**页表**

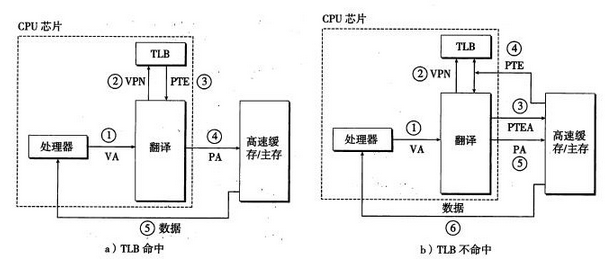
首先让我们来介绍一个概念-页表（page table）。页表就是存储在内存中的一张表，表中记录了将虚拟地址转换成物理地址的关键信息。MMU正是通过对页表进行查询，实现了地址之间的转换。也就是说，MMU每次工作的时候都要去查这张表，从中找出与虚拟地址相对应的物理地址，然后再进行数据存取。页表的作用如图3-3所示。

|  |
| --- |
| <http://images.51cto.com/files/uploadimg/20101221/132548734.jpg> |
| （点击查看大图）图3-3  页表的作用 |

页表中的条目被称为页表项（page table entry），一个页表项负责记录一段虚拟地址到物理地址的映射关系，稍后我们会详细介绍。

既然页表是存储在内存中的，那么程序每次完成一次内存读取时都至少会访问内存两次，相比于不使用MMU时的一次内存访问，效率被大大降低了，如果所使用的内存的性能比较差的话，这种效率的降低将会更明显。因此，如何在发挥MMU优势的同时使系统消耗尽量减小，就成为了一个亟待解决的问题。

于是，TLB产生了。TLB是什么呢？我们叫它转换旁路缓冲器，它实际上是MMU中临时存放转换数据的一组重定位寄存器。既然TLB本质上是一组寄存器，那么不难理解，相比于访问内存中的页表，访问TLB的速度要快很多。因此如果页表的内容全部存放于TLB中，就可以解决访问效率的问题了。



然而，由于制造成本等诸多限制，所有页表都存储在TLB中几乎是不可能的。这样一来，我们只能通过在有限容量的TLB中存储一部分最常用的页表，从而在一定程度上提高MMU的工作效率。

这一方法能够产生效果的理论依据叫做存储器访问的局部性原理。它的意思是说，程序在执行过程中访问与当前位置临近的代码的概率更高一些。因此，从理论上我们可以说，TLB中存储了当前时间段需要使用的大多数页表项，所以可以在很大程度上提高MMU的运行效率。

让我们接着聊页表。页表是由页表项组成的，每一个页表项都能够将一段虚拟地址空间映射到一段物理地址空间中。这里所谓的这段虚拟地址空间，更专业地讲，应该叫页，一个页对应了页表中的一项，页的大小通常是可选的。在ARM中，一个页可以被配置成1K、4K、64K或1M大小（ARM v6体系以后，不再支持1K大小的页），分别叫做微页、小页、大页和段页。页的大小决定了映射的粒度，是根据实际应用有选择地配置的。以1M为例，按照我们前面的描述，假设系统中将有64M内存需要被映射，那么我们一共需要64M/1M个页表项，而每个页表项需要占据4个字节，也就是说，有256字节的内存要专门负责地址映射，不能用于其他用途。

对于1K、4K和64K大小的页，MMU采用二级查表的方法，即首先由虚拟地址索引出第一张表的某一段内容，然后再根据这段内容搜索第二张表，最后才能确定物理地址。这里的第一张表，我们叫它一级页表，第二张表被称为是二级页表。采用二级查表法的主要目的是减小页表自身占据的内存空间，但缺点是进一步降低了内存的寻址效率。不同大小的页对查表方法的支持程度如表3-1所示。

表3-1  不同大小页的查表方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 页大小 | | | |
| 查表方法 | 1K | 4K | 64K | 1M |
| 一级查表 | 不支持 | 不支持 | 不支持 | 支持 |
| 二级查表 | 支持 | 支持 | 支持 | 不支持 |

下面，首先来研究一下相对简单的一级查表。

**一级查表**

一级查表只支持大小为1M的页。准确地讲，这里所谓的1M大小的页，应称为段（section）。此时，一级页表也被称为段页表。图3-4描述了段页表的内存分布情况和段页表项的具体格式。

|  |
| --- |
| <http://images.51cto.com/files/uploadimg/20101221/132708300.jpg> |
| （点击查看大图）图3-4  段页表项的结构 |

段页表中的每一项都类似于图3-4中的形式，其中：

（1）31~20位段表示物理地址的基地址，一共12位，也就是说，如果我们通过虚拟地址找到某一个段页表项，那么就可以确定这段虚拟地址所对应的物理地址的高12位了。因为段页表项后20位正好可以描述1M的内存，因此基地址每增加一个单位，物理地址就会增加1M空间。所以，我们也可以说该基地址表示了虚拟地址属于哪1M范围的物理地址。由此可知，使用段页表进行地址映射时，每一页能够描述1M的物理地址空间。进一步讲，段页表最多支持1024个页，最多占用系统4K字节的内存来存放页表。

（2）11~10位是AP位，区分了用户模式和特权模式对同一个页的不同访问权限。例如，当AP位为"11"时，表示任何模式下都可以对该空间进行读写，"10"则表示特权模式可读写该页，而用户模式只能读取该页。对AP位详细的描述请参考页权限一节。

（3）8~5位代表该页所属的域。在ARM体系结构中，系统中规定了16个域，因此使用4个位就能表示该页属于哪个域，而每一个域又有各自独立的访问权限，从而实现了初级的存储器保护。

（4）3位和2位分别代表cache和write buffer。相应的位为1则表示被映射的物理地址将使用cache或write buffer。关于cache和write buffer的有关内容，我们稍后会详述。

（5）1~0位。这两位用来区分页表类型，对于段页表，这两位的值总是为"10"。

总的来说，段页表项的内容虽然复杂，但归结起来无非就是两个问题，一是，如何解决某一虚拟地址属于哪段物理地址，二是，如何确定这段地址的访问权限。使用其他形式的页表，本质上也是要解决这两个问题。

现在，假设段页表已经被成功地添加到内存之中了，那么接下来的问题是我们应该怎样通过一个虚拟地址找到与之对应的段页表项呢？找到页表项之后，又是怎样找到对应的物理地址的呢？

很显然，虚拟地址本身就可以解决上述问题。

如图3-5所示，首先，我们要让MMU知道段页表在内存中的首地址，也就是图中所说的页表基地址，因为段页表是我们通过程序确定的，存储在什么位置程序员自然清楚。然后，在CPU需要寻址的时候，MMU就可以自动地利用页表将虚拟地址映射为物理地址并寻址物理地址，其步骤如下：

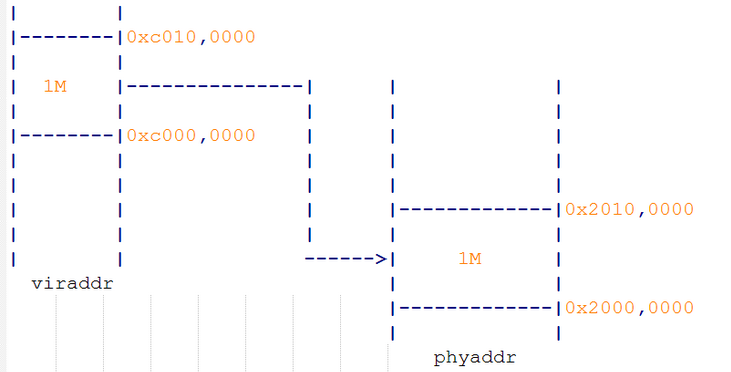
（1）MMU取出虚拟地址的前12位作为页表项的偏移，结合页表基地址，找到对应的页表项。具体来说，就是将这12位数取出，然后左移两位和页表基地址相或，就能得到相应页表项的地址了。例如，页表基地址是0x10000000，如果虚拟地址是0x00101000，则前12位数是0x001，那么根据上述步骤将其左移两位，结果为0x004，那么页表项地址就应该是0x10000000|0x004=0x10000004，从该地址中读取的32位的数据即是该虚拟地址所对应的页表项。

（2）找到与虚拟地址对应的页表项之后，就可以从该页表项中读出一个重要信息，如图3-4所示，页表项的前12位定位了该虚拟地址所对应的物理地址在哪个范围内。例如，从0x10000004中读出的页表项的内容为0x30000c12，那么我们就已经知道了，虚拟地址0x00101000对应的物理地址在0x30000000与0x30100000之间。既然地址范围已经清楚了，那么虚拟地址又该定位到该范围的哪个位置呢？这就要靠虚拟地址的后20位了。

|  |
| --- |
| <http://images.51cto.com/files/uploadimg/20101221/132741967.jpg> |
| （点击查看大图）图3-5  段页表映射过程 |

（3）MMU将虚拟地址后20位和页表项内容清除掉后20位之后的结果做与的操作，就得到了与虚拟地址对应的实际物理地址了。例如，页表项的内容为0x30000c12，清除掉后20位的结果为0x30000000，虚拟地址的后20位为0x01000，将其和0x30000000相与，结果为0x30001000，这便是最终的物理地址。

当然，MMU在进行地址映射期间，还要进行访问权限的检查，方法是读出页表项的权限位，按照既定规则去检查，如果允许对该地址进行访问则正常访问，如果不允许访问，则抛出异常，通过程序将其捕获并处理。这便是使用段页表时MMU的地址映射过程。



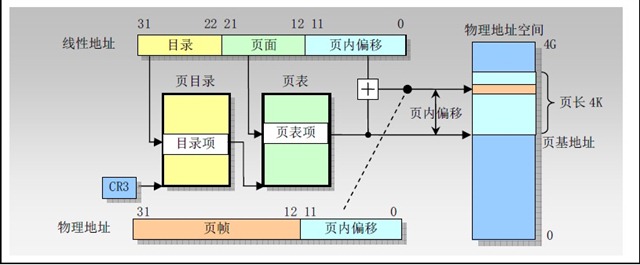
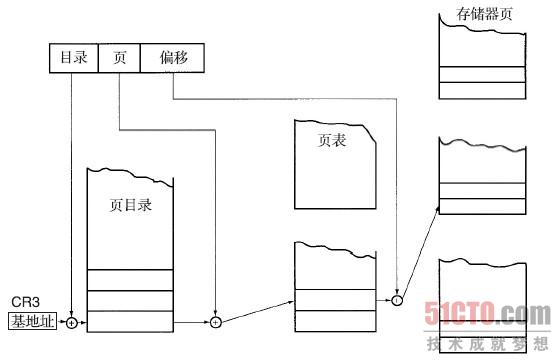
示例：viraddr是0xc000 0000 映射到 phyaddr 0x2000 000



二级页表：

可以让我们访问的内存不是连续的

我们也可以选择使用二级查表的方式去实现地址映射。从原理上讲，一级查表和二级查表其实并没有太大的差别。使用二级查表法，经过一级页表得到的数据不再是记录了物理地址信息的数据了，而是二级页表项的索引信息。而这些信息除了相应位和标志与一级页表项略有不同之外，与一级查表并无不同。



线性地址高10位---------索引页目录表----------->找到相应页表

线性地址中间10位---------索引页表----------->得到页表中相应的项，其中的高20位就是物理地址的高20位

线性地址低12位-------------------->物理地址的低12位

标准Linux的分页是三级页表结构，

**页权限**

前面我们已经介绍过，内存管理单元不仅能够给进程提供独立的地址空间，而且可以设置进程对某段地址空间的访问权限。总结起来，页的权限主要跟两个方面有关-域（domain）和访问权限（access permision）。

域的权限是以1M为单位的，ARM最多允许定义16个不同的域，并允许配置每1M的地址空间分属于16个域中的任意一个，从而可以通过使用改变域权限的方法来批量控制整块地址空间的访问权限。举个例子来说，假设我们配置系统中的前4M内存，使其由第一个域来控制，而将系统中的后4M内存进行配置，让第二个域去控制。那么当环境要求让系统中的前4M内存可以被自由访问、后4M内存只能被特定任务访问时，就可以简单地修改系统中这两个域的权限，迅速地实现要求。在ARM 920T体系结构中，这16个域的权限是由一个叫做CP15的协处理器中的C3寄存器来控制的，如图3-6所示。

|  |
| --- |
| <http://images.51cto.com/files/uploadimg/20101221/132906290.jpg> |
| （点击查看大图）图3-6  C3寄存器 |

在图3-6中，在32位的寄存器C3中，每两位代表一个域权限，一共是16个域。而每个域只有4种取值，分别是0b00、0b01、0b10、0b11，这4个值分别代表了该域的不同权限类型，如表3-2所示。

表3-2  域权限设置

|  |  |  |
| --- | --- | --- |
| 访问 | 域权限值 | 描述 |
| 管理者 | 0b11 | 该域不受来自页表的权限  限制，总可以被访问 |
| 保留 | 0b10 | 结果不可预料 |
| 用户 | 0b01 | 访问受页表中的权限限制 |
| 禁止访问 | 0b00 | 将产生域错误 |

因此，只要配置好某块内存对应域的权限，就可以实时地改变这块内存区的访问类型。接下来的问题是如何知道哪块内存属于哪个域呢？读者只要回头看一下我们前面讲的段页表项的内容就会明白了。在段页表项中，第8位~第5位的值用来表示该段页表项属于16个域中的哪一个。MMU在进行地址映射时，需要读出这些位，在CP15：C3中找到对应域的权限，根据表3-2中的表述确定该段内存的访问权限，然后进行访问，域就是通过这种方式实现了一种粗粒度的页权限。

另一种更有效的权限控制方法是访问权限（access permision），段页表项中11位和10位代表该段页表项的AP位，其可能的取值也只有4种，与之对应的读写权限如表3-3所示。

表3-3  AP权限设置

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AP位 | S位 | R位 | 特权模式 | 用户模式 |
| 0b11 | 忽略 | 忽略 | 可读写 | 可读写 |
| 0b10 | 忽略 | 忽略 | 可读写 | 只读 |
| 0b01 | 忽略 | 忽略 | 可读写 | 不可访问 |
| 0b00 | 0 | 0 | 不可访问 | 不可访问 |
| 0b00 | 0 | 1 | 只读 | 只读 |
| 0b00 | 1 | 0 | 只读 | 不可访问 |
| 0b00 | 1 | 1 | 结果不可预料 | 结果不可预料 |

表3-3中的S位和R位指的是协处理器CP15：C1寄存器中的两个位，专门负责全局修改存储器访问权限，正确地配置好这两个位能够提高系统访问大块内存时的速度。设定S位能够使用户任务无法对页进行访问，同时又允许了特权模式的读权限，而改变R位则可以允许特权模式和用户模式都对页有只读的权限。当然，R位和S位能够发挥作用的前提是页表项中的AP位必须为零值。当页表项中的AP位为非零值时，表示该页表项所对应的内存权限由AP位的值来决定，当AP为零，则该页表项所代表的内存的读写权限由R位和S位决定。修改R位和S位要比一项一项地去修改页表项更迅速，因此适用于对大块内存进行权限控制。

疑惑：

如果使用简单的一级页表，如果进程使用全部4G线性地址空间，那么将需要高达2^20表项（总共地址线是32位，每页大小为4kb，则页偏移量需要低12位，高20位当作页表地址）来保存表示每个进程的页表，若每项4B，则需要4MB的ram来存储页表。即使一个进程并不使用内的所有地址。

         书上提出：使用这种二级模式能够减少每个进程页表所需RAM数量。开始一直没想通，为什么节省了呢？从你最终要存储的表项来看，无论如何你存储的表项是不会少的，而且多级页表还会增加存储开销。

          其实是这样的，二级表只是从进程的角度来看，为进程节省了页表项（其实所有的页表存储空间增大了）。二级模式通过只为进程实际使用的那些虚拟内存区请求页表来减少页表，就是进程未使用的页暂时可以不用为其建立页表，因为如果使用一级页表的话，你就必须为所有的4G范围内分配页表，不能细分。每个活动进程必须有一个分配给它的页目录，不过没必要马上为进程的所有页表都分配ram，只有在进程实际需要一个页表时才给该页表分配ram，这样就提高了效率。

按每个page 4k算，一级页表要寻址4G，需要1M个地址，也就是4M的存储空间  
用2级页表并不会减少存储空间，否则这1M个地址往哪里存放?  
比如对于4G的空间，4K页  
那需要2^20 == 1M个页标项（无论用不用都需要，系统不知道哪个地址是否会被访问）  
1M ＊ 4 ＝＝ 4M  
每个进程都需要4M的页表  
  
而如果2级  
只需要一个页目录 ＋ 1个页表（比如只使用4M的情况）就足够了

常见CPU都带有mmu和缓存，使用二级页表性能几乎没有损失，实际上三级页表也很常见

分析：

http://www.360doc.com/content/11/0804/10/7204565\_137844381.shtml