Linux 2.6 内核启动分析

华清远见深圳中心 学员 李枝果 lizhiguo0532@163.com 2010-6-04

特别说明：

此文章的原始出处 http://www.docin.com/sz_farsight

此文章是 **华清远见 深圳培训中心** 学员在学习期间的兴趣专题文章，文章的部分内容可能参考了网络资源而没有标明出处，敬请见谅。此文档欢迎转载，但转载请**保留学员个人信息**及**此段说明**，谢谢！

特别说明之后的一般说明 ^_^：

本文的分析是基于之前分析 SD 卡驱动的内核。

本文由 **深圳 华清远见** 学员**李枝果**总结，易松华老师批注

一、Makefile 是如何生成 uImage 的

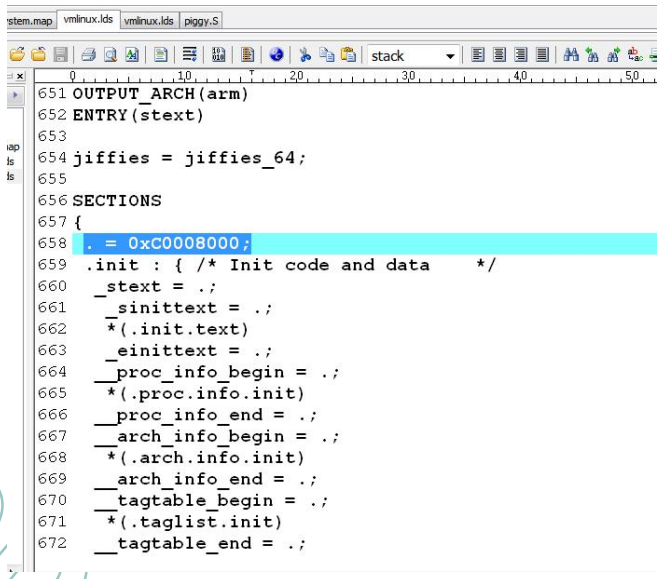
在研究内核之前，首先我们要认识内核镜像的结构。顶层的 Makefile 生成 uImage 是按照下面的几个步骤生成的（这里省略了各子目录下*.o 文件的生成）

1. arm-linux-gnu-ld 用 arch/arm/kernel/vmlinux.lds 、arch/arm/kernel/head.o、arch/arm/kernel/init_task.o、各子目录下的 built-in.o、lib/lib.a 、arch/arm/lib/lib.a 生成顶层目录下的 vmlinux （根据 arch/arm/kernel/vmlinux.lds 来链接 0xc0008000）

   vmlinux.lds 开始处

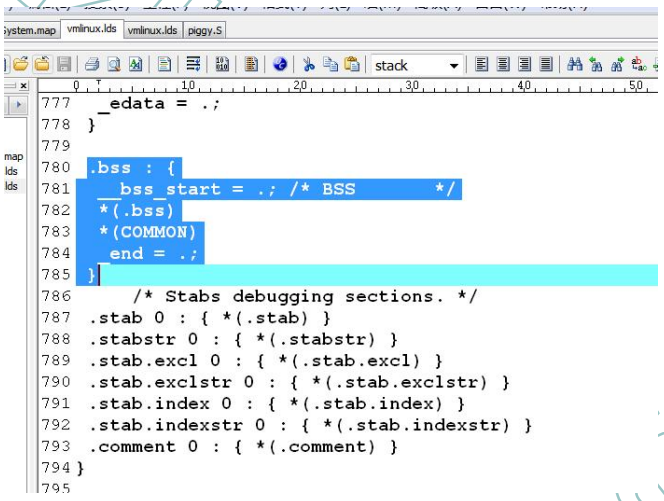
批注 [s1]: ELF 格式文件，这个是原汁原味，不添加任何防腐剂的真正 kernel，是 linux 杀手级的东东^_^

批注 [s2]: 让初学者伤透脑筋的东东，如知详情，先插播广告： 来“深圳华清远见”学习，先，哈



```
651 OUTPUT_ARCH(arm)
652 ENTRY(stext)
653
654 jiffies = jiffies_64;
655
656 SECTIONS
657 {
658  . = 0xC0008000;
659  .init : { /* Init code and data     */
660   _stext = .;
661    _sinittext = .;
662    *(.init.text)
663    _einittext = .;
664   __proc_info_begin = .;
665    *(.proc.info.init)
666   __proc_info_end = .;
667   __arch_info_begin = .;
668    *(.arch.info.init)
669   __arch_info_end = .;
670   __tagtable_begin = .;
671    *(.taglist.init)
672   __tagtable_end = .;
```

vmlinux.lds 结束处



```
777   _edata = .;
778  }
779
780  .bss : {
781    __bss_start = .; /* BSS        */
782    *(.bss)
783    *(COMMON)
784    end = .;
785  }
786       /* Stabs debugging sections. */
787  .stab 0 : { *(.stab) }
788  .stabstr 0 : { *(.stabstr) }
789  .stab.excl 0 : { *(.stab.excl) }
790  .stab.exclstr 0 : { *(.stab.exclstr) }
791  .stab.index 0 : { *(.stab.index) }
792  .stab.indexstr 0 : { *(.stab.indexstr) }
793  .comment 0 : { *(.comment) }
794 }
795
```

2. 生成 system.map，置于顶层目录之下
3. arm-linux-gnu-objcopy，去掉顶层 vmlinux 两个段-R .note -R .comment 的调试信息，减小映像文件的大小，此时大概 3.2M，生成 arch/arm/boot/Image
4. gzip -f -9 < arch/arm/boot/compressed/../Image > arch/arm/boot/compressed/piggy.gz，读入 arch/arm/boot/Image 的内容，以最大压缩比进行压缩，生成 arch/arm/boot/compressed/目录下的 piggy.gz
5. arm-linux-gnu-gcc，在 arch/arm/boot/compressed/piggy.S 文件中是直接引入 piggy.gz 的内容（piggy.gz 其实已经是二进制数据了），然后生成 arch/arm/boot/compressed/piggy.o 文件。下面是 piggy.S 的内容

批注 [s3]: 内核符号表，如果你想调试内核或者查找出错倪端，多记得关注这个“美女”^_^

批注 [s4]: 此时转换成了 BIN 格式文件

批注 [s5]: 通过 incbin 关键字引入的，sorry,你后面有说啊，重复了哈

```
1   .section .piggydata,#alloc
2   .globl  input_data
3 input_data:
4   .incbin "arch/arm/boot/compressed/piggy.gz"
5   .globl  input_data_end
6 input_data_end:
7
```

其中所选择的行就是加入了 piggy.gz 的内容，通过编译生成 piggy.o 文件，以备后面接下来的 ld 链接。

6. arm-linux-gnu-ld，在 arch/arm/boot/compressed/piggy.o 的基础上，加入重定位地址和参数地址的同时，加入内核解压缩的代码（arch/arm/boot/compressed/head.o 、 misc.o ），最后生成 arch/arm/boot/compressed 目录的 vmlinux，此时在解压缩代码中还含有调试信息（根据 arch/arm/boot/compressed/vmlinux.lds 来链接 0x0）

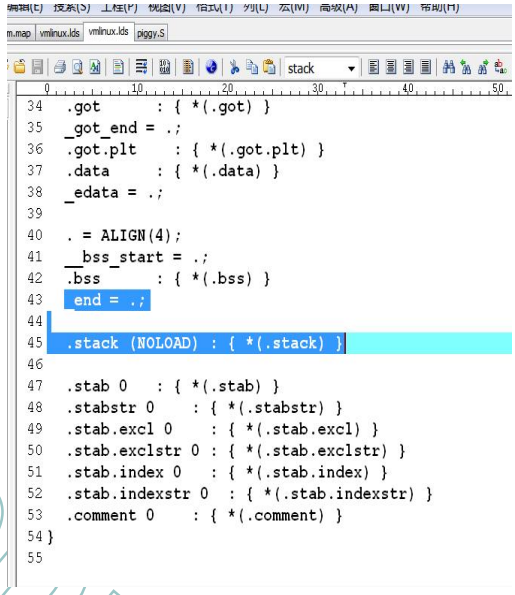
批注 [s6]: 还能说啥呢，从事解压缩这么辛苦又不受内核待见的工作就是他们了（因为进真正内核之后就不用他们了，为他人做嫁衣裳啊，其人生何其惨烈!，但我们还是很想听他悲惨的人生历程来乐呵乐呵，歹毒吧^_^）

vmlinux.lds 开始处

```
10 OUTPUT_ARCH(arm)
11 ENTRY(_start)
12 SECTIONS
13 {
14   . = 0;
15   _text = .;
16
17   .text : {
18     _start = .;
19     *(.start)
20     *(.text)
21     *(.fixup)
22     *(.gnu.warning)
23     *(.rodata)
24     *(.rodata.*)
25     *(.glue_7)
26     *(.glue_7t)
27     *(.piggydata)
28     . = ALIGN(4);
29   }
```

注意到了 27 行的吗？*(.piggydata)就表示需要将 piggydata 这个段放在这个位置，而 piggydata 这个段放的是什么呢？往后翻翻，看看第五步的图片，呵呵，其实就是将按最大压缩比压缩之后的 Image，压缩之后叫 piggy.gz 中的二进制数据。

再看看 vmlinux.lds 的结束处



注意到了 .stack 放在什么位置了吗？放在了 arch/arm/boot/compressed/vmlinux 的 bss_end 之后后，也就是说放在了 piggy.gz 代码段的后面。清楚这一点很重要，这关系到后边理解如何分布堆栈和堆，以及解压的地址空间分配。

7. arm-linux-gnu-objcopy，去掉解压缩代码中的调试信息段，最后生成 arch/arm/boot/目录下的 zImage
8. /bin/sh /home/farsight/Resources/kernel/linux-2.6.14/scripts/mkuboot.sh -A arm -O linux -T kernel -C none -a 0x30008000 -e 0x30008000 -n 'Linux-2.6.14' -d arch/arm/boot/zImage arch/arm/boot/uImage
   调用 mkimage 在 arch/arm/boot/zImage 的基础上加入 64 字节的 uImage 头，和内核的入口地址，装载地址， 最终生成 arch/arm/boot/目录下的 uImage 文件。

批注 [s7]: 这个说法欠妥，虽然快毕业了，但你还得跟我学哈，应该是放在了.BSS 段后面，由于链接的时候 BSS 实际数据为 0，所以严格来说，应该是链接的时候放在了整个内核压缩包（包括解压缩代码和内核本身压缩镜像后面）

批注 [s8]: 此时是 bin 格式文件

批注 [s9]: 嗯，如果你们班还有谁不懂这个，叫他来找我，我得严肃地，语重心长地，苦口婆心地和他谈心了，嘿嘿

由上可知，zImage 中包含了内核的自解压代码（head.o misc.o），所以在 uboot 将 64 字节的 uImage 头去掉了之后，内核实际上就变成了一个 zImage，所以接下来的工作首先会进行内核的自解压。

需要注意的是，内核代码（顶层 vmlinux-→arch/arm/boot/Image-→arch/arm/boot/compressed/piggy.gz-→arch/arm/boot/compressed/piggy.o）在编译链接时的地址是 0xc0008000，而自解压代码(arch/arm/boot/compressed/vmlinux-→ arch/arm/boot/zImage)的链接地址却是 0x0，所以在接下来的代码分析中所有的地址修正都是从 0x00000000（自解压代码的链接地址 0x0）开始的地址修正到 0x30008000 处，直到自解压代码执行完成，开始进入真正内核（由 Image 解压成 vmlinux 之后的链接地址是在 0xc0008000 开始处）

二、内核自解压代码分析

下面分析的是自解压代码，其链接地址是 0x0

一、 arch/arm/boot/compressed/head.s 和 misc.c 文件分析

1. 首先来一小段延时，接着保存由 uboot 的 thekernel 传递进来的三个参数中的前两个，r0—>r8,r1-→r7.
2. 导入 LC0 数据对象的数据到一系列的寄存器中，同时计算出这一系列链接地址值（0x0） 到当前空间 0x30008000 的偏移量，如下

```
    .text
    adr   r0 , LC0   /* 取得数据对象LC0的相对当前运行地址的地址 ，
                        也就是当前运行空间的地址0x30008000, 而非编译的地址0xc0008000*/
    ldmia   r0 , { r1 , r2 , r3 , r4 , r5, r6, ip , sp }
    subs    r0 , r0 , r1        @ calculate the delta offset

                    @ if delta is zero, we are
    beq not_relocated       @ running at the address we
                    @ were linked at.

    , ,
```

```
    .type LC0, #object    @定义了一个数据对象，有点类似于C中的结构体
LC0:  .word LC0       @ r1
    .word __bss_start   @ r2
    .word _end      @ r3
    .word zreladdr      @ r4  zreladdr 根据注释是virt_to_phys(TEXTADDR), 一
    .word _start        @ r5
    .word _got_start    @ r6
    .word _got_end      @ ip
    .word user_stack+4096   @ sp
LC1:    .word reloc_end - reloc_start
    .size LC0, . - LC0
```

3. 接下来对导入到各个寄存器中的值（链接时确定，基于链接地址 0x0）修正到当前运行空间 0x30008000，因为 CONFIG_ZBOOT_ROM 没定义，所以修正了 r2，r3，r5，r6，ip，sp 到当前运行空间，另外也对 r6，ip 两寄存器值所代表的 got（全局偏移表）区间中的所有函数入口 进行地址修正。
4. clear bss
5. 为调用解压函数而准备 C 语言运行环境，包括打开 cache，堆空间的分配堆栈空间是利用下面的语句来分配了一个 4K 的空间

```
.align
    .section ".stack", "w"
user_stack:    .space 4096
```

```
    .type LC0, #object    @定义了一个数据对象，有点类似于C中的结构体
LC0:  .word LC0       @ r1
    .word __bss_start   @ r2
    .word _end      @ r3
    .word zreladdr      @ r4  zreladdr 根据注释是virt_to_phys(TEXTADDR), 一般来说zreladdr = 0x30008
    .word _start        @ r5
    .word _got_start    @ r6
    .word _got_end      @ ip
    .word user_stack+4096   @ sp
LC1:    .word reloc_end - reloc_start
    .size LC0, . - LC0
```

```
bl  cache_on            /* 开启指令和数据Cache */

mov r1, sp              @ malloc space above stack
add r2, sp, #0x10000    @ 64k max
//在堆栈之后分配64K的临时堆空间
```

综上，可以看出，stack 的空间是直接安排在了 zImage 代码的_end 的后面，大小为 4K（具体的地址大小还需要看内核是多大），接着在 stack 的后面紧接着分配了 64K 的临时堆空间，用来进行自解压的实现。

这里需要特别说明一下的是 cache_on 函数中，会打开 mmu，并且会做一些页表初始化的工作，然后在该 head.S 结束的时候会关闭 mmu 和 cache。在需要注意的是在这里初始化的页表并没有真正的使用价值，只是为了打开解压缩代码使用 D-cache 而打开的，所以这里的页表中映射的虚拟地址和物理地址的关系式等价映射的，也就是 1:1 映射，也就是说将物理地址映射到等值的虚拟地址上去，这里的页表就是这样映射出来的。详细的注释可以参考

我的另一篇文章 ..compressed.head.S-v2-cache_on 注释

6. 根据 zImage 解压之后的 Image 入口地址 0x30008000（final kernel address），和 zImage 目前的运行地址（zImage 链接地址是 0x0，但是其前面的自解压代码是地址无关的，所以理论上可以放在 SDRAM 的任意空间，我们这里也是属于通常的情况，在 uboot 中将 uboot 头去掉后的 zImage 放在了 0x30008000 地址上），分三种情况来判断是否留有足够的存放 Image 的空间（4M 足够了，原则是尽可能得直接将 Image 解压到 0x30008000 地址上，避免了解压到其他地址之后再将代码搬运到 0x30008000 地址上来运行的繁琐和时间浪费）。不过通常情况下，kernel（Image）的入口地址和 uImage 的 load 地址或入口地址都设置成了相同的值 0x30008000。

那么关于 Image 的入口地址 zreladdr 和 uImage 的 load 地址、入口地址如何设置呢？

Zreladdr 值在将内核编译链接成 vmlinux 的时候已经确定了，但是这个值在哪里设定的呢？其实他是在 arch/arm/mach-s3c2410/Makefile.boot 定义的值，下图所示：

```
   zreladdr-y    := 0x30008000
params_phys-y    := 0x30000100
```

同时也看到了内核中约定的参数存放地址在哪里定义了，这里定义了之后，这个变量在哪里引用的呢？arch/arm/boot/Makefile

批注 [s10]: SP 指针指向了这个 4K 字节的顶端

批注 [s11]: 后面 C 代码 misc.c 的里面的 malloc()函数就是从这里分配内存的，类似我们平常说的堆栈中的堆，从这里可以看到，从内存布局来说，内核解压缩时候，“堆”在“栈”的上面，是不是有种“一切尽在掌握”，自己成了上帝的感觉 （我们日常生活中“顾客就是上帝”一般都是喊的口号，但你跟代码打交道嘛，我们还是可以真正做回“上帝”的，嘿嘿）？

批注 [s12]: 心真细啊，明显感觉“长江后浪推前浪，前浪死在沙滩上”啊，这个我可是没提到啊，怕是那天我要在沙滩上啊，好怕怕^_^

```
# Note: the following conditions must always be true:
#   ZRELADDR == virt_to_phys(TEXTADDR)
#   PARAMS_PHYS must be within 4MB of ZRELADDR
#   INITRD_PHYS must be in RAM
ZRELADDR    := $(zreladdr-y)
PARAMS_PHYS := $(params_phys-y)
INITRD_PHYS := $(initrd_phys-y)

export ZRELADDR INITRD_PHYS PARAMS_PHYS
```

在 arch/arm/boot/compressed/Makefile 文件中有下面片段，看英文注释的意思就是，将 ZRELADDR 通过链接符号提供给解压程序使用。

```
# Supply ZRELADDR, INITRD_PHYS and PARAMS_PHYS to the decompressor via
# linker symbols.  We only define initrd_phys and params_phys if the
# machine class defined the corresponding makefile variable.
LDFLAGS_vmlinux := --defsym zreladdr=$(ZRELADDR)
ifneq ($(INITRD_PHYS),)
LDFLAGS_vmlinux += --defsym initrd_phys=$(INITRD_PHYS)
endif
ifneq ($(PARAMS_PHYS),)
LDFLAGS_vmlinux += --defsym params_phys=$(PARAMS_PHYS)
endif
LDFLAGS_vmlinux += -p --no-undefined -X \
    $(shell $(CC) $(CFLAGS) --print-libgcc-file-name) -T
```

最后将顶层的 vmlinux 在最大压缩比压缩之后得到了 Image。

uImage 的 load 地址和入口地址在哪里设置的呢？其实应该是 zImage 的 load 地址和入口地址的设定。因为 uImage 只不过是在 zImage 的前面加了一个 uboot 头，这个头里面保存了 zImage 的 load 地址和 entry 地址。

怎么设定的是需要追寻到 uImage 的生成过程了，在得到 zImage 之后，通过工具 mkimage 才生成 uImage 的，但是是否还记得当时是传递了很多参数的呢？下图

```
  /bin/sh /home/farsight/Resources/kernel/linux-2.6.14/scripts/mkuboot.sh -A arm -O linux -T kernel -C none -a 0x30008000 -e
0x30008000 -n 'Linux-2.6.14' -d arch/arm/boot/zImage arch/arm/boot/uImage
//在zImage的基础上加入64字节的uImage头，和入口地址，装载地址，调用mkimage 最终生成arch/arm/boot/目录下的uImage文件
```

光标选中的部分是不是传递了两个参数-a 表示 data load 地址，-e 表示的就是 entry 地址。但是这个值是在哪里设定的呢？arch/arm/boot/Makefile 中，下图选中部分就是

```
quiet_cmd_uimage = UIMAGE  $@
      cmd_uimage = $(CONFIG_SHELL) $(MKIMAGE) -A arm -O linux -T kernel \
           -C none -a $(ZRELADDR) -e $(ZRELADDR) \
           -n 'Linux-$(KERNELRELEASE)' -d $< $@
```

好，现在知道了这两种地址是什么意思了吧！可以看出，通常这三个地址是一样的，都是 0x30008000，那接着往下分析，先贴出代码：

```
/*
 * Check to see if we will overwrite ourselves.
 *   r4 = final kernel address
 *   r5 = start of this image
 *   r2 = end of malloc space (and therefore this image)
 * We basically want:
 *   r4 >= r2 -> OK
 *   r4 + image length <= r5 -> OK
 */
        cmp r4, r2      /* r4 = 0x30008000 ,
                        r2 = (zImage+bss size)+stack size 4K + malloc size 64K */
        bhs wont_overwrite
        add r0, r4, #4096*1024      @ 4MB largest kernel size
        cmp r0, r5      /* r0= 0x30400000 , r5 = 0x30008000 */
        bls wont_overwrite

        mov r5, r2              @ decompress after malloc space
        mov r0, r5
        mov r3, r7              @ r0  = malloc end or decompress space,
                                @ r1  = sp  end or malloc begin
                                @ r2  = malloc end , r3  = architecture ID
        bl  decompress_kernel

        add r0, r0, #127
        bic r0, r0, #127        @ align the kernel length
/*
```

上图中 r4 代表 Image 内核的真正入口地址 0x30008000，r5 代表 zImage 的入口地址，r2 代表 zImage 的堆空间结束地址。

下面分三种情况来看这段代码是如何判定空间之间是否有覆盖区域的。

第一种情况：r4>=r2,假如 r4=0x30800000（也就是 Image 的入口地址），r2=0x30008000+(zImage+bss size)+stack size 4K + malloc size 64K 这中情况 zImage 区域没有覆盖掉解压后 Image 需要的空间，所以可以直接解压到 0x30800000 中去，也省去了解压后代码搬移的过程。

第二种情况：r4+4M<r5,假如 r5=0x30800000，r4=30008000。这种情况下也是可以直接将 zImage 解压到 r4 空间的。同第一种情况，这也不用解压后搬移代码。

第三种情况：也是我们普遍使用的情况。r4+4M>r5,就拿我们正常的地址值来说。此时 r4=r5=0x30008000，这样就得需要另一种处理方式，先将内核解压到其他空闲的地址，然后将解压后的代码搬运到 0x30008000 处。

```
@ r0 = malloc end or decompress space,
@ r1 = sp end or malloc begin,
@ r2 = malloc end ,
@ r3 = architecture ID
```

上面几个寄存器的值都是需要传递给 decompress_kernel 函数的，该函数将内核解压到 malloc 空间之后去了。之后可定要将解压后的代码搬移会到 0x30008000 处运行。

7. 调用 decompress_kernel 函数，in arch/arm/boot/compressed/misc.c

```
decompress_kernel(ulg output_start, ulg free_mem_ptr_p, ulg free_mem_ptr_
        int arch_id)
{
    output_data         = (uch *)output_start; /* Points to kernel start */
    free_mem_ptr        = free_mem_ptr_p;
    free_mem_ptr_end    = free_mem_ptr_end_p;
    __machine_arch_type   = arch_id;

    arch_decomp_setup();

    makecrc();
    putstr("Uncompressing Linux...");
    gunzip();
    putstr(" done, booting the kernel.\n");
    return output_ptr;
}
```

Gunzip()函数在 lib/inflate.c 中，在 gunzip 函数中调用的重要函数是 inflate()函数，也在 inflate.c 文件中。

这里不对解压算法做详细分析，只要知道解压完的代码是放在 maloc 空间之后就行了。

批注 [s13]: 嗯，其实这个解压缩也比较有技巧的哈

8. 在汇编中调用 c 函数返回的值保存在 r0 中的，即现在 r0 中保存的就是解压后代码的长度，但是需要进行 128 字节对齐，所以采用下面的两句代码

```
add r0, r0, #127
bic r0, r0, #127      @ align the kernel length
```

9. 下面的图片展示的就是将 head.S 本文件中的用来搬运解压后的内核的代码部分搬运到解压后内核的末尾处，以保证在运行该部分代码搬移解压后内核到 0x30008000 处的时候，自己不会被覆盖掉

```
        add r1, r5, r0          @ end of decompressed kernel
        adr r2, reloc_start
        ldr r3, LC1
        add r3, r2, r3
1:      ldmia   r2!, {r8 - r13}     @ copy relocation code
        stmia   r1!, {r8 - r13}
        ldmia   r2!, {r8 - r13}
        stmia   r1!, {r8 - r13}
        cmp r2, r3
        blo 1b

        bl  cache_clean_flush
        add pc, r5, r0          @ call relocation code

/*
```

R1 为解压后内核的结束地址（已经 128 字节对齐了），r2 为本文件中 reloc_start 函数的当前运行空间地址，r3 经过修正之后变成了 reloc_end 的相对当前运行空间的地址。这两个标号都是在本 head.S 文件中

搬运结束之后，调用 cache_clean_flush 刷新 cache，因为代码地址变化了不能让 cache 再命中即将被内核覆盖的老地址。接下来就跳到新地址继续运行，也就是到新地址处去执行 reloc_start 函数。

另外需要注意的是，zImage 在解压后的搬移和跳转会给 gdb 调试内核带来麻烦。因为用来调试的符号表是在编译是生成的，并不知道以后会被搬移到何处去，只有在内核解压缩完成之后，根据计算出来的参数“告诉”调试器这个变化。

9. 下面是 reloc_start 函数的代码：

```
 * r0     = decompressed kernel length
 * r1-r3  = unused
 * r4     = kernel execution address
 * r5     = decompressed kernel start
 * r6     = processor ID
 * r7     = architecture ID
 * r8-r14 = unused
        .align  5
reloc_start:    add r8, r5, r0
        debug_reloc_start
        mov r1, r4
1:
        .rept   4
        ldmia   r5!, {r0, r2, r3, r9 - r13}    @ relocate kernel
        stmia   r1!, {r0, r2, r3, r9 - r13}
        .endr

        cmp r5, r8
        blo 1b
        debug_reloc_end

call_kernel:    bl  cache_clean_flush
        bl  cache_off
        mov r0, #0
        mov r1, r7              @ restore architecture number
        mov pc, r4              @ call kernel
```

首先将解压后的内核代码搬运到 0x30008000 地址处，然后清除并关闭 cache，将 r0 和 r1 设置成对应的之后，最后将 pc 指针指向了 0x30008000 处。
http://forum.linuxbj.com/node/3

三、VMLINUX 内核启动代码分析

从下面开始就进入了真正的内核了，首先执行的代码是在 arch/arm/kernel/head.S 中的汇编代码，然后就进入了内核的 c 代码部分（init/Main.c),注意,此刻内核的编译链接地址不再是 0x0 了,而是 0xC0008000 了。

内核汇编部分：（关于该部分参考我的另一篇详细的分析）

刚刚进入内核汇编时的一些系统状态和寄存器值如下

Mmu 关闭，I Cache 和 D Cache 清除并关闭，r0=0，r1=architecture ID。

从 arch/arm/kernel/vmlinux.lds 文件中可以看出，解压后内核是从 stext 段开始执行的。

1. 确保系统处于 SVC 模式并且 FIR、IRQ 都已经关闭、
2. 调用__lookup_processor_type 函数,通过协处理器 cp15 读取出系统 cpu 的 id，然后通过查找内核映像中 .proc.info.init 段的所有 proc_info_list 数据结构，以判断该内核是否支持该款 cpu。
3. 调用__lookup_machine_type 函数，检查由 uboot 传递进来的 machine architecture number 是否被内核支持，也就是在内核的 .arch.info.init 段中查找看是否有对应 machine number 的 machine_desc 数据结构体存在。

……

关于该部分汇编代码的分析由于涉及到很多内容，所以将该部分的分析转移到一个单独的文件中去分析。arch-arm-kernel-head.S 文件中

arch-arm-kernel-head

参考网址：

http://forum.linuxbj.com/node/4

下面进入 start_kernel 函数去执行，in init/Main.c 文件

1. printk(linux_banner)打印内核的一些信息，版本，作者，编译器版本，日期等信息
2. setup_arch(&command_line); /* in arm/kernel/setup.c */

   函数原型：void __init setup_arch(char **cmdline_p)

   批注 [s14]: 嗯，重中之重啊

   a. setup_processor()→lookup_processor_type()获取对应处理器 id 的 proc_info_list 结构体 list，取出 cpu_name，打印关于 cpuname，id，proc_arch 的信息，设置上 system_utsname= list->arch_name（armv4t），elf_platform= list->elf_name（v4），elf_hwcap = list->elf_hwcap; /* 1|2|4 */，接着执行 cpu_proc_init()函数

      Cpu-single.h 中有如下定义

      ```
      #define cpu_proc_init      __cpu_fn(CPU_NAME,_proc_init)
      #define __cpu_fn(name,x)   __catify_fn(name,x)
      #define __catify_fn(name,x)  name##x
      ```

      CPU_NAME = cpu_arm920

      所以 cpu_proc_init();函数实际上是函数 cpu_arm920_proc_init()函数，在 proc_arm920.S 文件中定义的。

   b. mdesc = setup_machine(machine_arch_type)函数

      取出当前系统的在内核.arch.info.init 段内对应的 machine_desc 数据结构体的地址 list，并打印出 list-name 内容

   c. machine_name = mdesc->name 设置全局变量 machine_name

   d. tags = phys_to_virt(mdesc->boot_params)，修改默认的参数地址，使用 uboot 传递进来的参数的真实地址 0x30000100 并将其转换成虚拟地址 0xc0000100

   e. 参数解析

      ```
      if (tags->hdr.tag == ATAG_CORE) {
          if (meminfo.nr_banks != 0) /* meminfo defined in setup.c */
              squash_mem_tags(tags);
          parse_tags(tags);
      }
      static struct meminfo meminfo __initdata = { 0, };in steup.c
      ```

      所以这里会调用 parse_tags(tags)（in steup.c）函数

Steup_arch()->parse_tags()->parse_tag(),
all function in steup.c

__tagtable_begin, __tagtable_end 这两个参数是在 arch/arm/kernel/vmlinux.lds 文件中确定的，这个区间存放了各种参数的解析函数，可以直接引用
parse_tag()函数就是查找这个区间的各种参数头，来和传递寄来的参数头进行匹配，找到了之后就利用其中的函数指针 parse 来进行参数解析，具体怎么解析需要分析各种参数的作用了。如果匹配不到的话，会打印信息说 Ignoring unrecognised tag 0x%08x\n，表示 uboot 传递进来了一个内核识别不了的参数。

f. struct mm_struct init_mm = INIT_MM(init_mm);

init_mm.start_code = (unsigned long) &_text;
init_mm.end_code = (unsigned long) &_etext;
init_mm.end_data = (unsigned long) &_edata;
init_mm.brk = (unsigned long) &_end;
_text, _etext, _edata, _end 都是在 arch/arm/kernel/vmlinux.lds 文件中确定的

g. parse_cmdline(cmdline_p, from)解析 uboot 传递寄来的 commond_line 字符串，通过 from 指针内的参数解析到 command_line 这个全局数组里，然后将该数组的地址赋值给上上级传进来的参数 command_line（start_kernerl）。这里只分析了当中的 mem、initrd 部分，另外一处是 start_kernel()->parse_option()用于解析其余部分。将 mem 和 initrd 参数从原始 command_line 中扣出，留下的部分放在*cmdline_p 中（start_kernel 传进来的参数，第二次分析会用到）。因为接下来是建立内存的映射页表，需要用到 mem、initrd 两个参数的内容，所以在这里提前解析出来。

h. paging_init(&meminfo, mdesc);/* 这部分的主要工作建立页表，初始化内存 */ in arch/arm/mm/init.c 文件中

memtable_init(mi)为系统内存创建页表，是一个比较重要的函数，
详细的参考网址
http://bbs.sjtu.edu.cn/bbstcon,board,Embedded,reid,1165977462.html
的内容，需要注意的是这个函数中将中断向量表映射到了 0xffff0000 开始处

调用 mdesc->map_io() 函数，在 smdk2410 中位于 arch/arm/mach-s3c2410/mach-smdk2410.c 文件中的 smdk2410_map_io()函数，关于这部分的外设的静态映射分析请参考易松华老师的文档：ARM Linux 静态映射分析，smdk2410_map_io()函数主要做了下面几件事情：
1）、iotable_init(s3c_iodesc, ARRAY_SIZE(s3c_iodesc)) 建立 GPIO,IRQ,MEMCTRL,UART 的静态映射表

批注 [s15]: 别参考了，到你们这个学习阶段，应该当我已经死在沙滩上了，自己分析哈 ^_^

2)、(cpu->map_io)(mach_desc, size) 建立其他外设的静态映射表，包括网卡，LCD,看门狗等

这里需要明确一点的是在 map.h 文件中的

S3C2410_ADDR(x) ((void __iomem *)0xF0000000 + (x))

表明我们处理的 IO 和外设的寄存器被静态映射到了 0xF0000000 开始地方，其中每一种外设的空间占 1M 大小。

这里必须得提到的是内存的三种映射和使用方式：1.处理器的 GPIO，各种外设（IRQ，UART，MEMCRTL，WATCHDOG，USB 等）的寄存器的静态映射，也就是上面红色字体体现出来的。这是映射到内核空间内，所以用户程序是不能访问的，只能由内核来访问。2.内核空间访问的虚拟地址（比如是代码的地址或者是 kmalloc 空间的地址）都是在 0x30008000 开始后的空间，也就是说这中虚拟地址是通过静态映射得到的，这部分也只能由内核访问。phys_to_virt 和 virt_to_ phys ，3.除此之外的物理地址都是通过 mmu 的规则映射出来的，TTB 已经确定在 0x30004000 开始的 16K 空间就是一级页表的空间。通过这中方式映射的虚拟地址用户程序和内核程序都可以访问，不过解析式通过 mmu 来完成的而已。参考 arch/arm/kernel/head.S 中建立的 4M 空间手工页表就是按照 mmu 的规则建立的（只不过是按照 sector 的方式建立）

3)、对时钟，Uart 的简单初始化。

h.request_standard_resources(&meminfo, mdesc) 为 memory ，kernel_text，kernel_data，video_ram 在资源树种申请标准资源。它完成实际的资源分配工作，如果参数 new 所描述的资源中的一部分或全部已经被其它节点所占用，则函数返回与 new 相冲突的 resource 结构的指针。否则就返回 NULL

i. cpu_init()函数分析 in arch/arm/kernel/setup.c

打印一些关于 cpu 的信息，比如 cpu id，cache 大小等。另外重要的是设置了 IRQ、ABT、UND 三种模式的 stack 空间，分别都是 12 个字节。最后将系统切换到 svc 模式。

i. 用__mach_desc_SMDK2410_type 结构体中特定成员来初始化系统的这么写全局变量 init_arch_irq，init_machine，system_timer

3. sched_init()函数
初始化每个处理器的可运行队列，设置系统初始化进程即 0 号进程。也就是调用了 init_idle (current, smp_processor_id())初始化 idle 当前进程。

4. preempt_disable()禁止抢占
5. 建立系统内存页区(zone)链表 build_all_zonelists()
6. printk(KERN_NOTICE "Kernel command line: %s\n", saved_command_line);
打印出从 uboot 传递过来的 command_line 字符串，在 setup_arch 函数中获得的。
7. parse_early_param()，这里分析的是系统能够辨别的一些早期参数（这个函

数甚至可以去掉，__setup 的形式的参数），而且在分析的时候并不是以 setup_arch（&command_line）传出来的 command_line 为基础，而是以最原生态的 saved_command_line 为基础的。

8. parse_args("Booting kernel", command_line, __start___param,
__stop___param - __start___param,
&unknown_bootoption);

对于比较新的版本真正起作用的函数，与 parse_early_param();相比，此处对解析列表的处理范围加大了，解析列表中除了包括系统以 setup 定义的启动参数，还包括模块中定义的 param 参数以及系统不能辨别的参数。

command_line是setup_arch函数传递出来的值；

__start___param 是 param 参数的起始地址，在 System.map 文件中能看到

__stop___param - __start___param是参数个数

unknown_bootoption是对应与启动参数不是param的相应处理函数（查看parse_one()就知道怎么回事）

函数 parse_one()既可以处理 param 参数，又可以处理__setup 的形式的参数，还可以处理不能识别的参数。当然这些都是依靠传递进来的参数进行分支处理的。

参数的处理详情参考网络上另一篇文章：Linux 启动 bootargs 参数分析

9. sort_main_extable()
将放在__start___ex_table 到__stop___ex_table 之间的*(__ex_table)区域中的 struct exception_table_entry 型全局结构变量按 insn 成员变量值从小到大排序，即将可能导致缺页异常的指令按其指令二进制代码值从小到大排序。

10. 在前面的 setup_arch-→paging_init-→ memtable_init 函数中为系统创建页表的时候，中断向量表的虚地址 init_maps，是用 alloc_bootmem_low_pages 分配的，ARM 规定中断向量表的地址只能是 0 或 0xFFFF0000，所以该函数里有部分代码的作用就是映射一页到 0 或 0xFFFF0000。
trap_init 函数做了一下的工作：把放在.Lcvectors 处的系统 8 个意外入口跳转指令搬到高端中断向量 0xffff0000 处，再将__stubs_start 到__stubs_end 之间的各种意外初始化代码搬到 0xffff0200 处。将系统调用的返回句柄拷贝到 0xffff0500 处。刷新 0xffff0000 处 1 页范围的指令 cache，将 DOMAIN_USER 的访问权限由 DOMAIN_MANAGER 改成 DOMAIN_CLIENT 权限。

11. rcu_init()函数初始化当前 cpu 的读、复制、更新数据结构(struct rcu_data)全局变量 per_cpu_rcu_data 和 per_cpu_rcu_bh_data.

12. init_IRQ 函数
初始化系统中支持的最大可能中断数的中断描述结构 struct irqdesc 变量数组 irq_desc[NR_IRQS],把每个结构变量 irq_desc[n]都初始化成预先定义好的坏中断描述结构变量 bad_irq_desc，并初始化该中断的链表表头成员结构变量 pend。

执行 init_arch_irq 函数，该函数是在 setup_arch 函数最后初始化的一个全局函数指针，指向了 smdk2410_init_irq 函数（in mach-smdk2410.c），实际上是调用了 s3c24xx_init_irq 函数，在该函数中，首先清除所有的中断未决标志，之后就初始化中断的触发方式和屏蔽位，还有中断句柄初始化，这里不是最终用户的中断函数，而是 do_level_IRQ 或者 do_edge_IRQ 函数，在这两个函数中都使用过__do_irq 函数来找到真正最终驱动程序注册在系统中的中断处理函数。

```
status = 0;
do {
    ret = action->handler(irq, action->dev_id, regs);
    /* 这里才是用户的真正中断处理程序的执行 */
    if (ret == IRQ_HANDLED)
        status |= action->flags;
    retval |= ret;
    action = action->next;
} while (action);

if (status & SA_SAMPLE_RANDOM)
```

接着初始化外部中断的一些参数，最后补充初始化 uart 和 ADC 中断。

13.pidhash_init()函数

设置系统中每种 pid_hash 表中的 hash 链表数的移位值全局变量 pidhash_shift，将 pidhash_shift 设置成 min（12）。分别为每种 hash 表的连续 hash 链表表头结构申请内存，把申请到的内存虚拟基址分别传给 pid_hash[n](n=1~3)，并将每种 hash 表中的每个 hash 链表表头结构 struct hlist_head 中的 first 成员指针设置成 NULL

14.init_timers()函数

初始化当前处理器的时间向量基本结构 struct tvec_t_base_s 全局变量 per_cpu_tvec_bases，初始化 per_cpu_tvec_bases 的自选锁成员变量 lock

```
void __init init_timers(void)
{
    //这个函数就是timers_nb这个结构体的call函数
    timer_cpu_notify(& timers_nb , (unsigned long)CPU_UP_PREPARE,
                (void *)(long)smp_processor_id());

    //这个是用的机制和cpufreq的机制是一样的，通过notifier_chain_register(&cpu_chain, nb)注册的
    //只不过这里的链是cpu_chain,而cpufreq是其他的链
    register_cpu_notifier(& timers_nb );

    //设置软中断行动函数描述结构变量softirq_vec[=1](系统定时器)的设置
    //也就是设置timer定时器到之后的处理函数
    open_softirq(TIMER_SOFTIRQ, run_timer_softirq, NULL);
}
```

15.softirq_init 函数

内核的软中断机制初始化函数，

```
void __init softirq_init(void)
{
/*HI_SOFTIRQ用于实现bottom half, TASKLET_SOFTIRQ用于公共的tasklet*/
    open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
    open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
}
```

16.time_init()这个函数是用来做体系相关的 timer 的初始化

```
void __init time_init(void)
{
    if ( system_timer->offset == NULL)
        system_timer->offset = dummy_gettimeoffset;
    system_timer->init();
}
```

还记得在 setup_arch 最后初始化了三个全局的变量吗？下图所示

```
/*
 * Set up various architecture-specific pointers
 */
 //运行mach-smdk2410.c文件中定义的结构体__mach_desc_SMDK2410中的特定函数和结构
init_arch_irq  = mdesc->init_irq;/* smdk2410_init_irq */
system_timer   = mdesc->timer;    /* 3c24xx_timer */
init_machine   = mdesc->init_machine;/* smdk2410_init_fs2410 */
```

```
static void __init s3c2410_timer_init (void)
{
    s3c2410_timer_setup();
    setup_irq(IRQ_TIMER4, &s3c2410_timer_irq );
}

struct sys_timer s3c24xx_timer = {
    .init       = s3c2410_timer_init,
    .offset     = s3c2410_gettimeoffset,
    .resume     = s3c2410_timer_setup
};
```

很显然，time_init 函数中的 if 条件不成立，所以就是直接执行了函数 s3c2410_timer_init 函数。可以看出，系统使用了 timer4 来作为系统的定时器。

17.console_init 函数

初始化系统的控制台结构，该函数执行后调用 printk 函数将 log_buf 中所有符合打印级别的系统信息打印到控制台上

a. tty_register_ldisc（）注册默认的 TTY 线路规程

大家研究 tty 的驱动就会发现了在用户和硬件之间 tty 的驱动是分了三层最底层当然是 tty 驱动程序了，主要负责从硬件接受数据，和格式化上层发下来的数据后给硬件。在驱动程序之上就是线路规程了，他负责把从 tty 核心层或者 tty 驱动层接受的数据进行特殊的按着某个协议的

格式化，就像是 ppp 或者蓝牙协议，然后在分发出去的。在 tty 线路规程之上就是 tty 核心层了。

b. 接下来就是执行平台相关的控制台初始化代码

```
s3c24xx_serial_initconsole
/*
在 vmlinux.lds.S 中连接脚本汇编中有这段代码
        __con_initcall_start = .;
        *(.con_initcall.init)
        __con_initcall_end = .;
因此我们再此处调用的就是 con_initcall.init 段的代码，
将 fn 函数放到.con_initcall.init 的输入段中，如下:
        #define console_initcall(fn) \
        static initcall_t __initcall_##fn \
        __attribute_used__
__attribute__((__section__(".con_initcall.init")))=fn
        */
//在我们串口驱动里面有
这么一个注册语句:console_initcall(s3c24xx_serial_initconsole);
//因此我们的控制台初始化流
程就是:start_kernel->console_init->s3c24xx_serial_initconsole

call = __con_initcall_start;    /* console_initcall */
while (call < __con_initcall_end) {
    (*call)();
    call++;
}
```

18.profile_init()函数

```
/* 对系统剖析做相关初始化， 系统剖析用于系统调用*/
//profile 是用来对系统剖析的，在系统调试的时候有用
//需要打开内核选项，并且在 bootargs 中有 profile 这一项才能开启这个功能
/*
        profile 只是内核的一个调试性能的工具，这个可以通过 menuconfig 中 profiling support 打开。
        1. 如何使用 profile:
        首先确认内核支持 profile，然后在内核启动时加入以下参数: profile=1 或者其它参数， 新的内核支持 profile=schedule 1
        2. 内核启动后会创建/proc/profile 文件，这个文件可以通过 readprofile 读取，
        如 readprofile -m /proc/kallsyms | sort -nr > ~/cur_profile.log,
        或者 readprofile -r -m /proc/kallsyms |sort -nr,
        或者 readprofile -r && sleep 1 && readprofile -m /proc/kallsyms
```

```
|sort -nr >~/cur_profile.log
        3. 读取/proc/profile 可获得哪些内容?
        根据启动配置 profile=? 的不同，获取的内容不同:
        如果配置成 profile=? 可以获得每个函数执行次数，用来调试函数性能很有用
        如果设置成 profile=schedule ?可以获得每个函数调用 schedule 的次数，用来调试 schedule 很有用
*/
```

19.local_irq_enable()

使能 IRQ 中断

20.mem_init()

最后内存初始化，释放前边标志为保留的所有页面,这个函数结束之后就不能再使用 alloc_bootmem(),alloc_bootmem_low(),alloc_bootmem_pages()等申请低端内存的函数来申请内存，也就不能申请大块的连续物理内存了

21.kmem_cache_init()

执行高速缓存内存管理即 slab 分配器的初始化

```
22. numa_policy_init();
    if (late_time_init)
        late_time_init();
    calibrate_delay();
    //计算机系统的 BogMIPS 数值，即处理器没秒钟执行的指令数

23. pidmap_init();
    pgtable_cache_init();
    prio_tree_init();
    /*
    初始化无符号长整型全局数组 index_bits_to_maxindex[BITS_PER_LONG]的每个组员
    将每个组员 index_bits_to_maxindex[n] 设置成-1，将最后的 index_bits_to_maxindex[BITS_PER_LONG-1]设置成~0UL
    */

24. anon_vma_init();
    /*
    该函数调用 kmem_cache_creat() 函数，为匿名逆序内存区域链表结构 struct anon_vma
    创建高速缓存内存描述结构 kmem_cache_t 变量，为该变量命名为 anon_vma ,其对象的
    构造函数指针指向 void anon_vma_ctor 函数，析构函数指针为 NULL，将创建的
    kmem_cache_t 结构变量地址传给全局指针 anon_vma_chachep
```

```
    */

25. fork_init(num_physpages);/* 进程创建的相关初始化 */

26. proc_caches_init();
    buffer_init();
    /*
    调用 kmem_cache_create("buffer_head",sizeof(struct buffer_head),
0,SLAB_RECLAIM_ACCOUNT|SLAB_PANIC, init_buffer_head, NULL)函数为缓冲区
描述结构 struct buffer_head 创建高速缓存内存描述结构 kmem_cache_t 变量
    */

27. security_init();/* 打印安全架构方面的信息 */
28. vfs_caches_init(num_physpages);
    radix_tree_init();
    signals_init();
    调用 kmem_cache_create("sigqueue",
                sizeof(struct sigqueue),
                __alignof__(struct sigqueue),
                SLAB_PANIC, NULL, NULL)函数为信号队列结构 struct
sigqueue 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫 sigqueue，不
要求其对象按照处理器硬件 cache line 大小对齐，没有定义其对象的构造和析
构函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针
sihqueue_cachep.

29. page_writeback_init()
    统计系统中所遇内存节点的通用内存页区中高页数水印数之外的额外内存
总页数之和传给 buffer_pages.

30. proc_root_init();/* 只有在系统支持 proc 文件系统即配置了
CONFIG_PROC_FS 选项事才被调用 */

31. check_bugs();/* 在 arm 中是验证内存一致性 */

32. rest_init()，这是另一个重要的函数
    该函数创建 init 内核进程，也就是 1 号进程，然后原来的为 0 号系统启动
进程进入空闲状态
    linux 进程流程:0 号内核进程->1 号内核进程->1 号用户进程
    下面分析 rest_init()函数
    a. kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND)
       in arch/arm/kernel/process.c 文件
       在这个函数中创建了 1 号内核进程，即 init
    b. schedule()进入 idle 之前，主动调用下 schedule，看有没有其他进程
    c. cpu_idle()0 号进程进入空闲态
```

批注 [s16]: 笔误啊，应该是内核线程

init 函数分析 in main.c

a. lock_kernel() 这是 1 号进程的 lock

b. 不需要关注那些和 smp 相关的函数

c. populate_rootfs()函数，必须在 initcalls 之前调用它来安装文件系统，因为有些驱动可能会访问

d. do_basic_setup()驱动加载

```
#define module_init(x)   __initcall(x);
#define __initcall(fn) device_initcall(fn)

#define core_initcall(fn)        __define_initcall("1",fn)
#define postcore_initcall(fn)        __define_initcall("2",fn)
#define arch_initcall(fn)        __define_initcall("3",fn)
#define subsys_initcall(fn)      __define_initcall("4",fn)
#define fs_initcall(fn)          __define_initcall("5",fn)
#define device_initcall(fn)      __define_initcall("6",fn)
#define late_initcall(fn)        __define_initcall("7",fn)

#define __define_initcall(level,fn) \
static initcall_t __initcall_##fn __attribute_used__ \
__attribute__((__section__(".initcall" level ".init"))) = fn
```

上面所涉及的宏都来自 include/linux/init.h 文件

```
    if (board != NULL) {/* board 在本文件中定义和在本文件中初始化 */
        struct platform_device **ptr = board->devices;
        int i;

        for (i = 0; i < board->devices_count; i++, ptr++) {
            ret = platform_device_register(*ptr);

            if (ret) {
                printk(KERN_ERR "s3c24xx: failed to add board devi
            }
        }

        /* mask any error, we may not need all these board
         * devices */
        ret = 0;
    }

    return ret;
} ? end s3c_arch_init ?

arch_initcall(s3c_arch_init);
```

上图中的 arch_initcall 是属于 initcall 第 3 级的，驱动的 module_init 属于第 6 级，总之，不管是什么级别，被这些关键字修饰的说有函数都会在 init 1 号内核进程中的 do_basic_setup 函数中的 do_initcalls() 函数来调用执行，当然执行的先后顺序就是按照 1 到 7 排列的。

顺序是：

Start_kernel*
--→rest_init()*
--→ kernel_thread()* 开启 init 1 号内核进程
--→Init* 内核进程
--→ populate_rootfs() 安装文件系统
--→do_basic_setup()* 系统所有被 initcall 修饰的函数被调用，7 个级别的函数一次执行，其中 modlue_init 修饰的设备驱动程序是在第 6 个级别上。
--→init_workqueues()初始化工作队列
--→usermodehelper_init()启动用户态的 khelper 进程
--→driver_init()设备模型中一些基础结构体初始化和设备的注册
--→sysctl_init()系统调用初始化
--→sock_init()网络 socket 初始化
--→do_initcalls()*
--→(*call)()* 具体的 initcall 函数调用
--→prepare_namespace()挂载文件系统主要工作所在的地方
--→name_to_dev_t()这一步很重要，解析了参数中的 root 选项，获得要挂载的设备号。如果 bootargs 不是以 root=/dev/** 开始，则进入循环，我们也可以通过设备号，例 root=31:03 等价于 /dev/mtdblock3。root 参数的名字不能过长，即/dev/后面的个数不能超过 31。
--→mount_root();/* 挂载设备 */
--→free_initmem()将以 init 标示的函数空间释放，也就是我们启动的时候经常会冒出来:Freeing init memory: 116K
--→sys_open() and sys_dup(0)打开标准终端 0 1 2
--→run_init_process(execute_command)运行第一个用户进程，init=/initrd ，command_line 中传递进来的。
--→ cpu_idle() 系统启动 0 号进程进入空闲态

在 init 1 号进程运行完下面的代码时，整个系统就算起来了。

```
……
run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");
……
```

参考资料：

ARM Linux 静态映射分析--易松华老师，华清远见
内核启动过程-PXA255---易松华老师，华清远见
Linux 启动 bootargs 参数分析
- http://blog.chinaunix.net/u3/99423/article.html

http://bbs.sjtu.edu.cn/bbstcon,board,Embedded,reid,1165977462.html
http://forum.linuxbj.com/node/4

全文完

2010/6/4 lizgo 李枝果 lizhiguo0532@163.com

批注 [s17]: 后面的一段没看的太仔细，主要是困了哈,只能以后再仔细看吓了 *_*