Tarena Technology 达内科技

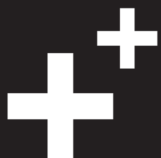
Unix系统高级编程

File 3

DAY06

内容

<table>
<tr><td rowspan="4">上午</td><td>09:00 ~ 09:30</td><td>作业讲解和回顾</td></tr>
<tr><td>09:30 ~ 10:20</td><td>复制文件描述符</td></tr>
<tr><td>10:30 ~ 11:20</td><td>文件同步与控制</td></tr>
<tr><td>11:30 ~ 12:20</td><td rowspan="2">文件锁</td></tr>
<tr><td rowspan="4">下午</td><td>14:00 ~ 14:50</td></tr>
<tr><td>15:00 ~ 15:50</td><td rowspan="2">文件元数据</td></tr>
<tr><td>16:00 ~ 16:50</td></tr>
<tr><td>17:00 ~ 17:30</td><td>总结和答疑</td></tr>
</table>



复制文件描述符

Technology Tarena 达内科技

dup

- 复制文件描述符表项

```
#include <unistd.h>

int dup (int oldfd);
```

成功返回目标文件描述符，失败返回-1

  – ***oldfd***：源文件描述符

- dup函数将***oldfd***参数所对应的文件描述符表项复制到文件描述符表第一个空闲项中，同时返回该表项所对应的文件描述符
- dup函数返回的文件描述符一定是调用进程当前未使用的最小文件描述符

dup（续1）

- dup函数只复制文件描述符表项，不复制文件表项和v节点，因此该函数所返回的文件描述符可以看做是参数文件描述符*oldfd*的副本，它们标识同一个文件表项


```
int oldfd = open (...);
int newfd = dup (oldfd);
...
close (newfd);
close (oldfd);
```

dup（续2）

Tarena Technology 达内科技

知识讲解

- 注意，当关闭文件时，即使是由dup函数产生的文件描述符副本，也应该通过close函数关闭，因为只有当关联于一个文件表项的所有文件描述符都被关闭了，该文件表项才会被销毁，类似地，也只有当关联于一个v节点的所有文件表项都被销毁了，v节点才会被从内存中删除，因此从资源合理利用的角度讲，凡是明确不再继续使用的文件描述符，都应该尽可能及时地用close函数关闭
- dup函数究竟会把***oldfd***参数所对应的文件描述符表项，复制到文件描述符表的什么位置，程序员是无法控制的，这完全由调用该函数时文件描述符表的使用情况决定，因此对该函数的返回值做任何约束性假设都是不严谨的

dup（续3）

- 由dup函数返回的文件描述符与作为参数传递给该函数的文件描述符标识的是同一个文件表项，而文件读写位置是保存在文件表项而非文件描述符表项中的，因此通过这些文件描述符中的任何一个，对文件进行读写或随机访问，都会影响通过其它文件描述符操作的文件读写位置。这与多次通过open函数打开同一个文件不同

dup（续4）

Tarena Technology 达内科技

知识讲解

进程表		
进程表项		
	文件描述符表	
…	…	…
fd1	…	文件表项指针
…	…	⇩
fd2	…	文件表项指针
…	…	…
fd3	…	文件表项指针
…	…	…
进程表项		

文件表项
文件状态标志
文件读写位置
v节点指针

文件表项
文件状态标志
文件读写位置
v节点指针

v节点
v节点信息
i节点信息
…

```
int fd1 = open ("same", ...);
int fd2 = dup (fd1);
int fd3 = open ("same", ...);
```

Technology Tarena 达内科技

dup2

知识讲解

- 复制文件描述符表项到指定位置

```
#include <unistd.h>

int dup2 (int oldfd, 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
  - *newfd*：目标文件描述符

- dup2函数的功能与dup函数几乎完全一样，唯一的不同就是允许调用者通过*newfd*参数指定目标文件描述符，正常情况下该函数的返回值应该与*newfd*参数的值相等

dup2（续1）

Technology Tarena 达内科技

知识讲解

- dup2函数在复制由*oldfd*参数所标识的源文件描述符表项时，会首先检查由*newfd*参数所标识的目标文件描述符表项是否空闲，若空闲则直接将前者复制给后者，否则会先将目标文件描述符*newfd*关闭，再行复制

复制文件描述符

【参见：same.c、dup.c】

- 复制文件描述符

文件同步与控制

文件同步

文件同步

- 写缓冲与延迟写
  - 当一个运行在用户空间的进程发起write系统调用时，系统内核进行几项检查，然后直接将通过参数传入的数据块拷贝到一个被称为写缓冲的缓冲区中，随即返回调用进程
  - 稍后，运行在后台的系统内核收集所有这样的“脏”缓冲区，将它们按照一定的顺序排入写队列，并逐一写入磁盘，这个过程被称为回写(writeback)，而这种将回写操作延迟执行的策略被称为延迟写
  - 延迟写一方面使运行在用户空间的调用进程不必等待写磁盘动作(通常较慢)的实际完成，提高其运行速度，另一方面使系统内核得以将实际的写磁盘工作推迟到相对空闲的时候完成，且以批量方式集中处理，提高内核的工作效率

Tarena Technology 达内科技

知识讲解

文件同步（续1）

- 延迟写的潜在风险
  - write系统调用的成功返回，仅表示通过其参数传入的数据块，已被成功地拷贝到由系统内核负责维护的写缓冲中，而此后任何在回写过程中发生错误，如物理磁盘驱动器故障等，都不会被报告给发出写请求的进程
  - 如果在“脏”缓冲区中的数据被实际写入磁盘之前，系统发生了某种不可预期的异常，如硬件故障、突然掉电等，写缓冲中的数据将永远失去被写入磁盘的机会，造成数据丢失(即失步)，这对许多关键业务系统而言无疑是致命的

Technology Tarena 达内科技

文件同步（续2）

- 延迟写的潜在风险
  – 为了降低延迟写的风险，保证写缓冲中的数据变化适时地被同步到磁盘，系统建立了一个写缓冲最大时效机制，并保证所有的“脏”缓冲区在它们超过给定时效前能够被强制同步到磁盘设备。系统管理员可以通过/proc/sys/vm/dirty_expire_centisecs来配置这个值，该值以厘秒(1厘秒=0.01秒=10毫秒)为单位
  – 如果对系统提供的上述风险防范机制仍然心存疑虑，或者希望能够严格控制数据被写入磁盘的时间，那么就必须考虑采用同步I/O的手段，以牺牲性能为代价，换取更高的可靠性和精确性。这对那些关键性的业务数据显得尤其重要

Tarena Technology 达内科技

文件同步（续3）

- 同步文件内容和元数据

```
#include <unistd.h>

int fsync (int fd);
```

成功返回0，失败返回-1

  – *fd*：需要同步的文件描述符
- 调用fsync函数可以保证*fd*参数所标识文件的“脏”数据立即被回写到磁盘上
- 文件描述符*fd*必须是以写方式打开的
- 该函数不仅回写文件的内容数据，诸如文件大小、时间戳等保存在i节点中的元数据也一并被回写

文件同步（续4）

Tarena Technology 达内科技

- fsync函数在磁盘驱动器确认所有与特定文件相关的“脏”数据都被成功回写之前不会返回，因此调用进程一旦从该函数返回成功，就完全有理由确信*fd*参数所标识的文件已经同步完成
- 注意，现代大多数磁盘设备也是带有缓冲区的，fsync函数的同步也只是把数据同步到磁盘设备的缓冲区里，至于这些数据是否真的被保存到物理介质上了，fsync函数是不可能知道的(硬盘可能会撒谎，它通知系统内核缓冲数据已经写到磁盘上了，但实际上它们仍在磁盘缓存中)

知识讲解

文件同步（续5）

Tarena Technology 达内科技

知识讲解

- 只同步文件内容不同步元数据

```
#include <unistd.h>

int fdatasync (int fd);
```

成功返回0，失败返回-1

  - *fd*：需要同步的文件描述符
- fdatasync函数与fsync函数的功能几乎完全一样，唯一的区别在于它只回写文件的内容数据，而诸如文件大小、时间戳等保存在i节点中的元数据，该函数不保证同步，因此fdatasync函数的执行速度比fsync函数要快一些

Technology
Tarena
达内科技

文件同步（续6）

- 同步所有文件的内容和元数据

```
#include <unistd.h>

void sync (void);
```

永远成功，无返回值

  – 标准中的sync函数将系统中所有存在“脏”数据的写缓冲统统排入写队列，随即返回，并不等待写磁盘操作的完成
  – Linux的sync函数一定要等到所有写缓冲中的数据，既包括文件的内容数据也包括文件的元数据，都被实际写入磁盘设备后才返回
  – 在一个繁忙的系统上，一次对sync函数的调用，其耗时可能要长达数分钟之久

文件同步（续7）

Tarena Technology 达内科技

- 如果在打开文件时使用了O_SYNC标志，则所有在这个文件描述符上的写操作都是同步的，即使不调用fsync、fdatasync或者sync函数
- 以同步模式打开的文件，会在I/O等待上消耗大量的运行时间，使用O_SYNC的进程比不使用O_SYNC的进程要慢一两个数量级，这显然是一种无奈之举
- 通过fsync或fdatasync函数实现同步的情况要好得多，毕竟只有那些关键性的业务数据才需要强制同步，总体开销比使用O_SYNC要小得多
- 读操作从来都是同步的，什么时候要什么时候读，要多少读多少，提前读显然不象延迟写那么有价值

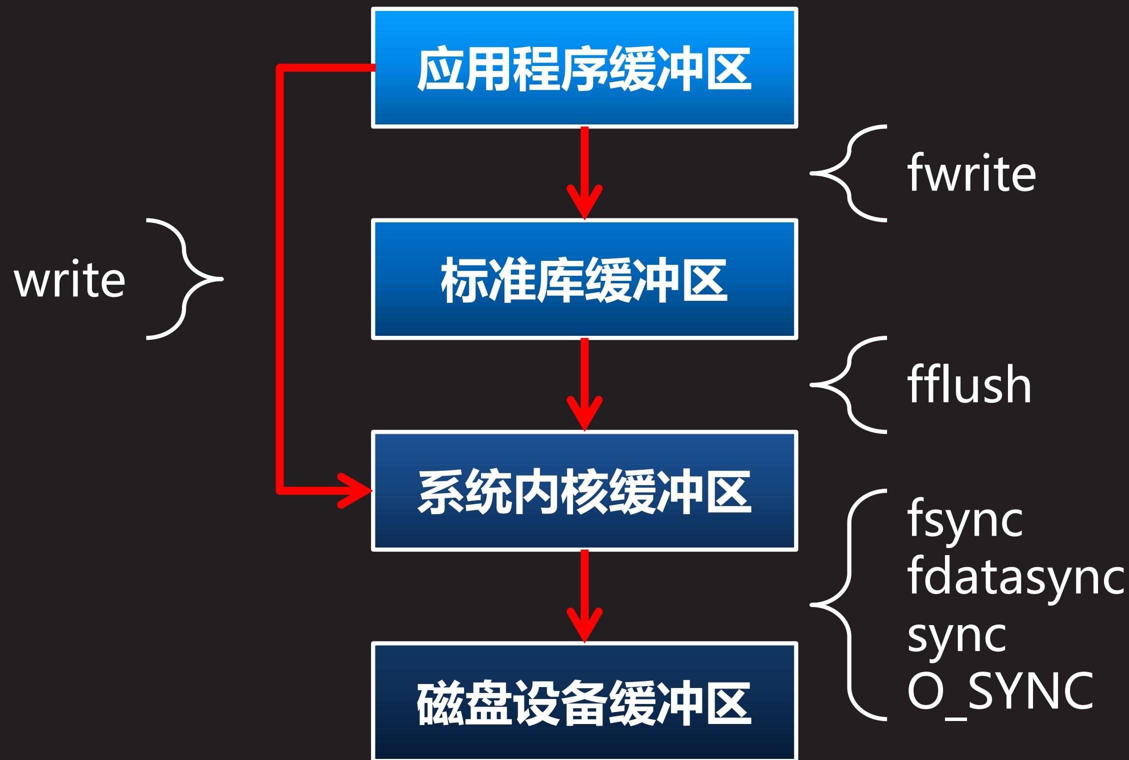
知识讲解

文件同步（续8）

Tarena Technology 达内科技

- 缓冲！缓冲！缓冲！
  - 从应用程序到标准库，再到系统内核，最后到硬件设备，每一层都会维护自己的缓冲区，以改善系统的整体性能

知识讲解



Technology
Tarena
达内科技

Tarena Technology 达内科技

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

复制文件描述符（续1）

Tarena Technology 达内科技

知识讲解

进程表		
进程表项		
	文件描述符表	
…	…	…
oldfd	…	文件表项指针
…	…	
newfd	…	文件表项指针
…	…	
retfd	…	文件表项指针
…	…	…
进程表项		

文件表项
文件状态标志
文件读写位置
v节点指针

v节点
v节点信息
i节点信息
…

```
int retfd = fcntl (oldfd, F_DUPFD,
    newfd);
```

文件表项
文件状态标志
文件读写位置
v节点指针

v节点
v节点信息
i节点信息
…

复制文件描述符

【参见：fcntl.c】

- 复制文件描述符

获取/设置文件描述符标志

Technology Tarena 达内科技

- 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）

Tarena Technology 达内科技

知识讲解

- 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);
```

获取/设置文件描述符标志（续2）

Tarena Technology 达内科技

知识讲解

- 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

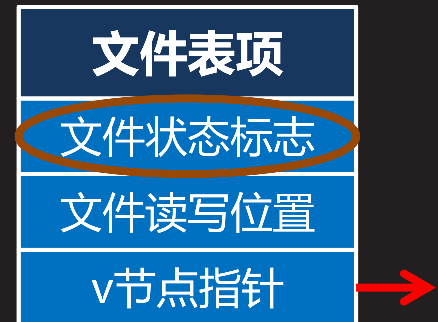
```
int flags = fcntl (fd, F_GETFD);
if (fcntl (fd, F_SETFD, flags | FD_CLOEXEC) == -1) {
    perror ("fcntl"); exit (EXIT_FAILURE); }
```

获取/追加文件状态标志

Tarena Technology
达内科技

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）

Tarena Technology 达内科技

知识讲解

- 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

获取/追加文件状态标志（续2）

Tarena Technology 达内科技

知识讲解

- 例如
  - int flags = fcntl (fd, F_GETFL);

```
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");
...
```

获取/追加文件状态标志（续3）

Technology Tarena 达内科技

知识讲解

- 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

获取/追加文件状态标志（续4）

Tarena Technology 达内科技

知识讲解

- 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

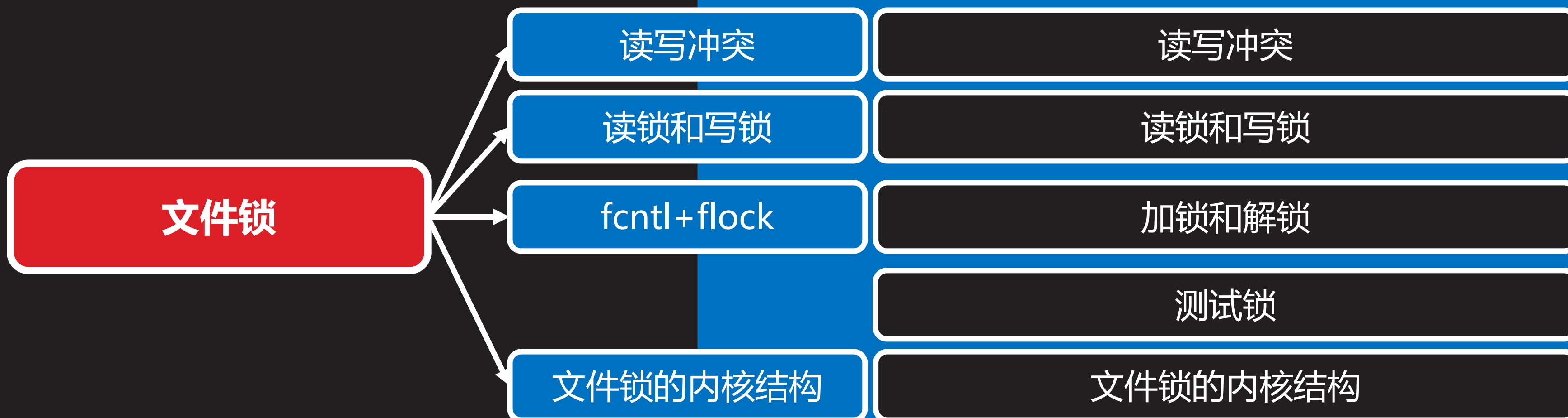
```
– if (fcntl (fd, F_SETFL, O_APPEND |
      O_NONBLOCK) == -1) {
      perror ("fcntl");
      exit (EXIT_FAILURE);
  }
```

文件状态标志

【参见：flags.c】

- 文件状态标志

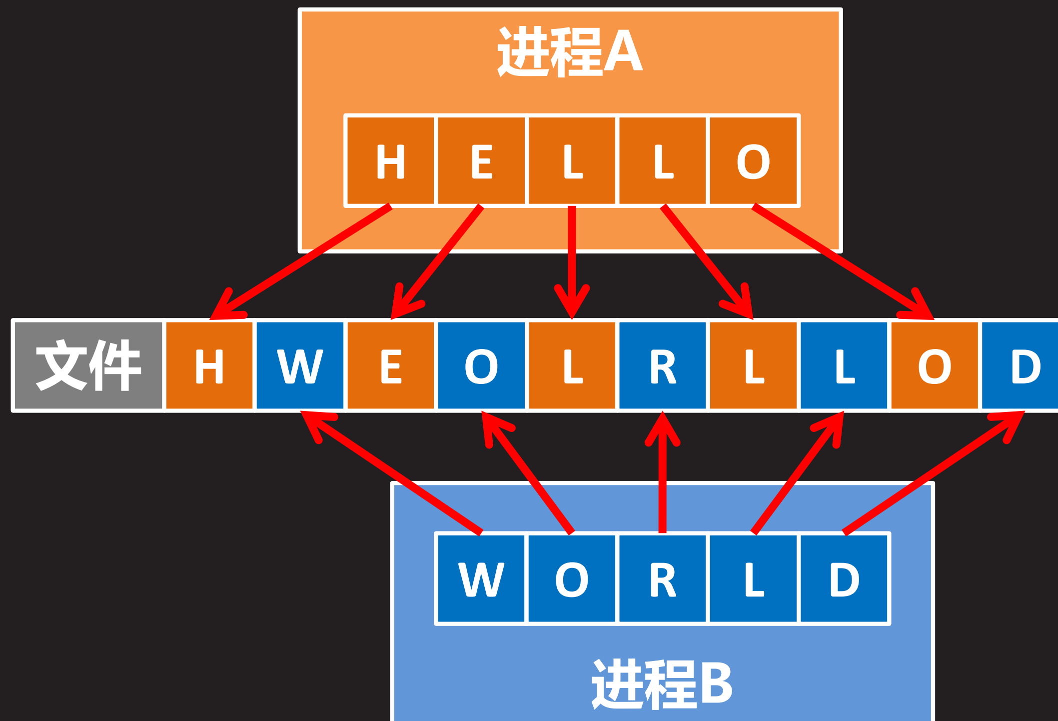
文件锁



Technology
Tarena
达内科技

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



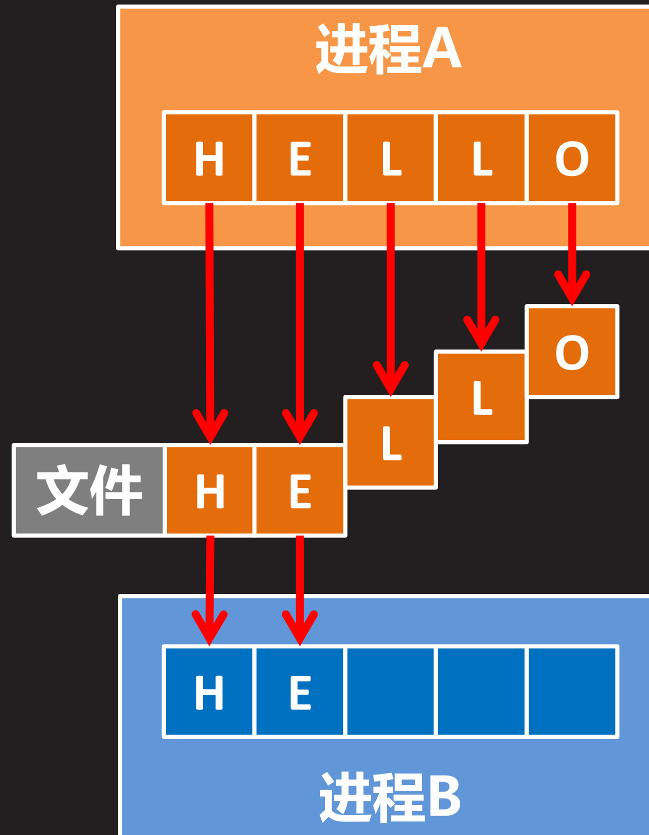
Technology Tarena 达内科技

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）

Technology Tarena 达内科技

知识讲解

- 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

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

读写冲突

【参见：write.c、read.c】

课堂练习

- 读写冲突

Technology Tarena 达内科技

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

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）

Tarena Technology 达内科技

- 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

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

知识讲解

Technology Tarena 达内科技

Tarena Technology 达内科技

读锁和写锁（续3）

知识讲解

- 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

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

进程A	进程B
打开文件，准备读A区	打开文件，准备读B区
给A区加读锁，成功	
读A区	给B区加读锁，成功
	读B区
读完，解锁A区	
	读完，解锁B区
关闭文件	关闭文件

读锁和写锁（续5）

Tarena Technology 达内科技

- 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）

Tarena Technology 达内科技

知识讲解

- 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

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)

加锁和解锁（续1）

Tarena Technology 达内科技

知识讲解

- 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）

Technology Tarena 达内科技

- 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

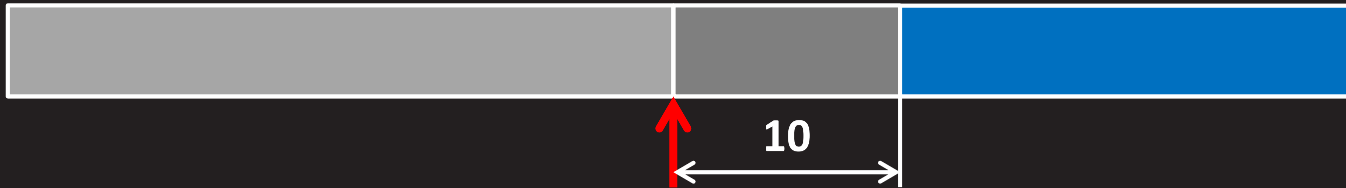
```
struct flock lock;
lock.l_type = F_RDLCK;
lock.l_whence = SEEK_SET;
lock.l_start = 10;
lock.l_len = 20;
lock.l_pid = -1;
if (fcntl (fd, F_SETLKW, &lock) == -1) {
    perror ("fcntl");
    exit (EXIT_FAILURE);
}
```

知识讲解

加锁和解锁（续3）

Tarena Technology 达内科技

- 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

Technology Tarena 达内科技

加锁和解锁（续5）

Tarena Technology 达内科技

- 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】

- 加锁和解锁

测试锁

Tarena Technology 达内科技

知识讲解

- 测试给定文件的特定区域是否可以加锁

```
#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，取F_GETLK
- *lock*：锁结构。调用函数时，输入欲加之锁的具体细节。函数返回时，输出当前锁的具体信息。若其l_type成员的值为F_UNLCK，表示欲加之锁可加，否则可以通过*lock*结构进一步了解究竟哪个进程对文件的哪个区域加了什么锁

测试锁（续1）

- 例如
  - 测试相对于文件头10字节开始的20字节是否可以加读锁，如不可加，打印具体原因

```
struct flock lock = {F_RDLCK, SEEK_SET, 10, 20, -1};
if (fcntl (fd, F_GETLK, &lock) == -1) {
    perror ("fcntl");
    exit (EXIT_FAILURE);
}
if (lock.l_type == F_UNLCK)
    printf ("此锁可加！\n");
else
    why_not (lock);
```

知识讲解

Tarena Technology 达内科技

测试锁（续2）

Tarena Technology 达内科技

知识讲解

- 例如

```
– void why_not (struct flock lock) {
    printf ("%d进程", lock.l_pid);
    switch (lock.l_whence) {
        case SEEK_SET: printf ("在距文件头"); break;
        case SEEK_CUR: printf ("在距当前位置"); break;
        case SEEK_END: printf ("在距文件尾"); break; }
    printf ("%ld字节处，为%ld字节加了",
        lock.l_start, lock.l_len);
    switch (lock.l_type) {
        case F_RDLCK: printf ("读锁。\n"); break;
        case F_WRLCK: printf ("写锁。\n"); break; }
  }
```

测试锁

【参见：lock1.c、lock2.c】

课堂练习

- 测试锁

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）

Technology Tarena 达内科技

知识讲解

	文件描述符表	
	…	…
fd1	…	文件表项指针
	…	…
fd2	…	文件表项指针
	…	…
fd3	…	文件表项指针
	…	…
	…	…

文件表项
文件状态标志
文件读写位置
v节点指针

文件表项
文件状态标志
文件读写位置
v节点指针

v节点
v节点信息
i节点信息
锁表指针

锁节点
锁的类型
锁区偏移
锁区大小
加锁进程PID
下一个锁节点

锁表

锁节点
…
下一个锁节点

锁节点
…
下一个锁节点

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

Tarena Technology 达内科技

文件元数据

Technology
Tarena
达内科技

获取文件元数据

Tarena Technology 达内科技

- 从i节点中提取文件的元数据，即文件的属性信息

```
#include <sys/stat.h>

int stat (const char* path, struct stat* buf);
int fstat (int fd, struct stat* buf);
int lstat (const char* path, struct stat* buf);
```

成功返回0，失败返回-1

知识讲解

  – *path*：文件路径
  – *buf*：文件元数据结构
  – *fd*：文件描述符
- 将指定文件的元数据填到***buf***参数所指向的元数据结构中
- lstat与另两个函数的区别仅在于它不跟踪符号链接

Technology Tarena 达内科技

文件元数据结构

知识讲解

- stat函数族通过stat结构体，向调用者输出文件的元数据
  - ```
    struct stat {
      dev_t       st_dev;     // 设备ID
      ino_t       st_ino;     // i节点号
      mode_t      st_mode;    // 文件类型和权限
      nlink_t     st_nlink;   // 硬链接数
      uid_t       st_uid;     // 用户ID
      gid_t       st_gid;     // 组ID
      dev_t       st_rdev;    // 特殊设备ID
      off_t       st_size;    // 总字节数
      blksize_t   st_blksize; // I/O块字节数
      blkcnt_t    st_blocks;  // 占用块(512字节)数
      time_t      st_atime;   // 最后访问时间
      time_t      st_mtime;   // 最后修改时间
      time_t      st_ctime;   // 最后状态改变时间
    };
    ```

    ```

Technology Tarena 达内科技

# 文件类型和权限

Tarena Technology 达内科技

知识讲解

- stat结构的st_mode成员表示文件的类型和权限，该成员在stat结构中被声明为mode_t类型，其原始类型在32位系统中被定义为unsigned int，即32位无符号整数，但到目前为止，只有其中的低18位有意义
- 用18位二进制数($B_{17}...B_0$)表示的文件类型和权限，从高到低可被分为五组
  - $B_{17}B_{16}B_{15}B_{14}B_{13}B_{12}$：文件类型
  - $B_{11}B_{10}B_9$：设置用户ID、设置组ID和粘滞
  - $B_8B_7B_6$：拥有者用户的读、写和执行权限
  - $B_5B_4B_3$：拥有者组的读、写和执行权限
  - $B_2B_1B_0$：其它用户的读、写和执行权限

# 文件类型和权限（续1）

- 文件类型：$B_{17}...B_{12}$

mode_t st_mode; // unsigned int

| $B_{17}$ | $B_{16}$ | $B_{15}$ | $B_{14}$ | $B_{13}$ | $B_{12}$ | | |
|---|---|---|---|---|---|---|---|
| | | 1 | | | | S_IFREG | 普通文件 |
| | | | 1 | | | S_IFDIR | 目录 |
| | | 1 | 1 | | | S_IFSOCK | 本地套接字 |
| | | | | 1 | | S_IFCHR | 字符设备 |
| | | | 1 | 1 | | S_IFBLK | 块设备 |
| | | 1 | | 1 | | S_IFLNK | 符号链接 |
| | | | | | 1 | S_IFIFO | 有名管道 |
| | | 1 | 1 | 1 | 1 | S_IFMT | 掩码 |

Tarena Technology 达内科技

# 文件类型和权限（续2）

知识讲解

- 设置用户ID、设置组ID和粘滞：$B_{11}B_{10}B_9$

mode_t st_mode; // unsigned int

| | | | | |
|---|---|---|---|---|
| 1 | | | S_ISUID | 设置用户ID |
| | 1 | | S_ISGID | 设置组ID |
| | | 1 | S_ISVTX | 粘滞 |
| 4 | 2 | 1 | | |

# 文件类型和权限（续3）

Tarena Technology 达内科技

知识讲解

- 设置用户ID、设置组ID和粘滞：$B_{11}B_{10}B_9$
  - 带有设置用户ID位($B_{11}$)的可执行文件
    - 系统中的每个进程其实都有两个用户ID，一个叫实际用户ID，一个叫有效用户ID
    - 进程的实际用户ID继承自其父进程的实际用户ID。当一个用户通过合法的用户名和口令登录系统以后，系统就会为他启动一个Shell进程，Shell进程的实际用户ID就是该登录用户的用户ID。该用户在Shell下启动的任何进程都是Shell进程的子进程，自然也就继承了Shell进程的实际用户ID
    - 一个进程的用户身份决定了它可以访问哪些资源，比如读、写或者执行某个文件。但真正被用于权限验证的并不是进程的实际用户ID，而是其有效用户ID。一般情况下，进程的有效用户ID就取自其实际用户ID，可以认为二者是等价的
    - 但是，如果用于启动该进程的可执行文件带有设置用户ID位，即$B_{11}$位为1，那么该进程的有效用户ID就不再取自其实际用户ID，而是取自可执行文件的拥有者用户ID

# 文件类型和权限（续4）

Tarena Technology 达内科技

知识讲解

- 设置用户ID、设置组ID和粘滞：$B_{11}B_{10}B_9$
  - 带有设置用户ID位($B_{11}$)的可执行文件
    - 系统管理员常用这种方法提升普通用户的权限，让他们有能力去完成一些本来只有root用户才能完成的任务。例如，他可以为某个拥有者用户为root的可执行文件添加设置用户ID位，这样一来无论运行这个可执行文件的实际用户是谁，启动起来的进程的有效用户ID都是root，凡是root用户可以访问的资源，该进程都可以访问。当然，具体访问哪些资源，以何种方式访问，还要由这个可执行文件的代码来决定。作为一个安全的操作系统，不可能允许一个低权限用户在高权限状态下为所欲为。如通过passwd命令修改口令
  - 带有设置组ID位($B_{10}$)的可执行文件
    - 与设置用户ID位的情况类似，如果一个可执行文件带有设置组ID位，即$B_{10}$位为1，那么运行该可执行文件所得到的进程，它的有效组ID同样不取自其实际组ID，而是取自可执行文件的拥有者组ID

# 文件类型和权限（续5）

Tarena Technology 达内科技

知识讲解

- 设置用户ID、设置组ID和粘滞：$B_{11}B_{10}B_9$
  - 带有设置用户ID位($B_{11}$)的不可执行文件
    - 一个不可执行的文件带有设置用户ID位是毫无意义的
  - 带有设置组ID位($B_{10}$)的不可执行文件
    - 一个不可执行的文件带有设置组ID位同样毫无意义，但在某些系统上，这种无意义的组合恰被用于作为强制锁的标志
  - 带有设置用户ID位($B_{11}$)的目录
    - 目录带有设置用户ID位没有任何意义
  - 带有设置组ID位($B_{10}$)的目录
    - 在该目录下创建的文件及子目录，其拥有者组取自该目录的拥有者组，而非创建者用户所隶属的组

# 文件类型和权限（续6）

Tarena Technology 达内科技

知识讲解

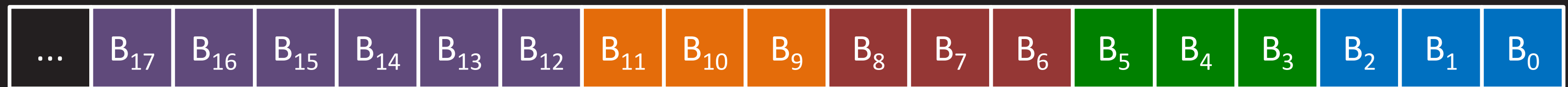
- 设置用户ID、设置组ID和粘滞：$B_{11}B_{10}B_9$
  - 带有粘滞位($B_9$)的可执行文件
    - 在其首次运行并结束后， 其代码区被连续地保存在磁盘交换区中，而一般磁盘文件的数据块是离散存放的。因此，下次运行该程序可以获得较快的载入速度
  - 带有粘滞位($B_9$)的不可执行文件
    - 不可执行文件带有粘滞位没有任何意义
  - 带有粘滞位($B_9$)的目录
    - 除root以外的任何用户在该目录下，都只能删除或者更名那些属于自己的文件或子目录，而对于其它用户的文件或子目录，既不能删除也不能改名
    - 如/tmp目录

# 文件类型和权限（续7）

Technology
Tarena
达内科技

- 拥有者用户的读、写和执行权限：$B_8B_7B_6$

mode_t st_mode; // unsigned int




| | | $B_8$ | $B_7$ | $B_6$ |
|---|---|---|---|---|
| 拥有者用户可读 | S_IRUSR | 1 | | |
| 拥有者用户可写 | S_IWUSR | | 1 | |
| 拥有者用户可执行 | S_IXUSR | | | 1 |
| 掩码 | S_IRWXU | 1 | 1 | 1 |
| | | 4 | 2 | 1 |

知识讲解

# 文件类型和权限（续8）

- 拥有者组的读、写和执行权限：$B_5B_4B_3$

mode_t st_mode; // unsigned int

| | | | | |
|---|---|---|---|---|
| 拥有者组可读 | S_IRGRP | 1 | | |
| 拥有者组可写 | S_IWGRP | | 1 | |
| 拥有者组可执行 | S_IXGRP | | | 1 |
| 掩码 | S_IRWXG | 1 | 1 | 1 |
| | | 4 | 2 | 1 |

知识讲解

# 文件类型和权限（续9）

- 其它用户的读、写和执行权限：$B_2B_1B_0$

mode_t st_mode; // unsigned int

| 其它用户可读 | S_IROTH | 1 | | |
|---|---|---|---|---|
| 其它用户可写 | S_IWOTH | | 1 | |
| 其它用户可执行 | S_IXOTH | | | 1 |
| 掩码 | S_IRWXO | 1 | 1 | 1 |
| | | 4 | 2 | 1 |

知识讲解

# 文件类型和权限（续10）

Tarena Technology 达内科技

- 辅助分析文件类型的实用宏
  - S_ISREG() ：是否普通文件
  - S_ISDIR() ：是否目录
  - S_ISSOCK()：是否本地套接字
  - S_ISCHR() ：是否字符设备
  - S_ISBLK() ：是否块设备
  - S_ISLNK() ：是否符号链接
  - S_ISFIFO()：是否有名管道

知识讲解

知识讲解

# 文件类型和权限（续11）

- 文件类型和权限字符串：$C_0C_1C_2C_3C_4C_5C_6C_7C_8C_9$

mode_t st_mode; // unsigned int

| ... | $B_{17}$ | $B_{16}$ | $B_{15}$ | $B_{14}$ | $B_{13}$ | $B_{12}$ | $B_{11}$ | $B_{10}$ | $B_9$ | $B_8$ | $B_7$ | $B_6$ | $B_5$ | $B_4$ | $B_3$ | $B_2$ | $B_1$ | $B_0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | $C_0$ | | | | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ | $C_8$ | $C_9$ |
| | | | | | | | | | | - | - | - | - | - | - | - | - | - |
| | | | | | | | | | | r | w | x | r | w | x | r | w | x |

| 文件类型 | $C_0$ |
|---|---|
| 普通文件 | - |
| 目录 | d |
| 本地套接字 | s |
| 字符设备 | c |
| 块设备 | b |
| 符号链接 | l |
| 有名管道 | p |

$B_{11}$ → $C_3$: S / s

$B_{10}$ → $C_6$: S / s

$B_9$ → $C_9$: T / t

# 文件类型和权限（续12）

Tarena Technology 达内科技

知识讲解

- 例如：将整数形式的文件类型和权限转换为字符串的函数

```
const char* mtos (mode_t m) {
    static char s[11];
    if   (S_ISDIR (m)) strcpy (s, "d");
    else if (S_ISSOCK (m)) strcpy (s, "s");
    else if (S_ISCHR (m)) strcpy (s, "c");
    else if (S_ISBLK (m)) strcpy (s, "b");
    else if (S_ISLNK (m)) strcpy (s, "l");
    else if (S_ISFIFO (m)) strcpy (s, "p");
    else   strcpy (s, "-");
    strcat (s, m & S_IRUSR ? "r" : "-"); strcat (s, m & S_IWUSR ? "w" : "-");
    strcat (s, m & S_IXUSR ? "x" : "-");
    strcat (s, m & S_IRGRP ? "r" : "-"); strcat (s, m & S_IWGRP ? "w" : "-");
    strcat (s, m & S_IXGRP ? "x" : "-");
    strcat (s, m & S_IROTH ? "r" : "-"); strcat (s, m & S_IWOTH ? "w" : "-");
    strcat (s, m & S_IXOTH ? "x" : "-");
    if (m & S_ISUID) s[3] = (s[3] == 'x' ? 's' : 'S');
    if (m & S_ISGID) s[6] = (s[6] == 'x' ? 's' : 'S');
    if (m & S_ISVTX) s[9] = (s[9] == 'x' ? 't' : 'T');
    return s; }
```

文件元数据

【参见：stat.c】

- 文件元数据

Technology Tarena
达内科技