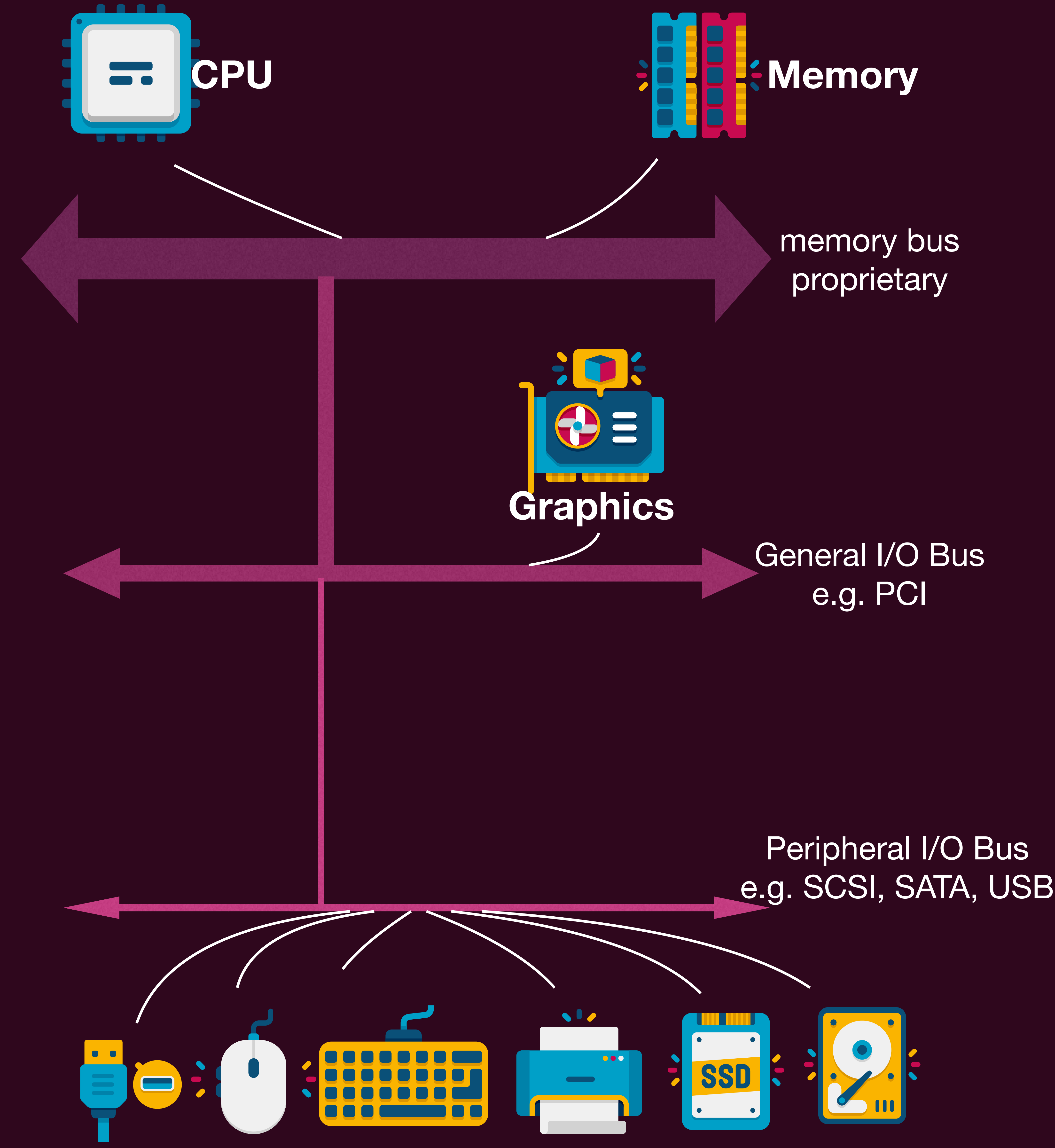
# 设备管理
# Device Management

钮鑫涛
南京大学
2026春

本门课由蒋炎岩和钮鑫涛共同构建

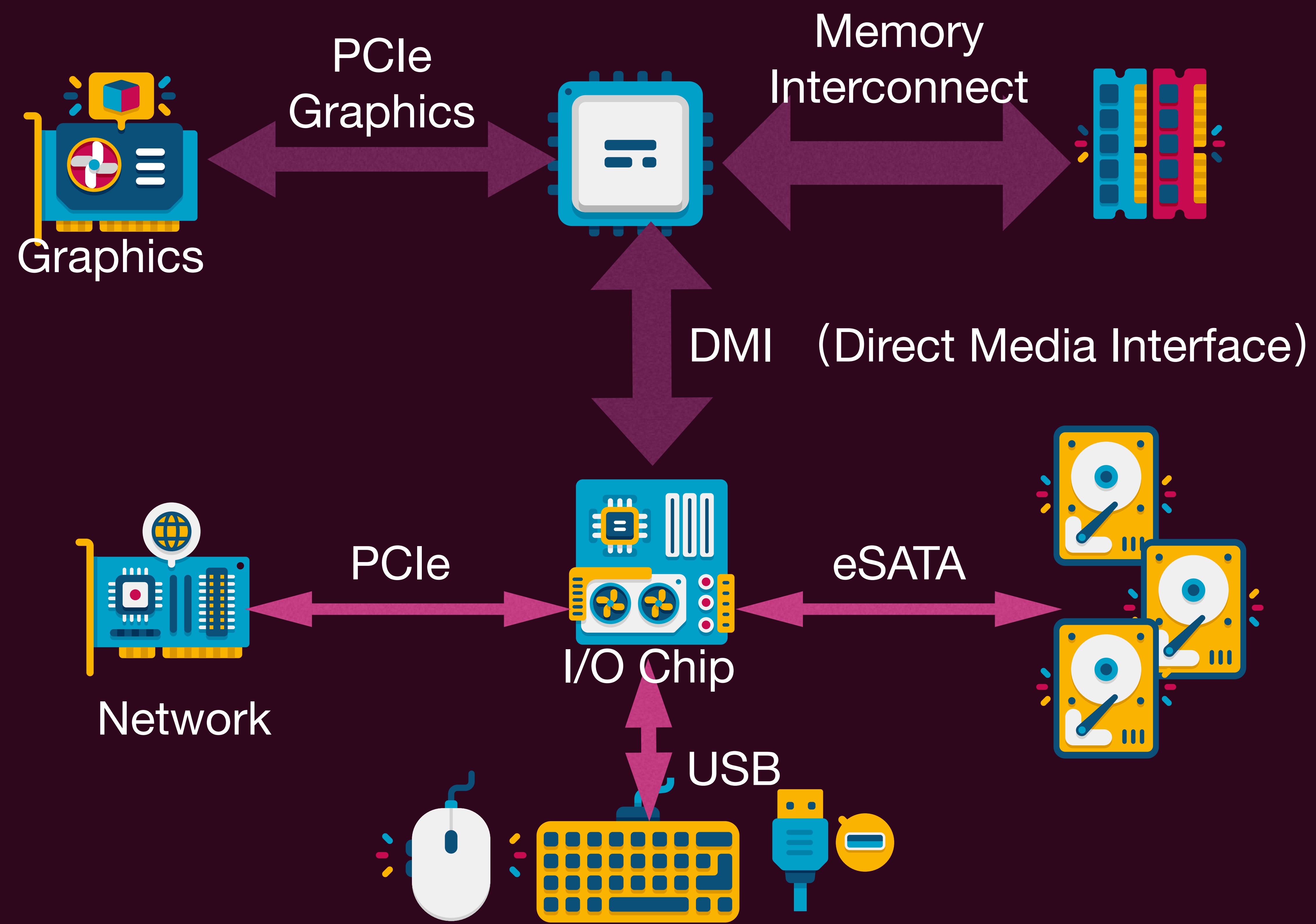
# 典型的系统架构

- 通用I/O总线和多个外围总线用于连接设备
  - 层次结构
    - 物理和成本：总线速度越快，长度就必须越短
    - 需要高性能的设备更靠近CPU

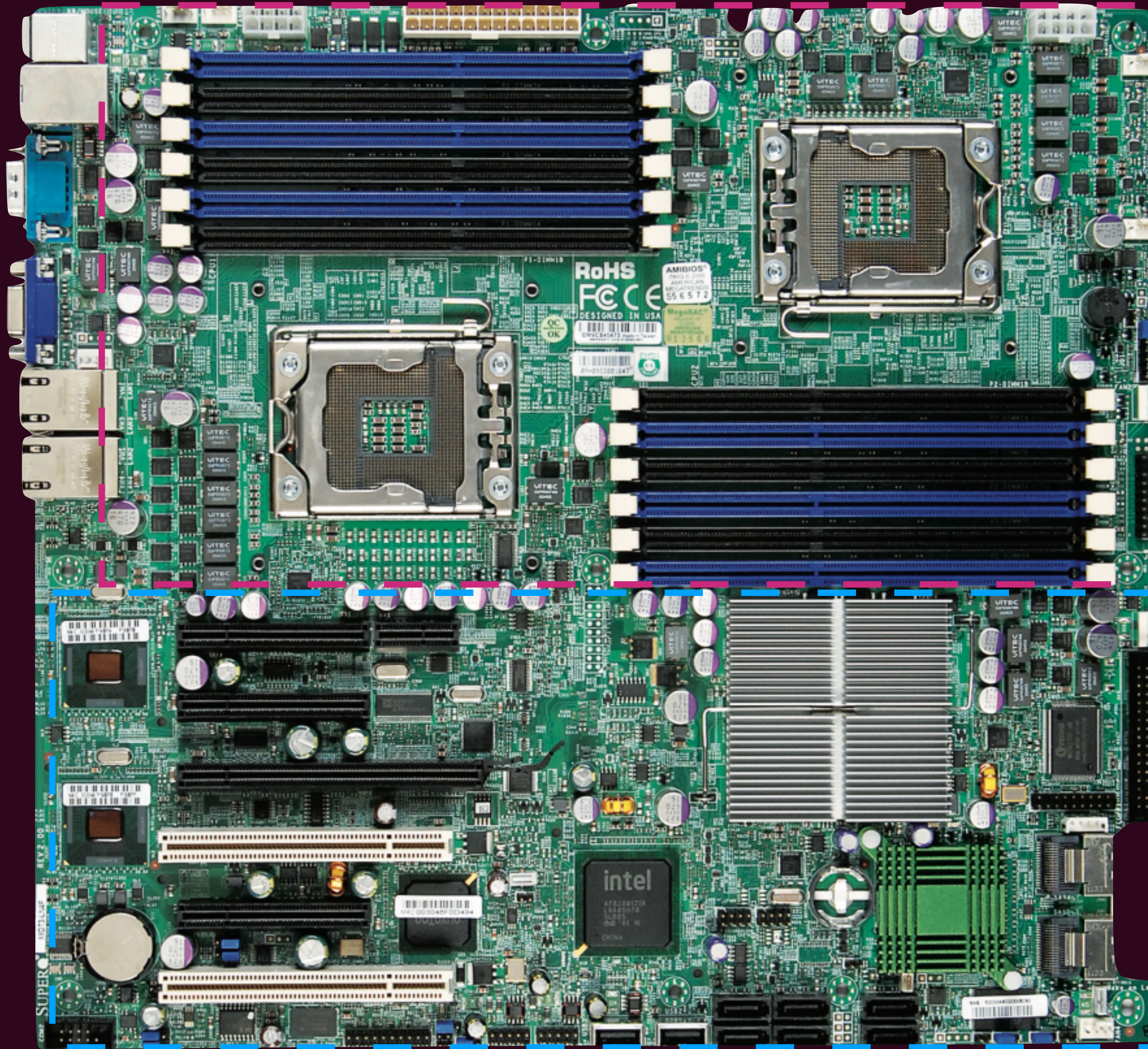


# 典型的系统架构

- 通用I/O总线和多个外围总线用于连接设备
  - 现代系统通常使用专用芯片组来提高性能



# 一个主板（Supermicro X8DT3，2011年）




- 2个CPU插槽和12个RAM插槽位于右上角

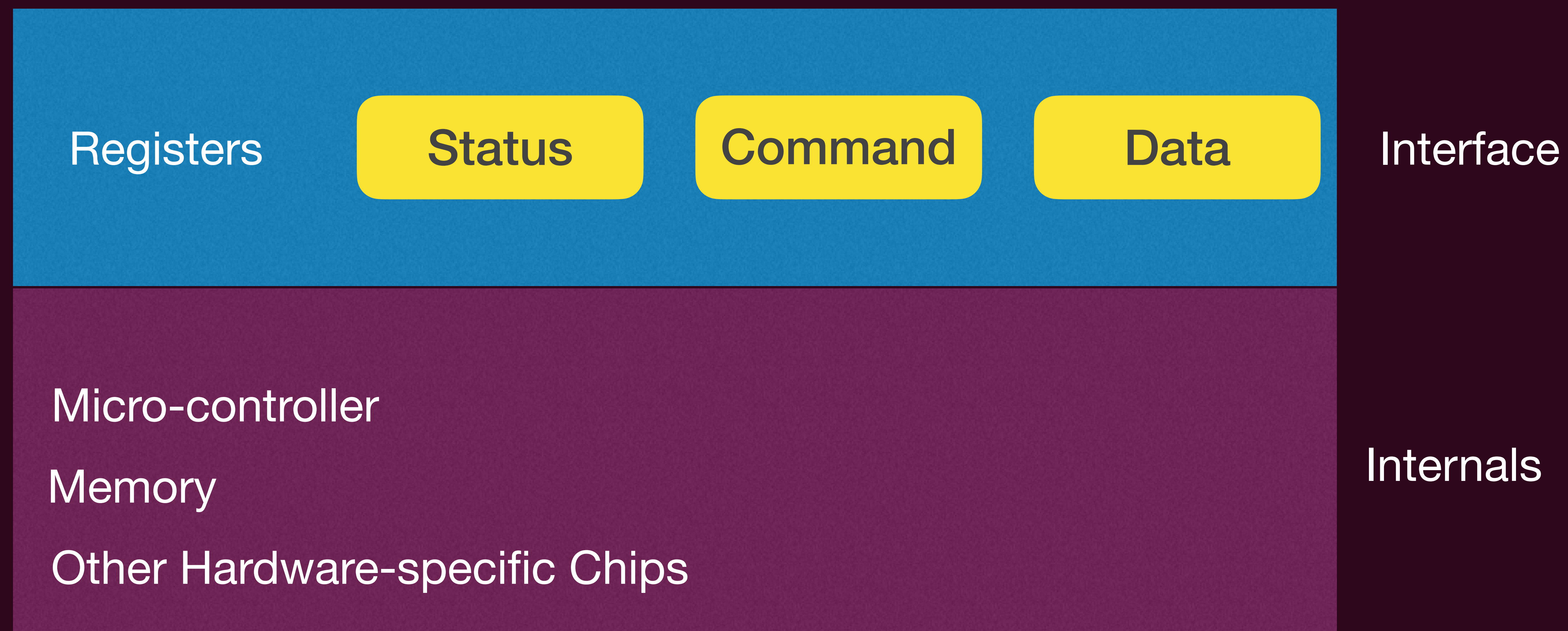
- 主板的其余部分专用于I/O控制器和设备。
  - 这部分因机器而异，操作系统必须以某种方式处理不同的硬件

# I/O设备

- 设备大体可以分类为:
  - 块设备（Block devices）
    - 以固定大小的块存储信息
    - 传输单位为整个块
  - 字符设备 (Character devices)
    - 传递或接受字符流
    - 不可寻址，没有任何寻址操作

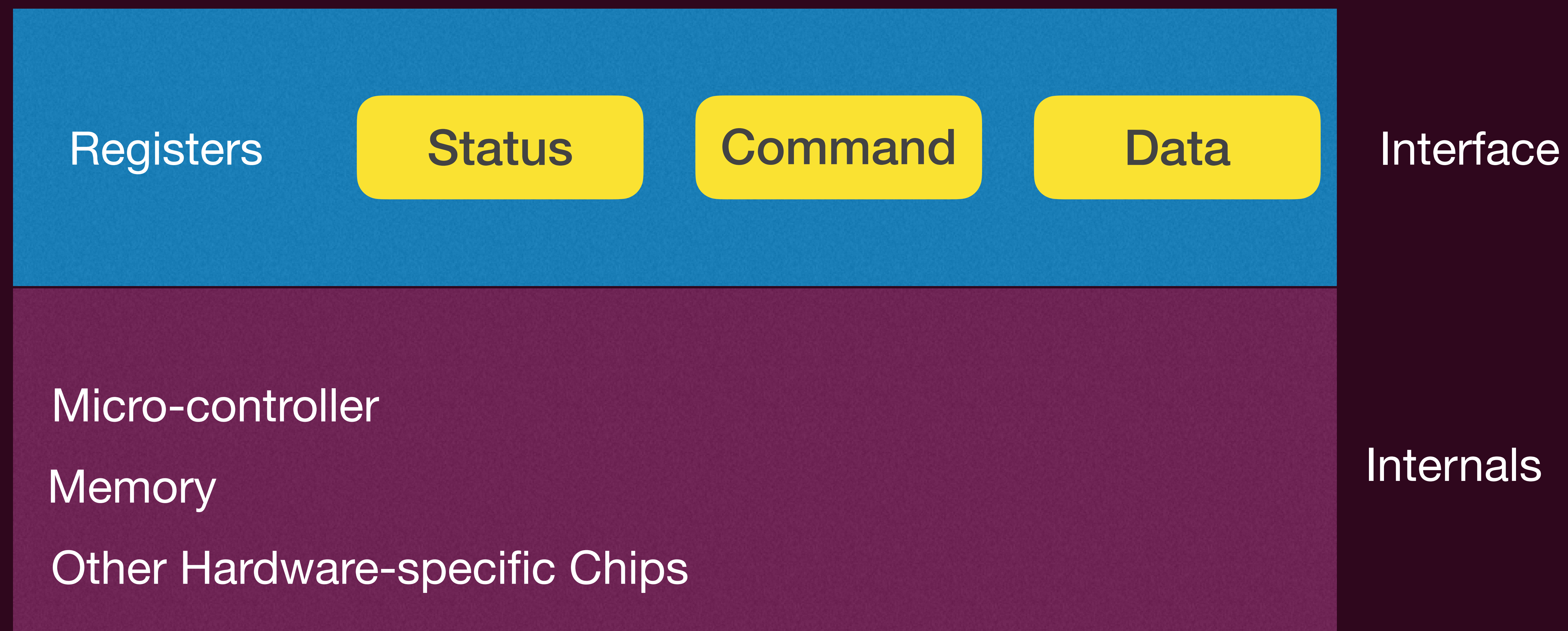
# 一个典型设备

- 一个设备有两个重要组成部分
  - 它向系统其余部分呈现的硬件接口（允许操作系统控制其运行）
  - 它的内部结构（具体实现）



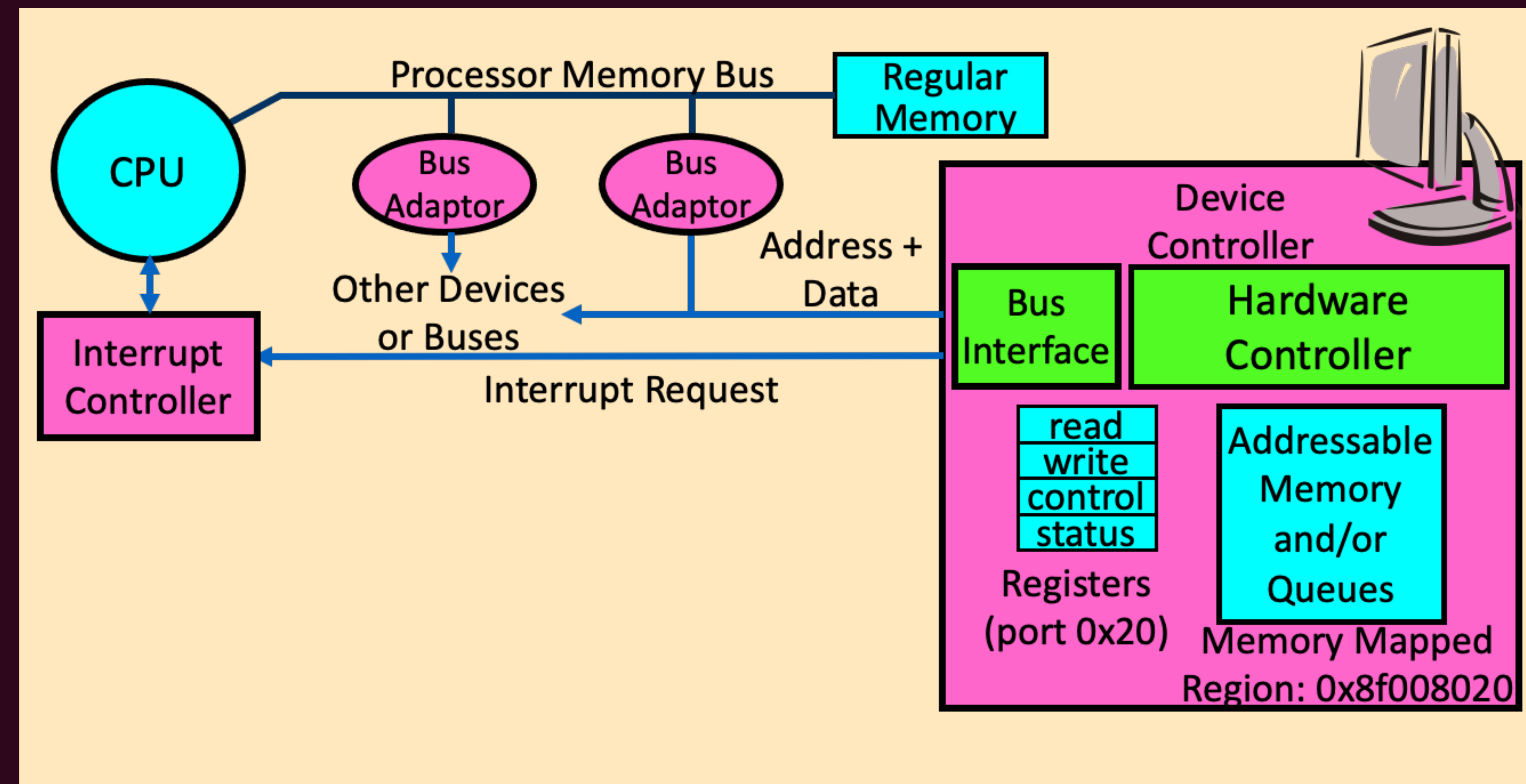
# 一个典型设备

- 设备接口由几个寄存器组成
  - 状态寄存器：读取以查看设备的当前状态
  - 命令寄存器：指示设备执行特定任务 数据寄存器：
  - 将数据传递给设备或从设备获取数据



# CPU和设备的通信

- CPU与控制器交互
  - 除了一组可读写的寄存器外，可能包含用于请求队列等的内存
- 处理器以两种方式访问寄存器：
  - 端口映射(Port-Mapped) I/O
  - 内存映射(Memory-Mapped) I/O

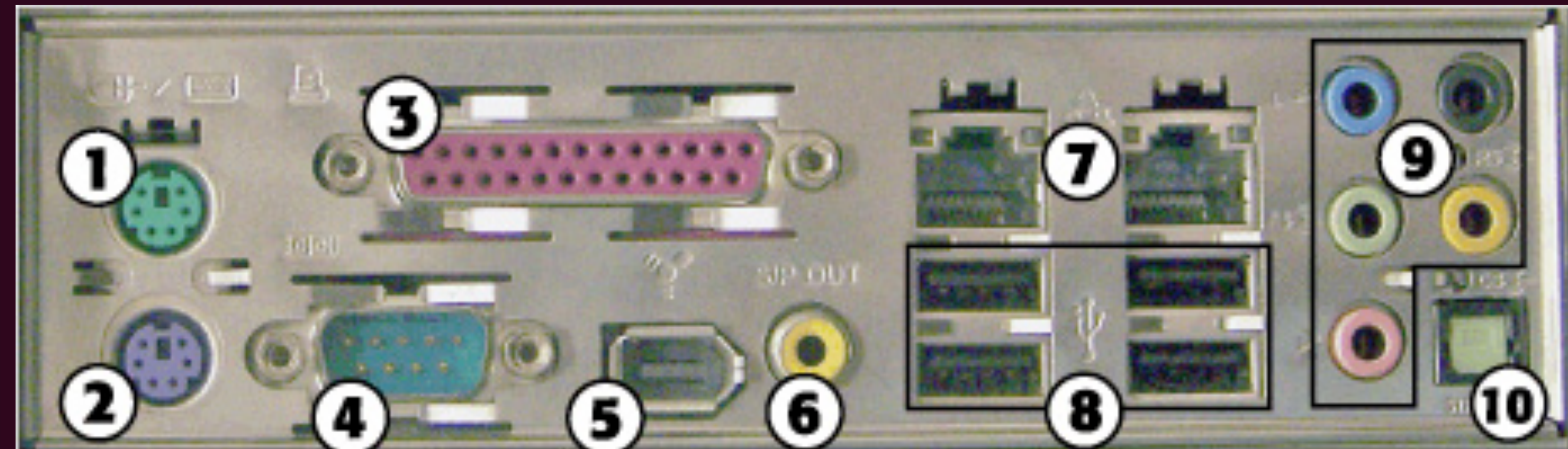


# 端口映射

- 端口映射：提供额外的I/O指令
  - 每个控制寄存器被分配一个I/O端口号
  - 使用特殊的I/O指令（在x86上为in和out）
  - 这些指令通常是特权指令

# 典型的端口列表

| Port range | Summary |
|---|---|
| 0x0000-0x001F | The first legacy DMA controller, often used for transfers to floppies. |
| 0x0020-0x0021 | The first Programmable Interrupt Controller |
| 0x0022-0x0023 | Access to the Model-Specific Registers of Cyrix processors. |
| 0x0040-0x0047 | The PIT (Programmable Interval Timer) |
| 0x0060-0x0064 | The "8042" PS/2 Controller or its predecessors, dealing with keyboards and mice. |
| 0x0070-0x0071 | The CMOS and RTC registers |
| 0x0080-0x008F | The DMA (Page registers) |
| 0x0092 | The location of the fast A20 gate register |
| 0x00A0-0x00A1 | The second PIC |
| 0x00C0-0x00DF | The second DMA controller, often used for soundblasters |
| 0x00E9 | Home of the Port E9 Hack. Used on some emulators to directly send text to the hosts' console. |
| 0x0170-0x0177 | The secondary ATA harddisk controller. |
| 0x01F0-0x01F7 | The primary ATA harddisk controller. |
| 0x0278-0x027A | Parallel port |
| 0x02F8-0x02FF | Second serial port |
| 0x03B0-0x03DF | The range used for the IBM VGA, its direct predecessors, as well as any modern video card in legacy mode. |
| 0x03F0-0x03F7 | Floppy disk controller |
| 0x03F8-0x03FF | First serial port |



# 内存映射

- 内存映射I/O：
  - 将所有控制寄存器映射到内存空间中
  - 每个控制寄存器被分配一个唯一的内存地址
  - 为了访问特定的寄存器，操作系统发出一个load指令（读取）或store指令（写入）该地址
  - 然后硬件将load/store指令指向到设备而不是主存储器

# 内存映射

- 内存映射时小心缓存！
  - 比如下面的例子：
    - 第一次test这个address的值会被缓存到cache中
    - 之后的轮训test是直接从cache中取值，无法反映设备的真正状态
    - 因此一般要禁掉相应内存地址的缓存

```
Loop:  test memory_mapped_io_arddress //check the status of device
       jz ready  // if it is 0, go to ready
       goto loop  // otherwise, continue testing
ready:
```

# 获知设备通信状态

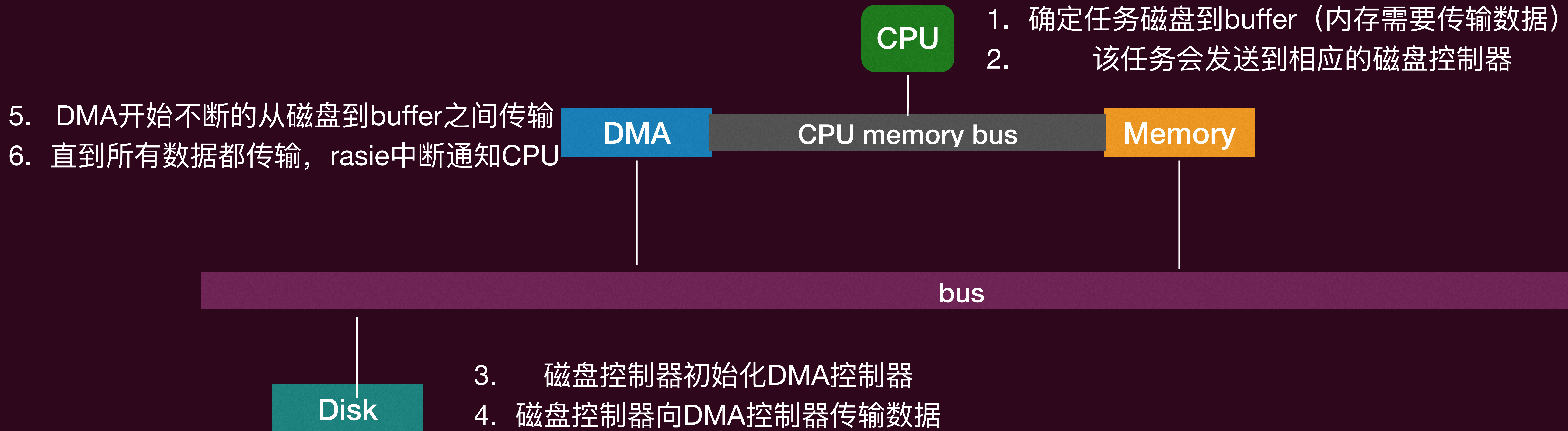
- 在发出与设备通信的指令后，操作系统需要知道以下情况：
  - I/O 设备已完成一个操作
  - I/O 操作遇到了错误
- 有两种获知状态的方法：
  - 轮询(polling)：操作系统定期检查设备特定的状态寄存器
    - 开销低（没有切换）， 但如果是低速的设备上会浪费CPU周期
  - I/O 中断：设备在需要服务时生成中断
    - 不会浪费CPU周期，但是开销高（伴随切换）
  - 一般来说两种混合

# 谁来控制数据传输命令?

- 由CPU来控制:
  - 即Programmed I/O
    - 直接通过处理器的in/out或load/store传输每个字节
    - 优点：硬件简单，易于编程
    - 消耗与数据大小成比例的处理器周期（因为每个数据的in/out都要一个指令）
    - 此外，CPU会一直接收到interrupt，速度变慢

# 谁来控制数据传输命令?

- 由直接内存访问（Direct Memory Access, DMA）控制
  - 给控制器访问内存和总线的权限
  - 要求它直接在内存和控制器之间传输数据块



# 关于“通用”和“专用”

- 之前，我们关于CPU的讨论都是通用CPU
  - 然而计算机系统内部不只是有通用的CPU
    - “专门”负责memcpy()的CPU：DMA
    - “专门”负责解析和显示图形的CPU：GPU
    - 很多设备都有自己的逻辑处理芯片（比如磁盘）
    - I/O管理可以看成是中央芯片（CPU）和外部芯片的交流！

# 关于“通用”和“专用”

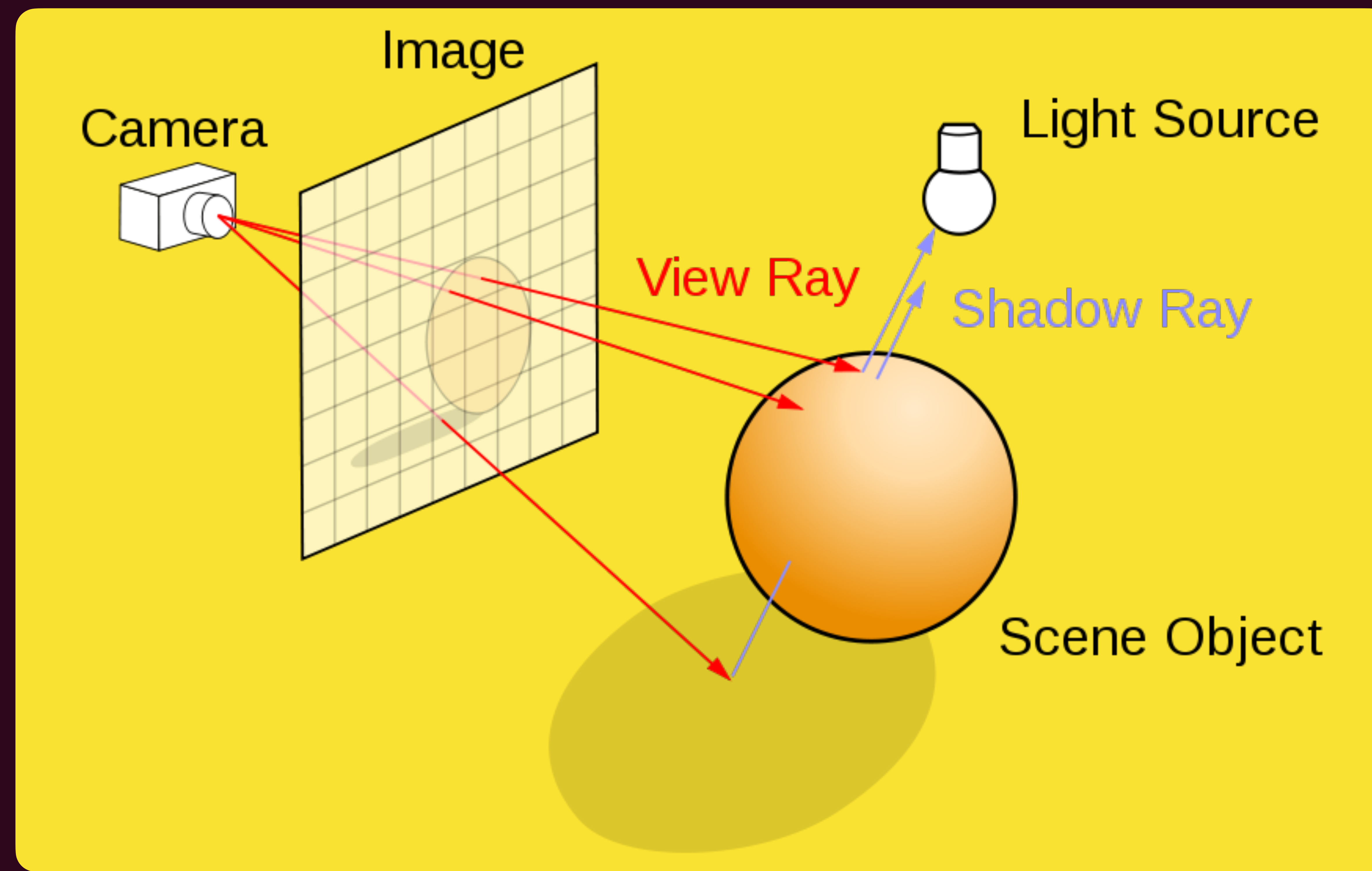
- 为什么不是人手一个通用CPU?
  - 因为代价！
    - CPU因为通用性增加了很多设计以及与之而来的功耗
  - 特定领域的任务不需要这么多通用性
    - 可以更加优化所需任务的指令（比如显卡中的并行计算部分）
- 其实语言也是如此（Domain Specific Language）
- 算法也是如此！

# GPU加速

- 显卡（Graphic Processing Unit）
  - 设计用于可并行化的问题
  - 最初专为图形处理创建
  - 后来变得更能够进行通用计算
  - 计算速度非常快且强大
  - 但需要消耗大量电力

# Raytracing 问题

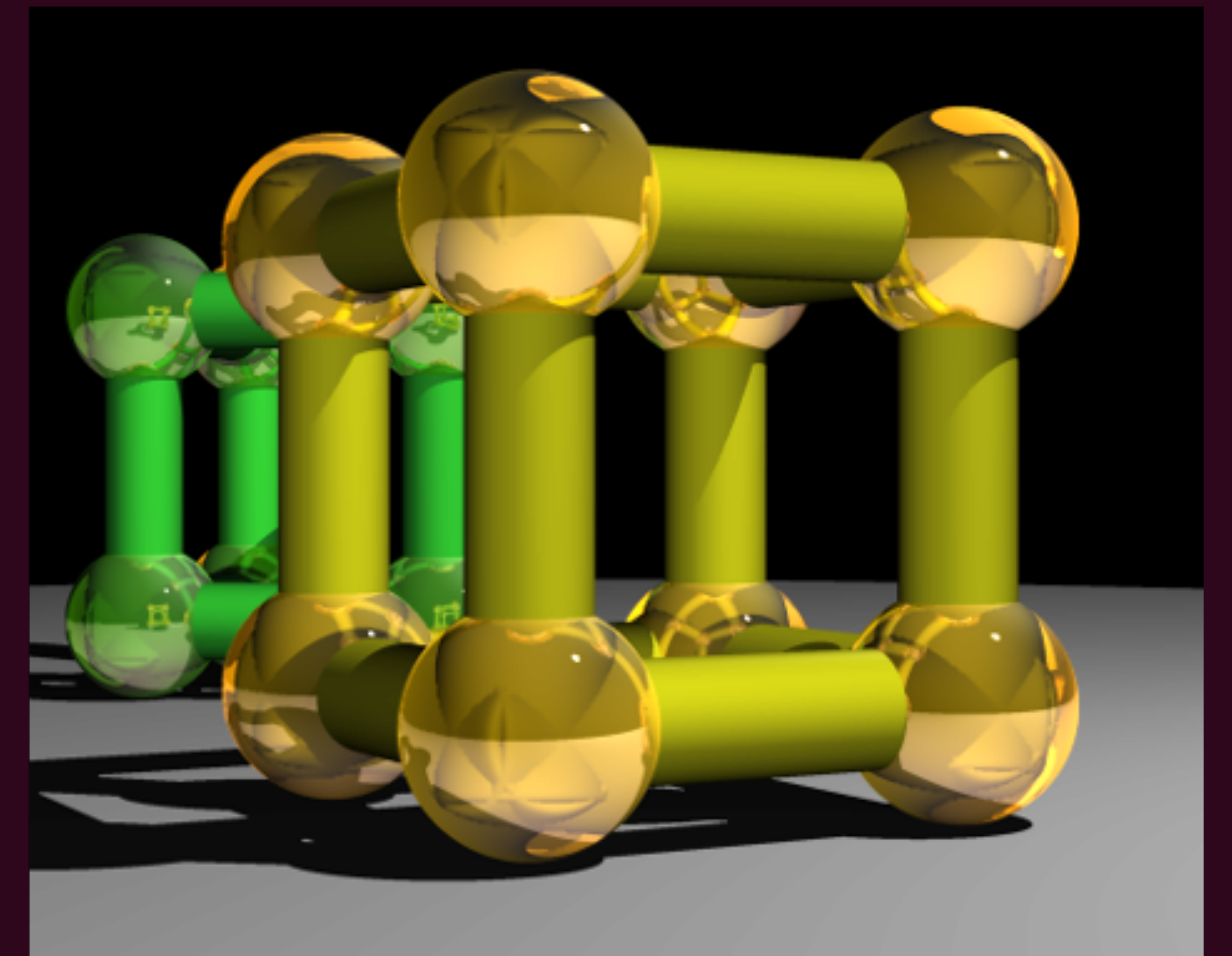
- 对于一个光线追踪问题（用于3维图形），从摄像机发出一条可视光线（view ray），找到场景中每个像素的可视点，然后从这个可视点向光源发射一条阴影光线（shadow ray），求出该可视点的颜色。
  - 根据材质不同 （如玻璃、水面等可能会产生反射、折射）会递归追踪



# Raytracing 问题

- 对于一个光线追踪问题（用于3维图形），下面算法的每个像素都可以并行的计算！

```
for all pixels (i,j) in image:
    From camera eye point,
        calculate ray point and direction in 3d space
    if ray intersects object:
        calculate lighting at closest object point
        store color of (i,j)
Assemble into image file
```



Superquadric Cylinders, exponent 0.1, yellow glass balls, Barr, 1981

# 另外一个问题

- 将两个数组相加，A和B生成数组C:  A[ ] + B[ ] -> C[ ]

- 在CPU上计算

```
float *C = malloc(N * sizeof(float));
for (int i = 0; i < N; i++)
    C[i] = A[i] + B[i];
return C;
```

- 当然，我们可以利用多线程编程来进行加速，然而真实的物理核心是有限的！更多的线程只是CPU的虚像！

# GPU的解决方法

- On the GPU:
  - （在GPU上为数组A、B、C分配内存） 创建“核”——每个线程将执行一个（或几个）加法操作
  - 指定以下核操作：
    - 对于分配给该线程的所有i（索引）：
    - C[i] <- A[i] + B[i]
  - 同时启动大约20000个线程！
  - 等待线程同步

A100最大可以同时支持221,184个线程！

# GPU计算

- 英伟达的GPU的计算平台是CUDA (统一计算设备架构)
- 首先定义核函数（每个线程都跑这个同样的函数）

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```

# GPU计算

- 调用核函数

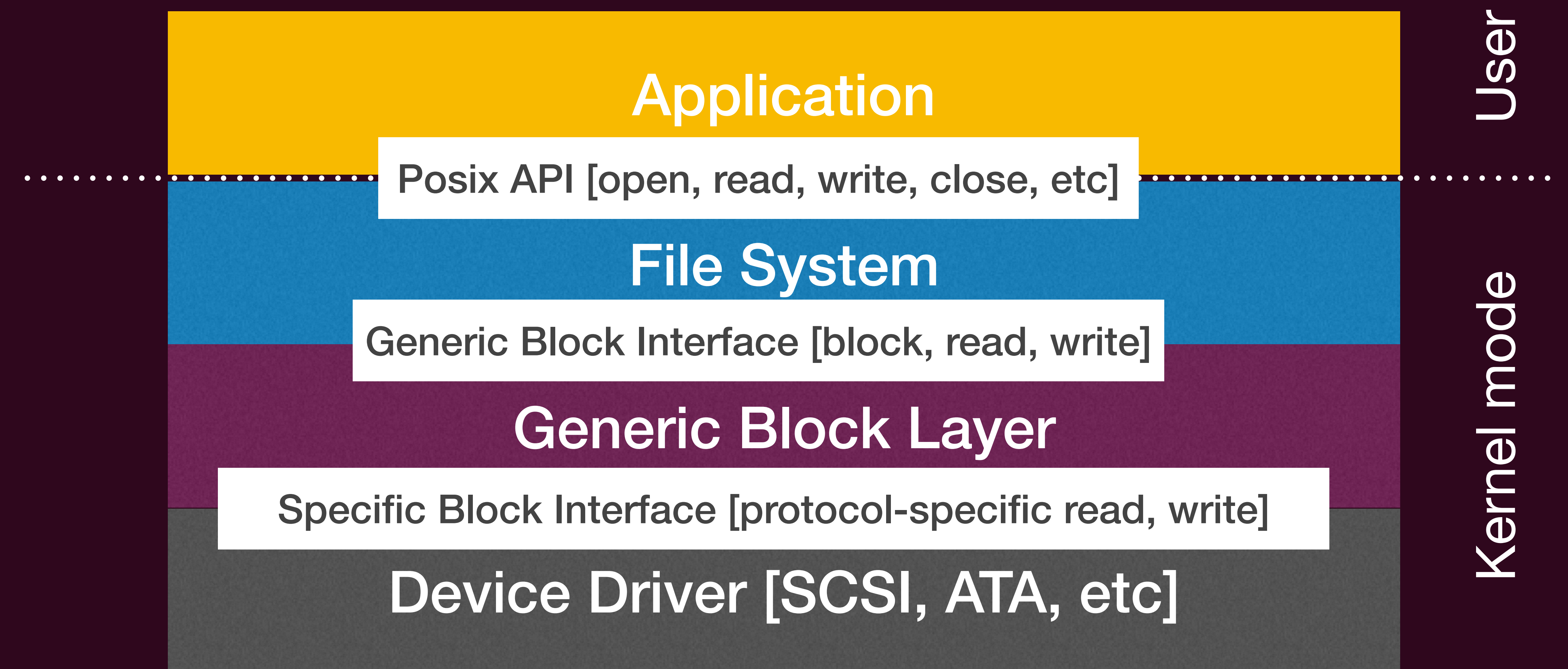
```
void cudaAddVectors(const float* a, const float* b, float* c, size){
    //For now, suppose a and b were created before calling this function

    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be
    // arrays on the GPU.

    float * dev_a;
    float * dev_b;

    float * dev_c;

    // Allocate memory on the GPU for our inputs:
    cudaMalloc((void **) &dev_a, size*sizeof(float));
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);


    // Allocate memory on the GPu for our outputs:
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```

```
    //At lowest, should be 32
    //Limit of 512 (Tesla), 1024 (newer)
    const unsigned int threadsPerBlock = 512;

    //How many blocks we'll end up needing
    const unsigned int blocks = ceil(size/float(threadsPerBlock));

    //Call the kernel!
    cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
        (dev_a, dev_b, dev_c);

    //Copy output from device to host (assume here that host memory
    //for the output has been calculated)

    cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

    //Free GPU memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

# 硬件抽象

# 硬件抽象

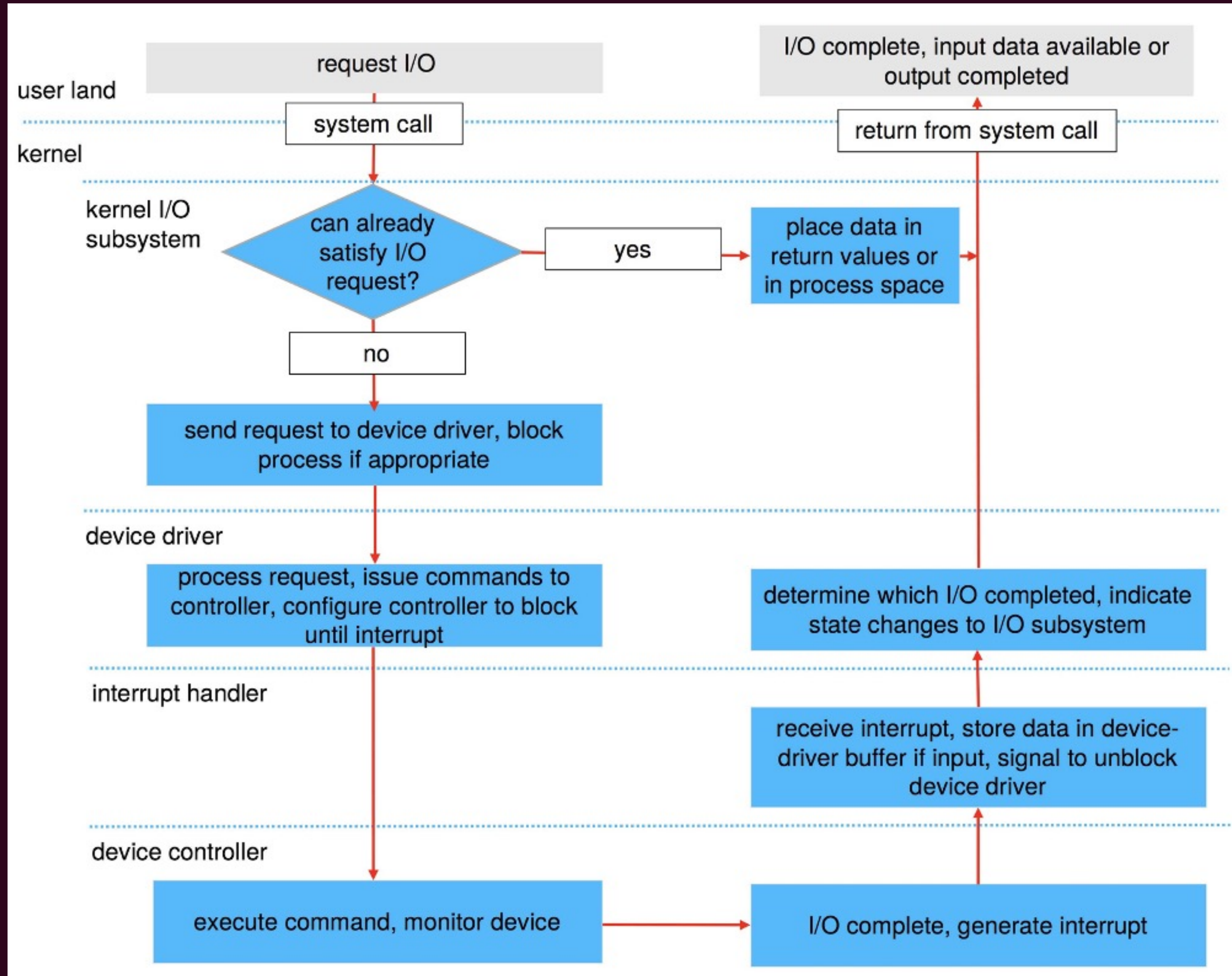
- 操作系统创建了一个分层的存储视图
  - 即使在内核内部抽象也是不断使用的技术！
- 分层方法允许更低层部分容易更改
  - 例如，文件系统的实现独立于磁盘类型
- 管理 I/O 设备的代码在内核设备**驱动程序(device drivers)**中
  - 70%的 Linux 代码是设备驱动程序，因为有太多不同的设备，每个设备都不同！（1530万行源代码！）
  - 设备驱动程序是内核错误的最常见来源
    - 这是由于给定的驱动程序可能只被少数系统使用，所以它不会被高度使用或仔细检查。



# 设备抽象

- 设备驱动程序通常分为两部分：
  - 上半部分：在系统调用的调用路径中访问
    - 实现一组标准的、跨设备的调用，如 open()、close()、read()、write()、ioctl()
    - 这是内核与设备驱动程序的接口
    - 上半部分将启动设备的 I/O 操作，可能会让线程休眠直到完成 (同步和异步I/O!)
  - 下半部分：作为中断例程运行
    - 获取输入或传输下一块输出
    - 如果 I/O 现在完成，可能会唤醒休眠的线程

# 一次I/O请求的生命周期



# 一个设备驱动样例(xv6 简化版)

```
Control Register:
    Address 0x3F6 = 0x80 (0000 1RE0): R=reset, E=0 means "enable interrupt”

Command Block Registers:
    Address 0x1F0 = Data Port
    Address 0x1F1 = Error
    Address 0x1F2 = Sector Count
    Address 0x1F3 = LBA low byte
    Address 0x1F4 = LBA mid byte
    Address 0x1F5 = LBA hi byte
    Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
    Address 0x1F7 = Command/status

Status Register (Address 0x1F7): 76543210
    BUSY READY FAULT SEEK DRQ CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when Status ERROR==1) 76543210
    BBK UNC MC IDNF MCR ABRT T0NF AMNF
    BBK = Bad Block
    UNC = Uncorrectable data error MC = Media Changed
    IDNF = ID mark Not Found
    MCR = Media Change Requested ABRT = Command aborted
    T0NF = Track 0 Not Found
    AMNF = Address Mark Not Found
```

- 9个用于与设备交互的
- 一个字节寄存器
- 使用x86 in/out指令
- 有时寄存器中的单个位被使用
- 0x3F6可以启用中断
- 0x1F0是一个数据缓冲区
- 0x1F7是你写入命令的地方
- 在访问设备之前获取锁!

# 一个设备驱动样例(xv6 简化版)

```c
static int ide_wait_ready() {
    while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
    ; // loop until drive isn't busy
}
static void ide_start_request(struct buf *b) {
    ide_wait_ready();
    outb(0x3f6, 0);  // generate interrupt
    outb(0x1f2, 1);  // how many sectors?
    outb(0x1f3, b->sector & 0xff);   // LBA goes here ...
    outb(0x1f4, (b->sector >> 8) & 0xff); // ... and here
    outb(0x1f5, (b->sector >> 16) & 0xff); // ... and here!
    outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
    if(b->flags & B_DIRTY){
        outb(0x1f7, IDE_CMD_WRITE); // this is a WRITE
        outsl(0x1f0, b->data, 512/4); // transfer data too!
    } else {
        outb(0x1f7, IDE_CMD_READ);  // this is a READ (no data)
    }
}
```

- 0x1f7 = command status
- 0x3f6 = control register
- 0x1f3-0x1f6 = disk address (Logical Block Addressing)
- 0x1f0 = data

# 一个设备驱动样例(xv6 简化版)

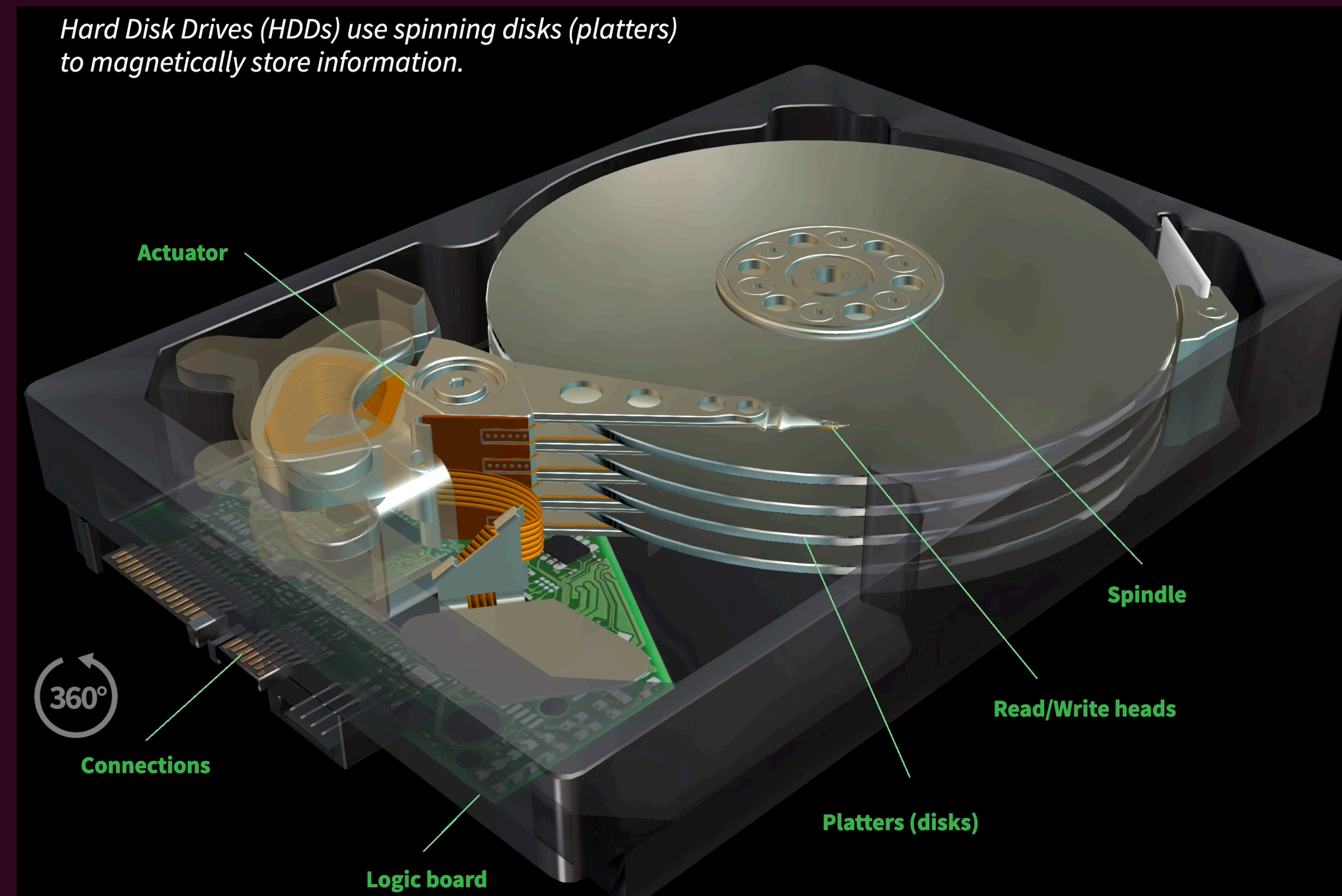
```
void ide_rw(struct buf *b) {
    acquire(&ide_lock);
    for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
    ;                         // walk queue
    *pp = b;                  // add request to end
    if (ide_queue == b)       // if q is empty
        ide_start_request(b);  // send req to disk
    while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
        sleep(b, &ide_lock); // wait for completion
    release(&ide_lock);
}

void ide_intr() {
    struct buf *b;
    acquire(&ide_lock);
    if (!(b->flags & B_DIRTY) && ide_wait_ready(1) >= 0)
        insl(0x1f0, b->data, 512/4); // if READ: get data
    b->flags |= B_VALID;
    b->flags &= ˜B_DIRTY;
    wakeup(b); // wake waiting process
    if ((ide_queue = b->qnext) != 0) // start next request
        ide_start_request(ide_queue); // (if one exists)
    release(&ide_lock);
}
```

- Queue new requests
- Interrupt handler

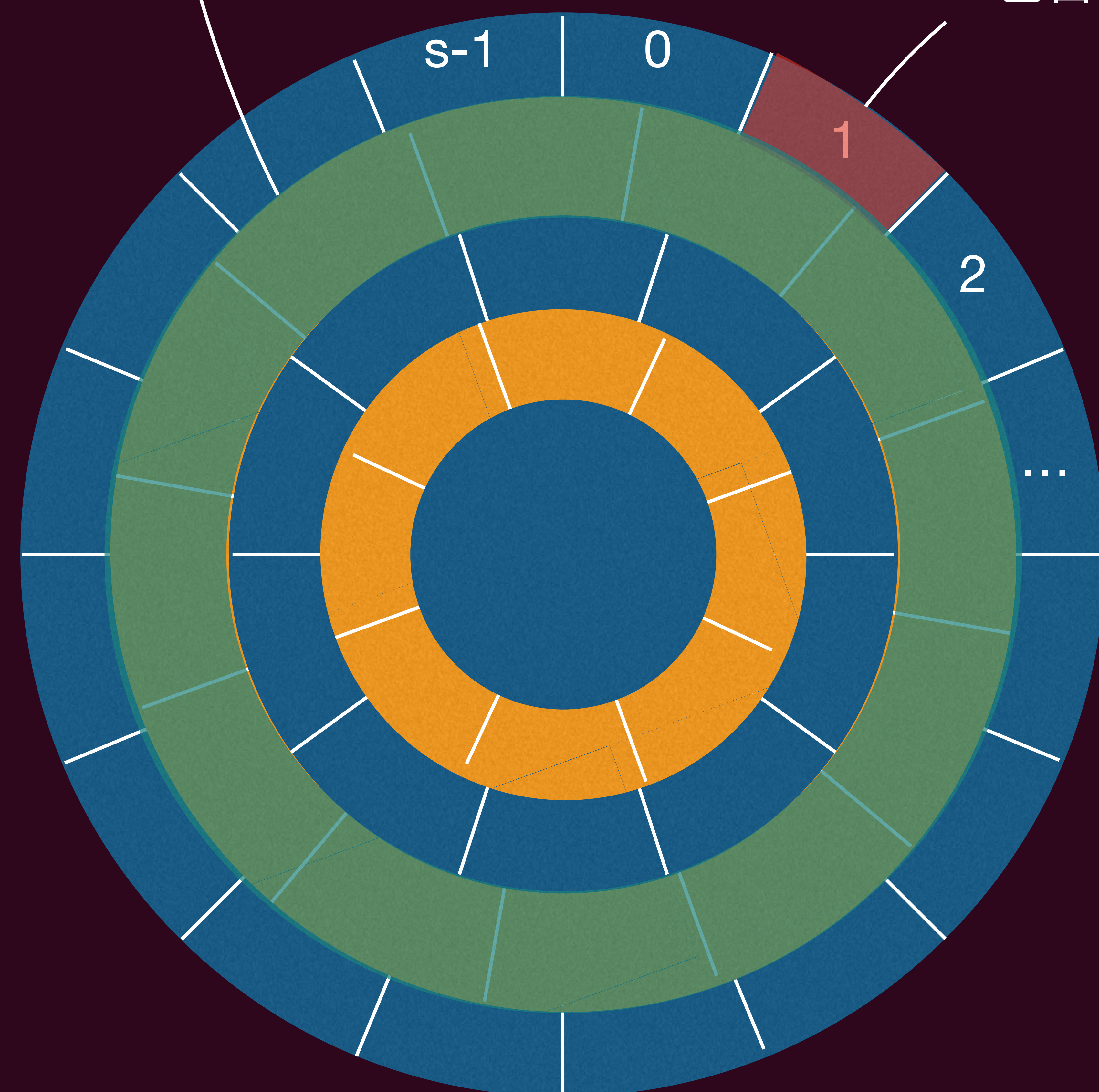
# 硬盘

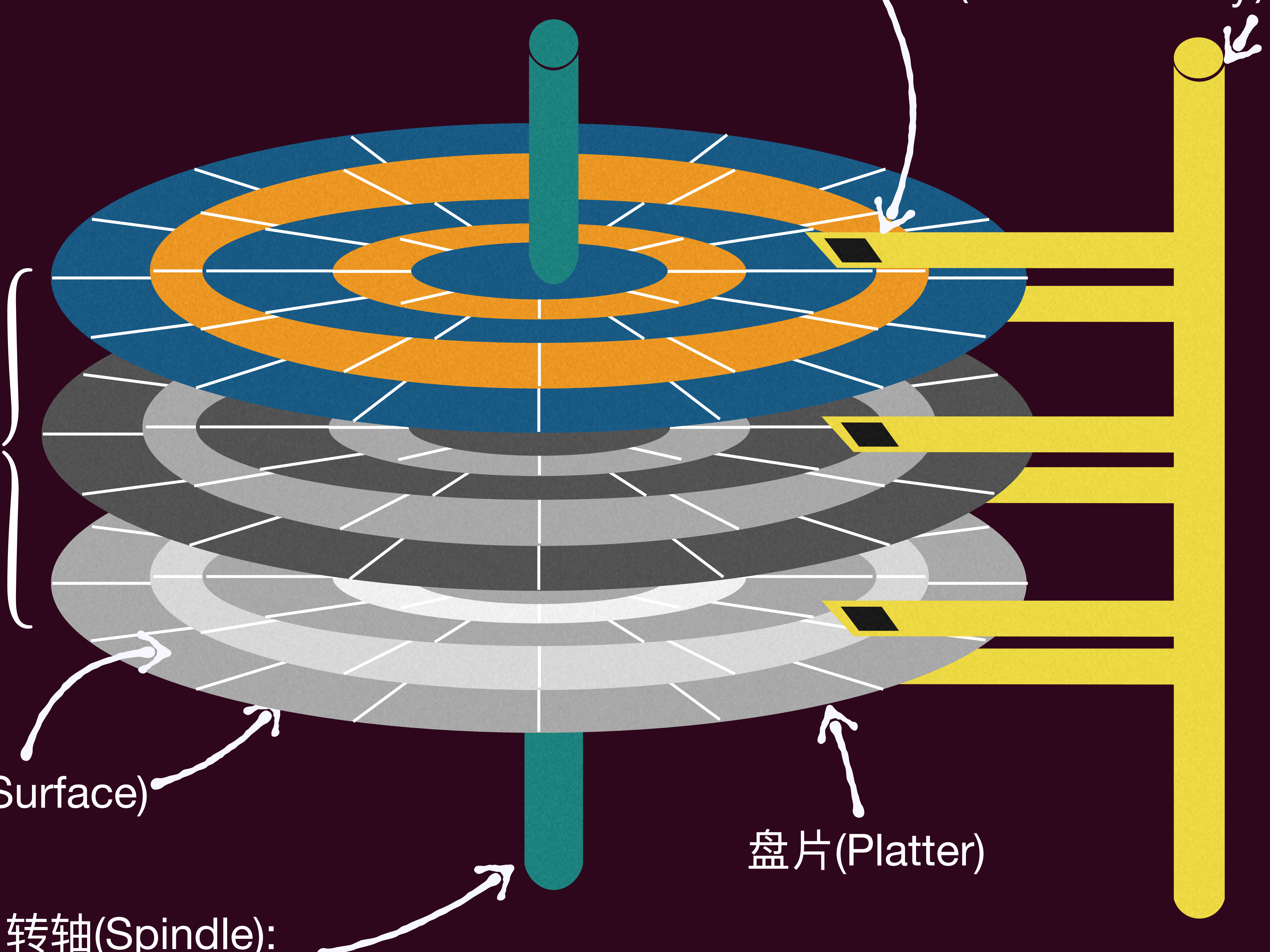
# 磁盘

- 将数据磁性地储存在与玻璃、陶瓷或铝等材料的旋转盘上粘合的薄金属膜上。



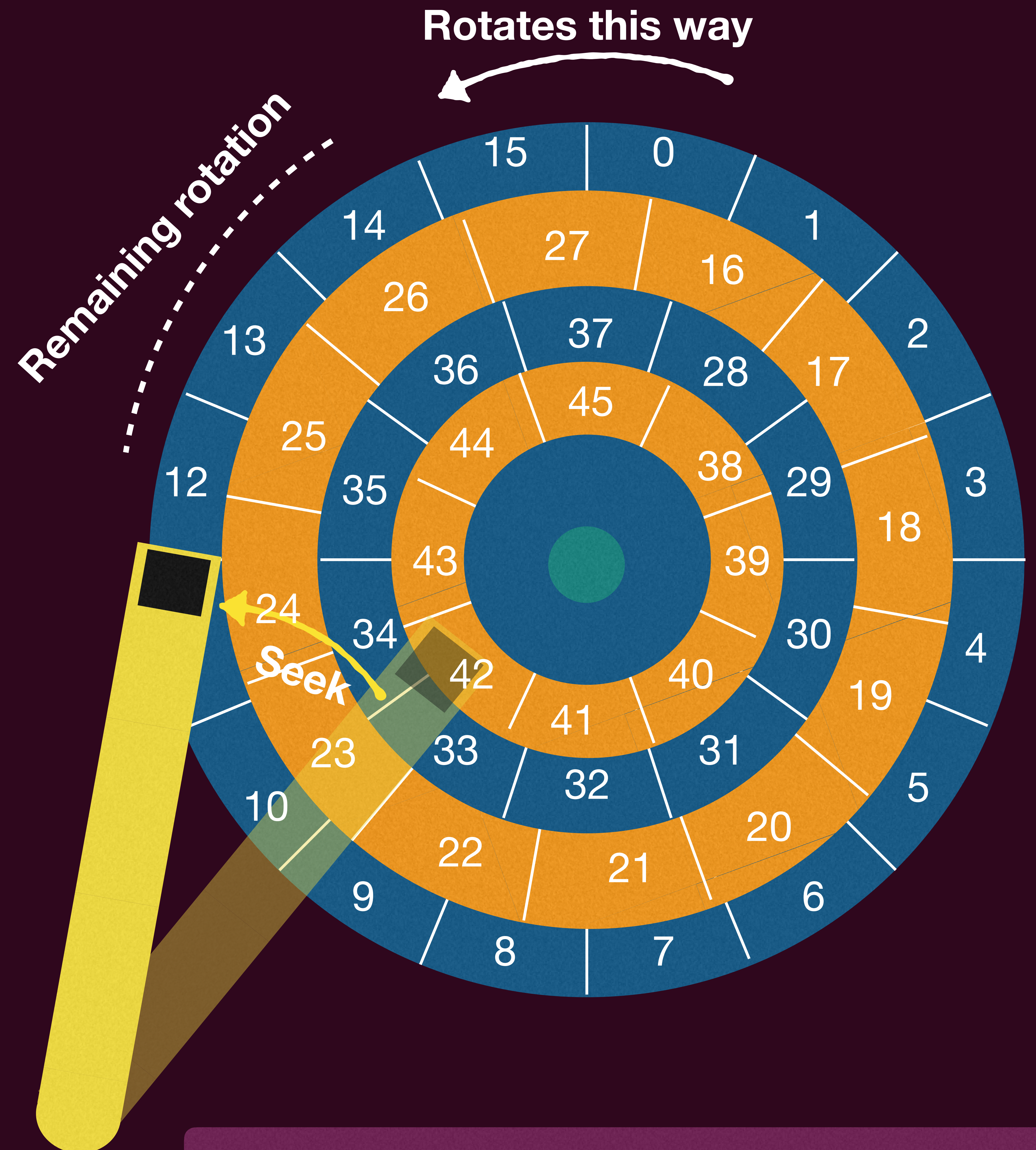
# 磁盘驱动器示意图



# 磁盘读写

- 将磁盘呈现为带有扇区地址：
  - 之前：直接地址 = （驱动器，表面，磁道，扇区）
  - 现在：逻辑块地址 （ Logical Block Address, LBA）
    - 线性地址0…N-1
- 磁头移动到适当的磁道
  - 寻道（seek）
  - 稳定（settle）
- 启用适当的磁头
- 等待扇区出现在磁头下
  - 旋转延迟（rotational latency）
- 读取/写入扇区
  - 传输时间



# 磁盘读写

- 磁盘访问时间：$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$
- 磁盘传输速率： $R_{I/O} = \frac{Size_{transfer}}{T_{I/O}}$
- 在顺序工作负载中传输100 MB数据
  - $T_{seek}$ = 4 ms (average seek)
  - $T_{rotation}$ = 2 ms (average rotation)
  - $T_{transfer}$ = 0.8 s
  - $R_{I/O}$ = 100 MB / 0.806 s = 124 MB/s

| | Cheetah 15K.5 | Barracuda |
|---|---|---|
| Capacity | 300 GB | 1 TB |
| RPM | 15,000 | 7,200 |
| Average Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |
| Platters | 4 | 4 |
| Cache | 16 MB | 16/32 MB |
| Connects via | SCSI | SATA |

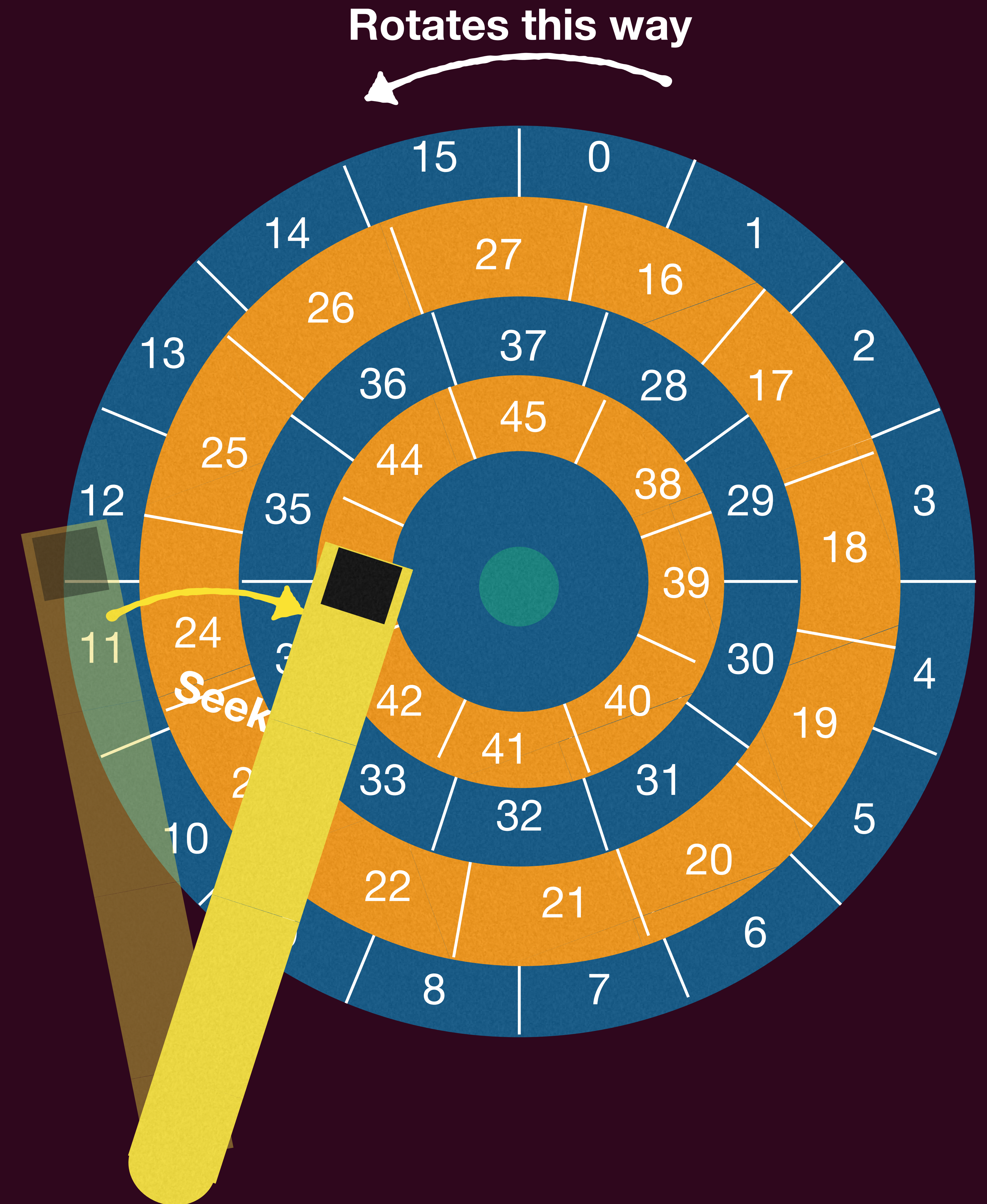
# 磁盘读写

- 原始IBM PC 360 KB软盘和（30 年后的）西部数据 WD 3000 HLFS 硬盘的磁盘参数

| parameter | IBM 360-KB floppy disk | WD 3000 HLFS hard disk |
|---|---|---|
| Number of cylinders | 40 | 36,481 |
| Tracks per cylinder | 2 | 255 |
| Sectors per track | 9 | 63 (avg) |
| Sectors per disk | 720 | 586,072,368 |
| Bytes per sector | 512 | 512 |
| Disk capacity | 360 KB | 300 GB |
| Seek time (adjacent cylinders) | 6 msec | 0.7 msec |
| Seek time (average case) | 77 msec | 4.2 msec |
| Rotation time | 200 msec | 6 msec |
| Time to transfer 1 sector | 22 msec | 1.4 usec |

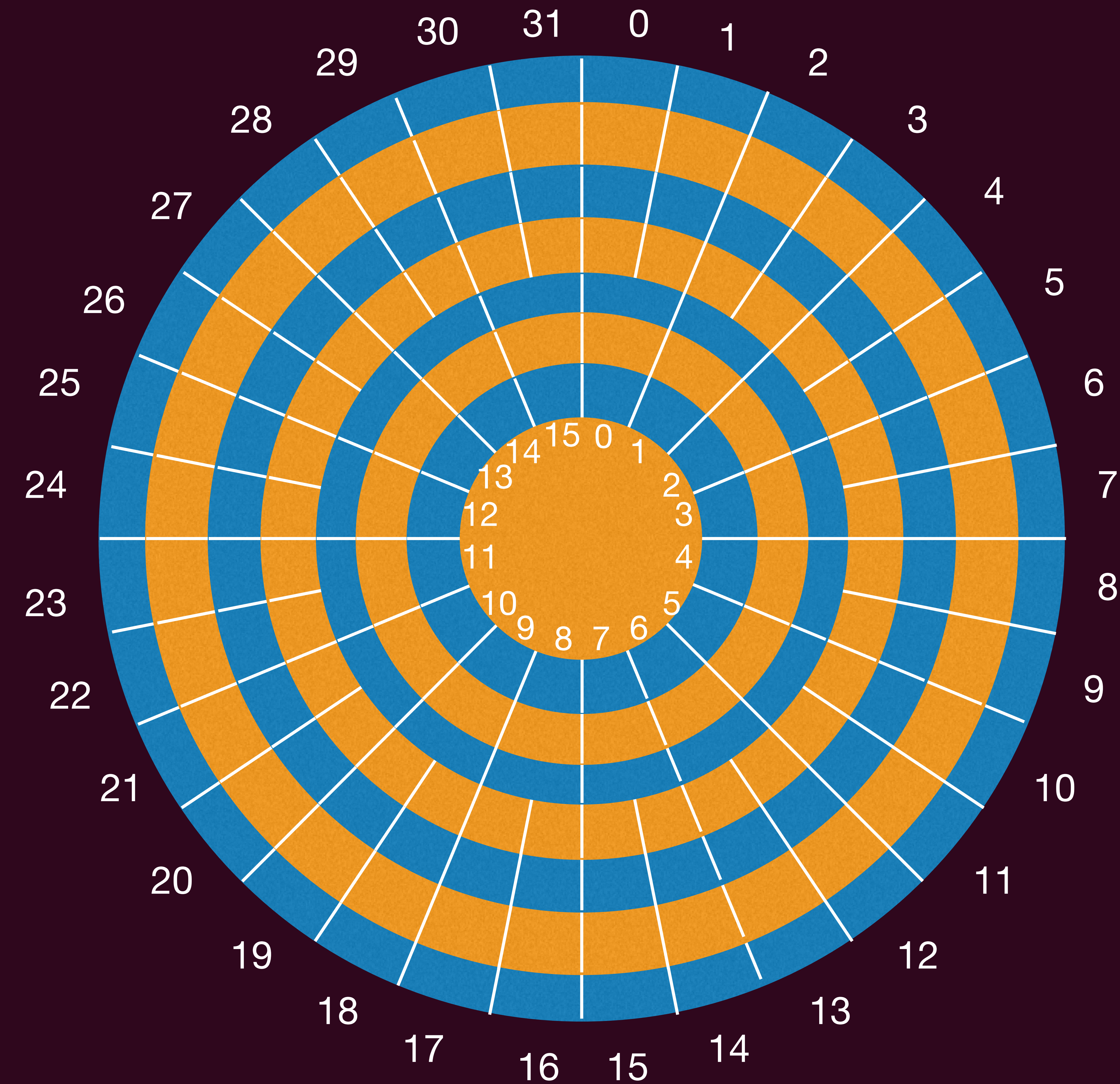
# 磁盘读写

- 磁道偏移（Track Skew）：确保在跨越磁道边界时能够正确处理顺序读取
    - 从一个磁道切换到另一个磁道时，磁盘需要时间重新定位磁头
    - 如果没有这样的偏移，磁头会被移动到下一个磁道，但所需的下一个数据块已经旋转到磁头后面了（得重新转一圈才能读到）



# 磁盘读写

- 多区域（Multi-Zoned） 磁盘驱动器：外部磁道比内部磁道有更多的扇区
  - 磁盘被组织成多个区域，每个区域是表面上一组连续的磁道
  - 每个区域的每条磁道的扇区数相同
- 现代磁盘支持逻辑块寻址（不考虑磁盘几何结构）



# 磁盘读写

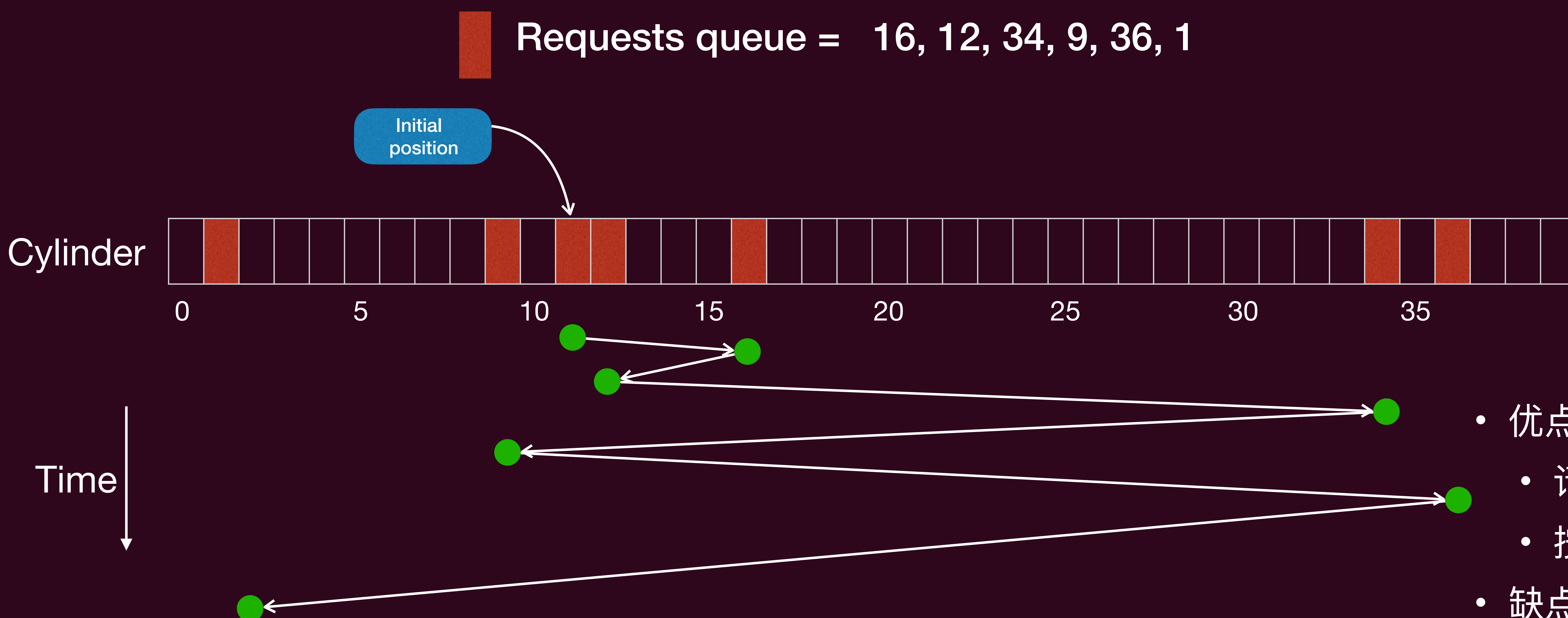
- 缓存（磁道缓冲区）：一些小容量的内存（8 到 16 MB），用于存储从磁盘读取或写入到磁盘的数据。
  - 读取扇区时，会读取该磁道上的所有扇区。
  - 写入时，有两种写入策略：当数据被放入缓存时（写回，write back）或数据实际写入磁盘后（直写, write through）才确认写入
  - 缓冲区在磁盘的逻辑板上

# 磁头调度

- 由于 I/O 成本高，操作系统历来在决定发往磁盘的 I/O 顺序方面发挥了作用。
- 面对一组 I/O 请求，磁盘调度程序会检查这些请求并决定下一个调度哪个请求。
    - 目标：通过磁头调度来最小化磁头移动，从而最大化磁盘 I/O 吞吐量

# FCFS

- 最简单的方法是先到先服务（FCFS），即按请求到达的顺序处理磁盘请求。

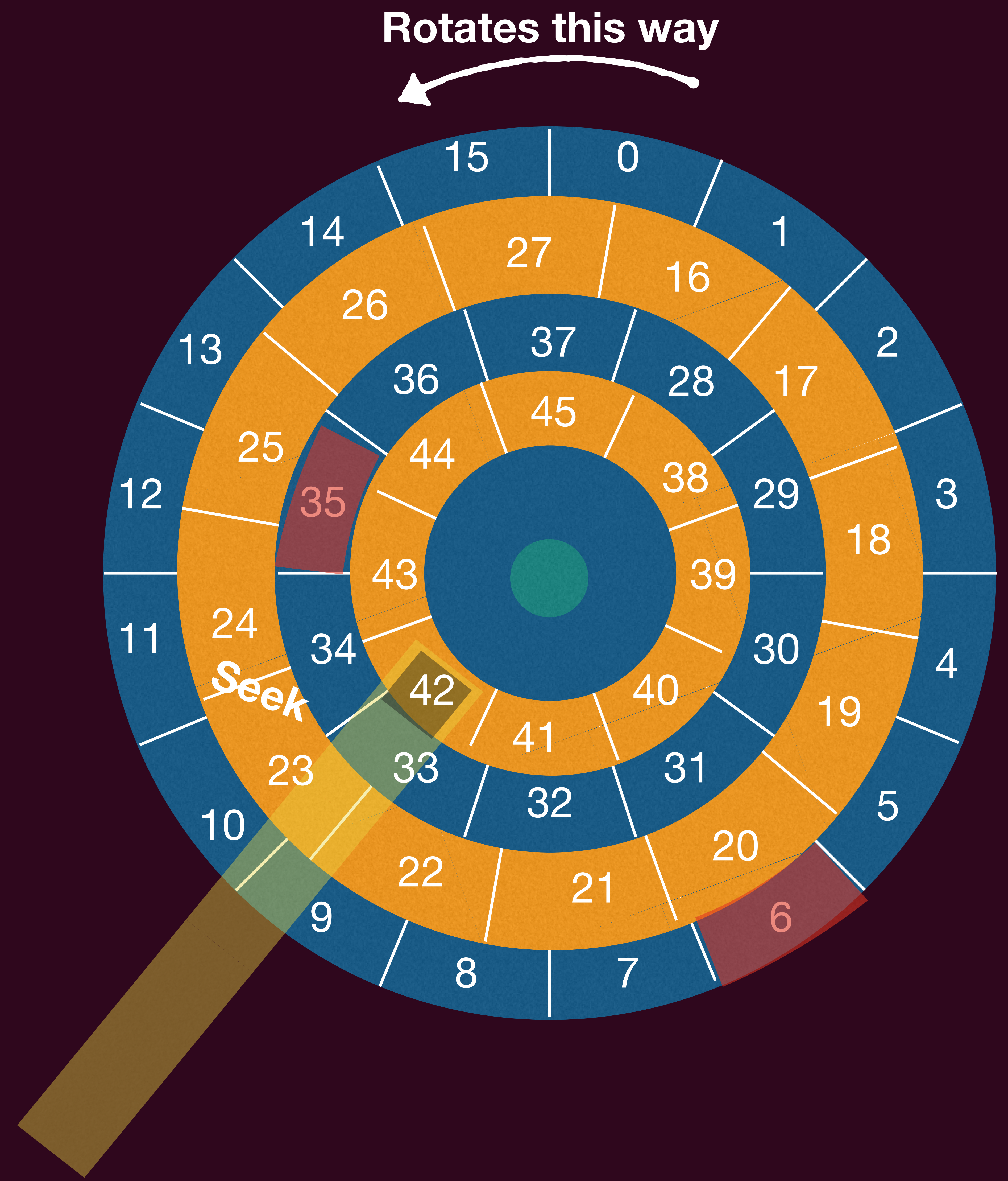



磁头总共需要移动 5 + 4 + 22 + 25 + 27 + 35 =118 个磁道

- 优点
  - 请求之间的公平性
  - 按应用程序预期的顺序
- 缺点
  - 到达的位置可能在磁盘上的随机点，导致较长的寻道时间。

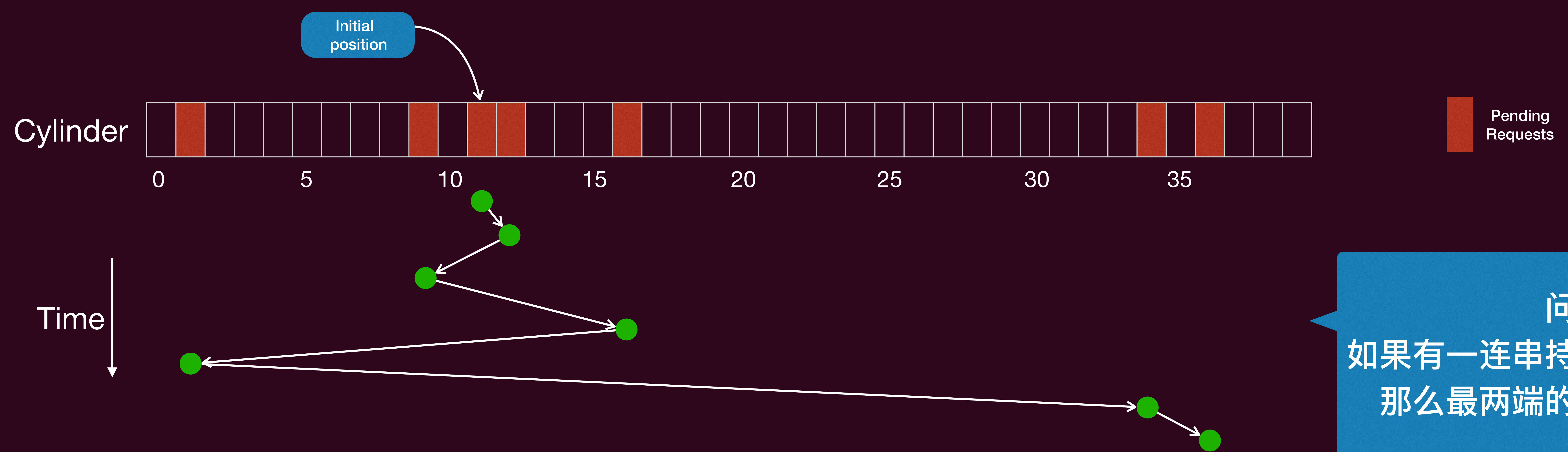
# Shortest Seek Time First(SSTF)

- 按磁道顺序排列 I/O 请求队列，优先处理最接近磁道的请求。
  - 最小化寻道时间。
  - 假设当前磁头位置在内轨道上，则依次处理请求 35（中间），然后再处理请求 6（外部）



# Shortest Seek Time First(SSTF)

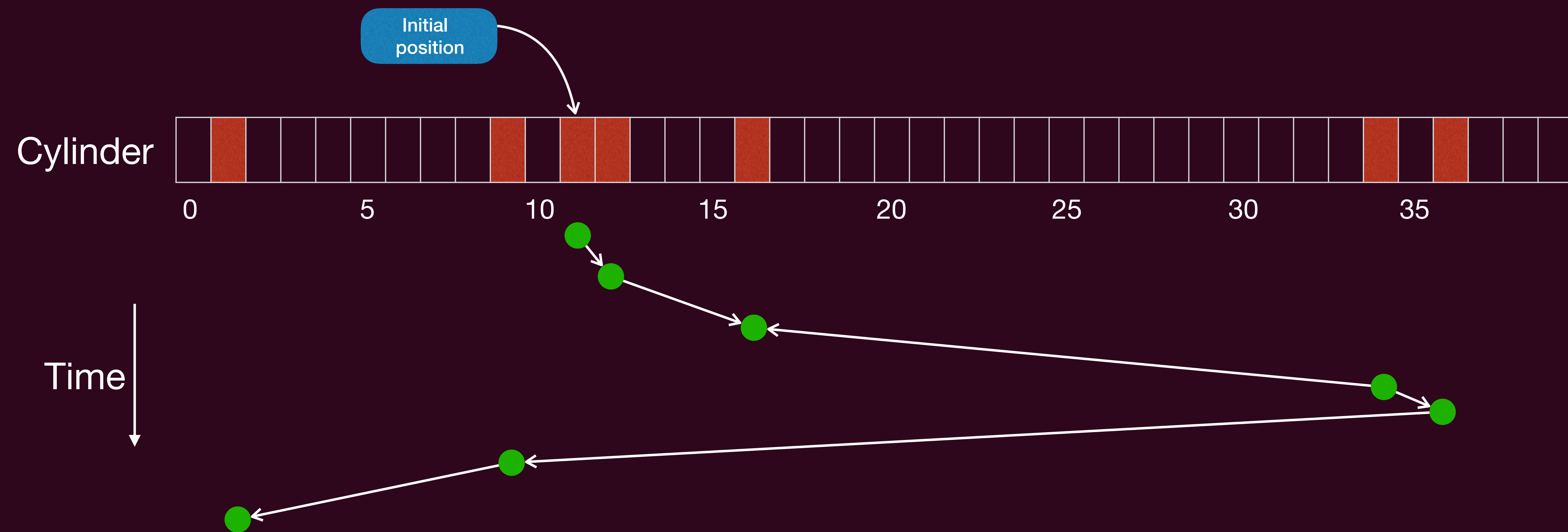
- 假设第一个请求是磁道（柱面）11，随后依次有新请求进入，分别是磁道 1、36、16、34、9 和 12。
  - 最短寻道时间优先：1 + 3 + 7 + 15 + 33 + 2 = 61 个磁道




问题：饿死！
如果有一连串持续不断的中间磁道请求，那么最两端的磁道请求将得不到服务

# 电梯算法 (也叫扫描算法， SCAN)

- 简单地在磁盘上来回移动，按顺序跨磁道处理请求（像电梯一样运行）。
  - 从磁盘的一端（从外轨到内轨或从内轨到外轨）的单次通过称为一次扫描。
  - 磁臂移动距离为：1 + 4 + 18 + 2 + 27 + 8 = 60 个磁道。

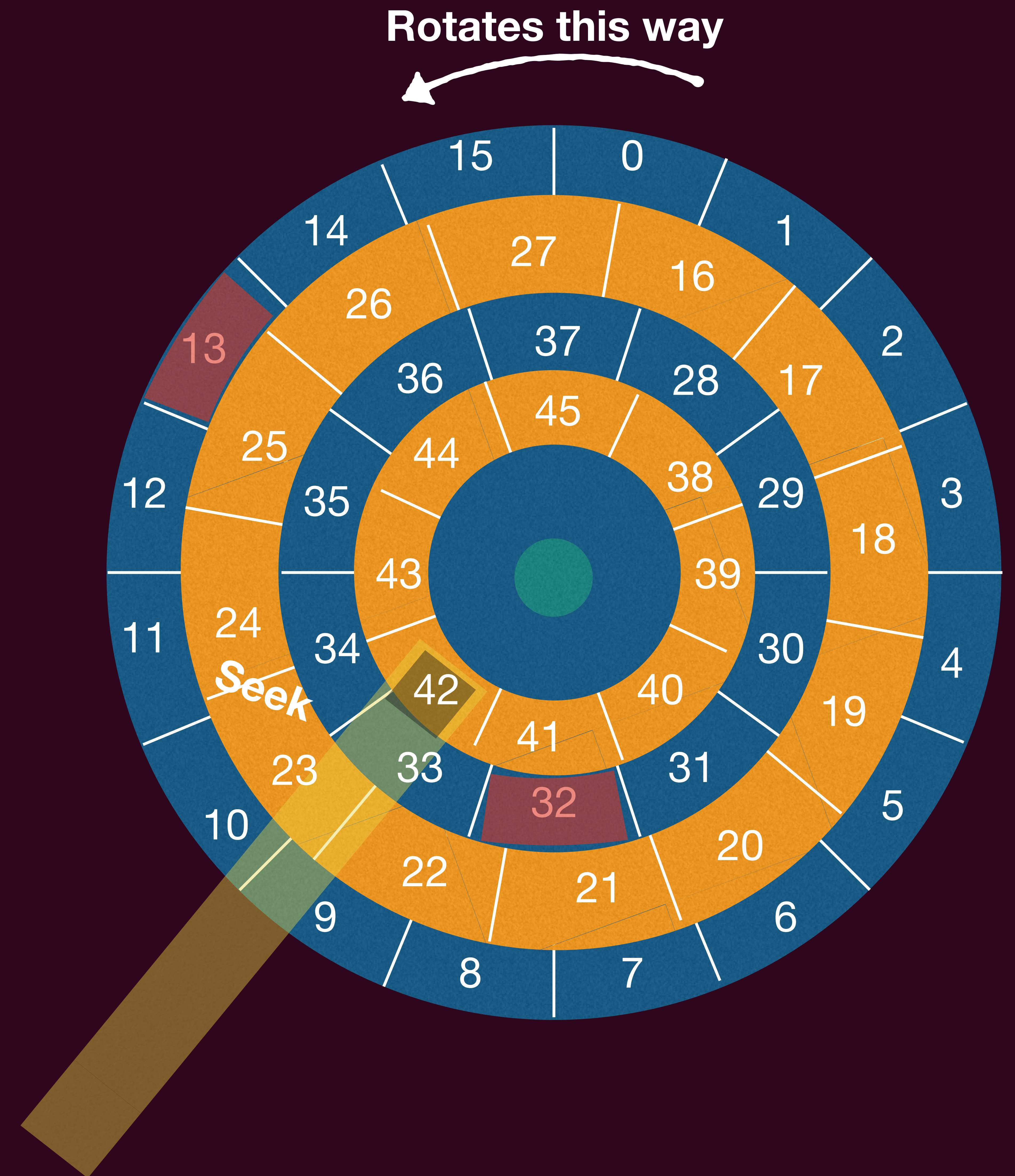


# 电梯算法

- 有多种变体可用：
  - F-SCAN：在执行扫描时暂时冻结要处理的队列（避免远距离请求的饥饿）。
  - C-SCAN：只从外轨到内轨扫描，然后重置到外轨重新开始
    - 更平均的等待时间，即对内轨和外轨更公平

# Shortest Positioning Time First (SPTF)

- 同时考虑寻道时间和旋转时间。
- 假设当前磁头位置在第 42 扇区（内轨），那么是处理请求 32（中间）还是请求 13（外轨）？
    - 如果寻道时间远高于旋转延迟，那么最短寻道优先（SSF）是好的选择。
    - 但如果寻道速度比旋转快得多，那么应先处理请求 13。
- 在操作系统中实现这一点很困难，因为操作系统通常不清磁头当前的具体位置

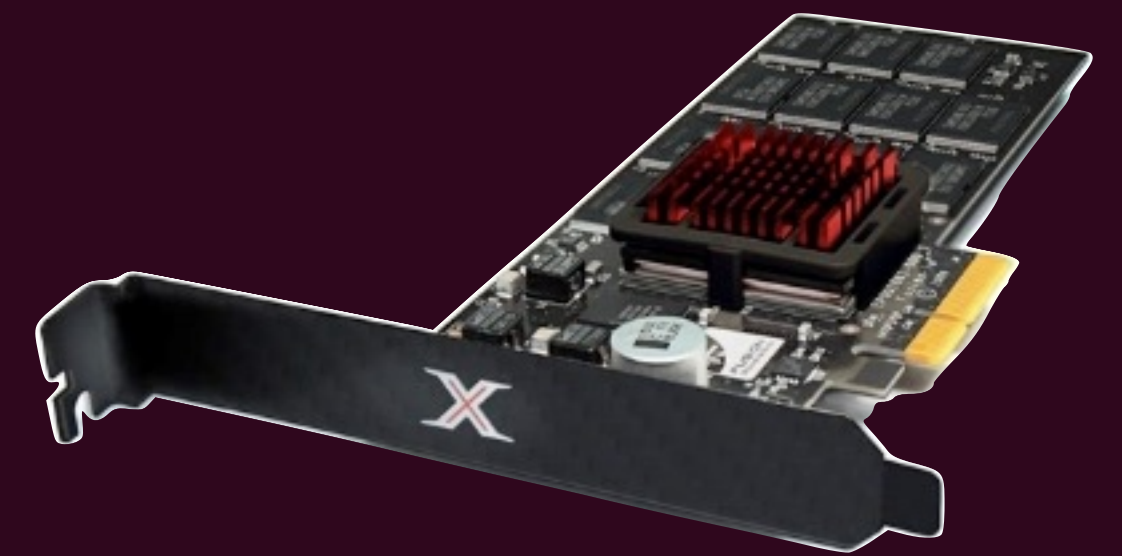


# 磁盘调度

- 过去的操作系统非常注重磁盘请求调度。
- 目前的做法是将许多请求发送到磁盘，让磁盘自行调度。如今的磁盘更智能，并且拥有较大的缓存。
  - 然而，操作系统至少可以假设相邻的扇区号在磁盘上也相邻，从而顺序访问会更快。

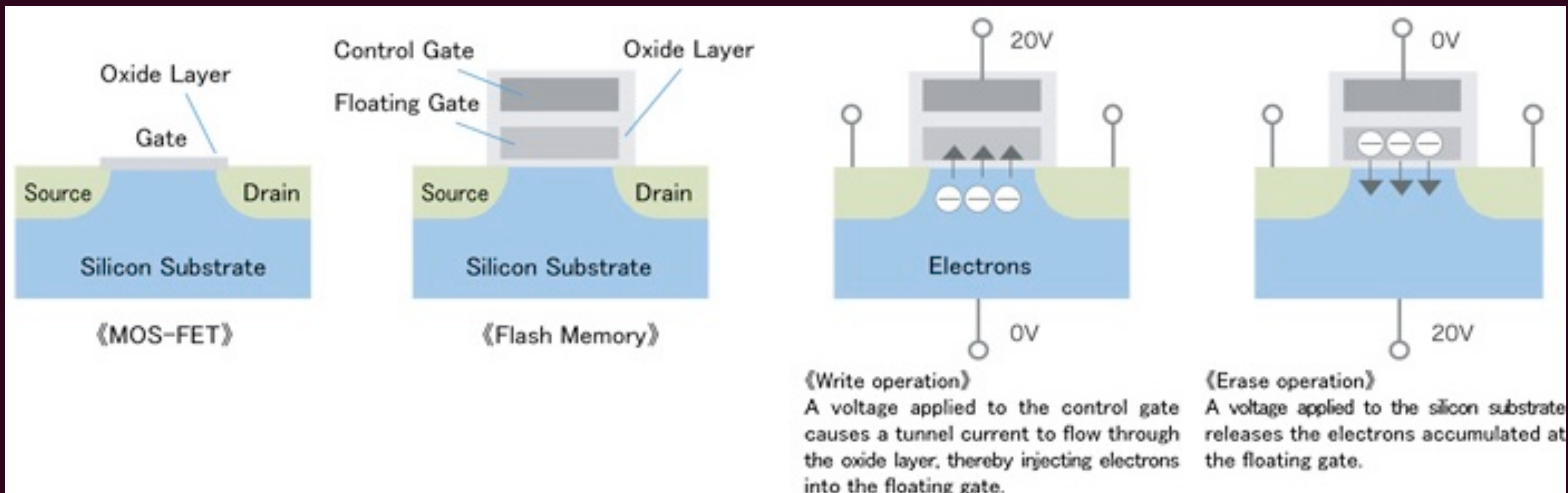
# 固态硬盘(Solid State Disks, SSDs)

- 使用 NAND 多级单元闪存存储器
  - 可寻址扇区（4 KB 页面），但每个存储块存储 4-64 个“页面”
  - 被困电子区分 1 和 0
- 没有运动部件（没有旋转/寻道电机）
  - 消除了寻道和旋转延迟（< 0.1-0.2 毫秒的访问时间）
  - 非常低的功耗和轻量级
  - 容量越大，读写速度越快 (电路级并行)
- 容量和成本方面的快速进展一直持续不断！



# 固态硬盘(Solid State Disks, SSDs)

- Flash Memory “闪存”
  - Floating gate 的充电/放电实现 1-bit 信息的存储



# 固态硬盘(Solid State Disks, SSDs)

- 放电 (erase) 做不到 100% 放干净
  - 放电数千/数万次以后，就好像是 “充电” 状态了
  - Dead cell; “wear out”
    - 必须解决这个问题 SSD 才能实用

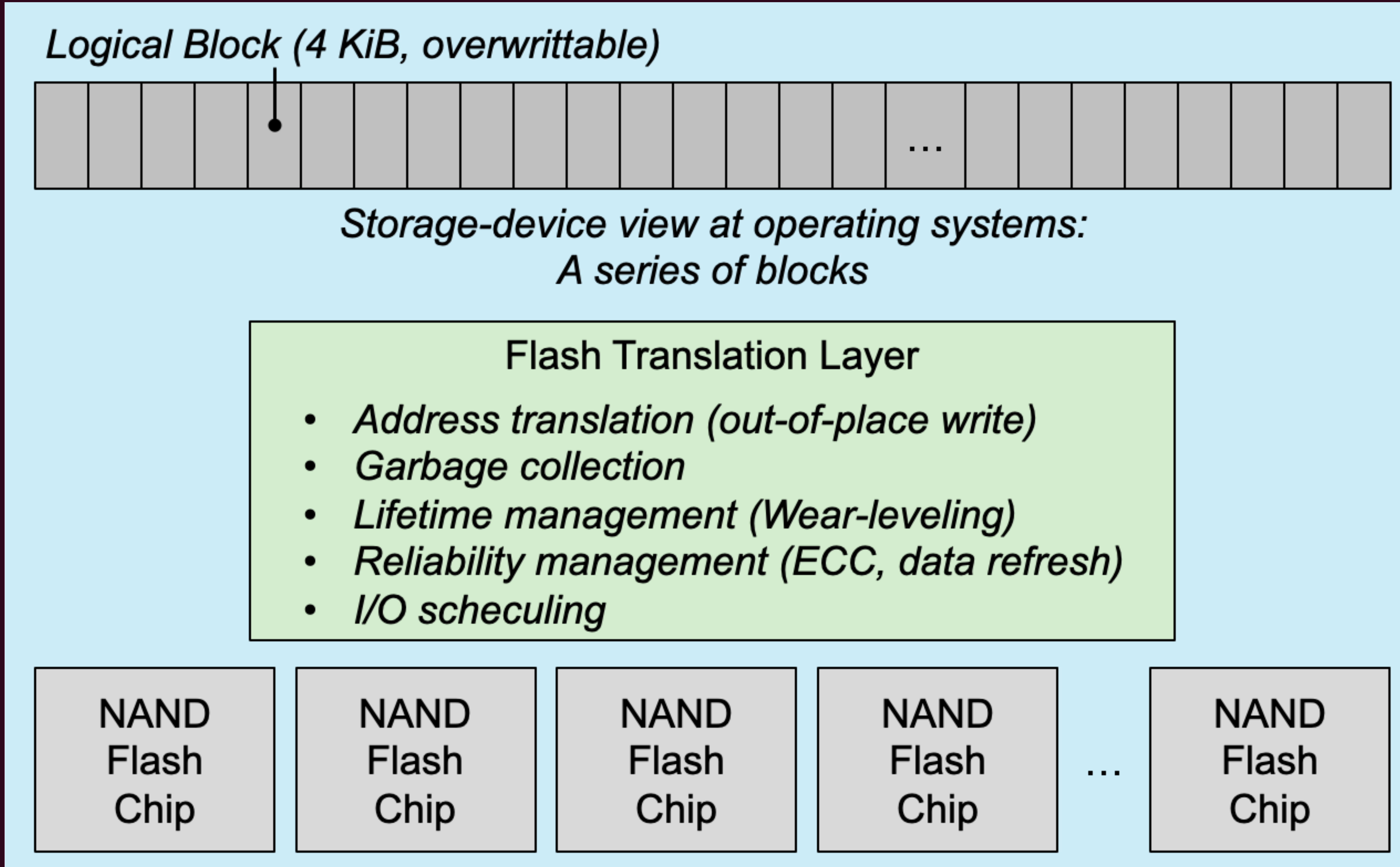
# 解决方案

- 间接层
  - 在固态硬盘（SSD）中维护一个闪存转换层（FTL）。
  - 将虚拟块编号（操作系统使用的）映射到物理页面编号（闪存存储控制器使用的）。
    - 现在可以自由重新定位数据，而无需操作系统知道。
- 写时复制
  - 当操作系统更新其数据时，不要覆盖页面（这样做很慢，需要先擦除页面，而擦除的效率很低，比写入低）。
  - 相反，将新版本写入一个空闲页面。
  - 更新 FTL 映射以指向新位置。

# 解决方案

- 固态硬盘（SSD）控制器可以分配映射以在页面之间均衡工作负载
  - 均衡磨损（Wear Leveling）
- 对旧版本页面应该怎么处理?
  - 在后台进行垃圾收集
  - 擦除带有旧页面的块，将其添加到空闲列表

# 闪存转换层（FTL）



# 新的问题

- Copy-on-write 意味着旧的数据还在！
  - logic block 被覆盖，physical block 依然存储了数据
    - 有研究发现实际的智能手机的一些被删除的数据之后仍然“存在”超过六个月
- 轻度格式化同样存在相同的问题

# 总结

- 操作系统同样利用抽象技术管理I/O
  - 各种drivers就是为了实现统一的接口
  - 另外一个视角：I/O就是各个CPU（通用和专用）的交流，drivers就是解释器！
- 硬盘
  - 磁盘和SSD
  - 各自有优缺点

# 阅读材料

- [OSTEP] 第36, 37, 38, 44章