伤神的博客

欢迎来到我的博客! 我的名字叫商洋，邮箱 comedshang@163.com；当前坐标在成都；

操作系统基础 – 内存管理(一) 虚拟内存 → 物理内存

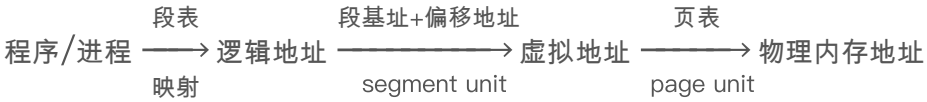
置顶 | 2019-11-19 | 2020-02-06 | 计算机科学与技术， 操作系统， 基础 | 阅读次数： 729

前言

本篇博文是操作系统基础系列之一；本文为作者的原创作品，转载需注明出处；本文以 32 位操作系统为基础讲解；

矛盾

现代操作系统对内存管理的设计和实现是异常复杂的，首先，程序指令、常量、以及堆栈等首先要通过段表映射为逻辑地址，然后通过 segment unit 将逻辑地址转换为虚拟地址，最后通过页表，将虚拟地址转换为物理内存地址；如图所示，



操作系统为什么不直接将程序指令放入物理内存中呢？而是非要采用这样复杂的设计呢？主要是因为用户(程序员)对编程的需求和硬件设计师在对物理内存的设计需求上出现了比较严重的矛盾；

- 硬件工程师对物理内存的设计需求
  - 内存需要被分割成足够小的 块 ， 且 没有段 的概念；
    这个块被称为内存的“页”，这个分割过程也就是“分页”的过程，Linux 将每页定义为 4K 大小；
    为什么需要把一个程序分割成这么小的页呢？目的就是为了避免过大的内存碎片，提高内存的使用率；

    - 内存不是被某个进程独占的，分配给进程(程序)的块也是 不连续的 ，
      每个内存块可以分配给任意一个进程，因此分配给进程的内存块必然不是连续的，块与块之间是有间隙的；
- 软件工程师对内存的设计需求
    - 需要内存被程序 独占 ；
    - 需要将程序使用分 段 保存，且段内的内存地址必须是 连续的 ；
      通常一段程序包含的段有，代码段、数据段、堆段、栈段等

上述的各自需求是相互矛盾的，如果不调和这个矛盾，必然导致，当需要为一段程序分配一段可用的内存的时候，软件工程师必须撰写相应的代码去检测当前物理内存哪些内存块是可用的，然后才能分配内存，也就是说，软件工程师必须深入的去了解底层硬件当前的使用情况才能成功的分配一个可用的物理内存块；并且站在硬件工程师的角度，他只会关注如何提升内存的使用效率，如何减少内存碎片，压根不会考虑是否需要对物理内存分“段”，也就是说，在软件工程师眼中的这个非常重要的“段”的概念，在硬件工程师那里，压根没有；

两个核心用户，对同一个物理内存的需求截然不同，那么如何才能调和这种矛盾呢？那么，唯一的办法，就是设立一个中间件，来避免直接使用物理内存，并且通过这个中间件，可以达到软件工程师使用内存的目的，这个中间件就是 虚拟内存 ，这个过程就称为软件工程师与硬件工程师之间的 解耦 过程；

既然有了 虚拟内存 ，那么自然就需要某种映射关系来建立虚拟内存到物理内存之间的联系，于是 页表 的概念也就自然而然的产生了；

虚拟内存

虚拟内存是面向软件工程师的，软件工程师希望他所使用到的内存是被他 独占 的，分 段 的，且段内的内存块是 连续 的；操作系统会为程序分配一个由用户所独占的虚拟内存，虚拟内存不是真实存在的，它只是由操作系统分派的一些列的地址所构成的；虚拟内存由两部分地址组成， 逻辑地址 (Logic Address)和 线性地址 (Linear Address)组成，通过 段表 ，将 逻辑地址 转换为 线性地址 ，最终通过 页表 将 线性地址 转换为 物理地址 ；

虚拟内存并不是真实存在的，它是由一系列连续的逻辑地址、线性地址所表述而成的；

逻辑地址

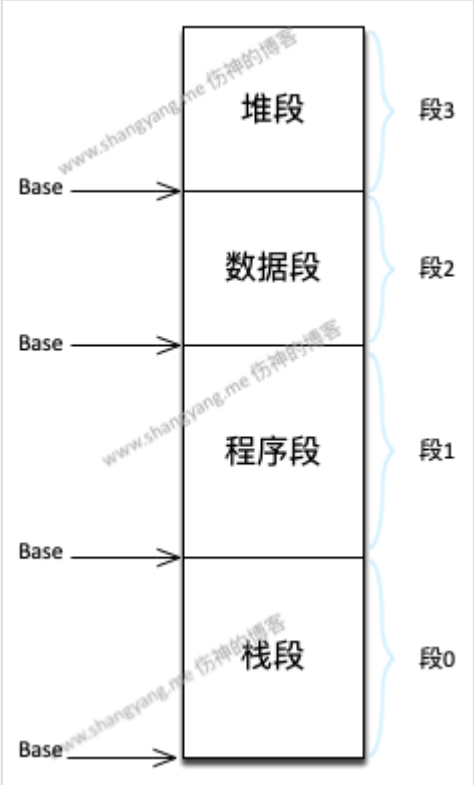
逻辑地址对 Program 而言，是被它独占的，而对于 32 位的操作系统而言，一个 Program 可以独享最多 3G 的逻辑地址空间，为什么不是 4G 呢？因为其中有 1G 逻辑地址由操作系统内核独享；逻辑地址由两部分组成 **Segment Selector** 和 **Offset**(偏移地址)，而 **Segment Selector** 指向 **Segment Descriptor**，一个 **Segment Descriptor** 完整的描述了一个 Segment 既段的信息，那么要明白逻辑地址是什么，因此，必须首先明白什么是段；

Segment

Segment 既段，只是操作系统对一个 Program 的虚拟内存从逻辑上的划分，通过不同的逻辑地址段来区分；

Segment

一个程序，在虚拟内存层面，通常被划分为程序段，数据段，堆段和栈段等，如图，



如图，这便是由 段 所虚拟出来的虚拟内存，该内存不是真实存在的；如何虚拟的呢？简而言之，就是为每个程序指令、数据赋予一个虚拟的内存地址，也就是虚拟地址(Linear Address)而实现的，最终通过虚拟地址 ↔ 物理地址的映射关系，将指令和数据映射到真实的内存物理地址上，完成虚实转换；

Base 表示基址，表示的是每一段所开始的起始地址，这样通过 Base + Offset，既可以得到该指令在虚拟内存中的地址，也就是虚拟地址(既 Linear Address)；

Segment Descriptor

段通过段描述(***Segment Descriptor***)保存段的信息，每一个 Segment Descriptor 作为一个 entry 保存到 GDT(全局段表)中；Segment Descriptor 由如下部分构成，

- Base

  长度 32 bits，Segment 在虚拟内存中的起始地址；
- Type

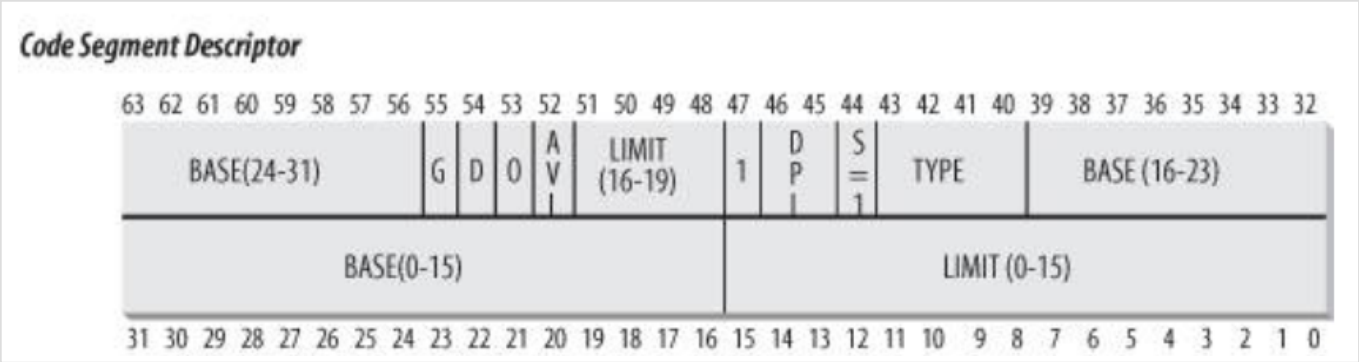
  长度 4 bits，表示访问权限，读、写、执行等权限；
- S

  长度 1 bit，系统标志位，表示该段是否是系统段，如果是，表示其中保存的是系统核心内容；

- D Or B

  表示该段包含的是 data 还是 code；

看一下 Code Segement Descriptor 的结构，

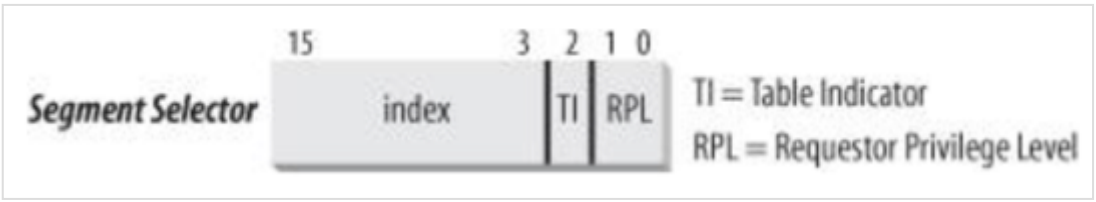


要注意的是，以 32 位操作系统为例，一个程序所有段使用的内存总和不能超过 3G，这是由页表的特性所决定的，这部分内容参考页表内容；

Segment Selector

Segment Selector 又称作 Segment Identifier 既段标识，它作为标识指向段表中该段所在的位置；

Segment Selector 总长 16 bits，由三部分组成，



- index

  index 是段描述（Segment Descriptor）在段表（GDT）中的偏移地址（直白点说，就是定位段在段表中的位置），长度 13 位；那么，如何计算某个段在段表中的具体位置呢？设 GDT 在内存中的其实地址为 $addr_g$，求某个段 S 在内存中的地址 $addr_s$，计算公式如下，设 index 为 i，

$$addr_s = addr_g + i \times 8$$

  为什么要将 index 乘以 8 呢？因为一个段描述符占用 8 个字节，因此这里要乘以 8；备注，GDT 在内存的起始地址存放在寄存器 GDTR 中的；

- TI

  Table Indicator；长度 1 bit，表示它指向的是 GDT 还是 LDT；

- RPL

  Request Privilege Level；长度 2 bits；指定 CPU 访问的权限级别；

  - 00 表示该段运行在内核态

- 11 表示该段运行在用户态

为了加速对 Selector 的访问速度，CPU 提供了六个寄存器来提高对 **Selector** 的访问速度，它们分别是

- cs
  Code Segment Register，代码段寄存器；它存储 Code Segment Selector；代码指令默认从这个段内读取；
- ds
  Data Segment Register，数据段寄存器；它存储 Data Segment Selector；
- ss
  Stack Segment Register，栈段寄存器；它存储 Stack Segment Selector；函数调用相关的压栈操作，push 和 pop，使用这个段的内存；
- es
  Extra Segment Register，额外的段寄存器；这个段的内存主要存储 String instructions，比如 `eos` 、 `movs` 等等；
- fs
- gs

最主要的就是 cs、ds 和 ss，程序运行时刻，CPU 计算出 Segment Selector 并将其保存到相应的寄存器中去；

Offset(偏移地址)

从虚拟内存的角度，程序拥有完整的内存空间，以 32 位操作系统为例，那么它可以任意的使用该 3G 的虚拟内存空间；因此，当某段代码开始执行，因为程序独占整个 3G 内存空间，因此 CPU 便可以从 0x00000000 开始为程序计算偏移地址，假设读取到第一个机器指令，main() 函数入口的机器指令，假设该指令长度为 2 Bytes，因此它的下一条指令的 Offset(偏移地址) 为 0x00000010，这便是下一条指令在虚拟内存中的入口地址；

> 机器指令是以 Byte 为最小单位，在 X86 指令架构下，指令长度在 1~13 Bytes 之间；

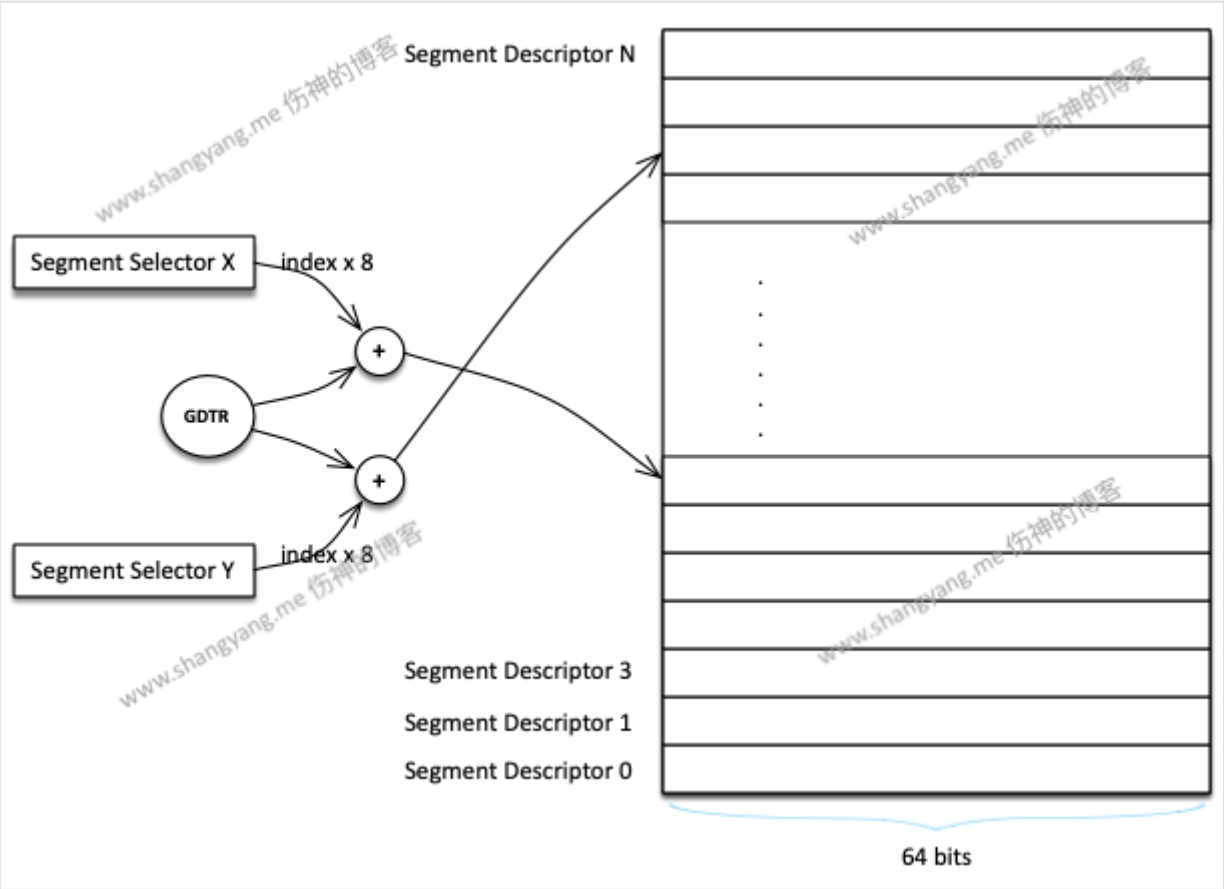
一个 Offset 的长度是 32 bits；

尤其要注意的是，偏移地址是指段内的偏移地址，所以，要得到完整的线性地址(虚拟地址)，就必须加上段基址；

GDT(段表)

Global Segment Descriptor Table，全局段表；它用来存储 Segment Descriptors；GDT 存储在主存中，在程序运行时刻，生成 Code Segment Descriptor，Data Segment Descriptor，Stack Segment

Descriptor...，然后，每一个 Descriptor 将会被作为一条记录保存到 GDT 中去，数据结构如下，



GDT 就是一张由 N 个 Segment Descriptor 所构成的表，GDTR 中通过 limit 限定了这张表的最大长度；通过 Segment Selector 中的 index × 8 再加上 GDTR 寄存器中 GDT 的内存地址可以唯一确定 GDT 中的一条 Segment Descriptor 记录（备注，这里为什么需要使用 index × 8 参考 Segment Selector 小节中有关 index 属性描述的部分）

备注：GDT 和 GDTR 都是在开机的时候由 BIOS 初始化的；

GDTR

段表寄存器，它由两部分组成，

1. Base

   长度 32 bits；GDT 在主存中的起始地址；
2. limit

   长度 16 bits；表示 GDT 中可以最多容纳多少个段；$2^{16} = 65536$ 也就是说，GDT 中最多可以容纳 65536 个段；引申，每个 Process 最多可以使用 6 个段，那么也就说，32 位操作系统最多可以创建 $65536/6 \approx 10922$ 个进程；

LDT

线性地址

Linear Address

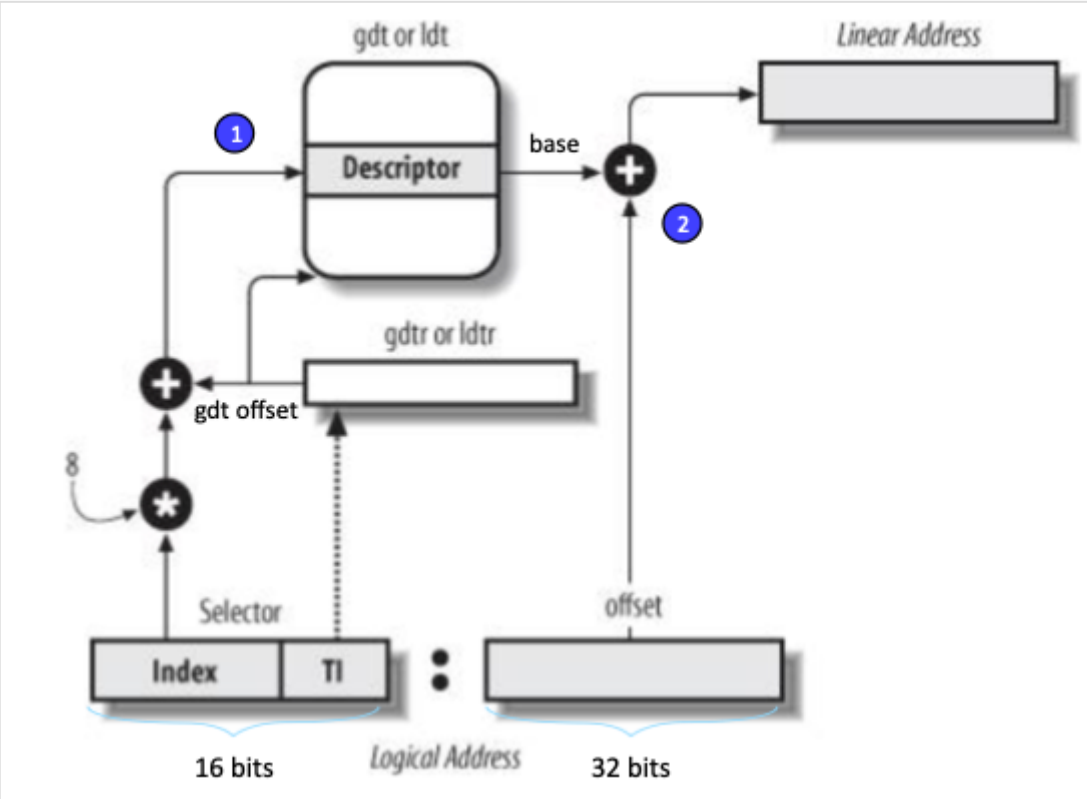
线性地址(Linear Address)，既常说的虚拟地址；线性地址由逻辑地址映射而来，物理地址由线性地址映射而来；

逻辑地址 → 线性地址

由逻辑地址转换而来，公式如下，

$$\text{Logic Address} = \text{Segment Base} + \text{Offset}$$

计算过程如下图所示，



主要就 2 个步骤，

1. 首先通过 Segment Selector 中的 index 和 GDT 在内存中的起始地址，找到该段(设为 β)在 GDT/LDT 中该段的位置；详细的计算过程参考段表小节；
2. 然后取 β 中的基址 Base，通过 Base + Offset 既得到线性地址；备注，Base 32 位，Offset 32 位，最终得到的线性地址为 32 位；

备注，该计算通过独立于 CPU 的芯片 Segment Unit 计算得来，因此不占用 CPU 的时间；

MMU

Segment Unit 和 Page Unit 统称为 MMU (Memory Mangement Unit)；

物理内存

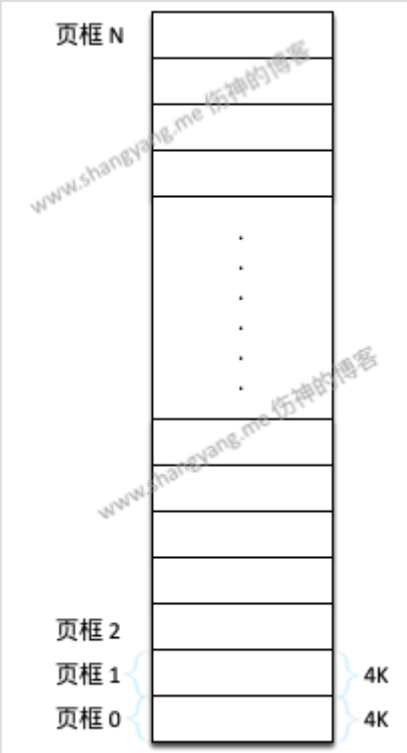
编址

物理内存编址的最小单位是 1 个字节，也就是说，1 个字节 对应一个物理编址；这里是有原因的，一是一个机器指令最小单位是 Byte，并且一个数据最小的存储单位也是 Byte；所以，为每个 Byte 单元才进行编址，顺理成章；

因此，我们有如下的对应关系，
32 位操作系统总共可以容纳 4G 个地址，每个地址指向一个 Byte 物理内存单元，因此所有的地址总共包含 $4G \times 1Byte = 4G\ Bytes$ 的内存，也就是 32G Bits 内存；

页框

Linux 系统是将整块物理内存按照 4K 大小进行划分的，如图所示



每一个 4K 单元被称为一个 页框 (Page Frame)，且每个页框都有自己的 页框编号 (Page Frame Number 简写为 PNF)，如图，页框编号是顺序递增的，且页框号是从 0 开始的计数的；

32 位操作系统最大的物理寻址空间是 4G，因此支持最大 4G 物理内存，假设，当前的物理内存为 4G，那么总共有 1M 个页框($\frac{4G}{4K} = 1M$)；

物理内存地址 $\iff$ 页框号 + 偏移地址

物理地址和页框号有如下的对应转换关系，

- 任一物理内存地址可以转换得到“页框号 + 偏移地址”，假设物理地址为 d，转换公式如下，

$$d/4K = n(余r)$$

  因为页框号(NFR)是从 0 开始计数，因此得到 NFR $= n - 1$，而 r 就是 偏移地址 ；要注意的是，操作系统并不会单独存储页框号与物理地址的映射关系，而是通过上述的除法根据物理地址动态计算出页框号的，该除法可以直接通过移位的方式进行，因此效率非常的高；
- 相反，任一 页框号(NFR) + 偏移地址(r)同样可以得到物理内存地址，计算公式如下，

$$(\text{NFR} + 1) \times 4K + r = d$$

虚拟内存 $\rightarrow$ 物理内存

虚拟内存映射为物理内存最重要的核心是，如何建立虚拟地址到物理地址之间的 映射关系 ； 而这层映射关系，主要由 页表 来来完成；

页表

映射原理

页表主要维护的是线性地址与物理地址之间的映射关系；那么这个映射怎么来完成的呢？假设我们有一个虚拟地址 V，使用同样的方法计算出虚拟地址 V 的页框号 V_{pfn} 和偏移地址 V_{offset}，假设我们有一个可用的物理内存页框号 P_{pfn}，那么这个映射关系就可以用下面这张表来描述了，

虚拟页框号	物理页框号	保护
V_{pfn}	P_{pfn}	保护位

这样通过将虚拟内存的页框号 V_{pfn} 就可以映射到物理内存的页框号 P_{pfn}，这样虚拟地址 V 通过这层映射关系，就映射为了物理地址 $P_{\text{pfn}} + V_{\text{offset}}$；

备注，由页框部分的分析可知，32 位操作系统最多有 1M (计算公式：$\frac{2^{32}}{4K} = 1M$) 个页框，因此页表中的页框号项需要 20 bits 正好代表 1M 个页框，保护位总共有 12 bits；所以，一个页表项总共 52 bits，头 40 bits 分别是虚拟页框和物理页框；

问题

假设，我们有一个 jump 指令；

```
1   JMP  L20 ; 假设当前指令的虚拟地址为 0x00000100
```

假设，需要跳转到的 L20 的地址为 0x02000000，0x00000100 对应的页框号是 0，而 0x02000000 对应的页框号是 8191，也就是说，地址需要从第 0 号页框直接跳到 8191 号页框；中间的 8190 个页框可能全部用不到，空的；那么我们就会想了，为了尽量的节省页表空间，我们可以这样来定义页表，

虚拟页框号	物理页框号	保护
0	X	保护位
8191	Y	保护位

然后，如果又有一个 jump 指令

```
1   JMP  L10 ; 假设 L10 指令的虚拟地址为 0x00020000
```

跳转到 L10，L10 对应的虚拟页框号为 32，因此，我们又在页表中添加了一项，

虚拟页框号	物理页框号	保护
0	X	保护位
8191	Y	保护位
…	…	保护位
32	Z	保护位

在我们需要的时候，才往页表中新增一条映射关系，这样，岂不是可以使得页表尽可能的小了？的确，从页表的占用空间上来说，这样做，页表的空间占用是最小化的；但问题是，它提升了查找的复杂性；假设，后续，我们需要在表中找一个页号为 600 的虚拟页框号，我们只能遍历查找，导致时间复杂度为 $O(n)$；这个极大的影响了 Page Unit 从也表中寻址的时间，从操作系统硬件层，这是绝对不能被接受的；

因此，从性能的角度上来说，如果不改动页表的结构，为了能够达到最快的查询速度，我们必须要在页表中做冗余；所以，对于该进程，页表的结构应该是这样的，

虚拟页框号	物理页框号	保护
0	X	保护位

虚拟页框号	物理页框号	保护
1	空	保护位
2	空	保护位
…	…	保护位
32	Z	保护位
…	…	保护位
512	Y	保护位
…	…	保护位
1024	…	保护位

为了能够使得查询速度为 $O(1)$，我们必须要维护一个完整的含 1M 页框的页表，因为，只有这样我们才可以直接通过一次位移操作，便可以直接定位到页表中的虚拟页框号；这样做，效率是最高的；但是，这样做的话，页表的体量就太大了，从虚拟内存的角度上来说，每个进程都可以独占 3G 的内存空间，也就说它总共拥有 0.75M 个虚拟页框号，页表中的每条记录占 32 bits，那么对于一个进程，一个页表就需要 24M 的内存空间，假设，我们有 100 个并发进程呢？那就需要 2.4 G 的内存空间了，显然，这是不可接受的；

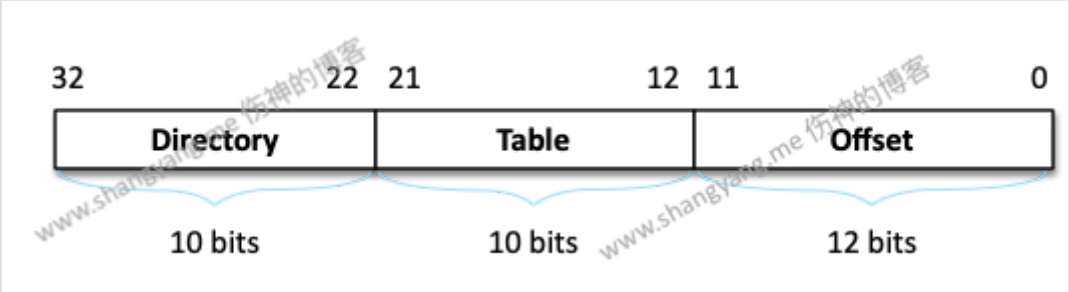
> 小记，让我突然想到了 Kafka 数据读取为什么这么快了，它的所有数据全是通过二进制的方式线性存储的，如果要取得某条数据，只需要通过 Offset 查找即可，因此 Kafka 查找数据的时间复杂度是 $O(1)$

由此可见，如果只使用一级页表，会导致空间和时间上的矛盾，两者无法调和；那么该如何处理呢？参看下一小节；

多级页表

划分线性地址

为了既能保障对页表的查询性能，又能保证对空间的占用足够小，操作系统的设计人员提出了多级页表的概念；首先，它将线性地址划分为如下三个部分，



天然形成这样一种对应关系，总共有 1024 个 Directories(目录)，每个 Directory 包含 1024 个页表，每个页表对应 4K 个偏移地址(这里这样做是有目的的，这样这好与一个物理页框的偏移地址范围相吻合)；那么总共有 $1024 \times 1024 \times 4K = 4G$ 的内存空间；

Offset

注意，逻辑地址中的 Offset 是 32 位的，而基于多级页表的线性地址的 offset 是 12 位的，为什么偏移由此转换呢？因此逻辑地址中 32 位的偏移地址是段内偏移地址，因此它的地址仍然是 32 位，而线性地址的偏移地址需要与物理页框的偏移地址对应起来，所以，需要设计为 12 位，否则虚拟地址和物理地址基于页框的偏移地址对应不起来，就得不到正确的转换；

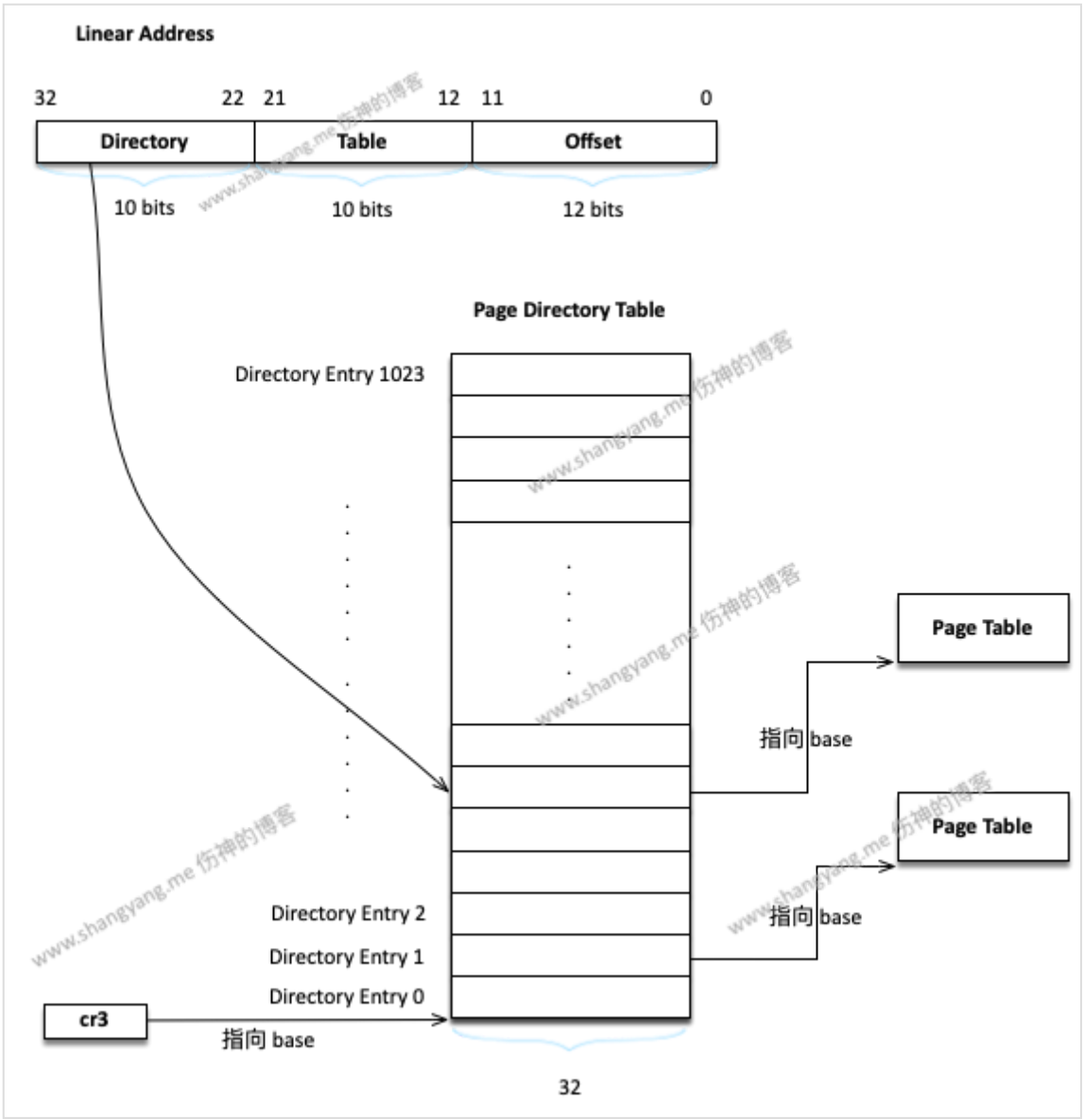
Page Directory Table

Page Directory Table 存放在主存中，控制寄存器 cr3(也称作 Page Directory Table Base Register) 保存着 Page Directory Table 在主存中的基址；Page Directory Table 存放着 1024 个 Page Directory Entries；

Page Directory Entry，32 位，高 20 位指向 Page Table 在主存中的起始地址；

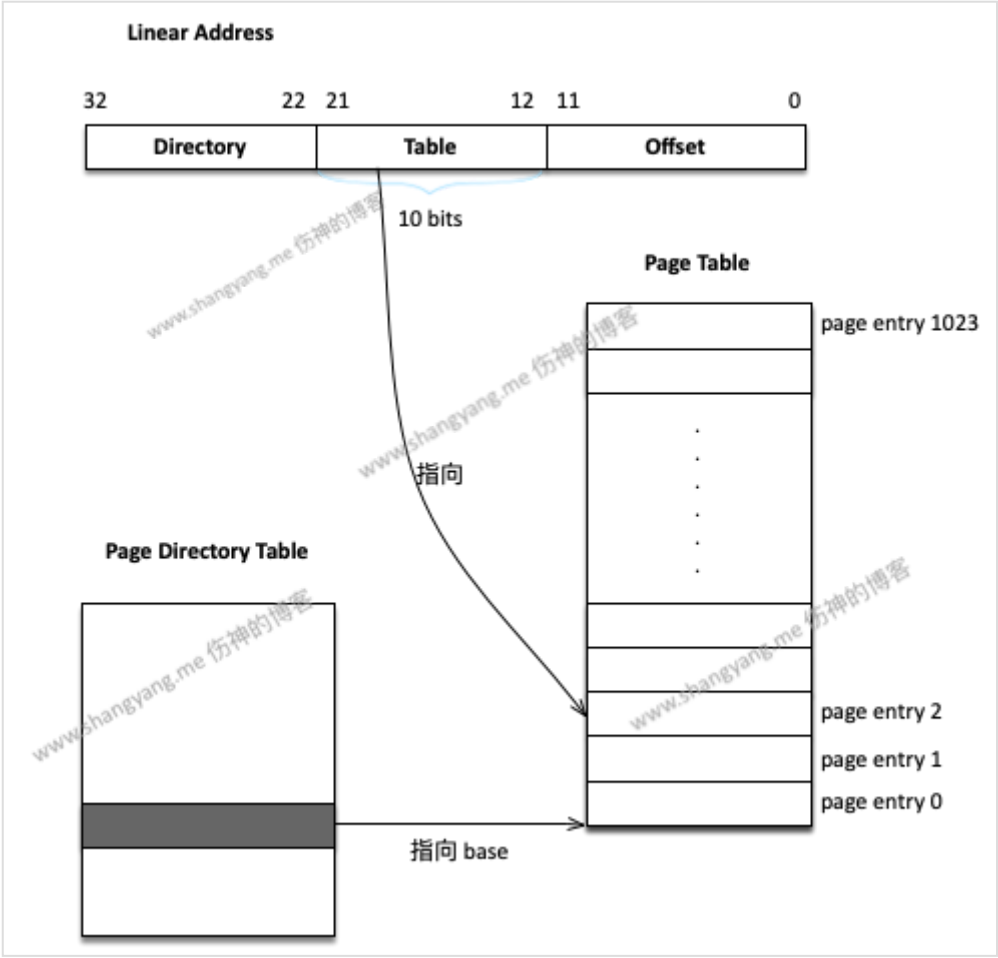
线性地址的高 10 位(既 Directory 部分)，指向的就是 *Page Directory Table* 中的某个 *Page Directory Entry*；

因此，由 *cr3* 和线性地址的 Directory 部分即可唯一确定 Page Directory Table 中的一个 Page Directory Entry；该逻辑可以由下图表示，



Page Table

Page Table 也是存储在主存中的；Directory Entry 的高 20 位指向 Page Table 在主存中的起始地址(Base Address)；线性地址 Table 部分的 10 bits 指向某个 Page Entry；如图所示，



页表的结构如下，

Bits	Meaning
31..12	Upper 20 bits of base address of Page (i.e., Frame Number)
11..9	Available for OS use (not used by MMU hardware)
8	Global Page (leave 0)
7	Page Size (0 = 4K)
6	Dirty (1 = Frame contents have been modified)
5	Accessed (1 = Frame has been accessed)
4	Cache Disabled (0)
3	Cache Policy (1= WriteThrough; 0=WriteBack)
2	User/Supervisor (0 = Supervisor and Page Table cannot be accessed in CPL3)
1	Read/Write (0 = Frame is Read-Only)
0	Present (1=present)

问题解决

再回到页表中所提到的问题，为了兼顾效率，我们必须缓存所有的页框号，这样当我们跳转到其它地址的时候，可以直接通过位移的方式，使得查询复杂度为 $O(1)$；多级页表只需要在主存中保存一个完整的 Page Directory Table，里面包含 1024 个 Directory Entries；通过这个方式，当需要

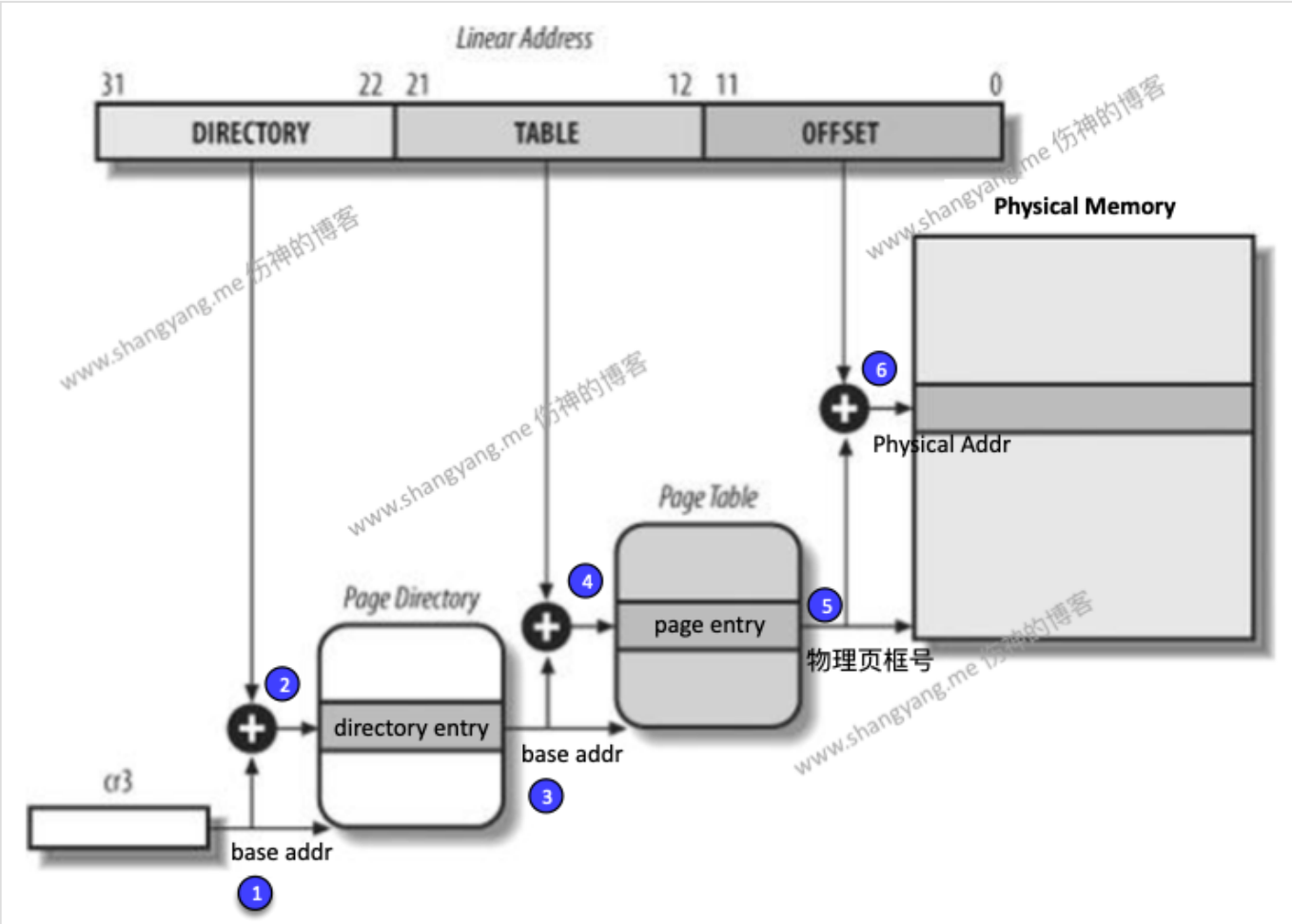
```
1   JMP  L20 ;
```

的时候，L20 的地址为 0x02000000，其二进制为 0000001000 0000000000 000000000000，由高 10 位可知，它对应的 Directory Entry 的编号为 8，因此，当从 0x00000000 跳转到 0x02000000 的时候，现在中间只需要冗余 6 个 Directory Entry 即可，而在但级页表中，我们需要冗余存储 8191 个页表项，因此，我们从 8190 个冗余项减少为 6 个冗余项，空间得到了极大程度的节省；不过要注意的是，单级页表定位一个 Page Entry 只要一次位移，也就是时间复杂度为 $O(1)$，而多级页表要定位一个 Page Entry 需要两次位移，首先需要位移从 Page Directory Table 中找到 Directory Entry，然后位移从 Page Table 中找到对应的 Page Entry，因此，以前的一次位移操作，现在总共需要三次位移操作了，也因此，通过多级页表定位一个 Page Entry 的时间复杂度为$O(2)$，依旧很快；

线性地址 ⟶ 物理地址

转换过程

转换过程可以由下图概括，



1. 通过控制寄存器 cr3 找到 Page Directory Table 在内存中的起始地址；
2. 通过线性地址中的 Directory 段从 Page Directory Table 中找到对应的 Directory Entry；
3. 通过 Directory Entry 的高 20 bits 找到 Page Table 在主存中的起始地址；

4. 通过线性地址中的 Table 段从 Page Table 中找到对应的 Page Entry；
5. 从 Page Entry 中通过虚拟页框号找到物理页框号 PFN_p；
6. 最后，通过 $\text{PFN}_p \times 4K + \text{OFFSET}$ = 物理内存地址；

上述的步骤由一个专门的控制单元 ***Page Unit*** 完成；

但是，其实在第 5 步，Page Entry 中的物理页框号在第一次映射的时候，还没有记录，这个时候，需要通过换入的方式，将物理页框号填充后，才能使用；这部分内容参看下一小节；

换入(Swap in)

回到虚拟地址 ⟶ 物理地址的第 5 步，第一次通过虚拟页框号查找物理页框号的时候，此时，没有对应的物理页框号；接下来，操作系统会通过如下步骤生成该物理页框号，

1. 首先 Page Unit 会发出一个 Page Fault Interrupt(INT 4)，缺页中断；
2. CPU 接收到中断信号，停下当前工作，然后立刻根据线性地址从 I/O Memory 或者硬盘中读取机器代码或者 data 的二进制数据，设为 α，单位是 Byte；
3. 操作系统通过 `get_free_page()` 方法找到当前空闲的物理内存块，将 α 放入，并返回相应的起始物理页框号；
4. Page Unit 将上述的物理页框号放入 Page Table 中，完成与虚拟页框号的对应关系；

归纳起来就是，发出中断，从磁盘中读入物理内存，取得物理页框号，最后与虚拟页框号建立起映射关系；

换出(Swap out)

这个故事发生在换入小节的第 3 步，通过 `get_free_page()` 方法去找空闲的内存块的时候，并不一定都能够找到，因为有可能已经没有多余的内存了；这个时候，就需要把内存中的物理页 换出 到硬盘中，怎么换呢？其实就是将 Page Entry 和物理内存中的数据一同置换到硬盘中去，这样方便随时换入进来；

TLB

Page Entry 是经常被访问的项，而 Page Entry 是被放在系统内存(System Memory)中的，因此，访问的效率会非常的慢；因此，在硬件层面，设计出了 TLB(Translation Lookaside Buffer)，缓存当前 Program 经常使用到的 Page Entries，下次当访问某个 Page Entry 的时候，先从 TLB 中取，这样对加速 Page Entry 的访问效率；

TLB 笔者映像当中只有几 KB的大小，那么为什么 TLB 会起到加速的作用呢？是因为程序代码中经常会有循环，比如 for 循环等等，因此 TLB 可以起到明显的提升作用；

要注意的是，当 Program 切换的时候，TLB 必须被清空，因为 TLB 是被某个 Program 独占的；通常是在新的 cr3 被加载的时候，清空 TLB；

3G → 1G

32 位操作系统的环境下，每个 Program 都可以独立拥有 3G 的虚拟内存，但通常我们只配备了 1G 的物理内存或者更小的内存，那么如何将 3G 的虚拟内存映射到只有 1G 的物理内存上呢？其实答案就在上述的换入和换出，当物理内存空间不够的时候，通过换出将物理页和相关的 Page Entry 序列化到硬盘中去，然后再为当前的 Program 进行换入操作即可；

A Program

独占

在 32 位操作系统中，每一个 Program 或者一个进程，都独享 3G 的虚拟地址(逻辑地址+线性地址)；那么，也就会独享与多级页表相关的映射关系，比如 Page Directory Table, Page Table 等等；

区分

每个 Program 都独占 3G 的内存空间，但虚拟地址空间、编址方式、起始地址都是一样的，那么在系统层面，我们如何区分多个 Program 呢？答案就在 Page Directory Register，cr3，它为每个 Program 保存了不同的 Page Directory Table 在内存中的起始地址，这样，就确保了每个不同的 Program 拥有完整的且不同于其它 Program 的 Directory Table → Directory Entry → Page Table → Page Entry 的链条，也就拥有了自己独一无二的虚拟内存→物理内存间的映射关系；

上下文切换

这里只讨论与操作系统内存管理相关的部分，一旦某个 Program 被换出，除了 PC 寄存器，中间结果的存放以外；与内存管理相关的步骤有，

1. 重新加载 cr3，将新的 Program 的 Page Directory Table 的在主存中的基址赋值给寄存器 cr3；这一部非常的关键，它确保了新切入的 Program 拥有自己完整的虚拟内存→物理内存的映射关系；
2. 清空 TLB；

操作系统

‹ 操作系统基础 01 – 起源(一) 图灵机

操作系统基础 – 内存管理(二) 虚拟地址是如何生成的 ›