系统上运行它时，它将失败，返回“坏的文件描述符”。根据这种情况，填写出 shell 在 fork 和 execve 调用之间必须执行的伪代码：

```
if (Fork() == 0) { /* child */
    /* What code is the shell executing right here? */
    Execve("fstatcheck", argv, envp);
}
```

** 10.10　修改图 10-5 中的 cpfile 程序，使得它有一个可选的命令行参数 infile。如果给定了 infile，那么复制 infile 到标准输出，否则像以前那样复制标准输入到标准输出。一个要求是对于两种情况，你的解答都必须使用原来的复制循环(第 9～11 行)。只允许你插入代码，而不允许更改任何已经存在的代码。

## 练习题答案

10.1　Unix 进程生命周期开始时，打开的描述符赋给了 stdin(描述符 0)、stdout(描述符 1)和 stderr(描述符 2)。open 函数总是返回最低的未打开的描述符，所以第一次调用 open 会返回描述符 3。调用 close 函数会释放描述符 3。最后对 open 的调用会返回描述符 3，因此程序的输出是“fd2=3”。

10.2　描述符 fd1 和 fd2 都有各自的打开文件表表项，所以每个描述符对于 foobar.txt 都有它自己的文件位置。因此，从 fd2 的读操作会读取 foobar.txt 的第一个字节，并输出

```
c = f
```

而不是像你开始可能想的

```
c = o
```

10.3　回想一下，子进程会继承父进程的描述符表，以及所有进程共享的同一个打开文件表。因此，描述符 fd 在父子进程中都指向同一个打开文件表表项。当子进程读取文件的第一个字节时，文件位置加 1。因此，父进程会读取第二个字节，而输出就是

```
c = o
```

10.4　重定向标准输入(描述符 0)到描述符 5，我们将调用 dup2(5,0)或者等价的 dup2(5,STDIN_FILENO)。

10.5　第一眼你可能会想输出应该是

```
c = f
```

但是因为我们将 fd1 重定向到了 fd2，输出实际上是

```
c = o
```