

# 1 引论

## 1.1.1 微内核结构

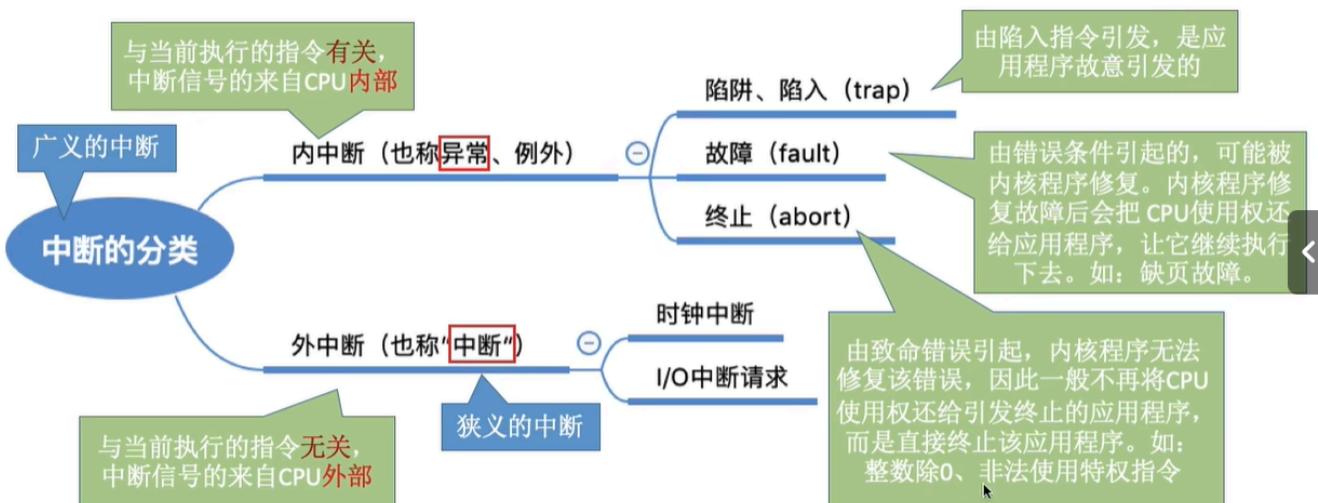
- 内核中只包括中断处理、进程通信（IPC）、基本调度等
- 文件系统、网络功能、内存管理、设备管理等作为服务在微内核上运行。
- 优点：
  - 内核易于实现、可靠性高、可移植性好、配置灵活、适应分布式环境（本地内核与远程内核对服务同样的支持）
- 缺点：
  - 速度较慢。（扩大内核减少切换）

## 1.3.2 中断和异常

类别				原因	返回行为	例子
异常	异步	中断 (interrupt)	可屏蔽中断	来自 I/O 设备的信号	总是返回到下一条指令	所有的 IRQ 中断
			不可屏蔽中断			电源掉电和物理存储器奇偶校验
	同步	陷阱 (trap)		程序内部有意设置	总是返回到下一条指令	系统调用、信号机制等 (通过中断指令实现)
		故障 (fault)		潜在可恢复的错误	返回到当前指令	缺页异常、除 0 错误、段错误
		终止 (abort)		不可恢复的错误	不会返回	硬件错误

- 内中断为同步异常（可预测），外中断为异步（不可预测）

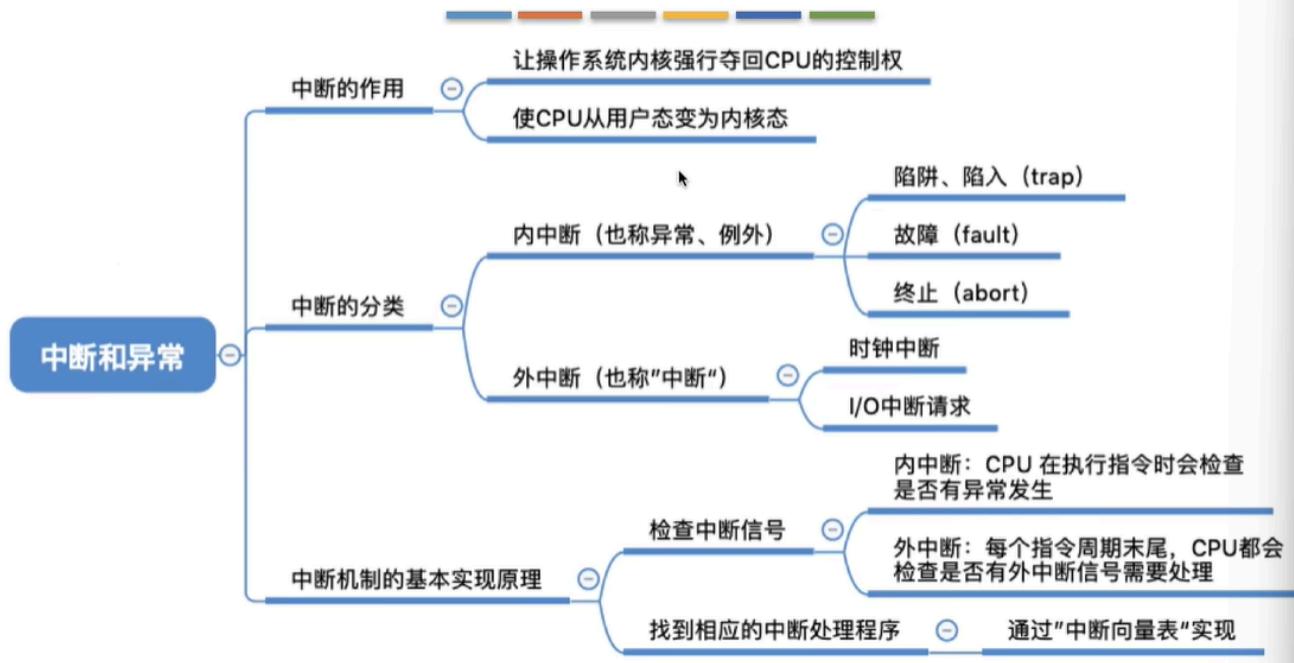
## 中断的分类



大多数的教材、试卷中，“中断”特指狭义的中断，即外中断。而内中断一般称为“异常”

王道考研/CSKAOLAN.COM

## 知识回顾与重要考点



## 2 进程与程序并发设计

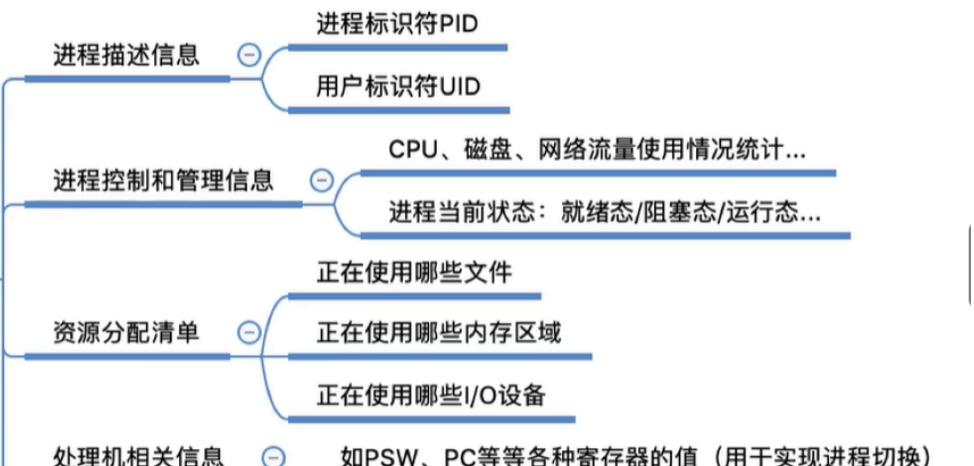
### 2.1.1 进程的概念、组成、特征

- 一个作业通常包括程序、数据和操作说明书3部分。每一个进程由进程控制块PCB、程序和数据集合组成。这说明程序是进程的一部分，是进程的实体。因此，一个作业可划分为若干个进程来完成，而每一个进程由其实体——程序和数据集合。
- 这些信息都被保存在一个数据结构 **PCB** (Process Control Block) 中，即**进程控制块**。操作系统需要对各个并发运行的进程进行管理，但凡管理时所需要的信息，都会被放在PCB中。

## 进程的组成——PCB

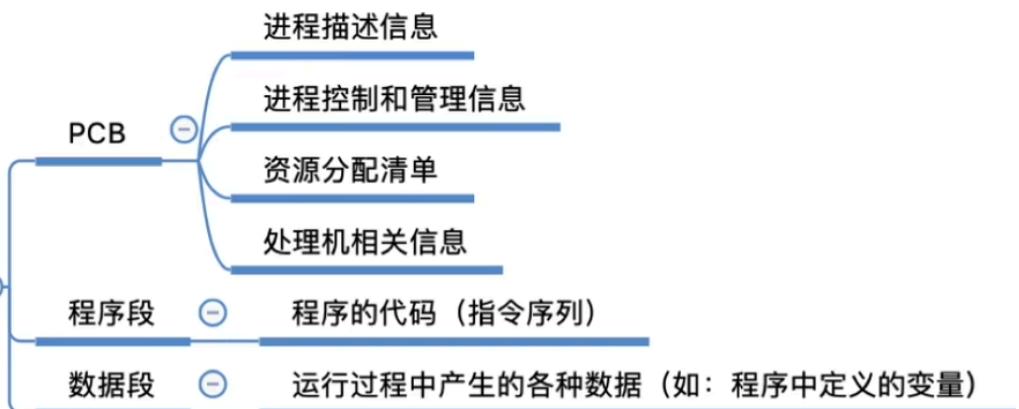
### 进程控制块 (PCB)

PCB是进程存在的唯一标志，当进程被创建时，操作系统为其创建PCB，当进程结束时，会回收其PCB。



操作系统对进程进行管理工作所需的信息都存在PCB中

### 进程的组成



### 进程的...

概念 ( ) 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

PCB是进程存在的唯一标志

进程描述信息

进程的管理者（操作系统）  
所需的数据都在PCB中

### 组成



程序段 (程序的代码 (指令序列))

数据段 (运行过程中产生的各种数据 (如：程序中定义的变量))

动态性 (进程的最基本特性)

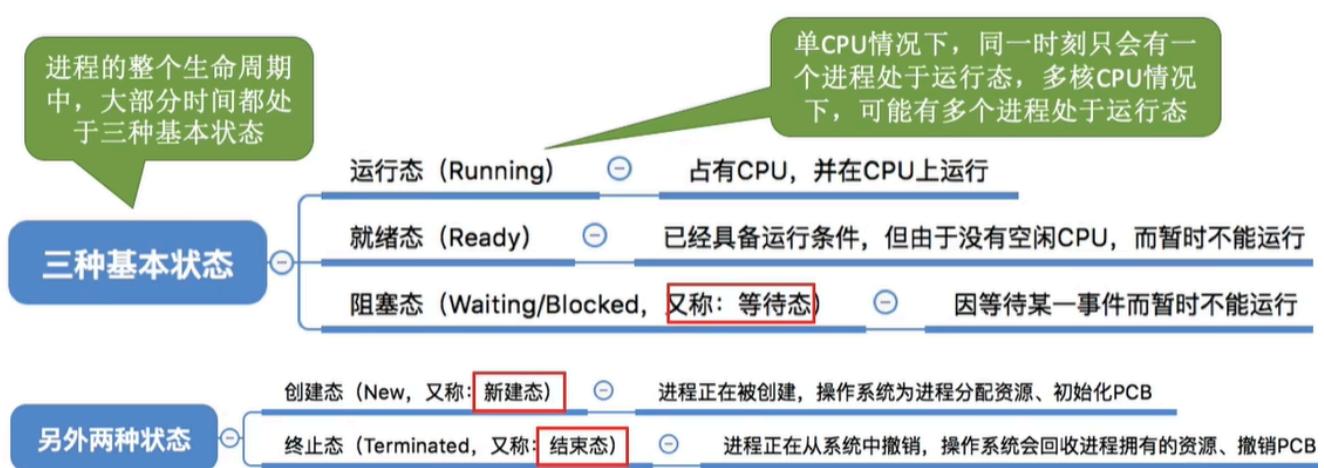
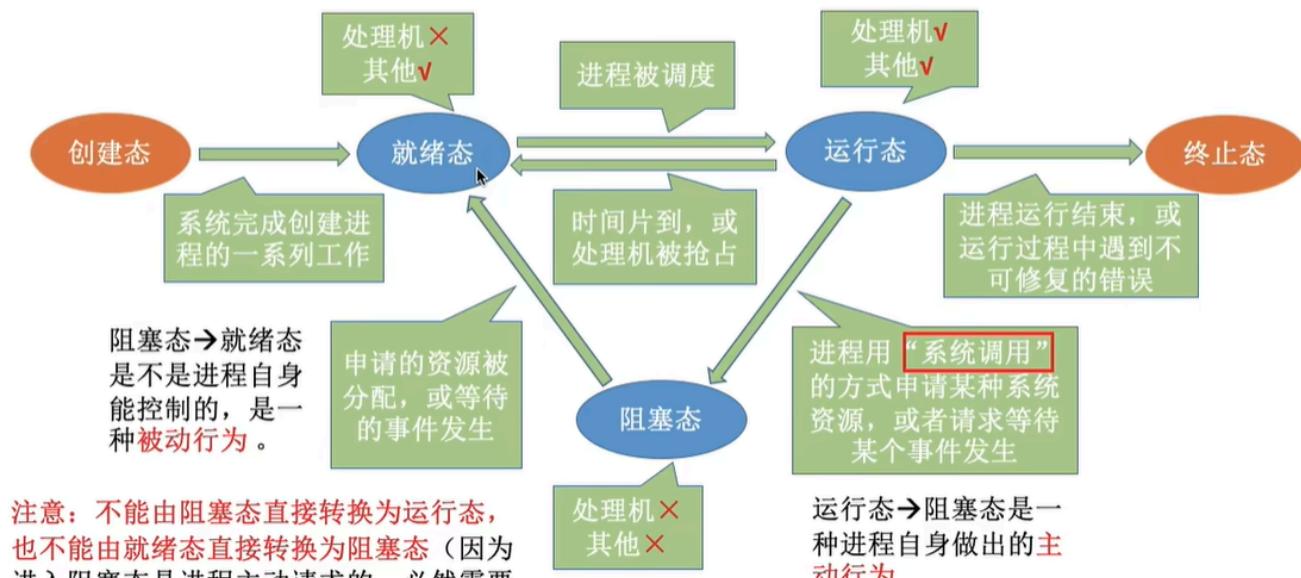
并发性

独立性 (进程是能独立运行、独立获得资源、独立接受调度的基本单位)

异步性 (各进程以不可预知的速度向前推进，可能导致运行结果的不确定性)

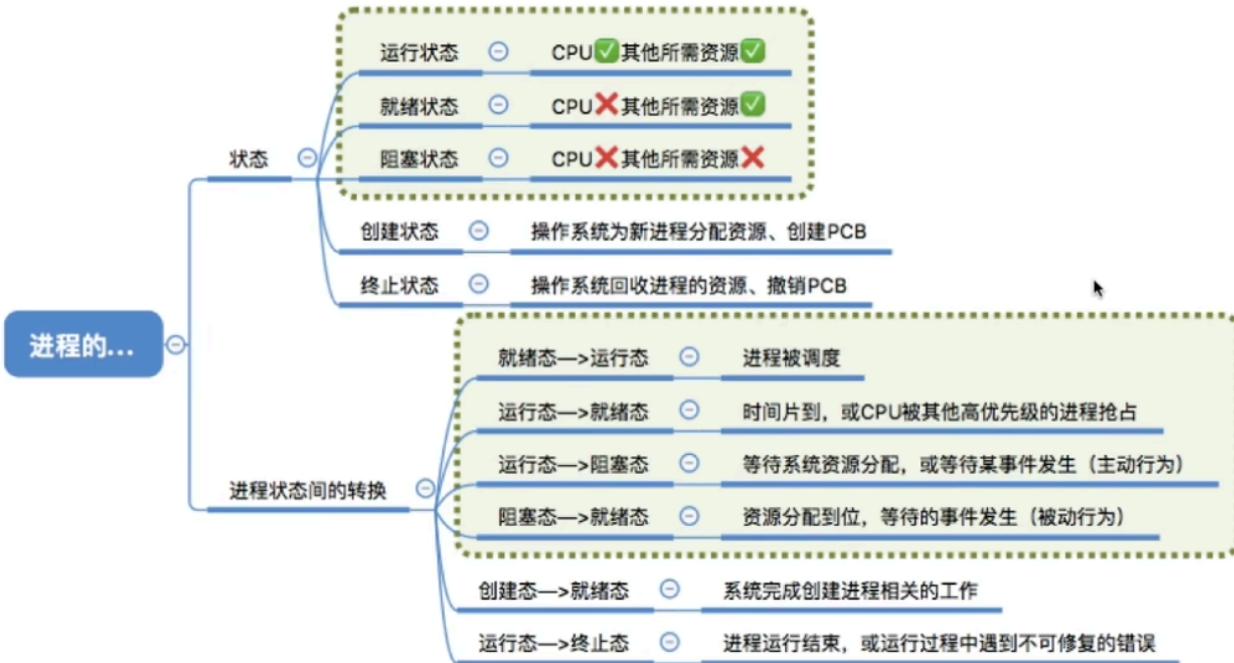
结构性

## 2.1.2 进程的状态与转换



进程的组织方式:

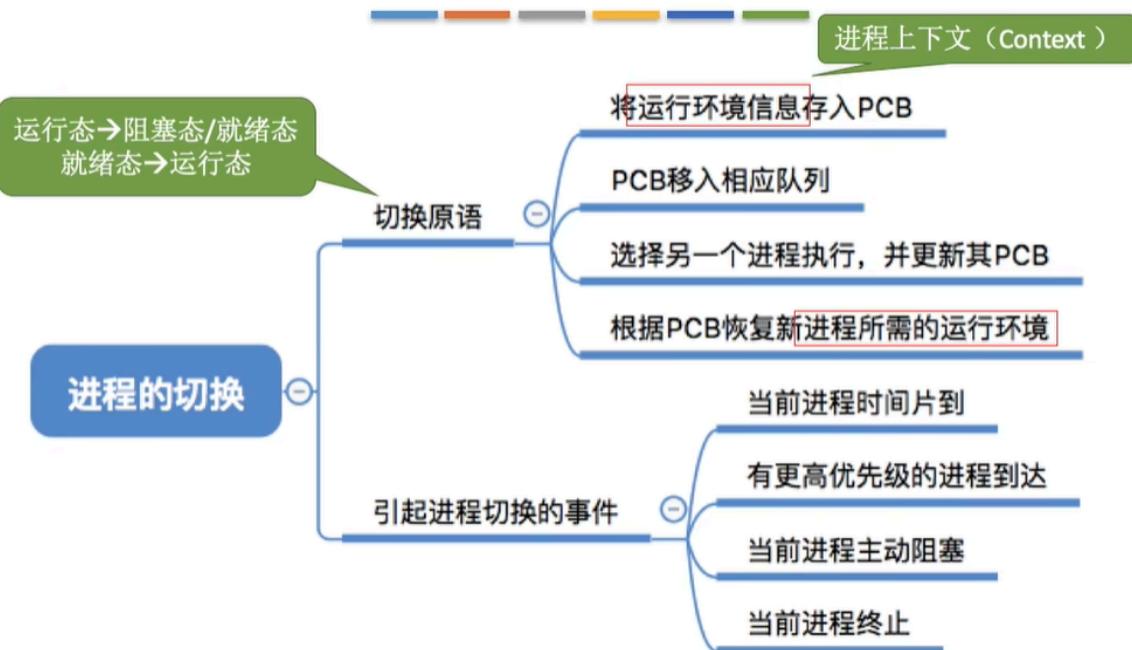
- 链接方式
  - 按照进程状态将PCB分为多个队列
  - 操作系统持有指向各个队列的指针
- 索引方式
  - 根据进程状态的不同, 建立几张索引表
  - 操作系统持有指向各个索引表的指针



### 2.1.3 进程控制

道心辰 书山有路

### 进程控制相关的原语



### 2.1.4 进程通信

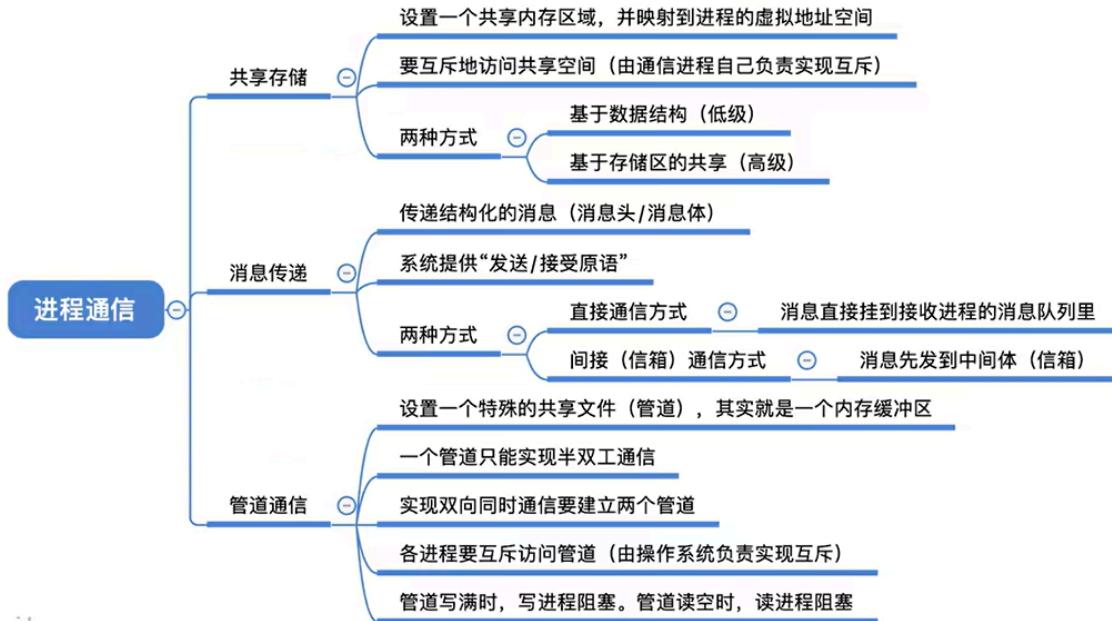
进程间通信：(Inter-Process Communication, IPC) 是指两个进程之间产生数据交互。

- 共享存储：

- 基于存储区的共享：操作系统在内存中划出一块共享存储区，数据的形式、存放位置都由通信进程控制，而不是操作系统。这种共享方式速度很快，是一种高级通信方式。
- 基于数据结构的共享：比如共享空间里只能放一个长度为10的数组。这种共享方式速度慢、限制多，是一种低级通信方式

- 消息传递：进程间的数据交换以格式化的消息（Message）为单位。进程通过操作系统提供的“发送消息/接收消息”两个原语进行数据交换。

- 直接通信方式：消息体直接说明对象
- 间接通信方式：通过信箱传递
- 管道通信：“管道”是一个特殊的共享文件，又名pipe文件。其实就是在内存中开辟一个大小固定的内存缓冲区。（先进先出）
  - 管道只能采用半双工通信，某一时间段内只能实现单向的传输。如果要实现双向同时通信，则需要设置两个管道。
  - 各进程要互斥地访问管道（由操作系统实现）
  - 当管道写满时，写进程将阻塞，直到读进程将管道中的数据取走，即可唤醒写进程。
  - 当管道读空时，读进程将阻塞，直到写进程往管道中写入数据，即可唤醒读进程。



- 信号量机制：

- 优点：
  - 简单，而且表达能力强（用P.V操作可解决任何同步互斥问题）
- 缺点：
  - 不够安全；P.V操作使用不当会出现死锁；
  - 遇到复杂同步互斥问题时实现复杂

### 2.1.5 线程的概念与特点

- 线程是一个基本的CPU执行单元，也是程序执行流的最小单位。引入线程之后，不仅是进程之间可以并发，进程内的各线程之间也可以并发，从而进一步提升了系统的并发度，使得一个进程内也可以并发处理各种任务。引入线程后，进程只作为除CPU之外的系统资源的分配单元（如打印机、内存地址空间等都是分配给进程的）。

# 线程的属性

线程是处理机调度的单位

多CPU计算机中，各个线程可占用不同的CPU

每个线程都有一个线程ID、线程控制块（TCB）

线程也有就绪、阻塞、运行三种基本状态

线程几乎不拥有系统资源

同一进程的不同线程间共享进程的资源

由于共享内存地址空间，同一进程中的线程间通信甚至无需系统干预

同一进程中的线程切换，不会引起进程切换

不同进程中的线程切换，会引起进程切换

切换同进程内的线程，系统开销很小

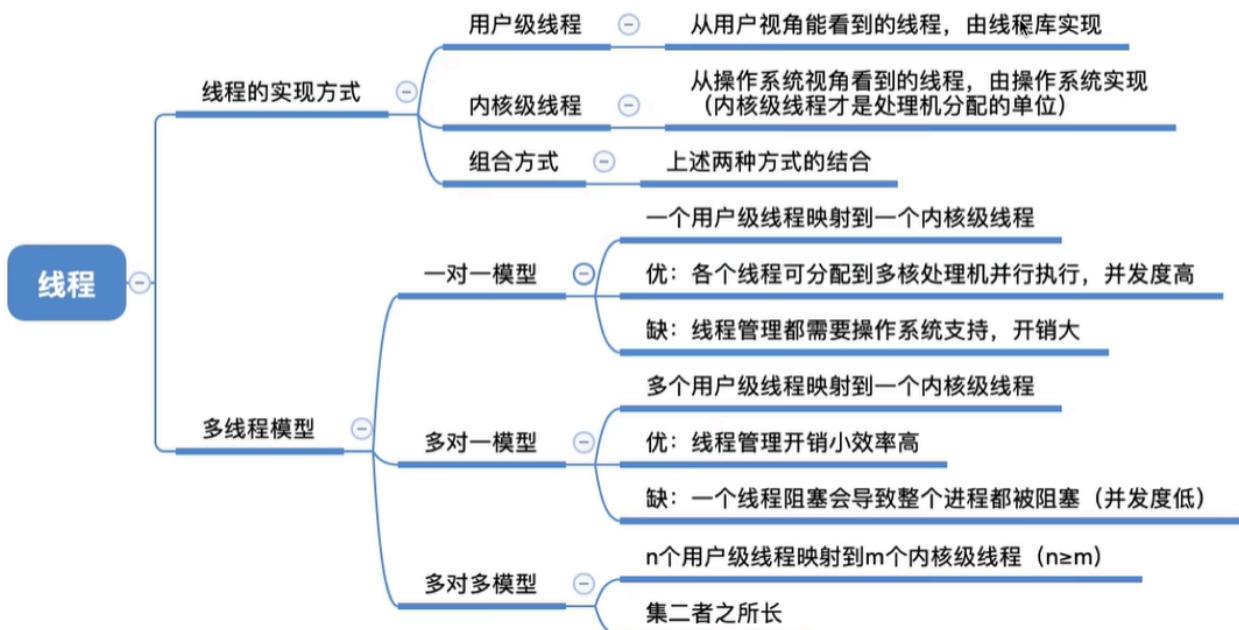
切换进程，系统开销较大

## 线程的属性

### 2.1.6 线程的实现方式和多线程模型

- 用户级线程：
  - 不需要CPU变态
  - 优点：用户级线程的切换在用户空间即可完成，不需要切换到核心态，线程管理的系统开销小，效率高
  - 缺点：当一个用户级线程被阻塞后，整个进程都会被阻塞，并发度不高。多个线程不可在多核处理机上并行运行。
- 内核级线程：
  - 需要CPU变态
  - 优点：当一个线程被阻塞后，别的线程还可以继续执行，并发能力强。多线程可在多核处理机上并行执行。
  - 缺点：一个用户进程会占用多个内核级线程，线程切换由操作系统内核完成，需要切换到核心态，因此线程管理的成本高，开销大。
- 多线程模型：
  - 一对一模型：
    - 优点：当一个线程被阻塞后，别的线程还可以继续执行，并发能力强。多线程可在多核处理机上并行执行。
    - 缺点：一个用户进程会占用多个内核级线程，线程切换由操作系统内核完成，需要切换到核心态，因此线程管理的成本高，开销大。
  - 多对一模型：

- 优点：用户级线程的切换在用户空间即可完成，不需要切换到核心态，线程管理的系统开销小，效率高
- 缺点：当一个用户级线程被阻塞后，整个进程都会被阻塞，并发度不高。多个线程不可在多核处理器上并行运行
- 多对多模型：
  - 克服了多对一模型并发度不高的缺点（一个阻塞全体阻塞），又克服了一对一模型中一个用户进程占用太多内核级线程，开销太大的缺点。



## 2.1.6 线程的状态与转换

### 2.3.1 进程的同步与互斥

## 3. 内存管理

### 3.1 内存管理 (1)

#### 3.1.1 存储的分配

- 存储组织：存储组织的功能是在存储技术和CPU寻址技术许可的范围内组织合理的存储结构，其依据是访问速度匹配关系、容量要求和价格。如：“寄存器-内存-外存”结构和“寄存器-缓存-内存-外存”结构；  
现在微机中的存储层次组织：访问速度越来越慢，容量越来越大，价格越来越便宜；**最佳状态应是各层次的存储器都处于均衡的繁忙状态。**
- 需求与存储管理的目标：
  - 需求：
    - 从每个计算机使用者（程序员）的角度：
      1. 整个空间都归我使用；
      2. 不希望任何第三方因素妨碍我的程序的正常运行；

- 从计算机平台提供者的角度：尽可能同时为多个用户提供服务；
- 分析：
  1. 计算机至少同时存在两个程序：一个用户程序和一个服务程序（操作系统）
  2. 每个程序具有的地址空间应该是相互独立的；
  3. 每个程序使用的空间应该得到保护；
- 存储管理的基本目标：
  1. 地址独立：程序发出的地址与物理地址无关
  2. 地址保护：一个程序不能访问另一个程序的地址空间
- 存储管理的功能：
  1. 存储分配和回收：是存储管理的**主要内容**。讨论其算法和相应的数据结构。
  2. 地址变换：可执行文件生成中的链接技术、程序加载时的重定位技术，进程运行时硬件和软件的地址变换技术和机构。
  3. 存储共享和保护：代码和数据共享，对地址空间的访问权限（读、写、执行）。
  4. 存储器扩充：它涉及存储器的逻辑组织和物理组织；
- 概念：
  - 地址空间：源程序经过编译后得到的目标程序，存在于它所限定的地址范围内，这个范围称为地址空间。简言之，地址空间是**逻辑地址**的集合。
  - 存储空间：存储空间是指主存中一系列存储信息的物理单元的集合，这些单元的编号称为物理地址或绝对地址。简言之，存储空间是**物理地址**的集合。

### 3.1.2 内存管理

- 单道程序的内存管理：
  - 条件：
    - 在单道程序环境下，整个内存里只有两个程序：一个用户程序和操作系统。
    - 操作系统所占的空间是固定的。因此可以将用户程序永远加载到同一个地址，即用户程序永远从同一个地方开始运行
  - 结论：用户程序的地址在运行之前可以计算。
  - 方法：
    - 静态地址翻译：即在程序运行之前就计算出所有物理地址。
    - 静态翻译工作可以由加载器实现。
  - 分析：
    - 地址独立？YES. 因为用户无需知道物理内存的相关知识。
    - 地址保护？YES. 因为没有其它用户程序。
  - 优缺点：
    - 优点：
      1. 最简单，适用于单用户、单任务的OS。CP/M和DOS
      2. 执行过程中无需任何地址翻译工作，程序运行速度快。
    - 缺点：
      1. 比物理内存大的程序无法加载，因而无法运行。

2. 造成资源浪费（小程序会造成空间浪费；不区分常用/非常用数据；I/O时间长会造成计算资源浪费）。

- 多道程序的存储管理：

- 空间的分配：分区式分配

- 把内存分为一些大小相等或不等的分区(partition)，每个应用程序占用一个或几个分区。操作系统占用其中一个分区。

- 适用于多道程序系统和分时系统，支持多个程序并发执行，但难以进行内存分区的共享。

- 方法：

- 固定（静态）式分区分配，程序适应分区。

- 可变（动态）式分区分配，分区适应程序。

- 固定式分区（静态存储区域）：当系统初始化时，把存储空间划分成若干个任意大小的区域；然后，把这些区域分配给每个用户作业。

- **把内存划分为若干个固定大小的连续分区。**

- 分区大小相等：只适合于多个相同程序的并发执行（处理多个类型相同的对象）。

- 分区大小不等：多个小分区、适量的中等分区、少量的大分区。根据程序的大小，分配当前空闲的、适当大小的分区。

- 优点：易于实现，开销小。

- 缺点：**内碎片造成浪费**，分区总数固定，限制了并发执行的程序数目。

- 采用的数据结构：分区表——记录分区的大小和使用情况

分配方式：

- 单一队列分配方式

- 多队列分配方式

- 可变式分区：

- 分区的边界可以移动，即分区的大小可变。

- 优点：没有内碎片。缺点：有外碎片。

- 系统中的碎片：**内存中无法被利用的存储空间称为碎片。**

- 1. 内部碎片：

- 指分配给作业的存储空间中未被利用的部分，如固定分区中存在的碎片。

- 单一连续区存储管理、固定分区存储管理等都会出现内部碎片。

- 内部碎片无法被整理，但作业完成后会得到释放。它们其实已经被分配出去了，只是没有被利用。

## 2. 外部碎片：

- 指系统中无法利用的小的空闲分区。如分区与分区之间存在的碎片。这些不连续的区间就是外部碎片。动态分区管理会产生外部碎片。
- 外部碎片才是造成内存系统性能下降的主要原因。外部碎片可以被整理后清除。
- **消除外部碎片的方法：紧凑技术。**

## • 空闲空间的管理：跟踪内存的使用，跟踪的办法有两种：位图表示法（分区表）和链表表示法（分区链表）

- 位图表示法：给每个分配单元赋予一个字位，用来记录该分配单元是否闲置。例如，字位取值为1表示单元闲置，取值为0则表示已被占用，这种表示方法就是位图表示法。

链表表示法：将分配单元按照是否闲置链接起来，这种方法称为链表表示法。

- 两种方法的特点：

### 1. 位图表示法：

- 空间成本固定：不依赖于内存中的程序数量。
- 时间成本低：操作简单，直接修改其位图值即可。
- 没有容错能力：如果一个分配单元为1，不能肯定应该为1还是因错误变成1。

### 2. 链表表示法：

- 空间成本：取决于程序的数量。
- 时间成本：链表扫描通常速度较慢，还要进行链表项的插入、删除和修改。
- 有一定容错能力：因为链表有被占空间和闲置空间的表项，可以相互验证。

## 3.1.3 内存分配管理算法

### • (动态分区算法，产生外碎片)

- 1. 首次适应算法（First Fit）：**每个空白区按其在存储空间中地址递增的顺序连在一起，在为作业分配存储区域时，从这个空白区域链的**始端开始查找，选择第一个足以满足请求的空白块。**
- 2. 下次适应算法（Next Fit）：**把存储空间中空白区构成一个循环链，每次为存储请求查找合适的分区时，**总是从上次查找结束的地方开始，只要找到一个足够大的空白区，就将它划分后分配出去。**
- 3. 最佳适应算法（Best Fit）：**为一个作业选择分区时，**总是寻找其大小最接近于作业所要求的存储区域。**
- 4. 最坏适应算法（Worst Fit）：**为作业选择存储区域时，**总是寻找最大的空白区。**

### • 算法特点：

首次适应：优先利用内存低地址部分的空闲分区。但由于低地址部分不断被划分，在低地址会留下许多难以利用的很小的空闲分区（碎片或零头），而每次查找又都是从低地址部分开始，增加了查找可用空闲分区的开销。

下次适应：使存储空间的利用更加均衡，不致使小的空闲区集中在存储区的一端，但这会导致缺乏大的空闲分区。

最佳适应：若存在与作业大小一致的空闲分区，则它必然被选中；若不存在与作业大小一致的空闲分区，则只划分比作业稍大的空闲分区，从而保留了大的空闲分区。最佳适应算法往往使剩下的空闲区非常小，从而在存储器中留下许多难以利用的小空闲区（碎片）。

最坏适应：总是挑选满足作业要求的最大的分区分配给作业。这样使分给作业后剩下的空闲分区也较大，可装下其它作业。由于最大的空闲分区总是因首先分配而划分，当有大作业到来时，其存储空间的申请往往得不到满足。

- 基于索引搜索的分配算法：基于顺序搜索的动态分区分配算法一般只是适合于较小的系统，如果系统的分区很多，空闲分区表（链）可能很大（很长），检索速度会比较慢。为了提高搜索空闲分区的速度，大中型系统采用了基于索引搜索的动态分区分配算法。

# 快速适应算法

- 快速适应算法，又称为**分类搜索法**，把空闲分区按容量大小进行分类，经常用到长度的空闲区设立单独的空闲区链表。系统为多个空闲链表设立一张管理索引表。

## 优点：

- 查找效率高，仅需要根据程序的长度，寻找到能容纳它的最小空闲区链表，取下第一块进行分配即可。  
该算法在分配时，**不会对任何分区产生分割**，所以能保留大的分区，也不会产生内存碎片。

## 缺点：

- 在分区归还主存时算法复杂，系统开销较大。在分配空闲分区时是以进程为单位，一个分区只属于一个进程，存在一定的浪费。

### 伙伴系统：

- 固定分区方式不够灵活，当进程大小与空闲分区大小不匹配时，内存空间利用率很低。
- 动态分区方式算法复杂，回收空闲分区时需要进行分区合并等，系统开销较大。
- 伙伴系统 (buddy system)是介于固定分区与可变分区之间的动态分区技术。
- 伙伴**：在分配存储块时将一个大的存储块分裂成两个**大小相等**的小块，这两个小块就称为“伙伴”。

□ Linux内核使用二进制伙伴算法来管理和分配物理内存页面。

1. 是否会产生内碎片？**会**
2. 是否会产生外碎片？**会**
3. 最大内碎片多大？**almost half max**
4. 如果既会产生内碎片，又会产生外碎片，那伙伴系统有什么优势？



## 3.2 内存管理 (2)

### 3.2.1 重定位

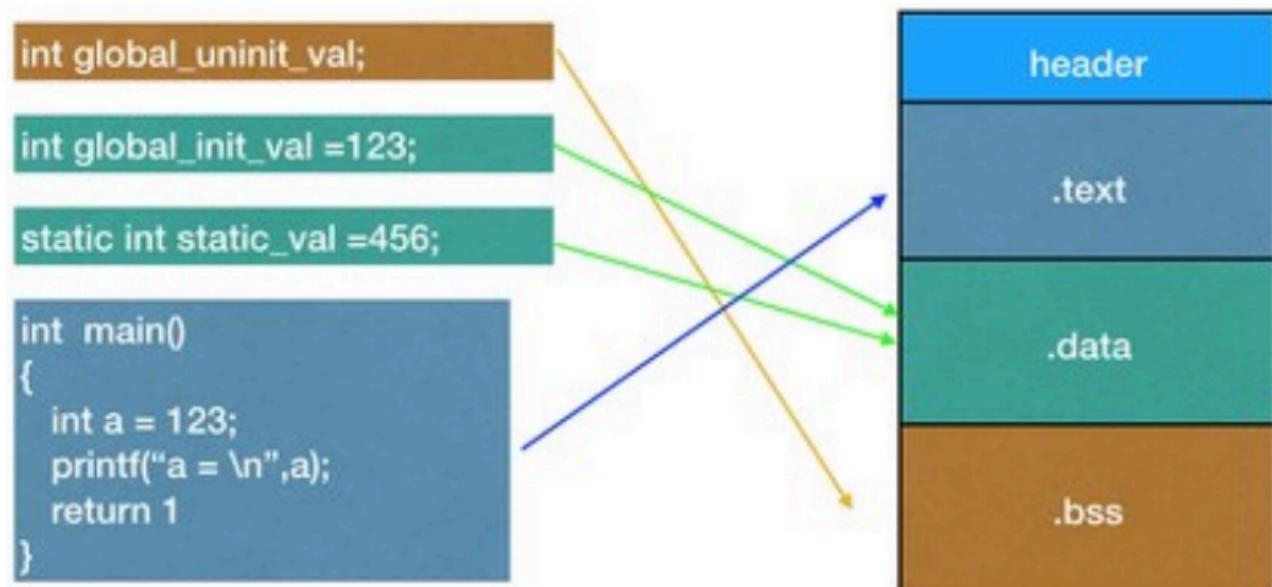
- 可重定位分区分配（紧凑）：定时的或在内存紧张时，移动某些已分配区中的信息，把存储空间中所有的空白区合并为一个大的连续区。
  - 缺点：
    - 性能开销
    - 设备依赖（DMA）
    - 间接寻址
- 程序的链接和装入
  - 一个用户源程序要变为在内存中可执行的程序，通常要进行以下处理：
    - 编译(compile)：由编译程序将用户源程序编译成若干个目标模块。
    - 链接(linking)：由链接程序将目标模块和相应的库函数链接成可装载模块（通常是单一可执行文件）。
    - 装入/loading)：由装载程序将可装载模块装入内存。
  - 程序的链接：
    - **静态链接**：用户一个工程中所需的多个程序采用静态链接的方式链接在一起。当我们希望共享库的函数代码直接链接入程序代码中，也采用静态链接方式。（占空间）
    - **动态链接**：用于链接共享库代码。当程序运行中需要某些目标模块时，才对它们进行链接，具有高效且节省内存空间的优点。但相比静态链接，使用动态链接库的程序相对慢。
  - 程序的装入：
    - 程序在内存中的位置经常要改变。程序在内存中的移动意味着它的物理位置发生了变化，这时必须对程序和数据的地址（绝对地址）进行修改后方能运行。

- 为了保证程序在内存中的位置可以改变。装入程序把装入模块装入内存后，并不立即把装入模块中相对地址转换为绝对地址，而是在程序运行时才进行。
- 这种方式需要一个重定位寄存器来支持，在程序运行过程中进行地址转换。

### 3.2.2 多重分区分配

- 多重分区分配：一个作业往往由相对独立的程序段和数据段组成，将这些片断分别装入到存储空间中不同的区域内的分配方式。
- 程序段：
  - 一个程序主要由 bss段、data段、text段三个组成的。
  - 在C语言之类的程序编译完成之后，已初始化的全局变量和静态变量保存在data 段中，未初始化的全局变量和静态变量保存在bss 段中。

段名	存储内容	生命周期
.text	程序代码（机器指令）	程序运行期间
.data	已初始化的全局/静态变量	程序运行期间
.bss	未初始化的全局/静态变量（默认为0）	程序运行期间
Heap	动态分配的内存（如 <code>malloc</code> ）	手动分配和释放
Stack	局部变量、函数调用上下文	函数调用期间



- 栈(stack)：存放、交换临时数据的内存区
  - 用户存放程序局部变量的内存区域，(但不包括static声明的变量，static意味着在数据段中存放变量)。
  - 保存/恢复调用现场。在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。
- 堆 (heap)：存放进程运行中动态分配的内存段

### 3.2.3 栈帧与调用规范

- 栈帧：

栈帧的布局示例（x86架构）



- Leaf vs non-leaf函数：

叶函数：不会调用别的函数的函数

### 3.2.4 Linux下可执行文件的格式

- 在Linux下可执行文件的格式为ELF (Executable and Linkable Format)， ELF文件分为三类：

1. 可重定位 (relocatable) 文件
2. 可执行 (executable) 文件
3. 共享object文件

- 链接的过程：

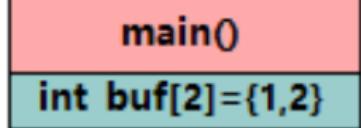
- 编译：.c→.o；编译时函数定义在不同文件，无法知道地址。
- 链接：
  - 将这些.o文件链接到一起，形成最终的可执行文件。
  - 在链接时，链接器会扫描各个目标文件，将之前未填写的地址填写上，从而生成一个真正可执行的文件。
- 链接过程的本质：

## 链接本质：合并相同的“节”

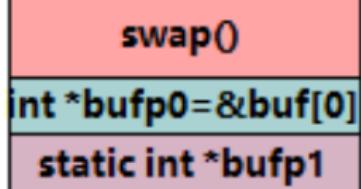
### 可重定位目标文件



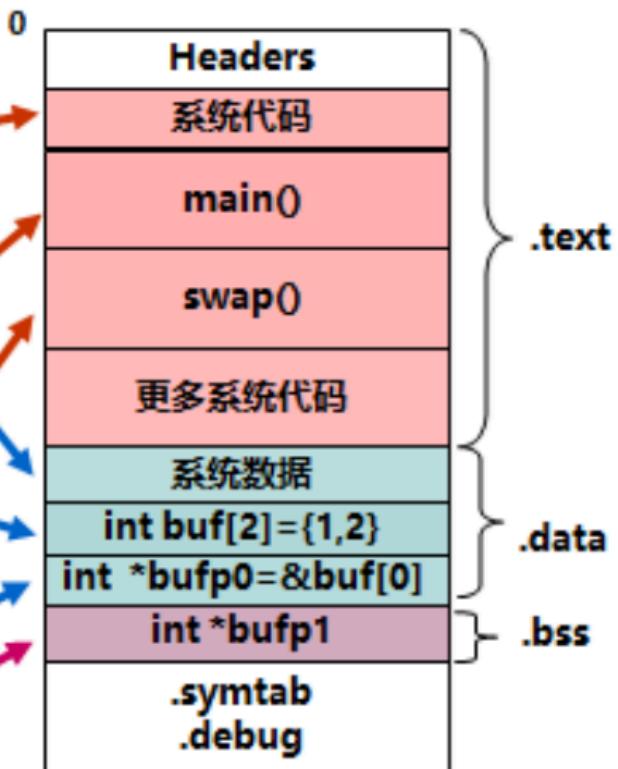
### main.o



### swap.o



### 可执行目标文件



- 重定位(Relocation):

将之前未填写的地址填写的过程。在符号解析的基础上将所有关联的目标模块合并，并确定运行时每个定义符号在虚拟地址空间中的地址，在定义符号的引用处重定位引用的地址。

### 3.2.5 程序的装载和运行

- 1. 装载前的工作:
  - shell调用fork()系统调用，创建出一个子进程。
- 2. 装载工作:
  - 子进程调用execve()加载program(即要执行的程序)。
- 3. 程序如何被加载:
  - 加载器在加载程序的时候只需要看ELF文件中和segment相关的信息即可。我们用readelf工具将segment读取出来。
- 细节:
  - 一个segment在文件中的大小是小于等于其在内存中的大小。
  - 如果在文件中的大小小于在内存中的大小，那么在载入内存时通过补零使其达到其在内存中应有的大小。

### 3.3 内存管理 (3)

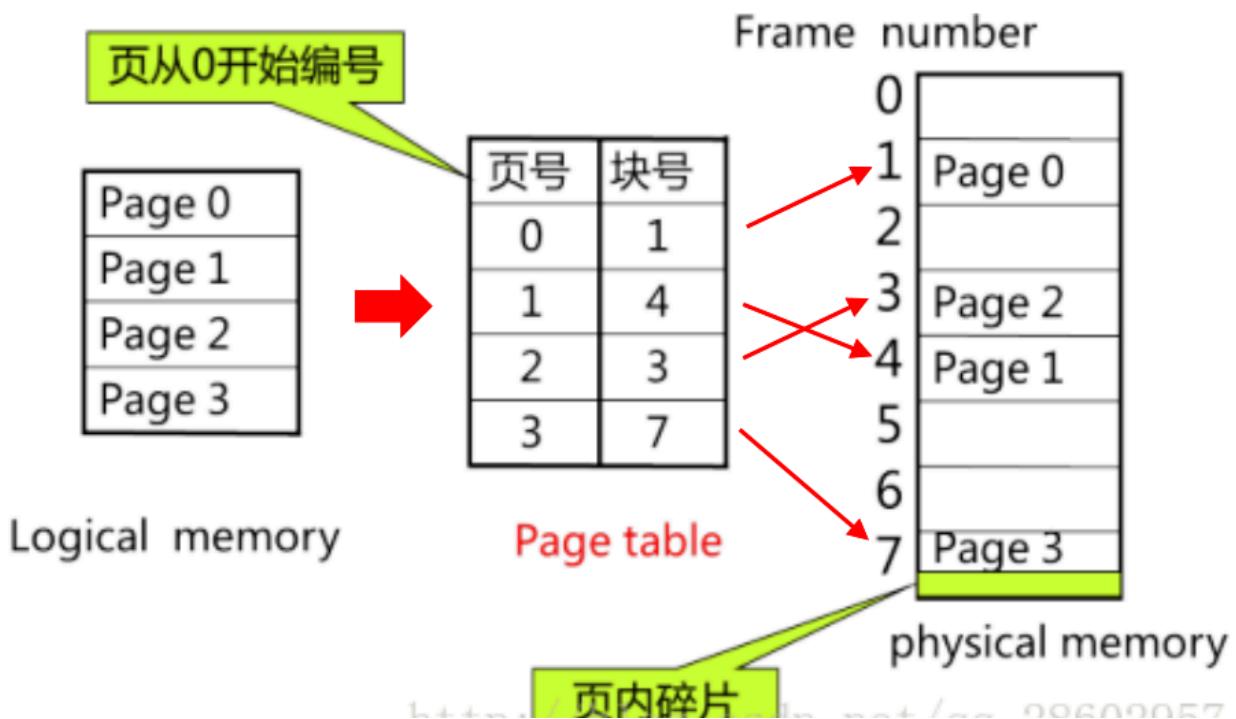
- 地址空间大小和存储空间大小没有必然联系。

### 3.3.1 分页式存储管理

- 针对问题
  - 动态内存分配
  - 碎片和紧凑问题
- 页：在分页存储管理系统中，把每个作业的地址空间分成一些大小相等的片，称之为页面或页。  
存储块：在分页存储管理系统中，把主存的存储空间也分成与页面相同大小的片，这些片称为存储块，或称为页框。

### 3.3.2 纯分页系统

- 在分页存储管理方式中，如果不具备页面置换功能，必须把它的所有页一次装到主存的页框内；如果当时页框数不足，则该作业必须等待，系统再调度另外作业。
- 优点：
  - 没有外碎片，每个内碎片不超过页大小。
  - 程序不必连续存放。便于改变程序占用空间的大小（主要指随着程序运行而动态生成的数据增多，要求地址空间相应增长，通常由系统调用完成而不是操作系统自动完成）。
- 缺点：程序全部装入内存。（时间连续性）
- 内存分配的基本思想
  - 以页为单位进行分配，并按进程（作业）的内存占用大小（页数）进行分配；
  - 逻辑上相邻的页，物理上不一定相邻。
- 数据结构——页表
  - 页表查找：

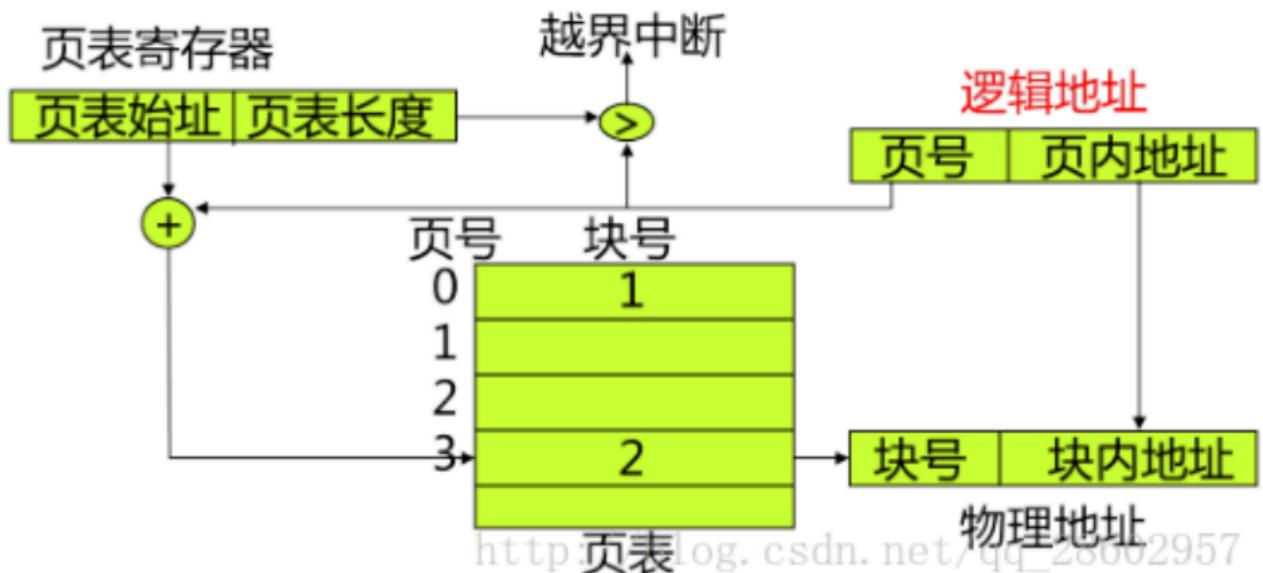


- 关于页表：
  - 页表存放在内存中，属于进程的现场信息。
  - 用途：
    1. 记录进程的内存分配情况

2. 实现进程运行时的动态重定位。

- 开销：访问一个数据需访问内存 2 次（页表一次，内存一次）。
  - 页表的基址及长度由页表寄存器给出。
- 地址变换机构：

- 当进程要访问某个逻辑地址中的数据时，分页地址变换机构会自动地将逻辑地址（有时也称有效地址、相对地址、线性地址）分为页号和页内地址两部分。
- 将页号与页表长度进行比较，如果页号大于或等于页表长度，则表示本次所访问的地址已超越进程的地址空间，产生地址越界中断。**（越界保护）**
- 将页表始址与页号和页表项长度的乘积相加，得到该表项在页表中的位置，于是可从中得到该页的物理块号，将之装入物理地址寄存器中。**（地址变换）**
- 将有效地址寄存器中的页内地址送入物理地址寄存器的块内地址字段中。

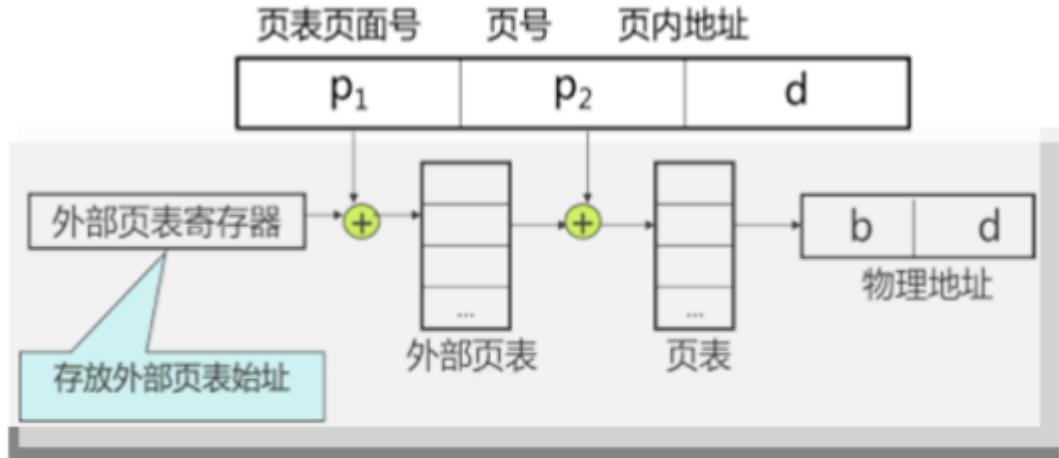


- 页面的大小的选取：

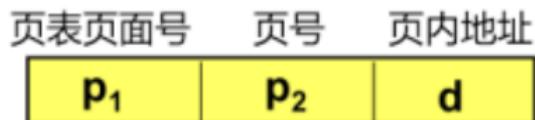
- 若页面较小
  - 优：减少页内碎片和总的内存碎片，有利于提高内存利用率。
  - 劣：每个进程页面数增多，使页表长度增加，占用内存较大。
  - 劣：页面换进换出速度将降低。（磁盘IO次数多）
- 若页面较大
  - 优：每个进程页面数减少，页表长度减少，占用内存较小。
  - 优：页面换进换出速度将提高。（磁盘IO次数少）
  - 劣：页内碎片增大，不利于提高内存利用率。

### 3.3.3 多级页表

- 二级页表：
  - 将页表再进行分页，离散地将各个页表页面存放在不同的物理块中，同时也再建立一张外部页表用以记录页表面对应的物理块号。
  - 正在运行的进程，必须把外部页表（页目录、页表的页表）调入内存，而按需动态调入内部页表。

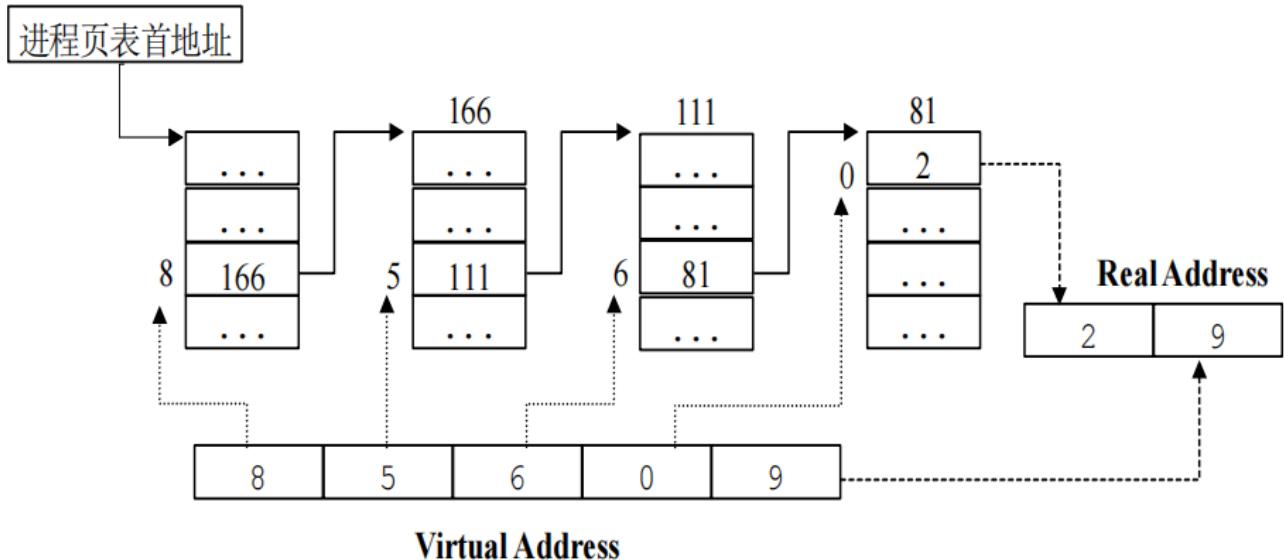


- 逻辑地址：



- 多级页表：

多级页表结构中，指令所给出的地址除偏移地址之外的各部分全是各级页表的页表号或页号，而各级页表中记录的全是物理页号，指向下属页表或真正的被访问页。



### 3.3.4 页表机制带来的访存性能问题

- 页表机制带来的严重问题就是内存访问效率的严重下降，以二级分页地址机制为例，访存次数由不分页时的 1 次，上升到了 3 次，这个问题必须解决。
- 页表快速访问机制——MMU：

为了提高地址转换效率，CPU内部增加了一个硬件单元，称为存储管理单元MMU（Memory Management Unit）。其内部主要部件：

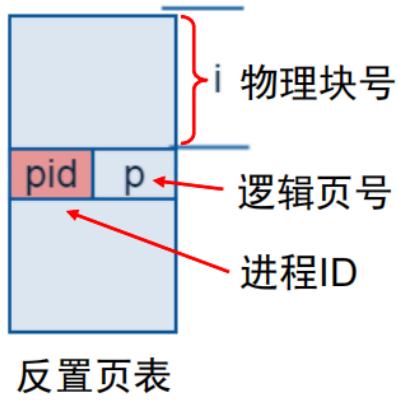
- 页表Cache：又称为TLB，用于存放虚拟地址与相应的物理地址；
- TLB控制单元：TLB内容填充、刷新、覆盖，以及越界检查。
- 页表（遍历）查找单元：若TLB未命中，自动查找多级页表，将找到的物理地址送与TLB控制单元。（可用软件实现）
- MMU的工作过程：（由于TLB的主导作用，一些OS教科书不区分MMU和TLB。）
  - MMU得到VA后先在TLB内查找，若没找到匹配的PTE条目就到外部页表查询，并置换进TLB。
  - 根据PTE条目中对访问权限的限定，检查该条VA指令是否符合，若不符合则不继续，并产生异常；
  - 符合后根据VA的地址分段查询页表，若该地址已映射到内存中（根据PTE的标识），保持offset不变，组合出物理地址，发送出去。
  - 若该地址尚未映射到内存中，则产生page fault异常。
- 快表（TLB）：专门针对页表项的高速缓存

TLB的性质和使用方法与Cache相同：

- TLB只包括表中的一小部分条目。当CPU产生逻辑地址后，其页号提交给TLB。如果页码不在TLB中（称为TLB失效），那么就需要访问内存中的页表，将页号和帧号增加到TLB中。
- 如果TLB中的条目已满，那么操作系统会选择一条替换。替换策略有很多，从最近最少使用替换（LRU）到随机替换等。
- 另外，有的TLB允许固定某些条目不被替换。通常内核代码的条目是固定下来的。

TLB的其它特性：

- 有的TLB在每个TLB条目中还保存**地址空间标识码**（address-space identifier，ASID）。ASID可用来唯一标识进程，并为进程提供地址空间保护。当TLB试图解析虚拟页号时，它确保当前运行进程的ASID与虚拟页相关的ASID相匹配。如果不匹配，那么就作为TLB失效。
- 除了提供地址空间保护外，ASID允许TLB同时包含多个进程的条目。如果TLB不支持独立的ASID，每次选择一个页表时（例如，上下文切换时），TLB就必须被冲刷（flushed）或删除，以确保下一个进程不会使用错误的地址转换。
- 反置页表：
  - 页表缺点之一是每个页表可能有很多项。这些表可能消耗大量物理内存，却仅用来跟踪物理内存是如何使用的。如每个使用32位逻辑地址的进程其页表长度均为4MB。为了解决这个问题，可以使用反向页表（inverted pagetable）。
  - 反置页表不是依据进程的逻辑页号来组织，而是依据该进程在内存中的物理页面号来组织（即：按物理页面号排列），其表项的内容是逻辑页号 P 及隶属进程标志符 pid。



反置页表

- 反置页表的大小只与物理内存的大小相关，与逻辑空间大小和进程数无关。如：64M主存，若页面大小为4kB，则反向页表只需64kB。
- 页共享与保护：

页式存储管理系统提供了两种方式：

- 地址越界保护
- 在页表中设置保护位（定义操作权限：只读，读写，执行等）

共享带来的问题：若共享数据与不共享数据划在同一块中，则：有些不共享的数据也被共享，不易保密。

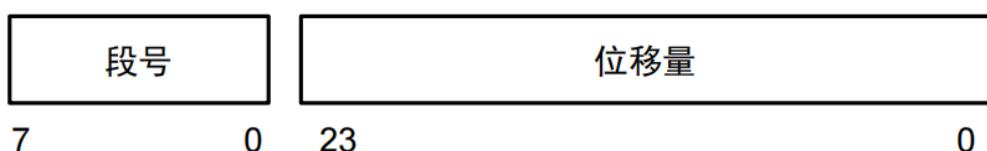
实现数据共享的最好方法：分段存储管理。

## 3.4 内存管理（4）

### 3.4.1 分段存储管理

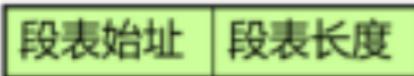
- 方便编程：通常一个作业是由多个程序段和数据段组成的，用户一般按逻辑关系对作业分段，并能根据名字来访问程序段和数据段。
- 信息共享：
  - 共享是以信息的逻辑单位为基础的。页是存储信息的物理单位，段却是信息的逻辑单位。
  - 页式管理中地址空间是一维的，主程序，子程序都顺序排列，共享公用子程序比较困难，一个共享过程可能需要几十个页面。
- 信息保护：
  - 页式管理中，一个页面中可能装有2个不同的子程序段的指令代码，不能通过页面共享实现共享一个逻辑上完整的子程序或数据块。
  - 段式管理中，可以以信息的逻辑单位进行保护。
- 地址结构：

**逻辑地址结构： 段号S + 位移量W**

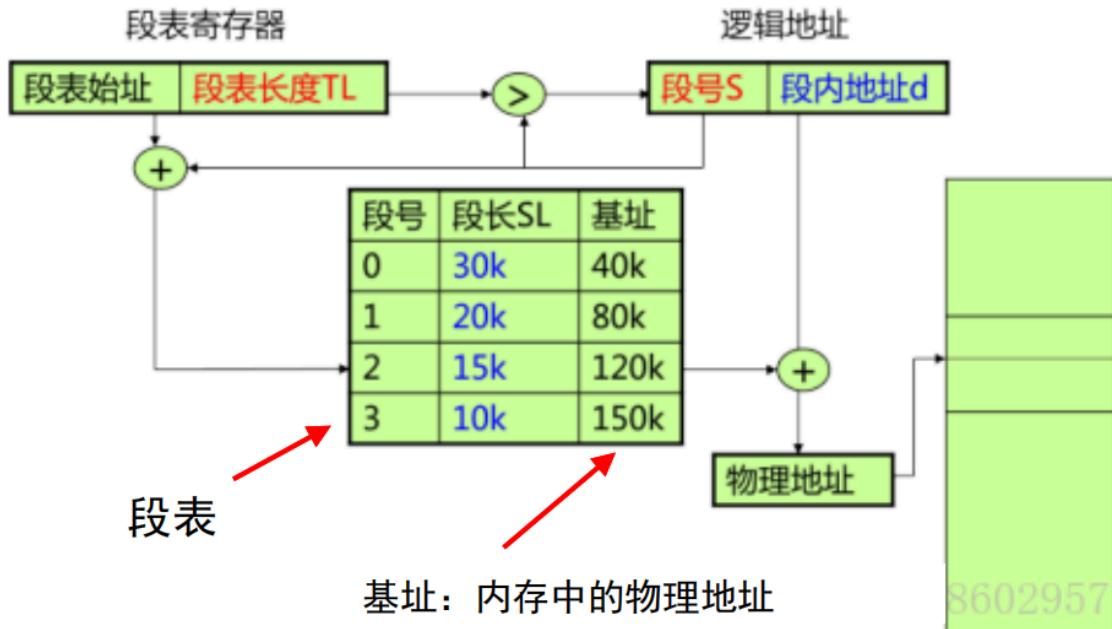


- 段表：
  - 段表记录了段与内存位置的对应关系。

- 段表保存在内存中。
- 段表的基址及长度由段表寄存器给出。



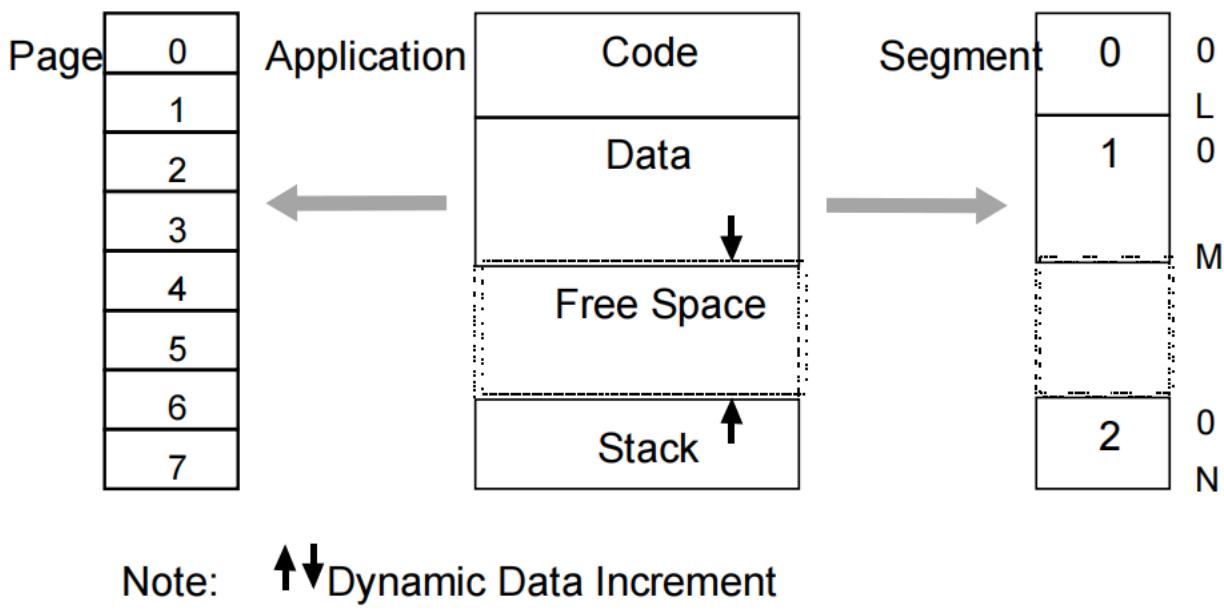
- 访问一个字节的数据/指令需访问内存两次 (段表一次, 内存一次)
- 地址变换机构:



- 地址变换过程:
  1. 系统将逻辑地址中的段号 S 与段表长度 TL 进行比较。
  2. 再检查段内地址 d, 是否超过该段的段长 SL。
- 分段管理的优缺点:
  - 优点: 分段系统易于实现段的共享, 对段的保护也十分简单。
  - 缺点:
    - 处理机要为地址变换花费时间; 要为表格提供附加的存储空间。
    - 为满足分段的动态增长和减少外碎片, 要采用拼接手段。
    - 在辅存中管理不定长度的分段困难较多。
    - 分段的最大尺寸受到主存可用空间的限制。
- 分页与分段的比较: 分页的作业的地址空间是单一的线性地址空间, 分段作业的地址空间是二维的。

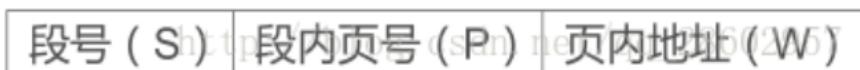
	页式存储管理	段式存储管理
目的	实现非连续分配，解决碎片问题	更好地满足用户需要
信息单位	页（物理单位）	段（逻辑单位）
大小	固定（由系统定）	不定（由用户程序定）
内存分配单位	页	段
作业地址空间	一维	二维
优点	有效解决了碎片问题（没有外碎片，每个内碎片不超过页大小）；有效提高内存的利用率；程序不必连续存放。	更好地实现数据共享与保护；段长可动态增长；便于动态链接

- 程序内存动态增长：



### 3.4.2 段页式存储管理

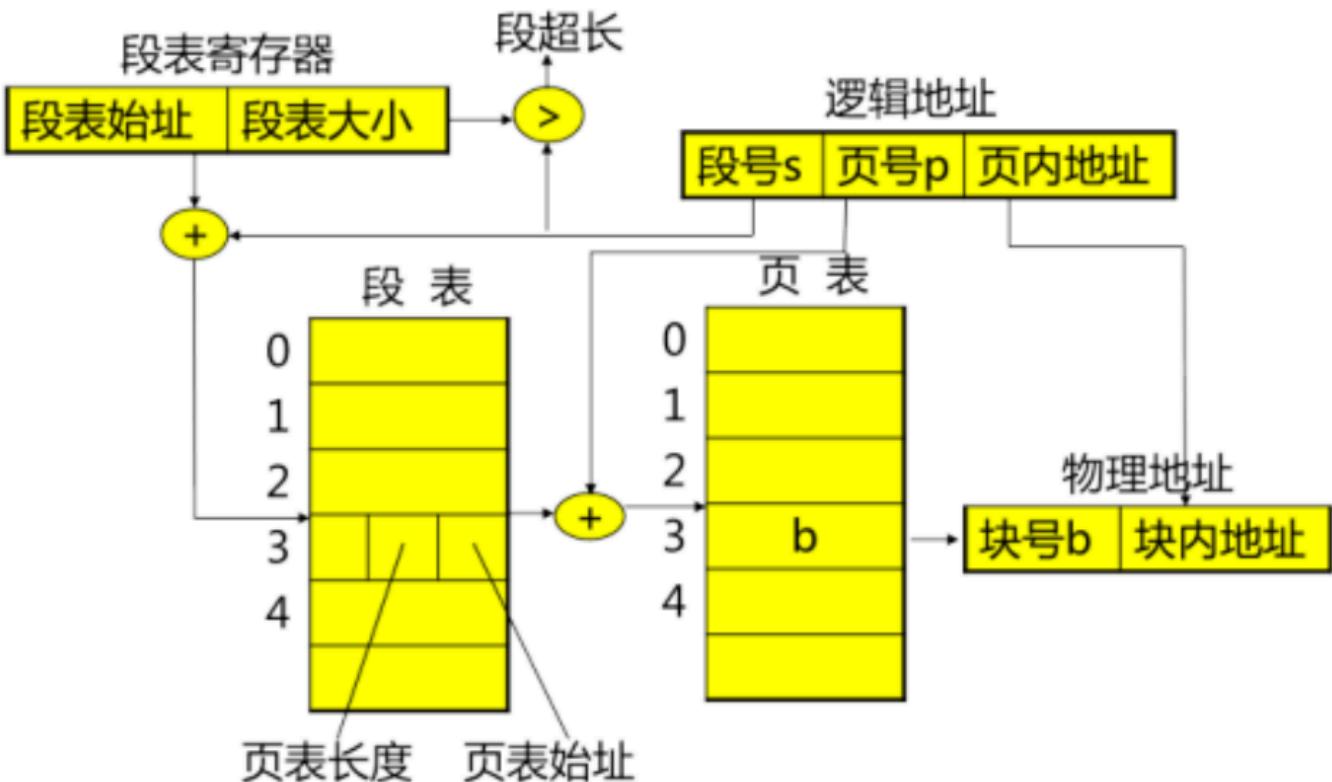
- 基本思想：用分段方法来分配和管理虚拟存储器，而用分页方法来分配和管理实存储器。
- 实现原理：
  - 段页式存储管理是分段和分页原理的结合，即先将用户程序分成若干个段（段式），并为每一个段赋一个段名，再把每个段分成若干个页（页式）。
  - 其地址结构由段号、段内页号、及页内位移三部分所组成。



- 利用段表和页表实现地址映射



- 段页式存储管理的地址变换：
  - 从 PCB 中取出段表始址和段表长度，装入段表寄存器。
  - 将段号与段表长度进行比较，若段号大于或等于段表长度，产生越界中断。
  - 利用段表始址与段号得到该段表项在段表中的位置。取出该段的页表始址和页表长度。
  - 将页号与页表长度进行比较，若页号大于或等于页表长度，产生越界中断。
  - 利用页表始址与页号得到该页表项在页表中的位置。
  - 取出该页的物理块号，与页内地址拼接得到实际的物理地址。



### 3.5 内存管理 (5)

#### 3.5.1 覆盖和交换

- 覆盖 (节约, 时间上扩展)
  - 重用内存
  - 突破内存占用空间一致性
- **覆盖：**“**覆盖**”管理，就是把一个大的程序划分成一系列的**覆盖**，每个**覆盖**是一个**相对独立的程序单位**。把程序执行时并**不要求同时装入主存**的**覆盖**组成一组，称其为**覆盖段**，这个**覆盖段**被分配到同一个存储区域。这个存储区域称之为**覆盖区**，它与**覆盖段**一一对应。
- **缺点：****编程时必须划分**程序模块和确定程序模块之间的**覆盖关系**，增加编程复杂度。从外存装入**覆盖文件**，以时间延长来换取空间节省。
- 交换 (借用，空间上扩展)
  - 辅助存储
  - 调度
  - 突破内存占用时间连续性

- **交换**: 广义的说，所谓交换就是把暂时不用的某个（或某些）程序及其数据的部分或全部从主存移到辅存中去，以便腾出必要的存储空间；接着把指定程序或数据从**辅存读到相应的主存中**，并将控制转给它，让其在系统上运行。
- **优点**: 增加并发运行的程序数目，并且给用户提供适当的响应时间；编写程序时不影响程序结构
- **缺点**: 对换入和换出的控制增加处理机开销；程序整个地址空间都进行传送，没有考虑执行过程中地址访问的**统计特性**。

### 3.5.2 局部性原理

- 指程序在执行过程中一个较短时期，所执行的指令地址和指令的操作数地址，分别局限于一定区域。还可以表现为：
  - 时间局部性，即一条指令的一次执行和下次执行，一个数据的一次访问和下次访问都集中在一个较短时期内；
  - 空间局部性，即当前指令和邻近的几条指令，当前访问的数据和邻近的数据都集中在一个较小区域内。
- 程序的局部性：
  - 程序在执行时，大部分是顺序执行的指令，少部分是转移和过程调用指令。
  - 过程调用的嵌套深度一般不超过5，因此执行的范围不超过这组嵌套的过程。
  - 程序中存在相当多的循环结构，它们由少量指令组成，而被多次执行。
  - 程序中存在相当多对一定数据结构的操作，如数组操作，往往局限在较小范围内。

### 3.5.3 虚拟存储

- 虚拟存储是计算机系统存储管理的一种技术。它为每个进程提供了一个大的、一致的、连续的可用的和私有的地址空间（一个连续完整的地址空间）。虚拟存储提供了3个能力：
  1. 给所有进程提供一致的地址空间，每个进程都认为自己是在独占使用单机系统的存储资源；
  2. 保护每个进程的地址空间不被其他进程破坏，隔离了进程的地址访问；
  3. 根据缓存原理，上层存储是下层存储的缓存，虚拟内存把主存作为磁盘的高速缓存，在主存和磁盘之间根据需要来回传送数据，高效地使用了主存；
- 虚拟存储的基本原理

- **按需装载**: 在程序装入时，不必将其全部读入到内存，而只需将当前需要执行的部分页或段读入到内存，就可让程序开始执行。
- **缺页调入**: 在程序执行过程中，如果需执行的指令或访问的数据尚未在内存（称为缺页或缺段），则由处理器通知操作系统将相应的页或段调入到内存，然后继续执行程序。
- **不用调出**: 另一方面，**操作系统将**内存中暂时不使用的页或段调出保存在外存上，从而腾出空间存放将要装入的程序以及将要调入的页或段——具有请求调入和置换功能，只需程序的一部分在内存就可执行，对于动态链接库也可以请求调入

缺页处理流程：

当进程执行过程中需访问的页面不在物理存储器中时，会引发发生缺页中断，进行所需页面换入，步骤如下：

1. 陷入内核态，保存必要的信息（OS及用户进程状态相关的信息）。（现场保护）
2. 查找出来发生页面中断的虚拟页面（进程地址空间中的页面）。这个虚拟页面的信息通常会保存在一个硬件寄存器中，如果没有的话，操作系统必须检索程序计数器，取出这条指令，用软件分析该指令，通过分析找出发生页面中断的虚拟页面。（页面定位）
3. 检查虚拟地址的有效性及安全保护位。如果发生保护错误，则杀死该进程。（权限检查）
4. 查找一个空闲的页框（物理内存中的页面），如果没有空闲页框则需要通过页面置换算法找到一个需要换出的页框。（新页面调入（1））
5. 如果找的页框中的内容被修改了，则需要将修改的内容保存到磁盘上<sup>1</sup>。（注：此时需要将页框置为忙状态，以防页框被其它进程抢占掉）（旧页面写回）
6. 页框“干净”后，操作系统将保存在磁盘上的页面内容复制到该页框中<sup>2</sup>。（新页面调入（2））
7. 当磁盘中的页面内容全部装入页框后，向操作系统发送一个中断。操作系统更新内存中的页表项，将虚拟页面映射的页框号更新为写入的页框，并将页框标记为正常状态。（更新页表）
8. 恢复缺页中断发生前的状态，将程序指针重新指向引起缺页中断的指令。（恢复现场）
9. 程序重新执行引发缺页中断的指令，进行存储访问。（继续执行）

## 3.6 内存管理（6）

### 3.6.1 页面置换策略

1. 最优置换策略（optimal page-replacement / OPT）：从主存中移出永远不再需要的页面，如无这样的页面存在，则应选择最长时间不需要访问的页面。
  - 是所有页置换算法中页错误率最低的，但它需要引用先验知识，因此无法被实现。
  - 它会将内存中的页 P 置换掉，页 P 满足：从现在开始到未来某刻再次需要页 P，这段时间最长。也就是 OPT 算法会置换掉**未来最久不被使用的页**。
  - OPT 算法通常用于比较研究，衡量其他页置换算法的效果。
2. 先进先出算法（First-in, First-out）：总选择作业中在主存驻留时间最长的一页淘汰。

- FIFO 的规则是：
    - 新页面：若页面不在队列中，则直接插入队列尾部。
    - 重复访问：若页面已在队列中，不进行任何操作（队列顺序保持不变）。
    - 淘汰规则：总是淘汰队列头部的页面（最早进入的页面）。
  - 性能较差。较早调入的页往往是经常被访问的页，这些页在FIFO 算法下被反复调入和调出。并且有Belady现象。
  - Belady现象：在使用FIFO算法作为缺页置换算法时，分配的页面增多，但缺页率反而提高，这样的异常现象称为Belady Anomaly。只有FIFO，Second Chance有。
  - 改进的FIFO算法—Second Chance：
- A是FIFO队列中最旧的页面，且其放入队列后没有被再次访问，则A被立刻淘汰；否则如果放入队列后被访问过，则将A移到FIFO队列头，并且将访问标志位清除。

### 3. Clock:

Clock是Second Chance的改进版，也称最近未使用算法(NRU, Not Recently Used)。通过一个环形队列，避免将数据在FIFO队列中移动。算法如下：

- 产生缺页错误时，当前指针指向C，如果C被访问过，则清除C的访问标志，并将指针指向D；
  - 如果C没有被访问过，则将新页面放入到C的位置，置访问标志，并将指针指向D。
  - FIFO类算法对比：
- 由于FIFO类算法命中率相比其他算法要低不少，因此实际应用中很少使用此类算法。（三个方法都会存在Belady 现象）

对比点	对比
命中率	$\text{Clock} = \text{Second Chance} > \text{FIFO}$
复杂度	$\text{Second Chance} > \text{Clock} > \text{FIFO}$
代价	$\text{Second Chance} > \text{Clock} > \text{FIFO}$

- 最近最少使用 (Least recently used)：

LRU算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

这是局部性原理的合理近似，性能接近最优算法。但由于需要记录页面使用时间的先后关系，硬件开销太大。

- 老化算法 (AGING)：LRU算法开销很大，硬件很难实现。老化算法是LRU的简化，但性能接近LRU。



### 3.6.2 工作集策略

- 基本概念：
  - 进程的工作集 (working set)：当前正在使用的页面的集合。
  - 进程的驻留集 (Resident Set)：虚拟存储系统中，每个进程驻留在内存的页面集合，或进程分到的物理页框集合。
- 工作集策略：引入工作集的目的是依据进程在过去的一段时间内访问的页面来调整常驻集大小。
 

定义：工作集是一个进程执行过程中所访问页面的集合，可用一个二元函数 $WS(k,t)$ 表示，其中：t是执行时刻；k是窗口尺寸(window size)；工作集是在 $[t - k, t]$ 时间段内所访问的页面的集合， $w(k,t) = |WS(k,t)|$  指工作集大小即页面数目；

  - 获得精确工作集的困难：
    - 工作集的过去变化未必能够预示工作集将来大小或组成页面的变化；
    - 记录工作集变化要求开销太大；
    - 对工作集窗口大小的取值难以优化，而且通常该值是不断变化的；
- 分配和置换策略：
  - 固定分配局部置换
  - 可变分配全局置换
  - 可变分配局部置换
- **抖动问题(thrashing):**
  - 随着驻留内存的进程数目增加，或者说**进程并发水平的上升**，处理器利用率先上升，然后下降。
  - 这里处理器利用率下降的原因通常称为虚拟存储器发生“抖动”，也就是：**每个进程的常驻集不断减小，缺页率不断上升，频繁调页使得调页开销增大。**
  - OS要选择一个适当的进程数目，以在并发水平和缺页率之间达到一个平衡。

- 抖动的预防与消除：

- 局部置换策略
- 引入工作集算法
- 预留部分页面
- 挂起若干进程

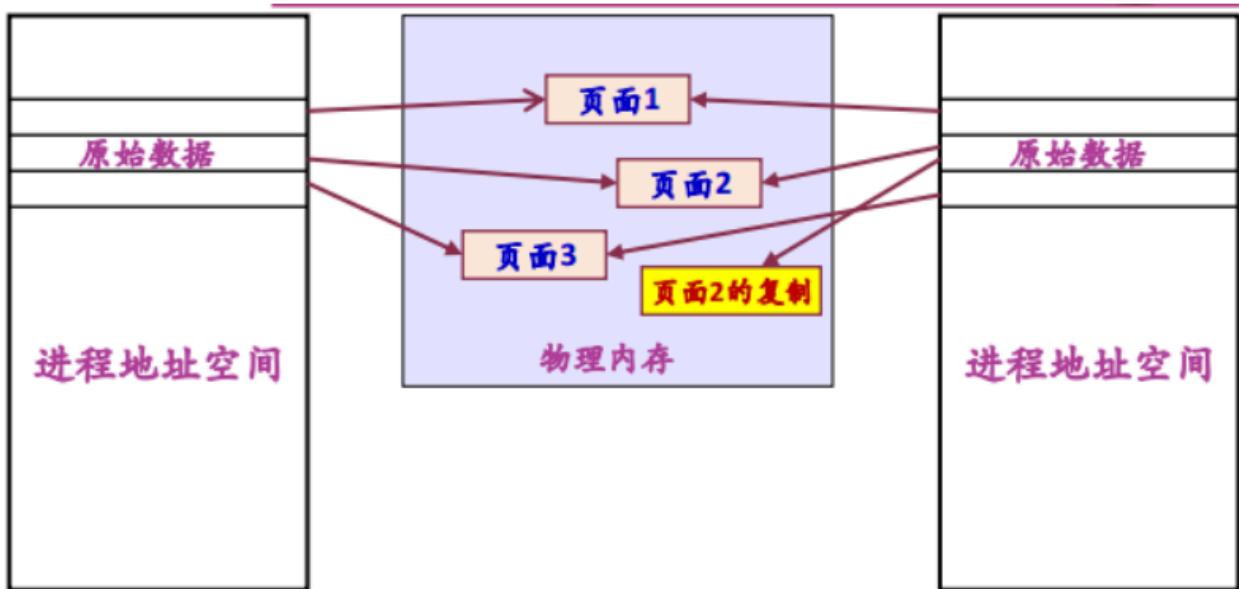
改善时间性能的途径：

- 降低缺页率：缺页率越低，虚拟存储器的平均访问时间延长得越小；
- 提高外存的访问速度：外存和内存的访问时间比值越大，则达到同样的时间延长比例，所要求的缺页率就越低；
- 高速缓存命中率。

### 2.6.3 虚拟内存其他用途

- 写时复制技术：

Linux的fork()使用写时拷贝(copy-on-write)实现，它可以**推迟甚至免除拷贝数据**的技术。内核此时并不复制整个进程地址空间，而是让父进程和子进程共享同一个拷贝。只有在需要写入的时候，数据才会复制，从而使各个进程都拥有各自的拷贝。也就是说，资源的复制只有在需要写入的时候才进行。



- 内存映射文件(Mem-Mapped File)

- 基本思想：将IO变成访存，简化读写操作，允许共享。
- 在多数实现中，在映射共享的页面时不会实际读入页面的内容，而是在访问页面时，页面才会被每次一页的读入，磁盘文件则被当作后备存储。
- 当进程退出或显式地解除文件映射时，所有被修改页面会写回文件。
- 采用内存映射方式，可方便地让多个进程共享一个文件。

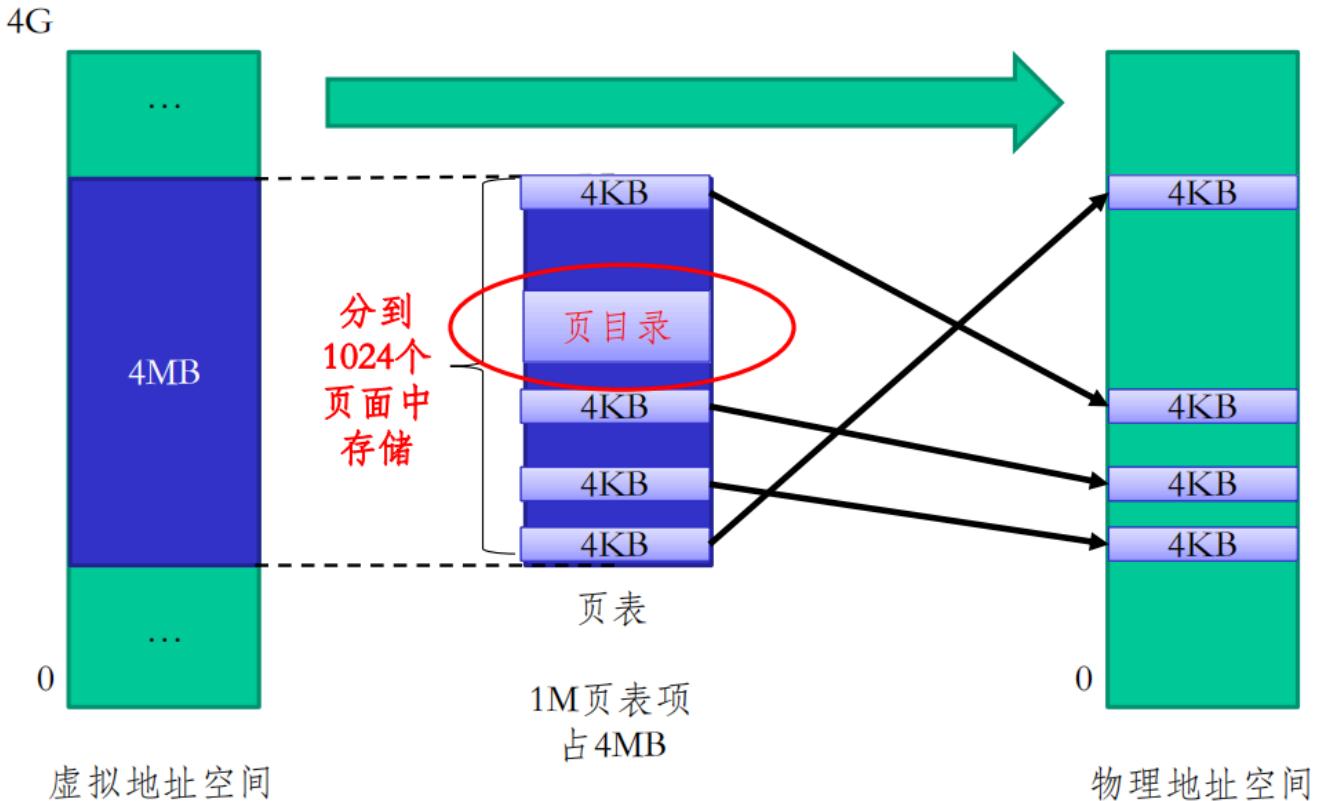
小结：

- 存储组织（层次），存储管理功能
- 重定位和装入
- 静态链接和动态链接
- 存储管理方式：单一连续区管理，分区管理（静态和动态分区）
- 覆盖，交换
- 页式和段式存储管理：原理，优缺点，数据结构，地址变换，分段的意义，两者比较
- 局部性原理，虚拟存储器的原理
- 种类（虚拟页式、段式、段页式），缺页中断
- 存储保护和共享
- 调入策略、分配策略和清除策略
- 置换策略
- 工作集策略

### 3.7 内存管理 (7)

#### 3.7.1 自映射

- 页表在虚拟地址空间中映射
  - 每个页表项需要4字节，所以1M个页表项需要4MB，所以整个页表占用的地址空间大小就是4MB
  - 4MB页表也要分页存储，共需要 $4MB/4KB=1024$ 个页面存储（页表页）
  - 每一页中存储 $4KB/4B=1024$ 项页表项
  - 由于1个页表项对应4KB内存，所以每1个页表页对应 $1024*4KB=4MB$ 内存
- 页目录中有一条PDE指向自身物理地址



- 构建方法

1. 给定一个页表基址  $PT_{base}$ , 该基址需4M对齐, 即:

$$PT_{base} = ((PT_{base}) >> 22) << 22;$$

不难看出,  $PT_{base}$  的低22位全为0。

2. 页目录基址  $PD_{base}$  在哪里?

$$PD_{base} = PT_{base} | (PT_{base}) >> 10$$

3. 自映射目录表项  $PDE_{self-mapping}$  在哪里?

$$PDE_{self-mapping} = PT_{base} | (PT_{base}) >> 10 | (PT_{base}) >> 20$$

- 特别强调:

- 只要给定4M对齐的页表基址（虚拟地址），就可以得到所有页表项对应的地址，也就包括页目录表基址和自映射页目录项在页目录表中的位置。因此页目录表基址和自映射页目录项在虚空间中是计算出来的。
- 页表主要供OS使用的，因此页表和页目录表通常放置在OS空间中（如Win的高2G空间）；
- “页目录自映射”的含义是页目录包含在页表当中，是我们采用的映射（或组织）方法的一个特征，是虚拟地址空间内的映射，与虚拟地址到物理地址的映射无关！
- 支持“页目录自映射”可节省4K（虚拟地址）空间

- 例题：

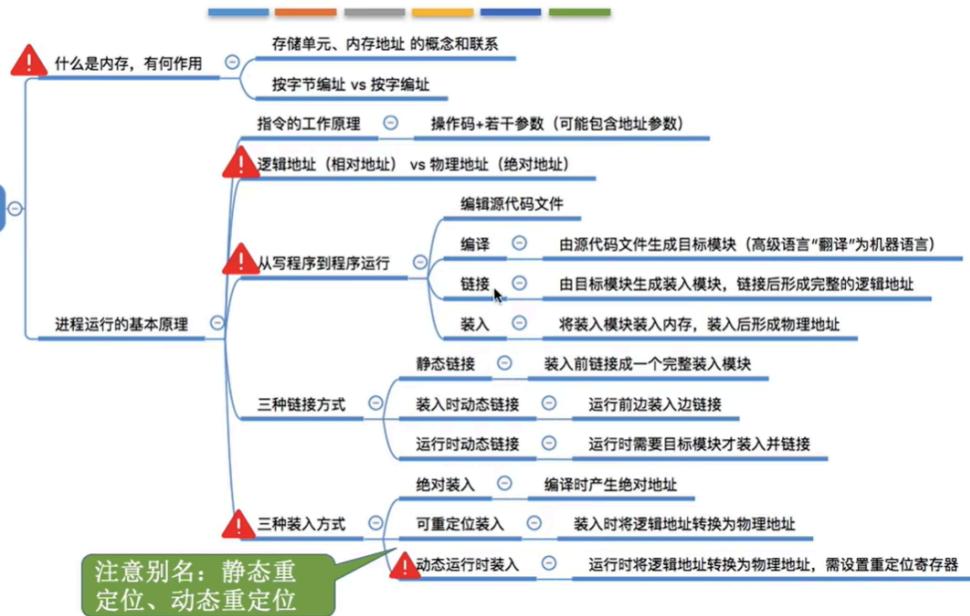
页目录在哪？（第二种理解、计算方式）

- 给定页表虚拟地址起始位置，例如 $0x7fc00000$
- 将整个4GB地址空间划分为1M个4KB页
- 上述地址对应于第 $(0x7fc00000 >> 12)$ 个4KB页，因此其逻辑-物理映射关系应该记录在第 $(0x7fc00000 >> 12)$ 个页表项中
- 每个页表项4个字节，所以该页表项对于的地址偏移为 $(0x7fc00000 >> 12) * 4 = 0x1ff000$

-OTHER-----

3.1.1\_1 内存的基础知识

## 知识回顾与重要考点



补:

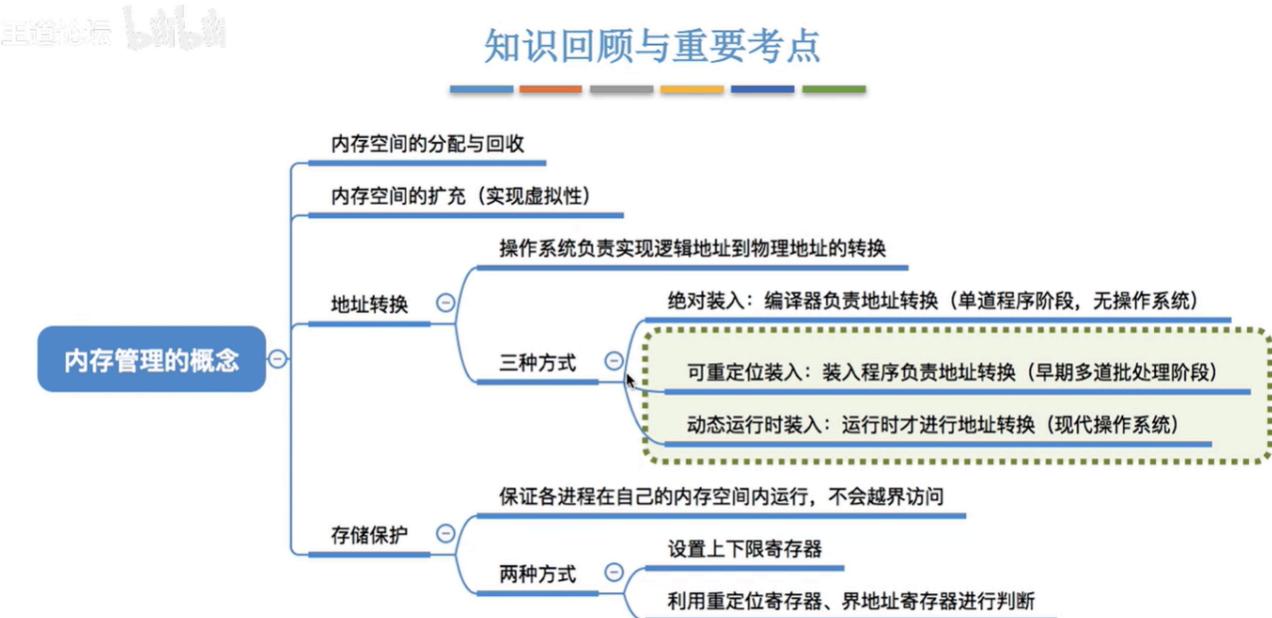
- 存储管理的基本目标:
  1. 地址独立: 程序发出的地址与物理地址无关
  2. 地址保护: 一个程序不能访问另一个程序的地址空间
- 存储管理要解决的问题: 分配和回收
- 地址空间: 源程序经过编译后得到的目标程序, 存在于它所限定的地址范围内, 这个范围称为地址空间。简言之, 地址空间是逻辑地址的集合。  
存储空间: 存储空间是指主存中一系列存储信息的物理单元的集合, 这些单元的编号称为物理地址或绝对地址。简言之, 存储空间是物理地址的集合。
- 单道程序的内存管理
 

**静态地址翻译:** 即在程序运行之前就计算出所有物理地址。静态翻译工作可以由加载器实现。

**多道程序的存储管理:** 把内存分为一些大小相等或不等的分区(partition), 每个应用程序占用一个或几个分区。操作系统占用其中一个分区。

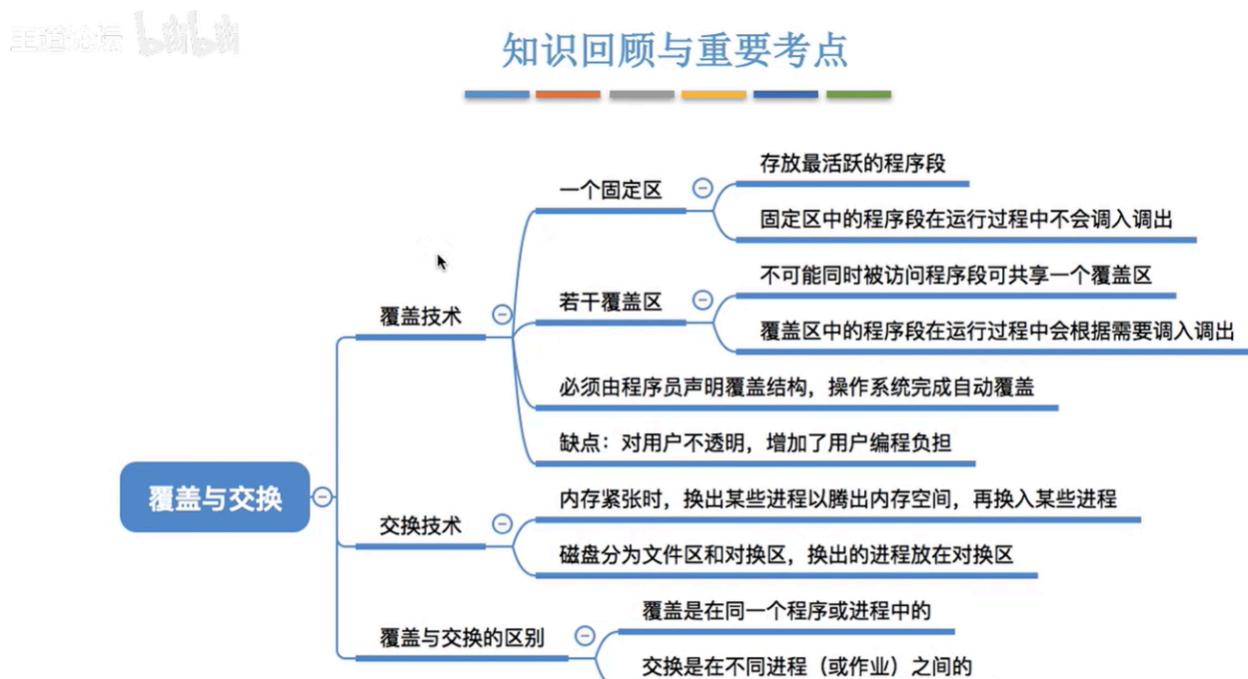
适用于多道程序系统和分时系统, 支持多个程序并发执行, 但难以进行内存分区的共享。

### 3.1.1\_2 内存管理的概念



- text和data段都在链接后的可执行文件中，由系统从可执行文件中加载，而bss段不在可执行文件中，由系统初始化
- 程序的装载：
  - 装载前的工作：shell调用fork()系统调用，创建出一个子进程。
  - 装载工作：子进程调用execve()加载program(即要执行的程序)。
- 一个segment在文件中的大小是小于等于其在内存中的大小。如果在文件中的大小小于在内存中的大小，那么在载入内存时通过补零使其达到其在内存中应有的大小。

### 3.1.1\_4 覆盖与交换



- 覆盖技术的思想：将程序分为多个段（多个模块），常用的段常驻内存，不常用的段在需要时调入内存。

- 重用内存
- 突破内存占用空间一致性

内存中分为一个“固定区”和若干个“覆盖区”。需要常驻内存的段放在“固定区”中，调入后就不再调出（除非运行结束）不常用的段放在“覆盖区”，需要用到时调入内存，用不到时调出内存。

- 缺点：编程时必须划分程序模块和确定程序模块之间的覆盖关系，增加编程复杂度。从外存装入覆盖文件，以时间延长来换取空间节省。

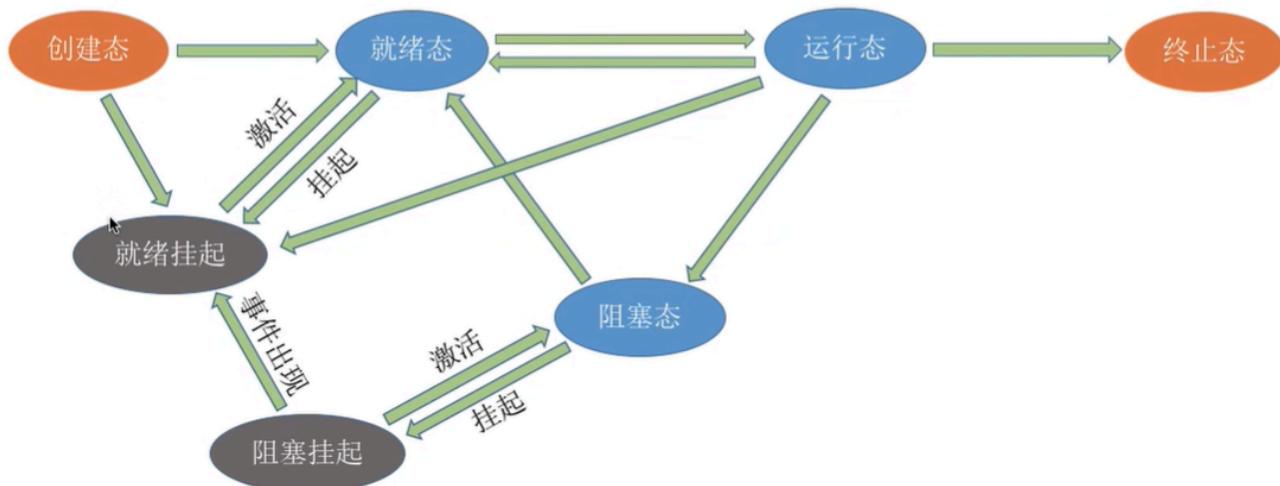
- 交换（对换）技术的设计思想：内存空间紧张时，系统将内存中某些进程暂时换出外存，把外存中某些已具备运行条件的进程换入内存（进程在内存与磁盘间动态调度）。

- 优点：增加并发运行的程序数目，并且给用户提供适当的响应时间；编写程序时不影响程序结构

- 缺点：对换入和换出的控制增加处理机开销；程序整个地址空间都进行传送，没有考虑执行过程中地址访问的统计特性。

暂时换出外存等待的进程状态为挂起状态（挂起态，suspend）

挂起态又可以进一步细分为就绪挂起、阻塞挂起两种状态



### 3.1.2\_1 连续分配方式

## 连续分配管理

### 知识回顾与重要考点



- **连续分配**: 指为用户进程分配的必须是一个连续的内存空间。
- 在**单一连续分配**方式中，内存被分为系统区和用户区。系统区通常位于内存的低地址部分，用于存放操作系统相关数据；用户区用于存放用户进程相关数据。内存中只能有一道用户程序，用户程序独占整个用户区空间。

优点：实现简单；无外部碎片；可以采用覆盖技术扩充内存；不一定需要采取内存保护。

缺点：只能用于单用户、单任务的操作系统中；有内部碎片；存储器利用率极低。

- **固定分区分配**: 把内存划分为若干个固定大小的连续分区。

优点：易于实现，开销小，无外部碎片。

缺点：内碎片造成浪费，分区总数固定，限制了并发执行的程序数目。

单一队列的分配方式、多队列分配方式

- **可变式分区**: 分区的边界可以移动，即分区的大小可变。

优点：没有内碎片。

缺点：有外碎片。

- 外部碎片才是造成内存系统性能下降的主要原因。外部碎片可以被整理后清除。消除外部碎片的方法：紧凑技术。

- 闲置空间的管理：

**位图表示法**:

- 空间成本固定：不依赖于内存中的程序数量。
- 时间成本低：操作简单，直接修改其位图值即可。
- 没有容错能力：如果一个分配单元为1，不能肯定应该为1还是因错误变成1。

**链表表示法**:

- 空间成本：取决于程序的数量。
- 时间成本：链表扫描通常速度较慢，还要进行链表项的插入、删除和修改。
- 有一定容错能力：因为链表有被占空间和闲置空间的表项，可以相互验证。

### 3.1.2\_2 动态分区分配算法

道学派

## 知识回顾与重要考点

算法	算法思想	分区排列顺序	优点	缺点
首次适应	从头到尾找适合的分区	空闲分区以地址递增次序排列	综合看性能最好。 <b>算法开销小</b> , 回收分区后一般不需要对空闲分区队列重新排序	
最佳适应	优先使用更小的分区, 以保留更多大分区	空闲分区以容量递增次序排列	会有更多的大分区被保留下来, 更能满足大进程需求	会产生很多太小的、难以利用的碎片; <b>算法开销大</b> , 回收分区后可能需要对空闲分区队列重新排序
最坏适应	优先使用更大的分区, 以防止产生太小的不可用的碎片	空闲分区以容量递减次序排列	可以减少难以利用的小碎片	大分区容易被用完, 不利于大进程; <b>算法开销大</b> (原因同上)
邻近适应	由首次适应演变而来, 每次从上次查找结束位置开始查找	空闲分区以地址递增次序排列 (可排列成循环链表)	不用每次都从低地址的小分区开始检索。 <b>算法开销小</b> (原因同首次适应算法)	会使高地址的大分区也被用完

- 外部碎片才是造成内存系统性能下降的主要原因，外部碎片可以被整理后清除（紧凑技术）
- 首次适应算法(First Fit): 优先利用内存低地址的部分空闲分区，但是出现很多外碎片。/\* 优先利用内存低地址部分的空闲分区。但由于低地址部分不断被划分，在低地址会留下许多难以利用的很小的空闲分区（碎片或零头），而每次查找又都是从低地址部分开始，增加了查找可用空闲分区的开销。 \*/

下次适应算法(Next Fit): 存储空间的利用更加均衡，不会导致小的空闲区集中在存储器的一端，但是会导致缺乏大的空闲分区。

最佳适应算法(Best Fit): 保留了大的空闲分区，留下了很多小碎片。

最坏适应算法(Worst Fit): 大作业往往得不到满足。

- 基于顺序搜索的动态分区分配算法一般只是适合于**较小的系统**，如果系统的分区很多，空闲分区表（链）可能很大（很长），检索速度会比较慢。为了提高搜索空闲分区的速度，大中型系统采用了**基于索引搜索的动态分区分配算法**。

**快速适应算法**: 又称为分类搜索法，把空闲分区按容量大小进行分类，经常用到长度的空闲区设立单独的空闲区链表。系统为多个空闲链表设立一张管理索引表。

- 优点**: 查找效率高，仅需要根据程序的长度，寻找到能容纳它的最小空闲区链表，取下第一块进行分配即可。该算法在分配时，不会对任何分区产生分割，所以能保留大的分区，也不会产生内存碎片。
- 缺点**: 在分区归还主存时算法复杂，系统开销较大。在分配空闲分区时是以进程为单位，一个分区只属于一个进程，存在一定的浪费。

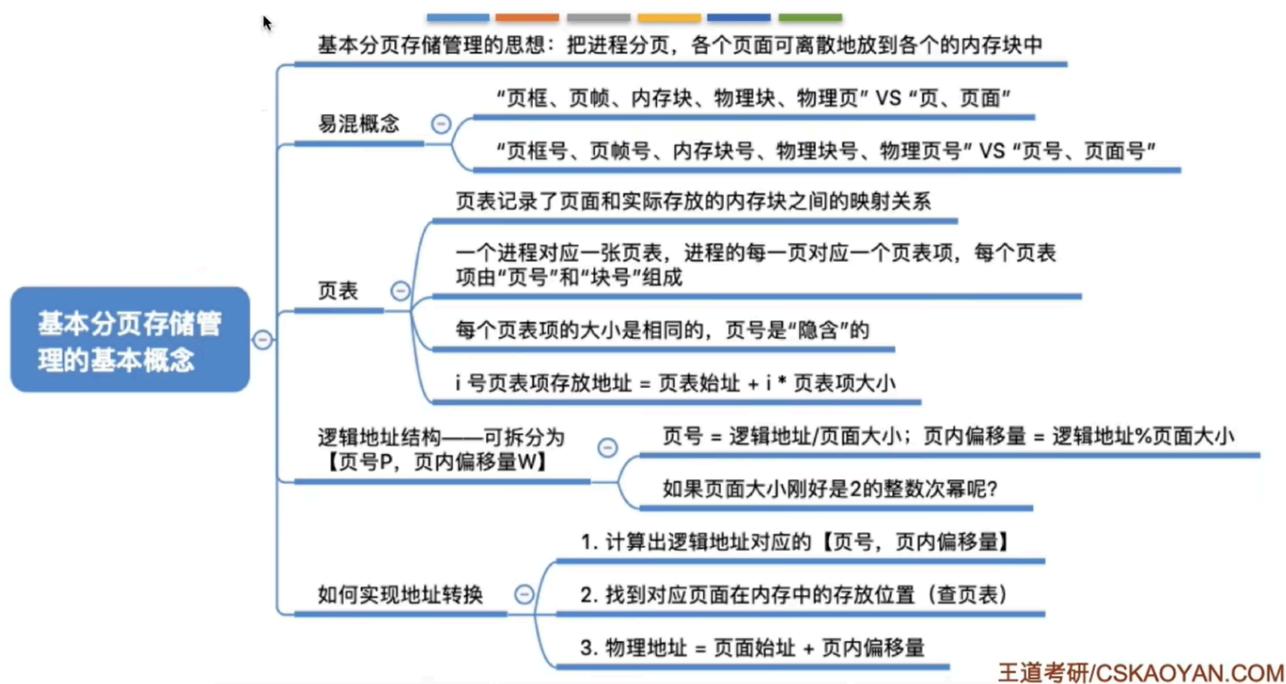
**伙伴系统**: 在分配存储块时将一个大的存储块分裂成两个大小相等的小块，这两个小块就称为“伙伴”。

- 优点**: 伙伴系统利用计算机二进制数寻址的优点，加速了相邻空闲分区的合并。
- 缺点**: 不能有效地利用内存。进程的大小不一定是2的整数倍，由此会造成浪费，内部碎片严重。

### 3.1.3\_1 基本分页存储管理的概念



## 知识回顾与重要考点



- **纯分页系统：**在分页存储管理方式中，如果不具备页面置换功能，必须把它的所有页一次装到主存的页框内；如果当时页框数不足，则该作业必须等待，系统再调度另外作业。

优点：没有外碎片，每个内碎片不超过页大小；程序不必连续存放。便于改变程序占用空间的大小（主要指随着程序运行而动态生成的数据增多，要求地址空间相应增长，通常由系统调用完成而不是操作系统自动完成）。

缺点：程序全部装入内存。（时间连续性）

- 逻辑上相邻的页，物理上不一定相邻。
- **数据结构——页表：**进程页表：每个进程有一个页表，描述该进程占用的物理页面及逻辑排列顺序；

记录映射关系：逻辑页号（本进程的地址空间） $\rightarrow$ 物理页面号（实际存储空间）

### 3.1.3\_2 基本地址变换机构

基础概念

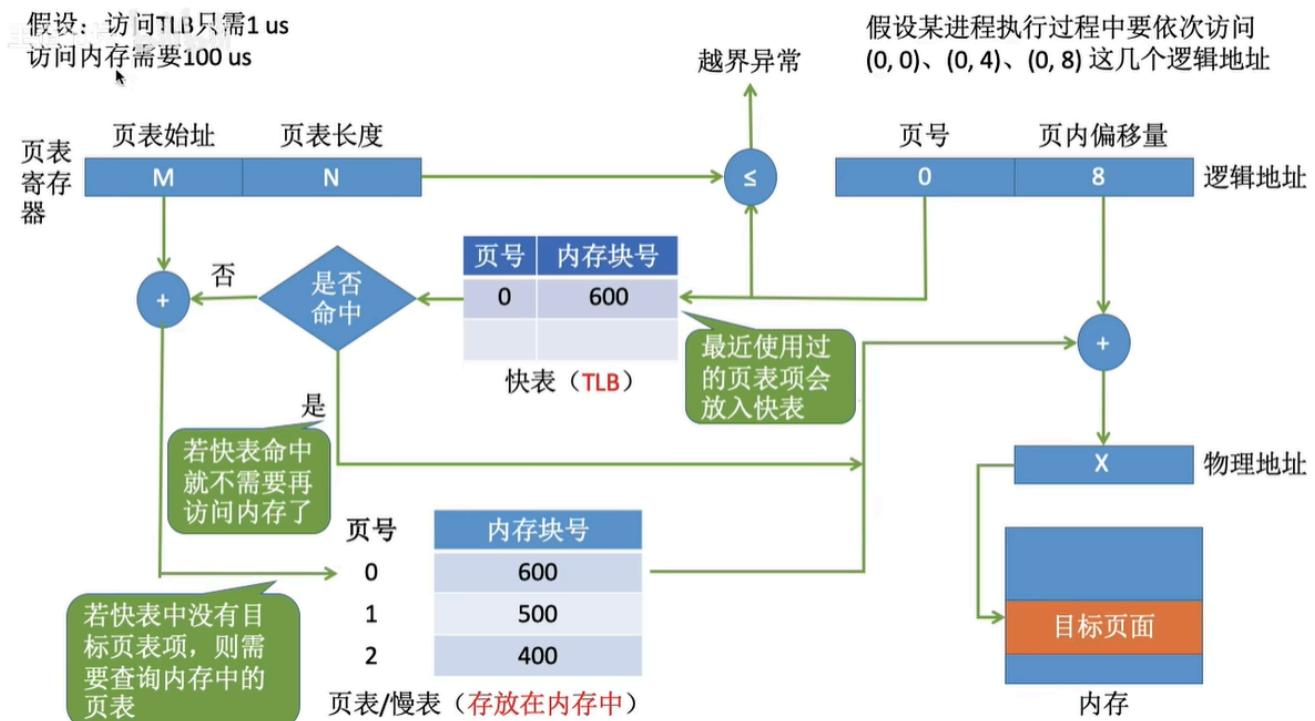
### 知识回顾与重要考点



- 基本地址变换机构可以借助进程的页表将逻辑地址转换为物理地址。
- 通常会在系统中设置一个页表寄存器 (PTR)，存放页表在内存中的起始地址F和页表长度M。进程未执行时，页表的始址和页表长度放在进程控制块 (PCB) 中，当进程被调度时，操作系统内核会把它们放到页表寄存器中。

### 3.1.3\_3 具有快表的地址变换机构

- 注意页表不是内存 (RAM) 而是高速缓存 (Cache)。



	地址变换过程	访问一个逻辑地址的访存次数
基本地址变换机构	①算页号、页内偏移量 ②检查页号合法性 ③查页表，找到页面存放的内存块号 ④根据内存块号与页内偏移量得到物理地址 ⑤访问目标内存单元	两次访存
具有快表的地址变换机构	①算页号、页内偏移量 ②检查页号合法性 ③查快表。若命中，即可知道页面存放的内存块号，可直接进行⑤；若未命中则进行④ ④查页表，找到页面存放的内存块号，并且将页表项复制到快表中 ⑤根据内存块号与页内偏移量得到物理地址 ⑥访问目标内存单元	快表命中，只需一次访存 快表未命中，需要两次访存

### 3.1.3\_3 两级页表

- 一级页表的问题：

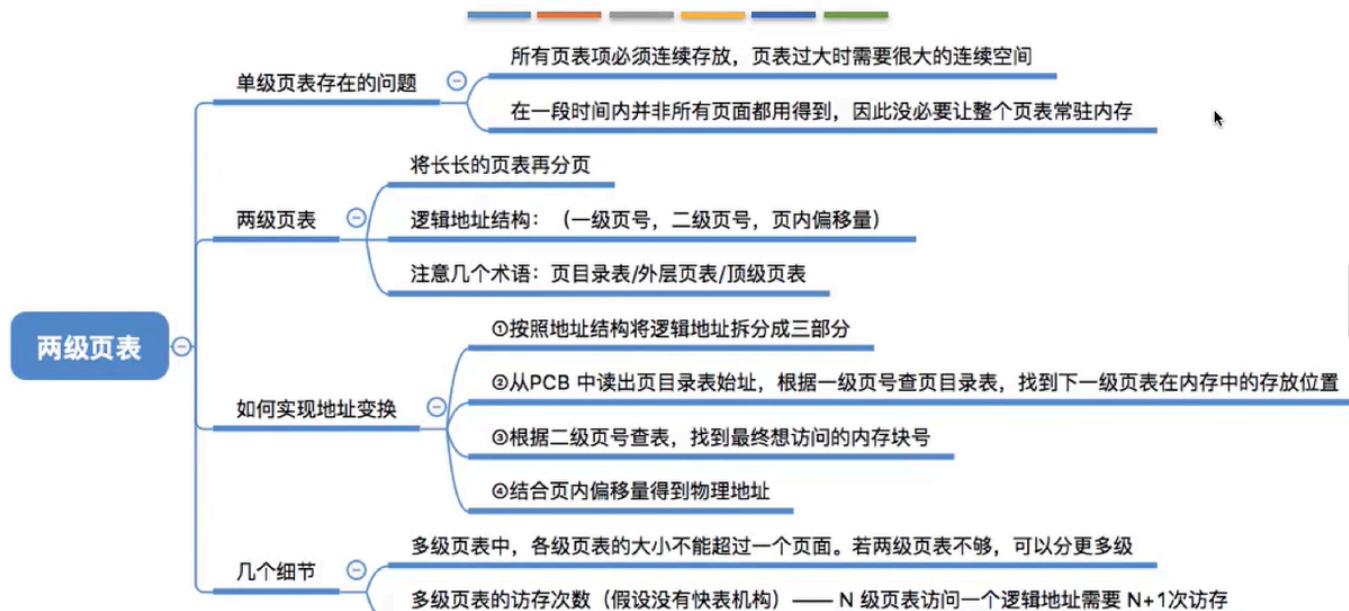
若逻辑地址空间很大 ( $2^{32} \sim 2^{64}$ )，则划分的页比较多，页表就很大，占用的存储空间大（要求连续），实现较困难。

- 若采用多级页表机制，则各级页表的大小不能超过一个页面。N级页表访问一个逻辑地址需要  $N + 1$  次访存。
- 为了提高地址转换效率，CPU内部增加了一个硬件单元，称为存储管理单元MMU（Memory Management Unit）。
- 反置页表：反置页表不是依据进程的逻辑页号来组织，而是依据该进程在内存中的物理页面号来组织（即：按物理页面号排列），其表项的内容是逻辑页号 P 及隶属进程标志符 pid。

- 页共享与保护：
  - 地址越界保护
  - 在页表中设置保护位（定义操作权限：只读，读写，执行等）

三级页表 例题分析

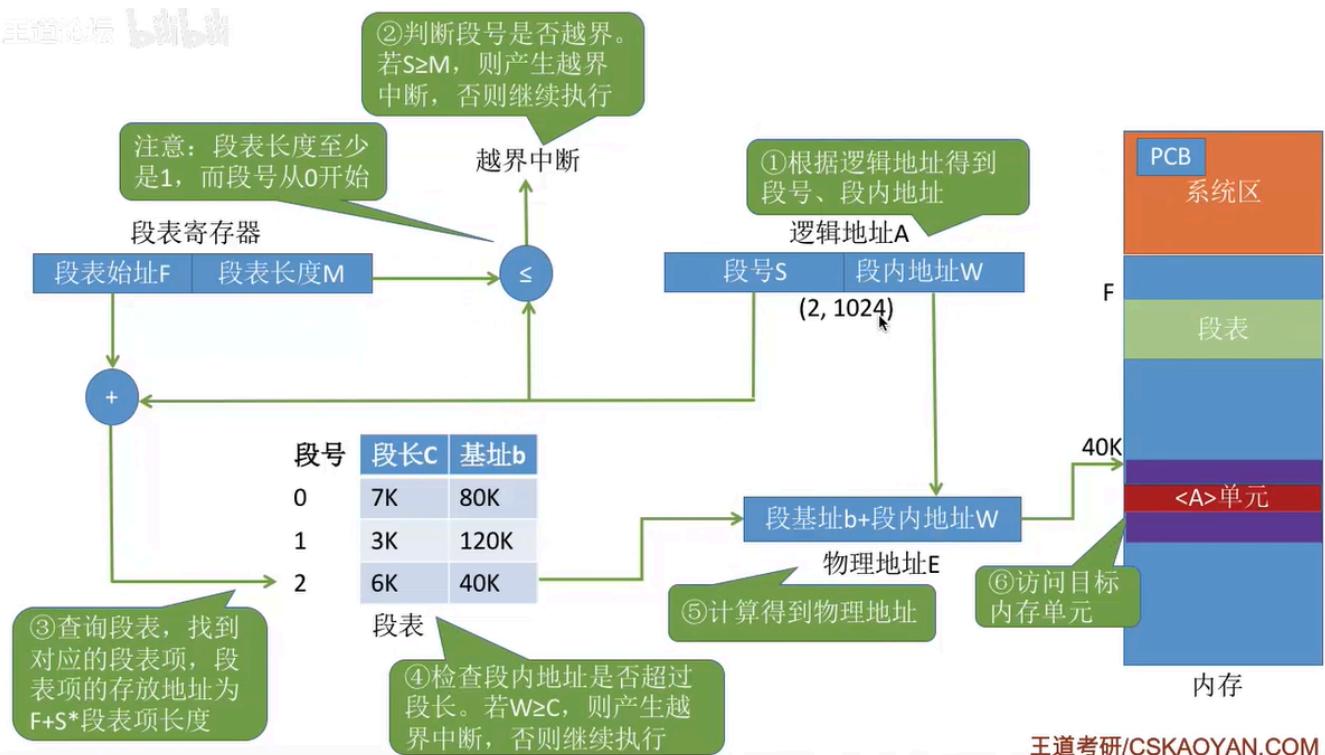
## 知识回顾与重要考点



### 3.1.4 基本分段存储管理方式

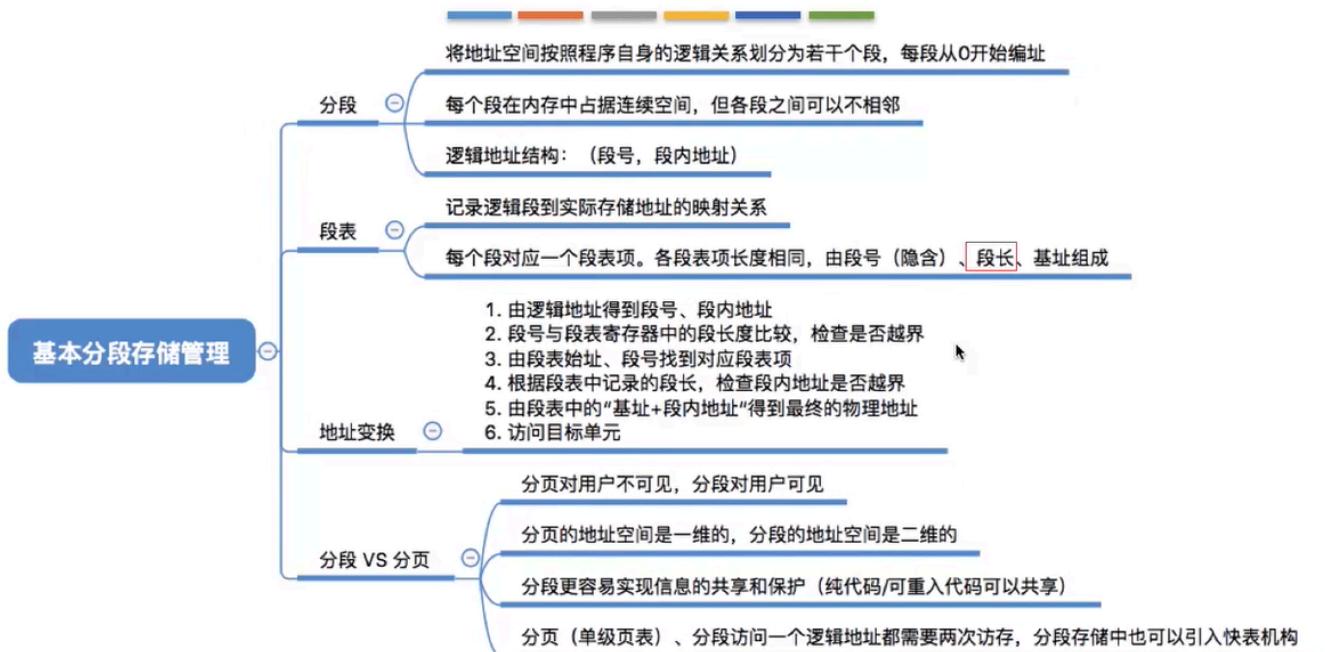
- 段号的位数决定了每个进程最多可以分几个段，段内地址位数决定了每个段的最大长度是多少。
- 程序分多个段，各段离散地装入内存，为了保证程序能正常运行，就必须能从物理内存中找到各个逻辑段的存放位置。为此，需为每个进程建立一张段映射表，简称“段表”；包含段表始址和段表长度。
- 分段管理的优缺点：
  - 优点：分段系统易于实现段的共享，对段的保护也十分简单。
  - 缺点：
    - 处理机要为地址变换花费时间；要为表格提供附加的存储空间。
    - 为满足分段的动态增长和减少外碎片，要采用拼接手段。
    - 在辅存中管理不定长度的分段困难较多。
    - 分段的最大尺寸受到主存可用空间的限制。
- 分页的作业的地址空间是单一的线性地址空间，分段作业的地址空间是二维的。

	页式存储管理	段式存储管理
目的	实现非连续分配，解决碎片问题	更好地满足用户需要
信息单位	页（物理单位）	段（逻辑单位）
大小	固定（由系统定）	不定（由用户程序定）
内存分配单位	页	段
作业地址空间	一维	二维
优点	有效解决了碎片问题（没有外碎片，每个内碎片不超过页大小）；有效提高内存的利用率；程序不必连续存放。	更好地实现数据共享与保护；段长可动态增长；便于动态链接



王道考研/CSKAOYAN.COM

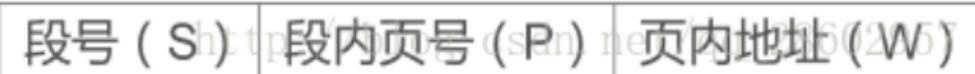
## 知识回顾与重要考点



### 3.1.5 段页式管理方式

- 实现原理：段页式存储管理是分段和分页原理的结合，即先将用户程序分成若干个段（段式），并为每一个段赋一个段名，再把每个段分成若干个页（页式）。

其地址结构由段号、段内页号、及页内位移三部分所组成。

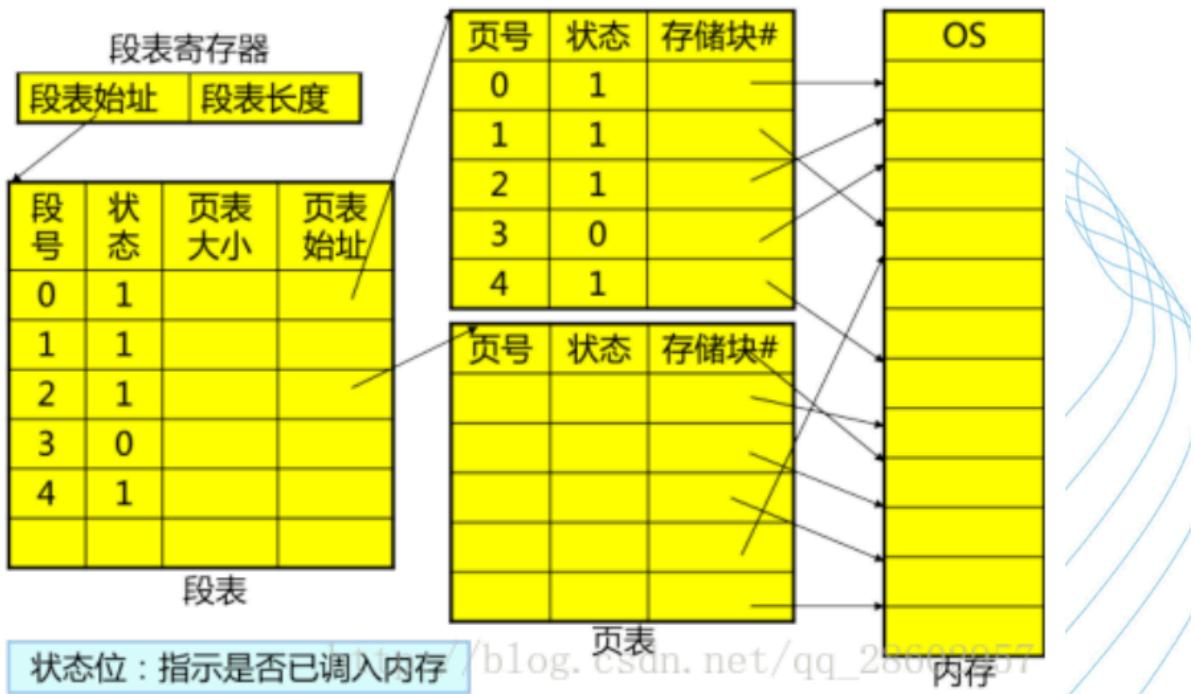


- 系统中设段表和页表，均存放于内存中。读一字节的指令或数据须访问内存三次。为提高执行速度可增设高速缓冲寄存器。

每个进程一张段表，每个段一张页表。段表含段号、页表始址和页表长度。页表含页号和块号。

# 利用段表和页表实现地址映射

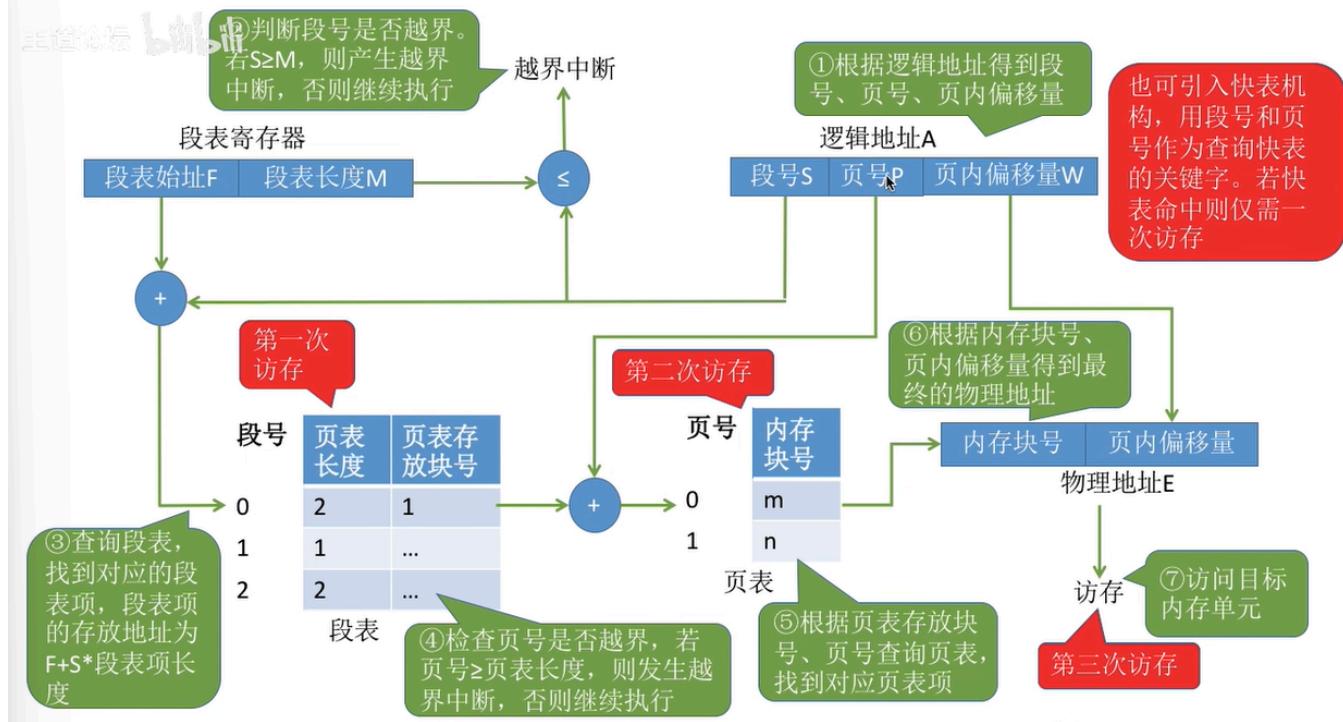
八卦



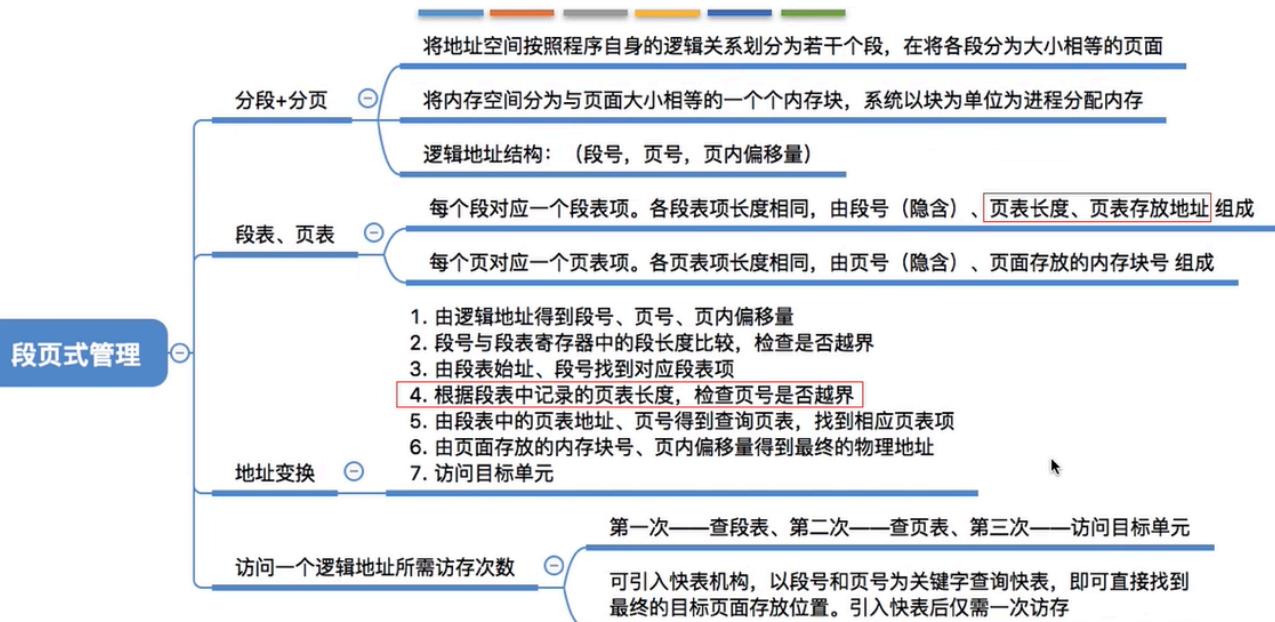
- 段页式存储管理的地址变换：

- 从 PCB 中取出段表始址和段表长度，装入段表寄存器。
- 将段号与段表长度进行比较，若段号大于或等于段表长度，产生越界中断。
- 利用段表始址与段号得到该段表项在段表中的位置。取出该段的页表始址和页表长度。
- 将页号与页表长度进行比较，若页号大于或等于页表长度，产生越界中断。
- 利用页表始址与页号得到该页表项在页表中的位置。
- 取出该页的物理块号，与页内地址拼接得到实际的物理地址。

	优点	缺点
分页管理	内存空间利用率高， <b>不会产生外部碎片</b> ，只会有少量的页内碎片	不方便按照逻辑模块实现信息的共享和保护
分段管理	很方便按照逻辑模块实现信息的共享和保护	如果段长过大，为其分配很大的连续空间会很不方便。另外，段式管理 <b>会产生外部碎片</b>



## 知识回顾与重要考点



### 3.2.1 虚拟内存的基本概念

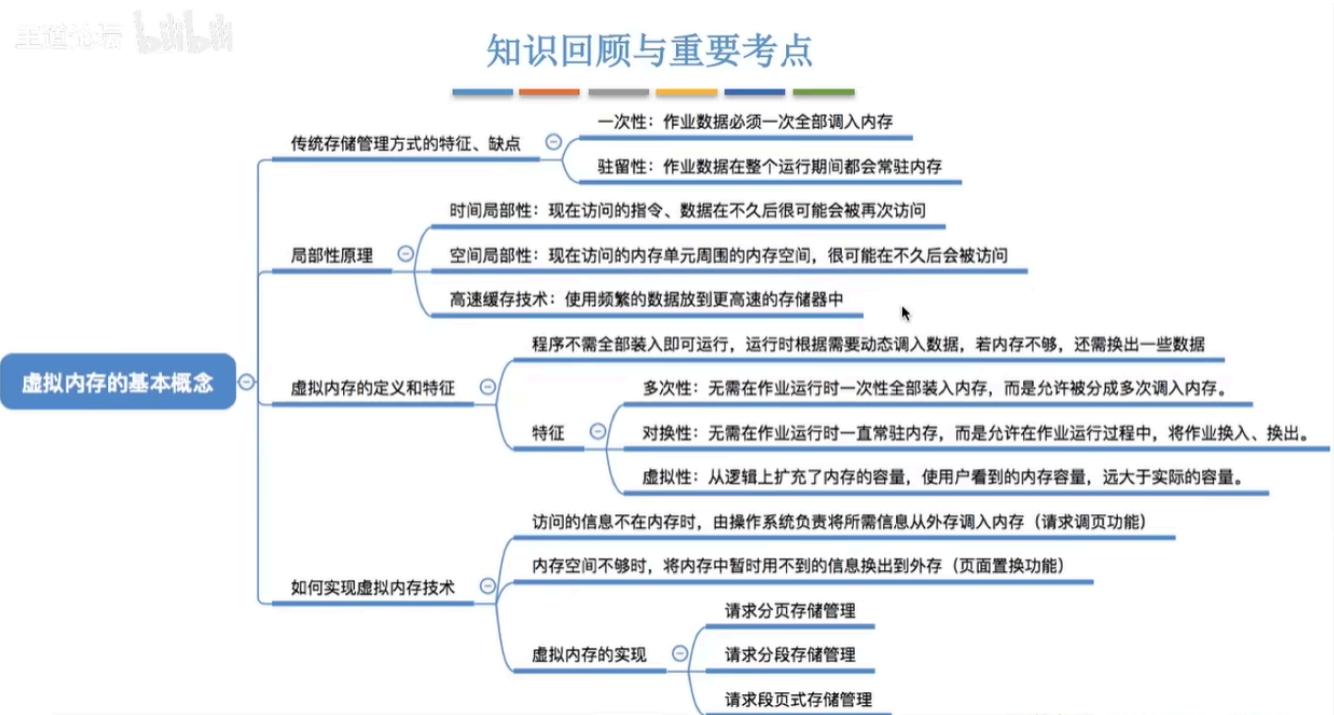
基础模块 理论知识

## 知识总览



- 局部性原理：指程序在执行过程中的一个较短时期，所执行的指令地址和指令的操作数地址，分别局限于一定区域。还可以表现为：
  - 时间局部性：如果执行了程序中的某条指令，那么不久后这条指令很有可能再次执行；如果某个数据被访问过，不久之后该数据很可能再次被访问。（因为程序中存在大量的循环）
  - 空间局部性：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很有可能被访问。（因为很多数据在内存中都是连续存放的，并且程序的指令也是顺序地在内存中存放的）
- 虚拟存储的基本原理：
  - 按需装载：在程序装入时，不必将其全部读入到内存，而只需将当前需要执行的部分页或段读入到内存，就可让程序开始执行。
  - 缺页调入：在程序执行过程中，如果需执行的指令或访问的数据尚未在内存（称为缺页或缺段），则由处理器通知操作系统将相应的页或段调入到内存，然后继续执行程序。
  - 不用调出：另一方面，操作系统将内存中暂时不使用的页或段调出保存在外存上，从而腾出空间存放将要装入的程序以及将要调入的页或段——具有请求调入和置换功能，只需程序的一部分在内存就可执行，对于动态链接库也可以请求调入
- 虚拟存储技术的特征：
  - 离散性：物理内存分配的不连续，虚拟地址空间使用的不连续（数据段和栈段之间的空闲空间，共享段和动态链接库占用的空间）
  - 多次性(分时复用)：作业被分成多次调入内存运行。正是由于多次性，虚拟存储器才具备了逻辑上扩大内存的功能。多次性是虚拟存储器最重要的特征，其它任何存储器不具备这个特征。
  - 对换性：许在作业运行过程中进行换进、换出。换进、换出可提高内存利用率。
- 优点、代价和限制：
  - 优点：
    - 可在较小的可用内存中执行较大的用户程序；

- 可在内存中容纳更多程序并发执行；
- 不必影响编程时的程序结构（与覆盖技术比较）（对用户（编程人员）透明）
- 提供给用户可用的虚拟内存空间通常大于物理内存(real memory)
- 代价：
  - 虚拟存储量的扩大是以牺牲 CPU 工作时间以及内外存交换时间为代价。
- 限制：
  - 虚拟内存的最大容量由计算机的地址结构决定。如32位机器，虚拟存储器的最大容量就是 4G，再大 CPU 无法直接访问。
- 物理内存不够的情况下，操作系统先把内存中暂时不用的数据，存到硬盘的交换空间，腾出物理内存来让别的程序运行。



### 3.3.2 请求分页管理方式



## 页表机制



与基本分页管理相比，请求分页管理中，为了实现“请求调页”，操作系统需要知道每个页面是否已经调入内存；如果还没调入，那么也需要知道该页面在外存中存放的位置。

当内存空间不够时，要实现“页面置换”，操作系统需要通过某些指标来决定到底换出哪个页面；有的页面没有被修改过，就不用再浪费时间写回外存。有的页面修改过，就需要将外存中的旧数据覆盖，因此，操作系统也需要记录各个页面是否被修改的信息。

The diagram illustrates the difference between basic page-based memory management and demand paging management. On the left, a table shows basic page-based memory management with three pages (0, 1, 2) mapped to memory blocks (a, b, c). On the right, a table shows demand paging management with three pages (0, 1, 2) mapped to memory blocks (无, b, c). The right table includes four additional fields: Status bit (状态位), Access field (访问字段), Modify bit (修改位), and External address (外存地址). Callouts explain the new fields:

- Request page table entry added four fields: "是否已调入内存" (whether it has been loaded into memory), "可记录最近被访问过几次，或记录上次访问的时间，供置换算法选择换出页面时参考" (records the number of recent accesses or the time of the last access for the page replacement algorithm to choose which page to swap out), "页面调入内存后是否被修改过" (whether it has been modified after being loaded into memory), and "页面在外存中的存放位置" (the location of the page in external storage).

页号	内存块号
0	a
1	b
2	c

基本分页存储管理的页表

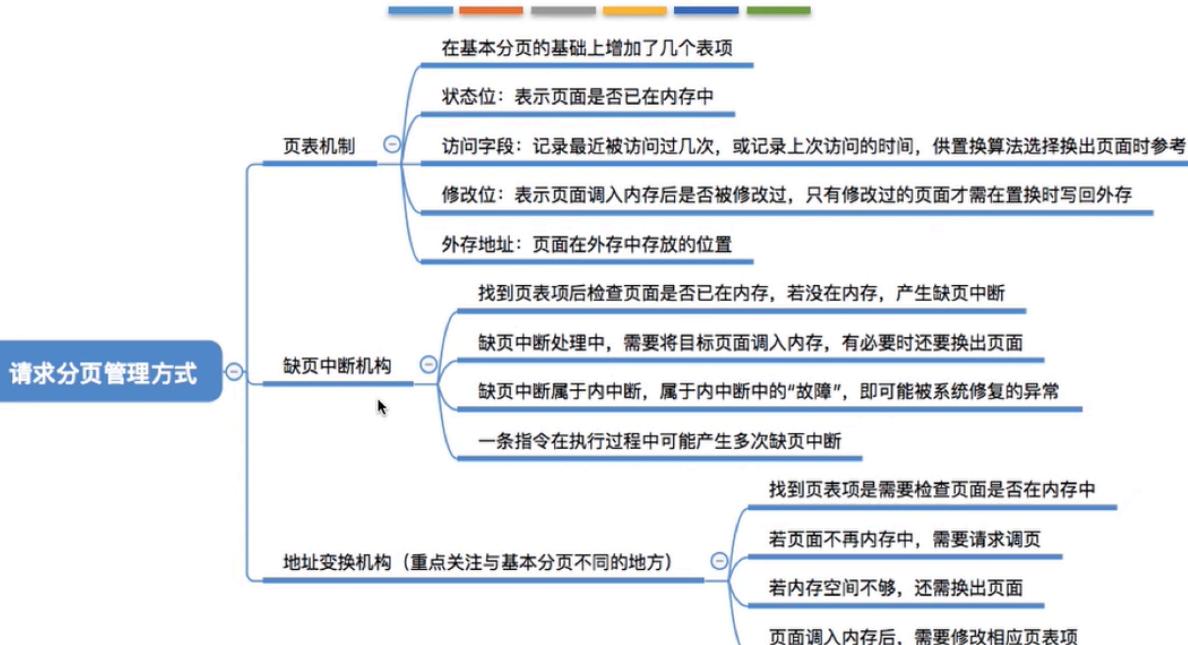
页号	内存块号	状态位	访问字段	修改位	外存地址
0	无	0	0	0	x
1	b	1	10	0	y
2	c	1	6	1	z

请求分页存储管理的页表

- 缺页中断：缺页中断是因为当前执行的指令想要访问的目标页面未调入内存而产生的，因此属于内中断
  - 在请求分页系统中，每当要访问的页面不在内存时，便产生一个缺页中断，然后由操作系统的缺页中断处理程序处理中断。  
此时缺页的进程阻塞，放入阻塞队列，调页完成后再将其唤醒，放回就绪队列。
  - 如果内存中有空闲块，则为进程分配一个空闲块，将所缺页面装入该块，并修改页表中相应的页表项。
  - 如果内存中没有空闲块，则由页面置换算法选择一个页面淘汰，若该页面在内存期间被修改过，则要将其写回外存。未修改过的页面不用写回外存。



## 知识回顾与重要考点



### 3.2.3 页面置换算法

- **最优置换算法 (OPT, Optimal)**: 每次选择淘汰的页面将是以后永不使用，或者在最长时间内不再被访问的页面，这样可以保证最低的缺页率。

虽然最佳置换算法可以保证最低的缺页率，但实际上，只有在进程执行的过程中才能知道接下来会访问到的是哪个页面。操作系统无法提前预判页面访问序列。因此，最佳置换算法是无法实现的。

- **先进先出置换算法 (FIFO)**: 每次选择淘汰的页面是最早进入内存的页面。

**Belady** 异常——在使用FIFO算法作为缺页置换算法时，分配的页面增多，但缺页率反而提高。

只有FIFO算法会产生 **Belady** 异常。另外，FIFO算法虽然实现简单，但是该算法与进程实际运行时的规律不适应，因为先进入的页面也有可能最经常被访问。因此，算法性能差。

- **最近最久未使用置换算法 (LRU, least recently used)**: 每次淘汰的页面是最近最久未使用的页面。

实现方法：赋予每个页面对应的页表项中，用访问字段记录该页面自上次被访问以来所经历的时间t。当需要淘汰一个页面时，选择现有页面中t值最大的，即最近最久未使用的页面。

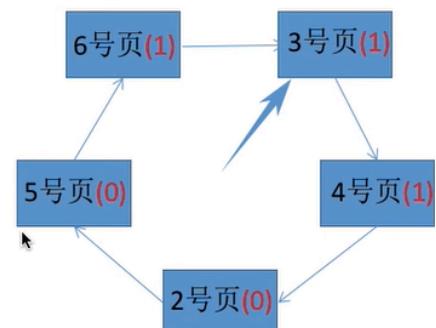
该算法的实现需要专门的硬件支持，虽然算法性能好，但是实现困难，开销大。

- 最佳置换算法性能最好，但无法实现；先进先出置换算法实现简单，但算法性能差；最近最久未使用置换算法性能好，是最接近OPT算法性能的，但是实现起来需要专门的硬件支持，算法开销大。所以提出了**时钟置换算法 (CLOCK)**

**简单的CLOCK算法**实现方法：为每个页面设置一个**访问位**，再将内存中的页面都通过链接指针**链接成一个循环队列**。当某页被访问时，其访问位置为1。当需要淘汰一个页面时，只需检查页的访问位。如果是0，就选择该页换出；如果是1，则将它置为0，暂不换出，继续检查下一个页面，若第一轮扫描中所有页面都是1，则将这些页面的访问位依次置为0后，再进行第二轮扫描（第二轮扫描中一定会有访问位为0的页面，因此**简单的CLOCK算法**选择一个淘汰页面**最多会经过两轮扫描**）



例：假设系统为某进程分配了**五个**内存块，并考虑到有以下页面号引用串：  
1, 3, 4, 2, 5, 6, 3, 4, 7



Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

- 简单的时钟置换算法仅考虑到一个页面最近是否被访问过。事实上，如果被淘汰的页面没有被修改过，就不需要执行I/O操作写回外存。只有被淘汰的页面被修改过时，才需要写回外存。

因此，除了考虑一个页面最近有没有被访问过之外，操作系统还应考虑页面有没有被修改过。在其他条件都相同时，应优先淘汰没有修改过的页面，避免I/O操作。这就是改进型的时钟置换算法的思想。修改位=0，表示页面没有被修改过；修改位=1，表示页面被修改过。**改进型的时钟置换算法**

**简单的时钟置换算法**仅考虑到一个页面最近是否被访问过。事实上，如果被淘汰的页面没有被修改过，就不需要执行I/O操作写回外存。**只有被淘汰的页面被修改过时，才需要写回外存。**

因此，除了考虑一个页面最近有没有被访问过之外，操作系统还应考虑页面有没有被修改过。**在其他条件都相同时，应优先淘汰没有修改过的页面，避免I/O操作。**这就是改进型的时钟置换算法的思想。

**修改位=0**，表示页面没有被修改过；**修改位=1**，表示页面被修改过。

为方便讨论，用**(访问位, 修改位)**的形式表示各页面状态。如**(1, 1)**表示一个页面近期被访问过，且被修改过。

**算法规则：**将所有可能被置换的页面排成一个循环队列

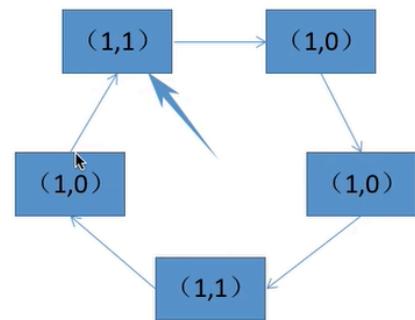
第一轮：从当前位置开始扫描到第一个**(0, 0)**的帧用于替换。本轮扫描不修改任何标志位

第二轮：若第一轮扫描失败，则重新扫描，查找第一个**(0, 1)**的帧用于替换。本轮将所有扫描过的帧访问位设为0

第三轮：若第二轮扫描失败，则重新扫描，查找第一个**(0, 0)**的帧用于替换。本轮扫描不修改任何标志位

第四轮：若第三轮扫描失败，则重新扫描，查找第一个**(0, 1)**的帧用于替换。

由于第二轮已将所有帧的访问位设为0，因此经过第三轮、第四轮扫描一定会有一页被选中，因此**改进型CLOCK置换算法**选择一个淘汰页面**最多会进行四轮扫描**



	算法规则	优缺点
OPT	优先淘汰最长时间内不会被访问的页面	缺页率最小，性能最好；但无法实现
FIFO	优先淘汰先进入内存的页面	实现简单；但性能很差，可能出现Belady异常
LRU	优先淘汰最近最久没访问的页面	性能很好；但需要硬件支持，算法开销大
CLOCK (NRU)	循环扫描各页面 第一轮淘汰访问位=0的，并将扫描过的页面访问位改为1。若第一轮没选中，则进行第二轮扫描。	实现简单，算法开销小；但未考虑页面是否被修改过。
改进型CLOCK（改进型NRU）	若用 <b>(访问位, 修改位)</b> 的形式表述，则 第一轮：淘汰 <b>(0, 0)</b> 第二轮：淘汰 <b>(0, 1)</b> ，并将扫描过的页面访问位都置为0 第三轮：淘汰 <b>(0, 0)</b> 第四轮：淘汰 <b>(0, 1)</b>	算法开销较小，性能也不错

### 3.2.4 页面分配策略

- 进程的工作集：指在某段时间间隔里，进程实际访问页面的集合。/当前正在使用的页面的集合。
- 驻留集：指请求分页存储管理中给进程分配的物理块的集合。在采用了虚拟存储技术的系统中，驻留集大小一般小于进程的总大小。/虚拟存储系统中，每个进程驻留在内存的页面集合，或进程分配到的物理页框集合。

若驻留集太小，会导致缺页频繁、系统要花大量的时间来处理缺页，实际用于进程推进的时间很少；驻留集太大，又会导致多道程序并发度下降，资源利用率降低。所以应该选择一个合适的驻留集大小。

- **固定分配**: 操作系统为每个进程分配一组固定数目的物理块，在进程运行期间不再改变。即，驻留集大小不变  
**可变分配**: 先为每个进程分配一定数目的物理块，在进程运行期间，可根据情况做适当的增加或减少。即，驻留集大小可变
- **局部置换**: 发生缺页时只能选进程自己的物理块进行置换。  
**全局置换**: 可以将操作系统保留的空闲物理块分配给缺页进程，也可以将别的进程持有的物理块置换到外存，再分配给缺页进程。
- **固定分配局部置换**: 系统为每个进程分配一定数量的物理块，在整个运行期间都不改变。若进程在运行中发生缺页，则只能从该进程在内存中的页面中选出一页换出，然后再调入需要的页面。这种策略的缺点是：很难在刚开始就确定应为每个进程分配多少个物理块才算合理。（采用这种策略的系统可以根据进程大小、优先级、或是根据程序员给出的参数来确定为一个进程分配的内存块数）

**可变分配全局置换**: 刚开始会为每个进程分配一定数量的物理块。操作系统会保持一个空闲物理块队列。当某进程发生缺页时，从空闲物理块中取出一块分配给该进程；若已无空闲物理块，则可选择一个未锁定的页面换出外存，再将该物理块分配给缺页的进程。采用这种策略时，**只要某进程发生缺页，都将获得新的物理块**，仅当空闲物理块用完时，系统才选择一个未锁定的页面调出。被选择调出的页可能是系统中任何一个进程中的页，因此这个被选中的进程拥有的物理块会减少，缺页率会增加。

**可变分配局部置换**: 刚开始会为每个进程分配一定数量的物理块。当某进程发生缺页时，只允许从该进程自己的物理块中选出一个进行换出外存。如果进程在运行中频繁地缺页，系统会为该进程多分配几个物理块，直至该进程缺页率趋势适当程度；反之，如果进程在运行中缺页率特别低，则可适当减少分配给该进程的物理块。

注：

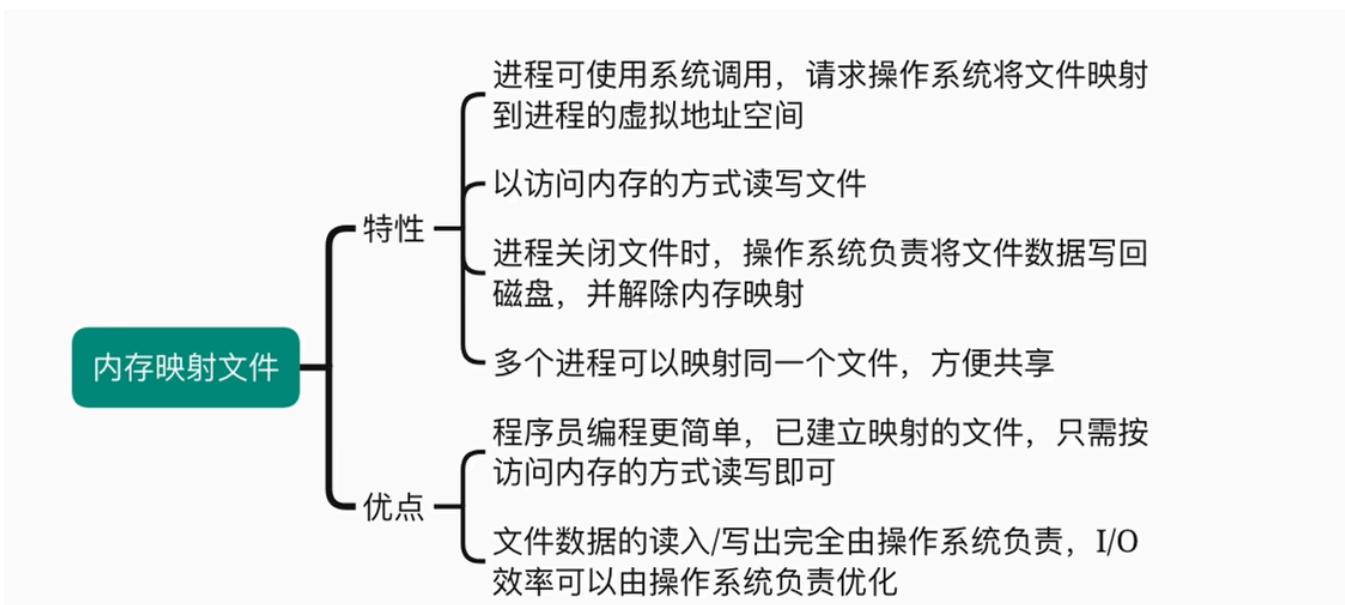
- 可变分配全局置换：只要缺页就给分配新物理
- 可变分配局部置换：要根据发生缺页的频率来动态地增加或减少进程的物理块
- **抖动（颠簸）现象**: 刚刚换出的页面马上又要换入内存，刚刚换入的页面马上要换出外存，这种频繁的页面调度行为称为抖动，或颠簸。产生抖动的主要原因是进程频繁访问的页面数目高于可用的物理块数（分配给进程的物理块不够）
- 抖动问题：随着驻留内存的进程数目增加，处理器的利用率先上升后下降
  - 消除与预防：
    - 局部置换策略
    - 引入工作集算法
    - 预留部分页面
    - 挂起若干进程

## 知识回顾与重要考点



### 3.2.5 内存映射文件

- 内存映射文件——操作系统向上层程序员提供的功能 (系统调用)
  - 方便程序员访问文件数据
  - 方便多个进程共享同一个文件



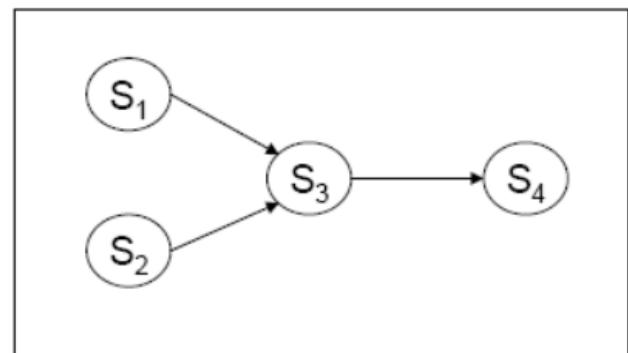
## 4. 进程与并发程序设计

### 4.1 进程与线程

#### 4.1.1 进程的概念引入

- 并发与并行的辨析：
  - **并发Concurrent**: 设有两个活动 $a_1$ 和 $a_2$ , 如果在某一指定的时间 $t$ , 无论 $a_1$ 和 $a_2$ 是在同一处理机上还是在不同的处理机上执行, 只要 $a_1$ 和 $a_2$ 都处在各自的起点和终点之间的某一处, 则称 $a_1$ 和 $a_2$ 是并发执行的。
  - **并行Parallel**: 如果考虑两个程序, 它们在同一时间度量下同时运行在不同的处理机上, 则称这两个程序是并行执行的。
  - 并发可能是伪并行, 也可能是真并行。
- 程序的顺序执行与特征：顺序性、封闭性、可再现性。
- 程序并发执行时的特征：间断性、非封闭性、不可再现性。
- 竞争：
  - 竞争：多个进程在读写一个共享数据时结果依赖于它们执行的相对时间，这种情形叫做竞争。
  - 竞争条件（Race Condition）：多个进程并发访问和操作同一数据且执行结果与访问的特定顺序有关，称为竞争（发生）条件。
- 并行性的确定：Bernstein条件
  - Bernstein条件：当下列条件同时成立时，两个进程 $S_1$ 和 $S_2$ 可并发：
    - $R(S_1) \cap W(S_2) = \emptyset$
    - $W(S_1) \cap R(S_2) = \emptyset$
    - $W(S_1) \cap W(S_2) = \emptyset$
  - 举例：

- $S_1$ :  $a := x + y$
- $S_2$ :  $b := z + 1$
- $S_3$ :  $c := a - b$
- $S_4$ :  $w := c + 1$

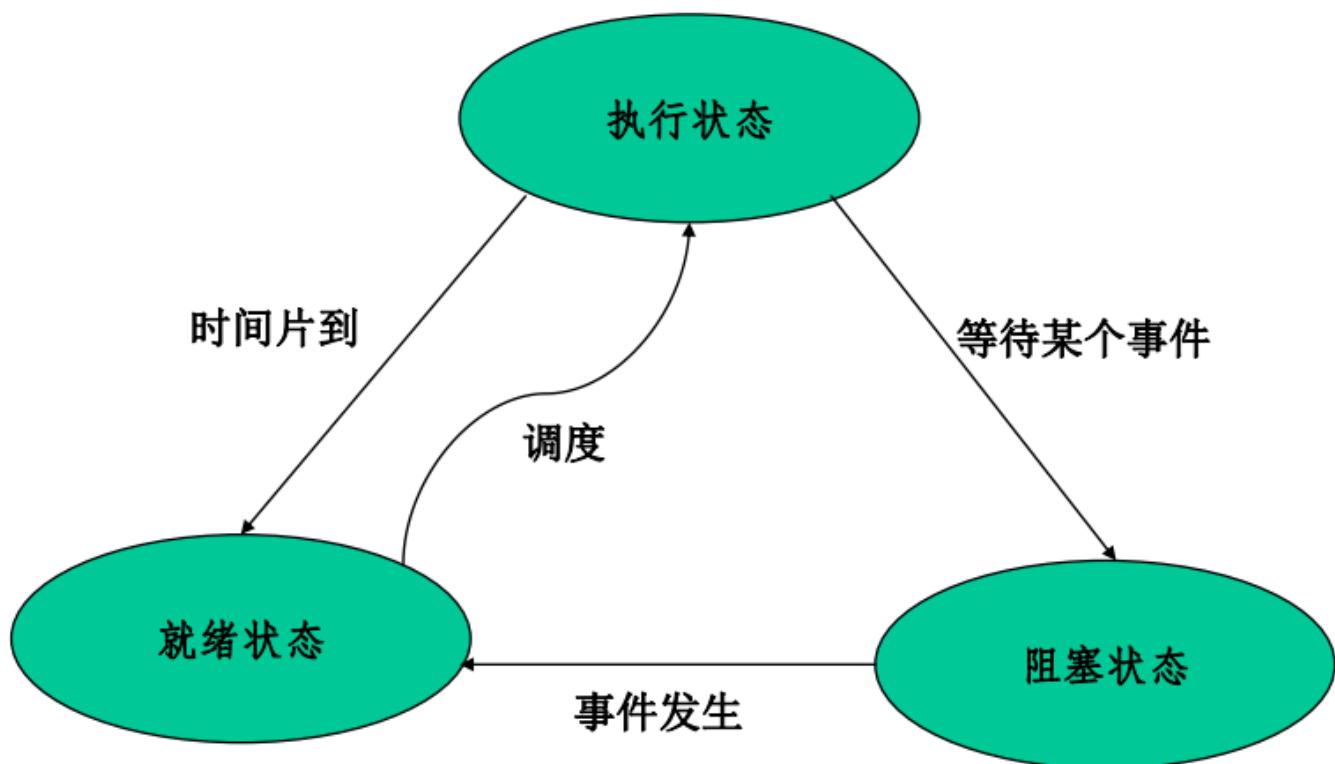


- 进程的引入：
  - 进程的定义和特征：
    - 进程是程序的一次执行；
    - 进程是可以和别的计算并发执行的计算；
    - 进程可定义为一个数据结构，及能在其上进行操作的一个程序；
    - 进程是一个程序及其数据，在处理机上顺序执行时所发生的活动；
    - 进程是程序在一个数据集合上运行的过程，它是系统进行资源分配和调度的一个独立单位。
  - 进程的特征：

- 动态性
- 并发性
- 独立性
- 异步性
- 引入进程的利弊:
  - 利：提高效率
  - 弊：空间开销、时间开销

#### 4.1.2 进程的状态及其演变

- 状态转换：



- 状态转换的条件：
  - 就绪→运行：调度程序选择一个进程运行
  - 运行→就绪：
    - 运行进程用完了时间片
    - 运行进程被中断，因为一高优先级进程处于就绪状态
  - 运行→阻塞：
    - 当一进程所需的资源必须等待时
    - OS尚未完成服务
    - 对一资源的访问尚不能进行
    - 初始化I/O 且必须等待结果
    - 等待某一进程提供输入(IPC)
  - 阻塞→就绪：当所等待的事件发生时

- 进程控制块：系统为每个进程定义了一个数据结构，进程控制块PCB（Process Control Block）

作用：

- 进程创建、撤销
- 进程的唯一标志
- 限制系统进程数

辨析：**进程上下文切换和陷入内核**（考虑是否背诵二者流程）。

进程的不足：

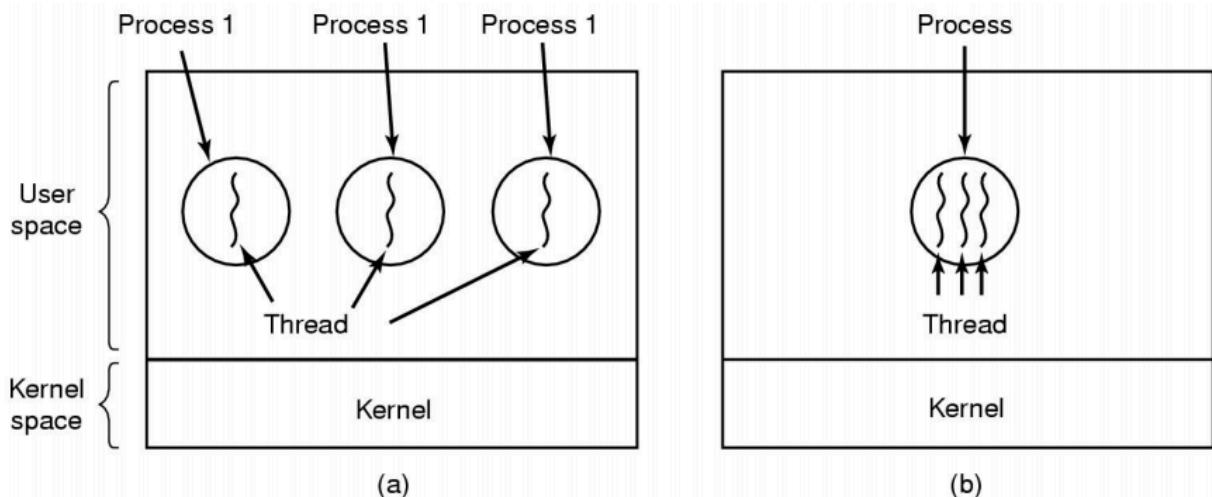
- 进程只能在一个时间干一件事，如果想同时干两件事或多件事，进程就无能为力了。
- 进程在执行的过程中如果阻塞，例如等待输入，整个进程就会挂起，即使进程中有些工作不依赖于输入的数据，也将无法执行。

#### 4.1.3 线程的概念引入

- 进程包含了两个概念：资源拥有者和可执行单元，现代操作系统将资源拥有者称为进程（process, task），可执行单元称为线程（thread）

线程：将资源与计算分离，提高并发效率。

多进程vs多线程图示如下：



- 引入线程目的：

- 减小进程切换的开销
- 提高进程内的并发程度
- 共享资源

- 好处：

- 引入进程的好处：多个程序可以并发执行，改善资源使用率，提高系统效率
- 引入线程的好处：减少并发程序执行时所付出的时空开销，使得并发粒度更细、并发性更好

- 每个线程拥有自己的栈

进程是资源分配的基本单位，线程是处理机调度的基本单位

#### 4.1.4 线程的实现方式

- 用户级线程：
  - 线程在用户空间，通过library模拟的thread，不需要或仅需要极少的kernel支。
  - 持上下文切换比较快，因为不用更改page table等，使用起来较为轻便快速。
  - 提供操控视窗系统的较好的解决方案。

用户级线程的主要功能：

- 创建和销毁线程
- 线程之间传递消息和数据
- 调度线程执行
- 保存和恢复线程上下文

典型的例子：

- POSIX Pthreads
- Mach C-threads
- Java Threads

用户级线程的优缺点：

- 优点：
  - 线程切换与内核无关
  - 线程的调度由应用决定，容易进行优化
  - 可运行在任何操作系统上，只需要线程库的支持
- 不足：
  - 很多系统调用会引起阻塞，内核会因此而阻塞所有相关的线程。
  - 内核只能将处理器分配给进程，即使有多个处理器，也无法实现一个进程中的多个线程的并行执行。

- 内核级线程：

- 内核级线程就是kernel有好几个分身，一个分身可以处理一件事。
- 这用来处理非同步事件很有用，kernel可以对每个非同步事件产生一个分身来处理。
- 支持内核线程的操作系统内核称作多线程内核

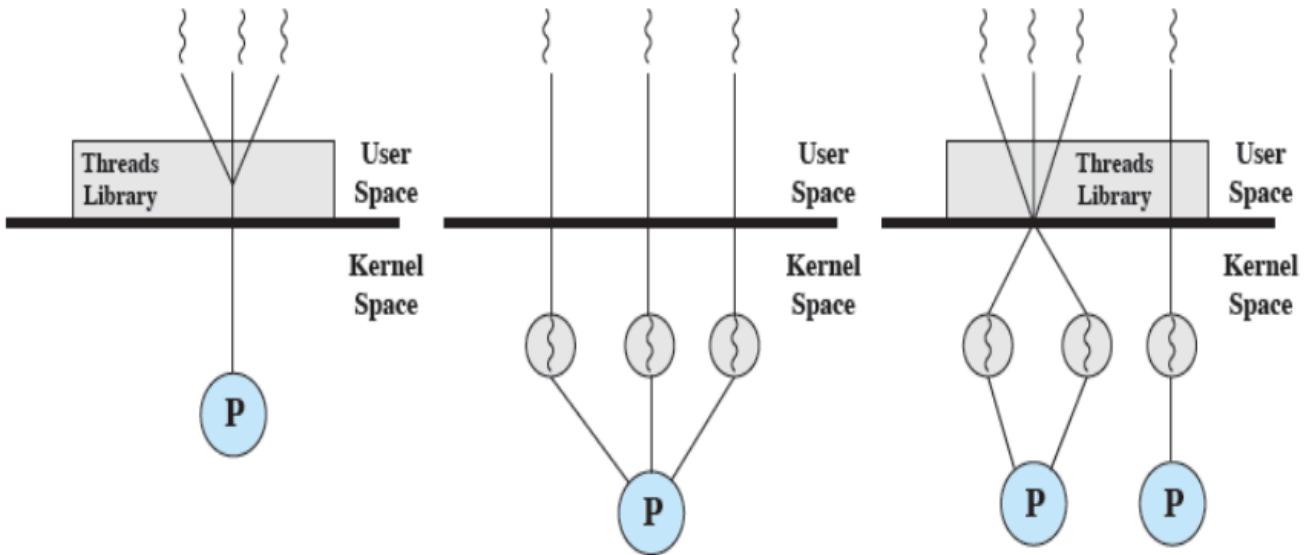
内核级线程的优缺点：

- 优点：
  - 内核可以在多个处理器上调度一个进程的多个线程实现同步并行执行。
  - 阻塞发生在线程级别
  - 内核中的一些处理可以通过多线程实现

缺点：

- 一个进程中的线切换需要内核参与线程的切换涉及两个模式的切（进程-进程、线程-线程）
- 降低效率

- 混合的实现方式



## 4.2 同步与互斥

### 4.2.1 同步与互斥问题

- 程序的并发执行：
  - 特征：
    - 并发：体现在进程的执行是间断性的；进程的相对执行速度是不可测的。（间断性）
    - 共享：体现在进程/线程之间的制约性（如共享打印机）（非封闭性）。
    - 不确定性：进程执行的结果与其执行的相对速度有关，是不确定的（不可再现性）。
  - 概念：
    - 竞争：两个或多个进程对同一共享数据同时进行访问，而最后的结果是不可预测的，它取决于各个进程对共享数据访问的相对次序。这种情形叫做竞争。
    - 竞争条件：多个进程并发访问和操作同一数据且执行结果与访问的特定顺序有关。
    - 临界资源：我们将一次仅允许一个进程访问的资源称为临界资源。
    - 临界区：每个进程中访问临界资源的那段代码称为临界区。
- 进程的同步与互斥
  - 进程互斥（间接制约关系）：
    - 两个或两个以上的进程，不能同时进入关于同一组共享变量的临界区域，否则可能发生与时间有关的错误，这种现象被称作进程互斥。
    - 进程互斥是进程间发生的一种间接性作用
  - 进程同步（直接制约关系）：
    - 系统中各进程之间能有效地共享资源和相互合作，从而使程序的执行具有可再现性的过程称为进程同步。
    - 进程同步是进程间的一种刻意安排的直接制约关系。即为完成同一个任务的各进程之间，因需要协调它们的工作而相互等待、相互交换信息所产生的制约关系。
- 同步与互斥的区别与联系：
  - 互斥：某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。互斥无法限制访问者对资源的访问顺序，即访问是无序访问。

- 同步：是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有对资源的写入的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。

#### 4.2.2 基于忙等待的互斥方法

- 软件方法

- Dekker算法：

```
P:
.....
ptrun = true; // P进程想要进入临界区
while(qturn) { // 检查Q进程是否也想要进入临界区
    if (turn == 1) { // 检查共享变量turn的值
        pturn = false; // P进程暂时放弃竞争
        while(turn == 1); // 忙等待 (Busy waiting)
        pturn = true; // P重新申明想进入临界区
    }
}
// 临界区代码
turn = 1; // 让权给另一个进程Q (规定)
ptrun = fasle; // 当前进程P完全退出临界区，释放竞争标志
.....
```

```
Q:
.....
qturn = true; // Q申请进入临界区
while(pturn) { // 检查P进程是否也想进入临界区
    if (turn == 0) { // 检查共享变量turn的值
        qturn = false; // Q进程暂时放弃竞争
        while(turn == 0); // 忙等待
        qturn = true; // Q重新申明想进入临界区
    }
}
// 临界区代码
turn = 0; // 让权给进程P (规定)
qturn = false; // 进程Q完全退出临界区，释放竞争标志
```

- Peterson算法：

```

#define FALSE 0
#define TRUE 1
#define N 2 // 进程的个数
int turn; // 轮到谁?
int interested[N]; // 兴趣数组, 初始值均为FALSE
void enter_region (int process)
{
    int other;
    other = 1 - process; // 另外一个进程的进程号
    interested[process] = TRUE; // 表明本进程感兴趣
    turn = process; // 设置标志位
    while(turn == process && interested[other] == TRUE); // 循环
}

```

```

void leave_region(int process)
{
    interested[process] = FALSE; // 本进程已离开临界区
}

```

```

// 进程
.....
enter_region(i);
// 临界区
leave_region(i);
.....

```

◦ 面包店算法：略

- 硬件方法：

1. 中断屏蔽方法：使用“开关终断”指令

- 执行“关中断”指令，进入临界区操作
- 退出临界区之前，执行“开中断”指令

优缺点：

- 简单
- 不适用于多CPU系统：往往带来很大的性能损失
- 单处理器使用：很多日常任务，都是靠中断的机制来触发的，比如时钟，如果使用屏蔽中断，会影响时钟和系统效率，而且用户进程的使用可能很危险！
- 使用范围：内核进程（少量使用）

2. 使用test and set指令

- TS (test-and-set) 是一种不可中断的基本原语（指令）

```

acquire(lock) {
    while(test_and_set(lock) == 1);
}
// critical section
release(lock) {
    lock = 0;
}

```

### 3. 使用swap指令

- 几个算法的共性问题：无论是软件解法（如Peterson）还是硬件（如TSL或XCHG）解法都是正确的，它们都有一个共同特点：当一个进程想进入临界区时，先检查是否允许进入，若不允许，则该进程将原地等待，直到允许为止。
  - 忙等待：浪费CPU时间
  - 优先级反转：低优先级进程先进入临界区，高优先级进程一直忙等

#### 4.2.3 基于信号量的方法

- 经典的信号量机制：

```

P(S): while S <= 0 do skip
      S := S - 1;
V(S): S := S + 1;

```

- 信号量的使用：
  - 必须置一次且只能置一次初值
  - 只能由P、V操作来改变
- 物理意义：
  - S.value为正时表示资源的个数
  - S.value为负时表示等待进程的个数 (wait)
  - P操作分配资源，如果无法分配则阻塞
  - V操作释放资源，如果有等待进程则唤醒 (signal)
- 信号量机制的实现：关中断、TS指令等
- 信号量的应用：
  - 互斥：利用信号量实现进程互斥 (S=1)
  - 同步：
    - 利用信号量实现进程同步(S=0)
    - 例如：描述进程执行的前驱/后继关系

互斥:

$P(s)$

临界区

$V(s)$

同步

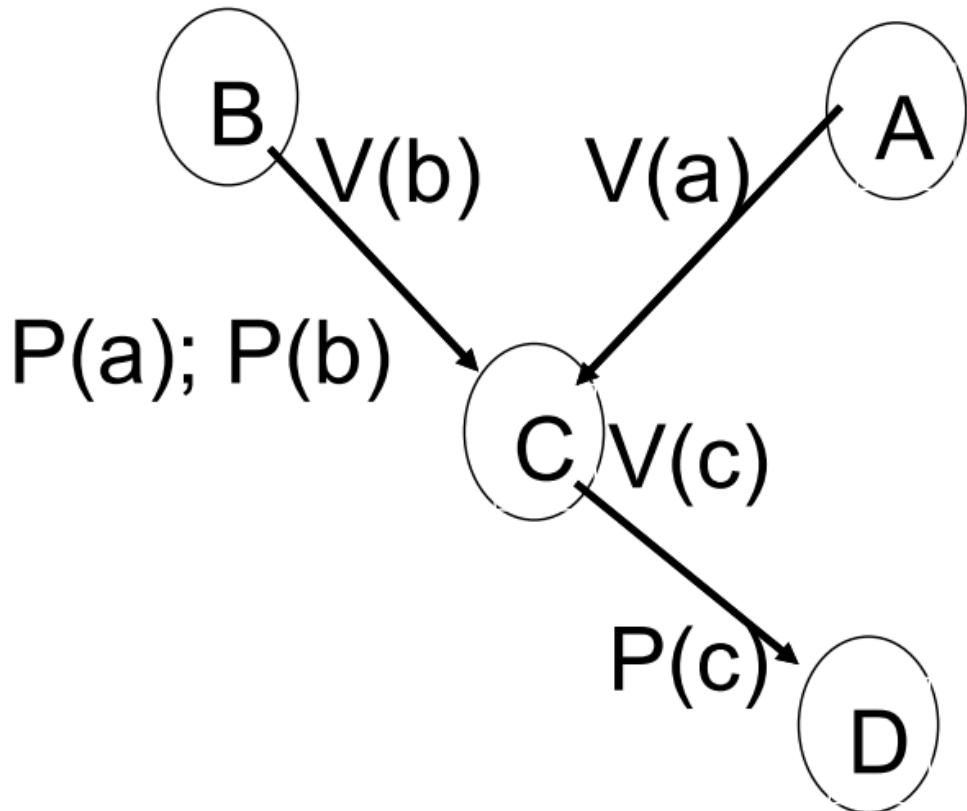
$P(s)$

后执行临界区

先执行临界区

$V(s)$

- 例如：有如下图前驱关系



- 信号量在并发中的典型应用：

应用	描述
互斥 (Mutual exclusion)	可以用初始值为1的信号量来实现进程间的互斥。一个进程在进入临界区之前执行semWait操作，退出临界区后再执行一个semSignal操作。这是实现临界区资源互斥使用的一个二元信号量。
有限并发 (Bounded concurrency)	是指有n ( $1 \leq n \leq c$ , c是一个常量) 个进程并发的执行一个函数或者一个资源。一个初始值为c的信号量可以实现这种并发。
进程同步 (Synchronization)	是指当一个进程P <sub>i</sub> 想要执行一个a <sub>j</sub> 操作时，它只在进程P <sub>j</sub> 执行完a <sub>j</sub> 后，才会执行a <sub>i</sub> 操作。可以用信号量如下实现：将信号量初始为0，P <sub>i</sub> 执行a <sub>i</sub> 操作前执行一个semWait操作；而P <sub>j</sub> 执行a <sub>j</sub> 操作后，执行一个semWait操作。

- P, V操作的优缺点
  - 优点：简单，而且表达能力强（用P.V操作可解决任何同步互斥问题）
  - 缺点：
  - 不够安全；P.V操作使用不当会出现死锁；遇到复杂同步互斥问题时实现复杂

#### 4.2.4 基于管程的同步与互斥

- 管程 (Monitor)：把分散的临界区集中起来，为每个可共享资源设计一个专门机构来统一管理各进程对该资源的访问，这个专门机构称为管程。

管程是一种高级同步原语。

一个管程是由过程、变量及数据结构等组成的一个集合，它们组成一个特殊的模块或者软件包。

互斥：任一时刻，管程中只能有一个活跃进程。

管程是一种语言概念，由编译器负责实现互斥。

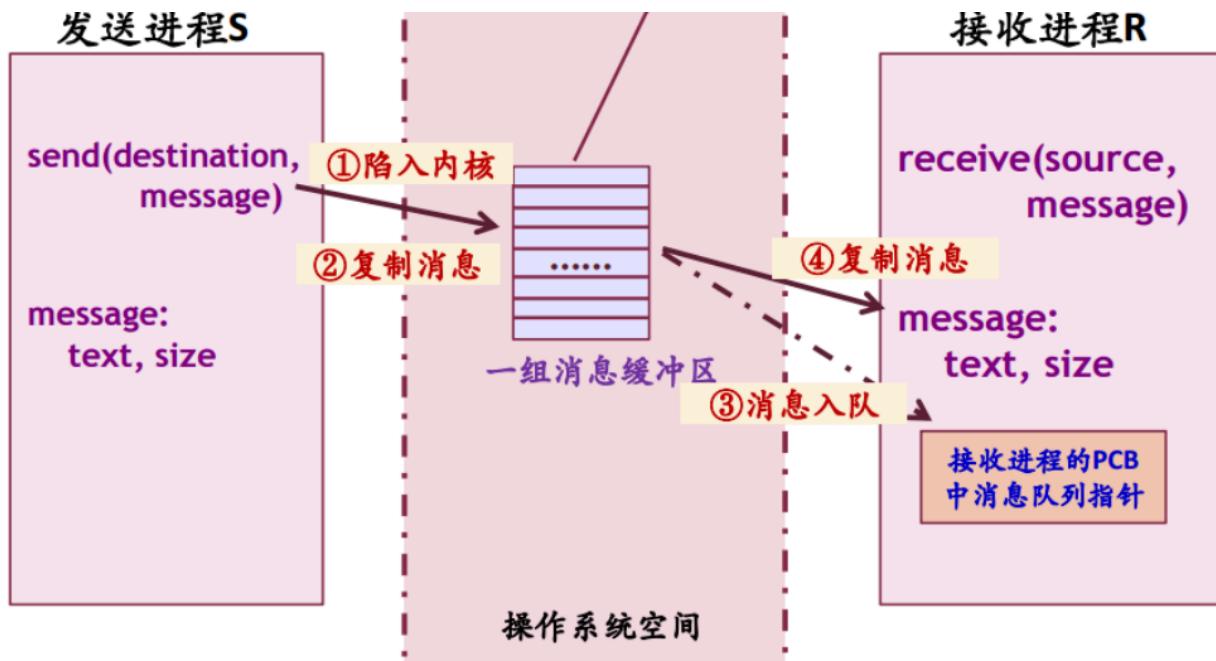
- 管程的实现：
  - Hoare管程与Hansen管程

#### 4.2.5 进程通信的主要方法

- 进程间通信 (IPC)：
  - 低级通信：只能传递状态和整数值（控制信息），包括进程互斥和同步所采用的信号量和管程机制。
    - 传送信息量小：效率低，每次通信传递的信息量固定，若传递较多信息则需要进行多次通信。
    - 编程复杂：用户直接实现通信的细节，编程复杂，容易出错。
  - 高级通信：适用于分布式系统，基于共享内存的多处理机系统，单处理机系统，能够传送任意数量的数据，可以解决进程的同步问题和通信问题，主要包括三类：管道、共享内存、消息系统。
- IPC概述：

- 管道 (Pipe) 及命名管道 (Named pipe或FIFO)
  - 消息队列 (Message)
  - 共享内存 (Shared memory)
  - 信号量 (Semaphore)
  - 套接字 (Socket)
  - 信号 (Signal)
- 名词解释：
    - 无名管道 (Pipe)：
      - 管道是半双工的，**数据只能向一个方向流动**；需要双方通信时，需要建立起两个管道；
      - **只能用于父子进程或者兄弟进程之间**（具有亲缘关系的进程）；
      - **单独构成一种独立的文件系统**：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在在内存中。
      - 数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。
    - 有名管道 (Named Pipe或FIFO)
      - 无名管道应用的一个重大限制是它没有名字，因此，只能用于具有亲缘关系的进程间通信，在有名管道提出后，该限制得到了克服。
      - FIFO不同于管道之处在于它提供一个路径名与之关联，以FIFO的文件形式存在于文件系统中。这样，即使与FIFO的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过FIFO相互通信（能够访问该路径的进程以及FIFO的创建进程之间），因此，通过FIFO不相关的进程也能交换数据。
      - FIFO严格遵循先进先出 (first in first out)，对管道及FIFO的读总是从开始处返回数据，对它们的写则把数据添加到末尾。
    - 消息传递 (message passing)：
      - 消息传递——两个通信原语 (OS系统调用)
        1. send (destination, &message)
        2. receive(source, &message)
      - 调用方式：
        1. 阻塞调用
        2. 非阻塞调用
      - 主要问题：
        1. 解决消息丢失、延迟问题 (TCP协议)
        2. 编址问题：mailbox

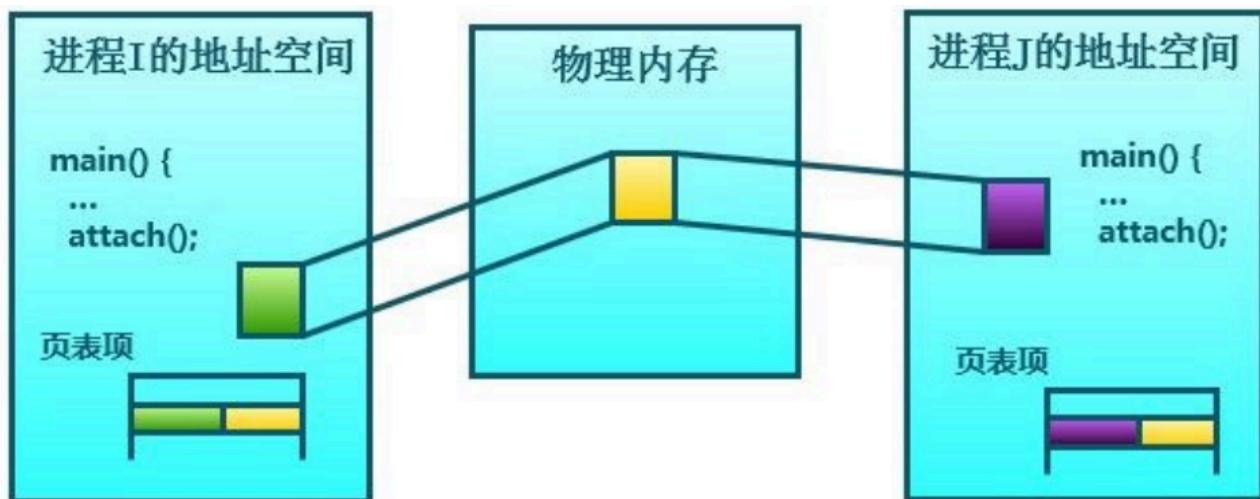
消息传递的过程：



- 共享内存:

- 共享内存是最有用的进程间通信方式，也是最快的IPC形式（因为它避免了其它形式的IPC必须执行的开销巨大的缓冲复制）。
- 两个不同进程A、B共享内存的意义是，同一块物理内存被映射到进程A、B各自的进程地址空间。
- 当多个进程共享同一块内存区域，由于共享内存可以同时读但不能同时写，则需要同步机制约束（互斥锁和信号量都可以）。
- 共享内存通信的效率高（因为进程可以直接读写内存）。
- 进程之间在共享内存时，保持共享区域直到通信完毕。

共享内存机制:



## 4.3 经典的进程同步与互斥问题

### 4.3.1 生产者消费者问题

- 问题描述：若干进程通过有限的共享缓冲区交换数据。其中，“生产者”进程不断写入，而“消费者”进程不断读出；共享缓冲区共有N个；任何时刻只能有一个进程可对共享缓冲区进行操作。
- 问题分析：
  - 两个隐含条件：

1. 消费者和生产者数量不固定。
  2. 消费者和生产者不能同时使用缓存区。
- 行为分析：
    - 生产者：生产产品，放置产品(有空缓冲区)。
    - 消费者：取出产品(有产品)，消费产品。
  - 行为分析：
    - 生产者之间：互斥(放置产品)
    - 消费者之间：互斥(取出产品)
    - 生产者与消费者之间：互斥(放/取产品) 同步(放置——取出)

- 问题解决：

- 用PV操作解决生产者/消费者问题

- 信号量设置：

```
semaphore mutex = 1; // 互斥
semaphore empty = N; // 空闲数量
semaphore full = 0; // 产品数量
```

- 实际上，full和empty是同一个含义：

full + empty == N

- 代码：

生产者：  
 P(empty);  
 P(mutex);  
 代码  
 V(mutex);  
 V(full);

消费者：  
 P(full);  
 P(mutex);  
 代码  
 V(mutex);  
 V(empty);

完整的伪代码：

```
Semaphore full = 0;
Semaphore empty = n;
Semaphore mutex = 1;
ItemType buffer[0...n-1];
int in = 0, out = 0;

main() {
    cobegin
        proceducer();
        consumer();
}
```

```

Coend

}

procedure() {
    while(true) {
        生产产品nextp
        P(empty);
        P(mutex);
        buffer[in] = nextp;
        in = (in + 1) MOD n;
        V(mutex);
        V(full);
    }
}

consumer() {
    while(true) {
        P(full);
        P(mutex);
        nextc = buffer[out];
        out = (out + 1) MOD n;
        V(mutex);
        V(empty);
        消费nextc中的产品
    }
}

```

#### 4.3.2 读者写者问题

- 问题描述：对共享资源的读写操作，任一时刻“写者”最多只允许一个，而“读者”则允许多个，即“读一写”互斥，“写一写”互斥，“读一读”允许；
- 生活中的实例：12306定票
- 问题解决：

- 用PV操作解决读者写者问题

- 信号量设置：

```

Semaphore mutex = 1; // 代表对readcount的互斥操作
Semaphore wmutex = 1; // 表示“允许写”
readcount = 0 // 公共变量，表示“正在读”的进程

```

- 代码：

```

writer
P(mutex);
write
V(mutex);

Reader
P(mutex);
if readcount = 0 then P(wmutex);
readcount = readcount + 1;

```

```
V(mutex)
read
P(mutex)
readcount = readcount - 1;
if readcount = 0 then V(mutex);
V(mutex)
```

该算法对读者有利，可能造成写者饥饿。

- 读写公平的算法：

```
Writer
P(rwmutex);
P(wmutex);
写数据
V(wmutex);
P(rwmutex);

Reader
P(rwmutex);
P(mutex);
if readcount = 0 then P(wmutex);
readcount = readcount + 1;
V(mutex);
V(rwmutex);
read
P(mutex);
readcount = readcount + 1;
if readcount = 0 then V(wmutex);
V(mutex);
```

## 4.4 调度

### 4.4.1 基本概念

# 什么是CPU调度？

CPU 调度的任务是控制、协调 多个进程对 CPU 的竞争。也就是按照一定的策略（调度算法），从就绪队列中 选择一个进程，并把 CPU 的控制权交给被选中的进程。

## CPU调度的场景

- N 个进程就绪，等待上 CPU 运行
- M 个CPU, M≥1
- OS 需要决策，给哪个进程分配哪个 CPU 。

### • 调度的分类：

- 高级调度：又称为“宏观调度”、“作业调度”。从用户工作流程的角度，一次提交的若干个作业，对每个作业进行调度。时间上通常是分钟、小时或天。
- 中级调度：内外存交换：又称为“中级调度”。指令和数据必须在内存里才能被CPU直接访问。从存储器资源的角度，将进程的部分或全部换出到外存上，将当前所需部分换入到内存。
- 低级调度：又称为“微观调度”、“进程或线程调度”。从CPU资源的角度，执行的单位，时间上通常是毫秒。因为执行频繁，要求在实现时达到高效率。

## 何时进行调度

- 当一个新的进程被创建时，是执行新进程还是继续执行父进程？
- 当一个进程运行完毕时；
- 当一个进程由于I/O、信号量或其他的某个原因被阻塞时；
- 当一个I/O中断发生时，表明某个I/O操作已经完成，而等待该I/O操作的进程转入就绪状态；
- 在分时系统中，当一个时钟中断发生时。

只要OS取得对CPU的控制，进程切换就可能发生：

- **用户调用**: 来自程序的显式请求(如：读写文件)，该进程多半会被阻塞
- **陷阱**: 最末一条指令导致出错，会引起进程移至退出状态
- **中断**: 外部因素影响当前指令的执行，控制被转移至中断处理程序

- **进程（上下文）切换的步骤：**

1. 保存处理器的上下文，包括程序计数器和其它寄存器
2. 用新状态和其它相关信息更新正在运行进程的PCB
3. 把进程移至合适的队列：就绪、阻塞
4. 选择另一个要执行的进程
5. 更新被选中进程的PCB（新状态等）
6. 从被选中进程中重装入CPU 上下文

- 面向用户的调度性能准则：

- **周转时间**: 作业从提交到完成（得到结果）所经历的时间。包括：在收容队列中等待，CPU上执行，就绪队列和阻塞队列中等待，结果输出等待——**批处理系统**（外存等待时间、就绪等待时间、**CPU执行时间**、I/O操作时间）
  - 平均周转时间 ( $T$ ) - 平均带权周转时间 ( $T/T_s$ )
- **响应时间**: 用户输入一个请求（如击键）到系统给出首次响应（如屏幕显示）的时间——**分时系统**

- **截止时间**: 开始截止(最早开始)时间和完成截止(最晚完成)时间——**实时系统**, 与周转时间有些相似。
- **优先级**: 可以使关键任务达到更好的指标。
- **公平性**: 不因作业或进程本身的特性而使上述指标过分恶化, 如长作业等待很长时间。
- 面向系统的调度性能准则:
- **吞吐量**: 单位时间内所完成的作业数, 跟作业本身特性和调度算法都有关系——**批处理系统**
  - 平均周转时间不是吞吐量的倒数, 因为并发执行的作业在时间上可以重叠。如: 在2小时内完成4个作业, 则吞吐量是2个作业/小时, 而平均周转时间可能是0.5小时、1小时、1.25小时、2小时、...
- **处理机利用率**: 忙碌时间/总时间——**大中型主机**
- **各种资源的均衡利用**: 如CPU密集型的作业和I/O密集型(指次数多, 每次时间短)的作业搭配——**大中型主机**

#### 4.4.2 设计调度算法要考虑的问题 (设计要点)

- 进程优先级数:

优先级和优先数是不同的, 优先级表现了进程的重要性和紧迫性, 优先数实际上是一个数值, 反映了某个优先级。

静态优先级

- 进程创建时指定, 运行过程中不再改变动态优先级
- 进程创建时指定了一个优先级, 运行过程中可以动态变化。如: 等待时间较长的进程可提升其优先级。

- 占用CPU的方式

不可抢占式方式

- 一旦处理器分配给一个进程, 它就一直占用处理器, 直到该进程自己因调用原语操作或等待I/O等原因而进入阻塞状态, 或时间片用完时才出处理器, 重新进行。

抢占式方式

- 就绪队列中一旦有优先级高于当前运行进程优先级的进程存在时，便立即进行进程调度，把处理器转给优先级高的进程。
- 进程的分类：

#### 批处理进程 (Batch Process)

- 无需与用户交互，通常在后台运行
- 不需很快的响应
- 典型的批处理程序：编译器、科学计算

#### 交互式进程 (Interactive Process)

- 与用户交互频繁，因此要花很多时间等待用户输入
- 响应时间要快，平均延迟要低于50~150ms
- 典型的交互式进程：Word、触控型GUI

#### 实时进程 (Real-time Process)

- 有实时要求，不能被低优先级进程阻塞
- 响应时间要短且要稳定
- 典型的实时进程：视频/音频、控制类

### 批处理系统的调度算法

一些基本概念：

- 吞吐量、平均等待时间和平均周转时间：

- 吞吐量 =  $\frac{\text{作业数}}{\text{总执行时间}}$ ，即单位时间CPU完成的作业数量
- 周转时间 (Turnover Time) = 完成时刻 - 提交时刻
- 带权周转时间=周转时间/服务时间（执行时间）
- 平均周转时间 =  $\frac{\text{作业周转时间之和}}{\text{作业数}}$
- 平均带权周转时间 =  $\frac{\text{作业带权周转时间之和}}{\text{作业数}}$

批处理系统中常用的调度算法

- 先来先服务 (FCFS: First Come First Serve)
- 最短作业优先 (SJF: Shortest Job First)
- 最短剩余时间优先 (SRTF: Shortest Remaining Time First)
- 最高响应比优先 (HRRF: Highest Response Ratio First)

#### 1. 先来先服务: (非抢占式)

这是最简单的调度算法，按先后顺序调度。

- 按照作业提交或进程变为就绪状态的先后次序，分派CPU；
- 当前作业或进程占用CPU，直到执行完或阻塞，才出让CPU (非抢占方式)。
- 在作业或进程唤醒后 (如I/O完成)，并不立即恢复执行，通常等到当前作业或进程出让CPU。
- 最简单的算法。

FCFS的特点

- 比较有利于长作业，而不利于短作业。
- 有利于CPU繁忙的作业，不利于I/O繁忙的作业。

#### 2. 短作业优先(SJF, Shortest Job First) (非抢占式)

又称为“短进程优先”SPN(Shortest Process Next)；这是对FCFS算法的改进，其目标是减少平均周转时间。

- 对预计执行时间短的作业 (进程) 优先分派处理机。通常后来的短作业不抢先正在执行的作业。

优点：

- 比FCFS改善平均周转时间和平均带权周转时间，缩短作业的等待时间；
- 提高系统的吞吐量；

缺点：

- 对长作业非常不利，可能长时间得不到执行；
- 未能依据作业的紧迫程度来划分执行的优先级；
- 难以准确估计作业 (进程) 的执行时间，从而影响调度性能。

#### 3. 最短剩余时间优先SRTF (抢占式)

将短作业优先进行改进，改进为抢占式，这就是最短剩余时间优先算法(SRTF)了，即一个新就绪的进程如果比当前运行进程具有更短的完成时间，则系统抢占当前进程，选择新就绪的进程执行。

缺点：源源不断的短任务到来，可能使长的任务长时间得不到运行，导致产生“饥饿”现象。

#### 4. 最高响应比优先HRRF (抢占式)

HRRF算法实际上是FCFS算法和SJF算法的折衷既考虑作业等待时间，又考虑作业的运行时间，既照顾短作业又不使长作业的等待时间过长，改善了调度性能。

在每次选择作业投入运行时，先计算后备作业队列中每个作业的响应比RP(响应优先级)，然后选择其值最大的作业投入运行。

$$\begin{aligned} \text{RR定义为: } RR &= \frac{\text{已等待时间} + \text{要求运行时间}}{\text{要求运行时间}} \\ &= 1 + \frac{\text{已等待时间}}{\text{要求运行时间}}。 \end{aligned}$$

响应比的计算时机：

- 每当调度一个作业运行时，都要计算后备作业队列中每个作业的响应比，选择响应比最高者投入运行。(非抢占)
- 最高响应比优先 (HRRF) 算法效果：
  - 短作业容易得到较高的响应比
  - 长作业等待时间足够长后，也将获得足够高的响应比
  - 饥饿现象不会发生
- 缺点：
  - 每次计算各道作业的响应比会有一定的时间开销，性能比SJF略差。

#### 4.4.3 交互式系统的调度算法

1. 时间片轮转(RR: Round Robin)：算法主要用于微观调度，设计目标是提高资源利用率。其基本思路是通过时间片轮转，提高进程并发性和响应时间特性，从而提高资源利用率；

步骤：

- 【排队】将系统中所有的就绪进程按照FCFS原则，排成一个队列。
- 【轮转】每次调度时将CPU分派给队首进程，让其执行一个时间片。时间片的长度从几个ms到几百ms。
- 【中断】在一个时间片结束时，发生时钟中断。
- 【抢占】调度程序据此暂停当前进程的执行，将其送到就绪队列的末尾，并通过上下文切换执行当前的队首进程。
- 【出让】进程可以未使用完一个时间片，就出让CPU (如阻塞)。

时间片长度变化的影响：

- 过长→退化为FCFS算法，进程在一个时间片内都执行完，响应时间长。
- 过短→用户的一次请求需要多个时间片才能处理完，上下文切换次数增加，响应时间长。

系统的响应时间： $T(\text{响应时间}) = N(\text{进程数目}) * q(\text{时间片})$

就绪进程的数目：数目越多，时间片越小

系统的处理能力：应当使用户输入通常在一个时间片内能处理完，否则会使响应时间、平均周转时间和平均带权周转时间延长。

#### 2. 优先级算法

本算法是平衡各进程对响应时间的要求。适用于作业调度和进程调度，可分成抢先式和非抢先式；

静态优先级：创建进程时就确定，直到进程终止前都不改变。通常是一个整数。依据：

- 进程类型 (系统进程优先级较高)
- 对资源的需求 (对CPU和内存需求较少的进程，优先级较高)

- 用户要求（紧迫程度和付费多少）

动态优先级：在创建进程时赋予的优先级，在进程运行过程中可以自动改变，以便获得更好的调度性能。如：

- 在就绪队列中，等待时间延长则优先级提高，从而使优先级较低的进程在等待足够的时间后，其优先级提高到可被调度执行；
- 进程每执行一个时间片，就降低其优先级，从而一个进程持续执行时，其优先级降低到出让CPU。

### 3. 多级队列算法(Multiple-level Queue):

本算法引入多个就绪队列，通过各队列的区别对待，达到一个综合的调度目标；

- 根据作业或进程的性质或类型的不同，将就绪队列再分为若干个子队列。
- 每个作业固定归入一个队列。

不同队列可有不同的优先级、时间片长度、调度策略等；在运行过程中还可改变进程所在队列。

如：系统进程、用户交互进程、批处理进程等。

### 4. 多级反馈队列算法(Round Robin with Multiple Feedback)

多级反馈队列算法：时间片轮转算法和优先级算法的综合和发展。优点：

- 为提高系统吞吐量和缩短平均周转时间而照顾短进程
- 为获得较好的I/O设备利用率和缩短响应时间而照顾I/O型进程
- 不必估计进程的执行时间，动态调节

设置多个就绪队列，分别赋予不同的优先级，如逐级降低，队列1的优先级最高。每个队列执行时间片的长度也不同，规定优先级越低则时间片越长，如逐级加倍

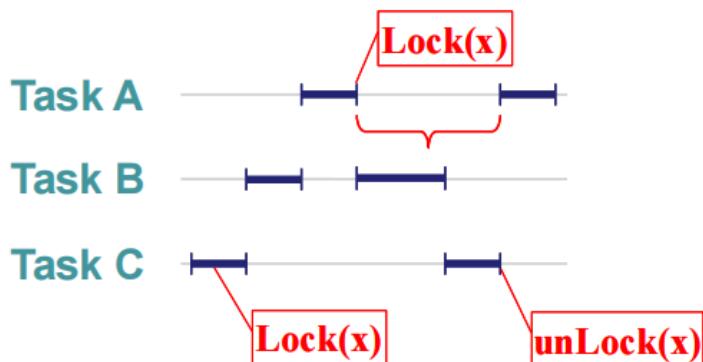
新进程进入内存后，先投入队列1的末尾，按“时间片轮转”算法调度；若按队列1一个时间片未能执行完，则降低投入到队列2的末尾，同样按“时间片轮转”算法调度；如此下去，降低到最后的队列，则按“FCFS”算法调度直到完成。

仅当较高优先级的队列为空，才调度较低优先级的队列中的进程执行。如果进程执行时有新进程进入较高优先级的队列，则抢先执行新进程，并把被抢先的进程投入原队列的末尾。

优先级倒置：

**优先级倒置现象**: 高优先级进程（或线程）被低优先级进程（或线程）延迟或阻塞。

- 例如: 有三个完全独立的进程 Task A、Task B 和 Task C, Task A 的优先级最高, Task B 次之, Task C 最低。Task A 和 Task C 共享同一个临界资源 X。

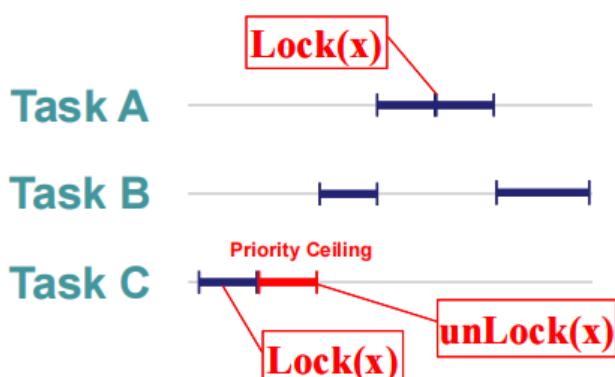


Task A 和 Task C 共享同一个临界资源, 高优先级进程 Task A 因低优先进程 Task C 被阻塞, 又因为低优先进程 Task B 的存在延长了被阻塞的时间。

解决方法:

## 优先级置顶 (Priority Ceiling)

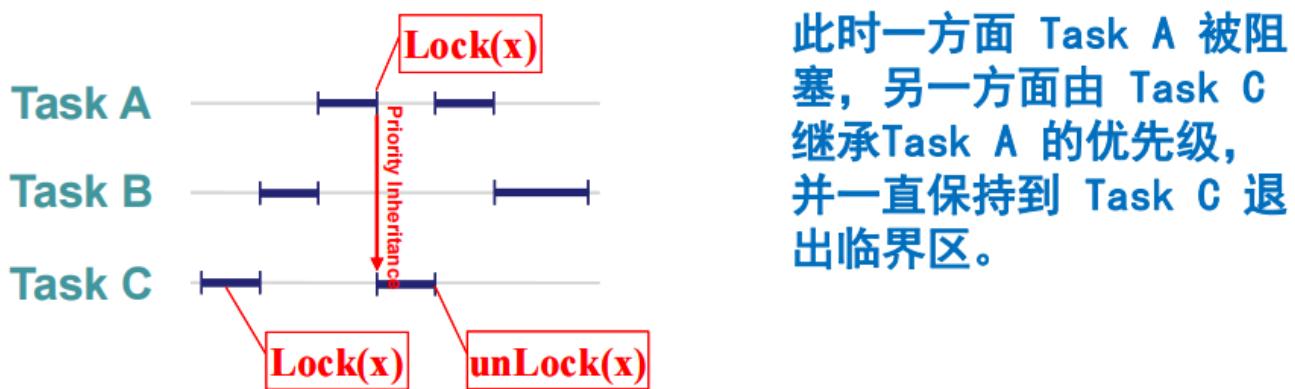
- 进程 Task C 在进入临界区后, Task C 所占用的处理机就不允许被抢占。这种情况下, Task C 具有最高优先级 (Priority Ceiling)。



如果系统中的临界区都较短且不多, 该方法是可行的。反之, 如果 Task C 临界区非常长, 则高优先级进程 Task A 仍会等待很长的时间, 其效果无法令人满意。

## 优先级继承（Priority Inheritance）

- 当高优先级进程 Task A 要进入临界区使用临界资源 X 时，如果已经有一个低优先级进程 Task C 正在使用该资源，可以采用优先级继（Priority Inheritance）的方法。



### 4.4.4 实时系统的调度算法

- 实时系统是一种时间起着主导作用的系统。当外部的一种或多种物理设备给了计算机一个刺激，而计算机必须在一个确定的时间范围内恰当地做出反应。对于这种系统来说，正确的但是迟到的应答往往比没有答案还要糟糕。
- 实时系统被分为硬实时系统和软实时系统。硬实时要求绝对满足截止时间要求（如：汽车和飞机的控制系统），而软实时可以偶尔不满足（如：视频/音频程序）。
- 实时系统通常将对不同刺激的响应指派给不同的进程（任务），并且每个进程的行为是可提前预测的。

- 要求更详细的调度信息：如，就绪时间、开始或完成截止时间、处理时间、资源要求、绝对或相对优先级（硬实时或软实时）。
- 采用抢先式调度。
- 快速中断响应，在中断处理时（硬件）关中断的时间尽量短。
- 快速任务分派。相应地采用较小的调度单位（如线程）。
- 静态表调度**Static table-driven scheduling**
- 单调速率调度**RMS: Rate Monotonic Scheduling**
  - 任务集可调度，**if**,  $\sum_{i=1}^n \frac{C_i}{T_i} \leq n(\sqrt[n]{2} - 1)$   

$$\lim_{n \rightarrow \infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 0.693147\dots$$
- 最早截止时间优先算法**EDF: Earliest Deadline First**
  - 任务集可调度，**iff**,  $\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$

#### 1. 静态表调度算法：

通过对所有周期性任务的分析预测（到达时间、运行时间、结束时间、任务间的优先关系），事先确定一个固定的调度方案。

特点：

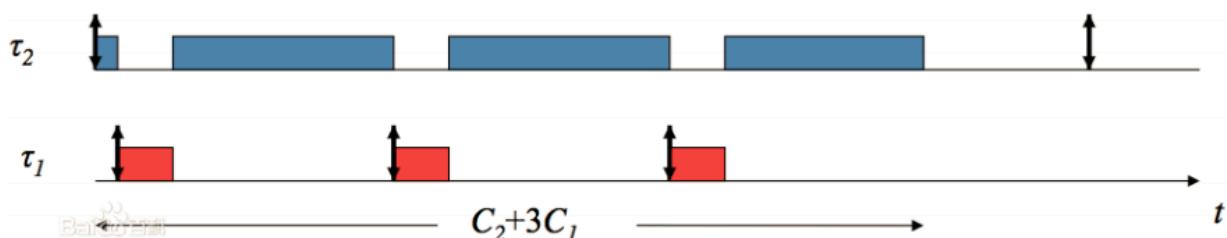
- 无任何计算，按固定方案进行，开销最小；
- 无灵活性，只适用于完全固定的任务场景。

#### 2. 单调速率调度**RMS**

## 算法

- **优先级静态固定分配：**优先级与周期成反比，周期越短优先级越高。优先级高的任务先被调度。如果两个任务的优先级一样，当调度它们时，RM算法将随机选择一个调度
- 任务在周期起点释放，**高优先级任务可抢占低优先级任务的执行。**

## 静态、抢先式调度



### 3. 最早截止期优先EDF

任务的绝对截止时间越早，其优先级越高，优先级最高的任务最先被调度（动态优先级）

如果两个任务的优先级一样，当调度它们时，EDF算法将随机选择一个调度

### 4. 最低松弛度优先算法LLF

LLF算法是根据任务紧急（或松弛）的程度，来确定任务的优先级。任务的紧急度越高，其优先级越高，并使之优先执行。

## 松弛度 (Laxity)

= 进程截至时间 - 本身剩余运行时间 - 当前时间  
进程最晚开始时间（否则就要miss deadline）

调度时机：有进程执行完或有进程的Laxity为0时（抢占）。

任务集可调度， iff,

—

### 4.4.5 多处理机调度

## 4.5 死锁

### 4.5.1 死锁的概念

- 一组进程中，每个进程都无限等待被该组进程中其它进程所占有的资源，在无外力介入的条件下，将因永远分配不到资源而无法运行的现象。
- 竞争资源引起死锁

- **可剥夺资源：**是指某进程在获得这类资源后，该资源可以再被其他进程或系统剥夺。如**CPU，内存**；
- **非可剥夺资源：**当系统把这类资源分配给某进程后，再不能强行收回，只能在进程用完后自行释放。如**磁带机、打印机**；
- **临时性资源：**这是指由一个进程产生，被另一个进程使用，短时间后便无用的资源，故也称为消耗性资源。如**消息、中断**；

- 必背：

四个点  
背下来

## 死锁发生的四个**必要条件**

1. **互斥条件：**指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。
2. **请求且占有条件：**指进程已经占有至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。
3. **不可剥夺条件：**指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。
4. **环路等待条件：**指在发生死锁时，必然存在一个进程—资源的环形链，即进程集合{P0, P1, P2, …, Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

- 活锁和饥饿：

- **活锁 (livelock)**：是指任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。
  - 活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，即所谓的“活”，而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。避免活锁的简单方法是采用先来先服务的策略。
- **饥饿 (starvation)**：某些进程可能由于资源分配策略的不公平导致长时间等待。当等待时间给进程推进和响应带来明显影响时，称发生了进程饥饿，当饥饿到一定程度的进程所赋予的任务即使完成也不再具有实际意义时称该进程被饿死(starve to death)。

#### 4.5.2 处理死锁的基本方法

- 不允许死锁发生

- 预防死锁（静态）：防患于未然，破坏死锁的产生条件
- 避免死锁（动态）：在资源分配之前进行判断

条件要求

- 允许死锁发生

- 检测与解除死锁
- 无所作为：鸵鸟算法

#### 1. 死锁的预防

1. 打破互斥条件：即允许进程同时访问某些资源。但是，有的资源是不允许被同时访问的，像打印机等等，这是资源本身的属性。
2. 打破申请且占有条件：可以实行资源预先分配策略。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程，否则不分配任何资源。由于运行的进程已占有了它所需的全部资源，所以不会发生占有资源又申请资源的现象，因此不会发生死锁。

但是，这种策略也有如下缺点：

- a) 在许多情况下，由于进程在执行时是动态的，**不可预测的**，因此不可能知道它所需要的全部资源。
- b) **资源利用率低**。无论资源何时用到，一个进程只有在占有所需的全部资源后才能执行。即使有些资源最后才被用到一次，但该进程在生存期间却一直占有。这显然是一种极大的资源浪费；
- c) **降低进程的并发性**。因为资源有限，又加上存在浪费，能分配到所需全部资源的进程个数就必然少了。

3. 打破不可剥夺条件：即允许进程强行从占有者那里夺取某些资源。就是说，当一个进程已占有了某些资源，它又申请新的资源，当不能立即被满足时，须释放所占有的全部资源，以后再重新申请。它所释放的资源可以分配给其它进程。这就相当于该进程占有的资源被隐蔽地抢占了。这种预防死锁的方法实现起来困难，会降低系统性能。

4. 打破循环等待条件：实行资源有序分配策略。即把资源事先分类编号，按号分配，使进程在申请，占用资源时不会形成环路。所有进程对资源的请求必须严格按资源序号递增的顺序提出。进程占用了小号资源，才能申请大号资源，就不会产生环路，从而预防了死锁。这种策略与前面的策略相比，资源的利用率和系统吞吐量都有很大提高，但存在以下缺点：

- a) 限制了进程对资源的请求，同时给系统中所有资源合理编号也是件困难事，并增加了系统开销；
- b) 为了遵循按编号申请的次序，暂不使用的资源也需要提前申请，从而增加了进程对资源的占用时间。

## 2. 死锁的避免

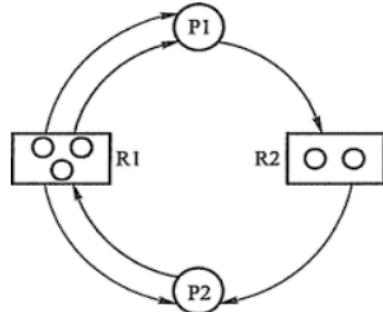
- 银行家算法：

$$\text{Need}(i, j) = \text{Max}(i, j) - \text{Allocation}(i, j)$$

## 3. 死锁的预防

- 资源分配 / 进程-资源图

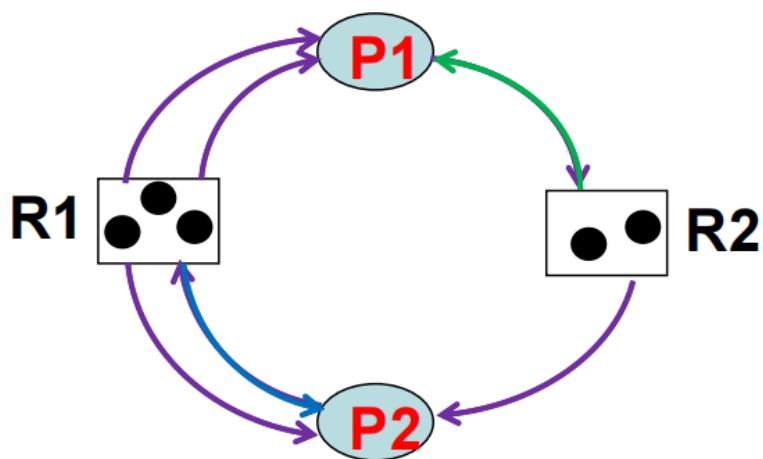
- 用有向图描述系统资源和进程的状态。二元组  $G = (N, E)$ ，  
 $N$ : 结点的集合,  $N = P \cup R$ 。
- $P$ 为进程,  $R$ 为资源,  $P = \{p_1, p_2, \dots, p_n\}$ ,  $R = \{r_1, r_2, \dots, r_m\}$ ,  
两者为互斥资源。
- E: 有向边的集合,  $e \in E$ ,  $e = (p_i, r_j)$  或  $e = (r_j, p_i)$ .
  - $e = (p_i, r_j)$ 是请求边, 进程  $p_i$ 请求一个单位的  $r_j$  资源;
  - $e = (r_j, p_i)$ 是分配边, 为进程  $p_i$ 分配了一个单位的  $r_j$  资源。
- 在资源分配图中, 圆圈表示进程, 矩形表示一类资源, 矩形中的小圈代表每个资源。
- 封锁进程**: 是指某个进程由于请求了超过了系统中现有的未分配资源数目的资源, 而被系统封锁的进程。
- 非封锁进程**: 即没有被系统封锁的进程
- 资源分配图的化简方法**: 假设某个RAG中存在一个进程  $P_i$ , 此刻  $P_i$ 是非封锁进程, 那么可以进行如下化简:
  - 当  $P_i$ 有请求边时, 首先将其请求边变成交配边(即满足  $P_i$  的资源请求), 而一旦  $P_i$ 的所有资源请求都得到满足,  $P_i$  就能在有限的时间内运行结束, 并释放其所占用的全部资源, 此时  $P_i$ 只有分配边, 删去这些分配边(实际上相当于消去了  $P_i$ 的所有请求边和分配边), 使  $P_i$ 成为孤立结点。(反复进行)



## 死锁定理：

系统中某个时刻t为死锁状态的充要条件是**t时刻系统的资源分配图是不可完全化简的。**

在经过一系列的简化后，若能消去图中的所有边，使所有的进程都成为孤立结点，则称该图是**可完全化简的**；反之的是**不可完全化简的**。



#### 4. 死锁的解除

## 两种方法：资源剥夺法、撤销进程法

- **撤销进程**: 使全部死锁的进程夭折掉；按照某种顺序逐个地撤销（回退）进程，直至有足够的资源可用，死锁状态消除为止，
- **剥夺资源**: 使用挂起/激活挂起一些进程，剥夺它们的资源以解除死锁，待条件满足时，再激活进程。

## 5. 输入输出系统

### 5.1 I/O硬件的基本原理

#### 5.1.1 I/O设备分类

- 类型：
  - 传输速度：低速、中速、高速
  - 信息交换单位：块设备和字符设备

- 共享属性：独占设备、共享设备、虚拟设备
- 设备与控制器之间的接口
  - 数据信号
  - 控制信号
  - 状态信号

### 5.1.2 设备管理的目标和功能

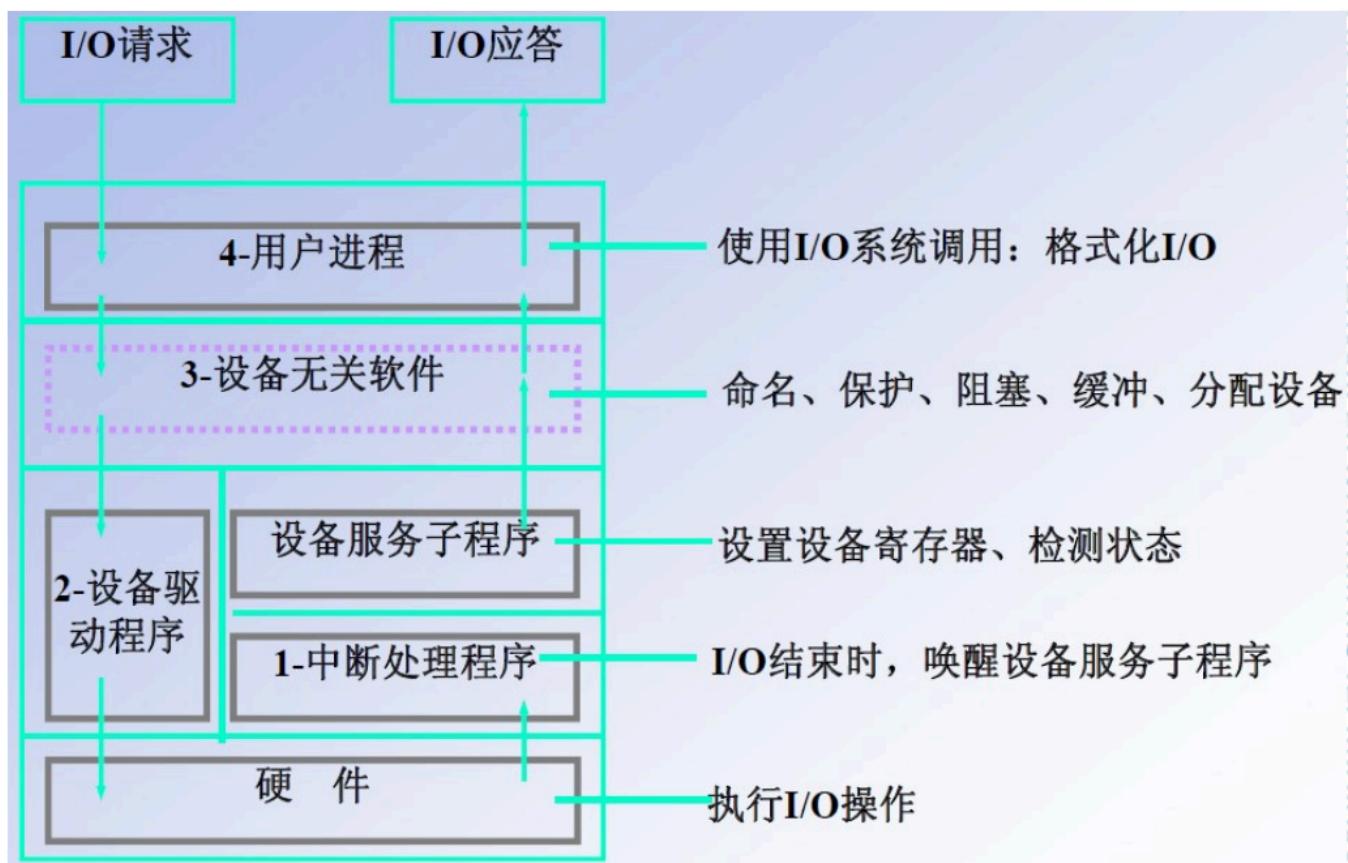
- 设备管理的目标：
  - 提高效率：提高I/O访问效率，匹配CPU和多种不同处理速度的外设
  - 方便使用：方便用户使用，对不同类型的设备统一使用方法，协调对设备的并发使用
  - 方便控制：方便OS内部对设备的控制。例如：增加和删除设备，适应新的设备类型
- 设备管理的功能：
  - 提供设备使用的用户接口：命令接口和编程接口。
  - 设备分配和释放：使用设备前，需要分配设备和相应的通道、控制器。
    - 设备的访问和控制：包括并发访问和差错处理。
    - I/O缓冲和调度：目标是提高I/O访问效率。

### 5.1.3 设备控制器

- 控制设备的功能：
  - 接收和识别CPU命令
  - 数据交换：CPU与控制器、控制器与设备
  - 设备状态的了解和报告
  - 设备地址识别
  - 缓冲区
  - 对设备传来的数据进行差错检测
- 组成：
  - 控制器与处理机接口：**数据寄存器、控制寄存器、状态寄存器**
    - 专门的I/O指令
    - 内存映射
  - 控制器与设备接口
  - I/O逻辑：用于实现CPU对I/O设备的控制

## 5.2 I/O软件的基本原理

### 5.2.1 I/O软件相关的层次关系



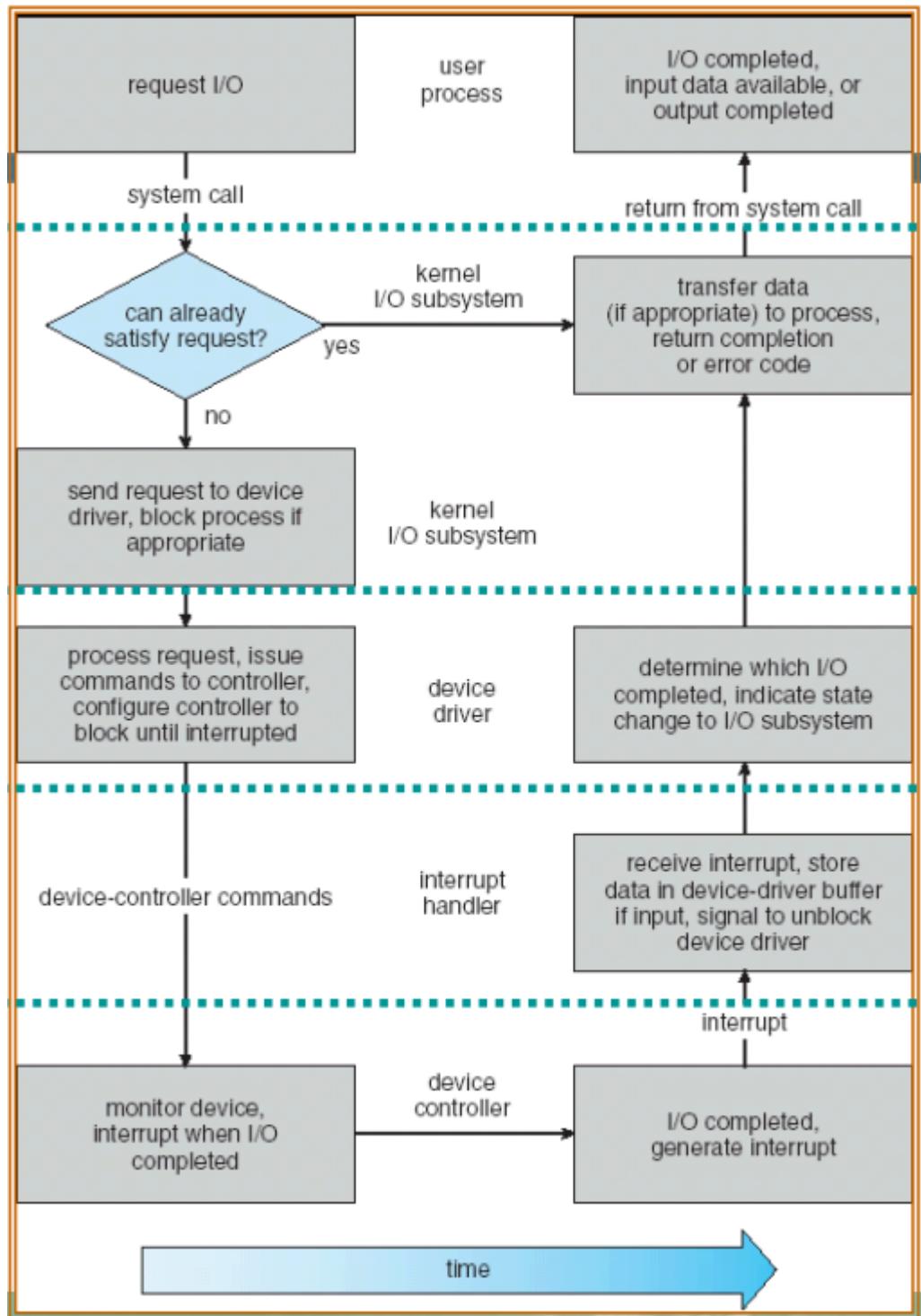
- 分为4个层次，从下至上一次是：中断处理程序、设备驱动程序、设备无关软件、用户进程。

任务	所属 I/O 软件层
计算磁道、扇区、磁头	设备驱动程序
向设备寄存器写命令	设备驱动程序
检查用户是否允许使用设备	设备无关软件
将二进制整数转换成 ASCII 码以便打印	用户进程

#### 关键区分原则

- 硬件相关操作（如寄存器访问、CHS 计算）→ 设备驱动。
- 通用管理（如权限、缓冲、分配设备）→ 设备无关层。
- 数据内容处理（如格式转换）→ 用户进程。
- 中断响应 → 中断处理程序（本题未涉及）

### 5.2.2 I/O请求的处理过程



- 从上到下顺序依次为：用户程序、内核I/O子系统、设备驱动程序上层部分、设备驱动程序下层部分、设备硬件。

### 5.2.3 I/O控制技术

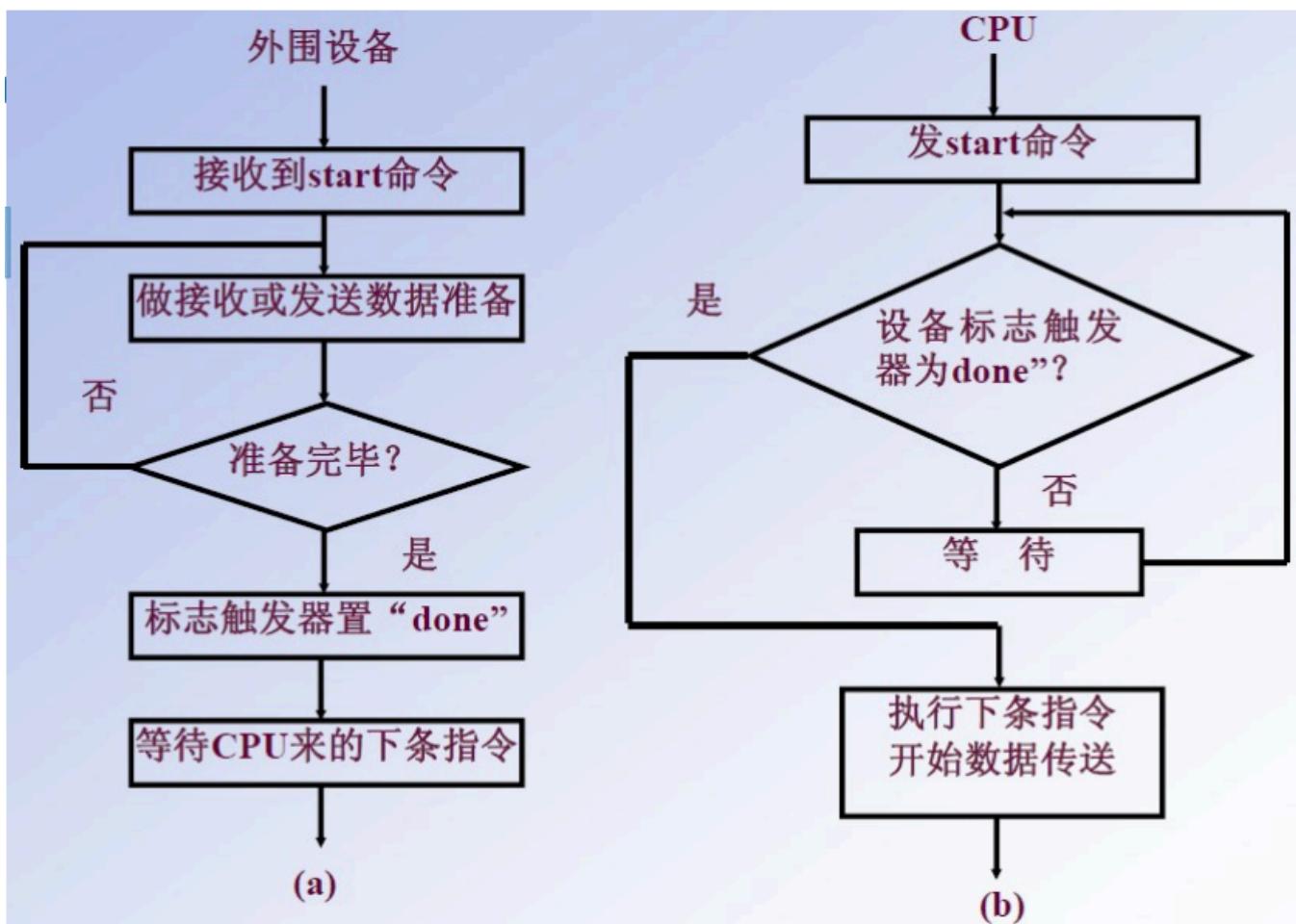
- I/O控制技术有以下四种：
  - 程序控制I/O (PIO, Programmed I/O) :
  - 中断驱动方式 (Interrupt-driven I/O)
  - 直接存储访问方式 (DMA, Direct Memory Access)
  - 通道技术 (Channel)

- 程序控制I/O，也称轮询或查询方式I/O，它由CPU代表进程向I/O模块发出指令，然后进入忙等状态，直到操作完成之后进程才能够继续执行

I/O操作由程序发起，并等待操作完成。数据的每次读写通过CPU。

- 优点：实现简单

- 缺点：CPU利用率相当低，在外设进行数据处理时，CPU只能等待，循环等待中浪费了大量时间。

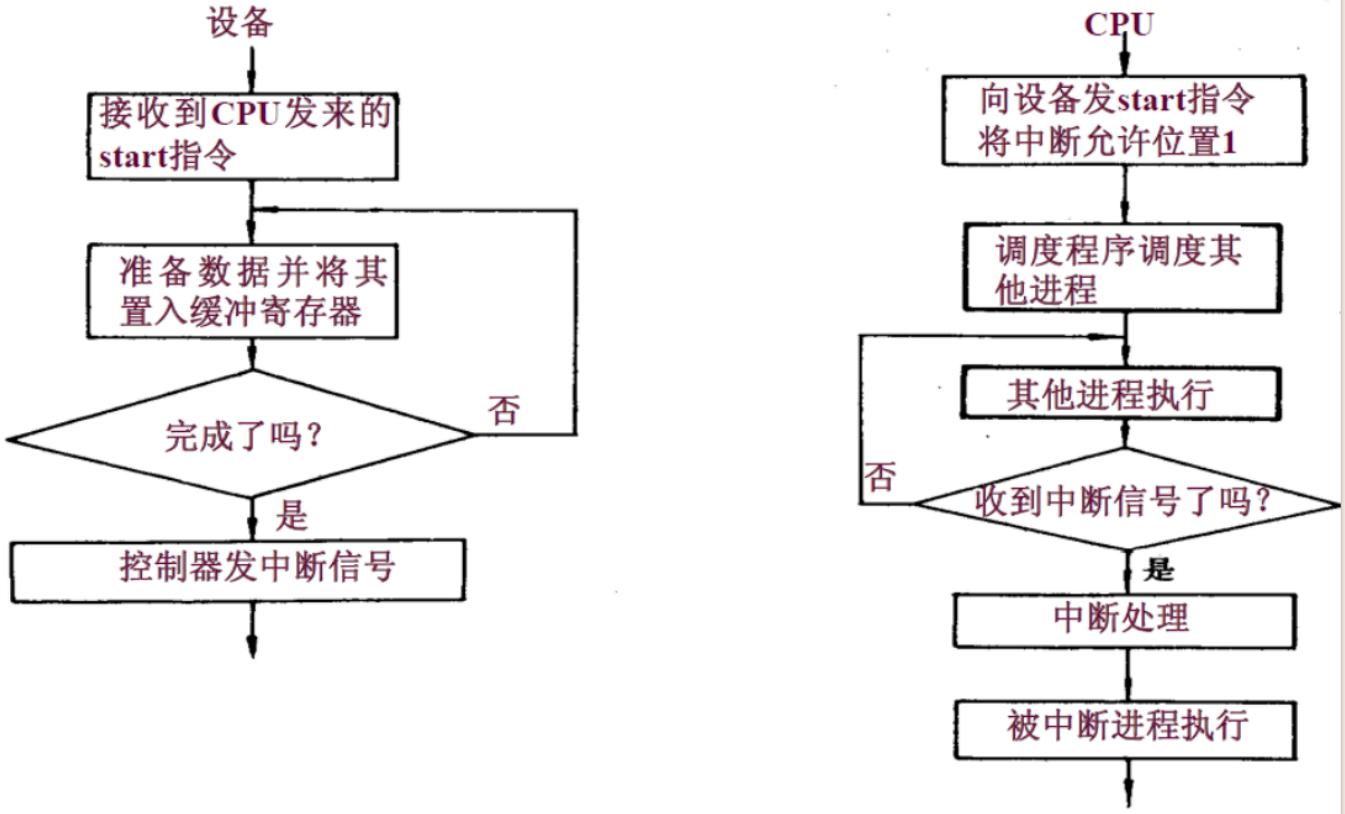


- 中断驱动，当I/O操作结束后由设备控制器主动地来通知设备驱动程序说这次结束，而不是设备驱动程序不断地去轮询看看设备的状态。

I/O操作由程序发起，在操作完成时（如数据可读或已经写入）由外设向CPU发出中断，通知该程序。数据的每次读写通过CPU。

- 优点：在外设进行数据处理时，CPU不必等待，可以继续执行该程序或其他程序，提高了CPU利用率，可以处理不确定事件。

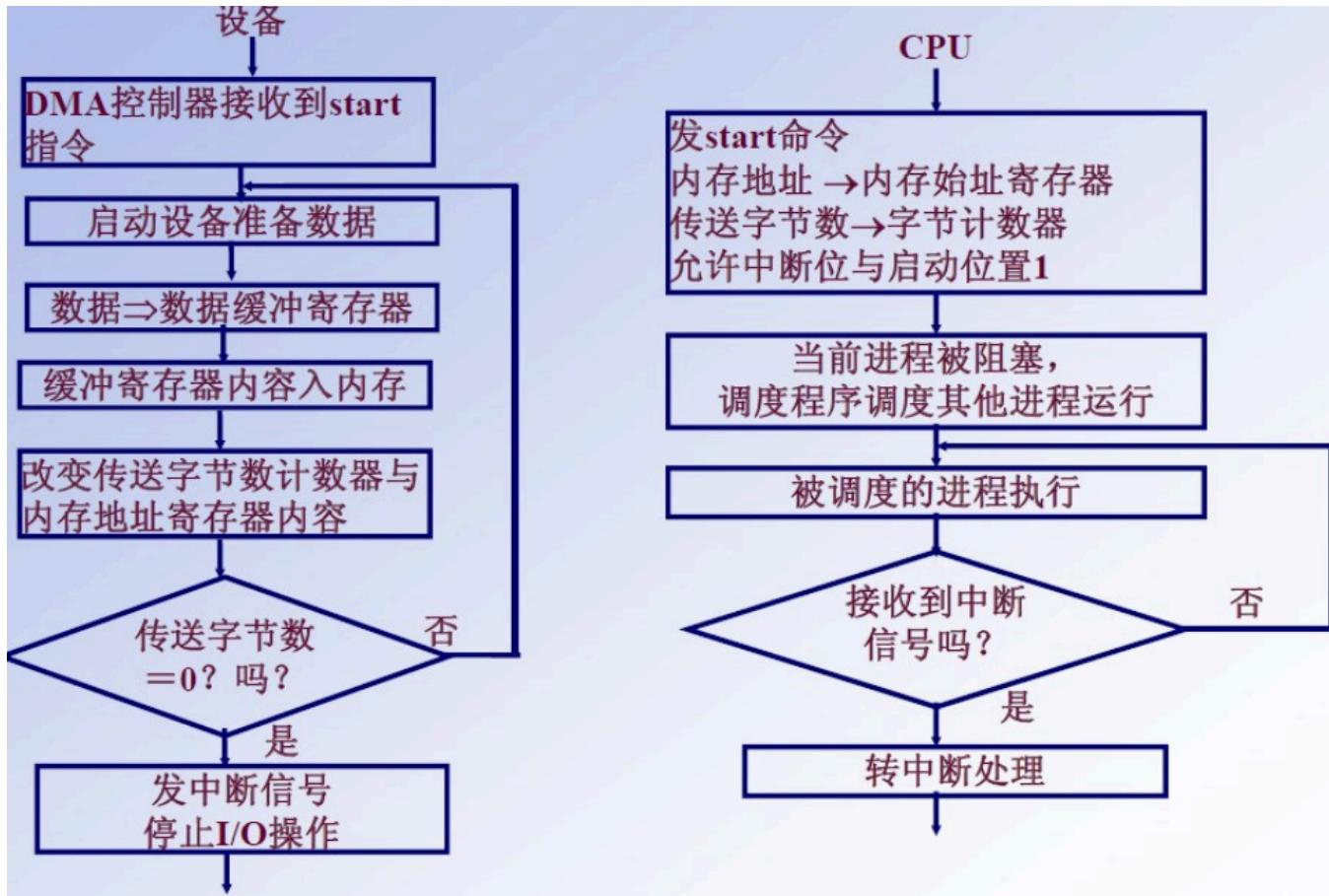
- 缺点：每次输入/输出一个数据都要中断CPU，多次中断浪费CPU时间，只适于数据传输率较低的设备。



- DMA，直接存储器访问方式，是由一个专门的控制器来完成数据从内存到设备或者是从设备到内存的传输工作。

由程序设置DMA控制器中的若干寄存器值（如内存始址，传送字节数），然后发起I/O操作，而后者完成内存与外设的成批数据交换，在操作完成时由DMA控制器向CPU发出中断。

- 优点：CPU只需干预I/O操作的开始和结束，而其中的一批数据读写无需CPU控制，适于高速设备。
- 缺点：数据传送的方向、存放数据的内存地址及传送数据的长度等都由CPU控制，占用了CPU时间。而且每个设备占用一个DMA控制器，当设备增加时，需要增加新的DMA控制器。



- 通道与DMA的原理几乎是一样的，通道是一个特殊功能的处理器，它有自己的指令和程序专门负责数据输入输出的传输控制。CPU将“传输控制”的功能下放给通道后只负责“数据处理”功能。这样，通道与CPU分时使用内存，实现了CPU内部运算与I/O设备的并行工作。

DMA方式的发展，进一步减少CPU干预。把对一个数据块的读写干预，减少为对一组数据块读写的干预。

I/O通道是专门负责输入输出的处理器，独立于CPU，有自己的指令体系。可执行由通道指令组成的通道程序，因此可以进行较为复杂的I/O控制。通道程序通常由操作系统所构造，放在内存里。

- 优点：执行一个通道程序可以完成几组I/O操作，与DMA相比，减少了CPU干预。
- 缺点：费用较高。

- DMA与中断方式的区别：

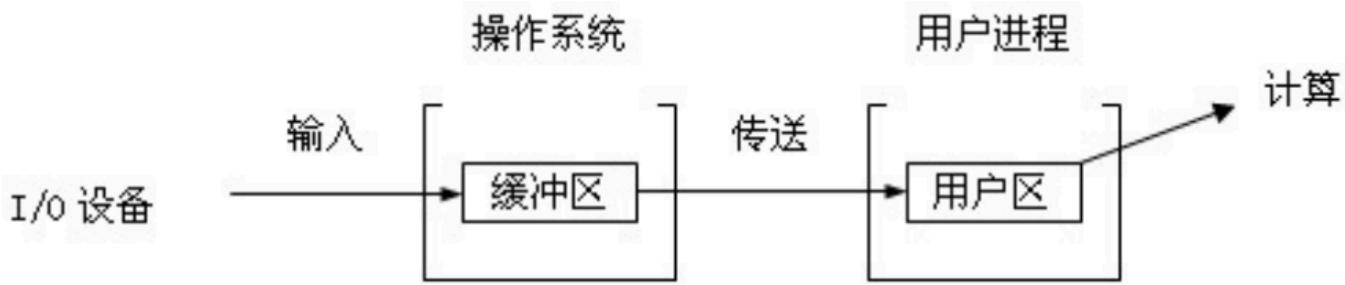
- 中断控制方式在每个数据传送完成后中断CPU；DMA控制方式是在要求传送的一批数据完成之后中断CPU。
- 中断控制方式的数据传送是在中断处理时由CPU控制完成的，由于程序陷入内核，需要保护和恢复现场。DMA方式下是由DMA控制器控制完成的，在传输过程中不需要CPU干预，DMA控制器直接在主存和I/O设备之间传送数据，只有开始和结束才需要CPU干预。
- 程序中断方式具有对异常事件的处理能力，而DMA控制方式适用于数据块的传输。

- I/O通道与DMA的区别：

- DMA方式下，数据的传送方向、存放数据的内存起始地址和数据块长度都由CPU控制；而通道是一个特殊的处理器，有自己的指令和程序，通过执行通道程序实现对数据传输的控制，所以通道具有更强的独立处理I/O的功能。
- DMA控制器通常只能控制一台或者少数几台同类设备；而一个通道可同时控制多种设备。

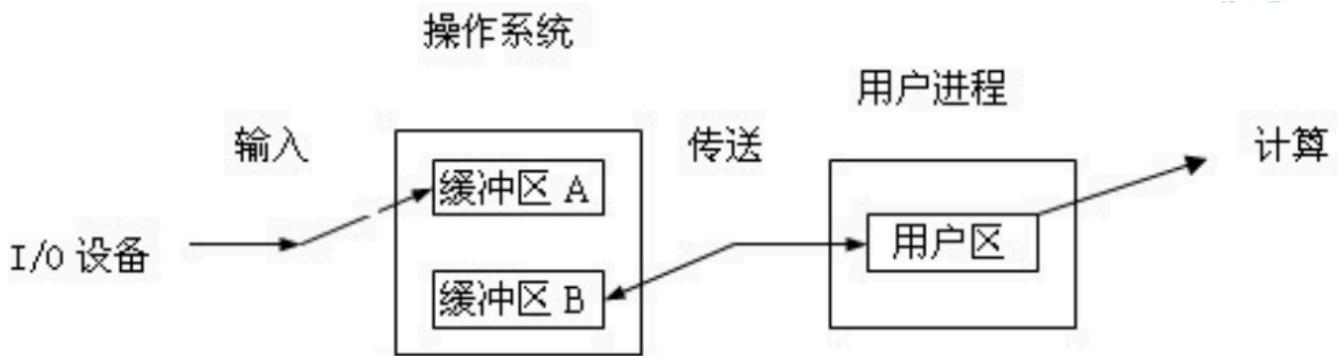
#### 5.2.4 缓冲技术

- 缓冲技术可提高外设利用率。
  - 匹配CPU与外设的不同处理速度
  - 减少对CPU的中断次数
  - 提高CPU和I/O设备之间的并行性
- 单缓冲：一个缓冲区，CPU和外设轮流使用，一方处理完之后接着等待对方处理。



I/O 的单缓冲方式

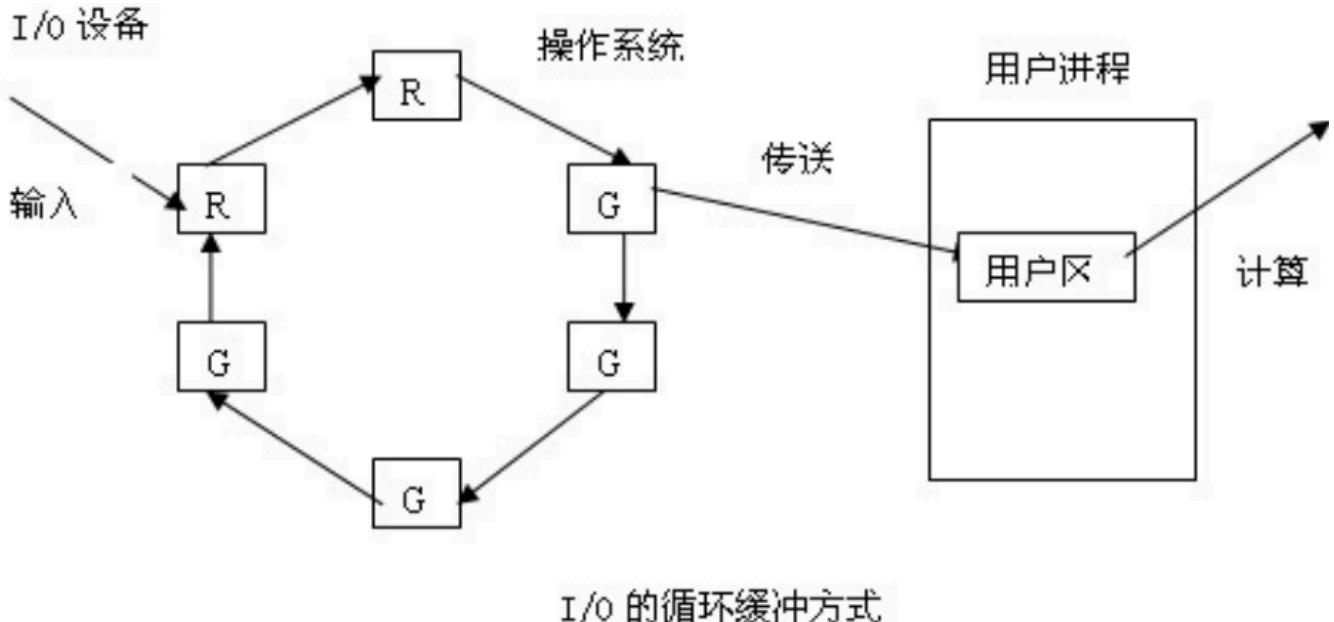
- 双缓冲：两个缓冲区，CPU和外设都可以连续处理而无需等待对方。要求CPU和外设的速度相近。



I/O 的双缓冲方式

- 环形缓冲：多个缓冲区，CPU和外设的处理速度可以相差较大。

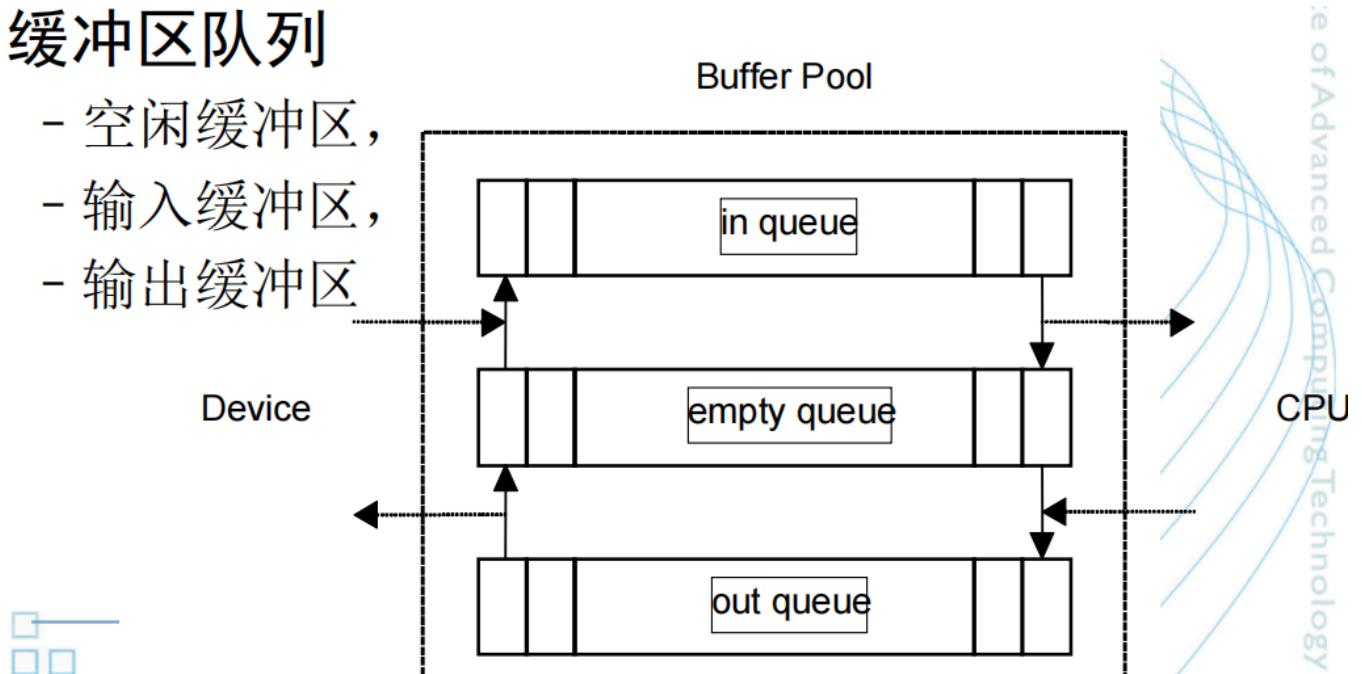
可参见“生产者—消费者问题”。



- 缓冲池：缓冲区整体利用率高

## 缓冲区队列

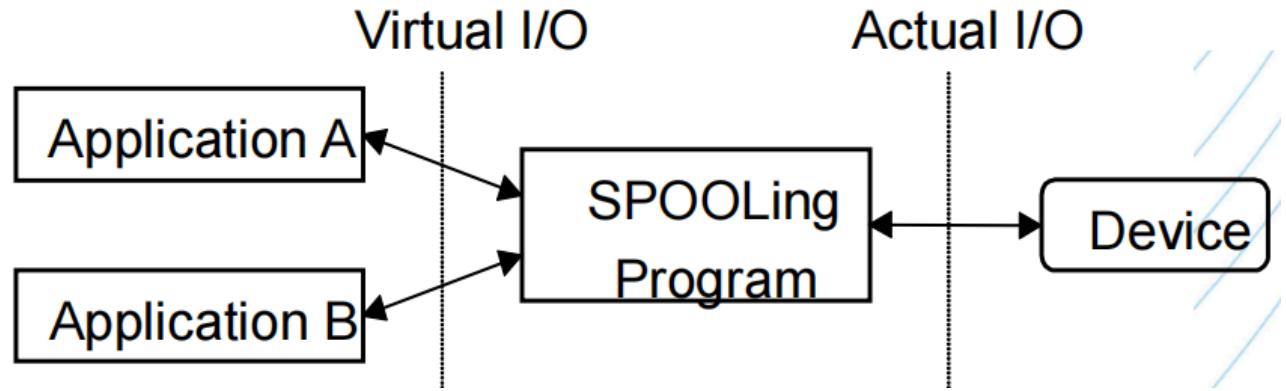
- 空闲缓冲区，
- 输入缓冲区，
- 输出缓冲区



### 5.2.5 设备分配

- 单通路：一个设备对应一个控制器，一个控制器对应一个通道。
- 多通路：一个设备与几个控制器相连，一个控制器与几个通道相连。
  - 分配设备
  - 分配控制器
  - 分配通道
- 假脱机技术：利用假脱机技术(SPOOLing, Simultaneous Peripheral Operation On Line, 也称为虚拟设备技术)可把独享设备转变成具有共享特征的虚拟设备，从而提高设备利用率。

在多道程序系统中，专门利用一道程序（SPOOLing程序）来完成对设备的I/O操作。无需使用外围I/O处理机。



特点：

- 高速虚拟I/O操作：应用程序的虚拟I/O比实际I/O速度提高，缩短应用程序的执行时间。另一方面，程序的虚拟I/O操作时间和实际I/O操作时间分离开来。
- 独享设备的共享：由SPOOLing程序提供虚拟设备，可以对独享设备依次共享使用。

举例：打印机设备和可由打印机管理器管理的打印作业队列。如：Windows NT中，应用程序直接向针式打印机输出需要15分钟，而向打印作业队列输出只需要1分钟，此后用户可以关闭应用程序而转入其他工作，在以后适当的时候由打印机管理器完成15分钟的打印输出而无需用户干预。

### 5.2.6 中断处理过程

- 中断处理过程：
  - 关中断
  - 保存现场（包括断点）
  - 转入设备中断处理程序
  - 进行中断处理
  - 恢复被中断进程的现场
  - 开中断
  - 返回断点（断点在哪里？）
- 设备驱动程序的功能：
  - 将抽象I/O请求转换为对物理设备的请求
  - 检查I/O请求的合法性
  - 初始化设备
  - 启动设备
  - 发出I/O命令
  - 响应中断请求
  - 构造通道程序
- 设备驱动程序的组成：
  - 自动配置和初始化子程序：检测所要驱动的硬件设备是否存在、是否正常。如果该设备正常，则对该设备及其相关的设备驱动程序需要的软件状态进行初始化。在初始化的时候被调用一次。
  - 服务于I/O请求的子程序：调用该子程序是系统调用的结果。执行该部分程序时，系统仍认为是和调用进程属同一个进程，只是由用户态变成核心态，具有进行此系统调用的用户程序的运行环境，可在其中调用sleep()等与进程运行环境有关的函数。

- 中断服务子程序：系统来接收硬件中断，再由系统调用中断服务子程序。因为设备驱动程序一般支持同一类型的若干设备，所以一般在系统调用中断服务子程序的时候，都带有一个或多个参数，以唯一标识请求服务的设备。
- 以下各项任务是在四个I/O软件层的哪一层完成的？
  - 为一个磁盘读操作计算磁道、扇区、磁头
  - 向设备寄存器写命令
  - 检查用户是否允许使用设备
  - 将二进制整数转换成ASCII码以便打印

解答

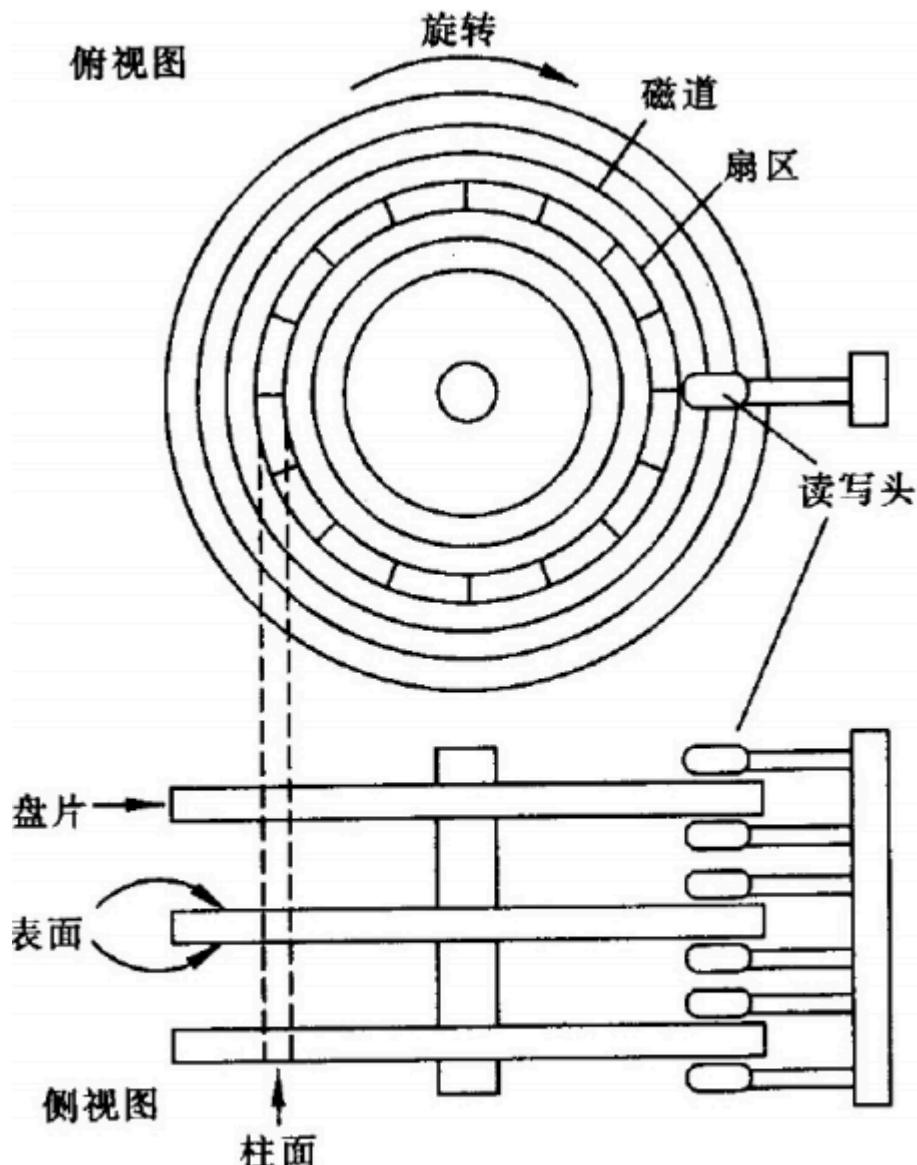
- 设备驱动程序
- 设备驱动程序
- 设备无关软件
- 用户程序

## 6. 磁盘存储管理

### 6.1 磁盘存储的工作原理

#### 6.1.1 磁盘的基本概念

- 扇区 (sector)：盘片被分成许多扇形的区域
- 磁道 (track)：盘片上以盘片中心为圆心，不同半径的同心圆。
- 柱面 (cylinder)：硬盘中，不同盘片相同半径的磁道所组成的圆柱。
- 注意：
  - 每个磁盘有两个面，每个面都有一个磁头(head)。
  - 柱面的数量 = 磁道的数量；盘面的数量 = 磁头的数量



磁盘设备结构示意图

### 6.1.2 磁盘的组织

- 读一个扇区需要柱面/磁头/扇区。
- 现代磁盘驱动器可以看做一个一维的逻辑块的数组，逻辑块是最小的传输单位。
- 一维逻辑块数组按顺序映射到磁盘的扇区。

### 6.1.3 磁盘的访问时间

- 寻道时间：这是把磁臂（磁头）从当前位置移动到指定磁道上所经历的时间。该时间是启动磁盘的时间s与磁头移动n条磁道所花费的时间之和。

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

- 旋转延迟时间：对于硬盘，假设旋转速度为3600 RPM，则每转需时16.7 ms，平均旋转延迟时间 $T_r = 8.3$  ms。

对于软盘，其旋转速度为300或600 r/min，这样， $T_r$ 为50~100 ms。

$$Tr = 1/(2r)$$

- **传输时间：** $T_t$ 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，旋转速度r以及磁道上的字节数N有关。
  - 传输数据的大小(通常是1个扇区): 512B
  - 旋转速度r: 3600 RPM ~ 15000 RPM
  - 典型的传输速度: 2MB~50 MB/秒

$$T_t = b/(rN)$$

- 则总访问时间 $T_a$ 为：

$$T_a = T_s + T_r + T_t = T_s + 1/(2r) + b/(rN)$$

- 可以看出：假定寻道时间和旋转延迟时间平均为30 ms，而磁道的传输速率为1 MB/s，如果传输1K字节，此时总的访问时间为31 ms，传输时间所占比例是相当地小。当传输10K字节的数据时，其访问时间也只是40 ms，即当传输的数据量增加10倍时，访问时间只增加了约30% ——抓主要矛盾

### 6.1.3 磁盘调度算法

- 先来先服务算法：(FCFS first come first served)
  - 算法思想：按访问请求到达的先后次序服务。
  - 优点：简单，公平。
  - 缺点：
    - 效率不高，相邻两次请求可能会造成最内到最外的柱面寻道，使磁头反复移动，增加了服务时间，对机械也不利。
- 最短寻道时间优先算法：(SSTF shortest seek time first)
  - 算法思想：优先选择距当前磁头最近的访问请求进行服务，主要考虑寻道优先。
  - 优点：改善了磁盘平均服务时间。
  - 缺点：可能产生“饥饿”现象，造成某些访问请求长期等待得不到服务。
- 扫描算法：(SCAN, 电梯算法)
  - 算法思想：当有访问请求时，磁头按一个方向移动，在移动过程中对遇到的访问请求进行服务，然后判断该方向上是否还有访问请求，如果有则继续扫描；否则改变移动方向，并为经过的访问请求服务，如此反复。
  - 优点：克服了最短寻道优先的缺点，既考虑了距离，同时又考虑了方向。
  - 缺点：但由于是摆动式的扫描方法，两侧磁道被访问的频率仍低于中间磁道。
- 循环扫描算法 (CSCAN)
  - 算法思想：按照所要访问的柱面位置的次序去选择访问者。移动臂到达最后一个柱面后，立即带动读写磁头快速返回到0号柱面。返回时不为任何的等待访问者服务。返回后可再次进行扫描。

- 由于SCAN算法偏向于处理那些接近最里或最外的磁道的访问请求，所以使用改进型的C-SCAN算法可避免这个问题。
- 算法比较：

	<b>优点</b>	<b>缺点</b>
<b>FCFS算法</b>	公平、简单	平均寻道距离大，仅应用在磁盘I/O较少的场合
<b>SSTF算法</b>	性能比“先来先服务”好	不能保证平均寻道时间最短，可能出现“饥饿”现象
<b>SCAN算法</b>	寻道性能较好，可避免“饥饿”现象	不利于远离磁头一端的访问请求
<b>C-SCAN算法</b>	消除了对两端磁道请求的不公平	--

- 其他改进算法：

- “磁臂粘着”现象：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。
- N-Step-SCAN：将请求队列分成分成长为N的子队列，队列之间采用FCFS，队列内部采用SCAN。N很大 $\sim$ SCAN； $N=1\sim FCFS$ 。
- FSCAN：Fixed Period SCAN：请求队列分为两个子队列（当前请求和新请求）

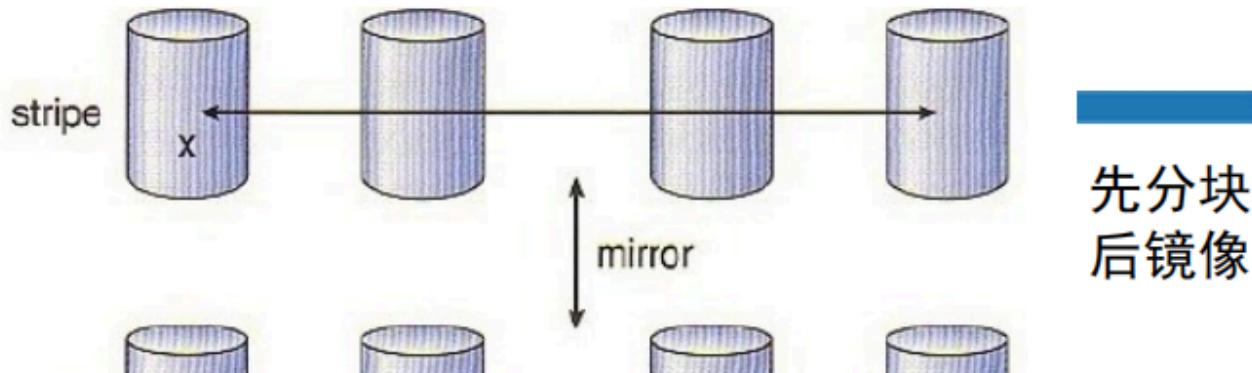
## 6.2 保证磁盘的可靠性

### 6.2.1 RAID

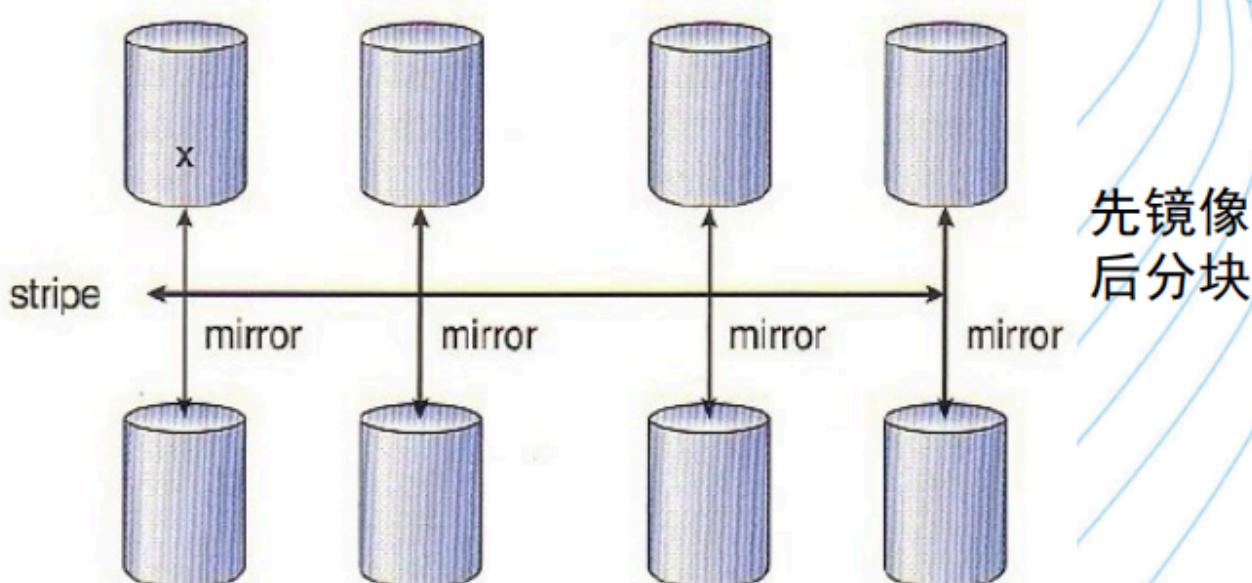
- 一种把多块独立的硬盘（物理硬盘）按照不同方式组合起来形成一个硬盘组（逻辑硬盘），从而提供比单个硬盘更高的存储性能和提供数据冗余的技术。
- 组成磁盘阵列的不同方式称为RAID级别(RAID Levels)。
- 数据冗余的功能是在用户数据一旦发生损坏后，利用冗余信息可以使损失数据得以恢复，从而保障了用户数据的安全性。
- RAID的优点：
  - a. 成本低，功耗小，传输速率高。
    - RAID比起传统的大直径磁盘驱动器来，在同样的容量下，价格要低许多。
    - RAID让很多磁盘驱动器并行传输数据，比单个磁盘驱动器提高几倍、几十倍甚至上百倍的速率。有效缓解了快速的CPU与慢速的磁盘之间的矛盾。
  - b. 可提供容错功能
    - 普通磁盘驱动器只能通过CRC(循环冗余校验)码提供简单的容错，RAID建立在每个磁盘驱动器的硬件容错功能之上，可提供更高的安全性。

### 6.1.2 RAID的分级

- RAID有六个级别,其级别分别是0、1、2、3、4、5、6,后来还出现了RAID 0+1、RAID 1+0等组合级别。
- RAID 0: 该级仅提供了并行交叉存取。它虽然有效提高了磁盘I/O速度,但并无冗余校验功能。
- RAID 1: 镜像磁盘冗余阵列,将每一数据块重复存入镜像磁盘,以改善磁盘机的可靠性。镜像盘也称拷贝盘,使有效容量下降了一半,成本较高。
  - RAID 1+0 和 0+1



a) RAID 0 + 1 with a single disk failure.



b) RAID 1 + 0 with a single disk failure.

- n块盘中1块盘故障都不会引起数据丢失。
- 2块盘故障引起数据丢失概率:
  - RAID 0+1:  $n/(2*n-2)$  或  $(n/2 * n/2) / C_n^2$
  - RAID 1+0:  $1/(n-1)$
- RAID 2: 采用海明码纠错的磁盘阵列,将数据位交叉写入几个磁盘中。按位条带化。
  - $a = b \text{ XOR } c \text{ XOR } d$

$b = a \text{ XOR } c \text{ XOR } d$

$c = a \text{ XOR } b \text{ XOR } d$

$d = a \text{ XOR } b \text{ XOR } c$

- XOR数据恢复原理：RAID容错就是利用了XOR运算的这些特点。a、b、c、d就可以看作是四颗磁盘上的数据，其中三个是应用数据，剩下一个校验。碰到故障的时候，不管哪个找不到了，都可以用剩下的三个数字XOR一下算出来。在实际应用中，阵列控制器一般要先把磁盘分成很多条带，然后再对每组条带做XOR。

- RAID 2的特点：

- 并行存取，各个驱动器同步工作。
- 使用海明编码来进行错误检测和纠正，数据传输率高。
- 需要多个磁盘来存放海明校验码信息，冗余磁盘数量与数据磁盘数量的对数成正比。
- 是一种在多磁盘易出错环境中的有效选择，并未被广泛应用，目前还没有商业化产品。

- RAID 3：采用奇偶校验冗余的磁盘阵列，也采用数据位交叉，阵列中只有一个校验盘。

- RAID 3的特点：将磁盘分组，采用字节级别的条带，读写要访问组中所有盘，每组中有一个盘作为校验盘。校验盘一般采用奇偶校验。

- 缺点：

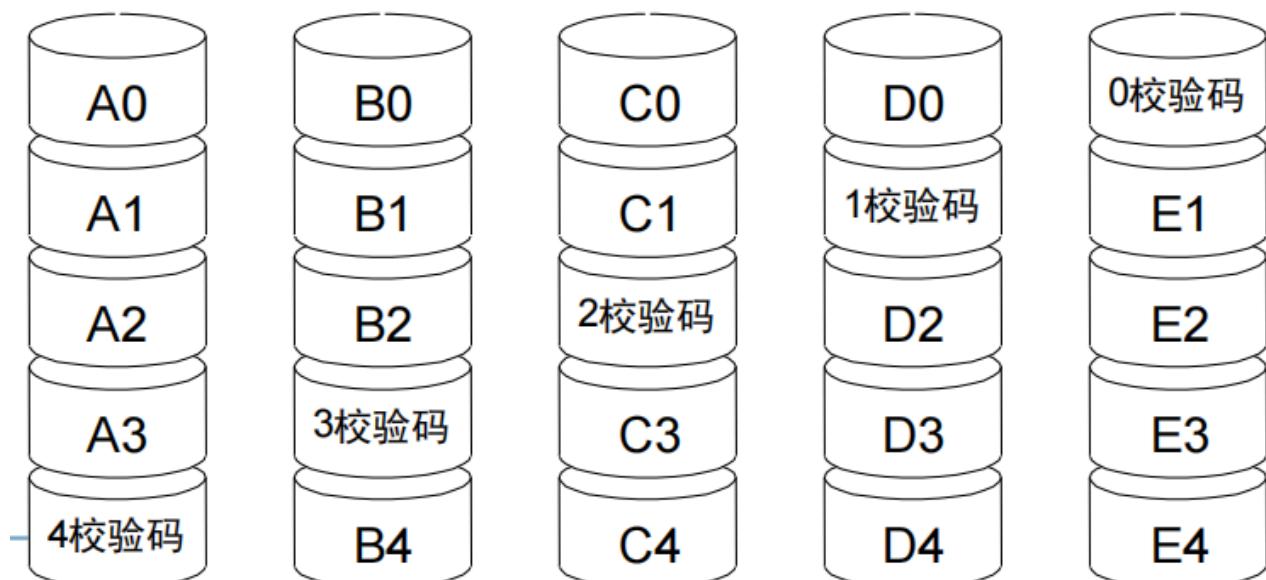
- 恢复时间较长
- 读写性能的水桶效应瓶颈

- RAID 4：并行处理磁盘阵列：一种独立传送磁盘阵列，采用数据块交叉，用一个校验盘。将数据按块交叉存储在多个磁盘上。

- RAID 4特点：

- 冗余代价与RAID 3相同
- 访问数据的方法与RAID 3不同，RAID4出现的原因：希望使用较少的磁盘参与操作，以使磁盘阵列可以并行进行多个数据的磁盘操作
- 随机读快，随机写慢（竞争同一个校验盘）

- RAID 5：一种独立传送磁盘阵列，采用数据块交叉和分布的冗余校验，将数据和校验都分布在各个磁盘中，没有专门的奇偶校验驱动器。



- RAID 6：双维校验独立存取盘阵列，数据以块（块大小可变）交叉方式存于各盘，检、纠错信息均匀分布在所有磁盘上。

- RAID 6的特点：
  - 写入数据要访问1个数据盘和2个冗余盘；
  - 可容忍双盘出错；
  - 存储开销是RAID5的两倍（多一个冗余盘）。

- RAID小结：

- 条带化：一个字节块可能存放在多个数据盘上
  - 优点：并行存取，性能好，磁盘负载均衡
  - 缺点：可靠性、不同IO请求需要排队
- 镜像：数据完全拷贝一份
  - 优点：可靠性
  - 缺点：存储开销
- 校验：数据通过某种运算（异或）得出，用以检验该组数字的正确性
  - 优点：可靠性，快速恢复
  - 缺点：开销

0、1、5、6

## RAID比较

级别	说明	特点	磁盘数量	最少磁盘数量
0	条带化	速度加倍，可靠性减半，容量不变	n	2
1	镜像	速度不变，可靠性加倍，容量减半	$n^2$	2
0+1	组内条带化、组间镜像	速度加倍，可靠性加倍，容量减半	$n^2$	4
1+0	组内镜像，组间条带化	速度加倍，可靠性加倍，容量减半	$2^n$	4
2	海明码校验	校验独立存储，数据位交叉（条带化），自动纠正1个盘错	$n+\log(n)$	7*
3	奇偶校验	校验独立存储，数据位交叉（条带化）容1块盘错	$n+1$	3
4	奇偶校验	校验独立存储，数据块交叉，容1块盘错	$n+1$	3
5	奇偶校验	校验交叉存储，数据块交叉，容1块盘错	$n+1$	3
6	奇偶校验	校验交叉存储，数据块交叉，容2块盘错	$n+2$	4

## 6.3 提高I/O访问速度

### 6.3.1 提高I/O速度的主要途径

- 选择性能好的磁盘
- 并行化
- 采用适当的调度算法
- 设置磁盘高速缓冲区

### 6.3.2 提高磁盘I/O速度——缓存

- 磁盘高速缓存的形式
  - 独立缓存（固定大小）
  - 以虚拟内存为缓存（弹性大小）
- 数据交付
  - 直接交付（copy开销）
  - 指针交付（内存管理复杂）
- 置换算法（LRU）
- 周期性写回
  - 周期性地将disk cache中被修改过的内容写回磁盘

### 6.3.3 优化数据布局

- 优化物理块的分布：连续摆放，使磁头的移动距离最小，比如：安排一个文件的两块数据块在同一磁道，消除磁头在磁道间的移动（Defreg）
- 优化索引节点的分布：
  - 访问文件时，先访问索引节点，然后再去访问盘块；
  - 将磁盘上的所有磁道分成若干个组，在每个组中都含有索引节点、盘块和空闲盘块表，使得索引节点盘块与存放文件的盘块间距离最小（如：Linux的磁盘文件逻辑结构）

### 6.3.4 提高磁盘I/O速度的其他方法

- 提前读：顺序访问时，常采用提前读入下一块到缓冲区中
- 延迟写：将本应立即写回磁盘的数据挂到空闲缓冲区的队列的末尾。直到该数据块移到链头时才将其写回磁盘，再作为空闲区分配出去。
- 虚拟盘：
  - 利用内存空间去仿真磁盘（RAM盘）
  - Virtual disk与disk cache的区别是：
    - Virtual disk的存放的内容由用户完全控制
    - Disk cache中的内容完全是由操作系统控制

## 6.4 磁盘管理实例

略

# 7. 文件系统

## 7.1 文件系统基本概念

### 7.1.1 文件的概念

- 定义：所谓文件是指一组带标识（标识即为文件名）的、在逻辑上有完整意义的信息项的序列。
- **信息项**：构成文件内容的基本单位（单个字节，或多个字节），各信息项之间具有顺序关系。
- 文件包括两部分：
  - 文件体：文件本身的内容；(data)
  - 文件说明：文件存储和管理的相关信息，如：文件名、文件内部标识、文件存储地址、访问权限、访问时间等；(meta-data)
- 文件可以视为一个**单独的**连续的逻辑地址空间，其大小即为文件的大小，与进程的地址空间无关。
- 文件的**本质**是一组字节序列

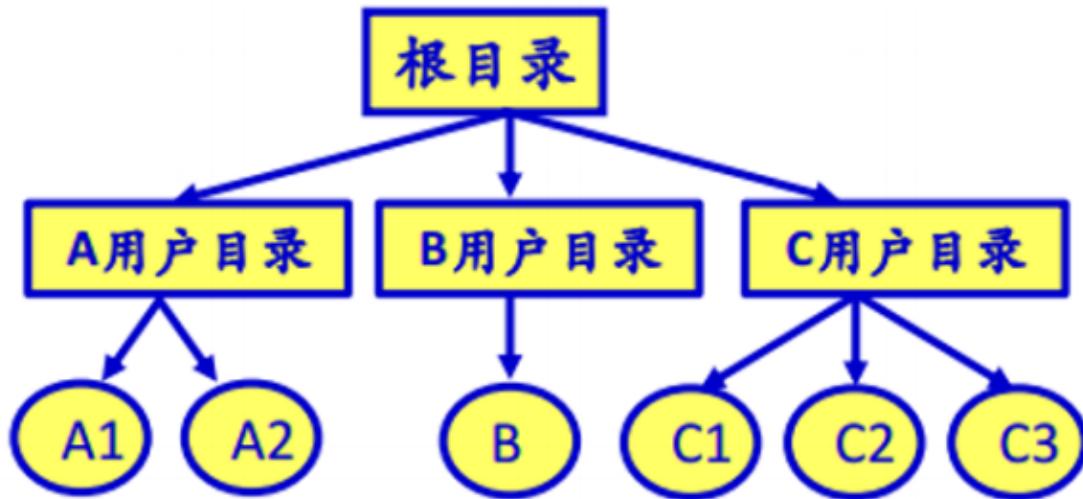
### 7.1.2 文件的管理

- 文件是通过操作系统来管理的，包括：文件的结构、命名、存取、使用、保护和实现方法。
- 文件管理的需求：
  - 用户视角（使用逻辑文件）：
    - 用户关心文件中要使用的数据，不关心具体的存放形式（和位置）。
    - 关心的是文件系统所提供的对外的用户接口，包括文件如何命名、如何保护、如何访问（创建、打开、关闭、读、写等）；
  - 操作系统视角（组织和管理物理文件）
    - 文件的描述和分类，关心的是如何来实现与文件有关的各个功能模块，包括如何来管理存储空间、文件系统的布局、文件的存储位置、磁盘实际运作方式（与设备管理的接口）等。
- 文件系统：文件系统是操作系统中统一管理信息资源的一种软件，管理文件的存储、检索、更新，提供安全可靠的共享和保护手段，并且方便用户使用。
- 文件名：当一个文件被创建时，必须给它指定一个名字，用户通过文件名来访问文件。
- 典型的文件结构：
  - 流式文件：构成文件的基本单位是字符。文件是有逻辑意义、无结构的一串字符的集合；
  - 记录式文件：文件由若干记录组成，可以按记录进行读写、查找等操作。每条记录有其内部结构。
- 文件存取方式：
  - 顺序存取（访问）；
  - 随机存取：提供读写位置（当前位置）。

### 7.1.3 目录

- 目录的内容是文件属性信息(properties)，其中的一部分是用户可获取的。
- 文件目录分类：
  - 单级文件
  - 二级文件
  - 多级文件

如图为二级文件：



多级目录特点：

- 层次清楚
- 可解决文件重名问题
- 查找速度快
- 目录级别太多时，会增加路径检索时间

### 7.1.4 文件系统

- 定义：操作系统中与文件管理有关的那部分软件和被管理的文件以及实施管理所需要的数据结构的总体。
- 文件系统管理的对象有：
  1. 文件
  2. 目录
  3. 磁盘存储空间

## 7.2 文件系统实现方法

- 文件物理结构的比较：

- **连续文件：**优点是不需要额外的空间开销，只要在文件目录中指出文件的大小和首块的块号即可，对顺序的访问效率很高。适应于顺序存取。缺点是动态地增长和缩小系统开销很大；文件创建时要求用户提供文件的大小；存储空间浪费较大。
- **串联文件：**克服了连续文件的不足之处，但文件的随机访问系统开销较大。适应于顺序访问的文件。DOS系统中改造了串联文件的结构，使其克服了串联文件的不足，但增加了系统的危险性。
- **索引文件：**既适应于顺序存访问，也适应于随机访问，是一种比较好的文件物理结构，但要有用于索引表的空间开销和文件索引的时间开销。UNIX系统是使用索引结构成功的例子。