# 0117401: Operating System
# 操作系统原理与设计

## Chapter 11: File system implementation(文件系统实现)


陈香兰

xlanchen@ustc.edu.cn

http://staff.ustc.edu.cn/~xlanchen

Computer Application Laboratory, CS, USTC @ Hefei
Embedded System Laboratory, CS, USTC @ Suzhou


May 24, 2023

# 温馨提示：

为了您和他人的工作学习，
请在课堂上**关机或静音。**

**不要**在课堂上**接打电话。**

# 提纲

- File-System Structure
- FS Implementation
- Directory Implementation
- Allocation Methods (分配方法)
- Free-Space Management
- Efficiency (空间) and Performance (时间)
- Recovery
- Log Structured File Systems
- 小结

# Outline

File-System Structure

# File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- FS resides on secondary storage (disks)
- FS organization
  - How FS should look to the user
  - How to map the logical FS onto the physical secondary-storage devices
- **FS organized into layers**

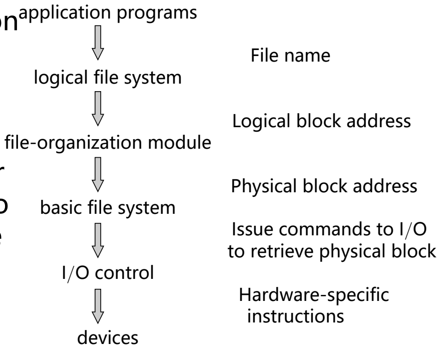



Figure: Layered File System

# Outline

FS Implementation

# FS Implementation

- Structures and operations used to implement file system operation, OS- & FS-dependment
  1. On-disk structures
  2. In-memory structures

# FS Implementation

1. **On-disk structures**

   1.1 **Boot control block**

   - To boot an OS from the partition (volume)
   - If empty, no OS is contained on the partition

   1.2 **Volume control block**

   1.3 **Directory structure**

   1.4 **Per-file FCB**

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

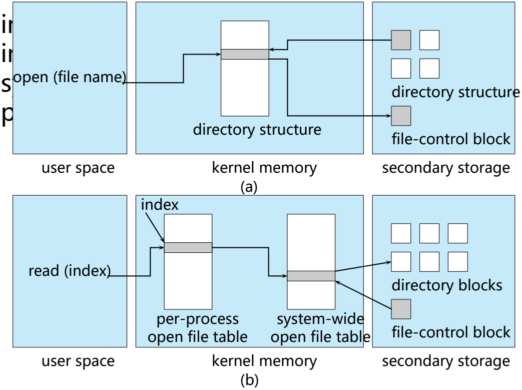
**Figure: A typical file control block**

# FS Implementation

2. **In-memory information**: For both FS management and performence improvement via caching
   - Data are loaded at mount time and discarded at dismount
   - Structures include:
     - in-memory mount table;
     - in-memory directory-structure cache
     - system-wide open-file table;
     - per-process open-file table

# FS Implementation

2. **In-memory information**: For both FS management and performence improvement via caching
   - Data are loaded at mount time and discarded at dismount
   - Structures include:

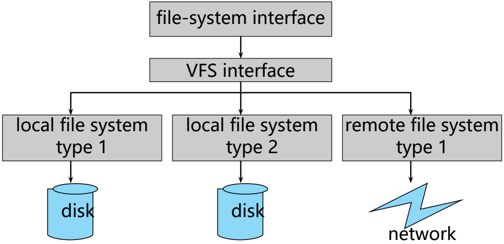


# Partitions and mounting

- Partition (分区)
  - Raw (E.g. UNIX swap space & some database) VS. cooked
  - Boot information, with its own format
    - Boot image
    - Boot loader unstanding multiple FSes & OSes Dual-boot
- **Root partition is mounted at boot time**
- Others can be automatically mounted at boot or manually mounted later

# Virtual File Systems (虚拟文件系统)

- **Virtual File Systems (VFS, 虚拟文件系统)** provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.




Schematic View of Virtual File System

# Outline

Directory Implementation

# Directory Implementation

1. **Linear list** of file names with pointer to the data blocks.
   - Simple to program
   - Time-consuming to execute
2. **Hash Table**– linear list with hash data structure.
   - Decreases directory search time
   - **Collisions** – situations where two file names hash to the same location
   - Fixed & variable size or chained-overflow hash table

# Outline

Allocation Methods (分配方法)

# Allocation Methods (分配方法)

- An allocation method refers to
  **how disk blocks are allocated** for files
  so that disk space is **utilized effectively**
  & files can be **accessed quickly**
  1. Contiguous allocation (连续分配)
  2. Linked allocation (链接分配)
  3. Indexed allocation (索引分配)
  4. Combined (组合方式)

# 1. Contiguous Allocation (连续分配) I

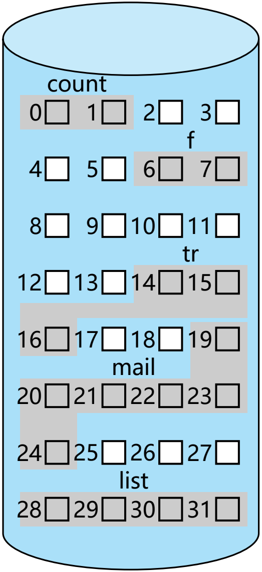
- Each file occupies **a set of contiguous blocks on the disk**
- Simple – directory entry only need
  - **starting location (block #)**
  - & **length (number of blocks)**
- Mapping from logical to physical

  $\text{LogicalAddress}/512 = Q \ldots\ldots R$

  **Block to be accessed = Q + starting address**

  **Displacement into block = R**

# 1. Contiguous Allocation (连续分配) II




directory

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

# 1. Contiguous Allocation (连续分配) III

- Advantages:
  - Support **both random & sequential** access
    - Start block: b;
      Logical block number: i
      $\Rightarrow$physical block number: $b + i$
    - **Fast** access speed, because of short head movement
- Disadvantages:
  - **External fragmentation**
    - **Wasteful of space** (dynamic storage-allocation problem).
  - **Files cannot grow**,
    or File size must be known in advance.
    $\Rightarrow$**Internal fragmentation**

# Extent-Based Systems

- Many newer file systems (I.e. Veritas File System) use **a modified contiguous allocation scheme**
- 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.

# 2. Linked Allocation (链接分配)

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.
- Two types
  1. **Implicit (隐式链接)**
  2. **Explicit (显式链接)**

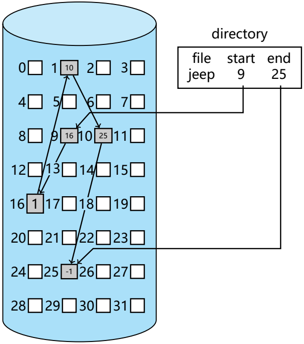
# 2. Linked Allocation (链接分配)

## 1. Implicit (隐式链接)

- **Directory** contains a pointer to the first block & last block of the file.
- **Each block** contains a pointer to to the next block.

a block = pointer

- **Allocate as needed, link together**
  - Simple – need only starting address
  - Free-space management system – **no waste of space**



# 2. Linked Allocation (链接分配)

## 1. Implicit (隐式链接)

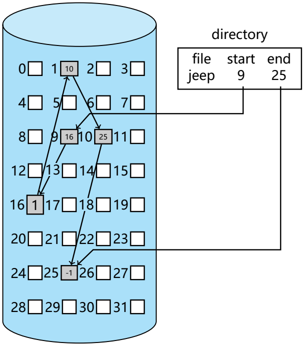
- **Disadvantage**:
  - **No random access**
  - **Link pointers need disk sapce**

    E.g.: 512 per block, 4 per pointer ⇒0.78%

    **Solution**: **clusters**

    ⇒ disk throughput ↑

    But internal fragmentation↑



# 2. Linked Allocation (链接分配)

## 1. **Implicit (隐式链接)**

- **Mapping:**

  Suppose

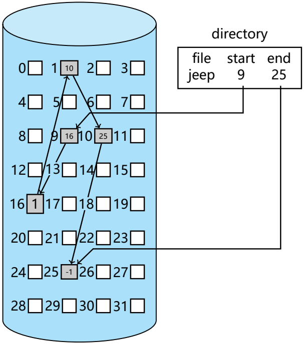
  1.1 block size=512B,
  1.2 block pointer size=1B, using the first byte of a block
  1.3 Logical addr in the file to be accessed$=$ A

  we have

  1.1 Data size for each block =$512 - 1 = 511$
  1.2 A$/511 =$ Q$\ldots\ldots$R

  then

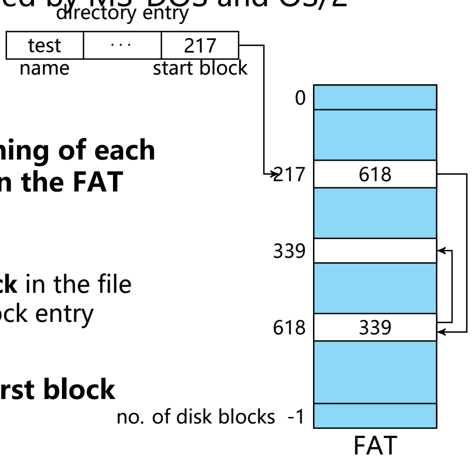
  1.1 Block to be accessed is the Q$^{th}$ block in the linked chain of blocks representing the file.
  1.2 Displacement into block = R + 1




- **How to reduce searching time?**

# 2. Linked Allocation (链接分配)

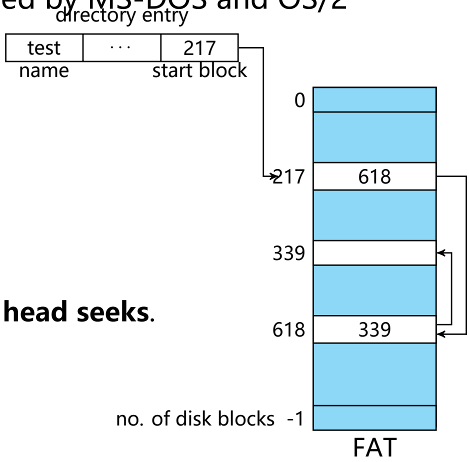
2. **Explicit linked allocation:**
   **File Allocation table, FAT**
   Disk-space allocation used by MS-DOS and OS/2




- **A section of disk at the beginning of each partition is set aside to contain the FAT**
  - Each disk block one entry
  - The entry contains
    (1) **the index of the next block** in the file
    (2) **end-of-file**, for the last block entry
    (3) **0**, for unused block
- **Directory entry contains the first block number**

# 2. Linked Allocation (链接分配)

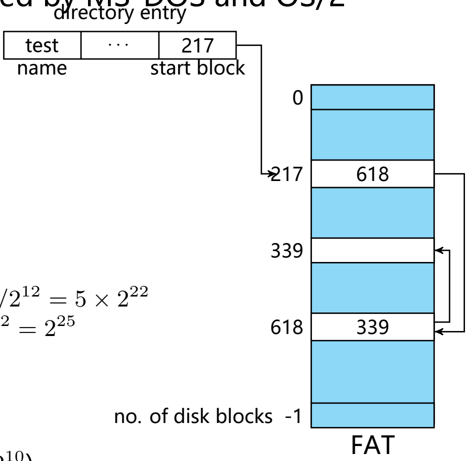
2. **Explicit linked allocation:**
   **File Allocation table, FAT**
   Disk-space allocation used by MS-DOS and OS/2




- Now **support random access**, but still not very efficient
- May result in a **significant disk head seeks**. **Solution**: Cached FAT

# 2. Linked Allocation (链接分配)

2. **Explicit linked allocation:**
   **File Allocation table, FAT**
   Disk-space allocation used by MS-DOS and OS/2




- **How to compute FAT size?**
  Suppose
  2.1 Disk space = 80 GB
  2.2 Block size = 4 KB

  Then
  2.1 Total block number = $80 \times 2^{30}/2^{12} = 5 \times 2^{22}$
  2.2 $4 \times 2^{22} = 2^{24} < 5 \times 2^{22} < 8 \times 2^{22} = 2^{25}$

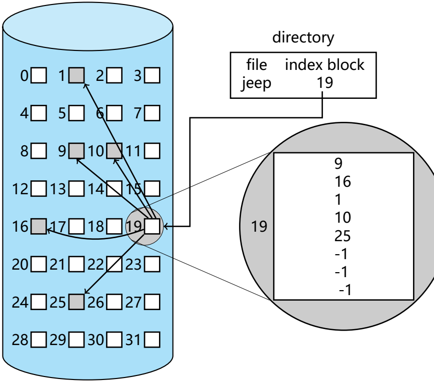
  - **Length of each FAT entry?**
    (25bits? 28bits? 32bits?)
  - **Length of FAT?**
    ($5 \times 2^{22} \times 4$B $= 80$MB $= 80$GB$/2^{10}$)

# 3. Indexed Allocation (索引分配)

- **Indexed Allocation (索引分配):** Brings all pointers together into one location – **the index block**.

- **Each file** has its own index block
- **Directory entry** contains the index block address
- **Each index block**: An array of pointers (an index table)

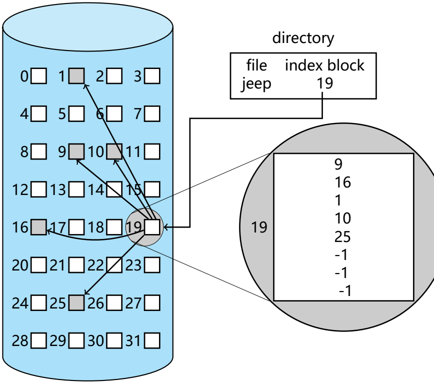
Logical block number i
$=$ the i$^{th}$ pointer



# 3. Indexed Allocation (索引分配)

- **Indexed Allocation (索引分配):** Brings all pointers together into one location – **the index block**.

- **Advantage**:
  - **Random** access
  - Dynamic access **without external fragmentation**
- **Disadvantage**:
  - have **overhead** of index block.
  - File **size limitation**, since one index block can contains limited pointers



# 3. Indexed Allocation (索引分配)

- **Indexed Allocation (索引分配):** Brings all pointers together into one location – **the index block**.

- **Mapping** from logical to physical

  Suppose

  (1) Block size = 1KB

  (2) Index size = 4B

  Then for logical address LA, we have

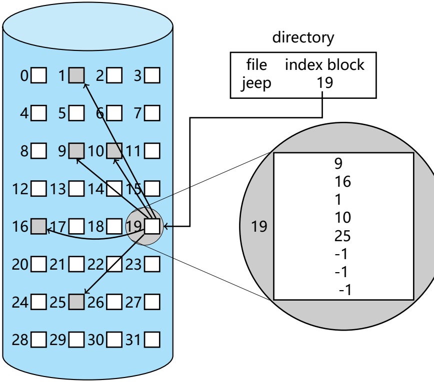
  $$\text{LA}/512 = \text{Q}\ldots\text{R}$$

  (3)Q = the index of the pointer

  (4)R = displacement into block

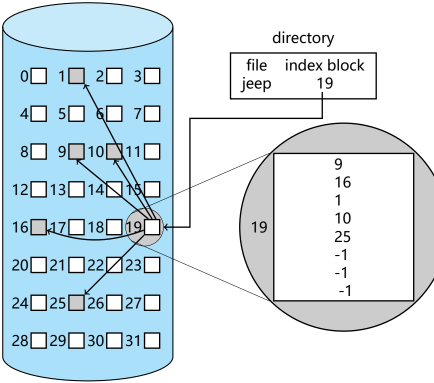
  We also have Max file size

  $$= 2^{10}/4 \times 1\text{KB} = 256\text{KB}$$



# 3. Indexed Allocation (索引分配)

- **Indexed Allocation (索引分配):**
  Brings all pointers together into one location – **the index block**.




- **How to support a file of unbounded length?**
  1. **linked scheme**
  2. **multi-level index scheme**

# 3. Indexed Allocation (索引分配)

1. **Linked scheme**
   - **Link blocks of index table (no limit on size)**.
   - **Mapping**

     Suppose

     (1) Block size=1KB

     (2) Index or link pointer size = 4B

     Then

     $$\mathrm{LA}/\left(1\mathrm{KB} \times (1\mathrm{K}/4 - 1)\right) = \mathrm{Q}_1 \ldots \mathrm{R}_1$$

     (3) $\mathrm{Q}_1$ = block of index table

     (4) $\mathrm{R}_1$ is used as follows:

     $$\mathrm{R}_1/1\mathrm{K} = \mathrm{Q}_2 \ldots\ldots \mathrm{R}_2$$

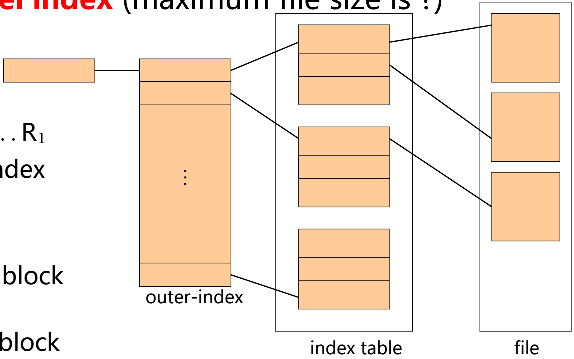
     (5) $\mathrm{Q}_2$ = index into block of index table

     (6) $\mathrm{R}_2$ = displacement into block of file:

# 3. Indexed Allocation (索引分配)

## 2. **multi-level index scheme**

Example: **Two-level index** (maximum file size is ?)




- We have

  $$\mathrm{LA}/(1\mathrm{K} \times 1\mathrm{K}/4) = \mathrm{Q}_1 \ldots \mathrm{R}_1$$

  (1) $\mathrm{Q}_1$ = index into outer-index

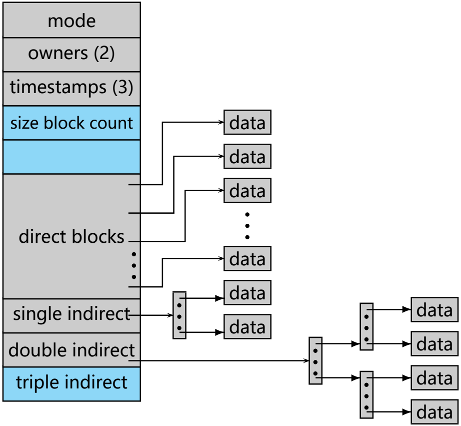
  (2) $\mathrm{R}_1$ is used as follows:

  $$\mathrm{R}_1/1\mathrm{KB} = \mathrm{Q}_2 \ldots \mathrm{R}_2$$

  (3) $\mathrm{Q}_2$ = displacement into block of index table

  (4) $\mathrm{R}_2$ = displacement into block of file

# 4. Combined Scheme (组合方式): UNIX (4K bytes per block) I



# 4. Combined Scheme (组合方式): UNIX (4K bytes per block) II

- if 4KB per block, and 4B per entry

$$\begin{aligned} \text{Direct blocks} &= 10 \times 4\text{KB} = 40\text{KB} \\ \text{Number of entries per block} &= 4\text{KB}/4\text{B} = 1\text{K} \\ \text{Single indirect} &= 1\text{K} \times 4\text{KB} = 4\text{MB} \\ \text{Double indirect} &= 1\text{K} \times 4\text{MB} = 4\text{GB} \\ \text{Triple indirect} &= 1\text{K} \times 4\text{GB} = 4\text{TB} \end{aligned}$$

**Maximnm file size = ?**

# Outline

Free-Space Management

# Free-Space Management

- **Disk Space**: limited
  - Free space management: **To keep track of free disk space**
  - **How**? Free-space list?
  - **Algorithms**
    1. **Bit vector**
    2. **Linked list**
    3. **Grouping (成组链接法)**
    4. **Counting**

# Free-Space Management

1. **Bit vector**
   - **Free-space list** is implemented as a **bit map** or **bit vector**
     - **1 bit for each block**
       1=free;
       0=allocated
     - Example:
       a disk where blocks 2,3,4,5,8,9,10,11,12,13,17,18,25,26,27 are free and the rest blocks are allocated. The bitmap would be

$$0011\ 1100\ 1111\ 1100\ 0110\ 0000\ 0111\ 0000\ 0\ldots$$

     - **Bit map length**.
       For n blocks, if the base unit is word, and the size of word is 16 bits, then

$$\text{bit map length} = (\text{n} + 15)/16$$

       U16 bitMap[bitMaptLength];

# Free-Space Management

1. **Bit vector**
   - **How to find the first free block or n consecutive free blocks on the disk?**
     - Many computers supply **bit-manipulation instructions**
     - To find the first free block:
       Suppose: base unit = word (**16** bits) or other
       (1) find **the first non-0 word**
       (2) find **the first 1 bit** in the first non-0 word
     - If first K words is $0$, & $(\mathrm{K}+1)^{\mathrm{th}}$ word $> 0$,
       the first $(\mathrm{K}+1)^{\mathrm{th}}$ word' s first $1$ bit has offset L,
       then

       **first free block number** $\mathrm{N} = \mathrm{K} \times 16 + \mathrm{L}$

# Free-Space Management

1. **Bit vector**
   - **Simple**
   - Must be kept on disk

     **Bit map requires extra space**,

     Example:

     block size = $2^{12}$ bytes

     disk size = $2^{30}$ bytes (1 gigabyte)

     n = $2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)

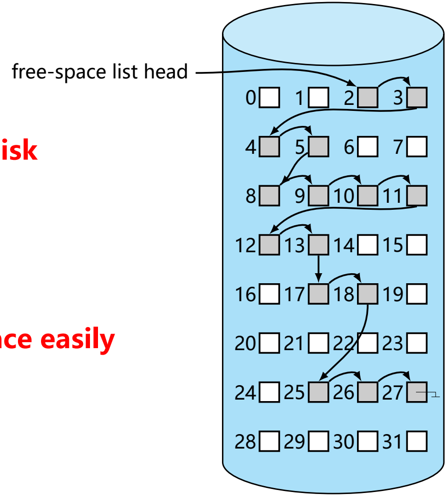
     - **Solution**: **Clustering**

# Free-Space Management

1. **Bit vector**
   - **Efficient** to get the first free block or n consecutive free blocks, **if we can always store the vector in memory**.
     - But **copy in memory and disk may differ**.
       E.g. **bit[i] = 1 in memory & bit[i] = 0 on disk**
     - **Solution**:
       Set bit[i] = 1 in memory.
       Allocate block[i]
       Set bit[i] = 1 in disk
   - Need to protect:
     - Pointer to free list
     - Bit map

# Free-Space Management

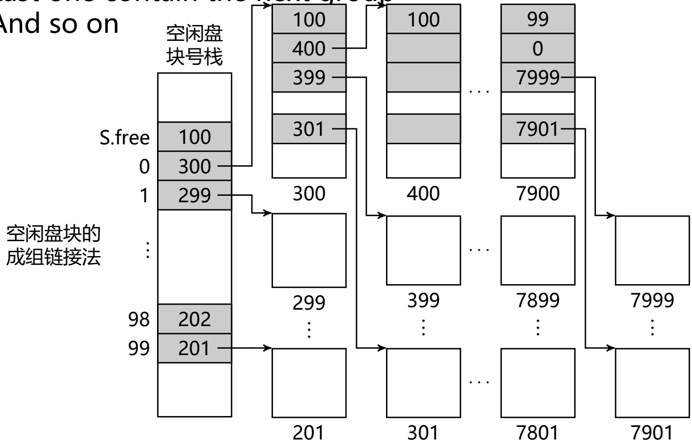
## 2. Linked Free Space List on Disk




- **Link together all the free disk blocks**
  - **First** free block
  - **Next** pointer
- Not efficient
- **Cannot get contiguous space easily**
- No waste of space

# Free-Space Management

3. **Grouping(成组链接法)**: To store the addresses of n free blocks (a group) in the first free block. E.g.: UNIX
   - First n-1 group members are actually free
   - Last one contain the next group
   - And so on



# Free-Space Management

4. **Counting**
   - Assume:
     Several contiguous blocks may be allocated or freed simultaneously
   - Each = first free block number & a counter (number of free blocks)
   - Shorter than linked list at most time, generally counter > 1

# Outline

Efficiency (空间) and Performance (时间)

# 1 Efficiency (空间)

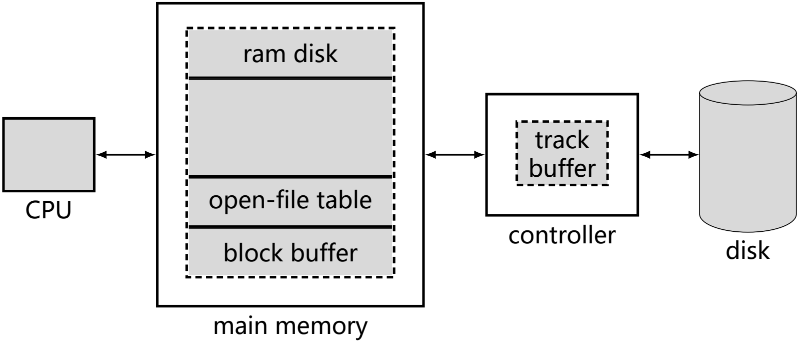
**Efficiency in usage of disk space** dependent on:

1. Disk allocation and directory algorithms
2. **Various approaches**
   - Inodes distribution
   - Variable cluster size
   - Types of data kept in file' s directory entry
   - Large pointers provides larger file length, but cost more disk space

# 2 Performance (时间)

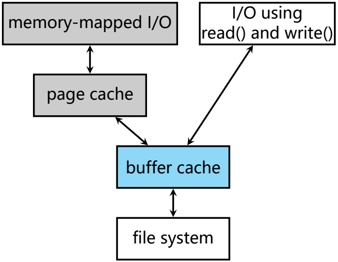
- **Performance**: other ways
  - **disk cache** - on disk controllers, large enough to store **entire tracks** at a time.
  - **buffer cache** – separate section of main memory for frequently used **blocks**
  - **page cache** - uses virtual memory techniques to **cache file data as pages** rather than as file-system-oriented blocks
  - **Synchronous writes** VS. **Asynchronous writes**
  - **free-behind and read-ahead** – techniques to optimize **sequential access**
  - improve PC performance by dedicating section of memory as virtual disk, or **RAM disk**

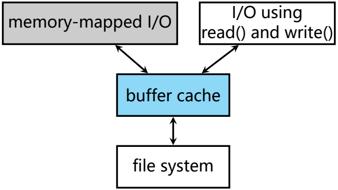
# 2 Performance (时间)



# Unified Buffer Cache

- **I/O Without a Unified Buffer Cache**
  - Memory-mapped I/O uses a page cache
  - Routine I/O through the file system uses the buffer (disk) cache
  - **Problem**: double caching
- **I/O Using a Unified Buffer Cache**
  - A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O



# Outline

Recovery

# Recovery

- **Consistency checking** (一致性检查)
  - **compares** data in directory structure with data blocks on disk, and **tries to fix** inconsistencies
  - UNIX: fsck
  - MS-DOS: chkdsk
- **Backup & restore**
  - Use system programs to back up data from disk to another storage device (floppy disk, magnetic tape, other magnetic disk, optical)
  - Recover lost file or disk by restoring data from backup
  - A typical backup schedule may be:
    Day1: **full backup**;
    Day2: **incremental backup**;
    ...
    DayN: incremental backup. Then go back to Day1.

# Outline

Log Structured File Systems

# Log Structured File Systems

- **Log-based transaction-oriented** (or journaling, 日志) file systems record each update to the file system as a **transaction**
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log
  - However, the file system may not yet be updated
- The transactions in the log are **asynchronously** written to the file system
  - When the file system is modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed

# Outline

小结

# 小结

- File-System Structure
- FS Implementation
- Directory Implementation
- Allocation Methods (分配方法)
- Free-Space Management
- Efficiency (空间) and Performance (时间)
- Recovery
- Log Structured File Systems
- 小结

**Thank you! Any question?**