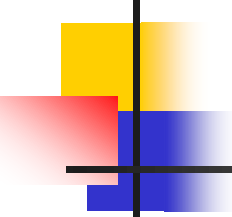


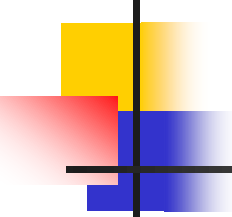
第5章 设备管理

I/O设备实现了计算机的输入与输出功能，如何提高**I/O**设备的使用效率是操作系统设备管理模块的主要任务。本章从设备管理的概念出发，介绍了设备管理中的相关技术与策略。



教学要求

- ◆ 熟悉四种I/O的控制方式；掌握通道的概念，熟悉通道类型。
- ◆ 熟悉缓冲的概念，熟悉缓冲池工作原理，了解UNIX系统的缓冲技术。
- ◆ 掌握设备分配的策略，理解设备的独立性概念；掌握设备分配中数据结构，熟悉设备分配的流程；掌握SP00Ling技术概念和SP00Ling系统的组成。
- ◆ 熟悉设备驱动程序的功能和处理方式，熟悉设备驱动程序的处理过程。



教学内容

5.1 设备管理概述

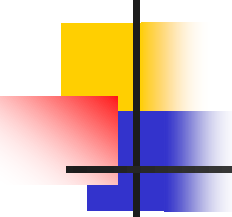
5.2 I/O控制方式

5.3 设备管理技术

5.4 设备的分配

5.5 I/O软件

5.6 Linux的设备管理

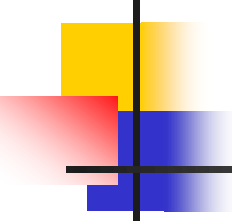


5.1 设备管理概述

5.1.1 设备的分类

1、按传输速率分类

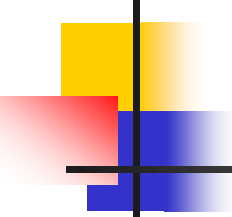
- ◆低速设备：1-10KB/S。如Modem、键盘
- ◆中速设备：10K-100K/S。如打印机
- ◆高速设备：100K-100M/S。如磁盘、磁带



5.1.1 设备的分类（续）

2、按信息组织方式分类

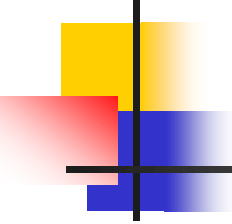
- ◆ 字符设备：I/O传输的单位是字节。
如打印机
- ◆ 块设备：I/O传输的单位是块。如磁盘、磁带



5.1.1 设备的分类（续）

3. 按资源属性分类

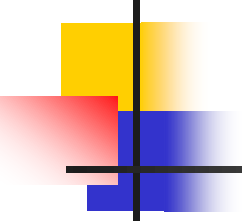
- ◆ 独占设备：在任一段时间内最多有一个进程占用它，如字符设备及磁带机。
- ◆ 共享设备：多个进程对它的访问可以交叉进行，除磁带机外的块设备属共享设备。
- ◆ 虚拟设备：利用虚拟技术将一台独占设备转换为能被多个用户使用的共享设备。这种被虚拟化后具有新特性的设备称为虚拟设备。



5.1.1 设备的分类（续）

4、按从属关系分类

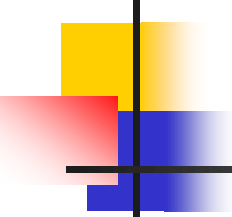
- ◆ 系统设备：
- ◆ 用户设备：

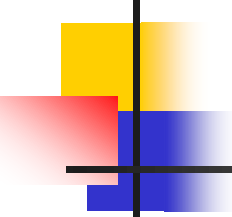


5.1.2 设备控制器

I/O设备包括一个机械部件和一个电子部件。为了达到设计的模块性和通用性，一般将其分开。

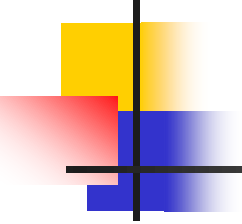
- 电子部件称为设备控制器或适配器，在PC中，它常常是插入主板扩充槽的印刷电路板；
- 机械部件则是设备本身。

- 
-
- 操作系统基本上与控制器打交道，而非设备本身。
 - 多数PC的CPU和控制器之间的通信采用单总线模型，CPU直接控制设备控制器进行I/O；而主机则采用多总线结构和通道方式，以提高CPU与输入输出的并程度。



1. 设备控制器的功能

- 接收和识别命令
- 数据交换
- 标识和报告设备的状态
- 设备地址识别
- 数据缓冲
- 差错控制



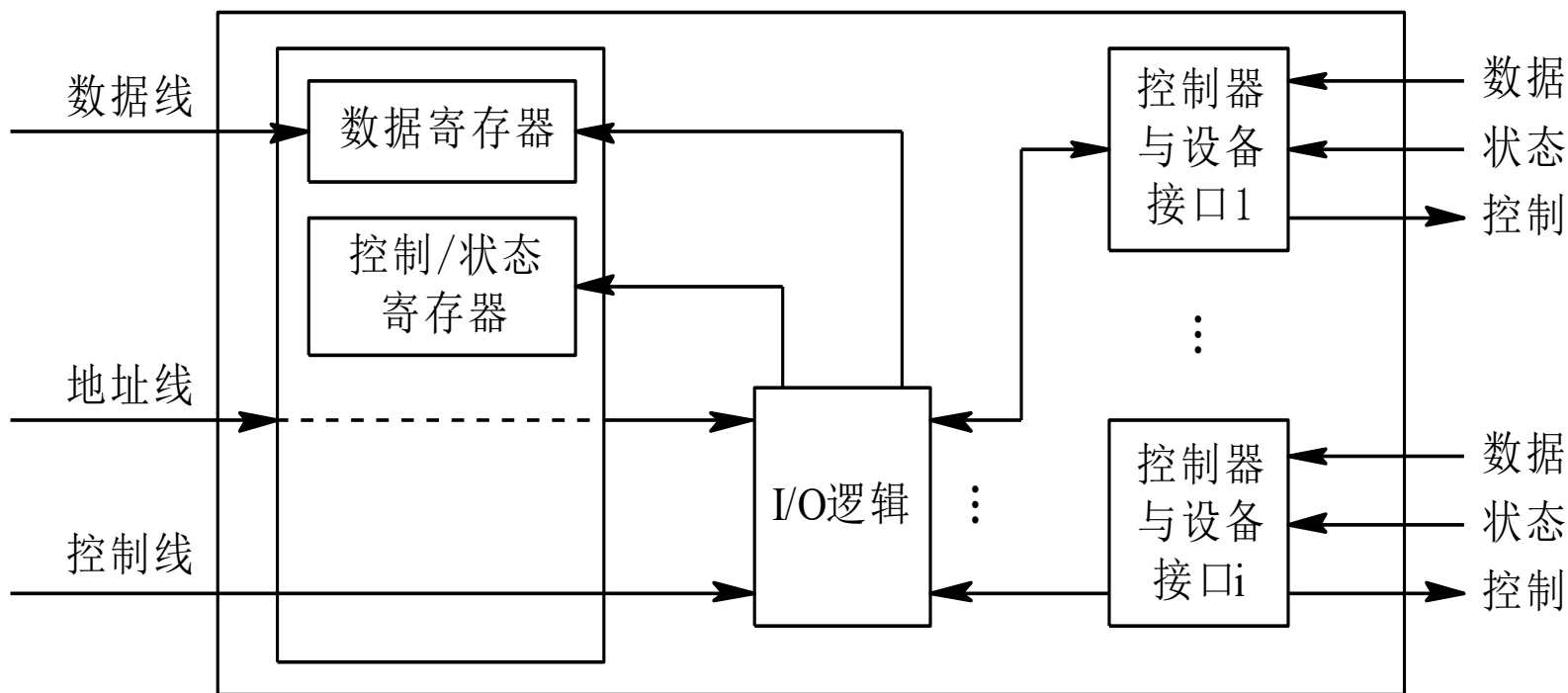
2. 设备控制器的组成

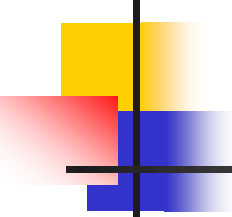
- 命令寄存器及译码器
- 数据寄存器
- 状态寄存器
- 地址译码器
- 用于对设备控制的I/O逻辑

设备控制器的组成

CPU与控制器接口

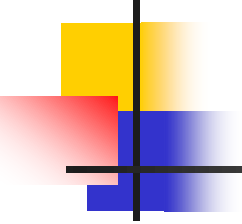
控制器与设备接口





设备控制器功能小结

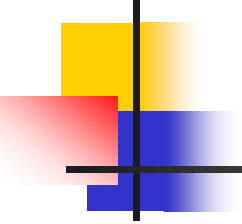
- 设备控制器是CPU和设备之间的一个接口, 它接收从CPU发来的命令, 控制I/O设备操作, 实现主存和设备之间的数据传输。
- 设备控制器是一个可编址设备, 当它连接多台设备时, 则应具有多个设备地址。



5.1.3 I/O通道

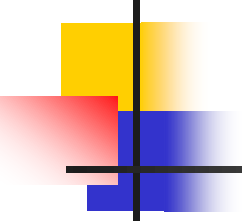
1. I/O通道的引入

为使中央处理机从繁忙的I/O处理中摆脱出来，现代大、中型计算机系统中设置了专门的处理I/O操作的处理机，并把这种处理机称为通道。



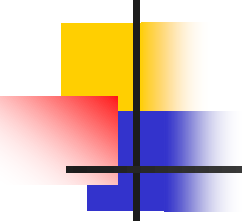
5.1.3 I/O通道（续）

- 通道在CPU的控制下独立地执行通道程序，对外部设备的I/O操作进行控制，以实现内存与外设之间成批的数据交换。
- 通道程序是由通道指令组成，一个通道可以分时的方式执行几个通道程序。每个程序控制一台外部设备，因此，**每个通道程序称为子通道。**



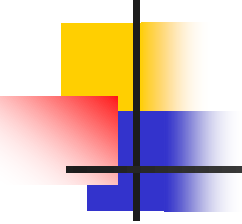
通道与一般处理机的区别：

- 指令类型单一，主要是I/O指令；
- 通道没有自己的内存，通道所执行的通道程序是放在主机的内存中的，通道与CPU共享内存；



2. 通道类型

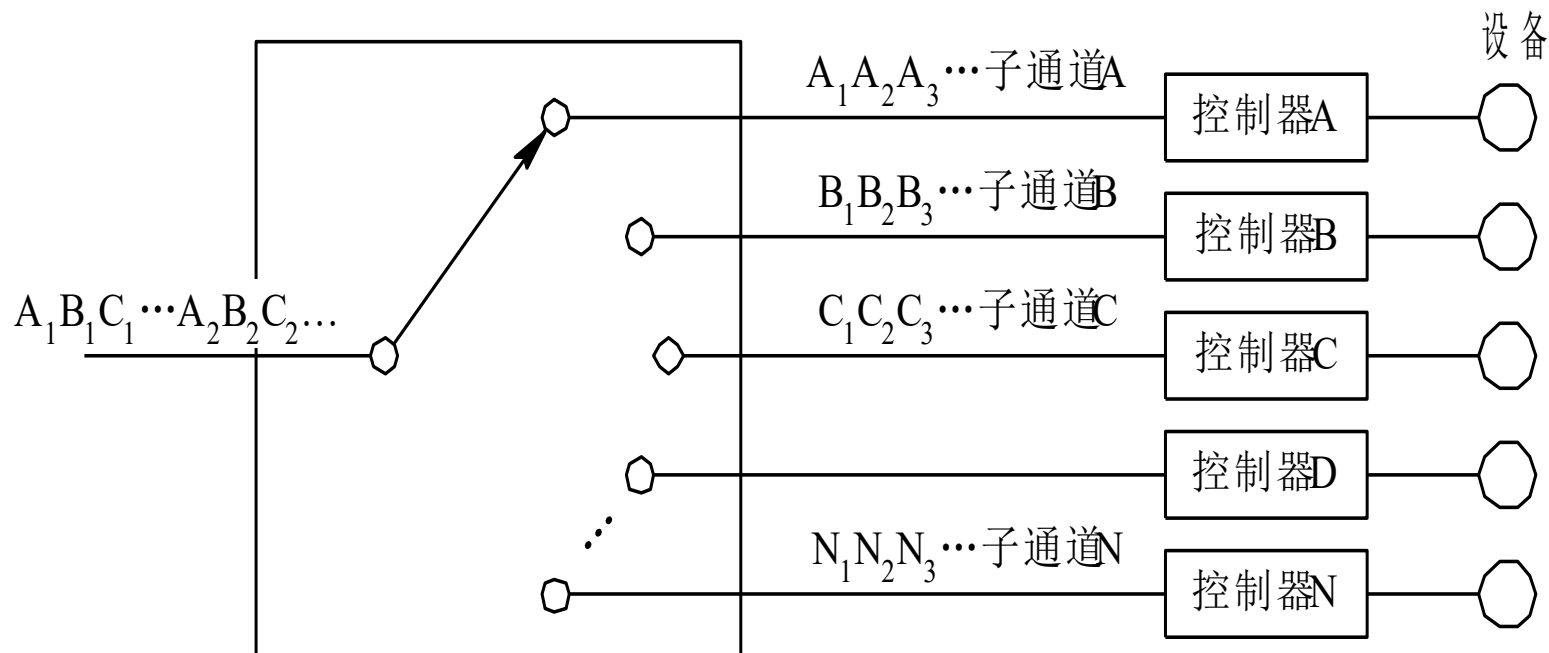
- 字节多路通道(Byte Multiplexor Channel)
- 选择通道(Selector Channel)
- 数组多路通道(Block Multiplexor Channel)

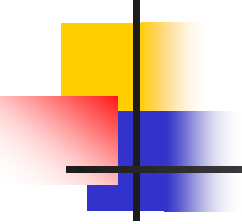


(1) 字节多路通道

- 子通道以字节为单位轮流为各自控制的外设传送数据。
- 主要用来控制低速、以字节为基本传送单位的设备。如打印机。

字节多路通道

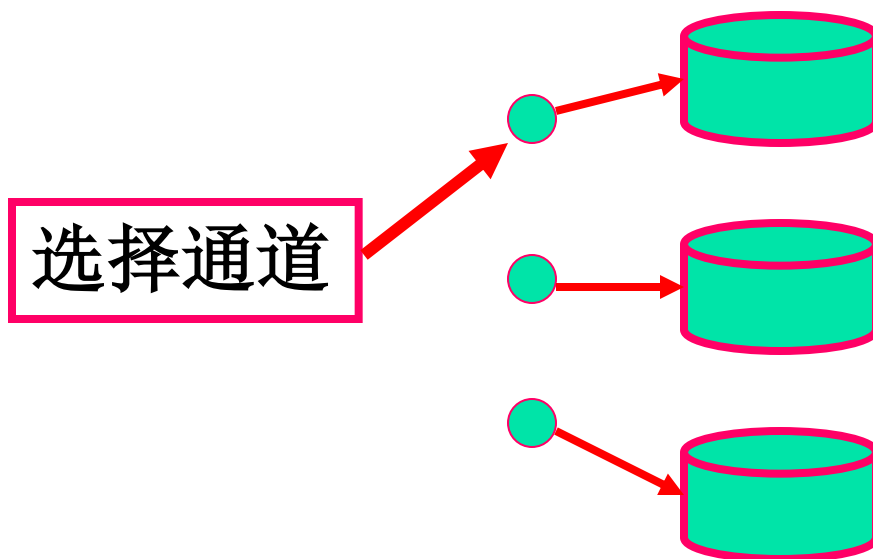
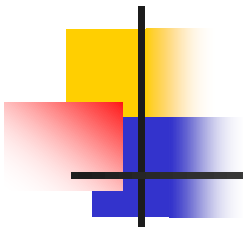


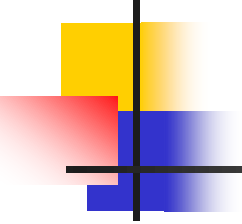


(2) 选择通道

一次执行一个通道程序，控制一台设备连续地传送一批数据，当一个程序执行完后，才转向下一个程序。

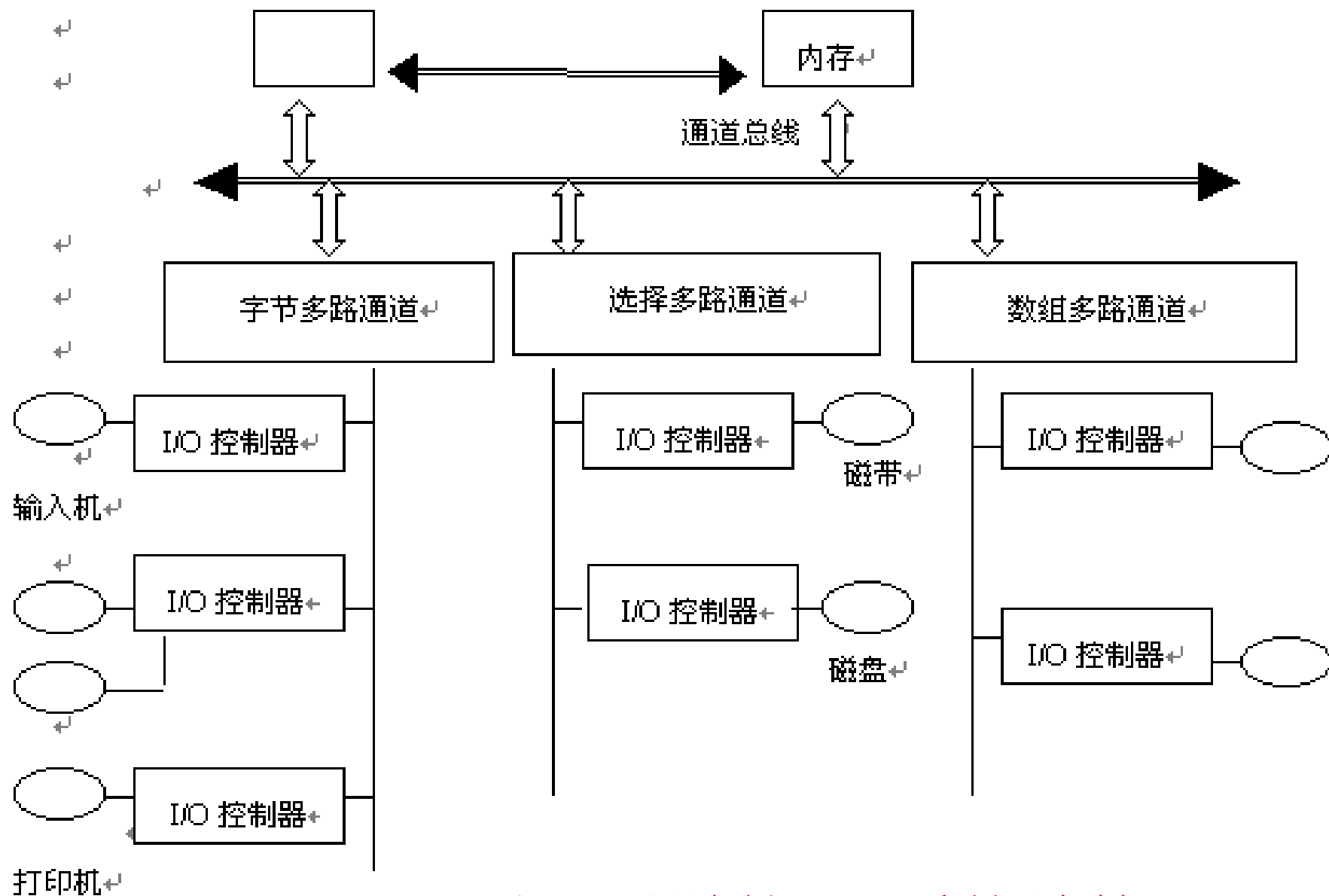
- **优点：**传输速度快，主要用来控制高速外设，如磁盘。
- **缺点：**一次只能控制一台设备进行I/O操作。



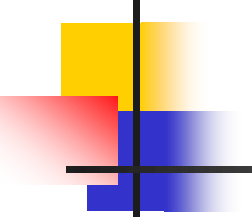


(3) 数组多路通道

是上述两种通道的折中，以分时的方式执行多道程序，每道程序可传送一组数据。用于连接多台中高速的外围设备。

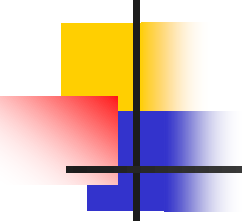


引入通道的I/O系统结构



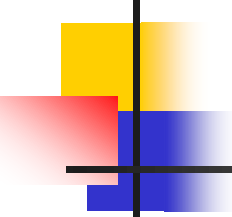
3、通道的发展

- 在**PC**中，芯片组中有专门的**I/O**处理芯片，称为**IOP (IO Processor)**，发挥通道的作用
- **IBM**于**1998**年推出光纤通道技术（称为**FICON**），可通过 **FICON** 连接多达**127**个大容量**I/O**设备。传输速度是**333MHz / s**，目前已经达到**1GHz / s**。
- 光纤通道技术具有数据传输速率高、传输距离远，可简化大型存储系统设计的优点
- 在大容量高速存储，如大型数据库、多媒体、数字影像等应用领域，有广泛前景



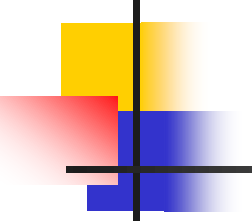
5.1.4 设备管理的功能

- 1、中断处理
- 2、缓冲区管理
- 3、设备分配
- 4、设备驱动调度
- 5、虚拟设备及其实现



5.2 I/O控制方式

- 1、程序直接控制方式
- 2、中断控制方式
- 3、DMA 方式
- 4、通道控制方式

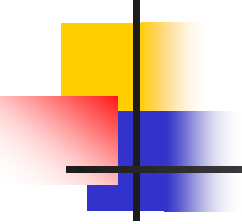


5.2.1 程序直接控制方式

这是早期的方式，CPU通过测试I/O设备的状态为Ready（就绪）后才启动数据的传输。若没有就绪，CPU不断地进行循环测试，直到其就绪为止。

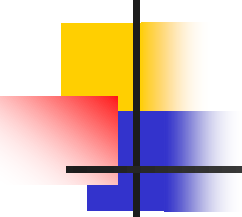
缺点：CPU效率低下

原因：无中断机构



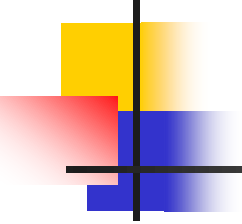
5.2.2 中断控制方式

CPU发出启动I/O设备后不再测试I/O设备的状态，而是执行其他程序。当I/O设备就绪后向CPU发出中断信号，CPU响应中断后执行中断处理程序，控制一个字（或字节）的传输过程。当数据传输未完成时，CPU再次启动设备，重复上述过程。



【例】 从终端输入一个字符的时间约为100 ms，而将字符送入终端缓冲区的时间小于0.1 ms。

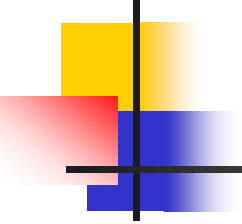
- 程序直接控制方式，CPU约有 99.9 ms的时间处于忙—等待中。
- 中断控制方式，CPU可利用这 99.9 ms的时间去做其它事情，而仅用 0.1 ms的时间来处理由控制器发来的中断请求。



中断控制方式（续）

缺点：每台设备每输入输出一个字（或字节）的数据都有一次中断。如果设备较多时，中断次数会很多，使**CPU**的计算时间大大减少。

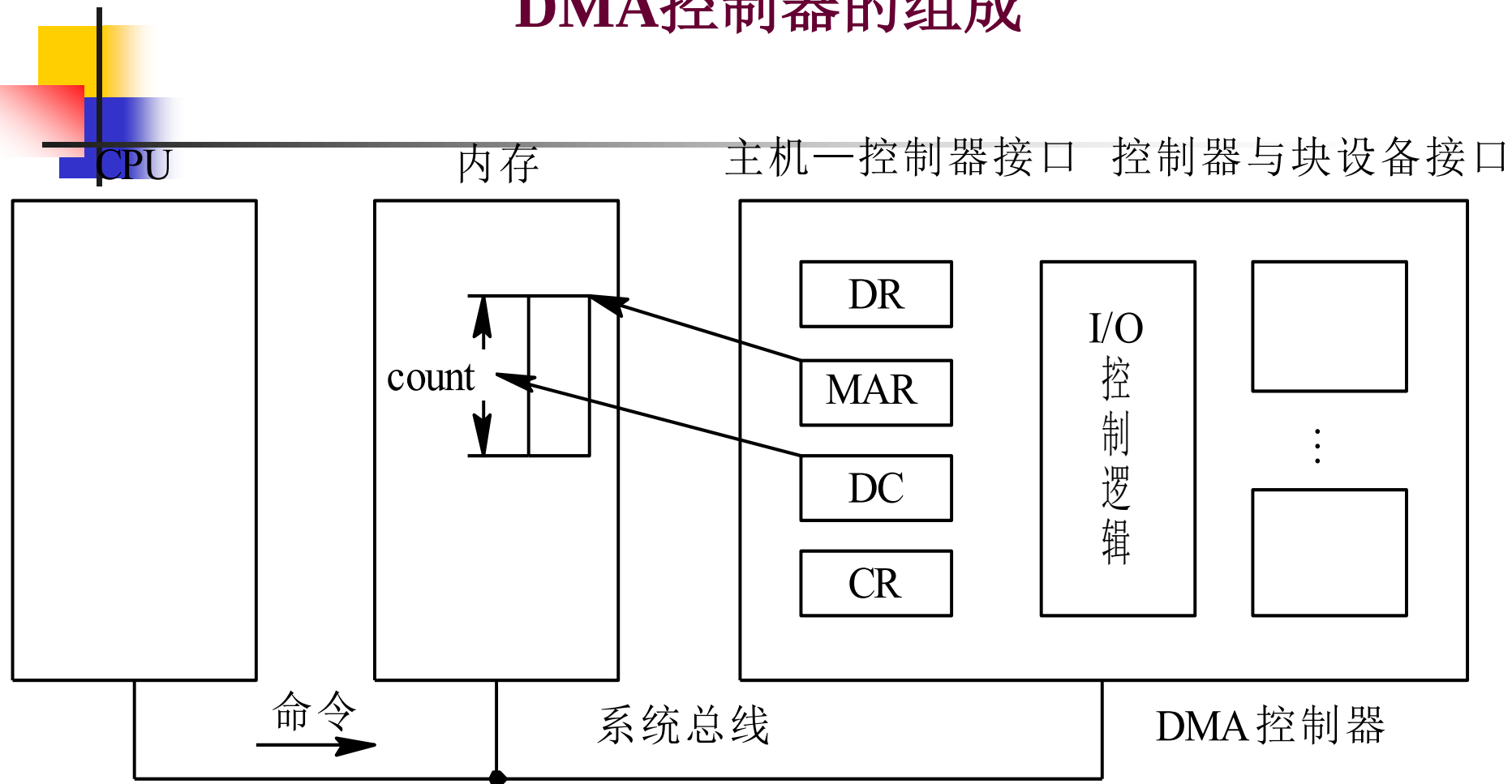
为减少中断对**CPU**造成的负担，可采用**DMA**方式和通道方式。

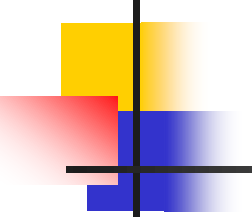


5.2.3 DMA方式

- 数据传输的基本单位是数据块，即在CPU与I/O设备之间，每次传送至少一个数据块；
- 仅在传送一个或多个数据块的开始和结束时，才需CPU干预，整块数据的传送是在DMA控制器的控制下完成的。

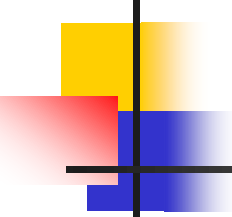
DMA控制器的组成





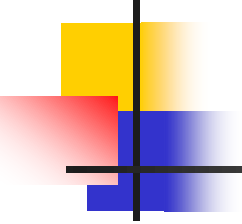
DMA控制器中的四类寄存器:

- **命令/状态寄存器CR**。用于接收从CPU发来的I/O命令或有关控制信息，或设备的状态。
- **内存地址寄存器MAR**。在输入时，它存放把数据从设备传送到内存的起始目标地址；在输出时，它存放由内存到设备的内存源地址。
- **数据寄存器DR**。用于暂存从设备到内存，或从内存到设备的数据。
- **数据计数器DC**。存放本次CPU要读或写的字(节)数。



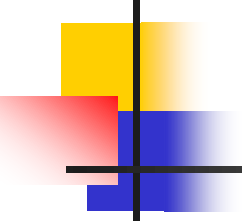
DMA方式与中断方式的区别

- 中断方式是在数据缓冲寄存区满后，发中断请求，**CPU**进行中断处理；**DMA**方式则是在所要求传送的数据块全部传送结束时要要求**CPU**进行中断处理，大大减少了**CPU**进行中断处理的次数
- 中断方式的数据传送是由**CPU**控制完成的；而**DMA**方式则是在**DMA**控制器的控制下不经过**CPU**控制完成的

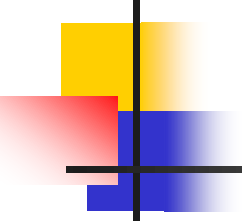


5.2.4 通道控制方式

通道控制方式是DMA方式的发展，它可进一步减少CPU的干预，即把对一个数据块的读(或写)为单位的干预，减少为对一组数据块的读(或写)及有关的管理为单位的干预。



例如，当CPU要完成一组相关的读(或写)操作及有关控制时，只需向I/O通道发送一条I/O指令，以给出其所要执行的通道程序的首址和要访问的I/O设备，通道接到该指令后，通过执行通道程序便可完成CPU指定的I/O任务。

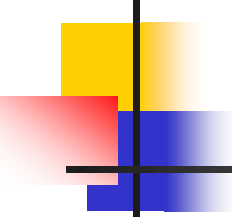


通道程序

- (1)操作码。表示指令执行的操作，如读、写、控制等。
- (2) 内存地址。
- (3) 计数。
- (4) 通道程序结束位P。P=1 表示最后一条指令
- (5) 记录结束标志R。R=0表示本通道指令与下一条指令所处理的数据是同属于一个记录；

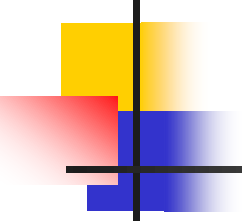
【示例】：将内存中不同地址的数据，写成多个记录。

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



5.3 设备管理技术

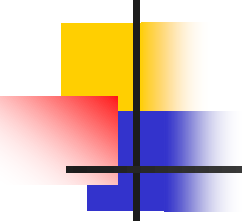
- 缓冲技术
- 磁盘驱动调度技术
- **SPOOLING**技术



5.3.1 缓冲技术

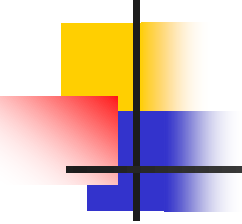
1、引入缓冲的原因

- (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)提高CPU和I/O设备之间的并行性



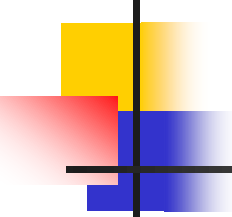
2、实现缓冲的基本思想

- 缓冲可以分为**硬件缓冲**和**软件缓冲**两种。
硬件缓冲通过硬件来实现，在设备控制器中有硬件缓冲区，通常容量较小，一般为几十到几百KB。
- 软件缓冲是在内存中开辟出一个存储区作为缓冲区，用于存放I/O数据。



软件缓冲的实现原理

- 进程执行写操作输出数据时，向系统申请一个缓冲区，再把数据填到缓冲区。此后，进程继续执行，系统将缓冲区内容写到I/O设备上。
- 进程执行读操作输入数据时，向系统申请一个缓冲区，用于保存读进的物理记录，然后再把进程需要的内容从缓冲区中选出。



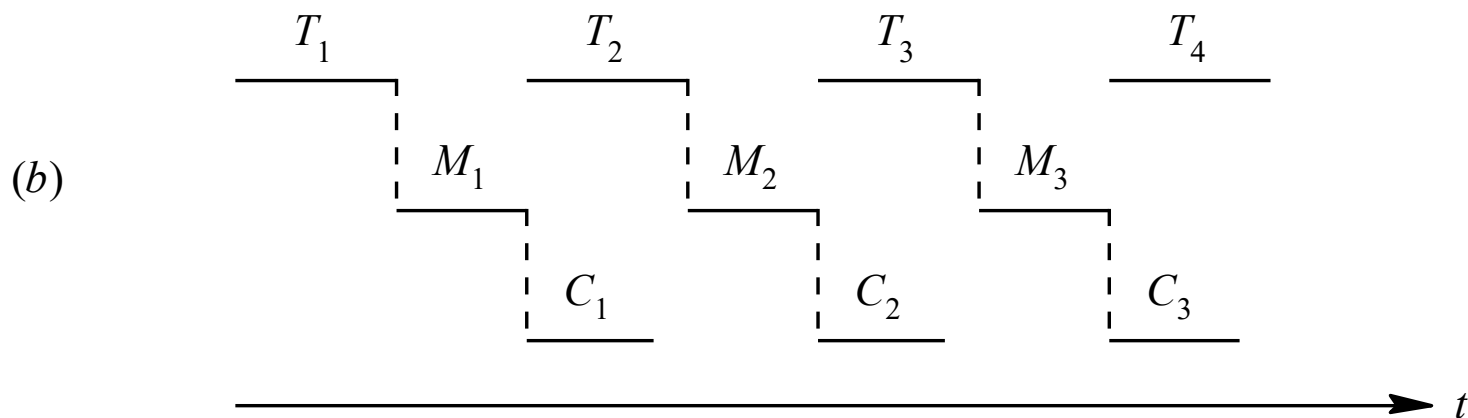
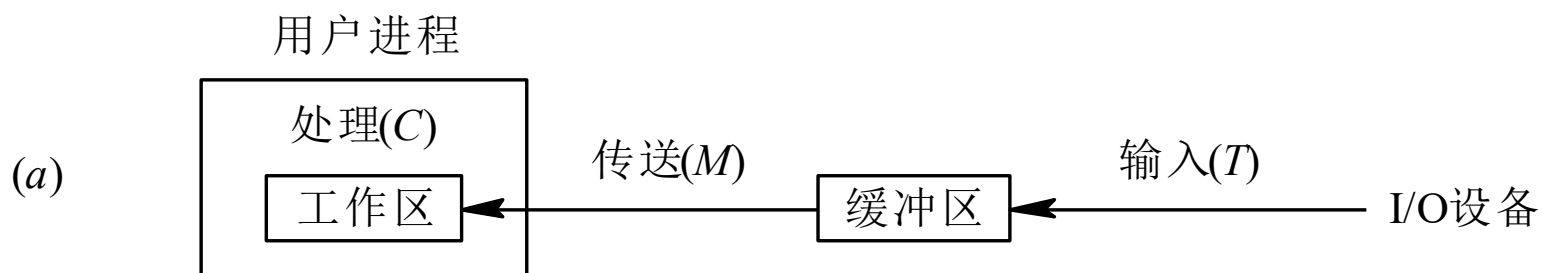
3、常用的缓冲技术

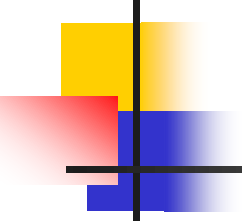
(1) 单缓冲(Single Buffer)

当进程发出一**I/O**请求时，**OS**为之分配一缓冲区

- 对于输入：设备先将数据送入缓冲区，**OS**再将数据传给进程。
- 对于输出：进程先将数据传入缓冲区，**OS**再将数据送出到设备。

单缓冲工作示意图

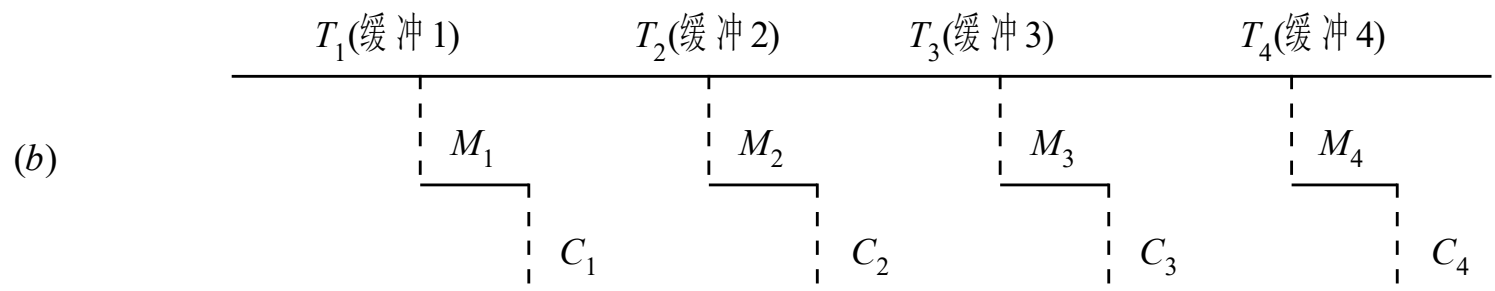
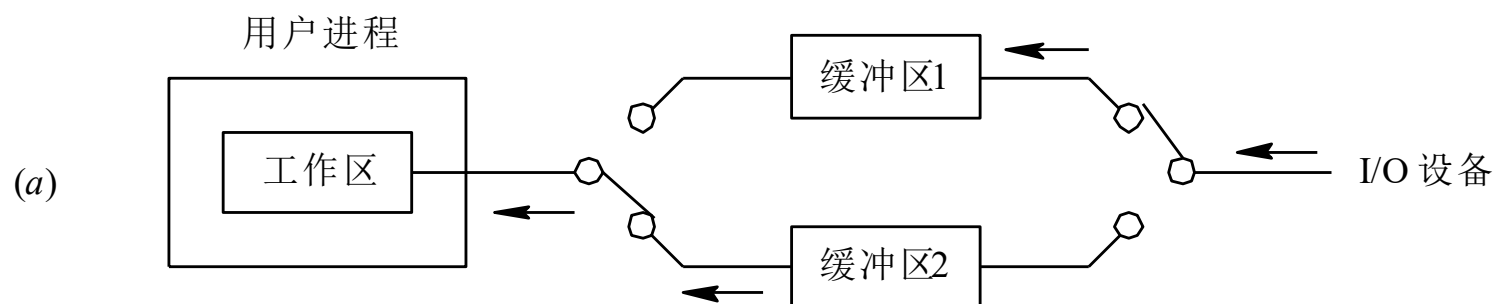




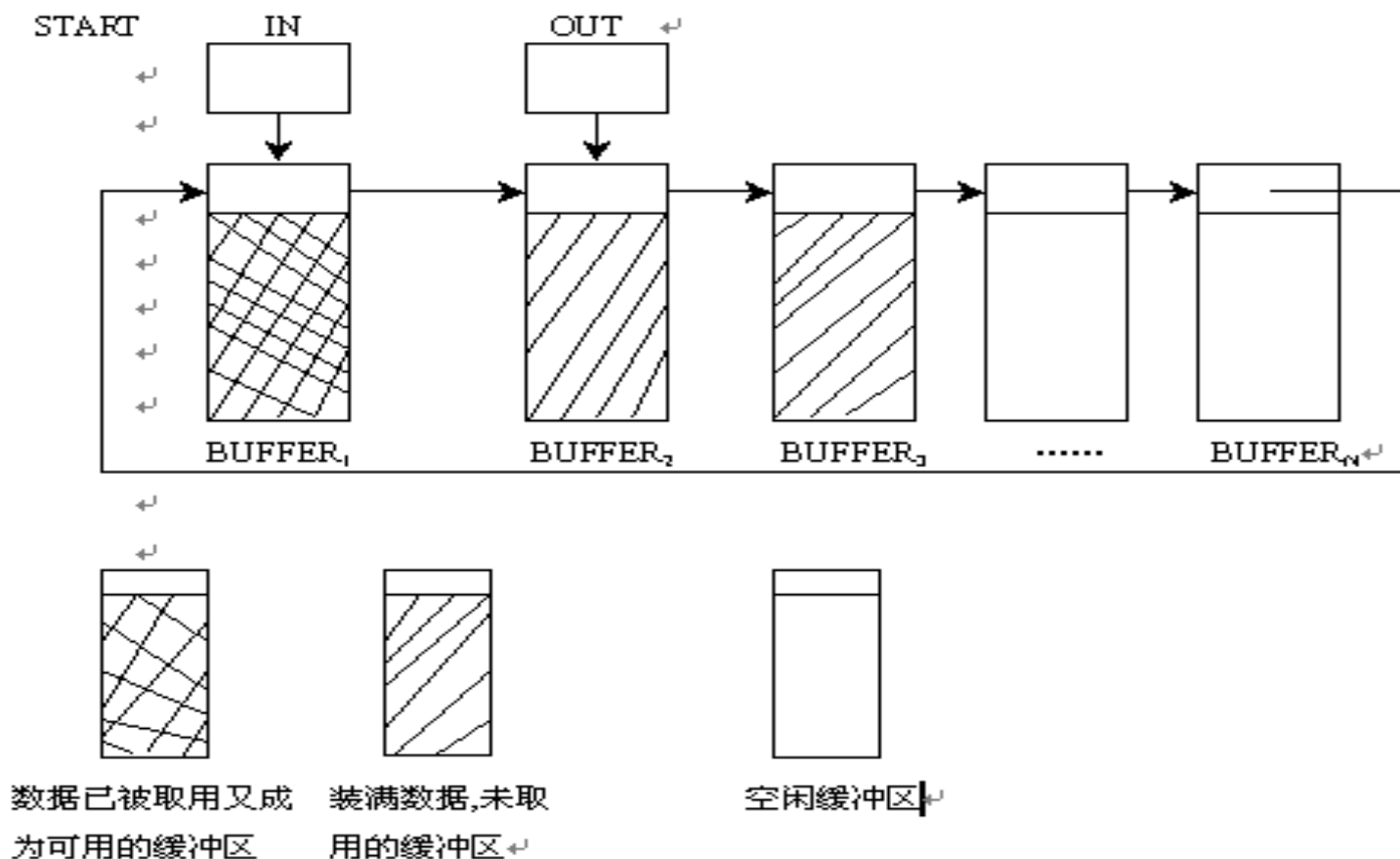
(2) 双缓冲(Double Buffer)

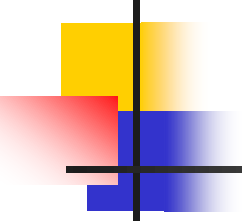
- 原理：设置两个缓冲区**buf1**和**buf2**。
- 读入数据时，输入设备向**buf1**填入数据，进程从**buf1**提取数据，在进程从**buf1**提取数据的同时，输入设备向**buf2**中填数据。
- 当**buf1**取空时，进程又从**buf2**中提取数据，与此同时输入设备向**buf1**填数。如此交替使用两个缓冲区。

双缓冲工作示意图



(3) 循环缓冲 (环形缓冲)

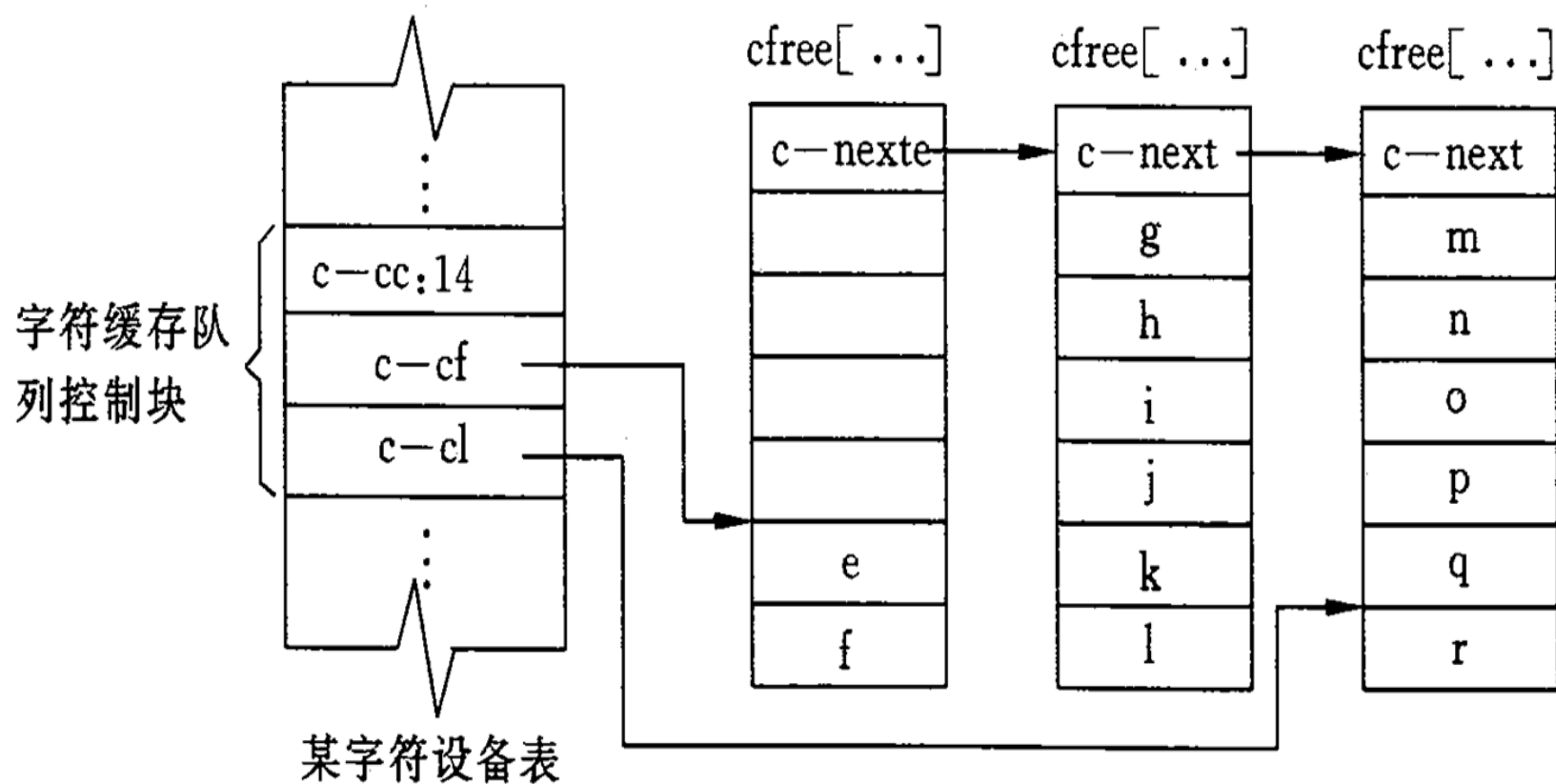


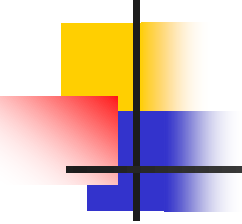


(3) 循环缓冲（续）

- IN指向读入数据的缓冲区首址，OUT指针指向装好数据且未取走的缓冲区首址。
- 系统初启时，指针初始化：使 $START=IN=OUT$ 。
- 输入信息时，信息输入到IN指向的缓冲区，当一个缓冲区装满后，IN指针指向下一个空闲缓冲区
- 当输出信息时，从由**OUT**指向的缓冲区中获取，然后将**OUT**指针指向下一个装满信息的缓冲区

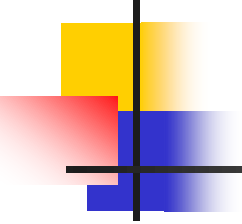
UNIX 中 I/O 字符缓存队列





(4) 缓冲池(Buffer Pool)

- 单缓冲、双缓冲和环形缓冲都为专用缓冲区，缓冲区的利用率不高。为此，引入缓冲池技术。
- 缓冲池由内存中一组大小相等的缓冲区组成，池中各缓冲区的大小与用于I/O的设备的基本信息单位相似，缓冲池属于系统资源，由系统进行管理。
- 缓冲池中各缓冲区可用于输出信息，也可用于输入信息，并可根据需要组成各种缓冲区队列。



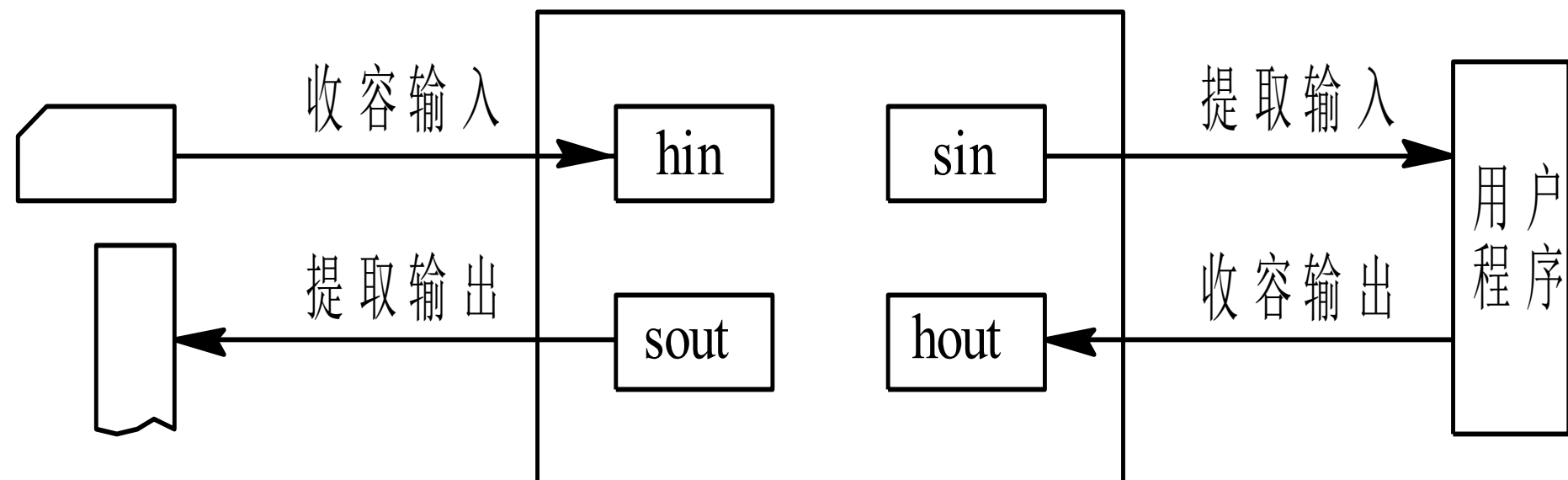
缓冲池的组成

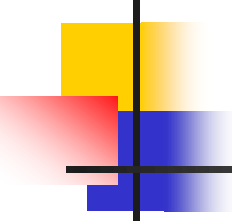
有三种类型的缓冲区：① 空缓冲区； ② 装满输入数据的缓冲区； ③ 装满输出数据的缓冲区。可形成以下三个队列：

- 1) 空缓冲队列emq。
- 2) 输入队列inq。
- 3) 输出队列outq。

缓冲区的工作方式

缓冲池

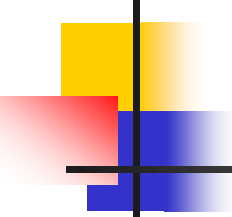




5.3.2 磁盘驱动调度技术

磁盘驱动调度是指当多个访盘请求在等待时，采用一定的策略，对这些请求的服务顺序调整安排，旨在降低平均磁盘服务时间，达到公平、高效

- 公平：一个I/O请求在有限时间内满足
- 高效：减少设备机械运动所带来的时间浪费

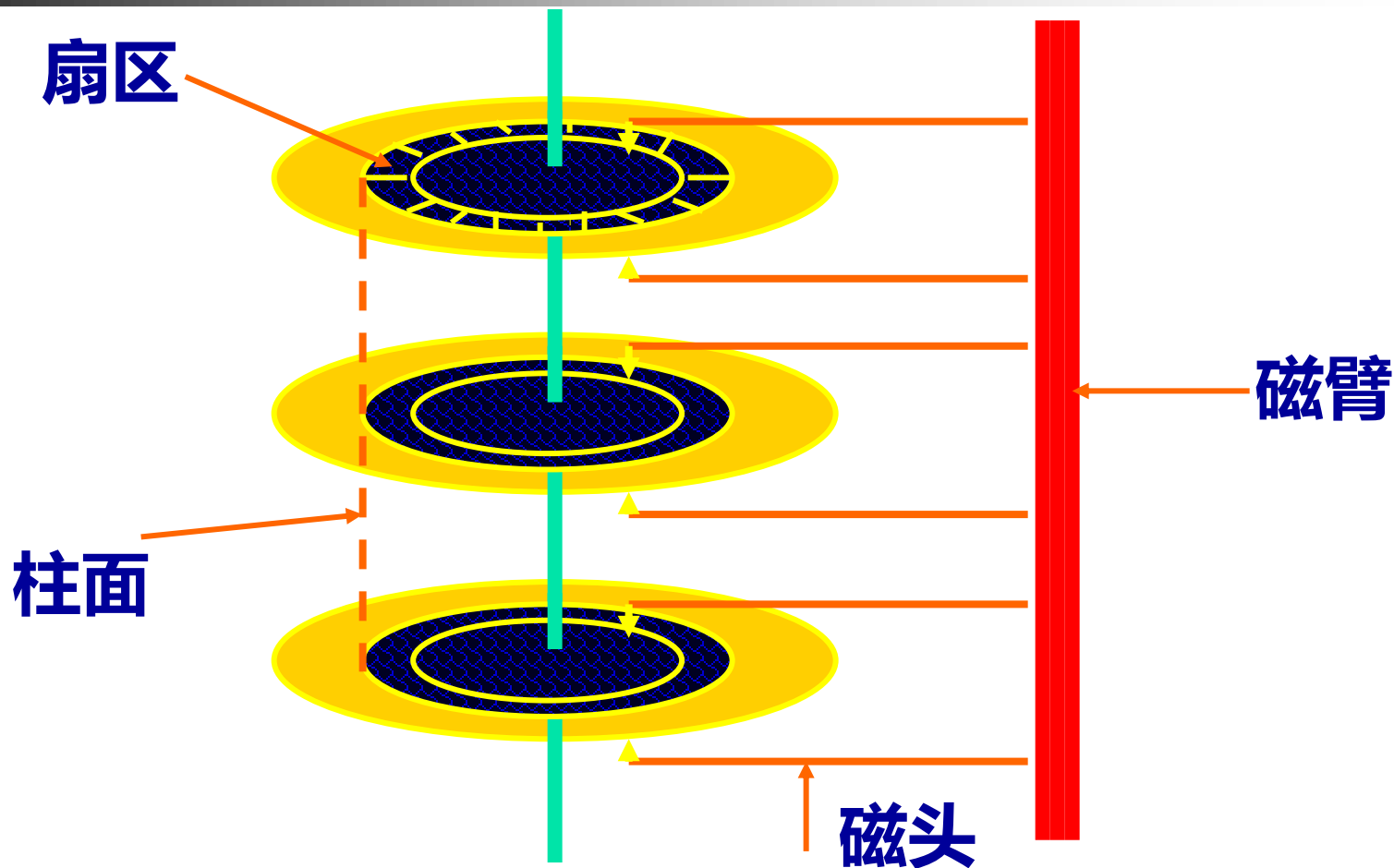


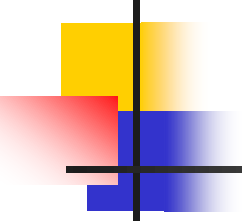
1、磁盘的结构与存取过程

磁盘主要分为硬盘和软盘。硬盘可分为：

- 固定头硬盘：每个磁道设置一个磁头，变换磁道时不需要磁头的机械移动，速度快但成本高
- 移动头硬盘：一个盘面只有一个磁头，变换磁道时需要移动磁头，速度慢但成本低

移动头硬盘的结构

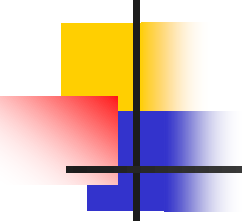




磁盘的存取过程

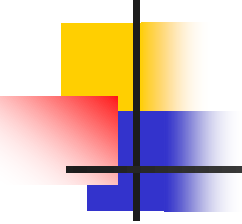
由三个动作组成：

- 寻道：磁头移动定位到指定磁道
- 旋转延迟：等待指定扇区从磁头下旋转经过
- 数据传输：数据的实际传输



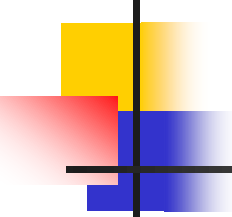
磁盘访问时间

- 寻道时间 T_s : 大约几ms到几十ms
- 旋转延迟时间 T_r : 对于7200转/分, 平均延迟时间为4.2ms
- 数据传输时间 T_t : 目前磁盘的传输速度一般有几十M/s, 传输一个扇区的时间小于0.05ms



要提高磁盘的访问速度，主要应从
以下两方面入手：

- 磁盘的调度算法
- 数据的合理组织

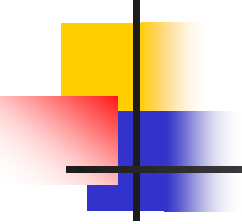


2、磁盘驱动调度算法

(1)先来先服务算法（FCFS）

按访问请求到达的先后次序服务。

- 优点：简单，公平；
- 缺点：效率不高，相邻两次请求可能会造成最内到最外的寻道过程，使磁头反复移动，增加了服务时间，对机械也不利；

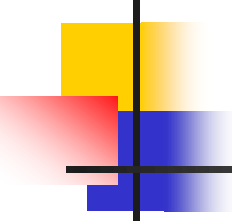


磁盘驱动调度算法（续）

(2)最短寻道时间优先（SSTF）

优先选择距当前磁头最近的访问请求进行服务，即跨越的磁道数最少的优先。

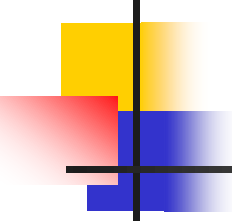
- 优点：改善了磁盘平均服务时间；
- 缺点：造成某些访问请求长期等待得不到服务



磁盘驱动调度算法（续）

(3)扫描算法（SCAN）

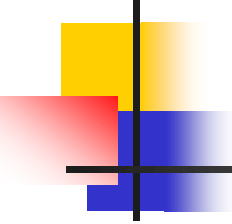
- 克服了最短寻道优先的缺点，既考虑了距离，同时又考虑了方向。
- 具体做法：依次为与当前磁头移动方向一致的访问请求服务，当该方向上已没有访问请求时，则改变移动方向，并为经过的访问请求服务，如此反复。
又称为电梯调度算法（Elevator）



磁盘驱动调度算法（续）

(4)单向扫描算法(SSCAN)

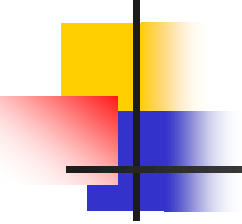
扫描时按磁头移动方向进行，直到最外（内）层上的请求，然后立刻返回至有请求的最内（外）磁道上，重新进行扫描，依次反复。



磁盘驱动调度算法（续）

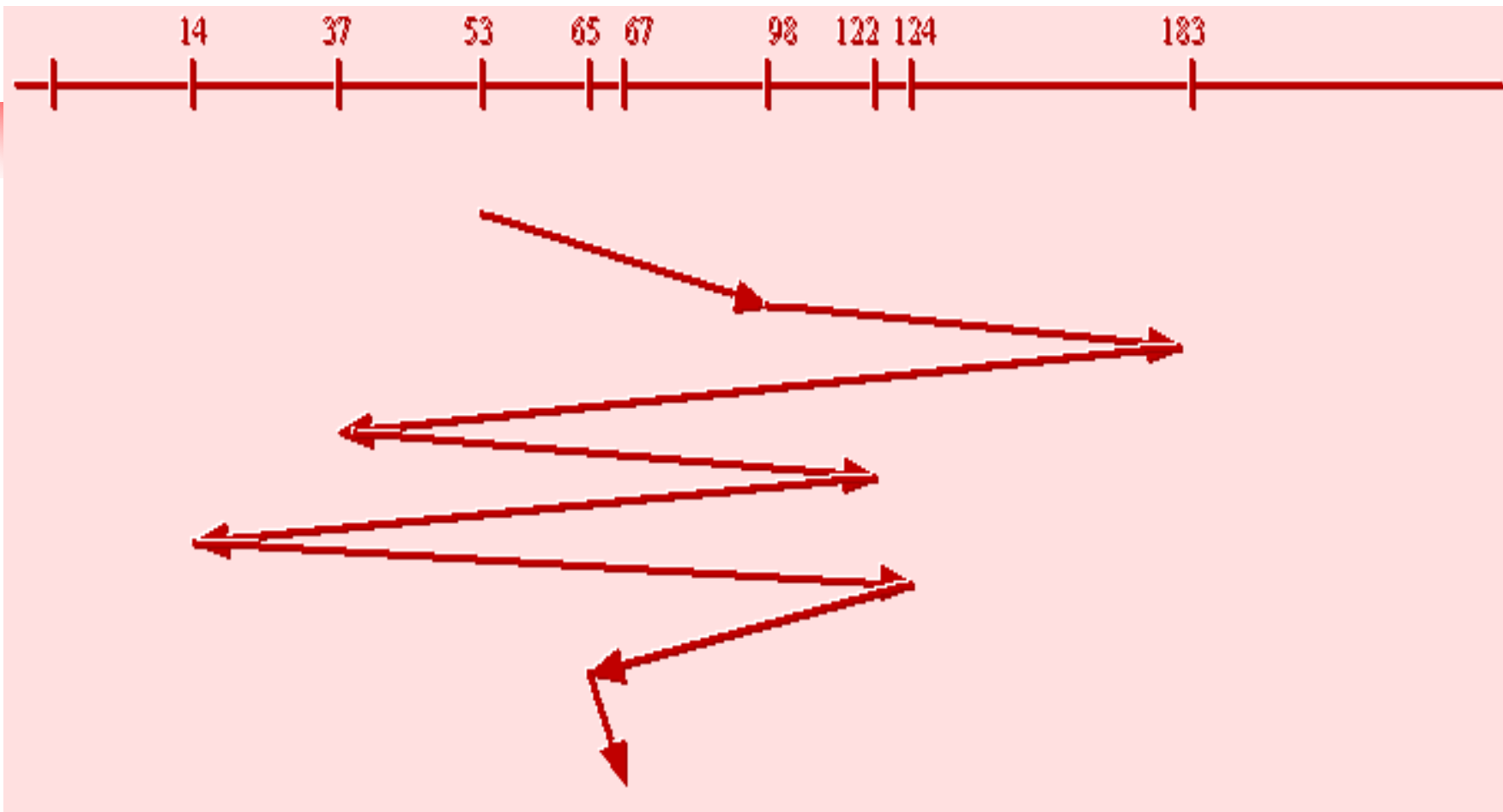
(5)循环扫描算法(CSCAN)

扫描时一律从0道开始，直到最大磁道号，然后立刻返回至0，重新进行扫描，依次反复。



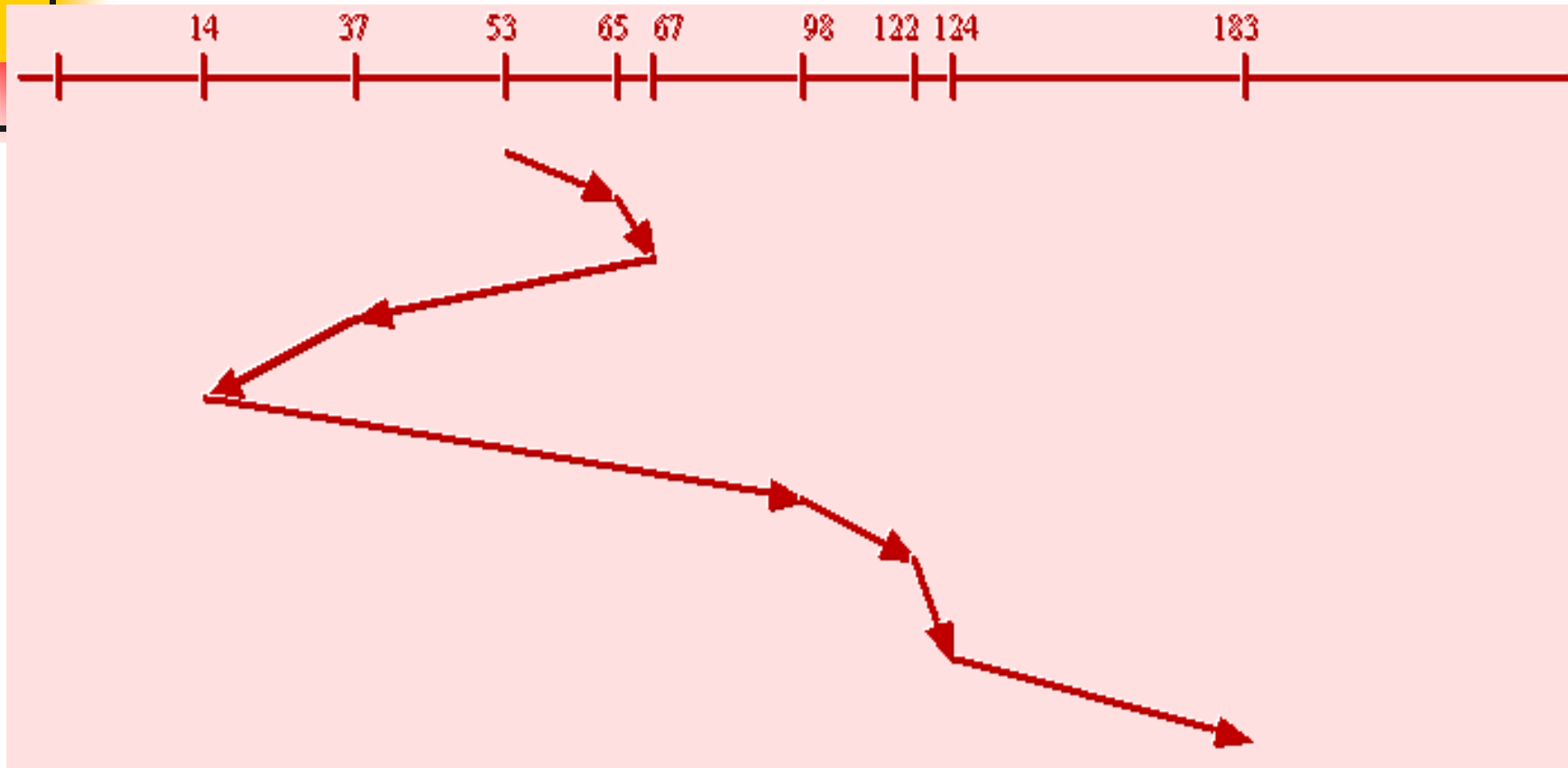
【例】 某磁盘共有184个磁道(0-183)，现有磁盘访问序列：98，183，37，122，14，124，65，67。在访问了64磁道后，磁头目前正在53磁道上访问。当分别采用FCFS、SSTF、SCAN算法时磁头的服务序列以及磁头移动总距离（道数）。

(1) 采用FCFS



顺序(53), 98, 183, 37, 122, 14, 124, 65, 67
磁头走过的总道数: $|53-98|+|98-183|+\dots+|65-67|$

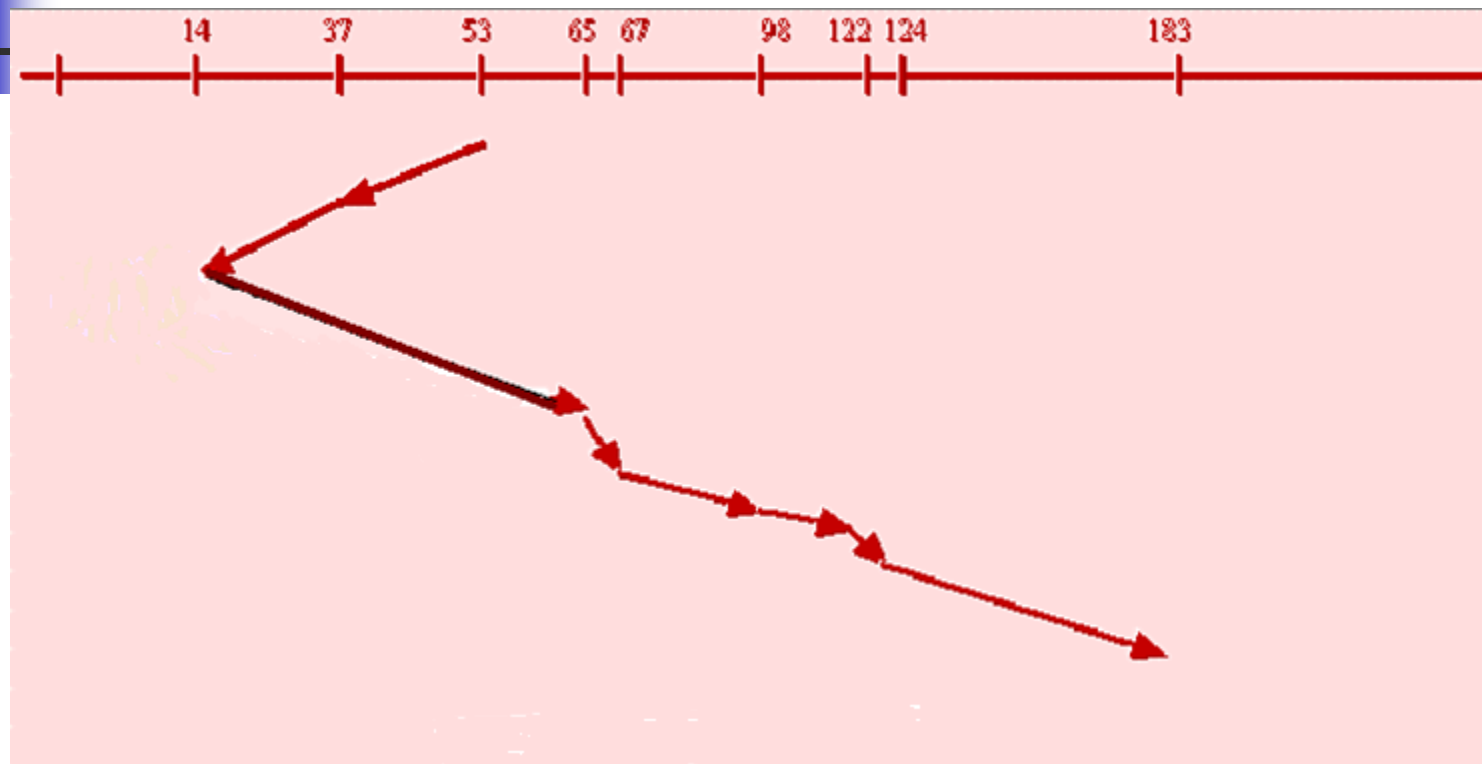
(2) 采用SSTF



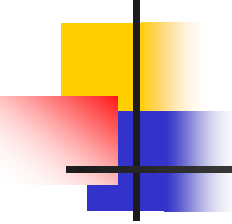
顺序(53),65, 67 , 37, 14, 98, 122, 124, 183

磁头走过的总道数: $|53-65|+|65-67|+\dots+|124-183|=$

(3) 采用SCAN

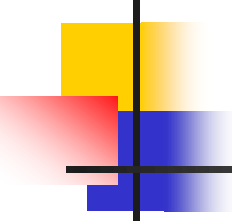


顺序: (53), 37, 14, 65, 67, 98, 122, 124, 183
磁头走过的总道数 $|53-37| + |37-14| + \dots + |124-183| =$



磁盘调度算法的选择

- 实际系统相当普遍采用最短寻道时间优先算法，因为它简单有效，性价比好。
 - 扫描算法杜绝饥饿，性能适中。
 - 循环扫描算法适应不断有大批量柱面均匀分布的I/O请求，且磁道上存放记录数量较大的情况。
- 一般要将磁盘调度算法作为操作系统的单独模块编写，利于修改和更换。

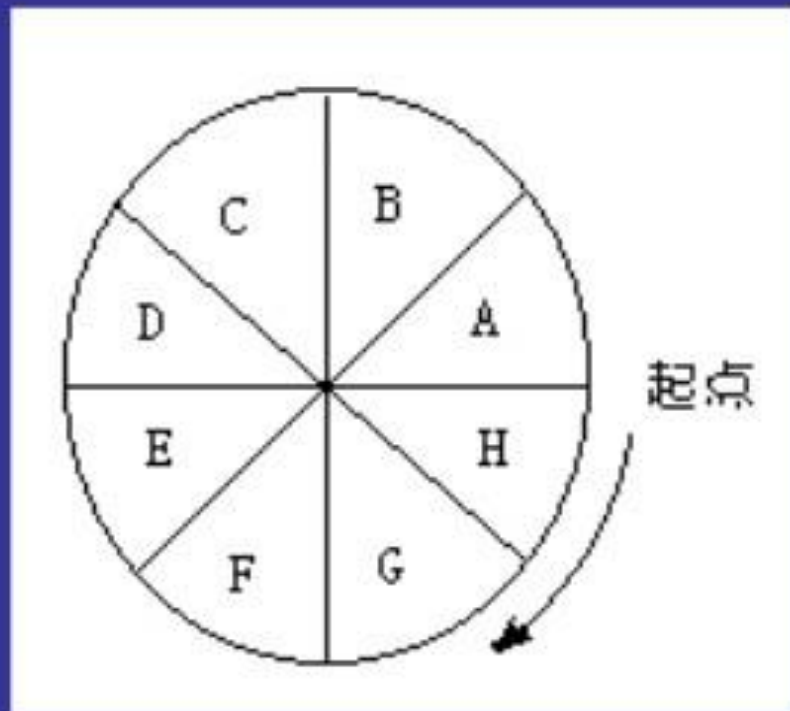


磁盘数据的组织问题（优化）

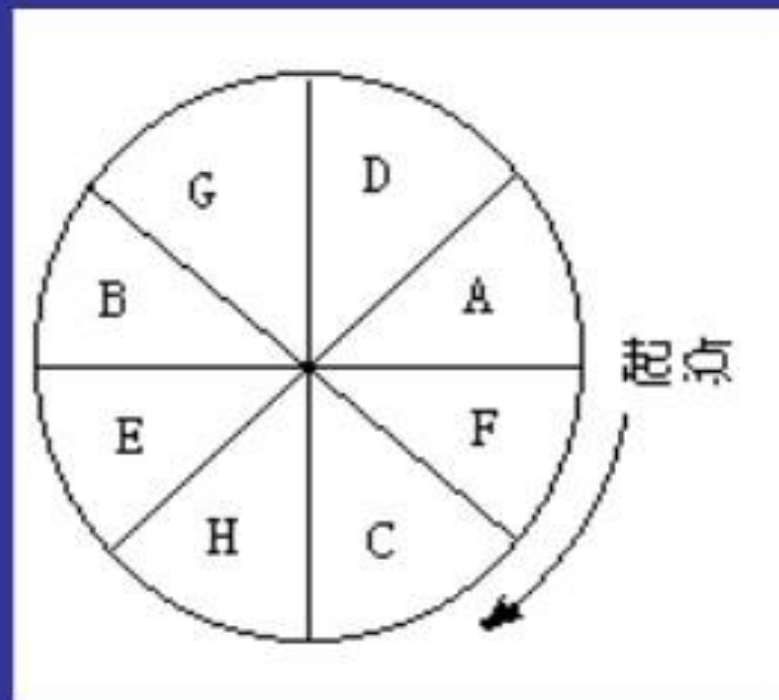
【例】 一条磁道上分为**8**个扇区，存放**8**条逻辑记录，现要对这些记录进行读出处理。若磁盘转速为**20**毫秒/圈，每条记录读出后的处理时间为**5**毫秒，问应该怎样组织逻辑记录，才能使整个处理时间最少？

【分析】

- 如果按顺序存放时，读第一条记录要花**2.5毫秒**的时间，加上处理的**5毫秒**后，当前的读写磁头已经定位在第**4**条记录上。
- 为处理第**2**条记录，必须等待磁盘把第**2**条记录旋转到磁头下方，要**15毫秒**的延迟时间。
- 处理所有**8**条记录所需时间：
$$8 * (2.5 + 5) + 7 * 15 = 165 \text{ (毫秒)}$$

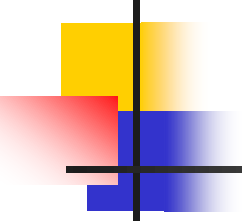


(a)



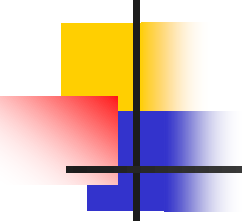
(b)

如果我们把这8条逻辑记录在磁道上的位置重新安排一下，(b)是这8条逻辑记录的最优分布。处理这8个记录所要花费的时间为： $8 \times (2.5 + 5) = 60(\text{ms})$ 。可见记录的优化分布有利于减少延迟时间，从而缩短了输入输出操作的时间。



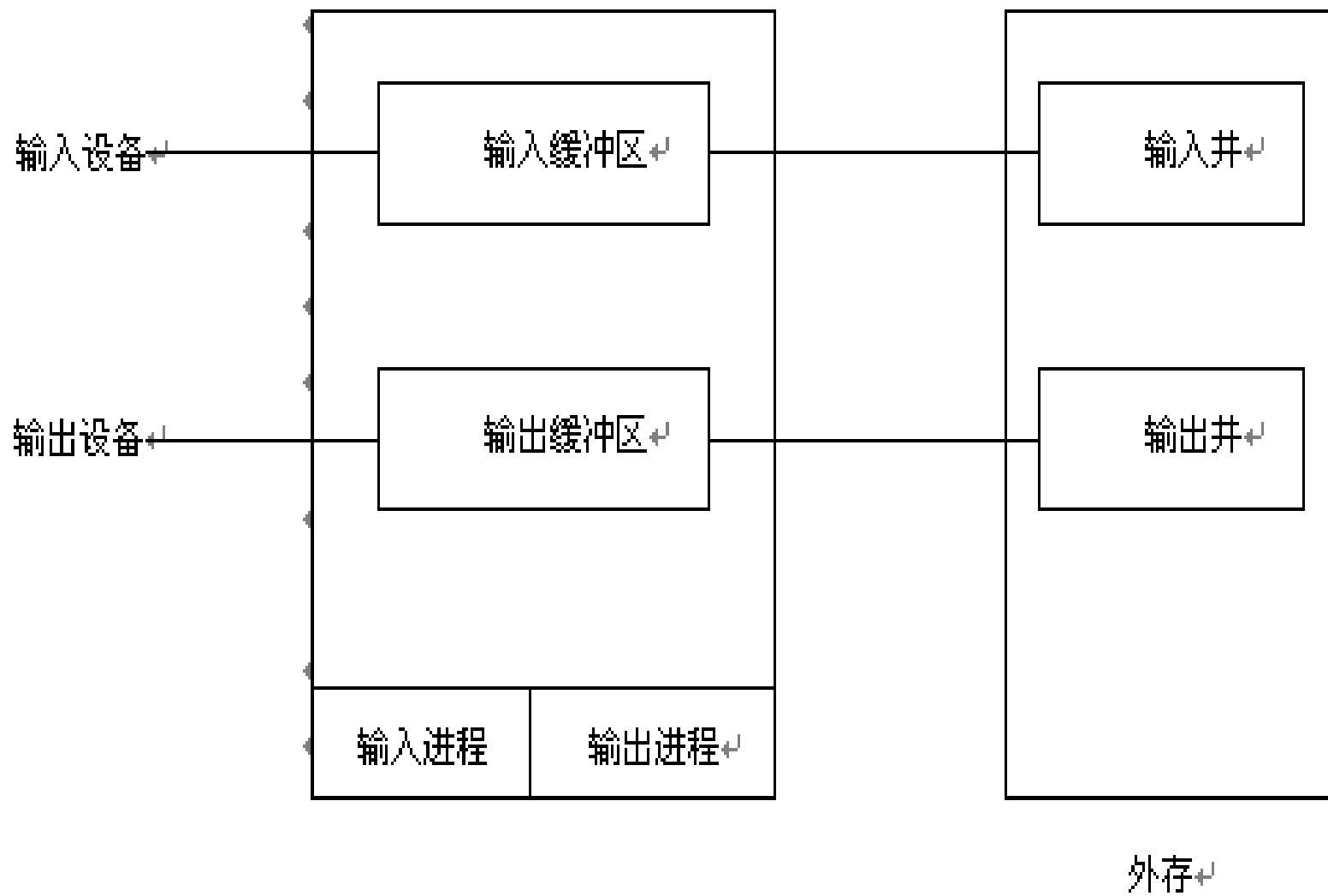
5.3.3 SPOOLing技术

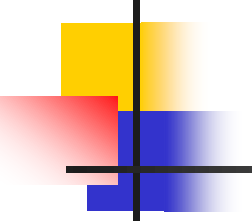
- SPOOLing(Simultaneous Peripheral Operations On-Line)外部设备同时联机操作
- 在单道批处理时期，用脱机**I/O**可以提高**CPU**利用率。多道系统出现后可以利用一道程序来模拟脱机**I/O**中的外围机，这样可实现主机控制下的脱机**I/O**功能。
- 把这种在联机情况下实现的同時外围操作称为**SPOOLing**，也称为**假脱机操作**。



1、SPOOLing系统的组成

- 输入井和输出井
- 输入缓冲区和输出缓冲区
- 输入进程和输出进程





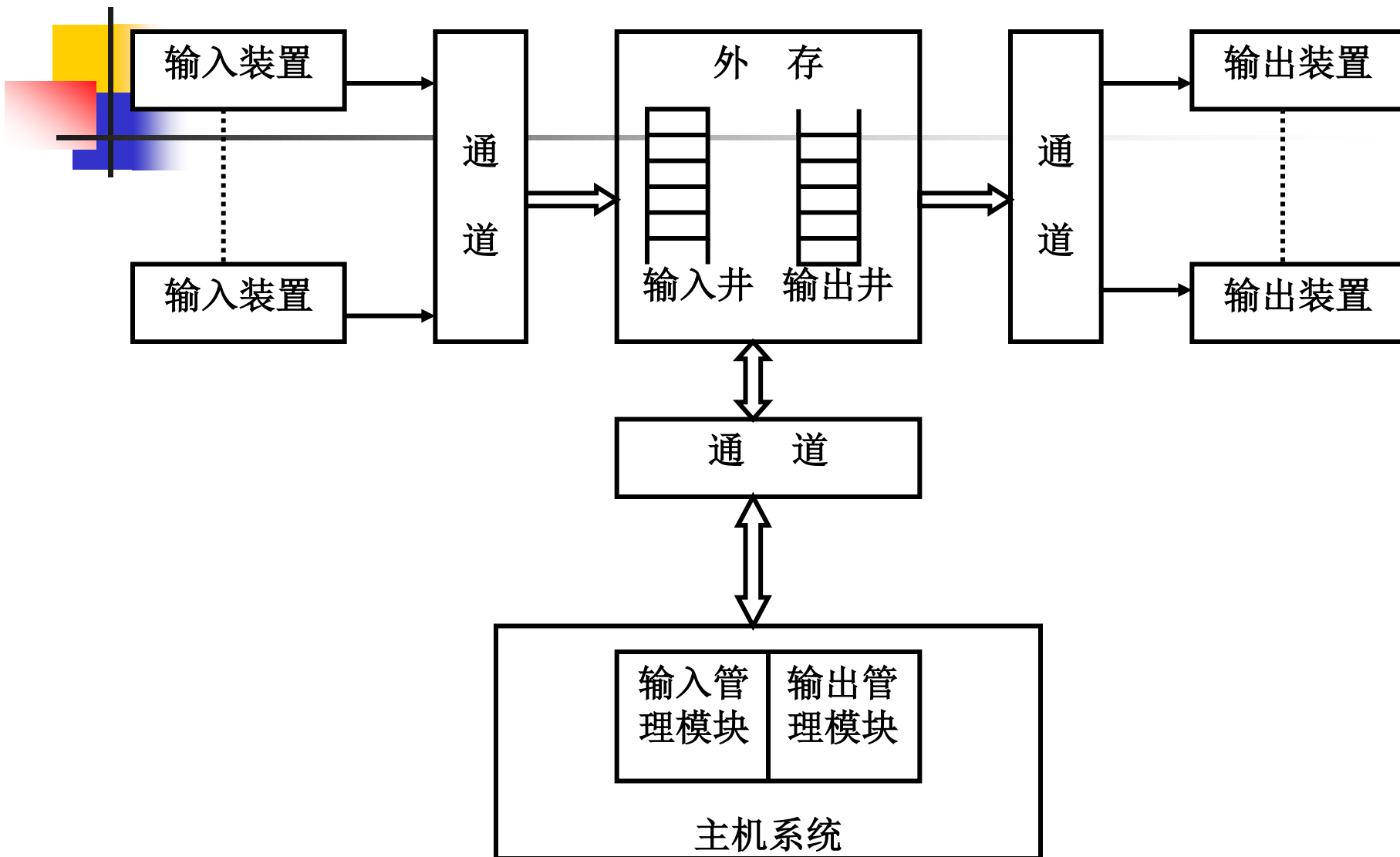
2、SPOOLing系统工作原理

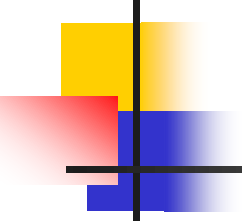
- 作业执行前预先将程序和数据输入到输入井中
- 作业运行后，使用数据时，从输入井中取出
- 作业执行不必直接启动外设输出数据，只需将这些数据写入输出井中
- 作业全部运行完毕，再由外设输出全部数据和信息

【好处】

- 实现了对作业输入、组织调度和输出的统一管理
- 外设与**CPU**并行工作（假脱机）

SPOOLing系统工作原理



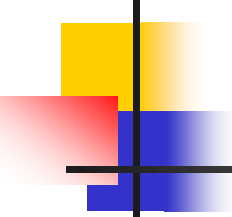


3、共享打印机

当用户进程请求打印输出时，SPOOLing系统同意其请求，但不真正打印，而只为它做两件事：

- 由输出进程把要打印的数据送入输出井中；
- 输出进程申请并填写用户请求打印表，并将其挂到请求打印队列上。

若打印机空闲，才从输出井取出数据进行真正打印。



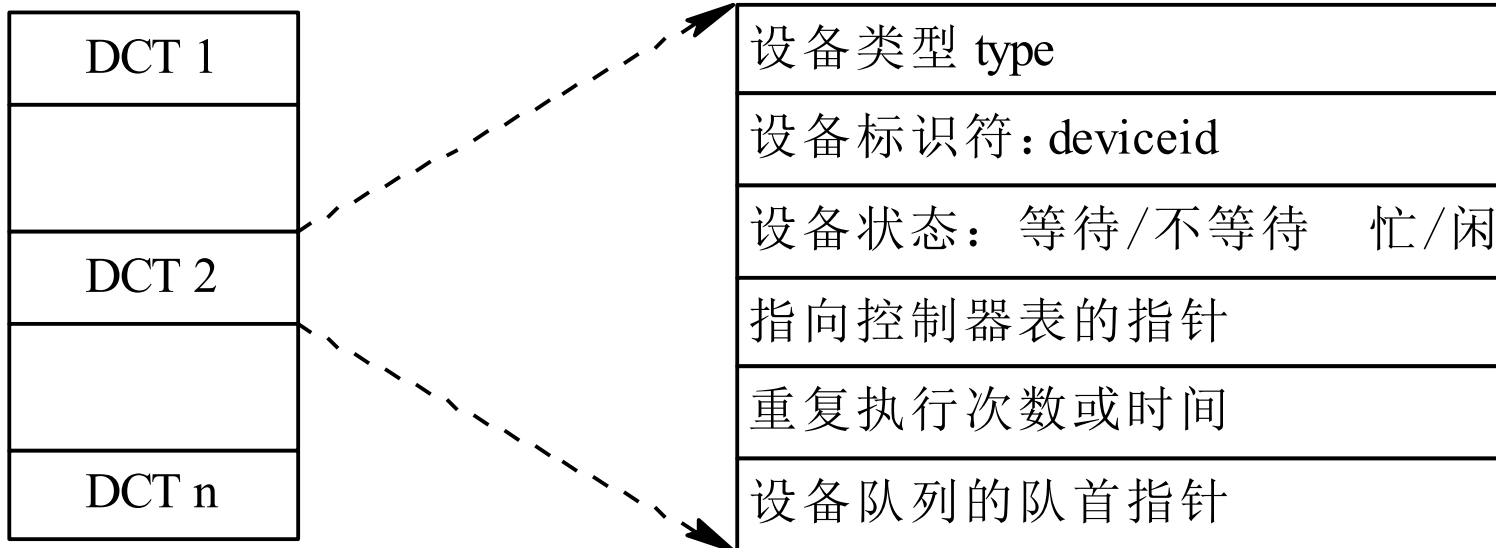
5.4 设备分配

5.4.1 设备分配中的数据结构

1. 设备控制表DCT
2. 控制器控制表COCT
3. 通道控制表CHCT
4. 系统设备表 SDT

设备控制表DCT

设备控制表集合



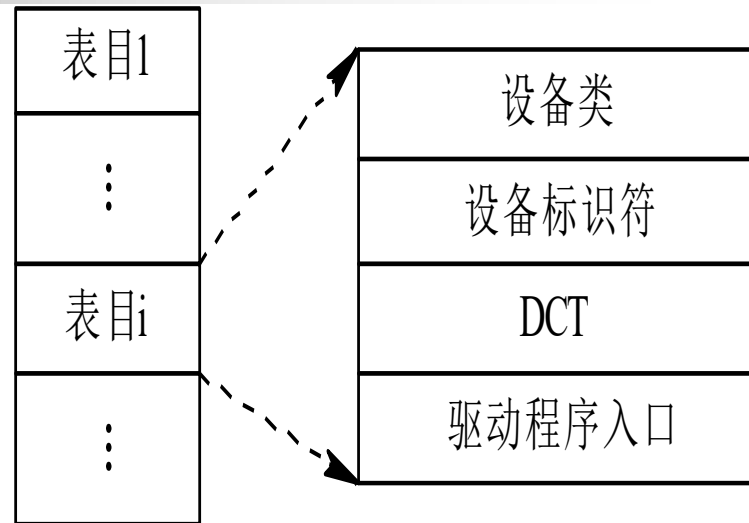
2. 控制器控制表、 通道控制表和系统设备表

控制器标识符: controllerid
控制器状态: 忙/闲
与控制器连接的通道表指针
控制器队列的队首指针
控制器队列的队尾指针

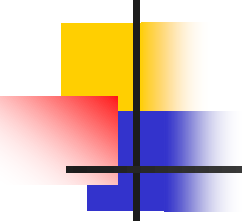
(a) 控制器表 COCT

通道标识符: channelid
通道状态: 忙/闲
与通道连接的控制器表首址
通道队列的队首指针
通道队列的队尾指针

(b) 通道表 CHCT



(c) 系统设备表 SDT



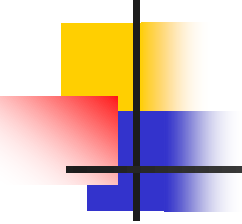
5.4.2 设备分配策略

1. 按设备的固有属性分配

(1) 独享分配

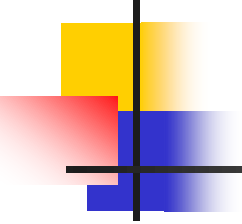
(2) 共享分配

(3) 虚拟分配



●独享分配

所谓独享设备是指这类设备被分配给一个作业后，被这个作业所独占使用，其他的任何作业不能使用，直到该作业释放该设备为止。常见的独享设备有打印机、光电输入机等。针对独享设备，系统一般采用静态分配方式。

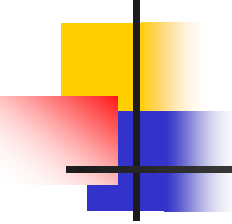


●共享分配

设备的共享有两层含义：

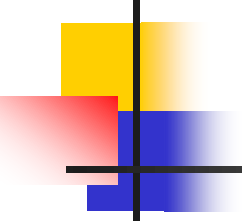
- 指对设备介质的共享，如磁盘上的各扇区。
- 指对磁盘等驱动器的共享，多个用户访问这些设备上的信息是通过驱动器来实现的。

对共享设备的分配一般采用**动态分配**这一方式。



● 虚拟分配

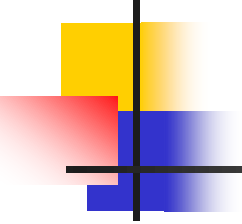
对虚拟设备采用的是虚拟分配，其过程是：当进程中请求独享设备时，系统将共享设备的一部分存储空间分配给它。进程与设备交换信息时，系统把要交换的信息存放在这部分存储空间，在适当的时候对信息作相应的处理。



2. 设备分配算法

(1) 先来先服务

(2) 优先级高者优先



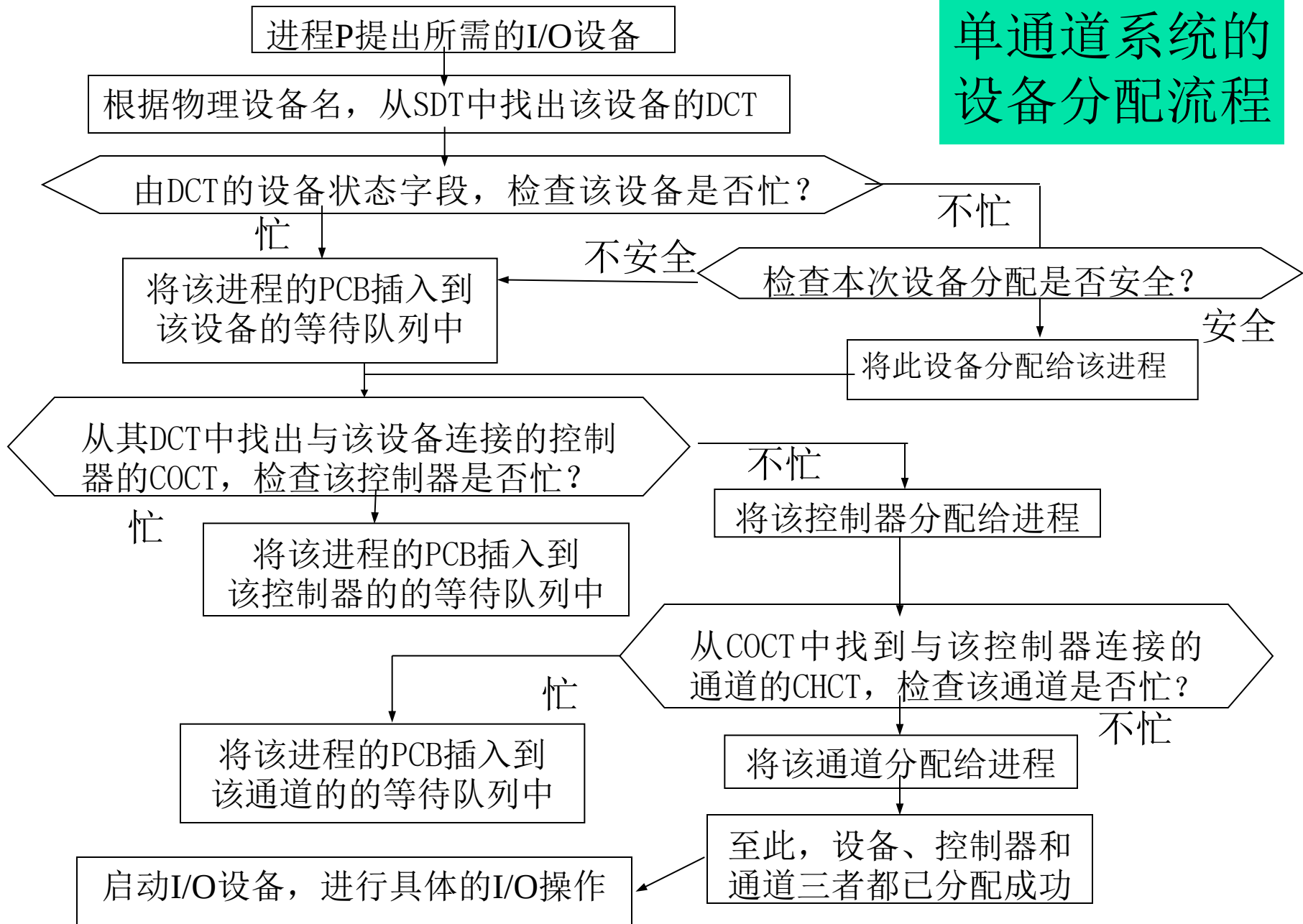
3. 设备分配的过程

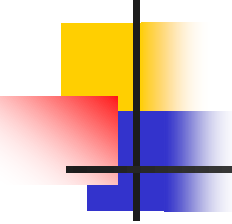
(1)分配设备

(2)分配控制器

(3)分配通道

单通道系统的设备分配流程



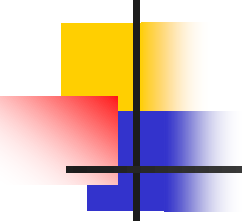


4. 设备分配中的安全性

(1) 静态分配：不会出现死锁

(2) 动态分配：

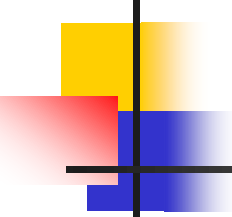
- 单请求方式：不会出现死锁
- 多请求方式：可能会出现死锁



5.4.3 设备独立性

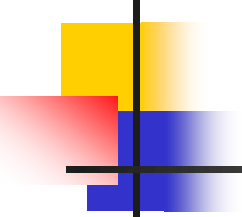
1、设备独立性的概念

也称设备无关性。它是指用户在编制程序时，使用的逻辑设备名与具体的物理设备无关。即用户能独立于具体物理设备而方便的使用设备。



2. 引入设备独立性的好处

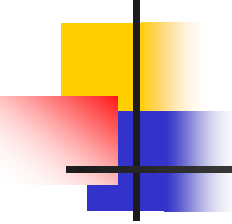
- (1) 方便用户编程
- (2) 便于程序的移植
- (3) 提高资源利用率
- (4) 适应多进程多用户需要



3. 设备独立性的实现

为了实现设备独立性，系统须具有将逻辑设备名转换为某物理设备名的功能，这类似于内存管理中逻辑地址到物理地址的转换。

具体通过**逻辑设备表**实现



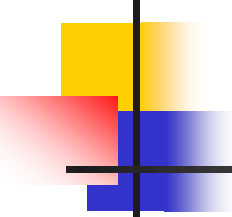
逻辑设备表 (LUT)

逻辑设备名	物理设备名	驱动程序 入口地址
/dev/tty	3	1024
/dev/printer	5	2046
⋮	⋮	⋮

(a)

逻辑设备名	系统设备表指针
/dev/tty	3
/dev/printer	5
⋮	

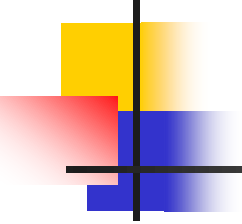
(b)



5.5 I/O软件

I/O软件的层次

- 用户级软件
- 设备无关软件
- 设备驱动程序
- 中断处理程序

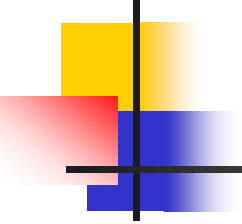


5.5.1 用户级软件

用户级I/O软件是指运行在用户层的I/O库函数或代码。例如，在C语言中，标准I/O函数库包含相当多的涉及I/O的库函数：

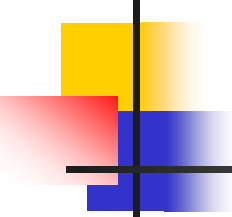
如**printf ()**、**fwrite(buffer,size,count,fp)**等；

还有其它高级语言中的输入输出函数等。它们都作为用户程序的一部分运行。



5.5.2 设备无关软件

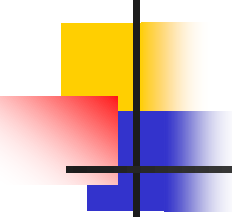
设备无关软件是指独立于每个设备之外的公共**I/O**软件，它包括所有设备驱动程序要调用的高级**I/O**原语指针（如**open**、**close**、**read**、**write**等）



设备无关软件（续）

设备无关软件的功能如下：

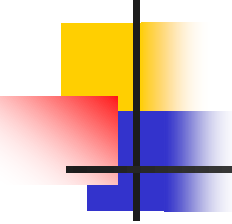
- 提供与设备驱动程序的统一接口
- 设备命名和设备保护
- 提供独立于设备的块大小
- 缓冲区管理和块设备的存储分配
- 独占性外围设备的分配和释放
- 设备错误报告



5.5.3 设备驱动程序

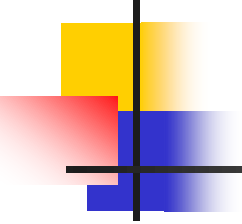
1. 设备驱动程序的功能

- (1) 向控制器发出控制命令，并监督执行
- (2) 对等待设备、控制器和通道的进程进行排队，对进程阻塞（或挂起）、唤醒等操作进行处理。
- (3) 执行确定的缓冲区策略
- (4) 进行比寄存器级别更高的一些特殊处理，如代码转换、ESC处理等



2、设备驱动程序的处理过程

- 将抽象要求转换为具体要求
- 检查I/O请求的合法性
- 读出和检查设备的状态，确保设备处于就绪态
- 传送必要的参数
- 工作方式的设置
- 启动I/O设备，并检查启动是否成功

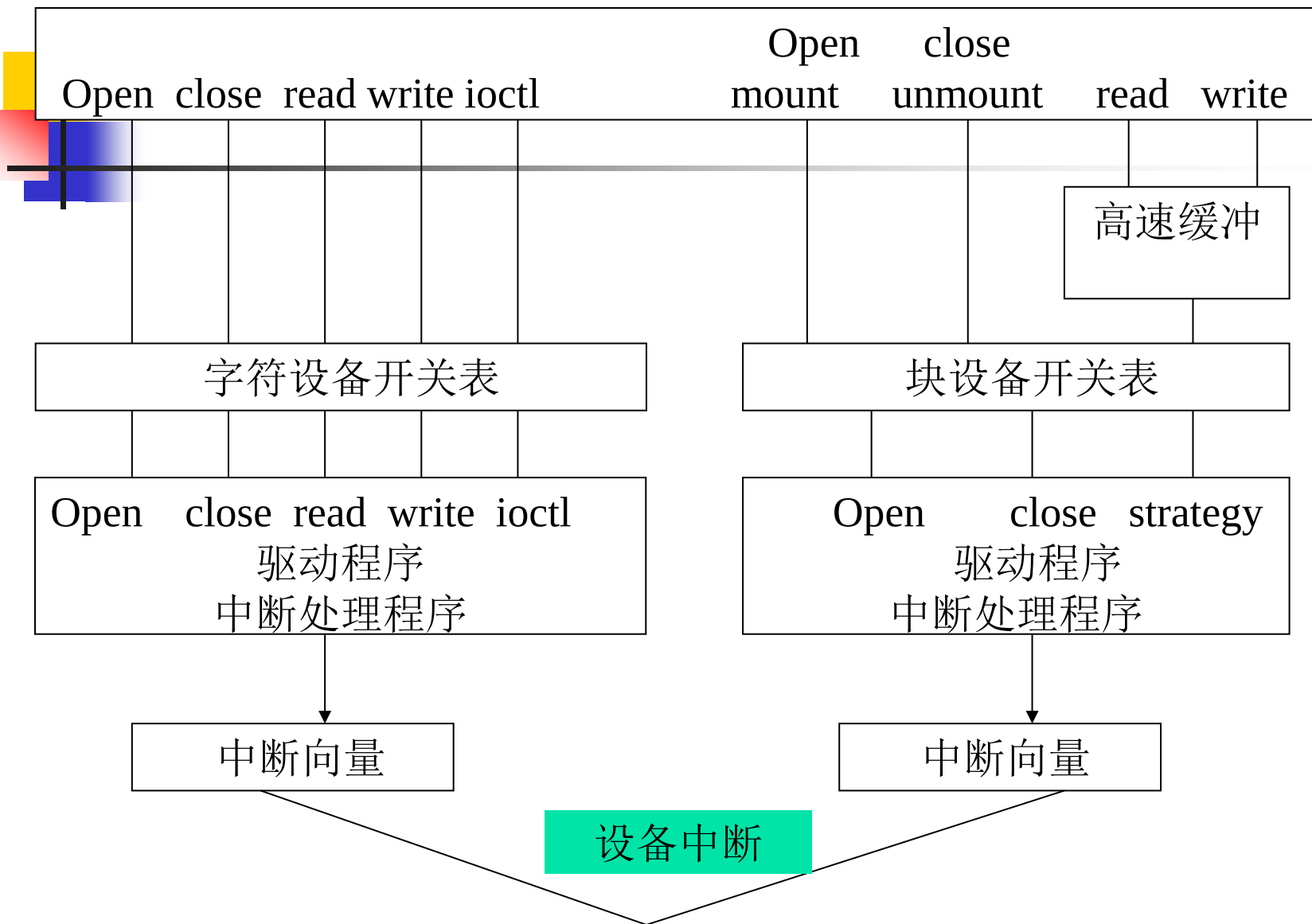


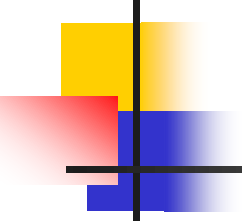
3、驱动程序与内核的接口

通过**设备开关表**实现，有两类设备开关表：字符设备开关表和块设备开关表。

块 设 备 开 关 表			
类型	open	read	...
0	gdopen	gdread	...
1	gtopen	gtread	...

设备无关软件

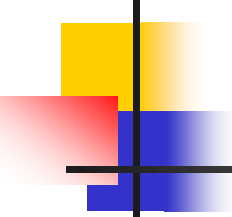




5.5.4 中断处理程序

中断处理程序也叫中断服务程序，它是在产生设备中断后处理的。通常在I/O操作完成后，由设备控制器向CPU发出中断请求的。其处理过程如下：

- 唤醒被阻塞的驱动（程序）进程
- 保护被中断进程的CPU环境
- 转入相应的中断处理程序
- 中断处理
- 恢复被中断进程的现场

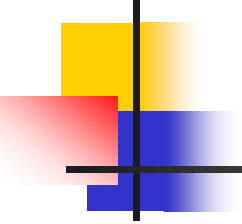


5.6 Linux的设备管理

5.6.1 Linux设备管理概述

1、设备分类

- 块设备：把信息存储在可寻址的固定大小的数据块中，数据块均可以被独立地读写，建立块缓冲，能随机访问数据块。
- 字符设备：可以发送或接收字符流，通常无法编址，也不存在任何寻址操作。

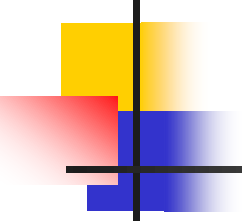


2、设备文件

- 硬件设备均当作特殊设备文件处理，如第一硬盘为/dev/hda，第一块以太网卡为/dev/eth0。
- 设备文件通过mknod系统调用创建：

```
mknod(const char*filename, int mode, dev_t dev)
```

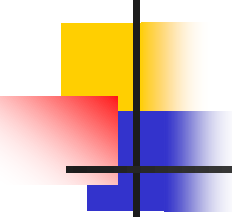
其中，dev的高8位保存主设备号，低8位保存次设备号。



5.6.2 Linux设备驱动程序

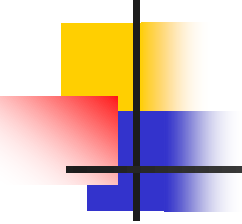
1、DDI/DKI规范

- 驱动程序与内核的接口: file_operations
- 驱动程序与系统引导的接口: 设备的初始化
- 驱动程序与设备的接口: 驱动程序如何与设备交互



2、设备驱动程序代码组成

- 驱动程序的注册与注销
- 设备的打开与释放
- 设备的读写操作
- 设备的控制操作
- 设备的中断和查询处理

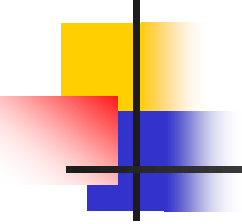


3、设备驱动程序的注册

以字符设备为例，用于注册字符设备的数据结构为 `device_struct`。其定义如下：

```
struct device_struct
{ const char *name;
  struct file_operations *fops;};
```

所有的 `device_struct` 描述符都在数组 `chrdevs[MAX_CHRDEV]` 中。



```
static struct device_struct  
chrdevs[MAX_CHRDEV];
```

注册字符设备驱动程序:

```
int register_chrdev(unsigned int major,  
const char *name, struct file_operations  
*fops);
```

其中, major是为设备驱动程序向系统申请的主设备号, 如果为0则系统为该设备驱动程序动态地分配一个主设备号, name是设备名。fops是对各个调用的入口点的说明。

5.6.3 简单设备驱动程序举例

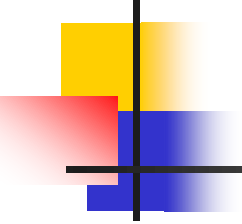
```
#include <linux/module.h>
#include <linux/types.h>
#include <linux/config.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/errno.h>
#include <asm/segment.h>
#include <asm/uaccess.h>
unsigned int dev_major = 0;  /*定义主设备号为0，由
系统分配*/
/*定义与file_operations的各个函数指针对应的功能函数*/
```



/*读设备功能函数*/

```
static int read_dev(struct inode *node, struct
    file *file, char *buf, int count)
    {
        int left;
        /*在向用户空间拷贝数据之前，必须使用函数
        verify_area 来验证 buf 是否可用 */

        if (verify_area(VERIFY_WRITE, buf, count) == -
            EFAULT
                return -EFAULT;
        for(left = count ; left > 0 ; left--)
            {
```



/*将内核空间的数据复制到用户空间，把用户的缓冲区全部写1*/

/*__put_user()是内核提供的一个函数，用于向用户传送数据*/

```
__put_user(1, buf); buf++;}  
return count;}
```

/*写设备功能函数*/

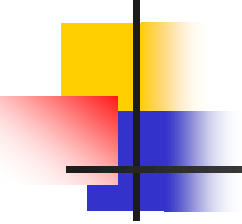
```
static int write_dev(struct inode *inode,  
struct file *file, const char *buf, int count)  
{  
    return count;
```

/*打开设备功能函数*/

```
static int open_dev(struct inode *inode, struct file
*file )
{
MOD_INC_USE_COUNT;  /*增加模块的使用计数*/
return 0;
}
```

/*释放设备功能函数*/

```
static void release_dev(struct inode *inode, struct
file *file )
{
MOD_DEC_USE_COUNT;  /*减少模块的使用计数*/
return 0;
}
```

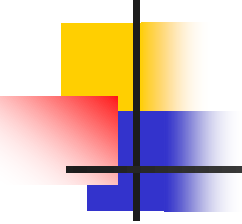


/*通过file_operations结构传递相应的函数指针*/

```
struct file_operations dev_fops = {  
    read: read_dev,  
    write: write_dev,  
    open: open_dev,  
    release: release_dev  
};
```




```
int init_module(void) /*注册模块*/
{ int tag;
  tag=register_chrdev(0, " mydev",
    &dev_fops);
  if (tag<0)
  {   printk(KERN_INFO "mydev: can't get
major number \n");
    return tag;}
  if (dev_major==0) dev_major=tag;
  return 0;
}
```



```
int cleanup_module() /*注销模块*/  
{  
    int re;
```

```
    re=unregister_chrdev(dev_major, " mydev"  
);  
    if( re<0)  
    { printk("erro unregister the  
module !!\n");  
        return 1; }  
    return 0;
```

```
struct file_operations {  
int (*open) (struct inode *, struct file *);  
int (*release) (struct inode *, struct file *);  
int (*read) (struct inode *, struct file *, char , int);  
int (*write) (struct inode *, struct file *, off_t , int);  
int (*readdir) (struct inode *, struct file *, struct dirent  
* , int);  
int (*select) (struct inode *, struct file *, int ,  
select_table *);  
int (*ioctl) (struct inode *, struct file *, unsigned int ,  
unsigned long);  
int (*mmap) (struct inode *, struct file *, struct  
vm_area_struct *);  
int (*seek) (struct inode *, struct file *, off_t , int);  
int (*fsync) (struct inode *, struct file *);  
int (*fasync) (struct inode *, struct file *, int);  
int (*check_media_change) (struct inode * , struct file *);  
int (*revalidate) (dev_t dev);
```