

# 第六章 输入输出系统

6.1 I/O系统的功能、模型和接口

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





## 6.1 I/O系统的功能、模型和接口

I/O系统管理的主要对象是I/O设备和相应的设备控制器。

其最主要的任务是：

完成用户提出的I/O请求，提高I/O速率，以及提高设备的利用率，并能为更高层的进程方便地使用这些设备提供手段。



## 6.1.1 I/O系统的基本功能

### 1. 隐藏物理设备的细节

I/O设备的类型非常多，且彼此间在多方面都有差异，诸如它们接收和产生数据的速度，传输方向、粒度、数据的表示形式及可靠性等方面。

## 2. 与设备的无关性

隐藏物理设备的细节，在早期的OS中就已实现，它可方便用户对设备的使用。与设备的无关性是在较晚时才实现的，这是在隐藏物理设备细节的基础上实现的。



### 3. 提高处理机和I/O设备的利用率

尽可能地让处理机和I/O设备并行操作，以提高它们的利用率。

为此，一方面要求处理机能快速响应用户的I/O请求，使I/O设备尽快地运行起来；另一方面也应尽量减少在每个I/O设备运行时处理机的干预时间。

#### 4. 对I/O设备进行控制

对I/O设备进行控制是驱动程序的功能。目前对I/O设备有四种控制方式：



- ① 采用轮询的可编程I/O方式；
- ② 采用中断的可编程I/O方式；
- ③ 直接存储器访问方式；
- ④ I/O通道方式。

## 5. 确保对设备的正确共享

从设备的共享属性上，可将系统中的设备分为如下两类：

(1) 独占设备：进程应**互斥地访问**这类设备，即系统一旦把这类设备分配给了某进程后，便由该进程独占，直至用完释放。典型的独占设备有**打印机、磁带机**等。系统在对独占设备进行分配时，还应考虑到分配的安全性。

(2) 共享设备：是指在一段时间内**允许多个进程**同时访问的设备。典型的共享设备是**磁盘**，当有多个进程需对磁盘执行读、写操作时，可以交叉进行，不会影响到读、写的正确性。



## 6. 错误处理

大多数的设备都包括了较多的机械和电气部分，运行时容易出现错误和故障。从处理的角度，可将错误分为临时性错误和持久性错误。对于临时性错误，可通过重试操作来纠正，只有在发生了持久性错误时，才需要向上层报告。



## 6.1.2 I/O系统的层次结构和模型

### 1. I/O软件的层次结构

通常把I/O 软件组织成四个层次，如图6-1所示。

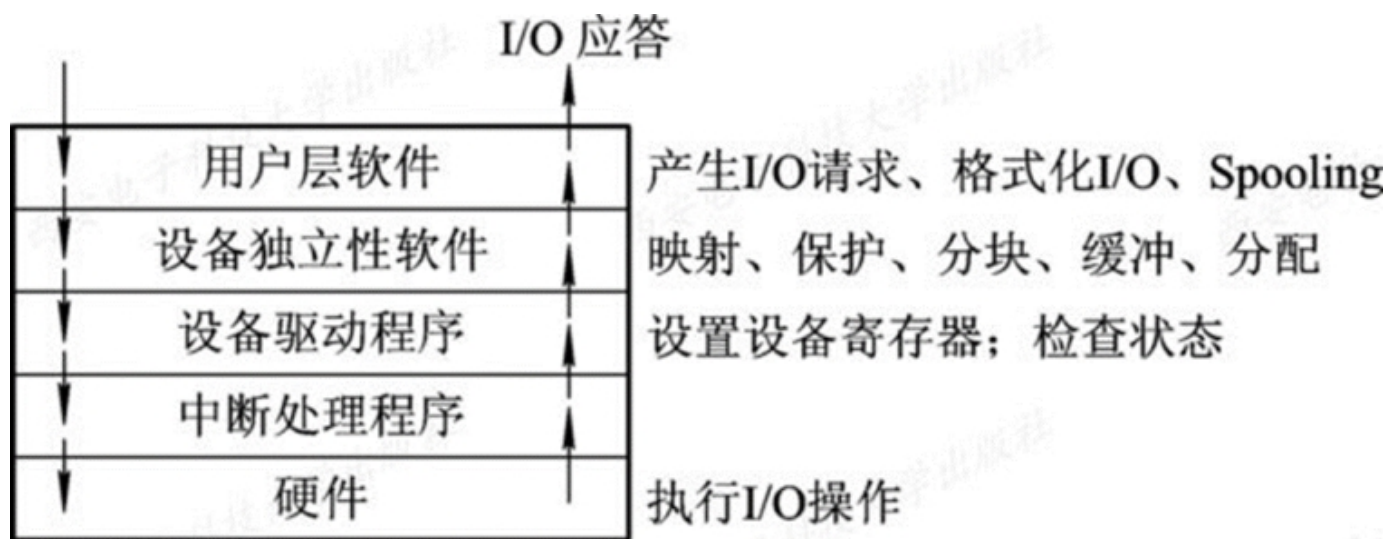


图6-1 I/O系统的层次结构

## 2. I/O系统中各种模块之间的层次视图

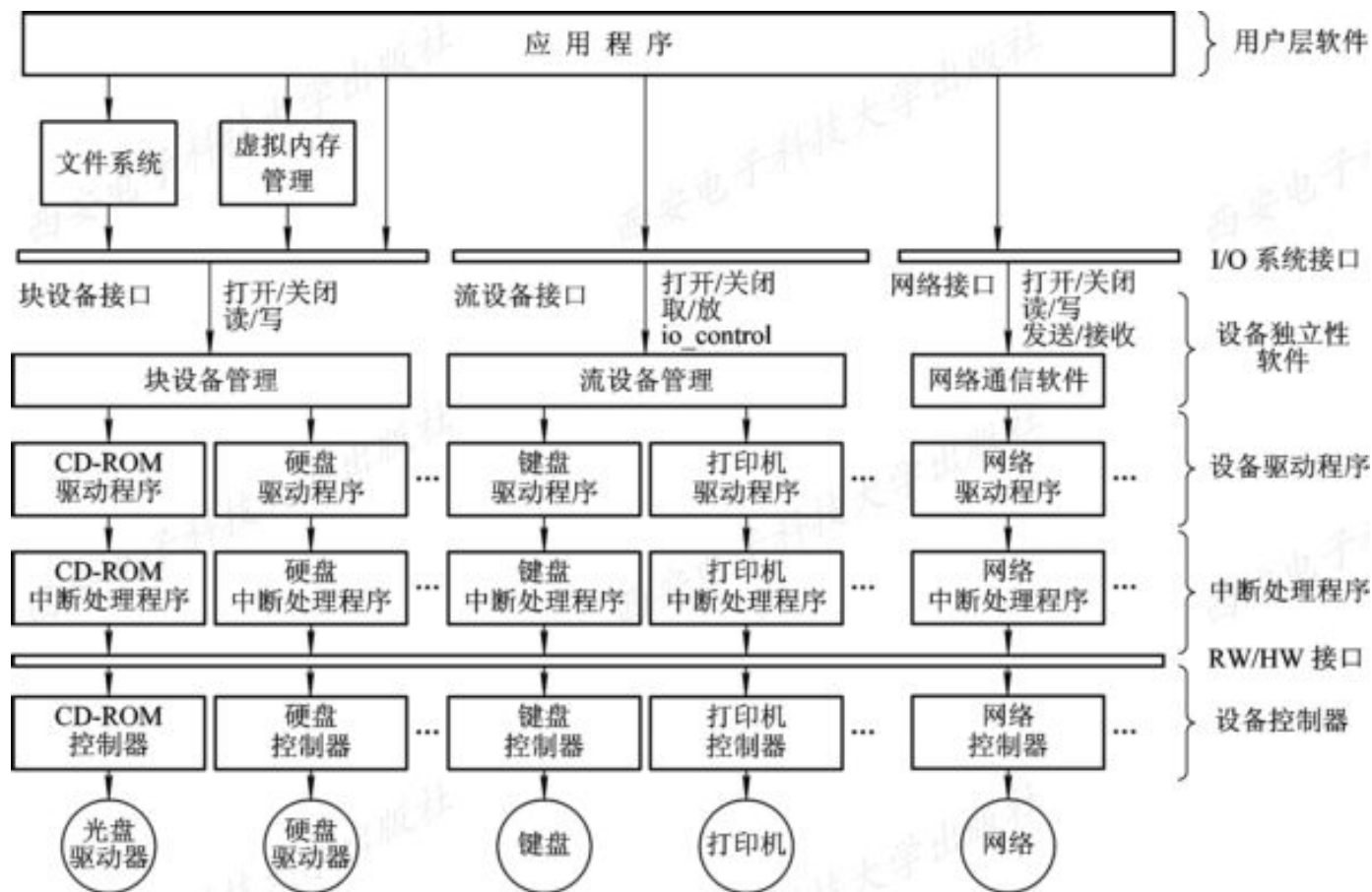




图6-2 I/O系统中各种模块之间的层次视图



## 1) I/O系统的上、下接口

(1) I/O系统接口



(2) 软件/硬件(RW/HW)接口

## 2) I/O系统的分层

(1) 中断处理程序

(2) 设备驱动程序

(3) 设备独立性软件





## 6.1.3 I/O系统接口

### 1. 块设备接口

(1) 块设备：数据的存取和传输都是以数据块为单位的设备。典型的块设备是磁盘。磁盘设备的I/O常采用DMA方式。

(2) 隐藏了磁盘的二维结构

(3) 将抽象命令映射为底层操作



## 2. 流设备接口

(1) 字符设备：数据的存取和传输是以字符为单位的设备，如键盘，打印机等。

(2) get和put操作：通常为字符设备建立一个字符缓冲区（队列）。get操作用于从字符缓冲区取得一个字符（到内存）。put操作用于把一个新字符（从内存）输出到字符缓冲区中，以待送出到设备。

(3) in-control指令：一种通用的指令，用于统一管理字符设备。

### 3. 网络通信接口

在现代OS中，都提供了面向网络的功能。但首先还需要通过某种方式把计算机连接到网络上。同时操作系统也必须提供相应的网络软件和网络通信接口，使计算机能通过网络与网络上的其它计算机进行通信或上网浏览。

## 6.2 I/O设备和设备控制器

I/O设备一般是由执行I/O操作的**机械部分**和执行控制I/O的**电子部件**组成。

通常将这两部分分开，执行I/O操作的机械部分就是一般的I/O设备，而执行控制I/O的电子部件则称为**设备控制器**或**适配器(adapter)**。

在微型机和小型机中的控制器常做成印刷电路卡形式，因而也常称为控制卡、接口卡或网卡，可将它插入计算机的扩展槽中。在有的大、中型计算机系统中，还配置了I/O通道或I/O处理机。





## 6.2.1 I/O设备

### 1. I/O设备的类型

#### 1) 按使用特性分类

(1) 存储设备

(2) I/O设备：输入设备、输出设备和交互式设备

#### 2) 按传输速率分类

(1) 低速设备：键盘、鼠标等

(2) 中速设备：行式打印机、激光打印机等

(3) 高速设备：磁带机、磁盘机、光盘机等



## 2. 设备与控制器之间的接口

通常，设备并不是直接与CPU进行通信，而是与设备控制器通信，因此，在I/O设备中应含有与设备控制器间的接口，在该接口中有三种类型的信号(见图6-3所示)，各对应一条信号线。

- (1) 数据信号线。
- (2) 控制信号线。
- (3) 状态信号线。

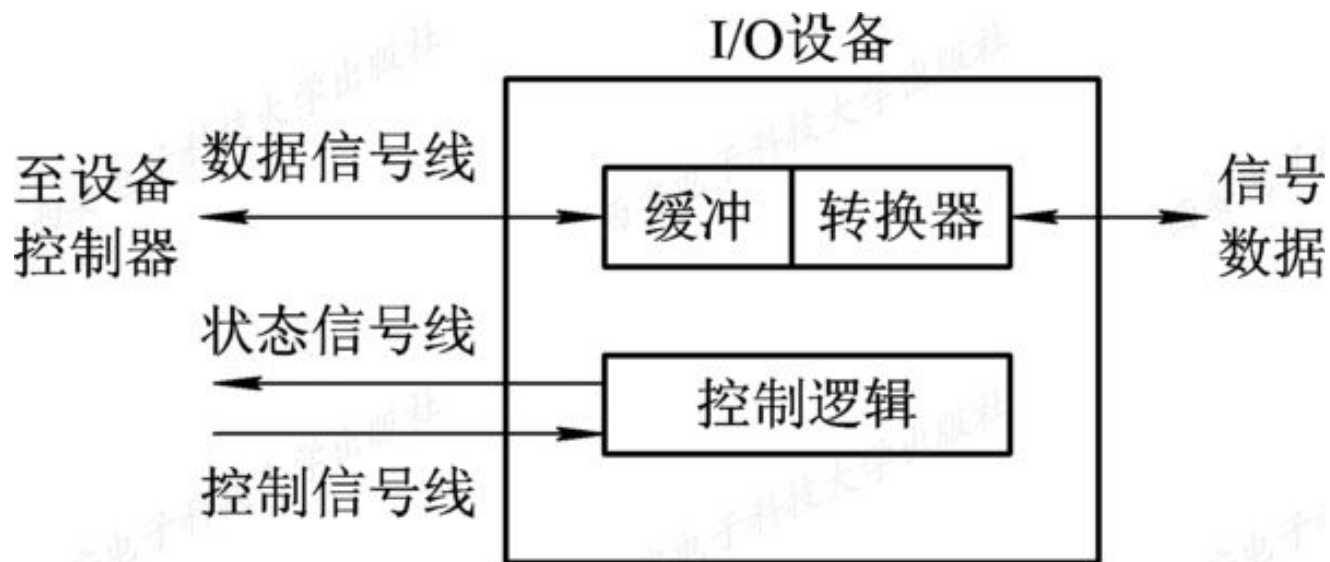


图6-3 设备与控制器间的接口



## 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) 数据交换
- (3) 标识和报告设备的状态
- (4) 地址识别
- (5) 数据缓冲区
- (6) 差错控制



## 2. 设备控制器的组成

现有的大多数控制器都是由以下三部分组成：

- (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

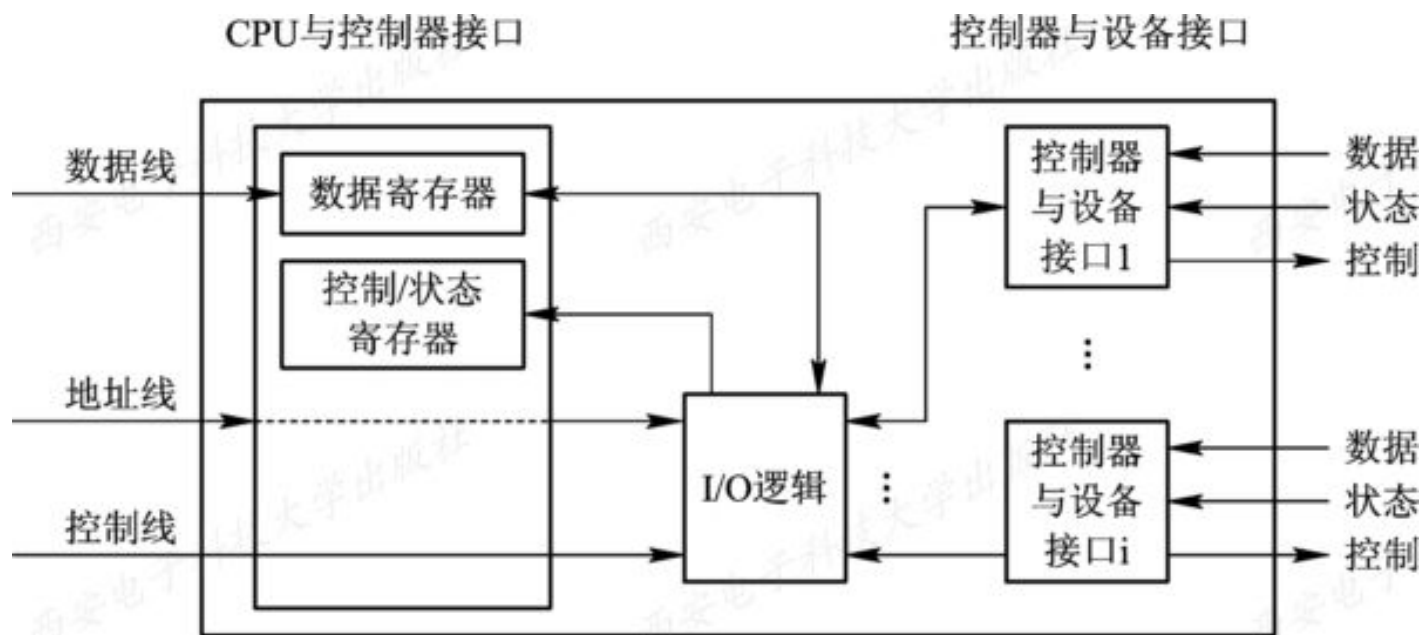


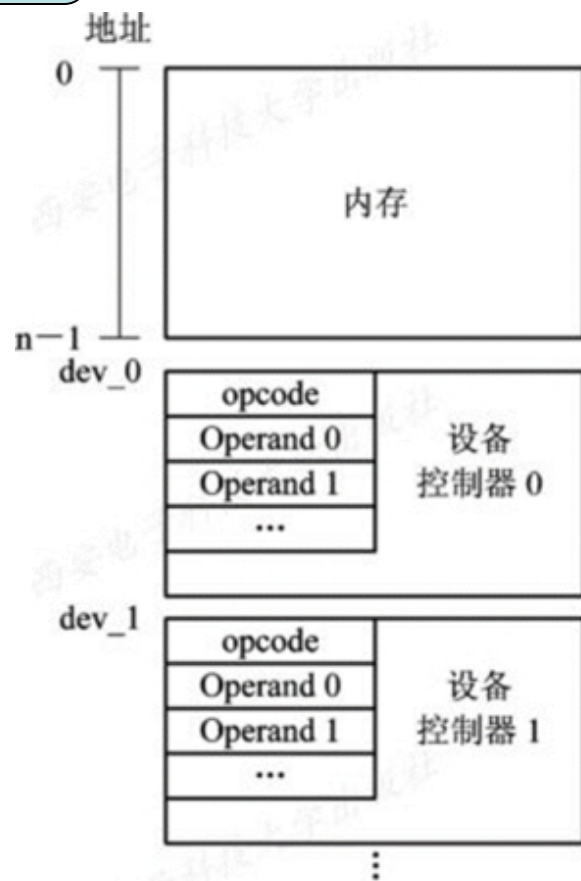
图6-4 设备控制器的组成

缺点：访问内存和设备  
需要两种不同的指令

## 6.2.3 内存映像I/O

### 1. 利用特定的I/O指令

在早期的计算机中，包括大型计算机，为实现CPU和设备控制器之间的通信，为每个控制寄存器分配一个I/O端口，这是一个8位或16位的整数，如图6-5(a)所示。另外还设置了一些特定的I/O指令。



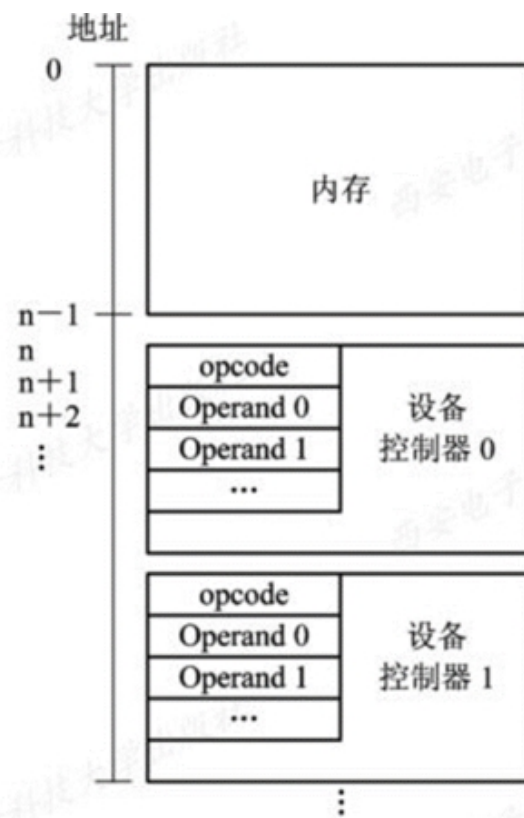
(a) 采用特定的指令形式

举例：x86系统

## 2. 内存映像I/O

在这种方式中，在编址上不再区分内存单元地址和设备控制器中的寄存器地址，都采用 $k$ 。当 $k$ 值处于 $0 \sim n-1$ 范围时，被认为是内存地址，若 $k$ 大于等于 $n$ 时，被认为是某个控制器的寄存器地址。

举例：ARM系统



(b) 内存映像I/O形式



## 6.2.4 I/O通道

### 1. I/O通道设备的引入

虽然在CPU与I/O设备之间增加了设备控制器后，已能大大减少CPU对I/O的干预，但当主机所配置的外设很多时，CPU的负担仍然很重。为此，在CPU和设备控制器之间又增设了I/O通道(I/O Channel)。

I/O通道是一种特殊的处理机。它具有执行I/O指令的能力，并通过执行通道（I/O）程序来控制I/O操作。

I/O通道又与一般的处理机不同：指令类型单一；没有自己的内存。





## 2. 通道类型

### 1) 字节多路通道(Byte Multiplexor Channel)

这是一种按字节交叉方式工作的通道。它通常都含有许多非分配型子通道，其数量可从几十到数百个，每一个子通道连接一台I/O设备，并控制该设备的I/O操作。这些子通道按**时间片轮转方式**共享主通道。

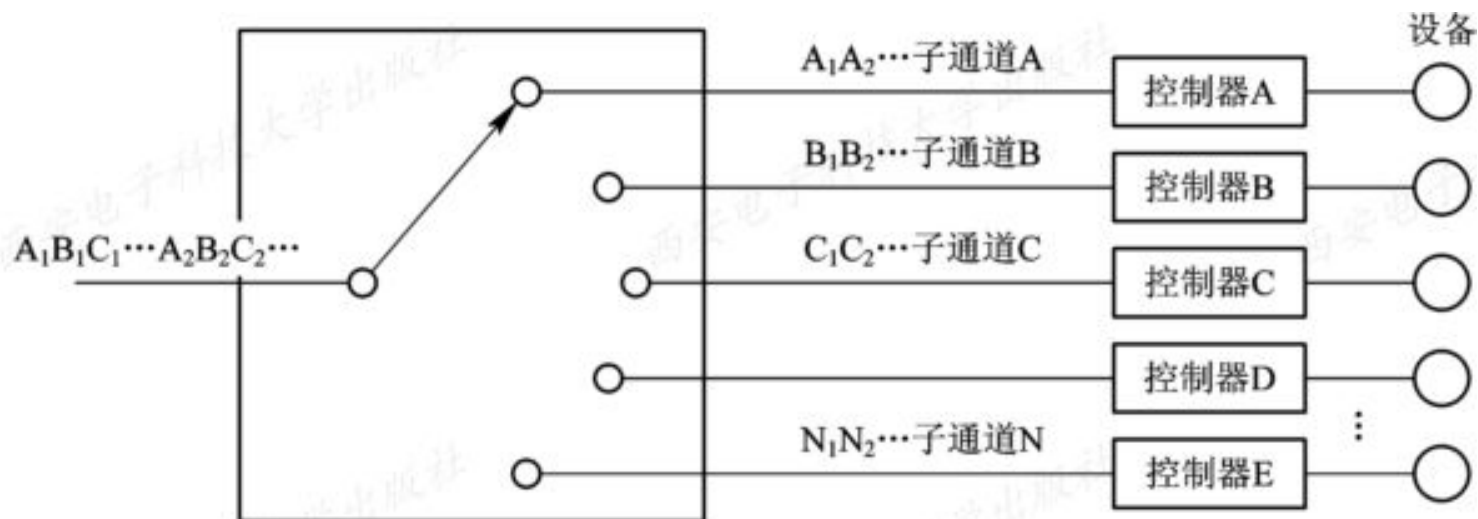


图6-6 字节多路通道的工作原理



## 2) 数组选择通道(Block Selector Channel)

字节多路通道不适于连接高速设备，这推动了按**数组**方式进行数据传送的数组选择通道的形成。



### 3) 数组多路通道(Block Multiplexor Channel)

数组选择通道虽有很高的传输速率，但它却每次只允许一个设备传输数据。

数组多路通道是将数组选择通道传输速率高和字节多路通道能使各子通道(设备)分时并行操作的优点相结合而形成的一种新通道。



### 3. “瓶颈”问题

由于通道价格昂贵，致使机器中所设置的通道数量势必较少，这往往又使它成了I/O的瓶颈，进而造成整个系统吞吐量的下降。



图6-7 单通路I/O系统

增加设备到主机间的通路而不增加通道

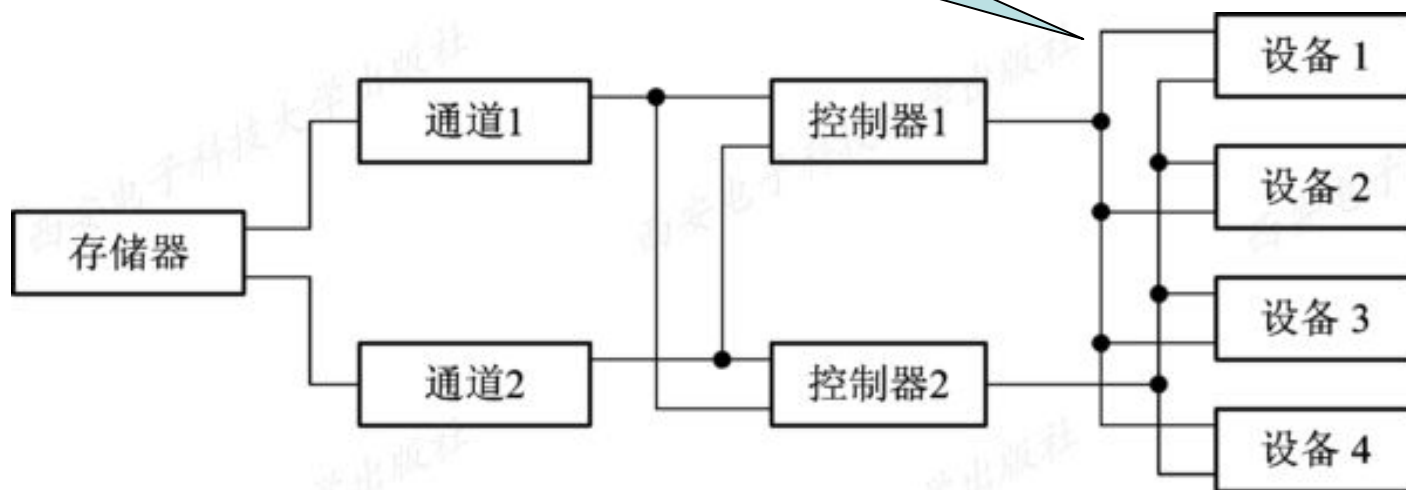


图6-8 多通路I/O系统

## 6.3 中断机构和中断处理程序

中断在操作系统中有着特殊重要的地位，它是**多道程序得以实现**的基础，没有中断，就不可能实现多道程序，因为进程之间的切换是通过中断来完成的。

另一方面，中断也是**设备管理的基础**，为了提高处理机的利用率和实现CPU与I/O设备并行执行，也必需有中断的支持。

中断处理程序是I/O系统中最低的一层，它是整个I/O系统的基础。

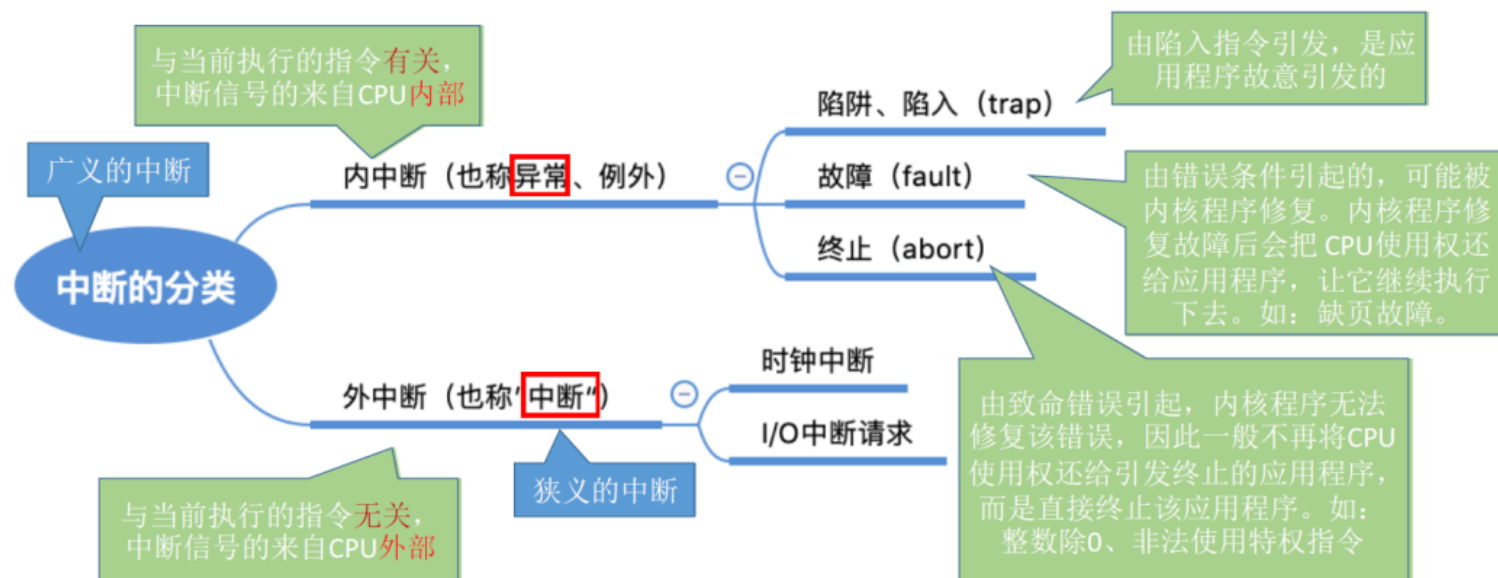


## 6.3.1 中断简介

### 1. 中断和陷入

#### 1) 中断

#### 2) 陷入



## 2. 中断向量表和中断优先级

1) 中断向量表：为每种设备配以相应的中断处理程序，并把该程序的入口地址放在中断向量表的一个表项中，并为每一个设备的中断请求规定一个中断号，它直接对应于中断向量表的一个表项中。

中断号	中断处理程序的入口地址

2) 中断优先级：为中断规定不同的优先级



### 3. 对多中断源的处理方式

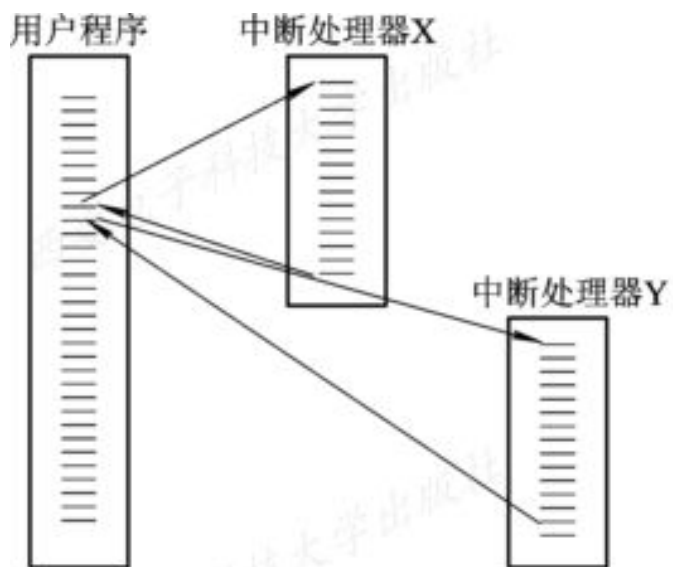
对于多中断信号源的情况，当处理机正在处理一个中断时，又来了一个新的中断请求，有两种处理方法：

1) 屏蔽(禁止)中断

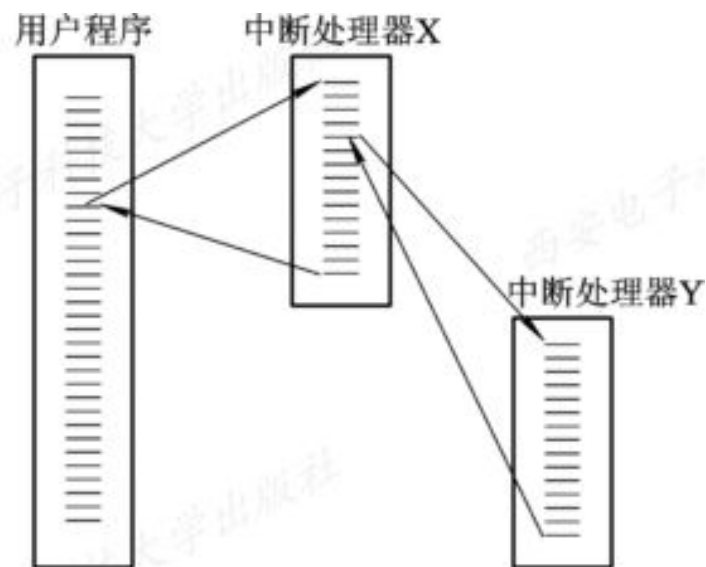
2) 嵌套中断

(1) 当同时有多个不同优先级的中断请求时，CPU优先响应最高优先级的中断请求；

(2) 高优先级的中断请求可以抢占正在运行的低优先级中断的处理机



(a) 顺序中断处理



(b) 嵌套中断处理

图6-9 对多中断的处理方式

### 6.3.2 中断处理程序

当一个进程请求I/O操作时，该进程将被挂起，直到I/O设备完成I/O操作后，设备控制器便向CPU发送一个中断请求，CPU响应后便转向中断处理程序，中断处理程序执行相应的处理，处理完后解除相应进程的阻塞状态。

中断处理程序可以分成5个步骤：

- (1) 测定是否有未响应的中断信号；
- (2) 保护被中断进程的CPU环境；
- (3) 转入相应的设备处理程序；
- (4) 中断处理；
- (5) 恢复CPU的现场并退出中断。

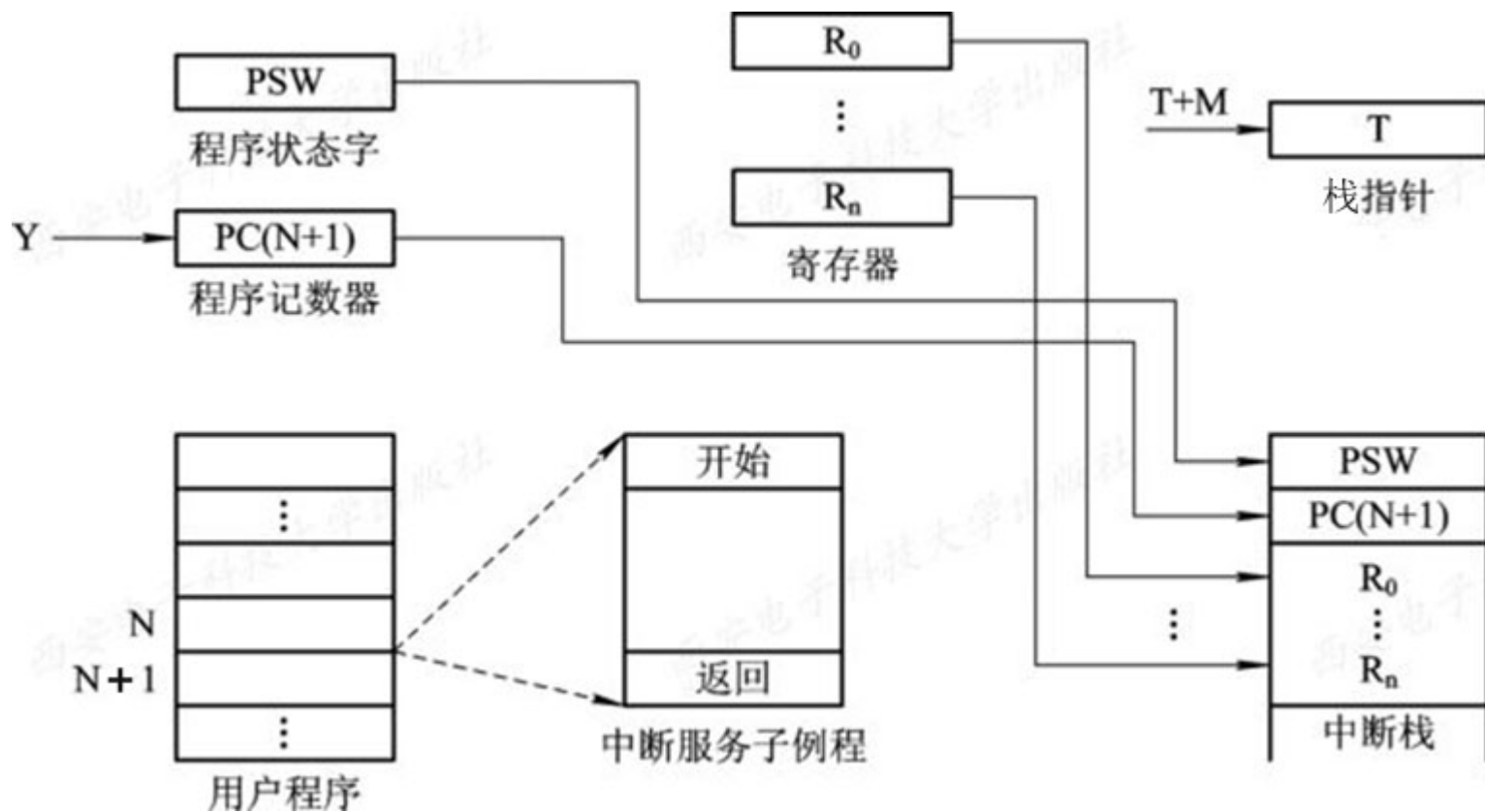


图6-10 中断现场保护示意图

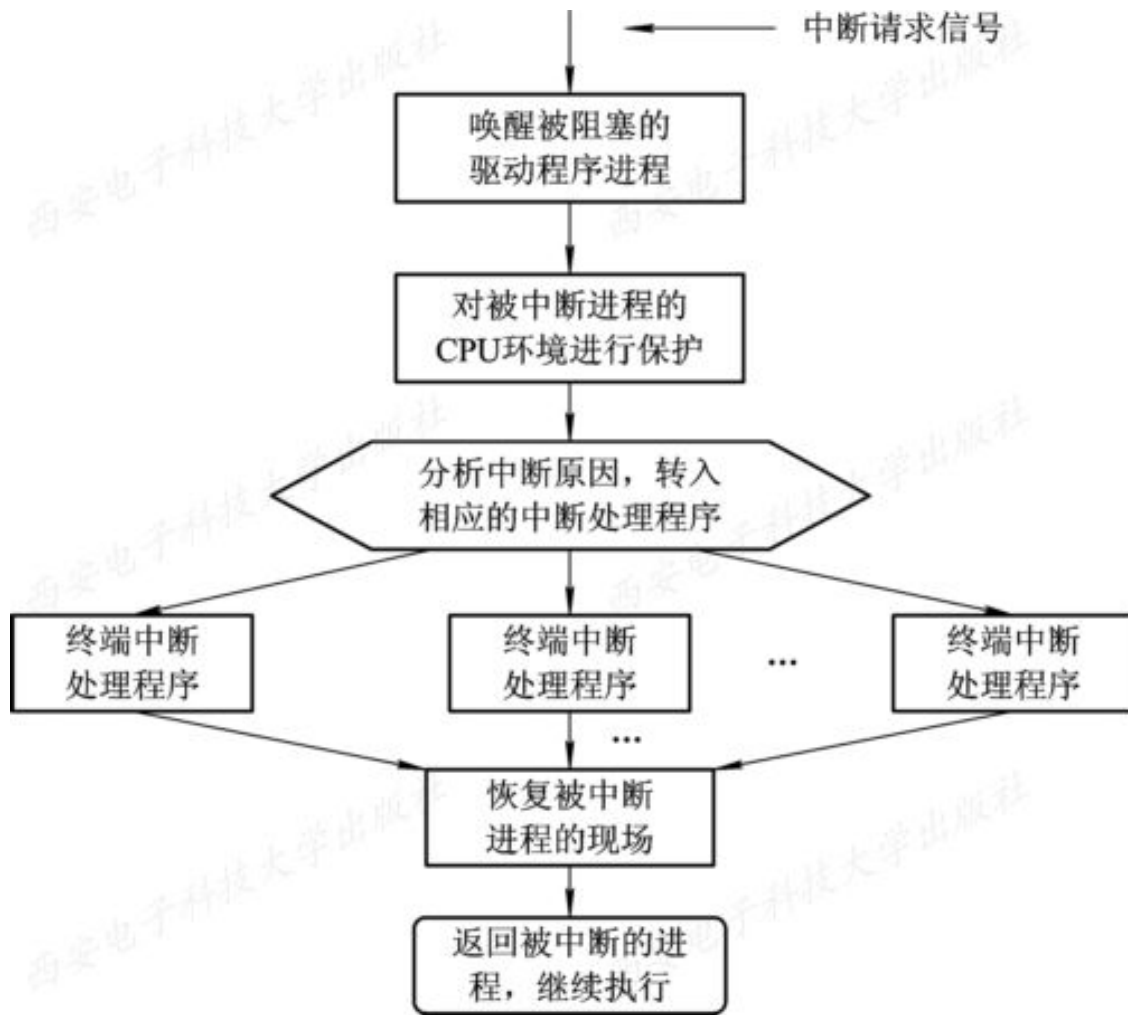
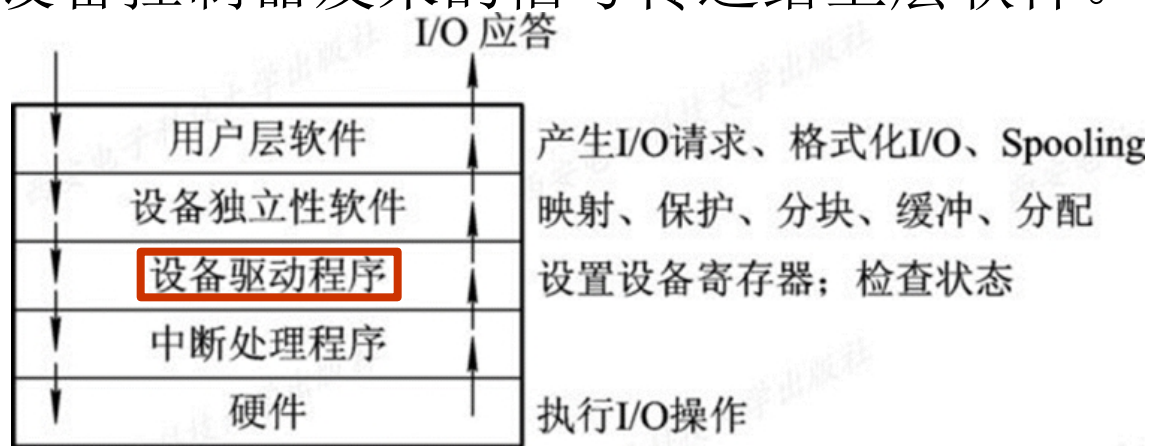


图6-11 中断处理流程



## 6.4 设备驱动程序

设备处理程序通常又称为设备驱动程序，它是**I/O系统**的**高层**与**设备控制器**之间的通信程序，其主要任务是接收上层软件发来的抽象I/O要求，如read或write命令，再把它转换为具体要求后，发送给设备控制器，启动设备去执行；反之，它也将由设备控制器发来的信号传送给上层软件。



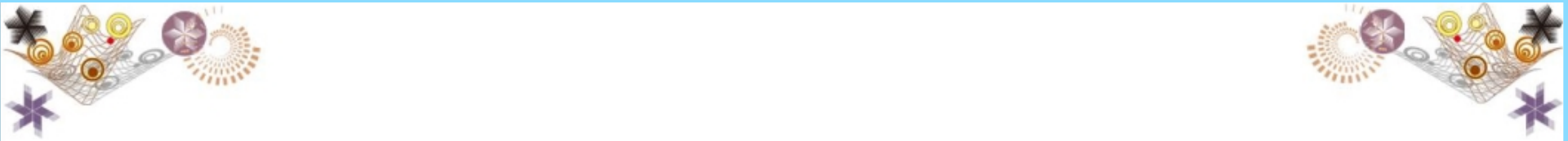




## 6.4.1 设备驱动程序概述

### 1. 设备驱动程序的功能

- (1) **接收**由**与设备无关**的软件发来的**命令和参数**，并将命令中的抽象要求**转换**为与设备相关的低层操作序列。
- (2) **检查用户I/O请求的合法性**，了解I/O设备的工作状态，传递与I/O设备操作有关的参数，设置设备的工作方式。
- (3) **发出I/O命令**，如果设备空闲，便立即启动I/O设备，完成指定的I/O操作；如果设备忙碌，则将请求者的请求块挂在设备队列上等待。
- (4) **及时响应**由设备控制器发来的中断请求，并根据其中断类型，调用相应的中断处理程序进行处理。



## 2. 设备驱动程序的特点

设备驱动程序属于低级的系统例程，它与一般的应用程序及系统程序之间有下列明显差异：

(1) **驱动程序是实现在与设备无关的软件和设备控制器之间通信和转换的程序**，具体说，它将抽象的I/O请求转换成具体的I/O操作后传送给控制器。又把控制器中所记录的设备状态和I/O操作完成情况，及时地反映给请求I/O的进程。

(2) **驱动程序与设备控制器以及I/O设备的硬件特性紧密相关**，对于不同类型的设备，应配置不同的驱动程序。但可以为相同的多个终端设置一个终端驱动程序。



(3) **驱动程序与I/O设备所采用的I/O控制方式**紧密相关，常用的I/O控制方式是中断驱动和DMA方式。

(4) 由于驱动程序**与硬件紧密相关**，因而其中的一部分必须用汇编语言书写。目前有很多驱动程序的基本部分已经固化在ROM中。

(5) **驱动程序应允许可重入**。一个正在运行的驱动程序常会在一次调用完成前被再次调用。





### 3. 设备处理方式

在不同的操作系统中，所采用的设备处理方式并不完全相同。主要分成以下三类：

- 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；
- 3) **不设置**专门的设备处理**进程**，而只为设备设置相应的设备驱动程序，供用户或系统进程调用。这种方式目前用得最多。

## 6.4.2 设备驱动程序的处理过程

设备驱动程序的主要任务是启动指定设备，完成上层指定的I/O工作。但在启动之前，应先完成必要的准备工作，如检测设备状态是否为“忙”等。在完成所有的准备工作后，才向设备控制器发送一条启动命令。处理过程有五步：

- (1) 将抽象要求转换为具体要求；
- (2) 对服务请求进行校验；
- (3) 检查设备的状态；
- (4) 传送必要的参数；
- (5) 启动I/O设备。

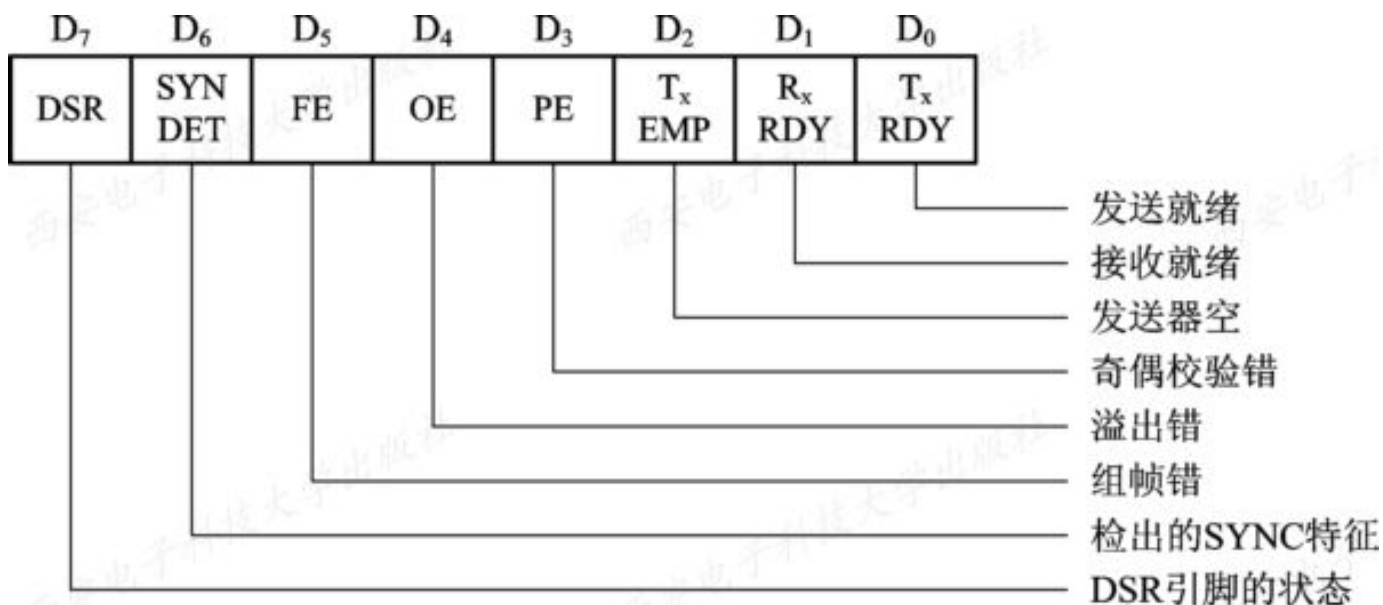


图6-12 状态寄存器中的格式



### 6.4.3 对I/O设备的控制方式

对设备的控制，早期是使用轮询的可编程I/O方式，后来发展为使用中断的可编程I/O方式。



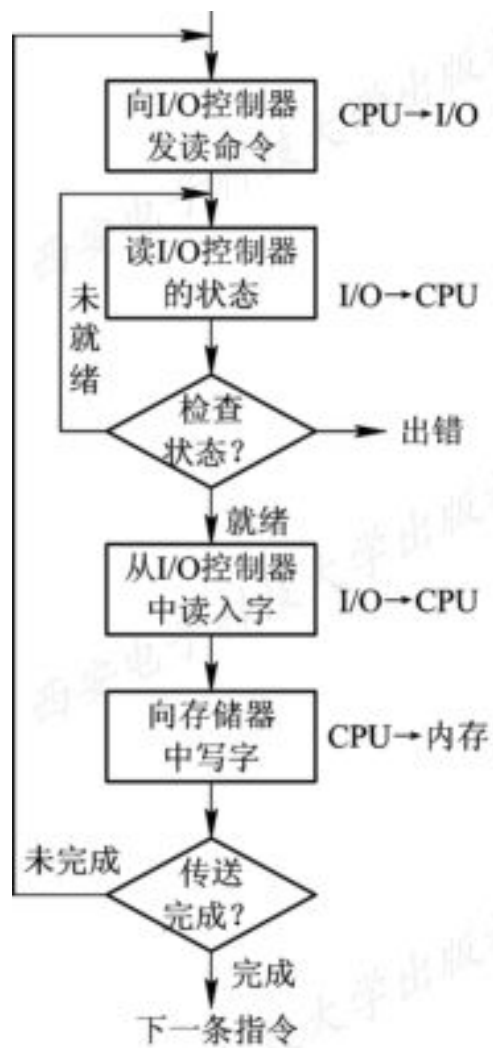
## 1. 使用轮询的可编程I/O方式

在处理器向控制器发出一条I/O指令，启动输入设备输入数据时，要同时把状态寄存器中的忙/闲标志busy置为1，然后便不断地循环测试busy(称为轮询)。

当busy=1时，表示输入机尚未输完一个字(符)，处理器应继续对该标志进行测试，直至busy=0，表明输入机已将输入数据送入控制器的数据寄存器中。

于是处理器将数据寄存器中的数据取出，送入内存指定单元中，这样便完成了一个字(符)的I/O。接着再去启动读下一个数据，并置busy=1。



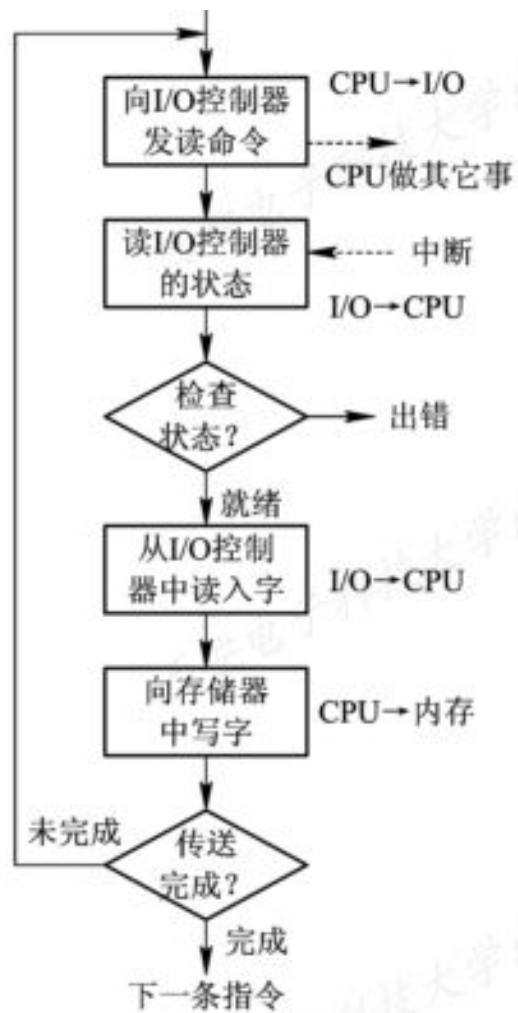


(a) 程序I/O方式

图6-13 程序I/O和中断驱动方式的流程

## 2. 使用中断的可编程I/O方式

当前，对I/O设备的控制，广泛采用中断的可编程I/O方式，即当某进程要启动某个I/O设备工作时，便由CPU向相应的设备控制器发出一条I/O命令，然后立即返回继续执行原来的任务。设备控制器于是按照该命令的要求去控制指定I/O设备。此时，CPU与I/O设备并行操作。



(b) 中断驱动I/O方式

图6-13 程序I/O和中断驱动方式的流程



### 3. 直接存储器访问方式

#### 1) 直接存储器访问方式的引入

虽然中断驱动I/O比程序I/O方式更有效，但它仍是以字(节)为单位进行I/O的。每当完成一个字(节)的I/O时，控制器便要向CPU请求一次中断。

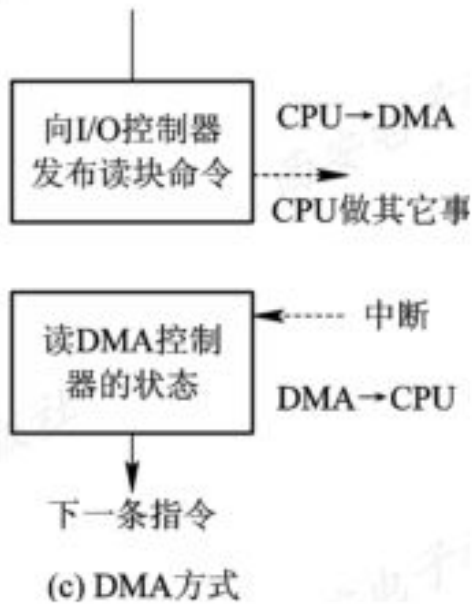




图6-13 程序I/O和中断驱动方式的流程





该方式的特点是：

(1) 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。

(2) 所传送的数据是从设备直接送入内存的，或者相反。

(3) 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控制器的控制下完成的。可见，DMA方式较之中断驱动方式又进一步提高了CPU与I/O设备的并行操作程度。



## 2) DMA控制器的组成

DMA控制器由三部分组成：主机与DMA控制器的接口；  
DMA控制器与块设备的接口；I/O控制逻辑。

## 2) DMA控制器的组成

DMA控制器由三部分组成：

DMA控制器与块设备的接口；I

DR: 数据寄存器  
MAR: 内存地址寄存器  
DC: 数据计数器  
CR: 命令/状态寄存器

的接口；

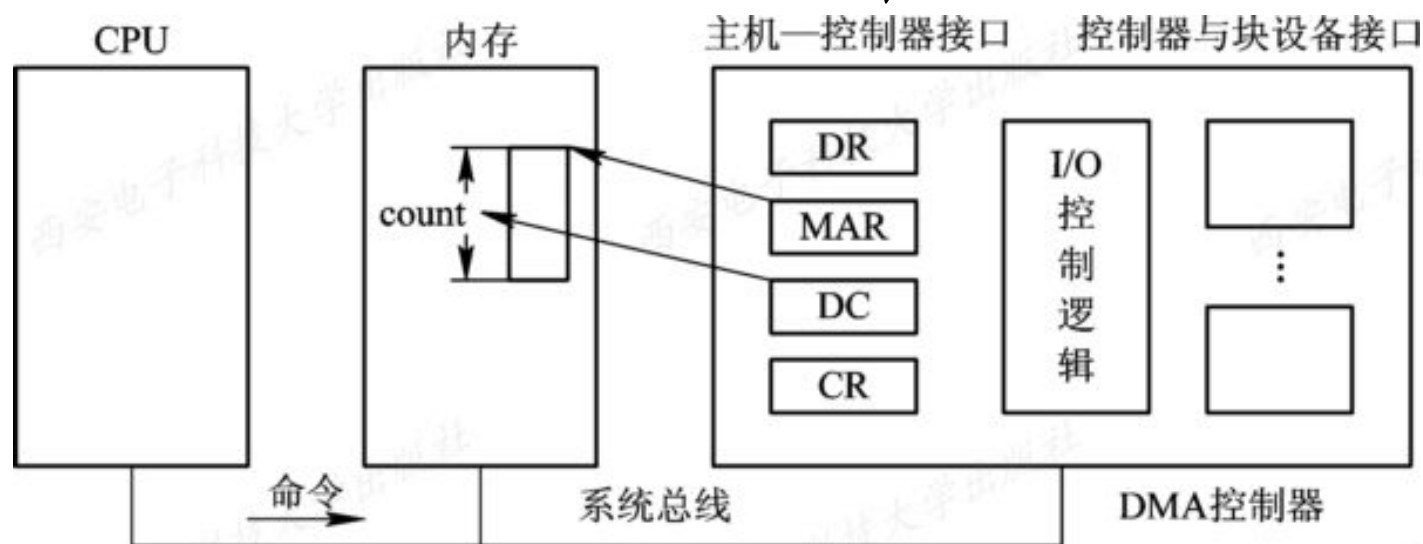


图6-14 DMA控制器的组成



### 3) DMA工作过程

当CPU要从磁盘读入一数据块时，便向磁盘控制器发送一条读命令。该命令被送入命令寄存器CR中。同时，需要将本次要读入数据在内存的起始目标地址送入内存地址寄存器MAR中。

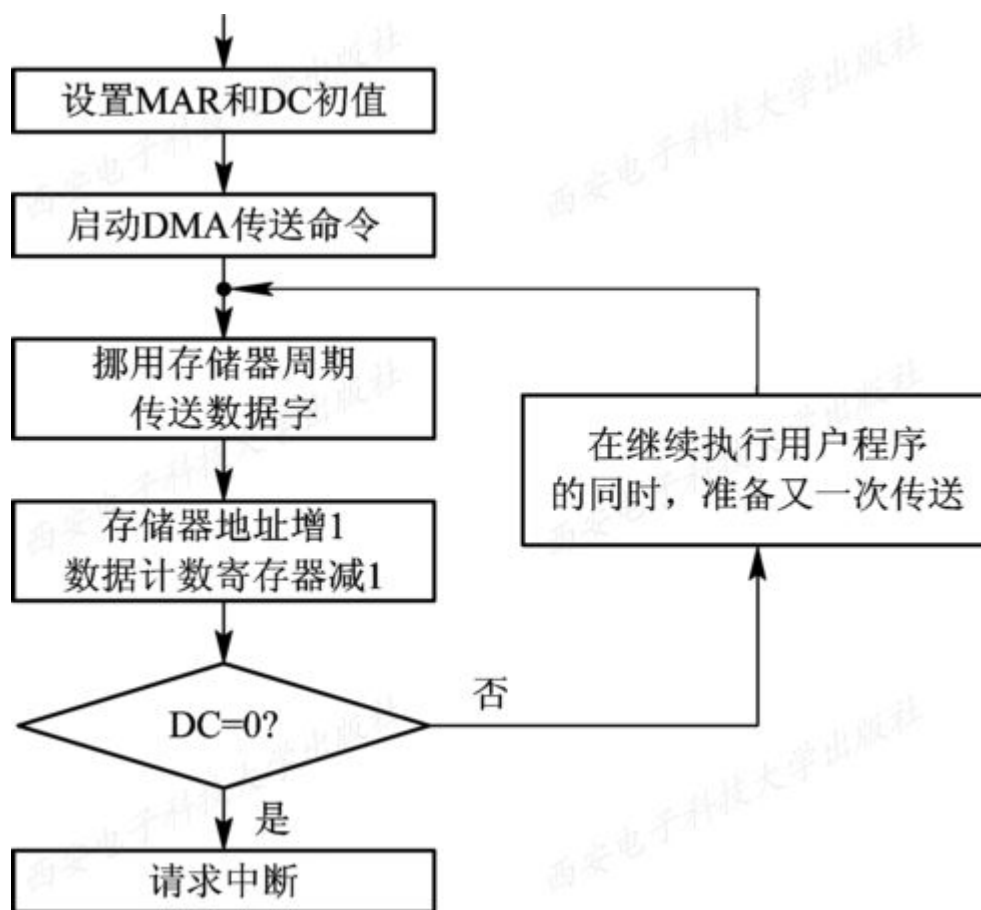


图6-15 DMA方式的工作流程图

## 4. I/O通道控制方式



### 1) I/O通道控制方式的引入

I/O通道方式是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写为单位的干预，减少为对**一组数据块的读写及有关的管理**为单位的干预。同时，又可以实现CPU、通道和I/O设备三者的并行操作，从而更有效地提高系统的资源利用率。

## 2) 通道程序

通道是通过执行通道程序并与设备控制器共同实现对I/O设备的控制的。通道程序是由一系列通道指令(或称为通道命令)所构成的。每条指令中都包含下列信息：

- (1) **操作码**：指令所执行的操作
- (2) **内存地址**：读写内存的内存首址
- (3) **计数**：读写数据的字节数
- (4) **通道程序结束位P**：通道是否结束，P=1代表本条指令是通道程序的最后一条指令；
- (5) **记录结束标志R**：R=0表示本通道与下一条指令所处理的数据是同属于一个记录；R=1表示这是处理某记录的最后一条指令。



下面示出了一个由六条通道指令所构成的简单的通道程序。该程序的功能是将内存中不同地址的数据写成多个记录。

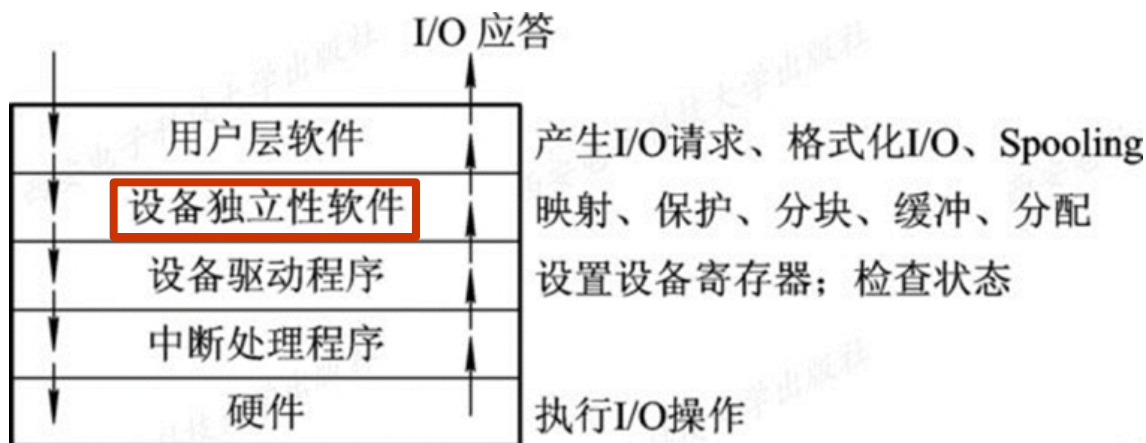
## 四种控制方式的比较

	完成一次读/写的过程	CPU干 预频率	每次I/O的数 据传输单位	数据流向	优缺点
程序直接控 制方式	CPU发出I/O命令后需要不断轮询	极高	字	设备→CPU→内存 内存→CPU→设备	每一个阶段的优点都是解决了上一阶段的 最大缺点。 总体来说，整个发展过程就是要尽量减少 CPU对I/O过程的干预，把CPU 从繁杂的I/O控制事务中解脱 出来，以便更多地去完成数 据处理任务。
中断驱动方 式	CPU发出I/O命令后可以 做其他事，本次I/O完成后 设备控制器发出中断信号	高	字	设备→CPU→内存 内存→CPU→设备	
DMA方式	CPU发出I/O命令后可以 做其他事，本次I/O完成后 DMA控制器发出中断信号	中	块	设备→内存 内存→设备	
通道控制方 式	CPU发出I/O命令后可以 做其他事。通道会执行通 道程序以完成I/O，完成后 通道向CPU发出中断信号	低	一组块	设备→内存 内存→设备	



## 6.5 与设备无关的I/O软件

为了方便用户和提高OS的可适应性与可扩展性，在现代OS的I/O系统中，都无一例外地增加了与设备无关的I/O软件，以实现**设备独立性**，也称为**设备无关性**。其基本含义是：应用程序中所用的设备，不局限于使用某个具体的物理设备。为每个设备所配置的设备驱动程序是与硬件紧密相关的软件。





## 6.5.1 与设备无关(Device Independence)软件的基本概念

### 1. 以物理设备名使用设备

在早期OS中，应用程序在使用I/O设备时，都使用设备的物理名称，这使应用程序与系统中的物理设备直接相关。





## 2. 引入了逻辑设备名

为了实现与设备的无关性而引入了逻辑设备和物理设备两个概念。**逻辑设备**是抽象的设备名。



### 3. 逻辑设备名称到物理设备名称的转换

在应用程序中，用逻辑设备名称使用设备虽然方便了用户，但系统却只识别物理设备名称，因此在实际执行时，还必须使用物理名称。为此，在系统中，必须具有将逻辑设备名称转换为某物理设备名称的功能。



## 6.5.2 与设备无关的软件

### 1. 设备驱动程序的统一接口

为了使所有的设备驱动程序有着统一的接口，一方面，**要求每个设备驱动程序与OS之间都有着相同的接口**，或者相近的接口，这样会使添加一个新的设备驱动程序变得很容易。同时，在很大程度上方便了开发人员对设备驱动程序的编制。



另一方面，要将抽象的设备名映射到适当的驱动程序上，或者说，**将抽象的设备名转换为具体的物理设备名**，并进一步可以找到相应物理设备的驱动程序入口。此外，还应对设备进行保护，禁止用户直接访问设备，以防止无权访问的用户使用。



## 2. 缓冲管理

无论是字符设备还是块设备，它们的运行速度都远低于CPU的速度。

为了缓和CPU和I/O设备之间的矛盾、提高CPU的利用率，在现代OS中都无一例外地分别为字符设备和块设备配置了相应的**缓冲区**。缓冲区有着多种形式，如单缓冲区、双缓冲区、循环缓冲区、公用缓冲池等，以满足不同情况的需要。



### 3. 差错控制

由于设备中有着许多的机械和电气部分，因此，它们比主机更容易出现故障，这就导致I/O操作中的绝大多数错误都与设备有关。错误可分为如下两类：

- (1) 暂时性错误。
- (2) 持久性错误。

#### 4. 对独立设备的分配与回收

在系统中有两类设备：独占设备和共享设备。



对于独占设备，为了避免诸进程对独占设备的争夺，必须由系统来统一分配，不允许进程自行使用。每当进程需要使用某(独占)设备时，必须先**提出申请**。

OS接到对设备的请求后，先对进程所请求的独占设备进行检查，**看该设备是否空闲**。若空闲，才把该设备分配给请求进程。否则，进程将被阻塞，放入该设备的请求队列中等待。等到其它进程释放该设备时，再将队列中的第一个进程唤醒，该进程得到设备后继续运行。

## 5. 独立于设备的逻辑数据块

不同类型的设备，其数据交换单位是不同的，读取和传输速率也各不相同，如字符型设备以单个字符(字)为单位，块设备是以一个数据块为单位。即使同一类型的设备，其数据交换单位的大小也是有差异的，如不同磁盘由于扇区大小的不同，可能造成数据块大小的不一致。

设备独立性软件应能够隐藏这些差异而被逻辑设备使用，并向高层软件提供大小统一的逻辑数据块。。



设备驱动程序的统一接口
缓冲
错误报告
分配与释放专用设备
提供与设备无关的块大小

图6-16 与设备无关软件的功能层次





### 6.5.3 设备分配

系统为实现对独占设备的分配，必须在系统中配置相应的数据结构。

#### 1. 设备分配中的数据结构

在用于设备分配的数据结构中，记录了对设备或控制器进行控制所需的信息。在进行设备分配时需要如下的数据结构。

##### 1) 设备控制表DCT

系统为每一个设备都配置了一张设备控制表，用于记录设备的情况，如图6-17所示。

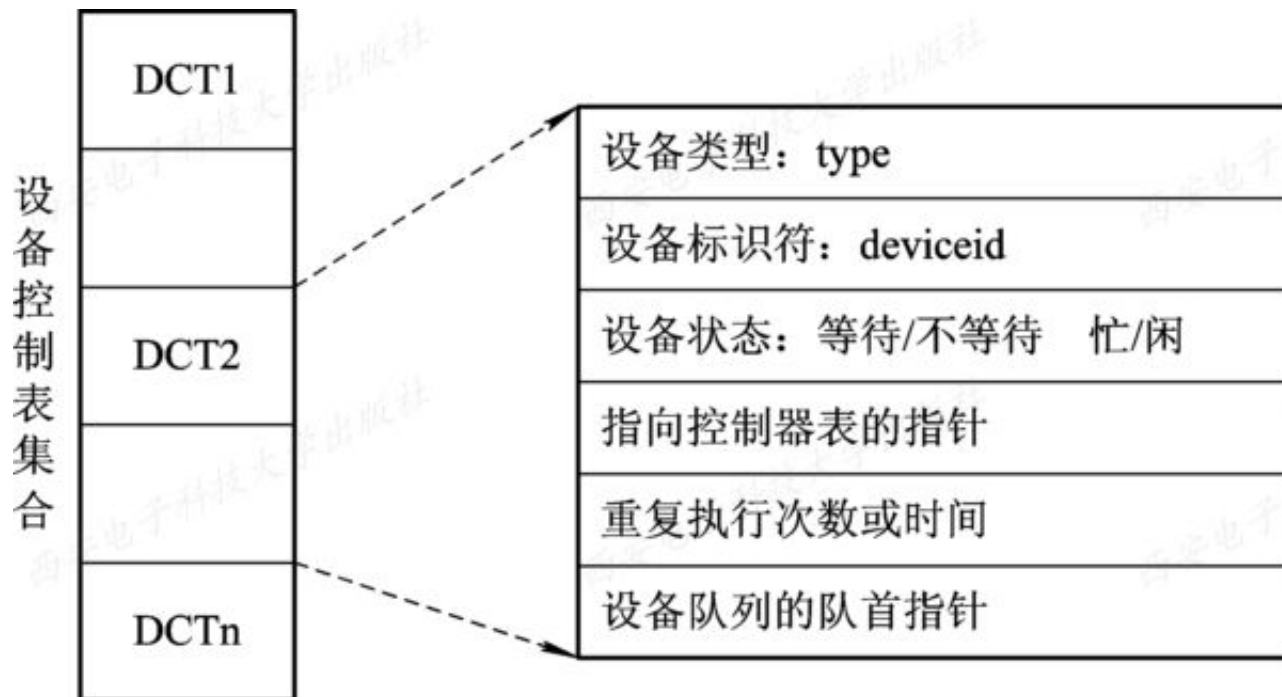



图6-17 设备控制表



## 2) 控制器控制表、通道控制表和系统设备表

(1) 控制器控制表(COCT)。系统为每一个控制器都设置了用于记录控制器情况的控制器控制表。

(2) 通道控制表(CHCT)。每个通道都有一张通道控制表。


(3) 系统设备表(SDT)。这是系统范围的数据结构，记录了系统中全部设备的情况，每个设备占一个表目，其中包括有设备类型、设备标识符、设备控制表及设备驱动程序的入口等项。

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) 控制器控制表(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) 通道控制表(CHCT)

表目1	
⋮	
表目i	
⋮	

设备类
设备标识符
DCT
驱动程序入口

(c) 系统设备表(SDT)

图6-18 COCT、CHCT和SDT表



## 2. 设备分配时应考虑的因素

系统在分配设备时，应考虑如下几个因素：

### 1) 设备的固有属性

设备的固有属性可分成三种，对它们应采取不同的分配策略：

(1) 独占设备的分配策略。

(2) 共享设备的分配策略。

(3) 虚拟设备的分配策略，虚拟设备属于可共享的设备，可以将它同时分配给多个进程使用，如Spooling技术。



## 2) 设备分配算法

对设备分配的算法，通常只采用以下两种分配算法：

- (1) 先来先服务。
- (2) 优先级高者优先。

### 3) 设备分配中的安全性

从进程运行的安全性上考虑，设备分配有以下两种方式：

(1) 安全分配方式：每当进程发出I/O请求后，便进入阻塞状态，直到其I/O操作完成时才被唤醒。本方式摒弃了“请求和保持”条件，因此是设备分配是安全的。

(2) 不安全分配方式：进程在发出I/O请求后仍继续运行，需要时又发出第二个I/O请求，第三个I/O请求等。本方式可能产生死锁。



### 3. 独占设备的分配程序

#### 1) 基本的设备分配程序

我们通过一个例子来介绍设备分配过程。当某进程提出I/O请求后，系统的设备分配程序可按下述步骤进行设备分配：

- (1) 分配设备
- (2) 分配控制器
- (3) 分配通道





## 2) 设备分配程序的改进

在上面的例子中，进程是以物理设备名提出I/O请求的。如果所指定的设备已分配给其它进程，则分配失败。或者说上面的设备分配程序不具有与设备无关性。为获得设备的独立性，进程应使用逻辑设备名请求I/O。

## 6.5.4 逻辑设备名到物理设备名映射的实现

### 1. 逻辑设备表LUT(Logical Unit Table)

在逻辑设备表的每个表目中包含了三项：逻辑设备名、物理设备名和设备驱动程序的入口地址，如图6-19(a)所示。


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) 表一

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

(b) 表二

图6-19 逻辑设备表



## 2. 逻辑设备表的设置问题

在系统中可采取两种方式设置逻辑设备表：

第一种方式，是在整个系统中只设置一张LUT。

第二种方式，是为每个用户设置一张LUT。

## 6.6 用户层的I/O软件

### 6.6.1 系统调用与库函数

#### 1. 系统调用

一方面，为使诸进程能有条不紊地使用I/O设备，且能保护设备的安全性，不允许运行在用户态的应用进程去直接调用运行在核心态(系统态)的OS过程。但另一方面，应用进程在运行时，又必须取得OS所提供的服务，否则，应用程序几乎无法运行。

为了解决此矛盾，OS在用户层中引入了一个中介过程——**系统调用**，应用程序可以通过它间接调用OS中的I/O过程，对I/O设备进行操作。

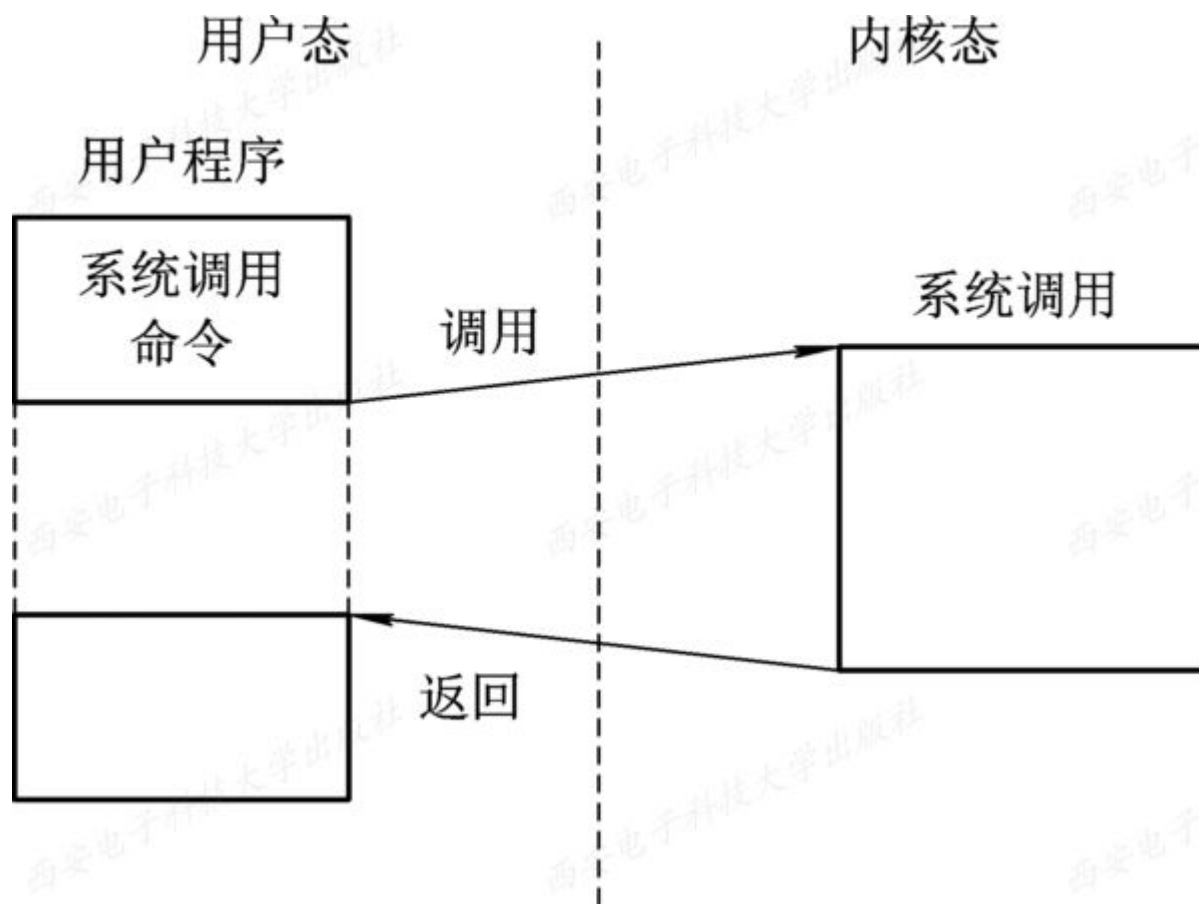
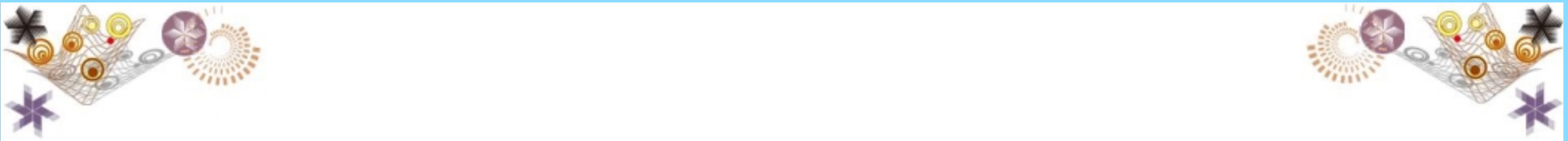


图6-20 系统调用的执行过程



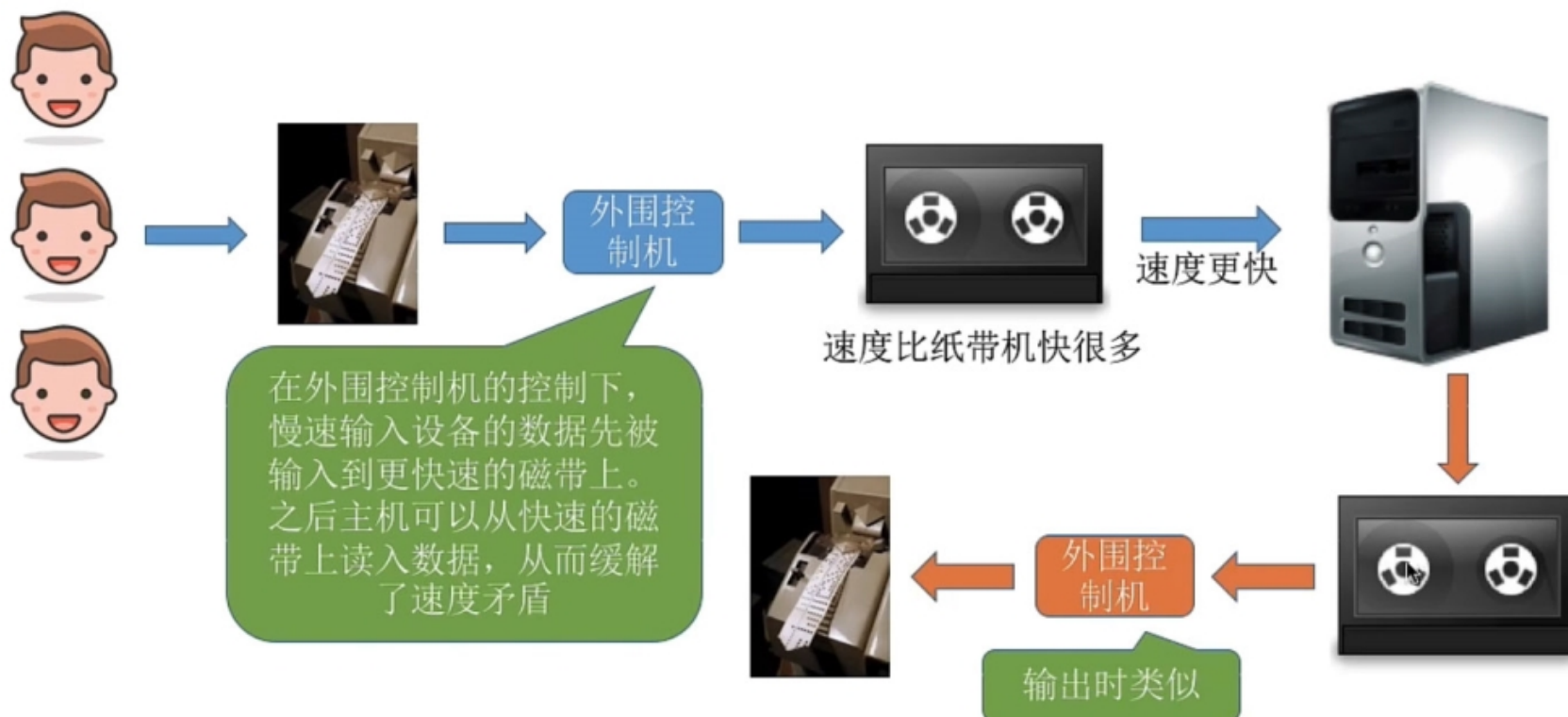
## 2. 库函数

在C语言以及UNIX系统中，系统调用(如read)与各系统调用所使用的库函数(如read)之间几乎是一一对应的。



而微软定义了一套过程，称为**Win32 API**的应用程序接口(Application Program Interface)，程序员利用它们取得OS服务，该接口与实际的系统调用并不一一对应。用户程序通过调用对应的库函数使用系统调用，这些库函数与调用程序连接在一起，被嵌入在运行时装入内存的二进制程序中。

## 6.6.2 假脱机(Spooling)系统

### 1. 假脱机技术







## 6.6.2 假脱机(Spooling)系统

### 1. 假脱机技术

在20世纪50年代，为了缓和CPU的高速性与I/O设备低速性间的矛盾，而引入了脱机输入、脱机输出技术。

该技术是利用专门的外围控制机，先将低速I/O设备上的数据传送到高速磁盘上，或者相反。这样当处理机需要输入数据时，便可以直接从磁盘中读取数据，极大地提高了输入速度。反之，在处理机需要输出数据时，也可以很快的速度把数据先输出到磁盘上，处理机便可去做自己的事情。



## 2. SPOOLing的组成

如前所述，SPOOLing技术是对脱机输入/输出系统的模拟，相应地，如图6-21(a)所示，SPOOLing系统建立在通道技术和多道程序技术的基础上，以高速随机外存(通常为磁盘)为后援存储器。SPOOLing的工作原理如图6-21(b)所示。

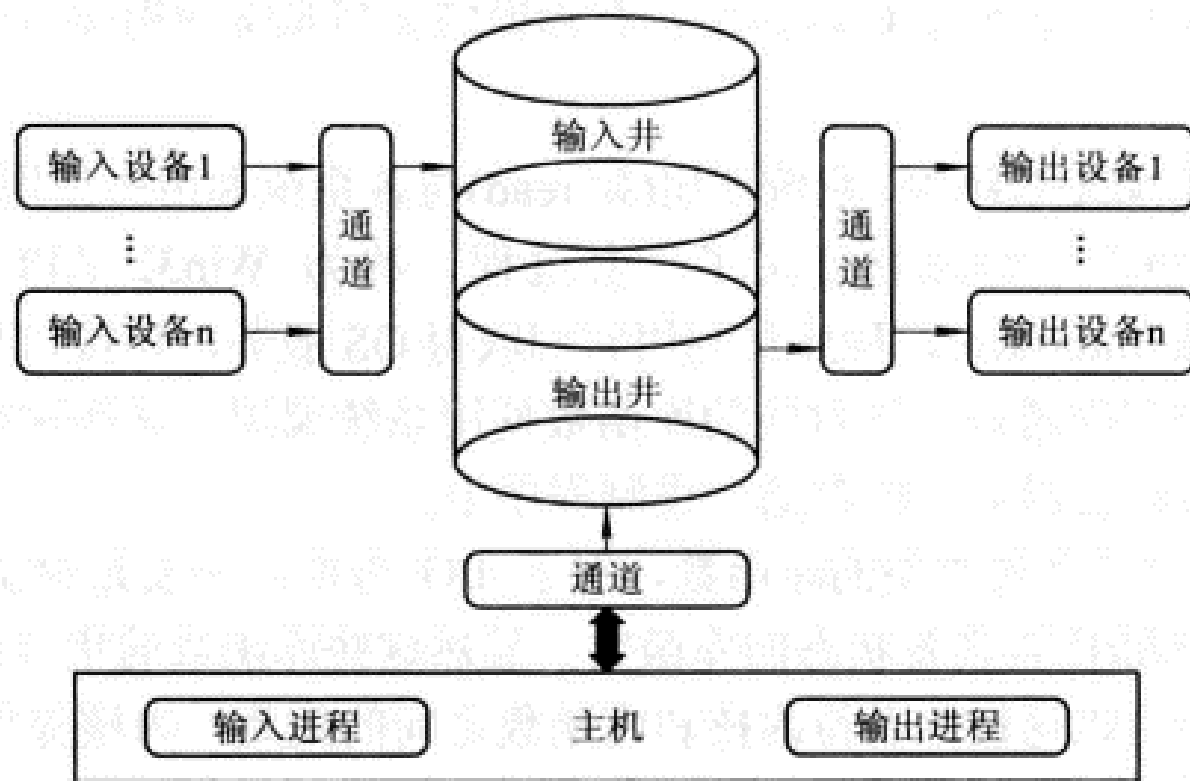


图6-21 SPOOLing系统组成

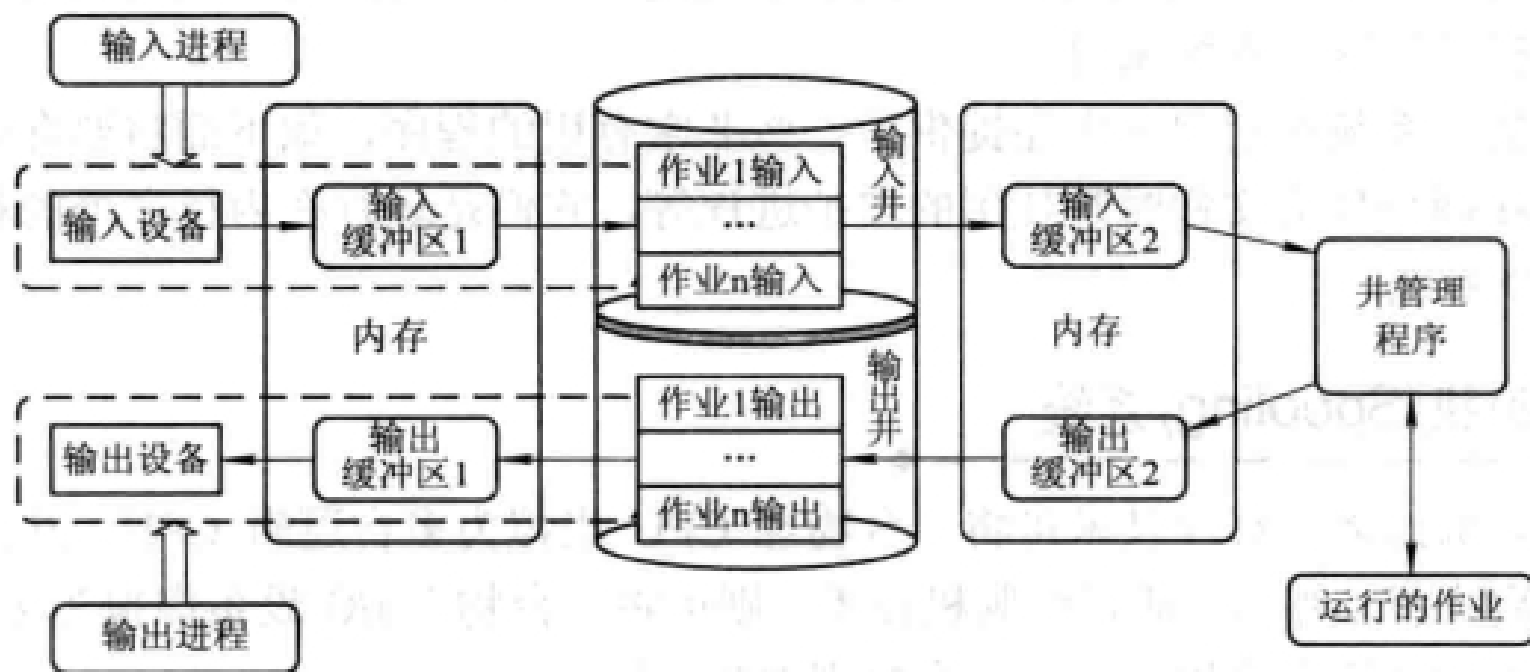




图6-21 SPOOLing的工作原理





SPOOLing系统主要由以下四部分构成：

(1) 输入井和输出井：在**磁盘**上开辟出来的两个存储区域。

**输入井模拟脱机输入时的磁盘**，用于收容I/O设备输入的数据。

**输出井模拟脱机输出时的磁盘**，用于收容用户程序的输出数据。这些数据以文件的形式组织管理，称为井文件。

(2) 输入缓冲区和输出缓冲区：在**内存**中开辟出的两个缓冲区，用于缓和CPU和磁盘之间速度不匹配的矛盾。



(3) 输入进程和输出进程：输入进程也称为预输入进程，用于模拟脱机输入时的外围控制机，将用户要求的数据从输入设备传送到输入缓冲区，再存放到输入井。输出进程也称为缓输出进程，用于模拟脱机输出时的外围控制机，把用户要求输入的数据从内存传送并存放到输出井，待输出设备空闲时，再将输出井中的数据经过输出缓冲区输出至输出设备上。

(4) 井管理程序：用于控制作业与磁盘井之间信息的交换。



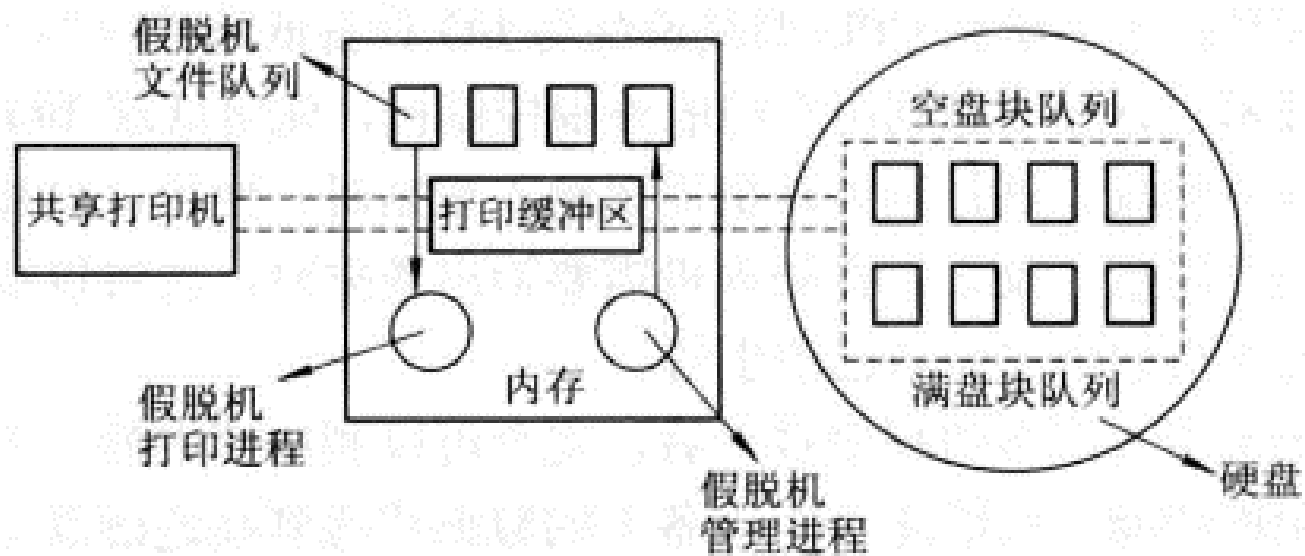
### 3. SPooling系统的特点

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

#### 4. 假脱机打印系统

打印机是经常用到的输出设备，属于独占设备。利用假脱机技术可将它改造为一台可供多个用户共享的打印设备，从而提高设备的利用率，也方便了用户。共享打印机技术已被广泛地用于多用户系统和局域网络中。假脱机打印系统主要有以下三部分：

- (1) 磁盘缓冲区：在磁盘上开辟的一个存储空间，用于暂存用户程序的输出数据。
- (2) 打印缓冲区：用于缓和CPU和磁盘之间速度不匹配的矛盾。
- (3) 假脱机管理进程和假脱机打印进程。



(c) 假脱机打印机系统的组成

假脱机打印系统的组成





## 5. 守护进程(daemon)

取消该方案中的假脱机管理进程，为打印机建立一个守护进程，由它执行一部分原来由假脱机管理进程实现的功能，另一部分由请求进程自己完成。

**守护进程是允许使用打印机的唯一进程。**

除了打印机守护进程之外，还可能有许多其他的守护进程。

## 6.7 缓冲区管理

在现代操作系统中，几乎所有的I/O设备在与处理机交换数据时都用了缓冲区。

缓冲区是一个存储区域，它可以由专门的硬件寄存器组成，但由于硬件的成本较高，容量也较小，一般仅用在对速度要求非常高的场合，如存储器管理中所用的联想存储器；设备控制器中用的数据缓冲区等。

在一般情况下，更多的是利用**内存**作为缓冲区。

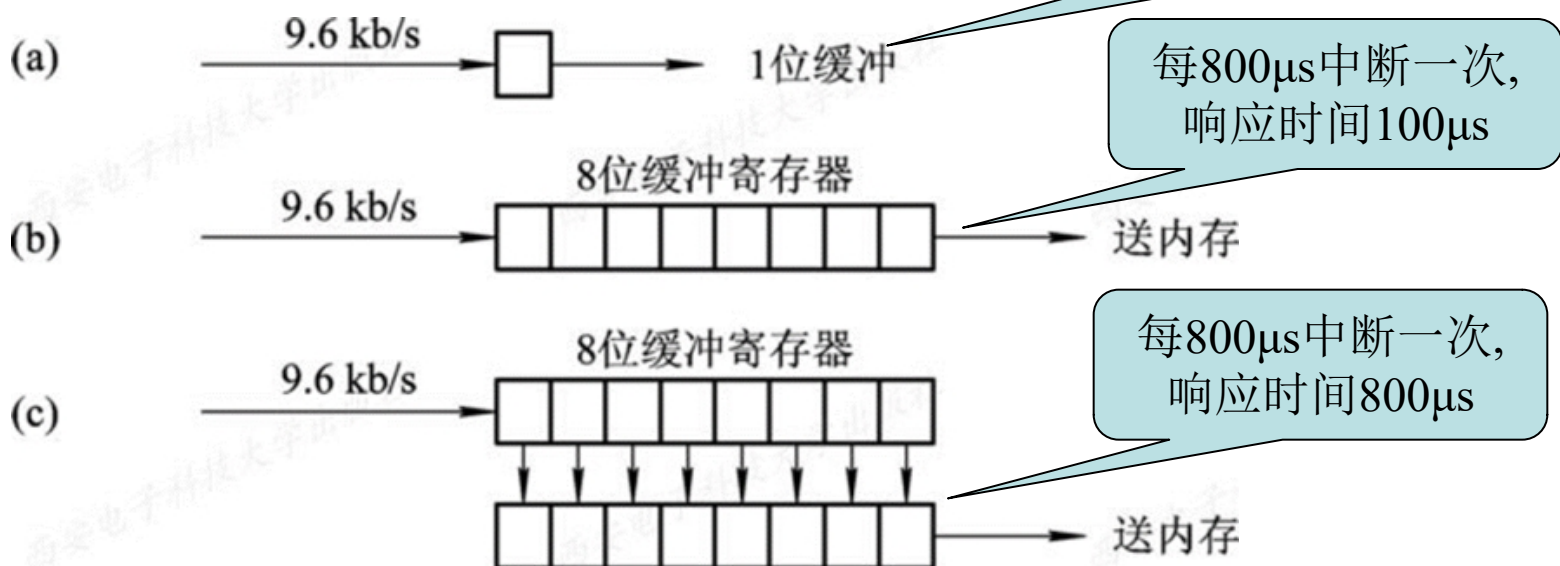
## 6.7.1 缓冲的引入

引入缓冲区的原因有很多，可归结为以下几点：

(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

限制。





(3) 解决生产者和消费者之间交换的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，提高系统的吞吐量和设备的利用率。



## 6.7.2 单缓冲区和双缓冲区

### 1. 单缓冲区(Single Buffer)

在单缓冲情况下，每当用户进程发出一I/O请求时，操作系统便在主存中为之分配一缓冲区。

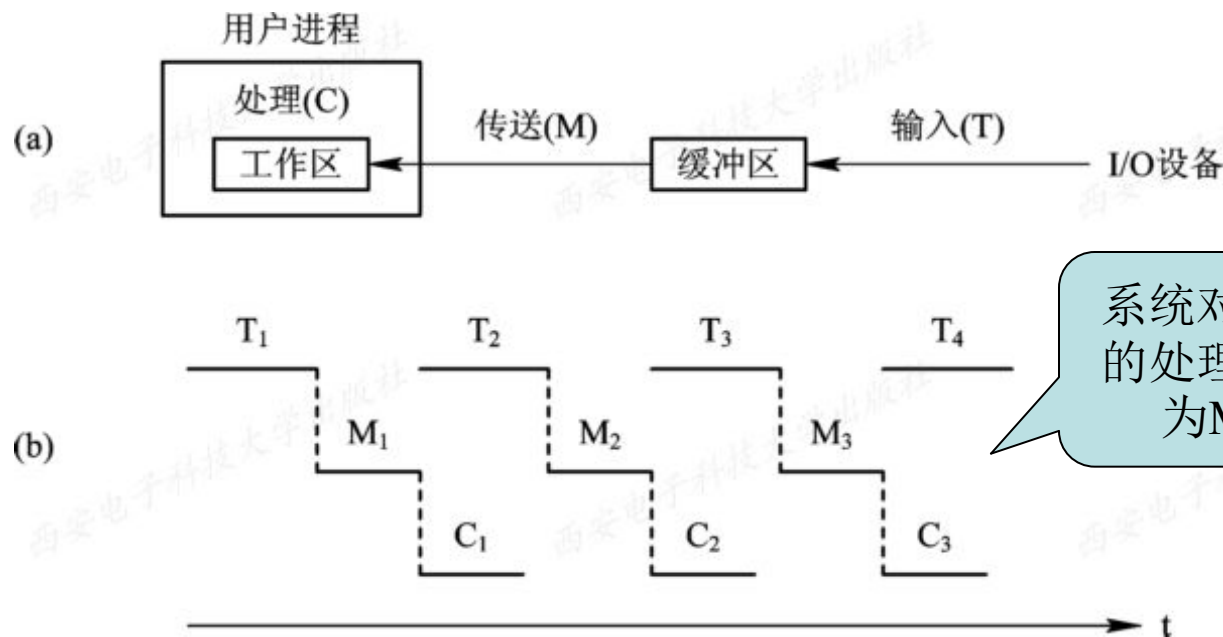
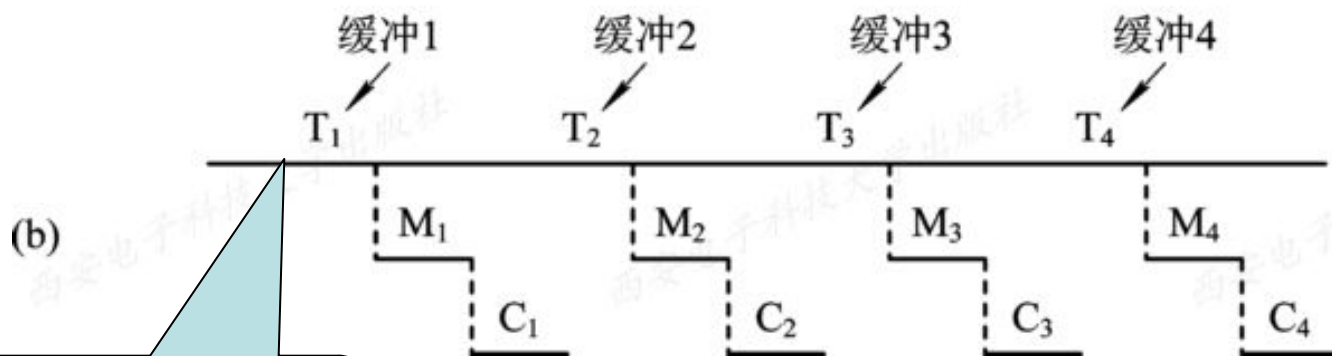
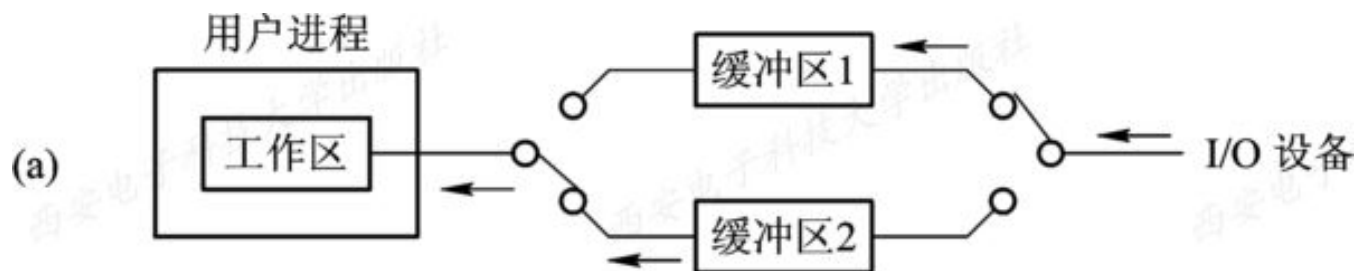


图6-23 单缓冲工作示意图

## 2. 双缓冲区(Double Buffer)

由于缓冲区是共享资源，生产者与消费者在使用缓冲区时必须**互斥**。如果消费者尚未取走缓冲区中的数据，即使生产者又生产出新的数据，也无法将它送入缓冲区，生产者等待。如果为生产者与消费者设置了**两个缓冲区**，便能解决这一问题。



如果  $C < T$ ，可以使块设备连续输入

如果  $C > T$ ，可以使CPU不必等待设备输入

图6-24 双缓冲工作示意图



如果在实现两台机器之间的通信时仅为它们配置了单缓冲，那么，它们之间在任一时刻都只能实现**单方向的数据传输**。为了实现**双向数据传输**，必须在两台机器中都设置两个缓冲区，一个用作发送缓冲区，另一个用作接收缓冲区。

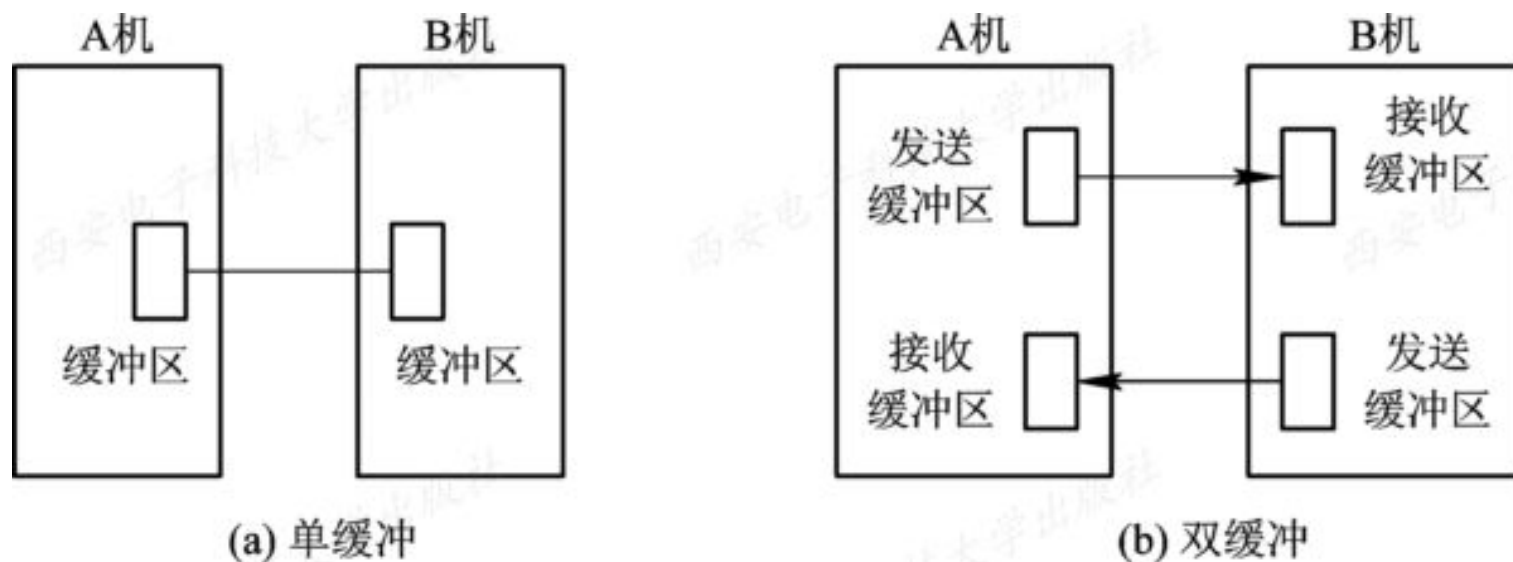


图6-25 双机通信时缓冲区的设置





### 6.7.3 环形缓冲区

#### 1. 环形缓冲区的组成

(1) 多个缓冲区。在环形缓冲中包括多个缓冲区，其每个缓冲区的大小相同。作为输入的多缓冲区可分为三种类型：用于装输入数据的**空缓冲区R**、已装满数据的**缓冲区G**以及计算进程正在使用的**现行工作缓冲区C**。

Nexti: 空缓冲区指针（输入进程使用）

Nextg: 可用缓冲区指针（计算进程使用）

Current: 当前计算进程正在使用的缓冲区

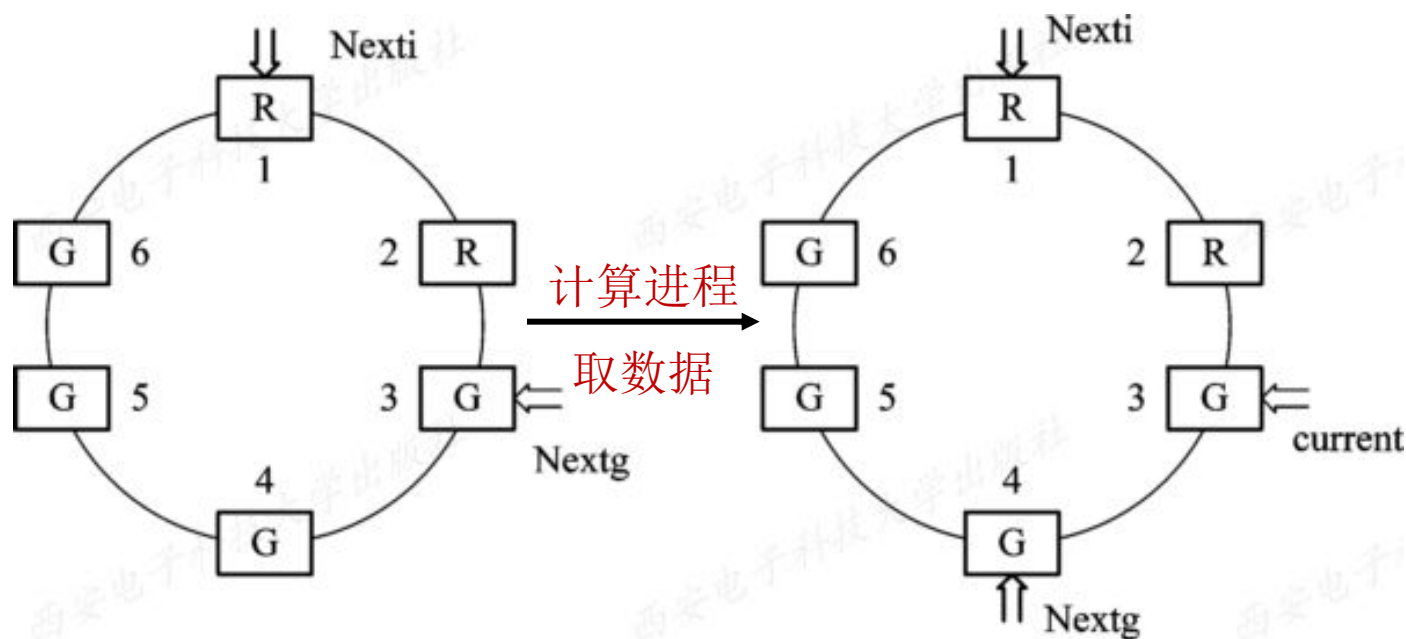


图6-26 环形缓冲区



## 2. 环形缓冲区的使用

计算进程和输入进程可利用下述两个过程来使用形环缓冲区。

(1) Getbuf过程：当计算进程要使用缓冲区中的数据时，可调用Getbuf过程。

(2) Releasebuf过程：当计算进程把C缓冲区中的数据提取完毕时，便调用Releasebuf过程，将缓冲区C释放。

### 3. 进程之间的同步问题

使用输入循环缓冲，可使输入进程和计算进程并行执行。相应地，指针Nexti和指针Nextg将不断地沿着顺时针方向移动，这样就可能出现下述两种情况：

(1) Nexti指针追赶上Nextg指针：输入进程输入数据的速度大于计算进程处理数据的速度，已把全部可用的空缓冲区装满，再无缓冲区可用。此时，**输入进程应阻塞**，直到出现空缓冲区后，由计算进程唤醒。

(2) Nextg指针追赶上Nexti指针：输入数据的速度低于计算进程处理数据的速度。此时，**计算进程应阻塞**。直至输入进程装满某个缓冲区后，才被唤醒。





## 6.7.4 缓冲池(Buffer Pool)

### 1. 缓冲池的组成

缓冲池管理着多个缓冲区，每个缓冲区由用于标识和管理的**缓冲首部**以及用于存放数据的**缓冲体**两部分组成。

缓冲首部一般包括**缓冲区号**、**设备号**、**设备上的数据块号**、**同步信号量**以及**队列链接指针**等。



为了管理上的方便，一般将缓冲池中具有相同类型的缓冲区链接成一个队列，于是可形成以下三个队列：

(1) 空白缓冲队列**emq**：空缓冲区所链成的队列，队首指针**F(emq)**和队尾指针**L(emq)**。

(2) 输入队列**inq**：装满输入数据的缓冲区所链成的队列，队首指针**F(inq)**和队尾指针**L(inq)**。

(3) 输出队列**outq**：装满输出数据的缓冲区所链成的队列，其队首指针**F(outq)**和队尾指针**L(outq)**。

## 2. Getbuf过程和Putbuf过程

对队列进行操作的两个过程：

(1) **Addbuf(type, number)过程**：用于将由参数number所指示的缓冲区B挂在type队列上。

(2) **Takebuf(type)过程**：用于从type所指示的队列的队首摘下一个缓冲区。

◆ **互斥信号量MS(type)**：为使诸进程能互斥地访问缓冲池队列，可为每一队列设置一个。

◆ **资源信号量RS(type)**：为了保证诸进程同步地使用缓冲区，又为每个缓冲队列设置了一个。



既可实现互斥又可保证同步的Getbuf过程和Putbuf过程描述如下：

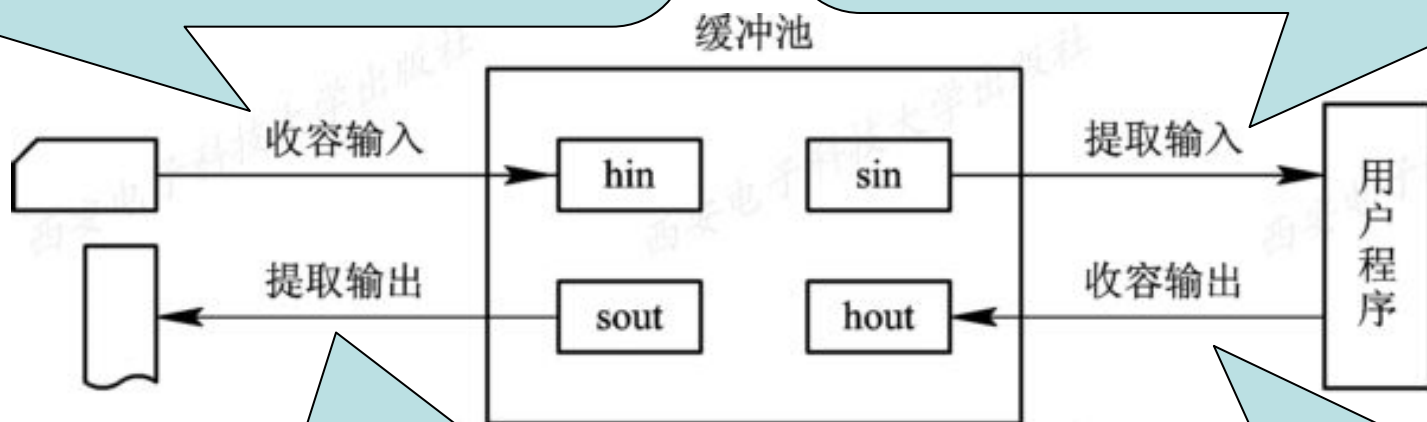
```
void Getbuf(unsigned type)  
{  
    Wait(RS(type));  
    Wait(MS(type));  
    B(number) = Takebuf(type);  
    Signal(MS(type));  
}
```

```
Void Putbuf(type, number)  
{  
    Wait(MS(type));  
    Addbuf(type, number);  
    Signal(MS(type));  
    Signal(RS(type));  
}
```



在**输入进程**需要输入数据时，便从空缓冲队列的队首摘下一空缓冲区，把它作为收容输入工作缓冲区。然后，把数据输入其中，装满后再该缓冲区挂在输入队列上。

当**计算进程**需要输入数据时，便从输入队列的队首取得一个缓冲区，作为提取输入工作缓冲区，计算进程从中提取数据。计算进程用完该数据后，再将该缓冲区挂到空缓冲队列上。



由**输出进程**从输出队列的队首取得一装满输出数据的缓冲区，作为提取输出工作缓冲区。在数据提取完后，将该缓冲区挂在空缓冲队列末尾。

当**计算进程**需要输出数据时，便从空缓冲队列的队首取得一个空缓冲区，作为收容输出工作缓冲区。当其中装满输出数据后，再将该缓冲区挂在输出队列队尾。

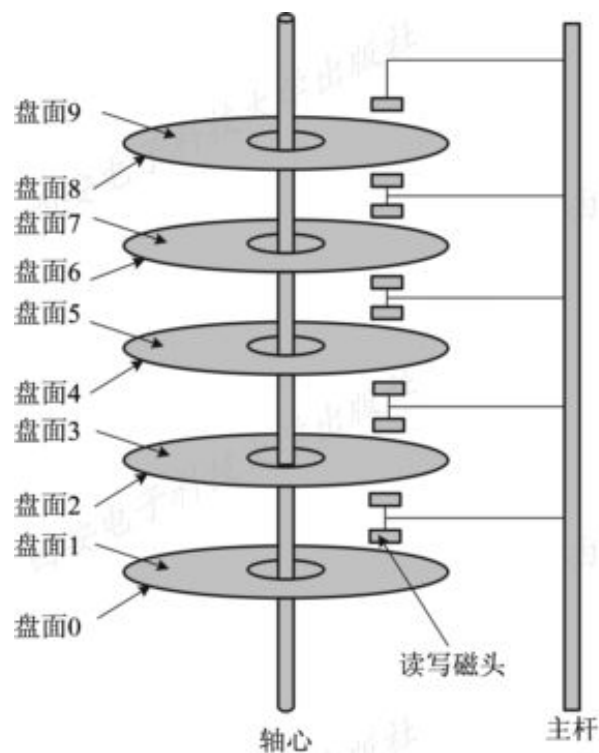
## 6.8 磁盘存储器的性能和调度

### 6.8.1 磁盘性能简述

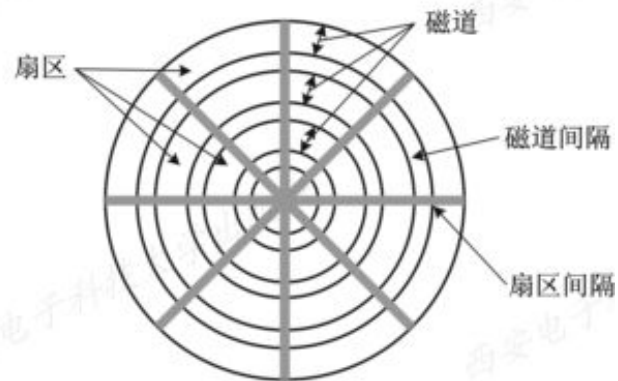
#### 1. 数据的组织和格式

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)，每个盘面上有若干个**磁道**(Track)，磁道之间留有必要的间隙(Gap)。在每条磁道上可存储相同数目的二进制位。内层磁道的密度较外层磁道的密度高。

每条磁道又被逻辑上划分成若干个**扇区**(sectors)，软盘大约为8~32个扇区，硬盘则可多达数百个。一个扇区称为一个盘块(或数据块)，常常叫做**磁盘扇区**。各扇区之间保留一定的间隙。



(a) 磁盘驱动器的结构



(b) 磁盘的数据布局

图6-28 磁盘的结构和布局



一个物理记录存储在一个扇区上，磁盘上存储的物理记录块数目是由扇区数、磁道数以及磁盘面数所决定的。

例如，一个10 GB容量的磁盘，有8个双面可存储盘片，共16个存储面(盘面)，每面有16 383个磁道(也称柱面)，63个扇区。



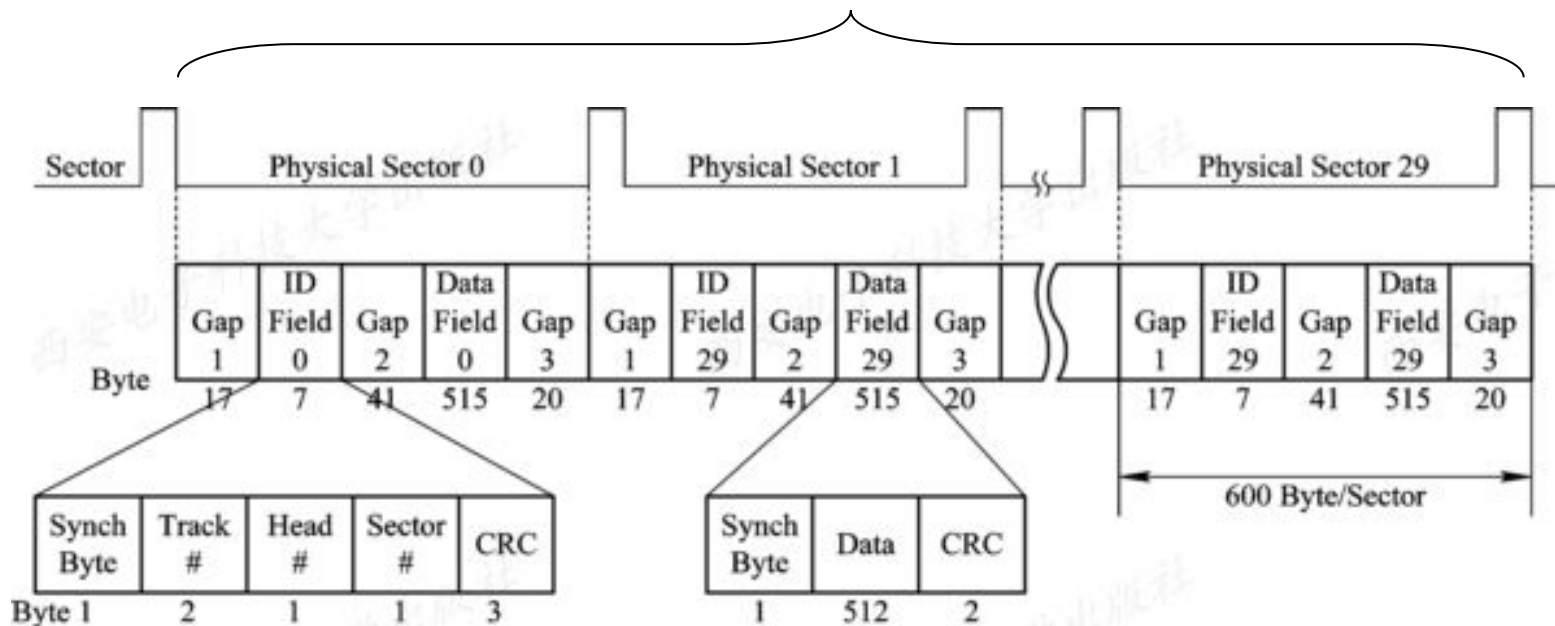


为了提高磁盘的存储容量，充分利用磁盘外面磁道的存储能力，现代磁盘不再把内外磁道划分为相同数目的扇区，而是利用外层磁道容量较内层磁道大的特点，将盘面划分成若干条**环带**，使得同一环带内的所有磁道具有相同的扇区数。

显然，外层环带的磁道拥有较内层环带的磁道更多的扇区。为了减少这种磁道和扇区在盘面分布的几何形式变化对驱动程序的影响，大多数现代磁盘都隐藏了这些细节，向操作系统提供虚拟几何的磁盘规格，而不是实际的物理几何规格。



每个磁道30个扇区，每个扇区600个字节



Synch Byte: 定界符，具备特定的位图像

Track #: 磁道号

Head #: 磁头号

Sectors #: 扇区号

Data: 数据

图6-29 磁盘的格式化



## 2. 磁盘的类型

对于磁盘，可以从不同的角度进行分类。最常见的有：将磁盘分成硬盘和软盘、单片盘和多片盘、固定头磁盘和活动头(移动头)磁盘等。

(1) 固定头磁盘：这种磁盘在**每条磁道上都有一读/写磁头**，所有的磁头都被装在一刚性磁臂中。通过这些磁头可访问所有各磁道，并进行并行读/写，有效地提高了磁盘的I/O速度。这种结构的磁盘主要用于大容量磁盘上。。

(2) 移动头磁盘：**每一个盘面仅配有一个磁头**，也被装入磁臂中。为能访问该盘面上的所有磁道，该磁头必须能移动以进行寻道。可见，移动磁头仅能以串行方式读/写，致使其I/O速度较慢；但由于其结构简单，故仍广泛应用于中小型磁盘设备中。。

### 3. 磁盘访问时间

#### 1) 寻道时间 $T_s$

这是指把磁臂(磁头)移动到指定磁道上所经历的时间。该时间是启动磁臂的时间 $s$ 与磁头移动 $n$ 条磁道所花费的时间之和, 即

$$T_s = m \times n + s$$

其中,  $m$ 是一常数, 与磁盘驱动器的速度有关。对于一般磁盘,  $m=0.2$ ; 对于高速磁盘,  $m \leq 0.1$ , 磁臂的启动时间约为2 ms。这样, 对于一般的温盘, 其寻道时间将随寻道距离的增加而增大, 大体上是5~30 ms。





## 2) 旋转延迟时间 $T_r$

这是指定扇区移动到磁头下面所经历的时间。不同的磁盘类型中，旋转速度至少相差一个数量级，如软盘为300 r/min，硬盘一般为7200~15 000 r/min，甚至更高。



对于磁盘旋转延迟时间而言，如硬盘，旋转速度为15 000 r/min，每转需时4 ms，平均旋转延迟时间 $T_r$ 为2 ms；而软盘，其旋转速度为300 r/min或600 r/min，这样，平均 $T_r$ 为50~100 ms。

### 3) 传输时间 $T_t$

这是指把数据从磁盘读出或向磁盘写入数据所经历的时间。 $T_t$ 的大小与每次所读/写的字节数 $b$ 和旋转速度有关:



$$T_t = \frac{b}{rN}$$

其中,  $r$ 为磁盘每秒钟的转数;  $N$ 为一条磁道上的字节数, 当一次读/写的字节数相当于半条磁道上的字节数时,  $T_t$ 与 $T_r$ 相同。因此, 可将访问时间 $T_a$ 表示为



由上式可以看出，在访问时间中，寻道时间和旋转延迟时间基本上都与所读/写数据的多少无关，而且它通常占据了访问时间中的大头。

**因此，适当地集中数据(不要太零散)传输，将有利于提高传输效率。**



## 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。

本算法由于未对寻道进行优化，致使平均寻道时间可能较长,平均寻道距离较大。

有9个进程先后提出磁盘I/O请求

(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)

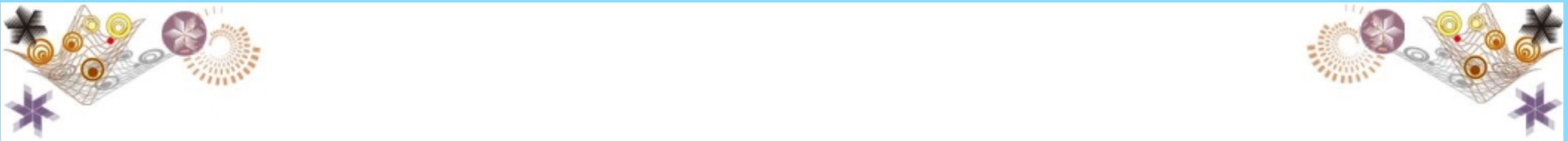
该算法选择这样的进程，其要求访问的磁道与当前磁头所在的磁道距离最近，以使每次的寻道时间最短，但这种算法不能保证平均寻道时间最短。

## 有9个进程先后提出磁盘I/O请求

(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

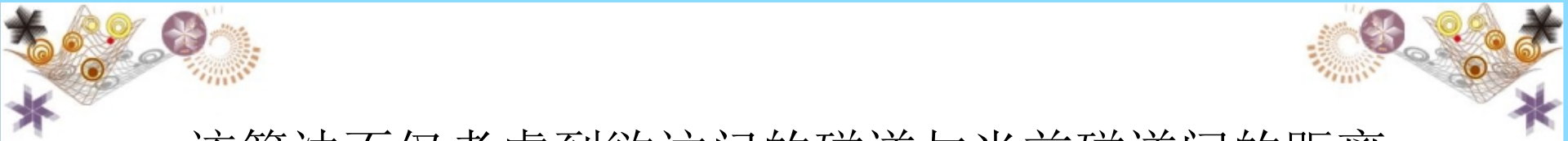
#### 1. 扫描(SCAN)算法

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。

在对SSTF算法略加修改后，则可防止低优先级进程出现“饥饿”现象。





该算法不仅考虑到欲访问的磁道与当前磁道间的距离，更优先考虑的是磁头当前的移动方向。

例如，当磁头正在自里向外移动时，**SCAN**算法所考虑的下一个访问对象，应是其欲访问的磁道既在当前磁道之外，又是距离最近的。这样自里向外地访问，直至再无更外的磁道需要访问时，才将磁臂换向为自外向里移动。这时，同样也是每次选择这样的进程来调度，即要访问的磁道在当前位置内距离最近者，这样，磁头又逐步地从外向里移动，直至再无更里面的磁道要访问，从而避免了出现“饥饿”现象。由于在这种算法中磁头移动的规律颇似电梯的运行，因而又常称之为**电梯调度算法**。

(从100# 磁道开始, 向磁道号 增加方向访问)	
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，故被广泛用于大、中、小型机器和网络中的磁盘调度。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。



为了减少这种延迟，CSCAN算法规定磁头单向移动，例如，只是自里向外移动，当磁头移到最外的磁道并访问后，磁头立即返回到最里的欲访问的磁道，亦即将最小磁道号紧接着最大磁道号构成循环，进行循环扫描。采用循环扫描方式后，上述请求进程的请求延迟将从原来的 $2T$ 减为 $T + S_{\max}$ ，其中， $T$ 为由里向外或由外向里单向扫描完要访问的磁道所需的寻道时间，而 $S_{\max}$ 是将磁头从最外面被访问的磁道直接移到最里面欲访问的磁道(或相反)的寻道时间。



(从100# 磁道开始, 向磁道号 增加方向访问)	
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。我们把这一现象称为“磁臂粘着”(Armstickiness)。在高密度磁盘上容易出现此情况。



N步SCAN算法是将磁盘请求队列分为若干个长度为N的子队列，磁盘调度将按FCFS算法依次处理这些子队列，而每处理一个子队列时，又是按照SCAN算法，当一个队列处理完后，再处理其他队列。当正在处理某子队列时，如果又出现新的磁盘I/O请求，则将其放入其他队列中，则可避免出现磁臂黏着现象。当N取很大时，N步扫描法会退化成SCAN算法，当N=1时，会退化成FCFS。





## 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。

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。

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。