《openEuler内核编程》

课程讲稿

第三章 第1讲

地址空间与寻址方式

软件所制

第三章 第1讲 地址空间与寻址方式

**学时：**2学时

**教学目的：**学习内存存储模型，了解内存、CPU缓存的作用。明确计算机中大端、小端的存储方法及区别。明白节点、管理区、物理页的功能和意义。了解计算机的寻址过程。

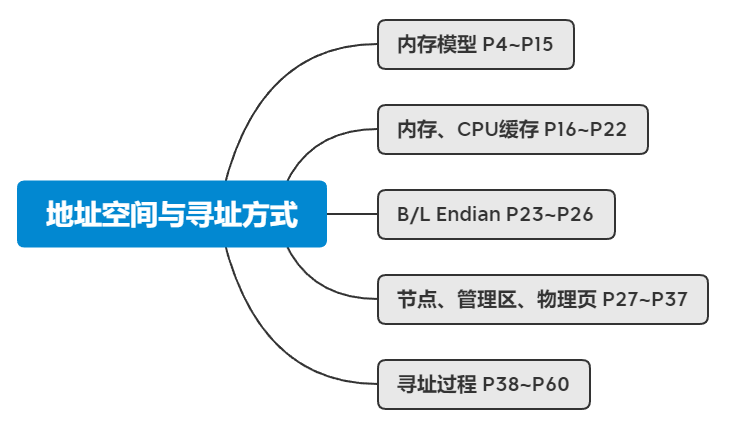
**课程时间线：**



**课外参考读物：**

《操作系统导论》

**知识框图：**

****

PPT讲稿：

4. 这一章内容我们来讲解内存管理。说到内存大家一定都不陌生。我们自己装机的时候都会去买内存条，早年的时候内存条容量非常小，一条128MB，装个256MB的内存还得插两条，叫双通道内存。现在单条内存16G的都属于是标配了。经常能看到一些高性能主机或者是服务器，配上8条32G的内存，一共几百G的内存，跑一些大型的服务器或者说是科研计算项目。都是非常给力的。前些年12306刚推出线上订火车票的时候，全国人民抢票的时候服务器压力是非常大的，因为车票信息更新速度非常快，需要第一时间响应，所以内存就需要非常大。当时我听说他们的服务器是128G的内存，我的第一个反应是什么呢，是不是自己听错了？因为那个年代，我电脑硬盘可能都没有128G，你和我说内存128G，我的概念是颠覆的。所以半天才反应过来，哦，原来内存可以这么大，一旦建立了这个概念，以后你再和我说内存1个T，2个T，我也觉得是很正常的事。学习内存管理这一章内容的时候，大家也会遇到一些颠覆自己常识的事情，先去接受它，然后慢慢理解它，就能学习的很快。
5. 再说一个内存相关的事情，内存上都是镶嵌很多内存颗粒，每个颗粒有一定的存储容量。比如单个颗粒512M，那我内存条上贴32个，就是一条16G的内存。过去这个东西常年被老外垄断，比如光韩国一个国家竟然占据了全球75%的市场份额，其中三星更是夸张的占据了45%的市场。他们那边一有点什么风吹草动，比如内存厂着火了，员工罢工了，新冠疫情了，全球的内存市场都跟着涨价。甚至有的时候你买一个内存条，过两年拿出来卖还能赚到钱。这样其实是非常不健康的一个状态。从技术上自力更生的角度上看更是如此。我们国家也一直在研制内存芯片，不是做不出来，而是工艺总是慢一步，做出来的东西不是比同行贵，就是比同行差，完全没有竞争力。但是在今年（2020），我们看到了希望。在2020年5月份，光威发布了第一款100%国产存储颗粒的内存条，弈PRO。售价是8GB 218元。在互联网上可以说是欢天喜地，大家都很兴奋，感觉挣脱了身上几十年的枷锁，无比的轻松。然后最可笑的是什么呢，对面的金士顿立刻把自己的同款内存条降价到215元，竟然是两年来的最低价。说明国外的这些厂商真的是感受到压力了，也侧面说明了我们国产内存正在崛起的事实。
6. 好，言归正传，我们讲讲内存管理。我们所编写的任何程序，最终都是要CPU来执行的，那么CPU执行的代码去哪里找呢？那就是内存了。我们的程序和数据平常都是放在磁盘上的，机械硬盘的读取速度非常慢，如果让CPU一句一句的从磁盘上读取，那会非常浪费时间，所以都会把程序和数据提前加载到运行速度更快的内存里。然后CPU从内存里读取数据，速度就非常快了。就像你去餐厅吃饭，肯定是厨师把饭做好端到你面前，你来吃，如果让你一遍一遍跑到后厨去吃，进去一次吃一口，每吃一口都折腾一遍，你肯定疯了。虽然现在发明了SSD固态硬盘，读取速度比机械硬盘快了许多，但和内存比还是差的很远。  
    我们计算机里有很多的程序，最大的一个程序就是操作系统，额外的我们上网用的浏览器，听歌的程序，玩游戏的程序、输入法程序。这些程序在运行的时候都需要提前加载到内存里。尤其现在的计算机都是多任务处理，势必在内存里需要存储大量的内容。但是我们也知道，内存一般来说比硬盘要小很多很多，一方面是内存的造价比较高，另一方面对于日常应用其实也用不了太大的内存，合适的就够用了。那么如何在有限的空间里管理好内存中驻扎的多个程序，如何让程序之间不打架，如何保证每一个程序的安全，如何让程序员写代码的时候更轻松。这些就都是内存管理需要考虑的事情。
7. 先举一个小例子，一个最朴素的内存模型。假设内存一个128K，然后0到64K分配给操作系统，剩下的部分分配各应用程序，完事了。最简单的内存管理。这也是最早期的操作系统对于内存的控制。程序员需要手动的关注真实的内存地址，写程序必须小心翼翼，一不小心碰到操作系统的部分，整个计算机系统就会崩溃。这个阶段一次只能运行一个程序。

早期的计算机都是非常贵的，所以在分时系统时代，就要求内存里要能存放多个程序。比如图中的这个例子。一共128K的内存，前64K内存是操作系统，后面全部是空白的区域。操作系统可以把不同的程序，更确切的说是进程，放在每一个区域里面，然后每个程序按照规则，来访问自己的那一段内存区域，这样程序互不干扰，实现了多进程驻留内存。那么这个设计有没有什么问题呢？大家想一下。其实这个阶段程序员还是要手动的指定物理内存位置的。尤其是每次我的程序不一定被加载到哪一块区域，每次我都要手动配置一遍物理内存地址。效率非常低下。同时由于每个程序都是直接读写物理内存地址，一个不小心就会碰到其他的进程或者操作系统，等于说前面的问题还是没解决。

1. 所以计算机科学家们非常希望能把程序员们解放出来，写程序的时候不要再去考虑哪一块内存能用哪一块不能用，我这一次的程序地址是多少，不小心覆盖了操作系统的内存怎么办。所以就提出了内存虚拟化的概念。我给你一个虚构的内存空间，这块内存空间非常完美，初始地址就是0，和操作系统一个待遇。一般来说我们叫它为“地址空间”。一个地址空间由以下几个部分组成（看图）。头部是代码段，用来加载你写的程序指令。再往下是堆段，这个就是系统自动，或者我们手动申请的内存空间，比如C语言里的malloc函数。每申请一块内存，这个堆段就会往下增加。这一段地址空间的底部是栈段，栈段用来存放一些局部变量、返回值、参数等等数据。它的新增方向和堆段相反，从高地址往低地址走。中间这一段就是未分配的空白部分。

这样一操作我们在写程序的时候就非常方便，不管写什么程序，对应的都是这样一段虚拟的地址空间，操作系统会自动把这一段虚拟的地址空间映射到物理内存上。这就是对内存的虚拟化。可以看做以对硬件的抽象。由于每个程序只能访问自己的那一段地址空间，这样不论你怎么操作，都不会对其他程序造成影响，进而实现了程序之间的隔离，对程序和操作系统提供了一种保护。那么问题来了，操作系统是如何实现内存的虚拟化的呢？

1. 好，那么揭晓这个问题之前大家先自己思考一下，你会怎么解决这个难题。…… 那么目前大多数的操作系统都采用了一种叫做地址转换的技术。前面我们说每一个进程都包含代码段、堆段、栈段三部分，占用的内存空间也是一定的。地址转换就是用一个基地址加偏移地址得到真实的物理地址。打比方说我是一个外卖小哥，我给你送餐的时候，肯定是先定位到你是哪个街道，那个小区，几号楼几单元、门牌号多少去送。这个小区就是基地址，几号楼就属于偏移地址。不然的话如果每一栋楼都有一个唯一编号，小哥一看，这一单送到22389号楼，另一单送到76443号楼，那他就疯了。
2. 还是拿一个具体的例子和大家说一下。首先是两个假设。1：用户的地址空间都是连续的存放在内存中的。2：用户的地址空间都不是很大。然后看这一段代码。简单的不能再简单，初始化一个变量，然后对这个变量加3。我们把这两行转换成汇编语言，一共是三行。第一行，初始化寄存器eax，第二行eax加3，第三行将eax内容存回内存。 借一张前面的内存图，但稍微细化一下。我们看图，第0~2k是程序代码，2k~4k是堆段，14k~16k是栈段。其余的区域都是空白。系统为这个进程一共分配了16k的空间。刚才那三行汇编代码，分别在地址128、132、135。如果执行这三条指令，从进程的角度来看，发生了以下几次内存访问。
   1. 从地址128获取指令
   2. 执行指令：从地址15k加载数据
   3. 从地址132获取指令
   4. 执行指令：加法运算
   5. 从地址135获取指令
   6. 执行指令：新值存入地址15k
   7. 从进程来看，所有的操作都是在内存0到16k的范围内发生的。
3. 但其实呢，这一部分地址空间是被操作系统重定位到一段真实的物理地址空间。比如从32k~48k的一段空间。实际上所有刚才所有的指令以及地址，都是需要在加上32k的。拿第一条指令为例。从地址32k+128获取指令，然后从地址32k+15k加载数据。这一套操作和我刚才讲的外卖小哥送外卖是一个意思。这种方法叫做动态重定位，也叫基址加界限，在上个世纪50年代的时候就已经有了。为了实现动态重定位，我们还需要额外的两个寄存器，一个用来存储起始地址，也就是所谓的基地址，另一个用来存储界限。有了这两个数据，我就可以把我的进程地址空间映射到物理内存中了。这个过程，就叫做地址转换技术。是不是非常简单，有的时候我们学习千万不要被各种高大上的术语给蒙住了，扒开来看其实都没什么。值得注意的是，动态重定位是完全基于硬件来实现的。当然早期也有软件实现的，就是程序在编译的时候，直接把基地址加上，然后程序最后是映射到物理地址，但是这种方法漏洞比较多，比如容易发生非法访问等等，在硬件重定位出现后，基本就消失在历史长河中了。我们的界限除了定位进程的地址空间外，还有一个作用，就是当我们的程序访问的内存空间超出界限后会报错，操作系统会把这样的进程直接挂掉。

说一点题外话，通过操作自己进程外的地址空间，可以做一些非法的事情，最常见的就是游戏外挂。比如我开着游戏，同时开着外挂，外挂修改游戏部分的内存，原来的一把枪的攻击力是100，改完之后变成10000，这在游戏里你就无敌了。有的同学可能会问，老师，你不是刚说过超过界限就会报错么，那外挂怎么还能控制别的内存空间？那是因为我现在讲的还是最早期的内存虚拟化技术，后来发展了几十年，技术越来越复杂，内存管理系统越来越庞大，紧接着就会带来各种各样的漏洞，难免会被利用。

1. 前面的内存管理方法其实存在一个比较大的问题，就是每个进程占用的内存中，有一大块是空白的。程序小的情况还好，一旦程序规模扩大，这种情况会更严重，甚至会出现单一程序占用系统大量内存却不使用，而其他程序却无内存可用的情况。于是在基址+界限的基础上发展出了分段。前面我一直在说每个进程可以分为三个段：代码段、堆段、栈段。分段的思想就是把这三个段分别放到物理内存中，把我原来整合到一起的、独立的进程地址空间打散，放到内存中。这样一来可以有效的解决大块内存的占用问题。之前一个进程配一组基址+界限寄存器。现在需要每个段一组，一共就是三组。我们看一下这个表格，简单的整理了当前这个进程在内存中的位置。虽然多占用了几个寄存器，但是这样一来进程在内存中的布置就可以非常的灵活。占用的空间也减少了。那针对一个进程，系统是如何知道我要操作的是地址属于哪个段呢？最常用的方法，是在地址前面加上一到两位的段标志。比如14位的虚拟地址，前两位代表段，后面的12位是代表偏移量。比如00代表的是代码段，01代表的是堆段，11代表的是栈段。
2. 直接上代码，让大家感受一下。第一行，获取虚拟地址的段，也就是前两位。第二行获取后面的偏移量。第三行这里有一个Bounds变量，就是我们前面看到的段寄存器里的大小值。获取完以后判断一下，要是当前的偏移值越界了，直接报错，没问题再往下。看第6行，有一个Base变量，对应的就是列表里的基地址，参数是当前的段，加上偏移，就是真实物理地址了。最后一行，调用函数，访问内存。这种方法就叫做显式方式。我们注意到虚拟地址的前两位都给用来表示当前段了，两位可以表示最多4个不同的值，但一个进程只有3个段，浪费了一个，所以有的系统索性就只用一位，0表示代码段，1表示堆段和栈段，那在内存里堆段和栈段就要紧密的排列在一起了。还有一种叫做隐式方式，就是系统根据当前虚拟地址的生成方式来判断属于哪个段，这样一来前面的标志位直接就省掉了。
3. 进程的三个段中，有一个段比较特殊，那就是栈段，它的增长方向是从高地址向低地址走的，刚好相反。所以我们在段寄存器中增加一个变量，表示增长方向。1表示从小到大，0表示从大到小。后来人们发现，如果能不同的段设置不同的权限，那代码写起来会更安全，效率也会更高。于是又增加了一个保护位，可以实现读、写、执行等不同权限的分割。这样段寄存器就一共有四种不同的参数了。那么写代码的时候需要检查的地方也多了起来。
5. 在计算机中，CPU要从内存中读取数据，但两者并不是直接相连的。内存和CPU都连接在北桥芯片上。北桥芯片上存在着内存控制器，内存控制器的类型决定了内存的类型。这种架构有一个问题，所有的CPU访问内存都是通过北桥芯片，同一时间只能有一个CPU访问RAM。实现不了多CPU的同时访问。
6. 一种解决思路是把内存控制器从北桥芯片中剥离出来，直接与内存相连，同时连着北桥芯片。如此两个CPU就可以独立的访问不同的内存。互不干扰。但是北桥芯片就会面临很大的压力，因为芯片的带宽是有限的。
7. 还有一种更加先进的设计思路是把内存控制器设计到CPU中，不再经过北桥芯片。每个CPU都有一个本地的内存，单个CPU访问自己的本地内存的时候比访问其他内存的速度要很多。同时北桥芯片也被解放出来，不用去承担内存读取的工作。这个设计架构叫做“非一致性内存访问架构”简称NUMA。
8. 内存分类两种类型，一类是静态RAM，简称SRAM，一类是动态RAM，简称DRAM。其中SRAM的速度更快，造价高，一般用作CPU的缓存。DRAM速度慢一些，使用的时候需要动态的刷新，不然就会丢失数据，但是价格便宜，容易大规模制造，一般我们的计算机内存，就是DRAM。
9. CPU缓存，即高速缓冲存储器，存在于内存和CPU之间。速度高于内存，接近CPU的运行速度。CPU在读取内存前，回去缓存中先尝试读取，如果没读取到的话就去内存中读取，然后把读到的内容放到缓存里，后续如果这部分数据需要反复访问，就可以加快系统的速度。
10. 实际上在CPU和内存之中不只有一种缓存。不同速度的缓存也分为几个级别，一级、二级、三级缓存。CPU读取数据会从123级的缓存去逐级的读取数据，最后才是从内存读取。不同级别的缓存的速度也不一样，离CPU越近的速度越快。

缓存呢也分三种，一种是指令缓存，一种是数据缓存，一种是TLB缓存，后面我们会学到，是页表的地址缓存。

2. 我们知道所有运行时候的数据都是需要存放在内存中的，那么对于比较大的数据，比如一个很大的数字，比如十六进制数0x12345678存储的时候会遇到一个问题，就是计算机内部都是从小到大来读取的，比如内存的顺序读取就是由低向高增长。但是人类自然的读数字都是从高位到低位的。所以在数据存储的问题上，有两派解决方案，一派是按照计算机的自然存储方法，即使很大的数据，也按照由低到高的顺序拆分开存储，叫小端模式。另一派是把数据按照反过来存储，低位的数据放到高位存储，高位的数据放到低位存储，叫大端模式。
3. 举一个例子，我国官方公布的人口数是14亿零5万人，这是一个非常大的数字，那这个数字在计算机中是怎么存储的呢？首先要转换成16进制，就是53731150。计算机中按照字节存储，所以需要四个字节来存储中国的人口数。我这里拆分开，并且用深浅不一的颜色来区分高低数据。深色是低位数据，浅色为高位数据。然后再来看一下大端小端分别是怎么存储这组数据的。假设我们内存地址从4000开始。小端就是从深色的数据开始存，大端就是从浅色的数据开始存。没有谁对谁错，只要知道你的系统是大端还是小端，然后按照对应的规则去读取数据就没问题了。
4. 那我们如何判断一台机器是大端还是小端呢。我们看下面一段代码，这里利用了联合体的特性。联合体的内部变量共享一块内存，我们声明两个变量，32位的int变量number和8位的字符变量。把一个超大的数赋值给number，然后4字节的内存就已经占满了。这时候再去读字符变量s。读取到的就是最低位的内存。通过判断这个位置中存储的值，就可以判断当前极其是大端还是小端了。

union{

int number;

char s;

}test;

bool testBigEndian(){

test.number=0x53731150;

return (test.s==0x01);

}

2. 这里我们介绍一下NUMA，非统一内存访问模型。在NUMA模型中，系统的物理内存会被划分为几个节点，nodes。我们需要了解的是，在一个单独的节点内，任意一个CPU的访问物理内存页面所需要的时间是一样的。但是不同CPU访问同一个节点的时间却有不同。这就造成了访问速度差异，所以对于不同的CPU来说，就有所谓“本地”和“远程”节点之分。本地的访问速度快，远程的内存访问速度稍慢。因此，可以通过设置NUMA来影响CPU访问Node的方式，包括均衡方式访问或本地访问的方式。在Linux里，每个Node以结构体pg\_data\_t来表示，并用pgdat\_list这个链表来维护。
3. 每个节点可以进一步划分为多个管理区，成为Zone。主要有三类管理器，包括DMA、Normal和HIGHEM三类Zone。其中DMA区域包含的页可以用来执行远程内存访问DMA操作。Normal包含正常映射的页。HIGHEM则包含高端内存，其中的页不能永久的映射到内核地址空间。
4. 每个管理区包含的是以4K大小的物理页面。每个物理页由page这个结构体描述。这些page由全局数组mem\_,ap来维护。
5. 这张图给出了节点、物理区和物理页之间的关系。大家可以看到，最底层叶子结点即物理页，向上组成Zone，不同的zone构成一个node节点。
6. 我们来看节点的数据结构pglist\_data,其中，node\_zones代表这个节点包含的管理区。Nr\_zones标识管理区的个数。Node\_mem\_map这个指针指向的是该节点的第一个物理页。其他的变量包含了节点的起始物理地址，物理页总数等信息。
7. 这个是管理区数据结构里包含的变量，Free\_pages表示了管理区空闲物理页的树木。Wait\_table指针标示等待物理页被释放的进程队列。
8. 我们可以看到，管理区zone的数据结构中包含三个水印，pages\_min, pages\_low, pages\_hoigh。当管理区内可用的物理页数目减少到pages\_low时，守护进程kswapd被唤醒，开始释放管理区内的物理页。当可用物理页数目减少到pages\_min时，kswapd将使用同步模式来释放物理页。当可用物理页数目达到pages\_high时，kswapd开始睡眠
9. 这张图给出了操作系统对于zone内空闲物理页数目不同阶段下kswapd守护进程的工作职能。
10. 我们来看物理页数据结构page，其中list标示了该物理页所在链表的链表头。Count为该物理页的引用计数。当count=0时，该物理页可以被释放。Flags是物理页状态的标示位。
11. 在Linux中，flags主要包括以下几种情况。PG\_dirty标示该页是脏页，被写过数据，但这些数据还未被写到磁盘上。PG\_highmem标示该物理页在高端地址，PG\_LOCK标示该页被上锁。PG\_LRU和PR\_REFERENCED标示该页被用于页面替换算法中，PG\_SLAB标示该物理页被用于SLAB分配器中，PG\_uptodata标示该页中数据从磁盘中读取后，该位被设定。
13. 接下来我们介绍操作系统中几个关于地址的概念。逻辑地址、线性地址和物理地址。其中，逻辑地址指的是机器语言指令中，用来指定一个操作数或者是一条指令的地址，每个逻辑地址由一个段和偏移量组成，偏移量指明了从段开始的地方到实际地址之间的距离。线性地址在32位系统中，是一个32位无符号整数，可以表示4GB地址空间，线性地址通常用16进制数字表示，值的范围从0x00000000到0xffffffff。物理地址用于内存芯片级的单元寻址，与从cpu的地址引脚发送到内存总线上的电信号相对应，物理地址由32位无符号整数表示
14. 逻辑转换的过程如下图所示，西安通过分段单元，亦即内存管理单元MMU转化为线性地址，再通过分页单元转化为物理地址。
15. 从80286开始，cpu开始使用两种不同的方式来执行地址转换，即实模式 和保护模式。操作系统启动后，只有实模式，然后通过打开A20，置cr0的PE位，跳转进入保护模式
16. 处理器是 按照字节编址的，因此内存是按照字节顺序进行组织和访问的。在访问一个或多个字节时，用字节地址对它们在内存中的位置进行定位。处理器也支持分段寻址，这种寻址方式是指程序可以有多个独立的地址空间，叫作段。比如一个程序可以分别把它的指令和数据保存在指令段和数据段中。这些段的起始地址被保存在段寄存器中。其中cs寄存器为 代码段寄存器，指向代码段；ss 为 堆栈寄存器，指向堆栈段；ds为 数据段寄存器，指向数据段。访问段中的地址的方式为段寄存器（段基地址）加上段内偏移。
17. 在实模式下，16位段寄存器只记录段基址的高16位，因此段基址必须4位对齐(末4位为0)。不采用虚拟地址空间，直接采用物理地址空间。物理地址=段寄存器\*16+段内偏移
18. 而当从实模式跳转为保护模式后，段寄存器中存储的不再是段基地址，而是段选择符（Segment selector）。因此逻辑地址为 段选择符+偏移量。段选择符为16位，不直接指向段，而是指向定义段的段描述符。段描述符存储在段描述符表中。段选择符包括索引号，表格指示器和请求特权级。其中索引号是段描述符表中的某一个段描述符。表格指示器标示使用哪一个描述符表。请求特权级则标示其ring 0到ring 3.
19. 段描述符存放在段描述符表中。操作系统中有两种段描述符表：GDT和LDT。其中GDT：global descriptor table, 是全局描述符。LDT：local descriptor table, 是局部描述符表通常只有一个GDT，每个进程除了存放在GDT中的段的时候，就需要创建附加的段，附加的段对应的段描述符就存放在LDT中。GDT在内存中的地址和大小存放在gdtr控制寄存器。LDT在内存中的地址和大小存放在ldtr控制寄存器中
20. 段描述符表是一个数组，长度可变，最多包括8192（2^13）个8字节的描述符。图中给出了GDT和LDT之间的关系。
21. 段描述符是GDT/LDT中的一个数据结构，为处理器提供诸如段基地址、段大小、访问权限及状态等信息。段描述符是8个字节大小，64位。
22. 其中段界限域指定段的大小。基址域确定段的第一个字节（字节0）在4GB线性地址空间中的位置。处理器将3个基地址域组合在一起构成了一个32位的地址值。类型域指明段的类型。包括应用程序代码段描述符，应用程序数据段描述符，系统描述符。S确定描述符时系统描述符（S=0）还是应用程序描述符(S=1)。P表明段当前是否在内存中。DPL：表明段的特权级，用来控制对段的访问
23. 当段描述符中的S标志位为1时，描述符为代码段或数据段描述符。不同的段描述符，类型域的编码不一样。类型域的最高位将决定该描述符为数据段描述符（为0）还是代码段描述符（为1）。数据段描述符的类型域中的低三位分别被解释为访问控制（A），是否可写（W），扩展方向（E）。代码段描述符的类型域中的低三位分别被解释为访问位（A），可读位（R），一致位（C）。
24. 这张表标示了代码段和数据段的不同类型域。指明了其代表的代码段还是数据段，以及其访问控制类型。
25. 当段描述符的S标志为0时，描述符为系统描述符。一般包括系统段描述符和门描述符。系统段描述符指向系统段门描述符或者持有放置在代码段中的过程入口点的指针，或者持有TSS的段选择子（任务门描述符）
26. 这张图给出了逻辑地址到线性地址的转换过程。逻辑地址通过段选择子在段描述附表中找到对应的段描述符，再通过段内偏移定位得到的地址，即为线性地址。
27. 当程序引用内存中的一个逻辑地址时，处理器首先将它转换为线性地址，然后使用分页机制将线性地址转换为相应的物理地址。在分页机制中，处理器将线性地址空间划分成固定尺寸的大小（4KB, 2MB或4MB）。这些页可以被映射到物理地址空间中。（物理页）。当包含线性地址的页不在物理内存中，就会产生一个缺页中断（page fault）.缺页中断会产生一个页故障异常，该异常对应一个处理程序，该程序指导OS将相应的页从磁盘上调入内存。页被调入内存后，就重新执行导致页故障异常的指令。
28. 处理器用来映射线性地址到物理地址的信息和产生页故障异常的信息存放在页目录表和页表中，页目录表和页表都存放在内存中。分页由控制寄存器中的标志位来控制。CR0控制寄存器的第31位时PG(分页标志)，该标志位置1时，表明要使用分页机制。
29. 一个由32位页表项（PTE）组成的数组, 放置在一个4KB的页中。一张页表最多包括1024个页表项。每个页表项指向一个4KB大小的物理页。那么一张页表可以索引多少大小的物理内存？（1024\*4KB）一张页目录表呢？（1024\*1024\*4KB）
30. 这张图给出了线性地址转换为物理地址的过程。通过页目录表查找到页表，再通过页表中的页表项定位到具体的物理页，再通过页内偏移，找到指定的物理内存地址。
31. 从上图可以看出，线性地址分成三部分。22到31位，为页目录表项在页目录表中的偏移, 提供所指页表的物理基地址。12位到2位1，为页表项在所选页表中的偏移，提供所选物理内存页的物理基地。0到11位 , 提供地址在所选物理页中的偏移。
32. 我们上述讲的都是主要以4K为大小的，所以其页内偏移需要12位来标示。但是当物理页为4M呢？我们就需要22位来标示页内偏移。这就是图里pentirum处理器进行扩展分页机制时，线性地址到物理地址的转换，里可以看到，页表项用了10位，而偏移量用了22位表证。
33. Linux中，同时采用了适用于32位和64位系统的普通分页模。Linux采用3级分页模式。包括页全局目录(Page Global Directory)，页中间目录(Page Middle Directory)和页表(Page Table)
34. 图为Linux中的三级分页机制。我们可以看到系统通过CR3寄存器找到对应全局页目录表，再根据线性地址，找到该表项，即而定位到对应的页中间目录表项和页表项，并通过页内偏移，找到对应的物理地址。