Linux内核启动流程分析(一) 2012-10-21 15:01:50

S3C2410 Linux 2.6.35.7启动分析(第一阶段)

arm linux 内核生成过程

1. 依据arch/arm/kernel/vmlinux.lds 生成linux内核源码根目录下的vmlinux，这个vmlinux属于未压缩，带调试信息、符号表的最初的内核，大小约23MB；

命令：arm-linux-gnu-ld -o vmlinux -T arch/arm/kernel/vmlinux.lds

arch/arm/kernel/head.o

init/built-in.o

--start-group

arch/arm/mach-s3c2410/built-in.o

kernel/built-in.o

mm/built-in.o

fs/built-in.o

ipc/built-in.o

drivers/built-in.o

net/built-in.o

--end-group .tmp\_kallsyms2.o

2. 将上面的vmlinux去除调试信息、注释、符号表等内容，生成arch/arm/boot/Image，这是不带多余信息的linux内核，Image的大小约3.2MB；

命令:arm-linux-gnu-objcopy -O binary -S vmlinux arch/arm/boot/Image

3.将 arch/arm/boot/Image 用gzip -9 压缩生成arch/arm/boot/compressed/piggy.gz大小约1.5MB； 命令:gzip -f -9 < arch/arm/boot/compressed/../Image > arch/arm/boot/compressed/piggy.gz

4. 编译arch/arm/boot/compressed/piggy.S 生成arch/arm/boot/compressed/piggy.o大小约1.5MB，这里实际上是将piggy.gz通过piggy.S编译进piggy.o文件中。而piggy.S文件仅有6行，只是包含了文件piggy.gz;

命令:arm-linux-gnu-gcc -o arch/arm/boot/compressed/piggy.o arch/arm/boot/compressed/piggy.S

5. 依据arch/arm/boot/compressed/vmlinux.lds 将arch/arm/boot/compressed/目录下的文件head.o 、piggy.o 、misc.o链接生成 arch/arm/boot/compressed/vmlinux，这个vmlinux是经过压缩且含有自解压代码的内核,大小约1.5MB;

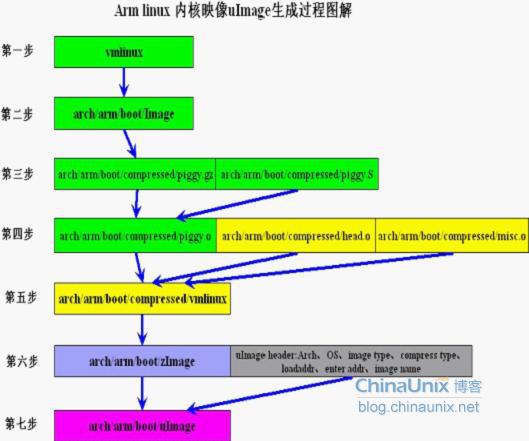
命令:arm-linux-gnu-ld zreladdr=0x30008000 params\_phys=0x30000100 -T arch/arm/boot/compressed/vmlinux.lds arch/arm/boot/compressed/head.o arch/arm/boot/compressed/piggy.o arch/arm/boot/compressed/misc.o -o arch/arm/boot/compressed/vmlinux

6. 将arch/arm/boot/compressed/vmlinux去除调试信息、注释、符号表等内容，生成arch/arm/boot/zImage大小约1.5MB;这已经是一个可以使用的linux内核映像文件了；

命令:arm-linux-gnu-objcopy -O binary -S arch/arm/boot/compressed/vmlinux arch/arm/boot/zImage

7. 将arch/arm/boot/zImage添加64Bytes的相关信息打包为arch/arm/boot/uImage大小约1.5MB;

命令: ./mkimage -A arm -O linux -T kernel -C none -a 0x30008000 -e 0x30008000 -n 'Linux-2.6.35.7' -d arch/arm/boot/zImage arch/arm/boot/uImage



内核启动分析：

本文着重分析S3C2410 linux-2.6.35.7 内核启动的详细过程，主要包括： zImage 解压缩阶段、 vmlinux 启动汇编阶段、 startkernel 到创建第一个进程阶段三个部分，一般将其称为 linux 内核启动一、二、三阶段，本文也将采用这种表达方式。对于 zImage 之前的启动过程，本文不做表述，可参考前面正亮讲得 “ u-boot的启动过程分析”。

本文中涉及到的术语约定如下：

基本内核映像：即内核编译过程中最终在内核源代码根目录下生成的 vmlinux 映像文件，并不包含任何内核解压缩和重定位代码；

zImage 内核映像：包含了内核piggy.o及解压缩和重定位代码，通常是目标板 bootloader 加载的对象；

zImage 下载地址：即 bootloader 将 zImage 下载到目标板内存的某个地址或者 nand read 将 zImage 读到内存的某个地址；

zImage 加载地址：由 Linux 的 bootloader 完成的将 zImage 搬移到目标板内存的某个位置所对应的地址值，默认值 0x30008000 。

1、 Linux 内核启动第一阶段：内核解压缩和重定位

该阶段是从 u-boot 引导进入内核执行的第一阶段，我们知道 u-boot 引导内核启动的最后一步是：通过一个函数指针 thekernel()带三个参数跳转到内核（ zImage ）入口点开始执行，此时， u-boot 的任务已经完成，控制权完全交给内核（ zImage ）。

稍作解释，在 u-boot 的文件arch\arm\lib\bootm.c(uboot-2010.9)中定义了 thekernel, 并在 do\_bootm\_linux 的最后执行 thekernel.

定义如下：void (\*theKernel)(int zero, int arch, uint params);

theKernel = (void (\*)(int, int, uint))ntohl(hdr->ih\_ep);

//hdr->ih\_ep----Entry Point Address uImage 中指定的内核入口点，这里是 0x30008000 。

theKernel (0, bd->bi\_arch\_number, bd->bi\_boot\_params);

其中第二个参数为机器 ID, 第三参数为 u-boot 传递给内核参数存放在内存中的首地址，此处是 0x30000100 。

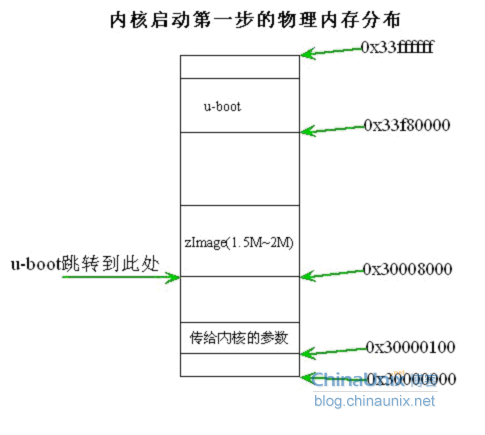
由上述 zImage 的生成过程我们可以知道，第一阶段运行的内核映像实际就是arch/arm/boot/compressed/vmlinux，而这一阶段所涉及的文件也只有三个：

(1)arch/arm/boot/compressed/vmlinux.lds

(2)arch/arm/boot/compressed/head.S

(3)arch/arm/boot/compressed/misc.c

下面的图是使用64MRAM时，通常的内存分布图：



下面我们的分析集中在 arch/arm/boot/compressed/head.S, 适当参考 vmlinux.lds 。

从linux/arch/arm/boot/compressed/vmlinux.lds文件可以看出head.S的入口地址为ENTRY(\_start)，也就是head.S汇编文件的\_start标号开始的第一条指令。

下面从head.S中得\_start 标号开始分析。(有些指令不影响初始化，暂时略去不分析)

代码位置在/arch/arm/boot/compressed/head.S中：

start:

.type start,#function /\*uboot跳转到内核后执行的第一条代码\*/

.rept 8 /\*重复定义8次下面的指令，也就是空出中断向量表的位置\*/

mov r0, r0 /\*就是nop指令\*/

.endr

b 1f @ 跳转到后面的标号1处

.word 0x016f2818 @ 辅助引导程序的幻数，用来判断镜像是否是zImage

.word start @ 加载运行zImage的绝对地址，start表示赋的初值

.word \_edata @ zImage结尾地址，\_edata是在vmlinux.lds.S中定义的，表示init,text,data三个段的结束位置

1: mov r7, r1 @ save architecture ID 保存体系结构ID 用r1保存

mov r8, r2 @ save atags pointer 保存r2寄存器 参数列表，r0始终为0

mrs r2, cpsr @ get current mode 得到当前模式

tst r2, #3 @ not user?，tst实际上是相与,判断是否处于用户模式

bne not\_angel @ 如果不是处于用户模式，就跳转到not\_angel标号处

/\*如果是普通用户模式，则通过软中断进入超级用户权限模式\*/

mov r0, #0x17 @ angel\_SWIreason\_EnterSVC，向SWI中传递参数

swi 0x123456 @ angel\_SWI\_ARM这个是让用户空间进入SVC空间

not\_angel: /\*表示非用户模式，可以直接关闭中断\*/

mrs r2, cpsr @ turn off interrupts to 读出cpsr寄存器的值放到r2中

orr r2, r2, #0xc0 @ prevent angel from running关闭中断

msr cpsr\_c, r2 @ 把r2的值从新写回到cpsr中

/\*读入地址表。因为我们的代码可以在任何地址执行，也就是位置无关代码（PIC），所以我们需要加上一个偏移量。下面有每一个列表项的具体意义。

LC0是表的首项，它本身就是在此head.s中定义的

.type LC0, #object

LC0: .word LC0 @ r1 LC0表的起始位置

.word \_\_bss\_start @ r2 bss段的起始地址在vmlinux.lds.S中定义

.word \_end @ r3 zImage（bss）连接的结束地址在vmlinux.lds.S中定义

.word zreladdr @ r4 zImage的连接地址，我们在arch/arm/mach-s3c2410/makefile.boot中定义的

.word \_start @ r5 zImage的基地址，bootp/init.S中的\_start函数，主要起传递参数作用

.word \_got\_start @ r6 GOT（全局偏移表）起始地址，\_got\_start是在compressed/vmlinux.lds.in中定义的

.word \_got\_end @ ip GOT结束地址

.word user\_stack+4096 @ sp 用户栈底 user\_stack是紧跟在bss段的后面的，在compressed/vmlinux.lds.in中定义的

@ 在本head.S的末尾定义了zImag的临时栈空间，在这里分配了4K的空间用来做堆栈。

.section ".stack", "w"

user\_stack: .space 4096

GOT表的初值是连接器指定的，当时程序并不知道代码在哪个地址执行。如果当前运行的地址已经和表上的地址不一样，还要修正GOT表。\*/

.text

adr r0, LC0 /\*把地址表的起始地址放入r0中\*/

ldmia r0, {r1, r2, r3, r4, r5, r6, ip, sp} /\*加载地址表中的所有地址到相应的寄存器\*/

@r0是运行时地址，而r1则是链接时地址，而它们两都是表示LC0表的起始位置，这样他们两的差则是运行和链接的偏移量，纠正了这个偏移量才可以运行与”地址相关的代码“

subs r0, r0, r1 @ calculate the delta offset 计算偏移量，并放入r0中

beq not\_relocated @ if delta is zero, we are running at the address we were linked at.

@ 如果为0，则不用重定位了，直接跳转到标号not\_relocated处执行

/\*

\* 偏移量不为零，说明运行在不同的地址，那么需要修正几个指针

\* r5 – zImage基地址

\* r6 – GOT（全局偏移表）起始地址

\* ip – GOT结束地址

\*/

add r5, r5, r0 /\*加上偏移量修正zImage基地址\*/

add r6, r6, r0 /\*加上偏移量修正GOT（全局偏移表）起始地址\*/

add ip, ip, r0 /\*加上偏移量修正GOT（全局偏移表）结束地址\*/

/\*

\* 这时需要修正BSS区域的指针，我们平台适用。

\* r2 – BSS 起始地址

\* r3 – BSS 结束地址

\* sp – 堆栈指针

\*/

add r2, r2, r0 /\*加上偏移量修正BSS 起始地址\*/

add r3, r3, r0 /\*加上偏移量修正BSS 结束地址\*/

add sp, sp, r0 /\*加上偏移量修正堆栈指针\*/

/\*

\* 重新定位GOT表中所有的项.

\*/

1: ldr r1, [r6, #0] @ relocate entries in the GOT

add r1, r1, r0 @ table. This fixes up the

str r1, [r6], #4 @ C references.

cmp r6, ip

blo 1b

not\_relocated: mov r0, #0

1: str r0, [r2], #4 @ clear bss 清除bss段

str r0, [r2], #4

str r0, [r2], #4

str r0, [r2], #4

cmp r2, r3

blo 1b

bl cache\_on /\* 开启指令和数据Cache ，为了加快解压速度\*/

@ 这里的 r1,r2 之间的空间为解压缩内核程序所使用，也是传递给 decompress\_kernel 的第二和第三的参数

mov r1, sp @ malloc space above stack

add r2, sp, #0x10000 @ 64k max解压缩的缓冲区

@下面程序的意义就是保证解压地址和当前程序的地址不重叠。上面分配了64KB的空间来做解压时的数据缓存。

/\*

\* 检查是否会覆盖内核映像本身

\* r4 = 最终解压后的内核首地址

\* r5 = zImage 的运行时首地址，一般为 0x30008000

\* r2 = end of malloc space分配空间的结束地址（并且处于本映像的前面）

\* 基本要求：r4 >= r2 或者 r4 + 映像长度 <= r5

(1)vmlinux 的起始地址大于 zImage 运行时所需的最大地址（ r2 ） , 那么直接将 zImage 解压到 vmlinux 的目标地址

cmp r4, r2

bhs wont\_overwrite /\*如果r4大于或等于r2的话\*/

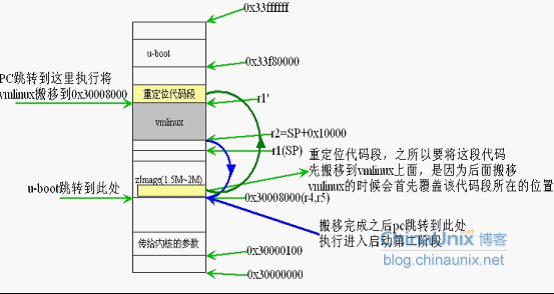
(2)zImage 的起始地址大于 vmlinux 的目标起始地址加上 vmlinux 大小（ 4M ）的地址，所以将 zImage 直接解压到 vmlinux 的目标地址

add r0, r4, #4096\*1024 @ 4MB largest kernel size

cmp r0, r5

bls wont\_overwrite /\*如果r4 + 映像长度 <= r5 的话\*/

@ 前两种方案通常都不成立，不会跳转到wont\_overwrite标号处，会继续走如下分支，其解压后的内存分配示意图如下：



mov r5, r2 @ decompress after malloc space

mov r0, r5 /\*解压程序从分配空间后面存放 \*/

mov r3, r7

bl decompress\_kernel

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*进入decompress\_kernel\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

@ decompress\_kernel共有4个参数，解压的内核地址、缓存区首地址、缓存区尾地址、和芯片ID，返回解压缩代码的长度。

decompress\_kernel(ulg output\_start, ulg free\_mem\_ptr\_p, ulg free\_mem\_ptr\_end\_p,

int arch\_id)

{

output\_data = (uch \*)output\_start;/\* Points to kernel start \*/

free\_mem\_ptr = free\_mem\_ptr\_p; /\*保存缓存区首地址\*/

free\_mem\_ptr\_end = free\_mem\_ptr\_end\_p;/\*保存缓冲区结束地址\*/

\_\_machine\_arch\_type = arch\_id;

arch\_decomp\_setup();

makecrc(); /\*镜像校验\*/

putstr("Uncompressing Linux...");

gunzip(); /\*通过free\_mem\_ptr来解压缩\*/

putstr(" done, booting the kernel.\n");

return output\_ptr; /\*返回镜像的大小\*/

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*从decompress\_kernel函数返回\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

add r0, r0, #127 + 128

bic r0, r0, #127 @ align the kernel length对齐内核长度

/\*

\* r0 = 解压后内核长度

\* r1-r3 = 未使用

\* r4 = 真正内核执行地址 0x30008000

\* r5 = 临时解压内核Image的起始地址

\* r6 = 处理器ID

\* r7 = 体系结构ID

\* r8 = 参数列表 0x30000100

\* r9-r14 = 未使用

\*/

@ 完成了解压缩之后，由于内核没有解压到正确的地址，最后必须通过代码搬移来搬到指定的地址0x30008000。搬运过程中有

@ 可能会覆盖掉现在运行的重定位代码，所以必须将这段代码搬运到安全的地方，

@ 这里搬运到的地址是解压缩了的代码的后面r5+r0的位置。

add r1, r5, r0 @ end of decompressed kernel 解压内核的结束地址

adr r2, reloc\_start

ldr r3, LC1 @ LC1: .word reloc\_end - reloc\_start 表示reloc\_start段代码的大小

add r3, r2, r3

1: ldmia r2!, {r9 - r14} @ copy relocation code

stmia r1!, {r9 - r14}

ldmia r2!, {r9 - r14}

stmia r1!, {r9 - r14}

cmp r2, r3

blo 1b

bl cache\_clean\_flush @清 cache

ARM(add pc, r5, r0) @ call relocation code 跳转到重定位代码开始执行

@ 在此处会调用重定位代码reloc\_start来将Image 的代码从缓冲区r5帮运到最终的目的地r4:0x30008000处

reloc\_start: add r9, r5, r0 @r9中存放的是临时解压内核的末尾地址

sub r9, r9, #128 @ 不拷贝堆栈

mov r1, r4 @r1中存放的是目的地址0x30008000

1:

.rept 4

ldmia r5!, {r0, r2, r3, r10 - r14} @ relocate kernel

stmia r1!, {r0, r2, r3, r10 - r14} /\*搬运内核Image的过程\*/

.endr

cmp r5, r9

blo 1b

mov sp, r1 /\*留出堆栈的位置\*/

add sp, sp, #128 @ relocate the stack

call\_kernel: bl cache\_clean\_flush @清除cache

bl cache\_off @关闭cache

mov r0, #0 @ must be zero

mov r1, r7 @ restore architecture number

mov r2, r8 @ restore atags pointer

@ 这里就是最终我们从zImage跳转到Image的伟大一跳了，跳之前准备好r0,r1,r2

mov pc, r4 @ call kernel

到此kernel的第一阶段zImage 解压缩阶段已经执行完。