存储器层次结构

Aug. 31, 2025

主要内容

- 存储技术与发展趋势
- 局部性
- 内存层次结构中的缓存机制

随机存取存储器（RAM）

- 主要特性
  - RAM, （Random-Access Memory）通常以芯片的形式封装。
  - 基本的存储单元通常称为 存储单元（cell），每个单元存储 1 位（bit）。
  - 多个 RAM 芯片组合在一起形成一个完整的 内存（memory）。

- RAM 的两种类型（RAM comes in two varieties）
  - SRAM（静态随机存取存储器，Static RAM）
  - DRAM（动态随机存取存储器，Dynamic RAM）

SRAM 与 DRAM 对比总结

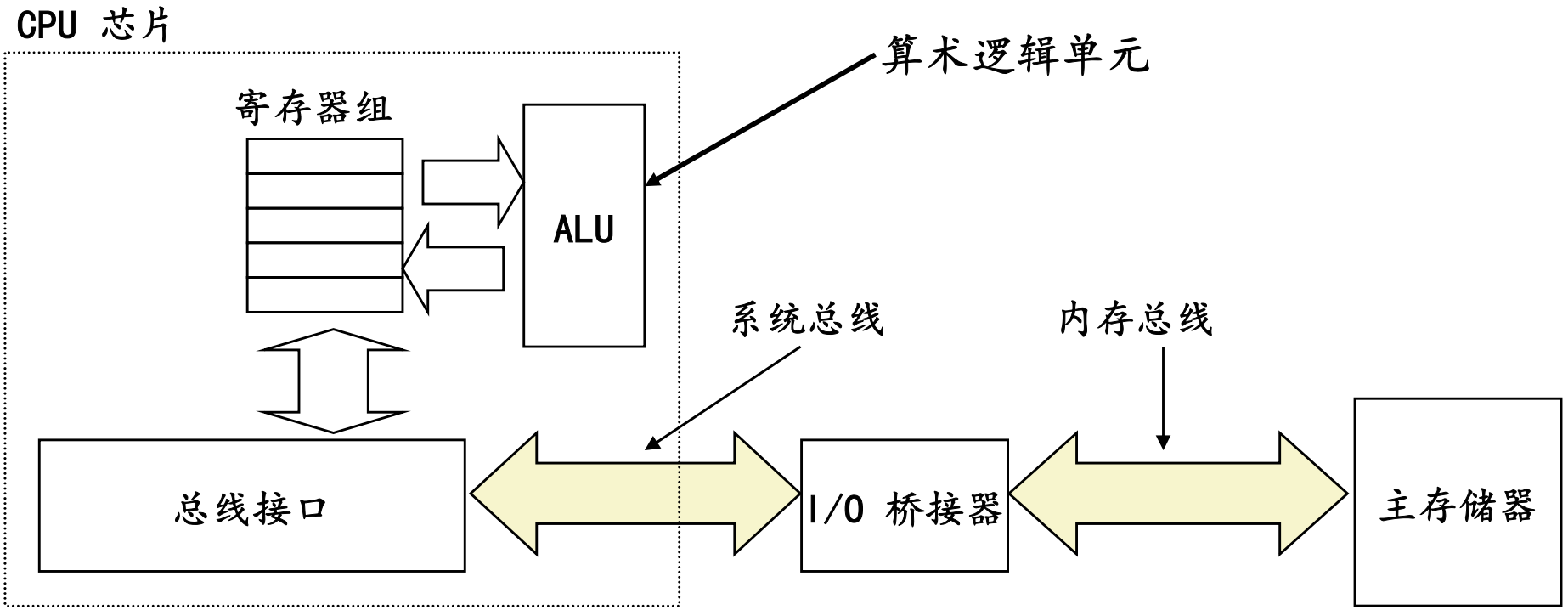
特性	**SRAM（静态 RAM）**	**DRAM（动态 RAM）**
每位晶体管数	4 或 6 个	1 个
访问时间	1X（速度快）	10X（速度较慢）
是否需要刷新	不需要	需要
是否需要纠错码	可能需要	需要
成本	高，大约是 DRAM 的 **100 倍**	低，1X
应用场景	**缓存存储器**	**主存储器**、帧缓冲区

非易失性存储器

- DRAM 和 SRAM 是易失性存储器
  - 断电后会丢失信息。
- 非易失性存储器在断电后仍然保持数据。
  - 只读存储器（ROM）：在生产过程中写入程序。
  - 可编程只读存储器（PROM）：可编程一次。
  - 可擦除可编程只读存储器（EPROM）：可整体擦除（使用紫外线或 X 射线）。
  - 电可擦除可编程只读存储器（EEPROM）：支持电子方式擦除。
  - 闪存（Flash memory）：属于 EEPROM，支持部分块级擦除功能。大约在擦写 100,000 次后会磨损失效。
- 非易失性存储器的用途
  - 固件程序存储在 ROM 中（例如 BIOS、磁盘控制器、网卡、图形加速器、安全子系统等）。
  - 固态硬盘（SSD）（在 U 盘、智能手机、MP3 播放器、平板电脑、笔记本电脑等设备中替代机械硬盘）。
  - 磁盘缓存（Disk caches）。

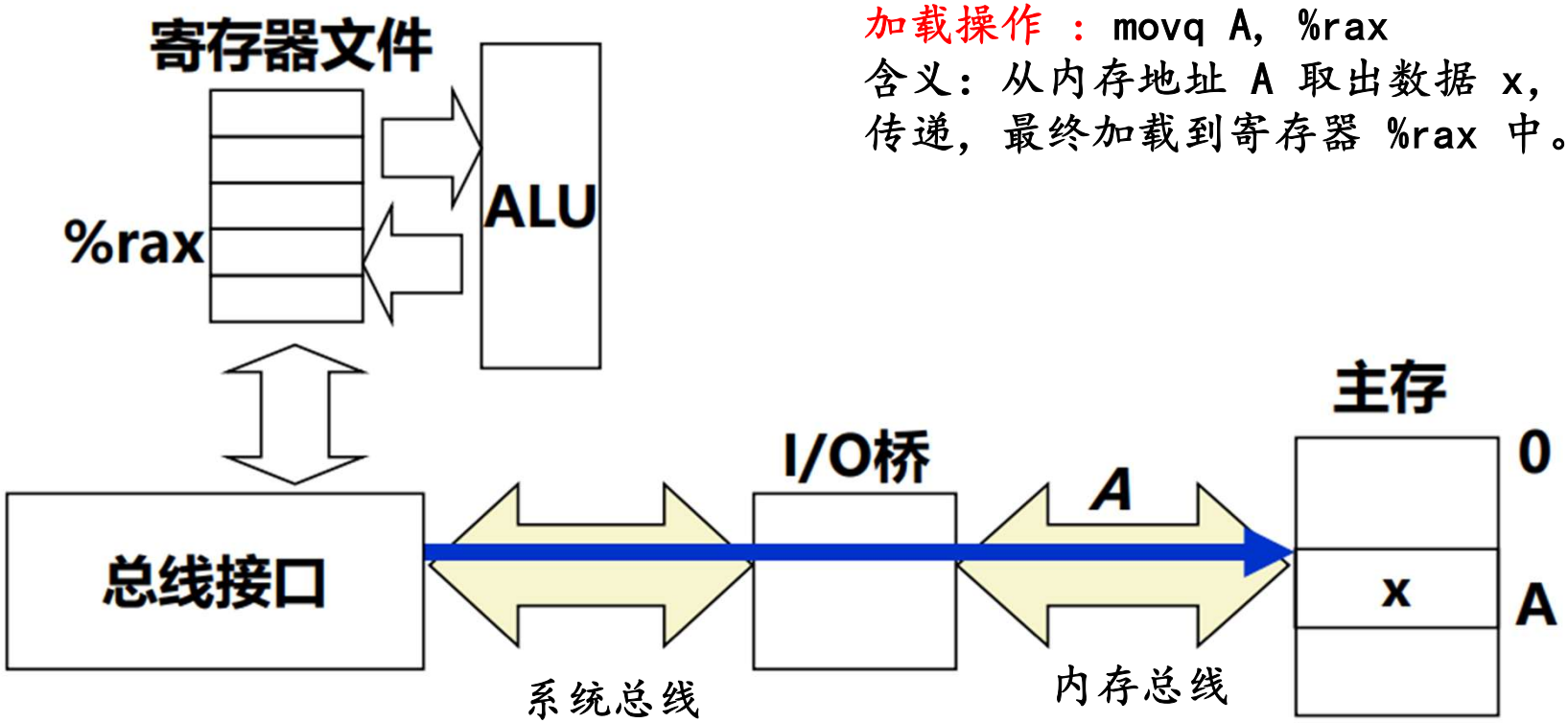
连接CPU和存储器的典型总线结构

- 总线（bus）是一组并行导线的集合，用于传输地址、数据和控制信号。
- 总线通常由多个设备共享。



内存读取（1）

- CPU 将地址 A 放置到内存总线上。

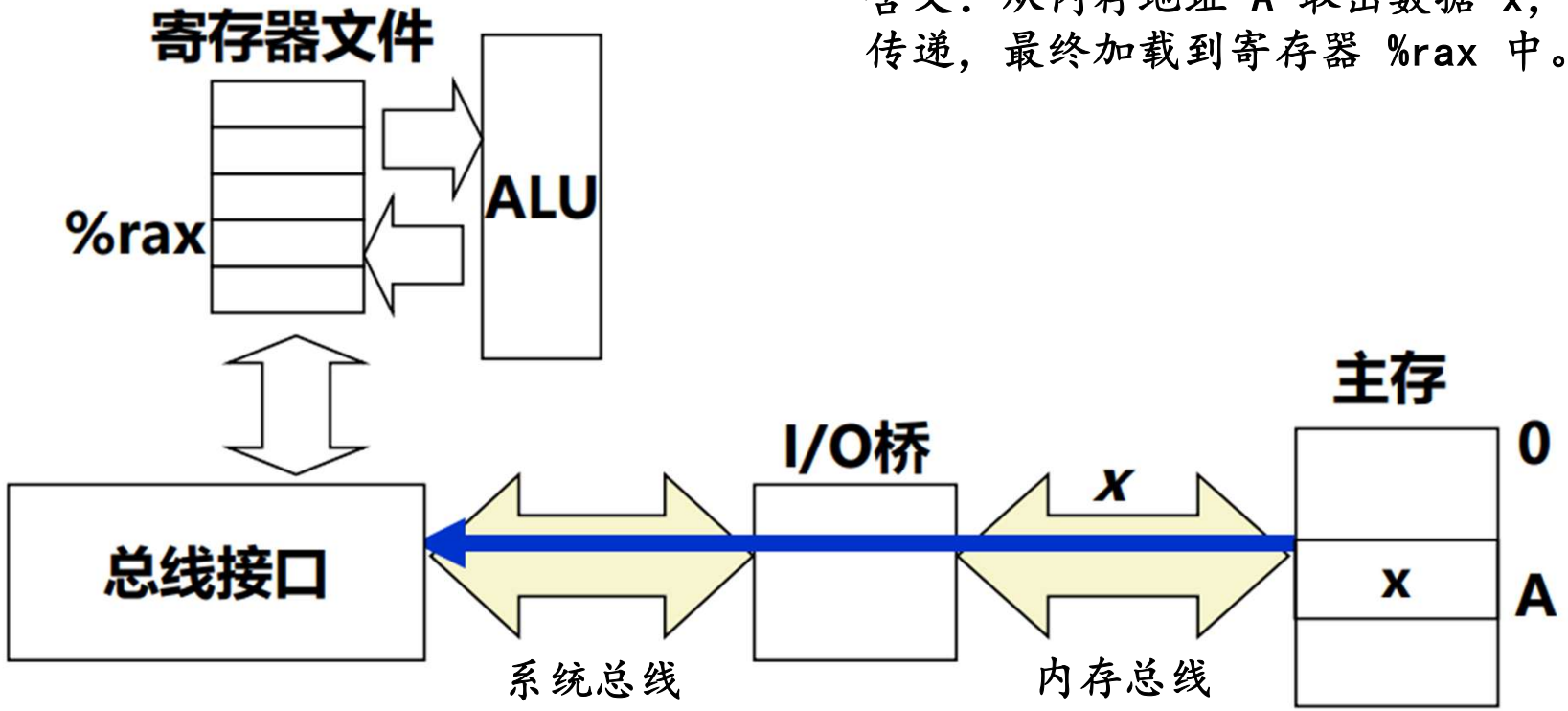


加载操作 ： movq A, %rax
含义：从内存地址 A 取出数据 x，通过总线传递，最终加载到寄存器 %rax 中。

内存读取 (2)

- 主存储器从内存总线读取地址 A，获取对应的字 x，并将 x 放到总线上。

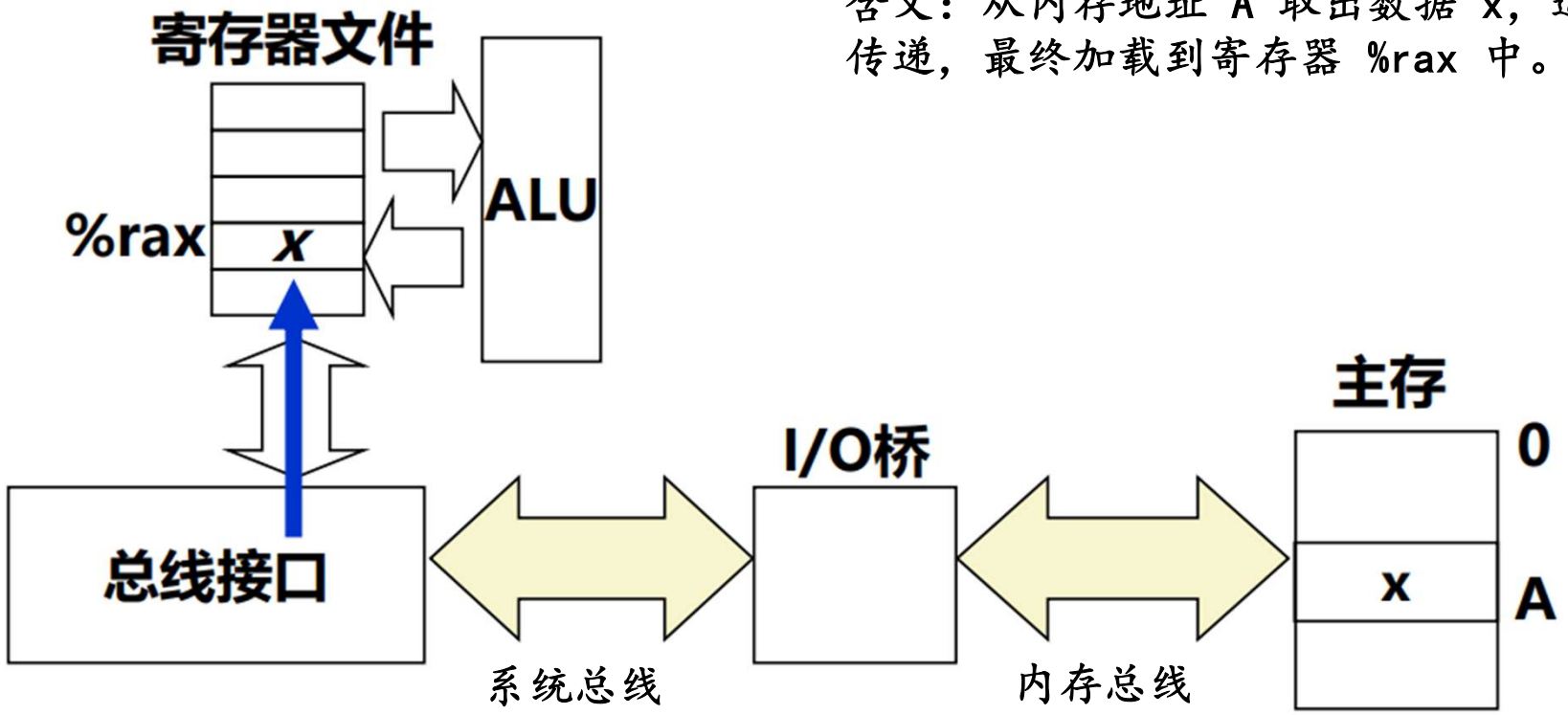
加载操作 ： movq A, %rax
含义：从内存地址 A 取出数据 x，通过总线传递，最终加载到寄存器 %rax 中。



内存读取(3)

- CPU 从总线上读入字x，并将其放入寄存器%rax

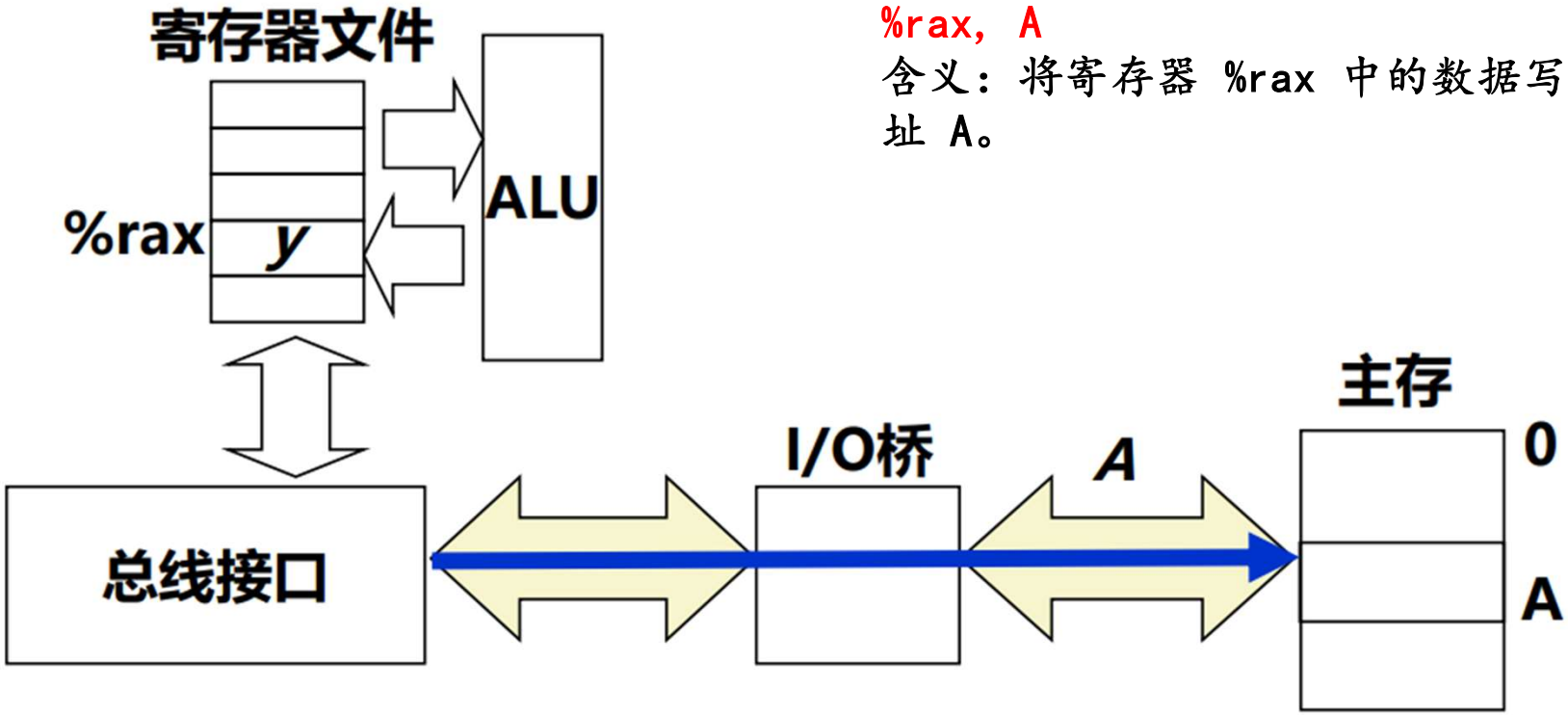
加载操作 ： movq A, %rax
含义：从内存地址 A 取出数据 x，通过总线传递，最终加载到寄存器 %rax 中。



内存写入(1)

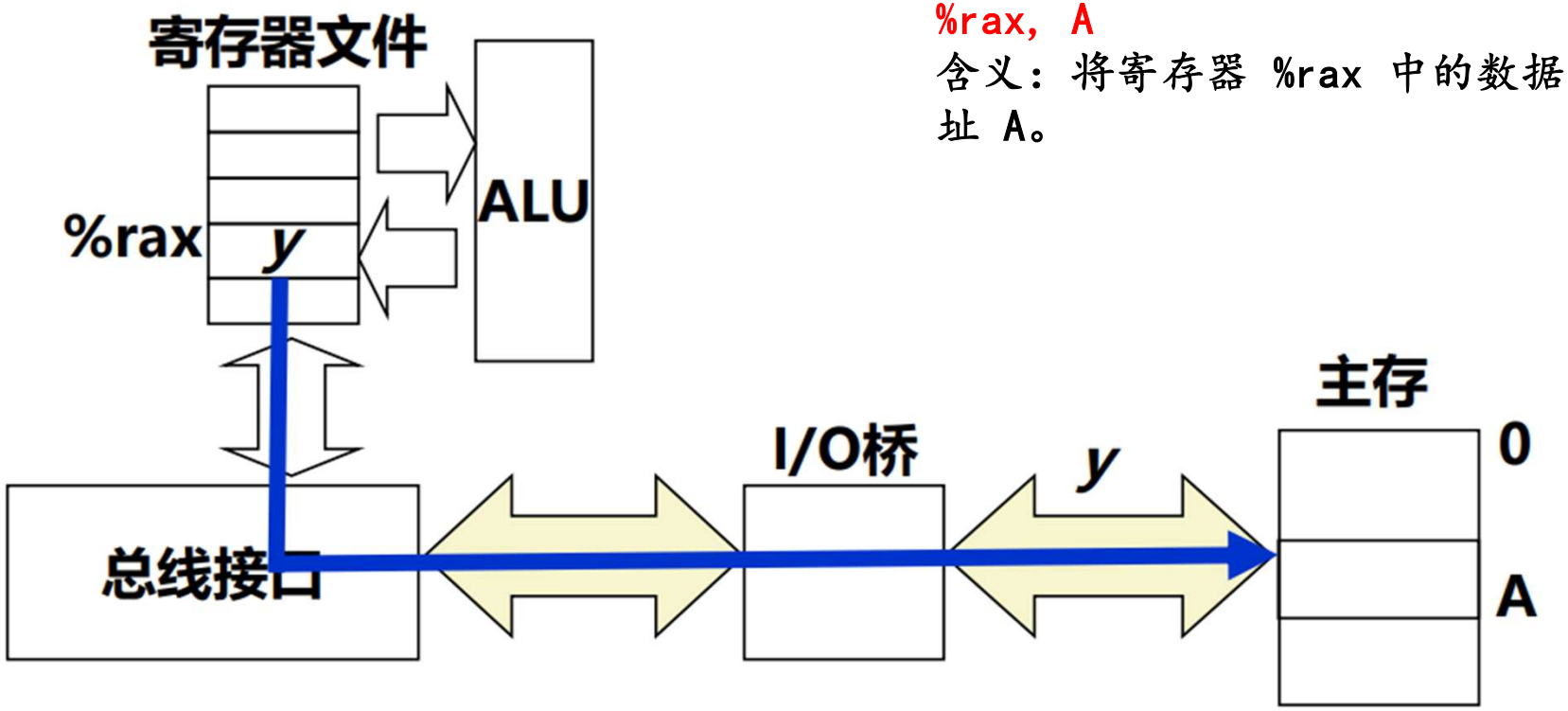
- CPU 将地址 A 放置在总线上。主存储器读取该地址，并等待接收对应的数据字。

存储操作（Store operation）：movq %rax, A
含义：将寄存器 %rax 中的数据写入内存地址 A。



内存写入(2)

- CPU 将数据字 y 放置在总线上。



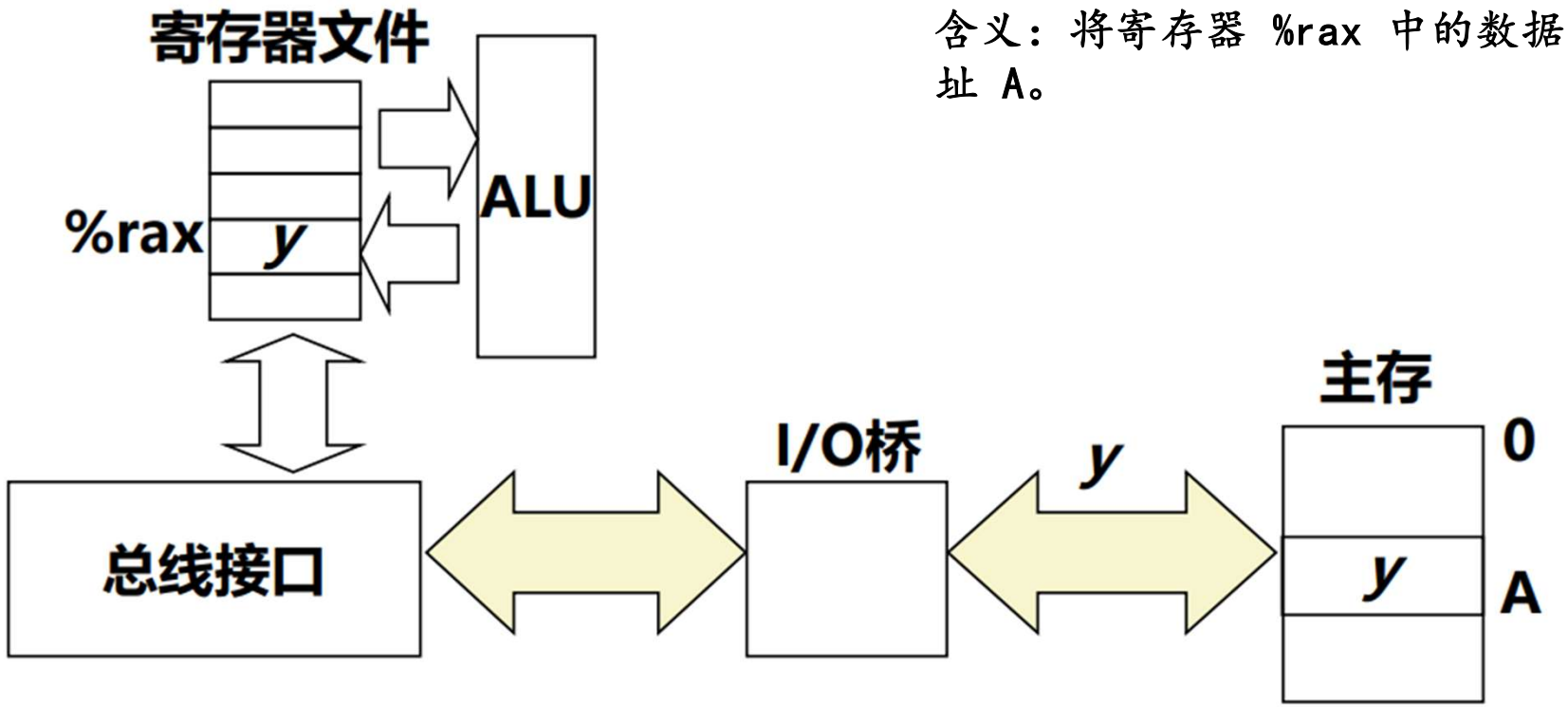
存储操作（Store operation）：movq %rax, A

含义：将寄存器 %rax 中的数据写入内存地址 A。

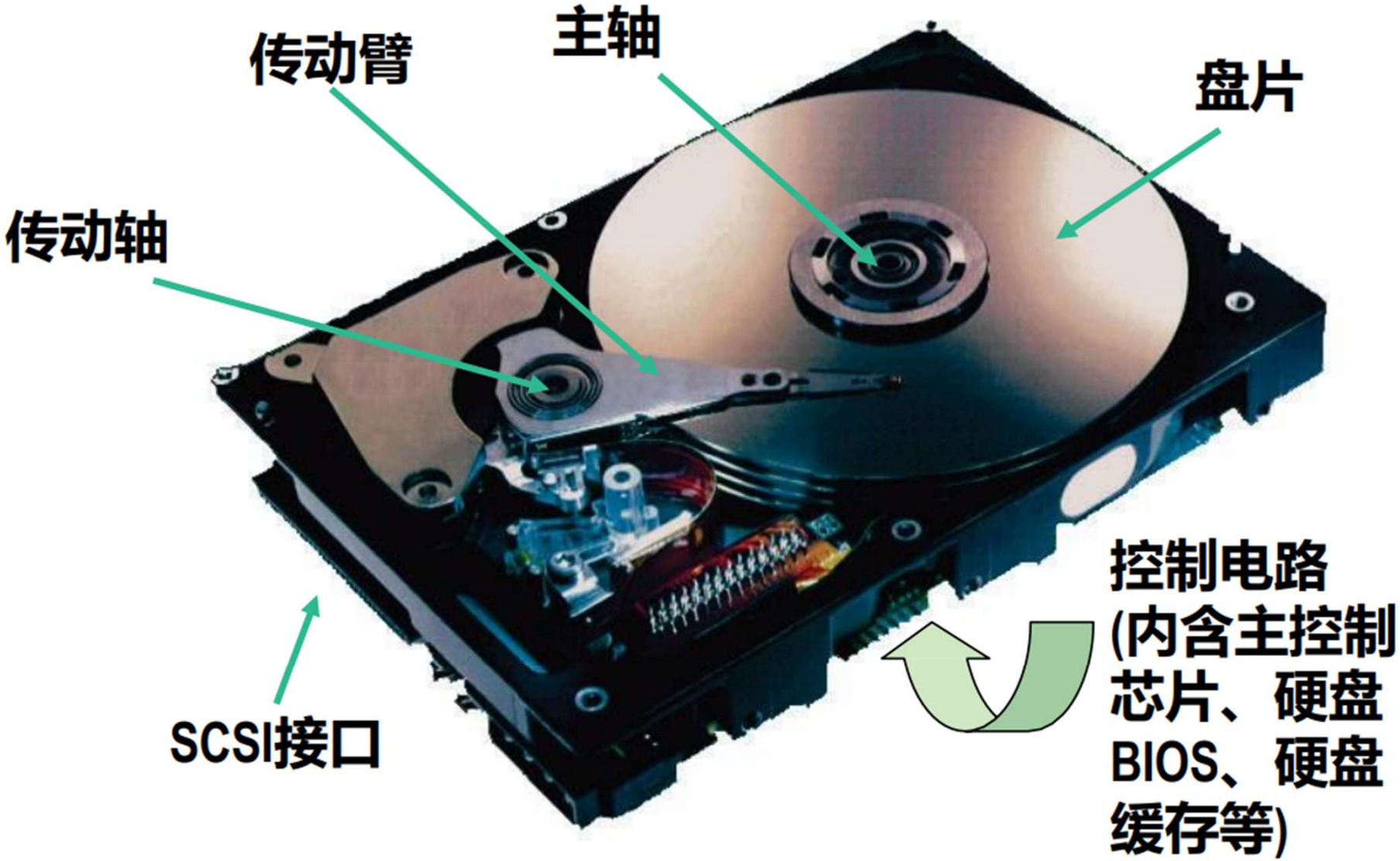
内存写入(3)

- 主存储器从总线上读取数据字 y，并将其存储到地址 A。

存储操作（Store operation）：movq %rax, A
含义：将寄存器 %rax 中的数据写入内存地址 A。

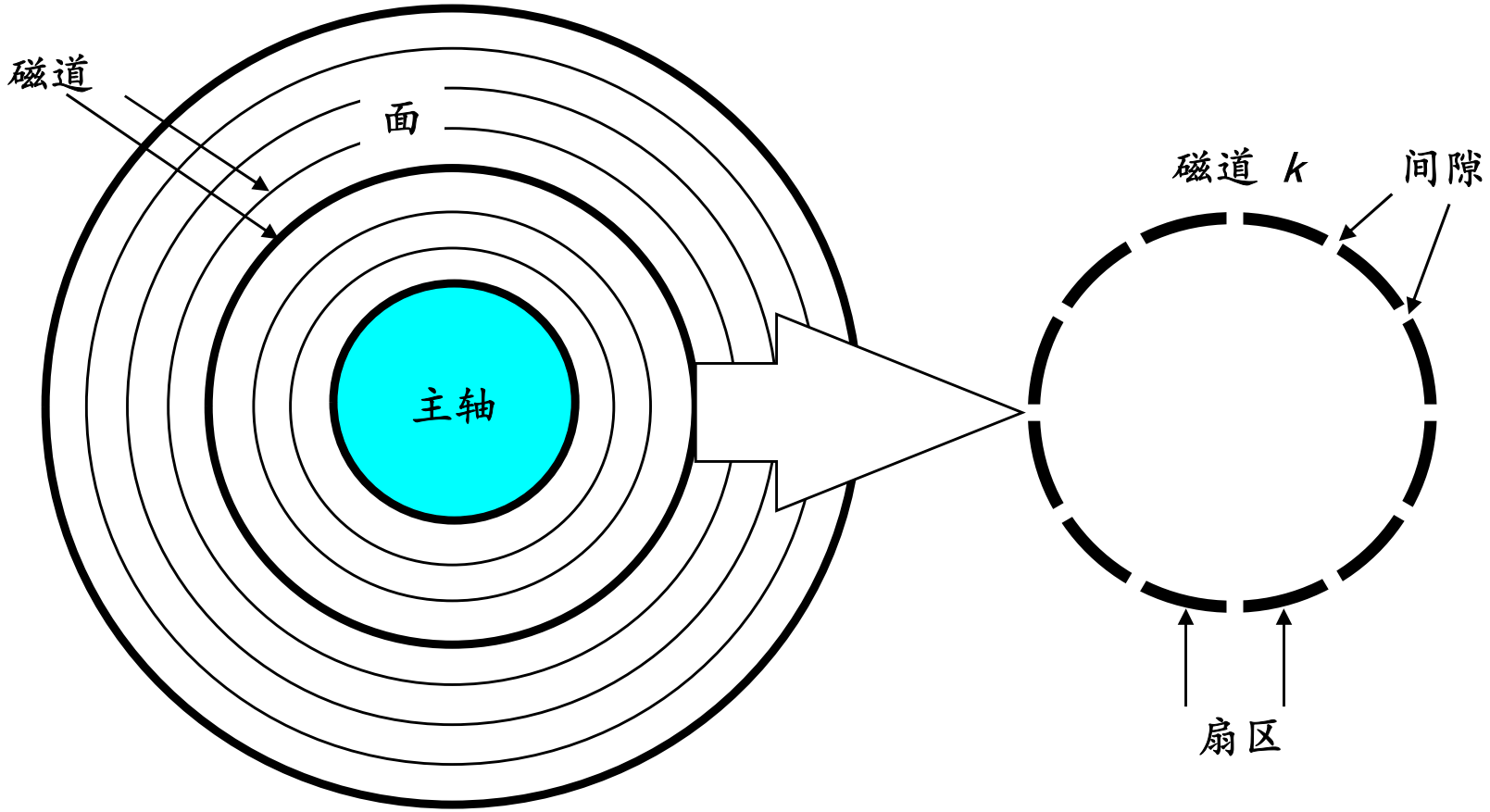


硬盘（HDD）内部结构



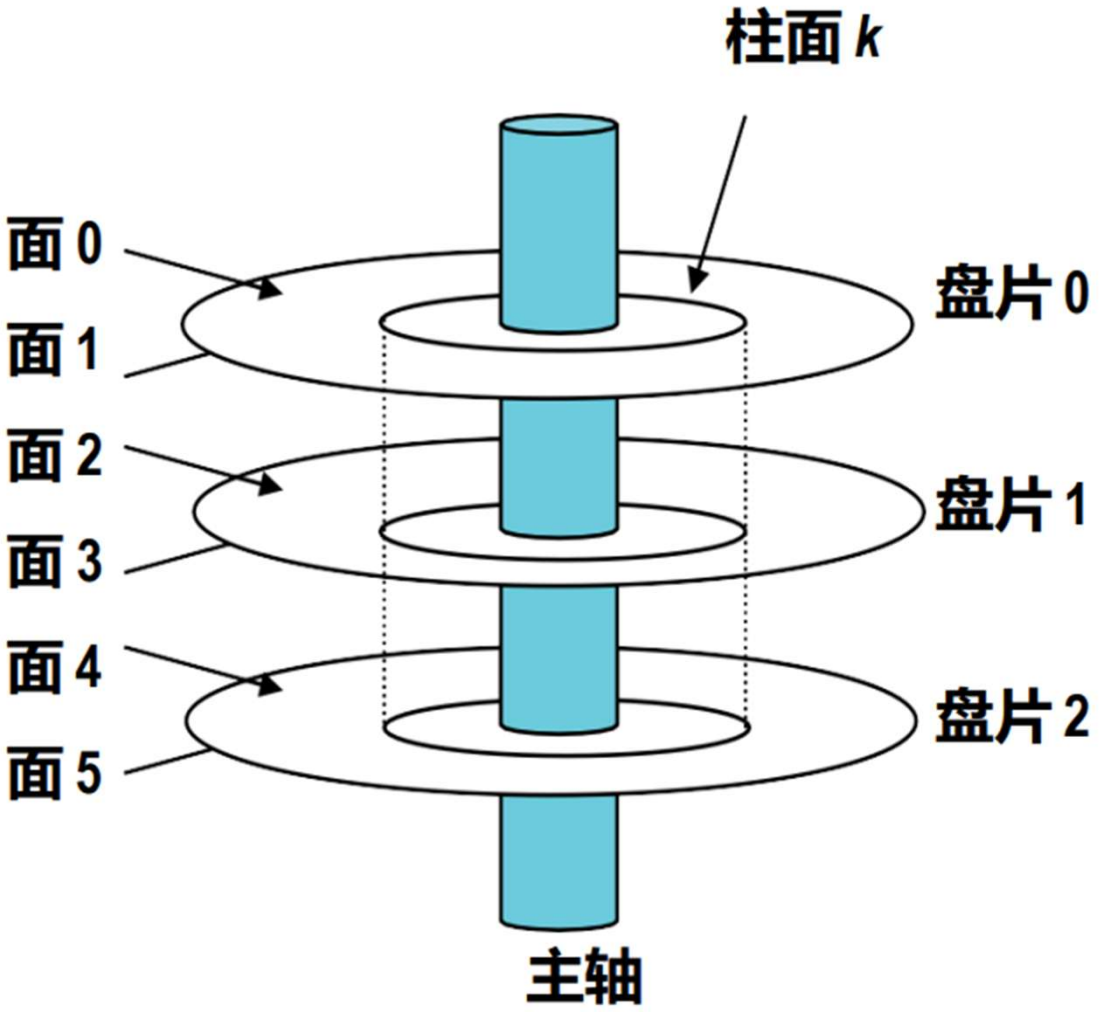
磁盘结构

- 磁盘由盘片(platter)构成，每个盘片包含两面(surface)。
- 每面由一组称为磁道(track)的同心圆组成。
- 每个磁道划分为一组扇区(sector)，扇区之间由间隙(gap)隔开。



磁盘结构(侧方多个盘片视角)

- 同一半径上的所有磁道组成一个柱面。

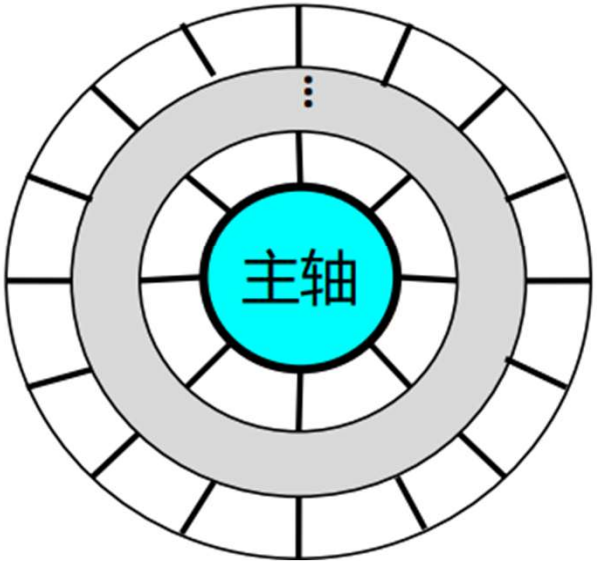


磁盘容量

- 容量(Capacity)：磁盘上可以存储的最大位(bits)数。
  - 制造商以千兆字节(GB)为单位来表达磁盘容量
  - 1 GB = 10^9 Bytes.
  - 硬盘厂商用 十进制（10^9），而操作系统一般使用 二进制（2^30）
  - 买了一块标称 500GB 的硬盘，厂商是按照十进制算的
    - 500 GB = 500 × 10^9 = 500,000,000,000 字节
  - 但当你插到电脑上时，操作系统会用二进制来显示：
    - 500,000,000,000 ÷ 2^30 ≈ 465 GB
- 磁盘容量由以下技术因素决定：
  - 记录密度(Recording density)（位/英寸）：磁道一英寸的段中可放入的位数。
  - 磁道密度(Track density)（道/英寸）：从盘片中心出发半径上一英寸的段内可以有的磁道数。
  - 面密度(Areal density)（位/平方英寸）：记录密度与磁道密度的乘积。

分区记录

- **现代磁盘将所有磁道划分为若干分组(recording zone)，组内各磁道相邻**
  - 区域内各磁道的扇区数目相同，扇区数取决于区域内最内侧磁道的圆周长
  - 各区域的每磁道扇区数都不同，外圈区域的每磁道扇区数比内圈区域多
  - 所以我们使用每磁道平均扇区数来计算磁盘容量。



计算磁盘容量

容量 = (字节数/扇区)
× (平均扇区数/磁道)
× (磁道数/面)
× (面数/盘片)
× (盘片数/磁盘)

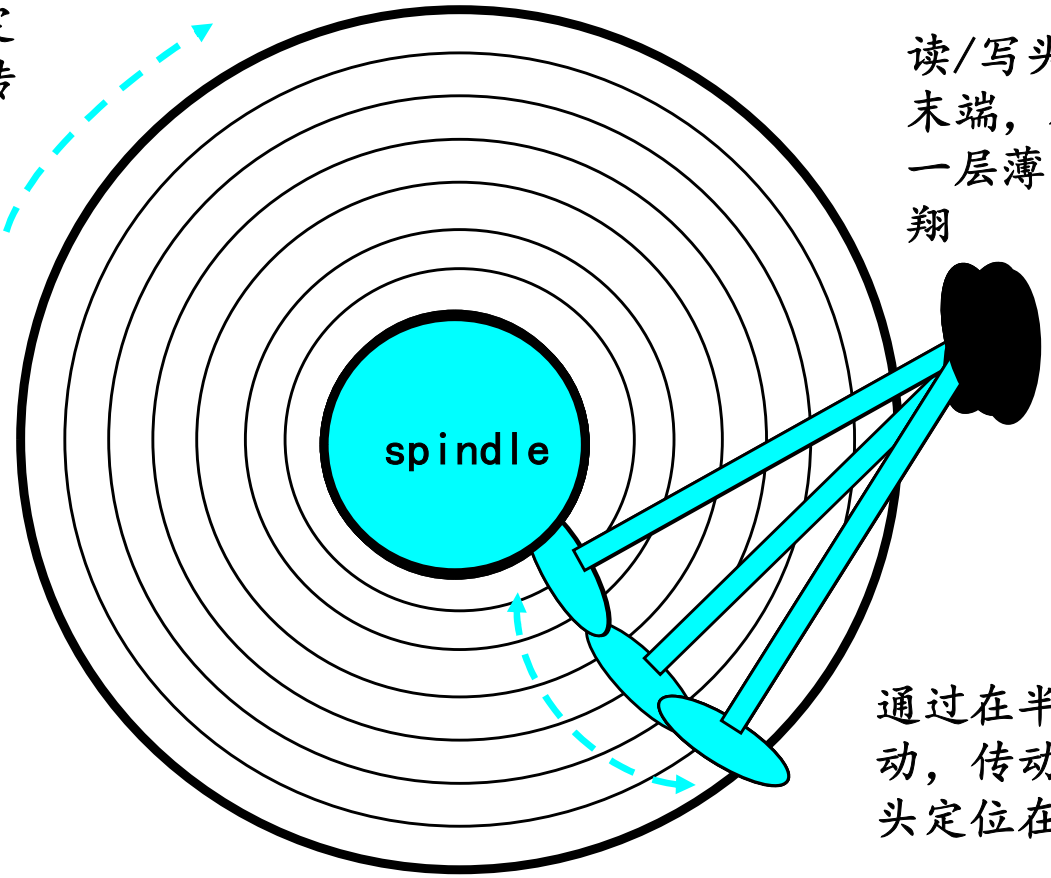
例子:

- 512 字节/扇区
- 300 扇区/磁道 (平均值)
- 20,000 磁道/面
- 2 面/盘片
- 5 盘片/磁道

容量 = 512 x 300 x 20000 x 2 x 5
= 30,720,000,000
= 30.72 GB

磁盘操作（单盘片视图）

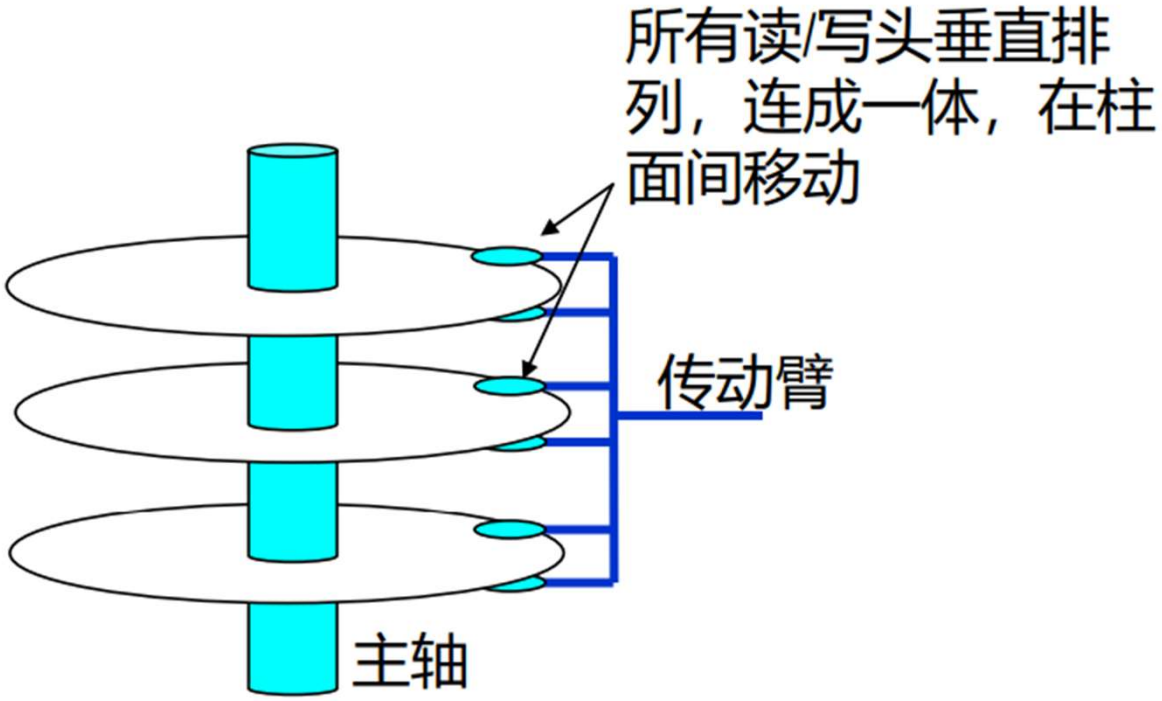
磁盘表面以固定的旋转速率旋转



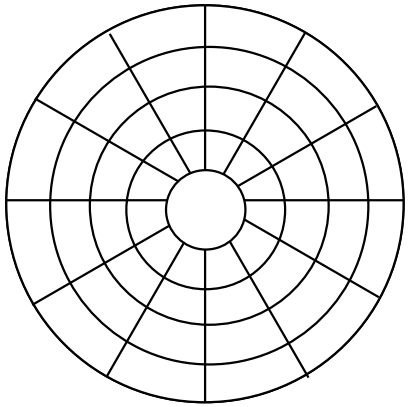
读/写头连到传动臂的末端，在磁盘表面上一层薄薄的气垫上飞翔

通过在半径方向上移动，传动臂可以将读/写头定位在任何磁道上.

磁盘操作(侧向多盘片视角)



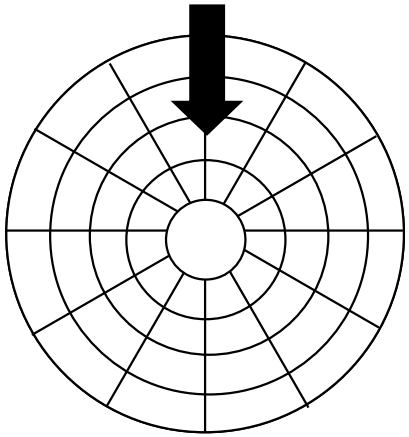
磁盘结构 - 单盘片俯视图



面由若干磁道构成

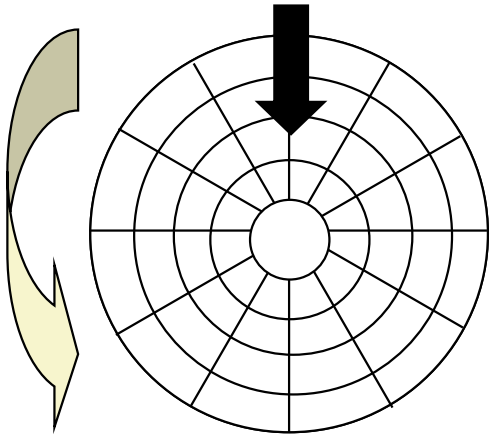
磁道被划分为若干扇区

磁盘访问



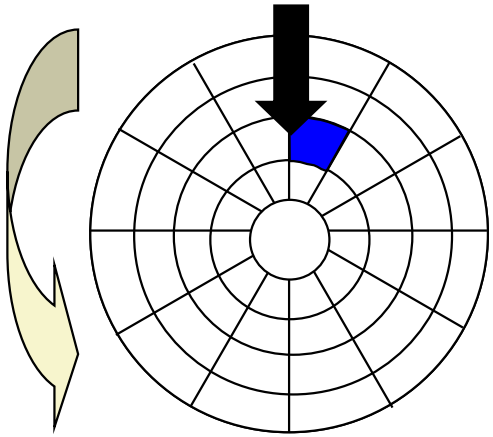
读/写头在磁道上方

磁盘访问



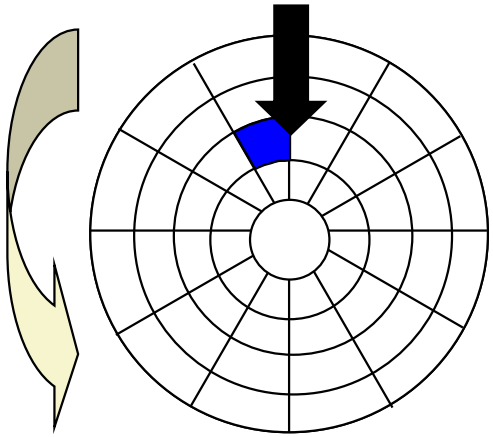
盘面逆时针旋转

磁盘访问 - 读



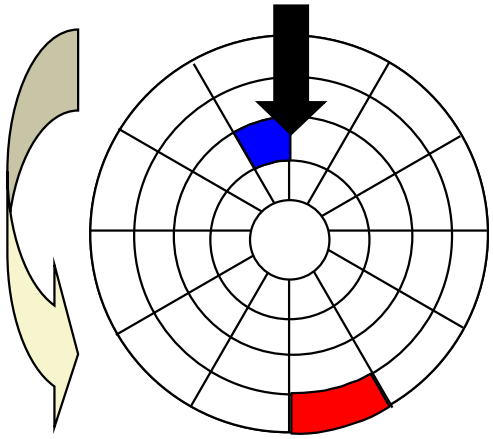
读取蓝色扇区

磁盘访问 - 读



读完蓝色扇区

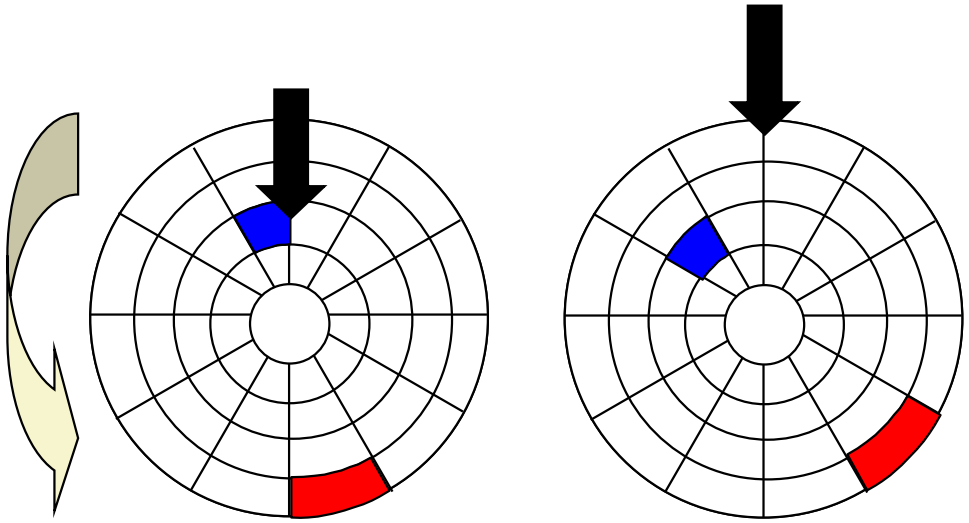
磁盘访问 - 读



读完蓝色扇区

请求读取红色扇区

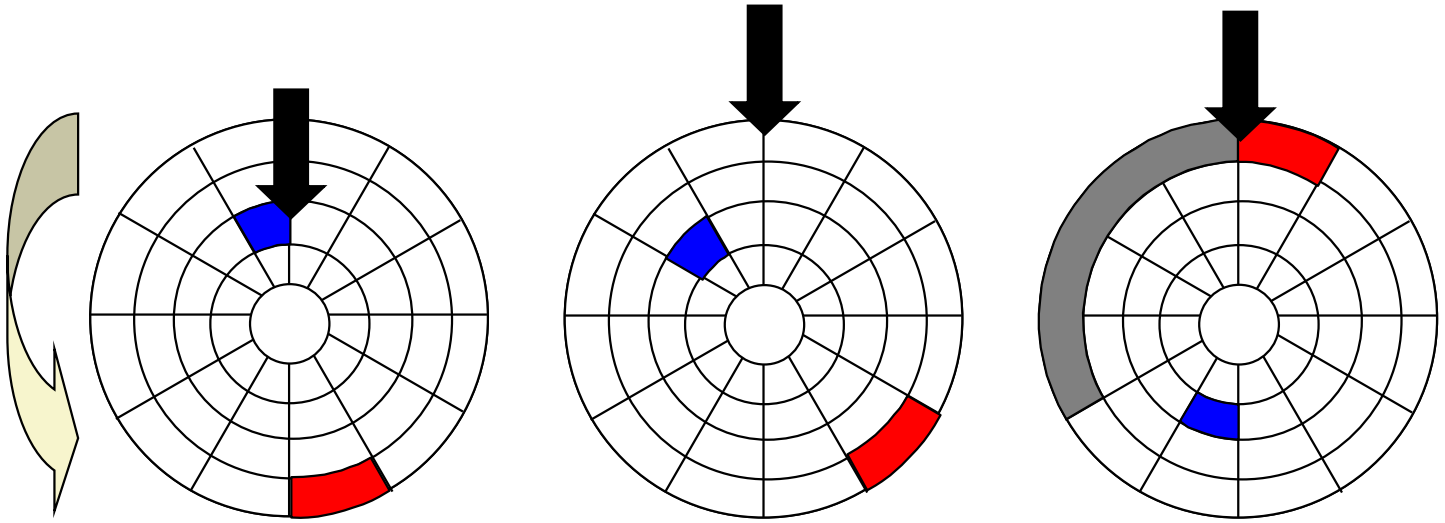
磁盘访问 - 寻道



读完蓝色扇区　　寻找红色扇区

寻找红色扇区所在磁道

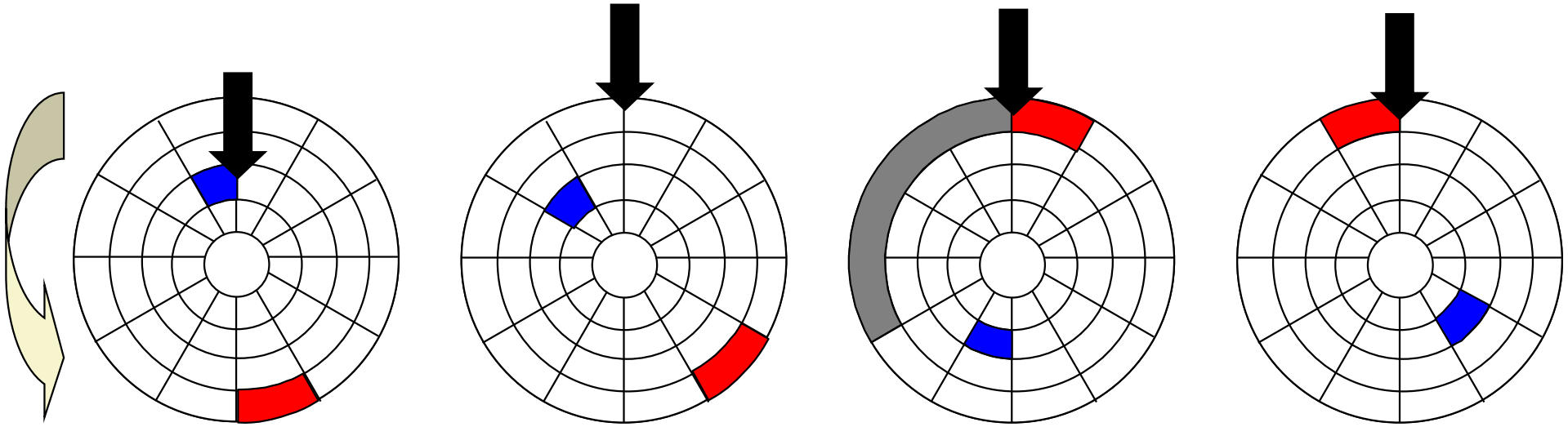
磁盘访问 - 旋转延迟



读完蓝色扇区　　寻找红色扇区　　旋转延迟

旋转盘面，使读/写头在红色扇区上方

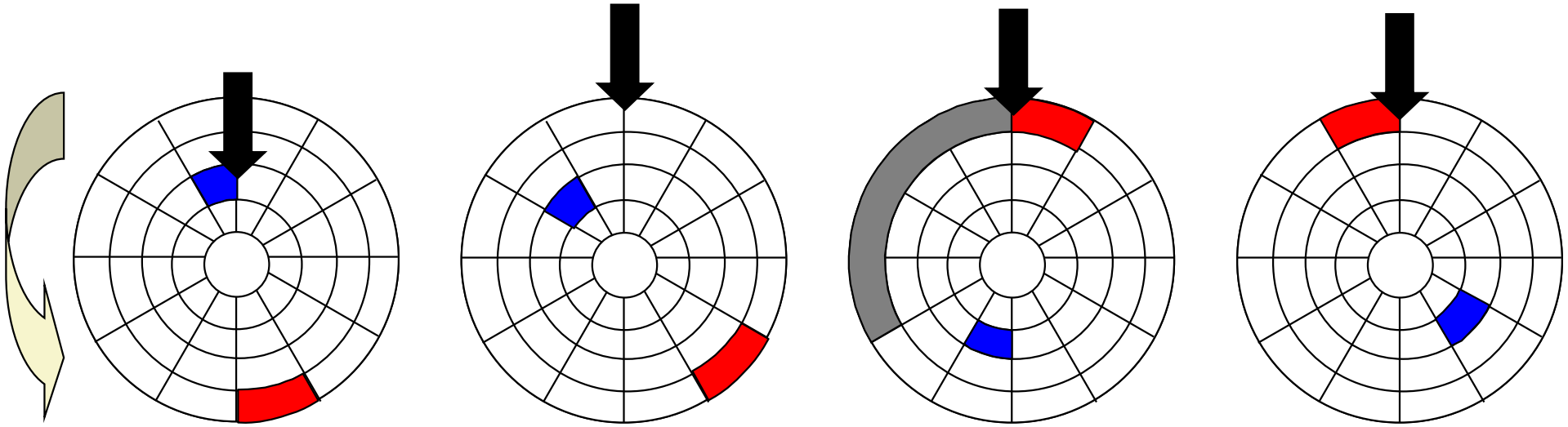
磁盘访问 - 读



读完蓝色扇区　寻找红色扇区　旋转延迟　读完红色扇区

完成读取红色扇区

磁盘访问 - 访问时间构成



磁盘访问时间

- **访问目标扇区的平均时间大致为**
  - 访问时间 = 寻道时间 + 平均旋转延迟 + 数据传输时间
- **寻道时间(Seek time)**
  - 读/写头移动到目标柱面所用时间
  - 通常寻道时间为：3—9 ms
- **旋转延迟(Rotational latency)**
  - 旋转盘面使读/写头到达目标扇区上方所用时间
  - 平均旋转延迟= 1/2 × 1/RPMs × 60 sec/1 min (RPM：转/分钟)
  - 通常 RPMs = 7,200
- **数据传输时间(Transfer time)**
  - 读目标扇区所用时间
  - 数据传输时间 = 1/RPM × 1/(平均扇区数/磁道) × 60 secs/1 min

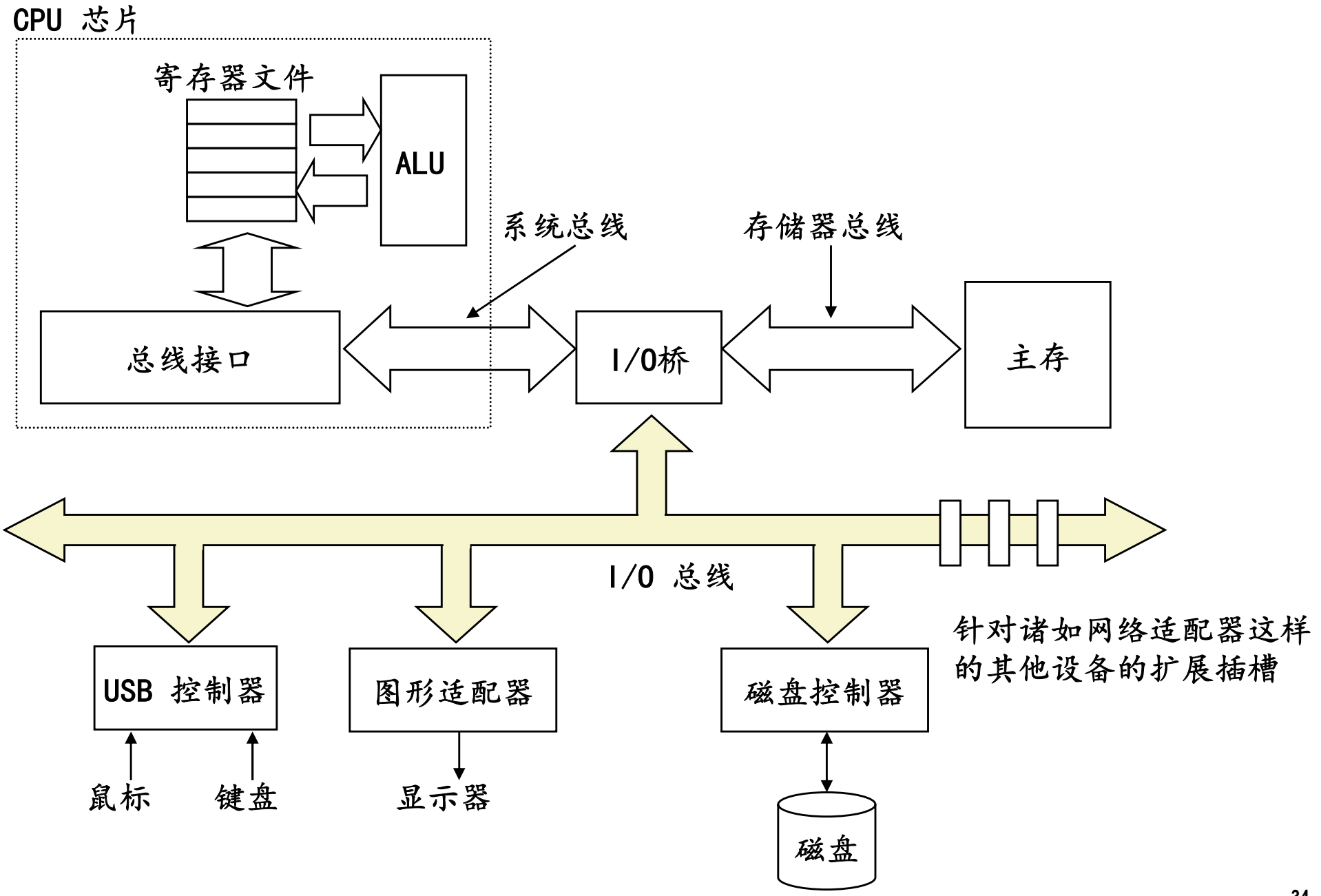
磁盘访问时间 – 举例

- 给定：
  - 旋转速度：7,200 RPM
  - 平均寻道时间：9 ms
  - 每磁道平均扇区数：400
- 推导：
  - 平均旋转延迟= 1/2 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms.
  - 平均传输时间= 60/7200 RPM x 1/400 secs/track x 1000 ms/sec = 0.02 ms
  - 访问时间 = 9 ms + 4 ms + 0.02 ms
- 要点：
  - 访问时间主要由寻道时间和旋转延迟主导。
  - 一个扇区中的第一位最昂贵，其余几乎“免费”。
  - SRAM 访问时间约为 4 ns/双字，DRAM 约为 60 ns。
    - 磁盘比 SRAM 慢约 40,000 倍，比 DRAM 慢约 2,500 倍。

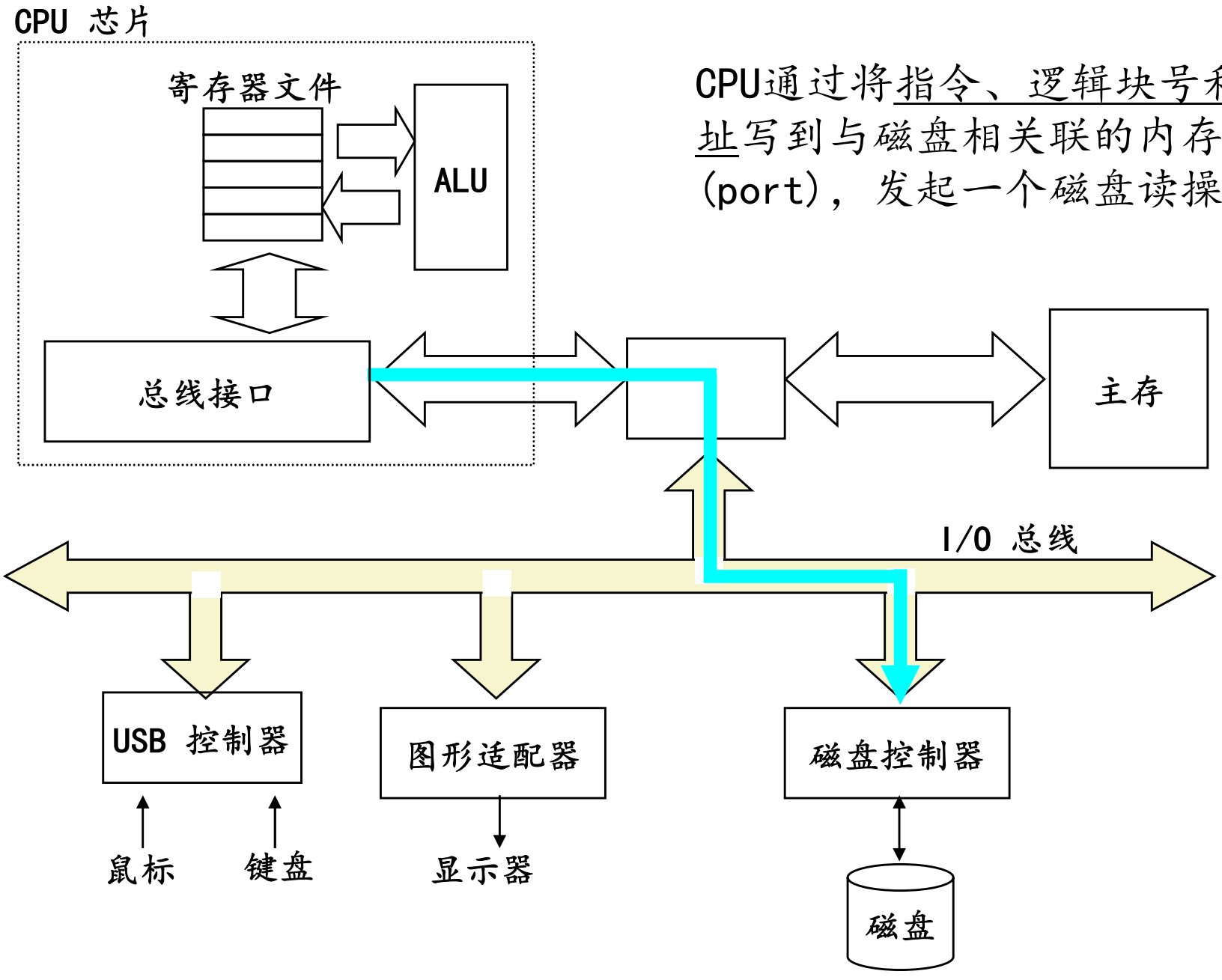
逻辑磁盘块（Logical Disk Blocks）

- 现代磁盘提供了复杂扇区几何结构的更简单抽象视图：
  - 可用扇区集合被建模为一系列大小为 b 的逻辑块（logical blocks），编号为 0、1、2、…
- 逻辑块与实际（物理）扇区之间的映射
  - 由称为磁盘控制器（disk controller）的硬件/固件设备维护。
  - 将对逻辑块的请求转换为（磁盘面，磁道，扇区）三元组。
- 允许控制器为每个区域预留备用柱面
  - 这解释了“格式化容量”与“最大容量”之间的差异。

I/O 总线 （I/O Bus）

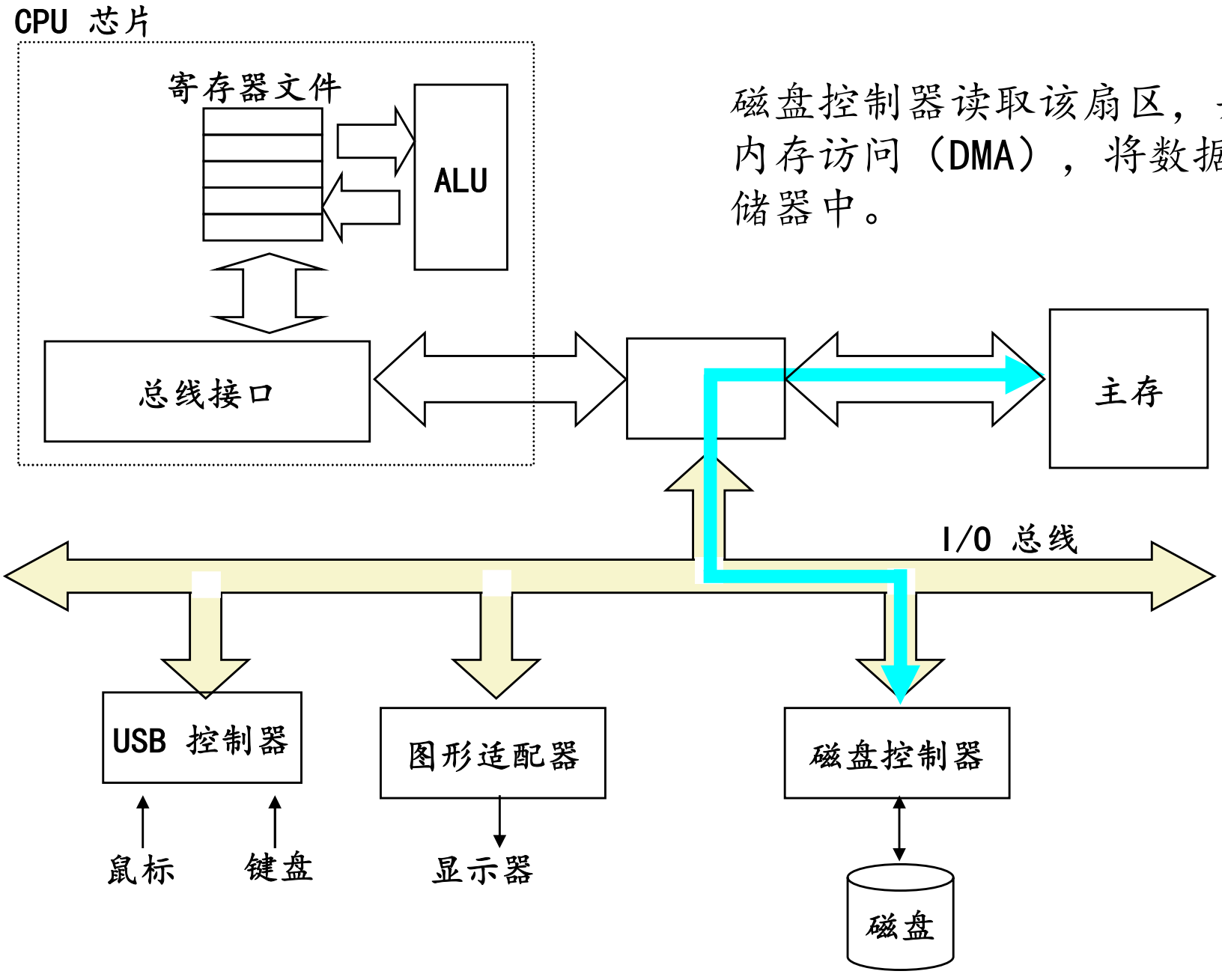

CPU 芯片
寄存器文件
ALU
系统总线
存储器总线
总线接口
I/O桥
主存
I/O 总线
USB 控制器
图形适配器
磁盘控制器
针对诸如网络适配器这样的其他设备的扩展插槽
鼠标
键盘
显示器
磁盘

读磁盘扇区(1)



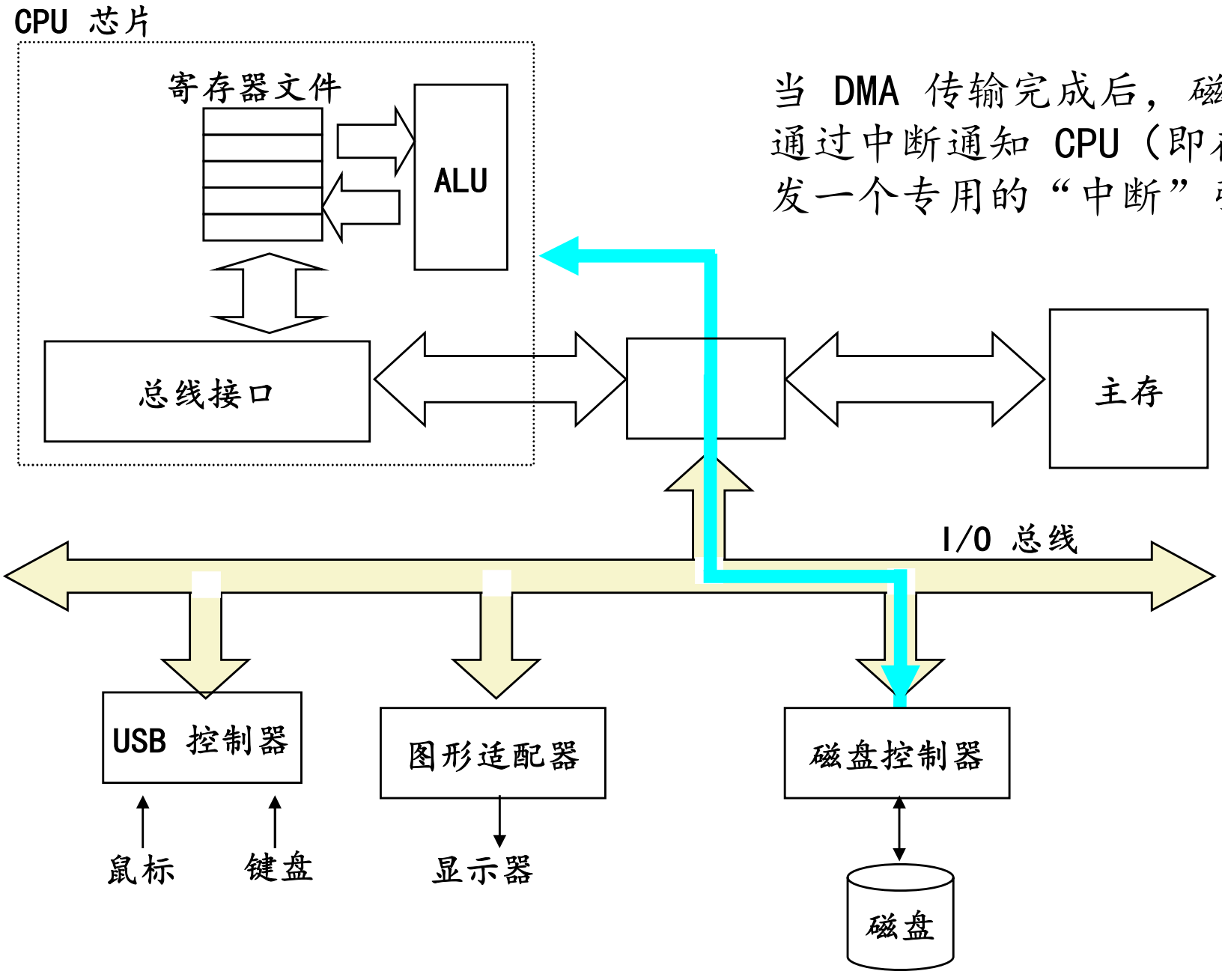
CPU通过将指令、逻辑块号和目的内存地址写到与磁盘相关联的内存映射 地址(port)，发起一个磁盘读操作。

读磁盘扇区(2)



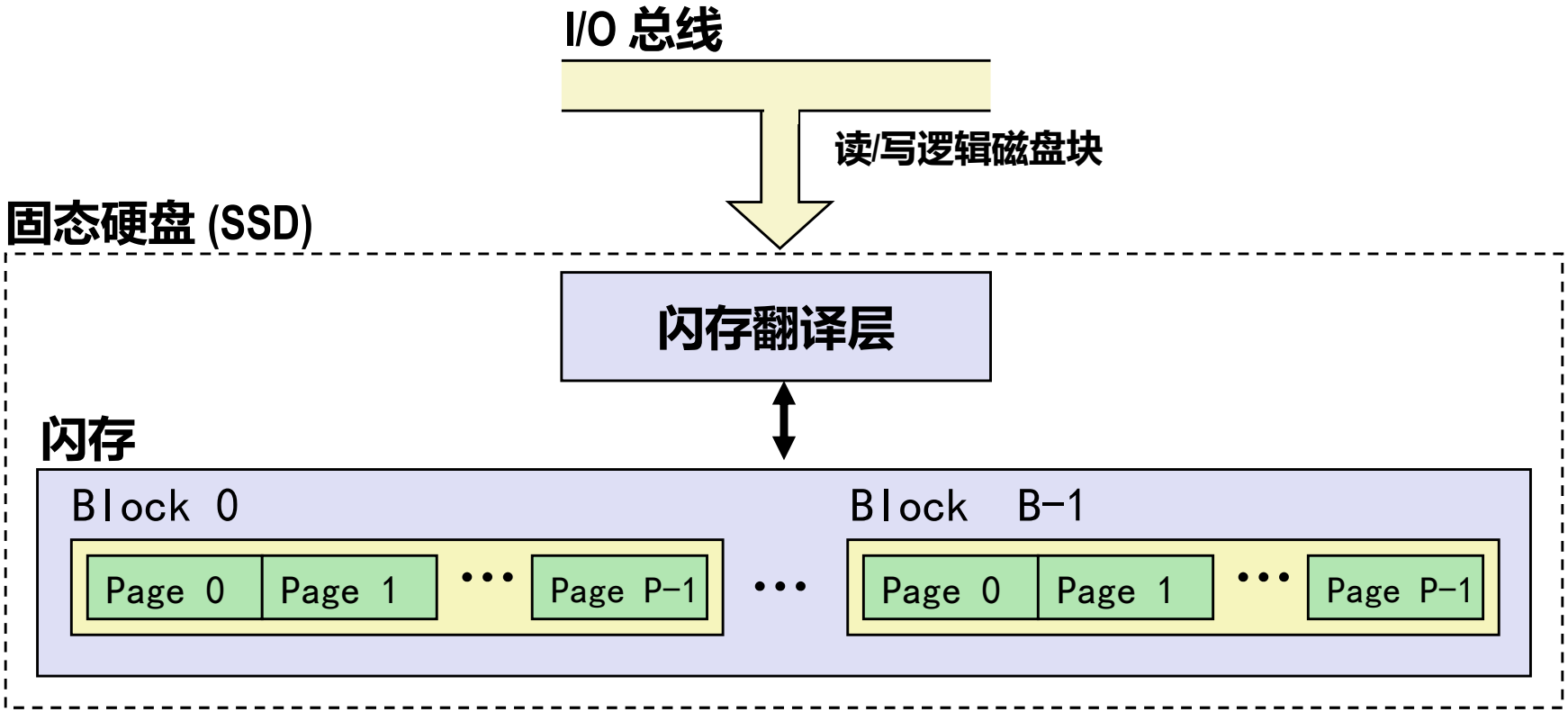
磁盘控制器读取该扇区，并执行直接内存访问（DMA），将数据传输到主存储器中。

读磁盘扇区(3)



当 DMA 传输完成后，磁盘控制器会通过中断通知 CPU（即在 CPU 上触发一个专用的“中断”引脚）。

固态硬盘（SSDs）



- 页面大小：4KB 到 512KB
- 每个块（Block）包含：32 到 128 个页面（Pages）
- 数据的读写单位是页面（Page）。
- 页面只能在所在的块被擦除后才能写入。
- 一个块在大约 100,000 次重复写入后会磨损失效。

SSD 性能特性

顺序读吞吐量	550 MB/s	顺序写吞吐量	470 MB/s
随机读吞吐量	365 MB/s	随机写吞吐量	303 MB/s
平均顺序读访问时间	50 us	平均顺序写访问时间	60 us

- 顺序访问比随机访问快
  - 典型存储器层次结构问题
- 随机写较慢
  - 擦除块需要较长的时间(~1ms)
  - 修改一页需要将块中所有页复制到新的块中
  - 早期SSD 读/写速度之间的差距更大

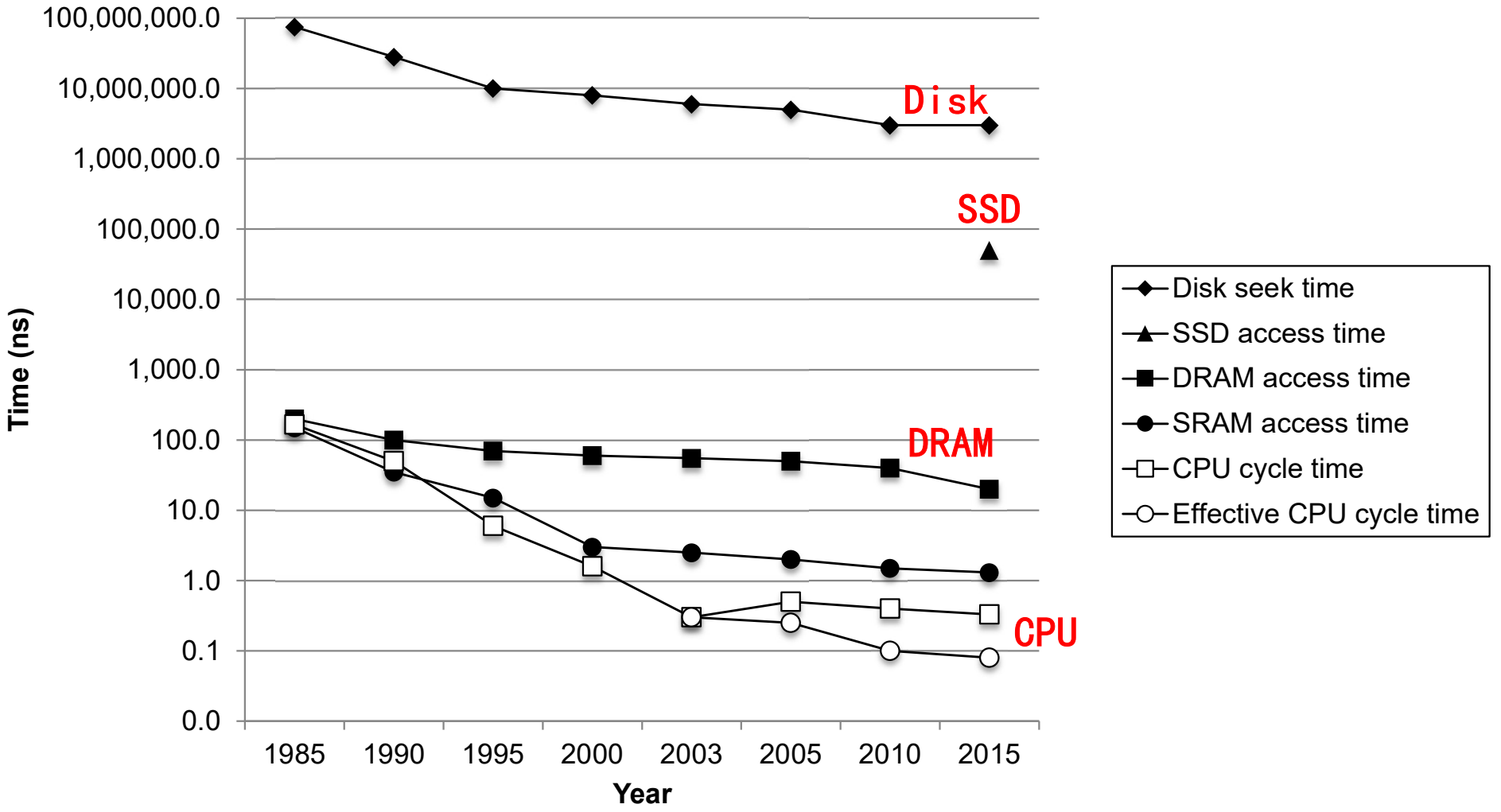
资料: Intel SSD 730 product specification.

SSD vs 机械磁盘

- 优点
  - 没有移动部件 → 更快、能耗更低、更结实（抗震）
- 缺点
  - 会磨损
  - 2015年，SSD每字节比机械磁盘贵大约30倍，目前约10倍
- 应用
  - 智能手机、笔记本电脑
  - 台式机和服务器中应用

CPU-储存器的速度差距

DRAM、磁盘和CPU之间的速度差距在变大.



局部性拯救一切！

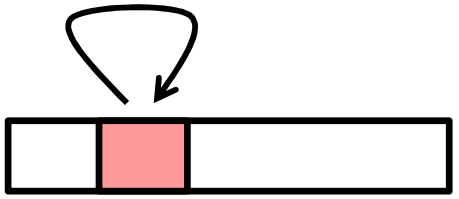
弥合 CPU 与内存速度差距 的关键，是计算机程序的一个基本特性，称为局部性（Locality）。

主要内容

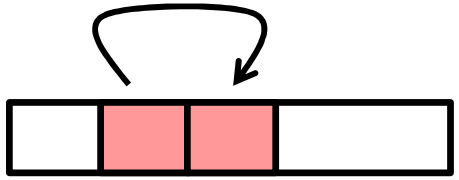
- 存储技术与发展趋势
- 局部性
- 内存层次结构中的缓存机制

局部性

- 局部性原理(Principle of Locality):
  - 程序倾向于使用距离最近用过的指令/数据地址相近或相等的指令/数据。



- 时间局部性(Temporal locality) :
  - 最近访问过的信息，很可能在近期还会被再次访问



- 空间局部性(Spatial locality):
  - 地址接近的数据项，被使用的时间也倾向于接近

局部性举例

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- 对数据的引用
  - 顺序访问数组元素(步长为1的引用模式) 空间局部性
  - 变量sum在每次循环迭代中被引用一次. 时间局部性
- 对指令的引用
  - 顺序读取指令 空间局部性
  - 重复循环执行for循环体 时间局部性

局部性举例

- Claim:能够通过阅读代码并定性判断其局部性，是专业程序员的一项关键技能。
- Question:对于数组 a 而言，这个函数的局部性表现是否良好?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

局部性举例

- Question:对于数组 a 而言，这个函数的局部性表现是否良好?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

存储器层次结构

- 一些硬件与软件的基本且长期不变的特性：
  - 更快的存储技术每字节成本更高、容量更小，并且需要更多的功耗（产生更多热量）。
  - CPU 与主存速度之间的差距正在不断扩大。
  - 编写良好的程序往往表现出较好的局部性。
- 这些基本特性相互完美互补。
- 它们共同启发了一种用于组织内存与存储系统的方法，称为 存储层次结构（memory hierarchy）。

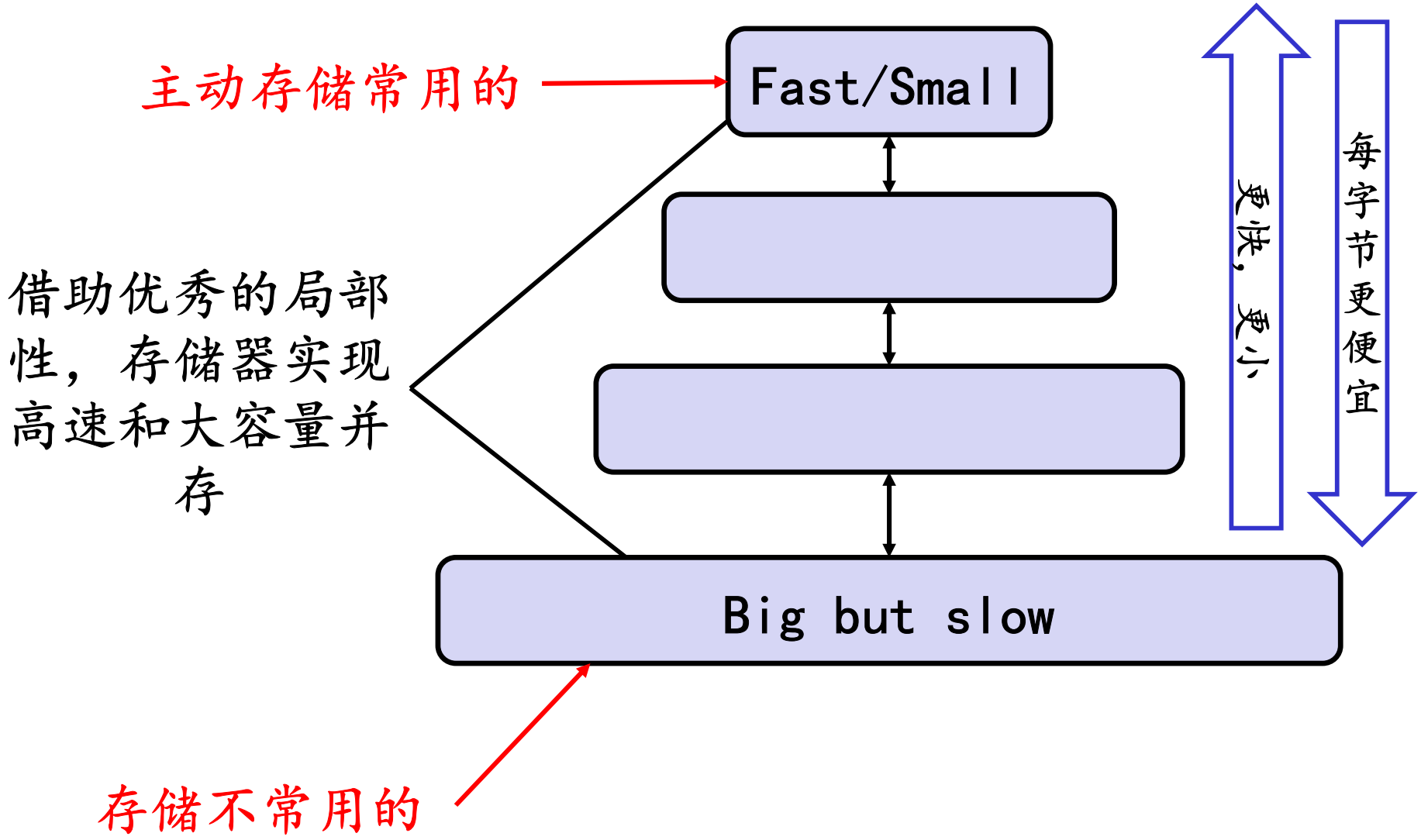
主要内容

- 存储技术与发展趋势
- 引用的局部性
- 内存层次结构中的缓存机制

为什么要有分层的Memory？

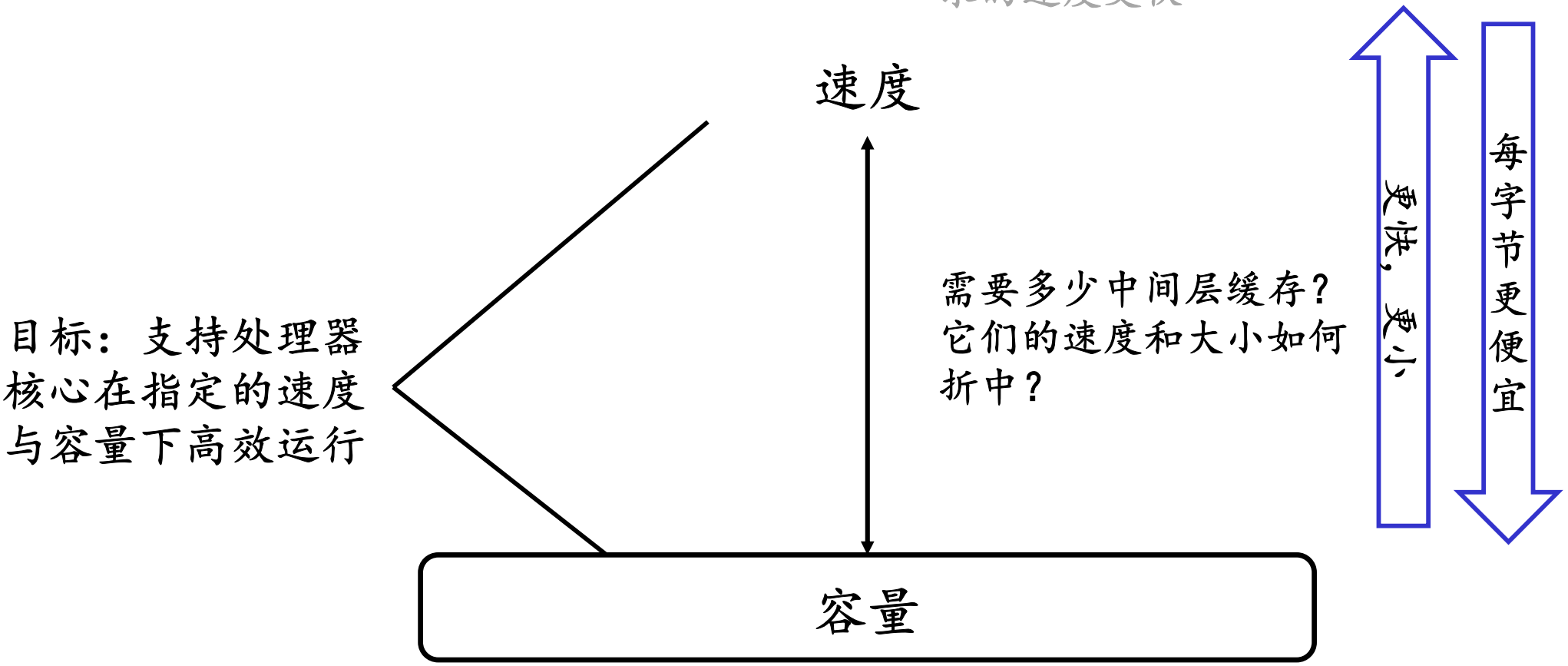
- 既高速存取，又要大容量存储
- 单一存储介质无法兼顾这两个需求
- 想法：多层的存储结构（根据层级不同从处理器访问时容量逐渐增大，访问逐渐变慢），并确保处理器需要的大部分数据保持在较快的层级

存储器层次结构


主动存储常用的
Fast/Small
借助优秀的局部性，存储器实现高速和大容量并存
Big but slow
存储不常用的
更快，更小
每字节更便宜

存储器层次结构设计

如何保证第一级内存（Level 1 memory）的速度和容量？必须比处理器要求的速度更快

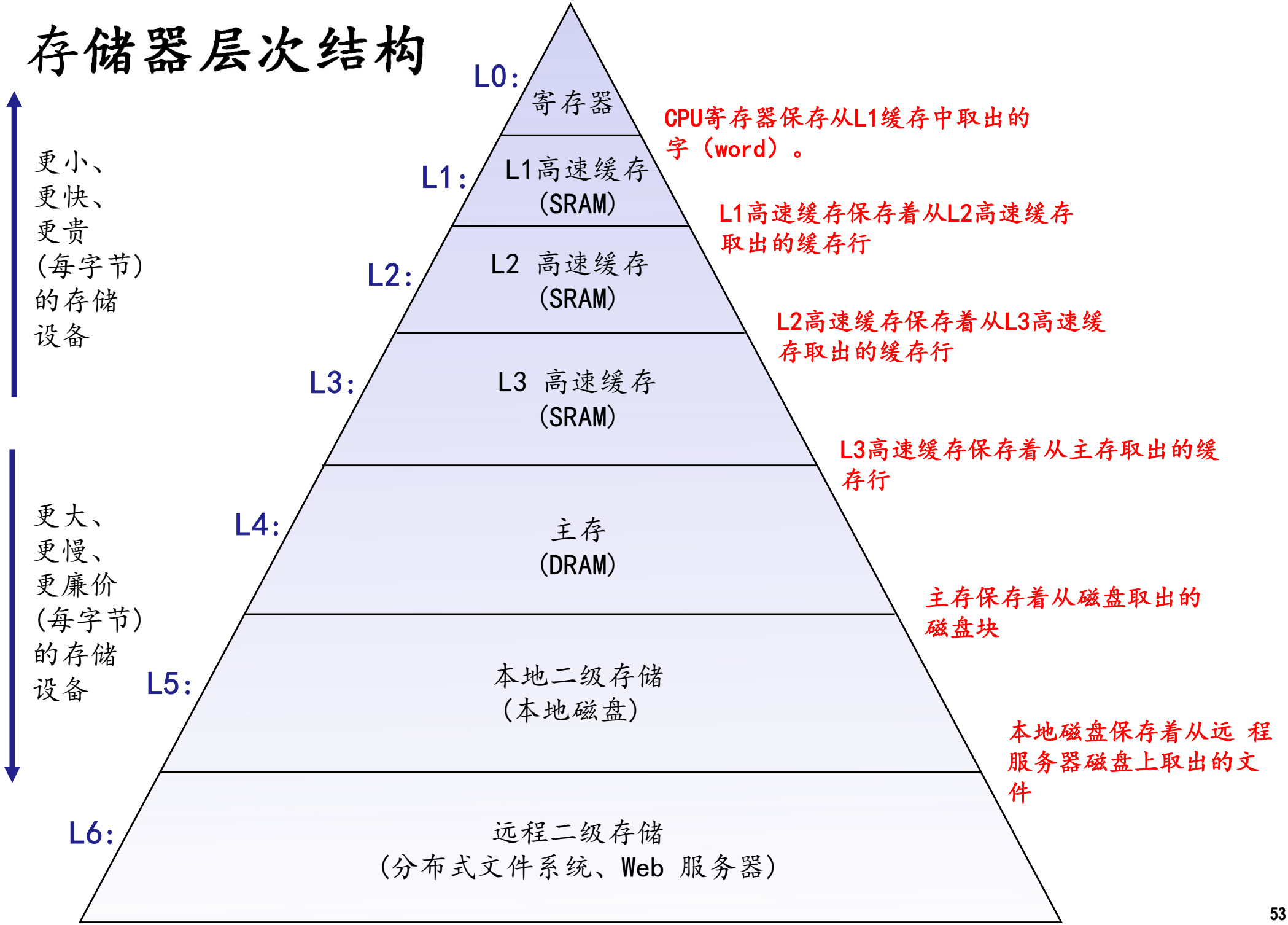


目标：支持处理器核心在指定的速度与容量下高效运行

需要多少中间层缓存？它们的速度和大小如何折中？

如何设计最后一级内存（Last Level memory）的速度和容量？容量必须足够大以满足数据需求

存储器层次结构


更小、
更快、
更贵
(每字节)
的存储
设备
更大、
更慢、
更廉价
(每字节)
的存储
设备
L0:
寄存器
CPU寄存器保存从L1缓存中取出的字（word）。
L1:
L1高速缓存
(SRAM)
L1高速缓存保存着从L2高速缓存取出的缓存行
L2:
L2 高速缓存
(SRAM)
L2高速缓存保存着从L3高速缓存取出的缓存行
L3:
L3 高速缓存
(SRAM)
L3高速缓存保存着从主存取出的缓存行
L4:
主存
(DRAM)
主存保存着从磁盘取出的磁盘块
L5:
本地二级存储
(本地磁盘)
本地磁盘保存着从远 程服务器磁盘上取出的文件
L6:
远程二级存储
(分布式文件系统、Web 服务器)

缓存

- 缓存（Cache）：一种更小、更快的存储设备，充当较大且更慢设备中部分数据的缓冲区。

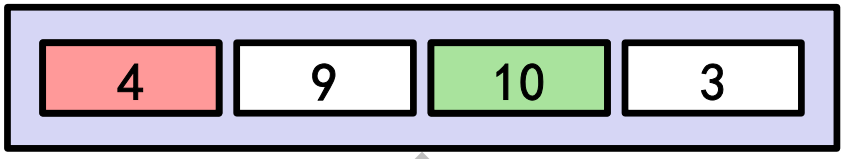
- 存储层次结构的基本思想：
  - 在每一层 k，更快、更小的存储设备作为其下一层 k+1（更大、更慢设备）的缓存。

- 为什么存储层次结构有效？
  - 由于局部性原理，程序往往更频繁地访问第 k 层的数据，而较少访问第 k+1 层的数据。
  - 因此，第 k+1 层的存储可以更慢，从而更大且每比特成本更低。

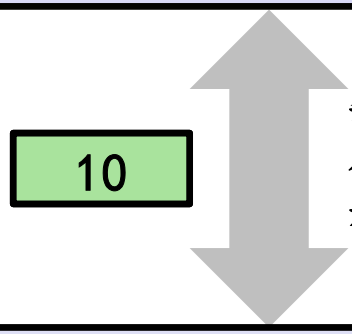
- 核心思想：存储层次结构创建了一个逻辑上巨大的存储池，成本接近底层廉价存储，性能接近顶层高速存储，既兼顾容量，又兼顾速度。

缓存的基本概念

缓存

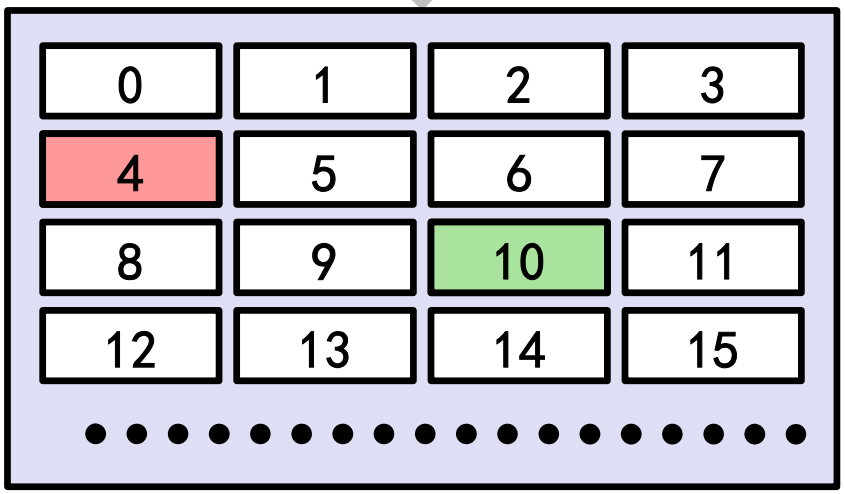


更小、更快、但更昂贵的高速缓存（cache），用于缓存较大、较慢且更便宜存储器中的部分数据块



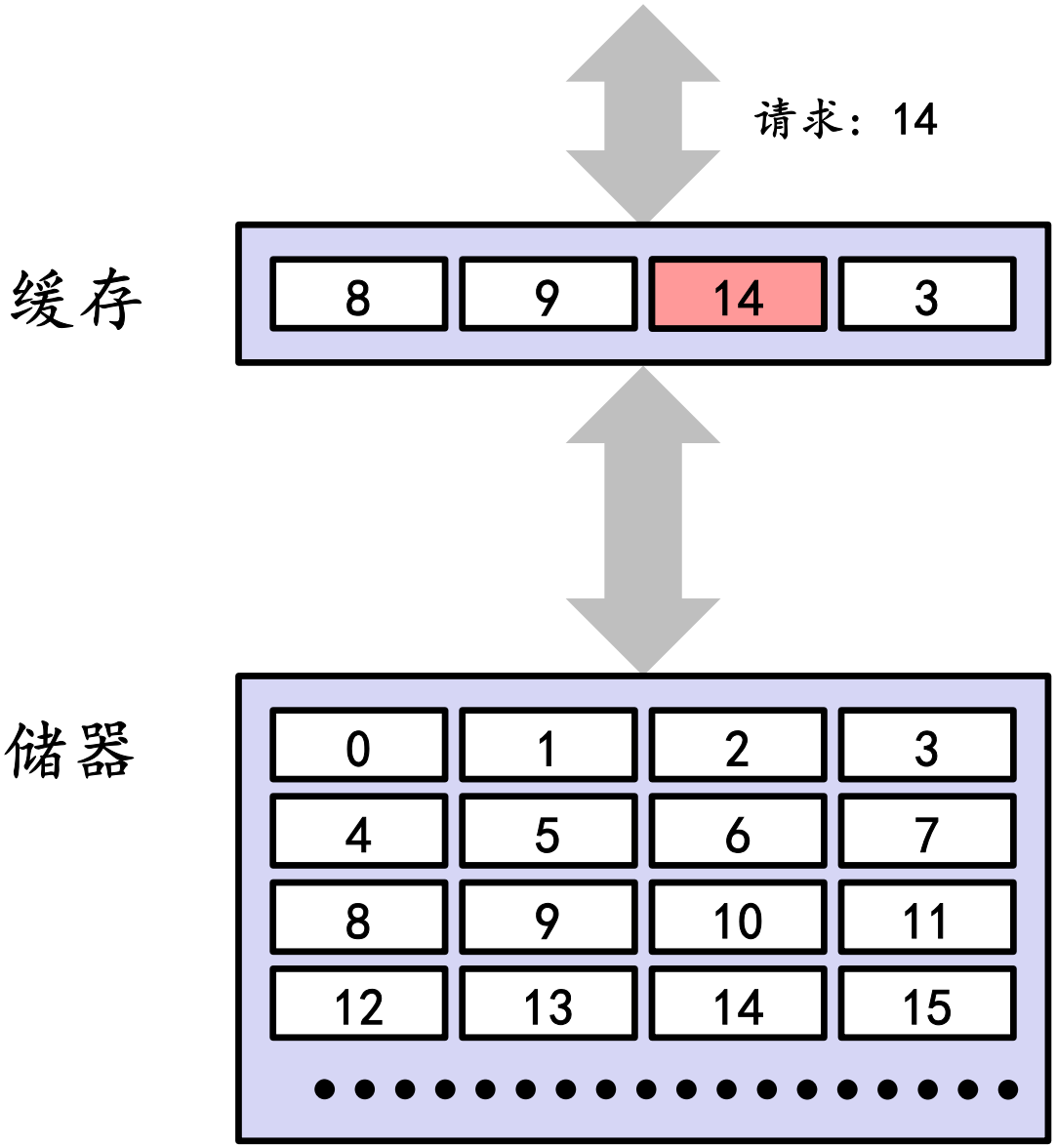
数据以块（block）**为单位在不同层级的存储之间传输和复制。

存储器



大容量、低速存储器可以被看作是划分成若干块（blocks）的集合。

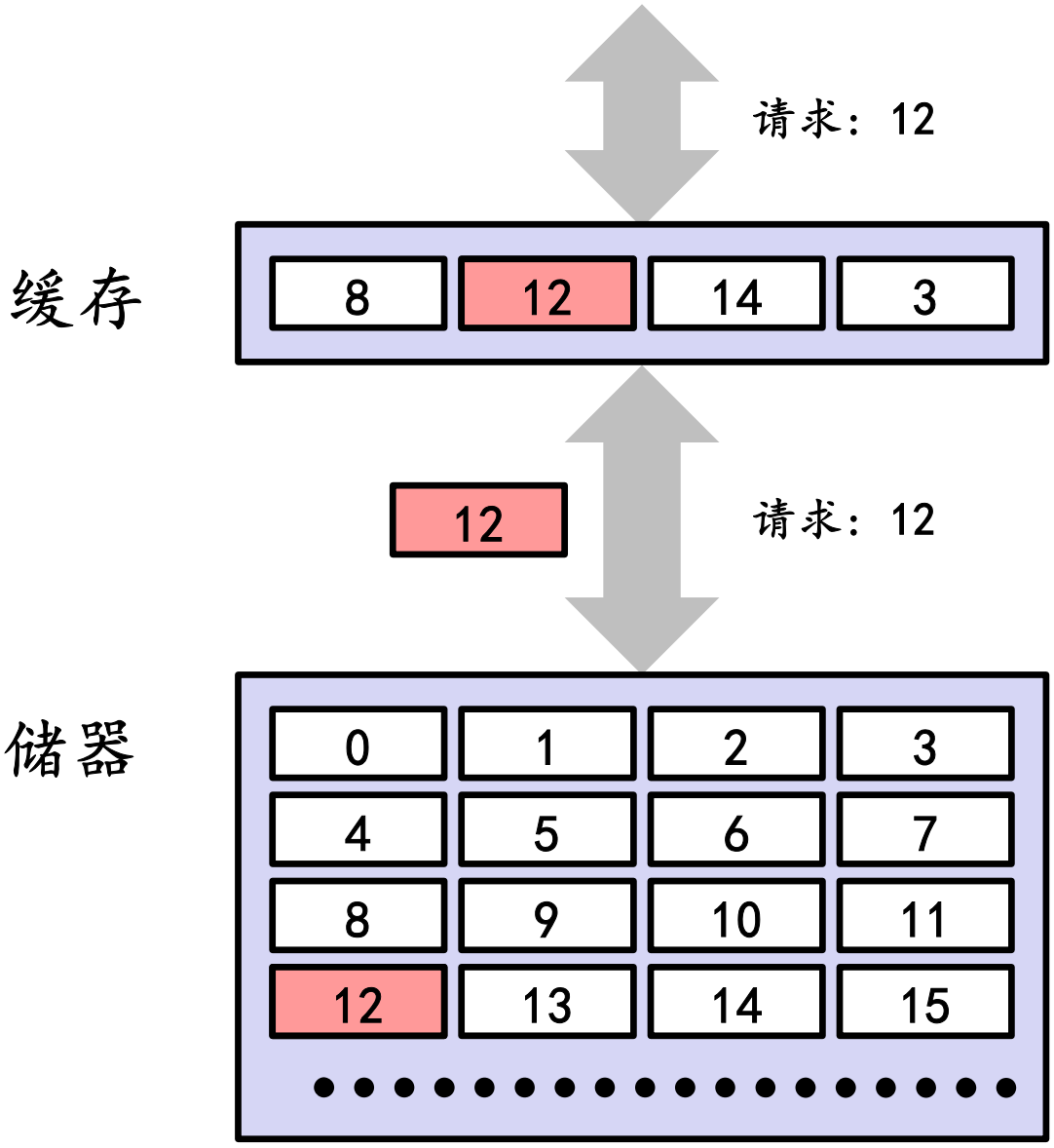
缓存的基本概念：命中



请求块 b 中的数据

块 b 在缓存中：命中！

缓存的基本概念：未命中（Miss）



请求块 b中的数据

块 b 不在缓存中：
未命中！

从存储器中取出块 b

块b存储在缓存中

- 放置策略(Placement policy)：决定块b放在缓存中的位置
- 替换策略(Replacement policy)：决定该替换存储器中的哪一块

缓存基本概念:缓存不命中的种类

- 冷未命中（Cold Miss / Compulsory Miss）
  - 由于缓存一开始是空的，第一次访问某个数据块时，缓存中必然没有该数据。
- 冲突未命中（Conflict Miss）
  - 原因：大多数缓存会将上一级存储（k+1 层）的数据块限制在下一级缓存（k 层）中特定位置，有时甚至是唯一位置。
  - 示例：如果第 k+1 层的 块 i 必须映射到第 k 层的 块（i mod 4），那么不同的块如果映射到相同的缓存位置，就会发生冲突未命中。
  - 案例：依次访问块 0，8，0，8，0，8，... 时，由于块 0 和块 8 在缓存中映射到同一位置，每次都会未命中。
- 容量未命中（Capacity Miss）
  - 当程序的工作集（working set）中需要同时活跃的数据块数量大于缓存容量时，即使映射没有冲突，仍然会因为空间不足而发生未命中。

存储器层次结构管理

- 在不同层次之间显式、手动地复制数据
  - 真空管 vs 选择管（Selectron）
  - “磁芯” vs “磁鼓”内存（20世纪50年代）
  - “暂存区”（scratchpad）SRAM：用于现代嵌入式系统和数字信号处理（DSP）
  - 寄存器文件（Register file） 也是存储层次结构的一个级别
- 单一地址空间，自动管理
  - 早在 1962 年的 ATLAS 计算机上就已实现
  - 在如今搭配慢速 DRAM 的高速处理器中非常常见
  - 对于典型程序来说，程序员不需要关心这些细节，就能让程序既快又正确
- 那么，非典型程序又该怎么办呢?

存储器层次结构中的缓存例子

缓存类型	缓存什么	存在何处	延迟(周期数)	由谁管理
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

层次结构的设计自由度

- DRAM
  - 针对容量/成本比进行优化（容量越大，性价比越高）。
  - T_DRAM（DRAM访问时间）几乎与容量无关，基本保持不变。

- SRAM
  - 针对在给定容量下的低延迟进行优化。
  - 可以在容量和延迟之间进行可调节的权衡：

- 内存层次结构的核心作用是桥接CPU速度与DRAM速度之间的差距。
  - 如果 T_pclk≈ T_DRAM
    → CPU主频周期与DRAM延迟差不多，不需要分层。
  - 如果 T_pclk<< T_DRAM
    → CPU比DRAM快很多，则需要一层或多层更大但更慢的SRAM缓存，用来降低T_1（一次内存访问延迟）。

平均访问时间

- L_1 级缓存的原始访问时间是 t_1。
- 平均访问时间 T_1 通常 比 t_1 更长，因为可能会发生缓存未命中：
  - 命中率 h_1：如果在 L_1 缓存中找到数据 → 访问时间 t_1。
  - 未命中率 m_1：如果在 L_1 缓存中没找到 → 访问时间 $t_1 + T_2$（去下一级找）。
  - $T_1 = h_1 \cdot t_1 + m_1 \cdot (t_1 + T_2)$且 $h_1 + m_1 = 1$。
- 一般情况：
  - $T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1}) = t_i + m_i \cdot T_{i+1}$
    - $m_i \cdot T_{i+1}$ 被称为 “未命中代价”（miss penalty）。
    - h_i 和 m_i 是条件概率，前提是“在 L_{i-1} 未命中”。最底层内存（比如 DRAM）必然命中。

举个例子

- 假设我们有三级缓存 + 主存：
- L_1 → L_2 → L_3 → DRAM
- 对于所有 CPU 请求：
  - L_1 的命中率是 h_1，未命中率是 m_1。
  - 如果在 L_1 未命中，才会去 L_2。这时 L_2 的命中率 h_2 是在 L_1 未命中的前提下计算的，而不是全局。
  - 如果 L_2 也未命中（m_2），就去 L_3，依此类推。
  - 最后，如果 L_3 也未命中，一定会在 DRAM 找到，因此 DRAM 的命中率就是 1.0。

优化平均访问时间

- $T_i = t_i + m_i \cdot T_{i+1}$
- 目标：在可接受的成本内，使 T_1 尽可能低。
- 保持 t_i 低：
  - t_i 越小 → 访问更快，但容量更小、成本更高。
- 保持 m_i 低：
  - 增加缓存容量 C_i → 降低未命中率 m_i，但会导致 t_i 变大。
  - 通过智能管理降低 m_i，例如：
    - 替换策略：预测哪些数据不需要，提前替换掉。
    - 预取策略：预测将要用到的数据，提前加载。
- 保持 T_{i+1} 低：
  - 通过更快的下一级内存降低 t_{i+1}，但会增加成本、减少容量。
  - 更好的方法：增加中间层级，在速度和容量之间做折中。

Intel P4 示例

- 工艺 & 频率：90nm，3.6 GHz
- 16KB L1 数据缓存
  - t_1 = 4 个周期（整数），9 个周期（浮点）
- 1024KB L2 数据缓存
  - t_2 = 18 个周期（整数/浮点相同）
- 主存（DRAM）
  - t_3 ≈ 50ns ≈ 180 个周期
- 注意事项
  - L1 非常小
  - 最佳情况的延迟也不是 1 个周期（深流水线带来的延迟）
  - 最坏情况下，访问延迟可能超过 300+ 个周期，取决于具体情况

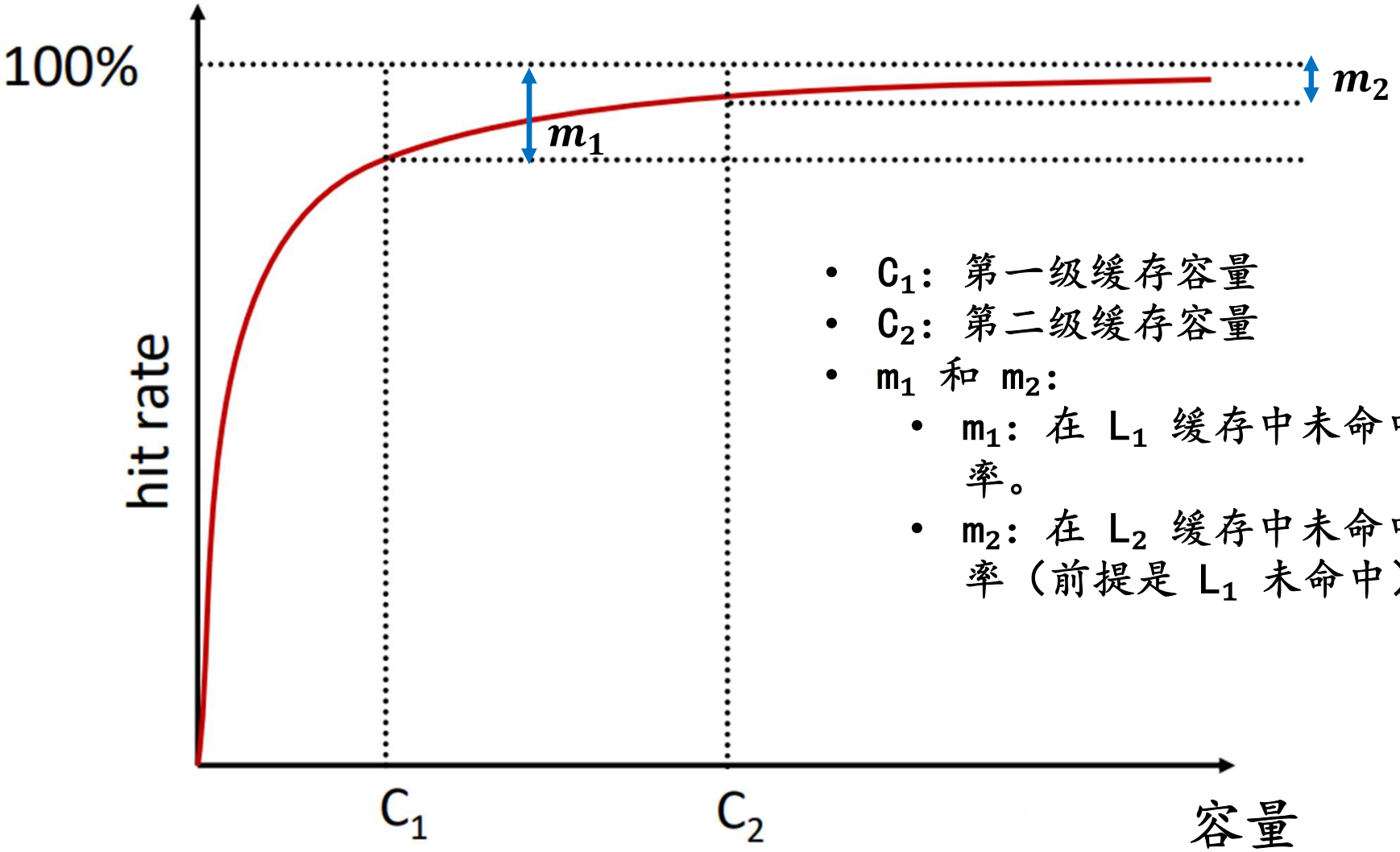
未命中率 VS 命中率

- 命中 or未命中：差距巨大
- 你会相信99%命中率要比97%好两倍?
  - 假设:
    - 缓存命中时间为1个周期
    - 缓存未命中惩罚要100个周期

- 平均访问时间
  - 97% 命中率：0.97*1 周期 + 0.03 x 100 周期 = 4 周期
  - 99% 命中率：0.99*1 周期 + 0.01 x 100 周期 = 2 周期

- 这就是为什么用“未命中率”而不是“命中率”

工作集/缓存容量/局部性/未命中率



- C_1：第一级缓存容量
- C_2：第二级缓存容量
- m_1 和 m_2：
  - m_1：在 L_1 缓存中未命中的概率。
  - m_2：在 L_2 缓存中未命中的概率（前提是 L_1 未命中）。

为什么DRAM 很慢？

- DRAM制造在超大规模集成电路中处于前沿，但按照摩尔定律发展的重点是容量和成本，而不是速度。
- 1980 ~ 2004 年：
  - 容量：从64Kbit → 1024Mbit，呈指数级增长（年均约 55%）。
  - 速度：访问延迟从 250ns → 50ns，只实现了线性提升。
- 这是刻意的工程设计选择：
  - 为了保持系统平衡，内存容量需要和处理器性能以接近线性的速度同步提升。
  - DRAM 与 CPU 速度的差距，通过多级 SRAM 缓存层次结构（L1、L2、L3…）来弥补。

总结

- CPU、主存、大容量存储设备之间的速度差距持续扩大
- 编写良好的程序表现出良好的局部性
- 利用局部性特点，基于高速缓存的存储器层次结构有利于缩小速度差距