

目录

简介	1.1
第一部分 计算机的组成	1.2
计算机概述篇	1.2.1
计算机组成篇	1.2.2
计算机计算篇	1.2.3
第二部分 操作系统	1.3
操作系统基础篇	1.3.1
第三部分 网络	1.4

计算机组成与操作系统

本书主要是对计算机组成、操作系统、网络等的学习总结！

计算机原理之概述篇

1、计算机的发展简史

1.1、电子管计算机 (1946~1957)

- 集成度小，空间占用大；
- 功耗高，运行速度慢；
- 操作复杂，更换程序需要接线；

1.2、晶体管计算机 (1957~1964)

1948 年贝尔实验室发明了晶体管，此后晶体管为计算机带来了革命性的进步！

- 集成度相对较高，空间占用相对较小；
- 功耗相对较低，运行速度较快；
- 操作相对简单，交互更加方便；

1.3、集成电路计算机 (1964~1980)

德州仪器的工程师发明了集成电路(IC)！

- 计算机变得更小；
- 功耗变得更低；
- 计算速度更快；

1.4、超大规模集成电路计算机 (1980~现在)

- 一个电路集成了上百万的晶体管；
- 速度更快，体积更小，价格更低，更能被大众接受；
- 用途丰富：文本处理、表格处理、高交互的游戏与应用；

1.5、微型计算机的发展历史

微型计算机主要受限于性能；

单核 CPU

- 1971~1973 高于 500KHZ 频率的微型计算机（字长 8 位）；
- 1973~1978 高于 1 MHZ 频率的微型计算机（字长 8 位）；
- 1978~1985 高于 500 MHZ 频率的微型计算机（字长 16 位）；
- 1985~2000 高于 1 GHZ 频率的微型计算机（字长 32 位）；
- 2000~现在 高于 2 GHZ 频率的微型计算机（字长 64 位）；

摩尔定律：微型计算机的发展历史，集成电路的性能，每 18 ~ 24 个月就会提升一倍！

多核 CPU

- 2005 年 Intel 奔腾系列双核 CPU、AMD 速龙系列；
- 2006 年 Intel 酷睿四核 CPU；
- 现在 Intel 酷睿系列十六核 CPU；
- Intel 至强系列五十六核 CPU；

2、计算机的分类

2.1、超级计算机

- 功能最强、运算速度最快、存储容量最大的计算机；
- 多用于国家高科技领域和尖端技术研究；
- 运算速度单位 TFlop/s : $1 \text{ TFlop/s} =$ 每秒一万亿次浮点计算；

2.2、大型计算机

- 又称大型机、大型主机、主机等；
- 具有高性能、可处理大量数据与复杂的计算；
- 在大型机市场领域，IBM 占据很大的市场份额；

由于 IOE 高维护费用的存储系统、不够灵活、伸缩性弱，阿里巴巴在 2008 年提出了去 IOE 行动：

- I : IBM 服务器提供商
- O : Oracle 数据库软件提供商
- E : EMC 存储设备提供商

2.3、迷你计算机（服务器）

普通服务器已经替代了传统的大型机，成为大规模企业计算的中枢

- 又称小型机、普通服务器；
- 不需要特殊的空调场所；
- 具备不错的运算力、可以完成较复杂的运算；

2.4、工作站

- 高端的通用微型计算机，提供比个人计算机更强大的性能；
- 类似于普通台式电脑、体积较大、性能强劲
- 适用于图片工作者、视频工作者

2.5、微型计算机

- 又称个人计算机，是最普通的一类计算机；
- 台式机、笔记本、一体机 iMac
- 麻雀虽小，五脏俱全

3、计算机的体系与构成

3.1、冯诺依曼体系

冯诺依曼体系“将程序指令和数据一起存储的计算机设计概念结构”

早期计算机仅含固定用途，如果想要改变用途，需要改变程序更改结构、重新设计电路！冯诺依曼：存储程序指令，设计通用电路！

现代计算机本质上都是冯诺依曼机，冯诺依曼体系的几个关键要素：

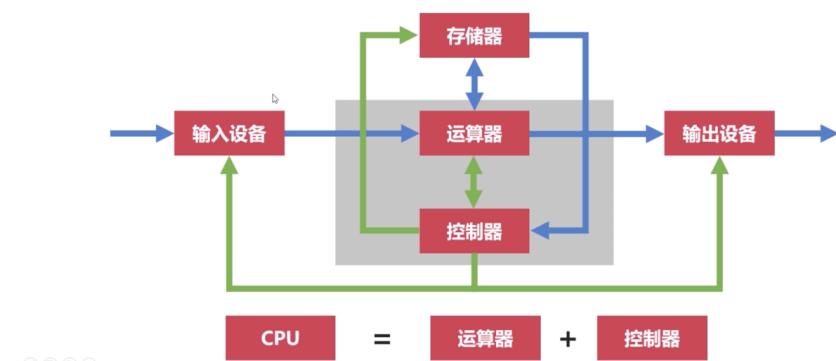
- 必须有一个存储器：存储运行的程序以及运行所需的数据；
- 必须有一个控制器：控制程序的流转；
- 必须有一个运算器：负责运算的操作；
- 必须有输入设备

- 必须有输出设备

冯诺依曼体系：

- 能够把需要的程序和数据送至计算机中 (输入设备：鼠标、键盘);
- 能够长期记忆程序、数据、中间结果以及最终运算结果的能力，通过记忆才能进行下一步的操作 (存储器);
- 能够具备算术、逻辑运算和数据传送等数据加工处理的能力 (运算器、控制器);
- 能够按照要求将处理结果输出给用户 (输出设备：显示器、打印机);

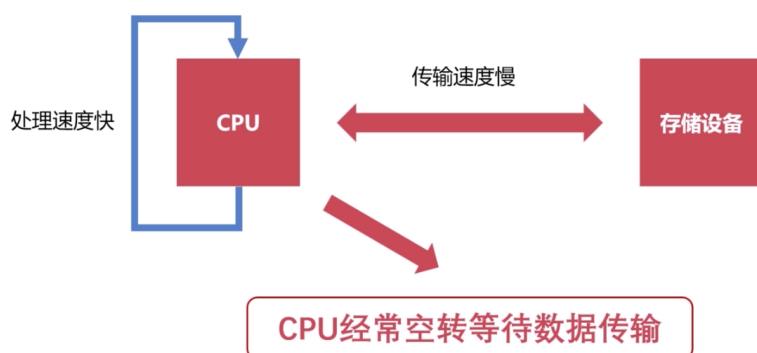
冯诺依曼体系



冯诺依曼瓶颈

CPU 和存储器速率之间的问题无法调和：CPU 的读写速率很快，而存储器的读写速率没有 CPU 快；导致 CPU 经常空转等待数据的传输！

冯诺依曼瓶颈

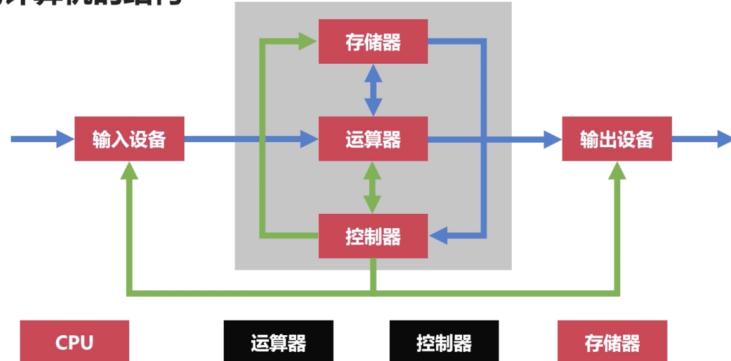


只有把 CPU 跑满，才是充分利用计算机的表现！

3.2、现代计算机的结构

现代计算机在冯诺依曼体系结构基础上进行修改；解决 CPU 与储存设备之间的性能差异问题！

现代计算机的结构



存储器：磁带、硬盘；更高速的设备，内存，CPU 的寄存器！

4、计算机的层次与编程语言

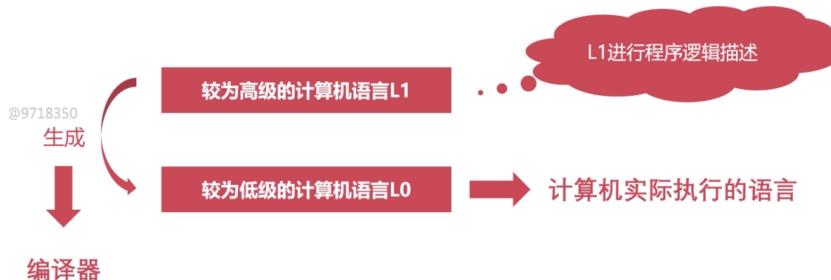
4.1、程序翻译与程序解释

- 计算机执行的指令都是 0 和 1（L0 程序）；
- 程序翻译生成新的 L0 程序；程序解释不生成新的 L0 程序；
- 解释过程，由 L0 编写的解释器去解释 L1 程序；

翻译+解释语言：`Java / C#`；

4.1.1、程序翻译

计算机的世界中，只有 0 与 1 两种表达；与人类的语言不相同，因此需要将人类语言翻译为计算机理解的程序语言！



常见语言：`C / C++ / Objective-C / Golang`；

4.1.2、程序解释



常见语言：`Python / PHP / JavaScript`；

4.2、计算机的层次与编程语言



硬件逻辑层：门、触发器等逻辑电路组成；属于电子工程的领域； 微程序机器层： 编程语言为 微指令集； 微指令所组成的微程序直接交由硬件执行； 传统机器层： 编程语言是 CPU 指令集； 编程语言直接和硬件相关； 不同架构的 CPU 使用不同的指令集； 指令集存储在 CPU 内部； 操作系统层： 向上为用户提供了简易的操作界面； 向下对接指令系统，管理硬件资源； 是在软件和硬件之间的适配层； 汇编语言层： 汇编语言可以翻译为直接执行的机器语言； 完成翻译过程的程序就是汇编器； 高级语言层： C / C++ / Objective-C / Python / PHP / Java / JavaScript / Golang 等； 应用层： 满足计算机针对某种用途而专门设计的程序；

5、计算机的计算单位

5.1、容量单位

- 比特位 bit : 在物理层面，高低电平 0 / 1 记录信息；
- 字节 Byte : 1 Byte = 8 bits ;
- 千字节 KB : 1 KB = 1024 Byte ; 常用于 CPU 的寄存器；
- 兆字节 MB : 1 MB = 1024 KB ; 常用于高效缓存
- 吉字节 GB : 1 GB = 1024 MB ; 常用于内存/硬盘；
- 太字节 TB : 1 TB = 1024 GB ; 常用于硬盘；
- 拍字节 PB : 1 PB = 1024 TB ; 常用于云硬盘；
- 艾字节 EB : 1 EB = 1024 PB ; 常用于数据仓库；

疑问：为什么网上买的移动硬盘 500 G , 格式化之后只剩下 465 G 了？ 答案：
因为硬盘商的换算单位为 1000 , 而计算机的换算单位为 1024 !

5.2、速度单位

5.2.1、网络速度

疑问：为什么电信拉的 100 M 光纤，测试峰值速度只有 12 M 每秒？ 答案： 网络速度单位为 Mbps , 即电信拉的 100 Mbps 光纤！ $100 \text{ M/s} = 100 \text{ Mbps} = 100 \text{ Mbit/s} = 100 / 8 \text{ MB/S} = 12.5 \text{ M/S}$!

5.2.2、CPU 频率

*CPU 的速度一般体现为 CPU 的时钟频率；

- CPU 的时钟频率单位一般为赫兹 Hz ；

- 赫兹 `Hz` 就是秒分之一，它是每秒钟的周期性变动重复次数的计量；
- 主流 CPU 的时钟频率都在 `2 GHz` 以上；
- $2 \text{ GHz} = 2 * 1000^3 \text{ Hz} = 20 \text{ 亿次/秒} !$

6、计算机的字符与编码集

6.1、字符编码集的历史

ASCII 码：使用 7 个比特位就可以完全表示 ASCII 码；包含 95 个可打印字符与 33 个不可打印字符（控制字符）！ASCII 码在很多应用和国家的符号都无法表示；如数学符号等！扩展的 ASCII 码，使用 8 个比特位表示，包含数学符号等！

字符编码集的国际化：

- 各个国家的语言多样性；
- 语言体系不一样，不以有限字符组合的语言；
- 中、日、韩等的语言最为复杂；

6.2、中文编码集

`GB 2312`：收录了 6763 个汉字和 682 个其它字符，总计 7445 个字符；`GBK`：向下兼容 `GB 2312`，向上支持国际标准 `ISO`；收录了 21003 个汉字，支持全部中日韩汉字。`Unicode`：兼容全球的字符集，定义了世界通用的字符集！

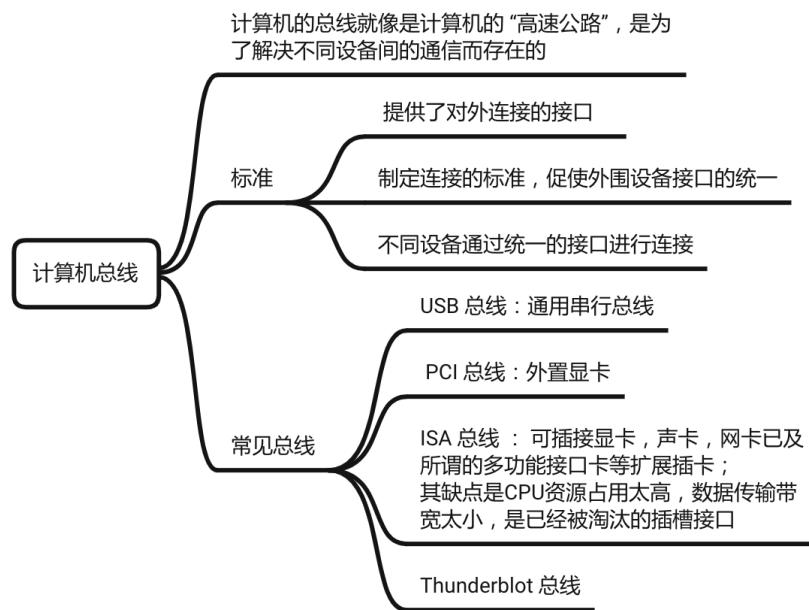
计算机原理之组成篇

1、计算机的总线

1.1、总线的概述

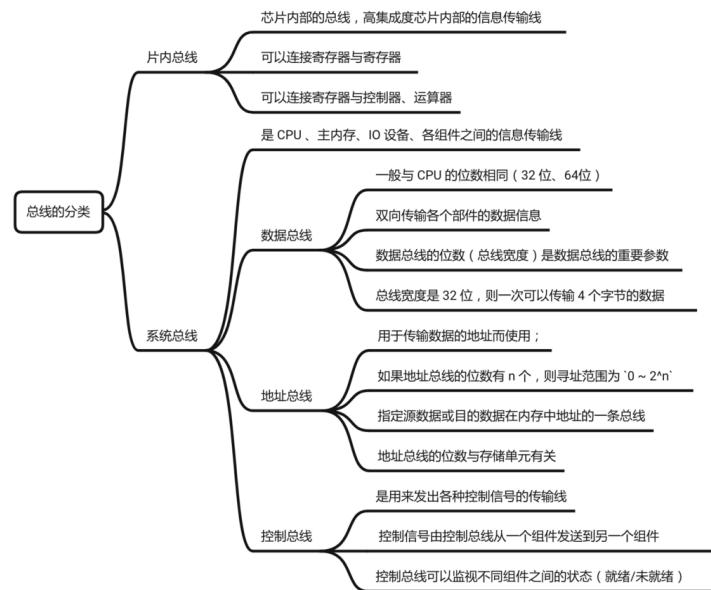
1.1.1、总线是什么？有什么用？

计算机的总线就像是计算机的“高速公路”，是为了解决不同设备间的通信而存在的！



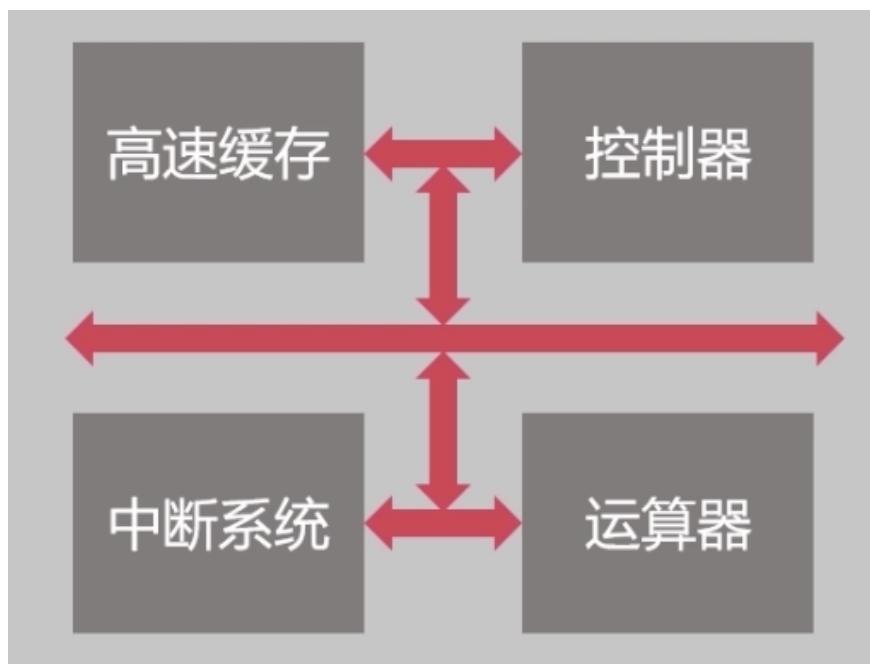
总线标准是系统与各模块、模块与模块之间的一个互连的标准界面。总线标准有利于各模块高效使用总线。如 USB、PCIe 等。

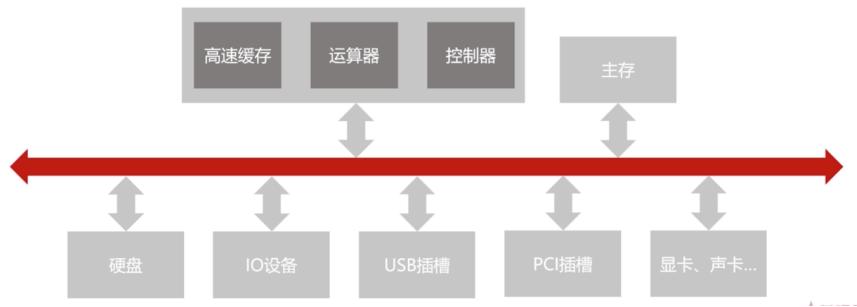
1.1.2、总线的分类



总线分为片内总线与系统总线！

- 片内总线：芯片内部的总线，高集成度芯片内部的信息传输线；可以连接寄存器与寄存器；可以连接寄存器与控制器、运算器；
- 系统总线：分为数据总线、地址总线、控制总线等；是 CPU、主内存、IO 设备、各组件之间的信息传输线；





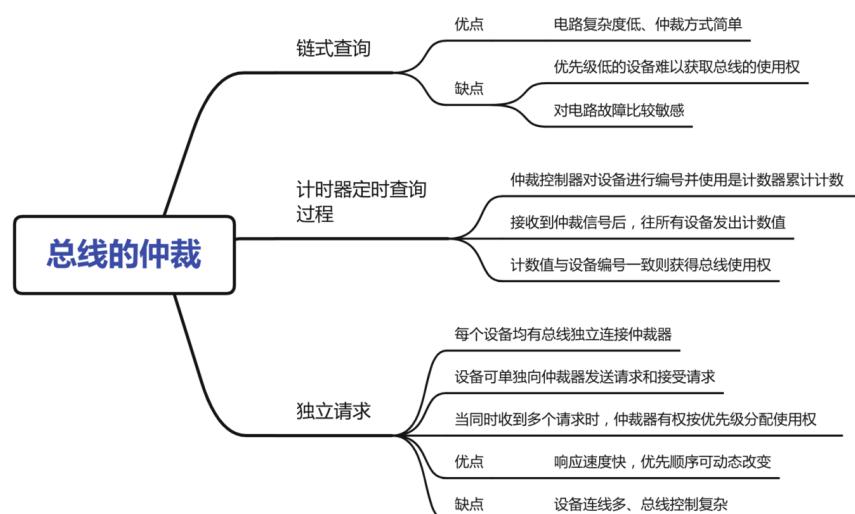
1.2、总线的仲裁

为了解决不同设备使用总线的优先顺序的设备；

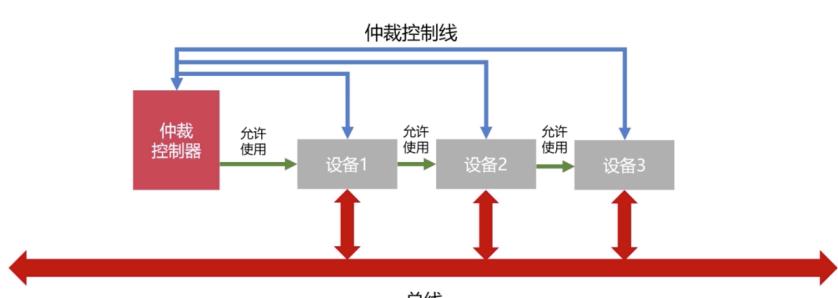
- 总线的仲裁**
- 1、假设主存需要和硬盘、IO设备交换数据；
 - 2、此时硬盘、IO设备都已准备就绪
 - 3、那么总线该由硬盘使用？还是由IO设备使用？



为了解决总线使用权的冲突问题，有三种仲裁方法：



1.2.1、链式查询

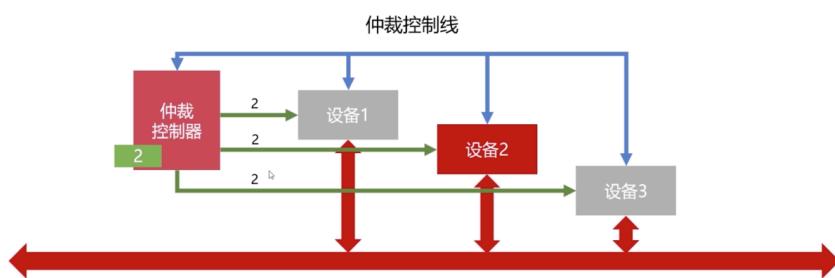


假设设备 2 需要使用总线：

- 那么设备 2 通过仲裁控制线向仲裁控制器发出“使用总线”的请求；
- 仲裁控制器收到请求后发出允许使用的信号；
- 该信号通过链式查询的方式会先进入设备 1；
- 如果设备 1 不需要使用，那么该信号会来到设备 2，设备 2 拿到信号后就可以使用总线；
- 假设设备 1、设备 2 同时发出使用总线的请求：那么设备 1 将优先拿到总线的使用权；

1.2.2、计时器定时查询

仲裁控制器分别与设备1、设备2、设备3 连接；仲裁控制器有一个计数器，比如值为 1；

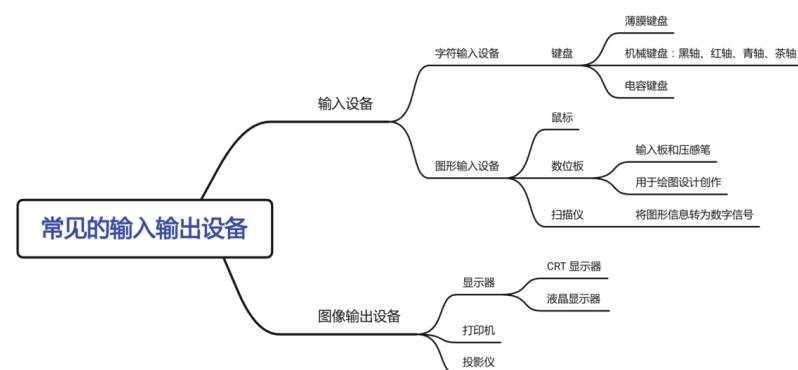


假设设备 2 需要使用总线，将通过仲裁控制线向仲裁控制器发出请求；

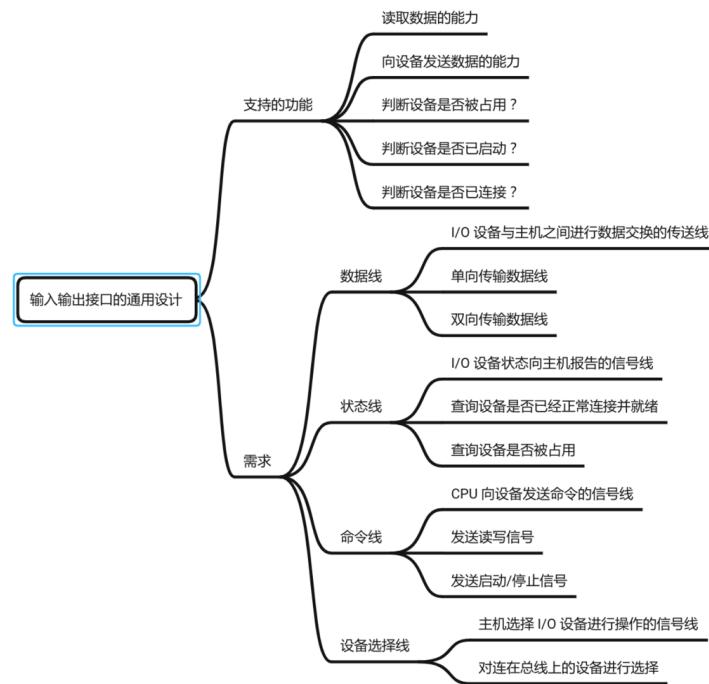
- 仲裁控制器会将当前的计数器值 1 发送给所有的设备；
- 设备 1 实际没有发出请求信号，所以仲裁控制器发出的计数器值 1 无用；
- 此时仲裁控制器会发出信号 2，此时设备 2 可以获得总线使用权；

2、计算机的输入与输出设备

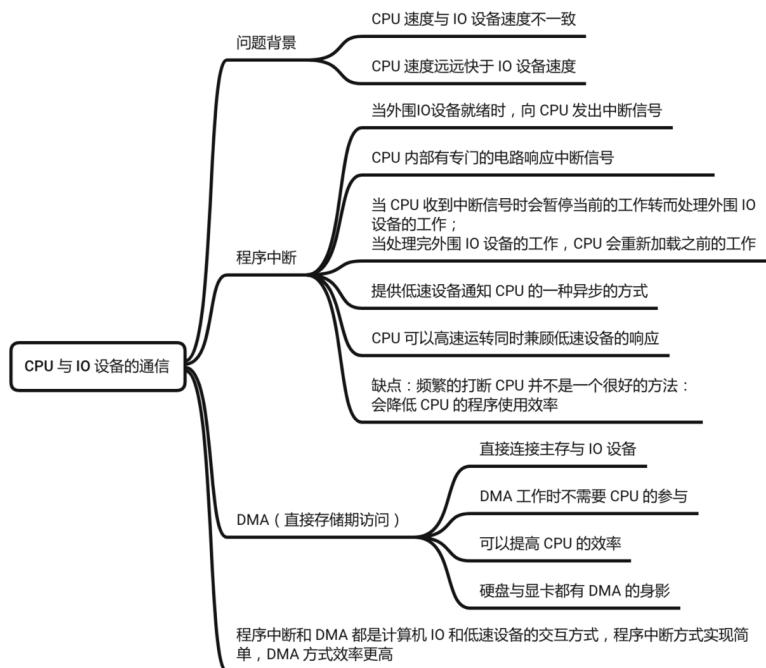
2.1、常见的输入输出设备



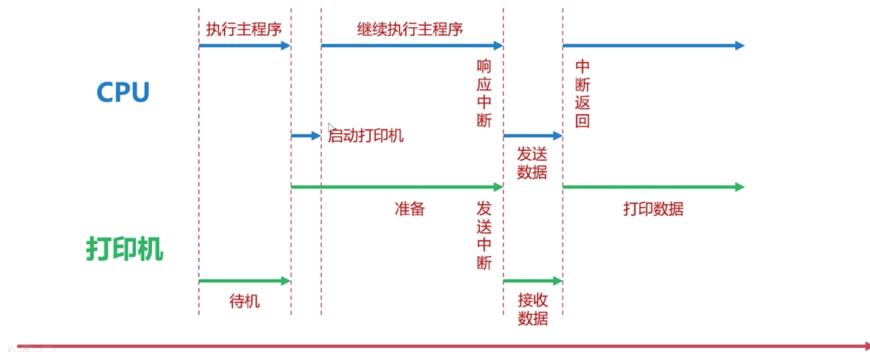
2.2、输入输出接口的通用设计



2.3、CPU 与 IO 设备的通信

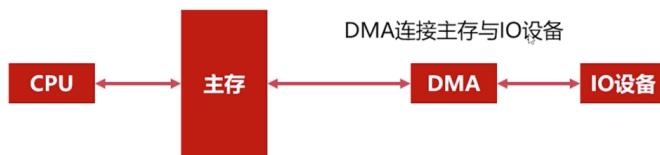


程序中断



- 在某一时刻，CPU 在执行主程序，打印机处于待机状态；
- 接着 CPU 发出了一个启动打印机的信号，发出信号后 CPU 继续执行主程序；
- 打印机在收到启动信号后，做一些准备工作来启动，准备工作完成后打印机向 CPU 发送一个中断信号；
- CPU 收到中断信号后，会响应中断信号（不是立即中断，可能延迟）
- 接着 CPU 发送数据给打印机，打印机接收数据；
- CPU 发送完数据，会中断返回，继续执行中断前执行的主程序；同时打印机也会开始打印数据；

DMA (直接内存存取)



当主存与IO设备交换信息时，不需要中断CPU

可以提高CPU的效率

- DMA (Direct Memory Access)：直接内存存取；
- 对于 主存与 IO 设备 并没有直接的连接；而是通过 DMA 设备连接；
- 当主存与 IO 设备交换信息时，不需要中断 CPU ；
- DMA 可以处理主存与 IO 设备交换信息的操作；
- IO 设备不用打断 CPU 的工作，因此 DMA 可以提高 CPU 的效率；

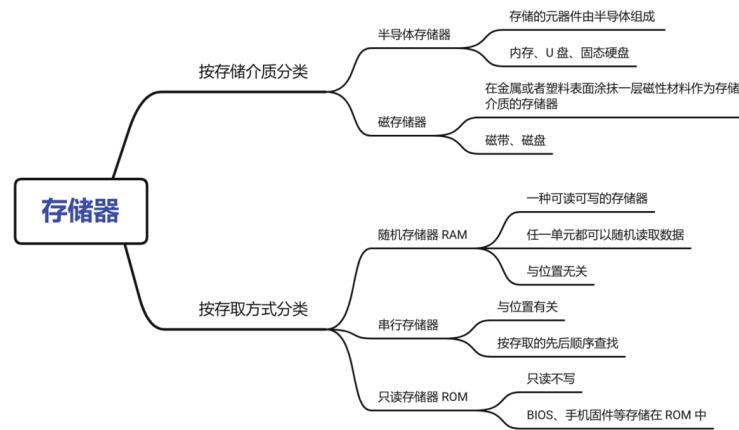
比较：程序中断和 DMA 都是计算机 IO 和低速设备的交互方式，程序中断方式实现简单，DMA 方式效率更高。

3、存储器

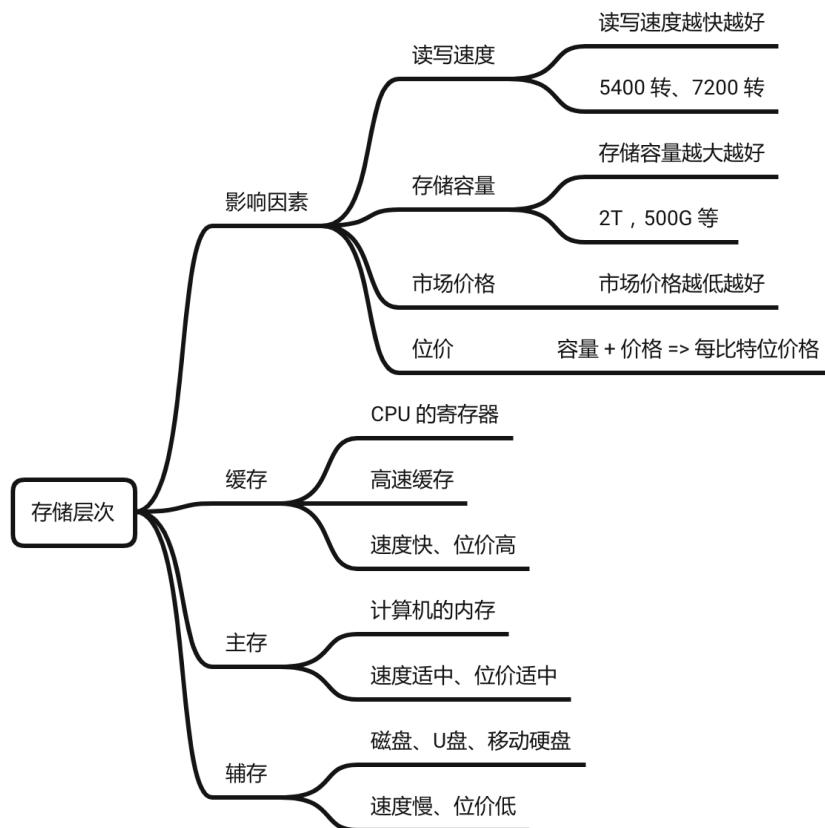
常见的存储器有以下几种：

- 主存：主存储器，即常说的计算机内存条，
- 辅存：辅助存储器，即常说的磁盘、U 盘、光盘、磁带等，
- Cache (高速缓存)：即常说的 L1、L2、L3 缓存，CPU 寄存器等，一般位于 CPU 上！

- RAM: 随机存取存储器 (Random Access Memory)
- ROM: 只读存储器 (Read Only Memory)



3.1、存储器的层次结构



存储器的层次结构可以简单划分为：缓存 - 主存 - 辅存 三个层次，缓存 - 主存 主要是为了解决主存速度不够的问题；主存 - 辅存 主要是为了解决主存容量不足的问题。



①②③④⑤⑥

3.1.1、缓存-主存 的层次

原理：局部性原理，指 CPU 访问存储器时，无论是存取指令还是存取数据，所访问的存储单元都趋于聚集在一个较小的区域中；实现：在 CPU 与主存之间增加一层速度快容量小的 Cache；目的：解决 CPU 与主存速度不匹配的问题！



存储器的层次结构

3.1.2、主存-辅存 的层次：

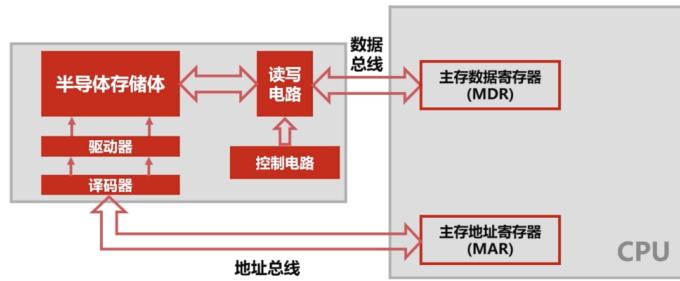
原理：局部性原理；实现：在主存之外增加辅助存储器（磁盘、SD 卡、U 盘等）
目的：为了解决主存容量不足的问题；例子：针对 8G 的 Mac 与 10 多 G 的 Xcode，Xcode 是如何运行到仅仅 8 G 的内存的？由于局部性原理，可以将一些必要数据加载到内存中，对于不使用的数据放到辅存中！

3.2、计算机的主存储器

主存储器 -> 内存

- 随机存取存储器 RAM (Random Access Memory)
- RAM 通过电容存储数据，必须隔一段时间刷新一次，刷新需要有电的存在；
- 如果掉电，那么一段时间后不能刷新，电容中的电子将丢失，即丢失所有数据；

3.2.1、主存储器如何与 CPU 的交互？



- CPU 中的主存数据寄存器 MDR 通过数据总线与主存储器中的读写电路交互；
- CPU 中的主存地址寄存器 MAR 通过地址总线与内存连接；
- 因为数据总线与地址总线的存在，CPU 可以通过地址总线指定数据的位置，同时通过数据总线传输相关的数据；

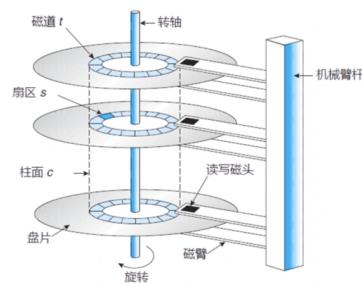
3.2.2、主存储器与操作系统的位数有关系

主存储器与操作系统位数有一定的关系，不同的操作系统对内存的支持是不同的：

- 32 位系统：最多支持 4 GB 的内存 ($2^{32} = 4 * 2^{30} = 4 \text{ GB}$)；在 32 位系统中加装更多的内存也是无用的，因为它的地址总线只有 32 位，寻址范围只有 4 GB 的大小；
- 64 位系统：最多支持 2^{34} GB 的内存 ($2^{64} = 2^{34} * 2^{30} = 2^{34} \text{ GB}$)；在 64 位系统中地址总线有 64 位，寻址范围有 2^{34} GB 大小；

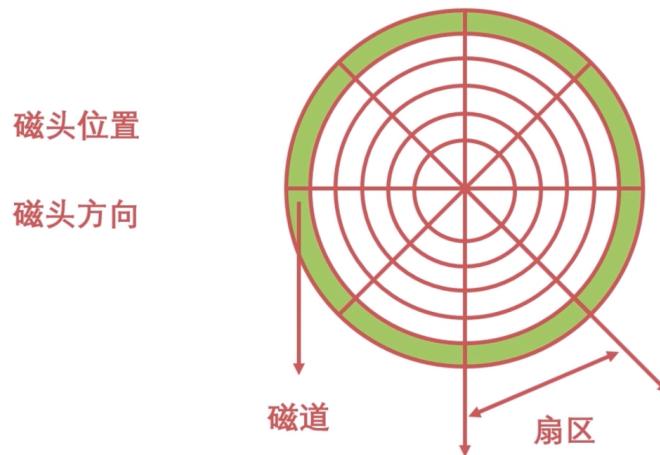
3.3、计算机的辅助存储器 - 磁盘

磁盘表面是可磁化的硬磁性材料，通过移动磁头径向运动读取磁道信息！



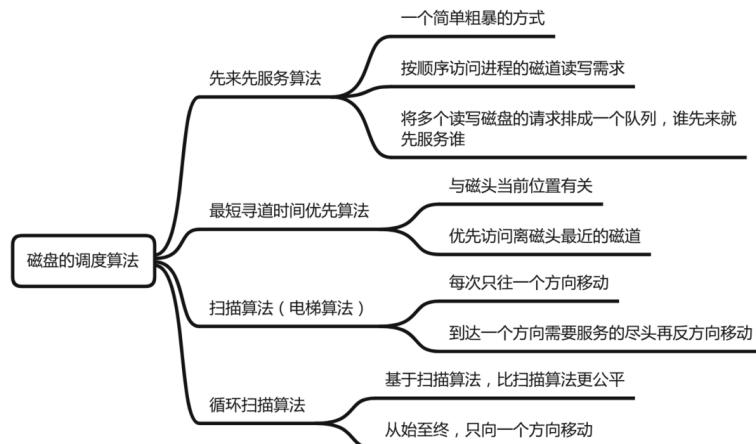
磁盘由两部分组成：

- 光滑的盘片：使用磁材料存储数据；
- 磁头（悬臂）：通过左右移动来读取特定地址的信息



- 磁头位置：当前磁头位于哪—个磁道；
- 磁头方向：磁头向里走或者向外走；

3.3.1、磁盘的调度算法



为方便计算，将最外层的磁道称为第 1 磁道，最里面的磁道称为第 n 磁道。

辅助存储器——磁盘

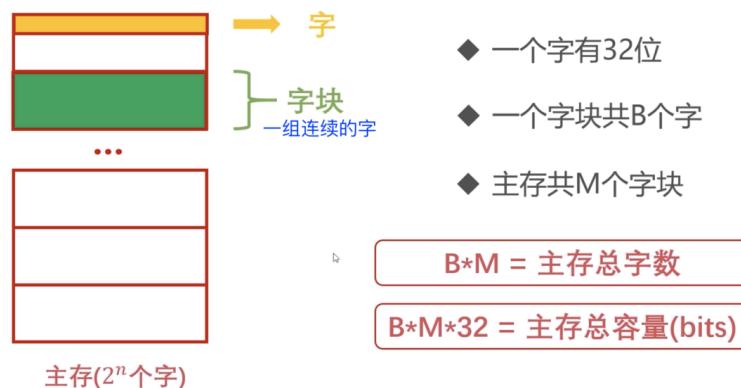


假设磁头位于磁道4，磁头方向由里向外，现在需要读取磁道 1、4、2、3、1、5！

- 先来先服务算法：磁头的移动痕迹 $4 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 5$ ！
- 最短寻道时间优先算法：磁头的移动痕迹 $4 \rightarrow 5 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 1$ ！
- 扫描算法（电梯算法）：磁头的移动痕迹 $4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 1 \rightarrow 5$ ！
- 循环扫描算法：磁头的移动痕迹 $4 \rightarrow 5 \rightarrow 1 \rightarrow 1 \rightarrow 2 \rightarrow 3$ ！

3.4、高速缓存的工作原理

3.4.1、字与字块

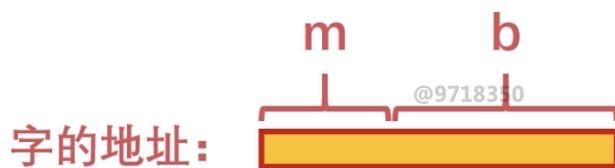


基本概念：

- 字：是指存放在一个存储单元中的二进制代码组合；是存储单元的最小单位；一个字可以表示一个数据、一个指令、一个字符串；
- 字块：存储在 **连续** 的存储单元中而被看作是一个单元的 **一组** 字；字块包含了多个字！

字的寻址

字的地址包括两个部分，分别由字块的部分（用来指示当前需要寻址的字是属于哪一个字块的）以及字的部分（用来寻找字块中哪一个字是这个地址所指定的字）组成！

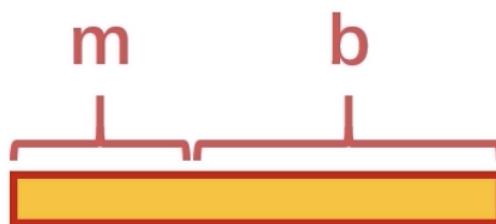


- ◆ 字的地址包含两个部分
- ◆ 前m位指定字块的地址
- ◆ 后b位指定字在字块中的地址

demo

假设主存空间为 4 G, 字块大小为 4 M, 字长位 32 位, 则对于字地址中的地址块 m 和块内地址 b 的位数, 至少应该是多少?

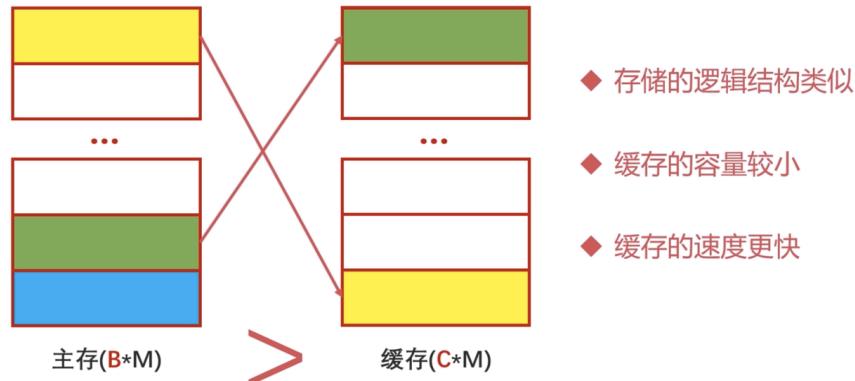
- ◆ 一个字有32位
- ◆ 一个字块共B个字
- ◆ 主存共M个字块



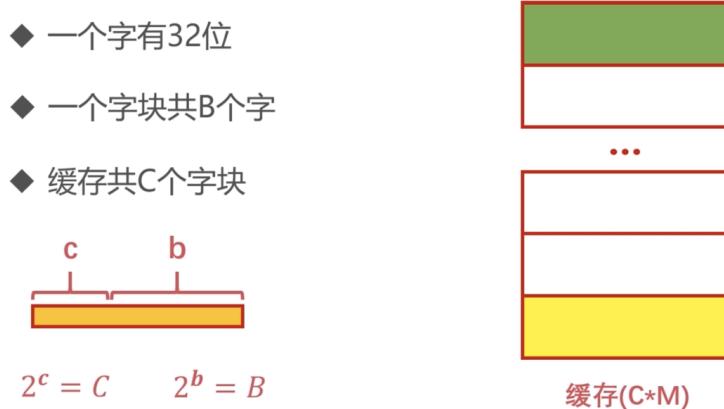
$$2^m = M \quad 2^b = B$$

```
4 G = 4096 M;  
字块数: 4096 / 4 = 1024  
/// 至少需要 10 位来表示 1024 个字块  
字块地址 m : log2(1024) = 10  
/// 每个字块内有 1048576 个字  
块内字数: 4 M / 32 bits = 1048576  
/// 至少需要 20 的块内地址来表示块内所有的字  
块内地址 b : log2(1048576) = 20  
因此: m >= 10 ; b >= 20;
```

3.4.2、高速缓存



- 主存的容量远远大于缓存的容量（主存的字块数远大于缓存的字块数）；
- 缓存存取的数据是对主存的某块数据的复制（缓存中的数据来自于主存）；



缓存是如何在 缓存-主存 的层次中工作的呢？



- ◆ CPU需要的数据在缓存里
- ◆ CPU需要的数据不在缓存里
- ◆ 不在缓存的数据需要去主存拿

命中率

CPU 需要高速缓存有两种情况：

- CPU 需要的数据在缓存中，CPU 直接高效快速的从高速缓存中拿到数据；

- CPU 需要的数据不在缓存中；CPU 从主存中拿数据；

高速缓存的读取速度比主存快很多，CPU 从主存拿数据，大大降低了 CPU 的效率；因此需要 CPU 尽可能的从高速缓存中读取数据，而非主存中取数据！此时需要一个量化的指标来量化 CPU 从高速缓存中取数据成功的机率（缓存命中率）！

缓存命中率是衡量缓存的重要性能指标；理论上 CPU 每次都能从高速缓存取数据的时候，命中率为 1（实际上永远不可能为 1）！假设访问主存 m 次，访问缓存 n 次，那么缓存命中率就是 $h = n / (m + n)$

假设访问主存时间为 T_m ，访问缓存时间为 T_c ，则访问缓存-主存 系统的平均时间为 $T_a = h * T_c + (1 - h) * T_m$ ！那么访问效率 $e = T_c / T_a = T_c / (h * T_c + (1 - h) * T_m)$ ！

demo

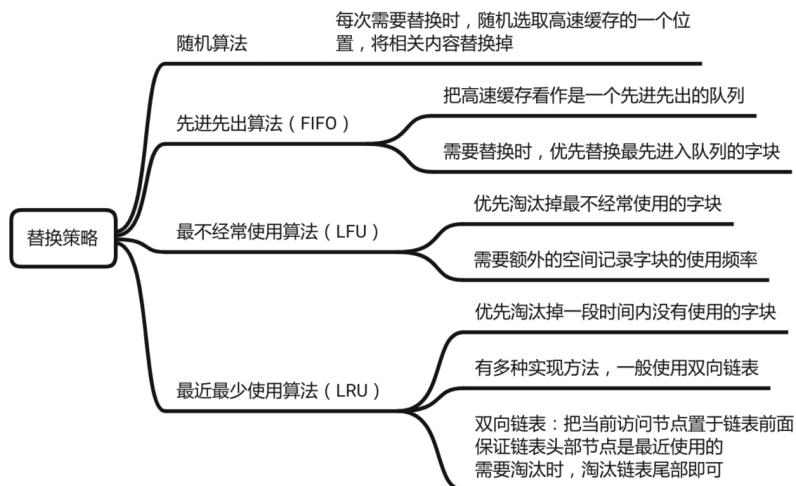
假设 CPU 在执行某段程序时，共访问了 Cache 命中了 2000 次，访问主存 50 次，已知缓存的存取时间为 50 ns，主存的存取时间为 200 ns，求 缓存-主存 系统的命中率、访问效率和平均访问时间？

$$\begin{aligned} \text{缓存命中率 } h &= 2000 / (2000 + 50) = 0.97; \\ \text{平均访问时间 } T_a &= 0.97 * 50 + (1 - 0.97) * 200 = 54.5 \text{ ns}; \\ \text{访问效率 } e &= 50 / 54.5 = 91.7\%; \end{aligned}$$

3.5、高速缓存的替换策略

为了使 CPU 的运行效率更高，需要缓存命中率越高越好，让 CPU 在每一次取数据时，都能从高速缓存中取数据，而不是从主存中取数据！此时我们需要一个良好的缓存替换策略，使得缓存中的数据都是 CPU 需要的数据！

替换时机：当 CPU 需要的数据不在高速缓存中时，需要从主存中载入所需的数据，然后替换到高速缓存中！



假设缓存 4 个字块，() 表示使用的字块，[] 表示淘汰的字块

最近最少使用算法 (LRU)

- (1) 1
- (2) 2、1
- (4) 4、2、1
- (7) 7、4、2、1
- (5) 5、7、4、2 [1]
- (4) 4、5、7、2
- (6) 6、4、5、7 [2]

4、计算机的指令系统

4.1、机器指令的形式

机器指令主要由 **操作码**、**地址码** 两部分组成；分为三地址指令、二地址指令和一地址指令！

- 操作码：指明指令所要完成的操作；操作码的位数反映了机器的操作种类，比如操作码有 8 位，则有 $2^8 = 256$ 种操作！
- 地址码：直接给出操作数或者操作数的地址；因为机器指令本质上还是对数据进行操作，所以地址码实际上还是指定数据或者数据的地址，使得 CPU 能够根据数据或者数据的地址进行运算；

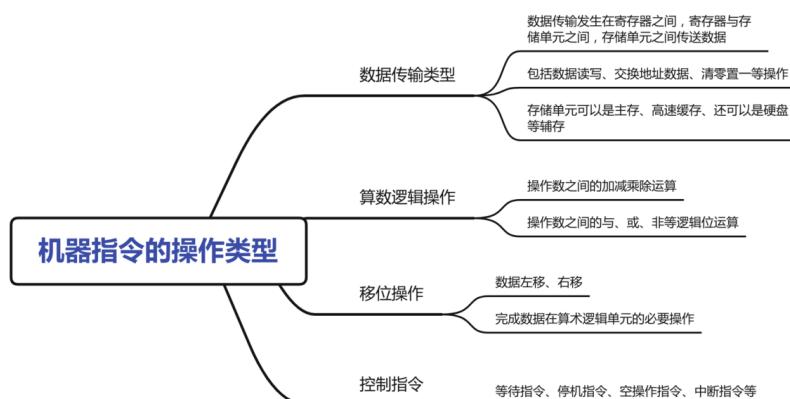
```
/// 三地址指令
操作码OP | addr1 | addr2 | addr3
(addr1)OP(addr2) -> (addr3)
如加法操作 (1)+(2) -> (3)

/// 二地址指令
操作码OP | addr1 | addr2
(addr1)OP(addr2) -> (addr1)或者(addr2) /// 把结果放到 addr1 或者 addr2

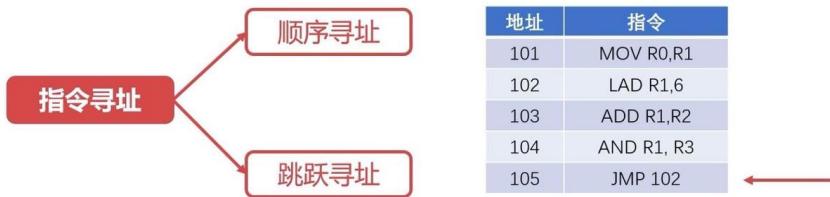
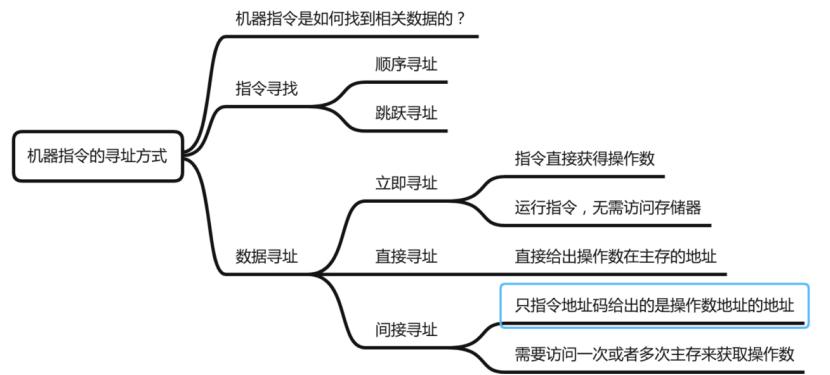
/// 一地址指令
操作码OP | addr1
(addr1)OP -> (addr1) /// 把结果放到 addr1 , 如自己对自己的操作
(addr1)OP(ACC) -> (addr1) /// 把结果放到 addr1
```

零地址指令，在机器指令中无地址码，空操作、停机操作、中断返回操作等！

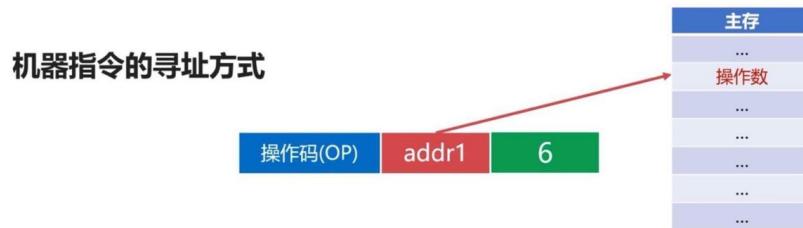
4.2、机器指令的操作类型



4.3、机器指令的寻址方式



- 顺序寻址：执行 101 指令、102 指令、103 指令、104 指令、105 指令；
- 跳跃寻址：105 指令指向 102 指令，因此跳跃到 102 指令！



- ◆ 直接给出操作数在主存的地址
- ◆ 寻找操作数简单，无需计算数据地址



- ◆ 指令地址码给出的是操作数地址的地址
- ◆ 需要访问一次或多次主存来获取操作数

间接寻址

数据寻址方式	优先	缺点
立即寻址	速度快（从机器指令直接拿到数据）	地址码位数 限制操作数表示范围 (数据存放在地址码中)
直接寻址	寻找操作数简单	地址码位数 限制操作数表示范围
间接寻址	操作数寻址范围大	速度较慢

5、计算机的控制器

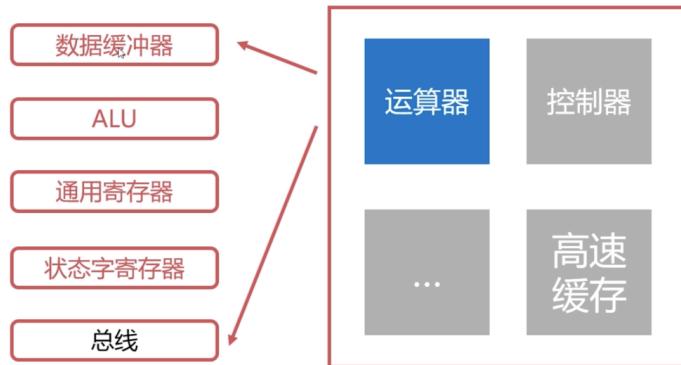
计算机的控制器用于协调和控制计算机运行！



- 程序计数器：用于存储下一条指令的地址；当 CPU 工作时程序计数器会循环不断的从计数器中拿出指令；当拿出一条指令后，会指向下一条指令；程序计数器 主要提供给其它控制单元当前指令执行的地址！
- 时序发生器：属于电气工程领域，用于发送时序 脉冲，CPU 依据不同的时序 脉冲有节奏的进行工作！
- 指令译码器：是控制器的重要部件之一，计算机指令由操作码与地址码组成，指令译码器会将操作码翻译为对应的操作数据、将地址码翻译为控制传输地址对应的数据！
- 指令寄存器：是 CPU 高效运转的重要部件之一，用于缓存从主存或者高速缓存取下来的计算机指令；当 CPU 需要执行相关指令时，就可以从指令寄存器取出相关的指令，而不需要从主存或者高速缓存去取！
- 主存地址寄存器：保存当前 CPU 正要访问的内存单元的地址，通过地址总线与主存通信的！
- 主存数据寄存器：保存当前 CPU 正要读、正要写的主存数据，通过数据总线与主存通信的！
- 通用寄存器：用于暂时存放或传送 数据与指令；也可以保存 ALU 的运算中间结果；容量比一般专用寄存器要大！

6、计算机的运算器

计算机的运算器主要用于数据的运算加工！



运算器是用来进行数据运算加工的

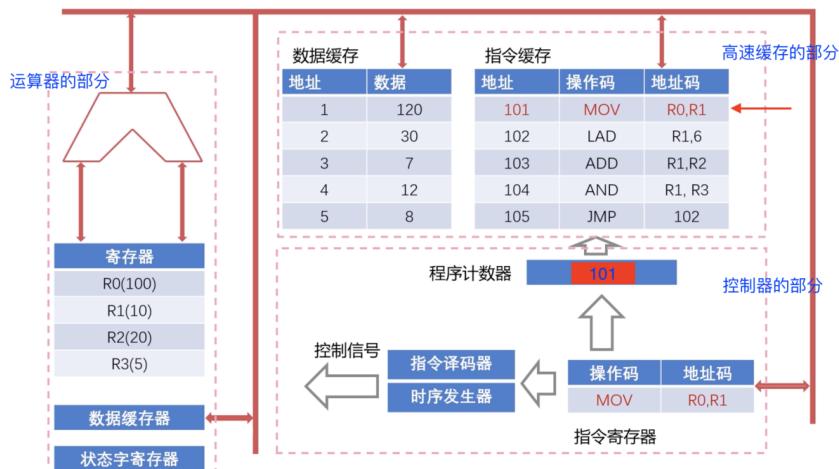
- 数据缓冲器：分为 输入缓冲与 输出缓冲；输入缓冲用于暂时存放外设传送来的数据，如果 ALU 正在运算则下一个运算数据保存在该缓冲中；输出缓冲暂时存放往外设的数据，当数据运算完毕从 ALU 输出时保存在输出缓冲中、等待控制器下一步的命令将数据送到相应位置！
- ALU（算数逻辑单元）：是运算器的主要组成，可以完成常见的位运算（左移、右移、或、与、非等），可以完成一些算术运算（加、减、乘、除等）！
- 状态字寄存器：用于存放运算中的状态（条件码、进位、溢出、结果正负等）；存放运算控制信息（调试跟踪标记位、允许中断位等）！
- 通用寄存器：用于暂时存放或传送 数据与指令；也可以保存 ALU 的运算中间结果；容量比一般专用寄存器要大！

7、计算机指令的执行过程

7.1、指令的执行过程

指令执行的一般操作：取指令 → 分析指令 → 执行指令！

- 取指令：从指令缓存中取指令，送到指令寄存器；
- 分析指令：将指令从指令寄存器取出，送到指令译码器译码；指令译码器发送控制信号，同时程序计数器 +1；
- 执行指令：首先将数据装载到寄存器，接着ALU 处理数据，记录运算状态，最后送出运算结果！



- 指令执行中涉及的设备：数据缓冲器、指令缓冲器、运算器（通用寄存器、数据寄存器、状态字寄存器）、程序计数器、指令译码器、时序发生器、指令寄存器；
- 运算器部分、控制器部分、高速缓存部分，通过片内总线连接起来；
- 执行指令时，首先发生数据缓存与指令缓存；将数据与指令缓存到 CPU 的高速缓存中；
- 接着程序计数器根据指令地址 101 ，程序计数器只知道指令的地址而不知道指令的具体内容，需要通过总线来到指令缓存中获取指令 $MOV R0, R1$ ；
- 通过片内总线来到指令寄存器，指令寄存器缓存具体内容 $MOV R0, R1$ ；
- 指令寄存器不知道相关内容，还需要将指令发送到指令译码器；同时程序计数器 $+1$ (102)；
- 指令译码器将指令译码，译码完成后理解指令的具体内容（将 $R0$ 的数据移到 $R1$ 中）；指令译码器发出控制信号；
- 控制信号通过片内总线来到运算器，运算器通过控制信号知道自己要干什么；
- 运算器首先将 $R0(100)$ 加载到 ALU 中，接着将 $R0(100)$ 通过总线送到数据缓存器；
- 数据缓存器会将 $R0(100)$ 覆盖到 $R1$ ，此时 $R1$ 的数据为 $R1(100)$ ！
- 完成该条指令后，CPU 又去执行下条指令！

疑问：在上述 取指令 与 分析指令 的过程中，是由控制器工作的；执行指令时，由运算器工作！也就是说，运算器与控制器不能同时工作，这就导致 **CPU 的综合利用率并不高**！因此需要改进指令执行过程，提高CPU 的综合利用率！

7.2、CPU 的流水线设计

CPU 的流水线设计类似于工厂的装配线，多个产品可以同时被加工；在同一时刻，不同产品位于不同的加工阶段！

指令	时间片	时间片	时间片	时间片	时间片	时间片	时间片	时间片	时间片
1	取指令	分析指令	执行指令						
2		取指令	分析指令	执行指令					
3			取指令	分析指令	执行指令				
4				取指令	分析指令	执行指令			
5					取指令	分析指令	执行指令		
6						取指令	分析指令	执行指令	
7							取指令	分析指令	执行指令

- 每一个时间片，都有一条或多条指令在执行；
- 如对于第 2 个时间片，第1条指令在分析，第 2 条指令在取指令；
- CPU 的流水线设计大大提升了CPU 的综合利用率！

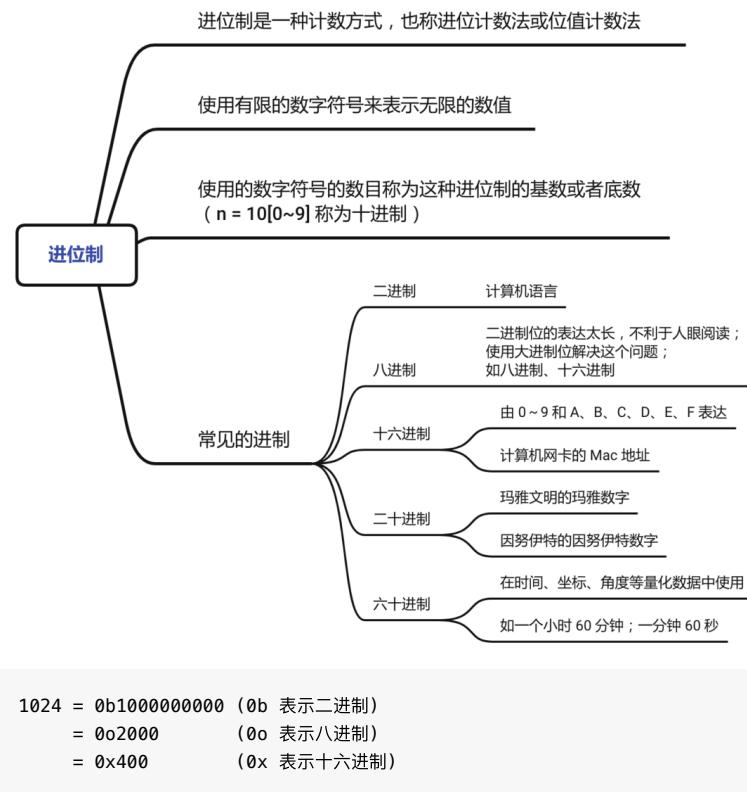
假设取指令、分析指令、执行指令的时间开销都是 t ：

- 串行执行 m 条指令： $T1 = 3t * m$ ；
- 流水线执行 m 条指令： $T2 = t * (m + 2)$ ；
- 流水线执行的效率 $H = T2 / T1 = (t * (m + 2)) / (3t * m)$ ；
- 当 m 很大时 流水线执行效率是串行执行的 3 倍！

计算机原理之计算篇

1、进制运算的基础

1.1、进制运算概述



1.2、二进制运算的基础

整数进制转换

```

/// 整数二进制转十进制：按权展开法
N = 01100101
= 1 * 2^6 + 1 * 2^5 + 1 * 2^2 + 1
= 101
N = 11101101
= 1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^3 + 1 * 2^2 + 1
= 237

/// 整数十进制转二进制：重复相除法 (01100101) (从下向上)
101 / 2 = 50 ... 1
50 / 2 = 25 ... 0
25 / 2 = 12 ... 1
12 / 2 = 6 ... 0
6 / 2 = 3 ... 0
3 / 2 = 1 ... 1
1 / 2 = 0 ... 1
  
```

浮点数进制转换

```

/// 浮点数二进制转十进制: 按权展开法
N = 0.11001
= 1 * 2^-1 + 1 * 2^-2 + 1 * 2^-5
= 0.78125
N = 0.01011
= 1 * 2^-2 + 1 * 2^-4 + 1 * 2^-5
= 0.34375

/// 浮点数十进制转二进制: 重复相乘法 (0.11001) (从上向下)
25/32 = 50/32 = 1 + 9/16 取 1
9/16 = 18/16 = 1 + 1/8 取 1
1/8 = 1/4 = 0 + 1/4 取 0
1/4 = 1/2 = 0 + 1/2 取 0
1/2 = 1/1 = 1 + 0 取 1

```

1.3、有符号数与无符号数

1.3.1、原码表示法

如何判断是数字位还是符号位? 原码表示法 使用 0 表示正数, 使用 1 表示负数, 规定符号位位于数值第一位; 表达简单明了, 是人类最容易理解的表示法!

在原码表示法之下, 0 有两种结果: 正 0 00 与负 0 10 ! 除此之外, 原码表示法进行运算也非常复杂, 特别是两个操作数符号不同的时候!

1.3.2、补码表示法

补码表示法: 使用正数代替负数! 但仍然没有实现 使用加法操作代替减法操作 的预期!

$$x = \begin{cases} x & 2^n > x \geq 0 \\ 2^{n+1} + x & 0 > x \geq -2^n \end{cases}$$

```

/// n = 4, x = 13; 计算 x 的二进制原码和补码
原码 x = 0 1101
补码 x = 0 1101

/// y = -13; 计算 y 的二进制原码和补码
原码 y = 1 1101
补码 2^(n+1) + y = 2^(4 + 1) - 13
= 10 0000 - 1101
= 10011
补码 y = 1 0011

/// z = -7; 计算 z 的二进制原码和补码
原码 z = 1 0111
补码 2^(n+1) + z = 2^(4 + 1) - 7
= 10 0000 - 0111
= 11001
补码 z = 1 1001

```

1.3.3、反码表示法

$$\text{反码表示法} \quad x = \begin{cases} x & 2^n > x \geq 0 \\ (2^{n+1}-1) + x & 0 > x \geq -2^n \end{cases}$$

$$\text{补码表示法} \quad x = \begin{cases} x & 2^n > x \geq 0 \\ 2^{n+1} + x & 0 > x \geq -2^n \end{cases}$$

/// n = 4, y = -13; 计算 y 的二进制原码、补码、反码

原码 y = 1 1101

$$\begin{aligned} \text{补码 } 2^{(n+1)} + y &= 2^{(4+1)} - 13 \\ &= 10\ 0000 - 1101 \\ &= 10011 \end{aligned}$$

补码 y = 1 0011

$$\begin{aligned} \text{反码 } 2^{(n+1)} - 1 + y &= 2^{(4+1)} - 1 - 13 \\ &= 011111 - 1101 \\ &= 10010 \end{aligned}$$

反码 y = 1 0010

/// z = -7; 计算 z 的二进制原码和补码

原码 z = 1 0111

$$\begin{aligned} \text{补码 } 2^{(n+1)} + z &= 2^{(4+1)} - 7 \\ &= 10\ 0000 - 0111 \\ &= 11001 \end{aligned}$$

补码 z = 1 1001

$$\begin{aligned} \text{反码 } 2^{(n+1)-1} + z &= 2^{(4+1)} - 1 - 7 \\ &= 011111 - 0111 \\ &= 11000 \end{aligned}$$

反码 z = 1 1000

十进制	原码	补码	反码
13	0 1101	0 1101	0 1101
-13	1 1101	1 0011	1 0010
-7	1 0111	1 1001	1 1000
-1	1 0001	1 1111	1 1110

- 负数的反码等于原码（除符号位）按位取反！

- 负数的补码等于其反码 + 1

/// n = 4, y = -13; 计算 y 的二进制原码、补码、反码

原码 y = 1 1101

反码 y = 1 0010

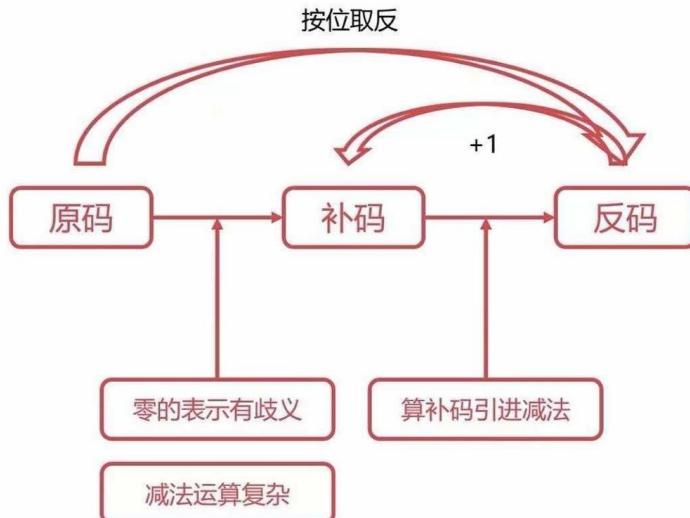
补码 y = 1 0011

/// z = -7; 计算 z 的二进制原码、补码、反码

原码 z = 1 0111

反码 z = 1 1000

补码 z = 1 1001



1.3.4、小数的补码

$$x = \begin{cases} x & 2^n > x \geq 0 \\ 2^{n+1} + x & 0 > x \geq -2^n \end{cases}$$

二进制整数的补码

$$x = \begin{cases} x & 1 > x \geq 0 \\ 2 + x & 0 > x \geq -1 \end{cases}$$

二进制小数的补码

```

/// n = 4, x = 9/16; 计算 x 的二进制原码、补码、反码
原码 x = 0 0.1001 (重复相乘法)
反码 x = 0 0.1001
补码 x = 0 0.1001

/// y = -11/32; 计算 y 的二进制原码、补码、反码
原码 y = 1 0.01011 (重复相乘法)
反码 y = 1 1.10100
补码 y = 1 1.10101

```

2、定点数与浮点数

2.1、定点数的表示方法

小数点固定在某个位置的数称为定点数！



2.2、浮点数的表示方法

计算机处理的很大程度上不是纯小数或纯正数；数据范围很大，定点数难以表达；此时需要使用浮点数来表示！

2.2.1、浮点数的表示格式

```
科学计数法
1 2345 0000 = 1.2345 * 10^8
/// 尾数 1.2345
/// 基数 10
/// 阶码 8
/// 科学计数法要求尾数的绝对值范围在 [1,10) 之间
```

浮点数在计算机存储中分为 4 个部分 (规格要求：尾数使用纯小数，且尾数最高位必须是 1) !

```
浮点数的表示格式 N = S * r^j
11.0101 = 0.110101 * 2^10
11.0101 = 0.0110101 * 2^11 //不符合要求：尾数最高位不是 1
11.0101 = 1.10101 * 2^1 // 不符合要求：尾数不是纯小数
```

阶码符号位	阶码数值位	尾数符号位	尾数数值位 (8位)
0	10	0	11010100

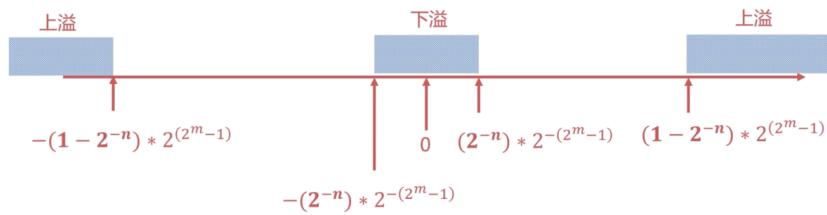
2.2.2、浮点数的表示范围

假设阶码数值取 m 位，尾数数值取 n 位， $N = S * r^j$!

- 阶码能够表示的最大值： $2^m - 1$ ；考虑有符号时 $[-(2^m - 1), 2^m - 1]$
- 尾数能够表示的最大值： $1 - 2^{-n}$ ；尾数全为 1 的时候是最大值；
- 尾数能够表示的最小值： 2^{-n} ；尾数全为 0 (除了最后一位是 1) 的时候是最小值；
- 尾数表示范围： $[2^{-n}, 1 - 2^{-n}]$ ；考虑有符号时 $[-(1 - 2^{-n}), -2^{-n}]$ ， $[2^{-n}, 1 - 2^{-n}]$ ；

阶码表示范围: $[-(2^m - 1), 2^m - 1]$

尾数表示范围: $[-(1 - 2^{-n}), -(2^{-n})] \quad [2^{-n}, 1 - 2^{-n}]$



超出浮点数的表示，有可能导致数据溢出：

- 上溢：绝对值太大，导致没有办法表示这么大的数；
- 下溢：绝对值太小，导致没有办法表示这么小的数；

单精度浮点数 `float`：使用 4 个字节 (32 位) 来表示浮点数; 双精度浮点数 `double`：使用 8 个字节 (64 位) 来表示浮点数;

Demo 1：假设浮点数字长位 16 位，阶码为 5 位，尾数为 11 位，将十进制数 $\frac{13}{128}$ 表示为二进制浮点数

原码=反码=补码 $x = 0.0001101000$ (重复相乘法)
浮点数规格化 $x = 0.1101000 * 2^{-11}$

阶码符号位	阶码数值位	尾数符号位	尾数数值位 (11位)
1	0011	0	1101000000

Demo 2：假设浮点数字长位 16 位，阶码为 5 位，尾数为 11 位，将十进制数 -54 表示为二进制浮点数

原码 $x = 1110110$ (重复相除法)
浮点数规格化 $x = -0.110110 * 2^{110}$

阶码符号位	阶码数值位	尾数符号位	尾数数值位 (11位)
0	0110	1	0010100000

2.3、定点数与浮点数的对比

- 当定点数与浮点数位数相同时，浮点数表示的范围更大；
- 当浮点数的尾数为规格化的小数时，浮点数的精度更高；
- 浮点数的运算包含阶码和尾数，浮点数的运算更为复杂；
- 浮点数在数的表示范围、精度、溢出处理、编程等方面均优于定点数；
- 浮点数在数的运算规则、运算速度、硬件成本等方面不如定点数；

2.4、定点数的加法与减法

2.4.1、定点数的加法计算

数值位与符号位一同运算，并将符号位产生的进位自然丢掉：

- 整数加法: $A[\text{补}] + B[\text{补}] = [A + B][\text{补}] (\bmod 2^{n+1})$
- 小数加法: $A[\text{补}] + B[\text{补}] = [A + B][\text{补}] (\bmod 2)$

```

/// 例1: A = -110010, B = 001101, 求 A + B
A[反] = 1 001101 A[补] = 1 001110
B[反] = 0 001101 B[补] = 0 001101
[A + B][补] = 1 011011
[A + B][反] = 1 011010
A + B = -100101

/// 例2: A = -0.1010010, B = 0.0110100, 求 A + B
A[反] = 1 1.0101101 A[补] = 1 1.0101110
B[反] = 0 0.0110100 B[补] = 0 0.0110100
[A + B][补] = 1 1.1100010
[A + B][反] = 1 1.1100001
A + B = -0.0011110

/// 例3: A = -10010000 (-144), B = -01010000 (-80), 求 A + B
A[反] = 1 01100000 A[补] = 1 01110000
B[反] = 1 10100000 B[补] = 1 10110000
[A + B][补] = 1 00100000
[A + B][反] = 1 00010000
A + B = -11100000 (-224)

/// 例4: A = -10010000 (-144), B = -11010000 (-208), 求 A + B
A[反] = 1 01100000 A[补] = 1 01110000
B[反] = 1 00100000 B[补] = 1 00110000
[A + B][补] = 0 10100000
[A + B][反] = 0 10100000
A + B = 10100000 (160) //? ? 发生了溢出操作: A + B 使用 8 位不够存储

```

判断溢出的方法:

- 双符号位判断法: 单符号位表示成双符号位, 运算时双符号位产生的进位丢弃; 如果结果的双符号位不同, 则表示溢出!

```

/// 例3: A = -10010000 (-144), B = -01010000 (-80), 求 A + B
A[反] = 11 01100000 A[补] = 11 01110000
B[反] = 11 10100000 B[补] = 11 10110000
[A + B][补] = 11 00100000
[A + B][反] = 11 00010000
A + B = -11100000 (-224)

/// 例4: A = -10010000 (-144), B = -11010000 (-208), 求 A + B
A[反] = 11 01100000 A[补] = 11 01110000
B[反] = 11 00100000 B[补] = 11 00110000
[A + B][补] = 10 10100000
[A + B][反] = 10 10100000
A + B = 10100000 (160) //? ? 发生了溢出操作: A + B 使用 8 位不够存储

```

2.4.2、定点数的减法计算

将减法转为上述的加法操作:

- 整数减法: $A[\text{补}] - B[\text{补}] = A[\text{补}] + (-B)[\text{补}] = [A + B][\text{补}] (\bmod 2^{n+1})$
- 小数减法: $A[\text{补}] - B[\text{补}] = A[\text{补}] + (-B)[\text{补}] = [A + B][\text{补}] (\bmod 2)$

$(-B)[\text{补}]$ 等于 $B[\text{补}]$ 连同符号位按位取反, 末位加一!

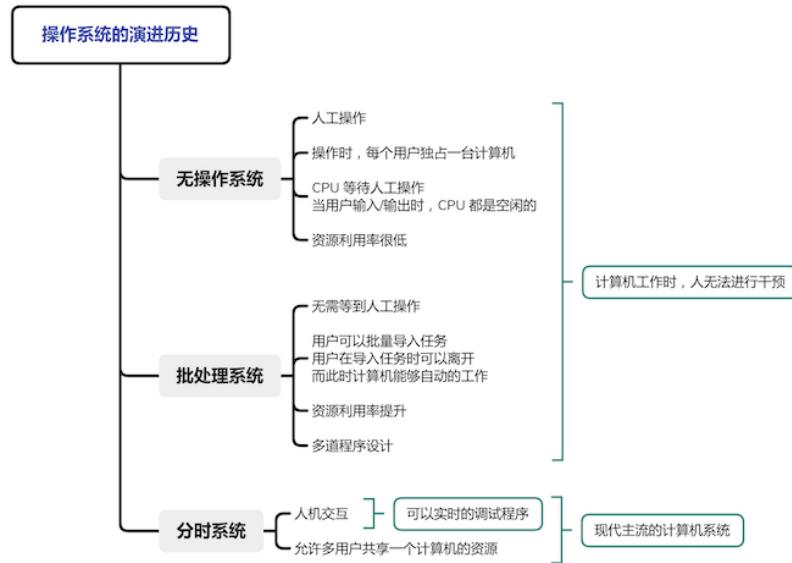
```
B[补] = 10010101  
(-B)[补] = 01101011
```

2.4、浮点数的加法与减法

一般步骤： 对阶 \rightarrow 尾数求和 \rightarrow 尾数规格化 \rightarrow 舍入 \rightarrow 溢出判断 !

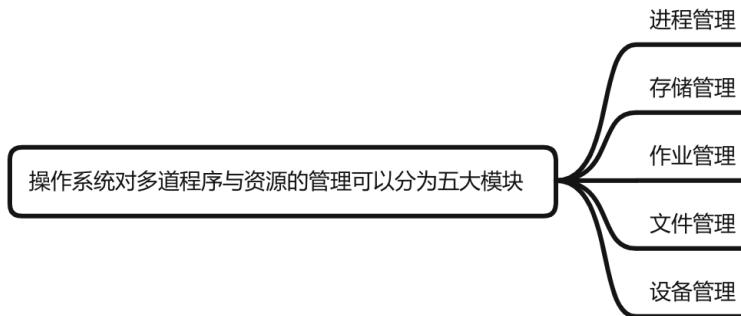
- 对阶：保证两个浮点数的阶码一致，使得尾数可以进行计算！阶码按照小阶看齐大阶的原则；
- 尾数规格化：对补码进行规格化需要判断两种情况： $s > 0$ 或 $s < 0$!

操作系统基础篇



多道程序设计：是指计算机内存中同时存放多个程序，并且这些程序互不干扰；

- 早起 批处理系统一次只能处理一个任务；
- 多道程序设计使得 批处理系统 可以一次处理多个任务；
- 多道程序在计算机的管理程序之下相互穿插运行；
- 因此，对多道程序的管理是操作系统的重要功能；



1、操作系统概览

1.1、什么是操作系统？为什么使用操作系统？

操作系统是管理硬件、提供用户交互的软件系统！

操作系统是管理计算机硬件和软件资源的 计算机程序；通过管理配置内存、决定资源供需顺序、控制输入输出设备等方法管理硬件资源！同时，操作系统也提供了让用户和系统交互的操作界面！

操作系统的种类是多种多样的，不局限于计算机；从手机到超级计算机，都有操作系统的存在（Android、iOS、Windows、Linux、MacOS）！在不同的设备，操作系统可以简单也可以复杂，向用户呈现多种操作手段（手机的触控操作）！

为什么使用操作系统?

- 我们不可能直接的操作计算机硬件 (如告诉 CPU 计算 $1 + 1$)
- 设备种类繁多复杂, 需要操作系统为用户提供了统一的界面, 屏蔽不同设备的差异;
- 操作系统的简易性使得更多的人可以使用计算机;

1.2、操作系统的基本功能

计算机的硬件资源

- 处理器资源 (CPU 资源)
- 存储器资源 (内存、硬盘)
- IO 设备资源 (打印机等)
- 文件资源

基本功能一：操作系统统一管理着计算机资源

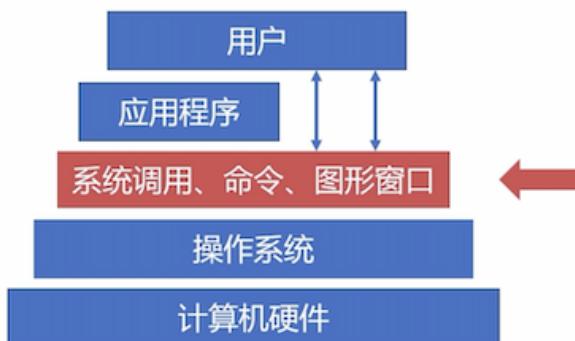
- 如用户需要操作某个文件, 并不是直接操作该文件的地址, 而是通过操作系统来访问!
- 如存储器资源, 在用户读取数据或者写入数据, 并不是直接控制存储器的设备读写, 而是通过操作系统去管理和读写的!
- 如处理器资源, 也不是直接告诉 CPU 需要计算的内容, 而是由操作系统来翻译需做的任务!

基本功能二：用户无需面向硬件接口编程

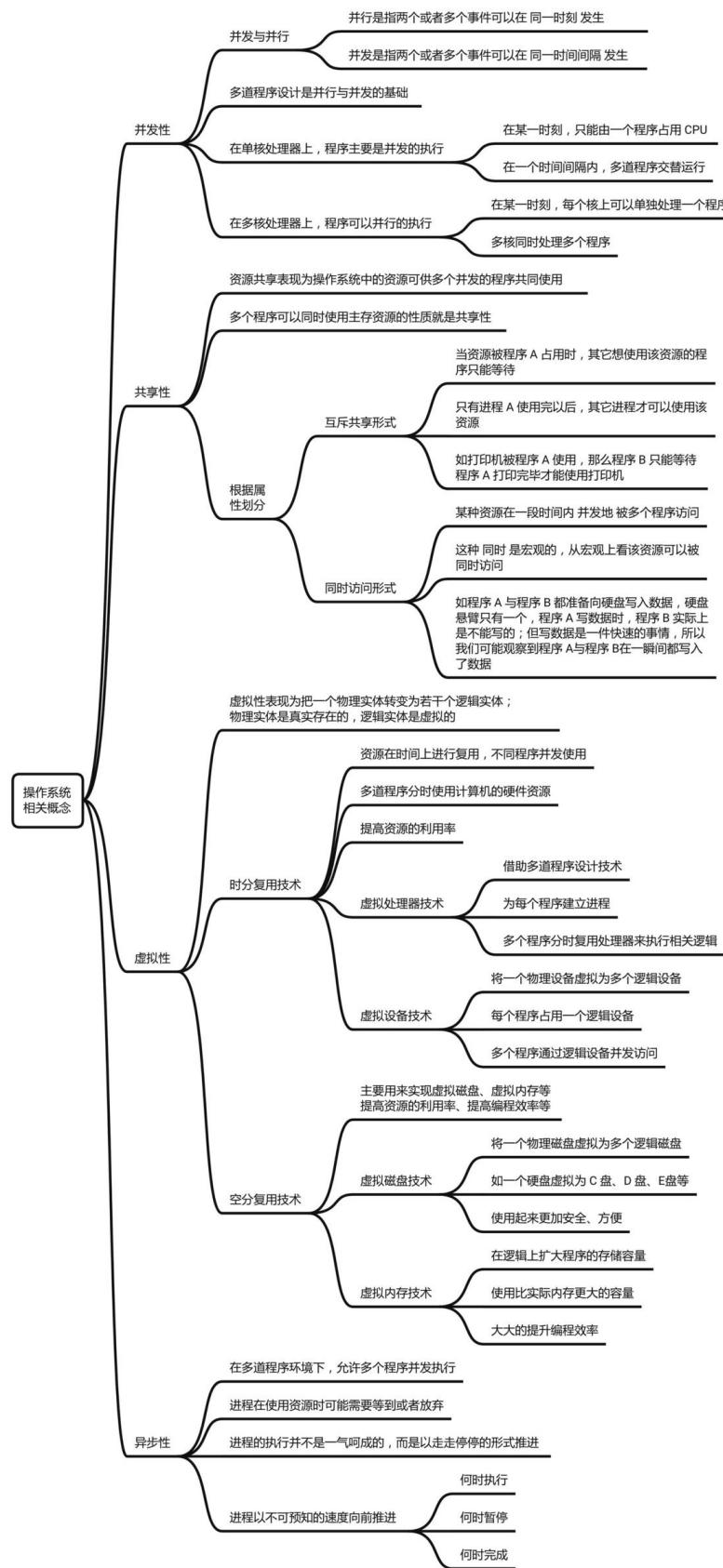
- 操作系统的 IO 设备管理软件, 提供读写接口;
- 文件管理软件, 提供操作文件的接口;
- 操作系统实现了对计算机资源的抽象; 通过管理软件来实现抽象! 管理软件屏蔽了硬件设备, 向上向用户提供了逻辑设备, 使得每个用户都使用相同的逻辑!

基本功能三：操作系统提供了用户和计算机之间的接口

- 图像窗口形式
- 命令形式
- 系统调用形式



1.3、操作系统的相关概念



2、进程管理之进程实体

2.1、为什么需要进程

在没有操作系统的年代，计算机只能运行一个程序，计算机资源属于当前运行的程序；有了操作系统之后，引入了多道设计的概念，通过合理的隔离资源、运行环境，提升资源利用率！

- 进程是系统进行资源分配和调度的基本单位；
- 进程作为独立运行的载体，保障程序正常执行；（多个进程可能有多个程序使用一个共同的设备、如存储器、CPU）
- 进程的存在使得操作系统资源的利用率大幅度提升；

2.2、进程的实体：主存中的进程形态

在主存中，进程也是一段连续存储的空间（进程控制块）：

进程控制块元素	细节
标识符	唯一标记一个进程，用于区别其它进程（如进程 ID 就是唯一标识符）
状态	标记进程的状态，如运行态、阻塞态
程序计数器	进程即将被执行的下一条指令的地址
内存指针	程序代码、进程数据相关指针；可能有多个内存指针分别指向程序的逻辑代码；
上下文数据	进程执行时处理器存储的数据
IO 状态信息	被进程 IO 操作所占用的文件列表
记账信息	存储进程使用处理器的时间、时钟数总和等；
优先级	
...	...

进程的实体由四种概念：进程标识符、处理机状态、进程调度信息、进程控制信息构成；

进程控制块 PCB

- 用于描述和控制进程运行的通用数据结构；
- 用于记录进程当前状态和控制进程运行的全部信息；
- PCB 是进程能够独立运行的基本单位；每个进程都依赖于进程控制块，被操作系统调度；
- PCB 是操作系统进行调度经常会被读取的信息；
- PCB 是常驻内存的，存放在系统专门开辟的 PCB 区域内；

2.3、进程与线程

- 进程 Process：是系统进行资源分配和调度的基本单位；
- 线程 Thread：是操作系统进行运行调度的最小单位；
- 一个进程可以有多个线程；

- 操作系统对进程的调度，实质上是对进程里面线程的调度；
- 线程包含在进程之中，是进程中实际运行工作的单位；
- 一个进程可以并发多个线程，每个线程执行不同的任务；
- 进程中的线程共享进程资源；

对比	进程	线程
资源	资源分配的基本单位	不拥有资源
调度	独立调度的基本单位	独立调度的最小单位
系统开销	进程系统开销大（管理线程、分配资源）	线程系统开销小
通信	进程 IPC	读写同一进程数据通信

3、进程管理之五状态模型

进程在系统中是有多个状态的，主要有 就绪、阻塞、执行、创建、终止 五个状态！

就绪状态：

- 当进程被分配到除 CPU 以外所有必要的资源后处于就绪状态；
- 此时只要再获得 CPU 的使用权，就可以立即执行；
- 其它资源（包括进程控制块，堆空间、栈空间等内存，）都准备好、只差 CPU 资源的状态为就绪状态；
- 在一个系统中多个处于就绪状态的进程通常排成一个队列（就绪队列）；

执行状态：

- 进程获得 CPU，其程序正在执行的状态；
- 在单处理机中，在某个时刻只能有一个进程处于执行状态；

阻塞状态：

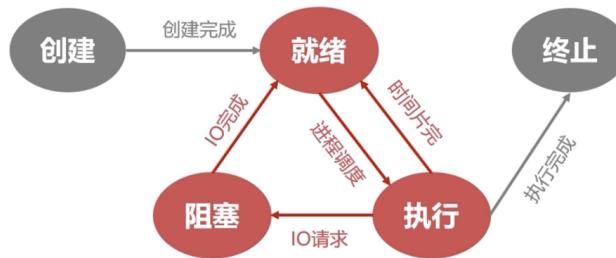
- 进程由于某些原因从而放弃 CPU 的状态称为阻塞状态
- 如进程需要获取某个设备，而对应的设备没有就绪导致进程无法继续执行；
- 如某个进程需要使用打印机，而打印机属于外围的 IO 设备、速度比较慢，进程在请求使用打印机后可能没有立刻得到反馈，这时候进程无法进行下一步工作，由于打印机的未就绪而处于阻塞状态；
- 阻塞队列：在操作系统中可能有一个或者多个阻塞进程；

创建状态：分配 PCB -> 插入就绪队列

- 创建进程时拥有 PCB，但其它资源尚未就绪的状态称为创建状态；
- 操作系统提供了 `fork()` 函数接口供程序员手动创建进程；

终止状态；系统清理 -> 归还 PCB

- 进程结束由系统清理或者归还 PCB 的状态称为终止状态；



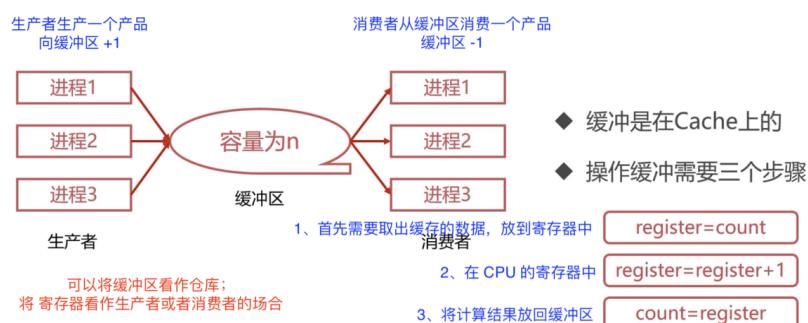
- 就绪 -> 执行状态：当就绪状态的进程发生 进程调度 时，就可以变为执行状态；
- 执行 -> 就绪状态：当执行状态的进程 CPU 资源（时间片）用完时，又会切换为就绪状态，插入到就绪队列中去；时间片资源指分配给某个进程执行的 CPU 时间段；
- 执行 -> 阻塞状态：进程发生 IO 请求时，可能变为阻塞状态；
- 阻塞 -> 就绪状态：当 IO 完成时，进程由阻塞状态切换为就绪；

4、进程管理之进程同步

4.1、为什么需要进程间同步？

question 1：生产者-消费者问题

有一群生产者进程在生产产品，并将这些产品提供给消费者进程进行消费；生产者进程和消费者进程可以并发执行，在两者之间设置了一个具有 n 个可缓冲区的缓冲池；生产者进程需要将所生产的产品放到一个缓冲区中，消费者进程可以从缓冲区取走产品消费。



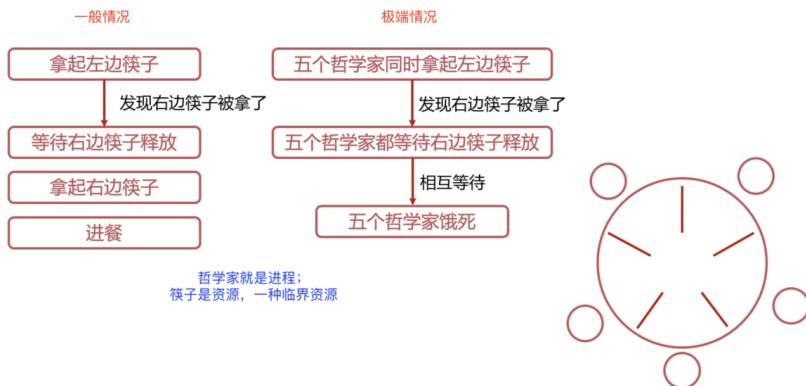
在现实的宏观生活中，上述的生产者-消费者模型没有任何问题，但是针对计算机的微观世界而言，是有一些问题的！

- 单从生产者程序或者消费者程序去看是没有任何问题的；
- 但如果两者并发执行时，可能出错！



question 2 : 哲学家进餐问题

有五个哲学家，他们的生活方式是交替地进行思考和进餐，哲学家们共同使用一张圆桌子，分别坐在周围的五张椅子上，在圆桌上有五个碗和五支筷子。平时哲学家们只进行思考，饥饿时则试图取靠近他们的左、右两支筷子，只有两支筷子都被拿到的时候才能进餐，进餐完毕后放下筷子左右思考！



上述两个模型的根源问题是彼此之间没有互相通信，如果生产者通知消费者我已经完成了一件生产、或者哲学家向旁边的哲学家说我要进餐了你们别拿我的筷子！因此需要进程间的同步！

进程间的同步可以对竞争资源在多进程间进行使用次序的协调；使得并发执行的多个进程之间可以有效使用资源和相互合作！

4.2、进程同步的原则

临界资源：指一些虽作为共享资源却又无法同时被多个线程共同访问的共享资源。当有进程在使用临界资源时，其它进程必须依据操作系统的同步机制等待占用进程释放该共享资源才可重新竞争使用共享资源！

为了对临界资源更好的约束，提出了四条原则

- 空闲让进：如果临界资源没有被占用、操作系统应该允许某个进程使用该临界资源；
- 忙则等待：如果有进程占用临界资源，此时应该防止别的进程使用该资源；使请求进程等待该临界资源的释放；
- 有限等待：在忙则等待的基础上，如果临界资源被占用，需要保证在有限等待时间内别的等待进程能够使用到资源，以避免外部等待进程僵死；

- 让权等待：当外部进程等待时，等待进程需要让出 CPU（进程从执行状态转为阻塞状态），保证 CPU 可以高效利用；

进程间同步的方法有：消息队列、共享存储、信号量。

4.3、线程同步

进程中的线程共享进程资源，当进程中的多线程并发的访问进程的资源时，可能发生数据竞态的问题；所以进程内多线程也需要同步！

线程同步的方法有：

- 互斥量：
- 读写锁：应对多读少写、少读多写的情况使用；
- 自旋锁：
- 条件变量：

5、Linux 的进程管理

5.1、Linux 进程的相关概念

进程的类型

- 前台进程：具有终端、可以和用户交互的进程；
- 后台进程：基本上不和用户交互，优先级比前台进程低；
- 守护进程：一种特殊的后台进程；很多守护进程在系统启动引导的时候启动，一直运行直到系统关闭；如进程名字以 `d` 结尾的一般都是守护进程：`crond`、`httpd`、`sshd`、`mysqld`！

进程的标记

- 进程 ID：进程的唯一标记，每个进程拥有不同的 ID；是一个非负整数，最大值由操作系统限定；
- ID 为 0 的进程为 `idle` 进程，是系统创建的第一个进程；
- ID 为 1 的进程为 `init` 进程，是 0 号进程的子进程，完成系统初始化；`init` 进程是所有用户进程的祖先进程；

```
/// 进程 ID 相关的关系：父子关系
进程A调用 fork() 函数创建进程B;
进程B调用 fork() 函数创建进程C;
进程A是进程B的父进程，进程B是进程A的子进程;

/// 进程的父子关系可以通过 pstree 命令来查看
MacBook-Pro:~ $ pstree
-+= 00001 root /sbin/launchd
|--- 00064 root /usr/sbin/syslogd
|--- 00065 root /usr/libexec/UserEventAgent (System)
|-+= 00073 root /usr/sbin/systemstats --daemon
|-+= 00075 root /usr/libexec/configd
|--- 00077 root endpointsecurityd
|--- 00081 root /usr/libexec/remoted
|--- 00083 root /usr/libexec/logd
|--- 00097 root /usr/libexec/kernelmanagerd
|--- 00098 root /usr/libexec/diskarbitrationd
|--- 00101 root /usr/libexec/coreduetd
```

进程的状态标记	状态说明
R	TASK_RUNNING 进程处于运行状态
S	TASK_INTERRUPTIBLE 进程处于睡眠状态
D	TASK_UNINTERRUPTIBLE 进程处于 IO 等待的睡眠状态
T	TASK_STOPPED 进程处于暂停状态
Z	TASK_DEAD / EXIT_ZOMBIE 进程处于退出状态，或僵尸进程

5.2、操作 Linux 进程的相关命令

- ps 命令：查看当前进程的相关信息；配合 aux 参数或 ef 参数和 grep 命令检索特定进程；
- top 命令：查看一些使用内存等
- kill 命令：发送特定信号给进程，kill -l 可以查看操作系统支持的信号；只有 kill 9 -0000 可以无条件终止进程，其它信号进程有权忽略；

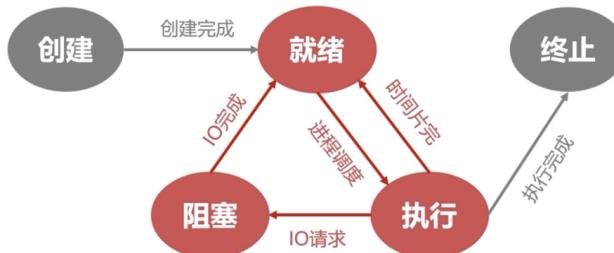
```
MacBook-Pro:~ $ ps
PID TTY      TIME CMD
21187 ttys001    0:00.01 -bash
```

6、作业管理之进程调度

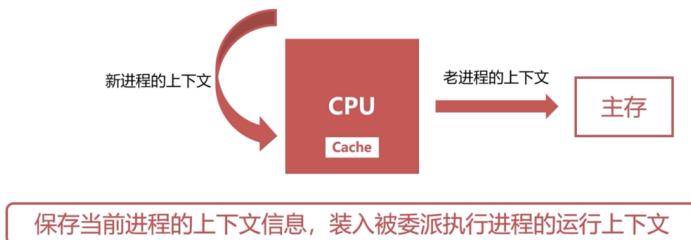
6.1、进程调度概述

进程调度指计算机通过决策决定哪个就绪进程可以获得 CPU 使用权！

- 保存旧进程的运行信息，请出旧进程（收拾包袱）
- 选择新进程，准备运行环境并分配 CPU（新进驻）



- 就绪队列的排队机制：为了提高进程的效率，事先将就绪进程按照一定的方式排成队列，以便调度程序可以最快找到就绪进程；
- 选择运行进程的委派机制：调度程序以一定的策略选择就绪进程，将 CPU 资源分配给它；
- 新老进程的上下文切换机制：如果需要把新的进程调度到 CPU 中，需要将旧的 CPU 中的进程备份出来，将新的进程切换到 CPU 中去；保存当前进程的上下文信息，装入被委派执行进程的运行上下文；



如果程序调度时，老进程还没有执行完会怎样呢？进程调度分为非抢占式调度、抢占式调度！

- 非抢占式调度：处理器一旦分配给某个进程，就让该进程一直使用下去；调度器不以任何原因抢占正在被使用的处理器；直到进程完成工作或因为 IO 阻塞才会让出处理器；
- 抢占式调度：允许调度程序以一定的策略暂停当前运行的进程；保存好旧进程的上下文信息，分配处理器给新进程；

对比	抢占式调度	非抢占式调度
系统开销	频繁切换、开销大	切换次数少、开销小
公平性	相对公平	不公平
应用	通用系统	专用系统

6.2、进程调度算法

- 先来先服务算法：在就绪队列，按先来先服务原则，优先取出先进入队列的就绪进程；
- 短进程优先调度算法：调度程序优先选择就绪队列中估计运行时间最短的进程；不利于长作业进程的执行；
- 高优先权优先调度算法：进程附带优先权，调度程序优先选择权重高的进程；使得紧迫的任务可以优先处理；（前台进程高于后台进程，是因为前台进程与用户交互，需要保证用户的体验，不卡顿！）
- 时间片轮转调度算法：按照先来先服务的原则排列就绪进程；每次从队列头部取出待执行进程，分配一个时间片，时间片用完后不管进程是否执行完、都会将进程重新插入队列尾部，然后取出第二个队列；是相对公平的调度算法，但不保证及时响应用户；

7、作业管理之死锁

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞现象；若无外力作用，它们都将无法推进下去。此时称操作系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程！

7.1、死锁的产生原因

死锁的产生是由于竞争资源、进程调度顺序不当导致的！

竞争资源：

- 共享资源数量不满足各个进程需求；
- 各个进程之间发生资源竞争导致死锁；

- 自身占用资源不释放，而一直在等待请求的资源被释放；



假设进程 1 申请传真机为步骤 A、进程 2 申请打印机为步骤 B、进程 2 申请传真机为步骤 C、进程 1 申请打印机为步骤 D！按照 $A \rightarrow B \rightarrow C \rightarrow D$ 的顺序调度，就会引起进程死锁；如果将进程调度顺序改为 $A \rightarrow D \rightarrow B \rightarrow C$ 就不存在死锁的情况！

7.2、死锁产生的四个必要条件

死锁的产生，必然同时满足以下四个条件，仅满足其中的某几个是不会产生死锁的！

- 互斥条件：进程对资源的使用具有 **排它性**；某个资源仅能由一个进程使用，其它进程需要使用只能等待；
- 请求保持条件：进程至少保持一个资源，又提出了新的资源请求；但新资源被占用，导致请求被堵塞；同时被阻塞的进程又不释放自己保持的资源；
- 不可剥夺条件：进程获得的资源在未完成使用前不能被剥夺；获得的资源只能由进程自身释放；
- 环路等待条件：发生死锁时，必然存在进程-资源环形链；

7.3、预防死锁的方法

破坏死锁产生的四个必要条件中的某一个或者某几个，就能有效预防死锁的产生！

- 摒弃 请求保持条件：系统规定进程在运行之前，一次性申请所有需要的资源；在进程运行中，不再去申请资源；
- 破坏 不可剥夺条件：当一个进程请求新的资源得不到满足时，必须释放其自身占有的资源；进程运行时占有的资源可以被释放，意味着可以被剥夺；
- 破坏 环路等待条件：可用资源线性排序，申请必须按照需要递增申请；线性申请不再形成环路，从而摒弃了环路等待条件；

7.4、银行家算法

银行家算法是一个可操作的著名的避免死锁的算法、以银行借贷系统分配策略为基础的算法！

银行家算法的策略基础：

- 假设客户申请的贷款是有限的，每次申请需声明最大资金量；
- 银行家在能够满足贷款时，都应该给用户贷款；
- 客户在使用完贷款后，能过及时归还贷款；

The diagram illustrates the state of resource allocation for four processes (P1, P2, P3, P4) across four resources (A, B, C, D). The tables are as follows:

- 所需资源表 (Required Resource Table):**

	A	B	C	D
P1	0	6	5	6
P2	1	9	4	2
P3	1	3	5	6
P4	1	7	5	0

- 已分配资源表 (Allocated Resource Table):**

	A	B	C	D
P1	0	0	1	4
P2	1	4	3	2
P3	1	3	5	4
P4	1	0	0	0

- 可分配资源表 (Available Resource Table):**

	A	B	C	D
1	5	2	0	0

- 还需分配资源表 (Remaining Resource Table):**

	A	B	C	D
P1	0	6	4	2
P2	0	5	1	0
P3	0	0	0	2
P4	0	7	5	0

如上图所示，有初始状态的 所需资源表、已分配资源表；所需资源表减去已分配资源表得到的 还需分配资源表；还有 可分配资源表！

- 所需资源表： P1 需要 0 个 A 资源、 6 个 B 资源、 5 个 C 资源、 6 个 D 资源；
- 已分配资源表： P1 已经拥有 0 个 A 资源、 0 个 B 资源、 1 个 C 资源、 4 个 D 资源；
- 还需分配资源表： P1 还需要 0 个 A 资源、 6 个 B 资源、 4 个 C 资源、 2 个 D 资源；
- 经过与 可分配资源表 对比后，发现不能满足 P1 、 P3 、 P4 的需求，仅仅能满足 P2 的需求；
- 因此先满足 P2 ，执行完 P2 后 P2 归还资源；
- 此时 可分配资源表 又可以分配资源给 P1 、 P3 、 P4 ！

8、存储管理之内存分配与回收

早期的计算机编程并不需要过多的存储管理；随着计算机和程序越来越复杂，存储管理称为一件必要的事情！

- 确保计算机有足够的内存来处理数据；
- 确保程序可以从可用内存中获取一部分内存使用；
- 确保程序可以归还使用后的内存以供其他程序使用！

8.1、内存分配的过程

单一连续分配：

- 最简单的内存分配方式；
- 只能在单用户、单进程的操作系统中使用；
- 将主存分为系统区、用户区；
- 系统区指的是内存给操作系统所使用；
- 用户区指的是所有的用户区内存都给用户区程序所使用；

固定分区分配：

- 支持多道程序的最简单存储分配方式；
- 将内存空间划分为若干个固定大小的区域；
- 每个分区只提供给一个程序使用，互不干扰；

动态分区分配：

- 根据进程实际需要、动态分配内存空间；
- 涉及到相关数据结构（如动态分区空闲表数据结构、动态分区空闲链数据结构等）、分配算法；

假设主存中有若干个分区，并且一些分区已经使用、一些分配还没有使用；这时候就需要一个数据结构来存储某个分区是否已使用！



把所有的空闲节点首尾相连，形成一个空闲链表



由于节点2、3，节点5、6是在一起的，
因此可以合并这些节点，减少空闲链表节点

* 由于每个节点的大小不一样
* 需要每个节点记录可存储的容量

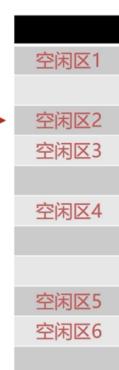
动态分区分配算法：

- 首次适应算法（FF 算法）：
- 最佳适应算法（BF 算法）；将空闲区链表按照容量大小进行排序；在每一次需要分配时遍历链表找到最佳合适空闲区；
- 快速使用算法（QF 算法）；要求有多个空闲区链表；每个链表存储一种容量的空闲区；

首次适应算法（FF 算法）

如某个进程 A 需要两个内存片；
首次适应算法会从链表头部开始查找；
节点 1 的容量为 1 不合适；
接着查找节点 2 的容量为 2；
将节点 2 的空闲区分配给进程 A 使用

* 每一次分配内存时，使用空闲链数据结构从开始顺序查找适合内存区
* 若在空闲链表上没有发现合适的空闲区，则该次分配失败
* 如果在空闲链表上找到了空闲区，则将该空闲区划分给该进程所使用
缺点是每次都从空闲链表头部开始查找，导致头部地址空间不断被划分
可能头部地址空间被划分为很多细小的碎片，如 1 个碎片 1 个碎片等
某些进程需要大内存片，比如进程 B 需要 5 个内存片
那么进程 B 就又要从头部一直找，可能找到尾部才找到合适的内存片；
久而久之，导致查找效率低下



改进算法：循环适应算法，每一次查找时不是从头部开始，而是从上次检索结束的位置开始查找

最佳适应算法

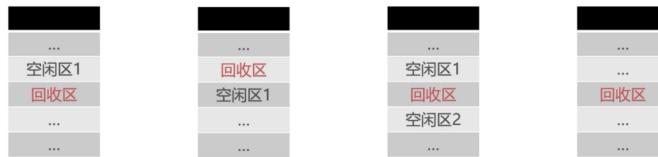
◆ 最佳适应算法要求空闲区链表按照容量大小排序 (容量从小到大)

◆ 遍历空闲区链表找到最佳合适空闲区



可以避免一些大材小用的情况，因为遍历链表是按照容量从小到大的顺序遍历的；因此匹配的空闲区肯定是大小刚适合进程所使用的；

8.2、内存回收的过程



内存回收过程分为四种情况：

- 情况1：需要回收的区域和空闲区连接在一起，并且位于空闲区后面；
- 回收1：不需要新建空闲链表节点；只需要把空闲区1的容量增大为包括回收区的空闲区即可；
- 情况2：需要回收的区域和空闲区连接在一起，并且位于空闲区前面；
- 回收2：将回收区和空闲区合并为一个新的节点，然后使用回收区的地址作为新的节点地址；
- 情况3：需要回收的区域和空闲区连接在一起，并且位于空闲区中间；
- 回收3：将空闲区1、回收区、空闲区2合并为一个新的节点，然后使用空闲区1的地址作为新的节点地址；
- 情况4：需要回收的区域和空闲区没有连接在一起，单一的回收区；
- 回收4：为回收区创建新的节点，然后将新的节点插入到相应的空闲区链表中即可；

9、存储管理之段页式存储管理

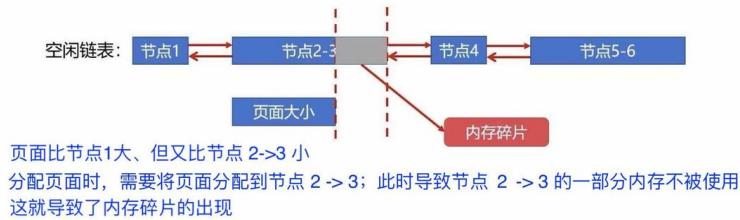
操作系统是如何管理进程的内存空间呢？

9.1、页式存储管理

前文提到的 字与字块 是相对于物理设备的定义，而此处的 页面 则是相对于逻辑空间的定义！

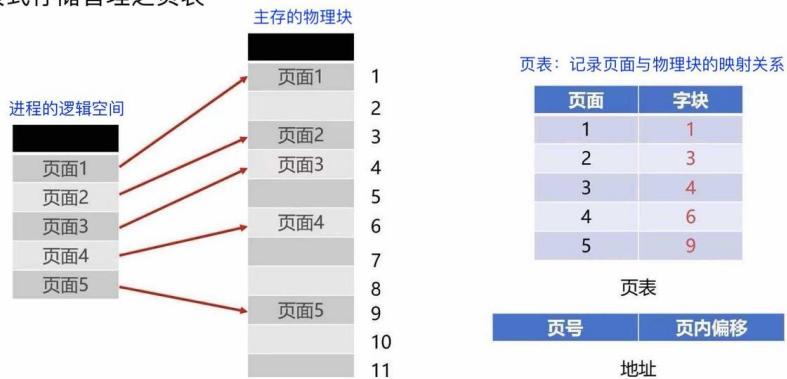
- 将进程逻辑空间 等分为若干个大小的页面；
- 相应的把物理内存空间分成与页面大小一样的物理块；
- 以页面为单位把进程空间装进物理内存中分散的物理块；
- 页面大小应该适中，过大难以分配，过小则内存碎片过多；
- 页面大小通常是 512B ~ 8 K；

页式存储管理



通过页式存储管理，可以把进程的逻辑空间的每个页面放到内存的物理块中去！但是又如何知道进程的某个页面具体被分配到哪一个字块中去呢？这时候就需要了解页表的概念！

页式存储管理之页表



页表：

- 页表是一个记录进程逻辑空间与物理空间的映射表；
- 在页式存储管理中，**地址**分为**页号**与**页内偏移**；

在现代计算机系统中，可以支持非常大的逻辑地址空间（ $2^{32} \sim 2^{64}$ ）；这样就会导致页表就变得非常大，要占用非常大的内存空间。如具有 32 位逻辑地址空间的分页系统，规定页面大小为 4 KB，则在每个进程页表中的页表项可达 1 M (2^{20}) 个，如果每个页表项占用 1 Byte，则每个进程仅仅页表就要占用 1 MB 的内存空间！

32 位系统进程的寻址空间位 $2^{32} = 4$ G
 4 G / 4KB = 2^{20}

使用多级页表可以解决页表占用高的问题！



- 首先，多级页表有一个根页表；
- 根页表的每个字块都指向内存的一片地址空间，这歌地址空间存储一个二级页表；
- 假设每个二级页表有 1024 项，每一项指向的字块才是进程实际使用的内存；
- 一个根页表可以指向二级页表，这样子大大的减少的进程的页表所占用的内存空间；
- 在运行时，只需要把根页表加载到内存中即可；如果调用某个字块，发现二级页表不在内存空间；此时只需要把二级页表加载到内存中，做到按需加载，节省内存空间；

页式存储管理仍然有一个问题：假如有一段连续的逻辑分布在多个页面中，将大大降低执行效率！此时提出了段式存储管理！

9.2、段式存储管理

- 将进程逻辑空间 非等分地 划分为若干段；
- 段的长度由进程的连续逻辑长度决定；
- 如进程的逻辑有 主函数 MAIN 、字程序段 X 、字函数 Y 等，此时按照每个函数的逻辑长度分配逻辑空间；
- 段表：段式存储管理也需要一个表来保存逻辑空间到物理空间的映射关系；



不管是段式存储管理、或者页式存储管理，都离散地管理了进程的逻辑空间；不同之处在于：

- 页是物理单位（从物理的角度划分），段是逻辑单位（从进程的逻辑划分）；
- 分页是为了合理的利用空间；分段是为了满足用户需求；
- 页大小由硬件固定；段长可动态变化；

- 页表信息是一维的，段表信息是二维的；段表中每一段的长度不同，因此需要把段的基址和长度都记录起来，所以段表信息是二维的！

9.3、段页式存储管理

分页可以有效提高内存利用率（虽然存在页内碎片），而分段可以更好的满足用户需求（因为逻辑是用户写的）；将两者结合，形成了段页式存储管理！

- 先将逻辑空间按段式管理分成若干段；
- 再把段内空间按页式管理分成若干页；
- 地址分为页号和页内偏移；
- 段地址分为段号和段内偏移；
- 段页地址：段号、段内页号、页内地址

段号	段内页号	页内地址
指定进程逻辑空间的具体哪一段	段里面具体的某一页	某一页的具体哪个字

