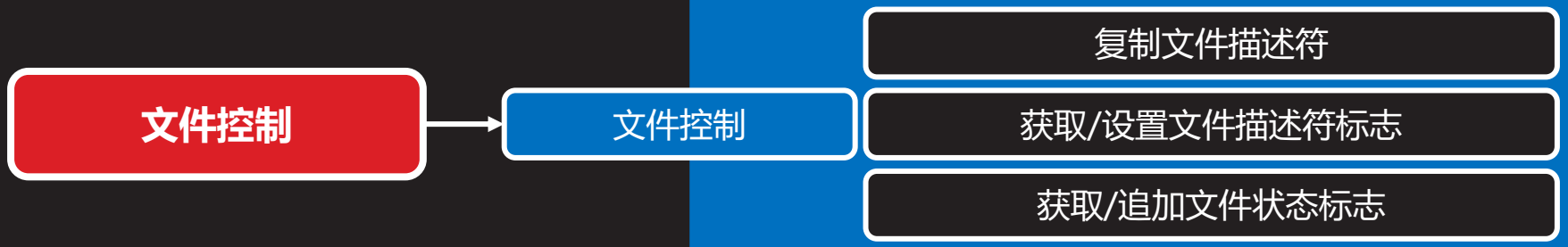
Technology Tarena 达内科技

Unix系统高级编程

文件控制和文件锁 Unit10

文件控制



Technology Tarena 达内科技

文件控制

复制文件描述符

- 复制文件描述符表项到指定位置

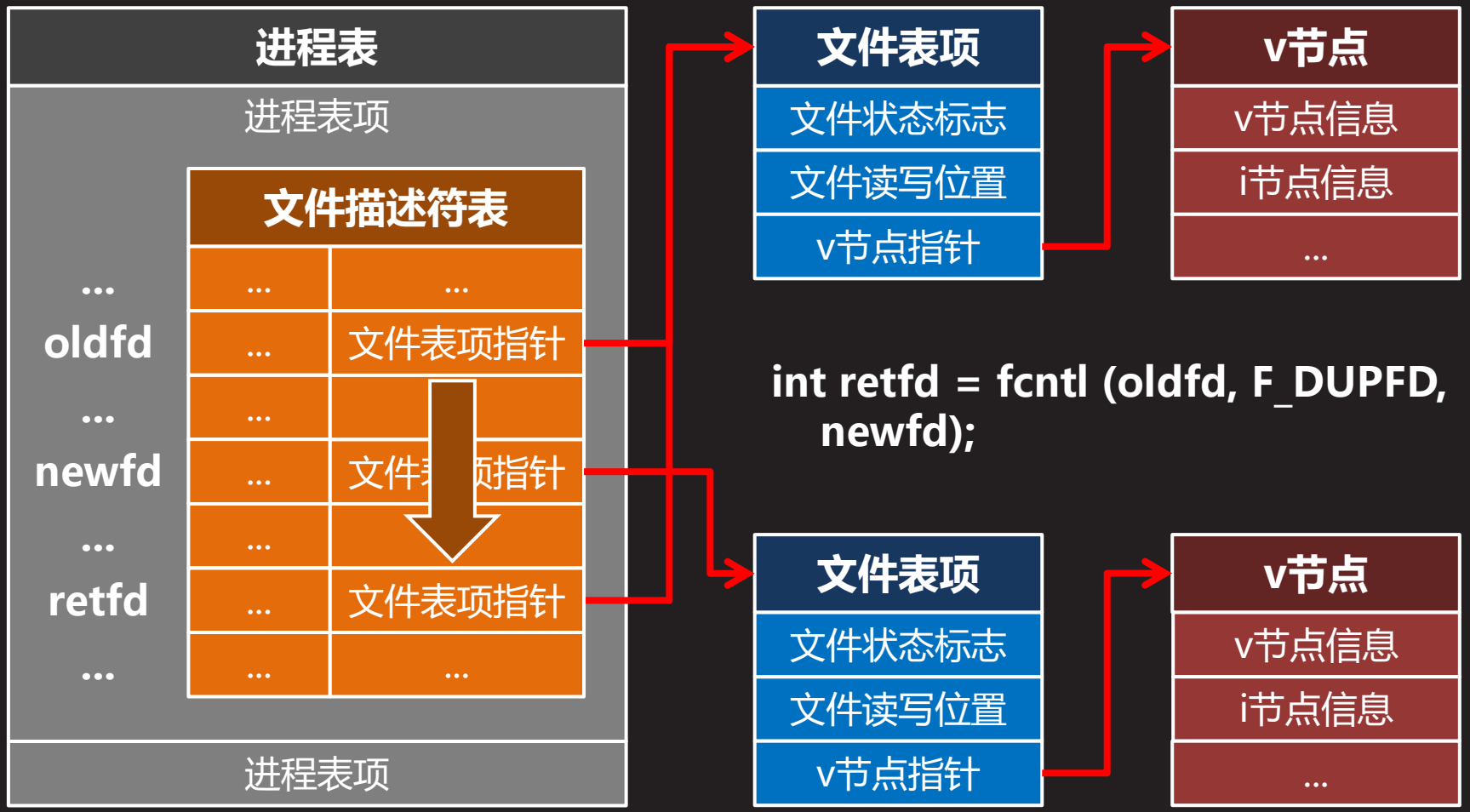
```
#include <fcntl.h>

int fcntl (int oldfd, int cmd, int newfd);

                        成功返回目标文件描述符，失败返回-1
```

  - ***oldfd***：源文件描述符
  - ***cmd***：控制命令，取F_DUPFD
  - ***newfd***：目标文件描述符
- fcntl(F_DUPFD)的功能与dup2函数几乎完全一样，唯一的不同就是当目标文件描述符***newfd***处于打开状态时，并不关闭它，而是另外找一个比它大的最小未用值作为目标

知识讲解

复制文件描述符（续1）

知识讲解



复制文件描述符

【参见：dupfd.c】

课堂练习

- 复制文件描述符

获取/设置文件描述符标志

知识讲解

- 文件描述符标志
  - 如前所述：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。其中文件描述符标志用于表示特定文件描述符的属性
  - 文件描述符标志由多个二进制位组合而成，每个二进制位表示某一方面的属性
  - 截至目前，只有一个文件描述符标志位有意义，该位被定义为FD_CLOEXEC宏，表示在通过exec函数所创建的进程中，相应的文件描述符是否被关闭
    - 0 - 不关闭(缺省)
    - 1 - 关闭

获取/设置文件描述符标志（续1）

- 获取文件描述符标志

```
#include <fcntl.h>

int fcntl (int fd, int cmd);
```

成功返回文件描述符标志，失败返回-1

  - *fd*：文件描述符
  - *cmd*：控制命令，取F_GETFD

- 例如
  -
```
int flags = fcntl (fd, F_GETFD);
if (flags & FD_CLOEXEC)
    printf ("文件描述符%d将在新进程中被关闭。\n", fd);
```

知识讲解

获取/设置文件描述符标志（续2）

- 设置文件描述符标志

```
#include <fcntl.h>

int fcntl (int fd, int cmd, int flags);
```

成功返回0，失败返回-1

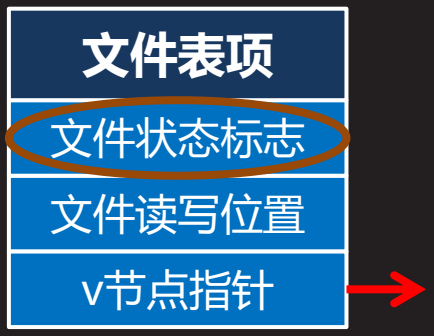
  - *fd*：文件描述符
  - *cmd*：控制命令，取F_SETFD
  - *flags*：文件描述符标志
- 例如
  - int flags = fcntl (fd, F_GETFD);
    if (fcntl (fd, F_SETFD, flags | FD_CLOEXEC) == -1) {
        perror ("fcntl"); exit (EXIT_FAILURE); }

知识讲解

获取/追加文件状态标志

知识讲解

- 文件状态标志
  - 如前所述：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。其中文件状态标志用于表示特定文件描述符的状态
  - 文件状态标志由多个二进制位组合而成，每个二进制位表示某一方面的状态，例如
    - O_RDONLY - 只读
    - O_WRONLY - 只写
    - O_RDWR - 读写
    - O_APPEND - 追加
    - O_CREAT - 创建
    - O_EXCL - 排斥
    - O_TRUNC - 清空
    - O_SYNC - 同步
    - O_ASYNC - 异步
    - O_NONBLOCK - 非阻塞



  - 文件状态标志在调用open函数时指定，保存在文件表项中

获取/追加文件状态标志（续1）

- 获取文件状态标志

```
#include <fcntl.h>

int fcntl (int fd, int cmd);

                    成功返回文件状态标志，失败返回-1
```

  - *fd*：文件描述符
  - *cmd*：控制命令，取F_GETFL
- 与文件创建有关的三个标志位O_CREAT、O_EXCL和O_TRUNC不能获取
- 只读标志O_RDONLY的值为0，不能用位与检测

知识讲解

获取/追加文件状态标志（续2）

- 例如
  - int flags = fcntl (fd, F_GETFL);
    if ((flags & O_ACCMODE) == O_RDONLY)
        printf ("只读\n");
    if (flags & O_WRONLY)
        printf ("只写\n");
    if (flags & O_RDWR)
        printf ("读写\n");
    if (flags & O_APPEND)
        printf ("追加\n");
    if (flags & O_SYNC)
        printf ("同步\n");
    …

知识讲解

获取/追加文件状态标志（续3）

- 追加文件状态标志

```
#include <fcntl.h>

int fcntl (int fd, int cmd, int flags);
```

成功返回0，失败返回-1

  - *fd*：文件描述符
  - *cmd*：控制命令，取F_SETFL
  - *flags*：文件状态标志
- 该函数并非用新的文件状态标志取代原有标志，而是在原有标志的基础上，追加(即位或)新的标志

知识讲解

获取/追加文件状态标志（续4）

- 只有O_APPEND和O_NONBLOCK两个标志位可以追加
  - O_RDONLY、O_WRONLY和O_RDWR本身就是互斥的，不可能组合，O_SYNC和O_ASYNC的情况也是一样，不可能既同步又异步，O_CREAT、O_EXCL和O_TRUNC只在创建文件时起作用，创建完了再加上去没有任何意义，因此只有O_APPEND和O_NONBLOCK两个标志位有与其它标志位组合使用的可能，如追加着写，或者非阻塞地读
- 例如
  - if (fcntl (fd, F_SETFL, O_APPEND | O_NONBLOCK) == -1) { perror ("fcntl"); exit (EXIT_FAILURE); }

知识讲解

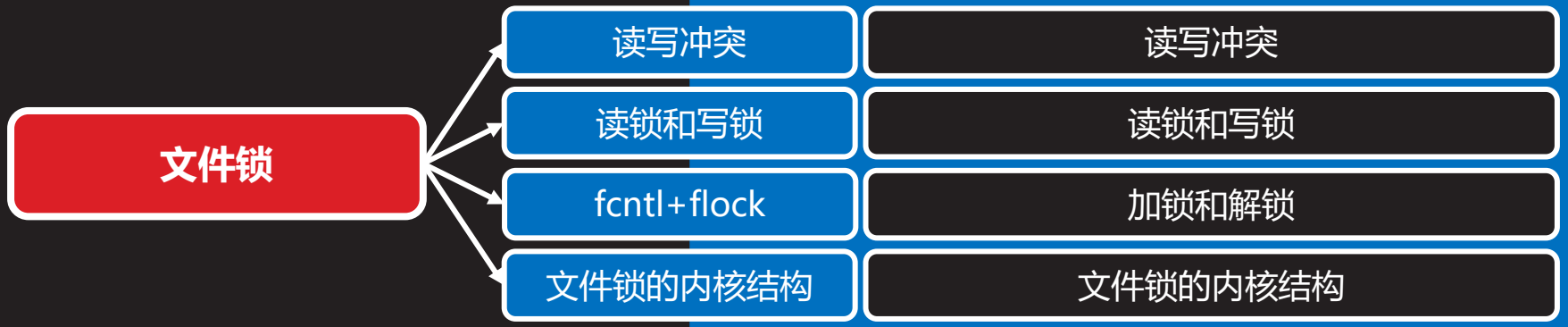
文件状态标志

【参见：flags.c】

课堂练习

- 文件状态标志

文件锁


文件锁
读写冲突
读锁和写锁
fcntl+flock
文件锁的内核结构
读写冲突
读锁和写锁
加锁和解锁
文件锁的内核结构

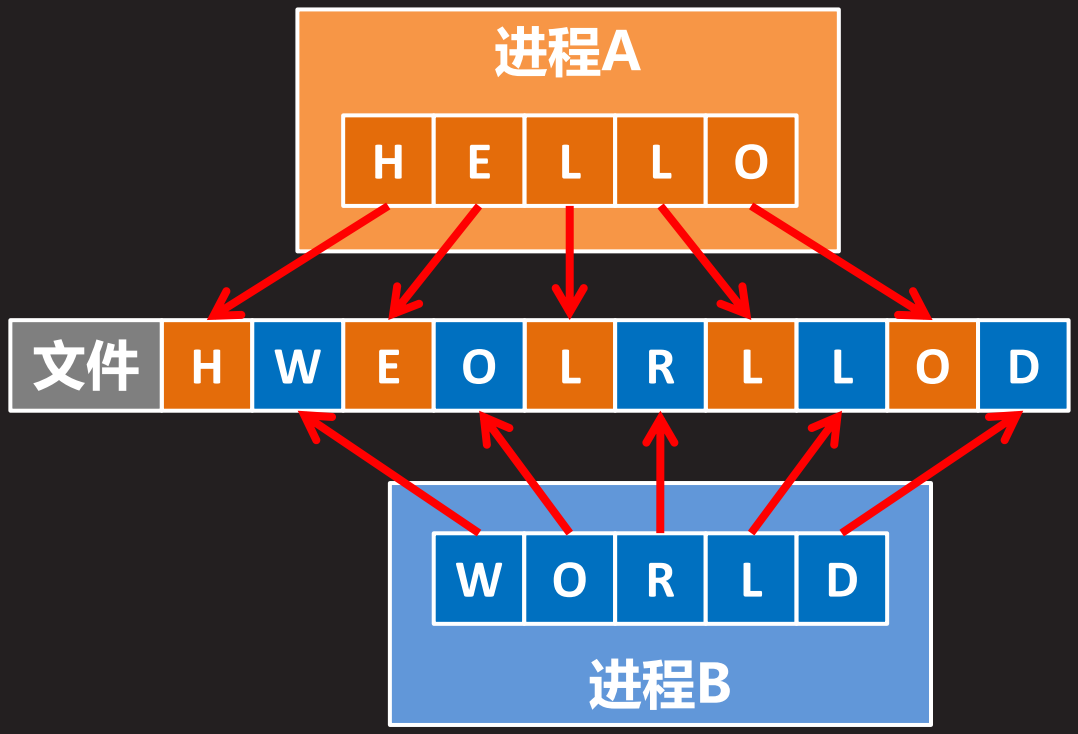
Technology Tarena 达内科技

读写冲突

读写冲突

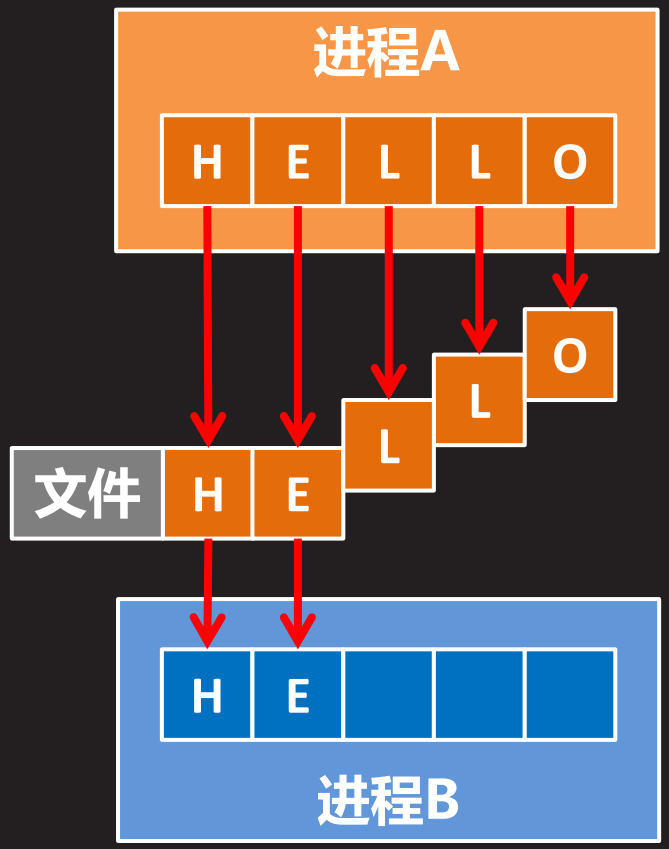
- 如果两个或两个以上的进程同时向一个文件的某个特定区域写入数据，那么最后写入文件的数据极有可能因为写操作的交错而产生混乱

知识讲解



读写冲突（续1）

- 如果一个进程写而其它进程同时在读一个文件的某个特定区域，那么读出的数据极有可能因为读写操作的交错而不完整



知识讲解

读写冲突（续2）

- 多个进程同时读一个文件的某个特定区域，不会有任何问题，它们只是各自把文件中的数据拷贝到各自的缓冲区中，并不会改变文件的内容，相互之间也就不会冲突
- 由此可以得出结论，为了避免在读写同一个文件的同一个区域时发生冲突，进程之间应该遵循以下规则
  - 如果一个进程正在写，那么其它进程既不能写也不能读
  - 如果一个进程正在读，那么其它进程不能写但是可以读

知识讲解

文件的某个区域正在被访问	期望访问	
	读取	写入
无人访问	OK	OK
多人在读	OK	NO
一人在写	NO	NO

Technology Tarena 达内科技

读写冲突

【参见：write.c、read.c】

课堂练习

- 读写冲突

读锁和写锁

读锁和写锁

- 为了避免多个进程在读写同一个文件的同一个区域时发生冲突，Unix/Linux系统引入了文件锁机制，并把文件锁分为读锁和写锁两种，它们的区别在于
  - 读锁：共享锁，对一个文件的特定区域可以加多把读锁
  - 写锁：排它锁，对一个文件的特定区域只能加一把写锁
- 基于锁的操作模型是：读/写文件中的特定区域之前，先加上读/写锁，锁成功了再读/写，读/写完成以后再解锁

知识讲解

文件的某个区域当前拥有锁	期望加锁	
	读锁	写锁
无任何锁	OK	OK
多把读锁	OK	NO
一把写锁	NO	NO

读锁和写锁（续1）

- 假设进程A期望访问某文件的A区，同时进程B期望访问该文件的B区，而A区和B区存在部分重叠，分情况讨论
  - 第一种情况：进程A正在写，进程B也想写

知识讲解

进程A	进程B
打开文件，准备写A区	打开文件，准备写B区
给A区加写锁，成功	
写A区	给B区加写锁，失败，阻塞
写完，解锁A区	从阻塞中恢复，B区被加上写锁
	写B区
	写完，解锁B区
关闭文件	关闭文件

读锁和写锁（续2）

- 假设进程A期望访问某文件的A区，同时进程B期望访问该文件的B区，而A区和B区存在部分重叠，分情况讨论
  - 第二种情况：进程A正在写，进程B却想读

知识讲解

进程A	进程B
打开文件，准备写A区	打开文件，准备读B区
给A区加写锁，成功	
写A区	给B区加读锁，失败，阻塞
写完，解锁A区	从阻塞中恢复，B区被加上读锁
	读B区
	读完，解锁B区
关闭文件	关闭文件

读锁和写锁（续3）

- 假设进程A期望访问某文件的A区，同时进程B期望访问该文件的B区，而A区和B区存在部分重叠，分情况讨论
  - 第三种情况：进程A正在读，进程B却想写

知识讲解

进程A	进程B
打开文件，准备读A区	打开文件，准备写B区
给A区加读锁，成功	
读A区	给B区加写锁，失败，阻塞
读完，解锁A区	从阻塞中恢复，B区被加上写锁
	写B区
	写完，解锁B区
关闭文件	关闭文件

读锁和写锁（续4）

- 假设进程A期望访问某文件的A区，同时进程B期望访问该文件的B区，而A区和B区存在部分重叠，分情况讨论
  - 第四种情况：进程A正在读，进程B也想读

知识讲解

<table>
<tr><th>进程A</th><th>进程B</th></tr>
<tr><td>打开文件，准备读A区</td><td>打开文件，准备读B区</td></tr>
<tr><td>给A区加读锁，成功</td><td></td></tr>
<tr><td rowspan="2">读A区</td><td>给B区加读锁，成功</td></tr>
<tr><td rowspan="2">读B区</td></tr>
<tr><td>读完，解锁A区</td></tr>
<tr><td></td><td>读完，解锁B区</td></tr>
<tr><td>关闭文件</td><td>关闭文件</td></tr>
</table>

读锁和写锁（续5）

- 劝谏锁和强制锁
  - 从前述基于锁的操作模型可以看出，锁机制之所以能够避免读写冲突，关键在于参与读写的多个进程都在按照一套模式——先加锁，再读写，最后解锁——按部就班地执行。这就形成了一套协议，只要参与者无一例外地遵循这套协议，读写就是安全的。反之，如果哪个进程不遵守这套协议，完全无视锁的存在，想读就读，想写就写，即便有锁，对它也起不到任何约束作用。因此，这样的锁机制被称为劝谏锁或协议锁

知识讲解

读锁和写锁（续6）

- 劝谏锁和强制锁
  - 另一种情况则更具有强制约束性，一旦某进程对文件的全部或部分加了锁，它会直接影响其它进程对该文件的I/O调用。比如前面例子中的第一种情况，进程A对A区加了写锁，即便进程B对B区不加写锁，它的write系统调用也会阻塞或者返回失败，直到进程A对A区解锁，进程B对B区的write操作才会成功。这样的锁机制一般被称为强制锁
  - 大多数Unix和类Unix操作系统都只提供劝谏锁，SVR4是提供强制锁的唯一操作系统，而且它要求能够被加强制锁的文件必须带有设置组ID位，同时不能带有组执行位

Technology Tarena 达内科技

fcntl+flock

加锁和解锁

- 对给定文件的特定区域加锁或解锁

```
#include <fcntl.h>

int fcntl (int fd, int cmd, struct flock* lock);

                                成功返回0，失败返回-1
```

  - *fd*：文件描述符
  - *cmd*：控制命令，可取以下值

    `F_SETLKW` - 阻塞模式

    `F_SETLK` - 非阻塞模式

    具体是加锁还是解锁，加的是读锁还是写锁，由*lock*参数决定。若因其它进程持有锁而导致加锁失败，该函数会阻塞(阻塞模式)或返回-1并设错误码为EAGAIN(非阻塞模式)

知识讲解

加锁和解锁（续1）

- 对给定文件的特定区域加锁或解锁
  - *lock*：锁结构，描述锁操作的具体细节

```
struct flock {
  short int l_type;    // 锁操作的类型：
                       // F_RDLCK/F_WRLCK/F_UNLCK
                       // 加读锁/加写锁/解锁
  short int l_whence; // 锁区偏移起点：
                       // SEEK_SET/SEEK_CUR/SEEK_END
                       // 文件头/当前位置/文件尾
  off_t     l_start;  // 锁区偏移字节：从l_whence开始
  off_t     l_len;    // 锁区字节长度：0表示锁到文件尾
  pid_t     l_pid;    // 加锁进程标识：-1表示自动设置
};
```


加锁和解锁（续2）

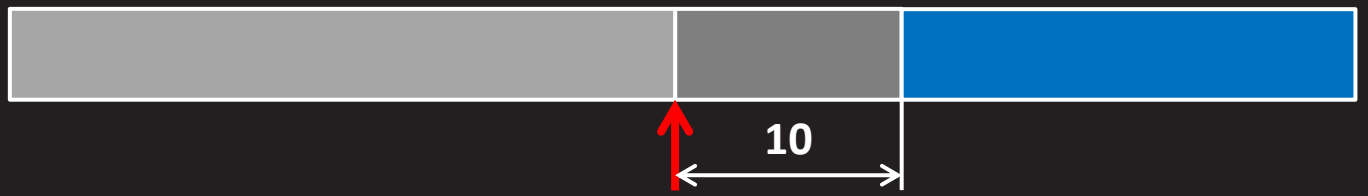
- 例如
  - 对相对于文件头10字节开始的20字节以阻塞模式加读锁


```
struct flock lock;
lock.l_type = F_RDLCK;
lock.l_whence = SEEK_SET;
lock.l_start = 10;
lock.l_len = 20;
lock.l_pid = -1;
if (fcntl (fd, F_SETLKW, &lock) == -1) {
    perror ("fcntl");
    exit (EXIT_FAILURE);
}
```

知识讲解

加锁和解锁（续3）

- 例如
  - 对相对于当前位置10字节开始到文件尾以非阻塞模式加写锁




```
struct flock lock;
lock.l_type = F_WRLCK;
lock.l_whence = SEEK_CUR;
lock.l_start = 10;
lock.l_len = 0;
lock.l_pid = -1;
if (fcntl (fd, F_SETLK, &lock) == -1) {
    if (errno != EAGAIN) {
        perror ("fcntl"); exit (EXIT_FAILURE); }
    printf ("暂时不能加锁，稍后再试...\n");
}
```

知识讲解

加锁和解锁（续4）

- 例如
  - 对整个文件解锁

```
struct flock lock;
lock.l_type = F_UNLCK;
lock.l_whence = SEEK_SET;
lock.l_start = 0;
lock.l_len = 0;
lock.l_pid = -1;
if (fcntl (fd, F_SETLK, &lock) == -1) {
    perror ("fcntl");
    exit (EXIT_FAILURE);
}
```

知识讲解

加锁和解锁（续5）

- 当通过close函数关闭文件描述符时，调用进程在该文件描述符上所加一切锁将被自动解除
- 当进程终止时，该进程在所有文件描述符上所加的一切锁将被自动解除
- 文件锁仅在不同进程之间起作用，同一个进程的不同线程不能通过文件锁解决读写冲突问题
- 通过fork/vfork函数创建的子进程，不继承父进程所加的任何文件锁
- 通过exec函数族创建的新进程，会继承调用进程所加的全部文件锁，除非某文件描述符带有FD_CLOEXEC标志

知识讲解

加锁和解锁

【参见：wlock.c、rlock.c】

课堂练习

- 加锁和解锁

Technology Tarena 达内科技

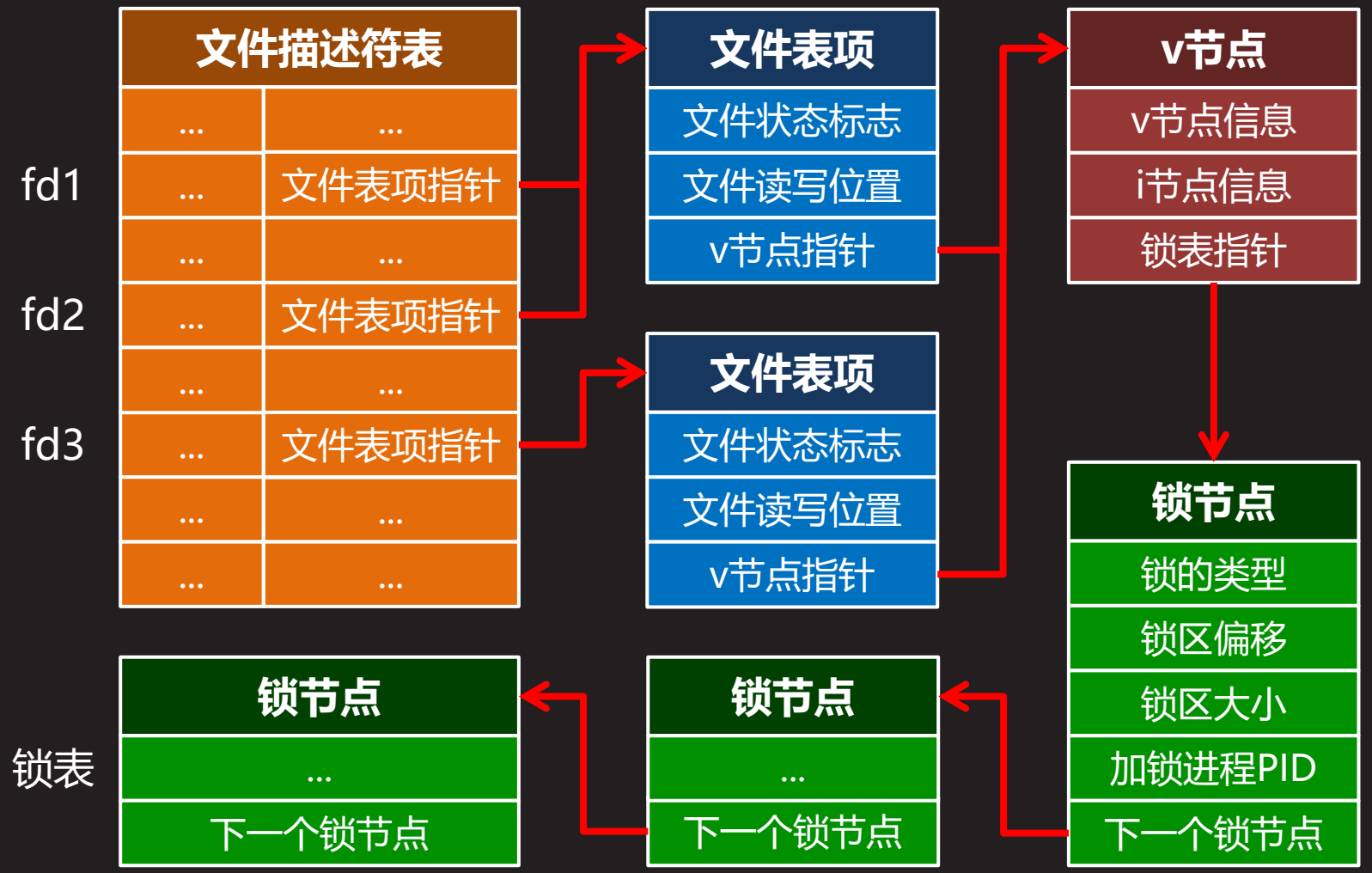
文件锁的内核结构

文件锁的内核结构

- 每次对给定文件的特定区域加锁，都会通过fcntl函数向系统内核传递flock结构体，该结构体中包含了有关锁的一切细节，诸如锁的类型(读锁/写锁)，锁区的起始位置和大小，甚至加锁进程的PID(填-1由系统自动设置)
- 系统内核会收集所有进程对该文件所加的各种锁，并把这些flock结构体中的信息，以链表的形式组织成一张锁表，而锁表的起始地址就保存在该文件的v节点中
- 任何一个进程通过fcntl函数对该文件加锁，系统内核都要遍历这张锁表，一旦发现有与欲加之锁构成冲突的锁即阻塞或报错，否则即将欲加之锁插入锁表，而解锁的过程实际上就是调整或删除锁表中的相应节点

文件锁的内核结构（续1）

知识讲解



文件锁的内核结构（续2）

- 关闭上面图中fd1、fd2和fd3中的任何一个文件描述符，都将解除调用进程对该文件所加的全部锁。系统内核并不清楚也不关心锁是用哪个文件描述符加的，它唯一在乎的，就是锁是哪个进程对哪个文件加的。只要进程关闭了一个文件描述符，在系统内核看来即是宣称不再使用该文件了(尽管可能还有其它文件描述符引用该文件)，既然如此，解除该进程对该文件所加的一切锁也就是顺理成章的事了

知识讲解

Technology Tarena 达内科技

总结和答疑