ARM Linux 启动过程分析

作者：张俊岭
EMAIL: sprite_zjl@sina.com; jlzhang@tangrae.com.cn
QQ: 251450387
日期：2008-8-9
说明：
本文档基于 AT91SAM9260EK 板的，所用的 Linux 内核版本为 2.6.21

1 压缩与非压缩内核映象

非压缩内核映象是真正的 Linux 内核代码。**压缩内核映象**是把**非压缩内核映象**作为数据进行压缩打包，并加上了解压缩代码。也就是说，它是一个自解压的可执行映象。**压缩内核映象**执行时，先解压内部包含的数据块（即**非压缩内核映象**），再去执行**非压缩内核映象**。

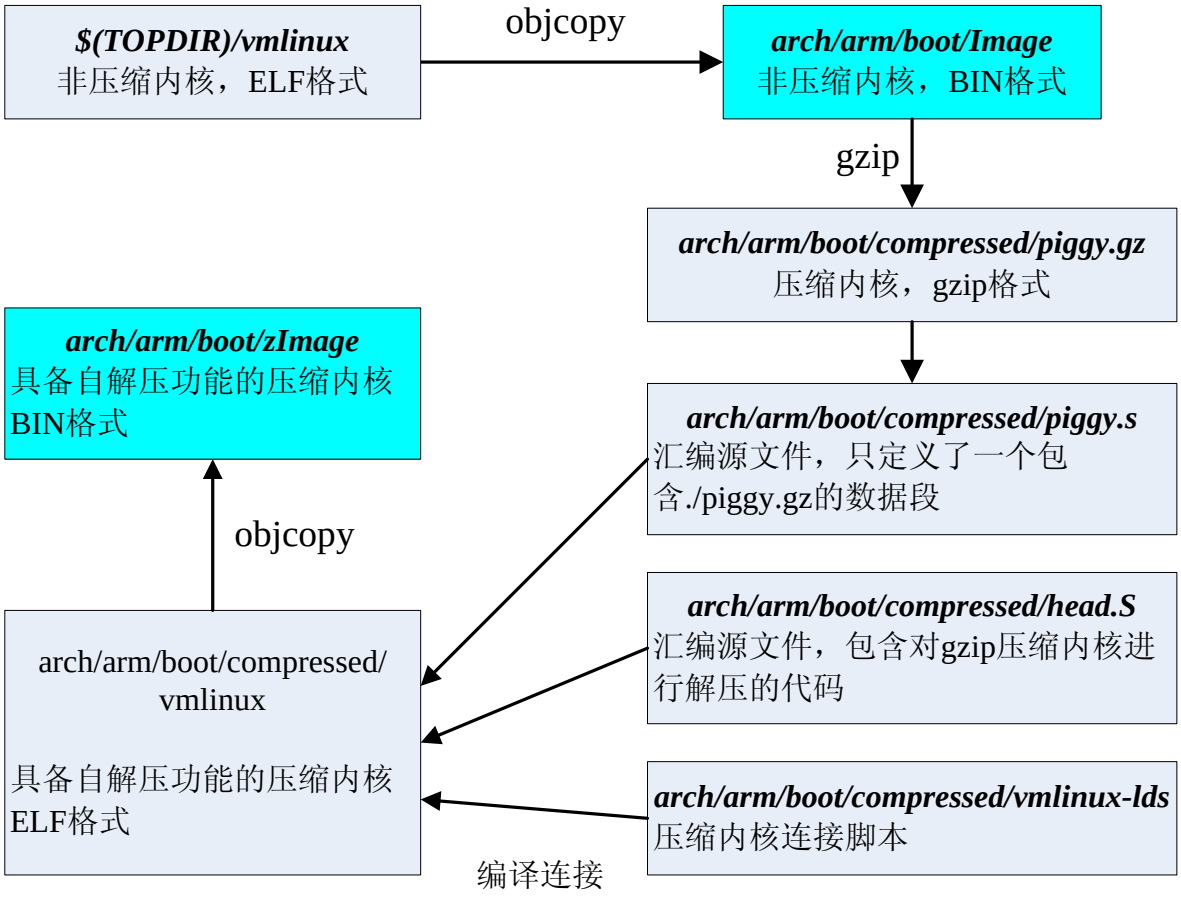
非压缩内核映象由 make Image 命令产生。其生成过程是：

(1) 内核的各个模块经过编译，链接，在内核源代码的顶层目录下生成 vmlinux 文件，这是一个 ELF 格式的映象
(2) 用 arm-linux-objcopy 命令把 vmlinux 转换为二进制格式映象 arch/arm/boot/Image

压缩内核映象由 make zImage 命令产生，其生成过程是：

(1) 用 gzip 对非压缩内核二进制映象 arch/arm/boot/ Image 进行压缩，生成 arch/arm/boot/compressed/piggy.gz 文件
(2) arch/arm/boot/compressed/目录下有三个文件：piggy.s，定义了一个包含./piggy.gz 文件的数据段；head.S 包含了对 gzip 压缩过的内核进行解压的代码；vmlinux-lds 是链接脚本。这几个文件经过编译链接，在 arch/arm/boot/compressed/目录下产生 vmlinux 文件，这是一个 ELF 格式的映象
(3) 用 arm-linux-objcopy 命令把 arch/arm/boot/compressed/vmlinux 转换为二进制格式映象：arch/arm/boot/compressed/zImage

两种内核映象的产生过程如下图：



2 内核入口

Linux 内核编译连接后生成的 ELF 映像文件是 vmlinux，从内核源代码顶层目录下的 Makefile(即顶层 Makefile)中可以找到 vmlinux 的生成规则：

vmlinux: $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) $(kallsyms.o) FORCE

其中$(vmlinux-lds)是编译连接脚本，对于 ARM 平台，就是 arch/arm/kernel/vmlinux-lds 文件。

vmlinux-init 也在顶层 Makefile 中定义：

vmlinux-init := $(head-y) $(init-y)

head-y 在 arch/arm/Makefile 中定义：

head-y:= arch/arm/kernel/head$(MMUEXT).o arch/arm/kernel/init_task.o

…

ifeq ($(CONFIG_MMU),)

MMUEXT := -nommu

endif

对于有 MMU 的处理器，MMUEXT 为空白字符串，所以 arch/arm/kernel/head.O 是第一个连接的文件，而这个文件是由 arch/arm/kernel/head.S 编译产生成的。

综合以上分析，可以得出结论，非压缩 ARM Linux 内核的入口点在 arch/arm/kernel/head.s 中。

3 汇编语言启动代码

先来看 arch/arm/kernel/head.S 的源代码（有精简）：

[arch/arm/kernel/head.S]

```
/* 定义内核的起始物理地址和起始虚拟地址  */
#define KERNEL_RAM_VADDR (PAGE_OFFSET + TEXT_OFFSET)
#define KERNEL_RAM_PADDR (PHYS_OFFSET + TEXT_OFFSET)

/* 定义初始页目录表的虚拟地址 swapper_pg_dir  */
#if (KERNEL_RAM_VADDR & 0xffff) != 0x8000
#error KERNEL_RAM_VADDR must start at 0xXXXX8000
#endif
    .globl    swapper_pg_dir
    .equ swapper_pg_dir, KERNEL_RAM_VADDR - 0x4000

    /* 宏定义：把初始页面目录表的物理地址装入 rd  */
    .macro    pgtbl, rd
    ldr   \rd, =(KERNEL_RAM_PADDR - 0x4000)
    .endm

    /* 内核启动入口点
    需要满足的条件（由 Bootloader 设置）：
    SVC 模式，关中断，MMU = off，D-cache = off，  R0=0，R1=机器类型代码
    */
    __INIT
    .type  stext, %function
ENTRY(stext)

    /* 设置处理器为 SVC 模式，关中断 */
    msr  cpsr_c, #PSR_F_BIT | PSR_I_BIT | SVC_MODE

    /*  CPUID => r9；调用__lookup_processor_type 查找处理器信息结构，
        proc_info 指针 => r10
     */
    mrc  p15, 0, r9, c0, c0          @ get processor id
    bl    __lookup_processor_type     @ r5=procinfo r9=cupid
    movs      r10, r5                 @ invalid processor (r5=0)?
```

```
    beq    __error_p                @ yes, error 'p'

     /*  调用__lookup_machine_type 查找机器类型信息结构,
         machine_desc 指针 => r8 */
    bl     __lookup_machine_type       @ r5=machinfo
    movs       r8, r5                  @ invalid machine (r5=0)?
    beq  __error_a              @ yes, error 'a'

     /*  调用__create_page_tables 为内核自身创建页表 */
    bl     __create_page_tables

    /*  调用__enable_mmu 函数后的返回地址 => r13  */
    ldr    r13, __switch_data        @ address to jump to after mmu has been enabled

   /*  __enable_mmu 函数的地址 => lr    */
    adr    lr, __enable_mmu          @ return (PIC) address

     /*  转移到 proc_info 结构中定义的处理器底层初始化函数，然后通过该函数的最后
         一条指令“mov pc，lr”跳转到__enalbe_mmu 函数
      */
    add    pc, r10, #PROCINFO_INITFUNC

/*
 * Setup common bits before finally enabling the MMU.    Essentially
 * this is just loading the page table pointer and domain access
 * registers.
 */
    /*  __enable_mmu 函数：打开 MMU    */
    .type  __enable_mmu, %function
__enable_mmu:
#ifdef CONFIG_ALIGNMENT_TRAP
    orr    r0, r0, #CR_A
#else
    bic    r0, r0, #CR_A
#endif
#ifdef CONFIG_CPU_DCACHE_DISABLE
    bic    r0, r0, #CR_C
#endif
#ifdef CONFIG_CPU_BPREDICT_DISABLE
    bic    r0, r0, #CR_Z
#endif
#ifdef CONFIG_CPU_ICACHE_DISABLE
    bic    r0, r0, #CR_I
#endif
```

```
        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      @ load domain access register
        mcr  p15, 0, r4, c2, c0, 0      @ load page table pointer
        b    __turn_mmu_on

        .align      5
        .type__turn_mmu_on, %function
__turn_mmu_on:
        mov r0, r0
        mcr  p15, 0, r0, c1, c0, 0      @ write control reg
        mrc  p15, 0, r3, c0, c0, 0      @ read id reg
        mov r3, r3
        mov r3, r3
        mov pc, r13

    /*  __create_page_tables 函数：为内核创建页表  */

    /*   r8   = machinfo     r9   = cupid     r10 = procinfo
         Returns:   r0, r3, r6, r7 corrupted    r4 = physical page table address
    */
        .type__create_page_tables, %function
__create_page_tables:
        pgtbl       r4                  @ page table address

    /*  清零页目录表  */
        mov r0, r4
        mov r3, #0
        add  r6, r0, #0x4000
1:      str   r3, [r0], #4
        str   r3, [r0], #4
        str   r3, [r0], #4
        str   r3, [r0], #4
        teq   r0, r6
        bne  1b

        ldr   r7, [r10, #PROCINFO_MM_MMUFLAGS]    @ mm_mmuflags

    /*  创建映射：VA= PA=物理内存起始地址，长度=1MB  */
        mov r6, pc, lsr #20             @ start of kernel section
        orr   r3, r7, r6, lsl #20       @ flags + kernel base
        str   r3, [r4, r6, lsl #2]      @ identity mapping
```

```
    /*  为内核的代码段和数据段创建映射：
        VA=0xC0008000  PA=物理内存起始地址+0x8000
        长度＝内核代码和数据的长度向上取整到 1MB 的整数倍
    */
     add  r0, r4,   #(KERNEL_START & 0xff000000) >> 18
     str   r3, [r0, #(KERNEL_START & 0x00f00000) >> 18]!
     ldr   r6, =(KERNEL_END - 1)
     add  r0, r0, #4
     add  r6, r4, r6, lsr #18
1:   cmp r0, r6
     add  r3, r3, #1 << 20
     strls r3, [r0], #4
     bls   1b

    /*  为物理内存的第 1 个 1MB 段（包含了 Bootloader 穿过来的启动参数）创建映射：
        VA=0xC0000000  PA=物理内存起始地址 长度＝1MB
    */
     add  r0, r4, #PAGE_OFFSET >> 18
     orr   r6, r7, #(PHYS_OFFSET & 0xff000000)
     orr   r6, r6, #(PHYS_OFFSET & 0x00e00000)
     str   r6, [r0]

    /*  如果定义了 CONFIG_DEBUG_LL，为 IO 设备创建内存映射，
        以便在页表系统初始化之前可以使用串行控制台
    */
#ifdef CONFIG_DEBUG_LL
     ldr   r7, [r10, #PROCINFO_IO_MMUFLAGS] @ io_mmuflags
     ldr   r3, [r8, #MACHINFO_PGOFFIO]
     add  r0, r4, r3
     rsb   r3, r3, #0x4000                  @ PTRS_PER_PGD*sizeof(long)
     cmp r3, #0x0800                        @ limit to 512MB
     movhi     r3, #0x0800
     add  r6, r0, r3
     ldr   r3, [r8, #MACHINFO_PHYSIO]
     orr   r3, r3, r7
1:   str   r3, [r0], #4
     add  r3, r3, #1 << 20
     teq   r0, r6
     bne  1b
#endif
     mov pc, lr
     .ltorg
```

```
#include "head-common.S"
```

文件的最后包含了 arch/arm/kernel/head-common.S，这个文件里定义了__mmap_switched, __lookup_processor_type 和, __lookup_machine_type 等几个函数，精简后的源代码如下：

[arch/arm/kernel/head-common.S]

```
/* __switch_data 结构  */
.type__switch_data, %object
__switch_data:
      .long     __mmap_switched
      .long     __data_loc              @ r4
      .long     __data_start            @ r5
      .long     __bss_start             @ r6
      .long     _end                    @ r7
      .long     processor_id            @ r4
      .long     __machine_arch_type         @ r5
      .long     cr_alignment            @ r6
      .long     init_thread_union + THREAD_START_SP @ sp

/* __mmap_switched 函数：重定位数据段，清零 BSS 数据段，调转到 C 语言入口函数
   start_kernel()
*/

.type__mmap_switched, %function
__mmap_switched:
      adr   r3, __switch_data + 4

      ldmia     r3!, {r4, r5, r6, r7}
      cmp r4, r5                        @ Copy data segment if needed
1:    cmpne     r5, r6
      ldrne     fp, [r4], #4
      strnefp, [r5], #4
      bne   1b

      mov fp, #0                        @ Clear BSS (and zero fp)
1:    cmp r6, r7
      strccfp, [r6],#4
      bcc   1b

      ldmia     r3, {r4, r5, r6, sp}
      str   r9, [r4]                @ Save processor ID
      str   r1, [r5]                @ Save machine type
      bic   r4, r0, #CR_A               @ Clear 'A' bit
```

```
        stmia       r6, {r0, r4}                    @ Save control register values
        b       start_kernel

 /* __error_p，__error_a 函数：异常处理，省略  */
 .type      __error_p, %function
__error_p:
…
        .type__error_a, %function
__error_a:
…

  /* __lookup_processor_type 函数：根据 CPUID 查找处理器信息结构  */
        .type__lookup_processor_type, %function
__lookup_processor_type:
        adr   r3, 3f                /*   标号 3 的物理地址  => r3   */


    /*   标号 3 的虚拟地址  => r7
         __proc_info_begin 的虚拟地址  => r5    __proc_info_end 的虚拟地址  => r6
    */
        ldmda       r3, {r5 - r7}

        sub   r3, r3, r7            /*   物理地址和虚拟地址之间的偏移量=> r3   */
        add   r5, r5, r3            /*   __proc_info_begin 的物理地址  => r5   */
        add   r6, r6, r3            /*   __proc_info_end 的物理地址  => r6   */

1:      ldmia       r5, {r3, r4}                    @ value, mask
        and   r4, r4, r9            @ mask wanted bits
        teq   r3, r4
        beq   2f
        add   r5, r5, #PROC_INFO_SZ             @ sizeof(proc_info_list)
        cmp r5, r6
        blo   1b
        mov r5, #0                      @ unknown processor
2:      mov pc, lr

ENTRY(lookup_processor_type)
        stmfd       sp!, {r4 - r7, r9, lr}
        mov r9, r0
        bl    __lookup_processor_type
        mov r0, r5
        ldmfd       sp!, {r4 - r7, r9, pc}

        .long       __proc_info_begin
```

```
        .long       __proc_info_end
3:      .long       .
        .long       __arch_info_begin
        .long       __arch_info_end

  /* __lookup_machine_type 函数：查找机器类型信息块  */
        .type__lookup_machine_type, %function
__lookup_machine_type:
        adr   r3, 3b
        ldmia       r3, {r4, r5, r6}
        sub   r3, r3, r4                @ get offset between virt&phys
        add   r5, r5, r3                @ convert virt addresses to
        add   r6, r6, r3                @ physical address space
1:      ldr   r3, [r5, #MACHINFO_TYPE]      @ get machine type
        teq   r3, r1                        @ matches loader number?
        beq   2f                        @ found
        add   r5, r5, #SIZEOF_MACHINE_DESC    @ next machine_desc
        cmp r5, r6
        blo   1b
        mov r5, #0                          @ unknown machine
2:      mov pc, lr

ENTRY(lookup_machine_type)
        stmfd       sp!, {r4 - r6, lr}
        mov r1, r0
        bl    __lookup_machine_type
        mov r0, r5
        ldmfd       sp!, {r4 - r6, pc}
```

通过对以上源代码的分析，总结出以下一些结论：

1) 重要常量

PAGE_OFFSET：内核空间起点，0xC0000000(3GB)
TEXT_OFFSET：内核代码段偏移量, 0x8000(32KB)
PHYS_OFFSET：物理内存起始地址，对于 AT91SAM9260EK，为 0x20000000
KERNEL_RAM_VADDR，内核起始虚拟地址 = 0xC0008000 (3GB+32KB)
KERNEL_RAM_PADDR，内核起始物理地址 = 0x20008000

从代码中还可以看出，Linux 内核的起始虚拟地址的低 16 位必须为 0x8000，
初始页目录表 swapper_pg_dir 的虚拟地址在 0xC0004000(3GB+16KB)。

2) 内核启动的执行流程

- 设置处理器为 SVC 模式，关中断
- 读取 CPUID，调用__lookup_processor_type 查找处理器信息结构 proc_info
- 调用__lookup_machine_type 查找机器类型信息结构 machine_desc
- 调用__create_page_tables 函数为内核创建页面映射表
- 调用处理器底层初始化函数，初始化 MMU，Cache，TLB
- 跳转到__enalbe_mmu 函数，打开 MMU
- 跳转到__mmap_switched 函数，建立 C 语言运行环境（搬移数据段，清理 BSS 段），保存 CPUID 和机器类型代码，最后跳转到 C 语言入口函数 start_kernel()

3) 处理器信息结构和机器类型信息结构的查找

所有 CPU 的信息结构都放在名为“.proc.info.init”的段中，连接后形成一个完整的段，段的起始地址为__proc_info_begin，结束地址为__proc_info_end。按关键字在__proc_info_begin 和 __proc_info_end 之间进行搜索就可以找到指定 CPU 的信息结构。CPU 信息结构的第一个数据成员就是关键字(CPUID)。搜索过程中要使用物理地址，因为此时 MMU 还没有打开，所以代码中有一个把__proc_info_begin 和__proc_info_end 从虚拟地址转换为物理地址的过程，处理得十分巧妙，详情可以参见源代码及其注释。

机器类型信息结构的查找也采用类似的方法。所有的机器类型信息结构都放在“.arch.info.init”段中，段的起始地址为__arch_info_begin，结束地址为__arch_info_end。搜索的关键字是机器类型代码，也保存在机器类型信息结构 machine_desc 的第一个数据成员中。

4) 存储映射

对于 AT91SAM9260EK，内核启动过程中通过__create_page_tables 函数创建了以下两个地址区间的映射：

序号	起始虚拟地址	起始物理地址	长度
1	0x20000000	0x20000000	1MB
2	0xC0000000	0x20000000	内核总长度向上调整到 1MB 的整数倍

其中第一个地址区间的映射是为了实现打开 MMU 前后地址空间的平稳过渡。

5) 处理器底层初始化函数

代码中通过以下指令转移到处理器底层初始化函数：

add pc, r10, #PROCINFO_INITFUNC

此时 r10 为 CPU 信息结构的指针，常量 PROCINFO_INITFUNC 定义为 16，程序调转到 CPU 信息结构块偏移 16 字节的地址，这里存放的是一条转向 CPU 底层初始化函数的跳转指令。对于 AT91SAM9260EK，CPU 信息结构和底层初始化函数都在 arch/arm/mm/proc-arm926.S 中：

[arch/arm/mm/proc-arm926.S]

```
/* 处理器底层初始化函数：__arm926_setup  */
.type__arm926_setup, #function
__arm926_setup:
      mov r0, #0
      mcr  p15, 0, r0, c7, c7          @ invalidate I,D caches on v4
      mcr  p15, 0, r0, c7, c10, 4      @ drain write buffer on v4
#ifdef CONFIG_MMU
      mcr  p15, 0, r0, c8, c7          @ invalidate I,D TLBs on v4
#endif

#ifdef CONFIG_CPU_DCACHE_WRITETHROUGH
      mov r0, #4                       @ disable write-back on caches explicitly
      mcr  p15, 7, r0, c15, c0, 0
#endif
      adr   r5, arm926_crval
      ldmia      r5, {r5, r6}
      mrc  p15, 0, r0, c1, c0          @ get control register v4
      bic   r0, r0, r5
      orr   r0, r0, r6
#ifdef CONFIG_CPU_CACHE_ROUND_ROBIN
      orr   r0, r0, #0x4000                  @ .1.. .... .... ....
#endif
      mov pc, lr
      .size __arm926_setup, . - __arm926_setup
…

/* 处理器信息结构  */
.align
      .section ".proc.info.init", #alloc, #execinstr
      .type__arm926_proc_info,#object
__arm926_proc_info:
      .long       0x41069260          @ ARM926EJ-S (v5TEJ)   /* CPUID  */
      .long       0xff0ffff0
      .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    __arm926_setup          /* 跳转指令，转向底层初始化函数   */
    .long    cpu_arch_name
    .long    cpu_elf_name
    .long
    HWCAP_SWP|HWCAP_HALF|HWCAP_THUMB|HWCAP_FAST_MULT|HWCAP_EDS
P|HWCAP_JAVA
    .long    cpu_arm926_name
    .long    arm926_processor_functions
    .long    v4wbi_tlb_fns
    .long    v4wb_user_fns
    .long    arm926_cache_fns
    .size __arm926_proc_info, . - __arm926_proc_info
```

可以看到，处理器底层初始化函数主要完成 MMU、Cache 和 TLB 的初始化，Write Buffer 回写策略的设置以及控制寄存器的读取等工作。

4 start_kernel()

start_kernel()函数是内核初始化 C 语言部分的主体。这个函数完成系统底层基本机制，包括处理器、存储管理系统、进程管理系统、中断机制、定时机制等的初始化工作。由于这个函数过于复杂，这里仅列出源代码，留待以后再按模块逐步深入分析。

[init/main.c: start_kernel()]

```
asmlinkage void __init start_kernel(void)
{
        char * command_line;
        extern struct kernel_param __start___param[], __stop___param[];

        smp_setup_processor_id();

        /*    Need to run as early as possible, to initialize the lockdep hash    */
        unwind_init();
        lockdep_init();

        local_irq_disable();
        early_boot_irqs_off();
        early_init_irq_lock_class();

 /* Interrupts are still disabled. Do necessary setups, then enable them    */
        lock_kernel();
        tick_init();
        boot_cpu_init();
        page_address_init();
        printk(KERN_NOTICE);
        printk(linux_banner);
        setup_arch(&command_line);   /*   架构相关的初始化，重点关注！    */

        setup_command_line(command_line);
        unwind_setup();
        setup_per_cpu_areas();
        smp_prepare_boot_cpu();      /* arch-specific boot-cpu hooks */

        /*
         * Set up the scheduler prior starting any interrupts (such as the
         * timer interrupt). Full topology setup happens at smp_init()
         * time - but meanwhile we still have a functioning scheduler.
         */
        sched_init();   /*   调度器初始化   */
        /*
```

```
 * Disable preemption - early bootup scheduling is extremely
 * fragile until we cpu_idle() for the first time.
 */
preempt_disable();
build_all_zonelists();
page_alloc_init();
printk(KERN_NOTICE "Kernel command line: %s\n", boot_command_line);
parse_early_param();
parse_args("Booting kernel", static_command_line, __start___param,
           __stop___param - __start___param,
           &unknown_bootoption);
if (!irqs_disabled()) {
      printk(KERN_WARNING "start_kernel(): bug: interrupts were "
                  "enabled *very* early, fixing it\n");
      local_irq_disable();
}
sort_main_extable();
trap_init();          /*   中断机制初始化   */
rcu_init();
init_IRQ();           /*   中断机制初始化   */
pidhash_init();       /*   PID Hash 机制初始化   */
init_timers();        /*   定时器初始化   */
hrtimers_init();
softirq_init();       /*   软中断初始化   */
timekeeping_init();
time_init();
profile_init();
if (!irqs_disabled())
      printk("start_kernel(): bug: interrupts were enabled early\n");
early_boot_irqs_on();
local_irq_enable();     /*   打开中断   */
/*
 * HACK ALERT! This is early. We're enabling the console before
 * we've done PCI setups etc, and console_init() must be aware of
 * this. But we do want output early, in case something goes wrong.
 */
console_init();     /*   控制台初始化   */
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 &&
                        initrd_start < min_low_pfn << PAGE_SHIFT) {
                printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
                        "disabling it.\n",initrd_start,min_low_pfn << PAGE_SHIFT);
                initrd_start = 0;
        }
#endif
        vfs_caches_init_early();
        cpuset_init_early();
        mem_init();
        kmem_cache_init();
        setup_per_cpu_pageset();
        numa_policy_init();
        if (late_time_init)
                late_time_init();
        calibrate_delay();
        pidmap_init();
        pgtable_cache_init();
        prio_tree_init();
        anon_vma_init();
#ifdef CONFIG_X86
        if (efi_enabled)
                efi_enter_virtual_mode();
#endif
        fork_init(num_physpages);
        proc_caches_init();
        buffer_init();
        unnamed_dev_init();
        key_init();
        security_init();
        vfs_caches_init(num_physpages);
        radix_tree_init();
        signals_init();
        /* rootfs populating might need page-writeback */
        page_writeback_init();
#ifdef CONFIG_PROC_FS
        proc_root_init();
#endif
```

```
    cpuset_init();
    taskstats_init_early();
    delayacct_init();

    check_bugs();

    acpi_early_init(); /* before LAPIC and SMP init */

    /* Do the rest non-__init'ed, we're now alive */
    rest_init();
}
```

函数完成基本的初始化工作后，最后调用了 rest_init()函数，这个函数的源代码如下：

[init/main.c：start_kernel()->rest_init()]

```
static void noinline rest_init(void)        __releases(kernel_lock)
{
    /* 创建内核线程，入口点是 init()函数 */
    kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND);
    numa_default_policy();
    unlock_kernel();

    preempt_enable_no_resched();
    schedule();
    preempt_disable();
    cpu_idle();      /* 禁止抢占，转入空闲 */

}
```

函数创建了一个入口点是 init()函数的内核线程，然后调用 cpu_idle()函数进入空闲状态。新创建的内核线程是系统的 1 号任务(pid =1)，放入了调度队列中，而原先的初始化代码是系统的 0 号任务，是不在调度队列中的。因此 1 号任务投入运行，系统转而执行 init()函数。这个函数同样定义在 init/main.c 中：

[init/main.c：init()]

```
static int __init init(void * unused)
{
    lock_kernel();

    set_cpus_allowed(current, CPU_MASK_ALL);

    init_pid_ns.child_reaper = current;
    cad_pid = task_pid(current);
    smp_prepare_cpus(max_cpus);
```

```
        do_pre_smp_initcalls();

        smp_init();
        sched_init_smp();
        cpuset_init_smp();
        do_basic_setup();   /* 重要函数，关注！*/

        if (!ramdisk_execute_command)
                ramdisk_execute_command = "/init";

        if (sys_access((const char __user *) ramdisk_execute_command, 0) != 0) {
                ramdisk_execute_command = NULL;
                prepare_namespace();
        }

        init_post();
        return 0;
}
```

init()函数接着完成系统更高层次，比如驱动程序，根文件系统等等的初始化工作。其中的do_basic_setup()函数比较重要，这个函数先调用 driver_init()函数完成驱动程序的初始化，又通过 do_initcalls()函数依次调用了系统中所有的初始化函数。do_initcalls()函数的主要源代码如下：

[init/main.c：init()->do_basic_setup()->do_initcalls()]

```
static void __init do_initcalls(void)
{
        initcall_t    *call;     int count = preempt_count();

 /* 循环调用__initcall_start 和__initcall_end 之间的所有函数指针指向的初始化函数   */
 for (call = __initcall_start; call < __initcall_end; call++) {
                …
                result = (*call)();    /* 调用 */
                …
        }
        …
}
```

符号__initcall_start 和__initcall_end 在链接脚本中 arch/arm/kernel/vmlinux.lds 定义：

```
__initcall_start = .;
    *(.initcall0.init)    *(.initcall0s.init)    *(.initcall1.init)    *(.initcall1s.init)    *(.initcall2.init)
*(.initcall2s.init)    *(.initcall3.init)    *(.initcall3s.init)    *(.initcall4.init)    *(.initcall4s.init)
*(.initcall5.init)    *(.initcall5s.init)    *(.initcallrootfs.init)    *(.initcall6.init)    *(.initcall6s.init)
```

```
*(.initcall7.init) *(.initcall7s.init)
  __initcall_end = .;
```

这两个符号之间包含了放在.initcallX.init，initcallXs.init（X=0~7）section 中的函数。这将按照顺序调用放置在这些 section 中的函数。

根据 include/linux/init.h 中的宏定义，可以整理出各种 initcall 宏和 section 的对应关系：

宏定义	Section
pure_initcall	.initcall0.init
core_initcall	.initcall1.init
core_initcall_sync	.initcall1s.init
postcore_initcall	.initcall2.init
postcore_initcall_sync	.initcall2s.init
arch_initcall	.initcall3.init
arch_initcall_sync	.initcall3s.init
subsys_initcall	.initcall4.init
subsys_initcall_sync	.initcall4s.init
fs_initcall	.initcall5.init
fs_initcall_sync	.initcall5s.init
device_initcall	.initcall6.init
device_initcall_sync	.initcall6s.init
late_initcall	.initcall7.init
late_initcall_sync	.initcall7s.init

在 init()函数的最后，调用了 init_post()函数，源代码如下：

[init/main.c：init()->init_post()]

```
static int noinline init_post(void)
{
      free_initmem();   /* 释放初始化内存 */
      unlock_kernel();
      mark_rodata_ro();
      system_state = SYSTEM_RUNNING;
      numa_default_policy();

    /* 打开控制台设备 handle=0   => stdin */
      if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
            printk(KERN_WARNING "Warning: unable to open an initial console.\n");

    /* 复制控制台设备到 handle 1,2 => stdout,stderr */
      (void) sys_dup(0);
      (void) sys_dup(0);

      /* 尝试执行 ramdisk_execute_command 指定的程序 */
```

```
    if (ramdisk_execute_command) {
            run_init_process(ramdisk_execute_command);
            printk(KERN_WARNING "Failed to execute %s\n",
                        ramdisk_execute_command);
     }
   /* 尝试执行 execute_command 指定的程序 */
     if (execute_command) {
            run_init_process(execute_command);
            printk(KERN_WARNING "Failed to execute %s.    Attempting "
                        "defaults...\n", execute_command);
     }

      /* 依次尝试执行四个外部程序  */
     run_init_process("/sbin/init");
     run_init_process("/etc/init");
     run_init_process("/bin/init");
     run_init_process("/bin/sh");

     panic("No init found.    Try passing init= option to kernel.");
}
```

函数打开了控制台设备/dev/console，并复制了两个 handle，这样 stdin,stdout,stderr 都指向/dev/console 设备。然后，函数依次尝试执行以下几个外部程序：

- 由 ramdisk_execute_command 指定的外部程序，即内核启动参数“rdinit=XXX”指定的程序
- 由 execute_command 指定的外部程序，即内核启动参数“init=XXX”指定的程序
- /sbin/init
- /etc/init
- /bin/init
- /bin/sh

这几个程序中任何一个加载执行成功，就进入了用户态，内核启动就宣告结束。

1 号任务原先是个内核线程，加载外部程序后就有了自己的用户态空间，成为一个进程，这就是系统中所有进程的祖先 1 号进程。然后 1 号进程再执行用户态的初始化程序，例如处理/etc/inittab，创建终端，等待用户登录等等，系统启动完成。