文件I/O

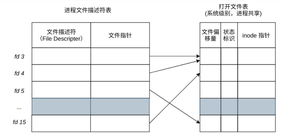
文件描述符

在UNIX文件I/O中，获取一个文件描述符的方法之一就是使用open系统调用 ⊙ *int open(const char *pathname, int flags, ...);*

- 成功时返回int类型的文件描述符
- 失败时(文件不存在、未有权限等原因)则返回-1

从系统调用返回类型来看，文件描述符就是一个int，比如25，35。其中，0，1，2这三个描述符分别用于指代标准输入、标准输出以及标准错误，由UNIX事先分配好 ⊙ 也就是说，当一个进程没有禁用这三个描述符时，第一个打开的文件的描述符大小通常为3

文件描述符并不等同于文件指针，在I/O操作以外，该整型的文件描述符几乎不能做任何事情 ⊙ 实际上，文件描述符和文件指针共同组成了进程中文件描述符表



⊙ 当在一个进程中对同一个文件执行多次open时，会返回不同的文件描述符，尽管它们的文件指针指向相同的结构

通过文件描述符来操作文件无疑要更加方便一些，不用拿着文件指针到处跑，就好比我们一般拿着锤子的柄，而不是直接拿着锤头。 ⊙ windows中称为文件句柄，还挺贴切

创建一个文件描述符的代价极低，因为打开一个文件仅仅只是获取文件的元信息，例如当前文件写入的偏移量，文件状态等等 ⊙ 但是进程会有最大打开文件数量的限制，通常为1024或者是8192

打开一个文件: open

open系统调用既能够打开一个已经存在的文件，也可以创建并打开一个新文件，调用返回文件描述符，发生错误时则返回-1，并将全局的errno置为相应的错误编码

调用参数

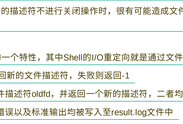
- char *pathname ⊙ 文件所在地址的字符串形式
- int flags
  - open调用中最为重要、并且也是最让人感到迷惑的参数，用于指定文件的访问模式，通常称为位掩码 ⊙ 在系统调用中，只要是"位掩码"类，那么在实际传参时，多半都会使用逻辑或(|)操作进行组合
  - O_RDONLY ⊙ 以只读方式打开
  - O_WRONLY ⊙ 以只写方式打开
  - O_RDWR ⊙ 以读写方式打开

  三者选其一
  - O_APPEND ⊙ **总是在文件尾部追加数据，内核的原子操作实现(文件偏移量的移动与数据写入纳为一原子操作)** ⊙ 同一个文件可能会在多个进程或者多个线程中写入时，务必添加该位掩码
  - O_ASYNC ⊙ **是否使用信号驱动I/O，该位掩码在open调用中无效，若想要使用信号驱动I/O，则需要调用fcntl()的F_SETFL操作来设置O_ASYNC标识**
  - O_CLOEXEC ⊙ **即close-on-exec，当进程或线程调用exec使用新的程序映像执行时，关闭原来已经打开的文件描述符**
  - O_CREAT ⊙ 当想要打开的文件不存在时，则创建一个新的文件，此时需要提供mode参数，即相关的文件权限
  - O_EXCL ⊙ 此标志通常和O_CREAT一起使用，表示如果当前文件已经存在，则不会打开该文件，且open调用返回失败

  两者组合符为一原子操作
  - O_DIRECT ⊙ **I/O操作绕过内核缓冲区，直接写入设备缓冲区(如果有的话)。该标识并不建议设置，应用代码与底层硬件直接交互并不是个好的选择，并且效率上不见得会比内核缓冲区要高**
  - O_NONBLOCK ⊙ 以非阻塞的方式打开文件，当使用此标识时，read以及write系统调用也将是非阻塞的 ⊙ 对于文件的I/O操作而言，由于存在内核缓冲区，所以不会陷入阻塞，因而文件I/O通常会忽略该标识。但是socket以及FIFO可能会使用到该标识位
  - O_SYNC ⊙ **以同步I/O的方式打开文件**
  - O_TRUNC ⊙ 截断已有文件，使其长度为0，相当于初始化空文件
  - O_DSYNC ⊙ 提供同步的I/O数据完整性(Linux内核版本需大于等于2.6.33)
- mode_t mode ⊙ 指定文件的访问权限

位掩码按位或(|)操作来确定多个选项的组合，是一个很有意思的话题

C语言自身支持的数据结构相当的单一，除了基本数据类型以外，没有散列表的支持，所以，什么样的数据结构能够支持诸如排列组合，并且存在分类情况的入参呢？ ⊙ bitmap是一种选择，但是一个bit只能有两个选项 ⊙ 那么8进制，甚至16进制呢？ ⊙ 诸如O_WRONLY就是8进制数，以01表示



⊙ 挑选奶茶的配料还挺合适使用该结构的，比如口味、冰量、要不要奶盖、糖度等等，分类很多，每个分类下的选项也很多，使用其它数据结构就很占空间

文件I/O

读取文件内容: read

比起open调用，read调用就显得"索然无味"许多

原型: ssize_t read(int fd, void *buffer, size_t count); ⊙

- fd即为open调用返回的文件描述符
- buffer为用户提供的缓冲区。系统调用并不会为用户提供缓冲区，库函数倒是有可能。
- count为读多少个字节至buffer中，所以sizeof(buffer) >= count

调用返回实际读取的字节数，可能小于count。返回0时表示已到达文件末尾，-1则表示错误，并将errno设置为对应的数值

内核缓冲区

磁盘会有缓冲区，用于缓存常用的数据。内核I/O同样存在缓冲区，用于保存用户常用数据以及提供更方便的读写功能

当用户在读取一个文件内容时，内核会默认读取更多的数据进入内核缓冲区中: 因为内核认为这部分数据可能很快就会被用到。事实也同样如此，当一块数据被读取时，其相邻的下一块数据很大可能也会被读取

⊙

- 内核缓冲区的作用不仅仅是缓存数据这么简单，还会帮助用户屏蔽硬件细节
- 例如，如果使用O_DIRECT标识位绕过内核缓冲区，那么用户在进行I/O操作时数据传输的开始点必须是块大小的整数倍

系统调用的函数风格

- C不支持异常(Exception)，所以只能使用错误返回的方式来表达调用出错
- C不支持多结果返回，一个函数调用仅能返回一个结果

所以，函数的返回值通常会表达两层含义: 大于0表示调用正常，返回-1则表示错误。结果将会由内核写入至用户提供的指针所指向的区域中

数据写入文件: write

原型: ssize_t write(int fd, void *buffer, size_t count)

- write系统调用将向内核缓冲区写入buffer内容中count字节的数据，并返回实际写入的数量
- 对于文件的写入而言，write一定会在有限的时间内返回，不管是正确还是错误，socket则不一定

当不使用O_DIRECT标识位时，数据写入至内核缓冲区后立即返回。再由内核寻找合适的时机将内核缓冲区中的数据写入至硬盘，从而减少实际落盘的次数，提高I/O效率 ⊙ 当然，也可能仅仅是写入了硬盘的缓冲区

如果进程将数据写入内核缓冲区后，在内核未将数据落盘之前，系统崩溃，或是电源断开，则该部分数据会丢失，且进程认为自己已经将数据写入磁盘。所以，主动刷新缓冲区数据至硬盘的操作很有必要 ⊙ 如MySQL的redo log，必须保证数据成功写入日志文件后，才可继续进行后续的事务操作

数据完整性与文件完整性

- 同步I/O数据完整性(synchronized I/O data integrity completion) ⊙ **确保文件的数据传递至磁盘** ⊙ **int fdatasync(int fd)，返回0表示成功，-1表示错误**
- 同步I/O文件完整性(synchronized I/O file integrity completion) ⊙ **确保文件的数据与元数据(如文件大小、更新时间戳等)传递至磁盘** ⊙ **int fsync(int fd)，返回0表示成功，-1表示错误** ⊙ 可以认为fsync()调用是fdatasync()调用的超集 ⊙ **void sync(void);** ⊙ 将内核缓冲区中所有数据与文件元数据同步至硬盘，是fsync()调用的超集

使所有写入同步: O_SYNC

- 当在open系统调用的flags中指定O_SYNC标识位时，后续全部的write调用都将自动地将文件数据与元数据同步至磁盘
- 即使文件处于synchronized I/O file integrity completion状态，或者说，相当于每次写入时均进行fsync()调用

该标识位会严重影响性能，慎用

使所有数据写入同步: O_DSYNC ⊙ 始于2.6.33内核版本，用于将文件状态转变为synchronized I/O data integrity completion，是一种更加精细化的处理

修改文件偏移量: lseek

对一个刚刚创建的文件来说，文件偏移量指向文件开始，随着read以及write调用的进行，将不断调整文件偏移量至正确的位置

原型: off_t lseek(int fd, off_t offset, in whence); ⊙

- offset指定了一个以字节为单位的数值
- whence则是一个锚定点，表明参照哪个基点来解释offset参数
  - SEEK_SET ⊙ 将文件偏移量设置为从文件头部起始点开始的offset个字节
  - SEEK_CUR ⊙ 相对于当前文件偏移量，将偏移量调整offset个字节
  - SEEK_END ⊙ 将偏移量设置为起始于文件尾部的offset个字节

关闭文件: close

close系统调用关闭一个以打开的文件描述符，并将其释放放回进程。当进程结束时，将会自动关闭已打开的所有文件描述符

进程对打开的文件数量是有限制的，当对不再使用的描述符不进行关闭操作时，很有可能造成文件描述符资源耗尽，造成进程无法再打开新的文件，此时会给出 Too many open files 的错误

文件描述符的复制

文件描述符的复制是个很有趣并且使用非常广泛的一个特性，其中Shell的I/O重定向就是通过文件描述符的复制实现的

dup

- *int dup(int oldfd);* ⊙ 调用成功将返回新的文件描述符，失败则返回-1
- dup()系统调用将复制一个已经打开的文件描述符oldfd，并返回一个新的描述符，二者均指向同一个文件句柄
- 例如 ./myscript > result.log 2>&1，标准错误以及标准输出均被写入至result.log文件中

dup2

- *int dup2(int oldfd, int newfd);* ⊙ 调用成功将返回新的文件描述符，失败则返回-1
- dup2()系统调用会为oldfd描述符创建副本，并且指定副本的文件描述符号，编号由newfd参数决定 ⊙ 如 dup2(1, 15)，复制标准输入文件描述符，并指定复制的描述符编号为15
- dup2()可用于I/O重定向，如./myscript > result.log 2>&1，标准错误以及标准输出均被写入至result.log文件中，其原理在于将文件描述符1复制到文件描述符2 ⊙ dup2(STDOUT_FILENO, STDERR_FILENO);
- **假设newfd所指向的文件描述符之前已经打开，那么在调用dup2时，则会将其关闭，期间并不会检查错误**

dup3

- *int dup3(int oldfd, int newfd, int flags);* ⊙ 调用成功将返回新的文件描述符，失败则返回-1
- 其作用与dup2()完全相同，只不过多了flags参数用于控制新的文件描述符的行为
- 目前仅支持O_CLOEXEC标识，当进程调用exec()使用新的进程影像执行时，关闭原有进程打开的文件描述符