Linux 内核（2.6.13.2）源代码分析

苗彦超

摘要:

1 系统启动

1.1 汇编代码 head.S 及以前

设置 CPU 状态初值，创建进程 0，建立进程堆栈：

movq init_rsp(%rip), %rsp，init_rsp 定义

```
.globl init_rsp
init_rsp:
.quad  init_thread_union+THREAD_SIZE-8
```

即将虚地址 init_thread_union+THREAD_SIZE-8 作为当前进程（进程 0）核心空间堆栈栈底，init_thread_union 定义于文件 arch/x86_64/kernel/init_task.c 中：

```
union  thread_union  init_thread_union  __attribute__((__section__(".data.init_task"))) =
        {INIT_THREAD_INFO(init_task)};
```

INIT_THREAD_INFO 定义于文件 include/asm-x86_64/thread_info.h 中，初始化 init_thread_union.task = &init_task，init_task 同样定义于文件 init_task.c 中，初始化为：

```
struct task_struct init_task = INIT_TASK(init_task);
```

INIT_TASK 宏在 include/linux/init_task.h 中定义。

全部利用编译时静态设置的初值，将进程 0 的控制结构设置完成，使进程 0 可以按普通核心进程访问。

```
init_task.mm = NULL; init_task.active_mm = INIT_MM(init_mm), init_task.comm = “swapper”
```

INIT_MM 将 init_mm.pgd 初始化为 swapper_pg_dir，即 init_level4_pgt，定义与 head.S 中。进程 0 的名称为 swapper。

利用下述汇编代码跳转到 C 函数执行：

```
movl%esi, %edi                  // 传递函数参数
movq    initial_code(%rip),%rax
jmp  *%rax
initial_code:
.quad    x86_64_start_kernel
```

开始执行文件 arch/x86_64/kernel/head64.c 中的 C 函数 x86_64_start_kernel(char * real_mode_data)，

1.2 函数 x86_64_start_kernel(char * real_mode_data)

1. 设置全部中断向量初始入口为 early_idt_handler，加载中断描述符 idt_descr
2. clear_bss()：BSS 段清 0
3. pda_init(0)：设置处理器 0 相关信息（processor datastructure area ？），重置 CR3 为 init_level4_pgt
4. copy_bootdata：复制 BIOS 启动参数到操作系统变量 x86_boot_params 中，再复制启动命令行参数由 x86_boot_params 到 saved_command_line 中，用 printk 显示 saved_command_line，从此不再与实模式数据打交道
5. cpu_set：设置 CPU 0 开始工作标志
6. 处理“earlyprintk=”、“numa”、“disableapic”等命令行参数
7. setup_boot_cpu_data()：设置 CPU 信息结构 boot_cpu_data，使用 cpuid 指令

8 执行 start_kernel()函数

1.3 start_kernel 函数

1.3.1 系统结构相关初始化前

lock_kernel()：文件 lib/kernel_lock.c 实现了 BKL（big kernel lock），使用：lock_kernel/unlock_kernel

如果开启 PREEMPT_BKL，则使用信号量 kernel_sem 实现，否则使用自旋锁 kernel_flag 实现。通常缺省开启 PREEMPT_BKL 选项。

当 task->lock_depth 等于-1 时，执行 down(&kernel_sem)操作 current->lock_depth++

unlock_kernel 执行--current->lock 和 up(&kernel_sem)操作

page_address_init()：在 x86-64 系统中为空函数。

printk(linux_banner)：打印特征信息

1.3.2 体系结构相关初始化 setup_arch(&command_line)

9 setup_memory_region()：

I sanitize_e820_map(E820_MAP, &E820_MAP_NR)：清理 E820 图，E820_MAP，E820_MAP_NR 均为 x86_boot_params 中的参数

II copy_e820_map()：调用 add_memory_region()函数将有效的地址区间加入到 E820 结构 struct e820map e820 中，最为系统 E820 图

III e820_print_map()：调用 printk 显示最终的 E820 图及数据来源 BIOS-e820、BIOS-e88 或 BIOS-e801

10 copy_edd()：如果开启编译选项 EDD，则复制 EDD 信息由 EDD_MBR_SIGNATURE 到变量 struct edd edd 中，EDD_MBR_SIGNATURE 由启动参数 x86_boot_params 设置。EDD：Enhanced Disk Dirve Services，参数将传递给 drivers/firmware/edd.c，参考 include/linux/edd.h

11 设置 init_mm、code_resource、data_resource 信息

12 parse_cmdline_early：分析早期使用的命令行参数

13 再次设置 CPU 参数信息结构 boot_cpu_data

14 end_pfn = e820_end_of_ram()：分析 E820 图，设置内存容量相关全局变量：end_user_pfn：启动参数 mem=xx 设置的页面数目；end_pfn_map：系统 RAM（主存）页面数，即建立直接映射页表的页面数，可通过__va、__pa 宏进行地址操作；end_pfn：操作系统直接管理的页面数

15 check_efer：读取 msr 寄存器 MSR_EFER，测试扩展特性 extended feature register

16 init_memory_mapping(0, end_pfn_map<<PAGE_SIZE)：建立直接映射页表

I find_early_table_space：根据映射内存总容量计算页表 pud 和 pmd 的需求量（2M 页面，3 级页表）tables 字节（页面容量 PAGE_SIZE 的整数倍），利用 E820 图，从物理地址 8000h 开始寻找容量为 tables 字节的连续物理内存，并跳过区间[640KB，_end]的保留内存，通常情况下寻找的结果 start 就是从 8000h 开始，设置全局变量 table_end = table_start = start >> PAGE_SHIFT，区间[table_start，table_end]即为直接映射页面表。

II 建立区间[0, end_pfn_map<<PAGE_SIZE)的直接映射，利用 alloc_low_page 分配页面并建立临时映射，利用 phys_pud_init 设置 pud，并设置 pgd，利用 unmap_low_page 解除临时映射。

III allow_low_page(&map, &pud_phys)：使用物理页面 table_end，并且 table_end++，将分配的物理页面临时映射到虚地址 40M 或 42M，使用临时页表 temp_boot_pmds（定义于文件 head.S）。

IV unmap_low_page(map)解除 allow_low_page 在 40M 或 42M 的临时映射

17 acpi_boot_table_init（arch/i386/kernel/acpi/boot.c 中）：ACPI 初始化

I acpi_table_init（drivers/acpi/tables.c）：ACPI 表初始化（Initialize the ACPI boot-time table parser）

i acpi_find_rsdp：定位 RSDP（Root System Description Pointer）位置，

A acpi_scan_rsdp(0, 0x400)：在区间[0,3FFh]搜索 RDSP 签字“RSD PTR”。

B acpi_scan_rsdp(0xE0000, 0x20000)：在区间[E_0000h,F_FFFFh]搜索 RDSP 签字“RSD PTR”。

C 搜索成功返回签字所在地址，否则返回 0。

ii 通过 printk 显示“RSDP(rsdp->version，rsdp->oem_id，rsdp_phys)”信息。

iii acpi_table_compute_checksum：计算 rsdp 校验和

iv acpi_table_get_sdt(rsdp)：以版本 2.0 及以上为例：

A std_pa = ((struct acpi20_table_rsdp*)rsdp)->xsdt_address：获取 XSDT 表物理地址

B header = __acpi_map_table(std_pa)：获取 ACPI 表头虚地址，x86-64 使用__va 直接映射，std_pa 不超过 8M 时 i386 也使用__va 直接映射，超过 8M 时使用固定映射

C mapped_xstd = __acpi_map_table(std_pa)，映射整个 XSDT（Extended System Description Table）

D 检查 XSDT 表头签名“XSDT”和校验和

E 设置 sdt_count 和 XSDT 表中各条目物理地址到 std_entry[i].pa 中

F acpi_table_print(header, sdt_pa)：用 printk 显示头部信息

G __acpi_map_table(sdt_entry[i].pa)：对 XSDT 中的每个表项物理地址 std_entry[i].pa 作为一个 acpi_table_header 结构进行地址映射，调用 acpi_table_print 显示并计算校验和，设置 std_entry[i].size 字段，将签名 header->signature 与数组 acpi_table_signatures 中的名字比较，设置 std_entry[i].id 字段。**数组 acpi_table_signatures 定义形式比较怪异。**

H acpi_get_table_header_early：搜索 ACPI_DSDT 并调用 acpi_table_print 打印，但物理地址不清楚，设置为 0

II acpi_table_parse(ACPI_BOOT, acpi_parse_sbf)：在 sdt_entry 中搜索 ACPI_BOOT 表项，并调用 acpi_parse_sb(sdt_entry[?].pa, sdt_entry[?].size)，映射 sb = __acpi_map_table(sdt_entry[?].pa, size)，设置 sbf_port = sb->sbf_cmos

III acpi_blacklisted（drivers/acpi/backlist.c）：sdt_entry[*]中是否有表 acpi_backlist[]中给出的 ACPI ID，符合条件则给出错误并可能调用 acpi_disable 关闭 acpi 功能

18 acpi_numa_init：需要开启编译选项 ACPI_NUMA，

I acpi_table_parse(ACPI_SRAT, acpi_parse_srat)：分析 SRAT（System Resource Affinity Table）

II acpi_table_parse_srat(ACPI_SRAT_PROCESSOR_AFFINITY,acpi_parse_processor_affinity,NR_CPUS);

i acpi_table_parse_madt_family(ACPI_SRAT, sizeof(struct acpi_table_srat), ACPI_SRAT_PROCESSOR_AFFINITY, acpi_parse_processor_affinity, NR_CPUS)：

A 定位 MADT，MADT 在 sdt_entry[*]中的 ID：ACPI_SRAT

B 查找 MADT 中 ID= ACPI_SRAT_PROCESSOR_AFFINITY 的表项，对每个表项调用函数 acpi_parse_processor_affinity

C acpi_parse_processor_affinity：

a acpi_table_print_srat_entry：打印信息

b acpi_numa_processor_affinity_init(processor_affinity)：

(1) pxm = processor_affinity->proximity_domain;

(2) setup_node(pxm)：

(I) nodes_weight(nodes_found)：最终调用 generic_hweight64(nodes_found)计算 nodes_found 中为 1 的 bit 个数

(II) fisrt_unset_node：查找第一个为 0 的 bit 序号 node

(III) node_set(node, nodes_found)

(IV) pxm2node[pxm] = node

(3) cpu_to_node[num_processors] = node，acpi_numa = 1

(4) 显示信息：printk(KERN_INFO "SRAT: PXM %u -> APIC %u -> CPU %u ->

Node %u\n", pxm, pa->apic_id, num_processors, node)

(5) 增加处理器计数：num_processors++

III acpi_table_parse_srat(ACPI_SRAT_MEMORY_AFFINITY, acpi_parse_memory_affinity, NR_NODE_MEMBLKS)，处理过程与上一步类似，只是最后一步调用函数 acpi_parse_memory_affinity，进而调用函数 acpi_numa_memory_affinity_init，处理内存节点。并设置 nodes_parsed 和 nodes 字段。**注释中说主要用于 IA64。**

IV acpi_table_parse(ACPI_SLIT, acpi_parse_slit)：分析 SLIT（System Locality Information Table）

V acpi_numa_arch_fixup：空函数

19 开启编译选项 NUMA 调用函数 numa_initmem_init(0, end_pfn)，否则调用函数 contig_initmem_init(0, end_pfn)

20 numa_initmem_init：

I 若开启 ACPI_EMU 编译选项，则执行 numa_emulation(0, end_pfn)，成功则 numa_initmem_init 返回；该选项主要用于调试

II 若开启 ACPI_NUMA 编译选项，则执行 acpi_scan_nodes(0, end_pfn<< PAGE_SHIFT)，成功则 numa_initmem_init 返回；

i compute_hash_shift：计算 memnode_shift

ii 显示特征信息：printk(KERN_DEBUG"Using %d for the hash shift. Max adder is %lx \n",shift,maxend);

iii 根据 nodes 字段设置，对每个节点调用函数 setup_node_bootmem(i, nodes[i].start, nodes[i].end)

iv numa_init_array：设置 cpu_to_node、node_to_cpumask 等序号映射字段。

III 若开启 K8_NUMA 编译选项，则执行 k8_scan_nodes(0, end_pfn << PAGE_SHIFT)，成功则 numa_initmem_init 返回；k8_scan_nodes 中定义的结构最大支持 8 个 NODE。

i find_northbirdgh：查找 CPU 中的北桥模块中的内存地址映射功能（功能 0：HyperTransport Technology Configuration，功能 1：Address Map），(VendorID : DeviceID) = (1022:1100/1101)，返回设备号

ii 特征信息：printk(KERN_INFO "Scanning NUMA topology in Northbridge %d\n", nb);

iii 读北桥设备功能 0(1022:1100)Offset 60h 信息（NodeID），计算系统中的 Node 数目，即处理器数目。

iv 显示表示信息：printk(KERN_INFO "Number of nodes %d\n", numnodes)

v 读北桥设备功能 1 偏移量 40h~7Ch，获取内存分布信息和各内存地址对应的 nodeid，记录到局部变量 nodes 中，nodes[nodeid].start = base, nodes[nodeid].end = limit，在 nodes_parsed 中标记有效 nodeid。

vi memnode_shift = compute_hash_shift(nodes, numnodes)

A maxend = MAX{nodes[*].end}

B 满足条件(1UL << shift) < maxend / NODEMAPSIZE 的最小 shift 值，NODEMAPSIZE=0xFF。返回 shift

C 对全部内存地址 addr，以粒度(1UL<<shift)设置 memnodemap[addr>>shift]=i，i 为 NODE 号（0~7）

注：

memnode_shift：将总物理内存等分成 255 段，每段容量的移位数

memnodemap[0..254]：每等分段归属 NODE 号

vii **标志信息**：printk(KERN_INFO "Using node hash shift of %d\n", memnode_shift)

viii 对于全部配置有物理内存的 NODE：设置 cpu_to_node[i] = i，setup_node_bootmem(i, nodes[i].start, nodes[i].end)：

A start = round_up(start, ZONE_ALIGN)：圆整起始地址，ZONE_ALIGN：

1<<(MAX_ORDER+PAGE_SHIFT) = 8MB

B **标志信息**：printk("Bootmem setup node %d %016lx-%016lx\n", nodeid, start, end)

注：*struct mem_section mem_section[NR_MEM_SECTIONS(8192)]，导出，共占用 64KB 空间*

C memory_present：对于本 NODE 的全部 section，标记 mem_section[section].section_mem_map 有效

D nodedata_phys = find_e820_area(start, end, pgdat_size)：分配 pg_data_t 结构内存，并页面容量对齐，从本 NODE 的内存中分配。

E node_data[nodeid] = phys_to_virt(nodedata_phys)，node_data：64 个分量

F node_data[nodeid]->bdata = &plat_node_bdata[nodeid]（全局变量）

G node_data[nodeid]->node_start_pfn = start_pfn（本 NODE 起始页面号）

H node_data[nodeid]->node_spanned_pages = end_pfn - start_pfn（本 NODE 页面数）

I bootmap_pages = bootmem_bootmap_pages(end_pfn - start_pfn)：计算本 NODE 全部内存建立位图需要的页面数

J bootmap_start = round_up(nodedata_phys + pgdat_size, PAGE_SIZE)：本 NODE 空闲物理内存页面容量对齐基地址

K bootmap_start = find_e820_area(bootmap_start, end, bootmap_pages<<PAGE_SHIFT)：分配本 NODE 内存位图空间

L bootmap_size = init_bootmem_node(node_data[nodeid], bootmap_start >> PAGE_SHIFT, start_pfn, end_pfn)，直接调用 init_bootmem_core(pgdat, freepfn/mapstart, startpfn, endpfn)，参数按顺序直接结合

a bdata = node_data[nodeid]->bdata(前面已设置为&plat_node_bdata[nodeid])

b bdata->node_bootmem_map = phys_to_virt(mapstart << PAGE_SHIFT)

c bdata->node_boot_start = (start << PAGE_SHIFT)（**重置**）

d bdata->node_low_pfn = end

e 位图区 bdata->node_bootmem_map 全部设置为 1，保留全部内存

f 返回位图容量，8 字节对齐

M e820_bootmem_free(node_data[nodeid], start, end)：根据 e820 表，对于本节点全部属于 E820_RAM、且 e820 图标志为有效的内存区域，调用函数 free_bootmem_node(node_data[nodeid], addr, last-addr)，进一步直接调用 free_bootmem_core(node_data[nodeid]->bdata, physaddr, size)，将全部有效页面在 bdata->node_bootmem_map 中的对应比特清 0，标记内存为空闲。

N 保留 pg_data_t 结构占用的内存 node_data[nodeid]

O 保留位图占用内存 bootmap_start

P 在 node_online_map 中标记本 NODE 有效

ix numa_init_array：对于其它 CPU（未配置有物理内存），设置 cpu_to_node[i]值，并在 node_online_map 中标记本 CPU 对应的 NODE 号有效，设置 node_to_cpumask[cpu_to_node(0)]的 bit0 置 1

IV 使用 No NUMA 配置：

i memnode_shift = 63，memnodemap[0] = 0，node_online_map 清 0，仅 NODE0 有效

ii cpu_to_node[*] = 0，node_to_cpumask[0] = cpumask_of_cpu(0)

iii setup_node_bootmem(0, 0, end_pfn << PAGE_SHIFT)：设置全部物理内存归 NODE 0 管理

21 contig_initmem_init(0, end_pfn)：

I memory_present(0, start_pfn, end_pfn)

II bootmap_size = bootmem_bootmap_pages(end_pfn)<<PAGE_SHIFT：计算全部内存所需要

的位图容量

III bootmap = find_e820_area(0, end_pfn<<PAGE_SHIFT, bootmap_size)：分配位图空间

IV bootmap_size = init_bootmem(bootmap >> PAGE_SHIFT, end_pfn)：

i max_low_pfn = pages，min_low_pfn = start

ii init_bootmem_core(NODE_DATA(0), start, 0, pages)：设置位图

V e820_bootmem_free(NODE_DATA(0), 0, end_pfn << PAGE_SHIFT)：根据 e820 图，释放全部有效内存

VI reserve_bootmem(bootmap, bootmap_size)：保留位图内存

22 reserve_bootmem_generic(table_start << PAGE_SHIFT, (table_end - table_start) << PAGE_SHIFT)：保留直接映射页表内存

I int nid = phys_to_nid(phys)：通过 memnodemap[addr>>memnode_shift]得到 nid

II reserve_bootmem_node(NODE_DATA(nid), phys, len)：直接调用 reserve_bootmem_core (pgdat->bdata, physaddr, size)，bdata->node_bootmem_map 中对应 bit 清 0，标记为保留

23 保留核心映像内存，即区域[1M，__pa(_end)]，保留第 0 页物理内存

24 reserve_ebda_region()：保留 EBDA 区

25 若开启 SMP 选项：保留第 1 页内存和 trampoline 区，即第 6 页

26 若开启 ACPI_SLEEP 选项，acpi_reserve_bootmem：调用 alloc_bootmem_low 分配 1 页物理内存，并保存到 acpi_wakeup_address 中

I find_smp_config：直接调用 find_intel_smp，利用函数 smp_scan_config 在区间[0，1K)、[639K, 640K)和[960K, 1024K)搜索 SMP 配置，成功则 find_intel_smp 返回，若失败则读取物理地址 40Eh 数据 addr，乘以 16 作为基地址 base，再次调用函数 smp_scan_config 搜索区间[base, base+4K)搜索 SMP 配置

II smp_scan_config：寻找以 MP 表签名“_MP_”起始的区域作为 MP 表，若校验和等参数匹配成功认为发现 MP 表，并保留该区域所在页面，置 smp_found_config=1。若再进一步 MP 表中有配置表地址，即签名后的第一个字段不为 0，则保留配置表页面

27 若开启 BLK_DEV_INITRD 选项，若 initrd 区域有效，则保留 initrd 区域，基地址：INITRD_START；长度：INITRD_SIZE

28 若开启 KEXEC 选项，则保留 crashk_res 指定的区域。该区域由启动参数“crashkernel=”指定，

KEXEC 注释：*kexec is a system call that implements the ability to shutdown your current kernel, and to start another kernel. It is like a reboot but it is indepedent of the system firmware. And like a reboot you can start any kernel with it, not just Linux.*

29 sparse_init：开启编译选项 SPARSEMEM 有效，对于全部有效内存 SECTIONS，执行下面操作：第 pnum 个 SECTIOON，调用接口 alloc_bootmem_node 从本 SECTION 对应的 NODE 内存中为其分配一个 map 区域，为每个页面设置一个 page 结构，设置 mem_section[pnum]. section_mem_map |= map - section_nr_to_pfn(pnum)

30 paging_init（NUMA 结构）：根据 node_possible_map 结构，对于每个有效的 NODE *i*，调用函数 setup_node_zones(i)：

I start_pfn、end_pfn：本 NODE 内存起、止页面号，可能存在空洞

II 设置 zones[ZONE_DMA]、zones[ZONE_NORMAL]：内存区间[start_pfn，end_pfn]在 e820 中有效容量

III 设置 holes[ZONE_DMA]、holes[ZONE_NORMAL]：内存区间[start_pfn，end_pfn]在 e820 中**无效**容量，即本 NODE 内的内存空洞

注：*若 start_pfn ≥ dma_end_pfn(16M)，则 zones[ZONE_DMA]、holes[ZONE_DMA] 都为 0*

IV free_area_init_node(nodeid, NODE_DATA(nodeid), zones, start_pfn, holes)

i pgdat = NODE_DATA(nodeid)

ii pgdat->node_id = nid; pgdat->node_start_pfn = node_start_pfn;

iii calculate_zone_totalpages(pgdat, zones_size, zholes_size)：

A 重置 pgdat->node_spanned_pages = SUM{zones_size[*]}，与原值相同

B 设置 pgdat->node_present_pages = SUM{zones_size[*]} – SUM{holes_size[*]}

C 标志信息：printk(KERN_DEBUG "On node %d totalpages: %lu\n", pgdat->node_id, realtotalpages)

iv alloc_node_mem_map(pgdat)

A 如果开启编译选项 FLAT_NODE_MEM_MAP，则执行下述操作，否则为空函数

B 设置 pgdat->node_mem_map = alloc_bootmem_node(pgdat, size)，在本 NODE 内存中分配内存，用于保存 page 结构

C 若进一步开启编译选项 FLAGMEM，且 pgdat = NODE_DATA(0)则设置 mem_map = NODE_DATA(0)->node_mem_map

v free_area_init_core(pgdat, zones_size, zholes_size)：初始化队列头部 pgdat->kswapd_wait，设置 pgdat->kswapd_max_order = 0，对于本 NODE 内部的每个 zone（x86-64 最多只有 2 个有效 zone），执行下述操作：

A 将本 zone 内的实际页面数加入到 nr_kernel_pages、nr_all_pages 中

B 设置 zone->spanned_pages 为包含空洞的页面数，zone->presend_pages 为实际有效页面数

C 设置 zone 的名字：zone->name 指向 zone_names

D 设置各 CPU 单页面管理集：zone->pageset[cpu] = &boot_pageset[cpu]，调用函数 setup_pageset 进行初始化：pcp->high：255*6，pcp->low：255*2，pcp->batch = 255。

E 标志信息：printk(KERN_DEBUG " %s zone: %lu pages, LIFO batch:%lu\n", zone_names[j], realsize, batch)，注：*batch 应该为 255*

F 初始化 wait_table：zone->wait_table_xx 等

G 设置 zone->zone_start_page = zone_start_pfn

H 设置 zone->zone_mem_map = pfn_to_page(zone_start_pfn)

I memmap_init(size, nid, j, zone_start_pfn)，直接调用函数 memmap_init_zone，参数完全一致

J 对于本 zone 内的每个物理页面，设置其 page 结构属性参数

a 设置 page->flags

b 设置页面为 Reserved

K zonetable_add(zone, nid, j, zone_start_pfn, size)：设置全局变量 zone_table

L zone_init_free_lists(pgdat, zone, zone->spanned_pages)：初始化 zone->free_area 为空

31 check_ioapic：处理 VIA 和 Nvidia 主板的一些特性

32 若开启编译选项 ACPI_BOOT 则执行 acpi_boot_init：处理 ACPI_BOOT、ACPI_FADT、MADT（Multiple APIC Description Table）和 ACPI_HPET（需要开启编译选项 HPET_TIMER）

33 get_smp_config：

I 若 acpi_lapic && acpi_ioapic 同时有效则打印特征信息 printk(KERN_INFO "Using ACPI (MADT) for SMP configuration information\n")后返回

II 否则打印标志信息：：printk("Intel MultiProcessor Specification v1.%d\n", mpf->mpf_specification)

III 调用函数 smp_read_mpc 分析 MPC（multiple processor config）表，处理多处理器信息

i 比较 MPC 表签名是否为“PCMP”、校验和、版本号以及 LAPIC 是否存在

ii 打印标志信息：printk(KERN_INFO "OEM ID: %s ",str)

iii 打印标志信息：printk(KERN_INFO "Product ID: %s ",str)

iv 打印标志信息：printk(KERN_INFO "APIC at: 0x%X\n",mpc->mpc_lapic)

v 分析 MPC 表

vi 如果是处理器表项，则调用函数 MP_processor_info：

A 打印标志信息：printk(KERN_INFO "Processor #%d %d:%d APIC version %d\n",...);

B 增加总 CPU 计数 num_processors，

C physid_set(m->mpc_apicid, phys_cpu_present_map)：设置当前 CPU 物理 APIC_ID 到全局变量 phys_cpu_present_map 中

D 设置 bios_cpu_apicid[cpu]=x86_cpu_to_apicid[cpu]=m->mpc_apicid，其中 BP 时 cpu=0，AP 时为 CPU 在 MPC 表中的序号

E 在全局变量 cpu_possible_map 和 cpu_present_map 表中标志当前 CPU 有效

vii 如果是总线表项，则调用函数 MP_bus_info：

viii 如果是 IOAPIC 表项，则调用函数 MP_ioapic_info：

打印标志信息：printk("I/O APIC #%d Version %d at 0x%X.\n",...);

ix 如果是中断源表项，则调用函数 MP_intsrc_info：

x 如果是本地中断源表项，则调用函数 MP_lintsrc_info：

34 init_apic_mappings

I 建立 FIX_APIC_BASE 固定映射

II 建立 FIX_IO_APIC_BASE_0 等 IOAPIC 的固定映射

35 probe_roms：记录 ROM 占用的资源（地址空间信息等）

36 e820_reserve_resources：根据 e820 信息，保留 e820 内存在 iomem_resource（资源空间初值为全部地址空间）中占用的空间，以及核心映像代码段、数据段在每个 e820 资源空间的占用情况

37 保留视频 RAM 占用的 iomem_resource 资源空间

38 保留标准 I/O 设备占用 ioport_resource 资源空间

39 如果开启编译选项 GART_IOMMU，则调用函数 iommu_hole_init

40 若开启编译选项 VGA_CONSOLE（通常都开启）则设置 conswitchp = &vga_con，否则若开启 DUMMY_CONSOLE，则设置 conswitchp = &dummy_con

41 setup_arch 结束

1.3.3 后期体系结构无关初始化 start_kernel

42 setup_per_cpu_areas：对于系统中的每个 CPU，调用函数 alloc_bootmem 分配一个存储区 ptr，将数据区[__per_cpu_start，__per_cpu_end]的内容复制到存储区 ptr，并设置 cpu_pda[cpu].data_offset = ptr - __per_cpu_start

43 smp_prepare_boot_cpu：在全局变量 cpu_online_map、cpu_callout_map、cpu_sibling_map[0]和 cpu_core_map 中标记本处理器有效

44 sched_init：初始化各 CPU 的进程运行队列 runqueue，增加 init_mm.mm_count 引用计数，为本 CPU 初始化 idle 进程

45 build_all_zonelists：对于系统中的每个 NODE *i* 调用函数 build_zonelists(NODE_DATA(i)/pgdat)：

I 初始化 pgdat->node_zonelists[*].zone[0] = NULL

II 根据本 NODE 与系统中的全部 NODE 之间的距离，由近及远遍历系统系统中的全部 NODE，调用函数 build_zonelists_node 将 pgdat->zone_zonelist[*].zone[i]指向每个 NODE 中对应类型的 zone 区

注：

在 pgdat（类型 struct pglist_data{}）中定义：

```
struct zone node_zones[MAX_NR_ZONES];        // MAX_NR_ZONES = 3
struct zonelist node_zonelists[GFP_ZONETYPES]      // GFP_ZONETYPES = 3
```

而 zonelist 的定义为：

```
struct zonelist {
    truct zone *zones[MAX_NUMNODES * MAX_NR_ZONES + 1];
```

};

即全部为指向 zone 的指针，共计为系统中的 NODE 数目乘以每个 NODE 中的 zone 数目，另外一个为 NULL 的结束指针，即 zonelist 中的指针可以指向系统中的全部 NODE 中的每个 zone。

上述初始化过程就是将每个 NODE 的 zonelists 中的每个指针指向系统中全部 NODE 中的 zone，并按 NODE 距离升序排序，距离最近的排在前面。

III 显示特征信息：printk("Built %i zonelists\n", num_online_nodes())

IV cpuset_init_current_mems_allowed：设置 current->mems_allowed = NODE_MASK_ALL

46 age_alloc_init，直接调用宏 hotcpu_notifier(page_alloc_cpu_notify, 0)，定义一个静态变量通知块 struct nodifier_block page_alloc_cpu_notify_nb = { page_alloc_cpu_notify, 0}，并调用函数 register_cpu_notifier 注册通知块，将通知块 page_alloc_cpu_notify_nb 注册到全局 CPU 活动通知链 cpu_chain 中

47 显示特征信息：printk(KERN_NOTICE "Kernel command line: %s\n", saved_command_line)

48 parse_early_param：分析早期启动参数

49 parse_args：分析命令行参数：("Booting kernel", command_line, __start___param, __stop___param - __start___param, &unknown_bootoption);

50 sort_main_extable：直接调用函数 sort_extable(__start___ex_table, __stop___ex_table)，进而调用 sort 函数，对异常表中的内容进行快速排序

51 trap_init：异常初始化

I 初始化异常处理向量（小于 32 的中断向量）

II cpu_init()：初始化 CPU

i 若不是 CPU 0，则调用函数 pda_init(cpu)设置 CPU 基本信息

ii 并显示特征信息：printk("Initializing CPU#%d\n", cpu)

iii 设置 GDT 和 IDT

iv syscall_init()：系统调用入口初始化

A wrmsrl(MSR_STAR, ((u64)__USER32_CS)<<48 | ((u64)__KERNEL_CS)<<32)：设置 x86 遗留模式（legacy x86 mode）系统调用入口地址

B wrmsrl(MSR_LSTAR, system_call)：设置长模式（long mode）下 64 位软件入口地址

C syscall32_cpu_init()，需要开启编译选项 IA32_EMULATION，设置长模式下的兼容软件系统调用入口地址

注释：*x86-64 中使用 syscall/sysret 指令进入/返回系统调用，入口地址使用 STAR（C000_0081h）、LSTAR（C000_0082h）和 CSTAR（C000_0083h）模式寄存器。对应汇编指令入口分别是 ia32_syscall、ia32_cstar_target 和 system_call。系统调用表位于文件 arch/x86_64/ia32/ia32entry.S 和 include/asm-x86_64/unistd.h 中。不再使用 80h 软中断，但开启编译选项 IA32_EMULATION 后 80h 中断仍可用。*

III fpu_init()：初始化浮点处理器

52 rcu_init：rcu 初始化，

I 调用函数 rcu_cpu_notify；

II 调用函数 register_cpu_notifier 注册 rcu 通知块 rcu_nb 到 cpu_chain 链表上，其中 rcu_nb 的回调函数即是 rcu_cpu_notify；

53 init_IRQ：

I init_ISA_irqs：

i 调用 init_bsp_APIC：若 SMP 模式或者 CPU 已有 APIC 直接返回，否则设置本地 APIC

ii 调用函数 init_8259A(0)：初始化 8259

iii 初始化中断描述结构 irq_desc[224]为空状态，对于前 16 个中断则使用 8259A 类型处理

iv 设置中断向量[32..255]到中断门中

II 对于 SMP 配置，设置处理器间中断和 APIC 中断

III 调用函数 setup_timer：访问 I/O 端口 0x43、0x40 初始化定时器

IV 如果 acpi_ioapic 为 0，则初始化中断请求号 2（中断向量 34），中断处理程序为 irq2

54 pidhash_init：PID 哈希表初始化

55 init_timers：定时器初始化

I 调用函数 timer_cpu_notify；

II 调用函数 register_cpu_notifier 注册定时器通知块 timers_nb 到 cpu_chain 链表上，其中 timers_nb 的回调函数即是 timer_cpu_notify。

III open_softirq(TIMER_SOFTIRQ, run_timer_softirq, NULL)：初始化定时器软中断（TIMER_SOFTIRQ）：softirq_vec [TIMER_SOFTIRQ].data = NULL; softirq_vec [TIMER_SOFTIRQ].action = run_timer_softirq;

56 softirq_init：调用 open_softirq 初始化其它软中断：

I open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);

II open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);

57 time_init：系统时间初始化，设置全局变量 xtime、wall_to_monotonic、vxtime_hz、cpu_khz 等

I get_cmos_time：获得 CMOS 时间，设置到 xtime.tv_sec 中

II set_normalized_timespec：设置 wall_to_monotonic

III hpet_init：初始化 HPET

i 设置固定映射 FIX_HPET_BASE 和 VSYSCALL_HPET

ii 通过接口 hpet_readl 获取 HPET 信息，若存在 HPET 则继续执行

iii hpet_timer_stop_set_go：初始化 HPET

IV 若 HPET 存在，即 hpet_use_timer 置 1，则置 cpu_khz=hpet_calibrate_tsc()，设置时钟的名称 timename 为“HPET”，否则向下执行

V 若开启编译选项 X86_PM_TIMER 且初始化 ACPI 过程中将 pmtmr_ioport 置 1，则执行 pit_init，并置 cpu_khz = pit_calibrate_tsc()，设置时钟的名称 timename 为“PM”，否则向下执行

VI pit_init，并置 cpu_khz = pit_calibrate_tsc()，设置时钟的名称 timename 为“PIT”

VII 标志信息：printk(KERN_INFO "time.c: Using %ld.%06ld MHz %s timer.\n", vxtime_hz / 1000000, vxtime_hz % 1000000, timename); 这里给出使用时钟的名称 timename

VIII printk(KERN_INFO "time.c: Detected %d.%03d MHz processor.\n", cpu_khz / 1000, cpu_khz % 1000);

IX rdtscll_sync(&vxtime.last_tsc)：将 vxtime.last_tsc 设置为当前 TSC 值

X setup_irq(0, &irq0);设置定时器中断处理程序为 irq0

XI set_cyc2ns_scale(cpu_khz / 1000)

XII time_init_gtod：需要开启 SMP 选项，“Decide after all CPUs are booted what mode gettimeofday should use”

i unsynchronized_tsc()

ii 标志信息：printk(KERN_INFO "time.c: Using %s based timekeeping.\n", timetype)

58 console_init：早期控制台初始化

I tty_register_ldisc(N_TTY, &tty_ldisc_N_TTY)：设置 tty_ldiscs[N_TTY]，Setup the default TTY line discipline

II disable_early_printk，若开启编译选项 EARLY_PRINTK，关闭早期打印输出

III 执行函数指针[__con_initcall_start，__con_initcall_end]，即执行 console_initcall 定义的各函数

59 profile_init：

若 prof_on = 0 则直接返回，否则继续执行，prof_on 由启动参数 profile=xx 开启，参见本节注释

prof_len = (_etext - _stext) >> prof_shift

pro_buffer = alloc_bootmem(prof_len*sizeof(atomic_t))

注：*若命令行参数包含“profile=schedule,n”或“profile=n”（n 为数字），则执行 profile_setup 函数设置 profile 功能：*

i 若“profile=schedule,n”则

设置 prof_on = SCHED_PROFILING（等于 2）

prof_shift = n

***标志信息：** printk(KERN_INFO"kernel schedule profiling enabled (shift: %ld)\n", prof_shift);*

ii 若“profile= n”则

设置 prof_on = CPU_PROFILING（等于 1）

prof_shift = n

***标志信息：** printk(KERN_INFO "kernel profiling enabled (shift: %ld)\n",prof_shift)*

60 local_irq_enable：开启中断

61 vfs_caches_init_early：

I dcache_init_early：dentry 哈希表初始化：调用函数 dentry_hashtable = alloc_large_system ("Dentry cache", ...)分配 dentry 哈希表，并初始化表头全部为空。在函数 alloc_large_system 中显示**标志信息**：printk("%s hash table entries: %d (order: %d, %lu bytes)\n", tablename, (1U << log2qty), long_log2(size) - PAGE_SHIFT,size);

II inode_init_early：inode 哈希表初始化：调用函数 inode_hashtable =alloc_large_system_hash ("Inode-cache", ...)分配哈希表，并初始化表头全部为空。在函数 alloc_large_system 中显示**标志信息**：printk("%s hash table entries: %d (order: %d, %lu bytes)\n", tablename, (1U << log2qty), long_log2(size) - PAGE_SHIFT,size);

62 mem_init：内存页面管理机制初始化

I 设置全局变量 max_low_pfn = max_pfn = num_physpages = end_pfn;

II high_memory = (void *) __va(end_pfn * PAGE_SIZE)

III memset(empty_zero_page, 0, PAGE_SIZE)：保留的 0 页面 empty_zero_page 清 0

IV 开启 NUMA 编译选项，则执行 totalram_pages += **numa_free_all_bootmem**()，函数 numa_free_all_bootmem 根据标记 node_online_map，对每个在线有效 NODE *i* 调用函数 free_all_bootmem_node(NODE_DATA(i))，进而直接调用函数 free_all_bootmem_core(pgdat)建立本 NODE 页面表和 Buddy 内存管理机制：

i 本 NODE 起始物理页面号：pfn = bdata->node_boot_start（bdata = pgdat->bdata）

ii 本 NODE 页面数目 bdata->node_low_pfn - (bdata->node_boot_start >> PAGE_SHIFT)

iii 本 NODE 位图基地址 map = bdata->node_bootmem_map

iv 如果 pfn 等于 0，或 64（long 类型位图）页面对齐则标记 gofast=1，一次可以处理连续的 64 个页面。

v __ClearPageReserved：清除当前处理页面的 Reserved 标记

vi __free_pages(page, order)，order 根据连续的页面数目确定。

A free_hotpage(page)：如果 order 等于 0。进而直接调用 free_hot_cold_page(page, 0)

a zone = page_zone(page)：获取页面 page 所在的 zone 指针

b free_pages_check：测试 page 中的标志位，页面是否可以安全释放，如果 PG_dirty 标记有效，则调用宏 ClearPageDirty(page)直接清除“脏”标记。

c per_cpu_pages * pcp = zone-> pageset[get_cpu()]->pcp[0]，即获得本页面所属 zone、当前处理器对应的的单页面链表管理结构 pcp

d 使用 page->lru 指针将页面 page 加入到 pcp->list 页面中，并增加 pcp 计数。如果 pcp 中页面数量超过上限 pcp->high，则调用函数 free_pages_bulk(zone, pcp->batch, &pcp->list, 0)释放 pcp->batch 个页面。

e free_pages_bulk：当队列 pcp->list 不空且处理的页面不超过 pcp->batch 时，调用函数__free_page_bulk，每次释放一个页面。__free_page_bulk：

(1) destroy_compound_page(struct page* page, unsigned long order)：当开启编译选项 HUGETLB_PAGE 且 order > 0 时调用

(I) 如果标记 PG_compound 未置位，不是一个复合页面则直接返回。

(II) if (page[1].index != order) 则出错，即大页面中的第二个 4KB 子页面对应的 page 结构中 index 成员用于记录页面大小

(III) ClearPageCompound：清除大页面的全部子页面的复合页面标记，同时判断各页面 page 的 private 成员是否指向第一个页面的 page 结构

(IV) page_idx = page_to_pfn(page) & ((1 << MAX_ORDER) – 1)，当前页面在 BUDDY 范围内序号

(V) while (ordef < MAX_ORDER – 1) {...}，整个循环实现 buddy 算法的内存回收过程

1 combined_idx = __find_combined_index (page_idx, order) = (page_idx & ~(1 << order))，结合后的第一个页面序号，page_idx 是 2^{order} 对齐的页面号。除非 order=0，page_idx 为任意值，否则是一个 BUDDY 的首页面

2 buddy_idx = page_idx ^ (1 << order)

3 bad_range(zone, buddy)：判断 buddy_idx 是否在正常返回内，且属于本 zone 等，非法则推出循环

4 page_is_buddy(buddy, order)：buddy 是否有效，非法则推出循环

5 (zone->free_area+order)->nr_free--; rmv_page_order(buddy); buddy 已结合成更大的页面，从当前 order 中删除

6 page_idx = combined_idx；order++；进行下一个循环处理

(VI) set_page_order(page, order)：page->private = order；设置 page->flags 中的 PG_private 标志有效

(VII) list_add(&page->lru, &zone->free_area[order].free_list)：将 page 结构加入到相应空闲链表中

B __free_pages_ok(page, order)，order 不等于 0 时调用

A LIST_HEAD(list)：定义一个临时表表头

B mod_page_state(pgfree, 1 << order)：page_states.pgfree += 1 << order，结构变量 page_states，每个 CPU 定义一个独立分量

C free_pages_check：核对每个页面的有效性

D list_add(&page->lru, &list)：将第一个页面加入到临时表头中，该链表只有这一个分量

E free_pages_bulk(page_zone(page), 1, &list, order)：释放页面，同前面说明、

63 kmem_cache_init：kmem_cache 机制初始化

I 初始化链表 cache_chain

II 初始化 cache_cache，将 cache_cache 加入到 cache_chain 中，每次调用函数 kmem_cache_create 创建一个 kmem_cache 后，都将使用 cache.next 域加入链表 cache_chain 中

III 初始化 malloc_sizes，为各 malloc_sizes[*]中成员 cs_cachep 和 cs_dmacachep 分别调用函数 kmem_cache_create，建立内存空间

IV 设置 cache_cache.array[smp_processor_id()]和 malloc_sizes[0].cs_cachep->array[smp_processor_id()]

V register_cpu_notifier(&cpucache_notifier)

64 setup_per_cpu_pageset：

I process_zones(smp_processor_id())：为 CPU 0 设置 per_cpu_pageset

i 对系统中的每个 zone，设置 zone->pageset[cpu]= kmalloc_node(...)；kmalloc_node(size_t size,

unsigned int __nocast flags, int node)：功能与 kmalloc 相同，也是在 malloc_size[*]中分配指定容量内存，不同的是优先分配指定节点（NODE）node 的内存，节点 node 没有内存再从其它节点分配

ii setup_pageset：设置 zone->pageset[cpu]->pcp[*]

II register_cpu_notifier(&pageset_notifier)：注册 CPU 启动通知链 pageset_notifier

65 numa_policy_init：

I 初始化专用缓存：policy_cache = kmem_cache_create ("numa_policy", sizeof(struct mempolicy), ...)

II 初始化专用缓存：sn_cache = kmem_cache_create("shared_policy_node", sizeof(struct sp_node), ...)

III sys_set_mempolicy(MPOL_ INTERLEAVE, ...)：设置内存为交存（MPOL_INTERLEAVE）策略，以使启动过程中分配的内存不全在 NODE 0 中；

66 calibrate_delay：标定时钟，设置计算能力标称值 BogoMIPS，设置全局变量 loops_per_jiffy

I 若使用了命令行参数“lpj=xxx”，直接置 loops_per_jiffy = preset_lpj，并显示**标志信息**：printk("Calibrating delay loop (skipped)... %lu.%02lu BogoMIPS preset\n", loops_per_jiffy/(500000/HZ), (loops_per_jiffy/(5000/HZ)) % 100);...);

II loops_per_jiffy = calibrate_delay_direct()：若返回值不为 0，则显示**标志信息**：printk("Calibrating delay using timer specific routine.."); printk("%lu.%02lu BogoMIPS (lpj=%lu)\n", loops_per_jiffy/(500000/HZ), (loops_per_jiffy/(5000/HZ)) % 100, loops_per_jiffy);

III 开始执行标定代码，计算 loops_per_jiffy，显示**标志信息**：printk(KERN_DEBUG "Calibrating delay loop... "); printk("%lu.%02lu BogoMIPS (lpj=%lu)\n", 同上);

注：*命令行参数“lpj=xxx”修改全局变量 preset_lpj，在此起作用*

67 pidmap_init：设置全局变量 pidmap_array[0]，为 pidmap_array[0].page 分配一个内存页面，调用函数 attach_pid(current,...)标志进程 0 PID 已经使用

68 pgtable_cache_init：空函数

69 prio_tree_init：基数优先级搜索树（radix priority search tree）初始化，初始化静态变量 index_bits_to_maxindex

70 anon_vma_init：初始化专用缓存：anon_vma_cachep = kmem_cache_create ("anon_vma", sizeof(struct anon_vma), ...)

71 如果 efi_enabled 则调用函数 efi_enter_virtual_mode，需要编译选项 EFI

72 fork_init(num_physpages)：进程管理初始化

I 初始化专用缓存：task_struct_cachep = kmem_cache_create("task_struct", sizeof(struct task_struct), ...);

II 设置 max_threads：max_threads = mempages / (8 * THREAD_SIZE / PAGE_SIZE)

III 设置 init_task.signal->rlim[RLIMIT_NPROC]等

73 proc_caches_init：调用函数 kmem_cache_create 初始化进程相关 cache

I sighand_cachep = kmem_cache_create("sighand_cache", sizeof(struct sighand_struct), ...);

II signal_cachep = kmem_cache_create("signal_cache", sizeof(struct signal_struct), ...);

III files_cachep = kmem_cache_create("files_cache", sizeof(struct files_struct), ...);

IV fs_cachep = kmem_cache_create("fs_cache", sizeof(struct fs_struct), ...);

V vm_area_cachep = kmem_cache_create("vm_area_struct", sizeof(struct vm_area_struct), ...);

VI mm_cachep = kmem_cache_create("mm_struct", sizeof(struct mm_struct), ...);

74 buffer_init：

I 初始化专用缓存：bh_cachep = kmem_cache_create("buffer_head", sizeof(struct buffer_head), ...);

II 设置静态变量 max_buffer_heads

III hotcpu_notifier(buffer_cpu_notify, 0)：注册 CPU 启动通知链 buffer_cpu_notify

75 unnamed_dev_init：仅调用函数 idr_init(&unnamed_dev_idr)：

I init_id_cache()：初始化专用缓存：idr_layer_cache = kmem_cache_create("idr_layer_cache", sizeof(struct idr_layer), ...);

II unnamed_dev_idr 空间清 0，初始化自旋锁 unnamed_dev_idr.lock;

76 key_init：初始化密钥（key）管理

I 初始化专用缓存：key_jar = kmem_cache_create("key_jar", sizeof(struct key)

II 将 key_type_keyring.link、key_type_dead.link 和 key_type_user.link 加入到链表 key_types_list 尾部

III 各相关全局变量初始化

77 security_init：安全机制初始化

I 显示**标志信息**：printk(KERN_INFO "Security Framework v" SECURITY_FRAMEWORK_VERSION " initialized\n");

II verify(&dummy_security_ops)，调用函数 security_fixup_ops(&dummy_security_ops)，对 dummy_security_ops 中的每个成员调用宏 set_to_dummy_if_null，设置 dummy_security_ops 各成员初值指向对应的空函数 dummy_xx 函数（security/dummy.c 中）

III 初始化指针 security_ops = &dummy_security_ops

IV do_security_initcalls：调用[__security_initcall_start，__security_initcall_end]中的每个指针函数，进行安全机制初始化，即调用由宏 security_initcall(fn)初始化的函数集合。目前源代码中仅有函数 security/capability.c/capability_init()、security/selinux/hooks.c/selinux_init()和 security/root_plug.c/rootplug_init()

78 vfs_caches_init(num_physpages)：文件系统 VFS 层初始化

I 计算保留内存 reserve = min((num_physpages - nr_free_pages()) * 3/2, mempages - 1)，即保留已使用内存页面的 1.5 倍，以剩余内存 mempages = num_physpages – reserve 作为以下初始化时使用的内存容量；

II 专用缓存初始化：names_cachep = kmem_cache_create("names_cache", PATH_MAX, ...)

III 专用缓存初始化：filp_cachep = kmem_cache_create("filp", sizeof(struct file), ...);

IV dcache_init(mempages)：

i 专用缓存初始化：dentry_cache = kmem_cache_create("dentry_cache", sizeof(struct dentry), ...)

ii set_shrinker(DEFAULT_SEEKS, shrink_dcache_memory)：分配一个 struct shrinker 结构 shrinker，设置 shrinker->shrinker = shrink_dcache_memory，并将 shrinker->list 加入到全局链表 shrinker_list 中

iii dcache_init_early()中已经初始化了 dentry_hashtable，此处不再初始化，为什么要早期初始化？

V inode_init(mempages)：

i 专用缓存初始化：inode_cachep = kmem_cache_create("inode_cache", sizeof(struct inode), ...);

ii set_shrinker(DEFAULT_SEEKS, shrink_icache_memory)：作用同前一步

iii inode_init_early()中已经初始化 inode_hashtable，此处不再初始化

VI files_init(mempages)：设置全局变量 files_stat 中成员 max_files 初值，根据剩余内存容量计算

VII mnt_init(mempages)：

i 专用缓存初始化：mnt_cache = kmem_cache_create("mnt_cache", sizeof(struct vfsmount),...);

ii 为 mount_hashtable 分配一个页面，并初始化为哈希表表头

iii sysfs_init()：sysfs 文件系统初始化，需要开启编译选项 SYSFS

A 专用缓存初始化：sysfs_dir_cachep = kmem_cache_create("sysfs_dir_cache",

sizeof(struct sysfs_dirent), ...);

B register_filesystem(&sysfs_fs_type)：注册文件系统 sysfs_fs_type

C sysfs_mount = kern_mount(&sysfs_fs_type)：安装内部文件系统 sysfs，直接调用函数 do_kern_mount(type->name, 0, type->name, NULL)，后续工作参见文件系统安装部分

iv init_rootfs()：直接调用函数 register_filesystem(&rootfs_fs_type)，注册文件系统 rootfs_fs_type

v init_mount_tree()：初始化文件系统安装树

A mnt = do_kern_mount("rootfs", 0, "rootfs", NULL)：安装内部文件系统 rootfs

B 分配一个 struct namespace 结构 namespace 并初始化

C list_add(&mnt->mnt_list, &namespace->list)：将 rootfs 安装点加入 namespace 成员链表

D namespace->root = mnt，mnt->mnt_namespace = namespace：rootfs 安装点作为根目录

E 设置 init_task.namespace = namespace

F 对系统中的当前每个线程 p，设置 p->namespace = namespace，p 的类型为 task_struct

G set_fs_pwd：设置当前进程当前目录和安装点分别为 namespace->root 和 namespace->root->mnt_root

H set_fs_root：设置当前进程根目录和安装点分别为 namespace->root 和 namespace->root->mnt_root

VIII bdev_cache_init()：块设备初始化

i 专用缓存初始化：bdev_cachep = kmem_cache_create("bdev_cache", sizeof(struct bdev_inode), ...);

ii register_filesystem(&bd_type)：注册块设备文件系统 bd_type

iii bd_mnt = kern_mount(&bd_type)：内部安装块设备文件系统

iv blockdev_superblock = bd_mnt->mnt_sb：设置块设备超级块

IX chrdev_init()：直接调用 cdev_map = kobj_map_init(base_probe, &chrdevs_lock)

79 radix_tree_init：基数树初始化

I 专用缓存初始化：radix_tree_node_cachep = kmem_cache_create("radix_tree_node", sizeof(struct radix_tree_node), ...);

II radix_tree_init_maxindex()：初始化静态变量 height_to_maxindex[*]各分量

III hotcpu_notifier(radix_tree_callback, 0)：注册 CPU 启动通知链

80 signals_init()：信号初始化，仅初始化专用缓存 sigqueue_cachep =kmem_cache_create("sigqueue", sizeof(struct sigqueue), ...);

81 page_writeback_init()：初始化页面回写机制

I 根据内存容量设置相关全局变量 dirty_background_ratio 和 vm_dirty_ratio

II mod_timer(&wb_timer, ...)：修改 wb_timer 超时时间

III set_ratelimit()：设置限定参数 ratelimit_pages

IV register_cpu_notifier(&ratelimit_nb)：注册 CPU 启动链 ratelimit_nb

82 proc_root_init：proc 文件系统初始化

I proc_init_inodecache()：初始化专用缓存 proc_inode_cachep = kmem_cache_create("proc_inode_cache", sizeof(struct proc_inode), ...);

II register_filesystem(&proc_fs_type)：注册 proc 文件系统

III proc_mnt = kern_mount(&proc_fs_type)：内部安装 proc 文件系统

IV proc_misc_init()：建立/proc 目录下各常规文件，通常为只读属性

V 在目录/proc 下建立子目录：net、net/stat、sysvipc（需开启编译选项 SYSVIPI）、sys（需开启编译选项 SYSCTL）、fs、dirver、fs/nfsd、bus 等

VI proc_tty_init：初始化子目录/proc/tty

83 cpuset_init()：需要开启编译选项 CPUSETS，初始化 CPU 工作集

I 初始化全局变量 top_cpuset，设置 init_task.cpuset = top_cpuset

II register_filesystem(&cpuset_fs_type)：注册 CPU 工作集文件系统 cpuset

III cpuset_mount = kern_mount(&cpuset_fs_type)：内部安装 CPU 工作集文件系统

IV 设置 top_cpuset.dentry = cpuset_mount->mnt_sb->s_root，及相关成员

V cpuset_populate_dir(cpuset_mount->mnt_sb->s_root)：多次调用函数 cpuset_add_file 向该 dentry 中增加相关文件

84 check_bugs()：体系结构相关功能进一步初始化

I identify_cpu(c = &boot_cpu_data)：CPU 相关参数进一步初始化

i early_identify_cpu(c)：CPU 初步识别，第一次设置 phys_proc_id[smp_processor_id()]为 APIC_ID

ii cpuid_eax()：进一步识别 CPU

iii init_amd(c)：若是 AMD 处理器

A get_model_name(c)：设置 CPU 类型记录 boot_cpu_data.x86_model_id 值

B display_cacheinfo(c)：显示 CPU Cache 信息

C 设置当前 CPU 核数到 c->x86_num_cores 中

D amd_detect_cmp(c)：检测 CPU 多核配置

a cpu_core_id[cpu] = phys_proc_id[cpu] & ((1 << bits)-1)：设置当前 CPU 在本封装内的 CPU 核 ID

b phys_proc_id[cpu] >>= bits：设置当前 CPU 本封装 ID（APIC_ID 去掉 CPU 核编号）

c 若 acpi_numa <= 0 则设置 cpu_to_node[cpu] = phys_proc_id[cpu]

d 显示**标志信息**：printk(KERN_INFO "CPU %d(%d) -> Node %d -> Core %d\n", …);

iv init_intel(c)：若是 INTEL 处理器

v display_cacheinfo(c)：若是其它公司的 X86 处理器

vi select_idle_routine(c)：如果 CPU 支持 Monitor/Mwait support 特性且 pm_idle 为空，设置 pm_idle = mwait_idle

注：*命令行参数 idle="poll"将 pm_idle = poll_idle*

vii detect_ht(c)：设置 Hyper-Threading 功能，需要开启编译选项 SMP

A cpuid(1,...)：获得每个 CPU 封装内的 sibling 数，**可能是以超线程 CPU 计算**

B 设置全局变量 phys_proc_id[smp_processor_id()] = phys_pkg_id(index_msb);

C 显示**标志信息**：printk(KERN_INFO "CPU: Physical Processor ID: %d\n", phys_proc_id[cpu]);

D 设置全局变量 cpu_core_id[smp_processor_id()]

viii mcheck_init(c)：需要开启编译选项 X86_MCE，

A mce_init：MCE（Machine Check Exception）功能初始化，调用函数 do_machine_check 等，并访问 CPU 内部相关寄存器

B mce_cpu_features：如果是 INTEL 处理器，则调用函数 mce_intel_feature_init()→intel_init_thermal()进一步初始化

II 如果没有开启编译选项 SMP，显示 CPU 特征**标志信息**：printk("CPU: ");...

III alternative_instructions()：调用函数 apply_alternatives(__alt_instructions, __alt_instructions_end), 用 CPU 结构相关的快速指令替换区间[__alt_instructions, __alt_instructions_end]中原指令，链接脚本指出替换区位置。若命令行参数包含 noreplacement，则不执行上述替换功能。

85 acpi_early_init：ACPI 早期初始化，

86 rest_init()：准备执行内核的下一步初始化功能

I kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND)：创建 init 内核线程

II numa_default_policy()：调用函数 sys_set_mempolicy(MPOL_DEFAULT, ...)设置 NUMA 结构内

存分配策略为默认值 MPOL_DEFAULT

III unlock_kernel()：内核解锁

IV preempt_enable_no_resched()：宏，barrier(); dec_preempt_count()：内核进入抢占式调度状态

V schedule()：进行一次进程调度

VI cpu_idle()：进程 0 开始执行 idle 进程

1.4 init 进程

87 lock_kernel()：内核加锁

88 set_cpus_allowed(current, CPU_MASK_ALL)：设置 init 进程允许在全部 CPU 上运行

I rq = task_rq_lock(p=*current*, &flags)：获取当前进程（init）所在的运行队列 rq

II cpus_intersects(new_mask=*CPU_MASK_ALL*, cpu_online_map)：测试新 CPU 掩码字与在线 CPU（cpu_online_map）掩码字是否为空，为空则出错

III p->cpus_allowed = new_mask：设置进程新的 CPU 掩码字

IV cpu_isset(task_cpu(p), new_mask)：测试进程 init 当前运行的 CPU 是否在新的掩码字中，是则成功推出

V migrate_task(p, any_online_cpu(new_mask), req)：进程 init 当前运行的 CPU 不在新的掩码字中，迁移进程 init 到新的 CPU 掩码字中的任何一个在线 CPU，

i 若 init 不在运行队列中（p->array= NULL && task_running(rq, p) == NULL）

A 调用函数 set_task_cpu()设置 p->thread_info->cpu = new_mask，

ii 若 init 在运行队列中，填写迁移命令 req，

A 将 req->list 加入 rq 的迁移命令能够队列 rq->migration_ queue 中

B wake_up_process(rq->migration_thread)：唤醒迁移线程

C wait_for_completion(&req.done)：等待迁移结束

D tlb_migrate_finish(p->mm)：迁移结束，刷新 TLB

89 child_reaper = current

90 smp_prepare_cpus(max_cpus)：准备启动 SMP 中的其它 CPU，参数 maxcpus 缺省值为 NR_CPUS，启动参数“maxcpus=xx”将重新设置参数 maxcpus 值为 xx：

I nmi_watchdog_default()：设置全局变量 nmi_watchdog，如果当前不是缺省值 NMI_DEFAULT，则直接返回（命令行参数 nmi_watchdog=xxx 将设置 nmi_watchdog 值），否则如果是 INTEL 或 AMD CPU 且类型（boot_cpu_data.x86）参数为 15，则设置 nmi_watchdog = NMI_LOCAL_APIC，否则设置 nmi_watchdog = NMI_IO_APIC

II current_cpu_data = boot_cpu_data：设置当前 CPU 特征参数，开启编译选项 SMP 时宏 current_cpu_data 定义为 cpu_data [smp_processor_id()]

III current_thread_info()->cpu = 0

IV enforce_max_cpus(max_cpus)：清除大于 max_cpus 的 CPU 在全局变量 cpu_possible_map 和 cpu_present_map 中的标记，使大于 max_cpus 的 CPU 号不可用

V prefill_possible_map()：需要开启编译选项 HOT_PLUG_CPU，将系统支持的 NR_CPUS 个 CPU 全部加入到 cpu_possible_map 中

VI smp_sanity_check(max_cpus)：验证 SMP 配置可行性，若失败则关闭 SMP 功能

i 测试当前 BP 是否在全局变量 phys_cpu_present_map 中已经标记，未标记则再次标记

ii 若 smp_found_config 为 0，则 SMP 配置失败，直接返回

iii boot_cpu_id 是否已经在全局变量 phys_cpu_present_map 中已经标记，未标记则再次标记

iv 是否存在本 APIC

VII connect_bsp_APIC()：若当前已是 APIC 模式则无动作，否则由当前 PIC 模式切换到 APIC 模式：调用函数 clear_local_APIC()复位本地 APIC，通过端口 22h 和 23h 写数据

VIII setup_local_APIC()：本地 APIC 初始化

IX setup_IO_APIC()：启动 I/O APIC：

i enable_IO_APIC()：

A 初始化全局变量 irq_2_pin[*]

B 如果没有命令行参数设置 pirq=xx，则初始化全局变量 pirq_entries[*]

C 根据全局变量 nr_ioapics 值访问各 IOAPIC，设置各 IOAPIC 中的中断引脚数目全局变量 nr_ioapic_registers[*]，全局变量 nr_ioapics 值由函数 mp_register_ioapic()和 MP_ioapic_info()设置

D clear_IO_APIC()：对全部 IOAPIC 的各引脚调用函数 clear_IO_APIC_pin()清除中断

ii 如果 ACPI 分析函数 acpi_process_madt()**已经**设置了 IOAPIC（acpi_ioapic=1），则设置 io_apic_irqs = ~0，即全部 IRQ 都通过 IOAPIC，否则设置 io_apic_irqs = ~ PIC_IRQS

iii 如果 ACPI 分析函数 acpi_process_madt()**没有**设置了 IOAPIC（acpi_ioapic=0），调用函数 setup_ioapic_ids_from_mpc 从 MPC 表中分析 IOAPIC，设置全部 IOAPIC 相关寄存器，显示**标志信息**：printk(KERN_INFO "Using IO-APIC %d\n", mp_ioapics[apic].mpc_apicid)

iv sync_Arb_IDs()

v setup_IO_APIC_irqs()：设置 IOAPIC 各引脚中断向量，

注 1：*命令行参数"apic=debug"或"apic=verbose"可以是内核启动过程中打印 APIC 相关信息，其中 debug 时打印更多*

注 2：*中断引脚数目参见结构 union IO_APIC_reg_01{}定义，IOAPIC 其它寄存器参见系列结构 IO_APIC_reg_xx{}定义*

注 3：***IOAPIC 中断引脚寄存器不能有空行，参见函数 setup_IO_APIC_irqs()***

vi init_IO_APIC_traps()：初始化 IOAPIC 中断向量入口，对于小于 16 的中断向量，调用函数 make_8259A_irq，否则设置 irq_desc[*].handler=no_irq_type

vii check_timer()：验证定时器，代码比较复杂

viii print_IO_APIC()：如果如果 ACPI 分析函数 acpi_process_madt()**没有**设置 IOAPIC（acpi_ioapic=0），打印当前 IOAPIC 设置

X setup_boot_APIC_clock()：设置 BP 中的 APIC 定时器

i 显示**标志信息**：printk(KERN_INFO "Using local APIC timer interrupts.\n")

ii calibration_result=calibrate_APIC_clock()：标定本地 APIC 时钟，显示**标志信息**：printk(KERN_INFO "Detected %d.%03d MHz APIC timer.\n",result / 1000 / 1000, result / 1000 % 1000);

iii setup_APIC_timer(calibration_result)：启动本地 APIC 中的定时器

91 do_pre_smp_initcalls()：

I migration_init()：

i migration_call(&migration_notifier, CPU_UP_PREPARE, cpu)：

A p = kthread_create(migration_thread, hcpu, "migration/%d",cpu)：为当前 CPU 创建进程迁移线程，名字为 migration*n*，n 为 CPU 号，这里 n=0，线程的入口函数为 migration_thread

B kthread_bind(p, cpu)：帮定迁移线程 migration*n* 只能运行于当前 CPU

C set_task_cpu(p, cpu)：p->thread_info->cpu = cpu

D p->cpus_allowed = cpumask_of_cpu(cpu)

E __setscheduler(p, SCHED_FIFO, MAX_RT_PRIO-1)：设置调度策略 FIFO 和调度优先级极高

F cpu_rq(cpu)->migration_thread = p：设定当前 CPU 的迁移线程为刚创建的线程

ii migration_call(&migration_notifier, CPU_ONLINE, cpu)：仅调用函数 wake_up_process (cpu_rq(cpu)->migration_thread)，唤醒当前 CPU 的迁移线程

iii register_cpu_notifier(&migration_notifier)：调用函数 notifier_chain_register(&cpu_chain, nb)

注册通知块 migration_notifier 到 CPU 启动/关闭通知链

II spawn_ksoftirqd()：

i cpu_callback(&cpu_nfb, CPU_UP_PREPARE, cpu)

A p = kthread_create(ksoftirqd, hcpu, "ksoftirqd/%d", hotcpu)：为当前 CPU 创建核心线程 ksoftirqd*n*，n 为 CPU 号，入口函数为 ksoftirqd

B kthread_bind(p, hotcpu)：帮定线程 ksoftirqd*n* 只能运行于当前 CPU

C per_cpu(ksoftirqd, hotcpu) = p：标记线程 p 到 CPU 管理结构中

ii cpu_callback(&cpu_nfb, CPU_ONLINE, cpu)：仅调用函数 wake_up_process (per_cpu(ksoftirqd, hotcpu))，唤醒当前 CPU 的软中断处理线程

iii register_cpu_notifier(&cpu_nfb)：注册通知块 cpu_nfb 到 CPU 启动/关闭通知链

92 fixup_cpu_present_map()：如果全局变量 cpu_present_map 为空，则将 cpu_possible_map 中记录的每个 CPU 标记到 cpu_present_map 中

93 smp_init()：SMP 功能启动其它 AP

I cpu_up(i)：对于 cpu_present_map 中记录的每个 CPU，如果当前 CPU *i* 未启动，且当前已启动的总 CPU 数目少于全局变量规定 max_cpus 个，则调用函数 cpu_up(i)启动当前 CPU：

i notifier_call_chain(&cpu_chain, CPU_UP_PREPARE, hcpu)：执行 CPU 启动链 cpu_chain 中的每个通知块

ii __cpu_up(i)：启动第 i 个 CPU

A apicid = cpu_present_to_apicid(i)：根据全局变量 bios_cpu_apicid[i]的记录获得当前 CPU 的物理 ID

B per_cpu(cpu_state, i) = CPU_UP_PREPARE：设置 CPU 为准备启动状态

C do_boot_cpu(i, apicid)：启动一个 CPU，逻辑序号 i，物理 ID：apicid

a 定义一个空闲线程管理结构 c_idle

b 定义一个工作队列结构 work：执行函数为 do_fork_idle

c c_idle.idle = get_idle_for_cpu(i)：获取当前 CPU 的空闲进程控制结构

d 如果 idle 进程已经创建（c_idle.idle ≠ 0），设置堆栈 c_idle.idle->thread.rsp，调用函数 init_idle 初始化 idle 进程：设置为极低优先级，设置掩码智能运行于当前 CPU，加入到当前 CPU 的调度队列中

e 如果 idle 进程未创建（c_idle.idle = 0），调用函数 schedule_work(&work)利用工作队列创建 idle 进程或调用函数 do_fork_idle（即 work.func 指向）直接创建 idle 进程，并设置刚创建的 idle 进程到 idle 进程管理结构：即调用宏 set_idle_for_cpu(i, c_idle.idle) 设置 idle_thread_array[i] = c_idle.idle，do_fork_idle()：调用函数 fork_idle()：

(I) task = copy_process(CLONE_VM, 0…)；复制当前 init 进程作为新 idle 进程

(II) init_idle(task, cpu)：设置新创建的 idle 进程参数

(i) rq = cpu_rq(cpu)：获取当前 CPU 的运行队列指针

(ii) idle->cpus_allowed = cpumask_of_cpu(cpu)：设置 idle 进程仅能在当前 CPU 上运行

(iii) set_task_cpu(idle, cpu)：设置 p->thread_info->cpu = cpu，

(iv) rq->curr = rq->idle = idle：设置当前 CPU 的运行队列的空闲进程

(III) unhash_process(task)：从进程 pid 哈希表删除新创建的 idle 进程

f 设置 CPU i 的当前进程为 idle 进程：cpu_pda[i].pcurrent = c_idle.idle

g start_rip=setup_trampoline()：获取跳板代码物理地址 SMP_TRAMPOLINE_BASE（6000h），并将跳板代码[trampoline_data, trampoline_end]复制到跳板区 6000h

h init_rsp = c_idle.idle->thread.rsp：修改 head.S 文件中定义的当前进程堆栈为 idle 进程堆栈

i per_cpu(init_tss,cpu).rsp0 = init_rsp：设置 TSS 结构

j initial_code = start_secondary：修改 head.S 文件中定义的第一个 C 代码函数入口为 start_secondary

k clear_ti_thread_flag(c_idle.idle->thread_info, TIF_FORK)：清除 idle 进程的 TIF_FORK 标志

l 显示标志信息：printk(KERN_INFO "Booting processor %d/%d APIC 0x%x\n", …);

m CMOS_WRITE(0xa, 0xf); *((unsigned short *) phys_to_virt(0x469)) = start_rip >> 4; *((unsigned short *) phys_to_virt(0x467)) = start_rip & 0xf： CMOS 地址 0xF 写为 0xA：指示当前系统状态为“当机复位从 40:67h 处开始执行”，即执行代码 start_rip

n wakeup_secondary_via_INIT(apicid, start_rip)：通过处理器间中断（IPI）向目标 CPU 发送启动命令，且目标 CPU 起始代码地址为 start_rip，即跳板代码

o BP 循环检测当前 AP 是否启动成功，成功返回 0，否则返回失败代码，以下给出 AP 的启动过程，从实模式汇编代码 trampoline_data 开始：

① 当前 CS:IP 值为 0600:0000

② 设置 DS 为 0600，代码段、数据段在一起

③ movl $0xA5A5A5A5, trampoline_data - r_base：将跳板代码最初位置写为标记值 A5A5_A5A5h，以便通知运行

④ 设置 idt/gdt

⑤ %ax = 1，lmsw %ax：进入保护模式

⑥ ljmpl $__KERNEL32_CS, $(startup_32-__START_KERNEL_map)：跳转到文件 head.S 中 startup_32 处开始执行

⑦ 设置 CR3、CR4 等，进入长模式

⑧ 从 startup_64 开始执行 64 位代码，设置 CR3，使用页表 init_level4_pgt

⑨ 设置堆栈为 init_rsp，即前面步骤 h 设置的 idle 进程堆栈

⑩ 跳转到 C 代码 initial_code 处执行，即前面步骤 j 设置的函数 start_secondary()处执行

⑪ cpu_init()：CPU 初始化，调用函数 pda_init(cpu)，初始化本 AP 的 pda 结构，设置 pda->cpunumber = cpu，以后可以使用函数 smp_processor_id()获取 CPU 号，并显示标志信息：printk("Initializing CPU#%d\n", cpu)

⑫ smp_callin()：向 BP 报告本 AP 已经开始运行

(I) cpuid = smp_processor_id()：获取当前 CPU 逻辑号

(II) setup_local_APIC()：设置当前 AP 本地 APIC

(III) calibrate_delay()：标定当前 AP 性能 bogmips

(IV) disable_APIC_timer()：关闭当前 APIC 定时器

(V) smp_store_cpu_info(cpuid)：存储当前 AP 信息

(i) 复制 boot_cpu_data 结构到 cpu_data[cpuid]中

(ii) identify_cpu(cpu_data+cpuid)：

(VI) cpu_set(cpuid, cpu_callin_map)：将当前 CPU ID 标记到全局变量 cpu_callin_map 中

⑬ setup_secondary_APIC_clock()：调用函数 setup_APIC_timer(calibration_result)设置当前 AP 的本地 APIC 定时器

⑭ 如果 nmi_watchdog 使用 NMI_IO_APIC，则使用 LVT0 作为 NMI 使用

⑮ enable_APIC_timer()：本地 APIC 定时器使能

⑯ set_cpu_sibling_map(smp_processor_id())：

⑰ tsc_sync_wait()：调用函数 sync_tsc(0)同步 TSC

⑱ cpu_set(smp_processor_id(), cpu_online_map)：设置当前 CPU 到全局变量

cpu_online_map 中

⑲ cpu_idle()：当前 AP 进入 idle 状态

(I) while(1){}：本函数为 1 死循环，以下全部功能均在该循环体内

(II) while(!need_resched()){}：如果没有调度需求，则执行下面的空闲操作，否则执行 schedule()函数，执行其它有效进程，循环内部操作：

(i) 如果 cpu_is_offline(smp_processor_id())为真，则执行函数 play_dead()

(A) idle_task_exit()：如果当前 active_mm 不是 init_mm，则切换 active_mm 为 init_mm

(B) mmdrop(init_mm)：释放 init_mm 结构

(C) 设置当前 CPU 为 CPU_DEAD 状态

(D) 进入死循环，执行函数 safe_halt()，即汇编指令("sti; hlt": : :"memory")

(ii) 如果 pm_idle 不为空，则执行 pm_idle()，否则执行函数 default_idle()

D while(!cpu_isset(I, cpu_online_map))：等待 AP 启动完成，即等待正在启动的 AP 设置全局变量 cpu_online_map，参见 AP 启动第⑱步。

iii notifier_call_chain(&cpu_chain, CPU_ONLINE, hcpu)：标记当前 CPU 启动完成，可以正常运行

II 显示**标志信息**：printk(KERN_INFO "Brought up %ld CPUs\n", (long)num_online_cpus())

III smp_cpus_done(max_cpus)：结束 SMP 中的 AP 启动

i zap_low_mappings()：当未开启编译选项 HOTPLUG_CPU 时执行，清除页表 init_level4_pgt 中虚地址 0 表项，即通过 init_mm 不能再访问用户空间，并调用函数 flush_tlb_all()刷新全部 TLB

ii smp_cleanup_boot()：CMOS 寄存器 0xF 清 0，物理地址 467h 清 0

iii 当未开启编译选项 HOTPLUG_CPU，则释放第 1 页（1000h）和 SMP 跳板页（6000h）

IV setup_ioapic_dest()：调用函数 set_ioapic_affinity_irq()设置每个 IOAPIC 中断引脚与 CPU 的亲和性

V time_init_gtod()：设置时间结构，设置指针函数 do_gettimeoffset 值，显示**标志信息**：printk(KERN_INFO "time.c: Using %s based timekeeping.\n", timetype)

VI check_nmi_watchdog()：效验 NMI 看门狗的有效性

i 显示**标志信息**：printk(KERN_INFO "testing NMI watchdog ... ")

ii 执行验证看门狗操作

iii 显示**标志信息**：printk("OK.\n")

94 sched_init_smp()：SMP 结构调度处理

I arch_init_sched_domains(&cpu_online_map)：建立调度域

i check_sibling_maps()：需要同时开启编译选项 SCHED_SMT 和 NUMA 才执行，检查位于同一个 CPU 内的 SMT 单元是否属于同一个 CPU NODE

ii cpus_andnot(cpu_default_map, cpu_online_map, cpu_isolated_map)：设置有效 CPU 掩码图，即 cpu_default_map = cpu_online_map & ~ cpu_isolated_map

iii build_sched_domains(&cpu_default_map)：建立有效 CPU 调度域

A 对于 cpu_default_map 中的每个 CPU 单元执行下述操作：

a nodemask = node_to_cpumask(cpu_to_node(i))：获取本 CPU 单元所在 NODE 的 CPU 掩码

b 获取第 i 个 CPU 对应的 node_domains 字段地址 sd，

c groupe = cpu_to_node_group(i)：获取第 i 个 CPU 单元对应的 NODE 组

d 设置 sd->span = *cpu_map：本调度域包含全部有效处理器

e 设置 sd->groups = &sched_grpup_nodes[group]
f 本调度域需要开启编译选项 NUMA

g p = sp：保存上一个调度域
h 获取第 i 个 CPU 对应的 phys_domains 字段地址 sd
i group = cpu_to_phys_groups(i)：获取第 i 个 CPU 单元对应 CPU 封装组
j sd->span = nodemask：本调度域仅包含本 NODE 内的 CPU 单元
k sd->parent = p：设置本调度的双亲调度域为前面的 NODE 调度域
l sd->groups= &sched_group_phys[group]：设置调度域中的调度组

m p = sd
n 获取第 i 个 CPU 对应的 cpu_domains 字段地址 sd
o group = cpu_to_cpu_groups(i)：获取第 i 个 CPU 单元对应 CPU 核内组
p sd->span = cpu_sibling_map[i]：本调度域仅包含本 CPU 核内内的 CPU 单元
q sd->parent = p：设置本调度的双亲调度域为前面的 CPU 封装调度域
r sd->groups= &sched_group_cpus[group]：设置调度域中的调度组
s 本调度域需要开启编译选项 SCHED_SMT

B 对每个在先 CPU 单元，调用函数 init_sched_build_groups(sched_group_cpus, this_sibling_map)：将 this_sibling_map 中的每个 CPU 不同的 sched_group_cpus 连成一个环形链表
C 对于系统中的全部 NODE，将不属于同一个 NODE 的 CPU 对应的 sched_group_phys 连成一个环形链表
D 将全部在线 CPU 对应的 sched_group_nodes 连成一个环形链表
E 对于系统中的每个 CPU 单元设置计算能力，包括 cpu_domain、phys_domains 和 node_domanis 中的 groups->cpu_power
F 对于每个在线 CPU，调用函数 cpu_attach_domain（sd,i)：设置当前 CPU 的调度域为 sd

II hotcpu_notifier(update_sched_domains, 0)：设置热插拔 CPU 调度域更新通知块

95 cpuset_init_smp()：设置全局变量 top_cpuset.cpus_allowed 为 cpu_online_map，设置 top_cpuset.mems_allowed = node_online_map

96 populate_rootfs()：
I unpack_to_rootfs(__initramfs_start, __initramfs_end - __initramfs_start, 0);：initramsfs 解压缩
II 设置 initrd，内容较多

97 do_basic_setup()
I init_workqueues()：
i hotcpu_notifier(workqueue_cpu_callback, 0)：设置热插拔 CPU 通知块 workqueue_cpu_callback
ii keventd_wq = create_workqueue("events")：建立工作队列 keventd_wq，并为每个 CPU 单元创建一个队列工作线程
II usermodehelper_init()：直接执行 khelper_wq = create_singlethread_workqueue("khelper")，创建一个内核帮助线程
III driver_init()：
i devices_init()：直接调用函数 subsystem_register(&devices_subsys)，注册设备子系统 devices_subsys
ii buses_init()：直接调用函数 subsystem_register(&bus_subsys)，注册总线子系统 bus_subsys
iii classes_init()：subsystem_register(&class_subsys) 注册类型子系统 class_subsys，

subsystem_init(&class_obj_subsys)初始化子系统 class_obj_subsys

iv firmware_init()：直接调用函数 subsystem_register(&firmware_subsys)，注册固件子系统 firmware_subsys

v platform_bus_init()：直接调用函数 device_register(&platform_bus) 和 bus_register(&platform_bus_type)

vi system_bus_init()：

vii cpu_dev_init()：直接调用函数 sysdev_class_register(&cpu_sysdev_class)

viii attribute_container_init()：初始化全局链表 attribute_container_list

IV sysctl_init()：需要同时开启编译选项 SYSCTL 和 PROC_FS 才有效，

i register_proc_table(root_table, proc_sys_root)：为 root_table 表中每个表项注册一个 proc 项

ii init_irq_proc()：注册目录/proc/irq，并为每个有效中断向量（irq_desc[irq].handler != &no_irq_ type）注册 smp_affinity 文件：/proc/irq/xx/smp_affinity，通过该文件设置对应中断向量的亲和性

V sock_init()：网络协议（BSD socket）初始化

i sk_init()：根据内存容量设置相关全局变量

ii skb_init()：sk_buff 专用缓存初始化，skbuff_head_cache = kmem_cache_create ("skbuff_ head_cache", sizeof(struct sk_buff)：

iii init_inodecache()：专用缓存初始化 sock_inode_cachep = kmem_cache_create("sock_inode_ cache", sizeof(struct socket_alloc)

iv register_filesystem(&sock_fs_type)：注册 sockfs 文件系统

v sock_mnt = kern_mount(&sock_fs_type)：核心安装文件系统 sockfs

vi netfilter_init()：需要开启编译选项 NETFILTER，网络过滤器初始化，初始化链表 nf_hooks[i][j]

VI do_initcalls()：执行[__initcall_start，__initcall_end]之间的初始化函数，即执行由宏 core_initcall(fn)、postcore_initcall(fn)、arch_initcall(fn)、subsys_initcall(fn)、fs_initcall(fn)、device_initcall(fn)、late_initcall(fn)、__initcall(fn)定义的初始化函数

98 如果文件/init 存在，则设置用户空间初始化程序为/init，否则调用函数 prepare_namespace()，通常文件/init 不存在，而需要执行函数 prepare_namespace()

I mount_devfs()：直接调用 sys_mount("devfs", "/dev", "devfs", 0, NULL)，安装 devfs

II md_run_setup()：设置 RAID

i create_dev("/dev/md0", MKDEV(MD_MAJOR, 0), "md/0")，建立设备/dev/md0

ii md_setup_drive()：

III initrd_load()：读入并使用 initrd

IV mount_root()

99 free_initmem()：释放初始化过程中使用的内存

100 numa_default_policy()：设置 NUMA 调度策略

101 sys_open("/dev/console")打开控制台

102 sys_dup(0)，sys_dup(0)：设置标准输出、标准出错

103 run_init_process("/sbin/init")：启动 init 进程

2 文件系统

2.1 mount 操作

2.1.1 物理文件系统（以 ext2 为例）

1 module_init(init_ext2_fs)：启动安装 ext2 文件系统

I init_ext2_xattr()：需要开启编译选项 EXT2_FS_XATTR，直接调用函数 ext2_xattr_cache = mb_cache_create("ext2_xattr", …)初始化文件系统元数据专用缓存 Filesystem Meta Information Block Cache (mbcache)

II init_inodecache()：初始化专用缓存 ext2_inode_cachep = kmem_cache_create("ext2_inode_cache", sizeof(struct ext2_inode_info)

III register_filesystem(fs = &ext2_fs_type)：注册 ext2 文件系统

i 初始化 fs->fs_supers 链表为空

ii file_system_type** p = find_filesystem(fs->name)：在以 file_systems 为表头的单向链表中搜索已注册文件系统中是否有名字为“ext2”的，并返回最后一个节点的 next 域地址

iii *p = fs：将 ext2_fs_type 结构加入到文件系统类型链表中

注：*1 若在模块中定义，即开启编译选项 MODULE，则 module_init(initfn)定义成*

static inline initcall_t __inittest(void) { return initfn;} // 定义一个函数返回函数 initfn 地址

int init_module(void) __attribute__((alias(#initfn)) // 定义函数 initfn 的别名函数 init_module

2 若在模块中定义，即关闭编译选项 MODULE，则 module_init(initfn)定义成“__initcall(initfn)”，而“__initcall(initfn)”定义为“device_initcall(initfn)”，最后“device_initcall(initfn)”定义为 __define_initcall("6", initfn)，最后由 do_initcalls()：执行[__initcall_start，__initcall_end]区初始化

2 命令“mount –t ext2 device dir”：安装一个 ext2 文件系统设备，执行 **sys_mount** 系统调用；

3 sys_mount()：

I copy_mount_options()：将用户空间参数复制到核心空间，每个参数占一个页面空间

II getname(dir_name)：复制安装点路径名

i tmp = __getname() = kmem_cache_alloc(names_cachep, SLAB_KERNEL)：从缓存 names_cachep 中分配内存，names_cachep 初始化时 SLAB 容量为 PATH_MAX（4KB）

ii do_getname(dir_name, tmp)：地址边界检查，防止非法内容进入核心空间，调用函数 strncpy_from_user()执行复制操作

III lock_kernel()：禁止内核抢占调度

IV **do_mount**()：执行安装操作

V unlock_kernel()：允许内核抢占调度

4 do_mount(dev_name, dir_name, type_page, flags, data_page)：执行安装操作实体

I 验证输入参数的有效性

II 分析 flags 参数

III **path_lookup**(dir_name, LOOKUP_FOLLOW, &nd)：查找目标安装点 nameidata 结构

IV **do_remount**()：重新安装，命令中包含 MS_REMOUNT 标记

V **do_lookback**()：安装回环设备，命令中包含 MS_BIND 标记

VI **do_move_mount**()：删除安装，命令中包含 MS_MOVE 标记

VII **do_new_mount**(&nd, type_page, flags, mnt_flags, dev_name, data_page)：新安装

VIII path_release(&nd)：释放 nd

注：*struct nameidata 结构说明：*

*struct dentry * dentry：指向目标点 dentry 结构*

*struct vfsmount *　mnt：指向目标点所在设备安装点*

strct qstr　　last：用于索引

5　do_new_mount()：安装一个新设备；

I　检验输入参数的正确性

II　capable(CAP_SYS_ADMIN)：验证进程是否具有管理员权限

III　mnt = **do_kern_mount**(&nd, type_page, flags, mnt_flags, dev_name, data_page)：执行内核安装

IV　do_add_mount(mnt, nd, …)：将一个安装点安装到 namespace 中的安装树中

i　while(d_mountpoint(nd->dentry) && follow_down(&nd->mnt, &nd->dentry))：

A　d_mountpoint(nd->dentry)：return nd->dentry->d_mounted，即返回当前 dentry 结构已安装的目录个数

B　follow_down：前进到已安装设备上的根节点，并且通过 while 循环进一步检测新的安装点，直到尽头，即前进到不再有设备安装的某个设备上的根节点为止。

ii　if (nd->mnt->mnt_sb==newmnt->mnt_sb && nd->mnt->mnt_root==nd->dentry) goto fail：同一个设备文件系统不能在同一个目录重复安装

iii　mnt->mnt_namespace = current->namespace

iv　graft_tree(mnt, nd)

A　if (S_ISDIR(nd->dentry->d_inode->i_mode) != S_ISDIR(mnt->mnt_root->d_inode->i_mode)) return：都必须是目录

B　nd->dentry 或者是根目录，或者 d_flags 中没有标记 DCACHE_UNHASHED，才能执行后续操作，文件系统不能安装在一个带有标记 DCACHE_UNHASHED 的目录上

C　attach_mnt(mnt, nd)：完成安装点连接操作

a　mnt->mnt_parent = mntget(nd->mnt)

b　mnt->mnt_mountpoint = dget(nd->dentry)

c　list_add(&mnt->mnt_hash, mount_hashtable+hash(..))：将本安装点加入全局安装点哈希表

d　list_add_tail(&mnt->mnt_child, &nd->mnt->mnt_mounts)：将本安装点加入到安装目录的安装点的字安装点链表中

e　nd->dentry->d_mounted++：增加安装目录的已安装计数

D　list_add_tail(&head, &mnt->mnt_list)在 mnt->mnt_list 中加入一个临时节点 head

E　list_splice(&head, current->namespace->list.prev)：将 head 所在链表（即 mnt->mnt_list）加入 current->namespace->list.prev 中

v　mntput(mnt)，释放计数器。至此，安装过程全部完成

6　do_kern_mount(type_page, flags, dev_name, data_page)：执行内部安装操作；

I　type = get_fs_type(fstype)：

i　type = find_filesystem(fstype)：在已注册文件系统中（file_systems 为表头的单向链表）中查找名字为 fstype 的文件系统

ii　若失败则试图加载文件系统 fstype 的模块，再次调用 find_filesystem 查找

II　mnt = alloc_vfsmnt(dev_name)：在专用缓存 mnt_cache 中分配 vfsmount 结构并初始化，分配内存用于保存设备名称 dev_name，并以 mnt->mnt_devname 指向之

III　sb = type->get_sb()：调用该文件系体的超级块读取函数，对于 ext2 文件系统，ext2_fs_type 中定义的 get_sb 为函数 ext2_get_sb()，而 ext2_get_sb()直接调用函数 **get_sb_bdev**(fs_type, flags, dev_name, data, ext2_fill_super)

IV　设置 vfsmount 结构 mnt：

i　mnt->mnt_sb = sb；

ii　mnt->mnt_root = sb->s_root；

iii　mnt->mnt_mount = sb->s_root：表明这是一个根安装

iv mnt->mnt_parent =mnt：表明这是一个根安装

V put_filesystem(type)：释放文件系统

VI return mnt

7 get_sb_bdev(..., ext2_fill_super)：设置超级块结构

I bdev = open_bdev_excl(dev_name, flags, fs_type)：打开块设备 dev_name，并且所有者为文件系统 fs_type

i bdev = lookup_bdev(dev_name)

A path_lookup(dev_name, …, &nd)：寻找设备 dev_name 的 nameidata 结构

B inode = nd.dentry->d_inode：设备 dev_name 的 inode 结构

C bdev = bd_acquire(inode)：获取本 inode 所属的 bdev_inode 结构的另外一个成员 block_device 结构 bdev

a 如果 inode->i_bdev 不为空且 inode 空闲，则直接返回 inode->i_bdev

b 否则调用函数 bdev = bdget(inode->i_rdev)：

(1) inode = iget5_locked(bd_mnt->mnt_sb, hash(dev), bdev_test, bdev_set, &i_rdev) 从 bdev(bd_type)，获取设备在 bdev 文件系统中的 inode

(I) head = inode_hashtable + hash(sb, hashval)：在 inode 哈希表中寻找块设备 indoe，设置哈希表表头，即 bdev_inode.vfs_inode

(II) inode = ifind(sb, head, bdev_test, …)，调用函数 find_inode()查找目标 indoe，查找失败，调用函数 get_new_inode()创建 inode

(III) get_new_inode()：创建 inode

(i) inode = alloc_inode(sb)：调用函数 sb->s_op->alloc_inode=bdev_alloc_ inode()从专用缓存 bdev_cachep 中分配一个 bdev_inode 数据结构 ei，并将其成员 ei->vfs_inode 作为 VFS inode 返回，并将该 inode 进行基本初始化，如 inode->i_mapping = inode->i_data

(ii) list_add(&inode->i_list, &inode_in_use)：将 inode 加入到 inode_in_ use 链表

(iii) list_add(&inode->i_sb_list, &sb->s_inodes)：将 inode 加入到 bd_mnt->mnt_sb->s_inodes 链表下

(2) bdev = &BDEV_I(inode)->bdev：获得与 inode 位于用一个 block_inode 下的 block_device 结构地址

(3) bdev->bd_inode = inode

(4) inode->i_rdev = dev：设置代表块设备的 inode 的目标设备号

(5) inode->i_bdev = bdev：块设备

(6) inode->i_data.a_ops = &def_blk_aops

(7) inode->i_data.backing_dev_info = &default_backing_dev_info

(8) list_add(&bdev->bd_linst, &all_bdevs)：记录块设备

注 1：*这里的 inode 是代表块设备的 inode，是结构 block_inode 中的一个成员，并与另外一个成员 block_device 结构之间有互指指针（bdev->bd_inode，inode->i_bdev），该 inode 也加入系统 inode 哈希表。*

注 2：*块设备管理：块设备通过通过虚拟文件系统 bdev(bd_type)进行管理，该文件系统由 start_kernel()→vfs_caches_init()→bdev_cache_init()注册，并进行核心安装（kern_mount()→do_kern_mount()）。每个块设备对应一个 bdev_inode 结构，通过虚拟文件系统 bdev 分配与释放，该结构中的成员 vfs_inode 作为设备 inode 加入 inode 哈希表中，按常规方式访问，并通过函数 struct bdev_inode* BDEV_I(inode)和 strcut block_device* I_BDDEV(inode)，将 bdev_indoe.vfs_inode 作为输入参数，来获取*

bdev_indoe 结构地址和另外一个成员 bdev 地址。

```
struct bdev_inode{
    struct block_device     bdev;
    struct inode            vfs_inode;
}
```

c inode->i_bdev = bdev，注：这里的 inode 是设备名（如 dev/hda1）对应的 inode

d inode->i_mapping = bdev->bd_inode->i_mapping，alloc_inode 赋初值外无其它初始化

e list_add(&inode->i_devices, &bdev->bd_inodes)：将代表设备的路径 inode 加入设备管理的 inode 链表在中

ii blkdev_get(bdev, mode, 0)：利用局部变量 fake_file 和 fake_dentry 执行 do_open(bdev, &fake_file)操作

disk = get_gendisk(bdev->bd_dev, &part)

未完待续

iii bd_claim(bdev, holder)：

II s = sget(fs_type, test_bdev_super, set_bdev_super, bdev)：分配一个超级块结构，并通过 set_bdev_super 函数设置 s->s_bdev = bdev，s->s_dev = s->s_bdev->bd_dev

III 调用函数 **ext2_fill_super**(fs_type, ...)：开始执行 ext2 文件系统操作接口

8 ext2_fill_super(struct super_block* sb, void* data, int silent)：读取设备填写超级块，设置超级块操作方法结构：

I 分配一个 ext2_sb_info 结构 sbi 并清 0，设置 sb->s_fs_info = sbi

II sb_block = get_sb_block(&data)：设置超级块的起始位置，若参数 data 中包含字符串"sb=xx"，则设置 sb_block=xx，否则设置为默认为 1

III 设置 logic_sb_block，超级块位置的逻辑块编号

IV bh = sb_read(sb, logic_sb_block)：读取超级块，直接调用函数__bread(sb->s_bdev, block, sb->s_blocksize)读取超级块：

i bh = __getblk(bdev, block, size)：未完待续，在 buffer_cache 查找目标 bh

A bh = __find_get_block(bdev, block, size)

a bh = lookup_bh_lru(bdev, block, size)：

(1) bh_lru_lock()：宏，开启 SMP 时定义为 local_irq_disable()，否则定义为 preempt_disable()

(2) lru = &__get_cpu_var(bh_lrus)：获取当前 CPU Buffer Cache 的 LRU 队列地址

(3) 在该 LRU 数组（BH_LRU_SIZE=8 个分量）中查找目标 Buffer Cache 块，即满足 bh->b_bdev = bdev && bh->b_blocknr = block && bh->b_size = size，找到则将该块提前到由 LRU 数组分量 0 指示之

(4) bh_lru_unlock()：宏，解锁

b bh = __find_get_block_slow(bdev, block, size)

(1) bd_mapping = bdev->bd_inode->i_mapping：设备 dev 的 address_space 结构

(2) index = block >>(PAGE_CACHE_SHIFT, bdev->bd_inode->i_blkbits)：将目标设备块转为 Buffer Cache 中的页面号

(3) page = find_get_page(bd_mapping, index)：调用函数 radix_tree_lookup (&mapping->page_tree, index)，在目标设备的基数树中查找目标块，若找到则调用宏 page_cache_get(page)→page_get(page)增加页面 page 的引用计数

(4) 若前一步返回 NULL，则推出，否则继续执行

(5) bh = page_buffers(page) = page->private：以 private 作为 Buffer Cache 头指针

(6) 检查当前页的全部 bh，若 bh->b_blocknr = block 则找到目标 Buffer Cache，

bh 加锁，解锁 page 结构

c 若 bh 找到则调用函数 bh_lru_install(bh)：将目标 bh 加入到当前处理器 Buffer Cache 的 LRU 队列前端，并可能释放到最末一个 BH

d toch_buffer(bh)：标记当前 bh 已经访问

B might_sleep()：可以进行一次进程调度

C bh = __getblk_slow(bdev, block, size)：利用一个无限循环分配一个 bh

a 再次调用函数__find_get_block(bdev, block, size)查找 bh

b grow_buffers(bdev, block, size)：为 Buffer Cache 分配一个物理页面，可用户缓存设备的 block 块，并加入到设备 bdev 的基数树中

ii bh = __bread_slow(bh)，若 bh 中的内容没有更新（可能是新分配的 bh），则执行从硬盘上读取目标 bh

A lock_buffer(bh)：锁定 bh

B 如果 bh 已经更新则直接返回

C get_bh(bh)：增加 bh 引用计数

D bh->b_end_io = end_buffer_read_sync：设置 io 完成处理函数

E submit_bh(READ, bh)：向驱动程序提交读请求，分配一个 struct bio 结构并进行相关设置，调用函数 submit_bio(READ, bio)向驱动程序提交请求

F wait_on_buffer(bh)：等待度完成

注释：*基数树（radix tree），Buffer Cache 使用 64 叉基数树管理，数据结构如下。树高最大 12，使用 bit63~60 作为索引，第 10 级使用 bit59~54 作为索引，以此类推，第 1 级使用 bit5~0 作为索引，叶子节点存放有效数据*

```
struct radix_tree_root{
    uint height;
    int gfp_mask;
    struct radix_tree_node* rnode
}
struct rasix_tree_node{
    uint count;
    void* slot[MAP_SIZE];      // 64
    ulong tags[TAGS][TAG_LONGS];//2, 1
}
```

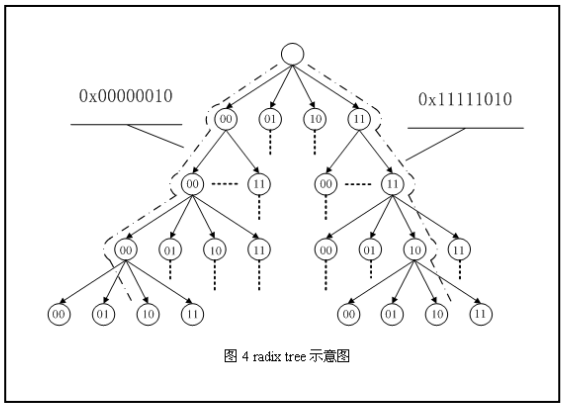

图 4 radix tree 示意图

V sbi 初始化，调用函数 parse_options(data, sbi)根据 data 中的参数进一步设置 sbi

VI 根据读取的超级块数据进行分析，进行相关设置，若块设备实际的分块与原读取时使用的不一致，则需要重新调用 sb_bread 函数读入一次超级块

VII 超级块其它参数初始化

VIII sb->s_export_op = &**ext2_export_ops**：只有 get_parent 和 get_dentry 两个操作方法

sb->s_op = &**ext2_sops**：定义 inode 操作方法及其它方法：

alloc_inode = ext2_alloc_inode　　destroy_inode = ext2_destroy_inode

read_inode = ext2_read_inode　　write_inode = ext2_write_inode

IX root = iget(sb, ino = EXT2_ROOT_INO)：在全局 inode 哈希表中查找目标 inode，若失败则分配一个新的 inode 并进行基本初始化

i iget_locked(sb, ino)：获取 inode 地址

A head = inode_headtable + hash(sb, ino)：准备在 inode 哈希表中查找目标 inode

B ifind_fast(sb, head, ino)：在哈希表中查找目标 inode

C inode = get_new_inode_fast(sb, head, ino)：

a inode = alloc_inode(sb)：调用函数 sb->s_op->alloc_inode(sb)分配一个 inode，即函

数 ext2_alloc_inode(sb)：从系统专用缓存 ext2_inode_cachep 中分配一个 ext2_inode_info 结构 ei，并将该结构的成员 vfs_inode 地址（&ei->vfs_inode）作为分配的 inode 结构返回。新 inode 通用初始化；

b 将 inode 加入 sb->s_inode 链表中和全局 inode 哈希表中

ii 调用函数 sb->s_op->read_inode→ext2_read_inode(inode)：读取 inode 内容

A ei = EXT2_I(inode)：获取 inode 所在结构 ext2_inode_info 的地址

B raw_inode = ext2_get_inode(inode->i_sb, ino, &bh)：读取目标 inode 在磁盘上的内容

a 根据超级块中的记录计算目标 inode 所在的组，并转为组内偏移量量和磁盘块号

b *bh = sb_bread(sb, block)：读取目标块，*bh 中保存 Buffer Cache 地址

c 计算 ext2_inode 在 Buffer Cache 中的偏移量

d return (struct ext2_inode *) (bh->b_data + offset)：返回目标地址

C inode->i_mode = raw_inode->i_mode，根据 inode->i_mode 判断 inode 类型作如下处理：

D 常规文件（S_ISREG(inode->i_mode)）：

a inode->i_op = &**ext2_file_inode_operations**;

b 若支持 XIP（excute in place，开启 EXT2_FS_XIP 编译选项）

(1) inode->i_mapping->a_ops = &ext2_aops_xip;

(2) inode->i_fop = &ext2_xip_file_operations;

c 文件系统不用 buffer head（安装时带"nobh"选项，即开启 EXT2_MOUNT_NOBH）

(1) inode->i_mapping->a_ops = &**ext2_nobh_aops**;

(2) inode->i_fop = &**ext2_file_operations**;

d 正常文件系统

(1) inode->i_mapping->a_ops = &**ext2_aops**;

(2) inode->i_fop = &**ext2_file_operations**;

E 目录（S_ISDIR(inode->i_mode)）：

a inode->i_op = &**ext2_dir_inode_operations**;

b inode->i_fop = &**ext2_dir_operations**;

c 文件系统不用 buffer head（安装时带"nobh"选项），则 inode->i_mapping->a_ops = &**ext2_nobh_aops**；否则 inode->i_mapping->a_ops = &**ext2_aops**;

F 符号链接（S_ISLNK(inode->i_mode)）：

a 快速符号连接（fast symlink）：inode->i_op = &**ext2_fast_symlink_inode_operations**

b 其它链接

(1) inode->i_op = &**ext2_**symlink**_inode_operations**;

(2) 文件系统不用 buffer head（安装时带"nobh"选项），inode->i_mapping->a_ops = &**ext2_nobh_aops**；否则 inode->i_mapping->a_ops = &**ext2_aops**

G 其它形式（设备等）

a inode->i_op = &**ext2_special_inode_operations**;

b 计算设备号：

(1) 如果 val=raw_inode->i_block[0]不等于 0，使用旧式 16 位设备号：rdev = MKDEV((val >> 8) & 255, val & 255)，即高 8 位主设备号和低 8 位次设备号

(2) raw_inode->i_block[0]等于 0 则使用新式 32 位设备号：dev = raw_inode->i_block[1]：major = (dev & 0xfff00) >> 8，minor = (dev & 0xff) | ((dev >> 12) & 0xfff00); rdev = MKDEV(major, minor)；即 bit19~8 为 12 位主设备号，bit31~20、bit8~7 为 20 位次设备号

c init_special_inode(inode, inode->i_mode, devt)：根据设备类型设置

(1) 字符设备：

inode->i_fop = &def_chr_fops

inode->i_rdev = rdev

(2) 块设备

inode->i_fop = &def_blk_fops

inode->i_rdev = rdev

(3) FIFO：

inode->i_fop = &def_fifo_fops

(4) SOCK：

inode->i_fop = &bad_sock_fops

(5) 其它则出错

注释：*XIP：The main value of XIP(eXecute In Place) lies in providing a means of allowing several copies of a program to be running without duplicating the text segment. Indeed the text segment can reside in flash memory and need not be copied to the system Ram at all. This is useful for tasks that have large program bodies with many executable instances running in the system.*

Only the Stack, BSS and data segments of an executable needs to be produced for each running program. The text segment can then reside in flash memory or, if execution speed is an issue, then copy the file system to ram first and mount it from there. If executables in the file system are compiled to support XIP and also flagged in their headers as XIP they will load and execute with just a single copy of the text segment.

(from http://www.ucdot.org/article.pl?sid=02/08/28/0434210)

(other definatinon: http://www.ucdot.org/article.pl?sid=02/08/28/0434210)

H ext2_set_inode_flags(inode)：设置 inode 相关标记

X sb->s_root = d_alloc_root(root)

i 设置路径名 struct qstr name{.name ="/", .len = 1}

ii res = d_alloc(NULL, name)：分配一个 dentry 结构 res，设置名称并初始化链表其它成员，res->d_op=NULL，res->d_flags = DCACHE_UNHASHED，表示未加入哈希表，加入 dentry 哈希表后去掉该标记。*ext2 文件系统好像未初始化 d_op 指针，始终为 NULL*

iii res->d_sb = root->i_sb：设置 dentry 结构中的超级块指针

iv res->d_parent = res：父结构指向自身，表明这是一个根节点

v d_instantiage(res, root)：将 dentry 结构与 inode 结构联系起来：

A list_add(&res->d_alias, &root->i_dentry)：将 dentry 结构以 d_alias 指针加入 inode 结构的 i_dentry 链表中

B res->d_inode = inode：dentry 结构的 d_inode 指针指向目标 inode

XI 文件系统判断：显示标志信息：printk(KERN_ERR "EXT2-fs: get root inode failed\n")；printk(KERN_ERR "EXT2-fs: corrupt root inode, run e2fsck\n")

XII ext2_setup_super (sb, es, sb->s_flags & MS_RDONLY)：

i 根据 sb、es 中内容进行相关判断，并显示相关信息

ii ext2_write_super(sb)：写超级块

iii 若开启编译选项 CONFIG_EXT2_CHECK 且包含有 EXT2_MOUNT_CHECK 安装标记（"check"）则执行 ext2_check_blocks_bitmap(sb)和 ext2_check_inodes_bitmap(sb)检验有效性

XIII 至此函数 get_sb_bdev()执行结束

2.1.2 内部虚拟文件系统

2.1.2.1 bdev 文件系统

该文件系统用于管理块设备，加载的过程为：start_kernel()→ vfs_caches_init()→bdev_cache_init()，并

进行核心安装（kern_mount()→do_kern_mount()，下面从函数 bdev_cache_init()开始分析

1 bdev_cachep = kmem_cache_create("bdev_cache", sizeof(struct bdev_inode)：为分配块设备专用管理结构 bdev_inode 建立专用缓存

2 register_filesystem(&bd_type)：注册专用文件系统 bdev（bd_type），将结构 bd_type 加入到已注册文件系统链表 file_systems 中

3 bd_mnt = kern_mount(type = &bd_type)：内部安装文件系统，直接调用函数 do_kern_mount (type->name, 0, type->name, NULL)，即文件系统类型为“bdev”，安装的目标设备名称也是“bdev”

I type = get_fs_type(fstype)

II mnt = alloc_vfsmnt(dev_name)：在专用缓存 mnt_cache 中分配 vfsmount 结构并初始化，分配内存用于保存设备名称 dev_name，并以 mnt->mnt_devname 指向之

III sb = type->get_sb()：调用该文件系统的超级块读取函数，对于 bdev 文件系统，bd_type 中定义的 get_sb 为函数 bd_get_sb()，而 bd_get_sb()直接调用函数 get_sb_pseudo(fs_type, "bdev:", &bdev_sops, …)：

i s = sget(fs_type, NULL, set_anon_super, NULL)获取超级块

A s = alloc_super()：分配一个超级块结构，并进行初始化

B set_anon_super(s, NULL)：利用 idr 数据结构产生一个 ID 用作此设备号，设置到 s->s_dev 中

a idr_pre_get(&unnamed_dev_idr,...)：在 unnamed_dev_idr 中准备一个节点

b idr_get_new(&unnamed_dev_idr, NULL, &dev)：在 unnamed_dev_idr 中申请一个节点，在 dev 中返回节点 ID，用作次设备号

c s->s_dev = MKDEV(0, dev)

C s->s_type = type：设置本超级块所属文件系统

D strlcpy(s->s_id, type->name, ...)：设置超级块所属文件系统名称

E list_add_tail(&s->s_list, &super_blocks)：将超级块加入到全局超级块链表中

F list_add(&s->s_instance, &type->fs_supers)：将超级块加入到本文件系统所属的超级块链表中

ii s->s_op = &bdev_sops，设置超级块 s 中其它成员

iii root = new_inode(sb = s)：分配一个 inode 结构并初始化

A inode = sb->s_op->alloc_inode(sb)：调用函数 bdev_sops.alloc_inode = bdev_alloc_inode ()：调用函数 kmem_cache_alloc()从专用缓存 bdev_cachep 中分配一个 bdev_inode 结构 ei，并返回其成员 vfs_inode 作为新分配的 inode，即返回&ei->vfs_inode

B 设置 inode 各成员

C inode->i_data.a_ops = &empty_aops

D inode->i_mapping = &inode->i_data

iv dentry = d_alloc(NULL, &d_name)：调用函数 kmem_cache_alloc()从专用缓存 dentry_cachep 中分配一个 dentry 结构，设置名称为 d_name 中值，初始化其它成员

v dentry->d_sb = s：设置 dentry 结构的超级块

vi dentry->d_parent = dentry：没有父目录，这已经是顶级目录

vii d_instantiate(dentry, root)：设置 dentry 和 root 结构指向关系：将 dentry->d_alias 加入到链表 root->i_dentry 中，设置 dentry->d_inode = root

viii s->s_root = dentry：设置超级块中根 dentry 结构

IV mnt->mnt_sb = sb; mnt->mnt_root = sb->s_root; mnt->mnt_mountpoint = sb->s_root

V mnt->mnt_parent = mnt：表明这是一个跟节点

VI return mnt：返回安装点结构

4 blockdev_superblock = bd_mnt->mnt_sb，保存本文件系统的超级块

注：最终由全局变量 bd_mnt 保存安装点信息，全局变量 blockdev_superblock 保存超级块信息

2.1.2.2 sockfs 文件系统

该文件系统用于管理网络连接，加载的过程为：start_kernel()→rest_init()→init()→do_basic_setup()→sock_init()，调用函数 register_filesystem(&sock_fs_type)注册 sockfs 文件系统，再执行 sock_mnt = kern_mount(&sock_fs_type)内部安装文件系统，注册过程前一节已说明，下面主要分析 kern_mount(&sock_fs_type)执行过程：

1 sock_mnt = kern_mount(type = &sock_fs_type)：内部安装文件系统，直接调用函数 do_kern_mount(type->name, 0, type->name, NULL)，即文件系统类型为“sockfs”，安装的目标设备名称也是“sockfs”

2 sb = type->get_sb()：调用该文件系统的超级块读取函数，对于 sockfs 文件系统，sock_fs_type 中定义的 get_sb 为函数 sockfs_get_sb()，而 sockfs_get_sb()直接调用函数 get_sb_pseudo(fs_type, "socket:", &sockfs_sops, …)：

3 与 bd_type 文件系统的安装过程不同的是超级块操作接口使用 sockfs_ops，其它过程完全相同

4 安装过程中，只有 alloc_inode 使用 sock_alloc_inode，其它步骤与 super_operations 无关。sock_alloc_inode 过程：调用函数 kmem_cache_alloc()从专用缓存 sock_inode_cachep 中分配一个 socket_alloc 结构，并进行初始化，最后将 socket_alloc.vfs_inode 作为 VFS 的 inode 结构返回，并加入到 inode 管理结构中，以后通过该 inode 可以找到所在的 socket_alloc 结构及该结构的另外一个成员 socket。

5 最终由全局变量 sock_mnt 保存安装点信息 vfsmount 结构。

2.1.2.3 procfs 文件系统

加载的过程为：start_kernel()→proc_root_init()，首先调用函数 register_filesystem(&proc_fs_type)注册 procfs 文件系统，再执行 proc_mnt = kern_mount(&proc_fs_type)内部安装文件系统，注册过程前一节已说明，下面主要分析 kern_mount(&proc_fs_type)执行过程与安装 bd_type 文件系统时的区别：

1 sb = type->get_sb()：调用该文件系统的超级块读取函数，对于 proc 文件系统，proc_fs_type 中定义的 get_sb 为函数 proc_get_sb()，而 proc_get_sb()直接调用函数 get_sb_single(fs_type, flags, data, proc_fill_super)，下面给出该函数执行：

I s = sget(fs_type, compare_single, set_anon_super, ...)：其过程与安装 bdev 文件系统时基本相同

II proc_fill_super(s, data, ...)：填写超级块 s 数据结构并设置根 inode

i s->s_op = &proc_sops：设置超级块操作接口

ii root_inode = proc_get_inode(s, ino = PROC_ROOT_INO, de = &proc_root)

A de_get(de)：增加引用计数

B inode = iget(sb, ino)：分配一个标准 inode 并初始化

a inode = iget_locked(sb, ino)：分配 inode

(1) head = inode_hashtable + hash(sb, ino)：准备在 inode 哈希表中搜索

(2) inode = ifind_fast(sb, head, ino)：proc 刚刚初始化，必须搜索失败

(3) get_new_inode_fast(sb, head, ino)：分配一个 inode，并进行初始化

(I) inode = alloc_inode(sb)：调用函数 sb->s_op->alloc_inode= proc_alloc_inode()，分配 inode，proc_alloc_inode()分配一个 proc_inode 结构 ei，初始化 ei 中的其它成员，并将其中的 inode 成员 vfs_inode 作为标准 inode 返回。并完成基本初始化，过程类似 bdev 文件系统

(II) 设置 inode 相关指针成员

b sb->s_op->read_inode(inode)：调用函数 proc_read_inode()函数填写 inode 内容，实际上该函数仅设置了 inode 中的时间信息为当前时间。

c unlock_new_inode(inode)：inode 解锁，可正常使用

C PROC_I(inode)->pde = de：设置该 inode 所在 proc_inode 结构中成员 pde = &proc_root

D 设置 inode 参数：

a inode->i_size = de->size

b inode->i_nlink = de->nlink(= 2)

c inode->i_op = de->proc_iops = proc_root_inode_operations

d inode->i_fop = de->proc_fops = proc_root_operations

iii s->s_root = d_alloc_root(root_inode)

A res = d_alloc(NULL, &name)：分配一个 dentry 结构，并设置路径名和相关链表初始化

B res->d_sb = root_inode->i_sb

C res->d_parent = res：设置一个根目录

D d_instantiate(res, root_inode)：设置 dentry 与 inode 之间的联系

III do_remount_sb(s, flags, data, 0)：待续

2.2 open

核心入口函数为 sys_open(const char __user * filename, int flags, int mode)，从这里开始分析

1 tmp = getname(filename)

I tmp = __getname()：宏，定义为 kmem_cache_alloc(names_cachep,...)，从专用缓存中分配内存

II do_getname(filename, page = tmp)：

i 如果 get_fs()=current_thread_info()->addr_limit 不等于 KERNEL_DS，表示用户空间进程进行访问，需要进行地址范围检查，如下：

A 如果 filename > TASK_SIZE(=TASK_SIZE64= 8000_0000_0000h – 1000h)则表示文件名位于核心空间，直接放回 EFAULT 段错误

B 如果 filename 到 TASK_SIZE 的距离小于 PATH_MAX（4096），则文件名最大长度为 len = TASK_SIZE-filename，确保不会复制核心空间数据

ii strncpy_from_user(page, filename, len)

A access_ok(VERIFY_READ, src, 1)：宏，测试目标数据是否全部位于进程用户地址空间段中，定义为(__range_not_ok(addr,size) == 0)，而__range_not_ok(addr,size)执行汇编指令，若 addr+size ≤ current_thread_info()-> addr_limit.seg 返回 0，否则返回 1。

B __do_strncpy_from_user(page, filename, len, res)：执行汇编指令，从用户空间地址 filename 复制数据到核心空间地址 page，长度 len 字节，出错代码设置到 res 中

2 fd = get_unused_fd()：

I files = current->files：当前进程的 files_struct 结构

II fd = find_next_zero_bit(files->open_fds->fds_bits, ...)：在打开的文件位图中找到一个空闲标记

III expand_files(files, fd)：

i 如果 fd 大于位图数(files->max_fdset)，则调用函数 expand_fdset(files, fd)扩展记录位图 files->open_fds、files->close_on_exec

ii 如果 fd 大于文件号记录数(files->max_fds)，则调用函数 expand_fd_array(files,fd)扩展文件号记录 files->fd

IV FD_SET(fd, files->open_fds)：open_fds 中位图 fd 置 1，表示对应文件号已不再空闲

V FD_CLR(fd, files->close_on_exec)：close_on_exec 所指向的位图 fd 清 0，表示如果当前进程通过 exec()系统调用执行一个可执行文件无需将这个文件关闭，这个位图的内容可以通过 ioctl()系统调用设置

3 f = filp_open(tmp, flags, mode)

I open_namei(filename,..., &nd)

i ACC_MODE(x)：宏，定义为：("\000\004\002\006"[(x)&O_ACCMODE])，常量 O_ACCMODE 定义为 3，当 x=0、1、2、3 时，宏 ACC_MODE(x)的取值分别为 0、4、2、

6，即字符串中定义的数字

ii 如果 flag 中没有 O_CREATE 标记，则调用函数 path_lookup(pathname, lookup_flags(flag) | LOOKUP_OPEN, nd)，然后转 xi 步执行 may_open 函数

iii 否则调用函数 path_lookup(pathname, LOOKUP_PARENT|LOOKUP_OPEN|LOOKUP_CREATE, nd)，然后继续执行

iv 如果 nd->last_type ≠ LAST_NORM 或 nd->last.name[nd->last.len]≠0 则出错

v path.dentry = __lookup_hash(name=&nd->last, base = nd->dentry, nd)：在父目录中查找目标文件，此时 nd 指向目标文件/目录的父目录，而结果 path.dentry 则指向目标文件/目录的 dentry 结构

A inode = base->d_inode

B permission(inode, MAY_EXEC, nd)：检查父目录的执行权限，即测试 inode->i_mode 中的权限

C dentry = cached_lookup(base, name, nd)：在目录 base 中查找目标文件/目录 name

a dentry = __d_lookup(base, name)：

(1) head = d_hash(base, name->hash)，在全局 dentry 哈希表 dentry_hashtable 中找到链表头部

(2) 在 head 中搜索全部节点，查找哈希关键字一致（dentry->d_name.hash==name->hash）、父目录一致（dentry->d_parent == base）、名字长度一致（dentry->d_name.len = name->len）、名字一致（dentry->d_name.name = name->name）的 dentry 结构，查找成功则增加 dentry 结构引用计数并返回该结构，否则返回 NULL

b d_lookup(base, name)：再次查找目标 dentry 结构，

(1) read_seqbegin(&rename_lock)：返回 rename_lock.sequence 值

(2) dentry = __d_lookup(base, name)：再次查找目标 dentry 结构

(3) read_seqretry(&rename_lock, seq)：返回值不等于 0 时重复执行第(1)步

D new = d_alloc(base, name)，如果 cached_lookup 查找失败，则分配一个 dentry 结构并进行初始化

E dentry = inode->i_op->lookup(dir = inode, new, nd)，在当前 inode 中查找目标 dentry 结构 new ，对于 ext2 文件系统执行函数 ext2_lookup()

a 如果 dentry->d_name.len > EXT2_NAME_LEN(255)则返回-ENAMETOOLONG，名字太长，ext2 文件系统最大支持 255

b ino = ext2_inode_by_name(dir, new)：在目标 inode 中查找 dentry 结构 new

(1) de = ext2_find_entry(dir, new, &page)

(I) ei = EXT2_I(dir)：获取 inode 结构 dir 所在的 ext2_inode_info 结构

(II) ext2_get_page(dir, n)

(i) page = read_cache_page(mapping, n, mapping -> a_ops -> readpage, NULL)：对于 ext2 文件系统，readpage 函数为 ext2_readpage

(A) page = __read_cache_page(mapping, index, filler=ext2_readpage, NULL)

(a) page = find_get_page(mapping, index)：调用函数 radix_tree_lookup(mapping-> page_tree, index)，在基数树中查找偏移量为 index 的页面，若找到则在调用函数 page_cache_get(page)=get_page(page)增加 page 引用计数，并返回 page 结构

(b) cache_page=page_cache_alloc_cold(mapping)：若前一步执行失败，则执行本步骤调用函数 alloc_pages()分配分配一个页面

(c) add_to_page_cache_lru(cache_page, mapping, index,..)：

- add_to_page_cache(cache_page, ...)：调用函数将 radix_tree_insert()将页面 cache_page 加入基数树 mapping->page_tree 中，设置 cache_page->mapping = mapping，cache_page->index = index
- lru_cache_add(cache_page)：将页面加入对应链表
  - pvec = &get_cpu_var(lru_add_pvecs)：关闭内核抢占，获取当前 CPU 的 lru_add_pvecs 结构地址
  - pagevec_add(pvec, page)：以 pvec->pages[pvec->nr++]指针指向 cache_page，即 cache_page 页面暂时由 pvec 管理，返回 pvec->pages[]数组中剩余空间
  - 若剩余空间为 0 则调用函数__pagevec_lru_add()，
    - 将 pvec 中的每个页面加入各自 zone 的不活动链表中（add_page_to_inactive_list(zone,)实现）
    - release_pages()：再次检查 pvec->pages 结构中的链表，对于引用计数为 0（page_count(page)）的页面，形成一个新的 pagevec 结构，并将其释放
    - pagevec_reinit(pvec)：重新初始化 pvec 结构，准备再次使用

(d) page = cache_page

(e) filler/ext2_readpage(NULL, page)：仅调用函数 mpage_readpage (page, ext2_get_block)

- do_mpage_readpage(bio,page,1,&last_block_in_bio,get_block)：未完待续
-

(B) mark_page_accessed(page)：设置页面访问状态

(ii) ext2_check_page()：特征参数检测

(III) de = page_address(page)：页面首地址作为 ext2_dir_entry_2 入口

(IV) ext2_match(namelen, name, de)：在读入的 dentry 结构中搜索目标文件/目录名称，匹配则退出，返回 de 结构，否则测试下一个 dentry 结构

(2) ino = le32_to_cpu(de->inode)：获取目标文件/目录的 inode 编号

c inode = iget(dir->i_sb, ino)：读入第 ino 号 inode，其过程参见安装 ext2 文件系统的第 8.IX 步

d d_splice_alias(inode, dentry)：设置 inode 与 dentry 之间的关系

vi path.mnt = nd->mnt

vii vfs_create(dir->d_inode, path.dentry, mode, nd)：创建目标文件/目录

A may_create(dir, dentry, nd)：测试目标文件/目录是否存在，以及权限

B dir->i_op->create(dir, dentry, mode, nd)，即调用函数 ext2_create()建立文件：

a inode = ext2_new_inode(dir, mode)：分配并设置 inode

(1) sb = dir->i_sb

(2) inode = new_inode(sb)：分配 inode 并初始化

(I) alloc_inode(sb)：调用 sb->s_op->alloc_inode(sb)，即函数 ext2_alloc_inode()，从专用缓存 ext2_inode_cachep 中分配一个 ext2_inode_info 结构 ei，并将其成员 vfs_inode 地址作为分配的 inode 返回。并设置 inode 结构各成员初值

(II) 利用成员 i_list 和 i_sb_list 分别加入 inode_in_use 和 sb_s_inodes 链表中

(III) list_add(&inode->i_list, &inode_in_use)：将 inode 加入全局使用链表

(IV) list_add(&inode->i_sb_list, &sb->s_inodes)：将 inode 加入当前超级块链表

(V) inode->i_ino = ++last_ino：last_ino 为静态变量，记录分配的 inode 数目

(3) 如果要创建目录且超级块中有标记 EXT2_MOUNT_OLDALLOC 则调用函数 find_group_dir，若创建目录且没有上述标记，则调用函数 find_group_orlov，若不是目录则调用函数 find_group_other，下面分别说明：

(4) group = find_group_dir(sb, dir)：为创建目录选择区组

(I) ngroups = EXT2_SB(sb)->s_groups_count：当前设备中包含的区组数

(II) avefreei = ext2_count_free_inodes(sb)/ngroups：平均每个区组空闲 inode 数

(i) desc = ext2_get_group_desc(sb, i, NULL)，获取第 i 个区组描述结构

(A) struct ext2_sb_info* sbi = (struct ext2_sb_info*)sb->s_fs_info

(B) group_desc =block_group >> sbi->s_desc_per_block_bits：根据区组号获取区组描述符所在 buffer cache（buffer_head）中的序号

(C) offset = block_group & (sbi->s_desc_per_block - 1)：描述符在当前 buffer cache 中的序号

(D) return (strcut ext2_group_desc*)sbi->s_group_desc[group_desc]->b_data + offset

(ii) desc_count += desc->bg_free_inodes_count;

(iii) 对于本设备中的全部区组执行上述操作

(III) 检查本设备中的每个区组，查找剩余 indoe 大于平均数中空闲块（desc->bg_free_blocks_count）最多的区组，作为选中区组

(5) group = find_group_orlov(sb, dir)：为创建目录选择区组

(I) 比上一组更有效、更复杂

(6) group = find_group_other(sb, dir)：为创建非目录（文件或链接）而选择区组

(I) 如果目录 dir 所在的区组有空闲 inode 和空闲块则直接选中 dir 所在区组

(II) 若上述条件不满足，则用二次哈希的方式选择一个其它有空闲 inode（desc->bg_free_inodes_count>0）和空闲块的区组

(III) 若仍旧未选中，则选择一个有空闲 inode 节点的区组，而不管空闲块

(7) 至此，区组选择完成，选中第 group 个区组

(8) gdp = ext2_get_group_desc(sb, group, &bh2)：获取区组 group 地址和所在 buffer cache 结构

(9) bitmap_bh = read_inode_bitmap(sb, group)：读取当前区组 inode 位图所在磁盘块

(I) desc = ext2_get_group_desc(sb, group, NULL)：获取区组 group 地址

(II) bh = sb_read(sb, desc->bg_inode_bitmap)：读取当前足取的 inode 位图磁盘块

(10) ino = ext2_find_next_zero_bit((unsigned long *)bitmap_bh->b_data, EXT2_INODES_PER_GROUP(sb), ino)：在位图中查找一个空闲点，分配磁盘 inode

(11) 如果测试 ino 失败，则在下一个区组中分配 inode

(12) ext2_set_bit_atomic(sb_bgl_lock(sbi, group), ino, bitmap_bh->b_data)，在位图 bitmap_bh->b_data 中标记序号为 ino 的 inode 已用

(13) mark_buffer_dirty(bitmap_bh)：标记该 buffer_cache 已修改

(14) sync_dirty_buffer(bitmap_bh)：如果 sb->s_flags 有标记 MS_SYNCHRONOUS，则表示同步修改，则调用函数 submit_bh(WRITE, bh)写回 inode 位图

(15) brelse(bitmap_bh)：释放位图

(16) ino += group * EXT2_INODES_PER_GROUP(sb) + 1：将 inode 序号转为块设备内的编号

(17) percpu_counter_mod(&sbi->s_freeinodes_counter, -1)：在超级块中减少一个空闲 inode

(18) percpu_counter_inc(&sbi->s_dirs_counter)：若创建的是目录，则增加目录计数

(19) gdp->bg_free_inodes_count -=1：区组内空闲 inode 减 1
(20) sb->s_dirt = 1：超级块已修改
(21) mark_buffer_dirty(bh2)：标记区组 buffer_cache 已修改
(22) inode->i_ino = ino：设置 inode 自身所在的 inode 编号
(23) 初始化 inode 其它结构
(24) mark_inode_dirty(inode)：标记 inode 已修改

b 设置 inode 操作函数指针：inode->i_op、inode->i_mapping->a_ops、inode->i_fop

c mark_inode_dirty(inode)：标记 inode 已修改

d ext2_add_nondir(dentry, inode)：

(1) ext2_add_link(dentry, inode)

(I) dir = dentry->d_parent->d_inode：父目录 inode 结构

(II) 在目录中查找目标文件名，若已经存在，则返回-EEXIST 错误，否则填写一个 ext2_dir_entry_2 结构 de，设置 de 相关成员：de->inode = inode->i_ino

(III) mark_inode_dirty(dir)：标记父目录已经修改

(2) d_instantiate(dentry, inode)：如果成功，则将 inode 与 dentry 结构建立指向关系，否则释放 inode，返回错误

viii nd->dentry = path.dentry：nd->dentry 指向新建目录/文件

ix

x __follow_mount(&path)：将 path 指向当前活动的安装设备

xi may_open(nd, acc_mode, flag)：测试属性标记及权限等，若带有标记 O_TRUNC，则序号获取写权限，并调用函数 locks_verify_locked 加锁，并将文件长度设置为 0

II dentry_open(nd.dentry, nd.mnt, ...)

i f = get_empty_filp()：分配一个 struct file 结构，并进行基本初始化

ii 如果有写请求，get_write_access(inode)获取写权限，即增加 inode->i_writecount 计数，用于与 mmap 互斥

iii 设置 f 相关成员：f->f_op = inode->i_fop，f->f_dentry = nd.dentry，f->f_vfsmnt= nd.mnt，f->f_mapping = inode->i_mapping

iv 调用函数 f->f_op->open(inode, f)，即 generic_file_open 函数，功能很少

v 检测 O_DIRECT 标记

vi 返回 f 结构

4 fsnotify_open(f->f_dentry)：只有开启编译选项 CONFIG_INOTIFY 在执行

5 fd_install(fd, f)：设置 files->fd[fd]= f

6 至此，文件打开完成

2.3 read

read 过程通过系统调用 sys_read 完成，在该系统调用中，首先调用函数 file = fget_light(fd, &fput_needed)获取文件描述符 file，随后调用主干函数 vfs_read()，后面再调用函数 file_pos_write()和 fput_light()，重置读文件位置和释放文件描述符，下面只分析主干函数 vfs_read(file, buf, count, &pos)

1 参数正确性检查，以及指针 file->f_op、file->f_op->read、file->f_op->aio_read 非空

2 access_ok(VERIFY_WRITE, buf, count)：验证用户空间区域[buf, buf+count)是否具有可写属性

3 rw_verify_area(READ, file, pos, count)：验证文件目标区域是否具有可读属性

I 验证读入的数据量 count 必须小于文件数据总量 file->f_maxcount

II 读取数据前后的文件位置必须有效，即 pos ≥ 0 && pos + count ≥ 0

III 若文件锁（inode->i_flock）有效且启用强制文件锁，宏 MANDATORY_LOCK(inode)=TRUE，即 inode->i_sb->s_flag 中标记 MS_MANDLOCK 置位，且设置了组 ID 但无执行标记，即满足 inode->i_mode &(S_ISGID | S_IXGRP) == S_ISGID，则调用函数 locks_mandatory_area

(read_write == READ ? FLOCK_VERIFY_READ : FLOCK_VERIFY_WRITE, inode, file, pos, count)测试强制文件所

4 若 file->f_op->read 不为空则调用函数 file->f_op->read(file, buf, count, pos)，否则调用函数 do_sync_read(file, buf, count, pos)，下面分别说明

5 file->f_op->read()即 generic_file_read(file, buf, count, ppos)

I struct iovec local_iov = {.iov_base = buf, .iov_len = count}；用户空间数据接收缓冲区段

II init_sync_kiocb(&kiocb, filp)：初始化数据结构 kiocb

III ret = __generic_file_aio_read(iocb=&kiocb, iov =&local_iov, nr_segs=1, ppos)：通过异步 I/O 方式读取数据

i 输入参数中的 1 表示用户空间仅有一个缓冲区段，即 iov 仅有一个分量

ii 利用一个 for 循环，验证 iov[..]各分量给出的缓冲区是否属于用户空间，读取的数据量及总数据量中是否有负值存在，若某个分量非法，则该分量及后续各分两都将无效，而只为该分量以前的各分量服务

iii 如果 filp->f_flags 设置了标记 O_DIRECT，则不使用文件系统的 Cache 机制，而是直接从用户缓冲区到设备进行数据访问，继续操作，否则执行第 vii 步操作

iv mapping = filp->f_mapping，inode = mapping->host，size = inode->i_size

v retval = generic_file_direct_IO(READ, iocb, iov, offset=pos, nr_segs=1)：

A file = iocb->ki_filp，mapping = file->f_mapping

B 如果是写操作，则执行下述操作：

a write_len = iov_length(iov, nr_segs)：计算当前 IO 操作块总数据量

b 如果该文件执行过地址映（mmap）射操作，即 mapping_mapped(mapping)为真，即 mapping->i_mmap->prio_tree_node≠NULL 或 mapping->i_mmap_nonlinear≠NULL，则调用函数 unmap_mapping_range(mapping, offset, write_len, 0)，释放区域[offset, offset+write_len]的地址映射

C filemap_write_and_wait(mapping)

a 如果没有地址映射，即 mapping->nrpages=0，则直接返回 0；否则继续执行

b filemap_fdatawrite(mapping)，直接调用函数__filemap_fdatawrite(mapping, WB_SYNC_ALL)，进一步直接调用函数__filemap_fdatawrite_range(mapping, start=0, end=0, sync_mode= WB_SYNC_ALL)

(1) 声明局部变量 struct writeback_control wbc={.nr_to_write = mapping->nrpages*2}，根据输入参数初始化

(2) mpping_cap_writeback_dirty(mapping)：检测 mapping->backing_dev_info->capabilities 中标记 BDI_CAP_NO_WRITEBACK，有返回 0，否则返回 1。

(3) do_writepages(mapping, &wbc)，调用指针函数 mapping->a_ops->writepages (mapping, wbc)，若该指针为 NULL，则调用函数 generic_writepages(mapping, wbc)，对于 ext2 文件系统，指针 mapping->a_ops->writepages 指向函数 ext2_writepages，该函数直接调用函数 mpage_writepages(mapping, wbc, ext2_get_block)

(I) 待续

c filemap_fdatawait(mapping)，直接调用函数 wait_on_page_writeback_range(mapping, 0, (i_size - 1) >> PAGE_CACHE_SHIFT)，等待数据写完

(1) 待续

D mapping->a_ops->direct_IO(rw, iocb, iov, offset, nr_segs)，即调用函数 ext2_direct_IO，进而直接调用函数 blockdev_direct_IO(rw, iocb, inode=iocb->ki_filp->f_mapping->host, inode->i_sb->s_bdev, iov, offset, nr_segs, ext2_get_blocks, NULL)，再进一步直接调用函数__blockdev_ direct_IO(rw, iocb, inode, bdev, iov, offset, nr_segs, get_blocks = ext2_

get_blocks, end_io = NULL, DIO_LOCKING)

a 待续

vi 直接访问（带标记 O_DIRECT）完成后，函数返回

vii filp->f_flags 未设置标记 O_DIRECT，使用文件系统的 Cache 机制

A 设置一个 for 循环，完成 nr_segs 个用户缓冲区段的读写，每一遍循环完成一段缓冲区的读写操作，下面给出一段缓冲区的读写过程

B 声明局部变量 read_descriptor_t des，并根据输入参数初始化

C do_generic_file_read(filp, ppos, &desc, file_read_actor)直接调用函数 do_generic_mapping_read(filp->f_mapping, &filp->f_ra, filp, ppos, desc, actor= file_read_actor)

a 待续

IV 如果返回 ret = -EIOCBQUEUED，则调用函数 wait_on_sync_kiocb(&kiocb)，使当前进程进入 TASK_UNINTERRUPTIBLE 状态，并进行进程调度，等待读取过程完成

6 do_sync_read(file, buf, count, pos)

I init_sync_kiocb(&kiocb, filp)：初始化数据结构 kiocb，并设置 kiocb.ki_pos = *ppos

II filp->f_op->aio_read(&kiocb, buf, len, kiocb.ki_pos)：即调用函数 ret = generic_file_aio_read()

i struct iovec local_iov = {.iov_base = buf, .iov_len = count};

ii __generic_file_aio_read(iocb, &local_iov, 1, &iocb->ki_pos)

III 若返回值 ret = -EIOCBRETRY 则调用函数 wait_on_retry_sync_kiocb(&kiocb)，并重复执行上述步骤，直至返回值不等于-EIOCBRETRY。函数__generic_file_aio_read 执行过程参见第 5.III 步

IV 若返回值 ret = -EIOCBQUEUED 则调用函数 wait_on_sync_kiocb(&kiocb)

V *ppos = kiocb.ki_pos

2.4 write

入口函数为系统调用 sys_write，与 sys_read 系统调用类似，调用 VFS 层函数 vfs_write，进一步若函数指针 file->f_op->write≠NULL，调用函数 file->f_op->write(file, buf, count, pos)，对于 ext2 文件系统调用函数 ext2_generic_write，否则调用函数 do_sync_ write(file, buf, count, pos)

generic_file_write(file, buf, count, pos)：

1 inode = file->f_mapping->host

2 定义局部变量 struct iov local_iov = {.iov_base = buf, .iov_len = count};

3 down(&inode->i_sem)

4 ret = __generic_file_write_nolock(file, &local_iov, 1, pos)

I init_sync_kiocb(&kiocb, file)

II ret = __generic_file_aio_write_noblock(&kiocb, iov, iov, nr_segs=1, pos)

i 利用一个 for 循环，验证 iov[..]各分量给出的缓冲区是否属于用户空间，写入的数据量及总数据量中是否有负值存在，若某个分量非法，则该分量及后续各分两都将无效，而只为该分量以前的各分量服务

ii vfs_check_frozen(inode->i_sb, SB_FREEZE_WRITE)，宏，定义为 wait_event((inode->sb)->s_wait_unfrozen, ((inode->sb)->s_frozen < (SB_FREEZE_WRITE)))

iii current->backing_dev_info = mapping->backing_dev_info

iv generic_write_checks(file, &pos, &count, S_ISBLK(inode->i_mode))：写数据之前完成必要的数据检查，如写入的数据量和位置等，某些错误可能触发信号 SIGXFSZ

v remove_suid(file->f_dentry)：准备参数之后调用函数 notify_change(dentry, &newattrs)

vi inode_update_time(inode, 1)：设置 inode->i_mtime 和 inode->i_ctime 为当前系统时间，并调用函数 mark_inode_dirty_sync(inode)标记该 inode 需要写回

vii 如果 file->f_flags 中有标记 O_DIRECT，则调用函数 generic_file_direct_write(iocb, iov,

&nr_segs, pos, ppos, count, ocount)写入数据

A 如果 count ≠ ocount，写并不是全部数据都写入，则调用函数*nr_segs = iov_shorten(iov, *nr_segs, count)调整写入的数据量

B written = generic_file_direct_IO(WRITE, iocb, iov, pos, *nr_segs)，参见 read 过程第 5.III.v 步

C 如果写入的数据末尾位置超过原 inode 中记录的末尾位置，则 i_size_write(inode, end)重置 inode->i_size，并将 inode 标记为已修改

D 如果采用同步方式访问文件（满足条件(written >= 0 && ((file->f_flags & O_SYNC) || IS_SYNC(inode)))），则调用函数 generic_osync_inode(inode, mapping, OSYNC_METADATA)，将 inode 所属的文件的全部更改数据写回磁盘

a 待续

viii 如果 file->f_flags 中有标记 O_DIRECT，则调用函数 generic_file_buffered_write(iocb, iov, nr_segs, pos, ppos, count, written)写入数据

A

III 如果返回值 ret = -EIOCBQUEUED 则调用函数 wati_on_sync_kiocb(&kiocb)

5 如果要求使用同步方式（即满足(ret > 0 && ((file->f_flags & O_SYNC) || IS_SYNC(inode))），则调用函数 sync_page_range(inode, mapping, *ppos-ret, ret)等待数据完成

do_sync_ write(file, buf, count, pos)：

1 init_sync_kiocb(&kiocb, filp)：初始化数据结构 kiocb，并设置 kiocb.ki_pos = *ppos

2 filp->f_op->aio_write(&kiocb, buf, len, kiocb.ki_pos)：即调用函数 ret = generic_file_aio_write()

I struct iovec local_iov = {.iov_base = buf, .iov_len = count};

II __generic_file_write_nolock(iocb, &local_iov, 1, &iocb->ki_pos)

III 如果要求使用同步方式（即满足(ret > 0 && ((file->f_flags & O_SYNC) || IS_SYNC(inode))），则调用函数 sync_page_range(inode, mapping, *ppos-ret, ret)等待数据完成

3 如果返回值 ret = -EIOCBRETRY，则调用函数 wait_on_retry_sync_kiocb(&kiocb)，并转第 2 步，直至返回其它置

4 如果返回值 ret = -EIOCBQUEUED，则调用函数 wait_on_sync_kiocb(&kiocb)，进程进入 TASK_UNINTERRUPTIBLE 状态，并进行进程调度。

5 *ppos = kiocb.ki_pos

2.5 mmap

对应的系统调用为 sys_mmap，若不是匿名映射（flags 中无标记 MAP_ANONYMOUS），即使用文件映射，则调用函数 file = fget(fd)获取文件描述符，并执行映射函数主体 do_mmap_pgoff(file, addr, len, prot, flags, off >> PAGE_SHIFT)，下面分析该函数实现

1 验证参数的有效性，并将映射区间圆整到页面大小的整数倍

2 addr = get_unmapped_area(file, addr, len, pgoff, flags)

I 如果标记 flags 中未指定标记 MAP_FIXED，即不必须使用指定地址 addr，则使用下面的指针函数分配地址初值 addr

II 若(p = file->f_op->get_unmapped_area) ≠ NULL，则执行该指针函数 p，否则执行函数 q = current->mm->get_unmapped_area，对于 ext2 文件系统，指针 p=NULL，因此执行指针函数 q = arch_get_unmapped_area()，位于文件 arch/x86-64/kernel/sys_x86_64.c 中

i find_start_end(flags, *begin, *end)：设置可以进行地址映射的虚地址空间，32 位应用程序 *begin = 4000_0000h、*end = 8000_0000h；64 位程序*begin = TASK_UNMAPPED_BASE = TASK_SIZE/3、*end = TASK_SIZE= 8000_0000_0000h – 1000h

ii 如果 addr≠0，即指定映射后的目标地址，则调用 find_vma(mm, addr)查找，若空闲空间满足需求，则直接返回虚地址 addr，否则继续执行

iii 设置 addr 初值：addr = mm->free_area_cache < begin ? begin : mm->free_area_cache

iv 在一个 for 循环中多次调用函数 find_vma(mm, addr)，寻找空闲虚地址空间，找到返回 addr，否则返回-ENOMEM 错误

注释：指针 current->mm->get_unmapped_area 在函数 arch_pick_mmap_layout(struct mm_struct * mm)中设置参数，文件 include/linux/sched.h 中，同时设置 3 个参数：

mm->mmap_base = TASK_UNMAPPED_BASE

mm->get_unmapped_area=arch_get_unmapped_area

mm->unmap_area = arch_unmap_area

而函数 arch_pick_mmap_layout(mm)的调用链如下：

- arch_pick_mmap_layout()←load_elf_binary()←elf_format.load_binary
- arch_pick_mmap_layout()←exec_mmap()←flush_old_exec()

←load_aout_binary() ←aout_format.load_binary

←load_elf_binary() ← elf_format.load_binary

←load_elf_fdpic_binary() ←elf_fdpic_format.load_binary

←load_som_binary() ←som_format.load_binary

←ia32_aout.c/load_aout_binary() ←ia32_aout.c/aout_format.load_binary

←load_flat_file() ←load_flat_binary() flat_format.load_binary

←load_flat_shared_library()←calc_reloc()←load_llat_file()

III 验证地址是否有效

IV 如果是文件映射且使用 hugetlbfs 文件系统，则调用函数 prepare_hugepage_range(addr, len)验证地址是否 hugepage 对齐，否则调用函数 is_hugepage_only_range(current->mm, addr, len)，x86-64 系统为空函数

3 can_do_mlock()：如果 flags 中标记 MAP_LOCKED 置位，则调用该函数测试进程是否拥有对映射内存加锁的权限，若拥有 CAP_IPC_LOCK 权限或 current->signal->rlim[RLIMIT_MEMLOCK]. rlim_cur ≠ 0，则继续执行并设置 VM_LOCKED 标记，否则返回-EPERM 错误

4 如果设置了 VM_LOCKED 标记，则计算加锁区域总量 mm->locked_vm 是否超过 rlim 中定义的限度，若超过则返回-EAGAIN 错误，否则继续执行

5 分析 flags 中的标记及 prot 中的属性与文件中的属性 file->f_mode 是否相符

6 locks_verify_locked(inode)：确保文件没有使用强制锁

7 vma = find_vma_prepare()：查找 vma，准备建立映射

8 do_munmap(mm, addr, len)：如果找到的 vma 管理区域与待映射区域有重叠，则调用该函数释放重叠区域，若释放失败返回-ENOMEM 错误，否则继续执行

9 may_expand_vm()：验证总的地址映射空间不超过系统限制

10 vma_merge()：如果是私有的匿名映射（即 file≠NULL && 没有 VM_SHARED 标记），则调用该函数扩展找到的映射区，成功则返回

11 从专用缓存 vm_area_cachep 中分配一个 vma 结构，并进行初始化

12 如果是文件映射（file≠NULL）执行如下操作：

I 则不能有增长方向标记 VM_GROWSDOWN 或 VM_GROWSUP，若带有写标记 VM_DENYWRITE，则调用函数 deny_write_access(file)文件当前是否可写

II vma->vm_file = file

III get_file(file)：增加文件使用计数 file->f_count

IV file->f_op->mmap(file, vma)：对于 ext2 文件系统则调用函数 generic_file_mmap()：

V mapping = file->f_mapping

VI 如果 mapping->a_ops->readpage=NULL 则返回-ENOEXEC 错误

VII file_accessed(file)：标记文件被访问，如果 file->f_flags 中没有标记 O_NOATIME，则调用函数 touch_atime(file->f_vfsmnt, file->f_dentry)，直接调用函数 update_atime(dentry->d_inode)更

新访问时间 inode->i_atime；

VIII vma->vm_ops = &generic_file_vm_ops：地址映射的关键操作，generic_file_vm_ops 只有 2 个有效成员：.nopage = filemap_nopage；.populate = filemap_populate

13 shmem_zero_setup(vma)：若不是文件映射且带有 VM_SHARED 标记，则调用该函数建立一个共享的匿名映射

I file = shmem_file_setup("dev/zero", size, vma->vm_flags)：建立内存共享文件

i root = shm_mnt->mnt_root；

ii dentry = d_alloc(root, &this)：在 shmfs 文件系统中分配 dentry 结构

iii file = get_empty_filp()：获取一个文件描述符

iv inode = shmem_get_inode(root->d_sb, ...)：在 shmfs 文件系统中分配 inode 结构

v d_instantiate(dentry, inode)：建立 dentry 与 inode 之间的联系]

vi file->f_vfsmnt = shm_mnt)；file->f_dentry = dentry；file->f_mapping = inode->i_mapping；file->f_op = &shmem_file_operations

II vma->vm_file = file

III vma->vm_ops = &shmem_vm_ops

14 vma_merge()：如果是匿名映射调用该函数试图合并 vma

15 atomic_inc(&inode->i_writecount)：如果文件映射且有写映射则增加写计数

16 mm->total_vm += len >> PAGE_SHIFT：记录映射数据量

17 如果带有 VM_LOCKED 标记，增加加锁的内存量 mm->locked_vm；调用函数 **make_pages_present** (addr, addr+len)：验证参数的正确性后调用函数 get_user_pages(current, current->mm, addr, len, write, force = 0, pages = NULL, vmas = NULL)

I vma = find_extend_vma(mm, addr = start)：

i vma = find_vma(mm, addr)：寻找第一个满足 start < vma->end 的 vma，找不到则返回 NULL

ii 如果 vma->vm_start <= addr，即 addr 位于 vma 区域中间，则直接返回 vma

iii 如果没有标记 VM_GROWSDOWN，则直接返回 NULL，即 addr 不在 vma 区域中且找到的区域不是向下增长的区间（栈）则直接返回 NULL

iv expand_stack(vma, addr)：此时 vma 是以栈方式管理的 vma，扩展空间使其包含 addr 地址

A anon_vma_prepare(vma)：寻找或分配一个匿名 vma，如果 vma->anon_vma≠NULL 则直接返回该值，否则继续执行

a anon_vma = find_mergeable_anon_vma(vma)：在邻近的可合并 vma 中找是否有匿名 vma，防止后面分配后再合并，找到即返回临近 vma 中的 anon_vma

b anon_vma = anon_vma_alloc()：未找到则该调用函数分配一个匿名

c vma->anon_vma = anon_vma

d list_add(&vma->anon_vma_node, &anon_vma->head)：加入匿名 vma 链表中

B 将地址 addr 对齐到页面的整数倍 address

C size = address – vma->vma_start；grow = (address – vma->vm_end) >> PAGE_SHIFT

D acct_stack_growth(vma, size, grow)：验证是否可以增长栈空间

E vma->vm_start = address；vma->vm_pgoff -= grow：接受扩展

v 如果 vma 中包含有 VM_LOCKED 标记，则调用函数 make_pages_present(addr, start)分配物理页面

II 此时，或者 vma 为 NULL，或者其中包含 start 地址

III 如果满足条件(vma=NULL && in_gate_area(tsk, start))，即 start 是 gate_vma 中地址（区间[VSYSCALL_START, VSYSCALL_END]中），对于已存在的页面，填写 pages 参数和 vmas 参数，遇到不存在页面则返回，若不满足本条则继续执行

IV page = follow_page(mm, start, write)：直接调用函数__follow_page(mm, address,0, write, 1)：获取地址 address 所在页面的 page 结构，如果 page=NULL，则调用函数__handle_mm_fault(mm,

vma, start, write)分配一个物理页面

i 待续

V 若 pages≠NULL，填写 pages[i] = page；若 vmas≠NULL，填写 vmas[i] = vma

VI start += PAGE_SIZE，len--

VII 若本 vma 中还有要处理的页面，即满足条件 len ≠ 0 && start < vma->vma_end，则转 IV 继续处理下一页

VIII 如果还有页面需要处理，即满足条件 len ≠ 0，则转 I 继续处理下一个 vma

IX 全部完成之后，虚地址都有一个对应的物理页面

18 如果参数带有标记 MAP_POPULATE，则调用函数 sys_remap_file_pages(addr, len, 0, pgof, flags & MAP_NONBLOCK)：重新映射

I 待续

对于 ext2 文件系统，地址映射完成后，访问时将产生缺页中断

.nopage = filemap_nopage；.populate = filemap_populate

IX

2.6 path_lookup

2.7 文件锁与计数器

2.8 文件系统小结

3 进程

linux 中有 3 个系统调用 fork、vfork 和 clone 用来产生进程，在核心中分别对应 sys_fork、sys_vfork 和 sys_cloen，进一步都调用内部函数 do_fork()完成，区别仅是调用 do_fork()的参数不同。

do_fork 参数：

unsigned long clone_flags： 特征参数

unsigned long stack_start： 子进程堆栈起始地址

struct pt_regs* regs： 寄存器结构指针

unsigned long stack_size： 堆栈容量，该参数未使用

int __user * parent_tidptr：父进程 tid 指针

int __user * child_tidptr： 子进程 tid 指针

sys_fork 参数：

struct pt_regs* regs

调用 do_fork 时格式：

clone_flags = SIGCHLD：子进程结束（terminate）或停止（stop）时向父进程发送该信号

stack_start = regs->rsp：共用父进程堆栈，使用 COW 机制进行复制

regs = regs

stack_size = 0

parent_tidptr = NULL

child_tidptr = NULL

sys_vfork 参数：

struct pt_regs* regs

调用 do_fork 时格式：

clone_flags = CLONE_VFORK | CLONE_VM | SIGCHLD：与父进程共用一个地址空间；

并且使父进程挂起进入等待状态，直至子进程释放地址空间，即结束或执行一个新程序；其他与 sys_fork 相同

sys_clone 参数：

unsigned long clone_flags
unsigned long newsp
void __user* parent_tid
void __user* child_tid
struct pt_regs* regs

调用 do_fork 时格式：

stack_start = newsp ? : regs->rsp
stack_start = 0
其它参数对应使用

下面分析 do_fork()过程

1 pid = alloc_pidmap()：分配一个空闲的 pid 号
2 检测 current->ptrace 标记，如果需要跟踪子进程，则在 clone_flags 加入 CLONE_PTRACE 标记
3 p = copy_process()：创建进程描述字
I 如果 clone_flags 同时带有 CLONE_NEWNS 和 CLONE_FS 标记则出错。CLONE_NEWNS 表示使用新的命名空间（namespace）；CLONE_FS 表示与父进程共享 current->fs 结构，即 fs_struct
II 如果 clone_flags 带有 CLONE_THREAD 标记但没有 CLONE_SIGHAND 标记则出错。CLONE_THREAD：将子进程加入到父进程的线程组中，强制子进程共享父进程的信号描述符。CLONE_SIGHAND：共享信号标示表，包括信号句柄（handler）、阻塞和挂起的信号
III 如果 clone_flags 带有 CLONE_SIGHAND 标记但没有 CLONE_VM 标记则出错。CLONE_VM：父子进程共享虚地址空间
IV p = dup_task_struct(orig = current)：复制进程控制字 task_struct
i prepare_to_copy(orig)：直接调用函数 unlazy_fpu(orig)，如果当前进程正在使用浮点单元（FPU），即 orig->thread_info->status 中有 TS_USEDFPU 标记，则调用函数 save_init_fpu (orig)，将浮点寄存器值保存到 orig->thread.i387.fxsave 中，同时清除 orig->thread_info->status 中的 TS_USEDFPU 标记，并调用宏 stts 将 CR0 中的 TS 标记（bit3）置 1，表示即将有进程切换，若使用 x87 指令或多媒体指令将产生设备不可用异常
ii tsk = alloc_task_struct()：从专用缓存 task_struct_cachep 中分配一个进程控制结构
iii ti = alloc_thread_info()：调用为宏__get_free_pages(,1)分配 2 个连续物理页面作为线程管理字和核心堆栈
iv *ti = *orig->thread_info：复制线程控制结构
v *tsk = *origf：复制进程控制结构
vi tsk->thread_info = ti；ti->task = tsk：建立进程、线程控制字对应关系
vii 设置 tsk->usage = 2：设置控制字计数，一个用于新创建进程自身，一个用于 release_task()执行者，通常为父进程
V 验证新进程拥有者的进程总数是否超过限制，即 p->user->processes >= p->signal->rlim[RLIMIT_NPROC].rlim_cur，若超过，且没有管理员权限也不是 root 用户，则出错
VI 增加计数：p->user->__count，p->user->processes
VII get_group_info(p->group_info)：增加 p->group_info->usage 计数
VIII copy_flags(clone_flags, p)：将 p->flags 中清除 PF_SUPERPRIV 标记，设置 PF_FORKNOEXEC 标记，如果 clone_flags 中没有 CLONE_PTRACE 标记，则置 p->ptrace = 0。PF_SUPERPRIV：超级用户标记，PF_FORKNOEXEC：fork 但不执行标记
IX p->pid = pid：为子进程设置 PID
X 如果 clone_flags 中带有 CLONE_PARENT_SETTID 标记，则将 p->pid 写入用户空间变量

parent_tidptr 中

XI 置 p->proc_dentry=NULL，初始化链表 p->children、p->sibling，初始化自旋锁 p->alloc_lock，p->proc_lock，init_sigpending(&p->pending)：初始化挂起信号管理结构，初始化 p 其他成员

XII copy_semundo(clone_flags, p)：

i 如果 clone_flags 中没有 CLONE_SYSVSEM 标记，则置 p->sysvsem.undo_list = NULL 后返回，否则继续执行

ii get_undo_list(undo_listp = &undo_list)：

A 如果当前进程 undo_list（current->sysvsem.undo_list）= NULL，则为其分配一个 semundo_list 结构

B *undo_lsitp = current->sysvsem.undo_list

iii 增加 undo_list 计数 undo_list->refcnt

iv p->sysvsem.undo_list = undo_list

XIII copy_files(clone_flags, p)

i 如果 clone_flags 中有标志 CLONE_FILES，则增加引用计数 current->files->count 返回

ii 从专用缓存 files_cachep 中分配一个 files_struct 结构并初始化

iii open_files = count_open_files()：计算当前进程使用的最大文件描述符 fd

iv 复制 files_struct 中的 open_fds 和 close_on_exec 标记，未使用的部分清 0

v 利用一个 for 循环，复制 current->files->fd[..]中的全部打开的文件到 p->files->fd[..]中，若文件没有打开，则清除 p->file->open_fds 中对应的位图

XIV copy_fs(clone_flags, p)：如果 clone_flags 标记 CLONE_FS 置位，则增加 current->fs->count 计数后返回，否则设置 p->fs = __copy_fs_struct(current->fs)：从专用缓存 fs_cachep 中分配一个 fs_struct 结构，并设置初值等于当前进程的 fs_struct 结构

XV copy_sighand(clone_flags, p)：复制信号处理函数

i 如果带有 CLONE_SIGHAND 或 CLONE_THREAD 标记，则直接增加 current->sighand->count 计数后返回

ii 从专用缓存 sighand_cachep 中分配一个 sighand_struct 结构 sig，并进行初始化

iii 复制 current->sighand->action 到 sig->action 中

iv 设置 p->sighand = sig，sig->count = 1

XVI copy_signal(clone_flags, p)

i 如果 clone_flags 中标记 CLONE_THREAD 置位则增加引用计数 current->signal->count 和 current->signal->live 后返回

ii 从专用缓存 signal_cachep 中分配一个 signal_struct 结构 sig

iii p->signal = sig，执行其它基本初始化

iv 复制 current->signal->rlim 到 sig->rlim 中

XVII copy_mm(clone_flags, p)：复制虚地址空间，从 init_level4_pgt 中复制核心空间页表

i 如果 clone_flags 中标记 CLONE_VM 置位则增加引用计数 current->mm->mm_users，否则继续执行

ii mm = allocate_mm()：从专用缓存 mm_cachep 中分配一个 mm_struct 结构

iii memcpy(mm, current->mm, sizeof(*mm))：复制 mm_struct 结构

iv mm = mm_init(mm)：分配 pgd，并从 init_level4_pgt 中复制核心空间页表

A 对 mm 结构进行基本初始化

B mm_alloc_pgd(mm)：设置 mm->pgd = pgd_alloc(mm)：

a pgd = __get_free_page()：分配一个物理页面

b boundary = pgd_index(__PAGE_OFFSET)：计算地址__PAGE_OFFSET 在 pgd 中的位置

c 将 pgd 中的 0～boundary 项清 0

d 复制 init_level4_pgt+boundary 到 pgd+boundary，共 PTRS_PER_PGD-boundary 项，即复制核心空间对应 pgd，设置子进程核心空间页表

v init_new_context(p, mm)：调用函数 copy_ldt(new = &mm->context, old = &current->mm->context)

A alloc_ldt(pc = new, mimcount = old->size, reload = 0)

a 如果 new->size >= old->size 将直接返回，此时复制进程时直接返回

b 将 mincount 向上圆整到 512 字节的整数倍

c 新的 LDT 容量为 mincount*LDT_ENTRY_SIZE，如果该值小于 1 页则调用 kmalloc 分配内存，否则调用 vmalloc 分配内存 newldt

d 复制原 LDT 内容（即 pc->ldt)，剩余空间清 0

e 释放原 LDT 内容（即 pc->ldt)

f 设置 pc->ldt = newldt

g 如果 reload≠0，则调用 load_LDT(pc)→load_LDT_noblock(pc, cpu)
→set_ldt_desc(cpu, pc->ldt, count)
→load_LDT_desc()

B memcpy(new->ldt, old->ldt, old->size*LDT_ENTRY_SIZE)：复制父进程 mm->context.ldt 内容到子进程 mm->context.ldt 中，指针内容保持一致

4 锁机制

5 内存管理

5.1 swap 机制

swap 机制回收的内存类型：

- page_launder()：回收 inactive_dirty_list 中的页面
- refill_inactive_scan()：将 active_list 中的页面转为不活跃状态
- swap_out()：从 init_mm.mmlinst 开始，扫描全部 mm_struct 结构，换出 vm_area_struct 管理的部分页面
- shrink_dcache_memory()：回收 dentry 结构
- shrink_icache_memory()：回收 inode 结构
- kmem_cache_reap()：回收空间 slab 结构

1

索引：

ext2_sb_ifno	526
ext2_sops	455
ext2_super_block	527
file	541
file_operations	416
file_system_type	497
files_struct	543
fs_struct	417
inode	423
nameidata	433
proc_dir_entry	657
proc_sops	657
super_block	524
vfsmount	510

6 核心线程

migration_thread()
ksoftirqd()

7 特殊功能函数

schedule()
path_lookup()

8 部分全局变量

CPU 单元	操作系统识别的 CPU 最小单元，若开启 SCHED_SMT，则指每个 CPU 线程，否则指每个 CPU 核
cpumask_t	cpu_sibling_map[NR_CPUS]：CPU 单元掩码，考虑超线程
cpumask_t	cpu_core_map[NR_CPUS]：CPU 核掩码
cpumask_t	cpu_online_map：全部有效 CPU 单元掩码
u8	phys_proc_id[NR_CPUS]：每个 CPU 单元所在的 CPU 封装的 APIC_ID，去掉核编号生于部分
u8	cpu_core_id[NR_CPUS]：每个 CPU 单元所在的 CPU 核编号，不包含封装编号
int	smp_num_siblings：当前 CPU 封装中包含的 CPU 单元数目，指出 SMT 结构时每个 CPU 线程作为一个 CPU 单元
u8	cpu_to_node[NR_CPUS]：如果 phys_proc_id[i]在线（位于位图 node_online_map 中），则 cpu_to_node[i] = phys_proc_id[i]，否则等于 fisrt_node(node_online_map)