

# sys I/O

杨子然 元培学院

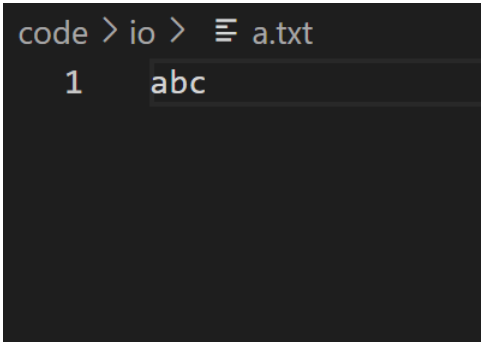
## Unix I/O

- stat/ fstat
  - 这是根据文件名/文件描述符来获取文件元数据的函数（教材课后习题有涉及）
  - 依赖平台，所以stdio库中没有(`#include <sys/stat.h>`)
- 没有缓冲区，不close也会直接写入，这一点我们之后会展开说明

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

int main(){
    int fp = open("./a.txt",O_WRONLY);
    char ch[] = "abc";
    write(fp, ch, 3);

    while(1);
    return 0 ;
}
```



```
code > io > ≡ a.txt
1  abc
```

## 目录文件

- Linux的文件和目录的权限，只有RWX三种。
- r(Read, 读取)：对文件而言，具有读取文件内容的权限；对目录来说，具有浏览目录的权限。
- w(Write, 写入)：对文件而言，具有新增,修改,删除文件内容的权限；对目录来说，具有新建，删除，修改，移动目录内文件的权限。
- x(execute, 执行)：对文件而言，具有执行文件的权限；对目录来说该用户具有进入目录的权限。

- 1、目录的只读访问不允许使用cd进入目录，必须要有执行的权限才能进入。（把cd看做一种执行？）
- 2、只有执行权限只能进入目录，不能看到目录下的内容，要想看到目录下的文件名和目录名，需要可读权限。
- 3、一个文件能不能被删除，主要看该文件所在的目录对用户是否具有写权限，如果目录对用户没有写权限，则该目录下的所有文件都不能被删除，文件所有者除外
- 4、目录文件的w位不设置，即使你拥有目录中某文件的w权限也不能写该文件

## 二进制文件和文本文件

- 1.在windows系统中，文本模式下，文件以"\r\n"代表换行。若以文本模式打开文件，并用fputs等函数写入换行符"\n"时，函数会自动在"\n"前面加上"\r"。即实际写入文件的是"\r\n"。
- 2.在类Unix/Linux系统中文本模式下，文件以"\n"代表换行。所以Linux系统中在文本模式和二进制模式下并无区别。

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

FILE * fp1 = fopen("a.txt", "r");
FILE * fp2 = fopen("a.txt", "rb");// in Linux, same
```

# 关于文件描述符

- 形式上是一个非负整数
- 实际上，它是一个索引值，指向内核为每一个进程所维护的该进程打开文件的记录表。

| 整数值 | 名称              | <unistd.h>符号常量 | <stdio.h>文件流 |
|-----|-----------------|----------------|--------------|
| 0   | Standard input  | STDIN_FILENO   | stdin        |
| 1   | Standard output | STDOUT_FILENO  | stdout       |
| 2   | Standard error  | STDERR_FILENO  | stderr       |

\* 10.6 下面程序的输出是什么？

```
1  #include "csapp.h"
2
3  int main()
4  {
5      int fd1, fd2;
6
7      fd1 = Open("foo.txt", O_RDONLY, 0);
8      fd2 = Open("bar.txt", O_RDONLY, 0);
9      Close(fd2);
10     fd2 = Open("baz.txt", O_RDONLY, 0);
11     printf("fd2 = %d\n", fd2);
12     exit(0);
13 }
```

1.若成功打开“foo.txt”：

- 1.1若成功打开“baz.txt”： 输出“4\n”
- 1.2若未能成功打开“baz.txt”： 输出“-1\n”

2.若未能成功打开“foo.txt”：

- 2.1若成功打开“baz.txt”： 输出“3\n”
- 2.2若未能成功打开“baz.txt”： 输出“-1\n”

## 文件描述符 → 文件名

- use `readlink` on `/proc/self/fd/NNN` where NNN is the file descriptor.
- This will give you the name of the file as it was when it was opened — however, if the file was moved or deleted since then, it may no longer be accurate

<https://stackoverflow.com/questions/1188757/retrieve-filename-from-file-descriptor-in-c>

## 来到stdio...

- UNIX I/O
  - open/close
  - read/write
  - lseek
  - dup/dup2
- stdio
  - fopen/fdopen/fclose
  - fscanf/fprintf, scanf/printf
    - fread/fwrite (较少用)
  - fseek
  - freopen
  - fflush

在stdio中——

- 将文件模型化为流 (stream)
- 缓冲区 (buffer)
  - 全缓冲：磁盘文件
  - 行缓冲：stdin, stdout
  - 无缓冲

## stdio缓冲区带来的问题

- 文件没有close：  
非正常退出时，在缓冲区中的数据不会被自动刷新

## What if... 忘了close?

- 使用stdio时：

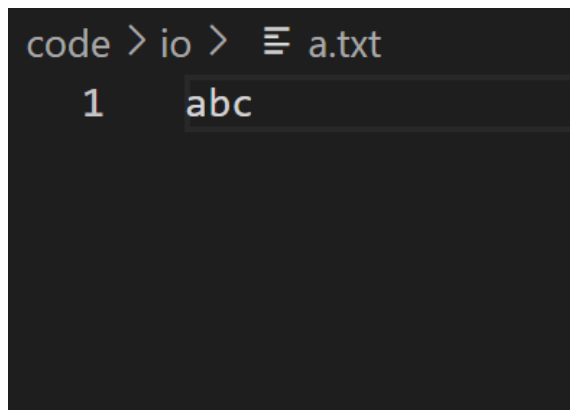
由于缓存区的存在，在程序结束生命被kernel回收前，并没有真正写入

Note：CS:APP page623，只要进程终止，内核都会关闭所有打开的文件，并回收资源

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

int main(){
    // int fp = open("./a.txt",O_WRONLY);
    FILE * fp = fopen("./a.txt","w");
    char ch[] = "abc";
    // write(fp, ch, 3);
    fprintf(fp,"%s",ch);

    // while(1);
    return 0 ;
}
```

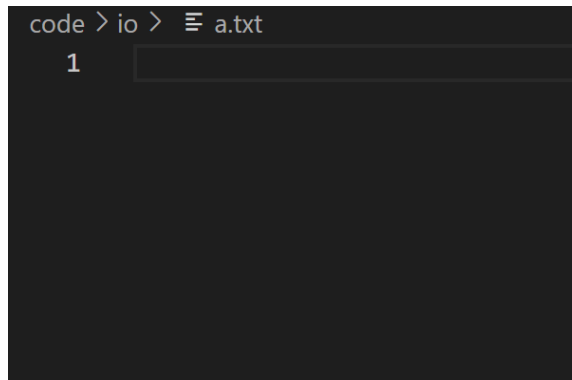


```
#include<stdio.h>
#include<stdlib.h>

int main(){
    FILE * fp = fopen("./a.txt","w");
    char ch[] = "abc";
    fprintf(fp,"%s",ch);

    while(1);
    // Ctrl + C
}
```

```
    return 0 ;  
}
```



## 忘了close还有可能...?

- 内存泄漏
  - 在Linux上，可以通过ulimit -n 来查看和更改当前session的限制数

```
$ ulimit -n  
7168  
$ ulimit -n 10000  
10000
```

- 信号安全：  
printf信号不安全，因为要获得缓冲区的锁
- 交替读写  
在读和写之间必须刷新缓冲区，因为读和写共享相同的缓冲区，并且读和写之间不会自动刷新缓冲区  
Note：这一点取决于实现，C标准不保证
- 使用建议：用一个文件不要用两个流去写，否则可能产生同步错误

## 算法题...为啥不能用cin&cout？

```
ios::sync_with_stdio(false);
```

---

cin、cout之所以效率低，是因为先把要输出的东西存入缓冲区，再输出，导致效率降低

真的吗？printf和scanf不也是要利用缓冲区？

默认情况下cin与stdin总是保持同步的，也就是说这两种方法可以混用，而不必担心文件指针混乱，同时cout和stdout也一样，两者混用不会输出顺序错乱。正因为这个兼容性的特性，导致cin有许多额外的开销

**下面看一下cin与scanf的读入速度比较（1e7与1e6的速度规模测试）**

1e7的数据规模

scanf读入需要时间：1.1s左右

cin读入需要时间：7.0s左右

std::ios::sync\_with\_stdio(false); 禁用同步：1.5s左右

1e6的数据规模

scanf读入需要时间：0.12s左右

cin读入需要时间：0.7s左右

std::ios::sync\_with\_stdio(false); 禁用同步：0.16s左右

总而言之，超时还是乖乖用scanf吧

- 
- <https://www.62042.com/cms/Content/detail/358449.html>

## 应用级缓存and...？

- 1) 它是所有进程共享的全局缓冲区吗？还是每个进程一个缓冲区？
- 2) 缓冲区在哪里？在堆栈、堆或静态区域？
- 3) 谁创造了它？

<https://stackoverflow.com/questions/14933629/where-is-the-stdout-buffer>

<https://www.anquanke.com/post/id/86945>

So, at least in this implementation, the default buffer probably lives in the FILE structure and is allocated on the heap.

- 输入输出缓冲区应该是调用stdio函数的时候应用层分配在堆上的，那么每个进程之间应该相互独立。

If you use low level routines like `write`, only the given bytes are written. Depending on the target, there will be buffering in the kernel. If the target is a tty, it might be written to the terminal directly.

但是也有人提到，write等更底层的函数会有由内核管理的缓存  
也许这就是教材里将这种缓冲区称为“应用级缓冲区”的原因？

- 带缓冲的输入函数。这些函数允许你高效地从文件中读取文本行和二进制数据，这些文件的内容缓存在应用级缓冲区内，类似于为 `printf` 这样的标准 I/O 函数提供的缓冲区。与[110]中讲述的带缓冲的 I/O 例程不同，带缓冲的 RIO 输入函数是线程安全的(12.7.1 节)，它在同一个描述符上可以被交错地调用。例如，你可以从一个描述符中读一些文本行，然后读取一些二进制数据，接着再多读取一些文本行。

## stderr

- 在stdin\stdout的抽象机制中加入stderr，防止混淆正常的输出和错误信息
- stdout是行缓冲的，stderr是无缓冲的

`printf(stdout, "xxxx")` 和 `printf(stdout, "xxxx\n")`，前者会憋住，直到遇到新行才会一起输出。

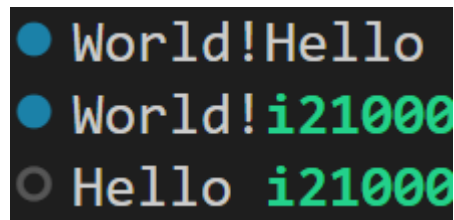
而`printf(stderr, "xxxxx")`，不管有没有\n，都输出。



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

int main(){
    fprintf(stdout,"Hello ");
    //fprintf(stderr,"Hello ");
    //while(1);
    fprintf(stderr,"World!");
    return 0;
}
```

```
$ ./a.out
$ ./a.out > .txt
$ ./a.out 2> .txt
```



A terminal window showing the output of a program. The first line is "World!Hello". The second line is "World!i21000". The third line is "Hello i21000". The "i21000" is in green, indicating it is the file descriptor for stderr. The "World!" is in white, indicating it is the file descriptor for stdout.

## Reference

- stderr [https://blog.csdn.net/origin\\_lee/article/details/41576975](https://blog.csdn.net/origin_lee/article/details/41576975)
- open & read <https://blog.csdn.net/hhhlizhao/article/details/71552588>
- stderr <https://zhuanlan.zhihu.com/p/29613516>
- <https://www.runoob.com/linux/linux-comm-stat.html>
- [https://blog.csdn.net/li\\_101357/article/details/78391589](https://blog.csdn.net/li_101357/article/details/78391589)