计算机操作系统原理

--汤子瀛

第一章 操作系统引论

1.1 操作系统的目标和作用

1.1.1 操作系统的目标

目前存在着多种类型的OS，不同类型的OS，其目标各有所侧重。通常在计算机硬件上配置的OS，其目标有以下几点：

1. 方便性

2. 有效性

3. 可扩充性

4. 开放性

1.1.2 操作系统的作用

1.OS作为用户与计算机硬件系统之间的接口

OS作为用户与计算机硬件系统之间接口的含义是：OS处于用户与计算机硬件系统之间，用户通过OS来使用计算机系统。或者说，用户在OS帮助下，能够方便、快捷、安全、可靠地操纵计算机硬件和运行自己的程序。应注意，OS是一个系统软件，因而这种接口是软件接口。

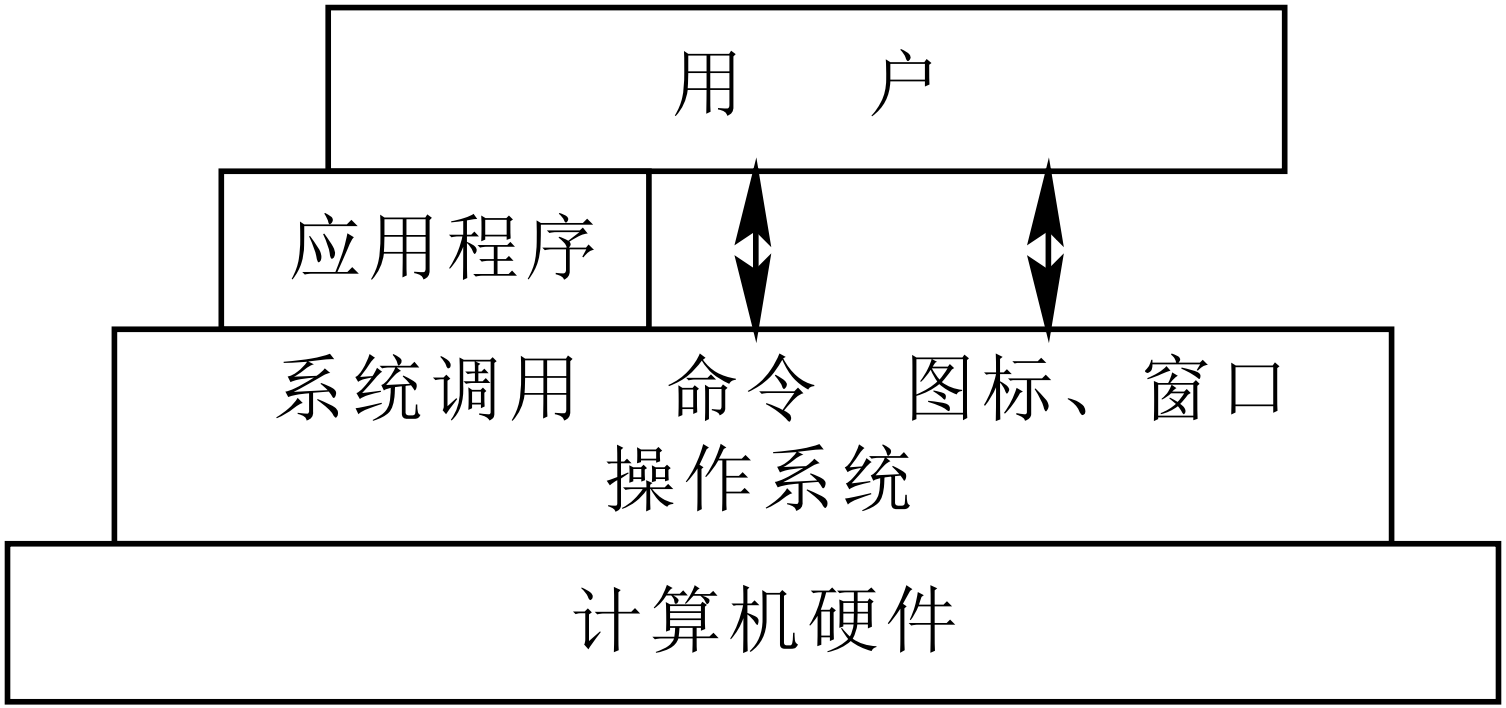


图 1-1 OS作为接口的示意图

(1) 命令方式。这是指由OS提供了一组联机命令(语言)， 用户可通过键盘输入有关命令，来直接操纵计算机系统。

(2) 系统调用方式。OS提供了一组系统调用，用户可在自己的应用程序中通过相应的系统调用，来操纵计算机。

(3) 图形、窗口方式。用户通过屏幕上的窗口和图标来操纵计算机系统和运行自己的程序。

2. OS作为计算机系统资源的管理者

在一个计算机系统中，通常都含有各种各样的硬件和软件资源。归纳起来可将资源分为四类：处理器、存储器、I/O设备以及信息(数据和程序)。相应地，OS的主要功能也正是针对这四类资源进行有效的管理，即：处理机管理，用于分配和控制处理机；存储器管理，主要负责内存的分配与回收；I/O设备管理，负责I/O设备的分配与操纵；文件管理，负责文件的存取、共享和保护。可见，OS确是计算机系统资源的管理者。事实上，当今世界上广为流行的一个关于OS作用的观点，正是把OS作为计算机系统的资源管理者。

3. OS用作扩充机器

对于一台完全无软件的计算机系统(即裸机)，即使其功能再强，也必定是难于使用的。如果我们在裸机上覆盖上一层I/O设备管理软件，用户便可利用它所提供的I/O命令，来进行数据输入和打印输出。此时用户所看到的机器， 将是一台比裸机功能更强、使用更方便的机器。通常把覆盖了软件的机器称为扩充机器或虚机器。如果我们又在第一层软件上再覆盖上一层文件管理软件，则用户可利用该软件提供的文件存取命令，来进行文件的存取。此时，用户所看到的是台功能更强的虚机器。如果我们又在文件管理软件上再覆盖一层面向用户的窗口软件，则用户便可在窗口环境下方便地使用计算机，形成一台功能更强的虚机器。

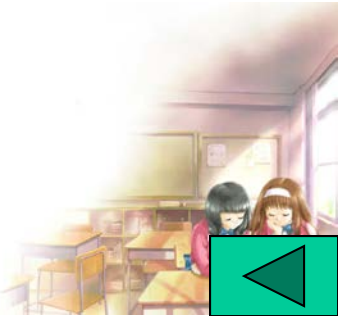
1.1.3 推动操作系统发展的主要动力

1. 不断提高计算机资源利用率

2. 方便用户

3. 器件的不断更新换代

4. 计算机体系结构的不断发展



1.2 操作系统的发展过程

1.2.1 无操作系统的计算机系统

1. 人工操作方式

从第一台计算机诞生(1945年)到50年代中期的计算机，属于第一代，这时还未出现OS。这时的计算机操作是由用户(即程序员)采用人工操作方式直接使用计算机硬件系统，即由程序员将事先已穿孔(对应于程序和数据)的纸带(或卡片)装入纸带输入机(或卡片输入机)，再启动它们将程序和数据输入计算机， 然后启动计算机运行。当程序运行完毕并取走计算结果后， 才让下一个用户上机。这种人工操作方式有以下两方面的缺点：

(1) 用户独占全机。 (2) CPU等待人工操作。

2. 脱机输入/输出(Off-Line I/O)方式

这种脱机I/O方式的主要优点如下：

(1) 减少了CPU的空闲时间。

(2) 提高I/O速度。

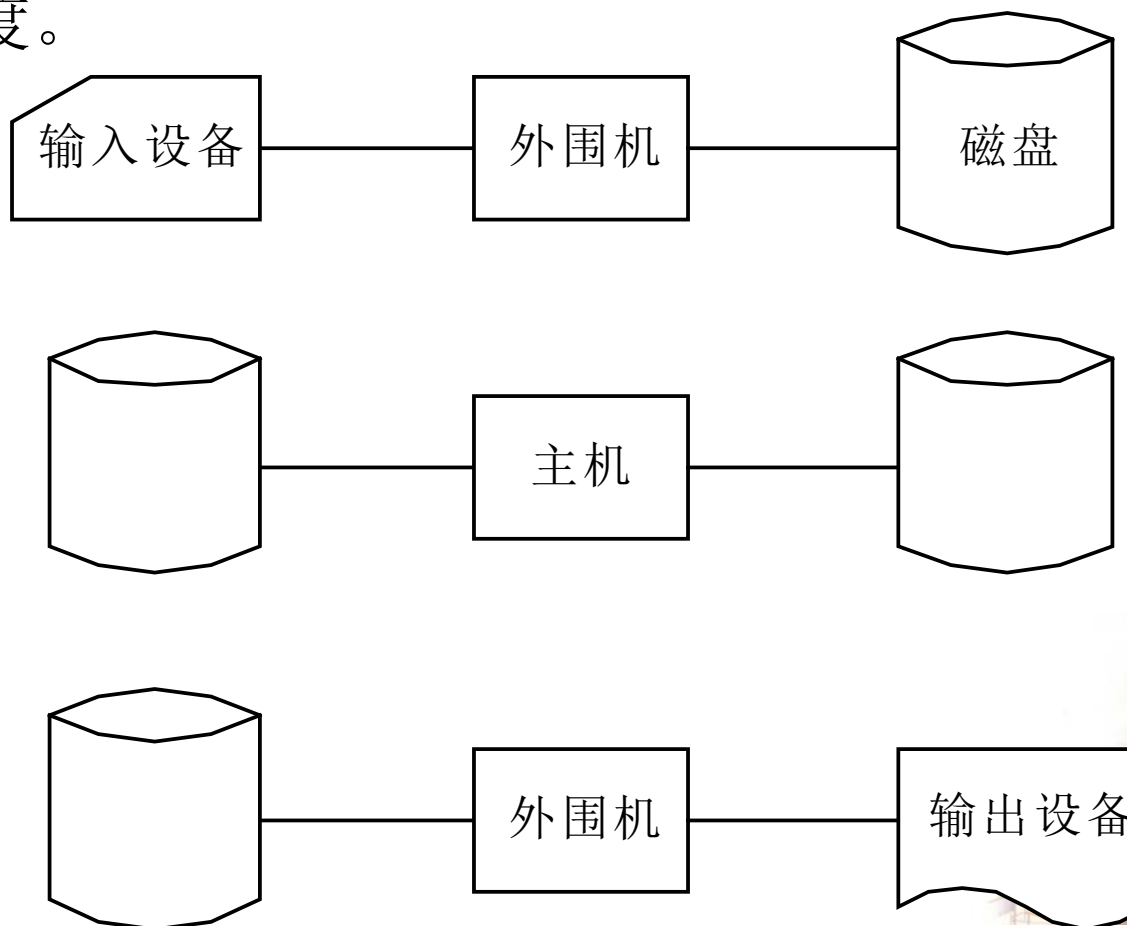


图 1-2 脱机I/O示意图

1.2.2 单道批处理系统

1. 单道批处理系统(Simple Batch Processing System)的处理过程

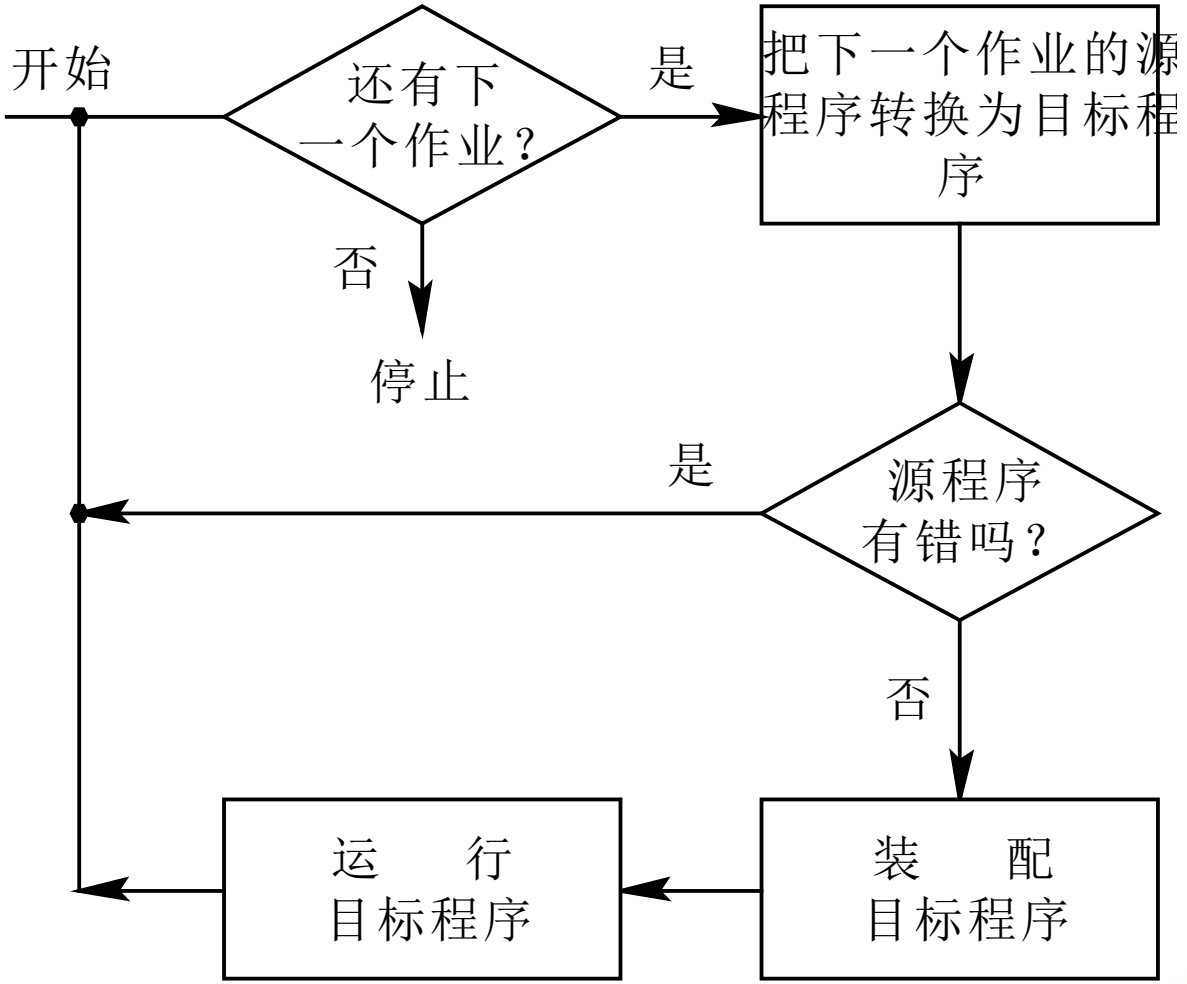


图 1-3 单道批处理系统的处理流程

2. 单道批处理系统的特征

单道批处理系统是最早出现的一种OS，严格地说，它只能算作是OS的前身而并非是现在人们所理解的OS。尽管如此，该系统比起人工操作方式的系统已有很大进步。 该系统的主要特征如下：

(1) 自动性。

(2) 顺序性。

(3) 单道性。

1.2.3 多道批处理系统

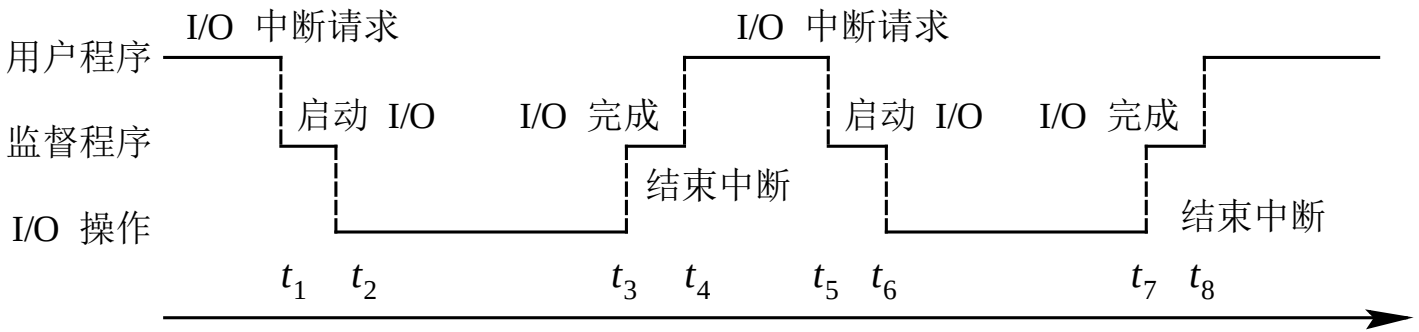
1. 多道程序设计的基本概念

在单道批处理系统中，内存中仅有一道作业，它无法充分利用系统中的所有资源，致使系统性能较差。为了进一步提高资源的利用率和系统吞吐量，在60年代中期又引入了多道程序设计技术，由此而形成了多道批处理系统(Multiprogrammed Batch Processing System)。在该系统中，用户所提交的作业都先存放在外存上并排成一个队列，称为“后备队列”；然后，由作业调度程序按一定的算法从后备队列中选择若干个作业调入内存，使它们共享CPU和系统中的各种资源。

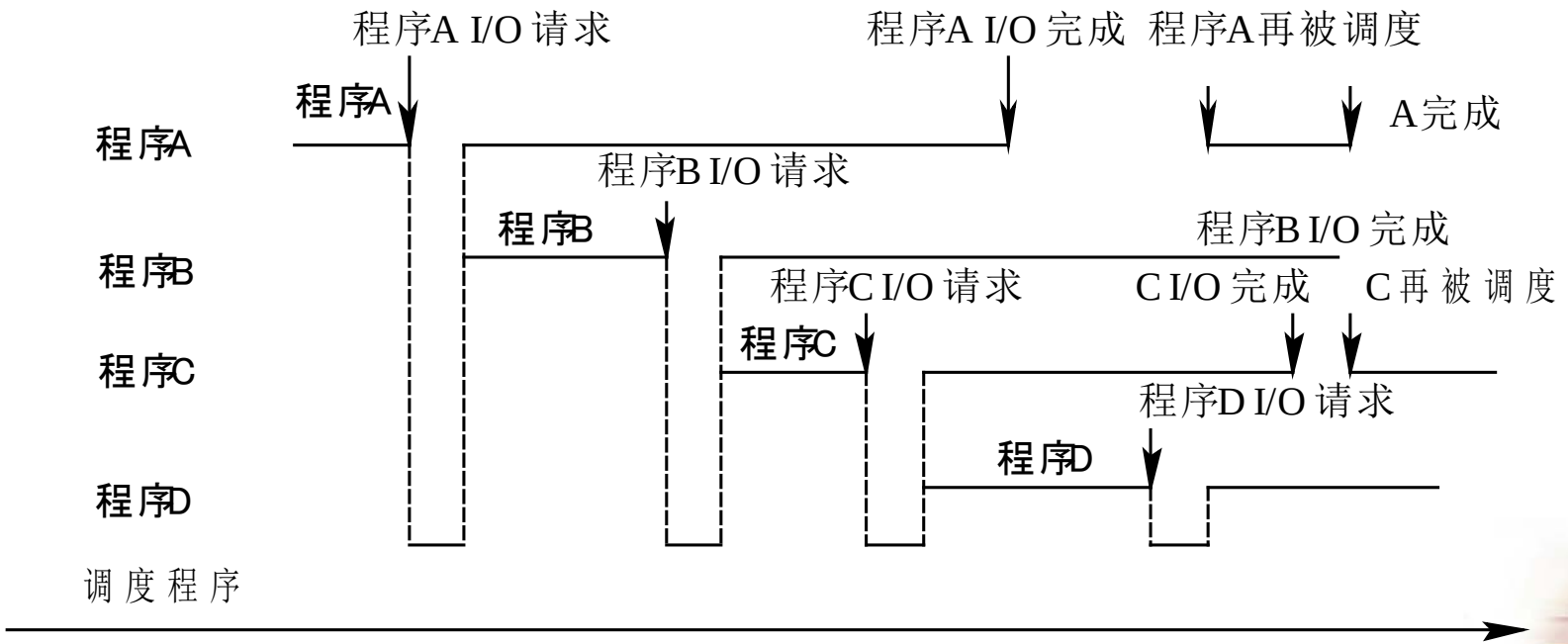
在OS中引入多道程序设计技术可带来以下好处：

(1) 提高CPU的利用率。

当内存中仅有一道程序时，每逢该程序在运行中发出I/O请求后，CPU空闲，必须在其I/O完成后才继续运行；尤其因I/O设备的低速性，更使CPU的利用率显著降低。图 1-4(*a*)示出了单道程序的运行情况，从图可以看出：在t_2~t_3、t_6~t_7时间间隔内CPU空闲。在引入多道程序设计技术后，由于同时在内存中装有若干道程序，并使它们交替地运行，这样，当正在运行的程序因I/O而暂停执行时，系统可调度另一道程序运行，从而保持了CPU处于忙碌状态。

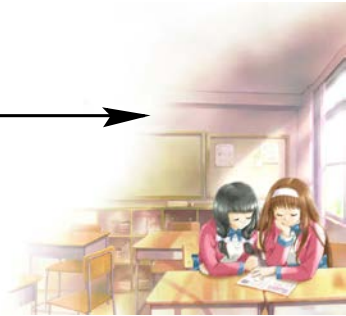


(*a*) 单道程序运行情况



(*b*) 四道程序运行情况

图 1-4 单道和多道程序运行情况



(2) 可提高内存和I/O设备利用率。为了能运行较大的作业，通常内存都具有较大容量，但由于80%以上的作业都属于中小型，因此在单道程序环境下，也必定造成内存的浪费。类似地，对于系统中所配置的多种类型的I/O设备， 在单道程序环境下也不能充分利用。如果允许在内存中装入多道程序， 并允许它们并发执行，则无疑会大大提高内存和I/O设备的利用率。

(3) 增加系统吞吐量。在保持CPU、I/O设备不断忙碌的同时，也必然会大幅度地提高系统的吞吐量，从而降低作业加工所需的费用。

2. 多道批处理系统的特征

(1) 多道性。

(2) 无序性。

(3) 调度性。

3. 多道批处理系统的优缺点

(1) 资源利用率高。

(2) 系统吞吐量大。

(3) 平均周转时间长。

(4) 无交互能力。

4. 多道批处理系统需要解决的问题

(1) 处理机管理问题。

(2) 内存管理问题。

(3) I/O设备管理问题。

(4) 文件管理问题。

(5) 作业管理问题。

1.2.4 分时系统

1. 分时系统(Time-Sharing System)的产生

如果说， 推动多道批处理系统形成和发展的主要动力，是提高资源利用率和系统吞吐量，那么，推动分时系统形成和发展的主要动力，则是用户的需求。或者说， 分时系统是为了满足用户需求所形成的一种新型OS。它与多道批处理系统之间，有着截然不同的性能差别。用户的需求具体表现在以下几个方面：

(1) 人—机交互。

(2) 共享主机。

(3) 便于用户上机。

2. 分时系统实现中的关键问题

为实现分时系统，其中，最关键的问题是如何使用户能与自己的作业进行交互，即当用户在自己的终端上键入命令时， 系统应能及时接收并及时处理该命令，再将结果返回给用户。 此后， 用户可继续键入下一条命令，此即人–机交互。应强调指出，即使有多个用户同时通过自己的键盘键入命令，系统也应能全部地及时接收并处理

(1) 及时接收。

(2) 及时处理。

3. 分时系统的特征

(1) 多路性。

(2) 独立性。

(3) 及时性。

(4) 交互性。

1.2.5 实时系统

所谓“实时”，是表示“及时”，而实时系统(Real-Time System)是指系统能及时(或即时)响应外部事件的请求，在规定的时间内完成对该事件的处理，并控制所有实时任务协调一致地运行。

1. 应用需求

(1) 实时控制。

(2) 实时信息处理。

2. 实时任务

1) 按任务执行时是否呈现周期性来划分

(1) 周期性实时任务。

(2) 非周期性实时任务。

外部设备所发出的激励信号并无明显的周期性，但都必须联系着一个截止时间(Deadline)。它又可分为：

① 开始截止时间—任务在某时间以前必须开始执行；

② 完成截止时间——任务在某时间以前必须完成。

2) 根据对截止时间的要求来划分

(1) 硬实时任务(hard real-time task)。系统必须满足任务对截止时间的要求，否则可能出现难以预测的结果。

(2) 软实时任务(Soft real-time task)。它也联系着一个截止时间， 但并不严格，若偶尔错过了任务的截止时间，对系统产生的影响也不会太大。

3. 实时系统与分时系统特征的比较

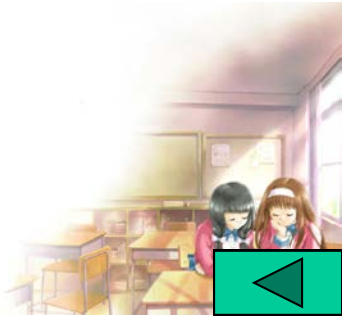
(1) 多路性。

(2) 独立性。

(3) 及时性。

(4) 交互性。

(5) 可靠性。



1.3 操作系统的基本特性

1.3.1 并发(Concurrence)

并行性和并发性是既相似又有区别的两个概念，并行性是指两个或多个事件在同一时刻发生；而并发性是指两个或多个事件在同一时间间隔内发生。在多道程序环境下，并发性是指在一段时间内，宏观上有多个程序在同时运行，但在单处理机系统中，每一时刻却仅能有一道程序执行，故微观上这些程序只能是分时地交替执行。倘若在计算机系统中有多个处理机，则这些可以并发执行的程序便可被分配到多个处理机上，实现并行执行，即利用每个处理机来处理一个可并发执行的程序，这样，多个程序便可同时执行。

1.3.2 共享(Sharing)

在操作系统环境下，所谓共享是指系统中的资源可供内存中多个并发执行的进程(线程)共同使用。由于资源属性的不同，进程对资源共享的方式也不同，目前主要有以下两种资源共享方式。

1. 互斥共享方式

系统中的某些资源，如打印机、磁带机，虽然它们可以提供给多个进程(线程)使用，但为使所打印或记录的结果不致造成混淆，应规定在一段时间内只允许一个进程(线程)访问该资源。为此，当一个进程A要访问某资源时，必须先提出请求， 如果此时该资源空闲，系统便可将之分配给请求进程A使用，此后若再有其它进程也要访问该资源时(只要A未用完)则必须等待。 仅当A进程访问完并释放该资源后， 才允许另一进程对该资源进行访问。我们把这种资源共享方式称为互斥式共享，而把在一段时间内只允许一个进程访问的资源称为临界资源或独占资源。 计算机系统中的大多数物理设备，以及某些软件中所用的栈、变量和表格，都属于临界资源，它们要求被互斥地共享。

2. 同时访问方式

系统中还有另一类资源，允许在一段时间内由多个进程“同时”对它们进行访问。这里所谓的“同时”往往是宏观上的，而在微观上，这些进程可能是交替地对该资源进行访问。典型的可供多个进程“同时”访问的资源是磁盘设备，一些用重入码编写的文件，也可以被“同时”共享，即若干个用户同时访问该文件。

并发和共享是操作系统的两个最基本的特征，它们又是互为存在的条件。一方面，资源共享是以程序(进程)的并发执行为条件的，若系统不允许程序并发执行，自然不存在资源共享问题；另一方面，若系统不能对资源共享实施有效管理，协调好诸进程对共享资源的访问，也必然影响到程序并发执行的程度，甚至根本无法并发执行。

1.3.3 虚拟(Virtual)

操作系统中的所谓“虚拟”，是指通过某种技术把一个物理实体变为若干个逻辑上的对应物。物理实体(前者)是实的， 即实际存在的；而后者是虚的，是用户感觉上的东西。相应地，用于实现虚拟的技术，称为虚拟技术。在OS中利用了多种虚拟技术，分别用来实现虚拟处理机、虚拟内存、 虚拟外部设备和虚拟信道等。

在虚拟处理机技术中，是通过多道程序设计技术，让多道程序并发执行的方法，来分时使用一台处理机的。此时， 虽然只有一台处理机，但它能同时为多个用户服务，使每个终端用户都认为是有一个CPU在专门为他服务。亦即， 利用多道程序设计技术，把一台物理上的CPU虚拟为多台逻辑上的CPU，也称为虚拟处理机，我们把用户所感觉到的CPU称为虚拟处理器。

类似地，可以通过虚拟存储器技术，将一台机器的物理存储器变为虚拟存储器，以便从逻辑上来扩充存储器的容量。此时，虽然物理内存的容量可能不大(如32 MB)，但它可以运行比它大得多的用户程序(如128 MB)。这使用户所感觉到的内存容量比实际内存容量大得多，认为该机器的内存至少也有128 MB。当然这时用户所感觉到的内存容量是虚的。我们把用户所感觉到的存储器称为虚拟存储器。

我们还可以通过虚拟设备技术，将一台物理I/O设备虚拟为多台逻辑上的I/O设备，并允许每个用户占用一台逻辑上的I/O设备，这样便可使原来仅允许在一段时间内由一个用户访问的设备(即临界资源)，变为在一段时间内允许多个用户同时访问的共享设备。例如，原来的打印机属于临界资源，而通过虚拟设备技术，可以把它变为多台逻辑上的打印机，供多个用户“同时”打印。此外，也可以把一条物理信道虚拟为多条逻辑信道(虚信道)。在操作系统中，虚拟的实现主要是通过分时使用的方法。显然，如果n是某物理设备所对应的虚拟的逻辑设备数，则虚拟设备的平均速度必然是物理设备速度的$1/n$ 。

1.3.4 异步性(Asynchronism)

在多道程序环境下，允许多个进程并发执行，但只有进程在获得所需的资源后方能执行。在单处理机环境下，由于系统中只有一个处理机，因而每次只允许一个进程执行，其余进程只能等待。当正在执行的进程提出某种资源要求时，如打印请求，而此时打印机正在为其它某进程打印，由于打印机属于临界资源，因此正在执行的进程必须等待，且放弃处理机，直到打印机空闲，并再次把处理机分配给该进程时，该进程方能继续执行。可见，由于资源等因素的限制，使进程的执行通常都不是“一气呵成”，而是以“停停走走”的方式运行。

内存中的每个进程在何时能获得处理机运行，何时又因提出某种资源请求而暂停，以及进程以怎样的速度向前推进，每道程序总共需多少时间才能完成，等等，都是不可预知的。由于各用户程序性能的不同，比如，有的侧重于计算而较少需要I/O；而又有的程序其计算少而I/O多，这样，很可能是先进入内存的作业后完成； 而后进入内存的作业先完成。或者说，进程是以人们不可预知的速度向前推进，此即进程的异步性。尽管如此，但只要运行环境相同，作业经多次运行，都会获得完全相同的结果。因此，异步运行方式是允许的，是操作系统的一个重要特征。

1.4 操作系统的主要功能

1.4.1 处理机管理功能

1. 进程控制

在传统的多道程序环境下，要使作业运行，必须先为它创建一个或几个进程，并为之分配必要的资源。当进程运行结束时，立即撤消该进程，以便能及时回收该进程所占用的各类资源。进程控制的主要功能是为作业创建进程、撤消已结束的进程，以及控制进程在运行过程中的状态转换。在现代OS中，进程控制还应具有为一个进程创建若干个线程的功能和撤消(终止)已完成任务的线程的功能。

2. 进程同步

为使多个进程能有条不紊地运行，系统中必须设置进程同步机制。进程同步的主要任务是为多个进程(含线程)的运行进行协调。有两种协调方式： ① 进程互斥方式， 这是指诸进程(线程)在对临界资源进行访问时， 应采用互斥方式；② 进程同步方式，指在相互合作去完成共同任务的诸进程(线程)间，由同步机构对它们的执行次序加以协调。

为了实现进程同步，系统中必须设置进程同步机制。最简单的用于实现进程互斥的机制，是为每一个临界资源配置一把锁W，当锁打开时，进程(线程)可以对该临界资源进行访问；而当锁关上时，则禁止进程(线程)访问该临界资源。

3. 进程通信

在多道程序环境下，为了加速应用程序的运行，应在系统中建立多个进程，并且再为一个进程建立若干个线程，由这些进程(线程)相互合作去完成一个共同的任务。而在这些进程(线程)之间，又往往需要交换信息。例如，有三个相互合作的进程，它们是输入进程、计算进程和打印进程。输入进程负责将所输入的数据传送给计算进程；计算进程利用输入数据进行计算，并把计算结果传送给打印进程；最后，由打印进程把计算结果打印出来。进程通信的任务就是用来实现在相互合作的进程之间的信息交换。

当相互合作的进程(线程)处于同一计算机系统时，通常在它们之前是采用直接通信方式，即由源进程利用发送命令直接将消息(message)挂到目标进程的消息队列上，以后由目标进程利用接收命令从其消息队列中取出消息。

4. 调度

在后备队列上等待的每个作业，通常都要经过调度才能执行。在传统的操作系统中，包括作业调度和进程调度两步。作业调度的基本任务，是从后备队列中按照一定的算法，选择出若干个作业，为它们分配其必需的资源(首先是分配内存)。 在将它们调入内存后，便分别为它们建立进程，使它们都成为可能获得处理机的就绪进程，并按照一定的算法将它们插入就绪队列。而进程调度的任务，则是从进程的就绪队列中选出一新进程，把处理机分配给它，并为它设置运行现场， 使进程投入执行。值得提出的是，在多线程OS中，通常是把线程作为独立运行和分配处理机的基本单位，为此，须把就绪线程排成一个队列，每次调度时，是从就绪线程队列中选出一个线程，把处理机分配给它。

1.4.2 存储器管理功能

1. 内存分配

OS在实现内存分配时，可采取静态和动态两种方式。在静态分配方式中，每个作业的内存空间是在作业装入时确定的；在作业装入后的整个运行期间， 不允许该作业再申请新的内存空间，也不允许作业在内存中“移动”； 在动态分配方式中，每个作业所要求的基本内存空间， 也是在装入时确定的，但允许作业在运行过程中，继续申请新的附加内存空间，以适应程序和数据的动态增涨，也允许作业在内存中“移动”。

为了实现内存分配，在内存分配的机制中应具有这样的结构和功能：

① 内存分配数据结构， 该结构用于记录内存空间的使用情况， 作为内存分配的依据；

② 内存分配功能，系统按照一定的内存分配算法，为用户程序分配内存空间；

③ 内存回收功能，系统对于用户不再需要的内存，通过用户的释放请求，去完成系统的回收功能。

2. 内存保护

内存保护的主要任务，是确保每道用户程序都只在自己的内存空间内运行，彼此互不干扰。

为了确保每道程序都只在自己的内存区中运行，必须设置内存保护机制。一种比较简单的内存保护机制，是设置两个界限寄存器，分别用于存放正在执行程序的上界和下界。系统须对每条指令所要访问的地址进行检查，如果发生越界，便发出越界中断请求，以停止该程序的执行。如果这种检查完全用软件实现，则每执行一条指令，便须增加若干条指令去进行越界检查，这将显著降低程序的运行速度。因此，越界检查都由硬件实现。当然， 对发生越界后的处理， 还须与软件配合来完成。

3. 地址映射

一个应用程序(源程序)经编译后，通常会形成若干个目标程序；这些目标程序再经过链接便形成了可装入程序。这些程序的地址都是从“0”开始的，程序中的其它地址都是相对于起始地址计算的； 由这些地址所形成的地址范围称为“地址空间”， 其中的地址称为“逻辑地址”或“相对地址”。此外，由内存中的一系列单元所限定的地址范围称为“内存空间”， 其中的地址称为“物理地址”。

在多道程序环境下，每道程序不可能都从“0”地址开始装入(内存)， 这就致使地址空间内的逻辑地址和内存空间中的物理地址不相一致。使程序能正确运行，存储器管理必须提供地址映射功能，以将地址空间中的逻辑地址转换为内存空间中与之对应的物理地址。该功能应在硬件的支持下完成。

4. 内存扩充

存储器管理中的内存扩充任务，并非是去扩大物理内存的容量，而是借助于虚拟存储技术，从逻辑上去扩充内存容量，使用户所感觉到的内存容量比实际内存容量大得多；或者是让更多的用户程序能并发运行。这样，既满足了用户的需要，改善了系统的性能，又基本上不增加硬件投资。为了能在逻辑上扩充内存，系统必须具有内存扩充机制，用于实现下述各功能：

(1) 请求调入功能。

(2) 置换功能。

1.4.3 设备管理功能

设备管理用于管理计算机系统中所有的外围设备， 而设备管理的主要任务是，完成用户进程提出的I/O请求；为用户进程分配其所需的I/O设备；提高CPU和I/O设备的利用率；提高I/O速度；方便用户使用I/O设备。为实现上述任务，设备管理应具有缓冲管理、设备分配和设备处理，以及虚拟设备等功能。

1. 缓冲管理

CPU运行的高速性和I/O低速性间的矛盾自计算机诞生时起便已存在。 而随着CPU速度迅速、大幅度的提高，使得此矛盾更为突出，严重降低了CPU的利用率。如果在I/O设备和CPU之间引入缓冲，则可有效地缓和CPU和I/O设备速度不匹配的矛盾，提高CPU的利用率，进而提高系统吞吐量。因此，在现代计算机系统中， 都毫无例外地在内存中设置了缓冲区，而且还可通过增加缓冲区容量的方法，来改善系统的性能。

最常见的缓冲区机制有单缓冲机制、能实现双向同时传送数据的双缓冲机制，以及能供多个设备同时使用的公用缓冲池机制。

2. 设备分配

设备分配的基本任务，是根据用户进程的I/O请求、系统的现有资源情况以及按照某种设备分配策略，为之分配其所需的设备。如果在I/O设备和CPU之间，还存在着设备控制器和I/O通道时，还须为分配出去的设备分配相应的控制器和通道。

为了实现设备分配，系统中应设置设备控制表、控制器控制表等数据结构，用于记录设备及控制器的标识符和状态。据这些表格可以了解指定设备当前是否可用，是否忙碌，以供进行设备分配时参考。在进行设备分配时，应针对不同的设备类型而采用不同的设备分配方式。对于独占设备(临界资源)的分配，还应考虑到该设备被分配出去后，系统是否安全。 设备使用完后，还应立即由系统回收。

3. 设备处理

设备处理程序又称为设备驱动程序。其基本任务是用于实现CPU和设备控制器之间的通信，即由CPU向设备控制器发出I/O命令，要求它完成指定的I/O操作；反之由CPU接收从控制器发来的中断请求，并给予迅速的响应和相应的处理。

处理过程是：设备处理程序首先检查I/O请求的合法性，了解设备状态是否是空闲的，了解有关的传递参数及设置设备的工作方式。然后，便向设备控制器发出I/O命令，启动I/O设备去完成指定的I/O操作。设备驱动程序还应能及时响应由控制器发来的中断请求，并根据该中断请求的类型，调用相应的中断处理程序进行处理。对于设置了通道的计算机系统，设备处理程序还应能根据用户的I/O请求，自动地构成通道程序。

1.4.4 文件管理功能

1. 文件存储空间的管理

由文件系统对诸多文件及文件的存储空间，实施统一的管理。其主要任务是为每个文件分配必要的外存空间，提高外存的利用率，并能有助于提高文件系统的运行速度。

为此，系统应设置相应的数据结构，用于记录文件存储空间的使用情况，以供分配存储空间时参考；系统还应具有对存储空间进行分配和回收的功能。为了提高存储空间的利用率，对存储空间的分配，通常是采用离散分配方式，以减少外存零头，并以盘块为基本分配单位。盘块的大小通常为512 B~8 KB。

2. 目录管理

为了使用户能方便地在外存上找到自己所需的文件，通常由系统为每个文件建立一个目录项。目录项包括文件名、文件属性、文件在磁盘上的物理位置等。由若干个目录项又可构成一个目录文件。目录管理的主要任务，是为每个文件建立其目录项，并对众多的目录项加以有效的组织，以实现方便的按名存取。即用户只须提供文件名， 即可对该文件进行存取。其次，目录管理还应能实现文件共享，这样，只须在外存上保留一份该共享文件的副本。此外，还应能提供快速的目录查询手段，以提高对文件的检索速度。

3. 文件的读/写管理和保护

(1) 文件的读/写管理。该功能是根据用户的请求，从外存中读取数据；或将数据写入外存。在进行文件读(写)时，系统先根据用户给出的文件名，去检索文件目录，从中获得文件在外存中的位置。然后，利用文件读(写)指针，对文件进行读(写)。一旦读(写)完成，便修改读(写)指针，为下一次读(写)做好准备。由于读和写操作不会同时进行，故可合用一个读/写指针。

(2) 文件保护。① 防止未经核准的用户存取文件； ② 防止冒名顶替存取文件； ③ 防止以不正确的方式使用文件。

1.4.5 用户接口

1. 命令接口

(1) 联机用户接口。这是为联机用户提供的，它由一组键盘操作命令及命令解释程序所组成。当用户在终端或控制台上每键入一条命令后，系统便立即转入命令解释程序，对该命令加以解释并执行该命令。在完成指定功能后，控制又返回到终端或控制台上，等待用户键入下一条命令。这样，用户可通过先后键入不同命令的方式，来实现对作业的控制，直至作业完成。

(2) 脱机用户接口。该接口是为批处理作业的用户提供的，故也称为批处理用户接口。该接口由一组作业控制语言JCL组成。批处理作业的用户不能直接与自己的作业交互作用，只能委托系统代替用户对作业进行控制和干预。这里的作业控制语言JCL便是提供给批处理作业用户的、为实现所需功能而委托系统代为控制的一种语言。用户用JCL把需要对作业进行的控制和干预，事先写在作业说明书上，然后将作业连同作业说明书一起提供给系统。当系统调度到该作业运行时，又调用命令解释程序，对作业说明书上的命令，逐条地解释执行。如果作业在执行过程中出现异常现象，系统也将根据作业说明书上的指示进行干预。这样，作业一直在作业说明书的控制下运行，直至遇到作业结束语句时，系统才停止该作业的运行。

2. 程序接口

该接口是为用户程序在执行中访问系统资源而设置的，是用户程序取得操作系统服务的惟一途径。它是由一组系统调用组成，每一个系统调用都是一个能完成特定功能的子程序，每当应用程序要求OS提供某种服务(功能)时，便调用具有相应功能的系统调用。早期的系统调用都是用汇编语言提供的，只有在用汇编语言书写的程序中，才能直接使用系统调用；但在高级语言以及C语言中，往往提供了与各系统调用一一对应的库函数，这样，应用程序便可通过调用对应的库函数来使用系统调用。但在近几年所推出的操作系统中，如UNIX、OS/2版本中，其系统调用本身已经采用C语言编写，并以函数形式提供，故在用C语言编制的程序中， 可直接使用系统调用。

3. 图形接口

用户虽然可以通过联机用户接口来取得OS的服务，但这时要求用户能熟记各种命令的名字和格式，并严格按照规定的格式输入命令，这既不方便又花时间，于是，图形用户接口便应运而生。图形用户接口采用了图形化的操作界面，用非常容易识别的各种图标(icon)来将系统的各项功能、各种应用程序和文件，直观、逼真地表示出来。用户可用鼠标或通过菜单和对话框，来完成对应用程序和文件的操作。此时用户已完全不必像使用命令接口那样去记住命令名及格式，从而把用户从繁琐且单调的操作中解脱出来。

1.5 操作系统的结构设计

1.5.1 软件工程的基本概念

1. 软件的含义

所谓软件，是指当计算机运行时，能提供所要求的功能和性能的指令和程序的集合，该程序能够正确地处理信息的数据结构；作为规范软件，还应具有描述程序功能需求以及程序如何操作使用的文档。如果说，硬件是物理部件， 那么，软件则是一种逻辑部件，它具有与硬件完全不同的特点。

2. 软件工程的含义

软件工程是指运用系统的、规范的和可定量的方法，来开发、运行和维护软件；或者说，是采用工程的概念、原理、技术和方法，来开发与维护软件，其目的是为了解决在软件开发中所出现的编程随意、软件质量不可保证以及维护困难等问题。

1.5.2 传统的操作系统结构

操作系统是一个十分复杂的大型软件。为了控制该软件的复杂性，在开发OS时，先后引入了分解、模块化、 抽象和隐蔽等方法。开发方法的不断发展，促进了OS结构的更新换代。这里，我们把第一代至第三代的OS结构，称为传统的OS结构，而把微内核的OS结构称为现代OS结构。

1. 无结构操作系统

在早期开发操作系统时，设计者只是把他的注意力放在功能的实现和获得高的效率上，缺乏首尾一致的设计思想。此时的OS是为数众多的一组过程的集合，各过程之间可以相互调用，在操作系统内部不存在任何结构，因此，这种OS是无结构的，也有人把它称为整体系统结构。

此时程序设计的技巧，只是如何编制紧凑的程序，以便于有效地利用内存、对GOTO语句的使用不加任何限制，所设计出的操作系统既庞大又杂乱，缺乏清晰的程序结构。这一方面会使所编制出的程序错误很多，给调试工作带来很多困难；另一方面也使程序难以阅读和理解，增加了维护人员的负担。

2. 模块化OS结构

1) 模块化结构

模块化程序设计技术，是最早(20世纪60年代)出现的一种程序设计技术。该技术是基于“分解”和“模块化”原则来控制大型软件的复杂度的。为使OS具有较清晰的结构，OS不再是由众多的过程直接构成，而是将OS按其功能划分为若干个具有一定独立性和大小的模块。每个模块具有某方面的管理功能，如进程管理模块、存储器管理模块、I/O设备管理模块和文件管理模块等，并规定好各模块间的接口， 使各模块之间能通过该接口实现交互，然后再进一步将各模块细分为若干个具有一定管理功能的子模块，如把进程管理模块又分为进程控制、 进程同步、 进程通信和进程调度等子模块， 同样也要规定各子模块之间的接口。若子模块较大时，再进一步将它细分。图 1-5 示出了由模块、子模块等组成的模块化OS结构。

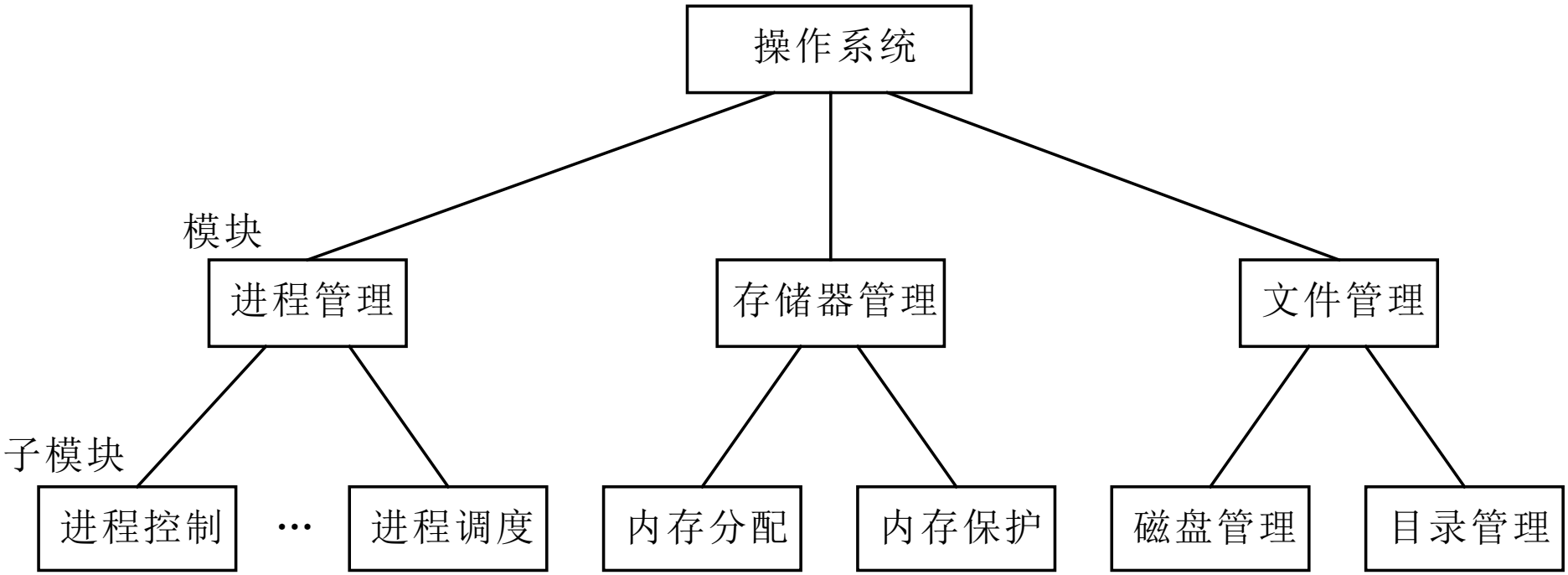


图 1-5 模块化操作系统结构

2) 模块化OS的优缺点

(1) 提高了OS设计的正确性、 可理解性和可维护性。

(2) 增强了OS的可适应性。

(3) 加速了OS的开发过程。

模块化结构设计的缺点有二。首先，在开始设计OS时，对模块的划分及对接口的规定并不精确， 而且还可能存在错误，因而很难保证按此规定所设计出的模块会完全正确， 这将使在把这些模块装配成OS时发生困难；其次，从功能观点来划分模块时，未能将共享资源和独占资源加以区别； 由于管理上的差异，又会使模块间存在着复杂的依赖关系使OS结构变得不清晰。

3. 分层式OS结构

1) 有序分层的基本概念

从改进设计方式上说，应使我们的每一步设计都是建立在可靠的基础上。我们可以从物理机器开始， 在其上面先添加一层具有一定功能的软件A_1，由于A_1是建立在完全确定的物理机器上的，在经过精心设计和几乎是穷尽无遗的测试后，可以认为A_1是正确的；然后再在A_1上添加一层新软件A_2，……，如此一层一层地自底向上增添软件层，每一层都实现若干功能，最后总能构成一个能满足需要的OS。

分层式结构设计的基本原则是：每一层都仅使用其底层所提供的功能和服务，这样可使系统的调试和验证都变得容易，例如，在调试第一层软件A_1时，由于它只使用了物理机器提供的功能，因此它将与其所有的高层软件A_2，……，A_n无关；同样在调试A_2时，它也只使用了A_1和物理机器所提供的功能，而与其高层软件A_3，……，A_n无关，这样，一旦发现A_i出现错误时，通常该错误只会局限于A_i，因为它与所有其高层的软件无关，而A_i层以下的各层软件，又都经过仔细的调试。

2) 层次的设置

(1) 程序嵌套。 通常OS的每个功能的实现， 并非是只用一个程序便能完成的，而是要经由若干个软件层才有可能完成。因此在划分OS层次时，首先要考虑在实现OS的每个功能时所形成的程序嵌套。例如，作业调度模块须调用进程控制模块；在为某作业创建一进程时，进程控制模块又须调用内存管理模块为新进程分配内存空间，可见，进程控制模块应在内存管理模块之上； 而作业调度模块又应在更高层。

(2) 运行频率。在分层结构中，各层次软件的运行速度是不同的，因为A_1层软件能直接在物理机器上运行，故它有最高的运行速度。随着层次的增高，其相应软件的运行速度就随之下降，因而A_n层软件的运行速度最低。为了提高OS的运行效率，应该将那些经常活跃的模块放在最接近硬件的A_1层，如时钟管理、进程调度，通常都放在A_1层。

(3) 公用模块。应把供多种资源管程程序调用的公用模块， 设置在最低层，不然，会使比它低的层次模块由于无法调用它而须另外配置相应功能的模块。例如，用于对信号量进行操作的原语Signal和Wait。

(4) 用户接口。为方便用户(程序)，OS向用户提供了“用户与OS的接口”，如命令接口、程序接口以及图形用户接口。这些接口应设置在OS的最高层，直接提供给用户使用。

1.5.3 微内核OS结构

1. 客户/服务器模式(Client-Server Model)

1) 基本概念

为了提高OS的灵活性和可扩充性而将OS划分为两部分， 一部分是用于提供各种服务的一组服务器(进程)，如用于提供进程管理的进程服务器、提供存储器管理的存储器服务器提供文件管理的文件服务器等，所有这些服务器(进程)都运行在用户态。 当有一用户进程(现在称为客户进程)要求读文件的一个盘块时，该进程便向文件服务器(进程)发出一个请求；当服务器完成了该客户的请求后，便给该客户回送一个响应。操作系统的另一部分是内核，用来处理客户和服务器之间的通信， 即由内核来接收客户的请求，再将该请求送至相应的服务器；同时它也接收服务器的应答， 并将此应答回送给请求客户。此外，在内核中还应具有其它一些机构，用于实现与硬件紧密相关的和一些较基本的功能。

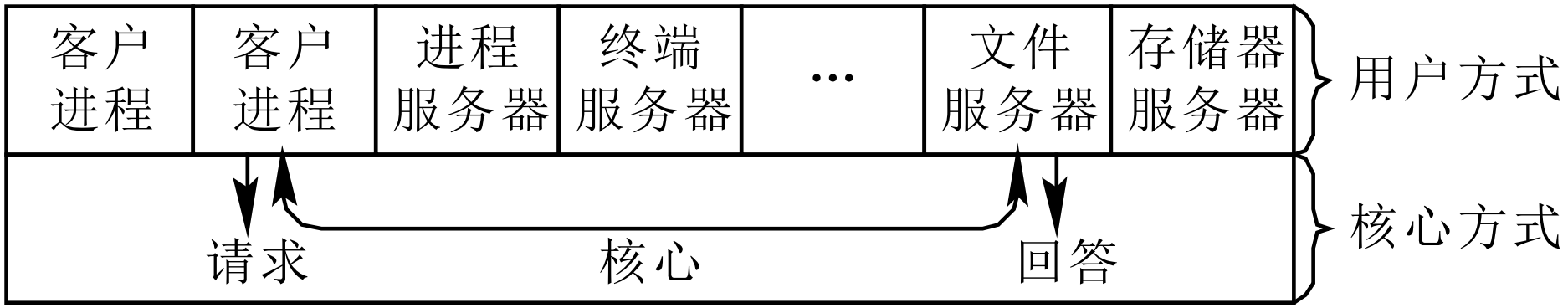


图 1-6 单机环境下的客户/服务器模式

2) 客户/服务器模式的优点

(1) 提高了系统的灵活性和可扩充性。

(2) 提高了OS的可靠性。

(3) 可运行于分布式系统中。

2. 面向对象的程序设计技术(Object-Orientated Programming)

1) 面向对象技术的基本概念

面向对象技术是20世纪80年代初提出并很快流行起来的。该技术是基于“抽象”和“隐蔽”原则来控制大型软件的复杂度的。所谓对象，是指在现实世界中具有相同属性、服从相同规则的一系列事物的抽象，而把其中的具体事物称为对象的实例。OS中的各类实体如进程、线程、消息、存储器等，都使用了对象这一概念，相应地，便有进程对象线程对象、 存储器对象等。

图 1-7 一个对象的示意图

2) 面向对象技术的优点

(1) 可修改性和可扩充性。由于隐蔽了表示实体的数据和操作，因而可以改变对象的表示而不会影响其它部分，从而可以方便地改变老的对象和增加新的对象。

(2) 继承性。继承性是面向对象技术所具有的重要特性。继承性是指子对象可以继承父对象的属性，这样，在创建一个新的对象时， 便可减少大量的时空开销。

(3) 正确性和可靠性。由于对象是构成操作系统的基本单元，可以独立地对它进行测试，这样，比较易于保证其正确性和可靠性，从而比较容易保证整个系统的正确性和可靠性。

3. 微内核技术

1) 微内核技术的引入

所谓微内核技术，是指精心设计的、能实现现代OS核心功能的小型内核，它与一般的OS(程序)不同， 它更小更精炼，它不仅运行在核心态，而且开机后常驻内存，它不会因内存紧张而被换出内存。微内核并非是一个完整的OS， 而只是为构建通用OS提供一个重要基础。由于在微内核OS结构中，通常都采用了客户/服务器模式，因此OS的大部分功能和服务，都是由若干服务器来提供的，如文件服务器、作业服务器和网络服务器等。

2) 微内核的基本功能

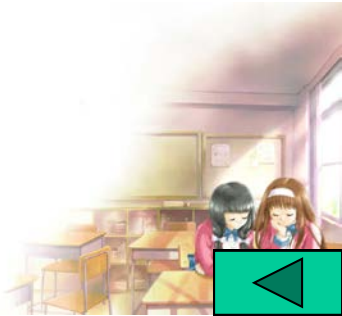
微内核所提供的功能，通常都是一些最基本的功能，如进程管理、存储器管理、进程间通信、 低级I/O功能。

(1) 进程管理。

(2) 存储器管理。

(3) 进程通信管理。

(4) I/O设备管理。



第二章 进程管理

2.1 进程的基本概念

2.1.1 程序的顺序执行及其特征

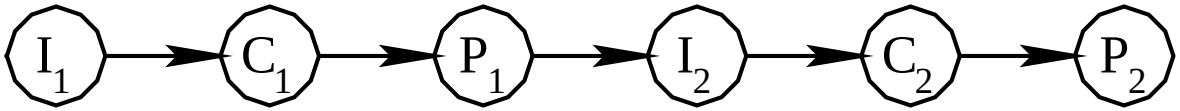
1. 程序的顺序执行

仅当前一操作(程序段)执行完后，才能执行后继操作。例如，在进行计算时，总须先输入用户的程序和数据，然后进行计算，最后才能打印计算结果。

S_1: a ：=x+y;

S_2: b ：=a-5;

S_3: c ：=b+1;



(*a*) 程序的顺序执行

(*b*) 三条语句的顺序执行

图 2-1 程序的顺序执行

2. 程序顺序执行时的特征

(1) 顺序性：

(2) 封闭性：

(3) 可再现性：

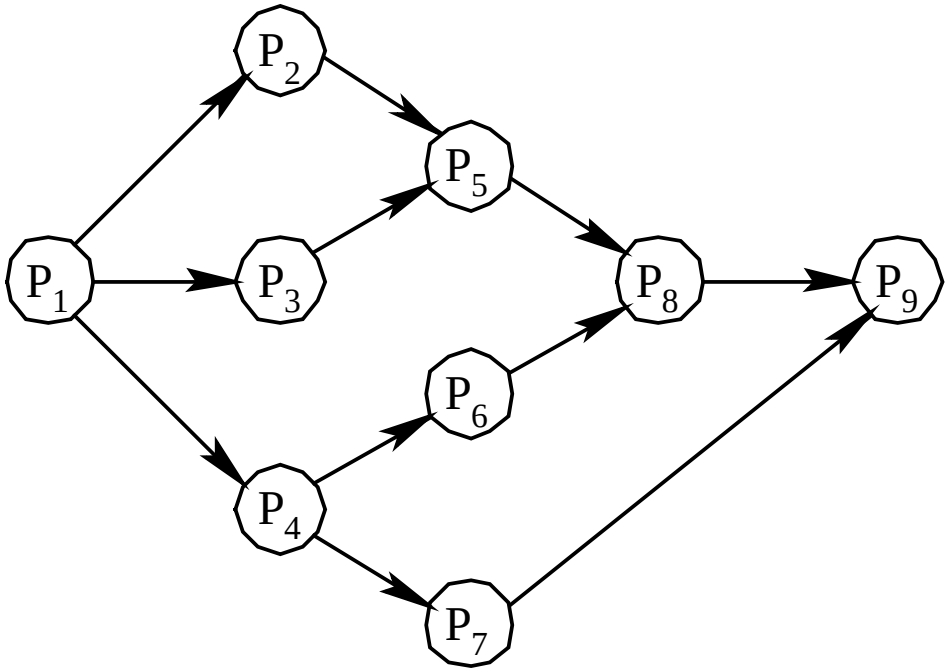
2.1.2 前趋图

前趋图(Precedence Graph)是一个有向无循环图，记为DAG(Directed Acyclic Graph)，用于描述进程之间执行的前后关系。图中的每个结点可用于描述一个程序段或进程，乃至一条语句；结点间的有向边则用于表示两个结点之间存在的偏序(Partial Order)或前趋关系(Precedence Relation)“→”。

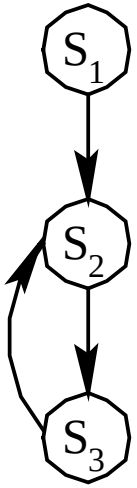
$\rightarrow=\{(P_i, P_j)|P_i$ must complete before P_j may start$\}$，如果$(P_i, P_j)\in\rightarrow$,可写成$P_i\rightarrow P_j$，称P_i是P_j的直接前趋，而称P_j是P_i的直接后继。在前趋图中，把没有前趋的结点称为初始结点(Initial Node)，把没有后继的结点称为终止结点(Final Node)。

每个结点还具有一个重量(Weight)，用于表示该结点所含有的程序量或结点的执行时间。

$I_i \rightarrow C_i \rightarrow P_i$和$S_1 \rightarrow S_2 \rightarrow S_3$



(*a*) 具有九个结点的前趋图



(*b*) 具有循环的前趋图

图 2-2 前趋图

对于图 2-2(*a*)所示的前趋图， 存在下述前趋关系：

$P_1 \rightarrow P_2$, $P_1 \rightarrow P_3$, $P_1 \rightarrow P_4$, $P_2 \rightarrow P_5$, $P_3 \rightarrow P_5$, $P_4 \rightarrow P_6$, $P_4 \rightarrow P_7$, $P_5 \rightarrow P_8$, $P_6 \rightarrow P_8$, $P_7 \rightarrow P_9$, $P_8 \rightarrow P_9$

或表示为：

$$P=\{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9\}$$

$$\rightarrow=\{ (P_1, P_2), (P_1, P_3), (P_1, P_4), (P_2, P_5), (P_3, P_5), (P_4, P_6), (P_4, P_7), (P_5, P_8), (P_6, P_8), (P_7, P_9), (P_8, P_9)\}$$

应当注意，前趋图中必须不存在循环，但在图2-2(*b*)中却有着下述的前趋关系：

$$S_2 \rightarrow S_3, S_3 \rightarrow S_2$$

2.1.3 程序的并发执行及其特征

1. 程序的并发执行

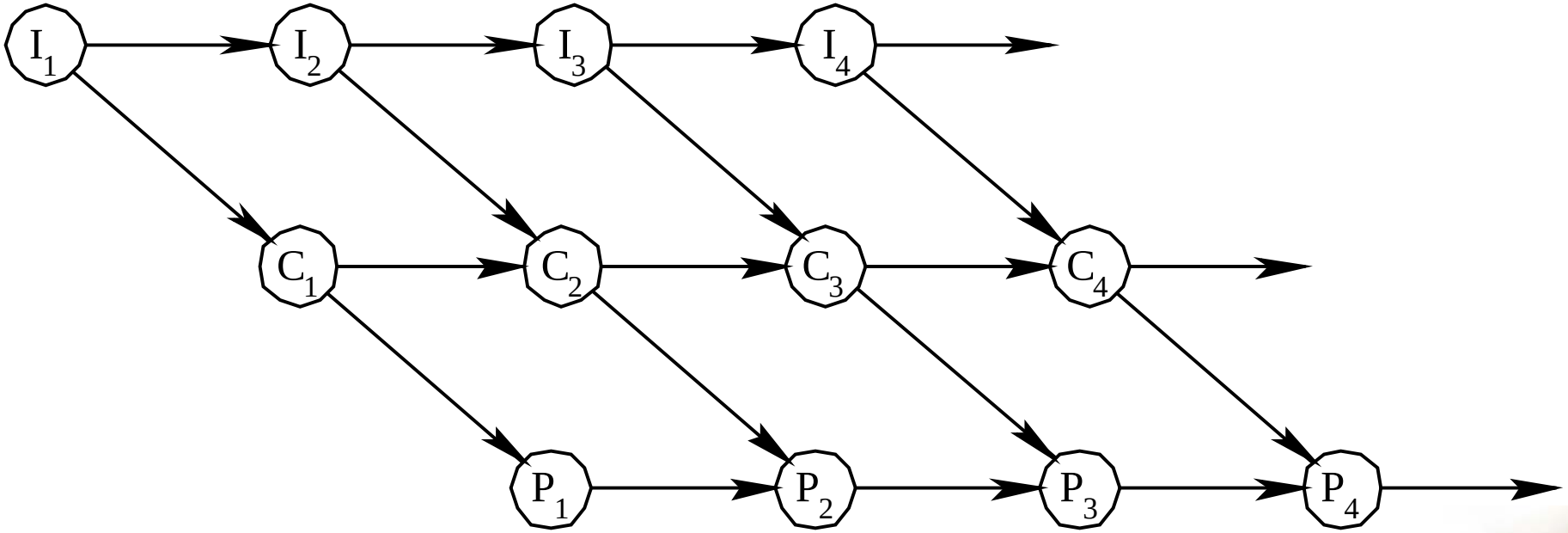


图 2-3 并发执行时的前趋图

在该例中存在下述前趋关系：

$I_i \rightarrow C_i$，$I_i \rightarrow I_{i+1}$, $C_i \rightarrow P_i$, $C_i \rightarrow C_{i+1}$，$P_i \rightarrow P_{i+1}$

而I_{i+1}和C_i及P_{i-1}是重迭的，亦即在P_{i-1}和C_i以及I_{i+1}之间，可以并发执行。 对于具有下述四条语句的程序段：

S_1: a：=x+2

S_2: b：=y+4

S_3: c：=a+b

S_4: d：=c+b

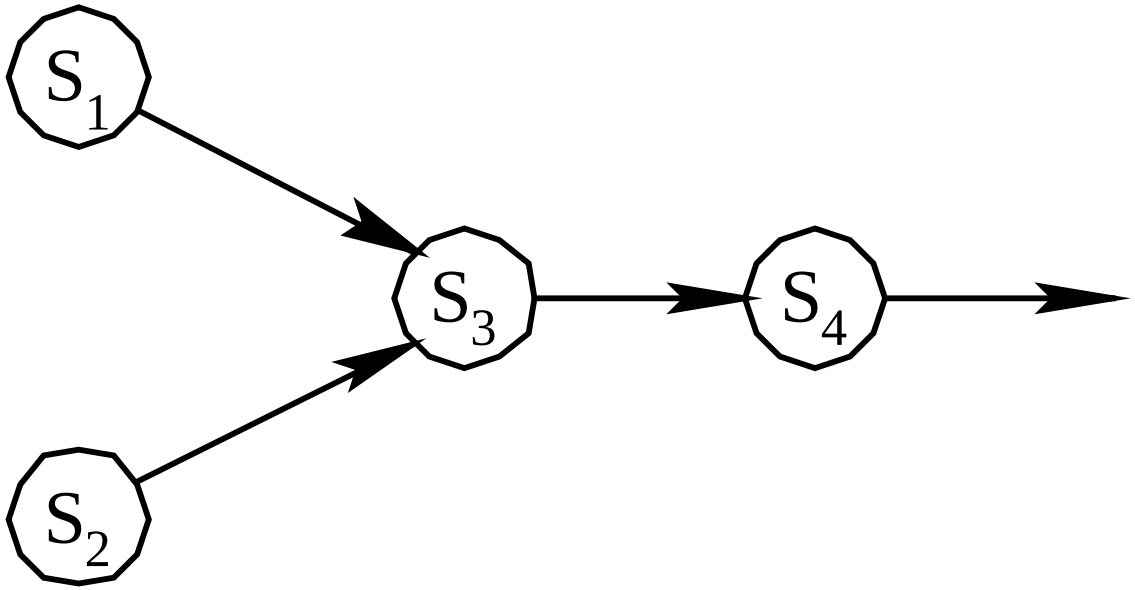


图 2-4 四条语句的前趋关系

2. 程序并发执行时的特征

1) 间断性

2) 失去封闭性

3) 不可再现性

例如，有两个循环程序A和B，它们共享一个变量N。程序A每执行一次时，都要做N：=N+1操作；程序B每执行一次时，都要执行Print(N)操作，然后再将N置成“0”。程序A和B以不同的速度运行。

(1) N：=N+1在Print(N)和N：=0之前，此时得到的N值分别为n+1, n+1, 0。

(2) N：=N+1在Print(N)和N：=0之后，此时得到的N值分别为n, 0, 1。

(3) N：=N+1在Print(N)和N：=0之间，此时得到的N值分别为n, n+1, 0。

2.1.4 进程的特征与状态

1. 进程的特征和定义

1) 结构特征

2) 动态性

3) 并发性

4) 独立性

5) 异步性

较典型的进程定义有：

(1) 进程是程序的一次执行。

(2) 进程是一个程序及其数据在处理机上顺序执行时所发生的活动。

(3) 进程是程序在一个数据集合上运行的过程，它是系统进行资源分配和调度的一个独立单位。

在引入了进程实体的概念后，我们可以把传统OS中的进程定义为：“进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位”。

2. 进程的三种基本状态

1) 就绪(Ready)状态

2) 执行状态

3) 阻塞状态

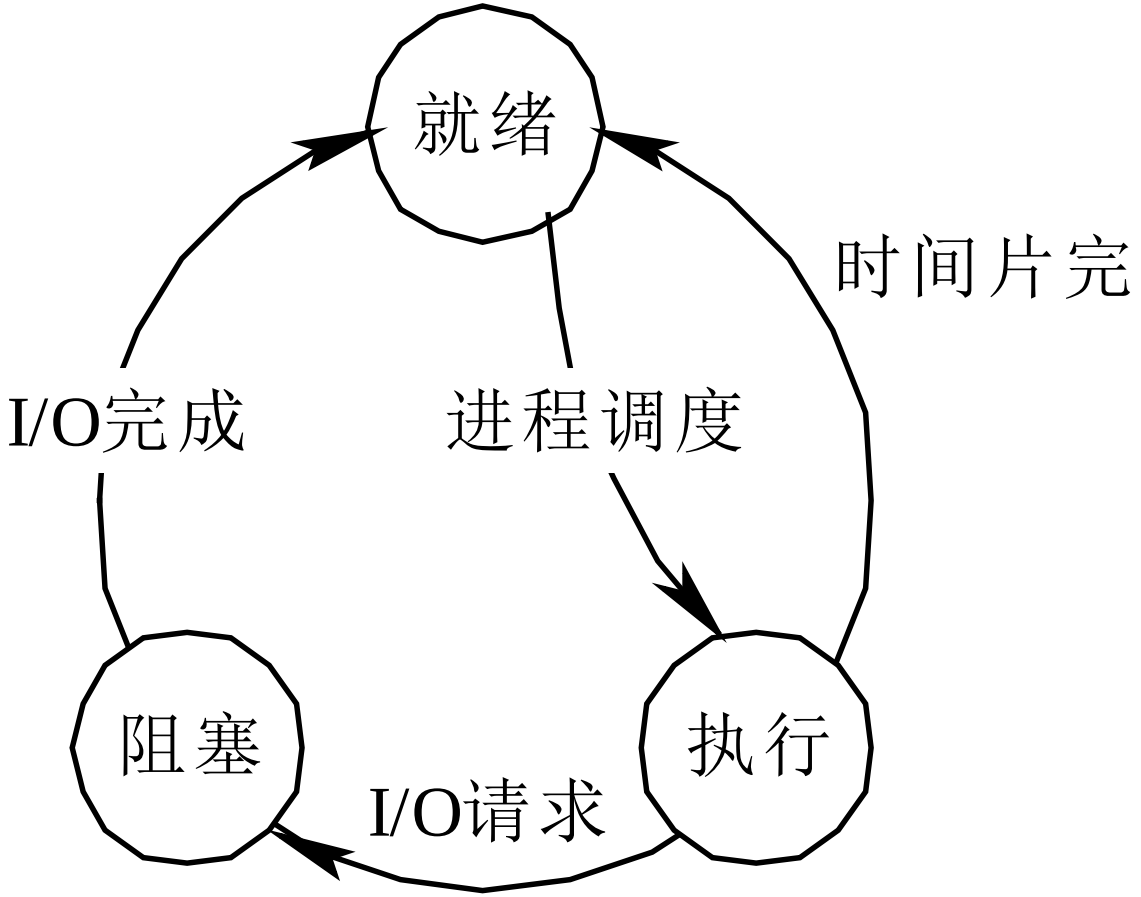


图 2-5 进程的三种基本状态及其转换

3. 挂起状态

1) 引入挂起状态的原因

(1) 终端用户的请求。

(2) 父进程请求。

(3) 负荷调节的需要。

(4) 操作系统的需要。

2) 进程状态的转换

(1) 活动就绪→静止就绪。

(2) 活动阻塞→静止阻塞。

(3) 静止就绪→活动就绪。

(4) 静止阻塞→活动阻塞。

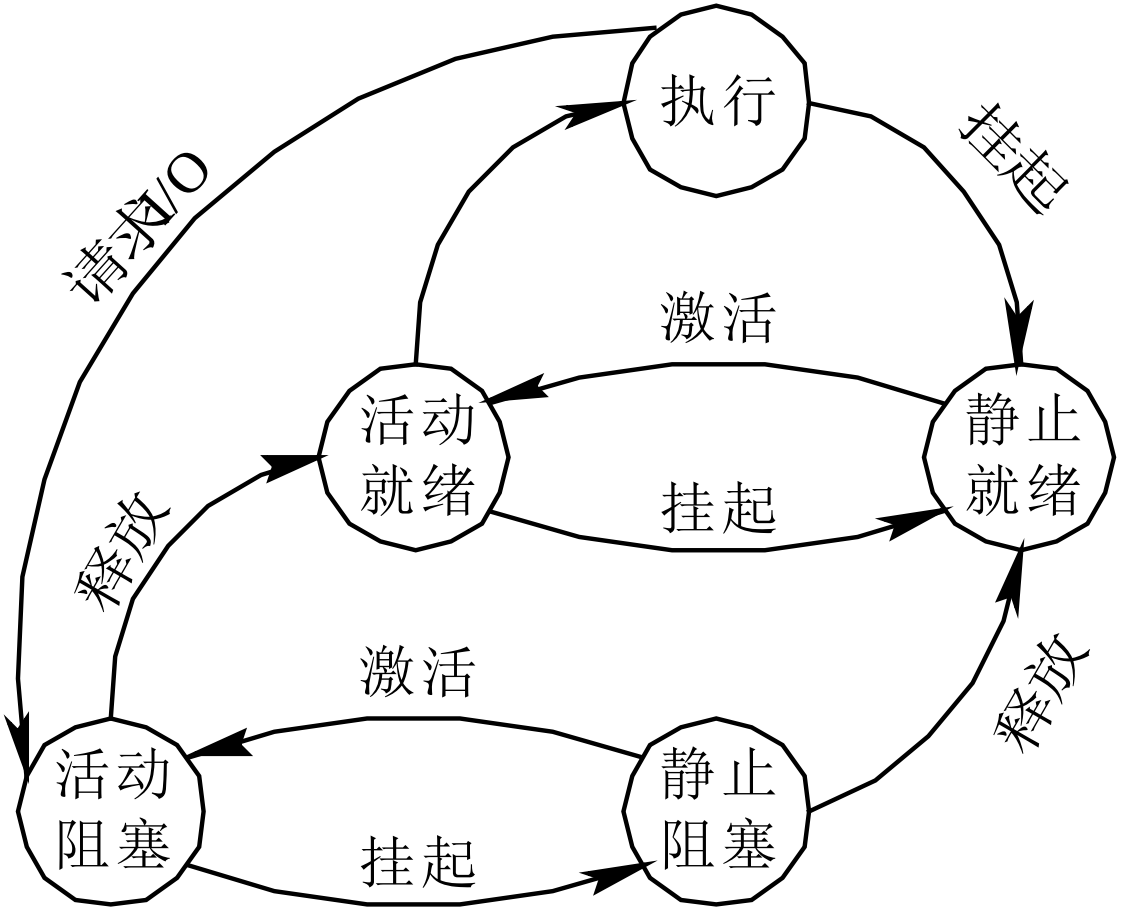


图 2-6 具有挂起状态的进程状态图

2.1.5 进程控制块

1. 进程控制块的作用

进程控制块的作用是使一个在多道程序环境下不能独立运行的程序(含数据)，成为一个能独立运行的基本单位，一个能与其它进程并发执行的进程。或者说，OS是根据PCB来对并发执行的进程进行控制和管理的。

2. 进程控制块中的信息

1) 进程标识符

进程标识符用于惟一地标识一个进程。一个进程通常有两种标识符：

(1) 内部标识符。在所有的操作系统中，都为每一个进程赋予一个惟一的数字标识符，它通常是一个进程的序号。设置内部标识符主要是为了方便系统使用。

(2) 外部标识符。它由创建者提供，通常是由字母、数字组成，往往是由用户(进程)在访问该进程时使用。为了描述进程的家族关系， 还应设置父进程标识及子进程标识。此外，还可设置用户标识，以指示拥有该进程的用户。

2) 处理机状态

处理机状态信息主要是由处理机的各种寄存器中的内容组成的。 ① 通用寄存器，又称为用户可视寄存器，它们是用户程序可以访问的，用于暂存信息， 在大多数处理机中，有 8~32 个通用寄存器，在RISC结构的计算机中可超过 100 个；② 指令计数器，其中存放了要访问的下一条指令的地址；③ 程序状态字PSW，其中含有状态信息，如条件码、执行方式、 中断屏蔽标志等； ④ 用户栈指针， 指每个用户进程都有一个或若干个与之相关的系统栈，用于存放过程和系统调用参数及调用地址。栈指针指向该栈的栈顶。

3) 进程调度信息

在PCB中还存放一些与进程调度和进程对换有关的信息，包括： ① 进程状态，指明进程的当前状态， 作为进程调度和对换时的依据；② 进程优先级，用于描述进程使用处理机的优先级别的一个整数， 优先级高的进程应优先获得处理机； ③ 进程调度所需的其它信息，它们与所采用的进程调度算法有关，比如，进程已等待CPU的时间总和、进程已执行的时间总和等；④ 事件，是指进程由执行状态转变为阻塞状态所等待发生的事件，即阻塞原因。

4) 进程控制信息

进程控制信息包括：① 程序和数据的地址， 是指进程的程序和数据所在的内存或外存地(首)址，以便再调度到该进程执行时，能从PCB中找到其程序和数据；② 进程同步和通信机制，指实现进程同步和进程通信时必需的机制，如消息队列指针、信号量等，它们可能全部或部分地放在PCB中； ③ 资源清单，是一张列出了除CPU以外的、进程所需的全部资源及已经分配到该进程的资源的清单；④ 链接指针， 它给出了本进程(PCB)所在队列中的下一个进程的PCB的首地址。

3. 进程控制块的组织方式

1) 链接方式

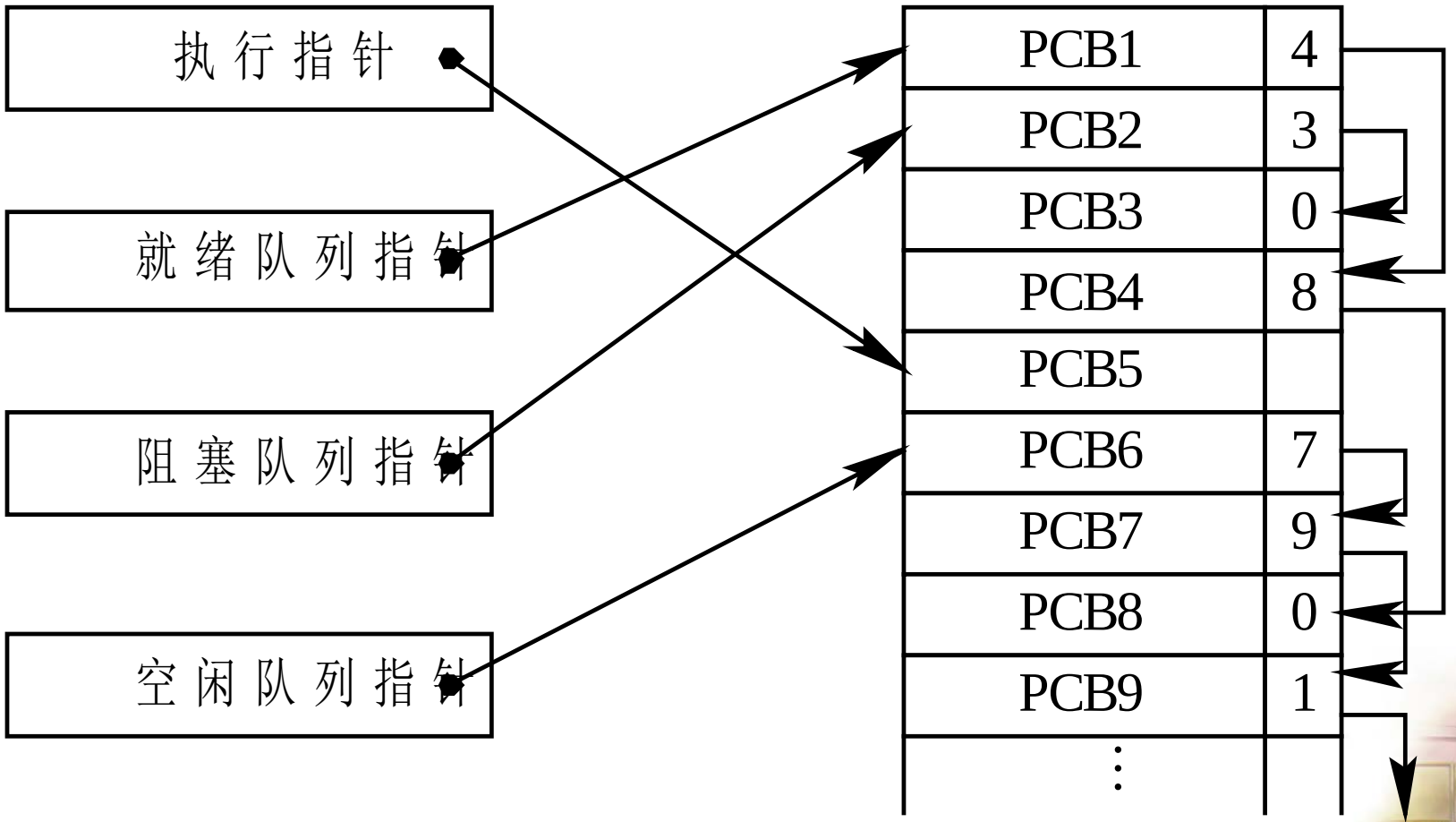


图 2-7 PCB链接队列示意图

2) 索引方式

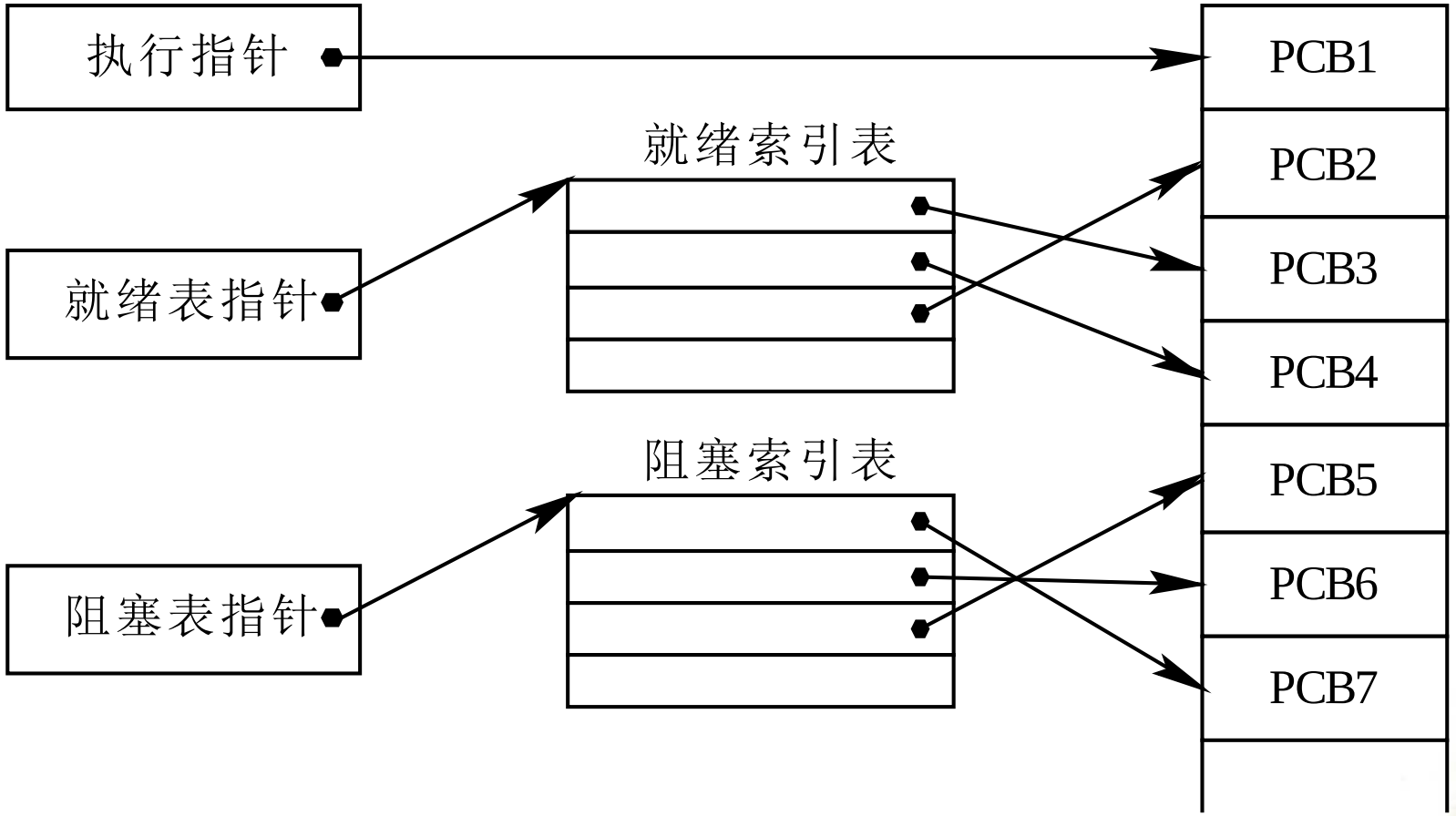


图 2-8 按索引方式组织PCB

2.2 进 程 控 制

2.2.1 进程的创建

1. 进程图(Process Graph)

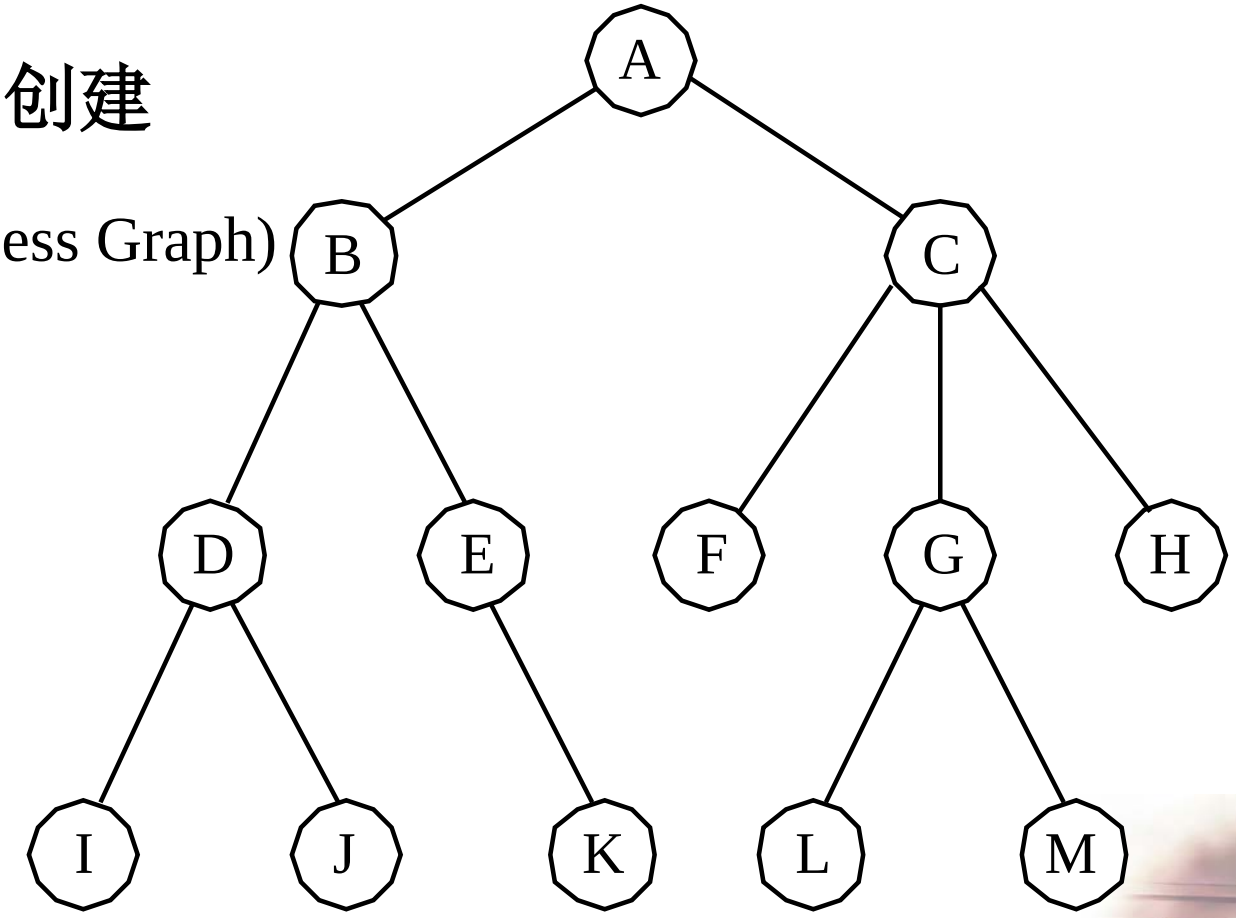


图 2-9 进程树

2. 引起创建进程的事件

(1) 用户登录。

(2) 作业调度。

(3) 提供服务。

(4) 应用请求。

3. 进程的创建(Creation of Progress)

(1)申请空白PCB。

(2) 为新进程分配资源。

(3) 初始化进程控制块。

(4) 将新进程插入就绪队列，如果进程就绪队列能够接纳新进程， 便将新进程插入就绪队列。

2.2.2 进程的终止

1. 引起进程终止(Termination of Process)的事件

1) 正常结束

在任何计算机系统中，都应有一个用于表示进程已经运行完成的指示。例如，在批处理系统中，通常在程序的最后安排一条Holt指令或终止的系统调用。当程序运行到Holt指令时，将产生一个中断，去通知OS本进程已经完成。在分时系统中，用户可利用Logs off去表示进程运行完毕，此时同样可产生一个中断，去通知OS进程已运行完毕。

2) 异常结束

在进程运行期间，由于出现某些错误和故障而迫使进程终止。这类异常事件很多，常见的有：① 越界错误。这是指程序所访问的存储区，已越出该进程的区域；② 保护错。进程试图去访问一个不允许访问的资源或文件，或者以不适当的方式进行访问，例如，进程试图去写一个只读文件；③ 非法指令。程序试图去执行一条不存在的指令。出现该错误的原因，可能是程序错误地转移到数据区，把数据当成了指令；④ 特权指令错。用户进程试图去执行一条只允许OS执行的指令；⑤ 运行超时。进程的执行时间超过了指定的最大值；⑥ 等待超时。进程等待某事件的时间，超过了规定的最大值；⑦ 算术运算错。进程试图去执行一个被禁止的运算，例如，被0除；⑧ I/O故障。这是指在I/O过程中发生了错误等。

3) 外界干预

外界干预并非指在本进程运行中出现了异常事件，而是指进程应外界的请求而终止运行。这些干预有：① 操作员或操作系统干预。 由于某种原因，例如，发生了死锁， 由操作员或操作系统终止该进程； ② 父进程请求。由于父进程具有终止自己的任何子孙进程的权利， 因而当父进程提出请求时，系统将终止该进程； ③ 父进程终止。 当父进程终止时，OS也将他的所有子孙进程终止。

2. 进程的终止过程

(1) 根据被终止进程的标识符，从PCB集合中检索出该进程的PCB，从中读出该进程的状态。

(2) 若被终止进程正处于执行状态，应立即终止该进程的执行，并置调度标志为真，用于指示该进程被终止后应重新进行调度。

(3) 若该进程还有子孙进程，还应将其所有子孙进程予以终止，以防他们成为不可控的进程。

(4) 将被终止进程所拥有的全部资源，或者归还给其父进程， 或者归还给系统。

(5) 将被终止进程(它的PCB)从所在队列(或链表)中移出，等待其他程序来搜集信息。

2.2.3 进程的阻塞与唤醒

1. 引起进程阻塞和唤醒的事件

1) 请求系统服务

2) 启动某种操作

3) 新数据尚未到达

4) 无新工作可做

2. 进程阻塞过程

正在执行的进程，当发现上述某事件时，由于无法继续执行，于是进程便通过调用阻塞原语block把自己阻塞。可见，进程的阻塞是进程自身的一种主动行为。进入block过程后，由于此时该进程还处于执行状态，所以应先立即停止执行，把进程控制块中的现行状态由“执行”改为阻塞，并将PCB插入阻塞队列。如果系统中设置了因不同事件而阻塞的多个阻塞队列，则应将本进程插入到具有相同事件的阻塞(等待)队列。最后，转调度程序进行重新调度，将处理机分配给另一就绪进程，并进行切换，亦即，保留被阻塞进程的处理机状态(在PCB中)，再按新进程的PCB中的处理机状态设置CPU的环境。

3. 进程唤醒过程

当被阻塞进程所期待的事件出现时，如I/O完成或其所期待的数据已经到达，则由有关进程(比如，用完并释放了该I/O设备的进程)调用唤醒原语wakeup( )，将等待该事件的进程唤醒。唤醒原语执行的过程是：首先把被阻塞的进程从等待该事件的阻塞队列中移出，将其PCB中的现行状态由阻塞改为就绪，然后再将该PCB插入到就绪队列中。

2.2.4 进程的挂起与激活

1. 进程的挂起

当出现了引起进程挂起的事件时，比如，用户进程请求将自己挂起，或父进程请求将自己的某个子进程挂起，系统将利用挂起原语suspend( )将指定进程或处于阻塞状态的进程挂起。挂起原语的执行过程是：首先检查被挂起进程的状态，若处于活动就绪状态，便将其改为静止就绪；对于活动阻塞状态的进程，则将之改为静止阻塞。为了方便用户或父进程考查该进程的运行情况而把该进程的PCB复制到某指定的内存区域。最后，若被挂起的进程正在执行，则转向调度程序重新调度。

2. 进程的激活过程

当发生激活进程的事件时，例如，父进程或用户进程请求激活指定进程，若该进程驻留在外存而内存中已有足够的空间时，则可将在外存上处于静止就绪状态的进程换入内存。这时，系统将利用激活原语active( )将指定进程激活。 激活原语先将进程从外存调入内存，检查该进程的现行状态，若是静止就绪，便将之改为活动就绪；若为静止阻塞便将之改为活动阻塞。假如采用的是抢占调度策略，则每当有新进程进入就绪队列时，应检查是否要进行重新调度，即由调度程序将被激活进程与当前进程进行优先级的比较，如果被激活进程的优先级更低，就不必重新调度；否则，立即剥夺当前进程的运行，把处理机分配给刚被激活的进程。

2.3 进 程 同 步

2.3.1 进程同步的基本概念

1. 两种形式的制约关系

(1) 间接相互制约关系。

(2) 直接相互制约关系。

2. 临界资源(Critical Resouce)

生产者-消费者(producer-consumer)问题是一个著名的进程同步问题。它描述的是：有一群生产者进程在生产产品，并将这些产品提供给消费者进程去消费。为使生产者进程与消费者进程能并发执行，在两者之间设置了一个具有n个缓冲区的缓冲池，生产者进程将它所生产的产品放入一个缓冲区中； 消费者进程可从一个缓冲区中取走产品去消费。尽管所有的生产者进程和消费者进程都是以异步方式运行的，但它们之间必须保持同步，即不允许消费者进程到一个空缓冲区去取产品；也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品。

我们可利用一个数组来表示上述的具有n个(0，1，…，n-1)缓冲区的缓冲池。用输入指针in来指示下一个可投放产品的缓冲区，每当生产者进程生产并投放一个产品后，输入指针加1；用一个输出指针out来指示下一个可从中获取产品的缓冲区，每当消费者进程取走一个产品后，输出指针加1。 由于这里的缓冲池是组织成循环缓冲的，故应把输入指针加1表示成 in：=(in+1)mod n；输出指针加1表示成out：=(out+1) mod n。当(in+1) mod n=out时表示缓冲池满；而in=out则表示缓冲池空。此外，还引入了一个整型变量counter, 其初始值为0。每当生产者进程向缓冲池中投放一个产品后，使counter加1；反之，每当消费者进程从中取走一个产品时，使counter减1。生产者和消费者两进程共享下面的变量：

```
Var n, integer;
type item=...;
var buffer:array［0, 1, ..., n-1］ of item;
in, out: 0, 1, ..., n-1;
counter: 0, 1, ..., n;
```

指针in和out初始化为1。在生产者和消费者进程的描述中，no-op是一条空操作指令，while condition do no-op语句表示重复的测试条件(condication)，重复测试应进行到该条件变为false(假)，即到该条件不成立时为止。在生产者进程中使用一局部变量nextp，用于暂时存放每次刚生产出来的产品；而在消费者进程中，则使用一个局部变量nextc,用于存放每次要消费的产品。

```
producer: repeat
        ...
    produce an item in nextp;
        ...
    while counter=n do no-op;
    buffer［in］ ： = nextp;
    in ： = in+1 mod n;
    counter ： = counter+1;
  until false;
consumer: repeat
    while counter=0 do no-op;
    nextc ： = buffer［out］;
    out ： = (out+1) mod n;
    counter ： = counter-1;
    consumer the item in nextc;
  until false;
```

虽然上面的生产者程序和消费者程序，在分别看时都是正确的，而且两者在顺序执行时其结果也会是正确的，但若并发执行时，就会出现差错，问题就在于这两个进程共享变量counter。生产者对它做加1操作，消费者对它做减1操作，这两个操作在用机器语言实现时， 常可用下面的形式描述：

register 1 ： = counter; register 2： = counter;

register1 ： = register 1+1; register 2： = register 2-1;

counter ： = register 1; counter ： = register 2;

假设：counter的当前值是5。如果生产者进程先执行左列的三条机器语言语句，然后消费者进程再执行右列的三条语句，则最后共享变量counter的值仍为5；反之，如果让消费者进程先执行右列的三条语句，然后再让生产者进程执行左列的三条语句，counter值也还是5，但是，如果按下述顺序执行：

register 1 ： = counter; (register 1=5)

register 1 ： = register 1+1; (register 1=6)

register 2 ： = counter; (register 2=5)

register 2 ： = register 2-1; (register 2=4)

counter ： = register 1; (counter=6)

counter ： = register 2; (counter=4)

3. 临界区(critical section)

可把一个访问临界资源的循环进程描述如下：

```
repeat
        entry section
    critical section;
        exit section
   remainder section;
until false;
```

4. 同步机制应遵循的规则

(1) 空闲让进。

(2) 忙则等待。

(3) 有限等待。

(4) 让权等待。

2.3.2 信号量机制

1. 整型信号量

最初由Dijkstra把整型信号量定义为一个整型量，除初始化外，仅能通过两个标准的原子操作(Atomic Operation) wait(S)和signal(S)来访问。这两个操作一直被分别称为P、V操作。 wait和signal操作可描述为：

wait(S): while S≤0 do no-op

S ：=S-1;

signal(S): S ：=S+1;

2. 记录型信号量

在整型信号量机制中的wait操作，只要是信号量S≤0， 就会不断地测试。因此，该机制并未遵循“让权等待”的准则， 而是使进程处于“忙等”的状态。记录型信号量机制，则是一种不存在“忙等”现象的进程同步机制。但在采取了“让权等待”的策略后，又会出现多个进程等待访问同一临界资源的情况。为此，在信号量机制中，除了需要一个用于代表资源数目的整型变量value外，还应增加一个进程链表L，用于链接上述的所有等待进程。记录型信号量是由于它采用了记录型的数据结构而得名的。它所包含的上述两个数据项可描述为：

```
type semaphore=record
   value:integer;
   L:list of process;
   end
```

相应地，wait(S)和signal(S)操作可描述为：

```
procedure wait(S)
   var S: semaphore;
   begin
   S.value ： = S.value-1;
   if S.value＜0 then block(S,L)
   end
  procedure signal(S)
   var S: semaphore;
   begin
  S.value ： = S.value+1;
  if S.value≤0 then wakeup(S,L);
   end
```

在记录型信号量机制中，S.value的初值表示系统中某类资源的数目， 因而又称为资源信号量，对它的每次wait操作，意味着进程请求一个单位的该类资源，因此描述为S.value：=S.value-1； 当S.value＜0时，表示该类资源已分配完毕，因此进程应调用block原语，进行自我阻塞，放弃处理机，并插入到信号量链表S.L中。可见，该机制遵循了“让权等待”准则。 此时S.value的绝对值表示在该信号量链表中已阻塞进程的数目。 对信号量的每次signal操作，表示执行进程释放一个单位资源，故S.value：=S.value+1操作表示资源数目加1。 若加1后仍是S.value≤0，则表示在该信号量链表中，仍有等待该资源的进程被阻塞，故还应调用wakeup原语，将S.L链表中的第一个等待进程唤醒。如果S.value的初值为1，表示只允许一个进程访问临界资源，此时的信号量转化为互斥信号量。

3. AND型信号量

在两个进程中都要包含两个对Dmutex和Emutex的操作， 即

process A:　　process B:

wait(Dmutex);　　wait(Emutex);

wait(Emutex);　　wait(Dmutex);

若进程A和B按下述次序交替执行wait操作：

process A: wait(Dmutex); 于是Dmutex=0

process B: wait(Emutex); 于是Emutex=0

process A: wait(Emutex); 于是Emutex=-1 A阻塞

process B: wait(Dmutex); 于是Dmutex=-1 B阻塞

AND同步机制的基本思想是：将进程在整个运行过程中需要的所有资源，一次性全部地分配给进程，待进程使用完后再一起释放。只要尚有一个资源未能分配给进程，其它所有可能为之分配的资源，也不分配给他。亦即，对若干个临界资源的分配，采取原子操作方式：要么全部分配到进程，要么一个也不分配。 由死锁理论可知，这样就可避免上述死锁情况的发生。为此，在wait操作中，增加了一个“AND”条件，故称为AND同步，或称为同时wait操作， 即Swait(Simultaneous wait)定义如下：

Swait(S_1, S_2, ..., S_n)

if $S_i \geq 1$ and ... and $S_n \geq 1$ then

for i ： = 1 to n do

S_i ：= S_i-1;

endfor

else

place the process in the waiting queue associated with the first S_i found with $S_i < 1$, and set the program count of this process to the beginning of Swait operation

endif

Ssignal(S 1, S 2, ..., S n)

for i ： = 1 to n do

$S_i = S_i + 1$;

Remove all the process waiting in the queue associated with S_i into the ready queue.

endfor;

4. 信号量集

Swait(S_1, t_1, d_1, ..., S_n, t_n, d_n)
if $S_i \geq t_1$ and ... and $S_n \geq t_n$ then
for i：=1 to n do
S_i：=S_i-d_i;
endfor
else
Place the executing process in the waiting queue of the first S_i with $S_i < t_i$ and set its program counter to the beginning of the Swait Operation.
endif

signal(S_1, d_1, ..., S_n, d_n)
for i ：=1 to n do
S_i ：= S_i+d_i;
Remove all the process waiting in the queue associated with S_i into the ready queue
endfor;

一般“信号量集”的几种特殊情况：

(1) Swait(S, d, d)。 此时在信号量集中只有一个信号量S，但允许它每次申请d个资源，当现有资源数少于d时，不予分配。

(2) Swait(S, 1, 1)。 此时的信号量集已蜕化为一般的记录型信号量(S＞1时)或互斥信号量(S=1时)。

(3) Swait(S, 1, 0)。这是一种很特殊且很有用的信号量操作。当S≥1时，允许多个进程进入某特定区；当S变为0后，将阻止任何进程进入特定区。换言之，它相当于一个可控开关。

2.3.3 信号量的应用

1. 利用信号量实现进程互斥

利用信号量实现进程互斥的进程可描述如下：

```
Var mutex:semaphore ： = 1;
 begin
 parbegin
  process 1: begin
                repeat
                wait(mutex);
                critical section
                signal(mutex);
                remainder seetion
                until false;
```

```
            end
process 2: begin
             repeat
               wait(mutex);
               critical section
               signal(mutex);
              remainder section
            until false;
            end
parend
```

2. 利用信号量实现前趋关系

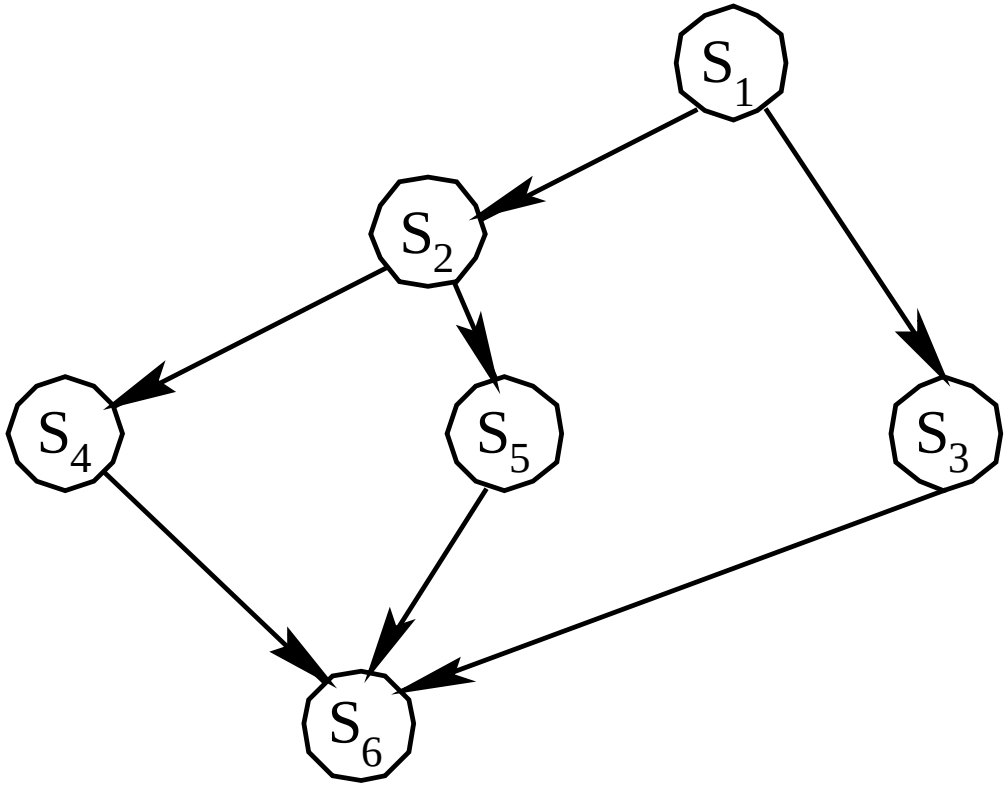
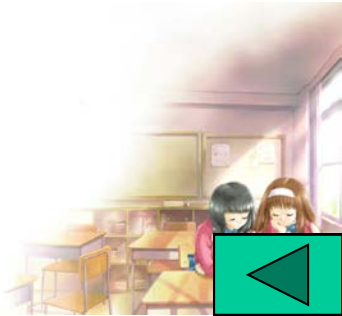


图 2-10 前趋图举例

```
Var a,b,c,d,e,f,g; semaphore ：= 0,0,0,0,0,0,0;
 begin
   parbegin
         begin S1; signal(a); signal(b); end;
         begin wait(a); S2; signal(c); signal(d); end;
         begin wait(b); S3; signal(e); end;
         begin wait(c); S4; signal(f); end;
         begin wait(d); S5; signal(g); end;
         begin wait(e); wait(f); wait(g); S6; end;
   parend
end
```



2.4 经典进程的同步问题

2.4.1 生产者–消费者问题

前面我们已经对生产者–消费者问题(The proceducer-consumer problem)做了一些描述，但未考虑进程的互斥与同步问题，因而造成了数据Counter的不定性。由于生产者–消费者问题是相互合作的进程关系的一种抽象，例如，在输入时，输入进程是生产者，计算进程是消费者；而在输出时，则计算进程是生产者，而打印进程是消费者，因此，该问题有很大的代表性及实用价值。

1. 利用记录型信号量解决生产者–消费者问题

假定在生产者和消费者之间的公用缓冲池中，具有n个缓冲区，这时可利用互斥信号量mutex实现诸进程对缓冲池的互斥使用；利用信号量empty和full分别表示缓冲池中空缓冲区和满缓冲区的数量。又假定这些生产者和消费者相互等效，只要缓冲池未满，生产者便可将消息送入缓冲池；只要缓冲池未空，消费者便可从缓冲池中取走一个消息。对生产者—消费者问题可描述如下：

```
Var mutex, empty, full:semaphore ：= 1,n,0;
 buffer:array［0, ..., n-1］ of item;
 in, out: integer ：= 0, 0;
 begin
  parbegin
 proceducer:begin
   repeat
        ...
   producer an item nextp;
        ...
   wait(empty);
   wait(mutex);
   buffer(in) ：= nextp;
   in ：= (in+1) mod n;
   signal(mutex);
   signal(full);
   until false;
   end
```

```
consumer:begin
    repeat
    wait(full);
    wait(mutex);
    nextc ： = buffer(out);
    out ： = (out+1) mod n;
    signal(mutex);
    signal(empty);
    consumer the item in nextc;
    until false;
    end
parend
end
```

在生产者–消费者问题中应注意：首先，在每个程序中用于实现互斥的wait(mutex)和signal(mutex)必须成对地出现； 其次，对资源信号量empty和full的wait和signal操作，同样需要成对地出现，但它们分别处于不同的程序中。例如，wait(empty)在计算进程中，而signal(empty)则在打印进程中，计算进程若因执行wait(empty)而阻塞， 则以后将由打印进程将它唤醒；最后，在每个程序中的多个wait操作顺序不能颠倒。应先执行对资源信号量的wait操作，然后再执行对互斥信号量的wait操作，否则可能引起进程死锁。

2. 利用AND信号量解决生产者—消费者问题

```
ar mutex, empty, full:semaphore ： = 1, n, 0;
 buffer:array［0, ..., n-1］ of item;
 in out:integer ： = 0, 0;
begin
 parbegin
 producer:begin
  repeat
       ...
  produce an item in nextp;
       ...
  Swait(empty, mutex);
  buffer(in) ： = nextp;
  in ： = (in+1)mod n;
  Ssignal(mutex, full);
   until false;
 end
```

```
consumer:begin
    repeat
     Swait(full, mutex);
     nextc ： = buffer(out);
     out ： = (out+1) mod n;
     Ssignal(mutex, empty);
     consumer the item in nextc;
    until false;
   end
parend
end
```

2.4.2 哲学家进餐问题

1. 利用记录型信号量解决哲学家进餐问题

经分析可知，放在桌子上的筷子是临界资源，在一段时间内只允许一位哲学家使用。为了实现对筷子的互斥使用，可以用一个信号量表示一只筷子，由这五个信号量构成信号量数组。其描述如下：

Var chopstick: array［0, ..., 4］ of semaphore;

所有信号量均被初始化为1， 第i位哲学家的活动可描述为：

```
repeat
        wait(chopstick［i］);
        wait(chopstick［(i+1) mod 5］);
                ...
        eat;
        ...
        signal(chopstick［i］);
        signal(chopstick［(i+1) mod 5］);
                ...
        think;
until false;
```

可采取以下几种解决方法：

(1) 至多只允许有四位哲学家同时去拿左边的筷子，最终能保证至少有一位哲学家能够进餐，并在用毕时能释放出他用过的两只筷子，从而使更多的哲学家能够进餐。

(2) 仅当哲学家的左、右两只筷子均可用时，才允许他拿起筷子进餐。

(3) 规定奇数号哲学家先拿他左边的筷子，然后再去拿右边的筷子；而偶数号哲学家则相反。按此规定，将是1、2号哲学家竞争1号筷子；3、4号哲学家竞争3号筷子。即五位哲学家都先竞争奇数号筷子，获得后，再去竞争偶数号筷子，最后总会有一位哲学家能获得两只筷子而进餐。

2. 利用AND信号量机制解决哲学家进餐问题

在哲学家进餐问题中，要求每个哲学家先获得两个临界资源(筷子)后方能进餐，这在本质上就是前面所介绍的AND同步问题，故用AND信号量机制可获得最简洁的解法。

```
Var chopsiick array ［0, ..., 4］ of semaphore ： = (1,1,1,1,1);
 processi
   repeat
   think;
   Sswait(chopstick［(i+1) mod 5］, chopstick ［i］);
   eat;
   Ssignat(chopstick ［(i+1) mod 5］, chopstick ［i］);
 until false;
```

2.4.3 读者-写者问题

1. 利用记录型信号量解决读者-写者问题

为实现Reader与Writer进程间在读或写时的互斥而设置了一个互斥信号量Wmutex。另外，再设置一个整型变量Readcount表示正在读的进程数目。由于只要有一个Reader进程在读，便不允许Writer进程去写。因此，仅当Readcount=0,表示尚无Reader进程在读时，Reader进程才需要执行Wait(Wmutex)操作。若wait(Wmutex)操作成功，Reader进程便可去读，相应地，做Readcount+1操作。同理，仅当Reader进程在执行了Readcount减1操作后其值为0时，才须执行signal(Wmutex)操作，以便让Writer进程写。又因为Readcount是一个可被多个Reader进程访问的临界资源，因此，应该为它设置一个互斥信号量rmutex。

读者-写者问题可描述如下：

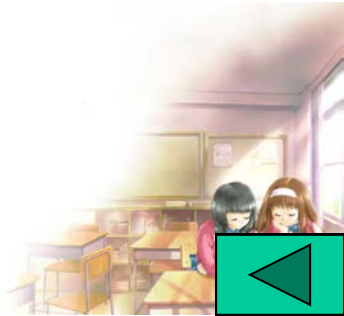
```
Var rmutex, wmutex:semaphore ： = 1,1;
 Readcount:integer ： = 0;
 begin
 parbegin
  Reader:begin
   repeat
  wait(rmutex);
  if readcount=0 then wait(wmutex);
   Readcount ： = Readcount+1;
  signal(rmutex);
        …
  perform read operation;
        …
```

```
wait(rmutex);
 readcount ： = readcount-1;
 if readcount=0 then signal(wmutex);
 signal(rmutex);
  until false;
 end
writer:begin
  repeat
 wait(wmutex);
 perform write operation;
 signal(wmutex);
  until false;
 end
parend
end
```

2. 利用信号量集机制解决读者-写者问题

```
Var RN integer;
 L, mx:semaphore ： = RN,1;
begin
 parbegin
  reader:begin
   repeat
  Swait(L,1,1);
  Swait(mx,1,0);
        ...
  perform read operation;
        ...
```

```
Ssignal(L,1);
 until false;
end
writer:begin
repeat
 Swait(mx,1,1; L,RN,0);
 perform write operation;
 Ssignal(mx,1);
 until false;
end
parend
end
```



2.5 管 程 机 制

2.5.1 管程的基本概念

1. 管程的定义

管程由三部分组成：① 局部于管程的共享变量说明；② 对该数据结构进行操作的一组过程；③ 对局部于管程的数据设置初始值的语句。此外，还须为管程赋予一个名字。

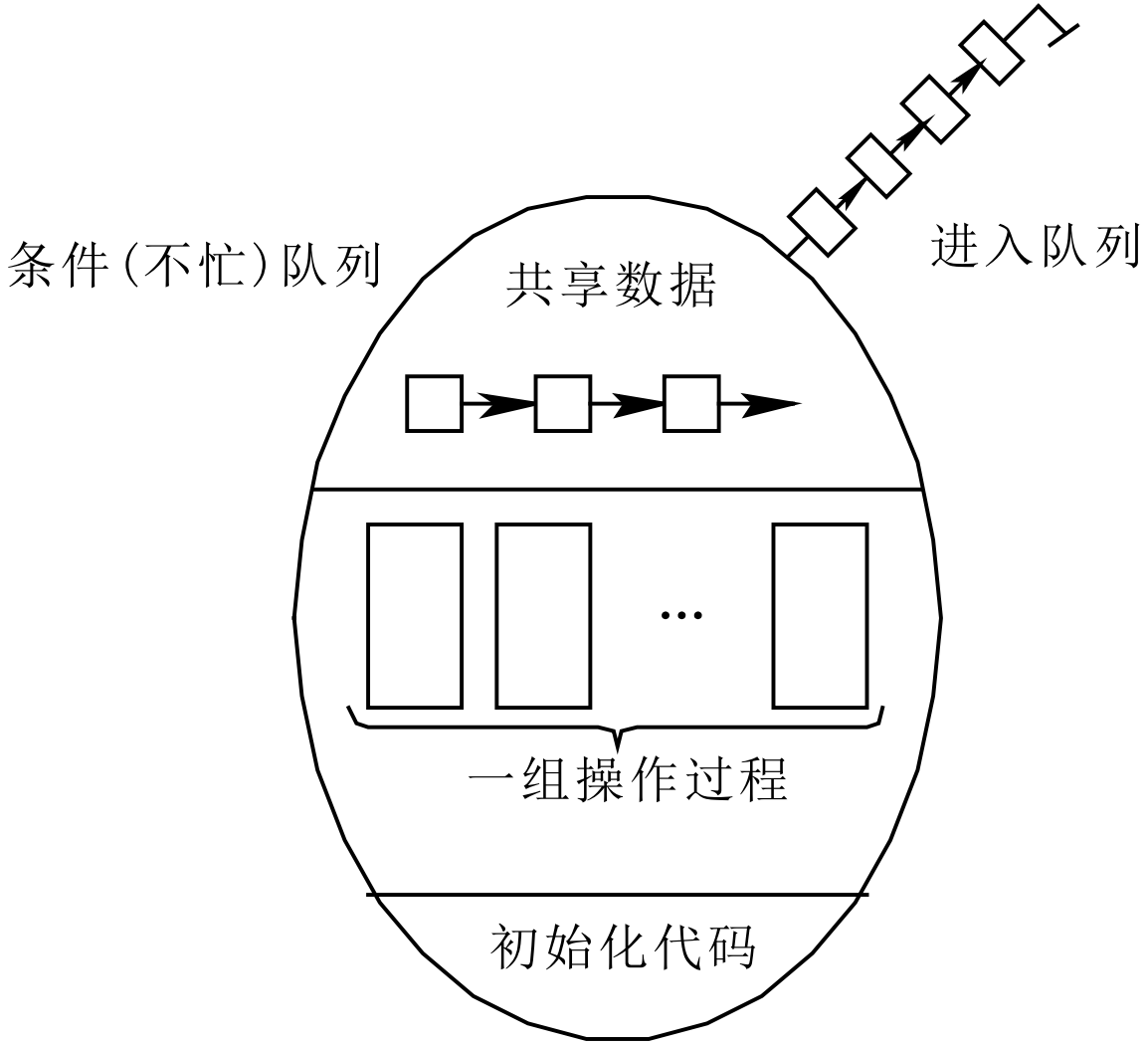


图 2-11 管程的示意图

管程的语法如下：

```
type monitor-name=monitor
variable declarations
procedure entry P1(...);
 begin ... end;
procedure entry P2(...);
  begin ... end;
       ...
procedure entry Pn(...);
  begin ... end;
begin
  initialization code;
end
```

2. 条件变量

管程中对每个条件变量，都须予以说明，其形式为：Var x, y:condition。该变量应置于wait和signal之前，即可表示为X.wait和X.signal。例如，由于共享数据被占用而使调用进程等待，该条件变量的形式为：nonbusy:condition。此时，wait原语应改为nonbusy.wait，相应地，signal应改为nonbusy.signal。

应当指出，X.signal操作的作用，是重新启动一个被阻塞的进程，但如果没有被阻塞的进程，则X.signal操作不产生任何后果。这与信号量机制中的signal操作不同。因为，后者总是要执行s： =s+1操作，因而总会改变信号量的状态。

如果有进程Q处于阻塞状态， 当进程P执行了X.signal操作后，怎样决定由哪个进行执行，哪个等待，可采用下述两种方式之一进行处理：

(1) P等待，直至Q离开管程或等待另一条件。

(2) Q等待，直至P离开管程或等待另一条件。

采用哪种处理方式， 当然是各执一词。 但是Hansan却采用了第一种处理方式。

2.5.2 利用管程解决生产者-消费者问题

在利用管程方法来解决生产者-消费者问题时， 首先便是为它们建立一个管程，并命名为Proclucer-Consumer, 或简称为PC。其中包括两个过程：

(1) put(item)过程。生产者利用该过程将自己生产的产品投放到缓冲池中，并用整型变量count来表示在缓冲池中已有的产品数目，当count≥n时， 表示缓冲池已满， 生产者须等待。

(1) put(item)过程。 生产者利用该过程将自己生产的产品投放到缓冲池中， 并用整型变量count来表示在缓冲池中已有的产品数目，当count≥n时，表示缓冲池已满，生产者须等待。

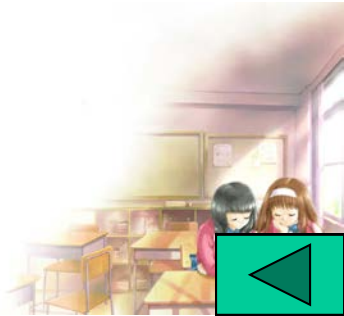
(2) get(item)过程。消费者利用该过程从缓冲池中取出一个产品，当count≤0时，表示缓冲池中已无可取用的产品，消费者应等待。

```
type producer-consumer=monitor
  Var in,out,count:integer;
   buffer:array［0,...,n-1］ of item;
   notfull, notempty:condition;
   procedure entry put(item)
   begin
    if count≥n then notfull.wait;
   buffer(in) ： = nextp;
   in ： = (in+1) mod n;
   count ： = count+1;
   if notempty.queue then notempty.signal;
    end
```

```
procedure entry get(item)
  begin
   if count≤0 then notempty.wait;
   nextc ： = buffer(out);
   out ： = (out+1) mod n;
   count ： = count-1;
   if notfull.quene then notfull.signal;
  end
begin in ： = out ： = 0; count ： = 0 end
```

在利用管程解决生产者-消费者问题时， 其中的生产者和消费者可描述为：

```
producer:begin
   repeat
    produce an item in nextp;
    PC.put(item);
   until false;
  end
 consumer:begin
   repeat
    PC.get(item);
    consume the item in nextc;
   until false;
  end
```



2.6 进 程 通 信

2.6.1 进程通信的类型

1. 共享存储器系统(Shared-Memory System)

(1) 基于共享数据结构的通信方式。

(2) 基于共享存储区的通信方式。

2. 消息传递系统(Message passing system)

不论是单机系统、多机系统，还是计算机网络，消息传递机制都是用得最广泛的一种进程间通信的机制。在消息传递系统中，进程间的数据交换，是以格式化的消息(message)为单位的；在计算机网络中，又把message称为报文。程序员直接利用系统提供的一组通信命令(原语)进行通信。操作系统隐藏了通信的实现细节，大大减化了通信程序编制的复杂性，而获得广泛的应用。消息传递系统的通信方式属于高级通信方式。又因其实现方式的不同而进一步分成直接通信方式和间接通信方式两种。

3. 管道(Pipe)通信

所谓“管道”，是指用于连接一个读进程和一个写进程以实现他们之间通信的一个共享文件，又名pipe文件。向管道(共享文件)提供输入的发送进程(即写进程)， 以字符流形式将大量的数据送入管道；而接受管道输出的接收进程(即读进程)，则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。这种方式首创于UNIX系统，由于它能有效地传送大量数据，因而又被引入到许多其它操作系统中。

为了协调双方的通信，管道机制必须提供以下三方面的协调能力：① 互斥，即当一个进程正在对pipe执行读/写操作时，其它(另一)进程必须等待。 ② 同步，指当写(输入)进程把一定数量(如4 KB)的数据写入pipe，便去睡眠等待， 直到读(输出)进程取走数据后，再把他唤醒。当读进程读一空pipe时，也应睡眠等待，直至写进程将数据写入管道后，才将之唤醒。③ 确定对方是否存在，只有确定了对方已存在时，才能进行通信。

2.6.2 消息传递通信的实现方法

1. 直接通信方式

这是指发送进程利用OS所提供的发送命令，直接把消息发送给目标进程。此时，要求发送进程和接收进程都以显式方式提供对方的标识符。通常，系统提供下述两条通信命令(原语)：

Send(Receiver, message); 发送一个消息给接收进程；

Receive(Sender, message); 接收Sender发来的消息；

例如，原语Send(P_2，m_1)表示将消息m_1发送给接收进程P_2；而原语Receive(P_1，m_1)则表示接收由P_1发来的消息m_1。

在某些情况下，接收进程可与多个发送进程通信，因此，它不可能事先指定发送进程。例如，用于提供打印服务的进程，它可以接收来自任何一个进程的“打印请求”消息。对于这样的应用，在接收进程接收消息的原语中的源进程参数，是完成通信后的返回值，接收原语可表示为：

Receive (id, message);

我们还可以利用直接通信原语，来解决生产者-消费者问题。当生产者生产出一个产品(消息)后，便用Send原语将消息发送给消费者进程；而消费者进程则利用Receive原语来得到一个消息。如果消息尚未生产出来，消费者必须等待，直至生产者进程将消息发送过来。生产者-消费者的通信过程可分别描述如下：

```
repeat
    ...
 produce an item in nextp;
    ...
 send(consumer, nextp);
until false;
repeat
 receive(producer, nextc);
    ...
 consume the item in nextc;
 until false;
```

2. 间接通信方式

(1) 信箱的创建和撤消。进程可利用信箱创建原语来建立一个新信箱。创建者进程应给出信箱名字、信箱属性(公用、私用或共享)；对于共享信箱， 还应给出共享者的名字。当进程不再需要读信箱时，可用信箱撤消原语将之撤消。

(2) 消息的发送和接收。当进程之间要利用信箱进行通信时，必须使用共享信箱，并利用系统提供的下述通信原语进行通信。

Send(mailbox, message); 将一个消息发送到指定信箱；

Receive(mailbox, message); 从指定信箱中接收一个消息；

信箱可由操作系统创建，也可由用户进程创建，创建者是信箱的拥有者。据此，可把信箱分为以下三类。

1) 私用信箱

用户进程可为自己建立一个新信箱，并作为该进程的一部分。信箱的拥有者有权从信箱中读取消息，其他用户则只能将自己构成的消息发送到该信箱中。这种私用信箱可采用单向通信链路的信箱来实现。 当拥有该信箱的进程结束时，信箱也随之消失。

2) 公用信箱

它由操作系统创建，并提供给系统中的所有核准进程使用。核准进程既可把消息发送到该信箱中，也可从信箱中读取发送给自己的消息。显然，公用信箱应采用双向通信链路的信箱来实现。通常，公用信箱在系统运行期间始终存在。

3) 共享信箱

它由某进程创建，在创建时或创建后，指明它是可共享的，同时须指出共享进程(用户)的名字。信箱的拥有者和共享者，都有权从信箱中取走发送给自己的消息。

在利用信箱通信时，在发送进程和接收进程之间，存在以下四种关系：

(1) 一对一关系。这时可为发送进程和接收进程建立一条两者专用的通信链路，使两者之间的交互不受其他进程的干扰。

(2) 多对一关系。允许提供服务的进程与多个用户进程之间进行交互，也称为客户/服务器交互(client/server interaction)。

(3) 一对多关系。允许一个发送进程与多个接收进程进行交互，使发送进程可用广播方式，向接收者(多个)发送消息。

(4) 多对多关系。允许建立一个公用信箱，让多个进程都能向信箱中投递消息；也可从信箱中取走属于自己的消息。

2.6.3 消息传递系统实现中的若干问题

1. 通信链路(communication link)

为使在发送进程和接收进程之间能进行通信，必须在两者之间建立一条通信链路。有两种方式建立通信链路。第一种方式是：由发送进程在通信之前，用显式的“建立连接”命令(原语)请求系统为之建立一条通信链路；在链路使用完后，也用显式方式拆除链路。

这种方式主要用于计算机网络中。第二种方式是发送进程无须明确提出建立链路的请求，只须利用系统提供的发送命令(原语)，系统会自动地为之建立一条链路。这种方式主要用于单机系统中。

根据通信链路的连接方法，又可把通信链路分为两类：① 点–点连接通信链路，这时的一条链路只连接两个结点(进程)；② 多点连接链路，指用一条链路连接多个(n＞2)结点(进程)。而根据通信方式的不同，则又可把链路分成两种： ① 单向通信链路，只允许发送进程向接收进程发送消息； ② 双向链路，既允许由进程A向进程B发送消息，也允许进程B同时向进程A发送消息。

2. 消息的格式

在某些OS中，消息是采用比较短的定长消息格式，这减少了对消息的处理和存储开销。这种方式可用于办公自动化系统中，为用户提供快速的便笺式通信；但这对要发送较长消息的用户是不方便的。在有的OS中，采用另一种变长的消息格式，即进程所发送消息的长度是可变的。系统在处理和存储变长消息时，须付出更多的开销，但方便了用户。 这两种消息格式各有其优缺点，故在很多系统(包括计算机网络)中，是同时都用的。

3. 进程同步方式

(1) 发送进程阻塞、 接收进程阻塞。

(2) 发送进程不阻塞、 接收进程阻塞。

(3) 发送进程和接收进程均不阻塞。

2.6.4 消息缓冲队列通信机制

1. 消息缓冲队列通信机制中的数据结构

(1) 消息缓冲区。在消息缓冲队列通信方式中，主要利用的数据结构是消息缓冲区。它可描述如下：

type message buffer=record

sender; 发送者进程标识符

size; 消息长度

text; 消息正文

next; 指向下一个消息缓冲区的指针

end

(2) PCB中有关通信的数据项。在利用消息缓冲队列通信机制时，在设置消息缓冲队列的同时，还应增加用于对消息队列进行操作和实现同步的信号量，并将它们置入进程的PCB中。在PCB中应增加的数据项可描述如下：

```
type processcontrol block=record
        ...
mq; 消息队列队首指针
mutex; 消息队列互斥信号量
sm; 消息队列资源信号量
    ...
end
```

2. 发送原语

发送进程在利用发送原语发送消息之前，应先在自己的内存空间，设置一发送区a，见图 2 - 12 所示，把待发送的消息正文、发送进程标识符、消息长度等信息填入其中，然后调用发送原语，把消息发送给目标(接收)进程。发送原语首先根据发送区a中所设置的消息长度a.size来申请一缓冲区i，接着，把发送区a中的信息复制到缓冲区i中。为了能将i挂在接收进程的消息队列mq上，应先获得接收进程的内部标识符j，然后将i挂在j.mq上。由于该队列属于临界资源，故在执行insert操作的前后，都要执行wait和signal操作。

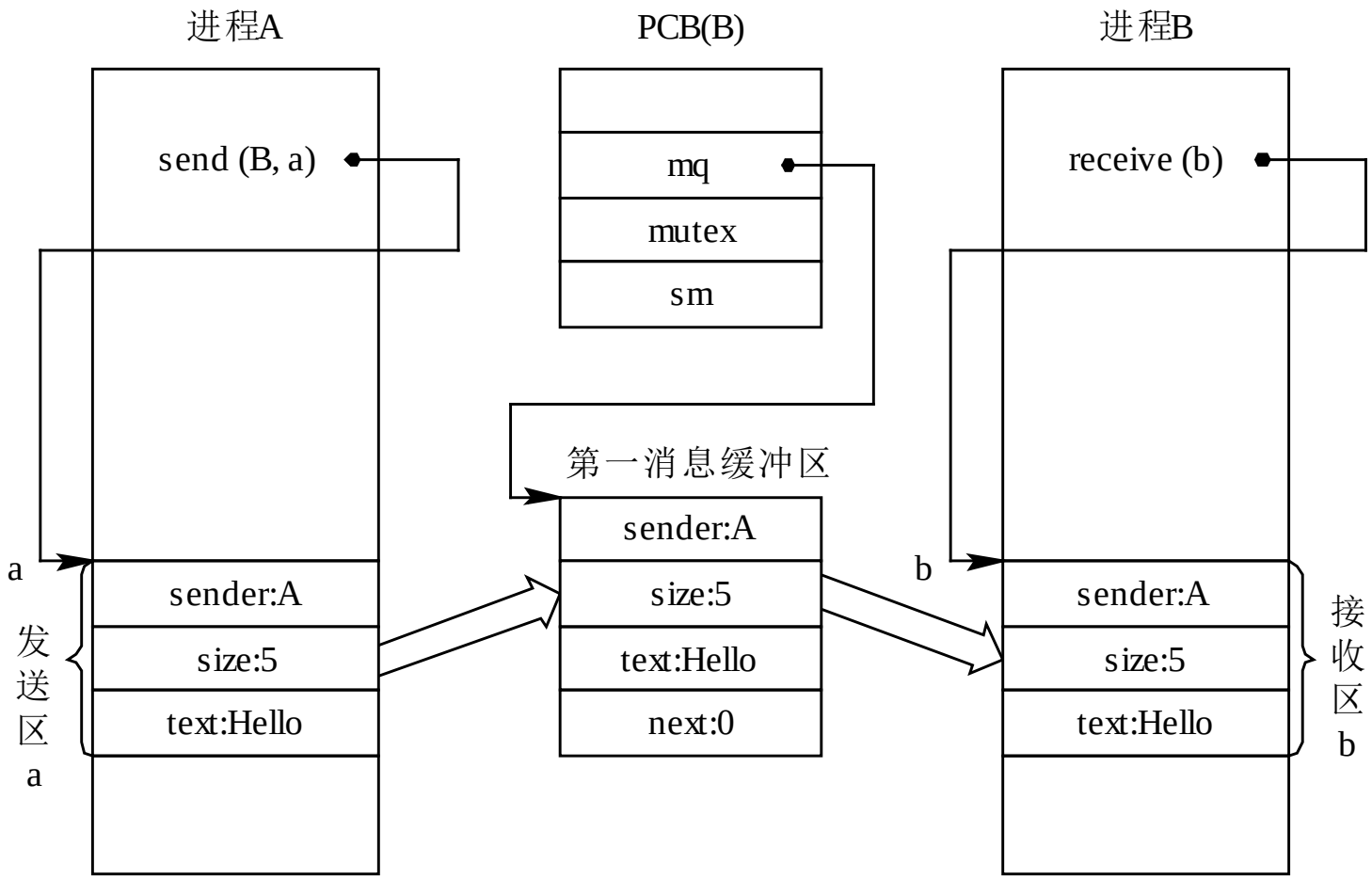

进程A
PCB(B)
进程B
send (B, a)
mq
mutex
sm
receive (b)
第一消息缓冲区
sender:A
size:5
text:Hello
next:0
a
发送区a
sender:A
size:5
text:Hello
b
sender:A
size:5
text:Hello
接收区b

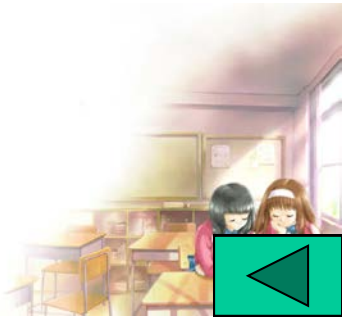
图 2 - 12 消息缓冲通信

```
procedure send(receiver, a)
  begin
   getbuf(a.size,i);        根据a.size申请缓冲区；
   i.sender： = a.sender;  将发送区a中的信息复制到消息缓冲区之中；
   i.size： = a.size;
   i.text： = a.text;
   i.next： = 0;
  getid(PCB set, receiver.j); 获得接收进程内部标识符；
  wait(j.mutex);
  insert(j.mq, i); 将消息缓冲区插入消息队列；
  signal(j.mutex);
  signal(j.sm);
 end
```

3. 接收原语

接收原语描述如下：

```
procedure receive(b)
 begin
  j ： = internal name; j为接收进程内部的标识符；
  wait(j.sm);
  wait(j.mutex);
  remove(j.mq, i); 将消息队列中第一个消息移出；
  signal(j.mutex);
  b.sender ： = i.sender; 将消息缓冲区i中的信息复制到接收区b;
  b.size ： = i.size;
  b.text ： = i.text;
  end
```



2.7 线 程

2.7.1 线程的基本概念

为使程序能并发执行，系统还必须进行以下的一系列操作。

1) 创建进程

2) 撤消进程

3) 进程切换

2. 线程的属性

(1) 轻型实体。

(2) 独立调度和分派的基本单位。

(3) 可并发执行。

(4) 共享进程资源。

3. 线程的状态

(1) 状态参数。

在OS中的每一个线程都可以利用线程标识符和一组状态参数进行描述。状态参数通常有这样几项： ① 寄存器状态，它包括程序计数器PC和堆栈指针中的内容； ② 堆栈， 在堆栈中通常保存有局部变量和返回地址； ③ 线程运行状态，用于描述线程正处于何种运行状态； ④ 优先级， 描述线程执行的优先程度； ⑤ 线程专有存储器， 用于保存线程自己的局部变量拷贝； ⑥ 信号屏蔽， 即对某些信号加以屏蔽。

(2) 线程运行状态。

如同传统的进程一样，在各线程之间也存在着共享资源和相互合作的制约关系，致使线程在运行时也具有间断性。相应地，线程在运行时，也具有下述三种基本状态：① 执行状态，表示线程正获得处理机而运行；② 就绪状态， 指线程已具备了各种执行条件，一旦获得CPU便可执行的状态；③ 阻塞状态，指线程在执行中因某事件而受阻，处于暂停执行时的状态。

4. 线程的创建和终止

在多线程OS环境下，应用程序在启动时，通常仅有一个线程在执行，该线程被人们称为“初始化线程”。它可根据需要再去创建若干个线程。在创建新线程时，需要利用一个线程创建函数(或系统调用)，并提供相应的参数，如指向线程主程序的入口指针、堆栈的大小，以及用于调度的优先级等。在线程创建函数执行完后，将返回一个线程标识符供以后使用。

终止线程的方式有两种：一种是在线程完成了自己的工作后自愿退出；另一种是线程在运行中出现错误或由于某种原因而被其它线程强行终止。

5. 多线程OS中的进程

在多线程OS中，进程是作为拥有系统资源的基本单位，通常的进程都包含多个线程并为它们提供资源，但此时的进程就不再作为一个执行的实体。 多线程OS中的进程有以下属性：

(1) 作为系统资源分配的单位。

(2) 可包括多个线程。

(3) 进程不是一个可执行的实体。

2.7.2 线程间的同步和通信

1. 互斥锁(mutex)

互斥锁是一种比较简单的、用于实现进程间对资源互斥访问的机制。由于操作互斥锁的时间和空间开锁都较低，因而较适合于高频度使用的关键共享数据和程序段。互斥锁可以有两种状态， 即开锁(unlock)和关锁(lock)状态。相应地，可用两条命令(函数)对互斥锁进行操作。其中的关锁lock操作用于将mutex关上，开锁操作unlock则用于打开mutex。

2. 条件变量

每一个条件变量通常都与一个互斥锁一起使用，亦即，在创建一个互斥锁时便联系着一个条件变量。单纯的互斥锁用于短期锁定，主要是用来保证对临界区的互斥进入。而条件变量则用于线程的长期等待， 直至所等待的资源成为可用的。

线程首先对mutex执行关锁操作，若成功便进入临界区，然后查找用于描述资源状态的数据结构，以了解资源的情况。只要发现所需资源R正处于忙碌状态，线程便转为等待状态，并对mutex执行开锁操作后，等待该资源被释放； 若资源处于空闲状态，表明线程可以使用该资源，于是将该资源设置为忙碌状态，再对mutex执行开锁操作。

下面给出了对上述资源的申请(左半部分)和释放(右半部分)操作的描述。

```
 Lock mutex            Lock mutex
check data structures;                mark resource as free;
while(resource busy);                 unlock mutex;
 wait(condition variable);  wakeup(condition variable);
mark resource as busy;
 unlock mutex;
```

3. 信号量机制

(1) 私用信号量(private samephore)。

当某线程需利用信号量来实现同一进程中各线程之间的同步时，可调用创建信号量的命令来创建一私用信号量，其数据结构是存放在应用程序的地址空间中。私用信号量属于特定的进程所有，OS并不知道私用信号量的存在，因此，一旦发生私用信号量的占用者异常结束或正常结束，但并未释放该信号量所占有空间的情况时，系统将无法使它恢复为0(空)，也不能将它传送给下一个请求它的线程。

(2) 公用信号量(public semaphort)。

公用信号量是为实现不同进程间或不同进程中各线程之间的同步而设置的。由于它有着一个公开的名字供所有的进程使用，故而把它称为公用信号量。其数据结构是存放在受保护的系统存储区中，由OS为它分配空间并进行管理，故也称为系统信号量。如果信号量的占有者在结束时未释放该公用信号量，则OS会自动将该信号量空间回收，并通知下一进程。可见，公用信号量是一种比较安全的同步机制。

2.7.3 内核支持线程和用户级线程

1. 内核支持线程

这里所谓的内核支持线程，也都同样是在内核的支持下运行的，即无论是用户进程中的线程，还是系统进程中的线程，他们的创建、撤消和切换等，也是依靠内核实现的。此外，在内核空间还为每一个内核支持线程设置了一个线程控制块，内核是根据该控制块而感知某线程的存在的，并对其加以控制。

2. 用户级线程

用户级线程仅存在于用户空间中。对于这种线程的创建、 撤消、线程之间的同步与通信等功能，都无须利用系统调用来实现。对于用户级线程的切换，通常是发生在一个应用进程的诸多线程之间，这时，也同样无须内核的支持。由于切换的规则远比进程调度和切换的规则简单，因而使线程的切换速度特别快。可见，这种线程是与内核无关的。

2.7.4 线程控制

1. 内核支持线程的实现

PTDA 进程资源

TCB # 1

TCB # 2

TCB # 3

图 2 - 13 任务数据区空间

2. 用户级线程的实现

1) 运行时系统(Runtime System)

所谓“运行时系统”，实质上是用于管理和控制线程的函数(过程)的集合，其中包括用于创建和撤消线程的函数、线程同步和通信的函数以及实现线程调度的函数等。正因为有这些函数，才能使用户级线程与内核无关。运行时系统中的所有函数都驻留在用户空间，并作为用户级线程与内核之间的接口。

2) 内核控制线程

这种线程又称为轻型进程LWP(Light Weight Process)。每一个进程都可拥有多个LWP，同用户级线程一样，每个LWP都有自己的数据结构(如TCB)，其中包括线程标识符、优先级、状态，另外还有栈和局部存储区等。它们也可以共享进程所拥有的资源。LWP可通过系统调用来获得内核提供的服务，这样，当一个用户级线程运行时，只要将它连接到一个LWP上，此时它便具有了内核支持线程的所有属性。

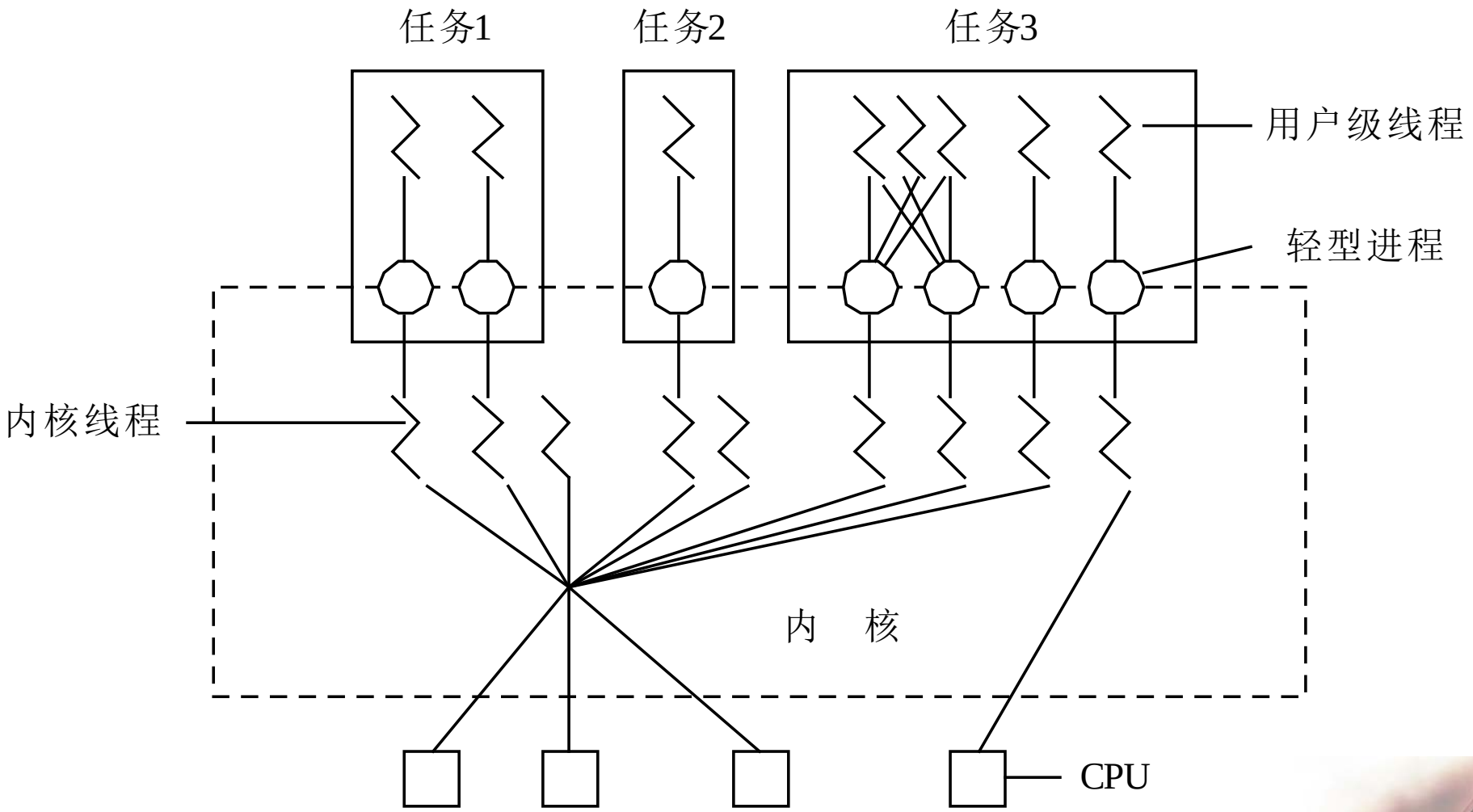


图 2 - 14 利用轻型进程作为中间系统

第三章 处理机调度与死锁

3.1 处理机调度的基本概念

3.1.1 高级、中级和低级调度

1. 高级调度(High Scheduling)

在每次执行作业调度时，都须做出以下两个决定。

1) 接纳多少个作业

2) 接纳哪些作业

2. 低级调度(Low Level Scheduling)

1) 非抢占方式(Non-preemptive Mode)

在采用非抢占调度方式时，可能引起进程调度的因素可归结为这样几个：① 正在执行的进程执行完毕， 或因发生某事件而不能再继续执行； ② 执行中的进程因提出I/O请求而暂停执行；③ 在进程通信或同步过程中执行了某种原语操作，如P操作(wait操作)、Block原语、Wakeup原语等。这种调度方式的优点是实现简单、系统开销小，适用于大多数的批处理系统环境。但它难以满足紧急任务的要求—立即执行，因而可能造成难以预料的后果。显然，在要求比较严格的实时系统中，不宜采用这种调度方式。

2) 抢占方式(Preemptive Mode)

抢占的原则有：

(1) 优先权原则。

(2) 短作业(进程)优先原则。

(3) 时间片原则。

3. 中级调度(Intermediate-Level Scheduling)

中级调度又称中程调度(Medium-Term Scheduling)。 引入中级调度的主要目的，是为了提高内存利用率和系统吞吐量。 为此，应使那些暂时不能运行的进程不再占用宝贵的内存资源，而将它们调至外存上去等待，把此时的进程状态称为就绪驻外存状态或挂起状态。当这些进程重又具备运行条件、且内存又稍有空闲时，由中级调度来决定把外存上的哪些又具备运行条件的就绪进程，重新调入内存，并修改其状态为就绪状态，挂在就绪队列上等待进程调度。

3.1.2 调度队列模型

1. 仅有进程调度的调度队列模型

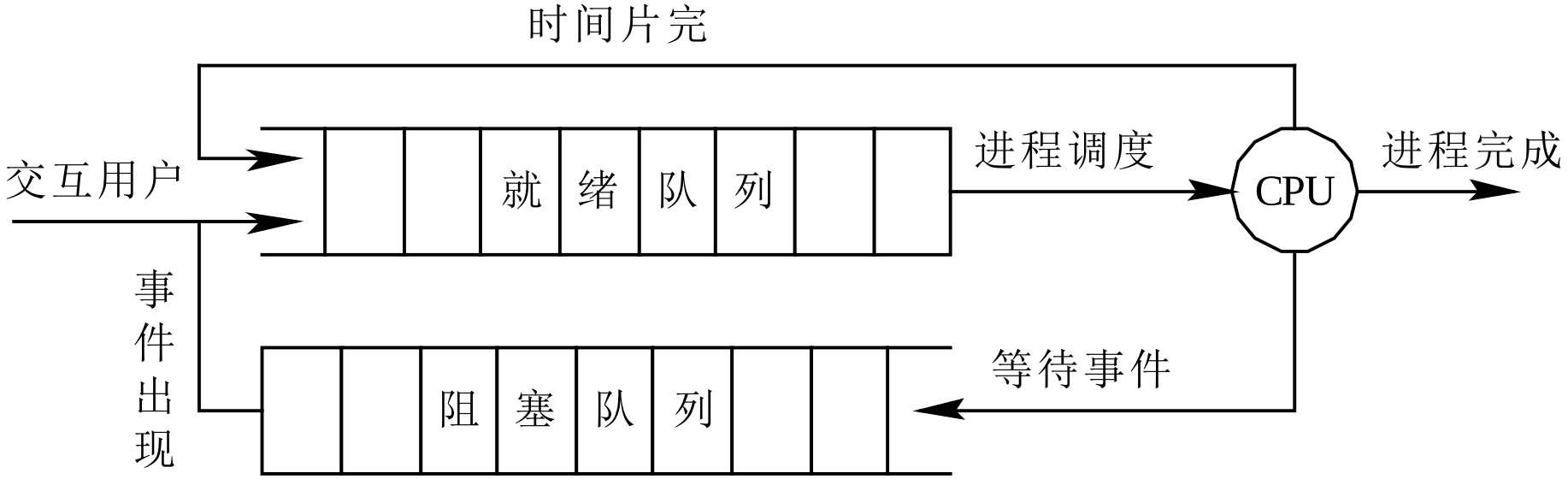


图 3 - 1 仅具有进程调度的调度队列模型

2. 具有高级和低级调度的调度队列模型

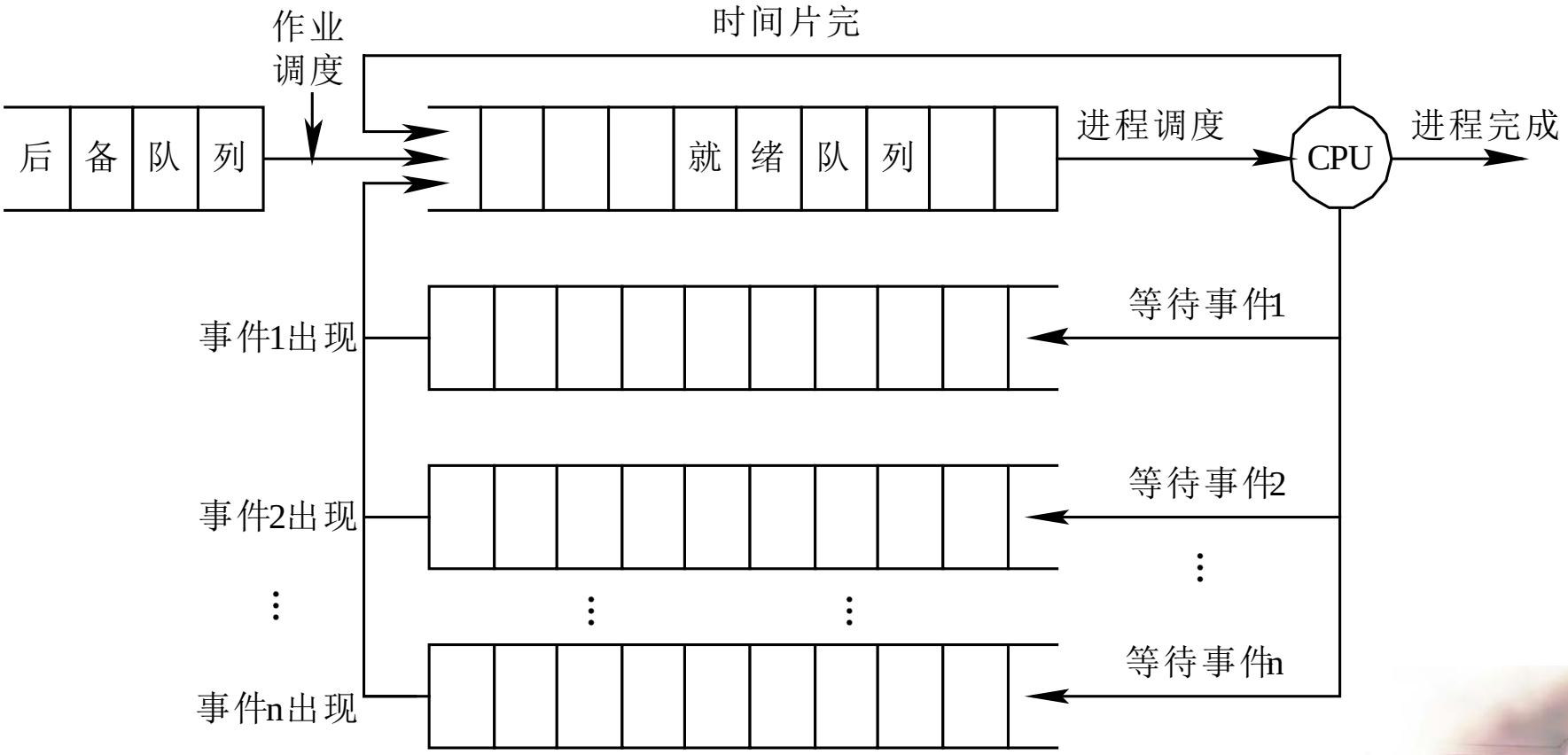


图 3-2 具有高、低两级调度的调度队列模型

图 3-2 示出了具有高、低两级调度的调度队列模型。该模型与上一模型的主要区别在于如下两个方面。

(1) 就绪队列的形式。

(2) 设置多个阻塞队列。

3. 同时具有三级调度的调度队列模型

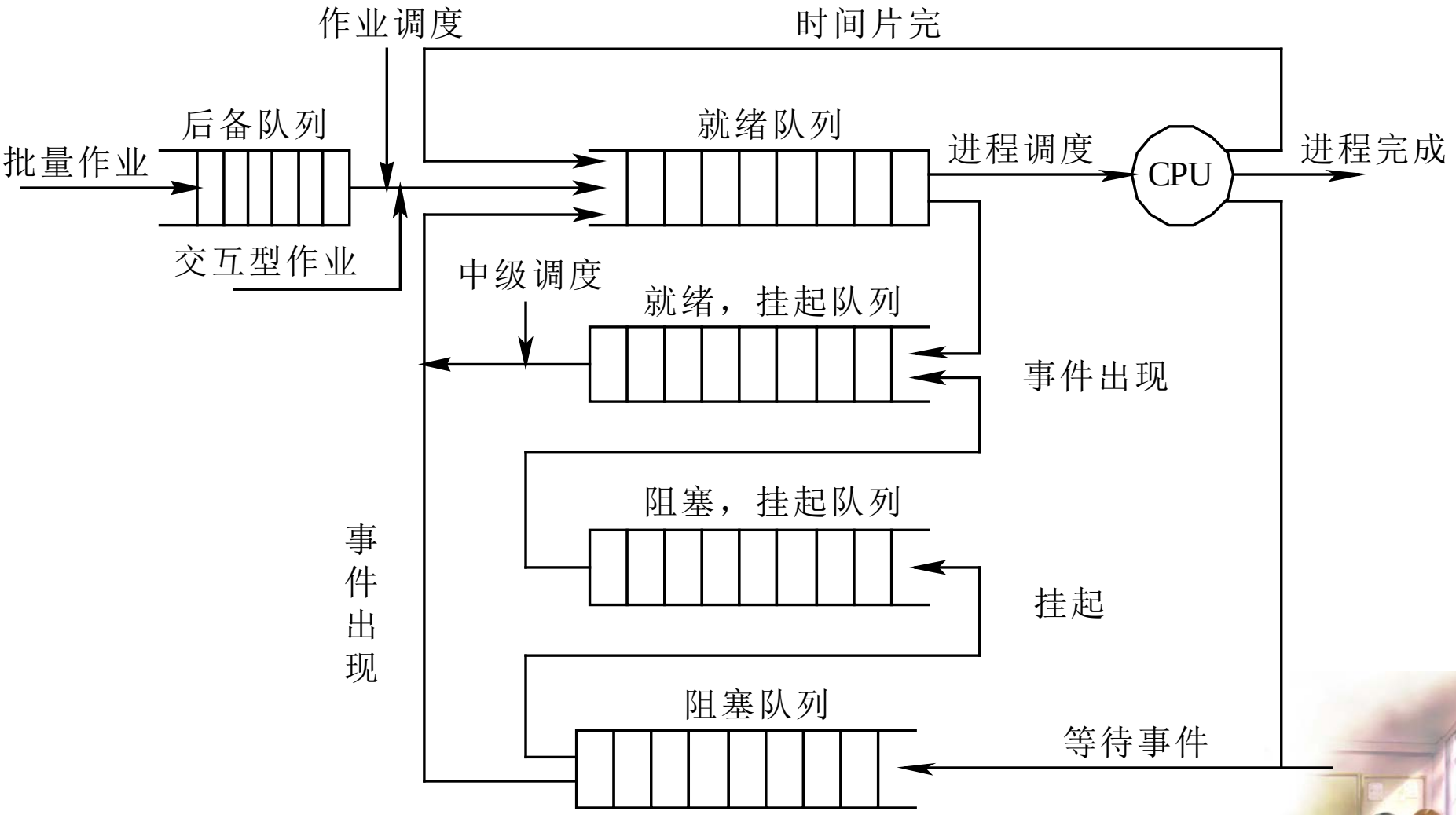


图 3-3 具有三级调度时的调度队列模型

3.1.3 选择调度方式和调度算法的若干准则

1. 面向用户的准则

(1) 周转时间短。

可把平均周转时间描述为：$$T=\frac{1}{n}\left[\sum_{i=1}^{i}T_i\right]$$

作业的周转时间T与系统为它提供服务的时间T_S之比，即$W=T/T_S$，称为带权周转时间，而平均带权周转时间则可表示为:

$$W=\frac{1}{n}\left[\sum_{i=1}^{n}\frac{T_i}{T_{Si}}\right]$$

(2) 响应时间快。

(3) 截止时间的保证。

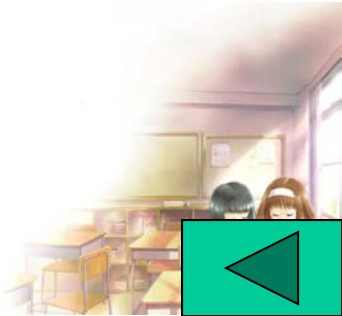
(4) 优先权准则。

2. 面向系统的准则

(1) 系统吞吐量高。

(2) 处理机利用率好。

(3) 各类资源的平衡利用。



3.2 调 度 算 法

3.2.1 先来先服务和短作业(进程)优先调度算法

1. 先来先服务调度算法

进程名	到达时间	服务时间	开始执行时间	完成时间	周转时间	带权周转时间
A	0	1	0	1	1	1
B	1	100	1	101	100	1
C	2	1	101	102	100	100
D	3	100	102	202	199	1.99

调度算法 \ 作业情况	进程名	A	B	C	D	E	平 均
	到达时间	0	1	2	3	4	
	服务时间	4	3	5	2	4	
FCFS (a)	完成时间	4	7	12	14	18	
	周转时间	4	6	10	11	14	9
	带权周转时间	1	2	2	5.5	3.5	2.8
SJF (b)	完成时间	4	9	18	6	13	
	周转时间	4	8	16	3	9	8
	带权周转时间	1	2.67	3.1	1.5	2.25	2.1

图 3-4 FCFS和SJF调度算法的性能

2. 短作业(进程)优先调度算法

短作业(进程)优先调度算法SJ(P)F，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的调度算法，是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(SPF)调度算法，则是从就绪队列中选出一估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时，再重新调度。

SJ(P)F调度算法也存在不容忽视的缺点：

(1) 该算法对长作业不利，如作业C的周转时间由10增至16，其带权周转时间由2增至3.1。更严重的是，如果有一长作业(进程)进入系统的后备队列(就绪队列)，由于调度程序总是优先调度那些(即使是后进来的)短作业(进程)，将导致长作业(进程)长期不被调度。

(2) 该算法完全未考虑作业的紧迫程度，因而不能保证紧迫性作业(进程)会被及时处理。

(3) 由于作业(进程)的长短只是根据用户所提供的估计执行时间而定的，而用户又可能会有意或无意地缩短其作业的估计运行时间，致使该算法不一定能真正做到短作业优先调度。

3.2.2 高优先权优先调度算法

1. 优先权调度算法的类型

1) 非抢占式优先权算法

在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

2) 抢占式优先权调度算法

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程i时，就将其优先权P_i与正在执行的进程j的优先权P_j进行比较。如果$P_i \leq P_j$，原进程P_j便继续执行；但如果是$P_i > P_j$，则立即停止P_j的执行，做进程切换，使i进程投入执行。显然，这种抢占式的优先权调度算法，能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中， 以及对性能要求较高的批处理和分时系统中。

2. 优先权的类型

1) 静态优先权

静态优先权是在创建进程时确定的，且在进程的整个运行期间保持不变。一般地，优先权是利用某一范围内的一个整数来表示的，例如，0~7或0~255中的某一整数， 又把该整数称为优先数。只是具体用法各异：有的系统用“0”表示最高优先权，当数值愈大时，其优先权愈低；而有的系统恰恰相反。

确定进程优先权的依据有如下三个方面：

(1) 进程类型。

(2) 进程对资源的需求。

(3) 用户要求。

2) 动态优先权

动态优先权是指，在创建进程时所赋予的优先权，是可以随进程的推进或随其等待时间的增加而改变的，以便获得更好的调度性能。例如，我们可以规定，在就绪队列中的进程，随其等待时间的增长，其优先权以速率a提高。若所有的进程都具有相同的优先权初值，则显然是最先进入就绪队列的进程，将因其动态优先权变得最高而优先获得处理机，此即FCFS算法。若所有的就绪进程具有各不相同的优先权初值，那么，对于优先权初值低的进程，在等待了足够的时间后，其优先权便可能升为最高，从而可以获得处理机。当采用抢占式优先权调度算法时，如果再规定当前进程的优先权以速率b下降，则可防止一个长作业长期地垄断处理机。

3. 高响应比优先调度算法

优先权的变化规律可描述为：

$$\text{优先权} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}}$$

由于等待时间与服务时间之和，就是系统对该作业的响应时间，故该优先权又相当于响应比R_P。据此，又可表示为：

$$\text{优先权} = \frac{\text{等待时间} + \text{要求服务时间}}{\text{要求服务时间}} = \frac{\text{响应时间}}{\text{要求服务时间}}$$

(1) 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。

(2) 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。

(3) 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高， 从而也可获得处理机。

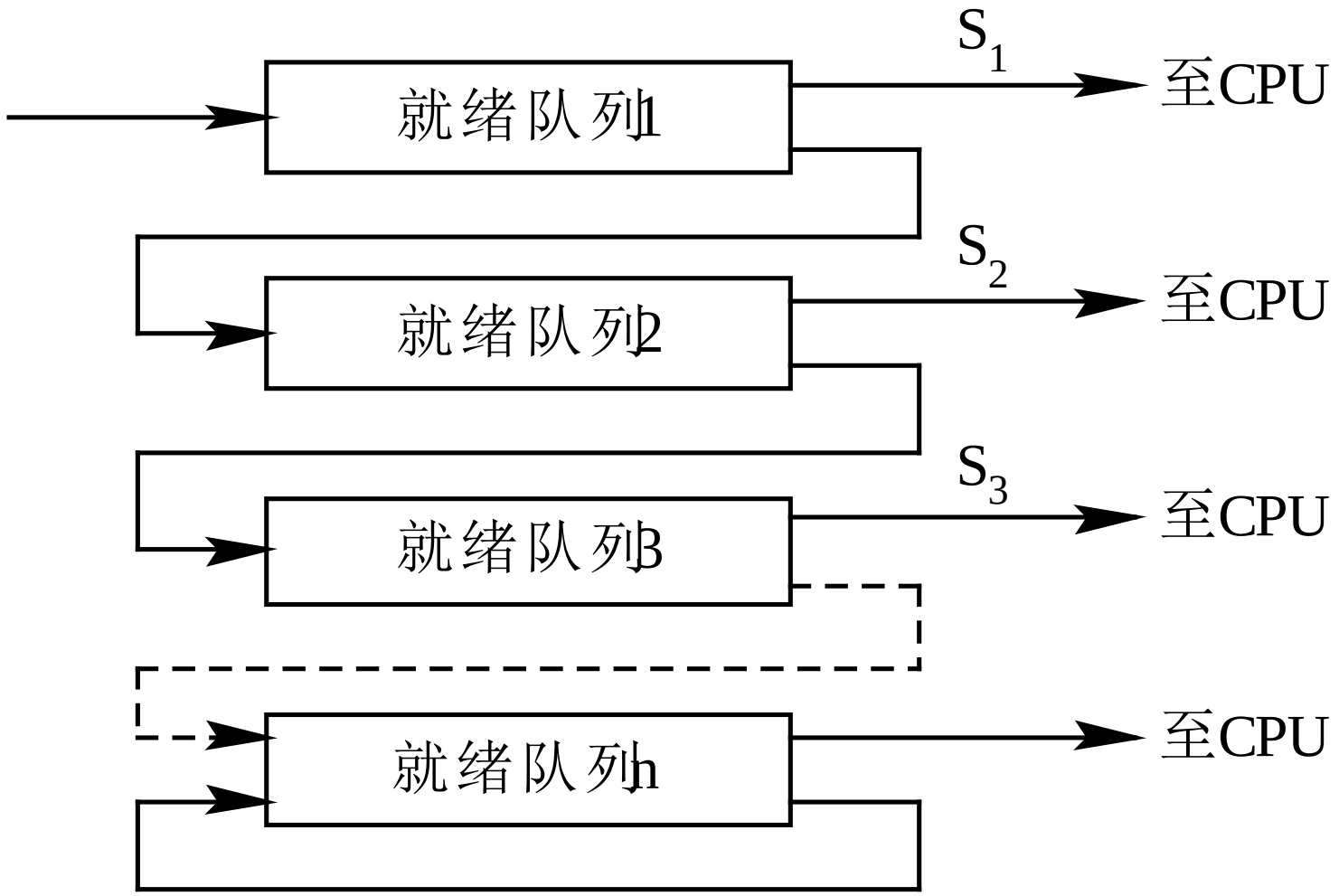
3.2.3 基于时间片的轮转调度算法

1. 时间片轮转法

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则，排成一个队列，每次调度时，把CPU分配给队首进程，并令其执行一个时间片。时间片的大小从几ms到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程，在一给定的时间内，均能获得一时间片的处理机执行时间。

2. 多级反馈队列调度算法

(1) 应设置多个就绪队列，并为各个队列赋予不同的优先级。 第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第i+1个队列的时间片要比第i个队列的时间片长一倍。 图 3-5 是多级反馈队列算法的示意。



（时间片：$S_1 < S_2 < S_3$）

图 3-5 多级反馈队列调度算法

(2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按FCFS原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，.....，如此下去，当一个长作业(进程)从第一队列依次降到第n队列后，在第n队列中便采取按时间片轮转的方式运行。

(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行； 仅当第1~(i-1) 队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1~(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i队列的末尾，把处理机分配给新到的高优先权进程。

3. 多级反馈队列调度算法的性能

(1) 终端型作业用户。

(2) 短批处理作业用户。

(3) 长批处理作业用户。

3.3 实 时 调 度

3.3.1 实现实时调度的基本条件

1. 提供必要的信息

(1) 就绪时间。

(2) 开始截止时间和完成截止时间。

(3) 处理时间。

(4) 资源要求。

(5) 优先级。

2. 系统处理能力强

在实时系统中，通常都有着多个实时任务。若处理机的处理能力不够强，则有可能因处理机忙不过来而使某些实时任务不能得到及时处理， 从而导致发生难以预料的后果。假定系统中有m个周期性的硬实时任务，它们的处理时间可表示为C_i，周期时间表示为P_i，则在单处理机情况下，必须满足下面的限制条件：

$$\sum_{i=1}^{m}\frac{C_i}{P_i}\leq 1$$

系统才是可调度的。假如系统中有6个硬实时任务，它们的周期时间都是 50 ms，而每次的处理时间为 10 ms，则不难算出，此时是不能满足上式的，因而系统是不可调度的。

解决的方法是提高系统的处理能力，其途径有二：其一仍是采用单处理机系统， 但须增强其处理能力， 以显著地减少对每一个任务的处理时间；其二是采用多处理机系统。假定系统中的处理机数为N，则应将上述的限制条件改为：

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq N$$

3. 采用抢占式调度机制

当一个优先权更高的任务到达时，允许将当前任务暂时挂起，而令高优先权任务立即投入运行，这样便可满足该硬实时任务对截止时间的要求。但这种调度机制比较复杂。

对于一些小的实时系统，如果能预知任务的开始截止时间，则对实时任务的调度可采用非抢占调度机制，以简化调度程序和对任务调度时所花费的系统开销。但在设计这种调度机制时，应使所有的实时任务都比较小，并在执行完关键性程序和临界区后，能及时地将自己阻塞起来，以便释放出处理机， 供调度程序去调度那种开始截止时间即将到达的任务。

4. 具有快速切换机制

该机制应具有如下两方面的能力：

(1) 对外部中断的快速响应能力。为使在紧迫的外部事件请求中断时系统能及时响应，要求系统具有快速硬件中断机构，还应使禁止中断的时间间隔尽量短， 以免耽误时机(其它紧迫任务)。

(2) 快速的任务分派能力。在完成任务调度后，便应进行任务切换。为了提高分派程序进行任务切换时的速度，应使系统中的每个运行功能单位适当的小，以减少任务切换的时间开销。

3.3.2 实时调度算法的分类

1. 非抢占式调度算法

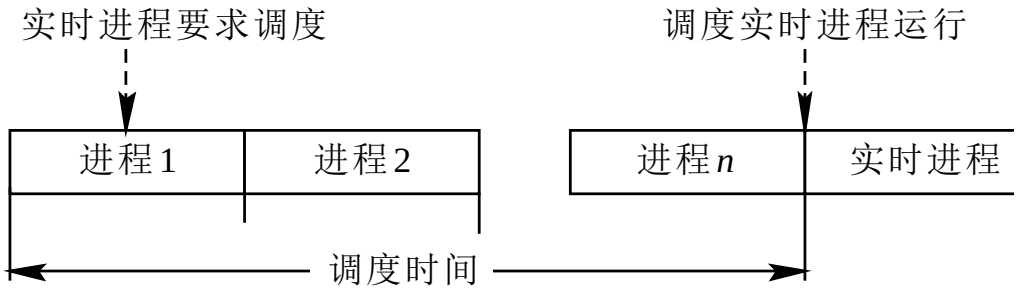
(1) 非抢占式轮转调度算法。

(2) 非抢占式优先调度算法。

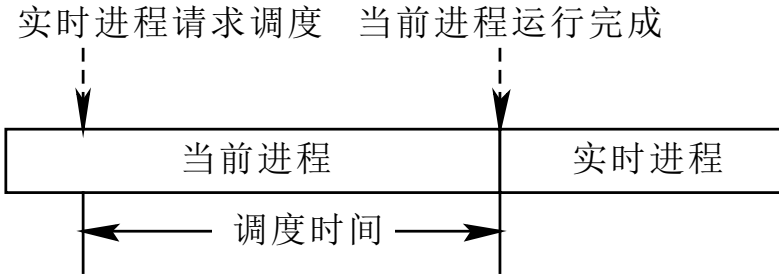
2. 抢占式调度算法

(1) 基于时钟中断的抢占式优先权调度算法。

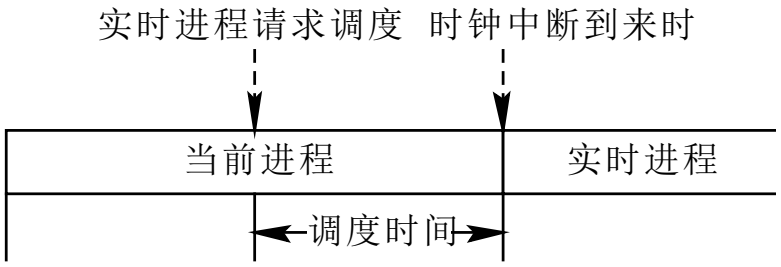
(2) 立即抢占(Immediate Preemption)的优先权调度算法。



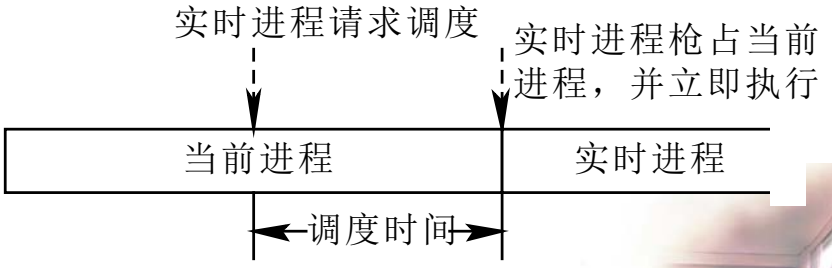
(*a*) 非抢占轮转调度



(*b*) 非抢占优先权调度



(*c*) 基于时钟中断抢占的优先权抢占调度



(*d*) 立即抢占的优先权调度

图 3-6 实时进程调度

3.3.3 常用的几种实时调度算法

1. 最早截止时间优先即EDF(Earliest Deadline First)算法

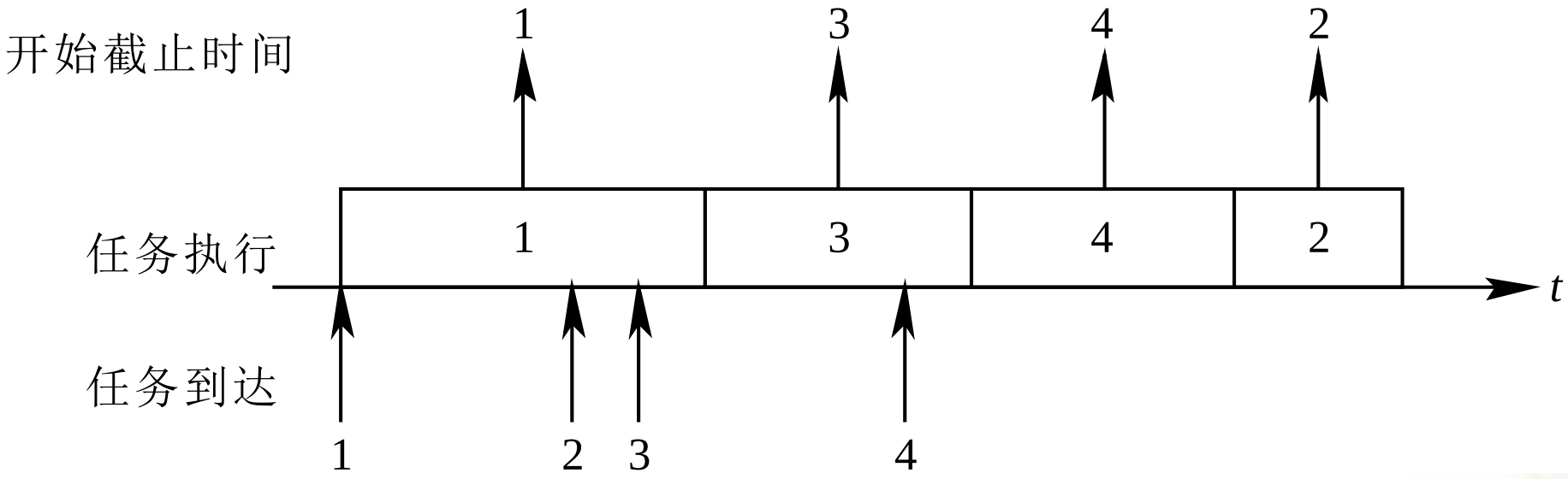


图 3-7 EDF算法用于非抢占调度方式

2. 最低松弛度优先即LLF(Least Laxity First)算法

该算法是根据任务紧急(或松弛)的程度，来确定任务的优先级。任务的紧急程度愈高，为该任务所赋予的优先级就愈高，以使之优先执行。例如，一个任务在200ms时必须完成，而它本身所需的运行时间就有100ms，因此，调度程序必须在100 ms之前调度执行，该任务的紧急程度(松弛程度)为100 ms。又如，另一任务在400 ms时必须完成，它本身需要运行 150 ms，则其松弛程度为 250 ms。在实现该算法时要求系统中有一个按松弛度排序的实时任务就绪队列，松弛度最低的任务排在队列最前面，调度程序总是选择就绪队列中的队首任务执行。该算法主要用于可抢占调度方式中。假如在一个实时系统中，有两个周期性实时任务A和B，任务A要求每 20 ms执行一次，执行时间为 10 ms；任务B只要求每50 ms执行一次，执行时间为 25 ms。

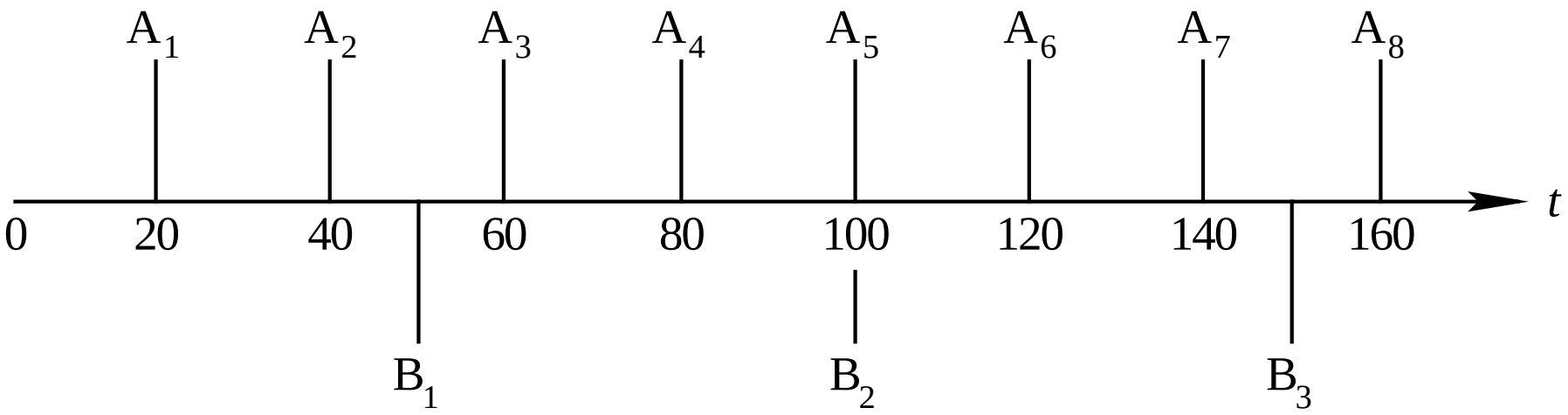


图 3-8 A和B任务每次必须完成的时间

在刚开始时(t_1=0)，A_1必须在20ms时完成，而它本身运行又需 10 ms，可算出A_1的松弛度为10ms；B_1必须在50ms时完成， 而它本身运行就需25 ms，可算出B_1的松弛度为25 ms，故调度程序应先调度A_1执行。在t_2=10 ms时，A_2的松弛度可按下式算出：

A_2的松弛度=必须完成时间-其本身的运行时间-当前时间

=40 ms-10 ms-10 ms=20 ms

类似地，可算出B_1的松弛度为15ms，故调度程序应选择B_2运行。在t_3=30 ms时，A_2的松弛度已减为0(即40-10-30)，而B_1的松弛度为15 ms(即50-5-30)，于是调度程序应抢占B_1的处理机而调度A_2运行。在t_4=40 ms时，A_3的松弛度为10 ms(即60-10-40)，而B_1的松弛度仅为5 ms(即50-5-40)，故又应重新调度B_1执行。在t_5=45 ms时，B_1执行完成，而此时A_3的松弛度已减为5 ms(即60-10-45)，而B_2的松弛度为30 ms(即100-25-45)，于是又应调度A_3执行。在t_6=55ms时，任务A尚未进入第4周期，而任务B已进入第2周期，故再调度B_2执行。在t_7=70 ms时，A_4的松弛度已减至0 ms(即80-10-70)，而B_2的松弛度为20 ms(即100-10-70)，故此时调度又应抢占B_2的处理机而调度A_4执行。

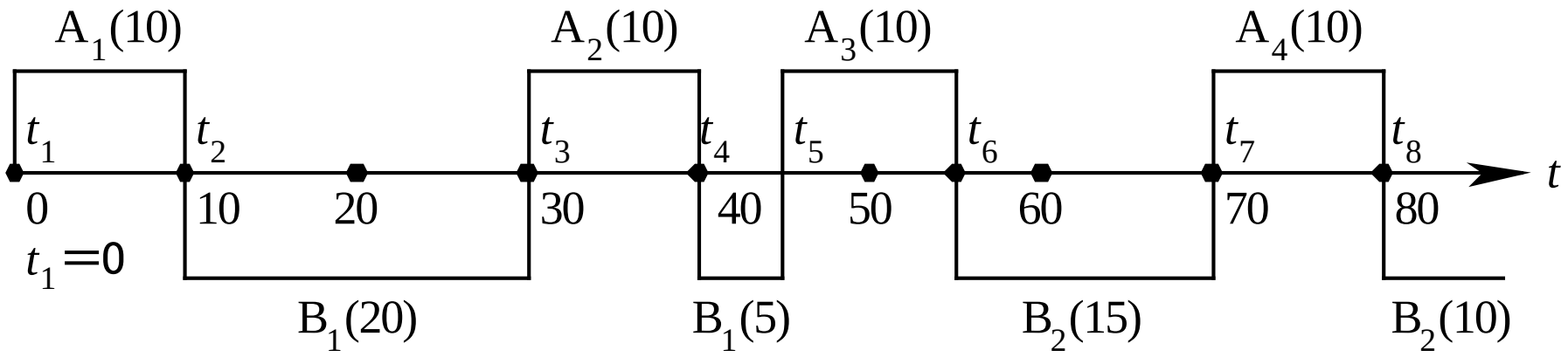


图 3-9 利用ELLF算法进行调度的情况

3.4 多处理机系统中的调度

3.4.1 多处理器系统的类型

(1) 紧密耦合(Tightly Coupted)MPS。

这通常是通过高速总线或高速交叉开关，来实现多个处理器之间的互连的。它们共享主存储器系统和I/O设备，并要求将主存储器划分为若干个能独立访问的存储器模块，以便多个处理机能同时对主存进行访问。系统中的所有资源和进程，都由操作系统实施统一的控制和管理。

(2) 松散耦合(Loosely Coupled)MPS。

在松散耦合MPS中，通常是通过通道或通信线路，来实现多台计算机之间的互连。每台计算机都有自己的存储器和I/O设备，并配置了OS来管理本地资源和在本地运行的进程。因此，每一台计算机都能独立地工作， 必要时可通过通信线路与其它计算机交换信息，以及协调它们之间的工作。

2. 对称多处理器系统和非对称多处理器系统

(1) 对称多处理器系统SMPS(Symmetric MultiProcessor System)。 在系统中所包含的各处理器单元，在功能和结构上都是相同的， 当前的绝大多数MPS都属于SMP系统。例如，IBM公司的SR/6000 Model F50, 便是利用4片Power PC处理器构成的。

(2) 非对称多处理器系统。在系统中有多种类型的处理单元， 它们的功能和结构各不相同，其中只有一个主处理器，有多个从处理器。

3.4.2 进程分配方式

1. 对称多处理器系统中的进程分配方式

在SMP系统中，所有的处理器都是相同的，因而可把所有的处理器作为一个处理器池(Processor pool)，由调度程序或基于处理器的请求， 将任何一个进程分配给池中的任何一个处理器去处理。在进行进程分配时，可采用以下两种方式之一。

1) 静态分配(Static Assigenment)方式

2) 动态分配(Dynamic Assgement)方式

2. 非对称MPS中的进程分配方式

对于非对称MPS， 其OS大多采用主–从(Master-Slave)式OS， 即OS的核心部分驻留在一台主机上(Master)， 而从机(Slave)上只是用户程序， 进程调度只由主机执行。 每当从机空闲时， 便向主机发送一索求进程的信号， 然后， 便等待主机为它分配进程。 在主机中保持有一个就绪队列， 只要就绪队列不空， 主机便从其队首摘下一进程分配给请求的从机。从机接收到分配的进程后便运行该进程， 该进程结束后从机又向主机发出请求。

3.4.3 进程(线程)调度方式

1. 自调度(Self-Scheduling)方式

1) 自调度机制

在多处理器系统中，自调度方式是最简单的一种调度方式。它是直接由单处理机环境下的调度方式演变而来的。 在系统中设置有一个公共的进程或线程就绪队列， 所有的处理器在空闲时，都可自己到该队列中取得一进程(或线程)来运行。在自调度方式中，可采用在单处理机环境下所用的调度算法，如先来先服务(FCFS)调度算法、最高优先权优先(FPF)调度算法和抢占式最高优先权优先调度算法等。

2) 自调度方式的优点

自调度方式的主要优点表现为：首先，系统中的公共就绪队列可按照单处理机系统中所采用的各种方式加以组织；其调度算法也可沿用单处理机系统所用的算法，亦即，很容易将单处理机环境下的调度机制移植到多处理机系统中，故它仍然是当前多处理机系统中较常用的调度方式。其次，只要系统中有任务，或者说只要公共就绪队列不空，就不会出现处理机空闲的情况，也不会发生处理器忙闲不均的现象，因而有利于提高处理器的利用率。

3) 自调度方式的缺点

(1) 瓶颈问题。

(2) 低效性。

(3) 线程切换频繁。

2. 成组调度(Gang Scheduling)方式

在成组调度时，如何为应用程序分配处理器时间，

1) 面向所有应用程序平均分配处理器时间

2) 面向所有线程平均分配处理器时间

	应用程序 A	应用程序 B
处理器 1	线程 1	线程 1
处理器 2	线程 2	空闲
处理器 3	线程 3	空闲
处理器 4	线程 4	空闲
	1/2	1/2

(a) 浪费 37.5%

	应用程序 A	应用程序 B
处理器 1	线程 1	线程 1
处理器 2	线程 2	空闲
处理器 3	线程 3	空闲
处理器 4	线程 4	空闲
	4/5	1/5

(b) 浪费 15%

图 3 - 10 两种分配处理器时间的方法

3. 专用处理器分配(Dedicated Processor Assigement)方式

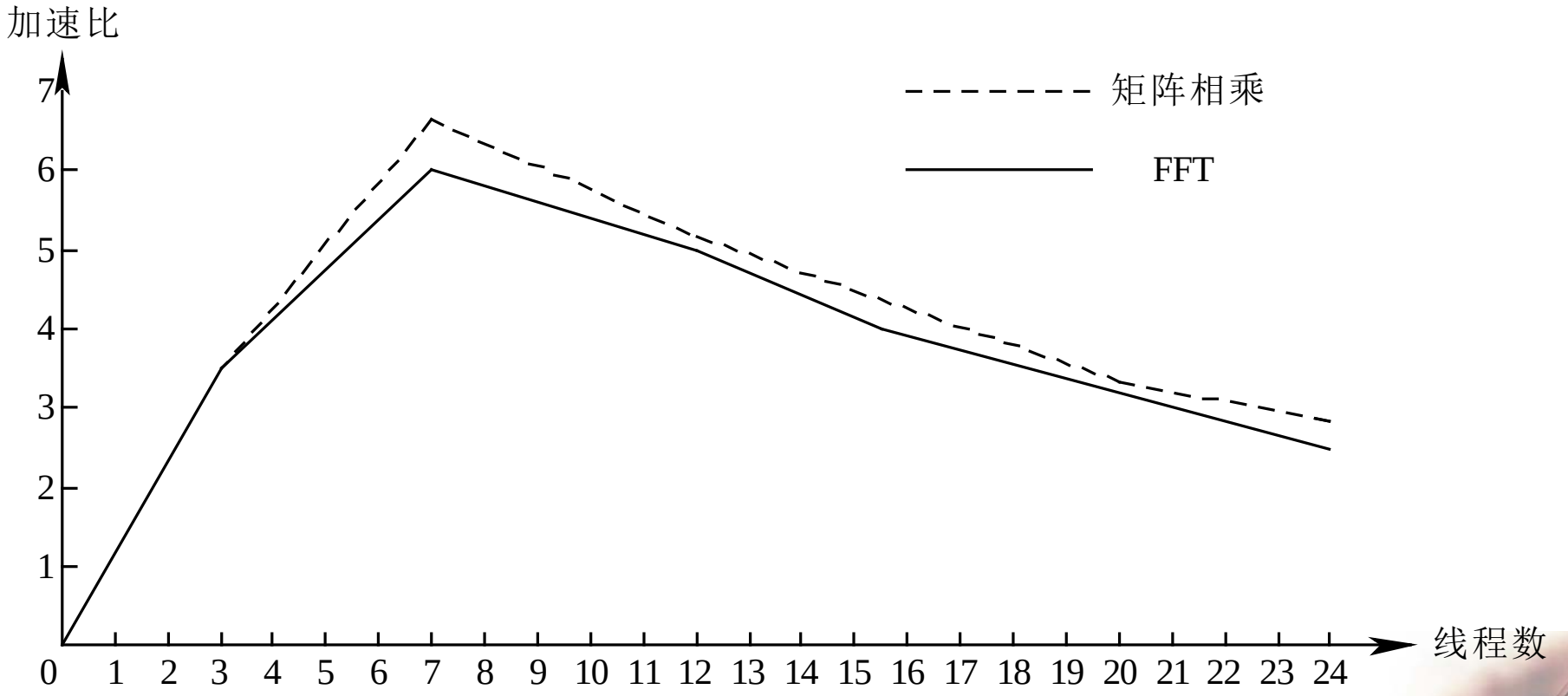


图 3-11 线程数对加速比的影响

3.5 产生死锁的原因和必要条件

3.5.1 产生死锁的原因

(1) 竞争资源。

(2) 进程间推进顺序非法。

1. 竞争资源引起进程死锁

1) 可剥夺和非剥夺性资源

2) 竞争非剥夺性资源

3) 竞争临时性资源

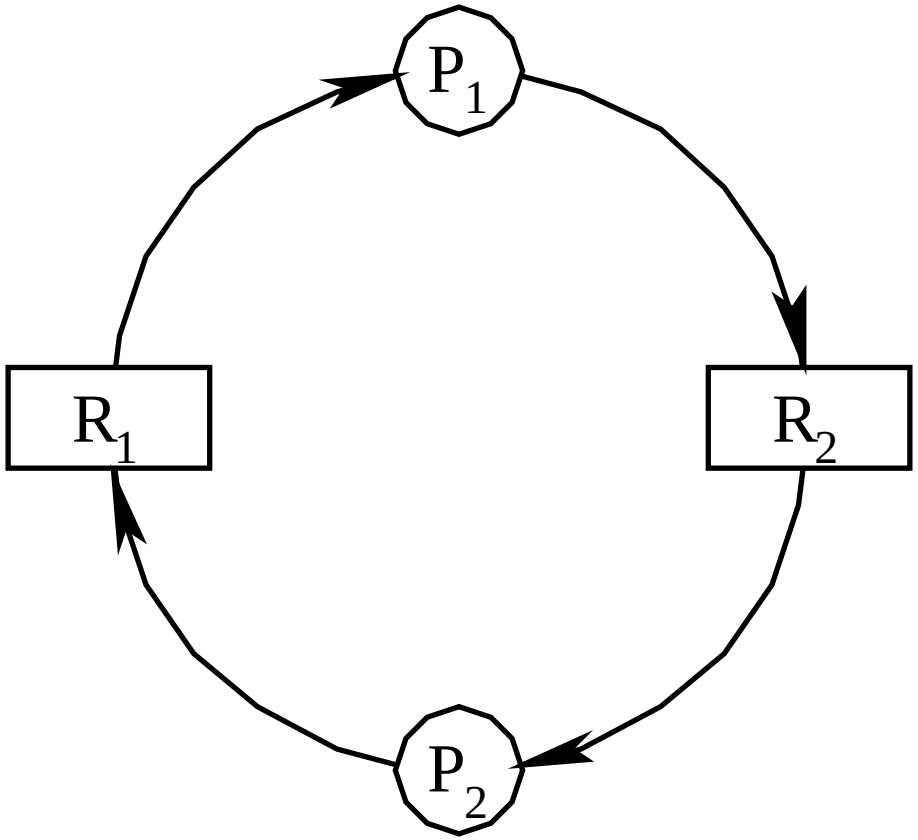


图 3-12 I/O设备共享时的死锁情况

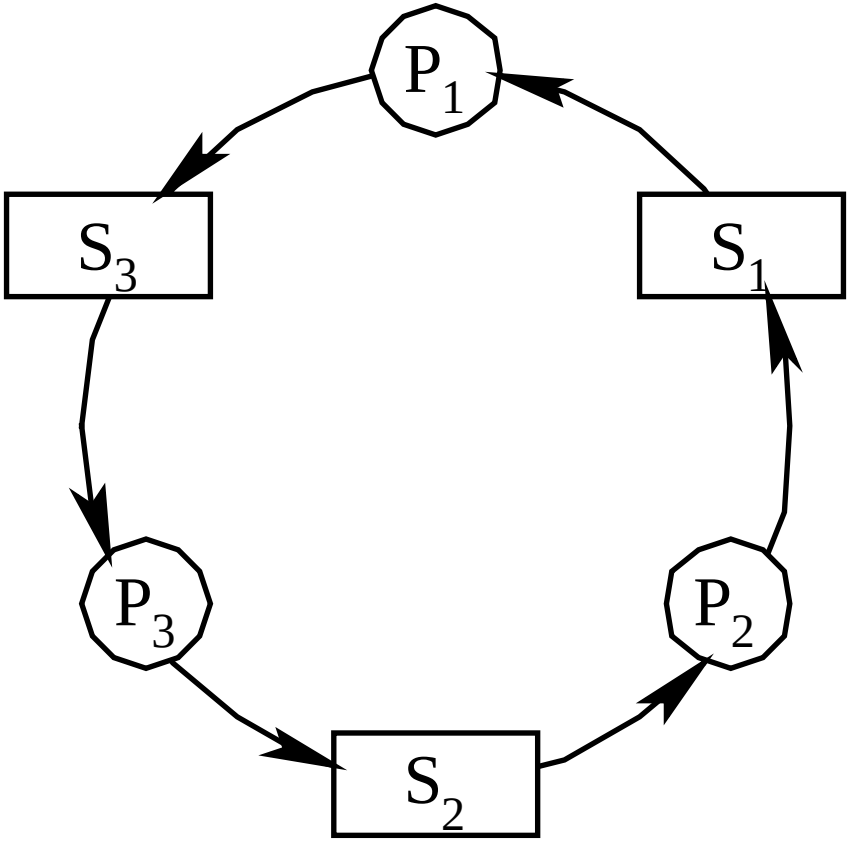


图 3-13 进程之间通信时的死锁

2. 进程推进顺序不当引起死锁

1) 进程推进顺序合法

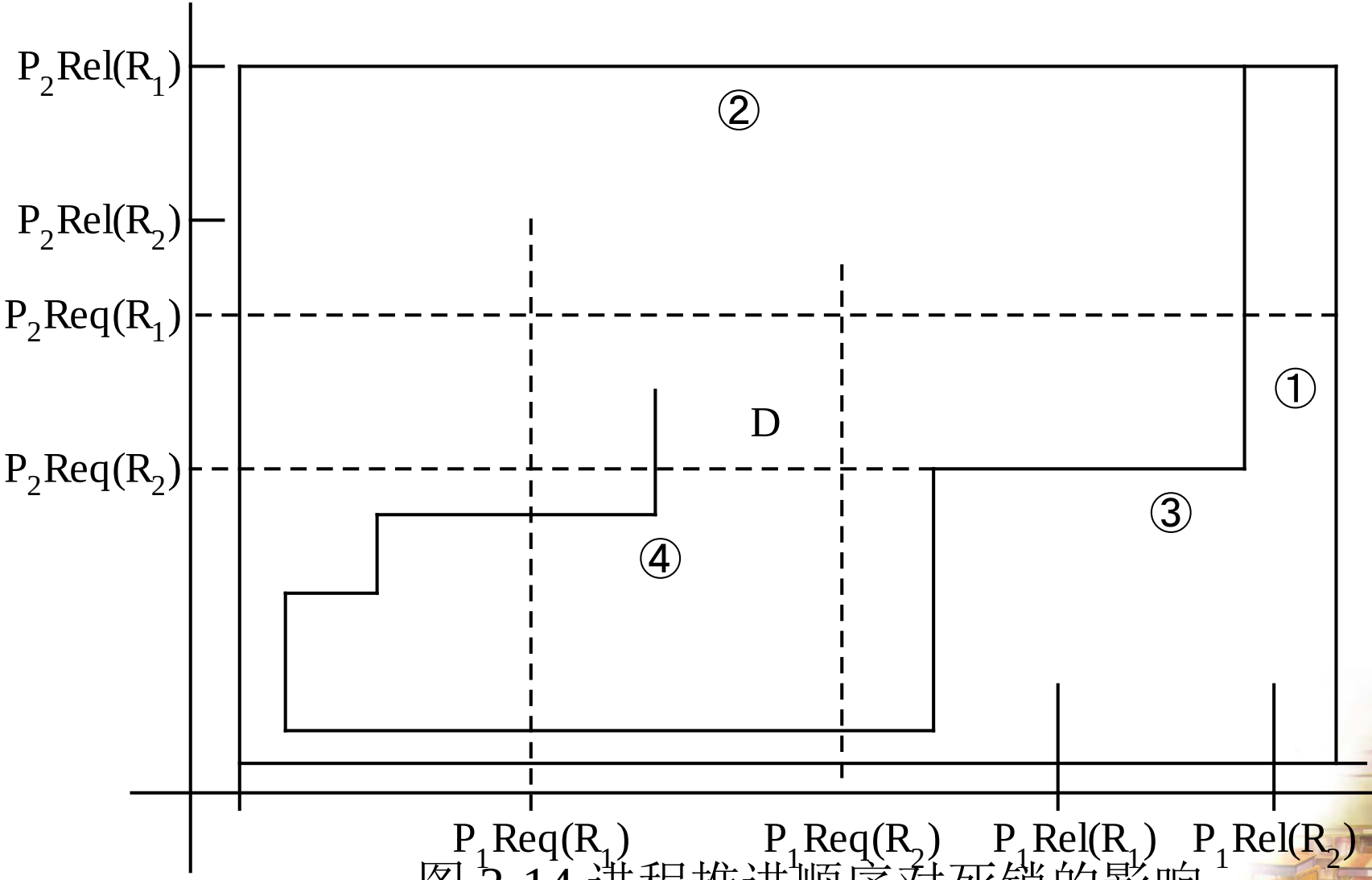


图 3-14 进程推进顺序对死锁的影响

2) 进程推进顺序非法

若并发进程P_1和P_2按曲线④所示的顺序推进，它们将进入不安全区D内。此时P_1保持了资源R_1, P_2保持了资源R_2, 系统处于不安全状态。因为，这时两进程再向前推进，便可能发生死锁。例如，当P_1运行到P_1:Request(R_2)时，将因R_2已被P_2占用而阻塞；当P_2运行到P_2: Request(R_1)时，也将因R_1已被P_1占用而阻塞，于是发生了进程死锁。

3.5.2 产生死锁的必要条件

(1) 互斥条件

(2) 请求和保持条件

(3) 不剥夺条件

(4) 环路等待条件

3.5.3 处理死锁的基本方法

(1) 预防死锁。

(2) 避免死锁。

(3) 检测死锁。

(4) 解除死锁。

3.6 预防死锁的方法

3.6.1 预防死锁

1. 摒弃“请求和保持”条件

2. 摒弃“不剥夺”条件

3. 摒弃“环路等待”条件

3.6.2 系统安全状态

1. 安全状态

在避免死锁的方法中，允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程； 否则，令进程等待。

所谓安全状态，是指系统能按某种进程顺序(P_1，P_2，…，P_n)(称〈P_1, P_2, ..., P_n〉序列为安全序列)，来为每个进程P_i分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺利地完成。如果系统无法找到这样一个安全序列，则称系统处于不安全状态。

2. 安全状态之例

我们通过一个例子来说明安全性。假定系统中有三个进程P_1、 P_2和P_3，共有12台磁带机。进程P_1总共要求10台磁带机，P_2和P_3分别要求4台和9台。假设在T_0时刻，进程P_1、P_2和P_3已分别获得5台、2台和2台磁带机，尚有3台空闲未分配，如下表所示：

进 程	最 大 需 求	已 分 配	可 用
P_1	10	5	3
P_2	4	2	
P_3	9	2	

3. 由安全状态向不安全状态的转换

如果不按照安全序列分配资源，则系统可能会由安全状态进入不安全状态。例如，在T_0时刻以后，P_3又请求1台磁带机，若此时系统把剩余3台中的1台分配给P_3，则系统便进入不安全状态。 因为，此时也无法再找到一个安全序列，例如，把其余的2台分配给P_2，这样，在P_2完成后只能释放出4台，既不能满足P_1尚需5台的要求，也不能满足P_3尚需6台的要求，致使它们都无法推进到完成，彼此都在等待对方释放资源，即陷入僵局，结果导致死锁。

3.6.3 利用银行家算法避免死锁

1. 银行家算法中的数据结构

(1) 可利用资源向量Available。这是一个含有*m*个元素的数组，其中的每一个元素代表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果Available［j］=K，则表示系统中现有R_j类资源*K*个。

(2) 最大需求矩阵Max。这是一个$n\times m$的矩阵，它定义了系统中n个进程中的每一个进程对m类资源的最大需求。如果Max［i,j］=K，则表示进程i需要R_j类资源的最大数目为K 。

(3) 分配矩阵Allocation。这也是一个$n\times m$的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果Allocation［i,j］=K，则表示进程i当前已分得R_j类资源的数目为K。

(4) 需求矩阵Need。这也是一个$n\times m$的矩阵，用以表示每一个进程尚需的各类资源数。如果Need［i,j］=K，则表示进程i还需要R_j类资源K个，方能完成其任务。

Need［i,j］=Max［i,j］-Allocation［i,j］

2. 银行家算法

设$Request_i$是进程P_i的请求向量，如果$Request_i[j]=K$，表示进程P_i需要K个R_j类型的资源。当P_i发出资源请求后，系统按下述步骤进行检查：

(1) 如果$Request_i[j] \le Need[i,j]$，便转向步骤2；否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。

(2) 如果$Request_i[j] \le Available[j]$，便转向步骤(3)；否则，表示尚无足够资源，$P_i$须等待。

(3) 系统试探着把资源分配给进程P_i，并修改下面数据结构中的数值：

Available［j］：=Available［j］-$Request_i$［j］；

Allocation［i,j］：=Allocation［i,j］+$Request_i$［j］；

Need［i,j］：=Need［i,j］-$Request_i$［j］；

(4) 系统执行安全性算法，检查此次资源分配后，系统是否处于安全状态。若安全，才正式将资源分配给进程P_i，以完成本次分配；否则， 将本次的试探分配作废，恢复原来的资源分配状态，让进程P_i等待。

3. 安全性算法

(1) 设置两个向量：① 工作向量Work: 它表示系统可提供给进程继续运行所需的各类资源数目，它含有*m*个元素，在执行安全算法开始时，Work：=Available; ② Finish: 它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做Finish［i］：=false; 当有足够资源分配给进程时，再令Finish［i］：=true。

(2) 从进程集合中找到一个能满足下述条件的进程：

① Finish［i］=false; ② Need［i,j］≤Work［j］； 若找到，执行步骤(3)， 否则，执行步骤(4)。

(3) 当进程P_i获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

Work［j］：= Work［i］+Allocation［i,j］;

Finish［i］：= true;

go to step 2;

(4) 如果所有进程的Finish［i］=true都满足， 则表示系统处于安全状态；否则，系统处于不安全状态。

4. 银行家算法之例

假定系统中有五个进程{P_0, P_1, P_2, P_3, P_4}和三类资源{A, B, C}，各种资源的数量分别为10、5、7，在T_0时刻的资源分配情况如图 3-15 所示。

进程 \ 资源情况	Max A B C	Allocation A B C	Need A B C	Available A B C
P_0	7 5 3	0 1 0	7 4 3	3 3 2 (2 3 0)
P_1	3 2 2	2 0 0 (3 0 2)	1 2 2 (0 2 0)	
P_2	9 0 2	3 0 2	6 0 0	
P_3	2 2 2	2 1 1	0 1 1	
P_4	4 3 3	0 0 2	4 3 1	

图 3-15 T_0时刻的资源分配表

(1) T_0时刻的安全性：

进程 \ 资源情况	Work A	Work B	Work C	Need A	Need B	Need C	Allocation A	Allocation B	Allocation C	Work+Allocation A	Work+Allocation B	Work+Allocation C	Finish
P_1	3	3	2	1	2	2	2	0	0	5	3	2	true
P_3	5	3	2	0	1	1	2	1	1	7	4	3	true
P_4	7	4	3	4	3	1	0	0	2	7	4	5	true
P_2	7	4	5	6	0	0	3	0	2	10	4	7	true
P_0	10	4	7	7	4	3	0	1	0	10	5	7	true

图 3-16 T_0时刻的安全序列

(2) P_1请求资源：P_1发出请求向量$Request_1(1，0，2)$，系统按银行家算法进行检查：

① $Request_1(1, 0, 2) \leq Need_1(1, 2, 2)$

② $Request_1(1, 0, 2) \leq Available_1(3, 3, 2)$

③ 系统先假定可为P_1分配资源，并修改Available，$Allocation_1$和$Need_1$向量，由此形成的资源变化情况如图 3-15 中的圆括号所示。

④ 再利用安全性算法检查此时系统是否安全。

进程 \ 资源情况	Work A B C	Need A B C	Allocation A B C	Work+Allocation A B C	Finish
P_1	2 3 0	0 2 0	3 0 2	5 3 2	true
P_3	5 3 2	0 1 1	2 1 1	7 4 3	true
P_4	7 4 3	4 3 1	0 0 2	7 4 5	true
P_0	7 4 5	7 4 3	0 1 0	7 5 5	true
P_2	7 5 5	6 0 0	3 0 2	10 5 7	true

图 3-17 P_1申请资源时的安全性检查

(3) P_4请求资源：P_4发出请求向量$Request_4(3，3，0)$，系统按银行家算法进行检查：

① $Request_4(3, 3, 0) \leq Need_4(4, 3, 1)$;

② $Request_4(3, 3, 0) < Available(2, 3, 0)$，让$P_4$等待。 (4) P_0请求资源：P_0发出请求向量$Requst_0(0，2，0)$，系统按银行家算法进行检查：

① $Request_0(0, 2, 0) \leq Need\ 0(7, 4, 3)$;

② $Request_0(0, 2, 0) \leq Available(2, 3, 0)$;

③ 系统暂时先假定可为P_0分配资源，并修改有关数据，如图 3-18 所示。

进程 \ 资源情况	Allocation A B C	Need A B C	Available A B C
P_0	0 3 0	7 2 3	2 1 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

图 3-18 为P_0分配资源后的有关资源数据

3.7 死锁的检测与解除

3.7.1 死锁的检测

1. 资源分配图(Resource Allocation Graph)

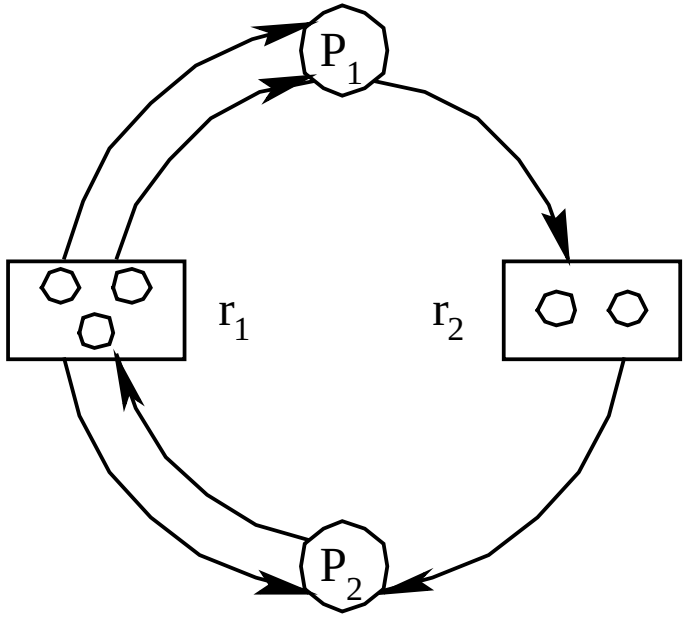


图 3-19 每类资源有多个时的情况

(2) 凡属于E中的一个边e∈E，都连接着P中的一个结点和R中的一个结点，e={p_i，r_j}是资源请求边，由进程p_i指向资源r_j， 它表示进程p_i请求一个单位的r_j资源。e={r_j，p_i}是资源分配边，由资源r_j指向进程p_i，它表示把一个单位的资源r_j分配给进程p_i。

2. 死锁定理

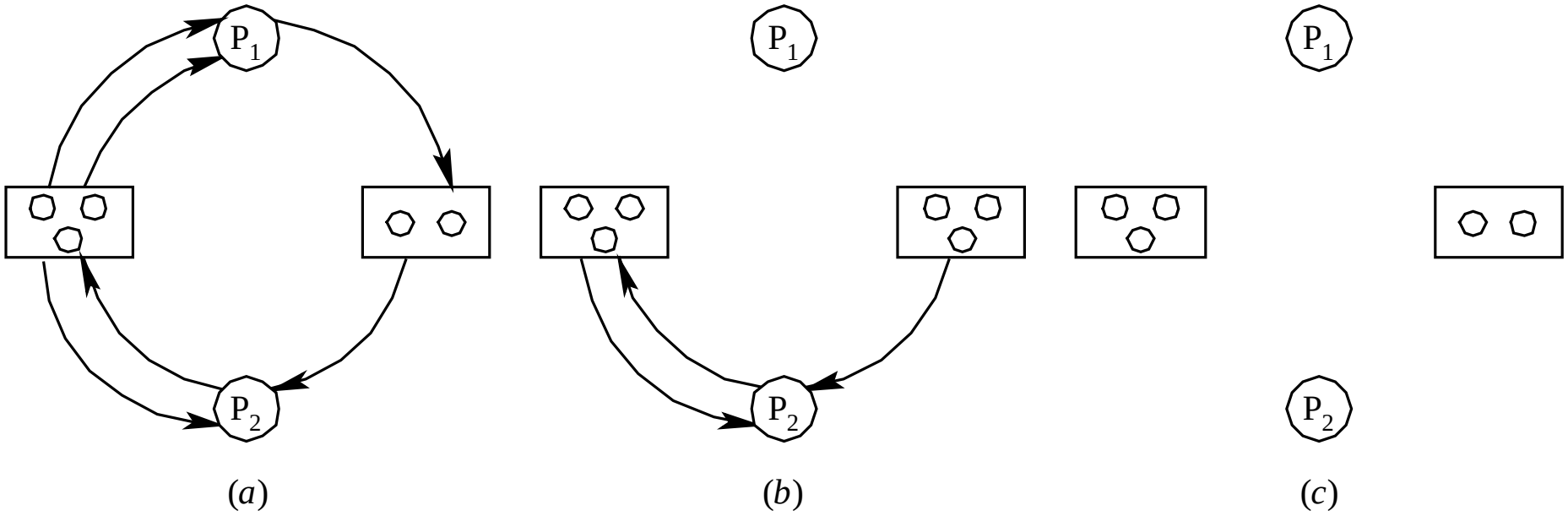


图 3-20 资源分配图的简化

3. 死锁检测中的数据结构

(1) 可利用资源向量Available，它表示了*m*类资源中每一类资源的可用数目。

(2) 把不占用资源的进程(向量Allocation：=0)记入L表中，即$L_i \cup L$。

(3) 从进程集合中找到一个$Request_i \leq Work$的进程，做如下处理：① 将其资源分配图简化，释放出资源，增加工作向量Work：=Work+$Allocation_i$。② 将它记入L表中。

(4) 若不能把所有进程都记入L表中， 便表明系统状态S的资源分配图是不可完全简化的。 因此，该系统状态将发生死锁。

Work ：=Available;

L ：= $\{L_i|Allocation_i=0 \cap Request_i=0\}$

for all $L_i \notin L$ do

begin

for all $Request_i \leq Work$ do

begin

Work ：=Work+$Allocation_i$;

$L_i \cup L$;

end

end

deadlock： = ($L=\{p_1, p_2, \ldots, p_n\}$);

3.7.2 死锁的解除

(1) 剥夺资源。

(2) 撤消进程。

为把系统从死锁状态中解脱出来，所花费的代价可表示为：

$$R(S)_{min}=min\{C_{ui}\}+min\{C_{uj}\}+min\{C_{uk}\}+\cdots$$

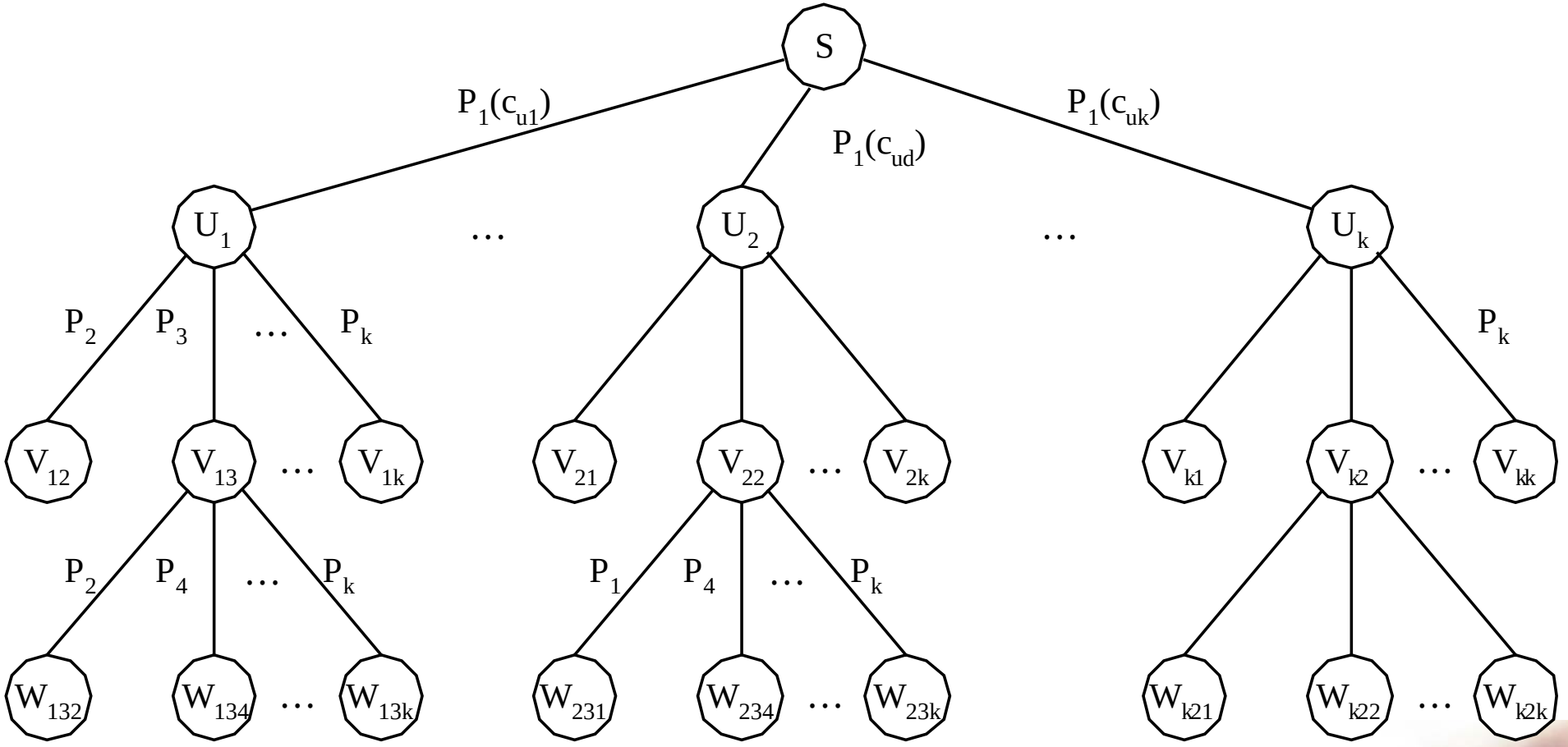


图 3-21 付出代价最小的死锁解除方法

第四章 存储器管理

4.1 程序的装入和链接

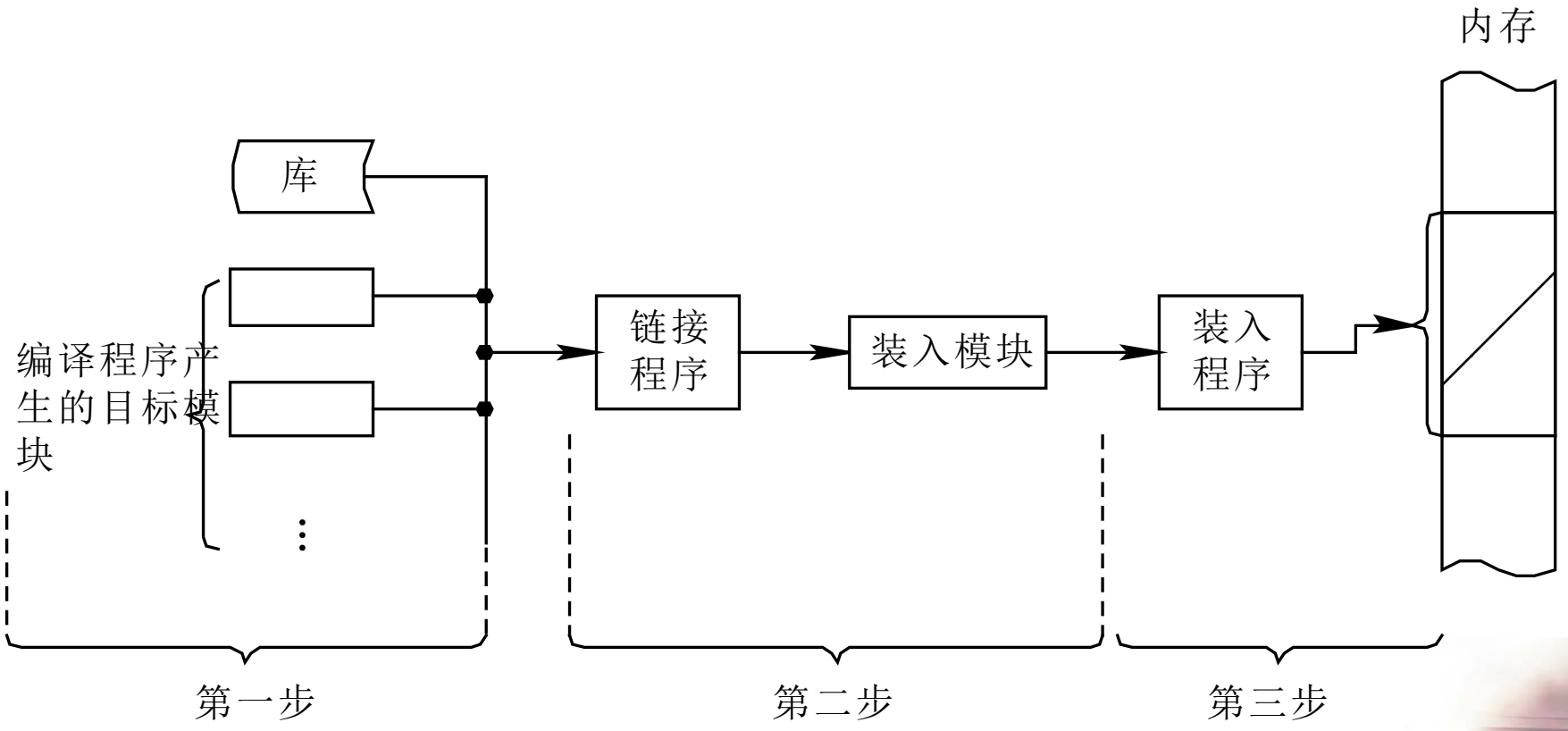


图 4-1 对用户程序的处理步骤

4.1.1 程序的装入

1. 绝对装入方式(Absolute Loading Mode)

程序中所使用的绝对地址，既可在编译或汇编时给出，也可由程序员直接赋予。 但在由程序员直接给出绝对地址时， 不仅要求程序员熟悉内存的使用情况，而且一旦程序或数据被修改后，可能要改变程序中的所有地址。因此，通常是宁可在程序中采用符号地址，然后在编译或汇编时，再将这些符号地址转换为绝对地址。

2. 可重定位装入方式(Relocation Loading Mode)

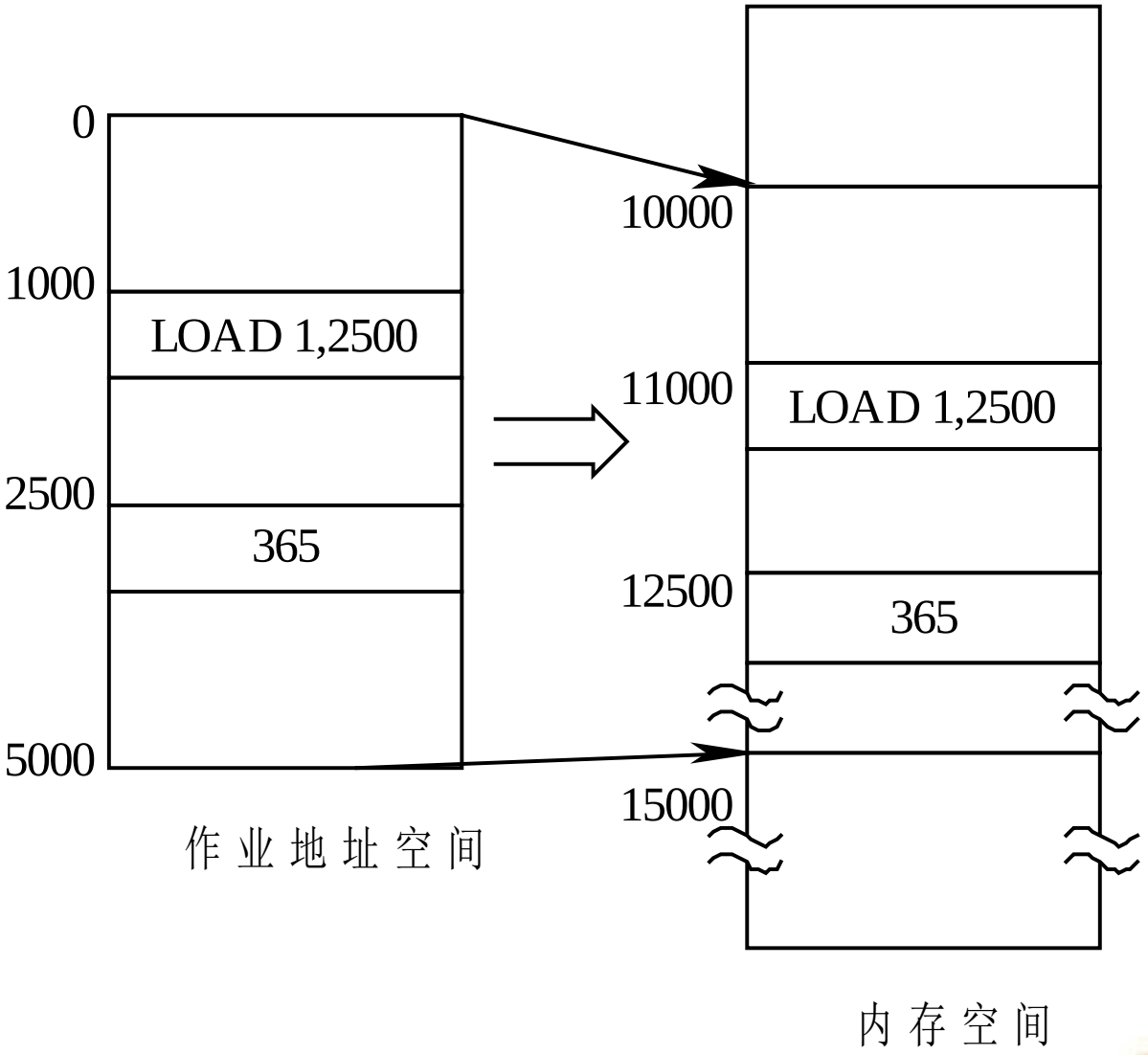


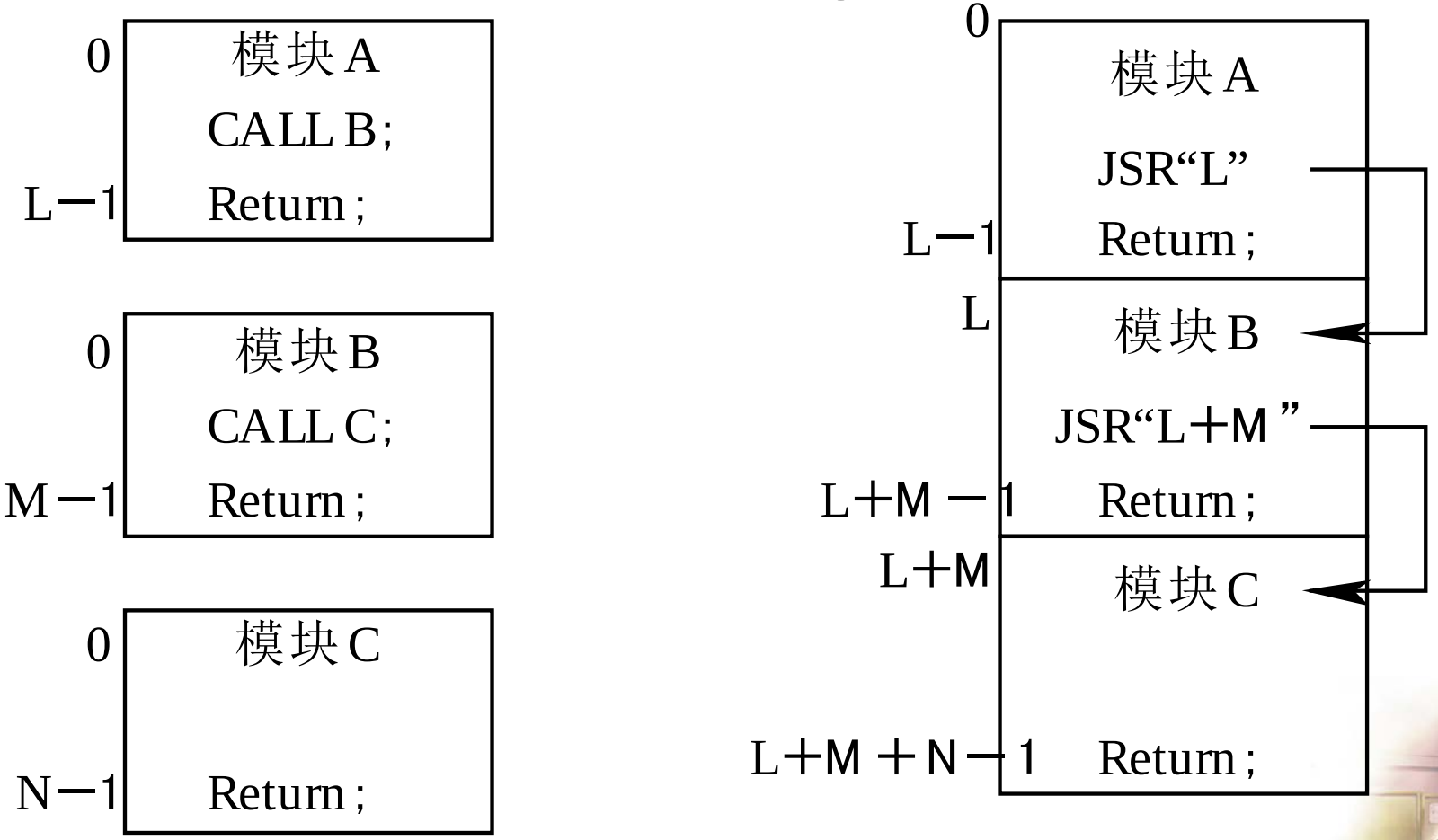
图 4-2 作业装入内存时的情况

3. 动态运行时装入方式(Denamle Run-time Loading)

动态运行时的装入程序，在把装入模块装入内存后，并不立即把装入模块中的相对地址转换为绝对地址，而是把这种地址转换推迟到程序真正要执行时才进行。因此， 装入内存后的所有地址都仍是相对地址。

4.1.2 程序的链接

1. 静态链接方式(Static Linking)



(*a*) 目标模块 图 4-3 程序链接示意图 (*b*) 装入模块

在将这几个目标模块装配成一个装入模块时，须解决以下两个问题：

(1) 对相对地址进行修改。

(2) 变换外部调用符号。

2. 装入时动态链接(Load time Dynamic Linking)

装入时动态链接方式有以下优点：

(1) 便于修改和更新。

(2) 便于实现对目标模块的共享。

3. 运行时动态链接(Run-time Dynamic Linking)

近几年流行起来的运行时动态链接方式，是对上述在装入时链接方式的一种改进。这种链接方式是将对某些模块的链接推迟到执行时才执行，亦即，在执行过程中，当发现一个被调用模块尚未装入内存时，立即由OS去找到该模块并将之装入内存， 把它链接到调用者模块上。凡在执行过程中未被用到的目标模块，都不会被调入内存和被链接到装入模块上，这样不仅可加快程序的装入过程，而且可节省大量的内存空间。

4.2 连续分配方式

4.2.1 单一连续分配

这是最简单的一种存储管理方式，但只能用于单用户、单任务的操作系统中。采用这种存储管理方式时，可把内存分为系统区和用户区两部分，系统区仅提供给OS使用，通常是放在内存的低址部分；用户区是指除系统区以外的全部内存空间， 提供给用户使用。

4.2.2 固定分区分配

1. 划分分区的方法

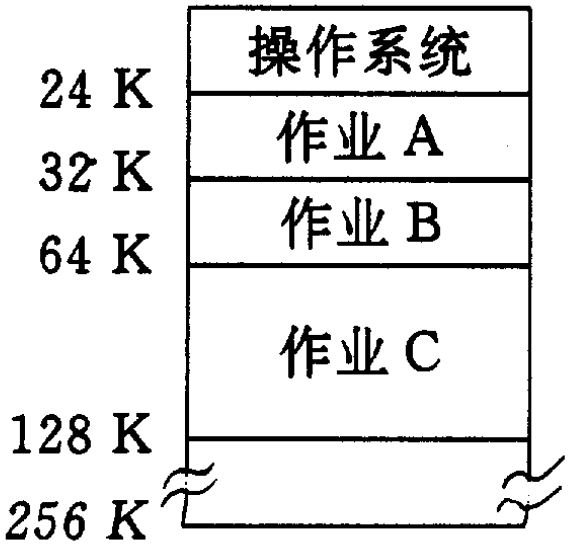
(1) 分区大小相等， 即使所有的内存分区大小相等。

(2) 分区大小不等。

2. 内存分配

分区号	大小(K)	起址(K)	状态
1	12	20	已分配
2	32	32	已分配
3	64	64	已分配
4	128	128	已分配

(a) 分区说明表



(b) 存储空间分配情况

图 4-4 固定分区使用表

4.2.3 动态分区分配

1. 分区分配中的数据结构

(1) 空闲分区表。

(2) 空闲分区链。

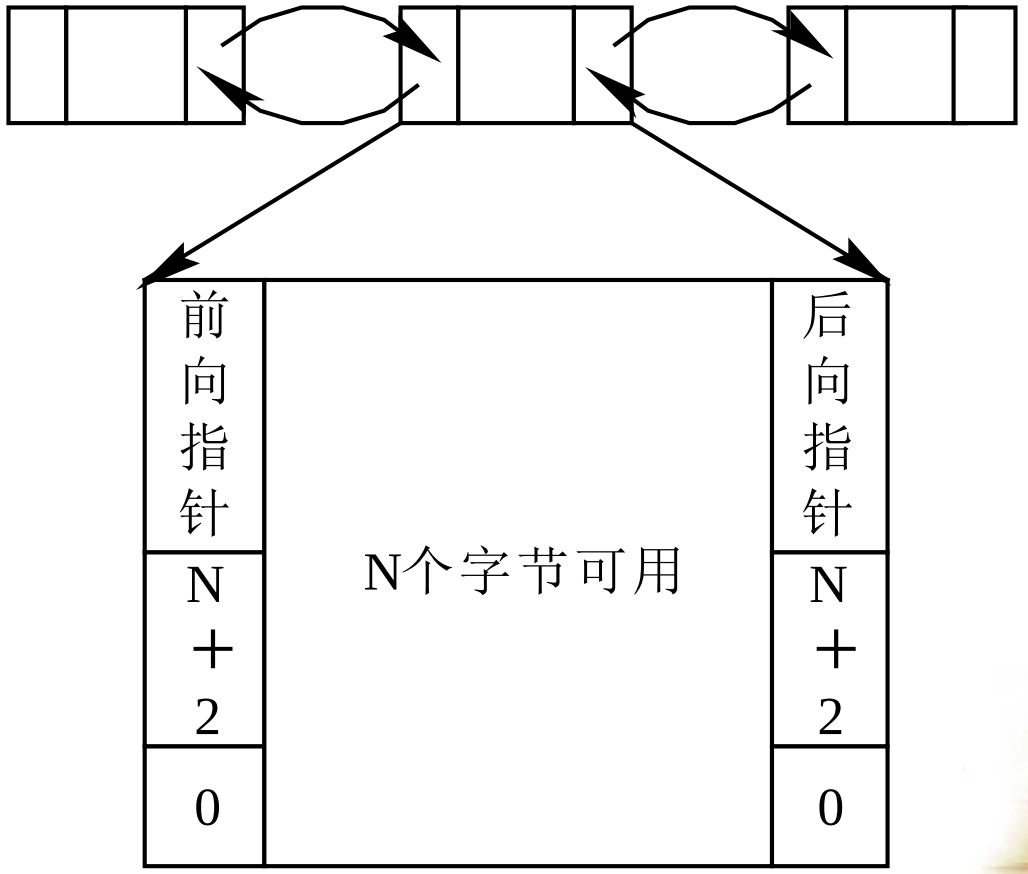


图 4-5 空闲链结构

2. 分区分配算法

(1) 首次适应算法FF。

(2) 循环首次适应算法，该算法是由首次适应算法演变而成的。

(3) 最佳适应算法。

3. 分区分配操作

1) 分配内存

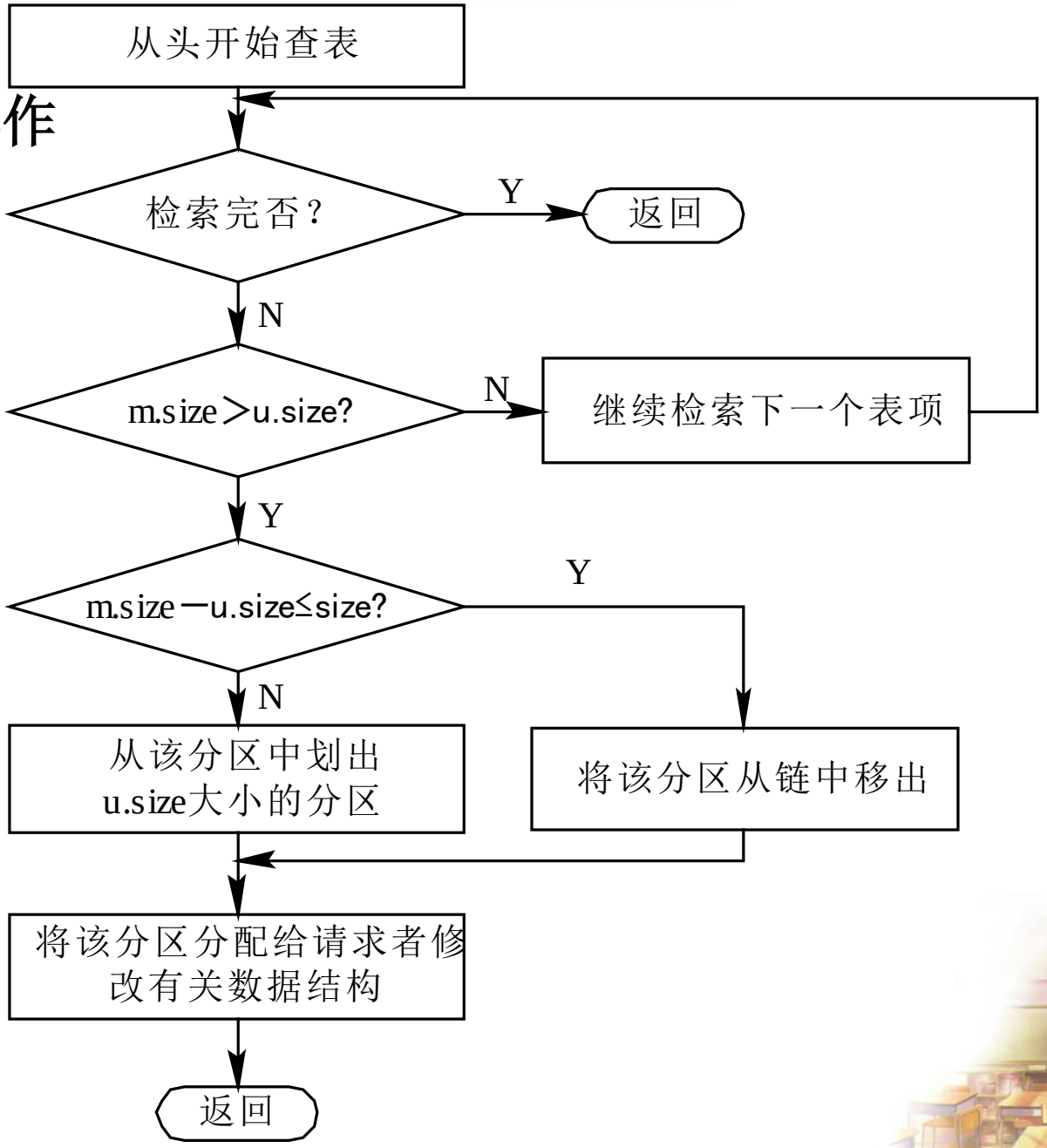


图4-6 内存分配流程

2) 回收内存

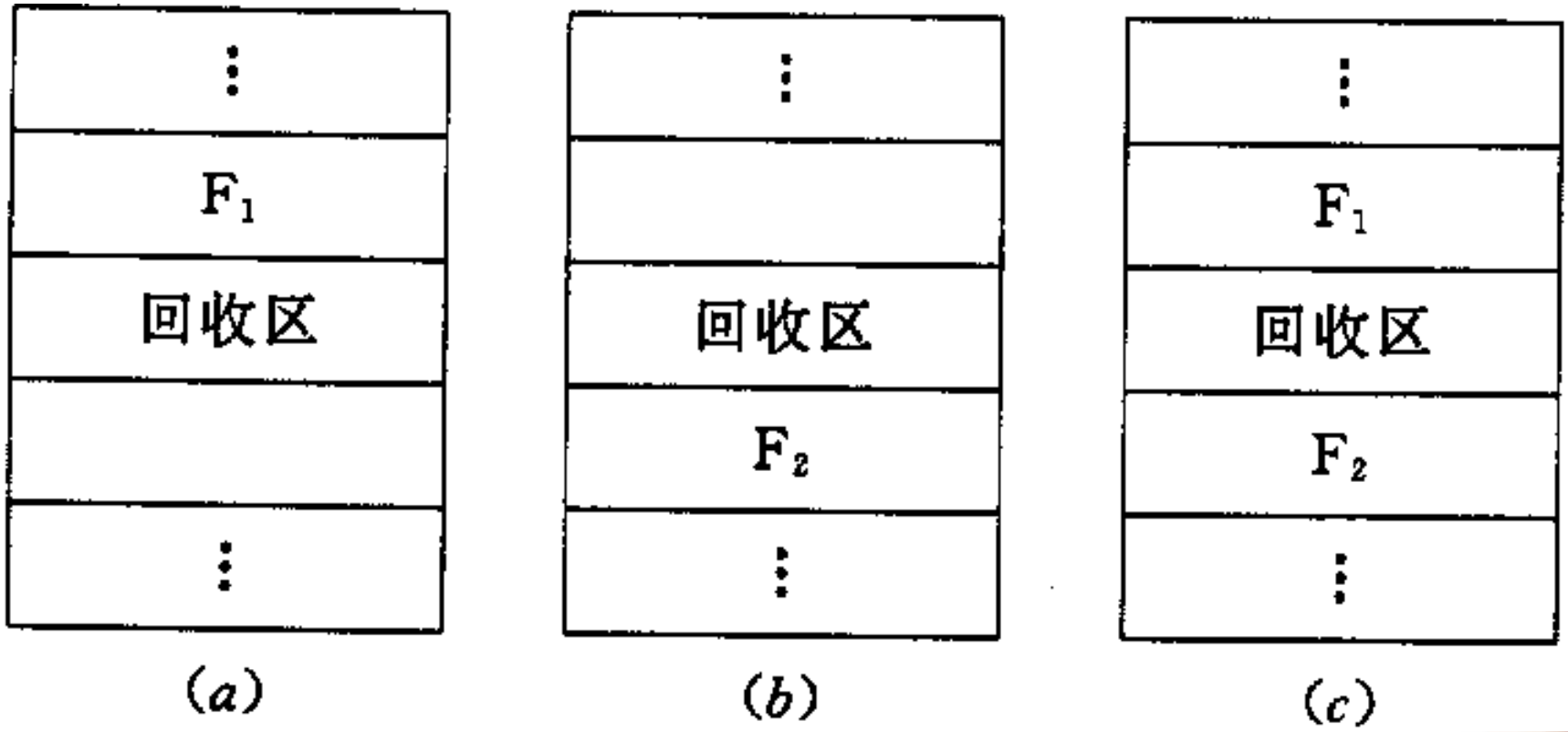
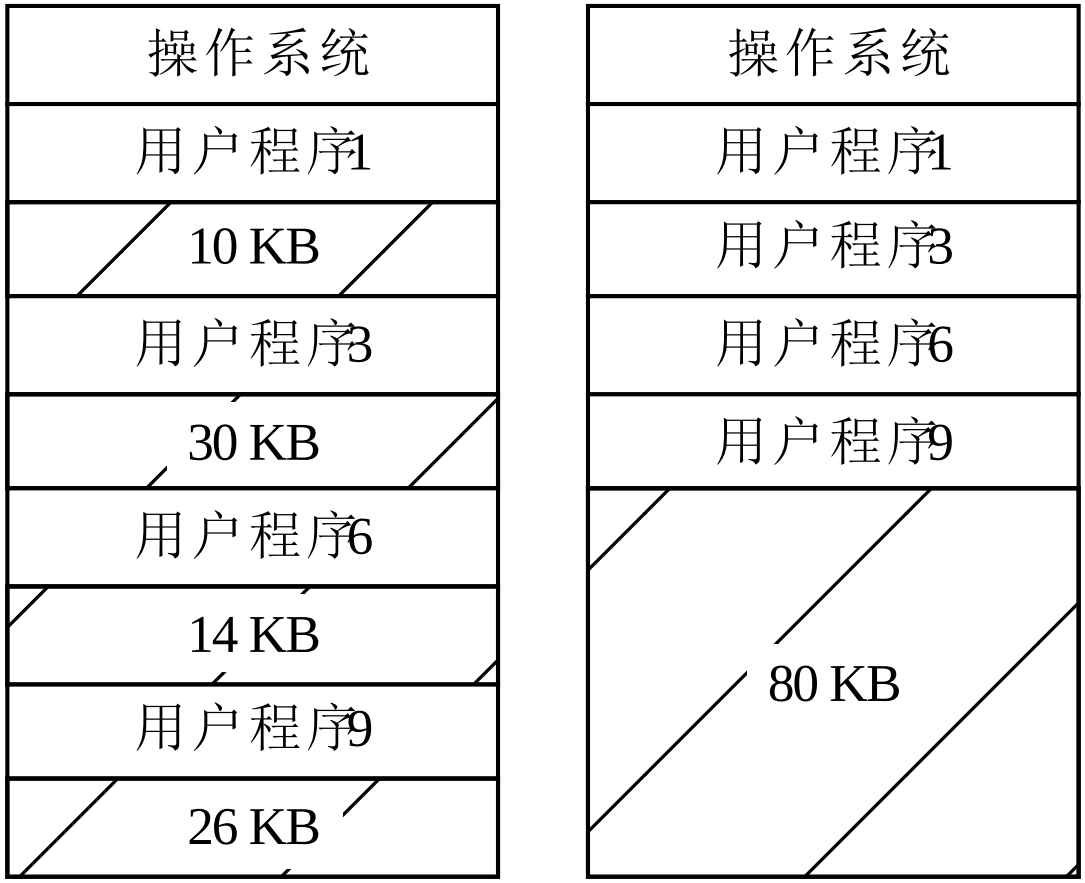


图 4-7 内存回收时的情况

4.2.4 可重定位分区分配

1. 动态重定位的引入



(*a*) 紧凑前　　(*b*) 紧凑后

图 4-8 紧凑的示意

2. 动态重定位的实现

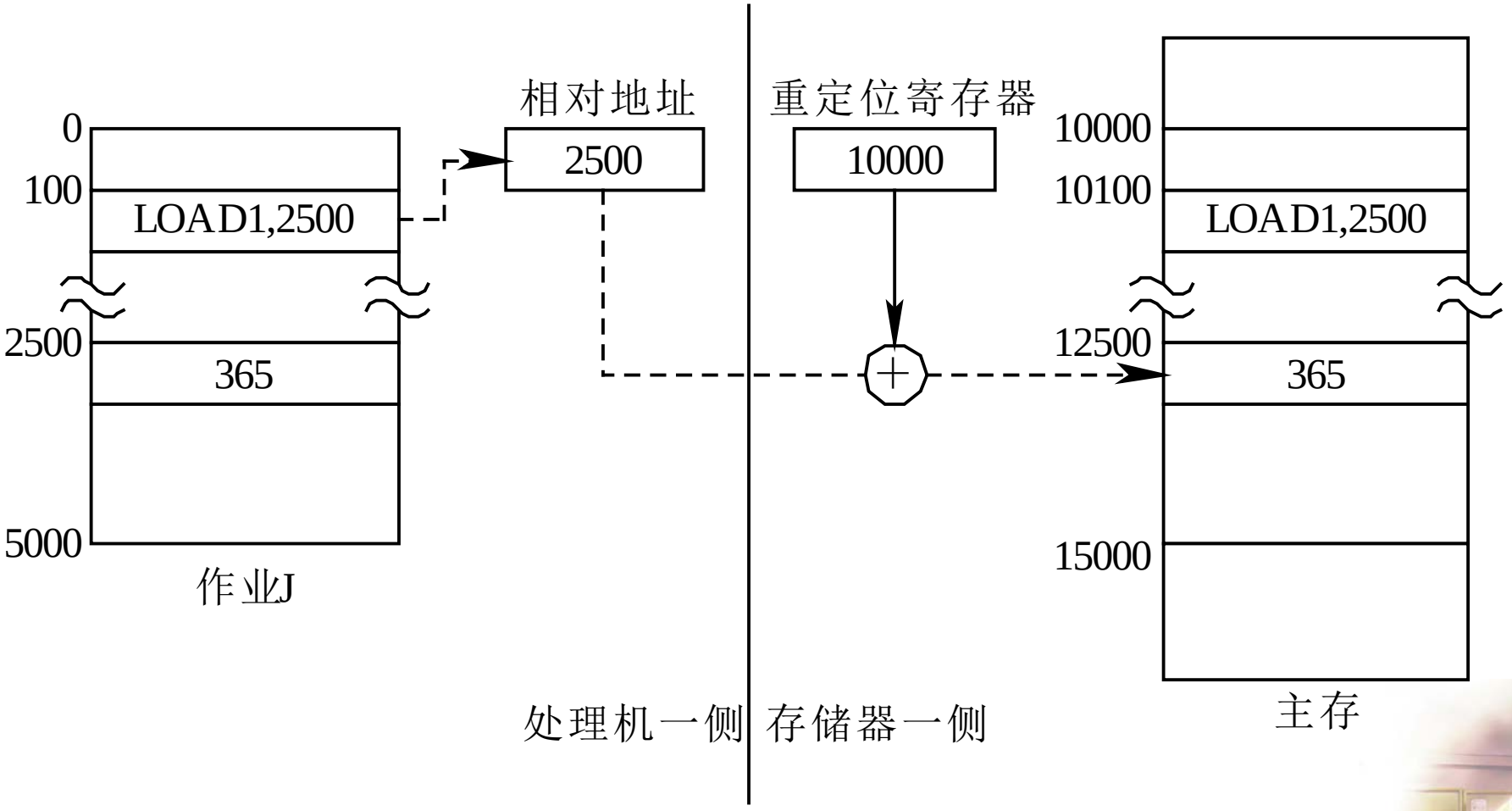


图 4-9 动态重定位示意图

3. 动态重定位分区分配算法

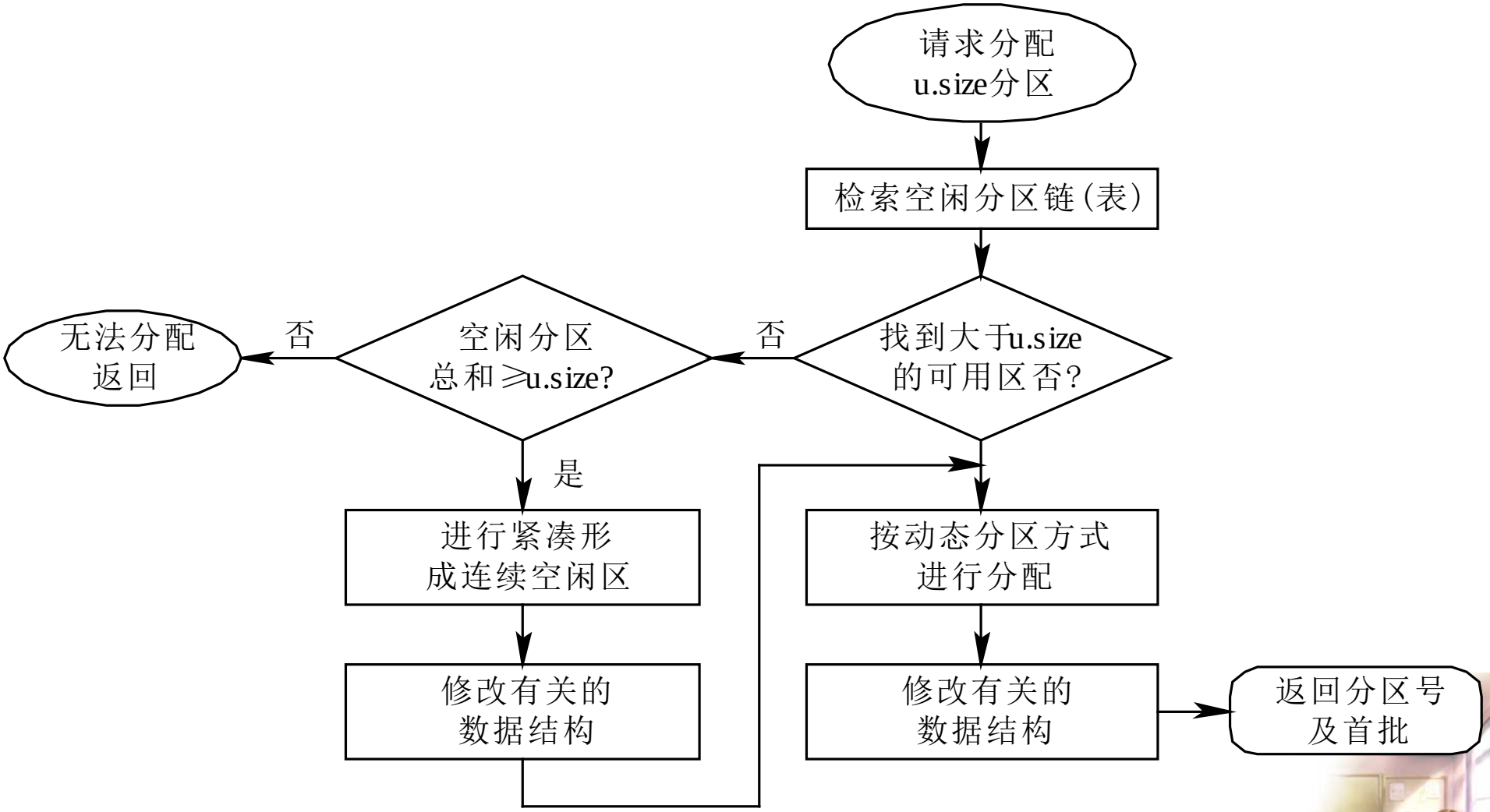


图 4-10 动态分区分配算法流程图

4.2.5 对换(Swapping)

1. 对换的引入

所谓“对换”， 是指把内存中暂时不能运行的进程或者暂时不用的程序和数据，调出到外存上，以便腾出足够的内存空间，再把已具备运行条件的进程或进程所需要的程序和数据，调入内存。对换是提高内存利用率的有效措施。

2. 对换空间的管理

为了能对对换区中的空闲盘块进行管理，在系统中应配置相应的数据结构，以记录外存的使用情况。其形式与内存在动态分区分配方式中所用数据结构相似，即同样可以用空闲分区表或空闲分区链。在空闲分区表中的每个表目中应包含两项， 即对换区的首址及其大小，它们的单位是盘块号和盘块数。

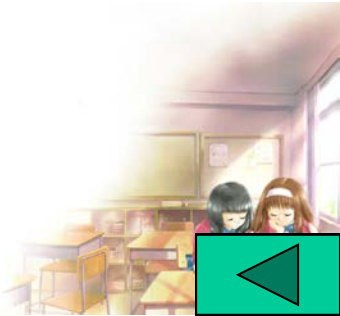
3. 进程的换出与换入

(1) 进程的换出。

每当一进程由于创建子进程而需要更多的内存空间，但又无足够的内存空间等情况发生时，系统应将某进程换出。其过程是：系统首先选择处于阻塞状态且优先级最低的进程作为换出进程，然后启动盘块，将该进程的程序和数据传送到磁盘的对换区上。若传送过程未出现错误，便可回收该进程所占用的内存空间，并对该进程的进程控制块做相应的修改。

(2) 进程的换入。

系统应定时地查看所有进程的状态，从中找出“就绪”状态但已换出的进程，将其中换出时间(换出到磁盘上)最久的进程作为换入进程，将之换入，直至已无可换入的进程或无可换出的进程为止。



4.3 基本分页存储管理方式

4.3.1 页面与页表

1. 页面

1) 页面和物理块

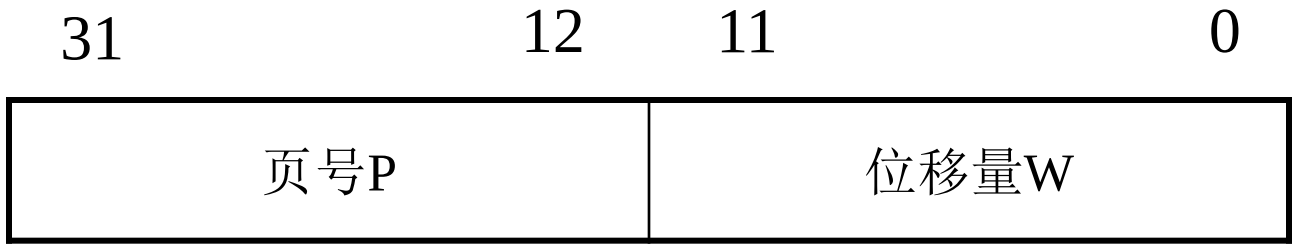
分页存储管理，是将一个进程的逻辑地址空间分成若干个大小相等的片，称为页面或页，并为各页加以编号，从0开始，如第0页、第1页等。相应地，也把内存空间分成与页面相同大小的若干个存储块，称为(物理)块或页框(frame)， 也同样为它们加以编号，如0#块、1#块等等。在为进程分配内存时，以块为单位将进程中的若干个页分别装入到多个可以不相邻接的物理块中。由于进程的最后一页经常装不满一块而形成了不可利用的碎片，称之为“页内碎片”。

2) 页面大小

在分页系统中的页面其大小应适中。页面若太小，一方面虽然可使内存碎片减小，从而减少了内存碎片的总空间，有利于提高内存利用率，但另一方面也会使每个进程占用较多的页面，从而导致进程的页表过长，占用大量内存； 此外，还会降低页面换进换出的效率。然而，如果选择的页面较大，虽然可以减少页表的长度，提高页面换进换出的速度，但却又会使页内碎片增大。因此，页面的大小应选择得适中，且页面大小应是2的幂，通常为512 B~8 KB。

2. 地址结构

分页地址中的地址结构如下：



对某特定机器，其地址结构是一定的。若给定一个逻辑地址空间中的地址为A，页面的大小为L，则页号P和页内地址d可按下式求得：

$$P = INT\left[\frac{A}{L}\right]$$

$$d = [A]MODL$$

3. 页表

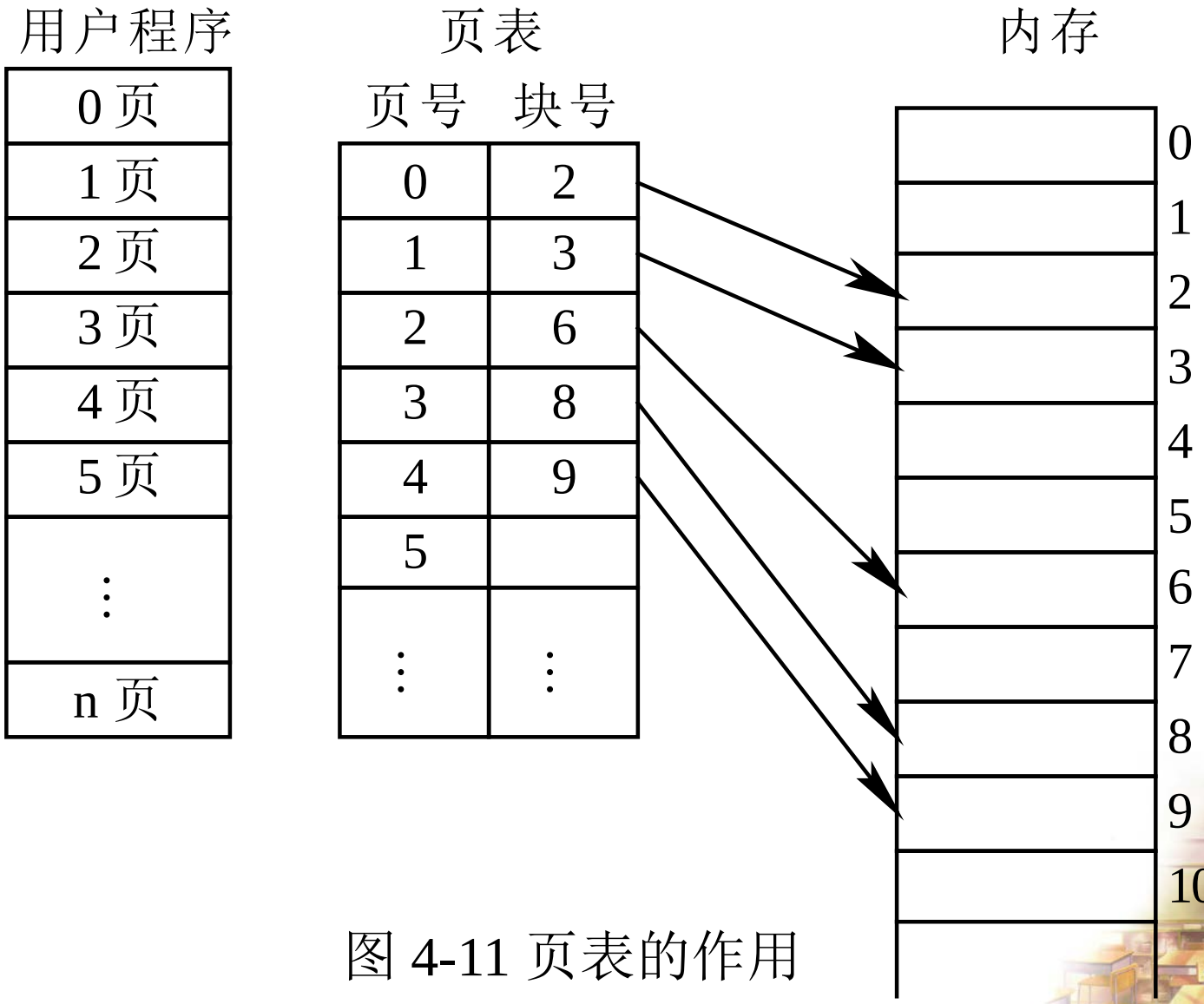


图 4-11 页表的作用

4.3.2 地址变换机构

1. 基本的地址变换机构

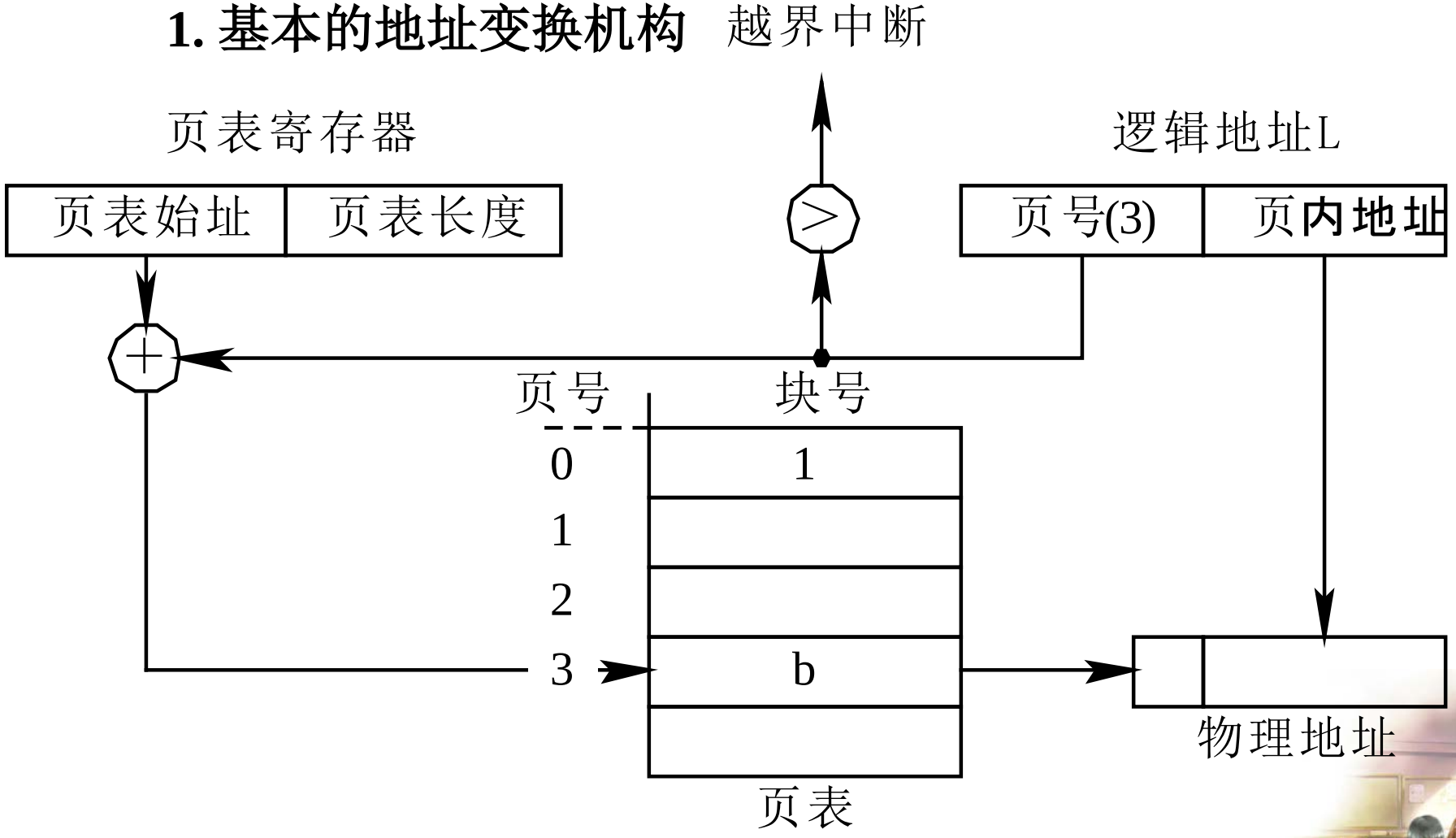


图 4-12 分页系统的地址变换机构

2. 具有快表的地址变换机构

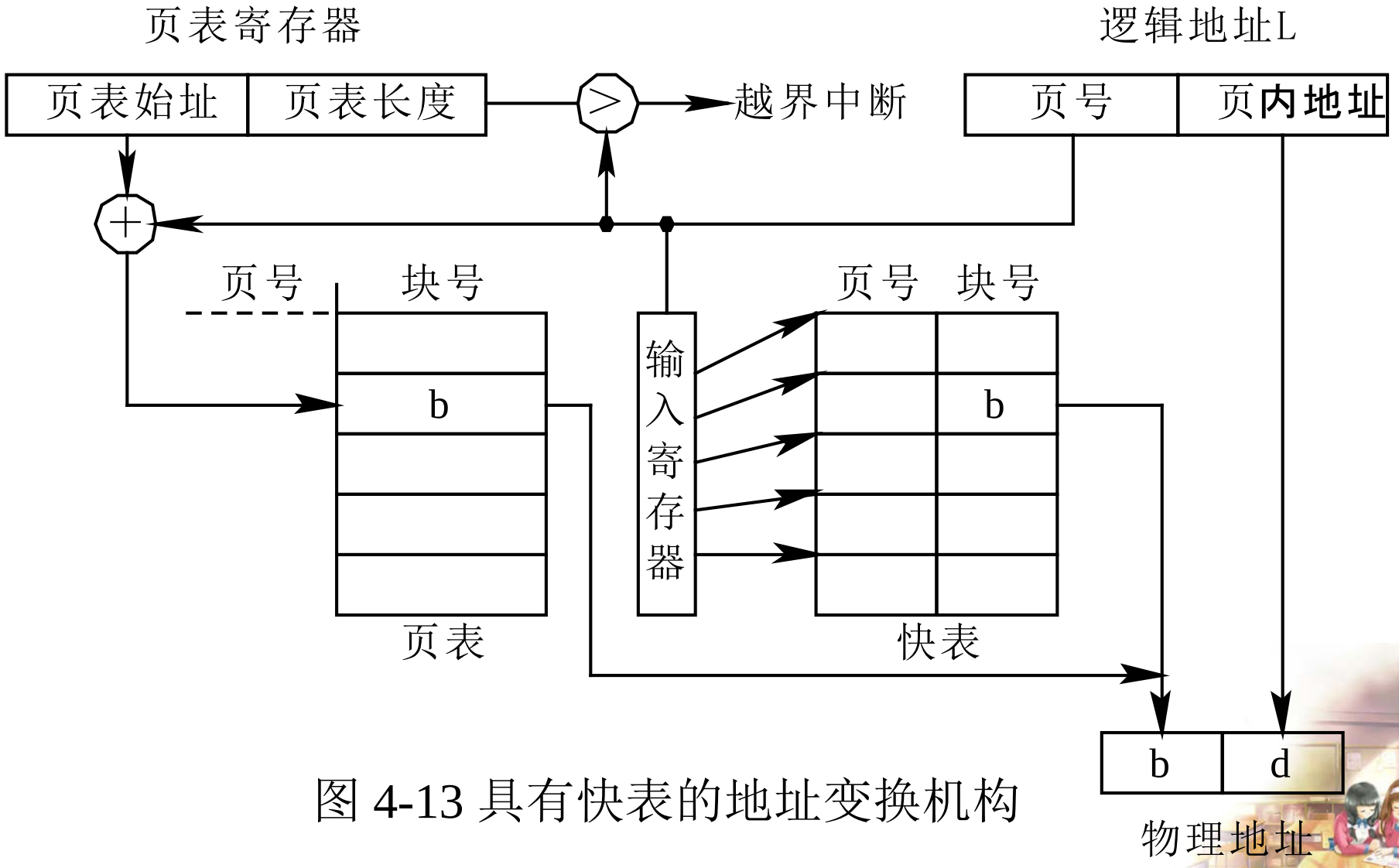


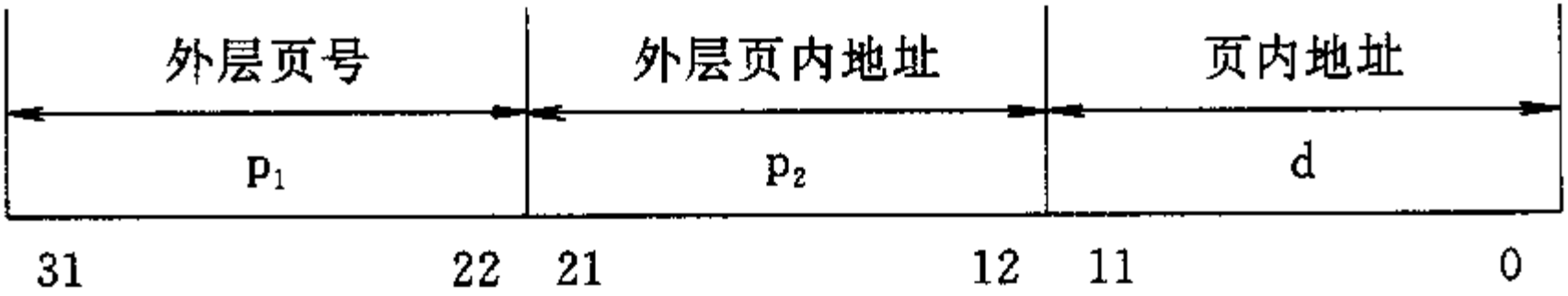
图 4-13 具有快表的地址变换机构

4.3.3 两级和多级页表

现代的大多数计算机系统，都支持非常大的逻辑地址空间(2^{32}~2^{64})。在这样的环境下，页表就变得非常大，要占用相当大的内存空间。例如，对于一个具有32位逻辑地址空间的分页系统，规定页面大小为4 KB即2^{12} B，则在每个进程页表中的页表项可达1兆个之多。又因为每个页表项占用一个字节，故每个进程仅仅其页表就要占用4 KB的内存空间，而且还要求是连续的。 可以采用这样两个方法来解决这一问题：① 采用离散分配方式来解决难以找到一块连续的大内存空间的问题：② 只将当前需要的部分页表项调入内存， 其余的页表项仍驻留在磁盘上，需要时再调入。

1. 两级页表(Two-Level Page Table)

逻辑地址结构可描述如下：



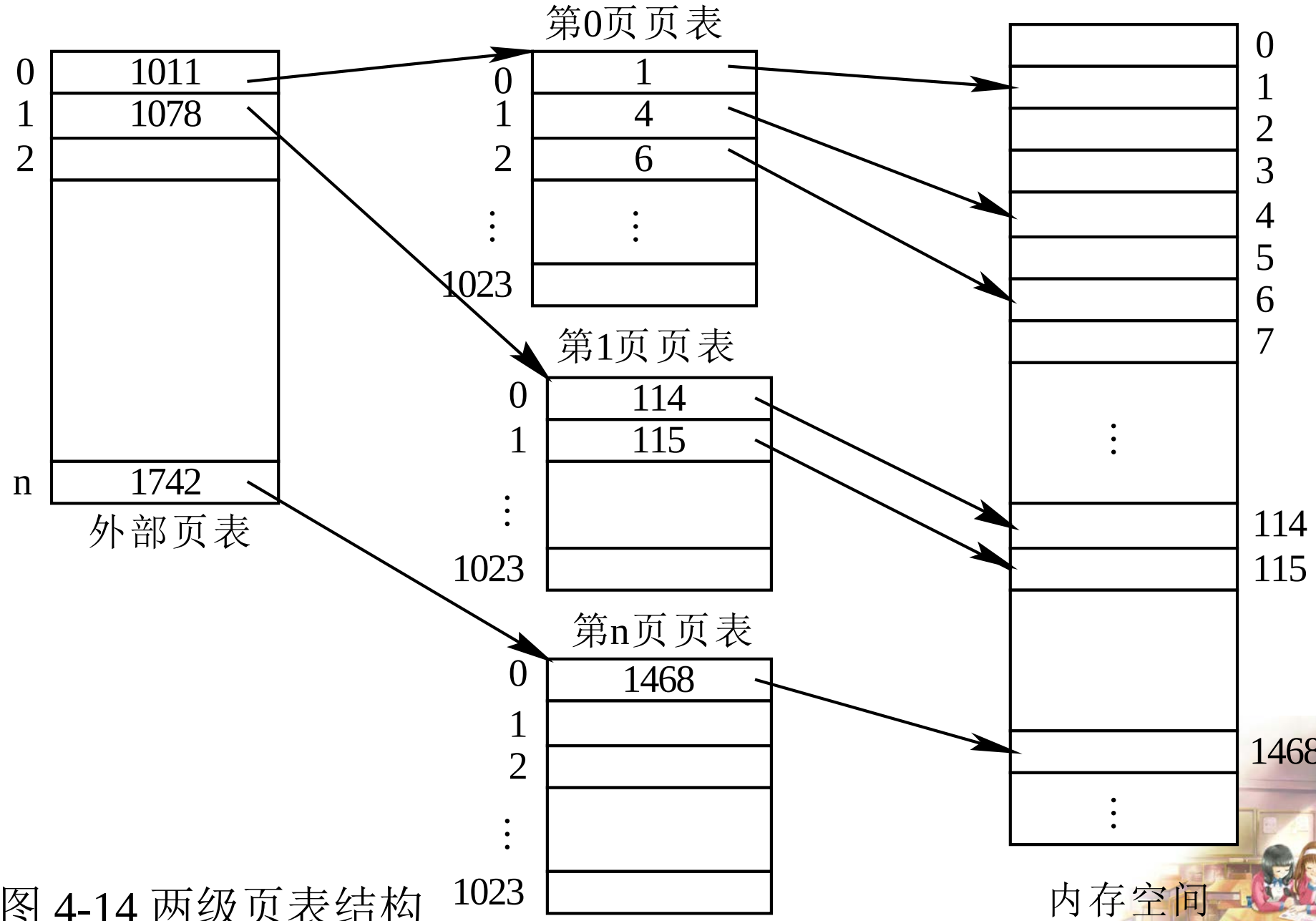

第0页页表
第1页页表
第n页页表
外部页表
内存空间
0
1
2
n
1011
1078
1742
1
4
6
1023
114
115
1468
0
1
2
3
4
5
6
7
114
115
1468

图 4-14 两级页表结构

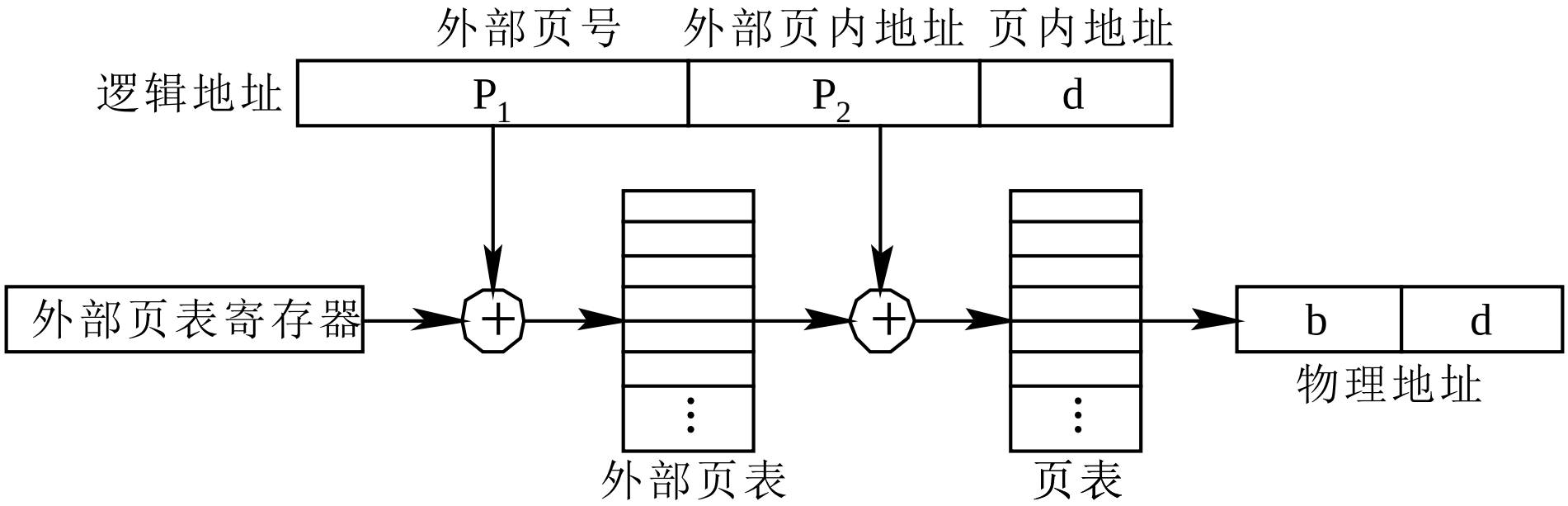


图 4-15 具有两级页表的地址变换机构

2. 多级页表

对于32位的机器，采用两级页表结构是合适的；但对于64位的机器，如果页面大小仍采用4 KB即2^{12} B，那么还剩下52位， 假定仍按物理块的大小(2^{12}位)来划分页表，则将余下的42位用于外层页号。此时在外层页表中可能有4096 G个页表项， 要占用16384 GB的连续内存空间。 必须采用多级页表，将外层页表再进行分页，也是将各分页离散地装入到不相邻接的物理块中，再利用第2级的外层页表来映射它们之间的关系。

对于64位的计算机，如果要求它能支持2^{64}(=1844744 TB)规模的物理存储空间，则即使是采用三级页表结构也是难以办到的；而在当前的实际应用中也无此必要。

4.4 基本分段存储管理方式

4.4.1 分段存储管理方式的引入

引入分段存储管理方式， 主要是为了满足用户和程序员的下述一系列需要：

1) 方便编程

2) 信息共享

3) 信息保护

4) 动态增长

5) 动态链接

4.4.2 分段系统的基本原理

1. 分段

分段地址中的地址具有如下结构：

段号	段内地址

31 16 15 0

2. 段表

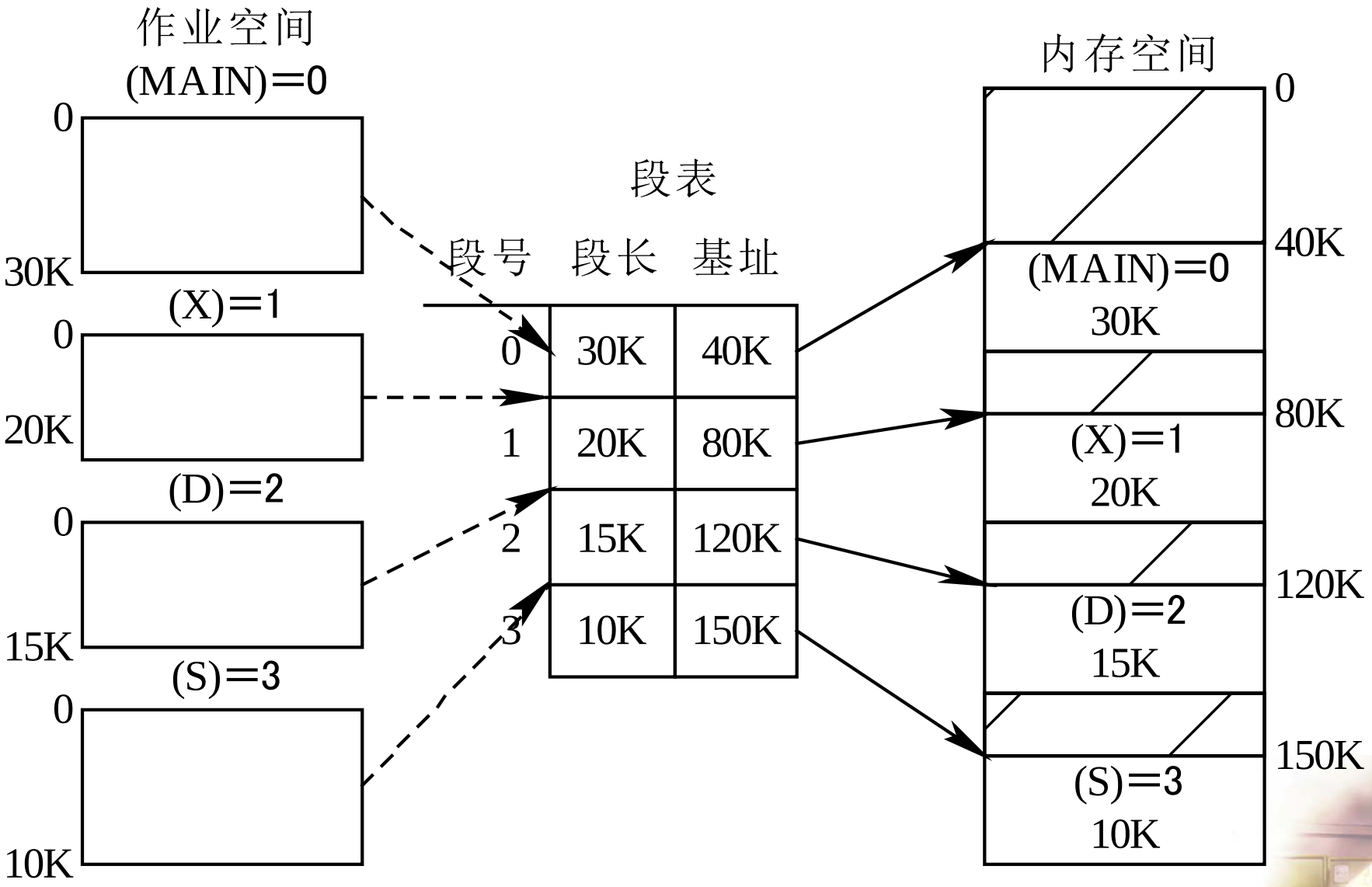


图 4-16 利用段表实现地址映射

3. 地址变换机构

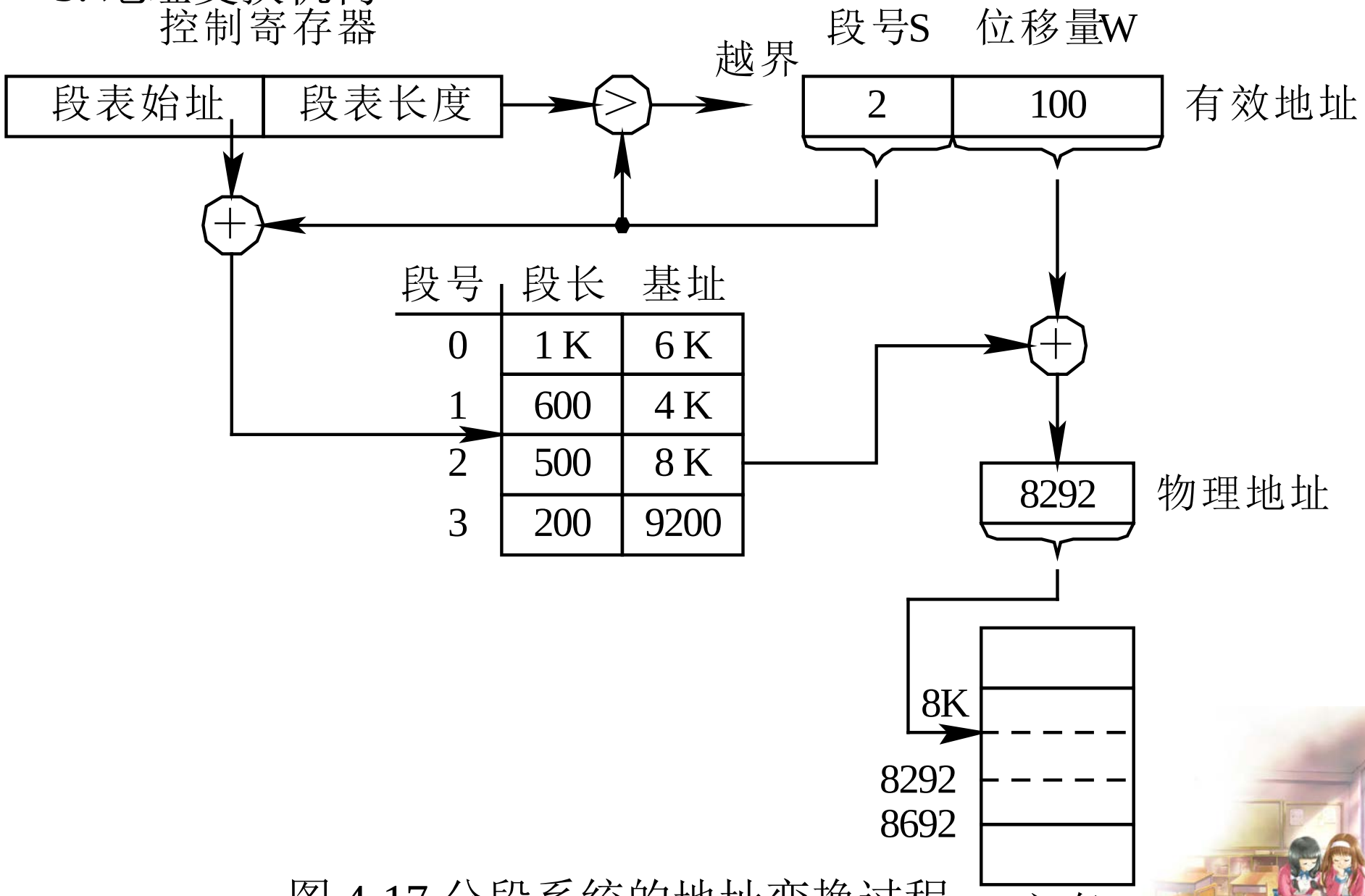


图 4-17 分段系统的地址变换过程

4. 分页和分段的主要区别

(1) 页是信息的物理单位，分页是为实现离散分配方式，以消减内存的外零头， 提高内存的利用率。或者说， 分页仅仅是由于系统管理的需要而不是用户的需要。段则是信息的逻辑单位，它含有一组其意义相对完整的信息。分段的目的是为了能更好地满足用户的需要。

(2) 页的大小固定且由系统决定，由系统把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而在系统中只能有一种大小的页面；而段的长度却不固定， 决定于用户所编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。

(3) 分页的作业地址空间是一维的，即单一的线性地址空间，程序员只需利用一个记忆符，即可表示一个地址；而分段的作业地址空间则是二维的，程序员在标识一个地址时，既需给出段名， 又需给出段内地址。

4.4.3 信息共享

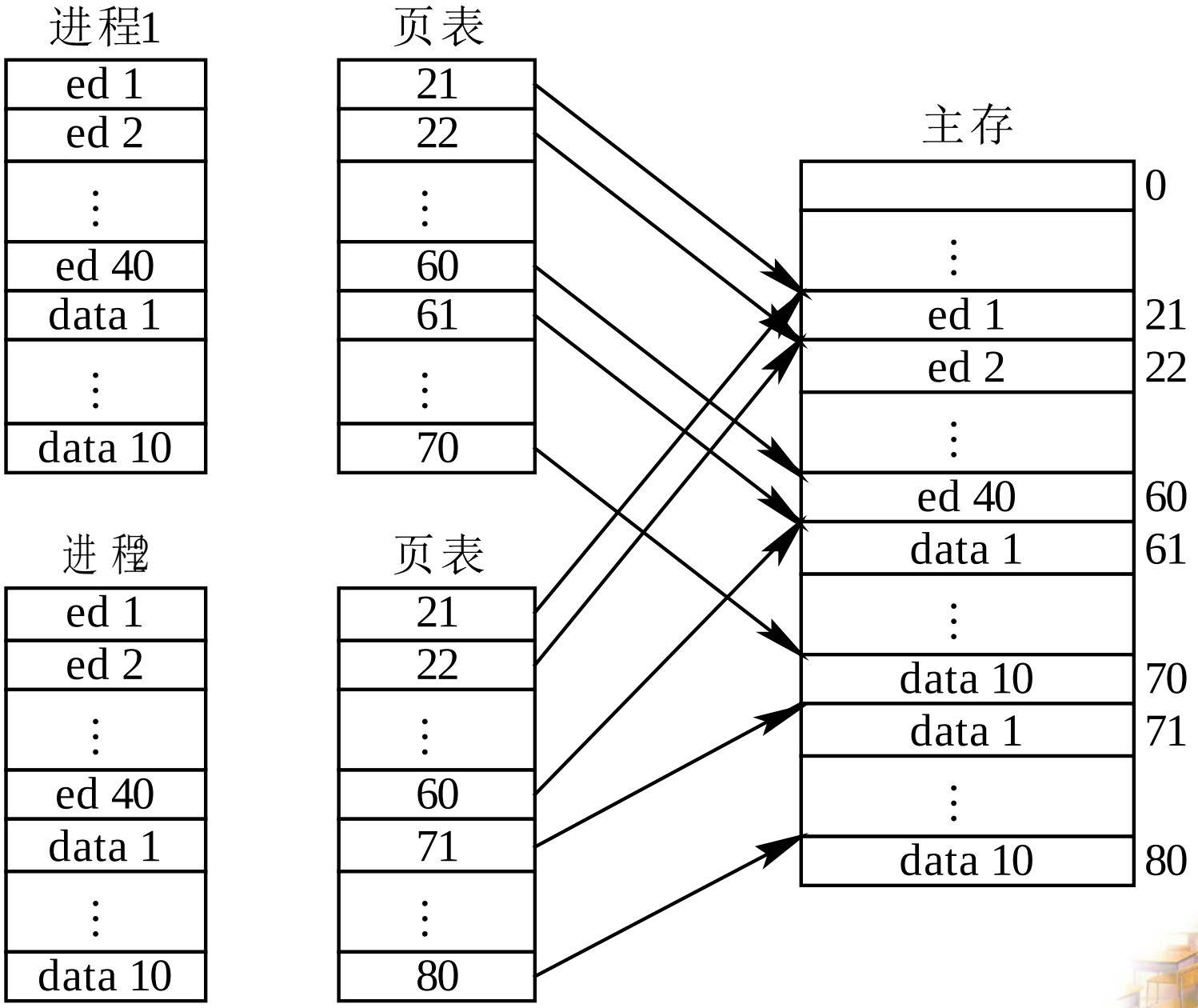


图 4-18 分页系统中共享editor的示意图

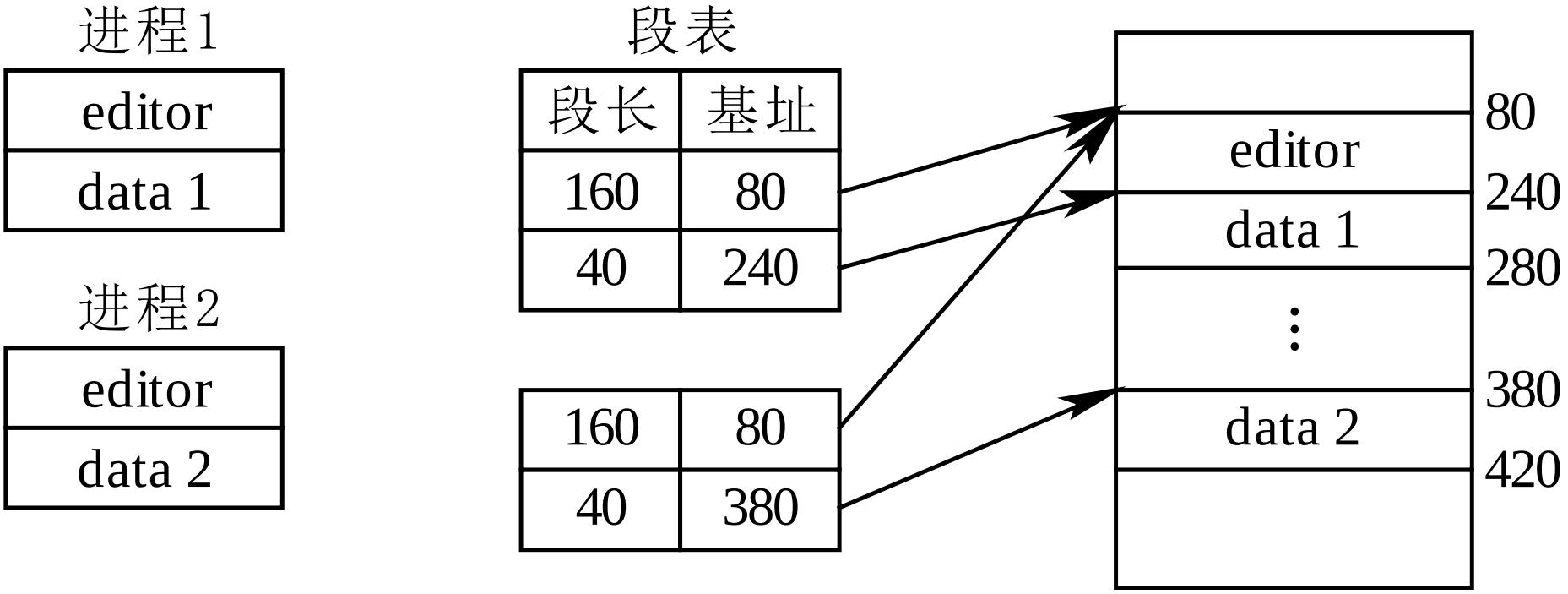
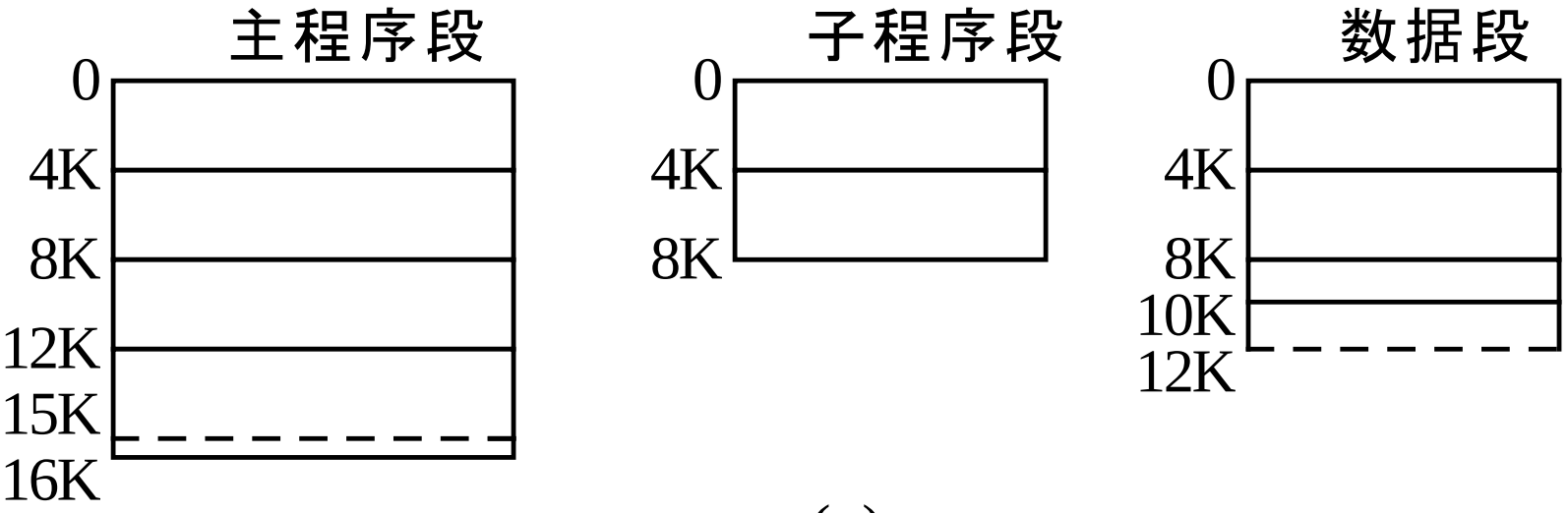


图 4-19 分段系统中共享editor的示意图

4.4.4 段页式存储管理方式

1. 基本原理



(*a*)

(*b*)

图 4-20 作业地址空间和地址结构

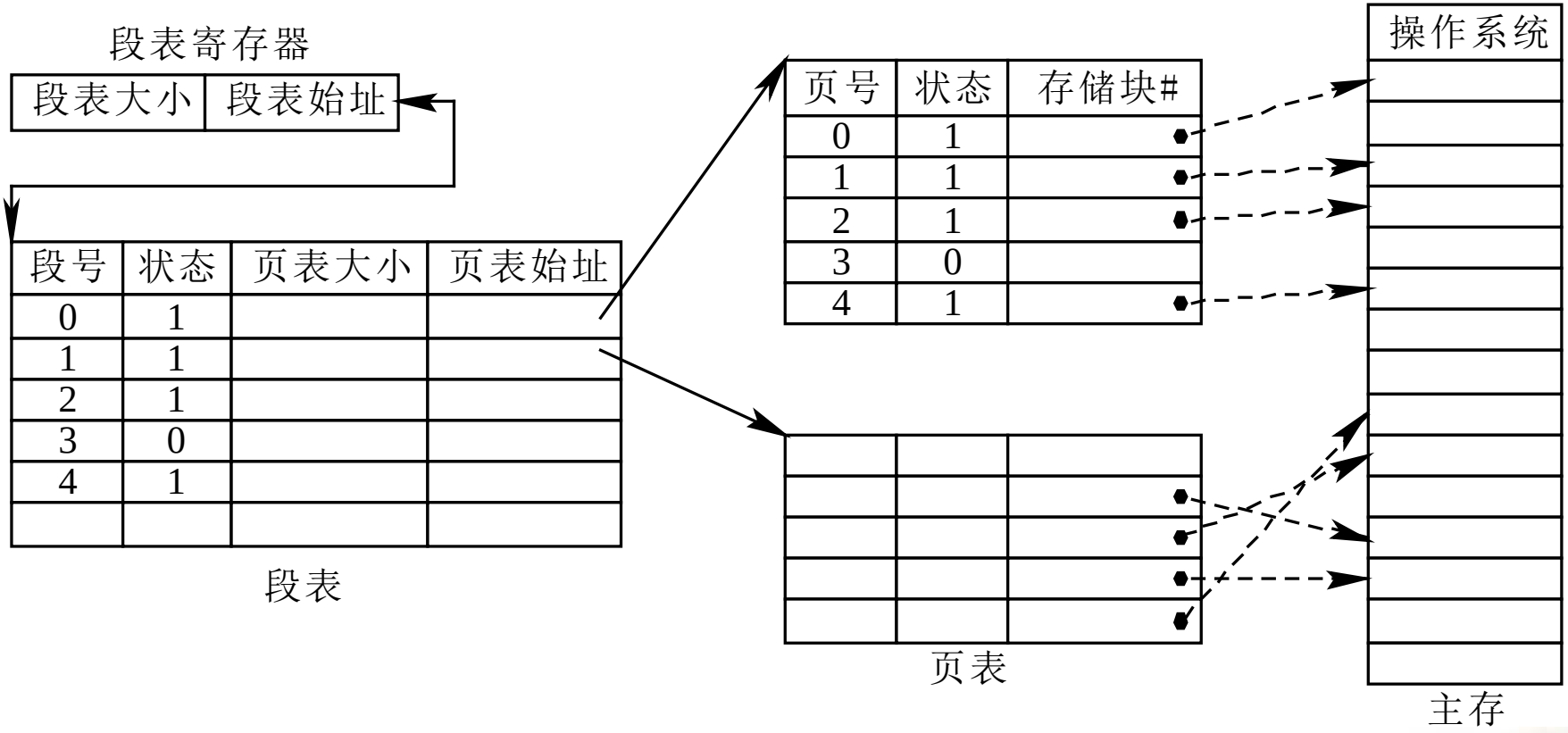


图 4-21 利用段表和页表实现地址映射

2. 地址变换过程

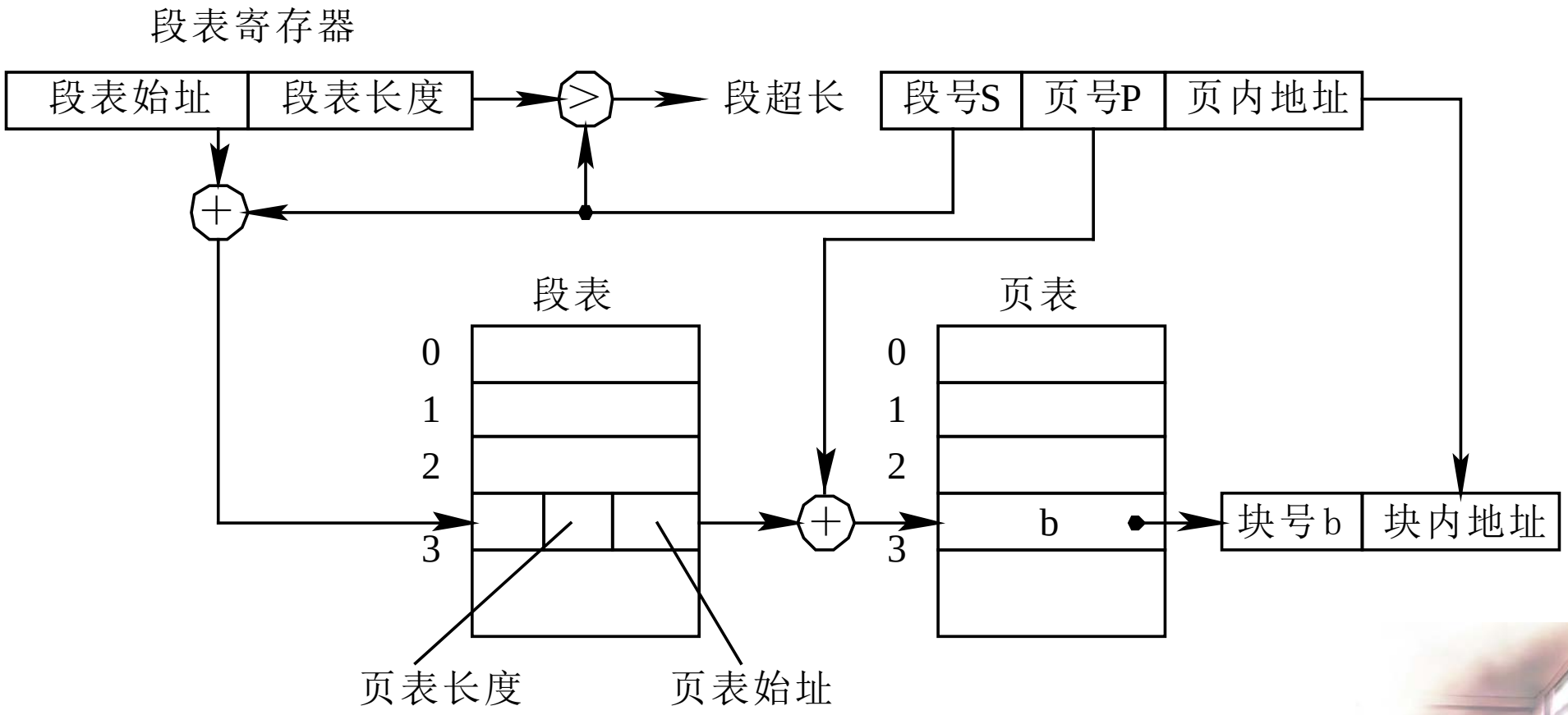


图 4-22 段页式系统中的地址变换机构

4.5 虚拟存储器的基本概念

4.5.1 虚拟存储器的引入

1. 常规存储器管理方式的特征

(1) 一次性。

(2) 驻留性。

2. 局部性原理

早在1968年， Denning.P就曾指出：

(1) 程序执行时， 除了少部分的转移和过程调用指令外，在大多数情况下仍是顺序执行的。

(2) 过程调用将会使程序的执行轨迹由一部分区域转至另一部分区域， 但经研究看出，过程调用的深度在大多数情况下都不超过5。

(3) 程序中存在许多循环结构， 这些虽然只由少数指令构成， 但是它们将多次执行。

(4) 程序中还包括许多对数据结构的处理， 如对数组进行操作， 它们往往都局限于很小的范围内。

局限性又表现在下述两个方面：

(1) 时间局限性。如果程序中的某条指令一旦执行， 则不久以后该指令可能再次执行；如果某数据被访问过，则不久以后该数据可能再次被访问。产生时间局限性的典型原因，是由于在程序中存在着大量的循环操作。

(2) 空间局限性。一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问，即程序在一段时间内所访问的地址，可能集中在一定的范围之内，其典型情况便是程序的顺序执行。

3. 虚拟存储器定义

所谓虚拟存储器， 是指具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储器系统。其逻辑容量由内存容量和外存容量之和所决定，其运行速度接近于内存速度，而每位的成本却又接近于外存。可见，虚拟存储技术是一种性能非常优越的存储器管理技术，故被广泛地应用于大、 中、 小型机器和微型机中。

4.5.2 虚拟存储器的实现方法

1. 分页请求系统

(1) 硬件支持。

① 请求分页的页表机制，它是在纯分页的页表机制上增加若干项而形成的，作为请求分页的数据结构；② 缺页中断机构，即每当用户程序要访问的页面尚未调入内存时 便产生一缺页中断，以请求OS将所缺的页调入内存；③ 地址变换机构， 它同样是在纯分页地址变换机构的基础上发展形成的。

(2) 实现请求分页的软件。

4.5.3 虚拟存储器的特征

1. 多次性

2. 对换性

3. 虚拟性

4.6 请求分页存储管理方式

4.6.1 请求分页中的硬件支持

1. 页表机制

页号	物理块号	状态位P	访问字段A	修改位M	外存地址

2. 缺页中断机构 页面

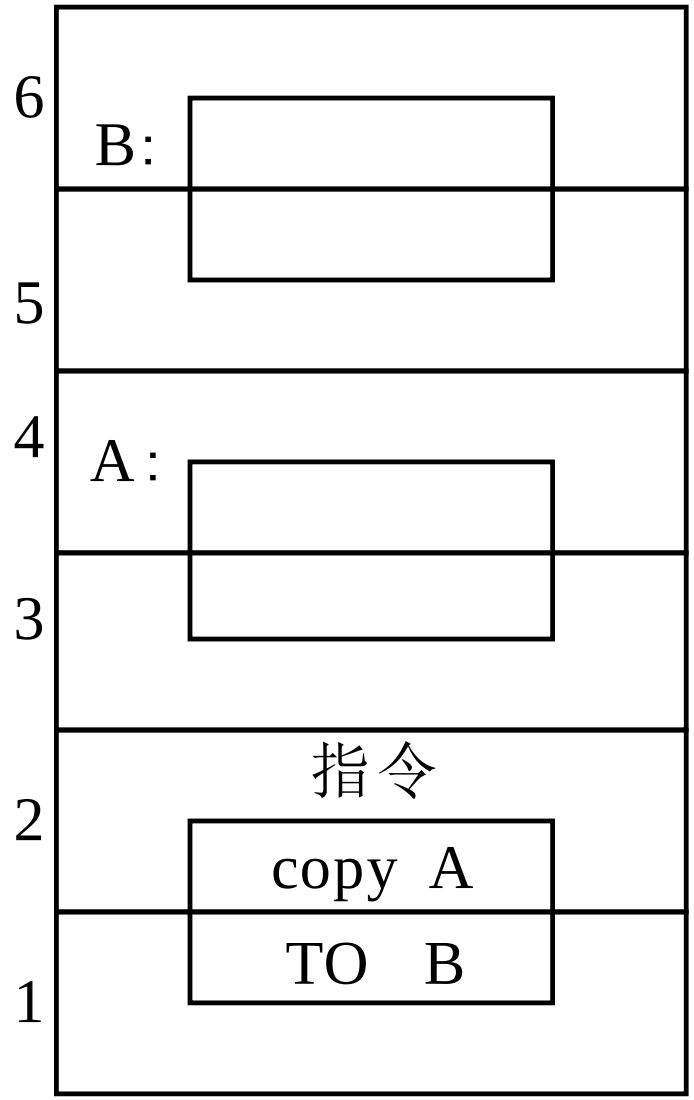


图 4-23 涉及6次缺页中断的指令

3. 地址变换机构

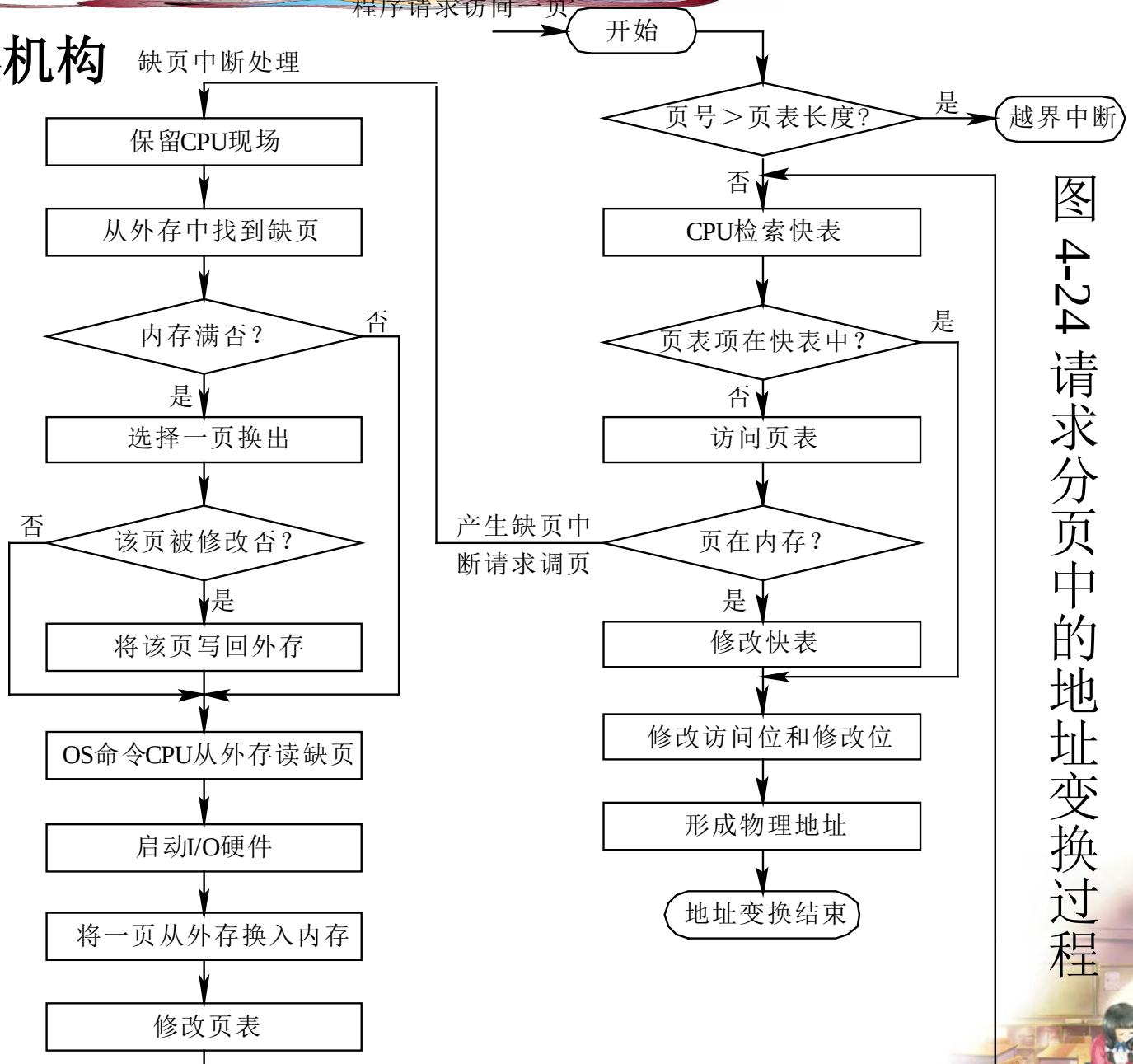


图 4-24 请求分页中的地址变换过程

4.6.2 内存分配策略和分配算法

1. 最小物理块数的确定

是指能保证进程正常运行所需的最小物理块数。当系统为进程分配的物理块数少于此值时，进程将无法运行。进程应获得的最少物理块数与计算机的硬件结构有关，取决于指令的格式、功能和寻址方式。对于某些简单的机器，若是单地址指令且采用直接寻址方式，则所需的最少物理块数为2。其中，一块是用于存放指令的页面，另一块则是用于存放数据的页面。如果该机器允许间接寻址时，则至少要求有三个物理块。对于某些功能较强的机器， 其指令长度可能是两个或多于两个字节，因而其指令本身有可能跨两个页面，且源地址和目标地址所涉及的区域也都可能跨两个页面。

2. 物理块的分配策略

在请求分页系统中，可采取两种内存分配策略，即固定和可变分配策略。在进行置换时， 也可采取两种策略，即全局置换和局部置换。于是可组合出以下三种适用的策略。

1) 固定分配局部置换(Fixed Allocation, Local Replacement)

2) 可变分配全局置换(Variable Allocation, Global Replacement)

3) 可变分配局部置换(Variable Allocation, Local Replacemen

3. 物理块分配算法

1) 平均分配算法

这是将系统中所有可供分配的物理块，平均分配给各个进程。 例如，当系统中有100个物理块，有5个进程在运行时，每个进程可分得20个物理块。这种方式貌似公平，但实际上是不公平的，因为它未考虑到各进程本身的大小。如有一个进程其大小为200页，只分配给它20个块，这样，它必然会有很高的缺页率；而另一个进程只有10页，却有10个物理块闲置未用。

2) 按比例分配算法

这是根据进程的大小按比例分配物理块的算法。如果系统中共有n个进程，每个进程的页面数为S_i，则系统中各进程页面数的总和为：

$$S = \sum_{i=1}^{n} S_i$$

又假定系统中可用的物理块总数为m，则每个进程所能分到的物理块数为b_i，将有：

$$b_i = \frac{S_i}{S} \times m$$

b应该取整，它必须大于最小物理块数。

3) 考虑优先权的分配算法

在实际应用中，为了照顾到重要的、紧迫的作业能尽快地完成， 应为它分配较多的内存空间。通常采取的方法是把内存中可供分配的所有物理块分成两部分：一部分按比例地分配给各进程；另一部分则根据各进程的优先权，适当地增加其相应份额后，分配给各进程。在有的系统中，如重要的实时控制系统，则可能是完全按优先权来为各进程分配其物理块的。

4.6.3 调页策略

1. 何时调入页面

1) 预调页策略

2) 请求调页策略

2. 从何处调入页面

在请求分页系统中的外存分为两部分：用于存放文件的文件区和用于存放对换页面的对换区。通常，由于对换区是采用连续分配方式，而事件是采用离散分配方式，故对换区的磁盘I/O速度比文件区的高。这样，每当发生缺页请求时，系统应从何处将缺页调入内存，可分成如下三种情况：

(1) 系统拥有足够的对换区空间，这时可以全部从对换区调入所需页面，以提高调页速度。为此，在进程运行前，便须将与该进程有关的文件，从文件区拷贝到对换区。

(2) 系统缺少足够的对换区空间，这时凡是不会被修改的文件，都直接从文件区调入；而当换出这些页面时，由于它们未被修改而不必再将它们换出，以后再调入时，仍从文件区直接调入。但对于那些可能被修改的部分，在将它们换出时，便须调到对换区，以后需要时，再从对换区调入。

(3) UNIX方式。由于与进程有关的文件都放在文件区，故凡是未运行过的页面，都应从文件区调入。而对于曾经运行过但又被换出的页面，由于是被放在对换区，因此在下次调入时，应从对换区调入。由于UNIX系统允许页面共享，因此， 某进程所请求的页面有可能已被其它进程调入内存，此时也就无须再从对换区调入。

3. 页面调入过程

每当程序所要访问的页面未在内存时，便向CPU发出一缺页中断，中断处理程序首先保留CPU环境，分析中断原因后，转入缺页中断处理程序。该程序通过查找页表，得到该页在外存的物理块后， 如果此时内存能容纳新页，则启动磁盘I/O将所缺之页调入内存，然后修改页表。如果内存已满，则须先按照某种置换算法从内存中选出一页准备换出；如果该页未被修改过，可不必将该页写回磁盘；但如果此页已被修改，则必须将它写回磁盘，然后再把所缺的页调入内存， 并修改页表中的相应表项，置其存在位为“1”，并将此页表项写入快表中。在缺页调入内存后，利用修改后的页表， 去形成所要访问数据的物理地址，再去访问内存数据。

4.7 页面置换算法

4.7.1 最佳置换算法和先进先出置换算法

1. 最佳(Optimal)置换算法

最佳置换算法是由Belady于1966年提出的一种理论上的算法。 其所选择的被淘汰页面，将是以后永不使用的， 或许是在最长(未来)时间内不再被访问的页面。采用最佳置换算法，通常可保证获得最低的缺页率。

假定系统为某进程分配了三个物理块， 并考虑有以下的页面号引用串：

7，0，1，2，0，3，0，4，2，3，0，3，2，1，2，0，1，7，0，1

进程运行时， 先将7，0，1三个页面装入内存。 以后，当进程要访问页面2时， 将会产生缺页中断。此时OS根据最佳置换算法， 将选择页面7予以淘汰。

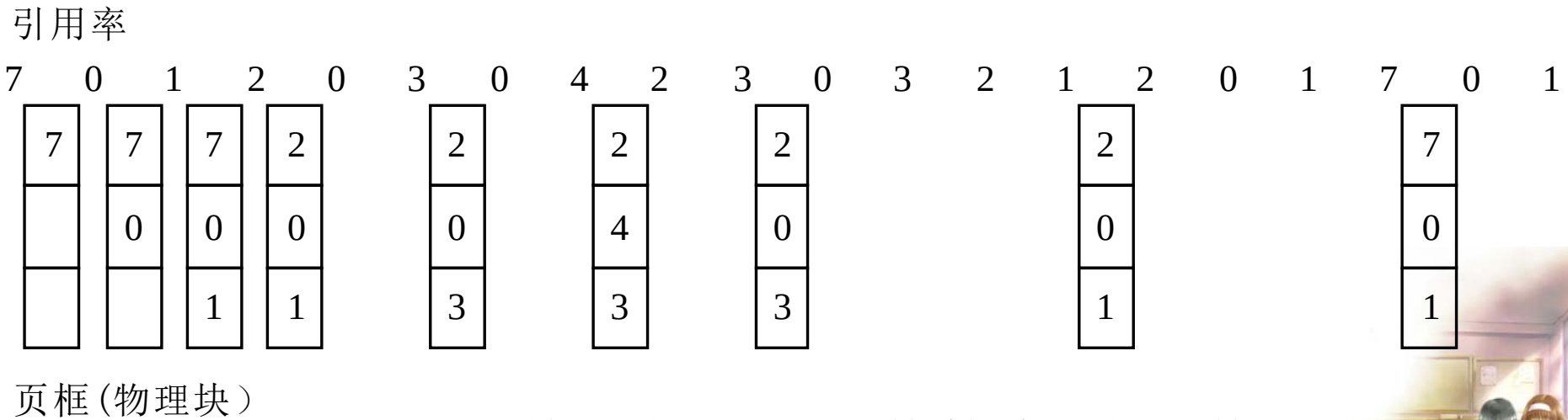


图 4-25 利用最佳页面置换算法时的置换图

2. 先进先出(FIFO)页面置换算法

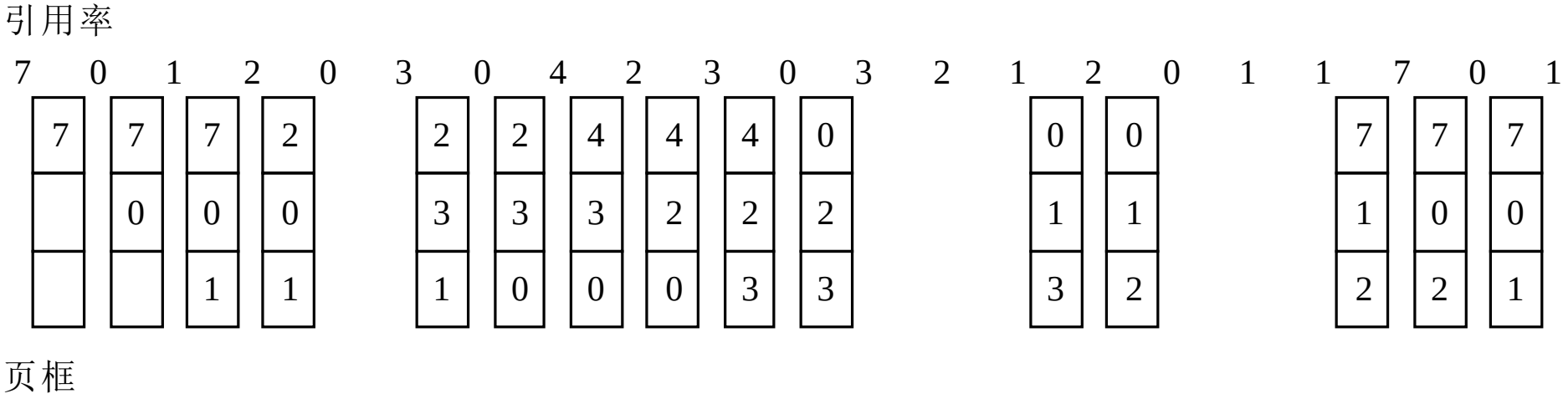


图 4-26 利用FIFO置换算法时的置换图

4.7.2 最近最久未使用(LRU)置换算法

1. LRU(Least Recently Used)置换算法的描述

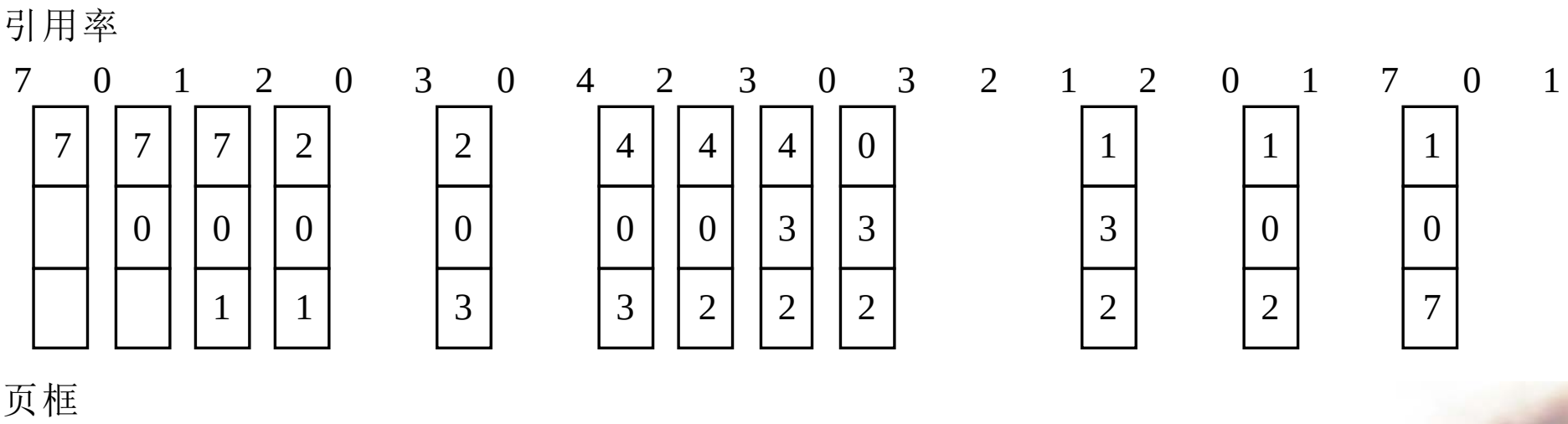


图 4-27 LRU页面置换算法

2. LRU置换算法的硬件支持

1) 寄存器

为了记录某进程在内存中各页的使用情况，须为每个在内存中的页面配置一个移位寄存器，可表示为

$$R=R_{n-1}R_{n-2}R_{n-3}\ldots R_2R_1R_0$$

实页 \ R	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0
1	0	1	0	1	0	0	1	0
2	1	0	1	0	1	1	0	0
3	0	0	0	0	0	1	0	0
4	0	1	1	0	1	0	1	1
5	1	1	0	1	0	1	1	0
6	0	0	1	0	1	0	1	1
7	0	0	0	0	0	1	1	1
8	0	1	1	0	1	1	0	1

图 4-28 某进程具有8个页面时的LRU访问情况

2) 栈

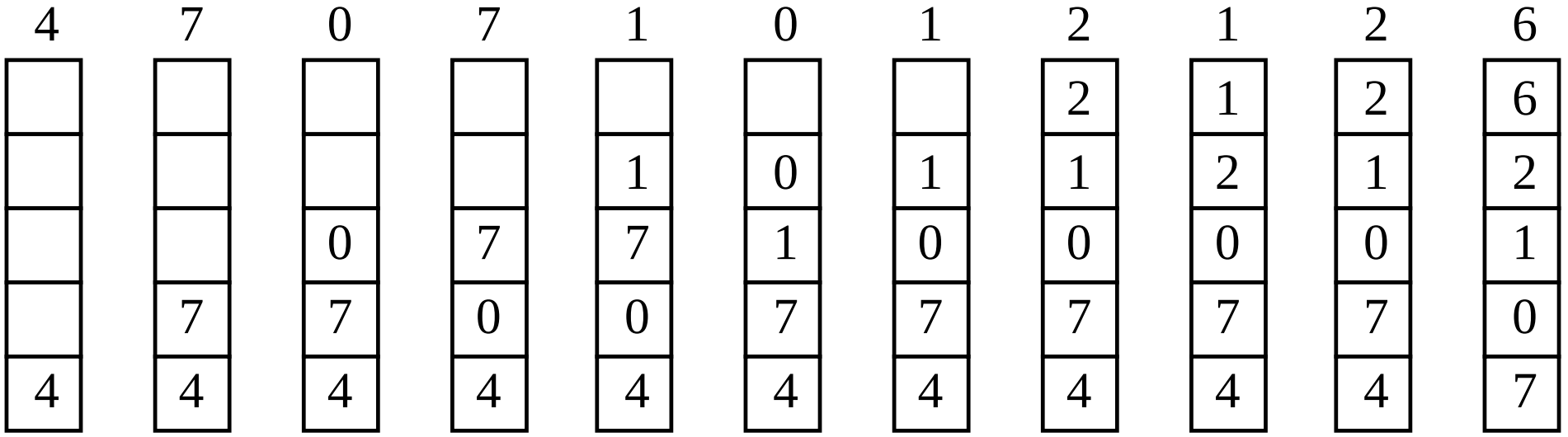
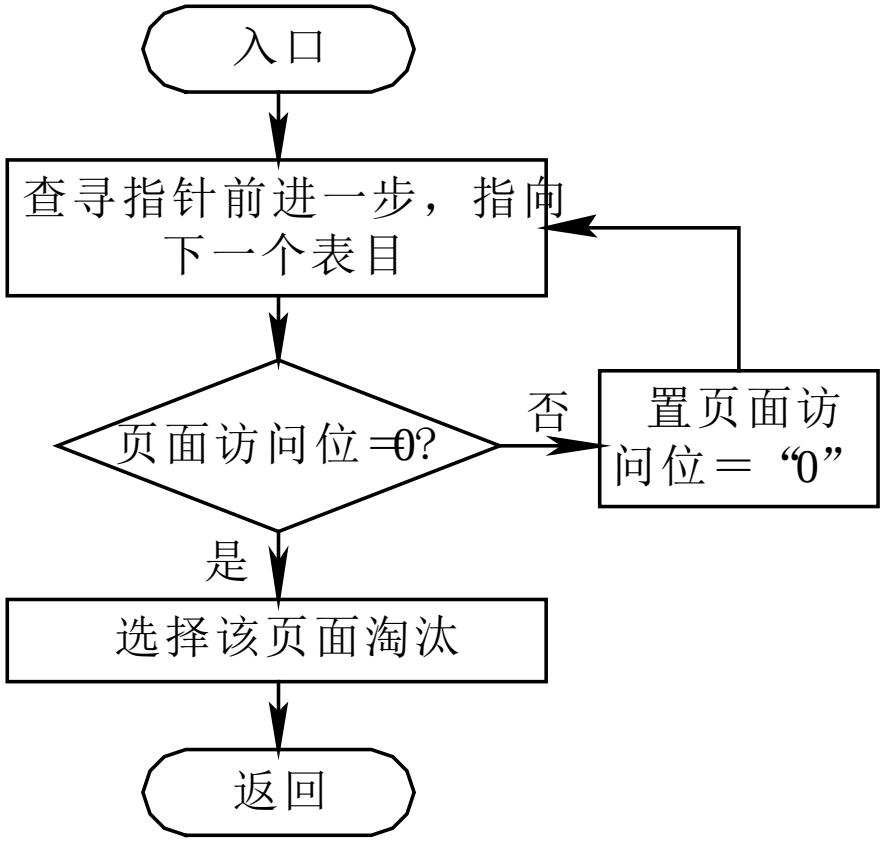


图 4-29 用栈保存当前使用页面时栈的变化情况

4.7.3 Clock置换算法

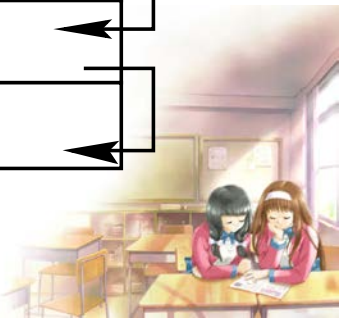
1. 简单的Clock置换算法



块号	页号	访问位	指针
0			
1			
2	4	0	
3			
4	2	1	
5			
6	5	0	
7	1	1	

替换指针

图 4-30 简单Clock置换算法的流程和示例



2. 改进型Clock置换算法

由访问位A和修改位M可以组合成下面四种类型的页面：

1类(A=0, M=0): 表示该页最近既未被访问， 又未被修改，是最佳淘汰页。

2类(A=0, M=1)： 表示该页最近未被访问， 但已被修改，并不是很好的淘汰页。

3类(A=1, M=0)： 最近已被访问， 但未被修改， 该页有可能再被访问。

4类(A=1, M=1): 最近已被访问且被修改， 该页可能再被访问。

其执行过程可分成以下三步：

(1) 从指针所指示的当前位置开始， 扫描循环队列， 寻找A=0且M=0的第一类页面， 将所遇到的第一个页面作为所选中的淘汰页。 在第一次扫描期间不改变访问位A。

(2) 如果第一步失败，即查找一周后未遇到第一类页面，则开始第二轮扫描，寻找A=0且M=1的第二类页面，将所遇到的第一个这类页面作为淘汰页。在第二轮扫描期间，将所有扫描过的页面的访问位都置0。

(3) 如果第二步也失败，亦即未找到第二类页面，则将指针返回到开始的位置，并将所有的访问位复0。 然后重复第一步，如果仍失败，必要时再重复第二步，此时就一定能找到被淘汰的页。

4.7.4 其它置换算法

1. 最少使用(LFU： Least Frequently Used)置换算法

2. 页面缓冲算法(PBA： Page Buffering Algorithm)

4.8 请求分段存储管理方式

4.8.1 请求分段中的硬件支持

1. 段表机制

段名	段长	段的基址	存取方式	访问字段A	修改位M	存在位P	增补位	外存始址

在段表项中， 除了段名(号)、 段长、 段在内存中的起始地址外， 还增加了以下诸项：

(1) 存取方式。

(2) 访问字段A。

(3) 修改位M。

(4) 存在位P。

(5) 增补位。

(6) 外存始址。

2. 缺段中断机构

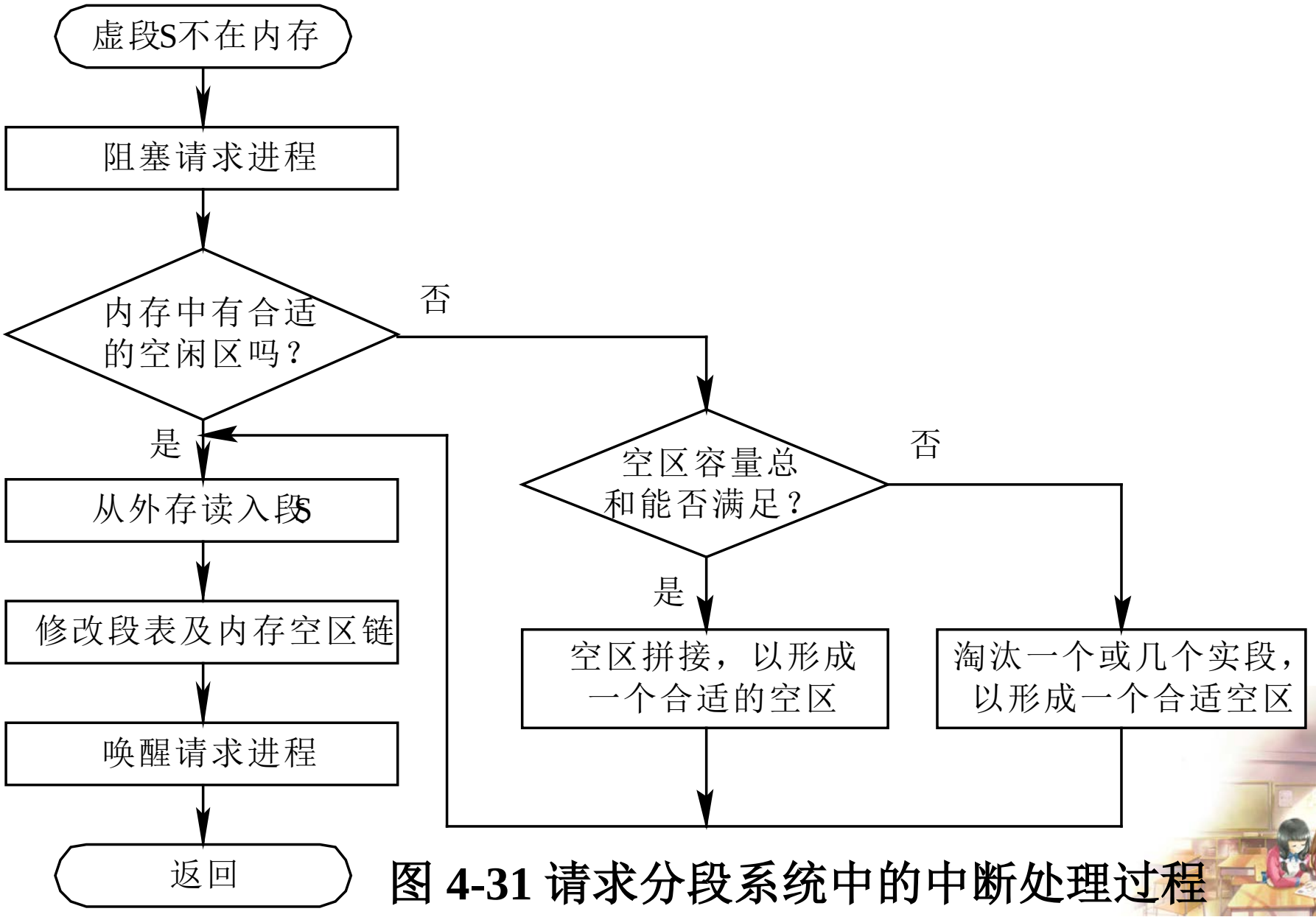


图 4-31 请求分段系统中的中断处理过程

3. 地址变换机构

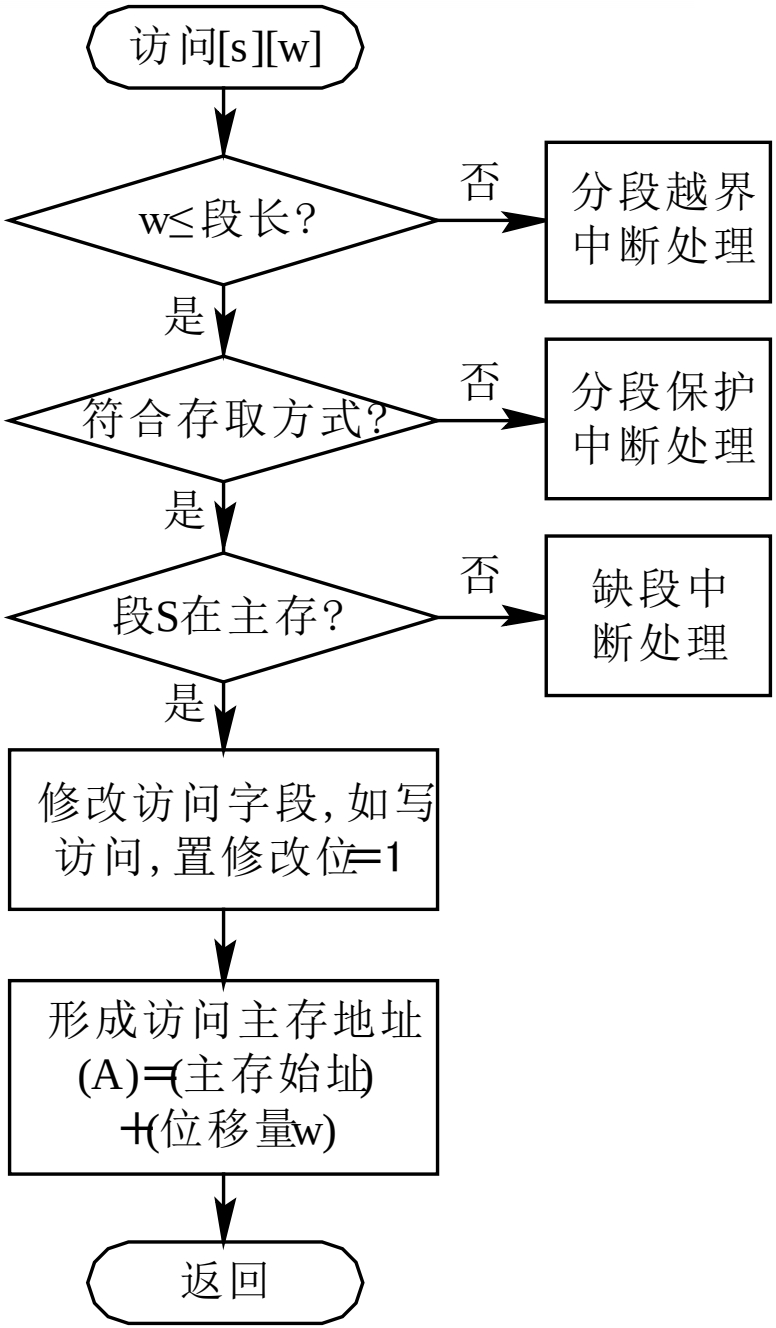


图 4-32 请求分段系统的地址变换过程

4.8.2 分段的共享与保护

1. 共享段表

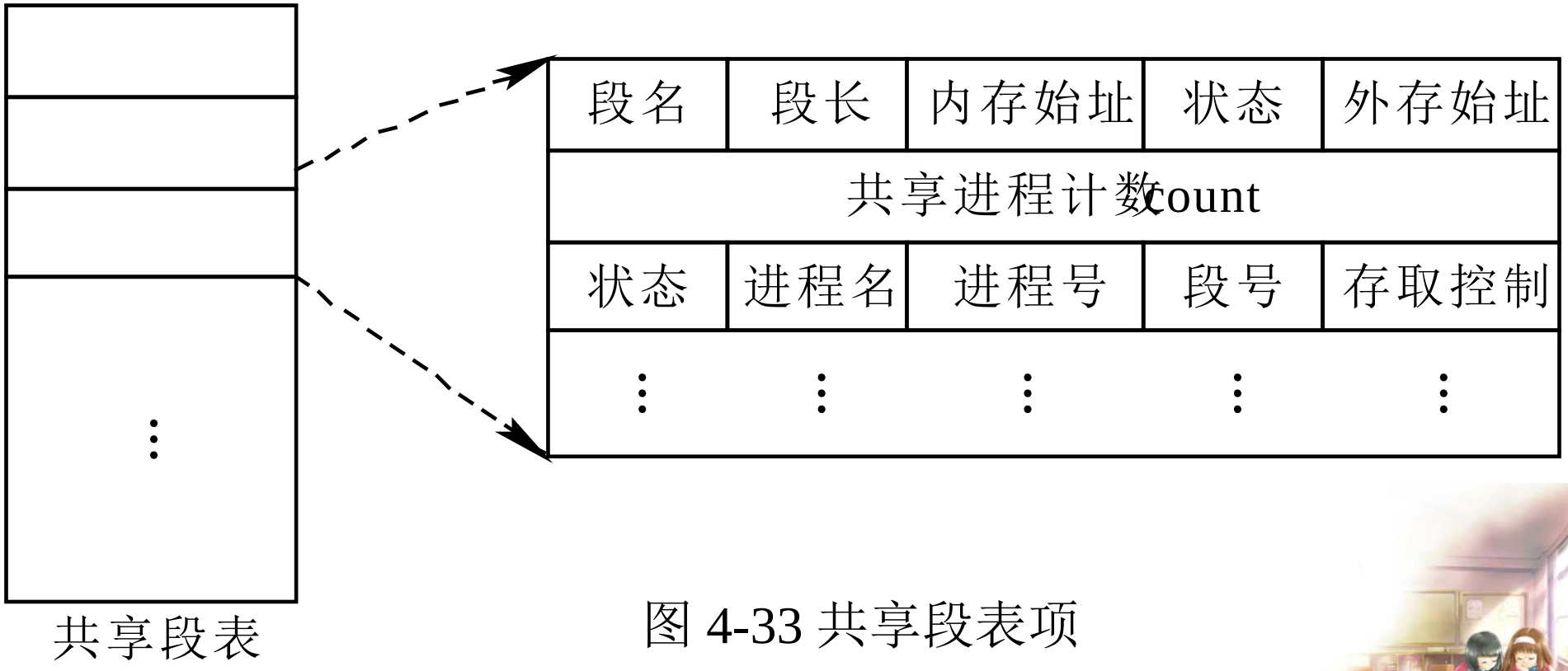
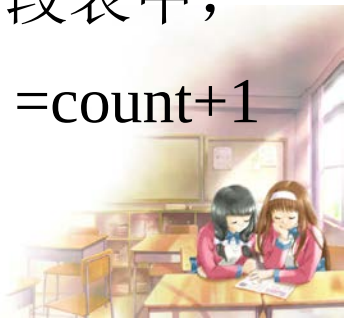


图 4-33 共享段表项

2. 共享段的分配与回收

1) 共享段的分配

在为共享段分配内存时，对第一个请求使用该共享段的进程，由系统为该共享段分配一物理区，再把共享段调入该区，同时将该区的始址填入请求进程的段表的相应项中，还须在共享段表中增加一表项，填写有关数据，把count置为1；之后，当又有其它进程需要调用该共享段时，由于该共享段已被调入内存，故此时无须再为该段分配内存，而只需在调用进程的段表中，增加一表项，填写该共享段的物理地址；在共享段的段表中，填上调用进程的进程名、存取控制等，再执行count：=count+1操作，以表明有两个进程共享该段。



2) 共享段的回收

当共享此段的某进程不再需要该段时，应将该段释放，包括撤在该进程段表中共享段所对应的表项，以及执行count：= count-1操作。若结果为0，则须由系统回收该共享段的物理内存，以及取消在共享段表中该段所对应的表项， 表明此时已没有进程使用该段；否则(减1结果不为0)，则只是取消调用者进程在共享段表中的有关记录。

3. 分段保护

1) 越界检查

2) 存取控制检查

(1) 只读

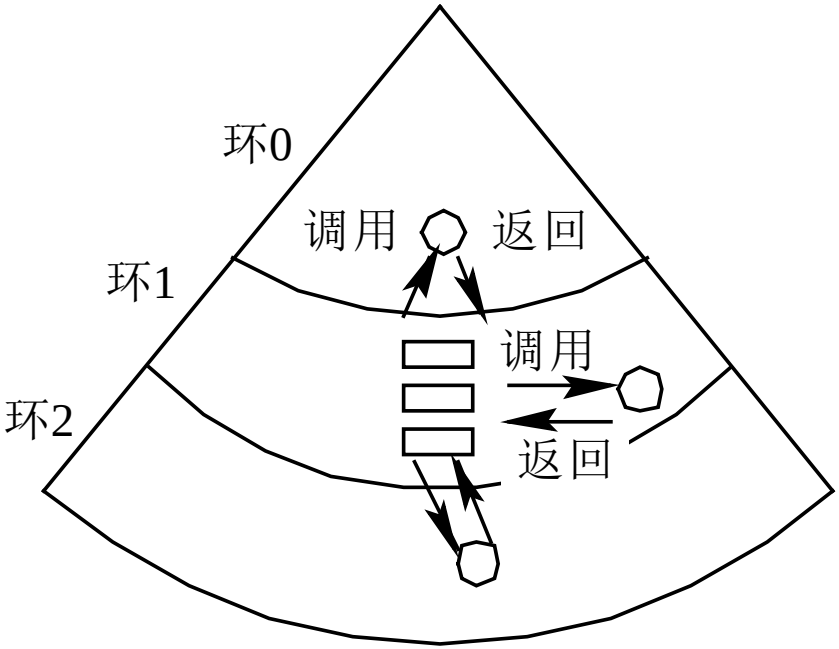
(2) 只执行

(3) 读/写

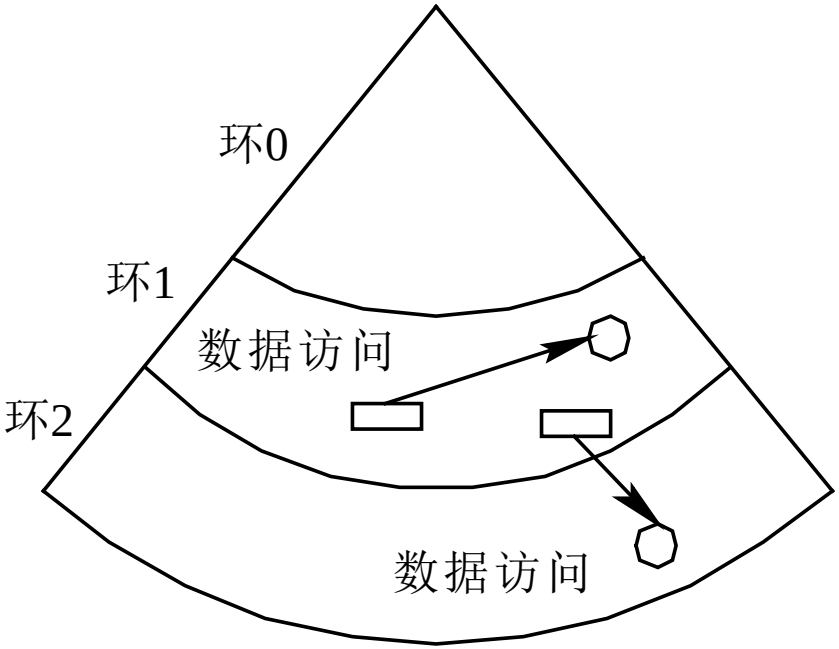
3) 环保护机构

(1) 一个程序可以访问驻留在相同环或较低特权环中的数据。

(2) 一个程序可以调用驻留在相同环或较高特权环中的服务。



(*a*) 程序间的控制传输



(*b*) 数据访问

图 4-34 环保护机构

第五章 设备管理

5.1 I/O 系 统

5.1.1 I/O设备

1. I/O设备的类型

1) 按传输速率分类

按传输速度的高低，可将I/O设备分为三类。第一类是低速设备，这是指其传输速率仅为每秒钟几个字节至数百个字节的一类设备。属于低速设备的典型设备有键盘、 鼠标器、语音的输入和输出等设备。第二类是中速设备，这是指其传输速率在每秒钟数千个字节至数万个字节的一类设备。典型的中速设备有行式打印机、激光打印机等。第三类是高速设备， 这是指其传输速率在数百千个字节至数十兆字节的一类设备。 典型的高速设备有磁带机、 磁盘机、 光盘机等。

2) 按信息交换的单位分类

可将I/O设备分成两类。第一类是块设备(Block Device)，这类设备用于存储信息。 由于信息的存取总是以数据块为单位，故而得名。 它属于有结构设备。典型的块设备是磁盘，每个盘块的大小为512 B~4 KB。磁盘设备的基本特征是其传输速率较高，通常每秒钟为几兆位；另一特征是可寻址，即对它可随机地读/写任一块；此外，磁盘设备的I/O常采用DMA方式。 第二类是字符设备(Character Device)，用于数据的输入和输出。 其基本单位是字符， 故称为字符设备。

3) 按设备的共享属性分类

这种分类方式可将I/O设备分为如下三类：

(1) 独占设备。

(2) 共享设备。

(3) 虚拟设备。

2. 设备与控制器之间的接口

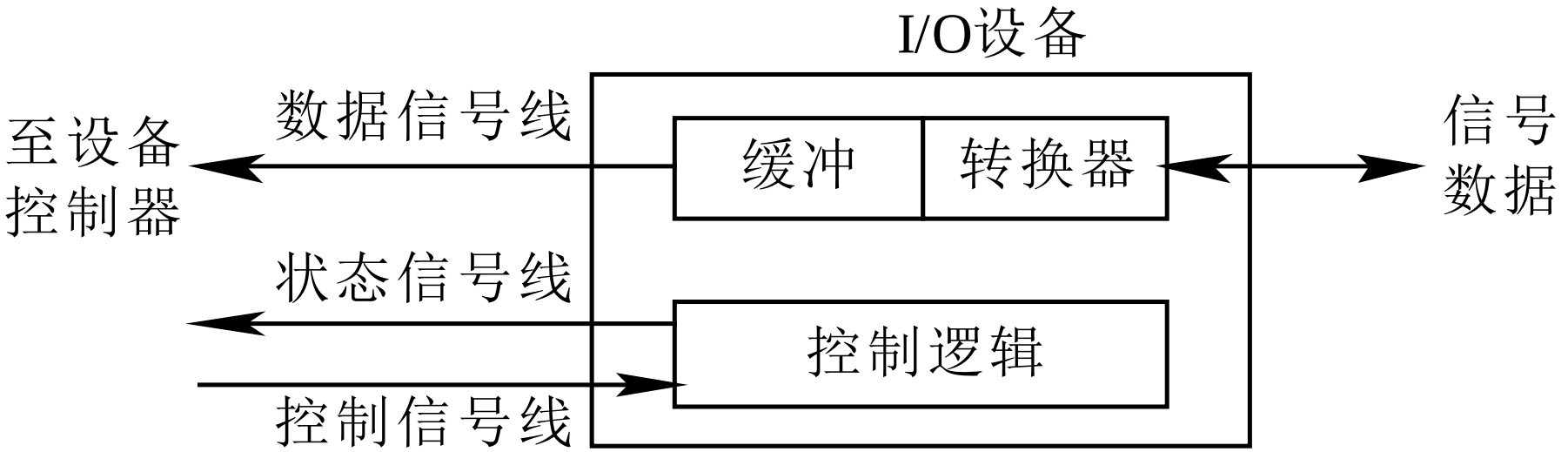


图 5-1 设备与控制器间的接口

5.1.2 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

6) 差错控制

2. 设备控制器的组成

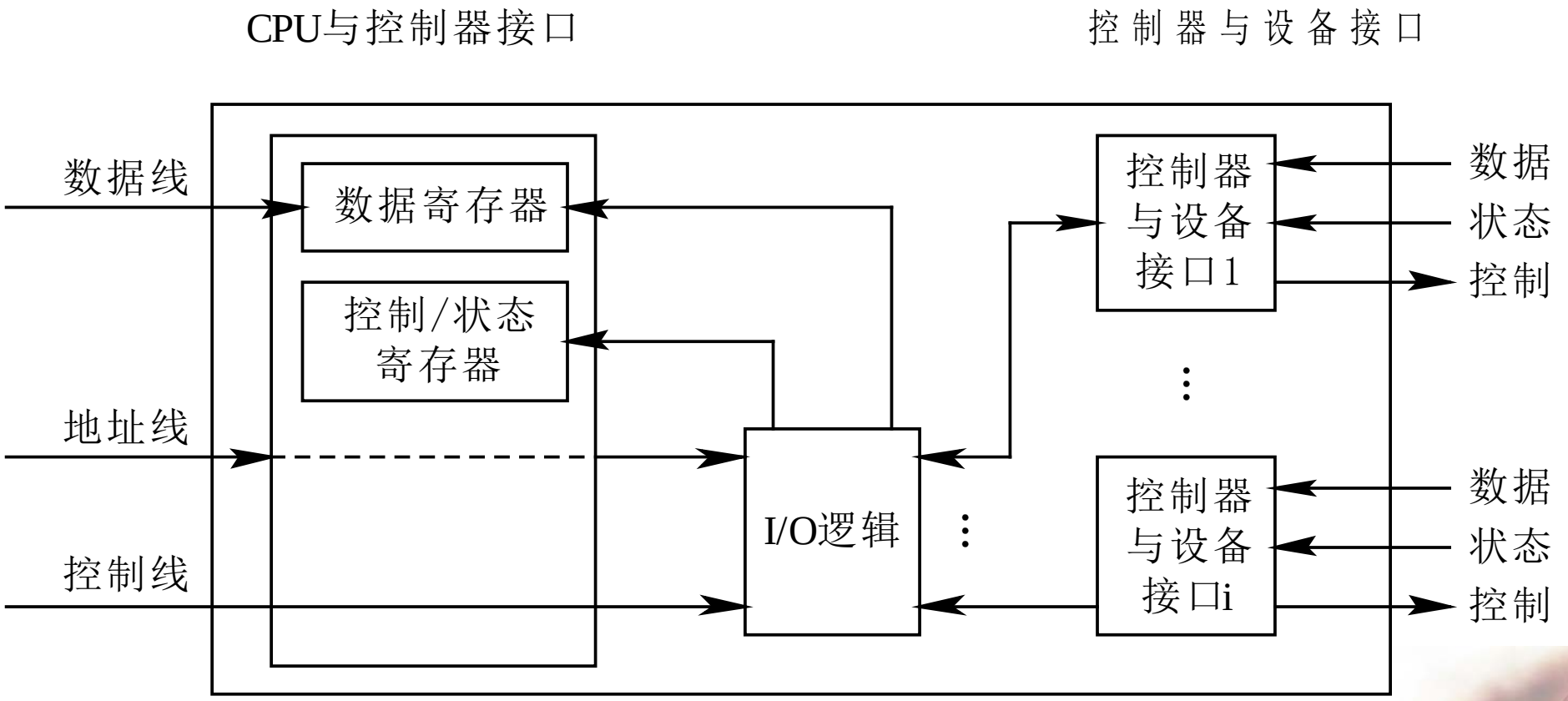


图 5-2 设备控制器的组成

5.1.3 I/O通道

1. I/O通道(I/O Channel)设备的引入

实际上，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。但I/O通道又与一般的处理机不同，主要表现在以下两个方面：

一是其指令类型单一，这是由于通道硬件比较简单， 其所能执行的命令，主要局限于与I/O操作有关的指令； 再就是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， 换言之，是通道与CPU共享内存。

2. 通道类型

1) 字节多路通道(Byte Multiplexor Channel)

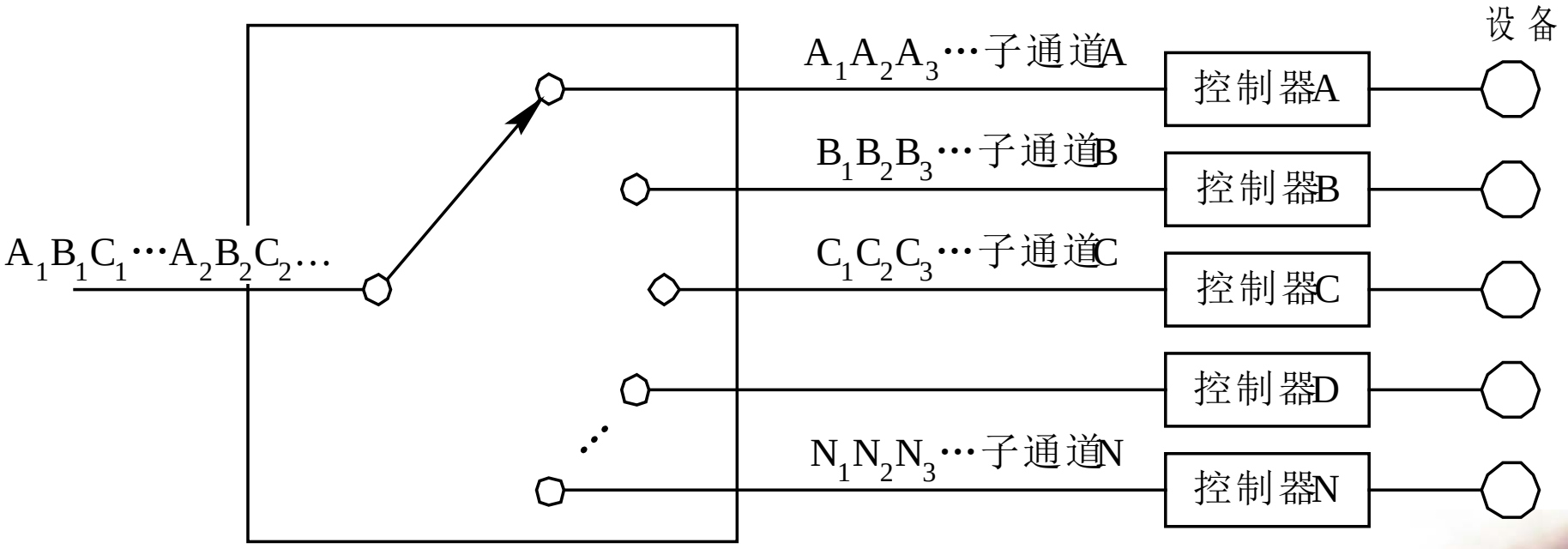


图 5-3 字节多路通道的工作原理

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。这种通道虽然可以连接多台高速设备，但由于它只含有一个分配型子通道，在一段时间内只能执行一道通道程序， 控制一台设备进行数据传送， 致使当某台设备占用了该通道后，便一直由它独占，即使是它无数据传送，通道被闲置， 也不允许其它设备使用该通道， 直至该设备传送完毕释放该通道。可见，这种通道的利用率很低。

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。它含有多个非分配型子通道， 因而这种通道既具有很高的数据传输速率，又能获得令人满意的通道利用率。也正因此，才使该通道能被广泛地用于连接多台高、中速的外围设备，其数据传送是按数组方式进行的。

3. “瓶颈”问题

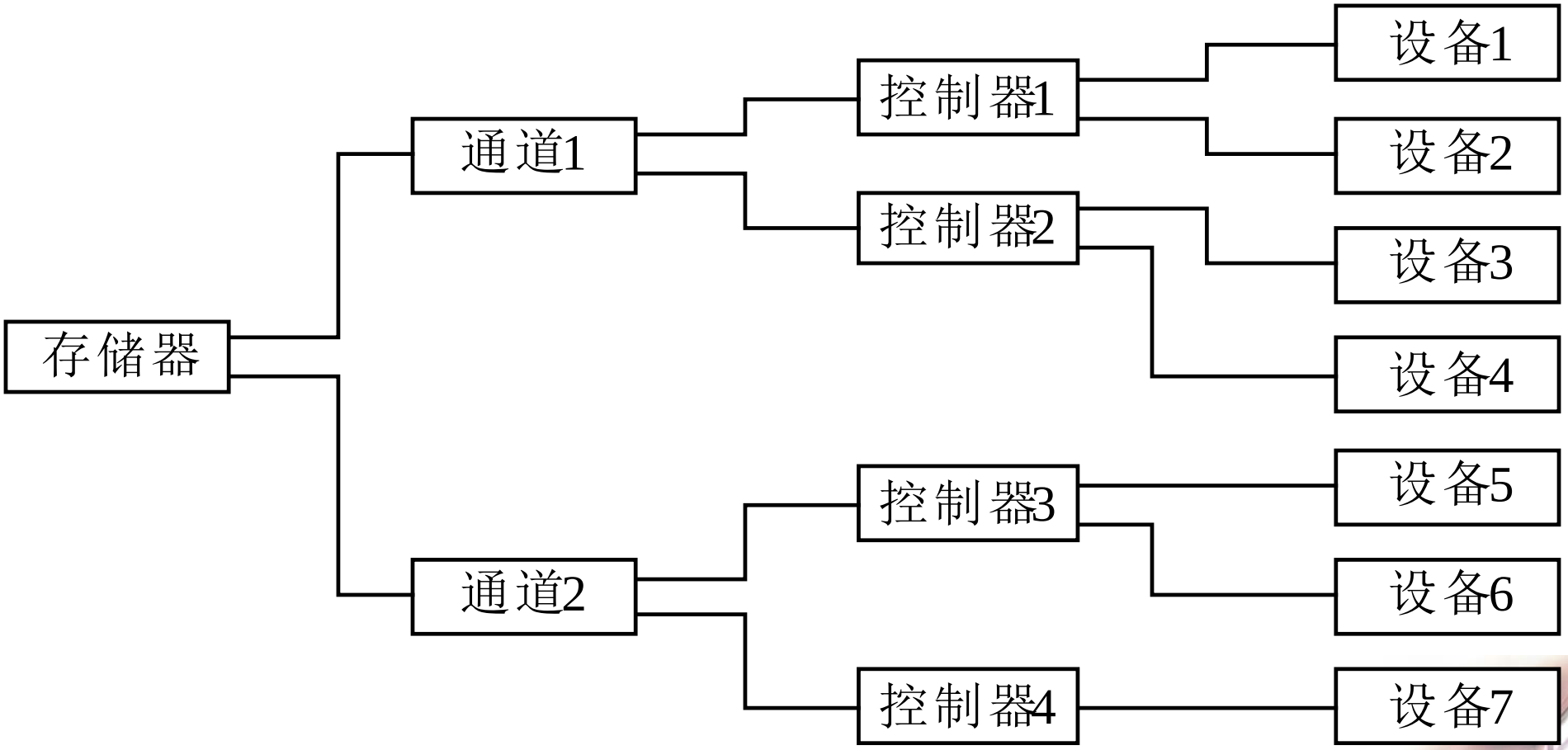
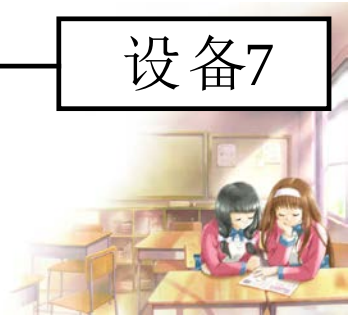


图 5-4 单通路I/O系统



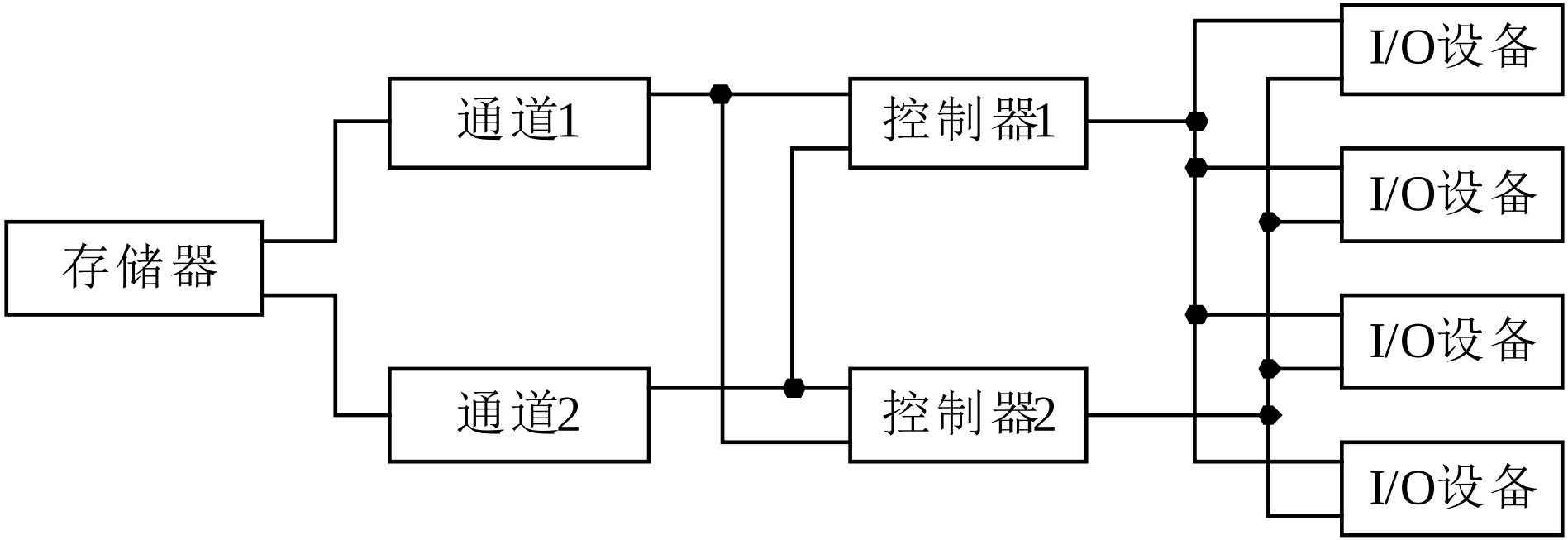


图 5-5 多通路I/O系统

5.1.4 总线系统

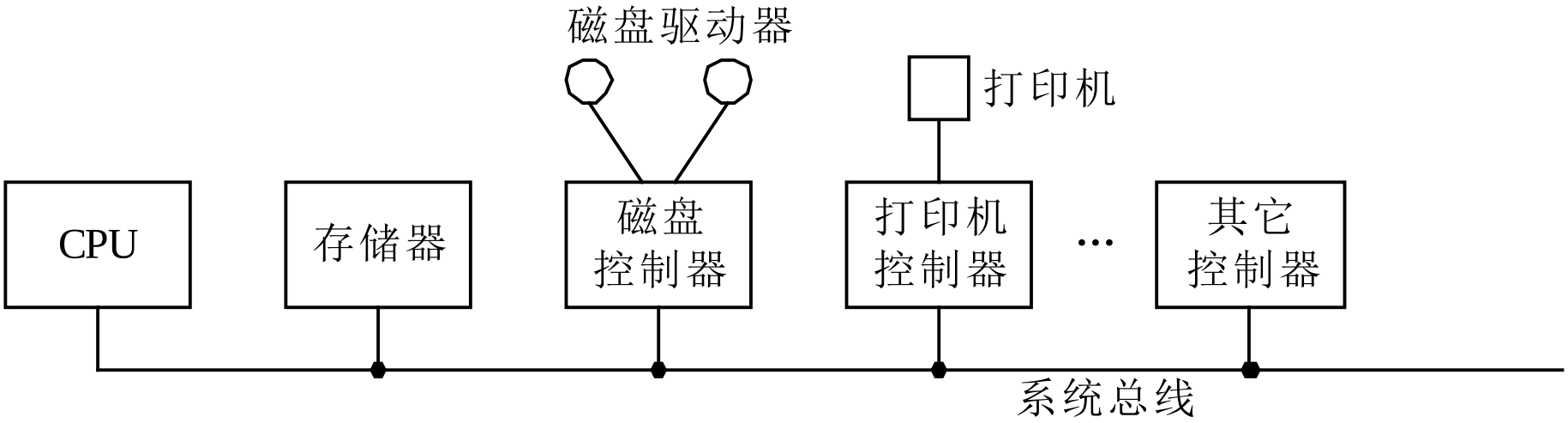


图 5-6 总线型I/O系统结构

1. ISA和EISA总线

1) ISA(Industry Standard Architecture)总线

这是为了1984年推出的80286型微机而设计的总线结构。 其总线的带宽为8位，最高传输速率为2 Mb/s。之后不久又推出了16位的(EISA)总线，其最高传输速率为8 Mb/s，后又升至16 Mb/s， 能连接12台设备。

2) EISA(Extended ISA)总线

到80年代末期，ISA总线已难于满足带宽和传输速率的要求，于是人们又开发出扩展ISA(EISA)总线，其带宽为32位，总线的传输速率高达32 Mb/s，同样可以连接12台外部设备。

2. 局部总线(Local Bus)

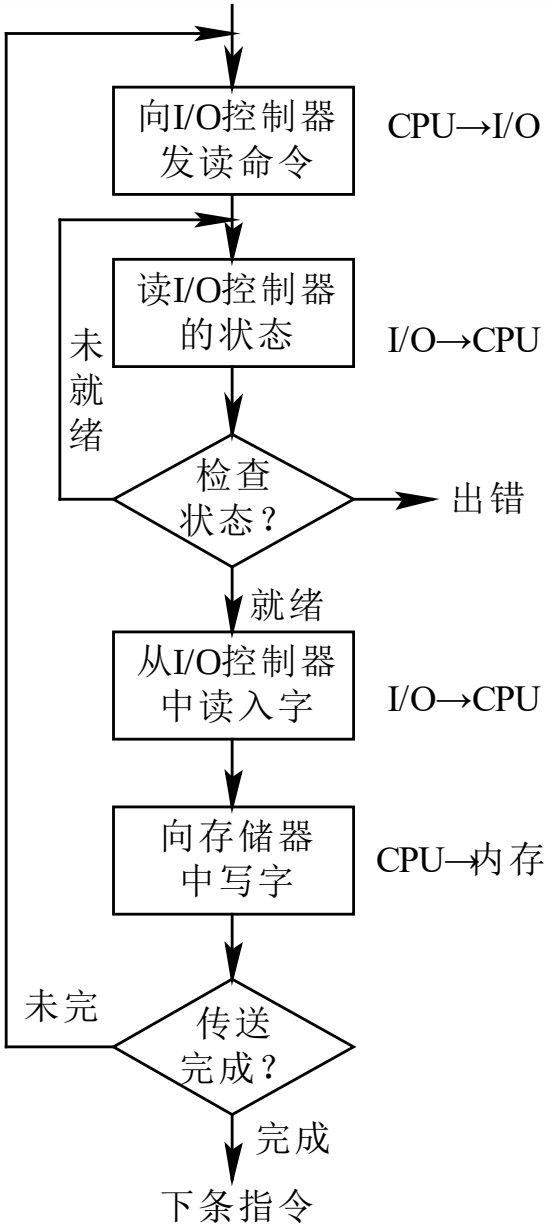
1) VESA(Video Electronic Standard Association)总线

2) PCI(Peripheral Component Interface)总线

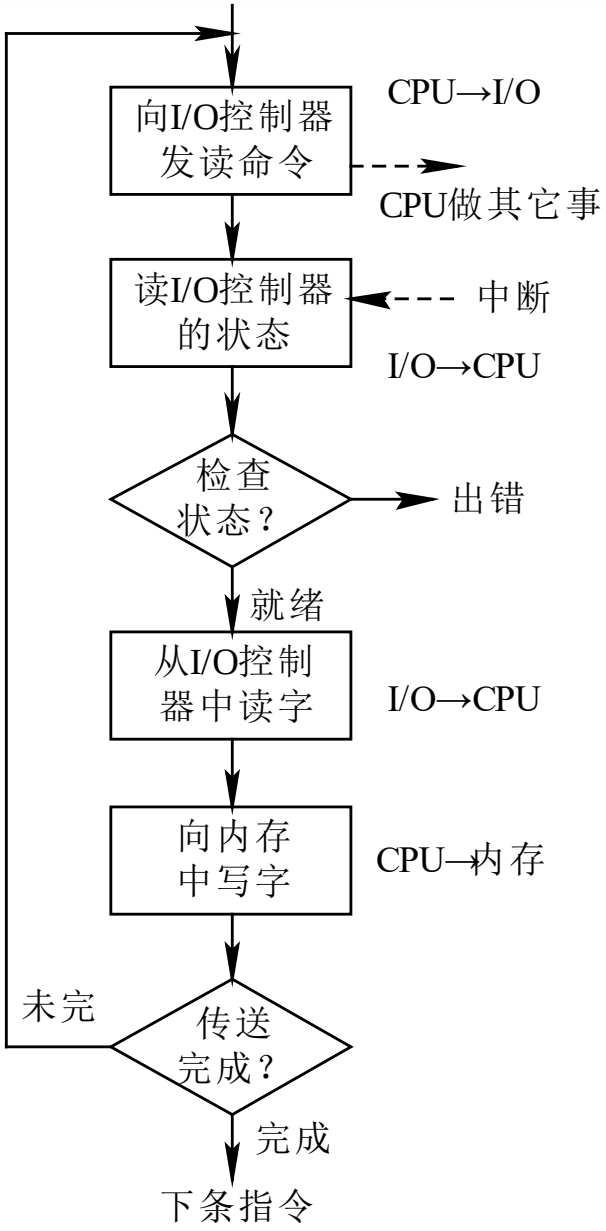
5.2 I/O控制方式

5.2.1 程序I/O方式

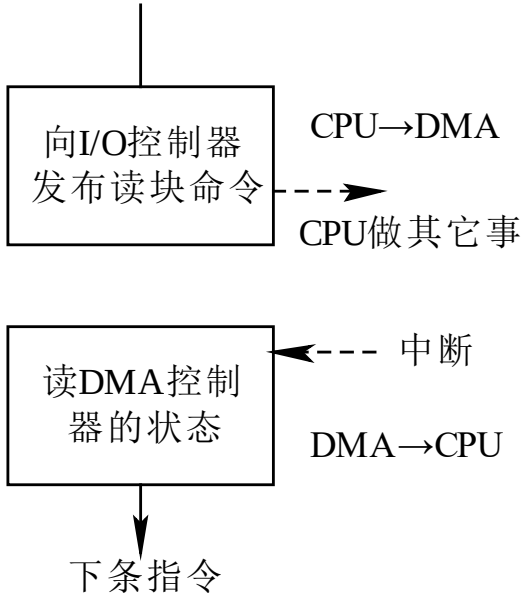
在程序I/O方式中，由于CPU的高速性和I/O设备的低速性，致使CPU的绝大部分时间都处于等待I/O设备完成数据I/O的循环测试中， 造成对CPU的极大浪费。在该方式中，CPU之所以要不断地测试I/O设备的状态，就是因为在CPU中无中断机构， 使I/O设备无法向CPU报告它已完成了一个字符的输入操作。



(*a*) 程序I/O方式



(*b*) 中断驱动方式



(*c*) DMA方式

图 5-7 程序I/O和中断驱动方式的流程

5.2.2 中断驱动I/O控制方式

在I/O设备输入每个数据的过程中，由于无须CPU干预，因而可使CPU与I/O设备并行工作。仅当输完一个数据时，才需CPU花费极短的时间去做些中断处理。可见，这样可使CPU和I/O设备都处于忙碌状态，从而提高了整个系统的资源利用率及吞吐量。例如，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。 若采用程序I/O方式，CPU约有 99.9 ms的时间处于忙–等待中。 采用中断驱动方式后，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。 可见，中断驱动方式可以成百倍地提高CPU的利用率。

5.2.3 直接存储器访问DMA I/O控制方式

1. DMA(Direct Memory Access)控制方式的引入

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，又是成百倍地减少了CPU对I/O的干预，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

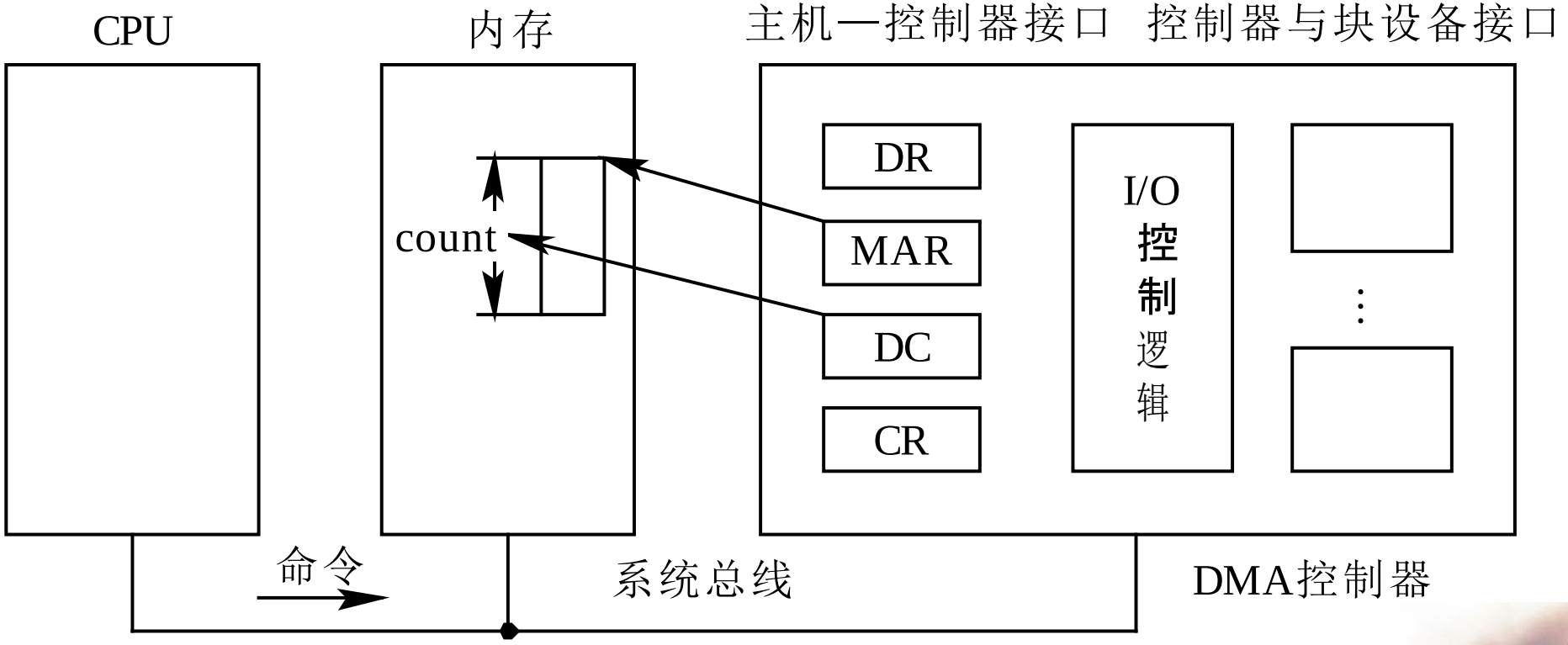


图 5-8 DMA控制器的组成

为了实现在主机与控制器之间成块数据的直接交换， 必须在DMA控制器中设置如下四类寄存器：

(1) 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。

(2) 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。

(3) 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。

(4) 数据计数器DC。 存放本次CPU要读或写的字(节)数。

3. DMA工作过程

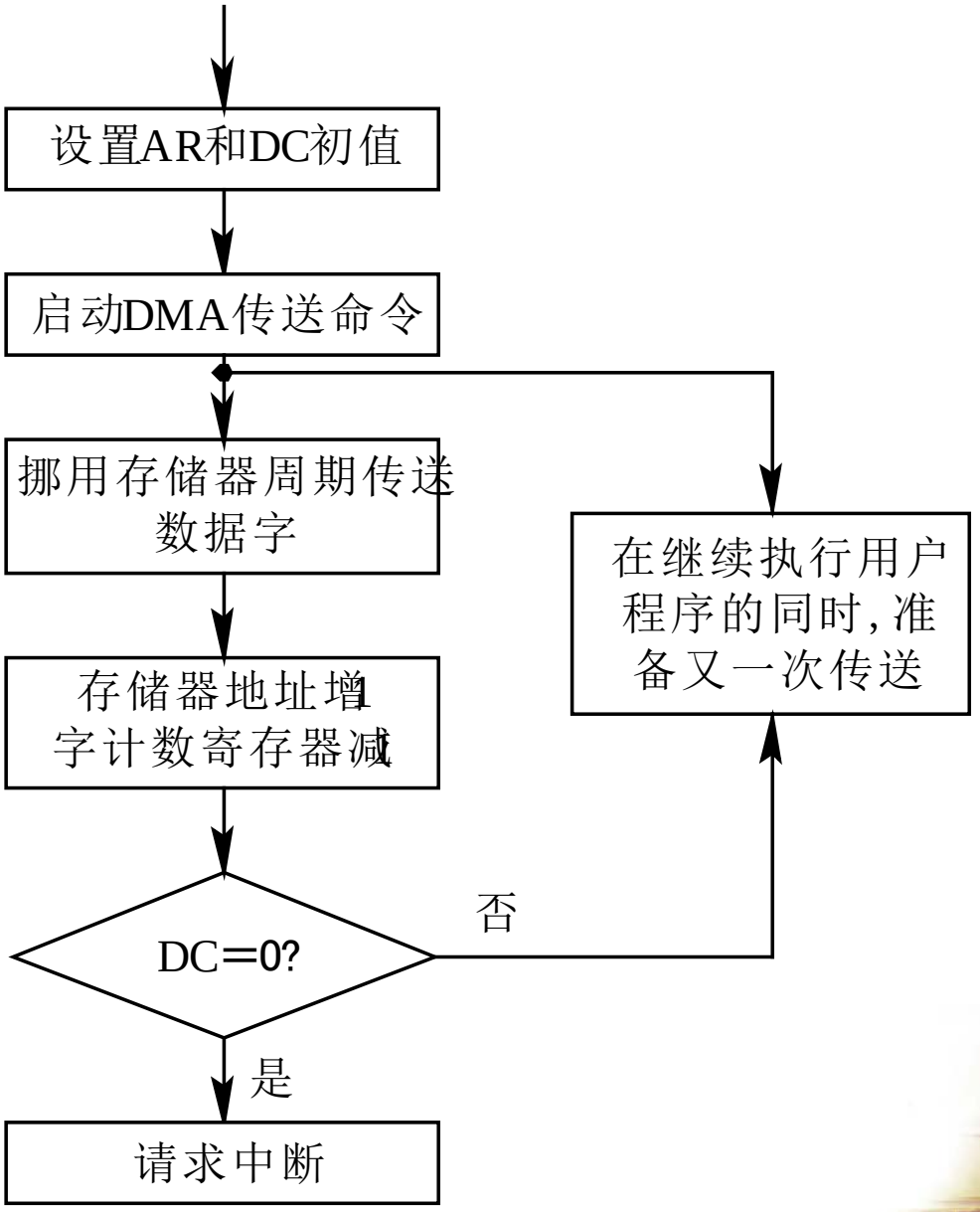


图 5-9 DMA方式的工作流程

5.2.4 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(或写)为单位的干预，减少为对一组数据块的读(或写)及有关的控制和管理为单位的干预。同时，又可实现CPU、通道和I/O设备三者的并行操作，从而更有效地提高整个系统的资源利用率。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。

2. 通道程序

(1) 操作码。

(2) 内存地址。

(3) 计数。

(4) 通道程序结束位P。

(5) 记录结束标志R。

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 缓 冲 管 理

5.3.1 缓冲的引入

(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。

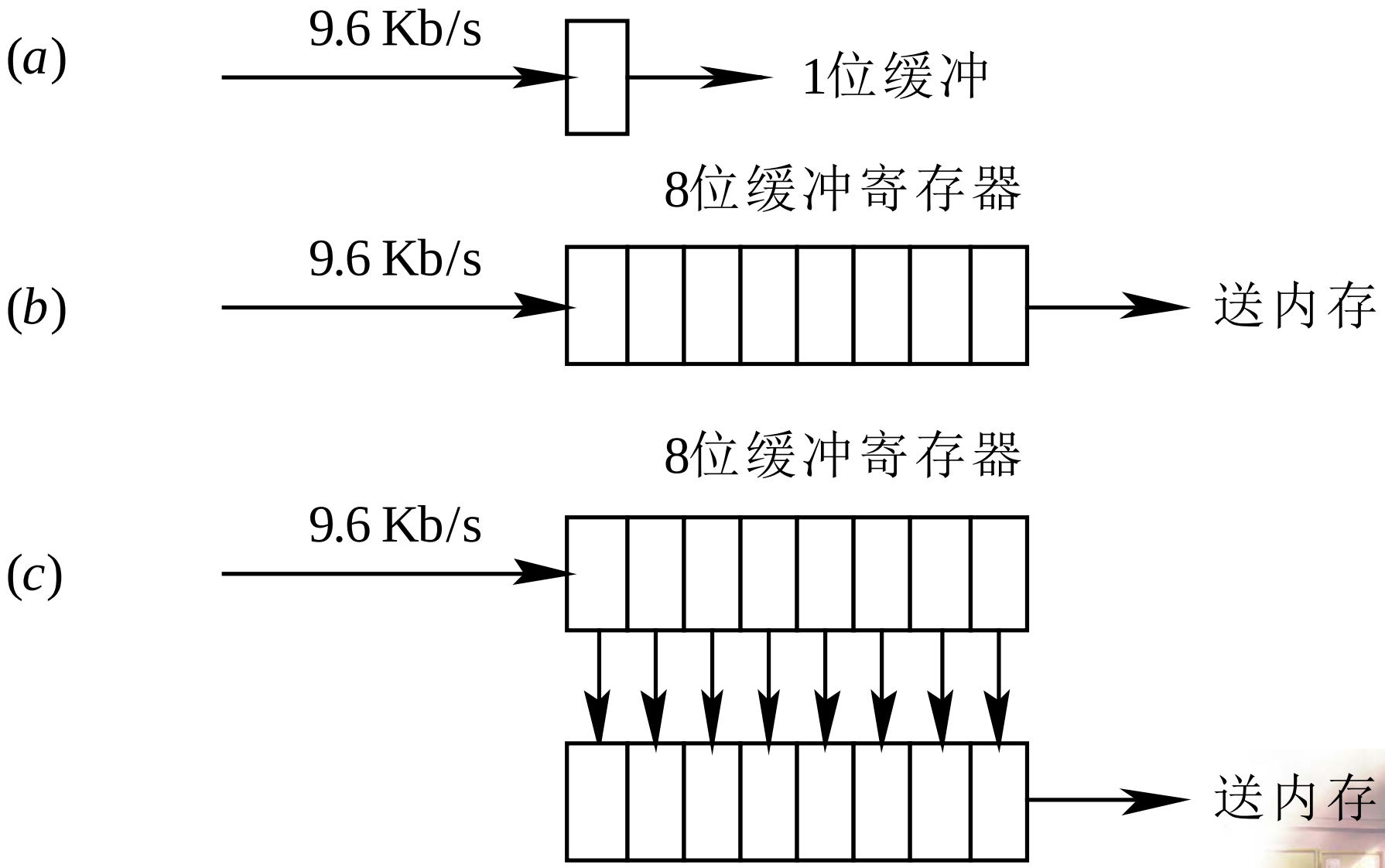


图 5-10 利用缓冲寄存器实现缓冲

5.3.2 单缓冲和双缓冲

1. 单缓冲(Single Buffer)

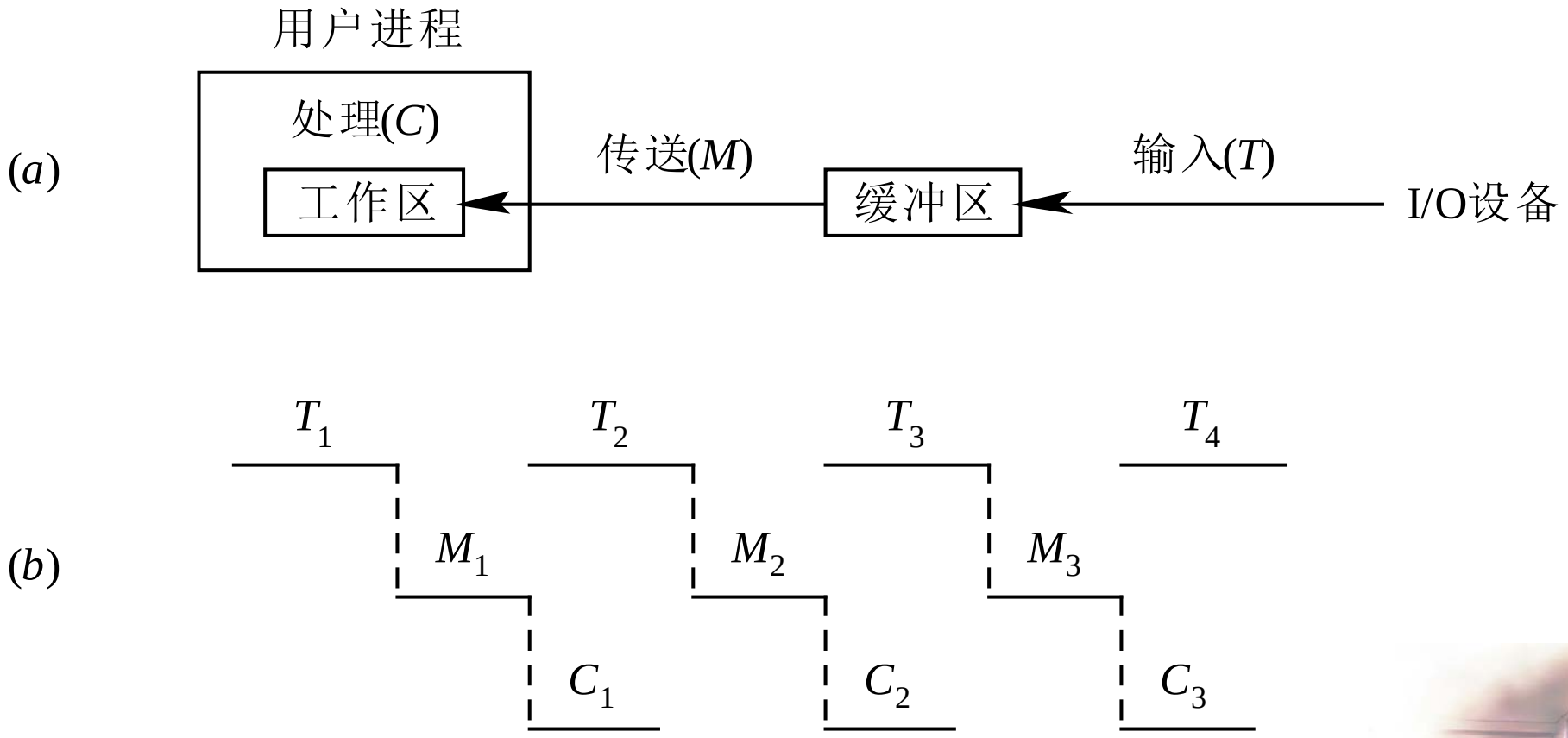
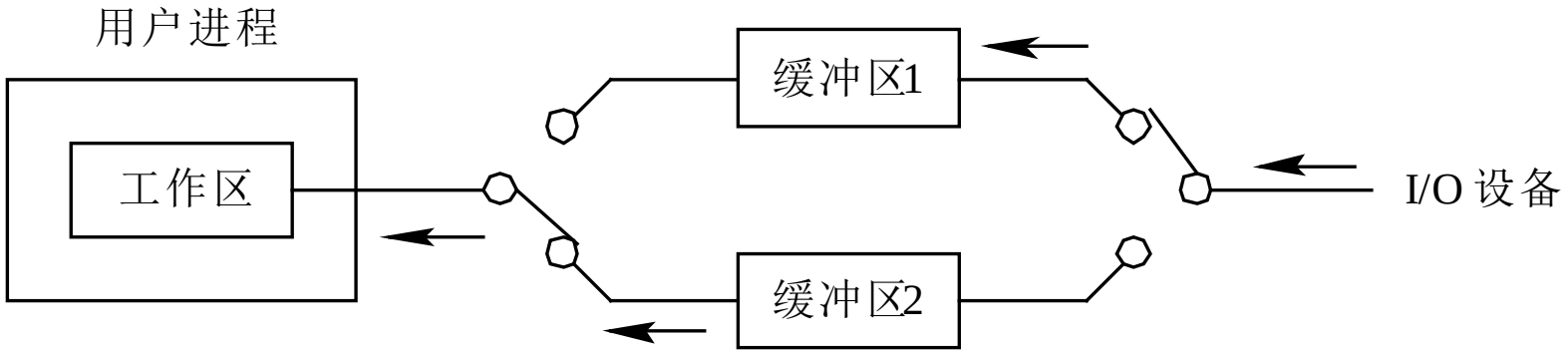


图 5-11 单缓冲工作示意图

2. 双缓冲(Double Buffer)

(*a*)



(*b*)

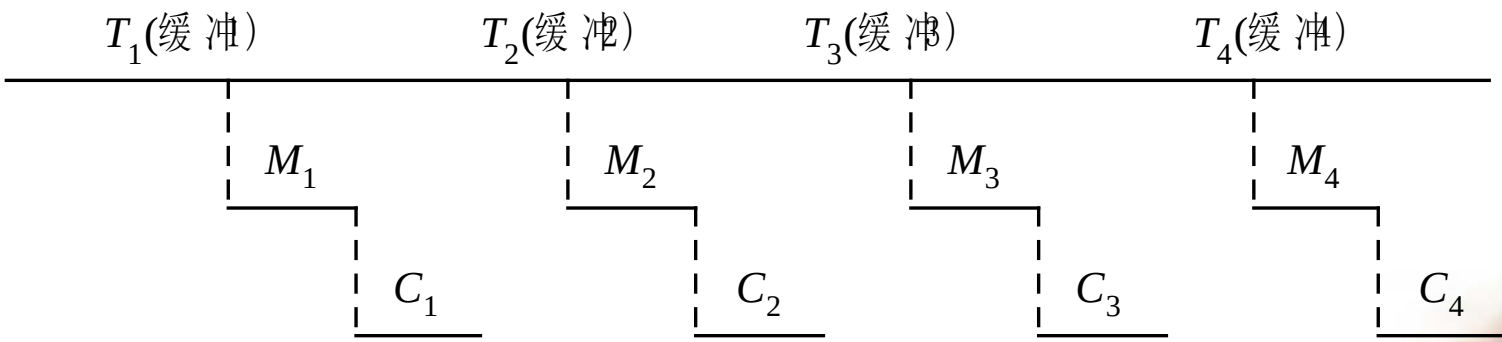
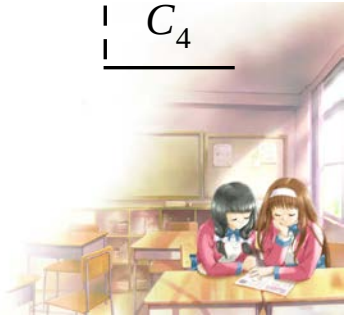
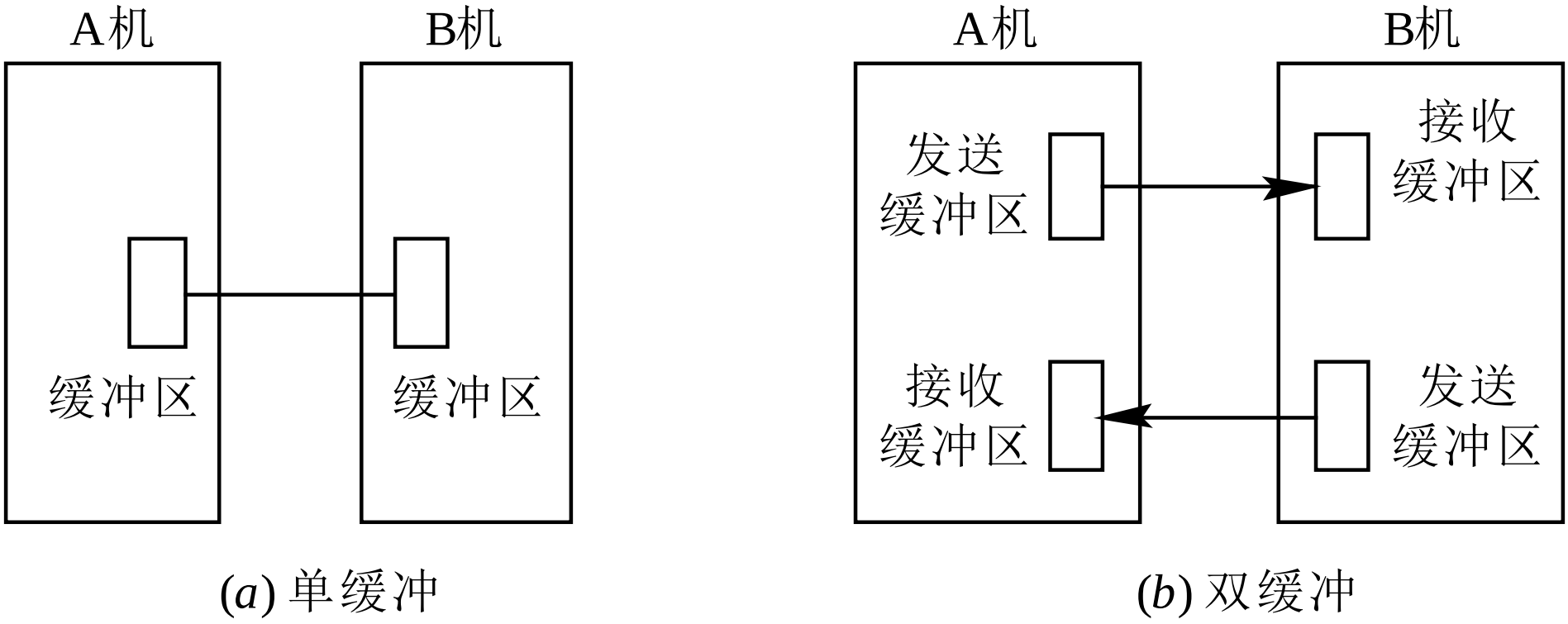


图 5-12 双缓冲工作示意图





(*a*) 单缓冲　　(*b*) 双缓冲

图 5-13 双机通信时缓冲区的设置

5.3.3 循环缓冲

1. 循环缓冲的组成

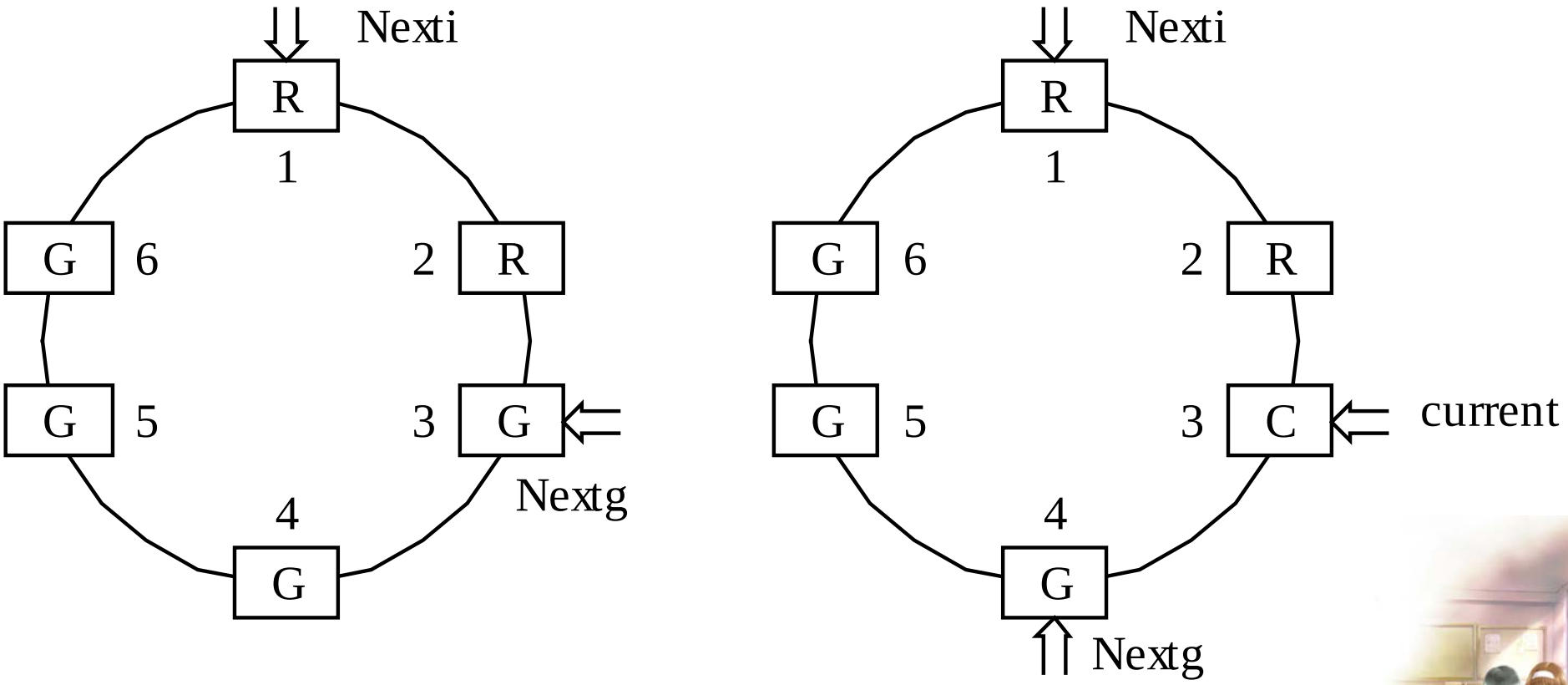


图 5-14 循环缓冲

2. 循环缓冲区的使用

(1) Getbuf过程。

(2) Releasebuf过程。

3. 进程同步

(1) Nexti指针追赶上Nextg指针。

(2) Nextg指针追赶上Nexti指针。

5.3.4 缓冲池(Buffer Pool)

1. 缓冲池的组成

1. 缓冲池的组成

对于既可用于输入又可用于输出的公用缓冲池， 其中至少应含有以下三种类型的缓冲区：① 空(闲)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。 为了管理上的方便，可将相同类型的缓冲区链成一个队列，于是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。

2. Getbuf过程和Putbuf过程

```
Procedure Getbuf(type)
begin
 Wait(RS(type));
 Wait(MS(type));
 B(number) ：[KG-*3]= Takebuf(type);
 Signal(MS(type));
 end
 Procedure Putbuf(type, number)
begin
 Wait(MS(type));
 Addbuf(type, number);
 Signal(MS(type));
 Signal(RS(type));
end
```

3. 缓冲区的工作方式

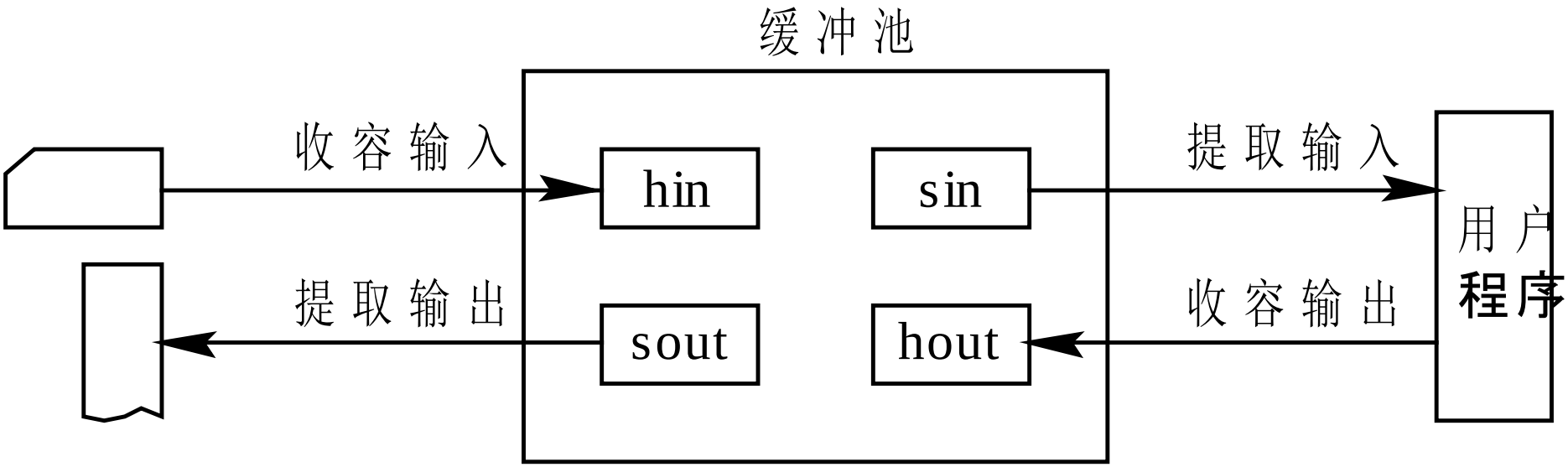


图 5-15 缓冲区的工作方式

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

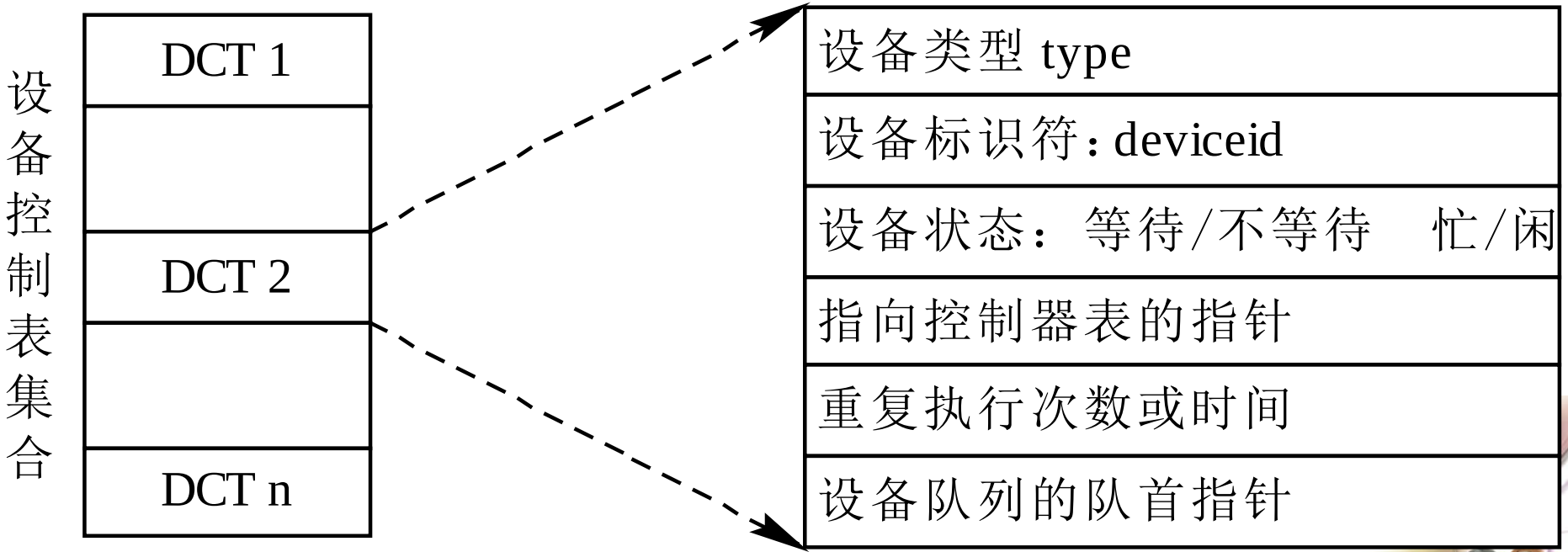


图 5-16 设备控制表

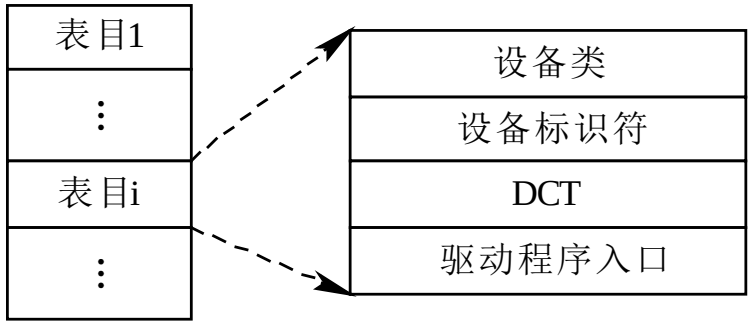
2. 控制器控制表、 通道控制表和系统设备表

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

图 5-17 COCT、 CHCT和SDT表

5.4.2 设备分配时应考虑的因素

1. 设备的固有属性

(1) 独享设备。

(2) 共享设备。

(3) 虚拟设备。

2. 设备分配算法

(1) 先来先服务。

(2) 优先级高者优先。

3. 设备分配中的安全性

1) 安全分配方式

2) 不安全分配方式

5.4.3 设备独立性

1. 设备独立性(Device Independence)的概念

为了提高OS的可适应性和可扩展性，在现代OS中都毫无例外地实现了设备独立性，也称为设备无关性。 其基本含义是：应用程序独立于具体使用的物理设备。为了实现设备独立性而引入了逻辑设备和物理设备这两个概念。在应用程序中，使用逻辑设备名称来请求使用某类设备；而系统在实际执行时， 还必须使用物理设备名称。因此，系统须具有将逻辑设备名称转换为某物理设备名称的功能，这非常类似于存储器管理中所介绍的逻辑地址和物理地址的概念。

在实现了设备独立性的功能后， 可带来以下两方面的好处。

1) 设备分配时的灵活性

2) 易于实现I/O重定向

2. 设备独立性软件

1) 执行所有设备的公有操作

这些公有操作包括：① 对独立设备的分配与回收； ② 将逻辑设备名映射为物理设备名，进一步可以找到相应物理设备的驱动程序； ③ 对设备进行保护，禁止用户直接访问设备； ④ 缓冲管理，即对字符设备和块设备的缓冲区进行有效的管理， 以提高I/O的效率；⑤ 差错控制。由于在I/O操作中的绝大多数错误都与设备无关，故主要由设备驱动程序处理，而设备独立性软件只处理那些设备驱动程序无法处理的错误。

2) 向用户层(或文件层)软件提供统一接口

无论何种设备， 它们向用户所提供的接口应该是相同的。 例如， 对各种设备的读操作，在应用程序中都使用read; 而对各种设备的写操作，也都使用write。

3. 逻辑设备名到物理设备名映射的实现

1) 逻辑设备表

2) LUT的设置问题

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*)

图 5-18 逻辑设备表

5.4.4 独占设备的分配程序

1. 基本的设备分配程序

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

1) 增加设备的独立性

2) 考虑多通路情况

5.4.5 SPOOLing技术

1. 什么是SPOOLing

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， 将低速I/O设备上的数据传送到高速磁盘上；或者相反。事实上，当系统中引入了多道程序技术后，完全可以利用其中的一道程序，来模拟脱机输入时的外围控制机功能，把低速I/O设备上的数据传送到高速磁盘上；再用另一道程序来模拟脱机输出时外围控制机的功能，把数据从磁盘传送到低速输出设备上。这样，便可在主机的直接控制下，实现脱机输入、 输出功能。 此时的外围操作与CPU对数据的处理同时进行， 我们把这种在联机情况下实现的同时外围操作称为SPOOLing(Simultaneaus Periphernal Operating On-Line)，或称为假脱机操作。

2. SPOOLing系统的组成

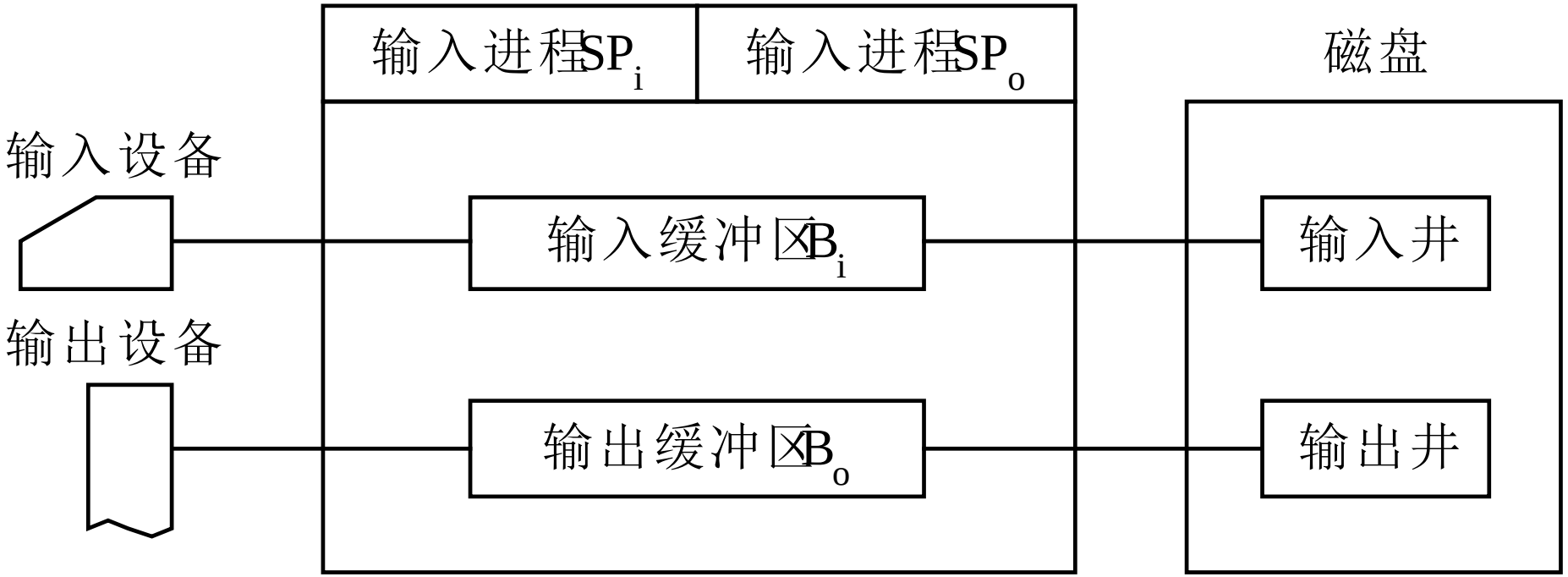


图 5-19 SPOOLing系统的组成

3. 共享打印机

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。当用户进程请求打印输出时， SPOOLing系统同意为它打印输出， 但并不真正立即把打印机分配给该用户进程， 而只为它做两件事： ① 由输出进程在输出井中为之申请一个空闲磁盘块区， 并将要打印的数据送入其中； ② 输出进程再为用户进程申请一张空白的用户请求打印表，并将用户的打印要求填入其中， 再将该表挂到请求打印队列上。

4. 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。

5.5 设 备 处 理

5.5.1 设备驱动程序的功能和特点

1. 设备驱动程序的功能

(1) 接收由I/O进程发来的命令和参数， 并将命令中的抽象要求转换为具体要求，例如，将磁盘块号转换为磁盘的盘面、 磁道号及扇区号。

(2) 检查用户I/O请求的合法性，了解I/O设备的状态，传递有关参数，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设备去完成指定的I/O操作；如果设备处于忙碌状态，则将请求者的请求块挂在设备队列上等待。

(4) 及时响应由控制器或通道发来的中断请求，并根据其中断类型调用相应的中断处理程序进行处理。

(5) 对于设置有通道的计算机系统，驱动程序还应能够根据用户的I/O请求，自动地构成通道程序。

2. 设备处理方式

(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。

(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。

3. 设备驱动程序的特点

(1) 驱动程序主要是指在请求I/O的进程与设备控制器之间的一个通信和转换程序。

(2) 驱动程序与设备控制器和I/O设备的硬件特性紧密相关，因而对不同类型的设备应配置不同的驱动程序。

(3) 驱动程序与I/O设备所采用的I/O控制方式紧密相关。

(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。

5.5.2 设备驱动程序的处理过程

1. 将抽象要求转换为具体要求

2. 检查I/O请求的合法性

3. 读出和检查设备的状态

4. 传送必要的参数

5. 工作方式的设置

6. 启动I/O设备

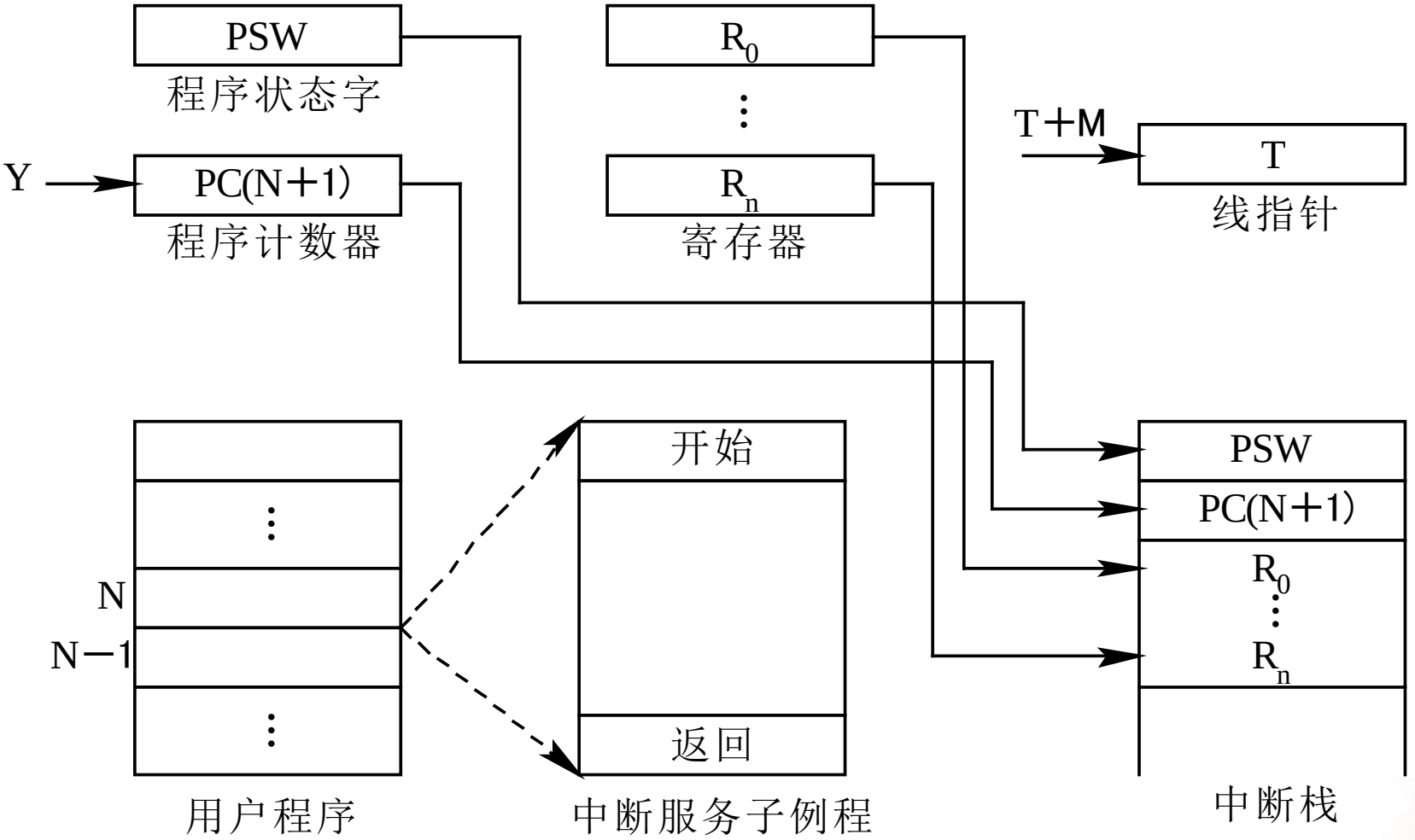


图 5-20 中断现场保护示意图

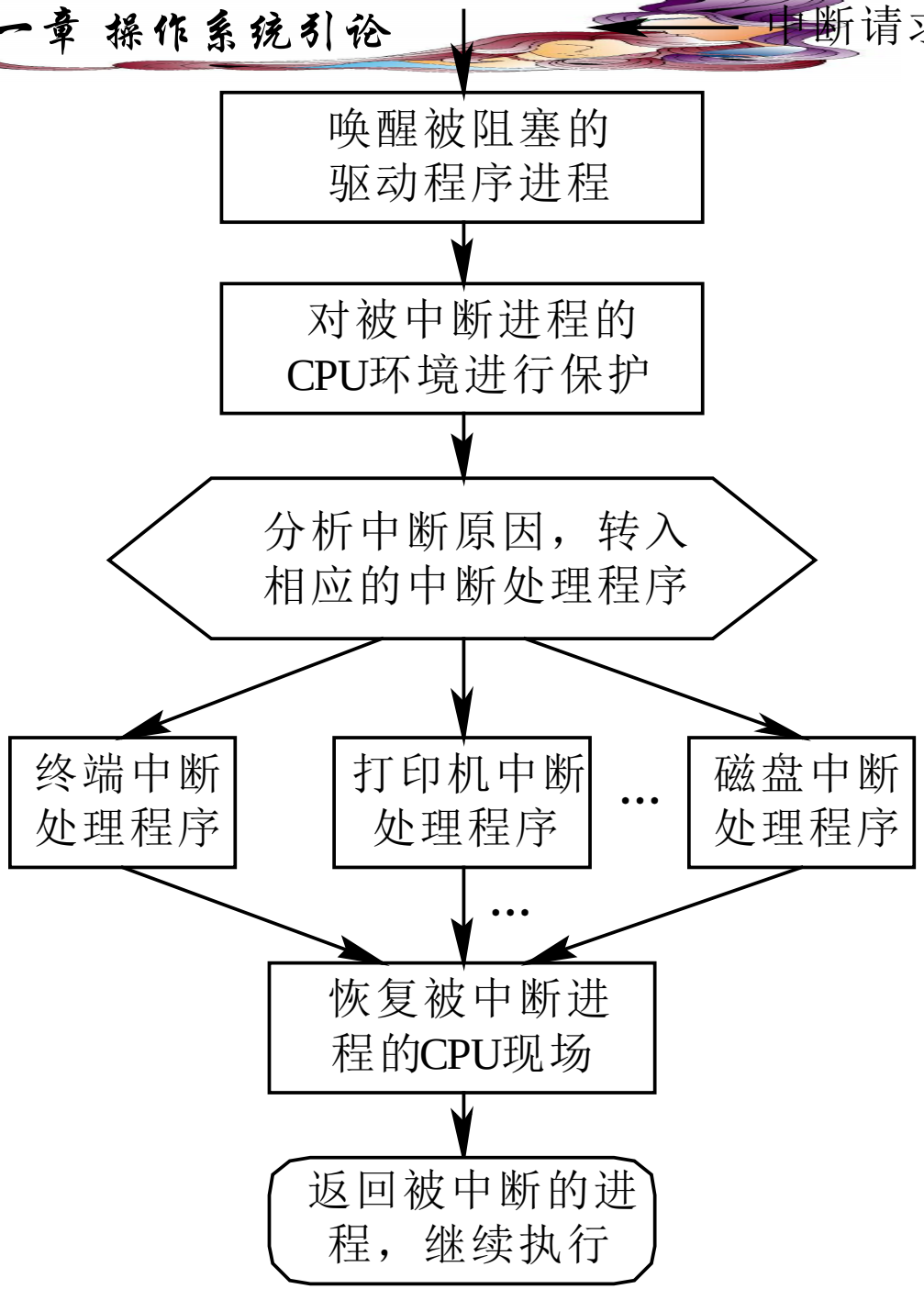

中断请求信号
唤醒被阻塞的
驱动程序进程
对被中断进程的
CPU环境进行保护
分析中断原因，转入
相应的中断处理程序
终端中断
处理程序
打印机中断
处理程序
…
磁盘中断
处理程序
…
恢复被中断进
程的CPU现场
返回被中断的进
程，继续执行

图 5-21 中断处理流程

5.6 磁盘存储器管理

5.6.1 磁盘性能简述

1. 数据的组织和格式

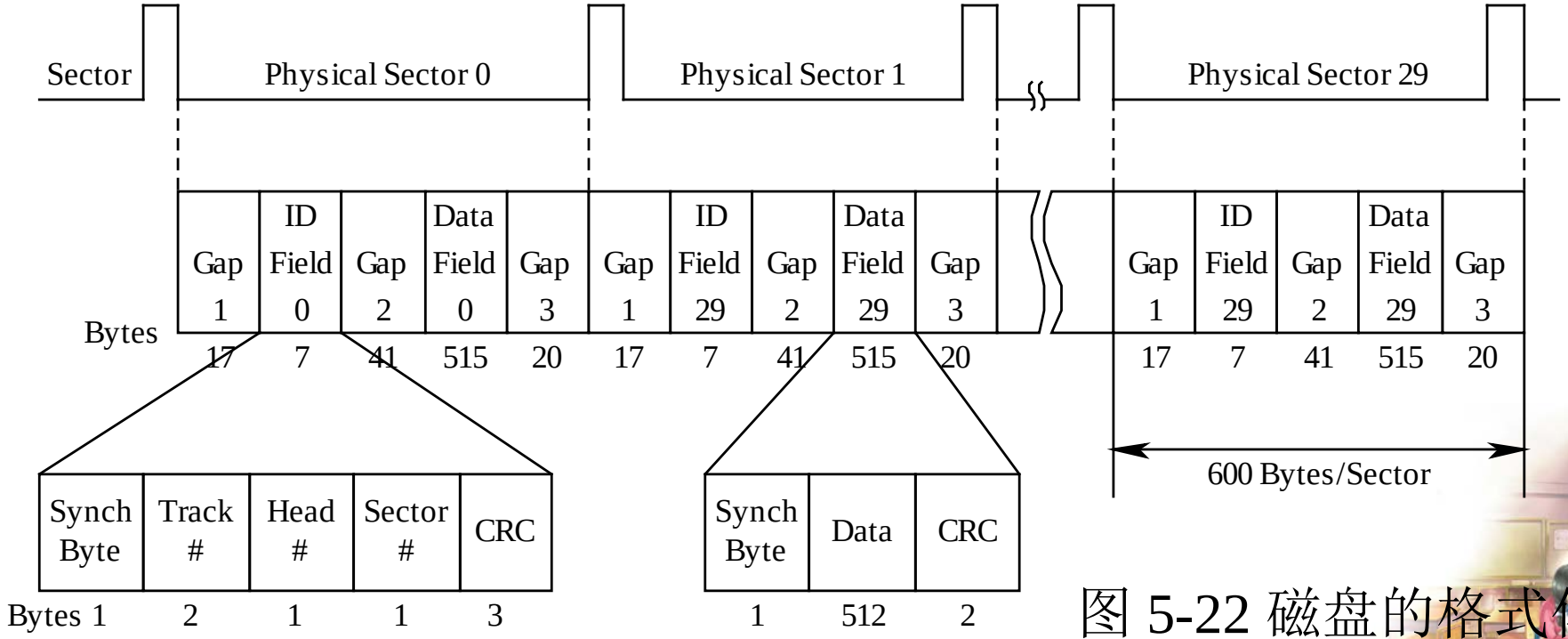


图 5-22 磁盘的格式化

2. 磁盘的类型

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，对一般磁盘，m=0.2；对高速磁盘，m≤0.1，磁臂的启动时间约为2 ms。 这样，对一般的温盘， 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_τ

这是指定扇区移动到磁头下面所经历的时间。对于硬盘，典型的旋转速度大多为5400 r/min，每转需时11.1 ms，平均旋转延迟时间T_τ为5.55 ms；对于软盘，其旋转速度为300 r/min或600 r/min，这样，平均T_τ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_τ相同， 因此， 可将访问时间T_a表示为：

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

5.6.2 磁盘调度

1. 先来先服务FCFS(First-Come, First Served)

（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 5-23 FCFS调度算法

2. 最短寻道时间优先SSTF(Shortest Seek Time First)

（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 5-24 SSTF调度算法

3. 扫描(SCAN)算法

1) 进程“饥饿”现象

SSTF算法虽然能获得较好的寻道性能， 但却可能导致某个进程发生“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请求必须优先满足。对SSTF算法略加修改后所形成的SCAN算法， 即可防止老进程出现“饥饿”现象。

2) SCAN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 5-25 SCAN调度算法示例

4. 循环扫描(CSCAN)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
18	166
38	20
39	1
55	16
58	3
90	32
平均寻道长度：27.5	

图 5-26 CSCAN调度算法示例

5. N-Step-SCAN和FSCAN调度算法

1) N-Step-SCAN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成若干个长度为N的子队列，磁盘调度将按FCFS算法依次处理这些子队列。 而每处理一个队列时又是按SCAN算法，对一个队列处理完后，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，便将新请求进程放入其他队列，这样就可避免出现粘着现象。 当N值取得很大时，会使N步扫描法的性能接近于SCAN算法的性能； 当N=1时， N步SCAN算法便蜕化为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。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。在扫描期间，将新出现的所有请求磁盘I/O的进程，放入另一个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。

5.6.3 磁盘高速缓存(Disk Cache)

1. 磁盘高速缓存的形式

是指利用内存中的存储空间，来暂存从磁盘中读出的一系列盘块中的信息。因此，这里的高速缓存是一组在逻辑上属于磁盘，而物理上是驻留在内存中的盘块。高速缓存在内存中可分成两种形式。第一种是在内存中开辟一个单独的存储空间来作为磁盘高速缓存，其大小是固定的，不会受应用程序多少的影响；第二种是把所有未利用的内存空间变为一个缓冲池，供请求分页系统和磁盘I/O时(作为磁盘高速缓存)共享。此时高速缓存的大小，显然不再是固定的。当磁盘I/O的频繁程度较高时，该缓冲池可能包含更多的内存空间；而在应用程序运行得较多时，该缓冲池可能只剩下较少的内存空间。

2. 数据交付方式

系统可以采取两种方式， 将数据交付给请求进程：

(1) 数据交付。这是直接将高速缓存中的数据， 传送到请求者进程的内存工作区中。

(2) 指针交付。只将指向高速缓存中某区域的指针， 交付给请求者进程。

后一种方式由于所传送的数据量少，因而节省了数据从磁盘高速缓存存储空间到进程的内存工作区的时

3. 置换算法

由于请求调页中的联想存储器与高速缓存(磁盘I/O中)的工作情况不同，因而使得在置换算法中所应考虑的问题也有所差异。因此，现在不少系统在设计其高速缓存的置换算法时，除了考虑到最近最久未使用这一原则外， 还考虑了以下几点：

(1) 访问频率。

(2) 可预见性。

(3) 数据的一致性。

4. 周期性地写回磁盘

在UNIX系统中专门增设了一个修改(update)程序， 使之在后台运行，该程序周期性地调用一个系统调用SYNC。该调用的主要功能是强制性地将所有在高速缓存中已修改的盘块数据写回磁盘。一般是把两次调用SYNC的时间间隔定为30 s。这样，因系统故障所造成的工作损失不会超过30 s的劳动量。而在MS-DOS中所采用的方法是：只要高速缓存中的某盘块数据被修改，便立即将它写回磁盘，并将这种高速缓存称为“写穿透、高速缓存”(write-through cache)。MS-DOS所采用的写回方式，几乎不会造成数据的丢失，但须频繁地启动磁盘。

5.6.4 提高磁盘I/O速度的其它方法

1. 提前读(Read-Ahead)

2. 延迟写

3. 优化物理块的分布

4. 虚拟盘

5.6.5 廉价磁盘冗余阵列

1. 并行交叉存取

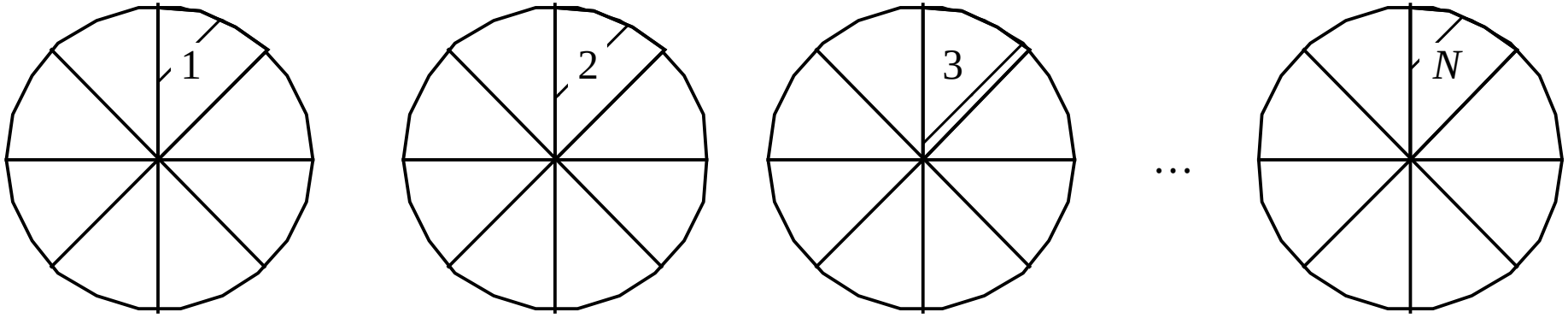


图 5-27 磁盘并行交叉存取方式

2. RAID的分级

(1) RAID 0级。

(2) RAID 1级。

(3) RAID 3级。

(4) RAID 5级。

(5) RAID 6级和RAID 7级。

3. RAID的优点

(1) 可靠性高。

(2) 磁盘I/O速度高。

(3) 性能/价格比高。

第六章 文 件 管 理

6.1 文件和文件系统

6.1.1 文件、记录和数据项

1. 数据项

(1) 基本数据项。这是用于描述一个对象的某种属性的字符集，是数据组织中可以命名的最小逻辑数据单位， 即原子数据，又称为数据元素或字段。它的命名往往与其属性一致。例如，用于描述一个学生的基本数据项有： 学号、姓名、 年龄、 所在班级等。

(2) 组合数据项。它是由若干个基本数据项组成的，简称组项。例如，经理便是个组项，它由正经理和副经理两个基本项组成。又如，工资也是个组项，它可由基本工资、工龄工资和奖励工资等基本项所组成。

基本数据项除了数据名外，还应有数据类型。因为基本项仅是描述某个对象的属性，根据属性的不同，需要用不同的数据类型来描述。例如，在描述学生的学号时，应使用整数；描述学生的姓名则应使用字符串(含汉字)；描述性别时，可用逻辑变量或汉字。可见，由数据项的名字和类型两者共同定义了一个数据项的“型”。 而表征一个实体在数据项上的数据则称为“值”。例如，学号/30211、姓名/王有年、性别/男等。

2. 记录

记录是一组相关数据项的集合，用于描述一个对象在某方面的属性。一个记录应包含哪些数据项，取决于需要描述对象的哪个方面。而一个对象，由于他所处的环境不同可把他作为不同的对象。 例如，一个学生，当把他作为班上的一名学生时， 对他的描述应使用学号、姓名、年龄及所在系班，也可能还包括他所学过的课程的名称、 成绩等数据项。但若把学生作为一个医疗对象时，对他描述的数据项则应使用诸如病历号、 姓名、 性别、 出生年月、 身高、 体重、血压及病史等项。

3. 文件

文件是指由创建者所定义的、 具有文件名的一组相关元素的集合，可分为有结构文件和无结构文件两种。 在有结构的文件中，文件由若干个相关记录组成；而无结构文件则被看成是一个字符流。文件在文件系统中是一个最大的数据单位，它描述了一个对象集。例如，可以将一个班的学生记录作为一个文件。一个文件必须要有一个文件名，它通常是由一串ASCII码或(和)汉字构成，名字的长度因系统不同而异。如在有的系统中把名字规定为8个字符，而在有的系统中又规定可用14个字符。

属性可以包括：

(1) 文件类型。

(2) 文件长度。

(3) 文件的物理位置。

(4) 文件的建立时间。

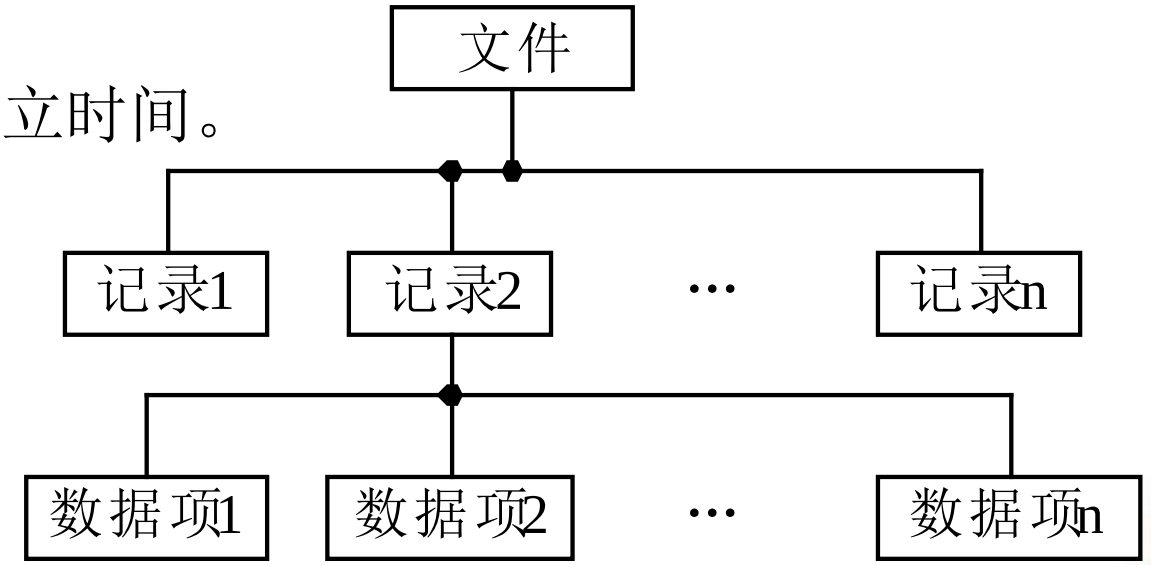


图 6-1 文件、 记录和数据项之间的层次关系

6.1.2 文件类型和文件系统模型

1. 文件类型

1) 按用途分类

(1) 系统文件。

(2) 用户文件。

(3) 库文件。

2) 按文件中数据的形式分类

(1) 源文件。

(2) 目标文件。

(3) 可执行文件。

3) 按存取控制属性分类

(1) 只执行文件。

(2) 只读文件。

(3) 读写文件。

2. 文件系统模型

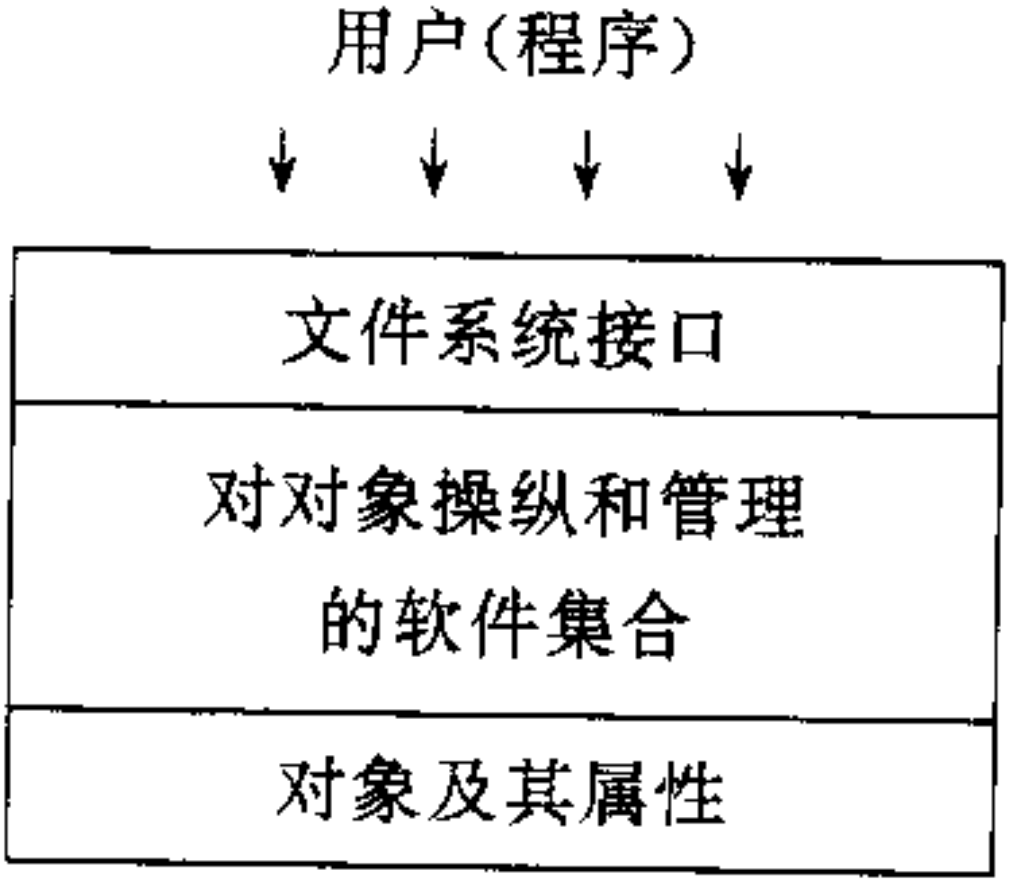


图 6-2 文件系统模型

1) 对象及其属性

文件管理系统管理的对象有： ① 文件。 它作为文件管理的直接对象。 ② 目录。为了方便用户对文件的存取和检索，在文件系统中必须配置目录。对目录的组织和管理是方便用户和提高对文件存取速度的关键。③ 磁盘(磁带)存储空间。 文件和目录必定占用存储空间，对这部分空间的有效管理，不仅能提高外存的利用率，而且能提高对文件的存取速度。

2) 对对象操纵和管理的软件集合

这是文件管理系统的核心部分。文件系统的功能大多是在这一层实现的，其中包括：对文件存储空间的管理、对文件目录的管理、用于将文件的逻辑地址转换为物理地址的机制、对文件读和写的管理，以及对文件的共享与保护等功能。

3) 文件系统的接口

为方便用户使用文件系统，文件系统通常向用户提供两种类型的接口：

(1) 命令接口。这是指作为用户与文件系统交互的接口。用户可通过键盘终端键入命令，取得文件系统的服务。

(2) 程序接口。这是指作为用户程序与文件系统的接口。用户程序可通过系统调用来取得文件系统的服务。

6.1.3 文件操作

(1) 创建文件。

(2) 删除文件。

(3) 读文件。

(4) 写文件。

(5) 截断文件。

(6) 设置文件的读/写位置。

2. 文件的“打开”和“关闭”操作

所谓“打开”，是指系统将指名文件的属性(包括该文件在外存上的物理位置)从外存拷贝到内存打开文件表的一个表目中，并将该表目的编号(或称为索引)返回给用户。以后， 当用户再要求对该文件进行相应的操作时，便可利用系统所返回的索引号向系统提出操作请求。系统这时便可直接利用该索引号到打开文件表中去查找，从而避免了对该文件的再次检索。这样不仅节省了大量的检索开销，也显著地提高了对文件的操作速度。如果用户已不再需要对该文件实施相应的操作时，可利用“关闭”(close)系统调用来关闭此文件，OS将会把该文件从打开文件表中的表目上删除掉。

3. 其它文件操作

为了方便用户使用文件，通常，OS都提供了数条有关文件操作的系统调用，可将这些调用分成若干类：最常用的一类是有关对文件属性进行操作的，即允许用户直接设置和获得文件的属性，如改变已存文件的文件名、改变文件的拥有者(文件主)、改变对文件的访问权，以及查询文件的状态(包括文件类型、大小和拥有者以及对文件的访问权等)；另一类是有关目录的，如创建一个目录，删除一个目录，改变当前目录和工作目录等；此外，还有用于实现文件共享的系统调用和用于对文件系统进行操作的系统调用等。

6.2 文件的逻辑结构

对于任何一个文件，都存在着以下两种形式的结构：

（1）文件的逻辑结构(File Logical Structure)。

(2) 文件的物理结构， 又称为文件的存储结构， 是指文件在外存上的存储组织形式。

6.2.1 文件逻辑结构的类型

1. 有结构文件

(1) 定长记录。

(2) 变长记录。

(1) 顺序文件。

(2) 索引文件。

(3) 索引顺序文件。

2. 无结构文件

如果说大量的数据结构和数据库，是采用有结构的文件形式的话，则大量的源程序、 可执行文件、 库函数等，所采用的就是无结构的文件形式，即流式文件。 其长度以字节为单位。对流式文件的访问，则是采用读写指针来指出下一个要访问的字符。可以把流式文件看作是记录式文件的一个特例。在UNIX系统中，所有的文件都被看作是流式文件；即使是有结构文件，也被视为流式文件；系统不对文件进行格式处理。

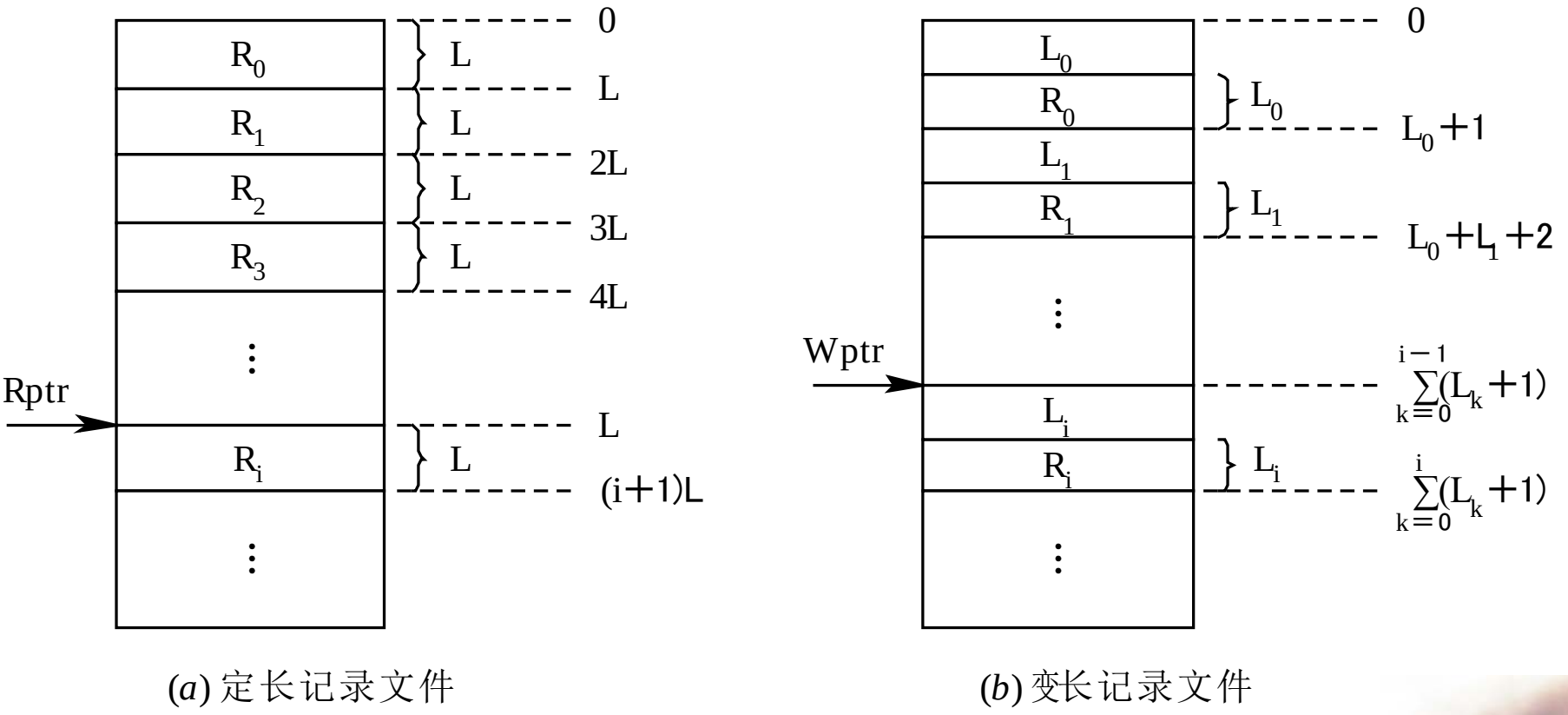
6.2.2 顺序文件

1. 逻辑记录的排序

第一种是串结构， 各记录之间的顺序与关键字无关。通常的办法是由时间来决定，即按存入时间的先后排列，最先存入的记录作为第一个记录，其次存入的为第二个记录， ..... 依此类推。

第二种情况是顺序结构，指文件中的所有记录按关键字(词)排列。可以按关键词的长短从小到大排序，也可以从大到小排序；或按其英文字母顺序排序。

2. 对顺序文件(Sequential File)的读/写操作



(*a*) 定长记录文件　　(*b*) 变长记录文件

图 6-3 定长和变长记录文件

3. 顺序文件的优缺点

顺序文件的最佳应用场合，是在对诸记录进行批量存取时，即每次要读或写一大批记录。此时，对顺序文件的存取效率是所有逻辑文件中最高的；此外，也只有顺序文件才能存储在磁带上， 并能有效地工作。

在交互应用的场合，如果用户(程序)要求查找或修改单个记录，为此系统便要去逐个地查找诸记录。 这时， 顺序文件所表现出来的性能就可能很差， 尤其是当文件较大时， 情况更为严重。例如，有一个含有10^4个记录的顺序文件，如果对它采用顺序查找法去查找一个指定的记录，则平均需要查找5×10^3个记录；如果是可变长记录的顺序文件，则为查找一个记录所需付出的开销将更大，这就限制了顺序文件的长度。

顺序文件的另一个缺点是， 如果想增加或删除一个记录， 都比较困难。 为了解决这一问题， 可以为顺序文件配置一个运行记录文件(Log File)或称为事务文件(Transaction File)， 把试图增加、 删除或修改的信息记录于其中， 规定每隔一定时间， 例如4小时，将运行记录文件与原来的主文件加以合并， 产生一个按关键字排序的新文件。

6.2.3 索引文件

对于定长记录文件，如果要查找第i个记录， 可直接根据下式计算来获得第i个记录相对于第一个记录首址的地址：

$$A_i = i \times L$$

然而，对于可变长度记录的文件，要查找其第i个记录时，须首先计算出该记录的首地址。为此，须顺序地查找每个记录，从中获得相应记录的长度L_i，然后才能按下式计算出第i个记录的首址。假定在每个记录前用一个字节指明该记录的长度，则

$$A_i = \sum_{i=0}^{i-1} L_i + i$$

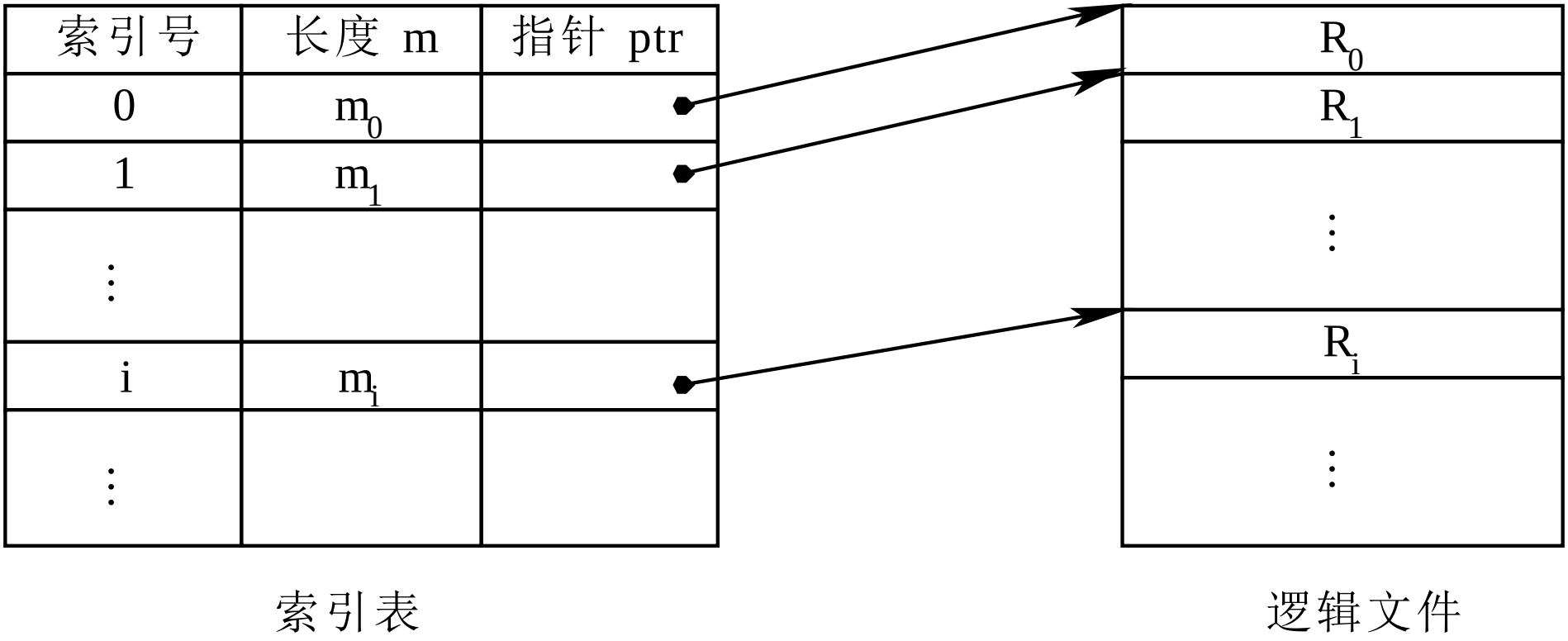


图 6-4 索引文件的组织

6.2.4 索引顺序文件

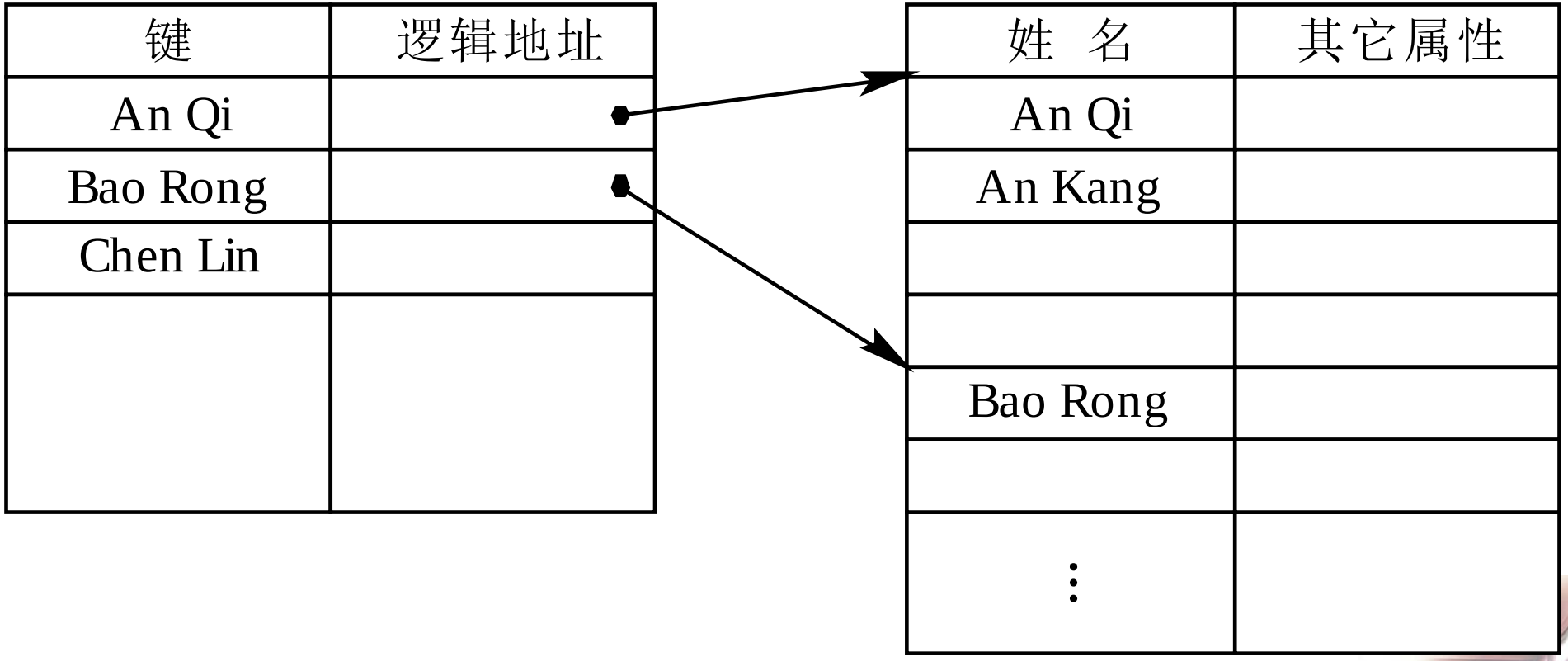


图 6-5 索引顺序文件

6.2.5 直接文件和哈希文件

1. 直接文件

对于直接文件，则可根据给定的记录键值，直接获得指定记录的物理地址。换言之，记录键值本身就决定了记录的物理地址。这种由记录键值到记录物理地址的转换被称为键值转换(Key to address transformation)。组织直接文件的关键，在于用什么方法进行从记录值到物理地址的转换。

2. 哈希(Hash)文件

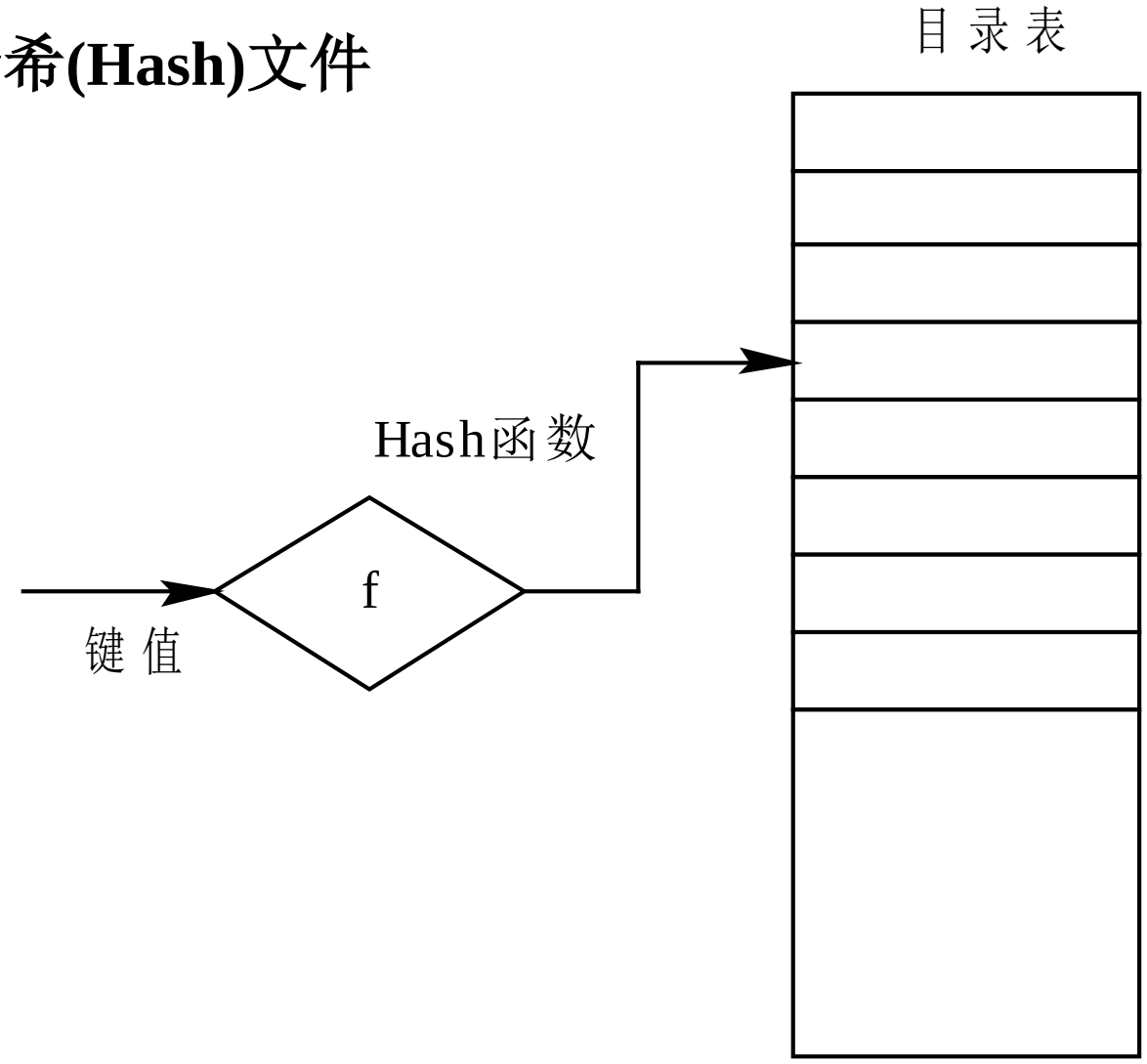
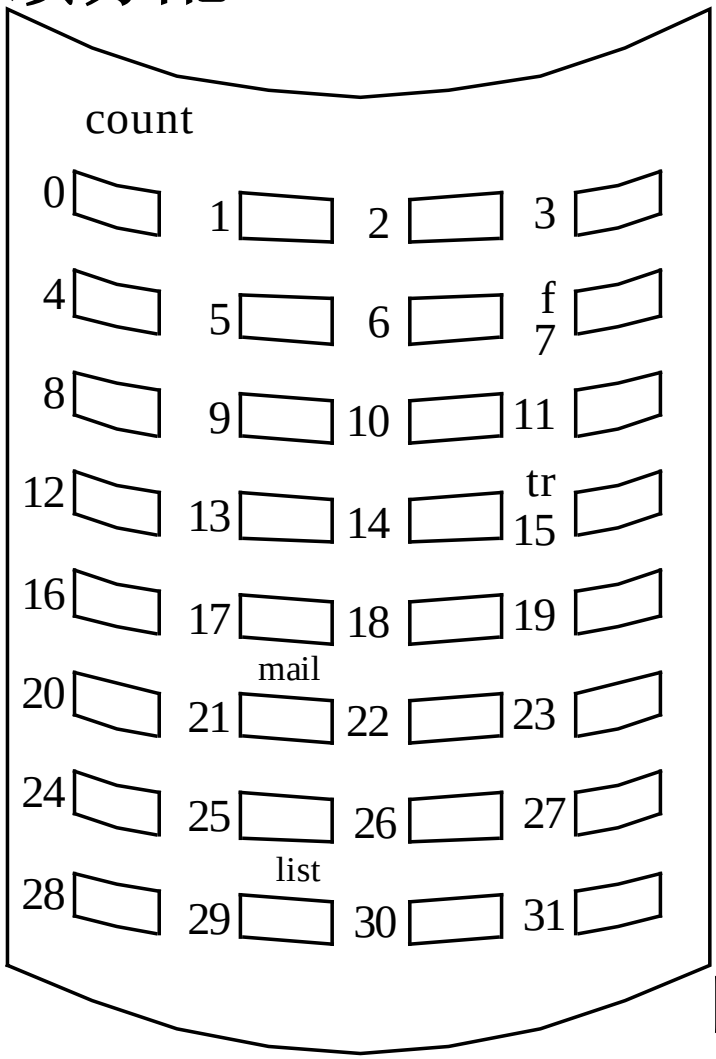


图 6-6 Hash文件的逻辑结构

6.3 外存分配方式

6.3.1 连续分配



目 录

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

图 6-7 磁盘空间的连续分配

2. 连续分配的主要优缺点

连续分配的主要优点如下：

(1) 顺序访问容易。

(2) 顺序访问速度快。

连续分配的主要缺点如下：

(1) 要求有连续的存储空间。

(2) 必须事先知道文件的长度。

6.3.2 链接分配

1. 隐式链接

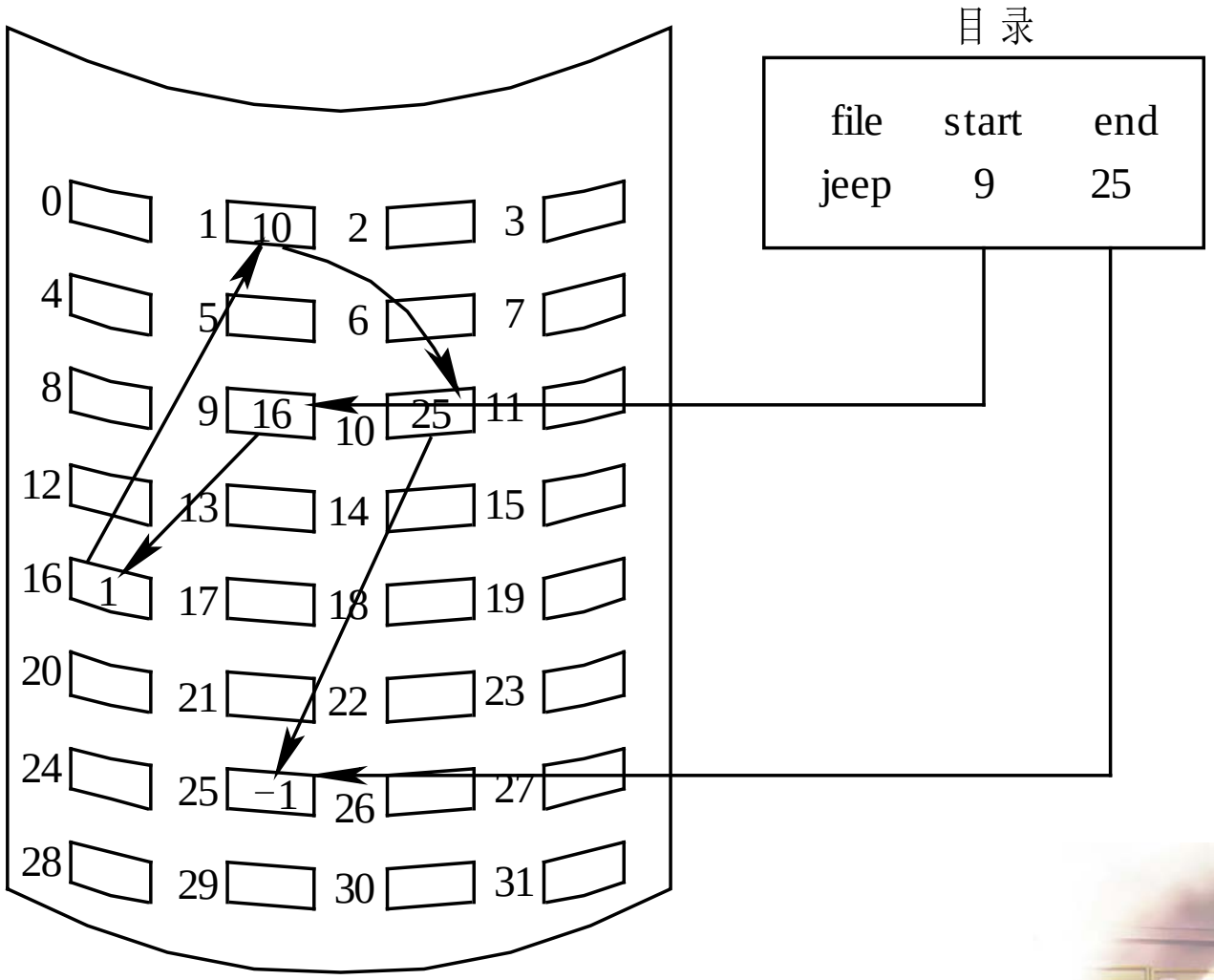


图 6-8 磁盘空间的链接式分配

2. 显式链接

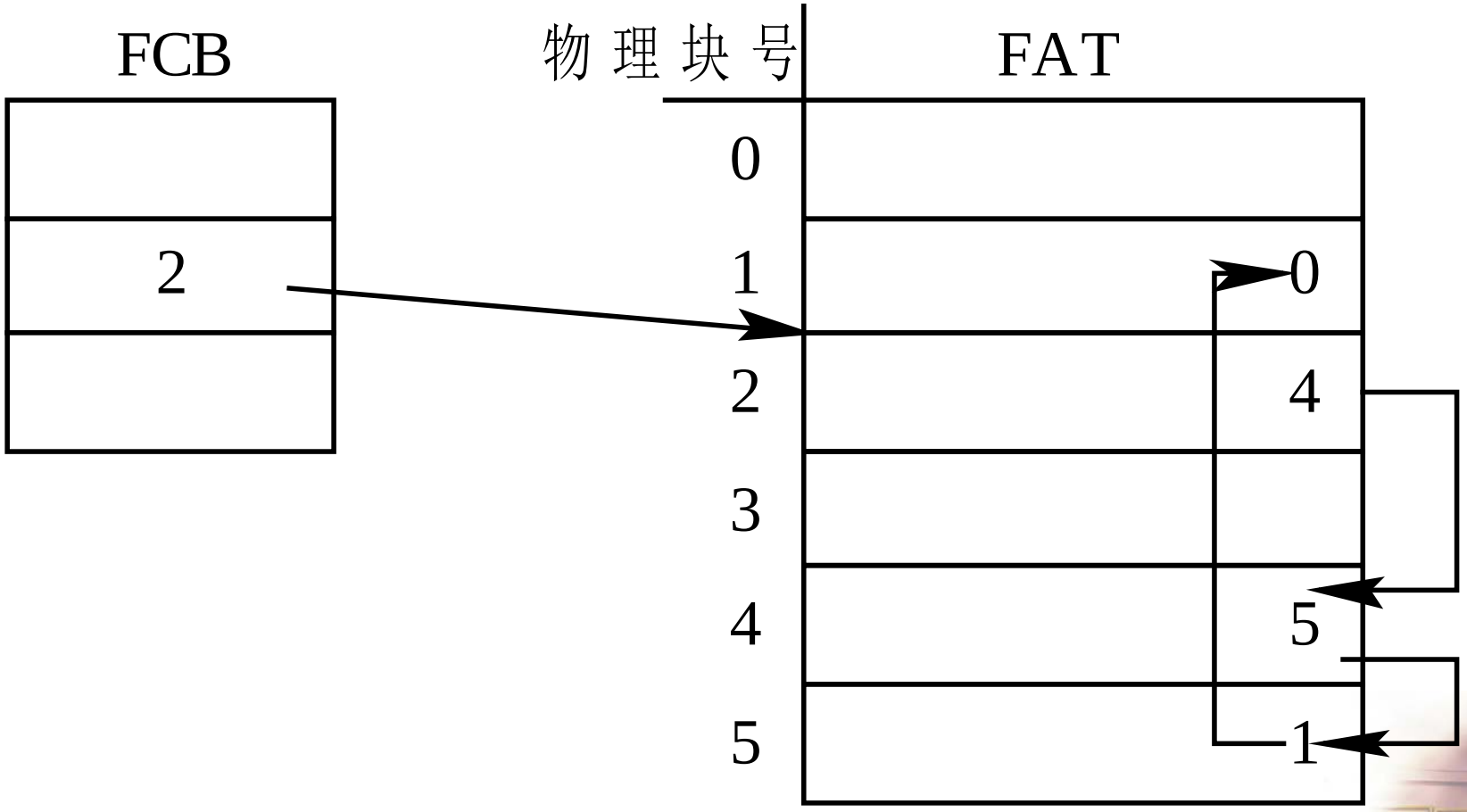


图 6-9 显式链接结构

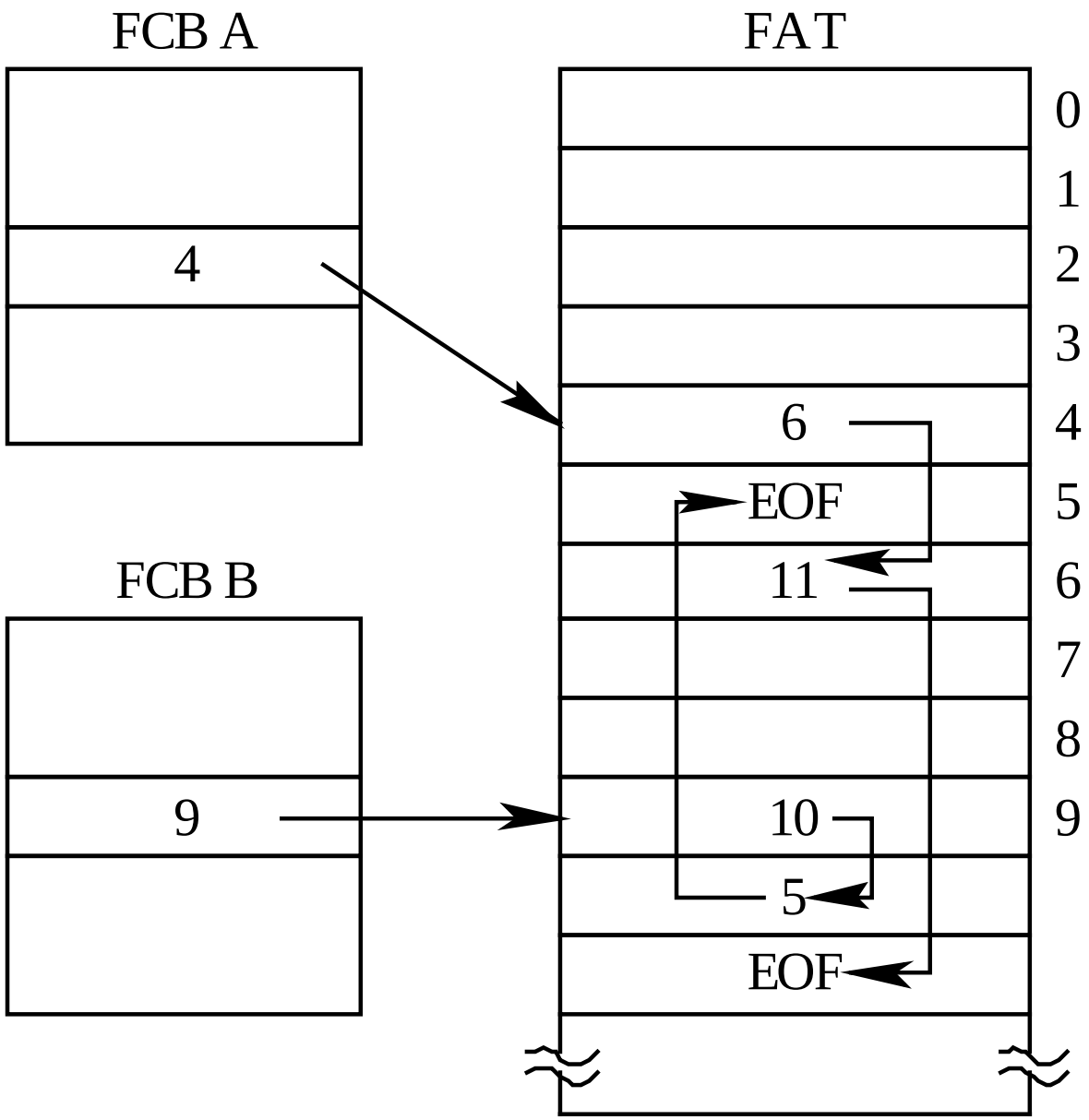


图 6-10 MS-DOS的文件物理结构

6.3.3 索引分配

1. 单级索引分配

链接分配方式虽然解决了连续分配方式所存在的问题，但又出现了另外两个问题， 即：

(1) 不能支持高效的直接存取。要对一个较大的文件进行直接存取，须首先在FAT中顺序地查找许多盘块号。

(2) FAT需占用较大的内存空间。

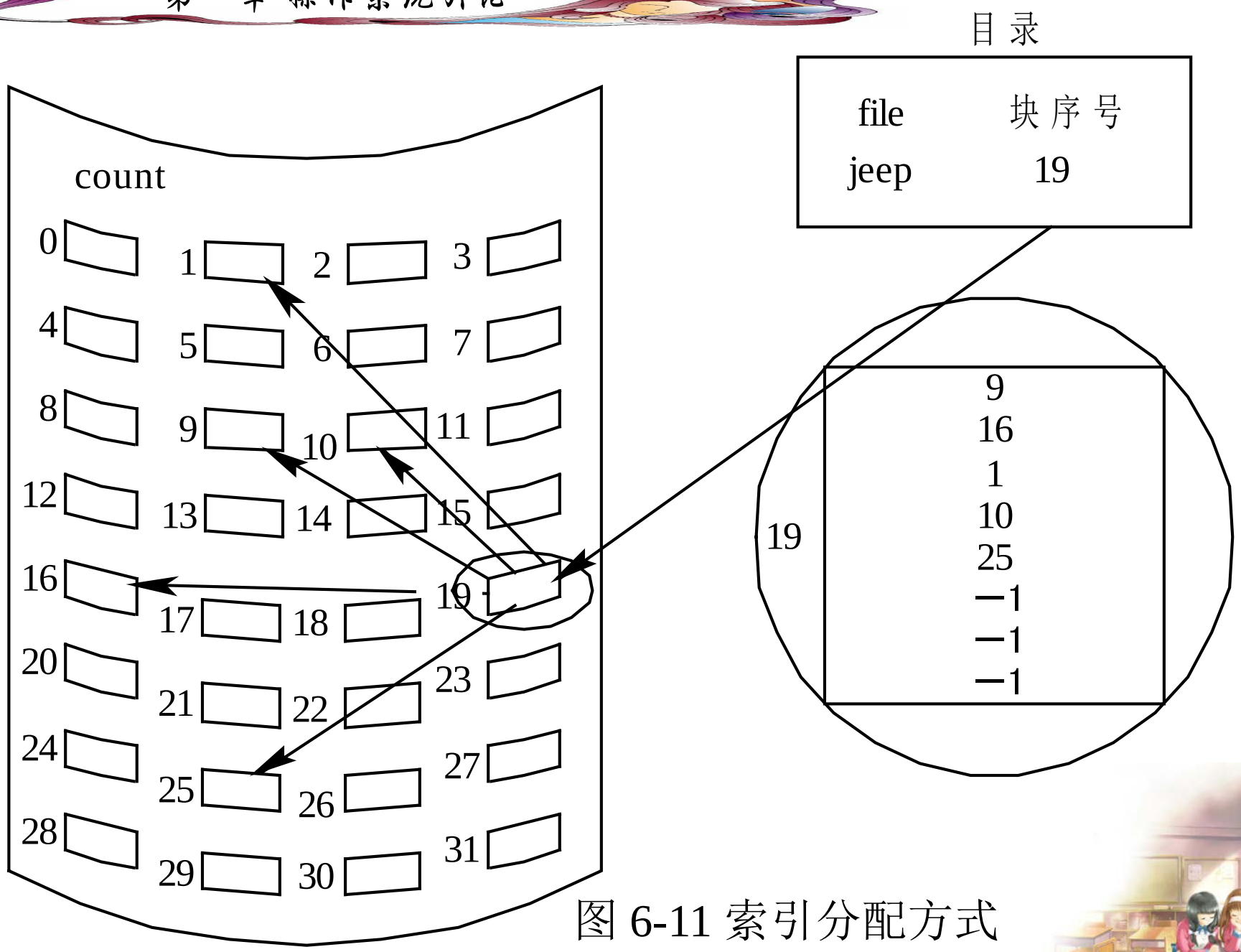

目录
file
块序号
jeep
19
count
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
19
9
16
1
10
25
−1
−1
−1

图 6-11 索引分配方式

2. 多级索引分配

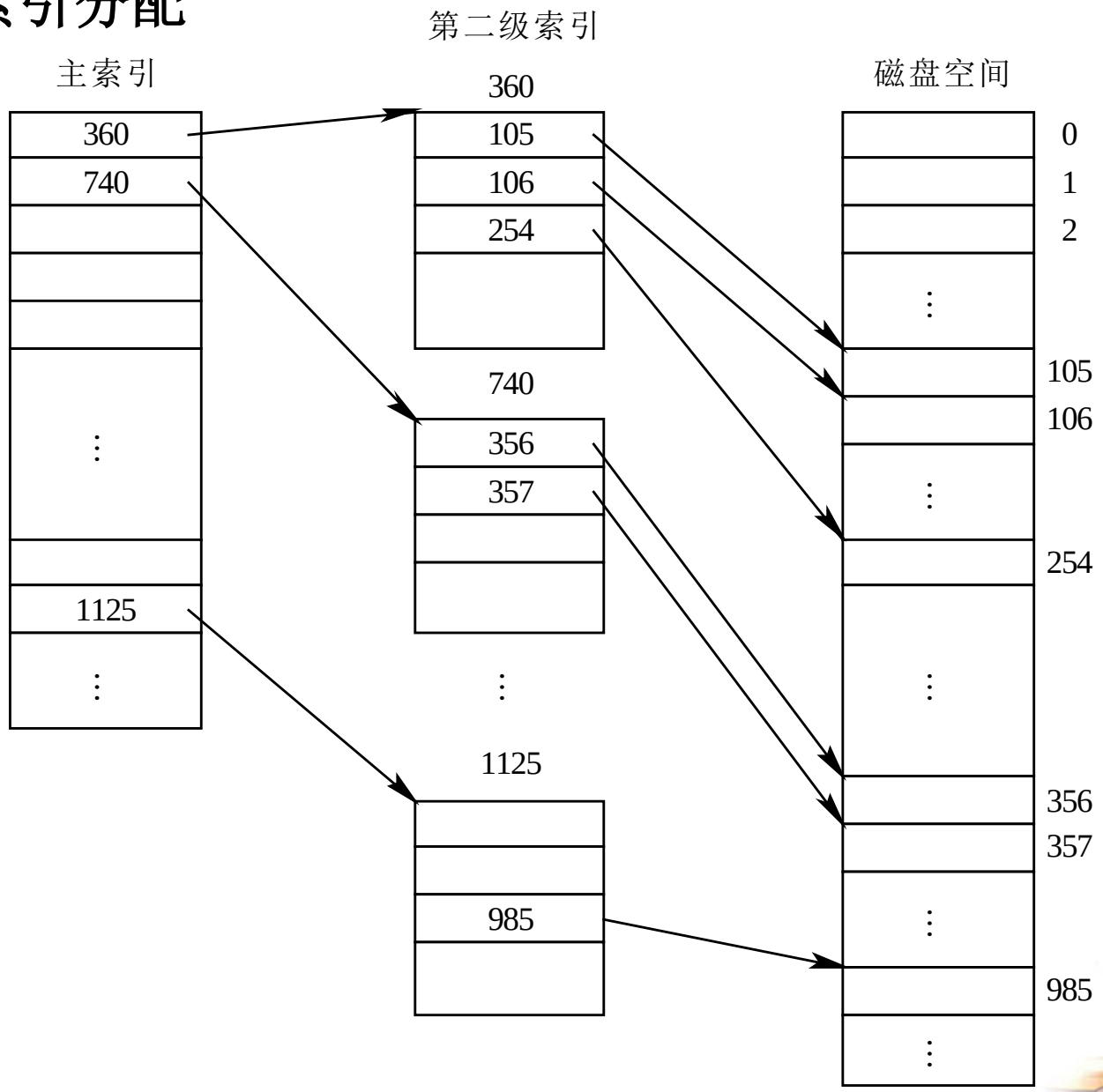


图 6-12 两级索引分配

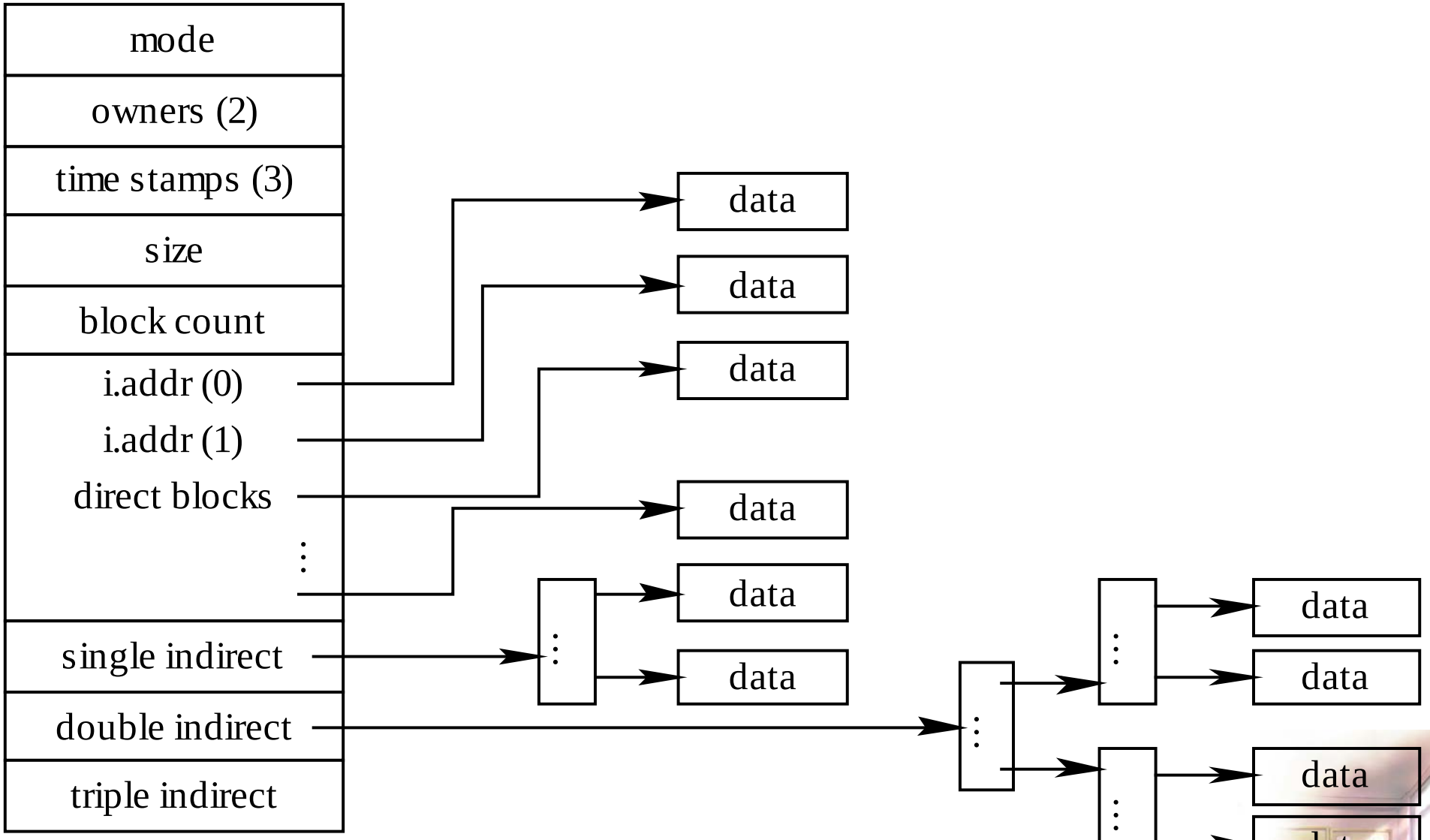

mode
owners (2)
time stamps (3)
size
block count
i.addr (0)
i.addr (1)
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
data
data
data
data
data
data

图 6-13 混合索引方式

(1) 直接地址。

为了提高对文件的检索速度， 在索引结点中可设置10个直接地址项， 即用iaddr(0)~iaddr(9)来存放直接地址。 换言之，在这里的每项中所存放的是该文件数据的盘块的盘块号。假如每个盘块的大小为 4 KB，当文件不大于40 KB时，便可直接从索引结点中读出该文件的全部盘块号。

(2) 一次间接地址。

对于大、 中型文件， 只采用直接地址是不现实的。为此，可再利用索引结点中的地址项iaddr(10)来提供一次间接地址。这种方式的实质就是一级索引分配方式。图中的一次间址块也就是索引块，系统将分配给文件的多个盘块号记入其中。在一次间址块中可存放1K个盘块号， 因而允许文件长达4 MB。

(3) 多次间接地址。

当文件长度大于4 MB+40 KB时(一次间址与10个直接地址项)， 系统还须采用二次间址分配方式。这时，用地址项iaddr(11)提供二次间接地址。该方式的实质是两级索引分配方式。系统此时是在二次间址块中记入所有一次间址块的盘号。在采用二次间址方式时，文件最大长度可达4 GB。 同理，地址项iaddr(12)作为三次间接地址， 其所允许的文件最大长度可达4 TB。

6.4 目 录 管 理

对目录管理的要求如下：

(1) 实现“按名存取”。

(2) 提高对目录的检索速度。

(3) 文件共享。

(4) 允许文件重名。

6.4.1 文件控制块和索引结点

1. 文件控制块

(1) 基本信息类

① 文件名； ② 文件物理位置； ③ 文件逻辑结构；

④ 文件的物理结构

(2) 存取控制信息类

(3) 使用信息类

文件名	扩展名	属性	备用	时间	日期	第一块号	盘块数

图 6-14 MS-DOS的文件控制块

2. 索引结点

1) 索引结点的引入

文件名	索引结点编号
文件名1	
文件名2	
⋮	⋮

图 6-15 UNIX的文件目录

2) 磁盘索引结点

(1) 文件主标识符

(2) 文件类型

(3) 文件存取权限

(4) 文件物理地址

(5) 文件长度

(6) 文件连接计数

(7) 文件存取时间

3) 内存索引结点

(1) 索引结点编号。 用于标识内存索引结点。

(2) 状态。 指示i结点是否上锁或被修改。

(3) 访问计数。 每当有一进程要访问此i结点时， 将该访问计数加1， 访问完再减1。

(4) 文件所属文件系统的逻辑设备号。

(5) 链接指针。 设置有分别指向空闲链表和散列队列的指针。

6.4.2 目录结构

1. 单级目录结构

文件名	物理地址	文件说明	状态位
文件名1			
文件名2			
⋮			

图 6-16 单级目录

单级目录的优点是简单且能实现目录管理的基本功能—按名存取，但却存在下述一些缺点：

(1) 查找速度慢

(2) 不允许重名

(3) 不便于实现文件共享

2. 两级目录

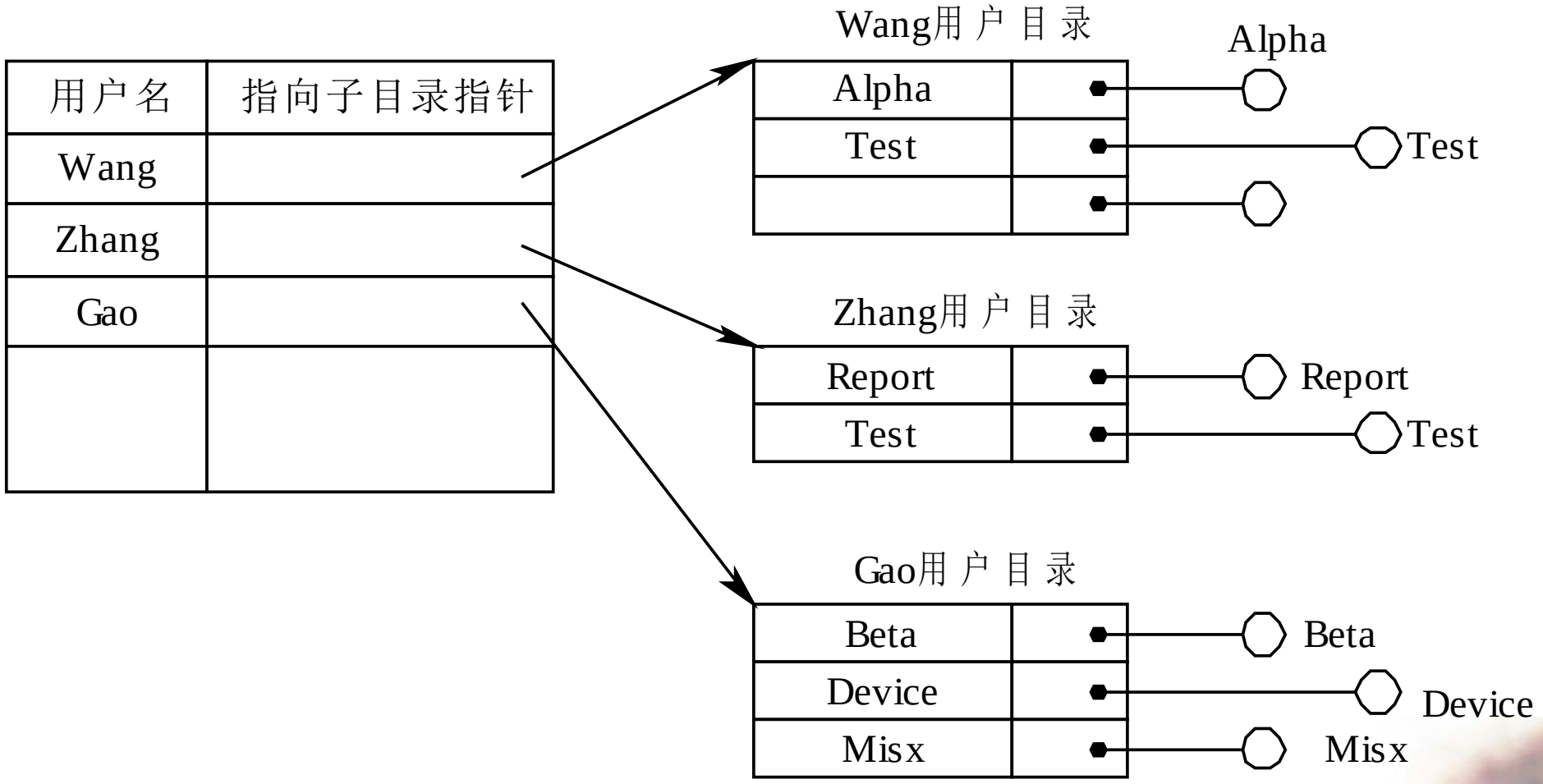


图 6-17 两级目录结构

具有以下优点：

(1) 提高了检索目录的速度

(2) 在不同的用户目录中， 可以使用相同的文件名。

(3) 不同用户还可使用不同的文件名来访问系统中的同一个共享文件

3. 多级目录结构

(1) 目录结构

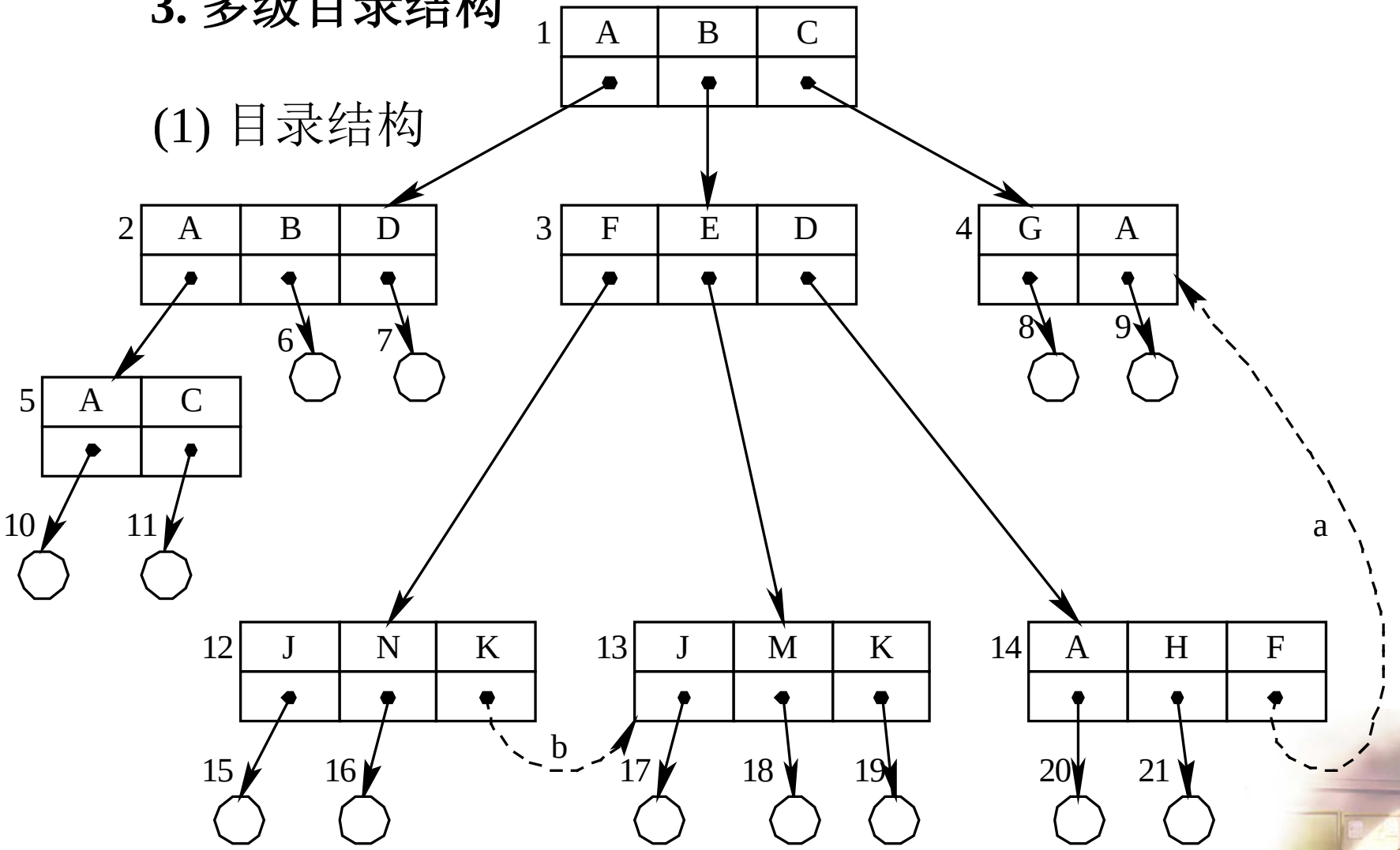


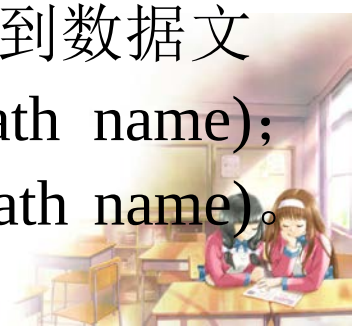
图 6-18 多级目录结构

(2) 路径名。

在树形目录结构中， 从根目录到任何数据文件， 都只有一条惟一的通路。 在该路径上从树的根(即主目录)开始，把全部目录文件名与数据文件名，依次地用“/”连接起来，即构成该数据文件的路径名(path name)。 系统中的每一个文件都有惟一的路径名。 例如，在图 6-18 中用户B为访问文件J， 应使用其路径名/B/F/J来访问。

(3) 当前目录(Current Directory)。

当一个文件系统含有许多级时，每访问一个文件，都要使用从树根开始直到树叶(数据文件)为止的、包括各中间结点(目录)名的全路径名。 这是相当麻烦的事，同时由于一个进程运行时所访问的文件，大多仅局限于某个范围，因而非常不便。 基于这一点，可为每个进程设置一个“当前目录”，又称为“工作目录”。进程对各文件的访问都相对于“当前目录”而进行。此时各文件所使用的路径名， 只需从当前目录开始， 逐级经过中间的目录文件，最后到达要访问的数据文件。把这一路径上的全部目录文件名与数据文件名用“/”连接形成路径名，如用户B的当前目录是F，则此时文件J的相对路径名仅是J本身。 这样， 把从当前目录开始直到数据文件为止所构成的路径名，称为相对路径名(relative path name)；而把从树根开始的路径名称为绝对路径名(absolute path name)。



4. 增加和删除目录

(1) 不删除非空目录。当目录(文件)不空时， 不能将其删除，而为了删除一个非空目录，必须先删除目录中的所有文件，使之先成为空目录， 后再予以删除。如果目录中还包含有子目录，还必须采取递归调用方式来将其删除， 在MS-DOS中就是采用这种删除方式。

(2) 可删除非空目录。当要删除一目录时，如果在该目录中还包含有文件，则目录中的所有文件和子目录也同时被删除。

6.4.3 目录查询技术

1. 线性检索法

根目录

1	.
1	. .
4	bin
7	dev
14	lib
9	etc
6	usr
8	tmp

在结点 6 中查找 usr 字段

结点 6 是 /usr 的目录

132

132 号盘块是 /usr 的目录

6	.
1	. .
19	dick
30	erik
51	jim
26	ast
45	bal

结点 26 是 /usr/ast 的目录

496

496 号盘块是 /usr/ast 的目录

26	.
6	. .
64	grants
92	books
60	mbox
81	minik
17	src

图 6-19 查找/usr/ast/mbox的步骤

2. Hash方法

一种处理此“冲突”的有效规则是：

(1) 在利用Hash法索引查找目录时，如果目录表中相应的目录项是空的，则表示系统中并无指定文件。

(2) 如果目录项中的文件名与指定文件名相匹配， 则表示该目录项正是所要寻找的文件所对应的目录项，故而可从中找到该文件所在的物理地址。

(3) 如果在目录表的相应目录项中的文件名与指定文件名并不匹配，则表示发生了“冲突”，此时须将其Hash值再加上一个常数(该常数应与目录的长度值互质)，形成新的索引值， 再返回到第一步重新开始查找。

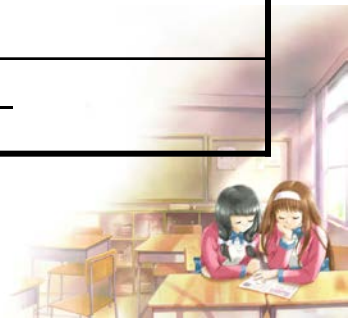
6.5 文件存储空间的管理

6.5.1 空闲表法和空闲链表法

1. 空闲表法

序号	第一空闲盘块号	空闲盘块数
1	2	4
2	9	3
3	15	5
4	—	—

图 6-20 空闲盘块表



(2) 存储空间的分配与回收。

空闲盘区的分配与内存的动态分配类似，同样是采用首次适应算法、循环首次适应算法等。例如，在系统为某新创建的文件分配空闲盘块时，先顺序地检索空闲表的各表项， 直至找到第一个其大小能满足要求的空闲区，再将该盘区分配给用户(进程)，同时修改空闲表。系统在对用户所释放的存储空间进行回收时，也采取类似于内存回收的方法， 即要考虑回收区是否与空闲表中插入点的前区和后区相邻接，对相邻接者应予以合并。

2. 空闲链表法

(1) 空闲盘块链。

(2) 空闲盘区链

6.5.2 位示图法

1. 位示图

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	1	0	0	0	1	1	1	0	0	1	0	0	1	1	0
2	0	0	0	1	1	1	1	1	1	0	0	0	0	1	1	1
3	1	1	1	0	0	0	1	1	1	1	1	1	0	0	0	0
4																
⋮																
16																

图 6-21 位示图

2. 盘块的分配

(1) 顺序扫描位示图，从中找出一个或一组其值为“0”的二进制位(“0”表示空闲时)。

(2) 将所找到的一个或一组二进制位， 转换成与之相应的盘块号。假定找到的其值为“0”的二进制位，位于位示的第i行、第j列，则其相应的盘块号应按下式计算：

$$b=n(i-1)+j$$

式中， n代表每行的位数。

(3) 修改位示图， 令map［i,j］=1。

3. 盘块的回收

(1) 将回收盘块的盘块号转换成位示图中的行号和列号。转换公式为：

$$i=(b-1)\text{DIV } n+1$$

$$j=(b-1)\text{MOD } n+1$$

(2) 修改位示图。 令map［i,j］=1。

6.5.3 成组链接法

1. 空闲盘块的组织

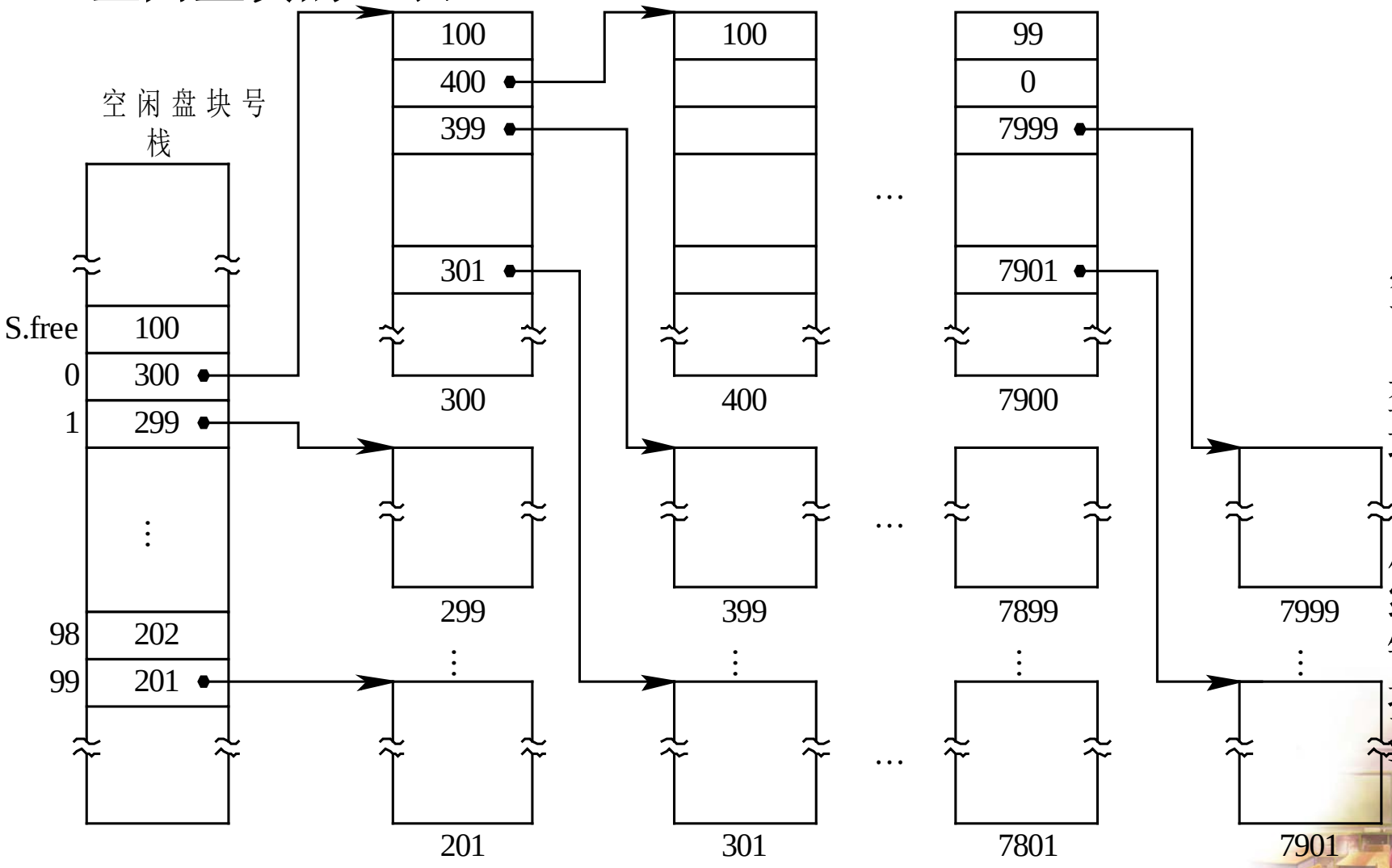


图 6-22 空闲盘块的成组链接法

2. 空闲盘块的分配与回收

当系统要为用户分配文件所需的盘块时，须调用盘块分配过程来完成。该过程首先检查空闲盘块号栈是否上锁，如未上锁，便从栈顶取出一空闲盘块号，将与之对应的盘块分配给用户，然后将栈顶指针下移一格。若该盘块号已是栈底，即S.free(0)，这是当前栈中最后一个可分配的盘块号。由于在该盘块号所对应的盘块中记有下一组可用的盘块号，因此，须调用磁盘读过程，将栈底盘块号所对应盘块的内容读入栈中，作为新的盘块号栈的内容，并把原栈底对应的盘块分配出去(其中的有用数据已读入栈中)。 然后，再分配一相应的缓冲区(作为该盘块的缓冲区)。最后，把栈中的空闲盘块数减1并返回。

在系统回收空闲盘块时，须调用盘块回收过程进行回收。它是将回收盘块的盘块号记入空闲盘块号栈的顶部，并执行空闲盘块数加1操作。当栈中空闲盘块号数目已达100时，表示栈已满，便将现有栈中的100个盘块号， 记入新回收的盘块中，再将其盘块号作为新栈底。

6.6 文件共享与文件保护

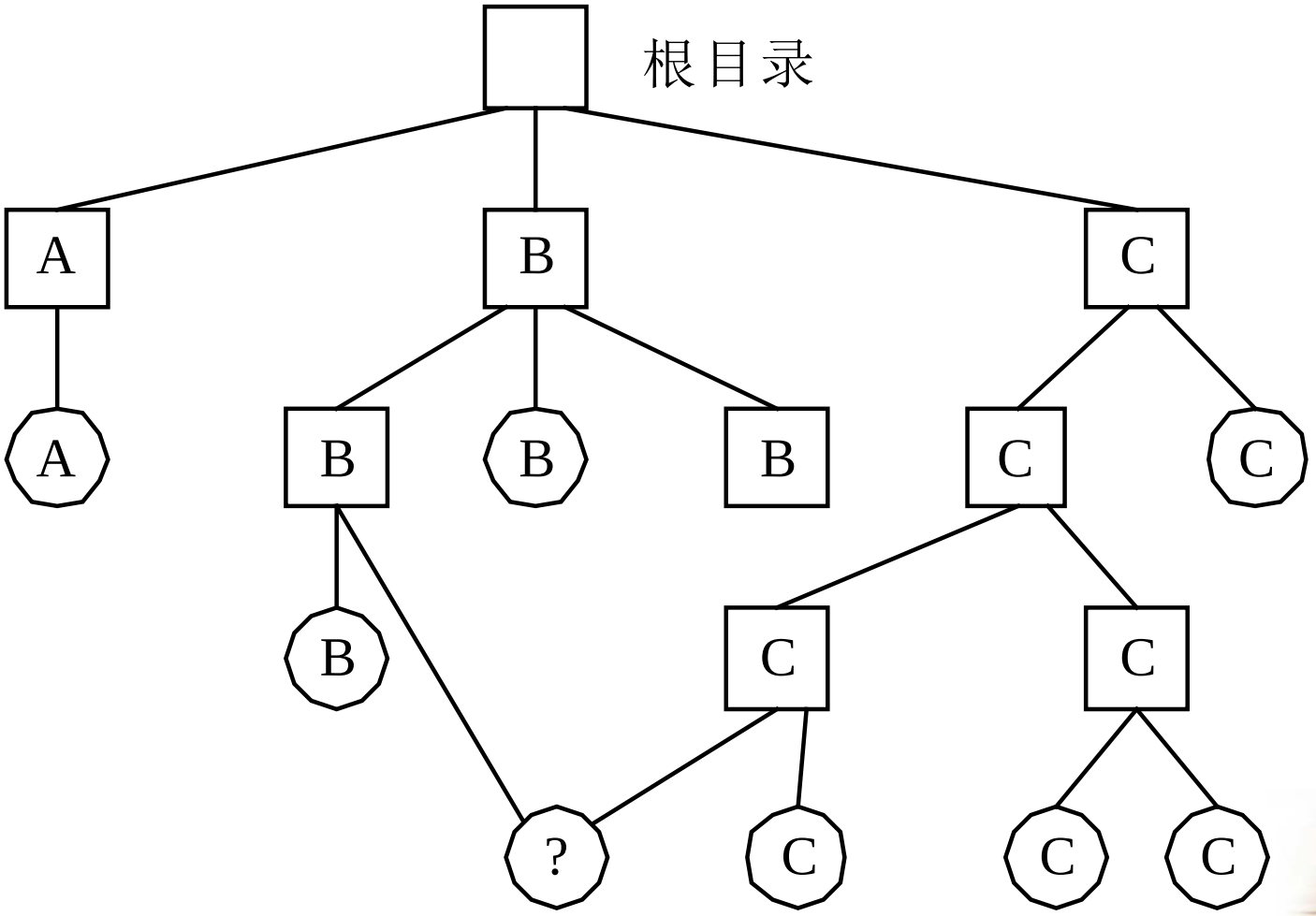
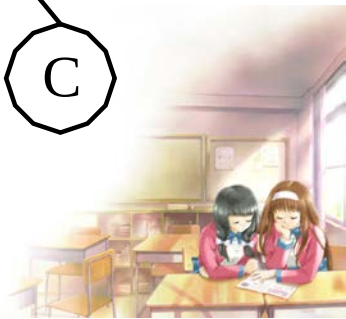


图 6-23 包含有共享文件的文件系统



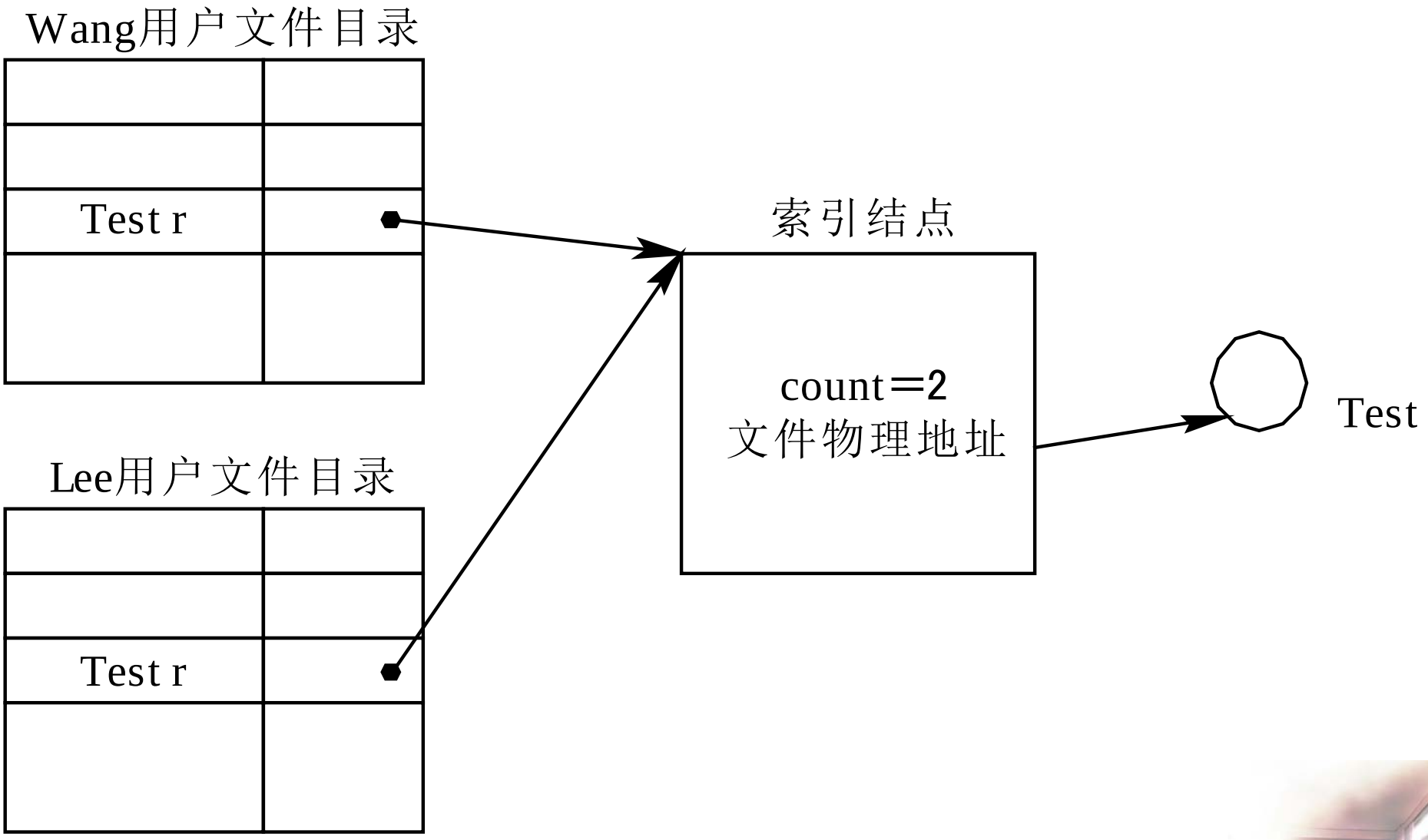


图 6-24 基于索引结点的共享方式

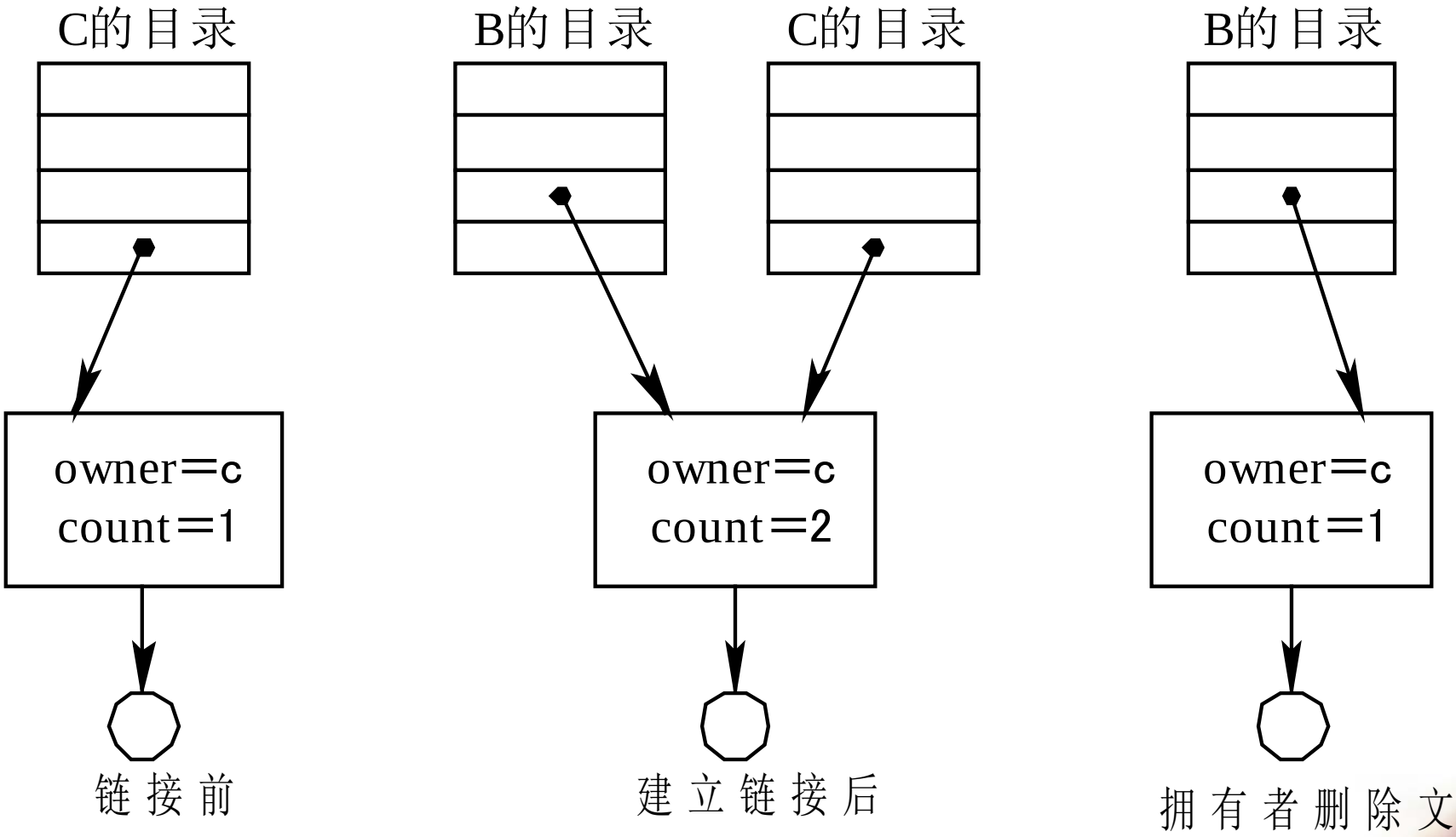


图 6-25 进程B链接前后的情况

6.6.2 利用符号链实现文件共享

在利用符号链方式实现文件共享时， 只是文件主才拥有指向其索引结点的指针；而共享该文件的其他用户，则只有该文件的路径名，并不拥有指向其索引结点的指针。这样， 也就不会发生在文件主删除一共享文件后留下一悬空指针的情况。 当文件的拥有者把一个共享文件删除后， 其他用户试图通过符号链去访问一个已被删除的共享文件时，会因系统找不到该文件而使访问失败，于是再将符号链删除，此时不会产生任何影响。

6.6.3 磁盘容错技术

(1) 通过存取控制机制来防止由人为因素所造成的文件不安全性。

(2) 通过磁盘容错技术， 来防止由磁盘部分的故障所造成的文件不安全性。

(3) 通过“后备系统”来防止由自然因素所造成的不安全性。

1. 第一级容错技术SFT-I

1) 双份目录和双份文件分配表

在磁盘上存放的文件目录和文件分配表FAT， 是文件管理所用的重要数据结构。如果这些表格被破坏， 将导致磁盘上的部分或全部文件成为不可访问的，因而也就等效于文件的丢失。为了防止这类情况发生，可在不同的磁盘上或在磁盘的不同区域中，分别建立(双份)目录表和FAT。 其中，一份被称为主目录及主FAT； 把另一份称为备份目录及备份FAT。

2) 热修复重定向和写后读校验

(1) 热修复重定向(Hot-Redirection)。

(2) 写后读校验(Read after write Verification)方式。

2. 第二级容错技术SFT-II

(1) 磁盘镜像(Disk Mirroring)。

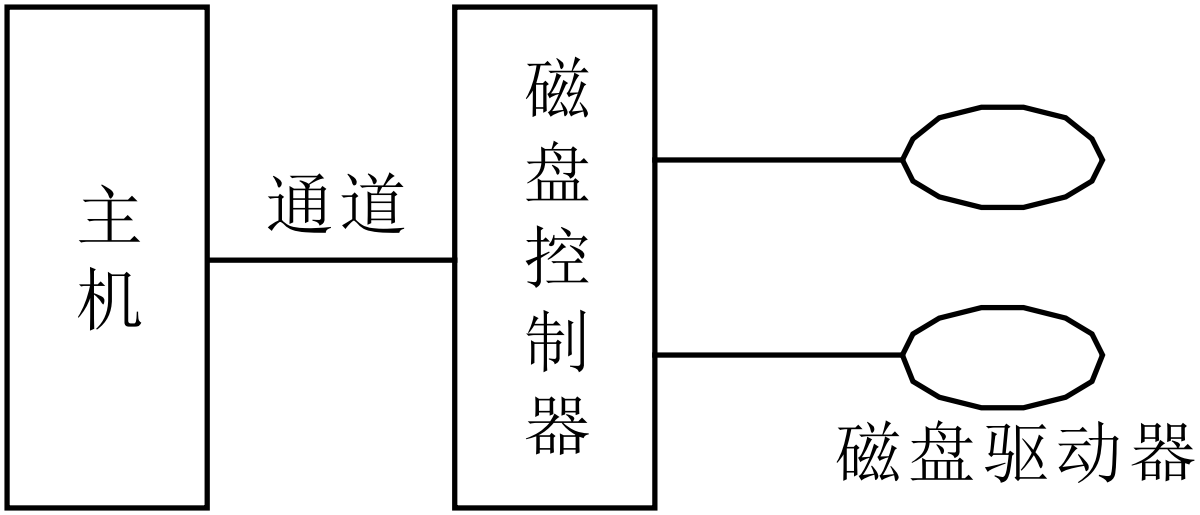


图 6-26 磁盘镜像示意

(2) 磁盘双工(Disk Duplexing)。

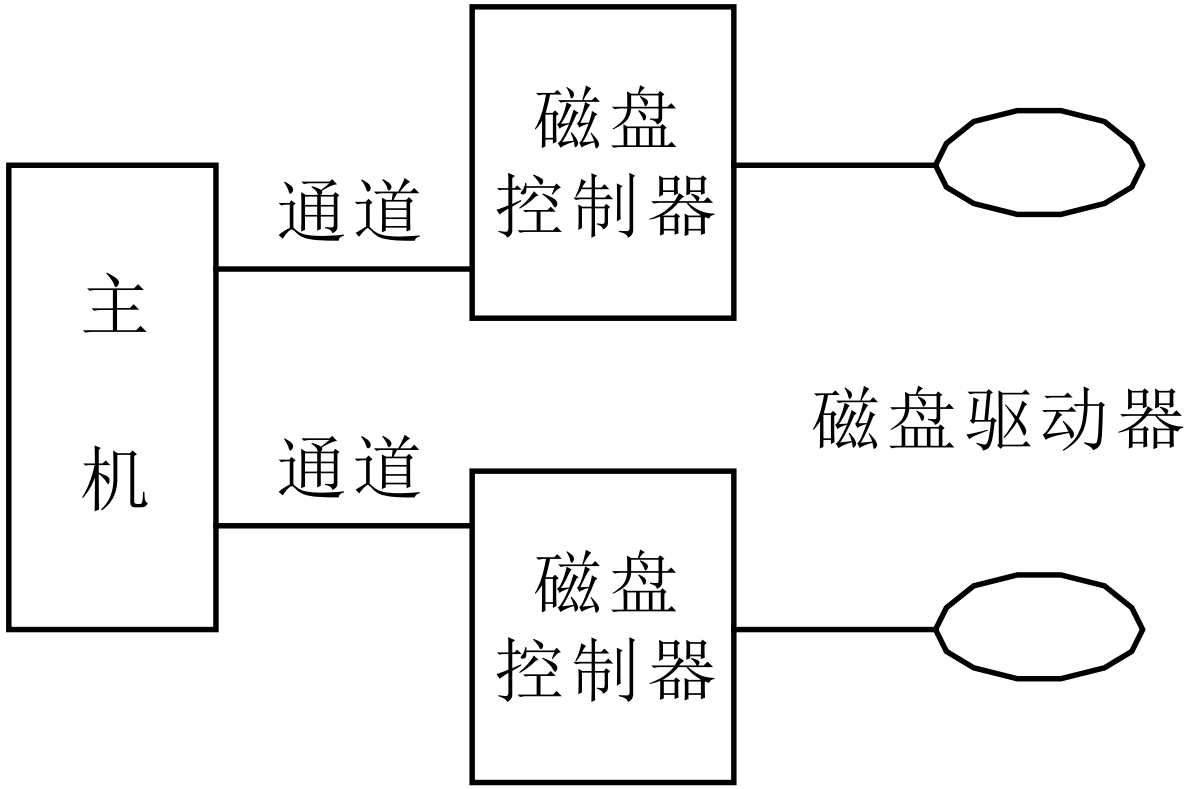


图 6-27 磁盘双工示意

6.7 数据一致性控制

6.7.1 事务

1. 事务的定义

事务是用于访问和修改各种数据项的一个程序单位。 事务也可以被看作是一系列相关读和写操作。被访问的数据可以分散地存放在同一文件的不同记录中，也可放在多个文件中。只有对分布在不同位置的同一数据所进行的读和写(含修改)操作全部完成时，才能再以托付操作(Commit Operation)来终止事务。 只要有一个读、写或修改操作失败，便须执行夭折操作(Abort Operation)。读或写操作的失败可能是由于逻辑错误，也可能是系统故障所导致的。

2. 事务记录(Transaction Record)

- 事务名： 用于标识该事务的惟一名字；
- 数据项名： 它是被修改数据项的惟一名字；
- 旧值： 修改前数据项的值；
- 新值： 修改后数据项将具有的值。

3. 恢复算法

恢复算法可利用以下两个过程：

(1) undo〈T_i〉。该过程把所有被事务T_i修改过的数据，恢复为修改前的值。

(2) redo〈T_i〉。该过程能把所有被事务T_i修改过的数据，设置为新值。

如果系统发生故障， 系统应对以前所发生的事务进行清理。

6.7.2 检查点

1. 检查点(Check Points)的作用

引入检查点的主要目的，是使对事务记录表中事务记录的清理工作经常化， 即每隔一定时间便做一次下述工作：首先是将驻留在易失性存储器(内存)中的当前事务记录表中的所有记录，输出到稳定存储器中；其次是将驻留在易失性存储器中的所有已修改数据，输出到稳定存储器中；然后是将事务记录表中的〈检查点〉记录，输出到稳定存储器中； 最后是每当出现一个〈检查点〉记录时，系统便执行上小节所介绍的恢复操作，利用redo和undo过程实现恢复功能。

2. 新的恢复算法

恢复例程首先查找事务记录表，确定在最近检查点以前开始执行的最后的事务T_i。在找到这样的事务后， 再返回去搜索事务记录表，便可找到第一个检查点记录，恢复例程便从该检查点开始，返回搜索各个事务的记录，并利用redo和undo过程对它们进行处理。

如果把所有在事务T_i以后开始执行的事务表示为事务集T，则新的恢复操作要求是：对所有在T中的事务T_K，如果在事务记录表中出现了〈T_K托付〉记录，则执行redo〈T_K〉操作；反之，如果在事务记录表中并未出现〈T_K托付〉记录，则执行undo〈T_K〉操作。

6.7.3 并发控制

1. 利用互斥锁实现“顺序性”

2. 利用互斥锁和共享锁实现顺序性

6.7.4 重复数据的数据一致性问题

1. 重复文件的一致性

文件名	i 结点
文件 1	17
文件 2	22
文件 3	12
文件 4	84

(a)

文件名	i 结点		
文件 1	17	19	40
文件 2	22	72	91
文件 3	12	30	29
文件 4	84	15	66

(b)

图 6-28 UNIX类型的目录

2. 盘块号一致性的检查

计数器组 \ 盘块号	0 1	2 3	4 5	6 7	8 9	10 11	12 13	14 15
空闲盘块号计数器组	1 1	0 1	0 1	1 1	1 0	0 1	1 1	0 0
数据盘块号计数器组	0 0	1 0	1 0	0 0	0 1	1 0	0 0	1 1

(a) 正常情况

计数器组 \ 盘块号	0 1	2 3	4 5	6 7	8 9	10 11	12 13	14 15
空闲盘块号计数器组	1 1	0 1	0 1	1 1	1 0	0 1	1 1	0 0
数据盘块号计数器组	0 0	0 0	1 0	0 0	0 1	1 0	0 0	1 1

(b) 丢失了盘块

图 6-29 检查盘块号一致性情况

盘块号 / 计数器组	0 1	2 3	4 5	6 7	8 9	10 11	12 13	14 15
空闲盘块号计数器组	1 1	0 1	2 1	1 1	1 0	0 1	1 1	0 0
数据盘块号计数器组	0 0	1 0	0 0	0 0	0 1	1 0	0 0	1 0

(c) 空闲盘块号重复出现

盘块号 / 计数器组	0 1	2 3	4 5	6 7	8 9	10 11	12 13	14 15
空闲盘块号计数器组	1 1	0 1	1 0	1 1	1 0	0 1	1 1	0 0
数据盘块号计数器组	0 0	1 0	0 2	0 0	0 1	1 0	0 0	1 1

(d) 数据盘块号重复出现

图 6-29 检查盘块号一致性情况

3. 链接数一致性检查

为每个盘块建立一个表项，其中含有该索引结点号的计数值。 在进行检查时，从根目录开始查找，每当在目录中遇到该索引结点号时， 便在该计数器表中相应文件的表项上加1。当把所有目录都检查完后，便可将该计数器表中每个表项中的索引结点号计数值与该文件索引结点中的链接计数count值加以比较， 如果两者一致，表示是正确的；否则，便是发生了链接数据不一致的错误。

如果索引结点中的链接计数count值大于计数器表中相应索引结点号的计数值，则即使在所有共享此文件的用户都不再使用此文件时，其count值仍不为0，因而该文件不会被删除。这种错误的后果是使一些已无用户需要的文件仍驻留在磁盘上，浪费了存储空间。解决的方法是用计数器表中的正确的计数值去为count重新赋值。

反之，如果出现count值小于计数器表中索引结点号计数值的情况时，就有潜在的危险。假如有两个用户共享一个文件，但是count值仍为1， 这样， 只要其中有一个用户不再需要此文件时， count值就会减为0，从而使系统将此文件删除， 并释放其索引结点及文件所占用的盘块，导致另一共享此文件的用户所对应的目录项，指向了一个空索引结点，最终是使该用户再无法访问此文件。如果该索引结点很快又被分配给其它文件，则又会带来潜在的危险。 解决的方法是将count值置为正确值。

第七章 操作系统接口

BACK

7.1 联机命令接口

7.1.1 联机命令的类型

为了能向用户提供多方面的服务，通常，OS都向用户提供了几十条甚至上百条的联机命令。根据这些命令所完成功能的不同，可把它们分成以下几类：① 系统访问类；② 磁盘操作类； ③ 文件操作类；④ 目录操作类；⑤ 通信类；⑥ 其他命令。

1. 系统访问类

在单用户微型机中，一般没有设置系统访问命令； 然而在多用户系统中，为了保证系统的安全性，都毫无例外地设置了系统访问命令， 即注册命令Login。 用户在每次开始使用某终端时，都须使用该命令，使系统能识别该用户。凡要在多用户系统的终端上上机的用户，都必须先在系统管理员处获得一合法的注册名和口令。以后，每当用户在接通其所用终端的电源后，便由系统直接调用并在屏幕上显示出以下的注册命令：

Login: /提示用户键入自己的注册名

当用户键入正确的注册名，并按下回车键后，屏幕上又会出现：

Password: /提示用户键入自己的口令

2. 键盘操作命令

(1) 磁盘格式化命令Format。

它被用于对指定驱动器上的软盘进行格式化。 每张新盘在使用前都必须先格式化。 其目的是使其记录格式能为操作系统所接受，可见，不同操作系统将磁盘初始化后的格式各异。此外，在格式化过程中，还将对有缺陷的磁道和扇区加保留记号，以防止将它分配给数据文件。

(2) 复制整个软盘命令Diskcopy。

该命令用于复制整个磁盘，另外它还有附加的格式化功能。 如果目标盘片是尚未格式化的，则该命令在执行时，首先将未格式化的软盘格式化，然后再进行复制。

(3) 软盘比较命令Diskcomp。

该命令用于将源盘与目标盘的各磁道及各扇区中的数据逐一进行比较。

(4) 备份命令Backup。

该命令用于把硬盘上的文件复制到软盘上；而RESTORE命令则完成相反的操作。

3. 文件操作命令

(1) 显示文件命令type。用于将指定文件显示在屏幕上。

(2) 拷贝文件命令copy。 用于实现文件的拷贝。

(3) 文件比较命令comp。该命令用于对两个指定文件进行比较。 两文件可以在同一个或不同的驱动器上。

(4) 重新命名命令Rename。该命令用于将以第一参数命名的文件， 改成用第二参数给定的名字。

(5) 删除文件命令erase。该命令用于删除一个或一组文件，当参数路径名为*.BAK时，表示删除指定目录下的所有其扩展名为.Bak的文件。

4. 目录操作命令

(1) 建立子目录命令mkdir。 用于建立指定名字的新目录。

(2) 显示目录命令dir。显示指定磁盘中的目录项。

(3) 删除子目录命令rmdir。该命令用于删除指定的子目录文件，但不能删除普通文件，而且，一次只能删除一个空目录(其中仅含“.”和“..”两个文件)，不能删除根及当前 目录。

(4) 显示目录结构命令tree。该命令用于显示指定盘上的所有目录路径及其层次关系。

(5) 改变当前目录命令chdir。该命令用于将当前目录改变为由路径名参数给定的目录。用“..”作参数时，表示应返回到上一级目录下。

5. 其它命令

(1) 输入输出重定向命令。

在有的OS中定义了两个标准I/O设备。通常，命令的输入取自标准输入设备，即键盘；而命令的输出通常是送往标准输出设备， 即显示终端。 如果在命令中设置输出重定向“＞”符，其后接文件名或设备名， 表示将命令的输出改向， 送到指定文件或设备上。类似地，若在命令中设置输入重定向“＜”符， 则不再是从键盘而是从重定向符左边参数所指定的文件或设备上，取得输入信息。

(2) 管道连接。

这是指把第一条命令的输出信息作为第二条命令的输入信息；类似地，又可把第二条命令的输出信息作为第三条命令的输入信息。这样，由两个(含两条)以上的命令可形成一条管道。在MS-DOS和UNIX中，都用“|”作为管道符号。其一般格式为：

Command1 |Command2| … | Commandn;

(3) 过滤命令。

在UNIX及MS-DOS中，都有过滤命令，用于读取指定文件或标准输入，从中找出由参数指定的模式，然后把所有包含该模式的行都打印出来。例如，MS-DOS中用命令

find/N“erase”(路径名)

可对由路径名指定的输入文件逐行检索，把含有字符串“erase”的行输出。其中，/N是选择开关，表示输出含有指定字串的行；如果不用N而用C，则表示只输出含有指定字串的行数；若用V，则表示输出不含指定字串的行。

(4) 批命令。

为了能连续地使用多条键盘命令，或多次反复地执行指定的若干条命令， 而又免去每次重敲这些命令的麻烦，可以提供一特定文件。在MS-DOS中提供了一种特殊文件，其后缀名用“.BAT”；在UNIX系统中称为命令文件。它们都是利用一些键盘命令构成一个程序，一次建立供多次使用。 在MS-DOS中用batch命令去执行由指定或默认驱动器的工作目录上指定文件中所包含的一些命令。

7.1.2 键盘终端处理程序

为了实现人机交互，还须在微机或终端上配置相应的键盘终端处理程序，它应具有下述几方面的功能：

(1) 接收用户从终端上打入的字符。

(2) 字符缓冲， 用于暂存所接收的字符。

(3) 回送显示。

(4) 屏幕编辑。

(5) 特殊字符处理。

1. 字符接收功能

为了实现人机交互，键盘终端处理程序必须能够接收从终端输入的字符，并将之传送给用户程序。有两种方式来实现字符接收功能：

(1) 面向字符方式。驱动程序只接收从终端打入的字符，并且不加修改地将它传送给用户程序。这通常是一串未加工的ASCII码。

(2) 面向行方式。 终端处理程序将所接收的字符暂存在行缓冲中， 并可对行内字符进行编辑。仅在收到行结束符后，才将一行正确的信息送命令解释程序。在有的计算机中，从键盘硬件送出的是键的编码(简称键码)，而不是ASCII码。例如，当打入a键时，是将键码“30″放入I/O寄存器，此时，终端处理程序必须参照某种表格，将键码转换成ASCII码。应当注意，某些IBM的兼容机使用的不是标准键码。此时，处理程序还须选用相应的表格将其转换成标准键码。

2. 字符缓冲功能

(1) 专用缓冲方式。 这是指系统为每个终端设置一个缓冲区，暂存用户键入的一批字符，缓冲区的典型长度为200个字符左右。这种方式较适合于单用户微机或终端很少的多用户机。当终端数目较多时，需要的缓冲数目可能很大，且每个缓冲的利用率也很低。例如，当有100个终端时，要求有20 KB的缓冲区。 但专用缓冲方式可使终端处理程序简化。

(2) 公用缓冲方式。 系统不必为每个终端设置专用缓冲区，只须设置一个由多个缓冲区构成的公用缓冲池。其中的每个缓冲区大小相同，如为20个字符，再将所有的空缓冲区链接成一个空缓冲区链。当终端有数据输入时，可先向空缓冲区链申请一空缓冲区来接收输入字符；当该缓冲区装满后，再申请一空缓冲区。这样， 直至全部输入完毕，并利用链接指针将这些装有输入数据的缓冲区链接成一条输入链。每当该输入链中一个缓冲区内的字符被全部传送给用户程序后，便将该缓冲区从输入链中移出，再重新链入空缓冲区链中。显然，利用公用缓冲池方式可有效地提高缓冲的利用率。图 7-1(*b*)示出了公用缓冲池方式。

3. 回送显示

回送显示(回显)是指每当用户从键盘输入一个字符后，终端处理程序便将该字符送往屏幕显示。有些终端的回显由硬件实现， 其速度较快，但往往会引起麻烦。如当用户键入口令时，为防止口令被盗用，显然不该有回显。此外，用硬件实现回显也缺乏灵活性，因而近年来多改用软件来实现回显，这样可以做到在用户需要时才回显。用软件实现回显，还可方便地进行字符变换，如将键盘输入的小写英文字母变成大写， 或相反。驱动程序在将输入的字符送往屏幕回显时，应打印在正确的位置上；当光标走到一行的最后一个位置后，便应返回到下一行的开始位置。例如，当所键入的字符数目超过一行的80个(字符)时， 应自动地将下一个字符打印到下一行的开始位置。

4. 屏幕编辑

(1) 删除字符键。

(2) 删除一行键。

(3) 插入键。

(4) 移动光标键。

(5) 屏幕上卷或下移键， 等等。

5. 特殊字符处理

(1) 中断字符。

当程序在运行中出现异常情况时，用户可通过键入中断字符的办法来中止当前程序的运行。在许多系统中是利用Break或Delete或Ctrl+C键作为中断字符。

(2) 停止上卷字符。

用户键入此字符后，终端处理程序应使正在上卷的屏幕暂停上卷，以便用户仔细观察屏幕内容。在有的系统中，是利用Ctrl+S键来停止屏幕上卷的。

(3) 恢复上卷字符。

有的系统利用Ctrl+Q键使停止上卷的屏幕恢复上卷。终端处理程序收到该字符后， 便恢复屏幕的上卷功能。

上述的Ctrl+S与Ctrl+Q两字符并不被存储，而是被用去设置终端数据结构中的某个标志。每当终端试图输出时，都须先检查该标志，若该标志已被设置，便不再把字符送至屏幕

7.1.3 命令解释程序

1. 命令解释程序的作用

在联机操作方式下，终端处理程序把用户键入的信息送键盘缓冲区中保存。一旦用户键入回车符，便立即把控制权交给命令处理程序。显然，对于不同的命令，应有能完成特定功能的命令处理程序与之对应。可见，命令解释程序的主要作用，是在屏幕上给出提示符，请用户键入命令，然后读入该命令，识别命令，再转到相应命令处理程序的入口地址，把控制权交给该处理程序去执行，并将处理结果送屏幕上显示。 若用户键入的命令有错，而命令解释程序未能予以识别，或在执行中间出现问题时，则应显示出某一出错信息。

2. 命令解释程序的组成

(1) 常驻部分。

这部分包括一些中断服务子程序。例如，正常退出中断INT 20，它用于在用户程序执行完毕后，退回操作系统；驻留退出中断INT 27，用这种方式，退出程序可驻留在内存中；还有用于处理和显示标准错误信息的INT 24等。常驻部分还包括这样的程序：当用户程序终止后，它检查暂存部分是否已被用户程序覆盖，若已被覆盖，便重新将暂存部分调入内存。

(2) 初始化部分。

它跟随在常驻内存部分之后，在启动时获得控制权。这部分还包括对AUTOEXEC.BAT文件的处理程序，并决定应用程序装入的基地址。每当系统接电或重新启动后，由处理程序找到并执行AUTOEXEC.BAT文件。由于该文件在用完后不再被需要，因而它将被第一个由COMMAND.COM装入的文件所覆盖。

(3) 暂存部分。

这部分主要是命令解释程序，并包含了所有的内部命令处理程序、批文件处理程序，以及装入和执行外部命令的程序。它们都驻留在内存中，但用户程序可以使用并覆盖这部分内存，在用户程序结束时，常驻程序又会将它们重新从磁盘

3. 命令解释程序的工作流程

系统在接通电源或复位后，初始化部分获得控制权，对整个系统完成初始化工作，并自动执行AUTOEXEC.BAT文件，之后便把控制权交给暂存部分。暂存部分首先读入键盘缓冲区中的命令，判别其文件名、扩展名及驱动器名是否正确。若发现有错，在给出出错信息后返回；若无错，再识别该命令。一种简单的识别命令的方法是基于一张表格，其中的每一表目都是由命令名及其处理程序的入口地址两项所组成。如果暂存部分在该表中能找到键入的命令，且是内部命令，便可以直接从对应表项中获得该命令处理程序的入口地址，然后把控制权交给该处理程序去执行该命令。

7.2 Shell命令语言

7.2.1 简单命令

所谓简单命令，实际上是一个能完成某种功能的目标程序的名字。UNIX系统规定的命令由小写字母构成(但仅前8个字母有效)。命令可带有参数表，用于给出执行命令时的附加信息。命令名与参数表之间还可使用一种称为选项的自变量， 用破折号开始，后跟一个或多个字母、数字。

$ Command-option argument list

例如：

$ LS file1 file2 ↙

这是一条不带选项的列目录命令，$是系统提示符。该命令用于列出file1和file2两个目录文件中所包含的目录项，并隐含地指出按英文字母顺序列表。若给出–tr选项，该命令可表示成：

$ LS-tr file1 file 2 ↙

其中，选项t和r分别表示按最近修改次序及按反字母顺序列表。通常，命令名与该程序的功能紧密相关，以便于记忆。命令参数可多可少，也可缺省。

例如：

$ LS ↙

表示自动以当前工作目录为缺省参数，打印出当前工作目录所包含的目录项。简单命令的格式比较自由，包括命令名字符的个数及用于分隔命令名、选项、各参数间的空格数等，都是任意的。简单命令的数量易于扩充。系统管理员与用户自行定义的命令，其执行方式与系统标准命令的执行方式相同。

1. 进入与退出系统

(1) 进入系统，也称为注册。事先， 用户须与系统管理员商定一个唯一的用户名。管理员用该名字在系统文件树上，为用户建立一个子目录树的根结点。当用户打开自己的终端时，屏幕上会出现Login:提示，这时用户便可键入自己的注册名，并用回车符结束。然后，系统又询问用户口令，用户可用回车符或事先约定的口令键入。

(2) 退出系统。 每当用户用完系统后，应向系统报告自己不再往系统装入任何处理要求。系统得知后，便马上为用户记账， 清除用户的使用环境。 若用户使用系统是免费的，退出操作仅仅是一种礼貌。如果用户使用的是多终端中的一个终端，为了退出，用户只须按下Control-D键即可，系统会重新给出提示符即Login，以表明该终端可供另一新用户使用。 用户的进入与退出过程，实际上是由系统直接调用Login及Logout程序完成的。

2. 文件操作命令

(1) 显示文件内容命令cat。如果用户想了解自己在当前目录中的某个或某几个指定文件的内容时，便可使用下述格式的cat命令：

$ cat filename1 filename2

(2) 复制文件副本的命令cp。其格式为：

cp source target

该命令用于对已存在的文件source建立一个名为target的副本。

(3) 对已有文件改名的命令mv。 其格式为：

mv oldname newname

用于把原来的老名字改成指定的新名字。

(4) 撤消文件的命令rm。它给出一个参数表， 是要撤消的文件名清单。

(5) 确定文件类型的命令file。该命令带有一个参数表， 用于给出想了解其(文件)类型的文件名清单。命令执行的结果，将在屏幕上显示出各个文件的类型。

3. 目录操作命令

(1) 建立目录的命令mkdir(简称md)。

(2) 撤消目录的命令rmdir(简称rd)。

(3) 改变工作目录的命令cd。

(4) 改变对文件的存取方式的命令chmod。 其格式为：

chmod［who］ op-code permission filename

4. 系统询问命令

（1）访问当前日期和时间命令date。例如，用命令

$ date

屏幕上将给出当前的日期和时间，如为：

Wed Ang 14 09:27:20 PDT 1991

表示当前日期是1991年9月14日、星期三，还有时间信息若在命令名后给出参数，则date程序把参数作为重置系统时钟的时间。

(2) 询问系统当前用户的命令who。 who命令可列出当前每一个处在系统中的用户的注册名、终端名和注册进入时间，并按终端标志的字母顺序排序。例如，报告有下列三用户：

Veronica bxo66 Aug 27 13:28

Rathomas dz24 Aug 28 07:42

Jlyates tty5 Aug 28 07:39

(3) 显示当前目录路径名的命令pwd。当前目录的路径名是从根结点开始，通过分支上的所有结点到达当前目录结点为止的路径上的所有结点的名字拼起来构成的。用户的当前目录可能经常在树上移动。如果用户忘记了自己在哪里，便可用pwd确定自己的位置。

7.2.2 重定向与管道命令

1. 重定向命令

在UNIX系统中，由系统定义了三个文件。其中，有两个分别称为标准输入和标准输出的文件，各对应于终端键盘输入和终端屏幕输出。它们是在用户注册时，由Login程序打开的。这样，在用户程序执行时，隐含的标准输入是键盘输入，标准输出即屏幕(输出)显示。但用户程序中可能不要求从键盘输入，而是从某个指定文件上读取信息供程序使用；同样，用户可能希望把程序执行时所产生的结果数据，写到某个指定文件中而非屏幕上。这就使用户必须去改变输入与输出文件，即不使用标准输入、标准输出，而是把另外的某个指定文件或设备，作为输入或输出文件。

Shell向用户提供了这种用于改变输入、输出设备的手段，此即标准输入与标准输出的重新定向。用重定向符“＜”和“＞”分别表示输入转向与输出转向。例如， 对于命令

$ cat file1

表示将文件file1的内容，在标准输出上打印出来。若改变其输出，用命令

$ cat file1＞file2

时，表示把文件file1的内容，打印输出到文件file2上。同理，对于命令

$ wc

表示对标准输入中的行中字和字符进行计数。 若改变其输入， 用命令

$ wc＜file3

则表示把从文件file3中读出的行中的字和字符进行计数。

须指明的是，在做输出转向时，若上述的文件file2并不存在，则先创建它；若已存在，则认为它是空白的，执行上述输出转向命令时，是用命令的输出数据去重写该文件；如果文件file2事先已有内容，则命令执行结果将用文件file1的内容去更新文件file2的原有内容。现在，如果又要求把file4的内容附加到现有的文件file2的末尾，则应使用另一个输出转向符“　”，即此时应再用命令

$ cat file4　file2

便可在文件file2中，除了上次复制的file1内容外，后面又附加了file4的内容。

当然，若想一次把两个文件file1和file4全部复制到file2中，则可用命令

$ cat file1 file4　file2

此外，也可在一个命令行中，同时改变输入与输出。例如，命令行

a.out＜file1＞file0

表示，在可执行文件a.out执行时，将从文件file1中提取数据，而把a.out的执行结果数据输出到文件file0中。

2. 管道命令

在有了上述的重定向思想后，为了进一步增强功能，人们又进一步把这种思想加以扩充，用符号“|”来连接两条命令，使其前一条命令的输出作为后一条命令的输入。即

$ command 1|command 2

例如，对于下述输入

cat file|wc

将使命令cat把文件file中的数据，作为wc命令的计数用输入。

7.2.3 通信命令

1. 信箱通信命令mail

它被作为在UNIX的各用户之间， 进行非交互式通信的工具。mail采用信箱通信方式。发信者把要发送的消息写成信件，“邮寄”到对方的信箱中。通常各用户的私有信箱采用各自的注册名命名，即它是目录/usr/spool/mail中的一个文件，而文件名又是用接收者的注册名来命名的。信箱中的信件可以一直保留到被信箱所有者消除为止。因而，用mail进行通信时，不要求接收者利用终端与发送者会话。亦即，在发信者发送信息时，虽然接收者已在系统中注册过，但允许他此时没有使用系统；也可以是虽在使用系统，但拒绝接收任何信息。

接收者也用mail命令读取信件，可使用可选项r、q或p等。其命令格式为：

mail［-r］［-q］［-p］［-file］［-F persons］

由于信箱中可存放所接收的多个信件，这就存在一个选取信件的问题。上述几个选项分别表示：按先进先出顺序显示各信件的内容；在打入中断字符(DEL或RETURN)后，退出mail程序而不改变信箱的内容；以及一次性地显示信箱全部内容而不带询问，把指定文件当作信件来显示。在不使用-p选项时，表示在显示完一个信件后，便出现“?”，以询问用户是否继续显示下一条消息，或选读完最后一条消息后退出mail。

2. 对话通信命令write

命令格式为：

write user［ttyname］

当接收者只有一个终端时，终端名可缺省。当接收者的终端被允许接收消息时，屏幕提示会通知接收者源用户名及其所用终端名。

3. 允许或拒绝接收消息的mesg命令

其格式为：

mesg［-n］［-y］

选项n表示拒绝对方的写许可(即拒绝接收消息)；选项y指示恢复对方的写许可，仅在此时，双方才可联机通信。当用户正在联机编写一份资料而不愿被别人干扰时，常选用n选项来拒绝对方的写许可。编辑完毕，再用带有y选项的mesg命令来恢复对方的写许可，不带自变量的mesg命令只报告当前状态而不改变它。

7.2.4 后台命令

UNIX系统提供了这种机制， 用户可以在这种命令后面再加上“&”号，以告诉Shell将该命令放在后台执行， 以便用户在前台继续键入其它命令。

在后台运行的程序仍然把终端作为它的标准输出和标准错误文件，除非对它们进行重新定向。其标准输入文件是自动地被从终端定向到一个被称为“/dev/null”的空文件中。若shell未重定向标准输入， 则shell和后台进程将会同时从终端进行读入。这时，用户从终端键入的字符可能被发送到一个进程或另一个进程， 并不能预测哪个进程将得到该字符。

7.3 系 统 调 用

7.3.1 系统调用的基本概念

(1) 运行在不同的系统状态。

(2) 通过软中断进入。

(3) 返回问题。

(4) 嵌套调用。

7.3.2 系统调用的类型

1. 进程控制类系统调用

(1) 创建和终止进程的系统调用。

(2) 获得和设置进程属性的系统调用。

(3) 等待某事件出现的系统调用。

2. 文件操纵类系统调用

(1) 创建和删除文件。

(2) 打开和关闭文件。

(3) 读和写文件。

3. 进程通信类系统调用

在OS中经常采用两种进程通信方式，即消息传递方式和共享存储区方式。当系统中采用消息传递方式时，在通信前， 必须先打开一个连接。为此，应由源进程发出一条打开连接的系统调用open connection, 而目标进程则应利用接受连接的系统调用accept connection表示同意进行通信；然后， 在源和目标进程之间便可开始通信。 可以利用发送消息的系统调用send message或者用接收消息的系统调用receive message来交换信息。通信结束后，还须再利用关闭连接的系统调用close connection结束通信。

7.3.3 系统调用的实现

1. 中断和陷入硬件机构

(1) 什么是中断和陷入。

中断是指CPU对系统发生某事件时的这样一种响应：CPU暂停正在执行的程序，在保留现场后自动地转去执行该事件的中断处理程序；执行完后，再返回到原程序的断点处继续执行 。

(2) 中断和陷入向量。

中断向量单元	外设种类	优先级	中断处理程序入口地址
060	电传输出	4	klrint
064	电传输入	4	klxint
070	纸带机输入	4	perint
074	纸带机输出	4	pcpint
⋮	⋮	⋮	⋮

(*a*) 中断向量

陷入向量单元	陷入种类	优先级	陷入处理程序入口地址
004	总线超时	7	trap
010	非法指令	7	trap
024	电源故障	7	trap
034	trap 指令	7	trap
⋮	⋮	⋮	⋮

(*b*) 陷入向量

图 7－5　中断向量与陷入向量

2. 系统调用号和参数的设置

(1) 直接将参数送入相应的寄存器中。

(2) 参数表方式。

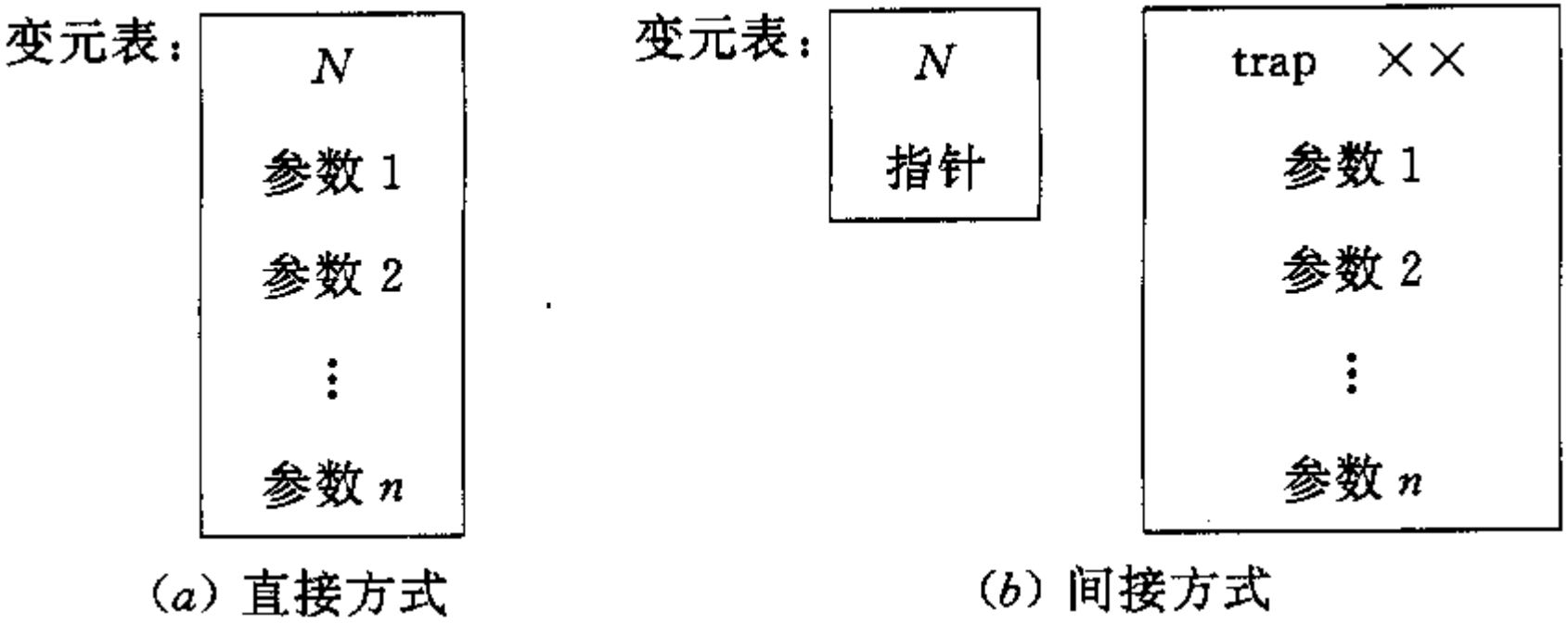


图 7 - 6 系统调用的参数形式

3. 系统调用的处理步骤

首先，将处理机状态由用户态转为系统态；之后，由硬件和内核程序进行系统调用的一般性处理，即首先保护被中断进程的CPU环境，将处理机状态字PSW、程序计数器PC、系统调用号、用户栈指针以及通用寄存器内容等，压入堆栈；然后，将用户定义的参数传送到指定的地方保存起来。

其次，是分析系统调用类型，转入相应的系统调用处理子程序。

最后，在系统调用处理子程序执行完后，应恢复被中断的或设置新进程的CPU现场，然后返回被中断进程或新进程，继续往下执行。

4. 系统调用处理子程序的处理过程

进入Creat的处理子程序后，核心将根据用户给定的文件路径名Path, 利用目录检索过程，去查找指定文件的目录项。查找目录的方式可以用顺序查找法，也可用Hash查找法。如果在文件目录中找到了指定文件的目录项，表示用户要利用一个已有文件来建立一个新文件。但如果在该已有(存)文件的属性中有不允许写属性，或者创建者不具有对该文件进行修改的权限，使认为是出错而做出错处理；若不存在访问权限问题，便将已存文件的数据盘块释放掉，准备写入新的数据文件。如未找到指名文件，则表示要创建一个新文件，核心便从其目录文件中找出一个空目录项，并初始化该目录项，包括填写文件名、文件属性、文件建立日期等，然后将新建文件打开。

7.4 UNIX系统调用

7.4.1 UNIX系统调用的类型

1. 进程控制

(1) 创建进程(fork)。

(2) 终止进程(exit)。

(3) 等待子进程结束(wait)。

(4) 执行一个文件(exec)。

(5) 获得进程ID。

(6) 获得用户ID。

(7) 进程暂停(pause)。

2. 文件操纵

(1) 创建文件(creat)。

(2) 打开文件(open)。

(3) 关闭文件(close)。

(4) 读和写文件read和write。

① 文件描述符fd； ② buf缓冲区首址。③ 用户要求传送的字节数*n*byte。

(5) 连接和去连接(link和unlink)。

3. 进程间通信

(1) 消息机制。

(2) 共享存储器机制。

(3) 信号量机制。

4. 信息维护

(1) 设置和获得时间。

(2) 获得进程和子进程时间(times)。

(3) 设置文件访问和修改时间(utime)。

(4) 获得当前UNIX系统的名称(uname)。

7.4.2 被中断进程的环境保护

1. CPU环境保护

当用户程序处在用户态，且在执行系统调用命令(即CHMK命令)之前，应在用户空间提供系统调用所需的参数表，并将该参数表的地址送入R_0寄存器。在执行CHMK命令后，处理机将由用户态转为核心态，并由硬件自动地将处理机状态长字(PSL)、程序计数器(PC)和代码操作数(code)压入用户核心栈，继而从中断和陷入向量表中取出trap.S的入口地址然后便转入中断和陷入总控程序trap.S中执行。

trap.S程序执行后，继续将陷入类型type和用户栈指针usp压入用户核心栈，接着还要将被中断进程的CPU环境中的一系列寄存器如R_0~R_{11}的部分或全部内容压入栈中。至于哪些寄存器的内容要压入栈中，这取决于特定寄存器中的屏蔽码，该屏蔽码的每一位都与R_0~R_{11}中的一个寄存器相对应。当某一位置成1时，表示对应寄存器的内容应压入栈中。

2. AP和FP指针

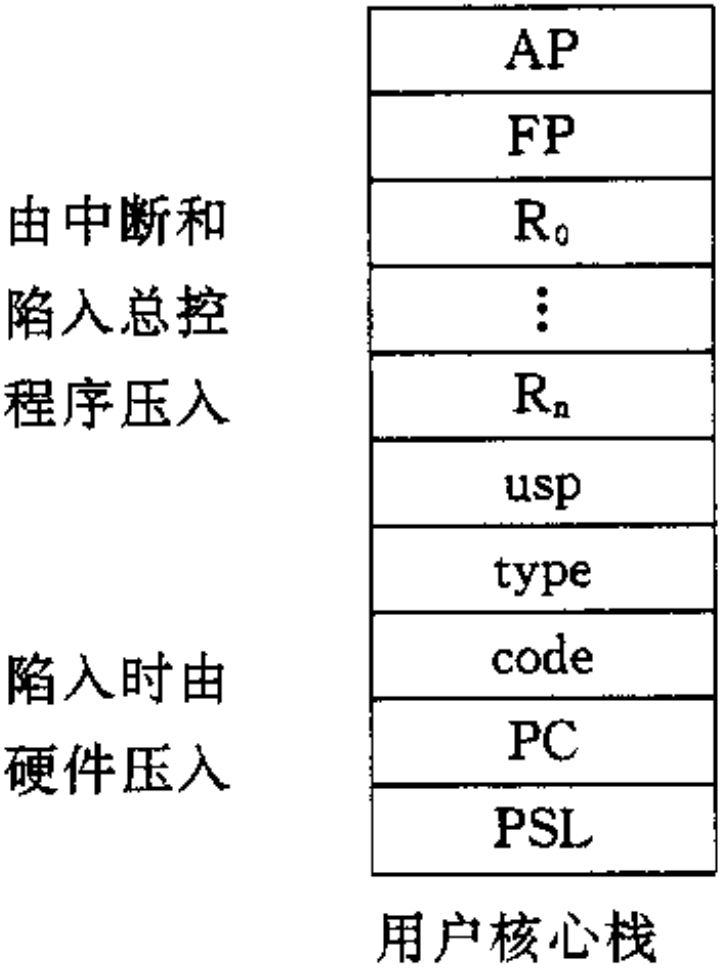


图 7 - 7 用户核心栈

7.4.3 系统调用陷入后需处理的若干公共问题

1. 确定系统调用号

trap(usp, type, code, PC, PSL)

其中，参数PSL为陷入时处理机状态字长，PC为程序计数器，code为代码操作数，type为陷入类型号，usp为用户栈指针。对陷入的处理可分为多种情况，如果陷入是由于系统调用所引起的，则对此陷入的第一步处理， 便是确定系统调用号。通常，系统调用号是包含在代码操作数中，故可利用code来确定系统调用号i。 其方法是： 令

i=code & 0377

2. 参数传送

这是对因系统调用引起的陷入的第二步处理。参数传送是指由trap.C程序将系统调用参数表中的内容，从用户区传送到User结构的U.U-arg［ ］中，供系统调用处理程序使用。由于用户程序在执行系统调用命令之前，已将参数表的首址放入R_0寄存器中，在进入trap.C程序后，该程序便将该首址赋予U.U-arg［ ］指针， 因此， trap.C在处理参数传送时，可读取该指针的内容，以获得用户所提供的参数表， 并将之送至U.U-arg［ ］中。应当注意，对不同的系统调用所需传送参数的个数并不相同，trap.C程序应根据在系统调用定义表中所规定的参数个数来进行传送，最多允许10个参数。

3. 利用系统调用定义表转入相应的处理程序

在UNIX系统中，对于不同(编号)的系统调用，都设置了与之相应的处理子程序。为使不同的系统调用能方便地转入其相应的处理子程序，也将各处理子程序的入口地址放入了系统调用定义表即Sysent［ ］中。该表实际上是一个结构数组，在每个结构中包含三个元素，其中第一个元素是相应系统调用所需参数的个数；第二个元素是系统调用经寄存器传送的参数个数；第三个元素是相应系统调用处理子程序的入口地址。在系统中设置了该表之后，便可根据系统调用号i从系统调用定义表中找出相应的表目，再按照表目中的入口地址转入相应的处理子程序，由该程序去完成相应系统调用的特定功能。在该子程序执行完后，仍返回到中断和陷入总控程序中的trap.C程序中，去完成返回到断点前的公共处理部分。

4. 系统调用返回前的公共处理

在UNIX系统中，进程调度的主要依据，是进程的动态优先级。随着进程执行时间的加长，其优先级将逐步降低。每当执行了系统调用命令、并由系统调用处理子程序返回到trap.C后，都将重新计算该进程的优先级；另外，在系统调用执行过程中，若发生了错误使进程无法继续运行时，系统会设置再调度标志。处理子程序在计算了进程的优先级后，又去检查该再调度标志是否已又被设置。若已设置，便调用switch调度程序，再去从所有的就绪进程中选择优先级最高的进程， 把处理机让给该进程去运行。

7.5 图形用户接口

7.5.1 桌面、图标和任务栏

1. 桌面与图标的初步概念

所谓桌面，是指整个屏幕空间，即在运行Windows时用户所看到的屏幕。该桌面是由多个任务共享。为了避免混淆，每个任务都通过各自的窗口显示其操作和运行情况，因此，Windows允许在桌面上同时出现多个窗口。所谓窗口是指屏幕上的一块矩形区域。应用程序(包括文档)可通过窗口向用户展示出系统所能提供的各种服务及其需要用户输入的信息；用户可通过窗口中的图标去查看和操纵应用程序或文档。

3. “开始”按钮和任务栏

(1) “开始”按钮。

(2) 任务栏。

(3) 任务栏的隐藏方式。

(4) 任务子栏。

7.5.2 窗口

1. 窗口的组成

(1) 标题栏和窗口标题。

(2) 菜单栏。

(3) 工具栏。

(4) 控制菜单按钮。

(5) 最大化、 最小化和关闭按钮。

(6) 滚动条。

(7) 窗口边框。

(8) 工作区域。

2. 窗口的性质

(1) 窗口的状态。

(2) 窗口的改变。

7.5.3 对话框

1. 对话框的用途

对话框的主要用途是实现人–机对话，即系统可通过对话框提示用户输入与任务有关的信息，比如提示用户输入要打开文件的名字、其所在目录、所在驱动器及文件类型等信息；或者对于对象的属性、窗口等的环境设置的改变等， 比如设置文件的属性、设置显示器的颜色和分辨率、设置桌面的显示效果；还可以提供用户可能需要的信息等。

2. 对话框的组成

1) 标题栏

2) 输入框

3) 按钮

(1) 命令按钮。

(2) 选择按钮。

(3) 滑块式按钮。

(4) 数字式增减按钮。