- 简单的解决方案: 更大的页
  - 一种简单的方法减小页表大小: 使用更大的页
  - 32位地址空间, 页大小修改为16KB(2^14),18bit的 VPN
    2^18x4Byte = 1MByte, 相对于4KB的页, 页表缩小到1/4
  - 许多体系结构都支持多种页大小,一个"聪明的"应用程序则可以为地址空间的特定部分使用一个大型页(如4MB),
    从而让这些应用程序可以将常用的大型的数据放入这样的空间,同时只占用一个 TLB 项.
    这种类型的大页在数据库管理系统中很常见, 然而多种页面大小的主要原因并不是为了节省页表空间,
    而是为了减少TLB的压力,让程序能够访问更多的地址空间而不会遭受太多的TLB未命中
  - 大内存页的主要问题在于: 会导致页内的浪费, 即内部碎片问题.
    应用程序会分配页, 但只用每页的一小部分, 而内存很快就会充满这些过大的页.
    因此大多数系统常见的情况下使用较小的页: 4KB或8KB
- 混合方法: 分页和分段
  - 将分页和分段相结合,以减少页表的内存开销
  - 假设一个地址空间,16KB 的小地址空间和 1KB的页,堆和栈的使用部分很小
  - 代码页 VPN0 映射到 PFN10, 堆页 VPN4映射到 PFN23
    地址空间另一端的两个栈页 VPN14和15被分别映射到 PFN28 和 4
  - 从图 20.1 中可以看到,大部分页表都没有使用,充满了无效的项.
    这是一个微小的 16KB 地址空间,想象一下 32位地址空间的页表和所有潜在的浪费空间
  - 杂合方法不是为进程的整个地址空间提供单个页表,而是为每个逻辑分段提供一个.
    在这个例子中, 我们有 3个页表: 地址空间的代码、堆和栈部分各有一个
  - 分段中,有一个基址(base)寄存器,告诉每个段在物理内存中的位置.
    还有一个界限(bound)或限制(limit)寄存器, 告诉该段的大小
  - 在杂合方案中,仍然在MMU中拥有这些结构,但是基址不是指向段本身,而是保存该
    段的页表的物理地址.界限寄存器用于指示页表的结尾(即它有多少有效页)
  - 假设 32 位虚拟地址空间, 4KB页面, 3个段:代码、堆、栈.
    用最高位2个bit表示段, 00(未使用), 01(代码段),10(堆),11(栈), 则虚拟地址表示为:

    

  - 假设硬件中有 3个基址/界限对,代码、堆和栈各一对.当进程运行时,每个段的基址寄存器都包含该段的线性页表的物理地址.
    因此系统中每个进程现在都有 3个与其关联的页表.在上下文切换时,必须更改这些寄存器,以反映新运行进程的页表的位置
  - 在 TLB未命中时(假设硬件处理 TLB 未命中),硬件使用分段位 (SN)来确定要用哪个基址和界对.
    然后硬件将其中的物理地址与 VPN 结合起来, 形成页表项(PTE)的地址,读取页表项中的 PFN,
    与Offset结合得到最终的物理地址.
  - 实现代码与18章线性页表查找过程非常相似,唯一的区别是使用 3个段基址寄存器中
    的一个来获取 PTE 的位置, 而不是单个页表基址寄存器 PTBR

    ```
    SN            = (VirtualAddress & SEG_MASK) >> SN_SHIFT
    VPN           = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
    AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
    ```

  - 杂合方案的关键区别在于每个分段都有界限寄存器,每个界限寄存器保存了段中最大有效页的值
    如果代码段使用前3个页(0,1,2),则代码段页表将只有3个项分配给它,且界限寄存器将被设置为3.
    内存访问超出段的末尾将产生一个异常,并可能导致进程终止.
    以这种方式,与线性页表相比,杂合方法实现了显著的内存节省.
    栈和堆之间未分配的页不再占用页表中的空间(仅将其标记为无效)
  - 杂合方案的问题. 首先它仍然要求使用分段,分段并不像我们需要的那样灵活,因为它假定地址空间有一定的使用模式.
    例如如果有一个大而稀疏的堆,仍然可能导致大量的页表浪费.
    其次杂合导致外部碎片再次出现,尽管大部分内存是以页面大小单位管理的,但页表现在可以是任意大小(PTE的倍数).
    因此在内存中为它们寻找自由空间更为复杂. 出于这些原因,人们继续寻找更好的方式来实现更小的页表.
- 分页: 较小的表
  - 基于数组的页表即线性页表太大, 占用太多内存.
  - 32位地址空间, 4KB的页,页表目4B,则页表需用4MB
  - 页表如何更小? 关键思路? 又会出现什么效率问题?
- 多级页表
  - 多级页表,不依赖于分段,但也试图解决相同的问题:
    如何去掉页表中的所有无效区域,而不是将它们全部保留在页表内存中
  - 多级页表的基本思想很简单.
    首先,将页表分成页大小的单元. 然后如果整页的页表项(PTE)无效,就完全不分配该页的页表.
    为了追踪页表的页是否有效(以及如果有效它在内存中的位置),使用页目录(page directory)这种新结构.
    页目录可以告诉你页表的页在哪里,或者页表的整个页是无效页.
  - 图 20.2 展示了一个例子.
    左边是经典的线性页表,即使地址空间的大部分中间区域无效,仍然需要为这些区域分配页表空间(页表中间两页),
    右侧是一个多级页表,页目录仅将页表的两页标记为有效(第一个和最后一个), 因此页表的这两页就驻留在内存中.

    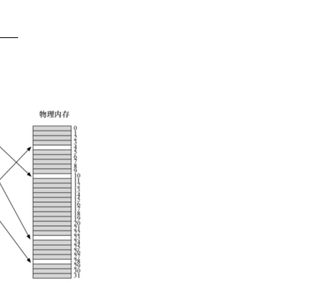

    图 20.1 1KB 的页和 16KB 的地址空间

  - 可以形象地看到多级页表的工作方式:
    它只是让线性页表的一部分消失(释放这些帧用于其他用途),
    并用页目录来记录页表的哪些页被分配

    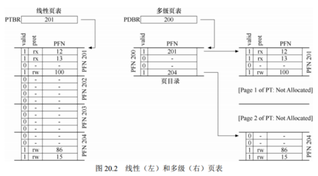

    图 20.2 线性（左）和多级（右）页表

  - 一个简单的两级页表中,页目录为每页页表包含了一项, 它由多个页目录项(Page Directory Entries,PDE)组成
    PDE(至少)拥有有效位(valid bit)和页帧号(page frame number,PFN),类似于 PTE.
    但是有效位的含义不同: 如果 PDE项是有效的, 则该项指向的页表(通过 PFN)中至少有一页是有效的,
    即在该 PDE所指向的页中,至少一个 PTE,其有效位被设置为 1.
    如果 PDE项无效, 则 PDE的其余部分没有定义.
  - 一些明显的优势
    - 多级页表分配的页表空间, 与正在使用的地址空间内存量成比例.
      因此它通常很紧凑,并且支持稀疏的地址空间
    - 如果仔细构建,页表的每个部分都可以整齐地放入一页中,从而更容易管理内存.
      操作系统可以在需要分配或增长页表时简单地获取下一个空闲页,将它与一个
      简单的线性页表相比,后者仅是按 VPN索引的 PTE数组.
    - 用这样的结构,整个线性页表必须连续驻留在物理内存中.对于一个大的页表(比如4MB).
      找到如此大量的、未使用的连续空闲物理内存,可能是一个相当大的挑战
    - 有了多级结构,我们增加了一个间接层(level of indirection),使用了页目录,它指向页表的各个部分
      这种间接方式,让我们能够将页表页放在物理内存的任何地方
  - 缺点
    - 在 TLB未命中时,需要从内存加载两次才能从页表中获取正确的地址转换信息.
      一次用于页目录,另一次用于 PTE本身,而用线性页表只需要一次加载
    - 多级表是一个时间—空间折中(time-space trade-off)的例子,我们得到了更小的表,但是有代价.
      尽管在常见情况下(TLB 命中),性能显然是相同的,但 TLB未命中时,有较高的成本

复杂性. 无论是硬件还是操作系统来处理页表查找(在TLB未命中时),无疑都比简单的线性页表查找更复杂.
通常我们愿意增加复杂性以提高性能或降低管理费用.
在多级表的情况下,为了节省宝贵的内存,我们使页表查找更加复杂

详细的
多级页表示例

假定一个大小为 16KB的地址空间,64Byte的页.如图20.3所示, 有256个虚拟页.
页0和1用于代码, 页4和5用于堆,页254和255用于栈

虚拟地址空间 14bit, Offset 占6Bit, VPN占8Bit.
如果是线性页表, 则页表项 2^8=256个, 页表大小 256x4Byte = 1KB.

构建一个两级页表, 鉴于我们有 64Byte的页,1KB页表可以分为16x64Byte的页,
每个页可以容纳16个PTE (16x16x4Byte)
如何获取 VPN,并用它来首先索引到页目录中,然后再索引到页表的页中.
每个都是一组项,因此需要弄清楚,如何为每个 VPN 构建索引.

首先索引到页目录. 这个例子中256个页表项,分布在16个页上.
页目录为页表的每页提供一个项.16个项则需要 4 位VPN来索引页目录

一、从VPN中提取页目录索引(PDIndex),通过计算找到页目录项(PDE)的地址:
PDEAddr = PageDirBase + (PDIndex×sizeof(PDE))

二、如果 PDE无效, 则访问无效,引发异常.
如果 PDE有效,则从PDE指向的页表中获取页表项(PTE)

三、要找到PTE, 则从 VPN中提取页表索引部分(PTIndex), 用来索引页表本身,
计算出PTE地址: PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))

使用二级页表,代入例子中一个实际的虚拟地址转换

从页目录开始(表 20.2左侧), 这个表是 16(PDE)x16(PTE)x4Bbyte的结构
在该表中,可以看到每个页目录项(PDE)都描述了有关地址空间页表的一些内容,
地址空间里有两个有效区域(开始和结束处),以及一些无效的映射

在物理页100(页表的第0页的物理帧号)中, 有1页包含16个页表项.
记录了地址空间中的前16个VPN,见表 20.2中间部分.
VPN0和1是有效的(代码段),4和5有效(堆). 因此,该表有这些页的映射信息,其余项标记为无效.

页表的另一个有效页在 PFN101中,该页包含地址空间的最后16个VPN的映射.
具体见表20.2右侧, VPN254和255(栈)包含有效的映射

从这个例子中可以看出,多级索引结构可以节省很多空间.
我们不是为一个线性页表分配完整的16页,而是分配3页:
一个用于页目录,两个用于页表的具有有效映射的块,在32位或
64位地址空间的节省显然要大得多

转换一个虚拟地址:0x3F80, 二进制 **11 1111 1000 0000**

一级页目录的第15个 PDE

二级页表的第14个 PTE, PFN=55

PFN = 55, offset = 000000,最终形成的物理地址:
PhysAddr = （ PTE.PFN << SHIFT） + offset = 00 1101 1100 0000 = 0x0DC0

超过两级

目前我们都是假定多级页表只有两个级别:一个页目录和几页页表.
在某些情况下,更深的树是可能的(并且确实需要)

我们构建多级页表的目标: 使页表的每一部分都能放入一个页.
到目前为止, 我们只考虑了页表本身. 但是如果页目录太大,该怎么办?

要确定多级表中需要多少级才能使页表的所有部分都能放入一页,首先要确定多少页表项可以放入一页.
假设我们有一个 30bit的虚拟地址空间和一个 512byte页. PTE占用4Byte.
则单个页上可以放入128个PTE,当索引页表时需要VPN的7个bit作为索引.

如果是两级页表, 则有14bit留给了页目录,页目录有 2^14 PDE = 2^18 Byte.
2^18/2^9 = 128个页,因此我们让多级页表的每一个部分放入一页目标失败了.

为了解决这个问题,将页目录继续拆分, 拆成多个页, 然后在其上添加
另一个页目录,指向页目录的页.我们可以按如下方式分割虚拟地址:

现在,当索引上层页目录时,使用虚拟地址的最高7位(PD索引0), 该索引用于从顶级页目录中获取页目录项.
如果有效,则通过组合来自顶级PDE的物理帧号和 VPN的下一部分(PD索引1来查阅页目录的第二级.
最后如果有效,则可以通过使用与第二级 PDE的地址组合的页表索引来形成 PTE 地址.
这会有很多工作,所有这些只是为了在多级页表中查找某些东西.

地址转换过程:
记住 TLB

图 20.4 总结使用两级页表的地址转换的整个过程
该图显示了每个内存引用在硬件中发生的情况(假设硬件管理的 TLB)

在任何复杂的多级页表访问发生之前,硬件首先检查 TLB. 命中时,物理地址直接形成;
未命中时, 硬件需要执行完整的多级查找.
在这条路径上,可以看到传统的两级页表的成本: 两次额外的内存访问来查找有效的转换映射

```
1   VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2   (Success, TlbEntry) = TLB_Lookup(VPN)
3   if (Success == True)   // TLB Hit
4       if (CanAccess(TlbEntry.ProtectBits) == True)
5           Offset   = VirtualAddress & OFFSET_MASK
6           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7           Register = AccessMemory(PhysAddr)
8       else
9           RaiseException(PROTECTION_FAULT)
10  else                   // TLB Miss
11      // first, get page directory entry
12      PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13      PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14      PDE     = AccessMemory(PDEAddr)
15      if (PDE.Valid == False)
16          RaiseException(SEGMENTATION_FAULT)
17      else
18          // PDE is valid: now fetch PTE from page table
19          PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20          PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
21          PTE     = AccessMemory(PTEAddr)
22          if (PTE.Valid == False)
23              RaiseException(SEGMENTATION_FAULT)
24          else if (CanAccess(PTE.ProtectBits) == False)
25              RaiseException(PROTECTION_FAULT)
26          else
27              TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28              RetryInstruction()
```

图 20.4 多级页表控制流

反向页表

在反向页表(inverted page table)中我们保留了一个页表,其中的项代表系统的每个物理页,
而不是有许多页表(系统的每个进程一个).
页表项告诉我们哪个进程正在使用此页,以及该进程的哪个虚拟页映射到此物理页

要找到正确的项,就是要搜索这个数据结构. 线性扫描是昂贵的,因此通常在
此基础结构上建立散列表,以加速查找. PowerPC 就是这种架构的一个例子

反向页表说明了我们从一开始就说过的内容: 页表只是数据结构.
你可以对数据结构做很多疯狂的事情,让它们更小或更大, 使它们变得更慢或更快.
多层和反向页表只是人们可以做的很多事情的两个例子

将页表交换到磁盘

到目前为止,我们一直假设页表位于内核拥有的物理内存中.
即使我们有很多技巧来减小页表的大小,但是它仍然有可能是太大而无法一次装入内存

因此一些系统将这样的页表放入内核虚拟内存(kernel virtual memory),
从而允许系统在内存压力较大时,将这些页表中的一部分交换(swap)到磁盘

小结

我们已经看到了如何构建真正的页表. 不一定只是线性数组,而是更复杂的数据结构.
这样的页表体现了时间和空间上的折中(表越大,TLB 未命中可以处理得更快,反之亦然).
因此结构的正确选择强烈依赖于给定环境的约束

在一个内存受限的系统中(像很多旧系统一样), 小结构是有意义的.
在具有较多内存,并且工作负载主动使用大量内存页的系统中,
用更大的页表来加速 TLB未命中处理,可能是正确的选择.

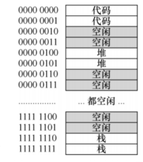

图 20.3 16KB 的地址空间和 64 字节的页

表 20.2 页目录和页表

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-