第8章 文件系统

- 本章内容
- 1. 文件系统的基本概念与管理功能
- 2. 文件系统的接口
- 3. 文件系统及目录结构的实现
- 4. 文件共享与保护
- 5. 虚拟文件系统

文件系统的核心挑战与解决路径

- 核心挑战：如何让用户方便地在持久化存储设备上管理数据
  - 如何表示和描述文件？（文件系统的抽象与元数据）
  - 如何在磁盘上布局文件数据？（文件的组织方式）
  - 如何组织和查找文件？（文件目录与路径）
  - 如何安全高效地访问文件？（文件操作接口）
  - 如何设计实际文件系统？（典型的文件系统）
  - 如何提升文件的访问速度？（文件系统的性能）
  - 如何统一多种文件系统？（虚拟文件系统）

文件系统的动机与基本需求

- 如果系统只有“读写磁盘扇区的驱动”，没有文件系统，程序员要把数据“存下去并在下次开机找回来”，他要自己解决哪些麻烦？
- 子问题：
  - 怎样确保数据不被覆盖？
  - 多个程序同时存数据会不会乱？
  - 下次开机，怎么知道数据在哪？

从扇区号到元数据

- 如果程序只记住“扇区 100”，会需要哪些关键信息，才能在下次正确读取这段数据？
- 子问题：
  - 数据有多长？怎么知道读到哪里就停？
  - 如果数据被更新过，扇区号会不会变？
  - 文件的所有者、权限、创建时间呢？

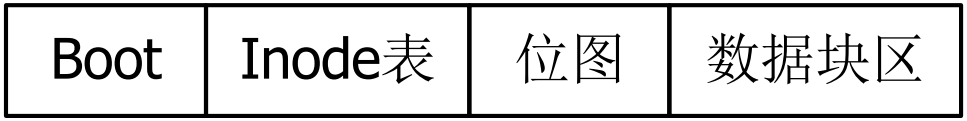
用户数据
Hello
World
…
（内容）

元数据
大小：1000
位置：扇区100-102
所有者：u1
权限：rw-r--
时间：…

inode：元数据的标准化结构

- 当磁盘上有成千上万个数据对象时，我们如何高效地组织和快速查找所有对象的身份信息？
  - 做一个大表格然后逐行扫描？会不会太慢？
  - 能不能用数学方法一步到位找到某个对象的信息？
  - 为什么要把元数据固定成 256 字节？

磁盘布局




```
struct inode {
    uint type;                  // 文件类型
    uint nlink;                 // 硬链接计数
    uint size;                  // 文件大小（字节）
    uint atime, mtime, ctime;   // 时间戳
    uint uid, gid;              // 所有者和组
    uint mode;                  // 权限位
    uint addrs[15];             // 数据块指针！（关键）
};
// 总共 256 字节
```

文件的逻辑视图与物理视图的分离

- 用户看到的是一串连续的字节序列（逻辑），而磁盘看到的是离散的块（物理）。这两者之间的翻译工作应该由谁完成，怎么完成？
  - 一个 10MB 文件，用户看到它是连续的，磁盘怎样装它？
  - 块在磁盘上可能不连续，系统怎么追踪？
  - inode 中的指针数组扮演什么角色？

用户视角（逻辑）

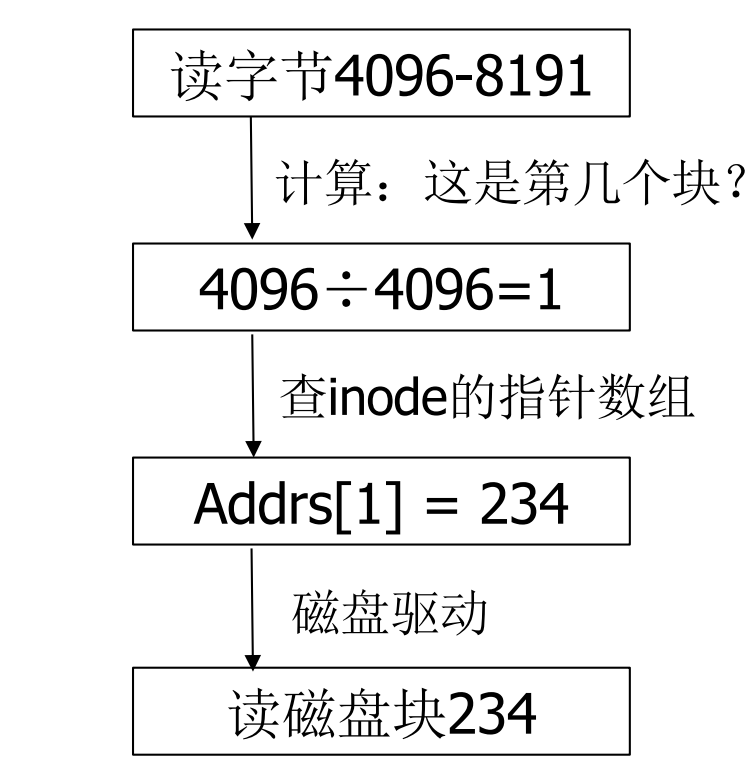
```
字节 0 → 字节 1 → ... → 字节 10485759   (连续序列)
```

系统视角（物理）

```
块号 0   → 磁盘块号 500
块号 1   → 磁盘块号 234   ← 不连续!
块号 2   → 磁盘块号 1000
块号 3   → 磁盘块号 667
...
```

从逻辑视图到物理视图

- 应用程序说'读文件字节 4096-8191'。系统怎么翻译？




```
struct inode {
    ...
    uint addrs[15];  // 指向数据块的指针
};

addrs[0] = 500;   // 块 0 在磁盘块 500
addrs[1] = 234;   // 块 1 在磁盘块 234
addrs[2] = 1000;  // 块 2 在磁盘块 1000
......
```

从应用程序到磁盘的三层视图

应用程序视角（逻辑）

```
字节 0-4095
字节 4096-8191   ← 应用要读这个
字节 8192-12287
......
```

⇩ 翻译（inode指针数组）

文件系统视角（块）

```
块 0 → addrs[0]
块 1 → addrs[1]   ← 对应磁盘块 234
块 2 → addrs[2]
......
```

⇩ 翻译（块号 → 扇区号）

磁盘视角（物理）

```
块 234
```

多级索引指针：支持任意大小文件

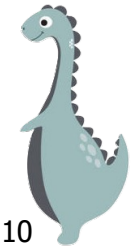
- 如果 inode 的 addrs 数组大小固定，怎样既支持几字节的小文件，又支持几 GB 的大文件，而不浪费 inode 空间？
  - inode 固定 256 字节，指针数组能放多少个？
  - 48 个指针最多支持多大的文件？
  - 怎样用间接指针突破这个容量限制？

在后面介绍文件的组织与
分配时将讨论这个问题

bmap：从逻辑块号到磁盘块号

- 对于高层的 read/write 系统调用来说，如果要读写“ inode 的第 n 个数据块”，底层如何根据 n 自动找到真正的磁盘块号？
  - 如何判断逻辑块号 bn 落在“直接/间接”中的哪一类？
  - 如果块还没被分配，谁负责新分配？

- 阅读xv6中的bmap函数

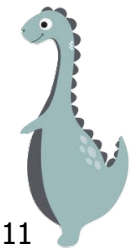


目录与文件名到 inode 号的映射

- 用户说“打开 report.txt”，系统内部需要的是 inode 号，这个“名字到 inode 号”的翻译工作由谁完成？
  - 为什么要用“目录文件”保存映射表？
  - 一个目录项需要包含哪些最小信息？

目录项结构

```
struct dirent {
    uint16 inum;      // inode 号
    char name[14];    // 文件名（最长 14 字符）
};

// 一个目录文件的数据块内容看起来像：
// {inum: 5, name: "report.txt"}
// {inum: 6, name: "data.xlsx"}
// {inum: 7, name: "photo.jpg"}
```



分层目录结构

- 当所有文件都堆在同一个目录里时，查找变得很慢，怎样组织文件能提升查找效率？
  - 建立目录树
  - 目录树如何减小单次查找范围？
  - "/a/b/c/d" 这样的路径，在内部怎样一步步解析？

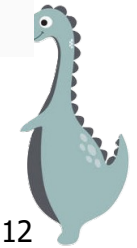
目录树

```
/
├── home/
│   ├── user1/
│   │   ├── documents/
│   │   │   ├── report.txt
│   │   │   └── data.xlsx
│   │   └── photos/
│   └── user2/
├── etc/
└── var/
```

目录也是文件
也有inode

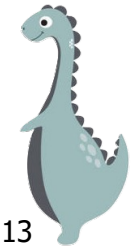
⇨

```
/ (inode 2)
├── home/ (inode 5)
│   ├── user1/ (inode 23)
│   │   ├── report.txt (inode 67)
│   │   └── data.xlsx (inode 68)
│   └── user2/ (inode 24)
├── etc/ (inode 6)
└── var/ (inode 7)
```



路径解析：从路径字符串到目标 inode

- 当程序调用 open("/a/b/c/d") 时，系统内部如何一步步从根目录走到目标文件？
  - 路径是如何被分段的？
  - 每一段名字对应的目录查找过程怎样？

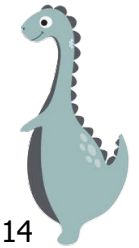


xv6中的namei的工作流程

- 功能：给一个路径字符串(比如'/home/user1/report.txt')，返回目标 inode
- 工作过程：
  - 1. 分割路径'/home/user1/report.txt' → ['home', 'user1', 'report.txt']
  - 2.从起点 inode 开始（根 inode 2 或当前目录） 当前 inode = inode 2
  - 3.对每一段名字

    for each name in ['home', 'user1', 'report.txt']:

    确认当前 inode 是目录
    在当前目录的数据块中查找 name
    找到对应的 inum
    读取该 inode
    当前 inode = 新的 inode
  - 4.返回最终 inode



硬链接：共享 inode

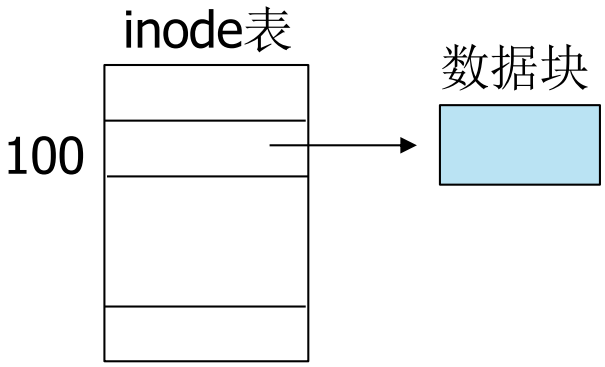
- 如果两个不同的目录项都指向同一个 inode，删除其中一个时，系统如何避免把文件内容错误删除？
  - inode 如何知道'有多少个名字指向我'？
  - 硬链接不能链接目录（防止目录树出现循环）
  - 硬链接不能跨越文件系统（inode 号是相对的）

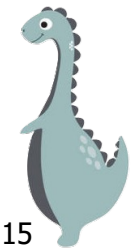
目录A

文件名	inode
original.txt	100

目录B

文件名	inode
link.txt	100





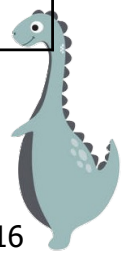
符号链接：通过路径重定向资源

- 当我们需要跨文件系统链接、链接目录，或链接一个尚未创建的目标时，该怎么办？
  - 采用符号链接——一个特殊的文件
  - 符号链接 inode 里存的是什么？
  - 跟随符号链接时，路径解析算法要在哪一步做'重定向'？

```
符号链接文件 link.txt
├─ inode 号：101（新的 inode）
├─ type：T_SYMLINK（特殊标记）
└─ 数据块内容：'/home/user/target.txt'
```

目录项

文件名	inode
link.txt	101

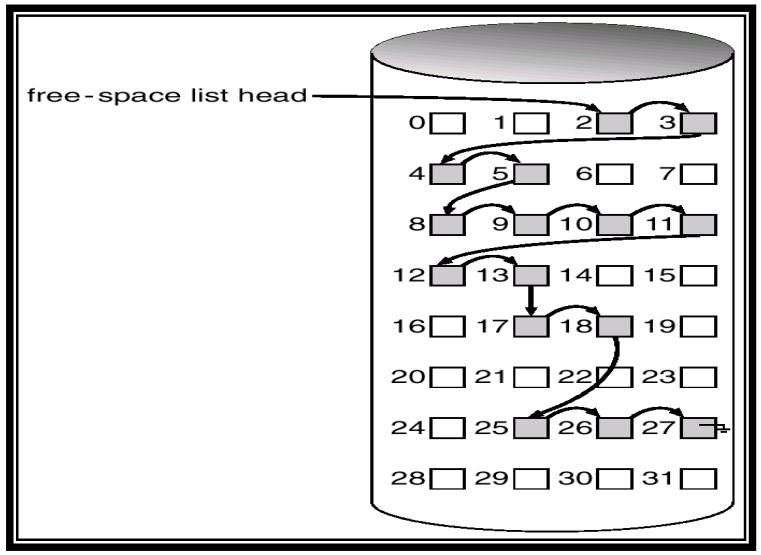


磁盘空闲空间管理

- 当成百上千个程序不断创建和删除文件时，文件系统如何知道哪些块已被占用，哪些仍然可用？
  - 位示图方案

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	0	0	1	1	0	1	1	1	0	1	1	1	1	1
1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	1
2	1	1	1	1	1	1	0	1	1	1	1	0	0	0	0	0
3																
4	…															
⋮																

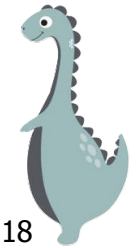
  - 空闲链表方案



- xv6 和 Linux 都用位图

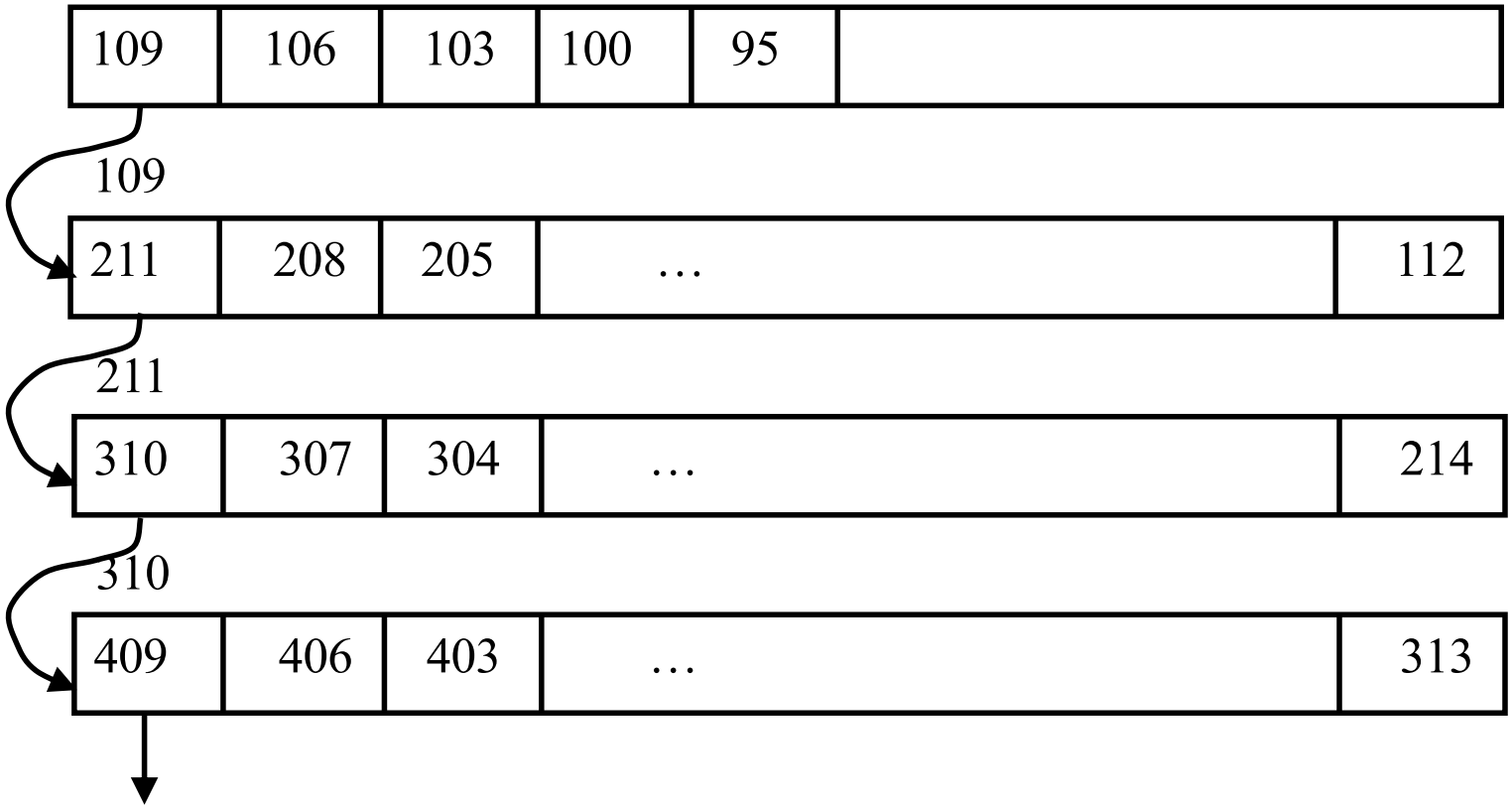
空闲链的改进——成组链接法

- UNIX系统采用成组链接法对空闲盘块加以组织。
- 空闲盘块的组织：将若干个空闲盘块划归一组，将每组中的所有盘块号存放在其前一组的第一个空闲盘块号指示的盘块中，而将第一组中的所有空闲盘块号放入超级块的空闲盘块号表中。



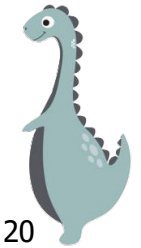
成组链接法示意图

超级块空闲盘块号表



空闲盘块的分配

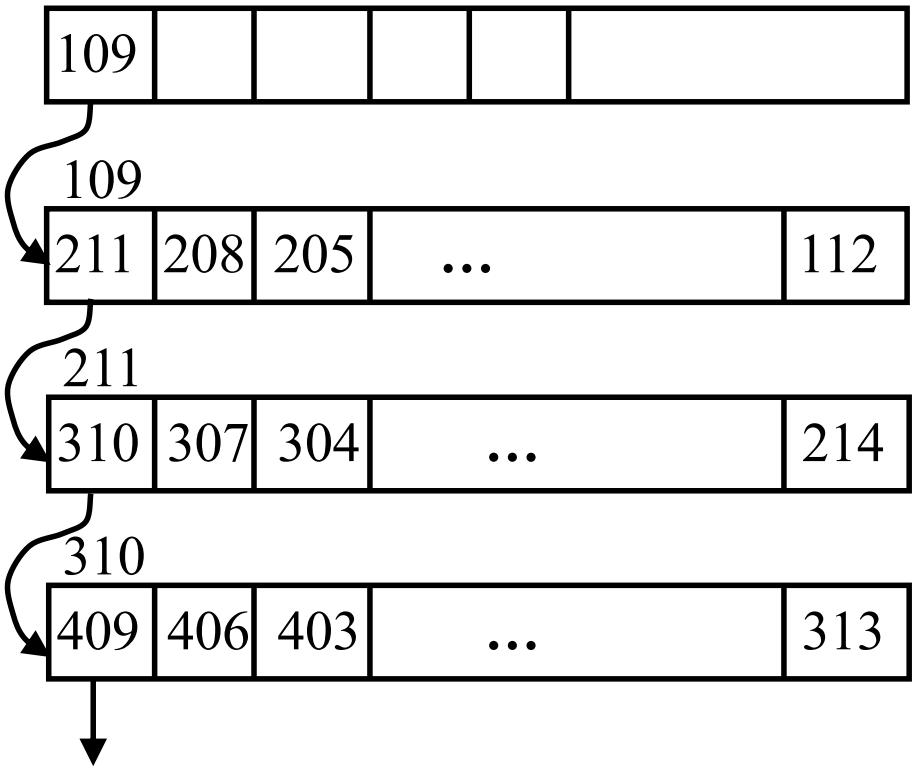
- 当要分配一个盘块时，首先将超级块空闲盘块号表中下一个可用盘块分配出去；
- 如果所分配盘块号是超级块中最后一个可用盘块号，则先将该盘块中的内容读入超级块空闲盘块号表中，然后才将该盘块分配出去。



分配超级块中最后一个盘块号例

- 分配前

超级块空闲盘块号表

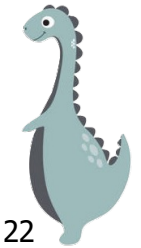


分配后

超级块空闲盘块号表

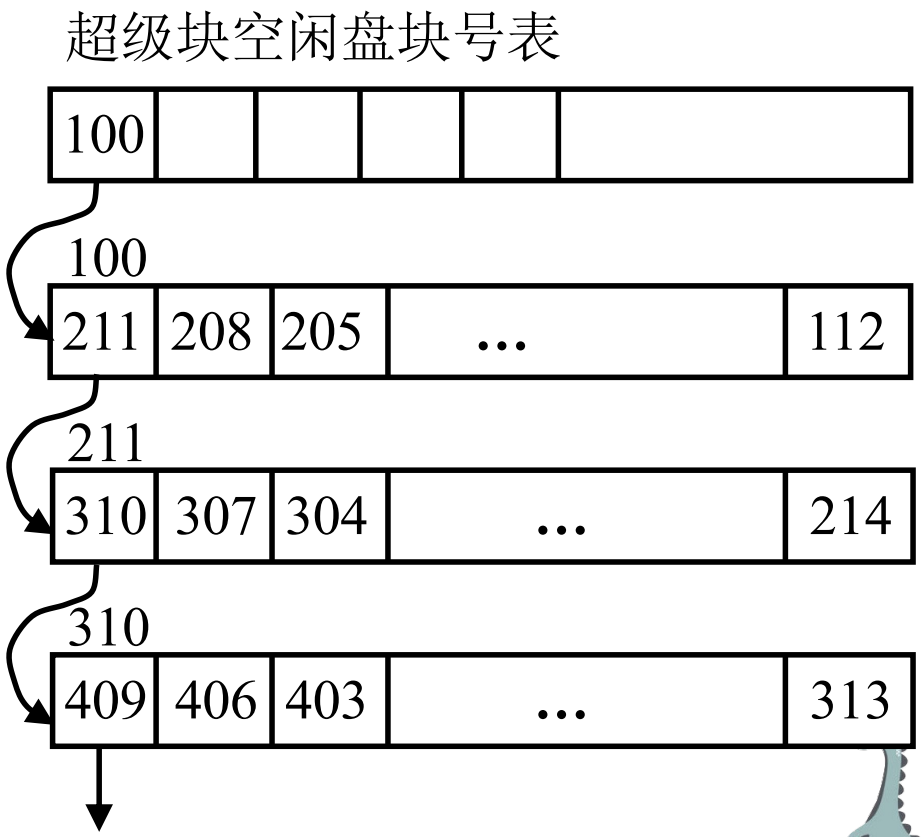
空闲盘块的回收

- 在回收空闲盘块时，如果超级块中的空闲盘块号表未满，可直接将回收盘块的编号放入空闲盘块号表中；
- 若空闲盘块号表已满，需先将空闲盘块号表中的所有盘块号复制到新回收的盘块中，再将新回收盘块的编号放到超级块空闲盘块号表中，此块号就成了表中惟一的盘块号。



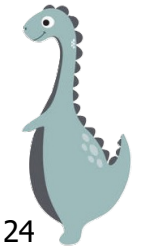
超级块已满时回收盘块例

- 回收前（回收100号盘块）　回收后



文件操作的系统接口

- 应用程序调用 open(filename) 时，系统内部怎样建立起“应用程序”与“磁盘上的数据”之间的连接？
  - open() 返回什么？为什么不直接返回 inode？
  - 文件描述符（fd）的作用是什么？
  - 每次 open() 都重新解析路径吗？

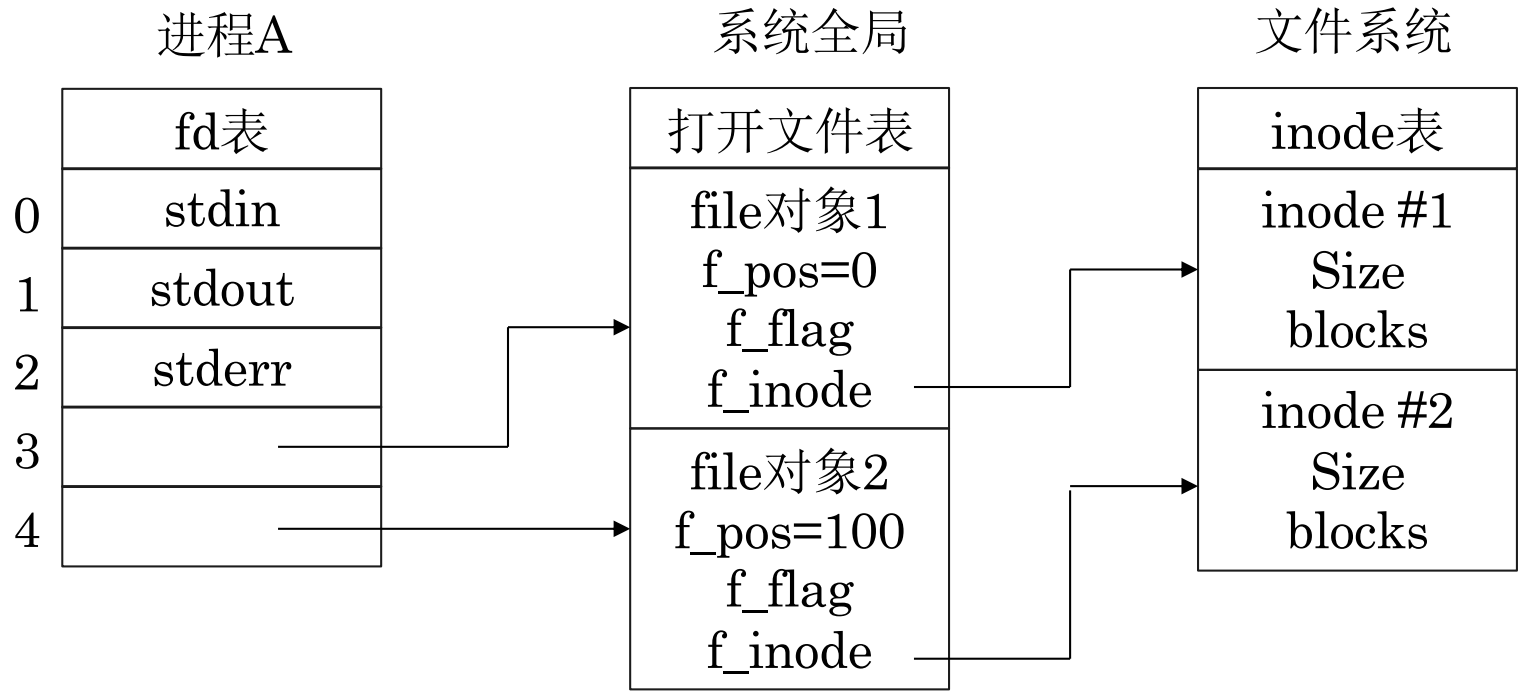


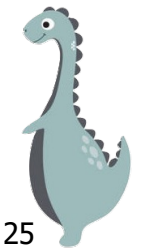
open() 打开文件的流程

- 文件描述符（fd）的作用
  - fd是进程访问文件的"句柄"
  - 用户空间只需记住一个整数（如3、4、5）
  - 内核维护fd到文件的映射关系
- 系统调用

```
int fd = open("/home/user/file.txt", O_RDONLY);
```

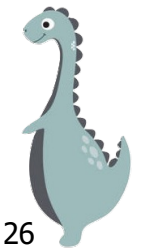
- 内核三层表结构





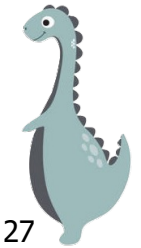
打开文件

- **Step 1：路径解析**
  - 系统调用 namei("report.txt")
  - 找到目标 inode 号（比如 67）
  - 读入内存（或从缓存返回）
- **Step 2：权限检查**
  - 检查当前进程的用户/组 ID
  - 检查 inode 中的权限位
  - 确认进程有权打开此文件
- **Step 3：创建打开文件结构**
  - 在内核中创建一个 'file' 结构体
  - 记录：inode、当前文件偏移、打开模式（O_RDONLY/O_WRONLY/O_RDWR ）等
  - 将这个结构体加入当前进程的'打开文件表'
- **Step 4：返回文件描述符**
  - 文件描述符（fd）是一个小整数（0, 1, 2, 3, ...）
  - 它是进程打开文件表中的索引
  - 返回 fd 给应用程序



关闭文件

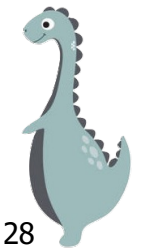
- close(fd); // 做以下事：
- 1. 找到 fd 对应的 file 结构
- 2. 递减 inode 的打开计数（可能有多个进程打开同一文件）
- 3. 如果计数变为 0，释释放 inode 缓存
- 4. 从打开文件表中删除 fd 5
- . 标记 fd 为可用（可被新的 open() 使用）



文件读写操作与文件指针

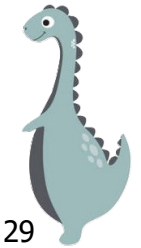
- 当应用程序对同一个打开的文件执行多次 read() 时，系统怎样记住“下次从哪里开始读”？
  - 文件偏移（file offset）保存在哪里？
  - 多个进程打开同一文件，它们的偏移独立吗？
  - seek() 操作怎样工作？

```
// 应用程序代码
fd = open("data.txt");
read(fd, buf1, 100);   // 读前 100 字节
read(fd, buf2, 100);   // 读接下来 100 字节
read(fd, buf3, 100);   // 再读 100 字节
```



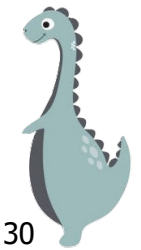
多进程对同一文件的访问

- 当多个进程同时打开和修改同一个文件时，系统怎样防止数据损坏和不一致？
  - 多个进程的读不会冲突吗？
  - 多个进程同时写会发生什么？
  - 文件锁的作用是什么？



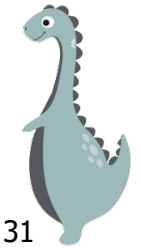
xv6的简单锁机制

```
// inode 中有一个锁
struct inode {
    struct spinlock lock;
    ......
};

// 打开文件时，获取锁
fd = open("data.txt", O_WRONLY);
  → ilock(ip);  // 获取 inode 锁
  → ... 访问 inode ...
  → iunlock(ip);  // 释放锁

// Linux 的更复杂方案：
// 支持 fcntl() 和 flock() 系统调用
//   fcntl(fd, F_SETLK, &lock);  // 设置文件锁
//   flock(fd, LOCK_EX);         // 独占锁
```



文件的读写执行权限

- 当多个用户共用一台计算机时，系统怎样防止用户 A 的数据被用户 B 恶意或意外修改？
  - 权限位是如何编码的？
  - open() 怎样检查权限？
  - 为什么还需要文件执行权限？



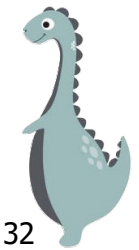
Unix权限模型

- 权限位结构

```
-rw-r--r--  1 user group 1024 Jan 1 10:00 file.txt
 \__/\_/\_/
  |   |   └─ Others权限: 只读
  |   └──── Group权限: 只读
  └──────── User(Owner)权限: 读+写
```

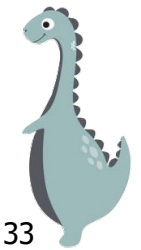
- 权限位含义（对文件）
  - r (read, 4): 读文件内容
  - w (write, 2): 修改文件内容
  - x (execute, 1): 执行文件（程序、脚本）
- 权限位含义（对目录）
  - r (read): 列出目录内容（ls）
  - w (write): 创建/删除目录中的文件
  - x (execute): 进入目录（cd）、访问目录中的文件
- 八进制表示

```
0644 = 110 100 100
       rw- r-- r--
       6   4   4
```

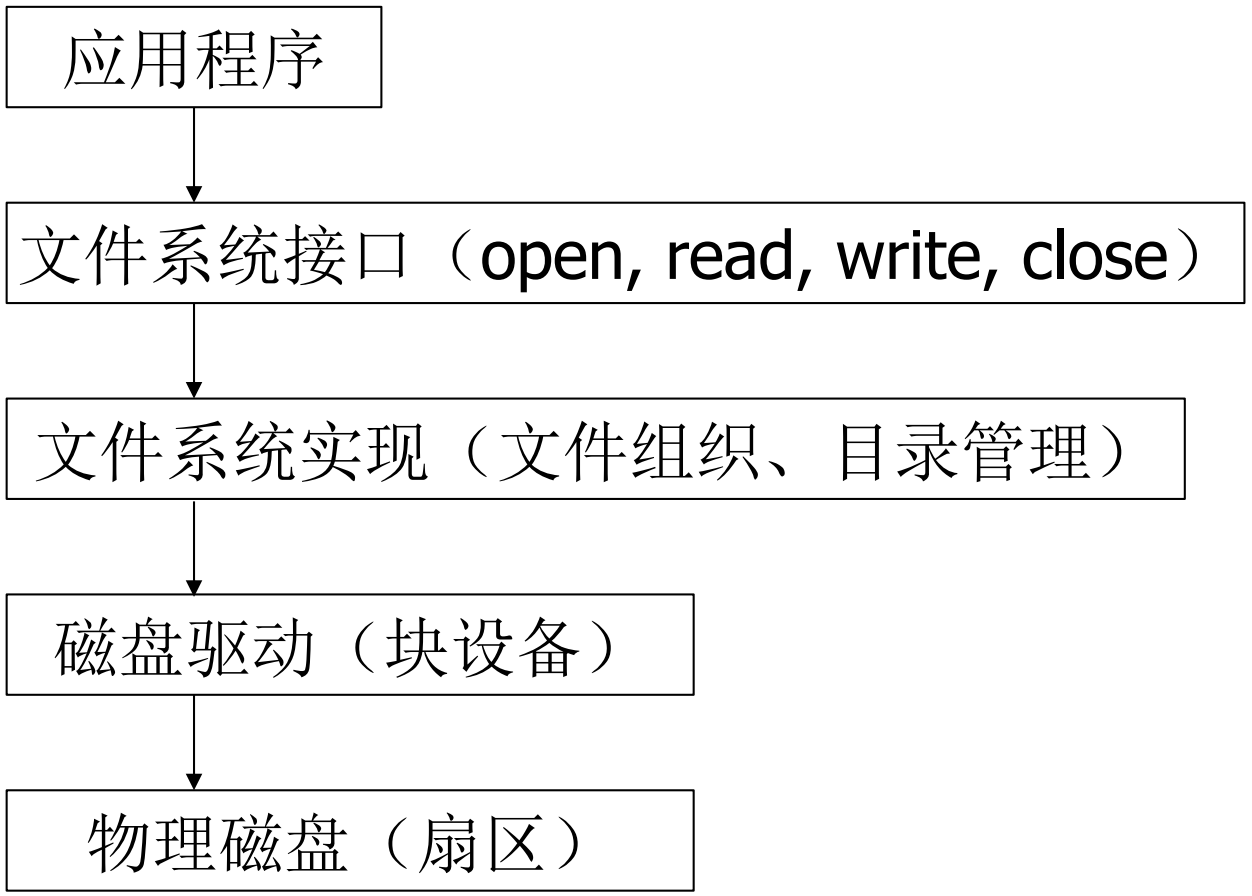


open()检查权限的流程

- 当进程调用open(path, flags)时：
  - **特权检查**：如果进程的**euid == 0**（即 root 用户），**直接放行**（除了执行权限需额外检查外），无视权限位。
  - **拥有者检查 (User)**：
    - 如果进程的 euid 等于文件的 Owner UID：
    - **只检查 User 位的权限**。
    - 如果 User 位允许（例如 r-- 且请求 O_RDONLY），则放行；否则拒绝。
    - **注意**：一旦匹配了 Owner，就不再看 Group 和 Others 了。即便 Group 有权限，只要 Owner 没权限，Owner 自己也被拒绝。
  - **群组检查 (Group)**：
    - 如果进程的 egid 等于文件的 Group GID，或者进程属于文件的 Group 组列表：
    - **只检查 Group 位的权限**。
    - 规则同上。
  - **其他人检查 (Others)**：
    - 如果以上都不匹配，**检查 Others 位的权限**。

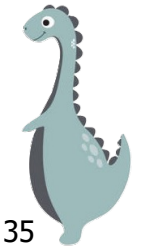


文件系统的抽象层次


应用程序
文件系统接口（open, read, write, close）
文件系统实现（文件组织、目录管理）
磁盘驱动（块设备）
物理磁盘（扇区）

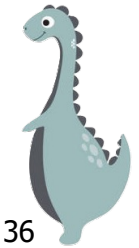
文件系统的功能

- 按名存取：文件名 → 物理地址映射
- 存储空间管理：分配与回收
- 访问控制：权限管理
- 数据保护：一致性、可靠性



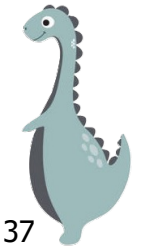
文件组织与分配

- **问题**：一个文件由多个数据块组成，如何在磁盘上组织这些块？
- **三种方案**：连续分配、链接分配、索引分配



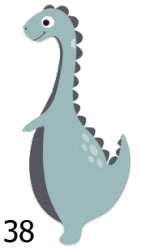
文件存储空间的分配

- 文件存储空间的分配常采用两种方式：
  - 静态分配：在文件建立时一次分配所需的全部空间。
  - 动态分配：根据需要进行分配。
- 在分配区域大小上，也可以采用不同方法。可以为文件分配一个连续区域，但文件存储空间的分配通常以块或簇（几个连续物理块称为簇，一般是固定大小）为单位。

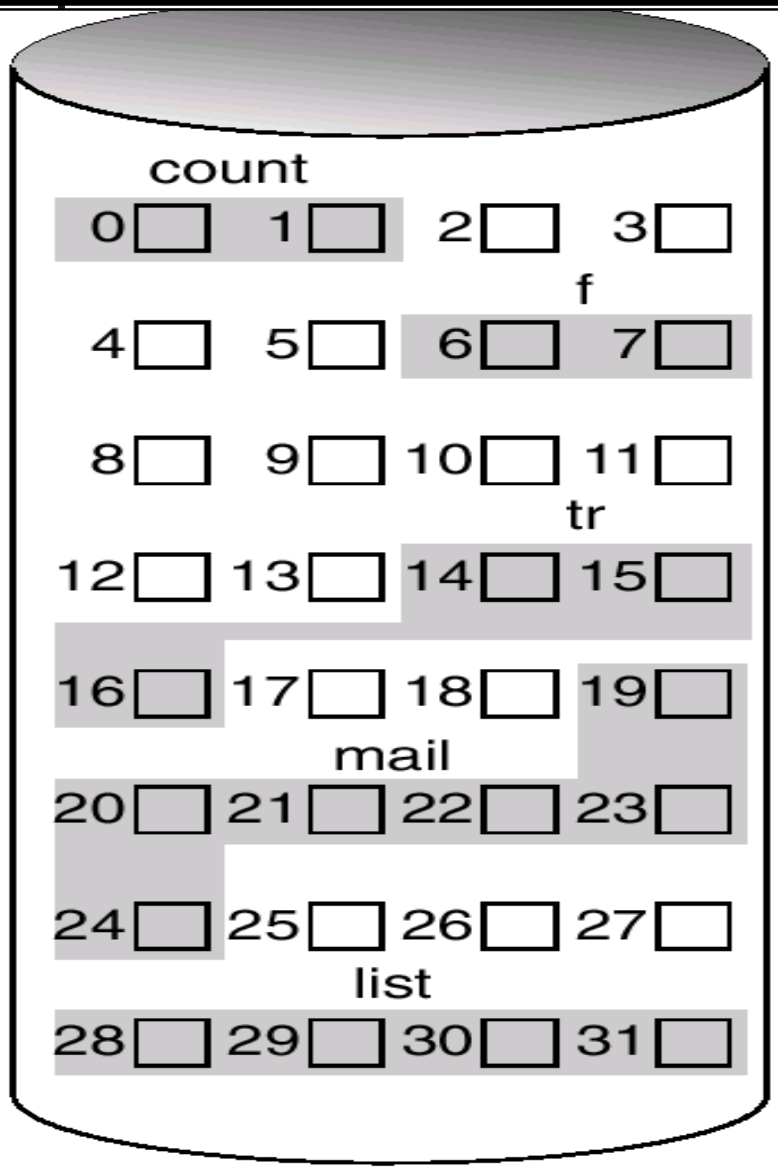


连续分配

- **连续分配**方法要求每个文件在磁盘上占有一组连续的块。
- 文件目录只需其起始位置（块号）及长度。
- 支持顺序及随机访问，但有外部碎片。



磁盘空间的连续分配

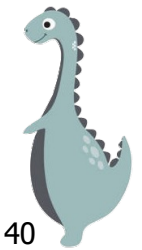


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

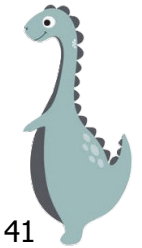
连续分配续

- 在这种分配方法中，用户必须在分配前说明待创建文件所需的存储空间大小。然后系统查找空闲区管理表格，若有就给文件分配所需的存储空间，否则文件不能建立。
- 连续分配的特点是：
  - 顺序访问容易且速度快，目录中文件存储位置信息简单；
  - 但容易产生碎片，需要定期对磁盘空间进行整理。
- 存在的问题
  - 为新文件找空间比较困难
  - 文件很难增长



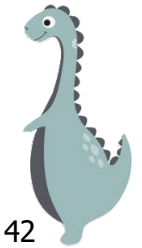
基于扩展的系统

- 许多新的文件系统采用一种修正的连续分配方法
- 该方法开始分配一块连续空间，当空间不够时，另一块被称为扩展的连续空间会添加到原来的分配中。
- 文件的块位置就为开始地址、块数、加上一个指向下一扩展的指针。



链接分配

- 链接分配有两种实现方案：
  - 以扇区为单位的链接分配
  - 以区段（或簇）为单位的链接分配

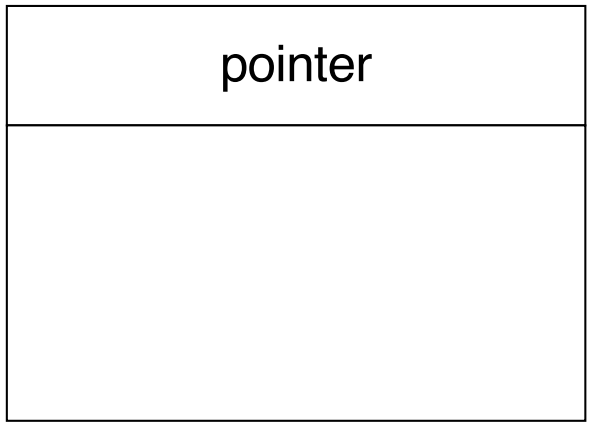


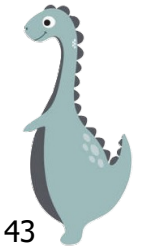
以扇区为单位的链接分配

- 每个文件是磁盘块的链表；磁盘块分布在磁盘的任何地方。

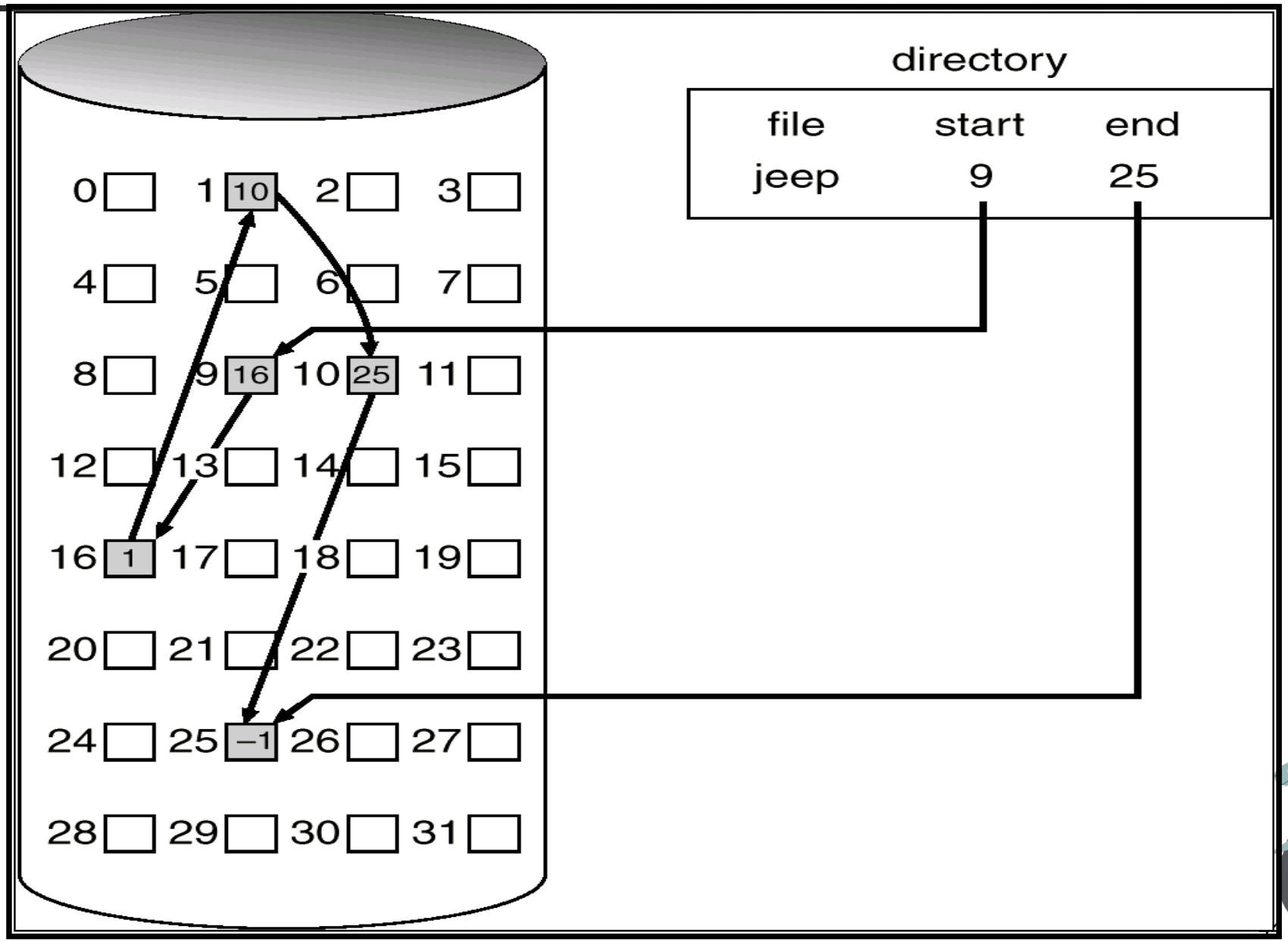
- 文件目录包含第一块的指针及最后一块的指针。

block =



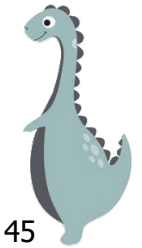


磁盘空间的链接分配


directory
file
start
end
jeep
9
25
0
1
10
2
3
4
5
6
7
8
9
16
10
25
11
12
13
14
15
16
1
17
18
19
20
21
22
23
24
25
−1
26
27
28
29
30
31

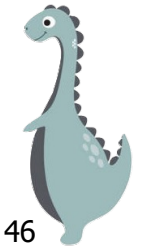
特点

- 优点：
  - 简单 — 只需起始位置
  - 文件创建与增长容易
- 缺点：
  - 不能随机访问
  - 块与块之间的链接指针需要占用空间
  - 存在可靠性问题，如指针损坏



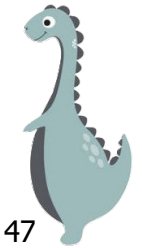
以区段（或簇）为单位分配

- 以区段（或簇）为单位分配：是连续分配和非连续分配的结合，现广为使用。区段由若干个连续扇区组成，文件所属各区段可以用链接指针、索引表等方法来管理。
- 此策略的优点是对辅存的管理效率较高，并减少了文件访问的查寻时间。

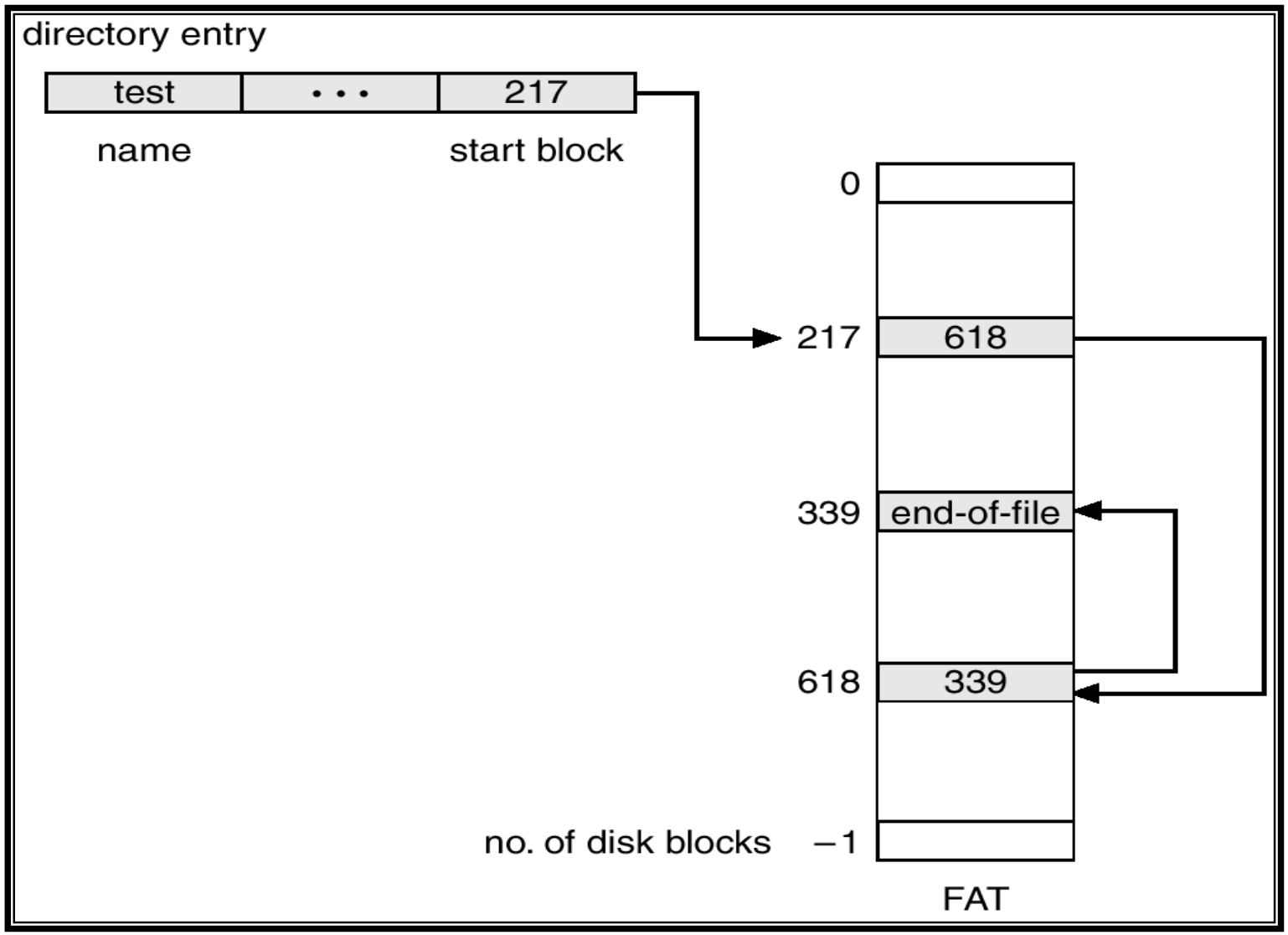


文件分配表

- 文件分配表FAT是以链接方式存储文件的系统中记录磁盘分配和跟踪空白盘块的数据结构。
- 该表整个文件系统仅设一张，其结构如下所示。表的序号是物理块号，从0开始直至N－1（N为盘块总数）。
- 每个表项中的内容为存放文件数据的下一个盘块号。
- 文件的首地址（第一个盘块号）存放在目录中。因此，从目录中找到文件的首地址后，就能找到文件在磁盘上的所有存放地址。

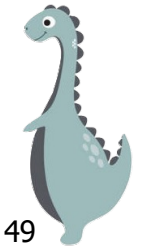


文件分配表示意图


directory entry
test
• • •
217
name
start block
0
217
618
339
end-of-file
618
339
no. of disk blocks
−1
FAT

文件分配表例1

- 假定磁盘块的大小为1KB，对于1.2MB的软盘，其文件分配表FAT需要占用多少存储空间？
- 若硬盘容量为200MB时，FAT需要占用多少空间？



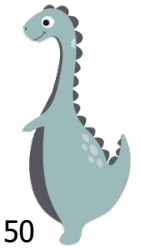
文件分配表例2

- 软盘大小为1.2MB，磁盘块的大小为1KB，所以该软盘共有盘块：1.2M/1K=1.2K（个）

  又　1K＜1.2K＜2K，

- 故1.2K个盘块号要用11位二进制表示，为了方便存取，每个盘块号用12位二进制描述，即文件分配表的每个表目为1.5个字节。
- FAT要占用的存储空间总数为：

  1.5×1.2K=1.8KB



文件分配表例3

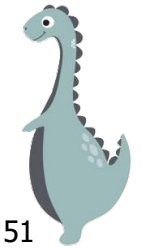
- 若硬盘大小为200MB，硬盘共有盘块：

  200M/1K=200K

  又 128K＜200K＜256K，

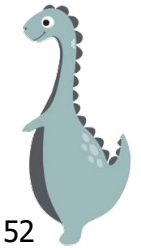
- 故200K个盘块号要用18位二进制表示。为方便文件分配表的存取，每个表目用20位二进制表示，即文件分配表的每个表目大小为2.5个字节。
- FAT要占用的存储空间总数为：

  2.5×200K=500KB



索引分配

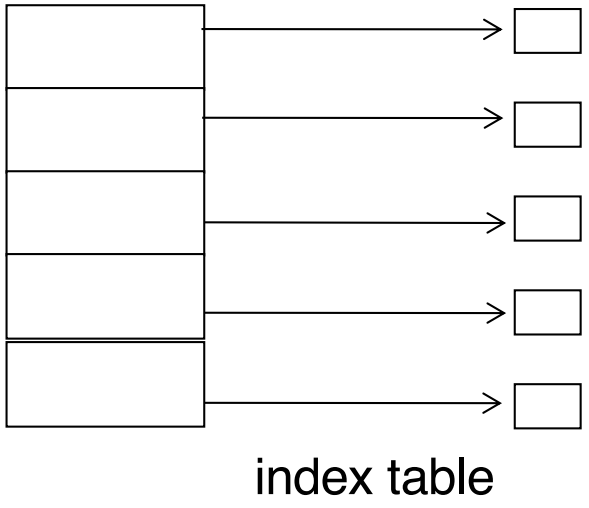
- 链接分配方式虽解决了连续分配方式中存在的问题，但又出现了新的问题：
  - 不支持随机存取
  - 链接指针要占用一定数量的磁盘空间

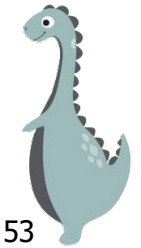


索引分配

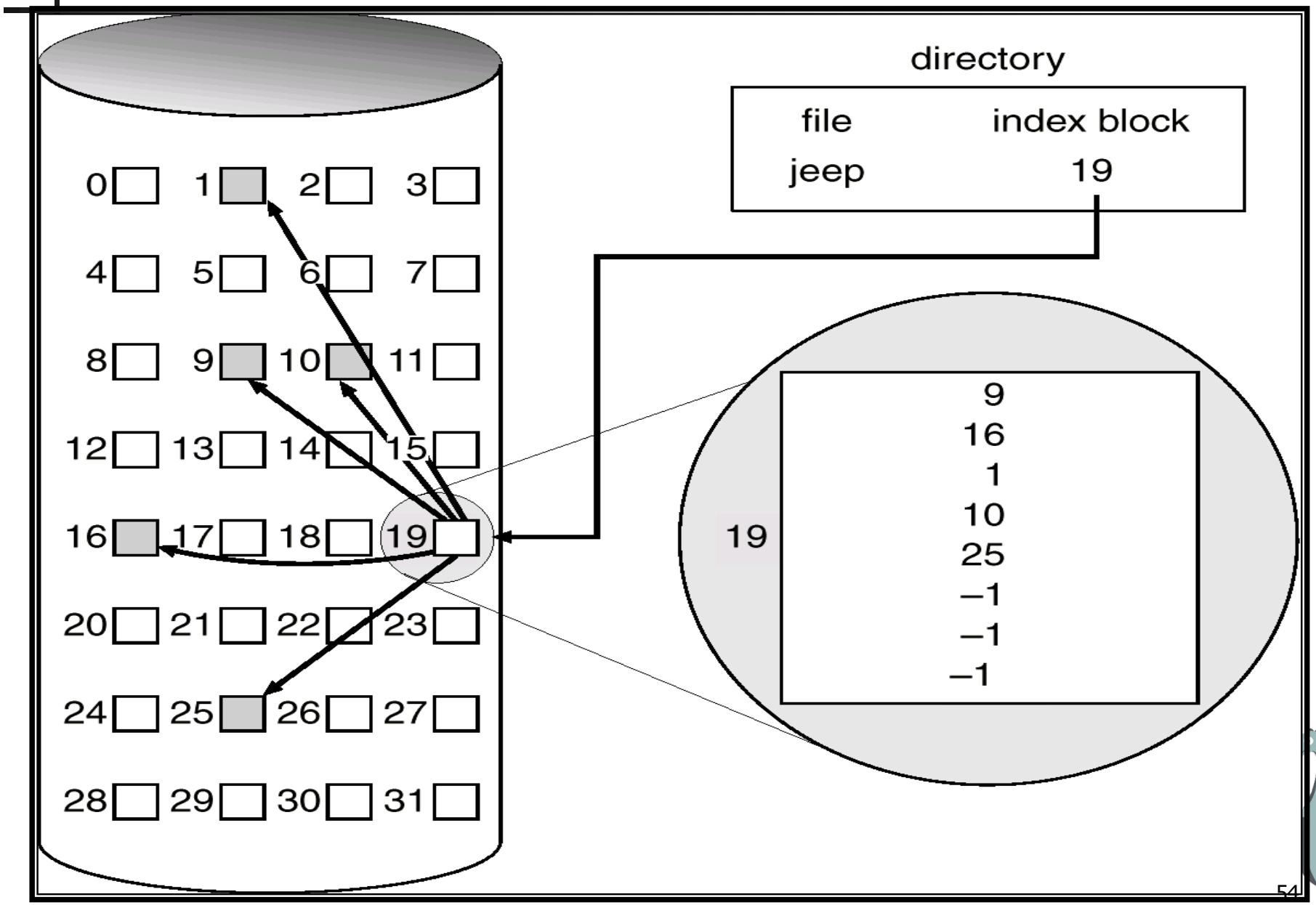
- 在索引分配方法中，系统为每个文件分配一个索引块，索引块中存放索引表，索引表中的每个表项对应分配给文件的一个物理块。

- 文件目录包含索引块地址。



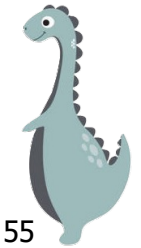


索引分配示意图


directory
file
index block
jeep
19
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31
19
9
16
1
10
25
−1
−1
−1

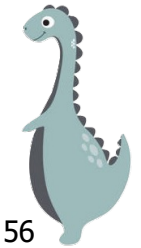
索引分配的特点

- 索引分配方法支持直接访问，不会产生外部碎片；
- 但索引块要占用一定的存储空间，存取文件需要两次访问外存。

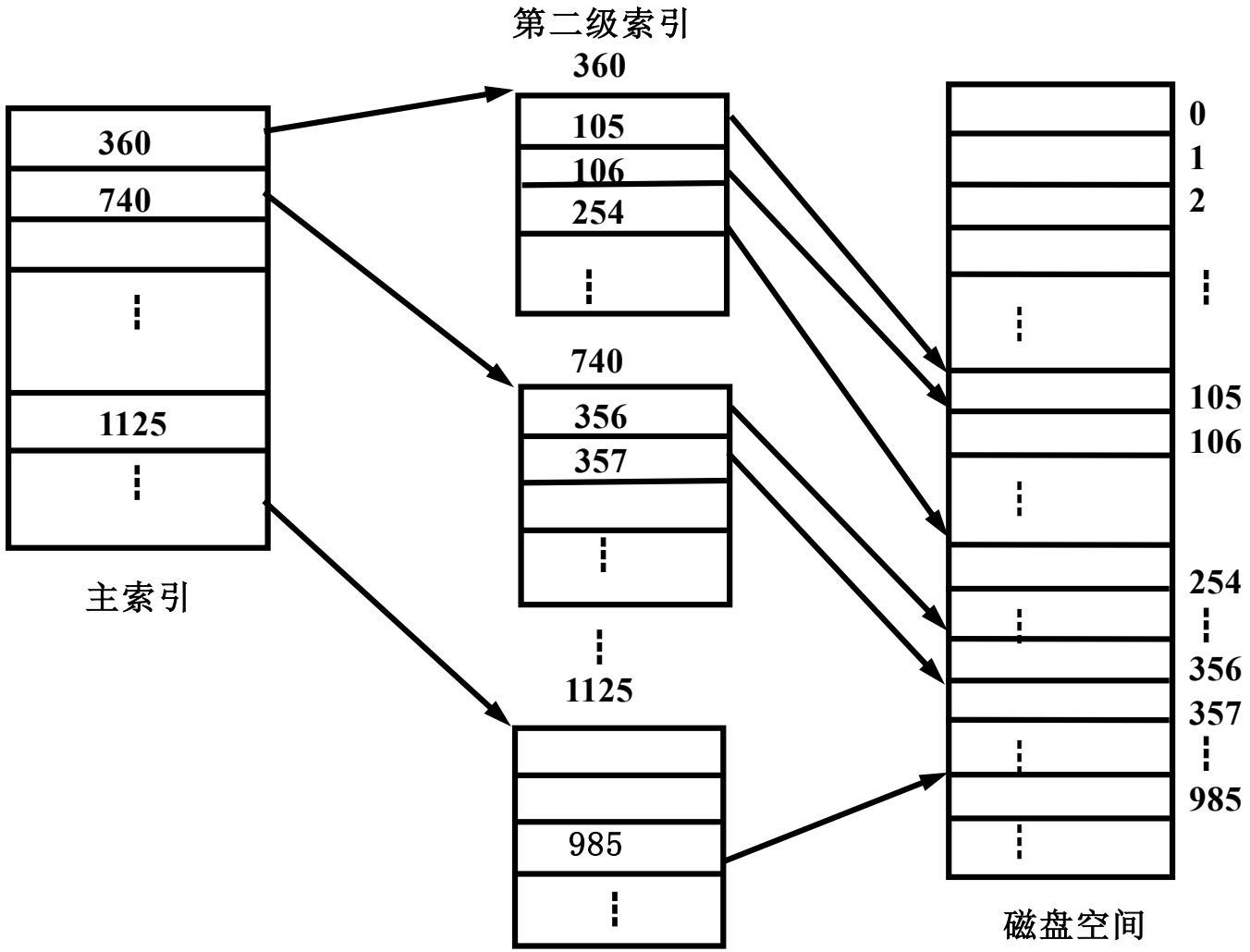


二级索引和多级索引

- 当文件很大，其索引表的大小超过了一个物理块时，可以将索引表本身作为一个文件，再为其建立一个“索引表”，该“索引表”是文件索引的索引，从而构成了**二级索引**。
- 第一级索引表的表目指向第二级索引，第二级索引表的表目指向文件信息所在的物理块号。以此类推可再逐级建立索引，进而构成多级索引。



两级索引分配示意图


第二级索引
360
360
740
1125
主索引
105
106
254
740
356
357
1125
985
0
1
2
105
106
254
356
357
985
磁盘空间

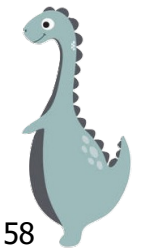
两级索引分配允许的文件最大长度

- 在两级索引分配方式下，如果每个盘块的大小为1KB，每个盘块号占4字节，则：
- 一个索引块中可以存放：

  1KB/4B=256个盘块号
- 两级索引最多可以存放的盘块数为：

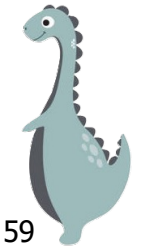
  256×256=64K个盘块号
- 因此可以允许的最大文件长度为：

  64K×1KB=64MB

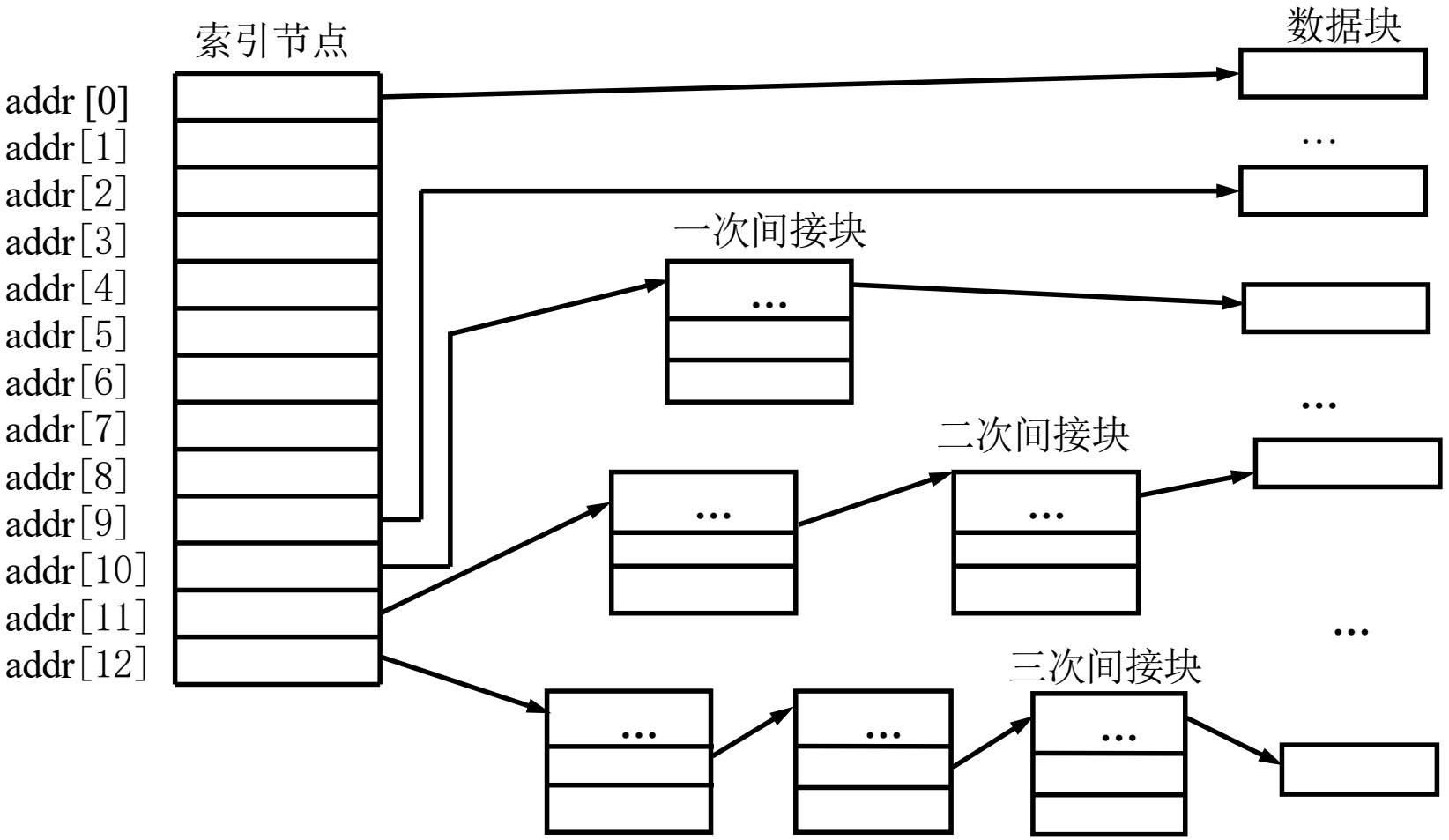


混合索引分配方式

- 混合索引分配方式是将多种索引分配方式相结合而形成的一种分配方式。这种方式已用于UNIX、Linux等系统中。
- 在UNIX System Ⅴ中，共设有13个地址项，包括10个直接地址项、一个一次间接地址项、一个二次间接地址项和一个三次间接地址项。

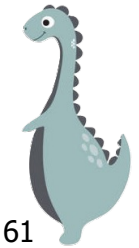


Unix混合索引方式示意图



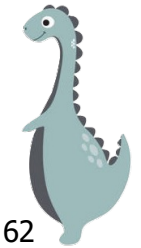
直接地址

- 为了提高对文件的检索速度，在索引节点中建立了10个直接地址项，每个地址项中存放相应文件所在的盘块号。
- 假定一个盘块的大小为4KB，当文件长度不大于40KB时，可以直接从索引节点中得到文件存储的所有盘块号。



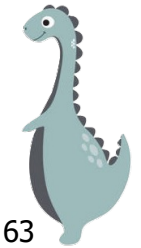
一次间接地址

- 一次间接地址项中存放的不是存储文件数据的盘块号，而是先将多个盘块号存放在一个磁盘块中，再将该磁盘块的块号存放在一次间接地址项中。
- 若盘块大小为4KB，一个盘块号占4字节，则一个盘块中可以存放下：4KB/4B=1K个磁盘块号。
- 一次间接地址项寻址范围为：1K×4KB=4MB。



多次间接地址

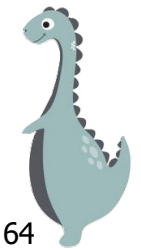
- 该地址结构中还有二次间接地址和三次间接地址。
- 二次间接地址的寻址范围是：1K×1K×4KB=4GB。
- 三次间接地址的寻址范围是：1K×1K×1K×4KB=4TB。



Xv6文件系统布局

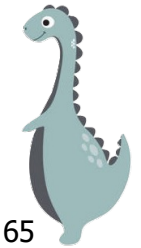
Boot	Super	Log	Inode Blocks	Bitmap	Data Blocks

- Boot: 引导代码
- Super: 文件系统元信息
- Log: 日志（崩溃恢复）
- Inode Blocks: Inode数组
- Bitmap: 空闲块位图
- Data Blocks: 实际数据



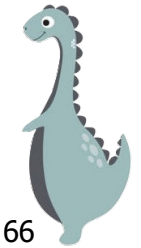
目录操作概览

- **核心观点：**目录不仅是数据结构，更是操作的集合
- **操作列表：**
  - 创建：mkdir
  - 删除：rmdir
  - 打开/关闭：opendir/closedir
  - 读取：readdir
  - 重命名：rename
  - 链接：link/symlink/unlink



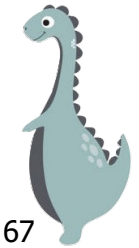
mkdir - 创建目录

- 系统调用：int mkdir(const char *pathname, mode_t mode);
- 内核步骤：
  - 1. 分配新Inode
  - 2. 在父目录添加目录项
  - 3. 初始化`.` 和 `..`
  - 4. 更新父目录修改时间



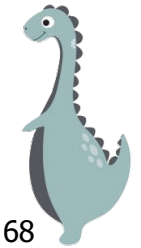
opendir/readdir - 遍历目录

```
DIR *dir = opendir("/tmp");
  struct dirent *entry;
  while ((entry = readdir(dir)) != NULL) {
    printf("%s (inode: %lu)\n",
           entry->d_name, entry->d_ino);
  }
  closedir(dir);
```



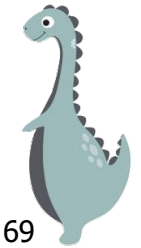
rename - 重命名操作

- 场景区分：
- 同目录：仅修改目录项中的文件名
- 跨目录：移动目录项，原子操作
- 示例：
- rename("/home/user/old.txt", "/tmp/new.txt")
- → 跨目录移动



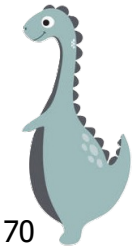
硬链接 vs 软链接

- 硬链接（Hard Link）：
  - 多个目录项 → 同一Inode
  - 引用计数机制
    - 删除一个，其他仍可访问
  - 限制：不能跨文件系统、不能链接目录
- 软链接（Symbolic Link）：
  - 独立文件，存储目标路径
  - 可跨文件系统、可链接目录
  - 风险：目标删除后断链



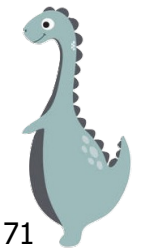
unlink - 解除链接

- **机制：**
- 1. 从目录删除目录项
- 2. Inode引用计数 -1
- 3. **延迟删除：** nlink=0 且无进程打开时，释放磁盘空间
- **问题：** 为什么删除文件后空间不立即释放？
- **答案：** 有进程持有文件描述符时延迟删除



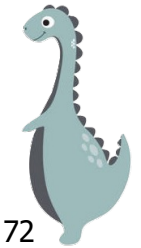
文件操作：open的实现

- 系统调用：int fd = open("file.txt", O_RDONLY);
- 内核数据结构：
  - 进程 → 文件描述符表 → 打开文件表 → Inode
- 步骤：
  - 1. 路径解析（逐级查找目录）
  - 2. 权限检查
  - 3. 分配打开文件表项
  - 4. 分配fd，返回



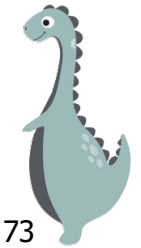
文件操作：read的实现

- 核心问题：一个简单的read()，内核做了什么？
- 执行流程：
  - 1. fd → 打开文件表项
  - 2. 获取文件指针位置
  - 3. 地址转换：逻辑偏移 → 物理块号
  - 4. 缓冲区管理：检查页缓存
  - 5. 复制数据到用户空间
  - 6. 更新文件指针



文件操作：write的延迟写入

- 写缓冲（Write Buffering）：先写页缓存
- 延迟写入（Delayed Write）：后台刷新
- 元数据更新：修改时间、文件大小
- 一致性风险：崩溃可能丢失数据
- 解决方案：日志文件系统（Journaling）



文件系统挂载——统一命名空间

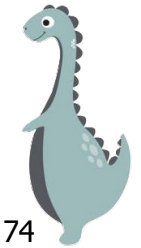
- 物理视角

独立的文件系统：

- 逻辑视角（挂载后）

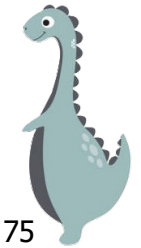
统一目录树：

```
/
├── home/          ← sda1挂载点
│   └── user/
├── mnt/
│   └── usb/       ← sdb1挂载点
└── tmp/           ← sdc1挂载点
```



挂载的本质

- **定义**：将外部文件系统的根目录连接到已有目录树的某个节点
- **示例**：
  - 挂载前：/mnt/usb (空目录)
  - 挂载后：/mnt/usb → /dev/sdb1的根目录



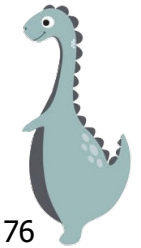
挂载点与挂载表

- 挂载示例

```
# 查看当前挂载
$ mount
/dev/sda1 on / type ext4 (rw,relatime)
/dev/sdb1 on /home type xfs (rw,nodev)
/dev/sdc1 on /mnt/usb type vfat (rw,noexec)

# 手动挂载
$ sudo mount -t ext4 /dev/sdd1 /mnt/data
              \_类型  \_设备  \_挂载点
```

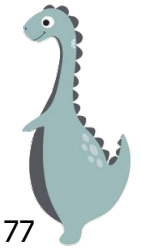
- 挂载表

```
# 设备        挂载点     类型   选项          dump fsck
/dev/sda1     /          ext4   defaults      0    1
/dev/sdb1     /home      xfs    defaults      0    2
UUID=xxx      /boot      ext4   defaults      0    2
```



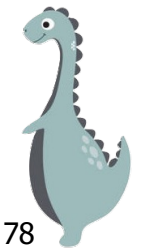
Linux挂载命令

- 挂载：
  - sudo mount -t ext4 /dev/sdb1 /mnt/usb
- 查看：
  - mount | grep sdb1
- 卸载：
  - sudo umount /mnt/usb
- 挂载选项：ro（只读）、rw（读写）、noexec（禁止执行）



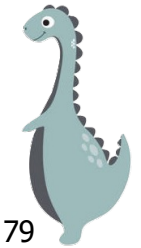
挂载的实现机制

- **步骤：**
- **1. 读取超级块：** 验证文件系统类型
- **2. 连接根目录：** 挂载点dentry → 新文件系统根Inode
- **3. 标记挂载点：** d_mounted标志
- **VFS挂载树：**
- /
- ├── /home (mount: /dev/sdb1, xfs)
- └── /mnt/usb (mount: /dev/sdc1, fat32)



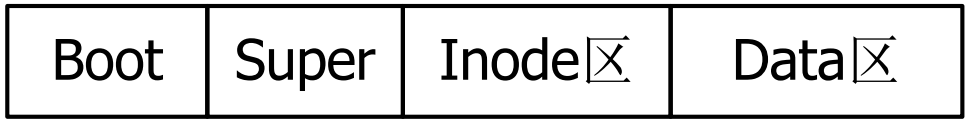
典型文件系统实现

- 两大典型文件系统：
  - 1. UNIX文件系统：服务器、工作站
  - 2. FAT文件系统：U盘、SD卡
- 对比视角：设计哲学、磁盘布局、性能特点

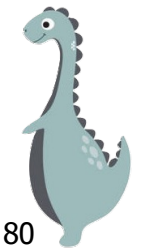


Unix文件系统的设计哲学

- 设计理念：
  - 简洁：Inode与目录分离
  - 灵活：硬链接、符号链接
  - 可靠：引用计数机制
- 磁盘布局：



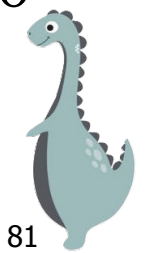
- 超级块内容：
  - 文件系统大小、块大小
  - 空闲Inode数、空闲块数
  - 魔数（标识文件系统类型）



Inode结构与容量计算

- Inode结构

- 最大文件大小计算题
  - 假设：块=4KB，指针=4B
  - 直接: 12×4KB = 48KB
  - 一级间接: 1024×4KB = 4MB
  - 二级间接: $1024^2 \times 4KB = 4GB$
  - 总计: ≈4GB
- 成组链接法
  - 超级块记录100个空闲块号 → 第一块记录下一组100个

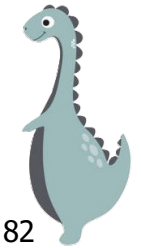


FAT的设计理念

- 磁盘布局

Boot	FAT1	FAT2（备份）	Root目录	Data区

- 设计特点：
  - 简单：FAT表集中管理
  - 兼容：所有OS支持
  - 应用：U盘、SD卡



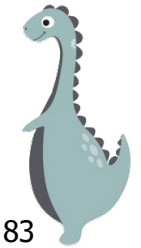
FAT表工作原理：簇链追踪

- FAT表示例：

```
FAT[0] = 保留
FAT[1] = 保留
FAT[2] = 5        (文件A: 簇2→5)
FAT[3] = 0        (空闲)
FAT[4] = 0        (空闲)
FAT[5] = 0xFFFF   (文件结束)
```

- FAT版本对比

版本	表项位数	最大簇数	应用
FAT12	12位	4K	软盘
FAT16	16位	64K	≤2GB
FAT32	28位	268M	U盘

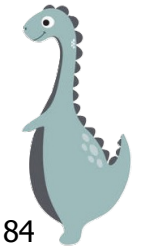


为什么FAT32单文件<=4GB?

- 目录项结构

```
struct fat_dirent {
  char name[8];
  char ext[3];
  uint8 attr;
  uint16 time;
  uint16 clus_hi;
  uint16 clus_lo;
  uint32 size;    // ← 32位!
};
```

- 限制分析
  - size字段：32位无符号整数
  - 最大值：$2^{32}-1 \approx 4GB$
- 解决方案
  - exFAT：64位size字段
  - NTFS：支持大文件



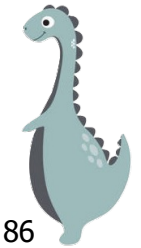
UNIX vs FAT对比

- 对比表

特性	UNIX	FAT32
元数据	Inode（分离）	目录项（集中）
分配方式	索引（多级）	链接（FAT表）
最大文件	TB级	4GB
长文件名	原生支持	VFAT扩展
可靠性	高	中（FAT备份）
复杂度	高	低
应用	服务器/工作站	可移动存储

文件系统性能瓶颈

- 磁盘I/O是瓶颈
  - 机械硬盘：10ms/次
  - SSD：0.1ms/次
  - 内存：100ns
  - 差距: 1000-100000倍！
- 优化策略
  - 减少磁盘I/O次数
    - 缓存（Caching）：最近访问的数据保留在内存
    - 预取（Read-Ahead）：预测性读取
    - 延迟写入（Delayed Write）：批量写入
    - I/O调度（I/O Scheduling）：优化访问顺序

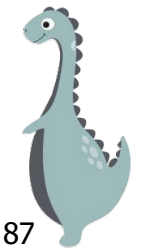


缓冲区缓存

- 原理
  - 单位：磁盘块（4KB）
  - 作用：缓存最近访问的块
- LRU替换算法

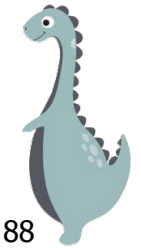
```
访问块100:
if (block 100 in cache):
    move to head
else:
    evict LRU block
    load block 100
```

- 写策略
  - 写通（Write-Through）：立即写磁盘（慢但安全）
  - 写回（Write-Back)：延迟写入（快但风险）



页面缓存与统一缓存

- 页面缓存
  - 单位：内存页（4KB）
  - 支持mmap()
- 统一缓存（现代Linux）
  - Buffer Cache + Page Cache → 统一Page Cache
- 优势
  - 避免重复缓存
  - 内存利用率↑

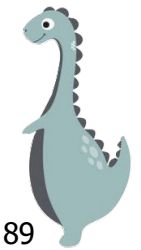


预取技术（Read-Ahead）

- 原理：检测顺序访问，提前读取后续块
- 实现：顺序访问检测

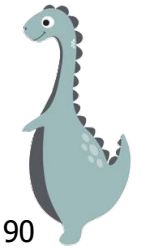
```
if current_offset == last_offset + block_size:
    # 顺序访问，触发预取
    read_ahead(next_N_blocks)
```

- 预取窗口：
  - 初始4-8块
  - 连续命中：则窗口扩大
  - 随机访问：则窗口缩小
- 配合：I/O调度器优化磁盘寻道



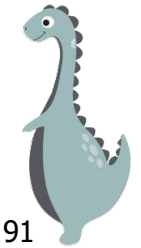
延迟写入（Delayed Write）

- 流程：
  - 1. write() → 写入Page Cache → 标记为脏页（Dirty）
  - 2. pdflush线程定期将脏页写回磁盘
  - 3. sync系统调用强制刷新
- 优势：批量写入，减少I/O次数
- 风险：崩溃可能丢失脏页
- 解决：
  - 日志文件系统（ext3/ext4、xfs）
  - 写时复制（btrfs/zfs）



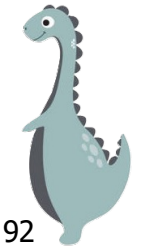
性能提升效果

- 示例：
  - 无缓存：每次read() → 10ms磁盘I/O
  - 有缓存：命中read() → 100ns内存访问
  - 加速比：100,000倍
- 缓存命中率：
  - 通常80-95%命中
  - 80%请求100ns，20%请求10ms
  - 平均：2.00008ms
- 综合提升：约5倍



虚拟文件系统VFS

- VFS设计动机—— 支持多文件系统的挑战
- 问题：
  - Linux支持100+文件系统
  - 如何统一接口？
- VFS目标
  - 对上：统一POSIX接口
  - 对下：抽象文件系统差异



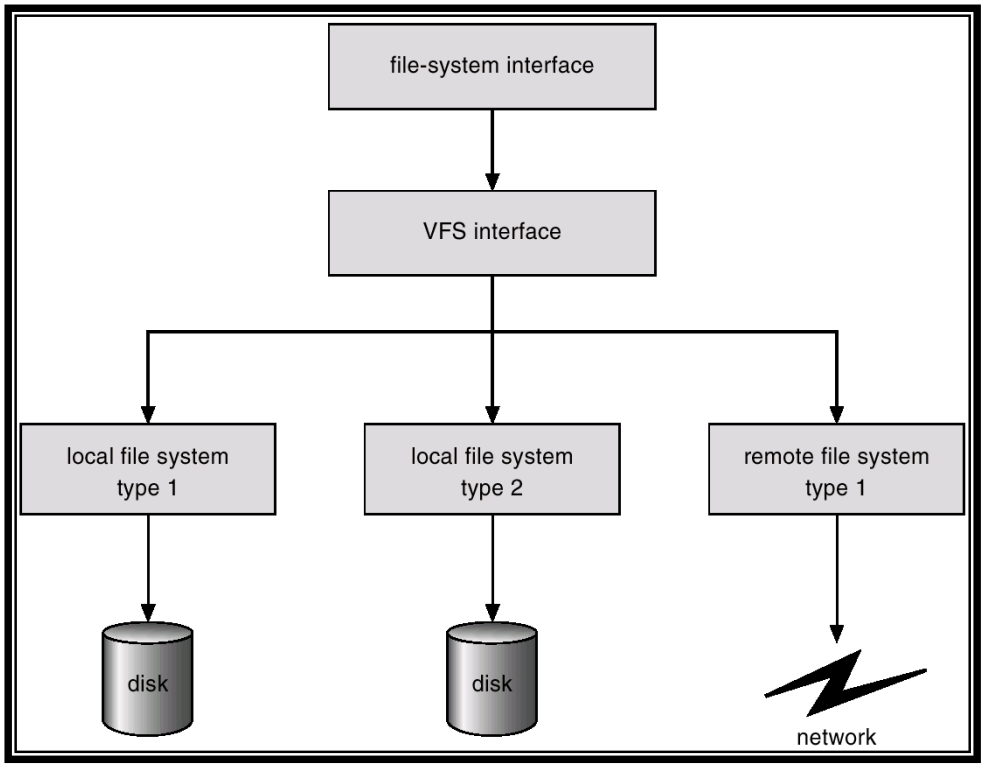
VFS三层架构

应用层：Open(), read(), write()

VFS层：统一接口

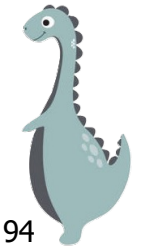
具体文件系统：ext4, xfs, ntfs, nfs

块设备层/网络层/内存



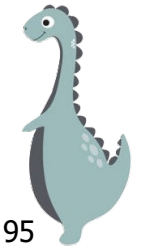
VFS四大核心数据结构

- 1. superblock
  - 代表已挂载的文件系统
- 2. inode
  - 文件元数据
- 3. dentry
  - 目录项缓存（dcache）
- 4. file
  - 打开的文件实例



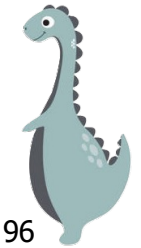
VFS操作接口

- 1. super_operations：涉及文件系统整体的操作
  - alloc_inode(): 分配一个新Inode
  - write_super(): 把超级块写回磁盘
- 2. inode_operations：涉及具体文件的元数据操作
  - create(): 创建新文件
  - lookup(): 在目录中查找文件名
  - mkdir(): 创建目录
- 3. file_operations：涉及文件内容的操作
  - read(): 读数据
  - write(): 写数据
  - open(): 打开
  - release(): 关闭
- ext4、xfs各自实现这些函数



read（）穿越VFS

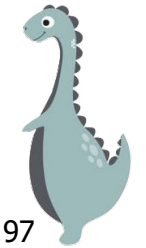
```
// 1. 系统调用
sys_read(fd, buf, count)
  ↓
// 2. VFS层
file = fget(fd)
file->f_op->read(file, buf, count)
  ↓
// 3. ext4实现
ext4_file_read(...)
  → generic_file_read_iter(...)
```



VFS的价值

- 支持多样性
  - 磁盘文件系统：ext4, xfs, ntfs
  - 网络文件系统：nfs, cifs
  - 伪文件系统：/proc, /sys
- 统一接口
  - 读设备：read(/dev/sda)
  - 读进程信息：read(/proc/1/status)
  - 读文件：read(/home/file.txt)

“一切皆文件”的基础



总结

7：VFS统一抽象

↓

6：性能优化（缓存/预取）

↓

5：典型实现（Unix/FAT）

↓

4：操作接口（系统调用）

↓

3：目录管理（name/链接）

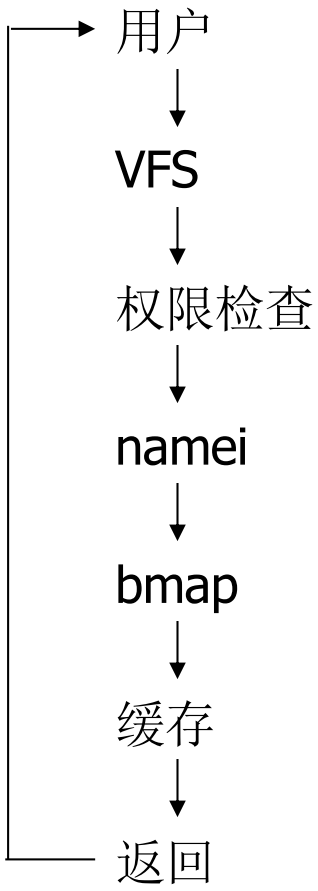
↓

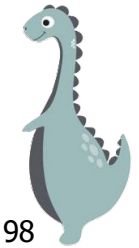
2：空间组织（分配策略）

↓

1：基础抽象（Inode/元数据）

Read(fd, buf, 100)完整路径





练习题

- 1. 假设一个 Unix 风格的文件系统，其 Inode 结构包含 10 个直接指针、1 个一级间接指针、1 个二级间接指针和 1 个三级间接指针。 已知：磁盘块大小（Block Size）= 8 KB，磁盘地址（指针）大小 = 8 字节。

  请计算：

（1）该文件系统支持的最大文件大小是多少？（请写出精确的计算公式，结果可以用 GB/TB 表示）

（2）如果要访问该文件偏移量为 128 GB 处的数据，需要访问 Inode 中的哪一级指针？需要进行几次磁盘 I/O（假设 Inode 已在内存，其他均未缓存）？

练习题

- 2.阅读以下 C 语言代码片段，假设 test.txt原本是一个空文件。请分析两种场景下的执行结果，并画出内核中 fd、file对象 和 inode 的连接关系图。

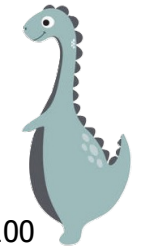
场景A：使用fork()

```c
int fd = open("test.txt", O_WRONLY);
if (fork() == 0) {
    write(fd, "hello", 5);
    exit(0);
} else {
    wait(NULL);
    write(fd, "world", 5);
}
```

场景B：两次独立的open()

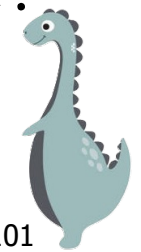
```c
int fd = open("test.txt", O_WRONLY);
if (fork() == 0) {
    int fd_child = open("test.txt", O_WRONLY);
    write(fd_child, "hello", 5);
    exit(0);
} else {
    wait(NULL);
    write(fd, "world", 5);
}
```

- **问题：**
  - 场景 A 执行后，test.txt 的内容是什么？（helloworld 还是 world？）
  - 场景 B 执行后，test.txt 的内容是什么？
  - **核心追问：**为什么结果不同？请用文件偏移量（f_pos）的存储位置来解释。



练习题

- 3.文件系统通常不允许对目录创建硬链接，以防止目录树变成“图”甚至出现环路。但是，rename 操作如果不加检查，也可能破坏树状结构。
- **问题：** 假设当前的目录结构是：/A/B/C。 如果用户尝试执行 rename("/A/B", "/A/B/C/D")（即试图把目录 B 移动到其子目录 C 下面并改名为 D），文件系统必须**拒绝**这个操作。
- 请解释：
  - 如果允许这个操作，目录结构会变成什么详细形状？（画出结构图）
  - 这种结构会对文件遍历程序（如 find寻找文件 或 du统计大小）造成什么致命后果？
  - 在实现 rename 系统调用时，操作系统如何检测这种非法操作？



练习题

- 13.5 Explain the purpose of the open() and close() operations.

- 14.1 Consider a file currently consisting of 100 blocks. Assume that the file-control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow at the beginning but there is room to grow at the end. Also assume that the block information to be added is stored in memory.

a. The block is added at the beginning.

b. The block is added in the middle.

c. The block is added at the end.

d. The block is removed from the beginning.

e. The block is removed from the middle.

f. The block is removed from the end.

考研题1

- 设文件索引节点中有7个地址项，其中4个地址项为直接地址索引，2个地址项是一级间接地址索引，1个地址项是二级间接地址索引，每个地址项大小为4字节，若磁盘索引块和磁盘数据块大小均为256字节，则可表示的单个文件最大长度是____。10

A.33KB B. 519KB C. 1057KB D. 16613KB

考研题2-1

- 某文件系统空间的最大容量为4TB（1TB=240），以磁盘块为基本分配单元。磁盘块大小为1KB。文件控制块（FCB）包含一个512B的索引表区。请回答下列问题。12
- （1）假设索引表区仅采用直接索引结构，索引表区存放文件占用的磁盘块号，索引项中块号最少占多少字节？可支持的单个文件最大长度是多少字节？
- （2）假设索引表区采用如下结构：第0～7字节采用<起始块号，块数>格式表示文件创建时预分配的连续存储空间。其中起始块号占6B，块数占2B，剩余504字节采用直接索引结构，一个索引项占6B，则可支持的单个文件最大长度是多少字节？为了使单个文件的长度达到最大，请指出起始块号和块数分别所占字节数的合理值并说明理由。

考研题2-2

- （1）文件系统存储空间共有块数=$2^{42}/2^{10}=2^{32}$。为表示2^{32}个块号，索引表项占32/8=4B。
- 512可存放2^7个索引表项，故最大文件长度=$2^7\times2^{10}=2^{17}$B=128KB。
- （2）块号占6字节，块数占2字节的情况下，最大文件长度=$2^{16}\times2^{10}+(504/6)\times2^{10}$=64MB+84KB。=65620KB
- 合理的起始块号和块数所占字节数分别为4，4（1，7或2，6或3，5）。因为块数占4B或以上，就可表示4TB大小的文件长度，达到文件系统的空间上限。

考研题3

- 下列文件物理结构中，适合随机访问且易于文件扩展的是（）09

  A、连续结构　　C、链式结构且磁盘块定长

  B、索引结构　　D、链式结构且磁盘块变长

- 设置当前工作目录的主要目的是____。10

A. 节省外存空间　　B. 节省内存空间

C. 加快文件的检索速度　　D. 加快文件的读/写速度

考研题4

- 文件系统中，文件访问控制信息存储的合理位置是（）。09

  A、文件控制块　　B、文件分配表

  C、用户口令表　　D、系统注册表

- 设文件F1的当前连接计数为1，先建立F1的符号链接（软连接）文件F2，再建立F1的硬链接文件F3，然后删除F1。此时F2和F3的连接计数值分别是（）。09

  A、0、1　　B、1、1

  C、1、2　　D、2、1

考研题5

- 某文件系统为一级目录结构，文件的数据一次性写入磁盘，已写入的文件不可修改，但可多次创建新文件。请回答如下问题。11
- （1）在连续、链式、索引三种文件的数据块组织方式中，哪种更合适？要求说明理由。为定位文件数据块，需在FCB中设计哪些相关字段？
- （2）为快速找到文件，对于FCB，是集中存储好，还是与对应的文件数据块连续存储好？要求说明理由。

考研题5答案

- (1)在磁盘中连续存放(采取连续结构)，磁盘寻道时间更短，文件随机访问效率更高；在FCB中加入的字段为：<起始块号，块数>或者<起始块号，结束块号>。
- (2)将所有FCB集中存放，文件数据集中存放。这样在随机查找文件名时，只需访问FCB对应的块，可减少磁头移动和磁盘I/O访问的次数。

考研题6

- 若一个用户进程通过read系统调用读取一个磁盘文件中的数据，则下列关于此过程的叙述中，正确的是 12

Ⅰ.若该文件的数据不在内存，则该进程进入睡眠等待状态

Ⅱ.请求read系统调用会导致CPU从用户态切换到核心态

Ⅲ.read系统调用的参数应包括文件的名称

- A.仅Ⅰ，Ⅱ　　B.仅Ⅰ，Ⅲ
- C. 仅Ⅱ，Ⅲ　　D. Ⅰ，Ⅱ，Ⅲ