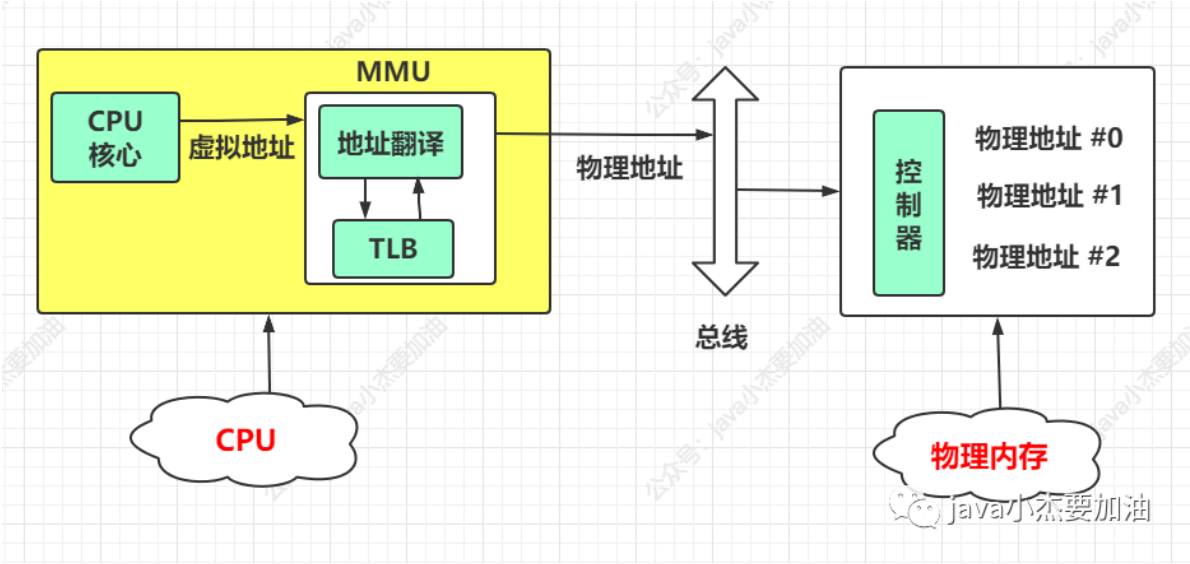
虚拟内存

一些概念

- **物理地址**：逻辑上，我们可以把物理内存看成一个大数组，其中每个字节都可以通过与之对应的地址进行访问，这个地址就叫做**物理地址**
- **虚拟地址** ： 应用程序在运行时使用的地址

CPU翻译虚拟地址的过程大概如图所示



他们的包含关系如下：**cpu包含MMU,MMU包含TLB**

- CPU
- **TLB(转址旁路缓存 Translation Lookaside Buffer)**:加速地址翻译的过程
- **MMU(内存管理单元 Memory Management Unit)**: 负责虚拟地址到物理地址的转换

平常加载程序的顺序是

1. 操作系统把程序从**磁盘**加载到**内存**中（程序一开始是在磁盘中存放的）
2. CPU去执行程序的第一条指令但是这个**指令现在在物理内存**中
3. cpu取指令取的是该指令的**虚拟地址**，由MMU翻译为**物理地址**
4. 这个读物理地址的请求将通过**总线**，传送到相应的**物理内存**中，然后**物理内存**把该指令发送给CPU

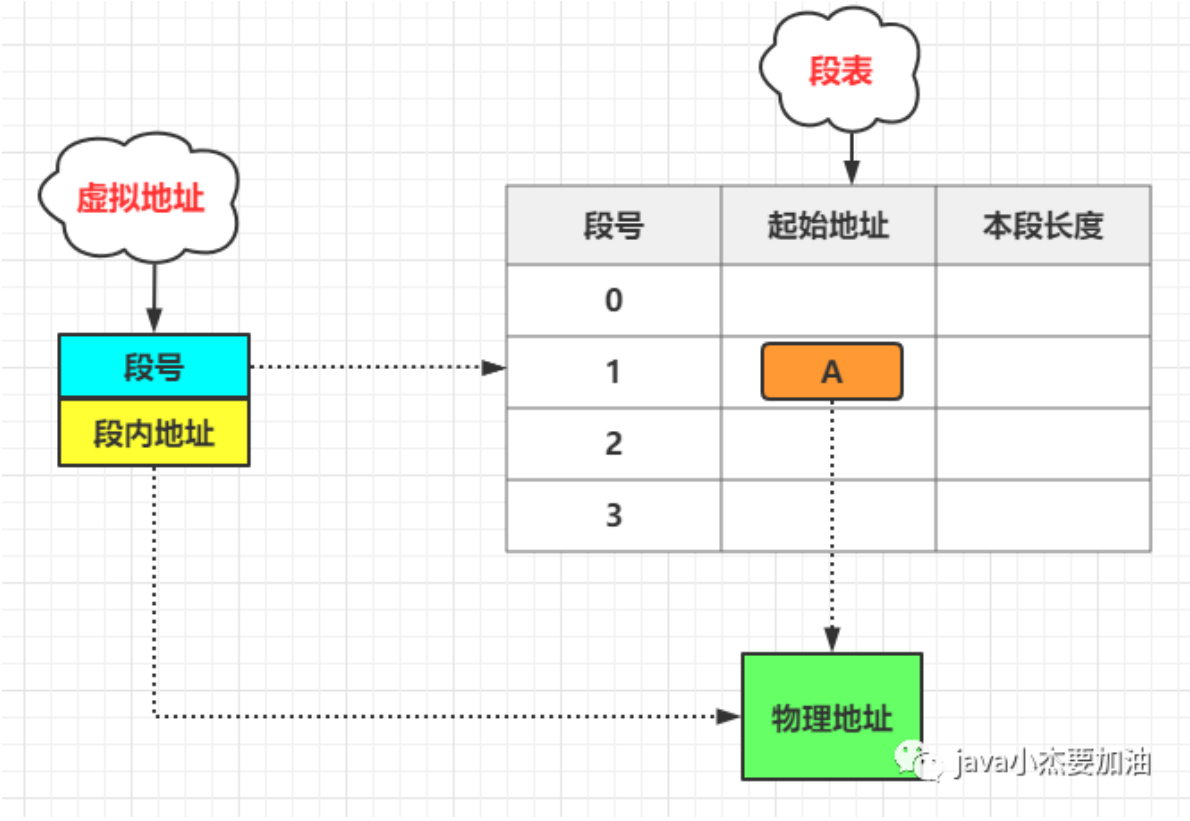
内存分段

内存分段的概念

- 操作系统以“段”（一段连续的物理内存）的形式管理/分配物理内存
- 应用程序的虚拟地址空间由若干个大小不同的段组成：代码段、数据段等等
- 当**CPU访问虚拟地址**中的**某一个段**的时候，MMU会通过查询**段表**来得到该段对应的**物理地址**
- **虚拟地址**:
    - **段号**： 标志着该虚拟地址属于整个虚拟地址空间中的哪一段
    - **段内地址（段内偏移）**： 相对于该段起始地址的偏移量

> 当 cpu 读取指令时，发现指令的地址是**虚拟地址**，那么CPU中的MMU 首先判断这个**段号**是否合法，如果合法， 则通过 **段表基址寄存器** 找到**段表**的位置，通过虚拟地址中的**段号**，找到该段的**起始地址**，再加上**段内地址**（段内偏移），就可以得到**最终的物理地址**

- 在分段机制下，**虚拟内存**和**物理内存**都划分成了**不同的段**

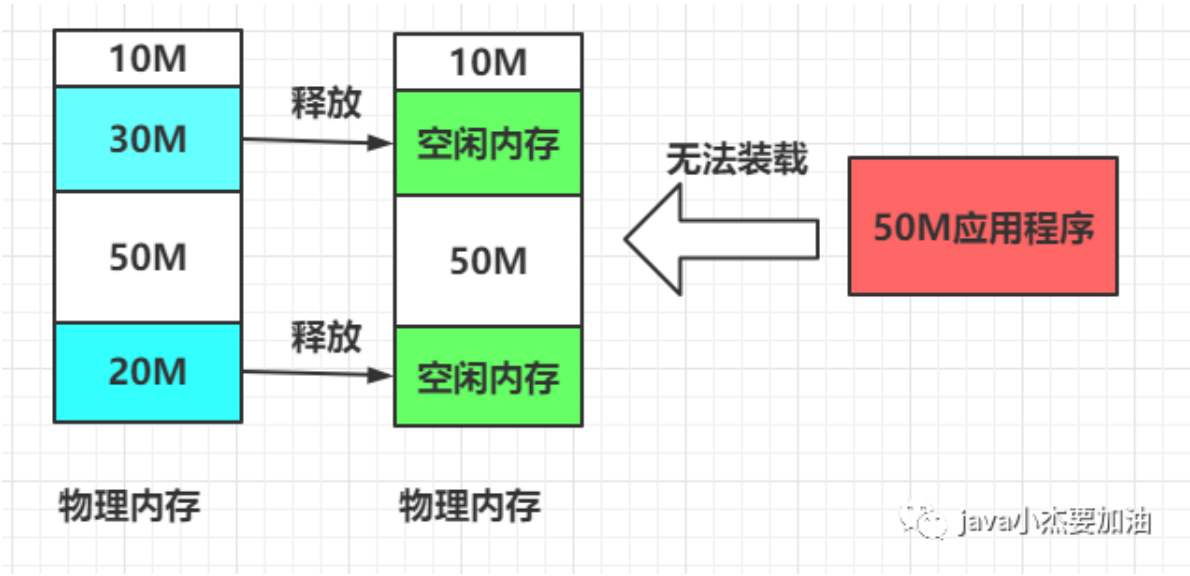


内存分段的问题

分段映射很简单，但是会导致**内存碎片**跟**内存交互效率低。**

- **内存碎片**
  - **内部碎片**：已经被分配出去的的内存空间不经常使用，并且分配出去的内存空间大于请求所需的内存空间。
  - **外部碎片**：指可用空间还没有分配出去，但是可用空间由于大小太小而无法分配给申请空间的新进程的内存空间空闲块。

在虚拟地址空间中，相邻的段所对应的物理内存空间可以不相邻，操作系统能够实现物理内存资源的离散分配，但是这种段式分配方式容易导致在物理内存上出现**外部碎片。**



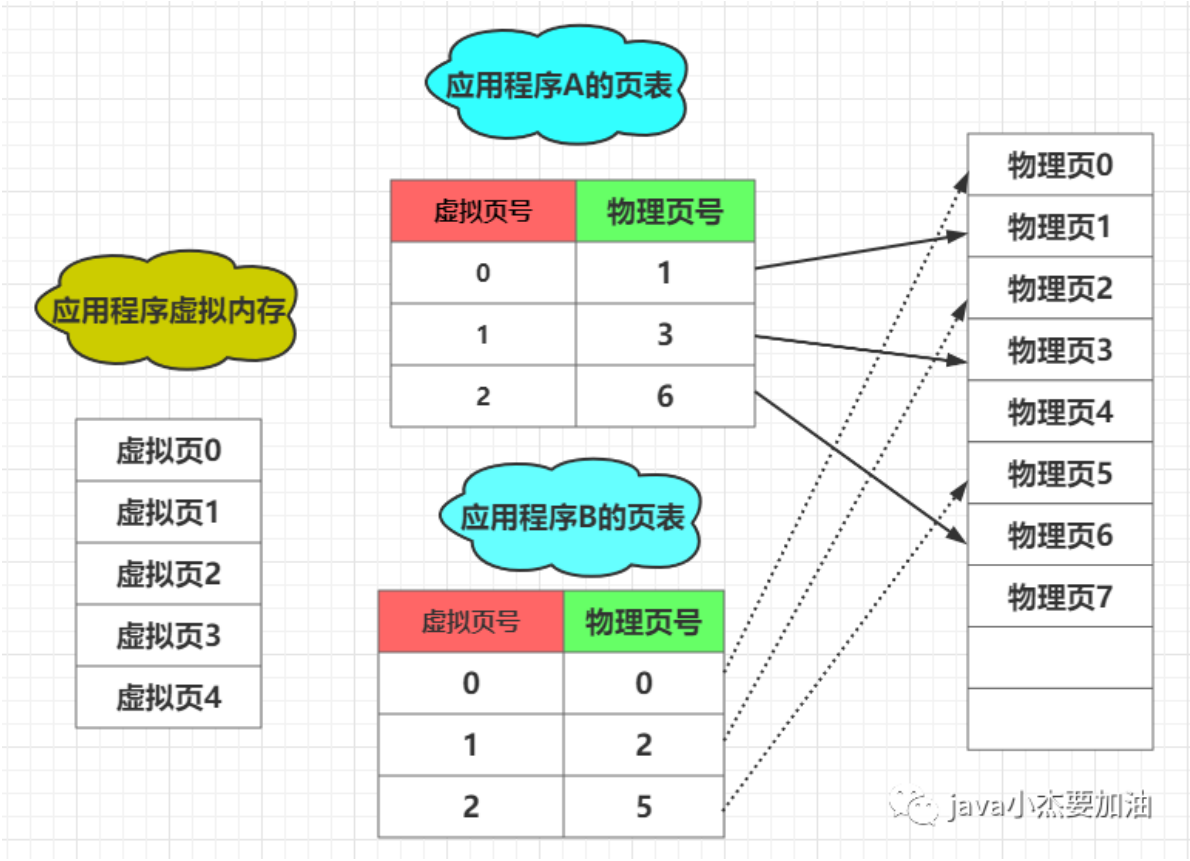
内存分页

概念

- **基本思想：**
- 将应用程序的**虚拟地址**空间划分为**连续的**、**等长的**虚拟页（4K）
  - **物理地址**也是划分为**连续的**、**等长的**的物理页
  - **物理页**和**虚拟页**页长**固定且相等**
  - > 每个固定大小的尺寸称之为**页Page**，在 Linux 系统中Page = 4KB。

之所以这样构造是因为会使**操作系统**很方便的为**每个应用程序**构造**页表**，即虚拟页和物理页**映射关系表**

- 在分页机制下，应用程序虚拟地址空间中的任意虚拟页可以被映射到物理内存中的任意物理页上，可以避免**外部碎片**的问题
- 分页机制下的**虚拟地址**也由两部分组成：**虚拟页号：** **页内偏移量：**



查询过程

> 1. MMU首先解析**虚拟地址**中的**虚拟页号**，检查这个虚拟页号**是否合法**，通过这个**虚拟页号**取该应用程序的**虚拟页表**中找到对应条目（页表起始地址放在**页表基地址寄存器**）
> 2. 然后取出该条目中的**物理页号**
> 3. 最后用该**物理页号**对应的**物理起始地址**加上**虚拟地址**中的**页内偏移**得到**最终的物理地址**

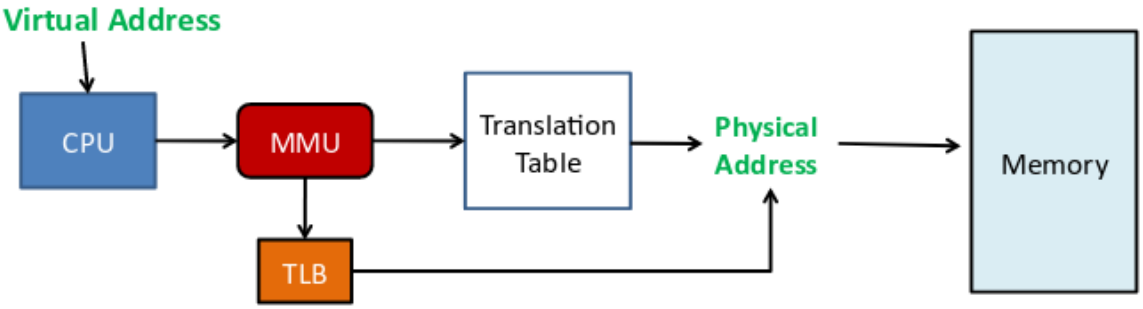
TLB

局部性原理

- **时间局部性：** 如果执行了程序中的某条指令，那么不久后这条指令很有可能再次执行，如果某个数据被访问过，不久后该数据很可能再次被访问（因为程序中存在大量的循环）
- **空间局部性：** 一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也很有可能会被访问（因为很多数据在内存中都是连续存放的）

概念

Translation Lookaside Buffer，属于CPU内部的一个模块，TLB是MMU的一部分，实质是cache，它所缓存的是最近使用的数据的页表项（虚拟地址到物理地址的映射）。他的出现是为了加快访问数据（内存）的速度，减少重复的页表查找。当然它不是必须要有的，但有它，速度就更快。

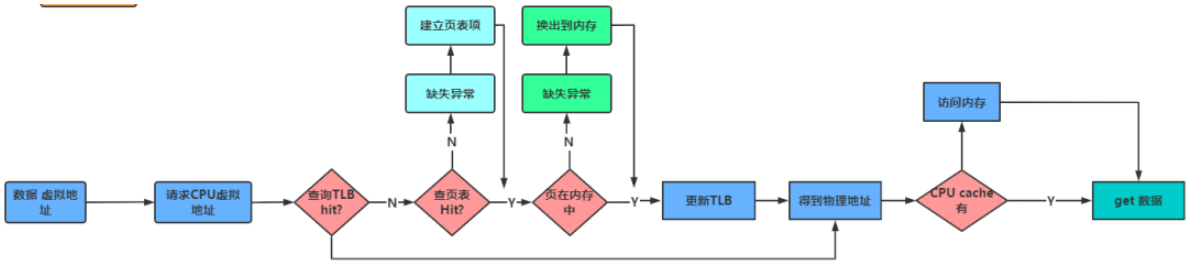


硬件架构与数据结构

- TLB 硬件采用分层架构，**分为L1、L2两层。**
- - LI又分为数据TLB和指令TLB，分别缓存数据和指令的地址翻译
    - L2不区分数据和指令
- TLB**缓存**了虚拟页号和物理页号的映射关系，类似map，key是虚拟页号，value是物理页号。
- - 如果在TLB中找到则称为**TLB命中**
    - 没有找到则称之为**TLB未命中**

加入的TLB查询过程

如果一个需要访问内存中的一个数据，给定这个数据的虚拟地址，查询TLB，发现有hit，直接得到物理地址，在内存根据物理地址取数据。如果TLB没有这个虚拟地址miss，那么只能费力的通过页表来查找了。日常CPU读取一个数据的流程如下：



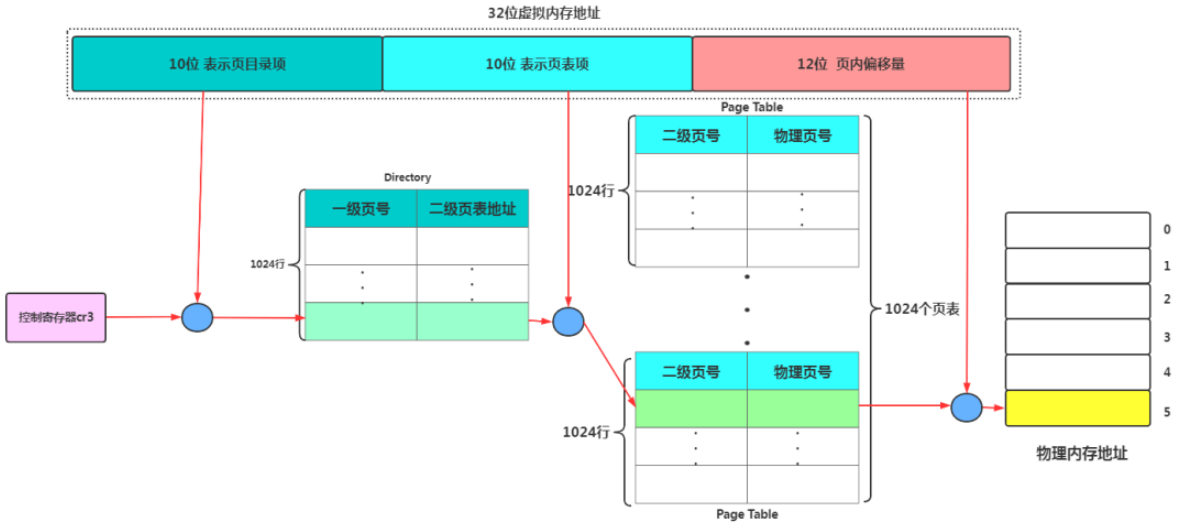
当进程地址空间进行了**上下文切换**时，比如现在是进程1运行，TLB中放的是进程1的相关数据的地址，突然切换到进程2，TLB中原有的数据不是进程2相关的，此时TLB刷新数据有两种办法。

1. **全部刷新**：很简单，但花销大，很多不必刷新的数据也进行刷新，增加了无畏的花销。
2. **部分刷新**：根据标志位，刷新需要刷新的数据，保留不需要刷新的数据。

多级页表

32位操作系统环境下进程可操作的虚拟地址是4GB，假设一个虚拟页大小为4KB，那需要4GB/4KB = **2^20** 个页信息。一行页表记录为4字节，**2^20**等价于4MB页表存储信息。这只是一个进程需要的，如果10个、100个、1000个呢？仅仅是页表存储都占据超大内存了。

在32位的操作系统中将将4G空间分为 1024 行页目录项目(4KB)，每个页目录项又对应1024行页表项。如下图所示：



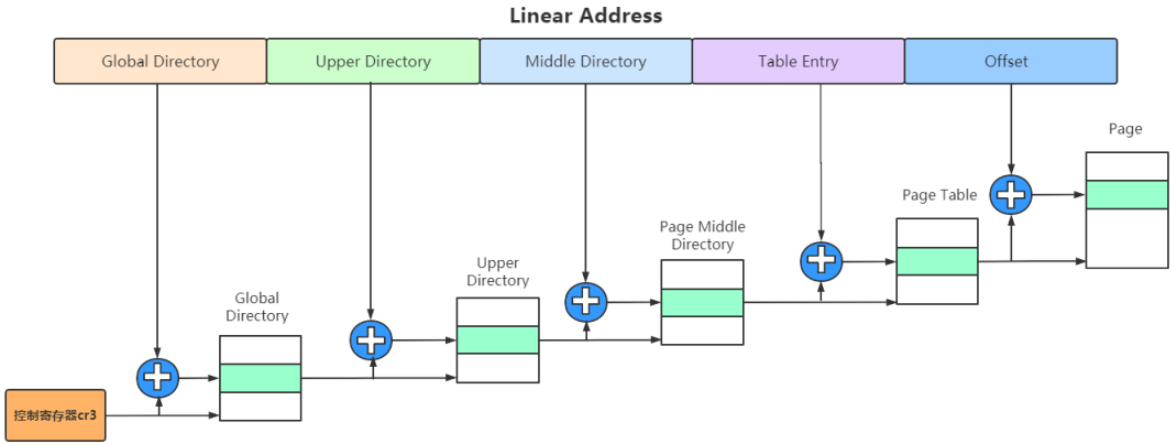
- 根据一级页号查找到物理页号，这个物理页号里面装的是二级页表的地址，找到此地址后，在根据二级页号 找到物理地址，此物理地址在加上页内偏移量则为最终的物理地址

多层分页的弊端就是访问时间的**增加**。

1. 使用页表时读取内存中一页内容需要2次访问内存，访问页表项 + 并读取的一页数据。
2. 使用二级页表的话需要三次访问，访问页目录项 + 访问页表项 + 访问并读取的一页数据。访存次数的增加也就意味着访问数据所花费的总时间增加。

而对于64位系统，二级分页就无法满足了，Linux 从2.6.11开始采用四级分页模型。

1. Page Global Directory 全局页目录项
2. Page Upper Directory 上层页目录项
3. Page Middle Directory 中间页目录项
4. Page Table Entry 页表项
5. Offset 偏移量。



换页与缺页异常

换页

> 虚拟内存中的**换页**：当物理内存容量不够的时候，操作系统应当把若干物理页的内容写到磁盘这种大容量的地方，然后**回收物理页**并继续使用

举例：有个应用程序A，A的虚拟页K对应物理页V，这个时候，操作系统想回收物理页V，要怎么做呢?

- 操作系统把V写到磁盘上
- 并且在A的页表中除去虚拟页K和物理页V的映射，同时记录物理页V被换到磁盘上的对应的位置

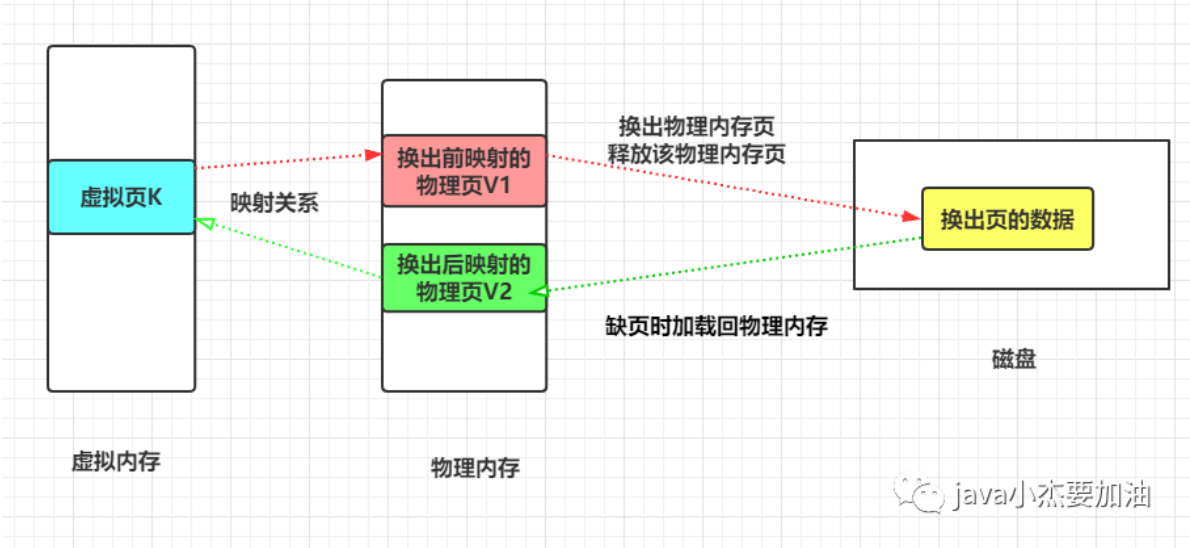
以上这两部被称为物理页V的**换出**

缺页异常

> 缺页异常是换页机制能够工作的前提，当应用程序访问**已经分配但是未映射至物理内存**的虚拟页时，就会触发缺页异常

- 如何解决：通过**换入**
-
    - cpu会运行操作系统预先设置的缺页异常处理函数，该函数会找到一个空闲的物理页，
        - 将以前写入到磁盘上的内容重新加载到该空闲的物理页
        - 然后将虚拟地址和此物理地址映射起来

处理完这一切后，cpu回到发生缺页异常的地方继续运行

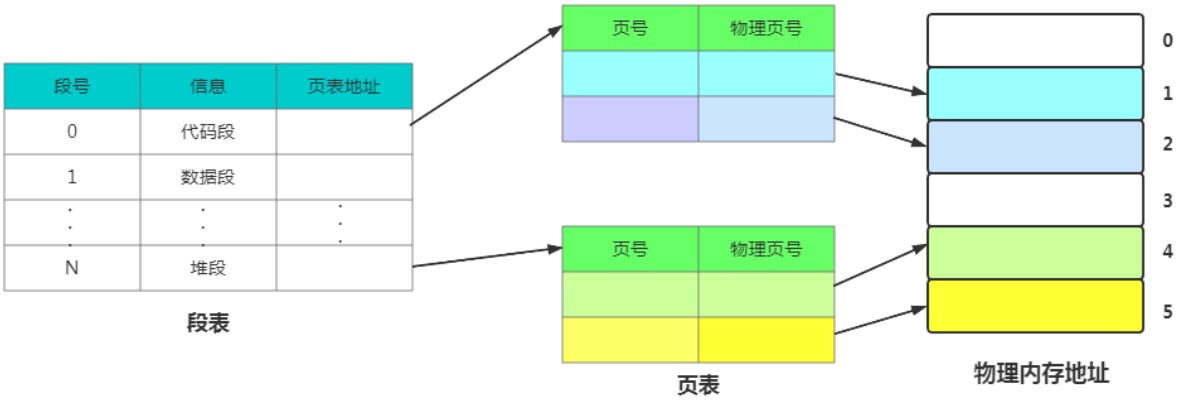


段页式管理

内存分段跟**内存分页**不是对立的，这俩可以组合起来在同一个系统中使用的，那么组合起来后通常称为**段页式内存管理**。段页式内存管理实现的方式：

1. 先对数据不同划分出不同的段，也就是前面说的分段机制。
2. 然后再把每一个段进行分页操作，也就是前面说的分页机制。
3. 此时 地址结构 = 段号 + 段内页号 + 页内位移。

每一个进程有一张段表，每个段又建立一张页表，段表中的地址是页表的起始地址，而页表中的地址则为某页的物理页号。



逻辑地址与**线性地址**。

1. **逻辑地址**：指的是没被段式内存管理映射的地址。
2. **线性地址**：通过段式内存管理映射且页式内存管理转换前的地址，俗称虚拟地址。

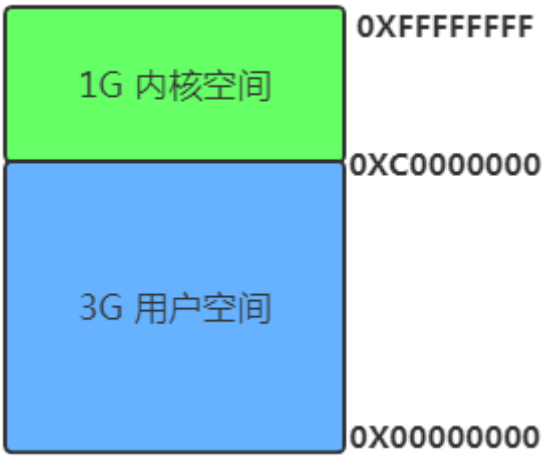
目前 Intel X86 CPU 采用的是内存分段 + 内存分页的管理方式，其中分页的意思是在由段式内存管理所映射而成的的地址上再加上一层地址映射。

虚拟地址翻译为物理地址的步骤变为

- 根据逻辑地址取出其中的段号，判断这个段号是否正常
- 如果正常，则找到该段号对应的页表初始地址
- 根据页号是否正常，若正常则根据页号找到物理初始地址，在加上页内偏移量则找到真正的物理地址

Linux 内存管理

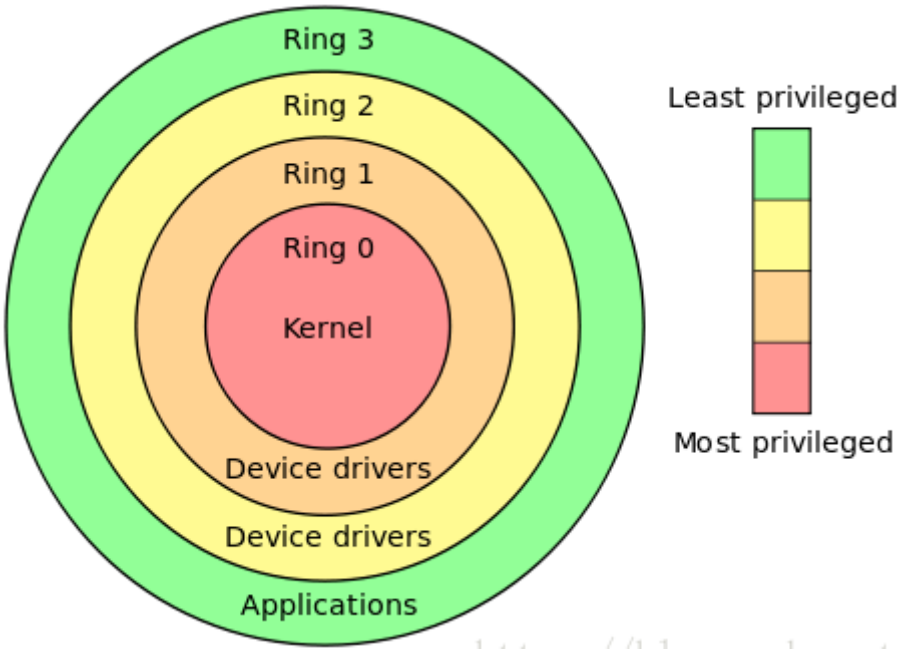
先说结论：Linux系统基于X86 CPU 而做的操作系统，所以也是用的段页式内存管理方式。



我们知道32位的操作系统可寻址范围是4G，操作系统会将4G的可访问内存空间分为**用户空间**跟**内核空间**。

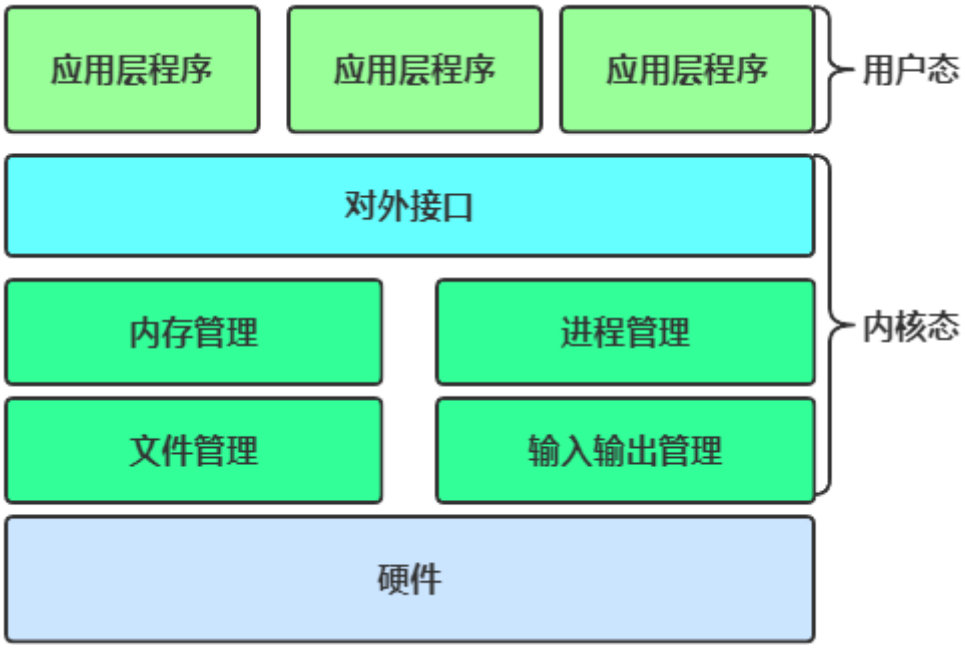
1. **内核空间**：操作系统内核访问的区域，独立于普通的应用程序，是受保护的内存空间。内核态下CPU可执行任何指令，可自由访问任何有效地址。
2. **用户空间**：普通应用程序可访问的内存区域。被执行代码会受到CPU众多限制，进程只能访问映射其地址空间的页表项中规定的在用户态下可访问页面的虚拟地址。

那为啥要搞俩空间呢?现在嵌入式环境跟以前的WIN98系统是没有区分俩空间的，须知俩空间是CPU分的，而操作系统是在上面运行的，单一用户、单一任务服务的操作系统，是没有分所谓用户态和内核态的必要。用户态和内核态是因为有多用户，多任务的需求，然后在CPU硬件厂商配合之后，产生的一个操作系统解决多用户多任务需求的方案。方案就是**限制**，通过硬件手段（也只能硬件手段才能做到），限制某些代码，使其无法控制整个物理硬件，进而使各个不同用户，不同任务的代码，无权修改整个物理硬件，再进而保护操作系统的核心底层代码和其他用户的数据不被无意或者有意地破坏和盗取。



后来研究者根据CPU的运行级别，分成了Ring0~Ring3四个级别。Ring0是最高级别，Ring1次之，Rng2更次之，拿Linux+x86来说，操作系统内核的代码运行在最高运行级别Ring0上，可以使用特权指令，控制中断、修改页表、访问设备等。 应用程序的代码运行在最低运行级别上Ring3上，不能做受控操作，只能访问用户被分配的空间。如果要做访问磁盘跟写文件等操作，那就要通过执行系统调用函数，执行系统调用的时候，CPU的运行级别会发生从Ring3到Ring0的切换，并跳转到系统调用对应的内核代码位置执行，这样内核就为你完成了设备访问，完成之后再从Ring0返回Ring3。这个过程也称作**用户态和内核态的切换**。

用户态想要使用计算机设备或IO需通过系统调用完成sys call，系统调用就是让内核来做这些操作。而系统调用是影响整个当前进程上下文的，CPU提供了个软中断来是实现保护线程，获取系统调用号跟参数，交给内核对应系统调用函数执行。



页面置换算法

当发生缺页中断时，如果当前内存中并没有空闲的页面，操作系统就必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。用来选择淘汰哪一页的规则叫做页面置换算法，我们可以把页面置换算法看成是淘汰页面的规则。

OPT 最佳页面置换算法

最佳(Optimal, OPT)置换算法所选择的被淘汰页面将是以后永不使用的，或者是在最长时间内不再被访问的页面,这样可以保证获得最低的缺页率。但由于人们目前无法预知进程在内存下的若干页面中哪个是未来最长时间内不再被访问的，因而该算法无法实现。一般作为衡量其他置换算法的方法。

FIFO（先进先出页面置换算法）

总是淘汰最先进入内存的页面，即选择在内存中驻留时间最久的页面进行淘汰。

LRU（最近最少使用页面置换算法）

Least Currently Used

LRU算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 T，当须淘汰一个页面时，选择现有页面中其 T 值最大的，即最近最久未使用的页面予以淘汰。

虽然 LRU 在理论上是可以实现的，但是从长远看来代价比较高。为了完全实现 LRU，会在内存中维护一个所有页面的链表，最频繁使用的页位于表头，最近最少使用的页位于表尾。困难的是在每次内存引用时更新整个链表。在链表中找到一个页面，删除它，然后把它移动到表头是一个非常耗时的操作，即使使用`硬件`来实现也是一样的费时。

LFU（最少使用页面置换算法）

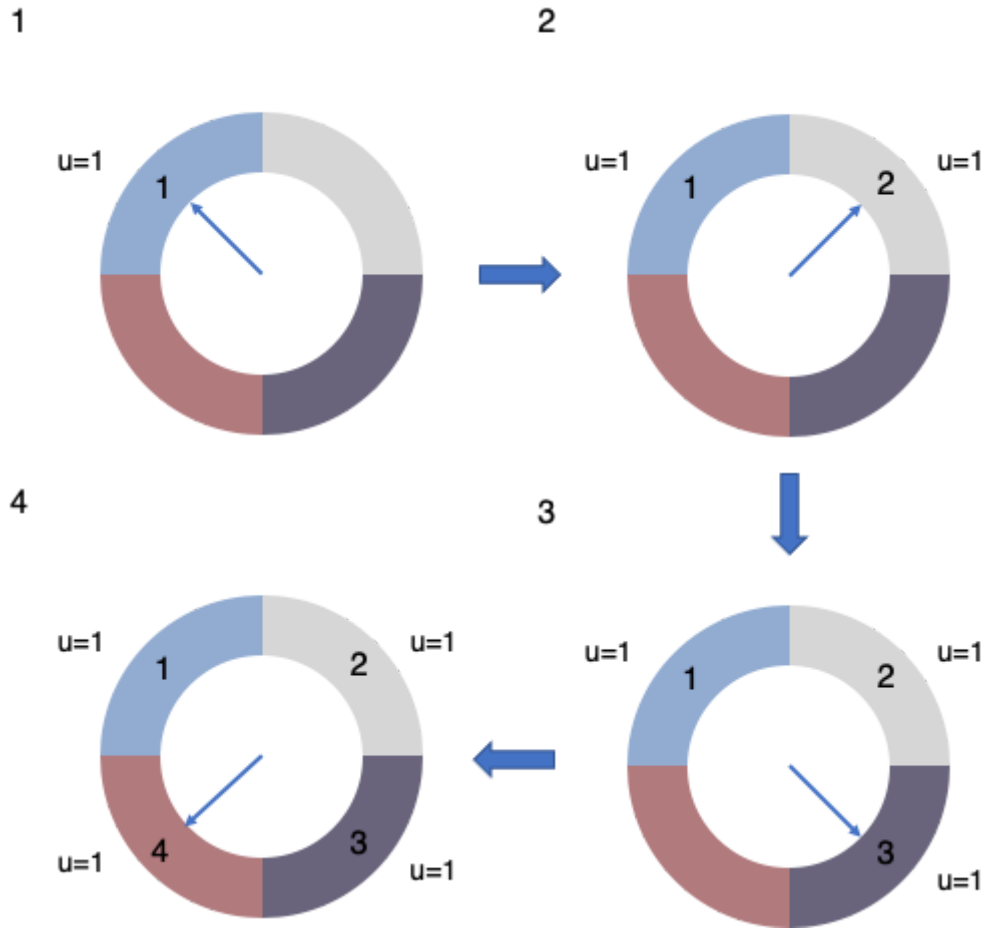
Least Frequently Used

该置换算法选择在之前时期使用最少的页面作为淘汰页。

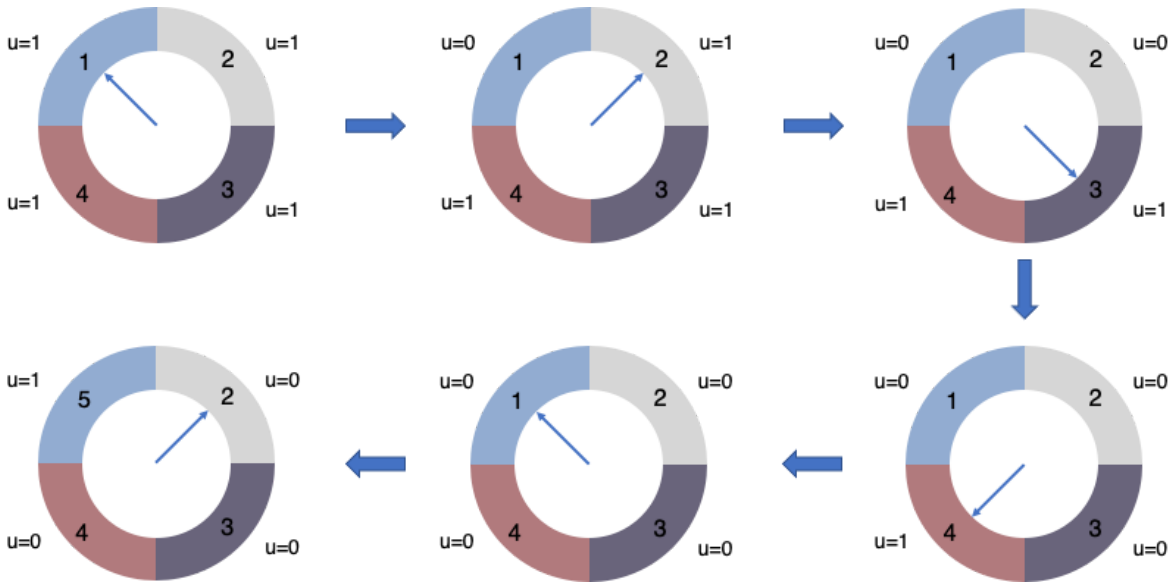
时钟置换算法

时钟置换算法可以认为是一种最近未使用算法，即逐出的页面都是最近没有使用的那个。我们给每一个页面设置一个标记位u，u=1表示最近有使用u=0则表示该页面最近没有被使用，应该被逐出。

按照1-2-3-4的顺序访问页面，则缓冲池会以这样的一种顺序被填满：



注意中间的指针，就像是时钟的指针一样在移动，这样的访问结束后，缓冲池里现在已经被填满了，此时如果要按照1-5的顺序访问，那么在访问1的时候是可以直接命中缓存返回的，但是访问5的时候，因为缓冲池已经满了，所以要进行一次逐出操作，其操作示意图如下：



最初要经过一轮遍历，每次遍历到一个节点发现u=1的，将该标记位置为0，然后遍历下一个页面，一轮遍历完后，发现没有可以被逐出的页面，则进行下一轮遍历，这次遍历之后发现原先1号页面的标记位u=0，则将该页面逐出，置换为页面5，并将指针指向下一个页面。

假设我们接下来会访问2号页面，那么可以直接命中指针指向的页面，并将这个页面的标记为u置为1。

改进型时钟置换算法

但是考虑一个问题，数据库里逐出的页面是要写回磁盘的，这是一个很昂贵的操作，因此我们应该优先考虑逐出那些没有被修改的页面，这样可以降低IO。

因此在时钟置换算法的基础上可以做一个改进，就是增加一个标记为m，修改过标记为1，没有修改过则标记为0。那么u和m组成了一个元组，有四种可能，其被逐出的优先顺序也不一样：

- （u=0，m=0）没有使用也没有修改，被逐出的优先级最高；
- （u=1，m=0）使用过，但是没有修改过，优先级第二；
- （u=0，m=1）没有使用过，但是修改过，优先级第三；
- （u=1，m=1）使用过也修改过，优先级第四。

在内存中的每个页必定是这四个类页面之一，在进行页面置换时，可采用与简单Clock算法相类似的算法，其差别在于该算法须同时检查访问位与修改位，以确定该页是四类页面中的哪一种。其执行过程可分成以下三步：

- 从指针所指示的当前位置开始，扫描循环队列，寻找A=0且M=0的第一类页面，将所遇到的第一个页面作为所选中的淘汰页。在第一次扫描期间不改变访问位A
- 如果第一步失败，即查找一周后未遇到第一类页面，则开始第二轮扫描，寻找A=0且M=1的第二类页面，将所遇到的第一个这类页面作为淘汰页。在第二轮扫描期间，将所有扫描过的页面的访问位都置0
- 如果第二步也失败，亦即未找到第二类页面，则将指针返回到开始的位置，并将所有的访问位复0。然后重复第一步，如果仍失败，必要时再重复第二步，此时就一定能找到被淘汰的页该算法与简单Clock算法比较，可减少磁盘的I/O操作次数。但为了找到一个可置换的页，可能须经过几轮扫描。换言之，实现该算法本身的开销将有所增加。