



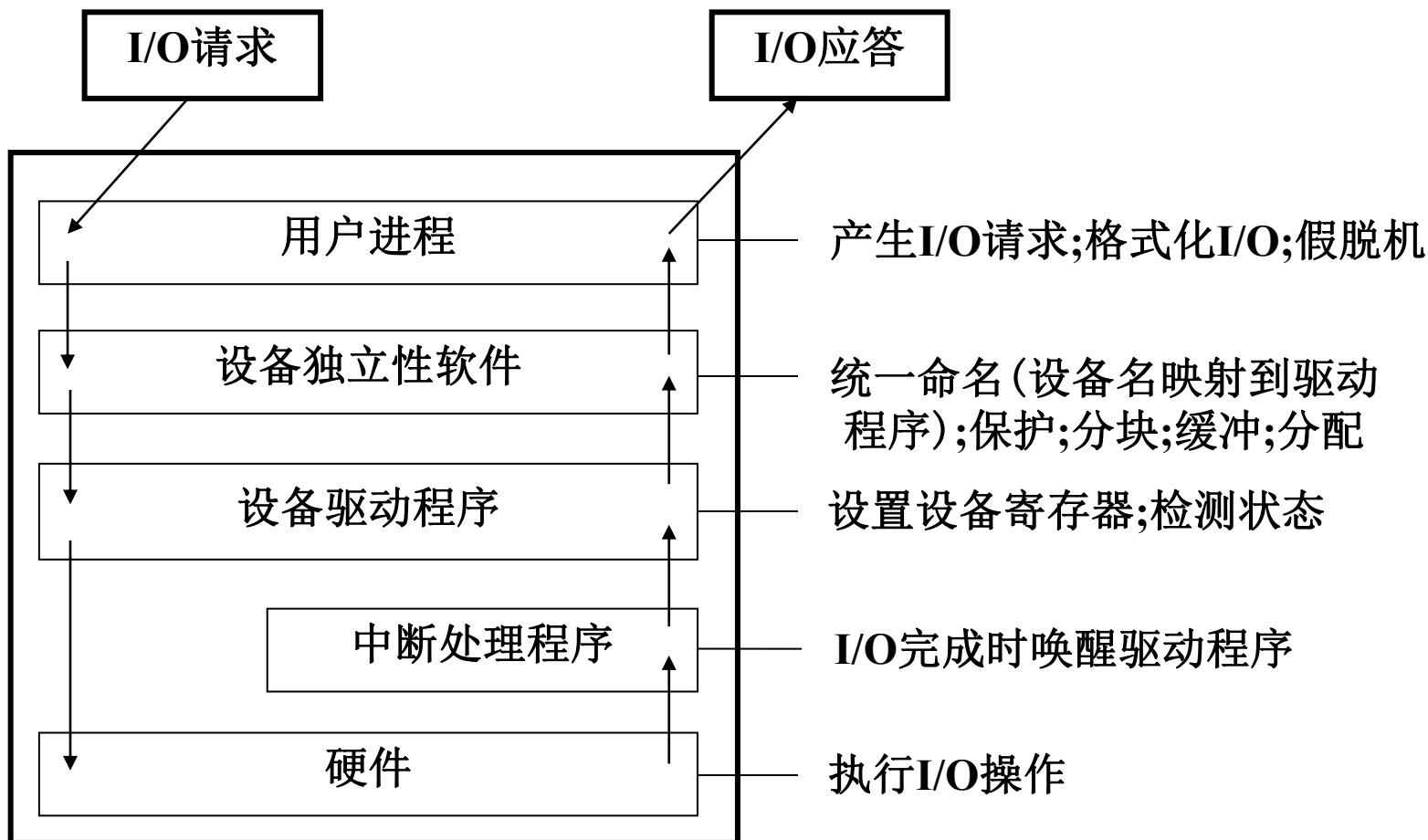
第六章 输入输出系统

- 6.1 I/O系统的层次及每层的功能
- 6.2 I/O设备和设备控制器
- 6.3 中断机构和中断处理程序
- 6.4 设备驱动程序
- 6.5 与设备无关的I/O软件
- 6.6 用户层的I/O软件
- 6.7 缓冲区管理
- 6.8 磁盘存储器的性能和调度

韩明峰 QQ 2022446359



6.1 I/O系统的层次及每层的功能





6.2 I/O设备和设备控制器

6.2.1 I/O设备

(1) 字符设备。以字符为单位发送或接收一个字符流，不可寻址。如网络接口和鼠标。

(2) 块设备。信息存储在固定大小的块中，每个块有自己的地址。如磁盘和USB盘。

(3) 其他设备。如时钟既不是块可寻址的，也不产生或接收字节流；内存映射的显示器也不适用字符设备和块设备的模型。



6.2.2 设备控制器

1. 设备控制器的基本功能

- (1) 接收和识别命令。
- (2) 与CPU和设备的数据交换。
- (3) 标识和报告设备的状态。
- (4) 地址识别:包括设备地址和控制器中寄存器的地址。
- (5) 数据缓冲区。
- (6) 差错控制。

2. 设备控制器的组成

(1) 设备控制器与处理机的接口。(2) 设备控制器与设备的接口。(3) I/O逻辑。

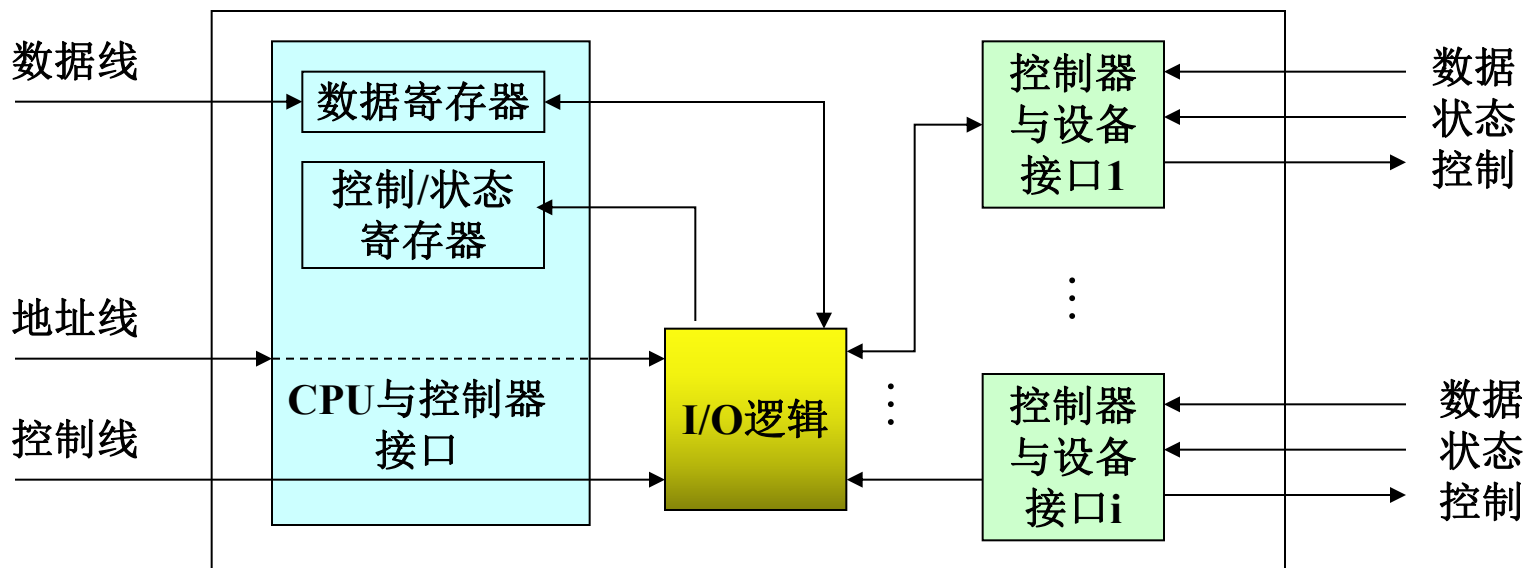


图6-4 设备控制器的组成

6.2.3 CPU如何访问设备控制器的寄存器或数据缓冲区

1. 利用特定的I/O指令

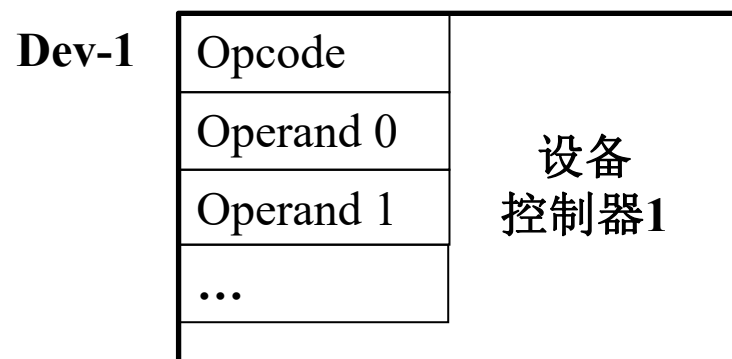
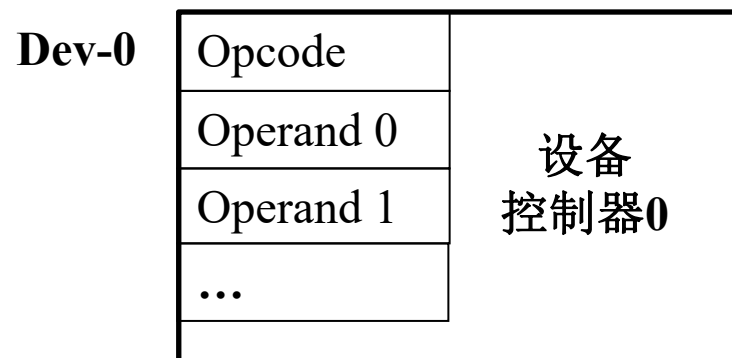
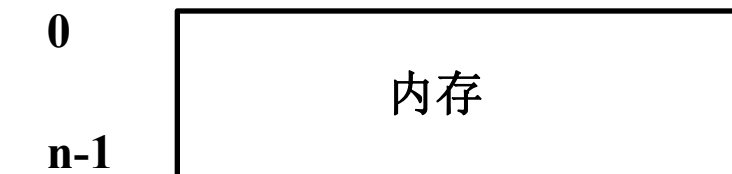
访问设备控制器: `io-store cpu-reg, dev-no, dev-reg`

访问内存: `store cpu-reg, k`

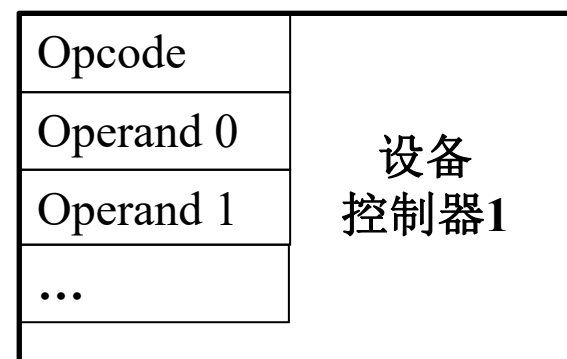
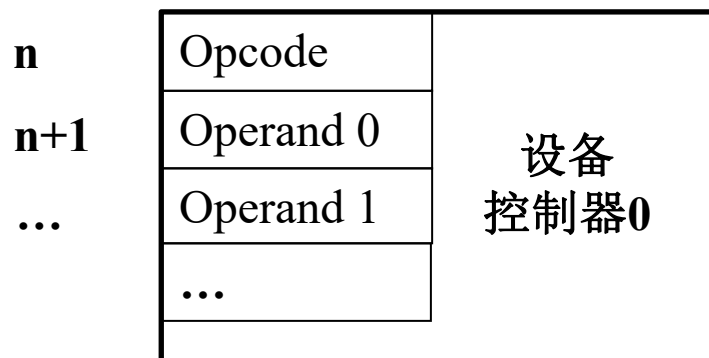
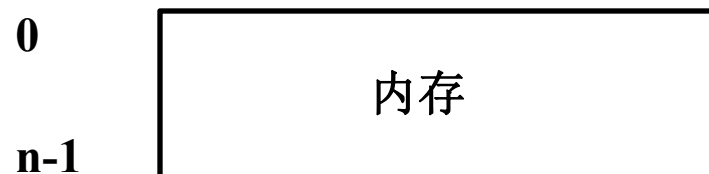
2. 内存映射I/O

访问设备控制器和访问内存使用一种指令:

`store cpu-reg, n`



...



...

图6-5 设备寻址形式

6.2.4 对I/O设备的控制方式

1. 程序直接控制I/O方式

以输入数据为例：

(1) CPU代表一个进程向控制器发出I/O指令，启动输入设备。

(2) 状态寄存器忙/闲标志busy置1。

(3) 循环测试busy，busy=1输入未完成。

(4) busy=0输入完成，CPU将数据从控制器的数据寄存器读到内存。

缺点：忙等待。

2. 中断驱动的I/O方式

(1) CPU代表一个进程向控制器发出I/O指令后，进程进入阻塞态（同步I/O），操作系统调度其他进程运行。

(2) 设备控制器控制I/O设备传输数据，CPU与I/O设备并行工作。

(3) 一旦数据输入寄存器，控制器向CPU发出中断。

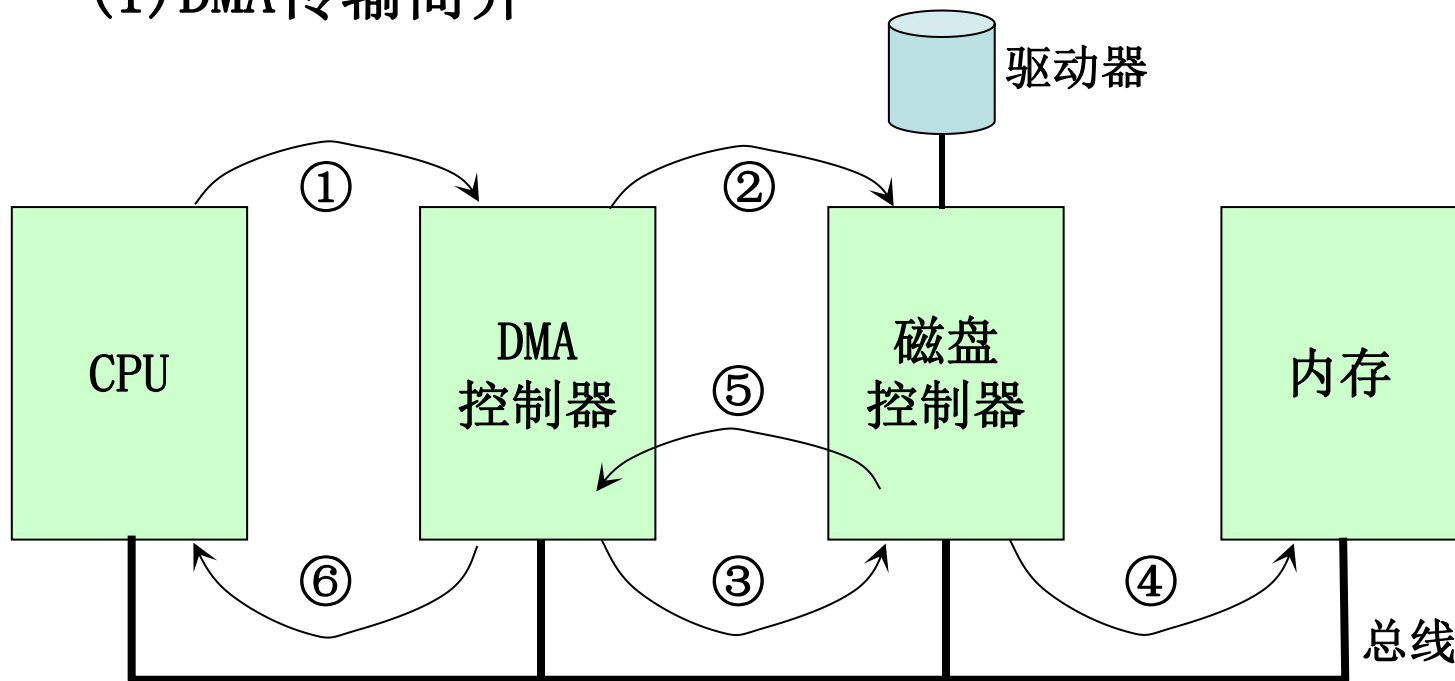
(4) CPU运行中断服务程序，检查数据正确性，把数据写入内存。

(5) 中断服务程序唤醒阻塞进程，执行操作系统的调度功能，运行选中的进程。

缺点：数据搬运工作仍然是由CPU来做的。

3. 直接存储器访问方式 (DMA)

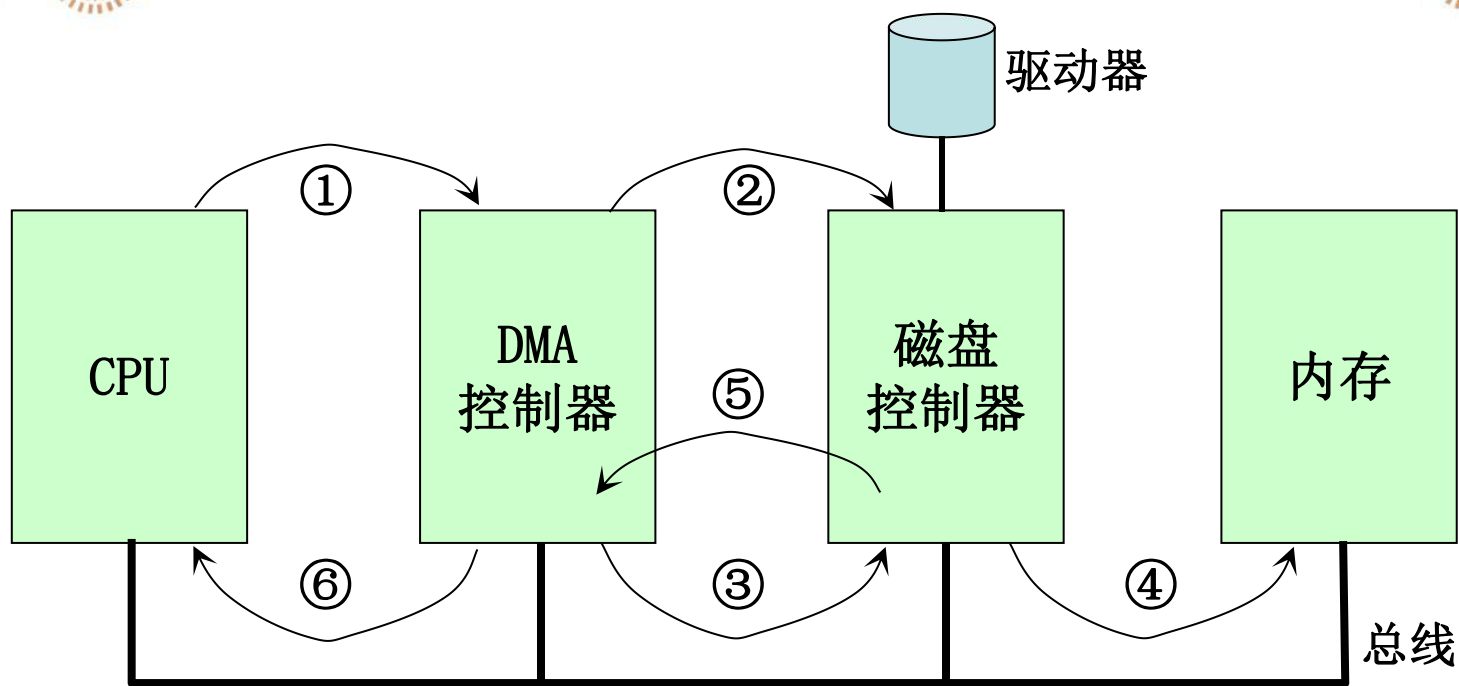
(1) DMA传输简介



DMA传输操作

① 主机将DMA命令块（源地址、目的地址、字节数、读写命令）写到内存，将命令块的地址写到DMA控制器。

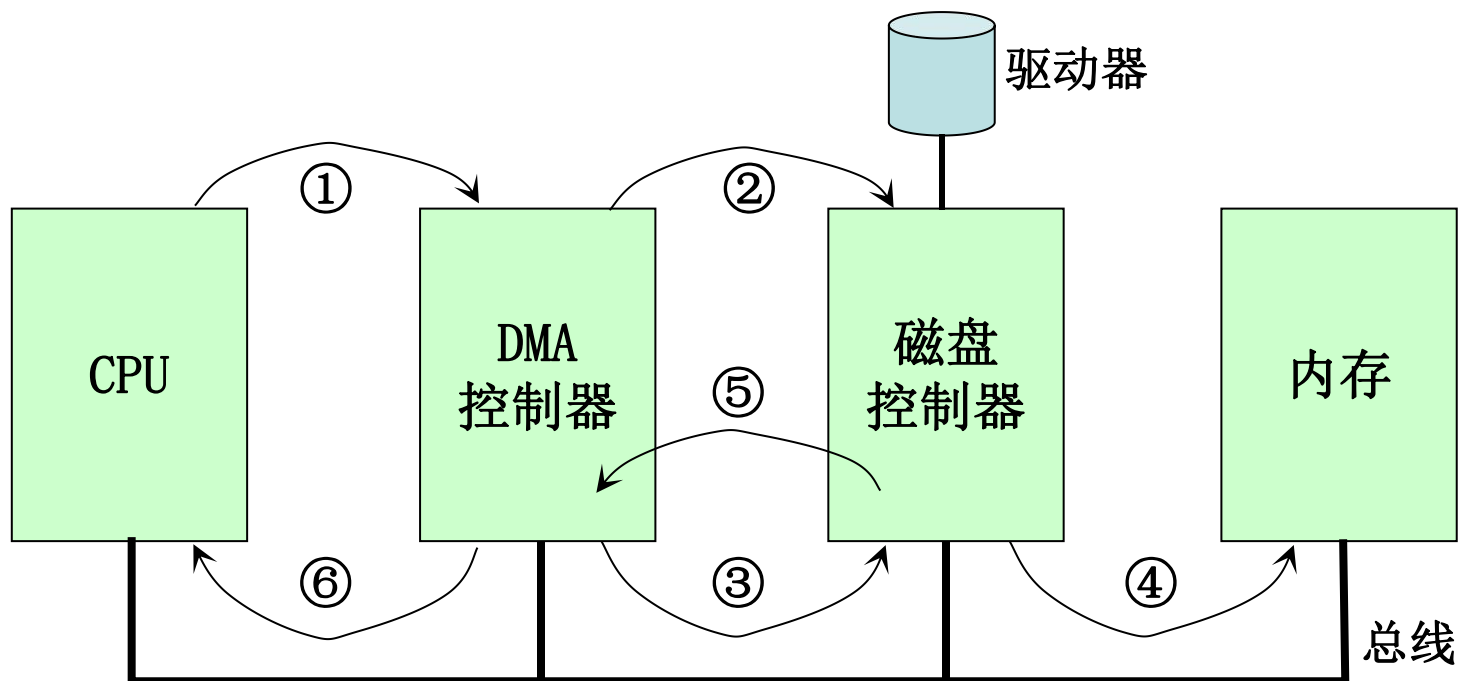
② DMA控制器设置并启动设备控制器工作。



DMA传输操作


- ③ DMA请求传送到内存。
- ④ 数据传送。
- ⑤ 应答。

DMA步增内存地址，步减字节计数。如果字节计数仍然大于0，重复③、④和⑤。



DMA传输操作

⑥ 传输结束，DMA控制器向CPU发出中断信号。此时数据已经在内存中了。



(2) 直接存储器访问方式 (DMA) 的注意事项

- 1) 有些设备控制器内置DMA控制器。
- 2) DMA控制器使用总线的机制包括周期窃取和突发模式。
- 3) DMA控制器可实现多路传送，每路控制一个设备控制器。其调度策略可基于轮转或优先级。
- 4) 链式DMA控制器一次可以传输多个离散的数据块。



4. I/O通道控制方式

(1) I/O通道

- 1) I/O通道是一种特殊的处理机。
- 2) I/O通道指令类型单一。
- 3) I/O通道没有自己的内存，与CPU共享内存。
- 4) CPU向通道**发送一条I/O指令**，给出通道程序首址和要访问的I/O设备，通道通过执行通道程序完成I/O任务。



(2) I/O通道类型

1) 字节**多路**通道。以字节为单位，子通道按时间片轮转方式共享主通道。

2) 数组**选择**通道。按数组方式进行数据传送，在一段时间内只能执行一道通道程序，很像一个**单道程序**的处理器。



3) 数组**多路**通道。按数组方式进行数据传送，很像一个**多道程序**的处理器。



(3) 通道程序

通道程序是由一系列通道指令所构成的。每条通道指令包含的信息：

- 1) 操作码
- 2) 内存地址
- 3) 计数
- 4) 记录结束标志R (R=0表示与下一条指令处理的数据属于同一记录; R=1表示某记录的最后一条指令)
- 5) 通道程序结束位P (P=1表示程序结束)



操作	P	R	计数	内存地址
WRITE	0	0	80	813
WRITE	0	0	140	1034
WRITE	0	1	60	5830
WRITE	0	1	300	2000
WRITE	0	0	250	1850
WRITE	1	1	250	720

由六条通道指令所构成的简单的通道程序

5. I/O处理机控制方式

相比于 I/O通道，I/O处理机有自己的内存，I/O程序放在本地存储器运行，事实上其本身就是一台计算机。

6.3 中断机构和中断处理程序

6.3.1 中断简介

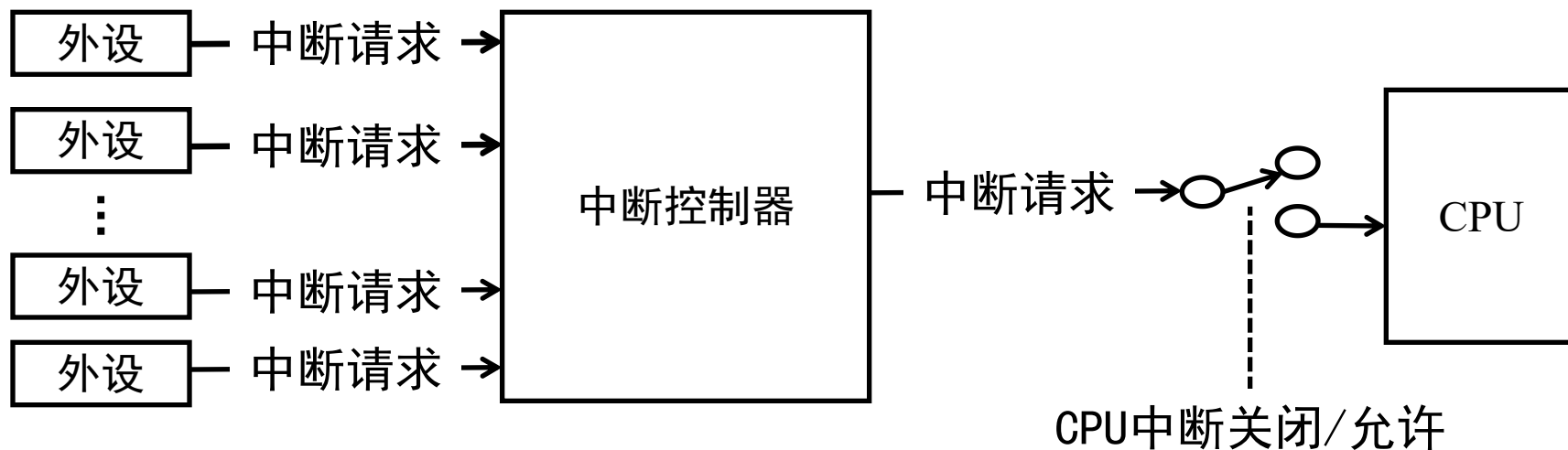
1. 中断

(1) **广义中断**：导致程序正常执行流程发生改变的事件，不包括程序分支指令和函数调用。



广义中断包括**中断**、**异常**和**自陷**。

(2) **中断**是由于CPU外部的原因而改变程序的执行流程，又称为**硬件中断**。

中断属于**异步事件**，可以屏蔽。



中断系统图



(3) 异常通常与正在执行的指令有关，如被0除、执行非法指令和内存保护故障等。

异常是同步事件，不可屏蔽。

(4) 自陷表示通过处理器所拥有的软件指令，可预期地使处理器的执行流程发生变化，以执行特定的程序，也叫软件中断。如：

Motorola 68000系列中的Trap指令；

ARM中的SWI指令；

Intel 80x86中的INT指令。

自陷是同步事件，不可屏蔽。

(5) 自陷可用于实现内核态的系统调用。如Linux在

Intel 80x86上利用INT 0x80软件中断来实现系统调用。



2. 中断向量表

中 断 向 量 号	中断处理程序
0	clockintr
1	diskintr
2	ttyintr
3	devintr
.....

中断向量表



3. 中断优先级

(1) 中断响应优先级：同时发生的多个中断应该优先响应哪一个。

(2) 中断处理优先级：新发生的高优先级中断可以打断正在处理的低优先级中断。

4. 对多中断源的处理方式

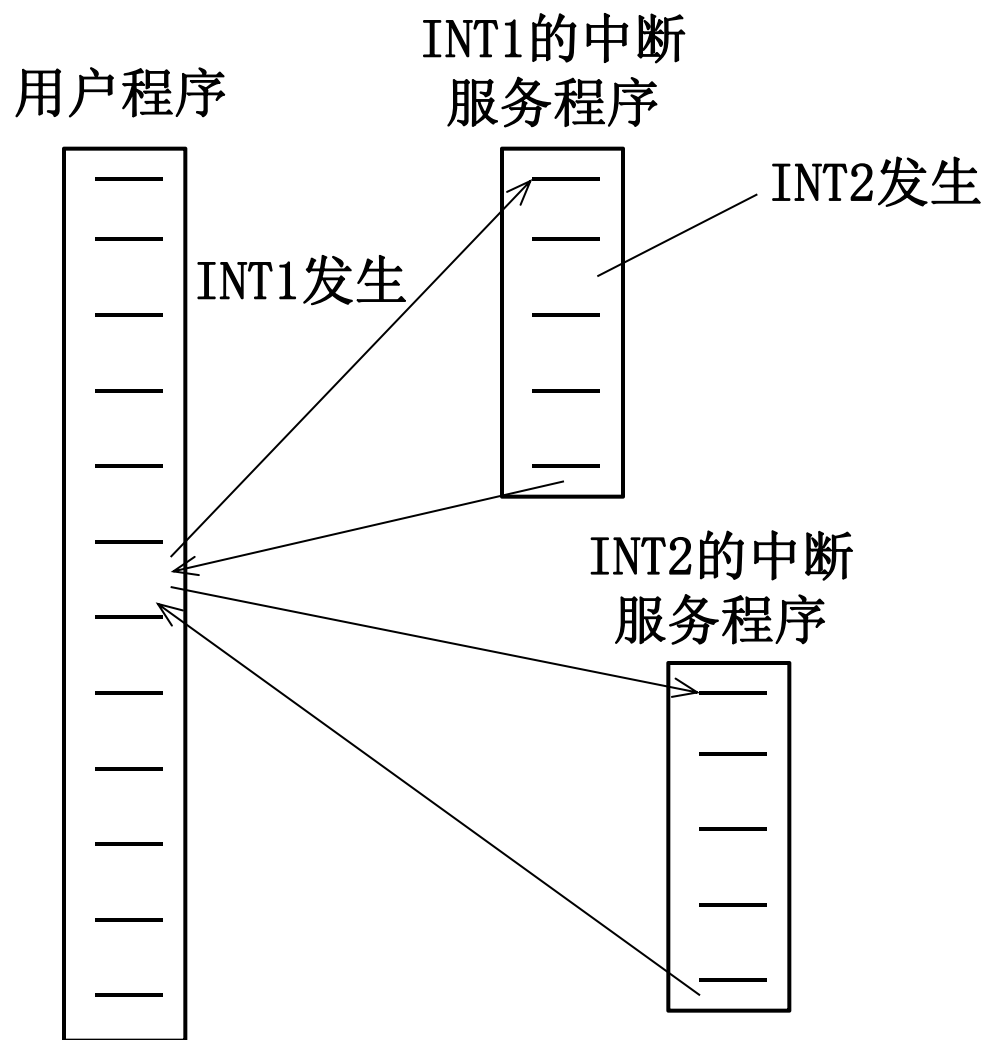
(1) 顺序中断处理

当一个中断正被处理期间，屏蔽其他的中断；

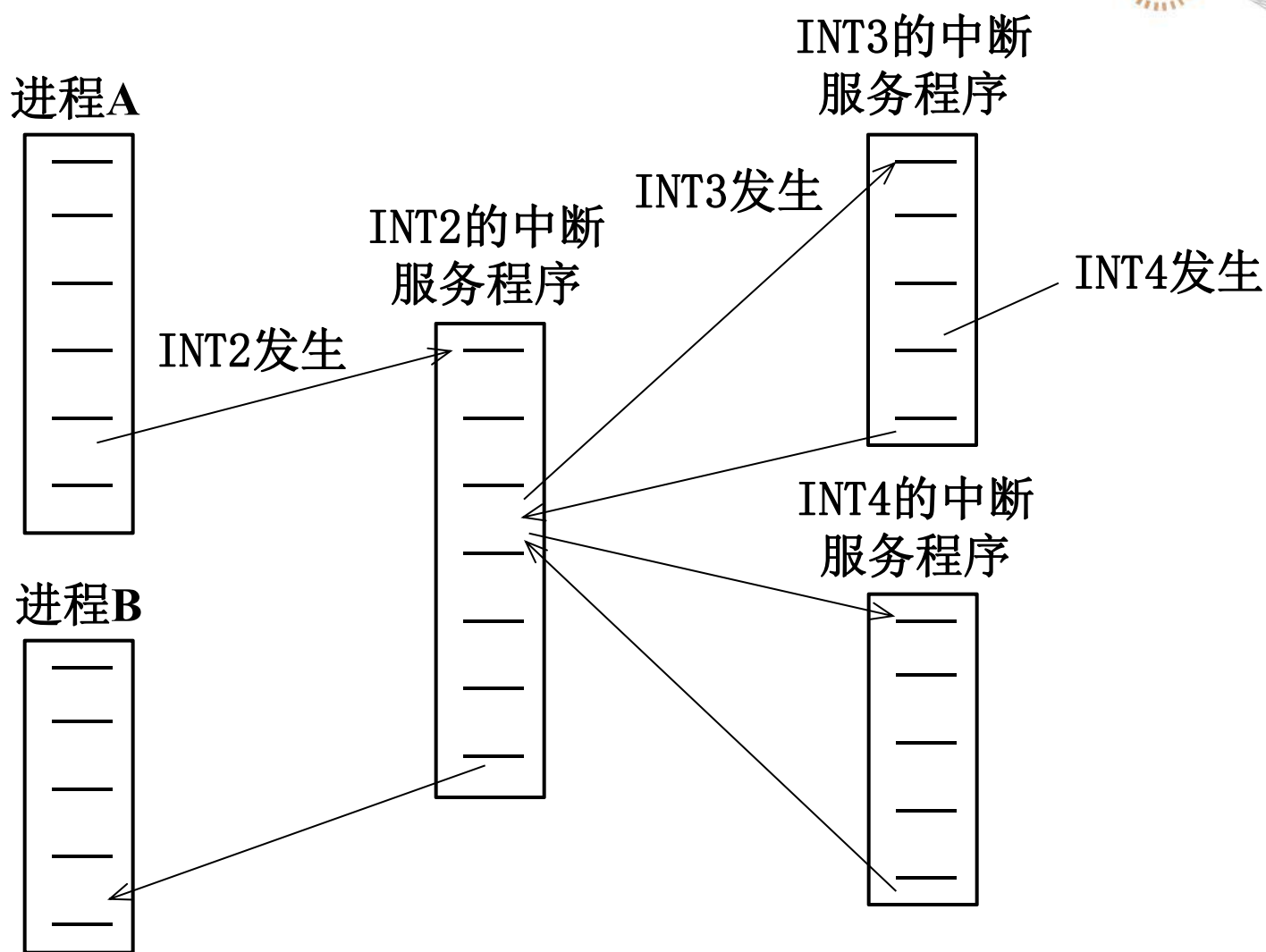
在该中断处理完后，开放中断。如有未处理的中断，则依次处理。

(2) 嵌套中断处理

对每类中断赋予不同的优先级，允许高优先级中断打断低优先级中断的处理程序。



多中断的顺序处理方式示意图



(中断优先级 $INT2 < INT4 \leq INT3$)

多中断的嵌套处理方式示意图



6.3.2 中断的处理过程（指外部硬件中断）

1. 中断检测

2. 中断响应，由硬件完成

(1) 中止当前程序的执行，保存原程序的断点信息PC和PSW；

(2) CPU从中断控制器得到中断向量号，以中断向量号作为索引号，查找中断向量表，转到相应的中断服务程序；

(3) 关中断。



3. 中断处理程序

(1) 继续保存现场

(2) 执行用户中断处理程序，处理完后解除相应进程的阻塞状态。

如果多个中断源共用一个中断向量号，则可以采用中断链技术，中断向量表中的中断处理程序入口地址指向中断处理程序队列的队首。

当发生中断时，一个接一个地调用相应队列中的处理程序，直至找到能为该请求提供服务的处理程序为止。



3. 中断处理程序

(3) 恢复现场

- 1) 如果发生了中断嵌套，返回上一级中断；
- 2) 可能返回到优先级更高的进程；
- 3) 也可能返回到原先被打断的进程。

恢复工作现场时，先恢复各通用寄存器，再恢复PSW与PC。

通常使用一条不可中断的特权指令来恢复PSW与PC。

4. 所有中断源共用一个中断服务程序

上述“中断处理程序”的(1)和(3)是一样的。

如果(2)使用相同的控制程序来调用不同的用户中断处理程序，那么中断向量表中各硬件中断向量号对应的中断处理程序入口地址是一样的，如Linux操作系统的中断向量表。




6.4 设备驱动程序

1 设备驱动程序的主要功能

(1) 接收与设备无关的软件发来的命令和参数，检查输入参数的有效性。

(2) 将命令中的抽象要求转换为与设备相关的低层操作序列。如将线性盘块号转换为磁盘的柱面号、磁道及扇区号。

(3) 如果设备空闲，便立即向设备控制器发出I/O命令，完成指定的I/O操作；如果设备忙碌，则将请求者的请求挂在设备队列上等待。



(4) 通常I/O操作需要一定时间，因此驱动程序发出I/O命令后把自己阻塞，I/O中断发生时被唤醒。

另有一些情况驱动程序不需要阻塞，如字符模式下滚动屏幕，只需写少许字节到控制器的寄存器即可。

(5) 操作完成后驱动程序检查错误，将数据传送给设备无关软件，向调用者返回错误报告。

(6) 如果还有其他未完成的请求在排队，则选择一个启动执行。否则驱动程序阻塞以等待下一个请求。



2 设备驱动程序的统一接口

每个设备驱动程序与OS的其余部分之间都有着相同的或者相近的接口，即设备驱动程序的函数指针表。



3 设备驱动程序的运行方式

(1) 为每一类设备设置一个进程，专门用于执行这类设备的I/O操作。

(2) 在整个系统中设置一个I/O进程，专门用于执行系统中所有各类设备的I/O操作。

也可以设置一个输入进程和一个输出进程，分别处理系统中所有各类设备的输入或输出操作。

(3) 不设置专门的设备处理进程，而只为各类设备设置相应的设备处理程序(模块)，供用户进程或系统进程调用。

6.5 与设备无关的I/O软件

1. 向用户层软件提供一个统一的接口。
2. 为设备统一命名，实现设备名与驱动程序的映射。

(1) 在UNIX系统中，一个设备名对应一个特殊的设备文件，如/dev/tty00唯一地确定了一个特殊文件的i节点，其中包含了主设备号和次设备号，主设备号用于定位相应的驱动程序，次设备号作为参数传给驱动程序。



(2) 设备访问举例：

```
fd=open( “/dev/sda” , O_RDWR) ;//申请U盘设备  
lseek(fd, 1024, 0) ;  
write(fd, buffer, 96) ;  
.....  
close(fd) ;
```

以上代码绕过了文件系统管理，直接读写U盘空间。

3. 对设备进行保护。

设备是作为命名对象出现在文件系统中的，对文件的常规保护也适用于I/O设备。



4. 缓冲管理

5. 差错控制

许多错误是设备特定的，必须由适当的驱动程序来处理，但错误处理的框架是设备无关的。

6. 设备的分配与回收

假设一个I/O操作是原子的。

(1) 独占设备：如果一个逻辑上完整的数据通过多次I/O操作才能完成，那么就需要独占使用设备，其方式类似于申请互斥信号量。

(2) 共享设备：一个逻辑上完整的数据通过一次I/O操作即可完成。此类设备总是能申请成功，但是需将I/O请求排入队列，由驱动程序来处理（可能涉及I/O调度）。



7. 独立于设备的逻辑数据块

设备独立性软件应能够隐藏数据交换单位大小的差异，而向高层软件提供统一的块大小。

例如，不同的磁盘扇区大小不同，可将若干个扇区当作一个逻辑块。

8. 文件系统管理模块也是设备无关的I/O层软件。

文件系统管理模块主要定位逻辑文件在物理辅存上的位置。



6.6 用户层的I/O软件

6.6.1 库函数

1. 系统调用

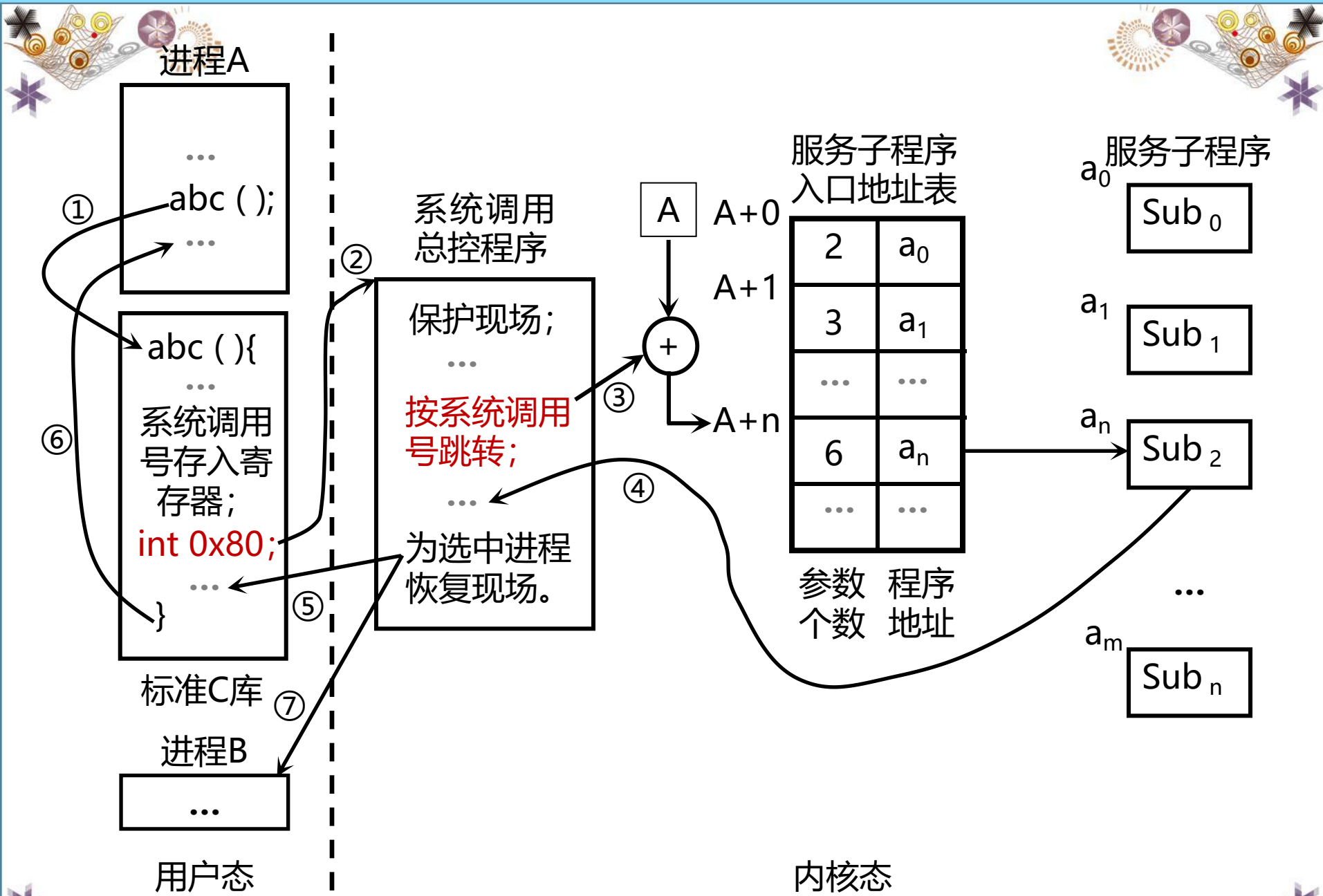
应用程序可以通过系统调用间接调用OS中的I/O过程，对I/O设备进行操作。



2. 库函数

引发系统调用的实现机制是非常依赖于机器的，而且必须用**汇编代码**表达，所以高级语言中使用函数库(如libc)提供的**应用编程接口 (API)**来设计程序，API函数再调用实际的系统调用。

输入输出的格式化是由库过程完成的，如C中的**printf**。

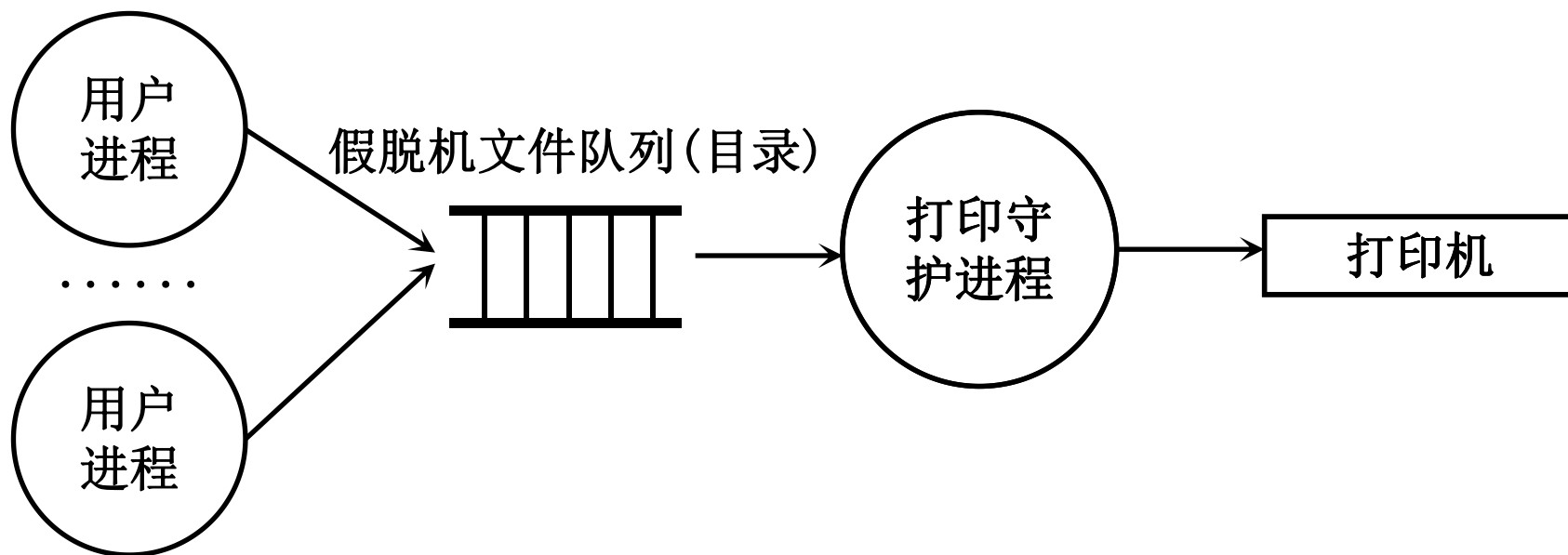


系统调用的执行过程示意图



6.6.2 假脱机(Spooling)系统

1. SP00Ling系统基本原理

利用文件模拟独占设备。



以SP00Ling方式使用外部设备示意图



6.6.2 假脱机(Spooling)系统

2. SP00Ling系统的特点

- (1) 提高了I/O的速度。
- (2) 将独占设备改造为共享设备
- (3) 实现了虚拟设备的功能。

6.7 缓冲区管理

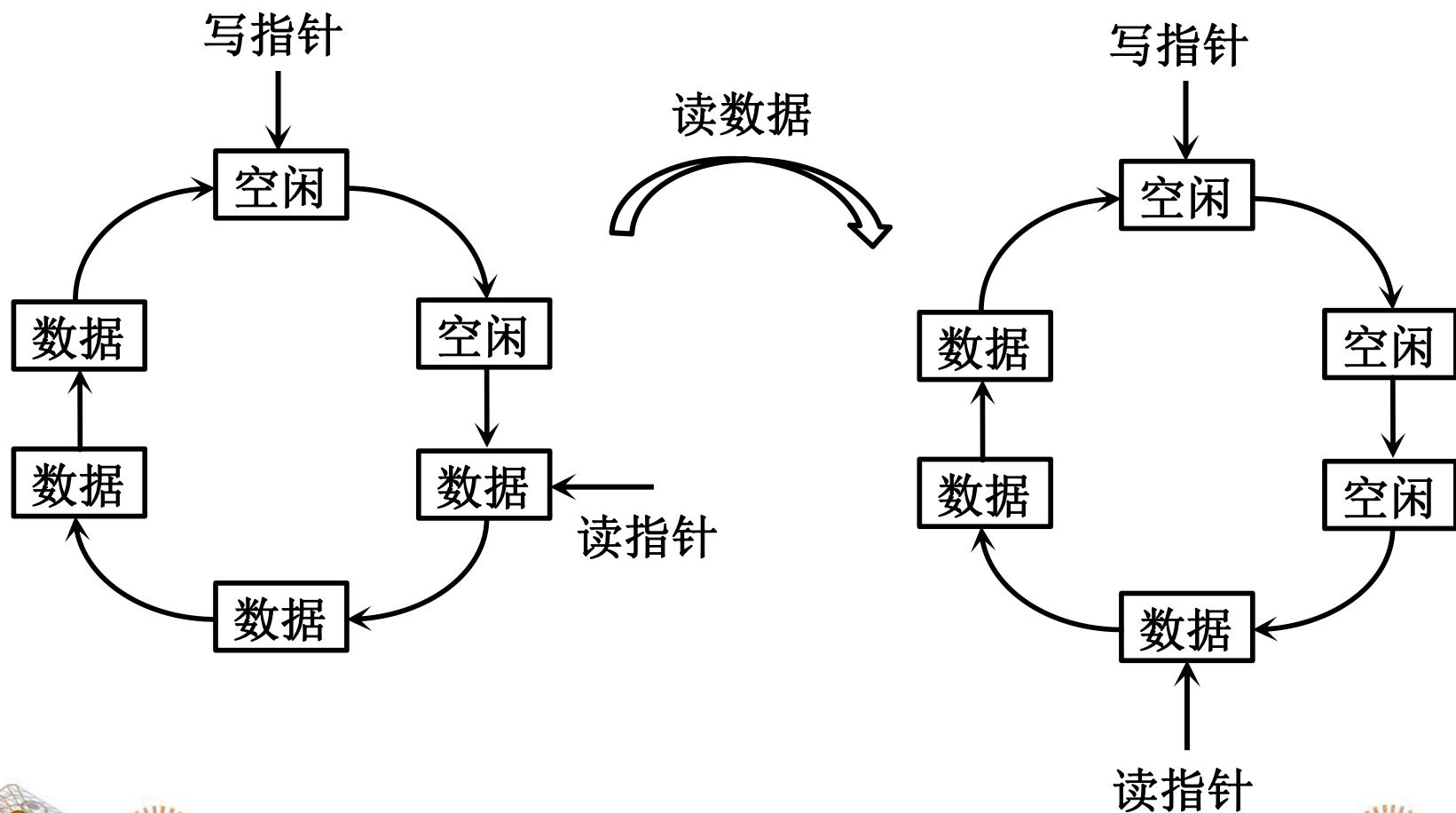
6.7.1 缓冲的引入

引入缓冲区的主要原因：

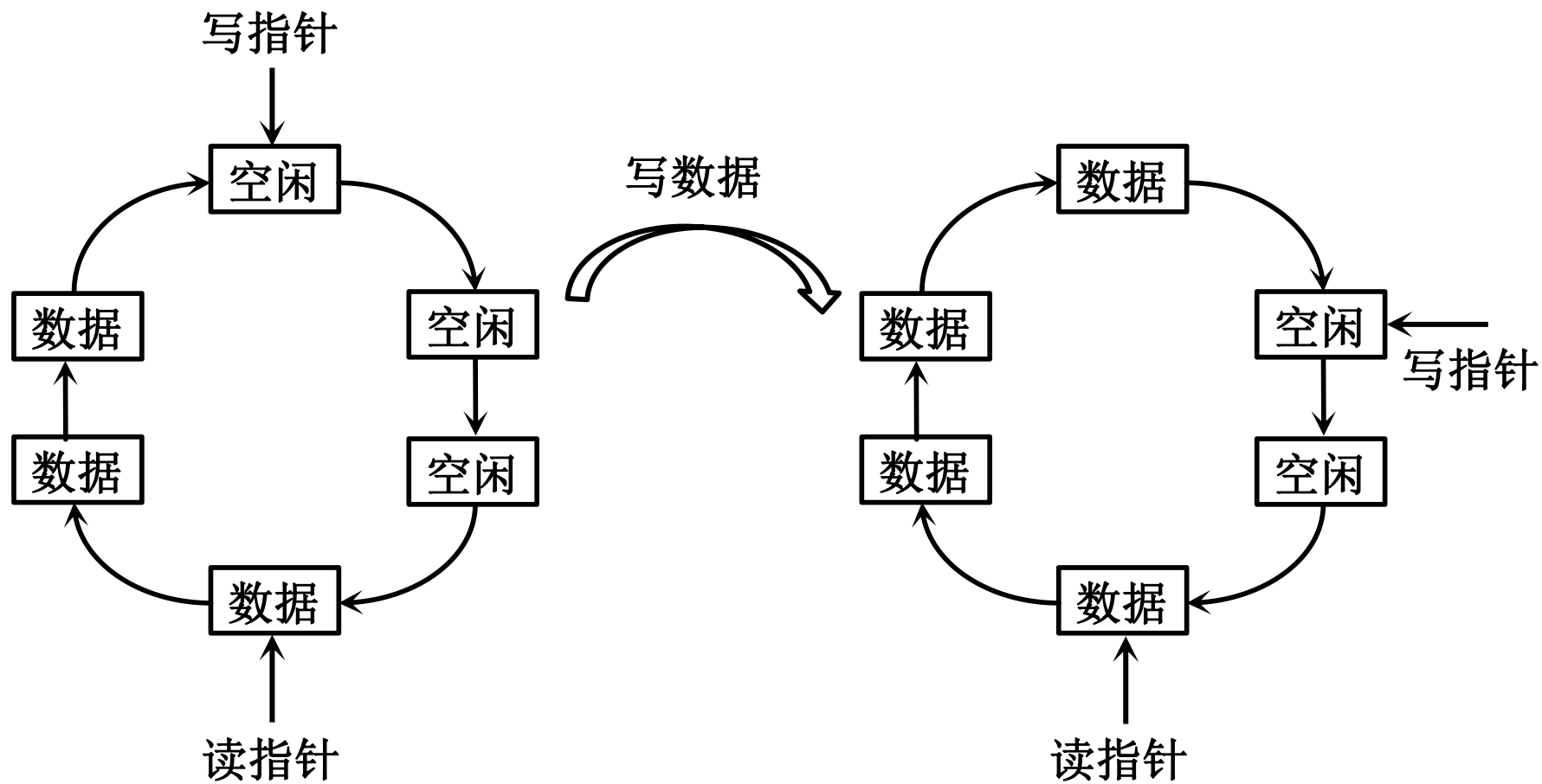
- (1) 缓和CPU与I/O设备间速度不匹配的矛盾。
- (2) 减少对CPU的中断频率，放宽对CPU中断响应时间的限制。
- (3) 解决数据粒度不匹配的问题。
- (4) 提高CPU和I/O设备之间的并行性。

6.7.2 环形缓冲区

1. 环形缓冲区的读操作



2. 环形缓冲区的写操作





3. 进程之间的同步问题

可能出现下述两种情况：

(1) 写指针追赶上读指针。全部buf满，这时写进程**阻塞**，直到读进程把某个缓冲区数据全部读完。

(2) 读指针追赶上写指针。全部buf空，这时读进程**阻塞**，直至写进程又写满某个缓冲区。





6.7.3 缓冲池(Buffer Pool)

1. 缓冲池的组成

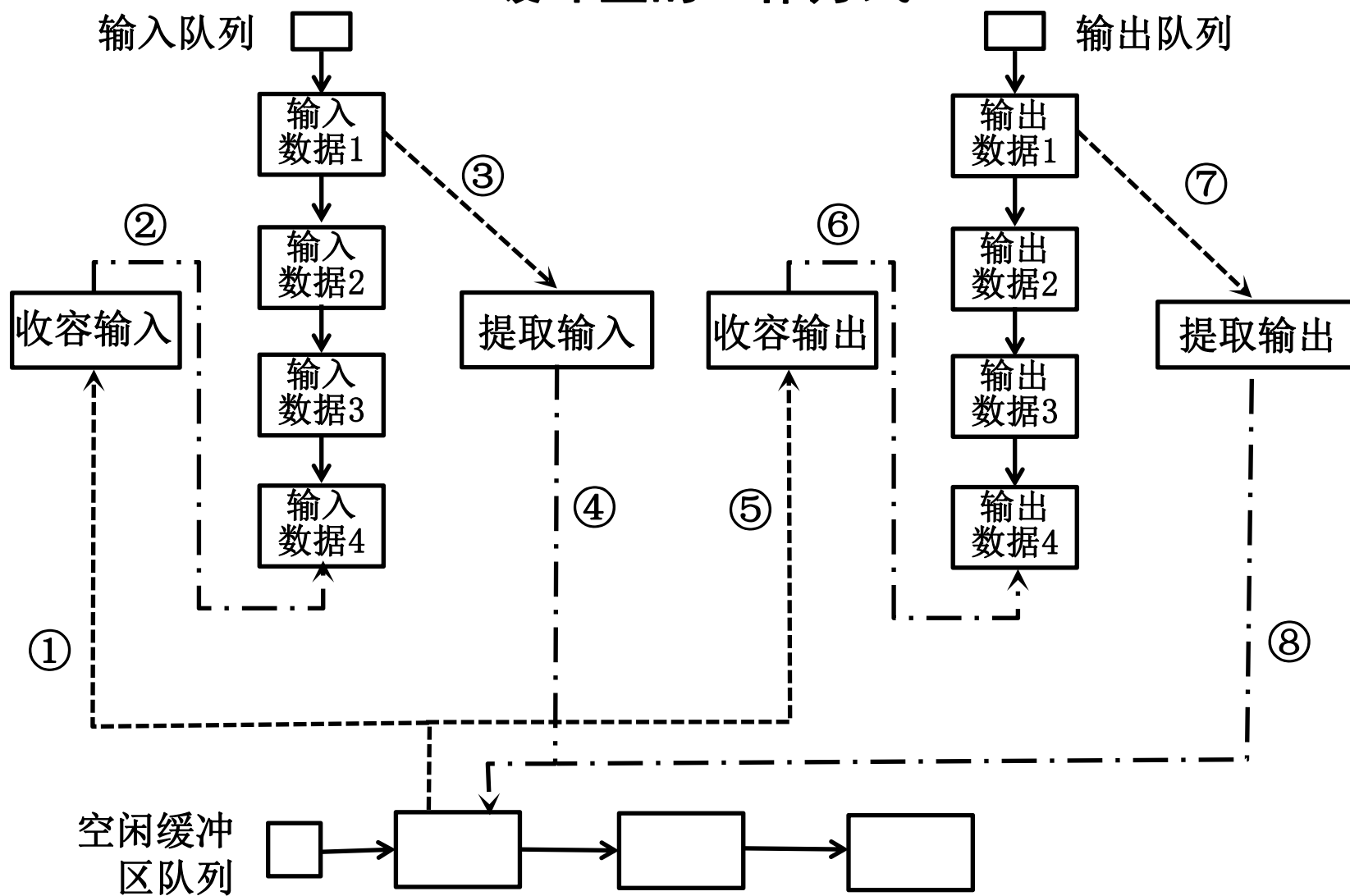
三个队列:

- (1) 空闲缓冲区队列emq。
- (2) 输入队列inq。
- (3) 输出队列outq。

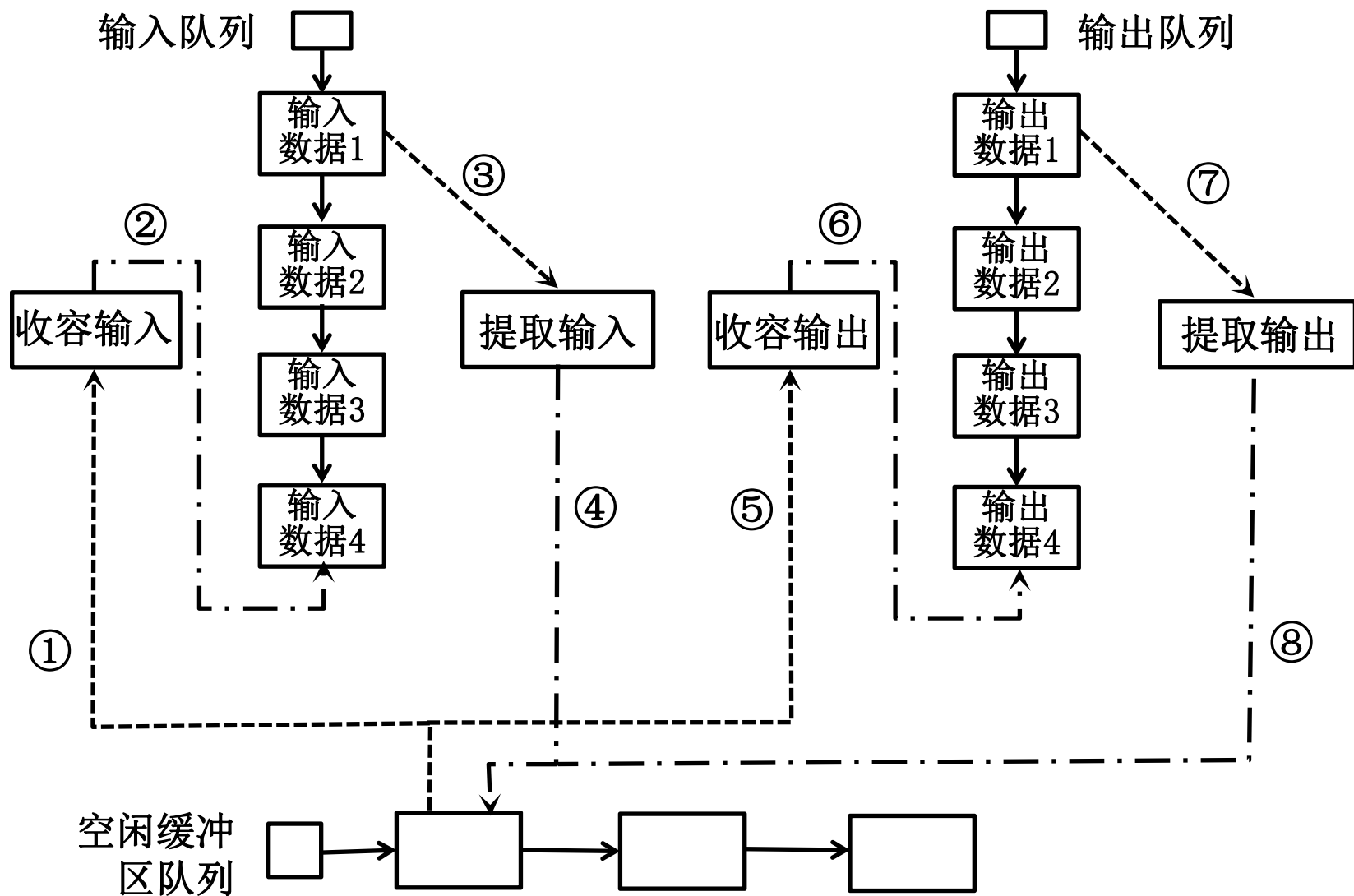
四个工作缓冲区:

- (1) hin: 收容输入数据
 - (2) sin: 提取输入数据
 - (3) hout: 收容输出数据
 - (4) sout: 提取输出数据
- 
- 

2. 缓冲区的工作方式



- ①输入进程获得收容输入缓冲区;②输入进程填满输入数据插入输入队列;
③计算进程获得提取输入缓冲区;④计算进程提取完数据释放到空闲缓冲区;



- ⑤计算进程获得收容输出缓冲区;⑥计算进程填满输出数据插入输出队列;
⑦输出进程获得提取输出缓冲区;⑧输出进程提取完数据释放到空闲缓冲区。

3. Getbuf过程和Putbuf过程

Getbuf(type)


```
{  
    Wait(Sem(type));  
    Wait(MS(type));  
    Buffer=Takebuf(type);  
    Signal(MS(type));  
}
```

Putbuf(type,Buffer)

```
{  
    Wait(MS(type));  
    Addbuf(type,Buffer);  
    Signal(MS(type));  
    Signal(Sem(type));  
}
```

思考:

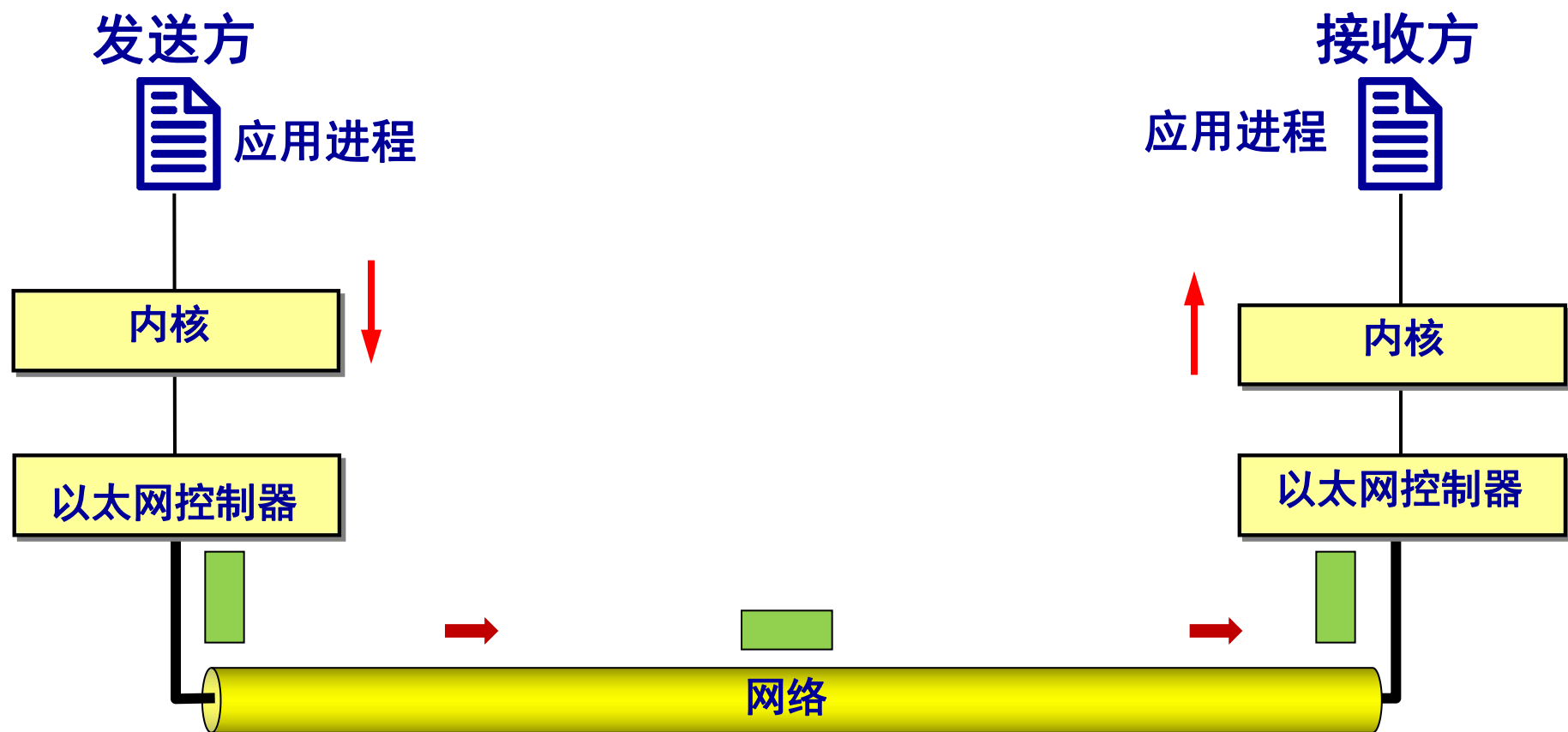
- (1) 哪个信号量用于同步?
- (2) 哪个信号量用于互斥?
- (3) 各自的使用有什么特点?





【例6.1】一个局域网以如下方式被使用：用户发出一条系统调用，请求将数据包写到网上，然后操作系统将数据拷贝到一个内核缓冲区中，再将数据拷贝到网络控制器上，当所有数据都安全存储于控制器中时，把它们在网上以10Mbit/s的速率传送。当最后一位到达时，目标CPU被中断，内核将新到达的数据包拷贝到内核缓冲区中进行检查。一旦它指明该数据包是发送给哪个用户进程的，内核就将数据拷贝到该用户进程空间。

数据包有1024字节（忽略头部），拷贝一个字节费时1微秒，将数据从一个进程注入另一个进程的最大速率是多少？

（假设发送者一直被阻塞直到接收端的工作完成并且返回一条确认消息。为简便起见，假设返回确认消息的时间可以忽略不计。）



传输过程的示意图



解：

(1) 数据包在端系统经历了4次复制，花费时间：

$$1024 \times 4 = 4096 \mu s = 0.004096 \text{ s}$$

(2) 数据包在端系统之间的发送时间：

$$1024 \times 8 \div (10 \times 10^6) = 0.0008192 \text{ s}$$

(3) 最大速率： $1024 \div (0.004096 + 0.0008192)$
 $= 208333 \text{ byte/s}$



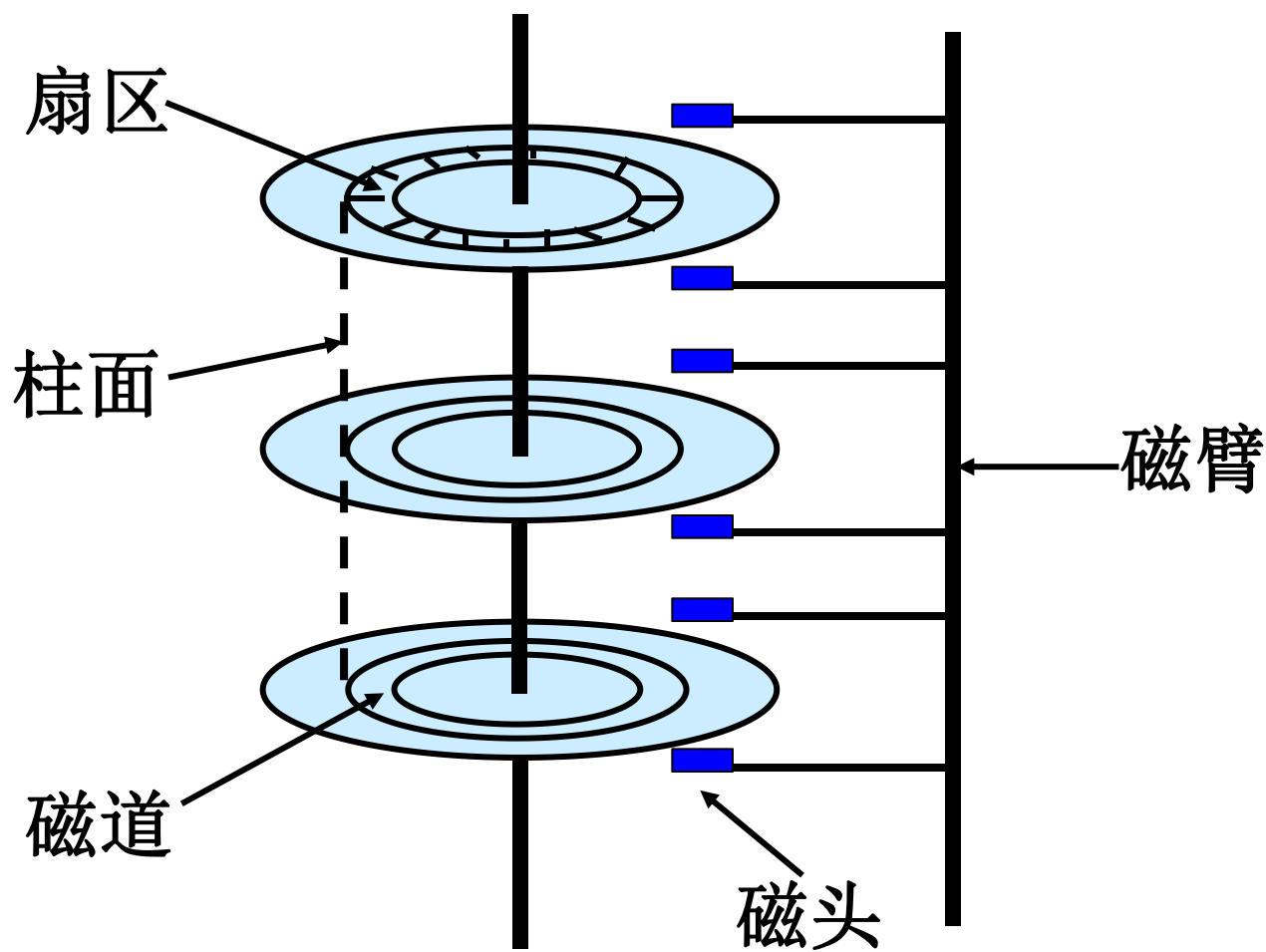
6.8 磁盘存储器的性能和调度

6.8.1 磁盘性能简述

1. 数据的组织和格式

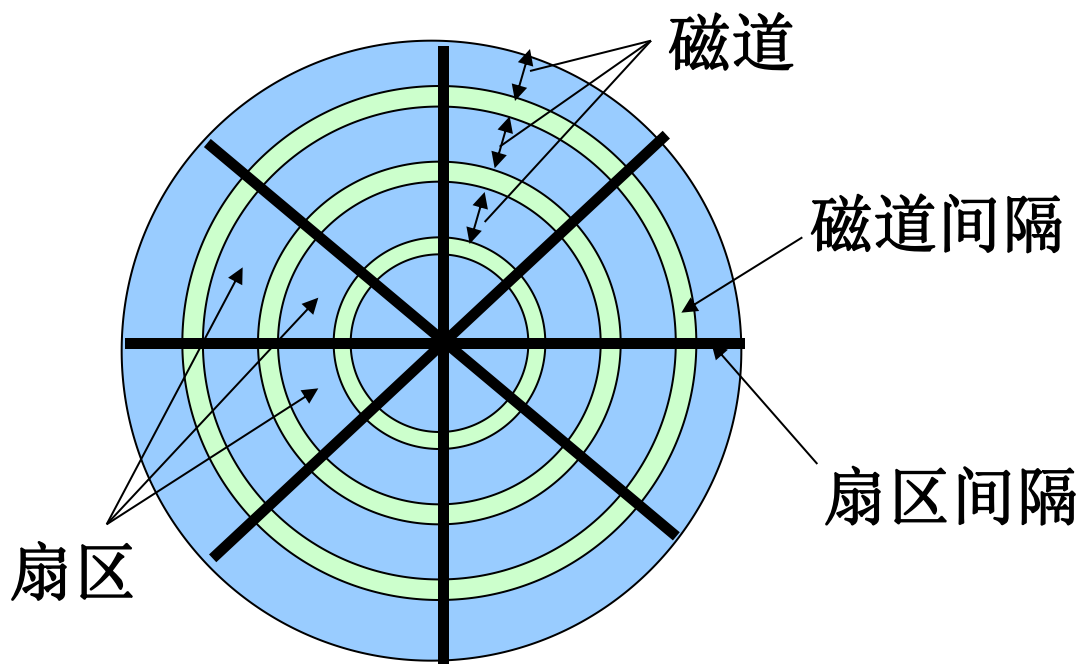
磁盘设备可包括一个或多个物理盘片，每个磁盘片分一个或两个存储面(Surface)(见图6-28(a))，每个盘面上有若干个磁道(Track)，磁道之间留有必要的间隙(Gap)。

为使处理简单起见，在每条磁道上可存储相同数目的二进制位。



(a) 磁盘驱动器的结构

图6-28 磁盘的结构和布局



(b)磁盘的数据布局

图6-28 磁盘的结构和布局

2. 磁盘的类型



- (1) 固定头磁盘：每条磁道上都有一读写磁头。
- (2) 移动头磁盘。

3. 磁盘访问时间

磁盘设备在工作时以恒定速率旋转。为了读或写，磁头必须能**移动**到所指定的磁道上，并等待所指定的扇区的开始位置**旋转**到磁头下，然后再开始**读或写**数据。

磁盘总的访问时间 T_a 表示为：

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$



(1) 寻道时间：把磁头从当前位置移到指定磁道上所经历的时间： $T_s = m \times n + s$

s 启动磁臂的时间。

m 磁头移动一条磁道花费的时间。

(2) 旋转延迟时间：指定扇区移动到磁头下面所经历的时间。平均： $T_r = 1/2r$



r 为磁盘每秒钟的转数。

(3) 传输时间：数据从磁盘读出或向磁盘写入数据所经历的时间： $T_t = b/rN$

b 为每次所读/写的字节数。

N 为一条磁道上的字节数。







【例6.2】一个快速SCSI-II总线上的磁盘转速为7200RPM，每磁道160个扇区，每扇区512字节。那么理想状态下其数据传输率为（ ）。

A. $7200 \times 160\text{KB/S}$

B. 7200KB/S

C. 9600KB/S

D. 19200KB/S



解：

- (1) 每磁道的字节数为： $160 \times 0.5\text{KB} = 80 \text{ KB}$;
- (2) 旋转一圈的时间为： $60/7200 = 1/120 \text{ S}$;
- (3) 数据传输率为： $80 / (1/120) = 9600 \text{ KB/S}$ 。

6.8.2 早期的磁盘调度算法

1. 先来先服务 (FCFS)

这是最简单的磁盘调度算法。它根据进程请求访问磁盘的先后次序进行调度。

(从100号磁道开始)	
被访问的下一个磁道号	移动距离 (磁道数)
55	45
58	3
39	19
18	21
90	72
160	70
150	10
38	112
184	146
平均寻道长度: 55.3	

图6-30 FCFS调度算法

2. 最短寻道时间优先 (SSTF)

该算法要求访问的磁道与当前磁头所在的磁道距离最近，以使每次的寻道时间最短，但这种算法不能保证平均寻道时间最短。

(从100号磁道开始)	
被访问的下一个磁道号	移动距离 (磁道数)
90	10
58	32
55	3
39	16
38	1
18	20
150	132
160	10
184	24
平均寻道长度: 27.5	

图6-31 SSTF调度算法



6.8.3 基于扫描的磁盘调度算法

1. 扫描(SCAN)算法

SSTF算法的实质是基于优先级的调度算法，因此就可能导致优先级低的进程发生“**饥饿**”(Starvation)现象。因为只要不断有新进程的请求到达，且其所要访问的磁道与磁头当前所在磁道的距离较近，这种新进程的I/O请求必然优先满足。

扫描算法不仅考虑所要访问的磁道与磁头当前所在磁道的距离较近，更优先考虑磁头当前的**移动方向**，也称为**电梯调度算法**。

(从100号磁道开始，向磁道号增加的方向访问)	
被访问的下一个磁道号	移动距离 (磁道数)
150	50
160	10
184	24
90	94
58	32
55	3
39	16
38	1
18	20
平均寻道长度：27.8	

图6-32 SCAN调度算法示例





2. 循环扫描 (CSCAN) 算法

SCAN算法既能获得较好的寻道性能，又能防止“饥饿”现象。

但也存在这样的问题：当磁头刚从里向外移动而越过了某一磁道时，恰好又有一进程请求访问此磁道，这时，该进程必须等待，待磁头继续从里向外，然后再从外向里扫描完处于外面的所有要访问的磁道后，才处理该进程的请求，致使该进程的请求被大大地推迟。

循环扫描算法规定磁头**单向移动**。



(从100号磁道开始，向磁道号增加的方向访问)	
被访问的下一个磁道号	移动距离 (磁道数)
150	50
160	10
184	24
18	166
38	20
39	1
55	16
58	3
90	32
平均寻道长度：35.8	

图6-33 CSCAN调度算法示例

3. NStepSCAN和FSCAN调度算法

(1) NStepSCAN算法

在SSTF、SCAN及CSCAN几种调度算法中，都可能出现磁臂停留在某处不动的情况。

例如，有一个或几个进程对某一磁道有较高的访问频率，即这个(些)进程反复请求对某一磁道的I/O操作，从而垄断了整个磁盘设备。这一现象称为“**磁臂粘着**”(Armstickiness)。



N步SCAN算法:

- 1) 将磁盘请求队列分成若干个长度为N的子队列。
- 2) 队列之间使用FCFS算法。
- 3) 队列内部使用SCAN算法。
- 4) 新的I/O请求，放入其他队列，避免粘着现象。

N步SCAN算法的特点:

当N值很大时，性能接近于SCAN算法。当N=1时，蜕化为FCFS算法。

(2) FSCAN算法

FSCAN算法实质上是N步SCAN算法的简化，即FSCAN只将磁盘请求队列分成两个子队列：

- 1) 一个是由当前所有请求磁盘I/O的进程形成的队列，由磁盘调度按SCAN算法进行处理；
- 2) 另一个是在扫描期间，将新出现的所有请求磁盘I/O的进程放入等待处理的请求队列。这样，所有的新请求都将被推迟到下一次扫描时处理。