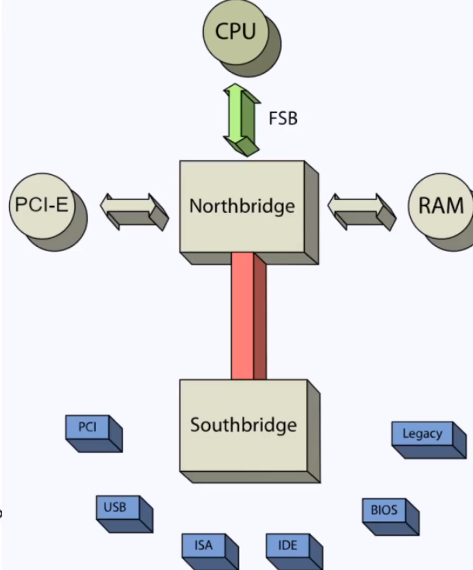
# 内存基础

## 概述

### 内存在计算机中的位置

内存是计算机中重要的组成部分，它通过内存总线与其他设备项链，内存本身是整个存储系统的一部分，访问速度比磁盘快几个数量级，可以视为磁盘到CPU之间的缓存盘。





**存储器的层次结构：**



注：在寄存器之上还有运算速度更快的ALU计算单元。

**CPU到内存计算时间：**



**多核CPU缓存：**



注：单核独享L1、L2，多核共享L3，多个槽共享内存。

**超线程：**一个ALU对应多个PC或Register，所谓的四核八线程（比如一个核有一个ALU，但是有两组Register+PC，这样ALU就可以分时计算两个组的数据，这比存储上下文然后恢复现场要快）。

缓存行（cache line，64字节）：

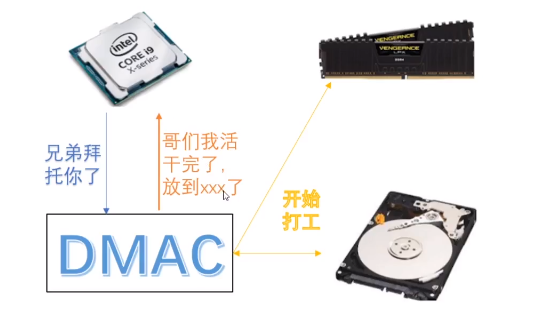


MESI cache一致性协议：



### DMAC

DMAC即DMA（直接内存访问）控制器，位于主板南桥。



DMAC接收CPU的指令（此时DMAC拥有总线的控制权限），将磁盘的数据拷贝到内存指定地址，然后通知CPU。这样可以节省CPU宝贵的计算资源。

### 作用

1. 暂时存放cpu的运算数据
2. 硬盘等外部存储器交换的数据
3. 保障cpu计算的稳定性和高性能



### 指令集架构的特权级

现代指令集架构一般都包含多个特权等级，每个等级下可以执行的指令集和范围不同。以IntelX86为例，Ring3为最低特权级，只能执行受限制令，Ring0为最高特权级，可以执行所有指令。



说明：图中的Ring1和Ring2是Intel预期保留给设备驱动程序的，但在Intel系统实现时，由于只使用Ring0和Ring3，所以设备驱动程序实际在Ring0执行。

### 用户态和内核态

操作系统可以基于特权控制应用程序对各种资源包括内存的访问。对IntelX86，Linux只使用Ring0和Ring3两个特权级，Ring0对应内核态，Ring3对应用户态。

用户态：Ring3运行于用户态的代码则要受到处理器的诸多

内核态：Ring0在处理器的存储保护中，核心态

用户态切换到内核态的 3 种方式：系统调用、异常、外设中断

区别：每个进程都有完全属于自己的，独立的，不被干扰的内存空间；用户态的程序就不能随意操作内核地址空间，具有一定的安全保护作用；内核态线程共享内核地址空间。

用户态一般不能直接访问系统的资源，也不能直接访问操作系统内核的代码与数据，而需要通过如系统调用的方式，切换为内核态之后，由操作系统去访问具体的资源或实现某些功能。很多系统调用常由库函数包装后提供给应用程序使用。

## 内存对齐

参考：<https://zhuanlan.zhihu.com/p/30007037>

### 概念

### 原理

### 原则

## 页与巨页

### 页

**内核把物理页作为内存管理的基本单位（文件系统以块作为基本单元，磁盘则是以扇区作为基本单元）**。尽管处理器的最小可寻址单位通常为字（甚至字节），但是，内存管理单元（MMU，管理内存并把虚拟地址转换为物理地址的硬件）通常以页为单位进行处理。正因为如此，MMU以页（page）大小为单位来管理系统中的页表（这也是页表名的来由）。**从虚拟内存的角度来看，页就是最小单位**。

体系结构不同，支持的页大小也不尽相同，还有些体系结构甚至支持几种不同的页大小。大多数32位体系结构支持4KB的页，而64位体系结构一般会支持8KB的页。

在使用虚拟地址空间时，虚拟地址不等于物理地址，需要两者建立映射关系，而这种映射关系也需要被保存在内存中以供MMU查询。显然，按字节来建立映射关系，整个内存都不够用。因此按照一定大小的内存块管理内存，并以块为单位来建立映射关系，就成为一种合理的选择。硬件系统本身也支持这种管理方式。

页：一定大小的内存块。操作系统以页为单位来管理虚拟地址空间和物理内存，在Linux上一页一般为4K大小。

### 页表

参考：

<https://blog.csdn.net/u011209099/article/details/9248525>

虚拟页与物理页可以通过两个页的首地址的映射关系来关联，这种映射关系的记录保存在所谓的页表中，因此被称为页表项。页表使用树状的结构来保存所有的页表项，以方便页表项的快速查找。页表的详细组织结构以及页表项的查找可以参考操作系统相关书籍。

每个进程都有自己的页表来记录自己的虚拟页的映射关系，页表并不需要记录该进程整个虚拟地址空间的页映射情况，只需要记录已分配的部分即可，这样可以节省大量内存空间。

### 区

由于硬件的限制，内核并不能对所有的页一视同仁。有些页位于内存中特定的物理地址上，所以不能将其用于一些特定的任务。由于存在这种限制，所以内核把页划分为不同的区（zone）。内核使用区对具有相似特性的页进行分组。Linux必须处理如下两种由于硬件存在缺陷而引起的内存存寻址问题：

1. 一些硬件只能用某些特定的内存地址来执行DMA（直接内存访问）；
2. 一些体系结构的内存的物理寻址范围比虚拟寻址范围大得多，这样一些内存不能永久地映射到内核空间上。

由于存在这些制约条件，Linux主要使用了四种区：

ZONE\_DMA：这个区包含的页能用来执行DMA操作。

ZONE\_DMA32：和ZONE\_DMA类似，该区包含的页面可用来执行DMA操作，而和ZONE\_DMA不同之处在于，这些页面只能被32位设备访问。在某些体系结构中，该区将比ZONE\_DMA更大。

ZONE\_NORMAL：这个区包含的都是能够正常映射的页。

ZONE\_HIGHEM：这个区包含“高端内存”，其中的页并不能永久地映射到内核地址空间。

区的实际使用和分布是与体系结构相关的。

Linux把系统的页划分为区，形成不同的内存池，这样就可以根据用途进行分配了。例如，ZONE\_DMA内存池让内核有能力为DMA分配所需的内存。如果需要这样的内存，那么，内核就可以从ZONE\_DMA中按照请求的数目取出页。注意，区的划分没有任何物理意义，这只不过是内核为了管理页而采取的一种逻辑上的分组。

### TLB（Translation Lookaside Buffer）

TLB/快表：旁路转换缓存，是集成在CPU芯片上的一个内存管理单元，用于**缓存页表项**。

在引入TLB之前，访问一个内存单元需要两步。首先要从内存中的页表查询页表项，然后根据页表项再访问与虚拟地址对应的物理地址。这样需要访问内存两次，效率比较低。在引入TLB之后，有了页表项缓存，可以提高内存访问效率。



问题：MMU本身是做地址翻译的，那么MMU是怎么知道内存中的页表在哪里呢？

回答：这个地方需要操作系统与硬件相配合。操作系统可以通过指令告知MMU页表的实际物理地址，MMU直接通过物理地址访问页表。

备注：如果MMU在TLB中查到映射关系，还是需要通过内存中的页表去查询的。

### 巨页

引入TLB后，内存访问效率有了提升。但现代的操作系统上运行的进程很多，所有进程的页表项数目非常可观，而TLB本身的空间有限，只能缓存所有页表项中较少的一部分，缓存命中率不高。**为了提升TLB命中率，Linux操作系统采用了巨页技术**，巨页大小为M量级甚至G量级，典型大小为2M或1G。一个2M的巨页等于512个4K页，原来需要512个页表项记录映射关系，现在仅需1个页表项，页表项总数大大缩减，TLB的缓存数/页表项总数的比值提高了，命中率也就提高了。

参考：

<https://www.zhihu.com/question/431192638/answer/1589853910>

#### 使用巨页操作系统

巨页可以由用户手动设置或由操作系统自行管理，手动设定可以用如下命令指定巨页文件系统：

mount –t hugetlbfs –o uid=<value>,gid=<value>,mode=<value>,size=<value>,

nr\_inodes=<value> none <mount\_path>

需要注意的是，该文件系统上不能使用常用的read/write访问文件，只能使用mmap即内存映射的模式或共享内存方式（shmget、shmat，最终仍然调用mmap）来访问。

命令参数中：uid和gid为指定用户的id和组id，mode为读写执行模式，size为要使用的巨页文件大小，nr\_inodes为该文件系统最大inode数，限制文件及文件夹的数目，mount\_path为挂载路径。

#### 透明巨页

由操作系统自行管理的巨页称为透明巨页（THP，Transparent Huge Page）。开启了透明巨页后，操作系统会根据内存使用情况判断是都需要将若干较小的页（比如4K的页）合并为较大的页（比如2M的页）进行管理。

使用以下命令可以查询是否启用了THP：

cat /sys/kernel/mm/transparent\_hugepage/enabled

always：尽量使用；never：禁用；madvise：避免内存改变。

启用/禁用透明巨页：

1、开机启动设定

可以通过修改grub.cfg实现，在启动项中增加：

transparent\_hugepage=[always|never|madvise]

2、临时设置，关机或重启后失效

echo [always|never|madvise] > sys/kernel/mm/transparent\_hugepage/enabled

#### 故障

在MySQL中出现这样的问题，巨页占据了太多的内存（130G/150G，这个原因可能是并发执行大结果集操作，也可能是某些线程申请内存没有释放的bug），bufferpool内存也很大，接着使用swap分区，最终导致omm（Out Of Memory Killer）。

参考：

<https://blog.csdn.net/weixin_42942173/article/details/103739003>

### 缺页异常

通过get\_free\_pages申请一个或多个物理页面

换算addr在进程pdg映射中所在的pte地址

将addr对应的pte设置为物理页面的首地址

系统调用：Brk—申请内存小于等于128kb，do\_map—申请内存大于128kb



### 页面调度算法

页面调度算法：FIFO，Optimal（最佳置换），LRU，Clock，改进的Clock算法

Redis内存置换算法：FIFO、LRU、LFU

#### FIFO

最简单的页面置换算法是先入先出（FIFO）法。这种算法的实质是，总是选择在主存中停留时间最长（即最老）的一页置换，即先进入内存的页，先退出内存。理由是：最早调入内存的页，其不再被使用的可能性比刚调入内存的可能性大。建立一个FIFO队列，收容所有在内存中的页。被置换页面总是在队列头上进行。当一个页面被放入内存时，就把它插在队尾上。

这种算法只是在按线性顺序访问地址空间时才是理想的，否则效率不高。因为那些常被访问的页，往往在主存中也停留得最久，结果它们因变“老”而不得不被置换出去。

FIFO的另一个缺点是，它会产生Belady现象,即在增加存储块的情况下，反而使缺页中断率增加了。

#### Optimal（最佳置换）

这是一种理想情况下的页面置换算法，但实际上是不可能实现的。该算法的基本思想是：发生缺页时，有些页面在内存中，其中有一页将很快被访问（也包含紧接着的下一条指令的那页），而其他页面则可能要到10、100或者1000条指令后才会被访问，每个页面都可以用在该页面首次被访问前所要执行的指令数进行标记。最佳页面置换算法只是简单地规定：标记最大的页应该被置换。这个算法唯一的一个问题就是它无法实现。当缺页发生时，操作系统无法知道各个页面下一次是在什么时候被访问。虽然这个算法不可能实现，但是最佳页面置换算法可以用于对可实现算法的性能进行衡量比较。

当请求页面不在内存中时，选择已在内存中的永不使用的或者是在最长时间内不再被访问的页面置换出去，将请求的页面换入。

#### LRU

当请求页面不在内存中时，将最近最久未用的页面置换出去。用栈来存储内存中的页面，将栈底页面换出，将请求页面换入压入栈顶。

LRU算法是与每个页面最后使用的时间有关的。当必须置换一个页面时，LRU算法选择过去一段时间里最久未被使用的页面。

LRU算法是经常采用的页面置换算法，并被认为是相当好的，但是存在如何实现它的问题。LRU算法需要实际硬件的支持。其问题是怎么确定最后使用时间的顺序，对此有两种可行的办法：

1、计数器。最简单的情况是使每个页表项对应一个使用时间字段，并给CPU增加一个逻辑时钟或计数器。每次存储访问，该时钟都加1。每当访问一个页面时，时钟寄存器的内容就被复制到相应页表项的使用时间字段中。这样我们就可以始终保留着每个页面最后访问的“时间”。在置换页面时，选择该时间值最小的页面。这样做，[1] 不仅要查页表，而且当页表改变时（因CPU调度）要 维护这个页表中的时间，还要考虑到时钟值溢出的问题。

2、栈。用一个栈保留页号。每当访问一个页面时，就把它从栈中取出放在栈顶上。这样一来，栈顶总是放有目前使用最多的页，而栈底放着目前最少使用的页。由于要从栈的中间移走一项，所以要用具有头尾指针的双向链连起来。在最坏的情况下，移走一页并把它放在栈顶上需要改动6个指针。每次修改都要有开销，但需要置换哪个页面却可直接得到，用不着查找，因为尾指针指向栈底，其中有被置换页。

#### Clock

当某一页首次装入内存中时，则将该页框的使用位设置为1；当该页随后被访问到时（在访问产生缺页中断之后），它的使用位也会被设置为1。

当请求页面不在内存中时，查找内存中的页面，每当遇到一个使用位为1的页框时，就将该位重新置为0；如果在这个过程开始时，缓冲区中所有页框的使用位均为0时，则选择遇到的第一个页框置换；如果所有页框的使用位均为1时，则指针在缓冲区中完整地循环一周，把所有使用位都置为0，再次循环遍历，置换第一个遇到的使用位为0的页面。

#### 改进的Clock算法

## 物理地址

通常计算机使用大容量的、单位容量价格较为低廉的DRAM作为内存。内存的最小单位为bit，不过访问内存时一般以8bit也就是字节为单位。硬件层会将计算机所有内存条抽象为一个连续的字节数组，内存条上的每一个字节大小的存储位置将对应于这个数组中的一个元素，该元素的数组下标称为这个字节的物理地址（Physical Address，简称PA）。

Linux 内核会将物理内存分为3个管理区（其实就是将页划分为区），分别是：



### ZONE\_DMA

ZONE\_DMA内存区域。**包含0MB~16MB之间的内存页框**，可以由老式基于ISA的设备通过DMA使用，**直接映射到内核的地址空间**。

### ZONE\_NORMAL

ZONE\_NORMAL普通内存区域。包含16MB~896MB之间的内存页框，常规页框，直接映射到内核的地址空间。

### ZONE\_HIGHMEM

ZONE\_HIGHMEM高端内存区域。包含896MB以上的内存页框，不进行直接映射，可以通过永久映射和临时映射进行这部分内存页框的访问。

## 逻辑地址/虚拟地址

### 背景

无论是编译型还是解释型的应用程序，其指令和静态数据都会被编码为一定的内存地址，这些地址在**程序运行之前即已确定**（即写死的，不是动态生成的）。当直接应用程序使用物理地址时，多个程序中的相同地址对应相同的内存块，在这些程序交替执行时，将不得不反复的替换物理内存中的内容。除非是单进程不需要映射，只要是多进程程序无法知道可用的内存地址，因此必须做映射。

有了虚拟地址后，操作系统可以将不同程序的相同的虚拟地址映射到不同的物理地址上，这样大大**提升了程序的执行效率**。同时操作系统还可以对不同的应用程序进行有效的隔离，**提高了安全性**。

### 大小

**虚拟内存＞物理内存**

假设你的计算机是32位，那么它的地址总线是32位的，也就是它可以寻址0~0xFFFFFFFF（4G）的地址空间，但如果你的计算机只有256M的物理内存0x~0x0FFFFFFF（256M），同时你的进程产生了一个不在这256M地址空间中的地址，那么计算机该如何处理呢？

先说明计算机的内存分页机制。计算机会对虚拟内存地址空间（32位为4G）分页产生页（page），对物理内存地址空间（假设256M）分页产生页帧（page frame），这个页和页帧的大小是一样大的，所以呢，在这里，虚拟内存页的个数势必要大于物理内存页帧的个数。在计算机上有一个页表（page table），就是映射虚拟内存页到物理内存页的，更确切的说是页号到页帧号的映射，而且是一对一的映射。但是问题来了，虚拟内存页的个数 > 物理内存页帧的个数，岂不是有些虚拟内存页的地址永远没有对应的物理内存地址空间？不是的，操作系统是这样处理的。操作系统有个页面失效（page fault）功能。操作系统找到一个最少使用的页帧，让他失效，并把它写入磁盘，随后把需要访问的页放到页帧中，并修改页表中的映射，这样就保证所有的页都有被调度的可能了。这就是处理虚拟内存地址到物理内存的步骤。

虚拟内存地址由页号和偏移量组成。页号对应的映射到一个页帧。那么，说说偏移量。偏移量就是我上面说的页（或者页帧）的大小，即这个页（或者页帧）到底能存多少数据。举个例子，有一个虚拟地址它的页号是4，偏移量是20，那么他的寻址过程是这样的：首先到页表中找到页号4对应的页帧号（比如为8），如果页不在内存中，则用失效机制调入页，否则把页帧号和偏移量传给MMU（CPU的内存管理单元）组成一个物理上真正存在的地址，接着就是访问物理内存中的数据了。总结起来说，虚拟内存地址的大小是与地址总线位数相关，物理内存地址的大小跟物理内存条的容量相关。

注意：这里的核心就是页面置换算法！

**虚拟内存＜物理内存**

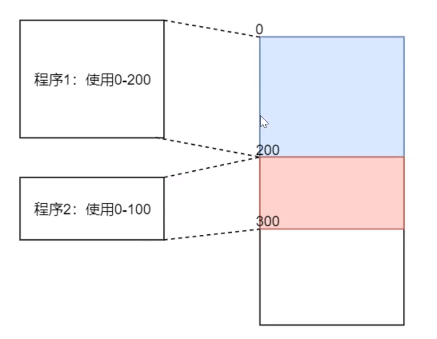
### 好处

虚拟地址的好处：

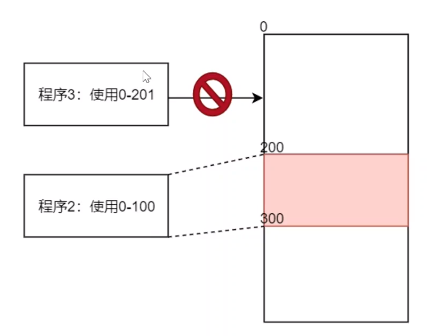
1. 避免用户直接访问物理内存地址，防止一些破坏性操作，保护操作系统
2. 每个进程都被分配了4GB的虚拟内存，用户程序可使用比实际物理内存更大的地址空间。这样势必会导致多个程序使用的内存总和大于物理内存，此时会借助磁盘，并将不着急使用的内存先放到磁盘，PT（Page Table）对应的帧号只显示磁盘。

### 映射方法

一种简单的映射思路：固定偏移量映射。

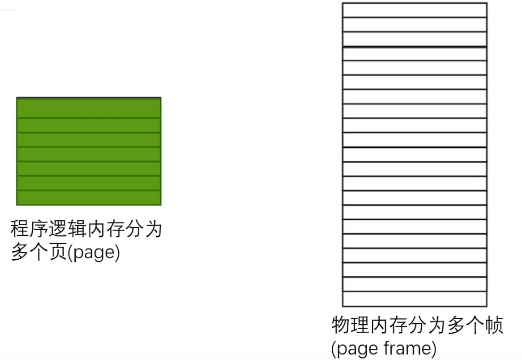


程序使用的内存区域为0~200，这可以通过提前预估分配程序的内存范围，这0~200区域内可能会有内存分配了未使用，称为内碎片。

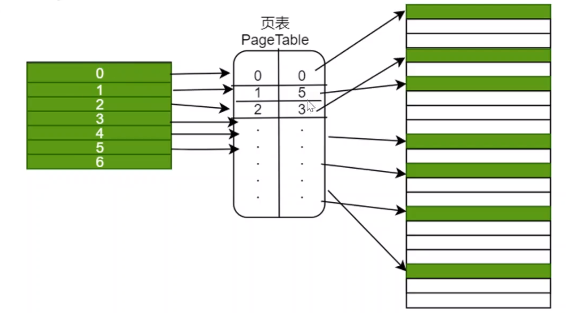


当0~200这部分内存使用完毕被释放出来，程序3使用0~201内存区域的时候，是无法直接使用这块连续内存的（因为不够用）。当很长一段时间都没有0~200大小的内存使用，那么这块内存将一直被闲置，这就是外碎片。

为了解决内碎片和外碎片，采用分页的思想。分页就是将逻辑地址和物理地址都切分为固定大小的page。



页和页帧之间通过页表实现映射：



页表中不仅存储了页和页帧映射，还存储了当前条目是否可用，页读写权限，当前帧是否脏等。

现代处理器中往往不再使用直接物理寻址的方式访问内存，而通过所谓的虚拟地址VA来访问内存。在CPU芯片中增加了一个叫内存管理单元（MMU=Memory Manager Unit）的部分用于地址翻译。

CPU将要访问的内存的虚拟地址发给MMU，MMU翻译为物理地址后请求内存，内存返回实际数据给CPU。

### 虚拟地址的划分

虚拟地址空间被划分为内核空间和用户空间。

用户进程能访问的是用户空间，每个进程都有自己独立的用户空间，虚拟地址范围从从0x00000000至0xBFFFFFFF总容量3G 。

用户进程通常只能访问用户空间的虚拟地址，只有在执行内陷操作或系统调用时才能访问内核空间。

当进程处于内核态时（通常是发生系统调用时，此时执行的是内核和设备驱动代码）可以访问内核空间和用户空间的所有地址。当进程处于内核态时，只能访问用户空间的内存地址。

下图是32位Linux系统的典型地址划分，用户空间：内核空间=3:1。



对于64位系统，2^64非常巨大，实际用不到那么大的虚拟地址空间。Linux一般使用42位或47位地址空间，而此时内核空间与用户空间也不是连续的，从0x0开始的128T虚拟地址空间为用户空间，从而虚拟地址空间顶部开始向下的128T空间为内核空间。

Linux上用轻量级进程实现线程，因此实际上每个线程都可以各自处于用户态或内核态。

**进程虚拟地址空间**

每个进程都有自己的虚拟地址空间，如下图所示，两个不同的进程A、B的地址空间内，各自只使用了部分地址，它们的实际地址可以映射到物理内存中的不同的段。注意进程的虚拟地址空间可以比物理内存大，比如1G物理内存可以支持4G的虚拟地址空间，系统可以将暂时不用的内存交换到磁盘上。



用户进程通常情况只能访问用户空间的虚拟地址，不能访问内核空间虚拟地址

内核空间是由内核负责映射，不会跟着进程变化；内核空间地址有自己对应的页表，用户进程各自有不同额页表

## MMU



MMU是一种硬件电路，它包含两个部件，一个是分段部件，一个是分页部件。

段页管理机制是分段式管理和分页式管理的组合，段式管理是逻辑上的管理方式，分页管理是偏物理上的管理方式。

**分段机制把一个逻辑地址转换为线性地址**。

**分页机制把一个线性地址转换为物理地址**。

注：MMU技术使得进程自己认为独占内存。

### 分段机制

我们平时所说的堆区、栈区都是分段的体现。现在X86结构已经不再采用分段分页机制，一般都是采用分页机制。

在实际中我们还会听到段错误等，其实分段只是保留了逻辑上的意义，即我们所说的段已经不再是影响内存管理的段了，而更多的是程序层面一种逻辑上的段。

#### 段选择符

1、为了方便快速检索段选择符，处理器提供了6个分段寄存器来缓存段选择符，它们是：cs，ss，ds，es，fs和gs

2、段的基地址(Base Address)：在线性地址空间中段的起始地址

3、段的界限(Limit)：在虚拟地址空间中，段内可以使用的最大偏移量

#### 分段实现

逻辑地址的段寄存器中的值提供段描述符，然后从段描述符中得到段基址和段界限，然后加上逻辑地址的偏移量，就得到了线性地址。



### 分页机制

#### 实现原理

分页机制是在分段机制之后进行的，它进一步将线性地址转换为物理地址。

10位页目录，10位页表项，12位页偏移地址

单页的大小为 4KB



**缺页中断( Page Fault )是只当软件试图访问一个虚拟地址时，经过段页转换为物理地址之后，此时发现该页并没有在内存中，这时cpu就会报出中断，再进行相关虚拟内存的调入工作或者分配工作，如果出现异常也可能直接中断**。

**举例：**

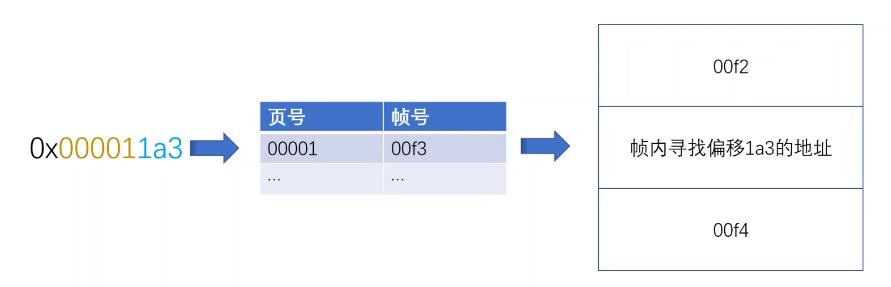
机器：32位系统，256MB内存，页大小为4KB

程序：32位程序

分析：4K=12bit，因而逻辑地址32bit=20bit页号+12bit偏移

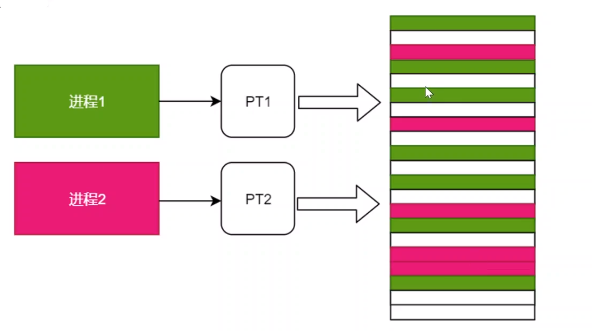
物理地址28bit（2^28=256MB）=16bit帧号+12bit偏移

通过页号查找帧号，找到内存中的地址：



如果对应帧号是磁盘，会发生缺页中断。会触发程序进入内核态，内核将这一帧对应的数据加载到物理内存的帧中，然后将帧号填写到page table中，重新进行寻址。如果当前所有帧都满了，那么应该怎么加载这个缺页中断的帧呢？这就是内存置换算法。

**每一个进程都有自己专属的页表**：



#### 特点

1. 分页使得每个程序都有很大的逻辑地址空间，通过映射磁盘和高效的置换算法，使得内存“无限大”；
2. 分页使不同的进程的内存隔离，保证了安全；
3. **分页降低了内存碎片问题**；
4. 分页过程中，需要两次读内存时间上有待优化（第一次从页表中读取页号对应的帧号，第二次根据帧号读取内存中具体数据），页表占用空间较大也有待优化：

4.1时间优化

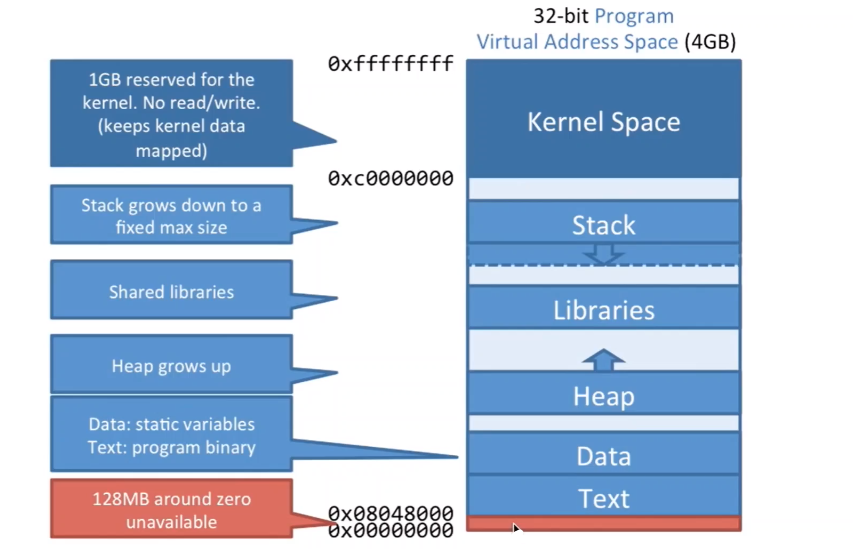
将最常访问的几个（一般8~128个左右）页表项存到访问速度最快的硬件中，一般是在MMU（内存管理单元），这个小表的名称为TLB（Translation Lookaside Buffer），可以称其为快表。

先寻址先查找TLB，然后miss后再查PT。快表命中率很高，因为一个事实：程序最常访问的页没几个。

4.2空间优化

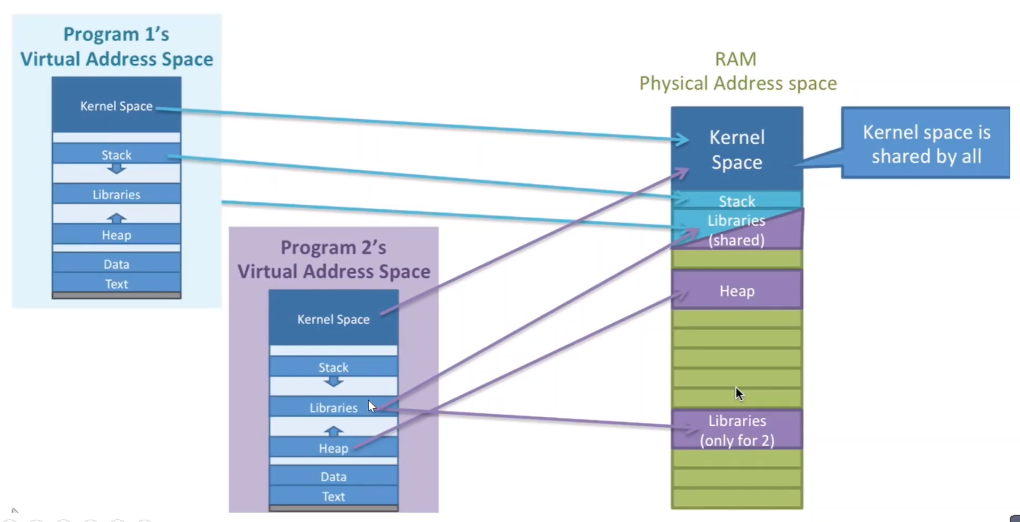
多级页表

#### 应用



malloc要申请大于128KB的内存会调用mmap，在堆和栈区域之间申请内存。这个和lib区（函数库区）其实是相同的位置，因为他们都是页映射磁盘（mmap就是文件映射内存的系统调用）。

lib区：Linux常见函数库，so文件，Windows的dll文件放置在这里。



## VFIO

1、UIO的出现，允许将驱动程序用到用户态空间里实现，但UIO有它的不足之处，如不支持DMA、中断等；

2、随着**虚拟化**的出现，IOMMU也随之出现，**IOMMU为每个直通的设备分配独立的页表，因此不同的直通设备(passthrough)，彼此之间相互隔离**；

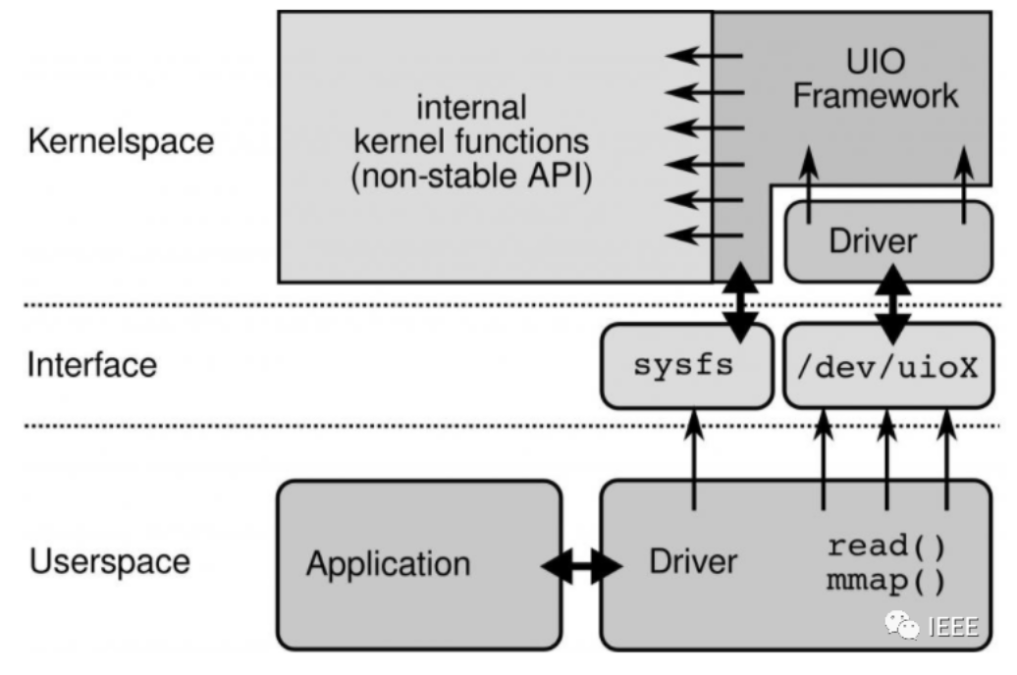
3、有一些场景，多个PCI设备之间是有相互联系的，他们互相组成一个功能实体，彼此之间是可以相互访问的，因此IOMMU针对这些设备是行不通的，随之出现VFIO技术，VFIO兼顾了UIO和IOMMU的优点，**在VFIO里，直通的最小单元不再是某个单独的设备了，而是分布在同一个group的所有设备**；VFIO可以安全地把设备IO、中断、DMA等暴露到用户空间；

4、KVM的PCI、PCIE设备直通，默认都是通过VFIO实现的（通过virsh attach-device xxx会自动插vfio的相关ko，自动生成vfio的container）；

5、PCIE与PCI直通的区别是：PCI只能直通给某个特定的虚拟机，而PCIE有可能可以给多个虚拟机用，如具有SR-IOV功能的PCIE设备，通过在HOST上抽象出多个的VF，每个VF再通过VFIO直通给虚拟机，最终的表现就是一个物理PCIE网卡可以直通给多个虚拟机用；

6、SR-IOV是针对PCIE设备的，PCI设备理论上不具有SR-IOV功能。

### UIO



UIO框架导出sysfs和/dev/uioX 2套用户态接口，用户对设备节点/dev/uioX进行设备控制，mmap()接口用于映射设备寄存器空间，write()接口用于控制中断关闭/打开，read()接口用于等待一个设备中断。

因为对于设备中断的应答必须在内核空间进行，所以在内核空间有一小部分代码用来应答中断和禁止中断，其余的工作全部留给用户空间处理。如果用户空间要等待一个设备中断，它只需要简单的阻塞在对/dev/uioX的read()操作上。当设备产生中断时，read()操作立即返回。UIO也实现了poll()系统调用，你可以使用select()来等待中断的发生。select()有一个超时参数可以用来实现有限时间内等待中断。

UIO的几个特点：

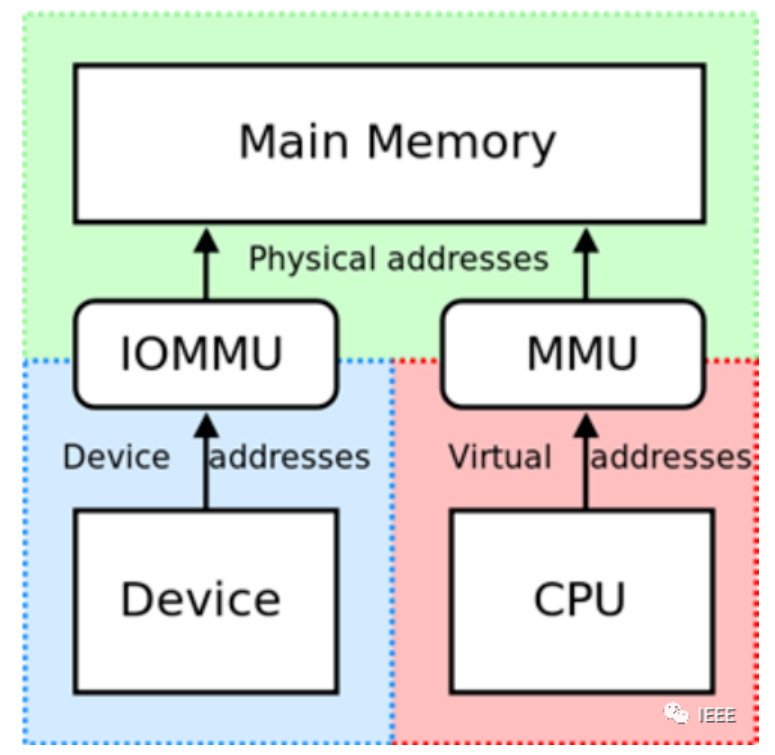
一个UIO设备最多支持5个mem和portio空间mmap映射。

UIO设备的中断用户态通信机制基于wait\_queue实现。

一个UIO设备只支持一个中断号注册，支持中断共享。

总的来说，UIO框架适用于简单设备的驱动，因为它不支持DMA，不能支持多个中断线，缺乏逻辑设备抽象能力。

### IOMMU



### VFIO

#### 概述

UIO不支持DMA，所以通过DMA传输大流量数据的IO设备，如网卡、显卡等设备，无法使用UIO框架，VFIO做为UIO的升级版，主要就是解决了这个问题。通过用户态配置IOMMU接口，可以将DMA地址空间映射限制在进程虚拟空间中。这对高性能驱动和虚拟化场景device passthrough尤其重要。

在VFIO框架中，有几个核心概念或对象：IOMMU、/dev/vfio、container、iommu\_group。

IOMMU是一个硬件单元，它可以把设备的IO地址映射成虚拟地址，为设备提供页表映射，设备通过IOMMU将数据直接DMA写到用户空间。之所以不共用MMU单元，是为了保证和进程的页表相互独立，防止设备访问进程的任意地址空间。所以VFIO的IOMMU功能保障了安全的非特权级别的用户态设备驱动机制。

 /dev/vfio是一个设备文件，作为一个IOMMU设备的用户态呈现。

container是内核对象，表示一个IOMMU设备，是一个IOMMU设备的内核态呈现。所以在VFIO中，container是IOMMU操作的最小对象（container中有多个iommu\_group）。

在虚拟化场景下，一个物理网卡可能要虚拟成几个虚拟网卡，或者说虚拟功能设备（VF），这几个VF共用一个IOMMU，所以VFIO模型增加一个iommu\_group的概念，用来表示共享同一个IOMMU的一组device。

#### 特点

VFIO的几个特点：

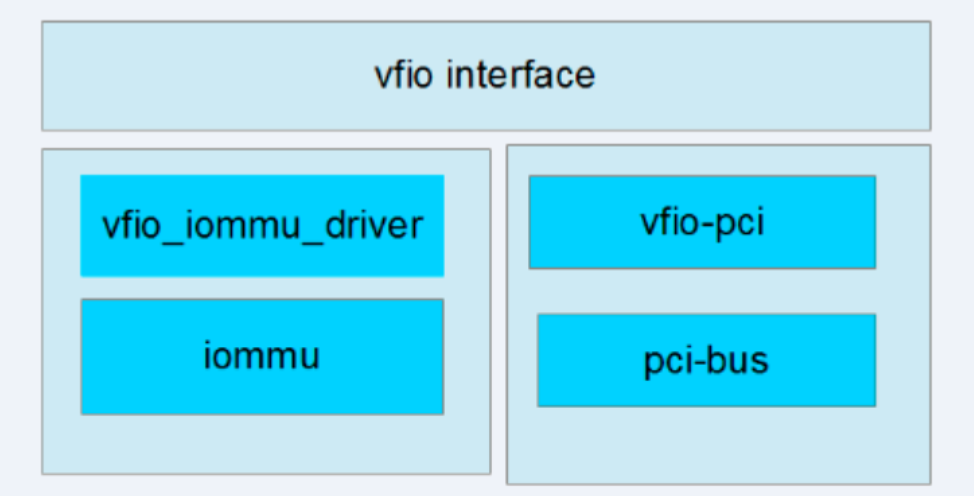
VFIO设备支持多中断号注册。

设备的中断用户态通信机制基于eventfd/irqfd实现。用户通过/dev/vfio设备select/poll/epoll，从而实现中断从内核态到用户态的异步事件通知。

支持对物理设备进行逻辑抽象。

仅支持pci intx中断共享，其他类型中断不支持共享。

VFIO仅支持特定IOMMU设备，如x86与PowerPC平台的PCI设备和ARM平台的platform设备。



#### 接口

VFIO是一套用户态驱动框架，它提供两种基本服务：

向用户态提供访问硬件设备的接口

向用户态提供配置IOMMU的接口

VFIO由平台无关的接口层与平台相关的实现层组成。接口层将服务抽象为IOCTL命令，规化操作流程，定义通用数据结构，与用户态交互。实现层完成承诺的服务。据此，可在用户态实现支持DMA操作的高性能驱动。在虚拟化场景中，亦可借此完全在用户态实现device passthrough。

VFIO实现层又分为设备实现层与IOMMU实现层。当前VFIO仅支持PCI设备。IOMMU实现层则有x86与PowerPC两种。VFIO设计灵活，可以很方便地加入对其它种类硬件及IOMMU的支持。

与KVM一样，用户态通过IOCTL与VFIO交互。可作为操作对象的几种文件描述符有：

1、Container文件描述符

打开/dev/vfio字符设备可得

2、IOMMU group文件描述符

打开/dev/vfio/N文件可得 (详见后文)

3、Device文件描述符

向IOMMU group文件描述符发起相关ioctl可得

逻辑上来说，IOMMU group是IOMMU操作的最小对象。某些IOMMU硬件支持将若干IOMMU group组成更大的单元。VFIO据此做出container的概念，可容纳多个IOMMU group。打开/dev/vfio文件即新建一个空的container。在VFIO中，container是IOMMU操作的最小对象。

要使用VFIO，需先将设备与原驱动拨离，并与VFIO绑定。

**用VFIO访问硬件的步骤：**

1、打开设备所在IOMMU group在/dev/vfio/目录下的文件

2、使用VFIO\_GROUP\_GET\_DEVICE\_FD得到表示设备的文件描述 (参数为设备名称，一个典型的PCI设备名形如0000:03.00.01)

3、对设备进行read/write/mmap等操作

**用VFIO配置IOMMU的步骤：**

1、打开/dev/vfio，得到container文件描述符

2、用VFIO\_SET\_IOMMU绑定一种IOMMU实现层

3、打开/dev/vfio/N，得到IOMMU group文件描述符

4、用VFIO\_GROUP\_SET\_CONTAINER将IOMMU group加入container

5、用VFIO\_IOMMU\_MAP\_DMA将此IOMMU group的DMA地址映射至进程虚拟地址空间

#### 逻辑

VFIO设备实现层与Linux设备模型紧密相连，当前，VFIO中仅有针对PCI的设备实现层(实现在vfio-pci模块中)。设备实现层的作用与普通设备驱动的作用类似。普通设备驱动向上穿过若干抽象层，最终以Linux里广为人知的抽象设备(网络设备，块设备等等)展现于世。VFIO设备实现层在/dev/vfio/目录下为设备所在IOMMU group生成相关文件，继而将设备暴露出来。两者起点相同，最终呈现给用户态不同的接口。欲使设备置于VFIO管辖之下，需将其与旧驱动解除绑定，由VFIO设备实现层接管。用户态能感知到的，是一个设备的消失(如eth0)，及/dev/vfio/N文件的诞生(其中N为设备所在IOMMU group的序号)。由于IOMMU group内的设备相互影响，只有组内全部设备被VFIO管理时，方能经VFIO配置此IOMMU group。

把设备归于IOMMU group的策略由平台决定。在PowerNV平台，一个IOMMU group与一个PE对应。PowerPC平台不支持将多个IOMMU group作为更大的IOMMU操作单元，故而container只是IOMMU group的简单包装而已。对container进行的IOMMU操作最终会被路由至底层的IOMMU实现层，这实际上将用户态与内核里的IOMMU驱动接连了起来。

#### 总结

VFIO是一套用户态驱动框架，可用于编写高效用户态驱动；在虚拟化情景下，亦可用来在用户态实现device passthrough。通过VFIO访问硬件并无新意，VFIO可贵之处在于第一次向用户态开放了IOMMU接口，能完全在用户态配置IOMMU，将DMA地址空间映射进而限制在进程虚拟地址空间之内。这对高性能用户态驱动以及在用户态实现device passthrough意义重大。

### DMA

## swap分区

### 背景

Linux内核为了提高读写效率与速度，会将文件在内存中进行缓存，这部分内存就是Cache Memory（缓存内存）。即使你的程序运行结束后，Cache memory也不会自动释放。这就是导致你在Linux系统中程序频繁读写文件后，你会发现可用物理内存变少。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swap空间中，等到那些程序要运行时，再从swap分区中恢复保存的数据到内存。这样，系统总是在物理内存不够时，才进行swap交换。

### 操作

#### 查看swap

Linux可以借助磁盘空间暂时保存内存中的内容，来实现虚拟地址空间大于物理内存，这部分磁盘空间就称为swap空间。Swap空间可以类比于windows系统上的虚拟内存。

查看swap分区的大小以及使用情况，一般使用free命令即可：

free –m

另外，我们还可以使用swapon命令查看当前swap相关信息，例如swap空间是swap partition，swap size，使用情况等详细信息：

swapon –s

cat /proc/swaps

#### 创建和使用swap

使用swap的相关命令：

mkswap <swap\_path>：在文件和设备上建立swap空间。

swapon/swapoff <swap\_path>：启动/关闭swap空间。

swapon/swapoff –a：启动/关闭所有swap空间。

启动时自动挂载swap空间，需要在/tec/fstab中增加：

<swap\_path> swap swap default 0 0

#### 设置大小

系统的swap分区大小设置多大才是最优呢？关于这个问题，应该说只能有一个统一的参考标准，具体还应该根据系统实际情况和内存的负荷综合考虑，像oracle的官方文档就推荐如下设置，这个是根据物理内存来做参考的。

|  |  |
| --- | --- |
| RAM | Swap Space |
| Up to 512MB | 2 times the size of RAM |
| Between 1024MB and 2048MB | 1.5 times the size of RAM |
| Between 2049MB and 8192MB | Equal to the size of RAM |
| More than 8192MB | 0.75 times the size of RAM |

在其他博客上看到下面的一个推荐设置，不清楚具体标准依据，作为一个参考：

4GB以为的物理内存，swap设置为内存的两倍。

4~8GB的物理内存，swap等于内存大小。

8~64GB的物理内存，swap设置为8GB。

64GB~256GB的物理内存，swap设置为16GB。

#### 启用swap分区

使用swapon启用交换分区：

swapon /dev/mapper/VolGroup01-LogicVolum01

此时查看交换分区的使用情况，你会发现used为0GB。

#### 关闭swap分区

使用swapoff关闭交换分区：

swapoff /dev/mapper/VolGroup01-LogicVolum01

#### 开机自启动

有两种临时修改swappiness参数的方法，系统重启后失效：

方法一：

echo 10 > /proc/sys/vm/swapiness

方法二：

sysctl vm\_swappiness=10

注：永久修改swappiness参数的方法就是在配置文件/etc/sysctl.conf里面修改vm\_swappinmess的值，然后重启系统。

#### 缩小swap分区

缩小swap分区大小的指令是：

lvcreate –L size /dev/mapper/\*

如果是增大swap分区大小，那么就需要拓展正在使用的swap分区的逻辑卷，此处使用lvreduce命令收缩逻辑卷。

#### 格式化swap分区

格式化swap分区的指令：

mkswap /dev/mapper/\*

### 内存换出

Linux通过内核线程kswapd来周期性检测内存使用情况，并执行将内存移出到swap的操作。内存的移出以页为单位，所以也称为内存页换出。选择哪些页被换出基于LRU原则，即最近最少使用的页被优先换出。内核中设定了high、low、min三个内存水位线，低于high表示有一定压力，此时就有可能触发内存换出，低于low表示压力很大，低于min表示内存非常紧张。High和low基于min计算，min水位线可用以下命令查询：

cat /proc/sys/vm/min\_free\_kbytes

注：在排查内存暴涨，是否启用swap分区的时候，可以查看该参数确认。

参考：

<https://www.cnblogs.com/muahao/p/8082997.html>

<https://blog.csdn.net/petib_wangwei/article/details/75135686>

### 内存换入

如果进程要访问的内存页已经被移出到swap空间中，而不在物理内存中，那么进程对该页的访问会触发缺页异常。缺页异常会被内存捕获，内核检查发现内存页在swap中，则会发出调页请求，从swap中将页载入到物理内存中，然后进程可继续正常运行。



### Linux释放内存的机制

Linux并非只有一种释放内存的机制。由于Linux倾向于尽量使用内存作为缓存使用，因此实际上内存中还会有文件缓存。在内存不足时，Linux也可以通过清除文件缓存的方式来释放内存。

swappiness值（0~100）指明了系统在释放内存时的倾向，0表示尽量不使用swap方式而通过释放文件缓存来缓解内存压力，100则表示以同样优先级使用两种方式，该值可使用cat /proc/sys/vm/swapiness查看。如果仍无法释放足够的内存，那么系统会选择杀死一些进程来释放内存。

杀死进程指的是发生OMM（Out Of Memory）时，系统按照一定的准则对各个用户进程的内存使用情况计算一个数值，按照这个数值排序，按顺序杀死进程，直至有足够的内存可以使用。如果出现进程莫名其妙退出的情况，可能就是由于系统OOM被杀死了，查看是否发生了OMM，可检查/var/log/message文件或者dmesg命令的输出中是否有Out Of Memery及kill的字样。

## Cache与Buffer

Cache和buffer都是内存中单独开辟出来的一段区域，用于暂存数据，但是用于不同。

**Cache（缓存）**和**Buffer（缓冲）**是两个不同的概念，简单的说，**Cache是加速“读”，而buffer是缓冲“写”，**前者解决读的问题，保存从磁盘上读出的数据，后者是解决写的问题，保存即将要写入到磁盘上的数据。在很多情况下，这两个名词并没有严格区分，常常把读写混合类型称为buffer cache。

注：在内存比较紧张的时候，可以通过释放cache（因为存储的是从磁盘读取的内容，所以一般都会比较大一些，buffer是缓存写到磁盘的内容不会太大）来提高内存利用空间。

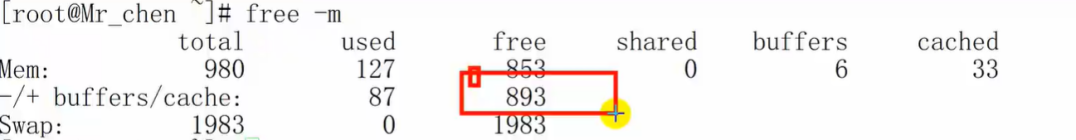
### 概述

查看内存使用：



注：这里的空闲表示没有被使用的内存，文件磁盘缓存将磁盘数据加载到内存，加速访问，是动态置换的，不是强制保留的。我们一般关注可用available即可。

剩余空间大小是真正剩余内存+cache/buffer：



Page Cache：以Page为单位，缓存文件内容，缓存在Page Cache中的文件数据，能够更快的被用户读取。同时对于带buffer的写入操作，数据再写入到Page Cache中即可立即返回，而不需等待数据实际持久化到磁盘，进而提高了上层应用读写文件的整体性能。

Buffer Cache：磁盘的最小数据单位为sector/扇区，每次读写磁盘都是以sector为单位对磁盘进行操作。Sector大小跟具体的磁盘类型有关，有的为512byte，有的为4K bytes。无论用户是希望读取1个byte，还是10个byte，最终访问磁盘时，都必须以sector为单位读取，如果裸读磁盘，那意味着数据读取的效率会非常低。同样，如果用户希望向某个位置写入（更新）1个byte的数据，它也必须刷新一个sector，言下之意，则是在写入这1个byte之前，我们需要先将该1byte所在的磁盘sector数据全部读出来，在内存中，修改对应的1个byte数据，然后再将整个修改后的sector数据，一口气写入到磁盘。为了降低这类低效访问，尽可能的提升磁盘访问性能，内核会在磁盘sector上构建一层缓存，它以sector的整数倍力度单位（block），缓存部分sector数据再内存中，当有数据读取请求时，它能够直接从内存中将对应数据读出。当有数据写入时，它可以直接在内存中直接更新指定部分的数据，然后再通过异步方式，把更新后的数据写回到对应磁盘的sector中，这层缓存则是块缓存Buffer Cache。

### 应用

Oracle中的log buffer是解决redo写入的问题，而data buffer cache则解决data block的读写问题。对于Oracle来说，如果IO没有在SGA中命中，都会发生物理IO，Oracle并不关心底层存储的类型，可能是一套存储系统，可能是本地磁盘，可能是RAID 10，也可能是RAID 5，可能是文件系统，也可能是裸设备，或是ASM。总之，Oracle把底层的存储系统称为存储子系统。

在存储系统中，cache几乎无处不在，文件系统有cache，存储有cache，RAID控制器上有cache，磁盘上也有cache。为了提高性能，Oracle的一个写操作，很有可能写在存储的cache上就返回了，如果这时存储系统发生问题，Oracle如何来保证数据一致性的问题。

Oracle数据库最重要的特性是：Write ahead logging，在data block在写入前，必须保证首先写入redo log，在事务commit时，同时必须保证redo log被写入。Oracle为了保证数据的一致性，对于redo log采用了direct IO，Direct IO会跳过了OS上文件系统的cache这一层。但是，OS管不了存储这一层，虽然跳过了文件系统的cache，但是依然可能写在存储的cache上。

一般的存储都有cache，为了提高性能，写操作在cache上完成就返回给OS了，我们称这种写操作为write back，为了保证掉电时cache中的内容不会丢失，存储都有电池保护，这些电池可以供存储在掉电后工作一定时间，保证cache中的数据被刷入磁盘，不会丢失。不同于UPS，电池能够支撑的时间很短，一般都在30分钟以内，只要保证cache中的数据被写入就可以了。存储可以关闭写cache，这时所有的写操作必须写入到磁盘才返回，我们称这种写操作为write throuogh，当存储发现某些部件不正常时，存储会自动关闭写cache，这时写性能会下降。

RAID卡上也有cache，一般是256M，同样是通过电池来保护的，不同于存储的是，这个电池并不保证数据可以被写入到磁盘上，而是为cache供电以保护数据不丢失，一般可以支撑几天的时间。还有些RAID卡上有flash cache，掉电后可以将cache中的内容写入到flash cache中，保证数据不丢失。如果你的数据库没有存储，而是放在普通PC机的本地硬盘之上的，一定要确认主机中的RAID卡是否有电池，很多硬件提供商默认是不配置电池的。当然，RAID卡上的cache同样可以选择关闭。

磁盘上的cache，一般是16M-64M，很多存储厂商都明确表示，存储中磁盘的cache是禁用的，这也是可以理解的，为了保证数据可靠性，而存储本身又提供了非常大的cache，相比较而言，磁盘上的cache就不再那么重要。SCSI指令中有一个FUA(Force Unit Access)的参数，设置这个参数时，写操作必须在磁盘上完成才可以返回，相当于禁用了磁盘的写cache。虽然没有查证到资料，但是我个人认为一旦磁盘被接入到RAID控制器中，写cache就会被禁用，这也是为了数据可靠性的考虑，我相信存储厂商应该会考虑这个问题。

至此，我们可以看到Oracle的一个物理IO是经历了一系列的cache之后，最终被写入到磁盘上。cache虽然可以提高性能，但是也要考虑掉电保护的问题。关于数据的一致性，是由Oracle数据库，操作系统和存储子系统共同来保证的。

## OMM

参考：

<https://www.zhihu.com/question/21972130>

<https://www.cnblogs.com/MacoLee/p/5666951.html>

<https://blog.csdn.net/liukuan73/article/details/43238623>

### 背景

Linux内核根据应用程序的要求分配内存，通常来说应用程序分配了内存但是并没有实际全部使用，为了提高性能，这部分没用的内存可以留作它用，这部分内存是属于每个进程的，内核直接回收利用的话比较麻烦，所以内核采用一种过度分配内存（over-commit memory）的办法来间接利用这部分“空闲”的内存，提高整体内存的使用效率。一般来说这样做没有问题，但当大多数应用程序都消耗完自己的内存的时候麻烦就来了，因为这些应用程序的内存需求加起来超出了物理内存（包括swap）的容量，内核（OOM killer）必须杀掉一些进程才能腾出空间保障系统正常运行。

### 概述

比如某天一台机器突然ssh远程登录不了，但能ping通，说明不是网络的故障，原因是sshd进程被OOM killer杀掉了。重启机器后查看系统日志/var/log/messages会发现Out of Memory:Killprocess 1865（sshd）类似的错误信息。又比如有时VPS的MySQL总是无缘无故挂掉，或者VPS 经常死机，登陆到终端发现都是常见的 Out of memory 问题。这通常是因为某时刻应用程序大量请求内存导致系统内存不足造成的，这时会触发 Linux 内核里的 Out of Memory (OOM) killer，OOM killer 会杀掉某个进程以腾出内存留给系统用，不致于让系统立刻崩溃。如果检查相关的日志文件（/var/log/messages）就会看到下面类似的Out of memory:Kill process 信息：

...

Out of memory: Kill process 9682(mysqld) score 9 or sacrifice child

Killed process 9682, UID 27,(mysqld) total-vm:47388kB, anon-rss:3744kB, file-rss:80kB

httpd invoked oom-killer:gfp\_mask=0x201da, order=0, oom\_adj=0, oom\_score\_adj=0

httpd cpuset=/ mems\_allowed=0

Pid: 8911, comm: httpd Not tainted2.6.32-279.1.1.el6.i686 #1

...

21556 total pagecache pages

21049 pages in swap cache

Swap cache stats: add 12819103,delete 12798054, find 3188096/4634617

Free swap  = 0kB

Total swap = 524280kB

131071 pages RAM

0 pages HighMem

3673 pages reserved

67960 pages shared

124940 pages non-shared

Linux内核有个机制叫OOM killer（Out-Of-Memory killer），该机制会监控那些占用内存过大，尤其是瞬间很快消耗大量内存的进程，为了防止内存耗尽内核会把该进程杀掉。

### 分析

如果检查相关的日志文件（/var/log/messages）就会看到下面类似的 Out of memory: Kill process 信息（也可以使用dmessage命令查看）：

...

Out of memory: Kill process 9682 (mysqld) score 9 or sacrifice child

Killed process 9682, UID 27, (mysqld) total-vm:47388kB, anon-rss:3744kB, file-rss:80kB

httpd invoked oom-killer: gfp\_mask=0x201da, order=0, oom\_adj=0, oom\_score\_adj=0

httpd cpuset=/ mems\_allowed=0

Pid: 8911, comm: httpd Not tainted 2.6.32-279.1.1.el6.i686 #1

...21556 total pagecache pages21049 pages in swap cache

Swap cache stats: add 12819103, delete 12798054, find 3188096/4634617

Free swap = 0kB

Total swap = 524280kB131071 pages RAM0 pages HighMem3673 pages reserved67960 pages shared124940 pages non-shared

### 参数

我们可以在用户空间通过操作每个进程的oom\_adj 内核参数来决定哪些进程不这么容易被OOM killer选中杀掉。

比如，如果不想MySQL进程被轻易杀掉的话可以找到MySQL运行的进程号后，调整/proc/PID/oom\_score\_adj为-15（注意points越小越不容易被杀）防止重要的系统进程触发(OOM)机制而被杀死。

内核会通过特定的算法给每个进程计算一个分数来决定杀哪个进程，每个进程的oom分数可以在/proc/PID/oom\_score中找到。每个进程都有一个oom\_score的属性，oom killer会杀死oom\_score较大的进程，当oom\_score为0时禁止内核杀死该进程。设置/proc/PID/oom\_adj可以改变oom\_score，oom\_adj的范围为【-17，15】，其中15最大-16最小，-17为禁止使用OOM，至于为什么用-17而不用其他数值（默认值为0），这个是由linux内核定义的，查看内核源码可知：路径为linux-xxxxx/include /uapi/linux/oom.h。

## NUMA

### 术语

core：物理核，基本的逻辑执行单元

CPU：指集成了多个core的物理芯片

Socket：主板上的CPU芯片槽位

Thread：这里不是指软件线程而是硬件线程。基于超线程技术，一个core可以支持多个指令并发执行，可认为 core上有多个硬件线程，每个硬件线程在系统中体现为一个逻辑核

Processor：处理器，有时候指core，有时候指CPU，依上下文语境而定

Socket就是常说的“几路几核”服务器找那个的“路”

CPU情况可以查看系统的/proc/cpuinfo信息获取，其中显示的每一个processor一般指thread，即逻辑core。

### UMA

UMA（uniform memory access），即一致性内存访问。在UMA架构下，所有CPU对所有内存的访问时间相同。UMA架构在CPU数量少的时候，性能非常好，但**随着CPU数目增加，各CPU访问内存时对内存总线的争用回越来越剧烈，从而导致性能下降**。因此，从长期发展来看，UMA架构越来越无法适应多核时代的要求。



注：实际上目前不论X86还是ARM架构，都是多核，这种UMA已经无法适用。

### NUMA

NUMA（non-uniform memory access），即非一致性内存访问。在NUMA架构下，每个CPU都有自己的本地内存，如下图所示，紫框内为多个内存条组成的CPU本地内存，红线则为本地内存访问，其访问速度比较快。而如果一个CPU上的核要访问其他CPU的本地内存（远端内存），则需要通过图中的蓝线所示的CPU互联模块来进行，速度会明显慢于本地内存访问。



### 查看NUMA信息

#### numactl

在NUMA架构下，拥有本地内存的CPU称为node。使用numactl –hardware命令可以看到基本的NUMA信息，其中会显示node上的所有逻辑核编号以及本地内存大小、使用量等信息。

最下方的node distances显示了各个node访问本地和远端内存的时间代价的相对值。

#### numastat

numastat –c命令可以查看系统的NUMA使用情况，其中NUMA hit表示访问本地内存的总量，NUMA Miss表示访问远端内存的总量。

numastat –c <pro\_name>可以查看进程内存在node上的分布。

### 启用NUMA自平衡

由于NUMA架构下访问本地内存和远端内存的差异，**将进程的内存集中在执行该进程命令的逻辑核所在的node上，显然可以提高进程的性能**。系统提供了numad命令来对进程的内存做调解，它会逐步将进程使用的逻辑核与内存集中在相同的node上以提高性能。

numad：系统自动执行NUMA自平衡

numad –p <pid>：对指定的进程执行NUMA自平衡

nomad –r <pid>：对指定的进程取消NUMA自平衡

### NUMA的影响

一般情况下，使用NUMA自平衡可以提高性能，但在特殊情况下，反而会导致性能大幅下降，典型的如数据库场景，**数据库进程往往需要大量内存，当启动NUMA自平衡后，这些内存集中在同一个node**。当数据库需要更多的内存时，由于Linux的策略原因，会优先将内存置换到swap空间上，而不是在其他node上分配空间，这样就会导致大量的**内存换页**，造成数据库性能大幅降低甚至卡死。因此在使用NUMA自平衡策略前，需要对进程的内存行为以及系统的策略设置有充分了解。

注：访问远端内存带来的开销小于swap分区（即与磁盘的交互）。

可以采用下面的命令监控页面置换的情况：

sar -B 1

参考：

<https://www.cnblogs.com/luoahong/articles/7909008.html>

<https://www.cnblogs.com/xuanbjut/p/12011224.html>

关于NUMA对数据库的影响，参见：

http://cenalulu.github.io/linux/numa

# 内存统计信息

系统中内存使用情况：/proc/meminfo

进程的内存使用情况：/proc/28040/status

查询内存总使用率：free

查询进程 cpu 和内存使用占比：top

虚拟内存统计：vmstat

进程消耗内存占比和排序：ps aux –sort -rss

释放系统内存缓存：

/proc/sys/vm/drop\_caches

To free pagecache, use echo 1 > /proc/sys/vm/drop\_caches

To free dentries and inodes, use echo 2 > /proc/sys/vm/drop\_caches

To free pagecache, dentries and inodes, use echo 3 >/proc/sys/vm/drop\_caches

## 内存指标(/proc/meminfo)

/proc/meminfo包含了系统内存的统计信息，可以使用cat 命令查看其中的内容。

MemTotal：物理内存总量

MemFree：空闲物理内存

MemAvailable：可用物理内存，注意可用物理内存不等于空间物理内存，而是大致上等于MemFree+Cached

**Cached：用来缓存文件数据的物理内存大小，内存不足时可以释放这部分来缓解内存压力**

Buffer：用来缓存块设备数据占用的物理内存大小，这部分占用一般比较小

SwapCached：内存中和swap空间中都存在的内存大小，在内存紧张时这部分内存无需换出到swap空间，可直接释放

Active：物理内存中较活跃的部分，即最近被使用过的部分

Inactive：物理内存中不活跃的部分，即最近未被访问过的部分

Active+Inactive：一般略小于MemTotal-MemFree

Active(file)：活跃内存中属于文件缓存的部分

Inactive(file)：不活跃内存中术语文件缓存的部分，有内存压力时优先释放的部分

Active(anon)：活跃内存中术语各进程使用的部分

Inactive(anon)：不活跃内存中术语各进程使用的部分

Active=Active(file)+Active(anon)

Inactive=Inactive(file)+Inactive(anon)

Cache大致等于Active(file)+Active(anon)

SwapTotal：swap空间总量

SwapFree：swap空间的空闲大小，前面的SwapCached也算SwapFree

Dirty：脏页大小。Linux处理I/O时一般不直接写入磁盘，而先缓存到内存中，后续再集中写入。这部分缓存可能包括新写入的数据，也有可能包括从磁盘中读入，但已做修改的数据，Linux中称这种数据为“脏”的，而数据又是以页为单位管理的，因而这些数据被称为脏页。如果机器突然断电，那么脏页中的数据将因为未及时写入磁盘而丢失。

/proc/meminfo中各项指标可以用来对系统的内存情况进行判断，比如MemFree很小但MemAvailabel很大，那说明再启动新的进程可能需要释放文件缓存或交换内存到swap空间来空出一部分内存，但如果Inactive(file)也比较小，那么说明文件缓存正在被使用，释放文件缓存可能无法空出足够的内存空间，而需要使用swap空间，则性能明显会受到影响。另外，Active(file)和Dirty两项的值比较大，可以从侧面表明系统的I/O比较活跃。

## 内存相关命令

### top/free

top命令：显示物理内存、swap空间、Cached、Buffer使用情况，另外还有进程内存使用情况。

#### VIRT

1、进程“需要的”虚拟内存大小，包括进程使用的库、代码、数据，以及malloc、new分配的堆空间和分配的栈空间等；

2、假如进程新申请10MB的内存，但实际只使用了1MB，那么它会增长10MB，而不是实际的1MB使用量。

3、VIRT = SWAP + RES

注：之前碰到过由于bug导致进程启动的时候VIRT一直上涨，最后导致进程crash的问题。

#### RES

1、进程当前使用的内存大小，包括使用中的malloc、new分配的堆空间和分配的栈空间，但不包括swap out量；

2、包含其他进程的共享；

3、如果申请10MB的内存，实际使用1MB，它只增长1MB，与VIRT相反；

4、关于库占用内存的情况，它只统计加载的库文件所占内存大小。

5、RES = CODE + DATA

#### SHR

1、除了自身进程的共享内存，也包括其他进程的共享内存；

2、虽然进程只使用了几个共享库的函数，但它包含了整个共享库的大小；

3、计算某个进程所占的物理内存大小公式：RES – SHR；

4、swap out后，它将会降下来。

参考：

<https://www.cnblogs.com/xudong-bupt/p/8643094.html>

free命令：

free –m/g：以M/G为单位显示内存使用信息

free –h：以可读性最佳的方式显示内存使用信息

free –s <N>：以N秒为间隔不断获取内存使用信息

free –c <N>：显示N次信息后退出

### vmstat

vmstate –s：显示类似于/proc/meminfo中的主要内存指标，另外还显示一些CPU统计信息

vmstate <N>：以N秒为间隔显示MemFree、SwapFree、Buffers、Cached、si、so信息

vmstate –a <N>：以N秒为间隔显示MemFree、SwapFree、Active、Inactive、si、so信息

其中，si、so分别是每秒从swap换入内存的数据量和每秒从内存换出到swap的数据量

### sar

sar命令可以显示CPU、内存、网络、磁盘等方面统计信息，是Linux上监控系统使用状况的最强大的命令之一。

sar <N>：以N秒为间隔不断显示统计信息，如果不携带N这个参数，那么会显示从当日0点开始至当前时间以10分钟为粒度的统计信息

sar –r：显示与/proc/meminfo类似的主要内存指标

sar –R：显示物理内存、Buffers、Cached单位时间内变化情况，单位为页/s

sar –B：以页为单位统计了内存与磁盘的交互、缺页异常发生情况、kswapd扫描情况等多种信息

sar –H：显示巨页的统计信息

sar –S：显示swap的使用情况

sar –W：显示单位时间内swap换页情况

另外，系统的/var/log/sa目录下以sar开头的文件记录了每天的系统性能统计信息，每一类统计信息均以10分钟为粒度，可以在排查问题时追溯当时的系统运行状况。

sar命令包含在sysstet工具中。

### nmon

nmon是一款IBM开发的AIX/Linux系统性能监控工具，它的功能与前面的sar功能上比较接近。

Nmon提供两种监控方式。一种与top类似，动态刷新性能数据，并提供不同的按键以选择具体的监控项，内存项监控为M；一种是通过设置命令行参数，将需要统计的信息输出到文件中。对包含输出信息的文件，可以使用nmon analyser工具将统计数据导入到excel中，以生成形象化的报表。

# 进程内存布局

## 应用程序内存布局

### 内存分段

应用程序的内存分为多段，一般包括：

text段：保存二进制形式的应用程序可执行代码

data段：保存代码中在定义时就被初始化的全局/静态变量

bss段：保存代码中在定义时未被初始化的全局/静态变量

stack段：进程栈区域，代码执行时使用

heap段：进程的堆，动态分配内存时使用

内存映射区：可用于动态库、文件映射以及动态内存分配

### 32位布局

Linux的经典内存布局（早期使用）如下所示，进程的内存地址空间一般为4G，上部为内核空间，一般为1G，下部为用户空间，一般为3G。从0地址开始有一段保留地址，然后依次是text、data、bss段，堆中bss顶部开始，向高地址增长。内存映射区从映射区基地址开始，向高地址增长。栈从用户空间顶部开始，向低地址增长。



现代的32位Linux布局与经典布局不同，主要有如下两点：

内存映射区改为从栈顶部开始向低地址增长，即和堆相向增长，这样可以最大限度的利用中间的空闲地址空间。

在栈、内存映射区和堆的开始处均增加了一段随机大小的偏移，以防御堆溢出这样的攻击行为。

### 64位局部

64位Linux的布局与32位有些不同。如下图所示，在64位场景下，内核空间与用户空间不再是连续的了，而是由一大块未定义区域隔开，两者各有128T的地址空间。在这种情况下，地址空间一般都很充裕，而且不同段距离很远，无需像现代的32位布局那样保留随机间隔，因而64位用户空间的内存布局实际与32位经典布局类似。



### 申请内存方式

申请内存的系统调用包括：brk、mmap。

注：进程申请内存的方式为brk和mmap，这与内核申请内存方式：alloc\_pages申请页，kmalloc申请连续内存地址，vmalloc申请非连续内存地址不同。

而malloc等则是对这些进程申请内存的系统调用的封装，根据分配内存大小调用不同的系统调用（小于128B采用brk，大于128B采用mmap）。

#### brk

进程有一个称为program break的指针指向堆的顶部的地址，通过系统函数brk可以改变这个指针的位置，从而改变堆的大小（堆可以扩大也可以缩小）。这种申请内存的方法存在一个缺陷，即如果堆顶部分的内存一直在使用，就算其他部分未被使用，堆页无法收缩，这可能会造成隐形的内存泄露。



#### mmap

mmap函数通常被应用程序用来将文件映射到内存中，其实它也可以用来向系统申请新的内存映射区域作为动态内存来使用。

**mmap在堆栈之间的游离区域申请内存，<128B采用brk，>128B采用mmap，避免内存碎片**。

与堆不同的是，mmap可以在内存映射区中寻找空闲的合适大小的区域来分配，应用程序可以多次调用从而申请多个内存映射区，每一块映射的内存只要不再使用，都可以单独释放掉，不像堆那样受到堆顶内存的影响。

mmap申请的内存大小以页为单位。

## 进程内存段分析

### 内存信息

/proc/<pids>/maps包含进程内存使用信息，每一行包含：

01b68000-01b69000：虚拟地址段的地址范围

rwxp：rwx读、写、执行权限；p/s：私有/共享

01568000：可执行文件/共享库文件不同段的偏移量

fd:00：可执行文件/共享文件主次设备号，其余为00:00

71778881：可执行文件/共享库文件的inode号

/opt/./\*\*\*：可执行文件名/共享库名/段名

### text段

text段如下所示，text有可执行权限，即x属性：



可以通过objdump命令来查看可执行文件的反编译信息：

objdump –D <exec\_path>

在输出信息中，可以看到<\_FRAME\_END\_>，这个就是text段的结束位置，与maps中的结束位置可以匹配上：



### data段和bss段

类似于text段，可以通过nm命令来获取可执行程序的**符号表**信息，来确定data段和bss段的一些信息：

nm <exec\_path> | sort

从nm的输出可以看到data段和bss段的起始地址和整个数据段的截止地址，从而确定两段的对应地址范围。

nm命令后面加上sort是为了按照虚拟地址排序，分析起来更加方便。

关于data段和bss段：

bss段的起始位置实际在data段的地址范围内，这个是由于maps中是以页（一般为4K）为单位来显示的。

有的变量已经初始化了，但却在bss段中。

全局/静态变量初始化时如果只由0或NULL或’\0’赋值得到，那么仍然认为是未初始化的，以下全局变量均在bss段：



### libc段

在text段和data段之间还有一小段内存地址是gcc编译时插入的。main函数并不是程序的真正入口，在main函数执行前和执行后libc都做了大量工作，包括程序的初始化与终结，这一小段就是供libc使用的。

nm的输出中，\_FRAME\_END\_和\_GLOBAL\_OFFSET\_TABLE\_之间的就是libc相关的段：



### 内存映射区

在/proc/<pid>/maps里面可以看到类似于如下的段，这些就是进程内存映射区中动态库占用的内存部分。

与应用程序类似，动态库页包含text段、data段、bss段，在下图中，属性为rxp的是动态库的text段，而属性为rwp的则是动态库的data段和bss段，属性为rp的为libc使用的段。



### 动态库

#### 映射方式

动态库一般会被多个进程使用，而且在每个进程中，动态库被加载的虚拟地址一般也不相同。但实际上动态库只需要在物理内存中放置一份，不同进程中不同虚拟地址的动态库都被映射到这同一块物理内存，从而大大减少了内存占用，真正实现共享。



#### 动态库数据修改

动态库的数据段是可能被进程修改的，在这种情况下，系统会使用所谓的COW（copy-on-write）技术，将数据段部分在物理内存中再拷贝一份，然后将进程内该动态库的虚拟地址映射到新拷贝出来的物理内存，然后在新的这块物理内存上进行修改，这样将不影响其他进程使用动态库。



#### 注意点

从前面所述动态库的特点，可以得到如下注意点：

由于修改时新建数据段拷贝，因此不可以通过动态库中的全局变量，在进程间传递数据。因为修改后两个进程中的动态库的全局变量已指向不同的物理地址。

由于代码段被共用，因此如果出现某动态库S正在被进程A使用，然后动态库S的文件被替换为新版本，再启动一个使用S的进程B，那么当A下次调用S中的函数时，就会崩溃，因此此时S已被系统重新加载，原来函数地址的内容已变。

第二种情况是实际发生过的，在某软件升级过程设计中，没有注意另一升级相关的进程引用了被替换的动态库，导致替换动态库后该进程崩溃，软件升级被中断。

### 栈

#### /proc/<pid>/maps

在/proc/<pid>/maps的末尾可以看到标注为[stack]的内存区域，这个与标准的布局一致。但在maps的中间段，还有标记为[stack:<number>]的区域，这是怎么回事？

#### C程序栈

我们在maps底部看到的[stack]实际上只是进程的主线程的栈。进程的其他线程都是POSIX线程，使用pthread\_create创建。POSIX线程栈由glibc中的相应代码来实现，栈空间则通过mmap函数在内存映射区创建，如下图所示：



这就是maps信息中间出现的[stack:<number>]来源。线程栈的默认大小一般为8M或者10M，依Linux发行版不同而有所区别。

较早的Linux不会在maps中显示线程栈信息。

#### 查看C程序栈

pstack命令可以查看进程的栈（包括线程），用法为：

pstack <pid>

另一个可以查看进程堆栈的命令为gstack，用法与pstack一样，实际上Linux的pstack一般为指向gstack的一个软链接。

pstack和gstack对Java程序也有效，只是输出内容包含的是native层的栈信息，不包含Java层的信息。

pstack和gstack在排查进程死锁等问题时非常有效。

有时候，线程本身已经挂死，pstack和gstack查看进程栈时也会挂住，这时可以通过proc文件系统来查看进程栈信息：

cat /proc/<pid>/stack

cat /proc/<pid>/task/\*/stack

task下包含了进程中的所有线程，上面的命令可以查到所有线程的栈信息。不过需要注意的是，这个方法只能显示内核栈，不包含用户栈。此方法在排查进程因系统调用（比如I/O）挂死时很有效。

#### Java程序栈

对于Java程序，除了启动时的主线程之外，其他线程栈都由jvm来实现，而jvm则基于glibc的栈来实现Java线程的栈，所以Java程序的栈也在内部映射区。Linux上Hotspot VM对普通Java程序设置的默认栈大小为1M，如果想要自行指定Java线程栈的大小，可以通过-Xss或-XX:ThreadStackSize参数进行设置。

实际上普通Java线程的栈只有1016K，因为1M的栈空间中会被jvm分出分别称为red page和yellow page的隔离区，这两个区域均为一页大小，所以要扣除8K。另外可以观察到，栈的隔离空间总共为12K，包含了glibc的隔离区和jvm的隔离区。

#### 查看Java程序栈

之前说过pstack命令对Java程序只能获得native层的栈信息，Java层的栈信息要使用jstack命令获取：

jstack <pid>

jstack输出中的nid对应于pstack输出中的LWP（Light Weight Process）值，可以据此将同一个线程的Java栈与native栈联系起来。

按照之前所说的内存布局，堆就在静态区之上。然而事实上，也可能如下图一样，与text段、静态区一起放在内存映射区之上。操作系统的实现实际上是比较灵活的。

另外，还可以使用pmap命令来查看maps中的信息：

pmap <pid>

### 堆

Java程序中有堆和非堆（方法区，又称永久代）的说法，这个说法其实是从jvm的角度来说的。从操作系统进程的角度来说，Java的堆和非堆都在进程的堆上，tomcat的永久代和堆共1.5G（anon），在pmap输出中则为完整的一块内存。

### 其他内存段

在/proc/<pid>/maps的末尾，可以看到名为vsyscall和vdso的两个段。Vsyscall是早期为了提升系统调用的效率而引入的，应用程序可以通过访问vsyscall段的内存地址来触发系统调用，而避免通过软中断触发系统调用的低效率。Vdso则提供了与时间相关的几个系统调用的快速实现。系统将时间信息和相关系统调用映射到vdso的段，应用程序通过C库调用该段中的函数，可以避免一般系统调用中用户态与内核态的切换。



# 进程内存使用情况

## 进程的内存分布

进程虚拟地址空间中的页可能处于以下四种状态：

页已分配，且正处于物理内存中

页已分配，已经交换到swap空间

页已分配，但还未填充，比如文件映射时，未读取的部分对应的虚拟内存页

页尚未分配



根据以上的内存分布类型，一般用以下术语来指称分布在不同部分的内存：

驻留内存：指进程当前处于物理内存中部分的大小

虚拟内存：指虚拟地址空间中所有已分配的页的总大小

另外，还有共享内存，这个指多个进程共享的物理内存大小，包括使用共享内存函数指定创建的部分，以及动态库的部分，一般计算共享内存大小时，会按照进程分摊计算。

## 系统对进程内存资源的限制(ulimit)

操作系统对各进程的内存使用有各种显示，使用ulimit –a命令可以查看相关限制信息，其中与内存相关的有：

max locked memory：进程可锁定的最大内存值

max memory size：进程驻留内存的最大值

stack size：进程栈的最大值

virtual memory：进程虚拟内存的最大值

## 单独指定应用程序内存限制(/proc/<pid>/limits)

进程的资源限制可以在/proc/<pid>/limit中查到，其中hand limits是由超级用户设置并内核强制限制的资源上限，soft limits由不同用户自行设置，但不能超出hard limits的范围。Limits中包含与ulimit类似的内存配置。

limits中的限制可以在运行时动态修改，方法为：

echo –n “<resource\_name>=<soft\_limit\_value><hard\_limits\_value>” >/proc/<pid>/limits

## 进程内存使用情况(top)

监控单个进程资源占用可以使用以下命令：

top –p <pid>

其中与内存相关的为：

VIRT：进程当前虚拟内存的大小

RES：进程驻留内存的大小

SHR：进程共享内存大小

%MEM：RES占系统总内存的百分比

其中，RES最为重要。

## 进程内存使用情况

## /proc/<pid>

除了top命令外，还可以使用proc文件系统下的各种信息查看进程内存使用量：

cat /proc/<pid>/status中vm开头的项为内存使用信息，其中VmRSS为驻留内存大小。

cat /proc/<pid>/statm信息比较简略，其中第二项为驻留内存代销，注意statm统计数值指每部分占用的页数。

cat /proc/<pid>/private\_rss仅包含进程驻留内存信息，以页数来统计。

## /proc/<pid>/smaps

/proc/<pids>/smaps中详细记录了maps中各内存段的使用情况，其中：

Size：该段的虚拟地址长度

Rss：实际驻留内存在物理内存中的内存大小

Shared/Private：共享或私有的

Clean/Dirty：是否有过修改

如果一段内存是文件映射使用的，那么dirty表示该段对应的文件内容从磁盘载入后，已经被修改过。

如果一段内存使用作堆栈内存使用的，那么dirty表示该段内存已经分配且在使用。

## Java程序

Java程序的内存与普通程序不同，Java本身提供了很多查看内存的工具：

jstat主要用来查看垃圾回收相关的信息

jmap可以查看java堆中不同区域的使用情况，堆当中不同对象的数目大小等信息，还可以使用以下命令导出堆内存：

jmap –dump:format=b,file=<outfile> <pid>

导出dump文件可以使用jhat、jprofiler等工具进行分析。

## 读取内存/proc/<pid>/mem

/proc/<pid>/mem将进程内存的地址空间抽象为一个文件，内存地址即相对于文件头部的偏移量，这样就可以通过open、lseek、resd等文件I/O函数去读取内存的内容。

需要注意的是，由于程序运行时内存内容一直在改变，通过mem读取内存前要先通过ptrace暂停进程，读取结束后要恢复进程运行。

读取mem可以作为排查内存泄露等问题的辅助手段。

## 读取内存gcore

gcore是gdb包中提供的命令，它可以生成进程的coredump文件，而且在执行结束后进程可以继续正常运行。gcore相当于提供了进程运行时的一个截面信息，生成coredump后再用gdb分析，可以说是排查复杂问题的首选方法。

gcore的使用方法为：

gcore <pid> -o <outputfile>

不过需要注意的是，gcore导出的部分要比驻留内存大，对内存泄露时，进程内存占用很多的情况，最好不要使用gcore。

## 进程的omm值

在/proc/<pid>/omm\*中查看进程的omm值：

omm=out of memory，当操作系统发现可用内存耗尽时，会杀死一些进程以释放内存，每个进程都有omm值，作为操作系统杀死进程时的排序参考。

omm\_score\_adj：在发生omm时，被系统杀死优先级配置，范围为0~1000,0表示不会被杀，1000表示优先被杀。

omm\_score：表示系统对该进程当前的omm估值，数值越大越容易被杀。