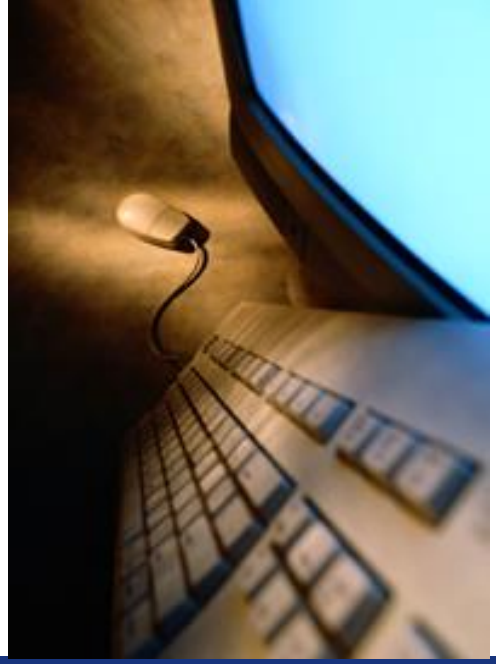
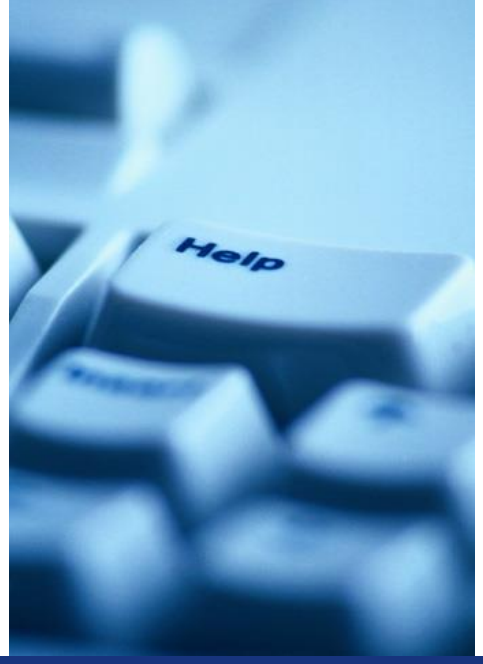
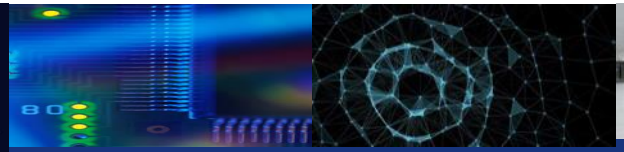



操作系统原理

Operating Systems Principles

张青
计算机学院

第十四讲 — 文件系统实现



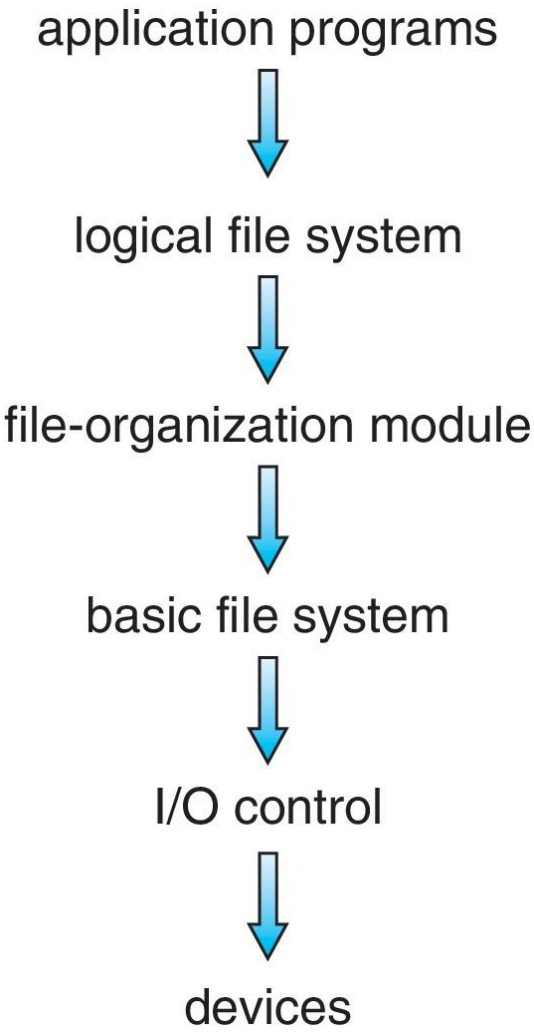
目标

- 本地文件系统和目录结构的实现细节；
- 远程文件系统；
- 块分配与空闲块的算法和权衡；

文件系统结构

- 磁盘提供大多数的外存，以便维护文件系统
- 为了提高I/O效率，内存和磁盘之间的I/O传输以块为单位进，每个块具有一个或多个扇区，扇区大小从32字节到4096字节不等
- 文件系统提供高效和便捷的磁盘访问。Linux标准文件系统是可扩展文件系统，如ext3和ext4;Unix文件系统是UFS，Windowsde的文件系统是NTFS
- 文件系统由许多不同的层组成
  - I/O控制层包括设备驱动程序和中断处理程序，以在主内存和磁盘系统之间传输信息
  - 基本文件系统只需向适当设备驱动程序发送通用命令，以读取和写入磁盘的物理块
  - 文件组织模块将逻辑块地址转成物理块地址，以供基本文件系统传输
  - 逻辑文件系统管理元数据信息。元数据包括文件系统的所有结构，而不包括实际数据

分层设计的文件系统


application programs
logical file system
file-organization module
basic file system
I/O control
devices

文件系统实现

- 操作系统实现了系统调用open()和close()，以便进程可以请求访问文件内容，这里深入介绍实现文件系统的结构和操作
- 文件系统的实现需要采用多个磁盘和内存的结构
- 文件系统一般包含如下信息：如何启动存储的操作系统、总的块数、空闲块的数量和位置、目录结构以及具体文件
  - 引导控制块：包含从该卷引导操作系统的所需信息。如果磁盘不包含操作系统，则这块的内容为空；UFS中称之为引导块；NTFS中称为分区引导扇区
  - 卷控制块：包含卷或分区的详细信息，如分区的块的数量、块的大小、空闲块的数量和指针、空闲的PCB数量和FCB指针等
  - 目录结构用于组织文件；UFS中它包含文件名和相关inode的号码；在NTFS中，它存储在主控文件表中
  - 每个文件的文件控制块（FCB）包含该文件的许多详细信息

文件系统实现

- 内存中的信息用于管理文件系统并通过缓存来提供性能
  - 内存中的安装表（mount table）包含每个安装卷的有关信息
  - 内存中的目录结构的缓存含有最近访问目录的信息
  - 整个系统的打开文件表（system-wide open-file table）包括每个打开文件的FCB的副本以及其他信息
  - 每个进程的打开文件表（per-process open-file table）包括一个指向整个系统的打开文件表中的适当条目的指针，以及其他信息
  - 当对磁盘读出或写入时，缓冲区保存文件系统的块
  - 创建新文件时，应用程序调用逻辑文件系统。逻辑文件系统知道目录结构的格式；它会分配一个新的FCB;然后，系统将相应的目录读到内存，使用新的文件名和FCB进行更新，并将它写回到磁盘
  - 打开文件表的条目有多种名称；Unix称之为文件描述符（file descriptor）,Windows称其为文件句柄（file handle）

文件控制块（FCB）

- **OS maintains FCB per file, which contains many details about the file，包含唯一标识号**
  - Typically, inode number, permissions, size, dates
  - Example

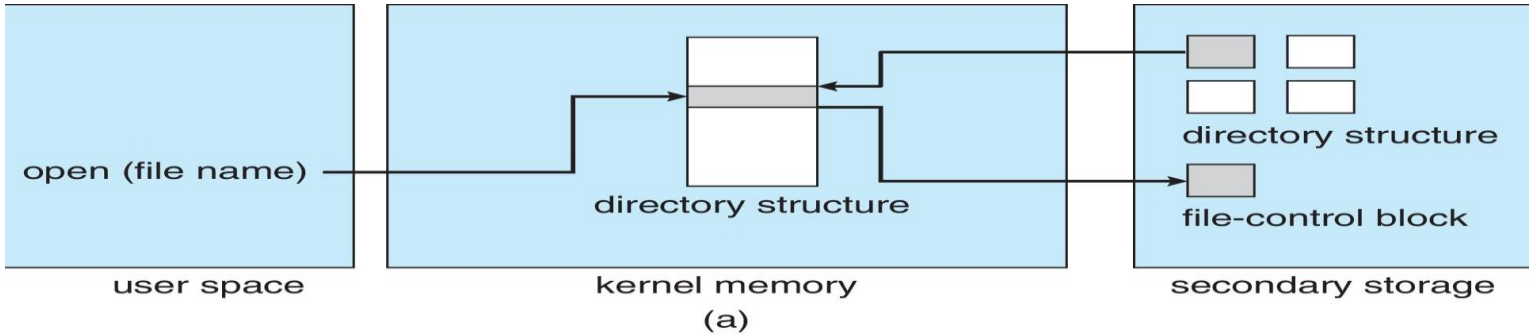
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

内存中的文件结构

- Mount table （安装表、挂载表）storing file system mounts, mount points, file system types
- System-wide open-file table contains a copy of the FCB of each file and other info
- Per-process open-file table contains pointers to appropriate entries in system-wide open-file table as well as other info

内存中的文件结构

- (a) refers to opening a file
- (b) refers to reading a file



Windows: 文件句柄；
Unix：文件描述符

分区与安装

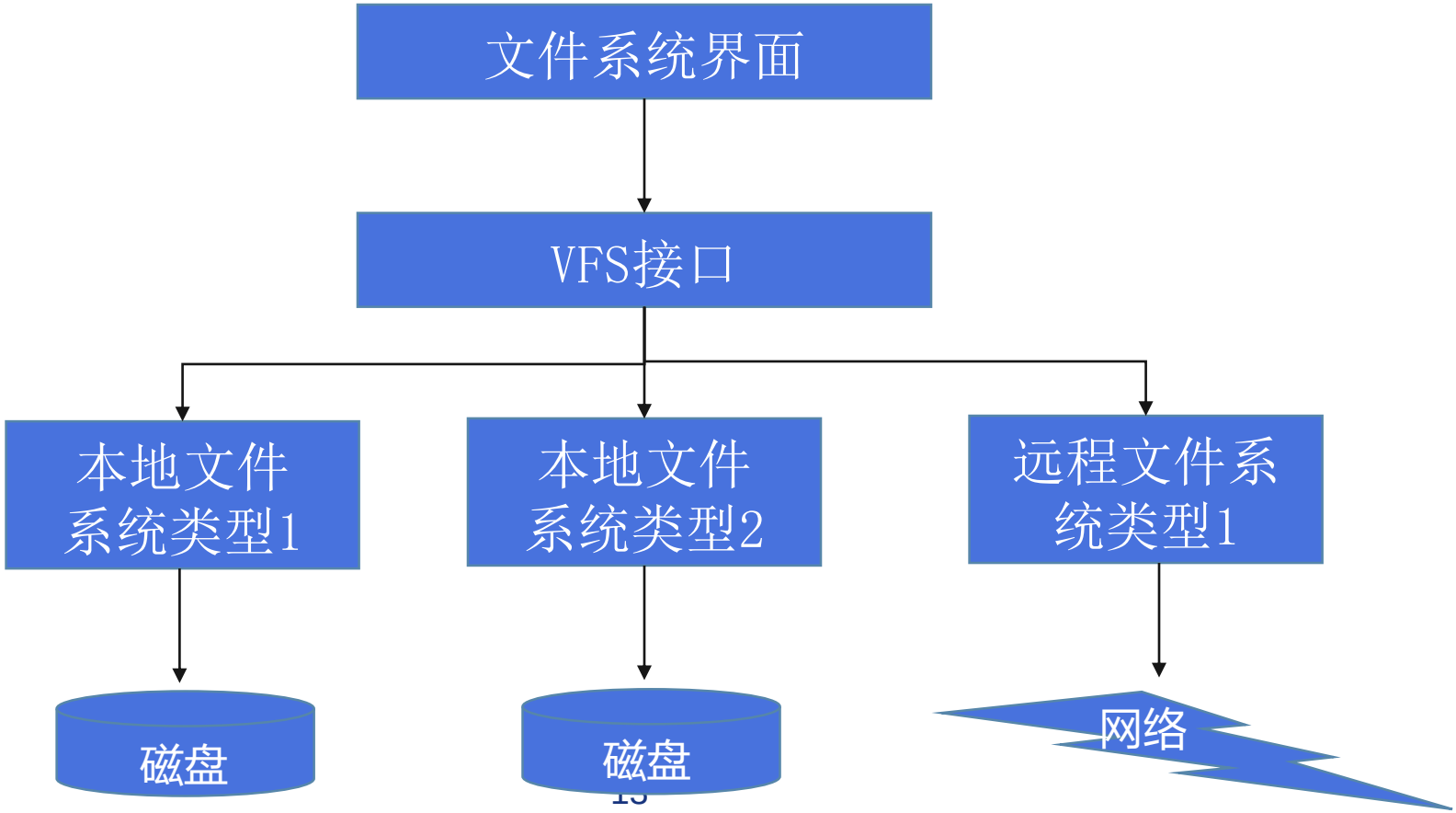
- 磁盘布局可以有多种，具体取决于操作系统
- 一个磁盘可以分成多个分区，一个卷可以跨越多个磁盘的分区
- 分区可以是原生的，没有任何文件系统；也可以是含有文件系统；当没有合适的文件系统时，可使用原始磁盘
- 引导信息可以存储在各自分区中。磁盘可以有多个分区，每个可以包含不同类型的文件系统和不同的操作系统
- 根分区（root partition），包括操作系统内核和其他系统文件，在启动时安装
- Windows每个卷安装在分开的名称空间中，用一个字母和一个冒号表示，如C:以及F:
- Unix可以将文件系统安装在任何目录上。安装的实现，采用在目录inode的内存副本上加上一个标志，指示为安装点

虚拟文件系统

- 操作系统如何将多个类型的文件系统集成到目录结构中？
- 用户如何在访问文件系统空间时，无缝地在文件系统类型之间迁移？
- 实现多种文件系统的简单但欠佳的方法是，为每种类型编写目录和文件程序
- 大多数操作系统，包括Unix，采用面向对象的技术来简化、组织和模块化实现
- 数据结构和程序用于隔离基本系统调用的功能和实现细节。文件系统的实现由三个主要的层组成
  - 第一层为文件系统接口
  - 第二层为虚拟文件系统
  - 第三层为实现文件系统或远程文件系统协议层

虚拟文件系统

- 文件系统接口基于open()、read()、write()和close调用及文件描述符



虚拟文件系统

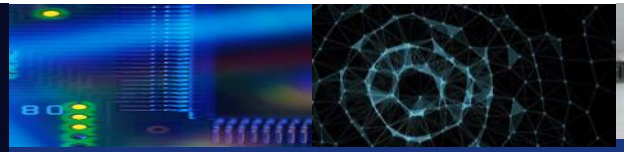
- 虚拟文件系统层（VFS）提供两个重要功能：
  - 通过定义一个清晰的VFS接口，它将文件系统的通用操作和实现分开。VFS接口的多个实现可以共存在同一台机器上，允许透明访问本地安装的不同类型的文件系统
  - 它提供一种机制，以唯一表示网络上的文件。VFS基于文件成为虚拟节点或v节点（vnode）的文件表示结构，它包含一个数字指示符以唯一表示网络上的一个文件。这种网络的唯一性需要用来支持网络文件系统。内核为每个活动节点（文件或目录）保存一个vnode结构
- VFS区分本地文件和远程文件
- VFS根据文件系统类型调用特定文件类型的操作以便处理本地请求，通过调用NFS协议程序来处理远程请求

虚拟文件系统

- Linux VFS定义4种主要对象类型：
  - 索引节点对象：表示一个单独的文件
  - 文件对象：表示一个已打开的文件
  - 超级块对象：表示整个文件系统
  - 目录条目对象：表示单个目录条目
- 对以上4种对象类型，VFS定义了一组可以进行的操作。文件对象操作的一些API包括：
  - ✓ int open(…) — 打开一个文件
  - ✓ int close(…) — 关闭一个已打开的文件
  - ✓ ssize_t read(…) — 读文件
  - ✓ ssize_t write(…) — 写文件
  - ✓ int mmap(…) — 内存映射一个文件

目录实现

- **Linear list of file names with pointer to the data blocks**
  - Simple to program
  - Time-consuming to execute
    - Linear search time
    - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table – linear list with hash data structure**
  - Decreases directory search time
  - **Collisions** – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method



分配方法

- **An allocation method refers to how disk blocks are allocated for files:**
  - Contiguous
  - Linked
  - File Allocation Table (FAT)

连续分配方法

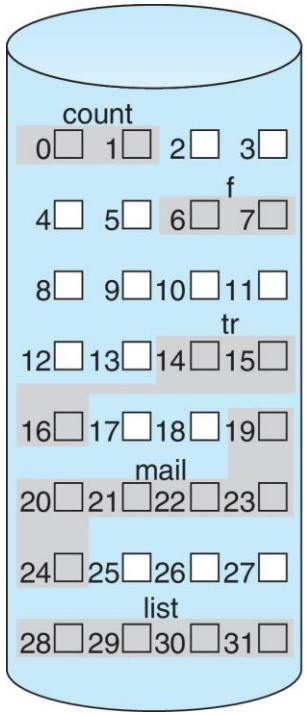
- **An allocation method refers to how disk blocks are allocated for files:**
- **Each file occupies set of contiguous blocks**
  - Best performance in most cases (寻道数量最小，寻道时间最小)
  - Simple – only starting location (block #) and length (number of blocks) are required
  - 支持顺序访问和直接访问；
  - Problems include:
    - Finding space on the disk for a new file,
    - Knowing file size,
    - External fragmentation, need for **compaction(合并) off-line** (**downtime**) or **on-line**

连续分配方法

- **Mapping from logical to physical (block size =512 bytes)**

LA/512 → Q, R

- **Block to be accessed = starting address + Q**
- **Displacement into block = R**

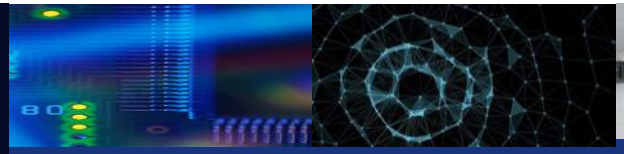


directory

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

基于扩展的系统

- **Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme — 最初分配一块连续空间，之后当数量不够时，会添加另一块连续空间（称为扩展-extend）**
- **Extent-based file systems allocate disk blocks in extents**
- **An extent is a contiguous block of disks**
  - Extents are allocated for file allocation
  - A file consists of one or more extents
  - 文件块的位置记录为：地址、块数、下一个扩展的首块的指针；
  - 有些系统上，文件所有者可以设置扩展大小；但如果设置太大会造成内部碎片

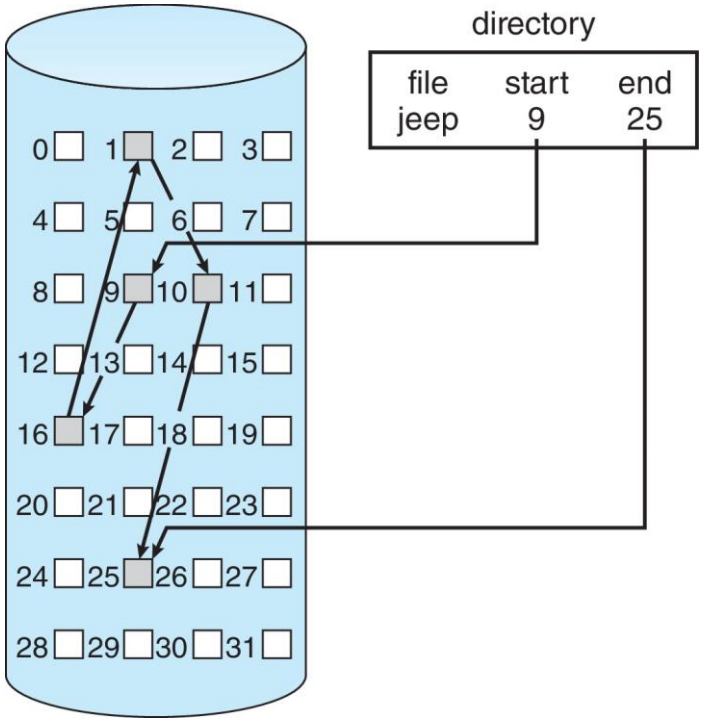


链接分配

- Each file is a linked list of blocks
- File ends at nil pointer
- No external fragmentation
- Each block contains pointer to next block
- No compaction, external fragmentation
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks

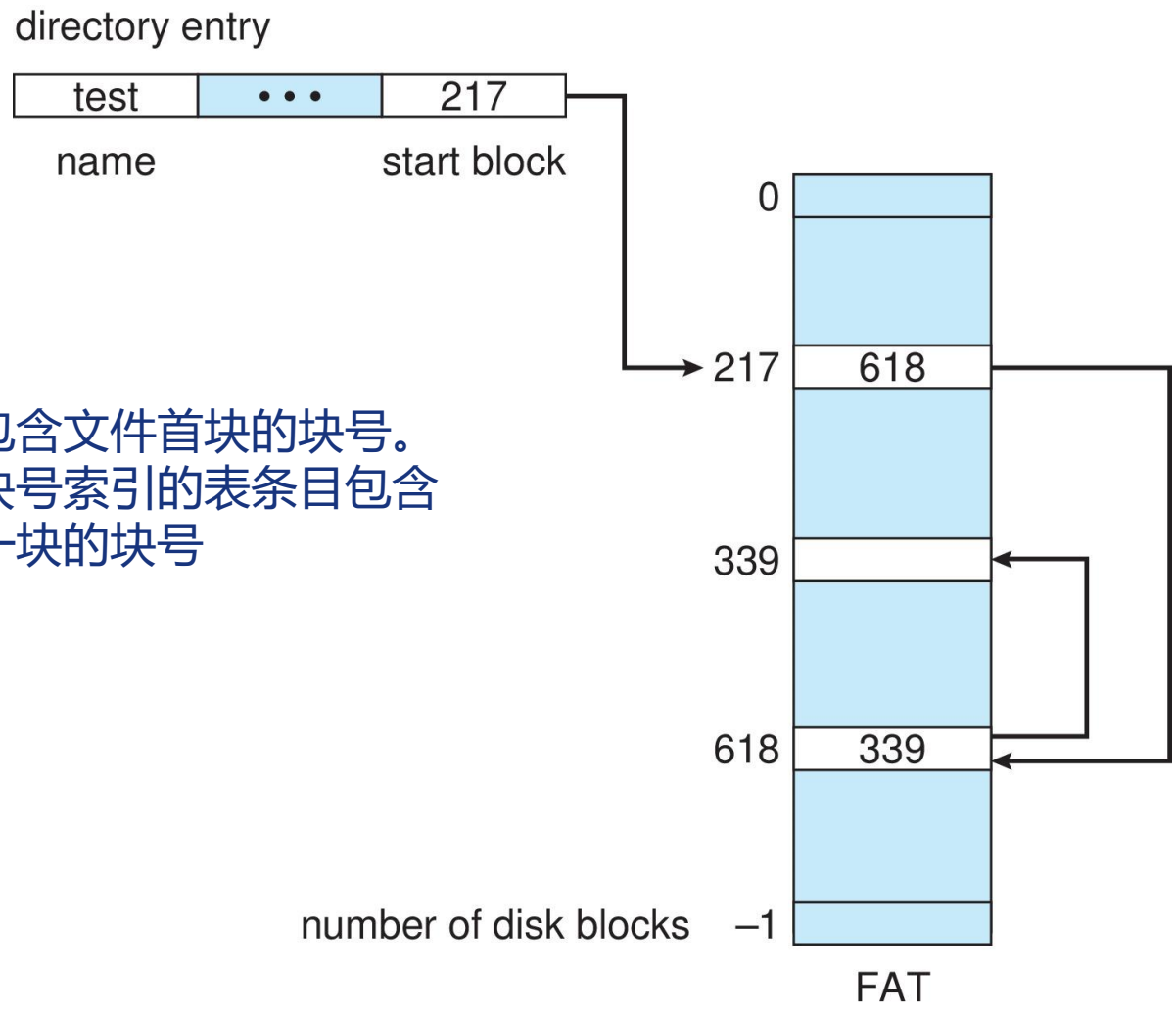
链接分配

- 解决了连续分配的所有问题（）——每个文件是磁盘块的链表；磁盘块可能会散布在磁盘的任何地方
- 缺点：只能有效用于顺序访问文件，不支持直接访问；需要为指针分配额外空间



链接分配的重要变种是文件分配表（file-allocation table）,用于MS-DOS

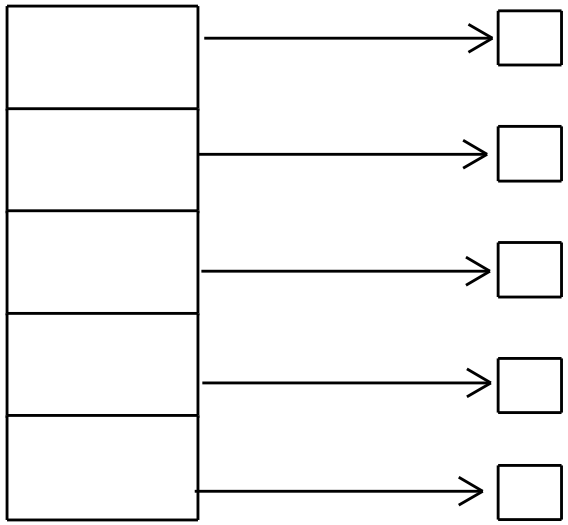
文件分配表



目录条目包含文件首块的块号。
通过这个块号索引的表条目包含
文件的下一块的块号

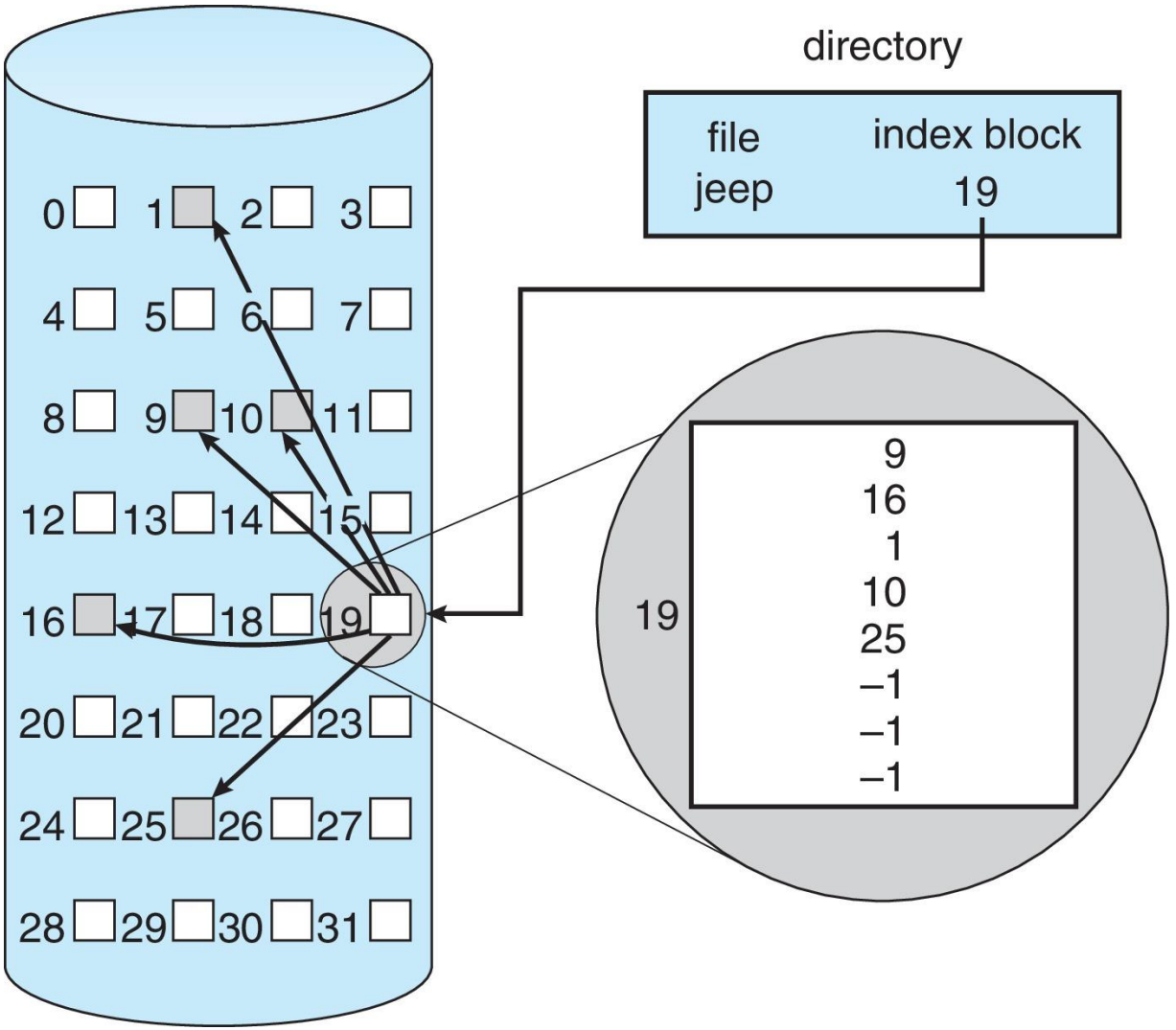
索引分配

- 索引分配解决了连续分配的外部碎片和大小声明的问题
- 在没有FAT时，链接分配不能支持高效的直接访问，因为块指针与块一起分散在整个磁盘上，并且必须按序读取
- 索引分配将所有指针放在一起，即索引块。每个文件都有自己的索引块



index table

磁盘空间的索引分配

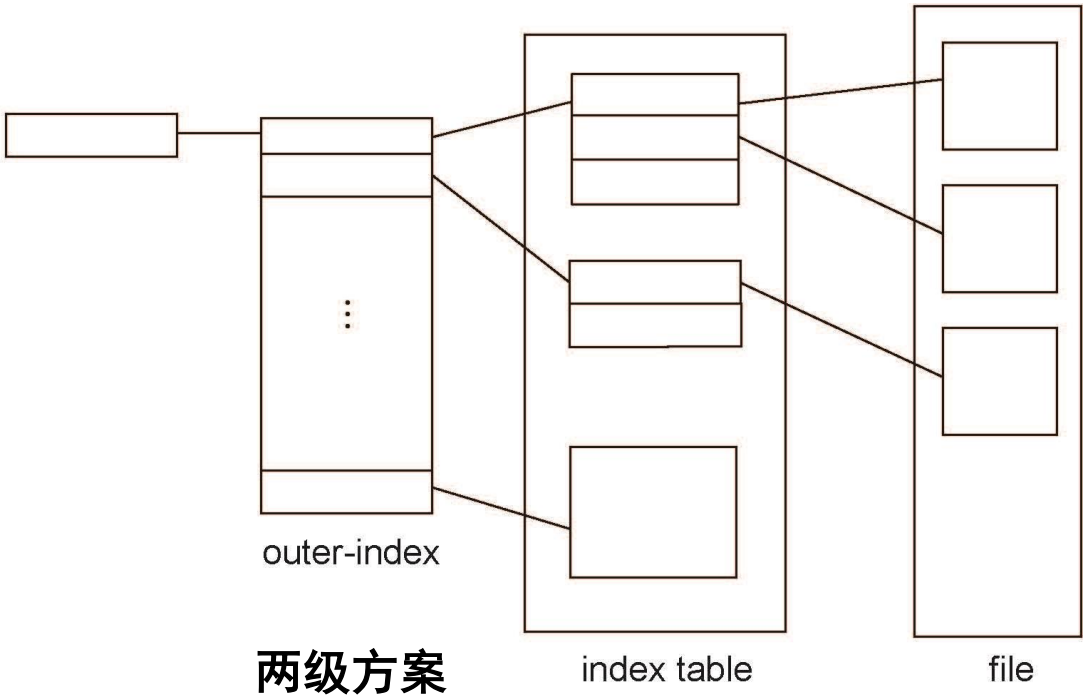

directory
file
jeep
index block
19
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31
19
9
16
1
10
25
−1
−1
−1

索引分配

- 每个文件都有一个索引块，因此索引块应尽可能小。然而，如果太小，它不能为大的文件存储足够多的指针。
- 处理这个问题的机制包括：
  - **链接方案**：一个索引块通常为一个磁盘块。因此，它本身不能直接读写。为了支持大的文件，可以将多个索引块链接起来。例如，一个索引块可以包括一个含有文件名的头部和一组头100个磁盘块的地址。下一个地址（索引块的最后一个字）为null，或另一个索引块的指针（对于大文件）
  - **多级索引**：链接表示的一种变种，通过第一集索引块指向一组第二级的索引块，它又指向文件块。当访问一块时，操作系统通过第一级索引查找第二级索引块，再采用这个块查找所需的数据块。这种做法可以持续到第三级或第四级，具体取决于最大文件大小。对于4096字节的块，可以在索引块中存入1024个4字节的指针。两级索引支持1048576的数据块和4GB的最大文件

索引分配

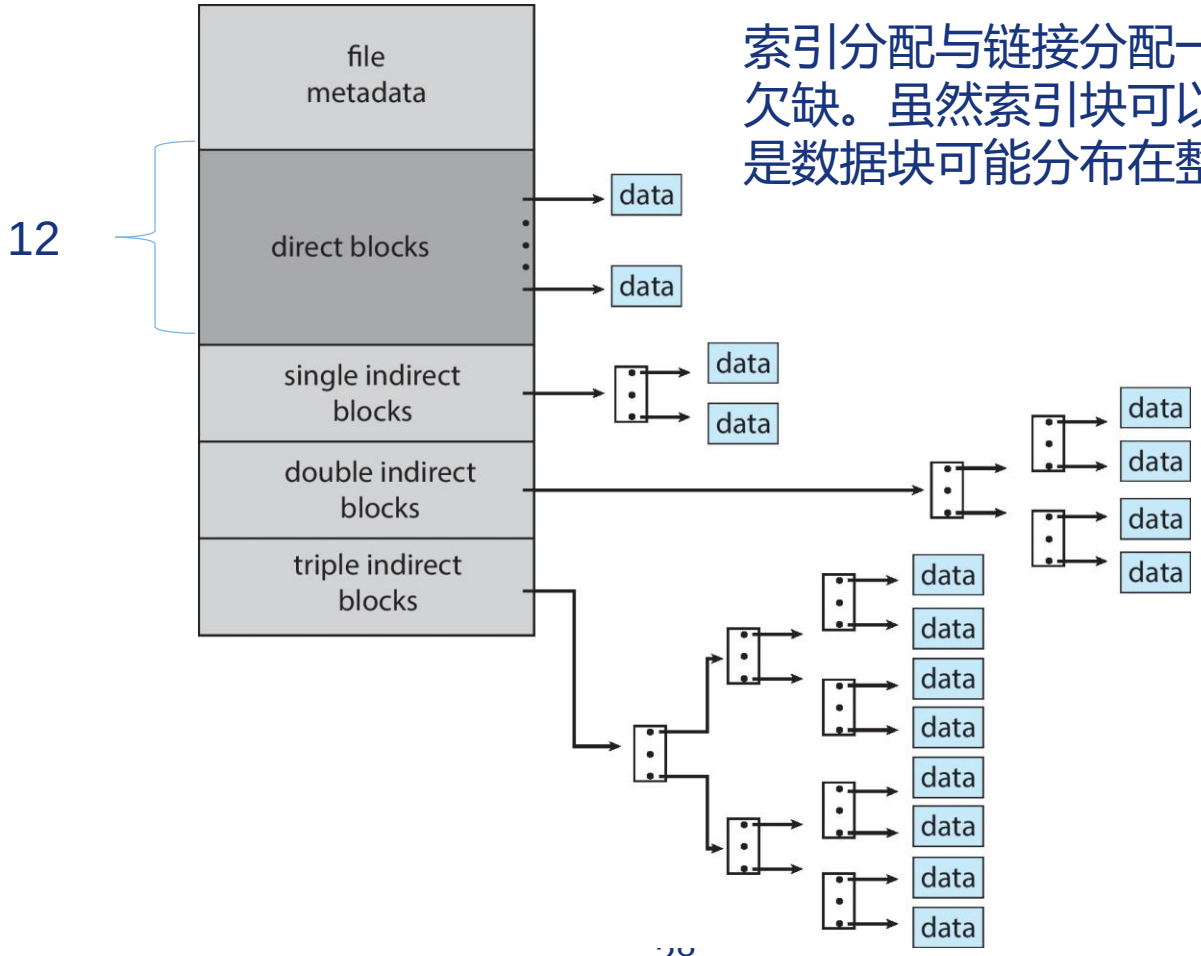
- **组合方案**：将索引块的前几个指针存在文件的inode中。这些指针的前12个指向直接块，即它们包含存储文件数据的块的地址。因此，小的文件（不超过12块）不需要单独的索引块。如果块大小为4KB，则不超过48KB的数据可以直接访问。接下来的3个指针指向3个间接块，间接指向包含数据块的地址



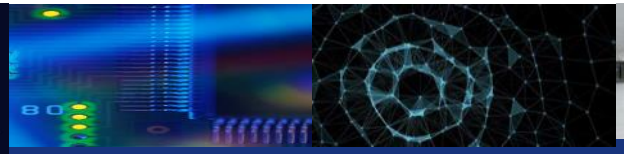
两级方案

索引分配——组合方案

- 1个文件的块数可以超过操作系统所用的4字节的文件指针所能访问的空间。32位指针只能访问2^32字节，即4GB



索引分配与链接分配一样在性能方面有所欠缺。虽然索引块可以缓存在内存中，但是数据块可能分布在整个卷上

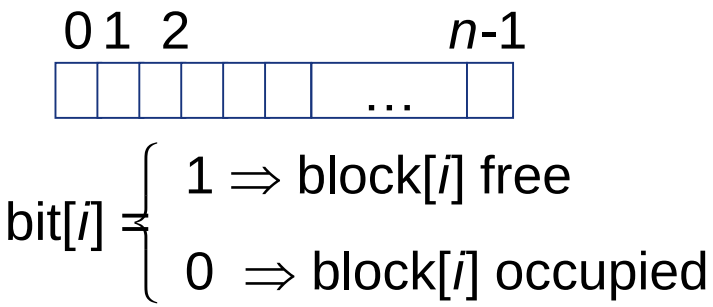


性能

- **最佳方法取决于文件访问类型**
  - 连续序列和随机序列
- **链接适用于顺序，而不是随机**
- **在创建时声明访问类型**
  - 选择连续分配或链接分配；
- **索引分配更复杂**
  - 单块访问可能需要读取2个索引块（二级索引），然后读取数据块；
  - 性能取决于：索引的结构、文件的大小以及所需块的位置；
- **对于NVM，没有磁盘头，因此需要不同的算法和优化**
  - 使用旧算法会占用大量CPU周期，试图避免不存在的头部移动；
  - 目标是减少CPU周期和I/O所需的总体路径；

空闲空间管理

- **File system maintains free-space list to track available blocks/clusters**
  - (Using term “block” for simplicity)
- **Bit vector or bit map （*n* blocks)**



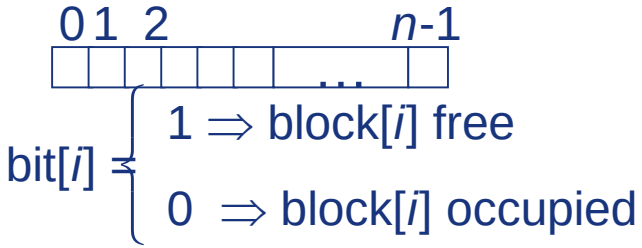
第一个空闲块号码计算=

(每个字的位数) *(值为0的字数) +第一个值为1的位的偏移

CPUs have instructions to return offset within word of first “1” bit

空闲空间管理

- File system maintains free-space list to track available blocks
- Bit vector or bit map (*n* blocks)



- Bit map requires extra space
  - Example:

    block size = 4KB = 2^{12} bytes

    disk size = 2^{40} bytes (1 terabyte)

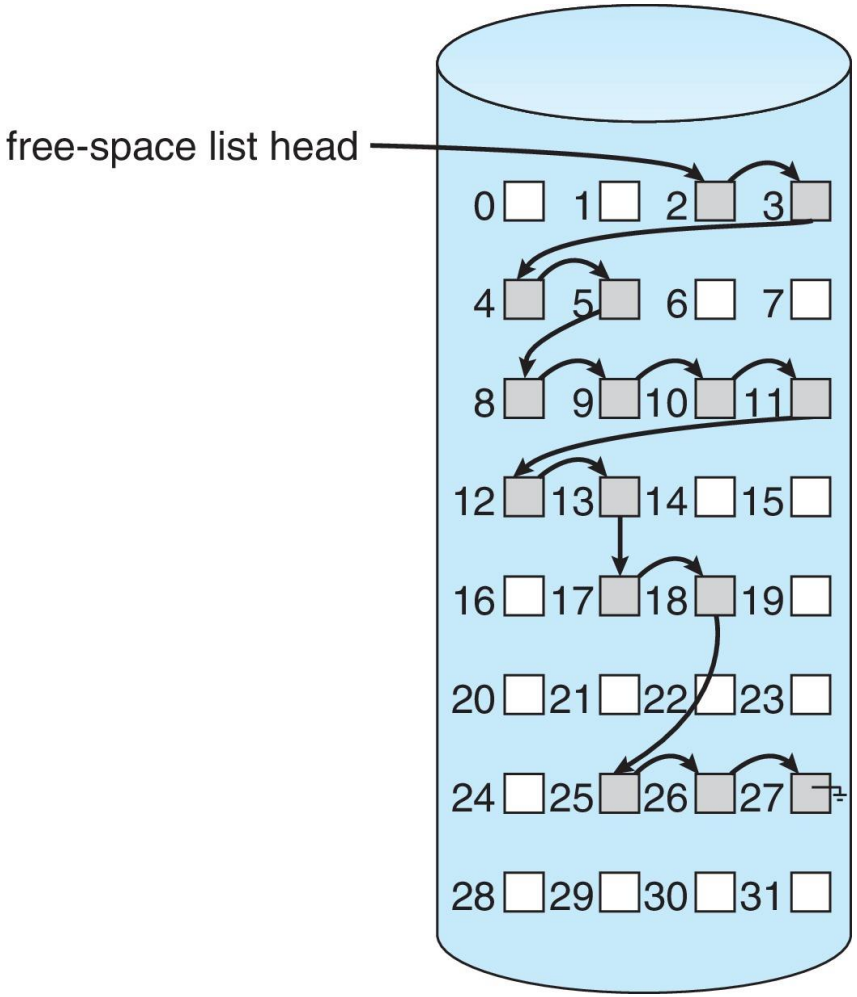
    ***n* = $2^{40}/2^{12} = 2^{28}$ bits (or 32MB)**

    if clusters of 4 blocks -> 8MB of memory

- Easy to get contiguous files

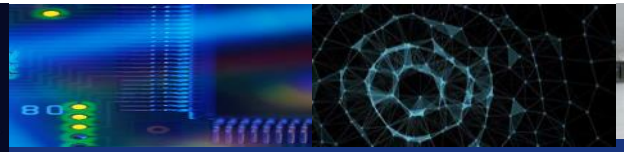
空闲空间管理——链表

- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste. Linked Free Space List on Disk of space
  - No need to traverse the entire list (if # free blocks recorded)



空闲空间管理

- **Grouping（组）**
  - 空闲列表方法的改进方案：在第一个空闲块中存储n个空闲块的地址。最后一块包含另外n个空闲块的地址，如此继续。大量空闲快的地址可以很快地找到，这一点有别于标准链表方法
- **Counting（计数）**
  - 通常，多个连续块可能同时分配或释放，尤其是采用连续区域分配算法或者采用簇来分配空间更是如此。因此，不记录n个空闲块的磁盘地址，转而：
    - 记录第一个空闲块的地址和后续空闲块的数量n
    - 空闲空间列表的每个条目包含磁盘地址和数量
    - 每个条目占用更多空间，但是表的总长度会更短
    - 这些条目可以存储在平衡树而不是链表中，以便于高效查找、插入和删除

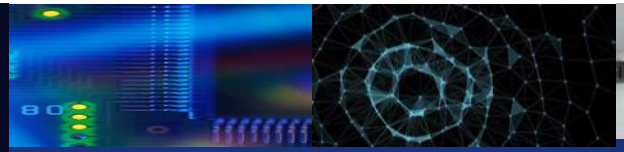



空闲空间管理

- **Space Maps （空间图）**
  - 在 ZFS 中使用
  - 考虑超大文件系统上的元数据 I/O
    - 像位图这样的完整数据结构无法放入内存—数以千计的 I/O
  - 将设备空间划分为 metaslab 单元并管理 metaslab
    - 给定的卷可以包含数百个 metaslabs
  - 每个 metaslab 都有关联的空间图
    - 使用计数算法
  - 但是记录到日志文件而不是文件系统
    - 所有块活动的日志，按时间顺序，以计数格式
  - Metaslab 活动—将空间映射加载到平衡树结构中的内存中，由偏移量索引
    - 重播登录到该结构
    - 将连续的空闲块组合成单个条目

效率和性能

- 磁盘空间的有效使用很大程度上取决于磁盘分配和目录算法
- 影响效率的因素：
  - 磁盘分配和目录算法
  - 保存在文件目录条目中的数据类型
  - 元数据结构的预分配或按需分配
  - 固定大小或可变大小的数据结构

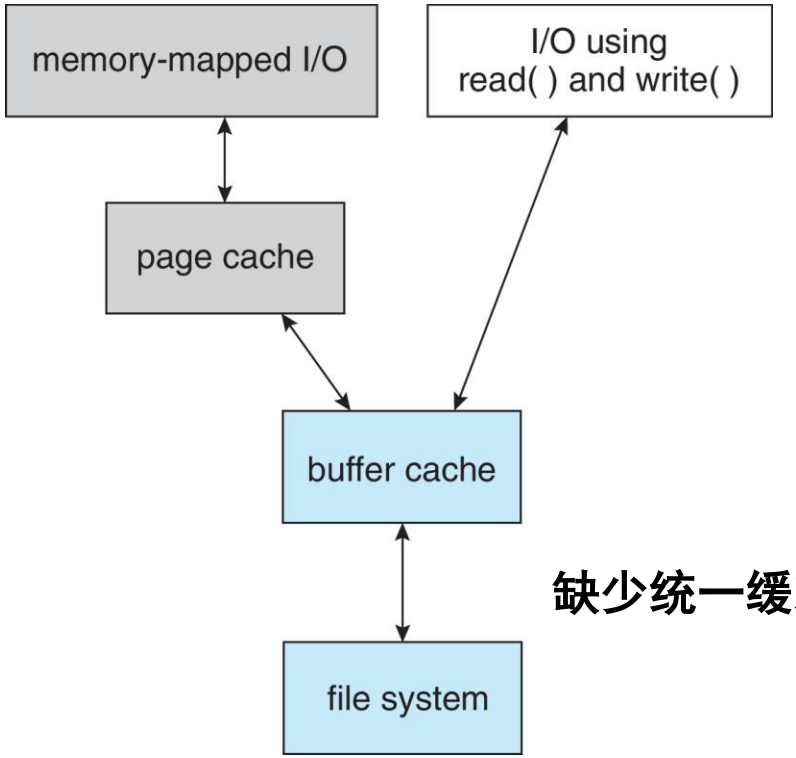


效率和性能

- **即时选择了基本的文件系统算法，仍然能够用多种方式来提高性能**
  - 将数据和元数据保持在一起
  - **Buffer cache** –主内存的独立部分，用于存放经常使用的块
  - **应用程序有时会请求或操作系统需要同步写入**
    - 无缓冲/缓存——写入必须在确认前到达磁盘
    - 异步写入更常见、可缓冲、更快
  - **随后释放（Free-behind） 和预先读取（read-ahead）** –优化顺序访问的技术
  - 读取经常比写入慢

效率和性能

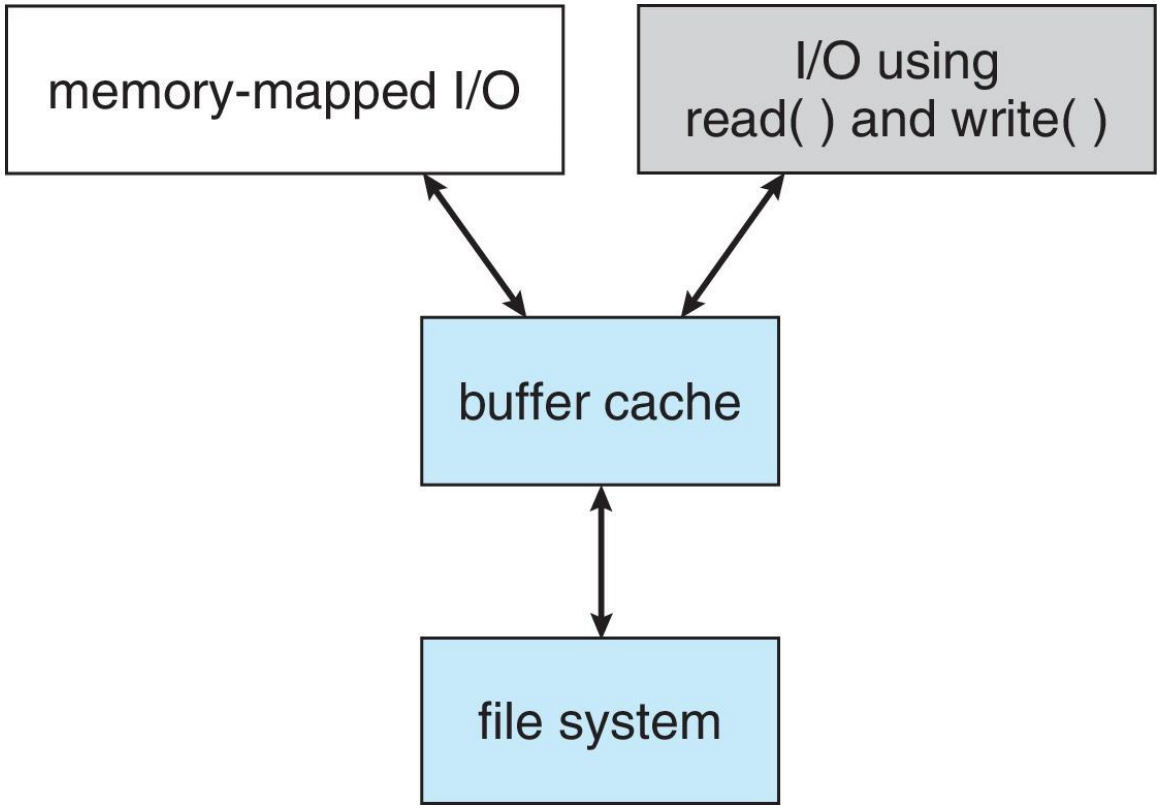
- 页面缓存使用虚拟内存技术和地址缓存页面而不是磁盘块
- 内存映射 I/O 使用页面缓存
- 通过文件系统的例程 I/O 使用缓冲区（磁盘）缓存

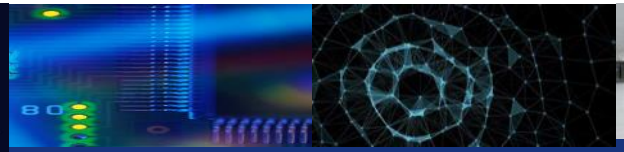


缺少统一缓冲区缓存的I/O

统一缓冲区缓存

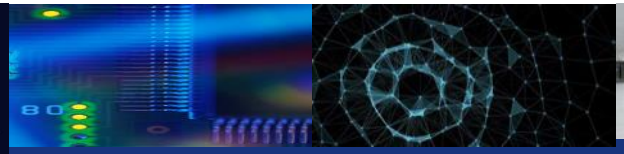
- 统一的缓冲区缓存使用相同的页面缓存来缓存内存映射页面和普通文件系统 I/O，以避免双重缓存





恢复

- 一致性检查——将目录结构中的数据与磁盘上的数据块进行比较，并尝试修复不一致之处
  - Can be slow and sometimes fails
- 使用系统程序将数据从磁盘备份到另一个存储设备（磁带、其他磁盘、光盘）
- 通过从备份恢复数据来恢复丢失的文件或磁盘

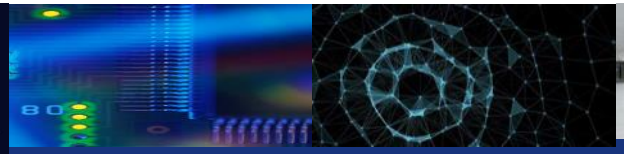


基于日志的文件系统

- **日志结构（或日志）文件系统将每个元数据更新记录为一个事务**
- **所有事务（transaction）都写入日志**
  - 事务一旦被写入日志（顺序地）就被视为已提交
  - 有时到单独的设备或磁盘部分
  - 但是，文件系统可能尚未更新
- **日志中的事务异步写入文件系统结构**
  - 修改文件系统结构时，事务将从日志中删除
- **如果文件系统崩溃，日志中所有剩余的事务仍必须执行**
- **更快地从崩溃中恢复，消除元数据不一致的可能性**

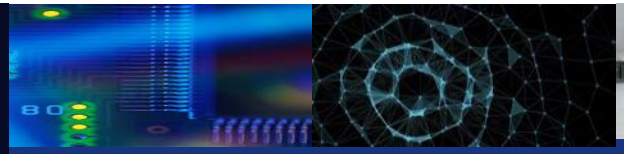
网络文件系统

- NFS是基于客户机-服务器的网络文件系统，它被广泛使用
- NFS将一组互连的工作站视作一组具有独立文件系统的独立机器，以允许共享这些文件系统
- 共享是基于客户机-服务器关系
- 每台机器可能是，而且往往，既是客户机也是服务器
- 任何两台之间允许共享
- NFS设计目标之一是，支持不同机器、操作系统和网络架构组成的异构环境
- NFS规范区分两种服务：
  - 安装机制的服务
  - 真正远程文件访问服务
- 实现这些服务有两个单独的协议：安装协议和远程访问协议，即NFS协议。协议是用RPC来表示的



小结

- 文件系统结构，文件系统的层次化结构
- 文件系统实现：分区与安装、虚拟文件系统（VFS）
- 目录实现：线性列表、哈希表
- 分配方法：连续分配、链接分配、索引分配
- 空闲空间管理：位向量、链表、组、计数、空闲图
- 效率与性能：缺少（采用）统一缓冲区缓存的I/O
- 恢复：一致性检查、基于日志的文件系统、备份与恢复



谢谢