# ARM64 Kernel Image Mapping的变化

## 引言

随着linux的代码更新，阅读linux-4.15代码，从中发现很多与众不同的地方。之所以与众不同，就是因为和我之前从网上博客或者书籍中看到的内容有所差异。当然了，并不是为了表明书上或者博客的观点是错误的。而是因为linux代码更新的太快，网上的博客和书籍跟不上linux的步伐而已。究竟是哪些发生了差异了？例如：kernel image映射区域从原来的linear mapping region（线性映射区域）搬移到VMALLOC区域。因此，我希望通过本篇文章揭晓这些差异。当然，我相信不久的将来这篇文章也将会成为一段历史。

注：文章代码分析基于linux-4.15，架构基于aarch64（ARM64）。涉及页表代码分析部分，假设页表映射层级是4，即配置CONFIG\_ARM64\_PGTABLE\_LEVELS=4。地址宽度是48，即配置CONFIG\_ARM64\_VA\_BITS=48。

## kernel启动页表在哪里

在ARM64架构上，汇编代码初始化阶段会创建两次地址映射。第一次是为了打开MMU操作的准备。因为再打开MMU之前，当前代码运行在物理地址之上，而打开MMU之后代码运行在虚拟地址之上。为了从物理地址（Physical Address，简称PA）转换到虚拟地址（Virtual Address，简称VA）的平滑过渡，ARM推荐创建VA和PA相等的一段映射（例如：虚拟地址addr通过页表查询映射的物理地址也是addr）。这段映射在linux中称为identity mapping。第二次是kernel image映射。而这段映射在linux-4.15代码上映射区域是VMALLOC区域。

kernel启动开始首先就会打开MMU，但是打开MMU之前，我们需要填充页表。也就是告诉MMU虚拟地址和物理地址的对应关系。系统启动初期使用section mapping，因此需要3个页面存储页表项。但是我们有identity mapping和kernel image mapping，因此总需要6个页面。那么这6个页面内存是在哪里分配的呢？可以从(./arch/arm64/kernel/vmlinux.lds.S)vmlinux.lds.S中找到答案。

1. BSS\_SECTION(0, 0, 0)
3. . = ALIGN(PAGE\_SIZE);
4. idmap\_pg\_dir = .;
5. . += IDMAP\_DIR\_SIZE;
6. swapper\_pg\_dir = .;
7. . += SWAPPER\_DIR\_SIZE;

从链接脚本中可以看到预留6个页面存储页表项。紧跟在bss段后面。idmap\_pg\_dir是identity mapping使用的页表。swapper\_pg\_dir是kernel image mapping初始阶段使用的页表。请注意，这里的内存是一段连续内存。也就是说页表（PGD/PUD/PMD）都是连在一起的，地址相差PAGE\_SIZE（4k）。

## 如何填充页表的页表项

从链接脚本vmlinux.lds.S文件中可以找到kernel代码起始代码段是".head.text"段，因此kernel的代码起始位置位于arch/arm64/kernel/head.S文件\_head标号。在head.S文件中有三个宏定义和创建地址映射相关。分别是：create\_table\_entry、create\_pgd\_entry和create\_block\_map。

create\_table\_entry实现如下。

1. /\*
2. \* Macro to create a table entry to the next page.
3. \*
4. \* tbl: 页表基地址
5. \* virt: 需要创建地址映射的虚拟地址
6. \* shift: #imm page table shift
7. \* ptrs: #imm pointers per table page
8. \*
9. \* Preserves: virt
10. \* Corrupts: tmp1, tmp2
11. \* Returns: tbl -> next level table page address
12. \*/
13. .macro create\_table\_entry, tbl, virt, shift, ptrs, tmp1, tmp2
14. lsr \tmp1, \virt, #\shift
15. and \tmp1, \tmp1, #\ptrs - 1 // table index
16. add \tmp2, \tbl, #PAGE\_SIZE
17. orr \tmp2, \tmp2, #PMD\_TYPE\_TABLE // address of next table and entry type
18. str \tmp2, [\tbl, \tmp1, lsl #3]
19. add \tbl, \tbl, #PAGE\_SIZE // next level table page
20. .endm

这里是汇编中的宏定义。汇编中宏定义是以.macro开头，以.endm结尾。宏定义中以\x来引用宏定义中的参数x。该宏定义的作用是创建一个level的页表项（PGD/PUD/PMD）。具体是哪个level是由virt、shift和ptrs参数决定。我总是喜欢帮你翻译成C语言的形式。C语言如果不懂的话，我也没办法了。既然汇编你不熟悉，没关系，下面帮你转换成C语言的宏定义。

1. #define PAGE\_SIZE (1 << 12)
2. #define PMD\_TYPE\_TABLE (3 << 0)
3. #define create\_table\_entry(tbl, virt, shift, ptrs, tmp1, tmp2) **do** { \
4. tmp1 = virt >> shift; /\* 1 \*/ \
5. tmp1 &= ptrs - 1; /\* 1 \*/ \
6. tmp2 = tbl + PAGE\_SIZE; /\* 2 \*/ \
7. tmp2 |= PMD\_TYPE\_TABLE; /\* 3 \*/ \
8. \*((**long** \*)(tbl + (tmp1 << 3))) = tmp2; /\* 4 \*/ \
9. tbl += PAGE\_SIZE; /\* 5 \*/ \
10. } **while** (0)
11. 根据virt和ptrs参数计算该虚拟地址virt的页表项在页表中的index。例如计算virt地址在PGD也表中的indedx，可以传递shift = PGDIR\_SHIFT，ptrs = PTRS\_PER\_PGD，tbl传递PGD页表基地址。所以，宏定义是一个创建中间level的页表项。
12. 既然要填充当前level的页表项就需要告知下一个level页表的基地址，这里就是计算下一个页表的基地址。还记得上面说的idmap\_pg\_dir和swapper\_pg\_dir吗？页表（PGD/PUD/PMD）都是连在一起的，地址相差PAGE\_SIZE。
13. 告知MMU这是一个中间level页表并且是有效的。
14. 页表项的真正填充操作，tmp1 << 3是因为ARM64的地址占用8bytes。
15. 更新tbl，也就只指向下一个level页表的地址，可以方便再一次调用create\_table\_entry填充下一个level页表项而不用自己更新tbl。

create\_pgd\_entry的实现如下。

1. /\*
2. \* Macro to populate the PGD (and possibily PUD) for the corresponding
3. \* block entry in the next level (tbl) for the given virtual address.
4. \*
5. \* Preserves: tbl, next, virt
6. \* Corrupts: tmp1, tmp2
7. \*/
8. .macro create\_pgd\_entry, tbl, virt, tmp1, tmp2
9. create\_table\_entry \tbl, \virt, PGDIR\_SHIFT, PTRS\_PER\_PGD, \tmp1, \tmp2
10. create\_table\_entry \tbl, \virt, SWAPPER\_TABLE\_SHIFT, PTRS\_PER\_PTE, \tmp1, \tmp2
11. .endm

create\_pgd\_entry可以用来填充PGD、PUD、PMD等中间level页表对应页表项。虽然名字是创建PGD的描述符，但是实际上是一级一级的创建页表项，最终只留下最后一级页表没有填充页表项。老规矩转换成C语言分析。

1. #define SWAPPER\_TABLE\_SHIFT PUD\_SHIFT
2. #define create\_pgd\_entry(tbl, virt, tmp1, tmp2) **do** { \
3. create\_table\_entry(tbl, virt, PGDIR\_SHIFT, PTRS\_PER\_PGD, tmp1, tmp2); /\* 1 \*/ \
4. create\_table\_entry(tbl, virt, SWAPPER\_TABLE\_SHIFT, PTRS\_PER\_PTE, tmp1, tmp2); /\* 2 \*/ \
5. } **while** (0)

* 这里的tbl参数相当于PGD页表地址，调用create\_table\_entry创建PGD页表中virt地址对应的页表项。
* 填充下一个level的页表项。这里是PUD页表。由于使用了ARM64初期使用section mapping，因此PUD页表就是最后一个中间level的页表，所以只剩下PMD页表的页表项没有填充，virt地址对应的PMD页表项最终会填充block descriptor。假设这里使用4级页表，那么下面还会创建PMD页表的页表项，也就是只留下PTE页表。所以，宏定义是创建所有中间level的页表项，只留下最后一级页表。

在经过create\_pgd\_entry宏的调用后，就填充好了从PGD开始的所有中间level的页表的页表项的填充操作。现在是不是只剩下PTE页表的页表项没有填充呢？所以最后一个create\_block\_map就是完成这个操作的。

1. /\*
2. \* Macro to populate block entries in the page table for the start..end
3. \* virtual range (inclusive). \*
4. \* Preserves: tbl, flags
5. \* Corrupts: phys, start, end, pstate
6. \*/
7. .macro create\_block\_map, tbl, flags, phys, start, end
8. lsr \phys, \phys, #SWAPPER\_BLOCK\_SHIFT
9. lsr \start, \start, #SWAPPER\_BLOCK\_SHIFT
10. and \start, \start, #PTRS\_PER\_PTE - 1 // table index
11. orr \phys, \flags, \phys, lsl #SWAPPER\_BLOCK\_SHIFT // table entry
12. lsr \end, \end, #SWAPPER\_BLOCK\_SHIFT
13. and \end, \end, #PTRS\_PER\_PTE - 1 // table end index
14. 9999: str \phys, [\tbl, \start, lsl #3] // store the entry
15. add \start, \start, #1 // next entry
16. add \phys, \phys, #SWAPPER\_BLOCK\_SIZE // next block
17. cmp \start, \end
18. b.ls 9999b
19. .endm

create\_block\_map宏的作用是创建虚拟地址（从start到end）区域映射到到phys物理地址。传入5个参数，分别如下意思。

* tbl：页表基地址
* flags：将要填充页表项的flags
* phys：创建映射的物理地址
* start：创建映射的虚拟地址起始地址
* end：创建映射的虚拟地址结束地址

我们还是依然翻译成C语言分析。

1. #define SWAPPER\_BLOCK\_SHIFT PMD\_SHIFT
2. #define SWAPPER\_BLOCK\_SIZE (1 << PMD\_SHIFT)
3. #define create\_block\_map(tbl, flags, phys, start, end) **do** { \
4. phys >>= SWAPPER\_BLOCK\_SHIFT; /\* 1 \*/ \
5. start >>= SWAPPER\_BLOCK\_SHIFT; /\* 2 \*/ \
6. start &= PTRS\_PER\_PTE - 1; /\* 2 \*/ \
7. phys = flags | (phys << SWAPPER\_BLOCK\_SHIFT);/\* 3 \*/ \
8. end >>= SWAPPER\_BLOCK\_SHIFT; /\* 4 \*/ \
9. end &= PTRS\_PER\_PTE - 1; /\* 4 \*/ \
10. **while** (start != end) { /\* 5 \*/ \
11. \*((**long** \*)(tbl + (start << 3))) = phys; /\* 6 \*/ \
12. start++; /\* 7 \*/ \
13. phys += SWAPPER\_BLOCK\_SIZE; /\* 8 \*/ \
14. } \
15. } **while** (0)
16. 针对phys的低SWAPPER\_BLOCK\_SHIFT位进行清零，和第三步骤的phys << SWAPPER\_BLOCK\_SHIFT收尾呼应。相当于对齐（这里的情况是2M对齐）。
17. 计算起始地址start的页目录项的index。
18. 构造描述符。
19. 计算结束地址end的页目录项的index。
20. 循环填充start到end的页目录项。
21. 根据页表基地址tbl和当前的start变量填充对应的页表项。start << 3是因为ARM64地址占用8 bytes。
22. 更新下一个页表项。
23. 更新下一个block的物理地址。

如何使用上述三个接口创建映射关系呢？其实很简单，首先我们需要先调用create\_pgd\_entry宏填充PGD以及所有中间level的页表项。最后的PMD页表的填充可以调用create\_block\_map宏来完成操作。

## 如何创建页表

在汇编代码阶段的head.S文件中，负责创建映射关系的函数是create\_page\_tables。create\_page\_tables函数负责identity mapping和kernel image mapping。前文提到identity mapping主要是打开MMU的过度阶段，因此对于identity mapping不需要映射整个kernel，只需要映射操作MMU代码相关的部分。如何区分这部分代码呢？当然是利用linux中常用手段自定义代码段。自定义的代码段的名称是".idmap.text"。除此之外，肯定还需要在链接脚本中声明两个标量，用来标记代码段的开始和结束。可以从vmlinux.lds.S中找到答案。

1. #define IDMAP\_TEXT \
2. . = ALIGN(SZ\_4K); \
3. VMLINUX\_SYMBOL(\_\_idmap\_text\_start) = .; \
4. \*(.idmap.text) \
5. VMLINUX\_SYMBOL(\_\_idmap\_text\_end) = .;

从链接脚本中可以看出idmap\_text\_start和idmap\_text\_end分别是.idmap.text段的起始和结束地址。在创建identity mapping的时候会使用。另外我们同样从链接脚本中得到\_text和\_end两个变量，分别是kernel代码链接的开始和结束地址。编译器的链接地址实际上就是最后代码期望运行的地址。在KASLR关闭的情况下就是kernel image需要映射的虚拟地址。当我们编译kernel后，可以根据符号表System.map文件查看哪些函数被放在".idmap.text"段。当然你也可以看代码，但是我觉得没有这种方法简单。

1. ffff200008fbc000 T \_\_idmap\_text\_start
2. ffff200008fbc000 T kimage\_vaddr
3. ffff200008fbc008 T el2\_setup
4. ffff200008fbc054 t set\_hcr
5. ffff200008fbc118 t install\_el2\_stub
6. ffff200008fbc16c t set\_cpu\_boot\_mode\_flag
7. ffff200008fbc190 T secondary\_holding\_pen
8. ffff200008fbc1b4 t pen
9. ffff200008fbc1c8 T secondary\_entry
10. ffff200008fbc1d4 t secondary\_startup
11. ffff200008fbc1e4 t \_\_secondary\_switched
12. ffff200008fbc218 T \_\_enable\_mmu
13. ffff200008fbc26c t \_\_no\_granule\_support
14. ffff200008fbc290 t \_\_primary\_switch
15. ffff200008fbc2b0 T cpu\_resume
16. ffff200008fbc2d0 T cpu\_do\_resume
17. ffff200008fbc340 T idmap\_cpu\_replace\_ttbr1
18. ffff200008fbc370 T \_\_cpu\_setup
19. ffff200008fbc3f0 t crval
20. ffff200008fbc408 T \_\_idmap\_text\_end

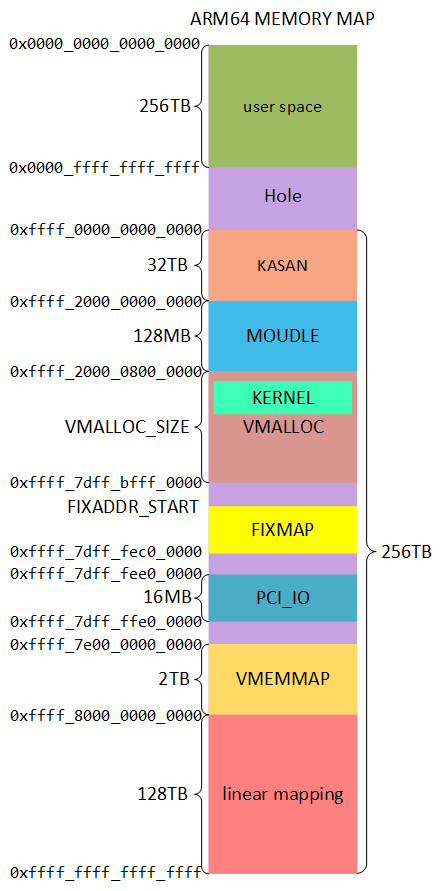
create\_page\_tables的汇编代码比较简单，就不转换成C语言讲解了。create\_page\_tables实现如下。

1. \_\_create\_page\_tables:
2. mov x7, SWAPPER\_MM\_MMUFLAGS
3. /\*
4. \* Create the identity mapping.
5. \*/
6. adrp x0, idmap\_pg\_dir /\* 1 \*/
7. adrp x3, \_\_idmap\_text\_start // \_\_pa(\_\_idmap\_text\_start) /\* 2 \*/
8. create\_pgd\_entry x0, x3, x5, x6 /\* 3 \*/
9. mov x5, x3 // \_\_pa(\_\_idmap\_text\_start) /\* 4 \*/
10. adr\_l x6, \_\_idmap\_text\_end // \_\_pa(\_\_idmap\_text\_end)
11. create\_block\_map x0, x7, x3, x5, x6 /\* 5 \*/
13. /\*
14. \* Map the kernel image.
15. \*/
16. adrp x0, swapper\_pg\_dir /\* 6 \*/
17. mov\_q x5, KIMAGE\_VADDR + TEXT\_OFFSET // compile time \_\_va(\_text)
18. add x5, x5, x23 // add KASLR displacement /\* 7 \*/
19. create\_pgd\_entry x0, x5, x3, x6 /\* 8 \*/
20. adrp x6, \_end // runtime \_\_pa(\_end)
21. adrp x3, \_text // runtime \_\_pa(\_text)
22. sub x6, x6, x3 // \_end - \_text
23. add x6, x6, x5 // runtime \_\_va(\_end)
24. create\_block\_map x0, x7, x3, x5, x6 /\* 9 \*/
25. x0寄存器PGD页表基地址，这里是idmap\_pg\_dir，是为了创建identity mapping。
26. adrp指令可以获取\_\_idmap\_text\_start符号的实际运行物理地址。
27. 填充PGD及中间level页表的页表项。
28. 因为我们为了创建虚拟地址和物理地址相等的映射，因此这里的x5和x3值相等。
29. 调用create\_block\_map创建identity mapping，注意这里传递的参数物理地址（x3）和虚拟地址（x5）相等。
30. 创建kernel image mapping，PGD页表基地址是swapper\_pg\_dir。
31. KASLR默认关闭的情况下，x23的值为0。
32. 填充PGD及中间level页表的页表项。
33. 填充PMD页表项。因为采用的是section mapping，所以一个页表项对应2M大小。注意汇编中的注释，va()代表得到的事虚拟地址，pa()得到的是物理地址。

经过以上初始化，页表就算是初始化完成。kernel映射区域从先行映射区域迁移到VMALLOC区域在哪里体现呢？答案就是KIMAGE\_VADDR宏定义。KIMAGE\_VADDR是kernel的虚拟地址。其定义在arch/arm64/mm/memory.h文件。[\arch\arm64\include\asm\memory.h]

1. #define VA\_BITS (CONFIG\_ARM64\_VA\_BITS)
2. #define VA\_START (UL(0xffffffffffffffff) - (UL(1) << VA\_BITS) + 1)
3. #define PAGE\_OFFSET (UL(0xffffffffffffffff) - (UL(1) << (VA\_BITS - 1)) + 1)
4. #define KIMAGE\_VADDR (MODULES\_END)
5. #define VMALLOC\_START (MODULES\_END)
6. #define VMALLOC\_END (PAGE\_OFFSET - PUD\_SIZE - VMEMMAP\_SIZE - SZ\_64K)
7. #define MODULES\_END (MODULES\_VADDR + MODULES\_VSIZE)
8. #define MODULES\_VADDR (VA\_START + KASAN\_SHADOW\_SIZE)
9. #define MODULES\_VSIZE (SZ\_128M)
10. #define VMEMMAP\_START (PAGE\_OFFSET - VMEMMAP\_SIZE)
11. #define PCI\_IO\_END (VMEMMAP\_START - SZ\_2M)
12. #define PCI\_IO\_START (PCI\_IO\_END - PCI\_IO\_SIZE)
13. #define FIXADDR\_TOP (PCI\_IO\_START - SZ\_2M)
14. #define TASK\_SIZE\_64 (UL(1) << VA\_BITS)

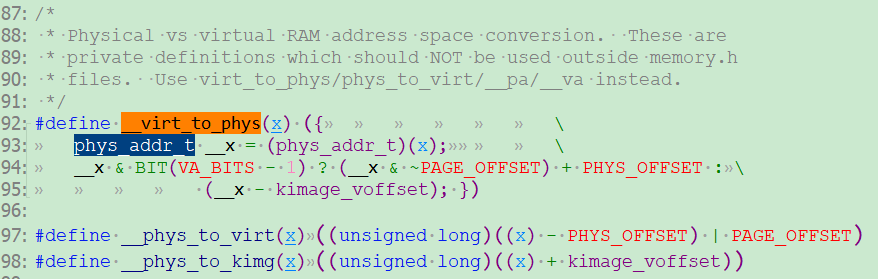
上面的宏定义显得不够直观，画张图表示现阶段kernel的地址空间分布情况。可以看出KIMAGE\_VADDR正好处在VMALLOC区域，因此kernnel的运行地址位于VMALLOC区域。

[](http://www.wowotech.net/content/uploadfile/201804/b8e81524313791.png)

## virt\_to\_phys和phys\_to\_virt怎么办

通过上面的介绍，你应该有所了解kernel image和linear mapping region不在一个区域。virt\_to\_phys宏的作用是将内核虚拟地址转换成物理地址（针对线性映射区域）。在kernel image还在线性映射区域的时候，virt\_to\_phys宏可以将kernel代码中的一个地址转换成物理地址，因为线性映射区域，物理地址和虚拟地址只有一个偏移。因此两者很容易转换。那么现在kernel image和线性映射区域分开了，virt\_to\_phys宏又该如何实现呢？**virt\_to\_phys**宏实现如下。

1. #define PHYS\_OFFSET ({ VM\_BUG\_ON(memstart\_addr & 1); memstart\_addr; })
2. #define \_\_is\_lm\_address(addr) (!!((addr) & BIT(VA\_BITS - 1)))
3. #define \_\_lm\_to\_phys(addr) (((addr) & ~PAGE\_OFFSET) + PHYS\_OFFSET)
4. #define \_\_kimg\_to\_phys(addr) ((addr) - kimage\_voffset)
5. #define \_\_virt\_to\_phys\_nodebug(x) ({ \
6. phys\_addr\_t \_\_x = (phys\_addr\_t)(x); \
7. \_\_is\_lm\_address(\_\_x) ? \_\_lm\_to\_phys(\_\_x) : \
8. \_\_kimg\_to\_phys(\_\_x); \
9. #define \_\_virt\_to\_phys(x) \_\_virt\_to\_phys\_nodebug(x)
11. **static** **inline** phys\_addr\_t virt\_to\_phys(**const** **volatile** **void** \*x)
12. {
13. **return** \_\_virt\_to\_phys((**unsigned** **long**)(x));
14. }



从\_\_virt\_to\_phys\_nodebug宏可以看出其中的奥秘。通过addr地址的(VA\_BITS - 1)位是否为1判断addr是位于kernel image区域还是线性映射区域（因为线性映射区域大小正好是kernel虚拟地址空间大小的一半）。针对线性映射区域，虚拟地址和物理地址的偏差是memstart\_addr。针对kernel image区域，虚拟地址和物理地址的偏差是kimage\_voffset。kimage\_voffset和memstart\_addr是如何计算的呢？先看看kimage\_voffset的计算。

1. #define KERNEL\_START \_text
2. #define \_\_PHYS\_OFFSET (KERNEL\_START - TEXT\_OFFSET)
3. ENTRY(kimage\_vaddr)
4. .quad \_text - TEXT\_OFFSET
5. /\* The following fragment of code is executed with the MMU enabled.
6. \* x0 = \_\_PHYS\_OFFSET \*/
7. \_\_primary\_switched:
8. ldr\_l x4, kimage\_vaddr // Save the offset between /\* 2 \*/
9. sub x4, x4, x0 // the kernel virtual and /\* 3 \*/
10. str\_l x4, kimage\_voffset, x5 // physical mappings /\* 4 \*/
12. b start\_kernel
14. \_\_primary\_switch:
15. bl \_\_enable\_mmu
16. ldr x8, =\_\_primary\_switched
17. adrp x0, \_\_PHYS\_OFFSET /\* 1 \*/
18. br x8
19. x0是\_primary\_switch函数中设置。x0寄存器通过adrp指令可以获取运行时的地址。也就是实际运行的物理地址。你是不是好奇此时不是已经打开MMU了嘛！为什么adrp得到的运行地址就是物理地址呢？请往上翻看看\_primary\_switch函数是不是位于".idmap.text"段，那么该段是identity mapping。因此获取的运行地址虽然是虚拟地址，但是它和实际运行的物理地址相等。
20. x4寄存器保存的是kernel image的运行的虚拟地址。你是不是又好奇这个地方为什么获取的运行地址和物理地址不相等呢？其实是因为\_\_primary\_switched函数映射在kernel image mapping区域。
21. 计算虚拟地址和物理地址的偏移。
22. 将偏移写入kimage\_voffset全局变量。

memstart\_addr是kernel选取的物理基地址，memstart\_addr在arm64\_memblock\_init函数中设置。arm64\_memblock\_init函数实现如下（截取部分代码）。

1. **void** \_\_init arm64\_memblock\_init(**void**)
2. {
3. **const** s64 linear\_region\_size = -(s64)PAGE\_OFFSET;
5. /\*
6. \* Ensure that the linear region takes up exactly half of the kernel
7. \* virtual address space. This way, we can distinguish a linear address
8. \* from a kernel/module/vmalloc address by testing a single bit.
9. \*/
10. BUILD\_BUG\_ON(linear\_region\_size != BIT(VA\_BITS - 1)); /\* 1 \*/
12. /\*
13. \* Select a suitable value for the base of physical memory.
14. \*/
15. memstart\_addr = round\_down(memblock\_start\_of\_DRAM(), /\* 2 \*/
16. ARM64\_MEMSTART\_ALIGN);
18. memblock\_remove(max\_t(u64, memstart\_addr + linear\_region\_size, /\* 3 \*/
19. \_\_pa\_symbol(\_end)), ULLONG\_MAX);
20. **if** (memstart\_addr + linear\_region\_size < memblock\_end\_of\_DRAM()) {
21. /\* ensure that memstart\_addr remains sufficiently aligned \*/
22. memstart\_addr = round\_up(memblock\_end\_of\_DRAM() - linear\_region\_size,
23. ARM64\_MEMSTART\_ALIGN); /\* 4 \*/
24. memblock\_remove(0, memstart\_addr); /\* 5 \*/
25. }
26. }
27. 从注释以及代码皆可以看出，PAGE\_OFFSET是线性区域的开始虚拟地址。线性区域大小是整个kernel虚拟地址空间的一半。
28. 选取一个合适的物理基地址，根据RAM的起始地址按照1G对齐。
29. memstart\_addr是选取的物理基地址。kernel虚拟地址空间一半大小作为线性映射区域。因此最大支持的内存范围是memstart\_addr + linear\_region\_size。所以告诉memblock，超过这个区域的范围都是非法的。
30. 如果memstart\_addr + linear\_region\_size的值小于RAM的结束地址，说明[memstart\_addr, memstart\_addr + linear\_region\_size]地址空间范围的区域无法覆盖整个RAM地址范围。这时候就需要从RAM结束地址减去linear\_region\_size的值作为memstart\_addr。什么时候会出现这种情况呢？当物理内存足够大时，if语句就可能满足条件了。
31. 既然4满足，自然这里[0, memstart\_addr]的地址空间需要从memblock中remove。

memstart\_addr的值定下来之后，虚拟地址和物理地址以memstart\_addr为偏差创建线性映射区域。在map\_mem函数中完成。phys\_to\_virt宏的实现就不用介绍了，就是virt\_to\_phys的反操作。

# Linux启动过程分析（head.s）

/\*\*\*\*\*\*\*\*判断内核是否支持硬件并启动MMU\*\*\*\*\*\*\*\*\*\*\*\*/

        bl         \_\_lookup\_processor\_type                 //查找CPU类型，判断是否支持这个CPU

        movs    r10, r5

        beq      \_\_error\_p @ yes, error 'p'                //如果不支持，跳到这个标号

        bl         \_\_lookup\_machine\_type                 //判断是否支持单板类型，就是根据uboot传参的那个机器id来判断

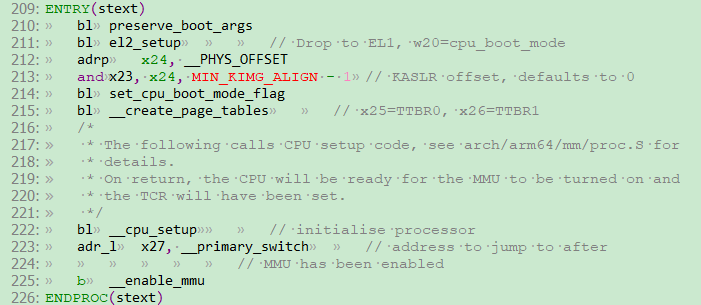
        movs    r8, r5

        beq      \_\_error\_a @ yes, error 'a'                //如果不支持，跳到标号处

        bl         \_\_vet\_atags

        bl       \_\_create\_page\_tables                        //建立页表

        ldr      r13, \_\_switch\_data                            //使能mmu后，跳到这个地址处执行



\_\_lookup\_machine\_type 在arch/arm/kernel/head-common.s中。

\_\_switch\_data也定义在 head-common.s中。 然后这个标号中，一直往下执行，会执行start\_kernel函数。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*启动内核，执行各种初始化\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

start\_kernel函数在/init/main.c中，这里会进行各种初始化，其中有以下几行代码：

printk(linux\_banner);                            //这个话是打印内核版本信息的

setup\_arch(&command\_line);

setup\_command\_line(command\_line);    //这两个函数会解析uboot中传进来的参数，注意，这两个函数只是解析参数，也就是将uboot中传进来的参数分解后保存起来，以便后面使用

接下来就是一直往下调用，挂载根文件系统:

rest\_init  —> kernel\_thread(kernel\_init, NULL, CLONE\_FS | CLONE\_SIGHAND) —> kernel\_init

 —>prepare\_namespace —>mount\_root 到这就挂载了根文件系统，而根文件系统要被挂载到哪个分区是由uboot传进来的参数root=路径名决定的，如：/dev/mtdblock3

如何知道分区的信息？

  这是在代码中已经写好的，如在文件/arch/arm/plat-s3c24xx/common-smdk.c中已经写好来的nandflash的分区信息：static struct mtd\_partition smdk\_default\_nand\_part[ ]中定义了分区信息

当调用mount\_root 挂载好根文件系统后，返回到prepare\_namespace 函数处，继续往下执行：init\_post；

在init\_post函数中,有以下几行代码：

/\*下面的代码主要是打开终端，并且设置标准输入输出，及标准出错\*/

   if (sys\_open((const char \_\_user \*) "/dev/console", O\_RDWR, 0) < 0)

       printk(KERN\_WARNING "Warning: unable to open an initial console.\n");

 (void) sys\_dup(0);   //复制文件描述符，用来做标准输入输出

 (void) sys\_dup(0);  //同上

/\*如果execute\_command不为0，则会执行在uboot定义的环境变量bootargs中的init = xxx。这时便会执行这个程序，注意，一旦执行了，便不会返回，后面的代码都不执行/\*

if (execute\_command) {

  run\_init\_process(execute\_command);   //这个就是执行在uboot中定义的环境变量 init = xxx，执行第一个应用程序

  printk(KERN\_WARNING "Failed to execute %s. Attempting "

     "defaults...\n", execute\_command);

 }

//如果execute\_command为0，则执行/sbin/init应用程序，并且不会返回

   run\_init\_process("/sbin/init");

//如果/sbin/init有错误，则再执行/etc/init应用程序，并且不会返回

   run\_init\_process("/etc/init");

//同上

   run\_init\_process("/bin/init");

//同上

   run\_init\_process("/bin/sh");

又以上代码可以看出，在这个函数中，首先会打开/dev/console，也就是打开终端。

然后执行各种应用程序，一旦其中的一个执行成功，便会一直运行下去，不再返回。