# CS61C Lec22-24 Virtual Memory Note



## 1. 3 Probelms with Memory

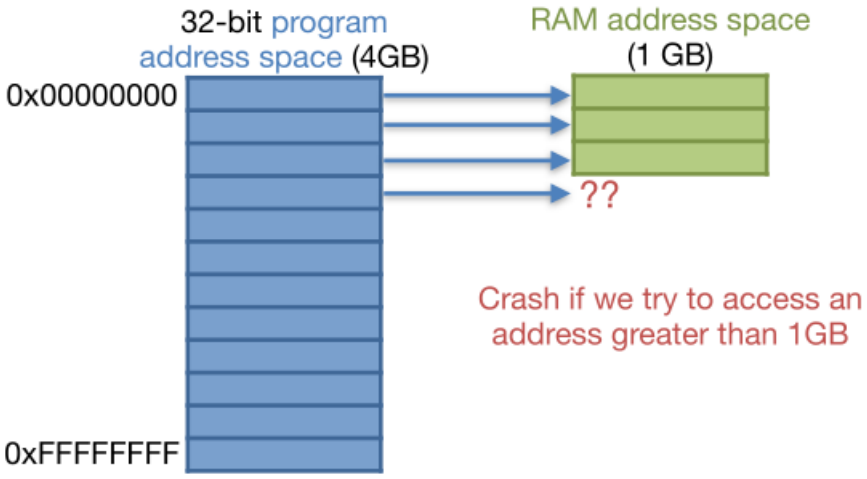
为什么要引入虚拟内存，先从三个问题说起

### #1: Not Enough Space

RISC-V 32 的版本用32bit 对地址空间进行编址，那么可访问多大的内存呢?

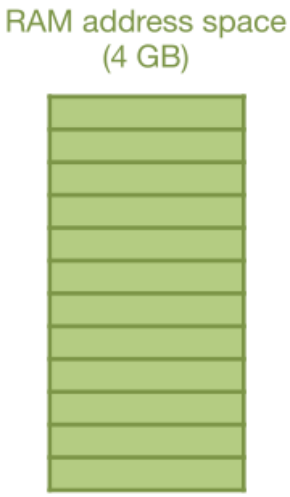
- $2^{32}bytes = 2^2 * 2^{30}bytes = 4GB$

如果电脑的物理内存RAM 只有1GB 大小的话，如下图一一映射访问地址，会发现超过1GB 以后的RAM 没有对应的地址了

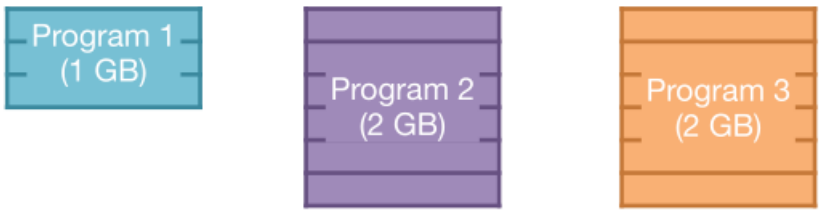



### #2: Holes in Address Space

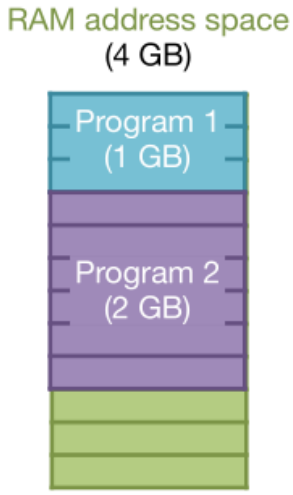
假设我们的RAM是4GB 大小



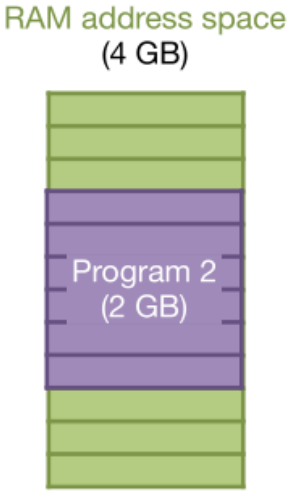

现在有三个程序，所需RAM空间分别为1GB, 2GB, 2GB




假设我们先顺序运行程序1 ， 程序2




然后再退出程序1

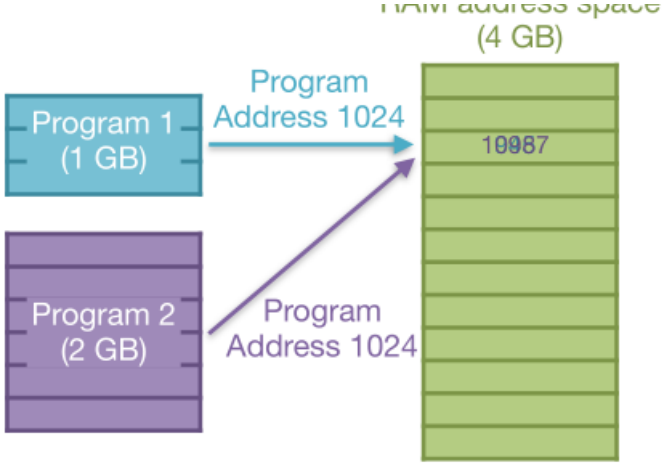



现在RAM剩下2 GB 的空间，而程序3刚好也是2GB ，但是由于程序运行是需要连续的地址空间，RAM的2GB是被分段的，一个1GB 在上面，一个1GB在下面，所以就算RAM空间足够容纳程序3，也运行不了(Memory Fragmentation)

## #3: Ensuring Protection from Other Programs

第三个问题是要保护进程间的孤立性，我们之前说了每个程序都有自己的地址空间，**在每个程序自己看来，它们能访问任何32bit 的地址**

例如你写了2个程序，程序1 需要访问address 1024处的值，程序2也可能有段代码是需要访问address 1024处的值，那么它们会发生相互覆盖



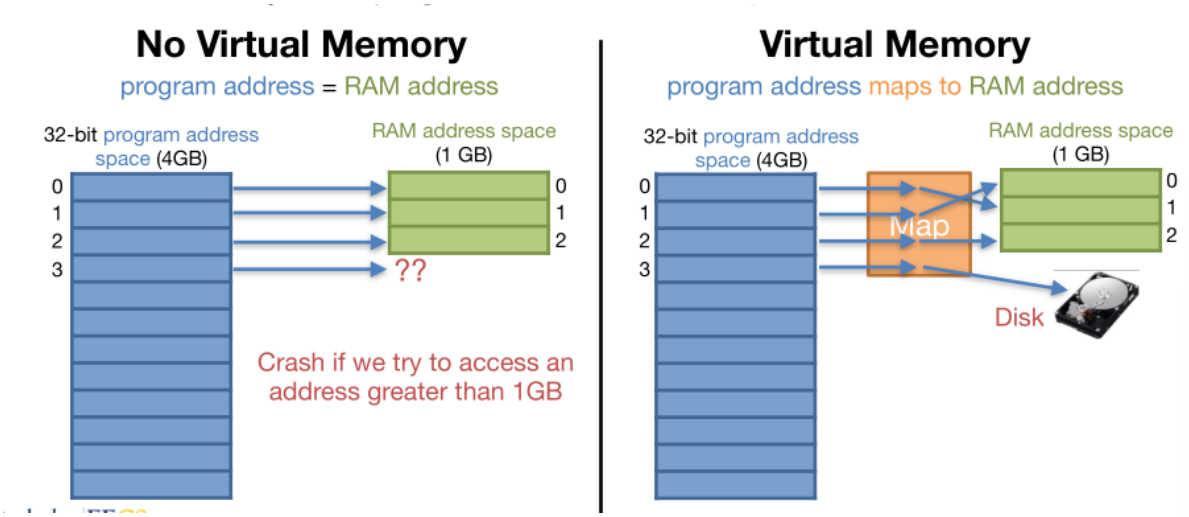

这也就是当前存在的三个问题，下面我们将通过引入虚拟内存的方式解决问题

---

## 2. Solving 3 Problems with Virtual Memory

虚拟内存就是假想中的内存，真实并不存在，也就是让每个程序都认为自己有一个可访问的32bit 地址空间，虚拟内存与真实物理内存之间通过某种映射一一对应。

### Solving Problem #1: Not Enough Memory




当真实的RAM空间不够时，也就是当虚拟内存大于物理内存时，多于的部分可以映射到磁盘(disk)上，相当于一个指向磁盘的指针

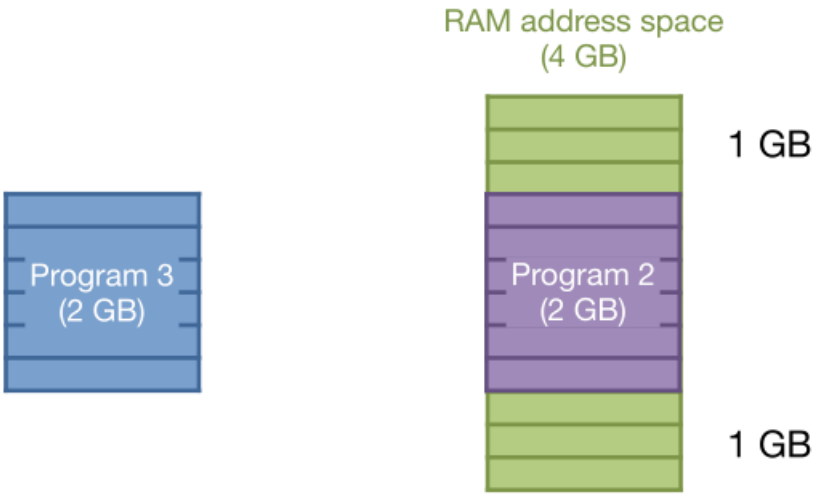
当需要的数据在磁盘上时，且RAM 已满时，从磁盘将数据载入RAM ，并选择一个RAM address替换

#### VM Performance

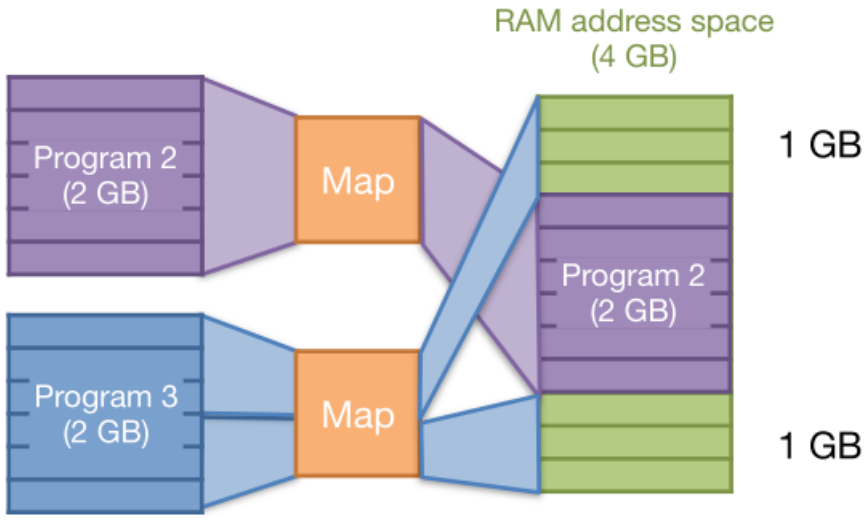
当需要的数据在磁盘上不在RAM 里时，会造成程序性能下降，因为访问磁盘是极其昂贵的，这也就是为什么买一台RAM 足够大的电脑性能更佳

### Solving Problem #2: Holes in Address Space

当出现Memory Fragmentation时




我们可以对虚拟内存与物理内存之间建立一种映射关系：

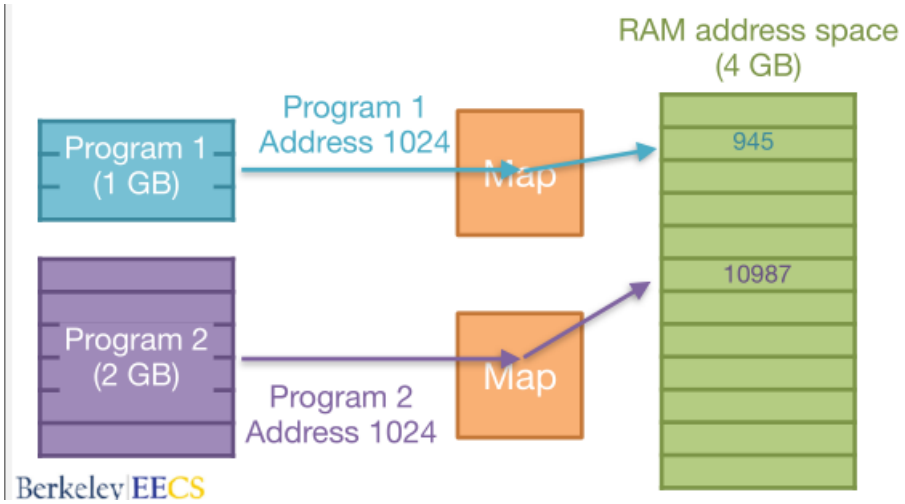



将程序3上部分映射到RAM 上方1GB 空间，下部分映射到RAM 下方1GB 空间

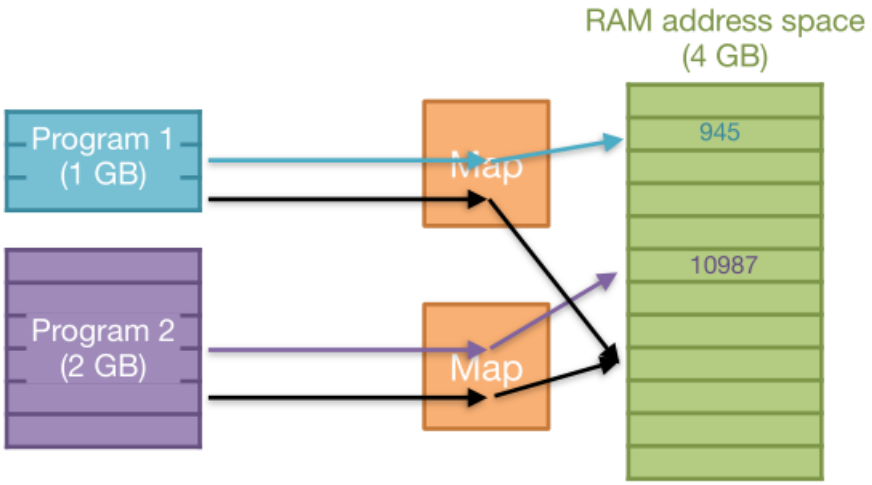
在程序3 看来，在它的虚拟内存中，地址空间仍然是连续的，通过映射解决Memory Fragmentation

## Solving Problem #3: Keeping Programs Secure

程序1 和程序2 访问相同的address 1024，在它们看来它们能够访问32bit 地址空间的任意位置，我们只需将它们相同的两个虚拟地址 映射到 真实物理内存的不同地址



当然，如果两个程序共享一些library(例如两个程序都有库，那么它们也可以映射到相同的物理地址（如下图黑色线条）




## 3. How Does VM Work?: Translation

例如一个RISC-V 指令

```
lw t1, 1024(x0)
```

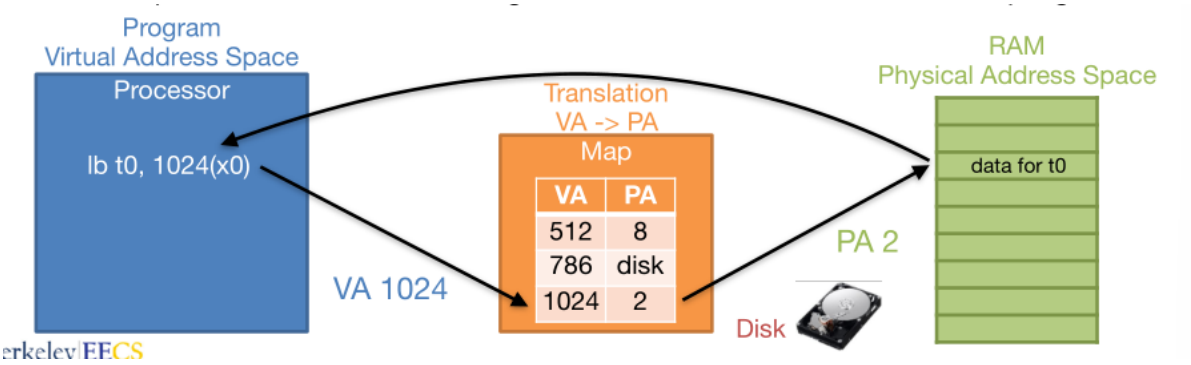
是把虚拟地址1024处的值加载至t1寄存器，需要通过映射，把VA 1024(VA is Virtual Address)转换为PA(Physical Address)，因为我们的RAM 是以PA 进行编址的

然后通过PA 找到想要的数据，这里有两种情况：

- VA 对应的PA 在RAM中，在RAM找到数据
- VA 对应的PA 在Disk中，将数据加载至RAM中，更新Translation Map

最后返回至t1寄存器

VA 转 PA 的过程则成为translation




## 4. Page Tables

定义：从虚拟地址 到 物理地址的 Translation Map 称为 Page Tables

Page Table Entry: 对于每条VA-->PA 的条目称为PTE

假设我们对每一个虚拟地址(1 word)都对应一个物理地址的映射的话，Page Tables需要多少条目?

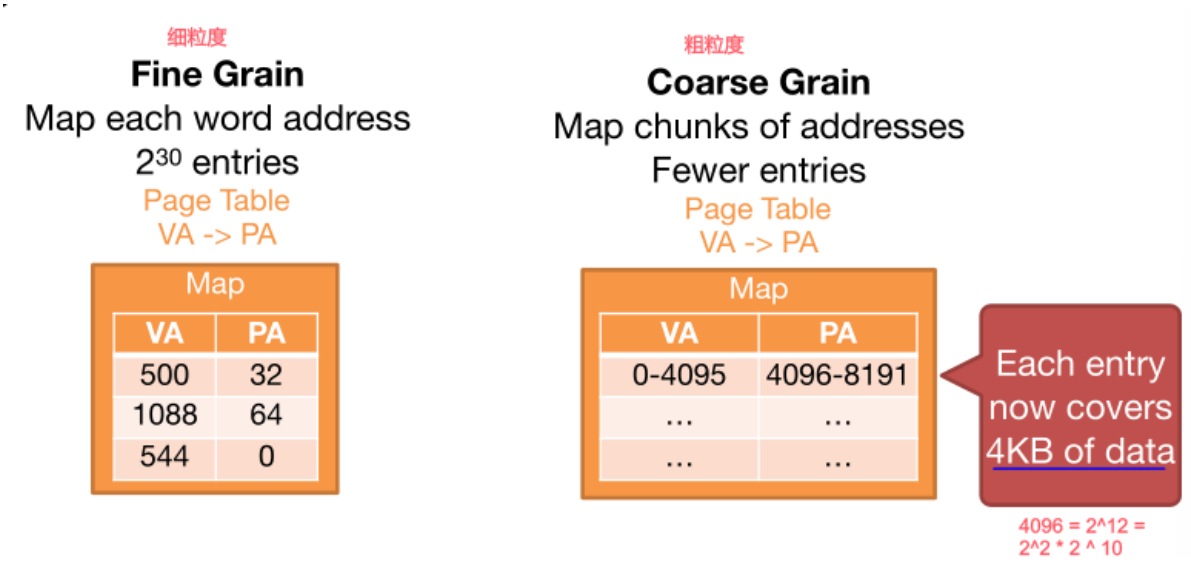
$32bit$编址有$2^{32}byte$的地址空间

$$1word = 4byte = 2^2byte$$

那么$PageTable$有条目数：$\frac{2^{32}}{2^2} = 2^{30}$条

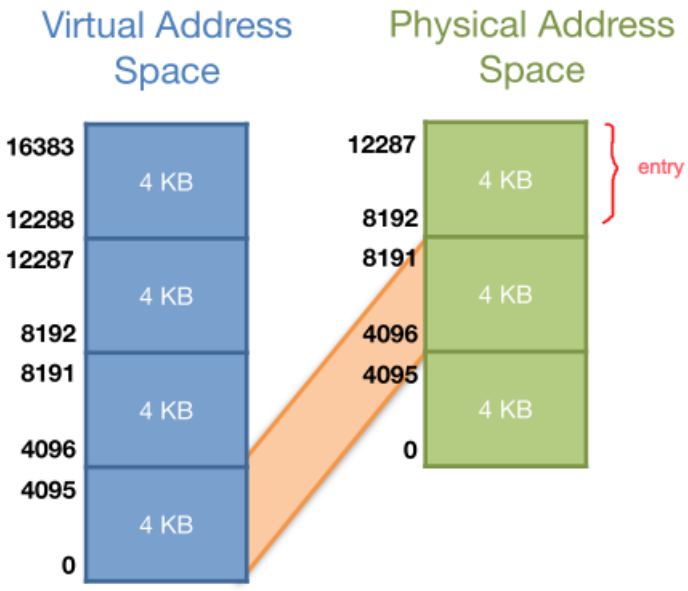
Q：如何去减少Page Table 中的条目数?

A：可以不细分到对每条VA 进行映射，而是对某个范围的VA 进行映射

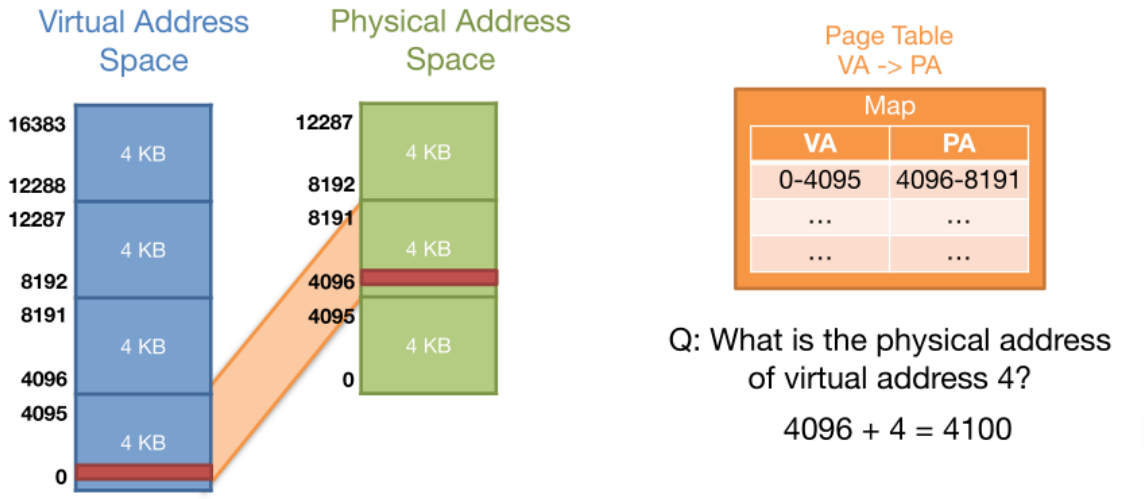



例如将虚拟地址0-4095 作为一个entry，那么条目数下降为：

$$\frac{2^{32}byte}{4KB} = \frac{2^{32}byte}{2^2 * 2^{10}byte} = 2^{20}条$$




**一个entry 和 一页page其实是同一个东西，一个完整的Page Table 有多少page 就相当于有多少Entry**



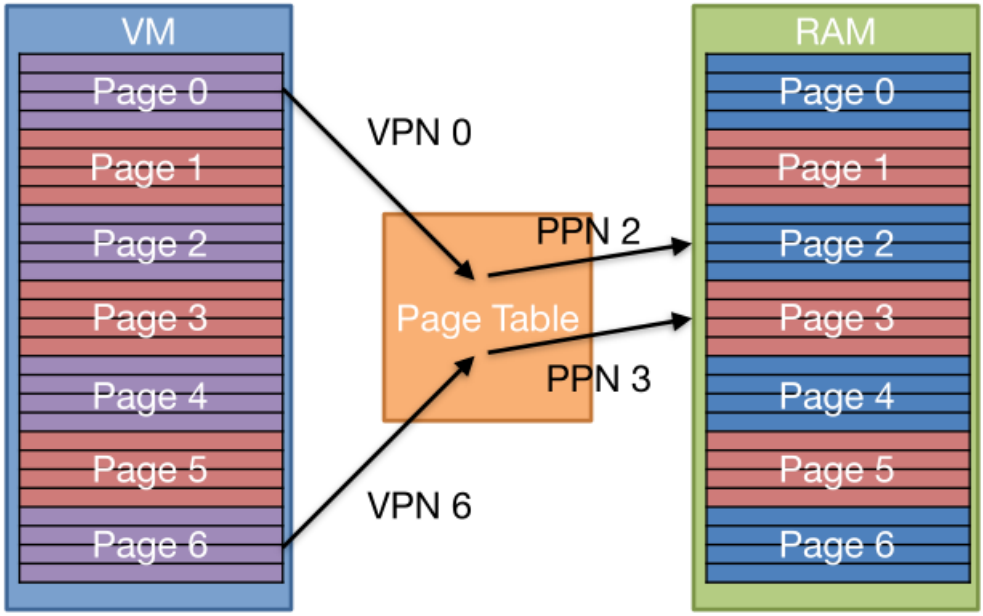

与Cache 相同，为了更好的利用时空局部性，现在当我们从磁盘上加载数据时，不再是加载单个word，而是加载一整页page(entry)：

例如要从磁盘加载物理地址 9120, OS 会将bytes 8192 - 12287 从磁盘载入RAM

---

## 5. Address Translation

将虚拟内存和物理内存分页，通过Page Table映射进行对应：




那么Translation的过程需要两点:

- Page Number
  - 索引，第几页
  - 所需编址的bit数 = $log_2 PageCount$
- Page offset
  - 索引，在该页的哪个位置
  - 所需编址的bit 数=$log_2 PageSize$

一些术语：

VPN: Virtual Page Number，虚拟内存的分页数量

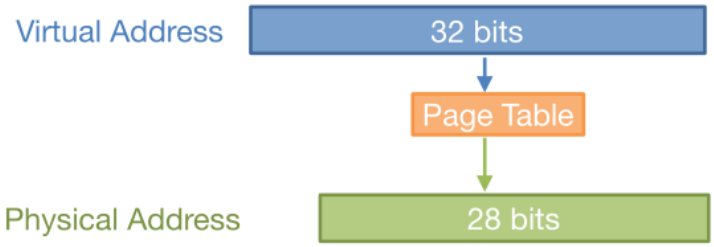
PPN: Physical Page Number, 物理内存的分页数量

那么Virtual Address可以拆分为 VPN + Page offset , Physical Address 可以拆分为 PPN + Page offset

**需要注意的是，VA 的 Page offset 与 PA 的Page offset 对应相同，无需Translation**

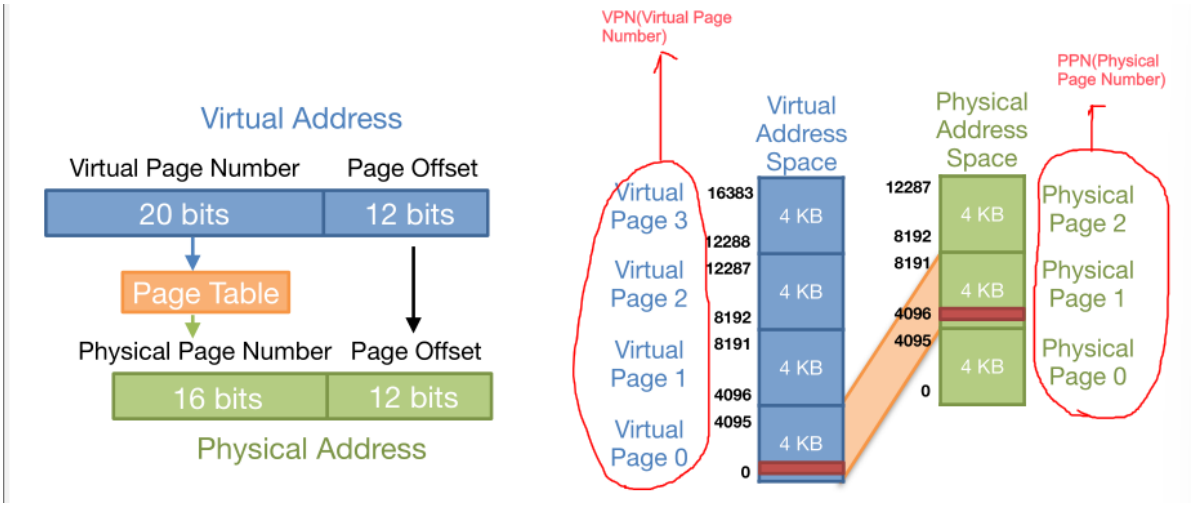
例子:

- What is the size of our virtual address and physical address on a 32 bit machine with 256 MB of RAM and 4KB pages?
  - VA size = 32 bits
  - PA size = $\log_2$(256 MB) = $\log_2(2^8 * 2^{20})$ = 28 bits

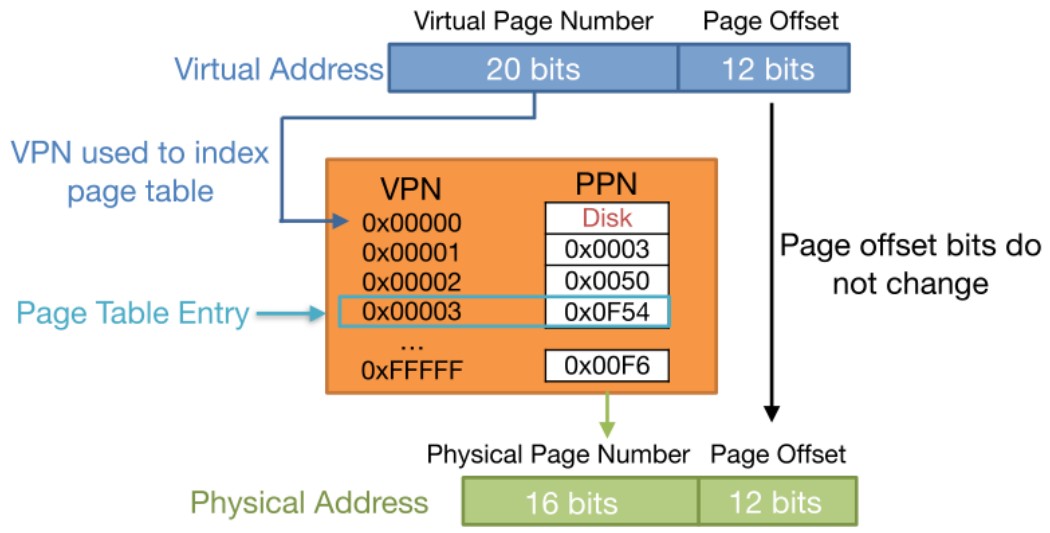



- VA size : 也就是对VM进行编址所需bit数，在多少bit的机器上就是多少bit，如图32bit
- PA size : 对PM 进行编址所需bit数，等于$log_2 RAMSize$
- Page offset: 在一页Page 内，对所有VA 的索引数编址所需bit 数，等于$log_2 PageSize$

这里page的大小是4KB，$Pageoffset = log_2 4KB = log_2(2^2 * 2^{10}) = 12bit$




以下是Fine Grain Page Table 的例子，**而且事实上Page Table中只存储PPN，VPN是索引，不需要存储**，就像是一个数组，不用考虑数组索引的存储



## 6. Page Faults

VA 和 PA 通过Page Table 进行映射，假如某VA在Page Table 中映射的结果为Disk,也就是对应的PA不在RAM中时，称为Page Faults

出现Page Faults时会发生什么?

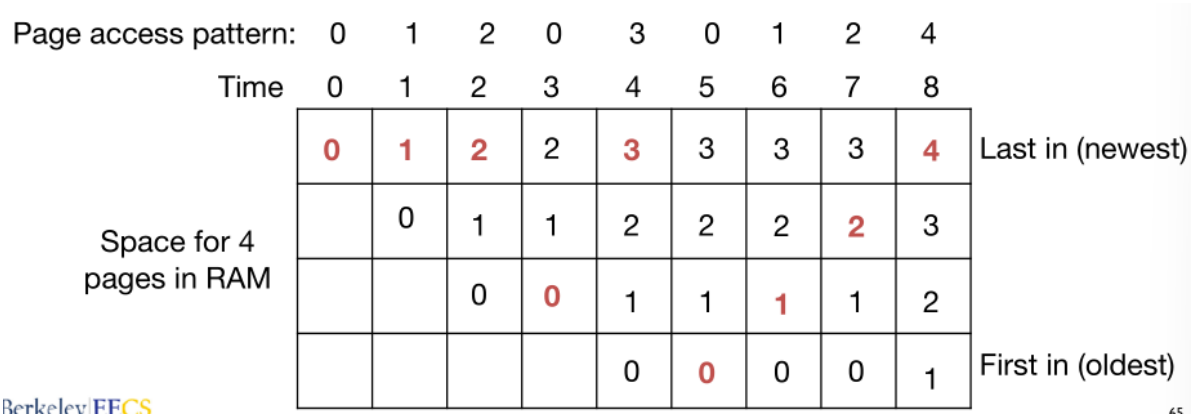
1. CPU生成page faults exception
2. 硬件将控制权交给OS ,OS 执行page fault handler 程序去修复
3. 如果RAM 没有足够的空间了，OS 会选择某一page驱逐
4. 在驱逐之前，如果该page is dirty，需要先写回disk，然后OS更新相应的Page Table Entry,指向Disk,表明已将该page放入Disk
5. OS 从Disk 中将我们想要的page 载入RAM中，并为新的Page 更新page table entry
6. OS 将控制权还给硬件，从造成page fault exception的指令处继续执行，但是由于已经被修复，这次不会再造成exception

最后想说的是，处理page faults的发生真的是一件非常非常慢的事，This is why having more RAM will make your computer faster

## 7. Page Replacement Policies

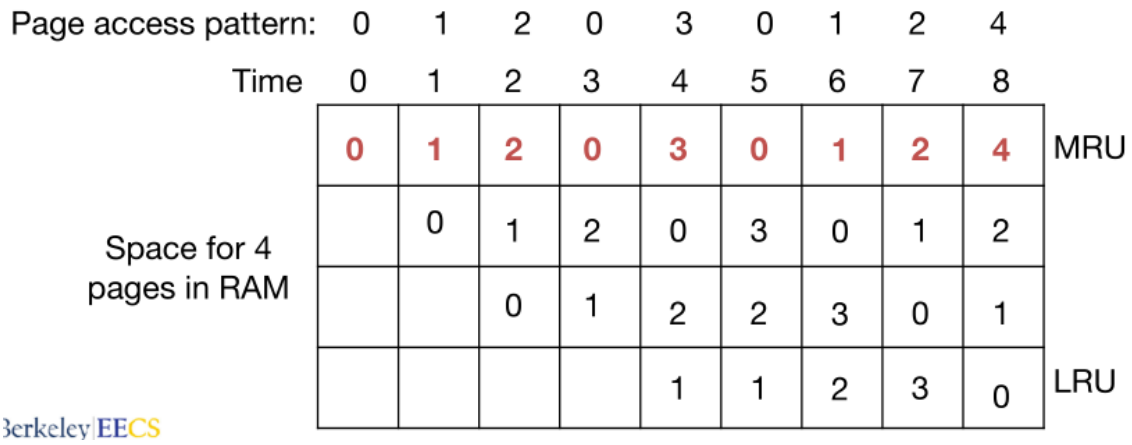
### First in, First out (FIFO)

与队列类似，驱逐最旧的(最先访问的)page,只有当所访问page 不在RAM中时才需要更新




### Least Recently Used (LRU)

驱逐最长时间未使用的Page,每次访问都需要更新Page Table



### Random

随机选择某Page 进行驱逐

### Cons

可以构造一种访问模式，使得LRU和 FIFO都失效：

如果RAM 只能同时容下4 Page,依次访问0，1，2，3，4，0，1，2，3

---

## 8. Additional Page Table Metadata

在Page table中，除了PPN 外，还有其他bit ， 如V,D,R,W,X

V: Valid ,1表示page in RAM 且 映射有效，0表示page is not in RAM

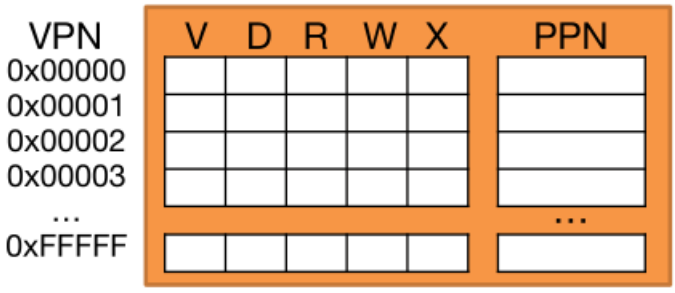
D: Dirty, 1表示page on RAM is **more up to date** than page on disk, 0表示page in RAM 和 page in Disk 相同

R: Read permission

W: Write permission

X: Execute permission

R W X 是保护位，阻止其他process访问无权访问的page，如果它们无权访问，会造成memory protection fault 或 segmentation fault



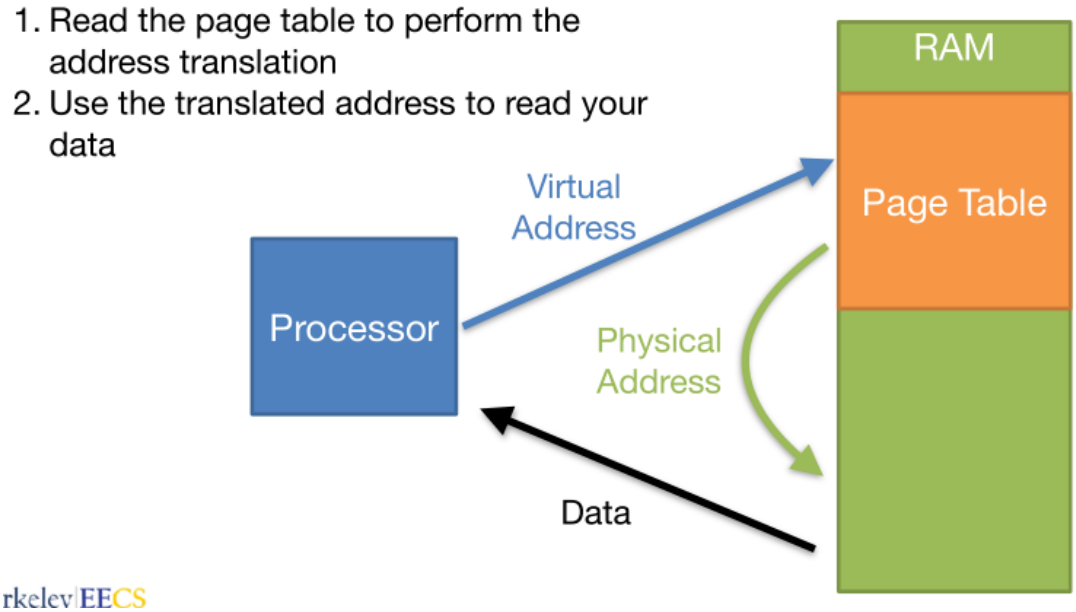

---

## 9. Translation Lookaside Buffer -- TLB

Q: 有了虚拟内存之后，当你每次想要访问某项数据，需要发生多少次内存访问?

A：2次

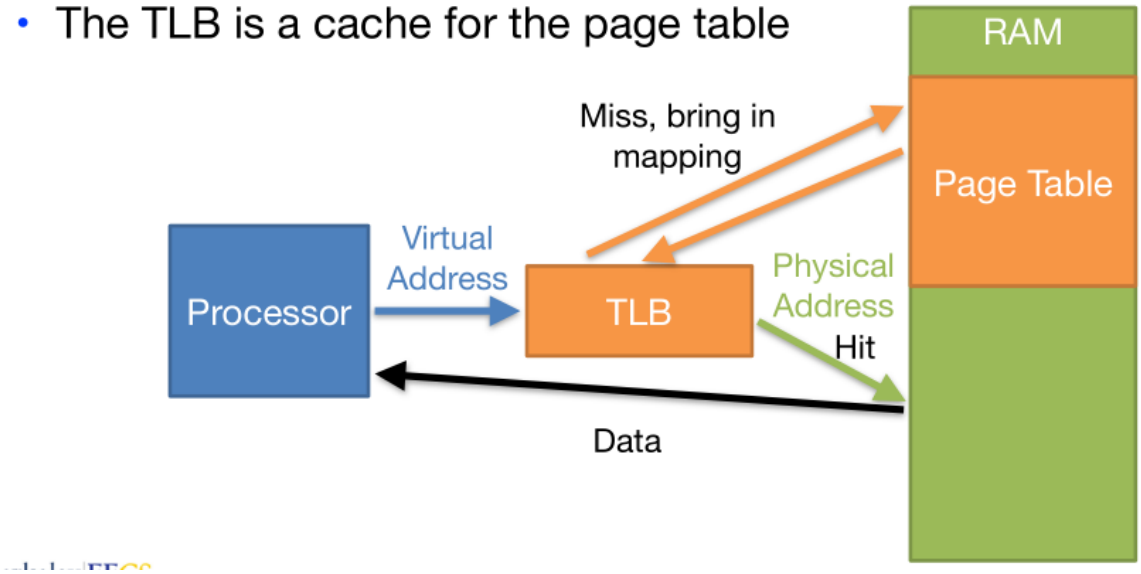
- 读Page Table ,得到translated physical address, page table 在 RAM中
- 使用physical address 在RAM中访问所需数据




rkelev|EECS

如何使其更快，也就是说就减少内存访问次数，我们知道对内存的访问是很慢的。

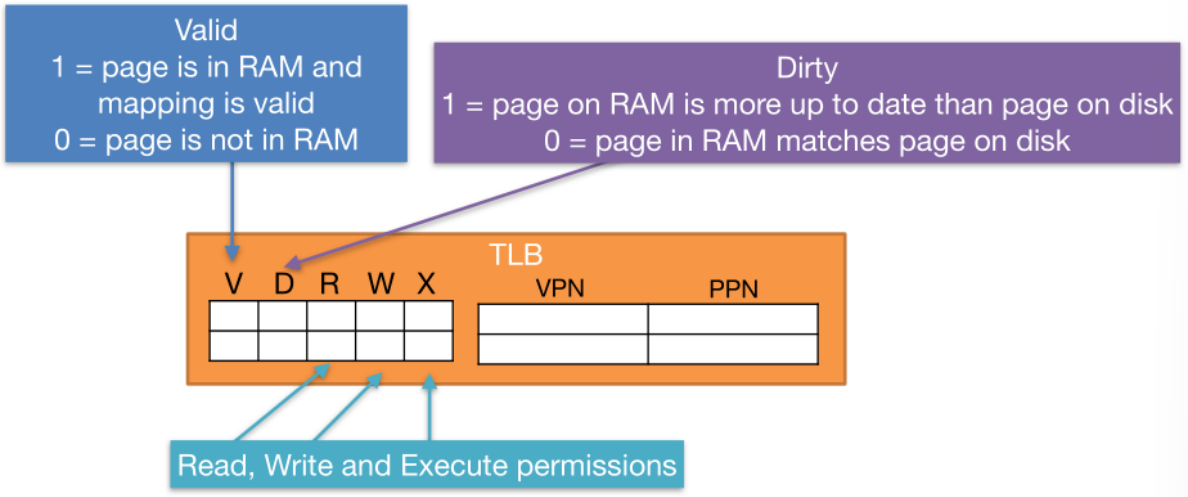
类比学过的Cache，这里对于Page table也需要一个Cache,称为Translation Lookaside Buffer(TLB)




过程如上：

- 处理器发出虚拟地址(VA)，先去TLB中找，看看有没有对应的物理地址(PA)
- 如果有，Hit,直接在RAM中访问PA，将数据返回，只存在一次Memory Access
- 如果没有，Miss, 去访问RAM中的page table，并更新TLB，执行第二步

那么同样地，TLB 也有额外的meta




TLB比Page table要更小，使用Fully Associative，一般有32 - 128 entries,但是每个entry可以映射到较大的Page，TLB的驱逐策略一般是：Random / FIFO

### TLB Flush

对于多核处理器而言，每一个Core都有自己的TLB，对于某个Core上正在活跃的Process来说，TLB与之相关，所以在发生context switch 的时候，也就是切换Process后，TLB需要被Flush,也就是之前与上一个活跃的进程有关的entry都无效了

---

## 10. Multi-level Page Tables

### why need it?

对于一台32bit 4GB RAM, 4KB pages的机器

$$虚拟内存空间大小 = 2^{32}byte$$

$$page\ table\ entry数量 = \frac{VMSpaceSize}{PageSize} = \frac{2^{32}byte}{4KB} = \frac{2^{32}byte}{2^2 * 2^{10}byte} = 2^{20} = 1M\ 条$$

$$PA = log_2 4GB = log_2 2^2 * 2^{30} bytes = 32bit$$

$$offset = log_2 PageSize = log_2 4KB = log_2(2^2 * 2^{10}) = 12bit$$

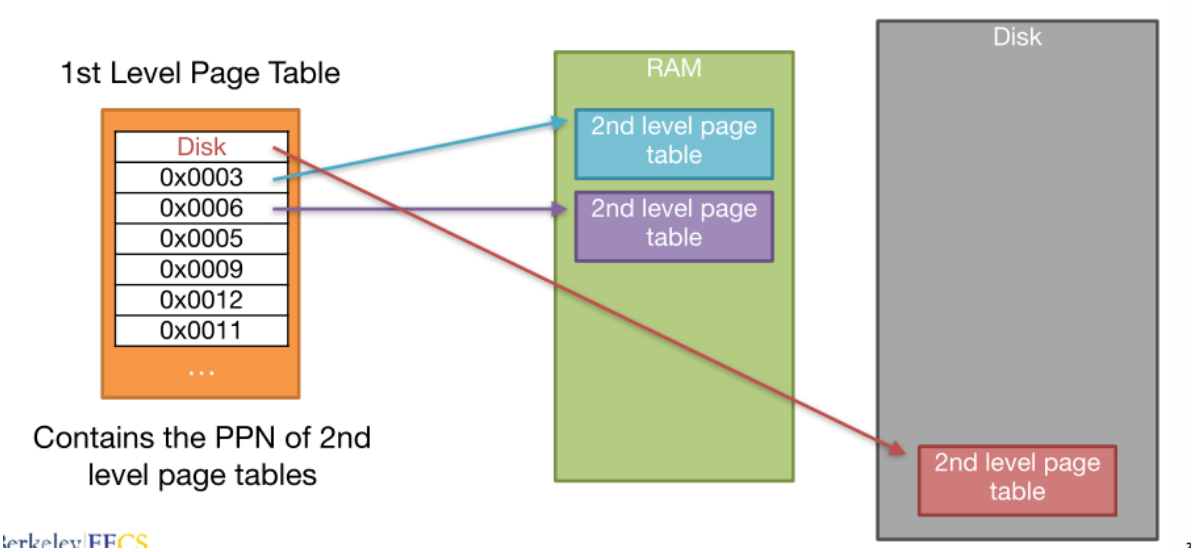
$$PPN = PA - offset = 32 - 12 = 20bit$$

$$每条PTE大约4bytes(20bits\ PPN + status\ bits)$$

$$所以Page\ Table\ Size = PTE数量 \times PTE大小 = 1M \times 4bytes = 4MB$$

看样子不算太坏，但是请注意：每一个程序都有自己的page table，如果有100个程序正在运行，那么有400MB的空间开销用于Page table

为了减小开销，引入Multi-level Page Tables




1级Page Table 存储2级Page Table的PPN，2级Page Table存储我们想要的数据的索引(PPN)

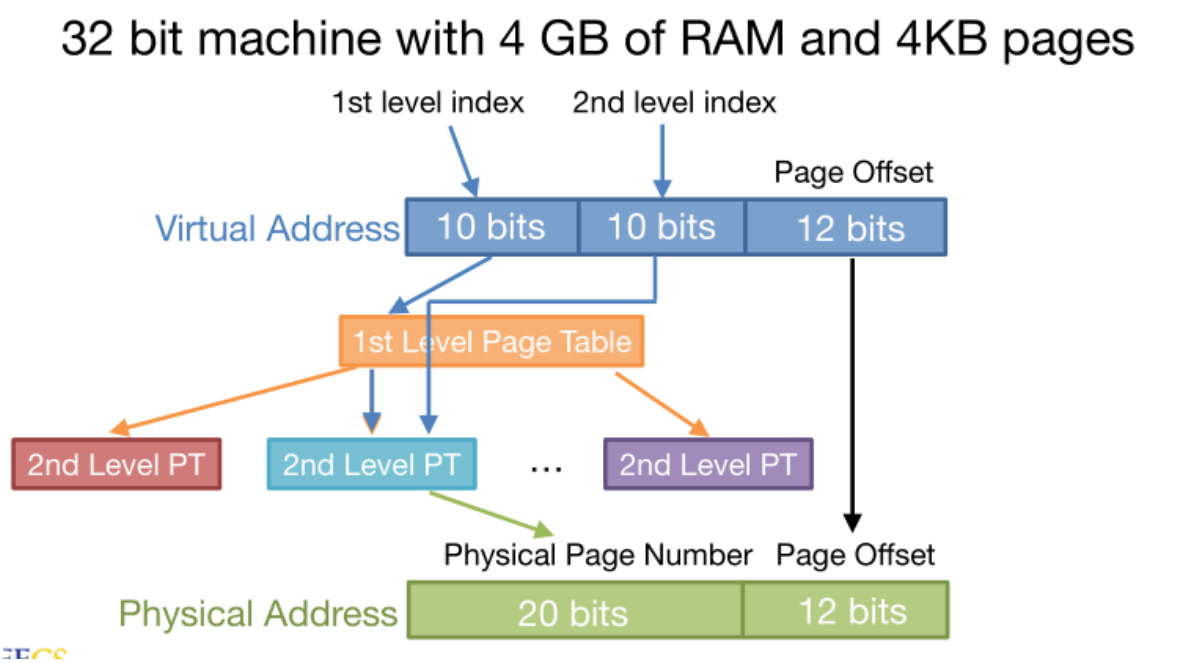
需要注意的是：

**1级page table一定在RAM中，而2级page table有的可以在RAM中，有的可以在Disk上**

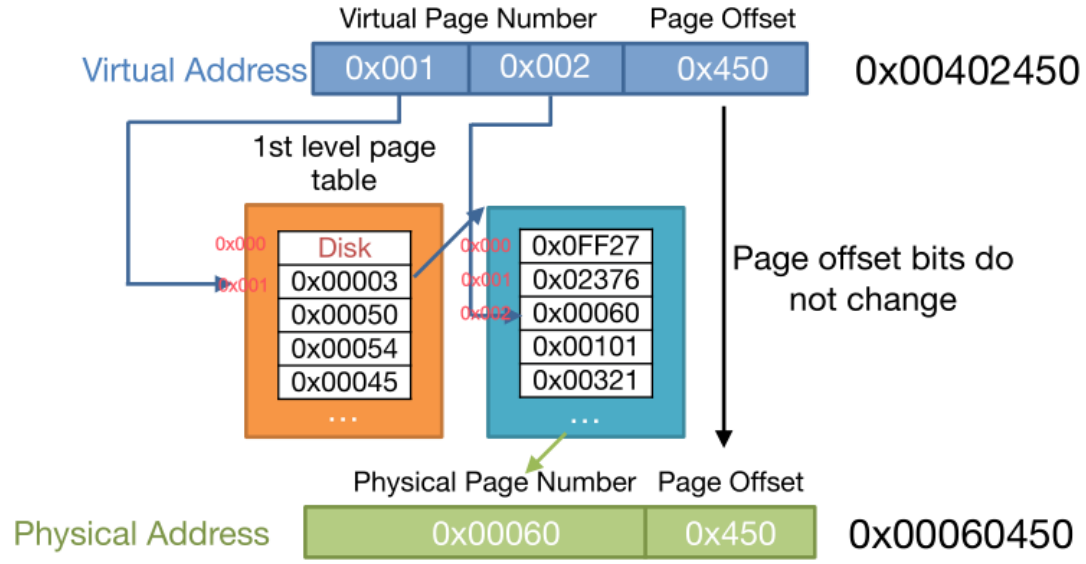
假设我们要求每页page都是4KB的大小，之前说了一条PTE大小大概是4byte(PPN + status bit)，那么一页page table 有$4KB/4byte = 1K = 1024$条 ,对于1024条PTE, 需要$log_2 1024 = 10bit$ 进行编址，same as 2级Page Table.

需要10bit 为 1st level table 的1024条PTE进行编址，同时也需要10bit 为2nd level table 的1024条PTE进行编址

也就是说1级Page Table可以映射1024个2级Page Table,每个2级Page Table可以映射1024条data,我们可以将VPN划分为2个10bit index **请注意都是index,而不是page table里面的存值**




例子：




使用多级Page Table之后，内存开销下降了很多：

只有1级Page Table时的内存开销：

- Q: If I'm running 10 applications on my 32-bit computer with 4GB RAM, 4KB pages and 1-level page tables, how much of my RAM is consumed by page tables? (size of PTE is 4 bytes)
  - VA size = 32 bits
  - PA size = $\log_2(4\ GB) = \log_2(2^2 * 2^{30}) = 32$ bits
  - # bits in page offset = $\log_2(4\ KB) = \log_2(2^2 * 2^{10}) = 12$
  - # bits in VPN = 32 - 12 = 20
  - # entries in page table = $2^{20}$
  - size of each entry is ~4 bytes
  - size of one page table = $2^{20} * 2^2 = 2^{22}$
  - total RAM consumed by pages tables = $10 * 2^{22}$ bytes = 40 MB

多级Page Table下，1级Page Table 的内存开销：

Q: If I'm running 10 applications on my 32-bit computer with 4GB RAM, 4KB pages and a 2-level page table, how much of my RAM is consumed by 1st level page tables? (size of PTE is 4 bytes)

- # bits in VPN = 32 - 12 = 20 = 10 bits for level 1 + 10 bits for level 2
- Size of 1st level page table = page size = 4KB
- total RAM consumed by 1st level pages tables = $10 * 2^{12}$ bytes = 40KB

当然，还有2级Page Table 的开销没算进去，但是有些2级Page Table是存储在Disk上的，开销一定比不分级前小

### Some Q and A

Q：在多级PageTable的情况下，当运行一个程序时，最少的page table数是多少?

A: 2 。1级Table必须在RAM中，然后至少有一个2级Table，索引我们想要访问的数据

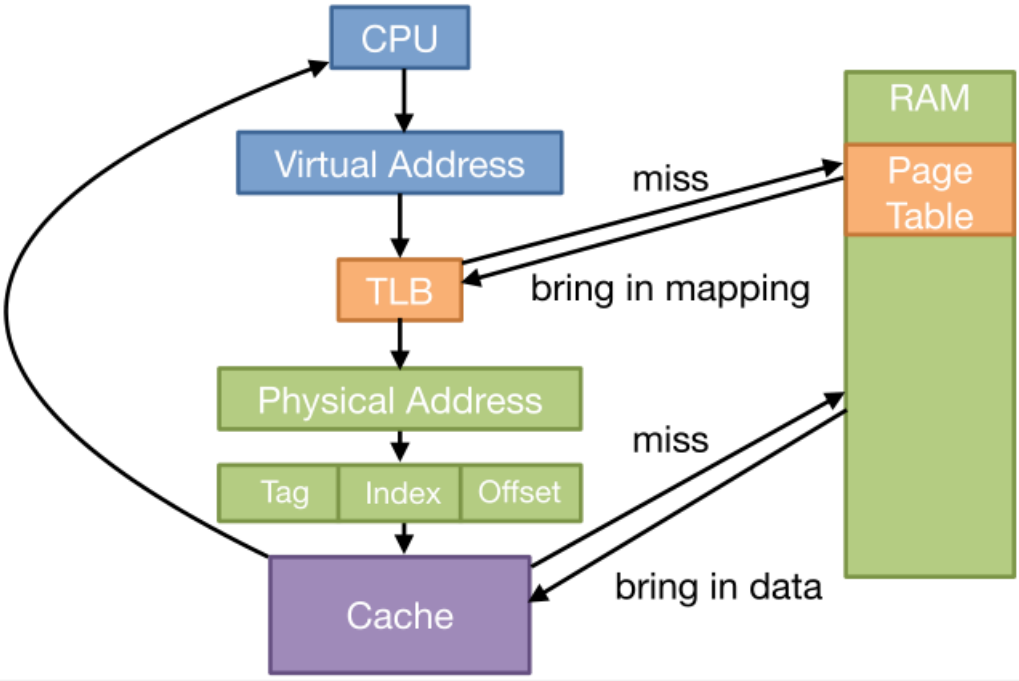
Q:多级table影响TLB吗?

A: 不影响。当TLB Hit时，直接访问RAM中的data，当TLB Miss 时，我们从原先的只需要访问1级Page Table 变成了 先访问1级page table,再访问 2级Page Table，再访问data，内存访问次数变成了 **3次**，但是减小了内存开销

---

## 11. Caches and Virtual Memory

### Physically Indexed, Physically Tagged Cache

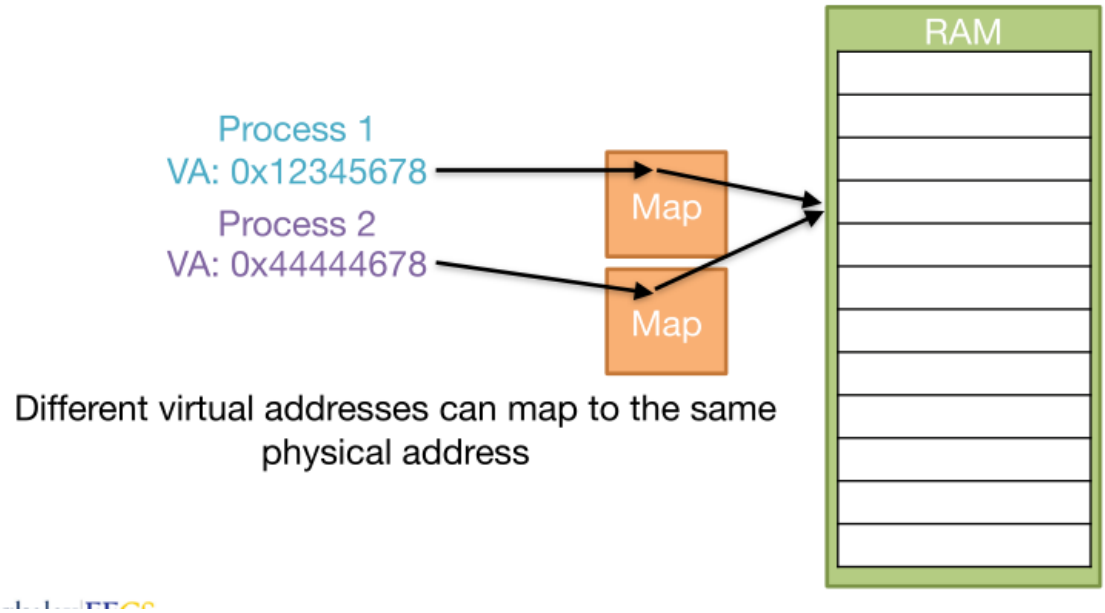



过程:

1. CPU想要访问某个VA下的data
2. 去TLB中查找
    1. 如果查找到VPN对应的PPN，Translation Hit
    2. 查找失败，Miss, 去RAM中的PageTable查找(多级PageTable),找到后更新TLB
3. 将PA拆成TIO, 去Cache中查找data
    1. Cache Hit, 返回data 至CPU
    2. Cache Miss ,访问RAM，更新Cache, 返回data至CPU

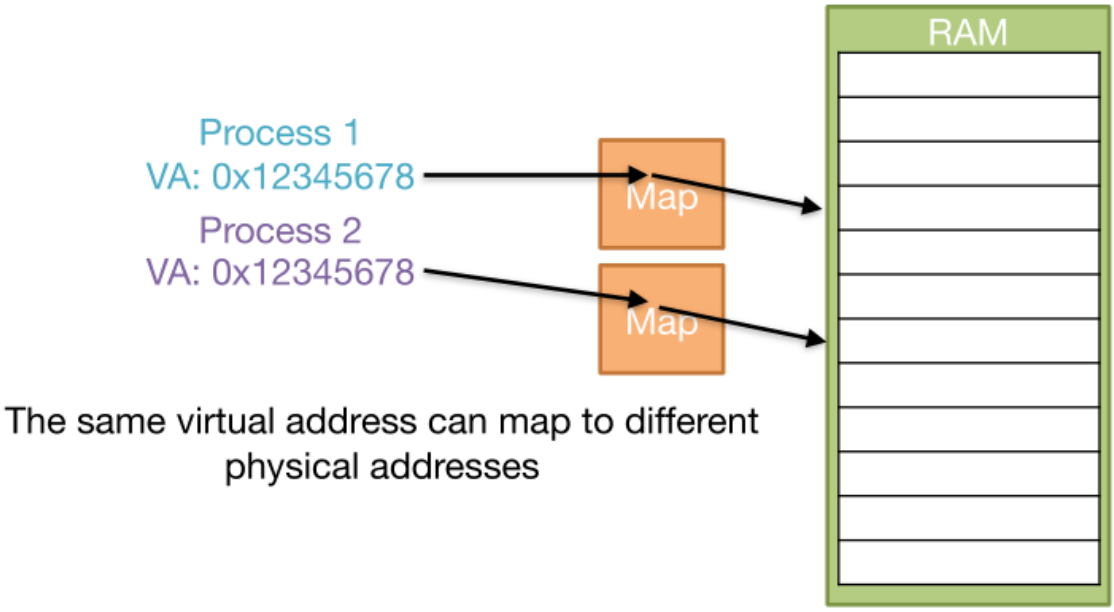
先介绍两个术语

**Synonyms**




不同的虚拟地址映射到相同的物理地址

**Homonyms**

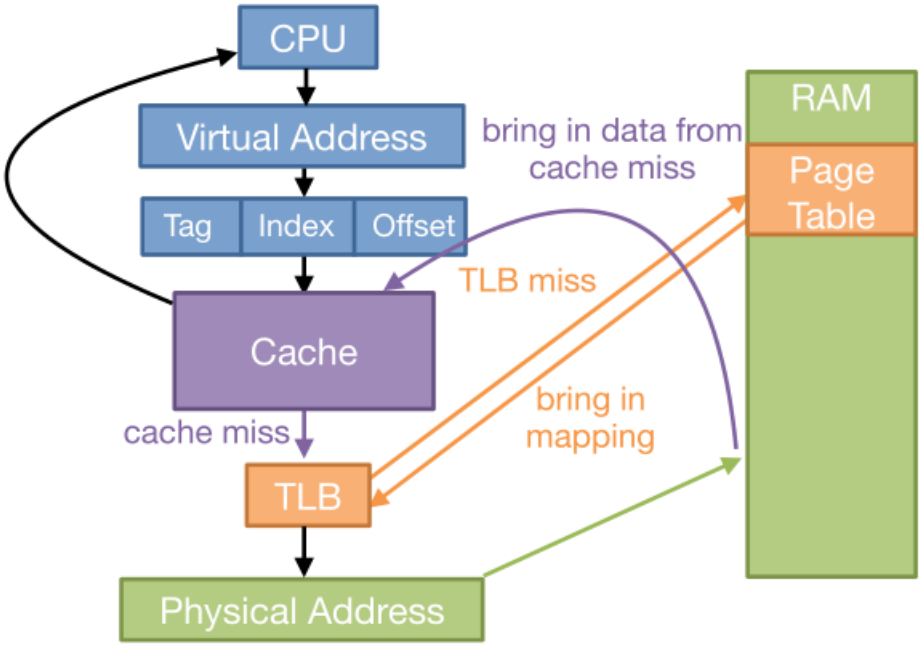



相同的虚拟地址映射到不同的物理地址

**PIPT Cache**

- 多个进程能使用相同的cache吗?
  - 可以
- 缺点
  - 在访问cache之前必须将VA translate to PA

### Virtually Indexed, Virtually Tagged Caches




过程：

1. CPU想要访问某个VA下的data
2. 直接将VA拆成TIO形式，去访问Cache
    1. Cache Hit, 返回data
    2. Cache Miss, 不得不去RAM中寻找data
3. 由于RAM是PA编址，我们不得不把VA translate to PA, 访问TLB
    1. TLB hit, 得到PA, 去RAM中加载data，更新Cache并返回至CPU
    2. TLB miss, 去访问Page Table，得到PA, 更新TLB，加载data，返回

#### VIVT Cache

- 多个进程能使用相同的cache吗?
    - 不可以，因为直接采用VA 访问Cache，如果两个Process访问相同VA, 会造成覆盖(synonyms and homonyms)
- 在每次context switch 时都需要flush Cache

#### PIPT Vs VIVT

| PIPT | VIVT |
| --- | --- |
| • Multiple processes can share the same cache<br>• Must translate the address before accessing the cache | • Multiple processes cannot share the same cache<br>  • must flush cache on context switch<br>• Do not need to translate the address before accessing the cache |

那么我们既不想在访问Cache 之前先Translate VA -->PA ,也不想受困于synonyms and homonyms

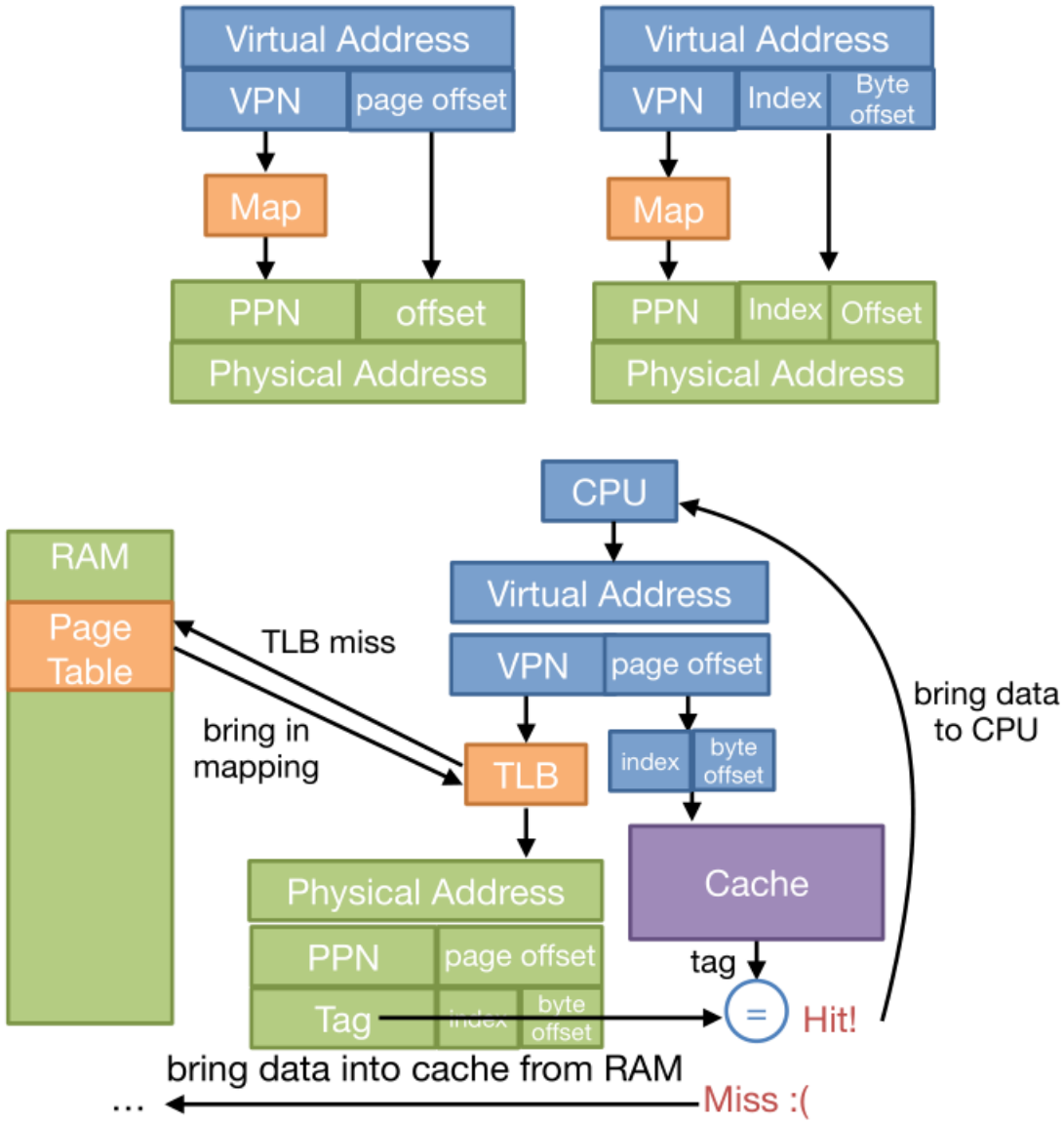
结合二者，有了VIPT Cache

## Virtually Indexed, Physically Tagged Caches

- 为避免synonyms and homonyms,使用PA 去访问Cache
- 并行执行TLB 和 Cache的浏览

Invariant

PA 与 VA 的invariant 是 page offset bit 都相同




执行过程：

1. CPU想要访问VA 处的数据，将VA拆分为VPN 和 page offset
2. 以下两件事同时发生
    1. 利用VPN去查看TLB
        1. TLB hit, 得到PA,得到PPN,也就是Tag
        2. TLB miss, 去访问Page Table,更新TLB
    2. 将page offset分为index 和 byte offset 去查看Cache
3. 将PPN 和 Cache tag对比，如果相等则Hit, 返回数据至CPU
    1. 否则Cache miss，访问RAM, 更新Cache, 返回

**Cons**

cache size 受限于page offset 的bit 数

Q : page size 4KB, 一个direct-mapped cache 能存储多少bytes?

A : 4KB, 我们只能用page offset bits 去索引 cache, $pageoffset = log_2 4KB = 12bit$

因此我们只能编址12bit, 也就是4KB 的数据

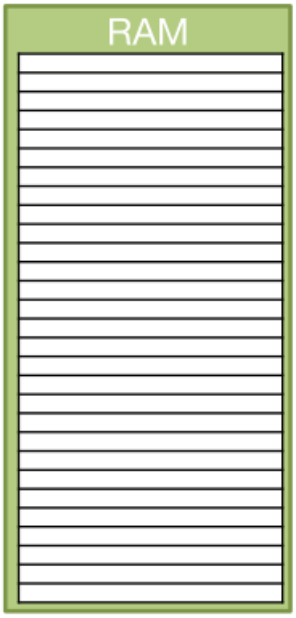
Q : 在相同page size 下， 如何让cache 更大?

A: 增加cache 的 associativity

---

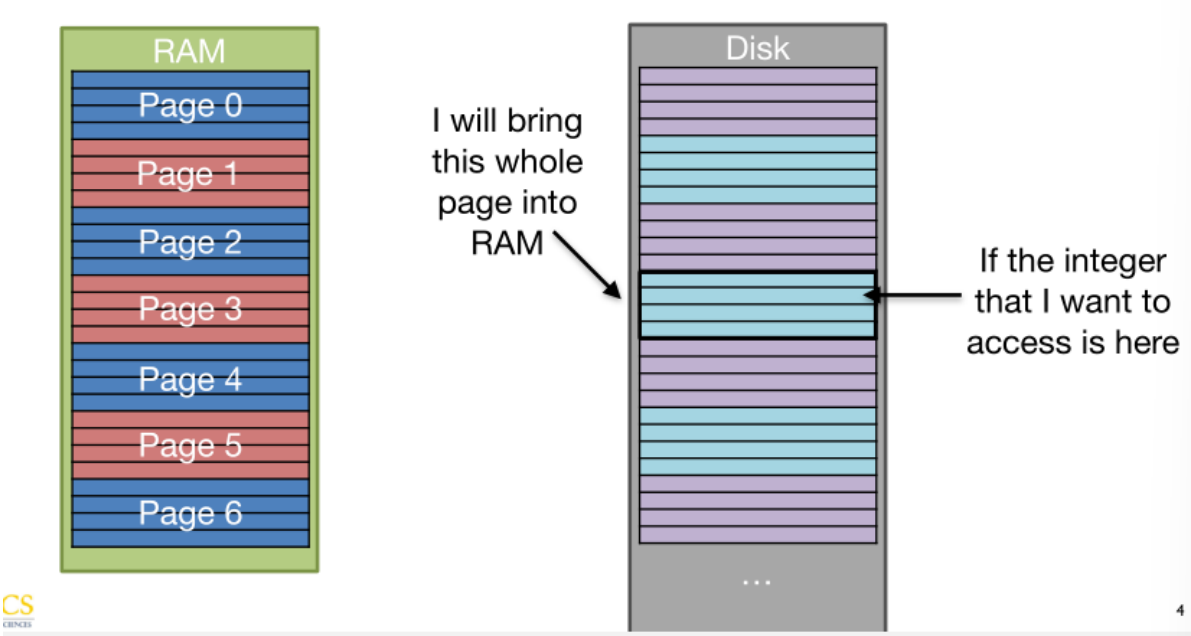
## 12. Review

- RAM = Main Memory
- RAM 包含data和code
- 为了访问data 或 code，需要将其从disk 载入 RAM




为了在Disk 和 RAM 之间传输data更加容易，我们为move data设置一个标准的granularity

- The granularity that we use is called a page
- The size of a page is typically 4KB
- 这意味着，如果我想访问某个整数，我需要将该整数所在的Page 全部载入RAM




为什么我们需要虚拟内存?

1. RAM 不足以一次性存储所有我们所需的data
2. 我们的程序认为自己的data是连续的
    - 当你写程序的时候，你并不知道它将会储存到内存的何处
    - 我们只需要认为内存是连续的，VM 会在背后为我们处理一切
3. 在进程之间起到保护作用