一、计算机操作系统概述

1.1 计算机系统概述

1.1.1 计算机发展历史

- 1945年，电子真空管出现，机器语言应用于科学计算
- 1956年，晶体管出现，批处理控制，数据处理领域
- 1959年，集成电路、多道程序出现，操作系统/数据库/高级语言
- 1976年，大规模/超大规模集成电路出现
- 1980年，微机出现，廉价化促使应用领域快速发展
- 1990年，图形化人机交互技术推动应用人群快速发展
- 2003年，移动计算出现

1.1.2 计算机系统的组成

计算机系统：包括硬件子系统和软件子系统。

- 硬件：CPU、主存储器、I/O控制设备、外围设备
- 软件：系统软件、支撑软件和应用软件
    - 最关键的系统软件：操作系统与语言处理程序

1.2 计算机硬件系统

1.2.1 计算机硬件系统的组成

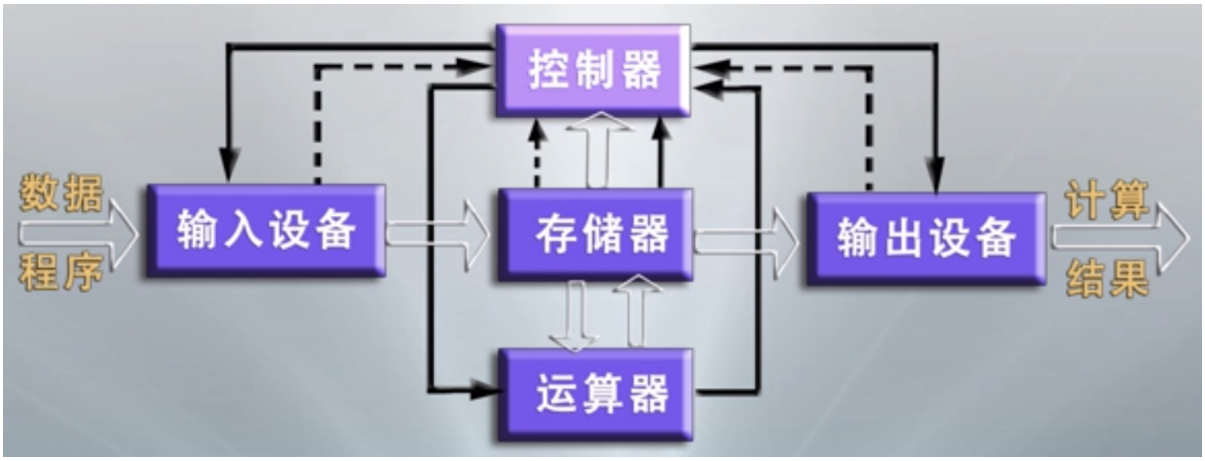
- 中央处理器：运算单元、控制单元
- 主存储器
- 外围设备：输入 / 输出设备、存储设备（硬盘）、网络通信设备
- 总线

1.2.2 储存程序计算机

> 主要特点

1. 以运算单元为中心，控制流由指令流产生
2. 采用存储程序原理，面向主存组织数据流
3. 主存是按地址访问、线性编址的空间
4. 指令由操作码和地址码组成

5. 数据以二进制编码



（当今计算机硬件的经典结构和主流组织方式，存储器为中心）

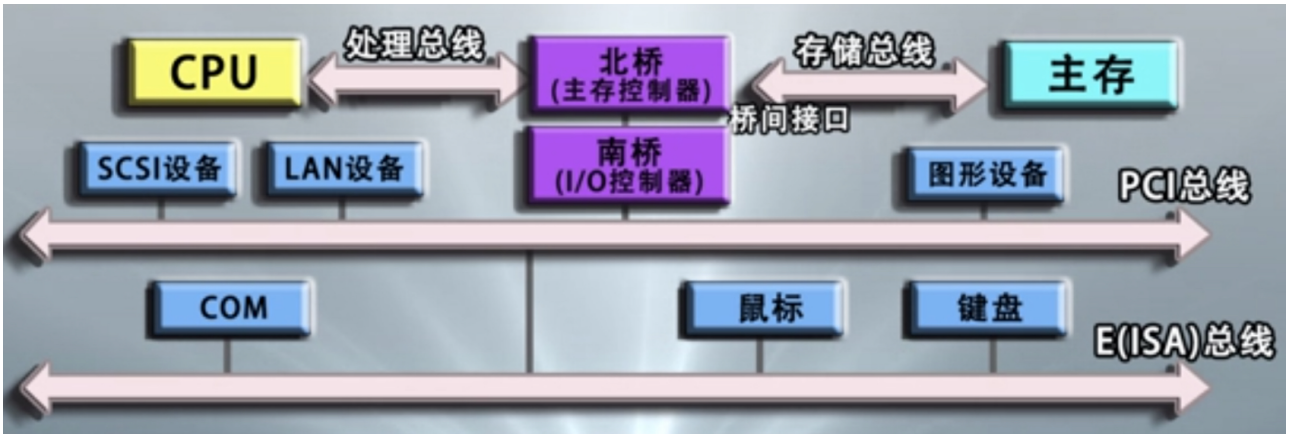
1.2.3 总线及其组成

> 简要介绍

1. 总线（Bus）是计算机各种功能部件之间传送信息的公共通信干线，它是CPU、内存、输入输出设备传递信息的公用通道。
2. 计算机的各个部件通过总线相连接，外围设备通过相应的接口电路再与总线相连接，从而形成了计算机硬件系统。
3. 按照所传输的信息种类，总线包括一组控制线、一组数据线和一组地址线。

> 总线的类型

1. 内部总线：用于CPU芯片内部连接各元件
2. 系统总线：用于连接CPU、存储器和各种I/O模块等主要部件
3. 通信总线：用于计算机系统之间通信

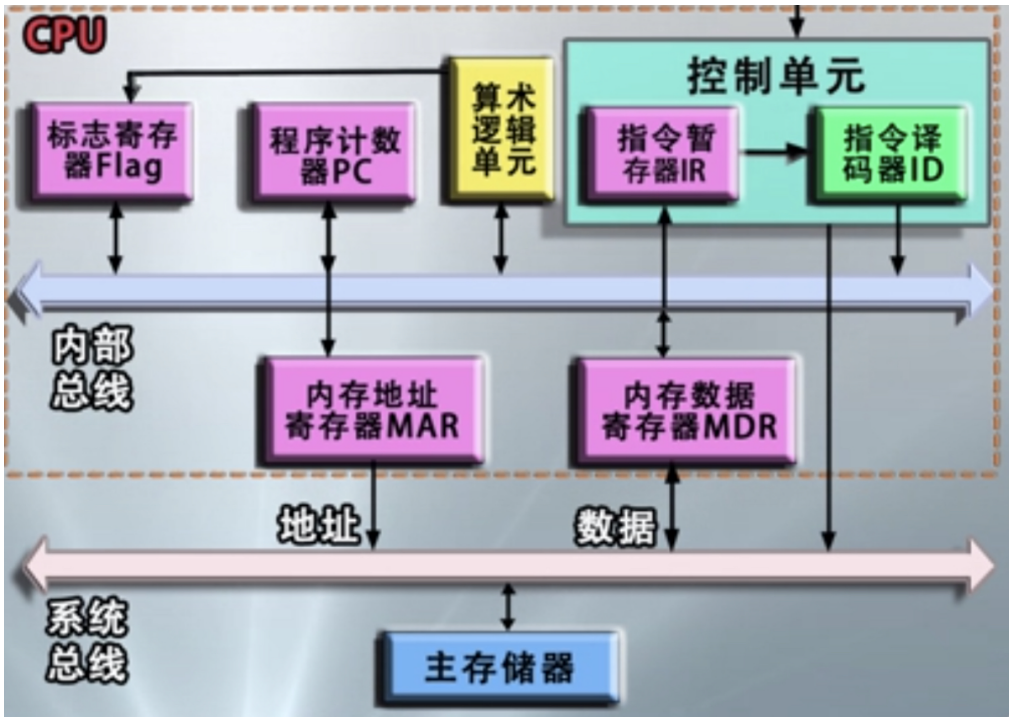


1.2.4 中央处理器（CPU）

> 简要介绍

中央处理器是计算机的运算核心（Core）和控制单元（Control Unit），主要包括：

1. 运算逻辑部件：一个或多个运算器
2. 寄存器部件：包括通用寄存器、控制与状态寄存器，以及高速缓冲存储器（Cache）
3. 控制部件：
    - 实现各部件间联系的数据、控制及状态的内部总线。
    - 负责对指令译码、发出为完成每条指令所要执行操作的控制信号、实现数据传输等功能的部件。



1.2.5 存储器的组织层次

> 组织架构

1. 寄存器
2. Cache（高速缓存）
3. 主存（DRAM - 动态随机存储器） ---------- 上述三类通电后工作，断电后信息清空
4. SSD（本地固态硬盘）
5. 本地外存储器（本地硬盘）
6. 远程外存储器（分布式文件系统、Web服务器）

从上往下，容量更大，速度更慢，价格更低。

1.2.6 外围设备及其控制

> 设备类型

输入设备、输出设备、存储设备、机机通信设备

> 设备控制方式

1. 轮询方式：CPU忙式控制，CPU执行内存数据交换
2. 中断方式：CPU启动外设，外设中断CPU，CPU执行内存数据交换
3. DMA方式：CPU启动DMA，DMA执行输入输出与内存数据交换，DMA中断CPU

1.3 计算机软件系统

1.3.1 计算机软件系统的组成

> 系统软件

1. 操作系统
    - 操作系统实施对各种软硬件资源的管理控制
2. 实用程序
    - 实用程序为方便用户所设，如文本编辑等
3. 语言处理程序
    - 语言处理程序把用汇编程序/高级语言编写的程序，翻译成可执行的机器语言程序
4. 数据库管理系统

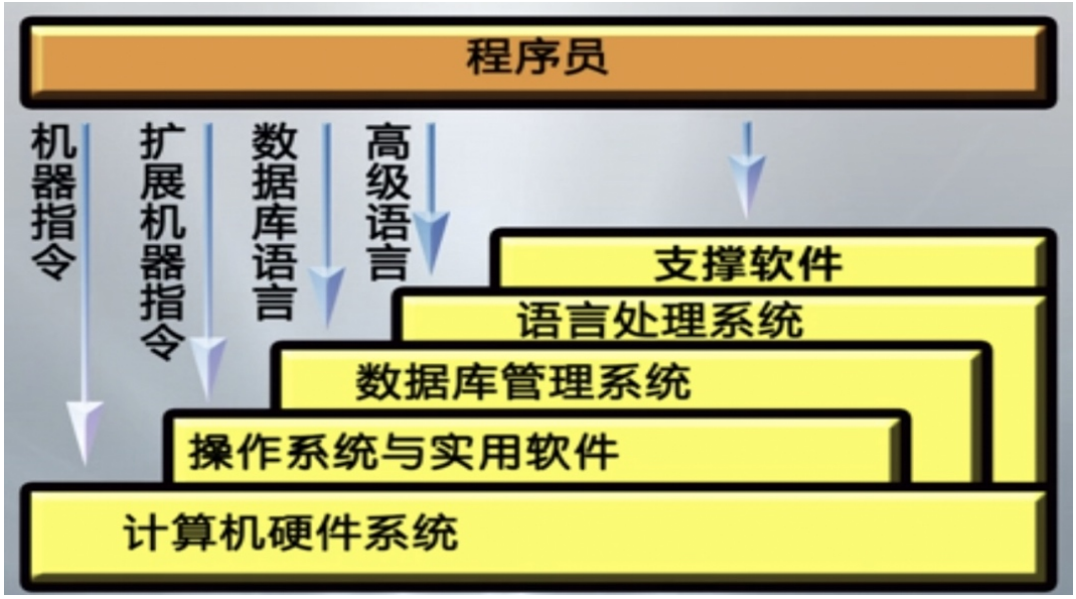
> 支撑软件

接口软件、工具软件、环境数据库，支持用户使用计算机的环境，提供开发工具。（也可认为是系统软件的一部分）

> 应用软件

即用户按其需要自行编写的专用程序。

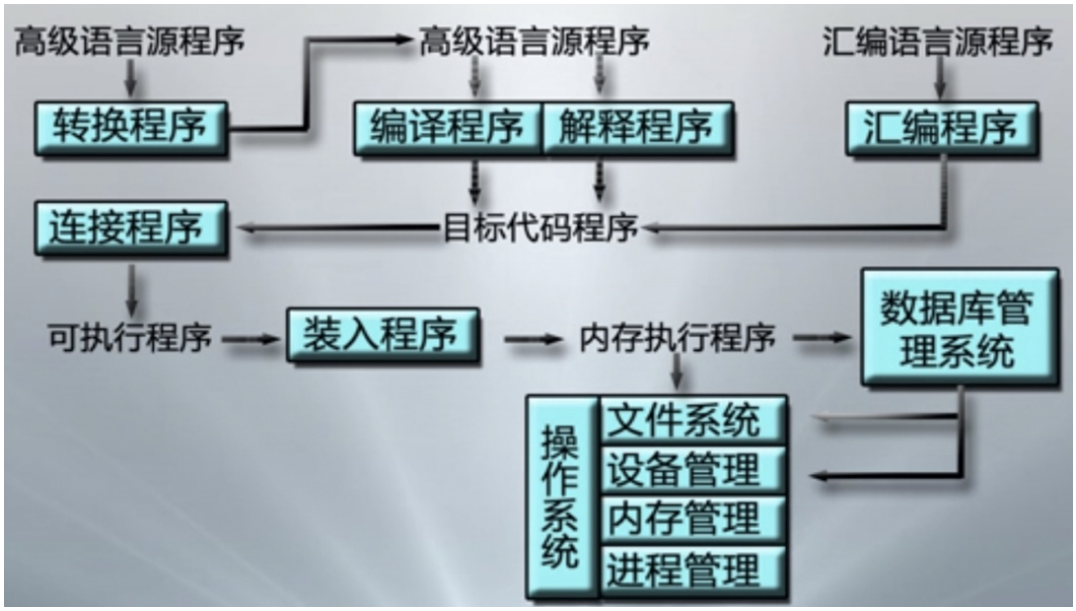
1.3.2 程序员的计算机系统视图



> 软件开发的不同层次

1. 计算机硬件系统：机器语言
2. 操作系统之资源管理：机器语言+广义指令（扩充了硬件资源管理）
3. 操作系统之文件系统：机器语言+系统调用（扩充了信息资源管理）
4. 数据库管理系统：++数据库语言（扩充了功能更强的信息资源管理）
5. 语言处理程序：面向问题的语言

1.3.3 计算机程序的执行过程



1.4 计算机操作技术的发展

1.4.1 计算机的手工操作

- 开关表示、按钮控制、亮灯显示

    - 开关置内存地址，按 “装入地址” 按钮
    - 开关置机器指令，按 “装入数据” 按钮（如此循环往复，将整个程序装入内存）
    - 开关置程序始地址，按 “运行” 按钮

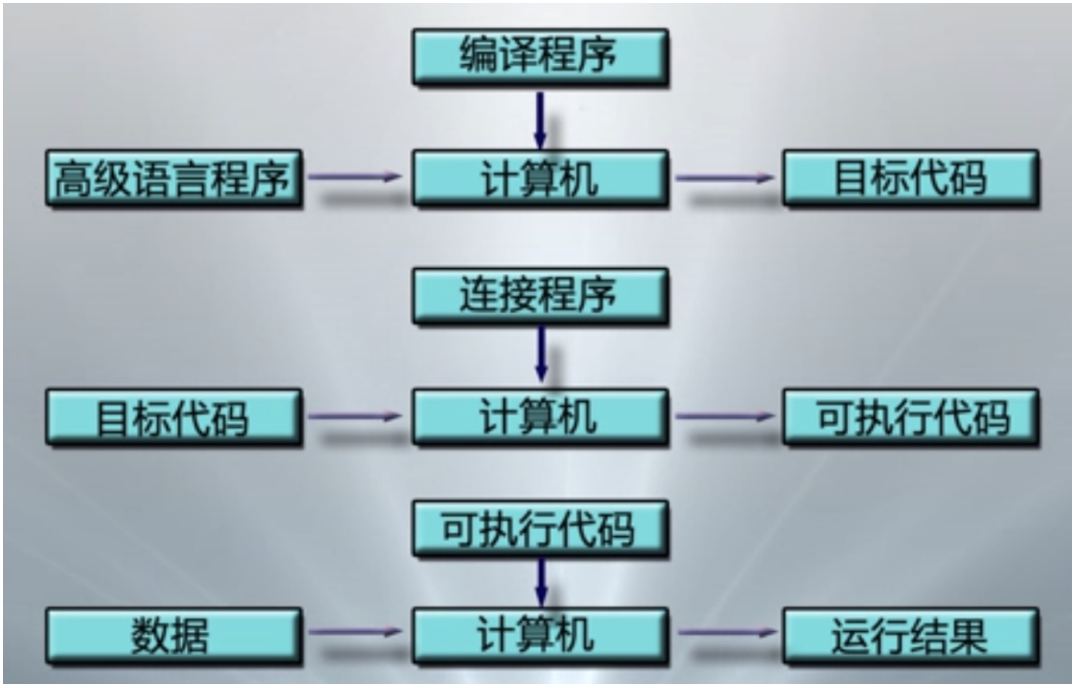
1.4.2 装入程序的引进

- 引入卡片和纸带描述程序指令与数据
- 引入装入程序（Loader）
    - 自动化执行程序装入，必要时进行地址转换
    - 通常存放在 ROM 中
- 机器程序语言 ⟶ 装入程序 ⟶ 内存储器

1.4.3 引入汇编语言后的计算机控制

将机器语言符号化，引入汇编语言。通过汇编程序，将汇编语言变成可执行代码。

1. 编译程序 —— 汇编代码变成可执行代码
2. 连接程序 —— 调用共有的函数库（高级语言之后）
3. 可执行代码 —— 计算机执行



1.4.4 简单批处理系统的操作控制

- 引入作业控制语言
    - 用户可以通过该语言编写作业说明书，描述对一次计算机求解（作业）的控制
- 操作员控制计算机成批输入作业，成批执行作业
- 这一方式明显缩短了手工操作的时间，提高了计算机系统利用率。
- 这一阶段，磁带的出现，使得卡片与纸带等机械输入方式得以进一步提高。

（ps：半自动化操作方式，还谈不上操作系统。）

> 缺点

- 虽然解决了手工操作与计算机的机械操作不匹配的矛盾，但是它并没有解决中央处理器与电子速度不匹配的矛盾。
- 而解决电子速度不匹配问题的关键在于让多个程序同时进入计算机系统，使得外围设备和CPU充分并行。

1.4.5 操作系统与自动化操作控制

- 1960年，磁盘设备出现
  - 计算机操作系统浓墨登场，实现了计算机系统的自动化控制。使得多个程序同时运行成为可能。

1.5 计算机操作系统

1.5.1 操作系统的概念

- 操作系统（Operating System），简称 OS
- OS是方便用户、管理和控制计算机软硬件资源的系统程序集合。

> 各个角度看待OS

1. 用户角度
   - OS管理计算机系统的各种资源，扩充硬件的功能，控制程序的执行
2. 人机交互角度
   - OS是用户与机器的接口，提供良好的人机界面，方便用高糊使用计算机
3. 系统结构
   - OS是一个大型软件系统，其功能复杂，体系庞大，采用层次式、模块化的程序结构

1.5.2 操作系统的组成

1. 进程调度子系统
2. 进程通信子系统
3. 内存管理子系统
4. 设备管理子系统
5. 文件管理子系统
6. 网络通信子系统
7. 作业控制子系统

1.5.3 操作系统的类型

> 从操作控制方式看

1. 多道批处理操作系统，采用脱机控制方式（程序员不接触计算机）
2. 分时操作系统，采用交互控制方式（最常见的操作控制方式）
3. 实时操作系统
    - 既支持分时的交互系统，同时的又有大量的进程待命用来处理突发的实时调度

> 从应用领域看

1. 服务器操作系统、并行操作系统
2. 网络操作系统、分布式操作系统
    - 网络与分布式主要区别在于透明和非透明
    - 分布式操作系统是透明的，用户不知道资源在哪，资源如何存储
3. 个人机操作系统、手机操作系统
4. 嵌入式操作系统、传感器操作系统

1.6 资源管理的角度

1.6.1 操作系统的资源

> 硬件资源

- 处理器、内存、外设

> 信息资源

- 数据、程序

1.6.2 管理计算机系统的软硬件资源

- **处理器资源**：哪个程序占有处理器运行?
- **内存资源**：程序/数据在内存中如何分布?
- **设备管理**：如何分配、去配和使用设备?
- **信息资源管理**：如何访问文件信息?
- **信号量资源**：如何管理进程之间的通信?

1.6.3 屏蔽资源使用的底层细节

> 驱动程序

- 最底层的、直接控制和监视各类硬件（或文件）资源的部分
- 职责是隐藏底层硬件的具体细节，并向其它部分提供一个抽象的、通用的接口

比如说，打印一段文字，既不需知道文件信息存储在硬盘上的细节，也不必知道具体打印机类型和控制细节。

1.6.4 资源的共享与分配方式

> 资源共享方式

- 独占使用方式（一段时间内只能被一个运行程序使用）
- 并发使用方式（涉及资源分配）

> 资源分配策略

- 静态分配方式
  - 这个运行程序进入内存前就拿到了所有资源，好处是不会死锁，坏处是效率低
- 动态分配方式
  - 谁要用就给谁分配，可能会发生死锁
- 资源抢夺方式
  - 定义一个规则，每个程序有一个优先级，我优先级比你高，我就把资源抢过来。
  - 优点：效率高且不会死锁
  - 缺点：我抢了你的资源，那你原来使用的资源如何回滚。

1.7 程序控制的角度

1.7.1 多道程序同时计算

> 问题：CPU速度与I/O速度不匹配的矛盾，非常突出

- 只有让多道程序同时进入内存争抢CPU运行，才可以使得CPU和外围设备充分并行，从而提高计算机系统的使用效率。
  - 程序A在读入时，程序B执行。程序A执行时，程序B输出。充分利用CPU。

> 多道程序设计及优点

- 多道程序设计：指让多个程序同时进入计算机的主存储器进行计算。
- 多道程序设计的特点
  - CPU与外部设备充分并行
  - 外部设备之间充分并行
  - 发挥CPU的使用效率
  - 提高单位时间的算题量
  - 单个程序的运行时间被延长。

1.7.2 多道程序系统的实现

> 进程（进入内存执行的程序建立管理实体）

- OS应能管理与控制进程程序的执行
- OS协调管理各类资源在进程间的使用
    - 处理器的管理和调度
    - 主存储器的管理和调度
    - 其他资源的管理和调度

> 多道程序系统的实现要点

- 如何使用资源：调用操作系统提供的服务例程（如何陷入操作系统）
- 如何复用CPU：调度程序（在CPU空闲时让其他程序运行）
- 如何使CPU与I/O设备充分并行：设备控制器与通道（专用的I/O处理器）
- 如何让正在运行的程序让出CPU：中断（中断正在执行的程序，引入OS处理）

1.8 操作系统控制计算机的角度

1.8.1 计算机系统操作方式

- OS规定了合理操作计算机的工作流程。
- OS的操作接口 —— 系统程序
    - OS提供给用户的功能级接口，为用户提供的解决操作计算机和计算共性问题的所有服务的集合。
    - OS的两类作业级接口
    - 脱机作业控制方式：作业控制语言
    - 联机作业控制方式：操作控制命令

> 脱机作业控制方式

1. （OS）提供作业说明语言
2. （用户）编写作业说明书，确定作业加工控制步骤，并与程序数据一并提交
3. （操作员）通过控制台输入作业
4. （OS）通过作业控制程序自动控制作业的执行

[例] 批处理是OS的作业控制方式，UNIX的shell程序，DOS的bat文件。

- 不需要用户干预，仅由系统根据作业说明书控制运行作业，又称为批处理作业。
- 直到输入缓存区满了，系统才来处理。

> 联机控制方式

1. （计算机）提供终端（键盘/显示器）
2. （用户）登录系统
3. （OS）提供命令解释程序
4. （用户）联机输入命令，直接控制作业步的执行

[例] 分时OS的交互控制方式。

- 系统立即处理或将数据存储起来之后处理。

> 假脱机技术（SPOOLing）

- 联机情况下实现的同时外围操作，也称假脱机输入输出操作，它是OS中一项将独占设备改为共享设备的技术。
- SPOOLing是低速输入输出设备与主机交换的一种技术，通常也成为 “假脱机真联机”，它的核心思想是以联机的方式得到脱机的效果。
- 独享打印机改造为多个用户共享的打印机
  - 系统对于用户的打印输出，但并不真正把打印机分配给该用户进程，而是先在输出井中申请一个空闲盘块区，并将要打印的数据送入其中；然后为用户申请并填写请求打印表，将该表挂到请求打印队列上。若打印机空闲，输出程序从请求打印队首取表，将要打印的数据从输出井传送到内存缓冲区，再进行打印，直到打印队列为空。

1.8.2 命令解释程序

> 命令解释程序概述

- 【核心功能】接受和执行一条用户提出的对作业的加工处理命令。
  - 当一个新的批作业被启动，或新的交互型用户登录进系统时，系统就自动地执行命令解释程序，负责读入控制卡或命令行，作出相应解释，并予以执行。
- 【命令解释程序的演变】
  - 会话语言：可编程的命令解释程序。（Linux 的 shell）
  - 图形化的命令控制方式。
  - 多通道交互的命令控制方式。

> 命令解释程序的处理过程

- OS启动命令解释程序，输出命令提示符，等待键盘中断/鼠标点击/多通道识别。
- 每当用户输入一条命令（暂存在命令缓冲区）并按回车换行时，申请中断。
- CPU进行响应，将控制权交给命令解释程序，读入缓冲区内容、分析命令、接受参数、执行代码。

> 前台命令与后台命令

- 前台命令

  - 命令之间必须严格按序执行，一个作业执行结束，下一个命令才能开始。

- 后台命令

  - 启动一个命令之后，不管这个命令有无结束，马上就可以接受下一条命令。

1.9 人机交互的角度

1.9.1 操作系统人机交互

> 人机交互设备组成

传统的终端设备、新型的模式识别设备。

> OS人机交互部分定性

控制有关设备运行、理解执行设备传来的命令。

> 人机交互功能定位

决定计算机系统友善性的重要因素，是当今OS研发热点。

1.9.2 人机交互的发展

> 初期发展

- 1960年，行命令控制方式（一个屏幕只有一行命令）
- 1970年，全屏幕控制方式（多行命令）
- 1960～1980年斯坦福研究所的发展计划
  - 两类成果
  - 【观念上】强调人机交互中心是人，而不是计算机效率。
  - 【设备上】鼠标、菜单与单窗口控制
  - 衡量操作系统好坏的标志已经不再是计算机系统的效率，而是这个计算机系统能否为更加广泛的人群所使用。

> 人机交互发展 —— WIMP界面（最常见）

- 缘起：70年代后期 Xerox 的原型机 Star
- 特征：窗口（Windows）、图标（Icons）、菜单（Menu）和指示装置（Pointing Devices）为基础的图形用户界面WIMP。
  - 窗口为多窗口。
  - 指示装置为单击、双击、左键、右键等。
- 发展
  - Apple最初采用并大力推动，1990年代开始广泛使用，ios操作系统是最能体现WIMP特性的OS。
- 不足
  - 不允许同时使用多个交互通道，从而产生人-机交互的不平衡。

> 人机交互发展 —— 多媒体计算机

- 缘起：1985年的MPC
  - 把音频视屏、图形图像和人机交互控制结合起来，进行综合处理的计算机系统。
- 构成：多媒体硬件平台、多媒体OS、图形用户接口、多媒体数据开发工具
  - 提供与时间有关的时变媒体界面，既控制信息呈现，也控制何时呈现/如何呈现。

> 人机交互发展 —— 虚拟现实系统

- 缘起：1980年代的虚拟现实新型用户界面。
- 特点
  - VR通过计算机模拟三维虚拟世界，根据观察点、观察点改变的导航和对周围对象的操作，来模拟临境的感觉。

1.10 程序接口的角度

1.10.1 系统调用

> 定位

系统调用是OS的程序接口。

> 定性

系统调用是OS实现的完成某种特定功能的过程；为所有运行程序提供访问OS的接口。

> 实现机制

- **陷入处理机制** —— （OS控制、实现系统调用的机制）
  - 陷入指令：访管指令或异常中断指令，OS为实现系统调用而引起处理器中断的指令。

- 每个**系统调用事先规定编号**，并在约定寄存器中规定了传递给内部处理程序的参数。

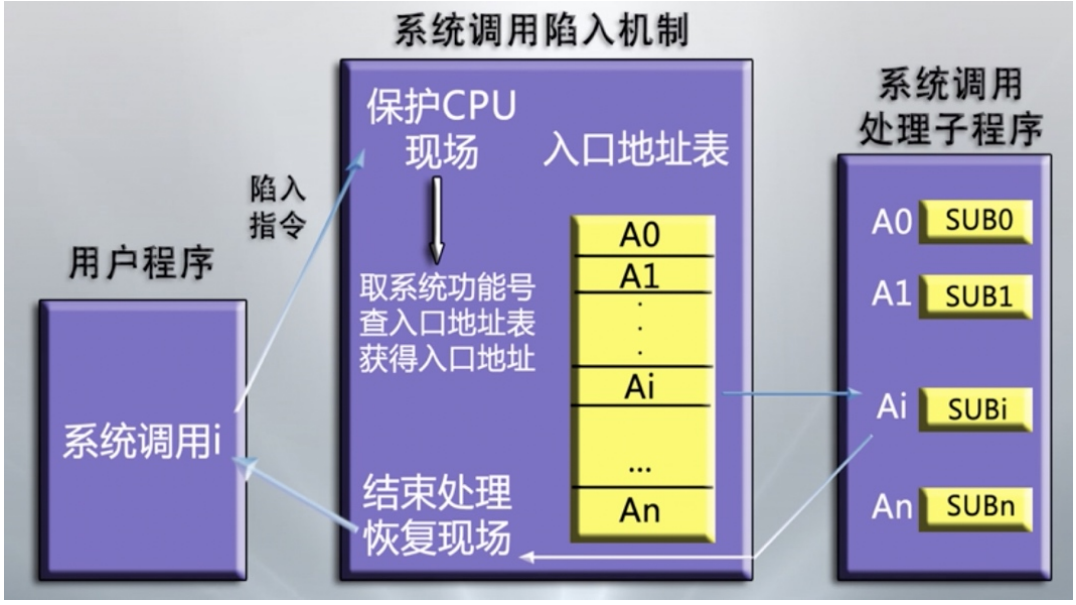
- **系统调用入口地址表**

  - 每个入口地址指向一个系统调用的处理程序，并包含系统调用自带参数的个数。

- **现场保护区**

  - 陷入处理机制需开辟现场保护区，以保存发生系统调用时的处理器现场。

> 实现过程



1.11 系统结构的角度

> OS 定位

OS最初触发了软件工程这个学科的诞生，并且是软件工程研究领域的最主要来源。（另一个来源是DB应用引发的需求）

1.11.1 操作系统软件的结构设计

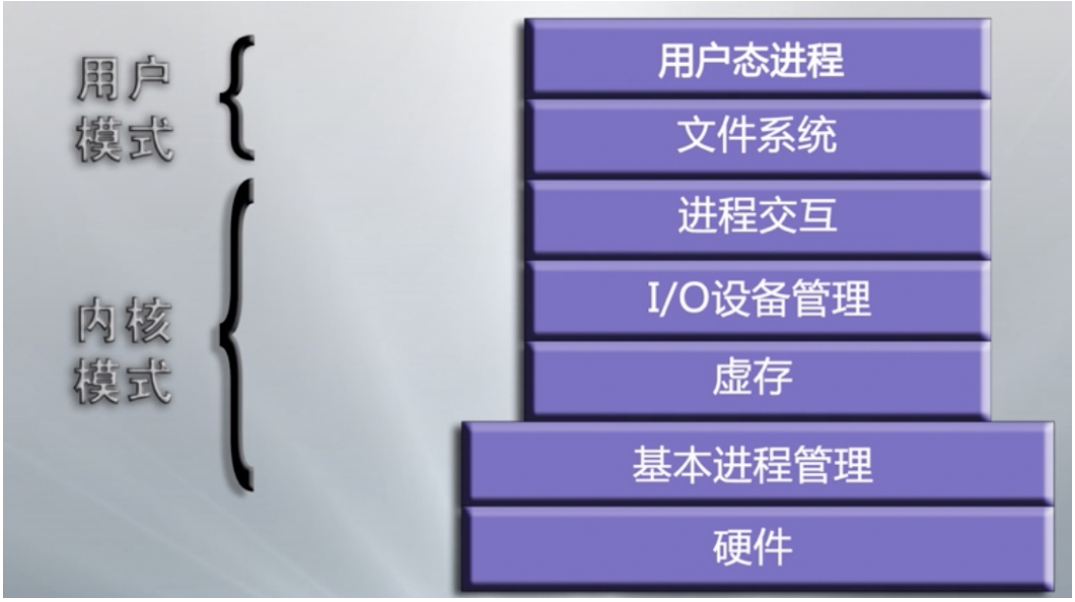
- OS构件
  - 内核、进程、线程、管程等。
- 设计概念
  - 模块化、层次化、虚拟化。

> 操作系统内核

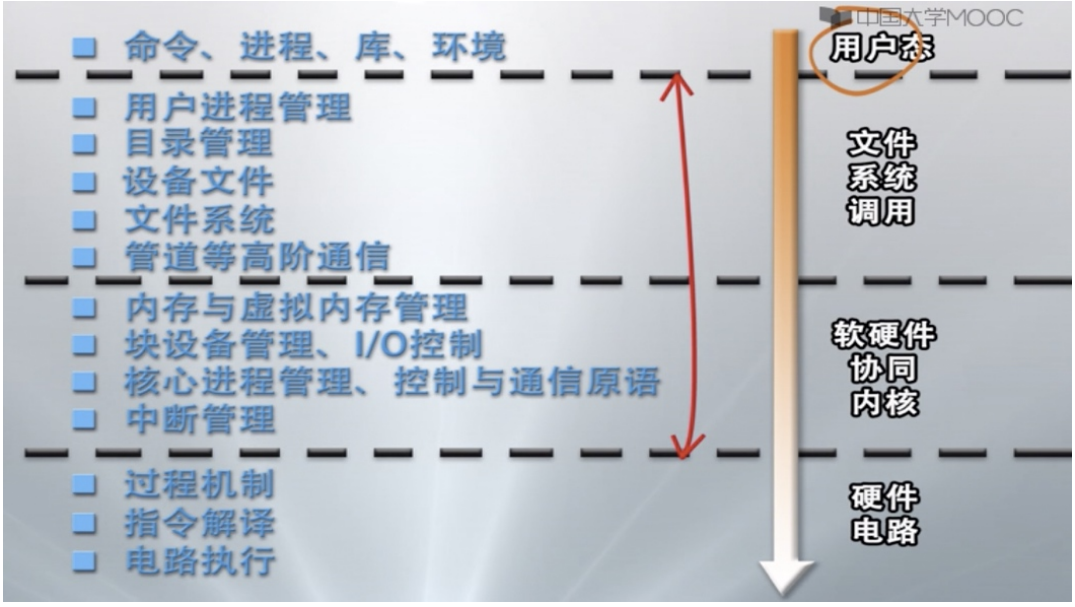
- 单内核【如今OS主流】
  - 内核中各部件杂染混居的形态，始于1960年，广泛使用。（Unix/Linux、Windows）
- 微内核
  - 1980年始，强调结构性部件与功能性部件的分离，大部分OS研究都集中于此。
- 混合内核
  - 微内核和单内核的折中，较多组件在核心态中运行。
- 外内核
  - 尽可能减少内核的软件抽象化和传统微内核的消息传递机制，使得开发者专注于硬件的抽象化；部分嵌入式系统使用。

1.11.2 OS层级结构

- 第一种层次结构（理想化）



- 第二种层次结构（中间两部分都属于OS部分）



二、处理器管理

2.1 处理器与寄存器

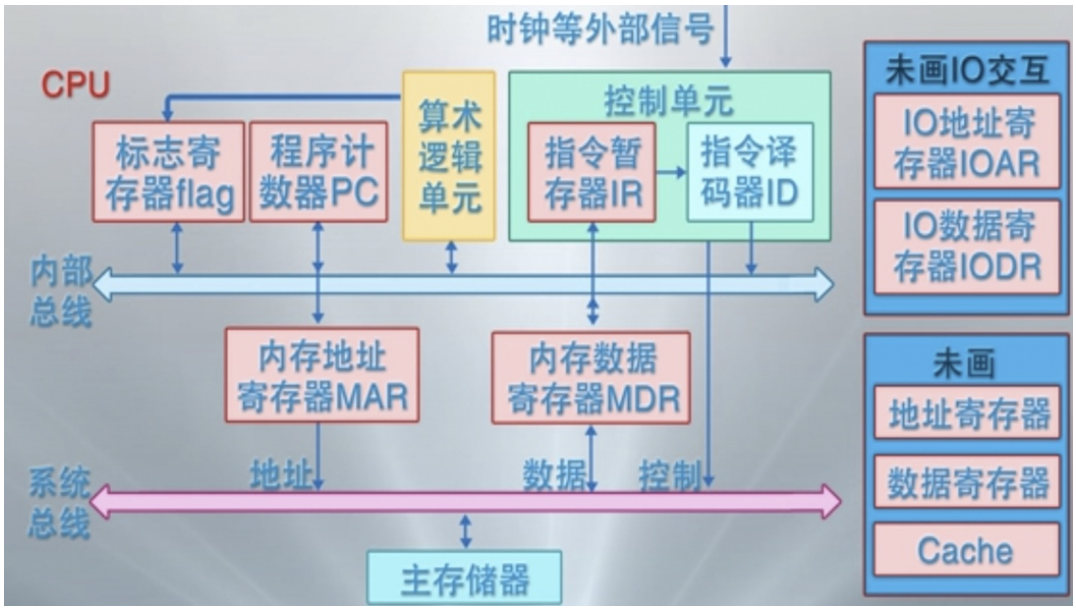
2.1.1 处理器

> CPU（运算单元+控制单元）

- 指令译码器（ID）：负责具体的解释指令的执行
- 指令暂存器（IR）：指令暂存在 IR 中

- 程序计数器（PC）：指向下一条要执行的指令的内存地址
- 标志寄存器（flag）：算术逻辑单元在执行结束后，会将结果汇总到标志寄存器（flag）中。
- MAR（内存地址寄存器）、MDR（内存数据寄存器）：实现对内存数据的访问。

上述所有部件均由内部总线连接通信。



2.1.2 寄存器

> 用户程序可见寄存器

- 可以使程序员减少访问主存储器的次数，提高指令执行的效率。
- 分类
  - **数据寄存器（通用寄存器）**：用于存储数据，如AX、BX、CX、DX
  - **地址寄存器**：索引（SI、DI）、栈指针（SP、BP）、段指针（CS、DS、SS、ES）

> 控制与状态寄存器

- 用于控制处理器的操作，主要**被具有特权的操作系统程序**使用，以控制程序的执行。
- 举例
  - 程序计数器（PC）、指令寄存器（IR）
- 还需记住指令执行的关键状态
  - 条件码 CC：CPU为指令操作结果设置的位，标志正/负/零/溢出等结果。
  - 标志位：中断位、中断允许位、中断屏蔽位、处理器模式位、内存保护位...

2.1.3 程序状态字（PSW）

> PSW

- 【**操作系统的概念**】，指记录**当前程序运行的动态信息**，通常包含：
  - 程序计数器、指令寄存器、条件码
  - 一系列标志位
- 【**计算机系统的寄存器**】
  - 通常设置一组控制与状态寄存器
  - 也可以专设一个 PSW 寄存器

2.2 指令与处理器模式

2.2.1 机器指令

> 定义

机器指令是计算机系统执行的基本命令，是中央处理器执行的基本单元。

> 定性

指令完成各种算术逻辑运算、数据传输、控制流跳转特征码。

> 指令构成

指令由一个或多个字节组成，包括操作码字段、一个或多个操作数地址字段、以及一些表征机器状态的状态字以及特征码。

2.2.2 指令执行过程

> 大致流程概述

根据 PC 取指，放入 IR，并对指令译码，发出控制命令，执行微操作，完成一条指令执行。

> 一种指令执行步骤

- 取指：根据 PC 从存储器或高速缓冲存储器中取指令到 IR
- 解码：解译 IR 中的指令来决定其执行行为
- 执行：连接 CPU，执行运算，产生结果并写回，设置CC；最后跳转指令跳转PC，其他指令递增PC

2.2.3 特权指令与非特权指令

> 背景

用户程序并非能够使用全部机器指令，那些与计算机核心资源相关的特殊指令会被保护。如启动 I/O 指令、置 PC 指令等。

> 具体分类

- 核心资源相关的指令只能被操作系统程序使用。
- 特权指令：只能被操作系统内核使用的指令。
- 非特权指令：能够被所有程序使用的指令。

2.2.4 处理器模式

> 背景

计算机通过设置处理器模式实现特权指令管理。

> 模式分类

- 计算机一般设置0、1、2、3等四种运行模式，分别对应：0操作系统内核、1系统调用、2共享库程序、3用户程序等保护级别。
- 0模式可以执行全部指令；3模式只能执行非特权指令；其他每种运行模式可以规定执行的指令子集。

> 真实现状

一般来说，现代操作系统只使用0和3两种模式，对应于内核模式和用户模式。

2.2.5 模式切换

> 分类

仅包括 “用户模式 -> 内核模式” 和 "内核模式 -> 用户模式" 的转换。

> 转换方式

中断、异常或系统异常等事件导致用户程序向 OS 内核切换，触发：**用户模式 -> 内核模式**。

1. 程序运行时发生并响应中断
2. 程序运行时发生异常
3. 程序请求操作系统服务

OS内核处理完成后，调用**中断返回指令**，触发：**内核模式 -> 用户模式**。

2.3 中断

2.3.1 中断的概念

> 定义

中断是指程序执行过程中，遇到急需处理的事件时，暂时中止CPU上现行程序的运行，转去执行相应的事件处理程序，待处理完成后再返回原程序被中断处或调度其他程序执行的过程。

> 定性

操作系统是“中断驱动”，即中断是激活操作系统的唯一方式。

中断分为广义与狭义，上述概念均指广义中断（广义中断 = 狭义中断 + 异常）。

> 中断、异常与系统异常

- 狭义中断

狭义的中断指来源于处理器之外的中断事件，即与当前运行指令无关的中断事件，如 I/O 中断、时钟中断、外部信号中断等。

- 异常

异常指当前运行指令引起的中断事件，如地址异常、算术异常、处理器硬件故障等。

- 系统异常（trap）

系统异常指执行陷入指令而触发系统调用引起的中断事件，如请求设备、请求 I/O、创建进程等。【系统异常可以被认为是异常中的一类】

2.4 中断源

> 处理器硬件故障中断事件（异常）

- 由处理器、内存储器、总线等硬件故障引起
- 处理原则
  - 保护现场、停止设备、停止CPU，向操作员报告，等待人工干预。

> 程序性中断事件（异常）

- 处理器执行机器指令引起
  - 算术异常（除数为0、操作数溢出）【简单处理】
  - 指令异常（非法指令、用户态使用特权指令、地址越界、非法存取）【终止进程】
  - 终止进程指令【终止进程】
  - 虚拟地址异常【调整内存后重新执行指令】

> 自愿性中断事件（系统调用）

- 处理器执行陷入指令请求OS服务引起（请求分配外设、请求I/O）
- 处理流程
  - 陷入OS、保护现场，根据功能号查入口地址，跳转具体处理程序。

> I/O中断事件（狭义中断）

- 来源于外围设备报告I/O状态的中断事件
  - I/O完成：调整进程状态，释放等待进程
  - I/O出错：等待人工干预
  - I/O异常：等待人工干预

> 外部中断事件

- 由外围设备发出的信号引起的中断事件
  - 时钟中断、间隔时钟中断：记时与时间片处理。
  - 设备报到与结束中断：调整设备表。
  - 键盘/鼠标信号中断：根据信号作出相应反应。
  - 关机/重启动中断：写回文件，停止设备与CPU。

2.5 中断系统

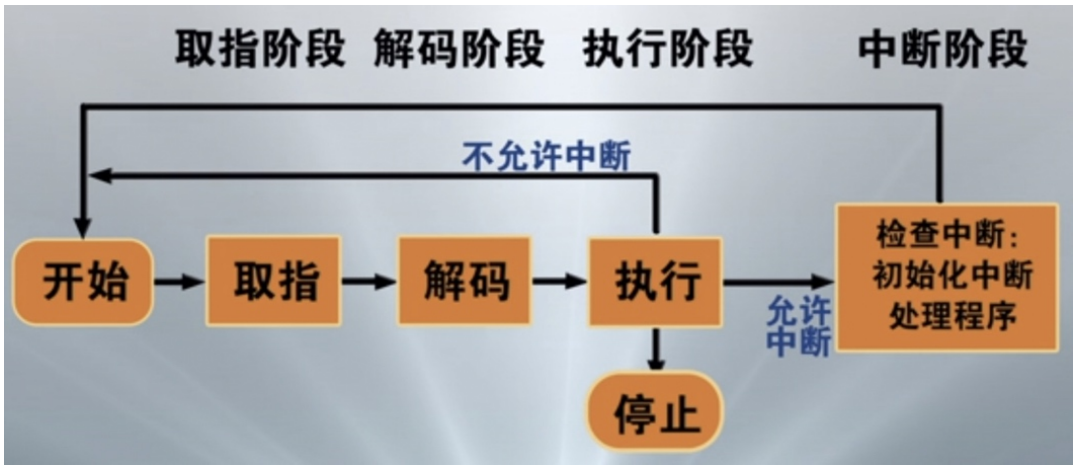
2.5.1 中断系统概述

> 定义

- 中断系统是计算机系统中响应和处理中断的系统，包括硬件子系统和软件子系统两部分。
  - 硬件子系统：负责中断响应
  - 软件子系统：负责中断处理

> 中断响应处理与指令执行周期

- 在指令执行周期最后增加一个微操作，以响应中断
- 注意：指令执行过程中不允许发生中断，指令具有原子性。



2.5.2 中断装置

> 定义

- 计算机系统中发现并响应中断 / 异常的硬件装置称为中断装置。

> 分类

- 由于中断源的多样性，存在多种中断装置，用于处理不同类型中断，一般因计算机而异，通常有：

1. **处理器外的中断**：由中断控制器发现和响应。
2. **处理器内的异常**：由指令的控制逻辑和实现线路发现和响应，响应机制称为陷阱。
3. **请求OS服务的系统异常**：处理器执行陷入指令时直接触发，响应机制称为系统陷阱。

2.5.3 中断控制器

> 定义

- CPU中的一个控制部件，包括中断控制逻辑线路和中断寄存器。
  - 中断寄存器：中断类型、中断来源

> 执行过程

1. **外部设备**向其发出中断请求 IRQ，在中断寄存器中设置已发生的中断。
2. 指令处理结束前，会检查中断寄存器，**若有不被屏蔽的中断产生，则改变处理器内操作的顺序**，引出操作系统中的中断处理程序。

> 特点

- **检查响应中断（由CPU完成）的过程**是一个**同步**的过程，而中断的请求由**外围设备**发生，是一个**异步**的过程。

2.5.4 陷阱与系统陷阱

> 定义

- 指令的逻辑实现线路的一部分。

> 执行过程

1. 执行指令出现异常后，会根据异常情况转向操作系统的异常处理程序。
2. 出现虚拟地址异常后，需要重新执行指令，往往越过陷阱独立设置页面异常处理程序。
3. 执行陷入指令后，越过陷阱处理，触发系统陷阱，激活系统调用处理程序。

2.5.5 中断响应过程

> 发现中断源，提出中断请求

1. 发现中断寄存器中记录的中断【发现】
2. 决定这些中断是否被屏蔽【是否屏蔽】

3. 当有多个要响应的中断源时，根据规定的优先级选择一个【优先级】

> 中断当前程序的执行

- **保存当前程序 PSW/PC 到核心栈（硬件保护）**

> 转向操作系统的中断处理程序

2.5.6 中断的处理

> **中断处理程序（软件保护）**

- 定性

OS 处理中断事件的控制程序，主要任务是处理中断事件和恢复正常操作。

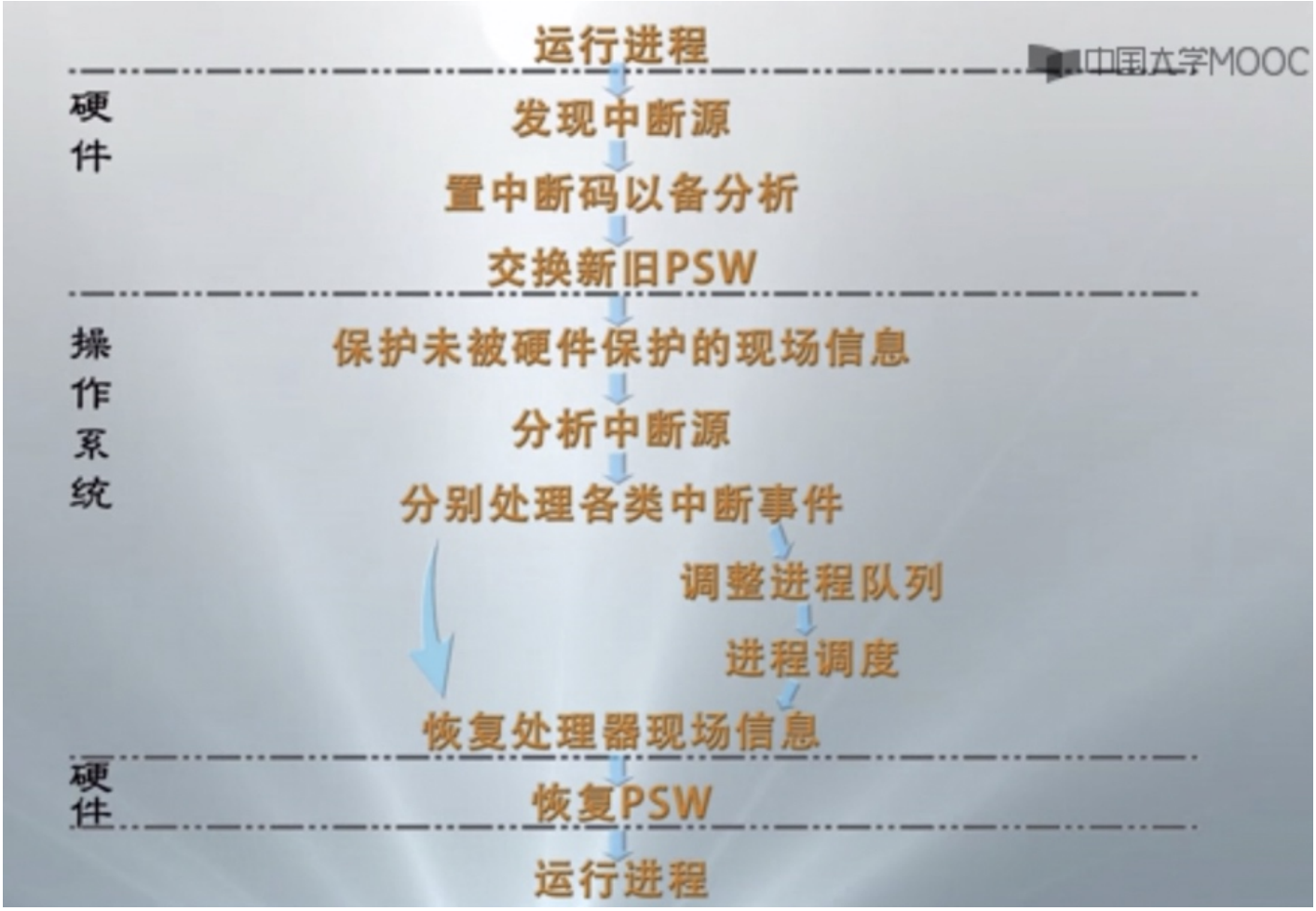
- 主要功能

1. 保护未被硬件保护的处理器状态（硬件只保护了 PSW/PC）
2. 分析被中断进程的 PSW 中断码字段，识别中断源
3. 分别处理发生的中断事件

> 恢复正常操作

1. 【**恢复原进程**】处理完毕后直接返回被中断进程（如时钟中断）
2. 【**恢复另一个进程**】中断当前进程运行，调整进程队列，启动进程调度，选择下一个执行的进程。（如输入输出进程）

2.5.7 中断系统处理流程



> 解析

1. **运行进程（硬件）**：在最后一个周期检测有无中断发生
2. **交换新旧PSW（硬件）**
   - 中断当前程序运行，保存当前程序的 PSW/PC 到核心栈
   - 并将OS内核系统的PC置到程序PC寄存器中，使得OS内核程序得以占用内核程序运行
3. **分析中断源（软件）**
   - 根据中断码字段，识别中断源
4. **分别处理各类中断事件（软件）**
   - 恢复自己的进程，或恢复其它进程
5. **恢复PSW（硬件）**
   - 弹出要恢复进程的PSW

2.6 多中断的响应与处理

2.6.1 中断屏蔽

> 特点（有选择的响应中断）

- 当计算机检测到中断时，中断装置通过中断屏蔽位决定是否响应已发生的中断。

2.6.2 中断优先级

> 特点（有优先度的响应中断）

- 当计算机同时检测到多个中断时，中断装置响应中断的顺序。

> 例子（不同OS有不同的中断优先级）

- 故障中断 > 自愿性中断 > 程序性中断 > （时钟中断）外部中断 > I/O中断 > 重启关机中断

2.6.3 中断的嵌套处理

> 特点

- 中断处理过程中，可以再响应其它中断
- **改变中断处理次序，先响应的可能后处理**

> 限制

- 中断响应处理有硬件要求，通常限制在 3 层之内

2.7 进程及其状态

2.7.1 进程的概念

> 进程的提出

- OS为了全方位地管理计算机系统中运行的程序，为正在运行的程序建立了一个管理实体 —— **进程**。

> 定义

1. 进程是 (一个具有一定独立功能的程序关于某个数据集合的) 一次运行活动。
2. 进程是OS进行资源分配和调度的一个独立单位。

> 进程分为 5 个实体部分

1. （OS管理运行程序的）数据结构 P
2. （运行程序的）内存代码 C

3. （运行程序的）内存数据 D
4. （运行程序的）通用寄存器信息 R
5. （OS控制程序执行的）程序状态字信息 PSW

> 进程的共享

1. **共享D**：不同程序在相同数据集上运行 —— 构成两个共享数据的交往进程

2. **共享C**

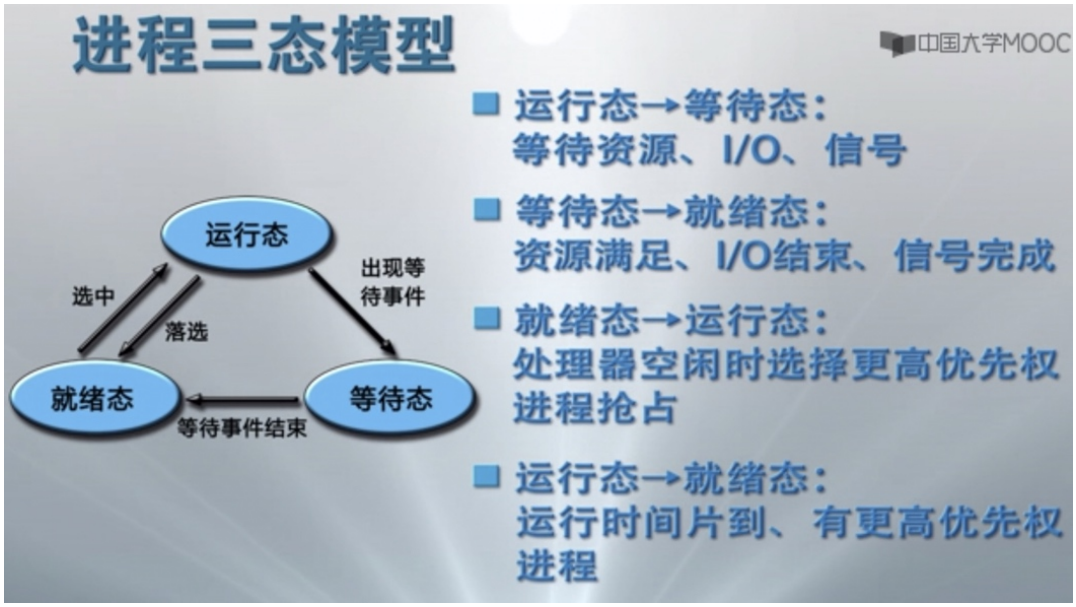
    - 相同代码在不同数据集上运行 —— 构成两个共享代码的无关进程
    - 共享的代码称为**可再入程序**，如编辑器。可再入程序必须是纯代码的，不能带任何局部数据区。

3. **注意**

    - 前述的程序与数据集均是内存级的。
    - 在不同时段中针对（同一个外存数据文件）运行（同一个外存程序文件），意味着完全不同的（P,C,D,R,PSW），因此两次运行构成两个不同的进程。

> 概念级的进程状态

1. **运行态**
    - 指进程占有处理器运行
2. **就绪态**
    - 指进程具备运行条件等待处理器运行
3. **等待态**
    - 指进程由于等待资源、输入输出、信号等而不具备运行条件



> 进程挂起的概念

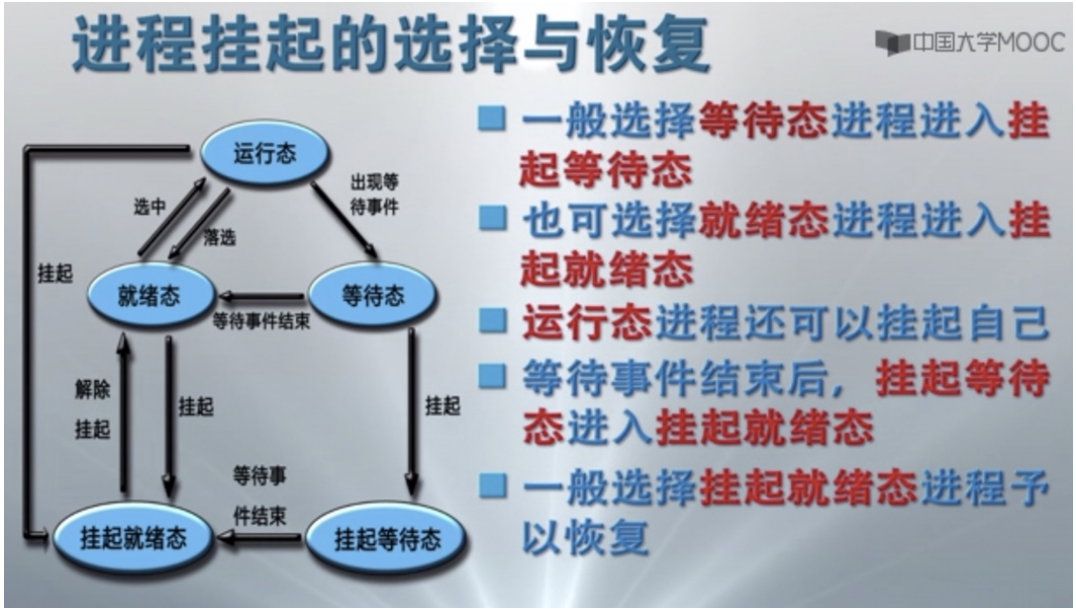
原因：OS无法预期进程的数目与资源需求，计算机系统在运行过程中可能出现资源不足的情况

运行资源不足的表现：性能低、死锁

解决办法（进程挂起）

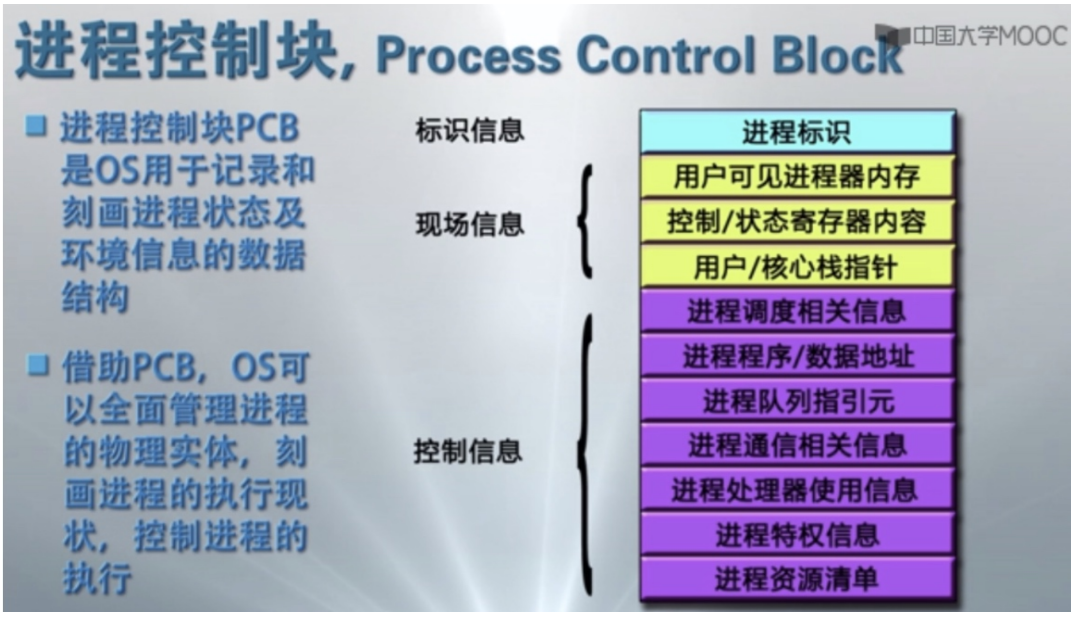
- 剥夺某些进程的内存及其他资源，调入OS管理的对换区，不参加进程调度，待适当时候再调入内存、恢复资源、参与运行

挂起态与等待态的区别：后者占有已申请到资源处于等待，前者没有任何资源。



2.8 进程的数据描述

2.8.1 进程控制块（PCB）



> 标识信息

功能：用于存放**唯一标识该进程**的信息

存放信息：系统分配的标识号（进程组标识号）、用户定义的进程名（进程组名）

> 现场信息

功能：用于存放该进程运行时的**处理器现场信息**

处理器现场信息：

- （用户可见寄存器内容）数据寄存器、地址寄存器
- （控制与状态寄存器内容）PC、IR、PSW
- （栈指针内容）核心栈与用户栈指针

> 控制信息

功能：用于存放与**管理、调度进程相关的信息**

管理、调度进程相关信息：

- （调度相关信息）状态、等待事件 / 原因、优先级
- （进程组成信息）代码 / 数据地址、外存映像地址
- （队列指引元）进程队列指针、父子兄弟进程指针
- （通信相关信息）消息队列、信号量、锁
- （进程特权信息）内存访问权限、处理器特权
- （处理器使用信息）占用的处理器、时间片、处理器使用时间 / 已执行总时间、记账信息
- （资源清单信息）正占有的资源、已使用的资源

2.8.2 进程映像（Process Image）

> 定义

某一时刻进程的内容及其执行状态集合。

进程映像是内存级的物理实体，又被称为进程的内存映像。

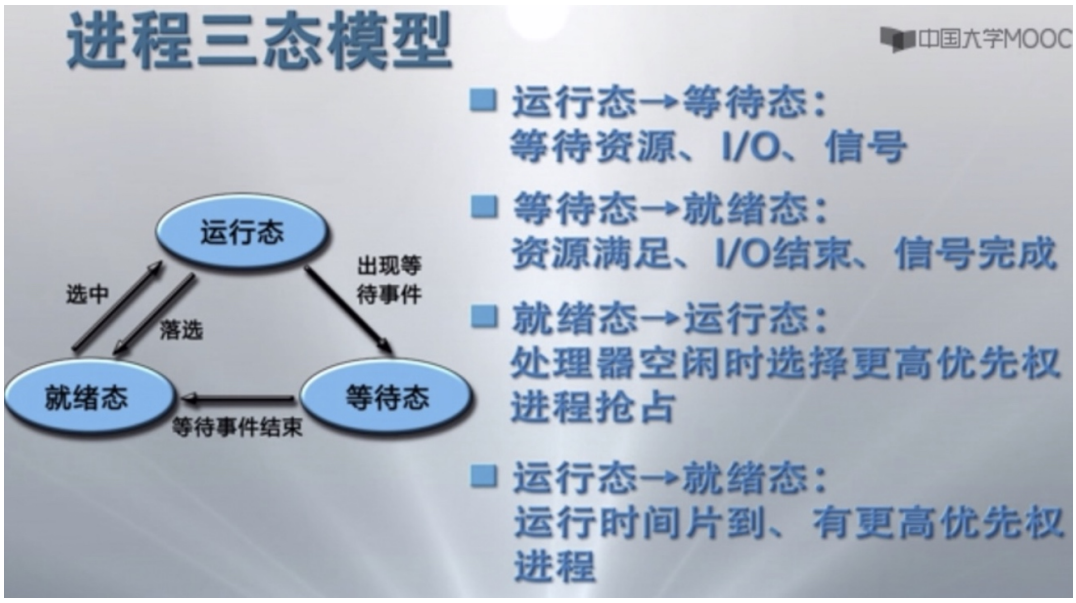
> 组成部分

1. 进程控制块
   保存进程的标识信息、状态信息和控制信息

2. 进程程序块
   进程执行的程序空间

3. 进程数据块
   进程处理的数据空间，包括数据、处理函数的用户栈和可修改的程序

4. 核心栈
   进程在内核模式下运行时使用的堆栈，中断或系统过程使用



2.8.3 进程上下文

> 出现原因

进程的执行需要环境支持，包括 CPU 现场和 Cache 中的执行信息

> 上下文分类

OS 中的进程物理实体和支持进程运行的环境共同构成进程上下文。

1. 用户级上下文
   用户程序块 / 用户数据区 / 用户栈 / 用户共享内存

2. 寄存器上下文
   PSW / 栈指针 / 通用寄存器

3. 系统级上下文
   PCB / 内存区表 / 核心栈

> 特点

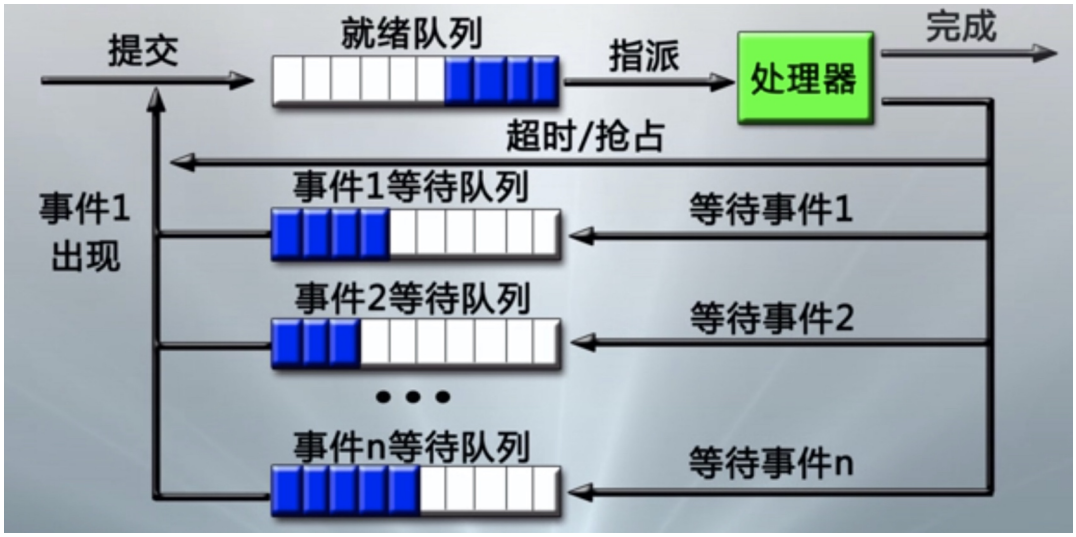
进程上下文刻画了进程的执行情况

2.9 进程的管理

2.9.1 概念级 OS 进程管理软件

1. 系统调用 / 中断 / 异常处理程序
2. 队列管理模块
3. 进程控制模块
4. 进程调度程序（独立进程居多）
5. 进程通信程序（多个程序包）

2.9.2 进程实现的队列模型



2.9.3 队列管理模块

1. 进程管理的核心模块
2. 多种进程队列，如就绪队列、等待队列、先进先出队列、优先队列
3. 队列中进程通过 PCB 中的队列指引元索引连接

2.9.4 进程的控制与管理

> 进程的创建

进程表加一项，申请PCB并初始化，生成标识，建立映像，分配资源，移入就绪队列

> 进程撤销

从队列中移除，归还资源，撤销标识，回收PCB，移除进程表项

> 进程阻塞

保存现场信息，修改PCB，移入等待队列，调度其他进程执行

> 进程唤醒（对应阻塞）

等待队列中移出，修改PCB，移入就绪队列（该进程优先级高于运行进程则触发抢占）

> 进程挂起

修改状态并移出入相关队列，收回内存等资源送至缓冲区

> 进程激活（对应挂起）

分配内存，修改状态并出入相关队列

2.9.5 原语与进程控制原语

目的：为了避免发生错误，对进程控制过程中涉及OS核心数据修改的部分采用原语实现。

> 原语定义

由若干条指令构成的完成某种特定功能的程序，执行上具有原子性，执行过程中不可以被中断、分割。即进入原语前把中断关掉，原语结束后把中断打开，因此原语必须是短小精悍的代码段。

> 分类

进程控制原语、进程通信原语

2.10 模式切换与进程切换

2.10.1 模式切换

> 概念

即用户模式和内核模式之间的相互切换。

> 分类

1. 用户模式 $\longrightarrow$ 内核模式：中断用户进程执行而触发（中断、异常、系统调用）
2. 内核模式 $\longrightarrow$ 用户模式：将控制权交还用户进程而触发（中断返回指令）

> 基本工作任务

正向切换（中断装置）：

- 处理器模式转为内核模式
- 保存当前进程的PC/PSW值到核心栈

- 转向中断/异常/系统调用处理程序

逆向切换（中断返回指令）：

- 从待运行进程核心栈中弹出PSW/PC值
- 处理器模式转为用户模式

2.10.2 进程切换

> 概念

进程切换指从正在运行的进程中收回处理器，让待运行进程来占有处理器运行。

进程切换实质上就是被中断运行进程与待运行进程的上下文切换。因此进程切换必须在OS内核模式下完成，需要模式切换。

> 具体处理过程

（保存旧的 ⟶ 找个新的 ⟶ 让新的上）

1. 【保存旧的】（中断/异常等触发）正向模式切换并压入PSW/PC
2. 保存被中断进程现场信息、处理具体中断
3. 把被中断进程的系统堆栈指针SP值保存到PCB、调整被中断进程的PCB信息、将PCB加入相关队列
4. 【找个新的】选择下一个占用CPU运行的进程
5. 【让新的上】修改被选中进程的PCB信息，如进程状态
6. 设置被选中进程的地址空间，恢复存储管理信息，恢复被选中进程的SP值以及现场信息
7. （中断返回指令触发）逆向模式转换并弹出

> 发生时机

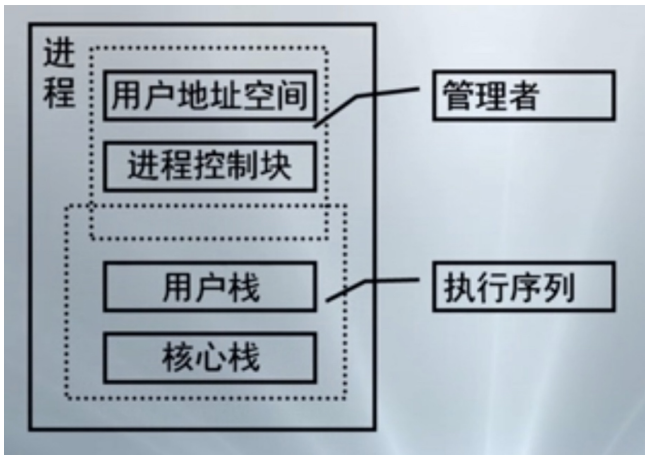
1. 进程进入等待态（阻塞式系统调用、虚拟地址异常）
2. 进程进入就绪态（时间片/IO中断、更高优先级进程插队）
3. 进程进入终止态

2.11 多线程技术概述

2.11.1 单线程结构进程

> 概念

由资源部分 + 一个执行序列组成，非常经典、传统的结构。



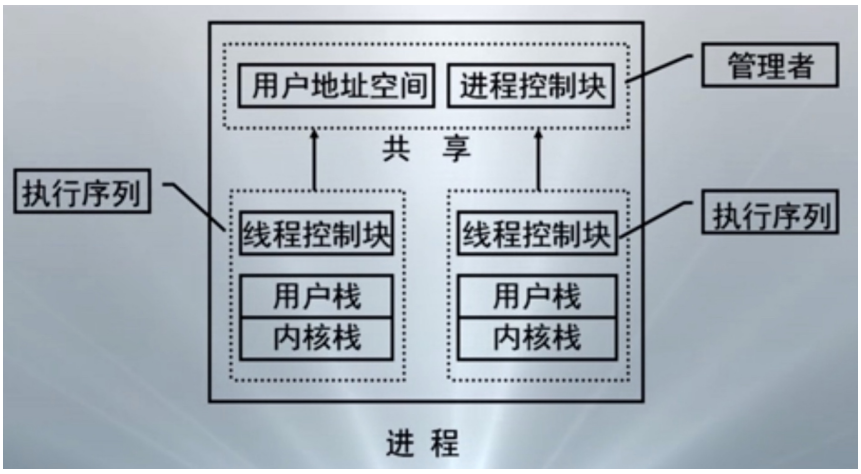
> 缺点

在并发程序设计中，存在进程切换开销大、进程通信开销大的问题，降低了并行计算的效率。

2.11.2 多线程结构进程

> 特点

出现了多个执行序列，提高了并发运算的效率。



> 多线程环境下进程的概念

在多线程环境中，进程是操作系统中进行保护和资源分配的独立单位，主要包含以下两个部分。

1. 用来容纳进程映像的虚拟地址空间
2. 对进程、文件和设备的存取保护机制

> 多线程环境下线程的概念

线程是进程的一条执行路径，是调度的基本单位，同一个进程中的所有线程共享进程获得的主

存空间和资源。主要包括以下内容：

1. 线程执行状态
2. 受保护的线程上下文，当线程不运行时，用于存储现场信息
3. 独立的程序指令计数器
4. 执行堆栈
5. 容纳局部变量的静态存储器

> 多线程环境下线程的状态与调整

线程状态：运行、就绪和睡眠，无挂起态

线程操作：孵化（创建）、封锁（进入等待态）、活化（结束等待态）、剥夺（离开CPU）、指派（分配CPU）、结束

OS感知线程环境下：

- 处理器调度对象是线程
- 进程没有三状态（或只有挂起态）

OS不感知线程环境下：

- 处理器调度对象仍是进程
- 由用户来调度线程

> 并发多线程程序设计的优点

快速线程切换、减少系统开销、线程通信易于实现、并行程度提高、节省内存空间

2.12 KLT与ULT

2.12.1 KLT

> 概念

KLT（Kernel-Level Threads），内核级线程机制。

线程管理的所有工作都由OS内核来做，OS直接调度KLT，OS提供了一个应用程序设计接口API，供开发者使用KLT。

> 特点

1. 进程中的一个线程被阻塞了，内核能调度同一进程的其它线程占有处理器运行。
2. 多处理器环境中，内核能同时调度同一进程中多个线程并行执行。
3. 内核自身也可用多线程技术实现，能提高操作系统的执行速度和效率。

4. 应用程序线程在用户态运行，线程调度和管理在内核实现，在同一进程中，**控制权从一个线程传送到另一个线程时需要模式切换。（开销较大）**

2.12.2 ULT

> 概念

ULT（User-Level Threads），用户级线程。

用户空间运行的线程库，提供多线程应用程序的开发和运行，任何应用程序均需通过线程库进行程序设计，再与线程库连接后运行。

线程管理的所有工作都由应用程序完成，内核没有意识到线程的存在。

> 特点

1. 所有线程管理数据均在进程用户空间中，线程切换不需要内核模式。（降低切换开销）
2. 允许进程定义调度算法，使调度更精准。
3. 可移植性强，能运行在任何OS上。
4. 【缺点1】不能利用多处理器的优点，OS调度进程，仅有一个ULT能执行。
5. 【缺点2】一个ULT的阻塞，将引起整个进程的阻塞。

2.12.3 Jacketing 技术

改造系统调用，将阻塞式系统调用改造成非阻塞式的。

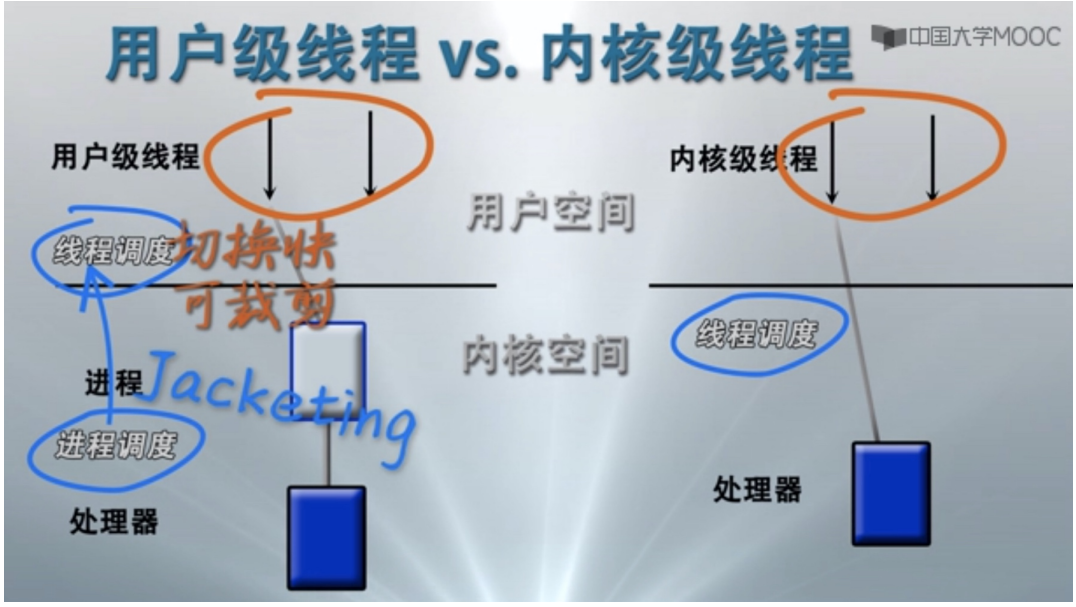
当线程陷入系统调用时，执行jacketing程序，使OS把控制权切换到用户线程调度程序，来选择下一个线程执行。

2.12.4 ULT vs. KLT

【相同点】进程是资源保护的单位，一个进程中可以有多个线程，且线程编程接口是内置的，且线程间可以共享资源。

【不同点】KLT线程调度由OS完成。ULT线程调度由用户定义，处理器仍然是进程调度，且利用jacketing技术进行进程调度和线程调度的切换。

【各自优势】ULT适用于解决逻辑并行性（用户定义线程调度），KLT适用于解决物理并行性（真正的多核并行线程）



2.13 多线程实现的混合策略

2.13.1 多线程实现的混合式策略

OS通过内核来支持内核级多线程，由OS程序库来支持用户级多线程。

将ULT和KLT进行多对多的绑定，使得逻辑并行性和物理并行性都得到较好的体现。

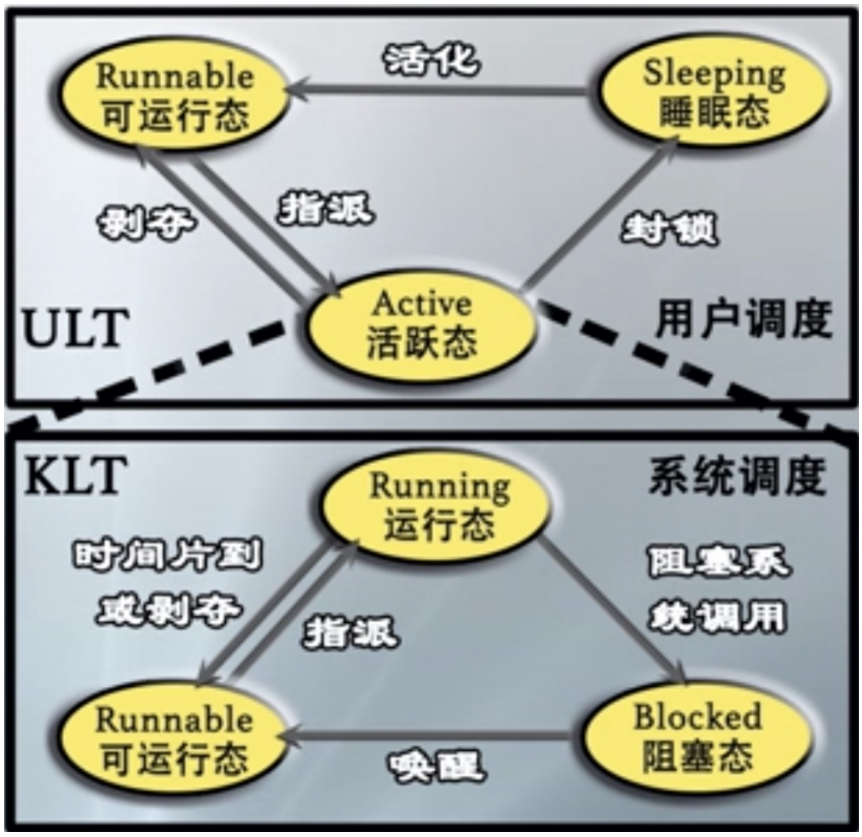
> 特点

1. 组合了KLT与ULT
2. 线程创建完全在用户空间中完成，线程的调度和同步也在应用程序中进行
3. 一个应用中的多个ULT被映射到一些KLT上
4. 程序员可以针对特定应用和机器调节内核级线程的数目，以达到整体最佳结果

2.13.2 线程混合式策略下的线程状态

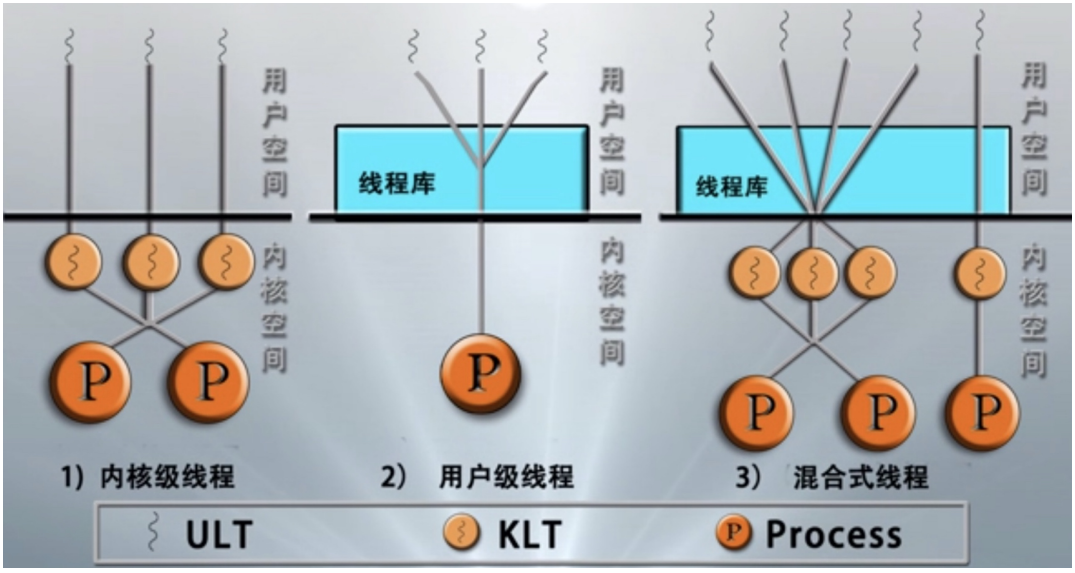
> 特点

1. KLT三态，系统调度负责
2. ULT三态，用户调度负责
3. **活跃态ULT代表绑定了一个KLT，而KLT有可能为三态中的任意一态**
4. 活跃态ULT运行时可激活用户调度
5. 非阻塞系统调用可使用Jacketing启动用户调度，调整活跃态ULT



> 总结

- **KLT**：OS直接调度KLT，物理并行性好
- **ULT**：OS调度进程，用户调度ULT，逻辑并行性好
- **混合式线程**：ULT可以直接分配到CPU上，也可以和KLT进行多对多匹配，实现了物理并行性与逻辑并行性共存



2.14 处理器调度

2.14.1 处理器调度的层次

> 高级调度（长程调度、作业调度）

决定一个程序能否加入到执行的进程池中。

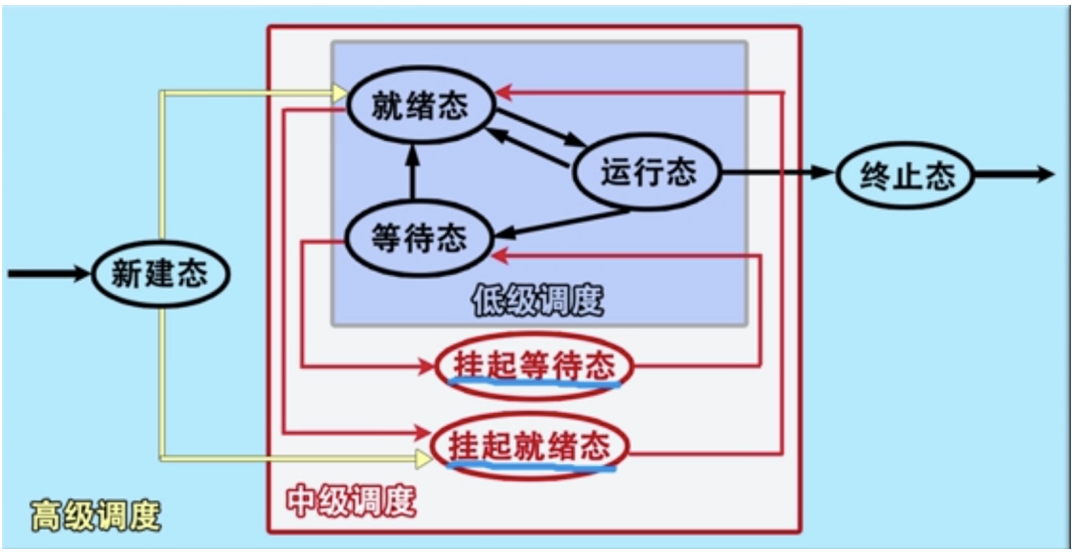
> 低级调度（短程调度、进程调度）

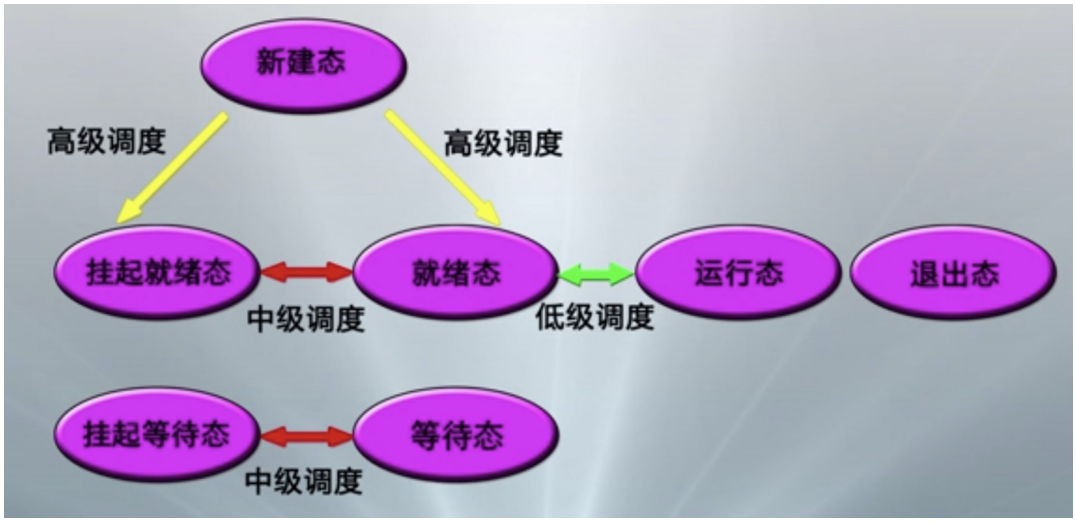
决定哪一个可用进程占用处理器执行。

> 中级调度（平衡负载调度）

当OS负荷过大时，可以将一些进程进入挂起状态，决定主存中的可用进程集合。

> 处理器调度层次与关键状态转换





从中可以发现三级调度在进程状态转换方面的主要功能，其中低级调度仅仅只是从就绪态到运行态的转换。而等待态 ⟶ 就绪态，运行态 ⟶ 等待态，运行态 ⟶ 终止态则不由调度决定，而由程序本身决定。

2.14.2 高级调度

> 分时OS中的功能

1. 决定是否接受一个终端用户的连接

2. 决定命令能否被系统接纳并构成进程
3. 决定新建态进程是否加入就绪进程队列

> 批处理OS中的功能

高级调度在批处理OS中又称为作业调度，功能是按照某种原则从后备作业队列中选取作业进入内存，并为作业做好运行前的准备工作和完成后的善后工作。

2.14.3 中级调度

> 目的

引入该调度的主要目的是为了提高内存利用率和作业吞吐率。

> 功能

中级调度决定哪些进程被允许驻留在主存中参与竞争处理器及其他资源，起到短期调整系统负荷的作用。

中级调度把一些进程换出主存，从而使之进入“挂起”状态，不参与进程调度，以平顺系统的负载。

2.14.4 低级调度

> 概念

低级调度即处理器调度，按照某种原则把处理器分配给就绪态进程或内核级线程。

> 进程调度程序

实现处理器调度的程序，是OS的最核心部分，是所有进程的父进程。其处理器调度策略优劣直接影响整个OS的性能。

> 主要功能

1. 记住进程或内核级线程的状态
2. 决定某个进程或内核级线程什么时候获得处理器，以及占用多长时间
3. 把处理器分配给进程或内核级线程
4. 收回处理器

2.15 处理器调度算法

2.15.1 调度算法原则

> 资源利用率（越高越好）

需要令CPU或其他资源的使用率尽可能高且能够并行工作。

> 响应时间（越短越好）

使交互式用户的响应时间尽可能小，或尽快处理实时任务。

响应时间 = 等待时间+运行时间（从提交第一个请求到产生第一个响应所用时间）

响应比 = （等待时间+运行时间）/ 运行时间。

> 周转时间（越短越好）

提交给系统开始到执行完成获得结果为止的这段时间间隔称周转时间。

周转时间 = 结束时间-提交时间

等待时间 = 周转时间-CPU使用时间

- **周转时间与响应时间的辨析**
  i. 在批处理系统中，产生第一次响应时，就是作业完成了。而在分时系统中，时间片结束后，就认为产生了第一个响应，而作业不一定完成了。
  ii. 因此二者仅在批处理系统中相等。
- 例题

考虑在单纯时间片轮转算法中，实现“优先级调度”，即优先级越高的进程一次分配时间片越多。有进程A、B、C、D、E依次几乎同时达到，其预计运行时间分别为10、6、2、4、8，其优先级数分别是3、5、2、1、4，一个优先级数对应一个时间片。对于前一个进程时间片有剩余的情况，操作系统会调度下一个进程运行。这种情况下总响应时间和总周转时间是_____。(时间片为1，忽略进程切换时间)

A.30、112

B.30、122

C.47、112

D.47、122

https://blog.csdn.net/weixin_42682806

A(10)	1	2	3	16	17	18	25	26	27	29	over
B(6)	4	5	6	7	8	19	over				
C(2)	9	10	over								
D(4)	11	20	28	30	over						
E(8)	12	13	14	15	21	22	23	24	over		

	响应时间	周转时间
A	3	29
B	8	19
C	10	10
D	11	30
E	15	24
总计	47	112

因此选 D。此题完美解释了分时系统中响应时间和周转时间的差别。

> 吞吐量（越多越好）

单位时间处理的进程数。

> 公平性

确保每个用户每个进程获得合理的CPU份额或其他资源份额。

2.15.2 优先数调度算法

> 基础概念

根据分配给进程的优先数决定运行进程的先后次序。

> 分类

1. 抢占式优先数调度算法：一旦高优先级出现，则必须占用CPU，不存在等待。
2. 非抢占式优先数调度算法：CPU空闲之后才去选择一个高优先级的。

> 优先数的确定准则

进程负担任务的紧迫程度、进程交互性、进程使用外设频率、进入系统的时间

> 与进入系统时间相关的优先数

1. 计算时间短优先
2. 剩余计算时间短优先
3. 响应比高者优先（$响应比 = \frac{等待时间 + 执行时间}{执行时间}$）
4. 先来先服务（先进队则先被选择）
    - 多用于高级调度；低级调度中则并不常用。

2.15.3 时间片轮转调度算法

> 算法概念

根据各个进程进入就绪队列的时间，先后轮流占有CPU一个时间片。时间片到了则采用时间片中断中断进程。

> 时间片长短确定

选择长短合适的时间片，过长则退化为先来先服务算法，过短则调度开销大。

> 类别

单时间片、多时间片、动态时间片

2.15.4 分级调度算法

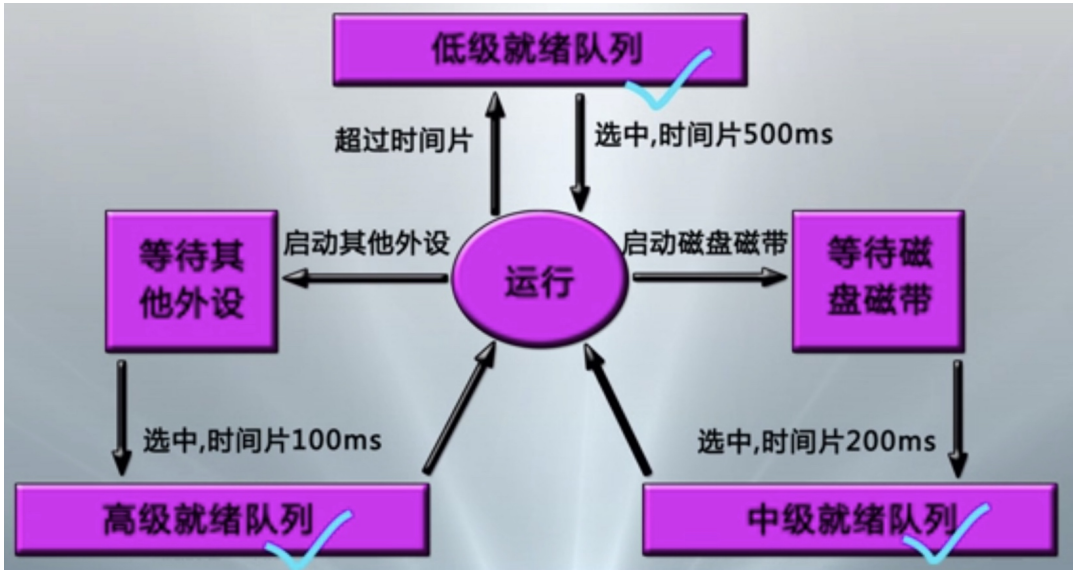
> 算法概念

分级调度算法又称为多队列策略，反馈循环队列。

> 基本思想

1. 建立多个不同优先级的就绪进程队列
2. 多个就绪进程队列间按照优先数调度
3. 高优先级就绪进程，分配的时间片短
4. 单个就绪进程队列中进程的优先数和时间片相同（先进先出）

> 举例



三个优先级的队列，外设进入高队列，磁盘进入中队列，如果时间片超时则进入低队列。

> 分级算法的分级原则

- 一般分级原则

    - 外设访问、交互性、时间紧迫程度、系统效率、用户立场...

- 现代操作系统的实现模型**（结合时间片+优先数+分级队列）**
    - 多个高优先级的实时进程队列，如：硬实时、网络、软实时
    - 多个分时任务的进程队列，根据基准优先数和执行行为调整
    - 队列数可能多达 32～128 个

2.15.5 彩票调度算法

> 基本思想

为进程发放针对系统各种资源（如CPU时间）的彩票；当调度程序需要做出决策时，随机选择一张彩票，持有该彩票的进程将获得系统资源。

OS发展到后期时出现的一个算法，非常有意义。

> 功能

比如一个服务器、十个客户机的程序。一开始给每个客户机10张彩票，不给服务器彩票。而当客户机请求服务器时，客户机则将其彩票交换到服务器上去。

当十个客户机都将彩票交换到服务器时，服务器直接获得CPU资源。因此彩票调度算法非常有意义，是根据具体运行过程中的需求进行分配。

2.16 内核调度算法

2.16.1 内核调度程序

> 应用程序类型

1. IO 消耗型：网络IO、磁盘IO
2. CPU 消耗型

> 优先队列

```
struct prio_array{
    unsigned int nr_active;  //等待执行的进程数
    unsigned long bitmap[BITMAP_SIZE]; //用位的方式，表示每个优先级队列的状态
    struct list_head queue[MAX_PRIO]; //对应于bitmap，存储所有等待运行的进程
}
```

`unsigned long bitmap[BITMAP_SIZE] 解析：`

- 一个 `unsigned long` 在内核中有 32 位，其用位的方式，表示某个优先级上有没有待处

理的队列，是实现快速找到最高待处理优先进程的关键。如果我定义了四种优先级，我只需要四位就能表示某个优先级上有没有进程要运行，例如优先级是 2 和 3 上有进程，那么就应该是 0110。

- `#define BITMAP_SIZE ((((MAX_PRIO+1+7)/8)+sizeof(long)-1)/sizeof(long))`，因此一共 140 种优先级，BITMAP_SIZE 为 4，$32 * 5 = 160 > 140$，包含了 140 位优先级。

> 如何快速找到优先级最高的队列

```
idx = sched_find_first_bit(array->bitmap);

// 返回值为最高优先级所在队列的序号，queue = array->queue + idx; 即可取到要处理的进程队列
static inline int sched_find_first_bit(unsigned long *b)
{
    if (unlikely(b[0]))
        return __ffs(b[0]);
    if (unlikely(b[1]))
        return __ffs(b[1]) + 32;
    if (unlikely(b[2]))
        return __ffs(b[2]) + 64;
    if (b[3])
        return __ffs(b[3]) + 96;
    return __ffs(b[4]) + 128;
}

static inline int __ffs(int x)
{
    int r = 0;

    if (!x)
        return 0;
    if (!(x & 0xffff)) {
        x >>= 16;
        r += 16;
    }
    if (!(x & 0xff)) {
        x >>= 8;
        r += 8;
    }
    if (!(x & 0xf)) {
        x >>= 4;
        r += 4;
    }
    if (!(x & 3)) {
        x >>= 2;
        r += 2;
```

```
    }
    if (!(x & 1)) {
        x >>= 1;
        r += 1;
    }
    return r;
}
```

> CPU 上的 runqueue

```
struct runqueue {
    spinlock_t lock;   //自旋锁，nginx里解决惊群现象时也是用这个。
    /*
     * nr_running and cpu_load should be in the same cacheline because
     * remote CPUs use both these fields when doing load calculation.
     */
    unsigned long nr_running;
#ifdef CONFIG_SMP
    unsigned long cpu_load;
#endif
    unsigned long long nr_switches;


    /*
     * This is part of a global counter where only the total sum
     * over all CPUs matters. A task can increase this counter on
     * one CPU and if it got migrated afterwards it may decrease
     * it on another CPU. Always updated under the runqueue lock:
     */
    unsigned long nr_uninterruptible;


    unsigned long expired_timestamp;
    unsigned long long timestamp_last_tick;
    task_t *curr, *idle;
    struct mm_struct *prev_mm;
    prio_array_t *active, *expired, arrays[2]; //优先级队列，runqueue中一共有两个
    int best_expired_prio;
    atomic_t nr_iowait;
    ... ...
};
```

自旋锁与普通锁的区别：

- 【普通锁】使用普通锁时，你去试图拿一把锁，结果发现已经被别人拿走了，你就在那睡觉，等别人锁用完了叫你起来。所以如果有一个人拿住锁了，一百个人都在门前睡觉等。当之前的人用完锁回来后，会叫醒所有100个等锁的人，然后这些人开始互相抢，抢到的

人拿锁进去，其他的人继续等。

- 【自旋锁】自旋锁不同，当他去拿锁发现锁被别人拿走了，他在那不睡觉的等，稍打个盹就看看自己主动看看锁有没有还回来，即反复检查锁变量是否可用。自旋锁效率高，但自旋锁仅适用于锁使用者保持锁时间比较短的情况下，因为反复检查锁的过程非常降低效率。（忙式等待）

> 调度函数

runqueue中的两个优先队列：

- $active$ 是还有时间片的进程队列，而 $expired$ 是时间片耗尽必须重新分配时间片的进程队列。

```
array = rq->active;
if (unlikely(!array->nr_active)) {
    /*
     * Switch the active and expired arrays.
     */
    schedstat_inc(rq, sched_switch);
    rq->active = rq->expired;
    rq->expired = array;
    array = rq->active;
    rq->expired_timestamp = 0;
    rq->best_expired_prio = MAX_PRIO;
} else
    schedstat_inc(rq, sched_noswitch);
```

当所有运行进程的时间片都用完时，就把 $active$ 和 $expired$ 队列互换指针，而时间片耗尽的进程在出 $active$ 队列入 $expired$ 队列时，已经单独的重新分配好新时间片了。

> $schedule$ 调度函数

```
asmlinkage void __sched schedule(void)
{
    long *switch_count;
    task_t *prev, *next;
    runqueue_t *rq;
    prio_array_t *array;
    struct list_head *queue;
    unsigned long long now;
    unsigned long run_time;
    int cpu, idx;


    /*
```

```
     * Test if we are atomic.  Since do_exit() needs to call into
     * schedule() atomically, we ignore that path for now.
     * Otherwise, whine if we are scheduling when we should not be.
     */
    if (likely(!(current->exit_state & (EXIT_DEAD | EXIT_ZOMBIE)))) {先看看当前运行进程的状态
        if (unlikely(in_atomic())) {
            printk(KERN_ERR "scheduling while atomic: "
                "%s/0x%08x/%d\n",
                current->comm, preempt_count(), current->pid);
            dump_stack();
        }
    }
    profile_hit(SCHED_PROFILING, __builtin_return_address(0));


need_resched:
    preempt_disable();
    prev = current;
    release_kernel_lock(prev);
need_resched_nonpreemptible:
    rq = this_rq(); //这行找到这个CPU对应的runqueue，再次强调，每个CPU有一个自己的runqueue


    /*
     * The idle thread is not allowed to schedule!
     * Remove this check after it has been exercised a bit.
     */
    if (unlikely(current == rq->idle) && current->state != TASK_RUNNING) {
        printk(KERN_ERR "bad: scheduling from the idle thread!\n");
        dump_stack();
    }


    schedstat_inc(rq, sched_cnt);
    now = sched_clock();
    if (likely(now - prev->timestamp < NS_MAX_SLEEP_AVG))
        run_time = now - prev->timestamp;
    else
        run_time = NS_MAX_SLEEP_AVG;


    /*
     * Tasks with interactive credits get charged less run_time
     * at high sleep_avg to delay them losing their interactive
     * status
     */
```

```
    if (HIGH_CREDIT(prev))
        run_time /= (CURRENT_BONUS(prev) ? : 1);


    spin_lock_irq(&rq->lock);


    if (unlikely(current->flags & PF_DEAD))
        current->state = EXIT_DEAD;
    /*
     * if entering off of a kernel preemption go straight
     * to picking the next task.
     */
    switch_count = &prev->nivcsw;
    if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
        switch_count = &prev->nvcsw;
        if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
                unlikely(signal_pending(prev))))
            prev->state = TASK_RUNNING;
        else {
            if (prev->state == TASK_UNINTERRUPTIBLE)
                rq->nr_uninterruptible++;
            deactivate_task(prev, rq);
        }
    }


    cpu = smp_processor_id();
    if (unlikely(!rq->nr_running)) {
go_idle:
        idle_balance(cpu, rq);
        if (!rq->nr_running) {
            next = rq->idle;
            rq->expired_timestamp = 0;
            wake_sleeping_dependent(cpu, rq);
            /*
             * wake_sleeping_dependent() might have released
             * the runqueue, so break out if we got new
             * tasks meanwhile:
             */
            if (!rq->nr_running)
                goto switch_tasks;
        }
    } else {
        if (dependent_sleeper(cpu, rq)) {
            next = rq->idle;
            goto switch_tasks;
        }
```

```
        /*
         * dependent_sleeper() releases and reacquires the runqueue
         * lock, hence go into the idle loop if the rq went
         * empty meanwhile:
         */
        if (unlikely(!rq->nr_running))
            goto go_idle;
    }


    array = rq->active;
    if (unlikely(!array->nr_active)) { //上面说过的，需要重新计算时间片时，就用已经计
算好的expired队列了
        /*
         * Switch the active and expired arrays.
         */
        schedstat_inc(rq, sched_switch);
        rq->active = rq->expired;
        rq->expired = array;
        array = rq->active;
        rq->expired_timestamp = 0;
        rq->best_expired_prio = MAX_PRIO;
    } else
        schedstat_inc(rq, sched_noswitch);


    idx = sched_find_first_bit(array->bitmap); //找到优先级最高的队列
    queue = array->queue + idx;
    next = list_entry(queue->next, task_t, run_list);


    if (!rt_task(next) && next->activated > 0) {
        unsigned long long delta = now - next->timestamp;


        if (next->activated == 1)
            delta = delta * (ON_RUNQUEUE_WEIGHT * 128 / 100) / 128;


        array = next->array;
        dequeue_task(next, array);
        recalc_task_prio(next, next->timestamp + delta);
        enqueue_task(next, array);
    }
    next->activated = 0;
switch_tasks:
    if (next == rq->idle)
        schedstat_inc(rq, sched_goidle);
```

```
    prefetch(next);
    clear_tsk_need_resched(prev);
    rcu_qsctr_inc(task_cpu(prev));


    prev->sleep_avg -= run_time;
    if ((long)prev->sleep_avg <= 0) {
        prev->sleep_avg = 0;
        if (!(HIGH_CREDIT(prev) || LOW_CREDIT(prev)))
            prev->interactive_credit--;
    }
    prev->timestamp = prev->last_ran = now;


    sched_info_switch(prev, next);
    if (likely(prev != next)) { //表面现在正在执行的进程，不是选出来的优先级最高的进程
        next->timestamp = now;
        rq->nr_switches++;
        rq->curr = next;
        ++*switch_count;


        prepare_arch_switch(rq, next);
        prev = context_switch(rq, prev, next); //所以需要完成进程上下文切换，把之前的
进程信息CACHE住
        barrier();


        finish_task_switch(prev);
    } else
        spin_unlock_irq(&rq->lock);


    prev = current;
    if (unlikely(reacquire_kernel_lock(prev) < 0))
        goto need_resched_nonpreemptible;
    preempt_enable_no_resched();
    if (unlikely(test_thread_flag(TIF_NEED_RESCHED)))
        goto need_resched;
}
```

2.16.2 负载均衡

> 负载均衡的两个实现方式

- 【方式 1】当一个处理器上的任务全部结束之后，每 $1ms$，它会通过系统调用到最忙的处理器上获得任意一个任务执行，调用 $load_balance$ 函数。

- 【方式 1 缺点】方式 1 不适用于运行队列一直不为空的情况。例如，$cpu0$ 上一直有 10 个可运行进程，$cpu1$ 上一直有 1 个可运行进程，显然 $cpu0$ 上的进程们得到了不公平的对待，它们拿到 cpu 的时间要小得多，第 1 种情况下的 $load_balance$ 函数也一直不会调用。所以，实际上，每经过一个时钟节拍，内核会调用 $scheduler_tick$ 函数，这个函数会做许多事，例如减少当前正在执行的进程的时间片，在函数结尾处则会调用 $rebalance_tick$ 函数，该函数决定以什么样的频率执行负载均衡。

- 【方式 2】而在全部处理器都有任务在执行时，则由时钟每 $200ms$ 通过系统调用去检查，若发现在 $Linux$ 标准下不均衡，则会发生处理器之间的就绪任务迁移。

- 【方式 2 具体实现】当 $idle$ 标志位是 $SCHED_IDLE$ 时，表示当前 CPU 处理器空闲，就会以很高的频率来调用 $load_balance$（1、2 个时钟节拍），反之表示当前 CPU 并不空闲，会以很低的频率调用 $load_balance$（$10 \sim 100ms$）。

> $rebalance_tick$ 函数

```
static void rebalance_tick(int this_cpu, runqueue_t *this_rq,
                enum idle_type idle)
{
    unsigned long old_load, this_load;
    unsigned long j = jiffies + CPU_OFFSET(this_cpu);
    struct sched_domain *sd;

    /* Update our load */
    old_load = this_rq->cpu_load;
    this_load = this_rq->nr_running * SCHED_LOAD_SCALE;
    /*
     * Round up the averaging division if load is increasing. This
     * prevents us from getting stuck on 9 if the load is 10, for
     * example.
     */
    if (this_load > old_load)
        old_load++;
    this_rq->cpu_load = (old_load + this_load) / 2;

    for_each_domain(this_cpu, sd) {
        unsigned long interval;

        if (!(sd->flags & SD_LOAD_BALANCE))
            continue;

        interval = sd->balance_interval;
        if (idle != SCHED_IDLE)
            interval *= sd->busy_factor;

        /* scale ms to jiffies */
```

```
        interval = msecs_to_jiffies(interval);
        if (unlikely(!interval))
            interval = 1;

        if (j - sd->last_balance >= interval) {
            if (load_balance(this_cpu, this_rq, sd, idle)) {
                /* We've pulled tasks over so no longer idle */
                idle = NOT_IDLE;
            }
            sd->last_balance += interval;
        }
    }
}
```

2.16.3 多核调度算法的优化

> 对任务的分配进行优化

使同一应用程序的任务尽量在一个核上执行，以便达到有共享数据的任务能够尽量在一个核上进行，而共享数据量少或者没有的任务尽量在不同核上进行。这样，可以显著地降低 cache 的缺失率，进而在很大程度上提升了系统的整体性能。

> 对任务的共享数据优化。

由于CMP体系结构共享二级缓存，可以考虑改变任务在内存中的数据分布，使任务在执行时尽量增加二级缓存的命中率。

> 对任务的负载均衡优化。

当任务在调度时，出现了负载不均衡，考虑将较忙处理器中与其他任务最不相关的任务迁移，以达到数据的冲突量最小。

2.16.4 绑定进程与CPU

> CPU 亲和性（$affinity$）

CPU 亲和性，即进程要在某个给定的CPU上尽量长时间地运行而不被迁移到其他处理器的倾向性。$Linux$ 内核进程调度器天生就具有被称为 `软CPU亲和性` 的特性，这意味着进程通常不会在处理器之间频繁迁移。

> 硬亲和性

2.6 版本 $Linux$ 内核还包含了一种机制，它让开发人员可以编程实现 `硬CPU亲和性` 。这意味着应用程序可以显式地指定进程在哪个（或哪些）处理器上运行。

> $cpus_allowed$

在 $Linux$ 内核中，所有的进程都有一个相关的数据结构，称为 $task_struct$。这个结构非常重要，其中与 亲和性（$affinity$）相关度最高的是 $cpus_allowed$ 位掩码。这个位掩码由 n 位组成，与系统中的 n 个逻辑处理器一一对应。 具有 4 个物理 CPU 的系统可以有 4 位。如果这些 CPU 都启用了超线程，那么这个系统就有一个 8 位的位掩码。

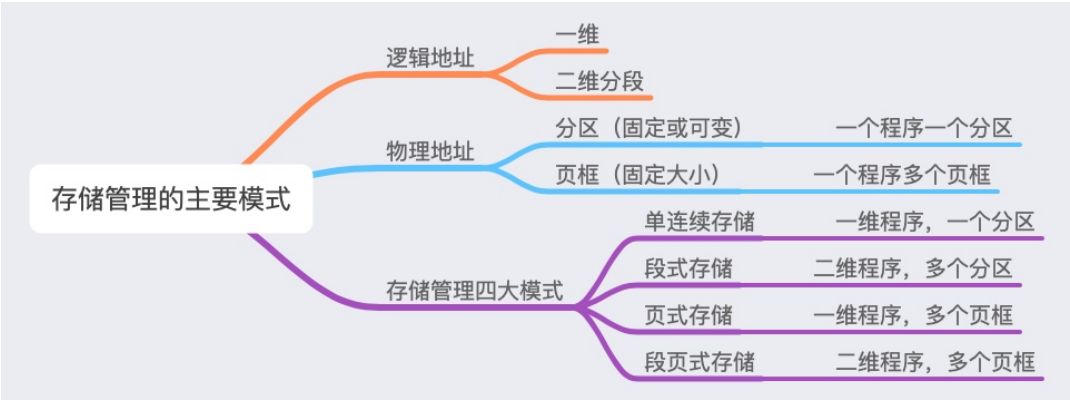
如果为给定的进程设置了给定的位，那么这个进程就可以在相关的 CPU 上运行。因此，如果一个进程可以在任何 CPU 上运行，并且能够根据需要在处理器之间进行迁移，那么位掩码就全是 1。实际上，这就是 $Linux$ 中进程的缺省状态。

$Linux$ 内核 API 提供了一些方法，让用户可以修改位掩码或查看当前的位掩码：

$sched_set_affinity()$ （用来修改位掩码）
$sched_get_affinity()$ （用来查看当前的位掩码）
注意，$cpu_affinity$ 会被传递给子线程，因此应该适当地调用 $sched_set_affinity$。

三、存储管理

3.1 存储管理的主要模式



3.1.1 逻辑地址

逻辑地址，又称相对地址，即用户编程所使用的地址空间。

逻辑地址从 0 开始编号，根据是否分段有两种形式:

1. 一维逻辑地址（地址）
2. 二维逻辑地址（段号,段内地址）

> 段式程序设计（逻辑地址分段）

方法: 把一个程序设计成多个段（代码段、数据段、堆栈段等）

段覆盖技术: 用户可以应用该技术扩充内存空间使用量。这一技术是程序设计技术，不是 OS

存储管理功能。

- 该技术是把程序划分为若干个功能上相对独立的程序段，按照其自身的逻辑结构使那些不会同时运行的程序段共享同一块内存区域。程序段先保存在磁盘上，当有关程序的前一部分执行结束后，把后续程序段调入内存，覆盖前面的程序段。

3.1.2 物理地址

物理地址，又称绝对地址，即程序执行所使用的地址空间。处理器执行指令时按照物理地址进行。

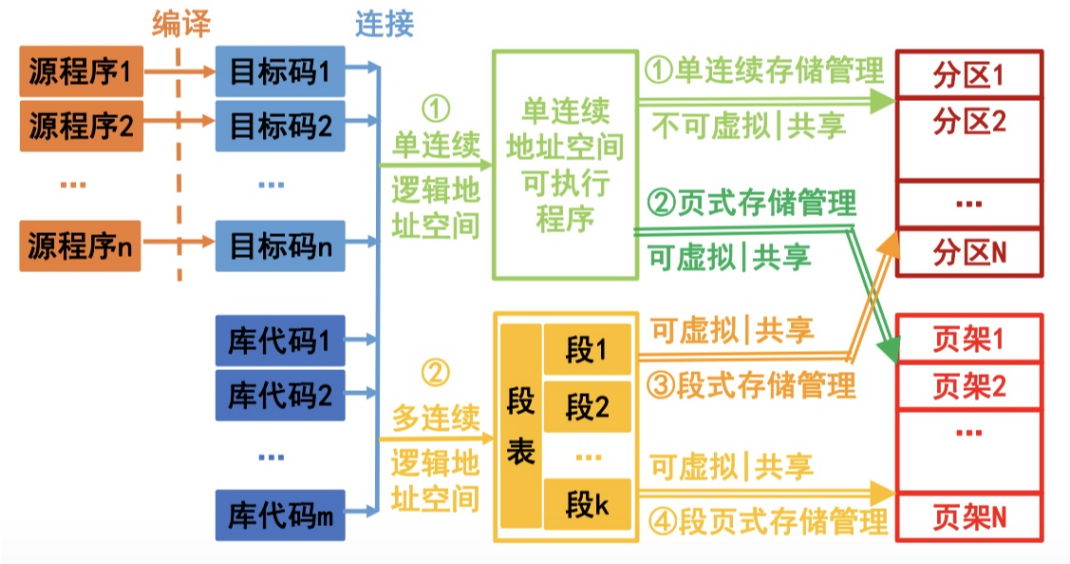
> 主存储器的复用

多道程序设计需要复用主存。

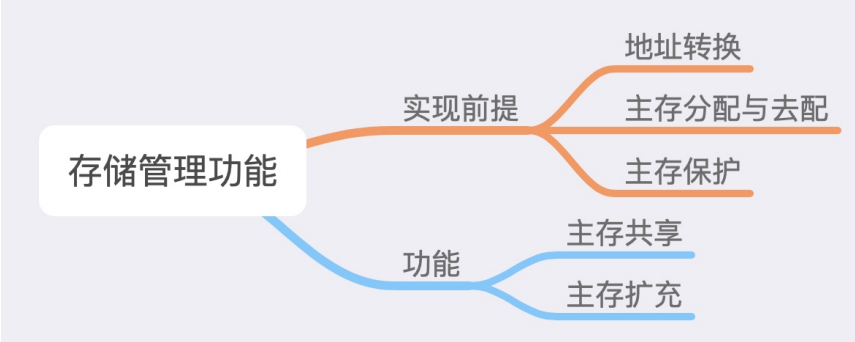
- 按照分区复用
  - 主存划分为多个固定 / 可变尺寸的分区
  - 一个程序 / 程序段占用一个分区
- 按照页框复用
  - 主存划分为多个固定大小的页框
  - 一个程序 / 程序段占用多个页框

3.1.3 存储管理的基本模式

1. 单连续存储管理
   - 程序（一维）
   - 对应一个主存固定分区或可变分区
2. 段式存储管理
   - 程序（分段，二维）
   - 对应多个主存可变分区
3. 页式存储管理
   - 程序（一维）
   - 对应多个主存页框
4. 段页式存储管理
   - 程序（分段，二维）
   - 对应多个主存页框



3.2 存储管理的功能



3.2.1 地址转换

> 概念

地址转换，即重定位，将逻辑地址转换成绝对地址。

根据地址转换的时间来进行分类。

> 静态重定位

在程序装入内存时进行地址转换。由装入程序执行，早期小型 OS 使用。

> 动态重定位

在 CPU 执行程序时进行地址转换，依赖硬件来进行转换。如果不用硬件，效率会极大地下降。

3.2.2 主存分配与去配

程序装入主存与离开主存时分别进行分配与去配操作。（符合常识）

> 分配

进程装入主存时，存储管理软件进行具体的主存分配操作，并设置一个表格记录主存空间的分配情况。

> 去配

当某个进程撤离或主动归还主存资源时，存储管理软件要收回它所占用的全部或者部分存储空间，调整主存分配表信息。

3.2.3 主存空间共享

> 共享主存资源

多个程序在主存中各自占用一定数量的存储空间，共同使用一个主存储器。（没有空间的复用）

> 共享主存区域

多个进程有共同的主存程序块或主存数据块。

3.2.4 存储保护

主存分为三类区域，我自己的、公共的、别人的。

> 私有主存区（我自己的）

可读可写

> 公共区

根据授权（可读/写授权）

> 非本进程信息（别人的）

不可读写

存储保护功能需要软硬件协同完成。

3.2.5 主存扩充

> 存储扩充技术（少调入、调别处）

把磁盘作为主存扩充，只把部分进程或进程的部分内容装入内存。

1. 对换技术（调别处）：把部分不运行的进程调出
2. 虚拟技术（少调入）：只调入进程的部分内容

> 软硬件实现存储扩充

1. 对换进程决定对换，则硬件机构执行调入
2. CPU处理到不在主存的地址，发出虚拟地址异常，OS将其调入，重新执行指令。

3.3 虚拟存储器的概念

3.3.1 提出原因

1. 主存容量限制非常不便于程序编写，且多道程序设计的道数受容量限制。
2. 而用户程序通常顺序执行，具有时空局部性，因此考虑只调入部分到主存中。

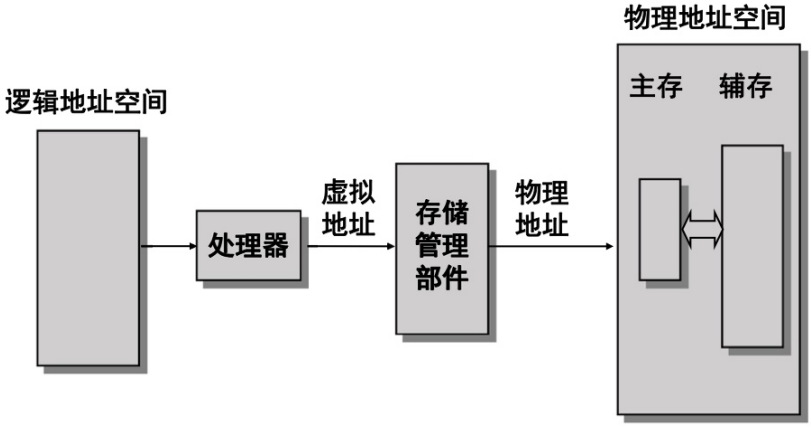
3.3.2 实现目标

1. 将有用的调出到主存。全部信息放入辅存，每次调入部分到主存，**随用随调**
2. 如果主存空闲空间不足，则将不用的调出到辅存。

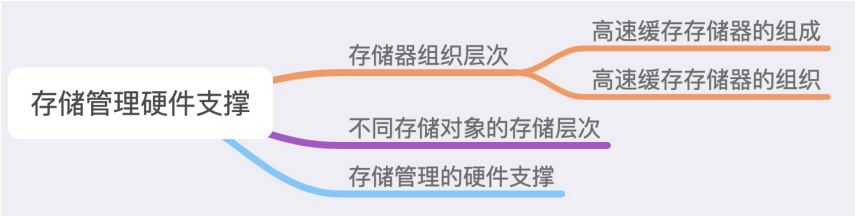
3.3.3 实现方法

- （辅存）虚拟地址空间: 位于磁盘的一块区域，容纳进程装入
- （主存）实际地址空间: 承载进程执行

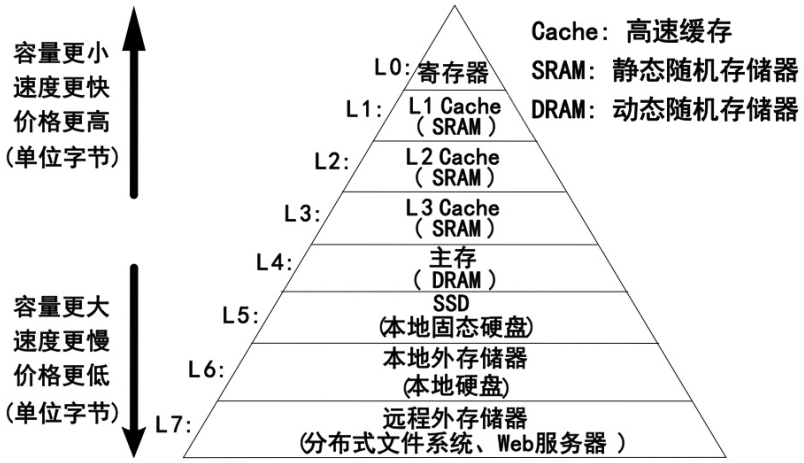
虚拟存储器是一种地址空间扩展技术，通常意义上对用户编程是透明的，除非用户需要进行高性能的程序设计。



3.4 存储管理的硬件支撑



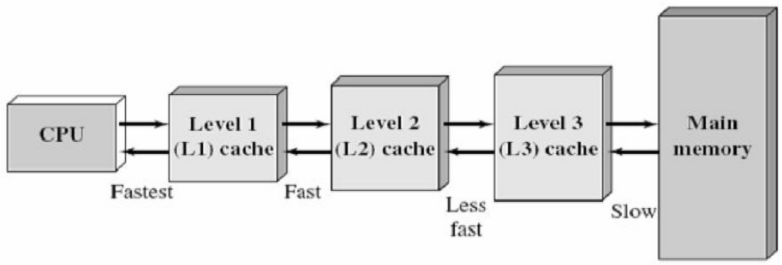
3.4.1 存储器组织层次



> 高速缓存存储器（Cache）的组成

- 高速存储器: 由 SRAM 组成的存储器
- 联想存储器: 根据内容进行寻址（与根据地址寻址进行区别）
- 地址转换部件
  - 通过联想存储器建立目录表，来实现快速地址转换。命中时直接访问 Cache，未命中则从主存中读取并放入 Cache。
- 替换部件
  - 在缓存已满时按一定策略进行数据块替换，并修改地址转换部件。

> 高速缓存存储器（Cache）的组织



- L1 Cache
  - 分为数据缓存和指令缓存，内置
  - 成本最高，对CPU性能影响最大，32KB～256KB之间
- L2 Cache

    - 外置和内置两种，512KB～8MB之间
- L3 Cache
    - 外置，在游戏和服务器领域有效
    - 对很多应用来说，总线改善比设置 L3 更加有利于提升系统性能

3.4.2 不同存储对象的存储层次

> Cache

关键性能数据被调入 Cache。

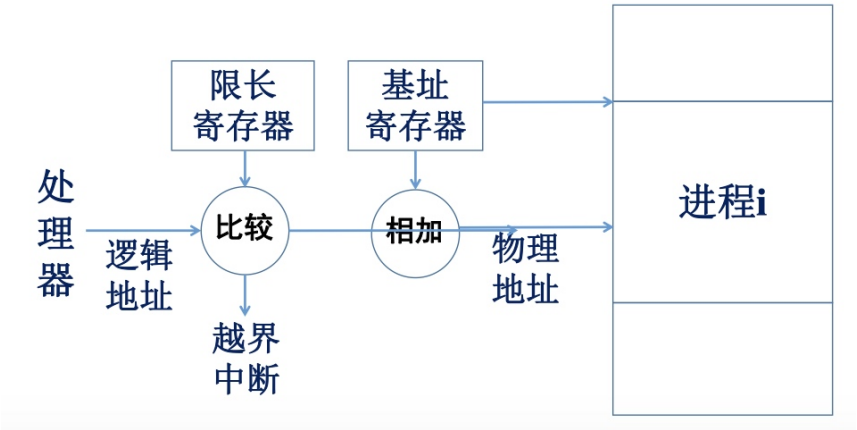
> 虚拟存储器

硬盘、固态硬盘、网络硬盘。

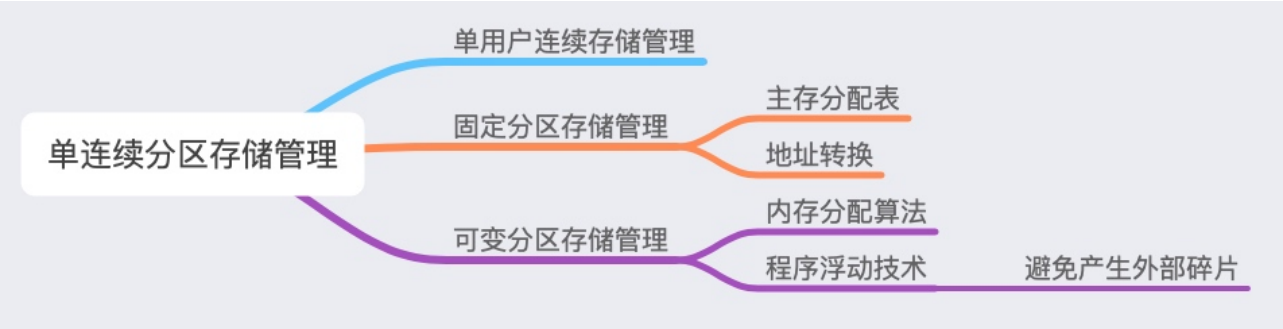
3.4.3 存储管理的硬件支撑

> 需要硬件支撑才有效果的功能

1. 动态地址转换
2. 存储保护（避免访问非法内存）
3. 虚拟地址越界判断
4. 页面替换



3.5 单连续分区存储管理



3.5.1 单连续分区概念

单连续分区: 每个进程占用一个物理上完全连续的存储空间

常见三种策略:

1. 单用户连续存储管理
2. 固定分区存储管理
3. 可变分区存储管理

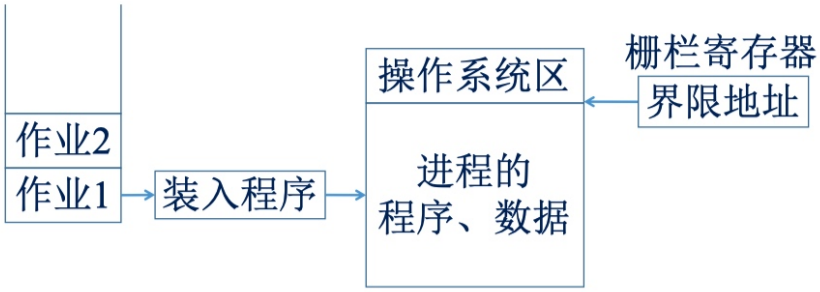
3.5.2 单用户连续存储管理

> 特点

1. 主存区域划分为系统区与用户区，设置一个栅栏寄存器划分两个区域，硬件用它在执行时进行存储保护。
2. 一般用静态重定位进行地址转换。
3. 适用于单用户单任务操作系统，如 DOS。

> 静态重定位方法

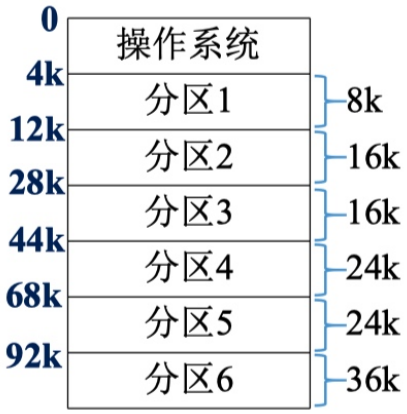
在装入一个作业时，把该作业中程序的指令地址和数据地址全部转换成绝对地址。



3.5.3 固定分区存储管理

> 特点

1. 支持多个分区
2. 分区数量固定
3. 分区大小固定
4. 可用静态/动态重定位
5. 早期 OS 采用



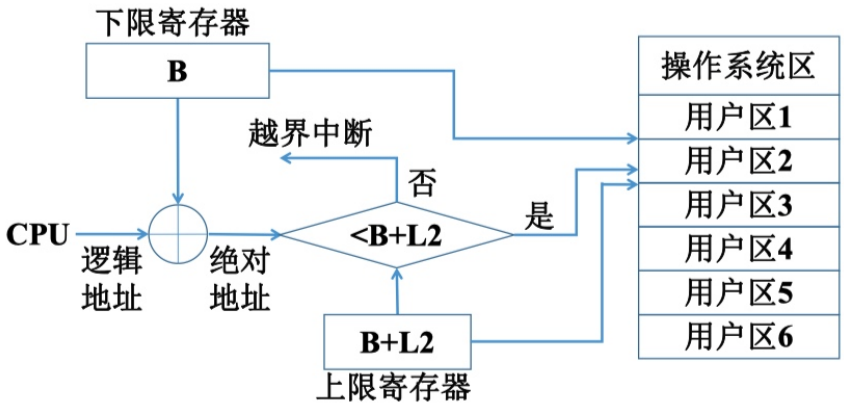
> 主存分配方式

维护一个主存分配表，实行分配与去配功能。

分区号	起始地址	长度	占用标志
1	4K	8K	0
2	12K	16K	Job1
3	28K	16K	0
4	44K	24K	0
5	68K	24K	Job2
6	92K	36K	0

> 地址转换方式

用硬件实现动态重定位。



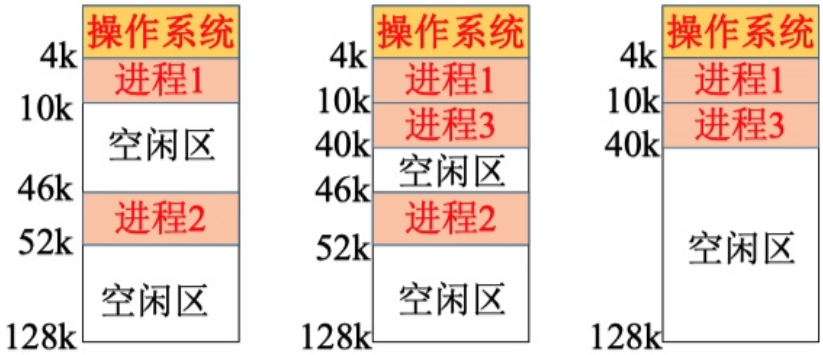
> 缺点

固定分区存储管理不够灵活，既不适应大尺寸程序，又存在内存内零头，有浪费。

3.5.4 可变分区存储管理

- 按进程的内存需求来动态划分分区

- 创建一个进程时，根据进程所需主存量查看主存中是否有足够的空闲空间
  - 若有，则按需要量分割一个分区
  - 若无，则令该进程等待主存资源
- 由于分区大小按照进程实际需要量来确定，因此分区个数是随机变化的



可变分区的主存分配表用链表实现。

起址	长度	标志
4k	6k	J1
46k	6k	J2
		空
		空
⋮	⋮	⋮

(a)已分配区情况表

起址	长度	标志
10k	36k	未分配
52k	76k	未分配
		空
		空
⋮	⋮	⋮

(b)未分配区情况表

3.5.4.1 可变分区内存分配算法

> 最先适应分配算法

空闲区表根据实址先后连成链表，每次选取最前面的分区进行分配。

> 邻近适应分配算法

空闲区表组织成循环链表，从上一次分配的地址开始查找符合要求的分区。

> 最优适应分配算法

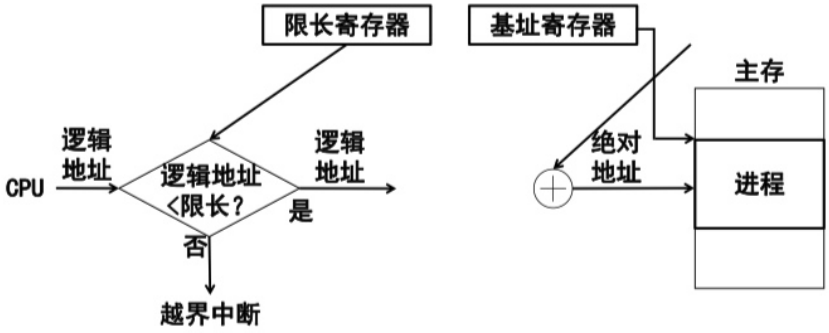
按照分区大小组织成链表，每次找最小的分区进行分配。

> 最坏适应分配算法

按照分区大小组织成链表，每次找最大的分区进行分配。

3.5.4.2 地址转换与存储保护

用硬件来实现动态重定位。



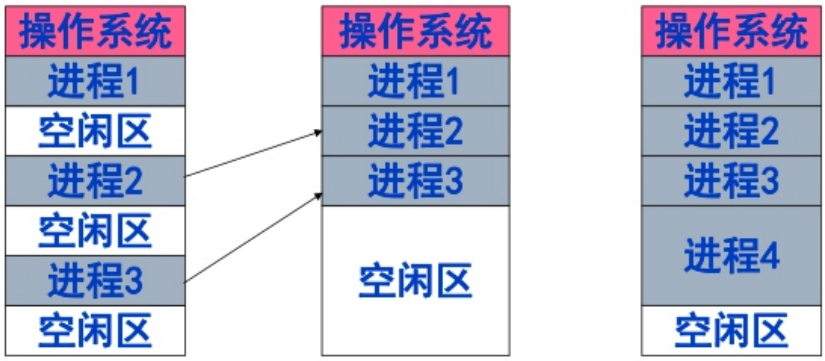
3.5.4.3 程序浮动技术

> 可变分区方式的内存外零头

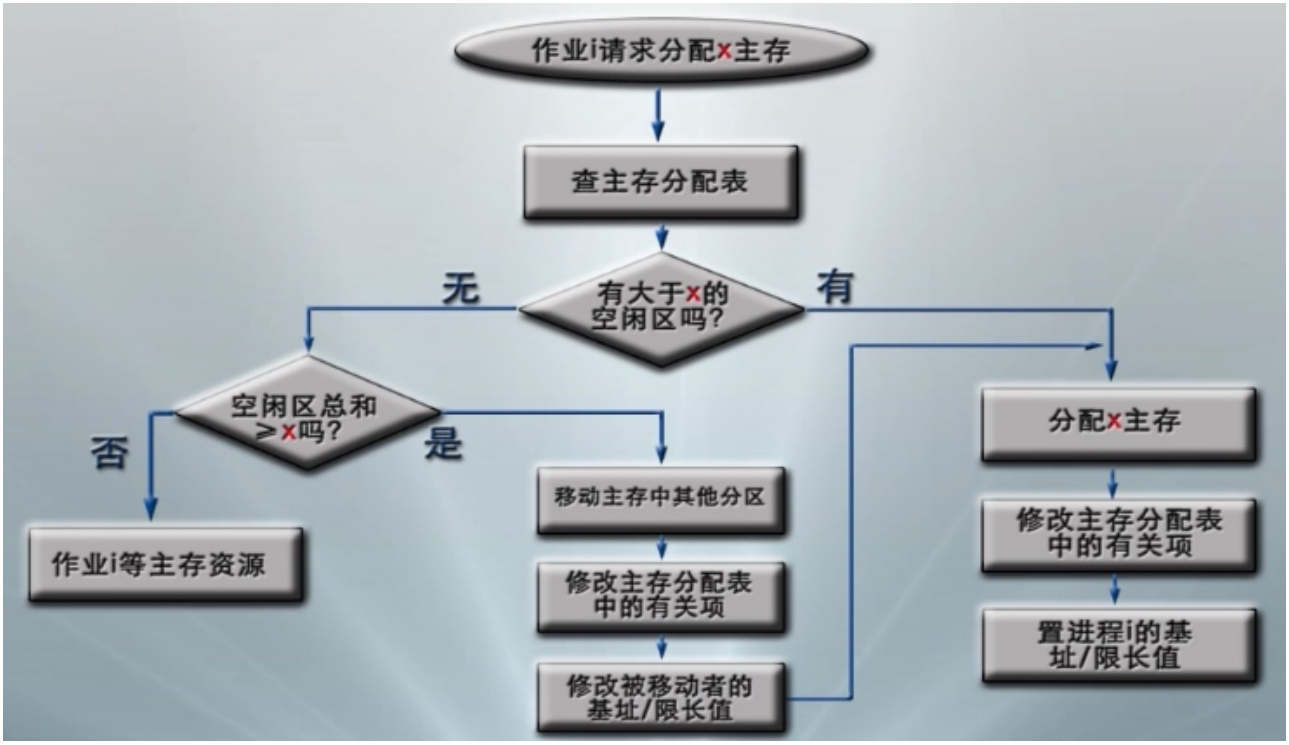
- 固定分区方式会产生内存内零头
- 可变分区方式也会随着进程的内存分配产生一些小的不可用的内存分区，称为内存外零头
- 最优适配算法最容易产生外零头
- 可变分区方式下，任何适配算法都不能避免产生外零头

> 移动技术（程序浮动技术）

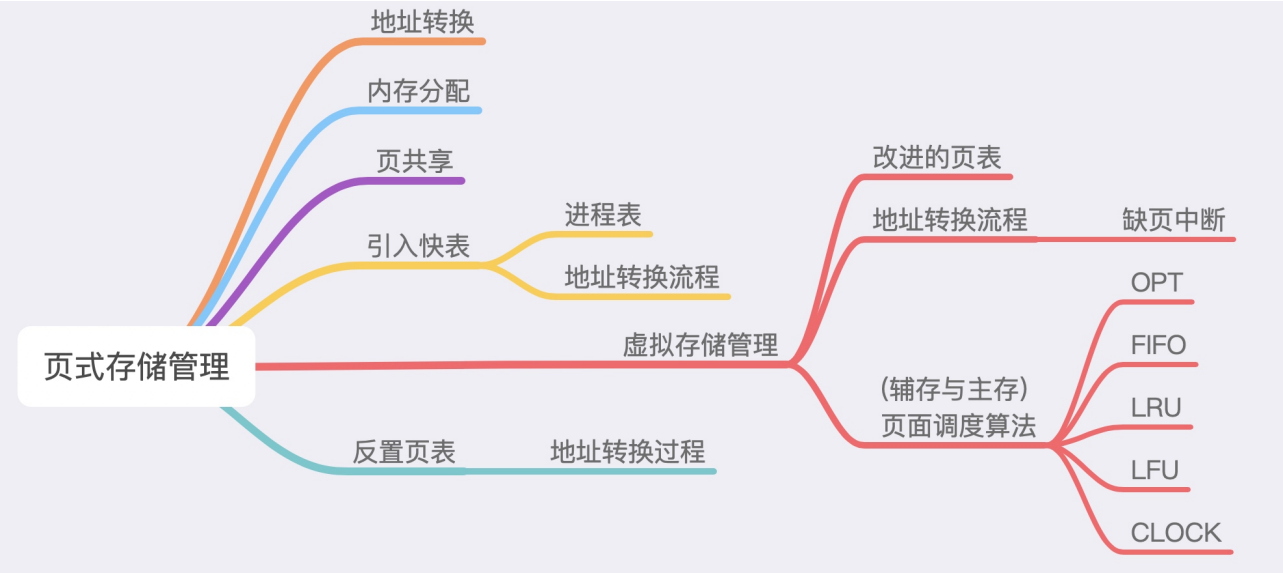
- 移动分区以解决内存外零头
- 由于移动分区会导致物理地址变化，因此需要动态重定位支撑



> 移动技术的工作流程



3.6 页式存储器



3.6.1 基本原理

- 分页存储器将主存划分成多个大小相等的页架
- 受页架尺寸限制，程序的逻辑地址也自然分成页
- 不同的页可以放在不同页架中，不需要连续
- 页表用于维系进程的主存完整性

进程页表

页0	页架号
页1	页架号
页2	页架号
页3	页架号
…	……

3.6.2 地址转换

> 逻辑地址

逻辑地址由页号与单元号组成。

页号	单元号

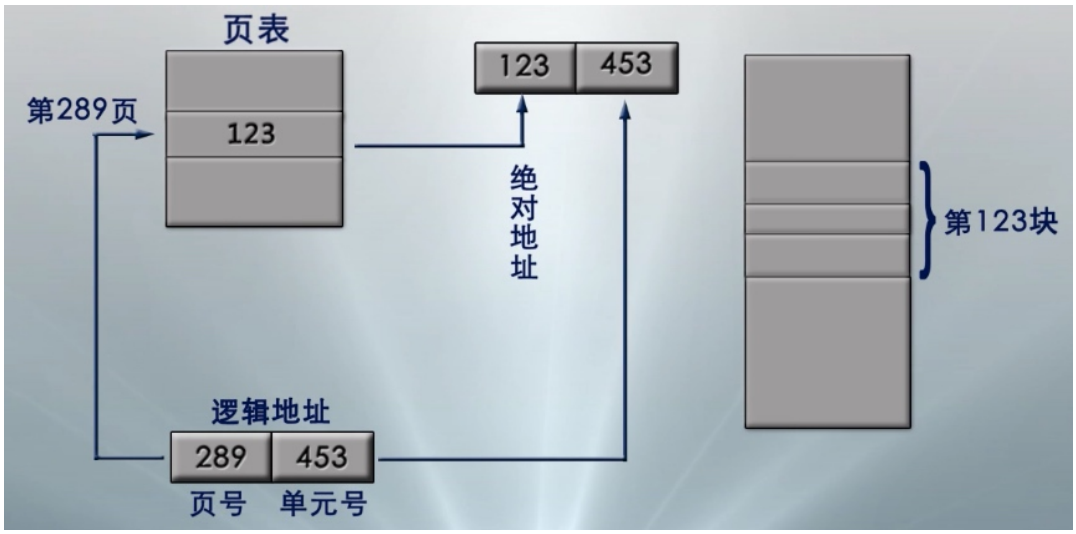
> 物理地址

物理地址由页框号与单元号组成。

页架号	单元号

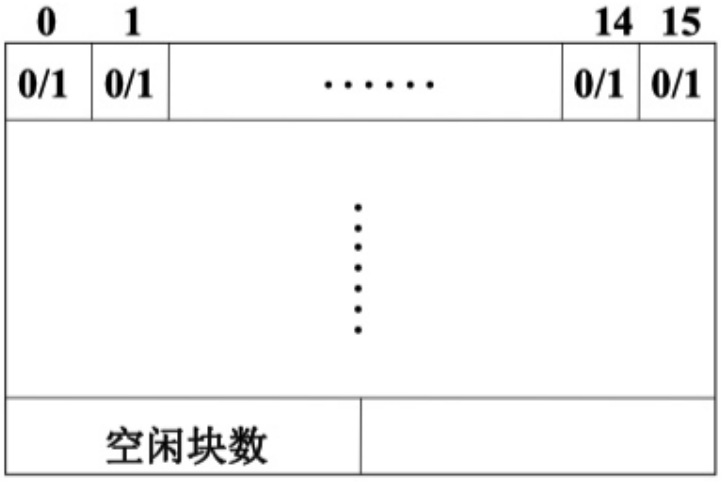
> 地址转换

上述的地址转换可以直接通过查页表完成。



3.6.3 内存分配

由于内存分成的页大小、数量均固定，因此我们可以用一张位示图来记录主存分配情况。同时用一张页表来维护主存逻辑完整性。



3.6.4 页共享

页式存储能够实现多个进程共享程序和数据。

> 数据共享

不同进程，尽管页号不同，也可以共享数据页。

> 程序共享

由于代码页中可能存在 (JMP <页内地址>) 这样的指令，因此不同进程，必须在同一页号内，才能共享代码页。

3.6.5 快表

> 引入快表的原因

页表放在主存，导致每次地址转换必须访问两次主存

1. 按页号读出页表中的相应页框号
2. 按计算出来的绝对地址进行读写

> 解决方法

为了提高地址转换速度，设置一个专用的高速存储器（Cache），用来存放页表的一部分。

- 快表，存放在高速存储器中的页表部分
- 快表表项，(页号，页框号)
- 存放快表的高速存储器是**联想存储器**，即按照内容寻址，而非按照地址访问

> 基于快表的地址转换流程

1. 按逻辑地址中的页号查快表
2. 若该页已在快表中，则由页框号和单元号形成绝对地址

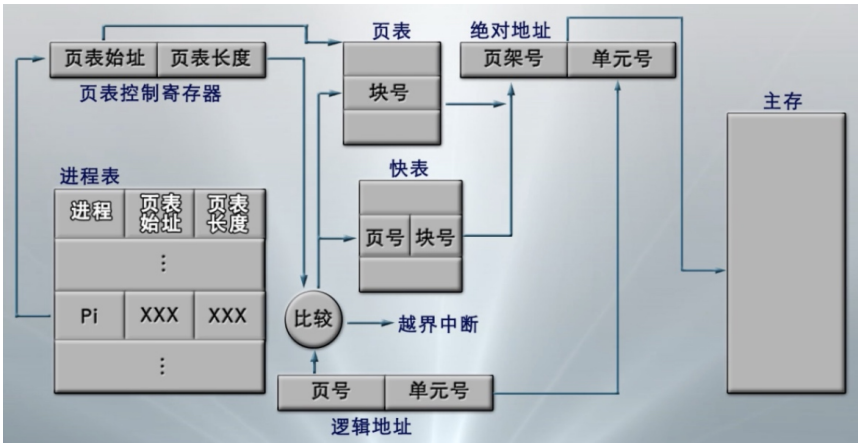
3. 若该页不在快表中，则再查主存页表形成绝对地址，同时将该页登记到快表中
4. 当快表填满后，又要登记新页时，则需在快表中按一定策略淘汰一个旧登记项目

> 进程表

- 每个进程都有一个页表，而进程表登记了每个进程的页表。
- 进程占有处理器运行时，其**页表起始地址和长度**送入页表控制寄存器。

用户作业名	页表始址	页表长度
AB	0010	4
CD	0014	3
EF	0017	7
...	...	...

> 地址转换具体过程



3.6.6 虚拟存储管理

> 基本思想

- 把进程全部页面装入虚拟存储器，执行时先把部分页面装入实际内存，然后根据执行行为，动态调入不在主存的页，同时进行必要的页面调出。
- 现代 OS 的主流存储管理技术。
- 首先只把进程第一页信息装入主存，称为**请求页式存储管理**。

> 改进页表

由于虚拟存储管理的引入，我们需要扩充原页表表项。

- 每页的虚拟地址、实际地址

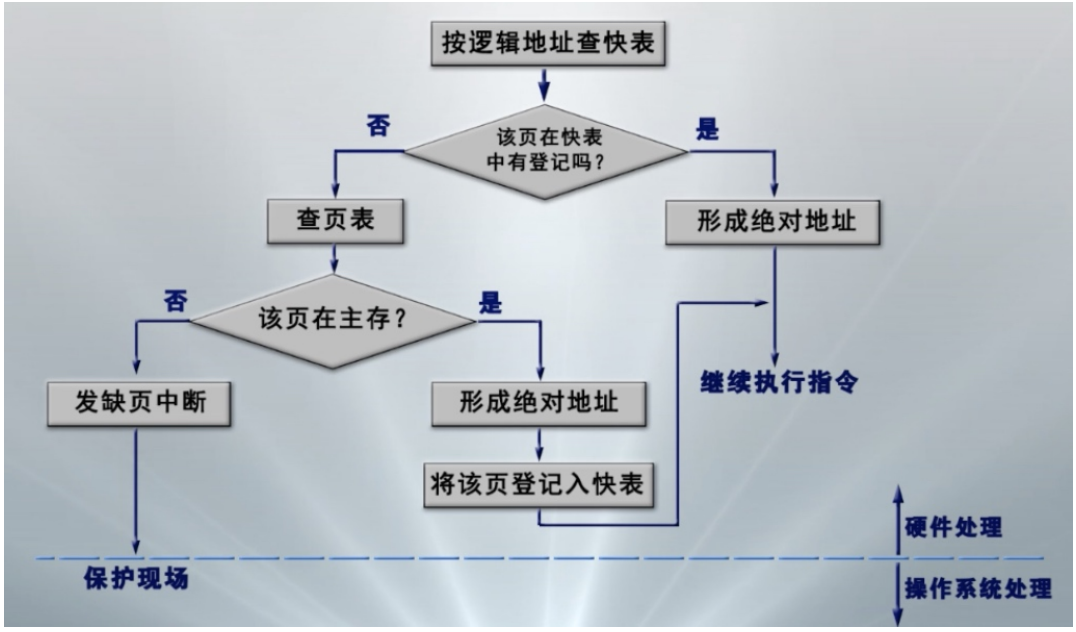
- 主存驻留标志（是否在主存内）
- 写回标志（是否被修改过，被修改过则要写回辅存中）
- 保护标志（用于存储保护）
- 引用标志（页面淘汰使用）
- 可移动标志（正在与外设进行交换的页面是不能被移动的）

	标志位	主存块号	辅助存储器地址
第0页			
...			
第n页			

> 地址转换实现

CPU 处理地址:

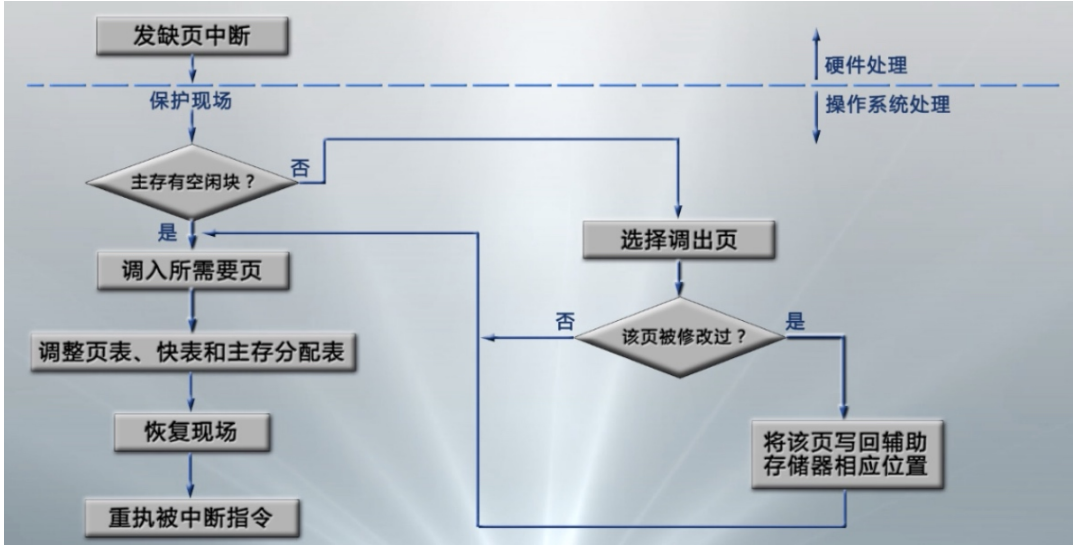
- 若页驻留，则获得块号形成绝对地址
- 若页不在主存，则 CPU 发出缺页中断



OS 处理缺页中断:

- 若有空闲页框，则根据辅存地址调入页，更新页表与快表等
- 若无空闲页框，则决定淘汰页，调出已修改页，调入页，更新页表与快表

由于这条指令没有执行完就中断了，因此中断处理结束后需要回退执行该指令。



3.6.7 辅存与主存的页面调度

3.6.7.1 页面调度概念

- 当主存空间已满但又需要装入新页时，页式虚拟存储管理必须按照一定的算法把已在主存的一些页调出去
- 选择淘汰页的工作称为**页面调度**
- 选择淘汰页的算法称为**页面调度算法**
- **抖动/颠簸:** 页面调度算法设计不当，使得刚被淘汰的页面立即又要调入。

3.6.7.2 缺页中断率

> 定义

假定进程 P 共 n 页，系统分配页框数为 m 个。P 运行中成功访问次数为 S，不成功访问次数为 F，总访问次数为 A = S + F，则缺页中断率为

$$f = \frac{F}{A}$$

缺页中断率是衡量存储管理性能和用户编程水平的重要依据。

> 影响缺页中断率的因素

1. 分配给进程的页框数
2. 页面大小: 页面尺寸越大，则缺页中断率就越低
3. 用户程序编制方法（矩阵按行访问还是按列访问）

3.6.7.3 OPT 页面调度算法

> 算法描述

当要调入新页面时，首先淘汰以后不再访问的页，然后选择距现在最长时间后再访问的页。

> 算法特点

该算法为理想的调度算法，由 Belady 提出，称为最佳算法，不具有实现性。

3.6.7.4 FIFO 页面调度算法

> 算法描述

考虑程序执行的顺序性，总是淘汰最先调入主存的那一页，或在主存驻留时间最长的那一页。

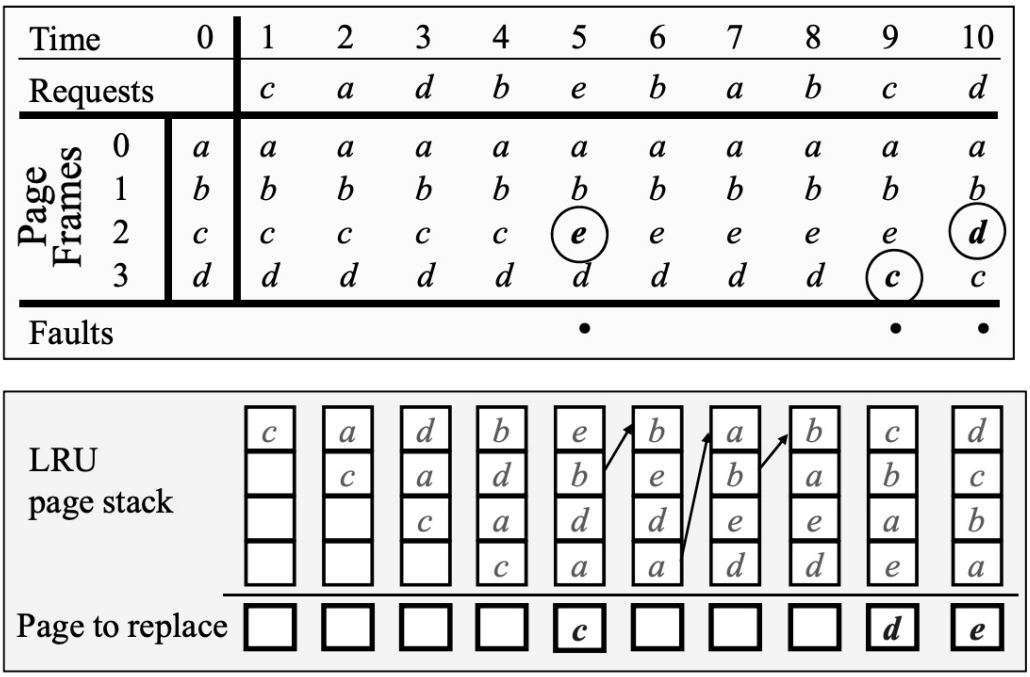
3.6.7.5 LRU 页面调度算法

> 算法描述

FIFO 仅考虑了顺序性，没有考虑循环性，因此我们改进这个算法。

淘汰最近一段时间较久未被访问的那一页，因为那些刚被使用过的页面可能马上还要被使用到。

- Maintain a “stack” of recently used pages



> 算法实现

该算法的严格实现需要用到优先队列或 set，时间开销非常大，因此考虑下述的近似实现方法。

1. 每页建一个引用标志，供硬件使用
2. 设置一个时钟中断，中断时页引用标志置 0
3. 地址转换时，页引用标志置 1
4. 淘汰页面时，从页引用标志为 0 的页中随机选择一个淘汰

该模拟算法不仅很难确定时间间隔，而且性能非常差。

3.6.7.6 LFU 页面调度算法

由于 LRU 算法精准实现开销太大，因此我们再换一种近似方式。

> 算法描述

淘汰最近一段时间内访问次数较少的页面，对 OPT 的模拟性比 LRU 更好。

> 算法实现

1. 基于时间间隔中断，并给每一页设置一个计数器
2. 时间间隔中断发生后，所有计数器清 0
3. 每访问页一次就给计数器加 1
4. 选择计数值最小的页面淘汰

3.6.7.7 时钟 CLOCK 页面调度算法

> 算法描述

- 采用循环队列机制构造页面队列，形成了一个类似于钟表面的环形表。
- 队列指针则相当于钟表面上的表针，指向可能要淘汰的页面。
- 使用页引用标志位。

> 算法实现

1. 页面调入主存时，其引用标志位置 1
2. 访问主存页面时，其引用标志位置 1
3. 淘汰页面时，从指针当前指向的页面开始扫描循环队列
    - 把所遇到的引用标志位是 1 的页面的引用标志位清 0，并跳过
    - 把所遇到的引用标志位是 0 的页面淘汰，调入请求页，并将指针移动一步

注意: 只有缺页查找替换页时指针才会移动，页面命中时指针不会移动。

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	e	e	e	e	e	d
	1	b	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	c	c	a	a	a	a
	3	d	d	d	d	d	d	d	d	d	c	c
Faults							•		•		•	•

Page table entries for resident pages:

1	a	1	e	1	e	1	e	1	e	1	e	1	e	1	d
1	b	0	b	1	b	0	b	1	b	1	b	0	b
1	c	0	c	0	c	1	a	1	a	1	a	0	a
1	d	0	d	0	d	0	d	0	d	1	c	0	c

3.6.8 反置页表

3.6.8.1 提出原因

页表以及相关硬件机制在地址转换、存储保护、虚拟地址访问中发挥了关键作用，因此为页式存储管理设置了专门硬件机构 —— MMU（内存管理单元）

> MMU

CPU 管理虚拟/物理存储器的控制线路，把虚拟地址映射为地理地址，并提供存储保护，必要时确定淘汰页面。

> IPT (Inverted Page Table)

反置页表 IPT 是 MMU 使用的数据结构。

3.6.8.2 反置页表设计

> 功能

1. 针对内存中的每个页框建立一个页表，按照块号排序
2. IPT 用来完成内存页框到访问进程页号的对应，即物理地址到逻辑地址的转换，因此为反置页表

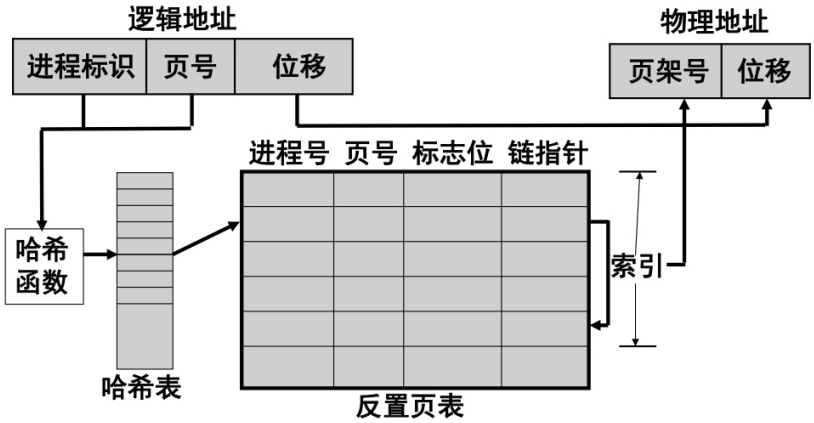
> 反置页表项

1. 页号: 虚拟地址页号
2. 进程标识: 使用该页的进程号
3. 标志位: 有效位、引用位、修改位、保护位
4. 链指针: 哈希链

3.6.8.3 IPT 地址转换

> 转换过程

1. MMU 通过哈希函数把进程标识和虚页号转换成一个哈希值，并通过哈希表找到对应的 IPT 表项
2. IPT 表项的索引就是页框号，通过拼接位移即可生成物理地址
3. 若遍历整个哈希表都未能找到匹配页表项，说明该页不在内存，产生缺页中断，请求 OS 调入



3.7 段式存储器

3.7.1 段式程序设计

每个程序可由若干段组成，每一段都可以从 “0” 开始编址，段内的地址是连续的。

分段存储器的逻辑地址由两部分组成。同时要与分页的逻辑地址进行区分。

- 段号:单元号 —— 用户程序设计自己设定的，用户可控制
- 页号:单元号 —— 系统自动切割，用户不知情。

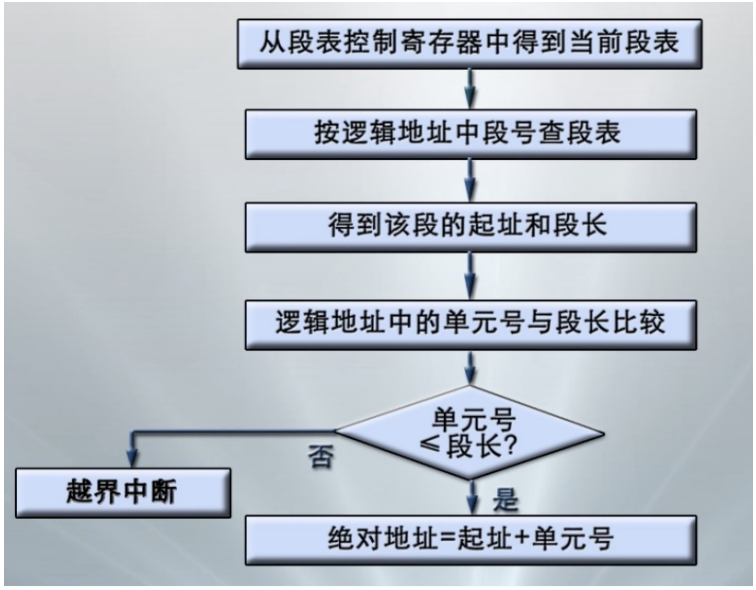
3.7.2 段式存储管理

- 段式存储管理**基于可变分区存储管理实现**，一个进程要占用多个分区
- 硬件需要增加一组用户可见的段地址寄存器（代码段、数据段、堆栈段、附加段），供地

址转换使用

- 存储管理需要给**每一个进程设置一个段表，每个段占用一个段表项**，其中包括: 段始址、段限长，以及存储保护、可移动、可扩充等标志位

> 地址转换



> 段的共享

- 通过不同进程段表中的项指向同一个段基址来实现
- 对共享段的信息必须进行保护，如规定 "只能读出不能写入，不满足保护条件则产生保护中断"

3.7.3 虚拟存储管理

> 基本方法

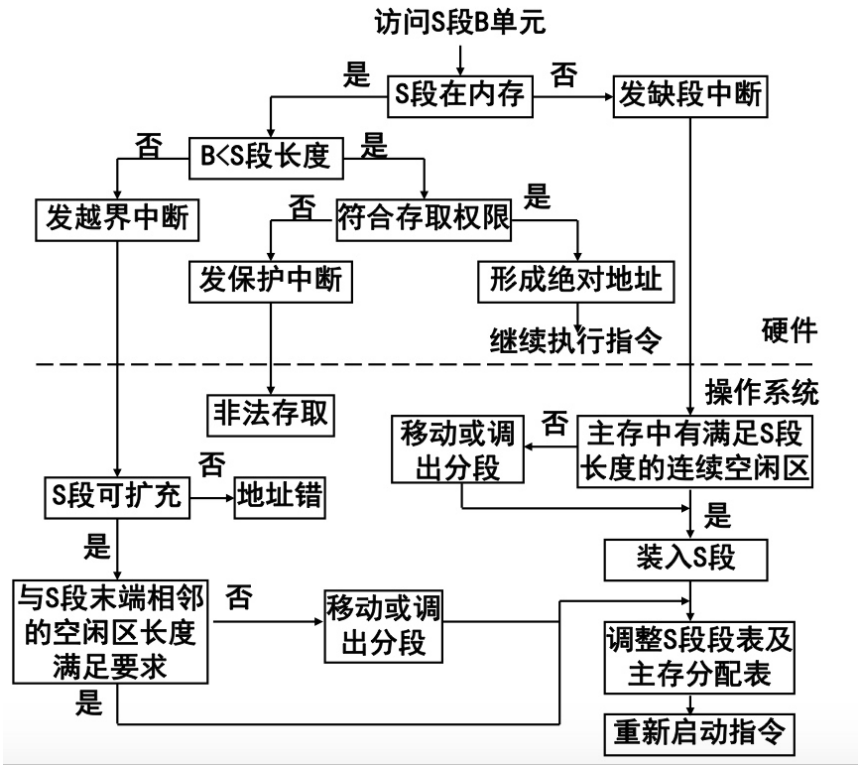
- 把进程的所有分段都存放在辅存中，进程运行时先把当前需要的一段或几段装入主存，在执行过程中访问到不在主存的段时再把它们动态装入
- 段式虚拟存储管理中的段调进调出是由 OS 自动实现的，对用户透明
- 与段覆盖技术不同，它是用户控制的主存扩充技术，OS不感知

> 段表扩充

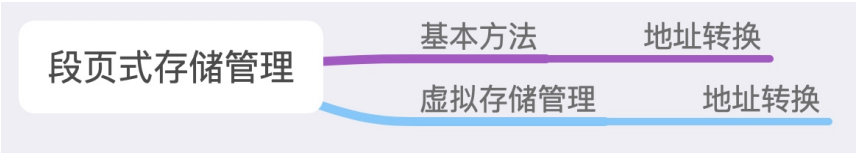
- 特征位: 00(不在内存)、01(在内存)、11(共享段)
- 存取权限: 00(可执行)、01(可读)、11(可写)
- 扩充位: 0(固定长)、1(可扩充)
- 标志位: 00(未修改)、01(已修改)、11(不可移动)

段号	特征	存取权限	扩充位	标志	主存始址	限长	辅存始址

> 地址转换



3.8 段页式存储器



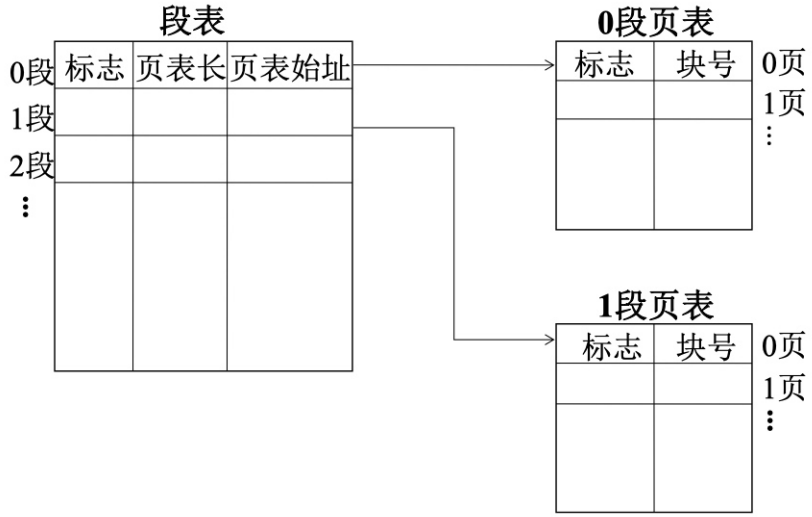
3.8.1 段页式管理方法

> 段页式存储管理

- 段式程序设计可以基于页式存储管理实现，从而实现段页式存储管理
- 每一段不必占据连续的存储空间，可存放在不连续的主存页框中
- 显然，程序执行时可以仅装入部分段或段中部分页面，实现虚拟存储管理

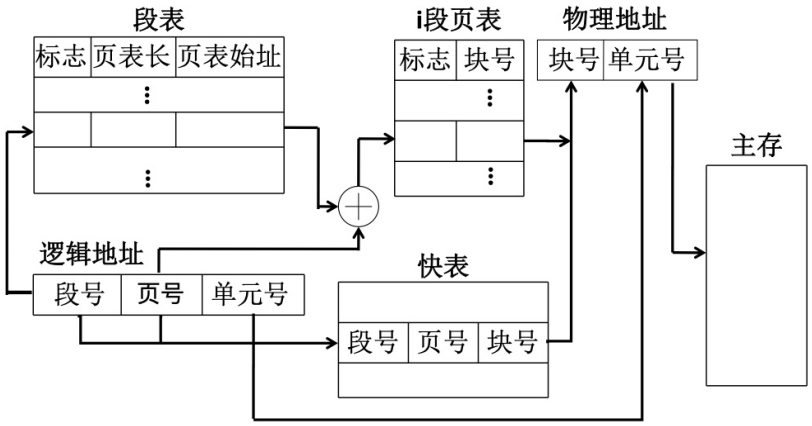
> 段表与页表

段表中从之前的起址与长度，改为起址与页表长。



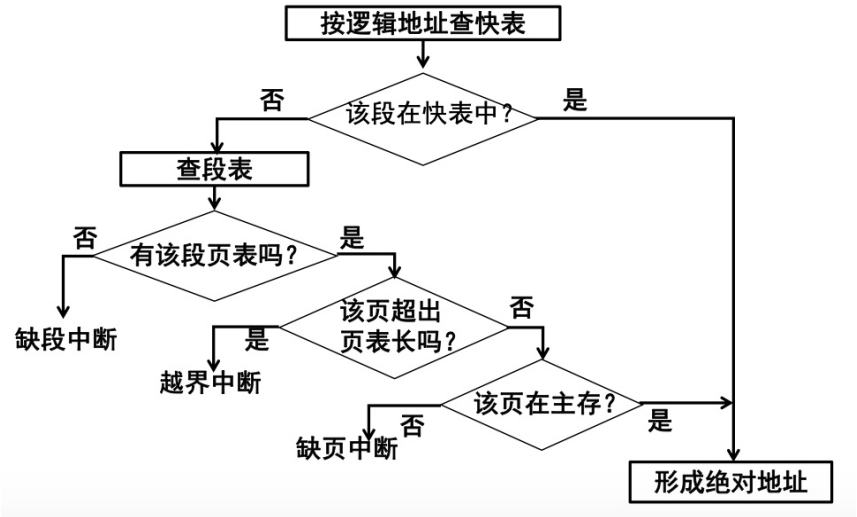
> 段页式存储管理的地址转换

逻辑地址根据段号找到段表，然后判断单元号有无超过页表长度，之后再在页表中找到对应的块号。



3.8.2 段页式虚拟存储管理

形成绝对地址之后还需要更新快表。



四、设备管理 —— I/O 管理

4.1 设备管理概述

4.1.1 I/O 设备

> I/O 设备概述

背景: 现代计算机系统通常配备大量的 I/O 设备，用于计算机系统与外部世界（如用户、其它计算机或电子设备等）进行信息交换或存储

- I/O 设备又称为外围设备或外部设备，简称外设
- I/O 操作: 内存和 I/O 设备之间的信息传送操作

> 按信息传输方向划分 I/O 设备

- 输入设备: 将外界信息输入计算机（键盘、鼠标）
- 输出设备: 将计算结果输出（显示屏、打印机）
- 输入输出设备: 既可以输入信息，也可以输出信息（网卡、磁盘）

> 按交互功能划分 I/O 设备

- 人际交互设备: 用于用户与计算机之间的交互通信（鼠标、键盘、显示器）
- 存储设备: 持久性地存储大量信息并快速检索（磁盘、光盘）
- 机机通信设备: 用于计算机和计算机之间的通信（网卡、调制解调器）

> 按设备管理划分 I/O 设备

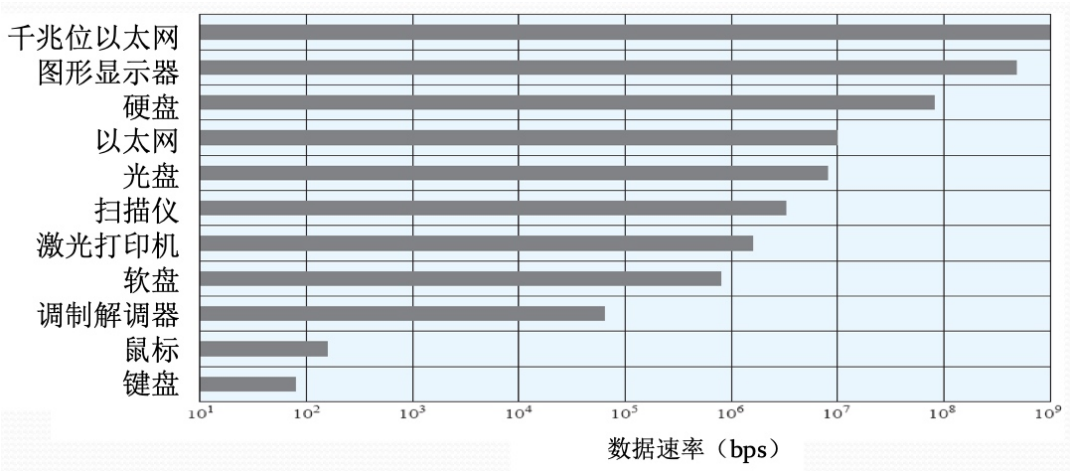
- 字符设备: 以字符为单位进行信息交换（大多数人机交互设备）
- 块设备: 以固定大小的数据块进行信息交换（磁盘）

- 网络设备: 用于与远程设备通信的设备（网卡）

4.1.2 设备管理

> 目标

1. 克服设备和 CPU 速度不匹配所引起的问题。令主机和设备并行工作，提高设备使用效率。
2. 对设备进行抽象，屏蔽设备的物理细节和操作过程，配置驱动程序，提供统一界面，供用户或高层软件使用。
    - 抽象为文件系统中的节点，统一管理
    - 裸设备: 不被操作系统直接管理，由应用程序读写，I/O 效率更高



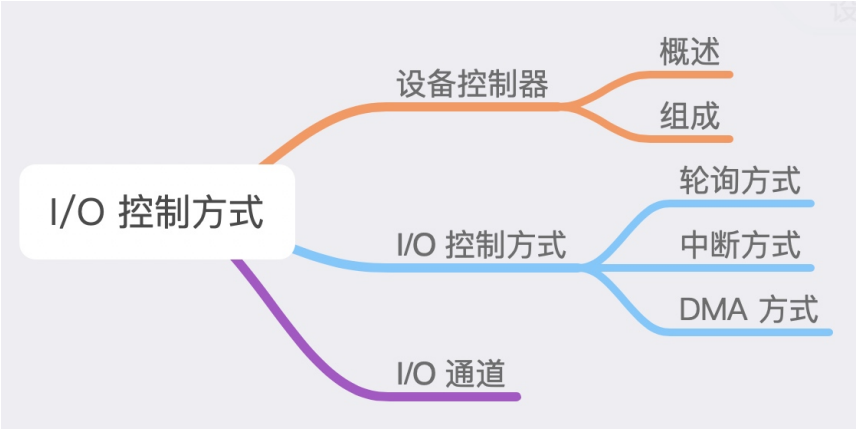
> 管理内容

1. 设备中断处理
2. 缓冲区管理
3. 设备的分配与去配
4. 设备驱动调度
5. 实现虚拟设备

> 管理层次

- I/O 硬件: I/O 设备、接口线路、控制部件、通道
- I/O 软件: 系统 I/O 软件、用户空间 I/O 软件

4.2 I/O 控制方式



4.2.1 设备控制器

> 概述

将 I/O 设备中的机械部件和电子部件分开处理，其中电子部件称为设备控制器。

设备控制器又称为设备适配器、I/O 控制器、I/O 控制接口，简称 I/O 模块或 I/O 接口。

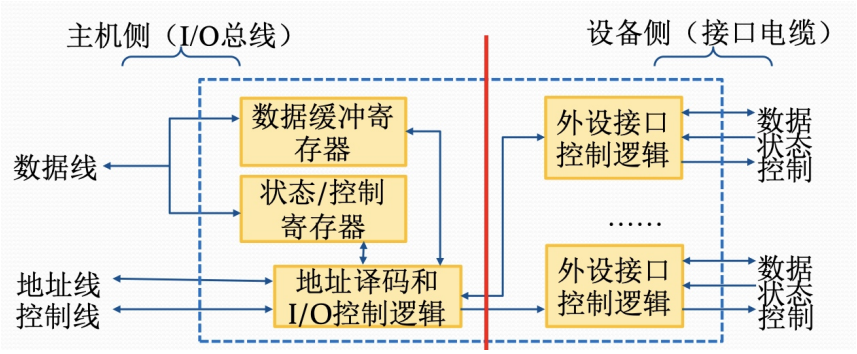
OS 与控制器交互，而非与设备交互。

> 功能

1. 接受、识别命令（来自 CPU 或通道）
2. 数据交换
3. 记录设备状态（供 CPU 处理）
4. 设备地址识别（连接多台设备时）

> 组成

- 与主机交互
  - （记录设备状态）状态/控制寄存器
  - （数据交换）数据缓冲寄存器
  - （接受命令）地址译码和 I/O 控制逻辑
- 与设备交互
  - 外设接口控制逻辑

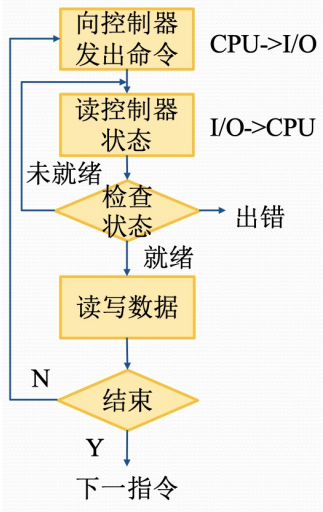


4.2.2 轮询方式

> 具体流程

轮询方式是忙等待的过程。

1. 处理器向控制器发送一个 I/O 命令
2. 如果设备未就绪，则反复查看 I/O 状态，直至设备就绪
3. 执行数据交换



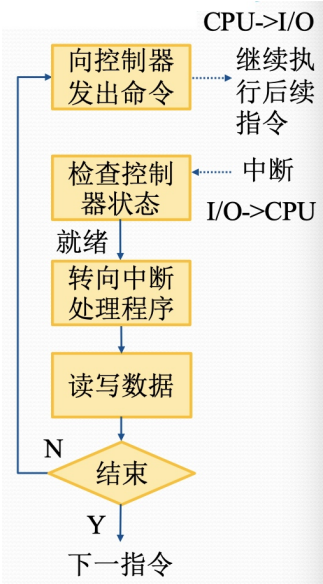
> 特点

1. 处理 I/O 请求会终止原程序的执行
2. CPU 需要参与数据传送
3. CPU 与设备串行工作，效率低下

4.2.3 中断方式

> 具体流程

1. 处理器向控制器发出 I/O 命令，然后继续执行后续指令。
    - 如果该进程支持异步 I/O，则继续执行该进程。
    - 否则，进程在这个中断点挂起，CPU 执行其他程序。
2. 控制器检查设备状态，就绪后发起中断。
3. CPU 响应中断，转向中断处理程序。
4. 中断处理程序执行数据读写操作。
5. 恢复执行原先程序。



> 特点

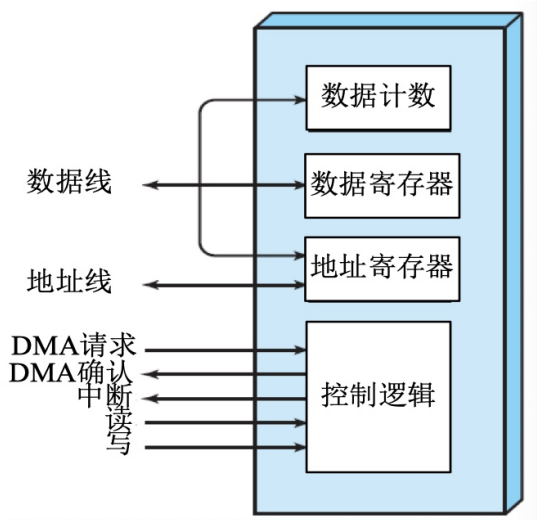
1. CPU 不需要等待 I/O 设备就绪
2. CPU 仍需要参与数据传送（响应中断后，原程序会停止执行）
3. CPU 和设备部分并行，效率提高

4.2.4 DMA 方式

> DMA 模块

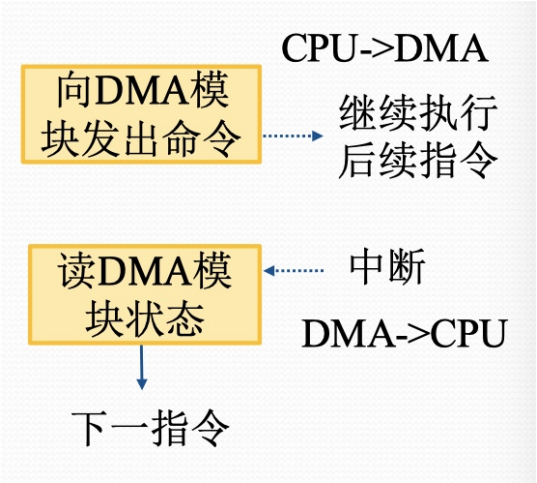
DMA: 直接存储器访问

DMA 模块: 模仿处理器来控制主存和设备控制器之间的数据交换



> 具体流程

1. 处理器向 DMA 模块发出 I/O 命令
2. 处理器继续执行其它工作，DMA 模块负责传送全部数据
3. 数据传送结束后，DMA 中断处理器



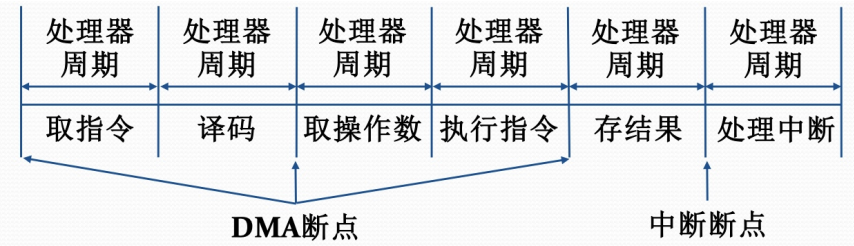
> 特点

1. CPU 不会终止原程序的执行（CPU 无须等待 I/O 设备就绪，也无需进行数据传送）
2. CPU 只在数据传送的开始和结束时参与
    - 开始时，CPU 需要对 DMA 模块进行初始化
    - 结束时，CPU 响应中断，但不必保存现场

> DMA 方式中的周期窃取

当 DMA 和 CPU 同时经总线访问内存时，CPU 总是将总线的占有权让给 DMA 一个或几个主存周期。

由于 CPU 大部分情况下与 Cache 进行数据交换，因此周期窃取对 CPU 与主存的数据交换速率影响不大。



4.2.5 三类 I/O 控制方式总结

> 轮询方式

CPU 需要等待设备就绪，且参与数据传送。

> 中断方式

CPU 无需等待设备就绪，但响应中断后参与数据传送。

> DMA 方式

CPU 在数据传送开始和结束时参与，与主存进行数据交换时不参与。

CPU 作用	等待设备	数据传送
轮询方式	需要	参与
中断方式	不需要	参与
DMA 方式	不需要	不参与

4.2.6 I/O 通道

> 概述

I/O 通道，又称为通道控制器、I/O 处理器。

设备控制器包含**自身专用的处理器**和通道程序。

- I/O 指令不再由处理器执行，而是存在主存中，由 I/O 通道所包含的处理器执行。
- 采用四级结构，**处理器-通道-控制器-设备**

> 带有局部存储器的 I/O 通道

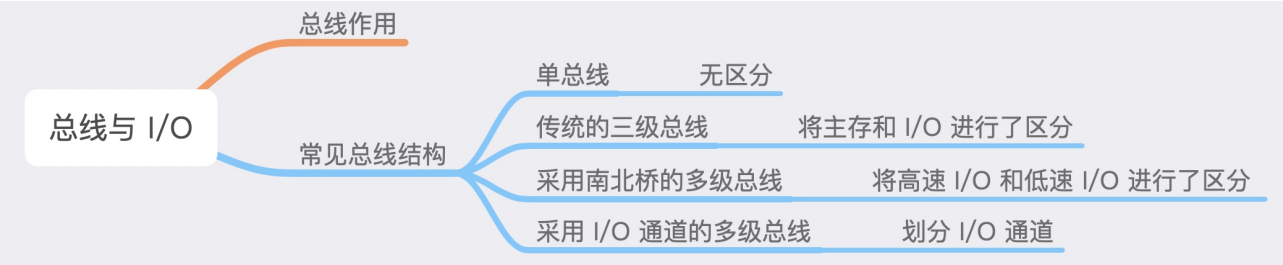
- 相当于一台自治的计算机
  - I/O指令存储在控制器自带的局部存储器中，并由 I/O 通道所包含的处理器执行
- 可以控制大量的 I/O 设备，同时最小化 CPU 的干涉
- 常用于交互式终端通信，负责包括控制终端在内的大部分任务

> 流程

1. CPU 遇到 I/O 请求，启动指定通道
2. 一旦启动成功，通道开始控制 I/O 设备进行操作，CPU 执行其他任务
3. I/O 操作完成后，I/O 通道发出中断，CPU 停止当前工作，转向处理 I/O 操作结束事件

CPU 与通道并行工作。

4.3 总线与 I/O



4.3.1 总线概述

总线目标: 解决 I/O 速度不匹配问题。

总线连接方式不同，会导致计算机性能的巨大差异，从而导致计算机价格的巨大差异。只有合理地组织总线，才能使得处理器和设备充分并行，提高系统效率。

4.3.2 单总线

> 总线结构

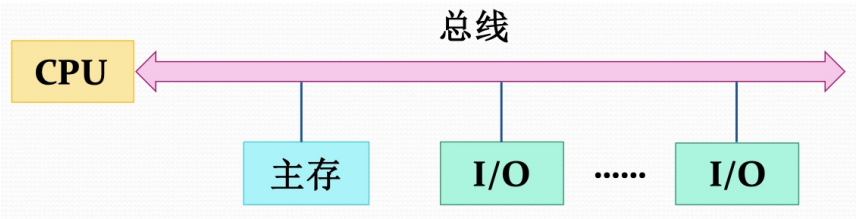
将 CPU、主存和 I/O 模块连接到同一组总线上。

> 优点

结构简单、易于扩充。

> 缺点

1. 主存需要和 I/O 模块共用总线
2. 设备增多会造成总线变长，进而增加传输时延
3. 无法适用于大量高速设备



4.3.3 传统的三级总线

> 总线结构

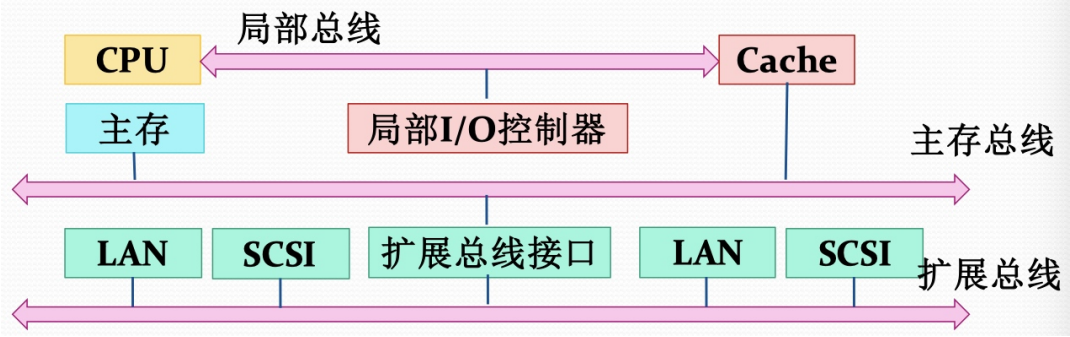
- 主存和 Cache 通过主存总线传送数据，主存总线和扩展总线上的 I/O 设备之间传送数据通过扩展总线接口缓冲

> 优点

1. 主存与 I/O 之间的数据传送与处理器的活动分离
2. 可以支持更多的 I/O 设备

> 缺点

不适用于 I/O 设备数据速率相差太大的情形。



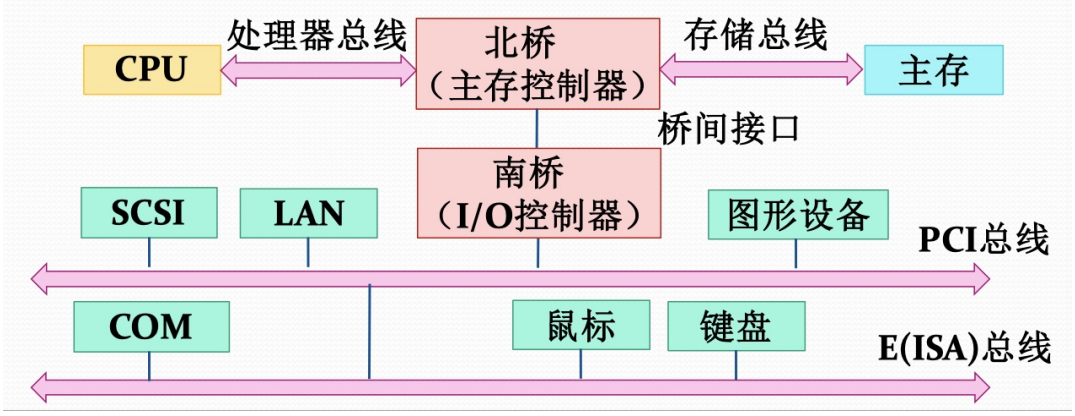
4.3.4 采用南北桥的多级总线

> 总线结构

通过存储总线、PCI 总线、E(ISA) 总线分别连接主存、高速 I/O 设备和低速 I/O 设备。

> 优点

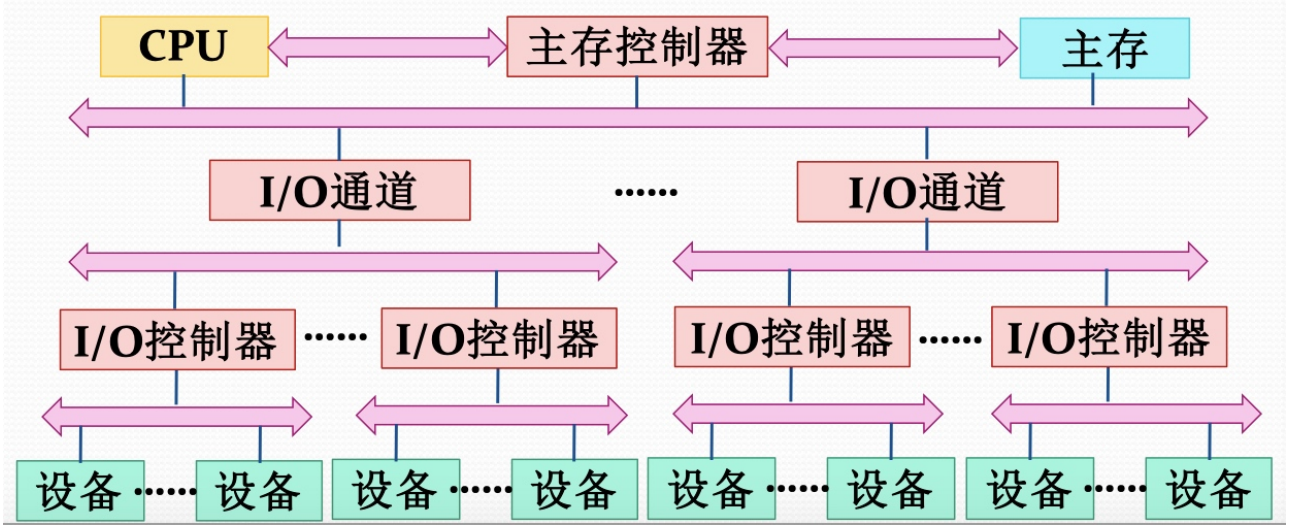
可以支持不同数据速率的 I/O 设备。



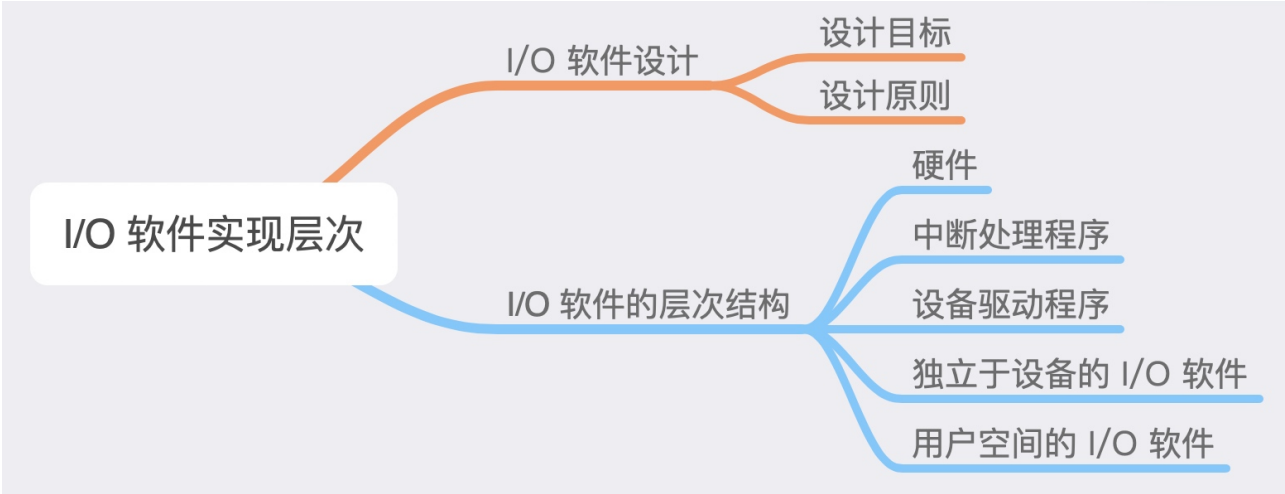
4.3.5 采用 I/O 通道的多级总线

> 总线结构

1. 支持 CPU、主存和多个 I/O 通道之间的数据传送
2. 支持 I/O 通道和 I/O 控制器，以及 I/O 控制器和设备之间的数据传送



4.4 I/O 软件的实现层次



4.4.1 I/O 软件设计

> 设计目标

高效率: 改善设备效率，尤其是磁盘 I/O 操作的效率。

通用性: 用统一的标准来管理所有设备。

> 设计思路

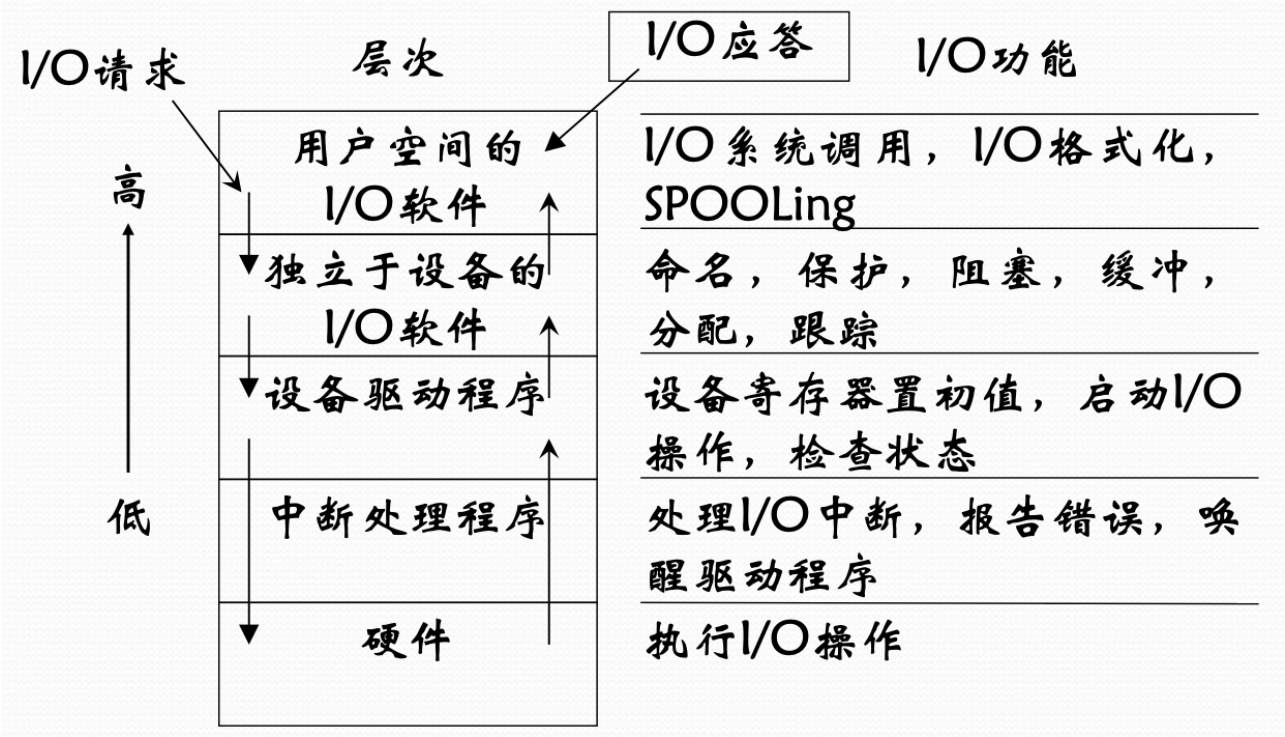
把软件组织成层次结构，低层软件用来屏蔽硬件细节，高层软件向用户提供简洁、友善的界面。

> 设计原则

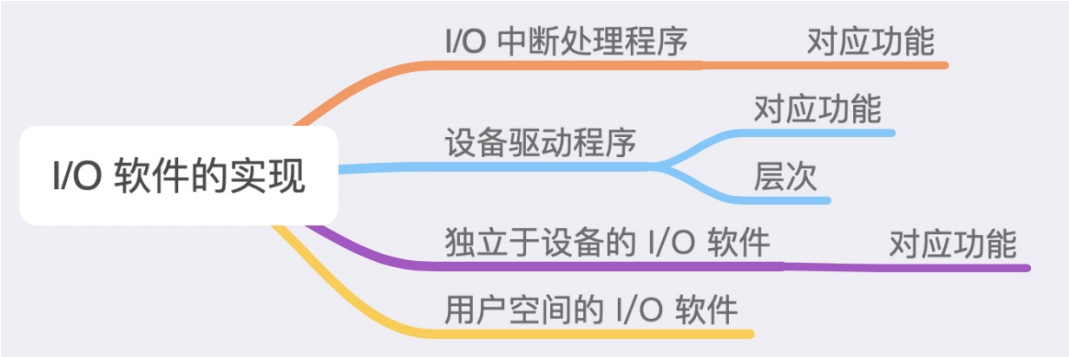
1. 设备无关性: 编写访问文件的程序与具体设备无关
2. 出错处理: 低层软件能处理的错误不让高层软件感知
3. 同步/异步传输: 支持阻塞和中断驱动两种工作方式

4. 缓冲技术: 建立数据缓冲区，提高吞吐率

4.4.2 I/O 软件层次结构



4.5 I/O 软件的实现



4.5.1 I/O 中断处理程序

> 概述

- 位于 OS 底层，与硬件设备密切相关，与系统其余部分尽可能少地发生联系
- 进程请求 I/O 操作时，通常被挂起，直到数据传输结束后并产生 I/O 中断时，操作系统接管 CPU 后转向中断处理程序
- 当设备向 CPU 提出中断请求时，CPU 响应请求并转入中断处理程序

> 功能

检查设备状态寄存器内容，判断产生中断的原因，根据 I/O 操作的完成情况进行相应的处理。

- （错误）如果数据传输有错，向上层软件报告设备的出错信息，实施重新执行
- （正常结束）如果正常结束，唤醒等待传输的进程，使其转换为就绪态
- （未结束）如果有等待传输的 I/O 命令，通知相关软件启动下一个 I/O 请求

4.5.2 设备驱动程序

> 概述

- 设备驱动程序包括与设备密切相关的所有代码
- 从独立于设备的软件中接收并执行 I/O 请求
  - 逻辑 I/O（用户提交）转化为物理 I/O
  - 监督设备是否正确执行，管理数据缓冲区，进行必要的纠错处理

> 设备驱动程序与中断处理程序的区别

中断处理程序: 响应并原子处理 I/O 中断

设备驱动程序: 负责设备中具体的处理细节

> 功能

- 设备初始化
  - 在系统初次启动或设备传输数据时，预置设备和控制器以及通道状态
- 执行设备驱动例程
  - 负责启动设备，进行数据传输
  - 对于具有通道方式的设备，还负责生成通道指令和通道程序，启动通道工作
- 调用和执行中断处理程序
  - 负责处理设备和控制器及通道所发出的各种中断

> 层次

每个设备驱动程序只处理一种设备，或者一类紧密相关的设备。

整体驱动程序

- 直接向 OS 提供接口和控制硬件，适用于功能简单的驱动程序，效率较高，但较难迁移

分层驱动程序

- 将驱动程序分成多层，放在栈中，系统接到 I/O 请求时先调用栈顶的驱动程序，栈顶的驱动程序可以直接处理请求或向下调用更低层的驱动程序，直至请求被处理。
- 适用于功能复杂、重用性要求较高的驱动程序，结构清晰且便于移植，但会增加一部分系统开销。

4.5.3 独立于设备的 I/O 软件

执行适用于所有设备的常用 I/O 功能，并向用户层软件提供一致性接口。

> 功能

1. 设备命名: 通过路径名寻址设备
2. 设备保护: 检查用户是否有权访问所申请设备
3. 提供与设备无关的数据单位: 字符数量，块尺寸
4. 缓冲技术: 传输速率，时间约束，不能直接送达目的地
5. 设备分配和状态跟踪: 分配不同类型的设备
6. 错误处理和报告: 驱动程序无法处理的错误

4.5.4 用户空间的 I/O 软件

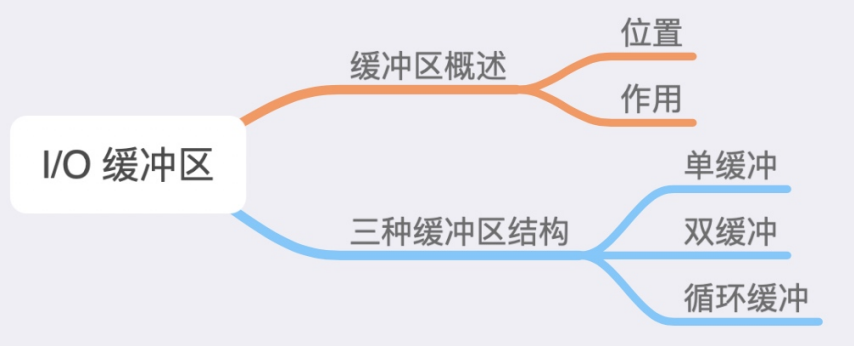
> 库函数

- 一小部分 I/O 软件不在 OS 中，是与应用程序链接在一起的库函数，甚至完全由运行于用户态的程序组成。
- 系统调用通常由库函数封装后供用户使用，封装函数只是将系统调用所用的参数放在合适位置，然后执行访管指令来陷入内核，再由内核函数实现真正的 I/O 操作。

> SPOOLing 软件

在内核外运行的系统 I/O 软件，采用预输入、缓输出和井管理技术，通过创建守护进程和特殊目录解决独占型设备的空占问题。

4.6 I/O 缓冲区



4.6.1 I/O 缓冲区概述

> 提出原因

为了解决 CPU 与设备之间速度不匹配的矛盾，协调逻辑记录大小和物理记录大小不一致的问题（每次读入大小），提高 CPU 和设备并行性，减少 I/O 操作对 CPU 的中断次数，放宽对 CPU 中断响应时间的要求。

> 缓冲区位置

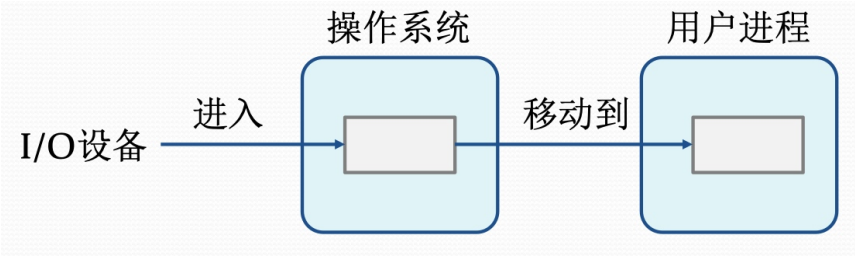
在内存中开辟的存储区，专门用于临时存放 I/O 操作的数据。

> 缓冲区的两个操作

- 写操作: 将数据送至缓冲区，直到装满，进程继续计算，同时系统将缓冲区的内容写到设备上。
- 读操作: 系统将设备上的物理记录读至缓冲区，根据要求将当前所需要的数据从缓冲区中读出并传送给进程。

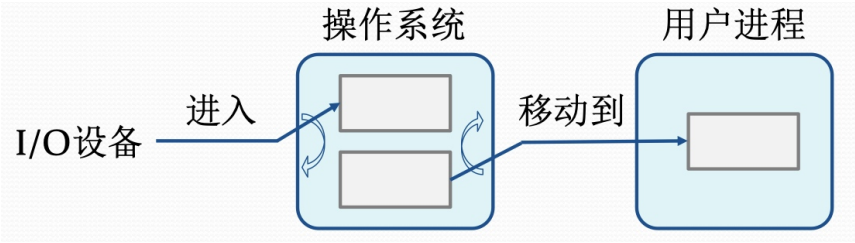
4.6.2 单缓冲区

只有一个缓冲区，I/O设备 → OS，OS → 用户进程，两个过程不能同时进行，效果一般。



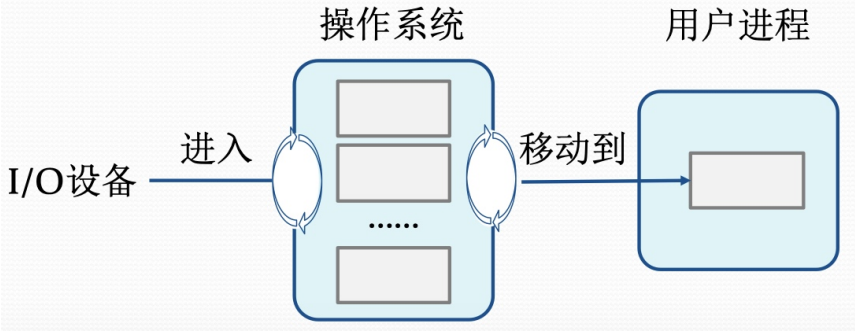
4.6.3 双缓冲区

使用两个缓冲区，I/O 先将数据传给 Buffer1，Buffer1 将数据传给用户的同时，I/O 将数据传给 Buffer2，使得 "I/O设备 → OS"，"OS → 用户" 两个过程可以并行，效率提高。

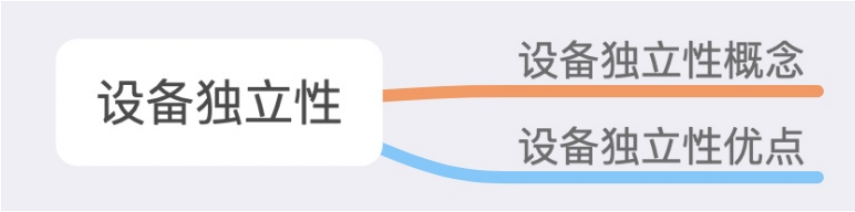


4.6.4 循环缓冲区

循环缓冲区，I/O 给 Buffer2 数据，Buffer1 把数据给用户；I/O 给 Buffer3 数据时，Buffer2 把数据给用户。



4.7 设备独立性



设备独立性的本质是对于 "用户申请物理设备" 这一过程再加了一层封装。

4.7.1 设备独立性概念

用户执行作业前不指定物理设备，而是指定逻辑设备，使得用户作业和物理设备分离开，再通过其它途径建立逻辑设备和物理设备之间的映射。

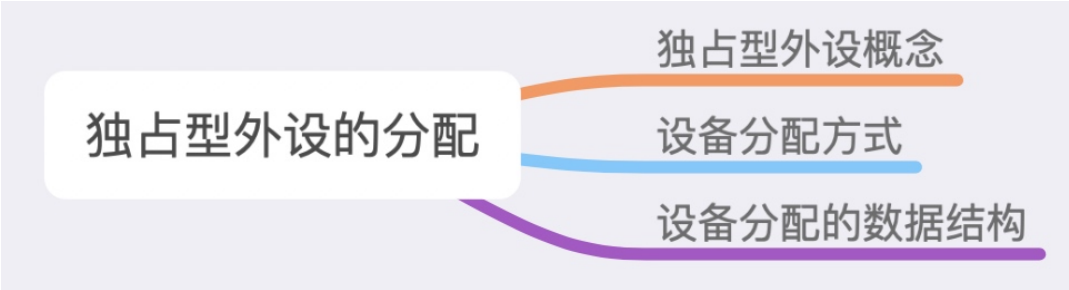
> 条件

设备管理中需要将逻辑设备名转换为物理设备名，因此系统需要提供逻辑设备名和物理设备名的对应表以供转换使用。

4.7.2 设备独立性优点

- 应用程序通用性更强。
  - 应用程序与具体物理设备无关，系统增减或变更设备时不需要修改源程序。
- 易于应对 I/O 设备故障，提高系统可靠性。
- 增加设备分配的灵活性，更有效地利用设备资源，实现多道程序设计。

4.8 独占型外设的分配



4.8.1 独占型外设概念

独占型外设: 只能由一个进程独占式使用

共享设备: 可以让多个进程同时使用。此类设备的管理主要是驱动调度和实施驱动，一般不必分配。

计算机中绝大多数设备属于独占型设备，如键盘、鼠标，而只有磁盘、光盘、磁带库不属于独占设备。

4.8.2 设备分配方式

> 静态分配

静态分配: 进程运行前申请

- 实现简单，能够防止系统发生死锁，但会降低设备利用率。

> 动态分配

动态分配: 进程随用随申请

- 提高设备利用率，但可能发生死锁问题。

例如: 一个系统只有一台磁带机和一台打印机，现在有两个进程 A 和 B 执行时都需要这两台机器，而 A 动态分配到了磁带机，B 动态分配到了打印机，此时就发生了死锁。

4.8.3 设备分配的数据结构

> 设备表

每条设备都有各自的设备表，用来登记这类设备中的每台物理设备。

内容: 物理设备名、逻辑设备名、占有设备的进程号、是否分配、好/坏标志

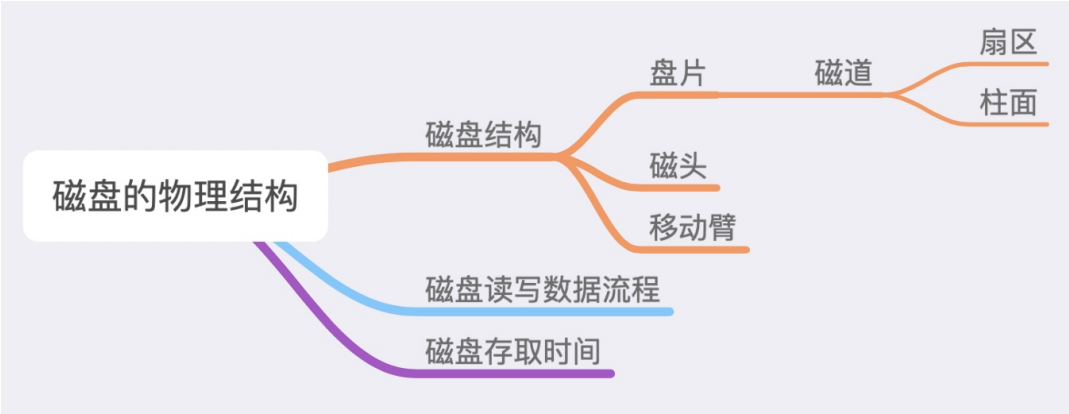
> 设备类表

每类设备对应于设备类表中的一栏。

内容: 设备类、总台数、空闲台数、设备表起始地址

支持设备独立性时才会使用设备类表。

4.9 磁盘的物理结构



4.9.1 磁盘结构

> 盘片

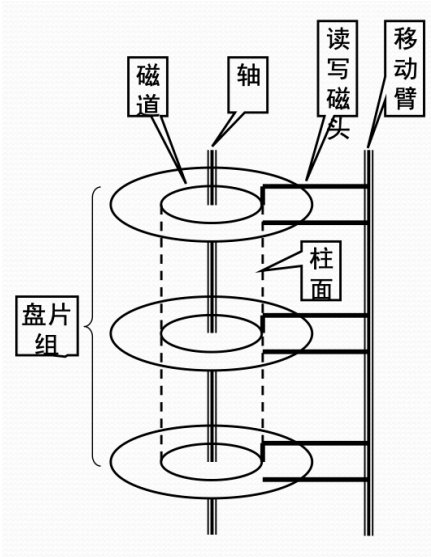
- 磁盘由多个盘片组成
- 每个盘片被划分为多个同心圆结构的磁道
  - 不同盘片上位于相同位置的磁盘构成的圆柱体称为柱面

> 扇区、簇（多个相邻扇区）

每个磁道分为固定多个或不等个数的扇区，为了对大量扇区寻址，OS 将相邻扇区组合成簇存储文件。

> 磁盘物理块地址

- (柱面号, 磁头号, 扇区号): 表示移动臂在哪一个柱面上，磁头定位了哪个盘面，扇区定位了哪一个具体的物理记录。
- (0面0道1扇区): 0盘面，0柱面，1扇区（盘面、柱面从 0 开始编号，扇区从 1 开始编号）



4.9.2 磁盘读取数据流程

磁盘读取数据时，磁头必须定位到指定的磁道上的指定扇区的开始处。

> 具体过程

1. 寻道: 控制移动臂到达指定柱面，选择磁头号
2. 旋转: 等待要读写的扇区旋转到磁头下
3. 数据传送

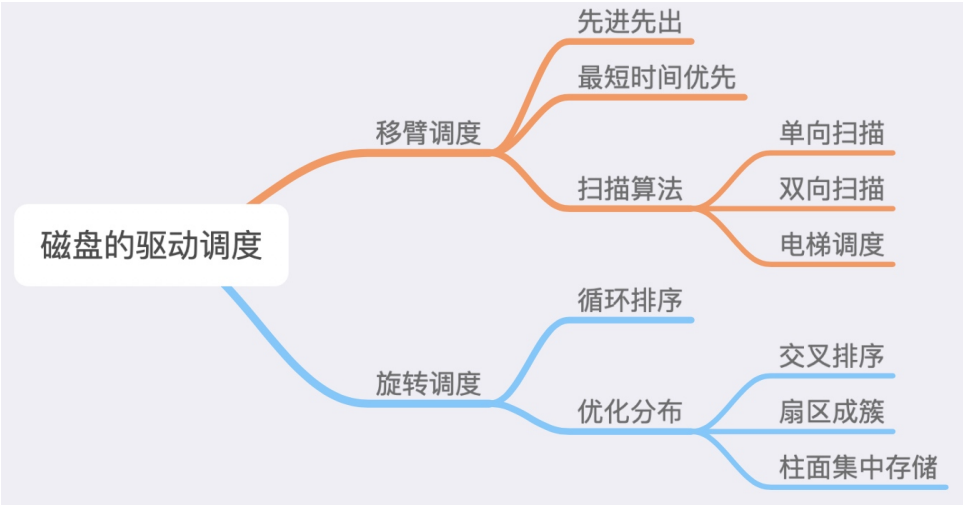
4.9.3 磁盘存取时间

磁盘完成数据读写所需时间 = 寻道时间 + 旋转延迟 + 传送时间

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

T_a : 存取时间
T_s : 寻道时间
r : 磁盘旋转速度（转/秒）
B : 要传送的字节数
N : 一个磁道中的字节数

4.10 磁盘的驱动调度



4.10.1 磁盘调度

> 调度原因

磁盘可能同时接收到若干 I/O 请求，如果随机选择并响应 I/O 请求，可能得到最坏的性能。

> 调度策略

磁盘存取时间主要是寻道与旋转，因此调度策略主要是移臂调度和旋转调度。

4.10.2 移动臂调度

> 目的

使移动臂的移动时间最短，从而减少寻道总时间。

> （1）先进先出

- 移动臂式随机移动，寻道性能较差
- 按顺序处理请求，对所有进程公平

> （2）最短查找时间优先

- 先执行查找时间最短的请求，具有较好的寻道性能
- 存在 “饥饿” 现象

> （3）扫描算法 ———— 单向扫描

- 扫描臂总向一个方面扫描，归途中不提供服务
- 适用于不断有大量柱面均匀分布的请求的情形

> （4）扫描算法 ———— 双向扫描

该算法是对单向扫描的一个改进。

- 移动臂每次向一个方向移动，遇到最近的 I/O 请求便进行处理，到达最后一个柱面后再向相反方向移动
- 对最近扫描所跨越区域的请求响应较慢

> （5）电梯调度

对扫描算法进行一个终版优化。电梯调度算法是解决移臂调度的经典算法。

- 无请求时移动臂停止不动，有请求时按电梯规律移动
- 每次选择沿移动臂的移动方向最近的柱面
- 如果当前移动方向上没有但相反方向有请求时，改变移动方向

4.10.3 旋转调度

> 目的

使得旋转延迟总时间最少。

> （1）循环排序

- 通过优化 I/O 请求排序，在最少旋转圈数内完成位于同一柱面的访问请求。
- 旋转位置测定硬件和多磁头同时读写技术有利于提高旋转调度的效率。

> （2）优化分布

通过优化信息在存储空间的排列方式来减少旋转延迟。

- 交替排序
  - 如果沿着磁道按序对扇区编号，可能由于磁盘转速太快，造成处理当前扇区的数据时，下一个扇区已经跳过，需要再转一圈才能继续读数据。
  - 因此，对扇区编号时会间隔编号，例如 2:1 表示相邻编号之间会间隔 1 个扇区，3:1 表示会间隔 2 个扇区。
- 相邻扇区成簇
  - 把相邻多个扇区集中成簇一起读写
- 柱面集中存储
  - 连续记录数据时，先记录在同一柱面的不同磁道上，然后再更换柱面
  - 减少数据读写时的移臂操作

4.11 SPOOLing 系统

SPOOLing 系统的实质就是用缓冲区实现 "输入预先完成，输出成批输出" 的效果，提高效率。

本质上还是将一个 “输入-执行-输出” 的连续过程切分为三个离散过程，来保证 CPU 与外设的并发性。

4.11.1 虚拟设备

虚拟设备: 使用一类物理设备模拟另一类物理设备的技术。

- 内存卡模拟磁盘
- 块设备模拟字符设备
- 输入输出重定向（文件）

4.11.2 SPOOLing 系统结构

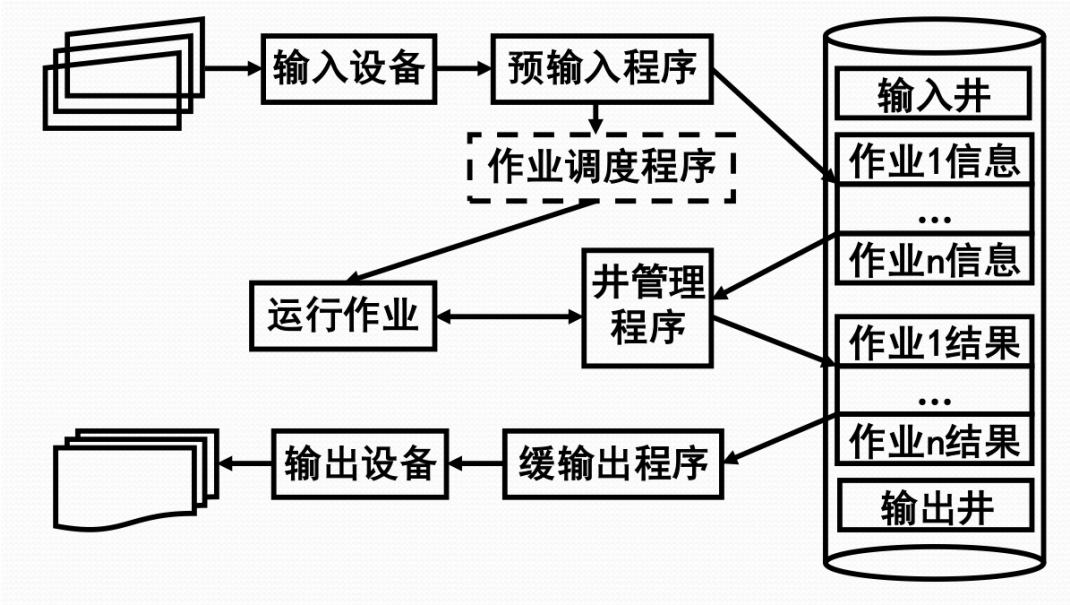
> 五大组成部件

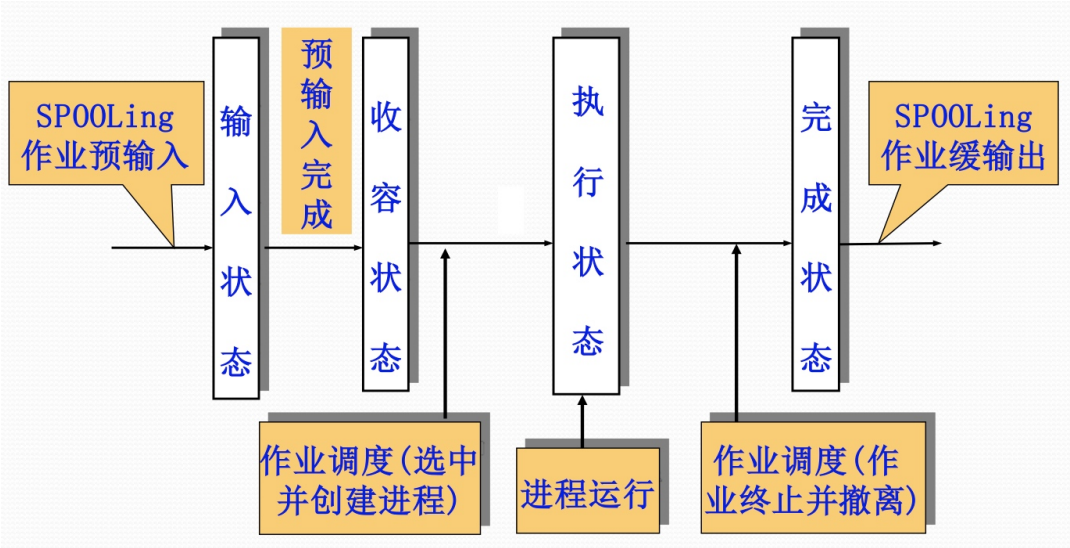
为存放输入、输出数据，系统在磁盘上开辟输入井和输出井，用作缓冲区。

1. 输入井
2. 输出井
3. 预输入程序: 将数据从输入设备传送到磁盘输入井
4. 预输出程序: 将数据从磁盘输出井传送到输出设备
5. 井管理程序: 控制作业和井之间的数据交换（I/O 重定向）

> SPOOLing 结构功能

- 预输入
  - OS 将作业需要的输入数据成批从输入设备上预先输入到磁盘的输入缓冲区中暂存
  - 调度作业执行时，作业使用数据不必再启动输入设备，从磁盘的输入缓冲区读入即可
- 缓输出
  - 作业不启动输出设备，只是将输出数据暂存到磁盘输出缓冲区中
  - 作业执行完毕后，数据由 OS 成批输出
- 优点
  - 设备利用率提高
  - 作业运行时间缩短
  - 每个作业都感觉各自拥有所需的独占设备





4.11.3 打印 SPOOLing 系统

> 打印机空占问题

如果用户进程通过打开打印机的设备文件来申请和使用打印机，往往会造成该进程打开设备文件后长达数小时不用，但其他进程又无法使用打印机。

> 解决方案

- 打印机守护进程是唯一有特权使用打印机设备的进程
- 打印文件前，用户进程先产生完整的待输出文件，并存放在打印目录下
- 打印机空闲时，启动守护进程，打印待输出文件

4.12 批处理系统作业管理

> 多道批处理 OS 的作业管理

- 作业说明语言: 用于作业控制的语言
- 作业说明书: 刻画具体作业的控制方式
- 作业状态
  - 输入状态（预输入）
  - 后备状态（空闲）
  - 运行状态（运行）
  - 完成状态（缓输出）
- 作业调度: 从后备状态作业中选择部分进入运行状态（前提是资源充足）
- 作业调度算法: 优先数、短作业、响应比、设备搭配等

五、设备管理 —— 文件管理

文件和进程是 OS 引入的两个重要概念。

5.1 文件的概念

5.1.1 文件概述

> 概念

- 文件是具有符号名的，在逻辑上具有完整意义的一组相关信息项的序列
- 文件（document）与计算机文件（file）具有一定区别
- 文件名是由字母、数字和其他符号组成的一个字符串，其格式和长度因系统而异

> 文件命名

文件命名一般包括文件名和扩展名。

- 前者用于识别文件，后者用于标识文件特性，两者之间用圆点隔开

每个 OS 都有约定的扩展名。

5.1.2 文件分类

> 按用途分

系统文件、库文件、用户文件

> 按保护级别分

只读文件、读写文件、不保护文件

> 按信息时限分

临时文件、永久文件、档案文件

> 按设备类型分

磁盘文件、磁带文件、光盘文件、软盘文件

5.1.3 文件优点

> 一、用户使用方便

使用者无需记住信息存放在辅存的物理位置，也无需考虑如何将信息存放到存储介质上，只要知道文件名，给出有关 OS 要求便可存取信息，实现了 “按名存取”

> 二、文件安全可靠

由于用户通过文件系统才能实现对文件的访问，而文件系统能提供各种安全、保密和保护措施，故而防止对文件信息的有意或无意的破坏或窃用。

> 三、文件可备份

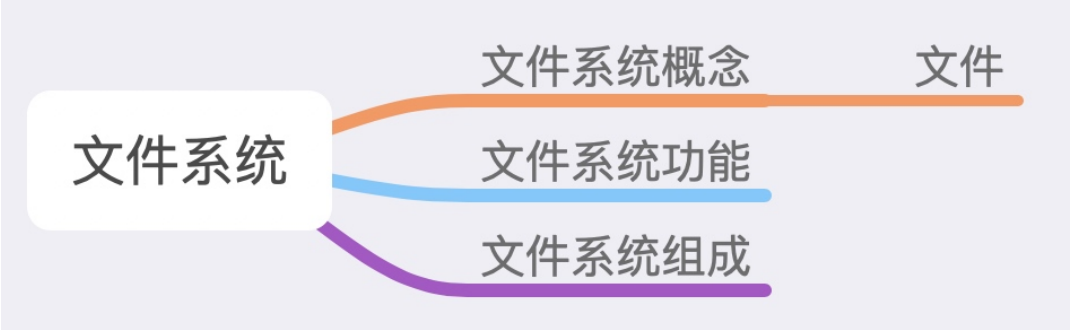
可组织转储或备份，在文件使用过程中出现硬件故障时，文件系统可组织重置，提高可靠性。

> 四、文件可共享

文件系统还能提供文件的共享功能，如不同的用户可以使用同名或异名的同一文件，提高了文件和文件空间的利用率。

因此，把数据组织成文件形式加以管理和控制是计算机数据管理的重大进展。

5.2 文件系统及其功能



5.2.1 文件系统的概念

文件系统是 OS 中负责存取和管理信息的模块，它用统一的方式管理用户和系统信息的存储、检索、更新、共享和保护，并为永辉提供一整套方便有效的文件使用和操作方法。

> 文件系统中的文件

- 文件这一术语不但反映了用户概念中的逻辑结构，而且和存放它的辅助存储器（文件存储器）的存储结构紧密相关
- 因此一个文件必须从逻辑文件和物理文件两个侧面来观察它

5.2.2 文件系统的功能

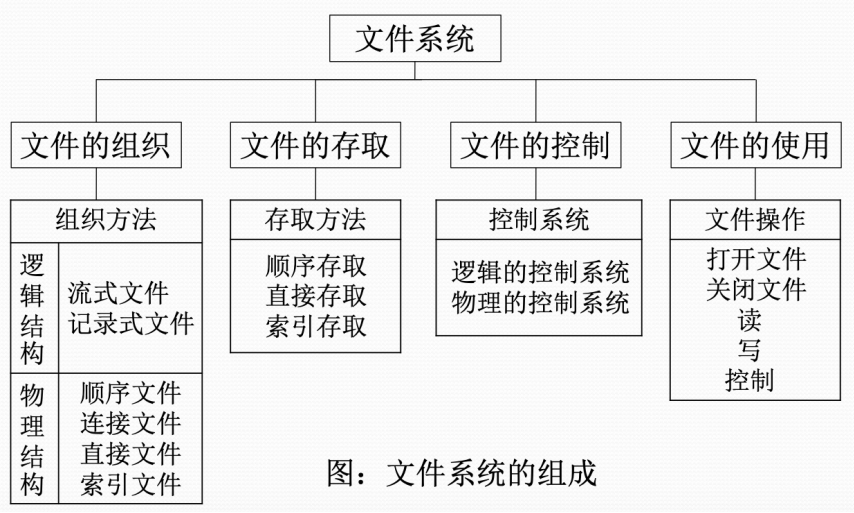
> 面向用户功能

1. 文件的按名存取
2. 文件的共享和保护
3. 文件的操作和使用

> OS 需要考虑的内容 —— 实现文件系统

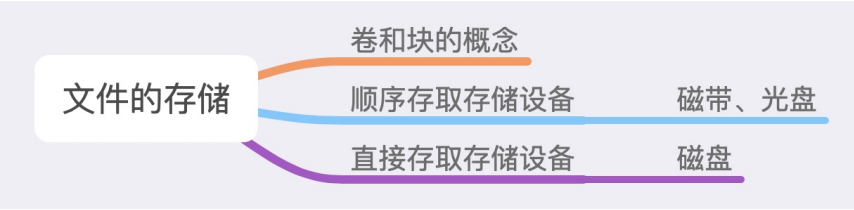
1. 文件目录的建立和维护
2. 存储空间的分配和回收
3. 数据的保密和保护
4. 监督用户存取和修改文件的权限
5. 实现在不同存储介质上信息的表示方式、编址方法、存储次序，以及信息检索等问题

5.2.3 文件系统的组成



图：文件系统的组成

5.3 文件的存储



5.3.1 卷和块

常见的文件存储介质有磁带、光盘和磁盘。

> 卷

卷是存储介质的物理单位，对应于一盘磁带、一块软盘、一个光盘片、一个硬盘分区。

> 块

块是存储介质上连续信息所组成的一个区域，也叫做物理记录。

块在磁盘上被称之为扇区，而在磁带和光盘上则被称之为一个块。

块是主存储器和辅助存储器进行信息交换的物理单位，每次总是交换一块或整数块信息。

> 块大小

- 决定块的大小要考虑用户使用方式、数据传输效率和存储设备类型等多种因素。
- 不同类型的存储介质，块的长短常常各不相同；对同一类型的存储介质，块的大小通常来说都相同，但也可以不同，取决于具体情况。
- 外围设备由于启停机械动作或识别不同块的要求，两个相邻块之间必须留有间隙。
  - 间隙是块之间不记录用户代码信息的区域。

5.3.2 顺序存取存储设备

顺序存取设备是严格依赖信息的物理位置次序进行定位和读写的存储设备。常见的顺序存取设备有磁带、光盘。

> 磁带

磁带机是最常用的一种顺序存取存储设备，它具有存储容量大、稳定可靠、卷可装卸和便于保存等优点，广泛用作存档。

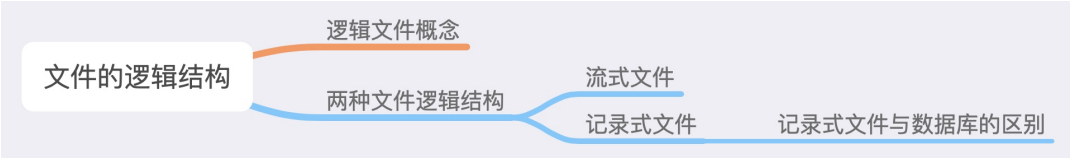
5.3.3 直接存取存储设备

磁盘是一种典型的直接存取存储设备，又叫随机存取存储设备。

> 磁盘

磁盘拥有移臂和旋转两维结构，存取速度高。它的每个物理记录都有确定的位置和唯一的地址，但其存取物理块所需时间随机，而且相差不大，几乎不依赖此信息的位置。

5.4 文件的逻辑结构



5.4.1 逻辑文件概念

> 逻辑文件概念

逻辑文件，又称为文件的逻辑结构。

- 独立于物理环境的，用户概念中的抽象信息组织方式
- 用户能观察到的，并加以处理的数据集合

> 两种文件逻辑结构

- 流式文件
- 记录式文件

5.4.2 流式文件

流式文件指文件内的数据不再组成记录，只是由一串依次的字节组成的信息流序列。

这种文件常常按长度来读取所需信息，也可以用插入的特殊字符作为分界。

注意: 流式文件不具有多维信息结构，只是单纯的连续字符串存取。

5.4.3 记录式文件

记录式文件是一种**有结构**的文件，它是**若干逻辑记录信息所组成**的记录流文件。

逻辑记录是文件中按信息在逻辑上的独立含义所划分的信息单位。

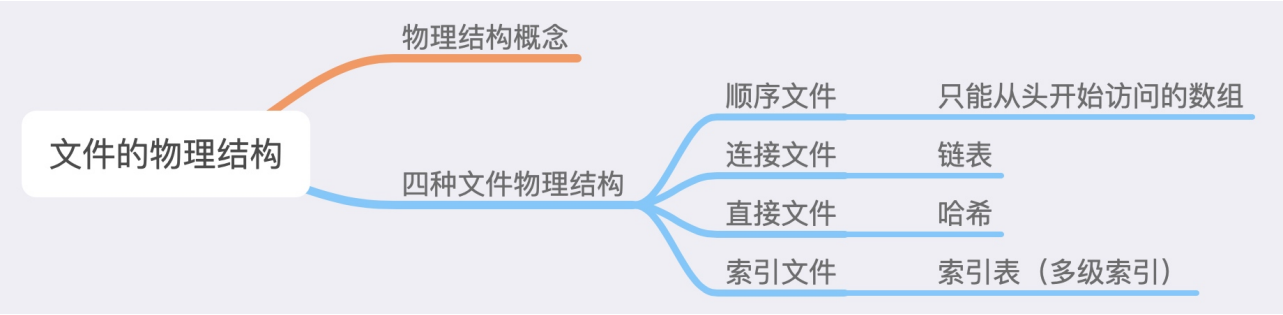
例如，每个职工的工资信息是一个逻辑记录；整个单位职工的工资信息便组成了该单位工资信息的记录式文件。

	数据项1	数据项2	数据项3	数据项4	数据项5			
	序号	姓名	部门	应发工资	扣除工资			
					房租	水	电	
记录1	00001	张三	计算机系	76.80	1.50	0.50	1.00	逻辑文件
记录2	00002	李四	计算机系	105.00	4.70			
记录3	00003	王五	物理系	123.60	5.40			
...	...	...	...	...	...			
记录n	0000n	贺六	经济系	92.00	1.50	0.50	1.00	

> 记录式文件 vs 数据库

数据库管理系统也支持逻辑记录，但数据库与记录式文件最大的区别在于，数据库可以通过一些表与表的数据冗余构成表间联系，并基于这些联系进行数据查询，而文件系统不行。

5.5 文件的物理结构



5.5.1 物理结构概念

文件的物理结构和组织是指文件在物理存储空间中的存放方法和组织关系，又称为物理文件。

文件的存储结构涉及块的划分、记录的排列、索引的组织、信息的搜索等问题。

文件物理结构的优劣直接影响文件系统的性能。

5.5.2 顺序文件

> 物理结构

类似于一个只能从头开始访问的数组。

将一个文件中逻辑上连续的信息存放到存储介质的依次相邻的块中，构成顺序结构，这类文件叫顺序文件，也称为连续文件。

顺序文件通常存储在同一个柱面上，减少移臂开销。

磁带文件、光盘文件是顺序文件的典例。

> 优点

顺序存取记录时速度较快，常用于批处理文件、系统文件。

采用磁带存放顺序文件时，总可以保持快速存取的优点。

> 缺点

建立文件前需要能预先确定文件长度，以便分配存储空间，修改、插入和增加文件均具有困难。

5.5.3 连接文件

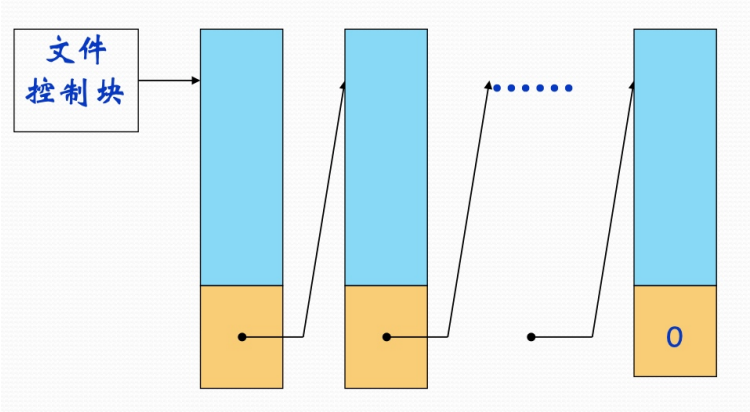
> 物理结构

连接文件是对于顺序文件存储空间连续的一种修改，**采用链表方式**使得存储空间不连续，但访问顺序依然连续。

连接文件，又称串联文件，使用连接字来表示文件中各个物理块之间的先后次序。

第一块文件信息的物理地址由文件目录给出，而每一块的连接字指出了文件的下一个物理块位置。连接字内容为 0 时，表示文件至本块结束。

输入井、输出井都用此类文件。



> 优点

1. 易于对文件记录做增、删、改，易于动态增长记录
2. 不必预先确知文件长度
3. 存储空间利用率高

> 缺点

1. 存放指针需额外的存储空间（但浪费空间少）
2. 连接文件仅适用于顺序存取。找第100个文件必须经过前99个文件，对于顺序存取文件可以接受，但对于随机存取文件则不可接受。

5.5.4 直接文件

采用了哈希方法进行文件映射，主要难点在于哈希冲突。

直接文件，又称散列文件，通过计算记录的关键字建立与其物理存储地址之间的对应关系。

> 哈希冲突

计算寻址结构可能出现 “冲突”，即不同关键字映射相同地址，常见解决方法有拉链法、循环探查法、二次散列法、溢出区法。

5.5.5 索引文件

> 物理结构

索引文件为每个文件建立了一张索引表，其中，每个表目包含一个记录的键（逻辑记录号）及其存储地址。

索引表的地址可由文件目录指出，查阅索引表先找到相应记录键，然后获得数据存储地址。

> 访问方式

索引文件在文件存储器上分为两个区: 索引区、数据区。

1. 查找索引表
2. 获得记录物理地址

上述过程需两次访问辅存，若文件索引已预先调入主存，则仅需一次内外存信息交换。

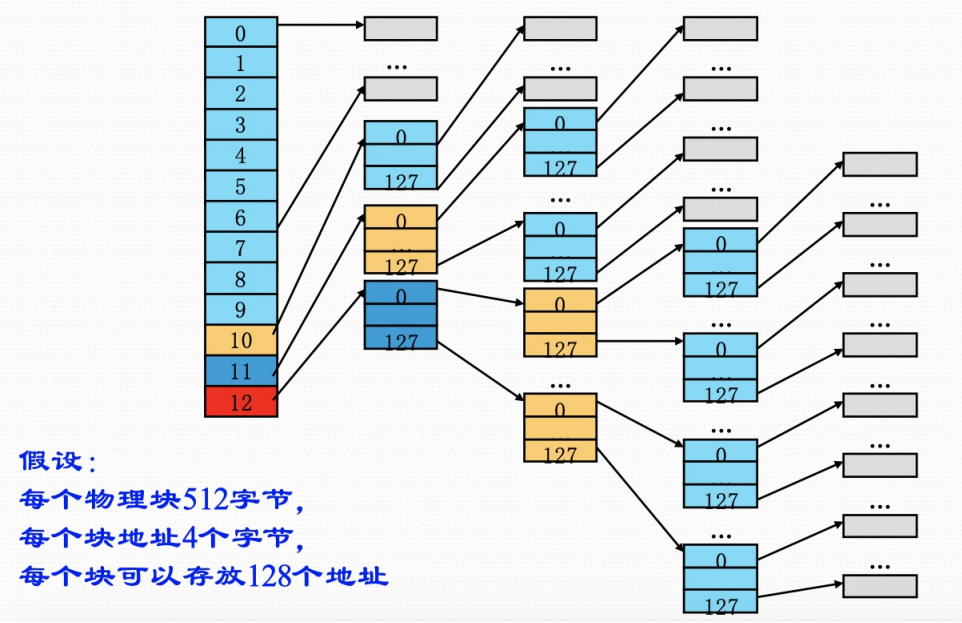
> 优点

索引结构可以被认为是连接结构的一种扩展，除了具备连接文件的优点外，还克服了顺序存取的缺点，具有直接读写任何一个记录的能力，便于文件的增、删、改。

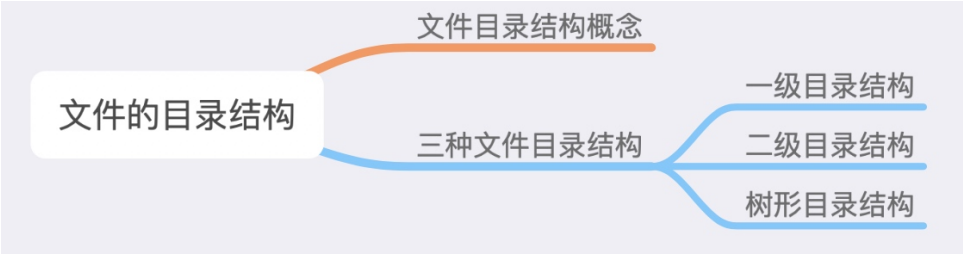
> 缺点

增加了索引表的时空开销。

> 多级索引结构



5.6 文件的目录结构



5.6.1 文件目录

> 文件目录概念

- 文件目录是实现文件的 “按名存取” 的关键数据结构
- 文件系统的基本功能之一就是负责文件目录的建立、维护和检索，要求编排的目录便于查找、防止冲突
- 文件目录需要永久保存，因此也组织成文件存放在磁盘上，称**目录文件**

> 文件目录与索引文件的区别

文件目录是面向用户设计的数据结构，目的是为了让用户可以 “按名存取“ 文件。

索引文件是 OS 用来寻找文件物理位置的数据结构，用户不可见。

5.6.2 一级目录结构

在 OS 中构造一张线性表，与每个文件的相关属性占用一个目录项，构成了一级目录结构。

但由于用户与文件众多，不但容易重名，而且用户难以查找。

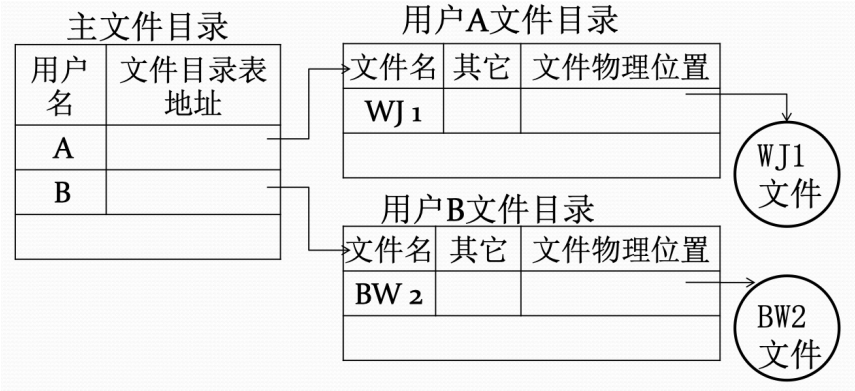
5.6.3 二级目录结构

> 目录结构

第一级是主文件目录，管理所有用户文件目录。目录项登记了系统接受的用户的名字，以及该用户文件目录的地址。

第二级为用户的文件目录，为该用户的每个文件保存一个登记栏，其内容与一级目录的目录项相同。

每一用户只允许查看自己的文件目录。



> 优点

1. 支持检查访问文件者的存取权限，实现了文件的保密和保护。因为任何文件的存取都必须通过主文件目录。
2. 允许不同用户出现同名文件。

5.6.4 树形目录结构

> 目录结构

每一级目录可以登记下一级目录，也可以登记文件，从而形成了层次文件目录结构。

层次目录结构通常采用树形目录结构，树根是根目录，树叶是文件。

> 优点

1. 较好地反映现实世界中具有层次关系的数据集合和较确切地反映系统内部文件的组织结构
2. 不位于同一末端子目录的不同文件可以重名
3. 易于规定不同层次或子树中文件的不同存取权限，便于文件的保护、保密和共享

> 文件定位

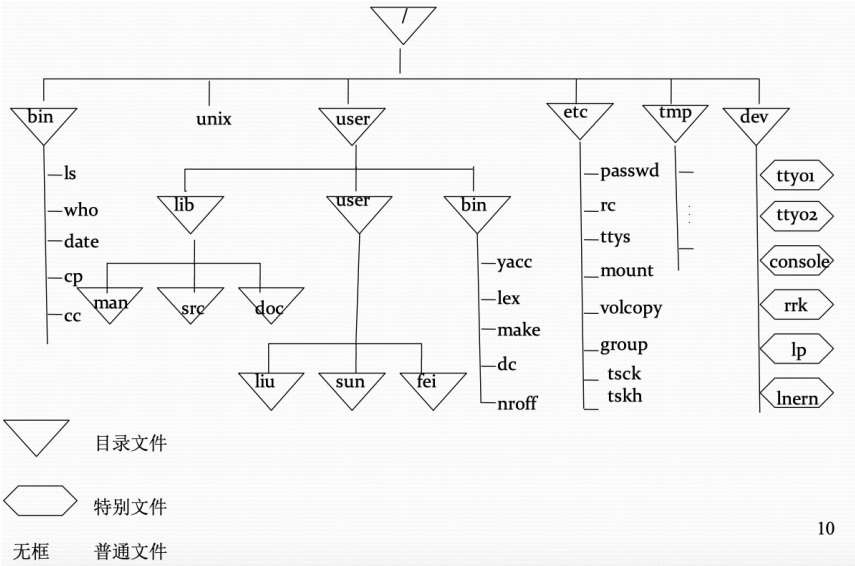
在树形目录结构中，一个文件的全名包括从根目录开始到文件为止，通路上遇到的所有子目录路径，又称为路径名。

各子目录名之间用斜线隔开。（UNIX 用正斜线 “/”，Windows 用反斜线 “\”）

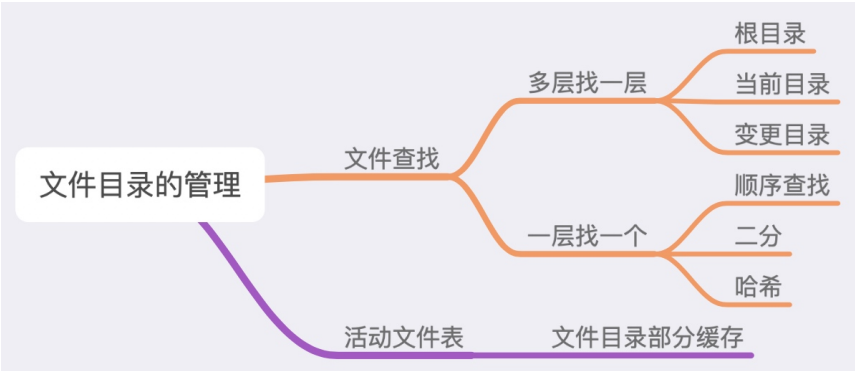
一个硬盘分区可以组织成一颗子树。

- 每棵子树可以对应于一个逻辑盘符（Win）
- 也可以把多棵子树合并成一颗大树（UNIX）

> 举例



5.7 文件目录的管理



5.7.1 文件查找

文件查找分为两步，（1）定位层次，（2）定位具体目录项

> 文件层次定位

根据用户提供的文件路径名来搜索各级文件目录，找到文件。

1. 从根目录查起（绝对路径名）
2. 从当前目录查起（相对路径名）
    - . 表示当前目录
    - .. 表示父目录
3. 变更当前工作目录（现代 OS 设置有改变工作命令）

> 目录项查找

1. 顺序查找法
2. 二分查找法
    - 将按键进行顺序编排，即可进行二分

3. 哈希法
    - 将每个文件名经过变换函数变换成唯一的目录表表项

5.7.2 活动文件表

> 树型目录结构存在问题

当一个文件经过许多目录节点时，系统需要多次访问文件存储器，文件查找非常耗时。

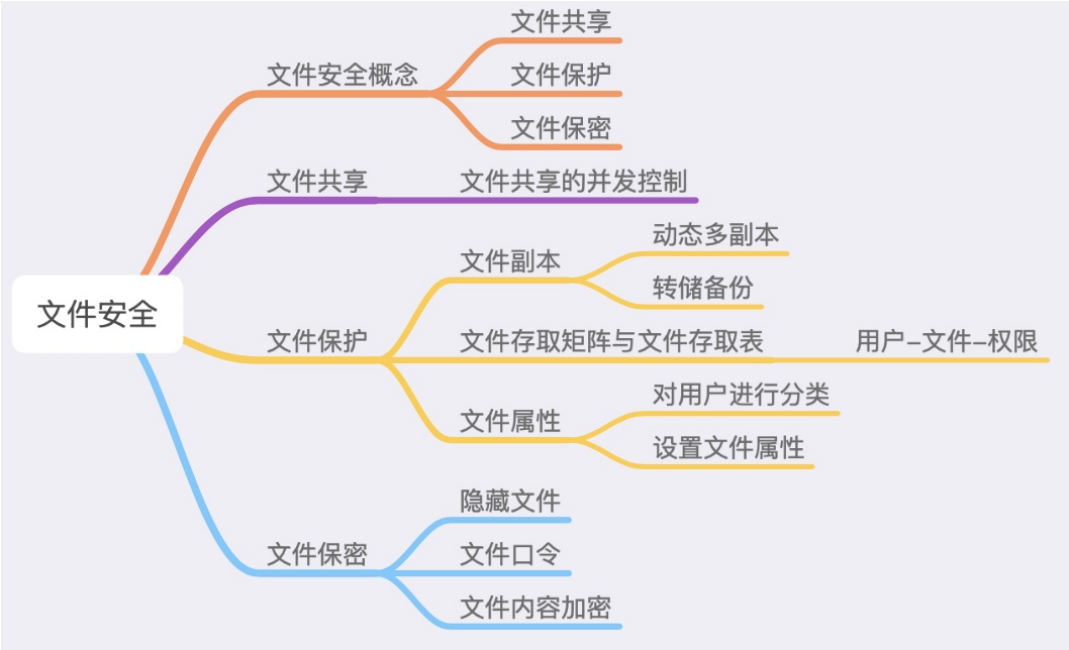
> 解决方法

采用 Cache 的方法，将部分常用或正在用的文件目录复制进主存，实现部分缓存的目的。

> 活动文件表

- 系统为每个用户进程建立一张活动文件表，当用户使用一个文件之前，先通过 “打开” 操作，把该文件有关目录信息复制到指定主存区域，有关信息填入活动文件表，以建立用户进程和该文件索引的联系。
- 当不再使用该文件时，使用 “关闭”，切断用户进程和这个文件的联系，同时，若该目录已被修改过，则应更新辅存中对应的文件目录。

5.8 文件安全



5.8.1 文件安全概念

文件是计算机系统的重要资源，因此要求文件系统具有保障文件安全的手段，提供文件保密的措施，有效地实现文件共享。

> 文件安全三大内容

- 文件共享: 不同用户共同使用某些文件
- 文件保护: 防止文件被破坏
- 文件保密: 防止文件及其内容被其他用户窃取

5.8.2 文件共享

> 文件共享概念

文件共享是计算机用户完成共同任务所必需的。

文件共享有许多好处:

1. 减少用户大量重复性劳动
2. 免除系统复制文件的工作
3. 节省文件占用的存储空间
4. 减少程序设计输入输出文件的次数

> 文件共享的并发控制

OS 应该提供手段实现对共享文件的同步控制。

- 同时读: OS 对文件进行公用控制
- 一读一写: OS 提供同步控制机制，保证文件数据的完整性
- 同时写: 不被允许

5.8.3 文件保护

5.8.3.1 文件保护概念

文件保护是指防止文件被破坏。OS 必须提供文件保护机制，有效实现文件的完整性。

> 常用的文件保护方法

1. 文件副本
2. 文件存取矩阵与文件存取表（稀疏矩阵）
3. 文件属性

5.8.3.2 文件副本

文件副本即存储多个文件副本，保证即使硬软件故障，信息仍能完整保存。

> 动态多副本技术

在多个介质上维持同一内容的文件，并且同时更新内容。

这种方法需要增加设备费用和系统负载。一般适用于容量较小且较为重要的文件，例如不需要更新的系统文件及专用文件，当文件发生故障时只要切换到备用设备即可。

这种方法是精细活，执行起来非常累。

> 文件转储（大多数 OS 采用）

文件转储: 定时把文件复制转储到其它介质上，当某介质上出现故障时，复原转储文件。

转储分为两种方式:

1. 在一定时间间隔或一个单位处理结束时，系统自动复写更新过的文件和数据。
    - 只能针对少量的关键数据
2. 每天或每周把文件信息全部复写一遍，需要时再通过装入转储文件来恢复系统，诸如 BACKUP、RESTORE 等命令
    - 通常需要系统管理员手动备份

5.8.3.3 文件的存取控制矩阵 / 表

> 存取控制矩阵

OS 为每个用户设置访问每个文件对象的存取属性，即 (用户-文件-权限) 三元组的关系。

所有三元组就组成了一个二维矩阵，称为存取控制矩阵。

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ & \vdots & & \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}$$

> 存取控制表

由于操作系统拥有很多用户和众多文件，因此存取控制矩阵是一个稀疏矩阵，存在大量无效位置。

因此我们将无效位置剔除，仅登记那些对文件拥有存取属性的部分，将二维矩阵优化成一维线性表，即存取控制表。

> 基于存取控制矩阵/表的文件保护

- 存取属性: 访问、读、写、执行、创建、删除、授权等
    - 一些非常安全的 OS 甚至设计了 20 多种属性
- 系统通过查阅矩阵/表来核对用户对文件的存取权限
- 文件属主使用 GRANT、REVOKE 等命令进行授权，甚至授权授权权
- 系统管理用户（超级用户）拥有所有文件的属主权限

5.8.3.4 文件属性（广泛采用）

该方法即是将原来的三位结构简化为 (用户-文件) 二维结构，大为简化文件管理。

> 具体方式

- 用户分类: 属主、合作者、其他
- 文件属性: 读、写、执行
- 每个用户对于每个文件都有三种身份，不同身份对于不同文件属性权限不同
- 文件属性可以放在文件目录项中，简化管理
- 用户使用文件时，根据该用户针对该文件的用户类别，以及文件属性，核对用户权限，实现保护

	读	写	执行
文件主	1	1	0
伙伴	1	0	0
其它用户	1	0	0

- chmod 命令可以改变文件属性
- chown 命令可以变更文件属主
- chgrp 命令可以变更用户伙伴

5.8.4 文件保密

文件保密是指文件及其内容不能被未经文件主授权的其他用户窃取。

> 文件保密常见措施

1. 隐藏文件目录
   - 将关键文件设置为隐藏文件（保密程度弱）
2. 设置口令
   - 在文件层面设置 读/写 打开口令
   - 这种方式较第一种有升级，但拦不住那些直接访问磁盘物理数据的恶意用户
3. 使用密码
   - 对文件内容进行加密，只有拿到密钥才能解密（保密程度强）

5.9 文件的存取方法

文件系统的一个重要目标: 尽可能地提供不同的存取方法

- 文件存取方法是 OS 为用户程序提供的使用文件的技术和手段
- 文件存取方法在某种程度上依赖于文件的物理结构

5.9.1 顺序存取

> 概念

按记录顺序进行读 / 写操作的存取方法称顺序存取。

- 读操作根据**读指针**读出当前记录，同时推进读指针，指向下一次要读出的记录
- 写操作设置**写指针**，把一个记录写到文件末端，同时推进写指针
- 允许对读指针进行前跳或后退 n 个记录的操作

> 适用范围

顺序存取方式对任何物理结构文件均适用。

5.9.2 直接存取

支持直接文件、索引文件的物理结构能进行直接存取。直接存取针对那些要求快速地以任意次序直接读写某个记录的应用场合。

例如，航空订票系统，把航班号作标识，把特定航班的所有信息存放在物理块中，用户预订某航班时，直接计算出该航班的存取位置。

5.9.3 索引存取

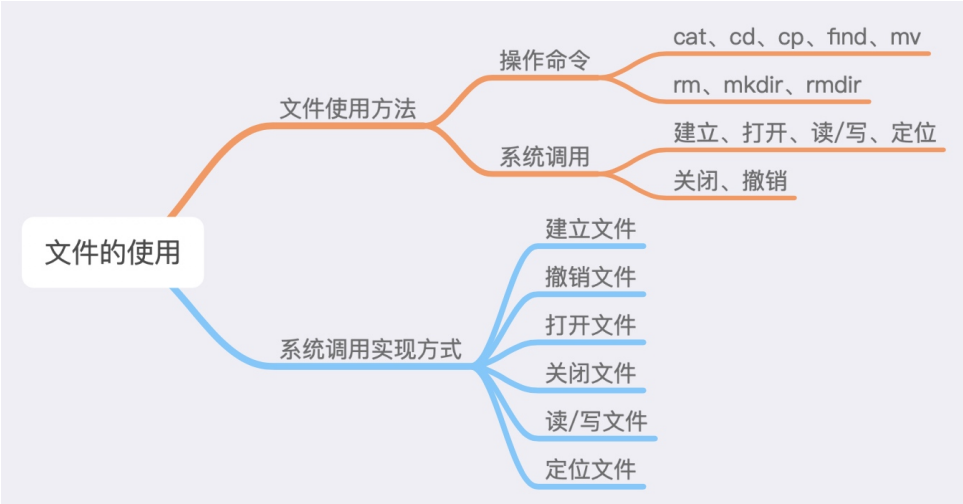
基于索引文件的物理结构，才能实现索引存取。其中信息块的地址都可以通过查找记录键而换算出。

> 特点

索引文件，支持顺序、直接、索引三种存取方式。

实际的系统中，大都采用多级索引，来加速记录查找过程。

5.10 文件的使用



5.10.1 文件使用方法

用户通过两类接口与文件系统联系。

1. 与文件有关的操作命令
    - UNIX 中的 cat、cd、cp、find、mv、rm、mkdir、rmdir
2. 提供给用户程序使用的文件类系统调用
    - 建立、打开、读/写、定位、关闭、撤销

5.10.2 建立文件

功能: 创建一个文件

所需参数: 文件名、设备类（号）、文件属性、存取控制信息

> 处理流程

1. 在相应设备上建立一个文件目录项，并分配第一个物理块
2. 在活动文件表中申请一个项，登记有关目录信息
3. 返回一个文件句柄

5.10.3 撤销文件

功能: 删除一个文件

所需参数: 文件名和设备类（号）

> 处理流程

1. 若文件没有关闭，关闭文件
2. 若为共享文件，进行联访处理
3. 在目录文件中删去相应目录项
4. 释放文件占用的文件存储空间

5.10.4 关闭文件

功能: 结束一个文件的读写

所需参数: 文件句柄

> 处理流程

1. 将活动文件表中该文件的 “当前使用用户数” 减 1
2. 若此值为 0，则收回此活动文件表
3. 完成 “推迟写”
4. 若活动文件表目内容已被改过，应先将表目内容写回文件存储器中相应表目位置，以使文件目录保持最新状态

5.10.5 读/写文件

功能: 读写文件

所需参数: 文件句柄、用户数据区地址、读写的记录或字节个数

> 处理流程

1. 按文件句柄从活动文件表中找到该文件的目录项信息
2. 根据目录项指出的该文件逻辑和物理组织结构，把相关逻辑记录转换成物理块

5.10.6 定位文件

功能: 调整所打开文件的读写指针位置

所需参数: 文件句柄、定位指针

5.11 辅存空间管理

随着用户文件不断建立和撤销，文件存储空间会出现许多 “碎片”。

OS 解决 “碎片” 的办法是整理 “碎片”，在整理过程中，往往对文件重新组织，让其存放在连续存储区中。

5.11.1 辅存空间分配方式

> 连续分配

特点: 存放在辅存空间连续存储区中（连续的物理块号）

优点: 顺序访问时速度快，管理较为简单，但为了获得足够大的连续存储区，需定时进行 “碎片” 整理。

> 非连续分配

特点: 动态分配给若干扇区或簇（几个连续扇区），不要求连续

优点: 辅存空间管理效率高，便于文件动态增长和收缩

5.11.2 空闲块的管理

> 位示图

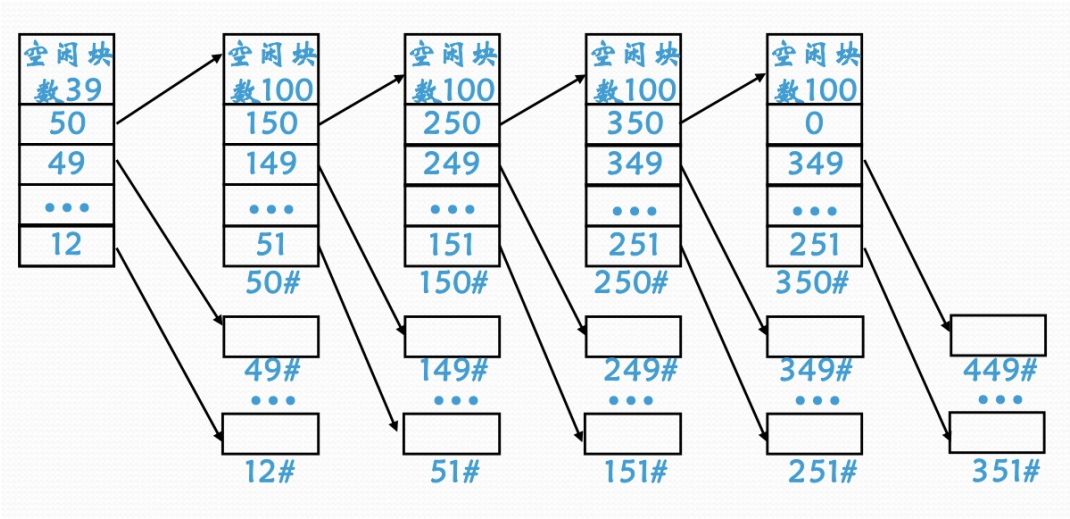
方式: 使用若干字节构成一张表，表中每一字位对应一个物理块，字位的次序与块的相对次序一致。字位为 “1” 表示相应块已占用，字位为 “0” 表示该块空闲。

优点: 把位示图全部或大部分保存在主存中，再配合位操作，实现高速物理块分配和去配。

> 成组链接法

从下图中可以看到，最左边的是空闲盘块号栈，12 为栈顶，50 为栈底。栈底块链接到右边的 100 个空闲块，可以看到最后一个栈底为 0，表示空闲盘块链的结束。

注意: 最左边的空闲盘块栈中的块数始终保持在 $[1, 100]$ 之间。



下图算法中，空闲块数即空闲盘块栈中记录的空闲块数。

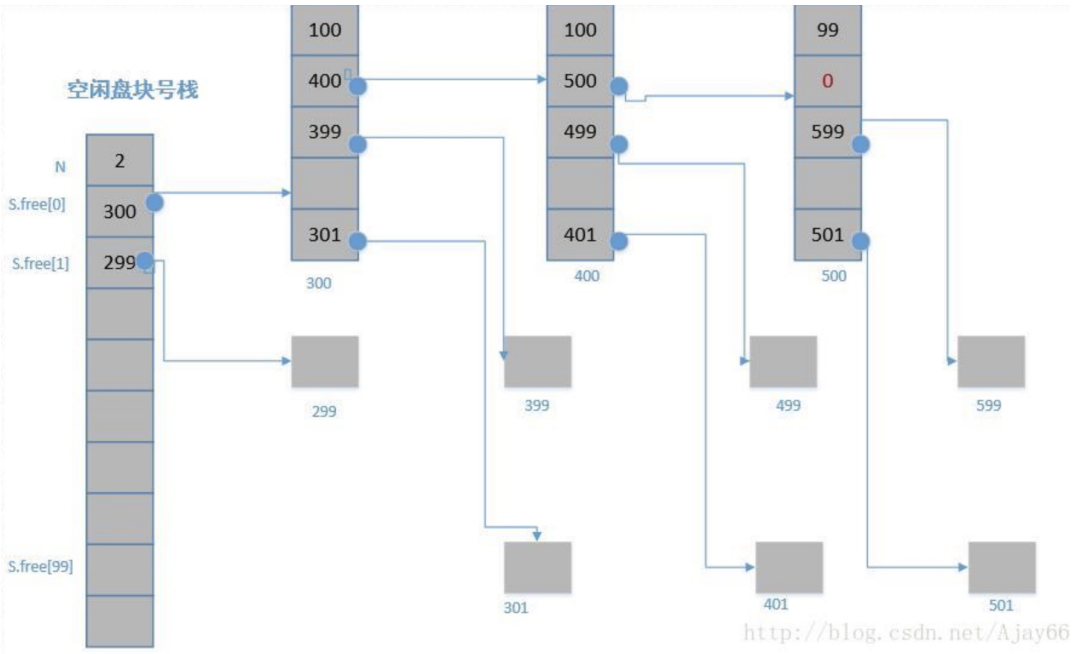
复制第 1 个单元对应块到专用块，表示一次性将 [51,150] 都复制进来，并分配 50。

如果归还时空闲块数超过 100，则留下第 101 个，移走前 100 个，并将第 101 个链接到移走

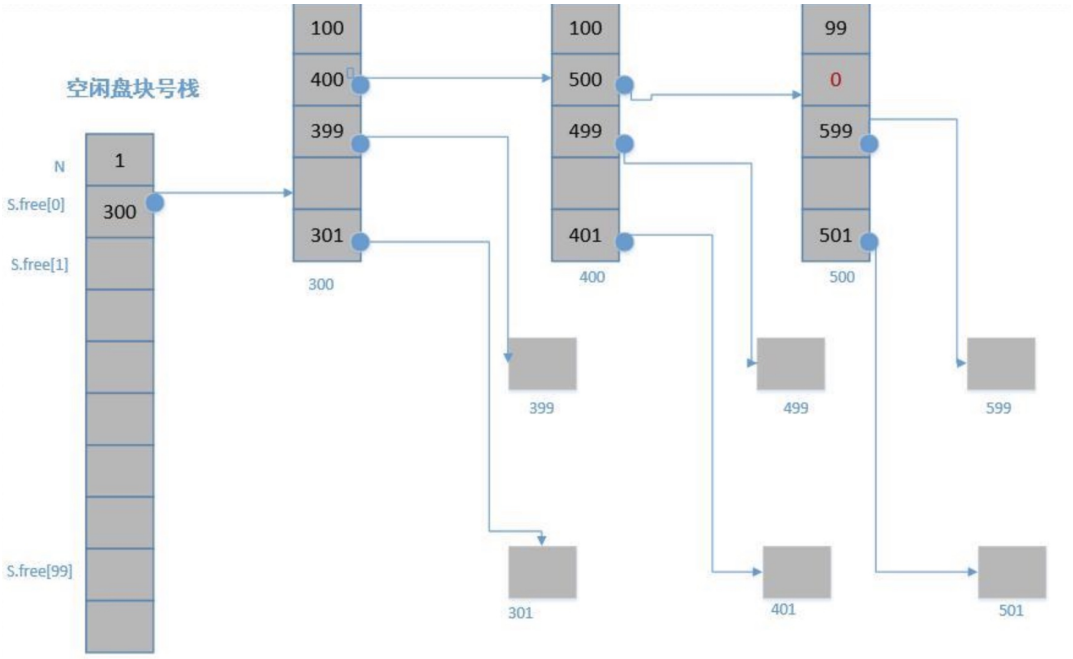
的 100 个上。

(磁盘)专用块←→(主存)专用块	
分配算法 IF 空闲块数=1 THEN IF 第1个单元=0 THEN等待 ELSE 复制第1个单元 对应块到专用块 并分配之 ELSE 分配第(空闲块数) 个单元对应块, 空闲块数减1	归还算法 IF 空闲块数<100 THEN 专用块的空闲块数加1, 第(空闲块数)个单元 置归还块号 ELSE 复制专用块到归还 块,专用块的空闲块 数置1,第1单元置归 还块号

下面举一个例子，下图为初态。



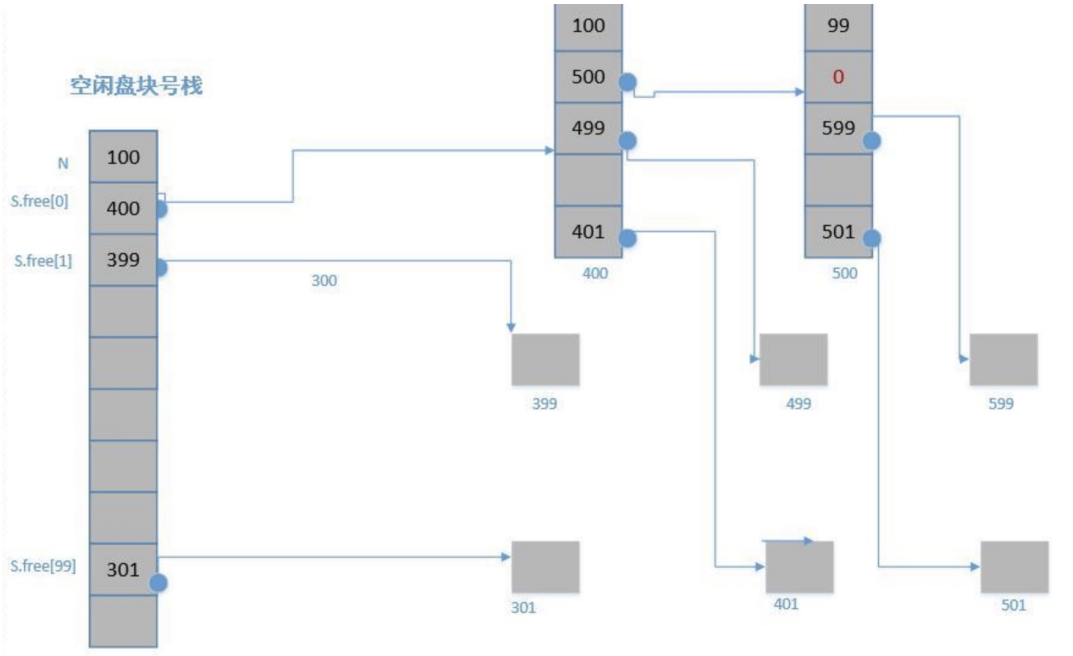
分配 299 号盘块。



分配299号盘块以后的磁盘情况

blog.csdn.net/Ajay6

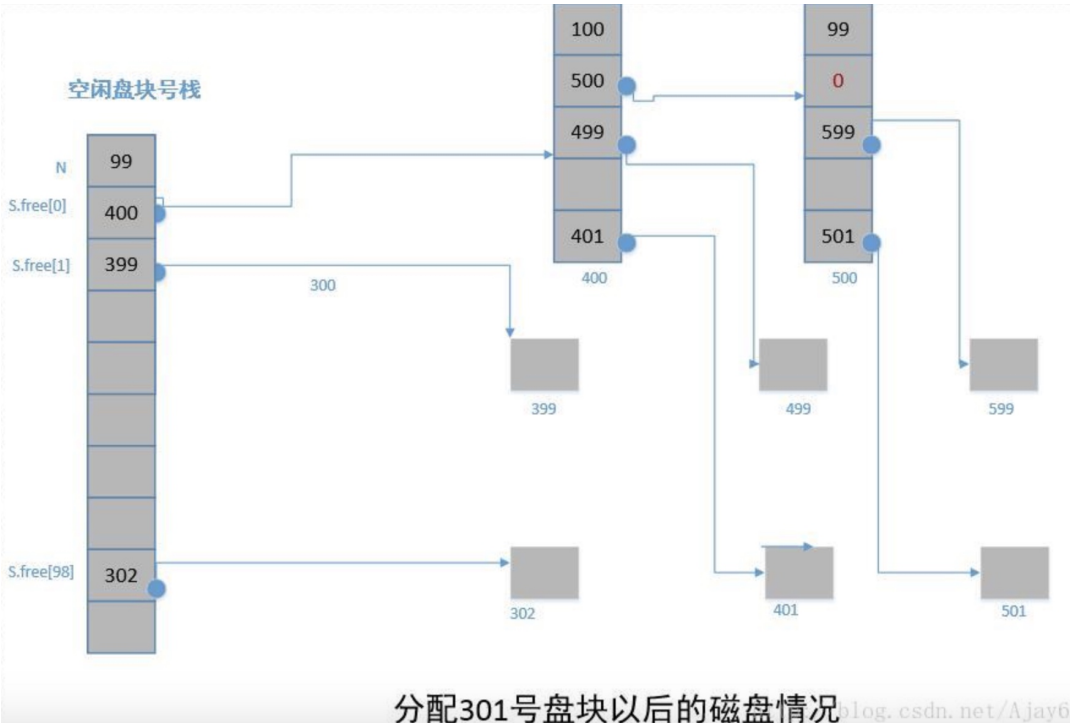
分配 300 号盘块。



分配300号盘块以后的磁盘情况

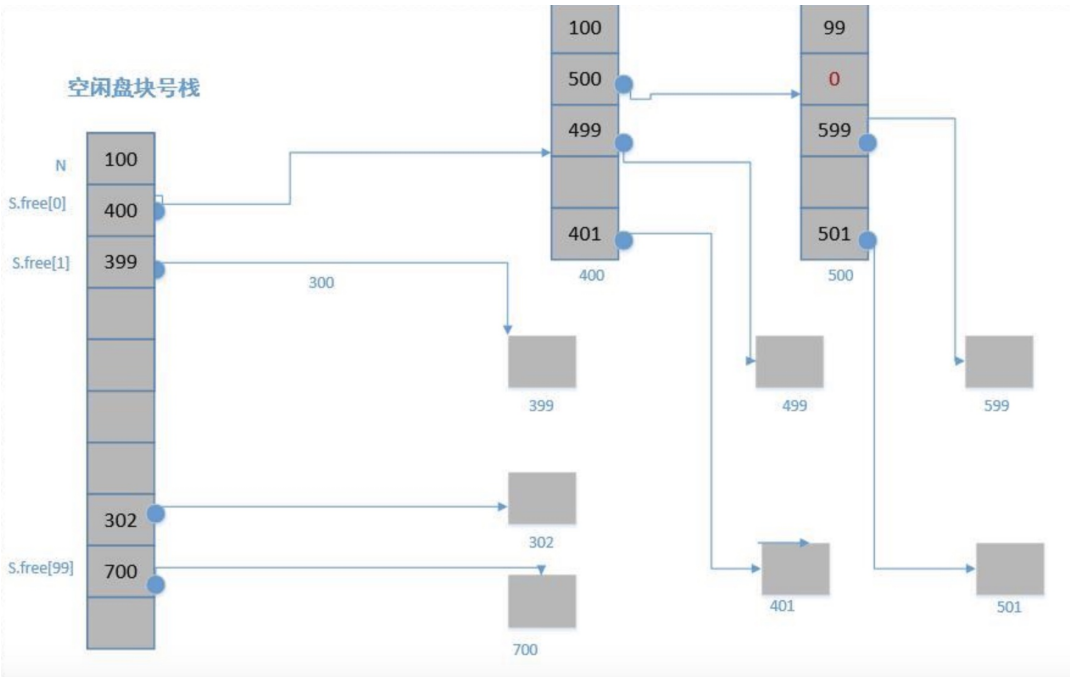
blog.csdn.net/Ajay6

分配 301 号盘块。

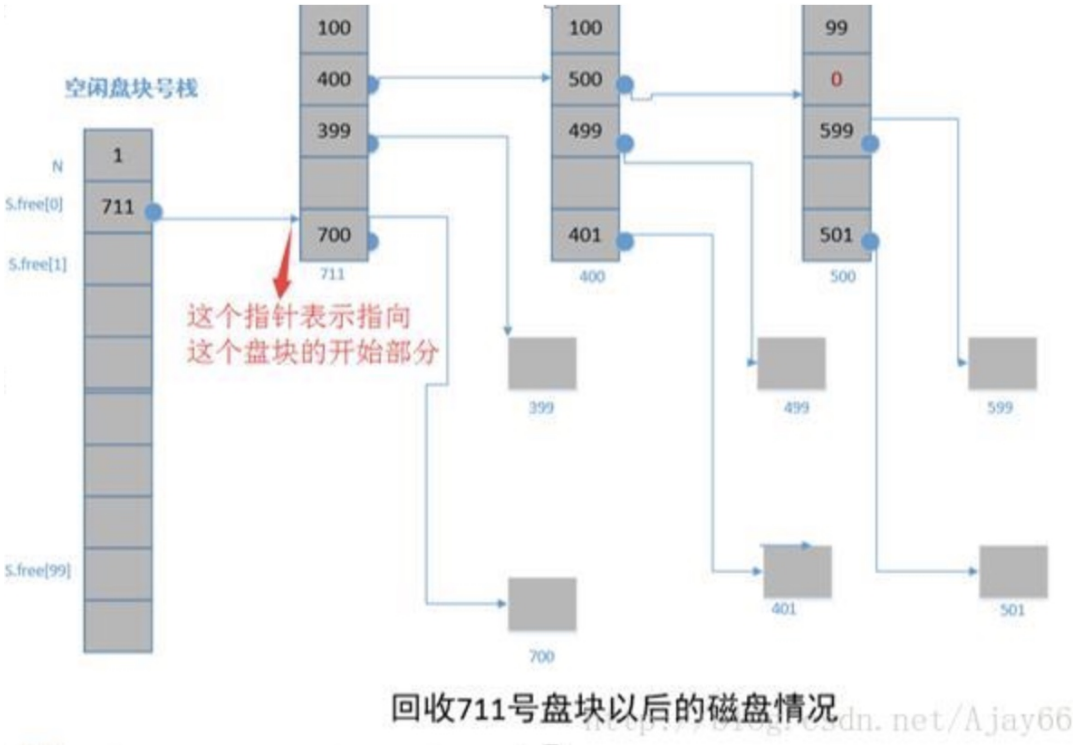


分配301号盘块以后的磁盘情况

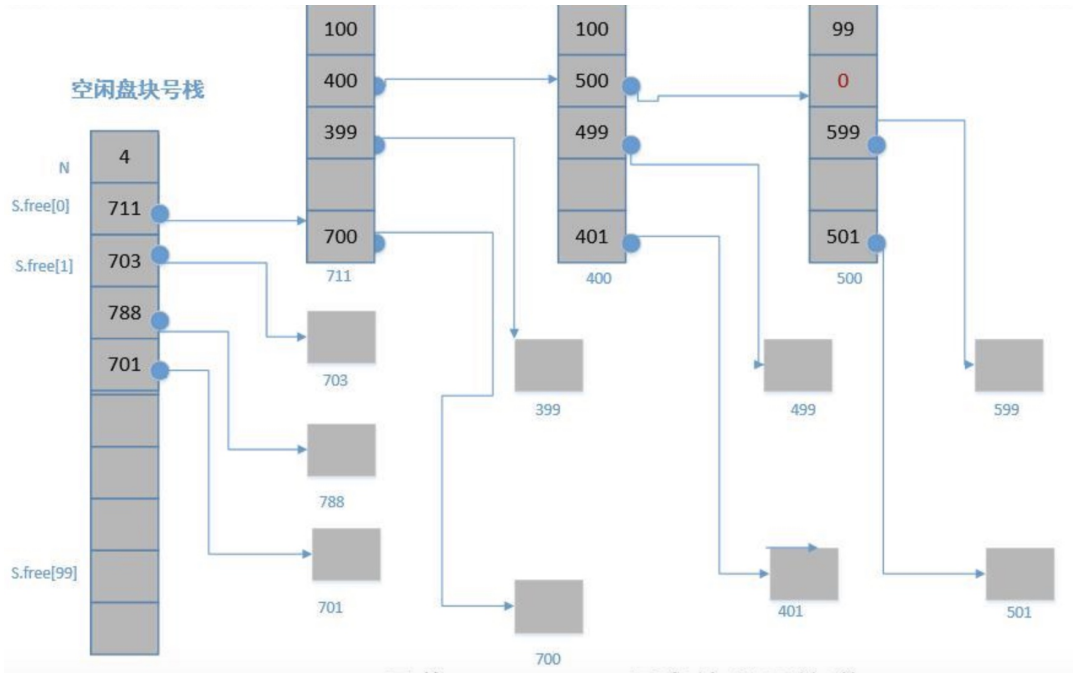
下图为回收了 700 号盘块。



回收了 711 号盘块。

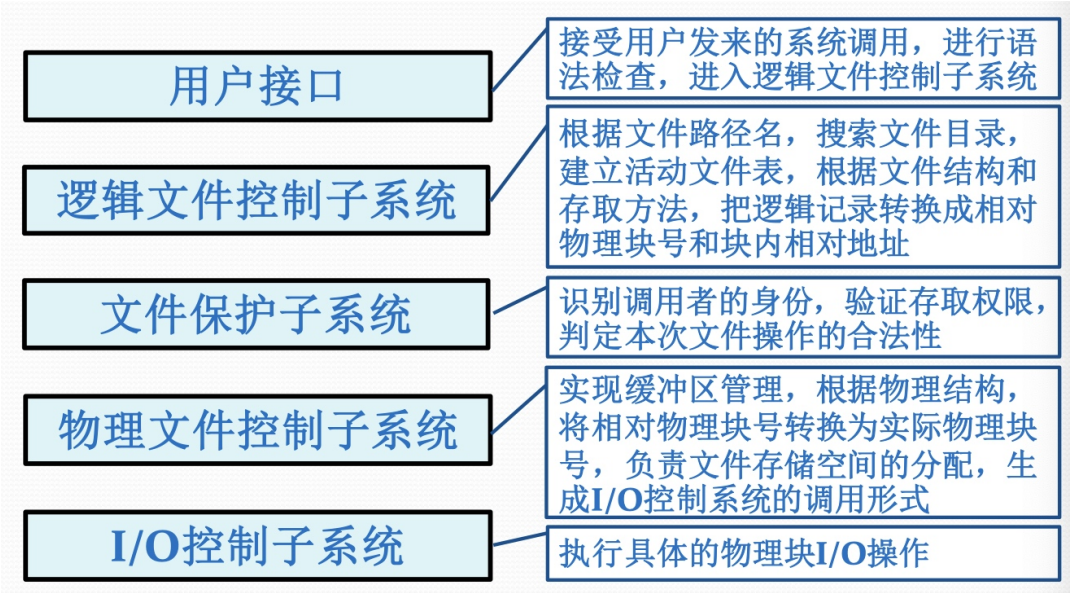


最后回收了 703、788、701 号盘块。



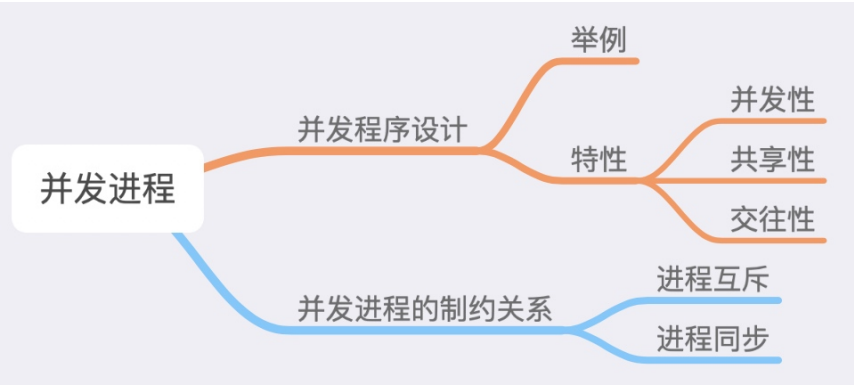
5.12 文件系统的实现层次

1. 检查用户发起的系统调用
2. 逻辑记录 -> 相对物理地址
3. 检查用户权限
4. 相对物理地址 -> 绝对物理地址
5. 具体物理块 I/O 操作



六、并发程序设计

6.1 并发进程

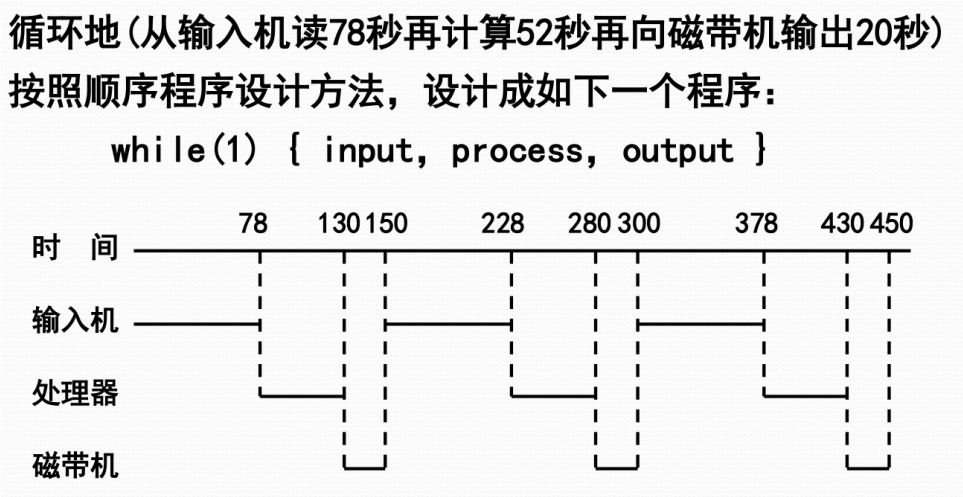


6.1.1 并发程序设计

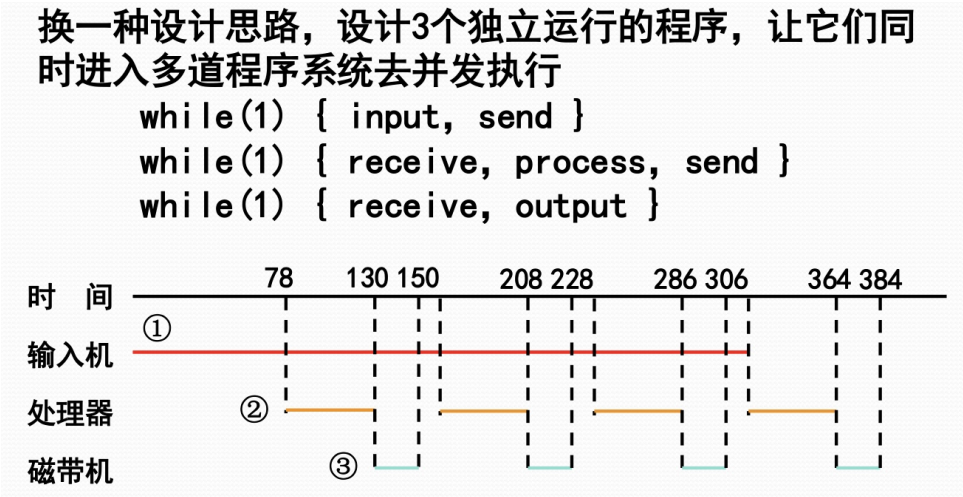
并发程序设计: 把一个具体问题求解设计成若干个可同时执行的程序模块的方法

> 并发设计举例

先是最简单的顺序程序设计。



不难发现，处理器利用率为，$\dfrac{52}{78+52+20} \approx 35\%$。因此我们对上述过程进行改进，引入并发程序设计，如下所示。



在这种设计方式中，我们可以得到处理器利用率为，$\dfrac{52*n}{78*n+52+20} \approx 67\%$。

> 特性

从上述例子中，我们很明显地看到了并发程序设计的优势，因此我们来总结这种程序设计方式的特性。

- 并行性: 多个进程在多道程序系统中并发执行或在多处理器系统中并行执行
  - 提高了计算效率
- 共享性: 多个进程共享软件资源
- **交往性**: 多个进程并发执行时存在制约，增加了程序设计的难度

就是这个交往性导致了并发程序设计中关于进程互斥与同步的问题。

6.1.2 并发程序设计

> 无关与交往的并发进程

无关的并发进程:

- 一组并发进程分别在不同的变量集合上运行，一个进程的执行与其他并发进程的进展无关。

交往的并发进程:

- 一组并发进程共享某些变量，一个进程的执行可能影响其他并发进程的结果。

> 进程互斥问题

进程互斥: 并发进程之间因相互争夺独占性资源而产生的竞争制约关系

```
Int A[1..m]          // 机票数共享变量
process Ti {         // i = 1, 2, …
  Int Xi;
  按旅客定票要求找到航班j;
  Xi = A[j];
  if (Xi>=1) {                 进程1在此区间内被中断，
      Xi = Xi - 1;             进程2又卖此航班的票
      A[j] = Xi;
      {输出一张票};                          卖出同一张票！
  } else {输出票已售完};
}
```

如在上图的 if 部分，进程 1 执行到此处恰好被中断，Xi 还没有变化，因此进程 2 进入卖掉了票。之后轮到进程 1 执行，进程 1 也把票卖了，造成了错误。

> 进程同步问题

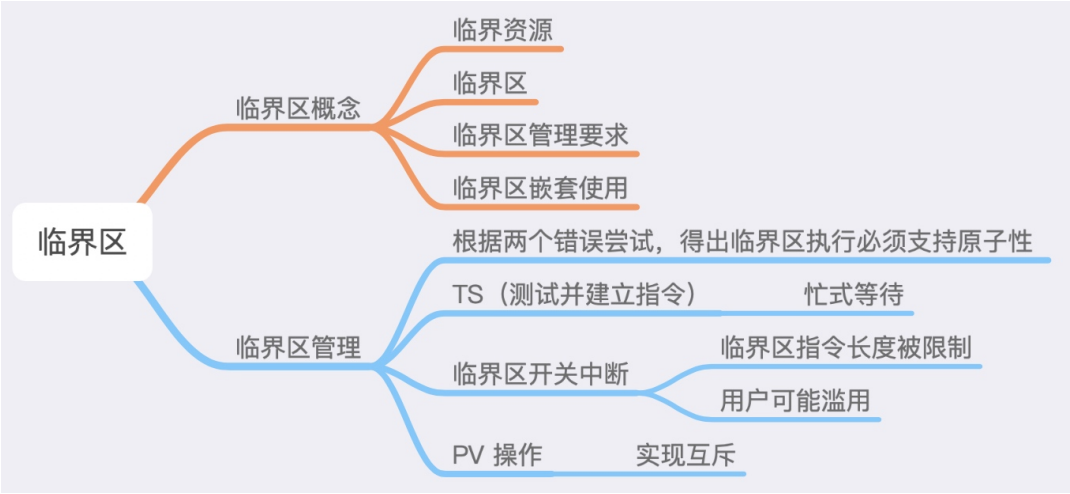
进程同步: 并发进程之间为完成共同任务基于某个条件来协调执行先后关系而产生的协作制约关系。

- 同步的关键在于等待。
- 例如计算进程必须在输入进程输入结束之后，才能对输入结果进行处理。

```
Int x;          //记录空闲内存量的共享变量
procedure borrow (Int B) {
  if (B>x)                                  1 此处中断
    {进入等待队列等待主存资源}              3 永远等待
  x=x-B; {修改主存分配表, 获得主存资源}
}
procedure return (int B) {
    x=x+B;
    {修改主存分配表}
    {释放等主存资源的进程}                  2 全部归还
}
```

如上述的进程获取与释放资源的同步过程。在分配函数中，进程 1 进入 if，但没有进入队列时被中断。此时进程 2 释放了所有进程，然后回到进程 1 进入等待队列，此时进程 1 将会永远等待。

6.2 临界区



6.2.1 临界区概念

> 临界资源

临界资源: 互斥共享变量所代表的资源，即一次只能被一个进程使用的资源。

> 临界区

临界区: 并发过程中与互斥共享变量相关的程序段。

如果两个进程同时停留在相关的临界区内，就会出现进程互斥或同步问题。

- 两个进程的临界区有相同的临界资源，就是相关的临界区，必须互斥进入。
- 两个临界区不相关，进入则没有限制。

> 临界区的描述

- 确定临界资源

```
shared <variable>
```

- 确定临界区

```
region <variable> do <statement_list>
```

> 临界区管理的三个要求

1. 一次至多允许一个进程停留在相关的临界区
2. 一个进程不能无限止地停留在临界区内
3. 一个进程不能无限止地等待进入临界区（避免饿死）

> 临界区的嵌套使用

如果要对临界区进行嵌套使用，则必须规定各临界区资源的优先级，否则就会出现下述的死锁情况。

6.2.2 临界区两个错误尝试

错误想法: 利用一个标志量来标记进程是否在临界区中，并与 while 一同使用。

> 尝试一

看上去没有问题，但是会导致两个进程同时进入临界区。

1. P1 刚执行完 while，发生中断，P2 执行
2. P2 刚进入临界区，发生中断，P1 执行
3. P1 进入临界区，此时两个进程同时进入临界区

process P1	process P2
begin	begin
while inside2 do [ ];	while inside1 do [ ];
inside1 := true;	inside2 = true;
临界区;	临界区;
inside1 := false;	inside2 := false;
end;	end;

> 尝试二

下述方式依然存在问题，两个进程会陷入死锁。

1. P1 刚执行完 inside1 的赋值，被中断，P2 执行
2. P2 执行完 inside2 的赋值后，P1 与 P2 陷入死锁

process P1	process P2
begin	begin
inside1 := true;	inside2 = true;
while inside2 do [ ];	while inside1 do [ ];
临界区;	临界区;
inside1 := false;	inside2 := false;
end;	end;

根据两次尝试，我们可以发现上述的赋值和 while 两条指令的执行过程不能发生中断，否则就会出现错误。

6.2.3 TS 指令（不可行）

因此我们将赋值与 while 两条指令进行合并，得到了测试并建立指令。

```
TS(x) {
   if (x==false) { x=true; return true;
   } else return false;
}
```

```
   Boolean lock;
   lock = false;                          // 临界区可用
process Pi {                              // i = 1,2,...,n
   Boolean pi;
   repeat pi=TS(lock) until pi;           // 循环请求锁
   临界区;
   lock = false;                          // 解锁
}
```

或稍微进行改进，形成 swap 指令。

```
swap (a,b) { temp=a; a=b; b=temp; }
```

```
   Boolean lock;
   lock = false;            //临界区可用
process Pi  {               // i = 1,2,...,n
   Boolean pi;
   pi = true;
   repeat swap(lock, pi) until !pi;  //循环请求锁
   临界区;
   lock = false;            //解锁
}
```

不难发现，两条指令虽然保证程序不会出问题，但会导致忙式等待，非常降低程序效率。

6.2.4 临界区开关中断（不可行）

还有一个非常简单的方法是在进出临界区时开关中断，这样临界区执行就不会发生中断了，执行具有原子性。

```
关中断;
      临界区
开中断;
```

> 缺点

这种方式仅适用于系统内部程序的临界区操作手段，而且要保证指令长度短小精悍。

如果把开关中断权限开放给用户，可能会发生滥用，即造成巨大破坏性。

6.2.5 PV 操作（信号量）

为了改进 TS 或开关中断方式，我们引入 PV 操作。

> 信号量

核心数据结构: 等待进程队列

信号量声明: 资源报到，建立队列

申请资源原语: 若申请不到，调用进程入队等待

归还资源原语: 若队列中有等待进程，需释放

信号量撤销: 资源注销，撤销队列

```
typedef struct semaphore{
```

```
    int value;          // 信号量值
    struct pcb *list;   // 等待进程队列指针
}
```

- 每个信号量建立一个等待进程队列
- 每个信号量对应一个整数值
    - 正值表示资源可复用次数
    - 0 值表示无资源且无进程等待
    - 负值表示资源等待队列中进程个数

> PV 操作

```
procedure P(semaphore:s) {
    s = s - 1;          // 信号量减 1
    if (s < 0) W(s);    // 若信号量小于 0，则进程进入等待队列
}
procedure V(semaphore:s) {
    s = s + 1;          // 信号量加 1
    if (s <= 0) R(s);   // 若信号量小于等于 0，则释放等待队列中一个进程
}
```

我们可以利用 PV 操作来解决进程互斥问题。

```
Semaphore s;
s = 1;
cobegin
process Pi {
    ...
    P(s);
    临界区
    V(s);
    ...
}
coend;
```

> PV 操作解决机票问题

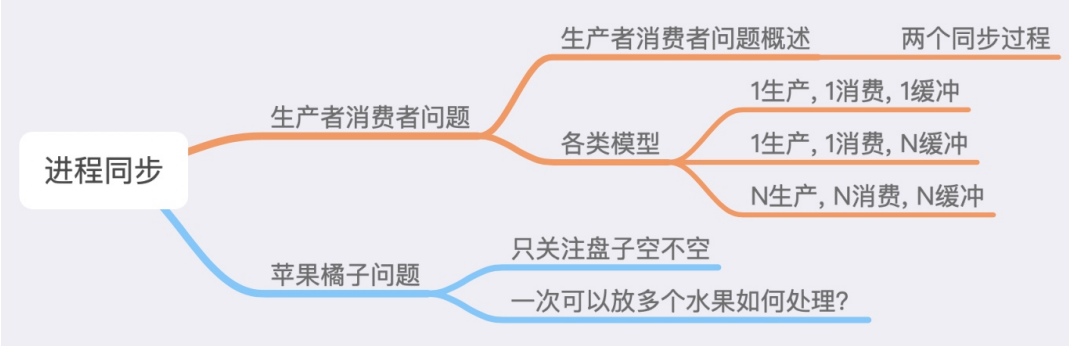
```
int A[m];
Semaphore s;
s = 1;
cobegin
process Pi {
    Li: 按旅客订票要去找到 A[j]
    P(s);
    if ( A[j] >= 1 ) {
        A[j]--;
```

```
        V(s);
        // 输出一张票
    }
    else {
        V(s);
        // 输出票已售完
    }
    goto Li;
}
coend;
```

注意上述代码中，P 操作与 V 操作在执行路径上一一匹配。除此之外，上述代码还有一个问题，即只有相同航班的票数才是相关的临界资源，因此用一个信号量处理全部机票会影响进程并发度，下述代码为修改后的代码。

```
int A[m];
Semaphore s[m];
for (int j = 0; j < m; j++) s[j] = 1;
cobegin
process Pi {
    L1: 按旅客订票要去找到 A[j]
    P(s[j]);
    if ( A[j] >= 1 ) {
        A[j]--;
        V(s);
        // 输出一张票
    }
    else {
        V(s);
        // 输出票已售完
    }
    goto Li;
}
coend;
```

6.3 进程同步



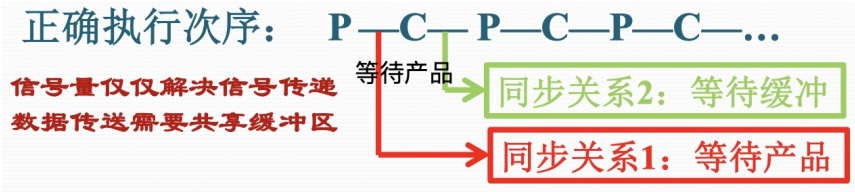
进程同步: 并发进程为完成共同任务基于某个条件来协调执行先后关系而产生的协作制约关系

- 一个进程的执行等待来自于其他进程的消息

6.3.1 生产者消费者问题概述

生产者、消费者共享缓冲区。

- 缓冲区有空位时，生产者可放入产品，否则等待
- 缓冲区有产品时，消费者可取出产品，否则等待



6.3.2 1生产，1消费，1缓冲

```
Int B;                    // 共享缓冲区
Semaphore sput;           // 可以使用的空缓冲区数
Semaphore sget;           // 缓冲区内可以使用的产品数
sput = 1;                 // 缓冲区内允许放入一件产品
sget = 0;                 // 缓冲区内没有产品
```

```
process producer {
    L1:
    produce a product;
    P(sput);
    B = product;
    V(sget);
    goto L1;
}
```

```
process consumer {
    L2:
    P(sget);
    Product = B;
    V(sput);
    consume a product;
    goto L2;
}
```

6.3.3 1生产，1消费，N缓冲

缓冲区扩大了，因此需要加上缓冲区首尾指针。

```
Int B[k];                    // 共享缓冲区队列
Semaphore sput;              // 可以使用的空缓冲区数
Semaphore sget;              // 缓冲区内可以使用的产品数
sput = k;                    // 缓冲区内允许放入 k 件产品
sget = 0;                    // 缓冲区内没有产品
Int putptr, getptr;          // 循环队列指针
putptr = 0; getptr = 0;
```

```
process producer_i {
    L1:produce a product;
    P(sput);
    B[putptr] = product;
    putptr =(putptr+1) mod k;
    V(sget);
    goto L1;
}
```

```
process consumer_j {
    L2:P(sget);
    product= B[getptr];
    getptr=(getptr+1) mod k;
    V(sput);
    consume a product;
    goto L2;
}
```

6.3.4 N生产，N消费，N缓冲

由于生产者和消费者数量增加，因此缓冲区的首尾指针也需要保证互斥性。

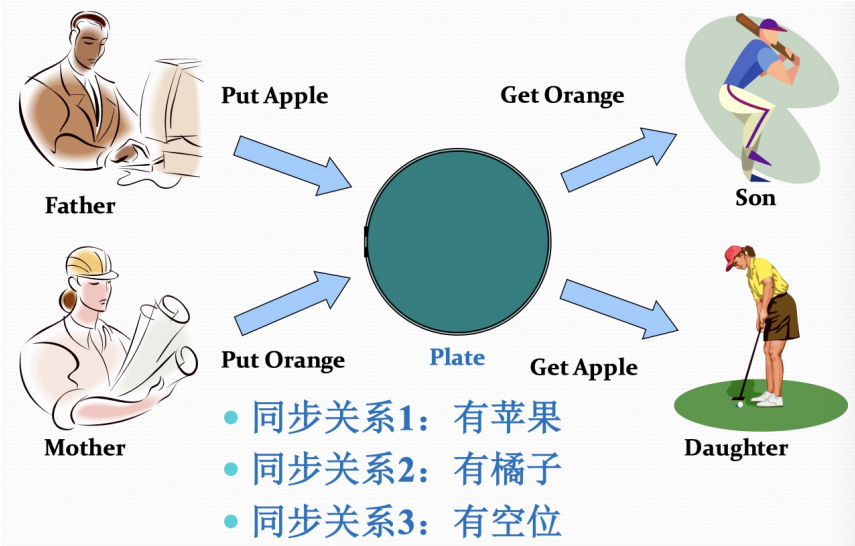
```
Int B [k];
Semaphore sput;              /* 可以使用的空缓冲区数 */
Semaphore sget;              /* 缓冲区内可以使用的产品数 */
sput = k;                    /* 缓冲区内允许放入 k 件产品 */
sget = 0;                    /* 缓冲区内没有产品 */
Int putptr, getptr; putptr = 0; getptr = 0;
s1,s2:semaphore;             /* 互斥使用 putptr,getptr */
s1 = 1; s2 = 1;
```

```
process producer_i {
    L1:produce a product;
    P(sput);
    P(s1);
    B[putptr] = product;
    putptr =(putptr+1) mod k;
    V(s1);
    V(sget);
    goto L1;
}
```

```
process consumer_j {
    L2:P(sget);
    P(s2);
    Product= B[getptr];
    getptr=(getptr+1) mod k;
    V(s2);
    V(sput);
    consume a product;
    goto L2;
}
```

该问题有个注意点，先同步后互斥，否则会死锁。如果是先互斥再同步的话，会出现生产者把缓冲区生产满了，此时先拿锁，然后进入等待队列，而消费者拿不到锁也没办法消费，导致死锁。

6.3.5 苹果橘子问题




```
Int plate;
Semaphore sp;          /* 盘子里可以放几个水果 */
Semaphore sg1;         /* 盘子里有桔子 */
Semaphore sg2;         /* 盘子里有苹果 */
sp = 1;                /* 盘子里允许放入一个水果*/
sg1 = 0;               /* 盘子里没有桔子 */
sg2 = 0;               /* 盘子里没有苹果*/
```

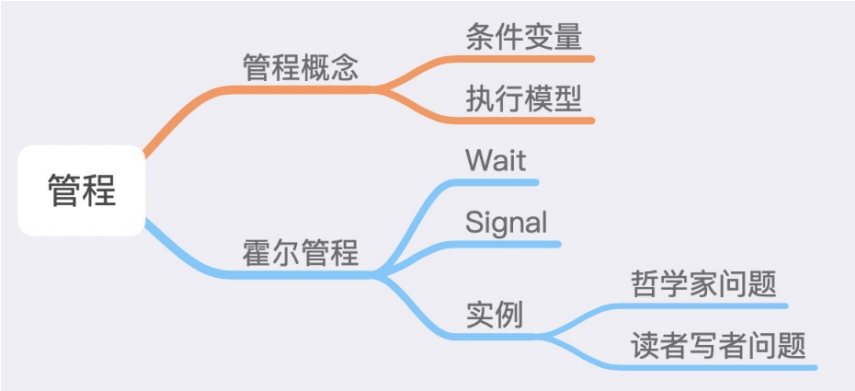
```
process father {
        L1: 削一个苹果；
        P(sp);
        把苹果放入plate；
        V(sg2);
        goto L1;
}
process mother {
        L2: 剥一个桔子；
        P(sp);
        把桔子放入plate；
        V(sg1);
        goto L2;
}
```

```
process son {
        L3: P(sg1);
        从plate中取桔子；
        V(sp);
        吃桔子；
        goto L3;
}
process daughter {
        L4: P(sg2);
        从plate中取苹果；
        V(sp);
        吃苹果；
        goto L4;
}
```

如果盘子里面可以放多个水果的话，首先扩大缓冲区，其次令 sp = n，然后设置一个缓冲区锁，每次只能有一个人放或取东西。每次拿或取的时候采用遍历搜寻的方式。

接下来需要考虑 P(s) 放的位置，P(s) 互斥操作要在同步操作之后，否则会发生错误。V(s) 操作无先后关系。

6.4 管程



6.4.1 管程概念

> 概念提出

管程本质是对信号量的类封装，简化操作。

- 管程试图抽象相关并发进程对共享变量访问，以提供一个友善的并发程序设计开发环境。
- 管程是由若干公共变量及其说明和所有访问这些变量的过程所组成。
- 管程把分散在各个进程中互斥地访问公共变量的那些临界区集中起来管理，管程的局部变量只能由该管程的过程存取。
- **进程只能互斥地调用管程中的过程。**

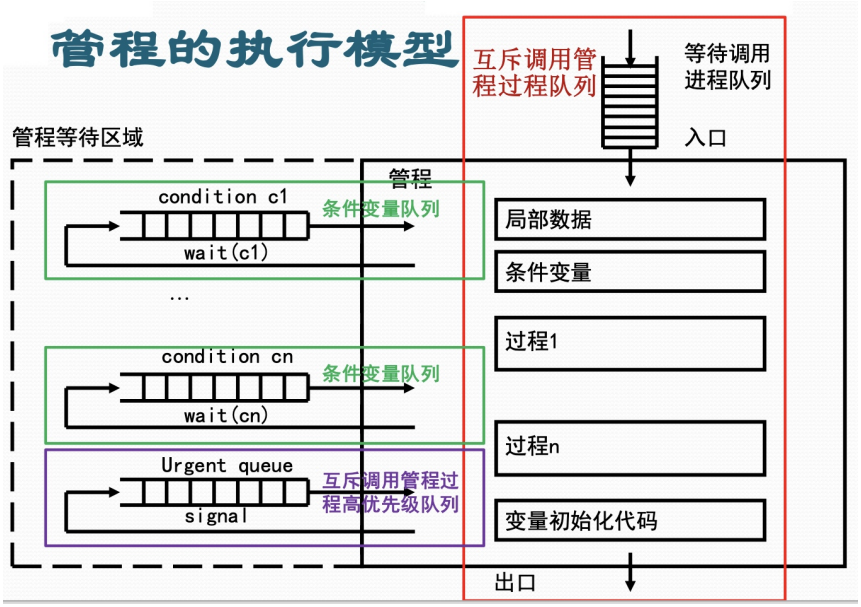
> 管程的条件变量

- 条件变量: 当调用管程过程的进程无法运行时，用于阻塞进程的信号量
- 同步原语 wait: 当一个管程过程发现无法继续时，它在某些条件变量上执行 wait，引起进程阻塞
- 同步原语 signal: 用于释放在条件变量上阻塞的进程

> 执行模型

注意，管程中一共有 3 个等待进程队列。由于进程只能互斥地调用管程中的过程，因此有了互斥调用管程过程队列。

又因为当前在管程中的进程一旦调用 signal，则会有两个进程在管程中，所以又有一个更高优先级的互斥调用队列。



> signal 问题

当使用 signal 释放一个等待进程时，可能出现两个进程同时停留在管程内，此时有三种解决方法。

1. 执行 signal 的进程等待，直到被释放进程退出管程或等待另一个条件（霍尔管程）
2. 被释放进程等待，直到执行 signal 的进程退出管程或等待另一个条件
3. 规定管程中的 signal 操作时过程体的最后一个操作（汉森法，增加程序设计难度）

6.4.2 霍尔管程

> 霍尔管程

使用 signal 释放一个等待进程时，霍尔管程让执行 signal 的进程等待，直到被释放进程退出管程或等待另一个条件。

霍尔管程基于 PV 操作原语实现。

- Wait 和 Signal 可以是程序过程
- 不需要扩展 OS 内核

> 信号量

两个互斥信号量。

```
TYPE interf = RECORD
  mutex : semaphore;
      //调用管程过程前使用的互斥信号量
  next : semaphore;
      //发出signal的进程挂起自己的信号量
  next_count : integer;
      //在next上等待的进程数
END;
```

```
if (IM.next_count > 0) V(IM.next);
else V(IM.mutex);
```

> 条件变量

```
x_sem: semaphore;   // 与资源相关的信号量
x_count: integer;   // 在 x_sem 上等待的进程数
```

> Wait

```
void Wait(semaphore, x_count, IM){
    x_count++;  // 等待数加上自己
    if(IM.next_count > 0) V(IM.next);
    else V(IM.mutex);
    P(x_sem);   // 挂起自己
    x_count--;  // 等待数减去自己
}
```

> Signal

```
void Signal(semaphore, x_count, IM){
    if(x_count > 0){
        IM.next_count++;    // 进程队列加入自己
        V(x_sem);           // 信号量操作
        P(IM.next);         // 将自己阻塞住，先执行被 signal 的进程
        IM.next_count--;    // 进程队列去除自己
    }
}
```

6.4.3 哲学家问题

> 问题描述

> 霍尔管程方法

其中 s 是盘子状态。

```
TYPE dining_philosophers = MONITOR
   var state : array[0..4] of (thinking, hungry, eating);
         s : array[0..4] of semaphore;
         s_count : array[0..4] of integer;
   define pickup, putdown;
   use wait, signal;
procedure test(k : 0..4) {
   if state[(k-1) mod 5]<>eating and state[k]=hungry
        and state[(k+1) mod 5]<>eating then
          { state[k] := eating;   signal(s[k], s_count[k], IM); }
}
procedure pickup(i:0..4) {
   state[i] := hungry;
   test(i);
   if state[i]<>eating
   then wait(s[i], s_count[i],IM);
}
procedure putdown(i:0..4) {
   state[i] := thinking;
   test((i-1) mod 5);
   test((i+1) mod 5);
}
begin   for   i := 0 to 4 do state[i] := thinking;   end;
```

6.4.4 读者写者问题

> 问题描述

有两组并发进程：读者与写者，共享一个文件，要求：1）允许多个读者同时执行读操作；2）任一写者出现后，新的读者与写者均被拒绝；3）写者在完成写操作之前不允许其他读者和写者工作；4）写者欲工作，要等待已存在的读者完成读操作

1、等待读的进程与计数；等待写的进程与计数

2、正在读的进程计数；正在与等待写的进程计数

> 霍尔管程

```
TYPE read-writer = MONITOR
  var Rc, Wc : integer; R, W : semaphore;   rc, wc: integer;
  define start_read, end_read, start_writer, end_writer;
  use wait, signal, check, release;
begin  rc:=0; wc:=0; Rc:=0; Wc:=0; R:=0; W:=0;  end;
procedure start_read;
begin
   if wc>0 then wait(R,Rc,IM);
   rc := rc + 1;
   signal(R, Rc, IM);
end;
procedure end_read;
begin
   rc := rc - 1;
   if rc=0 then signal(W,Wc,IM);
end;
procedure start_write;
begin
   wc := wc + 1;
   if rc>0 or wc>1 then wait(W,Wc,IM);
end;
procedure end_write;
begin
   wc := wc - 1;
   if wc>0 then signal(W,Wc,IM);
            else signal(R, Rc,IM);
end;
```

连续释放读者

6.5 进程通信

6.5.1 进程通信概念

- 交往进程通过信号量操作实现进程互斥和同步，这是一种低级通信方式。
- 进程有时还需要交换更多的信息（如何把数据传送给另一个进程），可以引入高级通信方式 —— 进程通信机制，实现进程间用信件来交换信息。
- 进程通信扩充了并发进程的数据共享。

6.5.2 进程直接通信

发送或接收信件的进程指出信件发给谁，或从谁那里接收信件。

- send(P, 信件): 把信件发送给进程 P
- receive(Q, 信件): 从进程 Q 接收信件

6.5.3 进程间接通信

> 间接通信概念

- 发送或者接收信件通过一个信箱来进行，该信箱有唯一标识符
- 多个进程共享一个信箱
  - send(A, 信件): 把信件传送到信箱 A
  - receive(A, 信件): 从信箱 A 接收信件

> 信箱

信箱是存放信件的存储区域，每个信箱可以分成信箱特征和信箱体两部分。

- 信箱特征: 信箱容量、信件格式、指针
- 信箱体: 分成若干个区，每个区可容纳一封信

> 发送信件原语流程

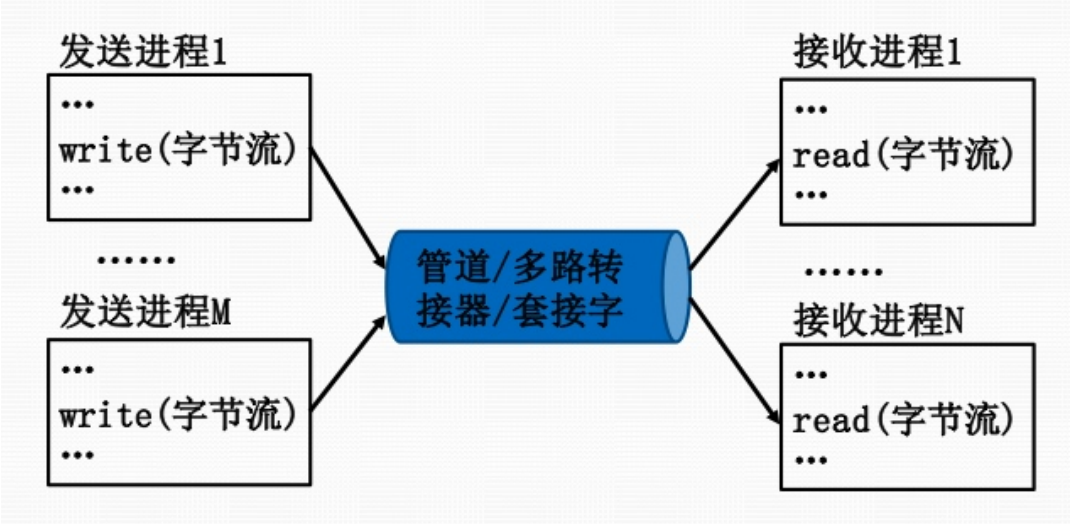
1. 若指定的信箱未满，则把信件送入信箱中指针所指示的位置，释放等待该信箱中信件的等待者
2. 否则，发送信件者被置成等待信箱的状态

> 接收信件原语流程

1. 若指定信箱中有信件，则取出一封信，释放等待信箱的等待者
2. 否则，接收信件者被置成等待信箱中信件的状态

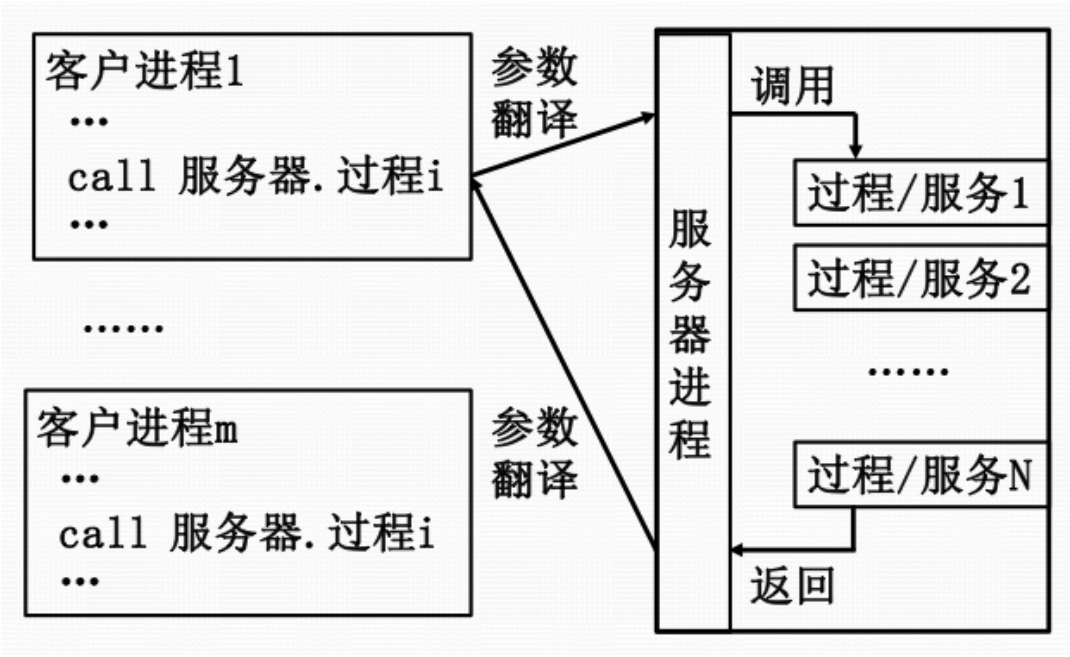
6.5.4 基于字节流的通信规约

- 多个进程使用一个共享的消息缓冲区（管道、多路转接器、套接字）
- 一些进程往消息缓冲区中写入字符流
- 一些进程从消息缓冲区中读出字符流
- 信息交换单位基于字符流，长度任意

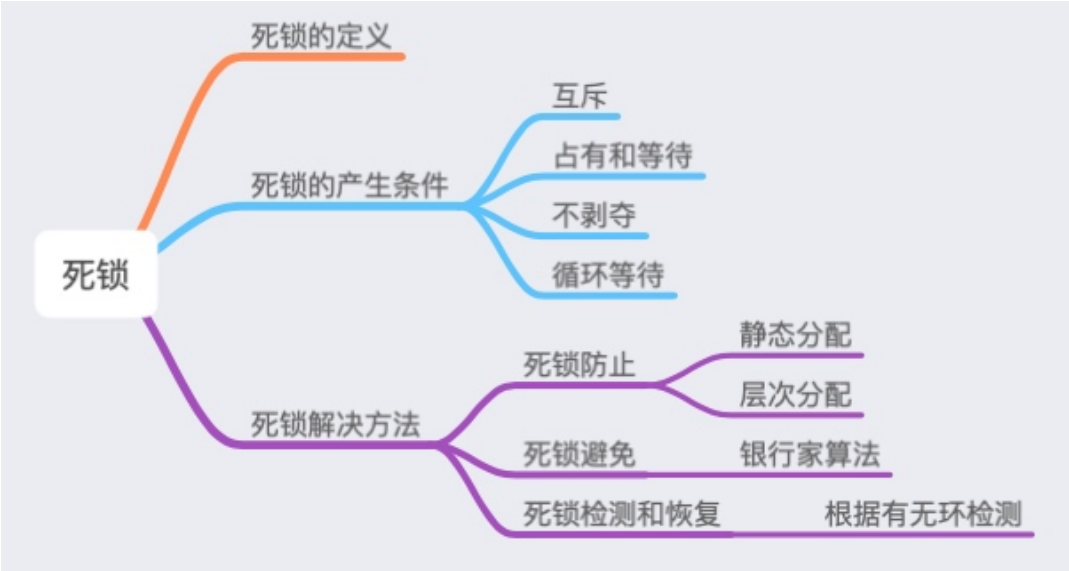


6.5.5 基于RPC的高级通信规约

- 采用客户/服务器计算模式
- 服务器进程提供一系列过程/服务，供客户进程调用
- 客户进程通过调用服务器进程提供的过程/服务获得服务
- 考虑到客户计算机和服务器计算机的硬件异构型，外部数据表示 XDR 被引入来转换每台计算机的特殊数据格式为标准数据格式



6.6 死锁



这里需要主要一下死锁防止和死锁避免的区别，下面给出一段英文描述。

> The main difference between deadlock prevention and deadlock avoidance is that deadlock prevention ensures that at least one of the necessary conditions to cause a deadlock will never occur while deadlock avoidance ensures that the system will not enter an unsafe state.

关键点在于死锁防止是从根本上解决死锁问题，而死锁避免是人为地尽可能避开，还是有出意外的可能。

6.5.1 死锁的定义

一组进程处于死锁状态是指，每一个进程都在等待被另一个进程所占有的、不能抢占的资源。

- 存在 n 个进程 $P_1, P_2, \ldots, P_n$
- 进程 P_i 因为申请不到资源 R_i 而处于等待状态
- 而 R_i 又被 P_{i+1} 占有，R_n 被 P_1 占有
- 显然，这 n 个进程的等待状态永远不能结束，这 n 个进程就处于死锁状态

6.5.2 死锁产生的四个必要条件

1. 互斥条件: 进程应互斥使用资源，任一时刻一个资源仅为一个进程独占
2. 占有和等待条件: 一个进程请求资源得不到满足而等待时，不释放已占有的资源
3. 不剥夺条件: 任一进程不能从另一进程那里抢夺资源
4. 循环等待条件: 存在一个循环等待链，每一个进程分别等待它前一个进程所持有的资源

6.5.3 死锁的防止

> 独占型资源改为共享（破坏条件1）

独占变共享，可以破坏条件 1，但这对许多资源往往是不能做到的。

> 剥夺式调度（破坏条件3）

剥夺式调度目前只适用于对主存资源和处理器资源的分配，而不适用于所有资源。

> 静态分配（预分配 - 破坏条件4）

具体方法: 一个进程必须在执行前就申请它所要的全部资源，并且直到它所要的资源都得到满足之后才开始执行。

缺点: 可能会出现一种情况。进程执行时，资源 A 只使用了一小段时间，但在整个执行过程中，资源 A 始终被占用，造成资源利用率的低效。

> 层次分配（破坏条件4）

具体方法: 将资源分成多个层次。一个进程得到某一层的一个资源后，它只能再申请在较高层的资源，当一个进程要释放某层的一个资源时，必须先释放所占用的较高层的资源。当一个进程获得了某一层的一个资源后，它想再申请该层中的另一个资源，那么必须先释放该层中的已占资源。

缺点: 低层次资源使用效率依然较低。

6.5.4 死锁的避免

当不能防止死锁的产生时，如果能掌握并发进程中与每个进程有关的资源申请情况，仍然可以避免死锁的发生。

只需在为申请者分配资源前先测试系统状态，若把资源分配给申请者会产生死锁的话，则拒绝分配，否则接收申请，为它分配资源。

> 银行家算法: 借钱给有偿还能力的客户

- 系统首先检查申请者对资源的最大需求量，如果现存的资源可以满足它的最大需求量时，就满足当前的申请
- 即仅仅在申请者可能无条件地归还它所申请的全部资源时，才分配资源给它

注意: 这种判断不仅是对单个用户的判断，而是对全体用户的判断。可以采用循环探测方法来判断。

例如下述系统，一共有三个进程 P、Q、R，系统只有 10 个资源。

进程	已占资源	还需要申请数
P	4	4
Q	2	2
R	2	7

则此时只有 Q 申请资源，系统才会予以分配，其余进程申请均不会分配。

6.5.5 死锁的检测

死锁的检测对资源的分配不加限制，但系统定时运行一个 “死锁检测” 程序，判断系统内是否已出现死锁，若检测到死锁则设法加以解除。

> 检测方法

- 设置两张表格来记录进程使用资源的情况。
  - 等待资源表: 记录每个被阻塞进程等待的资源
  - 占用资源表: 记录每个进程占有的资源

资源	占用进程
r1	P1
r2	P2
r3	P3
r4	P4
r5	P5
...	...

进程	等待资源
P1	r1
P2	r2
P3	r3
...	...

因此可以根据上述两张表构建出一张死锁的有向图，若 P_i 等待资源 r_k，且 r_k 被进程 P_j 占用，则 P_i 和 P_j 具有 “等待占用关系”，记为 $W(P_i, P_j)$。

我们可以构造一个二维矩阵，运行传递必包算法，如果对角线为 1，则存在环，出现死锁。也可以用拓扑排序的方法，O(n) 判环。

```
for(int k = 1; k <= n; k++)
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            b[i][j] = b[i][j] | (b[i][k] & b[k][j]);
```

进程 A[bij] 进程	P1	P2	.....	Pn
P1	b11	b12	.....	bn2
P2	b21	b22	.....	bn2
...	...	...	.....	...
Pn	bn1	bn2	.....	bn2

其中 $b_{ij}=\begin{cases}1 & \text{当Pi等待被Pj占用的资源时} \\ 0 & \text{当Pi与Pj不存在等待占用关系时}\end{cases}$

> 死锁后的解决方法

- 可以采用重新启动进程执行的办法，恢复工作应包含重启动一个或全部进程，以及从哪一点开始重启动
- 全部卷入死锁从头开始启动，但这样的代价是相当大的
- 在进程执行过程中定时设置校验点，从校验点开始重新执行
- 中止一个卷入死锁的进程，以后重执行