这些操作中的每一个都试图关闭同一个底层的套接字描述符，所以第二个 close 操作就会失败。对顺序的程序来说，这并不是问题，但是在一个线程化的程序中关闭一个已经关闭了的描述符是会导致灾难的(见 12.7.4 节)。

因此，我们建议你在网络套接字上不要使用标准 I/O 函数来进行输入和输出，而要使用健壮的 RIO 函数。如果你需要格式化的输出，使用 sprintf 函数在内存中格式化一个字符串，然后用 rio_writen 把它发送到套接口。如果你需要格式化输入，使用 rio_readlineb 来读一个完整的文本行，然后用 sscanf 从文本行提取不同的字段。

10.12　小结

Linux 提供了少量的基于 Unix I/O 模型的系统级函数，它们允许应用程序打开、关闭、读和写文件，提取文件的元数据，以及执行 I/O 重定向。Linux 的读和写操作会出现不足值，应用程序必须能正确地预计和处理这种情况。应用程序不应直接调用 Unix I/O 函数，而应该使用 RIO 包，RIO 包通过反复执行读写操作，直到传送完所有的请求数据，自动处理不足值。

Linux 内核使用三个相关的数据结构来表示打开的文件。描述符表中的表项指向打开文件表中的表项，而打开文件表中的表项又指向 v-node 表中的表项。每个进程都有它自己单独的描述符表，而所有的进程共享同一个打开文件表和 v-node 表。理解这些结构的一般组成就能使我们清楚地理解文件共享和 I/O重定向。

标准 I/O 库是基于 Unix I/O 实现的，并提供了一组强大的高级 I/O 例程。对于大多数应用程序而言，标准 I/O 更简单，是优于 Unix I/O 的选择。然而，因为对标准 I/O 和网络文件的一些相互不兼容的限制，Unix I/O 比之标准 I/O 更该适用于网络应用程序。

参考文献说明

Kerrisk 撰写了关于 Unix I/O 和 Linux 文件系统的综述 [62]。Stevens 编写了 Unix I/O 的标准参考文献[111]。Kernighan 和 Ritchie 对于标准 I/O 函数给出了清晰而完整的讨论[61]。

家庭作业

* 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  }
```

* 10.7　修改图 10-5 中所示的 cpfile 程序，使得它用 RIO 函数从标准输入复制到标准输出，一次 MAXBUF 个字节。

** 10.8　编写图 10-10 中的 statcheck 程序的一个版本，叫做 fstatcheck，它从命令行上取得一个描述符数字而不是文件名。

** 10.9　考虑下面对作业题 10.8 中的 fstatcheck 程序的调用：

```
linux> fstatcheck 3 < foo.txt
```

你可能会预想这个对 fstatcheck 的调用将提取和显示文件 foo.txt 的元数据。然而，当我们在