<http://blog.chinaunix.net/uid-20543672-id-3018233.html>

CPU在bootloader的帮助下将内核载入到了内存中，并开始执行。当然，bootloader必须为zImage做好必要的准备：

|  |  |
| --- | --- |
| 1． CPU 寄存器的设置： | R0＝0；R1＝Machine ID(即Machine Type Number，定义在linux/arch/arm/tools/mach-types)；  R2＝内核启动参数在 RAM 中起始基地址； |
| 2． CPU 模式： | 必须禁止中断（IRQs和FIQs）；  CPU 必须 SVC 模式； |
| 3． Cache 和 MMU 的设置： | MMU 必须关闭；  指令 Cache 可以打开也可以关闭；  数据 Cache 必须关闭； |

 载自：[一、Boot Loader的概念和功能](http://blog.chinaunix.net/space.php?uid=20543672&do=blog&id=94371)

知道内核zImage生成的朋友一定知道：**真正的内核执行映像其实是在编译时生成arch/$(ARCH)/boot/文件夹中的Image文件（bin文件），而zImage其实是将这个可执行文件作为数据段包含在了自身中，而zImage的代码功能就是将这个数据（Image）正确地解压到编译时确定的位置中去，并跳到Image中运行。**

所以实现bootloader引导的压缩映像zImage的入口是由arch/arm /boot/compressed/vmlinux.lds决定的（这个文件是由vmlinux.lds.in生成的）。所以从vmlinux.lds.in中可以看出压缩映像的入口在哪：

......

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)

}

. = TEXT\_START;

\_text = .;

.text : {

\_start = .;

\*(.start)

\*(.text)

......

我们可以在arch/arm/boot/compressed/head.S找到这个start入口，这样就可以从这里开始用代码分析的方法研究bootloader跳转到压缩内核映像后的自解压启动过程：

再看到MMU设置的时候，我只研究了armv7的指令。看这些代码，必须对ARM的MMU有一定的了解，建议参考ARMv7的构架手册和网上的一份PDF《ARM MMU中文详解》（就是ARM手册中MMU部分的翻译）

/\*

\* linux/arch/arm/boot/compressed/head.S

\*

\* Copyright (C) 1996-2002 Russell King

\* Copyright (C) 2004 Hyok S. Choi (MPU support)

\*

\* This program is free software; you can redistribute it and/or modify

\* it under the terms of the GNU General Public License version 2 as

\* published by the Free Software Foundation.

\*/

#include

/\*

\* 调试宏

\*

\* 注意：这些宏必须不包含那些非100%可重定位的代码

\* 任何试图这样做的结果是导致程序崩溃

\* 当打开调试时请选择以下一个使用

\*/

#ifdef DEBUG /\* 调试宏-中间层 \*/

#if defined(CONFIG\_DEBUG\_ICEDCC) /\* 使用内部调试协处理器CP14 \*/

#if defined(CONFIG\_CPU\_V6) || defined(CONFIG\_CPU\_V6K) || defined(CONFIG\_CPU\_V7)

.macro loadsp, rb, tmp

.endm

.macro writeb, ch, rb

mcr p14, 0, \ch, c0, c5, 0

.endm

#elif defined(CONFIG\_CPU\_XSCALE)

.macro loadsp, rb, tmp

.endm

.macro writeb, ch, rb

mcr p14, 0, \ch, c8, c0, 0

.endm

#else

.macro loadsp, rb, tmp

.endm

.macro writeb, ch, rb

mcr p14, 0, \ch, c1, c0, 0

.endm

#endif

#else /\* 使用串口作为调试通道 \*/

#include /\* 包含构架相关的的调试宏的汇编文件 调试宏-底层 \*/

.macro writeb, ch, rb

senduart \ch, \rb

.endm

#if defined(CONFIG\_ARCH\_SA1100)

.macro loadsp, rb, tmp

mov \rb, #0x80000000 @ physical base address

#ifdef CONFIG\_DEBUG\_LL\_SER3

add \rb, \rb, #0x00050000 @ Ser3

#else

add \rb, \rb, #0x00010000 @ Ser1

#endif

.endm

#elif defined(CONFIG\_ARCH\_S3C2410)

.macro loadsp, rb, tmp

mov \rb, #0x50000000

add \rb, \rb, #0x4000 \* CONFIG\_S3C\_LOWLEVEL\_UART\_PORT

.endm

#else

.macro loadsp, rb, tmp

addruart \rb, \tmp

.endm

#endif

#endif

#endif /\* DEBUG \*/

/\* 调试宏-上层 \*/

.macro kputc,val /\* 打印字符 \*/

mov r0, \val

bl putc

.endm

.macro kphex,val,len /\* 打印十六进制数 \*/

mov r0, \val

mov r1, #\len

bl phex

.endm

.macro debug\_reloc\_start /\* 重定位内核调试宏-开始 \*/

#ifdef DEBUG

kputc #'\n'

kphex r6, 8 /\* 处理器 id \*/

kputc #':'

kphex r7, 8 /\* 构架 id \*/

#ifdef CONFIG\_CPU\_CP15

kputc #':'

mrc p15, 0, r0, c1, c0

kphex r0, 8 /\* 控制寄存器 \*/

#endif

kputc #'\n'

kphex r5, 8 /\* 解压后的内核起始地址 \*/

kputc #'-'

kphex r9, 8 /\* 解压后的内核结束地址 \*/

kputc #'>'

kphex r4, 8 /\* 内核执行地址 \*/

kputc #'\n'

#endif

.endm

.macro debug\_reloc\_end /\* 重定位内核调试宏-结束 \*/

#ifdef DEBUG

kphex r5, 8 /\* 内核结束地址 \*/

kputc #'\n'

mov r0, r4

bl memdump /\* 打印内核起始处 256 字节 \*/

#endif

.endm

.section ".start", #alloc, #execinstr

/\*

\* 清理不同的调用约定

\*/

.align

.arm @ 启动总是进入ARM状态

### start:

.type start,#function

.rept 7

mov r0, r0

.endr

ARM( mov r0, r0 )

ARM( b 1f )

THUMB( adr r12, BSYM(1f) )

THUMB( bx r12 )

.word 0x016f2818 @ 用于boot loader的魔数

.word start @ 加载/运行zImage的绝对地址（编译时确定）

.word \_edata @ zImage结束地址

THUMB( .thumb )

1: mov r7, r1 @ 保存构架ID到r7（此前由bootloader放入r1）

mov r8, r2 @ 保存内核启动参数地址到r8（此前由bootloader放入r2）

#ifndef \_\_ARM\_ARCH\_2\_\_

/\*

\* 通过Angel调试器启动 - 必须进入 SVC模式且关闭FIQs/IRQs

\* (numeric definitions from angel arm.h source).

\* 如果进入时在user模式下，我们只需要做这些

\*/

mrs r2, cpsr @ 获取当前模式

tst r2, #3 @ 判断是否是user模式

bne not\_angel

mov r0, #0x17 @ angel\_SWIreason\_EnterSVC

ARM( swi 0x123456 ) @ angel\_SWI\_ARM

THUMB( svc 0xab ) @ angel\_SWI\_THUMB

not\_angel:

mrs r2, cpsr @ 关闭中断

orr r2, r2, #0xc0 @ 以保护调试器的运作

msr cpsr\_c, r2

#else

teqp pc, #0x0c000003 @ 关闭中断（此外bootloader已设置模式为SVC）

#endif

/\*

\* 注意一些缓存的刷新和其他事务可能需要在这里完成

\* - is there an Angel SWI call for this?

\*/

/\*

\* 一些构架的特定代码可以在这里被连接器插入，

\* 但是不应使用 r7（保存构架ID）, r8（保存内核启动参数地址）, and r9.

\*/

.text

/\*

\* 此处确定解压后的内核映像的绝对地址（物理地址），保存于r4

\* 由于配置的不同可能有的结果

\* （1）定义了CONFIG\_AUTO\_ZRELADDR

\* ZRELADDR是已解压内核最终存放的物理地址

\* 如果AUTO\_ZRELADDR被选择了, 这个地址将会在运行是确定：

\* 将当pc值和0xf8000000做与操作，

\* 并加上TEXT\_OFFSET（内核最终存放的物理地址与内存起始的偏移）

\* 这里假定zImage被放在内存开始的128MB内

\* （2）没有定义CONFIG\_AUTO\_ZRELADDR

\* 直接使用zreladdr（此值位于arch/arm/mach-xxx/Makefile.boot文件确定）

\*/

#ifdef CONFIG\_AUTO\_ZRELADDR

@ 确定内核映像地址

mov r4, pc

and r4, r4, #0xf8000000

add r4, r4, #TEXT\_OFFSET

#else

ldr r4, =zreladdr

#endif

bl cache\_on /\* 开启缓存（以及MMU） \*/

restart: adr r0, LC0

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

ldr sp, [r0, #28]

/\*

\* 我们可能运行在一个与编译时定义的不同地址上，

\* 所以我们必须修正变量指针

\*/

sub r0, r0, r1 @ 计算偏移量

add r6, r6, r0 @ 重新计算\_edata

add r10, r10, r0 @ 重新获得压缩后的内核大小数据位置

/\*

\* 内核编译系统将解压后的内核大小数据

\* 以小端格式

\* 附加在压缩数据的后面(其实是“gzip -f -9”命令的结果)

\* 下面代码的作用是将解压后的内核大小数据正确地放入r9中（避免了大小端问题）

\*/

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

/\*

\* 下面代码的作用是将正确的当前执行映像的结束地址放入r10

\*/

#ifndef CONFIG\_ZBOOT\_ROM

/\* 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

add r10, sp, #0x10000

#else

/\*

\* 如果定义了 ZBOOT\_ROM， bss/stack 是非可重定位的,

\* 但有些人依然可以将其放在RAM中运行,

\* 这时我们可以参考 \_edata.

\*/

mov r10, r6

#endif

/\*

\* 检测我们是否会发生自我覆盖的问题

\* r4 = 解压后的内核起始地址（最终执行位置）

\* r9 = 解压后内核的大小

\* r10 = 当前执行映像的结束地址, 包含了 bss/stack/malloc 空间（假设是非XIP执行的）

\* 我们的基本需求是:

\* （若最终执行位置r4在当前映像之后）r4 - 16k 页目录 >= r10 -> OK

\* （若最终执行位置r4在当前映像之前）r4 + 解压后的内核大小 <= 当前位置 (pc) -> OK

\* 如果上面的条件不满足，就会自我覆盖，必须先搬运当前映像

\*/

add r10, r10, #16384

cmp r4, r10 @ 假设最终执行位置r4在当前映像之后

bhs wont\_overwrite

add r10, r4, r9 @ 假设最终执行位置r4在当前映像之前

ARM( cmp r10, pc ) @ r10 = 解压后的内核结束地址

THUMB( mov lr, pc )

THUMB( cmp r10, lr )

bls wont\_overwrite

/\*

\* 将当前的映像重定向到解压后的内核之后（会发生自我覆盖时才执行，否则就被跳过）

\* r6 = \_edata（已校正）

\* r10 = 解压后的内核结束地址

\* 因为我们要把当前映像向后移动, 所以我们必须由后往前复制代码，

\* 以防原数据和目标数据的重叠

\*/

/\*

\* 将解压后的内核结束地址r10扩展（reloc\_code\_end - restart），

\* 并对齐到下一个256B边界。

\* 这样避免了当搬运的偏移较小时的自我覆盖

\*/

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

bic r10, r10, #255

/\* 获取需要搬运的当前映像的起始位置r5，并向下做32B对齐. \*/

adr r5, restart

bic r5, r5, #31

sub r9, r6, r5 @ \_edata - restart（已向下对齐）= 需要搬运的大小

add r9, r9, #31

bic r9, r9, #31 @ 做32B对齐 ，r9 = 需要搬运的大小

add r6, r9, r5 @ r6 = 当前映像需要搬运的结束地址

add r9, r9, r10 @ r9 = 当前映像搬运的目的地的结束地址

/\* 搬运当前执行映像，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

/\* 保存偏移量，用来修改sp和实现代码跳转 \*/

sub r6, r9, r6

#ifndef CONFIG\_ZBOOT\_ROM

/\* cache\_clean\_flush 可能会使用栈，所以重定向sp指针 \*/

add sp, sp, r6

#endif

bl cache\_clean\_flush @ 刷新缓存

/\* 通过搬运的偏移和当前的实际 restart 地址来实现代码跳转\*/

adr r0, BSYM(restart)

add r0, r0, r6

mov pc, r0

/\* 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

\* 最终执行位置r4在当前映像之前，r4 + 压缩后的内核大小 <= 当前位置 (pc)

\* 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.

\* r0 = delta

\* r2 = BSS start（编译值）

\* r3 = BSS end（编译值）

\* r4 = 内核最终运行的物理地址

\* r7 = 构架ID(bootlodaer传递值)

\* r8 = 内核启动参数指针(bootlodaer传递值)

\* r11 = GOT start（编译值）

\* r12 = GOT end（编译值）

\* sp = stack pointer（修正值）

\*/

teq r0, #0 @测试delta值

beq not\_relocated @如果delta为0，无须对GOT表项和BSS进行重定位

add r11, r11, r0 @重定位GOT start

add r12, r12, r0 @重定位GOT end

#ifndef CONFIG\_ZBOOT\_ROM

/\*

\* 如果内核配置 CONFIG\_ZBOOT\_ROM = n,

\* 我们必须修正BSS段的指针

\* 注意：sp已经被修正

\*/

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] @ 重定位GOT表的入口项

add r1, r1, r0 @ 这个修正了 C 引用

str r1, [r11], #4

cmp r11, r12

blo 1b

#else

/\*

\* 重定位所有GOT表的入口项.

\* 我们只重定向在（已重定向后）BSS段外的入口

\*/

1: ldr r1, [r11, #0] @ 重定位GOT表的入口项

cmp r1, r2 @ entry < bss\_start ||

cmphs r3, r1 @ \_end < entry table

addlo r1, r1, r0 @ 这个修正了 C 引用

str r1, [r11], #4

cmp r11, r12

blo 1b

#endif

/\*

\* 至此当前映像的搬运和调整已经完成

\* 可以开始真正的工作的

\*/

not\_relocated: mov r0, #0

1: str r0, [r2], #4 @ 清零 bss（初始化BSS段）

str r0, [r2], #4

str r0, [r2], #4

str r0, [r2], #4

cmp r2, r3

blo 1b

/\*

\* C运行时环境已经充分建立.

\* 设置一些指针就可以解压内核了.

\* r4 = 内核最终运行的物理地址

\* r7 = 构架ID

\* r8 = 内核启动参数指针

\*

\* 下面对r0～r3的配置是decompress\_kernel函数对应参数

\* r0 = 解压后的输出位置首地址

\* r1 = 可用RAM空间首地址

\* r2 = 可用RAM空间结束地址

\* r3 = 构架ID

\* 就是这个decompress\_kernel（C函数）输出了"Uncompressing Linux..."

\* 以及" done, booting the kernel.\n"

\*/

mov r0, r4

mov r1, sp @ malloc 获取的内存空间位于栈指针之上

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

mov r3, r7

bl decompress\_kernel

/\*

\* decompress\_kernel(misc.c)--调用-->

\* do\_decompress(decompress.c)--调用-->

\* decompress(../../../../lib/decompress\_xxxx.c根据压缩方式的配置而不同)

\*/

/\*

\* 以下是为跳入解压后的内核，再次做准备（恢复解压前的状态）

\*/

bl cache\_clean\_flush

bl cache\_off @ 数据缓存必须关闭（内核的要求）

mov r0, #0 @ r0必须为0

mov r1, r7 @ 恢复构架ID到r1

mov r2, r8 @ 恢复内核启动参数指针到r2

mov pc, r4 @ 跳入解压后的内核映像(Image)入口（arch/arm/kernel/head.S）

/\*

\* 以下是为了确定当前运行时的地址和编译时确定的地址偏差，

\* 而将编译时确定的映像数据保存如下，用于检测对比

\*/

.align 2

.type LC0, #object

LC0: .word LC0 @ r1

.word \_\_bss\_start @ r2

.word \_end @ r3

.word \_edata @ r6

.word input\_data\_end - 4 @ r10 (inflated size location)

.word \_got\_start @ r11

.word \_got\_end @ ip

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

.size LC0, . - LC0

#ifdef CONFIG\_ARCH\_RPC

.globl params

params: ldr r0, =0x10000100 @ params\_phys for RPC

mov pc, lr

.ltorg

.align

#endif

/\*

\* 开启缓存.

\* 我们必须创建页表（并开启MMU）才可以开启数据和指令缓存。

\* 我们把页表（节描述符）放在内核执行地址前16k（0x4000）的空间中,

\* 且我们希望没人会去用这段地址空间.

\* 如果我们使用了,可能会出问题的!

\*

\* 进入时,

\* r4 = 内核最终运行的物理地址

\* r7 = 构架ID

\* r8 = 内核启动参数指针

\* 退出时,

\* r0, r1, r2, r3, r9, r10, r12 被修改

\* 此例程必须保护:

\* r4, r7, r8

\*/

.align 5

cache\_on: mov r3, #8 @ 调用cache\_on 函数

b call\_cache\_fn

/\*

\* Initialize the highest priority protection region, PR7

\* to cover all 32bit address and cacheable and bufferable.

\*/

\_\_armv4\_mpu\_cache\_on:

mov r0, #0x3f @ 4G, the whole

mcr p15, 0, r0, c6, c7, 0 @ PR7 Area Setting

mcr p15, 0, r0, c6, c7, 1

mov r0, #0x80 @ PR7

mcr p15, 0, r0, c2, c0, 0 @ D-cache on

mcr p15, 0, r0, c2, c0, 1 @ I-cache on

mcr p15, 0, r0, c3, c0, 0 @ write-buffer on

mov r0, #0xc000

mcr p15, 0, r0, c5, c0, 1 @ I-access permission

mcr p15, 0, r0, c5, c0, 0 @ D-access permission

mov r0, #0

mcr p15, 0, r0, c7, c10, 4 @ drain write buffer

mcr p15, 0, r0, c7, c5, 0 @ flush(inval) I-Cache

mcr p15, 0, r0, c7, c6, 0 @ flush(inval) D-Cache

mrc p15, 0, r0, c1, c0, 0 @ read control reg

@ ...I .... ..D. WC.M

orr r0, r0, #0x002d @ .... .... ..1. 11.1

orr r0, r0, #0x1000 @ ...1 .... .... ....

mcr p15, 0, r0, c1, c0, 0 @ write control reg

mov r0, #0

mcr p15, 0, r0, c7, c5, 0 @ flush(inval) I-Cache

mcr p15, 0, r0, c7, c6, 0 @ flush(inval) D-Cache

mov pc, lr

\_\_armv3\_mpu\_cache\_on:

mov r0, #0x3f @ 4G, the whole

mcr p15, 0, r0, c6, c7, 0 @ PR7 Area Setting

mov r0, #0x80 @ PR7

mcr p15, 0, r0, c2, c0, 0 @ cache on

mcr p15, 0, r0, c3, c0, 0 @ write-buffer on

mov r0, #0xc000

mcr p15, 0, r0, c5, c0, 0 @ access permission

mov r0, #0

mcr p15, 0, r0, c7, c0, 0 @ invalidate whole cache v3

/\*

\* ?? ARMv3 MMU does not allow reading the control register,

\* does this really work on ARMv3 MPU?

\*/

mrc p15, 0, r0, c1, c0, 0 @ read control reg

@ .... .... .... WC.M

orr r0, r0, #0x000d @ .... .... .... 11.1

/\* ?? this overwrites the value constructed above? \*/

mov r0, #0

mcr p15, 0, r0, c1, c0, 0 @ write control reg

/\* ?? invalidate for the second time? \*/

mcr p15, 0, r0, c7, c0, 0 @ invalidate whole cache v3

mov pc, lr

/\*

\* 初始化MMU页表

\* 内核最终运行的物理地址向下16K的空间

\* 存放可以寻址4G空间节描述符

\* （16KB/4B=4K个描述符，每个描述符映射1MB空间，4K\*1MB = 4GB）

\* 进入时,

\* r4 = 内核最终运行的物理地址

\* r7 = 构架ID

\* r8 = 内核启动参数指针

\* 退出时,

\* r0, r1, r2, r3, r9, r10 被修改

\* 此例程必须保护:

\* r4, r7, r8

\*/

\_\_setup\_mmu: sub r3, r4, #16384 @ 页目录大小为16K

bic r3, r3, #0xff @ 页目录指针向下对齐

bic r3, r3, #0x3f00 @ 对齐方式-16KB

/\*

\* 对于这个对齐，是MMU硬件的要求

\* 转换表基址寄存器（CP15的寄存器2）保存着第一级转换表基址的物理地址。

\* 只有bits[31:14]有效，bits[13:0]应该是零（SBZ）。

\* 所以第一级表必须16KB对齐。

\*/

/\*

\* 初始化页表, 仅针对RAM（最大到256MB）开启

\* 缓存（cacheable）和缓冲（bufferable）位

\* r3 = 页目录基址(内核最终运行的物理地址向下16K的位置)

\*/

mov r0, r3 @ 页目录指针给r0

mov r9, r0, lsr #18

mov r9, r9, lsl #18 @ 通过移位清零低18bit，得到RAM基地址（推测值,r9）

add r10, r9, #0x10000000 @ 加一个合理的RAM大小（猜测值） = RAM结束地址（猜测值,r10）

mov r1, #0x12

orr r1, r1, #3 << 10 @ 初始化节描述符r1 = 0b110000010010(完全访问:0域:XN:节)

add r2, r3, #16384 @ r2 = 内核最终运行的物理地址（可能）

1: cmp r1, r9 @ if virt > start of RAM（针对RAM开启缓存和缓冲）

#ifdef CONFIG\_CPU\_DCACHE\_WRITETHROUGH

orrhs r1, r1, #0x08 @ 设置 cacheable

#else

orrhs r1, r1, #0x0c @ 设置 cacheable, bufferable

#endif

cmp r1, r10 @ if virt > end of RAM

bichs r1, r1, #0x0c @ 清除 cacheable, bufferable

str r1, [r0], #4 @ 设置节描述符-1:1 映射（虚拟地址 == 物理地址）

add r1, r1, #1048576 @ r1 + 1MB（每节管理的地址长度）下一个节描述符

teq r0, r2

bne 1b

/\*

\* 如果我们在flash中运行, 那么我们一定要为我们当前的代码开启缓存。

\* 我们映射2MB的代码，

\* 所以对于多达1MB压缩的内核没有映射重叠的问题？？

\* 如果我们在RAM中运行, 那么我们只需要完成上面的工作即可，下面重复了.

\*/

mov r1, #0x1e

orr r1, r1, #3 << 10 @ 初始化节描述符r1 = 0b110000011110(完全访问:0域:XN:cacheable:bufferable:节)

mov r2, pc

mov r2, r2, lsr #20 @ 当前执行地址的节基址

orr r1, r1, r2, lsl #20 @ 生成节描述符

add r0, r3, r2, lsl #2 @ 获得页目录中相应的入口

str r1, [r0], #4 @ 设置节描述符-1:1 映射（虚拟地址 == 物理地址）

add r1, r1, #1048576 @ r1 + 1MB（每节管理的地址长度）下一个节描述符

str r1, [r0] @ 设置节描述符（只做2MB映射）

mov pc, lr

ENDPROC(\_\_setup\_mmu)

\_\_arm926ejs\_mmu\_cache\_on:

#ifdef CONFIG\_CPU\_DCACHE\_WRITETHROUGH

mov r0, #4 @ put dcache in WT mode

mcr p15, 7, r0, c15, c0, 0

#endif

\_\_armv4\_mmu\_cache\_on:

mov r12, lr

#ifdef CONFIG\_MMU

bl \_\_setup\_mmu

mov r0, #0

mcr p15, 0, r0, c7, c10, 4 @ drain write buffer

mcr p15, 0, r0, c8, c7, 0 @ flush I,D TLBs

mrc p15, 0, r0, c1, c0, 0 @ read control reg

orr r0, r0, #0x5000 @ I-cache enable, RR cache replacement

orr r0, r0, #0x0030

#ifdef CONFIG\_CPU\_ENDIAN\_BE8

orr r0, r0, #1 << 25 @ big-endian page tables

#endif

bl \_\_common\_mmu\_cache\_on

mov r0, #0

mcr p15, 0, r0, c8, c7, 0 @ flush I,D TLBs

#endif

mov pc, r12

\_\_armv7\_mmu\_cache\_on:

mov r12, lr @保存lr到r12

#ifdef CONFIG\_MMU

mrc p15, 0, r11, c0, c1, 4 @ 读取CP15的ID\_MMFR0（内存模块特性）寄存器

tst r11, #0xf @ 测试VMSA（虚拟内存系统构架）A8 = 0x3

blne \_\_setup\_mmu @ 如果VMSA不是0xf，就进入mmu页表初始化（节模式）

mov r0, #0

mcr p15, 0, r0, c7, c10, 4 @ 数据内存屏障（保证上面的写操作完成才继续）

tst r11, #0xf @ 测试VMSA（虚拟内存系统构架）A8 = 0x3

mcrne p15, 0, r0, c8, c7, 0 @ flush I,D TLBs缓存

#endif

mrc p15, 0, r0, c1, c0, 0 @ 读系统控制寄存器

orr r0, r0, #0x5000 @ I-cache 使能, RR cache replacement

orr r0, r0, #0x003c @ write buffer

#ifdef CONFIG\_MMU

#ifdef CONFIG\_CPU\_ENDIAN\_BE8

orr r0, r0, #1 << 25 @ 大端模式页表

#endif

orrne r0, r0, #1 @ 设置MMU 开启位

movne r1, #-1

mcrne p15, 0, r3, c2, c0, 0 @ 载入页表基址到TTBR0

mcrne p15, 0, r1, c3, c0, 0 @ 载入域访问控制数据到DACR（所有域都是Manager，所以XN会被忽略）

#endif

mcr p15, 0, r0, c1, c0, 0 @ 写系统控制寄存器

mrc p15, 0, r0, c1, c0, 0 @ 回读系统控制寄存器

mov r0, #0

mcr p15, 0, r0, c7, c5, 4 @ 指令同步屏障（确保上面指令完成才返回）

mov pc, r12 @ 此处返回（此时MMU已启用，RAM缓存已开启）

\_\_fa526\_cache\_on:

mov r12, lr

bl \_\_setup\_mmu

mov r0, #0

mcr p15, 0, r0, c7, c7, 0 @ Invalidate whole cache

mcr p15, 0, r0, c7, c10, 4 @ drain write buffer

mcr p15, 0, r0, c8, c7, 0 @ flush UTLB

mrc p15, 0, r0, c1, c0, 0 @ read control reg

orr r0, r0, #0x1000 @ I-cache enable

bl \_\_common\_mmu\_cache\_on

mov r0, #0

mcr p15, 0, r0, c8, c7, 0 @ flush UTLB

mov pc, r12

\_\_arm6\_mmu\_cache\_on:

mov r12, lr

bl \_\_setup\_mmu

mov r0, #0

mcr p15, 0, r0, c7, c0, 0 @ invalidate whole cache v3

mcr p15, 0, r0, c5, c0, 0 @ invalidate whole TLB v3

mov r0, #0x30

bl \_\_common\_mmu\_cache\_on

mov r0, #0

mcr p15, 0, r0, c5, c0, 0 @ invalidate whole TLB v3

mov pc, r12

\_\_common\_mmu\_cache\_on:

#ifndef CONFIG\_THUMB2\_KERNEL

#ifndef DEBUG

orr r0, r0, #0x000d @ Write buffer, mmu

#endif

mov r1, #-1

mcr p15, 0, r3, c2, c0, 0 @ load page table pointer

mcr p15, 0, r1, c3, c0, 0 @ load domain access control

b 1f

.align 5 @ cache line aligned

1: mcr p15, 0, r0, c1, c0, 0 @ load control register

mrc p15, 0, r0, c1, c0, 0 @ and read it back to

sub pc, lr, r0, lsr #32 @ properly flush pipeline

#endif

#define PROC\_ENTRY\_SIZE (4\*5)

/\*

\* 这里是为不同的处理器提供遵循可重定向缓存支持的函数

\* 这是一个通用的为 定位入口 和 跳入一个（从块起始处到）特定偏移的指令 的钩子函数。

\* 请注意这是一个位置无关代码。

\*

\* r1 = 被修改

\* r2 = 被修改

\* r3 = 相对每个入口的功能函数位置偏移（on:#08|off:#12|flush:#16）

\* r9 = 被修改

\* r12 = 被修改

\*/

call\_cache\_fn: adr r12, proc\_types

#ifdef CONFIG\_CPU\_CP15

mrc p15, 0, r9, c0, c0 @ 动态获取处理器ID

#else

ldr r9, =CONFIG\_PROCESSOR\_ID @ 使用预编译的处理器ID

#endif

1: ldr r1, [r12, #0] @ 获取ID值

ldr r2, [r12, #4] @ 获取对应的掩码

eor r1, r1, r9 @ (real ^ match) 检测是否匹配

tst r1, r2 @ & mask 将检测结果做掩码

ARM( addeq pc, r12, r3 ) @ 如果匹配就调用缓存函数

THUMB( addeq r12, r3 )

THUMB( moveq pc, r12 ) @ call cache function

add r12, r12, #PROC\_ENTRY\_SIZE @ 如果不匹配就跳过这个入口，进入下个测试

b 1b

/\*

\* 缓存操作表. 这些是最基本的:

\* - CPU ID 匹配

\* - CPU ID 掩码

\* - 'cache on' 方法代码

\* - 'cache off' 方法代码

\* - 'cache flush' 方法代码

\*

\* 我们通过这个公式匹配入口: ((real\_id ^ match) & mask) == 0

\*

\* 写通式缓存一般只需要 'on' 和 'off' 方法

\* 回写式缓存必须有 flush 方法定义

\*

\*/

.align 2

.type proc\_types,#object

proc\_types:

.word 0x41560600 @ ARM6/610

.word 0xffffffe0

W(b) \_\_arm6\_mmu\_cache\_off @ 可以使用但是较慢

W(b) \_\_arm6\_mmu\_cache\_off

mov pc, lr

THUMB( nop )

@ b \_\_arm6\_mmu\_cache\_on @ 未测试

@ b \_\_arm6\_mmu\_cache\_off

@ b \_\_armv3\_mmu\_cache\_flush

.word 0x00000000 @ old ARM ID

.word 0x0000f000

mov pc, lr

THUMB( nop )

mov pc, lr

THUMB( nop )

mov pc, lr

THUMB( nop )

.word 0x41007000 @ ARM7/710

.word 0xfff8fe00

W(b) \_\_arm7\_mmu\_cache\_off

W(b) \_\_arm7\_mmu\_cache\_off

mov pc, lr

THUMB( nop )

.word 0x41807200 @ ARM720T (写通式)

.word 0xffffff00

W(b) \_\_armv4\_mmu\_cache\_on

W(b) \_\_armv4\_mmu\_cache\_off

mov pc, lr

THUMB( nop )

.word 0x41007400 @ ARM74x

.word 0xff00ff00

W(b) \_\_armv3\_mpu\_cache\_on

W(b) \_\_armv3\_mpu\_cache\_off

W(b) \_\_armv3\_mpu\_cache\_flush

.word 0x41009400 @ ARM94x

.word 0xff00ff00

W(b) \_\_armv4\_mpu\_cache\_on

W(b) \_\_armv4\_mpu\_cache\_off

W(b) \_\_armv4\_mpu\_cache\_flush

.word 0x41069260 @ ARM926EJ-S (v5TEJ)

.word 0xff0ffff0

W(b) \_\_arm926ejs\_mmu\_cache\_on

W(b) \_\_armv4\_mmu\_cache\_off

W(b) \_\_armv5tej\_mmu\_cache\_flush

.word 0x00007000 @ ARM7 IDs

.word 0x0000f000

mov pc, lr

THUMB( nop )

mov pc, lr

THUMB( nop )

mov pc, lr

THUMB( nop )

@ 以下使用新的 ID 系统.

.word 0x4401a100 @ sa110 / sa1100

.word 0xffffffe0

W(b) \_\_armv4\_mmu\_cache\_on

W(b) \_\_armv4\_mmu\_cache\_off

W(b) \_\_armv4\_mmu\_cache\_flush

.word 0x6901b110 @ sa1110

.word 0xfffffff0

W(b) \_\_armv4\_mmu\_cache\_on

W(b) \_\_armv4\_mmu\_cache\_off

W(b) \_\_armv4\_mmu\_cache\_flush

.word 0x56056900

.word 0xffffff00 @ PXA9xx

W(b) \_\_armv4\_mmu\_cache\_on

W(b) \_\_armv4\_mmu\_cache\_off

W(b) \_\_armv4\_mmu\_cache\_flush

.word 0x56158000 @ PXA168

.word 0xfffff000

W(b) \_\_armv4\_mmu\_cache\_on

W(b) \_\_armv4\_mmu\_cache\_off

W(b) \_\_armv5tej\_mmu\_cache\_flush

.word 0x56050000 @ Feroceon

.word 0xff0f0000

W(b) \_\_armv4\_mmu\_cache\_on

W(b) \_\_armv4\_mmu\_cache\_off

W(b) \_\_armv5tej\_mmu\_cache\_flush

#ifdef CONFIG\_CPU\_FEROCEON\_OLD\_ID

/\* this conflicts with the standard ARMv5TE entry \*/

.long 0x41009260 @ Old Feroceon

.long 0xff00fff0

b \_\_armv4\_mmu\_cache\_on

b \_\_armv4\_mmu\_cache\_off

b \_\_armv5tej\_mmu\_cache\_flush

#endif

.word 0x66015261 @ FA526

.word 0xff01fff1

W(b) \_\_fa526\_cache\_on

W(b) \_\_armv4\_mmu\_cache\_off

W(b) \_\_fa526\_cache\_flush

@ 这些匹配构架ID

.word 0x00020000 @ ARMv4T

.word 0x000f0000

W(b) \_\_armv4\_mmu\_cache\_on

W(b) \_\_armv4\_mmu\_cache\_off

W(b) \_\_armv4\_mmu\_cache\_flush

.word 0x00050000 @ ARMv5TE

.word 0x000f0000

W(b) \_\_armv4\_mmu\_cache\_on

W(b) \_\_armv4\_mmu\_cache\_off

W(b) \_\_armv4\_mmu\_cache\_flush

.word 0x00060000 @ ARMv5TEJ

.word 0x000f0000

W(b) \_\_armv4\_mmu\_cache\_on

W(b) \_\_armv4\_mmu\_cache\_off

W(b) \_\_armv5tej\_mmu\_cache\_flush

.word 0x0007b000 @ ARMv6

.word 0x000ff000

W(b) \_\_armv4\_mmu\_cache\_on

W(b) \_\_armv4\_mmu\_cache\_off

W(b) \_\_armv6\_mmu\_cache\_flush

.word 0x000f0000 @ new CPU Id

.word 0x000f0000

W(b) \_\_armv7\_mmu\_cache\_on

W(b) \_\_armv7\_mmu\_cache\_off

W(b) \_\_armv7\_mmu\_cache\_flush

.word 0 @ 未识别类型

.word 0

mov pc, lr

THUMB( nop )

mov pc, lr

THUMB( nop )

mov pc, lr

THUMB( nop )

.size proc\_types, . - proc\_types

/\*

\* 如果你获得了一个 "非常量的表达式".如果汇编器从这行返回" 申明"错误

\* 请检查下你是否偶尔在应该使用“W(b)”的地方写了"b"指令

\* 这是一个缓存方法跳转表的对齐检查机制

\* 在写汇编的时候可以借鉴

\*/

.if (. - proc\_types) % PROC\_ENTRY\_SIZE != 0

.error "The size of one or more proc\_types entries is wrong."

.endif

/\*

\* 关闭缓存和MMU. ARMv3不支持控制寄存器的读取，

\* 但ARMv4支持.

\*

\* 在退出时,

\* r0, r1, r2, r3, r9, r12 被篡改

\* 这个例程必须保护:

\* r4, r7, r8

\*/

.align 5

cache\_off: mov r3, #12 @ 缓存关闭函数

b call\_cache\_fn

\_\_armv4\_mpu\_cache\_off:

mrc p15, 0, r0, c1, c0

bic r0, r0, #0x000d

mcr p15, 0, r0, c1, c0 @ turn MPU and cache off

mov r0, #0

mcr p15, 0, r0, c7, c10, 4 @ drain write buffer

mcr p15, 0, r0, c7, c6, 0 @ flush D-Cache

mcr p15, 0, r0, c7, c5, 0 @ flush I-Cache

mov pc, lr

\_\_armv3\_mpu\_cache\_off:

mrc p15, 0, r0, c1, c0

bic r0, r0, #0x000d

mcr p15, 0, r0, c1, c0, 0 @ turn MPU and cache off

mov r0, #0

mcr p15, 0, r0, c7, c0, 0 @ invalidate whole cache v3

mov pc, lr

\_\_armv4\_mmu\_cache\_off:

#ifdef CONFIG\_MMU

mrc p15, 0, r0, c1, c0

bic r0, r0, #0x000d

mcr p15, 0, r0, c1, c0 @ turn MMU and cache off

mov r0, #0

mcr p15, 0, r0, c7, c7 @ invalidate whole cache v4

mcr p15, 0, r0, c8, c7 @ invalidate whole TLB v4

#endif

mov pc, lr

\_\_armv7\_mmu\_cache\_off:

mrc p15, 0, r0, c1, c0 @ 读取系统控制寄存器SCTLR

#ifdef CONFIG\_MMU

bic r0, r0, #0x000d @ 清零MMU和cache使能位

#else

bic r0, r0, #0x000c @ 清零cache使能位

#endif

mcr p15, 0, r0, c1, c0 @ 关闭MMU和cache

mov r12, lr @ 保存lr到r12

bl \_\_armv7\_mmu\_cache\_flush

mov r0, #0

#ifdef CONFIG\_MMU

mcr p15, 0, r0, c8, c7, 0 @ 废止整个TLB

#endif

mcr p15, 0, r0, c7, c5, 6 @ 废止BTC

mcr p15, 0, r0, c7, c10, 4 @ 数据同步屏障

mcr p15, 0, r0, c7, c5, 4 @ 指令同步屏障（确保上面指令完成才返回）

mov pc, r12

\_\_arm6\_mmu\_cache\_off:

mov r0, #0x00000030 @ ARM6 control reg.

b \_\_armv3\_mmu\_cache\_off

\_\_arm7\_mmu\_cache\_off:

mov r0, #0x00000070 @ ARM7 control reg.

b \_\_armv3\_mmu\_cache\_off

\_\_armv3\_mmu\_cache\_off:

mcr p15, 0, r0, c1, c0, 0 @ turn MMU and cache off

mov r0, #0

mcr p15, 0, r0, c7, c0, 0 @ invalidate whole cache v3

mcr p15, 0, r0, c5, c0, 0 @ invalidate whole TLB v3

mov pc, lr

/\*

\* 清空和flush缓存以保持一致性

\*

\* 退出时,

\* r1, r2, r3, r9, r10, r11, r12 被篡改

\* 这个例程必须保护:

\* r4, r6, r7, r8

\*/

.align 5

cache\_clean\_flush:

mov r3, #16

b call\_cache\_fn

\_\_armv4\_mpu\_cache\_flush:

mov r2, #1

mov r3, #0

mcr p15, 0, ip, c7, c6, 0 @ invalidate D cache

mov r1, #7 << 5 @ 8 segments

1: orr r3, r1, #63 << 26 @ 64 entries

2: mcr p15, 0, r3, c7, c14, 2 @ clean & invalidate D index

subs r3, r3, #1 << 26

bcs 2b @ entries 63 to 0

subs r1, r1, #1 << 5

bcs 1b @ segments 7 to 0

teq r2, #0

mcrne p15, 0, ip, c7, c5, 0 @ invalidate I cache

mcr p15, 0, ip, c7, c10, 4 @ drain WB

mov pc, lr

\_\_fa526\_cache\_flush:

mov r1, #0

mcr p15, 0, r1, c7, c14, 0 @ clean and invalidate D cache

mcr p15, 0, r1, c7, c5, 0 @ flush I cache

mcr p15, 0, r1, c7, c10, 4 @ drain WB

mov pc, lr

\_\_armv6\_mmu\_cache\_flush:

mov r1, #0

mcr p15, 0, r1, c7, c14, 0 @ clean+invalidate D

mcr p15, 0, r1, c7, c5, 0 @ invalidate I+BTB

mcr p15, 0, r1, c7, c15, 0 @ clean+invalidate unified

mcr p15, 0, r1, c7, c10, 4 @ drain WB

mov pc, lr

\_\_armv7\_mmu\_cache\_flush:

mrc p15, 0, r10, c0, c1, 5 @ read ID\_MMFR1

tst r10, #0xf << 16 @ hierarchical cache (ARMv7)

mov r10, #0

beq hierarchical

mcr p15, 0, r10, c7, c14, 0 @ clean+invalidate D

b iflush

hierarchical:

mcr p15, 0, r10, c7, c10, 5 @ DMB

stmfd sp!, {r0-r7, r9-r11}

mrc p15, 1, r0, c0, c0, 1 @ read clidr

ands r3, r0, #0x7000000 @ extract loc from clidr

mov r3, r3, lsr #23 @ left align loc bit field

beq finished @ if loc is 0, then no need to clean

mov r10, #0 @ start clean at cache level 0

loop1:

add r2, r10, r10, lsr #1 @ work out 3x current cache level

mov r1, r0, lsr r2 @ extract cache type bits from clidr

and r1, r1, #7 @ mask of the bits for current cache only

cmp r1, #2 @ see what cache we have at this level

blt skip @ skip if no cache, or just i-cache

mcr p15, 2, r10, c0, c0, 0 @ select current cache level in cssr

mcr p15, 0, r10, c7, c5, 4 @ isb to sych the new cssr&csidr

mrc p15, 1, r1, c0, c0, 0 @ read the new csidr

and r2, r1, #7 @ extract the length of the cache lines

add r2, r2, #4 @ add 4 (line length offset)

ldr r4, =0x3ff

ands r4, r4, r1, lsr #3 @ find maximum number on the way size

clz r5, r4 @ find bit position of way size increment

ldr r7, =0x7fff

ands r7, r7, r1, lsr #13 @ extract max number of the index size

loop2:

mov r9, r4 @ create working copy of max way size

loop3:

ARM( orr r11, r10, r9, lsl r5 ) @ factor way and cache number into r11

ARM( orr r11, r11, r7, lsl r2 ) @ factor index number into r11

THUMB( lsl r6, r9, r5 )

THUMB( orr r11, r10, r6 ) @ factor way and cache number into r11

THUMB( lsl r6, r7, r2 )

THUMB( orr r11, r11, r6 ) @ factor index number into r11

mcr p15, 0, r11, c7, c14, 2 @ clean & invalidate by set/way

subs r9, r9, #1 @ decrement the way

bge loop3

subs r7, r7, #1 @ decrement the index

bge loop2

skip:

add r10, r10, #2 @ increment cache number

cmp r3, r10

bgt loop1

finished:

ldmfd sp!, {r0-r7, r9-r11}

mov r10, #0 @ swith back to cache level 0

mcr p15, 2, r10, c0, c0, 0 @ select current cache level in cssr

iflush:

mcr p15, 0, r10, c7, c10, 4 @ DSB

mcr p15, 0, r10, c7, c5, 0 @ invalidate I+BTB

mcr p15, 0, r10, c7, c10, 4 @ DSB

mcr p15, 0, r10, c7, c5, 4 @ ISB

mov pc, lr

\_\_armv5tej\_mmu\_cache\_flush:

1: mrc p15, 0, r15, c7, c14, 3 @ test,clean,invalidate D cache

bne 1b

mcr p15, 0, r0, c7, c5, 0 @ flush I cache

mcr p15, 0, r0, c7, c10, 4 @ drain WB

mov pc, lr

\_\_armv4\_mmu\_cache\_flush:

mov r2, #64\*1024 @ default: 32K dcache size (\*2)

mov r11, #32 @ default: 32 byte line size

mrc p15, 0, r3, c0, c0, 1 @ read cache type

teq r3, r9 @ cache ID register present?

beq no\_cache\_id

mov r1, r3, lsr #18

and r1, r1, #7

mov r2, #1024

mov r2, r2, lsl r1 @ base dcache size \*2

tst r3, #1 << 14 @ test M bit

addne r2, r2, r2, lsr #1 @ +1/2 size if M == 1

mov r3, r3, lsr #12

and r3, r3, #3

mov r11, #8

mov r11, r11, lsl r3 @ cache line size in bytes

no\_cache\_id:

mov r1, pc

bic r1, r1, #63 @ align to longest cache line

add r2, r1, r2

1:

ARM( ldr r3, [r1], r11 ) @ s/w flush D cache

THUMB( ldr r3, [r1] ) @ s/w flush D cache

THUMB( add r1, r1, r11 )

teq r1, r2

bne 1b

mcr p15, 0, r1, c7, c5, 0 @ flush I cache

mcr p15, 0, r1, c7, c6, 0 @ flush D cache

mcr p15, 0, r1, c7, c10, 4 @ drain WB

mov pc, lr

\_\_armv3\_mmu\_cache\_flush:

\_\_armv3\_mpu\_cache\_flush:

mov r1, #0

mcr p15, 0, r1, c7, c0, 0 @ invalidate whole cache v3

mov pc, lr

/\*

\* Various debugging routines for printing hex characters and

\* memory, which again must be relocatable.

\*/

#ifdef DEBUG

.align 2

.type phexbuf,#object

phexbuf: .space 12

.size phexbuf, . - phexbuf

@ phex corrupts {r0, r1, r2, r3}

phex: adr r3, phexbuf

mov r2, #0

strb r2, [r3, r1]

1: subs r1, r1, #1

movmi r0, r3

bmi puts

and r2, r0, #15

mov r0, r0, lsr #4

cmp r2, #10

addge r2, r2, #7

add r2, r2, #'0'

strb r2, [r3, r1]

b 1b

@ puts corrupts {r0, r1, r2, r3}

puts: loadsp r3, r1

1: ldrb r2, [r0], #1

teq r2, #0

moveq pc, lr

2: writeb r2, r3

mov r1, #0x00020000

3: subs r1, r1, #1

bne 3b

teq r2, #'\n'

moveq r2, #'\r'

beq 2b

teq r0, #0

bne 1b

mov pc, lr

@ putc corrupts {r0, r1, r2, r3}

putc:

mov r2, r0

mov r0, #0

loadsp r3, r1

b 2b

@ memdump corrupts {r0, r1, r2, r3, r10, r11, r12, lr}

memdump: mov r12, r0

mov r10, lr

mov r11, #0

2: mov r0, r11, lsl #2

add r0, r0, r12

mov r1, #8

bl phex

mov r0, #':'

bl putc

1: mov r0, #' '

bl putc

ldr r0, [r12, r11, lsl #2]

mov r1, #8

bl phex

and r0, r11, #7

teq r0, #3

moveq r0, #' '

bleq putc

and r0, r11, #7

add r11, r11, #1

teq r0, #7

bne 1b

mov r0, #'\n'

bl putc

cmp r11, #64

blt 2b

mov pc, r10

#endif

.ltorg

reloc\_code\_end:

.align

.section ".stack", "aw", %nobits

.L\_user\_stack: .space 4096

.L\_user\_stack\_end:

