哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

操作系统

Operating Systems

刘川意　教授
liuchuanyi@hit.edu.cn

哈尔滨工业大学（深圳）

2021年10月

Module 6: I/O与存储

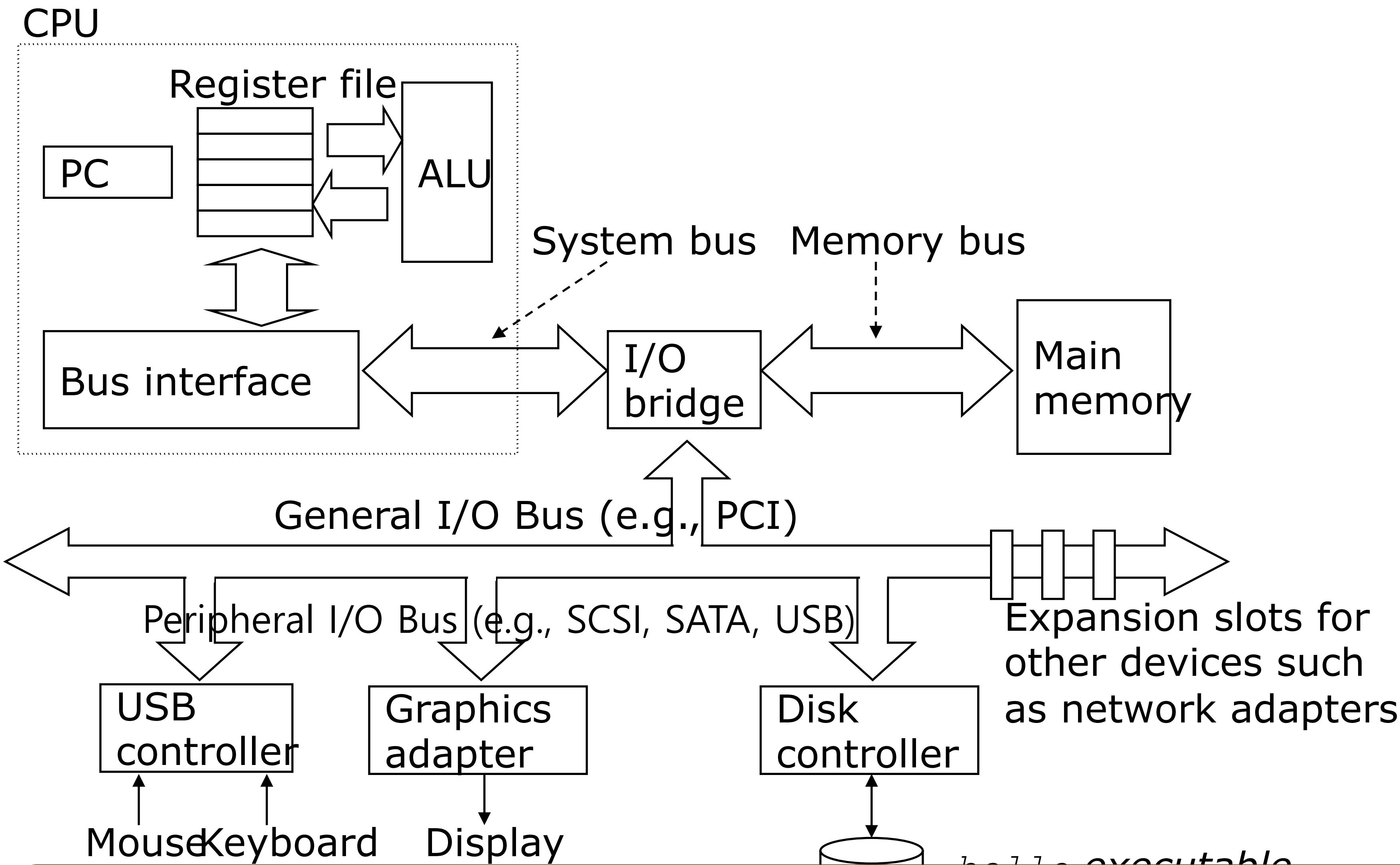
1. **I/O devices**
2. Hard Disk Drives
3. **RAID**
4. 其他I/O设备

I/O Devices

- I/O is **critical** to computer system to **interact with systems.**

- **I/O需要解决的重要问题：**
  - How should I/O be integrated into systems?
  - What are the general mechanisms?
  - How can we make the efficiently?

Structure of input/output (I/O) device



CPU is attached to the main memory of the system via some kind of memory bus.

Some devices are connected to the system via a general I/O bus.

I/O Architecture

- Buses
  - Data paths that provided to enable information between CPU(s), RAM, and I/O devices. 不同的I/O设备（如键盘、鼠标、磁盘等）需要通过相应的接口电路与总线相连接，这些接口电路由“控制器”或“适配器”提供（后面统称为“设备控制器”）。不同的设备控制器能够支持不同的接口协议

- I/O bus
  - Data path that connects a CPU to an I/O device.
  - I/O bus is connected to I/O device by three hardware components: I/O ports, interfaces and device controllers.
  - 根据接口协议的性能区别，现代计算机对I/O总线进行了分层。在上图中，图像或者其他高性能的I/O设备通过常规的I/O总线连接到系统，在许多现代系统中会是PCI或它的衍生形式。而一些相对较慢的I/O设备则通过外围总线（peripheral bus）连接到系统，比如使用SCSI、SATA或者USB等协议的I/O设备

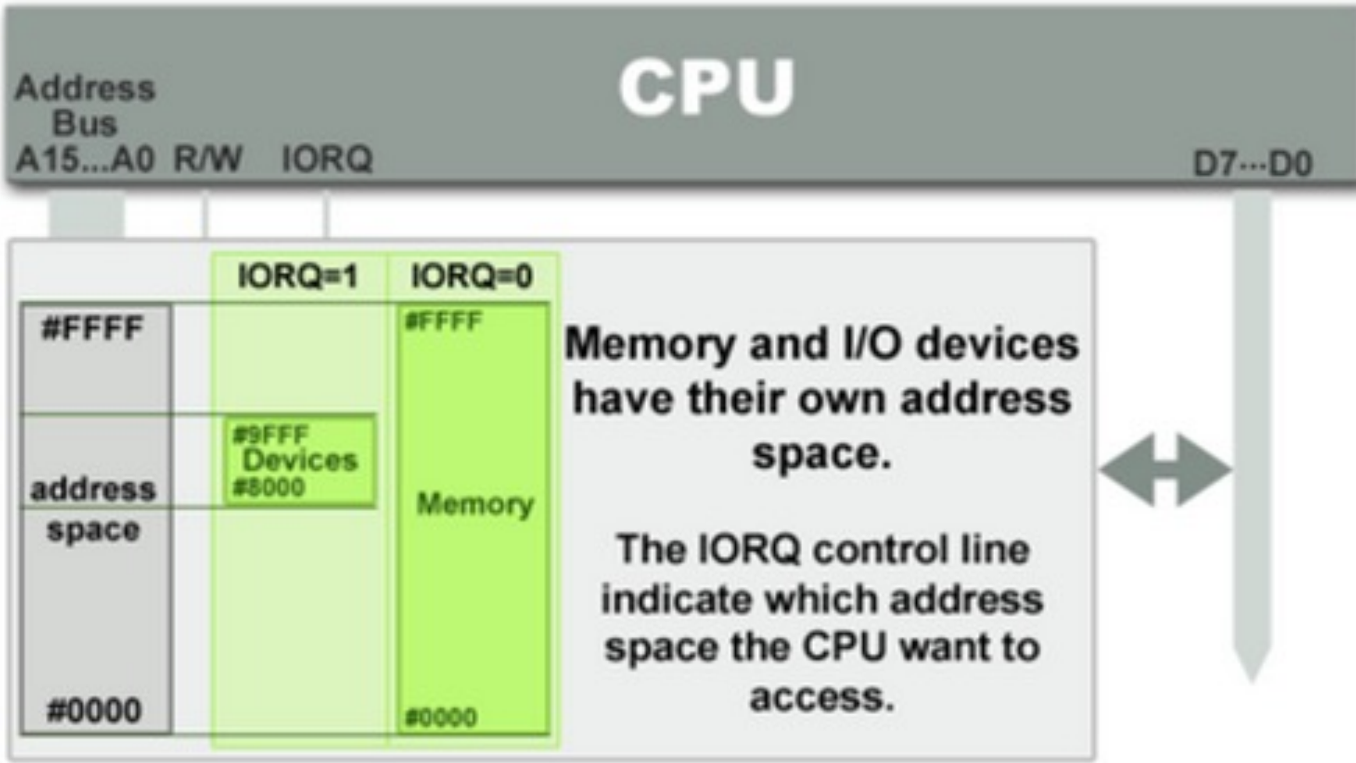
Device interaction

- How the OS communicates with the **device**? 主机对I/O设备进行访问的目标是I/O设备的寄存器或者内存。常见的I/O设备都只提供寄存器供主机访问，对于低速外设这样的模式是足够的，但是对于需要大量、高速数据交互的外设（如显卡、网卡），就需要主机能够直接访问外设的内存了

- Solutions：现代计算机提供了两种方式来访问I/O设备，它们分别是PMIO和MMIO
  - PMIO：端口映射I/O（Port-mapped I/O）。将I/O设备独立看待，并使用CPU提供的专用I/O指令访问；I/O instructions: a way for the OS to send data to specific device registers.
    - Ex) `in` and `out` instructions on x86
  - MMIO：内存映射I/O（Memory-mapped I/O）。将I/O设备看作内存的一部分，不使用单独的I/O指令，而是使用内存读写指令访问；memory-mapped I/O
    - Device registers available as if they were memory locations.
    - The OS `load` (to read) or `store` (to write) to the device instead of main memory.

PMIO（Port-mapped I/O）

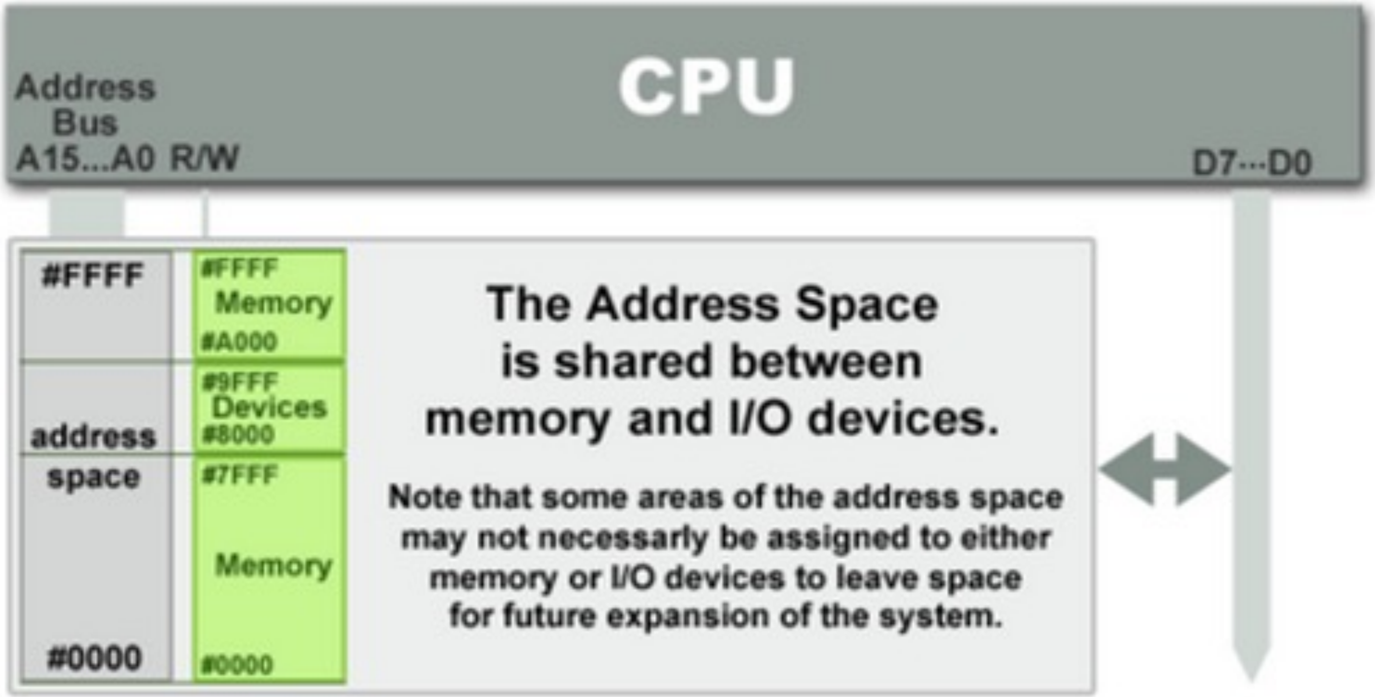
- 端口映射I/O，又叫做被隔离的I/O（isolated I/O），它提供了一个专门用于I/O设备“注册”的地址空间，该地址空间被称为I/O地址空间，最大寻址范围为64K



- 为了使I/O地址空间与内存地址空间隔离，要么在CPU物理接口上增加一个I/O引脚，要么增加一条专用的I/O总线。因此，并不是所有的平台都支持PMIO，常见的ARM平台就不支持PMIO。支持PMIO的CPU通常具有专门执行I/O操作的指令，例如在Intel-X86架构的CPU中，I/O指令是in和out，这两个指令可以读/写1、2、4个字节（outb, outw, outl）从内存到I/O接口上。由于I/O地址空间比较小，因此I/O设备一般只在其中“注册”自己的寄存器，之后系统可以通过PMIO对它们进行访问

MMIO（Memory-mapped I/O）

- 在MMIO中，物理内存和I/O设备共享内存地址空间（注意，这里的内存地址空间实际指的是内存的物理地址空间）



- 当CPU访问某个虚拟内存地址时，该虚拟地址首先转换为一个物理地址，对该物理地址的访问，会通过南北桥（现在被合并为I/O桥）的路由机制被定向到物理内存或者I/O设备上。因此，用于访问内存的CPU指令也可用于访问I/O设备，并且在内存（的物理）地址空间上，需要给I/O设备预留一个地址区域，该地址区域不能给物理内存使用。
- MMIO是应用得最为广泛的一种I/O方式，由于内存地址空间远大于I/O地址空间，I/O设备可以在内存地址空间上暴露自己的内存或者寄存器，以供主机进行访问

PCI设备

- PCI及其衍生的接口（如PCIE）主要服务于高速I/O设备（如显卡或网卡），使用PCI接口的设备又被称为PCI设备。与慢速I/O设备不同，计算机既需要访问它们的寄存器，也需要访问它们的内存。
- 每个PCI设备都有一个配置空间（实际就是设备上一组连续的寄存器），大小为256byte。配置空间中包含了6个BAR(Base Address Registers，基址寄存器)，BAR中记录了设备所需要的地址空间类型、基址以及其他属性

Memory Space BAR Layout

31 - 4	3	2 - 1	0
16-Byte Aligned Base Address	Prefetchable	Type	Always 0

I/O Space BAR Layout

31 - 2	1	0
4-Byte Aligned Base Address	Reserved	Always 1

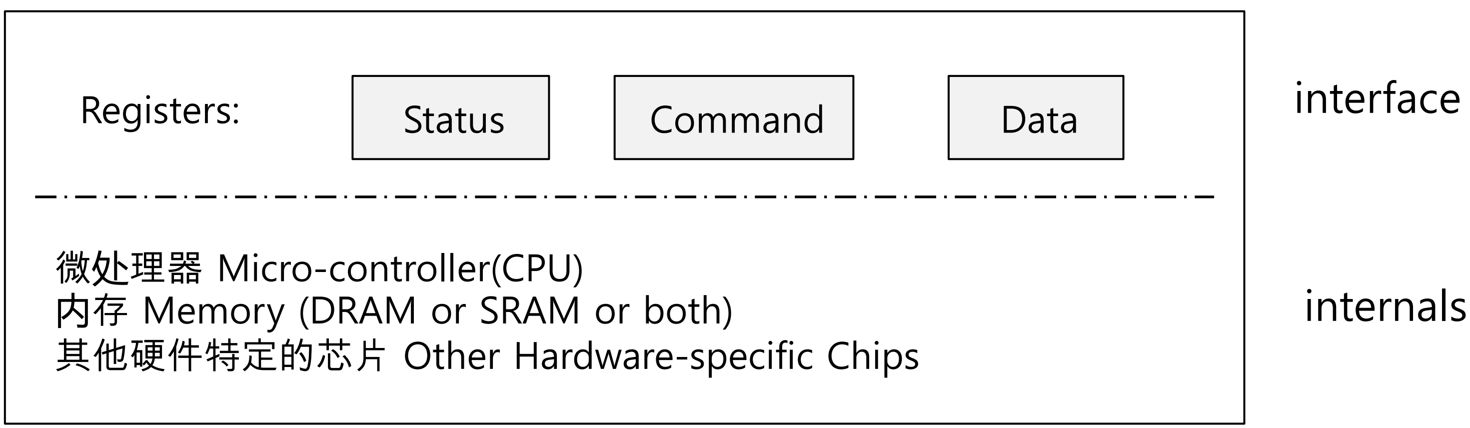
PCI设备（cont.）

- 可以看到，PCI设备能够申请两类地址空间，即内存地址空间和I/O地址空间，它们用BAR的最后一位区别开来。因此，PCI设备可以通过PMIO和MMIO将自己的I/O存储器（Registers/RAM/ROM）暴露给CPU（通常寄存器使用PMIO，而内存使用MMIO的方式暴露）。
- 配置空间中的每个BAR可以映射一个地址空间，因此每个PCI设备最多能映射6段地址空间，但实际上很多设备用不了这么多。PCI配置空间的初始值是由厂商预设在设备中的，也就是说，设备需要哪些地址空间都是其自己定的，这可能会造成不同的PCI设备所映射的地址空间冲突，因此在PCI设备枚举（也叫总线枚举，由BIOS或者OS在启动时完成）的过程中，会重新为其分配地址空间，然后写入PCI配置空间中。
- 在PCI总线之前的ISA总线是使用跳线帽来分配外设的物理地址，每插入一个新设备都要改变跳线帽以分配物理地址，这是十分麻烦且易错的，但这样的方式似乎我们更容易理解。能够分配自己总线上挂载设备的物理地址这也是PCI总线相较于I2C、SPI等低速总线一个最大的特色

10

标准外设（Canonical Device）

- Canonical Devices has two important components.
  - **Hardware interface** allows the system software to control its operation. 硬件接口本质就是I/O设备提供的各式寄存器，系统软件通过与这些寄存器进行交互，达到**控制**I/O设备的目的
  - **Internals** which is implementation specific. 实现硬件接口提供的功能，不同的I/O设备具有不同功能，因此它们的内部实现和包含的元器件也不尽相同
  - 使用I/O设备的目的是为了交互数据，不管是网卡、磁盘，亦或是键盘，总归要将数据进行输入输出



Canonical Device

11

Hardware interface of Canonical Device

- **status register**
  - See the current status of the device
- **command register**
  - Tell the device to perform a certain task
- **data register**
  - Pass data to the device, or get data from the device

By reading and writing above three registers, the operating system can control device behavior.

- Typical interaction example

```
while ( STATUS == BUSY)
    ; //wait until device is not busy
write data to data register
write command to command register
    (Doing so starts the device and executes the command)
while ( STATUS == BUSY)
    ; //wait until device is done with your request
```

Polling

- Operating system waits until the device is ready by **repeatedly** reading the status register. Device driver通过轮询读取设备状态
- 一般来说，主机与I/O设备要进行数据交互，会经过这样一个过程：
  - CPU通过I/O设备的硬件接口（以下简称I/O接口）获取设备状态（即状态寄存器的值），只有“就绪”状态的设备才能进行数据传输。
  - CPU通过I/O接口下达交互指令：如果是读数据，则向I/O接口的命令寄存器输入要获取的数据在I/O设备的内部位置以及读设备指令；如果是写数据，则向I/O接口的命令寄存器输入要存放的数据在I/O设备的内部位置、写设备指令，以及向数据寄存器写入数据。
  - I/O设备内部根据I/O接口中寄存器的值，开始执行数据传输工作。
  - CPU在I/O设备完成工作后，执行其他操作，完成数据传送。

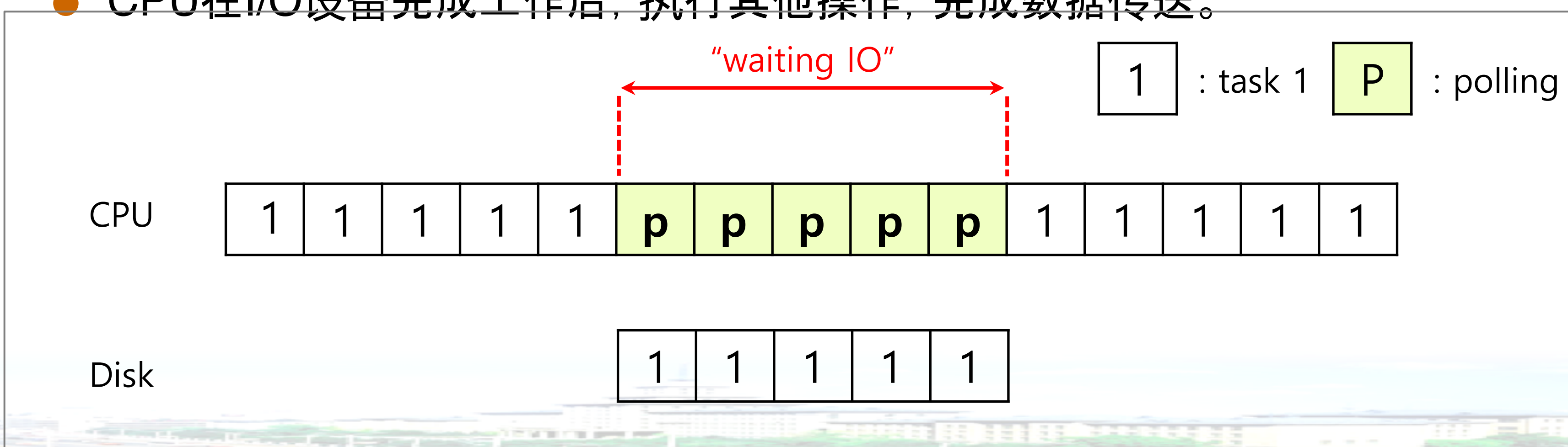


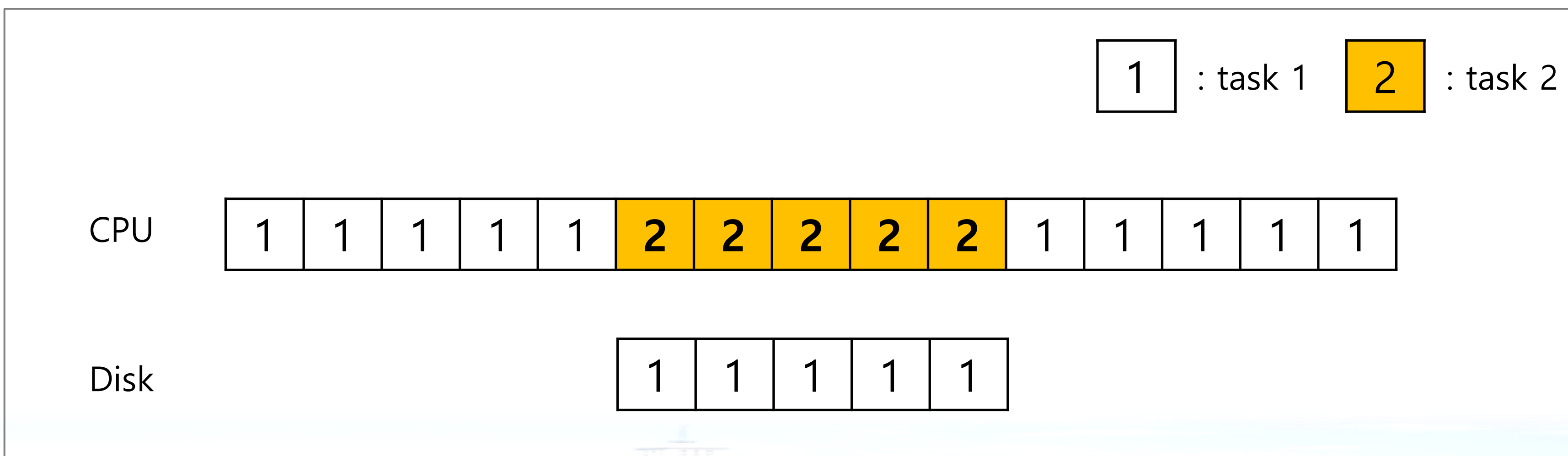
Diagram of CPU utilization by polling

Polling Evaluation

- 标准交互流程实现起来比较简单，但是难免会有一些低效和不方便。第一个问题就是轮询过程比较低效，在等待设备是否满足某种状态时浪费大量CPU时间(下图描述的就是磁盘在执行数据传输过程中，CPU不能执行其他任务，只能等待传输完成)，如果此时操作系统可以切换执行下一个就绪进程，就可以大大提高CPU的利用率。
  - Positive aspect is simple and working.
  - **However, it wastes CPU time just waiting for the device**.
    - Switching to another ready process is better utilizing the CPU.

interrupts

- 有了中断机制，CPU向设备发出I/O请求后，就可以让对应进程进入睡眠等待，从而切换执行其他进程。当设备完成I/O请求后，它会抛出一个硬件中断，引发CPU跳转执行操作系统预先定义好的中断处理程序，中断处理程序会挂起正在执行的进程，同时唤醒等待I/O的进程并继续执行
- **Put the I/O request process to sleep** and context switch to another. OS调度程序让执行I/O request的进程进入sleep状态，并切换成另一个进程去执行
- When the device is finished, wake the process waiting for the I/O by **interrupt**. 外设通过中断机制唤醒处于sleep状态的等待进程
  - Positive aspect is allow to **CPU and the disk are properly utilized.**



在磁盘执行进程1的I/O过程中，CPU同时执行进程2，并且在I/O请求执行完毕后，回过头来再次执行进程1

Polling vs interrupts

- *However,* **"interrupts is not always the best solution"**
  - If, device performs very quickly, interrupt will "slow down" the system.
  - Because **context switch is expensive (switching to another process)**

If a device is fast → poll is best.
If it is slow → interrupts is better.

CPU is once again over-burdened

- CPU **wastes a lot of time** to copy the *a large chunk of data* from memory to the device.

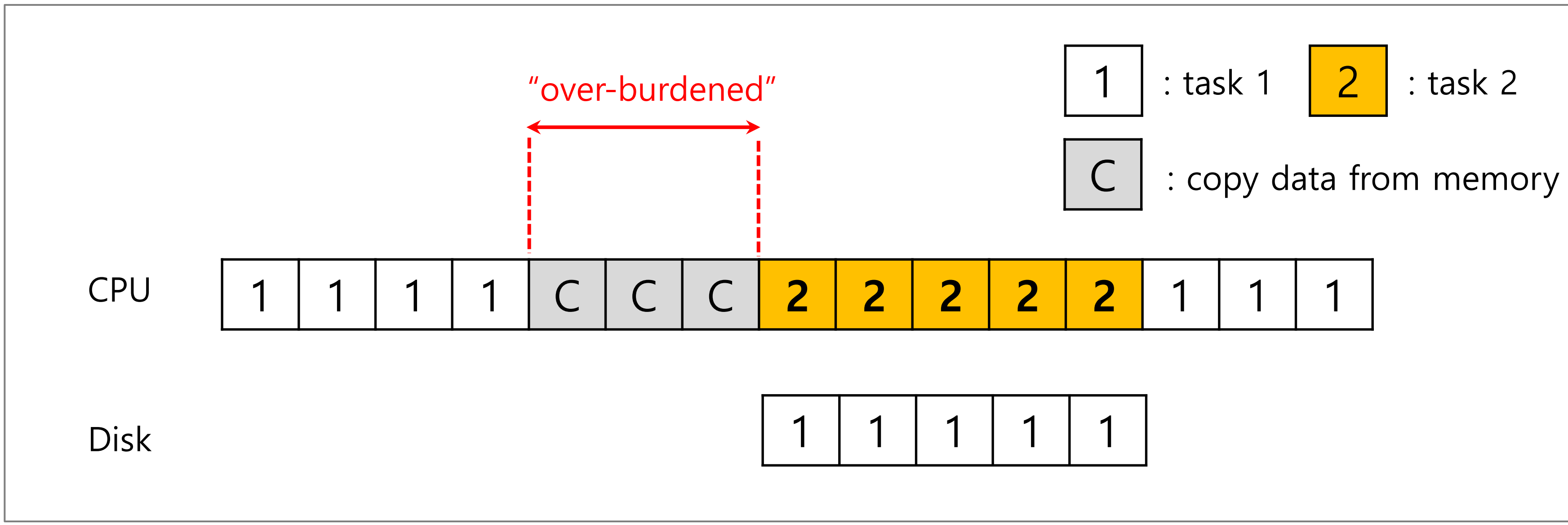


Diagram of CPU utilization

17

DMA (Direct Memory Access)

- **Copy data** in memory by knowing "where the data lives in memory, how much data to copy"
- When completed, DMA raises an interrupt, I/O begins on Disk.

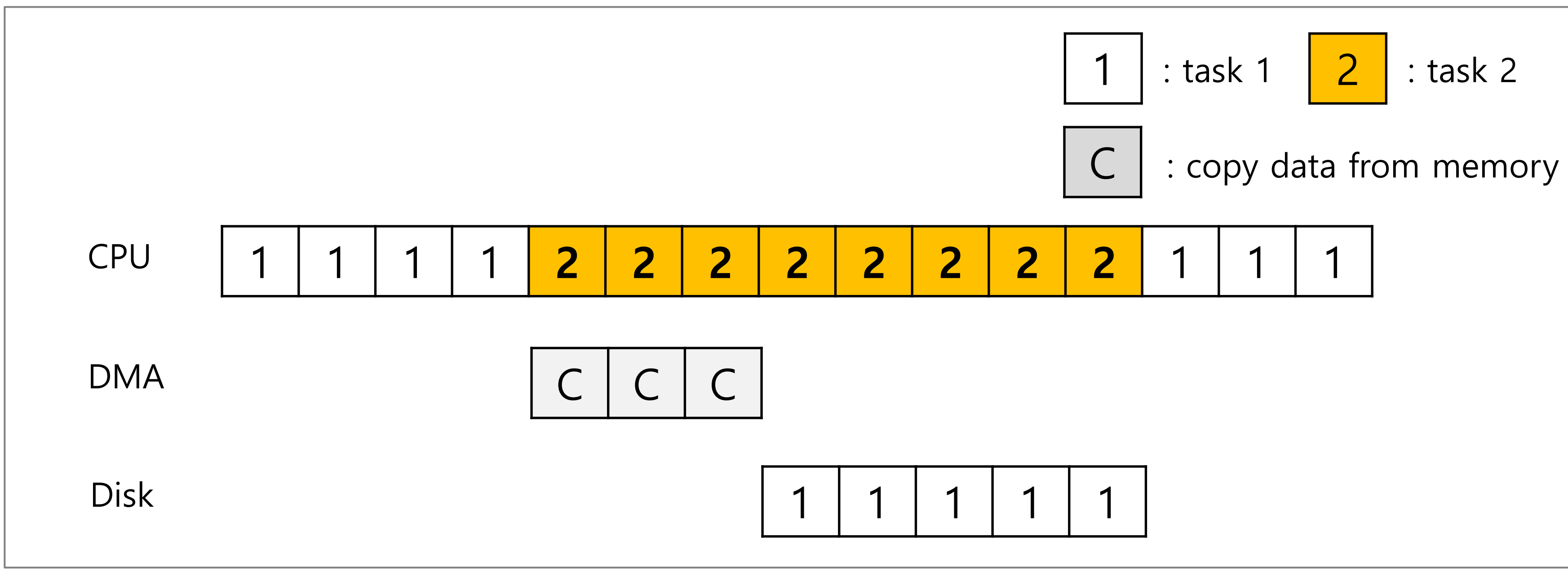


Diagram of CPU utilization by DMA

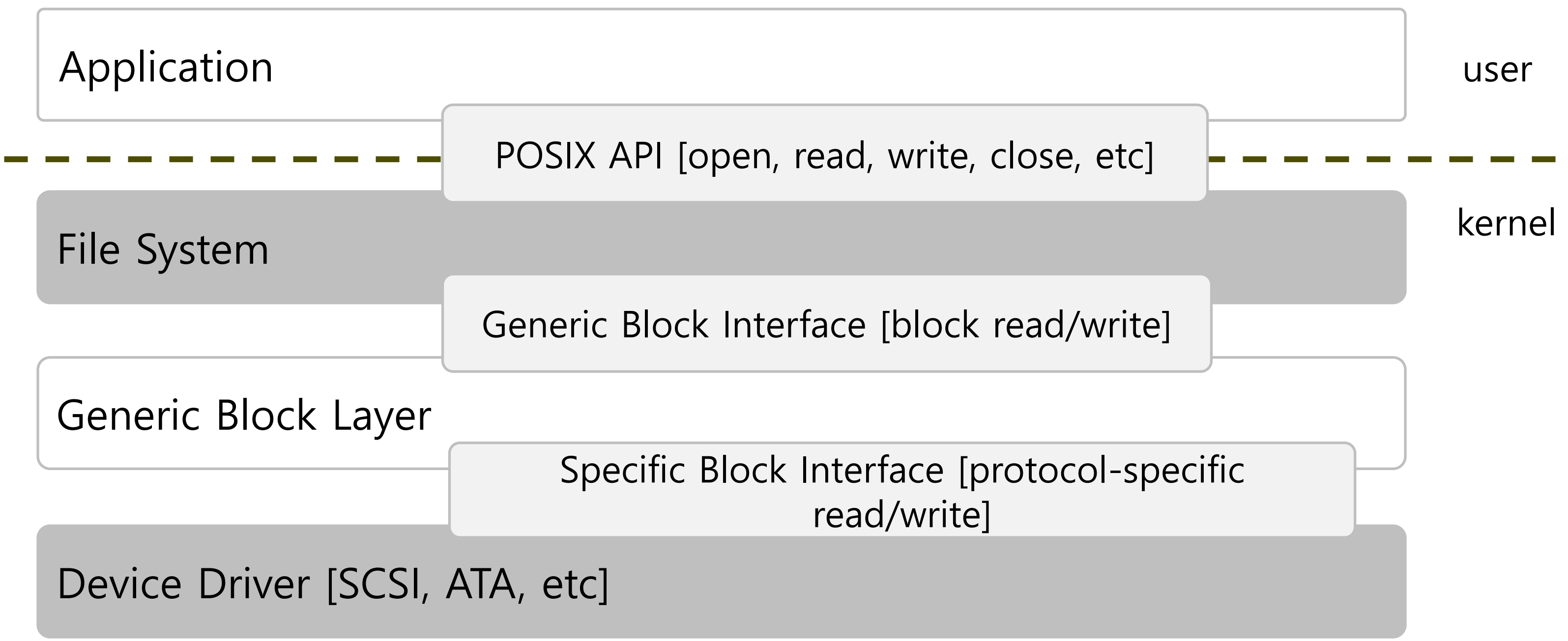
18

Device interaction (Cont.)

- How the OS interact with **different specific interfaces**?
  - Ex) We'd like to build a file system that worked on top of SCSI disks, IDE disks, USB keychain drivers, and so on.
- Solutions: Abstraction
  - Abstraction encapsulate any specifics of device interaction.

File system Abstraction

- File system specifics of which disk class it is using.
  - Ex) It issues **block read** and **write** request to the generic block layer.



The File System Stack

20

Problem of File system Abstraction

- If there is a device having many special capabilities, these capabilities will go unused in the generic interface layer.

- Over 70% of OS code is found in device drivers.
  - Any device drivers are needed because you might plug it to your system.
  - They are primary contributor to **kernel crashes**, making **more bugs**.

A Simple IDE Disk Driver

- Four types of register
  - Control, command block, status and error
  - Memory mapped IO
  - `in` and `out` I/O instruction

22

IDE Device Interface

- 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):

7	6	5	4	3	2	1	0
BUSY	READY	FAULT	SEEK	DRQ	CORR	IDDEX	ERROR

- Error Register (Address 0x1F1): (check when Status ERROR==1)

7	6	5	4	3	2	1	0
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

23

OS跟设备交互的典型协议

- **Wait for drive to be ready**. Read Status Register (0x1F7) until drive is not busy and READY.
- **Write parameters to command registers**. Write the sector count, logical block address (LBA) of the sectors to be accessed, and drive number (master=0x00 or slave=0x10, as IDE permits just two drives) to command registers (0x1F2-0x1F6).
- **Start the I/O**. by issuing read/write to command register. Write READ—WRITE command to command register (0x1F7).
- **Data transfer (for writes)**: Wait until drive status is READY and DRQ (drive request for data); write data to data port.
- **Handle interrupts**. In the simplest case, handle an interrupt for each sector transferred; more complex approaches allow batching and thus one final interrupt when the entire transfer is complete.
- Error handling. After each operation, read the status register. If the ERROR bit is on, read the error register for details.

Wait for drive to be ready

```c
static int ide_wait_ready()
/* ensure the drive is ready before issuing a request to it  */
{
    while (((int r = inb(0x1f7)) & IDE_BSY) ||
                !(r & IDE_DRDY))
        ; // loop until drive isn’t busy
}
```

Write parameters to command registers

```
static void ide_start_request(struct buf *b)
/* send a request (and perhaps data, in the case of a write) to the disk, in and out x86 instructions are called to read and write device registers */
{
    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)
    }
}
```

IO interface

```c
void ide_rw(struct buf *b)
  // queues a request (if there are others pending)
 // or issues it directly to the disk (via ide_start_request()
{
   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);
}
```

Handle interrupts

```
void ide_intr()
/* invoked when an interrupt takes place; it reads data from the device (if the request is a read, not a write), wakes the process waiting for the I/O to complete, and (if there are more requests in the I/O queue), launches the next I/O via ide start request() */
{
    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);
}
```

总结一些I/O系统要完成的工作!

write(buf, 10);

OS需要提供系统调用接口

DMA.addr = buf;

DMA.count = 10;

……

sleep_on(Disk);

查一下手册就可以找到该写什么命令?该向哪里写?

让出CPU?

需要写中断处理程序!

```
do_write_end()//中断处理
{
    wakeup(Disk);
}
```

- 总的感觉: 很简单

  处理流程是很简单，复杂的是一些细节问题，如滚屏

I/O设备管理总结

- 如何实现交互? ⇒ 首先需要了解I/O的工作原理
- 从用户如何I/O开始 ⇒ 用户发送一个命令(read)
- 系统调用read ⇒ 被展开成给一些寄存器发送命令的代码
- 发送完命令以后... ⇒ CPU轮询，CPU干其它事情并等中断
- 中断方案最常见 ⇒ 相比其他设备，CPU太快了
- 实现独享设备的共享 ⇒ 假脱机系统（SPOOLING）

Module 6: I/O与存储

1. I/O devices
2. **Hard Disk Drives**
3. **RAID**
4. 其他I/O设备

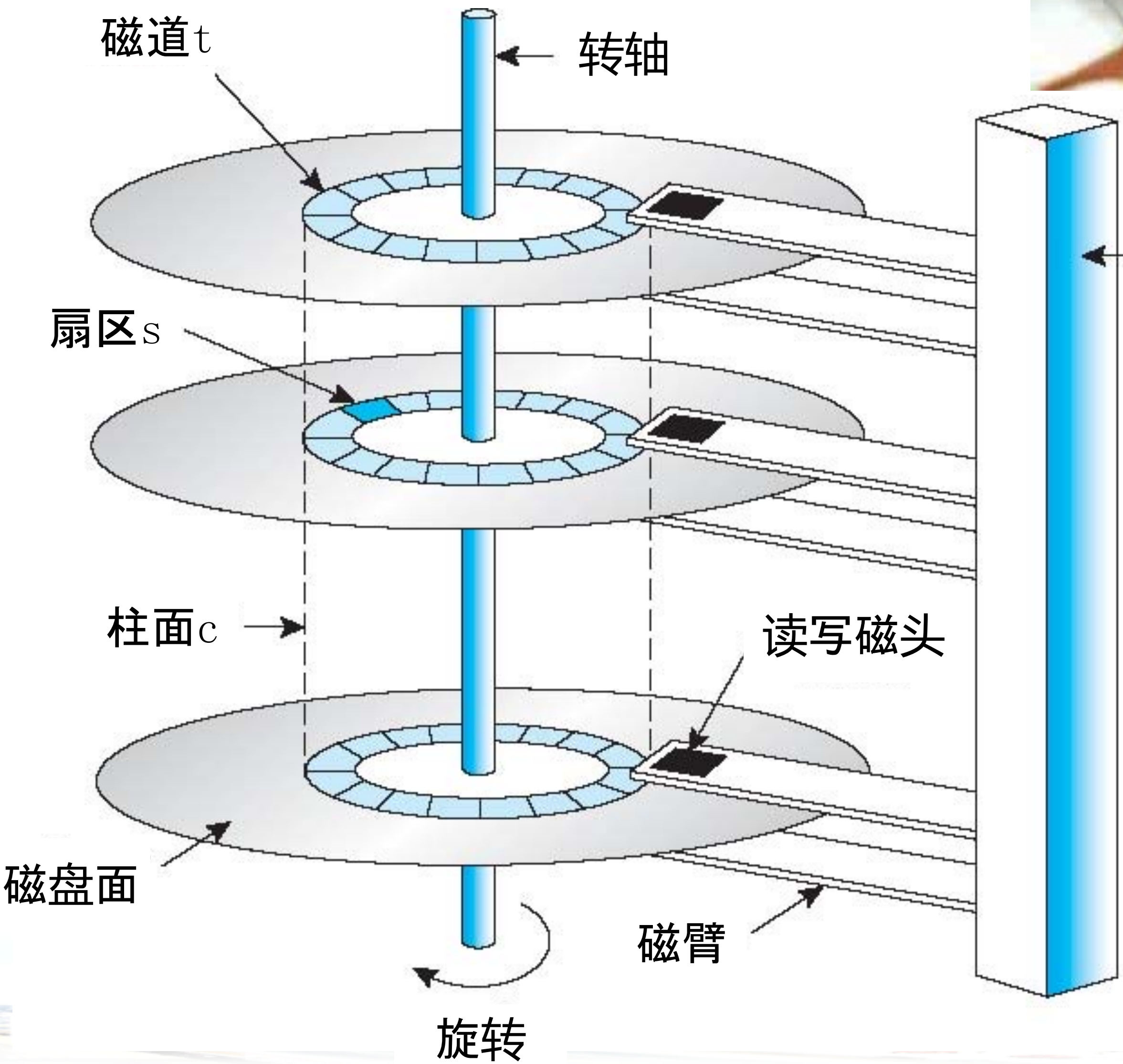
Tape is Dead
Disk is Tape
Flash is Disk
RAM Locality is King

Jim Gray

Microsoft

认识一下磁盘

认识一下磁盘

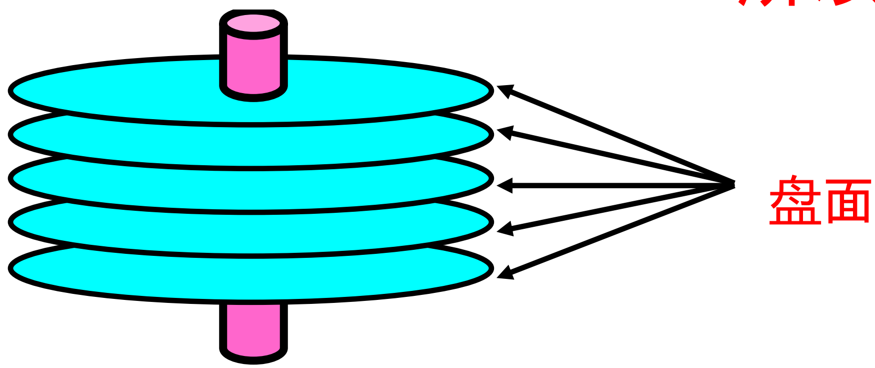


盘片高速旋转产生气流非常强，足以使磁头托起，并与盘面保持一个微小的距离。

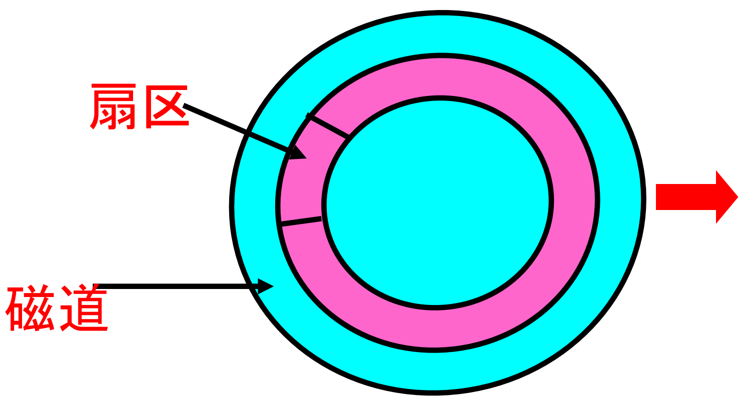
现在的水平已经达到0.005μm~0.01μm，这只是人类头发直径的千分之一。

认识一下磁盘

- 画一个示意图：



- 看看俯视图：



- 所以，磁盘被称为块设备！

扇区是磁盘的寻址单位、访问单位

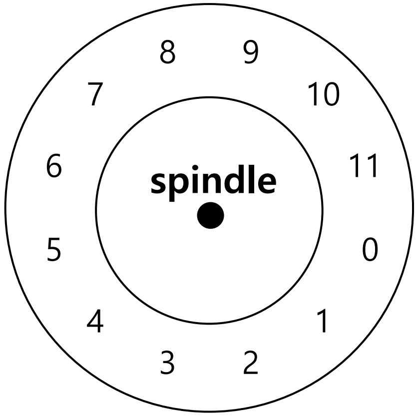
- 磁盘的数据单位是扇区
- 扇区大小：512字节
- 扇区的大小是传输时间和碎片浪费的折衷

Interface

- The only guarantee is that a single 512-byte write is atomic.
- Multi-sector operations are possible.
  - Many file systems will read or write 4KB at a time.
  - **Torn write**:
    - If an untimely power loss occurs, only a portion of a larger write may complete.
- Accessing blocks in **a contiguous chunk** is the fastest access mode.
  - A sequential read or write
  - Much faster than any more random access pattern.

39

Basic Geometry



A Disk with Just A Single Track (12 sectors)

- **Platter** (Aluminum coated with a thin magnetic layer)
  - A circular hard surface
  - Data is stored persistently by inducing magnetic changes to it.
  - Each platter has 2 sides, each of which is called a **surface**.

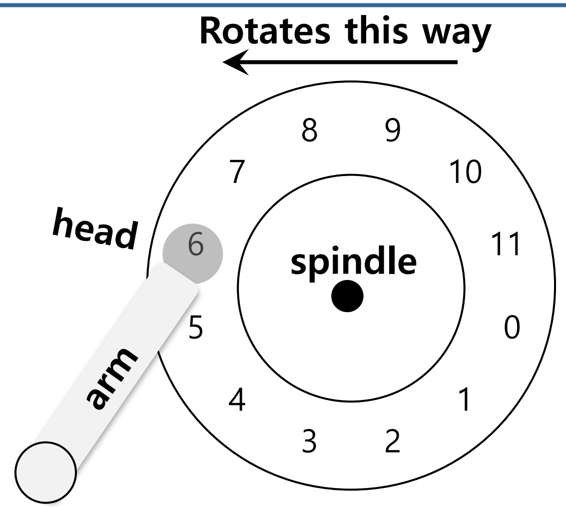
Basic Geometry (Cont.)

- **Spindle**
  - Spindle is connected to a motor that spins the platters around.
  - The rate of rotations is measured in **RPM** (Rotations Per Minute).
    - Typical modern values : 7,200 RPM to 15,000 RPM.
    - E.g., 10000 RPM : A single rotation takes about 6 ms.

- **Track**
  - Concentric circles of sectors
  - Data is encoded on each surface in a track.
  - A single surface contains many thousands and thousands of tracks.

41

A Simple Disk Drive

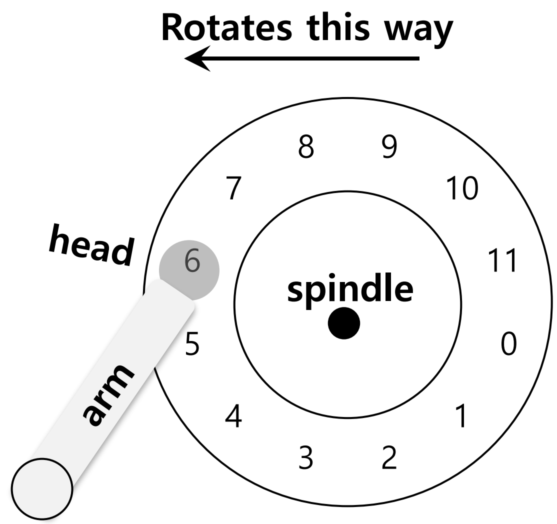


A Single Track Plus A Head

- **Disk head** (One head per surface of the drive)
  - The process of *reading* and *writing* is accomplished by the **disk head**.
  - Attached to a single disk arm, which moves across the surface.

42

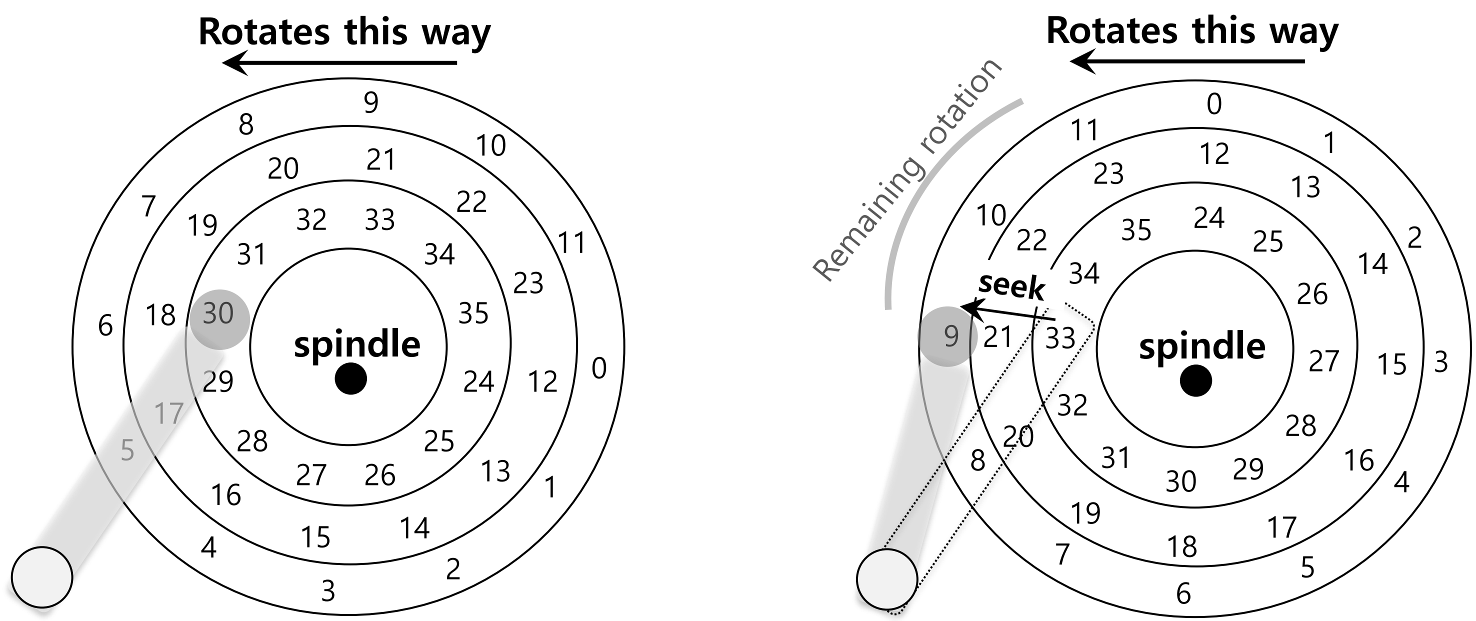
Single-track Latency: The Rotational Delay



A Single Track Plus A Head

- **Rotational delay:** Time for the desired sector to rotate
  - Ex) Full rotational delay is `R` and we start at `sector 6`
    - Read sector 0: Rotational delay = $\frac{R}{2}$
    - Read sector 5: Rotational delay = `R-1` (worst case)

Multiple Tracks: Seek Time



Three Tracks Plus A Head (Right: With Seek)
(e.g., read to sector 11)

- **Seek**: Move the disk arm to the correct track
  - Seek time: Time to move head to the track contain the desired sector.
  - One of the most costly disk operations.

Phases of Seek

- Acceleration → Coasting → Deceleration → Settling
  - **Acceleration**: The disk arm gets moving.
  - **Coasting**: The arm is moving at full speed.
  - **Deceleration**: The arm slows down.
  - **Settling**: The head is *carefully positioned* over the correct track.
    - The settling time is often quite significant, e.g., 0.5 to 2ms.

Transfer

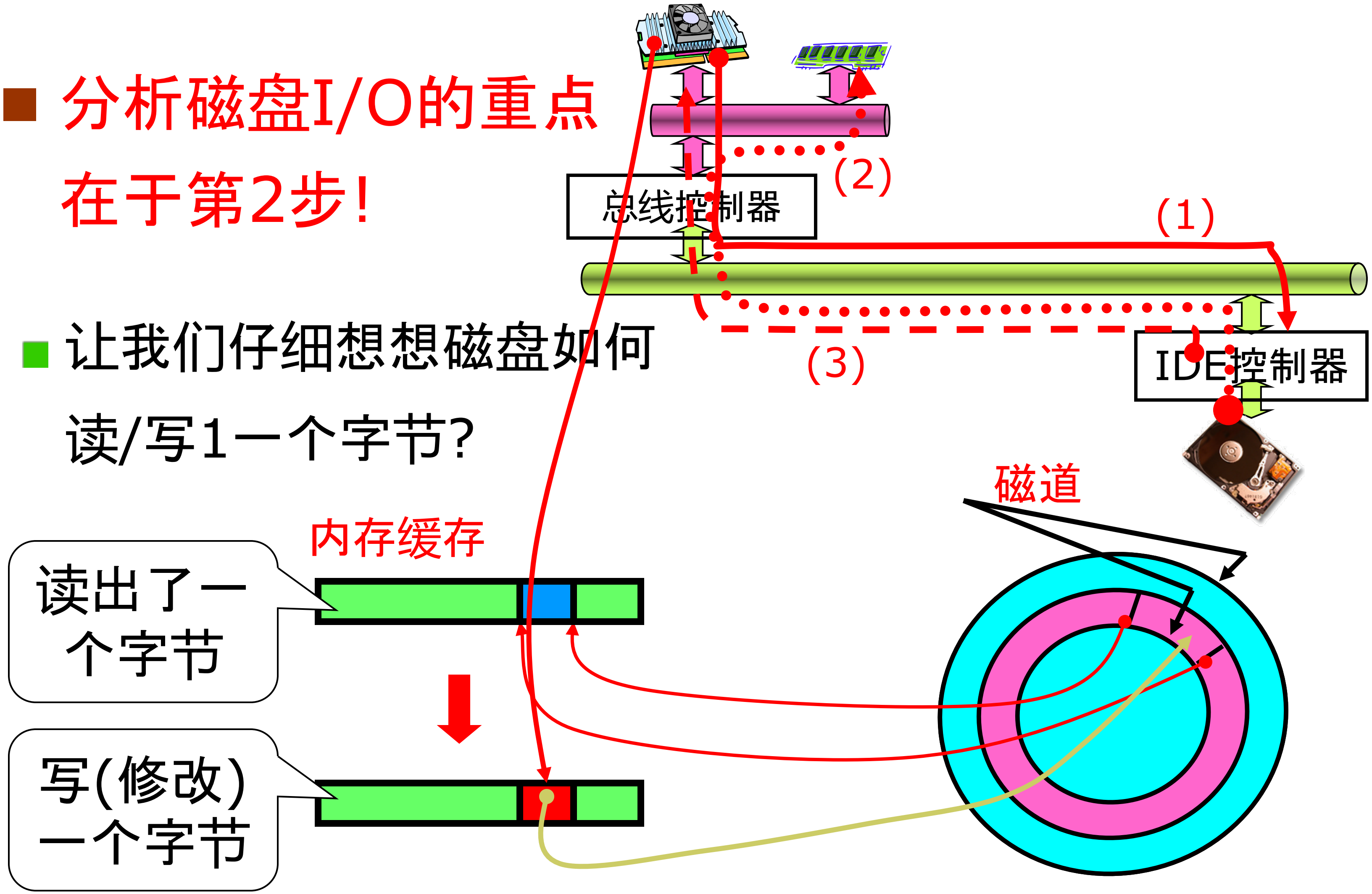
- The final phase of I/O
  - Data is either *read from* or *written* to the surface.

- Complete I/O time:
  - **Seek**
  - Waiting for the **rotational delay**
  - **Transfer**

46

磁盘的I/O

- 分析磁盘I/O的重点在于第2步!

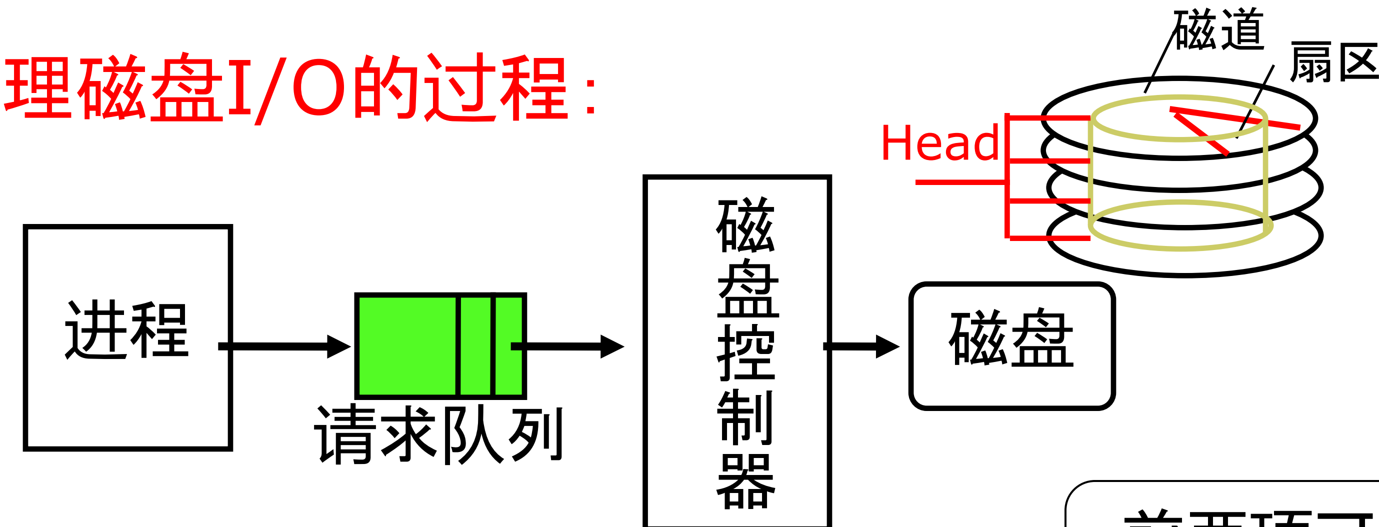


- 让我们仔细想想磁盘如何读/写1一个字节?

- 磁盘I/O：缓存队列 → 控制器 →寻道 →旋转 → 传输!

磁盘I/O的分析

- 整理磁盘I/O的过程：



- 我们最关心的磁盘什么时候读/写完？

前两项可以忽略！

磁盘访问延迟 = 队列时间 + 控制器时间 + 寻道时间 + 旋转时间 + 传输时间

3ms to 12 ms

(半周)：4 ms to 8 ms

约 0.25ms

- 关键所在：最小化寻道时间和旋转延迟！

I/O过程是解开许多磁盘问题的钥匙

- 磁盘调度：

前两项可以忽略！

磁盘访问延迟 = 队列时间 + 控制器时间 +
寻道时间 + 旋转时间 + 传输时间

12 ms to 8 ms

8 ms to 4 ms

约0.25ms

- 多个磁盘访问请求出现在请求队列怎么办？ 调度
- 调度的目标是什么？调度时主要考察什么？

目标当然是平均访问延迟小！

寻道时间是主要矛盾！

- 磁盘调度：输入多个磁道请求，给出服务顺序！

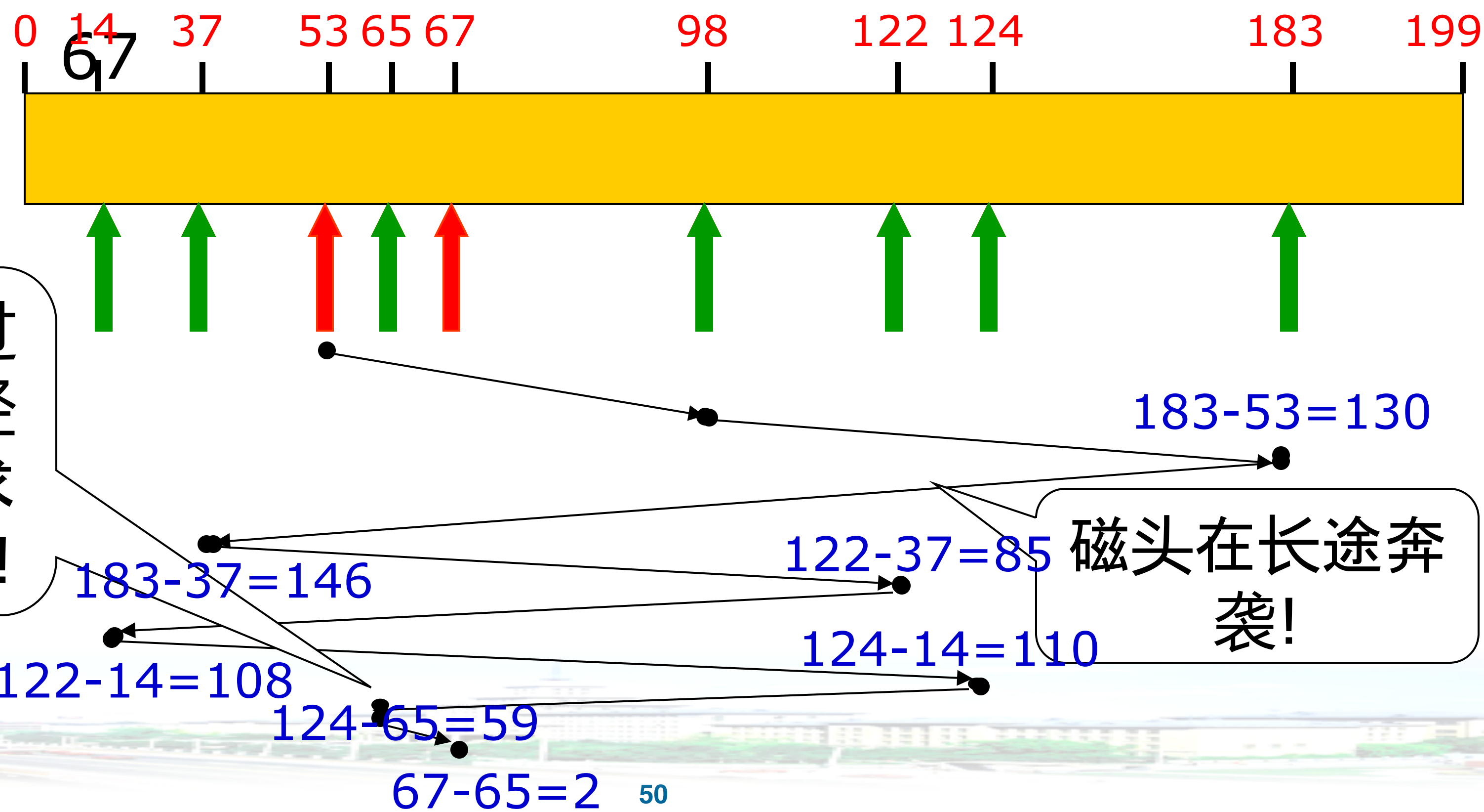
FCFS磁盘调度

130+146+85+108+110+59+2=640

- 最直观、最公平的调度：
  - 一个实例：磁头开始磁道位置=53，

    请求队列=98, 183, 37, 122, 14, 124, 65, 67

FCFS：磁头共移动640磁道！

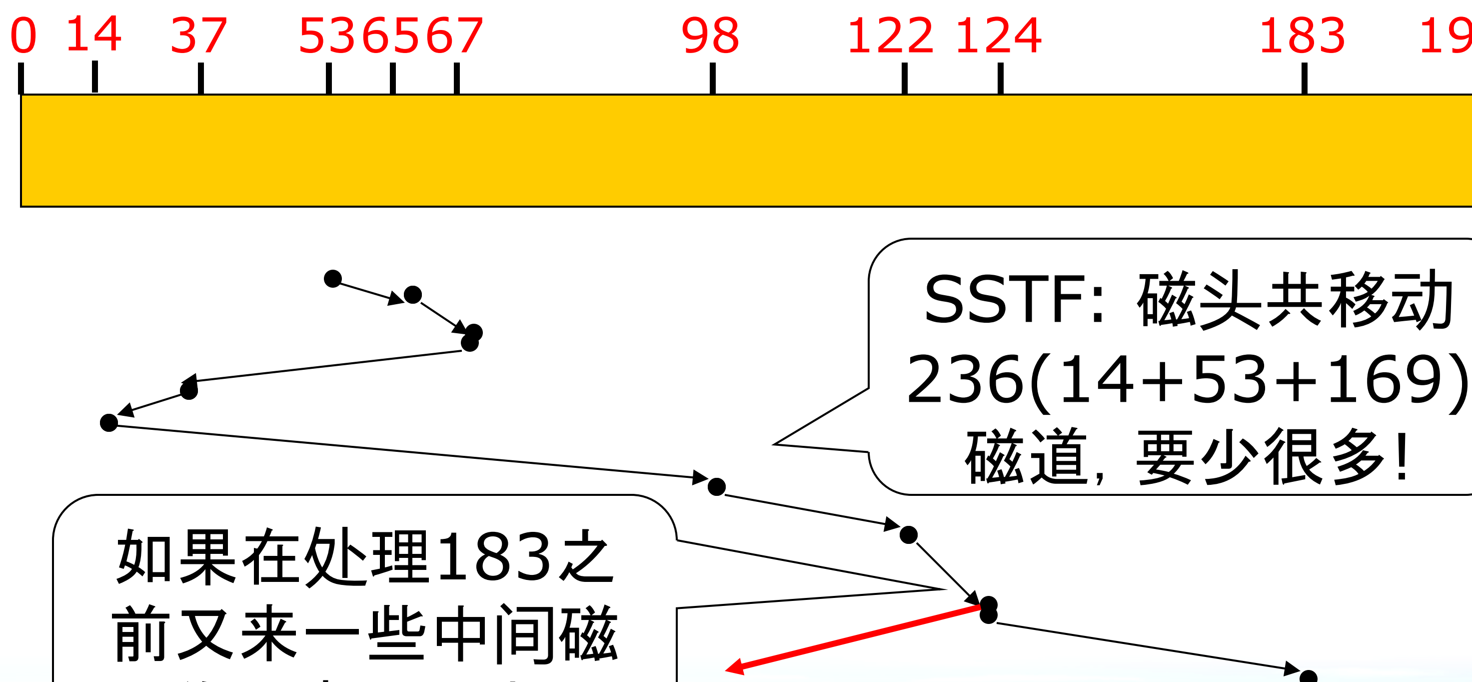


在移动过程中把经过的请求处理了?!

磁头在长途奔袭!

SSTF磁盘调度

- Shortest-seek-time First最短寻道时间优先：
  - 继续该实例：磁头开始位置=53；
    请求队列=98, 183, 37, 122, 14, 124, 65, 67



SSTF：磁头共移动236(14+53+169)磁道，要少很多！

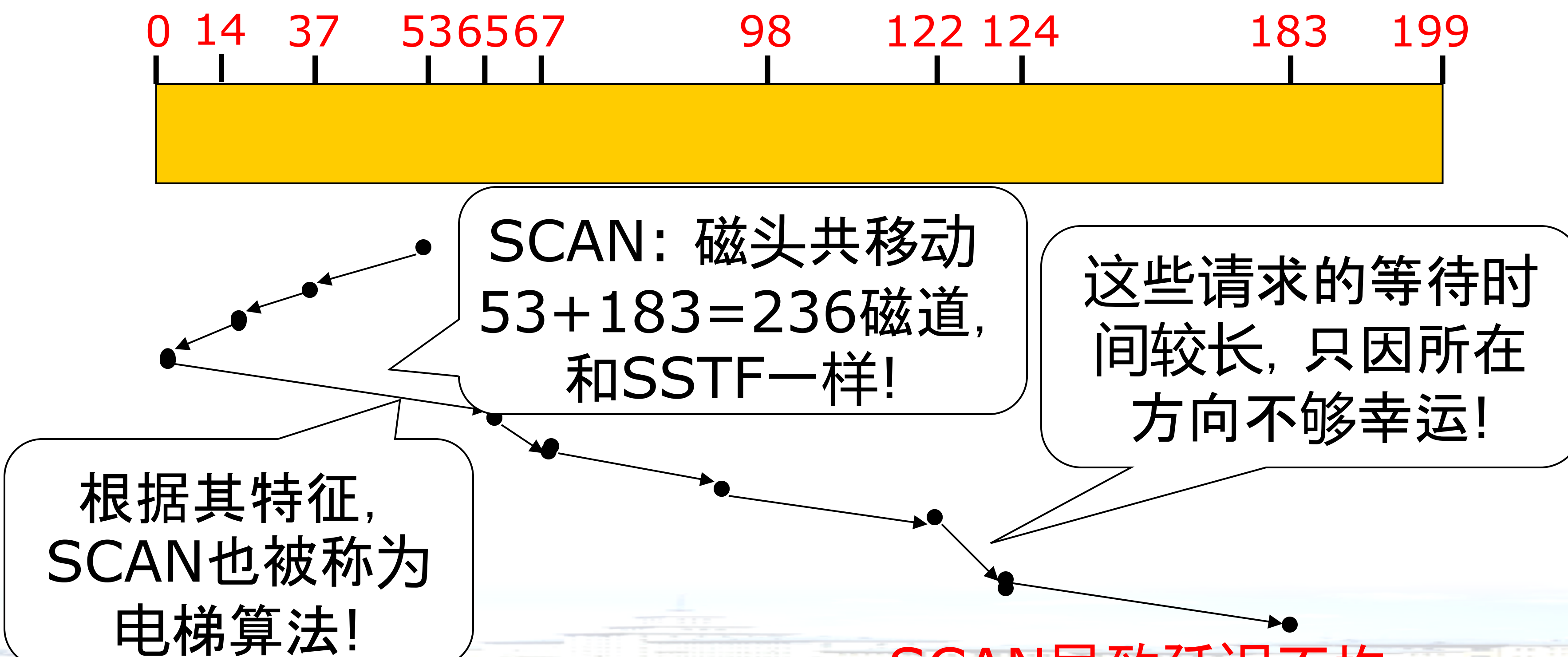
如果在处理183之前又来一些中间磁道的请求，则…

- SSTF存在饥饿问题

SCAN磁盘调度(扫描/电梯算法)

- SSTF+中途不回折:每个请求都有处理机会
  - 继续该实例：磁头开始位置=53;

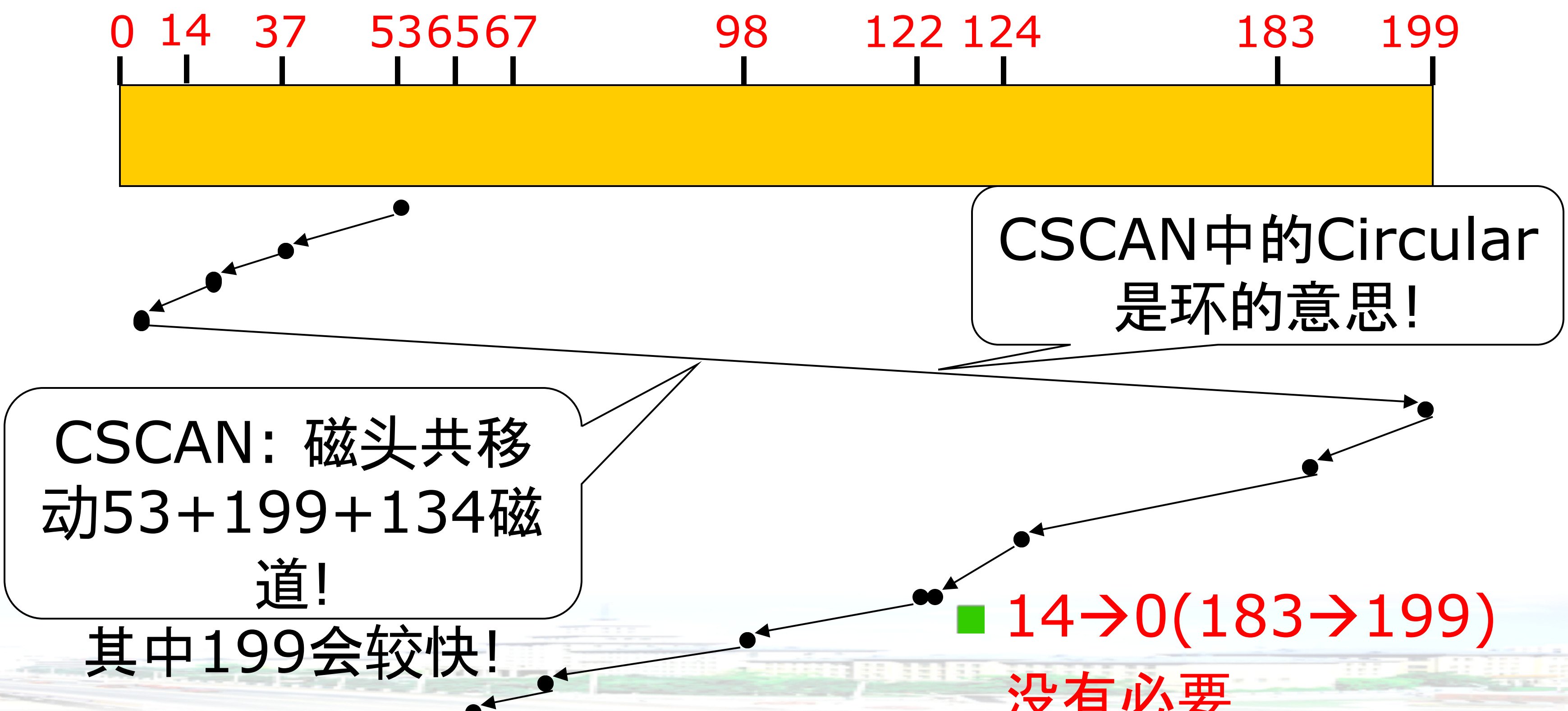
    请求队列=98, 183, 37, 122, 14, 124, 65, 67



- SCAN导致延迟不均

C-SCAN磁盘调度

- SCAN+直接移到另一端：两端请求都能很快处理
  - 继续该实例：磁头开始位置=53；
    请求队列=98, 183, 37, 122, 14, 124, 65, 67

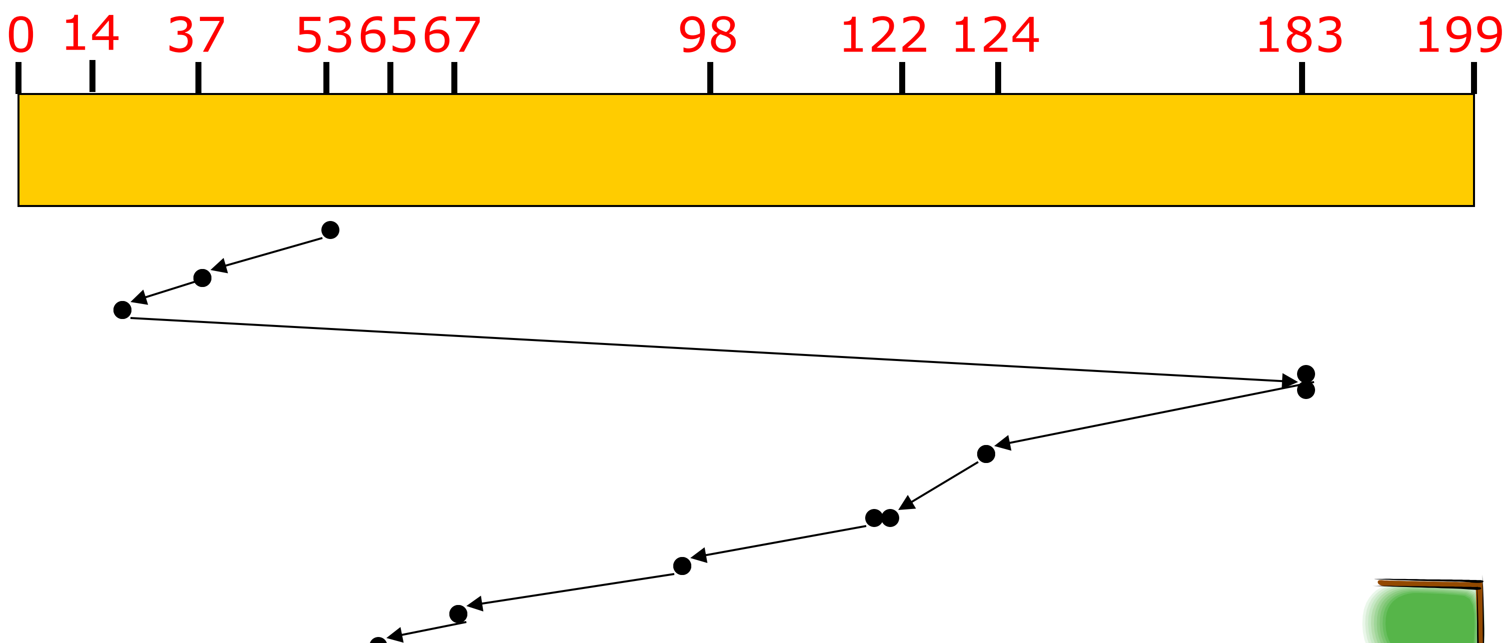


- 14→0(183→199)
  没有必要

C-LOOK磁盘调度

- CSCAN+看一看:前面没有请求就回移
  - 继续该实例：磁头开始位置=53；

    请求队列=98, 183, 37, 122, 14, 124, 65, 67



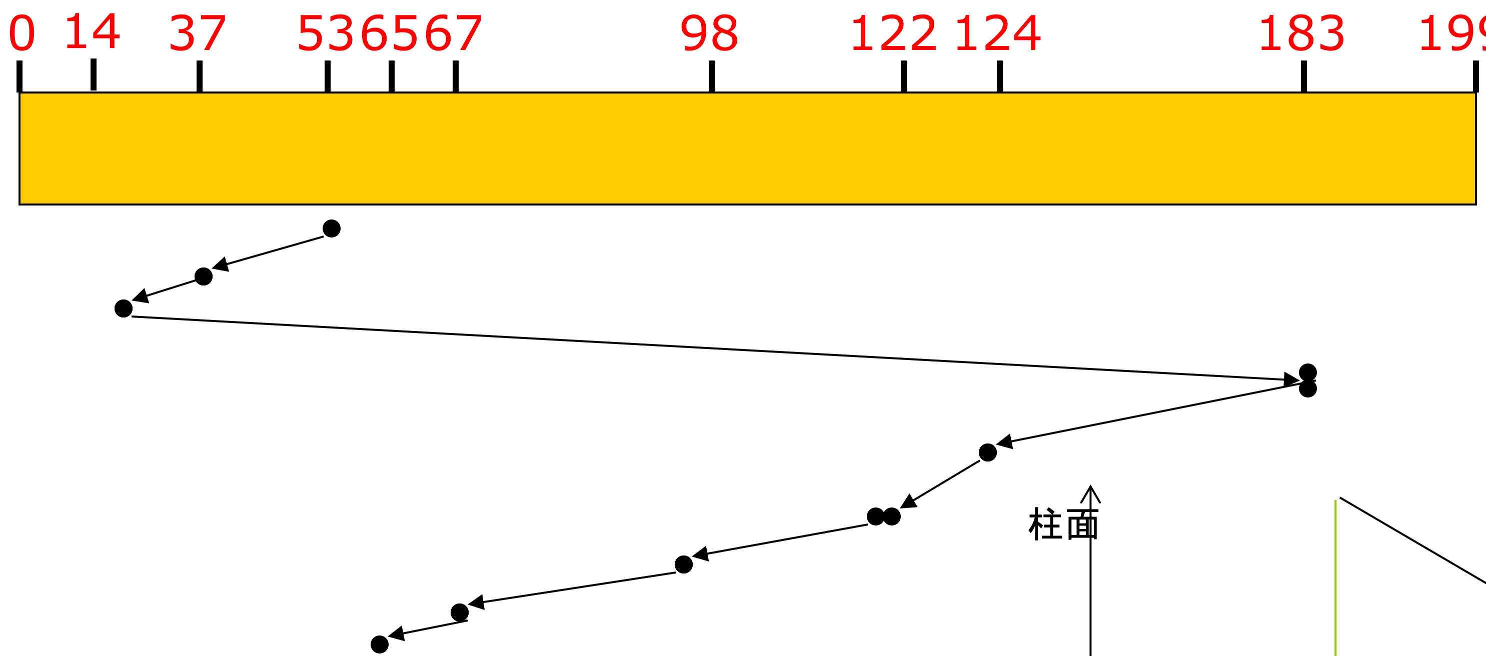
  - LOOK和C-LOOK是比较合理的缺省算法

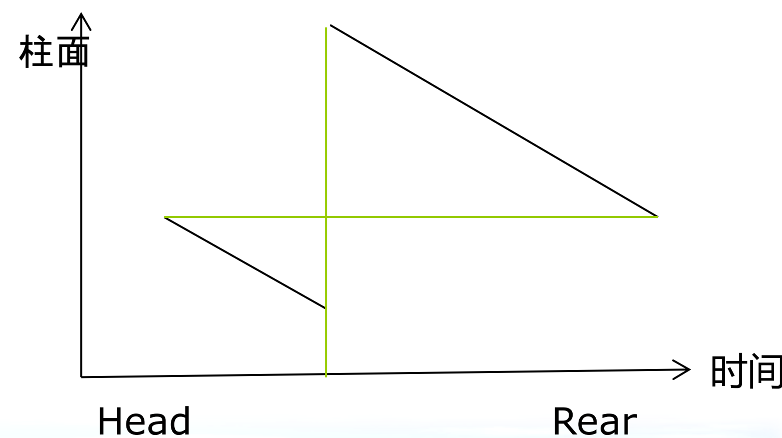
操作系统中所有的算法都要因地制宜!

C-LOOK磁盘调度

- 继续该实例：磁头开始位置=53；

  请求队列=98, 183, 37, 122, 14, 124, 65, 67





1）磁道请求队列的的形式

2）新磁道请求如何入队列

C[i+1]<X<c[i]或者
X>C[i+1]>c[i]

53-37-14- 183-124-122-98-67-65

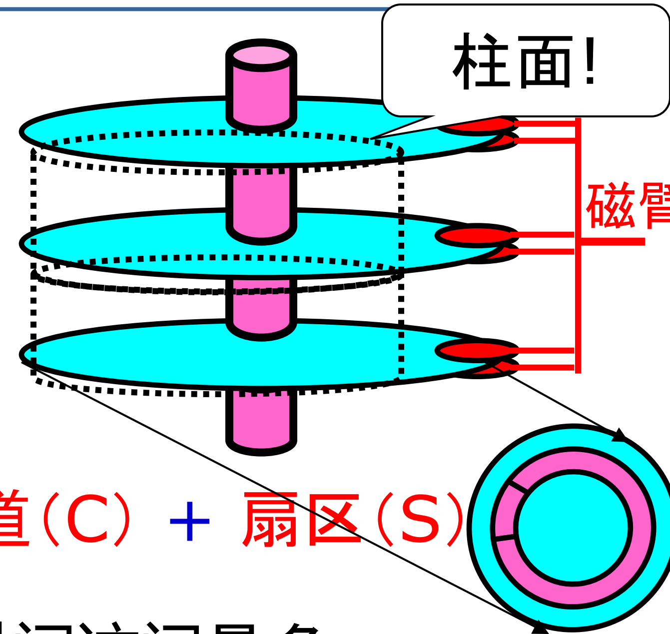
磁盘编址

如何管理磁盘，
首先对磁盘的扇区进行编号！

- 出厂的磁盘需要低级格式化(物理格式化)：
  将连续的磁性记录材料分成物理扇区
- 扇区 = 头 + 数据区 + 尾
- 头、尾中包含只有磁盘控制器能识别的扇区号码和纠 信息

什么是磁盘的逻辑格式化？
第12章 文件系统！

I/O过程是解开许多磁盘问题的钥匙



- 磁盘寻址：对于内存，我们往往更关心存放内容的地址
  - 实际上就是扇区怎么编址？
  - 显然这个地址是(盘面(H) + 磁道(C) + 扇区(S)
  - 寻道和旋转费时多 ⇒ 花最少时间访问最多扇区的方案：磁臂不动、磁盘旋转一周，访问磁头遇到的所有扇区。

让这些扇区的编址邻近：
因为局部性！

- 扇区编址(1)：CHS(Cylinder/Head/Sector)
- 扇区编址(2)：扇区编号(Logical Block Addressing LBA)

扇区编号—现代磁盘的常见寻址方式

整个磁盘

柱面编号c(0≤c≤C-1)

柱面0	柱面1	…	柱面C-1

一个柱面

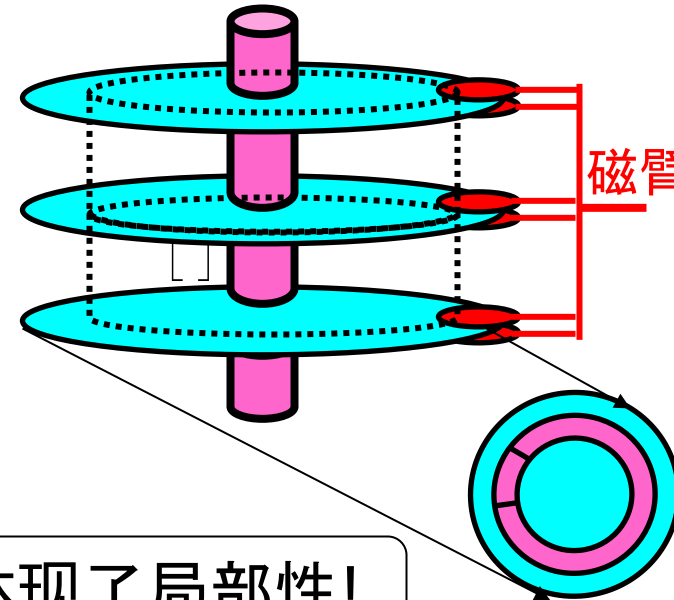
柱面内磁道(磁头)编号h(0≤h≤H-1)

磁道0	磁道1	…	磁道H-1

一个磁道

磁道内扇区编号s(0≤s≤S-1)

扇区0	扇区1	…	扇区S-1



体现了局部性!

- **扇区编号**，按照(C,H,S)将扇区形成一维扇区数组，数组索引就是扇区编号

某扇区(c,h,s)编号A = c*H*S + h*S + s　扇区总数 = C*H*S

已知A，则 s = A%S; h = [A/S]%H; c = [A/(H*S)]

扇区编号—现代磁盘的常见寻址方式

- chs(Cylinder/Head/Sector)模式
- 以前, 硬盘的容量还非常小, 采用与软盘类似的结构生产硬盘.
- 也就是**硬盘盘片的每一条磁道都具有相同的扇区数**
- 由此产生了所谓的3D参数 (Disk Geometry).:
- 磁柱面数(Cylinders),头数(Heads), 扇区数(Sectors per track),以及相应的寻址方式.

扇区编号—现代磁盘的常见寻址方式

- chs(Cylinder/Head/Sector)模式
- 磁头数(Heads) 表示硬盘总共有几个磁头,也就是有几面盘片, 最大为 256 (用 8 个二进制位存储);
- 柱面数(Cylinders) 表示硬盘每一面盘片上有几条磁道, 最大为 1024(用 10 个二进制位存储);
- 扇区数(Sectors per track) 表示每一条磁道上有几个扇区, 最大为 63 (用 6 个二进制位存储).
- 每个扇区一般是 512个字节;
- 所以磁盘最大容量为:
- 256 * 1024 * 63 * 512 / 1048576 = 8064 MB

扇区编号—现代磁盘的常见寻址方式

- chs(Cylinder/Head/Sector)模式
- 这种方式会浪费很多磁盘空间 (与软盘一样)
- 为了进一步提高硬盘容量, 产生了等密度结构硬盘，外圈磁道的扇区比内圈磁道多.
- 采用这种结构后, 硬盘不再具有实际的3D参数, 寻址方式也改为线性寻址, 即以扇区为单位进行寻址
- 为了与使用chs寻址的兼容 (如使用BIOS Int13H接口的软件), 在硬盘控制器内部安装了一个地址翻译器, 由它负责将老式3D参数翻译成新的线性参数.

IDE硬盘控制器的寄存器

- 有一组命令寄存器组(Task File Registers)，I/O的端口地址为1F0H~1F7H
- 1F2H 扇区计数寄存器
  1F3H 扇区号，或LBA块地址0~7
  1F4H 柱面数低8位，或LBA块地址8~15
  1F5H 柱面数高8位，或LBA块地址16~23
  1F6H 驱动器/磁头，或LBA块地址24~27
  1F7H 状态寄存器 命令寄存器
- CHS或LBA在磁头寄存器中指定

想一想……磁盘驱动应如何实现？


```
do_hd = intr_addr;                        // do_hd 函数会在中断程序中被调用。
outb_p(hd_info[drive].ctl,HD_CMD);        // 向控制寄存器输出控制字节。
port=HD_DATA;                             // 置 dx 为数据寄存器端口(0x1f0)。
outb_p(hd_info[drive].wpcom>>2, ++port);  // 参数：写预补偿柱面号(需除 4)。
outb_p(nsect, ++port);                    // 参数：读/写扇区总数。
outb_p(sect, ++port);                     // 参数：起始扇区。
outb_p(cyl, ++port);                      // 参数：柱面号低 8 位。
outb_p(cyl>>8, ++port);                   // 参数：柱面号高 8 位。
outb_p(0xA0|(drive<<4)|head, ++port);     // 参数：驱动器号+磁头号。
outb(cmd, ++port);                        // 命令：硬盘控制命令。
```

Linux 0.11下实现磁盘读写驱动片段

磁盘速度与内存速度的差异

1)磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定扇区长度的数据放入内存。

2)这样做的理论依据是计算机科学中著名的局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。

Module 6: I/O与存储

1. I/O devices
2. Hard Disk Drives
3. **RAID**
4. 其他I/O设备

RAID (Redundant Array of Inexpensive Disks)

- **Use multiple disks** in concert to build a faster, bigger, and more reliable disk system.
  - RAID just looks like a big disk to the host system.

- Advantage
  - **Performance** & **Capacity**: Using multiple disks in parallel
  - **Reliability**: RAID can tolerate the loss of a disk.

RAIDs provide these advantages transparently to systems that use them.

RAID Interface

- When a RAID receives I/O request,
  1. The RAID **calculates** which disk to access.
  2. The RAID **issue** one or more **physical I/Os** to do so.

- RAID example: A mirrored RAID system
  - Keep two copies of each block (each one on a separate disk)
  - Perform two physical I/Os for every one logical I/O it is issued.

80

RAID Internals

- A microcontroller
  - Run firmware to direct the operation of the RAID
- Volatile memory (such as DRAM)
  - Buffer data blocks
- Non-volatile memory
  - Buffer writes safely
- Specialized logic to perform parity calculation

Fault Model

- RAIDs are designed to **detect** and **recover** from certain kinds of disk faults.
- Fail-stop fault model
  - A disk can be in one of two states: *Working* or *Failed*.
    - Working: all blocks can be read or written.
    - Failed: the disk is permanently lost.
  - RAID controller can immediately observe when a disk has failed.

82

How to evaluate a RAID

- **Capacity**
  - How much useful capacity is available to systems?
- **Reliability**
  - How many disk faults can the given design tolerate?
- **Performance**

83

RAID Level 0: Striping

- RAID Level 0 is the simplest form as **striping** blocks.
  - **Spread the blocks** across the disks in a round-robin fashion.
  - No redundancy
  - Excellent performance and capacity

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Stripe
(The blocks in the same row)

RAID-0: Simple Striping
(Assume here a 4-disk array)

RAID Level 0 (Cont.)

- Example) RAID-0 with a bigger chunk size
  - Chunk size : 2 blocks (8 KB)
  - A Stripe: 4 chunks (32 KB)

Disk 0	Disk 1	Disk 2	Disk 3
0	2	4	6
1	3	5	7
8	10	12	14
9	11	13	15

chunk size: 2blocks

Striping with a Bigger Chunk Size

85

Chunk Sizes

- Chunk size mostly affects performance of the array
  - **Small chunk size**
    - Increasing the parallelism
    - Increasing positioning time to access blocks
  - **Big chunk size**
    - Reducing intra-file parallelism
    - Reducing positioning time

Determining the "best" chunk size is hard to do.

Most arrays use larger chunk sizes (e.g., 64 KB)

86

RAID Level 0 Analysis

N : the number of disks

- **Capacity** → RAID-0 is perfect.
  - Striping delivers N disks worth of useful capacity.
- **Performance** of striping → RAID-0 is excellent.
  - All disks are utilized often in parallel.
- **Reliability** → RAID-0 is bad.
  - Any disk failure will lead to data loss.

87

Evaluating RAID Performance

- Consider two performance metrics
  - Single request latency
  - Steady-state throughput

- Workload
  - **Sequential**: access 1MB of data (block (B) ~ block (B + 1MB))
  - **Random**: access 4KB at random logical address

- A disk can transfer data at
  - `S` MB/s under a sequential workload
  - `R` MB/s under a random workload

88

Evaluating RAID Performance Example

- sequential (S) vs random (R)
  - **Sequential** : transfer 10 MB on average as continuous data.
  - **Random** : transfer 10 KB on average.
  - Average seek time: 7 ms
  - Average rotational delay: 3 ms
  - Transfer rate of disk: 50 MB/s

- Results:
  - S = $\frac{Amount\ of\ Data}{Time\ to\ access} = \frac{10\ MB}{210\ ms}$ = 47.62 MB /s
  - R = $\frac{Amount\ of\ Data}{Time\ to\ access} = \frac{10\ KB}{10.195\ ms}$ = 0.981 MB /s

89

Evaluating RAID-0 Performance

N : the number of disks

- Single request latency
  - Identical to that of a single disk.

- Steady-state throughput
  - **Sequential** workload : $N \cdot S$ MB/s
  - **Random** workload : $N \cdot S$ MB /s

RAID Level 1 : Mirroring

- RAID Level 1 tolerates **disk failures**.
  - **Copy** more than one of **each block** in the system.
  - Copy block places on a separate disk.

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Simple RAID-1: Mirroring (Keep two physical copies)

- RAID-10 (RAID 1+0) : mirrored pairs and then stripe
- RAID-01 (RAID 0+1) : contain two large striping arrays, and then mirrors

91

RAID-1 Analysis

N : the number of disks

- **Capacity**: RAID-1 is Expensive
  - The useful capacity of RAID-1 is N/2.
- **Reliability**: RAID-1 does well.
  - It can tolerate the failure of any one disk (up to N/2 failures depending on which disk fail).

92

Performance of RAID-1

- Two physical writes to complete
  - It suffers the worst-case seek and rotational delay of the two request.
  - Steady-state throughput
    - **Sequential Write** : $\frac{N}{2} \cdot S$ MB/s
      - Each logical write must result in two physical writes.
    - **Sequential Read** : $\frac{N}{2} \cdot S$ MB/s
      - Each disk will only deliver half its peak bandwidth.
    - **Random Write** : $\frac{N}{2} \cdot R$ MB/s
      - Each logical write must turn into two physical writes.
    - **Random Read** : $N \cdot R$ MB/s
      - Distribute the reads across all the disks.

93

RAID Level 4 : Saving Space With Parity

- Add **a single parity block**
  - **A Parity block** stores the *redundant information* for that stripe of blocks.

* P: Parity

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	0	1	1	P0
2	2	3	3	P1
4	4	5	5	P2
6	6	7	7	P3

Five-disk RAID-4 system layout

RAID Level 4 (Cont.)

- **Compute parity** : the XOR of all of bits

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1)=0
0	1	0	0	XOR(0,1,0,0)=1

- **Recover from parity**
  - Imagine the bit of the C2 in the first row is lost.
    1. Reading the other values in that row : 0, 0, 1
    2. The parity bit is 0 → even number of 1's in the row
    3. What the missing data must be: a 1.

95

RAID-4 Analysis

N : the number of disks

- **Capacity**
  - The useful capacity is $(N-1)$.

- **Reliability**
  - RAID-4 tolerates 1 disk failure and no more.

96

RAID-4 Analysis (Cont.)

- **Performance**
  - Steady-state throughput
    - Sequential read: $(N-1) \cdot S$ MB/s
    - Sequential write: $(N-1) \cdot S$ MB/s

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Full-stripe Writes In RAID-4

    - Random read: $(N-1) \cdot R$ MB/s

Random write performance for RAID-4

- Overwrite a block + update the parity
- **Method 1**: *additive parity*
  - Read in all of the other data blocks in the stripe
  - XOR those blocks with the new block (1)
  - **Problem**: the performance <u>scales with</u> the number of disks

Random write performance for RAID-4 (Cont.)

- **Method 2**: *subtractive parity*

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1)=0

$$P(new) = \left(C2(old)\ XOR\ C2(new)\right)\ XOR\ \ P(old)$$

- Update C2(old) → C2(new)
  1. Read in the old data at C2 (C2(old)=1) and the old parity (P(old)=0)
  2. Calculate P(new):
     - If C2(new)==C2(old) → P(new)==P(old)
     - If C2(new)!=C2(old) → Flip the old parity bit

Small-write problem

- The parity disk can be a **bottleneck.**
  - Example: update blocks 4 and 13 (marked with *)

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

Writes To 4, 13 And Respective Parity Blocks.

- Disk 0 and Disk 1 can be accessed in parallel.
- Disk 4 <u>prevents any parallelism</u>.

RAID-4 throughput under random small writes is $(\frac{R}{2})$ MB/s (*terrible*).

A I/O latency in RAID-4

- **A single read**
  - Equivalent to the latency of a single disk request.

- **A single write**
  - Two reads and then two writes
    - Data block + Parity block
    - The reads and writes can happen <u>in parallel.</u>
  - Total latency *is about twice* that of a single disk.

RAID Level 5: Rotating Parity

- RAID-5 **is solution of** small write problem.
  - Rotate the parity blocks across drives.
  - Remove the parity-disk bottleneck for RAID-4

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

RAID-5 With Rotated Parity

RAID-5 Analysis

N : the number of disks

- **Capacity**
  - The useful capacity for a RAID group is $(N - 1)$.

- **Reliability**
  - RAID-5 tolerates 1 disk failure and no more.

103

RAID-5 Analysis (Cont.)

N : the number of disks

- **Performance**
  - Sequential read and write
  - A single read and write request

  Same as RAID-4

  - Random read : a little better than RAID-4
    - RAID-5 can utilize all of the disks.
  - Random write : $\frac{N}{4} \cdot R$ MB/s
    - The factor of four loss is cost of using parity-based RAID.

104

RAID Comparison: A Summary

N : the number of disks
D : the time that a request to a single disk take

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	N	N/1	N-1	N-1
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	N・S	(N/2) ・S	(N-1) ・S	(N-1) ・S
Sequential Write	N・S	(N/2) ・S	(N-1) ・S	(N-1) ・S
Random Read	N・R	N・R	(N-1) ・R	N・R
Random Write	N・R	(N/2) ・R	$\frac{1}{2}$R	$\frac{N}{4}$ R
Latency				
Read	D	D	D	D
Write	D	D	2D	2D

RAID Capacity, Reliability, and Performance

105

RAID Comparison: A Summary

- **Performance** and do not care about reliability → RAID-0 (Striping)
- **Random I/O** performance and **Reliability** → RAID-1 (Mirroring)
- **Capacity** and **Reliability** → RAID-5
- **Sequential I/O** and Maximize **Capacity** → RAID-5

106

谢 谢！