系统级I/O

徐子扬

引入

- 在Linux系统中，所有语言所使用的较高级别的I/O函数，都是通过**内核**提供的系统级的Unix I/O实现的。
- 大多数时候，没有必要直接使用Unix I/O，但是在以下的情况，除了使用Unix I/O以外别无选择：
  - 读取文件的元数据
  - 进行网络编程
- 因此，我们需要了解如何使用Unix I/O

Unix I/O

- 所有I/O设备都被模型化为文件，所有对于这些设备的输入、输出则被当作对应文件的读、写。
  - **文件**：在Linux系统中，文件就是一个m字节的序列。
- 通过这种模型，Linux内核可以利用一个简单、低级的应用接口来实现所有的I/O设备的输入、输出。这个接口就是**Unix I/O**。
- 它们的实现方式包括：
  - **打开文件**
    - 打开的文件具体内容由内核记录，应用程序则用描述符来区别文件。
    - 开始一个进程时，会打开3个文件，分别是标准输入、标准输出、标准错误。
  - **改变当前的文件位置**
    - 文件位置：文件开头起始的字节偏移量
  - **读写文件**
    - EOF：当需要读写的位置超过了文件本身的大小时，会触发EOF，
  - **关闭文件**

文件

- Linux文件都有一个类型，包括以下几种：
  - **普通文件**：包含数据，又有文本文件与二进制文件之分，但对内核而言这两者没有区别。
  - **目录**：包含一组链接，其中每个链接将一个文件名映射到一个文件。
  - **套接字**（socket）：与另一个进程进行跨网络通信的文件。
  - 其他
- Linux中的所有文件组织成一个**目录层次结构**，其中的位置用**路径名**确定。每个进程都有一个当前工作目录，来确定这个进程的位置。“/”为**根目录**。
- Unix I/O提供了一系列关于文件的函数，它们的功能包括：
  - 打开或关闭文件
  - 读或写文件
  - 读取文件的元数据
  - 读取目录的内容
  - I/O重定向
  - 其他

打开和关闭文件

- 进程通过open函数打开文件
  - 用法：`int open(char *filename, int flags, mode_t mode);`
    - 读取filename的文件，返回描述符
    - flags指明如何访问这个文件
    - mode指定新文件的访问权限
- 通过调用close函数关闭已打开的文件
  - 用法：`int close(int fd);`
    - fd为文件的描述符
    - 成功则返回0，失败则返回-1
    - 若关闭一个已关闭的描述符，则会出错

- flags的具体取值包括：
  - O_RDONLY,O_WRONLY,O_RDWR,O_CREAT,O_TRUNC,O_APPEND等，以及它们的或。
  - 其含义分别是：只读，只写，可读可写，若文件不存在则创建空文件，若文件存在则截断（置空），每次写操作把文件位置设置到结尾。
  - 前三个是必须包含其一的

掩码	描述
S_IRUSR	使用者（拥有者）能够读这个文件
S_IWUSR	使用者（拥有者）能够写这个文件
S_IXUSR	使用者（拥有者）能够执行这个文件
S_IRGRP	拥有者所在组的成员能够读这个文件
S_IWGRP	拥有者所在组的成员能够写这个文件
S_IXGRP	拥有者所在组的成员能够执行这个文件
S_IROTH	其他人（任何人）能够读这个文件
S_IWOTH	其他人（任何人）能够写这个文件
S_IXOTH	其他人（任何人）能够执行这个文件

- mode的具体取值如图
  - 只有在flags中包含**O_CREAT**时才会使用这个参数
  - 会将权限设置为mode & ~umask
  - umask是每个进程的上下文的一部分，通过调用umask函数来设置

读和写文件

- 程序通过read函数和write函数执行输入输出。
  - `ssize_t read(int fd, void *buf, size_t n);`
  - `ssize_t write(int fd, const void *buf, size_t n);`
  - fd为描述符，buf为读写的内存位置，n为字节数
  - 若读写成功则返回成功的字节数，失败则返回-1
  - read返回0时说明遇到了EOF
- 在某些情况下，read和write传送的字节数少于要求的值，称为不足值，这些情况包括：
  - 读到EOF：刚读到EOF会返回实际读取的字节数，之后则会返回0。
  - 从终端读文本行：返回值为文本行的大小。
  - 读写socket：网络延迟会引起不足值。
- 读写磁盘时，只有读到EOF才会遇到不足值，其他情况均不会。

RIO

- 我们可以使用**RIO包**来处理可能遇到的不足值。RIO包的核心思想在于，当读/写时实际传输的字节数小于理论上应该传输的字节数时，持续重复进行读/写，直到传输完所有应该传输的字节。
- RIO提供两类不同的函数：
  - 无缓冲的输入输出函数
  - 带缓冲的输入函数
    - 使用缓冲区的好处在于，可以减少系统调用的次数，从而减少开销。

读取文件的元数据

- **元数据**指的是关于文件的信息，包括文件的字节数、访问许可位、文件类型等。
  - Linux中定义了宏谓词来确定给定st_mode的文件类型
  - S_ISREG(m)—是否为普通文件；S_ISDIR(m)—是否为目录；S_ISSOCK—是否为socket。
- 程序可以通过**stat函数**和**fstat函数**获取元数据
  - `int stat(const char *filename, struct stat *buf);`
  - `int fstat(int fd, struct stat *buf);`
  - 分别是通过文件名和描述符获取元数据，并存入buf中。成功则返回0，出错则返回-1。

读取目录内容

- 程序可以通过readdir等一系列函数读取目录的内容。
  - `DIR *opendir(const char *name);`
    - 以**路径名**为参数，返回指向**目录流**的指针（失败则为NULL）。
  - `struct dirent *readdir(DIR *dirp);`
    - 参数为目录流，返回值为下一个目录项的指针，若没有下一项或失败则返回NULL。
    - 每个目录项至少包含两个成员：
      - d_ino：文件位置
      - d_name：文件名称
    - 区分失败和流结束的方法只有检查errno是否修改过。
  - `int closedir(DIR *dirp);`
    - 关闭目录流并释放资源

共享文件

- 内核用三个数据结构来表示打开的文件
  - **描述符表**：每个进程都有独立的描述符表，其中每个描述符指向文件表中的表项。
  - **文件表**：所有的进程共有同一张文件表，用来表示**打开文件的集合**。其中的表项有当前文件位置、引用计数、指向v-node表的指针。
    - 引用计数是指向该表项的描述符个数。
  - **v-node表**：也是所有的进程所共有的，每个表项包含stat的大多数信息。
- 多个描述符可以通过不同的文件表项来引用同一个文件，这是共享文件的一种情况。对同一个filename调用两次open就会发生。
- 父进程打开文件后调用fork，子进程会获得父进程描述符表的副本，父子进程就会共享这个文件。要关闭这个文件，必须父子进程都关闭。

假设磁盘文件foobar.txt由6个ASCII字符“foobar”组成，那么下面的程序分别输出什么？

```
int main()
{
    int fd1, fd2;
    char c;

    fd1 = Open("foobar.txt", O_RDONLY, 0);
    fd2 = Open("foobar.txt", O_RDONLY, 0);
    Read(fd1, &c, 1);
    Read(fd2, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

答案：f

指向同一个文件的不同描述符，它们的文件位置可能不同，因为它们在文件表中的项不同。

```
int main()
{
    int fd;
    char c;

    fd = Open("foobar.txt", O_RDONLY, 0);
    if (Fork() == 0) {
        Read(fd, &c, 1);
        exit(0);
    }
    Wait(NULL);
    Read(fd, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

答案：o

父子进程中对应的描述符，在文件表中指的是同一个项，共享一个文件位置。

2. 考虑以下代码，假设ICS.txt中的初始内容为"ICS!!!ics!!!"：

```
int fd = open("ICS.txt", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
for (int i = 0; i < 2; ++i){
    int fd1 = open("ICS.txt", O_RDWR | O_APPEND);
    int fd2 = open("ICS.txt", O_RDWR);
    write(fd2, "!!!!!!", 6);
    write(fd1, "ICS", 3);
    write(fd, "ics", 3);
}
```

假设所有系统调用均成功，则这段代码执行结束后，ICS.txt的内容为（）：

A．ICSics

B．!!!icsICS

C．!!!icsics!!!ICSICS

D．!!!icsICSICS

答案：D

1、由于fd打开时flags带有O_TRUNC，所以初始内容全部清空。

2、fd1的带有O_APPEND，所有每次write均在末尾添加。

3、fd在循环体外，会保持之前的文件位置。

I/O重定向

- Linux shell 提供了重定向操作符，将磁盘文件和标准I/O联系起来。
- 它的一种工作方式是dup2函数
  - `int dup2(int oldfd, int newfd);`
    - 它将oldfd的描述符表项复制到newfd并覆盖，若newfd已打开则先将它关闭。
    - 成功时返回newfd，失败时返回-1。
  - 在调用成功dup2函数之后，newfd和oldfd所指的文件表项都是之前oldfd所指的项。
  - 根据这一点，我们可以通过dup2(oldfd,STDOUT_FILENO)来将标准输出重定向至oldfd所指的文件。

标准I/O

- C语言定义了一组高级的输入输出函数，称为**标准I/O库**，它是Unix I/O的较高级别替代。
- 与Unix I/O相比，标准I/O使用一个FILE指针来指一个文件的流，而非描述符。初始的三个文件分别为stdin,stdout,stderr。
- 除此之外，FILE的流还具有缓冲区的功能，也可以使系统调用的次数尽可能减少，以此减少开销。

使用I/O的原则

- 1、尽可能使用标准I/O，除非迫不得已（例如需要stat函数）。
- 2、不使用scanf等专用于文本文件的函数来读取二进制文件。
- 3、对网络套接字的I/O，使用RIO。因为标准I/O在网络输入输出时可能会出现一些问题。