./log.txt","w");
    if(fp == NULL)
    {
        printf("fopen_ error");
        return 1;
    }
    //进行一些操作
    fputs_("one: hello world",fp);
    fputs_("    two: hello world",fp);
    fputs_("    three: hello world",fp);
    fputs_("    four: hello world",fp);
    fclose_(fp);
    return 0;
```

```
yufc@VM-12-12-centos:~/bit/1017$ ./myfile
one: hello world
one: hello world    two: hello world
one: hello world    two: hello world    three: hello world
one: hello world    two: hello world    three: hello world    four: hello world
yufc@VM-12-12-centos:~/bit/1017$
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                    bash - 1017

yufc@VM-12-12-centos:~/bit/1017$ cat log.txt
one: hello world    two: hello world    three: hello world    four: hello worldyufc@VM-12-12-centos:~/bit/1017
$
```

最后关闭文件的时候
把最终的
缓冲区里面的内容
刷新了

现在，我们要来定义一下刷新到策略

```c
void fputs_(const char* message, MyFILE*fp)
{
    assert(message);
    assert(fp);

    //向缓冲区中写入
    strcpy(fp->buffer+fp->end,message);
    fp->end += strlen(message);//把end往后移

    printf("%s\n",fp->buffer);//for debug

    //现在暂时没有刷新
    //现在要定义一下刷新策略
    //此时就要分类一下了
    if(fp->fd == 0)
    {
        //标准输入
    }
    else if(fp->fd == 1)
    {
        //标准输出
        if(fp->buffer[fp->end-1] == '\n')//如果有回车 -- 直接刷新
        {
            write(fp->fd, fp->buffer, fp->end);
        }
    }
    else if(fp->fd == 2)
    {
        //标准错误
    }
    else
```