Technology Tarena 达内科技

Unix系统高级编程

File 2

DAY05

内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	文件的打开与关闭
	10:30 ~ 11:20	文件的内核结构与描述符
	11:30 ~ 12:20	
下午	14:00 ~ 14:50	文件的读写与随机访问
	15:00 ~ 15:50	
	16:00 ~ 16:50	系统与标准I/O
	17:00 ~ 17:30	总结和答疑

文件的打开与关闭

Technology Tarena 达内科技

打开/创建文件

打开/创建文件

- 打开已有的文件或创建新文件

```
#include <fcntl.h>

int open (const char* pathname, int flags,
    mode_t mode);
```

成功返回文件描述符，失败返回-1

  - *pathname*：文件路径
  - *flags*：状态标志，可取以下值

    O_RDONLY - 只读
    O_WRONLY - 只写
    O_RDWR - 读写
    （只选一个）
    O_APPEND - 追加

知识讲解

打开/创建文件（续1）

- 打开已有的文件或创建新文件
  - ***flags***：状态标志，可取以下值

    O_CREAT - 创建，不存在即创建，已存在即打开，除非与以下两个标志之一合用，有此标志***mode***参数才有效

    O_EXCL - 排斥，已存在即失败

    O_TRUNC - 清空，已存在即清空，同时需要有**O_WRONLY**或**O_RDWR**标志

    （最多选一个，配合**O_CREAT**使用）

    O_SYNC - 同步，在数据被写到磁盘之前写操作不会完成，读操作本来就是同步的，此标志对读操作没有意义

    O_ASYNC - 异步，在文件可读写时产生一个SIGIO信号，只能用于终端设备或套接字，不能用于普通文件

    O_NONBLOCK - 非阻塞，任何后续操作都不会使调用进程在I/O中阻塞，只能用于有名管道

知识讲解

打开/创建文件（续2）

- 打开已有的文件或创建新文件
  - ***mode***：权限模式，仅在创建新文件时有效，可用形如0XXX的三位八进制数表示，由高位到低位依次表示拥有者用户、同组用户和其它用户的读、写和执行权限，读权限用4表示，写权限用2表示，执行权限用1表示，最后通过加法将不同的权限组合在一起
- 例如
  - int fd = open ("open.txt", O_RDWR | O_CREAT | O_TRUNC, 0666);
    if (fd == -1) {
    perror ("open");
    exit (EXIT_FAILURE);
    }

知识讲解

打开/创建文件（续3）

- 调用进程的权限掩码会屏蔽掉创建文件时指定的权限位
  - 如创建文件时指定权限0666，进程权限掩码0022，所创建文件的实际权限为：0666&~0022=0644 (rw-r--r--)
- 所返回的一定是当前未被使用的，最小文件描述符
- 一个进程最多能持有多少个处于打开状态的文件描述符，由limits.h头文件中的OPEN_MAX宏决定。合法文件描述符的值应为0到OPEN_MAX-1。POSIX要求该宏的值不低于16，传统Unix是64，现代Linux使用更大的值
  - #include <unistd.h>
    long openmax = sysconf (_SC_OPEN_MAX); // 1024

知识讲解

打开/创建文件（续4）

- 如果只想单纯地创建新文件，可以使用更简单的版本

```
#include <fcntl.h>

int creat (const char* pathname, mode_t mode);
```

成功返回文件描述符，失败返回-1

  - 该函数的实现类似下面这样

```
int creat (const char* pathname, mode_t mode) {
    return open (pathname, O_WRONLY | O_CREAT |
        O_TRUNC, mode);
}
```

知识讲解

打开/创建文件（续5）

- 如果只想打开已有的文件，可以使用更简单的版本

```
#include <fcntl.h>

int open (const char* pathname, int flags);
```

成功返回文件描述符，失败返回-1

  - open函数之所以存在多种参数表的形式，并非源自类似C++语言的函数重载机制，而是使用了标准C语言提供的可变长参数列表，手册中的写法只是为了便于阅读和使用

知识讲解

Technology Tarena 达内科技

关闭文件

关闭文件

- 关闭处于打开状态的文件描述符

```
#include <unistd.h>

int close (int fd);
```

成功返回0，失败返回-1

  - *fd*：处于打开状态的文件描述符
- 例如
  -

```
if (close (fd) == -1) {
    perror ("close");
    exit (EXIT_FAILURE);
}
```

知识讲解

文件的打开与关闭

【参见：open.c】

课堂练习

- 文件的打开与关闭

文件的内核结构与描述符

Technology Tarena 达内科技

文件的内核结构

文件的内核结构

- 一个处于打开状态的文件，系统会为其在内核中维护一套专门的数据结构，保存该文件的信息，直到它被关闭
  - v节点与v节点表
    - 文件的元数据和在磁盘上的存储位置都保存在其i节点中，而i节点保存在分区柱面组的i节点表中，在打开文件时将其i节点信息读入内存，并辅以其它的必要信息形成一个专门的数据结构，势必会提高对该文件的访问效率，这个存在于进程的内核空间，包含文件i节点信息的数据结构被称为v节点。多个v节点结构以链表的形式构成v节点表
  - 文件表项与文件表
    - 由文件状态标志(来自open函数的***flags***参数)、文件读写位置(最后一次读写的最后一个字节的下一个位置)和v节点指针等信息组成的内核数据结构被称为文件表项。通过文件表项一方面可以实时记录每次读写操作的准确位置，另一方面可以通过v节点指针访问包括该文件各种元数据和磁盘位置在内的i节点信息。多个文件表项以链表的形式构成文件表

知识讲解

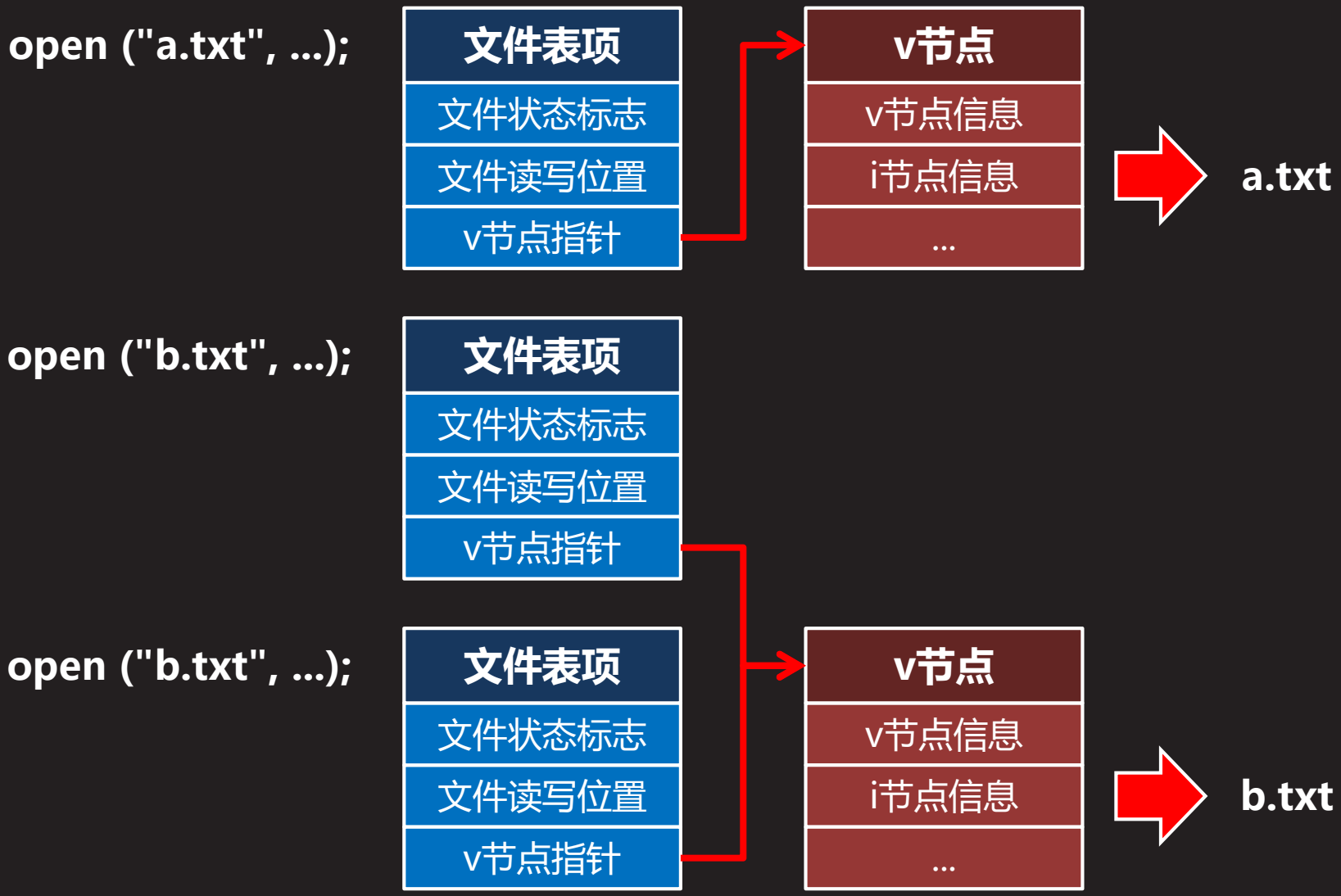
文件的内核结构（续1）

- 多次打开同一个文件，无论是在同一个进程中还是在不同的进程中，都只会在系统内核中产生一个v节点
- 每次打开文件都会产生一个新的文件表项，各自维护各自的文件状态标志和当前文件偏移，却可能因为打开的是同一个文件而共享同一个v节点
- 打开一个文件意味着内存资源(v节点、文件表项等)的分配，而关闭一个文件其实就是为了释放这些资源，但如果所关闭的文件在其它进程中正处于打开状态，那么其v节点并不会被释放，直到系统中所有曾打开过该文件的进程都显式或隐式地将其关闭，其v节点才会真正被释放
- 一个处于打开状态的文件也可以被删除，但它所占用的磁盘空间直到它的v节点彻底消失以后才会被标记为自由

知识讲解

文件的内核结构（续2）

知识讲解



Technology Tarena 达内科技

文件描述符

文件描述符

- 由文件的内核结构可知，一个被打开的文件在系统内核中通过文件表项和v节点加以标识。有关该文件的所有后续操作，如读取、写入、随机访问，乃至关闭等，都无一例外地要依赖于文件表项和v节点。因此有必要将文件表项和v节点体现在完成这些后续操作的函数的参数中。但这又势必会将位于内核空间中的内存地址暴露给运行于用户空间中的程序代码。一旦某个用户进程出现操作失误，极有可能造成系统内核失稳，进而影响其它正常运行的用户进程。这将对操作系统的安全运行造成极大的威胁

知识讲解

文件描述符（续1）

- 为了解决内核对象在可访问性与安全性之间的矛盾，Unix系统通过所谓的文件描述符，将位于内核空间中的文件表项间接地提供给运行于用户空间中的程序代码
- 为了便于管理在系统中运行的各个进程，内核会维护一张存有各进程信息的列表，谓之进程表。系统中的每个进程在进程表中都占有一个表项。每个进程表项都包含了针对特定进程的描述信息，如进程ID、用户ID、组ID等，其中也包含了一个被称为文件描述符表的数据结构
- 文件描述符表的每个表项都至少包含两个数据项——文件描述符标志和文件表项指针，而所谓文件描述符，其实就是文件描述符表项在文件描述符表中从0开始的下标

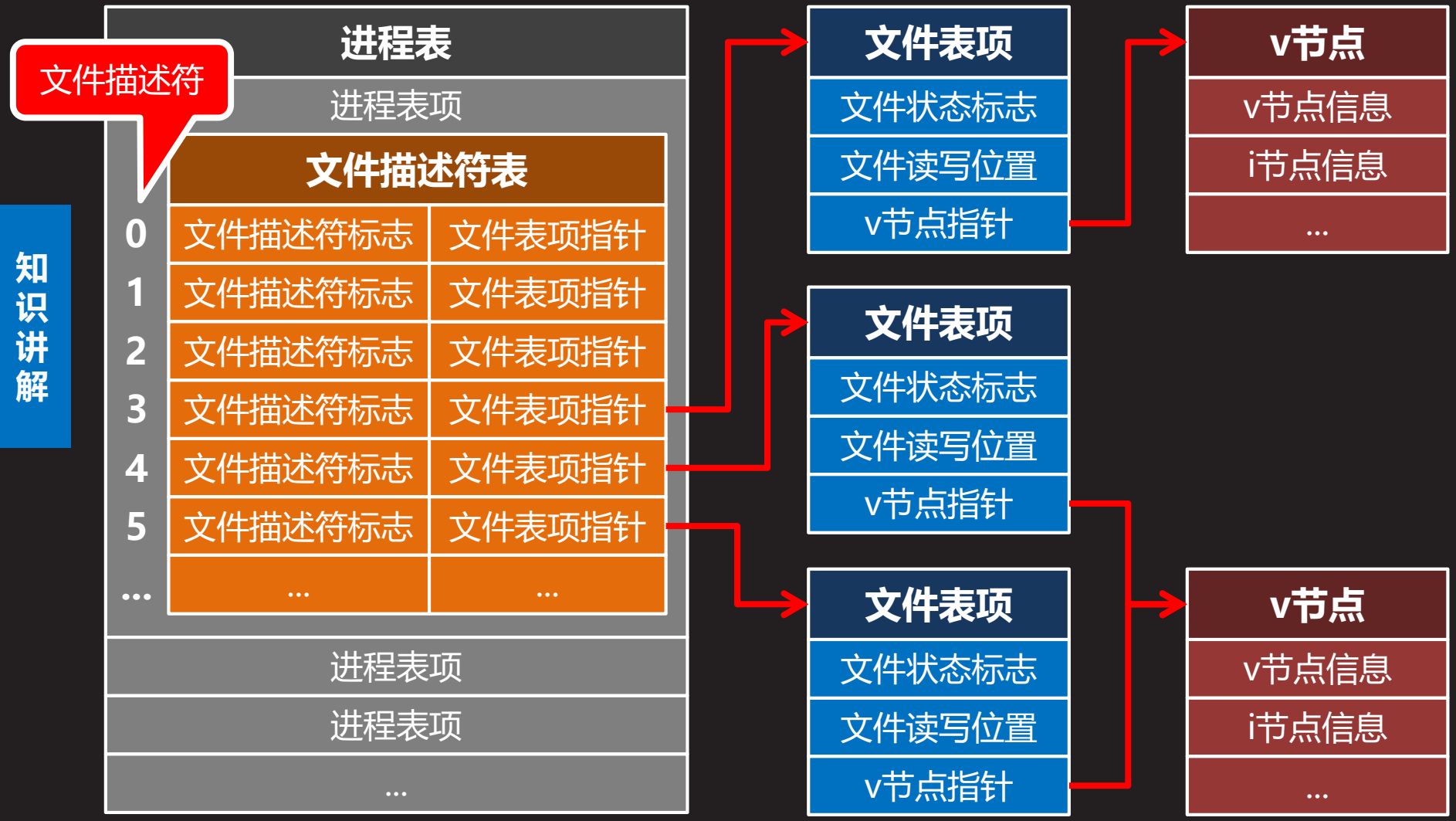
文件描述符（续2）

- 通过系统调用向用户代码返回文件描述符是有意义的。一方面，用户代码在随后的文件访问中，可以文件描述符为参数，通过后续函数调用传回系统内核，内核通过查询文件描述符表，找到与该下标对应的文件描述符表项，并从中得到文件表项指针，即可借由文件表项和v节点访问文件的内容或对文件施加必要的控制。另一方面，因为用户代码所持有的仅仅是文件描述符表项在文件描述符表中的下标，而永远不可能获得文件描述符表的起始地址，因此用户代码也就无法直接访问文件表项和v节点，位于内核空间中的代码和数据得到了有效的保护

知识讲解

文件描述符（续3）

知识讲解



文件描述符（续4）

- 作为文件描述符表项在文件描述符表中的下标，合法的文件描述符一定是个大于等于0的整数
- 每次产生新的文件描述符表项，系统总是从下标0开始在文件描述符表中寻找最小的未使用项
- 每关闭一个文件描述符，无论被其索引的文件表项和v节点是否被删除，与之对应的文件描述符表项一定会被标记为未使用，并在后续操作中为新的文件描述符所占用
- 系统内核缺省为每个进程打开三个文件描述符，它们在unistd.h头文件中被定义为三个宏
  - `#define STDIN_FILENO  0 // 标准输入`
    `#define STDOUT_FILENO 1 // 标准输出`
    `#define STDERR_FILENO 2 // 标准错误`

知识讲解

文件描述符（续5）

- 在C语言标准库中，一般用FILE*类型的I/O流指针标识一个打开的文件。该指针或由fopen函数返回，或由标准库以全局变量stdin、stdout和stderr的形式预定义并初始化。事实上FILE结构体中已经包含了文件描述符成员
  - /usr/include/stdio.h
    typedef struct _IO_FILE FILE;
  - /usr/include/libio.h
    struct _IO_FILE {
    ...
    int _fileno; // 文件描述符
    ...
    };

知识讲解

文件描述符（续6）

Technology Tarena 达内科技

- Shell对这样的命令行执行如下操作

```
$ a.out 0<i.txt 1>o.txt 2>e.txt
```

  - 关闭文件描述符0，打开i.txt，该文件即获得文件描述符0
  - 关闭文件描述符1，打开o.txt，该文件即获得文件描述符1
  - 关闭文件描述符2，打开e.txt，该文件即获得文件描述符2
  - 创建子进程，执行a.out
  - 子进程继承父进程的文件描述符表，因此a.out中所有从标准输入的读取和向标准输出及标准错误的写入，都被分别定向到i.txt、o.txt和e.txt文件中，这就是I/O重定向的原理

知识讲解

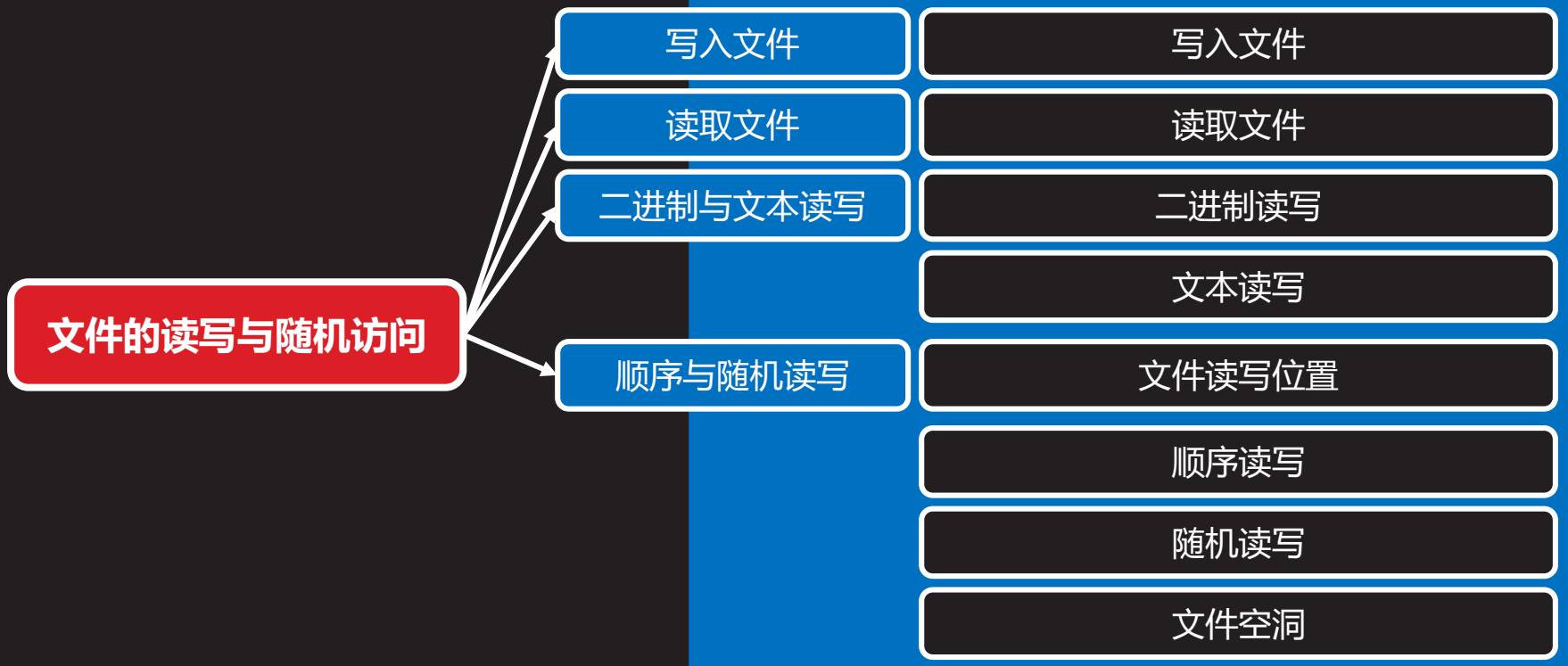
I/O重定向

【参见：redir.c】

课堂练习

- I/O重定向

文件的读写与随机访问


文件的读写与随机访问
写入文件
读取文件
二进制与文本读写
顺序与随机读写
写入文件
读取文件
二进制读写
文本读写
文件读写位置
顺序读写
随机读写
文件空洞

Technology Tarena 达内科技

写入文件

写入文件

- 向指定文件写入字节流

```
#include <unistd.h>

ssize_t write (int fd, const void* buf, size_t count);

        成功返回实际写入的字节数(0表示未写入)，失败返回-1
```

  - *fd*：文件描述符
  - *buf*：内存缓冲区
  - *count*：期望写入的字节数
- 例如
  - ssize_t written = write (fd, text, towrite);
    if (written == -1) { perror ("write"); exit (EXIT_FAILURE); }

知识讲解

写入文件

【参见：write.c】

课堂练习

- 写入文件

Technology Tarena 达内科技

读取文件

读取文件

- 从指定文件读取字节流

```
#include <unistd.h>

ssize_t read (int fd, void* buf, size_t count);

    成功返回实际读取的字节数(0表示读到文件尾)，失败返回-1
```

  - *fd*：文件描述符
  - *buf*：内存缓冲区
  - *count*：期望读取的字节数
- 例如
  - ssize_t readed = read (fd, text, toread);
    if (readed == -1) { perror ("read"); exit (EXIT_FAILURE); }

知识讲解

读取文件

【参见：read.c】

课堂练习

- 读取文件

Technology Tarena 达内科技

二进制与文本读写

二进制读写

- 基于系统调用的文件读写本来就是面向二进制字节流的，因此对二进制读写而言，无需做任何额外的工作
  - typedef struct Employee {
    char name[256];
    int age;
    float salary;
    } EMP;
  - EMP emp = {"赵云", 25, 8000};
    if (write (fd, &emp, sizeof (emp)) == -1) {
    perror ("write"); exit (EXIT_FAILURE); }
  - EMP emp = {}
    if (read (fd, &emp, sizeof (emp)) == -1) {
    perror ("read"); exit (EXIT_FAILURE); }

知识讲解

二进制读写

【参见：binary.c】

课堂练习

- 二进制读写

文本读写

- 因为基于系统调用的文件读写是面向二进制字节流的，因此对写入内容的格式化和对读取内容的解格式化，都必须通过自己编写的代码处理
  - EMP emp = {"赵云", 25, 8000};
    sprintf (buf, "%s %d %.2f", emp.name, emp.age,
    emp.salary);
    if (write (fd, buf, strlen (buf) * sizeof (buf[0])) == -1) {
    perror ("write"); exit (EXIT_FAILURE); }
  - if (read (fd, buf, sizeof (buf)) == -1) {
    perror ("read"); exit (EXIT_FAILURE); }
    EMP emp = {};
    sscanf (buf, "%s%d%f", emp.name, &emp.age,
    &emp.salary);

知识讲解

文本读写

【参见：text.c】

课堂练习

- 文本读写

Technology Tarena 达内科技

顺序与随机读写

文件读写位置

- 每个打开的文件都有一个与其相关的文件读写位置保存在文件表项中，用以记录从文件头开始计算的字节偏移
- 文件读写位置通常是一个非负的整数，用off_t类型表示，在32位系统上被定义为long int，而在64位系统上则被定义为long long int
- 打开一个文件时，除非指定了O_APPEND标志，否则文件读写位置一律被设为0，即文件首字节的位置

知识讲解

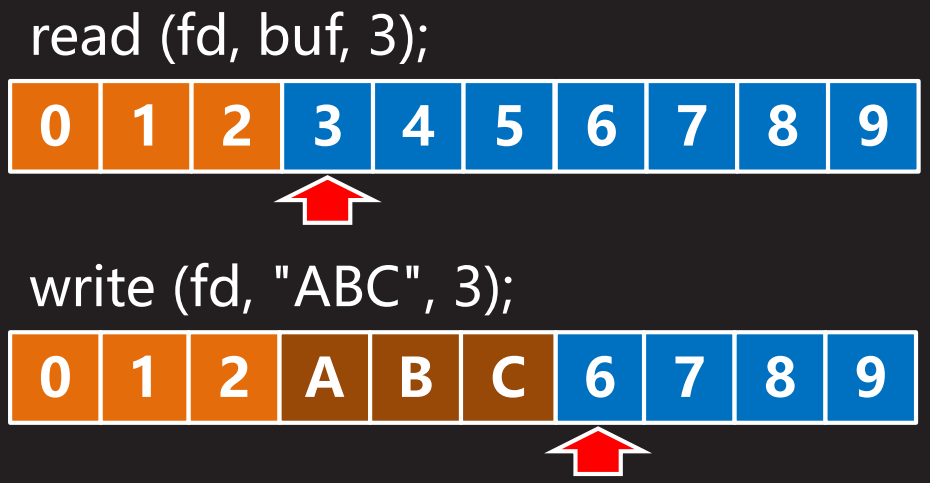
creat ("write.txt", 0644);

write (fd, "0123456789", 10);

顺序读写

- 每一次读写操作都从当前的文件读写位置开始，并根据所读写的字节数，同步增加文件读写位置，为下一次读写做好准备
- 因为文件读写位置是保存在文件表项而不是v节点中的，因此通过多次打开同一个文件得到多个文件描述符，各自拥有各自的文件读写位置

知识讲解



随机读写

- 人为调整文件读写位置

```
#include <unistd.h>

off_t lseek (int fd, off_t offset, int whence);

                    成功返回调整后的文件读写位置，失败返回-1
```

  - *fd*：文件描述符
  - *offset*：文件读写位置相对于*whence*参数的偏移量

知识讲解

随机读写（续1）

- 人为调整文件读写位置
  - ***whence***：根据***offset***参数计算文件读写位置的起点，可取以下值

    SEEK_SET - 从文件头(文件的第一个字节)开始

    SEEK_CUR - 从当前位置(上次读写的最后一个字节的下一个位置)开始

    SEEK_END - 从文件尾(文件最后一个字节的下一个位置)开始
- lseek函数的功能仅仅是修改保存在文件表项中的文件读写位置，并不实际引发任何I/O动作

知识讲解

随机读写（续2）

- 例如
  - lseek (fd, -7, SEEK_CUR); // 从当前位置向文件头偏移7字节
  - lseek (fd, 0, SEEK_CUR); // 返回当前文件读写位置
  - lseek (fd, 0, SEEK_END); // 返回文件总字节数

知识讲解

read (fd, buf, 3);

lseek (fd, 3, SEEK_CUR);

文件空洞

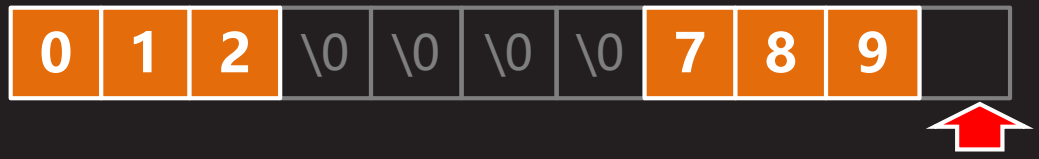
- 可以通过lseek函数将文件读写位置设到文件尾之后
- 在超过文件尾的位置上写入数据，将在文件中形成空洞，位于文件中但没有被写过的字节都被设为0
- 文件空洞不占用磁盘空间，但被计算在文件大小之内

知识讲解

write (fd, "012", 3);

lseek (fd, 4, SEEK_CUR);

write (fd, "789", 3);



顺序与随机读写

【参见：seek.c】

课堂练习

- 顺序与随机读写

系统与标准I/O

Technology Tarena 达内科技

系统I/O

系统I/O

- 当系统调用函数被执行时，需要在用户态和内核态之间来回切换，因此频繁执行系统调用函数会严重影响性能
  - unsigned int i;
    for (i = 0; i < 10000000; ++i)
        write (fd, &i, sizeof (i));

知识讲解

$ time sysio

```
real    0m33.385s
user    0m0.752s
sys     0m32.586s
```

系统I/O

【参见：sysio.c】

课堂练习

- 系统I/O

Technology Tarena 达内科技

标准I/O

标准I/O

- 标准库做了必要的优化，内部维护一个缓冲区，只在满足特定条件时才将缓冲区与系统内核同步，借此降低执行系统调用的频率，减少进程在用户态和内核态之间来回切换的次数，提高运行性能
  - unsigned int i;
    for (i = 0; i < 10000000; ++i)
        fwrite (&i, sizeof (i), 1, fp);

$ time stdio

```
real    0m13.012s
user    0m0.176s
sys     0m12.805s
```

知识讲解

标准I/O

【参见：stdio.c】

课堂练习

- 标准I/O

Technology Tarena 达内科技

总结和答疑