可以结合《hi3536 uboot引导内核全过程》一文一起看

# 一、uImage生成过程

## 1、vmlinux

根目录下vmlinux为kernel未经过任何处理的原始可执行文件。根据arch/arm/kernel/vmlinux.lds连接文件生成：

. = PAGE\_OFFSET + TEXT\_OFFSET; = 0xC0000000 + 0x8000 内核运行时虚拟起始地址(3G/1G内核情况，如果2G/2G则为0x80000000+0x8000)

#define PAGE\_OFFSET UL(CONFIG\_PAGE\_OFFSET)

textofs-y := 0x00008000

TEXT\_OFFSET := $(textofs-y) (arch/arm/Makefile文件中)

## 2、uImage依赖关系

在arch/arm/boot/Makefile：

$(obj)/uImage: $(obj)/zImage

$(obj)/zImage: $(obj)/compressed/vmlinux

$(obj)/compressed/vmlinux: $(obj)/Image

$(obj)/Image: vmlinux

在arch/arm/boot/compressed/Makefile：

$(obj)/vmlinux: $(obj)/vmlinux.lds $(obj)/$(HEAD) $(obj)/piggy.lzma.o $(addprefix $(obj)/, $(OBJS))

$(obj)/piggy.lzma: $(obj)/../Image

$(obj)/piggy.lzma.o: $(obj)/piggy.lzma

compressed/vmlinux：根据当前目录下的vmlinux.lds生成，其中有一个重要字段：

. = TEXT\_START; （=0，why?

arch/arm/boot/compressed/Makefile

/\* Decompressors running from RAM should not define ZTEXTADDR. Decompressors running directly from ROM or Flash must define ZTEXTADDR \*/

TEXT\_START = $(ZTEXTADDR) = 0；

BSS\_START = $(ZBSSADDR) = ALIGN(8)

）

.piggydata : {

\*(.piggydata)

}

存放的就是上面的piggy.lzma，其arch/arm/boot/compressed/piggy.lzma.S内容如下：

.section .piggydata,#alloc

.globl input\_data

input\_data:

.incbin "arch/arm/boot/compressed/piggy.lzma"

.globl input\_data\_end

input\_data\_end:

在自解压判断和解压时就会用到上面红色地址。

## 3、uImage生成流程

//根据vmlinux.lds生成（流程图见下一章内存布局图部分）

根目录vmlinux ——>

//去掉vmlinux调试信息

Arch/arm/boot/Image ——>

//lzma压缩末尾4个字节代表Image大小，汇编代码通过input\_data\_end获取

Arch/arm/boot/compressed/piggy.lzma——>

//根据vmlinux.lds，将压缩内核、引导代码、自解压代码等重新组成新的vmlinux

Arch/arm/boot/compressed/piggy.lzma.o+head.o+misc.o+ decompress.o——> Arch/arm/boot/compressed/vmlinux——>

//将新的vmlinux通过objcopy生成zImage

Arch/arm/boot/zImage——>

//添加64字节头生成uImage

Arch/arm/boot/uImage

arch/arm/mach-hi3536/Makefile.boot

zreladdr-y := 0x40008000 //需要和uboot保持一致

params\_phys-y := 0x00000100 //需要和uboot保持一致

initrd\_phys-y := 0x00800000

arch/arm/boot/Makefile

/\* Note: the following conditions must always be true:

ZRELADDR == virt\_to\_phys(PAGE\_OFFSET + TEXT\_OFFSET)

PARAMS\_PHYS must be within 4MB of ZRELADDR

INITRD\_PHYS must be in RAM \*/

ZRELADDR := $(zreladdr-y) = 0x40008000 //zImage解压后最终Image的起始地址，uboot引导时zImage一般也存放在这个地址，所以后面涉及到解压前当前程序搬运问题。

PARAMS\_PHYS := $(params\_phys-y) = 0x00000100 //内核参数地址

INITRD\_PHYS := $(initrd\_phys-y) = 0x00800000

# 二、内核代码自解压过程

## 1、arch/arm/boot/compressed/vmlinux.lds

连接文件内容

OUTPUT\_ARCH(arm)

ENTRY(\_start)

SECTIONS

{

/DISCARD/ : {

\*(.ARM.exidx\*)

\*(.ARM.extab\*)

/\*

\* Discard any r/w data - this produces a link error if we have any,

\* which is required for PIC decompression. Local data generates

\* GOTOFF relocations, which prevents it being relocated independently

\* of the text/got segments.

\*/

\*(.data)

}

. = 0; // TEXT\_START连接地址0x0，因为从RAM直接启动

\_text = .;

.text : {

\_start = .;

\*(.start) //入口地址

\*(.text)

\*(.text.\*)

\*(.fixup)

\*(.gnu.warning)

\*(.glue\_7t)

\*(.glue\_7)

}

.rodata : {

\*(.rodata)

\*(.rodata.\*)

}

.piggydata : {

\*(.piggydata) //存放piggy.lzma字段

}

. = ALIGN(4);

\_etext = .;

.got.plt : { \*(.got.plt) }

\_got\_start = .;

.got : { \*(.got) }

\_got\_end = .;

/\* ensure the zImage file size is always a multiple of 64 bits \*/

/\* (without a dummy byte, ld just ignores the empty section) \*/

.pad : { BYTE(0); . = ALIGN(8); }

\_edata = .; //结束地址

. = ALIGN(8);

\_\_bss\_start = .;

.bss : { \*(.bss) }

\_end = .;

. = ALIGN(8); /\* the stack must be 64-bit aligned \*/

.stack : { \*(.stack) }

.stab 0 : { \*(.stab) }

.stabstr 0 : { \*(.stabstr) }

.stab.excl 0 : { \*(.stab.excl) }

.stab.exclstr 0 : { \*(.stab.exclstr) }

.stab.index 0 : { \*(.stab.index) }

.stab.indexstr 0 : { \*(.stab.indexstr) }

.comment 0 : { \*(.comment) }

}

上面的.got，那么GOT表是什么？

GOT（Global Offset Table）表中每一项都是本运行模块要引用的一个全局变量或函数的地址。可以用GOT表来间接引用全局变量、函数，也可以把GOT表的首地址作为一个基准，用相对于该基准的偏移量来引用静态变量、静态函数

## 2、arch/arm/boot/compressed/head.S

以下是对该自解压代码分析：

.section ".start", #alloc, #execinstr //以下代码存放在vmlinux.lds的.start字段

/\*

\* sort out different calling conventions

\*/

.align

.arm //总是进入arm状态

start:

.type start,#function //定义start函数

.rept 7

mov r0, r0

.endr

ARM( mov r0, r0 )

ARM( b 1f )

THUMB( adr r12, BSYM(1f) )

THUMB( bx r12 )

.word 0x016f2818 @Magic numbers to help the loader

.word start //加载和运行zImage的绝对地址（编译时确定），在vmlinux.lds中zImage连接地址为0

.word \_edata //zImage的结束地址，在vmlinux.lds中可以看到

THUMB( .thumb )

1:

mrs r9, cpsr

#ifdef CONFIG\_ARM\_VIRT\_EXT

bl \_\_hyp\_stub\_install @ get into SVC mode, reversibly

#endif

mov r7, r1 //保存机器码到R7，这个由uboot传递的第二个参数

mov r8, r2 //保存参数列表指针到R8，这个由uboot传递的第三个参数

…

ldr r4, =zreladdr //最终Image的执行起始地址存放在R4，上一节中提到该值为0x40008000，0x40000000为物理内存起始地址，0x8000为偏移

#if defined(CONFIG\_ARCH\_HI3536) \

|| defined(CONFIG\_ARCH\_HI3536\_SLAVE) \

|| defined(CONFIG\_ARCH\_HI3521A)

/\*

\* set SMP bit ACTLR register to enable I cache and D cache

\*/

mrc p15, 0, r0, c1, c0, 1

orr r0, #(1 << 6)

mcr p15, 0, r0, c1, c0, 1

#endif

bl cache\_on

/\*开始cache以及MMU,并且会做一些页表初始化的工作，然后在head.S结束时关闭mmu和cache。初始化页表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是等价映射的（也就是将物理地址映射到等值的虚拟地址上）。\*/

restart: adr r0, LC0

/\* 获取LC0的运行地址，依次将相应的连接地址放入到下面的寄存器中。这里之所以是restart主要是判断解压后是否会覆盖当前镜像，如果会就先搬运当前镜像到解压结束地址+堆栈之后，再从这里开始运行解压内核。\*/

ldmia r0, {r1, r2, r3, r6, r10, r11, r12}

ldr sp, [r0, #28] //此时r0中存放的还是LC0的运行地址，所以加28后正好是LC0数组中的.L\_user\_stack\_end的值(栈的结束地址)，在head.S的结尾定义

{

.type LC0, #object

LC0: .word LC0 @ r1

.word \_\_bss\_start @ r2

.word \_end @ r3

.word \_edata @ r6 //zImage镜像的结束地址

.word input\_data\_end - 4 @ r10在上一节中提到过这个地址存放的就是image大小

.word \_got\_start @ r11

.word \_got\_end @ ip

.word .L\_user\_stack\_end @ sp

.size LC0, . - LC0

}

/\* 编译时连接地址从0x0开始，这里计算出当前实际运行的物理地址 \*/

sub r0, r0, r1 @ 计算偏移量

add r6, r6, r0 @ \_edata 实际运行地址

add r10, r10, r0 @ 获取Image大小存放的实际物理地址

/\*

内核编译系统将Image的大小4字节以小端形式添加到压缩数据的末尾。这里是将解压后的内核数据大小正确的放入R9寄存器里。

\*/

ldrb r9, [r10, #0]

ldrb lr, [r10, #1]

orr r9, r9, lr, lsl #8

ldrb lr, [r10, #2]

ldrb r10, [r10, #3]

orr r9, r9, lr, lsl #16

orr r9, r9, r10, lsl #24

/\* malloc获取的内存空间位于重定向的栈指针之上（64K max） \*/

add sp, sp, r0 //使用r0修正sp，得到堆栈的实际结束物理地址，为什么是结束地址？因为栈是向下生长的。

add r10, sp, #0x10000 //执行这句之前sp中存放的是栈的结束地址，执行后，r10中存放的是堆空间的结束物理地址

…

/\* 检查解压是否会自我覆盖的情况

\* r4 = 最终执行地址（解压后内核起始地址）

\* r9 = 解压后的内核大小

\* r10 = 当前执行映像的结束地址, 包括bss/stack/malloc空间

\* 判断是否会自我覆盖的的情况：

\* A. r4 - 16k页目录 >= r10 -> OK, 即最终执行地址在当前映像之后

\* B. r4 + 内核大小 <= wont\_overwrite的地址（当前位置PC） -> OK 即最终执行地址在当前映像之前

\* C. 不满足上面条件的就会自我覆盖，需要搬运当前映像到不会覆盖的地方运行。通常是这种情况，zImage和Image同一个地址。uboot将zImage搬运到0x40008000运行，上面的r4最终执行地址也是该地址。

\*/

add r10, r10, #16384

cmp r4, r10

bhs wont\_overwrite

add r10, r4, r9

adr r9, wont\_overwrite

cmp r10, r9

bls wont\_overwrite

只有发生自我覆盖时才会运行这里，否则直接运行wont\_overwrite开始的地方。

/\*

\* 将当前运行映像搬运到解压后的内核结束地址后面

\* r6 = \_edata

\* r10 = 解压后内核结束地址

\* 由于拷贝时当前映像向后执行移动，为了防止源数据和目标数据重叠，从后向前拷贝代码。

\*/

/\*

\* Bump to the next 256-byte boundary with the size of

\* the relocation code added. This avoids overwriting

\* ourself when the offset is small.

\*/

add r10, r10, #((reloc\_code\_end - restart + 256) & ~255)

bic r10, r10, #255

/\* Get start of code we want to copy and align it down. \*/

adr r5, restart //获取需要搬运的当前映像起始位置，并32B对齐

bic r5, r5, #31

/\* Relocate the hyp vector base if necessary \*/

#ifdef CONFIG\_ARM\_VIRT\_EXT

mrs r0, spsr

and r0, r0, #MODE\_MASK

cmp r0, #HYP\_MODE

bne 1f

bl \_\_hyp\_get\_vectors

sub r0, r0, r5

add r0, r0, r10

bl \_\_hyp\_set\_vectors

1:

#endif

sub r9, r6, r5 @ r6-r5 = 拷贝大小

add r9, r9, #31 @ 对齐

bic r9, r9, #31 @ ... of 32 bytes

add r6, r9, r5 //r6=当前映像需要搬运的结束地址

add r9, r9, r10 //当前映像搬运后的结束地址

//搬运当前映像，不包含bss/stack/malloc空间

1: ldmdb r6!, {r0 - r3, r10 - r12, lr}

cmp r6, r5

stmdb r9!, {r0 - r3, r10 - r12, lr}

bhi 1b

/\* r6存放当映像和目标映像地址偏移量，修正SP和实现代码跳转 \*/

sub r6, r9, r6

#ifndef CONFIG\_ZBOOT\_ROM

修正sp地址，修正后sp指向重定向后的代码的栈区的结束地址（栈向下生长），栈区后面紧跟的就是堆空间

add sp, sp, r6

#endif

bl cache\_clean\_flush //清除缓存

adr r0, BSYM(restart) //获得restart的运行地址

add r0, r0, r6 //获得重定向后的代码的restart的物理地址

mov pc, r0 //跳到重定向后的代码的restart处开始执行

/\* 在上面的跳转之后，程序又从restart开始。

\* 但这次在检查自我覆盖的时候，新的执行位置必然满足上面B情况

\* 所以必然直接跳到了下面的wont\_overwrite执行

\*/

wont\_overwrite:

/\*

\*如果delta（当前映像地址与编译时的地址偏移）为0, 我们运行的地址就是编译时确定的地址。但我们不一致，所以需要重定向。

\* r0 = delta

\* r2 = BSS start

\* r3 = BSS end

\* r4 = kernel execution address

\* r5 = appended dtb size (0 if not present)

\* r7 = architecture ID

\* r8 = atags pointer

\* r11 = GOT start

\* r12 = GOT end

\* sp = stack pointer

\*/

orrs r1, r0, r5

beq not\_relocated //如果delta为0，无须对GOT表项和BSS进行重定位，否则下面进行重新定向

add r11, r11, r0 //重定位GOT start

add r12, r12, r0 //重定位GOT end

add r2, r2, r0 //重定位BSS start

add r3, r3, r0 //重定位BSS end

//重定位所有GOT表的入口项

1: ldr r1, [r11, #0] @ relocate entries in the GOT

add r1, r1, r0 @ This fixes up C references

cmp r1, r2 @ if entry >= bss\_start &&

cmphs r3, r1 @ bss\_end > entry

addhi r1, r1, r5 @ entry += dtb size

str r1, [r11], #4 @ next entry

cmp r11, r12

blo 1b

// 重定向bbs

add r2, r2, r5

add r3, r3, r5

//到这里，当前映像搬运和重定向完成，开始内核解压

not\_relocated: mov r0, #0

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

str r0, [r2], #4

str r0, [r2], #4

str r0, [r2], #4

cmp r2, r3

blo 1b

/\*

\* C运行环境充分建立，设置部分指针，开始内核解压

\* r4 = 解压后内核执行地址0x40008000

\* r7 = 机器码

\* r8 = 内核参数列表指针

\*/

mov r0, r4

mov r1, sp @ malloc space above stack

add r2, sp, #0x10000 @ 64k max

mov r3, r7

bl decompress\_kernel //r0/r1/r2/r3分别是参数传递给decompress\_kernel，该函数是C语言解压函数。

{

ret = do\_decompress(input\_data, input\_data\_end - input\_data,

output\_data, error);

input\_data：上一节arch/arm/boot/compressed/piggy.lzma.S中存放piggy.lzma的起始地址

output\_data：r4的地址0x40008000

}

bl cache\_clean\_flush //清除缓存

bl cache\_off //关闭cache

mov r1, r7 @ restore architecture number

mov r2, r8 @ restore atags pointer

mrs r0, spsr @ Get saved CPU boot mode

and r0, r0, #MODE\_MASK

cmp r0, #HYP\_MODE @ if not booted in HYP mode...

bne \_\_enter\_kernel @ 进入解压后的内核运行

{

\_\_enter\_kernel:

mov r0, #0 @ must be 0

ARM( mov pc, r4 ) @ call kernel r4=解压后内核执行地址0x40008000

THUMB( bx r4 ) @ 跳转到0x40008000开始执行Image内核，入口为stext，细节见下文。

}

adr r12, .L\_\_hyp\_reentry\_vectors\_offset

ldr r0, [r12]

add r0, r0, r12

bl \_\_hyp\_set\_vectors

\_\_HVC(0) @ otherwise bounce to hyp mode

b . @ should never be reached

.align 2

.L\_\_hyp\_reentry\_vectors\_offset: .long \_\_hyp\_reentry\_vectors - .

## 3、内存布局图



uImage生成过程



uboot加载uImage内存布局图

uboot加载内核后，最终zImage在0x40008000位置开始执行内核代码，细节见《hi3536 uboot引导内核全过程》。



内核代码自解压内存布局图

解压前需要判断解压后是否会覆盖，如果会覆盖则需要将自解压代码搬运到不会覆盖内存中运行，然后再解压内核代码。这里zImage和Image的入口地址都是0x40008000，所以需要如图所示进行自我搬运，搬运位置根据Image的大小计算出不会覆盖的地方，然后PC调到转搬运后的地址开始执行解压，然后在跳转到Image入口执行真正的内核代码。

# 三、内核启动代码分析

## 1、arch/arm/kernel/head.S

/\*

\* swapper\_pg\_dir是初始化页表的虚拟地址

\* 页表放在KERNEL\_RAM\_VADDR前16K，所以KERNEL\_RAM\_VADDR >= PAGE\_OFFSET + 0x4000，vmlinux.lds是0xC0008000（虚拟地址）

\*/

#define KERNEL\_RAM\_VADDR (PAGE\_OFFSET + TEXT\_OFFSET) = 0xC0008000

#if (KERNEL\_RAM\_VADDR & 0xffff) != 0x8000

#error KERNEL\_RAM\_VADDR must start at 0xXXXX8000

#endif

#define PG\_DIR\_SIZE 0x4000 //页全局目录大小16K

#define PMD\_ORDER 2 //占据4个页

.globl swapper\_pg\_dir

.equ swapper\_pg\_dir, KERNEL\_RAM\_VADDR - PG\_DIR\_SIZE

.macro pgtbl, rd, phys

add \rd, \phys, #TEXT\_OFFSET - PG\_DIR\_SIZE

.endm

/\*

真正内核入口点：

\* 上一节解压后即跳转到这里，要求：

\* MMU = off, D-cache = off, I-cache = dont care, r0 = 0,

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

\* See linux/arch/arm/tools/mach-types 完整的机器码列表 for r1.

\*/

.arm

\_\_HEAD

ENTRY(stext)

THUMB( adr r9, BSYM(1f) ) @ Kernel is always entered in ARM.

THUMB( bx r9 ) @ If this is a Thumb-2 kernel,

THUMB( .thumb ) @ switch to Thumb now.

THUMB(1: )

#ifdef CONFIG\_ARM\_VIRT\_EXT

bl \_\_hyp\_stub\_install

#endif

@ ensure svc mode and all interrupts masked

safe\_svcmode\_maskall r9

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

bl \_\_lookup\_processor\_type @ r5=procinfo r9=cupid 判断是否是内核支持的cpu类型（即ARM核，这里为armv7）

{

arch/arm/kernel/head-common.S中定义：

\_\_lookup\_processor\_type

从.long \_\_proc\_info\_begin

.long \_\_proc\_info\_end 获取所有支持列表

这里armv7则由arch/arm/mm/proc-v7.S文件生成列表

}

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

THUMB( it eq ) @ force fixup-able long branch encoding

beq \_\_error\_p @ yes, error 'p' //不匹配就出错，执行不下去

#ifndef CONFIG\_XIP\_KERNEL

adr r3, 2f

ldmia r3, {r4, r8}

sub r4, r3, r4 @ (PHYS\_OFFSET - PAGE\_OFFSET)

add r8, r8, r4 @ PHYS\_OFFSET //计算物理起始地址

#else

ldr r8, =PHYS\_OFFSET @ always constant in this case

#endif

/\*

\* r1 = machine no, r2 = atags or dtb,

\* r8 = phys\_offset, r9 = cpuid, r10 = procinfo

\*/

bl \_\_vet\_atags //判断r2的合法性

#ifdef CONFIG\_SMP\_ON\_UP

bl \_\_fixup\_smp //修正smp内核在单处理器上的运作

#endif

#ifdef CONFIG\_ARM\_PATCH\_PHYS\_VIRT

bl \_\_fixup\_pv\_table

#endif

bl \_\_create\_page\_tables //创建内核全局页表（虚拟内存中会详细讲解），初始化内核代码要运行的页表映射（内核代码连接的是虚拟地址见System.map）

/\*

\* The following calls CPU specific code in a position independent

\* manner. See arch/arm/mm/proc-\*.S for details. r10 = base of

\* xxx\_proc\_info structure selected by \_\_lookup\_processor\_type

\* above. On return, the CPU will be ready for the MMU to be

\* turned on, and r0 will hold the CPU control register value.

\*/

ldr r13, =\_\_mmap\_switched //将mmu使能后的跳转地址放入LR，后面执行完后跳转到该函数执行

adr lr, BSYM(1f) @ return (PIC) address

mov r8, r4 @ set TTBR1 to swapper\_pg\_dir

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

THUMB( add r12, r10, #PROCINFO\_INITFUNC )

THUMB( mov pc, r12 )

1: b \_\_enable\_mmu //使能mmu

ENDPROC(stext)

.ltorg

#ifndef CONFIG\_XIP\_KERNEL

2: .long .

.long PAGE\_OFFSET

#endif

## 2、arch/arm/kernel/head-common.S

/\* 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/dtb pointer

\* r9 = processor ID

\*/

\_\_INIT

\_\_mmap\_switched:

adr r3, \_\_mmap\_switched\_data //依次将\_\_mmap\_switched\_data的连接地址加载

ldmia r3!, {r4, r5, r6, r7}

cmp r4, r5 @ Copy data segment if needed

1: cmpne r5, r6

ldrne fp, [r4], #4

strne fp, [r5], #4

bne 1b

mov fp, #0 @ Clear BSS (and zero fp)

1: cmp r6, r7

strcc fp, [r6],#4

bcc 1b

ARM( ldmia r3, {r4, r5, r6, r7, sp})

THUMB( ldmia r3, {r4, r5, r6, r7} )

THUMB( ldr sp, [r3, #16] )

str r9, [r4] @ Save processor ID 保存id到processor\_id变量里

str r1, [r5] @ Save machine type 保存机器码到\_\_machine\_arch\_type变量里

str r2, [r6] @ Save atags pointer 保存参数列表到\_\_atags\_pointer变量

cmp r7, #0

bicne r4, r0, #CR\_A @ Clear 'A' bit

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

b start\_kernel //跳转到C语言启动内核的代码处

ENDPROC(\_\_mmap\_switched)

.align 2

.type \_\_mmap\_switched\_data, %object

\_\_mmap\_switched\_data:

.long \_\_data\_loc @ r4

.long \_sdata @ r5

.long \_\_bss\_start @ r6

.long \_end @ r7

.long processor\_id @ r4

.long \_\_machine\_arch\_type @ r5

.long \_\_atags\_pointer @ r6

#ifdef CONFIG\_CPU\_CP15

.long cr\_alignment @ r7

#else

.long 0 @ r7

#endif

.long init\_thread\_union + THREAD\_START\_SP @ sp //start\_kernel使用的内核堆栈，也可以认为是空闲任务0使用的内核堆栈8K

.size \_\_mmap\_switched\_data, . - \_\_mmap\_switched\_data

## 3、init/main.c—start\_kernel

该函数涉及对内核所需要的各种初始化。

asmlinkage void \_\_init start\_kernel(void)

{

char \* command\_line;

extern const struct kernel\_param \_\_start\_\_\_param[], \_\_stop\_\_\_param[];

/\*

\* Need to run as early as possible, to initialize the

\* lockdep hash:

\*/

lockdep\_init();

smp\_setup\_processor\_id();

debug\_objects\_early\_init();

/\*

\* Set up the the initial canary ASAP:

\*/

boot\_init\_stack\_canary();

cgroup\_init\_early();

local\_irq\_disable();

early\_boot\_irqs\_disabled = true;

/\*

\* Interrupts are still disabled. Do necessary setups, then

\* enable them

\*/

boot\_cpu\_init();

page\_address\_init();

//打印内核信息

pr\_notice("%s", linux\_banner);

//重要，见后面一节（4）详细说明

setup\_arch(&command\_line);

mm\_init\_owner(&init\_mm, &init\_task);

mm\_init\_cpumask(&init\_mm);

//保存未操作过的命令行内容

setup\_command\_line(command\_line);

setup\_nr\_cpu\_ids();

setup\_per\_cpu\_areas();

smp\_prepare\_boot\_cpu(); /\* arch-specific boot-cpu hooks \*/

//建立所有内存管理区链表（DMA，NORMAL，HIGHMEM 3种管理区）

build\_all\_zonelists(NULL, NULL);

page\_alloc\_init();

pr\_notice("Kernel command line: %s\n", boot\_command\_line);

//分析系统能够辨别的一些早期参数，以未做任何修改的boot\_command\_line为基础

parse\_early\_param();

//解析命令行各个参数，mem、rootfs等

parse\_args("Booting kernel", static\_command\_line, \_\_start\_\_\_param,

\_\_stop\_\_\_param - \_\_start\_\_\_param,

-1, -1, &unknown\_bootoption);

jump\_label\_init();

/\*

\* These use large bootmem allocations and must precede

\* kmem\_cache\_init()

\*/

setup\_log\_buf(0);

//顺序扫描进程链表并检查进程描述符的pid字段是可行但是相当低效，为了加速查找，引入了4个散列表。为每种hash表申请空间，并将内存虚拟基址分别存放在pid\_hash数组中。

pidhash\_init();

vfs\_caches\_init\_early();

/\* 将放在\_\_start\_ex\_table到\_\_stop\_ex\_table之间的\*(\_ex\_table)区域中的struct exception\_table\_entry型全局结构变量按照insn成员变量值从小到大排序。

\*/

sort\_main\_extable();

trap\_init();

//内存初始化（包括伙伴算法和slab分配器初始化），释放前面标志为保留的所有页面，这个函数结束后就不能在使用alloc\_bootmem，alloc\_bootmem\_low alloc\_bootmem\_pages等申请低端内存的函数申请内存。

mm\_init();

/\* 初始化每个处理器的可运行队列。同时通过init\_idle(current, smp\_processor\_id());设置系统初始化进程即0号idle进程，当前的运行都可以算作是在idle这个进程，使用init\_task(静态分配的第一个任务描述符)和init\_thread\_union（静态分配的第一个指向任务描述符的堆栈结构体）（init/init\_task.c）

\*/

sched\_init();

//禁止内核抢占，因为这是调度还没准备好，直到cpu\_idle为止

preempt\_disable();

if (WARN(!irqs\_disabled(), "Interrupts were enabled \*very\* early, fixing it\n"))

local\_irq\_disable();

idr\_init\_cache();

perf\_event\_init();

rcu\_init();

tick\_nohz\_init();

radix\_tree\_init();

/\* init some links before init\_ISA\_irqs() \*/

early\_irq\_init();

//初始化中断，调用的是平台中断接口machine\_desc->init\_irq(); （hi3536\_gic\_init\_irq）。

init\_IRQ();

//初始化当前cpu的读、拷贝、更新数据结构全局变量per\_cpu\_rcu\_data和per\_cpu\_rcu\_bh\_data

tick\_init();

//初始化当前处理器的时间向量，打开TIMER\_SOFTIRQ定时器软中断

init\_timers();

//高精度定时器初始化

hrtimers\_init();

//软中断初始化，打开TASKLET\_SOFTIRQ和HI\_SOFTIRQ tasklet所需要的软中断

softirq\_init();

timekeeping\_init();

//初始化体系结构硬件定时器

time\_init();

//用来对系统剖析的，在系统调试的时候有用

profile\_init();

call\_function\_init();

WARN(!irqs\_disabled(), "Interrupts were enabled early\n");

early\_boot\_irqs\_disabled = false;

//使能IRQ中断

local\_irq\_enable();

kmem\_cache\_init\_late();

//初始化系统控制台结构，该函数执行后调用printk函数将log\_buf中所有符合打印的信息打印到控制台。

console\_init();

if (panic\_later)

panic(panic\_later, panic\_param);

lockdep\_info();

/\*

\* Need to run this when irqs are enabled, because it wants

\* to self-test [hard/soft]-irqs on/off lock inversion bugs

\* too:

\*/

locking\_selftest();

#ifdef CONFIG\_BLK\_DEV\_INITRD

if (initrd\_start && !initrd\_below\_start\_ok &&

page\_to\_pfn(virt\_to\_page((void \*)initrd\_start)) < min\_low\_pfn) {

pr\_crit("initrd overwritten (0x%08lx < 0x%08lx) - disabling it.\n",

page\_to\_pfn(virt\_to\_page((void \*)initrd\_start)),

min\_low\_pfn);

initrd\_start = 0;

}

#endif

page\_cgroup\_init();

debug\_objects\_mem\_init();

kmemleak\_init();

setup\_per\_cpu\_pageset();

//cpu系统的BogMIPS数值，即处理器每秒执行的指令数

numa\_policy\_init();

if (late\_time\_init)

late\_time\_init();

sched\_clock\_init();

calibrate\_delay();

//为了循环使用PID编号，内容通过pidmap管理已经分配的pid和空闲PID

pidmap\_init();

//为匿名逆序内存区域链表结构分配一个高速缓存内存

anon\_vma\_init();

#ifdef CONFIG\_X86

if (efi\_enabled(EFI\_RUNTIME\_SERVICES))

efi\_enter\_virtual\_mode();

#endif

thread\_info\_cache\_init(); //do nothing

cred\_init();

//创建进程相关的初始化

fork\_init(totalram\_pages);

proc\_caches\_init();

buffer\_init();

key\_init();

security\_init();

dbg\_late\_init();

vfs\_caches\_init(totalram\_pages);

signals\_init();

/\* rootfs populating might need page-writeback \*/

page\_writeback\_init();

#ifdef CONFIG\_PROC\_FS

proc\_root\_init();

#endif

cgroup\_init();

cpuset\_init();

taskstats\_init\_early();

delayacct\_init();

check\_bugs();

//add by hlb

gs\_proc\_init();

acpi\_early\_init(); /\* before LAPIC and SMP init \*/

sfi\_init\_late();

if (efi\_enabled(EFI\_RUNTIME\_SERVICES)) {

efi\_late\_init();

efi\_free\_boot\_services();

}

ftrace\_init();

//重要，见后面一节（5）详细说明

rest\_init();

}

## 4、arch/arm/kernel/setup.c——setup\_arch

void \_\_init setup\_arch(char \*\*cmdline\_p)

{

struct machine\_desc \*mdesc;

//获取arm处理相关信息

setup\_processor();

mdesc = setup\_machine\_fdt(\_\_atags\_pointer);

//获取uboot传递进来的参数列表，解析各个tags，其中一个重要的是boot\_command\_line

if (!mdesc)

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

machine\_desc = mdesc;

machine\_name = mdesc->name;

//设置DMA内存管理区，当前没有用

setup\_dma\_zone(mdesc);

if (mdesc->restart\_mode)

reboot\_setup(&mdesc->restart\_mode);

//初始化init\_mm内存描述符，后面的值在vmlinux.lds中编译时确定

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;

//将cmd\_line传递出去，后续解析使用

strlcpy(cmd\_line, boot\_command\_line, COMMAND\_LINE\_SIZE);

\*cmdline\_p = cmd\_line;

parse\_early\_param();

sort(&meminfo.bank, meminfo.nr\_banks, sizeof(meminfo.bank[0]), meminfo\_cmp, NULL);

sanity\_check\_meminfo();

arm\_memblock\_init(&meminfo, mdesc);

//为系统内存创建页表，初始化内存

paging\_init(mdesc);

request\_standard\_resources(mdesc);

….

}

## 5、init/main.c——rest\_init

### （1）reset\_init

该函数创建init内核进程（1号进程）和kthreadd内核进程（2号进程），原来的为0号系统启动进程进入空闲状态。

static noinline void \_\_init\_refok rest\_init(void)

{

int pid;

rcu\_scheduler\_starting();

//创建1号init进程

kernel\_thread(kernel\_init, NULL, CLONE\_FS | CLONE\_SIGHAND);

numa\_default\_policy();

//创建2号kthreadd进程

pid = kernel\_thread(kthreadd, NULL, CLONE\_FS | CLONE\_FILES);

rcu\_read\_lock();

kthreadd\_task = find\_task\_by\_pid\_ns(pid, &init\_pid\_ns);

rcu\_read\_unlock();

complete(&kthreadd\_done);

//初始化当前任务到空闲进程

init\_idle\_bootup\_task(current);

//禁止调度抢占

schedule\_preempt\_disabled();

//进入空闲内核

cpu\_startup\_entry(CPUHP\_ONLINE);

}

kernel\_init内核进程主要运行/sbin/init命令，完成内核态到用户态的切换，执行/etc下面的各种脚本，实现程序启动。

### （2）多核CPU启动次CPU过程

由kernel\_init完成除CPU0以外其他CPU的初始化及启动工作，其基本流程如下：

kernel\_init —> kernel\_init\_freeable —> smp\_init():

void \_\_init smp\_init(void)

{

unsigned int cpu;

//为所有CPU初始化idle进程

idle\_threads\_init();

/\* FIXME: This should be done in userspace --RR \*/

for\_each\_present\_cpu(cpu) { //循环处理每个CPU

if (num\_online\_cpus() >= setup\_max\_cpus)

break;

if (!cpu\_online(cpu)) //过滤CPU0

cpu\_up(cpu); //其他CPU初始化函数

}

/\* Any cleanup work \*/

printk(KERN\_INFO "Brought up %ld CPUs\n", (long)num\_online\_cpus());

smp\_cpus\_done(setup\_max\_cpus); //初始化结束

}

cpu\_up() —> \_cpu\_up() —> \_\_cpu\_up() —> boot\_secondary() (arch/arm/kernel/smp.c):

int \_\_cpuinit boot\_secondary(unsigned int cpu, struct task\_struct \*idle)

{

if (smp\_ops.smp\_boot\_secondary) //与体系结构相关

return smp\_ops.smp\_boot\_secondary(cpu, idle); = hi3536\_boot\_secondary

return -ENOSYS;

}

static int \_\_cpuinit hi3536\_boot\_secondary(unsigned int cpu,

struct task\_struct \*idle)

{

unsigned long timeout;

set\_scu\_boot\_addr(0x00000000,

(unsigned int)virt\_to\_phys(hi3536\_secondary\_startup)); 启动次CPU，汇编函数

…

}

hi3536\_secondary\_startup —> secondary\_startup —> \_\_secondary\_switched —> secondary\_start\_kernel：（arch/arm/kernel/smp.c，与体系结构无关）

实现次CPU的初始化，然后进入空闲任务。

# 附录A

参考资料：

http://www.cnblogs.com/pengdonglin137/p/3838245.html