1. ：计算机的运行究竟是在做什么？来看一下经典的冯诺依曼结构。计算机科学虽然飞速发展了几十年，但是依旧遵循冯诺依曼结构。

冯诺依曼结构

1. 关于程序与数据，数据就是一首MP3歌曲， 程序就是用来控制解析播放这首歌的代码，从底层来讲就是供CPU运行的指令.总之在计算机当中它们都是0和1，将它们理解为同一个意思，毕竟它们都属于0和1组成的流\*

3，为什么计算机需要存储器部分？这是显而易见的，我写好了程序,或者下载了一部电影，肯定得有个地方放啊。这样今后需要的时候，才能运行程序或者看电影啊。

+ 1.\*\*稳定，掉电不丢失数据：

+ 2.\*\*存储容量大：

+ 3.\*\*读写速度快： +

4.\*\*价格便宜：\*+

5.\*\*体积小：\*\*

完全满足我们理想条件的存储器目前还没发明出来呢

所以这也是目前计算机系统存储器系统比较复杂的原因，区分为内存，硬盘，光盘等不同的存储器，如果有个完美的符合我们理想条件的存储器，直接使用这种存储器就好了。

> 为了行文方便，文中直接将存储器用磁盘来代替了，一来大家对磁盘比较熟悉，二来磁盘也是最常见的存储设备。类似flash，SD卡，ROM等从广义上来讲，也可以称为磁盘。因为它们的作用都是存储数据，掉电后不丢失。(这在下面文章中也会讨论到)

> 磁盘和硬盘什么关系呢？其实是同一个意思。硬盘是最常见的磁盘类型。在很早之前，计算机使用\*\*软盘\*\*存储数据，所以那种软盘也被称为 磁盘，不过软盘都早就被历史淘汰了，(电脑硬盘分区从C盘开始，就是因为AB盘是之前软盘的编号)。所以现在我们说磁盘，直接理解成硬盘就好了。

> 在我们软件当中，有个概念叫做\*\*数据持久化\*\*，意思就是说将数据存储起来，掉电之后不丢失，这其实就是存储在磁盘上面。

计算机运行就是这样一个过程：将数据从磁盘送往CPU，供CPU进行计算，并将结果输出。

> 因为我们这片文章就是 讨论 内存，存储等问题，所以关于 输入设备，输出设备之类的，就不再涉及和讨论。

摩尔定律, intel i9的制程工艺已经到了14nm

闲扯了一段CPU的发展历史，想说明的是，现在的CPU集成度越来越高，速度也越来越快。每秒钟能执行的指令也越来越多。

流水线(pipeline)技术

超线程(Hyper-Threading)技术

乱序执行

**两个for循环**

磁盘

假设磁盘一秒钟可以读取100条指令。那么这中间就存在 巨大的速度差异。CPU再快，可是磁盘严重拖后腿，那CPU就相当于工作严重不饱和，如果直接从磁盘上 来读取数据，那么CPU相当于 99.9999%的时间都在闲置着。

存储器层次结构

内存中的读写速度比磁盘快了几十万倍。但是相对于CPU的速度依旧还是慢。那么主存和CPU之间，可以继续添加速度更快的过度层。所以intel i7的存储器层次结构是这样的。

现代的计算机系统，在CPU与磁盘/主存之间，加了多层过度层。

> 严格来讲，应该叫CPU的算术逻辑单元（ALU），但是简单的直接说CPU，大家肯定也能听得懂。

我们为什么需要内存(主存),那么理解了主存，自然也就理解了L3，L2，L1等各级缓存存在的意义。

实际上\*\*这是一种缓存思想。\*\*比如，本地磁盘也相当于 远方服务器的缓存。因为我们从网上下载数据/文件时，速度明显比从本地磁盘读取要慢。

App上下载远方图片。

> 我们是拿i7处理器来做例子，它有三级缓存，像低端一些的处理器，比如i3，只有两级缓存，但是道理是相同的。本文当中，都是拿i7的存储器层次来做例子。

RAM，ROM，总线等

+ \*\*闪存(Flash memory)\*\*：非易失性存储器。SSD，SD卡都属于Flash技术，如果从概念上来讲，他们都属于ROM，这类存储器经常用在手机，相机等设备上。而机械硬盘常用在个人计算机，服务器上。

其实我觉的把 Flash，ROM等都叫做磁盘，也没什么错。毕竟它们的作用和概念都是相似的，区别只是他们各自使用的半导体技术不同。

"图4：一个存储器层次结构的示例"，越往上，读写速度越快，价格更贵，存储容量也越小。。像L0 寄存器，每个寄存器只能存储一个字长的内容，但是CPU读写取寄存器耗费的时钟周期为0个。这是最快的速度。

我们在电脑主板上却没看到L3-L0。原因很简单，他们都是集成在CPU芯片内部的。

> CPU要通过I/O桥(就是主板的北桥/南桥芯片组)与外围设备连接，因为CPU的主频太高了，它的时钟周期一秒钟震荡几亿次，外围设备的时钟周期都较慢，所以他们不能直接通信。

> 本文是讨论软件的，所以硬件部分就一笔带过，读者知道有这回事就ok了。总线上携带地址，数据和控制信号， 如何区分不同信号，分辨它与哪个外围设备通信，这就是另外一个问题了。

硬盘是瓶颈，那么加缓存意义何在？

第二步磁盘操作很慢，但是在第一步CPU发出信号后。但是第二步和第三部时，CPU根本不参与。第二步很耗时，所以CPU在第一步发出信号后，就去在干其他事情啊。(切换到另一个线程)。所以此时的CPU依旧没有闲着。而待第三步时，通过中断，硬盘主动发信号给CPU，你需要的数据已经发送到内存了，然后此时它可以将线程再切换回来，接着执行这个该线程的任务。

App为什么不消失

int main(){

//我们执行任务的代码

return 0;

}

int main(){

boolean flag = true;

while (flag){

//我们执行任务的代码

}

return 0;

}

在一个程序内部，也有大量的for，while等循环语句。

局部性原理(Principle of locality)

一般而言，\*\*有良好局部性的程序比局部性差的程序运行的更快。\*\* 现代计算机系统的各个层次，从硬件到操作系统、再到应用程序，它们的设计都利用了局部性。

`volatile` 关键字

虚拟内存

虚拟内存是计算机系统最重要的概念之一，并且它成功的主要原因就是它一直在沉默的，自动的工作。如果不理解虚拟内存，你根本不可能理解程序的深层次运行原理。也不可能去理解汇编器，链接器，加载器，共享对象，文件和进程等概念。

首先让我们思考几个问题:

1. `mov eax,0x123456;

假设有一个音乐播放器应用的汇编代码中，引用了`0x123456`这个内存地址。但是同时运行的应用有很多，那其他应用也完全有可能引用 `0x123456`这个地址。那为什么竟然没起冲突和错误呢？

1. 进程是计算机领域最重要的概念之一，什么是进程？进程是关于某次数据集合的一次运行活动， 是运行在它自己地址空间的一段自包容程序， 解释的通俗的点， 一个程序在运行时，我们会得到一个假象，该进程好像是独占地使用CPU和内存，CPU是没有间断地一条接一条的执行该程序的指令，所有的内存空间都是供该进程的代码和数据分配使用的。(这点不严谨，其实内存还有一部分要分给`内核kernel`)。说起来，这个程序就好像得到了全世界一样。，CPU是我的，内存也全部我的，妹子们还是我的。当然这是假象而已。但是这些假象又是怎么做到的呢？
2. 程序中都会引用库API，比如每个C程序都要引用`stdio.h`库的`printf()`，在程序运行时，库代码也要被加入到内存，这么多程序都引用了这个库，难道我内存中需要加很多份吗？这自然不可能，那么库代码又是怎么被所有进程共享的呢？

物理和虚拟寻址

物理寻址

> 注意：在访问内存时，\*\*对于任意一个地址，(不管是第0个还是第M-1个)，访问该地址的时间总是相同的\*\*。

> 在各种数据结构中，我们都说hash表是最快的，比红黑树之类的都要快，那hash表为什么最快？那是因为hash表内部本质上是使用了数组。所以还是数组最快，那数组为什么最快？这是因为我们知道数组的起始地址以及某个元素的序号，就可以得到该元素在内存中的地址，而对于内存，访问任意一个地址，访问时间总是相同的。而类似链表，树等结构，却只能靠遍历了。

> 因为是关于L4级主存和磁盘之间的交互问题，为行文方便，文章中有时候直接说内存代指主存。所以这些不要误以为是指L1，L2之类的缓存。如果看不懂这段话啥意思，务必看看我的上一篇文章[TODO](http://www.yaoxiaowen.com/),然后再来看这篇文章。

虚拟寻址

> 有少数现代计算机系统依旧在使用物理寻址方式，比如DSP，嵌入式系统，超级计算机系统。这些系统的主要任务是执行单一任务，不像通用性计算机那样需要执行多任务。可以想象到，物理寻址方式更快。这个道理和理论上java比C++慢的道理是一样的。

进程地址空间

上图是一个64位的进程地址空间，编译器在编译程序时，将结果编译成32/64位的地址空间。虚拟寻址方式简化了编译器，链接器的工作。同样也因为虚拟内存，每个进程才能有很大的，一致的，私有的的地址空间。这方便了内存管理，保护了每个进程的地址空间不被其他进程破坏。同时也方便了共享库。

虚拟内存也是一种缓存思想

从概念上来说，虚拟内存被组织成为一个由存放在磁盘上的 N 个连续的字节大小的单元组成的数组，也就是字节数组。每个字节都有一个唯一的虚拟地址作为数组的索引。虚拟内存的地址和磁盘的地址之间建立影射关系。磁盘上活动的数组内容被缓存在主存中。在存储器层次结构中，磁盘(较低层L5，参见我们上篇文章图4)的数据被分割成块(block)，这些块作为和主存(较高层,L4)之间的传输单元。主存作为虚拟内存(或者说磁盘)的缓存。

虚拟内存（VM）系统将虚拟内存分割成称为大小固定的虚拟页（Virtual Page,VP），每个虚拟页的大小为 $P=2^p$ 字节。同样的，物理内存被分割为物理页（Physical Page,PP）,大小也为 $P=2^p$字节（物理页也称作页帧，page frame）。

> 其中\*\*未分配的VP\*\*不占用任何的实际物理空间，这点要理解。32位程序地址空间就有4G，至于64G的程序它的地址空间是一个非常大的天文数字(貌似是16777216T)，而目前我们的电脑高配的也就2T磁盘，16G内存。如果64位程序每个VP都映射着实际的PP。无论如何也对应不上的。并且也完全没必要一一映射,"图12:进程地址空间"中可以看到，地址空间内有大量的空白。毕竟程序不可能实际使用那么大的地址空间。

> 当然，那个图上标注的不对,VP 部分， `n-p`和`N-1`应该分别标注为`3`和`7`,不过我们找不到更合适的图了，(这种图自己画压力太大了)。所以大家知道我们假设共有8个VP就好了。

页表(page table)

+ 已经在主存中，就需要判断出该虚拟页存在于哪个物理页中。

+ 不在主存中，那么系统必须判断虚拟页存放在磁盘的哪个位置，并且在物理主存中选择一个牺牲页，并将该虚拟页从磁盘复制到 主存，替换这个牺牲页。

这些功能由软硬件联合提供，包括操作系统，CPU中的\*\*内存管理单元（Memory Management Unit,MMU）\*\*和一个存放在物理内存中叫\*\*页表（page table）\*\*的数据结构，页表将虚拟页映射到物理页。每次地址翻译硬件将一个虚拟地址转换成物理地址时都会读取页表。

上图展示了一个页表的基本结构，页表就是一个\*\*页表条目（Page Table Entry,PTE）\*\*的数组。虚拟地址的每个页在页表中都有一个对应的PTE。在这里我们假设每个 PTE 是由一个有效位（Valid bit）和一个 n 位地址字段组成的。有效位表明了该虚拟页当前是否被缓存在 主存 中。

+ 有效位为 1，则主存缓存了该虚拟页。地址字段就表示主存中相应的物理页的起始位置。

+ 有效位为 0，则地址字段的null表示这个虚拟页还未被分配，否则该地址就指向该虚拟页在磁盘上的起始位置。

页命中与缺页

> 我们在上篇文章[TODO](http://www.yaoxiaowen.com/)中说过缓存命中与不命中的问题，都是缓存思想，在这里肯定也会存在同样的问题。并且磁盘与主存之间的缓存不命中代价肯定大的多。因为L0-L4之间，每级缓存的速度大约相差10倍左右，但是L4主存与L5磁盘之间，它们的速度相差约十万倍。所以主存与磁盘之间交换的页容量是最大的，尽可能的增加命中率。相应的替换策略，操作系统也使用了更加复杂精密的算法。

> 在上篇文章[TODO](http://www.yaoxiaowen.com/)，每次替换的区域，我们用了\*\*块(block)\*\*,而这里我们却在说\*\*页(page)\*\*， 其实同一个意思。只是因为历史原因，叫法不同罢了。

> 我们在上篇文章中说过缓存命中与不命中的问题，都是缓存思想，在这里肯定也会存在同样的问题。并且磁盘与主存之间的缓存不命中代价肯定大的多。因为L0-L4之间，每级缓存的速度大约相差10倍左右，但是L4主存与L5磁盘之间，它们的速度相差约十万倍。所以主存与磁盘之间交换的页容量是最大的，尽可能的增加命中率。相应的替换策略，操作系统也使用了更加复杂精密的算法。

> 之前每次替换的区域，我们用了\*\*块(block)\*\*,而这里我们却在说\*\*页(page)\*\*， 其实同一个意思。只是因为历史原因，叫法不同罢了。

> 我们刚才说，缺页错误是一种异常，但是实际上，在计算机系统中，被0除，读写文件，还有上篇文章中我们所说的中断(interrupt)，甚至包括我们代码中写的`try catch`，都是一种异常。 比如被0除是intel 的CPU规定的的第0号故障(fault)类型的异常。而读写文件，分别是linux规定的第0号和第1号陷阱(trap)类型的异常。多任务的上下文切换，进程的创建回收等，等与系统中这种异常流的处理密切相关。当然，这是另外一个话题了。我们在这里不做累述。

虚拟内存作为内存管理和内存保护的工具

> 此时，大家再看一下"图:12 进程地址空间",就会发现在地址空间当中，"共享库的内存映射区域"对于每个进程起始地址都是相同的。再想想进程之间\*\*共享内存\*\*的通信方式, 所以说\*\*虚拟内存简化了共享机制\*\*

大家知道，C语言中存在指针，可以直接进行内存操作。因为有了虚拟内存，所以我们的指针操作也不会访问到其他进程的区域，但是哪怕是对于自己的地址空间，很多内存区域也应该是禁止访问的，这不仅包括kernel的区域，也包括自己的只读代码段。那么虚拟内存就提供了这样的一种内存保护工具。

内存保护

段 和 页

我们明白了页，页是操作系统为了管理主存方便而划分的，对用户不可见。但是思考这种情况，假设一个页的大小是1M。但是某个程序数据加起来也就0.5M，所以在内存和磁盘进行页交换明显的浪费内存了。所以还一种划分方式是\*\*分段\*\*。上面那个例子，我将该段划分为0.5M，在内存和磁盘之间交换，这样就避免了浪费。

段是信息的逻辑单元，是根据用户需求而灵活划分的，所以大小不固定，对用户是可见的，提供的是二维地址空间。

> 对于段，我没找到比较好的资料，所以也没有理解的更清楚，网上的很多文章都相互抄袭。据我所了解，汇编程序员是可以直接操作段的，\*\*但是我们写高级语言的程序员有响应的API能进行段操作吗？\*\*所以对于段的相关知识，真心不了解，

swap分区的作用

熟悉linux的同学，应该知道linux有一个swap分区。Swap空间的作用可简单描述为：当系统的物理内存不够用的时候，就需要将物理内存中的一部分空间释放出来，以供当前运行的程序使用。那些被释放的空间可能来自一些很长时间没有什么操作的程序，这些被释放的空间被临时保存到Swap空间中，等到那些程序要运行时，再从Swap中恢复保存的数据到内存中。这样，系统总是在物理内存不够时，才进行Swap交换。

在window下也有类作用的对用户不可见的匿名磁盘空间(在C盘)。

百度百科上对于\*\*虚拟内存\*\*的解释非常混乱

关于虚拟内存,看了[百度百科](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98)的内容，解释的一片混乱，有些地方是对的，但是有些地方解释的是关于swap分区的内容。诚然，把内存内容写到磁盘上备份，需要的时候再回写到内存中，起到了扩大内容的作用，如果光从字面意思开看，也可以叫做虚拟内存，但是此虚拟内存并非我们本文当中介绍的虚拟内存。我又查看了[维基百科](https://zh.wikipedia.org/wiki/%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98)的内容，该词条内容不长，但是下面这段话很重要。

> 注意：虚拟内存不只是“用磁盘空间来扩展物理内存”的意思——这只是扩充内存级别以使其包含硬盘驱动器而已。把内存扩展到磁盘只是使用虚拟内存技术的一个结果，它的作用也可以通过覆盖或者把处于不活动状态的程序以及它们的数据全部交换到磁盘上等方式来实现。对虚拟内存的定义是基于对地址空间的重定义的，即把地址空间定义为“连续的虚拟内存地址”，以借此“欺骗”程序，使它们以为自己正在使用一大块的“连续”地址。

所以我认为百度百科的解释是混乱的，部分内容是错误的。而维基百科上的应该才是正确的。