第 17 章

Linux 2.6.10@ARM 启动过程

17.1 Linux 2.6.10 中与 ARM 处理器平台硬件相关的结构和全局变量

Linux 的底层操作涉及对处理器平台硬件的操作，所以如果想彻底了解 Linux 内核的实现内幕，就一定要在了解处理器硬件平台的基础上了解 Linux 为操作处理器平台硬件而封装的一些数据结构、操作函数组结构以及全局变量等，所以在介绍启动过程之前先集中介绍一下这些处理器平台硬件相关的结构和全局变量。

17.1.1 相关数据结构

■ struct processor 处理器硬件操作函数组结构

void (*_data_abort)(unsigned long pc)：获取数据异常终止的地址和标志字函数指针。

void (*_proc_init)(void)：处理器初始化函数指针，基本都是什么都不做就返回。

void (*_proc_fin)(void)：禁止处理器某些功能函数指针，禁止 IRQ 与 FIQ，进入特权模式，禁止一些提高性能的功能部件如 cache、Write Buffer 等返回，armv6 中直接返回。

volatile void (*reset)(unsigned long addr)：软件复位操作函数指针，关闭 cache 和 MMU 然后跳转到复位地址 addr。

int (*_do_idle)(void)：使处理器进入空闲节能状态，即等待激活状态。

void (*dcache_clean_area)(void *addr, int size)：清空指定虚拟地址处的数据 cache 块，但不刷新。

void (*switch_mm)(unsigned long pgd_phys, struct mm_struct *mm)：在 pgd_phys 地址上重建页表，用于任务切换时保存重装旧线程状态。

void (*set_pte)(pte_t *ptep, pte_t pte)：设置 PTE 函数指针。

■ struct proc_info_list 处理器信息列表结构

unsigned int cpu_val：处理器架构的 ID 号。

unsigned int cpu_mask：处理器架构 ID 号对应屏蔽字。

unsigned long __cpu_mmu_flags：它是内存管理单元的第一级描述符控制字段，从表 17-1 中看出，Linux 为各种常用 ARM 处理器预先设定的该字段的 bit[11:0]值等于 2，也就是说 ARM Linux 中是预先设定为使用分段方式管理内存的，这个成员变量在 arch/arm/kernel/head.s 文件中用到。

unsigned long __cpu_flush：这里存着一条跳转到具体架构设置函数的指令。

const char *arch_name：体系架构名字字段；

常用 ARM 处理器的 ID 号、对应屏蔽字、MMU 标志字和体系架构名字、CPU 名字如表 17-1 所示：

表 17-1 常用 ARM 处理器信息表

处理器型号	ID 号	屏蔽字	MMU 标志字	体系架构名	CPU 名字
ARM6	0x41560600	0xfffffff0	0x00000c1e	armv3	ARM6
ARM610	0x41560610	0xfffffff0	0x00000c1e	armv3	ARM610
ARM7	0x41007000	0xffffff00	0x00000c1e	armv3	ARM7
ARM710	0x41007100	0xfff8ff00	0x00000c1e	armv3	ARM710
ARM710T	0x41807100	0xffffff00	0x00000c1e	armv4t	ARM710T
ARM720	0x41807200	0xffffff00	0x00000c1e	armv4t	ARM720T
ARM920	0x41009200	0xff00fff0	0x00000c1e	armv4t	ARM920T
ARM922	0x41009220	0xff00fff0	0x00000c1e	armv4t	ARM922T
ARM915	0x54029150	0xfffffff0	0x00000c12	armv4t	ARM925T
ARM925	0x54029250	0xfffffff0	0x00000c12	armv4t	ARM925T
ARM926	0x41069260	0xff0ffff0	0x00000c1e	armv5tej	ARM926EJ-S
ARM1020	0x4104a200	0xff0ffff0	0x00000c02	armv5t	ARM1020
ARM1020e	0x4105a200	0xff0ffff0	0x00000c12	armv5te	ARM1020E
ARM1022	0x4105a220	0xff0ffff0	0x00000c12	armv5te	arm1022

续表

处理器型号	ID 号	屏蔽字	MMU 标志字	体系架构名	CPU 名字
ARM1026	0x4106a260	0xff0ffff0	0x00000c12	armv5tej	ARM1026EJ-S
SA110	0x4401a100	0xfffffff0	0x00000c0e	armv4	StrongARM-110
SA1100	0x4401a110	0xfffffff0	0x00000c0e	armv4	StrongARM-1100
SA1110	0x6901b110	0xfffffff0	0x00000c0e	armv4	StrongARM-1110
ARMv6	0x0007b000	0x0007f000	0x00000c0e	armv6	Some Random V6 Processor
80200	0x69052000	0xfffffff0	0x00000c0e	armv5te	XScale-80200
8032x	0x69052420	0xfffff5e0	0x00000c0e	armv5te	XScale-IOP8032x Family
8033x	0x69054090	0xfffffb0	0x00000c0e	armv5te	XScale-IOP8033x Family
Pxa250	0x69052100	0xfffff7f0	0x00000c0e	armv5te	XScale-PXA250
pxa210	0x69052120	0xfffff3f0	0x00000c0e	armv5te	XScale-PXA210
ixp2400	0x690541a0	0xfffffff0	0x00000c0e	armv5te	XScale-IXP2400
ixp2800	0x69052100	0xfffffff0	0x00000c0e	armv5te	XScale-IXP2800
ixp42x	0x690541c0	0xffffffc0	0x00000c0e	armv5te	XScale-IXP42x Family
ixp46x	0x69054200	0xffffff00	0x00000c0e	armv5te	XScale-IXP46x Family
ixp255	0x69052d00	0xfffffff0	0x00000c0e	armv5te	XScale-PXA255
ixp270	0x69054110	0xfffffff0	0x00000c0e	armv5te	XScale-PXA270

const char *elf_name：体系架构版本号，如 v3、v4、v5、v6 等。

unsigned int elf_hwcap：处理器硬件功能标志字，一位代表一种功能，在头文件 include/asm-arm/procinfo.h 中有如下定义：

```
#define HWCAP_SWP       1 ；支持 SWP 和 SWPB 信号量指令，用于进程同步，armv2 以上版本都实现；
#define HWCAP_HALF      2 ；支持半字读写指令，armv3 上未实现；
#define HWCAP_THUMB 4 ；支持 Thumb 指令，StrongArm 和 armv3 上未实现；
#define HWCAP_26BIT     8 ；支持 26 位地址空间，StrongArm 和 armv3 上实现；
#define HWCAP_FAST_MULT 16 ；支持长乘法和长乘加法指令，arm926、arm1026，StrongArm、xscale、armv6 处理器上实现；
#define HWCAP_FPA   32 ；支持 FPA 型浮点单元和指令，还没有 ARM 处理器实现此功能；
#define HWCAP_VFP   64 ；支持 VFP 型浮点单元和指令，只能用 armv6 处理器实现；
#define HWCAP_EDSP  128；增加附加指令，增强了 DSP 算法处理能力，arm1020e、arm1022、arm926、arm1026、xscale、armv6 处理器上实现；
#define HWCAP_JAVA  256；Java 硬件加速功能 Jazelle。在 arm926、arm1026、armv6 等处理器上实现；
```

const char *cpu_name：CPU 名字；

struct processor *proc：处理器操作函数组结构指针；

struct cpu_tlb_fns　*tlb：使用户和内核地址空间对应 TLB 无效函数；

struct cpu_user_fns　*user：用户空间清空、复制函数；

struct cpu_cache_fns*cache：cache 操作函数；

■ struct cpu_tlb_fns 处理器 TLB 操作函数组结构

void (*flush_user_range)(unsigned long, unsigned long, struct vm_area_struct *)：使指定用户地址空间对应的 TLB 无效。

void (*flush_kern_range)(unsigned long, unsigned long)：使指定内核地址空间对应的 TLB 无效。

unsigned long tlb_flags：TLB 标志字，每一位定义了所有 ARM 处理器的 TLB 的某种属性或某种操作，每种 ARM 处理器只适用于其中的某些位，该标志字不与处理器中的寄存器相关联。标志字中各位的宏定义如下：宏名字中的 PAGE 表示单个 TLB；FULL 表示全部 TLB；U 表示统一 cache 中的 TLB；I 表示指令 cache 中的 TLB；D 表示数据 cache 中的 TLB；WB 表示写缓冲。该字段每一位的宏定义在 include/asm-arm/tlbflush.h 文件中，含义说明如下：

```
#define TLB_V3_PAGE     (1 << 0)；ARMv3 处理器中使用
#define TLB_V4_U_PAGE   (1 << 1)；使统一 cache 中的单个 TLB 条目无效
#define TLB_V4_D_PAGE   (1 << 2)；使数据 cache 中的单个 TLB 条目无效
#define TLB_V4_I_PAGE   (1 << 3)；使指令 cache 中的单个 TLB 条目无效
#define TLB_V6_U_PAGE   (1 << 4)；使统一 cache 中的单个 TLB 条目无效
#define TLB_V6_D_PAGE   (1 << 5)；使数据 cache 中的单个 TLB 条目无效
#define TLB_V6_I_PAGE   (1 << 6)；使指令 cache 中的单个 TLB 条目无效
#define TLB_V3_FULL     (1 << 8)
#define TLB_V4_U_FULL   (1 << 9)；使整个 cache 无效
#define TLB_V4_D_FULL   (1 << 10)；使整个数据 cache 无效
#define TLB_V4_I_FULL   (1 << 11)；使整个指令 cache 无效
#define TLB_V6_U_FULL   (1 << 12)；使整个 cache 无效
#define TLB_V6_D_FULL   (1 << 13)；使整个数据 cache 无效
#define TLB_V6_I_FULL   (1 << 14)；使整个指令 cache 无效
#define TLB_V6_U_ASID   (1 << 16)；用于 ARMv6 以上版本处理器
#define TLB_V6_D_ASID   (1 << 17)；用于 ARMv6 以上版本处理器
#define TLB_V6_I_ASID   (1 << 18)；用于 ARMv6 以上版本处理器
#define TLB_DCLEAN      (1 << 30)；定义通过清空数据 cache 指令来清空 TLB
#define TLB_WB          (1 << 31)；定义写缓存
```

■ struct cpu_user_fns 用户空间处理器相关操作函数组结构

void (*cpu_clear_user_page)(void *p, unsigned long user)：清空用户页及其 cache，并使对应 cache 无效。

void (*cpu_copy_user_page)(void *to, const void *from, unsigned long user)：复制用户空间页，注意 cache。

■ struct cpu_cache_fns 处理器 cache 操作函数组结构

void (*flush_kern_all)(void)：使整个系统 cache 无效。

void (*flush_user_all)(void)：和 flush_kern_all 效果一样，实际是同一个函数体。

void (*flush_user_range)(unsigned long, unsigned long, unsigned int)：依处理器架构而异，使范围内无效。

void (*coherent_kern_range)(unsigned long, unsigned long)：清空数据 cache 块、预取缓冲区、写缓冲区。

void (*coherent_user_range)(unsigned long, unsigned long)：与上一函数相同。

void (*flush_kern_dcache_page)(void *)：清空并使指定区域 cache 无效。

void (*dma_inv_range)(unsigned long, unsigned long)：清空并使指定区域 cache 无效。

void (*dma_clean_range)(unsigned long, unsigned long)：清空 cache，写缓冲。

void (*dma_flush_range)(unsigned long, unsigned long)：清空并使数据 cache 无效，清空写缓冲。

■ struct machine_desc 处理器平台描述结构

unsigned int　nr：处理器平台（板子）代号；目前 Linux 中支持的各种 ARM 处理器硬件平台有 600 多种，它们的编号在 include/asm-arm/mach-types.h 文件中定义；

unsigned int　phys_ram：系统内存物理基地址。

unsigned int　phys_io：处理器内部 I/O 寄存器（I/O 空间）物理基地址。

unsigned int　io_pg_offst：I/O 空间页表条目偏移量。

const char *name：硬件系统（板子）名称。

unsigned int　param_offset：系统参数偏移量是相对于谁的偏移量呢？它是相对于系统内存物理基地址的偏移量，一般为系统内存物理基地址加上 0x100。这里保存了 bootloader 中设

置好并传递给 Linux 的系统参数，这里设置的参数可以是 struct param_stuct 格式或连续存放的 struct tag 格式，在 linux2.6.10 中，如果 bootloader 传进来的参数是 struct param_struct 格式的，会将其转换成连续存放的 struct tag 格式。

unsigned int　video_start：显卡内存起始物理地址，嵌入式系统中很少使用该设置。

unsigned int　video_end：显卡内存结束物理地址，嵌入式系统中很少使用该设置。

unsigned int　reserve_lp0 :1：保留并口 0 指示位，1 表示系统有并口 0，嵌入式无此设置。

unsigned int　reserve_lp1 :1：保留并口 1 指示位，1 表示系统有并口 1，嵌入式无此设置。

unsigned int　reserve_lp2 :1：保留并口 2 指示位，1 表示系统有并口 2，嵌入式无此设置。

unsigned int　soft_reboot :1：软件重启位，很少设置。

void (*fixup)(struct machine_desc *, struct tag *, char **, struct meminfo *)：做一些前期补充初始化，设置系统参数，很少用，对于不同系统该函数处理的事情不一而足。

void (*map_io)(void)：I/O 映射操作函数。

void (*init_irq)(void)：IRQ 中断初始化函数。

struct sys_timer*timer：系统硬件定时器描述结构。

void (*init_machine)(void)：系统初始化函数。

■ struct new_utsname 新 uts 名字描述结构

char sysname[65]：操作系统名称。

char nodename[65]：节点名，初始为“none”，后由 sethostname()函数设置。

char release[65]：Linux 发行版本名。

char version[65]：Linux 最后的编译时间。

char machine[65]：处理器架构名，有时最后也附上“l”或“b”表示 little 或 big endian。

char domainname[65]：域名，初始为“none”，后由 setdomainname()函数设置。

17.1.2 相关全局变量

■ struct proc_info_list __proc_info_begin, __proc_info_end

在 arch\arm\kernel\vmlinux.lds.s 中定义，__proc_info_begin 和__proc_info_end 之间的区域设置为*(.proc.info)区。*(.proc.info)区中保存了处理器的信息，可以是单个结构变量，也可以是结构数组。.proc.info 区中的内容根据不同的 ARM 版本处理器其内容也不一样，它的定义在

arch\arm\mm\proc-armxxx.s 文件中，其中 armxxx 表示的 ARM 架构的版本号，如 arm6_7、arm920、arm1020、arm720 等。

■ cpu_tlb

struct cpu_tlb_fns 结构，处理器 tlb 操作函数组。

■ cpu_user

struct cpu_user_fns 结构，用户空间处理器相关操作函数组。

■ cpu_cache

struct cpu_cache_fns 结构，处理器 cache 操作函数组。

■ extern struct machine_desc, __arch_info_begin __arch_info_end

在 arch\arm\kernel\vmlinux.lds.s 中定义，__arch_info_begin 和__arch_info_end 之间的区域被设置为*(.arch.info)区。这里存的是一个或多个 machine_desc 型结构变量，而这些全局变量在每一个具体型号 CPU 对应的文件中定义，而且可能一个文件中定义若干个该结构变量，是用 MACHINE_START(_type,_name)宏来定义这种结构变量的。

■ system_utsname

struct new_utsname 结构：系统 UTS 名字描述结构。声明时初始化如下：

```
.sysname = UTS_SYSNAME
     #define UTS_SYSNAME "Linux"
.nodename = UTS_NODENAME
     #define UTS_NODENAME "(none)"
.release = UTS_RELEASE  ;在编译时生成的文件 include/linux/version.h 中定义如下
     #define UTS_RELEASE "2.6.10_mvl401-440ep_yosemite"
.version = UTS_VERSION  ;在编译时生成的文件 include/linux/compile.h 中定义如下
     #define UTS_VERSION "#1 Sat Feb 17 15:03:11 CST 2007"
.machine = UTS_MACHINE  ;在编译时生成的文件 include/linux/compile.h 中定义如下
     #define UTS_MACHINE "ppc"
.domainname = UTS_DOMAINNAME
     #define UTS_DOMAINNAME "(none)"
```

■ elf_platform[ELF_PLATFORM_SIZE（=8）]

char 数据类型；系统处理器体系架构版本号

17.2 Linux 汇编代码入口

Linux 汇编代码的入口是 Linux 内核的起始地址，也就是 Bootloader 启动 Linux 时所跳转到的地址。内核一般都是被压缩后存放到 flash 中的，Bootloader 启动 Linux 时先将内核解压缩到内核编译连接到的起始虚拟地址对应的物理地址上，然后跳转到该物理地址运行。

ARM 系统中 Linux 的汇编代码入口在 arch\arm\kernel\head.s 文件的开始处，如下所示：

```
__INIT
.type   stext, #function
ENTRY（stext）    ；这是 Linux 汇编程序的入口地址
```

17.3 Linux 汇编入口处 CPU 的状态

Linux 汇编代码入口处 CPU 的状态如下：

MMU 关闭；

D-cache 关闭；

I-cache 关闭或打开；

r0 等于 0；

r1 保存硬件平台的编号。

Linux 内核支持的各种 ARM 处理器平台的 ID 号都定义在 arch\arm\tools\mach-types 文件中，编译后会生成该文件对应的 include\asm-arm\mach-types.h 头文件。

由上面 Linux 汇编代码入口处处理器的状态可知，从 Linux 汇编代码入口处到 MMU 被使能之前，处理器一直工作在物理地址模式，系统中使用的都是物理地址，直到 MMU 被打开，系统才开始使用虚拟地址。

在 ARM 中，Linux 一般都被编译连接到虚拟地址 0xC0008000 处，所以调用 Linux 的入口时应该跳转到0xC0008000对应的物理地址__pa(0xC0008000)处，如果直接跳转到0xC0008000，系统将由于找不到程序入口而不能正常启动。因此这就要求从 Linux 汇编代码入口处到 MMU 被使能之前的代码都是与位置无关的，不能有绝对跳转指令，只能使用相对地址（如 adr 指令等），程序只能在一段比较小的地址范围内运行。一旦 MMU 被使能，系统将使用编译连接时生成的符号表对应的绝对地址来访问和调用全局数据变量和函数。

17.4 start_kernel()函数之前的汇编代码执行过程

说明：第 17.4 节~第 17.6 节的内容是我第一遍阅读 Linux2.6.10 源代码时的笔记和注释，这里只是把这些内容罗列到了一起，省去了源代码本身，仅供读者了解 Linux 内核中主要函数的大致功能时使用，如果想详细了解每个函数的具体功能实现，请务必对照 linux2.6.10 源代码阅读这些内容。这三节内容主要介绍 Linux 内核的启动流程，如果要了解内核的详细原理可以略过这三节，直接阅读后续章节。

保存 bootloader 传进来的系统参数的地址指针 r0 到 r12 中。

禁止 IRQ 和 FIQ 中断，将处理器置成超级用户（Supervisor）模式。

执行__lookup_processor_type()汇编函数：读出协处理器 CP15 的 C0,0 标识符寄存器的值保存到 r9 中，将其一一与预定义在*（.proc.info）区中的 Linux 系统支持的各种 ARM 处理器对应的 struct proc_info_list 结构中的处理器 ID 号掩码 cpu_mask 相与，然后与同样预定义*（.proc.info）区中的处理器 ID 号 cpu_val 进行比较，如果找到了匹配的处理器 ID 号，把该处理器对应的处理器 MMU 标志保存到 r8 中，r10 则保存了该处理器对应的 struct proc_info_list 结构的起始地址；如果没有找到匹配的处理器 ID 号，则系统死循环。

执行__lookup_architecture_type()汇编函数：一一读出预定义在*（.arch.info）区中编译配置时确定的 Linux 系统所支持的各种 ARM 硬件系统（板子）对应的硬件平台描述结构 struct machine_desc 变量中的硬件平台代号成员变量 nr，将 nr 与从 bootloader 探测到并传进来的保存在 r1 中的硬件平台代号进行比较，如果找到了匹配的硬件平台代号，将该硬件平台对应的预定义好的系统物理内存基地址 phys_ram、系统外设物理基地址 phys_io、I/O 页偏移量 io_pg_offset 分别保存到 r5、r6、r7 中；如果没有找到匹配的硬件平台代号，则系统死循环。

执行__create_page_tables()汇编函数：将内核的起始地址 stext=TEXTADDR=0xc0008000 减去 0x4000 即 0xc0004000 对应的物理地址 0x00004000 作为处理器 MMU 虚拟地址到物理地址映射表（一级页表）的基地址，把基址 0x00004000 保存在 r4 中，r4 对应的虚拟地址就是另外一个全局变量 swapper_pg_dir=（TEXTADDR - 0x4000），r4 会被赋给处理器的控制协处理器 CP15 的寄存器 C2 中，由此也看出 C2 中的 1 级页表基地址是物理地址而不是虚拟地址。先将从 0x00004000 开始的 16KB 内存（一级页表空间）即内核前面的 16KB 内存空间全部清 0（否则可能出现误映射），再把当前执行指令所在的内存段（1 段=1MB）即系统内存开始的第 1MB 物理地址当作虚拟地址映射到本身物理地址上，也就是说系统物理内存的开始的 1MB 物理地址与虚拟地址相同，然后将内核起始虚拟地址 TEXTADDR 所在内存地址段开始的连续 4 段虚拟地址空间映射到系统物理内存的开始 4MB 空间上，最后将系统的虚拟内存起始地址 VIRT_OFFSET=PAGE_OFFSET=0xc0000000 映射到保存在 r5 中的系统内存物理基地址的第

1MB 上，只映射 1 段=1MB 空间（到此为止映射了 6 段虚拟地址到系统物理内存开始的 4 段空间上，这里有重叠：物理内存的第一段映射到自己地址上；内核的第一段虚拟地址映射到物理内存的第一段上；系统内存虚拟地址的第一段映射到物理内存的第一段上，这三个虚拟地址映射到了同一段物理内存上。内核的第 2、3、4 段映射到物理内存的第 2、3、4 段）。如果系统定义了 CONFIG_XIP_KERNEL（XIP 是 eXecute-In-Place 的缩写）即代码就在保存的地方（flash，rom）中运行，那么再映射 3MB 内存。如果支持调试，再做调试相关配置。这样事实上只映射了最开始的 1MB 内存空间到物理内存。

从 r10 所指的当前处理器对应的 struct proc_info_list 结构中获取__cpu_flush 成员变量的值给 PC，__cpu_flush 是一条跳转到处理器设置函数__xxx_setup()的绝对跳转指令，以 ixp_425 处理器为例，其__cpu_flush 就是 b __xscale_setup 指令，所以这里要执行的就是__xscale_setup() 汇编函数，它做的事情是：禁止 IRQ 和 FIQ 中断，使指令和数据 caches 都无效，清空写缓存 writer buffer，使指令和数据 TLBs 都无效，把上面已设置好映射关系的新页表的基地址写入到控制协处理器 CP15 的 C2 寄存器中。设置 ARM 处理器的 16 个存储访问域的控制权限，域 3~15 没有访问权限，域 0~1 为管理者权限，不会产生访问失效，域 2 为客户类型访问权限，将根据 MMU 地址转换条目中的 AP 设置和 CP15 的 C1 控制寄存器中的 R 和 S 位来确定域 2 的访问权限，ixp425 的 struct proc_info_list 结构中 MMU 的预定义标志字为 0xc0e，即使用 D0（管理者权限，各种工作模式都可以任意访问），AP 位为 0b11（也表示各种特权模式和用户模式都可以读写），C 位为 1（cachable），B 位为 1（bufferable），把设置好了各域权限的值写入 CP15 的 C3 寄存器中。读取 CP15 的控制寄存器 C1 到 r0 中，清除 r0 的 ROM 保护位 bit9（R）和地址对齐检查位 bit1（A），设置 bit0（M）和 bit2（C），使能 MMU 和 cache，设置 bit8（S）、bit11（Z）、bit12（I）和 bit13（V），使能系统保护、跳转预测、指令 cache 和高端中断向量表 0xffff0000~0xffff001c。

执行__turn_mmu_on 汇编函数：查看系统是否要求地址对齐检查（开关宏为 CONFIG_ALIGNMENT_TRAP）。如果是，再将上面修改设置好的 r0 的 bit1 设置为 1，使能地址对齐检查功能，然后将 r0 写入到 CP15 的 C1 寄存器中，使能 MMU 和上面配置的功能，最后将 CP15 的 C0,0 即主标识符寄存器的值读到 r3 中，与 r9 中的值相同。

执行__mmap_switched()汇编函数：将 init_thread_union+8192 保存到栈指针寄存器 sp 中，也就是在系统正式创建第一个任务前，将 init_thread_union.init_thread_info 作为系统启动过程的线程信息结构变量。如果__data_loc 与__data_start 地址不相等，那么将__data_loc 处的数据复制到__data_start 处，将__bss_start 到_end 之间的内存全部清 0，将前面保存在 r9 中的的标识符赋给全局变量 processor_id，将由 bootloader 传进来保存在 r1 中的硬件平台代号赋给全局变量__machine_arch_type，将保存在 r0 中的 CP15 的 C1 控制寄存器的值的 bit1 清 0 后保存到 r2 中，再将 r0 和 r2 分别存储到地址 cr_alignment 和 cr_no_alignment 处，即 cr_alignment 全局变量保存了使能地址对齐检查的系统控制寄存器的值，而 cr_no_alignment 全局变量保存了禁止地址对

齐检查的系统控制寄存器的值，最后调用 start_kernel()函数。

17.5 start_kernel()函数中调用的函数介绍

该节内容请务必对照 Linux2.6.10 源代码阅读。

start_kernel()函数完成操作系统各种初始化并打印相关信息，然后创建 init()内核进程（1号进程），最后进入 idle()状态。

事实上，init()进程之前运行的程序包括汇编代码和 start_kernel()函数都属于 0 号进程，即空闲（idle）进程或启动进程，也叫交换（swapper）进程；交换进程在系统进入 init()进程后挂起，当系统中没有可运行的进程时被重新唤醒。

start_kernel()函数的执行过程按其调用的函数先后顺序详细说明如下。

说明：start_kernel()函数整个执行的过程很长很复杂，下面有几个深层子函数的解析，读者可以通过阅读源代码自己去完善。大家阅读源代码一定要有足够的耐心，读完并理解这些源代码之后，你一定能享受到无穷的乐趣和成就感。

17.5.1 lock_kernel()函数

如果内核被配置成支持抢占，那么在这里禁止抢占，即将 0 号进程的 init_thread_info.preempt_count 加 1；如果配置成不支持抢占，那么将内核全局自旋锁 kernel_flag 上锁，然后将 0 号进程的 init_task.lock_depth 加 1。

17.5.2 page_address_init()函数

该函数初始化高端内存（High Memory）线性地址空间中永久映射相关的全局变量（详见第 18 章）。

所以在不支持高端内存即没有配置 CONFIG_HIGHMEM（在配置内核时设置）选项的系统中，该函数是个空函数，什么也不做，在 Linux2.6.10 内核的 ARM 处理器部分，没有支持高端内存相关代码。

在支持高端内存的系统中，page_address_init()函数初始化高端内存页表池链表 struct list_head page_address_pool，将内存页地址映射表 struct page_address_map page_address_maps[LAST_PKMAP]中的每个页地址映射结构 page_address_maps[n]都通过其 list 成员加入到链表 page_address_pool 中。初始化内存页地址槽结构数组 static struct page_address_slot page_address_htable[1<<PA_HASH_ORDER（=7）]中每个结构变量的链表 lh 和自旋锁 lock，初始化高端内存池全局自旋锁 pool_lock。

17.5.3 printk(linux_banner)

将 linux_banner 字符串的内容打印到 log_buf 缓冲区中，此时还没有初始化控制台（console），所以不会打印到控制台上。

linux_banner 字符串的内容为"Linux version " UTS_RELEASE " (" LINUX_COMPILE_BY "@" LINUX_COMPILE_HOST ") (" LINUX_COMPILER ") " UTS_VERSION "\n"。

其中：

UTS_RELEASE 表示 linux 的发行版本号，在 include\linux\version.h 中定义；

LINUX_COMPILE_BY 表示编译主机上编译时的用户名；

LINUX_COMPILE_HOST 表示编译主机名字；

LINUX_COMPILER 表示编译器的版本号；

UTS_VERSION 表示该目标文件编译生成的时间。

以上 4 个宏都在 include\linux\compile.h 中定义，include\linux\version.h 和 include\linux\compile.h 这两个文件都是编译内核过程中新生成的文件。

17.5.4 setup_arch(&command_line)函数

该函数根据处理器、硬件平台具体型号设置系统。解析 Linux 系统命令行，设置 0 号进程（swapper 进程）的内存描述结构 init_mm，系统内存管理初始化，统计并注册系统各种资源，其他杂项初始化。

具体执行过程说明如下。

根据汇编代码中获取的 ARM 处理器的主 ID 寄存器的值 processor_id，从.proc.info 中找到该处理器对应的处理器信息结构 struct proc_info_list 型变量，把它的处理器名字 cpu_name 成员变量保存到全局变量 cpu_name 中；把它的处理器处理函数组结构指针 struct processor *proc 所指结构保存到全局结构变量 processor 中；把它的 TLB 操作函数组结构指针 struct cpu_tlb_fns * tlb 所指结构保存到全局结构变量 cpu_tlb 中；将它的处理器用户空间操作函数组结构指针 struct cpu_user_fns *user 所指结构保存到全局结构变量 cpu_user 中；将它的 cache 操作函数组结构指针 struct cpu_cache_fns *cache 所指结构保存到全局结构变量 cpu_cache 中。打印处理器信息，读出并解析处理器的 cache 标识寄存器，打印系统的 cache 信息，包括 cache 类型、组织形式、cache 容量、cache 块大小、cache 的相连特性。把它的处理器架构名字字符串 arch_name 加“b”（big endian）或“l”（little endian）字符传给 system_utsname.machine；把它的体系架构版本号字符串 elf_name 加“b”（big endian）或“l”（little endian）字符保存

到 elf_platform 中；把它的处理器硬件能力标志字 elf_hwcap 保存到全局变量 elf_hwcap 中。调用处理器操作函数组结构变量 processor 的处理器初始化成员函数_proc_init()对处理器进行初始化（事实上什么也没做）。

根据宏 machine_arch_type 定义的当前系统的硬件平台编号从.arch.info 中找到其对应的平台描述结构 struct machine_desc 型变量，并传给局部变量 mdesc。打印硬件平台的名字字符串，并把它保存到全局字符串变量 machine_name 中。如果 mdesc 的软件重启动标志 soft_reboot 不等于 0，将全局字符变量 reboot_mode 设置为“s”，这个变量被初始化为“h”即硬件重启动；如果 mdesc 的 param_offset 不等于 0，那么说明在系统的物理内存基地址加 param_offset 处保存了 bootloader 中设置并传进来的系统参数，保存在这里的参数可以是 struct param_stuct 格式（老格式）或扩展的 struct init_tags 格式（新格式）的，在 Linux 2.6.10 中，如果 bootloader 传进来的参数是 struct param_struct 格式的，这里会将其转换成连续存放的 struct tag 格式数据。如果 param_offset 等于 0，即不从 bootloader 读取 Linux 系统参数，那么系统将使用保存在 struct init_tags 格式全局变量 init_tags 中的数据作为系统的参数，把系统参数（_va(param_offset）处的数据或 init_tags）指针传给 truct tag 型局部结构指针 tags。如果 mdesc 的 fixup()成员函数有定义，则调用 fixup()成员函数做一些前期补充初始化并设置系统参数等，这个成员函数很少在处理器中有定义。再查看系统参数结构 tags 中是否设置了核心参数（参数标签字为 ATAG_CORE=0x54410001，其中设置的参数包括根文件系统是否只读、内存页的大小、根文件系统设备号），如果设置了核心参数，再检查保存在 struct meminfo 型全局结构变量 meminfo（这个变量的成员都被初始化为 0）中的系统内存信息，如果 meminfo 中的 nr_banks 不等于 0，说明它已经被前面的 fixup()成员函数操作更新过，那么将 tags 中设置内存参数的结构（标签字为 ATG_MEM= 0x54410002）清除掉，即将结构的标签字设置为 ATG_NONE= 0x00000000，tags 中的其他参数结构不变，然后分析连续排放在 struct tag 结构变量的地址 tags 处所有有效参数设置标签，从.taglist 区根据参数标签字找出每个有效参数标签对应的 struct tagtable 结构变量，执行 struct tagtable 结构变量中的标签解析函数解析相应参数设置标签的内容，在 Linux 2.6.10 中用__tagtable()宏定义了 11 个 struct tagtable 全局结构变量，也就是说 Linux 2.6.10 支持 11 种系统参数通过设置标签到 struct tagtable 全局结构变量中去查找对应的参数设置函数来设置参数（详见第 32 章）。

设置 struct mm_struct 型全局结构变量 init_mm（在声明时就对一部分成员变量进行了初始化）的成员变量，将 Linux 内核编译出来的代码段起始、结束地址、数据段结束地址和整个内核结束地址分别保存到 init_mm 的 start_code、end_code、end_data 和 brk 成员变量中。

将 1024 字节的系统默认命令行字符串全局变量 default_command_line 复制到系统备份命令行字符串全局变量 saved_command_line 中，然后解析 default_command_line 中的每个命令，系统认识的系统参数设置命令都有一个对应的 struct early_params 全局结构变量，变量中定义了命令参数名以及对应的命令解析函数，所有这些 struct early_params 全局结构变量都是通过

__early_param()宏定义的，它们被连续放置在__early_param 区，一条命令行可以设置多个参数命令，两个参数命令之间只能用空格隔开，参数命令内部不能用空格隔开，同一参数的不同属性用逗号隔开（详见第 32 章）。

调用 paging_init()函数，根据全局结构变量 meminfo 中的系统内存信息，将系统中属于同一内存 node 中的 bank 组成一个 node，汇总出系统中的内存 node 数以及每个内存 node 所占的内存空间（包括内存孔洞）大小、物理开始地址、该内存 node 空间（包括内存孔洞）的页帧位码占的内存页数，统计系统中所有内存 node 的页帧位码所占的总内存页数，统计系统中的低端内存最大可能页数 max_low_pfn（包括内存孔洞）和系统中物理内存最大可能页数 max_pfn（包括内存孔洞），把系统中的高端内存开始地址 high_memory 设置为系统中低端内存结束地址对应的虚拟地址。查询 meminfo 中属于 0 号内存 node 的所有 bank，看是否有一个足够大的 bank，这个 bank 要能装下编译出来的整个内核空间以及系统所有内存 nodes（包括内存孔洞）的页帧位码所占的内存空间，如果没有系统崩溃。查询 meminfo 中所有 bank，找到包含 bootloader 存放 initrd 空间的内存 bank 所属 node 号，如果没有 bank 能包含 initrd，那么将 phys_initrd_start 和 phys_initrd_size 清 0，删掉 initrd。在配置了 CONFIG_DISCONTIGMEM 选项即系统中有多个内存 node 的情况下，按从最高到 0 号内存 node 顺序，将系统中每个内存 node 对应的 pg_data_t 结构变量 discontig_node_data[n] 连接到开环链表 pgdat_list 中（最后 pgdat_list 指向 discontig_node_data[0]）（在没有配置 CONFIG_DISCONTIGMEM 选项即系统中只有一个内存 node 的情况下，系统中只有 1 个内存 node，pgdat_list 始终指向 contig_page_data），将该 node（包括内存孔洞）空间对应的页帧位码全部设置为 1，即将这个内存 node 中的内存都设置为不能使用状态，再将这个内存 node 中的每个内存 bank 对应的页帧位码清 0，即将事实存在的物理内存空间设置为可以使用（由此看出，那些内存 node 内各 bank 之间的内存孔洞对应的页帧位码被设置为 1，不能使用状态，这样系统使用内存时就不会踩空），对于 0 号节点，再把内核物理开始地址__pa（_stext）到内核物理结束地址__pa（_end）前面的整个内核空间、从__pa（swapper_pg_dir）到内核物理开始地址__pa（_stext）之间的 16KB 一级页表空间、从内核物理结束地址__pa（_end）开始的系统所有内存 nodes（包括内存孔洞）的页帧位码所占的内存空间根据具体处理器型号不同从系统物理内存开始地址到__pa（swapper_pg_dir）之间的一段内存空间对应的页帧位码设置为 1，不允许内核和用户使用 0 号内存 node 中的这 4 段内存空间，再将存放在 phys_initrd_start 处大小为 phys_initrd_size 的 initrd 内存空间对应的页帧位码设置为 1，保护 initrd 系统在被解压缩、挂接前不被系统破坏。把 initrd_start 设置成__phys_to_virt(phys_initrd_start)，把 initrd_end 设置成 initrd_start + phys_initrd_size。更新 meminfo，调用 alloc_bootmem_low_pages()函数从低端内存空间分配第 1 页空闲内存，将该页的页帧位码设置为 1，并将这页内存空间全部清 0。根据编译前在配置菜单中对 cache 的设置情况以及当前处理器的版本号修正前面设置的系统的 cache 政策编号全局结构变量 cachepolicy，如果处理器的版本编号小于 ARMv5 的版本编号，再将全局变量 ecc_mask 设置为 0，因为 ARMv5

以前处理器的一级描述符中没有定义第 9 位作为保护标志位。如果是 ARMv5 或之前的处理器，将所有的存储空间类型描述结构中的 prot_l1 和 prot_sect（即一级描述符）的第 4 位都设置为 1。根据当前系统的 cache 政策设置各种存储空间类型对应的描述结构 mem_types[n]中的成员变量 prot_l1（一级描述符）或者 prot_sect（段描述符）、prot_pte（页描述符，如果是二级分页管理）中与 cache 和 Write Buffer 相关的位，再将 RAM 存储空间、低端和高端中断向量表空间的一级描述符或上全局变量 ecc_mask。将全局数组 protection_map[16]（声明时被初始化，预定义了 16 个 Linux 版本页描述符中的控制位域设置）中的每个组员都或上当前 cache 政策描述结构中的 pte 成员变量值，设置全局变量 pgprot_kernel（内核空间 Linux 版本页描述符中的控制域设置）。调用 alloc_bootmem_low_pages()函数从低端内存空间分配第 2 页空闲内存，将该页的页帧位码设置为 1，并将这页内存空间全部清 0。先后为 ROM（flash）、各个内存 bank、内部高速 SRAM、内部 mini cache 等存储空间创建虚拟映射描述结构 struct map_desc 变量，这些结构变量先后连续存放。从虚拟地址 0 开始按两段（2MB）大小递增步进依次查询，如果查询到的虚拟地址不属于刚创建的虚拟映射描述结构表示的虚拟空间范围，那么将这两段（2MB）虚拟地址对应的两个一级虚拟映射条目设置为 0，如果当前 ARM 处理器是通过清空数据 cache 来清空 TLB，那么再将这两个一级映射条目所在地址对应的数据 cache 块清空，如果查询的虚拟空间在上述某个存储空间的虚拟空间范围内，那么调用 create_mapping()函数为该存储空间的整个空间创建一二级页表（二级页表空间是新申请的内存空间，每次申请两段虚拟空间 Linux 和硬件版本的二级页表空间，大小为 1024 个字，即 4096 个字节），创建页表的原则是，能分段管理（虚拟和物理开始地址按 1MB 对齐，空间大于 1MB）的就分段管理，不能的就分页管理，分段管理的存储空间剩余部分大于 1 页的进行分页管理，最后剩余小于 1 页大小的空间将不被创建映射条目，即不会被映射。设置高端中断向量空间的映射表述结构，再为高端向量空间创建一二级页表，高端中断向量虚拟地址 0xFFFF0000 处的 1 页空间映射到刚申请到的第 2 页内存空间对应的物理地址上。如果系统使用的是低端中断向量，那么将低端中断向量虚拟地址 0x0 处的 1 页空间也映射到刚申请到的第 2 页内存空间对应的物理地址上，然后刷新整个系统的 cache 和 TLB，调用当前系统硬件平台对应的平台描述结构 struct machine_desc 型变量中的 map_io()成员函数，该函数一般做两件事：第 1 件是为预先定义好的 struct map_desc 型全局结构变量描述的 I/O 空间建立一二级页表；第 2 件就是前期初始化系统中的 UART 接口，以 S3C2410 为例，map_io()成员函数就是 smdk2410_map_io()函数，该函数首先为 struct map_desc 型全局结构数组 smdk2410_iodesc[]中每个变量描述的映射空间建立第 1、2 级页表，然后调用 s3c2410_init_uarts()函数对处理器 UART 接口做前期初始化，该函数对 S3C2410 处理器的 UART 接口做前期初始化，将 S3C2410 处理器的 3 个 UART 接口的配置结构数组中各个变量 smdk2410_uartcfgs[n]（n=0~2）中的 UART 硬件端口编号对应的 UART 接口的平台设备描述结构指针 uart_devices[smdk2410_uartcfgs[n]->hwport]分别注册为 S3C2410 处理器的 UART 接口平台设备描述结构指针数组中的 s3c24xx_uart_devs[n]（n=0~2），并将 s3c24xx_uart_devs[n]中的平台设备数据指针指向 smdk2410_uartcfgs[n]。统计 S3C2410 处理器的 UART 接口总数传给全

局变量 s3c2410_uart_count，再调用 s3c24xx_set_board()函数初始化 USB、LCD、WDT、I2C、I2S 接口的时钟描述结构，初始化 S3C2410 系统中的板子描述结构全局变量 smdk2410_board 中的时钟描述结构指针 clocks 中的每个时钟描述结构变量，把每个时钟描述结构拥有的模块结构指针设置为 THIS_MODULE，使用计数器原子数 used 设置为 0，如果时钟使能函数指针 enable 为空，将 enable 指向 clk_null_enable()函数，在时钟描述结构变量信号量 clocks_sem 保护下将时钟描述结构变量连接到系统时钟描述结构连表全局变量 clocks 中。刷新整个系统的 TLB，针对系统中每个内存 node，统计其中的含空洞内存总页数，以及不含孔洞内存总页数，传给该 node 对应的 discontig_node_data[n]或 contig_page_data 中的 node_spanned_pages 和 node_present_pages 成员变量，为该内存 node 中的所有内存页（包括孔洞占的页数）申请页描述结构变量 struct page 空间，申请的数量要比该 node 中的含孔洞总页数多 1，在此申请内存首选在本 node 内系统 DMA 物理内存空间之后的空间申请，并要求申请到的内存的物理开始地址是按处理器的 cache line（cache 块大小=32B）对齐的，然后把申请到内存空间的虚拟开始地址传给该 node 对应的 discontig_node_data[n]或 contig_page_data 中的 node_mem_map 成员变量，如果系统中没有配置 CONFIG_DISCONTIGMEM 选项即系统中只有一个内存 node，那么将成员变量 node_mem_map 的值也传给全局变量 mem_map，初始化 discontig_node_data[n]或 contig_page_data 中的等待队列 kswapd_wait 成员变量，统计系统中所有内存 node 的 DMA 和 NORMAL 页区中实际存在的内存总页数（不包括内存孔洞）作为系统内核内存总页数传给全局变量 nr_kernel_pages，统计系统中所有内存 node 的所有页区中实际存在的内存总页数（不包括内存孔洞）作为系统内存总页数传给全局变量 nr_all_pages，将该 node 中的每个 ZONE 对应 struct zone 结构变量的地址作为全局指针数组 zone_table[]的第[NODEZONE(nid, j)]号组员，nid 为 node 号，j 为 ZONE 在该 node 中的编号（0~2），初始化 zone_table[NODEZONE(nid, j)]结构变量，初始化该 node 中每个 ZONE 中的每个内存页对应的 struct page 结构变量，初始化 zone_table[NODEZONE(nid, j)]结构变量中的 free_area[]成员数组，为 0~MAX_ORDER 的每个 order 值对应的 frea_area[order]的内存区域伙伴位码表空间申请内存，将申请到的内存的虚拟地址传给 frea_area[order].map，申请内存时首选在该 ZONE 所在的 node 中进行，申请到的内存物理开始地址应该按处理器的 cache line（cache 块大小=32B）对齐。如果系统中没有配置 CONFIG_DISCONTIGMEM 选项即系统中只有一个内存 node，那么再次将 contig_page_data. node_mem_map 的值传给全局变量 mem_map。将上面申请到的第 1 页内存空间再次清 0，并将它对应的页描述结构 truct page 变量的地址传给全局变量 empty_zero_page，然后清空 empty_zero_page 地址处的 cache。

调用 request_standard_resources()函数，设置内核代码段资源（kernel_code）空间的物理开始地址（virt_to_phys(&_text)）（kernel_code.start）和结束地址（virt_to_phys(&_etext - 1)）（kernel_code.end），设置内核数据段资源（kernel_data）空间的物理开始地址（virt_to_phys (&__data_start)）（kernel_data.start）和结束地址（virt_to_phys(&_end - 1)）（kernel_data.end），

即编译出来的内核文件中的内核代码段和数据段的开始和结束地址。依次为系统中的每个内存bank申请物理基址按处理器cache line对齐的资源结构变量空间struct resource *res，将内存bank的物理开始地址和结束地址-1 作为内存 bank 资源空间的物理开始地址 res->start 和结束地址res->end，资源名称res->name都被设置为“System RAM（系统内存）”，资源标志字都被设置为IORESOURCE_MEM | IORESOURCE_BUSY，然后将内存bank资源结构res注册到整个系统空间资源结构 iomem_resource 的子资源链表中去，如果内核代码段和数据段在该内存 bank空间内，那么将kernel_code和kernel_data注册到该内存bank的资源结构的子资源链表中去。如果当前系统硬件平台对应的平台描述结构 struct machine_desc 型变量 mdesc 中的 video_start成员变量不等于0，那么将mdesc->video_start和mdesc->video_end作为系统显示设备内存资源video_ram 空间的物理开始地址和结束地址，然后把 video_ram 也注册到系统总资源空间结构iomem_resource的子资源链表中去。如果mdesc->reserve_lp0/1/2不等于0，那么将并口资源结构变量lp0/1/2注册到系统I/O端口总资源结构变量ioport_resource的子资源链表中去。资源注册都在资源锁resource_lock的保护下进行，各子资源链表中的资源都是按资源空间物理地址从低到高顺序排列。

将当前系统硬件平台对应的平台描述结构struct machine_desc型结构变量mdesc中的中断请求初始化成员函数 init_irq()传给全局函数指针变量 init_arch_irq。将 mdesc 中的系统硬件定时器描述结构struct sys_timer指针timer传给全局结构变量system_timer。将mdesc中的平台初始化函数 init_machine()传给全局函数指针变量 init_machine。如果编译前配置了虚拟超级终端CONFIG_VT 选项，并且配置了 VGA 控制台选项 CONFIG_VGA_CONSOLE，那么将 struct consw全局结构指针conswitchp指向全局结构变量vga_con（在声明时被初始化），如果没有配置 CONFIG_VGA_CONSOLE，而是配置了 CONFIG_DUMMY_CONSOLE 选项，那么将conswitchp指向全局结构变量dummy_con（在声明时被初始化）。

17.5.5 setup_per_cpu_areas()函数

该函数为系统中每个处理器的per_cpu变量申请空间。

所以在单处理器（没有配置CONFIG_SMP选项）系统中该函数是空函数。在Linux2.6.10内核的ARM处理器部分没有配置CONFIG_SMP选项，即不支持ARM对称多处理器系统。

在支持多处理器系统中，该函数将__per_cpu_start 和__per_cpu_end 之间的.data.percpu 区的容量按处理器的 cache line 大小对齐，得到各处理器自己实现的多处理器专用变量区容量，如果系统支持运行时模块加载（配置了CONFIG_MODULES选项），那么各处理器自己实现的多处理器专用变量区容量还必须满足大于等于PERCPU_ENOUGH_ROOM（=32KB）的条件，因为有些多处理器专用变量会在模块中定义、加载。再申请(NR_CPUS)*(各处理器自己实现的多处理器专用变量区容量)大小的内存空间，申请到的空间的物理基址按页大小对齐，空间被

清 0，每个处理器的多处理器专用变量区虚拟基址等于(申请到的内存空间虚拟基址)+(处理器编号*各处理器自己实现的多处理器专用变量区容量)，把每个处理器的多处理器专用变量区虚拟基址相对于虚拟地址__per_cpu_start 的偏移量传给各处理器对应的全局变量__per_cpu_offset [cpu_id(cpu_id 为处理器编号)]中，然后将__per_cpu_start 和__per_cpu_end 之间的实际容量的所有多处理器专用变量初始数据复制到每个处理器的多处理器专用变量区虚拟基址上。

17.5.6 smp_prepare_boot_cpu()函数

该函数初始化多处理器系统中的处理器位码表，所以在单处理器（没有配置 CONFIG_SMP 选项）系统中该函数是空函数。

在支持多处理器系统中（如 MIPS 处理器），该函数将__cpu_number_map[0]和__cpu_logical_map[0]设置为 0，将 3 张处理器位码表 phys_cpu_present_map、cpu_online_map、cpu_callin_map 的第 0 位都设置为 1。

17.5.7 sched_init()函数

该函数初始化每个处理器的可运行进程队列，设置系统初始化进程即 0 号进程，具体执行情况说明如下。

获取系统中每个处理器的运行队列结构（struct runqueue）变量 per_cpu_runqueue，操作每个处理器的 per_cpu_runqueue 变量。初始化自旋锁成员变量 lock，将处理器的 per_cpu_runqueue 中的活动和超限优先级阵列结构指针 active、expired 分别指向处理器的 per_cpu_runqueue 中的优先级阵列结构数组 arrays[2]的第 0、1 个组员结构变量。如果配置了 CONFIG_SMP 选项，那么将处理器的 per_cpu_runqueue 中的调度域结构变量指针 sd 指向系统哑元（dummy）调度域全局结构变量 sched_domain_dummy，将 CPU 负载值 cpu_load 设置为 0，将活动平衡值 active_balance 设置为 0，将移动线程描述结构指针 migration_thread 设置为 NULL，初始化移动队列链表表头结构变量 migration_queue。

将每个处理器的 per_cpu_runqueue 中优先级阵列结构数组 arrays[2]中的每个优先级阵列结构变量中的优先级号 0~（MAX_PRIO-1）对应的位码清 0，初始化优先级阵列结构变量中优先级号 0~（MAX_PRIO-1）对应的队列连表表头结构变量，将优先级号 MAX_PRIO 对应的位码设置为 1。

将内存管理描述结构全局结构变量 init_mm 的计数器 mm_count 加 1，该变量声明时被初始化为 1。

调用 enter_lazy_tlb(&init_mm, current)函数，该函数是空，什么也不做。

调用 init_idle(current, smp_processor_id())函数，将当前任务即系统启动进程的任务描述结

构全局变量 init_task 的平均睡眠值成员变量 sleep_avg 设置为 0，将互信指数成员变量 interactive_credit 设置为 0，将优先级阵列结构指针 array 设置为 NULL，将优先级编号 prio 设置为最高（最大）优先级 MAX_PRIO，任务状态 state 设置为正在运行状态 TASK_RUNNING。将系统启动进程对应的线程信息结构全局变量 init_thread_info（init_thread_union.thread_info）的 CPU 成员变量设置为当前 CPU 号。禁止中断，并且在当前 CPU 的运行队列结构变量 per_cpu_runqueue 的自旋锁的保护下完成：将当前 CPU 的 per_cpu_runqueue 中的当前进程和空闲进程的描述结构指针都指向 init_mask，将 init_task 对应的线程信息结构变量 init_thread_info 的标志字成员变量 flags 的 TIF_NEED_RESCHED 位（第 2 位）设置为 1，表示系统启动进程需要调度，打开中断，将当前 CPU 的 per_cpu_runqueue 中的锁解锁。如果系统支持抢占，那么如果 init_task.lock_depth 大于等于 0（启动进程被锁），那么将 init_thread_info 的抢占计数器 preempt_count 设置为 1，启动进程禁止被抢占，否则设置为 0，启动进程可以被抢占；如果系统不支持抢占，直接将 init_thread_info 的抢占计数器 preempt_count 设置为 0。

17.5.8 build_all_zonelists()函数

该函数建立系统内存页区（zone）链表。

该函数在支持非统一内存访问（即配置了 CONFIG_NUMA 选项，该选项一般用于多处理器系统，详见第 18 章），系统中的定义比移植内存访问系统中的定义略有不同，主要表现在 build_zonelists()函数的实现上。

在不支持非统一内存访问的系统中，该函数初始化每个内存 node 对应的描述结构变量 discontig_node_data[node_id]或者 contig_page_data 的结构数组 node_zonelists[0~2]，把系统中所有内存 node 对应的描述结构变量 discontig_node_data[node_id]或者 contig_page_data 中的页区描述结构变量 node_zone[ZONE_NORMAL=1]的地址按从本节点往上循环到本节点前的节点的先后顺序存放到 node_zonelists[0]的页区描述结构指针数组中；把系统中所有内存 node 对应的描述结构变量 discontig_node_data[node_id]或者 contig_page_data 中的页区描述结构变量 node_zone[ZONE_DMA（=0）]的地址按从本节点往上循环到本节点前的节点的先后顺序存放到 node_zonelists[__GFP_DMA（=1）]的页区描述结构指针数组中；把系统中所有内存 node 对应的描述结构变量 discontig_node_data[node_id]或者 contig_page_data 中的页区描述结构变量 node_zone[ZONE_HIGHMEM（=2）]的地址按从本节点往上循环到本节点前的节点的先后顺序存放到 node_zonelists[__GFP_HIGHMEM（=2）]的页区描述结构指针数组中。

17.5.9 page_alloc_init()函数

该函数在不支持多处理器或者不支持处理器热插拔的系统中的失控函数。

在支持多处理器并且支持处理器热插拔的系统中，该函数设置一个通知者结构 struct

notifier_block 局部变量，这个变量的通知者回调函数为 page_alloc_cpu_notify()，优先级为 0，然后将该局部变量按从高到低的优先级顺序注册到 CPU 通知者结构变量链表 cpu_chain 中去，注册操作在 CPU 控制全局信号量 cpucontrol 的保护下执行。

17.5.10 printk("Kernel command line: %s\n", saved_command_line)

打印Bootloader中设置并传进来的系统命令字符串或者在编译配置系统时在配置菜单中设置的系统命令字符串。

17.5.11 parse_early_param()函数

该函数解析早期格式内核参数，详见第 26 章。

首先在 struct kernel_param 型命令行内核参数结构变量区*(__param)中查找，查看 saved_command_line 命令行中是否设置了*(__param)区中同名的参数。如果有，则调用命令行内核参数结构中的参数设置函数将命令行中的参数值设置到相应的全局变量中，但因为指定的*(__param)区间为空，所以实际上以上操作没有进行而是留到了下面的函数来做。在 struct obs_kernel_param 型老式命令行内核参数结构变量区从__setup_start 到__setup_end 之间的.init.setup 中查找，看 saved_command_line 命令行中是否设置了.init.setup 区中同名的参数，如果是，而且该老式命令行内核参数的描述结构的早期设置标志成员变量 early 不等于 0，那么调用老式命令行内核参数结构中的参数设置函数将命令行中的参数值设置到相应的全局变量中。

17.5.12 parse_args()函数

parse_args("Booting kernel", command_line, __start___param, __stop___param - __start___param, &unknown_bootoption)函数解析新格式内核参数，详见第 26 章。具体执行情况说明如下：

首先在__start___param 到__stop___param 之间的 struct kernel_param 型命令行内核参数结构变量区*(__param)区中查找，看 command_line 命令行中是否设置了*(__param)区中同名的参数，如果有则调用命令行内核参数结构中的参数设置函数将命令行中的参数值设置到相应的全局变量中。

然后对于不是*(__param)区中参数的设置命令进行相应的调整，在参数与参数值之间加入'='，去除多余的空格，然后再在 struct obs_kernel_param 型老式命令行内核参数结构变量区从__setup_start 到__setup_end 之间的.init.setup 中查找，看 saved_command_line 命令行中是否设置了.init.setup 区中同名的参数，如果是，而且该老式命令行内核参数的描述结构的早期设置标志成员变量 early 不等于 0，那么调用老式命令行内核参数结构中的参数设置函数将命令行中的参

数值设置到相应的全局变量中。

如果 command_line 中的命令不是以上两种参数设置命令格式，且设置了参数的参数值，那么把参数字符串（不是参数值）传给系统启动后初始的环境变量全局数组 envp_init[]中同名环境变量或空环境变量中。如果没有设置参数值，则把参数字符串（不是参数值）传给系统启动后初始的参数全局数组 argv_init[]中同名参数或空参数变量中。

17.5.13 sort_main_extable()函数

该函数将放在__start___ex_table 到__stop___ex_table 之间的*(__ex_table)区中的 struct exception_table_entry 型全局结构变量按 insn 成员变量值从小到大排序，即将可能导致缺页异常的指令按其指令二进制代码值从小到大排序。

17.5.14 trap_init()函数

该函数把放在.Lcvectors 处的系统 8 个意外的入口跳转指令搬到高端中断向量 0xffff0000 处，再将从__stubs_start 到__stubs_end 之间的各种意外初始处理代码搬到 0xffff0200 处。刷新 0xffff0000 处 1 页范围的指令 cache，将 DOMAIN_USER（0 号存储域）的访问权限由 DOMAIN_MANAGER 权限改设置成 DOMAIN_CLIENT 权限。

17.5.15 rcu_init()函数

该函数初始化当前 CPU 的读、复制、更新数据结构（struct rcu_data）全局变量 per_cpu_rcu_data 和 per_cpu_rcu_bh_data。将 per_cpu_rcu_data.curtail 指向 per_cpu_rcu_data.curlist，per_cpu_rcu_data.nxttail 指向 per_cpu_rcu_data.nxtlist，per_cpu_rcu_data.donetail 指向 per_cpu_rcu_data.donelist，per_cpu_rcu_data.quiescbatch 设置成 rcu_ctrlblk.Completed，per_cpu_rcu_data.qs_pending 设置成 0，per_cpu_rcu_data.cpu 设置为当前处理器号。将 per_cpu_rcu_bh_data.curtail 指向 per_cpu_rcu_bh_data.curlist，per_cpu_rcu_bh_data.nxttail 指向 per_cpu_rcu_bh_data.nxtlist，per_cpu_rcu_bh_data.donetail 指向 per_cpu_rcu_bh_data.donelist，per_cpu_rcu_bh_data.quiescbatch 设置成 rcu_bh_ctrlblk.Completed，per_cpu_rcu_bh_data.qs_pending 设置成 0，per_cpu_rcu_bh_data.cpu 设置为当前处理器号。设置当前 CPU 的小任务（tasklet）描述结构 struct tasklet_struct 全局变量 per_cpu_rcu_tasklet，将 per_cpu_rcu_tasklet.next 设置为 NULL，状态 per_cpu_rcu_ tasklet.state 设置为 0，per_cpu_rcu_tasklet.count 设置为 0，任务回调函数 per_cpu_rcu_tasklet.func 设置为 rcu_process_callbacks()，无符号长整型数据 per_cpu_rcu_tasklet.data（传给回调函数的参数）设置为 0。在配置了 CONFIG_SMP 并支持 CPU 热插拔即配置了 CONFIG_HOTPLUG_CPU 选项的系统中，将 struct notifier_block 型全局结构变量 rcu_nb 按从高到低的优先级顺序注册到 CPU 通知者结构变量链表 cpu_chain 中去，注册操作在 CPU 控制全局信号量 cpucontrol 的保护执行。

17.5.16 init_IRQ()函数

初始化系统中支持的最大可能中断数（ARM 中是 128 个）的中断描述结构 struct irqdesc 变量数组 irq_desc[NR_IRQS（=128）]，把每个结构变量 irq_desc[n]都初始化为预先定义好的坏中断描述结构变量 bad_irq_desc，并初始化该中断的链表表头成员结构变量 pend。

调用全局函数指针 init_arch_irq()所指的函数，该函数指针声明时被初始化为空函数，后在 setup_arch()函数中被赋值为 mdesc->init_irq，即当前系统硬件平台对应的平台描述结构 struct machine_desc 型结构变量中的中断请求初始化成员函数 init_irq()，以 IXP4XX 处理器平台为例，这个函数就是 ixp4xx_init_irq()，该函数首先将系统中的所有中断设置为 IRQ 中断而不是 FIQ 中断，并且禁止所有中断，然后，操作全局结构变量数组 irq_desc[NR_IRQS]的每个组员 irq_desc[n]，将 irq_desc[n]中的处理器硬件中断处理操作函数组结构指针 chip 指向 IXP4XX 自行定义的 ixp4xx_irq_chip 全局结构变量（该操作在中断请求控制自旋锁 irq_controller_lock 保护下进行，并且必须禁止中断），将 irq_desc[n]中的中断请求处理函数指针 handle 指向 do_level_IRQ()函数（该操作也在中断请求控制自旋锁 irq_controller_lock 保护下进行，并且必须禁止中断），设置 irq_desc[n]的有效标志位 valid 为 1，探测状态标志位清 0，禁止自动使能标志位清 0（该操作也在中断请求控制自旋锁 irq_controller_lock 保护下进行，并且必须禁止中断）。

调用 init_dma()函数，该函数调用 arch_dma_init(dma_chan)初始化 dma 通道，从各 CPU 具体的定义看 arch_dma_init()函数什么也不做，是个空函数。

17.5.17 pidhash_init()函数

该函数设置系统中每种 pid hash 表中的 hash 链表数的移位值全局变量 pidhash_shift，将 pidhash_shift 设置为 min(12，max(4,4*系统内核内存总数 nr_kernel_pages 再除以 106 得到的整数值的二进制数的第一位为 1 的位号))；分别为每种 hash 表的连续 hash 链表表头结构空间申请内存，把申请到的内存虚拟基址分别传给 pid_hash[n]（n=0~3），并将每种 hash 表中的每个 hash 链表表头结构 struct hlist_head 中的 first 成员指针设置为 NULL。

17.5.18 init_timers()函数

初始化当前处理器的时间向量基本结构 struct tvec_t_base_s 全局变量 per_cpu_tvec_bases，初始化 per_cpu_tvec_bases 的自旋锁成员变量 lock，定时器向量根结构成员变量 tv1 和定时器向量结构成员变量 tv2~tv5 中的每个链表表头结构变量，将定时器“滴答”数成员变量 timer_jiffies 设置为系统当前的“滴答”数全局变量 jiffies。

按从高到低的优先级顺序将系统定时器通知者块结构（struct notifier_block）全局变量

timers_nb 注册到 CPU 通知者结构结构变量链表 cpu_chain 中。

设置系统定时器软中断行动函数描述结构变量 softirq_vec [TIMER_SOFTIRQ（=1）]，将 softirq_vec [1]的行动函数指针 action()指向 run_timer_softirq()函数，参数指针设置为 NULL；

17.5.19 softirq_init()函数

设置系统小任务软中断行为函数描述结构变量 softirq_vec [TASKLET_SOFTIRQ（=6）]，将 softirq_vec [6]的行动函数指针 action 指向 tasklet_action()函数，参数指针设置为 NULL。

设置系统高级软中断行为函数描述结构变量 softirq_vec [HI_SOFTIRQ(=0)]，将 softirq_vec [0]的行动函数指针 action 指向 tasklet_hi_action()函数，参数指针设置为 NULL。

17.5.20 time_init()函数

检查系统定时器描述结构 struct sys_timer 全局变量 system_timer（它在 setup_arch()函数中被设置为当前系统硬件平台对应的平台描述结构 struct machine_desc 型结构变量中的成员结构指针 timer）中的 offset()成员函数指针（对于 IXP4XX 处理器，它指向 ixp4xx_gettimeoffset()函数）是否为空，如果是，将其指向 dummy_gettimeoffset()函数。执行 system_timer->init()函数，对系统定时器进行硬件初始化，对于 IXP4XX 处理器，system_timer->init()就是 ixp4xx_timer_init()函数，该函数清除中断等待寄存器中的定时器 1 对应位，设置定时器 1 的计数寄存器并使能定时器 1，定时周期设置为 10ms，将时间戳计数寄存器清 0。将系统启动到上一次“滴答”的时间值全局变量 last_jiffy_time 清 0。

安装定时器的中断请求行为结构 struct irqaction 变量（对于 ixp4xx 就是 ixp4xx_timer_irq）。如果要安装的中断请求行为描述结构变量的标志字 flags 中设置了 SA_SAMPLE_RANDOM 位（一般不设置该位），那么调用 rand_initialize_irq()函数，初始化随机中断。找到定时器中断号对应的中断请求描述结构全局变量 irq_desc[n（定时器中断号）]，然后禁止系统中断，并在中断请求控制器自旋锁 irq_controller_lock 的保护下进行以下操作：如果 irq_desc[n]中的中断请求行为描述结构指针 action 非空，那么查看已经安装到相同中断号上的的中断请求行为描述结构和要安装的中断请求描述结构变量的标志字，如果两个变量的标志字都设置了共享中断标志 SA_SHIRQ，那么把要安装的中断请求描述结构变量加到该中断号的中断请求行为描述结构结构链表的末尾；如果 irq_desc[n]中的中断请求行为描述结构指针 action 为空，那么将 action 直接指向要安装的中断请求行为描述结构变量，并且将 irq_desc[n]中的探测标志位 probing、运行标志位 running、等待标志位 pending 等清 0，将禁止深度 disable_depth 置 1，如果 irq_desc[n]中的禁止自动使能标志位 noautoenable 为 0，那么重新将禁止深度 disable_depth 清 0，并且执行 irq_desc[n]->chip->unmask()使能该中断，然后使能系统中断并且将 irq_controller_lock 解锁。

17.5.21 console_init()函数

该函数初始化系统的控制台结构，该函数执行完之后调用 printk()函数将真正将 log_buf 中符合打印级别要求的系统信息打印到控制台上。

禁止系统中断，并在电传打印设备行规则结构锁 tty_ldisc_lock 的保护下，将 N_TTY 型电传打印设备预定义好的行规则描述结构变量 tty_ldisc_N_TTY 注册为系统电传打印设备行规则描述结构数组中的变量 tty_ldiscs[N_TTY（=0）]，再将 tty_ldiscs[0]的编号成员变量 num 设置为 N_TTY，标志字 flags 或上标志位 LDISC_FLAG_DEFINED，将引用计数器 refcount 清 0，然后解锁，使能系统中断。

分别执行系统中已经配置编译进内核的通过宏 console_initcall(fn)定义的连续存放在从__con_initcall_start 到__con_initcall_end 之间的*(.con_initcall.init)区中的个中控制台初始化函数，以 S3C2410 处理器系统为例将要执行 s3c24xx_serial_initconsole()函数，该函数根据注册到 s3c24xx_uart_devs[0]中的 struct platform_device 结构变量的名字判断是 S3C2410 还是 S3C2440 的 UART 接口，将预定义好的 S3C24XX 系统中通用 struct console 结构变量 s3c24xx_serial_console 的数据指针 data 指向预定义好的 S3C24XX 系统通用 struct uart_driver 结构变量 s3c24xx_uart_drv，完善设置每个 UART 接口的 struct s3c24xx_uart_port 全局结构变量 s3c24xx_serial_ports[n]，将 s3c24xx_uart_devs[n]->dev 传给 s3c24xx_serial_ports[n].port.dev，将 struct s3c24xx_uart_info 全局结构变量传给 s3c2410_uart_inf_at 传给 s3c24xx_serial_ports[n].info，将 s3c2410_uart_inf_at-> fifosize 传给 s3c24xx_serial_ports[n].fifosize，将 s3c24xx_serial_ports[n]. uartclk 设置为 1，如果 smdk2410_uartcfgs. uart_flags 中设置了流控标志位 UPF_CONS_FLOW，将 s3c24xx_serial_ports[n]. flags 或上流控标志位 UPF_CONS_FLOW，根据每个 UART 接口 s3c24xx_uart_devs[n].resource 的设置情况设置 s3c24xx_serial_ports[n]的内存空间映射到的物理基址成员变量 mapbase、内存空间虚拟基址成员变量 membase、中断请求号成员变量 irq，从时钟描述结构变量连表 clocks 中找到 s3c24xx_serial_ports[n]接口的时钟描述结构指针传给 s3c24xx_serial_ports[n].clk，将 s3c24xx_serial_ports[n].clk 的使用计数器原子数 used 加 1，复位 UART 接口。

调用 register_console（&s3c24xx_serial_console），如果首选控制台编号 preferred_console 小于 0，则将命令行中选择的控制台编号 selected_console 传给 preferred_console，如果 preferred_console 仍小于 0，说明没有设置“console=”命令，此时如果 s3c24xx_serial_console. index<0，将其设置成 0，如果 s3c24xx_serial_console.setup()函数未定义或者定义了并执行正确，则将 s3c24xx_serial_console.flags 或上 CON_ENABLED| CON_CONSDEV，表示 s3c24xx_serial_console 描述的 UART 接口已经使能，而且已经作为当前系统的控制台接口，再将 preferred_console 设置为 0，如果设置了“console=”命令，则不执行上面两个判断操作。查询控制台命令行描述结构 console_cmdline[MAX_CMDLINECONSOLES]，看其中是否有与

s3c24xx_serial_console.name 同名的已经设置好的结构 console_cmdline[i]，如果有，而且 s3c24xx_serial_console.index<0，则将 console_cmdline[i].index 传给 s3c24xx_serial_console.index，如果 s3c24xx_serial_console.setup()函数未定义或者定义了并执行正确，将 s3c24xx_serial_console.flags 或上 CON_ENABLED，表示 s3c24xx_serial_console 描述的 UART 接口已经使能，如果 i 等于 preferred_console，再将 s3c24xx_serial_console.flags 或上 CON_CONSDEV，表示 s3c24xx_serial_console 描述的 UART 接口已经成为当前系统的控制台接口；如果 s3c24xx_serial_console.flags 已经设置 CON_ENABLED 位，减少控制台信号量 console_sem，将控制台被锁标志全局变量 console_locked 设置为 1，将控制台可以调度标志全局变量 console_may_schedule 设置为 1；如果 s3c24xx_serial_console.flags 设置了 CON_CONSDEV 标志位或者系统控制台结构 struct console 全局指针 console_drivers 为空，那么将 s3c24xx_serial_console 加入到 console_drivers 链表中 console_drivers 的位置，否则加入到 console_drivers 链表中 console_drivers 的后面，如果 s3c24xx_serial_console.flags 设置了 CON_PRINTBUFFER 标志位（几乎所有预定义好的控制台结构变量都设置了该位，意思是打印放在打印缓冲区 log_buf 中的数据），关闭系统中断并在 log 缓冲区锁 logbuf_lock 的保护下将 log 字符串的开始位置 log_start 传给控制台缓冲区的开始位置 con_start，然后解锁，开系统中断。关闭系统中断并在 log 缓冲区锁 logbuf_lock 的保护下，将 con_start 和 log_end 设置成打印开始和结束位置，即下面要打印从 con_start 到 log_end-1 位置的全部字符，解锁，开系统中断，调用 call_console_drivers()函数一行一行打印指定位置段的字符串，如果某行的开始用“<x>”指定了信息级别，那么只打印 x<7 的行，打印设置好的从开始到结束位置之间的字符串，这里打印字符串最后是通过依次调用已经注册到控制台结构连表 console_drivers 中从 console_drivers 开始的各个已经被使能（当然能注册到 console_drivers 控制台结构肯定已经被使能）的控制台结构变量中的 write()成员函数来实现的所以如果系统中成功注册了多个控制台结构变量，那么可以实现多个控制台输出（控制台设备只负责输出 printk()打印的字符，不能接收字符，因为控制台设备很少有实现 read()成员函数的，即便虽然 struct console 结构中定义了 read()成员函数，一般只实现 write()函数，也就是说不能实现交互，实现交互的是 shell 启动之后的事，是该设备的另一个驱动程序）。将 console_locked 和 console_may_schedule 清 0，增加控制台信号量 console_sem，解锁，开中断，如果等待链表 log_wait 中有任务在等待并且 log_buf 中还有未打印的字符并且 oops_in_progress 等于 0，那么唤醒等待链表 log_wait 中状态为 TASK_INTERRUPTIBLE 的任务。

如果 panic_later 指针不等于 NULL，即前面 unknow_bootoption()函数中解析启动选项出错，也就是说命令行中设置了错误的参数命令，那么调用 panic(panic_later, panic_param)打印 panic 错误信息以及错误参数。

17.5.22 profile_init()函数

该函数对系统剖析作相关初始化，系统剖析用于系统调试。

如果全局变量 prof_on 等于 0，什么也不做直接返回。prof_on 声明时没有初始化，默认为 0，如果系统配置命令行中设置了“profile=”命令，则在前面的初始化过程中会调用对应的命令解析函数 profile_setup()来解析“profile=”命令并设置全局变量 prof_on 和 prof_shift。

如果 prof_on 不等于 0，将从_stext 到_etext 的内核代码段和 rodata 段（已经初始化好的全局变量）空间右移 prof_shift 位得到 prof_len，然后为 prof_len 个 atomic_t 型原子数变量申请内存空间，申请到的内存空间的物理起始地址必须按处理器的 cache line 大小对齐，而且首选从 DMA 页区之后的物理内存空间开始分配，把申请到的内存空间的虚拟开始地址传给全局原子数指针 prof_buffer。

17.5.23 local_irq_enable()函数

该函数将处理器的当前系统状态寄存器 CPSR 的第 7 位清 0，使能 IRQ 中断。

判断 initrd 存放的地址是否合法；如果 initrd_below_start_ok 等于 0，即 initrd 不能放在内核空间范围或之下的位置，那么如果 initrd 的虚拟开始地址 initrd_start 不等于 0（表示 initrd 存在）并且小于内核空间的末端_end 所在的虚拟地址，那么这个 initrd 的存放位置就是违法的，必须清除掉，清除的方法是将 initrd_start 清 0。

17.5.24 vfs_caches_init_early()函数

计算系统目录条目结构（struct dentry）hash 表中的条目数（连表数）全局变量 dhash_entries，先将内核内存空间大小（nr_kernel_pages<<PAGE_SHIFT）按 1MB 对齐，然后右移 13 位，并将结果向上调整到为 2*n* 大小，最后得到的值作为 dhash_entries 的值。将系统总内存空间大小（nr_all_pages << PAGE_SHIFT）右移 4 位（除以 16）再除以 sizeof(struct hlist_head)（=4）得到 dhash_entries 的上限值，如果前面计算出的 dhash_entries 值比上限值大，则将上限值作为 dhash_entries 的值，也就是说，系统最多允许将整个系统的全部内存空间的 1/16 来存放系统目录条目结构 hash 表的条目。然后计算出 dhash_entries 以 2 为底的对数值作为系统目录条目结构 hash 表中条目数的移位数全局变量 d_hash_shift 的值，1<<d_hash_shift-1 作为系统目录条目结构 hash 表中条目数的掩码值全局变量 d_hash_mask 的值。然后从系统 DMA 页区之后的 NORMAL 页区开始为系统目录条目 hash 表的所有链表头条目申请内存空间，空间大小为 dhash_entries* sizeof(struct hlist_head)字节，申请到的内存空间的物理基址以处理器的 cache line 大小对齐，将申请到的内存空间的虚拟基址传给全局变量 dentry_hashtable。最后初始化 dentry_hashtable 所指目录条目结构 hash 表中的每个 hash 链表头结构变量。

计算系统索引节点结构（struct inode）hash 表中的条目数（连表数）全局变量 ihash_entries。先将内核内存空间大小（nr_kernel_pages<< PAGE_SHIFT）按 1MB 对齐，然后右移 14 位，并将结果向上调整到为 2*n* 大小，最后得到的值作为 ihash_entries 的值。将系统总内存空间大小

（nr_all_pages << PAGE_SHIFT）右移 4 位（除以 16）再除以 sizeof(struct hlist_head)（=4）得到 ihash_entries 的上限值，如果前面计算出的 ihash_entries 值比上限值大，则将上限值作为 ihash_entries 的值，也就是说，系统最多允许将整个系统的全部内存空间的 1/16 来存放系统索引节点结构 hash 表的条目。然后计算出 ihash_entries 以 2 为底的对数值作为系统索引节点结构 hash 表中条目数的移位数全局变量 i_hash_shift 的值，1<<d_hash_shift-1 作为系统索引节点结构 hash 表中条目数的掩码值全局变量 i_hash_mask 的值。然后从系统 DMA 页区之后的 NORMAL 页区开始为系统索引节点结构 hash 表的所有链表头条目申请内存空间，空间大小为 ihash_entries* sizeof(struct hlist_head)字节，申请到的内存空间的物理基址以处理器的 cache line 大小对齐，将申请到的内存空间的虚拟基址传给全局变量 ientry_hashtable。最后初始化 inode_hashtable 所指系统索引节点结构 hash 表中的每个 hash 链表头结构变量。

17.5.25 mem_init()函数

该函数执行完后不能再用像 alloc_bootmem()、alloc_bootmem_low()、alloc_bootmem_pages()等申请低端内存的函数来申请内存，也就不能再申请大块的连续物理内存了。

计算内核代码段空间大小&_etext - &_text，计算内核数据段的大小&_end - &__data_start，计算内核初始化函数指针空间段的大小&__init_end - &__init_begin(_init_begin=_stext)，如果系统没有配制 CONFIG_DISCONTIGMEM 选项，将 virt_to_page(high_memory) - mem_map 传给全局变量 max_mapnr；如果系统中有不止 1 个内存 bank 即 meminfo.nr_banks 不等于 1，那么检查系统中每个内存 node 中内存 bank，将同一个 node 内相邻两个 bank 之间的内存间隙空间对应的页描述结构所在内存页的页帧位码清 0（注意：内存孔洞空间本身对应的页位码并没有清 0，仍为 1，也就是说，内存孔洞空间仍然是不能使用的，下面释放的空间是所有位码为 0 的空闲内存空间，然后才把所有内存空间的页位码空间都释放掉，系统将这些内存页释放到内存页区的 free_area 中，以后就通过 free_area 来申请释放内存，原来页位码为 1，不在 free_area 中的内存页是不可能被内核分配使用的）。查看系统中每个内存节点（node），如果内存节点的内存页数不等于 0，则将那些页帧位码为 0 的内存页即空闲页的页描述结构（struct page）中的标志字 flags 中的 PG_reserved 标志位清 0，将页描述结构中的引用计数器_count 清 0，再调用__free_pages()函数释放所有这些空闲页，释放时根据实际情况一页一页释放或者 32 页 32 页批量释放，然后仿照上面操作将内存节点整个空间的所有页帧位码空间的内存页一页一页释放掉，再将内存节点的数据结构（struct bootmem_data）中的页帧位码表空间指针变量 node_bootmem_map 设置为 NULL，把从所有内存 node 释放出来的内存总页数传给全局变量 totalram_pages，打印系统内存信息，打印系统中每个内存 bank 的大小，统计系统中所有内存 bank 包含的实际物理内存总大小并打印，把系统中所有内存 bank 包含的实际物理内存总页数传给全局变量 num_physpages。打印系统空闲内存大小、内核代码段大小、数据段大小以及内核初始化函数指针空间段大小。如果系统中规定的内存每页的大小 PAGE_SIZE 大于等于

16384，并且系统实际物理内存总页数 num_physpages 小于等于 128，那么将系统过量使用内存标志全局变量 sysctl_overcommit_memory 改设置为 OVERCOMMIT_ALWAYS（=1）。

__free_pages(struct page *page, unsigned int order)函数执行过程说明如下。如果要释放的内存页的页描述结构 struct page 中的_count=-1，则系统崩溃；如果页描述结构中的标志字 flags 设置了保留位 PG_reserved 或者其中的_count 不等于 0，那么该页不能被__free_pages()释放，直接返回。只有页描述结构中的标志字 flags 没有设置保留位 PG_reserved 并且_count 等于 0，该页才能被__free_pages()释放。先将页描述结构中的_count 减 1，然后分两种情况进行释放：一种情况是传入的 order 等于 0，即只释放 1 页内存，此时调用 free_hot_page(page)释放内存；另一种情况是 order>0，此时调用__free_pages_ok(page, order)进行释放，分别说明如下。

free_hot_page(page)执行过程：直接调用 free_hot_cold_page(page, 0)。找到要释放的内存页所在的页区的描述结构变量 zone_table[page->flags >> NODEZONE_SHIFT]，调用空函数 arch_free_page()和 kernel_map_pages()（只有配置了 CONFIG_DEBUG_PAGEALLOC 才有意义），将当前处理器的内存页状态描述结构变量 per_cpu_page_states 中的空闲页数计数器加 1。如果该页是匿名页，即页描述结构变量的 mapping 的 bit0 为 1，那么将 mapping 设置为 NULL。释放之前对该页进行检查，如果该页的页描述结构变量的映射计数器_mapcount 大于等于 0，或者地址空间描述结构指针 mappint 不等于 NULL，或者引用计数器_count 不等于-1，或者标志字设置了 PG_lru、PG_private、PG_locked、PG_active、PG_reclaim、PG_slab、PG_swapcache、PG_writeback 中的某位或某些位，那么该页就是错误页，打印错误信息，提示重起系统以解决问题，将 flags 中的以上位全部清 0，将_count 设置为-1，将 mapping 设置为 NULL，将系统错误标志 tainted 或上内存页错误标志位 TAINT_BAD_PAGE，如果 flags 中设置了 PG_dirty 标志位，将 PG_dirty 位清 0，获取该页所在页区描述结构变量 zone_table[page->flags >> NODEZONE_SHIFT]中当前处理器的内存页组中的常用内存页描述结构变量指针 pageset[get_cpu()].pcp[0]。关闭系统中断，保护现场。如果该页所在页区中属于当前处理器的常用内存页数 pageset[get_cpu()].pcp[0].count 大于等于该页区中当前处理器的最高常用内存页数水印值 pageset[get_cpu()].pcp[0].high，则关闭系统中断，将该页所在页区描述结构上锁，将该页区描述结构中的所有页都不能收回标志成员变量 all_unreclaimable 清 0，将已扫描页数计数器成员变量 pages_scanned 清 0，然后将该页区中当前处理器的内存页组中的常用内存页链表中开始的 pageset[get_cpu()].pcp[0].batch 页逐页释放到该页区中 order 值为 0 的内存区域中，释放每页过程中将该页区的空闲页数成员变量 free_pages 加 1，修改对应的伙伴位码，如果相邻的内存区域也是空闲的，那么将它们组合成一个更高 order 值的内存区域，开系统中断，将该页所在页区描述结构解锁。将该页区中当前处理器的内存页组中的常用内存页数计数器 pageset[get_cpu()].pcp[0].count 减去 pageset[get_cpu()].pcp[0].batch，再将该页的描述结构变量通过 lru 成员变量连接到该页区中当前处理器的内存页组中的常用内存页连表中去，然后将该页区中当前处理器的内存页组中的常用内存页数计数器 pageset[get_cpu()].pcp[0].count 加 1。开系

统中断，恢复现场，使能抢占。

__free_pages_ok(page, order)执行过程：调用空函数 arch_free_page()，将当前处理器的内存页描述结构变量 per_cpu_page_state 中的空闲页数计数器加上 1<< order，释放前对每页要释放的内存进行检查，如果该页的页描述结构变量的映射计数器_mapcount 大于等于 0，或者地址空间描述结构指针 mappint 不等于 NULL，或者引用计数器_count 不等于-1，或者标志字设置了 PG_lru、PG_private、PG_locked、PG_active、PG_reclaim、PG_slab、PG_swapcache、PG_writeback 中的某位或某些位，那么该页就是错误页，打印错误信息，提示重启系统以解决问题，将 flags 中的以上位全部清 0，将_count 设置为-1，将 mapping 设置为 NULL，将系统错误标志 tainted 或上内存页错误标志位 TAINT_BAD_PAGE，如果 flags 中设置了 PG_dirty 标志位，将 PG_dirty 位清 0。将要释放的连续内存页的首页的描述结构变量通过 lru 成员变量链接到临时链表 list 中去。调用空函数 kernel_map_pages()（只有配置了 CONFIG_DEBUG_PAGEALLOC 才有意义）；禁止系统中断，将要释放的连续 1<<order 页内存空间所在页区描述结构上锁，将该页区描述结构中的所有页都不能收回标志成员变量 all_unreclaimable 清 0，将已扫描页数计数器成员变量 pages_scanned 清 0，然后将要释放的连续 1<<order 页内存空间一次性释放到该页区中 order 值对应的内存区域中，释放过程中将该页区的空闲页数成员变量 free_pages 加上 1<<order，修改对应的伙伴位码，如果伙伴内存区域也是空闲的，那么将它们组合成一个更高 order 值的内存区域，开系统中断，将该页所在页区描述结构解锁。

17.5.26 kmem_cache_init()函数

该函数执行高速缓存内存管理即 slab 分配器相关初始化。

如果系统物理内存总页数 num_physpages 大于 32MB/PAGE_SIZE，将 slab 指数值全局变量 slab_break_gfp_order 设置为 BREAK_GFP_ORDER_HI（=1）（声明时被设置为 BREAK_GFP_ORDER_LOW（=0））。初始化高速缓存内存互斥信号量全局变量 cache_chain_sem，初始化系统全局高速缓存内存链表 cache_chain，将系统全局高速缓存结构变量 cache_cache 加到高速缓存内存链表 cache_chain 中，将 cache_cache 的颜色块大小成员变量 colour_off 设置成处理器硬件 cache line 的大小，将 cache_cache 中当前处理器对应的高速缓存内存阵列描述结构指针 cache_cache.array[smp_processor_id()]指向系统全局初始高速缓存内存结构变量 initarray_cache 的成员变量 cache，也就相当于指向 initarray_cache。将 cache_cache 的对象大小成员变量 ojbsize 按处理器硬件高速缓存线(cache line)大小对齐，计算一页内存空间最多能容纳多少个高速缓存内存描述结构 struct kmem_cache_s（kmem_cache_t）。按处理器 cache line 大小对齐的对象而且将高速缓存辅助结构变量也放在该页内存空间内，把计算出的数目保存到 cache_cache.num 中，如果计算出的 cache_cache.num 结果为 0，则系统崩溃。然后将上面操作中一页内存最后剩余的内存空间除以颜色块大小 cache_cache.colour_off 得到一页中能包括的颜色块数目传给 cache_cache.colour，将下一个颜色块编号清 0，计算出高速缓存辅助结构变

量空间大小（也就是该高速缓存内存空间的辅助结构变量所占空间大小，包括与该高速缓存内存空间中对象数目相同个数的 kmem_bufctl_t（unsigned int）变量（即一个对象对应一个 kmem_bufctl_t 变量）加上一个 struct slab 结构变量之后按处理器硬件 cache line 大小对齐）传给 cache_cache 的厚片大小成员变量 slab_size。分别调用 kmem_cache_create()函数为 malloc_sizes[]数组中的每个高速缓存大小描述结构（struct cache_sizes）变量中的 cs_cachep 和 cs_dmacachep 分别创建并初始化一个 kmem_cache_t 结构变量，调用 kmalloc()函数为 struct arraycache_init 结构变量申请内核空间，禁止系统中断，将全局 struct arraycache_init 结构变量 initarray_cache 复制到申请到的内核空间中，然后再将 cache_cache.array[smp_processor_id()]改指向该内核空间，使能系统中断；再调用 kmalloc()函数为 struct arraycache_init 结构变量申请内核空间，禁止系统中断，将全局 struct arraycache_init 结构变量 initarray_generic 复制到申请到的内核空间中，然后再将 malloc_sizes[0].cs_cachep->array[smp_processor_id()]改指向该内核空间，使能系统中断。减少内核高速缓存内存连表操作互斥信号量 cache_chain_sem，然后调用 enable_cpucache()函数初始化 cache_chain 链表中每个 kmem_cache_t 结构变量的处理器高速缓存内存阵列结构数组 array[NR_CPUS]，增加内核高速缓存内存连表操作互斥信号量 cache_chain_sem，将全局变量 g_cpucache_up 改设置成 FULL（=2），按从高到低的优先级顺序将通知者结构变量 cpucache_notifier 注册到 CPU 通知者结构变量链表 cpu_chain 中。

kmem_cache_create()函数执行过程：如果名字为空，或者如果现在系统在进行硬件 IRQ 中断处理或软中断处理，或者传进来的要创建的高速缓存内存描述结构对应的对象大小小于系统一个字的大小，或者对象的大小大于最大允许块大小，或者只有 descontructor()函数而没有 contructor()函数，打印 slab 错误信息，系统崩溃。如果传进来的创建标志字中设置了 SLAB_DESTROY_BY_RCU 标志位，即由 RCU 来释放高速内存页，那么此时如果传进来的析构函数 dtor()有定义，系统崩溃，因为如果定义了 dtor()则由 dtor()函数来释放高速内存页。如果传进来的创建标志字设置了创建标志字 CREATE_MASK 之外的其他位，则系统崩溃。如果传进来的对齐值不等于 0，将创建标志字或上 SLAB_RED_ZONE 和 SLAB_STORE_USE 标志位。如果传进来的对齐值等于 0，而且创建标志字设置了 SLAB_HWCACHE_ALIGN 标志位，那么将对齐值设置为系统处理器硬件 cache line 大小，再根据传入的对象大小调整对齐值，如果对象大小比较小，将对齐值缩减 1/2，直到对象大小大于对齐值的 1/2，如果创建标志字没有设置 SLAB_HWCACHE_ALIGN 标志位，将对齐值设置为系统每个字的字节数 BYTES_PER_WORD。调用 kmem_cache_alloc()函数从 cache_cache 中分配一个 kmem_cache_t 结构变量高速缓存对象 cachep，如果申请 kmem_cache_t 结构变量高速缓存对象失败，打印信息，系统 panic()，将申请到的 kmem_cache_t 结构变量高速缓存对象 cachep 空间全部清 0；如果传进来的对象大小没有按 BYTES_PER_WORD 对齐，则将其按 BYTES_PER_WORD 对齐。如果对象大小大于等于 PAGE_SIZE 的 1/8，那么将创建标志字或上 CFLGS_OFF_SLAB 标志，即将高速缓存辅助结构变量放在 slab 对象空间之外，将对象大小按上面设定的对齐值对齐。如

果创建标志字设置了 SLAB_RECLAIM_ACCOUNT 标志位并且对象大小小于等于 PAGE_SIZE，那么将 cachep 的 gfporder 设置为 0，再计算一页（因为 gfporder=0）内存最多能容纳多少个大小为传入的大小值按对齐方式对齐的对象，把计算出的数量传给 cachep->num。如果创建标志字没有设置 SLAB_RECLAIM_ACCOUNT 标志位或者对象大小大于 PAGE_SIZE，将 gfporder 从 0 开始递增一个一个测试，看 2gfporder 页内存最多能够容纳多少个大小为传入的大小值按对齐方式对齐的对象，把计算出的数量传给 cachep->num，直到测试结果满足条件跳出测试循环，这些条件包括 gfporder>=MAX_GFP_ORDER、gfporder>=slab_break_gfp_order、剩余空间 left_over 小于等于 PAGE_SIZE*2gfporder/8。如果上面操作后 cachep->num 等于 0，打印不能创建 kmem_cache_t 结构高速缓存内存空间信息，调用 kmem_cache_free()函数把申请到的 kmem_cache_t 结构高速缓存对象 cachep 释放回 cache_cache 中，将 cachep 设置为 NULL，系统 panic()；计算新创建的高速缓存内存的辅助结构变量所占空间大小并将其按对齐值大小对齐传给局部变量 slab_size；如果创建标志字设置了 CFLGS_OFF_SLAB 标志但剩余空间 left_over 大于 slab_size，则将创建标志字中的 CFLGS_OFF_SLAB 标志清 0，将 left_over 减去 slab_size。如果仍设置了 CFLGS_OFF_SLAB 标志，那么重新计算 slab_size，不再将其按对齐值对齐，也就是说放在 slab 的对象空间之外的 slab 管理结构变量空间的大小不必按对齐值对齐，但与对象空间放在一起的 slab 管理结构变量空间的大小必须按对齐值对齐。将 cachep 的颜色块大小 colour_off 设置成处理器硬件 cache line 大小。如果 colour_off 小于对齐值，则将 colour_off 设置成对齐值大小，将 left_over 除以 colour_offset 作为 cachep 的颜色块数 colour，将 slab_size 作为 cachep 的 slab_size，将创建标志字作为 cachep 的 flags，将 cachep 的 gfpflags 清 0。如果创建标志字中设置了 SLAB_CACHE_DMA 即从 DMA 页区中创建高速缓存内存，则将 cachep 的 flags 或上 GFP_DMA，将 cachep 的 spinlock 上锁，将对象大小作为 cachep 的 objsize，初始化 cachep 的 lists 中的全满厚片链表 slabs_full、部分满厚片链表 slabs_partial、全空闲厚片链表 slabs_free。如果创建标志字设置了 CFLGS_OFF_SLAB 标志位，则从高速缓存内存大小描述结构 struct cache_sizes 全局结构数组 malloc_sizes[]中找出 cs_size 刚好大于等于这里的对象大小的 struct cache_sizes 结构变量，将其 cs_cachep 或 cs_dmacachep（如果 gfpflags 中设置了 GFP_DMA 标志）作为 cachep 的 slabp_cache。将传进来的高速缓存构造和析构函数指针作为 cachep 的 ctor()和 dtor()函数指针，将传进来的名字作为 cachep 的 name。如果系统支持处理器热插拔，将处理器控制信号量全局变量 cpucontrol 减少。如果全局变量 g_cpucache_up（初始设置为 NONE（=0））等于 FULL（=2），调用 enable_cpucache()函数初始化 cachep 中每个处理器的 array [CPU_ID]。如果全局变量 g_cpucache_up 等于 NONE，将 cachep 的 array[smp_processor_id()]指向全局变量 initarray_generic.cache，再将 g_cpucache_up 设置成 PARTIAL，如果 g_cpucache_up 等于 PARTIAL，则调用 kmalloc()为 cachep 的 array[smp_processor_id()]申请一个 struct arraycache_init 结构空间。如果 cachep->array[smp_processor_id()]指针为空，则系统崩溃，否则将 cachep->array[smp_processor_id()]的 avail 设置为 0，limit 设置为 BOOT_CPUCACHE_ENTRIES，batchcount 设置为 1，touched

设置为0，将cachep的batchcount设置为1，limit设置为BOOT_CPUCACHE_ENTRIES，free_limit设置为(1+在线处理器数量)*cachep->batchcount + cachep->num，将cachep的next_reap设置为jiffies + REAPTIMEOUT_LIST3 + ((unsigned long)cachep)%REAPTIMEOUT_LIST3，将高速缓存信号量cache_chain_sem减少，将当前进程的信息结构current_thread_info()的地址限制值addr_limit备份，再将其设置为KERNEL_DS（0x00000000），并将当前线程信息结构中的处理器域成员变量cpu_domain（对应CP15的C3寄存器）的DOMAIN_KERNEL即1号域的访问权限类型依据地址限制值是否为0设置为客户访问类型DOMAIN_CLIENT（地址限制值不等于0）或管理者访问类型DOMAIN_MANAGER（地址限制值等于0）。查询cache_chain链表中的每个kmem_cache_t结构变量，以用户模式读取每个kmem_cache_t结构变量的名字name，如果读取失败打印信息，查看下一个结构变量，如果读取成功，将kmem_cache_t结构变量的名字name与新创建的kmem_cache_t结构变量cachep的名字相比较，如果相同，打印名字重复错误，将高速缓存信号量cache_chain_sem增加，将处理器控制信号量全局变量cpucontrol增加，系统崩溃。如果cachep的名字与cache_chain链表所有的kmem_cache_t结构变量名字都不同，则将当前进程的信息结构current_thread_info()的地址限制值addr_limit恢复到前面保存的值，并将当前线程信息结构中的处理器域成员变量cpu_domain（对应CP15的C3寄存器）的DOMAIN_KERNEL即1号域的访问权限类型依据地址限制值是否为0设置为客户访问类型DOMAIN_CLIENT（地址限制值不等于0）或管理者访问类型DOMAIN_MANAGER（地址限制值等于0），将新创建的cachep通过其next连接到cache_chain中，将高速缓存信号量cache_chain_sem增加，将处理器控制信号量全局变量cpucontrol增加。如果cachep指针空并且创建标志字设置了SLAB_PANIC标志，打印创建slab失败，系统panic()，否则，返回cachep指针。

kmem_cache_alloc(kmem_cache_t *cachep, int flags)函数执行过程：直接调用并返回__cache_alloc (kmem_cache_t *cachep, int flags) 函数，所以这两个函数等价。首先调用调试用函数cache_alloc_debugcheck_before()，禁止系统中断，保存程序状态寄存器，获取cachep的array[smp_processor_id()]指针传给ac。如果ac->avail大于0，将cachep的分配成功计数器allochit加1，将ac->touched设置为1，将ac->avail减1，再将ac后面的第ac->avail个对象指针传给要返回的对象指针objp。如果ac->avail等于0，将cachep的分配失败计数器allocmiss加1，然后调用cache_alloc_refill()分配一个对象传给objp。使能系统中断，恢复程序状态寄存器，调用调试用函数cache_alloc_debugcheck_after()，返回objp指针。

kmem_cache_free (kmem_cache_t *cachep, void *objp)函数执行过程：禁止系统中断，保存程序状态寄存器的值，调用__cache_free(cachep, objp)函数，使能系统中断，恢复程序状态寄存器的值。__cache_free (kmem_cache_t *cachep, void* objp)函数：获取传入的要收回对象的高速缓存内存描述结构cachep的array[smp_processor_id()]指针传给ac。检查系统是否禁止了IRQ中断，如果没有禁止，系统崩溃。调用调试用函数cache_free_debugcheck()。如果ac中的avail小于limit，那么将cachep的成功收回对象计数器freehit加1，把传入的对象指针存入紧随ac之后

的第 avail 个对象指针位置，然后将 avail 加 1，返回。如果 ac 中的 avail 大于等于 limit，将 cachep 的收回对象失败计数器 freemiss 加 1，调用 cache_flusharray(cachep, ac)函数将 ac 中的对象释放掉，再把传入的对象指针存入紧随 ac 之后的第 avail 个对象指针位置，然后将 avail 加 1。

cache_alloc_refill(kmem_cache_t* cachep, int flags)函数执行过程：如果没有禁止系统 IRQ 中断，系统崩溃，将 cachep->array[smp_processor_id]传给 struct array_cache 结构指针 ac，batchcount 等于 ac->batchcount。如果 ac->touched 等于 0，并且 batchcount 大于 BATCHREFILL_LIMIT（=16），将 batchcount 设置为 BATCHREFILL_LIMIT，将&cachep->lists 传给 l3，如果 ac->avail 大于 0，系统崩溃。将 cachep 的 spinlock 上锁，如果 l3 中的共享高速缓存内存阵列结构指针 shared 不是空的，而且 shared 的 avail 大于 0，如果 batchcount 大于 shared 的 avail，将 batchcount 设置为 shared 的 avail，将 shared 中 avail 位置开始的 batchcount 个对象指针复制到紧随 ac 结构后面开始的位置，将 shared 的 touched 设置为 1，avail 减去 batchcount，将 ac 的 avail 设置为 batchcount。如果 l3 中的共享高速缓存内存阵列结构指针 shared 是空的，但 l3 中的 slabs_partial 或 slabs_free 链表中有空闲的 slab 结构，而且 slab 结构中有空闲的对象，那么就从这些 slab 中逐个把 batchcount 个空闲对象指针传到 ac 的对象地址空间中，操作完一个 slab 后要把该 slab 从其所在的 slabs_partial 或 slabs_free 链表中删除，然后根据该 slab 中是否还有空闲对象再将其添加到 slabs_partial 或 slabs_full 链表中，将 l3 的 free_objects 减去 ac 的 avail，将 cachep 的 spinlock 解锁。如果 l3 中的共享高速缓存内存阵列结构指针 shared 是空的，而且 l3 中的 slabs_partial 和 slabs_free 链表中也没有空闲的 slab 结构或者 slabs_partial 和 slabs_free 链表中的空闲对象数不够 batchcount 个，那么只能调用 cache_grow()函数对 cachep 新增加 slab 了，新增对象后，再将 ac 的 touched 设置为 1，返回 ac 中最后位置的空闲对象指针。

cache_grow (kmem_cache_t * cachep, int flags, int nodeid)函数执行过程：首先检查 flags 的设置情况，如果没有禁止系统 IRQ 中断，则系统崩溃。将 cachep 的 spinlock 上锁，offset 等于 cachep 的 colour_next* colour_off，再将 cachep 的 colout_next 加 1，如果 colout_next 大于等于 colour，则将 colour_next 设置为 1，也就是说 colour_next 是卷绕的。将 cachep 的 spinlock 解锁，如果设置了__GFP_WAIT 标志，使能 IRQ 中断。将 cachep 的 flags 或上 flags，然后根据 nodeid 的值调用___get_free_pages()函数（nodeid 等于-1）或 alloc_pages_node()函数（nodeid 不等于-1）从内核申请连续 1<<cachep->gfporder 页物理内存，如果 cachep 的 flags 设置了 SLAB_RECLAIM_ACCOUNT 标志，将全局原子数 slab_reclaim_pages 加上 1<<cachep->gfporder。关闭系统 IRQ 中断，保护现场，将当前处理器的 per_cpu_page_states.nr_slab 加上 1<<cachep->gfporder，开中断，恢复现场，再将刚申请到的 1<<cachep->gfporder 页内存逐页将每页的描述结构的标志字 flags 或上 PG_slab 标志。如果 cachep->flags 设置了 CFLGS_OFF_SLAB 标志，那么调用 kmem_cache_alloc()函数从 cachep->slabp_cache 中分配一个 cachep 的 slab_size 大小的高速缓存内存对象传给 struct slab 结构指针 slabp，用来存放 slab 管理结构变量，如果这里申请高速缓存对象失败，则将刚申请到的 1<<cachep->gfporder 页内存

逐页将每页的描述结构的标志字 flags 清除 PG_slab 标志，将当前处理器的 per_cpu_page_states.nr_slab 减去 1<<cachep->gfporder，如果当前进程的收回状态结构指针 reclaim_state 不空，将当前进程的 reclaim_state 的 reclaimed_slab 加上 1<<cachep->gfporder，调用 free_pages()函数释放刚申请到的 1<<cachep->gfporder 页内存，如果 cachep 的 flags 设置了 SLAB_RECLAIM_ACCOUNT 标志，将全局原子数 slab_reclaim_pages 减去 1<<cachep->gfporder，然后如果设置了__GFP_WAIT 标志，禁止 IRQ 中断，返回 0；若 cachep->flags 没有设置 CFLGS_OFF_SLAB 标志，那么 slabp 等于刚申请到内存起始地址加上 offset，再将 offset 加上 cachep->slab_size。将 slabp 的 inuse 清 0，colouroff 设置成 offset，s_mem（指向该 slab 连续存放对象空间的开始位置）设置成刚申请到的内存空间的起始地址加上 offset。将刚刚申请到的内存的每页的描述结构中的 lru.next 设置成 cachep，将 lru.prev 设置成 slabp，将 slabp 中每个对象对应的高速缓存控制变量 kmem_bufctl_t 设置成该对象本身的索引号加 1，也就是每个对象的控制变量实际上是其下一个（更高位置）对象的索引号，最后（最高位置）一个对象的控制变量设置为 BUFCTL_END，将 slabp 的 free 设置为 0，表示下次以如果要从该 slabp 中分配对象，首先从第一个对象开始分配。如果设置了__GFP_WAIT 标志，禁止 IRQ 中断，如果每个有禁止 IRQ 中断，则系统崩溃。将 cachep 的 spinlock 上锁，将 slabp 添加到 cachep->lists->slabs_free 链表的末尾，将 cachep 的新提案 slab 计数器 grown 加 1，将 cachep->lists 的空闲对象计数器 free_objects 加上 cachep->num，将 cachep 的 spinlock 解锁并返回 1。

cache_flusharray (kmem_cache_t* cachep, struct array_cache *ac)函数执行过程：将 ac->batchcount 传给 batchcount；检查是否禁止了 IRQ 中断，如果没有，则系统崩溃。将 cachep 的 spinlock 上锁，如果 cachep 的缓存链表结构变量 lists 中的共享高速缓存内存阵列描述结构指针 shared 不是空，将 shared->limit-shared->avail 传给 max，如果 max 不等于 0，如果 batchcount 大于 max，将 max 传给 batchcount，再将 ac 中对象指针表中最开始的 batchcount 个对象指针复制到 shared 的 avail 索引的对象指针位置，将 shared 的 avail 加上 batchcount。如果 cachep 的缓存链表结构变量 lists 中的共享高速缓存内存阵列描述结构指针 shared 是空的或者 max 等于 0，调用 free_block(cachep，&ac_entry(ac)[0]，batchcount)函数：将 cachep 的 lists 中的空闲对象总数计数器 free_objects 加上 batchcount，然后一个一个按以下方式操作 ac 中最开始的 batchcount 个对象：找到对象的存放地址的页描述结构的 lru.prev 作为该对象所属的 struct slab 结构的指针 slabp，根据对象地址计算出该对象在其所属的 slabp 中的对象索引号，将该对象对应的控制变量 kmem_bufctl_t 置成 slabp->free，再将该对象的索引号传给 slabp->free，将 cachep 的活动对象计数器 num_active 减 1，将 slabp 的使用计数器 inuse 减 1，如果 slabp 的 inuse 等于 0，且 cachep 的 lists.free_objects 大于 cachep 的限制值 free_limit，那么要把这个 slab 整个废除，将 cachep 的 lists.free_objects 减去 cachep 的 num，再调用 slab_destroy()函数将 slabp 废除，如果 cachep 的 lists.free_objects 小于等于 cachep 的限制值 free_limit，则将 slabp 添加到 cachep 的 lists.slabs_free 链表的最后，如果 slabp 的 inuse 不等于 0，则将 slabp 添加到 cachep 的 lists.slabs_partial 链表的

最后。如果 STATS>0，统计 cachep 的 lists 中空闲 slab 结构链表 slabs_free 中的 slab 个数，如果统计结果大于 cachep 的 max_freeable，则将统计结果传给 cachep 的 max_freeable，也就是说 cachep 的 max_freeable 存的是其 lists 中的 slabs_free 链表中曾经的最大 slab 结构数。将 cachep 的 spinlock 解锁，将 ac 的 avail 减去 batchcount，将 ac 中 batchcount 位置开始的 ac->avail 个对象指针移到 ac 的对象指针标开始位置。

slab_destroy (kmem_cache_t *cachep, struct slab *slabp)函数执行过程：将 slabp 的 s_mem 减去 colouroff 得到该 slab 的全部对象空间的起始地址传给指针 addr，如果 cachep 的析构成员函数指针 dtor()不空，则逐次调用 dtor()成员函数将 slabp 中的每个对象删除。如果 cachep 的 flags 设置了 SLAB_DESTROY_BY_RCU 标志，通过 rcu 进程调用 kmem_rcu_free()函数来释放 slabp 的所有空间。如果 cachep 的 flags 没有设置 SLAB_DESTROY_BY_RCU 标志，则将 addr 处的 1<<cachep->gfporder 页内存逐页将描述结构的标志字 flags 清除 PG_slab 标志，将当前处理器的 per_cpu_page_states.nr_slab 减去 1<<cachep->gfporder，如果当前进程的收回状态结构指针 reclaim_state 不空，将当前进程的 reclaim_state 的 reclaimed_slab 加上 1<<cachep->gfporder，调用 free_pages()函数释放 addr 处的 1<<cachep->gfporder 页内存，如果 cachep 的 flags 设置了 SLAB_RECLAIM_ACCOUNT 标志，将全局原子数 slab_reclaim_pages 减去 1<<cachep->gfporder。如果 cachep 的 flags 设置了 CFLGS_OFF_SLAB 标志，那么 slabp 指向的 cachep->slab_size 大小的管理结构变量空间另外存放在 cachep 的 slabp_cache 所描述的高速缓存内存内，是该高速缓存内存的一个对象，所以还要调用 kmem_cache_free()函数将 slabp 指向的 cachep->slab_size 大小高速缓存内存对象释放回 cachep->slabp_cache。

enable_cpucache (kmem_cache_t *cachep)函数执行过程：根据 cachep 的 objsize 字段大小设置限制值 limit，对象越大限制值越小，如果 objsize 大于 131072 将 limit 设置为 1，如果 objsize 大于页大小且小于等于 131072，则将 limit 设置为 8，如果 objsize 大于 1KB 且小于等于页大小，则将 limit 设置为 24，如果 objsize 大于 256B 且小于等于 1KB，则将 limit 设置为 54，如果 objsize 小于等于 256B，则将 limit 设置为 120。将共享值 shared 设置为 0；如果系统支持多处理器，那么如果 objsize 小于等于页大小 PAGE_SIZE，那么将共享值 shared 设置为 8。将 struct ccupdate_struct 结构变量 new 全部清 0，初始化 new 中每个处理器对应的高速缓存阵列描述结构 truct array_cache 变量指针 snew.new[cpu_id]。如果处理器 cpu_id 在线，将(limit+1)/2 作为 batchcount，调用 alloc_arraycache(cpu_id，limit，batchcount)函数创建高速缓存阵列描述结构 new.new[cpu_id]：计算 new.new[cpu_id]整个空间的大小为 limit* sizeof(void*)（存放对象指针的空间，紧随 new.new[cpu_id]所指的高速缓存阵列描述结构的后面）加上高速缓存阵列描述结构本身的大小传给 memsize。从 malloc_sizes[]数组中找到其 cs_size 刚好大于等于 memsize 的 struct cache_sizes 结构变量中的 cs_cachep/cs_dmacachep，然后调用 kmem_cache_alloc_node (cs_cachep/cs_dmacachep,cpu_to_node(cpu_id))（该函数在不支持 CONFIG_NUMA 的系统中就相当于 kmem_cache_alloc(cs_cachep/cs_dmacachep，GFP_KERNEL)函数）函数为 new.new[cpu_id]

空间分配高速缓存内存空间，这部分只有 cpu_id 等于-1 时才执行。如果分配失败，则调用 kmalloc()函数为 new.new[cpu_id]分配 memsize 大小的内存空间。将 new.new[cpu_id]的 avail 设置成 0，将 limit 设置成 limit，将 batchcount 设置成 batchcount，将 touched 设置成 0。如果处理器 cpu_id 不在线，将 new.new[cpu_id]设置为空，将 cachep 传给 new.cachep。如果系统的 IRQ 中断被关闭，则系统崩溃，禁止抢占，将系统 IRQ 中断关闭，将系统中所有处理器对应的 cachep->array[cpu_id]指针与 new.new[cpu_id]指针对调。使能 IRQ 中断，使能抢占，如果 IRQ 中断被关闭，则系统崩溃，将 cachep 的 spinlock 上锁，将 cachep 的 batchcount 也设置成 batchcount，将 limit 也设置成 limit，将 free_limit 设置成(在线处理器数+1)*batchcount + cachep->num，将 cachep 的 spinlock 解锁。操作每个处理器对应的 new.new[cpu_id]：如果 new.new[cpu_id]指针是空的，处理下一个处理器的 new.new[cpu_id]，否则，将 cachep 的 spinlock 上锁，调用 free_block(cachep，ac_entry(new.new[cpu_id])，new.new[cpu_id].avail)函数。将 cachep 的 lists 中的空闲对象总数计数器 free_objects 加上 new.new[cpu_id].avail，然后逐个按以下方式操作 new.new[cpu_id]中最开始的 new.new[cpu_id].avail 个对象。找到对象的存放地址的页描述结构的 lru.prev 作为该对象所属的 struct slab 结构的指针 slabp，根据对象地址计算出该对象在其所属的 slabp 中的对象索引号，将该对象对应的控制变量 kmem_bufctl_t 置成 slabp->free，再将该对象的索引号传给 slabp->free，将 cachep 的活动对象计数器 num_active 减 1，将 slabp 的使用计数器 inuse 减 1，如果 slabp 的 inuse 等于 0，如果 cachep 的 lists.free_objects 大于 cachep 的限制值 free_limit，那么要把这个 slab 整个废除，将 cachep 的 lists.free_objects 减去 cachep 的 num，再调用 slab_destroy()函数将 slabp 废除，如果 cachep 的 lists.free_objects 小于等于 cachep 的限制值 free_limit，则将 slabp 添加到 cachep 的 lists.slabs_free 链表的最后，如果 slabp 的 inuse 不等于 0，则将 slabp 添加到 cachep 的 lists.slabs_partial 链表的最后，将 cachep 的 spinlock 解锁，再调用 kfree()函数释放 new.new[cpu_id]占的空间（大小为 struct array_cache 本身大小加上后面的对象指针空间），再次调用 alloc_arraycache(-1，batchcount*shared，0xbaadf00d)函数创建共享高速缓存阵列描述结构 new_shared，如果创建成功，将 cachep 的 spinlock 上锁，将 new_shared 传给 cachep 的 lists.shared，如果 cachep 原来的 lists.shared 指针 old 也不是空的，那么再次调用 free_block(cachep, ac_entry(old), old->avail)将 cachep 原来的 lists.shared 指针 old 中的所有对象所在的 slab 结构一个一个添加到 cachep 的 lists.slabs_free 链表或 lists.slabs_partial 链表的末尾，最后调用 kfree()函数将 cachep 原来的 lists.shared 指针 old 所占的空间释放掉。如果以上操成功，则返回 0，否则打印使能处理器高速缓存失败告警信息。

kmalloc(size_t size, int flags)函数执行过程：如果 size 在编译时可以看作常数，那么从 malloc_size[]数组中找到其 cs_size 刚好大于等于 size 的 struct cache_sizes 结构变量 malloc_size[n]，然后调用并返回 kmem_cache_alloc(malloc_size[n].cs_dmacachep（flags 中设置了 GFP_DMA 标志）/ malloc_size[n].cs_cachep（flags 中没有设置 GFP_DMA 标志），flags)函数分配高速缓存内存对象，返回分配到的高速缓存内存对象指针。如果 size 大于 malloc_size[]数

组中所有组员的 cs_size，那么调用__you_cannot_kmalloc_that_much()函数。在 malloc_size[]数组中最大的 cs_size 在支持 CONFIG_LARGE_ALLOCS 系统中可以达到 32MB（MAX_GFP_ORDER 等于 13 即 213 页），在不支持 CONFIG_MMU 的系统中可以达到 1MB（MAX_GFP_ORDER 等于 8），在支持 CONFIG_MMU 的系统中也可以达到 128KB（MAX_GFP_ORDER 等于 5），所以调用 kmalloc()分配内存时要注意这几种不同系统中 malloc_size[]数组中最大的 cs_size 大小，不要申请超过最大的 cs_size 大小的内存。如果 size 在编译时不能看作常数，调用并返回__kmalloc(size，flags)函数，也就是说此时 kmalloc()就相当于__kmalloc()函数，__kmalloc(size，flags)函数也是从 malloc_size[]数组中找到其 cs_size 刚好大于等于 size 的 struct cache_sizes 结构变量 malloc_size[n]，然后调用并返回__cache_alloc(malloc_size[n].cs_dmacachep（flags 中设置了 GFP_DMA 标志）/ malloc_size[n].cs_cachep（flags 中没有设置 GFP_DMA 标志），flags)函数（事实上__cache_alloc()函数与 kmem_cache_alloc()函数等价）来分配高速缓存内存对象，返回分配到的高速缓存内存对象指针。如果 size 大于 malloc_size[]数组中所有组员的 cs_size，返回空指针。从以上分析看出，kmalloc(size，flags)在某种程度上就相当于 kmem_cache_alloc(malloc_size[n].cs_dmacachep（flags 中设置了 GFP_DMA 标志）/ malloc_size[n].cs_cachep（flags 中没有设置 GFP_DMA 标志），flags)。

kfree(const void *objp)函数执行过程：如果 objp 指针空，则返回。禁止 IRQ 中断并保护现场，调用调试用函数 kfree_debugcheck()，将 objp 所指地址的内存页描述结构中的 lru.next 作为该 objp 所在的 kmem_cache_t 结构变量指针 cachep，调用__cache_free(cachep，objp)函数释放 objp 所指高速缓存内存对象，使能 IRQ 中断并恢复现场。从以上分析可以看出 kfree(const void *objp)函数在某种程度上就相当于 kmem_cache_free(GET_PAGE_CACHE(virt_to_page(objp))，objp)。

__get_free_pages(unsigned int gfp_mask, unsigned int order)函数执行过程：直接调用 alloc_pages(gfp_mask, order)申请 1<<order 页系统内存，如果申请失败，则返回空指针；如果成功则调用 page_address(page)函数将申请到的内存首页的描述结构指针转换成首页起始地址并返回。alloc_pages(unsigned int gfp_mask, unsigned int order)函数有两种定义，在支持 CONFIG_NUMA 的系统中的定义执行过程是：如果 order 大于 MAX_ORDER，返回空指针，否则调用 alloc_pages_current(gfp_mask, order)函数申请内存；在不支持 CONFIG_NUMA 的系统中的定义就相当于 alloc_pages_node(numa_node_id()（在不支持 CONFIG_NUMA 的系统中 numa_node_id()等于 0），gfp_mask, order)函数。

alloc_pages_current(unsigned gfp, unsigned order)函数执行过程：将当前进程的内存访问政策描述结构 struct mempolicy 变量指针 current->mempolicy 传给 pol，如果 pol 指针空，或者系统正在进行硬件 IRQ 中断处理或软中断处理，再将默认的内存访问政策描述结构全局变量 default_policy 的地址传给 pol。如果 pol 的内存访问政策类型 policy 是插入模式

MPOL_INTERLEAVE，调用 alloc_page_interleave(gfp, order, interleave_nodes(pol))函数分配内存。如果 pol 的内存访问政策类型 policy 是其他模式，那么调用__alloc_pages(gfp, order, zonelist_policy(gfp, pol))函数分配内存。

alloc_pages_node (int nid, unsigned int gfp_mask, unsigned int order)函数执行过程：如果 order 大于等于 MAX_ORDER，则返回空指针，否则调用并返回__alloc_pages(gfp_mask, order, NODE_DATA(nid)->node_zonelists + (gfp_mask & GFP_ZONEMASK))函数。

alloc_page_interleave(unsigned gfp, unsigned order, unsigned nid)函数执行过程：如果内存节点 nid 不在线（即 node_online_map 的第 nid 位为 0），则系统崩溃，将 nid 内存节点中指定的（NORMAL/DMA/HIGH）页区列表结构指针 NODE_DATA(nid)->node_zonelists + (gfp & GFP_ZONEMASK)传给 struct zonelist 结构指针 zl，然后调用__alloc_pages(gfp, order, zl)分配内存，如果分配成功，并且分配到的内存的首页在 zl 表中的第一个页区中，那么将 zl 表中的第一个页区描述结构中当前处理器的内存页组结构 pageset[get_cpu()]的 interleave_hit 计数器加 1，最后返回__alloc_pages()函数的返回值。

__alloc_pages(unsigned int gfp_mask, unsigned int order, struct zonelist *zonelist)函数执行过程：如果当前进程的优先级变量 prio 小于 MAX_RT_PRIO 并且系统当前不是在软中断或硬件中断处理中，或者 gfp_mask 中没有设置__GFP_WAIT 标志，那么将 1 传给局部标志变量 can_try_harder，将 zonelist 的 zones 传给 struct zone 结构指针的指针变量 zones。如果 zones[0]指针空，也就是说 zonelist 链表中没有页区，直接返回空指针，将 zones[0]页区在其所在的内存节点中的类型值传给 alloc_type（DMA 页区为 0，NORMAL 页区为 1，HIGH_MEM 页区为 2），zones 中其他页区的类型号肯定也是 alloc_type，因为页区列表结构中的所有页区的页区类型都相同。对页区指针数组 zones []中所有不空的页区指针 z 作以下操作：计算如果从 z 所指页区中分配所需内存该页区至少需要的内存页数为 z->pages_low + (1<<order) + z->protection[alloc_type]，将其传给 min，如果页区 z 中的内存总页数小于 min，查看下一个有效的内存页区，如果内存总页数大于等于 min 则调用 buffered_rmqueue(z, order, gfp_mask)函数从该页区中分配空闲内存，如果分配失败继续查看下一个有效的内存页区。如果上面从页区列表 zones []中的每个内存页区中分配内存都失败，那么对页区列表 zones []中的每个页区指针 z 执行 wakeup_kswapd(z)函数唤醒每个页区中的内存置换进程。再次检查内存页区列表 zones []中的每个内存页区指针 z 并对其进行以下操作：将 z->pages_min 传给 min，如果 gfp_mask 中设置了__GFP_HIGH 标志位，那么将 min 减去其 1/2，如果 can_try_harder 不等于 0，再将 min 减去其 1/4，再将 min 加上(1<<order)和 z->protection[alloc_type]作为 min 值，如果页区 z 中的空闲页数 free_pages 小于 min，查看下一个有效的内存页区，如果 free_pages 大于等于 min，调用 buffered_rmqueue(z, order, gfp_mask)函数从该页区中分配空闲内存，如果分配失败，则继续查看下一个有效的内存页区。如果上面再次从页区列表 zones []中的每个内存页区中分配内存都失败，且当前进程 current 的标志字 flags 中设置了 PF_MEMALLOC 位或 PF_MEMDIE 位并

且系统当前不是在硬件中断处理和软件中断处理中，那么再次对页区列表 zones []中的每个页区指针 z 直接执行 buffered_rmqueue(z, order, gfp_mask)函数分配空闲内存，如果再次全部分配内存失败，如果 gfp_mask 没有设置__GFP_NOWARN 并且 printk_ratelimit()执行正确，打印告警信息，返回空指针。如果当前进程 current 的标志字 flags 中没有设置 PF_MEMALLOC 位或 PF_MEMDIE 位，或者系统正处在硬件中断或软件中断处理中，并且 gfp_mask 中也没有设置__GFP_WAIT 标志，那么如果 gfp_mask 没有设置__GFP_NOWARN 并且 printk_ratelimit()执行正确，打印告警信息，返回空指针。如果当前进程 current 的标志字 flags 中没有设置 PF_MEMALLOC 位或 PF_MEMDIE 位，或者系统正处在硬件中断或软件中断处理中，但 gfp_mask 中设置了__GFP_WAIT 标志，则进入下面的系统平衡操作循环 rebalance：将 current->flags 或上 PF_MEMALLOC，将局部 struct reclaim_state 结构变量 reclaim_state 清 0，将 p->reclaim_state 指向 reclaim_state，调用 try_to_free_pages(zones, gfp_mask, order)函数，将 p->reclaim_state 指针设置为空，清除 current->flags 中的 PF_MEMALLOC，再次检查内存页区列表 zones []中的每个内存页区指针 z 并对其进行以下操作：将 z->pages_min 传给 min，如果 gfp_mask 中设置了__GFP_HIGH 标志为，那么将 min 减去其 1/2，如果 can_try_harder 不等于 0，再将 min 减去其 1/4，再将 min 加上(1<<order) 和 z->protection[alloc_type]作为 min 值，如果页区 z 中的空闲页数 free_pages 小于 min，查看下一个有效的内存页区，如果 free_pages 大于等于 min，调用 buffered_rmqueue(z, order, gfp_mask)函数从该页区中分配空闲内存，如果分配失败，则继续查看下一个有效的内存页区，如果以上分配内存再次全部失败，将局部变量 do_retry 设置为 0，如果 gfp_mask 没有设置__GFP_NORETRY 但设置了__GFP_NOFAIL 或__GFP_REPEAT 标志，将 do_retry 设置为 1，如果 gfp_mask 没有设置__GFP_NORETRY 且 order 小于等于 3，将 do_retry 设置为 1，如果 do_retry 不等于 0，执行 blk_congestion_wait(WRITE, HZ/50)，然后重新执行系统平衡操作循环 rebalance。上面不管哪一部分成功分配所需内存后进行如下操作：调用 zone_statistics(zonelist, z)函数更新被成功分配内存的页区 z 以及页区列表中第一个页区 zones[0]的 struct per_cpu_pageset 结构变量 pageset[smp_processor_id()]中的各种计数器的值（在 CONFIG_NUMA 的系统中），再执行 kernel_map_pages(page, 1 << order, 1)函数（该函数只有在 x86 处理器中配置了 CONFIG_DEBUG_PAGEALLOC 时才有实际定义，没有配置 CONFIG_DEBUG_PAGEALLOC 时或其他处理器中该函数是空函数），然后返回申请到的内存首页的页描述结构变量指针。

buffered_rmqueue(struct zone *zone, int order, int gfp_flags)函数执行过程：将 gfp_flags & __GFP_COLD 传给局部变量 cold，如果 order 等于 0，也就是说只分配一页内存执行以下操作：将局部处理器内存页描述结构 struct per_cpu_pages 指针 pcp 指向页区 zone 的 pageset[get_cpu()（禁止系统抢占，获取当前处理器 ID 号）].pcp[cold]，禁止系统 IRQ 中断并保存现场，如果 pcp 中的页数 count 小于最低水印 low，那么调用 rmqueue_bulk(zone, 0, pcp->batch, &pcp->list)函数逐页从 zone 的空闲内存中转移 pcp->batch 页内存到 pcp 的 list 链表末尾，将 pcp 的 count 加上实际转移过来的空闲页数，如果 pcp->count 大于 0，将局部页描述结构指针 page 指向 pcp 的内

存页链表 list.next 对应的内存页描述结构，将 page 从 list 链表中删除，将 pcp->count 减 1，使能系统 IRQ 中断并恢复现场，使能系统抢占。如果 page 指针空，将 zone 的 lock 上锁，关闭系统 IRQ 中断并保护现场，调用__rmqueue(zone, order)函数从页区 zone 中分配空闲内存，page 指向其返回值，将 zone 的 lock 解锁，使能系统 IRQ 中断并恢复现场。如果 page 指针不空，但 page 所指内存地址不在页区 zone 的范围内，则系统崩溃，更新当前处理器的 struct page_state 结构全局变量 per_cpu_page_states 中 zone 所属页区类型（DMA/NORMAL/HIGH_MEM）对应的分配页数计数器 pgalloc_dma（pgalloc_normal/pgalloc_high），如果 page 中的地址空间结构 struct address_space 指针 mapping 不空或者影射计数器_mapcount 大于等于 0 或者 flags 中设置了 PG_private、PG_locked、PG_lru、PG_active、PG_dirty、PG_reclaim、PG_swapcache、PG_writeback 中的某位或某些位，那么该页就是错误页，打印错误信息，提示重启系统以解决问题，将 flags 中的以上位全部清 0，将_count 设置为-1，将 mapping 设置为 NULL，将系统错误标志 tainted 或上内存页错误标志位 TAINT_BAD_PAGE，清除 flags 中的 PG_uptodate、PG_error、PG_referenced、PG_arch_1、PG_checked、PG_mappedtodisk 标志位，将 page 的私有数据地址（指针）private 设置为 0，如果系统支持 CONFIG_MMU，只将申请到的首页的页描述结构 page 的引用原子计数器_count 设置为 0，如果系统不支持 CONFIG_MMU，则将申请到的全部 1<<order 页内存的页描述结构变量的引用原子计数器_count 都设置为 0，如果 order 大于 0，并且 gfp_flags 中设置了组合页标志__GFP_COMP，将申请到的内存的第二页的页描述结构变量 page[1]的地址空间结构指针 mapping 设置为空，将 page[1]的索引号 index 设置为 order，再将申请到的全部 1<<order 页内存的页描述结构变量的 flags 或上 PG_compound 标志，私有数据地址（指针）private 就指向该页描述结构变量本身，最后返回申请到的内存空间首页页描述结构变量指针 page。

rmqueue_bulk(struct zone *zone, unsigned int order, unsigned long count, struct list_head *list)函数执行过程：将页区 zone 的 lock 上锁，关闭系统 IRQ 中断并保护现场。循环 count 次执行以下操作：调用__rmqueue(zone, order)申请 1<<order 页内存，如果申请失败，跳出循环，如果申请成功，将申请成功计数器 allocated 加 1，再将申请到的内存首页也描述结构变量 page 连接到循环链表 list 的末尾。将页区 zone 的 lock 解锁，使能系统 IRQ 中断并恢复现场，返回申请成功计数器的值。

__rmqueue(struct zone *zone, unsigned int order)函数执行过程：循环针对从 order（包含）到 MAX_ORDER（不包含）之间的每个 order 值 order_x 进行如下操作：将 zone->free_area + order_x 传给局部 struct free_area 结构指针 area，如果 area 中空闲内存区域首页描述结构链表 free_list 中没有有效成员即链表是空的，查看下一个 order_x 值，执行下一次循环，如果 free_list 链表中有有效成员，将 free_list.next 对应的页描述结构变量指针传给 page，将 page 从其所在链表即 free_list 中删除，计算 page 相对于页区 zone 中起始页的偏移量即 page 页在页区 zone 中的索引号 page - zone->zone_mem_map 传给 index，如果 order_x 不等于 MAX_ORDER-1，将 area 中 page 页所在空闲区域的伙伴位码即 area->map 所指空间的第 index>>（1+order）位的值取反，

将页区 zone 中的空闲内存页计数器 free_pages 减去 1<<order，如果最后成功分配的 order_x 大于 order，因为事实只需要分配 1<<order 页空闲内存，而分配该内存的空闲内存区域有 1<<order_x 页，所以要将该空闲内存区域中的剩余空闲内存添加到从 order 到 order_x-1 对应的空闲区域中去，即：将 1<<order_x 传给 size，只要 order_x 大于 order 一直循环执行：将 area 减 1 得到 order_x-1 对应的空闲内存区域描述结构指针传给 area，将 order_x 减 1 传给 order_x，将 size>>1 传给 size，如果 page[size]所指的内存页（该页是 order_x-1 对应的某个空闲内存区域的起始页）没在页区 zone 的范围内，则系统崩溃，否则将 page[size]添加到 area 的空闲内存区域首页也描述结构链表中，再将 area 中 page[size]页所在空闲区域的伙伴位码即 area->map 所指空间的第（index+size）>>（1+order_x）位的值取反，返回申请到的连续空闲内存空间首页的页描述结构指针 page。如果以上全部循环操作都未成功分配到所需内存，则返回空指针。

wakeup_kswapd(struct zone *zone)函数执行过程：如果页区 zone 中的实际内存页数（不包括孔洞页数）present_pages 等于 0，则返回。如果页区 zone 中的空闲内存页数 free_pages 大于最低页数水印值 pages_low，返回；如果页区 zone 所在内存节点的描述结构 zone_pgdat 的内存置换等待队列链表结构 kswapd_wait 中的进程链表是空的，返回；调用 wake_up_interruptible (&zone->zone_pgdat->kswapd_wait)函数唤醒等待队列链表 kswapd_wait 中的进程。

wake_up_interruptible(wait_queue_head_t *q)函数执行过程：该函数是一个宏，被定义成 __wake_up(wait_queue_head_t *q, TASK_INTERRUPTIBLE, 1, NULL)，__wake_up(wait_queue_head_t *q, unsigned int mode, int nr_exclusive, void *key)函数执行过程如下：将等待队列链表结构 q 的 lock 上锁，禁止 IRQ 中断并保存现场。调用__wake_up_common(q, mode, nr_exclusive, 0, key)函数，将等待队列链表结构 q 的 lock 解锁，使能 IRQ 中断并恢复现场。__wake_up_common (wait_queue_head_t *q, unsigned int mode, int nr_exclusive, int sync, void *key)函数执行过程如下：对等待队列链表结构 q 的链表 task_list 中的每个等待队列结构（wait_queue_t）指针 curr 进行如下操作：执行等待队列 curr 中的处理函数 func（curr, mode, sync, key），将 curr->flags 与 WQ_FLAG_EXCLUSIVE 相与，将 nr_exclusive 减 1，如果前两步操作结果为真，第三步操作结果为 0，则退出循环，否则继续处理链表 q->task_list 中的下一个等待队列。

try_to_free_pages(struct zone **zones, unsigned int gfp_mask, unsigned int order)函数执行过程：将当前进程 current 的回收状态结构指针 reclaim_state 传给局部变量 reclaim_state，将扫描控制结构 struct scan_control 局部变量 sc 的 gfp_mask 设置为 gfp_mask，将 may_writepage 设置为 0，将当前处理器的页状态描述结构变量 per_cpu_page_states 的直接收回计数器 allocstall 加 1，将页区列表 zones 中所有页区的 temp_priority 都设置为 DEF_PRIORITY（=12），并统计 zones 中所有页区的活动页计数器 nr_active 与非活动页计数器 nr_inactive 的总和传给局部变量 lru_pages，将优先级局部变量 priority 循环设置成从 DEF_PRIORITY（包含）到 0（包含），对每次循环作如下操作：统计系统中所有可能处理器的页状态描述结构变量 per_cpu_page_states 中的已经进行虚拟映射的内存页数计数器 nr_mapped 的总和传给扫描控制结构的

sc.nr_mapped，将 sc 的 nr_scan 和 nr_reclaimed 都设置为 0，将 sc 的 priority 设置为 priority，执行 shrink_caches(zones, &sc)函数查看系统中每个内存页区，如果该页区没有设置所有页不能回收标志，或者扫描优先级别不等于默认级别，如果该页区的活动或者非活动页链表中的页数大于等于 SWAP_CLUSTER_MAX，那么从页区 zone 中删除 sc->nr_to_scan 数目的内存页，如果活动页链表中的页数大于 SWAP_CLUSTER_MAX，先将适当的活动页移动到非活动页链表中，然后从非活动页链表中删除所需页数，再执行 shrink_slab(sc.nr_scanned, gfp_mask, lru_pages)收缩 slab，如果 reclaim_state 指针不空，将 sc.nr_reclaimed 加上 reclaim_state->reclaimed_slab，将 reclaim_state->reclaimed_slab 清 0。如果 sc.nr_reclaimed 大于等于 SWAP_CLUSTER_MAX，则跳出循环，将 zones 中的每个页区的 prev_priority 设置为其 temp_priority 的值，然后返回 1，如果 sc.nr_reclaimed 小于 SWAP_CLUSTER_MAX，将扫描总数计数器局部变量 total_scanned 加上 sc.nr_scanned，将回收总数计数器局部变量 total_reclaimed 加上 sc.nr_reclaimed，如果 total_scanned 大于 SWAP_CLUSTER_MAX 的 1.5 倍，执行 wakeup_bdflush(laptop_mode ? 0 : total_scanned)函数唤醒 bdflush()任务进程，再将 sc.may_writepage 设置为 1，如果 sc.nr_scanned 不等于 0 且 priority 小于 DEF_PRIORITY-2，执行 blk_congestion_wait(WRITE, HZ/10)，然后将 priority 设置成下一个优先级，执行下一次循环操作。如果正常完成全部循环而没有中途跳出，且 gfp_mask 设置了__GFP_FS 但没有设置__GFP_NORETRY 标志，执行 out_of_memory(gfp_mask)函数，将 zones 中的每个页区的 prev_priority 设置为其 temp_priority 的值，然后返回 0。

out_of_memory(int gfp_mask)函数执行过程：将系统当前的 jiffies 传给局部变量 now，将 now 减去静态变量 last 传给 since，将 now 传给 last；如果 since 不大于 5*HZ，将 now 减去静态 first 传给 since，如果 since 不小于 HZ，将静态变量 count 加 1，如果 count 不小于 10，将 now 减去静态变量 last_kill 传给 since，如果 since 不小于 5*HZ，将 now 传给 lastkill，调用 show_free_areas()函数打印系统中空闲内存区域链表信息。调用 oom_kill()函数杀掉最坏的进程。如果 now 大于静态变量 first，将 now 传给 first，将静态变量 count 清 0。

shrink_caches(struct zone **zones, struct scan_control *sc)函数执行过程：将页区列表 zones 中的第一个页区描述结构指针 zones[n]传给页区描述结构指针 zone，如果 zone 的实际内存页数 present_pages 等于 0，重复以上操作处理 zones 中下一个页区，将扫描控制结构 sc 的级别 priority 传给页区 zone 的临时级别 temp_priority，如果 zone 的预设级别 prev_prioprity 大于 sc->priority，将 sc->priority 传给 zone->prev_priority，如果 zone->all_unreclaimable（表示该页区中的页都不能被回收）不等于 0 并且 sc->priority 不等于默认级别 DEF_PRIORITY，重复以上操作处理 zones 中下一个页区，执行 shrink_zone(zone, sc)函数回收 zone 中的内存，重复以上操作处理 zones 中下一个页区，直到 zones 中最后一个页区。

static void shrink_zone(struct zone *zone, struct scan_control *sc)函数执行过程：将 zone->nr_scan_active 加上(zone->nr_active >> sc->priority) + 1，将 zone->nr_scan_active 传给 nr_active。如果 nr_active 大于等于 SWAP_CLUSTER_MAX（=32），将 zone->nr_scan_active 设置为 0，否则

将 nr_active 设置为 0。将 zone->nr_scan_inactive 加上(zone->nr_inactive >> sc->priority) + 1；将 zone->nr_scan_inactive 传给 nr_inactive。如果 nr_inactive 大于等于 SWAP_CLUSTER_MAX，将 zone->nr_scan_inactive 设置为 0，否则将 nr_inactive 设置为 0。将 SWAP_CLUSTER_MAX 传给 sc->nr_to_reclaim。“循环开始位置”：只要 nr_active 或者 nr_inactive 不等于 0，如果 nr_active 大于 0，将 nr_activ 与 SWAP_CLUSTER_MAX 之中最小值传给 sc->nr_to_scan，将 nr_active 减去 sc->nr_to_scan，执行 refill_inactive_zone(zone, sc)将该页区活动链表 active_list 中适当数量已经不用的内存页移动（补充）到未活动链表中。如果 nr_inactive 大于 0，将 nr_inactive 与 SWAP_CLUSTER_MAX 之中最小值传给 sc->nr_to_scan，将 nr_inactive 减去 sc->nr_to_scan，执行 shrink_cache(zone, sc) 从该页区的非活动页链表 inactive_list 中释放 sc->nr_to_scan 页的内存，如果 sc->nr_to_reclaim 小于等于 0，则退出循环；“循环结束位置”。

static void refill_inactive_zone(struct zone *zone, struct scan_control *sc)函数执行过程：先将当前 CPU 的添加内存页向量描述结构（struct pagevec）per_cpu_lru_add_pvecs 和添加活动内存页向量描述结构（struct pagevec）per_cpu_lru_add_active_pvecs 中的所有内存页的描述结构分别链接到它们各自所在页区的非活动页链表 inactive_list 或活动页链表 active_list 中，然后调用 release_pages(pvec->pages, pvec->nr, pvec->cold)（pvec 指向 per_cpu_lru_add_pvecs 或 per_cpu_lru_add_active_pvecs）将所有这些内存页释放到其所在页区中页向量指定的冷或热内存区块中，再将 per_cpu_lru_add_pvecs.nr 和 per_cpu_lru_add_active_pvecs.nr 清 0；将该页区的活动内存页链表 active_list 中适当数量的已经不用并且映射计数器小于 0 的页移动到非活动内存页链表 inactive_list 中，并且释放这些内存空间。更新当前 CPU 的 per_cpu_page_states 结构变量中的 zone 对应的页区类型的补充页数计数器 pgrefill_(dma/normal/high)和由活动转为非活动计数器 pgdeactivate 的值。

void release_pages(struct page **pages, int nr, int cold)函数执行过程：将 pages 所指的 nr 页内存以 struct pagevec 结构 pvec 描述的内存页数（=14）为单位调用 pagevec_free(pvec) -> __pagevec_free(pvec) -> free_hot_cold_page(pvec->pages[i],pvec->cold)分批释放。

static void shrink_cache(struct zone *zone, struct scan_control *sc)函数执行过程：将当前 CPU 的添加内存页向量描述结构 per_cpu_lru_add_pvecs 和添加活动内存页向量描述结构 per_cpu_lru_add_active_pvecs 中的所有内存页的描述结构分别链接到它们各自所在页区的非活动页链表 inactive_list 或活动页链表 active_list 中，然后将所有这些内存页空间释放到其所在页区中页向量指定的冷或热内存区块中，再将 per_cpu_lru_add_pvecs.nr 和 per_cpu_lru_add_active_pvecs.nr 清 0。将页区 zone 的 inactive_list 链表中也描述结构引用计数器_count 不等于-1 的内存页从 inactive_list 链表中删除并加入到临时链表 page_list 中。如果当前进程是置换进程 kswapd()（进程标志字 flags 设置了 PF_KSWAPD 位），那么将当前处理器的 per_cpu_page_states.pgscan_kswapd_(dma/normal/high) 加上 page_list 链表中的内存页数 nr_scan，否则将 per_cpu_page_states.pgscan_direct_(dma/normal/high)加上 nr_scan。执行 shrink_list(&page_list, sc)

回收 page_list 链表中的内存并把实际回收的页数传给 nr_free。如果当前进程是置换进程 kswapd()，将当前处理器的 per_cpu_page_states.kswapd_steal 加上 nr_free，将当前处理器的 per_cpu_page_states.pgsteal_(dma/normal/high)加上 nr_free，将 zone->lru_lock 上锁，处理 page_list 链表中不能被释放的内存页，设置页描述结构标志字 flags 中的 PG_lru 位，将 flags 设置了 PG_active 位的页重新加入到页区的 active_list 链表中，未设置 PG_active 位的加入到页区的 inactive_list 链表中，同时将这些没有释放的页加入到 struct pagevec 结构 pvec 中，最后调用 __pagevec_release(&pvec)->release_pages(pvec->pages, pagevec_count(pvec), pvec->cold)或 pagevec_release(&pvec)-> __pagevec_release(&pvec)->release_pages(pvec->pages, agevec_count(pvec), pvec->cold)释放它们。

static int shrink_list(struct list_head *page_list, struct scan_control *sc)函数执行过程：如果 current->thread_info->flags 设置了 TIF_NEED_RESCHED 标志位，将 current->state 设置为 TASK_RUNNING，使系统 cache 和 Write Buffer 无效，执行 schedule()重新调度进程。“循环开始位置”：只要 page_list 链表不是空的，将链表中的页描述结构指针传给 page，将 page 从 page_list 链表中删除。如果 page->flags 设置了 PG_locked 位，跳转到“keep 标号位置”处。如果 page->flags 设置了 PG_writeback 位，跳转到“keep_locked 标号位置”处。将 sc->nr_scanned 加 1，如果 page->_mapcount 大于等于 0 或者 page->flags 设置了 PG_swapcache 标志位，将 sc->nr_scanned 再加 1。执行 page_referenced(page, 1, sc->priority <= 0)统计引用该页的数量并将返回值传给 referenced。如果 referenced 不等于 0 并且该页的映射仍在被使用即 page_mapping_inuse(page)不等于 0，跳转到“activate_locked 标号位置”处。

long blk_congestion_wait(int rw, long timeout)函数执行过程：声明并初始化等待队列结构 wait_queue_t 变量 wait。将局部等待队列链表结构指针 wqh 指向系统拥塞等待队列链表结构数组中的 congestion_wqh[rw]，清除 wait.flags 中的 WQ_FLAG_EXCLUSIVE 标志。将 wqh 的 lock 上锁，禁止 IRQ 中断并保护现场，如果 wait.task_list 链表中是空的，即 wait 还没有加入任何链表中，那么将 wait 通过其 task_list 连接到等待队列链表结构 wqh 的 task_list 链表中去，如果等待队列结构指针&wait 空或等待队列结构 wait 中的任务指针 task 不空，那么将当前进程的状态 current->state 设置为 TASK_UNINTERRUPTIBLE，即将当前进程放进等待队列中，等待资源有效时才唤醒，不能被其他进程通过信号量或定时中断唤醒，将 wqh 的 lock 解锁，使能 IRQ 中断并恢复现场。将当前处理器的运行队列结构变量 per_cpu_runqueues 的 io 等待原子计数器 nr_iowait 加 1，执行 schedule_timeout(timeout)函数并将其返回值传给 ret，再将 per_cpu_runqueues 的 io 等待原子计数器 nr_iowait 减 1。执行 finish_wait(wqh, &wait)函数，返回 ret。

17.5.27 numa_policy_init()函数

该函数初始化非统一内存放问系统中的内存访问政策，CONFIG_NUMA 一般用于多处理器系统，在不支持多处理器的系统中该函数是空函数。

在支持非议执行访问的系统中的执行过程说明如下：

该函数在不支持 CONFIG_NUMA 的系统中是空函数，在支持 CONFIG_NUMA 的系统中才有意义。为内存访问政策结构 struct mempolicy 创建一个高速缓存内存描述结构 kmem_cache_t 变量，命名为“numa_policy”，未定义其中的构造函数 ctor 和析构函 dtor，将全局高速缓存内存描述结构指针 policy_cache 指向该变量，为 struct sp_node 结构创建一个高速缓存内存描述结构 kmem_cache_t 变量，命名为“shared_policy_node”，未定义其中的构造函数 ctor 和析构函 dtor，将全局高速缓存内存描述结构指针 sn_cache 指向该变量。执行 sys_set_mempolicy(MPOL_INTERLEAVE, nodes_addr(node_online_map), MAX_NUMNODES)函数，如果返回错误，打印设置插入模式失败信息。

sys_set_mempolicy(int mode, unsigned long __user *nmask, unsigned long maxnode)函数执行过程：如果 mode 大于 MPOL_MAX（=MPOL_INTERLEAVE），则返回无效错误；如果 maxnode-1>MAX_NUMNODES，说明系统中还有不支持的内存节点，如果 maxnode-1 个内存节点的节点位掩码占的空间大于 1 页内存，返回无效错误，将所有内存节点的位掩码字空间从用户空间的 nmask 复制到内核空间的位掩码字空间局部指针 nodes，如果复制失败，则返回失败错误，如果从 MAX_NUMNODES 到 maxnode-1 号内存节点（这些内存节点系统不支持）的位掩码的值有不等于 0 的，则返回无效错误，如果内存访问政策模式 mode 是默认模式 MPOL_DEFAULT 而系统支持的 MAX_NUMNODES 个内存节点的位掩码并不全是 0，则返回无效错误，如果内存访问政策模式 mode 是绑定模式 MPOL_BIND 或插入模式 MPOL_INTERLEAVE 而系统支持的 MAX_NUMNODES 个内存节点的位掩码却全是 0 则返回无效错误，如果系统支持的 MAX_NUMNODES 个内存节点在在线内存节点位掩码字空间 node_online_map 中对应的位都是 0，说明系统当前没有内存节点在线，那么将 node_online_map 中的 0 位设置为 1，也就是将 0 号节点设置为在线，如果 node_online_map 与 nodes 的位掩码字空间的设置不完全相同，返回无效错误。调用 mpol_new(mode, nodes)函数创建一个新的内存访问政策描述结构 struct mempolicy 变量并将其地址传给局部 struct mempolicy 结构指针 new。如果 new 离系统允许的理论最高地址只差不到 1000，new 地址范围出错，返回错误指针 new 的值。调用 mpol_free(current->mempolicy)函数将当前进程的内存访问政策结构变量释放掉，再将新创建的内存访问政策结构指针 new 传给 current->mempolicy。如果 new 指针不为空且 new->policy 等于插入政策编号 MPOL_INTERLEAVE，将当前进程 current 的 il_next 设置为访问政策描述结构 new 中的内存节点掩码字空间 v.nodes 中第一位为 1 的位号，返回 0。

mpol_new(int mode, unsigned long *nodes)函数执行过程：如果 mode 等于默认访问模式 MPOL_DEFAULT，则返回空指针。调用 kmem_cache_alloc(policy_cache, GFP_KERNEL)从上面刚创建的 struct mempolicy 结构高速缓存内存描述结构 policy_cache 中申请 struct mempolicy 结构高速缓存内存对象并将申请到的对象指针传给局部 struct mempolicy 结构指针 policy。如果 policy 为空，返回没有足够内存错误，将 policy 的引用原字计数器 refcnt 设置为 1。如果 mode

等于插入模式 MPOL_INTERLEAVE，将内存节点位掩码字空间复制到 policy->v.nodes 中。如果 mode 等于首选模式 MPOL_PREFERRED，则将 policy->v.preferred_node 设置为位掩码字空间 nodes 中第一位为 1 的位号，如果 policy->v.preferred_node 大于 MAX_NUMNODES，将 policy->v.preferred_node 设置为-1。如果 mode 等于绑定模式 MPOL_BIND，将位掩码字空间 nodes 中所有为 1 的内存节点的所有实际存在的内存页数 present_pages 不为 0 的内存 zone 都添加到 policy 的内存页区列表 v.zonelist 中，如果 policy->v.zonelist 指针空，调用 kmem_cache_free(policy_cache, policy)释放刚申请到的 struct mempolicy 结构高速缓存对象 policy，返回没有足够内存错误，将 policy->mode 设置为 mode，返回 policy 指针。

mpol_free(struct mempolicy *pol)函数执行过程：该函数在不支持 CONFIG_NUMA 的系统中是空函数，在支持 CONFIG_NUMA 的系统中才有意义。如果 pol 指针不空，则调用 __mpol_free(pol)。__mpol_free(struct mempolicy *p)函数执行过程：查看并将 p 的引用原字计数器 refcnt 减 1，如果原来 refcnt 等于 0，则返回；如果 p 的 policy 等于绑定模式 MPOL_BIND，调用 kfree(p->v.zonelist)释放 p->v.zonelist 所指页区列表内存空间，将 p->policy 设置成 MPOL_DEFAULT，调用 kmem_cache_free(policy_cache, p)将 struct mempolicy 结构高速缓存内存对象释放回 policy_cache 中。

如果定义了后期定时器初始化函数 late_time_init()，则执行之。函数指针 late_time_init()由各处理器自行定义，只有在 ppc\platforms\sbc82xx.c 和 mips\pmc-sierra\yosemite\setup.c 及 i386\kernel\time.c 中给 late_time_init()函数指针赋了值，ARM 处理器都没定义该函数。

17.5.28 calibrate_delay()函数

该函数计算系统的 BogMIPS 数值，即处理起每秒钟之行的指令数。一般来说计算出的这个值越大，系统处理速度越快，但是不同处理器架构的嵌入式系统之间的 BogMIPS 的值不能直接用来比较衡量系统之间运行速度和能力，比如 Intel 公司 x86 系列的 Xeon2.4GHz 服务器平台的 BogMIPS 值能达到 2000 以上，而 BroadCom 公司 MIPS 架构 BCM91250 800 赫兹嵌入式服务器平台的 BogMIPS 值还不到 1000，但通过我们对两者之间的千兆网络处理能力以及大容量盘阵读写能力的综合测试结构来看，两者之间的性能是相当的，而且在网络处理能力上 BCM91250 平台更有优势。

该函数将局部变量 lps_precision 设置成 LPS_PREC（=8）。查看全局变量 preset_lpj（该变量由“lpj=xx”命令设置，如果在命令行中预先设置了“lpj=100”命令，则 preset_lpj 等于 100，如果没有设置“lpj=xx”命令，则 preset_lpj 等于 0），如果 preset_lpj 不等于 0，那么将 preset_lpj 传给全局变量 loops_per_jiffy（声明时被设置为 1<<12），然后将 loops_per_jiffy 转化成每秒系统执行的指令数（BogoMIPS 值）并打印。如果 preset_lpj 等于 0，计算系统每“嘀嗒”（click）一下处理器最多能循环执行延时函数__delay()（该函数只有两条指令）的次数传给

loops_per_jiffy，然后将 loops_per_jiffy 转化（loops_per_jiffy/(500000/HZ)）成每秒系统执行的指令数（BogoMIPS 值）并打印。

17.5.29 pidmap_init()函数

调用 get_zeroed_page(GFP_KERNEL)函数为系统进程 ID 号位掩码描述结构全局数组 pidmap_array[]的第一个结构变量申请第 1 页进程 ID 号位掩码内存空间，把申请到的内存空间的起始地址传给 pidmap_array->page。将第 1 页进程 ID 号位掩码内存空间的第 1 位（bit0）设置为 1，表示 0 号进程 ID 号已经被使用。将第 1 页进程 ID 号位掩码空间的空闲位掩码位数 pidmap_array->nr_free 减 1。循环依次查看 4 种类型 PID 对应的 hash 链表结构指针数组 pid_hash[x（0~3）]中 ID 号 0 对应的链表中是否有成员，即查看 ID 号为 0 对应的 struct pid 结构变量是否已经添加到 pid_hash[x]的链表中，也可以说是否已经有进程使用的 ID 号 0，如果 ID 号 0 尚未被使用，那么将当前进程（系统启动进程）current 的 pids[x]通过其 pid_chain 成员变量链接到 pid_hash[x][pid_hashfn(0)]链表的开始，并将 current->pids[x]从其所在的 pid_list 链表中删除即初始化其 pid_list 成员变量，然后将 current->pids[x].nr 设置成 0。如果 ID 号 0 已经被使用，那么将当前进程（系统启动进程）current 的 pids[x]通过其 pid_list 成员变量链接到 pid_hash[x][pid_hashfn(0)]链表中 ID 号 nr 等于 0 的 struct pid 结构 pid 的 pid_list 链表的末尾，并将 current->pids[x]从其所在的 pid_chain 链表中删除即初始化其 pid_chain 成员变量，然后将 current->pids[x].nr 设置成 0。

get_zeroed_page(unsigned int gfp_mask)函数执行过程：如果 gfp_mask 中设置了 __GFP_HIGHMEM 标志，则系统崩溃，调用 alloc_pages(gfp_mask, 0)函数申请一页内存空间并将其页描述结构地址传给 page。如果指针 page 不空，调用 page_address(page)将页描述结构指针转换成该页的虚拟起始地址传给 addr，再将该页空间全部清 0，返回 addr；如果指针 page 空，返回空指针。

attach_pid(task_t *task, enum pid_type type, int nr)函数执行过程：查看 type 类型 PID 对应的 hash 链表结构指针数组 pid_hash[type]中 ID 号为 nr 对应的链表 pid_hash[type][pid_hashfn(nr)]中是否已经链接了其 nr 成员变量等于传进来的 nr 的 struct pid 结构指针 pid，即查看 ID 号为 nr 的 struct pid 结构是否已经添加到 pid_hash[type] [pid_hashfn(nr)]链表中，也可以说是否已经有进程使用了 ID 号 nr，如果 nr 尚未使用即 pid 空，那么将 task->pid[type]通过其 pid_chain 成员链接到 pid_hash[type][pid_hashfn(nr)]链表的开始处，并将 task->pids[type]从其所在的 pid_list 链表中删除即初始化其 pid_list 成员变量。如果 ID 号 nr 已经被使用，即 pid 指针不空，那么先将 task->pids[type]从其所在的 pid_chain 链表中删除即初始化其 pid_chain 成员变量，再将 task->pids[type]通过其 pid_list 成员变量链接到 pid->pid_list 链表的末尾，将 task->pids[type].nr 设置成 nr。

17.5.30 pgtable_cache_init()函数

void __init pgtable_cache_init(void)函数只有在 arm26/i386/ppc64 等系统中有定义，在 ARM 等其他系统中 pgtable_cache_init(void)都被定义为空函数。

17.5.31 prio_tree_init()函数

该函数初始化无符号长整型全局数组 index_bits_to_maxindex[BITS_PER_LONG（对于 32 位系统等于 32）]的每个组员，将每个组员 index_bits_to_maxindex[n]设置成（1<<n）-1，将最后的 index_bits_to_maxindex[BITS_PER_LONG-1]设置成~0UL。

17.5.32 anon_vma_init()函数

该函数调用 kmem_cache_create("anon_vma", sizeof(struct anon_vma), 0, SLAB_DESTROY_BY_RCU|SLAB_PANIC, anon_vma_ctor, NULL)函数，为匿名虚拟内存区域链表结构 struct anon_vma 创建高速缓存内存描述结构 kmem_cache_t 变量，为该变量命名为“anon_vma”，其对象的构造函数指针指向 void anon_vma_ctor(void *data, kmem_cache_t *cachep, unsigned long flags)函数，析构函数指针空，将创建的 kmem_cache_t 结构变量地址传给全局指针 anon_vma_cachep。

如果配置了 CONFIG_X86 选项即 i386 系统而且 efi_enabled 不等于 0，则执行 efi_enter_virtual_mode()函数。所谓的 EFI 的意思是：Extensible Firmware Interface，即可扩展固件接口。

17.5.33 fork_init(num_physpages)函数

该函数执行进程创建相关初始化。

如果没有定义__HAVE_ARCH_TASK_STRUCT_ALLOCATOR 宏（只有 asm-ia64 处理器中定义了该宏，其他处理器都没有定义该宏），调用 kmem_cache_create("task_struct", sizeof(struct task_struct), ARCH_MIN_TASKALIGN, SLAB_PANIC, NULL, NULL)函数为任务描述结构 struct task_struct 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“task_struct”，没有定义其对象构造和析构函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 task_struct_cachep，如果也没定义 ARCH_MIN_TASKALIGN 宏（ppc/i386 系统中该宏被定义成 16，mips 系统中被定义成 8，arm 系统中未定义，在此定义），那么将 ARCH_MIN_TASKALIGN 定义为 L1_CACHE_BYTES，即处理器硬件 cache line 大小（字节数）。将系统实际物理内存总页数（不包括内存孔洞）num_physpages 除以每个线程空间所占内存页数（THREAD_SIZE / PAGE_SIZE）再除以 8 得到系统支持的最大线程数传给全局变量

max_threads，如果计算出的 max_threads 小于 20，将 max_threads 设置为 20，即 Linux 系统中至少支持 20 个线程；将初始任务（系统启动进程）描述结构 init_task 中的信号描述结构的最大进程数限制描述结构 rlim[RLIMIT_NPROC]的当前限制值 rlim_cur 设置为最大线程数 max_threads 的一半。将初始任务（系统启动进程）描述结构 init_task 中的信号描述结构的最大进程数限制描述结构 rlim[RLIMIT_NPROC]的最大限制值 rlim_max 也设置为最大线程数 max_threads 的一半。

17.5.34 proc_caches_init()函数

调用 kmem_cache_create("sighand_cache", sizeof(struct sighand_struct), 0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL, NULL)函数为信号处理描述结构 struct sighand_struct 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“sighand_cache”，没有定义其对象构造和析构函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 sighand_cachep。

调用 kmem_cache_create("signal_cache", sizeof(struct signal_struct), 0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL, NULL)函数为信号描述结构 struct signal_struct 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“signal_cache”，没有定义其对象构造和析构函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 signal_cachep。

调用 kmem_cache_create("files_cache", sizeof(struct files_struct), 0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL, NULL)函数为进程打开的文件表描述结构 struct files_struct 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“files_cache”，没有定义其对象构造和析构函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 files_cachep。

调用 kmem_cache_create("fs_cache", sizeof(struct fs_struct), 0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL, NULL)函数为文件系统描述结构 struct fs_struct 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“fs_cache”，没有定义其对象构造和析构函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 fs_cachep。

调用 kmem_cache_create("vm_area_struct", sizeof(struct vm_area_struct), 0, SLAB_PANIC, NULL, NULL)函数为虚拟内存区域描述结构 struct vm_area_struct 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“vm_area_struct”，没有定义其对象构造和析构函数，不要求其对象按处理器硬件 cache line 大小对齐，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 vm_area_cachep。

调用 kmem_cache_create("mm_struct", sizeof(struct mm_struct), 0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL, NULL)函数为内存管理描述结构 struct mm_struct 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“mm_struct”，没有定义其对象构造和析构函数，不要求其对象按处理器硬件 cache line 大小对齐，将创建好的 kmem_cache_t 结构变量的地址传

给全局指针 mm_cachep。

17.5.35 buffer_init()函数

该函数调用 kmem_cache_create("buffer_head", sizeof(struct buffer_head), 0, SLAB_PANIC, init_buffer_head, NULL)函数为缓冲区描述结构 struct buffer_head 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“buffer_head”，其对象构造函数指针指向 static void init_buffer_head(void *data, kmem_cache_t *cachep, unsigned long flags)函数，没有定义其对象析构函数，不要求其对象按处理器硬件 cache line 大小对齐，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 bh_cachep。统计系统中所有内存节点的通用（NORMAL）内存页区中高页数水印值页数之外的额外内存总页数传给局部变量 nrpages，计算每页内存能存放缓冲区描述结构 struct buffer_head 的个数 PAGE_SIZE / sizeof(struct buffer_head)，再计算用 nrpages 的 1/10 来存放 struct buffer_head 结构看总共能存放多少个 struct buffer_head 结构，把计算的结果传给全局变量 max_buffer_heads。按从高到低的优先级顺序将通知块结构变量 buffer_cpu_notify_nb 注册到 CPU 通知者结构变量链表 cpu_chain 中。

17.5.36 unnamed_dev_init()函数

直接调用并返回 idr_init(&unnamed_dev_idr)函数。void idr_init(struct idr *idp)执行过程如下：

如果 struct idr_layer 结构高速缓存内存描述结构指针 idr_layer_cache 是空的，调用 kmem_cache_create("idr_layer_cache",sizeof(struct idr_layer), 0, 0, idr_cache_ctor, NULL)函数为结构 struct mm_struct 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“idr_layer_cache”，其对象构造函数指针指向 static void idr_cache_ctor(void * idr_layer, kmem_cache_t *idr_layer_cache, unsigned long flags)函数。没有定义其对象析构函数，不要求其对象按处理器硬件 cache line 大小对齐，创建失败系统也不 panic()，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 idr_layer_cache。将 struct idr 结构 idp（在此为 unnamed_dev_idr）全部清 0，再将 idp-> lock（即 unnamed_dev_idr.lock）初始化为初始（即解锁）状态。

17.5.37 security_init()函数

打印“安全架构 v1.0.0 被初始化”。如果没有定义系统哑元安全操作函数组结构全局变量 dummy_security_ops，打印错误信息，返回 I/O 错误。查看哑元安全操作函数组结构变量 dummy_security_ops 中的每个成员操作函数指针是否为空，如果非空，则保留原来的操作函数，如果空则将该操作函数设置成对应的预先定义好的哑元操作函数即 dummy_xxx()函数。将系统当前使用的安全操作函数组结构全局指针 security_ops 指向哑元安全操作函数组结构 dummy_security_ops；分别执行一遍从__security_initcall_start 到__security_initcall_end 之间

即.security_initcall.init 区中的每个函数（包括 capability_init()函数、rootplug_init()函数和 selinux_init()函数），所有这些.security_initcall.init 区中的函数最终都要调用 register_security()函数将各自的安全操作函数组注册到系统的安全操作函数结构变量 security_ops 指针中，由于 security_ops 同一时刻只能指向一个安全操作函数组结构变量，而不能形成安全操作函数组的链表，所以事实上只有一个.security_initcall.init 区中的函数能够成功注册它的安全操作函数组，其他的函数只能等这一函数注册的安全操作函数组注销后才能注册。这样看来，.security_initcall.init 区中的每个函数代表着一种安全策略，同一时刻只能使用一种安全策略，返回 0。

int register_security(struct security_operations *ops)函数执行过程：如果安全操作函数组结构指针 ops 空，打印错误信息，返回无效错误。查看安全操作函数组结构 ops 中的每个成员操作函数指针是否为空，如果非空，则保留原来的操作函数，如果空则将该操作函数设置成对应的预先定义好的哑元操作函数即 dummy_xxx()函数，系统当前使用的安全操作函数组结构全局指针 security_ops 不是指向系统哑元安全操作函数组结构全局变量 dummy_security_ops，返回无效错误（从这里看出如果 security_ops 中的安全操作函数组结构变量不被注销，其他安全操作函数组结构变量就不能被注册到 security_ops 中去），将操作函数组结构指针 ops 指针 security_ops，返回 0。

17.5.38 vfs_caches_init(num_physpages)函数

vfs_caches_init(unsigned long mempages)函数执行过程：将（mempages 减去当前系统中的空闲物理内存总页数）乘以 1.5 与（num_physpages-1）相比较（mempages 在这里是系统中的实际物理内存总页数 num_physpages），取其小者传给局部变量 reserve。再将 mempages 减去 reserve 再传给 mempages，调用 kmem_cache_create ("names_cache", PATH_MAX, 0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL, NULL)函数为路径名字字符串（大小为 PATH_MAX=4096）创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“names_cache”，没有定义其构造和析构函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 names_cachep。调用 kmem_cache_create("filp", sizeof(struct file), 0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, filp_ctor, filp_dtor)函数为文件结构 struct file 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“filp”，其对象构造函数指针指向 void filp_ctor(void * objp, struct kmem_cache_s *cachep, unsigned long cflags)函数，对象析构函数指针指向 void filp_dtor(void * objp, struct kmem_cache_s *cachep, unsigned long dflags)函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 filp_cachep。执行 dcache_init(mempages)函数，执行 inode_init(mempages)函数，将 mempages 页内存的 1/10 转换成以 1KB（因为一个文件加上辅助的 inode 和 dcache 等数据结构差不多就是 1KB 大小）为单位的内存容量大小(mempages * (PAGE_SIZE / 1024)) / 10，然后传给系统文件状态结构变量 files_stat 的最大文件数成员变量 max_files，如果计算出的 files_stat.max_files 小于 NR_FILE（=8192），将 files_stat.max_files 设置成 NR_FILE，执行

mnt_init(mempages)函数，执行 bdev_cache_init()函数，执行 chrdev_init()函数。

dcache_init(unsigned long mempages) 函数执行过程：调用 kmem_cache_create ("dentry_cache", sizeof(struct dentry), 0, SLAB_RECLAIM_ACCOUNT|SLAB_PANIC, NULL, NULL)函数为目录条目结构 struct dentry 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“dentry_cache”，不要求其对象按处理器硬件 cache line 大小对齐，没有定义其构造和析构函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 dentry_cache。调用 kmalloc(sizeof(*shrinker), GFP_KERNEL)为局部回收器结构 struct shrinker 指针 shrinker 申请内存空间，将 shrinker 的回收函数指针 shrinker 指向 static int shrink_dcache_memory(int nr, unsigned int gfp_mask)函数，将 shrinker 的 seeks 设置成 DEFAULT_SEEKS（=2），将 shrinker 的等待删除计数器清 0。将回收器读写信号量 shrinker_rwsem 写减少，将回收器结构变量 shrinker 加入到全局回收器结构链表 shrinker_list 中去，将回收器读写信号量 shrinker_rwsem 写增加。

inode_init(unsigned long mempages)函数执行过程：调用 kmem_cache_create ("inode_cache", sizeof(struct inode), 0, SLAB_PANIC, init_once, NULL)函数为索引节点描述结构 struct inode 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“inode_cache”，不要求其对象按处理器硬件 cache line 大小对齐，其对象的构造函数指针指向 static void init_once(void * foo, kmem_cache_t * cachep, unsigned long flags)函数，没有定义其析构函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 inode_cachep。调用 kmalloc(sizeof(*shrinker), GFP_KERNEL)为局部回收器结构 struct shrinker 指针 shrinker 申请内存空间，将 shrinker 的回收函数指针 shrinker 指向 static int shrink_icache_memory(int nr, unsigned int gfp_mask)函数，将 shrinker 的 seeks 设置成 DEFAULT_SEEKS（=2），将 shrinker 的等待删除计数器清 0，将回收器读写信号量 shrinker_rwsem 写减少，将回收器结构变量 shrinker 加入到全局回收器结构链表 shrinker_list 中去，将回收器读写信号量 shrinker_rwsem 写增加。

mnt_init(unsigned long mempages)函数执行过程：调用 kmem_cache_create ("mnt_cache", sizeof(struct vfsmount), 0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL, NULL)函数为已挂接文件系统描述符结构 struct vfsmount 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“mnt_cache”，没有定义其构造和析构函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 mnt_cache。将局部变量 order 设置为 0，执行__get_free_pages(GFP_ATOMIC, order)函数申请一页内存空间，将申请到的内存页的虚拟起始地址传给系统虚拟文件系统挂接 hash 表链表结构指针 mount_hashtable。如果申请失败，系统 panic()，计算一页空间能存放多少个 hash 链表的链表头 struct list_head 结构，计算结果传给局部变量 nr_hash。计算 nr_hash 所占的位数传给全局变量 hash_bits，1<<hash_bits 应该小于或等于 nr_hash。将 nr_hash 设置为 1<<hash_bits，将全局变量 hash_mask 设置为（1<<hash_bits）-1，将 mount_hashtable 所指地址开始存放的连续 1<<hash_bits 个 struct list_head 结构全部初始化为初始即链表空状态。执行 sysfs_init()函数初始化系统文件系统 sysfs，执行 init_rootfs()函数注册系统根文件系统类型描述结构变量

root_fs_type，执行 init_mount_tree()函数挂接 root_fs_type。

sysfs_init(void)函数执行过程：在没有配置 CONFIG_SYSFS 选项（CONFIG_SYSFS 可以在编译前配置内核时设置，该选项在嵌入式系统中常用。系统文件系统 sysfs 是一个虚拟文件系统，内核使用 sysfs 来导出内核对象以及这些对象的属性以及各对象之间的关系。用户可以用 sysfs 来探测正在运行的内核的有用信息，比如内核在每条总线上找到的设备以及与它们绑定的设备驱动，sysfs 还可以用来调节系统设备和其他子系统的性能。/sbin/hotplug 的操作也要用到 sysfs，目前 sysfs 被块设备子系统用来挂接 root partition 即根分区，如果系统禁止了 sysfs，必须在系统启动命令行中指定启动设备的主设备号和次设备号。例如：如果要从 /dev/hda1 启动，应在命令行中加上“root=03:01”）的系统中，该函数直接返回 0 什么也不做。在配置了 CONFIG_SYSFS 选项的系统中执行如下：执行 register_filesystem(&sysfs_fs_type)函数将系统文件系统 sysfs 的描述结构全局变量 sysfs_fs_type 注册到整个系统的文件系统类型描述结构链表 file_systems 中，加到 file_systems 链表的最后面。如果注册成功，执行 kern_mount(&sysfs_fs_type)函数挂接系统文件系统 sysfs，将返回的已挂接文件系统描述符结构 struct vfsmount 指针传给全局结构指针 sysfs_mount。如果 sysfs_mount 指针错误，打印挂接系统文件系统错误，将 sysfs_mount 指针设置为 NULL，返回 register_filesystem(&sysfs_fs_type)函数的返回值。

register_filesystem(struct file_system_type * fs)函数执行过程：如果 fs 指针为空，返回无效错误。如果 fs->next 指针不空，说明该文件系统描述结构已经加入到链表中，返回忙错误。将文件系统读写锁 file_systems_lock 的写上锁。在全局结构指针 file_systems 中查找所有已经注册了的文件系统类型描述结构，看有没有与 fs->name 同名的文件系统，如果有，返回忙错误，如果没有则将 fs 加到 file_systems 链表的最后，将 file_systems_lock 的写解锁，返回 0。

kern_mount(struct file_system_type *type) 函数执行过程：直接调用并返回 do_kern_mount(type->name, 0, type->name, NULL)函数。do_kern_mount(const char *fstype, int flags, const char *name, void *data)函数执行过程如下：从系统注册了的文件系统类型描述结构链表 file_systems 重找出与 fstype 同名的文件系统类型描述结构指针 fs，如果 fs 非空，但拥有 fs 的模块描述结构指针 fs->owner 的状态字 state 等于 MODULE_STATE_GOING，即文件系统类型 fs 的拥有者模块已经不可用，不能被激活，那么把 fs 设置为 NULL，然后调用 request_module("%s", fstype)函数去装载并运行用户空间的/sbin/modprobe 程序来探测名叫 fstype 的模块，如果探测成功，再次从系统注册了的文件系统类型描述结构链表 file_systems 重找出与 fstype 同名的文件系统描述结构指针 fs，如果 fs 非空，但拥有 fs 的模块描述结构指针 fs->owner 的状态字 state 等于 MODULE_STATE_GOING，即文件系统类型 fs 的拥有者模块已经不可用，不能被激活，那么再次把 fs 设置为 NULL。如果 fs 仍空，返回没有设备错误。调用 kmem_cache_alloc(mnt_cache, GFP_KERNEL)从高速缓存内存 mnt_cache 中申请一个 struct vfsmount 对象并将其地址传给局部 struct vfsmount 结构指针 mnt，将 mnt 结构全部清 0，将 mnt

的挂接计数器 mnt_count 设置为 1，初始化 mnt 的链表结构 mnt_hash、mnt_child、mnt_mounts、mnt_list、mnt_fslink，为名字字符串 name 另外申请其同样大小的内存空间，把 name 复制到新申请到的内存空间中，然后将 mnt 的名字字符串指针 mnt_devname 指向新的名字字符串。如果数据指针 data 不空，调用 get_zeroed_page(GFP_KERNEL)函数申请一页内存空间并清 0，将其虚拟起始地址传给局部指针 secdata，再调用 security_sb_copy_data(fs, data, secdata)将 data 中的数据复制到刚申请到的内存中。执行文件系统类型描述结构 fs 的超级块获取函数指针所指成员函数 fs-> get_sb(type, flags, name, data)，返回超级块结构指针 sb。再执行 security_sb_kern_mount(sb, secdata)函数，将 sb 传给 mnt->mnt_sb，先将 sb->s_root->d_count 加 1，再将 sb->s_root 传给 mnt->mnt_root 以及 mnt->mnt_mountpoint，将 mnt 传给 mnt->mnt_parent，将当前任务的 current->namespace 传给 mnt->mnt_namespace。将 fs->owner->ref[get_cpu()]的计数器 count 减 1，再调用 wake_up_process(fs->owner->waiter)唤醒 waitrer 任务，返回 mnt。

int fastcall wake_up_process(task_t * p)函数执行过程：直接调用并返回 try_to_wake_up(p, TASK_STOPPED|TASK_TRACED|TASK_INTERRUPTIBLE|TASK_UNINTERRUPTIBLE,0)。

static int try_to_wake_up(task_t * p, unsigned int state, int sync)函数执行过程：将 success 设置为 0。禁止 IRQ 中断，保护现场，将 p 所在处理器的运行队列结构 per_cpu_runqueues 传给 rq，将 rq->lock 上锁。将 rq->ttwu_cnt 加 1，将 p->state 传给 old_state。如果 old_state&state 等于 0，跳转到 out 标号位置。如果 p->array 指针不空，说明进程 p 已经在等待队列中，跳转到 out_running 标号位置。将 p->thread_info->cpu 传给 cpu，将 smp_processor_id()传给 this_cpu。省略 CONFIG_SMP 系统操作部分。如果 old_state 等于 TASK_UNINTERRUPTIBLE，即进程 p 处于等待队列中，待资源有效时唤醒，不可由其他进程通过信号或者定时中断唤醒，将 rq->nr_uninterruptible 加 1，将 p->activate 设置为-1。将系统从启动到现在运行的纳秒数传给 now，执行 recalc_task_prio(p, now)重新计算进程 p 的优先级。如果 p->activated 等于 0，如果系统正在进行 interrupt 处理，将 p->activated 设置为 2，否则将 p->activated 设置为 1，将 now 传给 p->timestamp。如果 p->sched_info.last_queued 等于 0，将 jiffies 传给 p->sched_info.last_queued，将进程 p 通过 p->run_list 连接到 rq->active->queue[p->prio]队列链表末尾。将 rq->active->bitmap 位码表中的第 p->prio 位设置为 1。将 rq->active->nr_active 加 1；将 rq->active 传给 p->array。将 rq->nr_running 加 1，如果 sync 等于 0 或者 cpu 不等于 this_cpu，且 p->prio 小于 rq->curr->prio，则执行 resched_task(rq->curr)将 rq->curr->thread_info->flags 的 TIF_NEED_RESCHED 位设置为 1，并将 success 设置为 1。“out_running 标号位置”：将 p ->state 设置为 TASK_RUNNING；“out 标号位置”：使能 IRQ 中断，恢复现场，将 rq->lock 解锁，返回 success。

init_rootfs(void)函数执行过程：直接调用并返回 register_filesystem(&rootfs_fs_type)函数。将系统根文件系统类型描述结构变量 rootfs_fs_type 注册到系统文件系统类型描述结构链表 file_systems 的末尾。

init_mount_tree(void)函数执行过程：执行 do_kern_mount("rootfs", 0, "rootfs", NULL)挂接系统根文件系统 root_fs_type，将返回的已挂接文件系统描述符结构指针传给局部指针 mnt。为局部名字空间结构指针 namespace 申请大小为 sizeof(struct namespace)的内存空间。将 namespace->count 设置为 1，将 namespace 的已挂接文件系统描述符结构链表 list 初始化为空链表，初始化 namespace 的读写信号量 sem。将 rootfs 的挂接描述结构变量 mnt 通过其 mnt_list 链接到 namespade 的链表 list 中的开始，将 mnt 传给 namespace->root，将 namespace 传给初始任务（系统启动进程）的 init_task.namespace，将进程链表读写锁 tasklist_lock 读上锁。从系统进程描述结构链表中 init_task 的下一个进程开始依次对系统中当前所有的进程进行如下操作，每次先将 namespace 的 count 加 1，再将其传给该进程描述结构变量的 namespace 指针。将进程链表读写锁 tasklist_lock 读解锁，执行 set_fs_pwd(current->fs, namespace->root, namespace->root-> mnt_root) 将根文件系统 rootfs 的挂接描述结构变量指针 mnt 和它的根目录条目结构指针 mnt_root 分别设置成当前任务 current 的文件系统描述结构 fs 的当前挂接结构变量指针 pwdmnt 和当前目录条目结构指针 pwd。执行 set_fs_root(current->fs, namespace->root, namespace->root->mnt_root)函数将根文件系统 rootfs 的挂接结构变量指针 mnt 和它的根目录条目结构指针 mnt_root 分别设置成当前任务 current 的文件系统描述结构的根挂接结构变量指针 rootmnt 和根目录条目结构指针 root。

bdev_cache_init(void)函数执行过程：调用 kmem_cache_create ("bdev_cache", sizeof(struct bdev_inode), 0, SLAB_HWCACHE_ALIGN|SLAB_RECLAIM_ACCOUNT|SLAB_PANIC, init_once, NULL)函数为块设备索引节点描述结构 struct bdev_inode 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“bdev_cache”，其对象构造函数指针指向 void filp_ctor(void * objp, struct kmem_cache_s *cachep, unsigned long cflags)函数，对象析构函数指针指向 static void init_once(void * foo, kmem_cache_t * cachep, unsigned long flags)函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 bdev_cachep。执行 register_filesystem(&bd_type)函数将系统块设备文件系统类型描述结构变量 bd_type 注册到系统文件系统类型描述结构链表 file_systems 的末尾。执行 kern_mount(&bd_type)函数，挂结 bd_type，将返回的已挂接文件系统描述符结构指针传给全局指针 bd_mnt。将 bd_mnt->mnt_sb 传给全局指针 blockdev_superblock。

chrdev_init(void)函数执行过程：执行 subsystem_init(&cdev_subsys)函数，执行 kobj_map_init (base_probe, &cdev_subsys)函数，将返回的内核对象映射结构指针传给字符设备内核对象映射结构全局指针 cdev_map。

subsystem_init(struct subsystem * s)函数执行过程：初始化 s 的读写信号量 rwsem，执行 kset_init(&s->kset)函数初始化 s 的内核组件描述结构成原变量 kset，将 s.kset.kobj.kref.refcount 设置为 1，将 s.kset..kobj.entry 链表初始化为空链表，将 s.kset.kobj.kref.refcount 加 1，将 s.kset.list 链表初始化为空链表。

kobj_map_init(kobj_probe_t *base_probe, struct subsystem *s)函数执行过程：调用 kmalloc (sizeof(struct kobj_map), GFP_KERNEL)函数为内核对象映射结构 struct kobj_map 局部指针 p 申请内存空间。调用 kmalloc(sizeof(struct probe), GFP_KERNEL)函数为内核探测结构 struct probe 局部指针 base 申请内存空间，将 base 所指结构空间清 0，将 base 的设备号 dev 设置为 1，将 base 的探测设备号返回 range 设置为~0，将 base 的内核对象探测函数指针 get()指向传入的 base_probe()函数（对于 cdev_subsys 子系统，get()指向 static struct kobject *base_probe (dev_t dev, int *part, void *data)函数）。将 p 中的探测结构指针数组的所有探测结构指针 probes[n]（n=0~254）都设置为 base，将 p 的读写信号量指针 sem 指向传入的子系统结构的读写信号量 s->rwsem 并返回 p。

17.5.39 radix_tree_init()函数

该函数调用 kmem_cache_create ("radix_tree_node", sizeof(struct radix_tree_node), 0, SLAB_PANIC, radix_tree_node_ctor, NULL)函数为基数树节点描述结构 struct radix_tree_node 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“radix_tree_node”，不要求其对象按处理器硬件 cache line 大小对齐，其对象的构造函数指针指向 static void radix_tree_node_ctor(void *node, kmem_cache_t *cachep, unsigned long flags)函数，没有定义其析构函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 radix_tree_node_cachep。初始化无符号长整型全局数组 height_to_maxindex[RADIX_TREE_MAX_PATH=7]的每个变量，将最后一个变量之外的其他变量 height_to_maxindex[n(n=0~5)]设置为最高 n* RADIX_TREE_MAP_SHIFT 位为 0，其他位为 1，将 height_to_maxindex[6]设置为 0xffffffff。按从高到低的优先级顺序将通知块结构变量 radix_tree_callback_nb 注册到 CPU 通知者结构变量链表 cpu_chain 中。

17.5.40 signals_init()函数

该函数调用 kmem_cache_create("sigqueue", sizeof(struct sigqueue), __alignof__(struct sigqueue), SLAB_PANIC, NULL, NULL)函数为信号队列结构 struct sigqueue 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“sigqueue”，不要求其对象按处理器硬件 cache line 大小对齐，没有定义其对象的构造和析构函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 sigqueue_cachep。

17.5.41 page_writeback_init()函数

该函数统计系统中所有内存节点的通用（NORMAL）内存页区中高页数水印值页数之外的额外内存总页数之和传给 buffer_pages。统计系统中所有内存节点的 HIGH_MEM 内存页区中高页数水印值页数之外的额外内存总页数传给全局变量 total_pages。将 buffer_pages*100*4/total_pages 传给 correction，如果 correction 小于 100，将后台脏页比率全局

变量 dirty_background_ratio*correct/100 传给 dirty_background_ratio，将虚拟脏页比率全局变量 vm_dirty_ratio*correct/100 传给 vm_dirty_ratio。如果 dirty_background_rati 小于等于 0，dirty_background_rati 设置为 1，如果 vm_dirty_ratio 小于等于 0，vm_dirty_ratio 设置为 1。修改写回定时器链表全局结构 wb_timer，将（total_pages/32/当前在线 CPU 数）传给 ratelimit_pages，如果 ratelimit_pages 小于 16，将 ratelimit_pages 设置为 16，如果 ratelimit_pages 大于 1024，将 ratelimit_pages 设置为 1024。按从高到低的优先级顺序将通知者块结构变量 ratelimit_mb 注册到 CPU 通知者结构变量链表 cpu_chain 中。

17.5.42 proc_root_init()函数

该函数只有在系统支持 proc 文件系统即配置了 CONFIG_PROC_FS 选项时才被调用。

调用 kmem_cache_create ("proc_inode_cache", sizeof(struct proc_inode), 0, SLAB_RECLAIM_ACCOUNT, init_once, NULL)函数为 proc 文件系统索引节点描述结构 struct proc_inode 创建高速缓存内存描述结构 kmem_cache_t 变量，名字叫“proc_inode_cache”，不要求其对象按处理器硬件 cache line 大小对齐，其对象的构造函数指针指向 static void init_once(void * foo, kmem_cache_t * cachep, unsigned long flags)函数，没有定义其析构函数，将创建好的 kmem_cache_t 结构变量的地址传给全局指针 proc_inode_cachep。调用 register_filesystem (&proc_fs_type)函数将 proc 文件系统类型描述结构 proc_fs_type 注册到系统的文件系统类型描述结构链表 file_systems 的末尾。调用 kern_mount(&proc_fs_type)函数为 proc 文件系统创建 struct inode、struct dentry、struct super_block 以及 struct vfsmount 结构变量，并把 struct vfsmount 结构变量的地址传给全局结构指针 proc_mnt。分别调用 create_proc_read_entry()函数，在/proc 目录下创建并初始化以下文件“loadavg”、“uptime”、“meminfo”、“version”、“hardware”、“stram”、“devices”、“filesystems”、“cmdline”、“locks”、“execdomains”对应的 struct proc_dir_entry 结构。调用 proc_symlink("mounts", NULL, "self/mounts")在/proc 目录下创建一个名叫“mounts”的符号链接到/proc/self/mounts。创建/proc/kmsg 文件，拥有者用户只读。调用 create_seq_entry ("cpuinfo", 0, &proc_cpuinfo_operations)函数，创建/proc/cpuinfo 目录条目，该条目的文件操作函数集 proc_fops 设置为 proc_cpuinfo_operations。创建/proc/partitions 目录条目，该条目的文件操作函数集 proc_fops 设置为 proc_ partitions_operations。创建/proc/stat 目录条目，该条目的文件操作函数集 proc_fops 设置为 proc_stat_operations。创建/proc/interrupts 目录条目，该条目的文件操作函数集 proc_fops 设置为 proc_interrupts_operations。创建/proc/slabinfo 目录条目，该条目的文件操作函数集 proc_fops 设置为 proc_ slabinfo_operations。创建/proc/buddyinfo 目录条目，该条目的文件操作函数集 proc_fops 设置为 proc_buddyinfo_ operations。创建/proc/vmstat 目录条目，该条目的文件操作函数集 proc_fops 设置为 proc_ vmstat_operations。创建 /proc/diskstats 目录条目，该条目的文件操作函数集 proc_fops 设置为 proc_diskstats_operations。创建/proc/modules 目录条目，该条目的文件操作函数集 proc_fops 设置为

proc_ modules_operations。创建/proc/schedstat 目录条目，该条目的文件操作函数集 proc_fops 设置为 proc_ schedstat_operations。创建/proc/kcore 目录条目，创建/proc/sysrq-trigger 目录条目，创建/proc/ppc_htab 目录条目。调用 proc_mkdir("net", NULL)创建/proc/net/目录，对应目录条目描述结构地址传给全局指针 proc_net。创建/proc/net/stat/目录，对应目录条目描述结构地址传给全局指针 proc_net_stat。创建/proc/sysvipc/目录，创建/proc/sys/目录，创建/proc/sys/fs/binfmt_misc/目录。创建/proc/fs/目录，对应目录条目描述结构地址传给全局指针 proc_root_fs；创建/proc/driver/目录，对应目录条目描述结构地址传给全局指针 proc_root_driver。创建/proc/fs/nfsd/目录，创建/proc/tty/目录，创建/proc/tty/ldisc/目录，对应目录条目描述结构地址传给全局指针 proc_tty_ldisc。创建/proc/tty/driver/目录，对应目录条目描述结构地址传给全局指针 proc_tty_driver。创建/proc/tty/drivers 文件，其 proc_fops 指向 proc_tty_drivers_operations。创建/proc/bus/目录。

create_proc_read_entry (const char *name, mode_t mode, struct proc_dir_entry *base, read_proc_t *read_proc, void * data)函数执行过程：调用 create_proc_entry(name,mode,base)函数创建 proc 目录条目结构 struct proc_dir_entry，创建的目录条目结构地址传给局部指针 res。如果创建成功，将 proc 目录条目读取函数指针 read_proc 传给 res->read_proc，把数据指针传给 res->data，返回 res。

proc_dir_entry *create_proc_entry(const char *name, mode_t mode, struct proc_dir_entry *parent)函数执行过程：如果 mode 设置了目录标志 S_IFDIR 即要创建一个 proc 目录，如果 mode 没有设置 S_IALLUGO 标志组合中的任何标志位，将 mode 或上 S_IRUGO | S_IXUGO 即拥有者用户、组用户、其他用户都可以读/执行/查询该目录。将局部变量 nlink 设置为 2，如果 mode 没有设置目录标志 S_IFDIR，即创建的不是目录，如果没有设置 S_IFMT 中的任何位，将 mode 或上 S_IFREG 即创建一个 proc 文件。如果 mode 没有设置 S_IALLUGO 标志组合中的任何标志位，将 mode 或上 S_IRUGO |即拥有者用户、组用户、其他用户都可以读/查询该条目。将局部变量 nlink 设置为 2。proc_create(&parent,name,mode,nlink)函数执行过程是：如果 name 的上一级目录条目指针 parent 是空的，根据该目录的名字 name 从 proc 文件系统根目录条目 proc_root 中逐级查找到 name 的上一级目录作为它的 parent，如果找到了它的 parent，再为 name 创建一个新的 proc 目录条目结构 struct proc_dir_entry 指针 ent，初始化 ent 相关成员变量。如果 ent 指针不空，如果 mode 设置了 S_IFDIR 标志位，将 ent->proc_fops 指向 proc_dir_operations，将 ent->proc_iops 指向 proc_dir_inode_operations。调用 proc_register(parent, ent) 将目录条目 ent 和父目录 parent 链接起来，并根据具体模式设置 ent 的文件操作函数集和 inode 操作函数集，如果链接失败，释放 ent 空间，将 ent 设置为 NULL，并返回 ent。

static struct proc_dir_entry *proc_create(struct proc_dir_entry **parent, const char *name, mode_t mode, nlink_t nlink)函数执行过程：如果名字指针 name 空或名字长度为 0，返回 NULL。如果*parent 指针空，那么从 proc_root 根目录往下找到 name（name 是绝对路径，如

/proc/myproc/myentry）的上一级目录的 proc 目录条目描述结构指针地址（传给 parent）以及最后一级的名字字符串指针 fn，获取字符串 fn 的长度传给 len，调用 kmalloc(sizeof(struct proc_dir_entry) + len + 1, GFP_KERNEL)为新的 proc 目录条目 ent 和它的名字申请内核空间。将 ent 指向的 struct proc_dir_entry 结构空间清 0，将字符串 fn 复制到紧随 ent 所指 struct proc_dir_entry 结构后面的空间内，复制长度为 len 加 1（包括结束字符'\0'，也就是说 strlen(fn)得出的是长度是不包含结束字符的）。将 ent->name 指向刚才复制名字 fn 的目的起始位置，将 ent->namelen 设置为 len，将 ent->mode 设置为 mode，将 ent->nlink 设置为 nlink，并返回 ent。

proc_register(struct proc_dir_entry * dir, struct proc_dir_entry * dp)函数执行过程：通过 proc_inum_idr 中生成 id 号再加上 PROC_DYNAMIC_FIRST 传给 dp->low_ino。将 dir->subdir 传给 dp->next，将 dir 传给 dp->parent，将 dp 传给 dir->subdir，如果 dp 对应的是一个目录，如果 dp->proc_iops 指针为空，将 dp->proc_fops 指向 proc_dir_operations，将 dp->proc_iops 指向 proc_dir_inode_operations，将 dir->nlink 加 1。如果 dp 对应的是一个符号链接，如果 dp->proc_iops 指针空，将 dp->proc_iops 指向 proc_link_inode_operations。如果 dp 对应的是一个普通文件，如果 dp->proc_iops 指针空，将 dp->proc_fops 指向 proc_file_operations，如果 dp->proc_iops 指针空，将 dp->proc_iops 指向 proc_file_inode_operations，并返回 0。

struct proc_dir_entry *proc_symlink(const char *name, struct proc_dir_entry *parent, const char *dest)函数操作过程：先调用 proc_create(&parent,name, (S_IFLNK | S_IRUGO | S_IWUGO | S_IXUGO),1)创建一个名叫 name 的符号链接的 struct proc_dir_entry 结构 ent，如果创建成功，调用 kmalloc((ent->size=strlen(dest))+1, GFP_KERNEL)申请一个目的条目名字 dest 字符串长度加 1 大小的内存空间，将 ent->data 指向这个空间起始地址，如果申请内存成功，调用 proc_register (parent, ent)将 ent 注册到 parent 的子目录 subdir 链表开始，并设置其操作函数组，返回 ent。

create_seq_entry(char *name, mode_t mode, struct file_operations *f)函数执行过程：调用 create_proc_entry(name, mode, NULL)创建 name 目录条目 ent，如果创建成功，将 ent->proc_fops 指向 f。

proc_mkdir(const char *name, struct proc_dir_entry *parent)函数执行过程：直接调用并返回 proc_mkdir_mode(name, S_IRUGO | S_IXUGO, parent)函数。proc_mkdir_mode(const char *name, mode_t mode, struct proc_dir_entry *parent)函数执行过程：调用 proc_create(&parent, name, S_IFDIR | mode, 2)创建 name 目录条目 ent，如果创建成功，将 ent-> proc_fops 指向 proc_dir_operations，将 ent-> proc_iops 指向 proc_dir_inode_operations，将 ent 注册到 parent 的 subdir 链表的开始，返回 ent。

17.5.43 check_bugs()函数

check_bugs(void)函数由各种处理器架构自行定义，很多定义了此函数的系统中此函数事实

上什么也没做，有的就是空函数，在 ARM 中被定义成了 check_writebuffer_bugs(voide)函数。void __init check_writebuffer_bugs(voide)函数执行过程：调用 alloc_page(GFP_KERNEL)申请一页内核内存，将该页的 struct page 结构指针传给局部指针 page。如果 page 非空，则设置该页的二级页描述符控制字段，连续两次调用 vmap(&page, 1, VM_IOREMAP, prot)将 page 所指的物理页内存先后映射到两个不同的虚拟地址 p1 和 p2 上。如果 p1 和 p2 指针都不空，那么禁止 IRQ 中断，执行 mb()，在虚拟地址 p1 处写 1，执行 mb()，在虚拟地址 p2 处写 0，执行 mb()，读取虚拟地址 p1 处的值到 v，执行 mb()，使能 IRQ 中断，调用 vunmap(p1)和 vunmap(p2)去掉虚拟地址 p1 和 p2 的物理映射，调用 put_page(page)释放刚才申请到的测试用物理内存页。如果 v 的值不等于 0，说明映射失败，也就是说 p1 和 p2 没有都映射到 page 所指的物理内存地址，或者因为其他原因导致系统的内存访问一致性出了问题，打印失败原因，如果 v 等于 0，说明映射正确，前后内存访问一致性得到保证，打印“ok”。

void *vmap(struct page **pages, unsigned int count, unsigned long flags, pgprot_t prot)函数执行过程：如果 count 即要映射的内存页数大于系统中实际存在的全部物理内存总页数 num_physpages，返回 NULL；调用 get_vm_area((count << PAGE_SHIFT), flags) 获取可以映射 count 页大小物理空间的虚拟内存空间，创建该虚拟内存块的 struct vm_struct 结构指针 area，并将 area 插入到系统全局 struct vm_struct 结构链表 vmlist 中它的起始虚拟地址对应的位置。如果指针 area 为空，则返回 NULL。调用 map_vm_area(area, prot, &pages)函数将 area 描述的虚拟内存空间映射到 pages 所指内存的物理地址上，如果映射失败，调用 vunmap(area->addr)取消虚拟内存空间映射，返回 NULL，如果映射成功，返回虚拟地址 area->addr。

struct vm_struct *get_vm_area(unsigned long size, unsigned long flags)函数执行过程：直接调用并返回__get_vm_area(size, flags, VMALLOC_START, VMALLOC_END)函数。struct vm_struct *__get_vm_area(unsigned long size, unsigned long flags, unsigned long start, unsigned long end)函数执行过程：如果 flags 中设置了 VM_IOREMAP 标志位，检查 size 中最高的为 1 的位号 bn，如果该位号大于 IOREMAP_MAX_ORDER（=7+ PAGE_SHIFT），那么将 bn 设置为 IOREMAP_MAX_ORDER，也就是说一次最大能映射 128 页大小的 I/O 空间，如果 bn 小于 PAGE_SHIFT，将 bn 设置为 PAGE_SHIFT，也就是说，一次最少要映射 1 页大小的 I/O 空间，将对齐值 align 设置为 1<<bn。如果 flags 中没有设置 VM_IOREMAP 标志位，align 等于 1；将映射范围的起始虚拟地址 start 按 align 对齐后传给 addr，调用 kmalloc(sizeof(*area),GFP_KERNEL)为 struct vm_struct 结构指针 area 申请内存，area 等于 NULL 即申请内存失败返回 NULL。将 size 加上 PAGE_SIZE，如果 size 等于 0，调用 kfree(area)释放刚申请到的内存，返回 NULL。将虚拟内存块描述结构链表读写锁 vmlist_lock 写上锁，查看系统 struct vm_struct 结构链表 vmlist，如果连标中相邻两个结构所指虚拟空间之间的空隙能容纳本息映射空间，那么将 area 插入到链表这个间隙中，如果没有，那么将 area 架到 vmlist 链表的末尾，area->next 等于 NULL，将 flags 传给 area->flags，将找到的合适的 addr 传给 area->addr，将 size 传给

area->size，area->page 设置为 NULL，area->nr_pages 设置为 0，area->phys_addr 设置为 0。将虚拟内存块描述结构链表读写锁 vmlist_lock 写解锁，返回 area。如果没有找到合适的用于映射的虚拟空间，那么将虚拟内存块描述结构链表读写锁 vmlist_lock 写解锁，调用 kfree(area)释放 area 空间，打印信息，返回 NULL。

int map_vm_area(struct vm_struct *area, pgprot_t prot, struct page ***pages)函数执行过程：将 area->addr 传给 address，将 address+（area->size-PAGE_SIZE）传给 end。找到虚拟地址 address 对应的一级描述符应存放的位置转给 pgd_t 指针 pgd（所有的一级描述符都存放在 init_mm->pgd 也就是 swapper_pg_dir 所指的 16KB 空间中，一个 pgd_t 数据对应两条一级描述符条目，一条一级描述符条目对应 1MB 空间的映射关系）。将 init_mm.page_table_lock 锁上锁，进入循环。将 pgd 传给 pmd_t 指针 pmd（因为许多嵌入式系统中内存最多进行两级分页管理，比 i386 的三级管理少一级，所以 pgd 与 pmd 合二为一，即 pgd 和 pmd 相等）。如果 pmd 空，设置错误跳出循环。执行 map_area_pmd(pmd, address, end - address, prot, pages)函数，如果 address 所在的连续两段（2MB）内存空间都未被映射，则同时为两张分别包含 Linux 和硬件版本的页描述符申请一页内存空间，获取第一张和第二张二级页表的一级描述条目数值，分别将它们存入第一张和第二张二级页表对应的一级描述符地址条目地址 pmd 和 pmd+1 处，然后刷新 pmd 地址的 cache 和 Write Buffer，设置从 address 到 end 之间空间的 Linux 版本和硬件版本页描述符，如果映射出错，设置错误跳出循环。将(address+PGDIR_SIZE)&PGDIR_MASK 传给 address，将 pgd 加 1。如果 address 小于 end，进行下一次循环操作。将 init_mm.page_table_lock 锁解锁，调用 flush_cache_vmap((unsigned long) area->addr, end)刷新 area->addr 到 end 之间的 cache。

static int map_area_pmd(pmd_t *pmd, unsigned long address, unsigned long size, pgprot_t prot, struct page ***pages)执行过程：为 address 地址所在连续 2MB 内存空间申请 4 张页表内存空间（刚好是 1 页大小），再一兆一兆设置内从 address 位置开始的一级描述条目以及二级页表，先将 address 与上 PGDIR_MASK 传给 base，将 address 与上~PGDIR_MASK 传回给 address，address 加上 size 传给 end，如果 end 大于 PGDIR_SIZE，将 end 设置成 PGDIR_SIZE，然后循环执行以下操作：执行 pte_alloc_kernel(&init_mm, pmd, base + address) 为 address 地址所在连续 2MB 内存空间申请 4 张页表内存空间（刚好是 1 页大小），将申请到的空间清 0，清空前 2KB 内存（两张硬件版本页表）对应的 cache，设置两段内存的一级描述条目 pmd[0]和 pmd[1]，刷新 pmd 处的 cache 和 Write Buffer，将指向 Linux 版本页表起始地址传给 pte_t 指针 pte，如果 pte 指针空，返回没有足够内存错误，调用 map_area_pte(pte, address, end - address, prot, pages) 逐页设置 address 地址所在 1MB 内存空间内从 address 位置开始的页的描述条目，如果映射失败，返回没有足够内存错误，将(address+PMD_SIZE)与上 PMD_MASK 传给 address，将 pmd 加 1，如果 address 小于 end，执行下一次循环操作，返回 0。

pte_t fastcall *pte_alloc_kernel(struct mm_struct *mm, pmd_t *pmd, unsigned long address)函数执行过程：如果 pmd 地址处的条目值等于 0，说明连续 2MB 内存空间都没有被映射，因为

连续 2MB 内存空间的一级页描述条目是同时设置的，将 mm->page_table_lock 上锁，调用 pte_alloc_one_kernel(mm, address)为连续 2MB 内存空间的 4 张页表申请 1 页内存空间，将申请到的前 2KB 空间（硬件版本页表）清 0，并把 Linux 版本页表的起始地址传给 pte_t 指针 new，将 mm->page_table_lock 上解锁，如果 new 指针空，返回 NULL，再次检查 pmd 处的值是否为 0，如果是 0，调用 pmd_populate_kernel(mm, pmd, new)设置连续 2MB 内存空间的一级页描述条目 pmd[0]和 pmd[1]，并且刷新 pmd 处的 cache 和 Write Buffer，如果前面再次检查 pmd 处的值不是 0，调用 pte_free_kernel（new）释放刚申请到的内存页。返回虚拟地址 address 对应的 linux 版本页描述条目存放地址。

static inline pte_t *pte_alloc_one_kernel(struct mm_struct *mm, unsigned long addr)执行过程：调用__get_free_page(GFP_KERNEL|__GFP_REPEAT)为连续 2MB 内存的 4 张页表申请一页内存空间（1MB 内存需要两张页表，一张是 Linux 版本，一张是硬件版本，每张页表含 256 个页描述条目，一个页描述条目占 4 字节描述 1 页 4KB 内存的映射，所以 2MB 内存的 4 张页表刚好占用 4KB 即 1 页内存空间），把申请到的内存起始地址传给 pte_t 指针 pte。如果 pte 不空即申请内存成功，将 pte 所指页空间全部清 0，并清空前 2KB 内存（两张硬件版本页表）对应的 cache，将 pte 加上 PTRS_PER_PTE，也就是说将 pte 指向 linux 版本二级页表起始位置，返回 pte。

static inline void pmd_populate_kernel(struct mm_struct *mm, pmd_t *pmdp, pte_t *ptep)执行过程：将 ptep 传给 pte_ptr，如果 mm 不指向 init_mm，则系统崩溃。将 pte_ptr 减去 PTRS_PER_PTE * sizeof(void *)，使得 pte_ptr 指回到硬件版本二级页表的开始（因为 ptep 指向 Linux 版本二级页表的开始），将 pte_ptr 对应的物理地址或上_PAGE_KERNEL_TABLE 传给 pmdval 作为第一段内存（前 1MB）的一级页描述条目，将 pmdval 传给 pmdp[0]，将 pmdval + 256 * sizeof(pte_t)作为第二段内存（后 1MB）的一级页描述条目传给 pmdp[1]，调用 flush_pmd_entry(pmdp)刷新 pmdp 地址处的 cache 和 Write Buffer。

static int map_area_pte(pte_t *pte, unsigned long address, unsigned long size, pgprot_t prot, struct page ***pages)函数执行过程：逐页设置 address 地址所在 1MB 内存空间内从 address 位置开始的页的描述条目，先将 address 与上~PMD_MASK 传回 address，address 加上 size 传给 end，如果 end 大于 PMD_SIZE，将 end 设置成 PMD_SIZE，然后循环执行以下操作：如果 pte 地址处的页描述条目不等于 0 也就是说已经设置了页描述条目，系统告警，如果**pages 指针为空，返回没有足够内存错，执行 set_pte(pte, mk_pte(**page, prot))设置 pte 地址处的 Linux 版本页描述条目以及 pte 减去 PTRS_PER_PTE（=512）处的硬件版本页描述条目，该条目将 page 所指物理内存页地址映射成 address 所指虚拟地址，将 address 加上 PAGE_SIZE，将 pte 加 1，将*pages 加 1，然后设置下一页的映射条目，返回 0。

17.5.44 acpi_early_init()函数

void __init acpi_early_init (void)函数只有在配置了 CONFIG_ACPI 选项的系统中才有定义，而在 Linux2.6.10 中只有 X86 或 IA64 处理器平台才能打开 CONFIG_ACPI 配置开关。ACPI 是 Advanced Configuration and Power Interface（先进配置和电源接口）的缩写，配置 CONFIG_ACPI 将使系统体积增加 70KB，ACPI 需要硬件和 firmware 的支持，并且需要系统中已经存在操作系统直接配置和电源管理软件。

17.5.45 rest_init()函数

该函数创建 init()内核进程即 1 号进程，然后是系统启动进程即 0 号进程空闲，具体执行过程说明如下。

调用 pid_t kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND)创建 init()进程。执行 numa_default_policy()->sys_set_mempolicy(MPOL_DEFAULT, NULL, 0) 检查所有内存节点情况，如果检查没有问题，则根据具体情况从内存访问政策高速缓存空间 policy_cache 为指定内存访问政策模式申请一个 struct mempolicy 结构变量空间，并对其进行初始化。如果当前任务的 mempolicy 指针非空，把 mempolicy 指针所指的原来的 memorypolicy 结构变量空间释放掉，再将其指向刚创建的内存访问政策结构变量。如果是插入模式，则将下一个在线的内存节点号传给当前进程描述结构的 il_next 成员变量。执行 unlock_kernel()，如果当前任务的锁深度值 lock_depth 原来为负数，则系统崩溃，如果原来为 0，将其减 1，再将内核锁 kernel_lock 原子数清 0，解锁，然后使能抢占。执行 cpu_idle()，使启动进程空闲。

long pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)函数执行过程：将局部 struct pt_regs 结构指针 regs 所指空间全部清 0，将 arg 传给 regs.ARM_r1，fn 传给 regs.ARM_r2，do_exit 传给 regs.ARM_r3，kernel_thread_helper 传给 regs.ARM_pc，SVC_MODE 传给 regs.ARM_cpsr，然后调用并返回 do_fork(flags|CLONE_VM|CLONE_UNTRACED, 0, &regs, 0, NULL, NULL)。

long do_fork(unsigned long clone_flags, unsigned long stack_start, struct pt_regs *regs, unsigned long stack_size, int __user *parent_tidptr, int __user *child_tidptr)函数执行过程：调用 alloc_pidmap()申请一个空闲 pid 号传给局部变量 pid。如果 pid 小于 0，返回重试错误；如果 current->ptrace 不等于 0，调用 fork_traceflag (clone_flags)将返回值传给 trace，如果 trace 不等于 0，将 clone_flags 或上 CLONE_PTRACE。执行 copy_process(clone_flags, stack_start, regs, stack_size, parent_tidptr, child_tidptr, pid)函数并将返回值传给局部 struct task_struct 结构指针 p。如果 p 指针有错误，执行 ree_pidmap(pid)将 pid 号对应的位码清 0，将 pid 号对应的 pidmap_t 结构的空闲 id 号计数器 nr_free 加 1，返回 p；如果 clone_flags 设置了 CLONE_VFORK 标志位

即要调试 vfork，让父进程休眠，那么初始化局部 struct completion 结构变量 vfork，并将 p->vfork_done 指向 vfork，如果 p->ptrace 设置了 PT_PTRACED 标志，即要跟踪调试子进程。或者 clone_flags 设置了 CLONE_STOPPED，即将子进程设置为停止状态，那么将 p->pending.signal 的第 SIGSTOP(=19)位设置为 1 。将 p->thread_info->flags 的第 TIF_ SIGPENDING 位设置为 1。如果 clone_flags 没有设置 CLONE_STOPPED 标志位，执行 wake_up_new_task(p, clone_flags)唤醒进程 p，否则将 TASK_STOPPED 传给 p->state，将进程 p 置为暂停状态，只能通过其他进程的信号唤醒。将系统调用 do_fork()成功创建进程计数器 total_forks 加 1。如果不幸 trace 不等于 0，即一定要调试进程 p，将 pid 传给 current-> ptrace_ message，执行 ptrace_notify ((trace << 8) | SIGTRAP)。如果 clone_flags 设置了 CLONE_ VFORK 标志，执行 wait_for_completion(&vfork)，如果 current->ptrace 设置了 PT_TRACE_VFORK_ DONE 标志，则执行 ptrace_notify ((PTRACE_EVENT_VFORK_DONE << 8) | SIGTRAP)，返回 pid。

int alloc_pidmap(void)函数执行过程：将全局变量 last_pit 加 1 传给局部变量 pid，如果 pid 大于全局变量 pid_max（pid_max 声明时设置为 PID_MAX_DEFAULT=0x8000=32768），将 pid 设置为 RESERVED_PIDS(=300)。将 pid 与上 BITS_PER_PAGE_MASK(=0x8000-1)传给 offset。将 pid 的位掩码所在页的 struct pidmap(pidmap_t)结构的地址&pidmap_array[pid/BITS_PER_PAGE]传给局部 pidmap_t 结构指针 map，将要扫描的最大的 pidmap_t 结构变量个数(pid_max + BITS_PER_PAGE - 1)/BITS_PER_PAGE - !offset 传给 max_scan，然后依次查看从第 0（包括）到第 max_scan（包括）个 pidmap_t 结构变量，并作如下操作：如果 map->page 指针为空，调用 get_zeroed_page(GFP_KERNEL)申请一页内存空间并清 0，将申请到的内存起始地址传给局部指针 page，将 pidmap_lock 上锁。再次检查 map->page 是否空，若是，调用 free_page(page)释放刚申请到内存页，否则将 page 传给 map->page，将 pidmap_lock 解锁，如果 map->page 仍空，退出循环，返回-1，如果 map->nr_free 大于 0，进入嵌套循环，依次查看 map->page 所指内存页从 offset 位置开始的所有位，进行如下操作：检查并设置 map->page 所指内存页的第 offset 位，如果原来该位是 0，将 map->nr_free 减 1，将 pid 传给 last_pid，直接返回 pid，如果该位原来就是 1，那么查找 map->page 所指内存页的第 offset 位置后首个为 0 的位号传给 offset，再根据 map 相对于 pidmap_array 的位置以及新的 offset 计算出 offset 对应的 pid 号传给 pid，进入下一次嵌套循环。如果上一个嵌套循环内没有找到空闲的 pid 号，而且 map 又不是 pidmap_array[]中最后的 pidmap_t 结构变量，那么将 map 加 1，指向 pidmap_array[]中下一个 pidmap_t 结构变量，将 offset 设置为 0，如果 map 已经是 pidmap_array[]中最后的 pidmap_t 结构变量，那么将 map 指向 pidmap_array[0]，将 offset 设置成 RESERVED_PIDS，如果不幸 last 等于 offset，那么退出循环，返回-1，然后根据 map 和 offset 生成其对应的 pid 号，进入下一次循环操作。如果以上循环最后没有找到空闲的 pid 号，返回-1。

static inline int fork_traceflag (unsigned clone_flags)函数执行过程：如果 clone_flags 设置了 CLONE_UNTRACED 标志，返回 0。如果 clone_flags 设置了 CLONE_VFORK 标志且 curretn->

ptrace 设置了 PT_TRACE_VFORK 标志，返回 PTRACE_EVENT_VFORK。如果 clone_flags 与上 CSIGNAL 不等于 SIGCHLD 并且 current->ptrace 设置了 PT_TRACE_CLONE 标志，返回 PTRACE_EVENT_CLONE。如果 current->ptrace 设置了 PT_TRACE_CLONE 标志，返回 PTRACE_EVENT_FORK。其他情况返回 0。

static task_t *copy_process(unsigned long clone_flags, unsigned long stack_start, struct pt_regs *regs, unsigned long stack_size, int __user *parent_tidptr, int __user *child_tidptr, int pid)函数执行过程：如果 clone_flags 中同时设置了 CLONE_NEWNS 和 CLONE_FS 即子进程与父进程共享文件系统信息，但是创建新的名字空间，返回无效错误。如果 clone_flags 中设置了 CLONE_THREAD 却没有设置 CLONE_SIGHAND，即父子进程在同一线程组内却不共享信号处理，返回无效错误。如果 clone_flags 中设置了 CLONE_SIGHAND 却没有设置 CLONE_VM，即父子进程共享信号处理却不共享虚拟内存地址空间，返回无效错误。调用 security_task_create(clone_flags)将返回值传给 retval，如果 retval 不等于 0，返回 retval 值。执行 dup_task_struct(current)创建并初始化一个新的 struct task_struct 结构并将其地址传给局部指针 p。如果 p 指针为空，返回 p。将 retval 设置为-EAGAIN，如果 p->user->processes 大于等于 p->signal->rlim[RLIMIT_NPROC].rlim_cur 即拥有进程 p 的用户所拥有的进程数超过了进程 p 的信号结构 signal 的进程资源当前限制值，那么如果当前进程的有效能力标志字 cap_effective 既没有设置 CAP_SYS_ADMIN 标志位也没有设置 CAP_SYS_RESOURCE 标志位而且 p->user 不指向 root_user，也就是说如果进程 p 既不是系统管理任务也不能忽略资源限制值，而且 p 的拥有用户又不是 root_user，执行 free_task(p)释放刚申请到的 struct task_struct 结构 p 高速缓存内存以及 struct thread_info 结构空间 p->thread_info，然后返回 retval。将 p->user 的计数器__count 加 1，将 p->user 的所有进程计数器 processes 的加 1，将进程 p 的拥有组信息结构 group_info 的使用计数器 usage 加 1。如果系统当前的进程数 nr_threads 大于等于系统支持的最大进程数 max_threads 或者进程 p 的信息结构 thread_info 的可执行区域 exec_domain 的拥有者模块指针 module 非空但该模块处于不可被激活状态，检查 p 的拥有组信息结构 group_info 的使用计数器 usage 并将其减 1，如果原来 usage 不是 0 而且 p->group_info 的 block[0]不等于其 small_block，那么将 p->group_info 中的 blocks[0~nblocks]所指的内存页全部释放，然后释放 p->group_info 所占内存，再将 p->user->processes 减 1，执行 free_uid(p->user)释放 struct user_struct 结构 p->user 回到高速缓存内存 uid_cachep 中（但是 root_user 是不能释放的，因为 root_user 是声明时即被初始化的，不是从 uid_cachep 中申请到的），返回 retval。如果进程 p 的二进制描述结构指针 binfmt 非空，但该 binfmt 的拥有者模块处于不可被激活使用状态，执行 module_put(p->thread_info->exec_domain->module)，返回 retval。将进程 p 的已经执行标志位 did_exec 清 0；将 p->flags 中的超级用户权限标志位 PF_SUPERPRIV 清 0，再将 p->flags 或上 PF_FORKNOEXEC 标志位，表示进程 p 刚创建还未执行，如果 clone_flags 的没有设置 CLONE_PTRACE 标志位，将 p->ptrace 清 0。将 pid 传给 p->pid，将 retval 设置成-EFAULT。

如果 clone_flags 设置了 CLONE_PARENT_SETTID 标志位，将 pid 号复制到用户空间父进程的 id 指针 parent_tidptr 处，如果复制出错，且 p->binfmt 非空，执行 module_put(p->binfmt->module) 返回 retval。将进程 p 的 proc 文件系统目录条目结构 struct dentry 指针 proc_dentry 设置为 NULL，将进程 p 的子进程链表 children 和兄弟进程链表 sibling 初始化为初始状态，初始化进程 p 的子进程退出等待队列链表结构 wait_chldexit；将进程 p 的完成量结构 struct completion 指针 vfork_done 设置为 NULL。初始化进程 p 的分配自旋锁 alloc_lock 和 proc 文件系统自旋锁 proc_lock。清除 p->thread_info.flags 中的 TIF_SIGPENDING 标志位，将进程 p 的信号停靠描述结构（struct sigpending）pending 的 signal.sig[0~1]清 0，将 pending.list 设置为初始状态，将进程 p 的实际定时器、虚拟定时器、prof 定时器的当前计数值 it_real_value、p->it_virt_value、p->it_prof_value 全部清 0。将进程 p 的实际定时器、虚拟定时器、prof 定时器的初始计数值 it_real_incr、p->it_virt_ incr、p->it_prof_ incr 全部清 0。将进程 p 的实际定时器定时器列表描述结构（struct timer_list）real_timer 的 base 设置为 NULL，magic 设置为 TIMER_MAGIC()，lock 设置为初始状态。将 p 传给 p->real_timer.data，将进程 p 的微秒计数器 utime 和秒计数器 stime 都清 0，将 p 的 lock_depth 设置为-1，执行 do_posix_clock_monotonic_gettime(&p->start_time) 函数获取从上次 wall_jiffies 更新以来过去的时间，把秒数和纳秒数存入时间描述结构 struct timespec 变量 tp 中，再将 tp 中的秒数和纳秒数分别再加上 wall_to_monotonic 变量中的秒数和纳秒数，最后把得到的结果传给 p->start_time。将 p->security、io_contex、io_wait、audit_context 指针设置为 NULL。如果系统配置了 CONFIG_NUMA，而且 p->mempolicy 指针非空，执行 mpol_copy(p->mempolicy)申请一个新的内存访问结构 struct mempolicy 高速缓存空间，将老的内存访问结构变量复制到新的内存访问结构变量，再把新的内存访问结构变量指针传给 p->mempolicy，如果 p->mempolicy 指针错误，将 retval 设置为 p->mempolicy，将 p->mempolicy 设置为 NULL，返回 retval。将 p->pid 传给 p->tgid，如果 clone_flags 设置了 CLONE_THREAD 即父子进程在同一个线程组中，将 current->tgid 传给 p->tgid。执行 security_task_alloc(p)并把返回值传给 retval，如果 retval 不等于 0 而且系统配置了 CONFIG_NUMA，那么执行 mpol_free(p->mempolicy)，返回 retval。执行 audit_alloc(p)（该函数在没有配置 CONFIG_AUDITSYSCALL 选项的系统中直接返回 0，Linux2.6.10 中只有 X86、PPC64、ARCH_S390、IA64 等系统才能配置 CONFIG_AUDITSYSCALL 选项）并将返回值传给 retval，如果 retval 不等于 0，执行 security_task_free(p)，返回 retval。执行 copy_semundo(clone_flags, p) 将 current->sysvsem.undo_list 复制给 p->sysvsem.undo_list 并把返回值传给 retval，如果 retval 不等于 0，执行 audit_free(p)，返回 retval。执行 copy_files(clone_flags, p)根据情况设置 p->files 并把返回值传给 retval，如果 retval 不等于 0，执行 exit_sem(p)，返回 retval。执行 copy_fs(clone_flags, p)设置 p->fs 并把返回值传给 retval，如果 retval 不等于 0，执行 exit_files(p)，返回 retval；执行 copy_sighand(clone_flags, p)设置 p->sighand 并把返回值传给 retval，如果 retval 不等于 0，执行 exit_fs(p)，返回 retval。执行 copy_signal(clone_flags, p)设置 p->signal 并把返回值传给 retval，如果 retval 不等于 0，执行 exit_sighand(p)，返回 retval。执行 copy_mm(clone_flags,

p)设置 p->mm 并把返回值传给 retval，如果 retval 不等于 0，执行 exit_signal(p)，返回 retval。执行 copy_keys(clone_flags, p)并把返回值传给 retval，如果 retval 不等于 0，如果 p->mm 非空，执行 mmput(p->mm)，返回 retval；执行 copy_namespace(clone_flags, p)设置 p->namespace 并把返回值传给 retval，如果 retval 不等于 0，执行 exit_keys(p)，返回 retval。执行 copy_thread(0, clone_flags, stack_start, stack_size, p, regs)设置 p->thread_info 并把返回值传给 retval，如果 retval 不等于 0，exit_namespace(p)，返回 retval。如果 clone_flags 设置了 CLONE_CHILD_SETTID 标志，将 child_tidptr 传给 p->set_child_tid，否则 p->set_child_tid 设置为 NULL。如果 clone_flags 设置了 CLONE_CHILD_CLEARTID 标志，将 child_tidptr 传给 p->clear_child_tid，否则 p->clear_child_tid 设置为 NULL。清除 p->thread_info->flags 中的 TIF_SYSCALL_TRACE 标志位，不跟踪系统调用。把本进程的执行区域 id 号 p-> self_exec_id 传给父进程执行区域 id 号 p->parent_exec_id。如果 clone_flags 设置了 CLONE_THREAD 标志位即父子进程共享同一线程组，将 p->exit_signal 设置为-1，否则将 clone_flags & CSIGNAL 传给 p->exit_signal，也就是说 clone_flags 的低 8 位是进程退出时要传递的信号标志位。将该杀死父进程时要发送的信号 p->pdeath_signal 设置为 0，将退出状态标志 p->exit_state 设置为 0。执行 sched_fork(p)函数设置进程调度相关字段，将 p 传给 p->group_leader，初始化该进程的跟踪子进程链表 p->ptrace_childre 和跟踪链表 p->ptrace_list，禁止 IRQ 中断，禁止抢占，将任务链表读写锁 tasklist_lock 写上锁。将 current->cpus_allowed 传给 p->cpus_allowed，将当前处理器的 id 号 smp_processor_id() 传给 p->thread_info->cpu。如果 current->pending.signal 中设置了 SIGKILL，即要立即结束当前进程的运行，将任务链表读写锁 tasklist_lock 写解锁，使能 IRQ 中断，使能抢占，返回-EAGAIN；如果 clone_flags 设置了 CLONE_PARENT 或设置了 CLONE_THREAD 标志，即要设置同一个共有父进程或者共享同一个线程组，那么将 current->real_parent 传给 p->real_parent，否则将 current 传给 p->real_parent。如果 clone_flags 设置了 CLONE_THREAD 标志，将 current->sighand->siglock 上锁，如果 current->signal->group_exit 不等于 0，将 current->sighand->siglock 解锁，禁止 IRQ 中断，禁止抢占，将任务链表读写锁 tasklist_lock 写上锁，返回-EAGAIN，将 current->group_leader 传给 p->group_leader，如果 current->signal->group_stop_count 大于 0，将 current->signal->group_stop_count 加 1，将设置 p->thread_info.flags 中的 TIF_SIGPENDING 位，将 current->sighand->siglock 解锁。将该进程 p 通过 p->sibling 链接到父进程 p->parent->children 链表末尾，如果 p->pid 等于 p->tgid，还得将该进程通过 p->tasks 链接到 init_task.tasks 链表末尾。如果 p->ptrace 不幸设置了 PT_PTRACED 标志，如果 p->ptrace_list 链表非空，则系统崩溃，如果 p->parent 等于 current->parent，不做下面的操作直接跳到下一段，否则将该进程通过 p->ptrace_list 链接到 p->parent->ptrace_children 链表开始处，将该进程从其所在的 p->tasks 链表和 p->sibling 链表中删除，将 current->parent 传给 p->parent，重新将 p 添加到 init_task.tasks 链表末尾和 p->parent->children 链表末尾。设置 p->pids[PIDTYPE_PID]，并根据 p->pid 号是否已经被使用过将其链接到 hash 链表 pid_hash[PIDTYPE_PID][pid_hashfn(p->pid)]的适当位置。设置 p->pids[PIDTYPE_TGID]，并根据 p->tgid 号是否已经被

使用过将其链接到 hash 链表 pid_hash[PIDTYPE_TGID] [pid_hashfn(p->tgid)]的适当位置。如果 p->pid 等于 p->tgid，设置 p->pids[PIDTYPE_PGID]，并根据 p->signal->pgrp 号是否已经被使用过将其链接到 hash 链表 pid_hash[PIDTYPE_PGID] [pid_hashfn(p->signal->pgrp)]的适当位置，设置 p->pids[PIDTYPE_SID]，并根据 p->signal->session 号是否已经被使用过将其链接到 hash 链表 pid_hash[PIDTYPE_SID] [pid_hashfn(p->signal->session)]的适当位置。如果 p->pid 不等于 0，将当前 CPU 的处理器变量 per_cpu_process_counts 加 1。将系统中的进程数 nr_threads 加 1，将任务链表读写锁 tasklist 写解锁，使能 IRQ 中断和抢占，返回 0。

static struct task_struct *dup_task_struct(struct task_struct *orig)函数执行过程：执行空函数 prepare_to_copy(orig)，执行 alloc_task_struct()从 struct task_struct 结构高速缓存内存 task_struct_cachep 中申请一个新的 struct task_struct 结构对象，将其地址传给局部指针 tsk。如果 tsk 指针空，则返回 NULL。执行 alloc_thread_info(tsk)函数申请一个新的 struct thread_info 结构内存空间（两页）并将其起始地址传给局部指针 ti。如果 ti 指针空，将刚申请到的 task_struct 结构 tsk 释放回高速缓存内存 task_struct_cachep 中，返回 NULL。将*orig->thread_info 传给*ti，即将 orig 进程的 thread_info 复制到 ti 所指空间中。将*orig 传给*tsk，将 orig 进程的描述结构复制到 tsk 所指空间中。将 ti 传给 tsk->thread_info，将 tsk 传给 ti->task，将 tsk 的使用计数器 usage 设置为 2，返回 tsk 指针。

struct thread_info *alloc_thread_info(struct task_struct *task)函数执行过程：如果 EXTRA_TASK_STRUCT（=4，只有 arm 系统中有定义）不等于 0，将 thread_info_head 传给局部指针 p，如果 p 非空，将 p[0]传给 thread_info_head 并将 nr_thread_info 减 1，将 p 传给局部 struct thread_info 结构指针 thread。如果 thread 指针为空，调用 ll_alloc_task_struct()-> __get_free_pages(GFP_KERNEL,1)申请 2 页（刚好一个 struct thread_info 结构空间大小）物理内存空间，将申请到的内存空间起始地址传给 thread，将 thread 所指内存空间的第 2 页全部请 0，返回 thread 指针。

int do_posix_clock_monotonic_gettime(struct timespec *tp) 函数执行过程：执行 do_posix_clock_monotonic_gettime_parts(tp, &wall_to_mono)函数，获取从上次 wall_jiffies 更新以来过去的时间，把秒数和纳秒数存入时间描述结构变量 tp 中，wall_to_mono 设置为 wall_to_monotonic，将 tp->tv_set 加上 wall_to_mono.tv_sec，将 tp->tv_nsec 加上 wall_to_mono.tv_nsec，如果 tp->tv_nsec 大于 NSEC_PER_SEC（=1000000000），将 tp->tv_nsec 减去 NSEC_PER_SEC，将 tp->tv_set 加 1，返回 0。

static u64 do_posix_clock_monotonic_gettime_parts(struct timespec *tp, struct timespec *mo)函数执行过程：读取定时器顺序锁结构（seqlock_t）xtime_lock.sequence 的值传给 seq，执行 getnstimeofday(tp)函数获取从上次 wall_jiffies 更新以来过去的时间，把秒数和纳秒数分别存入 tp->tv_sec 和 tp->tv_nsc 中，将 mo 指向 wall_to_monotonic，将 jiffies_64 传给 jiff，再次读取 xtime_lock.sequence 的值，如果这次读取的值是奇数或者与上次读取的值 seq 不等，重复以上

操作。返回 jiff。

void getnstimeofday(struct timespec *tv)函数执行过程：该函数根据是否配置了 CONFIG_TIME_INTERPOLATION 选项有不同定义，CONFIG_TIME_INTERPOLATION 选项只有在 ia64 和 sparc64 系统中可配置，在没有配置 CONFIG_TIME_INTERPOLATION 选项的系统中该函数定义如下：执行 do_gettimeofday(&x)函数，其中 x 为时间值结构 struct timeval 局部变量，获取从上次 wall_jiffies 更新以来过去的时间，把秒数和微秒数分别存入时间值变量 x.tv_sec 和 x.tv_usec 中。将 x.tv_sec 传给 tv->tv_sec，将 x.tv_usec* NSEC_PER_USEC（=1000）传给 tv->tv_nsec。

void do_gettimeofday(struct timeval *tv)函数执行过程：禁止 IRQ 中断，保护现场，读取定时器顺序锁结构（seqlock_t）xtime_lock.sequence 的值传给 seq；执行 system_timer 的 offset()成员函数，对于 ixp4xx 处理器平台，该成员函数为 ixp4xx_gettimeoffset()函数，获取系统从上一次硬件“滴答”到现在经过的时间微秒数传给变量 usec。将 jiffies 减去 wall_jiffies 传给 lost，如果 lost 大于 0，将 usec 加上 lost * USECS_PER_JIFFY（=1000000/HZ），将 xtime.tv_sec 传给 sec，将 usec 加上 xtime.tv_usec/1000。再次读取 xtime_lock.sequence 的值，使能 IRQ 中断，恢复现场，如果这次读取的值是奇数或者与上次读取的值 seq 不等，那么重复前面的操作。将 usec 换算成秒数加到 sec 上，将 sec 传给 tv->tv_sec，将 usec 传给 tv->tv_usec。

static unsigned long ixp4xx_gettimeoffset(void)函数执行过程：返回时间戳偏移量寄存器 IXP4XX_OSTS 的值减去上一次软件“滴答”时的时间值 last_jiffy_time，单位为微秒。

int copy_semundo(unsigned long clone_flags, struct task_struct *tsk)函数执行过程：该函数在没有配置 CONFIG_SYSVIPC 选项的系统中直接返回 0；所谓 SYSVIPC 是 system virtual inter process communication 即“系统虚拟进程间通信”的缩写，它由一系列让进程（正在运行的程序）间同步和交换信息的库函数和系统调用组成，CONFIG_SYSVIPC 选项开关很有用，通常都打开，如果想在 Linux 下使用 DOS 必须配置该选项。在配置了 CONFIG_SYSVIPC 选项的系统中的定义如下：如果 clone_flags 没有设置 CLONE_SYSVSEM 标志位，即父子进程不共享 struct sem_undo_list 结构，将 tsk->sysvsem.undo_list 设置为 NULL，直接返回 0。将当前进程 current 的系统虚拟信号量 sysvsem 的无为链表结构指针 undo_list 传给 struct sem_undo_list 结构指针 undo_list，如果 undo_list 指针为空，调用 kmalloc()为 struct sem_undo_list 结构申请内存空间并将申请到的地址传给 undo_list，如果 undo_list 指针空，返回没有内存错误，将 undo_list 所指 struct sem_undo_list 结构空间清 0，将 undo_list->refcnt 设置为 1，将 undo_list 传给 current->sysvsem.undo_list，将 undo_list 传给 undo_list。如果 undo_list->refcnt 等于 1，初始化 undo_list->lock，将 undo_list->refcnt 加 1，将 undo_list 传给 tsk->sysvsem.undo_list，返回 0。

static int copy_files(unsigned long clone_flags, struct task_struct * tsk)函数执行过程：将当前任务 current 的打开文件描述结构（struct files_struct）指针 files 传给 oldf，如果 oldf 空，返回 0。

如果 clone_flags 设置了 CLONE_FILES，即父子进程共享打开的文件描述结构，将 oldf->count 加 1，返回 0。将 tsk->files 设置为 NULL，从 struct files_struct 结构高速缓存内存 files_cachep 中申请一个新的 struct files_struct 结构，将其地址传给 newf，如果 newf 指针空，返回没有内存错误。将 newf->count 设置为 1，newf->file_lock 设置为初始状态，newf->next_fd 设置为 1，newf->max_fds 设置为 NR_OPEN_DEFAULT（=32），newf->max_fdset 设置为__FD_SETSIZE（=1024），newf->close_on_exec 指向 newf->close_on_exec_init，newf->open_fds 指向 newf->open_fds_init，newf->fd 指向 newf->fd_array[0]。如果 oldf->max_fdset 大于__FD_SETSIZE，将 newf->max_fdset 设置为 0，将 newf->file_lock 上锁，执行 expand_fdset(newf, oldf->max_fdset-1) 扩展 newf 的 max_fdset，将 newf->file_lock 上锁解锁，如果扩展失败，释放 newf->close_on_exec 与 newf->open_fds 所指空间的多余空间，释放 newf 回 files_cachep，返回错误。将 oldf->file_lock 上锁。根据 oldf->open_fds->fds_bits 中不为 0 的字的个数统计 oldf 打开的文件数目传给 open_files，如果 open_files 大于 NR_OPEN_DEFAULT，将 oldf->file_lock 解锁，将 newf->max_fds 设置为 0，将 newf->file_lock 上锁，执行 expand_fd_array(newf, open_files-1)扩展 newf->maxfds，将 newf->file_lock 解锁，如果扩展 newf->maxfds 出错，释放 newf->close_on_exec 与 newf->open_fds 所指空间的多余空间，将 newf 释放回 files_cachep，返回错误，将 newf->maxfds 传给 nfds，将 oldf->file_lock 上锁。将 oldf->fd 传给 old_fds，将 newf->fd 传给 new_fds，将 oldf->open_fds->fds_bits 中的数据复制到 newf->open_fds->fds_bits 中，将 oldf->close_on_exec->fds_bits 中的数据复制到 newf->close_on_exec->fds_bits，查看 old_fds 处的所有 struct file 指针 f，如果指针为空，将 newf->open_fds 中对应的位清 0，否则将 f->f_count 加 1，将 f 传给对应的 new_fds 处。将 oldf->file_lock 解锁，将 new_fds 指针所指最后的未用的位码字清 0。如果 newf->max_fdset 大于 open_files，将 newf->open_fds 中未用的位码字清 0，将 newf->close_on_exec 中未用的位码字清 0，将 newf 传给 tsk->files，返回 0。

static inline int copy_fs(unsigned long clone_flags, struct task_struct * tsk)函数执行过程：如果 clone_flags 设置了 CLONE_FS 标志即父子进程共享文件系统描述结构 struct fs_struct，将当前进程 current 的文件系统描述结构 fs 的引用计数器 count 加 1，返回 0。执行__copy_fs_struct(current->fs)函数并将其返回值传给 tsk->fs；如果 tsk->fs 指针为空，返回没有内存错误。

static inline struct fs_struct *__copy_fs_struct(struct fs_struct *old)函数执行过程：从 struct fs_struct 结构高速缓存内存 fs_cachep 中申请一个新的 struct fs_struct 结构，将其指针传给 fs，如果 fs 非空，将 fs->count 设置为 1，初始化 fs->lock 读写锁，将 old->umask 传给 fs->umask。将 old->lock 读上锁；将 old->rootmnt.mnt_count 加 1 传给 fs->rootmnt，将 old->root.d_count 加 1 传给 fs->root，old->pwdmnt.mnt_count 加 1 传给 fs->pwdmnt，将 old->pwd.d_count 加 1 传给 fs->pwd。如果 old->altroot 指针存在，将 old->altrootmnt.mnt_count 加 1 传给 fs->altrootmnt，将 old->altroot.d_count 加 1 传给 fs->altroot，否则将 fs->altrootmnt 和 fs->altroot 设置为 NULL。将 old->lock 读解锁，返回 fs。

static inline int copy_sighand(unsigned long clone_flags, struct task_struct * tsk)函数执行过程：如果 clone_flags 设置了 CLONE_SIGHAND 或者 CLONE_THREAD，即父子共享信号处理或共享同一线程组，将 current->sighand->count 加 1，返回 0。从高速缓存内存 sighand_cachep 申请一个新的 struct sighand_struct 结构，将其指针传给 sig。将 sig 传给 tsk->sighand。如果 sig 为空，返回没有内存错误。初始化 sig->siglock 锁，将 sig->count 设置为 1，将 current->sighand->action 所指空间复制到 sig->action 中，返回 0。

static inline int copy_signal(unsigned long clone_flags, struct task_struct * tsk)函数执行过程：如果 clone_flags 设置了 CLONE_THREAD 标志即父子进程共享同一线程组，那么将 current->signal->count 加 1，current->signal->live 加 1，返回 0。从高速缓存内存 signal_cachep 中申请一个新的 struct signal_struct 结构空间并将其地址传给指针 sig，再将 sig 传给 tsk->signal。如果 sig 指针为空，返回没有内存错误。将 sig->count 和 sig->live 都设置为 1，将 group_exit 设置为 0，将 sig->group_exit_code 设置为 0，将 sig->group_exit_task 设置为 NULL，将 sig->group_stop_count 设置为 0，将 sig->stop_state 设置为 0，将 sig->curr_target 设置为 NULL，将 sig->share_pending.signal 清 0，初始化其链表 sig->share_pending.list，初始化定时器链表结构 sig->posix_timers，将 current->signal->tty 指针传给 sig->signal，将 currrent->signal-> pgrp 传给 sig-> pgrp，将 currrent->signal->session 传给 sig->session，将 sig->leader 设置为 0，将 sig->tty_old_pgrp 设置为 0，将 utime、stime、cutime、cstime、nvcsw、nivcsw、cnvcsw、cnivcsw、min_flt、maj_flt、cmin_flt、cmaj_flt 设置为 0，将当前任务的线程组领头进程描述结构 current->group_leader-> alloc_lock 上锁，将 current->signal->rlim 整个复制到 sig->rlim 处，将当前任务的线程组领头进程描述结构 current->group_leader-> alloc_lock 解锁，返回 0。

static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)函数执行过程：tsk->min_flt 和 tsk->maj_flt 设置为 0，将 tsk->mm 和 tsk->active_mm 设置为 NULL，将 current->mm 传给 oldmm，如果 oldmm 指针为空，返回 0；如果 clone_flags 设置了 CLONE_VM 标志即父子进程共享虚拟内存空间，将 oldmm->mm_users 加 1，将 oldmm 传给 mm，等到 oldmm->page_table_lock 锁被解开，再将 mm 传给 tsk->mm 和 tsk->active_mm，返回 0。将 retval 设置为-ENOMEM。从高速缓存内存 mm_cachep 申请一个新的 struct mm_struct 结构空间并将其地址传给 mm，如果 mm 为空，返回 retval。将 oldmm 处 struct mm_struct 结构数据复制到 mm 处。将 mm->mm_users 设置为 1，将 mm->mm_count 设置为 1，初始化读写信号量 mm->mmap_sem，初始化链表 mm->mmlist，将 mm->core_waiters 设置为 0，将 mm->nr_ptes 设置为 0，初始化 mm->page_table_lock 锁，初始化读写锁 mm->ioctx_list_lock，将 mm->ioctx_list 设置为空，执行初始化宏 INIT_KIOCTX(mm->default_kioctx, *mm)，将 mm->free_area_cache 设置为 TASK_UNMAPPED_BASE（=0x40000000），执行 mm_alloc_pgd(mm)申请页表空间并将其地址传给 mm->pgd，如果申请失败，将 mm 释放回 mm_cachep，返回 retval，将 mm->def_flags 设置为 0。如果是 ARMv6 以上处理器，将 mm-> context.id 设置为 0。执行 dup_mmap(mm, oldmm)

并将返回值传给 retval，如果 retval 不等于 0，执行 mmput(mm)，返回 retval。将 mm 传给 tsk->mm 和 tsk->active_mm，返回 0。

static inline int mm_alloc_pgd(struct mm_struct * mm)函数执行过程：执行 pgd_alloc(mm)即 get_pgd_slow(mm)函数并将返回值传给 mm->pgd。如果 mm->pgd 为空，返回没有内存错误，返回 0。

pgd_t *get_pgd_slow(struct mm_struct *mm)执行过程：执行__get_free_pages(GFP_KERNEL, 2)申请 4 页（4GB 的存储空间的一级页描述符表占 4 页内存空间）连续内存并将内存起始地址传给 pgd_t 指针 new_pgd。如果 new_pgd 为空，返回 NULL。将 new_pgd 所指的 TASK_SIZE/PGDIR_SIZE* sizeof(pgd_t)大小的空间清 0（这里要映射 0~TASK_SIZE 之间的虚拟内存空间了）。通过 pgd_offset_k(0)将虚拟地址 0 对应的一级描述符存放地址传给 init_pgd，这里的一级描述符地址是在 init_mm.pgd 所指空间内，而不是 mm->pgd 所指空间内。如果 CP15 的 C1 寄存器的 bit13 等于 0 即系统使用低端中断向量，将 mm->page_table_lock 上锁，将 new_pgd 传给 pmd_t 指针 new_pmd，执行 pte_alloc_map(mm, new_pmd, 0)为从虚拟地址 0 开始的 2MB 内存空间对应的二级页表申请一页内存空间，设置虚拟地址 0 开始的 2MB 存储空间对应的一级描述符（因为低端中断向量地址在 0x0~0x20 范围，所以必须对包含这段空间的双段进行映射），返回指向虚拟地址 0 对应的 Linux 版本二级页表的起始地址传给 new_pte，如果 new_pte 指针空，将 new_pgd 和 new_pmd 都释放掉，返回 NULL，将 init_pgd 传给 init_pmd，将 init_pmd 页表中（即系统启动时的映射表）虚拟地址 0 对应的 Linux 版本二级页表起始地址传给 init_pte，执行 set_pte(new_pte, *init_pte)将 init_pte 处的 Linux 版本和硬件版本二级页描述符条目存入到 new_pte 中（这里只映射了虚拟地址 0 开始的 1 页内存空间，而且映射关系保留原来的设置，只是把原来的二级页描述符搬到了新进程中，可是这样有点浪费，一页空间的映射 Linux 版本和硬件版本二级页描述符只需要总共 8 字节，这里却申请了一页内存来存放这 8 字节，太奢侈了，如果每个进程都不共虚拟内存空间，都建立一张单独的页表，那浪费太大了，所以最好还是用高端中断向量），执行两个空函数 pte_unmap_nested(init_pte)和 pte_unmap(new_pte)，将 mm->page_table_lock 解锁。将 init_pgd（swapper_pg_dir）处 FIRST_KERNEL_PGD_NR 位置开始(PTRS_PER_PGD - FIRST_KERNEL_PGD_NR) * sizeof(pgd_t)大小的数据复制到 new_pgd 处 FIRST_KERNEL_PGD_NR 位置开始的地方，也就是把 init_task 的 TASK_SIZE 地址以上的虚拟空间的一级页表复制到新进程的一级页表中。清空 new_pgd 处 16KB 一级页表空间的数据 cache 块，但不刷新。返回 new_pgd。

pte_t fastcall * pte_alloc_map(struct mm_struct *mm, pmd_t *pmd, unsigned long address)函数执行过程：如果 pmd 处的 pmd_t 条目为 0，将 mm->page_table_lock 解锁，执行 pte_alloc_one(mm, address)为二级页表申请一页内存空间，将申请到的内存页的 struct page 结构指针传给 new，将 mm->page_table_lock 上锁，如果 new 空，返回 NULL。如果 pmd 处 pmd_t 条目不等于 0，释放 new 所指内存页，返回虚拟地址 address 对应的 Linux 版本二级页表地址，

将 mm->nr_ptes 加 1，将当前 CPU 的页状态结构变量 per_cpu_page_state 中的二级页表内存页计数器 nr_page_table_pages 加 1。执行 pmd_populate(mm, pmd, new)设置 address 所在 2MB 内存空间的一级描述符。返回虚拟地址 address 对应的 Linux 版本二级页表地址。

static inline struct page * pte_alloc_one(struct mm_struct *mm, unsigned long addr)函数执行过程：调用 alloc_pages(GFP_KERNEL|__GFP_REPEAT, 0)申请一页内存空间并将内存起始地址传给 struct page 结构指针 pte，如果 pte 指针非空，执行 page_address(pte)将获取 pte 所描述的内存空间的虚拟起始地址传给 pte_t 指针 page，将 page 页清 0，清空该页前 2KB（硬件版本二级页表）内存的数据 cache，但不刷新。返回 pte 指针。

static inline void pmd_populate(struct mm_struct *mm, pmd_t *pmdp, struct page *ptep)函数执行过程：如果 mm 指向 init_mm，则系统崩溃。将 ptep 所描述的物理内存的物理地址或上 _PAGE_USER_TABLE 传给 pmdval 作为其所在地址的一级描述符条目。将 pmdval 传给 pmdp[0]，将 pmdval + 256 * sizeof(pte_t)传给 pmdp[1]。调用 flush_pmd_entry(pmdp)刷新 pmdp 地址处的 cache 和 Write Buffer。

static inline int dup_mmap(struct mm_struct * mm, struct mm_struct * oldmm)函数执行过程：在没有配置 CONFIG_MMU（即不支持 MMU）的系统中，该函数直接返回 0。在配置了 CONFIG_MMU 的系统中执行如下：将 oldmm->mmap_sem 读写信号量写减少。如果当前处理器在 mm->cpv_vm_mask 中的位码被设置为 1，执行 cpu_cache.flush_user_all 使指令 cache 无效，清空写缓存，将 mm->locked_vm 设置为 0。将 mm->mmap 和 mm->mmap_cache 设置为 NULL，将 oldmm->mmap_base 传给 mm->free_area_cache，将 mm->count、mm->rss、mm->anon_rss 清 0，将 mm->cpu_vm_mask 清 0，将 RB_ROOT 传给 mm->mm_rb，将 struct rb_node **rb_link 指向 mm->mm_rb.rb_node，将 struct rb_node *rb_parent 设置为 NULL，将 struct vm_area_struct **pprev 指向 mm->mmap。“循环开始位置”：查看 current->mm->mmap.vm_next 链表中的每一个 struct vm_area_struct 结构，首先将 current->mm->mmap 传给 struct vm_area_struct *mpnt，如果 mpnt->vm_flags 设置了 VM_DONTCOPY 标志即该结构不能被复制，如果 mpnt->vm_file 指针非空，那么将 mm->shared_vm 减去((mpnt->vm_end - mpnt->vm_start) >> PAGE_SHIFT)（之所以减是因为 mm 是从 oldmm 中复制过来的，所有的成员都是从父进程继承过来的，已经初始化，现在是在校正这些设置），如果 mpnt->vm_flags & (VM_EXEC|VM_WRITE)) == VM_EXEC，将 mm->exec_vm 也减去((mpnt->vm_end - mpnt->vm_start) >> PAGE_SHIFT)，如果 mpnt->vm_file 指针空并且(mpnt->vm_flags&VM_STACK_FLAGS &(VM_GROWSUP| VM_GROWSDOWN))不等于 0，那么只将 mm->stack_vm 减去((mpnt->vm_end - mpnt->vm_start) >> PAGE_SHIFT)，如果 mpnt->vm_flags&(VM_RESERVED|VM_IO)不等于 0，将 mm->reserved_vm 减去((mpnt->vm_end - mpnt->vm_start) >> PAGE_SHIFT)，回到循环开始位置查看 current->mm->mmap.next 链表中下一个结构。将 charge 设置为 0，如果 mpnt->vm_flags 设置了 VM_ACCOUNT，将(mpnt->vm_end - mpnt->vm_start) >> PAGE_SHIFT 传给 charge，执行

security_vm_enough_memory(charge)如果返回非 0，跳转到“fail_nomem 标号位置”。调用 kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL)申请一个 struct vm_area_struct 结构高速缓存内存并把申请到的内存地址传给 struct vm_area_struct *tmp；如果 tmp 空，跳转到“fail_nomem 标号位置”。将*mpnt 传给*tmp。如果 mpnt->vm_policy 不是空，申请一个新的内存访问政策描述结构 struct mempolicy 高速缓存空间，将 mpnt->vm_policy 的内容全部复制到新的结构中，再把新的结构指针传给 pol。将 retval 设置成 PTR_ERR(pol)，如果 pol 地址有错，跳转到“fail_nomem_policy 标号位置”。将 pol 传给 tmp->vm_policy，清除 tmp->vm_flags 中的 VM_LOCKED 标志位。将 mm 传给 tmp->vm_mm，将 tmp->next 设置为 NULL。如果 tmp->anon_vma 指针不空，将 tmp->anon_vma->lock 上锁，将 tmp 通过其 anon_vma_node 链接到自身的 anon_vma->head 链表中，执行 rmap 调试（RMAP_DEBUG）用函数 validate_anon_vma(vma)，将 tmp->anon_vma->lock 解锁，将 tmp->vm_file 传给 struct file *file。如果 file 指针不空，将 file->file->f_dentry->d_inode 传给 struct inode *inode，将 file->f_count 加 1，如果 tmp->vm_flags 设置了 VM_DENYWRITE 标志即拒绝写，将 inode->i_writecount 减 1，将 file->f_mapping->i_mmap_lock 上锁。将 tmp->shared.vm_set.head 和 tmp->shared.vm_set.parent 设置为 NULL，如果 mpnt->shared.vm_set.parent 空，将 tmp->shared.vm_set.list 链接到 mpnt->shared.vm_set.list 链表开头，如果 mpnt->shared.vm_set.head（mpnt 的父结构）指针不空，则将 tmp->shared.vm_set.list 链接到 mpnt->shared.vm_set.head->shared.vm_set.list 末尾，如果以上两个条件都不成立，初始化 tmp->shared.vm_set.list，将 mpnt 传给 tmp->shared.vm_set.head，将 tmp 传给 mpnt->shared.vm_set.head；禁止 IRQ 中断保护现场，将 file->f_mapping->tree_lock 上锁，将 file->f_mapping->tree_lock 上锁，使能 IRQ 中断恢复现场。将 file->f_mapping->i_mmap_lock 解锁，将 mm->page_table_lock 上锁，将 tmp 传给*pprev，将 pprev 指向 tmp->vm_next，将 tmp 传给 mm->mmap，将 pprev 指向 tmp->vm_next。执行__vma_link_rb(mm, tmp, rb_link, rb_parent)，将 rb_link 指向 tmp->vm_rb.rb_right，将 rb_parent 指向 tmp->vm_rb，将 m->mmap_count 加 1。执行 copy_page_range(mm, current->mm, tmp)函数并把返回值传给 reval，将 mm->page_table_lock 解锁。如果 tmp->vm_ops->open 函数有定义，执行 tmp->vm_ops->open(tmp)。如果 retval 不等于 0，跳转到“out 标号位置”。回到循环开始位置查看 current->mm->mmap.next 链表中下一个结构；“循环结束位置”：retval 设置为 0；“out 标号位置”：清除系统 Write Buffer，使系统指令和数据 TLBs 无效，将存储访问失效寄存器 CP15，C6 清 0。将 oldmm->mmap_sem 读写信号量写增加，返回 retval。“fail_nomem_policy 标号位置”：释放高速缓存 tmp 回 vm_area_cachep。“fail_nomem 标号位置”：将 retval 设置为- ENOMEM，执行 vm_unacct_memory(charge)，跳转到 out 标号处。

int copy_keys(unsigned long clone_flags, struct task_struct *tsk)函数执行过程：该函数在没有配置 CONFIG_KEYS 的系统中是空函数。

int copy_namespace(int flags, struct task_struct *tsk)函数执行过程：将 tsk->namespace 传给

struct namespace 结构指针 namespace，struct vfsmount 结构指针 rootmnt、pwdmnt、altrootmnt 设置为 NULL，tsk->fs 传给 struct fs_struct 结构指针 fs。如果 namespace 指针为空，返回 0。将 namespace->count 加 1，如果 flags 没有设置 CLONE_NEWNS 标志即不为子进程创建新的 struct namespace 结构，返回 0；如果当前任务 current 的有效能力标志字 cap_effective 没有设置管理者权限标志位 CAP_SYS_ADMIN，将 namespace 处理掉，返回不允许错值-EPERM。调用 kmalloc(sizeof(struct namespace), GFP_KERNEL)申请一个新的 struct namespace 结构空间，将其地址传给 new_ns，如果 new_ns 为空，将 namespace 处理掉，返回没有内存错-ENOMEM。将 new_ns->count 设置为 1，初始化 new_ns->sem 读写信号量，初始化 new_ns->list 链表，将 tsk->namespace->sem 读写信号量写减少。执行 copy_tree(namespace->root, namespace->root->mnt_root)将父进程的名字空间的挂接和目录树复制过来，将返回的 struct vfsmount 结构指针传给 new_ns->root。如果 new_ns->root 空，将 tsk->namespace->sem 读写信号量写增加，执行 kfee(new_ns)释放 new_ns 空间，将 namespace 处理掉，返回没有内存错-ENOMEM。将 vfsmount_lock 上锁，将 new_ns 通过 list 链接到 new_ns->root 的 mnt_list 链表末尾，将 vfsmount_lock 解锁。将 namespace->root 传给 struct vfsmount 结构指针 p，new_ns->root 传给 q；如果 p 不空，将 new_ns 传给 q->mnt_namespace，如果 fs 非空，如果 p 等于 fs->rootmnt，将 p 传给 rootmnt，将 q->mnt_count 加 1，将 q 传给 fs->rootmnt，如果 p 等于 fs->pwdmnt，将 p 传给 pwdmnt，将 q->mnt_count 加 1，将 q 传给 fs->pwdmnt，如果 p 等于 fs->altrootmnt，将 p 传给 altrootmnt，将 q->mnt_count 加 1，将 q 传给 fs->altrootmnt，查找挂接结构变量 namespace->root 所在挂接结构树的中的下一个挂接结构变量传给 p，查询的顺序是，先查子挂接结构变量，其次同级挂接结构变量，最后是父挂接结构变量，查找挂接结构变量 new_ns->root 所在挂接结构树的中的下一个挂接结构变量传给 q 重复本段操作。将 tsk->namespace->sem 读写信号量写增加，将 new_ns 传给 tsk->namespace。如果 rootmnt 非空，将其处理掉，如果 pwdmnt 非空，将其处理掉，如果 altrootmnt 非空，将其处理掉。将 namespace 处理掉，返回 0。

int copy_thread(int nr, unsigned long clone_flags, unsigned long stack_start, unsigned long stk_sz, struct task_struct *p, struct pt_regs *regs)函数执行过程：将 p->thread_info 传给 struct thread_info 结构指针 thread；将((struct pt_regs *)((unsigned long)thread + THREAD_SIZE – 8)) – 1 传给 struct pt_regs 结构指针 childregs。将*regs 传给*childregs，将 childregs->ARM_r0 设置为 0，将 childregs->ARM_sp 设置为 stack_start，将 thread->cpu_context 全部清 0，将 childregs 传给 thread->cpu_context.sp，将 ret_from_fork()函数指针传给 thread->cpu_context.pc，返回 0。

void fastcall sched_fork(task_t *p)函数执行过程：将 TASK_RUNNING 传给 p->state，将进程 p 设置为正在运行或在就绪队列 run-queue 中准备运行状态，实际参与进程调度。初始化该进程的运行链表 p->run_list，将该进程的优先级阵列结构指针 p->array 设置为 NULL。如果配置了 CONFIG_SCHEDSTATS 选项，将 p->sched_info 结构全部清 0。如果配置了 CONFIG_PREEMPT 选项，将 p->thread_info->preempt_count 设置为 1，禁止被抢占。禁止 IRQ

中断，将当前进程的时间片（current->time_slice+1）>>1 作为该进程的时间片 p->time_slice。将 p->first_time_slice 设置为 1，将 jiffies * (1000000000 / HZ)传给该进程的时间戳 p->timestamp，也就是说进程的时间戳实际就是从系统启动到该进程被创建系统运行的纳秒数。如果 current->time_slice 等于 0，将 current->time_slice 设置为 1，禁止抢占，执行 scheduler_tick(0, 0)，使能 IRQ 中断，使能抢占，返回。如果 current->time_slice 不等于 0，使能 IRQ 中断，返回。

void scheduler_tick(int user_ticks, int sys_ticks)函数执行过程：将当前运行的处理器号 smp_processor_id()传给 CPU。将当前处理器的内核状态结构的处理器使用状态描述结构 kstat_this_cpu.cpustat 的地址传给 struct cpu_usage_stat *cpustat。把当前处理器的运行队列 per_cpu_runqueue 的地址传给 runqueue_t *rq。将 current 传给 struct task_struct *p，将系统从启动到现在运行的纳秒数 sched_clock()传给 rq->timestamp_last_tick。如果当前处理器的（per_cpu_rcu_data.curlist 非空并且 rcu_ctrlblk.completed 大于等于 per_cpu_rcu_data.batch）或者（per_cpu_rcu_data.curlist 空但是 per_cpu_rcu_data.nextlist 非空）或者（per_cpu_rcu_data.donelist 非空）或者（per_cpu_rcu_data. quiescbatch 不等于 rcu_ctrlblk.cur）或者（per_cpu_rcu_data. qs_pending 等于 0）或者（per_cpu_bh_rcu_data.curlist 非空并且 rcu_bh_ctrlblk.completed 大于等于 per_cpu_rcu_bh_data.batch）或者（per_cpu_rcu_bh_data.curlist 空但是 per_cpu_rcu_bh_data. nextlist 非空）或者（per_cpu_rcu_bh_data.donelist 非空）或者（per_cpu_rcu_bh_data. quiescbatch 不等于 rcu_bh_ctrlblk.cur）或者（per_cpu_rcu_bh_data. qs_pending 等于 0），执行 rcu_check_callbacks(cpu, user_ticks)。如果 current->thread_info->preempt_count&0xfff0000 不等于 0x10000，将 per_cpu_kstat.cpustat.irq 加上 sys_ticks，将 sys_ticks 设置为 0，如果 current->thread_info->preempt_count&0xff00 不等于 0，将 per_cpu_kstat.cpustat. softirq 加上 sys_ticks，将 sys_ticks 设置为 0。如果 p 等于 rq->idle，且 rq->nr_iowait 大于 0，将 per_cpu_kstat.cpustat.iowait 加上 sys_ticks，否则将 per_cpu_kstat.cpustat.idle 加上 sys_ticks，执行空函数 wake_priority_sleeper(rq)；执行 rebalance_tick(cpu, rq, SCHED_IDLE)（只有配置了 CONFIG_SMP 选项的系统中有实现），返回；如果 p->static_prio-MAX_RT_PRIO(=100)-20 大于 0，将 per_cpu_kstat.cpustat.nice 加上 user_ticks，否则将 per_cpu_kstat.cpustat.user 加上 user_ticks，将 per_cpu_kstat.cpustat.system 加上 sys_ticks，如果 p->array 不等于 rq->active，将 p->thread_info->flags 设置 TIF_NEED_RESCHED 位，返回。将 rq->lock 上锁，如果 p->prio 小于 MAX_RT_PRIO，如果 p->policy 等于 SCHED_RR 并且 p->time_slice 减 1 后等于 0，将 task_timeslice(p) 传给 p->time_slice，将 p->first_time_slice 设置为 0，将 p->thread_info->flags 设置 TIF_NEED_RESCHED 位，将 p 从 rq->active->queue+p->prio 链表中删除，然后再重新加到 rq->active->queue+p->prio 链表末尾。将 rq->lock 解锁，返回。如果 p->time_slice 减 1 后等于 0，将 p 从 rq->active->queue+p->prio 链表中删除，将 p->thread_info->flags 设置 TIF_NEED_RESCHED 位，调用 effective_prio(p)重新计算优先级传给 p->prio，将 task_timeslice(p) 传给 p->time_slice，将 p->first_time_slice 设置为 0，如果 rq->expired_timestamp 等于 0，将 rq->expired_timestamp 设置为 jiffies。如果 TASK_INTERACTIVE(p)等于 0 或者

EXPIRED_STARVING(rq)，将 p 加到 rq->expired->queue+p->prio 链表末尾，如果 p->static_prio 小于 rq->best_expired_prio，将 p->static_prio 传给 rq->best_expired_prio，否则，将 p 加到 rq->active->queue+p->prio 链表末尾。如果上面 p->time_slice 减 1 后不等于 0，如果(TASK_INTERACTIVE(p) && !((task_timeslice(p) -p->time_slice) % TIMESLICE_GRANULARITY (p)) &&(p->time_slice >= TIMESLICE_GRANULARITY(p)) &&(p->array == rq->active))，将 p 从 rq->active->queue+p->prio 链表中删除，将 p->thread_info->flags 的 TIF_NEED_RESCHED 位置 1，调用 effective_prio(p)重新计算优先级传给 p->prio，将 p 加到 rq->active->queue+p->prio 链表末尾。

void rcu_check_callbacks(int cpu, int user)函数执行过程：如果 user 不等于 0 或者（当前处理器的 per_cpu_runqueues.curr 等于 per_cpu_runqueues.idle 且系统正在执行 softirq，即 current->thread_info->preempt_count&0xff00 不等于 0 且 current->thread_info->preempt_count&0xfff0000 小于等于 1<<16），将 per_cpu_rcu_data.qsctr 加 1，将 per_cpu_rcu_bh_data.qsctr 加 1，如果上面条件不成立但系统正在执行 softirq（即 current->thread_info->preempt_count&0xff00 不等于 0），那么只将 per_cpu_rcu_bh_data.qsctr 加 1，执行 tasklet_schedule (&per_cpu(rcu_tasklet, cpu))。

static inline void tasklet_schedule(struct tasklet_struct *t)函数执行过程：查看并设置 t->state 中的 TASKLET_STATE_SCHED 位，如果原来该位为 0，禁止 IRQ 中断，保护现场，将当前处理器的 per_cpu_tasklet_vec.list 传给 t->next，将 t 传给 per_cpu_tasklet_vec.list，即将 t 加入到 per_cpu_tasklet_vec 链表的开头。将 irq_stat[smp_processor_id()].__softirq_pendin 或上 1<<TASKLET_SOFTIRQ。如果 current->thread_info->preempt_count 与上 0xfffff00 等于 0，即系统不是在处理 interrupt，如果当前处理器的 per_cpu_ksoftirqd 非空且 per_cpu_ksoftirqd.state 不等于 TASK_RUNNING，执行 wake_up_process(per_cpu_ksoftirqd)唤醒该进程的软中断看护进程。使能 IRQ 中断，恢复现场。

void fastcall wake_up_new_task(task_t * p, unsigned long clone_flags)函数执行过程：禁止 IRQ 中断，保护现场，获取处理器 p->thread_info->cpu 对应的运行队列结构 per_cpu_runqueues 的地址传给 runqueue_t 结构指针 rq，将 rq->lock 上锁。将 p->thread_info->cpu 传给局部变量 CPU。将当前处理器 id 号 smp_processor_id()传给局部变量 this_cpu。如果 p->state 不等于 TASK_RUNNING，系统崩溃。将 rq-> wunt_cnt 加 1。将 JIFFIES_TO_NS(CURRENT_BONUS (p)*CHILD_PENALTY/100*MAX_SLEEP_AVG/MAX_BONUS) 传给 p->sleep_avg，将 p->interactive_credit 设置为 0。如果 p->prio 大于等于 MAX_RT_PRIO，将 CURRENT_BONUS(p) - MAX_BONUS / 2 传给 bonus，将 p->prio-bonus 传给 prio，如果 prio 小于 MAX_RT_PRIO，将 MAX_RT_PRIO 传给 prio，如果 prio 大于 MAX_PRIO-1，将 MAX_PRIO-1 传给 prio，将 prio 传给 p->prio。如果 cpu 等于 this_cpu，且 clone_flags 没有设置 CLONE_VM 标志，如果当前进程的优先级阵列结构 prio_array_t 指针 current->array 是空的，执行__activate_task(p, rq) 将进程 p 通过 p->run_list 连接到该进程所属处理器 p->thread_info->cpu 对应的 per_cpu_runqueues.

active->queue[p->prio]队列链表末尾。否则将 current->prio 传给 p->prio，将 p 通过 p->run_list 连接到 current->run_list 链表的末尾，将 current->array 传给 p->array，将 p->array->nr_active 加 1，将 rq->nr_running 加 1。将系统当前的堆栈指针 sp 所在的 struct thread_info 结构其实就是 current->thread_info 的 flags 的 TIF_NEED_RESCHED 位设置为 1，前面如果 clone_flags 设置了 CLONE_VM 标志执行__activate_task(p, rq)，将进程 p 通过 p->run_list 连接到该进程所属处理器 p->thread_info->cpu 对应的 per_cpu_runqueues.active->queue[p->prio]队列链表末尾，将 rq 传给 this_rq。前面如果 cpu 不等于 this_cpu，将处理器 this_cpu 的运行队列结构 per_cpu_runqueues 的地址传给 this_rq，将(p->timestamp - this_rq->timestamp_last_tick) + rq->timestamp_last_tick 传给 p->timestamp，执行__activate_task(p, rq) 将进程 p 通过 p->run_list 连接到该进程所属处理器 p->thread_info->cpu 对应的 per_cpu_runqueues.active->queue[p->prio]队列链表末尾，如果 p->prio 小于 rq->curr->prio，执行 resched_task(rq->curr)将 rq->curr->thread_info->flags 的 TIF_NEED_RESCHED 位设置为 1(单处理器系统中)，将 rq-> wunt_moved 加 1，将 rq->lock 解锁，使能中断，恢复现场，禁止 IRQ 中断，保护现场，获取处理器 current->thread_info->cpu 对应的运行队列结构 per_cpu_runqueues 的地址传给 runqueue_t 结构指针 this_rq，将 rq->lock 上锁。将 JIFFIES_TO_NS(CURRENT_BONUS(current) * PARENT_PENALTY / 100 * MAX_SLEEP_AVG / MAX_BONUS)传给 current->sleep_avg，将 this_rq->lock 解锁，使能中断，恢复现场。

static inline void __activate_task(task_t *p, runqueue_t *rq)函数执行过程：如果 p->sched_info.last_queued 等于 0，将 jiffies 传给 p->sched_info.last_queued，将进程 p 通过 p->run_list 连接到 rq->active->queue[p->prio]队列链表末尾，将 rq->active->bitmap 位码表中的第 p->prio 位设置为 1，将 rq->active->nr_active 加 1，将 rq->active 传给 p->array，将 rq->nr_running 加 1。

void ptrace_notify(int exit_code)函数执行过程：如果退出代号 exit_code&0xffff007f 不等于 SIGTRAP，将局部信号信息结构（siginfo_t）info 全部清 0。将 SIGTRAP 传给 info.si_signo，将 exit_code 传给 info.si_code，将 current->pid 传给 info._sifields._kill._pid，将 current->uid 传给 info._sifields._kill._uid。将 current->sighand->siglock 上锁，执行 ptrace_stop(exit_code, &info)函数停止跟踪，将 current->sighand->siglock 解锁。

static void ptrace_stop(int exit_code, siginfo_t *info)函数执行过程：如果 current->ptrace 没有设置 PT_PTRACED 标志，则系统崩溃。如果当前进程的信号描述结构中的组停止计数器 current->signal->group_stop_count 大于 0，将 current->signal->group_stop_count 减 1，将 info 传给 current->last_siginfo，将 exit_code 传给 current->exit_code，将 TASK_TRACED 传给 current->state，将 current->sighand->siglock 解锁，将进程链表读写锁 tasklist_lock 读上锁。执行 do_notify_parent_cldstop(current, current->parent, CLD_TRAPPED)，将进程链表读写锁 tasklist_lock 读解锁，执行 schedule()。将 current->sighand->siglock 上锁，将 current->last_siginfo 设置为 NULL，如果 current->signal->group->group_stop_count 大于 0 或者 current->pending.signal 的值与 current->blocked 中的值不完全相同或者 current->signal->shared_pending.signal 的值与

current->blocked 中的值不完全相同，检查设置 current->thread_info->flags 中的 TIF_SIGPENDING 标志位，否则检查清除 current->thread_info->flags 中的 TIF_SIGPENDING 标志位。

void fastcall __sched wait_for_completion(struct completion *x)函数执行过程：执行 might_sleep()，将完成量结构 x 的等待队列链表结构的自旋锁 x->wait.lock 上锁。如果 x->done 大于 0，将 x->done 减 1，将 x->wait.lock 解锁，返回。如果 x->done 等于 0，将局部 wait_queue_t 结构 wait.flags 或上 WQ_FLAG_EXCLUSIVE，wait.task 等于 current，将 wait 通过 wait. task_list 链接到 x->wait.task_list 链表的末尾。将 TASK_UNINTERRUPTIBLE 传给 current->state 即将 current 设置成处于等待队列（不是运行队列）中等待资源有效时唤醒，不可由其他进程通过信号或者定时中断唤醒状态，将 x->wait.lock 解锁，执行 schedule()重新调度系统中的进程，将 x->wait.lock 上锁，如果 x->done 仍等于 0，重复本段操作。将 wait 从 x->wait.task_list 链表中删除，返回。

asmlinkage void __sched schedule(void)函数执行过程：如果有幸当前进程的退出状态标志字 current->exit_state 既没有设置 EXIT_DEAD 也没有 EXIT_ZOMBIE，即当前进程没有消亡也没有结束，而且不幸当前系统正在进行原子操作，打印“原子操作是进行调度”错误信息，执行 dump_stack()，死机。执行 profile_hit(SCHED_PROFILING, __builtin_return_address(0))将当前指令的计数器加 1。“need_resched 标号”：将 current_thread_info()->preempt_count 加 1 禁止抢占。将 current 传给 prev，执行 release_kernel_lock(prev)。“need_resched_nonpreemptible 标号”：将 runqueue_t *rq 指向当前处理器的 per_cpu_runqueues。如果 current 等于 rq->idle 并且 current->state 不等于 TASK_RUNNING，打印“从空闲进程调度错误”，执行 dump_stack()，死机。将 rq->sched_cnt 计数器加 1，将系统从启动到现在运行的纳秒数传给 now，如果 now 减去 prev->timestamp 小于 NS_MAX_SLEEP_AVG，将 now - prev->timestamp 传给 run_time，否则将 NS_MAX_SLEEP_AVG 传给 run_time。如果 prev->interactive_credit 大于 CREDIT_LIMIT (=100)，如果 CURRENT_BONUS(prev)不等于 0，将 run_time 除以 CURRENT_BONUS(prev)传给 run_time。将 rq->lock 上锁，如果 current->flags 设置了 PF_DEAD 标志，那么将 EXIT_DEAD 传给 current->state。将 long *指针 switch_count 指向 prev->nivcsw，如果 prev->state 不等于 0 且 current_thread_info()->preempt_count 没有设置抢占标志 PREEMPT_ACTIVE（=0x40000000）。将 switch_count 指向 prev->nvcsw，如果 prev->state 设置了 TASK_INTERRUPTIBLE 标志位（即进程 prev 处于等待队列中等待资源有效时被唤醒，也可以由其他进程通过信号或定时中断唤醒进入运行队列）并且将 prev->thread_info->flags 的 TIF_SIGPENDING 位清 0，如果不幸原来该位为 1（表示进程 prev 停靠在某个信号上，等待某个信号来唤醒），那么将 TASK_RUNNING 传给 prev->state，否则如果 prev->state 设置了 TASK_UNINTERRUPTIBLE 标志位（即进程 prev 处于等待队列中等待资源有效时被唤醒，不可由其他进程通过信号或定时中断唤醒进入运行队列），那么将 rq->nr_uninterruptible 加 1，将 rq->nr_running 减 1，将 prev->array->nr_active 减 1，

将 prev 从其所在的运行队列中的活动优先级阵列队列链表 prev->run_list 中删除，如果链表 prev->array->queue+prev->prio 为空，将 prev->array->bitmap 的第 prev->prio 位清 0，将 p->array 设置为 NULL。将当前处理器 smp_processor_id()传给 CPU，如果 rq->nr_running 等于 0，“go_idle 标号”：执行 idle_balance(cpu, rq)（在不支持 CONFIG_SMP 的系统中该函数什么也不做），如果 rq->nr_running 仍等于 0，将运行队列中的空闲进程描树结构指针 rq->idle 传给 next，将运行队列的期满时间戳 rq->expired_timestamp 设置为 0，执行 wake_sleeping_dependent(cpu, rq)（该函数在没有配置 CONFIG_SCHED_SMT 选项的系统中什么也不做，CONFIG_SCHED_SMT 选项只有在支持 CONFIG_SMP 的系统中才可以配置，含义是 SMT (Hyperthreading) scheduler support，即支持超线程调度器。在 Linux2.6.10 中只有 i386、ppc64、x86_64 平台上才可以配置该选项），如果 rq->nr_running 仍等于 0，跳转到“switch_tasks”处切换进程。如果前面 rq->nr_running 不等于 0，执行 dependent_sleeper(cpu, rq)（该函数在没有配置 CONFIG_SCHED_SMT 选项的系统中什么也不做，直接返回 0），如果执行结果不等于 0，将 rq->idle 传给 next 然后跳转到“switch_tasks 标号”处切换进程，否则如果 rq->nr_running 等于 0，跳转到“go_idle 标号”处。将运行队列的活动优先级阵列结构指针 rq->active 传给 array。如果 array->nr_active 等于 0，将运行队列的调度切换计数器 rq->sched_switch 加 1，将运行队列的到期优先级阵列结构指针 rq->expired 传给 rq->active，将 array 传给 rq->expired，再将 rq->active 传给 array，将 rq->expired_timestamp 设置为 0，将 MAX_PRIO（=140）作为运行队列最好的期满优先级 rq->best_expired_prio，如果 array->nr_active 不等于 0，将运行队列的未发生切换的调度计数器 rq-> sched_noswitch 加 1。将优先级阵列位码表 array->bitmap 中第一位为 1 的位号传给 idx，将优先级 idx 对应的进程描述结构链表 array->queue + idx 传给链表指针 queue。将 queue->next 对应的进程描述结构 struct task_struct 指针传给 next，如果 next->prio 大于等于 MAX_RT_PRIO（=100）并且 next->activated 大于 0，将 now-next->timestamp 传给 delta，如果 next->activated 等于 1，将 delta * (ON_RUNQUEUE_WEIGHT * 128 / 100) / 128 传给 delta，将 next->array 传给 array，将 array->nr_active 减 1，将 next 从其所在的 next->run_list 链表中删除，如果 array->queue+p->prio 链表是空的，将 array->bitmap 中的第 p->prio 位清 0，执行 recalc_task_prio(next, next->timestamp + delta) 重新计算进程 next 的优先级，如果 next->sched_info.last_queued 等于 0，将 jiffies 传给 next->sched_info.last_queued，将进程 next 通过 next->run_list 连接到 array->queue[next->prio]队列链表末尾，将 array->bitmap 位码表中的第 next->prio 位设置为 1，将 array->nr_active 加 1，将 array 传给 next->array，将 next->activated 设置为 0。“switch_tasks 标号”：如果 next 等于 rq->idle，将 rq->sched_goidle 加 1。pretch(next)；清除 prev->thread_info->flags 中的 TIF_NEED_RESCHED 位。将 prev 进程所属处理器的 per_cpu_rcu_data.qsctr 加 1，将 prev->sleep_avg 减去 run_time，如果(long)prev->sleep_avg 小于等于 0，将 prev->sleep_avg 设置成 0，如果 prev->interactive_credit 小于等于 CREDIT_LIMIT 或大于等于-CREDIT_LIMIT 将 prev->interactive_credit 减 1，将 now 传给 prev->timestamp 和 prev->last_ran。执行 sched_info_switch(prev, next)通知切换进程，设置 prev 和 next 以及各自所

在运行队列中的调度信息结构中的变量。如果 prev 等于 next，将 rq->lock 解锁，否则如果 prev 不等于 next，将 now 传给 next->timestamp，将 next 传给 rq->curr，将局部切换计数器*switch_count 加 1，执行 prepare_arch_switch(rq, next)准备进行切换（该函数在单处理器中只是势能 IRQ 中断，在多处理器系统中也只是将 next->switch_lock 和 rq->lock 上锁）。执行 context_switch(rq, prev, next)并把返回指针传给 prev，执行 barrier()使系统所有 cache 和 Write Buffer 都无效，重新将内存中的数据读到 cache 中，执行 finish_task_switch(prev)，将 current 传给 prev。如果 reacquire_kernel_lock(prev)小于 0，跳转到“need_resched_nonpreemptible 标号”处。使系统的 cache 都无效，将 current->thread_info->preempt_count 减 1，如果 current->thread_info->flags 设置了 TIF_NEED_RESCHED 标志，跳转到“need_resched 标号”处。

static void recalc_task_prio(task_t *p, unsigned long long now)函数执行过程：将 now - p->timestamp 传给 unsigned long long __sleep_time，如果__sleep_time 大于 NS_MAX_SLEEP_AVG，将 unsigned long sleep_time 设置为 NS_MAX_SLEEP_AVG，否则将(unsigned long)__sleep_time 传给 sleep_time。如果 sleep_time 大于 0 即进程 p 的睡眠时间大于 0，如果 p->mm 指针不空且 p->activated 不等于-1 且 sleep_time 大于 INTERACTIVE_SLEEP(p)，将 JIFFIES_TO_NS(MAX_SLEEP_AVG - DEF_TIMESLICE)传给 p->sleep_avg，如果 p->interactive_credit 小于等于 CREDIT_LIMIT，将 p->interactive_credit 加 1。如果前面 3 个判断条件不全成立，且 MAX_BONUS - CURRENT_BONUS(p)不等于 0，将 sleep_time*(MAX_BONUS - CURRENT_BONUS(p))传给 sleep_time，如果 p->interactive_credit 小于（-CREDIT_LIMIT）且 sleep_time 大于 JIFFIES_TO_NS(task_timeslice(p)，将 JIFFIES_TO_NS(task_timeslice(p))传给 sleep_time，如果 p->activated 等于-1，且 p->interactive_credit 小于等于 CREDIT_LIMIT 且 p->mm 指针非空，如果 p->sleep_avg 大于 INTERACTIVE_SLEEP(p)，将 sleep_time 设置为 0，否则如果 p->sleep_avg + sleep_time 大于等于 INTERACTIVE_SLEEP(p)，将 INTERACTIVE_SLEEP(p)传给 p->sleep_avg，将 sleep_time 设置为 0，将 sleep_time 加入到 p->sleep_avg 中，如果 p->sleep_avg 大于 NS_MAX_SLEEP_AVG，将 NS_MAX_SLEEP_AVG 传给 p->sleep_avg，如果 p->interactive_credit 小于等于 CREDIT_LIMIT，将 p->interactive_credit 加 1。如果 p->static_prio - CURRENT_BONUS(p) + MAX_BONUS / 2 小于 MAX_RT_PRIO，将 MAX_RT_PRIO 传给 p->prio。如果 p->static_prio - CURRENT_BONUS(p) + MAX_BONUS / 2 大于 MAX_PRIO-1，将 MAX_PRIO-1 传给 p->prio。

static inline void sched_info_switch(task_t *prev, task_t *next)函数执行过程：将 prev->thread_info->cpu 对应的运行队列结构 per_cpu_runqueues 的地址传给 struct runqueue *rq；如果 prev 不等于 rq->idle，将 prev->sched_info.cpu_time 加上 jiffies-prev->sched_info.last_arrival 累计进程 prev 所占其所属处理器的运行时间，rq->rq_sched_info.cpu_time 加上 jiffies-prev->sched_info.last_arrival 累计 prev 所在运行队列中的所有进程占用其所属处理器的运行时间。如果 next 不等于 rq->idle，将 next->thread_info->cpu 对应的运行队列结构 per_cpu_

runqueues 的地址传给 rq。如果 next->sched_info.last_queued 不等于 0，将 jiffies - next->sched_info.last_queued 传给 diff。将 next->sched_info.last_queued 设置为 0，将 diff 传给 next->sched_info.run_delay，将 jiffies 传给 next->sched_info.last_arrival，将 next->sched_info.pcnt 加 1。如果 rq 空，则返回。将 rq->rq_sched_info.run_delay 加上 diff，将 rq->rq_sched_info.pcnt 加 1。

static inline task_t * context_switch(runqueue_t *rq, task_t *prev, task_t *next)函数执行过程：将 next->mm 传给 struct mm_struct *mm；将 prev->active_mm 传给 struct mm_struct *oldmm。如果 mm 指针为空，将 oldmm 传给 next->active_mm，将 oldmm->mm_count 加 1，执行空函数 enter_lazy_tlb(oldmm, next)，如果 mm 指针非空，执行 switch_mm(oldmm, mm, next)。如果 prev->mm 指针空，将 prev->active_mm 设置成 NULL，如果 rq->prev_mm 指针非空，系统告警，将 oldmm 传给 rq->prev_mm。执行 switch_to(prev, next, prev)，返回 prev。

static inline void switch_mm(struct mm_struct *prev, struct mm_struct *next, struct task_struct *tsk)函数执行过程：将当前处理器 id 号 smp_processor_id()传给 cpu。如果 prev 不等于 next，将 next->cpu_vm_mask 的第 cpu 位设置为 1，执行 check_context(next)（该函数在 ARMv6 以下处理器中是空函数），执行 cpu_switch_mm(next->pgd, next)（cpu_switch_mm(pgd,mm)函数被定义成 cpu_do_switch_mm(virt_to_phys(pgd),mm)，而 cpu_do_switch_mm()对于 xscale 处理器如 ixp425 来说就是 cpu_xscale_switch_mm()汇编函数），将 virt_to_phys(next->pgd)存入到 CP15 的 C2 寄存器中，将 next->cpu_vm_mask 的第 cpu 位清 0。

cpu_xscale_switch_mm(virt_to_phys(pgd),mm)函数执行过程如下：先执行 clean_d_cache()，使整个指令 cache 无效。清空 Write Buffer，将 r0 即 virt_to_phys(pgd)写入 CP15 的 C2 寄存器中，使整个统一 TLBs 或指令和数据 TLBs 无效。

switch_to(prev, next, last)函数执行过程：直接调用汇编函数__switch_to(prev,prev->thread_info,next->thread_info)并将返回指传给 last；truct task_struct *__switch_to(struct task_struct *prev, struct thread_info *prev->thread_info, struct thread_info *next->thread_info)执行过程如下：将 next->thread_info->cpu_domain 的值（域控制寄存器的值）装入 r3，将系统当前寄存器 r4~r10(sl)、fp、sp、lr 的值存入 prev->thread_info->cpu_context 中，将 r3 的值写入 CP15 的域控制寄存器 C3 中。将保存在 next->thread_info->cpu_context 中的 r4~r10(sl)、fp、sp、lr 寄存器的值分别装入系统当前的 r4~r10(sl)、fp、sp、pc 寄存器中，这样就进入到了 next 所描述的进程，开始执行 pc 处的指令，如果 next 进程已经运行过，那么 pc 指向 next 进程上次退出时最后执行的指令的下一条指令，如果 next 进程刚创建是第一次运行，那么 pc 指向 ret_from_fork()函数。此时 r0 指向 prev，即传递给被调用函数的参数或者返回的值就是 prev 指针。

asmlinkage void ret_from_fork(void)函数执行过程：执行 finish_task_switch(prev)。如果 current->set_child_tid 指针不空，将 current->pid 复制到用户空间的 current->set_child_tid，将当前进程的线程信息结构指针 current->thread_info 传给 tsk，将 tsk->flags 传给 r1，将 1 传给 why。

如果 r1 中没有设置_TIF_SYSCALL_TRACE 位，执行 ret_slow_syscall()。将 sp 传给 r1，将 1 传给 r0，执行 syscall_trace()，执行 ret_slow_syscall()。

static void finish_task_switch(task_t *prev)函数执行过程：将当前处理器的运行队列结构 per_cpu_runqueues 的地址传给 runqueue_t *rq。将 rq->prev_mm 传给 struct mm_struct *mm，将 rq->prev_mm 设置为 NULL，将 prev->flags 传给 prev_task_flags。执行 finish_arch_switch(rq, prev)（该函数在 smp 系统中将 prev->switch_lock 解锁，在 up 系统中将 prev->lock 解锁）。如果 mm 指针非空，执行 mmdrop(mm)处理掉 struct mm_struct 结构 mm。如果 prev_task_flags 设置了 PF_DEAD 标志，执行 put_task_struct(prev)处理掉 struct task_struct 结构 prev。

汇编函数 ret_slow_syscall()执行过程：禁止 IRQ 中断，使系统进入 supervisor 模式，将 tsk->flags 传给 r1，如果 r1 设置了_TIF_WORK_MASK 位，跳转到 work_pending()，执行 slow_restore_user_regs ()。

汇编函数 work_pending()执行过程：如果 r1 设置了_TIF_NEED_RESCHED 位，调用 schedule()。如果 r1 既没有设置_TIF_NOTIFY_RESUME，也没有设置_TIF_SIGPENDING，执行 slow_restore_user_regs ()。将 sp 传给 r0，将 why 传给 r2。调用 do_notify_resume()，禁止 IRQ 中断，进入 supervisor 模式，执行 slow_restore_user_regs ()。

汇编函数 slow_restore_user_regs ()执行过程：将堆栈指针 sp 所指的系统各寄存器空间的 cpsr 的值传给 r1，将 pc 的值传给 lr，将 r1 保存到 spsr_cxsf 即 spsr_svc 中，把 sp 处的寄存器值分别装入 r0~lr 中，将 r0 传给 r0，将 sp 加上 S_FRAME_SIZE - S_PC，将 lr 传给 pc。

asmlinkage void do_notify_resume(struct pt_regs *regs, unsigned int thread_flags, int syscall)函数执行过程：如果 thread_flags 中设置了_TIF_SIGPENDING 标志，即当前进程有挂起信号，那么调用 do_signal(&current->blocked, regs, syscall)处理当前进程的挂起信号。

void cpu_idle(void)函数执行过程：使能 FIQ 中断，然后进入 while(1)死循环。将函数指针 idle()指向 pm_idle()函数（该函数未定义）。如果 idle 指针空，将 idle 指向 default_idle()函数。将 current->thread_info->preempt_count 加 1，执行空函数 leds_event(led_idle_start)（这个函数有硬件平台决定，如果板子有系统空闲指示灯可以实现这个函数）。如果 current->thread-info->flags 没有设置 TIF_NEED_RESCHED 位，即不需要重新调度，执行调用 idle()函数。执行空函数 leds_event(led_idle_end)，将 current->thread_info->preempt_count 减 1，执行 schedule()。

void default_idle(void)函数执行过程：禁止 IRQ 中断。如果 curretn->thread-info->flags 没有设置 TIF_NEED_RESCHED 位，即不需要重新调度，而且 hlt_counter 等于 0，执行 arch_idle()函数（该函数由不同的处理器型号具体定义，有空闲模式的处理器一般是操作处理器的某个或某些寄存器中的某个或某些位，使处理器进入空闲模式，没有空闲模式的处理器一般什么都不做），使能 IRQ 中断。

17.6 init()进程执行过程

init()进程是内核创建的第一个内核进程，它的 pid 为 1，即 1 号进程。init()进程首先继续完成系统初始化，然后挂接根文件系统并执行根文件系统中的 init 应用程序，从而使 init()由内核进程成为普通进程，最后执行用户 shell 应用程序进入与用户交互的 shell 界面等待用户的输入。

init()进程的具体执行过程如下：

调用 lock_kernel()函数，将当前进程的描述结构指针 current 传给 child_reaper，child_reaper 在声明时被设置为指向 0 号进程描述结构 init_task，在这里又该指向 current 即 init()进程（1 号进程）的描述结构。

17.6.1 smp_prepare_cpus(max_cpus)函数

该函数在不支持 CONFIG_SMP 的系统中是空函数，在 i386/PowerPC/MIPS 架构处理器系统中有自己具体定义，ARM 处理器中没有自己定义。

17.6.2 do_pre_smp_initcalls()函数

该函数在不支持多处理器即没有配置 CONFIG_SMP 选项的系统中直接调用 spawn_ksoftirqd()创建当前处理器的 ksoftirqd()看护进程，然后返回。此时具体执行过程如下：

__init int spawn_ksoftirqd(void)函数执行过程：将(void*) smp_processor_id()传给(void*)cpu。执行 cpu_callback(&cpu_nfb, CPU_UP_PREPARE, cpu)；执行 cpu_callback(&cpu_nfb, CPU_ONLINE, cpu)；执行 register_cpu_notifier(&cpu_nfb)将 cpu_nfb 按从高到低的优先级顺序注册到 CPU 通知者结构变量链表 cpu_chain 中，返回 0。

static int __devinit cpu_callback(struct notifier_block *nfb, unsigned long action, void *hcpu)函数执行过程：将 hcpu 传给 hotcpu，根据 action 的值进行不同处理。“如果 action 等于 CPU_UP_PREPARE 即准备启动处理器 hotcpu”：如果 hotcpu 的小任务描述结构链表 per_cpu_tasklet_vec.lists 指针非空，则系统崩溃；如果 hotcpu 的高级小任务描述结构链表 per_cpu_tasklet_hi_vec.lists 指针非空，则系统崩溃。执行 kthread_create(ksoftirqd, hcpu, "ksoftirqd/%d", hotcpu)创建名叫“ksoftirqd/hotcpu”的进程执行函数为 static int ksoftirqd(void * __bind_cpu)的看护进程并把进程描述结构指针传给 p。如果 p 地址错误，打印错误信息，返回 NOTIFY_BAD。执行 kthread_bind(p, hotcpu)将 ksoftirqd 进程与 hotcpu 绑定。将 p 传给 hotcpu 的 per_cpu_ksoftirqd。“如果 action 等于 CPU_ONLINE 即处理器 hotcpu 已经启动”：执行 wake_up_process(per_cpu(ksoftirqd, hotcpu))唤醒 hotcpu 的软中断看护进程 per_cpu_ksoftirqd。

“如果 action 等于 CPU_UP_CANCELED 即要取消启动处理器 hotcpu”：执行 kthread_bind(per_cpu(ksoftirqd, hotcpu), smp_processor_id())将 hotcpu 的软中断看护进程 per_cpu_ksoftirqd 与当前处理器 smp_processor_id()绑定。“如果 action 等于 CPU_DEAD 即处理器 hotcpu 死机”：将 hotcpu 的软中断看护进程 per_cpu_ksoftirqd 传给 struct task_struct *p。将 hotcpu 的软中断看护进程 per_cpu_ksoftirqd 设置为 NULL，执行 kthread_stop(p)，执行 takeover_tasklets(hotcpu)，返回 NOTIFY_OK。

struct task_struct *kthread_create(int (*threadfn)(void *data), void *data, const char namefmt[],...)函数执行过程：声明 struct kthread_create_info create，声明并初始化 struct work_struct work：work.entry 链表为空，将 work.func 指向 static void keventd_create_kthread(void *_create)，将 work.data 指向 create，work.timer.magic 等于 TIMER_MAGIC（=0x4b87ad6e）。将 threadfn 传给 create.threadfn，将 data 传给 create.data。初始化完成量结构 create.started 和 create.done。如果内核事件看护进程工作队列指针 keventd_wq 为空，执行 work.func(work.data)即 keventd_create_kthread(work.data)，否则，执行 schedule_work(&work)和 wait_for_completion (&create.done)。如果 create.result 指针非空，将 namefmt 按格式（比如“ksoftirqd/n(n 是 cpu 号)”）传给 create.result->comm，返回 create.result。

static void keventd_create_kthread(void *_create)函数执行过程：将_create 传给 struct kthread_create_info *create，执行 kernel_thread(kthread, create, CLONE_FS | CLONE_FILES | SIGCHLD)创建内核进程 kthread()，将其 id 号传给 pid。如果 pid 小于 0，将 ERR_PTR(pid)传给 create->result，否则，执行 wait_for_completion(&create->started)，将 pid 对应的进程描述结构指针传给 create->result，执行 complete(&create->done)，禁止 IRQ 中断，保护现场，将 create->done.wait.lock 上锁，将 create->done.done 加 1，执行__wake_up_common (&create->done.wait, TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE, 1, 0, NULL)，使能 IRQ 中断，恢复现场，将 create->done.wait.lock 解锁。

int fastcall schedule_work(struct work_struct *work)执行过程：直接调用返回 queue_work (keventd_wq, work)；int fastcall queue_work(struct workqueue_struct *wq, struct work_struct *work)函数执行过程：将当前处理器 ID 号 get_cpu()传给 cpu，将 ret 设置为 0。查看并设置 work->pending 的第 0 位为 1，如果该位原来为 0，如果工作队列中的链表 wq->list 是空的，将 cpu 设置为 0，如果工作描述结构中的链表 work->entry 非空，系统崩溃。将 wq->cpu_wq + cpu 传给 struct cpu_workqueue_struct *cwq，禁止 IRQ 中断，保护现场，将 cwq->lock 上锁。将 cwq 传给 work->wq_data，将 work 通过其 entry 连接到处理器工作队列链表 cwq->worklist 末尾。将 cwq->insert_sequence 加 1；执行__wake_up(&cwq->more_work, TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE, 1, NULL)。将 cwq->lock 解锁，使能 IRQ 中断，恢复现场，将 ret 设置为 1，返回 ret。

void kthread_bind(struct task_struct *k, unsigned int cpu)函数执行过程：如果 k->state 不等于 TASK_INTERRUPTIBLE，系统崩溃。执行 wait_task_inactive(k)函数（该函数在不支持 CONFIG_SMP 的系统中是空函数）。将 cpu 传给 k->thread_info->cpu，将 k->cpus_allowed.bits[0] 设置为 1<<cpu。

int kthread_stop(struct task_struct *k)函数执行过程：将互斥信号量 kthread_stop_lock 减少，将 k->usage 加 1。初始化线程停止信息结构中的完成量结构 kthread_stop_info.done，使系统全部 cache 无效，将 k 传给 kthread_stop_info.k。执行 wake_up_process(k)唤醒进程 k，将 k->usage 减 1，如果 k->usage 原来就大于 0，执行__put_task_struct(k)。执行 wait_for_completion(&kthread_stop_info.done)，将 kthread_stop_info.k 设置为 NULL，将 kthread_stop_info.err 传给 ret，将互斥信号量 kthread_stop_lock 增加，返回 ret。

static void takeover_tasklets(unsigned int cpu)函数执行过程：禁止 IRQ 中断，将 cpu 的 per_cpu_tasklet_vec 连接到当前处理器的 per_cpu_tasklet_vec.list 链表的末尾，将 irq_stat[smp_processor_id()].__softirq_pending 或上 1<<TASKLET_SOFTIRQ。如果 current->thread_info->preempt_count 与上 0xfffff00 等于 0，即系统不是在处理 interrupt，如果当前处理器的 per_cpu_ksoftirqd 非空且 per_cpu_ksoftirqd.state 不等于 TASK_RUNNING，执行 wake_up_process(per_cpu_ksoftirqd)唤醒该进程的软中断看护进程。将 cpu 的 per_cpu_tasklet_hi_vec 连接到当前处理器的 per_cpu_tasklet_hi_vec.list 链表的末尾，将 irq_stat[smp_processor_id()].__softirq_pending 或上 1<<HI_SOFTIRQ。如果 current->thread_info->preempt_count 与上 0xfffff00 等于 0，即系统不是在处理 interrupt，如果当前处理器的 per_cpu_ksoftirqd 非空且 per_cpu_ksoftirqd.state 不等于 TASK_RUNNING，执行 wake_up_process(per_cpu_ksoftirqd)唤醒该进程的软中断看护进程，使能 IRQ 中断。

17.6.3 fixup_cpu_present_map()函数

该函数在没有配置 CONFIG_SMP 选项即不支持多处理器的系统中什么都不做。

在配置了 CONFIG_SMP 选项即支持多处理器的系统中，该函数检查系统中当前存在的处理器位码表变量 cpu_present_map 中是否所有的位都是 0，如果是，那么将处理器位码表变量 cpu_possible_map 中所有为 1 的位对应的 cpu_present_map 中的位也设置为 1。

17.6.4 smp_init()函数

显然，该函数执行多处理器相关初始化，在没有配置 CONFIG_SMP 选项即不支持多处理器的系统中是空函数，什么都不做。

在配置了 CONFIG_SMP 选项即支持多处理器的系统中，该函数检查系统中当前存在的处理器位码表变量 cpu_present_map（系统中实际存在的处理器位码）中每个等于 1 的位对应于当

前在线的处理器位码表变量 cpu_online_map（系统中在线处理器即已经启动工作的处理器位码）中的位是否也是 1，如果不等于 1，那么调用 cpu_up(i)启动该位号 i 对应的处理器即 i 号处理器。在调用 cpu_up(i)之前，先检查 cpu_online_map 中等于 1 的位数是否大于系统中允许的最大处理器数 max_cpus，即如果系统中已经启动运行的处理器数已经达到系统支持的最大处理器数，则不再启动新的处理器，退出 smp_init()函数。

17.6.5 sched_init_smp()函数

该函数执行多处理器相关初始化，在没有配置 CONFIG_SMP 选项即不支持多处理器的系统中是空函数，什么都不做。

在配置了 CONFIG_SMP 选项即支持多处理器的系统中，该函数初始化系统的调度域（schedule domains），然后从高到低的优先级顺序将通知块结构变量 update_sched_domains_nb 注册到 CPU 通知者结构变量链表 cpu_chain 中。

17.6.6 populate_rootfs()函数

该函数检查并解压缩 initramfs 以及 initrd。详细执行过程说明如下：执行 unpack_to_rootfs(&__initramfs_start,&__initramfs_end-&__initramfs_start,0)将&__initramfs_start 处的 initramfs 包解析出来作为 rootfs，并将返回值传给 char *err。如果 err 指针不空，也就是解析 initramfs 包出错或者 initramfs 包不是 cpio 格式，系统 panic；如果系统没有配置 CONFIG_BLK_DEV_INITRD 即系统没有配置 initrd(init ramdisk)设备或者 initrd_start 即__phys_to_virt(phys_initrd_start)等于 0，也就是说如果 initrd 设备不存在，则返回。打印"checking if image is initramfs..."。执行 unpack_to_rootfs((char *)initrd_start, initrd_end - initrd_start, 1) 检查 initrd_start 处即 initrd 设备中的包是不是 cpio 格式，并将返回值传给 err。如果 err 指针空即 initrd 设备中的包是 cpio 格式的，打印" it is\n"。执行 unpack_to_rootfs((char *)initrd_start, initrd_end - initrd_start, 0)解析 initrd 设备中的 initramfs 包作为 rootfs，执行 free_initrd_mem(initrd_start, initrd_end)，如果命令行中没有设置 keepinitrd 命令，即系统参数全局变量 keepinitrd 等于 0（表示不保留 initrd 所占内存空间），那么按页释放从 initrd_start 到 initrd_end 之间 initrd 设备所占的全部内存空间，返回，相反如果 err 指针非空，打印“不是，好像是一个 initrd”，执行 sys_open("/initrd.image", O_WRONLY|O_CREAT, 700) 按只写方式创建并打开/initrd.image 文件并将返回的文件句柄传给 fd，如果 fd 大于等于 0，执行 sys_write(fd, (char *)initrd_start, initrd_end - initrd_start)将 initrd_start 到 initrd_end 之间的数据写入到/initrd.image 文件中，执行 sys_close(fd)关闭/initrd.image 文件，执行 free_initrd_mem(initrd_start, initrd_end)，如果命令行中没有设置 keepinitrd 命令，即变量 keepinitrd 等于 0，按页释放从 initrd_start 到 initrd_end 之间 initrd 设备所占的全部内存空间，返回。

char * __init unpack_to_rootfs(char *buf, unsigned len, int check_only)函数执行过程：首先申请 110 字节的内核空间，将其地址传给静态字符指针 header_buf。再申请 PATH_MAX （=4096）(链接目标文件名的最大长度)+ N_ALIGN(PATH_MAX)（链接符号名的最大长度） + 1（结束字符）大小的内核空间，将其地址传给静态字符指针 symlink_buf。再申请 N_ALIGN(PATH_MAX)大小的内核空间，将其地址传给静态字符指针 name_buf。再申请 WSIZE（=0x8000=32KB）大小的内核空间，将其地址传给静态无符号字符指针 Window。只要上面 4 次调用 malloc()函数申请内核空间又一次申请失败，打印“不能分配缓冲区”，系统 panic。然后开始解析 buf 地址处的 cpio 包，解析过程为：首先过滤掉 buf 位置处全部为 0 的字节，然后查看第一个非 0 字节，如果该字节等于'0'即 0x30，且该节的地址相对于起始 buf 地址的偏移量是 4 字节对齐的，则认为是 cpio 包，调用 write_buffer(buf, len)来解析 buf 处的 cpio 包，这里的 buf 和 len 是更新之后的 buf 和 len，buf 递增而 len 递减。如果 buf 处是正常的 cpio 包，那么 write_buffer(buf，len)会调用 8 个状态（Start、Collect、GotHeader、SkipIt、GotName、CopyFile、GotSymlink、Reset）对应的函数（do_start、do_collect、do_header、do_skip、do_name、do_copy、do_symlink、do_reset）中的部分函数来解析处理 cpio 包中的每个文件/目录/设备节点/符号链接等，调用的顺序为：do_start()->read_into(header_buf, 110, GotHeader)（读出 cpio 包最前面的 110 字节的包头）-> do_header()，检查包头的最开始 6 个字符是不是“070701”，如果不是则不是 cpio 包格式，将“没有 cpio 标示字”错误信息传给 message，返回 1；如果是，则将 cpio 包中第一个文件/目录/设备节点/符号链接的参数分别解析到对应的静态全局变量中，然后将 next_header 指向下一个文件/目录/设备节点/符号链接的开始位置，4 字节对齐。然后检查 check_only，如果 check_only 是 1 也就述说只检查包是不是 cpio 包而不解析处理，那么依次反复调用 do_name()->do_skip()->do_start()->do_header()->…检查包中的每段是否都是 cpio 段格式，如果是，最后调用 do_reset；如果 check_only 是 0，则查看名字长度 name_len 是否合法；如果不合法，返回 0，否则检查模式字 mode，如果 mode 指示该段描述的是一个符号链接，调用 do_collect()将“070701”后 N_ALIGN(name_len) + body_len 个字节的数据复制到 symlink_buf 中，再将 symlink_buf[N_ALIGN(name_len) + body_len]设置为'\0'，然后依次执行 sys_symlink(symlink_buf + N_ALIGN(name_len), collected)和 sys_lchown(symlink_buf, uid, gid)。如果 mode 指示该段描述的是一个文件或者 body_len 等于 0（描述的是目录或设备节点），那么调用 do_name()分析“070701”后 N_ALIGN(name_len)个字节的名字及模式字 mode。do_name()首先判断名字是不是等于"TRAILER!!!"，如果是表示到了 cpio 包的末尾段，不处理下一步执行 do_reset()，结束 cpio 包的解析处理，返回 0，如果不是 cpio 包的末尾段，但 check_only 等于 1，直接返回 0，否则判断模式字 mode，如果 mode 指示该段描述的是个文件，那么执行 maybe_link()（maybe_link(void)判断 nlink 是否大于等于 2，如果不是直接返回 0，否则执行 find_link(major, minor, ino, collected)并将返回值传给 char *old，如果 old 指针非空，返回((sys_link(old, collected) < 0) ? -1 : 1)，如果执行结果大于等于 0，则执行 sys_open(collected, O_WRONLY|O_CREAT, mode)将返回值传给文件句柄 wfd，如果 wfd 大于等于 0，则依次执行 sys_fchown(wfd, uid, gid)、sys_fchmod(wfd, mode)、

sys_write(wfd, victim, body_len)、sys_close(wfd)，如果 mode 指示该段描述的是一个目录，那么依次执行 sys_mkdir(collected, mode)、sys_chown(collected, uid, gid)、sys_chmod(collected, mode)，如果 mode 指示该段描述的是一个块设备或者是字符设备或者是 FIFO 或者是 socket，且如果 maybe_link()等于 0，那么依次执行 sys_mknod(collected, mode, rdev)、sys_chown(collected, uid, gid)、sys_chmod(collected, mode)。然后再重复上面操作检查处理下一段。如果第一个非 0 字节不等于'0'或者第一个非 0 字节相对于起始 buf 地址的偏移量不是 4 字节对齐的，那么，会认为是一个 gzip 压缩包，然后就调用 gunzip()函数解压缩这个 gzip 包，如果确实是 gzip 压缩包（前两个字节是“0x1F8B”或者“0x1F9E”），而且解压缩正常，那么会在解压缩的过程中判断解压缩之后的文件是不是 cpio 包，如果是那么和上面一样调用 write_buffer()来解析处理解压缩之后的 cpio 包，如果不是 cpio 包则将“压缩包中是垃圾”错误信息传给 messag。最后释放掉前面申请的 window、name_buf、symlink_buf、header_buf 内核空间，返回 message。

asmlinkage long sys_symlink(const char __user * oldname, const char __user * newname)函数执行过程：调用 getname(oldname)从名字高速缓存 names_cachep 中申请一个名字空间，空间大小为最大路径字符串字符数 PATH_MAX，并把用户空间的名字 oldname 复制到这个高速缓存名字空间，char *from 指向这个高速缓存名字空间。如果 from 指针有错，返回 PTR_ERR(from)。调用 getname(newname)从名字高速缓存 names_cachep 中申请一个名字空间，空间大小为最大路径字符串字符数 PATH_MAX，并把用户空间的名字 newname 复制到这个高速缓存名字空间，char *to 指向这个高速缓存名字空间。如果 to 指针无误，执行 path_lookup(to, LOOKUP_PARENT, &（struct nameidata nd）)，将返回值传给 error，如果 error 不等于 0，依次执行 putname(to)、putname(from)，返回 error。如果 error 等于 0，执行 lookup_create(&nd, 0)，将返回值传给 struct dentry *dentry，如果 dentry 指针无误，执行 vfs_symlink(nd.dentry->d_inode, dentry, from, S_IALLUGO)，将返回值传给 error，执行 dput(dentry)，执行 path_release(&nd)。执行 putname(to)，执行 putname(from)，返回 error。

int fastcall path_lookup(const char *name, unsigned int flags, struct nameidata *nd)函数执行过程：将 LAST_ROOT（等于 1）传给 nd->last_type，将 flags 传给 nd->flags，将 0 传给 nd->depth。如果 name 处的第一个字符是'/'，即要从根目录下查找，那么如果 current->fs->altroot 指针非空且 nd->flags 中没有设置 LOOKUP_NOALT 标志位，那么将 current->fs->altrootmnt->mnt_count 加 1 后将 current->fs->altrootmnt 传给 nd->mnt，将 current->fs->altroot->d_count 加 1 后将 current->fs->altroot 传给 nd->dentry，执行(__emul_lookup_dentry(name,nd)，如果执行结果为 0，返回 0。如果 current->fs->altroot 指针空或者 nd->flags 中设置了 LOOKUP_NOALT 标志位，那么将 current->fs->rootmnt->mnt_count 加 1 后将 current->fs->rootmnt 传给 nd->mnt，将 current->fs->root->d_count 加 1 后将 current->fs->root 传给 nd->dentry。如果 name 处第一个字符不是'/'，那么将 current->fs->pwdmnt->mnt_count 加 1 后将 current->fs->pwdmnt 传给 nd->mnt，将 current->fs->pwd->d_count 加 1 后将 current->fs->pwd 传给 nd->dentry，将 current->total_link_

count 设置为 0，执行 link_path_walk(name, nd)并将返回值传给 retval。如果 current->audit_context 及 nd && nd->dentry 及 nd->dentry->d_inode 指针都非空，那么执行 audit_inode(name, nd->dentry->d_inode->i_ino, nd->dentry->d_inode->i_rdev)，返回 retval。

static int __emul_lookup_dentry(const char *name, struct nameidata *nd)函数执行过程：执行 path_walk(name, nd)，如果执行结果不为 0 即执行出错，返回 0。如果 nd->dentry->d_inode 指针空或者 nd->dentry->d_inode->i_mode 指示 nd->dentry->d_inode 对应的是一个目录，将 nd->dentry 传给 struct dentry *old_dentry，将 nd->mnt 传给 struct vfsmount *old_mnt，将 nd->last 传给 struct qstr last，将 nd->last_type 传给 int last_type，将 LAST_ROOT 传给 nd->last_type，将 current->fs->rootmnt->mnt_count 加 1，然后将 current->fs->rootmnt 传给 nd->mnt。将 current->fs->root->d_count 加 1，将 current->fs->root 传给 nd->dentry。执行 path_walk(name, nd)，如果返回 0，且 nd->dentry->d_inode 指针非空，依次执行 dput(old_dentry)、mntput(old_mnt)，返回 1。如果 nd->dentry->d_inode 指针空，执行 path_release(nd)（依次执行 dput(nd->dentry)、mntput(nd->mnt)）。将 old_dentry 传给 nd->dentry，将 old_mnt 传给 nd->mnt，将 last 传给 nd->last，将 last_type 传给 nd->last_type，返回 1。

int fastcall path_walk(const char * name, struct nameidata *nd) 执行过程：将 current->total_link_count 设置为 0，执行并返回 link_path_walk(name, nd)。

int fastcall link_path_walk(const char * name, struct nameidata *nd)执行过程：将 nd->flags 传给 unsigned int lookup_flags，过滤掉 name 处的全部连续的'/'字符。如果'/'字符之后第一个字符就是结束字符，跳转到“return_reval 标号”处。将 nd->dentry->d_inode 传给 struct inode *inode；如果 nd->depth 不等于 0，将 LOOKUP_FOLLOW 传给 lookup_flags。进入 for(;;)死循环：执行 exec_permission_lite(inode, nd)检查当前进程能否执行 inode 并将返回值传给 err。如果 err 等于 -EAGAIN，执行 permission(inode, MAY_EXEC, nd)并将返回值传给 err。如果 err 不等于 0 即当前进程没有执行 inode 的权限，跳出死循环。将 name 传给 struct qstr this.name，将*(const unsigned char *)name 传给 unsigned int c，将 unsigned long hash 设置为 0，将从 name 位置开始结束字符或'/'字符之前的所有字符值 c 左移 4 位加上 c 右移 4 位再加上 hash，将它们的和乘以 11 再传给 hash。将 name 字符串长度传给 this.len，将(unsigned int) hash 传给 this.hash。如果 name 后就是结束字符，跳转到“last_component 标号”处；如果 name 后是'/'，然后是结束字符，跳转到“last_with_slashes 标号”处。如果 this.name[0]等于'.'，且 this.len 等于 2 且 this.name[1]等于'.'，执行 follow_dotdot(&nd->mnt, &nd->dentry)，将 nd->dentry->d_inode 传给 inode，如果 this.len 等于 1，继续查看下一级目录或文件。如果 nd->dentry->d_op->d_hash 不空，执行 nd->dentry->d_op->d_hash(nd->dentry, &this)将返回值传给 err，如果 err 小于 0 即执行出错，跳出死循环。将 nd->flags 或上 LOOKUP_CONTINUE 标志，执行 do_lookup(nd, &this, &（struct path）next)，将返回值传给 err。如果 err 不等于 0，跳出死循环。执行 follow_mount(&next.mnt, &next.dentry)，将-ENOENT 传给 err，将 next.dentry->d_inode 传给 inode。如果 inode 指针空，跳转到“out_put

标号”处。将-ENOTDIR 传给 err，如果 inode->i_op 指针空，跳转到“out_put 标号”处。

“last_with_slashes 标号位置”：将 lookup_flags 或上 LOOKUP_FOLLOW | LOOKUP_DIRECTORY。“last_component 标号位置”：清除 nd->flags 中的 LOOKUP_CONTINUE 标志。如果 lookup_flags 设置了 LOOKUP_PARENT 标志（创建目录和创建设备节点时会设置该标志），跳转到“lookup_parent 标号”处。如果 this.name[0] 等于'.'，那么如果 this.len 等于 2 且 this.name[1] 也等于'.'，执行 follow_dotdot(&nd->mnt, &nd->dentry)，将 nd->dentry->d_inode 传给 inode，如果 this.len 等于 1，跳转到“return_reval 标号位置”。如果 nd->dentry->d_op->d_hash 指针非空（根目录'/'的 dentry->d_op 是空的），执行 nd->dentry->d_op->d_hash(nd->dentry, &this)并将返回值传给 err，如果 err 小于 0，跳出循环。执行 do_lookup(nd, &this, &next) 查找名字快速字符串与 this 相同的 nd->dentry 的子目录条目结构，将返回值传给 err。如果 err 不等于 0，跳出循环。执行 follow_mount(&next.mnt, &next.dentry)。将 next.dentry->d_inode 传给 inode，如果 lookup_flags 设置了 LOOKUP_FOLLOW 标志（user_path_walk ()会设置该标志）并且 inode->i_op->follow_link 非空（根目录'/'的 inode->i_op->follow_link 为空），执行 mntget(next.mnt)，执行 do_follow_link(next.dentry, nd)将返回值传给 err，执行 dput(next.dentry)，执行 mntput(next.mnt)，如果 err 不等于 0，跳转到“return_err 标号”处，将 nd->dentry->d_inode 传给 inode。如果前面条件不成立，执行 dput(nd->dentry)，将 next.mnt 传给 nd->mnt，将 next.dentry 传给 nd->dentry。将-ENOENT 传给 err。如果 inode 为空，跳出循环。如果 lookup_flags 设置了 LOOKUP_DIRECTORY 标志，将-ENOTDIR 传给 err。如果 inode->i_op 为空或者 inode->i_op->lookup 为空，跳出循环，跳转到“return_base 标号”位置。“lookup_parent 标号位置”：将快速字符串结构 this 传给 nd->last，将 LAST_NORM 传给 nd->last_type。如果 this.name[0] 不等于'.'，返回 0。如果 this.len 等于 1，将 LAST_DOT 传给 nd->last_type，如果 this.len 等于 2 且 this.name[1]也等于'.'，将 LAST_DOTDOT 传给 nd->last_type，否则返回 0。“return_reval 标号”：如果 nd->dentry->d_sb->s_type->fs_flags 设置了 FS_REVAL_DOT 标志，将-ESTALE 传给 err，如果执行 nd->dentry->d_op->d_revalidate(nd->dentry, nd)返回 0，跳出循环。“return_base 标号”：返回 0；“out_dput 标号”：执行 dput(next.dentry)，跳出循环。循环结束。执行 path_release(nd)。“return_err 标号”：返回 err。

static inline int exec_permission_lite(struct inode *inode, unsigned long hash)函数执行过程：将 inode->i_mode 传给 umode_t mode；如果 inode->i_op 指针非空且 inode->i_op->permission 指针也非空（rootfs 的根目录'/'的 inode->i_op->permission 指针是空的，没有定义），返回-EAGAIN。如果 current->fsuid 等于 inode->i_uid（rootfs 的 inode->i_uid 就等于 current->fsuid），那么将 mode 右移 6 位；如果 current->fsuid 不等于 inode->i_uid 但 in_group_p(inode->i_gid)不等于 0，即 inode->i_gid 在 current->group_info 描述的群组 id 号范围内，也就是说 inode->i_gid 和 current->fsgid 相等或在同一组群组内，那么将 mode 右移 3 位。如果 mode 设置了 MAY_EXEC（等于 1）标志，返回 0。如果模 mode 设置了用户、群组和其他用户都可以执行标志 S_IXUGO，

并且当前任务 current 的有效能力标志字 cap_effective 设置了忽略对文件的所有 DAC 访问限制标志位 CAP_DAC_OVERRIDE，返回 0。如果 inode 对应的是一个目录，并且当前任务 current 的有效能力标志字 cap_effective 设置了忽略对文件的所有 DAC 访问限制标志位 CAP_DAC_OVERRID，返回 0；否则返回-EACCES。

int permission(struct inode * inode,int mask, struct nameidata *nd)函数执行过程：将 mask 与上~MAY_APPEND（等于 8）传给 int submask。如果 inode->i_op->permission 指针非空，执行 inode->i_op->permission(inode, submask, nd)（根目录'/'对应的 inode 的 i_op->permission 是空的），否则执行 generic_permission(inode, submask, NULL)，将返回值传给 retval。返回 retval，如果有访问权限，retval 等于 0。

int generic_permission(struct inode *inode, int mask, int (*check_acl)(struct inode *inode, int mask))函数执行过程：将 inode->i_mode 传给 umode_t mode；如果 mask 设置了 MAY_WRITE 位即要求 inode 可写，如果(inode)->i_sb->s_flags 设置了 MS_RDONLY 位且 inode 对应的是文件或者目录或者是符号链接，返回-EROFS，如果(inode)->i_flags 设置了 S_IMMUTABLE 标志即 inode 对应的是不可变文件，返回-EACCES。如果 current->fsuid 等于 inode->i_uid，将 mode 右移 6 位，否则如果(inode)->i_sb->s_flags 设置了 MS_POSIXACL 位且 mode 设置了 S_IRWXG 且 check_acl 函数指针非空，执行 check_acl(inode, mask)将返回值传给 int error，如果 error 等于-EACCES 跳转到“check_capabilities 标号”处，否则返回 error，如果 inode->i_gid 属于 current->group_info 所描述的群组 id 号内，将 mode 右移 3 位。如果((mode & mask & (MAY_READ|MAY_WRITE|MAY_EXEC))等于 mask，返回 0。“check_capabilities 标号位置”：如果 mask 没有设置 MAY_EXEC 位或者 inode->i_mode 设置了 S_IXUGO 或者 inode->i_mode 对应的是一个目录，如果 current->cap_effective 设置了 CAP_DAC_OVERRIDE 即忽略对文件的所有 DAC 访问限制，返回 0。如果 mask 等于 MAY_READ 或者 inode->i_mode 对应的是一个目录且 mask 没有设置 MAY_WRITE，如果 current->cap_effective 设置了 CAP_DAC_READ_SEARCH 即忽略对文件的所有读、搜索访问限制，返回 0，返回-EACCES。

static inline void follow_dotdot(struct vfsmount **mnt, struct dentry **dentry)函数执行过程：while(1)死循环：将*dentry 传给 struct dentry *old，如果*dentry 等于 current->fs->root 并且*mnt 等于 current->fs->rootmnt，退出死循环。如果 *dentry 不等于(*mnt)->mnt_root，将(*dentry)->d_parent->d_count 加 1 然后将(*dentry)->d_parent 传给*dentry。执行 dput(old)处理掉 old，退出死循环。将(*mnt)->mnt_parent 传给 struct vfsmount *parent。如果 parent 等于*mnt，退出死循环。将 parent->mnt_count 加 1，将(*mnt)->mnt_mountpoint->d_count 加 1，将(*mnt)->mnt_mountpoint 传给*dentry。执行 dput(old)处理掉 old，执行 mntput(*mnt)处理掉*mnt，将 parent 传给*mnt，while(1)死循环结束，执行 follow_mount(mnt, dentry)。

static int follow_mount(struct vfsmount **mnt, struct dentry **dentry)函数执行过程：将 res

设置为 0。只要*dentry->d_mounted 大于 0，执行一下循环（也就是查询所有挂接到该目录的文件系统）。执行 lookup_mnt(*mnt, *dentry)将返回值传给 struct vfsmount *mounted。如果 mounted 指针为空，则退出循环。执行 mntput(*mnt)处理掉*mnt，将 mounted 传给*mnt，执行 dput(*dentry)处理掉*dentry，将 mounted->mnt_root->d_count 加 1，将 mounted->mnt_root 传给*dentry，将 res 设置为 1，while()循环结束，返回 res。

struct vfsmount *lookup_mnt(struct vfsmount *mnt, struct dentry *dentry)函数执行过程：将 mount_hashtable + hash(mnt, dentry)传给 struct list_head * head；将 head 传给 struct list_head * tmp，执行 for(;;)死循环。将 tmp->next 传给 tmp，将 NULL 传给 struct vfsmount *p。如果 tmp 等于 head，退出循环。将 tmp 对应的 struct vfsmount 结构地址传给 p。如果 p->mnt_parent 等于 mnt 并且 p->mnt_mountpoint 等于 dentry，将 p->mnt_count 加 1 然后再传给 struct vfsmount *found，退出循环。for(;;)循环结束，返回 found。

static inline unsigned long hash(struct vfsmount *mnt, struct dentry *dentry)函数执行过程：将 (unsigned long) mnt / L1_CACHE_BYTES（等于 32）传给 unsigned long tmp，将 tmp 加上(unsigned long) dentry / L1_CACHE_BYTES 再传给 tmp，再将 tmp 加上(tmp >> hash_bits（等于 9））传给 tmp，返回 tmp & hash_mask（等于 511）。

static int do_lookup(struct nameidata *nd, struct qstr *name, struct path *path)函数执行过程：执行__d_lookup(nd->dentry, name)在目录条目 nd->dentry 下查找快速名字字符串结构为 name 的子目录的目录条目结构并将其指针传给 struct dentry *dentry。如果 dentry 非空，那么检查 dentry->d_op->d_revalidate()函数是否有定义，如果有则执行 dentry->d_op->d_revalidate()使 dentry 生效，如果使生效不成功则执行 d_invalidate(dentry)使 dentry 无效，如果使无效失败则相当于 dentry 有效。如果使能成功或者没有定义 dentry->d_op->d_revalidate，将 nd->mnt 传给 path->mnt，dentry 传给 path->dentry，返回 0。如果查找 dentry 失败或使 dentry 生效失败，那么执行 real_lookup(nd->dentry, name, nd) 在顺序锁 rename_lock 保护下反复在父目录 nd->dentry 下查找快速名字字符串结构为 name 的子目录或文件对应的 dentry 结构指针，如果仍没有找到，那么创建一个新的快速名字字符串结构 d_name 为 name 的 nd->dentry 的子目录传给 struct dentry *dentry，如果 dentry 错误，返回 PTR_ERR(dentry)；如果 dentry 有效，将 nd->mnt 传给 path->mnt，将 dentry 传给 path->dentry，返回 0。

struct dentry * __d_lookup(struct dentry * parent, struct qstr * name)函数执行过程：将 name->len 传给 unsigned int len，将 name->hash 传给 unsigned int hash，将 name->name 传给 const unsigned char *str，将 d_hash(parent,hash)传给 struct hlist_head *head，将 NULL 传给 struct dentry *found。在目录条目 hash 链表 head 中找到其 d_name 与 name 完全相同且其 d_parent 等于 parent 的目录条目结构指针 dentry。如果 dentry 非空且 dentry->d_flags 没有设置 DCACHE_UNHASHED 位，那么将 dentry->d_count 加 1 后再将 dentry 传给 found，返回 found。

static inline struct hlist_head *d_hash(struct dentry *parent, unsigned long hash)函数执行过程：将 hash 加上(unsigned long) parent ^ GOLDEN_RATIO_PRIME) / L1_CACHE_BYTES 传给 hash，将 hash ^ ((hash ^ GOLDEN_RATIO_PRIME) >> D_HASHBITS)传给 hash，返回 dentry_hashtable + (hash & D_HASHMASK)。

static struct dentry * real_lookup(struct dentry * parent, struct qstr * name, struct nameidata *nd)函数执行过程：将 parent->d_inode 传给 struct inode *dir，执行 d_lookup(parent, name)将返回值传给 struct dentry * result。如果 result 指针空，那么调用 d_alloc(parent, name)为 parent 创建一个新的子目录条目结构 dentry，将 name 复制到 dentry->d_name，然后将 ERR_PTR(-ENOMEM)传给 result，如果 dentry 非空，执行 dir->i_op->lookup(dir, dentry, nd)（对于 rootfs 的'/'目录，dir->i_op->lookup(dir, dentry, nd)就是 simple_lookup()函数）并将返回值传给 result，如果 result 非空即失败，执行 dput(dentry)处理掉 dentry，则将 dentry 传给 result，返回 result；如果前面 d_lookup(parent, name)返回的 result 非空，且 result->d_op->d_revalidate 非空，那么执行 result->d_op->d_revalidate(result, nd)和 d_invalidate(result)，如果两个执行结果都是 0，执行 dput(result)处理掉 result，将 ERR_PTR(-ENOENT)传给 result。

struct dentry * d_lookup(struct dentry * parent, struct qstr * name)函数执行过程：将 NULL 传给 struct dentry * dentry，执行 read_seqbegin(&rename_lock)将返回值传给 unsigned long seq，执行__d_lookup(parent, name)将返回值传给 dentry。如果 dentry 指针非空，跳出循环。只要 read_seqretry(&rename_lock, seq)不等于 0，重复以上操作，返回 dentry。

struct dentry *simple_lookup(struct inode *dir, struct dentry *dentry, struct nameidata *nd)函数执行如下：声明并初始化静态结构变量 static struct dentry_operations simple_dentry_operations {.d_delete = simple_delete_dentry,}。如果 dentry->d_name.len > NAME_MAX，则返回 ERR_PTR(-ENAMETOOLONG)。将 &simple_dentry_operations 传给 dentry->d_op；执行 d_add(dentry, NULL)，返回 NULL。

static inline void d_add(struct dentry *entry, struct inode *inode)函数执行过程：执行 d_instantiate(entry, inode)，执行 d_rehash(entry)。

void d_instantiate(struct dentry *entry, struct inode * inode)函数执行过程：如果 entry->d_alias 链表非空，则系统崩溃。如果 inode 指针非空，将 entry 通过 entry->d_alias 连接到 inode->i_dentry 链表开始位置，将 inode 传给 entry->d_inode。

void d_rehash(struct dentry * entry)函数执行过程：将 d_hash(entry->d_parent, entry->d_name.hash)传给 struct hlist_head *list，执行__d_rehash(entry, list)。

static void __d_rehash(struct dentry * entry, struct hlist_head *list)执行过程：清除 entry->d_flags 中的 DCACHE_UNHASHED 位，执行 hlist_add_head_rcu(&entry->d_hash, list)将

entry 通过其 d_hash 连接到目录条目结构 hash 表 dentry_hashtable 的 list 链表开始。

asmlinkage long sys_mkdir(const char __user * pathname, int mode)函数执行过程：将 int error 设置为 0，执行 tmp=getname(pathname)将用户空间的目录名字 pathname 复制到内核空间的 char * tmp 位置。将 PTR_ERR(tmp)传给 int error，如果 tmp 指针无误，执行 path_lookup(tmp, LOOKUP_PARENT, &（struct nameidat）nd)并将返回值传给 error，如果 error 不等于 0，则执行 putname(tmp)，返回 error。执行 lookup_create(&nd, 1)并将返回值传给 struct dentry *dentry，将 PTR_ERR(dentry)传给 error。如果 dentry 指针无误，且 nd.dentry->d_inode->i_sb->s_flags 没有设置 MS_POSIXACL 标志位（根目录'/'对应的 inode->i_sb 只设置了 MS_ACTIVE 位），那么清除 mode 中 current->fs->umask 为 1 的位，执行 vfs_mkdir(nd.dentry->d_inode, dentry, mode)将返回值传给 error，执行 dput(dentry)。执行 path_release(&nd)，执行 putname(tmp)，返回 error。

struct dentry *lookup_create(struct nameidata *nd, int is_dir) 函数执行过程：将 ERR_PTR(-EEXIST)传给 struct dentry *dentry。如果 nd->last_type 不等于 LAST_NORM，则返回 dentry。清除 nd->flags 中的 LOOKUP_PARENT 标志，执行 lookup_hash(&nd->last, nd->dentry)将返回值传给 dentry。如果 dentry 指针出错返回 dentry。如果 is_dir 等于 0，即不是要创建目录且 nd->last.name[nd->last.len]不等于 0 且 dentry->d_inode 指针空，执行 dput(dentry)将 ERR_PTR(-ENOENT)传给 dentry，返回 dentry。

struct dentry * lookup_hash(struct qstr *name, struct dentry * base)函数执行过程：直接执行并返回__lookup_hash(name, base, NULL)。

static struct dentry * __lookup_hash(struct qstr *name, struct dentry * base, struct nameidata *nd)函数执行过程：将 base->d_inode 传给 struct inode *inode，执行 permission(inode, MAY_EXEC, nd)检查 inode->i_mode 设否设置了 MAY_EXEC 位即该 inode 是否可以执行并将返回值传给 err。将 ERR_PTR(err)传给 dentry。如果 err 不等于 0，返回 dentry。如果 base->d_op->d_hash 指针非空（跟文件系统 rootfs 的根目录'/'对应的目录条目结构 dentry->d_op 是 NULL），执行 base->d_op->d_hash(base, name)并将返回值传给 err，将 ERR_PTR(err)传给 dentry，如果 err 小于 0，返回 dentry。执行 cached_lookup(base, name, nd)并将返回值传给 dentry。如果 dentry 为空，执行 d_alloc(base, name)为 base 创建一个*d_name 与*name 相同的子目录的目录条目结构，将返回值传给 struct dentry *new。将 ERR_PTR(-ENOMEM)传给 dentry，如果 new 为空，返回 dentry。执行 inode->i_op->lookup(inode, new, nd)（如果 base 是与根目录'/'对应的目录条目，那么 inode->i_op->lookup()就是 simple_lookup()函数）并将返回值传给 dentry。如果 dentry 为空，说明上面的操作是正确的，将 new 传给 dentry，否则执行 dput(new)。返回 dentry。

static struct dentry * cached_lookup(struct dentry * parent, struct qstr * name, struct nameidata *nd)函数执行过程：执行__d_lookup(parent, name)在 dentry_hashtable 的链表中是否已经创建要找的目录条目结构，将返回值传给 struct dentry * dentry。如果 dentry 指针为空，执行

d_lookup(parent, name)再次检查 dentry_hashtable 的链表中是否已经创建要找的目录条目结构将返回值传给 dentry。如果 dentry 非空，且 dentry->d_op->d_revalidate 指针非空，那么执行 dentry->d_op->d_revalidate(dentry, nd)和 d_invalidate(dentry)，如果执行结果都是 0，执行 dput(dentry)，将 NULL 传给 dentry，返回 dentry。

int vfs_mkdir(struct inode *dir, struct dentry *dentry, int mode)函数执行过程：执行 may_create(dir, dentry, NULL)检查 dir 是否能被写并且能被执行，将返回值传给 int error。如果 error 不等于 0，返回 error。如果 dir->i_op 指针为空或者 dir->i_op->mkdir 指针为空（根目录'/'对应的节点 inode->i_op 指向 ramfs_dir_inode_operations，其 mkdir 指向 ramfs_mkdir()函数），返回-EPERM。清除 mode 中的 S_IRWXUGO|S_ISVTX 标志。执行 security_inode_mkdir(dir, dentry, mode)并将返回值传给 error，如果 error 不等于 0，返回 error；执行 DQUOT_INIT(dir)（只有配置了 CONFIG_QUOTA 选项即支持磁盘配额的系统执行有意义）。执行 dir->i_op->mkdir(dir, dentry, mode)并将返回值传给 error。如果 error 等于 0 即创建目录对应的索引节点结构 struct inode 成功，执行 inode_dir_notify(dir, DN_CREATE)通知系统创建了目录，执行 security_inode_post_mkdir(dir,dentry, mode)，返回 error。

static inline int may_create(struct inode *dir, struct dentry *child, struct nameidata *nd)函数执行过程：如果 child->d_inode 指针非空，则返回-EEXIST，如果 dir->i_sb->s_flags 设置了 S_DEAD 标志，返回-ENOENT。执行并返回 permission(dir,MAY_WRITE | MAY_EXEC, nd)。

static int ramfs_mkdir(struct inode * dir, struct dentry * dentry, int mode)函数执行过程：执行 ramfs_mknod(dir, dentry, mode | S_IFDIR, 0)将返回值传给 int retval，如果 retval 等于 0，将 dir->i_nlink 加 1，返回 retval。

static int ramfs_mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t dev)函数执行过程：执行 ramfs_get_inode(dir->i_sb, mode, dev)像获取'/'目录对应的 struct inode 结构一样获取 dentry 对应的 struct inode 结构，将返回值传给 struct inode *inode，也就是说 dentry 的 inode 继承了'/'目录的 inode 中大部分的设置。将-ENOSPC 传给 int error。如果 inode 指针非空，且 dir->i_mode 设置了 S_ISGID 标志（根目录'/'的 inode->i_mode 等于 S_IFDIR | 0755 即 040755，没有设置 S_ISGID 标志），将 dir->i_gid 传给 inode->i_gid，如果 mode 设置了 S_IFDIR 即 dentry 对应一个目录，将 inode->i_mode 或上 S_ISGID。执行 d_instantiate(dentry, inode)，将 dentry 通过 dentry->d_alias 连接到 inode->i_dentry 链表开始位置。将 inode 传给 entry->d_inode，执行 dget(dentry)，将 error 设置为 0，返回 error。

static inline void inode_dir_notify(struct inode *inode, unsigned long event)函数执行过程：如果系统没有配置 CONFIG_DNOTIFY 选项（嵌入式系统中都配置，用来通知系统文件文件的访问、改变、创建、删除、重命名、改变属性等），该函数什么也不做，是空函数；如果配置了 CONFIG_DNOTIFY 选项，且 inode->i_dnotify_mask('/'目录的 inode->i_dnotify 为 0，)与上 event

不等于 0，执行__inode_dir_notify(inode, event)函数，后续略。

asmlinkage long sys_chown(const char __user * filename, uid_t user, gid_t group)函数执行过程：执行 user_path_walk(filename, &nd)->__user_walk(filename, LOOKUP_FOLLOW, nd)将返回值传给 int error。如果 error 等于 0，执行 chown_common(nd.dentry, user, group)将返回值传给 error，执行 path_release(&nd)，返回 error。

int fastcall __user_walk(const char __user *name, unsigned flags, struct nameidata *nd)函数执行过程：执行 getname(name)将用户空间名字 name 复制到内核空间 char *tmp，将 PTR_ERR(tmp)传给 int err。如果 tmp 无误，执行 path_lookup(tmp, flags, nd)将返回值传给 err，执行 putname(tmp)，返回 err。

static int chown_common(struct dentry * dentry, uid_t user, gid_t group)函数执行过程：将-ENOENT 传给 int error，将 dentry->d_inode 传给 struct inode * inode，如果 inode 为空，打印“chown_common：空 inode”，跳转到“out 标号”处，将-EROFS 传给 error。如果 inode->i_sb->s_flags 设置了 MS_RDONLY（根目录'/'所在的文件系统的 super_block 结构的 s_flags 值设置了 MS_ACTIVE）标志即文件系统只读，跳转到“out 标号”处，将-EPERM 传给 error。如果(inode)->i_flags 设置了 S_IMMUTABLE 标志即 inode 对应的是不可变文件或(inode)->i_flags 设置了 S_APPEND 即 inode 只能往里添加，跳转到“out 标号”处。将 ATTR_CTIME 传给 struct iattr newattrs. ia_valid。如果 user 不等于-1，将 newattrs.ia_valid 或上 ATTR_UID，将 user 传给 newattrs.ia_uid。如果 group 不等于-1，将 newattrs.ia_valid 或上 ATTR_GID，将 group 传给 newattrs.ia_gid。如果 inode->i_mode 没有设置 S_IFDIR 即 inode 对应的不是目录，将 newattrs.ia_valid 或上 ATTR_KILL_SUID|ATTR_KILL_SGID，执行 notify_change(dentry, &newattrs)通知改变将返回值传给 error。“out 标号”：返回 error。

int notify_change(struct dentry * dentry, struct iattr * attr)函数执行过程：将 dentry->d_inode 传给 struct inode *inode，将 inode->i_mode 传给 mode_t mode，将 CURRENT_TIME 即此刻系统的 xtime 传给 struct timespec now，将 attr->ia_valid 传给 unsigned int ia_valid。如果 inode 为空，则系统崩溃。将 now 传给 attr->ia_ctime。如果 ia_valid 没有设置 ATTR_ATIME_SET，将 now 也传给 attr->ia_atime。如果 ia_valid 没有设置 ATTR_MTIME_SET，将 now 再传给 attr->ia_mtime。如果 ia_valid 设置了 ATTR_KILL_SUID，清除 attr->ia_valid 中的 ATTR_KILL_SUID 标志，如果 mode 设置了 S_ISUID（根目录'/'的 inode->i_mode 等于 S_IFDIR | 0755 即 040755，没有设置 S_ISGID 标志），如果 ia_valid 没有设置 ATTR_MODE，将 attr->ia_valid 或上 ATTR_MODE 然后传给 ia_valid，将 inode->i_mode 传给 attr->ia_mode，清除 attr->ia_mode 中的 S_ISUID 标志。如果 ia_valid 设置了 ATTR_KILL_SGID（如果 inode 对应的是目录则不会设置该位），清除 attr->ia_valid 中的 ATTR_KILL_SGID 标志，如果 mode 同时设置了 S_ISGID | S_IXGRP 标志，如果 ia_valid 没有设置 ATTR_MODE，将 attr->ia_valid 或上 ATTR_MODE 再传给 ia_valid，将

inode->i_mode 传给 attr->ia_mode，清除 attr->ia_mode 中的 S_ISGID 标志。如果 attr->ia_valid 等于 0，返回 0；如果 inode->i_op->setattr 指针非空（根目录'/'及其子目录的 inode->i_op->setattr 为空），执行 security_inode_setattr(dentry, attr)将返回值传给 error，如果 error 等于 0，执行 inode->i_op->setattr(dentry, attr)将返回值传给 error，如果 inode->i_op->setattr 指针空，执行 inode_change_ok(inode, attr) 将返回值传给 error，如果 error 等于 0，执行 security_inode_setattr(dentry, attr)将返回值传给 error，如果 error 等于 0，且 ia_valid & ATTR_UID && attr->ia_uid != inode->i_uid（inode 被创建时其 i_uid 被设置为 current->fsuid）或者 ia_valid & ATTR_GID && attr->ia_gid != inode->i_gid（inode 被创建时其 i_gid 被设置为 current-> fsgid），执行 DQUOT_TRANSFER(inode, attr) ? -EDQUOT : 0，如果 error 等于 0，执行 inode_setattr(inode, attr)将返回值传给 error。如果 error 等于 0，执行 setattr_mask(ia_valid)将返回值传给 unsigned long dn_mask。如果 dn_mask 不等于 0，执行 dnotify_parent(dentry, dn_mask)，返回 error。

asmlinkage long sys_chmod(const char __user * filename, mode_t mode)函数执行过程：执行 user_path_walk(filename, &nd)->__user_walk(filename, LOOKUP_FOLLOW, nd)将返回值传给 int error。如果 error 不等于 0，返回 error。将 nd.dentry->d_inode 传给 struct inode * inode，将-EROFS 传给 error。如果 inode->i_sb->s_flags 设置了 MS_RDONLY（根目录'/'所在的文件系统的 super_block 结构的 s_flags 值设置了 MS_ACTIVE）标志即文件系统只读，执行 path_release(&nd)，返回 error。将-EPERM 传给 error，如果(inode)->i_flags 设置了 S_IMMUTABLE 标志即 inode 对应的是不可变文件或(inode)->i_flags 设置了 S_APPEND 即 inode 只能往里添加，执行 path_release(&nd)，返回 error。如果 mode 等于-1，将 inode->i_mode 传给 mode，将(mode & S_IALLUGO) | (inode->i_mode & ~S_IALLUGO)传给 struct iattr newattrs.ia_mode，将 ATTR_MODE | ATTR_CTIME 传给 newattrs.ia_valid，执行 notify_change(nd.dentry, &newattrs)将返回值传给 error，执行 path_release(&nd)，返回 error。

asmlinkage long sys_mknod(const char __user * filename, int mode, unsigned dev)函数执行过程：将 int error 设置为 0，将用户空间的目录名字 pathname 复制到内核空间的 char * tmp 位置。如果 mode 设置了 S_IFDIR 标志即要创建的是目录，返回-EPERM。执行 tmp=getname(pathname)将用户空间的目录名字 pathname 复制到内核空间的 char * tmp 位置。如果 tmp 指针有误，返回 PTR_ERR(tmp)；执行 path_lookup(tmp, LOOKUP_PARENT, &（struct nameidat）nd)并将返回值传给 error，如果 error 不等于 0 则执行 putname(tmp)，返回 error。执行 lookup_create(&nd, 0)为设备节点创建目录条目结构并将返回值传给 struct dentry *dentry，将 PTR_ERR(dentry)传给 error。如果 nd.dentry->d_inode->i_sb->s_flags 没有设置 MS_POSIXACL 标志，将 mode 与上 ~current->fs->umask。如果 dentry 指针无误，根据(mode & S_IFMT)的值进行操作：如果等于 0 或 S_IFREG，执行 vfs_create(nd.dentry->d_inode,dentry,mode,&nd)将返回值传给 error。如果等于 S_IFCHR 或 S_IFBLK，执行 vfs_mknod(nd.dentry->d_inode,dentry,mode, new_decode_dev(dev))将返回值传给 error。如果等于 S_IFIFO 或 S_IFSOCK，执行 vfs_mknod(nd.dentry->d_inode,

dentry,mode,0)将返回值传给 error。如果等于 S_IFDIR，将-EPERM 传给 error。如果等于其他值，将-EINVAL 传给 error。执行 dput(dentry)，执行 path_release(&nd)，执行 putname(tmp)，返回 error。

int vfs_create(struct inode *dir, struct dentry *dentry, int mode, struct nameidata *nd)执行过程：执行 may_create(dir, dentry, nd)检查是否可以写并执行 dir（dir 是 dentry 的父目录的 d_inode）将返回值传给 error。如果 error 不等于 0，返回 error，如果 dir->i_op 为空或 dir->i_op->create 为空（根目录'/'的 dir->i_op 指向 ramfs_dir_inode_operations，dir->i_op->create 指向 ramfs_create），返回-EACCES。将 mode 与上 S_IALLUGO，将 mode 或上 S_IFREG。执行 security_inode_create(dir, dentry, mode)将返回值传给 error。如果 error 不等于 0，返回 error，执行 DQUOT_INIT(dir)，执行 dir->i_op->create(dir, dentry, mode, nd)将返回值传给 error。如果 error 等于 0，执行 inode_dir_notify(dir, DN_CREATE)，执行 security_inode_post_create(dir, dentry, mode)，返回 error。

static int ramfs_create(struct inode *dir, struct dentry *dentry, int mode, struct nameidata *nd)函数执行过程：直接执行返回 ramfs_mknod(dir, dentry, mode | S_IFREG, 0)。

int vfs_mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t dev)函数执行过程：执行 may_create(dir, dentry, nd)检查是否可以写并执行 dir（dir 是 dentry 的父目录的 d_inode）将返回值传给 error。如果 error 不等于 0，返回 error。如果 mode 中设置了 S_IFCHR 或 S_IFBLK 且 current->cap_effective 中并没有设置 CAP_MKNOD 位即当前进程没有创建节点的权限，返回-EPERM。如果 dir->i_op 为空或 dir->i_op->mknod 为空（根目录'/'的 dir->i_op 指向 ramfs_dir_inode_operations，dir->i_op->mknod 指向 ramfs_mknod），返回-EPERM。执行 security_inode_mknod(dir, dentry, mode, dev)将返回值传给 error。如果 error 不等于 0，返回 error。执行 DQUOT_INIT(dir)，执行 dir->i_op->mknod(dir, dentry, mode, nd)将返回值传给 error。如果 error 等于 0，执行 inode_dir_notify(dir, DN_CREATE)，执行 security_inode_post_ mknod(dir, dentry, mode)，返回 error。

int vfs_symlink(struct inode *dir, struct dentry *dentry, const char *oldname, int mode)函数执行过程：执行 may_create(dir, dentry, NULL)检查 dir 是否能被写并且能被执行，将返回值传给 int error；如果 dir->i_op 为空或 dir->i_op->symlink 为空（根目录'/'的 dir->i_op 指向 ramfs_dir_inode_operations，dir->i_op->symlink 指向 ramfs_symlink），返回-EPERM。执行 security_inode_symlink(dir, dentry, oldname)将返回值传给 error；如果 error 不等于 0，返回 error。执行 DQUOT_INIT(dir)，执行 dir->i_op->symlink(dir, dentry, oldname)将返回值传给 error。如果 error 等于 0，执行 inode_dir_notify(dir, DN_CREATE)，执行 security_inode_post_symlink(dir, dentry, oldname)，返回 error。

static int ramfs_symlink(struct inode * dir, struct dentry *dentry, const char * symname)执行过程：将-ENOSPC 传给 error，执行 ramfs_get_inode(dir->i_sb, S_IFLNK|S_IRWXUGO, 0)将返回值传给 struct inode *inode。如果 inode 非空，将 sysname 字符串的长度加 1 传给 int l。执行

page_symlink(inode, symname, l)将返回值传给 error。如果 error 等于 0，且 dir->i_mode 设置了 S_ISGID 标志（根目录'/'的 inode->i_mode 等于 S_IFDIR | 0755 即 040755，没有设置 S_ISGID 标志），将 dir->i_gid 传给 inode->i_gid，执行 d_instantiate(dentry, inode)，执行 dget(dentry)，如果 error 不等于 0，执行 iput(inode)，返回 error。

int page_symlink(struct inode *inode, const char *symname, int len)函数执行过程：将 inode->i_mapping 传给 struct address_space *mapping，执行 grab_cache_page(mapping, 0)将返回值传给 struct page *page。如果 page 为空，返回-ENOMEM。执行 mapping->a_ops->prepare_write(NULL, page, 0, len-1)（rootfs 文件系统中所有 inode 的 inode->i_mappint->a_ops 指向 ramfs_aops，inode->i_mappint->a_ops->prepare_write 等于 simple_prepare_write）将返回值传给 err。如果 err 不等于 0，跳转到“fail_map 标号”位置。将 kmap_atomic(page, KM_USER0)传给 char *kaddr，执行 memcpy(kaddr, symname, len-1)，执行 mapping->a_ops->commit_write(NULL, page, 0, len-1)（rootfs 文件系统中的 inode->i_mappint->a_ops->commit_write 等于 simple_commit_write）。如果 (page)->flags 没有设置 PG_uptodate 标志位，执行 mapping->a_ops->readpage(NULL, page)（rootfs 文件系统中的 inode->i_mappint->a_ops->readpage 等于 simple_readpage）将返回值传给 err，执行 wait_on_page_locked(page)，如果 (page)->flags 设置了 PG_uptodate 标志位，执行 unlock_page(page)，执行 page_cache_release(page)。如果 err 小于 0，返回 err。执行 mark_inode_dirty(inode)，返回 0。“fail_map 标号位置”：执行 unlock_page(page)，执行 page_cache_release(page)，返回 err。

static inline struct page *grab_cache_page(struct address_space *mapping, unsigned long index)函数执行过程：直接执行并返回 find_or_create_page(mapping, index, mapping_gfp_mask(mapping))。

static inline int mapping_gfp_mask(struct address_space * mapping)函数执行过程：返回 mapping->flags & __GFP_BITS_MASK（新创建的 ionde 的 inode->i_mapping->flags 都被设置为 0）。

struct page *find_or_create_page(struct address_space *mapping, unsigned long index, unsigned int gfp_mask)函数执行过程：将 struct page *cached_page 设置为 NULL。“repeat 标号位置”：执行 find_lock_page(mapping, index)将返回值传给 struct page *page。如果 page 为空，且 cached_page 为空，执行 alloc_page(gfp_mask)申请一页内存并将返回值传给 cached_page，如果 cached_page 为空，返回 NULL，执行 add_to_page_cache_lru(cached_page, mapping, index, gfp_mask)将返回值传给 err，如果 err 等于 0，将 cached_page 传给 page，将 cached_page 设置为 NULL，否则如果 err 等于-EEXIST，跳转到“repeat 标号位置”。如果 cached_page 不等于 NULL，执行 page_cache_release(cached_page)，返回 page。

struct page *find_lock_page(struct address_space *mapping, unsigned long offset)函数执行过

程："repeat 标号位置"：执行 radix_tree_lookup(&mapping->page_tree, offset)（新创建的 inode 的 inode->imappint->（struct radix_tree_root）page_tree 被设置为 height 等于 0，gfp_mask 等于 GFP_ATOMIC，rnode 等于 NULL;）将返回值传给 struct page *page。如果 page 非空，执行 page_cache_get(page)，返回 page。

void *radix_tree_lookup(struct radix_tree_root *root, unsigned long index)执行过程：将 root->height 传给 unsigned int height。如果 index 大于 radix_tree_maxindex(height)，返回 NULL。将(height-1) * RADIX_TREE_MAP_SHIFT 传给 shift（如果 height 等于 0，则 shift<0），将 &root->rnode 传给 slot。while (height > 0)循环：如果*slot 等于 NULL，返回 NULL。将(*slot)->slots+((index >> shift) & RADIX_TREE_MAP_MASK)即 index 索引的对象号所属该 slot 中的(void*)slots 指针组员的地址传给 slot，将 shift 减去 RADIX_TREE_MAP_SHIFT，将 height 减 1，while 循环结束位置，返回*slot（此时的*slot 是 index 索引的对象所在的 struct radix_tree_node 结构的 slots，事实上就是 index 索引的对象的地址）。

asmlinkage long sys_lchown(const char __user * filename, uid_t user, gid_t group)函数执行过程：执行 user_path_walk_link(filename, &nd)将返回值传给 int error。如果 error 等于 0，执行 chown_common(nd.dentry, user, group)将返回值传给 error，执行 path_release(&nd)，返回 error。

asmlinkage long sys_fchown(unsigned int fd, uid_t user, gid_t group)函数执行过程：将 -EBADF 传给 int error，执行 fget(fd)将句柄 fd 对应的文件描述结构 struct file 的引用计数器 f_count 加 1 然后将该文件描述结构地址传给 struct file *file。如果 file 非空，执行 chown_common(file->f_dentry, user, group)，将返回值传给 error，执行 fput(file)，返回 error。

asmlinkage long sys_fchmod(unsigned int fd, mode_t mode)函数执行过程：将-EBADF 传给 int err；执行 fget(fd)将句柄 fd 对应的文件描述结构 struct file 的引用计数器 f_count 加 1 然后将该文件描述结构地址传给 struct file *file。如果 file 为空，返回 err。将 file->f_dentry 传给 struct dentry * dentry，将 dentry->d_inode 传给 struct inode * inode，将-EROFS 传给 err。如果(inode)->i_sb->s_flags 设置了 MS_RDONLY 即该文件只读，跳转到"out_putf 标号"处，将-EPERM 传给 err。如果(inode)->i_flags 设置了 S_IMMUTABLE 标志即 inode 对应的是不可变文件或(inode)->i_flags 设置了 S_APPEND 即 inode 只能往里添加，跳转到"out_putf 标号"处。如果 mode 等于-1，将 inode->i_mode 传给 mode，将(mode & S_IALLUGO) | (inode->i_mode & ~S_IALLUGO)传给 struct iattr newattrs.ia_mode，将 ATTR_MODE | ATTR_CTIME 传给 newattrs.ia_valid。执行 notify_change(dentry, &newattrs)将返回值传给 err。"out_putf 标号"位置：执行 fput(file)，返回 err。

asmlinkage ssize_t sys_write(unsigned int fd, const char __user * buf, size_t count)函数执行过程：将-EBADF 传给 ssize_t ret，执行 fget_light(fd, &(int)fput_needed)将返回值传给 file。如果 file 非空，将 file->f_pos 传给 loff_t pos，执行 vfs_write(file, buf, count, &pos)将返回值传给 ret，

将 pos 传给 file->f_pos，执行 fput_light(file, fput_needed)，返回 ret。

void free_initrd_mem(unsigned long start, unsigned long end)函数执行过程：如果 keep_initrd 等于 0，执行 free_area(start, end, "initrd")。

static inline void free_area(unsigned long addr, unsigned long end, char *s)函数执行过程：将 (end - addr) >> 10 传给 unsigned int size；for (; addr < end; addr += PAGE_SIZE)循环。将 virt_to_page(addr)传给 struct page *page，将 page->flags 中的 PG_reserved 标志位清除，将 page->_count 设置为 0，执行 free_page(addr)，将系统空闲内存总页数 totalram_pages 加 1，循环结束位置。如果 size 和 s 都不等于 0，打印"Freeing %s memory: %dK\n", s, size 内核信息。

17.6.7 do_basic_setup()函数

执行 init_workqueues()，执行 usermodehelper_init()，执行 key_init()，执行 driver_init()，如果配置了 CONFIG_SYSCTL 选项，执行 sysctl_init()，执行 sock_init()，执行 do_initcalls()。

void init_workqueues(void)函数执行过程：执行 hotcpu_notifier(workqueue_cpu_callback, 0)按从高到低的优先级顺序将通知块结构变量 workqueue_cpu_callback_nb 注册到 CPU 通知者结构变量链表 cpu_chain 中；执行 create_workqueue("events")创建名叫“事件”的工作队列结构并将返回值传给静态全局结构指针 static struct workqueue_struct *keventd_wq。如果 keventd_wq 为空，则系统崩溃。

hotcpu_notifier(fn, pri)宏在没有配置 CONFIG_HOTPLUG_CPU 的系统中是空函数。在配置了 CONFIG_HOTPLUG_CPU 选项即支持 CPU 热插拔的系统中执行过程为：先声明一个 static struct notifier_block 结构变量#fn_nb，将 fn 传给#fn_nb.notifier_call，将 pri 传给#fn_nb.priority，然后执行 register_cpu_notifier(&fn##_nb)按从高到低的优先级顺序将通知块结构变量 fn##_nb 注册到 CPU 通知者结构变量链表 cpu_chain 中。

create_workqueue(name)宏定义为__create_workqueue((name), 0)。struct workqueue_struct *__create_workqueue(const char *name, int singlethread)函数执行过程如下：如果 name 超过 10 个字符，则系统崩溃。执行 kmalloc(sizeof(*wq), GFP_KERNEL)将返回值传给 struct workqueue_struct *wq。如果 wq 为空，返回 NULL。将 wq 整个结构空间清 0，将 name 传给 wq->name。如果 singlethread 不等于 0，初始化 wq->list 链表，执行 create_workqueue_thread(wq, 0)为工作队列创建看护进程将返回值传给 struct task_struct *p，如果 p 为空，将 int destroy 设置为 1，否则执行 wake_up_process(p)唤醒进程 p。如果 singlethread 等于 0，将 wq 通过 wq->list 链接到系统全局工作队列链表 workqueues 中，然后对每个在线的 cpu（cpu_online_map 对应位为 1 的 cpu）执行 create_workqueue_thread(wq, cpu)将返回值传给 p，如果 p 非空，执行 kthread_bind(p, cpu)，执行 wake_up_process(p)，否则将 int destroy 设置为 1。如果 int destroy 不等于 0，执行 destroy_workqueue(wq)销毁 wq，将 wq 设置为 NULL，返回 wq。

static struct task_struct *create_workqueue_thread(struct workqueue_struct *wq, int cpu)函数执行过程：将 wq->cpu_wq + cpu 传给 struct cpu_workqueue_struct *cwq，将 wq 传给 cwq->wq，将 cwq->thread 设置为 NULL。将 cwq->insert_sequence 和 cwq->remove_sequence 设置为 0，初始化 cwq->worklist 进程链表，初始化尚未完成的等待队列链表 cwq->more_work 和已经完成的等待队列链表 cwq->work_done。如果 wq->list 链表是空的，执行 kthread_create(worker_thread, cwq, "%s", wq->name)创建与工作队列名字同名的看护进程将返回值传给 p，否则，执行 kthread_create(worker_thread, cwq, "%s/%d", wq->name, cpu)将返回值传给 p。如果 p 指针错误，返回 NULL，将 p 传给 cwq->thread，返回 p。

void __init usermodehelper_init(void)函数执行过程：执行 create_singlethread_workqueue ("khelper")创建名叫“内核帮助器”的工作队列结构并将返回值传给静态全局结构指针 static struct workqueue_struct *khelper_wq。如果 khelper_wq 为空，则系统崩溃。

create_singlethread_workqueue(name)宏定义为__create_workqueue((name), 1)。

void __init key_init(void)函数执行过程：key_init(void)函数在没有配置 CONFIG_KEYS 选项的系统中是空函数，什么也不做。CONFIG_KEYS 选项用于支持内核保持认证口令和访问密钥，一般不配置。在配置了 CONFIG_KEYS 选项的系统中的执行过程略。

void __init driver_init(void)函数执行过程：执行 devices_init()，执行 buses_init()，执行 classes_init()，执行 firmware_init()，执行 platform_bus_init()，执行 system_bus_init()，执行 cpu_dev_init()。

int __init devices_init(void)函数执行过程：直接调用并返回 subsystem_register (&devices_subsys)。

int subsystem_register(struct subsystem * s)函数执行过程：执行 subsystem_init(s)，初始化 s 的读写信号量 rwsem，执行 kset_init(&s->kset)函数初始化 s 的内核套件描述结构成原变量 kset。将 s.kset.kobj.kref.refcount 设置为 1，将 s.kset.kobj.entry 链表初始化为空链表，将 s.kset.kobj.kref.refcount 加 1，将 s.kset.list 链表初始化为空链表。打印("subsystem %s: registering\n",s->kset.kobj.name)，执行 kset_add(&s->kset)将 s->kset 的对象添加到内核中返回值传给 int error，如果 error 等于 0，且 s->kset.subsys 为空，将 s 传给 s->kset.subsys，返回 error。

int kset_add(struct kset * k)函数执行过程：如果 k->kobj.parent 为空且 k->kobj.kset 为空且 k->subsys 非空，将 k->kobj.parent 指向 k->subsys->kset.kobj，执行并返回 kobject_add(&k->kobj)。

int kobject_add(struct kobject * kobj)函数执行过程：执行 kobject_get(kobj)将 kobj->kref->refcount 加 1 再传回给 kobj，如果 kobj 指针为空，返回-ENOENT。如果 kobj->k_name 指针为空，将 kobj->name 传给 kobj->k_name。先将 kobj->parent->kref->refcount 加 1 再传给 struct kobject * parent，打印("kobject %s: registering. parent: %s, set: %s\n", kobject_name(kobj), parent ? kobject_name(parent) : "<NULL>", kobj->kset ? kobj->kset->kobj.name : "<NULL>")。如果

kobj->kset 非空，且 parent 为空，将 kobj->kset->kobj.kref->refcount 加 1 再传给 parent，将 kobj 通过 kobj->entry 添加到 kobj->kset->list 链表末尾，将 parent 传给 kobj->parent，执行 create_dir(kobj)将返回值传给 int error。如果 error 不等于 0，执行 unlink(kobj) 将内核对象 kobj 从其所在的 entry 连表中删除并初始化 kobj->entry 最后调用 kobject_release()释放 kobj->kref，如果 parent 非空，执行 kobject_put(parent)，如果 error 等于 0，执行 kobject_hotplug(kobj, KOBJ_ADD)（略，该函数在没有配置 CONFIG_HOTPLUG 选项的系统中什么也不做，只有在 ARCH_S390 中默认配置 CONFIG_HOTPLUG，支持热插拔设备），返回 error。

static int create_dir(struct kobject * kobj) （这里的 create_dir()是 lib\kobject.c 中的函数）函数执行过程：将 int error 设置为 0。如果 kobj->k_name 指针非空，执行 sysfs_create_dir(kobj) 将返回值传给 error，如果 error 等于 0，执行 populate_dir(kobj) 遍历内核对象 kobj 对应的内核对象类型结构的各个默认属性结构，为每种属性分别创建一个文件型的新系统文件系统目录条目结构，将属性结构指针作为该系统文件系统目录条目的要素指针，传给 error，如果 error 不等于 0，执行 sysfs_remove_dir(kobj)，返回 error。

int sysfs_create_dir(struct kobject * kobj)函数执行过程：（所有 sysfs_开始的函数都有两种定义，一种是在系统定义了 CONFIG_SYSFS 宏时的定义，一种是没有定义该宏时的定义，本文中详细说明的都是定义了 CONFIG_SYSFS 宏时各函数的定义，因为没有定义该宏时各函数不做任何事直接返回 0）如果 kobj 为空，则系统崩溃；如果 kobj->parent 非空，将 kobj->parent->dentry 传给 parent，否则如果 sysfs_mount 非空且 sysfs_mount->mnt_sb 非空，将 sysfs_mount->mnt_sb->s_root 传给 parent，否则返回 return –EFAULT。执行 create_dir(kobj,parent, kobject_name(kobj),&dentry) （这里的 create_dir()是 fs\sysfs\dir.c 中的函数）在父目录 parent 下查找名字为内核对象 kobj 的内核名字 kobj->k_name 的子目录或文件对应的目录条目结构指针。如果找到的目录条目的索引节点指针是空的，为其分配并初始化一个新的索引节点与 sysfs_sb 关联，将该索引节点连接到系统在使用索引节点链表 inode_in_use 中，调用 init_dir()初始化该索引节点，再将该目录条目通过其 d_alias 连接到该索引节点的目录条目链表中。创建并初始化一个新的系统文件系统目录条目结构，将内核对象指针 kobj 作为该系统文件系统目录条目结构的要素数据指针，将该系统文件系统目录条目结构通过其 sibling 连接到父系统文件系统目录条目结构 parent->d_fsdata 的子系统文件系统目录条目结构链表 children 中，将该系统文件系统目录条目的目录条目指针指向找到的目录条目结构，类型设置为 SYSFS_DIR，再将该系统文件系统目录条目的计数器加 1 并传给找到的目录条目结构的 d_fsdata 指针，再将找到的目录条目结构的操作函数组指针指向 sysfs_dentry_ops。最后清除找到的目录条目标志字的 DCACHE_UNHASHED 标志位，再将其通过 d_hash 加入到 dentry_hashtable 链表中，dentry 指向这个找到的目录条目结构，将返回值传给 error，如果 error 等于 0，将 dentry 传给 kobj->dentry，返回 error。

int __init buses_init(void)函数执行过程：执行并返回 subsystem_register(&bus_subsys)。

int __init classes_init(void)函数执行过程：执行 subsystem_register(&class_subsys)将返回值传给 int retval。如果 retval 不等于 0，返回 retval。执行 subsystem_init(&class_obj_subsys)。如果 class_obj_subsys.kset.subsys 为空，将 class_obj_subsys 传给 class_obj_subsys.kset.subsys，返回 0。

int __init firmware_init(void) 函数执行过程：直接执行并返回 subsystem_register(&firmware_subsys)。

int __init platform_bus_init(void)函数执行过程：执行 device_register(&platform_bus)；执行并返回 bus_register(&platform_bus_type)。

int device_register(struct device *dev)函数执行过程：执行 int device_initialize(dev)初始化设备 dev 的内核对象成员变量以及设备的各种链表，执行 device_add(dev) 函数，添加设备到内核中，并返回。

void device_initialize(struct device *dev) 函数执行过程：执行 kobj_set_kset_s(dev, devices_subsys) 将 dev->kobj.kset 指向设备子系统结构的内核组件指针 devices_subsys.kset。执行 kobject_init(&dev->kobj) 初始化设备的内核对象成员结构变量 dev->kobj，初始化设备链表 dev->node，初始化设备子设备链表 dev->children，初始化设备驱动程序链表 dev->driver_list；初始化设备总线链表 dev->bus_list，初始化设备 DMA 内存池链表 dev->dma_pools。

void kobject_init(struct kobject * kobj)函数执行过程：将 kobj->kref->refcount 设置为 1，初始化 kobj->entry 链表，将 kobj->kset->kobj->kref->refcount 加 1 再将 kobj->kset 传回给 kobj->kset。

int device_add(struct device *dev)函数执行过程：将 struct device *parent 设置为 NULL；将 int error 设置成-EINVAL。将 dev->kobj->kref->refcount 加 1 再将 dev 返回给 dev，如果 dev 为空或者字符串 dev->bus_id 的长度为 0，跳转到“Error 标号”处。将 dev->parent->kobj->kref->refcount 加 1 再将 dev->parent 返回给 parent，打印("DEV: registering device: ID = '%s'\n", dev->bus_id)。执行 kobject_set_name(&dev->kobj, "%s", dev->bus_id) 将设备的总线 ID 号作为该设备的对象结构的名字字符串。如果 parent 非空，将 dev->kobj.parent 指向 parent->kobj。执行 kobject_add(&dev->kobj) 将对象添加到内核中并将返回值传给 error，如果 error 不等于 0，跳转到“Error 标号”处。执行 device_pm_add(dev)（略，该函数在没有配置 CONFIG_PM 选项的系统中直接返回 0）将设备 dev 加入到电源管理中去并返回给 error，如果 error 不等于 0，跳转到“PMError 标号”处。执行 bus_add_device(dev) 将设备通过 bus_list 连接到设备所在总线类型的设备套件的链表末尾，如果该设备的驱动没有定义，先调用驱动程序所属总线类型的匹配函数来匹配设备，如果匹配成功，再调用驱动的探测函数来探测设备，如果匹配探测成功，将设备 dev 通过 driver_list 连接到其驱动程序的所有适用设备链表末尾，为该设备（所属总线类型的设备）的每种属性结构与该设备结构本身分别创建并初始化一个新的系统文件系统目录条目结构，创建系统文件系统符号连接，将返回值传给 error。如果 error 不等于 0，跳转到

“BusError 标号”处。如果 parent 非空，将 dev 通过 dev->node 添加到 parent->children 链表的末尾。如果函数指针 platform_notify()非空（该函数指针只在 arch\arm\mach-ixp4xx\common-pci.c 中被赋了值），执行 platform_notify(dev)。“Done 标号”位置：put_device(dev)，返回 error。“BusError 标号”位置：执行 device_pm_remove(dev)（略，该函数在没有配置 CONFIG_PM 选项的系统中直接返回 0）将设备从电源管理中删除。“PMError 标号”位置：执行 kobject_del(&dev->kobj)调用/sbin/hotplug 将设备的对象 dev->kobj 从内核中删除，删除 dev->kobj，将 dev->kobj 从其所在的 entry 链表中删除，并初始化 dev->kobj.entry，最后释放 dev->kobj.kref。“Error 标号”位置：如果 parent 非空，执行 put_device(dev)，跳转到“Done 标号”处。

int kobject_set_name(struct kobject * kobj, const char * fmt, ...)函数执行过程：将 KOBJ_NAME_LEN（20）传给 int limit，将格式字符串 fmt 后第 1 个参数的地址传给 va_list args，执行 vsnprintf(kobj->name,limit,fmt,args)将 fmt 及后面的参数按格式打印到 kobj->name 中，打印长度为 limit，将返回值传给 need（使整个字符串的长度）。执行 va_end(args)，如果 need 小于 limit，将 kobj->name 传给 name，否则将 need+1 传给 limit，执行 kmalloc(limit,GFP_KERNEL)申请内存空间传给 name，如果 name 为空，将 error 设置为-ENOMEM，跳转到“Done 标号”处，执行 va_start(args,fmt)将 fmt 后面的第一个参数的地址传给 args，执行 vsnprintf(name,limit, fmt,args)，执行 va_end(args)，如果 need 大于等于 limit，执行 kfree(name)，将 error 设置为-EFAULT，跳转到“Done 标号”处。如果 kobj->k_name 非空且 kobj->k_name 不等于 kobj->name，执行 kfree(kobj->k_name)，将 name 传给 kobj->k_name。“Done 标号”位置：返回 error。

int bus_add_device(struct device * dev)函数执行过程：如果 dev->bus 指针非空，将 dev->bus->subsys.kset.kobj.kref.refcount 加 1 再将 dev->bus 传给 struct bus_type *bus，否则将 NULL 传给 bus；将 int error 设置为 0。如果 bus 非空，打印("bus %s: add device %s\n", bus->name, dev->bus_id)，将 dev 通过 dev->bus_list 连接到 dev->bus->devices.list 链表末尾，执行 device_attach(dev)，执行 device_add_attrs(bus, dev)，执行 sysfs_create_link(&bus->devices.kobj, &dev->kobj, dev->bus_id)，返回 error。

int device_attach(struct device * dev)函数执行过程：将 dev->bus 传给 struct bus_type * bus。如果 dev->driver 非空，执行 device_bind_driver(dev)，返回 1。如果 bus->match 函数指针非空（platform_bus_type.match 指向 platform_match()函数），针对 bus->drivers.list 链表中每个设备驱动结构 truct device_driver * drv 执行以下操作，执行 driver_probe_device(drv, dev)返回值传给 int error，如果 error 等于 0，返回 1，如果 error 不等于-ENODEV，打印(KERN_WARNING "%s: probe of %s failed with error %d\n", drv->name, dev->bus_id, error)，返回 0。

void device_bind_driver(struct device * dev)函数执行过程：打印("bound device '%s' to driver '%s'\n", dev->bus_id, dev->driver->name)，将 dev 通过 dev->drvier_list 连接到 dev->driver->devices 链表末尾。执行 sysfs_create_link(&dev->driver->kobj, &dev->kobj, kobject_name(&dev->kobj))，

执行 sysfs_create_link(&dev->kobj, &dev->driver->kobj, "driver")。

int driver_probe_device(struct device_driver * drv, struct device * dev)函数执行过程：如果 drv->bus->match 非空且 drv->bus->match(dev, drv)返回 0，返回-ENODEV。将 drv 传给 dev->driver；如果 drv->probe 非空，执行 drv->probe(dev)将返回值传给 int error。如果 error 不等于 0，将 dev->driver 设置为 NULL，返回 error。执行 device_bind_driver(dev)，返回 0。

static int platform_match(struct device * dev, struct device_driver * drv)函数执行过程：执行 container_of(dev, struct platform_device, dev)将 dev 所在的 struct platform_device 结构指针传给 struct platform_device *pdev。比较 drv->name 和 pdev->name，比较 BUS_ID_SIZE（20）个字符，如果比较结果是两个字符串相同，返回 1，否则返回 0。

static int device_add_attrs(struct bus_type * bus, struct device * dev)函数执行过程：将 int error 设置为 0。如果总线上设备属性指针 bus->dev_attrs 非空，那么对于该总线上设备的每种属性结构 bus->dev_attrs[i]进行如下操作：执行 device_create_file(dev,&bus->dev_attrs[i])将返回值传给 error，如果 error 不等于 0，跳转到“Err 标号”处。“Done 标号”位置：返回 error。“Err 标号”位置：对所有属性进行如下操作 device_remove_file(dev,&bus->dev_attrs[i])。跳转到“Done 标号”处。

int sysfs_create_file(struct kobject * kobj, const struct attribute * attr)函数执行过程：如果 kobj 为空或者 kobj->dentry 为空或者 attr 为空，执行并返回 sysfs_add_file(kobj->dentry, attr, SYSFS_KOBJ_ATTR) 创建并初始化一个新的系统文件系统目录条目结构，将属性结构指针 attr 作为该系统文件系统目录条目结构的要素数据指针，将该系统文件系统目录条目结构通过其 sibling 连接到对象的目录条目的文件系统数据结构 kobj->dentry->d_fsdata 的子系统文件系统目录条目结构链表 children 中，将该系统文件系统目录条目的目录条目指针设置为空，类型设置为系统文件系统对象属性 SYSFS_KOBJ_ATTR，模式字设置为普通文件标志。

int device_create_file(struct device * dev, struct device_attribute * attr)函数执行过程：将 error 设置为 0。如果 dev 非空，将 dev->kobj->kref->refcount 加 1，执行 sysfs_create_file(&dev->kobj, &attr->attr) 创建并初始化一个新的系统文件系统目录条目结构，将设备属性结构的属性结构指针&attr->attr 作为该系统文件系统目录条目结构的要素数据指针，将该系统文件系统目录条目结构通过其 sibling 连接到设备的对象的目录条目的文件系统数据结构(&dev->kobj)->dentry->d_fsdata 的子系统文件系统目录条目结构链表 children 中，将该系统文件系统目录条目的目录条目指针设置为空，类型设置为系统文件系统对象属性 SYSFS_KOBJ_ATTR，模式字设置为普通文件标志，将返回值传给 error，执行 put_device(dev)，返回 error。

void put_device(struct device * dev)函数执行过程：如果 dev 非空，执行 kobject_put (&dev->kobj)。

void device_remove_file(struct device * dev, struct device_attribute * attr)函数执行过程：如果dev 非空，将 dev->kobj->kref->refcount 加 1，执行 sysfs_remove_file(&dev->kobj, &attr->attr)，执行 put_device(dev)。

void sysfs_remove_file(struct kobject * kobj, const struct attribute * attr)函数执行过程：执行 sysfs_hash_and_remove(kobj->dentry,attr->name)。

int bus_register(struct bus_type * bus)函数执行过程：执行 kobject_set_name(&bus->subsys.kset.kobj, "%s", bus->name)。将 bus->name 打印到 bus->subsys.kset.kobj.name 和 bus->subsys.kset.kobj.k_name 中将返回值传给 int retval。如果 retval 不等于 0，跳转到“out 标号”处。执行 subsys_set_kset(bus, bus_subsys)将 bus_subsys.kset 传给 bus->subsys.kset.kobj.kset。执行 subsystem_register(&bus->subsys)注册 bus 总线的子系统结构将返回值传给 retval；如果 retval 不等于 0，跳转到“out 标号”处。执行 kobject_set_name(&bus->devices.kobj, "devices")将"devices"打印到 bus->devices.kobj.name 和 bus->devices.kobj.k_name 中。将 bus->subsys 传给 bus->devices.subsys，执行 kset_register(&bus->devices)注册总线的设备组件描述结构返回值传给 retval。如果 retval 不等于 0，跳转到“bus_devices_fail 标号”处。执行 kobject_set_name(&bus->drivers.kobj, "drivers")将"drivers"打印到 bus->drivers.kobj.name 和 bus->drivers.kobj.k_name 中。将 bus->subsys 传给 bus->drivers.subsys，将 ktype_driver 传给 bus->drivers.ktype，执行 kset_register(&bus->drivers)注册总线的驱动组件描述结构返回值传给 retval。如果 retval 不等于 0，跳转到“bus_drivers_fail 标号”处。执行 bus_add_attrs(bus)，打印("bus type '%s' registered\n", bus->name)，返回 0。“bus_drivers_fail 标号”位置：执行 kset_unregister(&bus->devices)。“bus_devices_fail 标号”位置：执行 subsystem_unregister (&bus->subsys)。“out 标号”位置：返回 retval。

static int bus_add_attrs(struct bus_type * bus)函数执行过程：将 int error 设置为 0。如果总线属性指针 bus->bus_attrs 非空，对该总线的各种属性结构 bus->bus_attrs[i]作如下操作：执行 bus_create_file(bus,&bus->bus_attrs[i])将返回值传给 error，如果 error 不等于 0，跳转到“Err 标号”处，“Done 标号”位置：返回 error。“Err 标号”位置：对该总线的各种属性结构 bus->bus_attrs[i]执行 bus_remove_file(bus,&bus->bus_attrs[i])，跳转到“Done 标号”处。

int bus_create_file(struct bus_type * bus, struct bus_attribute * attr)函数执行过程：如果 bus 非空，将 bus->subsys.kset.kobj.kref.refcount 加 1，执行 sysfs_create_file(&bus->subsys.kset.kobj, &attr->attr) 创建并初始化一个新的系统文件系统目录条目结构，将设备属性结构的属性结构指针&attr->attr 作为该系统文件系统目录条目结构的要素数据指针，将该系统文件系统目录条目结构通过其 sibling 连接到总线的子系统的内核组件的对象的目录条目的文件系统数据结构(&bus->subsys.kset.kobj)->dentry->d_fsdata 的子系统文件系统目录条目结构链表 children 中，将该系统文件系统目录条目的目录条目指针设置为空，类型设置为系统文件系统对象属性 SYSFS_KOBJ_ATTR，模式字设置为普通文件标志，将返回值传给 error，执行

put_bus(bus)->subsys_put(&bus->subsys)->kset_put(&(&bus->subsys)->kset->kobject_put(&(&(&bus->subsys)->kset)->kobj))。如果 bus 为空，将 error 设置为-EINVAL，返回 error。

void bus_remove_file(struct bus_type * bus, struct bus_attribute * attr)函数执行过程：如果 bus 非空，将 bus->subsys.kset.kobj.kref.refcount 加 1，执行 sysfs_remove_file(&bus->subsys.kset.kobj, &attr->attr)，执行 put_bus(bus)。

int __init system_bus_init(void)函数执行过程：将 system_subsys.kset.kobj.parent 指向 devices_subsys.kset.kobj，执行并返回 subsystem_register(&system_subsys)。

int __init cpu_dev_init(void)函数执行过程：执行并返回 sysdev_class_register(&cpu_sysdev_class)。

int sysdev_class_register(struct sysdev_class * cls)函数执行过程：打印("Registering sysdev class '%s'\n", kobject_name(&cls->kset.kobj))。初始化该系统设备类的驱动链表 cls->drivers；将 cls->kset.subsys 指向 system_subsys，将 cls->kset.kobj.kset 指向 system_subsys.kset，执行并返回 kset_register(&cls->kset) 初始化该系统设备类结构的内核组件的 kobj 成员结构和链表，将系统设备类结构的内核组件的对象添加到内核中。

int kset_register(struct kset * k)函数执行过程：执行 kset_init(k)函数初始化内核套件描述结 k。将 k.kobj.kref.refcount 设置为 1，将 k.kobj.entry 链表初始化为空链表，将 k.kobj.kref.refcount 加 1，将 k.list 链表初始化为空链表，执行并返回 kset_add(k)。

void __init sysctl_init(void)函数执行过程：该函数在没有配置 CONFIG_PROC_FS 选项的情况下是空函数，什么也不做。在配置了 CONFIG_PROC_FS 选项的系统中执行过程如下：执行 register_proc_table(root_table, proc_sys_root)，执行 init_irq_proc()。

static void register_proc_table(ctl_table * table, struct proc_dir_entry *root)函数执行过程：for (; table->ctl_name; table++)循环：如果 table->procname 为空，查看下一个系统控制 proc 目录/文件描述结构。如果 table->proc_handler（只有系统控制的 proc 文件该指针才有定义，目录没有定义）指针和 table->child 指针都为空，打印不能注册信息，查看下一个系统控制 proc 目录/文件描述结构。将 strlen(table->procname)（不含结束字符）传给 int len，将 table->mode 传给 mode_t mode，将 struct proc_dir_entry *de 设置为 NULL。如果 table->proc_handler 非空，说明是一个 proc 文件，将 mode 或上 S_IFREG 标志，否则，将 mode 或上 S_IFDIR 标志，再执行 for (de = root->subdir; de; de = de->next)循环：如果(proc_match(len, table->procname, de)不等于 0 即找到了与 table->procname 同名的 proc 目录，退出本 for 循环。如果在/proc/sys/子目录中没有找到与该系统控制表结构 proc 名字同名的子目录，执行 create_proc_entry(table->procname, mode, root)在/proc/sys/目录下创建一个新的与该系统控制表结构 proc 名字同名的子目录，如果创建失败，查看下一个系统控制 proc 目录/文件描述结构，将 table 传给 de->data，如果 table->proc_handler

即是一个 proc 文件，将 de->proc_fops 指向 proc_sys_file_operations。将 de 传给 table->de。如果 de->mode 设置了 S_IFDIR 即是一个 proc 目录，那么执行 register_proc_table(table->child, de)继续注册其子目录或子文件。

void init_irq_proc(void)函数执行过程：执行 proc_mkdir("irq", NULL)创建/proc/irq 目录的 proc 目录结构并将返回值传给全局结构指针 static struct proc_dir_entry *。如果 root_irq_dir 为空，返回。for (i = 0; i < NR_IRQS; i++)循环：register_irq_proc(i)为系统每个 irq 中断号创建/proc/irq/n 文件对应的 proc 目录结构。

void register_irq_proc(unsigned int irq)函数执行过程：声明字符数组 char name [MAX_NAMELEN]；如果 root_irq_dir 为空或者 irq_desc[irq].handler 等于&no_irq_type 或者 irq_dir [irq]非空，返回。将 name 空间全部清 0，打印“irq 号”。执行 proc_mkdir(name, root_irq_dir)将返回值传给 irq_dir[irq]。如果系统配置了 CONFIG_SMP 选项即支持多处理器，那么执行 create_proc_entry("smp_affinity", 0600, irq_dir[irq])将返回值传给 struct proc_dir_entry *entry，如果 entry 非空，将 entry->nlink 设置为 1，将(void *)(long)irq 传给 entry->data，将 irq_affinity_read_proc 传给 entry->read_proc，将 irq_affinity_write_proc 传给 entry->write_proc，将 entry 传给 smp_affinity_entry[irq]，返回。

void __init sock_init(void)函数执行过程：将系统支持的对多 32 种协议族描述结构指针 net_families[n（n 等于 0~31）]都设置为 NULL。执行 sk_init()函数创建 struct sock 结构高速缓存描述结构变量 sk_cachep，根据系统中物理内存总页数调整 sysctl_wmem_max、sysctl_rmem_max、sysctl_wmem_default、sysctl_rmem_ default 等网络部分系统控制参数。如果定义了 SLAB_SKB（在 include\linux\skbuff.h 中有定义），那么执行 skb_init()创建 struct sk_buff 结构高速缓存描述结构变量 skbuff_head_cache。执行 init_inodecache()（该函数在 linux 中有许多定义，都是静态函数，只在本文件内有效，这里指的是 net\socket.c 中的函数）。执行 register_filesystem(&sock_fs_type) 将套接字在网络层表述文件系统类型结构 sock_fs_type 注册到整个系统的文件系统类型链表 file_systems 末尾，如果系统配置了 CONFIG_NETFILTER 选项，执行 netfilter_init()。

void __init sk_init(void)函数执行过程：执行 kmem_cache_create("sock", sizeof(struct sock), 0, SLAB_HWCACHE_ALIGN, NULL, NULL)为网络 sock 结构 struct sock 创建高速缓存内存描述结构 kmem_cache_t 变量，为该变量命名为“sock”，按硬件 cache 大小对齐，其对象的构造函数和析构函数指针空，将创建的 kmem_cache_t 结构变量地址传给全局指针 sk_cachep。如果 sk_cachep 为空，打印(KERN_CRIT "sk_init: Cannot create sock SLAB cache!")。如果系统中全部物理内存的总页数 num_physpages 小于等于 4096 即 16MB，将 sysctl_wmem_max、sysctl_rmem_max、sysctl_wmem_default、sysctl_rmem_default 都设置为 32767，否则将 sysctl_wmem_max 和 sysctl_rmem_max 设置为 131071。

void __init skb_init(void)函数执行过程：执行 kmem_cache_create("skbuff_head_cache", sizeof(struct sk_buff), 0, SLAB_HWCACHE_ALIGN, NULL, NULL) 为网络套接字缓冲区描述结构 struct sk_buff 创建高速缓存内存描述结构 kmem_cache_t 变量，为该变量命名为“skbuff_head_cache”，按硬件 cache 大小对齐，其对象的构造函数和析构函数指针空，将创建的 kmem_cache_t 结构变量地址传给全局指针 skbuff_head_cache。如果 skbuff_head_cache 为空，系统 panic("cannot create skbuff cache")。

void __init netfilter_init(void)函数执行过程：将 nf_hooks[m(0~31)][n(0~7)]全部初始化为空链表。

static void __init do_initcalls(void)函数执行过程：将 preempt_count()（即 current_thread_info()->preempt_count）传给 int count，然后对连接在&__initcall_start 到&__initcall_end 之间的所有初始化函数指针 call()作如下操作：如果 initcall_debug（初始为 0，可由"initcall_debug"命令设置），打印(KERN_DEBUG "Calling initcall 0x%p", *call)，打印(": %s()", (unsigned long) *call)，执行(*call)()，将 char *msg 设置为 NULL，如果 preempt_count()不等于 count，将 msg 设置为"preemption imbalance"，将 count 传给 preempt_count()，如果 irqs_disabled()（即当前程序状态寄存器 CPSR 设置了 PSR_I_BIT 位，irq 中断被禁止），那么将 msg 设置为"disabled interrupts"，执行 local_irq_enable()使能 IRQ 中断，如果 msg 非空，打印("error in initcall at 0x%p: ""returned with %s\n", *call, msg)，执行 flush_scheduled_work()。

void flush_scheduled_work(void)函数执行过程：直接执行 flush_workqueue(keventd_wq)。

17.6.8 sys_access()函数

调用 sys_access((const char __user *) "/init", 0)函数，如果返回 0 即能够正确访问前面 initramfs 或 initrd 文件系统中的"/init"用户应用程序，那么将"/init"传给 execute_command 字符串指针，否则调用 prepare_namespace()函数。

asmlinkage long sys_access(const char __user * filename, int mode)函数执行过程：如果 mode 与上~S_IRWXO 不等于 0，返回-EINVAL。将 current->fsuid 传给 int old_fsuid，将 current->fsgid 传给 old_fsgid，将 current->cap_effective 传给 kernel_cap_t old_cap，将 current->uid 传给 current->fsuid，将 current->gid 传给 current->fsgid。如果 current->uid 不等于 0，将 current->cap_effective 清 0，否则将 current->cap_permitted 传给 current->cap_effective。执行 __user_walk(filename, LOOKUP_FOLLOW|LOOKUP_ACCESS, &nd)并将返回值传给 int res。如果 res 等于 0，则执行正确，执行 permission(nd.dentry->d_inode, mode, &nd) 检查当前任务是否有权限访问刚找到的目录条目的索引节点 nd.dentry->d_inode 并将返回值传给 res，如果 res 等于 0 即由访问权限且 mode 设置了 S_IWOTH 标志即其他用户可以写该文件且 nd.dentry->d_inode->i_sb->s_flags 设置了 MS_RDONLY 即该文件所属文件系统只读且这个

nd.dentry->d_inode 对应的不是字符设备和块设备和 FIFO 和 SOCK 等特殊文件，那么将-EROFS 传给 res，执行 path_release(&nd)。将 old_fsuid 传回 current->fsuid，将 old_fsgid 传回 current->fsgid，将 old_cap 传回 current->cap_effective，返回 res。

prepare_namespace()函数执行过程：执行 mount_devfs()，执行 md_run_setup()，如果 saved_root_name[0] 不等于 0，将 saved_root_name 传给 root_device_name；将 name_to_dev_t(root_device_name)传给 ROOT_DEV。如果 root_device_name 前 5 个字符是“/dev/”，将 root_device_name 加上 5 指向“/dev/”后面的字符。如果 MAJOR(ROOT_DEV)等于 FLOPPY_MAJOR，将 is_floppy 设置为 1 否则设置为 0。如果执行 initrd_load()不等于 0，跳转到“out 标号”处。如果 is_floppy 不等于 0 且 rd_doload 不等于 0 且 rd_load_disk(0)也不等于 0，将 Root_RAM0 传给 ROOT_DEV。执行 mount_root()。“out 标号”位置：执行 umount_devfs("/dev")取消/dev 上 devfs 文件系统的挂接，执行 sys_mount(".", "/", NULL, MS_MOVE, NULL)将挂接在当前进程的当前目录下的文件系统（这里"."就是刚刚挂接了根文件系统设备"/root"目录）移动到"/"目录上，即将"/root"上挂接的根文件系统一到"/"目录上，由此看出，挂接根文件系统事实上是先将根文件系统挂接到"/root"目录上然后再移到"/"目录上，执行 sys_chroot(".")将当前目录作为根目录，执行 security_sb_post_mountroot()，执行 mount_devfs_fs()。

dev_t __init name_to_dev_t(char *name)函数执行过程：将 dev_t res 设置为 0。如果配置了 CONFIG_SYSFS 选项即支持系统文件系统 sysfs，执行 sys_mkdir("/sys", 0700)创建/sys 目录将返回值传给 int mkdir_err，如果执行 sys_mount("sysfs", "/sys", "sysfs", 0, NULL)将 sysfs 文件系统挂接到/sys 目录返回值小于 0，跳转到“out 标号”处（宏 CONFIG_SYSFS 条件判断结束）。如果 name 的前 5 个字符不是"/dev/"（也就是说不是以“/dev/…”名字如/dev/ram0 来表示根文件系统设备），那么如果执行 sscanf(name, "%u:%u", &maj, &min)返回值等于 2（也就是说以“主设备号：次设备号”方式如 01：00(/dev/ram0)来表示根文件系统设备），那么将 MKDEV(maj, min)传给 res，如果 maj 不等于 MAJOR(res)或者 min 不等于 MINOR(res)即如果主次设备号有超出范围的（主设备号 12 位次设备号 20 位），跳转到“fail 标号”处。如果 sscanf(name, "%u:%u", &maj, &min)不等于 2，则执行 new_decode_dev(simple_strtoul(name, &(char *)p, 16))（也就是说以老的设备编号方式如 0x00100000(/dev/ram0)来表示根文件系统设备）将返回之传给 res，如果*p 非空，跳转到“fail 标号”处，也就是说设备号后面不能有其他字符，必须是结束字符，跳转到“done 标号”处。如果 name 的前 5 个字符是"/dev/"，将 name 加 5 指向“/dev”后面的字符串。将 Root_NFS 传给 res。如果“/dev”后面的字符串是“nfs”，跳转到“done 标号”处。将 Root_RAM0 传给 res，如果“/dev”后面的字符串是“ram”，跳转到“done 标号”处。如果“/dev”后面的字符串长度大于 31，跳转到“fail 标号”处。将 name 复制到 char s[32]中，将 s 中的'/'字符改成'!'字符。执行 try_name(s, 0)将返回值传给 res。如果 res 不等于 0，跳转到“done 标号”处。如果 p 位置前面是数字字符（16 进制），将 p 往前挪到非数字字符前面；如果 p 等

于 s 或者*p 等于 0 或*p 等于'0'，跳转到“fail 标号”处。将 simple_strtoul(p, NULL, 10)传给 int part，将*p 设置为结束字符'\0'，执行 try_name(s, part)传给 res，如果 res 不等于 0，跳转到“done 标号”处。如果 p 小于 s+2 或者 p[-2]等于 0 或者 p[-1]不等于'p'， 跳转到“fail 标号”处。将 p[-1]设置为结束字符'\0'；再次执行 try_name(s, part)传给 res。“done 标号”位置：如果配置了 CONFIG_SYSFS 选项执行 sys_umount("/sys", 0)。“out 标号”位置：如果 mkdir_err 等于 0，执行 sys_rmdir("/sys")删除“/sys”目录（宏 CONFIG_SYSFS 条件判断结束），返回 res。“fail 标号”位置：将 res 设置为 0，跳转到“done 标号”处。

void __init mount_devfs(void)函数执行过程：在没有配置 CONFIG_DEVFS_FS 选项的系统中该函数是空函数，什么也不做，在配置了 CONFIG_DEVFS_FS 选项的系统中，直接执行并返回 sys_mount("devfs", "/dev", "devfs", 0, NULL)。

asmlinkage long sys_mount(char __user * dev_name, char __user * dir_name, char __user * type, unsigned long flags, void __user * data)函数执行过程：执行 copy_mount_options (type, &（unsigned long）type_page) 申请一页内核内存空间，将用户空间的类型字符串 type 复制到申请到的内核内存页地址，type_page 指向该内核内存页地址，将返回值传给 int retval，如果 retval 小于 0 即操作失败，返回 retval。执行 getname(dir_name)将返回值传给 char *dir_page，将 PTR_ERR(dir_page)传给 retval，如果 dir_page 指针错误，跳转到“out1 标号”处。执行 copy_mount_options (dev_name, &dev_page) 申请一页内核内存空间，将用户空间的设备名字字符串 dev_name 复制到申请到的内核内存页地址，dev_page 指向该内核内存页地址，将返回值传给 int retval，如果 retval 小于 0 即操作失败，跳转到“out2 标号”处。执行 copy_mount_options (data, &data_page) 申请一页内核内存空间，将用户空间的数据 data 复制到申请到的内核内存页地址，data_page 指向该内核内存页地址，将返回值传给 int retval，如果 retval 小于 0 即操作失败，跳转到“out3 标号”处。执行 lock_kernel()，执行 do_mount((char*)dev_page, dir_page, (char*)type_page, flags, (void*)data_page)将返回值传给 retval。执行 unlock_kernel()。“out3 标号位置”：执行 free_page(dev_page)。“out2 标号位置”：执行 putname(dir_page)。“out1 标号位置”：执行 free_page(type_page)，返回 retval。

long do_mount(char * dev_name, char * dir_name, char *type_page, unsigned long flags, void *data_page)函数说明：如果 flags 与上 MS_MGC_MSK 等于 MS_MGC_VAL 即设置了老的挂接标志，将 flags 与上~MS_MGC_MSK 清除老的挂接标志。如果 dir_name 指针为空或者 dir_name 指针所指位置没有字符串直接就是结束字符或者 dir_name 所指的整页内存都没有 0 即没有结束字符字符串过长，返回无效错误-EINVAL。如果 dev_name 指针非空即 dev_name 字符串存在但是 dev_name 所指的整页内存都没有 0 即没有结束字符，字符串过长，返回无效错误-EINVAL。如果 data_page 非空，将 data_page 所指内存页的最后字节设置为 0。如果 flags 设置了忽略 suid 和 sgid 位标志 MS_NOSUID，将 mnt_flags 或上 MNT_NOSUID。如果 flags 设置了不允许访问设备特殊文件标志位 MS_NODEV，将 mnt_flags 或上 MNT_NODEV。如果 flags 设置了不允许

程序执行标志位 MS_NOEXEC，将 mnt_flags 或上 MNT_NOEXEC，清除掉 flags 中的 MS_NOSUID|MS_NOEXEC|MS_NODEV|MS_ACTIVE 标志，执行 path_lookup(dir_name, LOOKUP_FOLLOW, &nd)找到挂接目录对应的dentry和inode并将返回值传给retval。如果retval 不等于 0 即没有找到，返回 retval。执行 security_sb_mount(dev_name, &nd, type_page, flags, data_page)将返回值传给 retval。如果 retval 不等于 0，跳转到“dput_out 标号”处。如果 flags 设置了重新挂接标志位 MS_REMOUNT，执行 do_remount(&nd, flags & ~MS_REMOUNT, mnt_flags, data_page)。如果 flags 设置了 MS_BIND 标志位，执行 do_loopback(&nd, dev_name, flags & MS_REC)。如果 flags 设置了 MS_MOVE 标志，执行 do_move_mount(&nd, dev_name)，否则执行 do_new_mount(&nd, type_page, flags, mnt_flags, dev_name, data_page)。“dput_out 标号”位置：执行 path_release(&nd)，返回 retval。

static int do_new_mount(struct nameidata *nd, char *type, int flags, int mnt_flags, char *name, void *data)函数执行过程：如果 type 为空或者 type 所指内存页中没有字符串结束字符 0 即字符串过长，返回无效错误-EINVAL。如果当前进程的有效能力字 current->cap_effective 没有设置 CAP_SYS_ADMIN 标志即当前进程没有管理者权限，返回权限错误-EPERM。执行 do_kern_mount(type, flags, name, data)将返回值传给 struct vfsmount *mnt。如果 mnt 指针错误，返回 PTR_ERR(mnt)，执行并返回 do_add_mount(mnt, nd, mnt_flags, NULL)。

int do_add_mount(struct vfsmount *newmnt, struct nameidata *nd, int mnt_flags, struct list_head *fslist)函数执行过程：执行 while(d_mountpoint(nd->dentry) && follow_down(&nd->mnt, &nd->dentry))循环，即只要 nd->dentry->d_mounted 不等于 0 且 follow_down(&nd->mnt, &nd->dentry)执行结果不等于 0 就一直循环执行这两个函数（因为在 Linux 中同一目录可以挂接多个文件系统，但用户只能看见最后挂接到该目录的文件系统，如果某个目录已经挂接了文件系统要继续在该目录挂接文件系统事实就是挂接在最后挂接到该目录的文件系统的 mnt->mnt_root 目录下，执行 unmount 可以依次看到挂接到该目录的所有文件系统）。将-EINVAL 传给 int err。如果执行 check_mnt(nd->mnt)结果为 0，跳转到“unlock 标号”处，将-EBUSY 传给 err，如果 nd->mnt->mnt_sb 等于 newmnt->mnt_sb 并且 nd->mnt->mnt_root 等于 nd->dentry，跳转到“unlock 标号”处（因为不能在同一目录下挂接同一个超级块，即一个文件系统的具体实现）。将-EINVAL 传给 err。如果 newmnt->mnt_root->d_inode->i_mode 设置了 S_IFLNK 标志即该挂接结构的根目录是一个符号连接，跳转到“unlock 标号”处。将 mnt_flags 传给 newmnt->mnt_flags；执行 graft_tree(newmnt, nd)将返回值传给 err。如果 err 等于 0 且 fslist 非空，将 newmnt 通过 newmnt->mnt_fslink 链接到 fslist 链表的末尾。“unlock 标号”位置：执行 mntput(newmnt)，返回 err。

int follow_down(struct vfsmount **mnt, struct dentry **dentry)函数执行过程，直接执行返回 __follow_down(mnt,dentry)。

static inline int __follow_down(struct vfsmount **mnt, struct dentry **dentry)函数执行过程：执行 lookup_mnt(*mnt, *dentry)在 mount_hashtable+hash(mnt,dentry)链表中找到挂接点目录 mnt_mountpoint 就是 dentry 的 mnt 的子挂接结构，将返回值传给 struct vfsmount *mounted。如果 mounted 非空，执行 mntput(*mnt)，将 mounted 传给*mnt，执行 dput(*dentry)，将 mounted->mnt_root->mnt_count 加 1 然后将 mounted->mnt_root 传给*dentry，返回 1。如果 mounted 空，返回 0。

static inline int check_mnt(struct vfsmount *mnt 函数执行过程：如果 mnt->mnt_namespace 等于 current->namespace 返回 1，否则返回 0。

static int graft_tree(struct vfsmount *mnt, struct nameidata *nd)函数执行过程：如果 mnt->mnt_sb->s_flags 设置了 MS_NOUSER 标志（rootfs 文件系统、tmpfs_fs_type 文件系统以及以 get_sb_pseudo()函数作为获取超级块函数的文件系统（如 pfm_fs_type、bd_type、eventpoll_fs_type、pipe_fs_type、futex_fs_type、sock_fs_type 等）的超级块的 s_flags 设置了 MS_NOUSER 标志，从这里代码看这些文件系统是不能挂接到某个目录上的），返回-EINVAL。如果 S_ISDIR(nd->dentry->d_inode->i_mode)不等于 S_ISDIR(mnt->mnt_root->d_inode->i_mode)返回-ENOTDIR。将-ENOENT 传给 int err。如果 nd->dentry->d_inode->i_flags 设置了 S_DEAD 标志位即该目录虽然已经打开单已经不能使用，跳转到“ount_unlock 标号”处。执行 security_sb_check_sb(mnt, nd)将返回值传给 err，如果 err 不等于 0，跳转到“ount_unlock 标号”处。将-ENOENT 传给 err，如果 nd->dentry->d_parent 就是其自身即该目录是根目录或者 nd->dentry->d_flags 没有设置 DCACHE_UNHASHED 标志位即该目录已经连接到了 dentry_hashtable 表的某个 hash 链表中，执行 attach_mnt(mnt, nd)，将 struct list_head head 连接到 mnt->mnt_list 链表的末尾，再执行 list_splice(&head, current->namespace->list.prev)将 mnt->mnt_list 链表与 current->namespace->list 链表黏合起来，把 mnt->mnt_list 链表放在 current->namespace->list 链表的开始位置，执行 mntget(mnt)，将 err 设置为 0。“ount_unlock 标号”位置：如果 err 等于 0，执行 security_sb_post_addmount(mnt, nd)，返回 err。

static void attach_mnt(struct vfsmount *mnt, struct nameidata *nd)函数执行过程：将 nd->mnt->mnt_count 加 1 然后将 nd->mnt 传给 mnt->mnt_parent，将 nd->dentry->d_count 加 1 然后将 nd->dentry 传给 mnt->mnt_mountpoint，将 mnt 通过 mnt->mnt_hash 连接到 mount_hashtable+hash(nd->mnt, nd->dentry)链表的开始，这里的 hash()函数是 fs\namespace.c 中的 static inline unsigned long hash(struct vfsmount *mnt, struct dentry *dentry)函数，将 mnt 通过 mnt->mnt_child 连接到其父挂接结构的子挂接链表 nd->mnt->mnt_mounts 的末尾，将该目录的挂接计数器 nd->dentry->d_mounted 加 1。

static int do_move_mount(struct nameidata *nd, char *old_name)执行过程：如果当前任务的有效能力标志字 current->cap_effective 没有设置管理者权限标志位 CAP_SYS_ADMIN，返回

-EPERM。如果 old_name 为空或者 old_name 字符长度为 0，返回-EINVAL。执行 path_lookup(old_name, LOOKUP_FOLLOW, &(struct nameidata old_nd)old_nd)将返回值传给 err，如果 err 不等于 0，返回 err。执行 while(d_mountpoint(nd->dentry) && follow_down(&nd->mnt, &nd->dentry)) 循环。将 err 设置为-EINVAL。如果 nd->mnt->mnt_namespace 或 old_nd.mnt->mnt_namespac 不等于当前任务的名字空间指针 current->namespace，跳转到“out 标号”处，将-ENOENT 传给 err。如果 nd->dentry->d_inode ->i_flags 设置了 S_DEAD 标志，即 nd 的目录条目的索引节点对应的是一个已经离开但仍然打开的目录，跳转到“out1 标号”处。如果 nd->dentry 不是系统的根目录（即其父目录就是其自身）但 nd->dentry->d_flags 设置了 DCACHE_UNHASHED 标志位，即尚未加入到 dentry_hashtable 链表中，跳转到“out2 标号”处，将-EINVAL 传给 err。如果 old_nd.dentry 不等于 old_nd.mnt->mnt_root，跳转到“out2 标号”处。如果 old_nd.mnt 等于 old_nd.mnt->mnt_parent，跳转到“out2 标号”处。如果(S_ISDIR(nd->dentry->d_inode->i_mode)不等于 old_nd.dentry->d_inode->i_mode，跳转到“out2 标号”处，将 err 设置为-ELOOP。如果 nd->mnt 的某几个父挂接结构就是 old_nd.mnt，跳转到“out2 标号”处，将 err 设置为 0。执行 detach_mnt(old_nd.mnt, &parent_nd) 把 old_nd 的挂接结构变量的相应数据保存到父名字索引数据结构 parent_nd 中，再处理掉该挂接结构变量。再执行 attach_mnt(old_nd.mnt, nd)将 nd->mnt 作为 old_nd.mnt 的父挂接结构变量，nd->dentry 作为 old_nd.mnt 的挂接点目录条目，将 old_nd.mnt 连接到 mount_hashtable 链表中，将 old_nd.mnt 连接到 nd->mnt 的子挂接结构变量链表末尾。将 old_nd.mnt 从其所在的 mnt_fslink 链表中删除并初始化 mnt_fslink。“out2 标号”位置：执行 spin_unlock(&vfsmount_lock) 将虚拟文件系统挂接锁解锁；“out1 标号”位置：执行 up(&nd->dentry->d_inode->i_sem)将名字索引数据结构 nd 的目录条目的索引节点的信号量计数器增加；“out 标号”位置：如果 err 等于 0，执行 path_release(&parent_nd)，执行 path_release(&old_nd)，返回 err。

int __init initrd_load(void)函数执行过程：该函数在没有配置 CONFIG_BLK_DEV_INITRD 选项的系统中直接返回 0。在配置了 CONFIG_BLK_DEV_INITRD 选项的系统中执行过程如下：如果 mount_initrd 不等于 0（声明时默认设置为 1；该变量由"noinitrd"设置，如果命令行中设置了"noinitrd"命令，则将 mount_initrd 设置为 0），执行 create_dev("/dev/ram", Root_RAM0, NULL)创建/dev/ram 块设备节点，设备号为 Root_RAM0 即 01:00 或 0x00100000，权限为只有拥有者用户可以读写，执行 rd_load_image("/initrd.image")，如果执行结果不等于 0 即装载 RAM disk 正确但 ROOT_DEV 不等于 Root_RAM0，执行 sys_unlink("/initrd.image")，执行 handle_initrd()，返回 1，如果装载 RAM disk 正确且 ROOT_DEV 等于 Root_RAM0，执行 sys_unlink("/initrd.image")，返回 0。

int __init rd_load_image(char *from)函数执行过程：执行 sys_open("/dev/ram", O_RDWR, 0)按读写方式打开设备"/dev/ram"并僵局并返回给 int out_fd。如果 out_fd 小于 0，跳转到“out 标号”处。执行 sys_open(from, O_RDONLY, 0)按只读方式打开 from 所指文件将其句柄传给 int

in_fd。如果 in_fd 小于 0，跳转到“noclose_input 标号”处。执行 identify_ramdisk_image(in_fd, rd_image_start)解析 from 所指文件头辨认其所属文件系统类型，将返回之传给 int nblocks。如果 nblocks 小于 0，跳转到“done 标号”处。如果 nblocks 等于 0 说明 from 是压缩文件，如果定义了 BUILD_CRAMDISK，执行 crd_load(in_fd, out_fd)解压缩 in_fd 文件并装载到 out_fd 中，如果返回 0 即解压缩装载成功，跳转到“successful_load 标号”处，如果没有定义 BUILD_CRAMDISK，打印("RAMDISK: Kernel does not support compressed RAM disk images\n")，跳转到“done 标号”处。执行 sys_ioctl(in_fd, BLKGETSIZE, (unsigned long)&devblocks)读取/dev/ram0 设备的块数到 devblocks 中，如果返回值小于 0 即读取错误，将 devblocks 设置为 0，否则将右移 1 位即除以 2，因为块设备的块大小都是 512B 的，要将其转换成文件系统的块大小 BLOCK_SIZE（1024B）就须将块设备的块数除以 2。如果 from 所指文件名就是"/initrd.image"，将 nblocks 传给 devblocks。如果 devblocks 等于 0，打印("RAMDISK: could not determine device size\n")，跳转到“done 标号”处。执行 kmalloc(BLOCK_SIZE, GFP_KERNEL)申请 1024 字节内存空间传给 char *buf。如果 buf 等于 0，打印("RAMDISK: could not allocate buffer\n")，跳转到“done 标号”处。打印("RAMDISK: Loading %dKiB [%ld disk%s] into ram disk... ", nblocks, ((nblocks-1)/devblocks)+1, nblocks>devblocks ? "s" : "")。然后依次执行 nblocks 次 sys_read(in_fd, buf, BLOCK_SIZE)和 sys_write(out_fd, buf, BLOCK_SIZE)将 in_fd 中的数据全部写入 out_fd 中。最后打印("done disk #%d.\n",nblocks)。将 unsigned short rotate 设置为 0，执行 sys_close(in_fd)关闭 in_fd，如果执行结果不等于 0 即不惯比失败，跳转到“noclose_input 标号”处。执行 change_floppy("disk #%d", disk) 打开 in_fd 文件在下一个磁盘上的数据，执行 sys_open(from, O_RDONLY, 0)再次以只读方式打开 from 文件将返回值传给 in_fd。如果 in_fd 小于 0，打印("Error opening disk.\n")，跳转到“noclose_input 标号”处，打印("Loading disk #%d... ", disk)，打印"done.\n"。“noclose_input 标号”位置：将 res 设置为 1。“done 标号”位置：执行 sys_close(in_fd)。“noclose_input”标号位置：执行 sys_close(out_fd)。“out 标号”位置：执行 kfree(buf)，执行 sys_unlink("/dev/ram")，返回 res。

static int __init identify_ramdisk_image(int fd, int start_block)函数执行过程：将 const int size 设置为 512，执行 kmalloc(size, GFP_KERNEL)申请内存将返回地址传给 unsigned char *buf。如果 buf 等于 0，返回-1。将 struct minix_super_block *minixsb、struct ext2_super_block *ext2sb、struct romfs_super_block *romfsb、struct cramfs_super *cramfsb 都指向 buf 所指位置。将 buf 所指 size 大小空间全部设置为 0xe5。执行 sys_lseek(fd, start_block * BLOCK_SIZE, 0)将文件头指针指向文件的 start_block * BLOCK_SIZE（文件系统的块大小，等于 1024B）位置，执行 sys_read(fd, buf, size)从 from 文件的 start_block * BLOCK_SIZE 位置读取 size（512B 即只读了半块）大小的数据到 buf 中。如果 buf[0]等于 037（0x1F）并且 buf[1]等于 0213（0x8B）或 0236（0x9E）说明 from 是一个 gzip 压缩文件，打印("RAMDISK: Compressed image found at block %d\n"start_block)，将 nblocks 设置为 0，跳转到“done 标号”处。如果 romfsb->word0 等于

ROMSB_WORD0 并且 romfsb->word1 等于 ROMSB_WORD1 说明 from 是属于 romfs 文件系统类型，打印("RAMDISK: romfs filesystem found at block %d\n", start_block)，将(ntohl(romfsb->size)+BLOCK_SIZE-1)>>BLOCK_SIZE_BITS 传给 nblocks，跳转到“done 标号”处。如果 cramfsb->magic 等于 CRAMFS_MAGIC 说明 from 是属于 cramfs 文件系统类型，打印("RAMDISK: cramfs filesystem found at block %d\n", start_block)，将(cramfsb->size + BLOCK_SIZE - 1) >> BLOCK_SIZE_BITS 传给 nblocks，跳转到“done 标号”处。执行 sys_lseek(fd, (start_block+1) * BLOCK_SIZE, 0)将 from 文件指针指向下一个 block 即(start_block+1) * BLOCK_SIZE 位置，执行 sys_read(fd, buf, size)再次从第二个 block 中读取 size（512B 即只读了半块）大小的数据到 buf 中。如果 minixsb->s_magic 等于 MINIX_SUPER_MAGIC 或者 minixsb->s_magic 等于 MINIX_SUPER_MAGIC2 说明 from 属于 minixfs 文件系统，打印("RAMDISK: Minix filesystem found at block %d\n",start_block)，将 minixsb->s_nzones << minixsb->s_log_zone_size 传给 nblocks，跳转到“done 标号”处。如果 ext2sb->s_magic 等于 cpu_to_le16(EXT2_SUPER_MAGIC 说明 from 属于 ext2 文件系统，打印("RAMDISK: ext2 filesystem found at block %d\n", start_block)，将 le32_to_cpu(ext2sb->s_blocks_count) << le32_to_cpu(ext2sb->s_log_block_size)传给 nblocks，跳转到“done 标号”处。打印("RAMDISK: Couldn't find valid RAM disk image starting at %d.\n", start_block)。“done 标号”位置：执行 sys_lseek(fd, start_block * BLOCK_SIZE, 0)将 from 文件指针指回到 start_block * BLOCK_SIZE 位置。执行 kfree(buf)释放 buf 内存空间，返回 nblocks。从这个函数的执行看出，所有实际的文件系统都有自己的文件组织形式，这种组织形式都由一个名叫 struct xxx_super_block（如 struct minix_super_block、struct ext2_super_block、struct romfs_super_block、struct cramfs_super）的结构来描述，这个结构对应的数据一般放在文件系统存储介质或文件系统映像文件开始的若干块（1 块等于 BLOCK_SIZE=1024B），可以通过解析文件系统映像文件的开始若干块中的 struct xxx_super_block 结构数据来辨别其文件系统类型以及该文件系统当前的组织形式和架构。

int __init rd_load_disk(int n)函数执行过程：如果 rd_prompt 不等于 0，执行 change_floppy ("root floppy disk to be loaded into RAM disk")，执行 create_dev("/dev/root", ROOT_DEV, root_device_name)创建/dev/root 节点，对应的设备号为 ROOT_DEV，执行 create_dev("/dev/ram", MKDEV(RAMDISK_MAJOR, n), NULL)创建/dev/ram 节点，对应的设备号为 MKDEV(RAMDISK_MAJOR, n)即主设备号为 RAMDISK_MAJOR，次设备号为 n，执行 rd_load_image("/dev/root")。

void __init change_floppy(char *fmt, ...)函数执行过程：该函数只有配置了 CONFIG_BLK_DEV_RAM 或 CONFIG_BLK_DEV_FD 选项才有定义。执行 va_start(args, fmt)将 fmt 格式字符串后面第一个参数的地址传给 va_list(typedef char *) args；执行 vsprintf(buf, fmt, args)将 fmt 及后面的参数按格式打印到 char buf[80]字符串中，执行 va_end(args)（等于 0）。执行 sys_open("/dev/root", O_RDWR | O_NDELAY, 0)打开/dev/root 设备将返回的句柄传给 fd。如

果 fd 大于等于 0，执行 sys_ioctl(fd, FDEJECT, 0)，拒绝 fd 弹出 fd 所指/dev/root 设备，执行 sys_close(fd)。打印("VFS: Insert %s and press ENTER\n", buf)，执行 sys_open("/dev/console", O_RDWR, 0)以读写方式打开/dev/console 将返回句柄传给 fd。如果 fd 大于等于 0，执行 sys_ioctl(fd, TCGETS, (long)&termios)，将 struct termios termios.c_lflag 与上~ICANON，执行 sys_ioctl(fd, TCSETSF, (long)&termios)，执行 sys_read(fd, &c, 1)，执行 termios.c_lflag |= ICANON，执行 sys_ioctl(fd, TCSETSF, (long)&termios)，执行 sys_close(fd)。

void __init mount_root(void)函数执行过程：如果配置了 CONFIG_ROOT_NFS 选项，那么：如果 MAJOR(ROOT_DEV)等于 UNNAMED_MAJOR，执行 mount_nfs_root()，如果执行结果不等于 0，返回，否则打印("VFS: Unable to mount root fs via NFS, trying floppy.\n")。将 Root_FD0 传给 ROOT_DEV；#endif；#ifdef CONFIG_BLK_DEV_FD（该选项设置的依赖条件是(!ARCH_S390 && !M68K && !IA64) || Q40 || (SUN3X && BROKEN)）。如果 MAJOR(ROOT_DEV)等于 FLOPPY_MAJOR，且 rd_doload 等于 2，且执行 rd_load_disk(1)不等于 0，将 Root_RAM1 传给 ROOT_DEV，将 root_device_name 设置为 NULL，如果 rd_doload 不等于 2，执行 change_floppy("root floppy")；执行 create_dev("/dev/root", ROOT_DEV, root_device_name)创建/dev/root 节点，对应设备号为 ROOT_DEV，执行 mount_block_root ("/dev/root", root_mountflags)。

void __init mount_block_root(char *name, int flags)函数执行过程：定义 char b[BDEVNAME_SIZE]。执行__getname()从名字高速缓存 names_cachep 中申请一个名字空间，空间大小为最大路径字符串字符数 PATH_MAX，将返回值传给 char *fs_names。执行 get_fs_names(fs_names)获取 root_fs_names 所指的根文件系统类型名字符串或者已经注册到 file_systems 中的文件系统类型（需要设备支持的文件系统类型）的名字将其复制到 fs_names 中，然后针对 fs_names 中的每个文件系统类型名字 fsname 作如下操作：执行 do_mount_root(name, fsname, flags, root_mount_data)也就是将 name 所指设备的文件系统超级块尝试按各种文件系统类型进行挂接将返回值传给 int err，如果 err 等于 0 跳转到“out 标号”处，且 err 等于-EACCES，将 flags 或上 MS_RDONLY，然后重新开始针对每个文件系统类型名字作以上操作，如果 err 等于-EINVAL 继续针对下一个文件系统类型名字进行操作，如果所有文件系统名字都操作完毕都挂接失败，执行 __bdevname(ROOT_DEV, b) 打印 ("unknown-block(%u,%u)"， MAJOR(ROOT_DEV), MINOR(ROOT_DEV))到 b 中，打印("VFS: Cannot open root device \"%s\" or %s\n", root_device_name, b)，打印("Please append a correct \"root=\" boot option\n")。打印("VFS: Unable to mount root fs on %s", b)，系统 panic。打印("VFS: Unable to mount root fs on %s", __bdevname(ROOT_DEV, b))，系统 panic。“out 标号”位置：执行 putname(fs_names)。

static void __init get_fs_names(char *page)函数执行过程：将 page 传给 char *s，如果 root_fs_names 非空，执行 strcpy(page, root_fs_names)将 root_fs_names 复制到 page 中，将 page 中的','改成'\0'；如果 root_fs_names 为空，执行 get_filesystem_list(page)将返回值传给 int len，将

page 中所有需要设备支持的文件系统类型名字后面的'\n'改成'\0'，最后在字符串结尾再加上一个'\0'。

int get_filesystem_list(char * buf)函数执行过程：将已经注册到 file_systems 的文件系统类型的名字依次按一下格式打印到 buf 中且打印的长度不能超过 PAGE_SIZE – 80，最后返回打印的长度值。

static int __init do_mount_root(char *name, char *fs, int flags, void *data)函数执行过程：执行 sys_mount(name, "/root", fs, flags, data)将设备 name 按 fs 文件系统类型挂接到/root 目录上将返回值传给 int err。如果 err 不等于 0，返回 err。执行 sys_chdir("/root")将/root 作为当前进程的当前目录，将 current->fs->pwdmnt->mnt_sb->s_dev 传给 ROOT_DEV；打印("VFS: Mounted root (%s filesystem)%s.\n", current->fs->pwdmnt->mnt_sb->s_type->name, current->fs->pwdmnt->mnt_sb->s_flags & MS_RDONLY ? " readonly" : "")，返回 0。

asmlinkage long sys_chroot(const char __user * filename)函数执行过程：执行 __user_walk(filename, LOOKUP_FOLLOW | LOOKUP_DIRECTORY | LOOKUP_NOALT, &(struct nameidata)nd)将返回值传给 int err。如果 err 不等于 0，跳转到“out 标号”处。执行 permission(nd.dentry->d_inode,MAY_EXEC,&nd)将返回值传给 err。如果 err 不等于 0，跳转到“dput_and_out 标号”处。将 err 设置为-EPERM，如果 current->cap_effective 没有设置 CAP_SYS_CHROOT 标志即当前进程没有全县修改根目录，跳转到“dput_and_out 标号”处。执行 set_fs_root(current->fs, nd.mnt, nd.dentry)将 nd.mnt 传给 current->fs->rootmnt，将 nd.dentry 传给 current->fs->root，执行 set_fs_altroot()，将 err 设置为 0。“dput_and_out 标号”位置：执行 path_release(&nd)。“out 标号”位置：返回 err。

17.6.9 free_initmem()函数

void free_initmem(void)函数执行过程：如果既不是“integrator”平台也不是“cintegrator”平台，则执行 free_area((unsigned long)(&__init_begin), (unsigned long)(&__init_end), "init")按页释放__init_begin 到__init_end 之间的名叫“init”的全部内存空间（这里属于.init.data 区，由宏__initdata 和__INITDATA 定义，结束地址 8KB 对齐），更新系统空闲内存总页数。

17.6.10 unlock_kernel()函数

void __lockfunc unlock_kernel(void)函数执行过程：如果当前进程的锁深度值 current->lock_depth 原来为负数，则系统崩溃，如果原来等于 0 则将其减 1，再将内核锁 kernel_lock 原子数清 0，最后将 current-> thread_info->preempt_count 减 1 使能抢占。

将 SYSTEM_RUNNING 传给全局枚举变量 system_state，即将系统设置为正在运行状态。

17.6.11 numa_default_policy()函数

该函数将当前进程的内存访问模式设置为默认模式，具体执行过程说明如下。

void numa_default_policy(void)函数执行过程：直接执行并返回 sys_set_mempolicy(MPOL_DEFAULT, NULL, 0)。

asmlinkage long sys_set_mempolicy(int mode, unsigned long __user *nmask, unsigned long maxnode)函数执行过程：检查所有内存节点情况，如果检查没有问题，则根据具体情况从内存访问政策高速缓存空间 policy_cache 为指定内存访问政策模式申请一个 struct mempolicy 结构变量空间，并对其进行初始化。如果当前任务的 mempolicy 指针非空，把 mempolicy 指针所指的原来的 memorypolicy 结构变量空间释放掉，再将其指向刚创建的内存访问政策结构变量。如果是插入模式，则将下一个在线的内存节点号传给当前进程描述结构的 il_next 成员变量。

调用 sys_open((const char __user *) "/dev/console", O_RDWR, 0)函数，如果返回值为负值，说明不能打开系统的控制台设备，那么打印"Warning: unable to open an initial console.\n"系统告警信息。

17.6.12 sys_dup()函数

连续调用两次 sys_dup(0)函数。

asmlinkage long sys_dup(unsigned int fildes)函数将-EBADF 传给 int ret。将 fget(fildes)传给 struct file *。如果 file 非空，执行 dupfd(file, 0)将返回值传给 ret，返回 ret。

如果字符串指针 execute_command 不等于 NULL，调用 run_init_process(execute_command)函数。

static void run_init_process(char *init_filename)函数执行过程：将 init_filename 传给 argv_init[0]，执行 execve(init_filename, argv_init, envp_init)。

17.6.13 execve()函数

long execve(const char *filename, char **argv, char **envp)函数执行过程：该函数由各处理器架构自行定义，在 ARM 处理器中的定义如下：声名寄存器结构 struct pt_regs regs 并将其空间全部清 0。执行 do_execve((char *)filename, (char __user * __user *)argv, (char __user * __user *)envp, &regs)将返回值传给 int ret。如果 ret 小于 0，返回 ret。将 regs.ARM_r0 设置为 ret，然后执行一段汇编代码，这段代码将 regs 结构中全部寄存器的值复制到当前进程栈顶，即 current 的 struct thread_info 结构 current_thread_info()的地址加上 THREAD_SIZE - 8 - sizeof(regs)位置，然后禁止 IRQ 中断，进入超级用户模式，检查 current_thread_info()->flags 中设否设置了

_TIF_WORK_MASK 中的位，如果是，跳转到 work_pending()汇编函数处，否则执行 no_work_pending 处的 slow_restore_user_regs()汇编函数，返回 0。

int do_execve(char * filename, char __user *__user *argv, char __user *__user *envp, struct pt_regs * regs)函数执行过程：将-ENOMEM 传给 int retval，将 kmalloc(sizeof(*bprm), GFP_KERNEL)传给 struct linux_binprm *bprm。如果 bprm 为空，跳转到“out_ret 标号”处。将 bprm 空间清 0。执行 open_exec(filename) 当前任务的可选根挂接和根目录/根挂接和根目录/当前挂接和当前目录下查找名字 filename 对应的挂接和目录条目结构并保存到 nd 中，检查当前任务是否有权限执行该名字对应的目录条目的索引节点 inode，从文件结构 struct file 高速缓存 filp_cachep 中为新的文件结构变量 file 分配空间，然后初始化并打开 file，返回文件结构指针传给 struct file *file。将 PTR_ERR(file)传给 retval。如果 file 指针错，跳转到“out_kfree 标号”处。执行 sched_exec()，将 PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *)传给 bprm->p，将 file 传给 bprm->file，将 filename 传给 bprm->filename，将 filename 传给 bprm->interp，执行 mm_alloc()在 struct mm_struct 结构高速缓存 mm_cachep 中为 bprm->mm 申请空间并初始化 bprm->mm 并将返回值传给 bprm->mm。将-ENOMEM 传给 retval，如果 bprm->mm 为空，跳转到“out_file 标号”处。执行 init_new_context(current, bprm->mm) （该函数只在 ARMv6 以上版本由定义）初始化二进制程序描述结构的内存管理结构的上下文结构将返回值传给 retval。如果 retval 小于 0，跳转到“out_mm 标号”处。执行 count(argv, bprm->p / sizeof(void *))计算 argv 参数字符串组中的参数个数传给 bprm->argc。如果(retval = bprm->argc)小于 0，跳转到“out_mm 标号”处。执行 count(envp, bprm->p / sizeof(void *))计算 envp_init 系统环境变量字符串组中的环境变量个数传给 bprm->envc。如果(retval = bprm->envc)小于 0，跳转到“out_mm 标号”处。执行 security_bprm_alloc(bprm)传给 retval。如果 retval 不等于 0，跳转到“out 标号”处。执行 prepare_binprm(bprm)准备二进制程序。执行 copy_strings_kernel(1, &bprm->filename, bprm)传给 retval。如果 retval 不等于 0，跳转到“out 标号”处。将 bprm->p 传给 bprm->exec，执行 copy_strings(bprm->envc, envp, bprm)函数并将返回值传给 retval。如果 retval 不等于 0，跳转到“out 标号”处。执行 copy_strings(bprm->argc, argv, bprm)传给 retval。如果 retval 不等于 0，跳转到“out 标号”处。执行 search_binary_handler(bprm,regs)传给 retval；如果 retval 大于等于 0，执行 free_arg_pages(bprm)，执行 security_bprm_free(bprm)，执行 kfree(bprm)，返回 retval。“out 标号”位置：逐页释放二进制程序描述结构的所有参数页。如果 bprm->security 不等于 0，执行 security_bprm_free(bprm)。“out_mm 标号”位置：如果 bprm->mm 不等于 0，执行 mmdrop (bprm->mm)。“out_file 标号”位置：如果 bprm->file 不等于 0，将 bprm->file->f_dentry->d_inode->i_writecount 加 1，执行 fput(bprm->file)。“out_kfree 标号”位置：执行 kfree(bprm)。“out_ret 标号”位置：返回 retval。

slow_restore_user_regs()汇编函数执行过程：将保存在 current_thread_info()中的系统当前状态寄存器 CPSR(reg[16])的值保存到 spsr_cxsf 中，将 PC 值保存到 lr 寄存器中，再将保存的 r0~lr

寄存器的值全部恢复到 r0~lr 寄存器中，再将 sp 指向保存 r3 的位置，将 lr 传给 pc 跳转到 lr 地址处执行。

work_pending()汇编函数执行过程：如果 current_thread_info()->flags 设置了_TIF_NEED_RESCHED 位，调用 schedule()。如果 current_thread_info()->flags 既没有设置_TIF_NOTIFY_RESUME 也没有设置_TIF_SIGPENDING，执行 slow_restore_user_regs ()。将 sp 传给 r0，将 why 传给 r2。调用 do_notify_resume()，禁止 IRQ 中断，进入 supervisor 模式。执行 slow_restore_user_regs ()。

调用 run_init_process("/sbin/init")函数，运行用户/sbin/init 程序。

调用 run_init_process("/etc/init")函数，运行用户/etc/init 脚本。

调用 run_init_process("/bin/init")函数，运行用户/bin/init 程序。

调用 run_init_process("/bin/sh")函数，运行用户 shell 程序，进入 shell 交互界面。

调用 panic("No init found. Try passing init= option to kernel.")函数，如果前面不能进入 shell 人机交互界面，那么系统 panic。