第 2 章 内核启动

内核代码就是一个程序，与用户程序一样，经过编译链接后，生成单一的可执行目标文件加载到内存中运行。内核目标文件与用户目标文件又有区别，内核目标文件在处理器内核态下运行，用户目标文件通常在处理器用户态下运行；内核执行最底层、最重要的软、硬件资源管理操作，并向用户程序提供操作内核资源的接口函数（系统调用）。另外，用户目标文件由内核加载到内存并运行，内核目标文件由系统引导加载程序加载到内存中并运行。

内核目标文件入口是一段体系结构相关的代码，而后是体系结构无关的启动函数。启动函数对内核各组件（子系统）进行初始化，加载外部根文件系统，最后运行第一个用户进程，系统启动完成。

2.1 运行内核

内核提供了非常好的配置、构建内核的机制，用户通过配置选择内核的功能、设置内核参数等。构建包括编译和链接，编译操作根据配置选项编译内核源文件，生成目标文件，链接操作将各目标文件链接成单一的可执行目标文件。内核配置、构建操作读者可先参考第 14 章内容。

系统上电后，处理器首先执行固件（通常是 ROM）中的引导加载程序（U-boot、BIOS 等），引导加载程序将存储在外部介质中的内核可执行目标文件加载到内存中，将命令行参数和环境变量复制到内存，并将地址传递给内核，最后使 CPU 程序指针跳转至内核代码入口地址，开始执行内核代码。

下面先简要介绍内核链接脚本，了解内核目标文件结构，入口地址等，然后简要介绍引导加载程序加载内核目标文件的过程。

2.1.1 链接脚本

内核代码链接脚本位于体系结构相关的目录下。MIPS 架构链接脚本为/arch/mips/kernel/vmlinux.lds.S，由文件内容我们可以看出内核可执行目标文件的布局，链接文件内容简列如下：

```
ENTRY(kernel_entry)     /*内核可执行目标文件的入口地址，位于/arch/mips/kernel/head.S 文件内*/
...
SECTIONS
{
    ...
    . = VMLINUX_LOAD_ADDRESS;      /*内核可执行目标文件起始地址（加载的虚拟地址）*/
                    /*等于 load-y 变量值（0xffffffff80100000），/arch/mips/loongson32/Platform*/
    _text = .;      /*代码段起始地址*/
    .text : {
        TEXT_TEXT     /*代码段，宏定义在/include/asm-generic/vmlinux.lds.h 头文件*/
        SCHED_TEXT     /*调度相关代码段*/
        LOCK_TEXT      /*自旋锁相关代码段*/
        KPROBES_TEXT
        IRQENTRY_TEXT
        *(.text.*)
        *(.fixup)
        *(.gnu.warning)
    } :text = 0
```

```
_etext = .;       /*代码段结束地址*/

EXCEPTION_TABLE(16)         /*异常向量表*/

/* Exception table for data bus errors */
__dbe_table : {
        __start___dbe_table = .;
        *(__dbe_table)
        __stop___dbe_table = .;
}

NOTES :text :note
.dummy : { *(.dummy) } :text

_sdata = .;                     /*数据段开始地址*/
RODATA              /*只读数据段*/

/*可写数据段*/
.data : {    /* Data */
        . = . + DATAOFFSET;          /* for CONFIG_MAPPED_KERNEL */

        INIT_TASK_DATA(THREAD_SIZE)      /*init_task 段*/
        NOSAVE_DATA
        CACHELINE_ALIGNED_DATA(1 << CONFIG_MIPS_L1_CACHE_SHIFT)
        READ_MOSTLY_DATA(1 << CONFIG_MIPS_L1_CACHE_SHIFT)
        DATA_DATA
        CONSTRUCTORS
}
_gp = . + 0x8000;
...
.sdata : {
        *(.sdata)
}
_edata =    .;                  /*数据段结束地址*/

/*初始化段起始地址，内核启动后释放*/
. = ALIGN(PAGE_SIZE);
__init_begin = .;               /*初始化段包含只在内核初始化时调用/使用的函数/数据*/
INIT_TEXT_SECTION(PAGE_SIZE)
INIT_DATA_SECTION(16)

. = ALIGN(4);
.mips.machines.init : AT(ADDR(.mips.machines.init) - LOAD_OFFSET) {
```

```
        __mips_machines_start = .;
        *(.mips.machines.init)
        __mips_machines_end = .;
    }

    .exit.text : {
        EXIT_TEXT
    }
    .exit.data : {
        EXIT_DATA
    }
#ifdef CONFIG_SMP
    PERCPU_SECTION(1 << CONFIG_MIPS_L1_CACHE_SHIFT)     /*静态定义 percpu 变量段*/
#endif
#ifdef CONFIG_MIPS_RAW_APPENDED_DTB     /*设备树目标文件段*/
    __appended_dtb = .;         /*设备树目标文件起始地址*/
    . += 0x100000;
#endif
    . = ALIGN(0x10000);
    __init_end = .;         /*初始化段结束地址*/

    BSS_SECTION(0, 0x10000, 0)       /*未初始化数据段*/

    _end = . ;     /*内核镜像结束地址*/
    ...
}
```

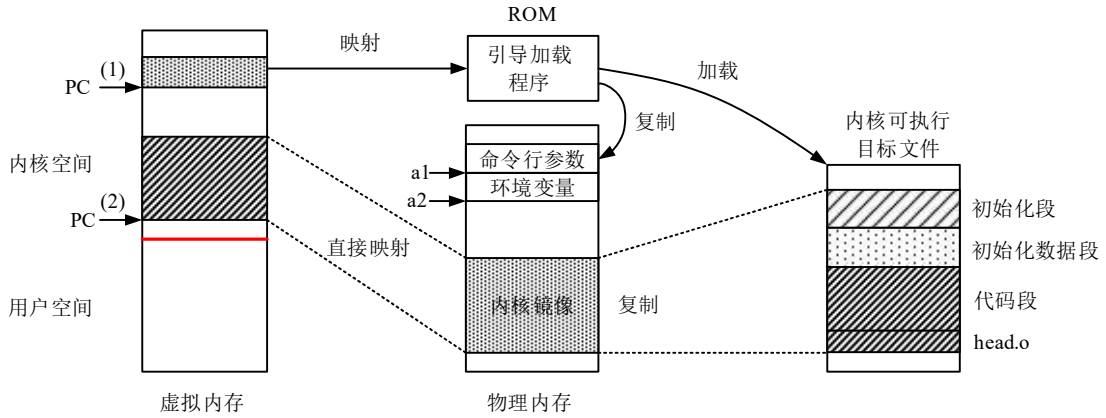
链接脚本中各大段的起始和结束地址标号将导出到内核代码，在内核代码中可以使这些段的起始结束地址。各大段内又划分成许多的小段，在链接脚本中由定义在/include/asm-generic/vmlinux.lds.h 头文件的宏表示。表示各小段的段名的宏定义在/include/linux/init.h 头文件，读者在阅读内核源代码时要特别注意显式指明了链接到哪个段的函数和变量。

2.1.2 加载运行

处理器上电后，从固定的地址开始取指执行，而这个固定的地址通常映射到 ROM 等非易失性存储介质。在这些存储介质中保存着引导加载程序，如 U-boot、BIOS 等。引导加载程序可理解成一个小型的操作系统，它负责检测并初始化硬件，从指定地址加载内核可执行目标文件至内存，向内核传递命令行参数和环境变量等，最后将内核代码入口地址赋予处理器 PC 指针，处理器开始运行内核代码。

用户可以设置引导加载程序参数，通过它设置内核目标文件路径，向内核传递命令行参数等。

系统通过引导加载程序加载内核可执行目标文件流程如下图所示：



由前面介绍的链接脚本可知，内核可执行目标文件中主要包含存储内核代码（指令）的代码段、已初始化的数据段、初始化段和未初始化数据段（不占目标文件空间）等，这些段中的数据将由引导加载程序加载进物理内存（未初始化数据段只需要预留空间）。加载后，内存中数据与目标文件中的数据是映射关系（内容一样），称之为内核镜像。

引导加载程序还会将其中设置的命令行参数和环境变量复制到物理内存，并将其地址保存到通用寄存器中以传递给内核，最后将内核入口地址加载到处理器程序指针（PC 寄存器），处理器开始执行内核代码。

内核可执行目标文件代码段的开头链接的是 head.S 文件中的代码（体系结构实现的文件），这是内核运行的起点，详见下一节。初始化段中保存的是只在内核启动阶段调用的函数和访问的数据，在内核代码中通常用“__init*”修饰，内核启动的末期将释放初始化段占用的内存（无效段内函数和数据，内存可被内核再次使用）。初始化段中主要链接的是内核初始化函数、设备驱动初始化函数、设备树目标文件、初始文件系统内容（基于物理内存的文件系统）等。

2.2 内核起点

由内核链接脚本可知，内核可执行目标文件的入口地址为 **kernel_entry**，这个地址定义在体系结构相关的 head.S 文件内，这是内核运行的起点。由构建文件内容可知，head.S 文件也是第一个被链接的文件，其代码位于可执行目标文件开头。

对于 MIPS 体系结构 head.S 文件路径为/arch/mips/kernel/head.S，下面将简要介绍此文件内容。

我们先来看一下 head.S 文件内定义的几个宏：

```
.macro setup_c0_status set clr      /*设置 CP0_STATUS 寄存器*/
.set   push
mfc0       t0, CP0_STATUS           /*读取状态寄存器值*/
or     t0, ST0_CU0|\set|0x1f|\clr   /*置相应位*/
xor    t0, 0x1f|\clr                /*清零低 5 位和 clr 标记的位*/
mtc0   t0, CP0_STATUS               /*写状态寄存器*/
.set   noreorder
sll    zero,3
.set   pop
.endm
```

setup_c0_status 宏的作用是清零处理器状态寄存器低 5 位（内核态、关中断）和 clr 参数中标记的位，置位标记位 ST0_CU0（使能协处理器 0）和 set 参数标记的位。

```
        .macro    setup_c0_status_pri
    #ifdef CONFIG_64BIT
        setup_c0_status   ST0_KX   0
    #else
        setup_c0_status 0 0     /*设置状态寄存器，设置处理器内核态、关中断*/
    #endif
        .endm
```

MIPS 体系结构 head.S 文件中代码完成的主要工作如下：

（1）设置 CP0 协处理器状态寄存器值，使处理器处于内核态，关中断。

（2）查找链接到内核可执行目标文件的设备树目标文件，如果存在则将其基址写入 a1 寄存器。

（3）清零内核未初始化数据段。

（4）将引导加载程序通过 a0,a1,a2,a3 寄存器传递的参数写入全局变量。

（5）设置内核栈。

（6）跳转至 start_kernel()函数运行，执行体系结构无关的内核启动函数。

下面列出 head.S 文件内主要的程序代码：

```
    #ifndef   CONFIG_NO_EXCEPT_FILL       /*处理器配置选项，一般没有选择*/
        .fill  0x400                      /*预留空间用于异常向量*/
    #endif

    EXPORT(_stext)        /*导出标号*/
    ...
    NESTED(kernel_entry, 16, sp)  /*定义 kernel_entry 函数，内核入口地址*/

        kernel_entry_setup    /*默认实现为空，由平台（处理器）相关代码实现*/

        setup_c0_status_pri      /*设置处理器状态寄存器，内核态，关中断*/

        PTR_LA   t0, 0f        /*加载地址至 t0，PTR_LA：la*/
        jr       t0          /*跳转到前面 0 标号处运行*/
0:
    #ifdef CONFIG_MIPS_RAW_APPENDED_DTB     /*内核可执行目标文件中链接了设备树目标文件*/
        PTR_LA t0, __appended_dtb     /*设备树目标文件起始地址，/arch/mips/kernel/vmlinux.lds.S*/

      #ifdef CONFIG_CPU_BIG_ENDIAN
        li     t1, 0xd00dfeed
      #else
        li     t1, 0xedfe0dd0     /*设备树目标文件魔数*/
      #endif
        lw    t2, (t0)       /*设备树目标文件魔数，在文件开头处*/
        bne   t1, t2, not_found     /*魔数比对不成功表示不是设备树文件，跳转至 not_found*/
```

```
        nop

        move    a1, t0    /*设备树目标文件比对成功，a1 保存设备树目标文件起始地址*/
        PTR_LI  a0, -2    /*a0=-2，PTR_LI=li*/
not_found:
    #endif      /*CONFIG_MIPS_RAW_APPENDED_DTB 结束*/

        PTR_LA  t0, __bss_start         /*未初始化数据段起始地址，清零未初始化数据段*/
        LONG_S  zero, (t0)              /*LONG_S：sw*/
        PTR_LA  t1, __bss_stop - LONGSIZE
                                        /*LONGSIZE 表示整型数所含的字节数*/
    1:
        PTR_ADDIU  t0, LONGSIZE      /*PTR_ADDIU：addiu*/
        LONG_S zero, (t0)
        bne     t0, t1, 1b              /*未初始化数据段清零完成*/

        /*
        *将引导加载程序传递的参数写入内核全局变量，
        *全局变量 fw_arg0，fw_arg1,fw_arg2,fw_arg3 定义在/arch/mips/kernel/setup.c 文件内。
        */
        LONG_S      a0, fw_arg0      /*命令行参数数量，或为-2（传递了设备树目标文件）*/
        LONG_S      a1, fw_arg1      /*命令行参数字符串指针数组基地址，或设备树目标文件地址*/
        LONG_S      a2, fw_arg2      /*环境变量指针数组基地址（不是用于用户进程的环境变量）*/
        LONG_S      a3, fw_arg3      /**/

        MTC0        zero, CP0_CONTEXT       /*清零处理器 context 寄存器*/
        PTR_LA      $28, init_thread_union      /*$28 寄存器保存内核自身 thread_union 实例地址*/

        PTR_LI      sp, _THREAD_SIZE - 32 - PT_SIZE
        PTR_ADDU    sp, $28         /*内核栈顶地址，pt_regs 实例基址*/
        back_to_back_c0_hazard
        set_saved_sp  sp, t0, t1      /*将内核栈地址存入全局 kernelsp[cpu]数组项*/
        PTR_SUBU    sp, 4 * SZREG       /*SZREG=4，以上代码用于设置内核栈 sp 值*/

        j       start_kernel      /*跳转到内核启动函数 start_kernel()*/
      END(kernel_entry)       /*kernel_entry 函数结束*/
```

head.S 文件内获取引导加载程序传递的设备树目标文件地址、命令行参数指针数组、环境变量指针数组，清零未初始化数据段，最后调用体系结构无关的内核启动函数 **start_kernel()**。

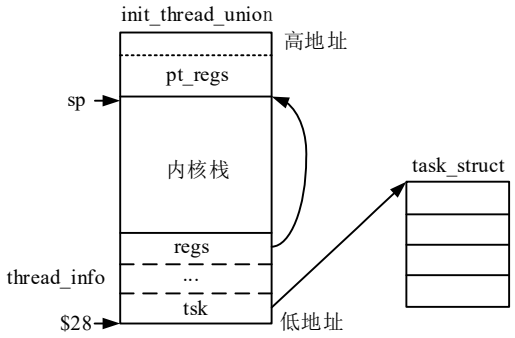
head.S 文件内还有一项工作就是将$28 通用寄存器设置为内核 thread_union 实例地址，并设置内核栈。

内核中每一个进程/线程，包含内核自身（可认为是一个内核线程），内核为其创建一个 thread_union 联合体实例。联合体定义如下（/include/linux/sched.h）：

union thread_union {

```
    struct thread_info thread_info;          /*thread_info 结构体实例*/
    unsigned long stack[THREAD_SIZE/sizeof(long)];    /*进程/线程的内核栈，通常为 8KB*/
};
```

thread_union 联合体底部是一个 thread_info 结构体实例，表示进程的底层信息，其中包括进程 task_struct 结构体实例的指针，如下图所示。联合体大小由 stack[]数组大小决定，通常为 8KB，用于进程/线程在内核态运行时的栈。内核栈顶部是一个 pt_regs 结构体实例，用于保存进程由用户空间进入内核空间运行时的上下文信息，第 5 章。



内核自身 thread_union 联合体实例 **init_thread_union** 定义在/init/init_task.c 文件内，在以上代码中，寄存器**$28** 存入 init_thread_union 实例基地址。栈顶预留 pt_regs 结构体实例再加上 32 字节空间，然后再往下预留 16 字节，最终地址作为内核栈顶地址赋予 sp 寄存器。

2.3 启动函数

在体系结构相关的 head.S 文件中最后调用体系结构无关的启动函数 **start_kernel()**启动内核。启动函数 start_kernel()内完成引导加载程序传递的命令行参数的处理，内核各子系统和设备驱动的初始化，创建内核线程，挂载根文件系统等工作，最后运行第一个用户进程，内核启动完成。

start_kernel()函数在/init/main.c 文件内实现，代码如下：

```
asmlinkage __visible void __init start_kernel(void)
{
    char *command_line;        /*命令行参数指针，setup_arch()函数中使用*/
    char *after_dashes;

    lockdep_init();
    set_task_stack_end_magic(&init_task);
            /*在内核栈最低位置保存魔数 STACK_END_MAGIC，/kernel/fork.c*/
    smp_setup_processor_id();    /*体系结构代码实现，默认为空操作，/init/main.c*/
    debug_objects_early_init();

    boot_init_stack_canary();

    cgroup_init_early();        /*控制组早期初始化*/

    local_irq_disable();       /*关闭本地中断*/
    early_boot_irqs_disabled = true;
```

```
boot_cpu_init();            /*设置启动 CPU 核在 CPU 位图中标记位，/init/main.c*/
page_address_init();        /*初始化内核持久映射区管理数据结构，/mm/highmem.c*/
pr_notice("%s", linux_banner);
setup_arch(&command_line);      /*体系结构相关启动函数，/arch/mips/kernel/setup.c*/
mm_init_cpumask(&init_mm);    /*清 mm->cpu_vm_mask_var 位图，/include/linux/mm_types.h*/
setup_command_line(command_line);  /*复制命令行参数，/init/main.c*/
setup_nr_cpu_ids();         /*设置 nr_cpu_ids 变量，表示 SMP 处理器实际的核数量，/kernel/smp.c*/
setup_per_cpu_areas();          /*处理静态定义的 percpu 变量，/mm/percpu.c*/
smp_prepare_boot_cpu();       /*设置 CPU0 在 CPU 位图中标记位，/arch/mips/kernel/smp.c*/

build_all_zonelists(NULL, NULL);   /*借用内存列表初始化，/mm/page_alloc.c*/
page_alloc_init();                  /*mm/page_alloc.c*/

pr_notice("Kernel command line: %s\n", boot_command_line);
parse_early_param();          /*处理由 early_param(str, fn)宏声明处理函数的命令行参数*/

after_dashes = parse_args("Booting kernel", static_command_line, __start___param,
                __stop___param - __start___param,-1, -1, NULL, &unknown_bootoption);
                        /*处理需后期处理及内核未定义的命令行参数，__setup(str, fn)*/
if (!IS_ERR_OR_NULL(after_dashes))
    parse_args("Setting init args", after_dashes, NULL, 0, -1, -1,NULL, set_init_arg);
                                /*处理“--”参数之后的命令行参数*/

jump_label_init();

setup_log_buf(0);       /*系统控制台相关的初始化，/kernel/printk/printk.c*/
pidhash_init();         /*upid 实例全局散列表创建及初始化，/kernel/pid.c*/
vfs_caches_init_early();    /*虚拟文件系统早期初始化，/fs/dcache.c*/
sort_main_extable();
trap_init();    /*初始化异常（中断）向量（中断处理），见第 6 章，/arch/mips/kernel/traps.c*/
mm_init();  /*停用自举分配器，启用伙伴系统，初始化 slab 分配器等，见第 3 章，/init/main.c*/

sched_init();           /*进程调度器初始化，见第 5 章，/kernel/sched/core.c*/

preempt_disable();      /*禁止内核抢占*/
if (WARN(!irqs_disabled(),"Interrupts were enabled *very* early, fixing it\n"))
local_irq_disable();    /*关闭本地 CPU 核中断*/
idr_init_cache();       /*创建 idr 数据结构 idr_layer 结构体缓存，/lib/idr.c*/
rcu_init();    /*RCU 机制数据结构初始化，见第 6 章，/kernel/rcu/tree.c*/

trace_init();

context_tracking_init();
```

```
    radix_tree_init();      /*基数树初始化，创建数据结构缓存等，/lib/radix_tree.c*/

    early_irq_init();       /*初始化中断描述符 irq_desc[]数组成员，见第 6 章，/kernel/irq/irqdesc.c*/
    init_IRQ();             /*完成 irq_desc[]数组平台相关的初始化，见第 6 章，/arch/mips/kernel/irq.c*/
    tick_init();            /*广播及动态时钟初始化，见第 6 章，/kernel/time/tick-common.c*/
    rcu_init_nohz();        /*/kernel/rcu/tree_plugin.h*/
    init_timers();          /*低分辨率定时器初始化，见第 6 章，/kernel/time/timer.c*/
    hrtimers_init();        /*高分辨率定时器初始化，见第 6 章，/kernel/time/hrtimer.c*/
    softirq_init();         /*软中断机制初始化，见第 6 章，/kernel/softirq.c*/
    timekeeping_init();     /*记时器初始化，见第 6 章，/kernel/time/timekeeping.c*/
    time_init();            /*注册时钟源、时钟事件设备，见第 6 章，/arch/mips/kernel/time.c*/
    sched_clock_postinit();     /*调度时钟初始化*/
                        /*需选择 GENERIC_SCHED_CLOCK 选项，/kernel/time/sched_clock.c*/
    perf_event_init();
    profile_init();             /*/kernel/profile.c*/
    call_function_init();   /*向 CPU 通知链注册通知，/kernel/smp.c*/
    WARN(!irqs_disabled(), "Interrupts were enabled early\n");
    early_boot_irqs_disabled = false;
    local_irq_enable();     /*打开本地 CPU 核中断*/

    kmem_cache_init_late();     /*slab 分配器初始化（后期），/mm/slab.c*/

    console_init();         /*系统控制台初始化，/drivers/tty/tty_io.c*/
    if (panic_later)
        panic("Too many boot %s vars at `%s'", panic_later,panic_param);

    lockdep_info();

    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_ext_init();        /*需选择 SPARSEMEM（稀疏内存）配置选项，否则为空操作*/
    debug_objects_mem_init();
    kmemleak_init();
    setup_per_cpu_pageset();    /*为各内存域创建 per_cpu_pageset 实例并初始化，/mm/page_alloc.c*/
```

```
    numa_policy_init();
    if (late_time_init)
        late_time_init();        /*空操作，/init/main.c*/
    sched_clock_init();          /*设置 sched_clock_running = 1，/kernel/sched/clock.c*/
    calibrate_delay();
    pidmap_init();               /*初始 PID 命名空间初始化，分配位图，创建 pid 缓存等，/kernel/pid.c*/
    anon_vma_init();             /*初始化匿名映射结构（虚拟内存），/mm/rmap.c*/
    acpi_early_init();
     ...
    thread_info_cache_init();
                    /*thread_union 大于一页为空操作，否则为 thread_union 创建 slab 缓存，/kernel/fork.c*/
    cred_init();                  /*创建 cred 结构体（用户信息）slab 缓存，/kernel/cred.c*/
    fork_init();                  /*创建 task_struct 结构体 slab 缓存等，/kernel/sched/fork.c*/
    proc_caches_init();           /*创建信号、文件、内存域等结构体 slab 缓存，/kernel/fork.c*/
    buffer_init();       /*块缓存初始化，为 buffer_head 结构创建 slab 缓存，/fs/buffer.c*/
    key_init();              /*密钥初始化*/
    security_init();          /*安全子系统初始化*/
    dbg_late_init();
    vfs_caches_init();       /*虚拟文件系统早期初始化，/fs/dcache.c*/
    signals_init();           /*为 sigqueue 结构体创建 slab 缓存，/kernel/signal.c*/
    page_writeback_init();     /*设置数据回写脏页限制值初始值，/mm/page-writeback.c*/
    proc_root_init();            /*proc 文件系统初始化，/fs/proc/root.c*/
    nsfs_init();
    cpuset_init();            /*/kernel/cpuset.c*/
    cgroup_init();         /*控制组初始化，/kernel/cgroup.c*/
    taskstats_init_early();
    delayacct_init();
    check_bugs();
    acpi_subsystem_init();
    sfi_init_late();

    if (efi_enabled(EFI_RUNTIME_SERVICES)) {
        efi_late_init();
        efi_free_boot_services();
    }
    ftrace_init();
    rest_init();         /*完成剩余初始化工作，/init/main.c*/
}
```

start_kernel()函数内主要是处理引导加载程序传递的命令行参数，执行体系结构定义的启动函数，调用各子系统（组件）定义的初始化函数完成初始化工作，具体函数将在后面的章节中详细介绍。本章主要介绍体系结构实现的启动函数 setup_arch()、命令行参数的处理以及 rest_init()函数的实现。

在 rest_init()函数中将继续完成内核初始化工作、创建内核线程、挂载根文件系统、运行第一个用户进程，完成系统的启动。rest_init()函数，即内核自身，最后进入一个无限循环，成为系统中的空闲进程。也

就是说内核执行程序完成初始化工作和系统启动后，转化成空闲进程，系统工作将由用户进程和内核线程完成。内核随后以系统调用（类似于库函数）的形式向用户进程提供服务。

2.4 setup_arch()

在启动函数 start_kernel()中将调用 **setup_arch()**函数，完成由体系结构定义的启动操作。MIPS 体系结构 setup_arch()函数在/arch/mips/kernel/setup.c 文件内实现，代码如下：

```
void __init setup_arch(char **cmdline_p)
/*cmdline_p：start_kernel()中局部变量 cmdline_p 的指针*/
{
    cpu_probe();   /*探测 CPU 核信息，/arch/mips/kernel/cpu-probe.c*/
    prom_init();
        /*平台代码实现，主要完成命令行参数复制等，/arch/mips/loongson32/common/prom.c*/
    setup_early_fdc_console();      /*没选择 MIPS_EJTAG_FDC_EARLYCON 配置选项时，为空操作*/

#ifdef CONFIG_EARLY_PRINTK
    setup_early_printk();
#endif
    cpu_report();                   /*主要用于 CPU 信息的输出，/arch/mips/kernel/cpu_probe.c*/
    check_bugs_early();             /*选择 64BIT 配置选项时才有定义，arch/mips/include/asm/bugs.h*/

#if defined(CONFIG_VT)
  #if defined(CONFIG_VGA_CONSOLE)
    conswitchp = &vga_con;
  #elif defined(CONFIG_DUMMY_CONSOLE)
    conswitchp = &dummy_con;
  #endif
#endif

    arch_mem_init(cmdline_p);      /*物理内存管理初始化，见第 3 章，/arch/mips/kernel/setup.c*/

    resource_init();               /*资源管理初始化，见第 8 章，/arch/mips/kernel/setup.c*/
    plat_smp_setup();              /*多核处理器初始化，见第 5 章，/arch/mips/include/asm/smp-op.h*/

    prefill_possible_map();  /*设置 possible 处理器核位图（nr_cpu_ids），/arch/mips/kernel/setup.c*/

    cpu_cache_init();              /*CPU 核缓存初始化，设置操作接口函数，/arch/mips/mm/cache.c*/
}
```

setup_arch()函数中调用的都是体系结构相关的初始化函数，这里我们先介绍与平台（处理器）密切相关的 cpu_probe()、prom_init()和 cpu_cache_init()函数的实现，其它子系统（组件）初始化函数将在后续章节中详细介绍。

cpu_probe()函数由体系结构代码实现，用于获取 MIPS 处理器的硬件信息，并保存至 cpuinfo_mips 结构体实例中，供后续代码使用。

prom_init()函数由板级（处理器）代码实现，主要用于获取引导加载程序（固件）传递的参数和初始

化必要的硬件设备。

cpu_cache_init()函数由体系结构代码实现，主要完成处理器缓存（cache）的初始化以及全局的缓存操作接口函数指针的设置，内核代码中可直接调用这些接口函数对处理器缓存进行操作。

2.4.1 探测处理器信息

MIPS 体系结构代码中定义了 cpuinfo_mips 结构体（/arch/mips/include/asm/cpu-info.h）用于保存处理器核的硬件信息，内核为处理器中每个 CPU 核创建了 cpuinfo_mips 结构体实例。

cpuinfo_mips 结构体定义如下：

```
struct cpuinfo_mips {
    unsigned long       asid_cache;    /*当前使用用户地址空间 ASID 值，进程切换时会用到*/
    unsigned long       ases;
    unsigned long long  options;    /*标记 CPU 核信息，如中断类型、缓存类型等*/
    unsigned int        udelay_val;
    unsigned int        processor_id;   /*PRID 寄存器值，处理器核编号，/arch/mips/include/asm/cpu.h*/
    ...
    unsigned int        cputype;        /*CPU 类型，/arch/mips/include/asm/cpu.h*/
    int         isa_level;
    int         tlbsize;            /*TLB 信息*/
    int         tlbsizevtlb;
    int         tlbsizeftlbsets;
    int         tlbsizeftlbways;
    struct  cache_desc  icache;     /*指令缓存信息*/
    struct  cache_desc  dcache;     /*数据缓存信息*/
    struct  cache_desc  scache;     /*二级缓存信息*/
    struct  cache_desc  tcache;     /*三级缓存信息*/
    int         srsets;         /*影子寄存器组数*/
    int         package;        /* physical package number */
    int         core;           /*CPU 核数量*/
  ...
} __attribute__((aligned(SMP_CACHE_BYTES)));
```

cpuinfo_mips 结构体中主要成员简介如下：

●**options**：每个比特位表示 CPU 核的某一特性，各位含义定义在/arch/mips/include/asm/cpu.h 头文件内：

```
#define   MIPS_CPU_TLB          0x00000001ull   /*具有 TLB */
#define   MIPS_CPU_4KEX         0x00000002ull   /* "R4K"异常向量模式*/
#define   MIPS_CPU_3K_CACHE     0x00000004ull   /* R3000 类型缓存*/
#define   MIPS_CPU_4K_CACHE     0x00000008ull   /* R4000 类型缓存*/
#define   MIPS_CPU_TX39_CACHE   0x00000010ull /* TX3900-style caches，缓存特性*/
#define   MIPS_CPU_FPU          0x00000020ull   /* CPU 具有 FPU，浮点协处理器  */
#define   MIPS_CPU_32FPR            0x00000040ull   /* 2 位浮点寄存器*/
#define   MIPS_CPU_COUNTER          0x00000080ull   /*具有 count/compare  寄存器*/
#define   MIPS_CPU_WATCH            0x00000100ull   /*具有 watchpoint 寄存器  */
#define   MIPS_CPU_DIVEC            0x00000200ull   /* dedicated interrupt vector */
```

```
#define   MIPS_CPU_VCE          0x00000400ull             /* virt. coherence conflict possible */
#define   MIPS_CPU_CACHE_CDEX_P   0x00000800ull   /* Create_Dirty_Exclusive CACHE op */
#define   MIPS_CPU_CACHE_CDEX_S   0x00001000ull   /* ... same for seconary cache ... */
#define   MIPS_CPU_MCHECK         0x00002000ull         /* Machine check exception */
#define   MIPS_CPU_EJTAG          0x00004000ull         /* EJTAG exception */
#define   MIPS_CPU_NOFPUEX        0x00008000ull         /* no FPU exception */
#define   MIPS_CPU_LLSC           0x00010000ull         /* CPU has ll/sc instructions */
#define   MIPS_CPU_INCLUSIVE_CACHES   0x00020000ull       /* P-cache subset enforced */
#define   MIPS_CPU_PREFETCH       0x00040000ull       /* CPU has usable prefetch */
#define   MIPS_CPU_VINT           0x00080000ull       /* CPU 支持 MIPSR2 向量中断模式*/
#define   MIPS_CPU_VEIC           0x00100000ull       /* CPU  支持 MIPSR2 EIC 中断模式*/
#define   MIPS_CPU_ULRI           0x00200000ull       /* CPU has ULRI feature */
#define   MIPS_CPU_PCI            0x00400000ull       /* CPU has Perf Ctr Int indicator */
#define   MIPS_CPU_RIXI           0x00800000ull       /* CPU has TLB Read/eXec Inhibit */
#define   MIPS_CPU_MICROMIPS      0x01000000ull       /* CPU has microMIPS capability */
#define   MIPS_CPU_TLBINV         0x02000000ull       /* CPU supports TLBINV/F */
#define   MIPS_CPU_SEGMENTS       0x04000000ull     /* CPU supports Segmentation Control registers */
#define   MIPS_CPU_EVA            0x80000000ull     /* CPU supports Enhanced Virtual Addressing */
#define   MIPS_CPU_HTW            0x100000000ull     /* CPU support Hardware Page Table Walker */
#define   MIPS_CPU_RIXIEX         0x200000000ull     /* CPU 具有读写阻碍异常处理程序    */
#define   MIPS_CPU_MAAR           0x400000000ull     /* MAAR(I) registers are present */
#define   MIPS_CPU_FRE            0x800000000ull     /* FRE & UFE bits implemented */
#define   MIPS_CPU_RW_LLB         0x1000000000ull    /* LLADDR/LLB writes are allowed */
#define   MIPS_CPU_XPA            0x2000000000ull /* CPU supports Extended Physical Addressing */
#define   MIPS_CPU_CDMM           0x4000000000ull    /* CPU has Common Device Memory Map */
#define   MIPS_CPU_BP_GHIST       0x8000000000ull     /* R12K+ Branch Prediction Global History */
```

●**cache_desc 结构体成员**：表示处理器各级缓存信息，结构体定义在/arch/mips/include/asm/cpu-info.h 头文件：

```
struct cache_desc {
    unsigned    int waysize;        /*每路字节数*/
    unsigned    short sets;         /*行数*/
    unsigned    char ways;          /*路数*/
    unsigned    char linesz;       /*每行字节数*/
    unsigned    char waybit;        /*Bits to select in a cache set */
    unsigned    char flags;         /*标记*/
};
```

内核在/arch/mips/kernel/setup.c 文件内静态定义了 cpuinfo_mips 结构体数组，每个 CPU 核对应一个数组项：

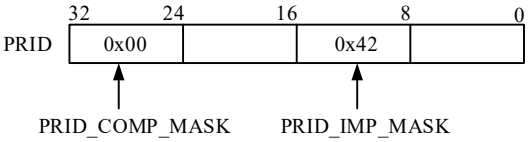
```
struct cpuinfo_mips    cpu_data[NR_CPUS]    __read_mostly;
```

获取当前运行 CPU 核 cpuinfo_mips 实例（指针）的宏定义在/arch/mips/include/asm/cpu-info.h 头文件：

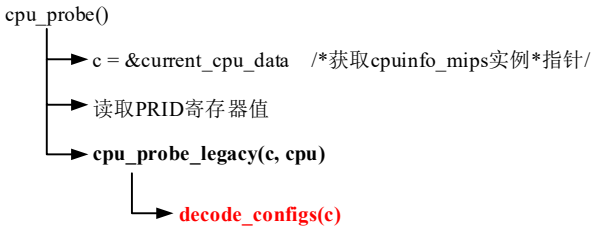
#define current_cpu_data　cpu_data[smp_processor_id()]　/*返回 cpuinfo_mips 实例地址*/

参数 NR_CPUS 定义在/include/linux/threads.h 头文件，NR_CPUS 与配置参数 CONFIG_NR_CPUS 相同，表示内核支持的最大 CPU 内核数量，这并不是实际处理器具有的 CPU 核数量。通常 32 位系统 NR_CPUS 值为 32，64 位系统为 64。

cpu_probe()函数在 setup_arch()函数中被调用，用于设置处理器核对应的 cpuinfo_mips 实例。cpu_probe()函数定义在/arch/mips/kernel/cpu-probe.c 文件内，函数内首先读取协处理器 PRID 寄存器值，以确定处理器产商和处理器版本编号。如下图所示，PRID 寄存器高 8 位表示产商编号，8-15 位表示处理器版本编号。cpu_probe()函数随后根据这两个编号调用相应的函数以填充 cpuinfo_mips 实例。



对于龙芯 1x 处理器，产商编号为 0，处理器版本编号为 0x42（PRID_IMP_LOONGSON_32），据此 cpu_probe()函数的调用函数如下：



cpu_probe()函数将调用 cpu_probe_legacy(c, cpu)函数设置 cpuinfo_mips 实例。cpu_probe_legacy()函数根据 PRID 寄存器值中处理器版本编号执行相应的操作，例如对于龙芯 1x 处理器，函数代码如下：

```
static inline void cpu_probe_legacy(struct cpuinfo_mips *c, unsigned int cpu)
{
    switch (c->processor_id & PRID_IMP_MASK) {
    ...
    case PRID_IMP_LOONGSON_32:  /* Loongson-1 */
        decode_configs(c);        /*设置 cpuinfo_mips 实例*/

        c->cputype = CPU_LOONGSON1;

        switch (c->processor_id & PRID_REV_MASK) {
        case PRID_REV_LOONGSON1B:
            __cpu_name[cpu] = "Loongson 1B";
            break;
        }
        break;
    }
}
```

cpu_probe_legacy()函数内调用 decode_configs()函数获取处理器配置寄存器信息，以设置 cpuinfo_mips

实例，decode_configs()函数代码简列如下（/arch/mips/kernel/cpu-probe.c）：

```
static void decode_configs(struct cpuinfo_mips *c)
{
    int ok;

    /*设置 MIPS32 和 MIPS64 处理器默认选项参数*/
    c->options = MIPS_CPU_4KEX | MIPS_CPU_4K_CACHE | MIPS_CPU_COUNTER |
                 MIPS_CPU_DIVEC | MIPS_CPU_LLSC | MIPS_CPU_MCHECK;

    c->scache.flags = MIPS_CACHE_NOT_PRESENT;

    set_ftlb_enable(c, !mips_ftlb_disabled);

    ok = decode_config0(c);          /*读取 config0 寄存器值，设置 cpuinfo_mips 实例*/
    BUG_ON(!ok);                     /* Arch spec violation!*/
    if (ok)
        ok = decode_config1(c);      /*读取 config1 寄存器值，设置 cpuinfo_mips 实例*/
    if (ok)
        ok = decode_config2(c);      /*读取 config2 寄存器值，设置 cpuinfo_mips 实例*/
    if (ok)
        ok = decode_config3(c);      /*读取 config3 寄存器值，设置 cpuinfo_mips 实例*/
    if (ok)
        ok = decode_config4(c);      /*读取 config4 寄存器值，设置 cpuinfo_mips 实例*/
    if (ok)
        ok = decode_config5(c);      /*读取 config5 寄存器值，设置 cpuinfo_mips 实例*/

    mips_probe_watch_registers(c);

    if (cpu_has_rixi) {
        set_c0_pagegrain(PG_IEC);
        back_to_back_c0_hazard();
        if (read_c0_pagegrain() & PG_IEC)
            c->options |= MIPS_CPU_RIXIEX;
    }
    ...
}
```

decode_configs()函数首先设置 c->options 成员的默认标记位，然后依次读取配置寄存器 0 至 5 的值，根据配置寄存器值设置 cpuinfo_mips 实例，decode_config*()函数都定义在/arch/mips/kernel/cpu-probe.c 文件内，函数比较简单，请读者自行阅读。

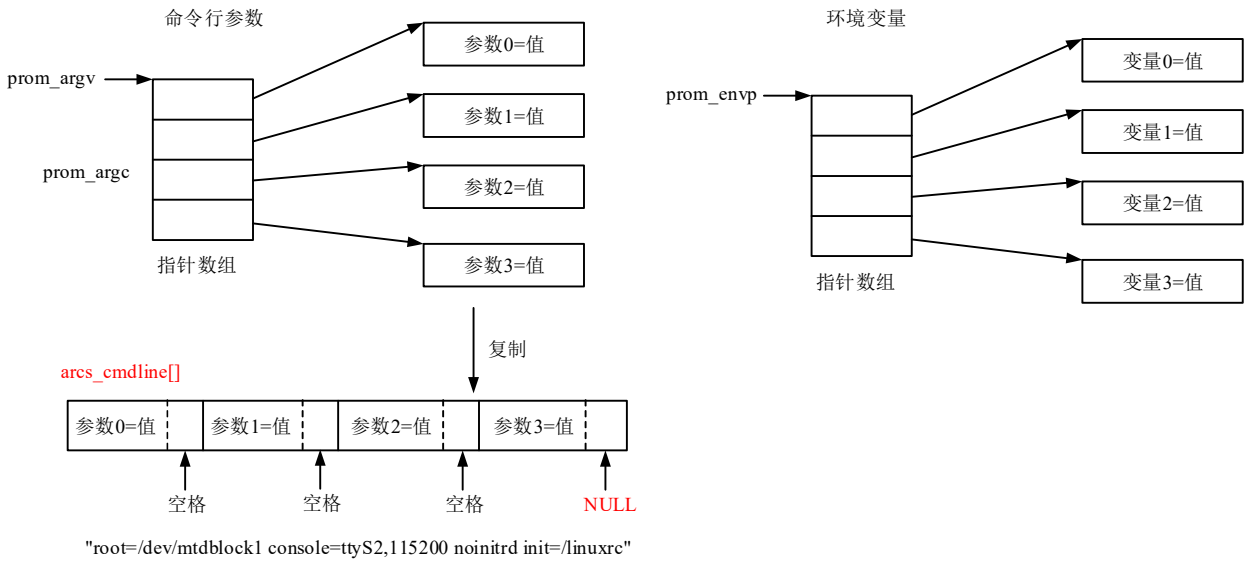
2.4.2 获取固件信息

prom_init()函数由板级（处理器）代码实现，主要用于获取引导加载程序（固件）传递的参数信息以及初始化必要的硬件。龙芯 1B 开发板源代码在/arch/mips/loongson32/common/prom.c 文件内实现了该函数。

```
void __init prom_init(void)
{
    void __iomem *uart_base;
    prom_argc = fw_arg0;     /*命令行参数数量*/
    prom_argv = (char **)fw_arg1;     /*命令行参数指针数组基地址*/
    prom_envp = (char **)fw_arg2;     /*环境变量指针数组基地址*/

    prom_init_cmdline();
                /*复制命令行参数字符串到 arcs_cmdline[]，/arch/mips/loongson32/common/prom.c*/

    memsize = env_or_default("memsize", DEFAULT_MEMSIZE);
                                        /*若环境变量中无 memsize，则设置默认值为 60MB*/
    highmemsize = env_or_default("highmemsize", 0x0);
                                        /*若环境变量中无 highmemsize，则设置默认值为 0x0*/

    if (strstr(arcs_cmdline, "console=ttyS3"))
        uart_base = ioremap_nocache(LS1X_UART3_BASE, 0x0f);      /*设置串口寄存器基地址*/
    else if (strstr(arcs_cmdline, "console=ttyS2"))
        uart_base = ioremap_nocache(LS1X_UART2_BASE, 0x0f);
    else if (strstr(arcs_cmdline, "console=ttyS1"))
        uart_base = ioremap_nocache(LS1X_UART1_BASE, 0x0f);
    else
        uart_base = ioremap_nocache(LS1X_UART0_BASE, 0x0f);

    setup_8250_early_printk_port((unsigned long)uart_base, 0, 0);
}
```

prom_init()函数内首先调用 prom_init_cmdline()函数将引导加载程序传递的命令行参数复制到全局字符数组 **arcs_cmdline[]**（以 NULL 结束），然后从环境变量中读取参数分别设置 **memsize** 和 **highmemsize** 全局变量，这两个变量分别表示物理内存大小和高端内存大小（单位 MB），最后设置串口设备 IO 寄存器基地址并输出信息。

命令行参数是形如**"root=/dev/mtdblock1 console=ttyS2,115200 noinitrd init=/linuxrc"**的“参数名称=值 参数名称=值”的字符串，用于控制内核行为或设置内核参数等。引导加载程序需要将字符串按空格划分成形如“参数名称=值”的单个参数（字符串），并将各参数基地址保存在数组中（指针数组），将数组基地址和命令行参数数量传递给内核，环境变量的传递也类似，如下图所示。



prom_init_cmdline()函数由平台（处理器）代码实现，它将命令行参数复制到 **arcs_cmdline[]**全局字符数组中，其中参数与参数间用空格字符隔开，字符数组结尾（最后一个参数结尾）是 NULL 字符（0x00）。

2.4.3 缓存操作初始化

处理器内缓存的操作是特定于体系结构（或处理器）的，因此对缓存的操作函数必须由体系结构代码实现。

MIPS 架构在/arch/mips/mm/cache.c 文件内声明并导出了处理器缓存操作接口函数，供内核其它部分调用（需包含/arch/mips/include/asm/cacheflush.h 头文件），而具体的实现函数由各处理器类型的代码实现。例如，处理器缓存操作接口函数声明如下：

```
void (*flush_cache_all)(void);          /*刷新整个缓存*/
void (*__flush_cache_all)(void);
void (*flush_cache_mm)(struct mm_struct *mm);       /*刷新指定进程地址空间的缓存*/
void (*flush_cache_range)(struct vm_area_struct *vma, unsigned long start,unsigned long end);
                                                    /*刷新指定地址区域的页*/
void (*flush_cache_page)(struct vm_area_struct *vma, unsigned long page,unsigned long pfn);
                                                    /*刷新单个页*/
void (*flush_icache_range)(unsigned long start, unsigned long end);        /*刷新指定区域的指令缓存*/
EXPORT_SYMBOL_GPL(flush_icache_range);
void (*local_flush_icache_range)(unsigned long start, unsigned long end);
EXPORT_SYMBOL_GPL(local_flush_icache_range);
...
/* MIPS specific cache operations */
void (*flush_cache_sigtramp)(unsigned long addr);
void (*local_flush_data_cache_page)(void * addr);
void (*flush_data_cache_page)(unsigned long addr);
void (*flush_icache_all)(void);

EXPORT_SYMBOL_GPL(local_flush_data_cache_page);
EXPORT_SYMBOL(flush_data_cache_page);
```

EXPORT_SYMBOL(flush_icache_all);

start_kernel()函数中调用 **cpu_cache_init()**函数用于初始化处理器缓存，并设置各处理器缓存操作接口函数。cpu_cache_init()函数定义在/arch/mips/mm/cache.c 文件内，函数内根据处理器缓存类型的不同调用不同的初始化函数。

在前面介绍的 decode_configs()函数中，MIPS32 和 MIPS64 兼容的处理器设置 **MIPS_CPU_4K_CACHE** 标记位，表示缓存类型为 4k_cache，因此 cpu_cache_init()函数调用关系如下：

```
void cpu_cache_init(void)
{
    ...
    if (cpu_has_4k_cache) {
        extern void __weak r4k_cache_init(void);

        r4k_cache_init();     /*/arch/mips/mm/c-r4k.c*/
    }
    ...
    setup_protection_map();     /*/arch/mips/mm/c-r4k.c*/
}
```

内核在/arch/mips/mm/c-r4k.c 文件内实现了缓存操作接口函数，以及 r4k_cache_init()初始化函数。

r4k_cache_init()函数初始化处理器指令、数据缓存，并设置处理器缓存操作接口函数指针，内核代码中可以通过这些接口函数对处理器核缓存进行操作，函数源代码请读者自行阅读。

另外，处理器 TLB 的操作函数与缓存操作函数类似，内核声明公共的操作函数，由具体处理器（架构）实现操作函数，本书后面将作介绍。

2.5 处理命令行参数

命令行参数是引导加载程序传递给内核的系统参数，是形如“参数名称=值 参数名称=值”的字符串，例如：**"root=/dev/mtdblock1 console=ttyS2,115200 noinitrd init=/linuxrc"**。命令行参数用于控制内核的行为或设置内核参数。在前面介绍的 prom_init()函数中命令行参数将被复制到全局字符数组 **arcs_cmdline[]**，内核在启动阶段将会对命令行参数进行处理。

模块参数是在模块代码中设置的参数（可视为外部变量，对外可见），在内核启动阶段可对内置模块（永久编译进内核的源文件）参数进行设置，在命令行参数中嵌入“**模块名.参数名称=值**”的字符可对内置模块参数进行赋值。在使用 insmode 或 modprobe 命令加载外部模块时，可设置模块参数值，例如：“insmode 模块名称 **参数名称=值**”。

内核对命令行参数和模块参数的处理采用了相同的函数，本节主要介绍命令行参数和模块参数处理函数的定义以及内核处理参数的机制。

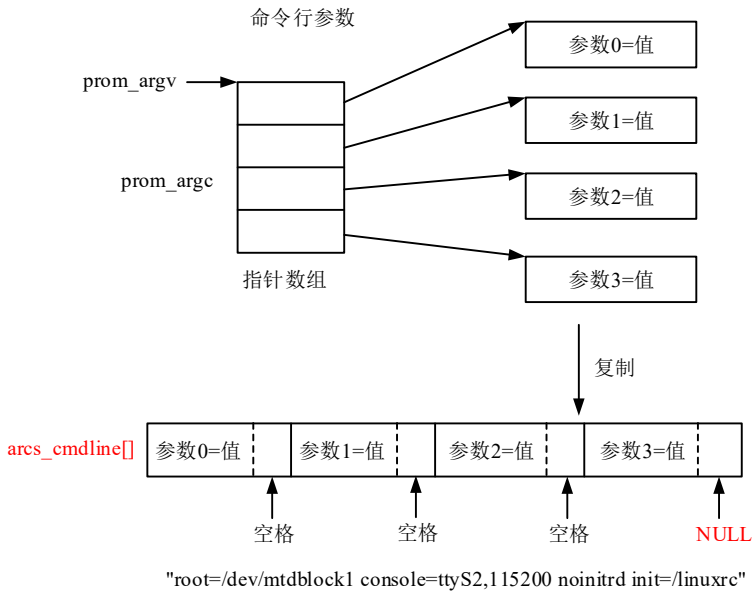
2.5.1 命令行参数

引导加载程序可向内核传递命令行参数，以控制内核的行为和参数，内核需要为各个命令行参数定义相应的处理函数，在启动函数中将扫描命令行参数并调用内核定义的处理函数对各命令行参数进行处理。

1 参数传递

在 MIPS 体系结构中，引导加载程序需要将命令行参数拆分成形如“参数名称=值”的分量，建立指

针数组指向各分量，参数数量保存至 a0 寄存器，指针数组地址保存至 a1 寄存器。在前面介绍的板级（平台）实现的 prom_init()函数中会复制引导加载程序传递的命令行参数至 **arcs_cmdline**[]字符数组，各参数之间用空格字符隔开，数组结尾字符为 NULL（0x00），如下图所示。



在体系结构相关的/arch/mips/kernel/setup.c 文件内定义了字符数组用于保存命令行参数字符串：

```
static char __initdata    command_line[COMMAND_LINE_SIZE];      /*传递给公共 start_kernel()函数*/
char __initdata    arcs_cmdline[COMMAND_LINE_SIZE];            /*体系结构相关代码使用*/
```

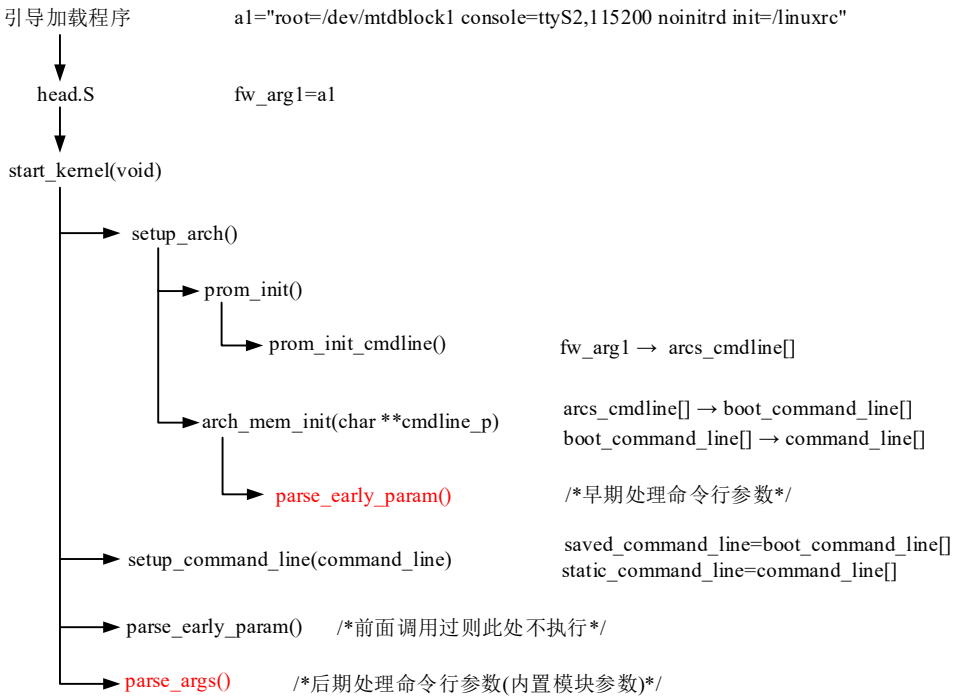
常数 COMMAND_LINE_SIZE 定义在/arch/mips/include/uapi/asm/setup.h 头文件内：

```
#define    COMMAND_LINE_SIZE      4096
```

在体系结构无关的/init/main.c 文件内定义了下列变量用于保存和指示命令行参数：

```
char __initdata    boot_command_line[COMMAND_LINE_SIZE];   /*体系结构无关的字符数组*/
char    *saved_command_line;                /*指向命令行参数字符串指针*/
static char    *static_command_line;        /*指向命令行参数字符串指针*/
```

在体系结构相关的 arch_mem_init()函数（/arch/mips/kernel/setup.c）会将 **arcs_cmdline**[]数组内容复制到 **command_line**[]和 **boot_command_line**[]字符数组，函数调用关系如下图所示。



启动函数中调用的 setup_command_line()函数将会为 saved_command_line 和 static_command_line 指针指向的字符数组分配空间，并复制命令行参数字符数组到分配的空间中。

内核在 **parse_early_param()**和 **parse_args()**函数中将会扫描命令行参数并对各参数（含内置模块参数）调用内核定义的处理函数对其进行处理。在加载外置模块时也将调用 **parse_args()**函数对模块参数进行处理。

2 注册参数处理函数

在内核代码中需要定义命令行参数的处理函数，并调用 **early_param(str, fn)**或**__setup(str, fn)**宏注册参数处理函数，str 为指向参数名称字符串的指针，fn 为处理函数指针。

early_param(str, fn)和__setup(str, fn)宏定义在/include/linux/init.h 头文件内：

```
#define early_param(str, fn)        \        /*str 名称字符串最后通常没有等号*/
        __setup_param(str, fn, fn, 1)

#define __setup(str, fn)            \      /*str 名称字符串最后通常有等号*/
        __setup_param(str, fn, fn, 0)

#define __setup_param(str, unique_id, fn, early)                \
    static const char   __setup_str_##unique_id[]   __initconst   __aligned(1) = str; \    /*创建字符数组*/
    static struct obs_kernel_param   __setup_##unique_id     \    /*创建 obs_kernel_param 实例*/
            __used __section(.init.setup)            \   /*obs_kernel_param 实例链接到指定的.init.setup 段*/
            __attribute__((aligned((sizeof(long))))) \
            = { __setup_str_##unique_id,   fn,   early }        /*设置 obs_kernel_param 实例*/
```

由以上宏定义可知 early_param(str, fn)和__setup(str, fn)宏都是通过__setup_param(str, unique_id, fn, early)宏创建命令行参数名称字符数组和 **obs_kernel_param** 结构体实例并初始化。

特别需要注意的是 **obs_kernel_param** 实例链接到内核目标文件**.init.setup** 段中（嵌入到初始化段），

也就是说所有的 obs_kernel_param 实例都链接在一起，组成一个数组。

在/include/asm-generic/vmlinux.lds.h 头文件中定义了**.init.setup** 段起始、结束地址的标号（地址）：

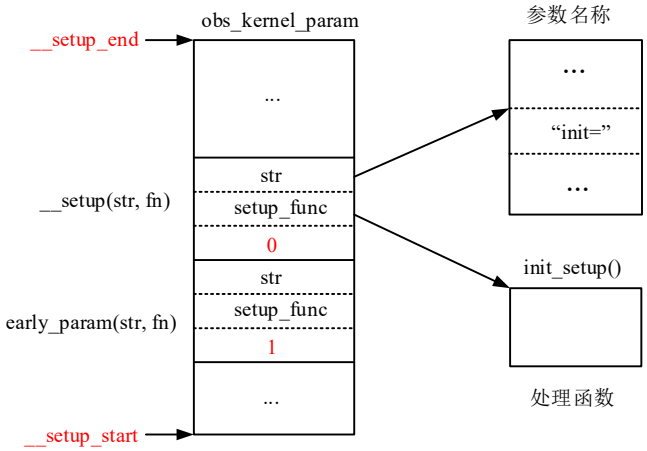
```
#define INIT_SETUP(initsetup_align)                  \
        . = ALIGN(initsetup_align);                  \
        VMLINUX_SYMBOL(__setup_start) = .;       \
        *(.init.setup)                               \
        VMLINUX_SYMBOL(__setup_end) = .;
```

下面看一下 obs_kernel_param 结构体的定义（/include/linux/init.h）：

```
struct obs_kernel_param {
    const char *str;                /*指向参数名称字符串*/
    int (*setup_func)(char *);      /*处理函数指针*/
    int early;                      /*是否在启动早期处理，1 表示是，0 表示在后期处理*/
};
```

__setup_param()宏中创建的 obs_kernel_param 实例 str 成员指向命令行参数字符串，setup_func 保存参数处理函数指针（入口地址），early 值为 0 或 1，1 表示参数需要在内核启动早期处理，0 表示在后期处理。

__setup(str, fn)和 early_param(str, fn)宏创建的 obs_kernel_param 实例链接成数组，如下图所示：



内核在启动阶段对传递进来的每个命令行参数扫描**.init.setup** 段的 obs_kernel_param 实例，比对参数名称与实例 str 成员指向的字符串，如果相同则调用实例中 setup_func 成员指向的处理函数，并将参数值字符数组指针（参数值指针）传递给处理函数作为参数。

下面以一个简单的例子来说明内核代码中如何定义并注册命令行参数处理函数。例如，在体系结构相关的/arch/mips/kernel/setup.c 文件内定义了“rd_size”参数的处理函数为 rd_size_early()：

```
static int __init rd_size_early(char *p)       /*p 指向参数值字符串*/
{
        initrd_end += memparse(p, &p);
        return 0;
}
early_param("rd_size", rd_size_early);     /*注册早期处理命令行参数处理函数*/
```

在/init/main.c 内定义了“init”参数的处理函数为 init_setup()：

```
static int __init    init_setup(char *str)      /*str 指向参数值字符串*/
{
    unsigned int i;

    execute_command = str;      /*execute_command 指向参数值字符数组*/
    for (i = 1; i < MAX_INIT_ARGS; i++)
        argv_init[i] = NULL;
     return 1;
}
__setup("init=", init_setup);       /*后期处理命令行参数处理函数*/
```

3 常用命令行参数

下面简要介绍几个常用的命令行参数及其用途：

●**root=***：设置挂载外部根文件系统的设备文件名称，处理函数定义在/init/do_mounts.c 文件内，负责将参数值字符数组复制到 saved_root_name[]全局字符数组。

●**rdinit=***：在初始根文件系统中查找第一个用户进程目标文件的路径（文件名称）。

●**init=***：在挂载的外部介质根文件系统中查找第一个用户进程目标文件的路径（文件名称）。

●**rootwait**：设置加载根文件系统时是否等待，以便外部设备初始化完成，处理函数在/init/do_mounts.c 文件内，不需要参数值。

●**rw/ro**：设置挂载外部根文件系统的读写/只读属性，系统默认为只读挂载，不需要参数值，处理函数定义在/init/do_mounts.c 文件内。

●**rootfstype=***：设置 rootfs 根文件系统的类型，处理函数定义在/init/do_mounts.c 文件内，参数值为文件系统类型名称。

●**rootdelay=***：设置加载根文件系统前等待的时间，以便外部介质初始化完成。处理函数定义在文件/init/do_mounts.c 内，参数值为整数值。

●**noinitrd**：表示采用 ramfs，而不采用 ramdisk，不需要参数值，处理函数定义在/init/do_mounts_initrd.c 文件内。

2.5.2 模块参数

模块参数可理解成模块代码中定义的，可导出到模块外部的变量，用户可以在模块执行前对模块参数进行赋值。在加载内核时，用户可以通过命令行参数对于内置模块（持久编译入内核）的模块参数进行赋值。对内置模块参数进行赋值的命令行参数形式为“**模块名.模块参数名称=值**”，对于外置的模块，可以在执行 insmode 或 modprobe 命令加载模块时，通过命令行参数对其参数赋值，形式为“**模块参数名称=值**”。内核在启动阶段或加载模块时，会将外部命令行参数传递的模块参数值赋予模块中参数（变量）。

本小节介绍模块代码中如何定义模块参数，下一小节将介绍在处理命令行参数时对模块参数的赋值。

模块参数其实就是模块代码中的全局变量，只不过在加载模块时可从外部对其赋初值。模块参数的类型包括：byte，short，ushort，int，uint，long，ulong，charp(字符指针)，bool，invbool（反 bool）等。

内核在/include/linux/moduleparam.h 头文件内定义了声明模块内变量为模块参数的宏，例如，如下是声明 book_num 为 int 类型模块参数的代码（其它声明模块参数的宏请读者自行阅读）：

```
static int book_num=4000;        /*参数初值*/
module_param(book_num,int,S_IRUGO);
```

module_param()宏定义如下：

```
#define module_param(name, type, perm)   \   /*name：变量名称，type：类型，perm：读写权限*/
    module_param_named(name, name, type, perm)

#define module_param_named(name, value, type, perm)    \
    param_check_##type(name, &(value));                \    /*参数类型检查*/
    module_param_cb(name, &param_ops_##type, &value, perm);   \/*&value：变量指针（地址）*/
    __MODULE_PARM_TYPE(name, #type)          /*模块信息*/

#define module_param_cb(name, ops, arg, perm)    \
    __module_param_call(MODULE_PARAM_PREFIX, name, ops, arg, perm, -1, 0)
                                        /*ops：kernel_param_ops 实例指针，arg：变量指针*/

#define __module_param_call(prefix, name, ops, arg, perm, level, flags)   \   /*level=-1，flags=0*/
    static const char __param_str_##name[] = prefix #name;       \       /*模块参数名称字符数组*/
    static struct kernel_param __moduleparam_const __param_##name \     /*kernel_param 实例*/
    __used __attribute__ ((unused,__section__ ("__param"),aligned(sizeof(void *)))) \
    = { __param_str_##name, THIS_MODULE, ops,                \
        VERIFY_OCTAL_PERMISSIONS(perm), level, flags, { arg } }
```

module_param()宏最终的效果是创建 kernel_param 结构体实例，并将实例链接到指定的**"__param"**段内，段内其实就是 kernel_param 实例数组。

kernel_param 结构体定义如下（/include/linux/moduleparam.h）：

```
struct kernel_param {
    const char *name;       /*参数名称字符数组指针*/
    struct module *mod;     /*模块指针*/
    const struct kernel_param_ops *ops;      /*参数类型操作结构*/
    const u16 perm;      /*读写权限*/
    s8 level;           /*等级*/
    u8 flags;          /*标记*/
    union {
        void *arg;      /*指向变量的指针，变量地址*/
        const struct kparam_string *str;
        const struct kparam_array *arr;
    };
};
```

kernel_param_ops 结构体定义如下，表示对模块参数（变量）的操作：

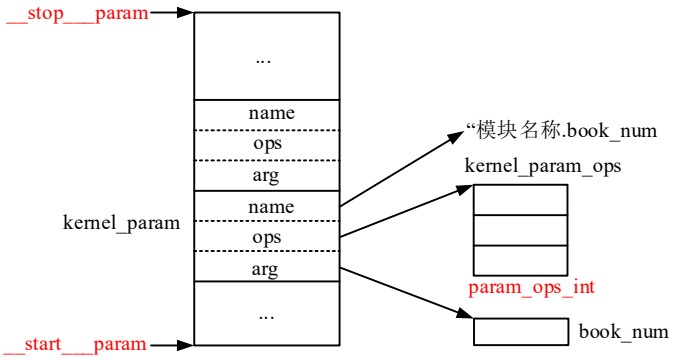
```
struct kernel_param_ops {
    unsigned int flags;      /*标记*/
    int (*set)(const char *val, const struct kernel_param *kp);   /*设置变量值*/
    int (*get)(char *buffer, const struct kernel_param *kp);      /*获取变量值*/
```

```
    void (*free)(void *arg);          /*释放模块参数*/
};
```

kernel_param_ops 结构体中 set()成员函数用于依据 val 指向的命令行参数（或加载模块命令行参数）中传递的模块参数字符数组，设置模块内变量的值。get()成员函数用于获取模块内变量的值。

内核在/kernel/params.c 文件内定义了各种类型模块参数对应的 kernel_param_ops 实例，实例名称为 param_ops_*type*，type 为参数类型，例如，charp 类型对应实例为 param_ops_charp。

module_param()宏中定义的 kernel_param 实例 **ops** 成员指向 param_ops_*type* 实例，**arg** 为变量指针，**name** 成员指向模块参数名称，对于内置模块，参数名称为：模块名称.变量名称，对于外置模块，参数名称与模块中的变量名称相同。**module_param()**宏创建的 kernel_param 实例如下图所示：



kernel_param 实例链接到内核目标文件的只读数据段（/include/asm-generic/vmlinux.lds.h）：

```
#define RO_DATA_SECTION(align)            \     /*只读数据段*/
    . = ALIGN((align));                   \
    .rodata     : AT(ADDR(.rodata) - LOAD_OFFSET) {
    ...
    __param : AT(ADDR(__param) - LOAD_OFFSET) {          \
        VMLINUX_SYMBOL(__start___param) = .;             \
        *(__param)                                       \
        VMLINUX_SYMBOL(__stop___param) = .;              \
    }                                                    \
    ...
}
```

__start___param 和__stop___param 标号为 kernel_param 实例数组的起始、结束地址。

内核启动阶段处理命令行参数传递的模块参数时，逐个扫描 kernel_param 实例数组，查找名称相同的 kernel_param 实例，对名称匹配的实例调用其关联的 kernel_param_ops 实例中的 set()函数设置模块中相应变量的值。

2.5.3 参数处理

现在我们可以来讨论内核如何处理命令行参数了。由前面的介绍可知，内核定义的命令行参数处理函数由 obs_kernel_param 实例数组表示，内置模块中定义的模块参数（变量）由 kernel_param 实例数组表示。

内核对每个传递的命令行参数搜索 kernel_param 实例数组或 obs_kernel_param 实例数组，查找名称匹配的实例，如果匹配的是 kernel_param 实例，则调用其关联的 kernel_param_ops 实例中的 set()函数设置模块中相应变量的值，如果匹配的是 obs_kernel_param 实例，则调用其中的 setup_func()函数，处理参数。

1 通用处理函数

内核在/kernel/params.c 文件内定义了处理命令行参数和模块参数的通用函数 **parse_args**()，函数定义如下：

```
char *parse_args(const char *doing,char *args,const struct kernel_param *params,unsigned num,
                 s16 min_level,s16 max_level,void *arg,
                 int (*unknown)(char *param, char *val,const char *doing, void *arg))
/*
*doing：字符串，标识当前工作（处理的是哪类参数），主要用于信息显示，
*args：命令行参数字符数组指针，
*params：指向 kernel_param 实例数组，用于处理模块参数，num：kernel_param 实例数组项数，
*min_level、max_level：标识遍历 kernel_param 实例中的最小等级和最大等级，
*arg：指针参数，通常为 NULL，注意与 args 参数区分开来，
*unknown：处理单个命令行参数的函数指针，不同的时机调用 parse_args()函数，unknown 参数不同。
*/
{
    char *param, *val;
    args = skip_spaces(args);        /*跳过字符串开头的空格*/
    ...
    while (*args) {    /*扫描命令行参数字符数组，直至结束（扫描到 NULL）或遇到"--"参数*/
        int ret;
        int irq_was_disabled;

        args = next_arg(args, &param, &val);       /*/kernel/params.c*/
        /*逐个取出参数，参数间由空格隔开，param 指向取出参数名称字符数组，
        *val 指向参数值字符数组，函数返回下一个参数的指针，赋予 args 参数。*/

        if (!val && strcmp(param, "--") == 0)
            return args;                /*遇到"--"参数（没有值），返回"--"参数后面一个参数指针*/
        irq_was_disabled = irqs_disabled();
        ret = parse_one(param, val, doing, params, num,min_level, max_level, arg, unknown);
                                                        /*处理单个参数，/kernel/params.c*/
        ...
        switch (ret) {
        ...          /*根据处理函数返回结果，输出信息*/
        }
    }      /*扫描命令行参数结束*/
    return NULL;
}
```

parse_args()函数对 **args** 参数传递的命令行参数字符数组以空格字符进行拆分，遍历命令行中参数，直至命令行参数末尾或遇到“--”参数。如果在遍历命令行参数时，遇到了“--”参数（没有值），parse_args()函数将返回，返回值为“--”参数后面一个参数的指针。

parse_args()函数对遍历的每个参数调用函数 **parse_one()**进行处理。parse_one()函数 params 参数指向参数名称字符数组（‘=’字符改为 NULL），val 指向参数值字符数组（数组结尾空格改成 NULL）。

parse_one()函数用于处理单个命令行参数，函数定义如下（/kernel/params.c）：

```
static int parse_one(char *param,char *val,const char *doing,const struct kernel_param *params,
            unsigned num_params,s16 min_level,s16 max_level,void *arg,
            int (*handle_unknown)(char *param, char *val,const char *doing, void *arg))
/*param：指向参数名称，val：指向参数值，handle_unknown：参数处理函数，其它参数同 parse_args()*/
{
    unsigned int i;
    int err;

    for (i = 0; i < num_params; i++) {      /*遍历 kernel_param 实例数组，处理模块参数*/
        if (parameq(param, params[i].name)) {  /*比对模块参数名称，“模块名.参数名称”*/
            if (params[i].level < min_level|| params[i].level > max_level)     /*等级不在指定范围内*/
                return 0;
            if (!val &&!(params[i].ops->flags & KERNEL_PARAM_OPS_FL_NOARG))
                return -EINVAL;
            pr_debug("handling %s with %p\n", param,params[i].ops->set);
            kernel_param_lock(params[i].mod);
            param_check_unsafe(&params[i]);
            err = params[i].ops->set(val, &params[i]);
                                /*kernel_param_ops 实例中的 set()函数设置模块中变量值*/
            kernel_param_unlock(params[i].mod);
            return err;      /*函数返回*/
        }
    }

    /*函数没有传递 kernel_param 实例数组，或在 kernel_param 实例数组中没有找到匹配的项*/
    if (handle_unknown) {
        pr_debug("doing %s: %s='%s'\n", doing, param, val);
        return handle_unknown(param, val, doing, arg);     /*调用 handle_unknown()函数处理参数*/
    }
    pr_debug("Unknown argument '%s'\n", param);
    return -ENOENT;
}
```

parse_one()函数不难理解，如果参数传递了 kernel_param 实例数组，则先在 kernel_param 实例数组中查找与参数名称匹配的项，如果找到匹配的项，调用关联 kernel_param_ops 实例中的 set()函数设置模块中变量值，函数返回。如果 parse_one()函数没有传递 kernel_param 实例数组，或在 kernel_param 实例数组中没有查找到与参数名称匹配的项，则调用参数传递的 **handle_unknown()**函数处理参数。

2 早期处理命令行参数

内核在体系结构相关的 arch_mem_init()函数或启动函数 start_kernel()中将调用 **parse_early_param()**函

数处理需要在早期处理的命令行参数（由 **early_param(str, fn)**宏声明的处理函数，“--”参数之前的参数），随后调用 **parse_args()**函数处理“--”参数之前的其它命令行参数和内置模块参数，最后调用 **parse_args()**函数处理“--”参数之后的参数。下面先看 parse_early_param()函数的实现。

parse_early_param()函数定义在/init/main.c 文件内，代码如下：

```
void __init parse_early_param(void)
{
    static __initdata int done = 0;     /*静态变量，保证此函数只被执行一次*/
    static __initdata char tmp_cmdline[COMMAND_LINE_SIZE];     /*保存命令行参数的静态变量*/

    if (done)
        return;

    strlcpy(tmp_cmdline, boot_command_line, COMMAND_LINE_SIZE);
                                                    /*复制命令行参数字符串至静态变量*/
    parse_early_options(tmp_cmdline);        /*处理各个参数，/init/main.c*/
    done = 1;                                /*标记函数已经执行过*/
}
```

parse_early_param()函数内部保证此函数只被执行一次，函数内调用 **parse_early_options()**函数处理命令行参数，函数定义如下：

```
void __init parse_early_options(char *cmdline)
{
    parse_args("early options", cmdline, NULL, 0, 0, 0, NULL,do_early_param);    /*/kernel/params.c*/
}
```

此处没有向 parse_args()函数传递 kernel_param 实例数组，因此所有命令行参数都由 **do_early_param()**函数处理。

do_early_param()函数定义在 init/main.c 文件内，用于处理单个参数，代码如下：

```
static int __init do_early_param(char *param, char *val,const char *unused, void *arg)
{
    const struct obs_kernel_param *p;

    /*扫描 obs_kernel_param 实例数组，找到与参数名称匹配的项，调用其 setup_func(val)函数*/
    for (p = __setup_start; p < __setup_end; p++)            /*扫描 obs_kernel_param 实例段*/
    {
        if ((p->early && parameq(param, p->str)) ||      /*名称相同且 early 成员值非 0 的实例*/
            (strcmp(param, "console") == 0 &&
             strcmp(p->str, "earlycon") == 0))
        {
            if (p->setup_func(val) != 0)     /*调用处理函数，val 为参数值指针*/
                pr_warn("Malformed early option '%s'\n", param);
        }
    }
    return 0;
```

}

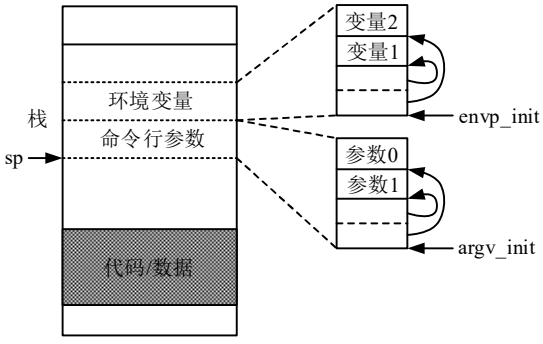
do_early_param()函数比较简单，函数内扫描 obs_kernel_param 实例数组，比对实例 str 指向的字符数组与参数名称字符数组，如果相同且实例 early 成员值非 0，则调用 obs_kernel_param 实例中 setup_func()函数处理参数。

3 处理其它参数

启动函数 start_kernel()在调用 parse_early_param()函数处理需要在早期处理的命令行参数后，将继续调用 parse_args()函数处理其它参数，如下所示：

```
asmlinkage __visible void __init start_kernel(void)
{
    ...
    parse_early_param();          /*处理由 early_param(str, fn)宏声明处理函数的命令行参数*/

    after_dashes = parse_args("Booting kernel", static_command_line, __start___param,
                    __stop___param - __start___param,-1, -1, NULL, &unknown_bootoption);
                        /*处理“--”参数之前的其它命令行参数（含模块参数），__setup(str, fn)*/
    if (!IS_ERR_OR_NULL(after_dashes))
        parse_args("Setting init args", after_dashes, NULL, 0, -1, -1,NULL, set_init_arg);
                                                    /*处理“--”参数之后的命令行参数*/
    ...
}
```

在介绍其它命令行参数的处理之前，先介绍一下用户进程的命令行参数和环境变量。在创建进程时，进程栈底保存了传递给进程的命令行参数和环境变量，如下图所示。



用户地址空间

命令行参数和环境变量都是形如“名称=值”的字符串。栈中保存了命令行参数和环境变量指针数组，数组项指向命令行参数或环境变量。用户可以通过引导加载程序传递的命令行参数设置系统中第一个用户进程的命令行参数和环境变量。

内核在/init/main.c 文件内定义的两个字符串指针数组，分别用于向第一个用户进程（init 进程）传递命令行参数和环境变量。

```
static const char *argv_init[MAX_INIT_ARGS+2] = { "init", NULL, };        /*初始进程命令行参数*/
const char *envp_init[MAX_INIT_ENVS+2] = { "HOME=/", "TERM=linux", NULL, }; /*初始环境变量*/
static const char *panic_later, *panic_param;
```

对于“--”参数之前参数（不含模块参数），如果内核没有声明处理函数，且具有参数值，参数将作

为第一个用户进程的环境变量，保存至 envp_init[]数组。如果 envp_init[]数组中已存在同名的环境变量，则覆盖它。如果是没有参数值的参数，将作为第一个用户进程的命令行参数，保存至 argv_init[]数组。

对于“--”参数之后的参数将一律设为第一个用户进程的命令行参数，详见下文。

■处理“--”之前参数

在 start_kernel()中调用 parse_args()函数处理“--”参数之前参数，如下所示：

```
after_dashes = parse_args("Booting kernel", static_command_line, __start___param,
                          __stop___param - __start___param,-1, -1, NULL, &unknown_bootoption);
```

parse_args()函数此处处理的是“--”参数之前的命令行参数，函数返回“--”参数之后的命令行参数指针，保存在 after_dashes 变量。函数传递了 kernel_param 实例数组，在此处将处理模块参数，对于非模块参数将调用 unknown_bootoption()函数处理。

unknown_bootoption()函数定义在/init/main.c 文件内，代码如下：

```
static int __init unknown_bootoption(char *param, char *val,const char *unused, void *arg)
{
    repair_env_string(param, val, unused, NULL);  /*恢复参数名称后面的‘=’字符*/

    /*处理命令行参数，由__setup(str, fn)宏声明处理函数的命令行参数*/
    if (obsolete_checksetup(param))      /*处理命令行参数，/init/main.c*/
        return 0;

    /*以下是处理内核没有定义处理函数的命令行参数和模块参数*/
    /*没有对应 kernel_param 实例的模块参数，返回 0，不处理*/
    if (strchr(param, '.') && (!val || strchr(param, '.') < val))
        return 0;

    if (panic_later)
        return 0;

    /*在内核中没有定义处理函数的命令行参数，设置为第一个用户进程命令行参数或环境变量*/
    if (val) {      /*参数值不为空的命令行参数，设为进程环境变量*/
        unsigned int i;
        for (i = 0; envp_init[i]; i++) {
            if (i == MAX_INIT_ENVS) {
                panic_later = "env";
                panic_param = param;
            }
            if (!strncmp(param, envp_init[i], val - param))  /*如果参数名称等于环境变量名称*/
                break;          /*跳出循环，覆盖现有的同名环境变量*/
        }       /*for 循环结束*/
        envp_init[i] = param;       /*添加到默认环境变量列表*/
    } else {          /*参数值为空的命令行参数，设为第一个用户进程命令行参数*/
```

```
            unsigned int i;
            for (i = 0; argv_init[i]; i++) {
                if (i == MAX_INIT_ARGS) {
                    panic_later = "init";
                    panic_param = param;
                }
            }
            argv_init[i] = param;     /*添加到默认命令行参数列表*/
        }
        return 0;
    }
```

unknown_bootoption()函数首先调用 obsolete_checksetup(param)函数，检查内核是否通过__setup(str, fn)宏声明了此参数的处理函数，如果是则调用 fn()处理函数，函数返回；如果内核没有声明处理函数，且是模块参数，将直接跳过，函数返回（不处理模块参数）；如果是没有声明处理函数的命令行参数，将做如下处理：

（1）如果是具有参数值的参数，将作为第一个用户进程的环境变量，保存至 envp_init[]数组。如果 envp_init[]数组中已存在同名的环境变量，则覆盖它。

（2）如果是没有参数值的参数，将作为第一个用户进程的命令行参数，保存至 argv_init[]数组。

■处理“--”之后参数

启动函数 start_kernel()在第二次调用 parse_args()函数时将处理命令行参数中“--”参数之后的参数，函数调用如下：

```
if (!IS_ERR_OR_NULL(after_dashes))
        parse_args("Setting init args", after_dashes, NULL, 0, -1, -1,NULL, set_init_arg);
```

此处没有传递 kernel_param 实例数组，因此不处理模块参数（不识别模块参数），处理单个参数的函数为 **set_init_arg()**，函数定义在/init/main.c 文件内。

```
static int __init set_init_arg(char *param, char *val,const char *unused, void *arg)
{
    unsigned int i;

    if (panic_later)       /*命令行参数或环境变量指针数组是否已满*/
        return 0;

    repair_env_string(param, val, unused, NULL);     /*参数名称设置‘=’字符*/

    for (i = 0; argv_init[i]; i++) {
        if (i == MAX_INIT_ARGS) {
            panic_later = "init";
            panic_param = param;
            return 0;
        }
    }
```

```
        argv_init[i] = param;     /*一律设为第一个用户进程命令行参数*/
        return 0;
    }
```

set_init_arg()函数比较简单，就是将“--”参数之后的参数一律设为第一个用户进程的命令行参数。内核在运行第一个用户进程时会将 argv_init[]和 envp_init[]指示的命令行参数和环境变量传递给进程。

2.5.4 小结

用户通过引导加载程序向内核传递的命令行参数形式如下：

"root=/dev/mtdblock1 console=ttyS2,115200 noinitrd init=/linuxrc"

命令行参数是形如“参数名称=值”分量，也可以是内置模块的模块参数，形如“模块名.模块参数名称=值”。

内核需要定义各命令行参数的处理函数，并用 early_param(str, fn)或__setup(str, fn)宏声明这些处理函数，前者表示参数需要在早期处理，后者表示参数在后期处理，这两个宏的作用是定义 obs_kernel_param 结构体实例，所有的 obs_kernel_param 实例链接成一个数组。

模块参数用于设置内置模块中变量的值，模块代码中需要通过 module_param(name, type, perm)宏等，将模块中变量声明为模块参数（可以从外部设置其初始值），这些宏的作用是定义 kernel_param 结构体实例，这些实例链接成数组。

启动函数处理命令行参数时，通过“--”参数将命令行参数分成两部分，参数处理流程如下：

（1）处理“--”参数前需要早期处理的命令行参数，对参数调用 early_param(str, fn)宏声明的处理函数 fn()。

（2）处理“--”参数前的后期处理参数，调用__setup(str, fn)宏声明的处理函数 fn()，如果是模块参数，通过同名 kernel_param 实例中关联 ernel_param_ops 实例中的 set()函数设置模块变量的值。对于没有声明处理函数（不含模块参数），且具有参数值的参数，将作为第一个用户进程的环境变量。如果是没有参数值的参数，将作为第一个用户进程的命令行参数。

（3）处理“--”参数之后的命令行参数，一律设为第一个用户进程的命令行参数。

2.6 剩余初始化

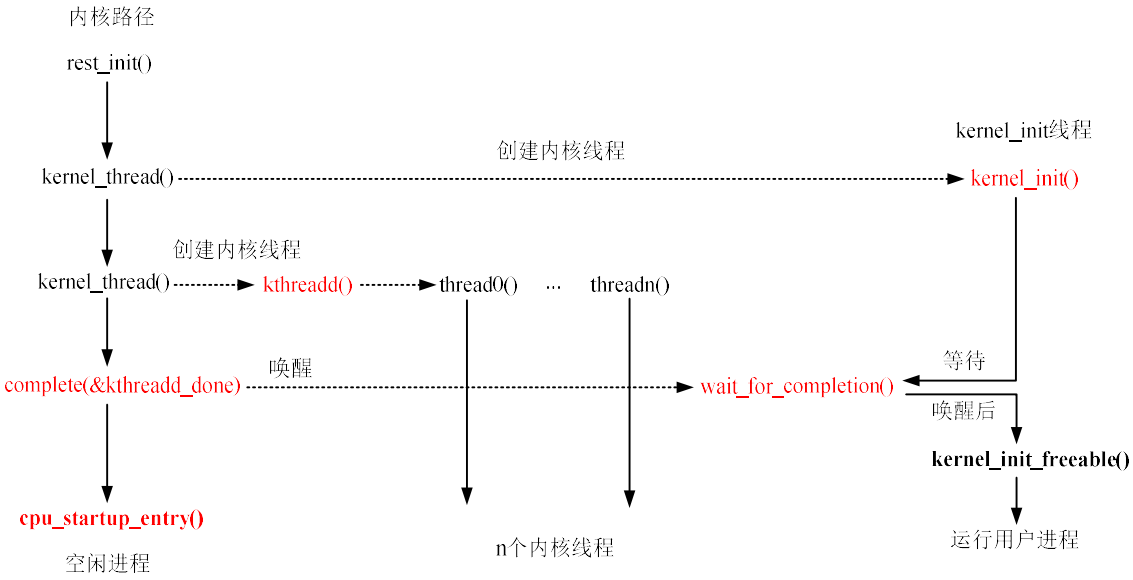
内核启动函数 start_kernel()在最后调用 rest_init()函数完成剩余的初始化工作，其实这里还有许多工作要做。rest_init()函数内将创建一批内核线程，其中 kernel_init 线程继续完成内核初始化工作并最后转变成第一个用户进程。内核路径将继续往下执行 rest_init()函数，最后进入一个无限循环，变成系统中的空闲进程（实际为内核线程），在系统无事可做时将运行该空闲进程。

2.6.1 内核路径

内核在 rest_init()函数中将创建内核线程，从此处开始系统内就有多个进程（线程）在并行运行了。rest_init()函数在创建完内核线程后将进入一个无限循环，成为系统中的空闲进程，这是内核自身运行的终点。

1 创建内核线程

rest_init()函数执行流程简列如下图所示，其中实线表示函数调用，虚线表示并行执行：



rest_init()函数内首先创建 kernel_init 内核线程，kernel_init 线程并不会立即运行，而是进入睡眠等待，rest_init()函数随后又通过创建 kthreadd 线程来创建一批内核线程，而后唤醒睡眠等待的 kernel_init 线程，rest_init()函数最后执行函数 cpu_startup_entry()，进入无限循环，内核路径转为空闲进程。

rest_init()函数调用之后，系统中存在多个并行运行的内核线程，其中 kernel_init 线程继续完成内核各子系统的初始化工作，挂载外部存储介质根文件系统等，最后选择用户进程目标文件运行第一个用户进程。

rest_init()函数在/init/main.c 文件内实现，代码如下：

```
static noinline void __init_refok rest_init(void)
{
    int pid;

    rcu_scheduler_starting();
                    /*/include/linux/rcutiny.h，没有配置 DEBUG_LOCK_ALLOC 则为空操作*/
    smpboot_thread_init();

    kernel_thread(kernel_init, NULL, CLONE_FS);        /*创建 kernel_init 内核线程，见下一小节*/
    numa_default_policy();
    pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
                                /*创建 kthreadd 内核线程，用于创建内核线程，见第 5 章*/
    rcu_read_lock();
    kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
                                    /*由 pid 查找 kthreadd 线程 task_struct 实例*/
    rcu_read_unlock();
    complete(&kthreadd_done);     /*唤醒 kernel_init 线程*/

    init_idle_bootup_task(current);
                    /*设置内核自身（空闲进程）调度器类为 idle_sched_class，/kernel/sched/core.c*/
    schedule_preempt_disabled();
    cpu_startup_entry(CPUHP_ONLINE);   /*空闲进程，/kernel/sched/idle.c*/
```

}

rest_init()函数内先后调用 kernel_thread()函数创建 kernel_init 和 kthreadd 内核线程，在创建完 kthreadd 线程后唤醒 kernel_init 线程。kernel_init 线程执行的工作将在下一小节中介绍，kthreadd 内核线程用于创建其它的内核线程，它是一个用来创建内核线程的内核线程，详见第 5 章。

rest_init()函数最后调用 cpu_startup_entry()函数进入一个无限循环，内核路径最终转为系统的空闲进程，这是内核自身执行路径最终的归宿。

2 空闲进程

cpu_startup_entry()函数定义在/kernel/sched/idle.c 文件内，代码简列如下：

```
void cpu_startup_entry(enum cpuhp_state state)
{
    ...
    arch_cpu_idle_prepare();  /*体系结构代码定义，否则为空操作，MIPS 为空，/kernel/sched/idle.c*/
    cpu_idle_loop();    /*空闲进程执行函数，/kernel/sched/idle.c*/
}
```

空闲进程最终的执行函数为 cpu_idle_loop()，函数代码如下：

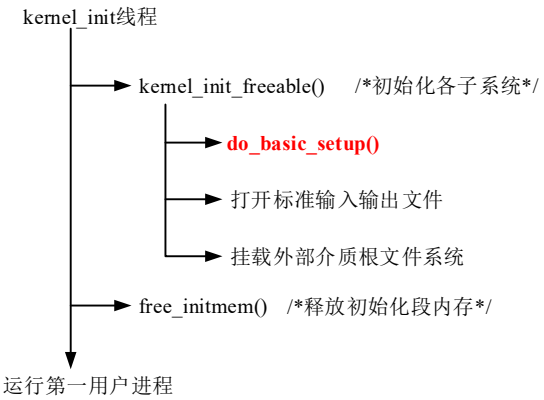
```
static void cpu_idle_loop(void)
{
    while (1) {         /*无限循环*/
        __current_set_polling();    /*设置进程 TIF_POLLING_NRFLAG 标记位*/
        tick_nohz_idle_enter();     /*激活动态时钟，/kernel/time/tick-sched.c*/

        while (!need_resched()) {   /*不需要重调度，进入循环*/
            check_pgt_cache();
            rmb();

            if (cpu_is_offline(smp_processor_id())) {
                rcu_cpu_notify(NULL, CPU_DYING_IDLE, (void *)(long)smp_processor_id());
                smp_mb();                   /* all activity before dead. */
                this_cpu_write(cpu_dead_idle, true);
                arch_cpu_idle_dead();
            }

            local_irq_disable();    /*关中断*/
            arch_cpu_idle_enter();  /*空操作*/

            if (cpu_idle_force_poll || tick_check_broadcast_expired())
                cpu_idle_poll();
            else
                cpuidle_idle_call();    /*空闲进程主要的执行函数，电源管理，/kernel/sched/idle.c*/

            arch_cpu_idle_exit();   /*空操作*/
```

```
            }       /*while 循环结束*/
            /*需要执行进程调度，执行以下代码*/
            preempt_set_need_resched();       /*空操作*/
            tick_nohz_idle_exit();            /*退出动态时钟，/kernel/time/tick-sched.c*/
            __current_clr_polling();

            smp_mb__after_atomic();

            sched_ttwu_pending();
            schedule_preempt_disabled();      /*进程调度，运行其它进程，/kernel/sched/core.c*/
        }
    }
```

空闲进程中将激活动态时钟（见第 6 章），执行其它的一些内核工作，并检测是否需要重调度，如果需要则执行进程调度，使处理器运行其它的就绪进程。

2.6.2 kernel_init 线程

kernel_init 内核线程由 rest_init()函数创建，并在 kthreadd 内核线程创建后被唤醒。kernel_init 内核线程执行函数为 kernel_init()，函数定义在/init/main.c 文件内。kernel_init()函数执行流程简列如下图所示：



kernel_init_freeable()函数将继续初始化内核各子系统，为线程打开标准输入输出文件（将传递给第一个用户进程），并挂载外部存储介质作为内核根文件系统（如果需要）。

free_initmem()函数将释放内核镜像中初始化段占用的内存，将其释放给伙伴系统，供系统使用。

kernel_init 线程最后根据命令行参数等选择第一个用户进程的可执行目标文件，并加载目标文件至线程用户空间，线程转而运行目标文件中代码，kernel_init 线程转变在第一个用户进程，内核启动最终完成。

在介绍 kernel_init()函数前，先看两个定义在/init/main.c 文件内的全局变量：

```
static char    *execute_command;          /*在外部存储介质中查找第一个用户进程的可执行目标文件*/
static char    *ramdisk_execute_command;  /*在 rootfs 根文件系统中查找的第一个用户进程目标文件*/
```

execute_command 指向保存"init=***"命令行参数的值，ramdisk_execute_command 指向保存"rdinit=***"命令行参数的值，这两个指针变量在处理命令行参数时设置。前者用于指定在外部介质根文件系统中查找第一个用户进程可执行目标文件的路径，后者用于指示在内核初始挂载的 rootfs 文件系统中查找第一个用户进程可执行目标文件的路径。

内核在文件系统初始化过程中将会挂载初始的 rootfs 文件系统作为根文件系统，这是一个基于 RAM

内存的文件系统（详见第 7 章），文件系统内容由用户设置。kernel_init 线程首先在 rootfs 文件系统中查找"rdinit=***"参数指示的目标文件或/init 文件是否存在，存在则加载运行，作为第一个用户进程，不再挂载外部存储介质文件系统。

如果不能在 rootfs 文件系统中运行"rdinit=***"参数指示的目标文件或/init 目标文件，kernel_init 线程将挂载外部存储介质（由命令行参数“root=***”指示）作为根文件系统，并查找由"init=***"参数指示的目标文件，如果查找到则加载运行，作为第一个用户进程。如果没有查找到"init=***"参数指示的目标文件，则查找内核默认的可执行目标文件并加载运行，作为第一个用户进程。

下面来看一下 kernel_init()函数的实现，代码如下（/init/main.c）：

```
static int __ref kernel_init(void *unused)
{
    int ret;

    kernel_init_freeable();      /*初始化各子系统，挂载外部根文件系统等，/init/main.c*/
    async_synchronize_full();    /*/kernel/async.c*/
    free_initmem();              /*释放初始化段占用内存，详见第 3 章，/arch/mips/mm/init.c*/
    mark_rodata_ro();
    system_state = SYSTEM_RUNNING;
    numa_default_policy();

    flush_delayed_fput();

    /*如果 rootfs 文件系统中存在"rdinit=***"指示的目标文件或/init 目标文件，则加载运行*/
    if (ramdisk_execute_command)          {
        ret = run_init_process(ramdisk_execute_command);    /*运行新进程，/init/main.c*/
        if (!ret)
            return 0;
        pr_err("Failed to execute %s (error %d)\n",ramdisk_execute_command, ret);
    }

    /*如果 ramdisk_execute_command 为 NULL，尝试运行"init=***"指示的可执行文件*/
    if (execute_command)
    {
        ret = run_init_process(execute_command);  /*运行新进程，目标文件位于外部根文件系统*/
        if (!ret)
            return 0;
        panic("Requested init %s failed (error %d).",execute_command, ret);
    }

    /*如果还是运行不成功，尝试运行默认的可执行目标文件，运行第一个存在的目标文件*/
    if (!try_to_run_init_process("/sbin/init") ||        /*/init/main.c*/
        !try_to_run_init_process("/etc/init") ||
        !try_to_run_init_process("/bin/init") ||
        !try_to_run_init_process("/bin/sh"))
```

```
        return 0;

    panic("No working init found.  Try passing init= option to kernel. "
                                  "See Linux Documentation/init.txt for guidance.");
}
```

kernel_init 线程内调用 **kernel_init_freeable()**函数完成内核各子系统的初始化、挂载外部存储介质作为内核根文件系统（如果需要）等工作，随后调用 free_initmem()函数释放内核镜像初始化段占用的内存，最后选择可执行目标文件运行第一个用户进程。

需要注意的是如果 run_init_process()或 try_to_run_init_process()函数执行成功，将会用目标文件内容覆盖 kernel_init 线程地址空间，kernel_init 线程原代码不再执行，也就是说这两个函数返回到目标文件代码中运行了，kernel_init 线程的代码了。全局字符指针数组 argv_init[]和 envp_init[]指示的命令行参数和环境变量将传递给第一个用户进程。

1 继续初始化

kernel_init 线程中调用 kernel_init_freeable()函数完成内核剩余的初始化工作，并按需挂载外部存储介质作为内核根文件系统。函数定义在/init/main.c 文件内，代码如下：

```
static noinline void __init kernel_init_freeable(void)
{

    wait_for_completion(&kthreadd_done);       /*等待 rest_init()函数创建 kthreadd 线程完成*/
    gfp_allowed_mask = __GFP_BITS_MASK;
    set_mems_allowed(node_states[N_MEMORY]);    /*可在所有内存结点中分配内存*/
    set_cpus_allowed_ptr(current, cpu_all_mask);
                                /*设置当前进程与所有 CPU 核的亲和性，/kernel/sched/core.c*/
    cad_pid = task_pid(current);
    smp_prepare_cpus(setup_max_cpus);  /*完成多核处理器准备工作，/arch/mips/kernel/smp.c*/
    do_pre_smp_initcalls();       /*调用 early_initcall(fn)注册的初始化函数，见下文，/init/main.c*/
    lockup_detector_init();
    smp_init();    /*多核处理器数据初始化，见第 5 章，/kernel/smp.c*/
    sched_init_smp();       /*多核处理器调度初始化，见第 5 章，/kernel/sched/core.c*/
    page_alloc_init_late();
            /*没有选择 DEFERRED_STRUCT_PAGE_INIT 为空操作，/include/linux/gfp.h*/

    do_basic_setup();          /*初始化各子系统，/init/main.c*/

     /*打开位于 rootfs 文件系统中的/dev/console 文件作为线程标准输入文件*/
    if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
          pr_err("Warning: unable to open an initial console.\n");

    (void) sys_dup(0);     /*标准输出文件，同/dev/console*/
    (void) sys_dup(0);       /*标准错误输出文件，同/dev/console*/

    if (!ramdisk_execute_command)     /*如果没有传递“rdinit=***”参数，则赋值为"/init"*/
```

```
            ramdisk_execute_command = "/init";

        if (sys_access((const char __user *) ramdisk_execute_command, 0) != 0)
        {       /*访问 ramdisk_execute_command 指向文件不成功则挂载外部介质作为根文件系统*/
            ramdisk_execute_command = NULL;   /*不从 rootfs 文件系统中运行第一个用户进程*/
            prepare_namespace();
              /*挂载外部文件系统作为根文件系统，详见第 7 章，/init/do_mounts.c*/
        }

        integrity_load_keys();
        load_default_modules();       /*加载块设备驱动 IO 调度器模块，/init/main.c*/
    }
```

kernel_init_freeable()函数的主要工作是初始化内核各子系统，并判断是否需要挂载外部存储介质作为根文件系统，如果需要则调用 **prepare_namespace()**函数挂载，不需要则不挂载，仍使用初始挂载的根文件系统。

需要挂载外部根文件系统的条件是：在初始 rootfs 根文件系统中不能打开（或不存在）“rdinit=***”参数指示的目标文件，也不能打开（或不存在）/init 目标文件（没有传递“rdinit=***”参数时）。

kernel_init_freeable()函数中还将打开初始 rootfs 文件系统中的**/dev/console** 设备文件作为线程（第一个用户进程）的标准输入、标准输出、标准错误输出文件，随后就可以向用户空间（终端）输出信息了。

kernel_init_freeable()函数中继续初始化内核各子系统的工作主要由 **do_basic_setup()**函数完成，函数定义在/init/main.c 文件内：

```
    static void __init do_basic_setup(void)
    {
        cpuset_init_smp();     /*设置 CPU 核位图等，/kernel/cpuset.c*/
        usermodehelper_init();
        shmem_init();          /*注册并挂载共享内存使用的 shmem 伪文件系统，/mm/shmem.c*/
        driver_init();         /*驱动模型初始化，见第 8 章*/
        init_irq_proc();       /*将 irq 信息导出到 proc 文件系统/proc/irq 目录下，/kernel/irq/proc.c*/
        do_ctors();
        usermodehelper_enable();
        do_initcalls();        /*调用各子系统、驱动程序注册的初始化函数，/init/main.c*/
        random_int_secret_init();
    }
```

这里我们主要关注一下 **do_initcalls()**函数，它完成各子系统、设备驱动程序等注册的初始化函数。

内核各子系统、驱动程序等需要在内核启动阶段调用的初始化函数，通常用下列宏注册：

```
#define   early_initcall(fn)            __define_initcall(fn, early)     /*/include/linux/init.h*/
#define   pure_initcall(fn)             __define_initcall(fn, 0)
#define   core_initcall(fn)             __define_initcall(fn, 1)
#define   core_initcall_sync(fn)        __define_initcall(fn, 1s)
#define   postcore_initcall(fn)          __define_initcall(fn, 2)
#define   postcore_initcall_sync(fn)     __define_initcall(fn, 2s)
#define   arch_initcall(fn)              __define_initcall(fn, 3)
```

```
#define   arch_initcall_sync(fn)        __define_initcall(fn, 3s)
#define   subsys_initcall(fn)           __define_initcall(fn, 4)
#define   subsys_initcall_sync(fn)      __define_initcall(fn, 4s)
#define   fs_initcall(fn)               __define_initcall(fn, 5)
#define   fs_initcall_sync(fn)          __define_initcall(fn, 5s)
#define   rootfs_initcall(fn)           __define_initcall(fn, rootfs)
#define   device_initcall(fn)           __define_initcall(fn, 6)
#define   device_initcall_sync(fn)      __define_initcall(fn, 6s)
#define   late_initcall(fn)             __define_initcall(fn, 7)
#define   late_initcall_sync(fn)        __define_initcall(fn, 7s)
```

以上宏内 fn 参数表示初始化函数指针（函数入口地址），函数类型如下：

```
typedef int (*initcall_t)(void);      /*函数不能带参数，/include/linux/init.h*/
```

__define_initcall(fn, id)宏定义在/include/linux/init.h 头文件：

```
#define __define_initcall(fn, id) \
    static initcall_t __initcall_##fn##id __used \     /*initcall_t 为函数指针类型*/
    __attribute__((__section__(".initcall" #id ".init"))) = fn; \
    LTO_REFERENCE_INITCALL(__initcall_##fn##id)
```

__define_initcall(fn, id)宏的效果是创建 **initcall_t** 类型的函数指针变量并赋值 fn，fn 为注册的初始化函数入口地址，并将变量链接到指定的内核目标文件**".initcall" #id ".init"**段内，参数 id 值嵌入到段名中，表示段的等级。

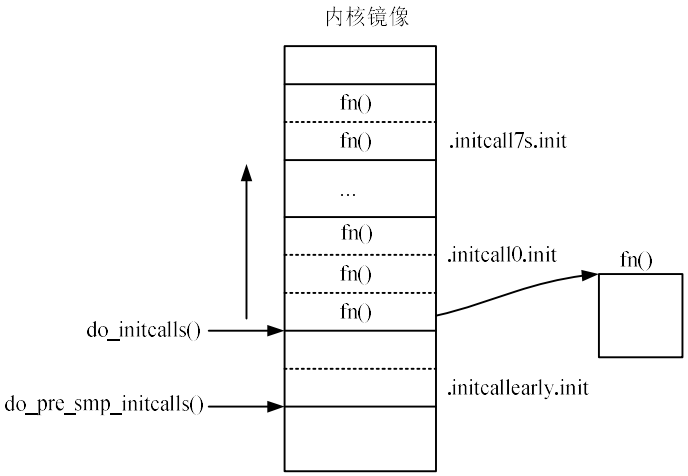
由__define_initcall()宏定义的函数指针变量都链接到同一个段内，参数 id 又将此段划分成小段，id 值表示小段在**".initcall*.init"**段内的排序，值越小的段排在越前面。例如：pure_initcall(fn)宏定义的变量排在 core_initcall(fn)宏定义的变量前面。

以下是段名在链接文件中的表示（/include/asm-generic/vmlinux.lds.h），这些段链接到初始化段：

```
#define INIT_CALLS                                  \
        VMLINUX_SYMBOL(__initcall_start) = .;       \
        *(.initcallearly.init)                      \
        INIT_CALLS_LEVEL(0)                         \     /*等级 0*/
        INIT_CALLS_LEVEL(1)                         \
        INIT_CALLS_LEVEL(2)                         \
        INIT_CALLS_LEVEL(3)                         \
        INIT_CALLS_LEVEL(4)                         \
        INIT_CALLS_LEVEL(5)                         \
        INIT_CALLS_LEVEL(rootfs)                    \
        INIT_CALLS_LEVEL(6)                         \
        INIT_CALLS_LEVEL(7)                         \
        VMLINUX_SYMBOL(__initcall_end) = .;
```

do_initcalls()函数执行的工作如下图所示，从等级 0 开始扫描各段，依次调用执行 **initcall_t** 变量指示的函数，完成内核代码中注册的初始化函数。

early_initcall(fn)宏注册的初始化函数将在调用 do_basic_setup()函数前由 **do_pre_smp_initcalls()**函数处理，处理方式与 **do_initcalls()**函数相同，只不过它只处理**.initcallearly.init** 段中注册的函数。



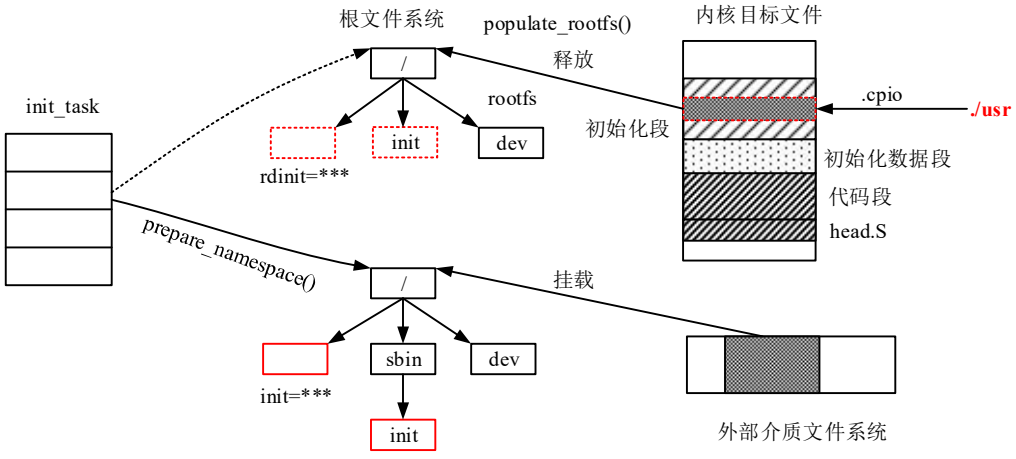
2 运行第一个用户进程

前面大致介绍了 kernel_init 线程在完成初始化工作后转为第一个用户进程的流程。这里有必要再梳理一下，以使读者对外部根文件系统的挂载和运行第一个用户进程有更清晰的认识，更详细的内容见第 7 章和第 14 章。

启动函数 start_kernel()中将调用 vfs_caches_init()函数挂载内核初始的根文件系统，这是一个基于内存盘的 ramfs 或 tmpfs 文件系统，统称为 rootfs，初始状态此文件系统内容初始为空。

在配置内核时，如果选择了 **BLK_DEV_INITRD** 配置选项（且不选择 BLK_DEV_RAM 选项），编译内核时将会将源文件**./usr** 目录下的内容编译为**.cpio** 格式的目标文件，并将此目标文件链接到内核目标文件的初始化段。

在初始化各子系统的 do_initcalls()函数中将会调用初始化函数 **populate_rootfs()**将内核镜像初始化段中.cpio 格式目标文件的内容释放到初始 rootfs 类型的内核根文件系统中，如下图所示。



在前面介绍的 kernel_init_freeable()函数中将会判断在初始根文件系统中能否打开"**rdinit=***"参数值指示的目标文件，或“**/init**”目标文件（没有传递"rdinit=***"参数时），如果可以打开将不挂载外部介质作为内核根文件系统，kerne_init 线程运行此目标文件作为第一个用户进程。

如果不能打开以上两个文件中的任意一个，kernel_init_freeable()函数将调用 **prepare_namespace()**函数挂载**"root=***"**命令行参数（参数值为块设备文件路径名）指示的块设备中作为内核新的根文件系统，如

上图所示。随后，判断如果传递了**"init=***"**命令行参数（参数值为第一个用户进程目标文件路径名），则运行参数值指示的目标文件作为第一个用户进程，目标文件位于刚挂载的外部介质文件系统中。如果没有传递**"init=***"**命令行参数，内核将依次在挂载的外部介质文件系统中查找以下目标文件，以先到者为先，运行第一个用户进程，内核启动大功告成。

- **/sbin/init**
- **/etc/init**
- **/bin/init**
- **/bin/sh**

2.7 小结

本章开头简要介绍了内核源代码链接文件，以初步了解内核目标文件结构，以及内核入口函数。内核可执行目标文件由引导加载程序加载到内存，并使处理器程序指针跳转至内核入口地址，开始执行内核代码。引导加载程序可通过命令行参数（和环境变量）向内核传递信息。

内核入口函数定义在体系结构相关的 head.S 文件内，这是一个由汇编代码编写的源文件，此文件汇编代码最后将跳转到体系结构无关的内核启动函数 start_kernel()。

启动函数 start_kernel()主要完成体系结构相关的初始化，调用内核各子系统定义的初始化函数，创建内核线程，加载外部根文件系统等工作，最后加载第一个用户进程可执行目标文件，运行用户进程，内核启动完成。

本章主要介绍了 start_kernel()函数中调用的体系结构相关的 setup_arch()函数的实现，命令行参数的处理，以及最后调用的 rest_init()函数的实现。启动函数中调用的各子系统初始化函数是研究内核各子系统的起点，在后面介绍各子系统时，还将详细介绍各子系统的初始化函数。