

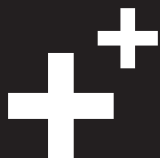
# Unix系统高级编程

File 4

DAY07

# 内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	访问测试与权限掩码
	10:30 ~ 11:20	修改权限与拥有者
	11:30 ~ 12:20	修改大小与内存映射文件
下午	14:00 ~ 14:50	硬链接与符号链接
	15:00 ~ 15:50	目录
	16:00 ~ 16:50	
	17:00 ~ 17:30	总结和答疑



# 访问测试与权限掩码

---



# 访问测试



# 访问测试

- 测试调用进程对指定文件是否拥有足够的访问权限

```
#include <unistd.h>
```

```
int access (const char* pathname, int mode);
```

成功返回0, 失败返回-1

- *pathname*: 文件路径
- *mode*: 访问权限, 可取以下值
  - R\_OK - 可读否
  - W\_OK - 可写否
  - X\_OK - 可执行否
  - F\_OK - 存在否



# 访问测试 (续1)

- 例如

```
– printf ("文件%s", pathname);  
if (access (pathname, F_OK) == -1)  
    printf ("不存在(%m)。 \n");  
else {  
    if (access (pathname, R_OK) == -1)  
        printf ("不可读(%m), ");  
    else printf ("可读, ");  
    if (access (pathname, W_OK) == -1)  
        printf ("不可写(%m), ");  
    else printf ("可写, ");  
    if (access (pathname, X_OK) == -1)  
        printf ("不可执行(%m)。 \n");  
    else printf ("可执行。 \n"); } }
```



## 访问测试 (续2)

- `access`函数在测试调用进程对指定文件的访问权限时，使用的是调用进程的实际用户ID和实际组ID，而非有效用户ID和有效组ID，因此如果调用进程的可执行文件带有设置用户ID位或者设置组ID位，`access`函数返回无权访问并不意味着就真的无权访问



# 访问测试

【参见：[access.c](#)】

- 访问测试





# 权限掩码

---

# 权限掩码

- 设置调用进程的权限掩码

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

```
mode_t umask (mode_t cmask);
```

永远成功，返回原来的权限掩码

– *cmask*: 新权限掩码

- 进程的权限掩码会屏蔽掉该进程创建文件时指定的权限
  - 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： $0666 \& \sim 0022 = 0644$  (rw-r--r--)
- 调用umask函数可以人为改变调用进程的权限掩码，如改为0000，则不屏蔽任何权限



# 权限掩码 (续1)

- 例如
  - 希望屏蔽掉所有用户对该文件的写和执行权限  
`umask (0333);`
  - 或者写成  
`umask (S_IWUSR | S_IXUSR | S_IWGRP | S_IXGRP | S_IWOTH | S_IXOTH);`
  - 其中用于表示权限位的宏定义与stat函数完全一致
- 注意，权限掩码是进程的属性之一，umask函数所影响的仅仅是调用该函数的进程，对其父进程，比如Shell进程，没有任何影响



# 权限掩码

【参见：umask.c】

- 权限掩码



# 修改权限与拥有者

---



# 修改权限



# 修改权限

- 修改指定文件的权限

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

```
int chmod (const char* path, mode_t mode);  
int fchmod (int fd, mode_t mode);
```

成功返回0，失败返回-1

- *path*: 文件路径
- *mode*: 文件权限
- *fd*: 文件描述符
- 调用进程的有效用户ID必须与文件的拥有者用户ID匹配，或者是root用户，才能修改该文件的权限，且受权限掩码的影响



# 修改权限 (续1)

- 例如
  - 设置文件的权限为rwSr-sr-T, 即拥有者用户可读可写不可执行, 拥有者组可读不可写可执行, 其它用户可读不可写不可执行, 同时带有设置用户ID位、设置组ID位和粘滞位  
`fchmod (fd, 07654);`
  - 或者写成  
`fchmod (fd, S_IRUSR | S_IWUSR | S_IRGRP | S_IXGRP | S_IROTH | S_ISUID | S_ISGID | S_ISVTX);`
  - 其中用于表示权限位的宏定义与stat函数完全一致





# 修改权限

【参见：chmod.c】

- 修改权限



# 修改拥有者



# 修改拥有者

- 修改指定文件的拥有者用户和(或)拥有者组

```
#include <unistd.h>
```

```
int chown (const char* path, uid_t owner, gid_t group);  
int fchown (int fd, uid_t owner, gid_t group);  
int lchown (const char* path, uid_t owner, gid_t group);
```

成功返回0，失败返回-1

- *path*: 文件路径
  - *owner*: 拥有者用户ID， -1表示不修改
  - *group*: 拥有者组ID， -1表示不修改
  - *fd*: 文件描述符
- lchown与另两个函数的区别仅在于它不跟踪符号链接



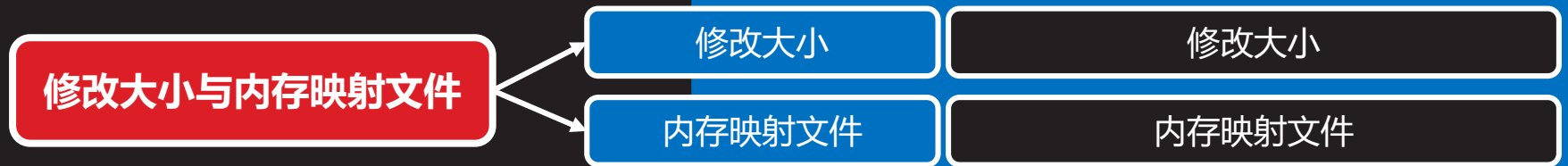
# 修改拥有者 (续1)

- 如果调用进程的有效用户ID为root用户，则它可以任意修改任何文件的拥有者用户和组
- 如果调用进程的有效用户ID为普通用户，则它只能把自己名下文件的拥有者组改为自己隶属的某个组
- 例如
  - `chown ("readme", -1, 1002);`



# 修改大小与内存映射文件

---



# 修改大小



# 修改大小

- 修改指定文件的大小

```
#include <unistd.h>
```

```
int truncate (const char* path, off_t length);  
int ftruncate (int fd, off_t length);
```

成功返回0，失败返回-1

- *path*: 文件路径
  - *length*: 文件大小
  - *fd*: 文件描述符
- 该函数既可以把文件截短，也可以把文件加长，对于后者，新增加的部分全部用数字0填充



# 修改大小

- 例如
  - 学生信息文件中存有大量学生记录(STUDENT), 从中删除第5个学生的记录(假设该记录存在)

```
STUDENT student;
```

```
lseek (fd, 5 * sizeof (student), SEEK_SET);
```

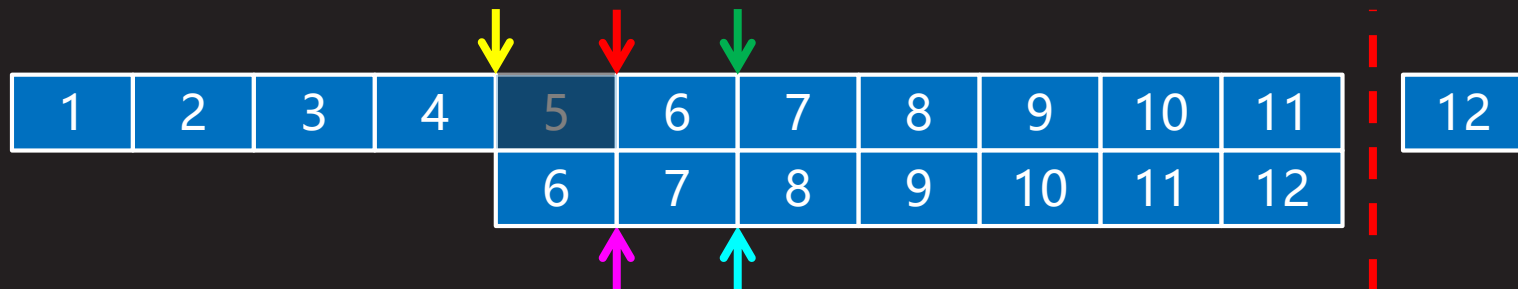
```
while (read (fd, &student, sizeof (student) > 0) {
```

```
    lseek (fd, -2 * sizeof (student), SEEK_CUR);
```

```
    write (fd, &student, sizeof (student));
```

```
    lseek (fd, sizeof (student), SEEK_CUR); }
```

```
ftruncate (fd, lseek (fd, 0, SEEK_CUR) - sizeof (student));
```





# 内存映射文件



# 内存映射文件

- 建立虚拟内存到物理内存或文件的映射

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

```
void* mmap (void* start, size_t length, int prot,  
            int flags, int fd, off_t offset);
```

成功返回映射区内存起始地址，失败返回MAP\_FAILED(-1)

- *start*: 映射区内存起始地址，NULL系统自动选定后返回
- *length*: 映射区字节长度，自动按页(4K)圆整



# 内存映射文件 (续1)

- 创建虚拟内存到物理内存或文件的映射
  - *prot*: 映射区访问权限, 可取以下值
    - PROT\_READ - 映射区可读
    - PROT\_WRITE - 映射区可写
    - PROT\_EXEC - 映射区可执行
    - PROT\_NONE - 映射区不可访问



# 内存映射文件 (续2)

- 创建虚拟内存到物理内存或文件的映射
  - *flags*: 映射标志, 可取以下值
    - `MAP_ANONYMOUS` - 匿名映射, 将虚拟内存映射到物理内存而非文件, 忽略 *fd* 和 *offset* 参数
    - `MAP_PRIVATE` - 对映射区的写操作只反映到缓冲区中, 并不会真正写入文件
    - `MAP_SHARED` - 对映射区的写操作直接反映到文件中
    - `MAP_DENYWRITE` - 拒绝其它对文件的写操作
    - `MAP_FIXED` - 若在 *start* 上无法创建映射, 则失败(无此标志系统会自动调整)
    - `MAP_LOCKED` - 锁定映射区, 保证其不被换出



# 内存映射文件 (续3)

- 创建虚拟内存到物理内存或文件的映射
  - *fd*: 文件描述符
  - *offset*: 文件偏移量, 自动按页(4K)对齐

- 例如

- `ftruncate (fd, 8192);`  
`char* p = (char*)mmap (NULL, 8192,`  
`PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);`  
`if (p == MAP_FAILED) {`  
`perror ("mmap");`  
`exit (EXIT_FAILURE); }`
- `strcpy (p, "Hello, File !"); // 看似写内存, 实则写文件`  
`printf ("%s\n", p); // 假读内存之名, 行读文件之实`



# 内存映射文件 (续4)

- 解除虚拟内存到物理内存或文件的映射

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

```
int munmap (void* start, size_t length);
```

成功返回0, 失败返回-1

- *start*: 映射区内存起始地址, 必须是页的首地址
- *length*: 映射区字节长度, 自动按页(4K)圆整



# 内存映射文件 (续5)

- 例如

```
– if (munmap (p, 4096) == -1) {  
    perror ("munmap");  
    exit (EXIT_FAILURE);  
}  
strcpy (p += 4096, "Hello, File !");  
printf ("%s\n", p);  
if (munmap (p, 4096) == -1) {  
    perror ("munmap");  
    exit (EXIT_FAILURE);  
}
```

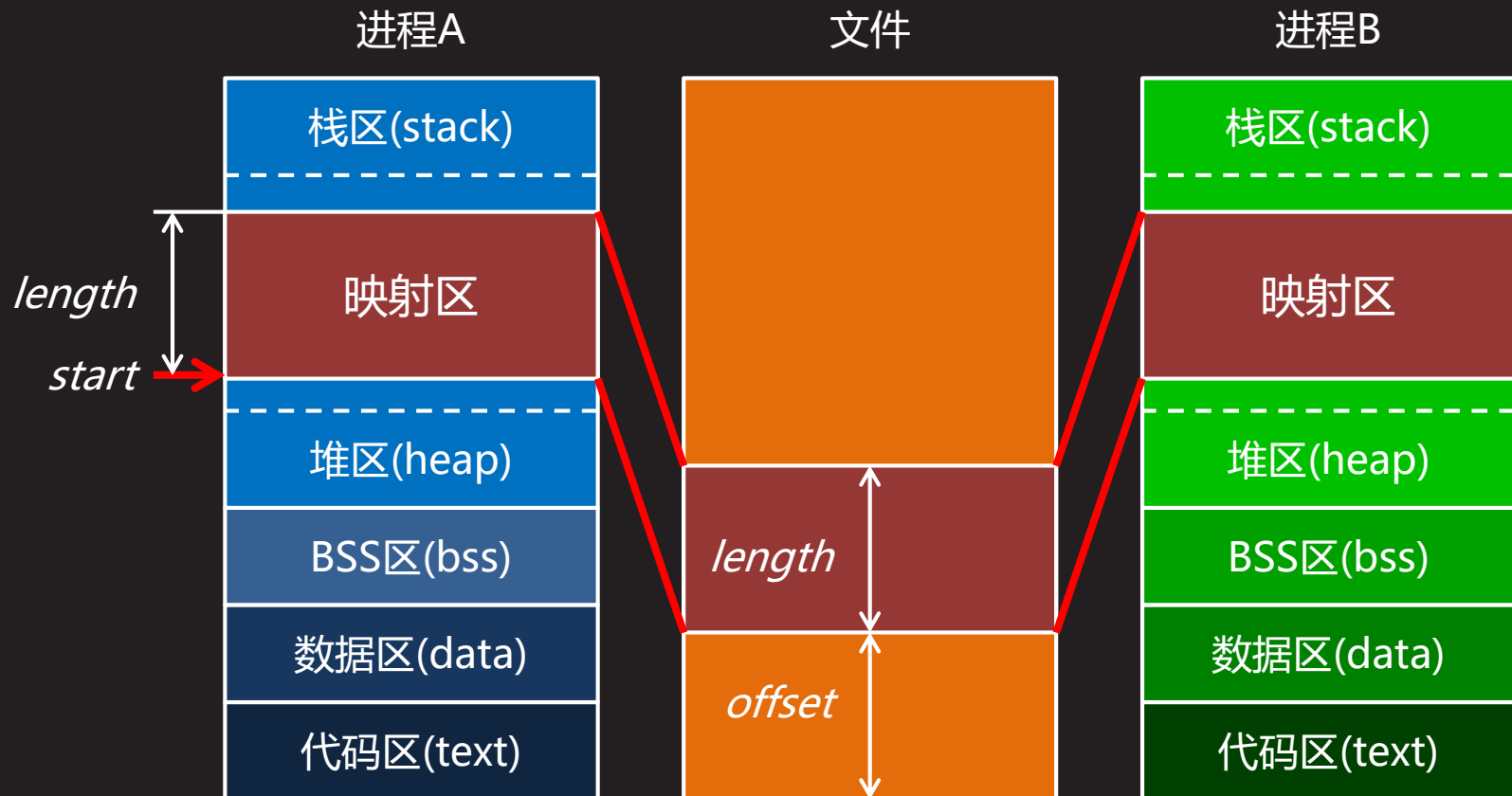
- munmap允许对映射区的一部分解映射，但必须按页



# 内存映射文件 (续6)

- 内存映射文件不仅提供了一种以访问内存的方式读写文件的方法，而且还在多个进程之间打通了一条基于文件共享的数据通道

知识讲解





# 内存映射文件

【参见：trunc.c、mmap.c】

- 内存映射文件



# 硬链接与符号链接

## 硬链接与符号链接

### 硬链接

硬链接的本质

创建硬链接

删除硬链接

修改硬链接

### 符号链接

符号链接的本质

创建符号链接

读取符号链接

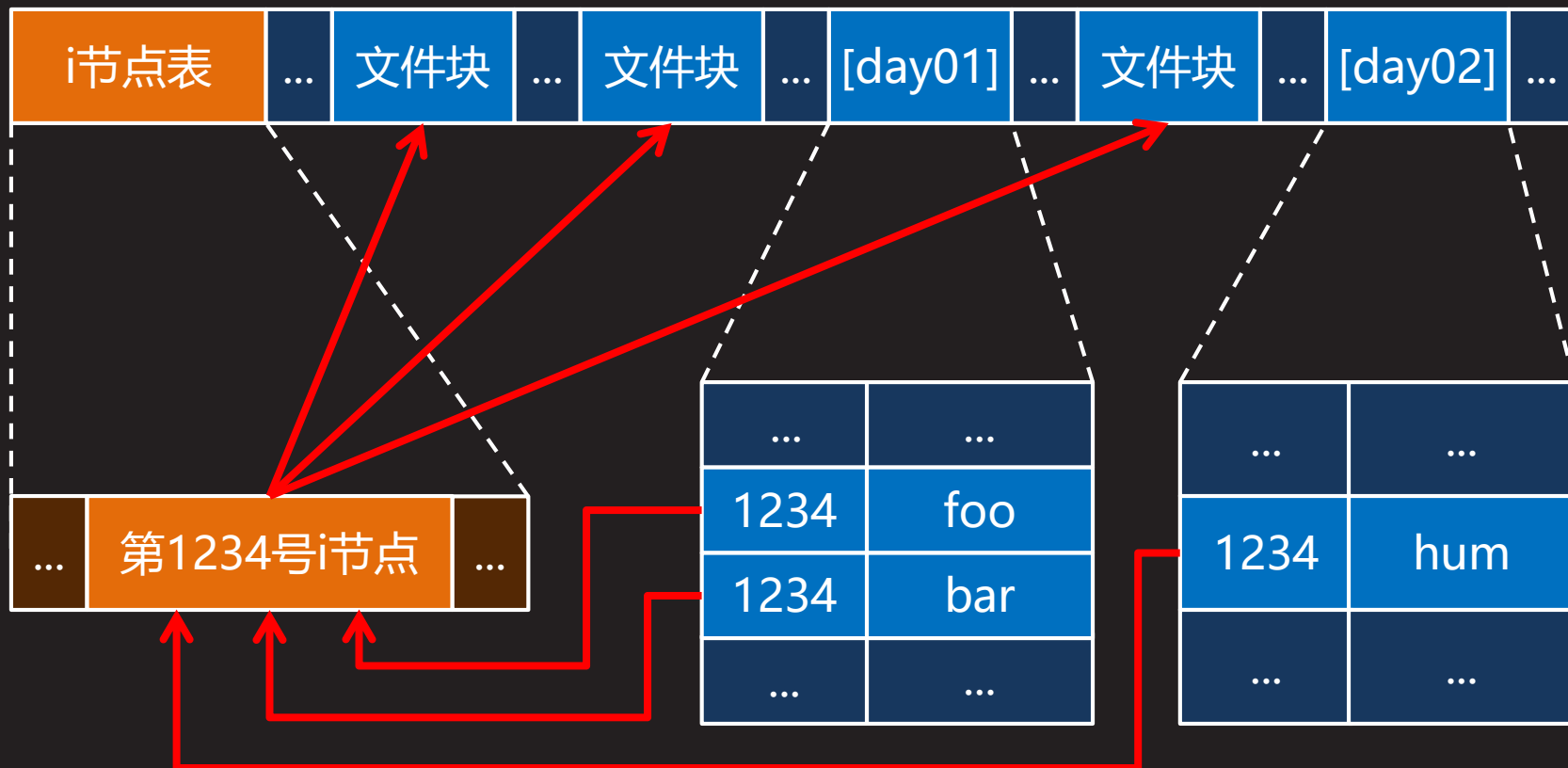
# 硬链接



# 硬链接的本质

- 硬链接的本质就是目录文件里一个文件名和i节点号的对应条目。通过该条目，就可以根据一个文件的文件名迅速地找到与之相对应的i节点号，进而访问该文件的数据

知识讲解



# 创建硬链接

- 根据一个已有的硬链接创建一个新的硬链接

```
#include <unistd.h>
```

```
int link (const char* oldpath, const char* newpath);
```

成功返回0，失败返回-1

- *oldpath*: 原有路径
- *newpath*: 新建路径
- link函数创建一个与*oldpath*引用相同目标的硬链接*newpath*，若后者已存在则什么也不做
- *oldpath*必须存在，且不能是目录
- *newpath*中不能包含不存在的目录名



# 创建硬链接 (续1)

- 例如
  - 假设/home/tarena/unixc/day01目录下有foo文件
  - `link ("foo", "bar");`
  - `link ("bar", "../day02/hum");`
  - 在day01目录下执行进程
  - 第一个link调用, 在day01的目录文件里增加了一个条目, i节点号取自foo文件的i节点号, 文件名为bar
  - 第二个link调用, 在day02的目录文件里增加了一个条目, i节点号取自bar文件的i节点号(也就是foo文件的i节点号), 文件名为hum
  - /home/tarena/unixc/day01/foo
  - /home/tarena/unixc/day01/bar
  - /home/tarena/unixc/day02/hum

其实都是一个文件



# 删除硬链接

- 删除目录文件里的一个硬链接条目

```
#include <unistd.h>
```

```
int unlink (const char* pathname);
```

成功返回0，失败返回-1

– *pathname*: 文件路径

- *pathname*只能是文件的路径，不能是目录的路径，即unlink函数不能删除引用目录的硬链接
- Unix/Linux系统没有直接删除文件的系统调用，所谓删除文件其实就是删除硬链接，硬链接删干净了，文件自然就被删除了



# 删除硬链接（续1）

- 一个文件可以同时拥有多个硬链接，通过unlink函数删除其中的一个硬链接并不会导致该文件被删除(释放磁盘上的i节点和数据块)，因为必须保证其它引用该文件的硬链接继续有效，但该文件的硬链接数(作为文件的元数据保存在其i节点中)会被减1
- 如果删除的是该文件的最后一个硬链接，其硬链接数将被减到0，这表示系统中已经没有任何硬链接引用该文件，直到此刻，该文件才会真正被删除
- 即便删除的是该文件的最后一个硬链接，但如果此时该文件正被某些进程打开，其磁盘上的数据也不会立即被删除，直到所有打开它的进程都显式或隐式地关闭了该文件，其在磁盘上的数据才会被删除





# 删除硬链接（续2）

- 删除目录文件里的一个硬链接条目

```
#include <stdio.h>
```

```
int remove (const char* pathname);
```

成功返回0，失败返回-1

- *pathname*: 文件或目录路径
- remove函数与unlink函数的功能几乎完全一样，唯一的区别是remove函数不但可以删除文件硬链接，也可以删除目录硬链接，唯一的条件是被删除的目录必须为空



# 修改硬链接

- 改变一个文件的名称或位置

```
#include <stdio.h>
```

```
int rename(const char* oldpath, const char* newpath);
```

成功返回0，失败返回-1

- *oldpath*: 源路径
- *newpath*: 目标路径
- 若 *newpath* 已存在，则被改为引用 *oldpath* 的目标
- 若 *oldpath* 和 *newpath* 本来引用的就是同一个目标，则该函数什么也不做且返回成功
- 若 *oldpath* 是目录，则 *newpath* 必须不存在或为空目录



# 硬链接

【参见：link.c、unlink.c、remove.c、rename.c】

- 硬链接



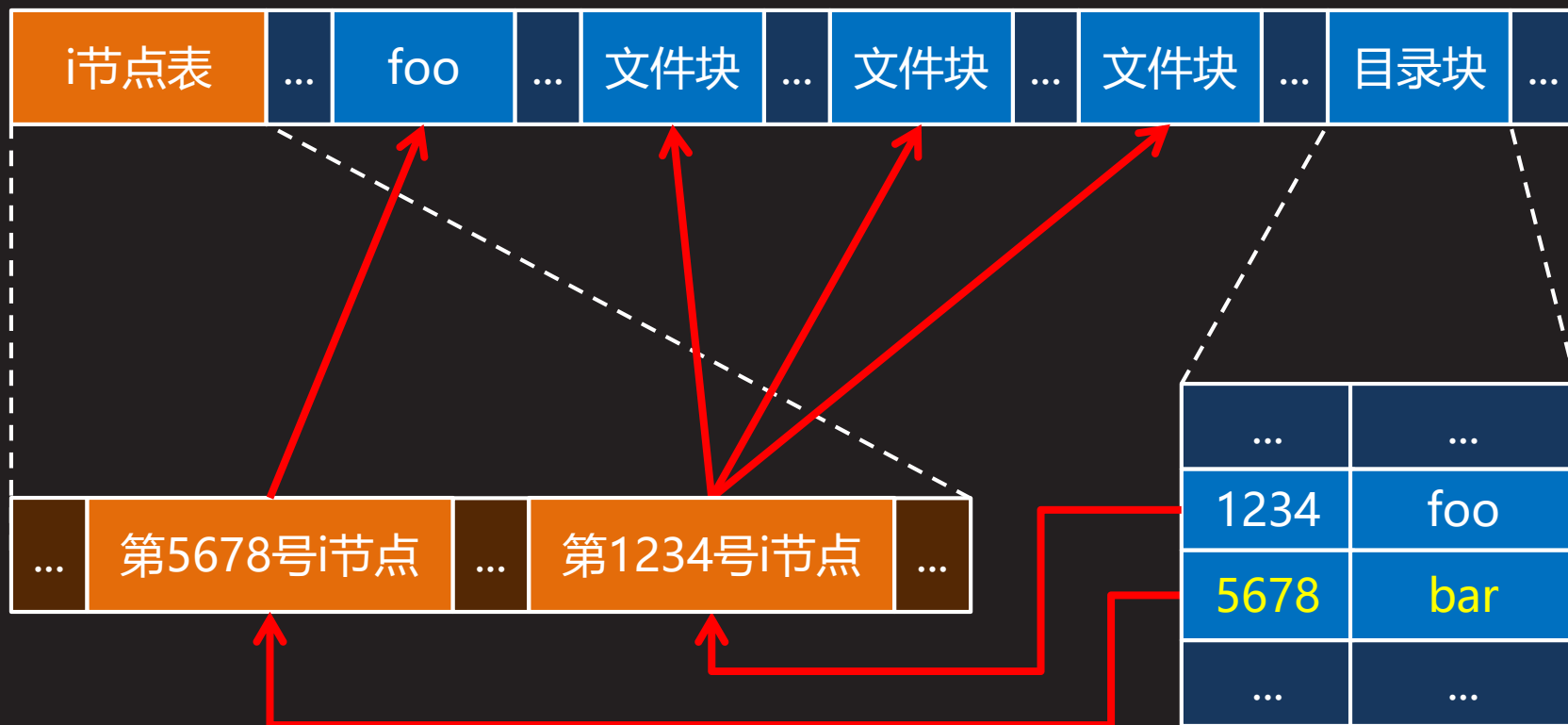
# 符号链接

---

# 符号链接的本质

- 符号链接的本质就是一个保存着另一个文件或目录的路径的文件。所有针对符号链接文件的访问，最后都会被定向到该符号链接所引用的目标文件或目录

知识讲解



# 创建符号链接

- 根据一个已有的硬链接创建一个符号链接

```
#include <unistd.h>
```

```
int symlink (const char* oldpath, const char* newpath);
```

成功返回0，失败返回-1

- *oldpath*: 原有路径
- *newpath*: 新建路径
- `symlink`函数创建一个保存*oldpath*的符号链接*newpath*，若后者已存在则返回失败
- *oldpath*可以是文件也可以是目录，甚至可以不存在
- *newpath*中不能包含不存在的目录名



# 读取符号链接

- 读取符号链接文件的内容

```
#include <unistd.h>
```

```
ssize_t readlink (const char* path, char* buf,  
                 size_t size);
```

成功返回拷入*buf*的符号链接文件内容的字节数，失败返回-1

- *path*: 符号链接文件路径
- *buf*: 缓冲区
- *size*: 缓冲区大小(字节)
- 符号链接文件本身不能用open和read函数打开并读取
- readlink函数不负责追加结尾空字符



# 读取符号链接 (续1)

- 例如
  - ```
char buf[PATH_MAX+1] = {0};  
if (readlink (path, buf, sizeof (buf) -  
    sizeof (buf[0])) == -1) {  
    perror ("readlink");  
    exit (EXIT_FAILURE);  
}  
printf ("%s -> %s\n", path, buf);
```





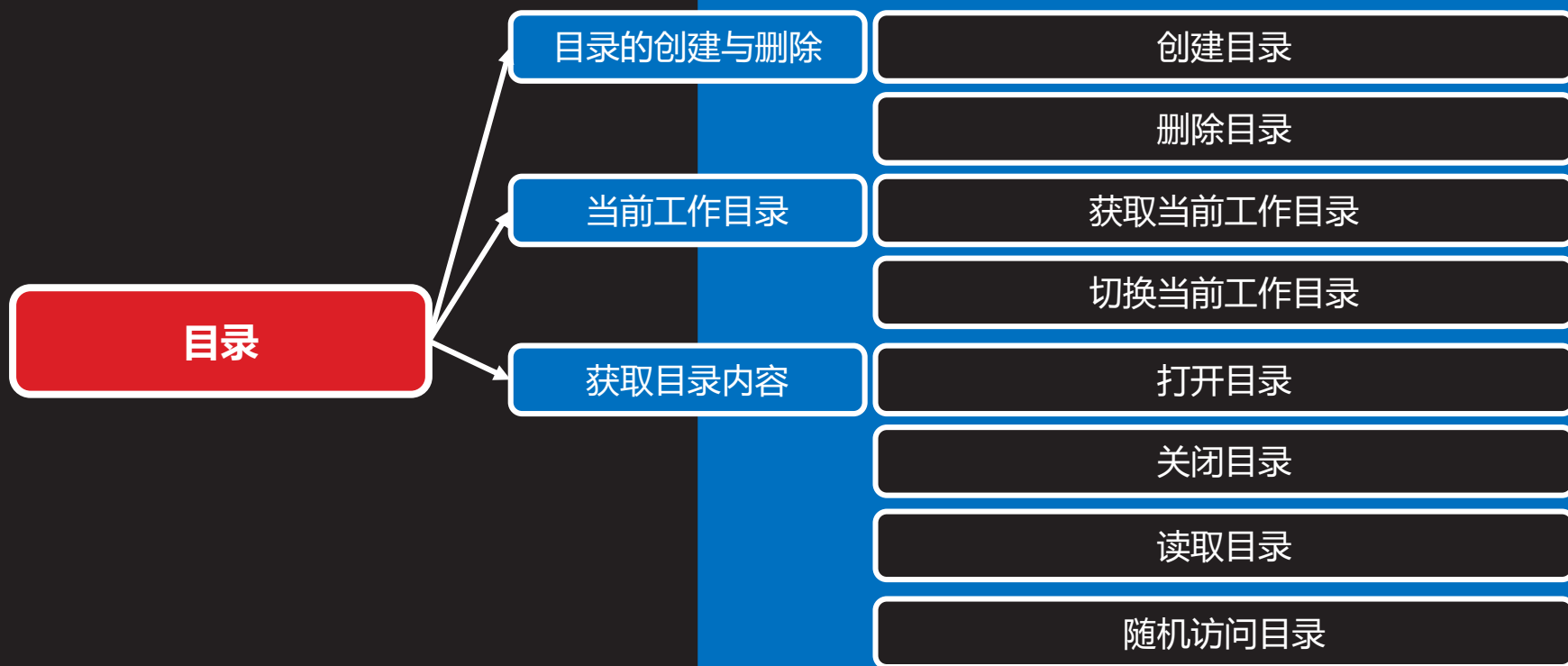
# 符号链接

【参见：slink.c】

- 符号链接



# 目录



# 目录的创建与删除



# 创建目录

- 创建一个空目录

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

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

成功返回0，失败返回-1

- *pathname*: 目录路径
- *mode*: 访问权限，目录的执行权限(x)表示可进入



# 删除目录

- 删除一个空目录

```
#include <unistd.h>
```

```
int rmdir (const char* pathname);
```

成功返回0，失败返回-1

– *pathname*: 目录路径

- *pathname*不能是文件只能是目录，且必须为空
- remove函数可以看成是unlink函数和rmdir函数的合并



# 目录的创建与删除

【参见：dir.c】

课堂  
练习

- 目录的创建与删除



# 当前工作目录

---

# 获取当前工作目录

- 获取当前工作目录的路径

```
#include <unistd.h>
```

```
char* getcwd (char* buf, size_t size);
```

成功返回当前工作目录路径字符串指针，失败返回NULL

- *buf*: 缓冲区
- *size*: 缓冲区大小(字节)
- getcwd函数在将当前工作目录的路径字符串拷入缓冲区 *buf* 时，会自动追加结尾空字符
- 当前工作目录是每个进程的属性之一，保存在进程表中，它是系统内核为进程解析相对路径的起点





# 切换当前工作目录

- 将当前工作目录切换到指定路径

```
#include <unistd.h>
```

```
int chdir (const char* path);  
int fchdir (int fd);
```

成功返回0，失败返回-1

- *path*: 目标工作目录路径
- *fd*: 目标工作目录文件描述符(由open函数返回)
- 该函数只切换调用进程的当前工作目录，对其父进程，如Shell进程的当前工作目录，不构成任何影响



# 当前工作目录

【参见：dir.c】

- 当前工作目录



# 获取目录内容

---

# 打开目录

- 打开指定的目录

```
#include <dirent.h>
```

```
DIR* opendir (const char* name);
```

```
DIR* fdopendir (int fd);
```

成功返回目录流指针，失败返回NULL

- name: 目录路径
- fd: 目录文件描述符(由open函数返回)
- 目录流类似于标准库中的文件流(通过FILE\*引用)，所有后续针对所打开目录的函数调用，皆需传入此参数



# 关闭目录

- 关闭指定的目录

```
#include <dirent.h>

int closedir (DIR* dirp);
```

成功返回0，失败返回-1

- *dirp*: 目录流指针

- 例如

```
– DIR* dirp = opendir ("/usr/bin");
  if (! dirp) {
    perror ("opendir"); exit (EXIT_FAILURE); }
...
  closedir (dirp);
```



# 读取目录

- 读取目录中的一个条目

```
#include <dirent.h>
```

```
struct dirent* readdir (DIR* dirp);
```

成功返回目录条目指针，读完(不置errno)或失败返回NULL

– *dirp*: 目录流指针

- 目录由若干条目组成，每个条目均包含文件名、文件类型、i节点号等信息。readdir函数可以连续调用，调用一次返回一个条目，再调用一次返回下一个条目，直到返回NULL表示已读完整个目录或者发生错误，为了区分这两种情况，可以通过检查errno是否被重置来进行判断



# 读取目录 (续1)

- 目录条目结构

```
- struct dirent {  
    ino_t          d_ino;      // i节点号  
    off_t         d_off;      // 下一条目位置  
    unsigned short d_reclen;  // 记录长度  
    unsigned char  d_type;    // 文件类型  
    char          d_name[];   // 文件名  
};
```



# 读取目录 (续2)

- 目录条目结构
  - 其中, 表示文件类型的d\_type成员, 可取以下值

|            |         |
|------------|---------|
| DT_REG     | - 普通文件  |
| DT_DIR     | - 目录    |
| DT_SOCK    | - 本地套接字 |
| DT_CHR     | - 字符设备  |
| DT_BLK     | - 块设备   |
| DT_LNK     | - 符号链接  |
| DT_FIFO    | - 有名管道  |
| DT_UNKNOWN | - 未知    |

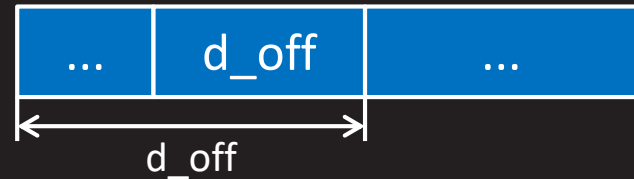
这与该记录在文件i节点中的文件类型是一致的





# 读取目录 (续3)

- 目录中还有两个特殊的条目，其d\_name成员的值“.”和“..”分别表示该目录本身和其父目录。根目录没有父目录，故其“..”和“.”一样都表示根目录本身
- 目录中的各个条目在目录流中连续排列，但其d\_name成员的长度却并不相同，因此需要通过d\_off成员指示下一个条目在目录流中的偏移位置



- 现代文件系统中的目录条目，多采用树或散列表作为其在内存中的存储结构。每个条目中的d\_off成员可被视为于该结构中检索下一个条目的键



# 读取目录 (续4)

- 例如

```
– for (;;) {  
    errno = 0;  
    struct dirent* direntp = readdir (dirp);  
    if (! direntp) {  
        if (! errno)  
            break;  
        perror ("readdir");  
        exit (EXIT_FAILURE);  
    }  
    printf ("%s\n", direntp->d_name);  
}
```



# 随机访问目录

- 设置目录流读取位置

```
#include <dirent.h>
```

```
void seekdir (DIR* dirp, long offset);
```

无返回值

- *dirp*: 目录流指针
  - *offset*: 目录流读取位置
- 默认情况下readdir函数是一条挨一条地读取目录流中的目录条目的，每读取一个目录条目即用相应dirent结构中的d\_off成员设置下一次读取的位置。通过seekdir函数，可以人为设置对目录流的读取位置，实现随机访问



# 随机访问目录 (续1)

- 获取目录流读取位置

```
#include <dirent.h>
```

```
long telldir (DIR* dirp);
```

成功返回目录流当前读取位置，失败返回-1

– *dirp*: 目录流指针

- 刚刚通过opendir函数打开目录流，telldir函数返回0
- 刚刚通过readdir函数读取目录流，telldir函数返回最近一次读到的dirent结构中d\_off成员的值
- 刚刚通过seekdir函数设置目录流读取位置，telldir函数返回传递给seekdir函数的*offset*参数的值



# 随机访问目录 (续2)

- 复位目录流读取位置

```
#include <dirent.h>

void rewinddir (DIR* dirp);
```

无返回值

– *dirp*: 目录流指针

- 任何时候调用rewinddir函数，都可以立即将目录流读取位置恢复到目录流的最开始处(通常是0)，即读取第一个目录条目的位置



# 获取目录内容

【参见：list.c、seek.c】

- 获取目录内容



# 总结和答疑

