章节6：文件管理

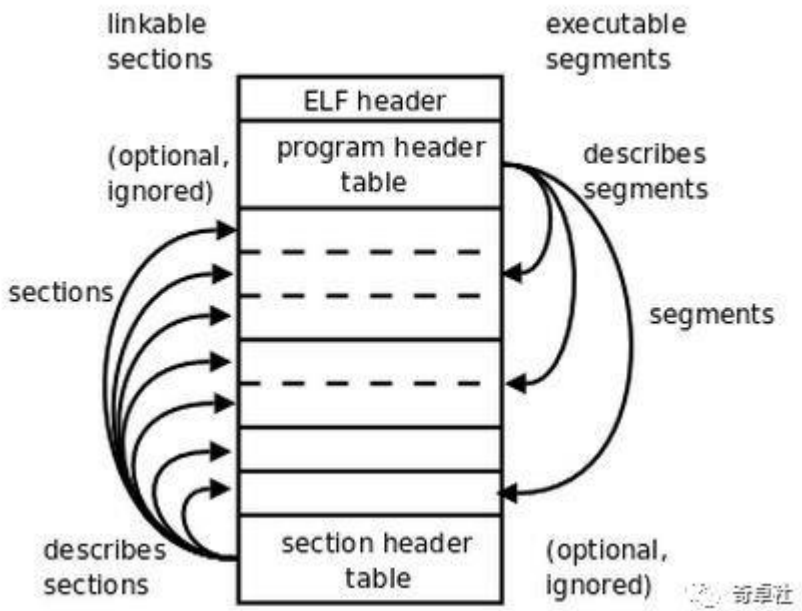
前情提要

- 文件系统是将固定大小的块数据（扇区）组织成任意长度（文件，或者磁带）的数据结构，并提供增删改查机制的方法。参考数据结构课程，会有连续、链表、树等模式
- 文件夹是文件的索引，其中保存了文件名，以及其他附属信息
- 文件夹也是一个文件，也需要存在磁盘上
- 一个磁盘（分区）上所有的文件都按树状组织，打开一个磁盘，就需要找到树根。为此，树根存在了磁盘固定的位置上
- 文件系统需要处理的最重要的问题之一是性能差异。因为磁盘和内存、CPU之间存在数万倍以上的性能带差，因此需要借助内存设计若干种缓存机制以尽可能的减少对磁盘的读写操作
- 打开文件句柄、文件指针用于暂存文件的位置，以及上次读取的位置，以避免反复读磁盘查找
- 文件系统中设置了数据缓存机制，以期待尽可能利用数据的局部性，减少重复的读操作，合并相临的写操作
- 文件系统的缓存机制与虚拟内存机制存在很强的关联性和相似性，因此两个机制合并为一套页面缓存(page cache)机制，以4K为单位缓存数据，并按照访问时间频度判定数据写回的时机
- 与Cache一样，缓存机制放大了读区域，延后了写操作。放大的读区域造成的代价可以被批量读掩盖，延后的写操作给系统带来了很多的困扰，最典型的情况是写回时无法知道数据的优先级和来源等语义
- 为了优化文件系统的性能，OS做了很多的优化，其中之一是预取机制。但是预取目前只能简单的判定程序读写文件的线性趋势，无法实现对复杂趋势的拟合
- 除了缓存外，另一个优化的手段是尽可以发挥硬件的性能。例如，可以将多个硬盘组成阵列，以提升系统的总体带宽，提高容错性；也可以减少磁头的移动，以减少磁盘数据读写过程中的延迟。

文件系统的性能思考

- OS喜欢做顺序的读写以改进性能，为什么？—>线性的（磁盘是顺着读的）
  - 简单
  - 尊重磁盘的物理特性
- 用户会按照什么样的方式读写磁盘呢？
  - 执行程序
  - 播放电影
  - 读写数据库
  - ……

示例：elf文件格式

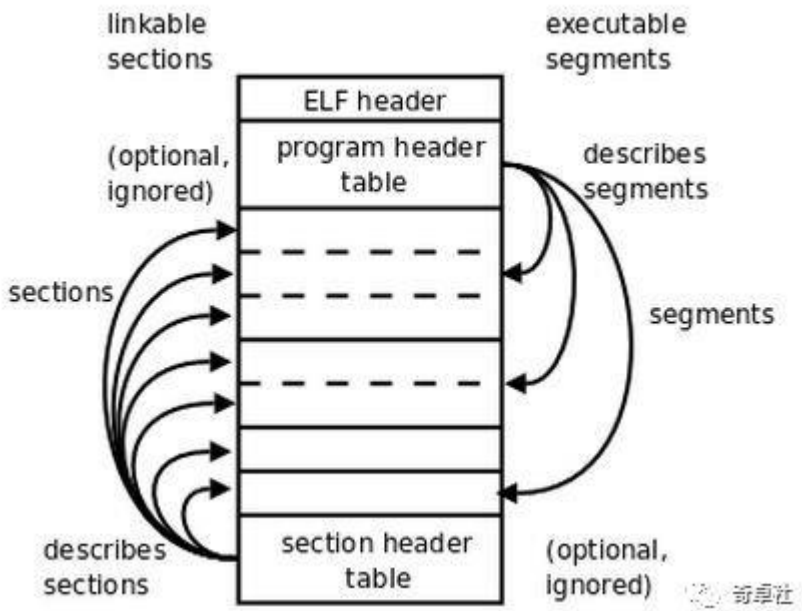



```
#define EI_NIDENT       16

typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
```

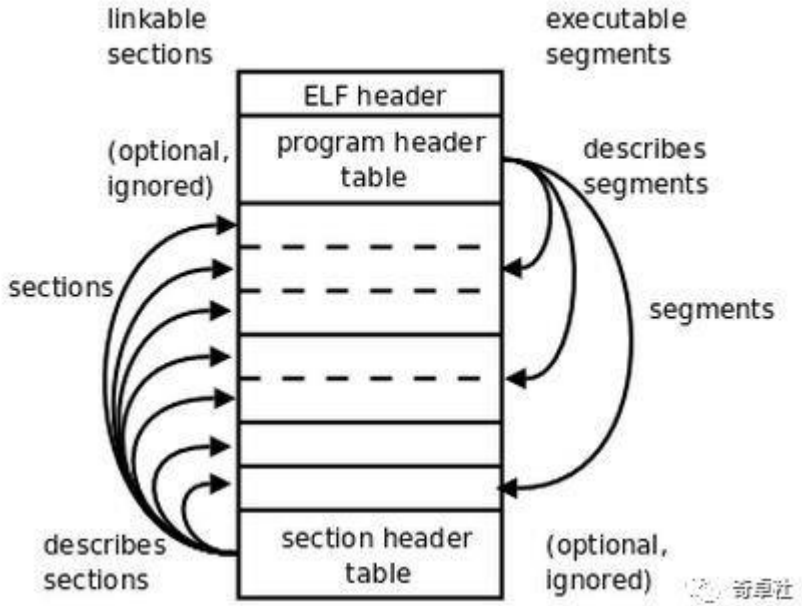
磁盘上存储：可执行程序、数据库、电影
可执行程序存储：内容稀疏，存储：在虚拟地址上的起始位置和长度、在磁盘上的开始位置

示例：elf文件格式




```
typedef struct {
        Elf32_Word      p_type;
        Elf32_Off       p_offset;
        Elf32_Addr      p_vaddr;
        Elf32_Addr      p_paddr;
        Elf32_Word      p_filesz;
        Elf32_Word      p_memsz;
        Elf32_Word      p_flags;
        Elf32_Word      p_align;
} Elf32_Phdr;
```

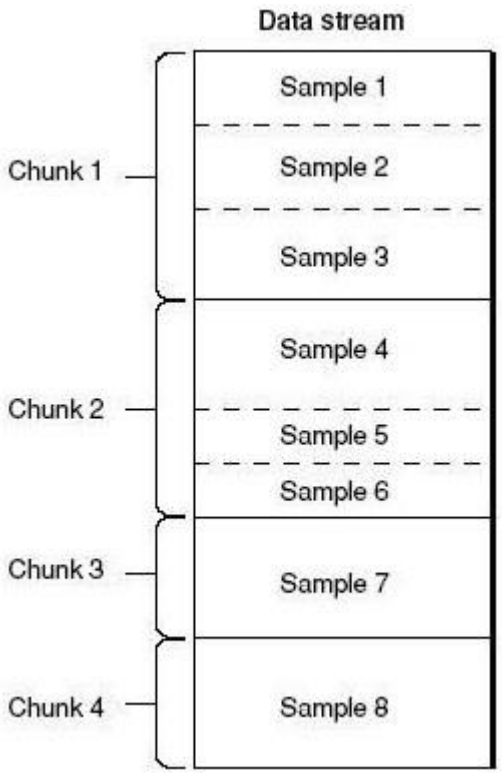
示例：elf文件格式

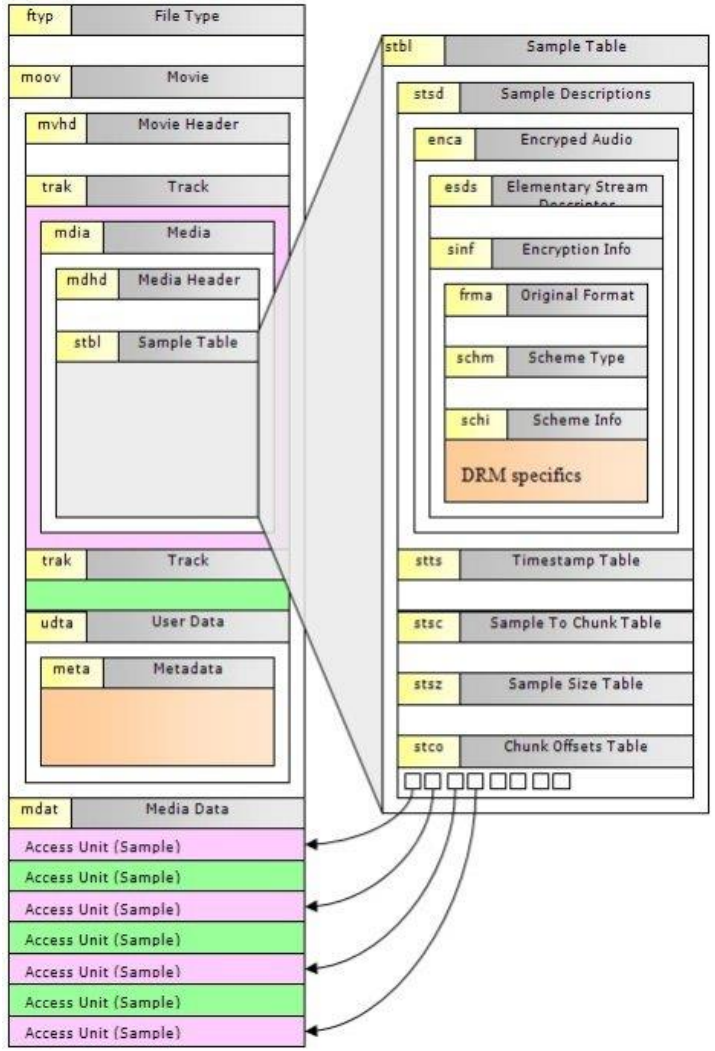



```
typedef struct {
        Elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

https://cloud.tencent.com/developer/news/244988

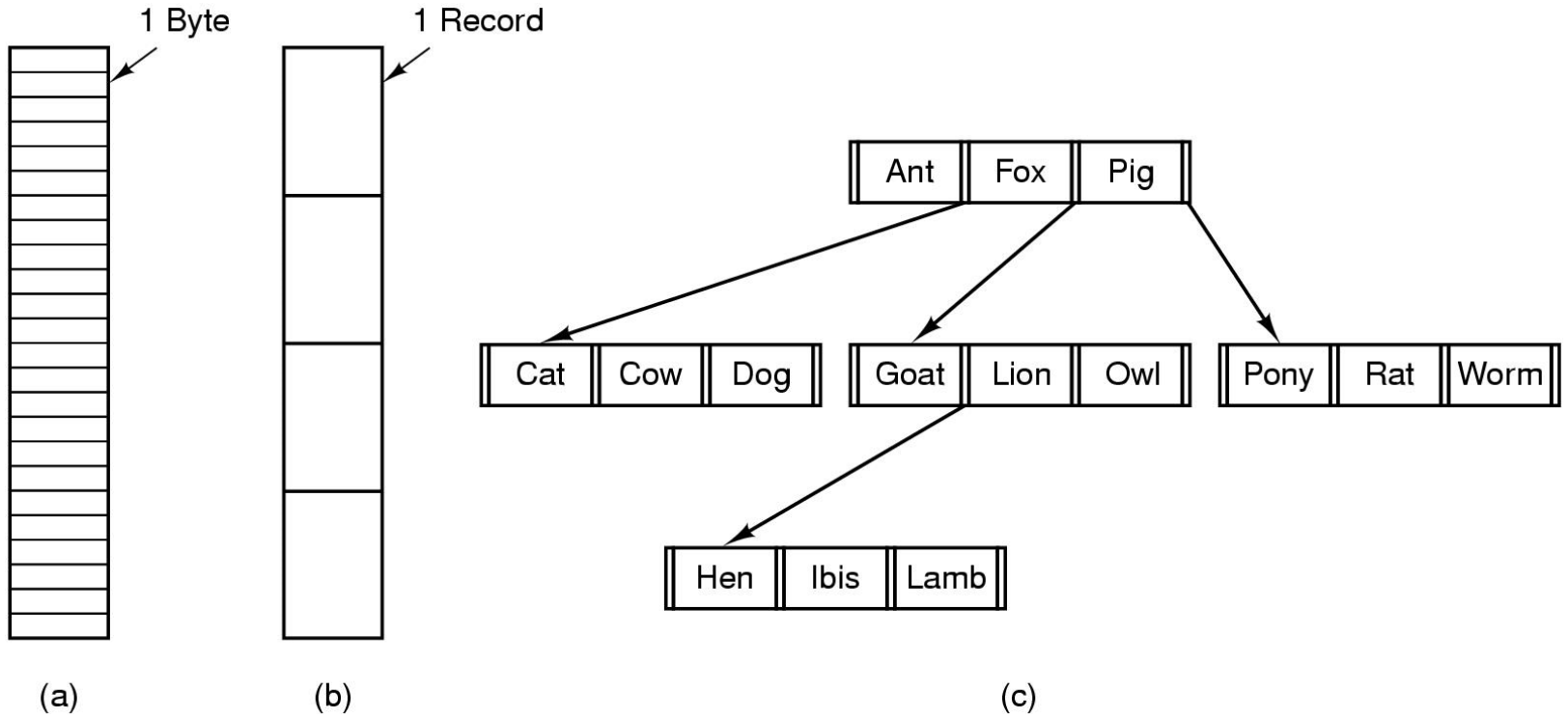
Description of Media File format





倒放过程很慢，难以连续倒放
帧：符合磁盘，随着时间推移而播放

Logical structure of file



←加速查找的树

关于文件性能的思考

- OS喜欢做顺序的预取以改进性能
  - 简单
  - 尊重磁盘的物理特性
- 数据文件的读取顺序是由文件格式决定的
  - 应用程序决定使用顺序
  - 格式的标准化以实现通用性
  - 合理使用内存，减少IO访问是程序员的主要优化目标
- 如果二者之间存在矛盾，如何解决?
  - 训练程序员
  - 提供更多的程序逻辑信息（fadvice, madvice)
  - Bypass OS（停止OS进行预取等行为）

文件系统的一些特殊问题

- 磁盘空间管理与容错
- 文件去重
- 多文件系统兼容

Disk space management

磁盘是有份额的

- How to allocate disk space?
  - Block/sector: base unit of disk space
  - Bitmap or list: empty blocks management
  - The size of sector and the data structure of empty blocks is very important for system performance
- How to distribute the content of file/directory?
  - Continuous and intersectant distribution
  - Fragment management
- How to improve data security and stability?
  - RAID(0~5): redundant data storage
  - Check the consistency of file system

Disk space allocation

物理内存切块，每块大小是4K

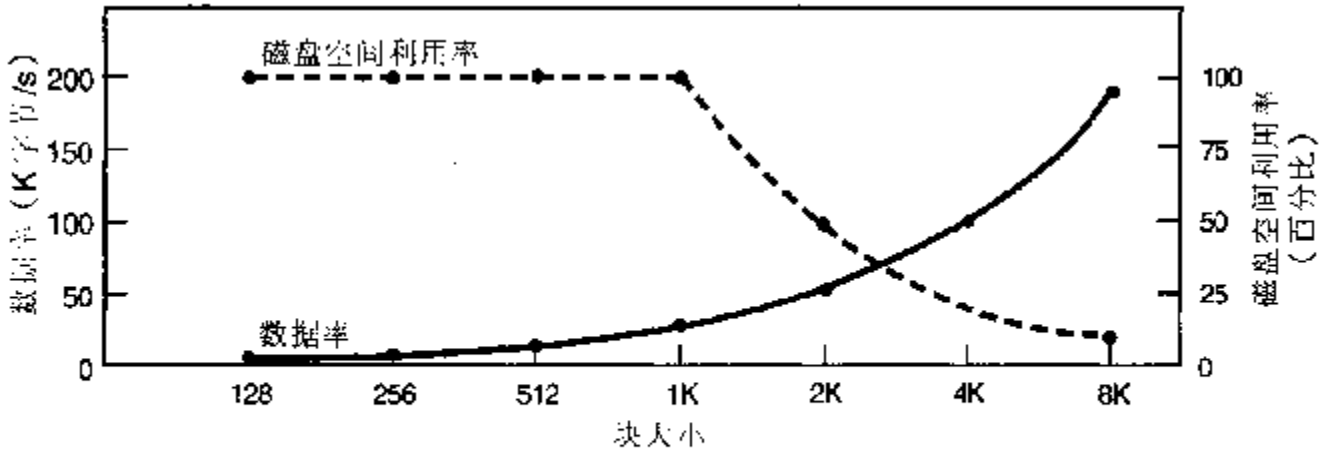
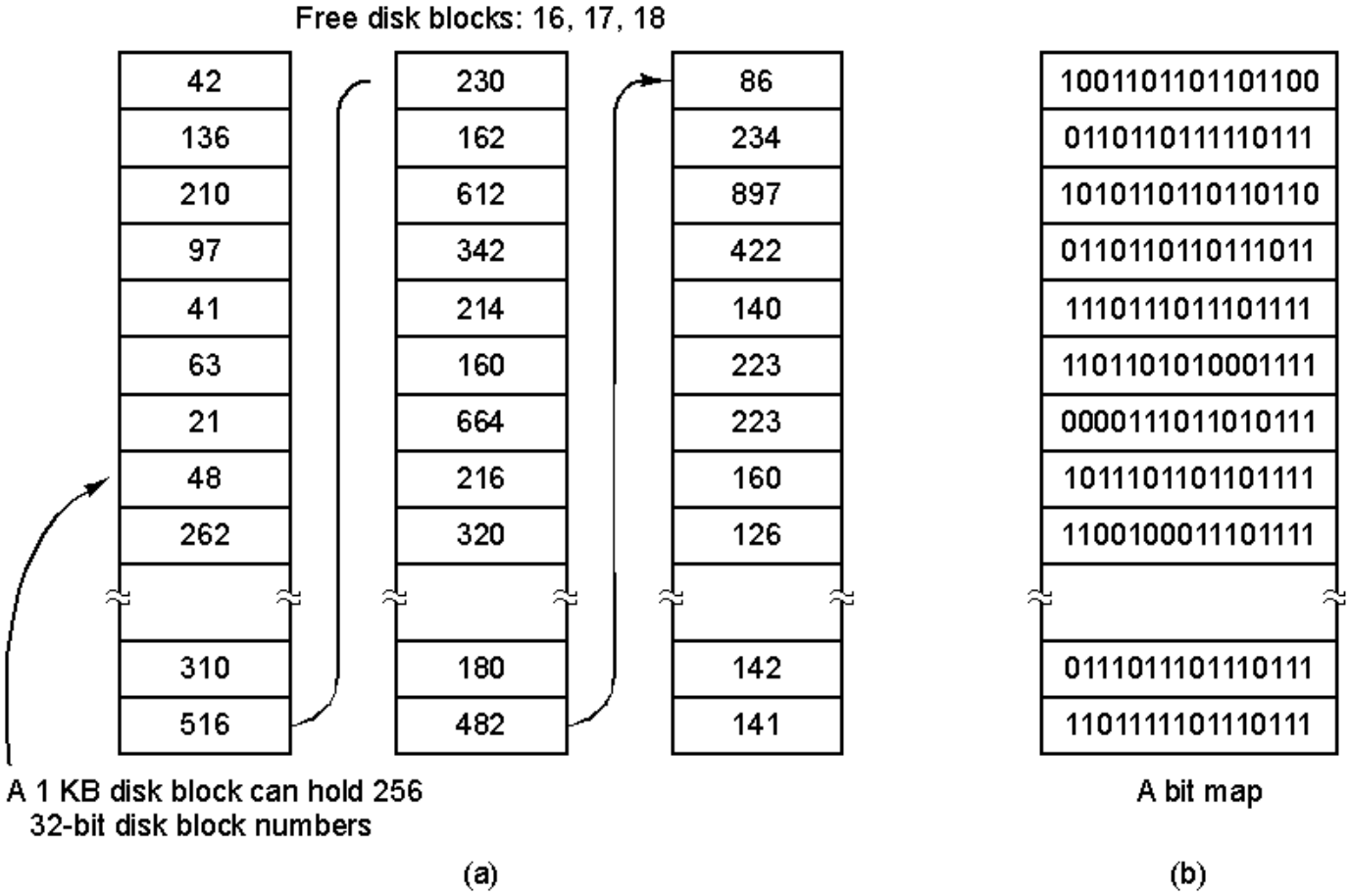


图 5-15　实线(左边标度)给出磁盘数据率，虚线(右边标度)给出磁盘空间效率，所有文件均为 1K

文件系统的**数据块越大**，单次传输时的磁盘控制动作就越简单，传输效率越高
但数据本身并不可能总是能够与数据块对齐，因此存储效率就越低

Empty disk blocks management

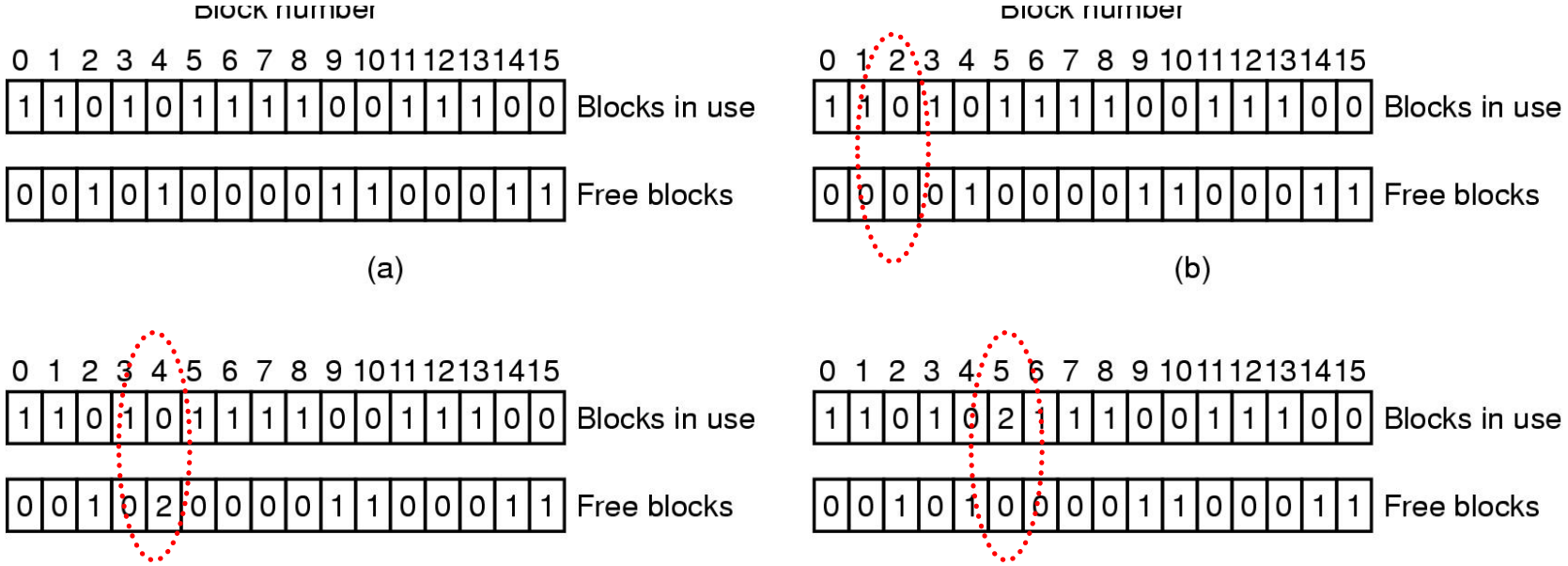
磁盘上的块与扇区渐渐无关，主要看单个分区越大，通常OS在上面创建的文件系统块越大，一个块对应若干扇区



与内存管理类似，可以将空闲的块号串成链表，也可以存成位图

b)文件系统的一致性检查

两张表：
一张表表示哪些块被使用
另一张表表示哪些块没有被使用
对应块相与为0



通过对磁盘块的检测，发现文件系统是否正确。

(a) 一致

(b) 块丢失----添加到空闲块中

(c) 空闲表中有重复块----重置空闲表

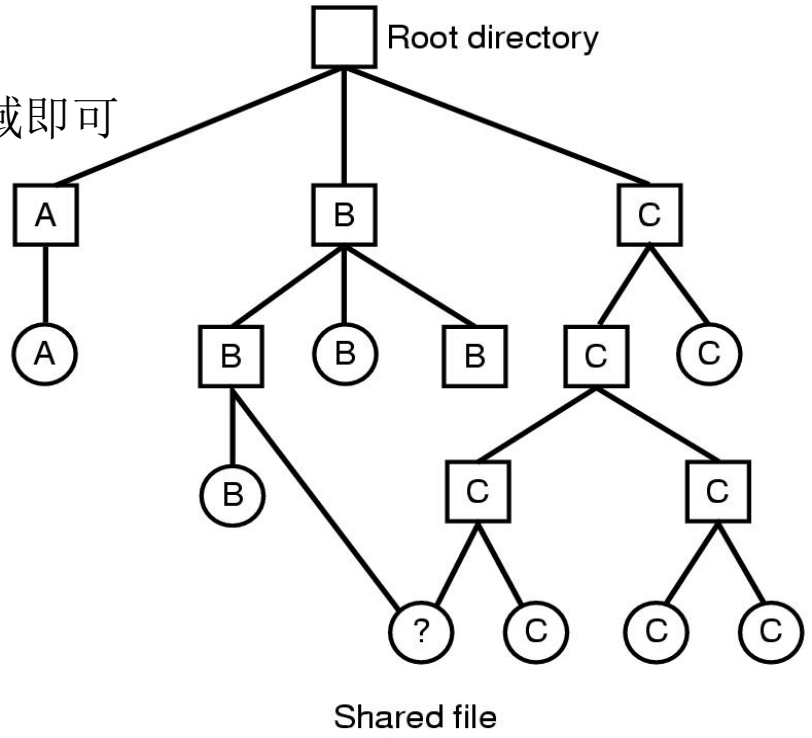
(d) 有重复的数据块----拷贝到一文件中，提交用户认可

注意：一旦出现检测结果不一致时，OS也无法保障完全正确的修复过程，有时候修复只能靠“猜”——宁愿多保留

5．共享文件管理实现

- 当文件系统中用链接实现共享时，系统目录树结构会发生变化：

一个文件取两个名
文件夹的链表指针指向同一个区域即可

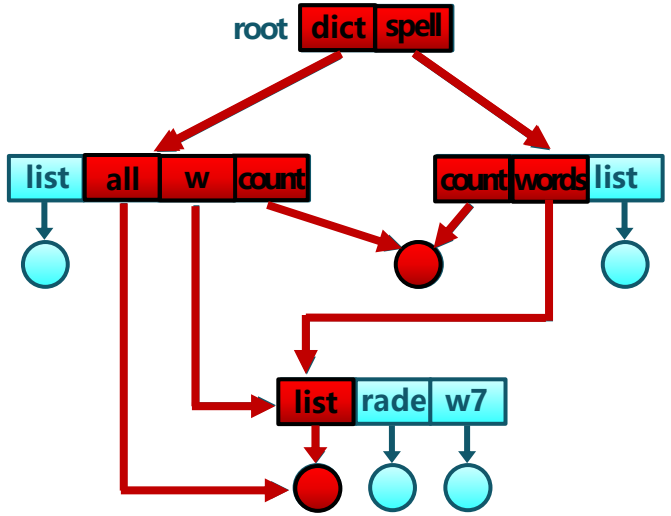


这时树变成了“有向无环图”，增加了文件系统复杂性.

文件别名

最后一个引用它的指针消失才能释放掉

- 两个或多个文件名关联同一个文件



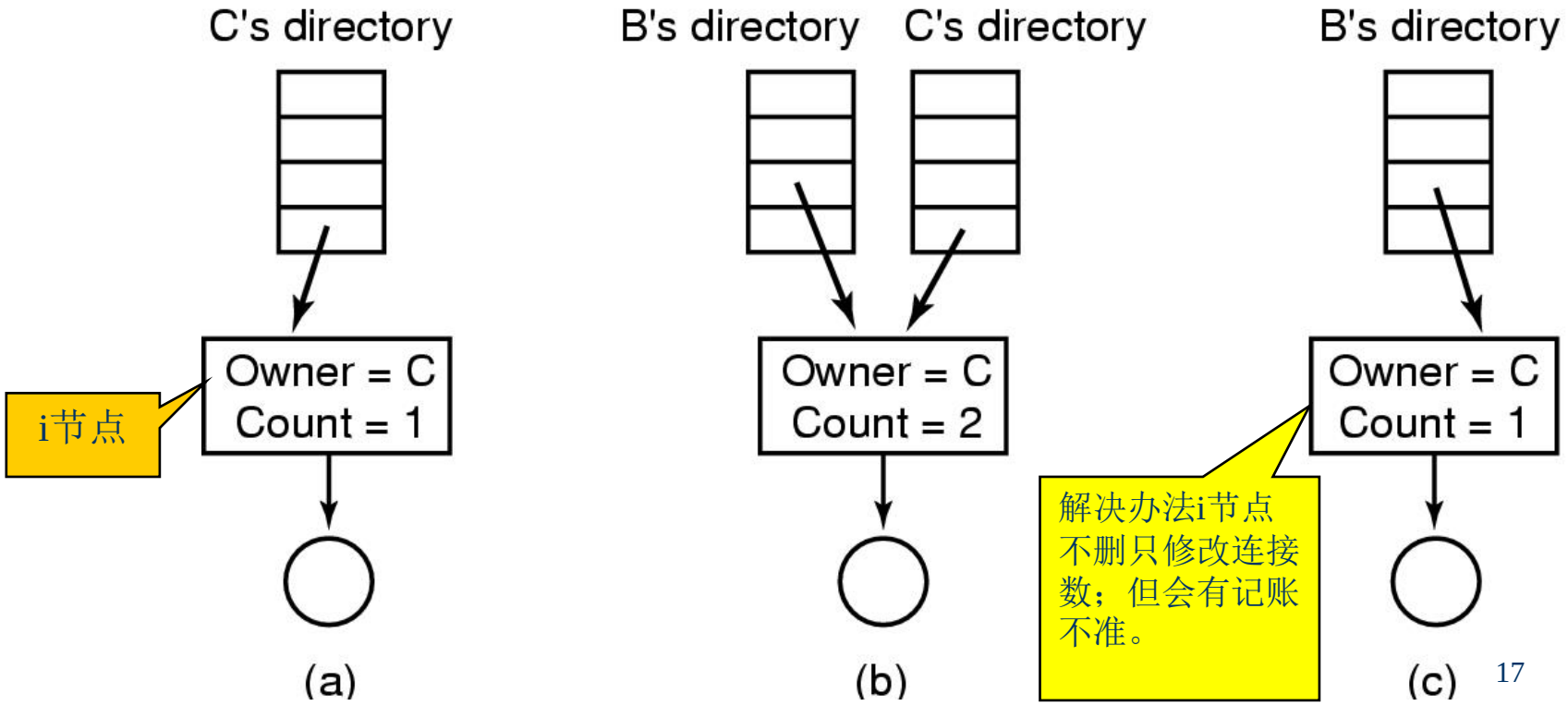

```
/dict/count
/spell/count

/dict/w/list
/dict/all
/spell/words/list
```

- 硬链接: 多个文件夹中存储指向同一个文件的“链表头”（树根）
- 软链接: 以“快捷方式”指向其他文件，快捷方式也是一个文件
  - 通过存储真实文件的逻辑名称来实现

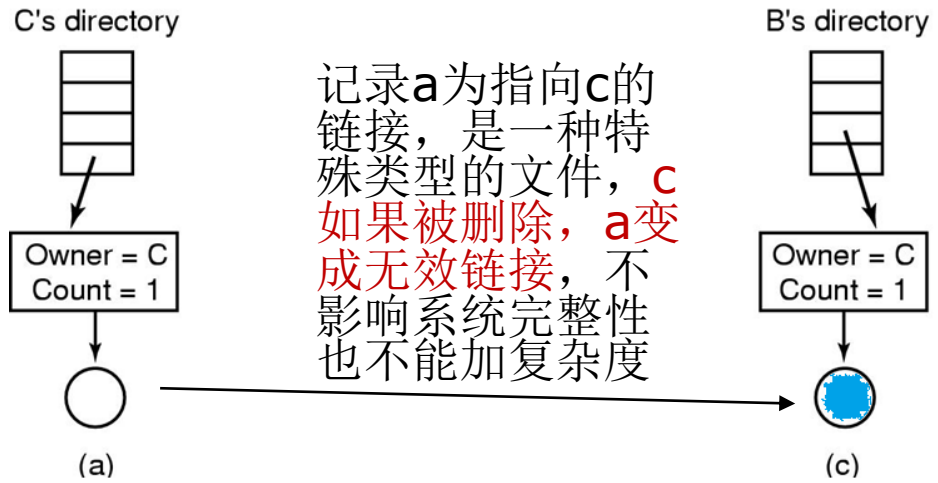
用链接实现文件共享控制（i节点方式）

- 每个文件有指向自己i节点的指针；
- 创建共享时多指针指向i节点并增加**引用数**；实现太麻烦
- 当属主文件被删除并连带i节点删除时会出错。

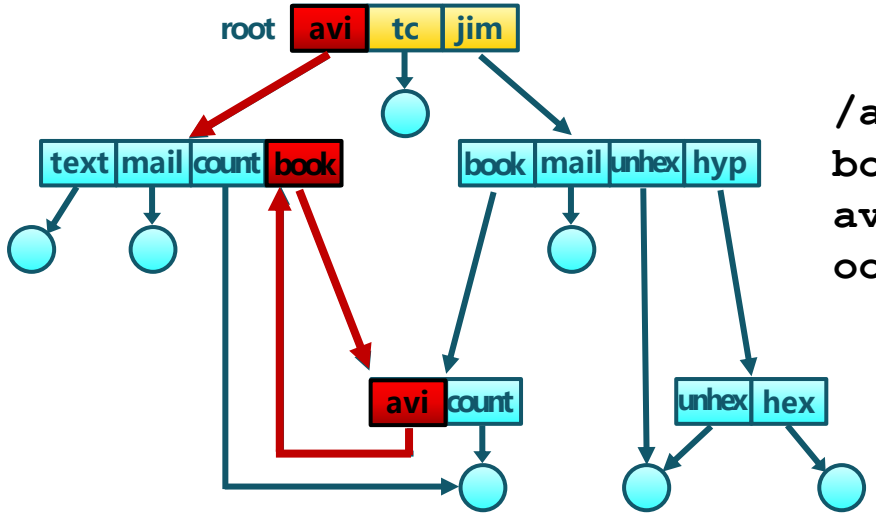


采用符号链接可较好解决问题

- 只有文件属主有指向i节点的指针
- 建立一个新文件类型link，用做目录管理
- 有共享需求的用户只了解找到共享文件的目录，而无指向i节点的指针
- **删除符号链接不会影响原文件**
- 若属主删除被链接的文件，只能带来**链接文件找不到正确目录文件**，无大错



文件目录中的循环



/avi/book/avi/book/avi/
book/avi/book/avi/book/
avi/book/avi/book/avi/b
ook/avi/...

- 如何保证没有循环？
  - 只允许到文件的链接，不允许在子目录的链接
  - 增加链接时，用循环检测算法确定是否合理
- 更多实践
  - 限制路径可遍历文件目录的数量

文件共享还需要访问类型和权限的支持

- 在具有共享机制文件系统中，要设置文件访问类型及文件权限，才可保证多个用户间的有效文件共享
- 文件访问类型：
  - 读read：可读出文件内容；
  - 写write（修改update或添加append）：可把数据写入文件；
  - 执行execute：可由系统读出文件内容，作为代码执行；
  - 删除delete：可删除文件；
  - 修改访问权限change protection：修改文件属主或访问权限

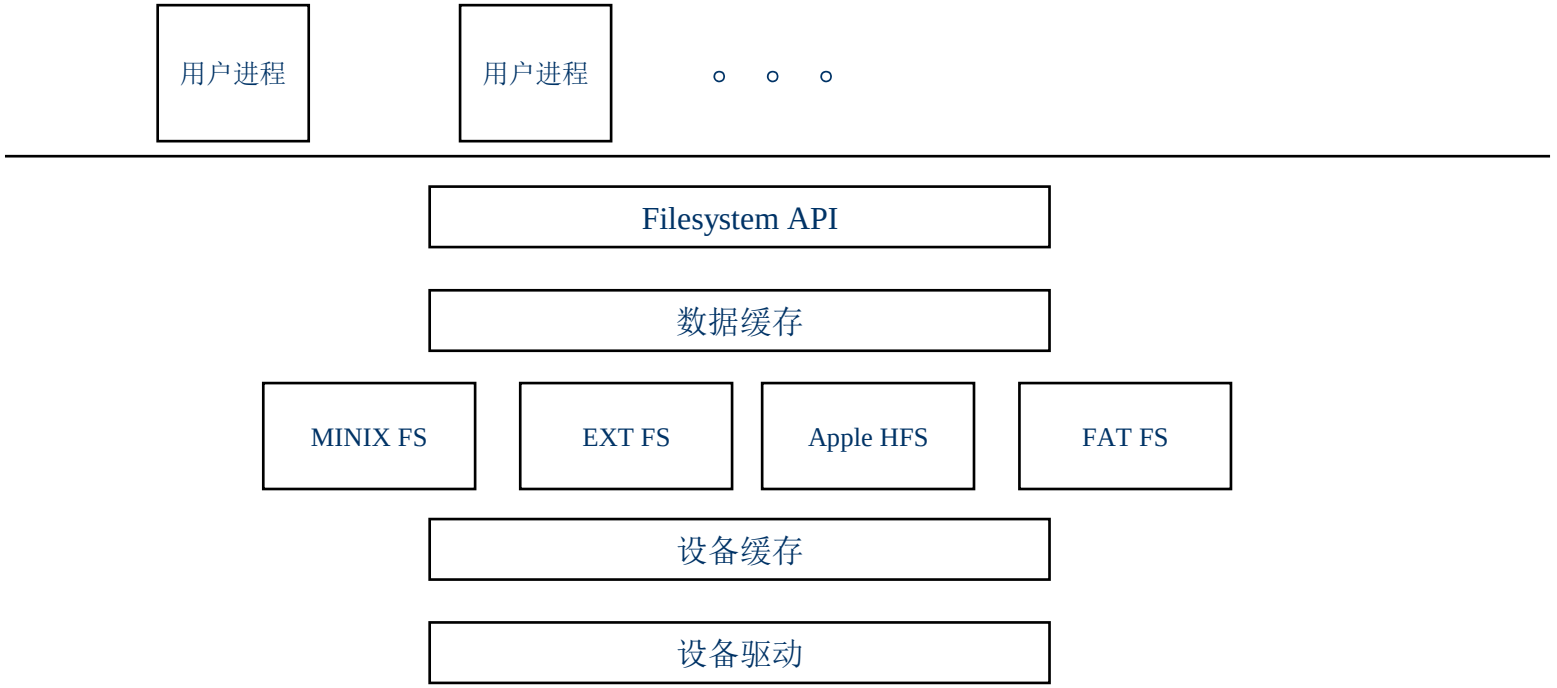
文件共享还需要访问类型和权限的支持

- 管理用户类型范围：
  - 指定用户
  - 用户组
  - 任意用户
- 访问类型和用户范围建立有机组合：
  - 建立访问矩阵：矩阵中一维是包含的目录和文件，另一维是用户，每个元素是允许的访问方式

	wang	liu	zhang	zhao
a.c	RWE	RE	E	R
b.c	RE	E	RE	RWE
d.c	R	R	E	E

有共享访问时查询该矩阵

多种文件并存的时候....



如果文件系统各具特色，是不是每个应用程序都需要为不同的文件系统各自开发？

如果文件系统中的机制大同小异，是不是可以在多个系统间共享代码？

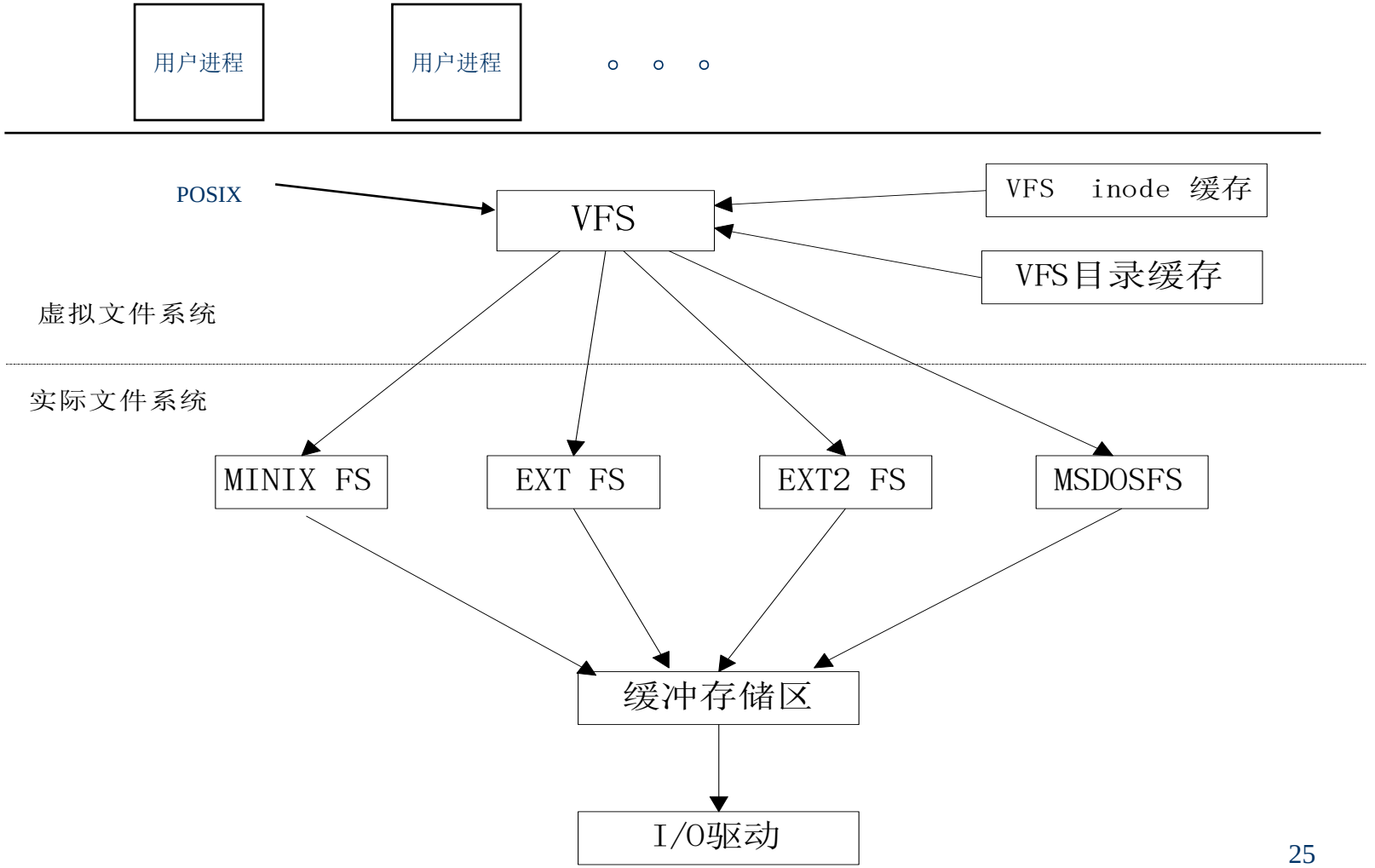
虚拟文件系统

- 为解决多种文件系统识别问题，提出了虚拟文件系统：
  - 在内存建立一个解释文件系统的抽象软件VFS
  - 用VFS建立物理设备与文件系统服务的接口
  - VFS对每个文件系统细节进行抽象，使不同的文件系统在系统内部被管理进程看成相似的文件系统
  - 系统启动时建立，系统关闭时消失

虚拟文件系统

- VFS主要完成功能:
  - 记录可用文件系统类型
  - 建立设备与文件系统的关联
  - 实现面向文件级的通用性操作
  - 将对特定文件系统的操作影射到物理文件系统中

VFS与实际文件系统的逻辑关系



openEuler中的VFS和文件系统

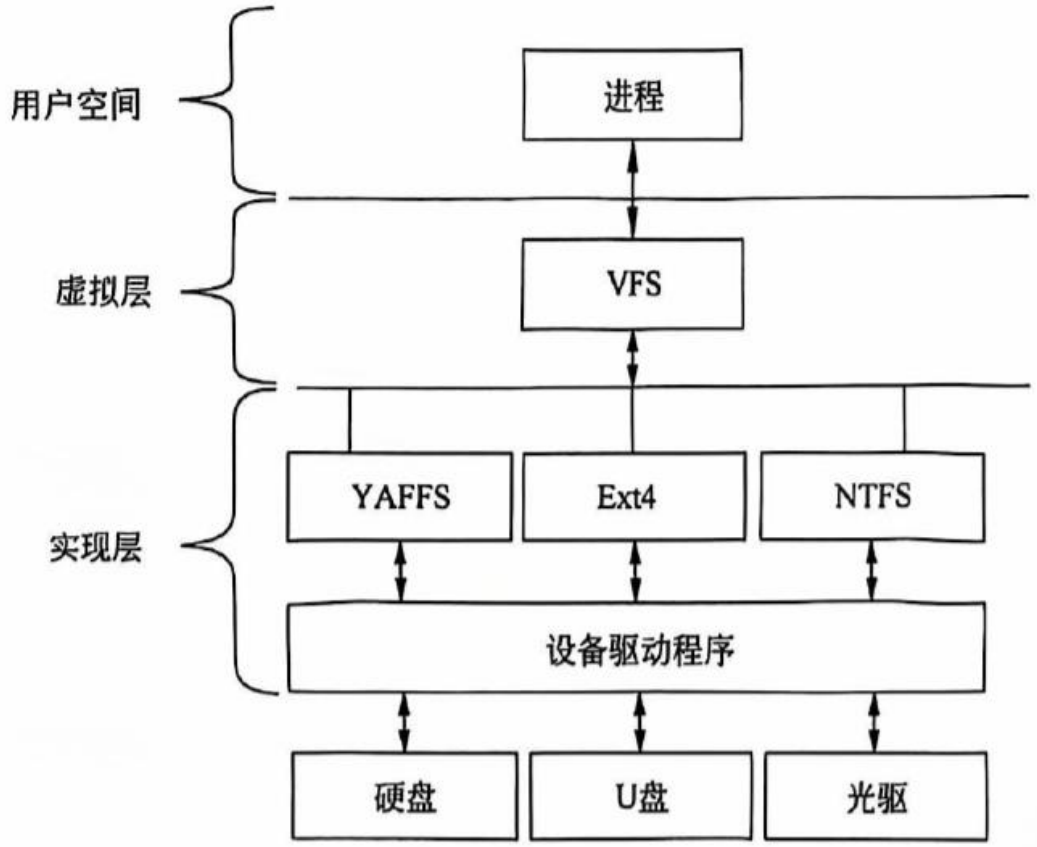


图 7-4　openEuler 中的文件系统架构

虚拟文件系统 (VFS)

- 目的
  - 对所有不同文件系统的抽象
- 功能
  - 提供相同的文件和文件系统**接口**
  - 管理所有文件和文件系统关联的**数据结构**
  - 高效查询**例程**，遍历文件系统
  - 与特定文件系统模块的**交互**

文件系统基本数据结构

- 文件卷控制块 (Unix: “superblock”)
  - 每个文件系统一个
  - 文件系统详细信息
  - 块、块大小、空余块、计数/指针等

文件系统基本数据结构

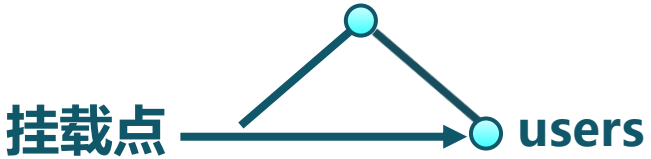
- 文件卷控制块 (Unix: “superblock”)
- 文件控制块(Unix: “vnode” or “inode”)
  - 每个文件一个
  - 文件详细信息
  - 访问权限、拥有者、大小、数据块位置等

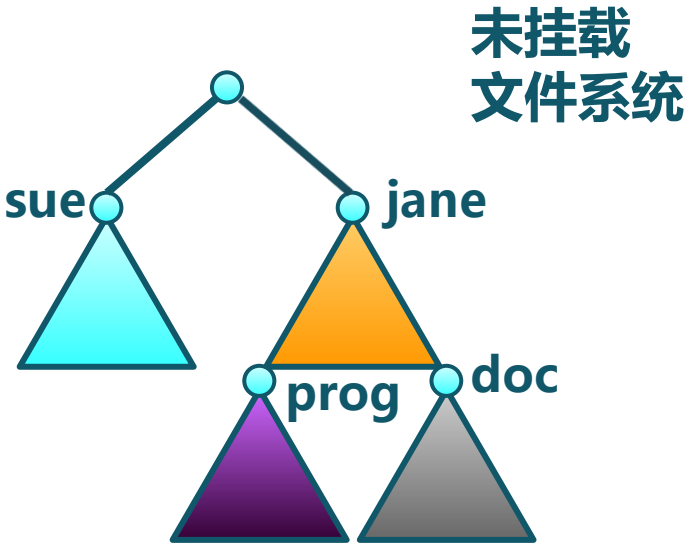
文件系统基本数据结构

- 文件卷控制块 (Unix: “superblock”)
- 文件控制块(Unix: “vnode” or “inode”)
- 目录项 (Linux: “dentry”)
  - 每个目录项一个(目录和文件)
  - 将目录项数据结构及树型布局编码成树型数据结构
  - 指向文件控制块、父目录、子目录等

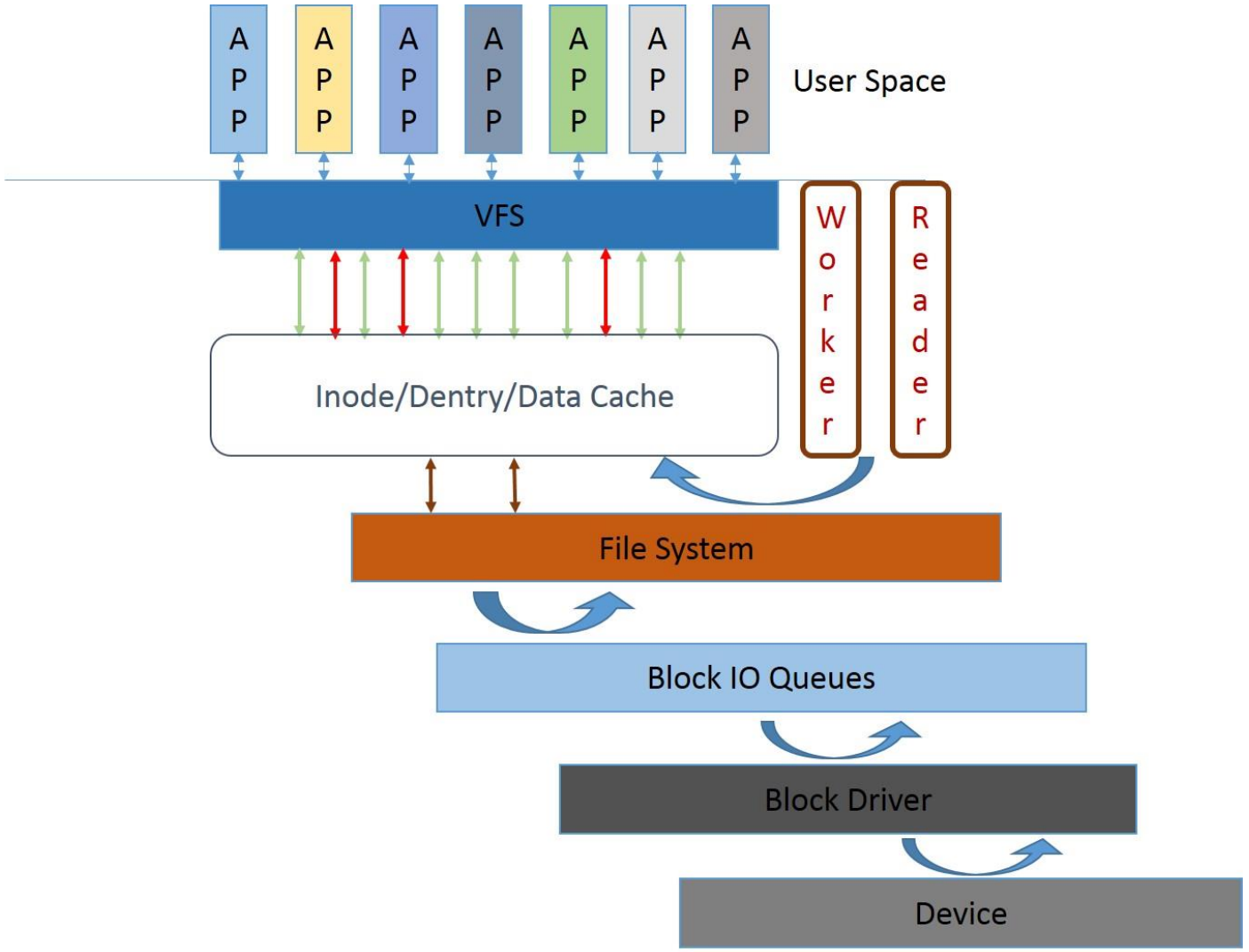
文件系统挂载

- 文件系统需要先挂载才能被访问
- 未挂载的文件系统被挂载在挂载点上





文件系统挂载



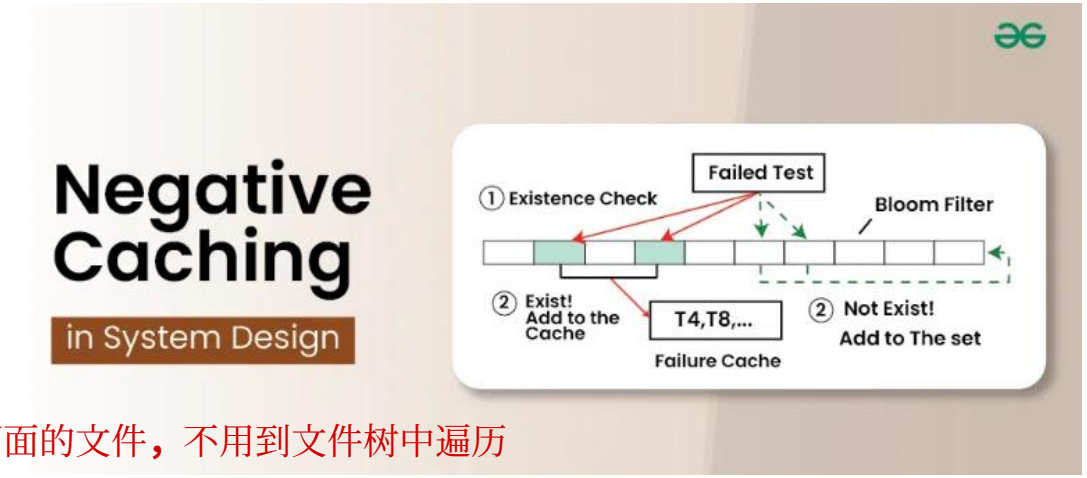
⑩https://msreekan.com/2015/04/24/linux-storage-cache/

VFS功能举例：名字解析（路径遍历）

- 名字解析: 把逻辑名字转换成物理资源（如文件）
  - 依据路径名，在文件系统中找到实际文件位置
  - 遍历文件目录直到找到目标文件
- 举例: 解析“/bin/ls”
  - 读取根目录的文件头 (在磁盘固定位置)
  - 读取根目录的数据块，搜索“bin”项
  - 读取bin的文件头
  - 读取bin的数据块; 搜索“ls”项
  - 读取ls的文件头
- 当前工作目录 (PWD) 现在的文件夹，更容易找到这个文件夹下面的文件，不用到文件树中遍历
  - 每个进程都会指向一个文件目录用于解析文件名
  - 允许用户指定相对路径来代替绝对路径
    如，用 PWD=“/bin” 能够解析 “ls”

要想知道文件不存在，要遍历完，确信该文件不存在也需要缓存

反向思考 ： 如何证明一个路径(文件)无法解析？



设计细节：

1. 负缓存，要存多少项呢？
2. 负缓存怎么跟文件操作同步？
3. 负缓存什么时候失效？

https://www.geeksforgeeks.org/negative-caching-system-design/

Thanks for your time!
Questions & Answers