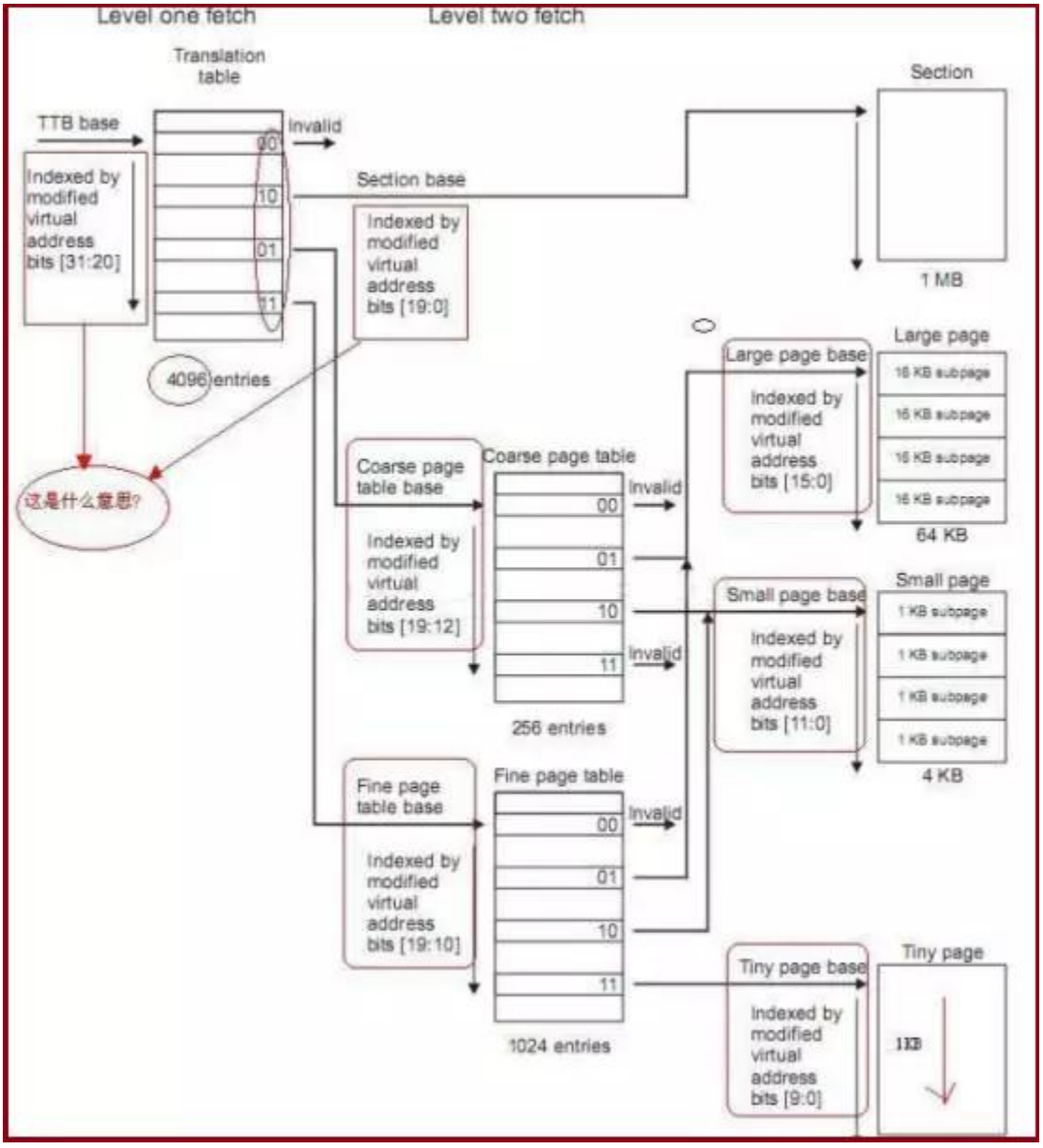
嵌入式 ARM MMU 神秘的内部世界

ARM MMU 页表框架

先上一张 arm mmu 的页表结构的通用框图（以下的论述都由该图来逐渐展开）：



以上是 arm 的页表框图的典型结构，即是二级页表结构。

其中第一级页表（L1）是由虚拟地址的高 12bit（bits[31：20]）组成,所以第一级页表有 4096 个 item,每个 item 占 4 个字节,所以一级页表的大小为 16KB,而在第一级页表中的每个 entry 的最低 2bit 可以用来区分具体是什么种类的页表项，2bit 可以区分 4 种页表项，具体每种页表项的结构如下：

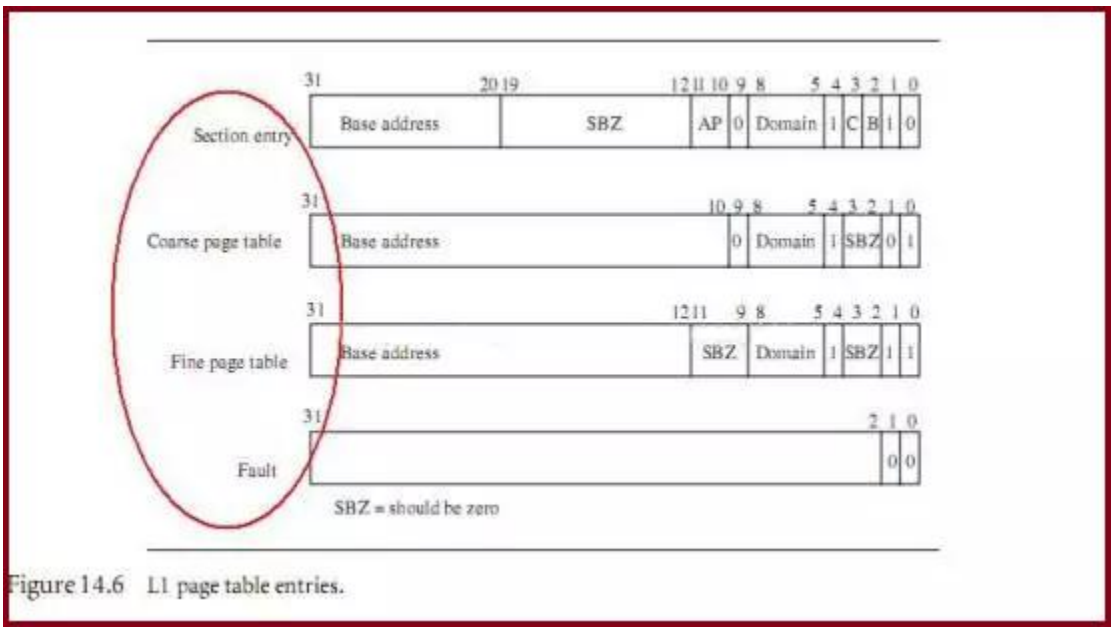


Figure 14.6 L1 page table entries.

简而言之 L1 页表的页表项主要有两大类:

- 第一大类是指向第二级页表（L2 页表）的基地址;
- 第二类直接指向 1MB 的物理内存。

在 L1 页表中每个表项可以覆盖 1MB 的内存，由于有 4096K 个选项（item），所以总计可以覆盖 4096K*1MB=4GB 的内存空间。

具体对应到 Linux，由于 Linux 的软件架构是支持 3 级页表结构，而 arm 架构实际只有 2 级的页表结构,所以 linux 代码中的中间级页表的实现是空的。在 linux 代码中，第一级的页表的页目录表项用 pgd 表示，中间级的页表的页目录表项用 pud 表示（arm 架构其实不需要），第三级的页表的页目录表项用 pmd 表示（由于中间 pud 是空的,所以 pgd=pmd),另外目前 arm 体系的移动设备中 RAM 的 page 大小一般都是 4KB/page,所以 L1 页表中的页表项都是指向 fine page table 的。

但在 linux 内核启动的初始化阶段，临时建立页表（initial page tables）以供 linux 内核初始化提供执行环境，这时 L1 的页表项使用的就是第二种页表项（section enty），他直接映射的是 1M 的内存空间。具体的可以参考 arch/arm/kernel/head.S 中的__create_page_tables 函数，限于篇幅，这里就不展开说了。

针对这种 section page translation，mmu 硬件执行虚拟地址转物理地址的过程如下：

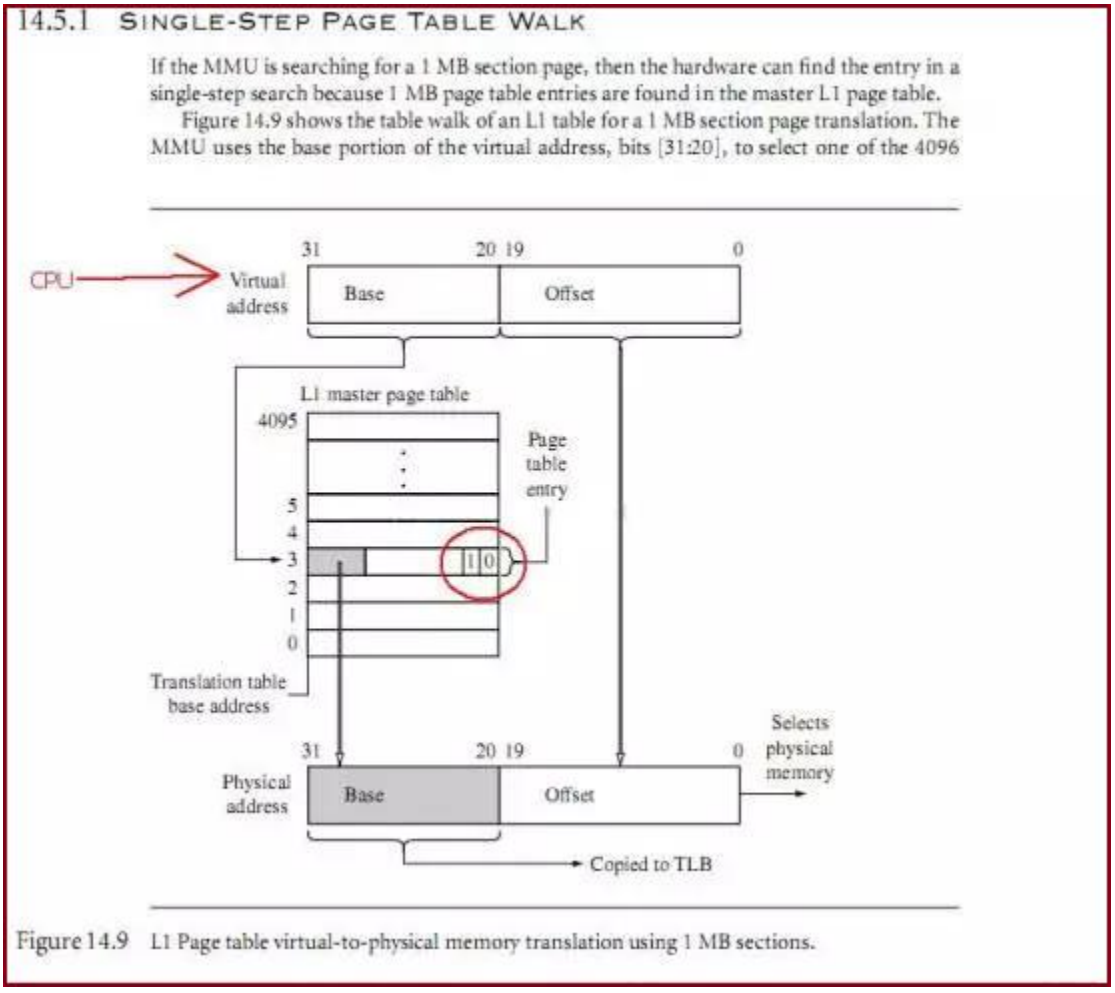


Figure 14.9 L1 Page table virtual-to-physical memory translation using 1 MB sections.

以上在初始化过程使用的临时页表（initial page tables），在内核启动的后期会被覆盖掉，即在 paging_init--->map_lowmem 函数中会重新建立页表，该函数为物理内存从 0 地址到低端内存(lowmem_limit)建立一个一一映射的映射表。所谓的一一映射就是物理地址和虚拟地址就差一个固定的偏移量，该偏移量一般就是 0xc0000000（呵呵，为什么是 0xc0000000？）。

说到这里引入一个重要的概念，就是与低端内存相对的高端内存，什么是高端内存？为什么需要高端内存？为了解析这个问题，我们假设我们使用的物理内存有 2GB 大小，另外由于我们内核空间的地址范围是从 3G-4G 的空间，并且前面也说到了，linux 内核的低端内存空间都是一一映射的，如果不引入高端内存这个概念，全部都使用一一映射的方式，那内核只能访问到 1GB 的物理内存，但实际上，我们是需要内核在内核空间能够访问所有的 4GB 的内存大小的，那怎么做到呢？

方法就是我们不让 3G-4G 的空间都使用一一映射，而是将物理地址的[0x00，fix_addr]（fix_addr<1GB）映射到内核空间虚拟地址[0x00+3G，fix_addr+3G]，然后将[fix_addr+3G，4G]这段空间保留下来用于动态映射，这样我们可以通过这段虚拟地址来访问从 fix_addr 到 4GB 的物理内存空间。怎么做到的呢？

譬如我们想要访问物理地址[fix_addr，4GB]这段区间中的任何一段，我就用宝贵的内核虚拟地址[fix_addr+3G，4G]的一段去映射他，建立好 mmu 硬件使用的页表，访问完后，将映射清除，将内核的这段虚拟地址释放，以供下次访问其他的物理内存使用。这样就可以达到访问所有 4GB 的物理内存的目的。

那么内核代码是如何建立映射表的呢？

我们着重从 arch/arm/mm/mmu.c 中的 create_mapping 函数来分析。在分析之前我们先看下 arm mmu 硬件是如何在二级页表结构中,实现虚拟地址转物理地址的。

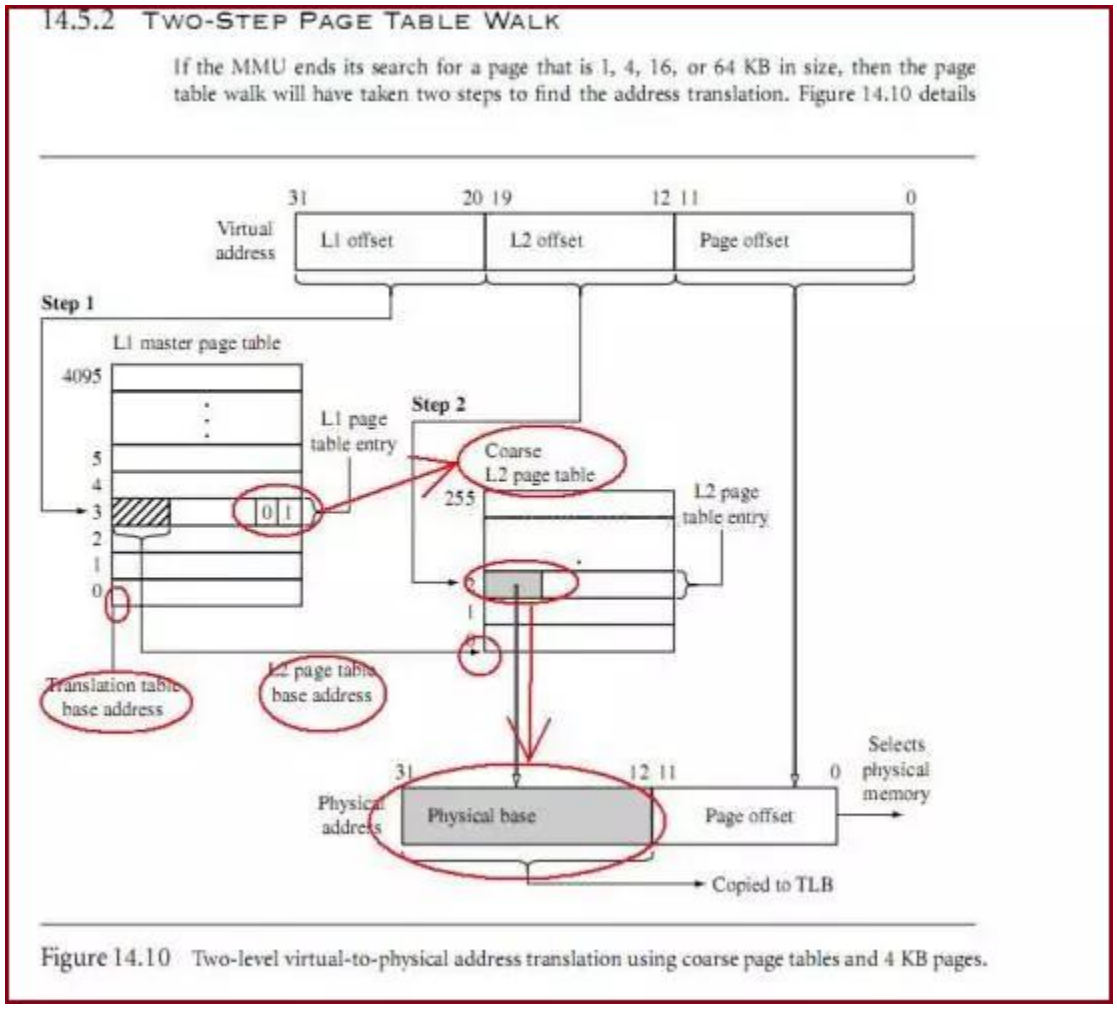


Figure 14.10 Two-level virtual-to-physical address translation using coarse page tables and 4 KB pages.

先贴出原代码（arch/arm/mm/mmu.c），该函数的功能描述如下：

```
Create the page directory entries and any necessary
   page tables for the mapping specified by `md'.   We
   are able to cope here with varying sizes and address
```

offsets, and we take full advantage of sections and supersections.

```
00730: static void __init create_mapping(struct map_desc *md, bool force_pages)
00731: {
00732:     unsigned long addr, length, end;
00733:     phys_addr_t phys;
00734:     const struct mem_type *type;
00735:     pgd_t *pgd;
00736:
00737:     if (md->virtual != vectors_base() && md->virtual < TASK_SIZE) {
00738:         printk(KERN_WARNING "BUG: not creating mapping for 0x%08llx"
00739:                " at 0x%08lx in user region\n",
00740:                (long long)__pfn_to_phys((u64)md->pfn), md->virtual);
00741:         return;
00742:     }
00743:
00744:     if ((md->type == MT_DEVICE || md->type == MT_ROM) &&
00745:         md->virtual >= PAGE_OFFSET &&
00746:         (md->virtual < VMALLOC_START || md->virtual >= VMALLOC_END)) {
00747:         printk(KERN_WARNING "BUG: mapping for 0x%08llx"
00748:                " at 0x%08lx out of vmalloc space\n",
00749:                (long long)__pfn_to_phys((u64)md->pfn), md->virtual);
00750:     }
00751:
00752:     type = &mem_types[md->type];
00753:
00754: #ifndef CONFIG_ARM_LPAE
00755:     /*
00756:      * Catch 36-bit addresses
00757:      */
00758:     if (md->pfn >= 0x100000) {
00759:         create_36bit_mapping(md, type);
00760:         return;
00761:     }
00762: #endif
00763:
00764:     addr = md->virtual & PAGE_MASK;
00765:     phys = __pfn_to_phys(md->pfn);
00766:     length = PAGE_ALIGN(md->length + (md->virtual & ~PAGE_MASK));
00767:
00768:     if (type->prot_l1 == 0 && ((addr | phys | length) & ~SECTION_MASK)) {
00769:         printk(KERN_WARNING "BUG: map for 0x%08llx at 0x%08lx can not "
00770:                "be mapped using pages, ignoring.\n",
00771:                (long long)__pfn_to_phys(md->pfn), addr);
00772:         return;
00773:     }
00774:
00775:     pgd = pgd_offset_k(addr);
00776:     end = addr + length;
00777:     do {
00778:         unsigned long next = pgd_addr_end(addr, end);
00779:
00780:         alloc_init_pud(pgd, addr, next, phys, type, force_pages);
00781:
00782:         phys += next - addr;
00783:         addr = next;
00784:     } while (pgd++, addr != end);
00785: } ? end create_mapping ?
```

line737-line742:参数合法性检查，该函数不为用户空间的虚拟地址建立映射表（记得多问自己一个为什么？）

line744-line750：如果是 iomemory，则映射的虚拟地址范围应属于高端内存区间，由于我们这里是常规的 memory，即 type 为 MT_MEMORY，所以不会进入该分支。

line775： 获得该虚拟地址 addr 属于第一级页表（L1）的哪个表项，详细跟踪 pgd_offset_k 函数（定义在：arch/arm/include/asm/pgtable.h），你会发现，我们内核的 L1 页目录表的基地址位于 0xc0004000，而我们的内核代码则是放置在 0xc0008000 开始的位置。而从 0xc0004000 到 0xc0008000 区间大小是 16KB，刚好就是 L1 页表的大小（见文章开头的描述）。

在这里需要注意一个概念：内核的页目录表项和进程的页目录表项，内核的页目录表项是对系统所有进程都是公共的；而进程的页目录表项则是跟特定进程相关的，每个应用进程都有自己的页目录表项，但各个进程对应的内核空间的页目录表相都是一样的。正是由于每个进程都有自己的页目录表相，所以才能做到每个进程都可以独立拥有属于自己的[0，3GB]的内存空间。

line778 pgd_addr_end()确保[addr,next]地址不会跨越一个 L1 表项所能映射的最大内存空间 2MB（为什么是 2MB 而不是 1MB 呢？这个是 linux 的一个处理技巧，以后再详细展开说）。

line780 alloc_init_pud()函数为定位到的 L1 页目录表项 pgd 所指向的二级页表（L2）建立映射表。

line784 pdg++下移 L1 页目录表项 pgd，映射下一个 2MB 空间的虚拟地址到对应的 2MB 的物理空间。

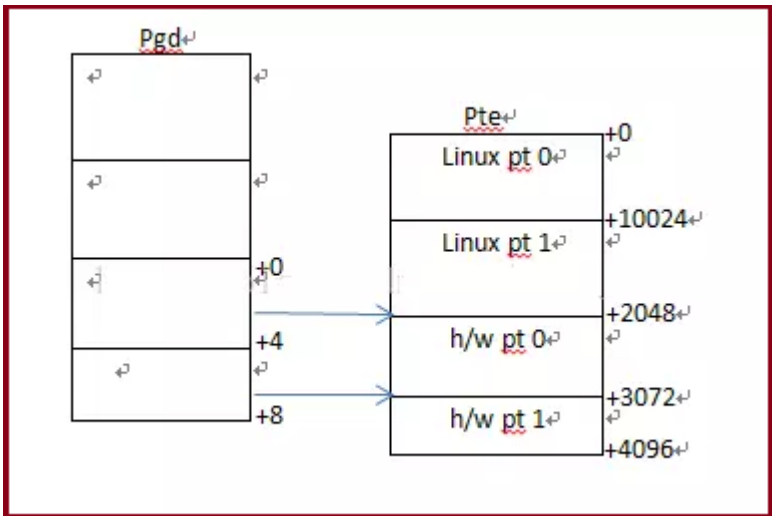
在这里解析下，为什么 L1 页目录表项 pgd 能够映射 2MB 的虚地地址空间。

在本文的第一个图中，他是 arm 典型的 mmu 映射框架图，但并不是 linux 的，linux 映射框架图在它的基础做了些调整和优化。

linux 所做的调整描述如下（以下摘自 linux 内核：
arch/arm/include/asm/pgtable-2level.h 中提供的注释说明）：

```
/*
 * Hardware-wise, we have a two level page table structure, where the first
 * level has 4096 entries, and the second level has 256 entries.  Each entry
 * is one 32-bit word.  Most of the bits in the second level entry are used
 * by hardware, and there aren't any "accessed" and "dirty" bits.
 *
 * Linux on the other hand has a three level page table structure, which can
```

```
 * be wrapped to fit a two level page table structure easily - using the PGD
 * and PTE only.  However, Linux also expects one "PTE" table per page, and
 * at least a "dirty" bit.
 *
 * Therefore, we tweak the implementation slightly - we tell Linux that we
 * have 2048 entries in the first level, each of which is 8 bytes (iow, two
 * hardware pointers to the second level.)  The second level contains two
 * hardware PTE tables arranged contiguously, preceded by Linux versions
 * which contain the state information Linux needs.  We, therefore, end up
 * with 512 entries in the "PTE" level.
 *
 * This leads to the page tables having the following layout:
 *
```



重要调整说明如下：
L1 页表从 4096 个 item 变为 2048 个 item，但每个 item 的大小从原来的 4 字节变为 8 个字节。

一个 page 中，放置 2 个 L2 页表，每个还是 256 项，每项是 4 个字节，所以总计是 256*2*4=2KB，放置在 page 页的下半部，而上部分放置对应的 linux 内存管理系统使用的页表，mmu 硬件是不会去使用它的。所以刚好 占满一个 page 页的大小（4KB），这样就不浪费空间了。

有了上面基础，下面再详细的分析以上的 line780 的函数 alloc_init_pud，该函数会最终调用到 alloc_init_pte 函数：

```
00594: static void __init alloc_init_pte(pmd_t *pmd, unsigned long addr,
00595:                       unsigned long end, unsigned long pfn,
00596:                       const struct mem_type *type)
00597: {
00598:     pte_t *start_pte = early_pte_alloc(pmd);
00599:     pte_t *pte = start_pte + pte_index(addr);
00600:
00601:     /* If replacing a section mapping, the whole section must be replaced */
00602:     BUG_ON(pmd_bad(*pmd) && ((addr | end) & ~PMD_MASK));
00603:
00604:     do {
00605:         set_pte_ext(pte, pfn_pte(pfn, __pgprot(type->prot_pte)), 0);
00606:         pfn++;
00607:     } while (pte++, addr += PAGE_SIZE, addr != end);
00608:     early_pte_install(pmd, start_pte, type->prot_l1);
00609: }
```

line598 early_pte_alloc 函数判断对应的 pmd 所指向的 L2 页表是否存在，如果不存在就分配 L2 页表，如果存在就返回 L2 页表所在 page 页的虚地址。

```
00570: static pte_t * __init early_pte_alloc(pmd_t *pmd)
00571: {
00572:     if (pmd_none(*pmd) || pmd_bad(*pmd))
00573:         return early_alloc(PTE_HWTABLE_OFF + PTE_HWTABLE_SIZE);
00574:     return pmd_page_vaddr(*pmd);
00575: }
00576:
```

line572 判断 pmd 所指向的 L2 页表是否存在，不存在则通过 early_alloc 函数分配 PTE_HWTABLE_OFF（512*4=2KB）+PTE_HWTABLE_SIZE（512*4=2KB）总计 4KB 的一个物理页来存储 2 个 linuxpet 页表+2 个 hwpte 页表。

line574 返回这个物理页所在虚拟地址，
回到 alloc_init_pte 函数的 line599。

line183 pte_index 用来确定该虚拟地址在 L2 页表中的偏移量。即虚拟地址的 bit[12~21]共计 9 个 bit，刚好用于寻址两个 L2 页表（总计 512 项）。

回到 alloc_init_pte 函数，其中 line605 行，是设置 L2 页表中 addr 所定位到的页表项（即 pte），主要工作就是填充对应物理页的物理地址，以供 mmu 硬件来实现地址的翻译。

line604~line607 循环填充完两个 hwpte 页表，完成一个 2M 物理内存的映射表的建立。

line608 将最终调用如下函数：static inline void __pmd_populate(pmd_t *pmdp, phys_addr_t pte, pmdval_t prot)

```
00129: static inline void __pmd_populate(pmd_t *pmdp, phys_addr_t pte,
00130:                                    pmdval_t prot)
00131: {
00132:     pmdval_t pmdval = (pte + PTE_HWTABLE_OFF) | prot;
00133:     pmdp[0] = __pmd(pmdval);
00134: #ifndef CONFIG_ARM_LPAE
00135:     pmdp[1] = __pmd(pmdval + 256 * sizeof(pte_t));
00136: #endif
00137:     flush_pmd_entry(pmdp);
00138: }
```

在执行这个函数之前，2 个 L2 页表已经建立，该函数的作用就是设置 L1 页表的对应表项，使其指向刚建立的 2 个 L2 页表（hwpte0，hwpte1），正如前面所说，由于 linux 的 L1 页表项是 8 个字节大小，所以：

- line133 将头 4 个字节指向 hwpte0 页表，
- line135 将后 4 个字节指向 hwpte1 页表，至此 L1--->L2 页表的关联已经建立。
- line137 是刷新 TLB 缓冲，使系统的 cpu 都可以看见该映射的变化

至此已完成 struct map_desc *md 结构体所指定的虚拟地址到物理地址的映射关系的建立，以供硬件 mmu 来自动实现虚拟到物理地址的翻译。

以上过程，有选择的将某些细节给省略了，限于篇幅，另外如果明白了这个过程，很细节的可以自己去看相关的代码。譬如上面的 set_pte_ext 函数，会调用的汇编函数来实现 pte 表项的设置。