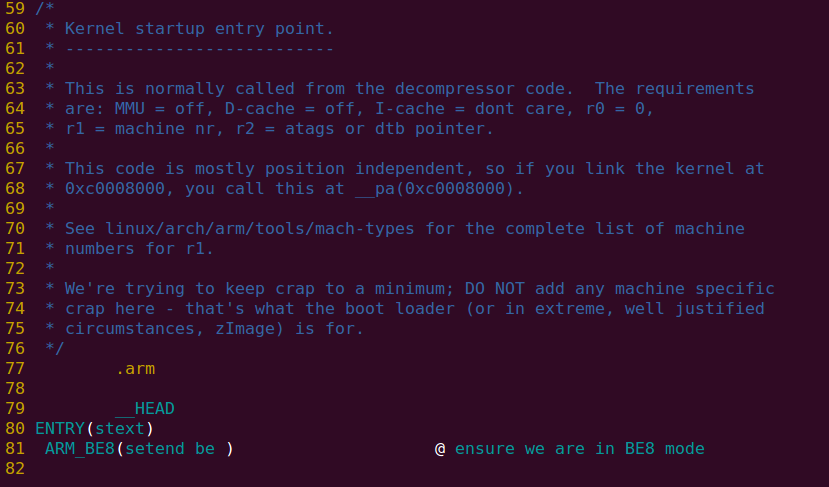
本篇linux内核启动流程分析基于linux3.14源码。

内核启动分为两个个阶段，第一是运行head.S文件和head-common.S，第二个阶段是运行main.c文件 。

  对于ARM的处理器，内核第一个启动的文件是arc/arm/kernel下面的head.S文件。当然arc/arm/boot/compress下面也有这个文件，这个文件和上面的文件略有不同，当要生成压缩的内核时zImage时，启动的是后者，后者与前者不同的时，它前面的代码是做自解压的，后面的代码都相同。我们这里这分析arc/arm/kernel下面的head.S文件。当head.S所作的工作完成后它会跳到init/目录下跌的main.c的start\_kernel函数开始执行。

第一阶段：

首先看看arch/arm/kernel/head.S文件，

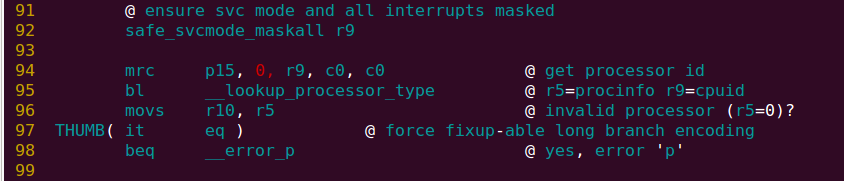


正如60~75行注释所言，80行是在zImage自解压之后整个内核代码的入口点。

这个时候，MMU = off, D-cache = off, I-cache = dont care, r0 = 0,

65 \* r1 = machine nr, r2 = atags or dtb pointer.

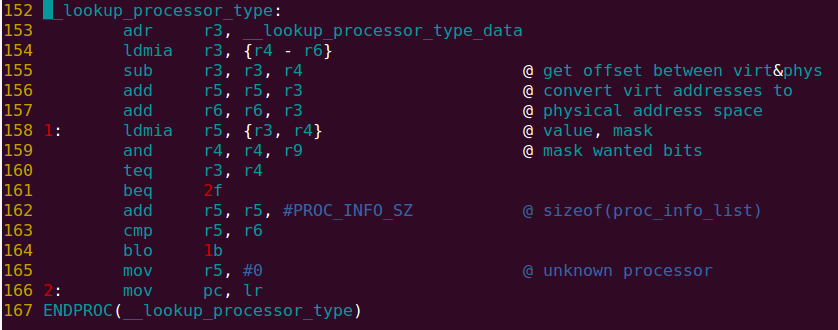
92行，保持CPU为SVC模式，并且屏蔽所有中断。



94行，从cp15协处理器cr0寄存器获取硬件处理器ID存到r9寄存器。

95行，跳到\_\_lookup\_processor\_type子代码段执行，看看具体实现，

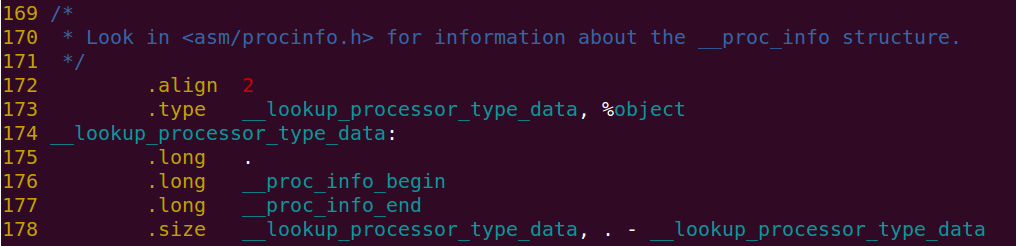
定义在arch/arm/kernel/head-common.S 文件 152行：



主要是拿从硬件读出来的处理器 ID和proc\_info\_list列表里面的所有linux内核支持的处理器型号进行对比，看看linux内核是否支持。

下面具体看看：

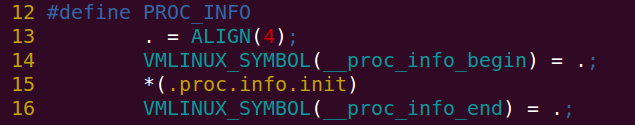
153行，取地址指令,这里的\_\_lookup\_processor\_type\_data的位置,即第174行,将该地址存入r3. 这里需要注意的是,adr指令取址,获得的是基于pc的一个地址,要格外注意,这个地址是\_\_lookup\_processor\_type\_data处的"运行时地址",由于此时MMU还没有打开,也可以理解成物理地址(实地址).



154行: 因为r3中的地址是175行的位置的地址,因而执行完后:

 r4存的是当前的链接地址，为虚拟地址.   
        r5存的是176行符号 \_\_proc\_info\_begin的地址;   
        r6存的是177行符号 \_\_proc\_info\_end的地址;   
        这里需要注意链接地址和运行时地址的区别. r3存储的是运行时地址(物理地址),而r4中存储的是链接地址([虚拟](http://clk.qunsee.com/click/click.php?cpid=1021&ads_id=1090&pid=99005765&cid=5794&url=http%3A//www.chanet.com.cn/click.cgi%3Fa%3D59068%26d%3D41911%26u%3D%26e%3D&k=%u865A%u62DF&s=http%3A//www.chmhome.com/knowledge/linux/20080801/164699_5.html&rn=492730&v=1&ref=http%3A//www.chmhome.com/knowledge/linux/20080801/164699_4.html&province=UNKNOWN&city=UNKNOWN)地址)，所以，

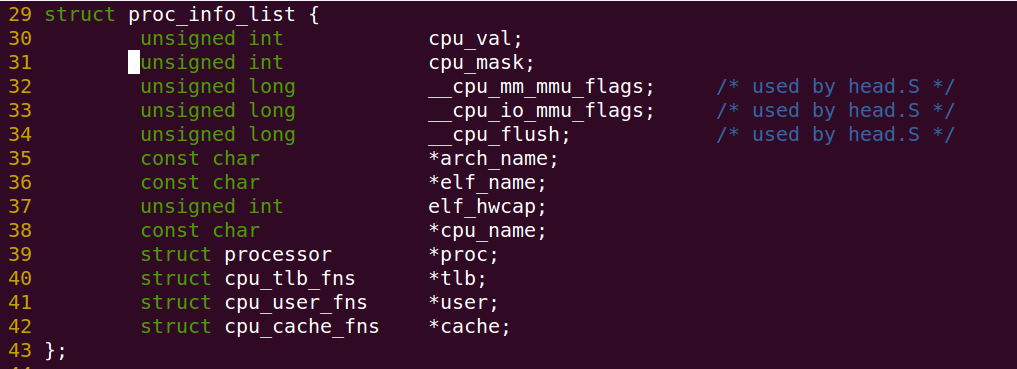
155行，r3-r4得到的是物理地址与虚拟地址offset，存于r3。

\_\_proc\_info\_begin和\_\_proc\_info\_end是在arch/arm/kernel/vmlinux.lds.S中:  


这里是声明了两个变量:\_\_proc\_info\_begin 和 \_\_proc\_info\_end,其中等号后面的"."是location counter。这三行的意思是: \_\_proc\_info\_begin 的位置上,放置所有文件中的 ".proc.info.init" 段的内容,然后紧接着是 \_\_proc\_info\_end 的位置.

kernel 使用struct proc\_info\_list来描述processor type.

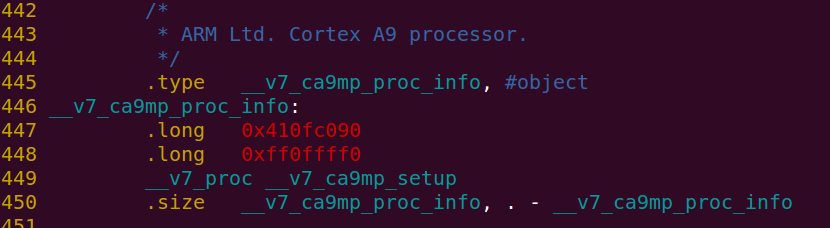
proc\_info\_list定义在arch/arm/include/asm/procinfo.h的29行。



具体每一种类型处理器都单独在自己对应的arch/arm/mm/proc-xxx.S汇编文件中填充实现。

我们这里是ARM Cortex A9，linux内核一般支持。Cpu\_info定义在arch/arm/mm/proc-v7.S

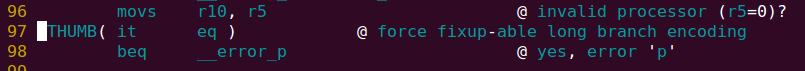
中



447行，就是proc\_info\_list结构体第一个成员 cpu\_val,448行是第二个成员cpu\_mask..

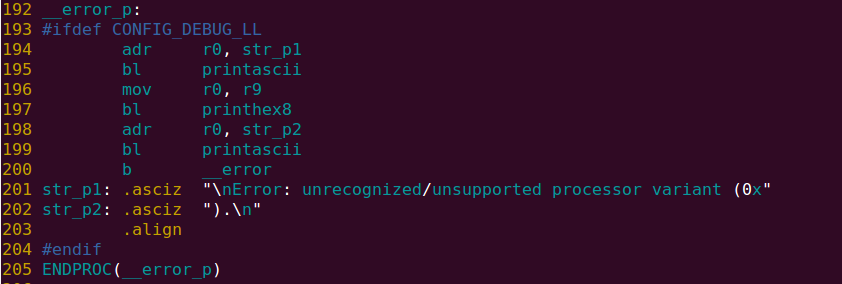
继续arch/arm/kernel/head-common.S 的158行~164行，这个小循环就是在不断比较的过程，如果找到，可以看到161行，直接跳到166行，返回，Cpu\_info\_list的地址放到r5寄存器中。如果没有找到，可以看到执行到165行，r5的值是0，然后退出\_\_lookup\_processor\_type。

回到arch/arm/kernel/head.S 96~98行，



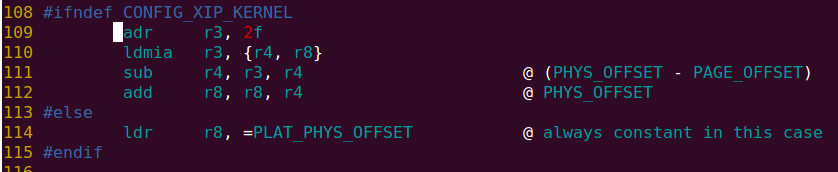
可以看到如果r5为0，则执行\_\_error\_p，否则处理器ID保存到r10寄存器。

\_\_error\_p子函数定义在arch/arm/kernel/head-common.S 文件 192行



只是往串口打印Error: unrecognized/unsupported processor variant一句话。

继续head.S,108行~115行，计算物理地址与虚拟地址的offset。



注意，我们这个时候还没有使能MMU，所以用的地址都是物理地址，所以要把所有变量的虚拟地址加上offset得到物理地址。Offset存到r8中。

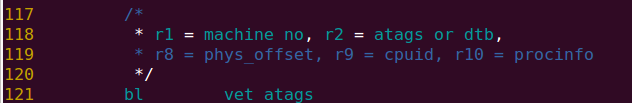
关于物理地址虚拟地址offset计算原理：

因为kernel要开启MMU，所以kernel编译链接地址是虚拟地址（物理地址经过MMU转换后CPU看到的地址），并不是物理地址，链接确定了变量的绝对地址（虚拟地址），但在现阶段，没开启MMU，CPU看到的sdram地址就是其物理地址（0x40000000起始）。  
 如果直接运行，对于变量的寻址则会出现问题（函数寻址没问题，因为arm函数寻址使用相对跳转指令b bl）比如，kernel image中全局变量i链接地址在0xc0009000，但现阶段i物理地址是在0x80009000，对于CPU来说，只能在0x80009000上才能找到i。  
 去0xc0009000寻址，程序运行就出错了。这就是为什么我们所理解的，链接地址 加载地址 运行地址必须一致的原因。

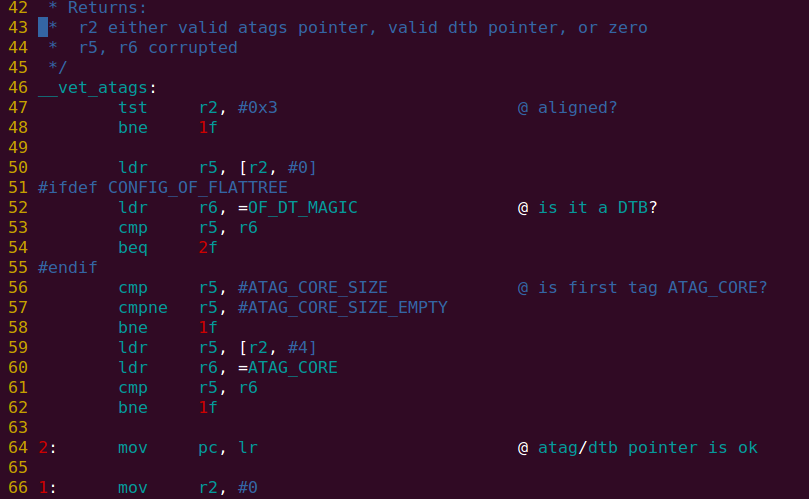
 kernel现阶段给出的解决方法，就前3行汇编：

adr r3， 2f @加载label 2地址（实际运行地址，这里就是物理地址）到r3  
ldmia r3， {r4 – r8} @获取以r3 r3+4 r3+8为地址的变量到r4，r5，r6.  
地址变量值是在链接时确定的，所以r4中存的是label 2的链接地址（虚拟地址）。  
sub r3 ，r3 ，r4     @r3中存储的是物理地址与虚拟地址的偏移。

121行，调到\_\_vet\_atags子代码段执行。



\_\_vet\_atags定义在arch/arm/kernel/head-common.S 的46行：



该函数主要检测bootlooader传递给kernel的atags的合法性。

每一个参数是以tag的形式传送的，包含三部分组成：

1.大小 2.类型 3.具体内容

47行~48行，检查atags地址是否是否4字节对齐

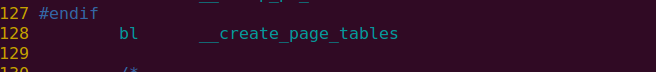
50~54行，检查校验r2中指向的 是否是有效dtb pointer 。如果读取的内容不是OF\_DT\_MAGIC则认为无效tags

56~62行，是检查ATGA\_CORE 的size和类型是否正确，如果第一个起始tag的size部分不等于ATAG\_COSE\_SIZE或者type不等于ATAG\_CORE则认为是失败的，这是bootloader和kernel约定好的。

我们知道linux内核最终在连接的 时候指定的都是虚拟地址，到目前为止我们一直用的都是物理地址，在使用一些符号的时候，由于这些符号都是虚拟地址，我们为了操作需要先计算出物理地址和虚拟地址的offset，然后再讲虚拟地址先转换成物理地址再用。每次这样，显然很麻烦，我们需要做好虚拟地址和物理地址之间的映射，然后打开MMU，由MMU自动完成虚拟地址和物理地址之间的转换。

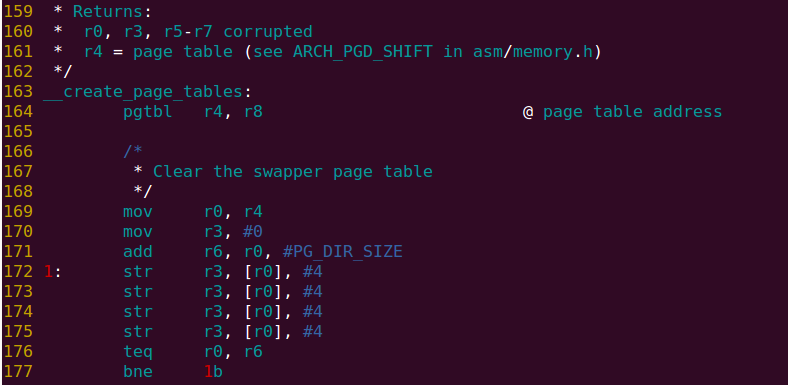
物理地址和物理地址之间的映射其实是构造页表的过程。

回到head.S继续看。128行跳到\_\_create\_page\_tables



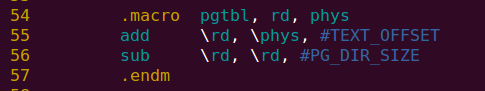
那么\_\_create\_page\_tables就是创建页表的过程，为后面开启MMU做准备。

看看具体实现，在head.S 的163行：



164行，调用pgtbl 这个宏来得到页表的物理地址。

宏的定义在head.S的上面



将phys+TEXT\_OFFSET-PG\_DIR\_SIZE得到的是页表的物理地址

其中phys传去的是r8，我们知道前面算出来虚拟地址和物理地址的offset是存在r8中。

TEXT\_OFFSET定义在arch/arm/Makefile 228行



表示内核image在内存中的位置与RAM起始地址的偏移值。

TEXT\_OFFSET := $(textofs-y)

所以看 $(textofs-y) 的值具体多少。

定义在Makefile



为0x00008000。

phys+TEXT\_OFFSET 算出来的是image在ram中存放的物理地址，

而PG\_DIR\_SIZE 是表示页表占用空间大小。定义在head.S中为0x4000，也即是16k。

Phys + TEXT\_OFFSET - PG\_DIR\_SIZE 表示页表起始地址，在r4中存放。

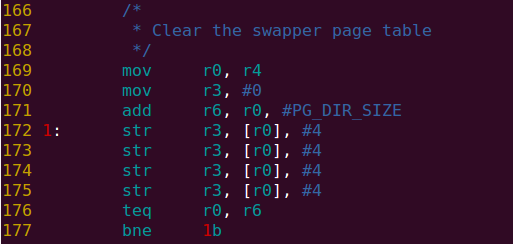
页表在image在ram存放位置下方16k空间存放。

为什么页表总共是16K呢？

映射是采取段式映射，每个段表示1M的虚拟地址空间，那么4G的虚拟地址空间需要4096个段，

而每一个虚拟地址空间的段需要占用4byte的物理空间来映射。所以一共需要

4096\*4byte = 16K的物理空间来做页表。



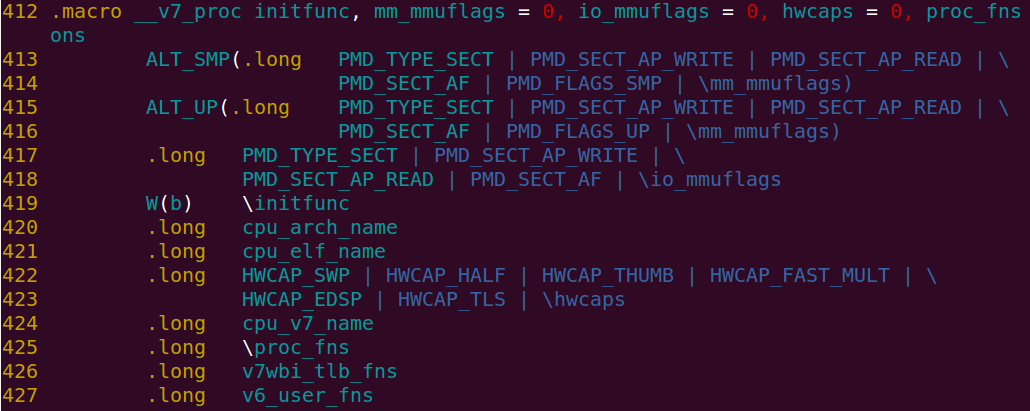
166行~177行，就是清除页表16K内存。

179行~205行，是如果定义CONFIG\_ARM\_LPAE，在PGD与PMD之前还要再加一级页表，这里不详解这种情景。



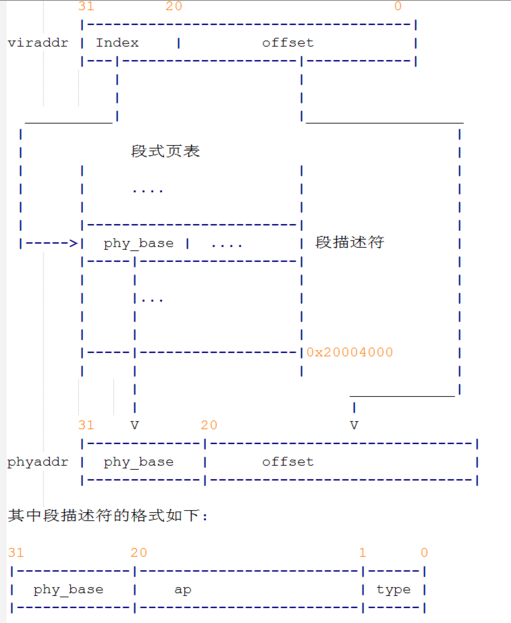
207行，获取CPU相关 proc\_info\_list的成员mm\_mmuflags.前面分析r10放的是proc\_info\_list地址。PROCINFO\_MM\_MMUFLAGS肯定是8，因为mmu\_flags是第3个成员。

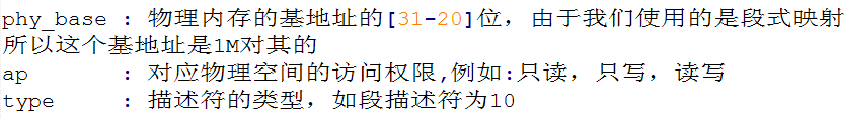
得到的mmuflags是在arch/arm/mm/proc-v7.S的417行~418行，



这些flags宏的含义是设置映射后的物理空间访问权限和页表中存放的描述符类型。

物理地址与虚拟地址通过MMU段式映射的原理：



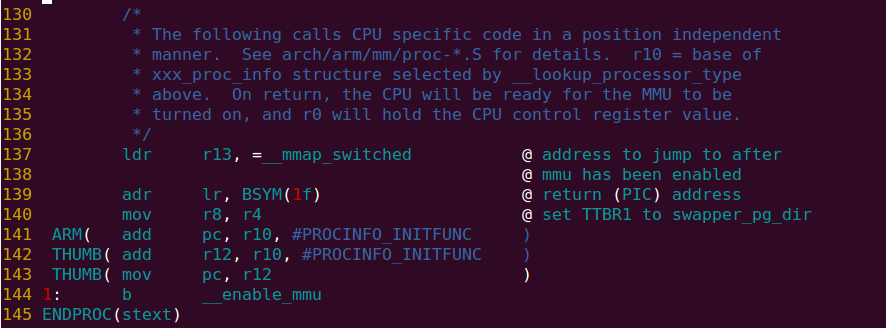


关于\_\_create\_page\_tables后面具体的代码在此就不在详细说了。

反正就是\_\_create\_page\_tables在内核代码区TEXT\_OFF下部的16KB区域内进行页表的配置，完成turn\_mmu\_on的平映射以及kernel image的线性映射。

接下来就需要开启MMU，让整个CPU进入虚拟地址运行的新阶段。

回到head.S的130行，继续看后面。



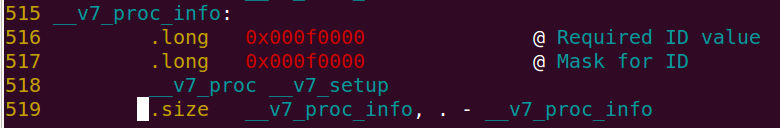
如130~136行注释所讲接下来要完成的2件工作：执行CPU特定处理代码，开启MMU。

137行，r13存下\_\_mmap\_switched的地址。

139行，把label 1的地址给lr

141行，pc指针的值修改为r10+PROCINFO\_INITFUNC,

前面分析我们知道，r10保存的 是proc\_info\_list基地址，其中PROCINFO\_INITFUNC的值是16。所以 r10+PROCINFO\_INITFUNC肯定指向的是proc\_info\_list地址加+16地址处。



而armv7处理器proc\_info定义在汇编文件 arch/arm/mm/proc-v7.S 的515行，

其中16 offset是指向正是 \_\_v7\_setup label。\_\_v7\_setup具体实现我们就不看了，

主要做了以下事情:

1. 刷新cache，v7\_flush\_dcache\_all
2. 无效掉cache
3. 无效掉指令和数据TLB

TLB(Translation Lookaside Buffer)传输后备缓冲器是一个内存管理单元用于改进虚拟地址到物理地址转换速度的缓存。

TLB是一个小的，虚拟寻址的缓存，其中每一行都保存着一个由单个PTE组成的块。如果没有TLB，则每次取数据都需要两次访问内存，即查页表获得物理地址和取数据。  
（4）从CP15 c0寄存器读出到r0，进行处理，做为下一步写入CP15 c0的值

回到head.S。

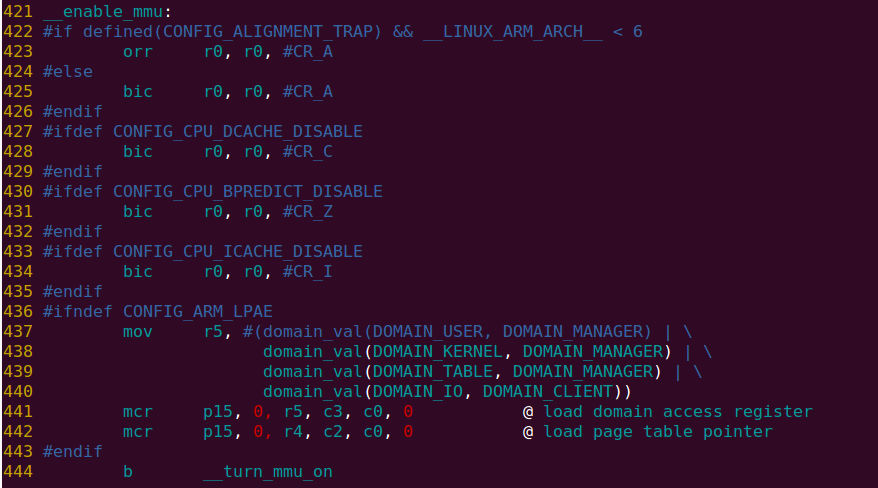
前面分析中我们知道调到\_\_v7\_setup之前，lr寄存器保存了label 1f的地址，

而label 1f处，是b \_\_enable\_mmu



所以下面我们来看看 \_\_enable\_mmu具体实现。

\_\_enable\_mmu主要是实现MMU的开启。

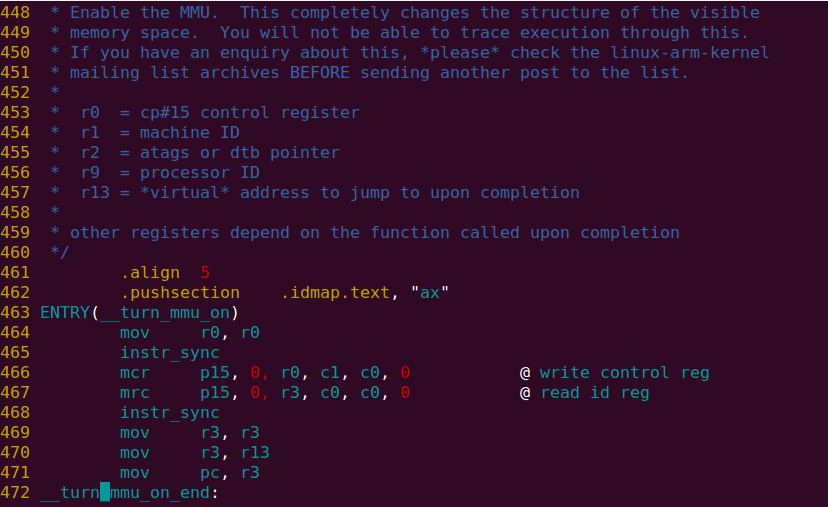


还是在head.S,421行，

该函数主要是437行到444行，

437行~442，先设置了一些MMU的参数，

444行，跳到\_\_turn\_mmu\_on执行。



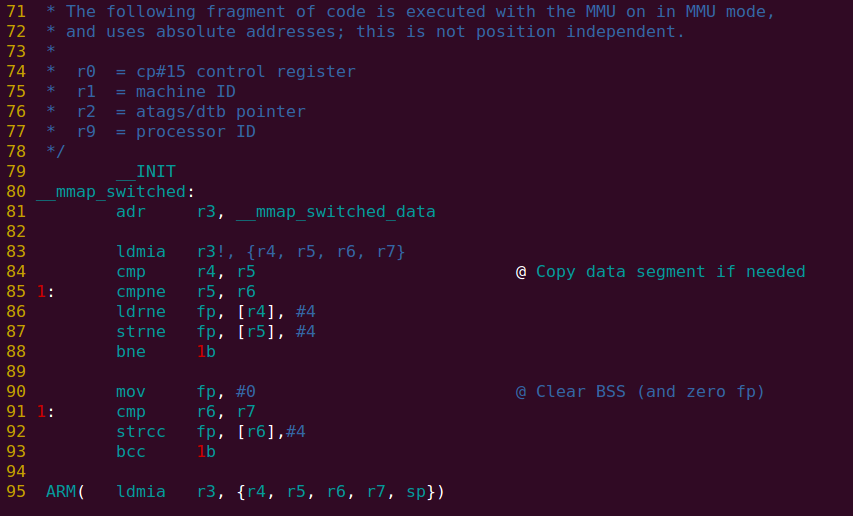
开启MMU主要是设置协处理器cr0。该函数将r0值写入CP15的c0寄存器，查看CP15说明，c0寄存器的第0位表征MMU的开启，刚才\_\_v7\_setup中v7\_crval给出的mmuset值已经将第0位置1了，因此写入c0后MMU就开启了！

471行，将pc修改为r3，而r3等于r13，前面分析我们知道， r13存下的是\_\_mmap\_switched的地址。

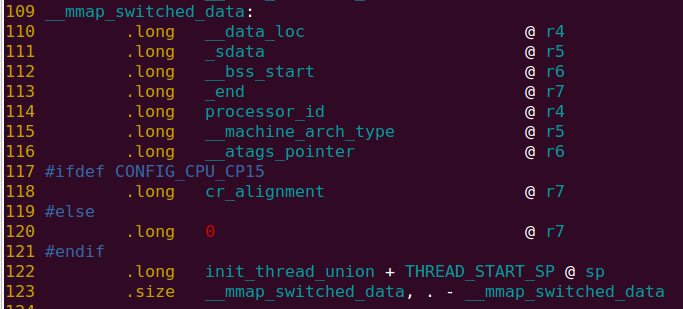
所以程序跳到\_\_mmap\_switched子程序段执行。定义在arch/arm/kernel/head-common.S

80行。

那么从\_\_mmap\_switched开始的指令又有MMU已经开启，所以是虚拟地址使用。



81行，r3存放\_\_mmap\_switched\_data地址。\_\_mmap\_switched\_data定义在109行。



所以r4 = \_data\_loc

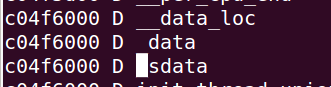
R5 = \_sdata

r6 = \_\_bss\_start

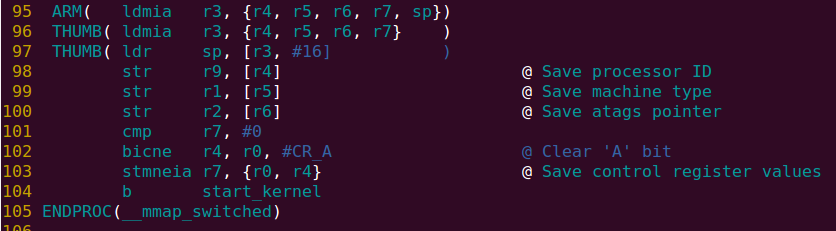
r7 = \_end

84行，如果 r4和r5不相等，则需要copy数据。

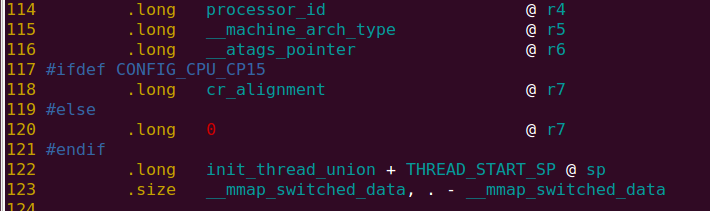
查看System.map \_data\_loc和\_sdata相等，都是c04f6000，所以不需要copy转移。



90~93行，清除bbs段。



95行，由于前面r3更新，r3指向114行，



所以95行指令执行之后，

R4 = processer\_id

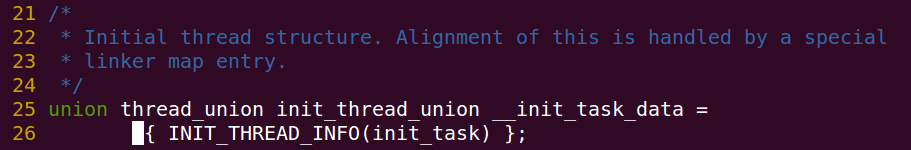
R5 = \_\_machine\_arch\_type

R6 = \_\_atags\_pointer

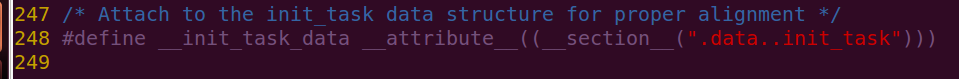
R7 = cr\_alignment

SP = init\_thread\_union + THREAD\_START\_SP

init\_thread\_union代表的是内核第一个进程，pid = 0。在init/init\_task.c中定义了它。



其中是一个宏。定义在include/linux/init\_task.h中。



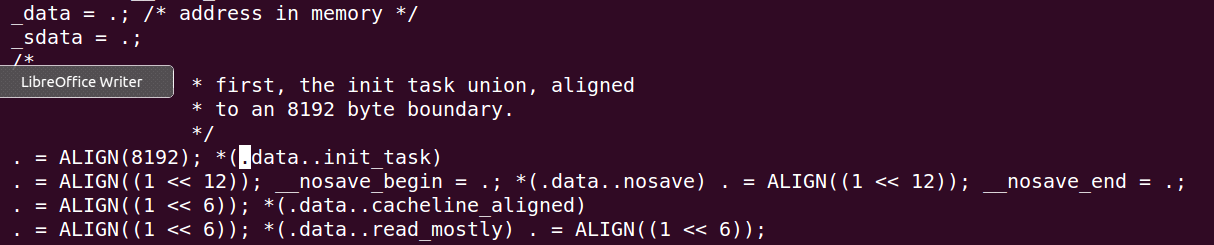
那么就等价于

union thread\_union init\_thread\_union \_\_attribute\_\_((\_\_section\_\_(".data..init\_task"))) =

{ INIT\_THREAD\_INFO(init\_task) };

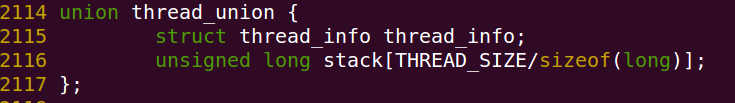
将init\_thread\_union联合体强制放到.data..init\_task段，

由arch/arm/kernel/vmlinux.lds链接脚本知道，该段位于data段起始位置，而且是8K对齐。



这就和union thread\_union 大小一致。表示保存一个进程信息。

thread\_union定义在include/linux/sched.h中。



可以看出主要两部分：

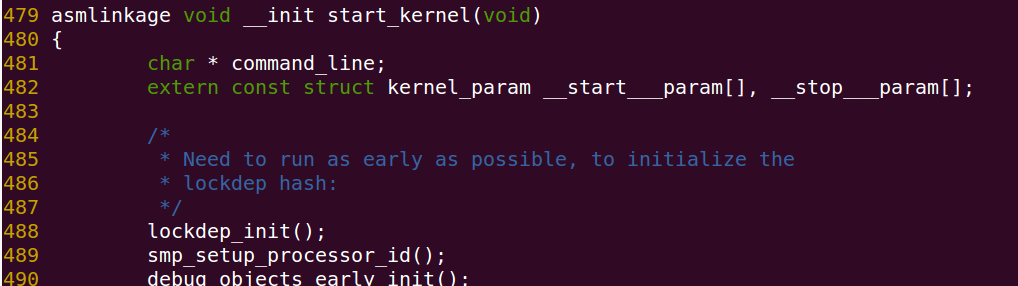
1. 作为struct thread\_info结构体空间
2. 作为内核栈空间。

这样SP设置好，为后面运行C语言做准备。

98~100,就是将processer id，mach-type和atags pointer分别放到对应的段。

104行，无返回的 跳到 start\_kernel中执行。由此汇编部分结束，剩下的都是C语言实现。

start\_kernel函数的实现在init/main.c 479行。



Start\_kernel函数主要是做了一系列的初始化。这里初始化函数就不去一个个细究了。

下面我列出了里面主要的初始化函数的定义位置。

start\_kernel( ) {

printk(linux\_banner);

setup\_arch(&commaind\_line); - arch/arm/kernel/setup.c

setup\_processor()‏

mdesc = setup\_machine\_fdt(\_\_atgs\_pointer, \_\_machine\_arch\_type);

mdesc = setup\_machine\_tags(\_\_atags\_pointer, \_\_machine\_arch\_type);

setup\_command\_line(command\_line); - init/main.c

trap\_init(); - arch/arm/kernel/traps.c

init\_IRQ(); - arch/arm/kernel/irq.c

sched\_init(); - kernel/sched.c

softirq\_init(); - kernel/softirq.c

time\_init(); - arch/arm/kernel/time.c

console\_init(); - drivers/char/tty\_io.c

init\_modules(); - kernel/module.c

kmem\_cache\_init(); - mm/slab.c

sti(); /\* interrupt first open \*/

calibrate\_delay(); - init/main.c

mem\_init(); - arch/arm/mm/init.c

……

printk("POSIX conformance testing by UNIFIX\n");

smp\_init(); - init/main.c

rest\_init(); - init/main.c

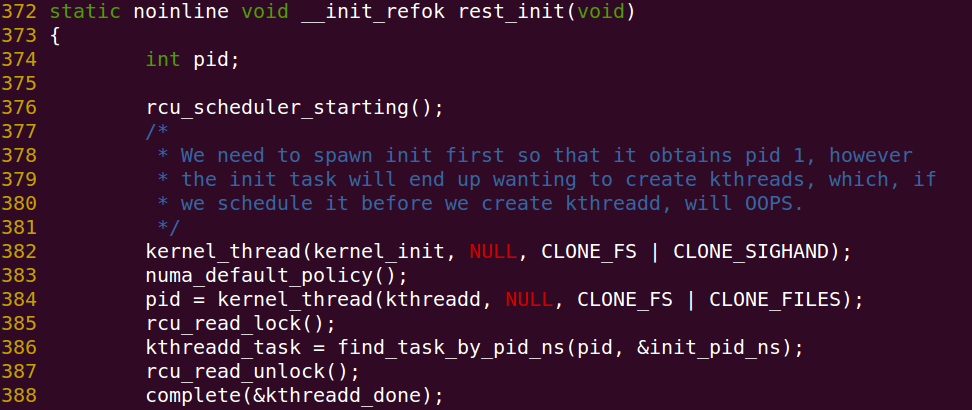
kernel\_thread(kernel\_init,NULL, CLONE\_FS | CLONE\_SIGNAL)

}

在start\_kernel函数的最后，651行，调用了函数rest\_init函数。

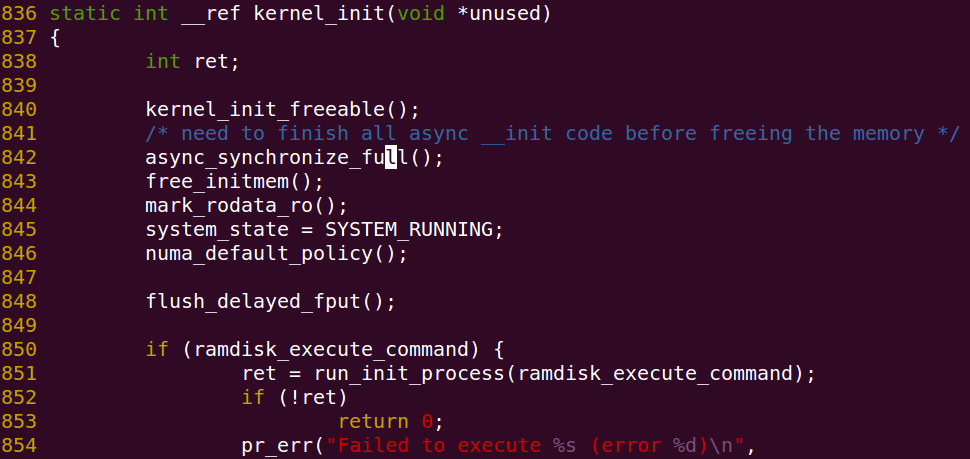


Rest\_init函数也定义在init/main.c 372行。



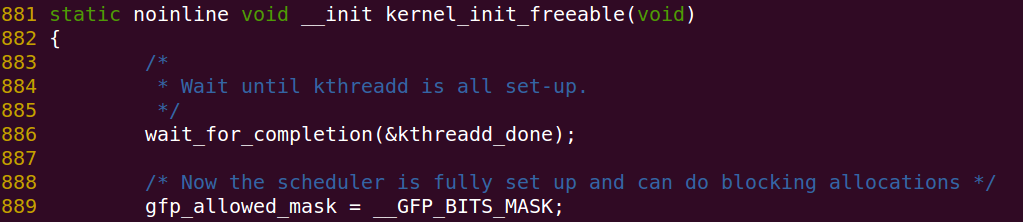
382行，可以看到创建一个kernel\_init线程。

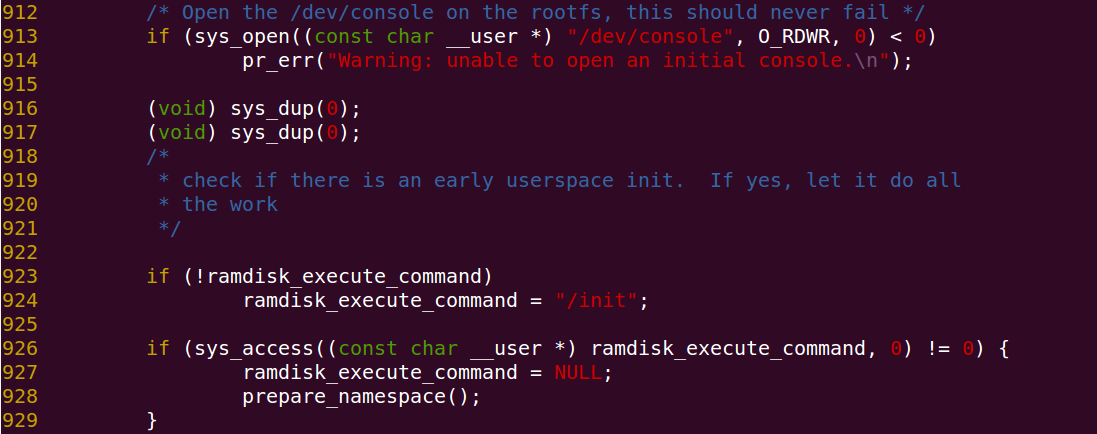
我们看看kernel\_init函数。还是定义在本文件 836行。



Kernel\_init函数中第 840行，调用到kernel\_init\_freeable函数，我们看看这个函数。

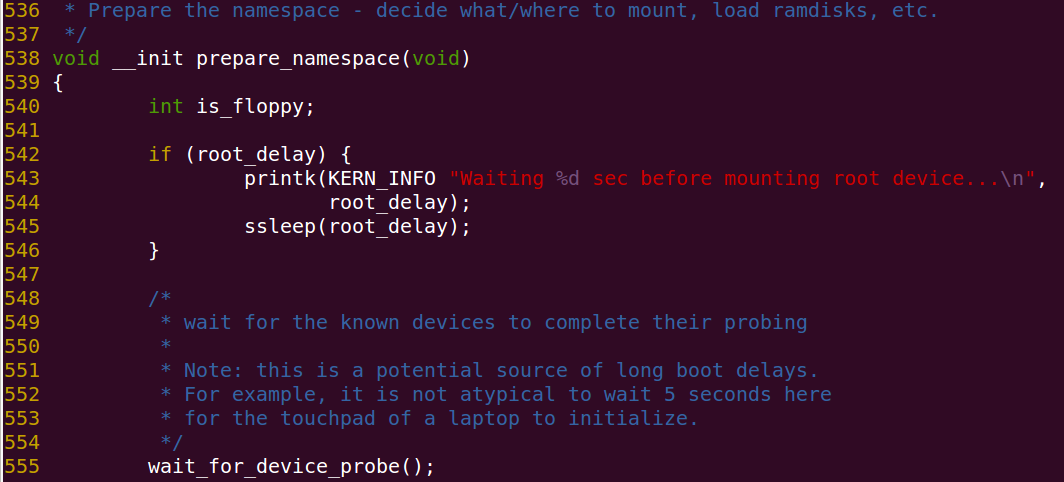
在881行定义。



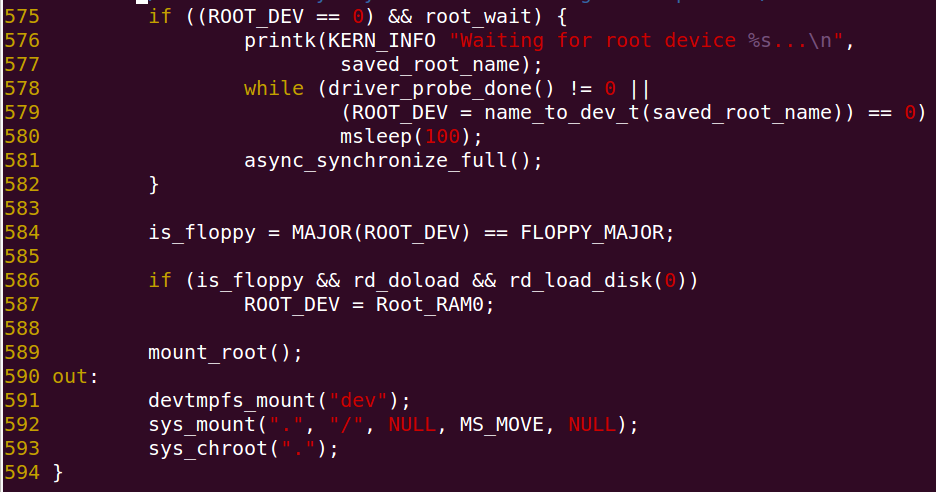


该函数924行，注意ramdisk\_execute\_command 字符串被赋值 “/init”，这个后面会用到。

928行，调用函数prepare\_namespace函数，该函数定义在init/do\_mount.c中538行。

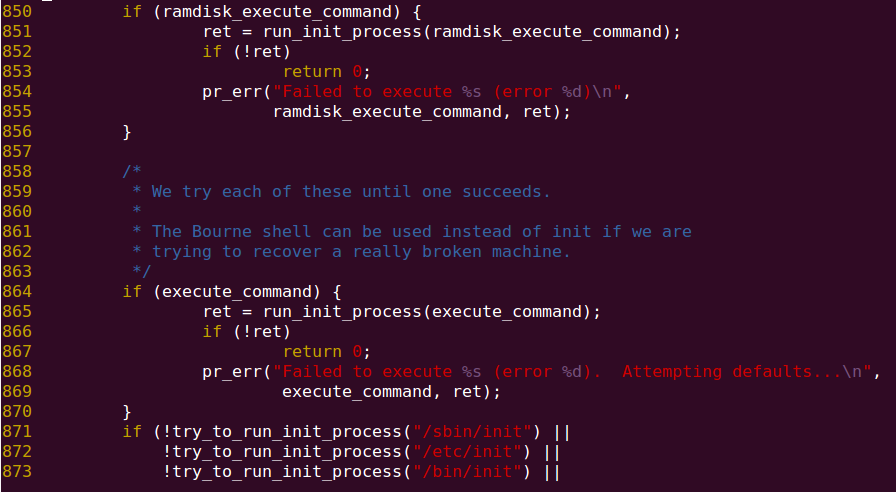


主要看该函数后面589挂载根文件系统。



挂在了根文件系统在prepare\_namespace函数中，这样根文件系统就挂载了。

回到init/main.c kernel\_init函数。850行~873行。



850行，我们知道根据前面ramdisk\_execute\_command是“/init”

851行，运行init进程，注意，是运行的/init,根目录下的init可执行程序。

如果失败，接着 865行，运行execute\_command，这个根据“init=”得到，由bootloader传给。

871~873行，试着去执行/sbin/init,/etc/init,/bin/init下面的init可执行程序。

这样init进程启动完成。

总结一下整个过程就是：

（1）检查CPU SVC 模式 ,屏蔽中断

（2）检查linux 内核是否支持 CPUID . lookup\_processer\_type  
（3）获取phy&virt offset，为了后面算虚拟地址对应物理地址  
（4）检查atags参数或者dtb poniter有效性

（5）create\_page\_table完成了3种地址映射的页表空间填写：

（6）使能MMU，MMU硬件会自动根据页表将虚拟地址转换成物理地址

（7）如果有必要则进行data段的拷贝搬移   
（8）清空bss段   
（9）保存处理器ID ，atag地址等到指定全局变量中，设置SP，跳转到start\_kernel

（10）初始化系统必须要的好多东西

（11）挂载根文件系统 mount\_root

（12）运行init进程