**二：内核引导阶段：**

**S3C2410 Linux 2.6.35.7启动分析(第二阶段)**

接着上面的分析，第一阶段的代码跳转后，会进入第二阶段的代码。

第二阶段的代码是从\arch\arm\kernel\head.S开始的。

**内核启动第二阶段主要完成的工作有:**

**cpu ID检查，**

**machine ID(也就是开发板ID)检查，**

**创建初始化页表，**

**设置C代码运行环境，**

**跳转到内核第一个真正的C函数startkernel开始执行。**

**这一阶段涉及到两个重要的结构体：**

**(1) 一个是struct proc\_info\_list 主要描述CPU相关的信息，定义在文件arch\arm\include\asm\procinfo.h中，与其相关的函数及变量在文件arch/arm/mm/proc\_arm920.S中被定义和赋值。**

**(2) 另一个结构体是描述开发板或者说机器信息的结构体struct machine\_desc，定义在\arch\arm\include\asm\mach\arch.h文件中，其函数的定义和变量的赋值在板极相关文件arch/arm/mach-s3c2410/mach-smdk2410.c中实现，这也是内核移植非常重要的一个文件。**

**该阶段一般由前面的解压缩代码调用,进入该阶段要求：**

**MMU = off, D-cache = off, I-cache = dont care,r0 = 0, r1 = machine id.**

**所有的机器ID列表保存在arch/arm/tools/mach-types 文件中，在编译时会将这些机器ID按照统一的格式链接到基本内核映像文件vmlinux的\_\_arch\_info\_begin和\_\_arch\_info\_end之间的段中。存储格式定义在include/asm-arm/mach/arch.h文件中的结构体struct machine\_desc {}。这两个结构体的内容最终会被连接到基本内核映像vmlinux中的两个段内，分别是\*(.proc.info.init)和\*(.arch.info.init)，可以参考下面的连接脚本。**

**链接脚本：arch/arm/kernel/vmlinux.lds**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*链接脚本\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

SECTIONS

{

. = TEXTADDR;

.init : { /\* 初始化代码段\*/

\_stext = .;

\_sinittext = .;

\*(.init.text)

\_einittext = .;

\_\_proc\_info\_begin = .;

\*(.proc.info.init)

\_\_proc\_info\_end = .;

\_\_arch\_info\_begin = .;

\*(.arch.info.init)

\_\_arch\_info\_end = .;

\_\_tagtable\_begin = .;

\*(.taglist.init)

\_\_tagtable\_end = .;

. = ALIGN(16);

\_\_setup\_start = .;

\*(.init.setup)

\_\_setup\_end = .;

\_\_early\_begin = .;

\*(.early\_param.init)

\_\_early\_end = .;

\_\_initcall\_start = .;

\*(.initcall1.init)

\*(.initcall2.init)

\*(.initcall3.init)

\*(.initcall4.init)

\*(.initcall5.init)

\*(.initcall6.init)

\*(.initcall7.init)

\_\_initcall\_end = .;

\_\_con\_initcall\_start = .;

\*(.con\_initcall.init)

\_\_con\_initcall\_end = .;

\_\_security\_initcall\_start = .;

\*(.security\_initcall.init)

\_\_security\_initcall\_end = .;

. = ALIGN(32);

\_\_initramfs\_start = .;

usr/built-in.o(.init.ramfs)

\_\_initramfs\_end = .;

. = ALIGN(64);

\_\_per\_cpu\_start = .;

\*(.data.percpu)

\_\_per\_cpu\_end = .;

#ifndef CONFIG\_XIP\_KERNEL

\_\_init\_begin = \_stext;

\*(.init.data)

. = ALIGN(4096);

\_\_init\_end = .;

#endif

}

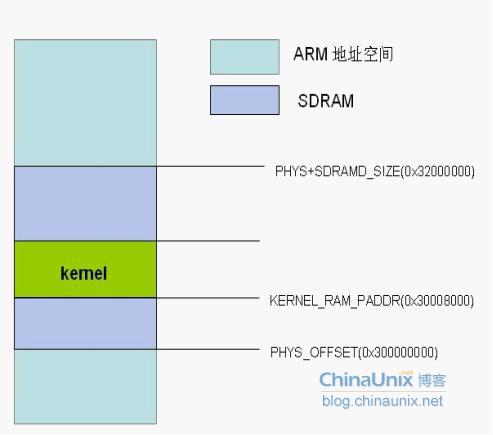
**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*链接脚本\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**下面开始代码\arch\arm\kernel\head.S的注释：**

开始分析前先看下一点基础知识：

1. kernel运行的史前时期和内存布局

在arm平台下，zImage.bin压缩镜像是由bootloader加载到物理内存，然后跳到zImage.bin里一段程序，它专门于将被压缩的kernel解压缩到KERNEL\_RAM\_PADDR开始的一段内存中，接着跳进真正的kernel去执行。该kernel的执行起点是stext函数，定义于arch/arm/kernel/head.S。此时内存的布局如下图所示

**[](http://blog.chinaunix.net/attachment/201210/21/25909619_13508041824F55.jpg)**

在开发板3c2410中，SDRAM连接到内存控制器的Bank6中，它的开始内存地址是0x30000000，大小为64M，即0x20000000。

ARM Linux kernel将SDRAM的开始地址定义为PHYS\_OFFSET。经bootloader加载kernel并由自解压部分代码运行后，最终kernel被放置到KERNEL\_RAM\_PADDR（=PHYS\_OFFSET + TEXT\_OFFSET，即0x30008000）地址上的一段内存，经此放置后，kernel代码以后均不会被移动。

在进入kernel代码前，即bootloader和自解压缩阶段，ARM未开启MMU功能。

因此kernel启动代码一个重要功能是设置好相应的页表，并开启MMU功能。

为了支持MMU功能，kernel镜像中的所有符号，包括代码段和数据段的符号，在链接时都生成了它在开启MMU时，所在物理内存地址映射到的虚拟内存地址。

以arm kernel第一个符号（函数）stext为例，在编译链接，它生成的虚拟地址是0xc0008000，而放置它的物理地址为0x30008000（还记得这是PHYS\_OFFSET+TEXT\_OFFSET吗？）。实际上这个变换可以利用简单的公式进行表示：va = pa – PHYS\_OFFSET + PAGE\_OFFSET。Arm linux最终的kernel空间的页表，就是按照这个关系来建立。

之所以较早提及arm linux 的内存映射，原因是在进入kernel代码，里面所有符号地址值为清一色的0xCXXXXXXX地址，而此时ARM未开启MMU功能，故在执行stext函数第一条执行时，它的PC值就是stext所在的内存地址（即物理地址，0x30008000）。因此，下面有些代码，需要使用地址无关技术。

\_\_HEAD  /\*该宏定义了下面的代码位于".head.text"段内\*/

.type stext, %function                           /\*声明stext为函数\*/

ENTRY(stext)                                      /\*第二阶段的入口地址\*/

setmode PSR\_F\_BIT | PSR\_I\_BIT | SVC\_MODE, r9  @ ensure svc mode and irqs disabled 进入超级权限模式，关中断

/\*从协处理器CP15，C0读取CPU ID,然后在\_\_proc\_info\_begin开始的段中进行查找，如果找到，则返回对应处理器相关结构体在物理地址空间的首地址到r5，最后保存在r10中\*/

mrc p15, 0, r9, c0, c0                  @ get processor id 取出cpu id

bl \_\_lookup\_processor\_type           @ r5=procinfo r9=cpuid

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_lookup\_processor\_type函数的具体解析开始（\arch\arm\kernel\ head-common.S）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

在讲解该程序段之前先来看一些相关知识，内核所支持的每一种CPU 类型都由结构体proc\_info\_list来描述。

该结构体在文件arch/arm/include/asm/procinfo.h 中定义：

struct proc\_info\_list {

unsigned int cpu\_val;

unsigned int cpu\_mask;

unsigned long \_\_cpu\_mm\_mmu\_flags; /\* used by head.S \*/

unsigned long \_\_cpu\_io\_mmu\_flags; /\* used by head.S \*/

unsigned long \_\_cpu\_flush;        /\* used by head.S \*/

const char \*arch\_name;

const char \*elf\_name;

unsigned int elf\_hwcap;

const char \*cpu\_name;

struct processor \*proc;

struct cpu\_tlb\_fns \*tlb;

struct cpu\_user\_fns \*user;

struct cpu\_cache\_fns \*cache;

};

对于 arm920 来说，其对应结构体在文件 linux/arch/arm/mm/proc-arm920.S 中初始化。

.section ".proc.info.init", #alloc, #execinstr /\*定义了一个段，下面的结构体存放在该段中\*/

.type \_\_arm920\_proc\_info,#object              /\*声明一个结构体对象\*/

\_\_arm920\_proc\_info:                            /\*为该结构体赋值\*/

.long 0x41009200

.long 0xff00fff0

.long  PMD\_TYPE\_SECT | \

PMD\_SECT\_BUFFERABLE | \

PMD\_SECT\_CACHEABLE | \

PMD\_BIT4 | \

PMD\_SECT\_AP\_WRITE | \

PMD\_SECT\_AP\_READ

.long  PMD\_TYPE\_SECT | \

PMD\_BIT4 | \

PMD\_SECT\_AP\_WRITE | \

PMD\_SECT\_AP\_READ

b \_\_arm920\_setup

…………………………………

.section ".proc.info.init"表明了该结构在编译后存放的位置。在链接文件 arch/arm/kernel/vmlinux.lds 中：

SECTIONS

{

#ifdef CONFIG\_XIP\_KERNEL

. = XIP\_VIRT\_ADDR(CONFIG\_XIP\_PHYS\_ADDR);

#else

. = PAGE\_OFFSET + TEXT\_OFFSET;

#endif

.text.head : {

\_stext = .;

\_sinittext = .;

\*(.text.head)

}

.init : { /\* Init code and data \*/

INIT\_TEXT

\_einittext = .;

\_\_proc\_info\_begin = .;

\*(.proc.info.init)

\_\_proc\_info\_end = .;

\_\_arch\_info\_begin = .;

\*(.arch.info.init)

\_\_arch\_info\_end = .;

\_\_tagtable\_begin = .;

\*(.taglist.init)

\_\_tagtable\_end = .;

………………………………

｝

所有CPU类型对应的被初始化的 proc\_info\_list结构体都放在 \_\_proc\_info\_begin和\_\_proc\_info\_end之间。

/ \*

\* r9 = cpuid

\*  Returns:

\* r5 = proc\_info pointer in physical address space

\* r9 = cpuid (preserved)

\*/

\_\_lookup\_processor\_type:

adr r3, 3f                     @r3存储的是标号 3 的物理地址（由于没有启用 mmu ，所以当前肯定是物理地址）

ldmia r3, {r5 - r7}              @ R5=\_\_proc\_info\_begin，r6=\_\_proc\_info\_end，r7=标号4处的虚拟地址，即4: .long . 处的地址

add r3, r3, #8                 @ 得到4处的物理地址，刚好是跳过两条指令

sub r3, r3, r7       @ get offset between virt&phys得到虚拟地址和物理地址之间的offset

       /\*利用offset ，将 r5 和 r6 中保存的虚拟地址转变为物理地址\*/

add r5, r5, r3 @ convert virt addresses to

add r6, r6, r3 @ physical address space

1: ldmia r5, {r3, r4} @ value, mask  r3= cpu\_val , r4= cpu\_mask

and r4, r4, r9 @ mask wanted bits;r9 中存放的是先前读出的 processor ID ，此处屏蔽不需要的位

teq r3, r4                      @ 查看代码和CPU 硬件是否匹配（ 比如想在arm920t上运行为cortex-a8编译的内核？不让）

beq 2f                          @ 如果相等则跳转到标号2处，执行返回指令

add r5, r5, #PROC\_INFO\_SZ @ sizeof(proc\_info\_list结构的长度，在这等于48)如果没找到， 跳到下一个proc\_info\_list 处

cmp r5, r6                             @ 判断是不是到了该段的结尾

blo 1b                                 @ 如果没有，继续跳到标号1处，查找下一个

mov r5, #0        @ unknown processor ，如果到了结尾，没找到匹配的，就把0赋值给r5，然后返回

2: mov pc, lr                             @ 找到后返回，r5指向找到的结构体

ENDPROC(\_\_lookup\_processor\_type)

.align 2

3: .long \_\_proc\_info\_begin

.long \_\_proc\_info\_end

4: .long .                                  @“.”表示当前这行代码编译连接后的虚拟地址

.long \_\_arch\_info\_begin

.long \_\_arch\_info\_end

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_lookup\_processor\_type函数的具体解析结束（\arch\arm\kernel\ head-common.S）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

movs r10, r5                     @ invalid processor (r5=0)?

beq \_\_error\_p @ yes, error 'p'

/\*机器 ID是由u-boot引导内核是通过thekernel第二个参数传递进来的，现在保存在r1中,在\_\_arch\_info\_begin开始的段中进行查找，如果找到，则返回machine对应相关结构体在物理地址空间的首地址到r5，最后保存在r8中。

bl \_\_lookup\_machine\_type @ r5=machinfo

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_lookup\_machine\_type函数的具体解析开始（\arch\arm\kernel\ head-common.S）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

每一个CPU 平台都可能有其不一样的结构体，描述这个平台的结构体是 machine\_desc 。

这个结构体在文件arch/arm/include/asm/mach/arch.h 中定义：

struct machine\_desc {

unsigned int nr;          /\* architecture number \*/

unsigned int phys\_io; /\* start of physical io \*/

………………………………

};

对于平台smdk2410 来说其对应 machine\_desc 结构在文件linux/arch/arm/mach-s3c2410/mach-smdk2410.c中初始化：

MACHINE\_START(SMDK2410, "SMDK2410")

.phys\_io = S3C2410\_PA\_UART,

.io\_pg\_offst = (((u32)S3C24XX\_VA\_UART) >> 18) & 0xfffc,

.boot\_params = S3C2410\_SDRAM\_PA + 0x100,

.map\_io = smdk2410\_map\_io,

.init\_irq = s3c24xx\_init\_irq,

.init\_machine = smdk2410\_init,

.timer = &s3c24xx\_timer,

MACHINE\_END

对于宏MACHINE\_START 在文件 arch/arm/include/asm/mach/arch.h 中定义：

#define MACHINE\_START(\_type,\_name) /

static const struct machine\_desc \_\_mach\_desc\_##\_type /

 \_\_used /

 \_\_attribute\_\_((\_\_section\_\_(".arch.info.init"))) = { /

.nr = MACH\_TYPE\_##\_type, /

.name = \_name,

#define MACHINE\_END /

};

\_\_attribute\_\_((\_\_section\_\_(".arch.info.init")))表明该结构体在并以后存放的位置。

在链接文件 链接脚本文件 arch/arm/kernel/vmlinux.lds 中

SECTIONS

{

#ifdef CONFIG\_XIP\_KERNEL

. = XIP\_VIRT\_ADDR(CONFIG\_XIP\_PHYS\_ADDR);

#else

. = PAGE\_OFFSET + TEXT\_OFFSET;

#endif

.text.head : {

\_stext = .;

\_sinittext = .;

\*(.text.head)

}

.init : { /\* Init code and data \*/

INIT\_TEXT

\_einittext = .;

\_\_proc\_info\_begin = .;

\*(.proc.info.init)

\_\_proc\_info\_end = .;

\_\_arch\_info\_begin = .;

\*(.arch.info.init)

\_\_arch\_info\_end = .;

………………………………

｝

在\_\_arch\_info\_begin和 \_\_arch\_info\_end之间存放了linux内核所支持的所有平台对应的 machine\_desc 结构体。

/\*

\*  r1 = machine architecture number

 \* Returns:

\*  r5 = mach\_info pointer in physical address space

 \*/

\_\_lookup\_machine\_type:

adr r3, 4b                      @ 把标号4处的地址放到r3寄存器里面

ldmia r3, {r4, r5, r6}            @ R 4 = 标号4处的虚拟地址 ，r 5 = \_\_arch\_info\_begin ，r 6= \_\_arch\_info\_end

sub r3, r3, r4 @ get offset between virt&phys 计算出虚拟地址与物理地址的偏移

/\*利用offset ，将 r5 和 r6 中保存的虚拟地址转变为物理地址\*/

add r5, r5, r3 @ convert virt addresses to

add r6, r6, r3 @ physical address space

/\*读取machine\_desc结构的 nr 参数，对于smdk2410 来说该值是 MACH\_TYPE\_SMDK2410,这个值在文件linux/arch/arm/tools/mach-types 中:

smdk2410    ARCH\_SMDK2410 SMDK2410  193 \*/

1: ldr r3, [r5, #MACHINFO\_TYPE] @ get machine type

teq r3, r1 @ matches loader number?把取到的machine id和从uboot中传过来的machine id（存放r1中）相比较

beq 2f @ found 如果相等，则跳到标号2处，返回

add r5, r5, #SIZEOF\_MACHINE\_DESC@ next machine\_desc 没有找到，则继续找下一个，加上该结构体的长度

cmp r5, r6                      @ 判断是否已经到该段的末尾

blo 1b                          @ 如果没有，则跳转到标号1处，继续查找

mov r5, #0 @ unknown machine 如果已经到末尾，并且没找到，则返回值r5寄存器赋值为0

2: mov pc, lr                      @ 返回原函数，且r5作为返回值

ENDPROC(\_\_lookup\_machine\_type)

.align 2

3: .long \_\_proc\_info\_begin

.long \_\_proc\_info\_end

4: .long .                                  @“.”表示当前这行代码编译连接后的虚拟地址

.long \_\_arch\_info\_begin

.long \_\_arch\_info\_end

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_lookup\_machine\_type函数的具体解析结束（\arch\arm\kernel\ head-common.S）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

movs r8, r5 @ invalid machine (r5=0)?

beq \_\_error\_a @ yes, error 'a'

/\*检查 bootloader传入的参数列表 atags 的 合法性\*/

bl \_\_vet\_atags

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_vet\_atags函数的具体解析开始（\arch\arm\kernel\ head-common.S）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

关于参数链表：

内核参数链表的格式和说明可以从内核源代码目录树中的\arch\arm\include\asm\setup.h中找到，参数链表必须以ATAG\_CORE开始，以ATAG\_NONE结束。这里的 ATAG\_CORE，ATAG\_NONE是各个参数的标记，本身是一个32 位值，例如： ATAG\_CORE=0x54410001 。 其它的参数标记还包括： ATAG\_MEM32  ，  ATAG\_INITRD  ，  ATAG\_RAMDISK  ， ATAG\_COMDLINE 等。每个参数标记就代表一个参数结构体，由各个参数结构体构成了参数链表。参数结构体的定义如下：

struct tag {   
      struct  tag\_header  hdr;   
      union {   
             struct tag\_core  core;   
             struct tag\_mem32  mem;   
          struct tag\_videotext videotext;   
          struct tag\_ramdisk   ramdisk;   
          struct tag\_initrd    initrd;   
          struct tag\_serialnr  serialnr;   
          struct tag\_revision  revision;   
          struct tag\_videolfb  videolfb;   
          struct tag\_cmdline   cmdline;   
          struct tag\_acorn     acorn;   
          struct tag\_memclk    memclk;   
        } u;   
};

参数结构体包括两个部分，一个是 tag\_header 结构体 , 一个是 u 联合体。

tag\_header结构体的定义如下：

struct tag\_header {

                 u32 size;

                 u32 tag;

};

其中 size ：表示整个  tag  结构体的大小 ( 用字的个数来表示，而不是字节的个数 ) ，等于tag\_header的大小加上  u 联合体的大小，例如，参数结构体  ATAG\_CORE  的size=(sizeof(tag->tag\_header)+sizeof(tag->u.core))>>2，一般通过函数  tag\_size(struct \* tag\_xxx) 来获得每个参数结构体的 size 。其中  tag ：表示整个  tag  结构体的标记，如： ATAG\_CORE 等。

/\* r8  = machinfo

\* Returns:

 \*  r2 either valid atags pointer, or zero

\*/

\_\_vet\_atags:

tst r2, #0x3 @ aligned? r2指向该参数链表的起始位置，此处判断它是否字对齐

bne 1f                          @ 如果没有对齐，跳到标号1处直接返回，并且把r2的值赋值为0，作为返回值

ldr r5, [r2, #0] @ is first tag ATAG\_CORE? 获取第一个 tag 结构的 size

cmp r5, #ATAG\_CORE\_SIZE         @ 判断该 tag 的长度是否合法

cmpne r5, #ATAG\_CORE\_SIZE\_EMPTY

bne 1f                          @ 如果不合法，异常返回

ldr r5, [r2, #4]                @ 获取第一个 tag 结构体的标记

ldr r6, =ATAG\_CORE              @ 取出标记ATAG\_CORE的内容

cmp r5, r6                      @ 判断该标记是否等于ATAG\_CORE

bne 1f                          @ 如果不等，异常返回

mov pc, lr @ atag pointer is ok，如果都相等，则正常返回

1: mov r2, #0                      @ 异常返回值

mov pc, lr @ 异常返回

ENDPROC(\_\_vet\_atags)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_vet\_atags函数的具体解析结束（\arch\arm\kernel\ head-common.S）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*创建内核初始化页表\*/

bl \_\_create\_page\_tables

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_create\_page\_tables函数的具体解析开始（\arch\arm\kernel\ head.S）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* r8  = machinfo

 \* r9  = cpuid

 \* r10 = procinfo

\* Returns:

\*  r4 = physical page table address

 \*/

/\*在该文件的开头有如下宏定义\*/

#define KERNEL\_RAM\_PADDR (PHYS\_OFFSET + TEXT\_OFFSET)

.macro pgtbl, rd

ldr\rd, =(KERNEL\_RAM\_PADDR - 0x4000)

.endm

其中：PHYS\_OFFSET在arch/arm/mach-s3c2410/include/mach/memory.h定义，为UL(0x30000000)，而TEXT\_OFFSET在arch/arm/Makefile中定义，为内核镜像在内存中到内存开始位置的偏移（字节），为$(textofs-y) textofs-y也在文件arch/arm/Makefile中定义，为textofs-y   := 0x00008000，r4 = 30004000为临时页表的起始地址，首先即是初始化16K的页表，高12位虚拟地址为页表索引，每个页表索引占4个字节，所以为4K\*4 = 16K，大页表，每一个页表项，映射1MB虚拟地址.

\_\_create\_page\_tables:

/\*为内核代码存储区域创建页表，首先将内核起始地址-0x4000到内核起始地址之间的16K存储器清0 ，将创建的页表存于此处\*/

pgtbl r4 @ r4中存放的为页表的基地址，最终该地址会写入cp15的寄存器c2，这个值必须是 16K 对齐的

mov r0, r4                      @ 把页表的基地址存放到r0中

mov r3, #0                      @ 把r3清0

add r6, r0, #0x4000             @ r6指向16K的末尾

1: str r3, [r0], #4                @ 把16K的页表空间清0

str r3, [r0], #4

str r3, [r0], #4

str r3, [r0], #4

teq r0, r6

bne 1b

/\*从proc\_info\_list结构中获取字段 \_\_cpu\_mm\_mmu\_flags ，该字段包含了存储空间访问权限等, 此处指令执行之后r7=0x00000c1e\*/

ldr r7, [r10, #PROCINFO\_MM\_MMUFLAGS] @ mm\_mmuflags

/\*为内核的第一MB创建一致的映射，以为打开MMU做准备，这个映射将会被paging\_init()移除，这里使用程序计数器来获得相应的段的基地址\*/

mov r6, pc

mov r6, r6, lsr #20 @ start of kernel section

orr r3, r7, r6, lsl #20 @ flags + kernel base

str r3, [r4, r6, lsl #2] @ identity mapping

/\* MMU是通过 C2 中基地址（高 18 位）与虚拟地址的高 12 位组合成物理地址，在转换表中查找地址条目。 R4 中存放的就是这个基地址 0x30004000\*/

add r0, r4,  #(KERNEL\_START & 0xff000000) >> 18   @ r0 = 0x30007000 r0存放的是转换表的起始位置

str r3, [r0, #(KERNEL\_START & 0x00f00000) >> 18]! @ r3存放的是内核镜像代码段的起始地址

ldr r6, =(KERNEL\_END - 1)                         @ 获取内核的尾部虚拟地址存于r6中

add r0, r0, #4                                    @ 第一个地址条目存放在 0x30007004 处，以后依次递增

add r6, r4, r6, lsr #18                           @ 计算最后一个地址条目存放的位置

1: cmp r0, r6                                        @ 填充这之间的地址条目

/\*每一个地址条目代表了 1MB 空间的地址映射。物理地址将从0x30100000开始映射。0X30000000 开始的 1MB 空间将在下面映射\*/

add r3, r3, #1 << 20

strls r3, [r0], #4

bls 1b

…………………………………

…………………………………………

/\*为了使用启动参数，将物理内存的第一MB映射到内核虚拟地址空间的第一个MB，r4存放的是页表的地址。映射0X30000000开始的 1MB 空间PAGE\_OFFSET = 0XC0000000,PHYS\_OFFSET = 0X30000000, r0 =  0x30007000, 上面是从 0x30007004开始存放地址条目的\*/

add r0, r4, #PAGE\_OFFSET >> 18

orr r6, r7, #(PHYS\_OFFSET & 0xff000000)  @ r6= 0x30000c1e

.if (PHYS\_OFFSET & 0x00f00000)

orr r6, r6, #(PHYS\_OFFSET & 0x00f00000)

.endif

str r6, [r0]                            @ 将0x30000c1e 存于0x30007000处。

………………………

………………………………

mov pc, lr                              @子程序返回

ENDPROC(\_\_create\_page\_tables)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_create\_page\_tables函数的具体解析结束（\arch\arm\kernel\ head.S）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*把\_\_switch\_data标号处的地址放入r13寄存器，当执行完\_\_enable\_mmu函数时会把r13寄存器的值赋值给pc，跳转到\_\_switch\_data 处执行\*/

ldr r13, \_\_switch\_data @ address to jump to after mmu has been enabled

/\*把\_\_enable\_mmu函数的地址值，赋值给lr寄存器，当执行完\_\_arm920\_setup时，返回后执行\_\_enable\_mmu \*/

adr lr, BSYM(\_\_enable\_mmu) @ return (PIC) address

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_enable\_mmu函数的具体解析开始（\arch\arm\kernel\ head.S）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_enable\_mmu:

#ifdef CONFIG\_ALIGNMENT\_TRAP

orr r0, r0, #CR\_A   //使能地址对齐错误检测

#else

bic r0, r0, #CR\_A

#endif

#ifdef CONFIG\_CPU\_DCACHE\_DISABLE

bic r0, r0, #CR\_C   //禁止数据 cache

#endif

#ifdef CONFIG\_CPU\_BPREDICT\_DISABLE

bic r0, r0, #CR\_Z

#endif

#ifdef CONFIG\_CPU\_ICACHE\_DISABLE

bic r0, r0, #CR\_I  //禁止指令 cache

#endif             //配置相应的访问权限并存入 r5 中

mov r5, #(domain\_val(DOMAIN\_USER, DOMAIN\_MANAGER) | /

      domain\_val(DOMAIN\_KERNEL, DOMAIN\_MANAGER) | /

      domain\_val(DOMAIN\_TABLE, DOMAIN\_MANAGER) | /

      domain\_val(DOMAIN\_IO, DOMAIN\_CLIENT))

mcr p15, 0, r5, c3, c0, 0 //将访问权限写入协处理器

mcr p15, 0, r4, c2, c0, 0 //将页表基地址写入基址寄存器 C2 ， 0X30004000

b \_\_turn\_mmu\_on          //跳转到程序段去打开 MMU

ENDPROC(\_\_enable\_mmu)

文件linux/arch/arm/kernel/head.S 中

\_\_turn\_mmu\_on:

mov r0, r0

mcr p15, 0, r0, c1, c0, 0 //打开 MMU 同时打开 cache 等。

mrc p15, 0, r3, c0, c0, 0 @ read id reg 读取 id 寄存器

mov r3, r3

mov r3, r3    //两个空操作，等待前面所取的指令得以执行。

mov pc, r13  //程序跳转

ENDPROC(\_\_turn\_mmu\_on)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_enable\_mmu函数的具体解析结束（\arch\arm\kernel\ head.S）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*执行\_\_arm920\_setup函数(\arch\arm\mm\ proc-arm920.S),该函数完成对数据cache，指令cache，write buffer等初始化操作\*/

  ARM( add pc, r10, #PROCINFO\_INITFUNC )

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_arm920\_setup函数的具体解析开始（\arch\arm\mm\ proc-arm920.S）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

在上面程序段.section ".text.head", "ax" 的最后有这样几行：

add pc, r10, #PROCINFO\_INITFUNC

R10中存放的是在函数 \_\_lookup\_processor\_type 中成功匹配的结构体 proc\_info\_list。对于arm920 来说在文件 linux/arch/arm/mm/proc-arm920.S 中有：

.section ".proc.info.init", #alloc, #execinstr

.type  \_\_arm920\_proc\_info,#object

\_\_arm920\_proc\_info:

.long 0x41009200

.long 0xff00fff0

.long   PMD\_TYPE\_SECT | /

PMD\_SECT\_BUFFERABLE | /

PMD\_SECT\_CACHEABLE | /

PMD\_BIT4 | /

PMD\_SECT\_AP\_WRITE | /

PMD\_SECT\_AP\_READ

.long   PMD\_TYPE\_SECT | /

PMD\_BIT4 | /

PMD\_SECT\_AP\_WRITE | /

PMD\_SECT\_AP\_READ

b \_\_arm920\_setup

………………………………

add pc, r10, #PROCINFO\_INITFUNC的意思跳到函数 \_\_arm920\_setup去执行。

.type \_\_arm920\_setup, #function  //表明这是一个函数

\_\_arm920\_setup:

mov r0, #0                      //设置 r0 为 0 。

mcr p15, 0, r0, c7, c7          //使数据 cahche,  指令 cache 无效。

mcr p15, 0, r0, c7, c10, 4      //使 write buffer 无效。

#ifdef CONFIG\_MMU

mcr p15, 0, r0, c8, c7          //使数据 TLB, 指令 TLB 无效。

#endif

adr r5, arm920\_crval            //获取 arm920\_crval 的地址，并存入 r5 。

ldmia r5, {r5, r6}              //获取 arm920\_crval 地址处的连续 8 字节分别存入 r5,r6 。

mrc p15, 0, r0, c1, c0          //获取 CP15 下控制寄存器的值，并存入 r0 。

bic r0, r0, r5                  //通过查看 arm920\_crval 的值可知该行是清除 r0 中相关位，为以后对这些位的赋值做准备

orr r0, r0, r6                  //设置 r0 中的相关位，即为 mmu 做相应设置。

mov pc, lr                      //上面有操作 adr lr, \_\_enable\_mmu ，此处将跳到程序段 \_\_enable\_mmu 处。

.size \_\_arm920\_setup, . - \_\_arm920\_setup

.type arm920\_crval, #object

arm920\_crval:

crval clear=0x00003f3f, mmuset=0x00003135, ucset=0x00001130

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

\_\_arm920\_setup函数的具体解析结束（\arch\arm\mm\ proc-arm920.S）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ENDPROC(stext)

接着往下分析linux/arch/arm/kernel/head-common.S中：

.type \_\_switch\_data, %object      @定义\_\_switch\_data为一个对象

\_\_switch\_data:

.long \_\_mmap\_switched

.long \_\_data\_loc @ r4

.long \_data @ r5

.long \_\_bss\_start @ r6

.long \_end @ r7

.long processor\_id @ r4

.long \_\_machine\_arch\_type @ r5

.long \_\_atags\_pointer @ r6

.long cr\_alignment @ r7

.long init\_thread\_union + THREAD\_START\_SP @ sp

/\*

 \* The following fragment of code is executed with the MMU on in MMU mode,

 \* and uses absolute addresses; this is not position independent.

\*  r0  = cp#15 control register

 \*  r1  = machine ID

 \*  r2  = atags pointer

 \*  r9  = processor ID

 \*/

 /\*其中上面的几个段的定义是在文件arch/arm/kernel/vmlinux.lds 中指定\*/

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* vmlinux.lds开始\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 SECTIONS

 {

 ……………………

 #ifdef CONFIG\_XIP\_KERNEL

 \_\_data\_loc = ALIGN(4); /\* location in binary \*/

 . = PAGE\_OFFSET + TEXT\_OFFSET;

 #else

 . = ALIGN(THREAD\_SIZE);

  \_\_data\_loc = .;

 #endif

 .data : AT(\_\_data\_loc) {  //此处数据存储在上面\_\_data\_loc处。

  \_data = .; /\* address in memory \*/

  \*(.data.init\_task)

…………………………

.bss : {

\_\_bss\_start = .; /\* BSS \*/

\*(.bss)

\*(COMMON)

\_end = .;

}

………………………………

｝

init\_thread\_union 是 init进程的基地址.在 arch/arm/kernel/init\_task.c 中:

union thread\_union init\_thread\_union \_\_attribute\_\_((\_\_section\_\_(".init.task"))) = { INIT\_THREAD\_INFO(init\_task) };

对照 vmlnux.lds.S 中,我们可以知道init task是存放在 .data 段的开始8k, 并且是THREAD\_SIZE(8k)对齐的 \*/

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* vmlinux.lds结束\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\_\_mmap\_switched:

adr r3, \_\_switch\_data + 4

ldmia r3!, {r4, r5, r6, r7}

……………………

………………………………

mov fp, #0 @ 清除bss段

1: cmp r6, r7

strcc fp, [r6],#4

bcc 1b

 ARM( ldmia r3, {r4, r5, r6, r7, sp})  /\*把\_\_machine\_arch\_type变量值放入r5中，把\_\_atags\_pointer变量的值放入r6中\*/

str r9, [r4] @ Save processor ID 保存处理器id到processor\_id所在的地址中

str r1, [r5] @ Save machine type 保存machine  id到\_\_machine\_arch\_type中

str r2, [r6] @ Save atags pointer 保存参数列表首地址到\_\_atags\_pointer中

bic r4, r0, #CR\_A @ Clear 'A' bit

stmia r7, {r0, r4} @ Save control register values

b start\_kernel                @程序跳转到函数 start\_kernel 进入 C 语言部分。

ENDPROC(\_\_mmap\_switched)

到处我们的启动的第二阶段分析完毕。

**后面会接着分析第三阶段。第三阶段完全是C语言代码，从start\_kernel函数开始。**