多道程序调度

多道程序设计是在计算机内存中同时存放几道相互独立的程序（只有将程序放到内存 CPU 才会执行），使它们在管理程序控制之下，相互穿插地运行（内存中多道程序轮流地或时获分地占有 CPU），交替地执行（单 CPU 情况），使他们共享 CPU 和系统中的各种资源。当某一程序因为某种原因不能继续执行时（如等待外部设备输入输出数据或者其他中断处理），操作系统的管理程序将会让 CPU 执行内存中的另一道程序，如此可以相对减少 CPU 和其他外部设备的空闲时间（即处于忙碌状态），从而提高计算机的使用效率。

优点：

（1）提高 CPU 的利用率；

（2）提高内存和 I/O 设备的利用率；

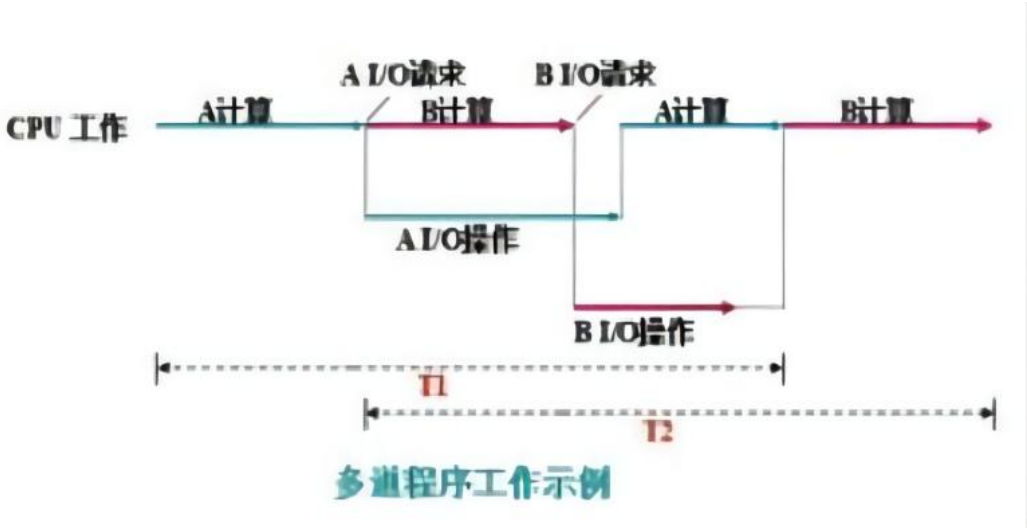
（3）增加系统吞吐量。

特征：

（1）多道：计算机中同时存放几道相互独立的程序；

（2）宏观上并行：同时进入系统的几道程序都处于运行过程中，即它们先后开始六各自的执行，但都未运行完毕；

（3）微观上串行：内存中的多道程序轮流地或分时地占有 CPU，交替地执行（单 CPU 情况）。



虚拟内存技术

虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

现代所有用于一般应用的操作系统都对普通的应用程序使用虚拟内存技术，老一些的操作系统，如 DOS 和1980年代的 Windows，或者那些1960年代的大型机，一般都没有虚拟内存的功能。

直接使用物理内存的状态下就会产生一些问题：

1. **内存空间利用率的问题**

   各个进程对内存的使用会导致内存碎片化，当要用 malloc 分配一块很大的内存空间时，可能会出现虽然有足够多的空闲物理内存，却没有足够大的连续空闲内存这种情况，东一块西一块的内存碎片就被浪费掉了。

2. **读写内存的安全性问题**

   物理内存本身是不限制访问的，任何地址都可以读写，而现代操作系统需要实现不同的页面具有不同的访问权限，例如只读的数据等等。

3. **进程间的安全问题**

   各个进程之间没有独立的地址空间，一个进程由于执行错误指令或是恶意代码都可以直接修改其它进程的数据，甚至修改内核地址空间的数据，这是操作系统所不愿看到的。

4. **内存读写的效率问题**

   当多个进程同时运行，需要分配给进程的内存总和大于实际可用的物理内存时，需要将其他程序暂时拷贝到硬盘当中，然后将新的程序装入内存运行。由于大量的数据频繁装入装出，内存的使用效率会非常低。

虚拟内存技术成功的解决了直接使用物理内存会出现的问题，比如物理内存中离散式存储，虚拟内存中连续存储解决了物理内存碎片化资源利用率过低的问题；每个进程只能访问自己独立的用户空间而内核空间是共用的解决了进程间的安全问题；缺页异常和选择牺牲页的算法提高了内存读写的效率等等。

内存一致性

对于单个处理器，很容易发现独立操作可以以并行内存操作按程序顺序执行，例如，读取按程序顺序返回最后一次写入的值。

对称多处理器 (SMP) 系统的多种内存一致模型

- **顺序一致（Sequential consistency）**：同一个线程的原子操作还是按照 happens-before 关系，但不同线程间的执行关系是任意。
- **松弛一致（Relaxed consistency，允许某种类型的重排序）**：如果某个操作只要求是原子操作，除此之外，不需要其它同步的保障，就可以使用 Relaxed ordering。程序计数器是一种典型的应用场景。
- **弱一致（Weak consistency）**：读写任意排序，受显式的内存屏障限制。

如果任何执行的结果与所有处理器的操作以某种顺序执行的结果相同，并且每个单独处理器的操作按程序建立的顺序出现，则称多处理器系统是顺序一致的。

1 内存模型介绍

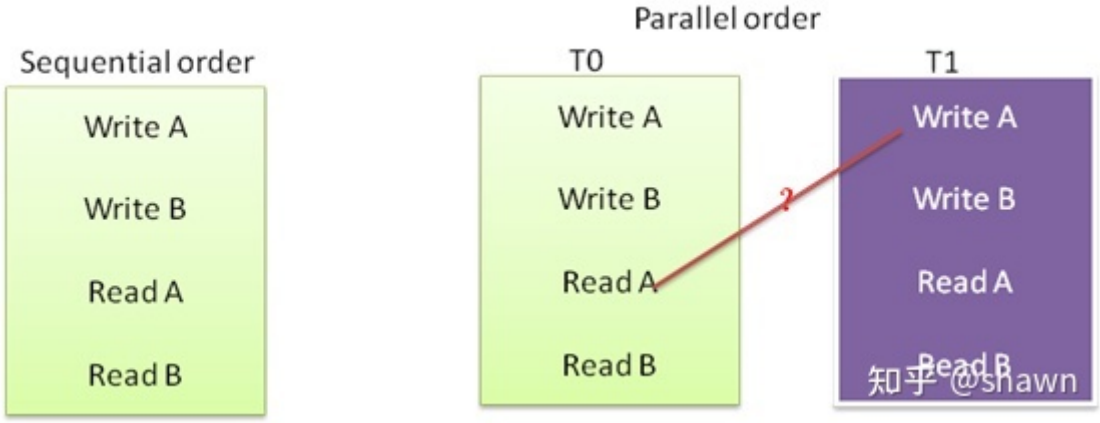
1.1 Sequential 内存一致性模型

上个世纪70年代，Lamport 提出了顺序一致性（Sequential consistency）的概念，他把基本不可实现的全局精确时钟要求转换到了局部时钟要求。Sequential 模型有三个特点：

（1） 所有的内存访问都是原子操作（中间不应该有write buffer和 cache）。

（2） CPU 核内的访问**严格按照程序代码顺序**进行，对于顺序执行 (in-order) CPU 严格按照程序要求的顺序执行对内存的存取（load和store）操作，对于乱序执行（out-of-order）cpu 也需要遵循该要求：严格按照程序要求的顺序完成对 load 和 store 指令的提交和退休(retire)。

（3） 多核之间代码可以以任意的顺序进行交织运行（但每个核内还是严格的顺序执行）



在 sequential 模型下，上述问题的运行结果就只能是先 A=1 ,然后才发生 flag=1 ,所以不可能打印出 A=0 的结果。

Sequentia l模型的优点是实现简单（对硬件和软件人员都简单），但问题是性能太差，无法发挥硬件的特性, 所以现在的 CPU 几乎无一例外地偏离了 Sequential 模型。

1.2 Relaxed 内存一致性模型

为了表达的方便，我们用 X->Y 来表示先必须完成 X 操作后才能执行 Y 操作。

前面介绍的 Sequential 模型就是要求严格保持 4 种可能的内存读写顺序场景（即使这些操作之间没有控制上的，数据上的，流水线上的依赖关系）：

宽松模型的关键思想是允许乱序执行读取 (load) 和写入 (store) 操作,通过使用同步操作来实现处理器之间的同步。宽松模型的方式可以多种多样的，我们可以根据放松了哪种读取和写入顺序来进行分类。

（1） 放松W->R顺序：我们就得到了 强顺序模型 TSO (total store ordering)，它允许 CPU 先执行读操作然后在执行写操作而不严格按照代码的指示顺序来进行。由于这种模型保持了写入操作之间的顺序，所以很多在 Sequential 模型下能够运行的代码也能在TSO模型下正常运行。

（2） 放松W->W顺序：我们就得到了 PSO (partial store ordering) 模型，允许多个写操作也被打乱顺序。

（3） 放松 R->W 和 R->R 顺序：将会得到很多模型，包括弱内存模型 WMO（Weak Memory Ordering），released 模型，等。

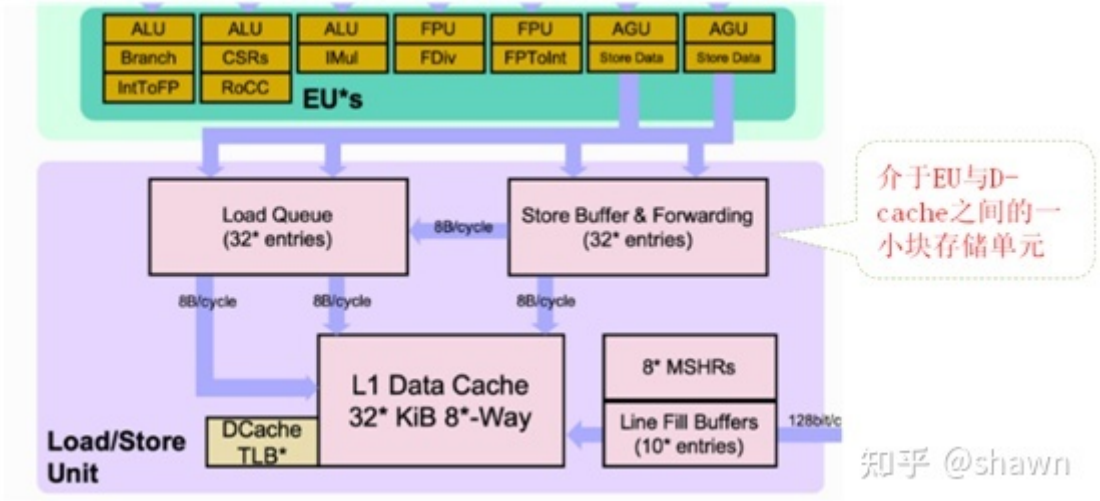
通过放松这些顺序约束，处理器性能可以得到显著的提升。

【注：这些顺序放松但依旧保证了在本 CPU 与程序代码一致的存取顺序，但在其它 CPU 节点看来顺序就可能被打乱了】

2 内存模型背后的原因

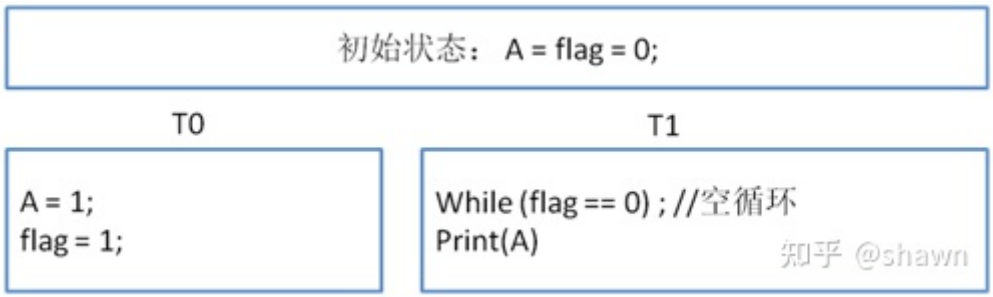
我们知道 CPU 访问主 memory 需要的开销非常大（上100个cycles），如果 CPU 想存储一个数据到 memory 而该数据又不在 data cache 中，则这个过程是相当耗时的，大大地降低了 CPU 性能。为了进一步加快内存访问速度，CPU 设计中引入了 store buffer 这个部件，store buffer是位于执行单元（比如浮点执行单元FPU, 算术逻辑单元ALU等）与 data cache之间的小一块存储单元。(借用一下

BOOMv3中的一个架构图)



使用 store buffer 的好处不仅在于加快存储的速度，另外还可以利用 store buffer 实现乱序执行以及实现推断执行 (speculative execution) 失败后的回退 roll back，等

当CPU需要写(store)数据时, 数据不直接写到 memory 或者 cache 中，而是先写到store buffer 中，store buffer 负责后续以某种次序写入 L1 data Cache 。我们还是用前面那个例子来解释一下什么情况下会输出另外一种结果：



假设 T0 在 CPU0 上执行，T1 在 CPU1 上执行：

- CPU0 将 A=1 写入 store buffer。
- CPU0 将 flag=1 写入 store buffer。
- CPU1 读取 flag 的值，由于 flag 新值还在 CPU0 的 store buffer 里面，所以 CPU1 看到的 flag 值依然是 0 。所以 CPU1 继续自旋等待。
- CPU0 的 store buffer 将 flag 的值写入 Cache，此时 A 的值依然在 store buffer 里面没有更新到 Cache（比如因为 A 没有在 cache 中而 flag 在 Cache 中，cache 的一致性保证了 CPU1 处看到的 flag=1）。
- CPU1 发现 flag 的值是1，退出循环。然后读取 A 的值，此时 A 的值是旧的数据，也就是 0。因为 A 的新值还在 CPU0 的 store buffer 里面，所以打印出 A=0。

缓存一致性

缓存的基本定律

如果我们只处理读操作，那么事情会很简单，因为所有级别的缓存都遵守以下规律，我称之为：

> ***基本定律***：在任意时刻，任意级别缓存中的缓存段的内容，等同于它对应的内存中的内容。

一旦我们允许写操作，事情就变得复杂一点了。这里有两种基本的写模式：直写（write-through）和回写（write-back）。直写更简单一点：我们透过本级缓存，直接把数据写到下一级缓存（或直接到内存）中，如果对应的段被缓存了，我们同时更新缓存中的内容（甚至直接丢弃），就这么简单。这也遵守前面的定律：缓存中的段永远和它对应的内存内容匹配。

回写模式就有点复杂了。缓存不会立即把写操作传递到下一级，而是仅修改本级缓存中的数据，并且把对应的缓存段标记为“脏”段。脏段会触发回写，也就是把里面的内容写到对应的内存或下一级缓存中。回写后，脏段又变“干净”了。当一个脏段被丢弃的时候，总是先要进行一次回写。回写所遵循的规律有点不同。

> ***回写定律***：当所有的脏段被回写后，任意级别缓存中的缓存段的内容，等同于它对应的内存中的内容。

换句话说，回写模式的定律中，我们去掉了“在任意时刻”这个修饰语，代之以弱化一点的条件：要么缓存段的内容和内存一致（如果缓存段是干净的话），要么缓存段中的内容最终要回写到内存中（对于脏缓存段来说）。

直接模式更简单，但是回写模式有它的优势：它能过滤掉对同一地址的反复写操作，并且，如果大多数缓存段都在回写模式下工作，那么系统经常可以一下子写一大片内存，而不是分成小块来写，前者的效率更高。

问题

当计算机加入了缓存，cpu 读取数据的时候首先会从缓存读取，如果缓存中不存在，则从主存读取，同时把该数据加入到缓存中，这样下次再次使用的时候就可以直接从缓存中读取。当修改了某些数据，先把修改后的数据写入到缓存中，然后再刷到主存中，以此来提高效率。

这样的过程在单线程的环境下是不会出现问题的，但是在多线程环境下就会出现问题，现在的计算机几乎都是多个核心的，多个线程运行在不同的核心中，每个核心都有自己的缓存。这时就会出现多个 cpu 同时修改了同一个数据的问题，这就是著名的缓存一致性问题。

早期 cpu 中，为了解决缓存一致性问题，计算机厂商们通过在消息总线上加锁来解决的，也就是说同时只有一个cpu能操作同一块数据。这样的后果就是，加锁期间其他 cpu 无法访问内存，导致效率低下，因此出现了第二种解决方案，就是通过缓存一致性协议来解决缓存一致性问题。

缓存一致性协议（MESI）

MESI 是取自缓存行（Cache line，缓存中存储数据的单元）中数据的四种状态的英文首字母，缓存行中数据具有四种状态，它们分别是：

> - Modified（修改）：数据有效，数据被修改了，和内存中数据不一致，数据只存在于本Cache中。
> - Exclusive（独享）：数据有效，数据和内存中的数据一致，数据只存在于本Cache中。
> - Shared（共享）：数据有效，数据和内存中的数据一致，数据存在多个Cache中。
> - Invalid（无效）：数据无效，一旦数据被标记为无效，那效果就等同于它从来没被加载到缓存中。

例如现在有一个 project，需要四个人（core）参与到 coding 工作，为了版本控制，我们将代码传到 github（内存）上，四个人分别是甲乙丙丁，对应着 core0，core1，core2 和 core3，每个人都有自己的计算机（local cache），平时在自己的计算机上 coding。

现在，假设甲从 github（内存）上下载（load）了某一个类文件（cache block）到他的计算机（甲的 local cache），而乙丙丁的计算机并没有这个类文件（cache block）的副本，这时候甲独占着这个类文件的副本，并且和 github（内存）上一样，所以此时这个类文件的状态是exclusive。

假设这时候丁也从 github上下载了这个类文件到他的计算机上，这时候，这个类文件在甲和丁两个人的计算机上都有副本，并且和github 上一样，所以此时这个类文件的状态是 shared。

假设现在丁在自己的计算机上对这个类文件进行了修改，此时，甲的计算机上的这个类文件就过时了，成了 invalid 状态，而这个类文件的最新副本在丁的计算机上，且只有丁的计算机上有这个类文件的最新副本，同时这个类文件和 github 上的内容不一致，是脏的（dirty），所以这个类文件的状态就是 modified。

MESI状态迁移

如果当前core的cache block状态是invalid

那么当前core读取这个cache block（local read）的时候，考虑两种情况：
　　如果其他core的cache上没有这个cache block的副本，那么从内存中加载，此时只有当前core拥有和内存中一致的cache block副本，所以转移到exclusive状态；
　　如果其他core的cache上有这个cache block的副本，那么从LLC中加载，此时不止一个core拥有和内存中一致的cache block副本，所以转移到shared状态。

当前core修改这个cache block（local write）的时候，这个cache block的内容与内存中不一致（dirty），并且只有当前core拥有这个最新的副本，所以转移到modified状态；

其他core读取这个cache block（remote read）以及其他core修改这个cache block（remote write）的时候，对当前core的cache中的cache block没有任何影响。

如果当前core的cache block状态是exclusive，

那么当前core读取这个cache block（local read）的时候，当前core的cache block的状态还是exclusive，因为local read不会让其他core拥有这个cache block的副本，并且也不会对这个cache block进行修改

当前core修改这个cache block（local write）的时候，因为对这个cache block进行了修改，也就和内存中不一致了，所以当前core的cache block转移到modified状态

其他core读取这个cache block（remote read）的时候，其他core的cache中也会拥有这个cache block的副本，所以此时当前core的cache block转移到shared状态

其他core修改这个cache block（remote write）的时候，当前core的cache block数据过时，不再有效，所以当前core的cache block转移到invalid状态

如果当前core的cache block处于shared状态

那么当前core对cache block的读取（local read）以及其他core对cache block的读取（remote read）对当前core的cache block状态无影响，所以还是shared状态

当前core对cache block的修改（local write）会使得当前core的cache block内容与内存中不一致，所以会转移到modified状态

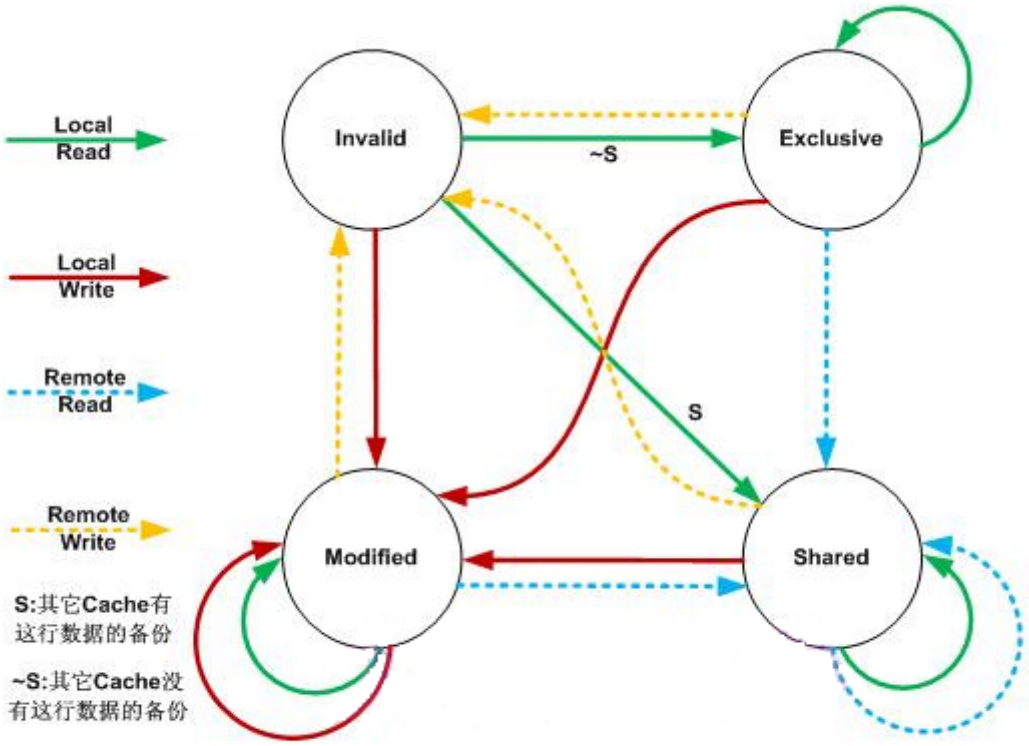
其他core对cache block的修改（remote write）会使得当前core的cache block内容过时失效，所以转移到invalid状态

如果当前core的cache block处于modified状态

那么当前core对cache block的读取（local read）以及当前core对cache block的修改（local write）对当前core的cache block状态不产生影响，所以还是modified状态

其他core对cache block的读取（remote read）会导致其他core拥有这个cache block的最新副本，所以此时不再是当前core独占，所以转移到shared状态

其他core对cache block的修改（remote write）会导致当前core的cache block内容过时失效，所以转移到invalid状态



MESI协议状态迁移图

闪存

闪存是一种电可擦除可编程只读存储器（Electrically Erasable ProgrammableRead Only Memory，EEPROM），具有非易失、读写速度快、抗震、低功耗、体积小等特性，目前己广泛应用于嵌入式系统、航空航天、消费电子等领域。[1]

闪存存储器主要分为NOR型和NAND型两种，NOR型闪存有独立的地址线和数据线，它支持按位进行访问，具有高可靠性且随机读取速度较快，但NOR闪存的擦除和写操作速度较慢、容量小、价格昂贵，主要用于存储程序代码并在内存中直接运行。NOR闪存在手机上得到了广泛的应用。NAND闪存相对于NOR型闪存拥有更大的容量，适合进行数据存储。

此外，它类似于传统的机械硬盘，对于小数据块的操作较慢，对于大数据块操作较快。NAND闪存根据其芯片单元所能存储的比特位数又可分为单级晶胞(SLC)和多级晶胞(MLC)两类。MLC技术具有显著的存储密度优越性，相对于SLC每个单元仅能存储1位比特，MLC可以存储多位比特，但MLC在速度和可靠性方面还有一定的提升空间。

原理

闪存结合了EPROM的高密度和EEPROM结构的变通性的优点。

EPROM是指其中的内容可以通过特殊手段擦去，然后重新写入。其基本单元电路如下图所示。常采用浮空栅雪崩注入式MOS电路，简称为FAMOS。它与MOS电路相似，是在N型基片上生长出两个高浓度的P型区，通过欧姆接触分别引出源极S和漏极D。在源极和漏极之间有一个多晶硅栅极浮空在 绝缘层中，与四周无直接电气联接。这种电路以浮空栅极是否带电来表示存1或者0，浮空栅极带电后（例如负电荷），就在其下面，源极和漏极之间感应出正的导电沟道，使MOS管导通，即表示存入0.若浮空栅极不带电，则不能形成导电沟道，MOS管不导通，即存入1。

EEPROM基本存储单元电路的工作原理如图2.2所示。与EPROM相似，它是在EPROM基本单元电路的浮空栅极的上面再生成一个浮空栅，前者称为第一级浮空栅，后者称为第二级浮空栅。可给第二级浮空栅引出一个电极，使第二级浮空栅极接某一电压VG。若VG为正电压，第一浮空栅极与漏极之间产生隧道效应，使电子注入第一浮空栅极，即编程写入。若使VG为负电压，强使第一浮空栅极的电子散失，即擦除。擦除后可重新写入。

闪存的基本单元电路与EEPROM类似，也是由双层浮空栅MOS管组成。但是第一层栅介质很薄，作为隧道氧化层。写入方法与EEPROM相同，在第二级浮空栅加正电压，使电子进入第一级浮空栅。读出方法与EPROM相同。擦除方法是在源极加正电压利用第一级浮空栅与漏极之间的隧道效应，将注入到浮空栅的负电荷吸引到源极。由于利用源极加正电压擦除，因此各单元的源极联在一起，这样，擦除不能按字节擦除，而是全片或者分块擦除。随着半导体技术的改进，闪存也实现了单晶体管设计，主要就是在原有的晶体管上加入浮空栅和选择栅，

NAND闪存阵列分为一系列128kB的区块(block)，这些区块是NAND器件中最小的可擦除实体。擦除一个区块就是把所有的位(bit)设置为“1”(而所有字节(byte)设置为FFh)。有必要通过编程，将已擦除的位从“1”变为“0”。最小的编程实体是字节(byte)。一些NOR闪存能同时执行读写操作(见下图1)。虽然NAND不能同时执行读写操作，它可以采用称为“映射(shadowing)”的方法，在系统级实现这一点。这种方法在个人电脑上已经沿用多年，即将BIOS从速率较低的ROM加载到速率较高的RAM上。

分类

Flash又分NAND Flash和NOR Flash，NOR型存储内容以编码为主，其功能多与运算相关；NAND型主要功能是存储资料，如数码相机中所用的记忆卡。

现在大部分的SSD都是用来存储不易丢失的资料，所以SSD存储单元会选择NAND Flash芯片。这里我们讲的就是SSD中的NAND Flash芯片。

Nor Flash：主要用来执行片上程序

优点：具有很好的读写性能和随机访问性能，因此它先得到广泛的应用；

缺点：单片容量较小且写入速度较慢，决定了其应用范围较窄。

NAND Flash：主要用在大容量存储场合

优点：优秀的读写性能、较大的存储容量和性价比，因此在大容量存储领域得到了广泛的应用；

缺点：不具备随机访问性能。

NAND 规则

- Flash都不支持覆盖，即**写入操作只能在空或已擦除的单元内进行。**更改数据时，将整页拷贝到缓存（Cache）中修改对应页，再把更改后的数据挪到新的页中保存，将原来位置的页标记为无效页；
- **以page为单位写入，以Block为单位擦除**；擦除Block前需要先对里面的有效页进行搬迁。
- 每个Block都有擦除次数限制（有寿命），擦除次数过多会成为坏块（bad block）

写入放大

一个page有三种状态：空；无效数据；有效数据。

向SSD写入数据时，以page为单位，但只能向空的page里写。如果要向存有无效数据的page里写，则需要先进行擦除，将无效数据的page变为空page，再写入。但是擦除数据时，是以block为单位的。

举例：向一个block写入一个page的新数据时，一个block里已经没有空的page了，只有有效数据以及无效数据的page，所以主控就把新数据以及有效数据读到缓存形成一个新的block，再擦除原先的block，再把新block写回去。这个操作带来的写入放大就是: 实际写一个page的4KB的数据，造成了整个block共512KB的写入操作，这就是128倍的写入放大（WA，Write Amplification）。写入放大造成了SSD实际对NAND的写入量大于主机要求的写入量。

同时，原本仅须一步写入page的操作变成：缓存读取新数据以及有效数据→闪存擦除→缓存写入，造成延迟大大添加，速度变慢。
所以说，写入放大是影响SSD随机写入性能和寿命的关键因素。以100%随机4KB来写入，眼下的大多数SSD主控，在最坏的情况下WA能够达到100以上。假设是100%持续的从低LBA写入到高LBA的话，WA能够做到1，实际使用中写入放大会介于这两者之间。

当一个block里的page只有空、有效数据两种状态时，在写入数据的时候不需要进行擦除，trim就是通知SSD在空闲状态时，将无效数据擦除，以减少SSD在写入数据时需要进行的数据擦除工作。

写入放大倍数 = 闪存写入数据量 / 主控写入数据量 = 实际写入数据量 / 要求写入数据量

一些知识

HDD

HDD是指机械硬盘，是传统普通的硬盘。

介质：采用磁性碟片来存储。

包括：盘片、磁头、磁盘旋转轴及控制电机、磁头控制器、数据转接器、接口、缓存。

机械式硬盘最大速率约为100MB/s，由于容易发热等原因已经无法再进一步提升速度，所以引入了固态硬盘

SSD

SSD（Solid State Drives）是固态硬盘。

介质：采用闪存颗粒来存储。

包括：控制单元、存储单元（DRAM芯片/FLASH芯片）。

性能区别

HDD是机械式寻找数据，所以**防震**远低于SSD，**数据寻找时间**也远低于SSD