

Memory Hierarchy

[零 参考资料 2](#_Toc45481940)

[一 Memory Hierarchy 2](#_Toc45481941)

[1 概述 2](#_Toc45481942)

[2 Random-Access Memory(RAM) 2](#_Toc45481943)

[2.1 概述 2](#_Toc45481944)

[2.2 Static Random-Access Memory(SRAM) 3](#_Toc45481945)

[2.3 Dynamic Random-Access Memory(DRAM) 3](#_Toc45481946)

[2.4 读取指定spuercell内容 3](#_Toc45481947)

[2.5 从内存模块中读取 4](#_Toc45481948)

[2.6 访问主存 5](#_Toc45481949)

[3 Disk Storage 5](#_Toc45481950)

[3.1 概述 5](#_Toc45481951)

[3.2 访问Disk 6](#_Toc45481952)

[3.3 Solid State Disk(SSD) 7](#_Toc45481953)

[4 Cache Memories 8](#_Toc45481954)

[4.1 概述 8](#_Toc45481955)

[4.2 通用的cache memory 组织结构 9](#_Toc45481956)

[4.3 Direct-Mapped Caches 10](#_Toc45481957)

[4.4 Set Associative Cache 11](#_Toc45481958)

[4.5 Full Associative Caches 11](#_Toc45481959)

[二 虚拟内存(Virtual Memory,VM) 12](#_Toc45481960)

[1 概述 12](#_Toc45481961)

[2 VM作为缓存工具 12](#_Toc45481962)

[3 VM作为memory管理工具 14](#_Toc45481963)

[4 VM作为保护memory工具 14](#_Toc45481964)

[5 地址翻译(Address Translation) 15](#_Toc45481965)

[6 早期Intel Core i7的Address translation 18](#_Toc45481966)

[三 操作系统中的Memory 18](#_Toc45481967)

[四 操作系统中的Virtual Memory 18](#_Toc45481968)

[五 LINUX中的Memory 18](#_Toc45481969)

[六 LINUX中的Virtual Memory 18](#_Toc45481970)

[七 局部性 18](#_Toc45481971)

# 零 参考资料

《深入理解计算机系统 第三版》

《操作系统概念 第九版》

《深入理解Linux内核 第三版》

《深入Linux内核架构》

PS：整理顺序，除了《深入理解计算机系统》部分，其他两个仅整理差异部分

1. 深入理解计算机系统
2. 操作系统概念
3. 深入理解Linux内核 & 深入Linux 内核架构

# Memory Hierarchy

## 概述

* + - 1. 存储的层次结构中心思想：
         1. 对于每个层（k），位于k层的更小更快的存储设备作为位于k+1层的更大更慢的存储设备的缓存
         2. 即层次结构中的每一层都缓存来自较低一层的数据对象



## Random-Access Memory(RAM)

### 概述

* + - 1. 又两类：static（SRAM）、dynamic(DRAM)
         1. SRAM：速度快、价格贵、容量小、用于高速缓存，
         2. DRAM：相对SRAM，速度慢，价格便宜，容量大，用于主存和图形系统的frame buffer
      2. 整体比较

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | transistors/bit | 相对访问时间 | Persistent | sensitive | 相对成本 | 应用 |
| SRAM | 6 | 1 | Y | N | 1000 | cache memory |
| DRAM | 1 | 10 | N | Y | 1 | 主存，frame buffer |

### Static Random-Access Memory(SRAM)

* + - 1. 将每bit存储在一个bitstable存储器单元中
         1. 每个单元用一个six-transistor circuit实现
         2. 可以无限期地保持在两个不同的电压配置或状态之一
         3. 其他任何状态都是不稳定的，类似下图的inverted pendulum

当摆钟倾斜在最左或最右时，是stable的

原则上也能在垂直位置无期限地保持平衡，但此时状态是metastable



* + - 1. 只要有电，SRAM存储单元就会永远保持它地值
      2. 对干扰不是很敏感，即使有干扰，在干扰消除时，电路就会恢复稳定值

### Dynamic Random-Access Memory(DRAM)

* + - 1. 用电容器上的电荷存储每bit,每个电容只有大约30 femtofarads——30X10-15farad
      2. 对干扰很敏感，当电容电压受到干扰，就会永远丢失数据：
         1. 比如暴露在光线下
         2. 数码相机和摄像机中的传感器本质上就是DRAM单元阵列
      3. 很多原因导致DRAM单元在10-100毫秒时间内失去电荷：
         1. 计算机运行时钟周期是纳秒级，相对保持时间较长
         2. 计算机系统必须周期性通过读&写刷新每bit
         3. 有些系统还是用error-correcting codes，可以用来发现并纠正erroneous bit
      4. 一般DRAM chip：
         1. cell(bit)被分组成d个supercell
         2. 每个supercell 由w个cell(bit)组成
         3. 所有supercell被组成r行c列的矩形阵列
         4. 每个supercell 都有一个(i,j)地址，i表示行，j表示列

### 读取指定spuercell内容

* + - 1. 每个DRAM芯片连接到Memory controller，一次可以跟每个DRAM chip交互w bit数据
      2. 信息通过pin进出DRAM chip，每个pin带1bit信号
      3. 实际读取supercell流程，以从下图chip中读取supercell(2,1)为例：
         1. Memory controller 发送RAS(Row Access Strobe)=2到chip
         2. chip复制行2所有数据复制到Internal row buffer
         3. Memory controller 发送CAS(Column Access Strobe)=1到chip
         4. chip从internal row buffer复制出指定supercell中所有bit并发送给Memory controller
      4. DRAM之所以是二维阵列，而不是线性数组，是为了降低chip上的address pin数量，但因为必须分两次发送地址，增加了访问时间



### 从内存模块中读取

* + - 1. DRAM chip封装在memory modules中，插到main system board上的expansion slot中
      2. 下图展示了，内存模块的基本思路：
         1. 模块用了由8个 8\*8M的DRAM组成的64MB Memory module，8个DRAM编号0-7
         2. 每个supercell存储main memory的1byte
         3. main memory 中地址A数据由，用依序读取DRAM中supercelle（i，j）的信息组成(64 bit word)
      3. 读取地址A过程示例：
         1. 请求获取地址A数据
         2. Memory controller 将地址A转换成supercell(i,j)
         3. 将supercell(i,j)发送给memory module
         4. memory module广播supercell到每个DRAM
         5. 每个DRAM输出对应的supercell内容
         6. 合并成各个DRAM输出的内容，返回给Memory controller
         7. Memory controller返回地址A数据
      4. 通过将多个memory module 连接到memory controller 聚合成main memory，这时，当controller收到地址A时，只会将A转成supercell(i,j)发送给包含A的memory module



### 访问主存

* + - 1. 数据流通过Bus在processor和DRAM main memory之间交互
      2. 执行 movq A,%rax：
         1. CPU通过 bus insterface 发起 read transcation
         2. CPU将地址A放到System bus上，
         3. 通过I/O bridge将信号从System bus转换并发送到Memory bus
         4. Main memory 扫描到memory bus上的地址A
         5. 根据地址A，从DRAM读取数据并写到memory bus
         6. 通过I/O bridge将信号从Memory bus 转换并发送到System bus
         7. CPU从System bus获取地址A数据，并复制到register中



## Disk Storage

### 概述

* + - 1. disk 结构
         1. disk由一个或多个platter叠放在一起组成：

每个platter 有两个覆盖着magnetic recording material的surface

platter中间有个使platter以固定rotational rate旋转的spindle

* + - * 1. 一个surface由一组track组成，

每个track被划分为一组sector，

每个sector中包含相同数量的编码在magnetic recording material上的data bits

sector之间由gap隔开，这些gap不存储数据，存储的是identify sector的formatting bits

所有surface上的跟spindle距离相同的track集合称为cylinder

在multiple zone recording技术中，cylinder被分成不相交的子集，称为recording zone

recording zone 中每个cylinder的每个track都有相同数量的sector，sector数量由该zone最里面track所能包含的sector数量决定

* + - 1. disk 容量计算
         1. recording desity(bits/in):track上1英寸的segment中可以放入的bits数量
         2. track desity(tracks/in)：从platter 中心开始，半径上1英寸的segment上track数量
         3. areal density(bit/in2)：recording desity 乘以 track desity
         4. 容量公式：

 

### 访问Disk

* + - 1. disk用read/write head 读写存储在magnetic surface的bits：
         1. read/write head连接到arm末端
         2. 通过半径轴前后移动arm，可以将read/write head定位到任何track上，即seek
         3. 当read/write head定位到指定track上，可以读取/修改通过head下面的每bit
         4. 当有多个platter时：

每个surface都有独立read/write head

所有read/write head垂直排列

一致行动，任何时刻都在同一个cylinder上

* + - 1. 当read/write head遇到异物时，会停下来
      2. Seek Time:
         1. 为读取目标sector的内容，arm先将read/write head定位到sector所在的track
         2. 依赖于read/write head之前的位置和arm在surface上移动的速度
      3. Rotational latency：
         1. 当read/write head定位到目标track，等待目标sector的第一个bit旋转到read/write head下，所需时间
         2. 依赖于当read/write head到达指定track时所在surface位置以及rotational speed
         3. 最坏的情况，read/write head正好错过了目标sector，必须等待disk转一圈
         4. 最大rotational latency：
         5. 平均rotational:
      4. Transfer time:
         1. 当read/write head到达目标 sector的第一个bit时，读写目标sector内容所需时间
         2. 依赖于rotational speed和每条track上sector数量
         3. 粗略估算平均时间：
      5. 访问数据所需时间估算：
      6. 读取数据流程：
         1. 假设设备支持DMA(Direct Memory Access)
         2. 流程图



### Solid State Disk(SSD)

* + - 1. SSD是一种基于 flash memory的存储技术，由一个或多个flash memory chips和flash translation layer组成
         1. 一个flash memory由 B个块依序组成

一个块由P页组成

一个页大小512 kytes到4KB

块一般由32-128页组成

* + - * 1. flash translation layer 是一个硬件/固件设备，将对logical blocks的请求翻译成对底层物理设备的访问
      1. 数据是以页为单位读写的
         1. 只有一页所属的整个block被擦除后才能写这页
         2. 一旦一个block被擦除了，block中的页可以不再擦除就可以写。
      2. 随机写速度比读慢：
         1. 擦除block需要时间相对较长，比读所需时间高一个数量级
         2. 如果写操作试图修改一个已有数据的页p，必须将对应的block中有数据的页都复制到新的可写的block后才能对页p修改
      3. 基本思想示意图



## Cache Memories

### 概述

* + - 1. 一般来说，cache是一个小而快速的存储设备，作为层次更大、更慢的设备中的数据对象的staging area，使用cache的过程称为caching
      2. cache的一般性概念
         1. 第k+1层的存储器被划分成连续的数据对象chunk，称为block

每个block都有唯一的地址或地址

block可以是固定大小，也可以是可变大小

* + - * 1. 第k层的存储器

跟k+1层相似，被划分成较少的较小的block集合，每个block跟k+1层的大小一样

任何时候k层的cache包含k+1层blocks的一个子集副本

* + - * 1. 数据以block为transfer unit在k层和k+1层之间来回复制

任何相邻的层次之间block大小是固定的，其他层次对之间可以由不同的block大小

一般而言，层次结构中远离CPU的设备，倾向于使用较大的block，来补偿较长的访问时间



* + - 1. cache确定请求是否命中，然后抽取请求的数据的过程有三步：1 set selection，2 line matching，3 word extraction
      2. 缓存命中：当程序需要k+1层某个数据对象d时，它首先在k层中的blocks中查找d，如果找到了，就是 cache hit
      3. 缓存不命中:如果k层没有数据对象d，即为cache miss
         1. 当发生cache miss时，k层的cache从k+1层取出包含d的block，如果k层已经满了，可能就会覆盖现有的block
         2. 覆盖现存的block的过程称为replacing或evicting 这个block。
         3. 被evicted的block有时候被称为victim block
         4. 决定替换哪个块是由cache的replacement policy决定
         5. 缓存不命中种类：

强制性不命中或冷不命中:一个空的缓存有时被称为cold cache，这时缓存不命中称为compulsory miss或cold miss

冲突不命中：即使k层cache空闲空间足够，当因为特定的replacement policy，k+1层的多个block会映射到k层同一个cache block中，导致不命中，称为conflict miss

容量不命中：当working set的大小超过cache大小时，就会发生capacity miss

### 通用的cache memory 组织结构

* + - 1. 每个存储器地址有m位，形成M=2m个不同地址
      2. 高速缓存被组织成一个有S=2s个cache set的数组：
         1. 每个set包含E行cache line
         2. 每行由：

1个表明此行是否包含有效信息的valid bit

t=m-(b+s)个tag bit作为存储在当前cache line的block的唯一标识

B=2b bytes的data block 组成

* + - 1. Cache size= B\*E\*S
      2. main memory 中的地址A：
         1. 参数S和B将地址A划分为三个字段：

t bit tag bit：定位数据在指定的set中的哪一行，但仅当指定行的valid bit值为有效时，数据才可能存在于指定行

s bit set index：指向S个set的数组下标，是一个无符号整数，表明当前地址内容必须存储在哪个set中

b bit block offset：当成功定位到某一行，此值给出了数据在B个byte的data block中的offset



* + - 1. set index 之所以在地址中间，而不是高位，是为了将存连续地址块内容到不同缓存行

### Direct-Mapped Caches

* + - 1. 每个set只有一行(E=1)的cache
      2. 获取缓存



* + - 1. cache miss 替换策略：用新行直接替换当前行

### Set Associative Cache

* + - 1. 每set都保存多于1个的cache line, 即 1< E < C/B
      2. 替换策略有：随机替换、Least-Frequently-Used(LFU)、Least-Recently-Used(LRU)



### Full Associative Caches

* + - 1. 只有一个Set组，即E = C/B
      2. 这里地址被划分成 tag和 block Offset 两部分



# 虚拟内存(Virtual Memory,VM)

## 概述

* + - 1. 提供了三个能力：
         1. 将main memory作为存储在disk上的地址空间(address space)的cache，只将active areas保存在main memory中，并根据需要在disk和main memory之间来回传送数据，通过这种方式，高效地使用了main memory
         2. 为每个process提供了统一的地址空间，简化了memory管理
         3. 保护了每个process的地址空间不被其他process破坏
      2. Physical Address(PA):main momery 被组织成一个由M个连续的1 byte-size的cell的数组(从0开始)，每个byte 都有一个唯一的PA
      3. Physical Addressing: CPU使用 PA访问main memory的方式，称为物理寻址(Physical Addressing)
      4. Virtual Addressing:CPU通过生成一个Virtual Address(VA)访问main momery
         1. VA在被送到main momery之前会先通过Address translation task 翻译成适当 PA
         2. CPU 上有个Memory Management Unit（MMU）专门硬件，利用存放在main memory中的lookup table动态翻译VA
      5. Address Space：一个非负整数地址的有序集合
         1. 如果地址空间中的整数是连续的，则称为 linear address space
         2. 这里假定使用的是linear address space
      6. Virtual Address Space：在一个有个VM的系统中，CPU从一个有N=2n个地址的address space生成Virtual Address,称为虚拟地址空间(Virtual Address Space)
      7. 一个地址空间的大小是由表示最大地址所需的位数来描述的
      8. 物理地址空间(Physical Address Space)
      9. VM基本思想：
         1. 每个数据对象有多个独立的地址，每个地址都选自不同的地址空间
         2. main memory 的每byte都有一个VA和一个PA



## VM作为缓存工具

* + - 1. VM系统为解决数据传输问题
         1. 将VM分割为大小固定为P=2pbyte的Virtual Page(VP)，通常大小为4KB-2MB
         2. 将physical memory分割为P byte的Physical Page(PP)，PP也称为page frame
      2. 任意时刻，VP的集合都分配三个不相交的子集：
         1. Unallocated：未分配或未创建的page，不占用任何空间
         2. Cached： 缓存在physical memory中的已分配页
         3. Unched： 未缓存在physical memory中的已分配页
      3. Page Table:提供VP映射PP的关系的数据结构
         1. 常驻在 physical memory中
         2. 每次MMU中的address translation hardware将一个VA转成PA时，都会读取page table
         3. OS负责维护page table内容以及disk和DRAM之间传送数据页
      4. Page Table基本组织结构
         1. 一个Page Table Entry(PTE)的数组
         2. Virtual Address Space中每个page在page Table中一个固定offset处都有个PTE
         3. 假设每个PTE是由一个valid bit和一个n bit地址字段组成

valid bit=1，表明地址字段就表示DRAM中相应PP的起始位置，这个PP缓存了该VP

valid bit=0，则地址字段为空，表明VP未被分配

* + - 1. Page Hits如下图的访问VA1请求
      2. Page Fault：VM习惯说法中，缓存不命中称为page fault,下图VA2请求演示了page fault
         1. page fault exception调用内核中缺页异常处理程序，选择一个victim page
         2. 如果victim page对应VP已被修改，那么内核会将它复制回disk，并修改对应的PTE，表示VP不再缓存在main memory
         3. 接着内核从dis复制目标VP到PM中，并更新对应的PTE，然后返回，并重启导致page fault的指令
         4. 这时VA2请求就可以page hits了，如下面第二张图
      3. Allocating page，如下面第二张图的PTE5：
         1. 在disk上创建空间，
         2. 更新对应的PTE，指向disk上新建的page





## VM作为memory管理工具

* + - 1. 实际上，OS为每个process提供独立的page table和virtual address space
      2. Memory mapping(mmap):将一组连续的VP映射到任意文件中的任意位置，称为内存映射(memory mapping),Linux中，可以这种自定义mmap
      3. Simplifying Linking：独立的地址空间允许每个process的memory image使用相同的basic format，而不用官代码和数据实际存放在为physical memory何处
      4. Simplifying loading: VM还使想memory加载可执行文件和共享对象文件变得容易
      5. Simplifying sharing: 独立的地址空间为OS提供了一致的user process和OS自身之间共享的机制
      6. Simplifying memory allocation:VM为user process提供了简单的分配additional memory的机制



## VM作为保护memory工具

* + - 1. 因为每次CPU生成一个地址时，MMU中的address translation hardware都会读取一个PTE，所以通过在PTE上添加一些额外permission bits控制VP内容访问十分简单
      2. 如果一条指令违反了permission，CPU就触发一个 general protection fault，将控制传递给一个内核中的异常处理程序



## 地址翻译(Address Translation)

* + - 1. Address Translation: N element的Virtual address space(VAS)中的elements和M element的physical address space(PAS)中的elements之间的映射，称为地址翻译(address translation)
         1. 即 MAP：VAS -> PAS U
      2. MMU利用page table实现map：
         1. 页表基址寄存器(Page Table Base Register)指向当前page table
         2. n bit的VA包含两部分：p bit的虚拟页面偏移（Virtual Page Offset）和n-p bit 的虚拟页号(Virtual page number)
         3. MMU利用VPN定位PTE,
         4. 将PTE中物理页号(Physical page number)和虚拟地址中的VPO结合起来定位到物理地址(physical address)



* + - 1. CPU执行步骤
         1. page hits



* + - * 1. page fault



* + - 1. 结合Cache 和VM
         1. 主要思路是：address translation发生在cache查找之前



* + - 1. 利用Translation Lookaside Buffer(TLB)加速address translation
         1. 有些系统会在MMU中加一个小cache来缓存PTE，称为翻译后被缓冲器(Translation Lookaside Buffer)
         2. TLB中每行存储着单个PTE的block，通常有高度相联度
         3. TLB hit流程，下图步骤1>2>3>6>7
         4. TLB miss流程，下图步骤1>2>3>4>5>6>7



* + - 1. 多级page table
         1. VA 被划分成k个VPN和1个VPO
         2. 每个VPN I 都一个到第i级page table的index，其中
         3. 第j()级page table中每个PTE都指向第j+1级的某个page table的基址
         4. 第k级page table中的每个PTE包含某个physical page的PPN或者disk block的地址
         5. 为了构造PA，在能够确定PPN前，MMU必须访问k个PTE，这里可以通过TLB优化address translation速度



## 早期Intel Core i7的Address translation

# 操作系统概念中的Memory

# 操作系统概念中的Virtual Memory

# LINUX中的Memory

# LINUX中的Virtual Memory

# 局部性