start_kernel之物理内存管理

by zenhumay

2012-03-26——2012-04-26

目录

1. 概述

本章中涉及的源代码全部来自 linux 内核 2.6.34

start_kernel 是内核初始化的最后一部分，包含了物理内存管理数据结构、中断、文件系统等各方面的初始化。由于涉及的内容较多，不可能一下子将清楚，在这里通过关注点（物理内存、文件系统、中断、进程调度）的不同从不同的角度来剖析该函数。

由于本篇是第一次剖析该函数，所以会将该函数的所有内容都做一个大致打概述，但关注的重点是物理内存管理，其他方面的部分只会大致的介绍器功能，留待已有分析该模块时在详细讲述。

2. cpu初始化

init/main.c

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

	smp_setup_processor_id();

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

```
	/*
	 * Set up the the initial canary ASAP:
	 */
	boot_init_stack_canary();

	cgroup_init_early();

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

```

/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
	lock_kernel();
	tick_init();
	boot_cpu_init();
```

前面这些代码主要是与 SMP CPU、中断等有关的代码，在此不关心这些，后面的讲解将略过。

```
	page_address_init();   //初始化散列表
	printk(KERN_NOTICE "%s", linux_banner);   //输出操作系统旗标
	setup_arch(&command_line);   //处理体系结构相关的内容
	mm_init_owner(&init_mm, &init_task);
	setup_command_line(command_line);
	setup_nr_cpu_ids();
	setup_per_cpu_areas();
```

```
	smp_prepare_boot_cpu(); /* arch-specific boot-cpu hooks */

	build_all_zonelists();	//初始化内存管理区中备选列表
	page_alloc_init();	//

	printk(KERN_NOTICE "Kernel command line: %s\n", boot_command_line);
	parse_early_param();
	parse_args("Booting kernel", static_command_line, __start___param,
		   __stop___param - __start___param,
		   &unknown_bootoption);
	/*
	 * These use large bootmem allocations and must precede
	 * kmem_cache_init()
	 */
	pidhash_init();
	vfs_caches_init_early();
	sort_main_extable();
	trap_init();
	mm_init();
	/*
	 * Set up the scheduler prior starting any interrupts (such as the
	 * timer interrupt). Full topology setup happens at smp_init()
	 * time - but meanwhile we still have a functioning scheduler.
	 */
	sched_init();
	/*
	 * Disable preemption - early bootup scheduling is extremely
	 * fragile until we cpu_idle() for the first time.
	 */
	preempt_disable();
```

```
	if (!irqs_disabled()) {
		printk(KERN_WARNING "start_kernel(): bug: interrupts were "
				"enabled *very* early, fixing it\n");
		local_irq_disable();
	}
	rcu_init();
	radix_tree_init();
	/* init some links before init_ISA_irqs() */
	early_irq_init();
	init_IRQ();
	prio_tree_init();
	init_timers();
	hrtimers_init();
	softirq_init();
	timekeeping_init();
	time_init();
	profile_init();
	if (!irqs_disabled())
		printk(KERN_CRIT "start_kernel(): bug: interrupts were "
				 "enabled early\n");
	early_boot_irqs_on();
	local_irq_enable();

	/* Interrupts are enabled now so all GFP allocations are safe. */
	gfp_allowed_mask = __GFP_BITS_MASK;

	kmem_cache_init_late();

	/*
```

```
	 * HACK ALERT! This is early. We're enabling the console before
	 * we've done PCI setups etc, and console_init() must be aware of
	 * this. But we do want output early, in case something goes wrong.
	 */
	console_init();
	if (panic_later)
		panic(panic_later, panic_param);

	lockdep_info();

	/*
	 * Need to run this when irqs are enabled, because it wants
	 * to self-test [hard/soft]-irqs on/off lock inversion bugs
	 * too:
	 */
	locking_selftest();

#ifdef CONFIG_BLK_DEV_INITRD
	if (initrd_start && !initrd_below_start_ok &&
		page_to_pfn(virt_to_page((void *)initrd_start)) < min_low_pfn) {
		printk(KERN_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "
			"disabling it.\n",
			page_to_pfn(virt_to_page((void *)initrd_start)),
			min_low_pfn);
		initrd_start = 0;
	}
#endif
```

```
	page_cgroup_init();
	enable_debug_pagealloc();
	kmemtrace_init();
	kmemleak_init();
	debug_objects_mem_init();
	idr_init_cache();
	setup_per_cpu_pageset();
	numa_policy_init();
	if (late_time_init)
		late_time_init();
	sched_clock_init();
	calibrate_delay();
	pidmap_init();
	anon_vma_init();
#ifdef CONFIG_X86
	if (efi_enabled)
		efi_enter_virtual_mode();
#endif
	thread_info_cache_init();
	cred_init();
	fork_init(totalram_pages);
	proc_caches_init();
	buffer_init();
	key_init();
	security_init();
	vfs_caches_init(totalram_pages);
	signals_init();
	/* rootfs populating might need page-writeback */
	page_writeback_init();
#ifdef CONFIG_PROC_FS
	proc_root_init();
```

```
#endif
	cgroup_init();
	cpuset_init();
	taskstats_init_early();
	delayacct_init();

	check_bugs();

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

	ftrace_init();

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

3. page_address_init()初始化地址散列表

mm/highmem.c

```
static struct page_address_map page_address_maps[LAST_PKMAP];

void __init page_address_init(void)
{
	int i;

	INIT_LIST_HEAD(&page_address_pool);
	for (i = 0; i < ARRAY_SIZE(page_address_maps); i++)
```

```
		list_add(&page_address_maps[i].list, &page_address_pool);
	for (i = 0; i < ARRAY_SIZE(page_address_htable); i++) {
		INIT_LIST_HEAD(&page_address_htable[i].lh);
		spin_lock_init(&page_address_htable[i].lock);
	}
	spin_lock_init(&pool_lock);
}
```

初始化一个全局的 page_address_pool 列表。

Page_address_map 结构定义如下：

```
 * Describes one page->virtual association
 */
struct page_address_map {
	struct page *page; //物理页面
	void *virtual;//地址
	struct list_head list;
};
```

4. setup_arch处理结构相关的内容

函数所在文件:arch/x86/kernel/setup.c

4.1 setup_arch函数全貌

该函数主要处理与 cpu 架构相关的内容，包括内存处理等。该函数总共 300 多行，在这里只分析我比较关心的部分，在下图中已经用蓝色标出。

```
void __init setup_arch(char **cmdline_p)
{
	int acpi = 0;
	int k8 = 0;

```

```
#ifdef CONFIG_X86_32
	memcpy(&boot_cpu_data, &new_cpu_data, sizeof(new_cpu_data));
	visws_early_detect();
#else
	printk(KERN_INFO "Command line: %s\n", boot_command_line);
#endif

	/* VMI may relocate the fixmap; do this before touching ioremap area */
	vmi_init();

	early_cpu_init();
	early_ioremap_init();

	ROOT_DEV = old_decode_dev(boot_params.hdr.root_dev);
	screen_info = boot_params.screen_info;
	edid_info = boot_params.edid_info;
#ifdef CONFIG_X86_32
	apm_info.bios = boot_params.apm_bios_info;
	ist_info = boot_params.ist_info;
	if (boot_params.sys_desc_table.length != 0) {
		set_mca_bus(boot_params.sys_desc_table.table[3] & 0x2);
		machine_id = boot_params.sys_desc_table.table[0];
		machine_submodel_id = boot_params.sys_desc_table.table[1];
		BIOS_revision = boot_params.sys_desc_table.table[2];
	}
#endif
	saved_video_mode = boot_params.hdr.vid_mode;
```

```
	bootloader_type = boot_params.hdr.type_of_loader;
	if ((bootloader_type >> 4) == 0xe) {
		bootloader_type &= 0xf;
		bootloader_type |= (boot_params.hdr.ext_loader_type+0x10) << 4;
	}
	bootloader_version    = bootloader_type & 0xf;
	bootloader_version |= boot_params.hdr.ext_loader_ver << 4;
#ifdef CONFIG_BLK_DEV_RAM
	rd_image_start = boot_params.hdr.ram_size & RAMDISK_IMAGE_START_MASK;
	rd_prompt = ((boot_params.hdr.ram_size & RAMDISK_PROMPT_FLAG) != 0);
	rd_doload = ((boot_params.hdr.ram_size & RAMDISK_LOAD_FLAG) != 0);
#endif
#ifdef CONFIG_EFI
	if (!strncmp((char *)&boot_params.efi_info.efi_loader_signature,
#ifdef CONFIG_X86_32
		     "EL32",
#else
		     "EL64",
#endif
	 4)) {
		efi_enabled = 1;
		efi_reserve_early();
	}
#endif
	x86_init.oem.arch_setup();
```

```

	setup_memory_map();
	parse_setup_data();
	/* update the e820_saved too */
	e820_reserve_setup_data();

	copy_edd();

	if (!boot_params.hdr.root_flags)
		root_mountflags &= ~MS_RDONLY;
	init_mm.start_code = (unsigned long) _text;
	init_mm.end_code = (unsigned long) _etext;
	init_mm.end_data = (unsigned long) _edata;
	init_mm.brk = _brk_end;

	code_resource.start = virt_to_phys(_text);
	code_resource.end = virt_to_phys(_etext)-1;
	data_resource.start = virt_to_phys(_etext);
	data_resource.end = virt_to_phys(_edata)-1;
	bss_resource.start = virt_to_phys(&__bss_start);
	bss_resource.end = virt_to_phys(&__bss_stop)-1;

#ifdef CONFIG_CMDLINE_BOOL
#ifdef CONFIG_CMDLINE_OVERRIDE
	strlcpy(boot_command_line, builtin_cmdline, COMMAND_LINE_SIZE);
#else
	if (builtin_cmdline[0]) {
		/* append boot loader cmdline to builtin */
		strlcat(builtin_cmdline, " ", COMMAND_LINE_SIZE);
		strlcat(builtin_cmdline, boot_command_line, COMMAND_LINE_SIZE);
```

```
		strlcpy(boot_command_line, builtin_cmdline, COMMAND_LINE_SIZE);
	}
#endif
#endif

	strlcpy(command_line, boot_command_line, COMMAND_LINE_SIZE);
	*cmdline_p = command_line;

	/*
	 * x86_configure_nx() is called before parse_early_param() to detect
	 * whether hardware doesn't support NX (so that the early EHCI debug
	 * console setup can safely call set_fixmap()). It may then be called
	 * again from within noexec_setup() during parsing early parameters
	 * to honor the respective command line option.
	 */
	x86_configure_nx();

	parse_early_param();

	x86_report_nx();

	/* Must be before kernel pagetables are setup */
	vmi_activate();

	/* after early param, so could get panic from serial */
	reserve_early_setup_data();
```

```

	if (acpi_mps_check()) {
#ifdef CONFIG_X86_LOCAL_APIC
		disable_apic = 1;
#endif
		setup_clear_cpu_cap(X86_FEATURE_APIC);
	}

#ifdef CONFIG_PCI
	if (pci_early_dump_regs)
		early_dump_pci_devices();
#endif

	finish_e820_parsing();

	if (efi_enabled)
		efi_init();

	dmi_scan_machine();

	dmi_check_system(bad_bios_dmi_table);

	/*
	 * VMware detection requires dmi to be available, so this
	 * needs to be done after dmi_scan_machine, for the BP.
	 */
	init_hypervisor_platform();

	x86_init.resources.probe_roms();

	/* after parse_early_param, so could debug it */
```

```
	insert_resource(&iomem_resource, &code_resource);
	insert_resource(&iomem_resource, &data_resource);
	insert_resource(&iomem_resource, &bss_resource);

	trim_bios_range();
#ifdef CONFIG_X86_32
	if (ppro_with_ram_bug()) {
		e820_update_range(0x70000000ULL, 0x40000ULL, E820_RAM,
				  E820_RESERVED);
		sanitize_e820_map(e820.map, ARRAY_SIZE(e820.map), &e820.nr_map);
		printk(KERN_INFO "fixed physical RAM map:\n");
		e820_print_map("bad_ppro");
	}
#else
	early_gart_iommu_check();
#endif

	/*
	 * partially used pages are not usable - thus
	 * we are rounding upwards:
	 */
	max_pfn = e820_end_of_ram_pfn();

	/* preallocate 4k for mptable mpc */
	early_reserve_e820_mpc_new();
	/* update e820 for memory not covered by WB MTRRs */
	mtrr_bp_init();
	if (mtrr_trim_uncached_memory(max_pfn))
		max_pfn = e820_end_of_ram_pfn();

```

```
#ifdef CONFIG_X86_32
        /* max_low_pfn get updated here */
        find_low_pfn_range();
#else
        num_physpages = max_pfn;

        check_x2apic();

        /* How many end-of-memory variables you have, grandma! */
        /* need this before calling reserve_initrd */
        if (max_pfn > (1UL<<(32 - PAGE_SHIFT)))
                max_low_pfn = e820_end_of_low_ram_pfn();
        else
                max_low_pfn = max_pfn;

        high_memory = (void *)__va(max_pfn * PAGE_SIZE - 1) + 1;
        max_pfn_mapped = KERNEL_IMAGE_SIZE >> PAGE_SHIFT;
#endif

#ifdef CONFIG_X86_CHECK_BIOS_CORRUPTION
        setup_bios_corruption_check();
#endif

        printk(KERN_DEBUG "initial memory mapped : 0 - %08lx\n",
                        max_pfn_mapped<<PAGE_SHIFT);

        reserve_brk();

        /*
         * Find and reserve possible boot-time SMP configuration:
         */
```

```
	find_smp_config();

	reserve_ibft_region();

	reserve_trampoline_memory();

#ifdef CONFIG_ACPI_SLEEP
	/*
	 * Reserve low memory region for sleep support.
	 * even before init_memory_mapping
	 */
	acpi_reserve_wakeup_memory();
#endif
	init_gbpages();

	/* max_pfn_mapped is updated here */
	max_low_pfn_mapped = init_memory_mapping(0, max_low_pfn<<PAGE_SHIFT);
	max_pfn_mapped = max_low_pfn_mapped;

#ifdef CONFIG_X86_64
	if (max_pfn > max_low_pfn) {
		max_pfn_mapped = init_memory_mapping(1UL<<32,
						     max_pfn<<PAGE_SHIFT);
		/* can we preseve max_low_pfn ?*/
		max_low_pfn = max_pfn;
	}
#endif

	/*
	 * NOTE: On x86-32, only from this point on, fixmaps are ready for use.
```

```
         */

#ifdef CONFIG_PROVIDE_OHCI1394_DMA_INIT
        if (init_ohci1394_dma_early)
                init_ohci1394_dma_on_all_controllers();
#endif

        reserve_initrd();

        reserve_crashkernel();

        vsmp_init();

        io_delay_init();

        /*
         * Parse the ACPI tables for possible boot-time SMP configuration.
         */
        acpi_boot_table_init();

        early_acpi_boot_init();

#ifdef CONFIG_ACPI_NUMA
        /*
         * Parse SRAT to discover nodes.
         */
        acpi = acpi_numa_init();
#endif

#ifdef CONFIG_K8_NUMA
        if (!acpi)
```

```
                k8 = !k8_numa_init(0, max_pfn);
#endif

        initmem_init(0, max_pfn, acpi, k8);
#ifndef CONFIG_NO_BOOTMEM
        early_res_to_bootmem(0, max_low_pfn<<PAGE_SHIFT);
#endif

        dma32_reserve_bootmem();

#ifdef CONFIG_KVM_CLOCK
        kvmclock_init();
#endif

        x86_init.paging.pagetable_setup_start(swapper_pg_dir);
        paging_init();
        x86_init.paging.pagetable_setup_done(swapper_pg_dir);

        tboot_probe();

#ifdef CONFIG_X86_64
        map_vsyscall();
#endif

        generic_apic_probe();

        early_quirks();

        /*
         * Read APIC and some other early information from ACPI tables.
         */
```

```
	acpi_boot_init();

	sfi_init();

	/*
	 * get boot-time SMP configuration:
	 */
	if (smp_found_config)
		get_smp_config();

	prefill_possible_map();

#ifdef CONFIG_X86_64
	init_cpu_to_node();
#endif

	init_apic_mappings();
	ioapic_init_mappings();

/* need to wait for io_apic is mapped */
	probe_nr_irqs_gsi();

	kvm_guest_init();

	e820_reserve_resources();
	e820_mark_nosave_regions(max_low_pfn);

	x86_init.resources.reserve_resources();

	e820_setup_gap();

```

```
#ifdef CONFIG_VT
#if defined(CONFIG_VGA_CONSOLE)
        if (!efi_enabled || (efi_mem_type(0xa0000) != EFI_CONVENTIONAL_MEMORY))
                conswitchp = &vga_con;
#elif defined(CONFIG_DUMMY_CONSOLE)
        conswitchp = &dummy_con;
#endif
#endif
        x86_init.oem.banner();

        mcheck_init();
}
```

4.1.1 early_ioremap_init

所在文件 arch/x86/mm/ioremap.c

4.1.1.1 内核地址空间简介

在 x86-32 为系统的保护模式下，用户空间和内核空间的分配比例是 3:1
其中：

用户空间地址范围：0x00000000—0xbfffffffff

内核空间地址范围：0xc0000000—0xfffffff

由于内核的线性地址和物理地址之间有线性的关系，对应的关系为：
内核物理地址=内核线性地址-0Xc0000000

所以内核对应的物理地址空间为 0-1G。如果物理地址超过 1G，内核将无法直接访问，为了访问 1G 以上的内存，内核将

0-896M 的物理地址线性的映射到内核线性地址空间，896M 的物理地址成为高端内存，使用动态映射的方法，动态的映射到内核线性地址空间的后 128M。

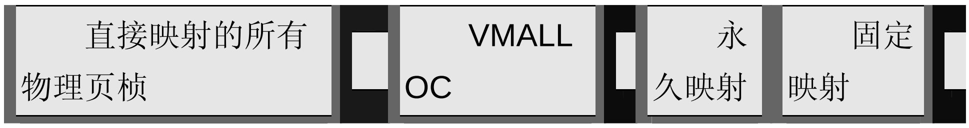
内核地址动态映射分为三种方式：

非连续内存分配：虚拟内存中连续、物理地址中不联系的内存区，使用 vmalloc 区域分配。

永久映射：将高端内存域中的非持久页映射到内核中。

固定映射：与物理地址空间中固定页关联的虚拟地址空间项，具体关联的页桢可以自由选择。可以自由选择。

内核示意图图如下：



|<---0xC0000000------→

4.1.1.2 函数源码

该函数的功能是将高端内存映射到内核后 128M 的线性地址空间内，使得内核可以访问高端内存。

```
369 void __init early_ioremap_init(void)
370 {
371         pmd_t *pmd;
372         int i;
373
374         if (early_ioremap_debug)
375                 printk(KERN_INFO "early_ioremap_init()\n");
376    //FIX_BITMAPS_SLOTS=4 NR_FIX_BITMAPS=64
377         for (i = 0; i < FIX_BTMAPS_SLOTS; i++)
378                 slot_virt[i] = __fix_to_virt(FIX_BTMAP_BEGIN - NR_FIX_BTMAPS*i);
```

从后面__fix_to_virt 可以看出，一个固定的插槽索引，对应 4K 的地址空间，

```
379
380         pmd = early_ioremap_pmd(fix_to_virt(FIX_BTMAP_BEGIN));
```

380 行：获得固定索引地址 FIX_BITMAP_BEGIN 对应的虚拟地址的页中间目录项。

```
381         memset(bm_pte, 0, sizeof(bm_pte));
382         pmd_populate_kernel(&init_mm, pmd, bm_pte);
```

将页表 bm_pte 的地址赋值给 pmd。

```

	/*
	 * The boot-ioremap range spans multiple pmds, for which
	 * we are not prepared:
	 */
#define __FIXADDR_TOP (-PAGE_SIZE)
	BUILD_BUG_ON((__fix_to_virt(FIX_BTMAP_BEGIN) >> PMD_SHIFT)
		     != (__fix_to_virt(FIX_BTMAP_END) >> PMD_SHIFT));
#undef __FIXADDR_TOP
	if (pmd != early_ioremap_pmd(fix_to_virt(FIX_BTMAP_END))) {
		WARN_ON(1);
		printk(KERN_WARNING "pmd %p != %p\n",
		       pmd, early_ioremap_pmd(fix_to_virt(FIX_BTMAP_END)));
		printk(KERN_WARNING "fix_to_virt(FIX_BTMAP_BEGIN): %08lx\n",
			fix_to_virt(FIX_BTMAP_BEGIN));
		printk(KERN_WARNING "fix_to_virt(FIX_BTMAP_END):   %08lx\n",
			fix_to_virt(FIX_BTMAP_END));

		printk(KERN_WARNING "FIX_BTMAP_END:       %d\n", FIX_BTMAP_END);
		printk(KERN_WARNING "FIX_BTMAP_BEGIN:     %d\n",
		       FIX_BTMAP_BEGIN);
	}
}
```

该函数涉及到的宏和变量定义如下：

```
static unsigned long slot_virt[FIX_BTMAPS_SLOTS] __initdata;
```

```
 * Here we define all the compile-time 'special' virtual
 * addresses. The point is to have a constant address at
 * compile time, but to set the physical address only
 * in the boot process.
 * for x86_32: We allocate these special addresses
 * from the end of virtual memory (0xfffff000) backwards.
 * Also this lets us do fail-safe vmalloc(), we
 * can guarantee that these special addresses and
 * vmalloc()-ed addresses never overlap.
 *
 * These 'compile-time allocated' memory buffers are
 * fixed-size 4k pages (or larger if used with an increment
 * higher than 1). Use set_fixmap(idx,phys) to associate
 * physical memory with fixmap indices.
 *
 * TLB entries of such buffers will not be flushed across
 * task switches.
 */
enum fixed_addresses {
#ifdef CONFIG_X86_32
        FIX_HOLE,
        FIX_VDSO,
#else
        VSYSCALL_LAST_PAGE,
        VSYSCALL_FIRST_PAGE = VSYSCALL_LAST_PAGE
                            + ((VSYSCALL_END-VSYSCALL_START) >> PAGE_SHIFT) - 1,
        VSYSCALL_HPET,
```

```
#endif
        FIX_DBGP_BASE,
        FIX_EARLYCON_MEM_BASE,
#ifdef CONFIG_PROVIDE_OHCI1394_DMA_INIT
        FIX_OHCI1394_BASE,
#endif
#ifdef CONFIG_X86_LOCAL_APIC
        FIX_APIC_BASE,    /* local (CPU) APIC) -- required for SMP or not */
#endif
#ifdef CONFIG_X86_IO_APIC
        FIX_IO_APIC_BASE_0,
        FIX_IO_APIC_BASE_END = FIX_IO_APIC_BASE_0 + MAX_IO_APICS - 1,
#endif
#ifdef CONFIG_X86_VISWS_APIC
        FIX_CO_CPU, /* Cobalt timer */
        FIX_CO_APIC,      /* Cobalt APIC Redirection Table */
        FIX_LI_PCIA,      /* Lithium PCI Bridge A */
        FIX_LI_PCIB,      /* Lithium PCI Bridge B */
#endif
#ifdef CONFIG_X86_F00F_BUG
        FIX_F00F_IDT,     /* Virtual mapping for IDT */
#endif
#ifdef CONFIG_X86_CYCLONE_TIMER
        FIX_CYCLONE_TIMER, /*cyclone timer register*/
#endif
#ifdef CONFIG_X86_32
        FIX_KMAP_BEGIN, /* reserved pte's for temporary kernel mappings */
        FIX_KMAP_END = FIX_KMAP_BEGIN+(KM_TYPE_NR*NR_CPUS)-1,
```

```
#ifdef CONFIG_PCI_MMCONFIG
        FIX_PCIE_MCFG,
#endif
#endif
#ifdef CONFIG_PARAVIRT
        FIX_PARAVIRT_BOOTMAP,
#endif
        FIX_TEXT_POKE1, /* reserve 2 pages for text_poke() */
        FIX_TEXT_POKE0, /* first page is last, because allocation is backward */
        __end_of_permanent_fixed_addresses,
        /*
         * 256 temporary boot-time mappings, used by early_ioremap(),
         * before ioremap() is functional.
         *
         * If necessary we round it up to the next 256 pages boundary so
         * that we can have a single pgd entry and a single pte table:
         */
#define NR_FIX_BTMAPS           64
#define FIX_BTMAPS_SLOTS        4
#define TOTAL_FIX_BTMAPS        (NR_FIX_BTMAPS * FIX_BTMAPS_SLOTS)
        FIX_BTMAP_END =
         (__end_of_permanent_fixed_addresses ^
          (__end_of_permanent_fixed_addresses + TOTAL_FIX_BTMAPS - 1)) &
         -PTRS_PER_PTE
         ? __end_of_permanent_fixed_addresses + TOTAL_FIX_BTMAPS -
           (__end_of_permanent_fixed_addresses & (TOTAL_FIX_BTMAPS - 1))
         : __end_of_permanent_fixed_addresses,
```

```
        FIX_BTMAP_BEGIN = FIX_BTMAP_END + TOTAL_FIX_BTMAPS - 1,
#ifdef CONFIG_X86_32
        FIX_WP_TEST,
#endif
#ifdef CONFIG_INTEL_TXT
        FIX_TBOOT_BASE,
#endif
        __end_of_fixed_addresses
};
```

可以看出 fixed_addresses 是一个枚举类型的变量，在其中定义的

FIX_BTMAP_BEGIN

FIX_BTMAP_END

FIX_KMAP_BEGIN

FIX_KMAP_END

__end_of_permanent_fixed_addresses

__end_of_fixed_addresses

都是枚举变量中的成员。

函数__fix_to_virt 的定义如下

```
#define __fix_to_virt(x)        (FIXADDR_TOP - ((x) << PAGE_SHIFT))
#define __virt_to_fix(x)        ((FIXADDR_TOP - ((x)&PAGE_MASK)) >> PAGE_SHIFT)
```

其中 FIXADDR_TOP 定义如下

```
extern unsigned long __FIXADDR_TOP;
#define FIXADDR_TOP ((unsigned long)__FIXADDR_TOP)
```

其中__FIXADDR_TOP 的定义在 arch/x86/mm/pgtable32.c 中如下：

```
unsigned long __FIXADDR_TOP = 0xfffff000;
EXPORT_SYMBOL(__FIXADDR_TOP);
```

__fix_to_virt 的含义为：从地址 0Xfffff000(4G 后退一个页面的地址)回退固定地址(x)个页面后的虚拟地址。

4.1.2 setup_memory_map

进行内存管理，将前面通过 e820 调用获得的内存信息拷贝到 e820 变量中。该函数调用 x86_init.resources.memory_setup，获得内存信息，将 e820 中的信息赋值一份保存在 e820_saved 中，然后打印内存布局图。

```
void __init setup_memory_map(void)
{
        char *who;

        who = x86_init.resources.memory_setup();
        memcpy(&e820_saved, &e820, sizeof(struct e820map));
        printk(KERN_INFO "BIOS-provided physical RAM map:\n");
        e820_print_map(who);
}
```

X86_init 的定义在 arch/x86/kernel/x86_init.c 中：

```
struct x86_init_ops x86_init __initdata = {

        .resources = {
                .probe_roms         = x86_init_noop,
                .reserve_resources  = reserve_standard_io_resources,
                .memory_setup       = default_machine_specific_memory_setup,
        },

        .mpparse = {
                .mpc_record         = x86_init_uint_noop,
                .setup_ioapic_ids   = x86_init_noop,
```

```
		.mpc_apic_id			= default_mpc_apic_id,
		.smp_read_mpc_oem		= default_smp_read_mpc_oem,
		.mpc_oem_bus_info		= default_mpc_oem_bus_info,
		.find_smp_config		= default_find_smp_config,
		.get_smp_config			= default_get_smp_config,
	},

	.irqs = {
		.pre_vector_init		= init_ISA_irqs,
		.intr_init			= native_init_IRQ,
		.trap_init			= x86_init_noop,
	},

	.oem = {
		.arch_setup			= x86_init_noop,
		.banner				= default_banner,
	},

	.paging = {
		.pagetable_setup_start	= native_pagetable_setup_start,
		.pagetable_setup_done	= native_pagetable_setup_done,
	},

	.timers = {
		.setup_percpu_clockev	= setup_boot_APIC_clock,
		.tsc_pre_init			= x86_init_noop,
		.timer_init			= hpet_time_init,
	},

	.iommu = {
		.iommu_init			= iommu_init_noop,
```

```
	},

	.pci = {
		.init		= x86_default_pci_init,
		.init_irq	= x86_default_pci_init_irq,
		.fixup_irqs	= x86_default_pci_fixup_irqs,
	},
};
```

对 x86_init.resources.memory_setup()的调用将会调用函数，default_machine_specific_memory_setup，该函数定义在 arch/x86/kernel/e820.c 中：

该函数的将启动参数中内存分段的信息拷贝到全局变量 e820 中，过程比较简单，就不进一步分析了。

```
char *__init default_machine_specific_memory_setup(void)
{
	char *who = "BIOS-e820";
	u32 new_nr;
	/*
	 * Try to copy the BIOS-supplied E820-map.
	 *
	 * Otherwise fake a memory map; one section from 0k->640k,
	 * the next section from 1mb->appropriate_mem_k
	 */
	new_nr = boot_params.e820_entries;
	sanitize_e820_map(boot_params.e820_map,
			ARRAY_SIZE(boot_params.e820_map),
			&new_nr);
	boot_params.e820_entries = new_nr;
	if (append_e820_map(boot_params.e820_map, boot_params.e820_entries)
	  < 0) {
```

```
		u64 mem_size;

		/* compare results from other methods and take the greater */
		if (boot_params.alt_mem_k
		    < boot_params.screen_info.ext_mem_k) {
			mem_size = boot_params.screen_info.ext_mem_k;
			who = "BIOS-88";
		} else {
			mem_size = boot_params.alt_mem_k;
			who = "BIOS-e801";
		}

		e820.nr_map = 0;
		e820_add_region(0, LOWMEMSIZE(), E820_RAM);
		e820_add_region(HIGH_MEMORY, mem_size << 10, E820_RAM);
	}

	/* In case someone cares... */
	return who;
}
```

4.1.3 初始化 0 号进程变量

782 行到 792 行，初始化 0 号进程 init_mm 中的相关变量。

4.1.4 x86_configure_nx() 设置页面不可执行标志

函数定义在 arch/x86/mm/setup_nx.c 中：

```
void __cpuinit x86_configure_nx(void)
{
```

```
34         if (cpu_has_nx && !disable_nx)
35                 __supported_pte_mask |= _PAGE_NX;
36         else
37                 __supported_pte_mask &= ~_PAGE_NX;
38 }
```

由于之前没有区分代码段和堆、堆栈之间的区别，导致堆溢出和缓冲去攻击横行，后来 cpu 的硬件厂商支持 nx 技术，对非代码区页面设置 nx 标志，将不容许 cpu 在该页面上执行代码。

上述代码 34 行检测 cpu 是否支持 nx 技术，以及是否禁用 nx 技术，如果 cpu 支持且没有禁用 nx，则在__supported_pte_mask 中职位_PAGE_NX 位。

4.1.5 e820_end_of_ram_pfn获得物理页面中页面数（可能包含空洞）

该函数所在文件 arch/x86/kernel/e820.c 中：

```
889 unsigned long __init e820_end_of_ram_pfn(void)
890 {
891         return e820_end_pfn(MAX_ARCH_PFN, E820_RAM);
892 }
```

891 行调用 e820_end_pfn.该函数检测 e820 变量中内存区段，将区段中所在内存的最大的内存页面号返回。过程比较简单，不进一步分析。

```
static unsigned long __init e820_end_pfn(unsigned long limit_pfn, unsigned type)
{
        int i;
        unsigned long last_pfn = 0;
        unsigned long max_arch_pfn = MAX_ARCH_PFN;

        for (i = 0; i < e820.nr_map; i++) {
                struct e820entry *ei = &e820.map[i];
```

最大页桢号。

```
		unsigned long start_pfn;
		unsigned long end_pfn;

		if (ei->type != type)
			continue;

		start_pfn = ei->addr >> PAGE_SHIFT;
		end_pfn = (ei->addr + ei->size) >> PAGE_SHIFT;

		if (start_pfn >= limit_pfn)
			continue;
		if (end_pfn > limit_pfn) {
			last_pfn = limit_pfn;
			break;
		}
		if (end_pfn > last_pfn)
			last_pfn = end_pfn;
	}

	if (last_pfn > max_arch_pfn)
		last_pfn = max_arch_pfn;

	printk(KERN_INFO "last_pfn = %#lx max_arch_pfn = %#lx\n",
			 last_pfn, max_arch_pfn);
	return last_pfn;
}
```

4.1.6 find_low_pfn_range

物理内存被分为 DMA,NORMAL,HIGH 三个区。该函数获得 normal 内存区域的最大页桢号。

4.1.7 init_memory_mapping建立永久内核页表

在前面的分析过程中，已经看到初始化过程中为加载进内存的内核建立了页表，现在要为所有的物理内存建立页表。

函数所在文件：arch/x86/mm/init.c

```
 * Setup the direct mapping of the physical memory at PAGE_OFFSET.
 * This runs before bootmem is initialized and gets pages directly from
 * the physical memory. To access them they are temporarily mapped.
 */
unsigned long __init_refok init_memory_mapping(unsigned long start,
                                               unsigned long end)
{
        unsigned long page_size_mask = 0;
        unsigned long start_pfn, end_pfn;
        unsigned long ret = 0;
        unsigned long pos;

        struct map_range mr[NR_RANGE_MR];
        int nr_range, i;
        int use_pse, use_gbpages;

        printk(KERN_INFO "init_memory_mapping: %016lx-%016lx\n", start, end);

#if defined(CONFIG_DEBUG_PAGEALLOC) || defined(CONFIG_KMEMCHECK)
        /*
         * For CONFIG_DEBUG_PAGEALLOC, identity mapping will use small pages.
```

```
	 * This will simplify cpa(), which otherwise needs to support splitting
	 * large pages into small in interrupt context, etc.
	 */
	use_pse = use_gbpages = 0;
#else
	use_pse = cpu_has_pse;
	use_gbpages = direct_gbpages;
#endif

	/* Enable PSE if available */
	if (cpu_has_pse)
		set_in_cr4(X86_CR4_PSE);

	/* Enable PGE if available */
	if (cpu_has_pge) {
		set_in_cr4(X86_CR4_PGE);
		__supported_pte_mask |= _PAGE_GLOBAL;
	}

	if (use_gbpages)
		page_size_mask |= 1 << PG_LEVEL_1G;
	if (use_pse)
		page_size_mask |= 1 << PG_LEVEL_2M;

	memset(mr, 0, sizeof(mr));
	nr_range = 0;

	/* head if not big page alignment ? */
	start_pfn = start >> PAGE_SHIFT;
	pos = start_pfn << PAGE_SHIFT;
#ifdef CONFIG_X86_32
```

```
	/*
	 * Don't use a large page for the first 2/4MB of memory
	 * because there are often fixed size MTRRs in there
	 * and overlapping MTRRs into large pages can cause
	 * slowdowns.
	 */
	if (pos == 0)
		end_pfn = 1<<(PMD_SHIFT - PAGE_SHIFT);
	else
		end_pfn = ((pos + (PMD_SIZE - 1))>>PMD_SHIFT)
				 << (PMD_SHIFT - PAGE_SHIFT);
#else /* CONFIG_X86_64 */
	end_pfn = ((pos + (PMD_SIZE - 1)) >> PMD_SHIFT)
			<< (PMD_SHIFT - PAGE_SHIFT);
#endif
	if (end_pfn > (end >> PAGE_SHIFT))
		end_pfn = end >> PAGE_SHIFT;
	if (start_pfn < end_pfn) {
		nr_range = save_mr(mr, nr_range, start_pfn, end_pfn, 0);
		pos = end_pfn << PAGE_SHIFT;
	}

	/* big page (2M) range */
	start_pfn = ((pos + (PMD_SIZE - 1))>>PMD_SHIFT)
			 << (PMD_SHIFT - PAGE_SHIFT);
#ifdef CONFIG_X86_32
	end_pfn = (end>>PMD_SHIFT) << (PMD_SHIFT - PAGE_SHIFT);
#else /* CONFIG_X86_64 */
	end_pfn = ((pos + (PUD_SIZE - 1))>>PUD_SHIFT)
			 << (PUD_SHIFT - PAGE_SHIFT);
```

```
        if (end_pfn > ((end>>PMD_SHIFT)<<(PMD_SHIFT - PAGE_SHIFT)))
                end_pfn = ((end>>PMD_SHIFT)<<(PMD_SHIFT - PAGE_SHIFT));
#endif

        if (start_pfn < end_pfn) {
                nr_range = save_mr(mr, nr_range, start_pfn, end_pfn,
                                page_size_mask & (1<<PG_LEVEL_2M));
                pos = end_pfn << PAGE_SHIFT;
        }

#ifdef CONFIG_X86_64
        /* big page (1G) range */
        start_pfn = ((pos + (PUD_SIZE - 1))>>PUD_SHIFT)
                         << (PUD_SHIFT - PAGE_SHIFT);
        end_pfn = (end >> PUD_SHIFT) << (PUD_SHIFT - PAGE_SHIFT);
        if (start_pfn < end_pfn) {
                nr_range = save_mr(mr, nr_range, start_pfn, end_pfn,
                                page_size_mask &
                                 ((1<<PG_LEVEL_2M)|(1<<PG_LEVEL_1G)));
                pos = end_pfn << PAGE_SHIFT;
        }

        /* tail is not big page (1G) alignment */
        start_pfn = ((pos + (PMD_SIZE - 1))>>PMD_SHIFT)
                         << (PMD_SHIFT - PAGE_SHIFT);
        end_pfn = (end >> PMD_SHIFT) << (PMD_SHIFT - PAGE_SHIFT);
        if (start_pfn < end_pfn) {
                nr_range = save_mr(mr, nr_range, start_pfn, end_pfn,
```

```
				page_size_mask & (1<<PG_LEVEL_2M));
		pos = end_pfn << PAGE_SHIFT;
	}
#endif

	/* tail is not big page (2M) alignment */
	start_pfn = pos>>PAGE_SHIFT;
	end_pfn = end>>PAGE_SHIFT;
	nr_range = save_mr(mr, nr_range, start_pfn, end_pfn, 0);

	/* try to merge same page size and continuous */
	for (i = 0; nr_range > 1 && i < nr_range - 1; i++) {
		unsigned long old_start;
		if (mr[i].end != mr[i+1].start ||
			mr[i].page_size_mask != mr[i+1].page_size_mask)
			continue;
		/* move it */
		old_start = mr[i].start;
		memmove(&mr[i], &mr[i+1],
			(nr_range - 1 - i) * sizeof(struct map_range));
		mr[i--].start = old_start;
		nr_range--;
	}

	for (i = 0; i < nr_range; i++)
		printk(KERN_DEBUG " %010lx - %010lx page %s\n",
				mr[i].start, mr[i].end,
			(mr[i].page_size_mask & (1<<PG_LEVEL_1G))?"1G":(
			 (mr[i].page_size_mask & (1<<PG_LEVEL_2M))?"2M":"4k"));

	/*
```

```
261          * Find space for the kernel direct mapping tables.
262          *
263          * Later we should allocate these tables in the local node of the
264          * memory mapped. Unfortunately this is done currently before the
265          * nodes are discovered.
266          */
```

169-266 行是根据物理页面的不同大小（4K,2M），将物理内存分为不同的区，并将分区结果保存在变量 mr 中。

```
267         if (!after_bootmem)
268                 find_early_table_space(end, use_pse, use_gbpages);
269
270         for (i = 0; i < nr_range; i++)
271                 ret = kernel_physical_mapping_init(mr[i].start, mr[i].end,
272                                                    mr[i].page_size_mask);
273
274 #ifdef CONFIG_X86_32
275         early_ioremap_page_table_range_init();
276
277         load_cr3(swapper_pg_dir);
278 #endif
279
280 #ifdef CONFIG_X86_64
280 #ifdef CONFIG_X86_64
281         if (!after_bootmem && !start) {
282                 pud_t *pud;
283                 pmd_t *pmd;
284
285                 mmu_cr4_features = read_cr4();
286
287                 /*
```

```
		 * _brk_end cannot change anymore, but it and _end may be
		 * located on different 2M pages. cleanup_highmap(), however,
		 * can only consider _end when it runs, so destroy any
		 * mappings beyond _brk_end here.
		 */
		pud = pud_offset(pgd_offset_k(_brk_end), _brk_end);
		pmd = pmd_offset(pud, _brk_end - 1);
		while (++pmd <= pmd_offset(pud, (unsigned long)_end - 1))
			pmd_clear(pmd);
	}
#endif
	__flush_tlb_all();

	if (!after_bootmem && e820_table_end > e820_table_start)
		reserve_early(e820_table_start << PAGE_SHIFT,
				 e820_table_end << PAGE_SHIFT, "PGTABLE");

	if (!after_bootmem)
		early_memtest(start, end);

	return ret >> PAGE_SHIFT;
}
```

该函数的调用过程是：

find_early_table_space：查找可疑容纳所有物理内存页表项的连续内存区域。
kernel_physical_mapping_init：为不同的分区建立物理内存映射。
early_ioremap_page_table_range_init()
load_cr3

4.1.7.1 find_early_table_space

该函数实现的功能是在 e820 内存区中，查找一块可以容量所有物理内存页表项的连续物理内存块。实现过程比较简单，重点查看 find_e820_area 函数。

```
static void __init find_early_table_space(unsigned long end, int use_pse,
                                          int use_gbpages)
{
        unsigned long puds, pmds, ptes, tables, start;

        puds = (end + PUD_SIZE - 1) >> PUD_SHIFT;
        tables = roundup(puds * sizeof(pud_t), PAGE_SIZE);

        if (use_gbpages) {
                unsigned long extra;

                extra = end - ((end>>PUD_SHIFT) << PUD_SHIFT);
                pmds = (extra + PMD_SIZE - 1) >> PMD_SHIFT;
        } else
                pmds = (end + PMD_SIZE - 1) >> PMD_SHIFT;

        tables += roundup(pmds * sizeof(pmd_t), PAGE_SIZE);

        if (use_pse) {
                unsigned long extra;

                extra = end - ((end>>PMD_SHIFT) << PMD_SHIFT);
#ifdef CONFIG_X86_32
                extra += PMD_SIZE;
#endif
                ptes = (extra + PAGE_SIZE - 1) >> PAGE_SHIFT;
        } else
```

```
59              ptes = (end + PAGE_SIZE - 1) >> PAGE_SHIFT;
60
61          tables += roundup(ptes * sizeof(pte_t), PAGE_SIZE);
62
63 #ifdef CONFIG_X86_32
64          /* for fixmap */
65          tables += roundup(__end_of_fixed_addresses * sizeof(pte_t), PAGE_SIZE);
66 #endif
67
68          /*
69           * RED-PEN putting page tables only on node 0 could
70           * cause a hotspot and fill up ZONE_DMA. The page tables
71           * need roughly 0.5KB per GB.
72           */
73 #ifdef CONFIG_X86_32
74          start = 0x7000;
75 #else
76          start = 0x8000;
77 #endif
78          e820_table_start = find_e820_area(start, max_pfn_mapped<<PAGE_SHIFT,
79                                          tables, PAGE_SIZE);
80          if (e820_table_start == -1UL)
81                  panic("Cannot find space for the kernel page tables");
82
83          e820_table_start >>= PAGE_SHIFT;
84          e820_table_end = e820_table_start;
85          e820_table_top = e820_table_start + (tables >> PAGE_SHIFT);
```

83-85 行：将获得的内存地址的起始页面号保存到 e820_table_end 中，终止页面号保存到 e820_table_top 中，以后分配页表时，e820_table_end，e820_table_top 将起到重要的作用。

```
	printk(KERN_DEBUG "kernel direct mapping tables up to %lx @ %lx-%lx\n",
		end, e820_table_start << PAGE_SHIFT, e820_table_top << PAGE_SHIFT);
}
```

find_e820_area 所在文件 arch/x86/kernel/e820.c

```
u64 __init find_e820_area(u64 start, u64 end, u64 size, u64 align)
{
	int i;

	for (i = 0; i < e820.nr_map; i++) {
		struct e820entry *ei = &e820.map[i];
		u64 addr;
		u64 ei_start, ei_last;
//必须是 RAM 区域
		if (ei->type != E820_RAM)
			continue;

		ei_last = ei->addr + ei->size;
		ei_start = ei->addr;
		addr = find_early_area(ei_start, ei_last, start, end,
					 size, align);

		if (addr != -1ULL)
			return addr;
	}
	return -1ULL;
}
```

find_e820_area 转而调用在 kernel/early_res.c 中的函数 find_eary_area：

该函数的六个参数如下：

ei_start:e820 内存区起始区域地址

ei_last:e820 内存区域终止地址

start:可供分配内存的起始地址

end:可供分配内存的终止地址

size:期望分配内存的大小

align:对齐方式

```
/*
 * Find a free area with specified alignment in a specific range.
 * only with the area.between start to end is active range from early_node_map
 * so they are good as RAM
 */
u64 __init find_early_area(u64 ei_start, u64 ei_last, u64 start, u64 end,
                        u64 size, u64 align)
{
        u64 addr, last;

        addr = round_up(ei_start, align);
        if (addr < start)
                addr = round_up(start, align);
```

addr 必须属于[start,end]

```
        if (addr >= ei_last)
                goto out;
```

addr 必须不超过 ei_last

```
        while (bad_addr(&addr, size, align) && addr+size <= ei_last)
                ;
        last = addr + size;
        if (last > ei_last)
```

```
553                 goto out;
```

last 必须属于[ei_start,ei_last]

```
554         if (last > end)
555                 goto out;
556
557         return addr;
558
559 out:
560         return -1ULL;
561 }
```

上述函数的基本功能是：

在[ei_start,ei_last]区域分配一块大小为 size 的内存。如果 ei_last-ei_start<size，分配失败。在大于的情况下，还要保证分配内存的区间在[start,end]之间。

在上述函数的 549 行 bad_addr(&addr, size, align)函数我们还没有介绍，该函数的作用是检测[addr,addr+size]是否落与某个已经分配出去的内存区域存在交集。如果是则继续查找合适的区域。

4.1.7.1.1 冲突检测

与冲突检测的相关数据结构

```
/*
 * Early reserved memory areas.
 */
/*
 * need to make sure this one is bigger enough before
 * find_fw_memmap_area could be used
 */
#define MAX_EARLY_RES_X 32

struct early_res {
        u64 start, end;
        char name[15];
```

```
	char overlap_ok;
};
static struct early_res early_res_x[MAX_EARLY_RES_X] __initdata;

static int max_early_res __initdata = MAX_EARLY_RES_X;
static struct early_res *early_res __initdata = &early_res_x[0];
static int early_res_count __initdata;
```

内核定义了全局变量 early_res_x，该变量是一个包含了 32 个 early_res 结构体元素的变量，用来保存已经分配出去的内存区域。

early_res 变量指向该数组的起始地址。

max_early_res 表示数组的大小。

下面看看 early_res_x 中的元素是如何被设置的。

当调用__alloc_memory_core_early 函数分配内存后，会调用 reserve_early_checkout_check 函数将已分配的内存保存到 early_res_x 中。

```
void __init reserve_early_without_check(u64 start, u64 end, char *name)
{
	struct early_res *r;

	if (start >= end)
		return;

	__check_and_double_early_res(start, end);
```

如果 early_res 内存空间不足，则扩充内存。

```

	r = &early_res[early_res_count];

	r->start = start;
	r->end = end;
	r->overlap_ok = 0;
```

309-311 设置已分配内存区。

```
312         if (name)
313                 strncpy(r->name, name, sizeof(r->name) - 1);
314         early_res_count++;
315 }
```

4.1.7.2 kernel_physical_mapping_init物理内存页表项设置

函数所在文件 arch/x86/mm/init_32.c

函数参数含义：

start:起始虚拟地址

end：结束虚拟地址

page_size_mask:页表大小标志

```
238 unsigned long __init
239 kernel_physical_mapping_init(unsigned long start,
240                              unsigned long end,
241                              unsigned long page_size_mask)
242 {
243         int use_pse = page_size_mask == (1<<PG_LEVEL_2M);
244         unsigned long last_map_addr = end;
245         unsigned long start_pfn, end_pfn;
246         pgd_t *pgd_base = swapper_pg_dir;
```

246 行：pgd_base(页目录起始地址)页目录第一项

```
247         int pgd_idx, pmd_idx, pte_ofs;
248         unsigned long pfn;
249         pgd_t *pgd;
250         pmd_t *pmd;
251         pte_t *pte;
252         unsigned pages_2m, pages_4k;
253         int mapping_iter;
```

```
254
255         start_pfn = start >> PAGE_SHIFT;
256         end_pfn = end >> PAGE_SHIFT;
```

255-256 行：将虚拟地址装换为页面号

```
257
258         /*
259          * First iteration will setup identity mapping using large/small pages
260          * based on use_pse, with other attributes same as set by
261          * the early code in head_32.S
262          *
263          * Second iteration will setup the appropriate attributes (NX, GLOBAL..)
264          * as desired for the kernel identity mapping.
265          *
266          * This two pass mechanism conforms to the TLB app note which says:
267          *
268          *     "Software should not write to a paging-structure entry in a way
269          *      that would change, for any linear address, both the page size
270          *      and either the page frame or attributes."
271          */
272         mapping_iter = 1;
273
274         if (!cpu_has_pse)
275                 use_pse = 0;
```

检测 cpu 是否设置物理页面扩充位。

```
276
277 repeat:
278         pages_2m = pages_4k = 0;
```

```
        pfn = start_pfn;
        pgd_idx = pgd_index((pfn<<PAGE_SHIFT) + PAGE_OFFSET);
        pgd = pgd_base + pgd_idx;
```

pgd：保存虚拟地址对应的页目录项

```
        for (; pgd_idx < PTRS_PER_PGD; pgd++, pgd_idx++) {
                pmd = one_md_table_init(pgd);
```

在二级页表映射中，pmd=pgd

```

                if (pfn >= end_pfn)
                        continue;
#ifdef CONFIG_X86_PAE
                pmd_idx = pmd_index((pfn<<PAGE_SHIFT) + PAGE_OFFSET);
                pmd += pmd_idx;
#else
                pmd_idx = 0;
#endif
                for (; pmd_idx < PTRS_PER_PMD && pfn < end_pfn;
                     pmd++, pmd_idx++) {//在二级页表映射中，for 循环只执行一次
                        unsigned int addr = pfn * PAGE_SIZE + PAGE_OFFSET;

                        /*
                         * Map with big pages if possible, otherwise
                         * create normal page tables:
                         */
                        if (use_pse) {
                                unsigned int addr2;
                                pgprot_t prot = PAGE_KERNEL_LARGE;
                                /*
```

```
305                      * first pass will use the same initial
306                      * identity mapping attribute + _PAGE_PSE.
307                      */
308                     pgprot_t init_prot =
309                         __pgprot(PTE_IDENT_ATTR |
310                              _PAGE_PSE);
311
312                     addr2 = (pfn + PTRS_PER_PTE-1) * PAGE_SIZE
+
313                         PAGE_OFFSET + PAGE_SIZE-1;
314
315                     if (is_kernel_text(addr) ||
316                         is_kernel_text(addr2))
317                         prot = PAGE_KERNEL_LARGE_EXEC;
318
319                     pages_2m++;
320                     if (mapping_iter == 1)
321                         set_pmd(pmd, pfn_pmd(pfn, init_prot));
322                     else
323                         set_pmd(pmd, pfn_pmd(pfn, prot));
324
325                     pfn += PTRS_PER_PTE;
326                     continue;
327                 }
328                 pte = one_page_table_init(pmd);
```

328 行：为目录项分配页表

```
329
330                 pte_ofs = pte_index((pfn<<PAGE_SHIFT) +
PAGE_OFFSET);
331                 pte += pte_ofs;
```

330 行：求出虚拟地址在页表 pte 中的偏移

331 行：pte 中保存对应的页表项

```
332                     for (; pte_ofs < PTRS_PER_PTE && pfn < end_pfn;
333                          pte++, pfn++, pte_ofs++, addr += PAGE_SIZE) {
```

332-333 行：for 循环用来设置页表项，指向相应的物理页面的地址。

```
334                         pgprot_t prot = PAGE_KERNEL;
335                         /*
336                          * first pass will use the same initial
337                          * identity mapping attribute.
338                          */
339                         pgprot_t init_prot = __pgprot(PTE_IDENT_ATTR);
340
341                         if (is_kernel_text(addr))
342                             prot = PAGE_KERNEL_EXEC;
343
344                         pages_4k++;
345                         if (mapping_iter == 1) {
346                             set_pte(pte, pfn_pte(pfn, init_prot));
347                             last_map_addr = (pfn << PAGE_SHIFT) + PAGE_SIZE;
348                         } else
349                             set_pte(pte, pfn_pte(pfn, prot));
```

设置页表项。

```
350                     }
351                 }
352             }
353             if (mapping_iter == 1) {
354                 /*
355                  * update direct mapping page count only in the first
356                  * iteration.
357                  */
```

```
                update_page_count(PG_LEVEL_2M, pages_2m);
                update_page_count(PG_LEVEL_4K, pages_4k);

                /*
                 * local global flush tlb, which will flush the previous
                 * mappings present in both small and large page TLB's.
                 */
                __flush_tlb_all();

                /*
                 * Second iteration will set the actual desired PTE attributes.
                 */
                mapping_iter = 2;
                goto repeat;
        }
        return last_map_addr;
}
```

4.1.7.2.1 页表分配函数one_page_table_init

```
105  * Create a page table and place a pointer to it in a middle page
106  * directory entry:
107  */
108 static pte_t * __init one_page_table_init(pmd_t *pmd)
109 {
110         if (!(pmd_val(*pmd) & _PAGE_PRESENT)) {
111                 pte_t *page_table = NULL;
112
113                 if (after_bootmem) {
```

113 行：现在还未设置 after_bootmem

```
114 #if defined(CONFIG_DEBUG_PAGEALLOC) || defined(CONFIG_KMEMCHECK)
115                     page_table = (pte_t *) alloc_bootmem_pages(PAGE_SIZE);
116 #endif
117                     if (!page_table)
118                             page_table =
119                             (pte_t *)alloc_bootmem_pages(PAGE_SIZE);
120             } else
121                     page_table = (pte_t *)alloc_low_page();
122
123             paravirt_alloc_pte(&init_mm, __pa(page_table) >> PAGE_SHIFT);
124             set_pmd(pmd, __pmd(__pa(page_table) | _PAGE_TABLE));
```

124 行：pmd 保存刚分配页表的物理地址

```
125             BUG_ON(page_table != pte_offset_kernel(pmd, 0));
126     }
127
128     return pte_offset_kernel(pmd, 0);
129 }
```

在 bootmem 分配机制还没建立起来之前，该函数调用 alloc_low_page 分配页表

```
static __init void *alloc_low_page(void)
{
        unsigned long pfn = e820_table_end++;
```

在 find_early_table_space 中分配的页表项在这里派上了用场。

```
        void *adr;

        if (pfn >= e820_table_top)
                panic("alloc_low_page: ran out of memory");

```

```
69          adr = __va(pfn * PAGE_SIZE);
70          memset(adr, 0, PAGE_SIZE);
71          return adr;
```

69-71 行：将页面号转换为虚拟地址，将页表内容设置为零，返回页表的虚拟地址。

```
72 }
```

4.1.7.3 early_ioremap_page_table_range_init

固定内核映射区页表设置，不设置页表项。

该函数所在文件：

Arch/x86/mm/init_32.c

```
529 void __init early_ioremap_page_table_range_init(void)
530 {
531         pgd_t *pgd_base = swapper_pg_dir;
532         unsigned long vaddr, end;
533
534         /*
535          * Fixed mappings, only the page table structure has to be
536          * created - mappings will be set by set_fixmap():
537          */
538         vaddr = __fix_to_virt(__end_of_fixed_addresses - 1) & PMD_MASK;
539         end = (FIXADDR_TOP + PMD_SIZE - 1) & PMD_MASK;
```

538-539 行：固定内存区域起始地址与终止地址对齐设置

```
540         page_table_range_init(vaddr, end, pgd_base);
541         early_ioremap_reset();
542 }
```

4.1.7.4 load_cr3(swapper_pg_dir)

将控制 swapper_pg_dir 送入控制寄存器 cr3. 每当重新设置 cr3 时， CPU 就会将页面映射目录所在的页面装入 CPU 内部高速缓存中的 TLB 部分。现在内存中(实际上是高速缓存中）的映射目录变了，就要再让 CPU 装入一次。由于页面映射机制本来就是开启着的，所以从这条指令以后就扩大了系统空间中有映射区域的大小，使整个映射覆盖到整个物理内存(高端内存)除外。实际上此时 swapper_pg_dir 中已经改变的目录项很可能还在高速缓存中，所以还要通过 __flush_tlb_all()将高速缓存中的内容冲刷到内存中，这样才能保证内存中映射目录内容的一致性。最后，init_memory_mapping 返回了所映射页的数量值，并将值保存在 max_low_pfn_mapped 中。而 32 位 x86 体系中 max_pfn_mapped 的值也为 max_low_pfn_mapped。

4.1.8 initmem_init(0, max_pfn, acpi, k8)

该函数启用初始化期间的内存管理器 early。根据是否设置编译选项 CONFIG_NEED_MULTIPLE_NODES，该函数分为 UMA 版本和 NUMA 版本。尽管这两个版本的代码有较大差异，但是最后都殊途同归：建立 early_node_map 数据结构，存储内核可用内存（RAM）,为后面的内存管理做好准备。

下面分别讨论这里个版本的函数。

一、UMA 版本

该版本的函数在 arch/x86/mm/init_32.c 中：

```
#ifndef CONFIG_NEED_MULTIPLE_NODES
void __init initmem_init(unsigned long start_pfn, unsigned long end_pfn,
                                int acpi, int k8)
{
#ifdef CONFIG_HIGHMEM
        highstart_pfn = highend_pfn = max_pfn;
        if (max_pfn > max_low_pfn)
                highstart_pfn = max_low_pfn;
        e820_register_active_regions(0, 0, highend_pfn);
        sparse_memory_present_with_active_regions(0);
        printk(KERN_NOTICE "%ldMB HIGHMEM available.\n",
```

```
        pages_to_mb(highend_pfn - highstart_pfn));
    num_physpages = highend_pfn;
    high_memory = (void *) __va(highstart_pfn * PAGE_SIZE - 1) + 1;
#else
    e820_register_active_regions(0, 0, max_low_pfn);
    sparse_memory_present_with_active_regions(0);
    num_physpages = max_low_pfn;
    high_memory = (void *) __va(max_low_pfn * PAGE_SIZE - 1) + 1;
#endif
#ifdef CONFIG_FLATMEM
    max_mapnr = num_physpages;
#endif
    __vmalloc_start_set = true;

    printk(KERN_NOTICE "%ldMB LOWMEM available.\n",
            pages_to_mb(max_low_pfn));

    setup_bootmem_allocator();
}
#endif /* !CONFIG_NEED_MULTIPLE_NODES */
```

该函数调用 e820_register_active_regions，sparse_memory_present_with_active_regions 完成相关任务。

4.1.8.1 e820_register_active_regions

函数参数：

nid: cpu 节点号

start_pfn:起始页面号

last_pnf:终止页面号

```
931 /* Walk the e820 map and register active regions within a node */
932 void __init e820_register_active_regions(int nid, unsigned long start_pfn,
933                                     unsigned long last_pfn)
934 {
935         unsigned long ei_startpfn;
936         unsigned long ei_endpfn;
937         int i;
938
939         for (i = 0; i < e820.nr_map; i++)
940                 if (e820_find_active_region(&e820.map[i],
941                                     start_pfn, last_pfn,
942                                     &ei_startpfn, &ei_endpfn))
943                         add_active_range(nid, ei_startpfn, ei_endpfn);
944 }
```

940 行：调用 e820_find_active_region 检测 e820.map[i]是否为合法的 active region，如果是调用 add_active_range 将其加入到 early_node_map 中。

4.1.8.1.1 e820_find_active_region

函数参数：

ei：e820 中的一个类型为 RAM 的内存区段

start_pfn:节点起始页面号

last_pfn:节点终止页面号

ei_startpfn:输出参数，合法活动区域的起始页面号

ei_endpfn:输出参数，合法活动区域的终止页面号

```
899  * Finds an active region in the address range from start_pfn to last_pfn and
900  * returns its range in ei_startpfn and ei_endpfn for the e820 entry.
901  */
902 int __init e820_find_active_region(const struct e820entry *ei,
903                             unsigned long start_pfn,
904                             unsigned long last_pfn,
```

```
905                         unsigned long *ei_startpfn,
906                         unsigned long *ei_endpfn)
907 {
908         u64 align = PAGE_SIZE;
909
910         *ei_startpfn = round_up(ei->addr, align) >> PAGE_SHIFT;
911         *ei_endpfn = round_down(ei->addr + ei->size, align) >> PAGE_SHIFT;
```

E820 区间内存起始和终止节点页面对齐

```
912
913         /* Skip map entries smaller than a page */
914         if (*ei_startpfn >= *ei_endpfn)
915                 return 0;
```

914-915 行：略过小于一页的区间

```
916
917         /* Skip if map is outside the node */
918         if (ei->type != E820_RAM || *ei_endpfn <= start_pfn ||
919                                 *ei_startpfn >= last_pfn)
920                 return 0;
```

918-920 行：跳过不属于节点的内存区域

```
921
922         /* Check for overlaps */
923         if (*ei_startpfn < start_pfn)
924                 *ei_startpfn = start_pfn;
```

923-924 行：e820 中的起始页面号小于节点（node）的起始页面号，则 ei_startpfn 赋值为 node 起始页面号。

```
925         if (*ei_endpfn > last_pfn)
926                 *ei_endpfn = last_pfn;
```

925-926 行：e820 中的终止页面号大于节点（node）的终止页面号，则 ei_endpfn 赋值为终止页面号。

```
927
```

```
        return 1;
}
```

4.1.8.1.2 add_active_range

该函数所在文件为 mm/page_alloc.c 中：
从注释中可以看出：
nid:CPU 节点号
start_pfn：可用物理内存的起始页面号
end_pfn：终止节点的起始页面号

该函数的功能：将[start_pfn,end_pfn]对应的范围保存到 early_node_map 数组中，在后面的过程中该数组将会被 free_area_init_nodes 调用，用于计算各个区域的大小以及是否存在空洞。

```
/**
 * add_active_range - Register a range of PFNs backed by physical memory
 * @nid: The node ID the range resides on
 * @start_pfn: The start PFN of the available physical memory
 * @end_pfn: The end PFN of the available physical memory
 *
 * These ranges are stored in an early_node_map[] and later used by
 * free_area_init_nodes() to calculate zone sizes and holes. If the
 * range spans a memory hole, it is up to the architecture to ensure
 * the memory is not freed by the bootmem allocator. If possible
 * the range being registered will be merged with existing ranges.
 */
void __init add_active_range(unsigned int nid, unsigned long start_pfn,
                                        unsigned long end_pfn)
{
        int i;
```

```
3988
3989	mminit_dprintk(MMINIT_TRACE, "memory_register",
3990			"Entering add_active_range(%d, %#lx, %#lx) "
3991			"%d entries of %d used\n",
3992			nid, start_pfn, end_pfn,
3993			nr_nodemap_entries, MAX_ACTIVE_REGIONS);
3994
3995	mminit_validate_memmodel_limits(&start_pfn, &end_pfn);
3996
3997	/* Merge with existing active regions if possible */
3998	for (i = 0; i < nr_nodemap_entries; i++) {
3999		if (early_node_map[i].nid != nid)
4000			continue;
```

3999-4000：节点号不匹配，跳入下一次循环

```
4001
4002		/* Skip if an existing region covers this new one */
4003		if (start_pfn >= early_node_map[i].start_pfn &&
4004				end_pfn <= early_node_map[i].end_pfn)
4005			return;
```

4003-4005：已经存在的区域，直接返回

```
4006
4007		/* Merge forward if suitable */
4008		if (start_pfn <= early_node_map[i].end_pfn &&
4009				end_pfn > early_node_map[i].end_pfn) {
4010			early_node_map[i].end_pfn = end_pfn;
4011			return;
4012		}
```

4008-4012：前向合并

```
4013
4014		/* Merge backward if suitable */
4015		if (start_pfn < early_node_map[i].start_pfn &&
```

```
4016                        end_pfn >= early_node_map[i].start_pfn) {
4017                    early_node_map[i].start_pfn = start_pfn;
4018                    return;
4019            }
```

4014-4019：后向合并

```
4020        }
4021
4022        /* Check that early_node_map is large enough */
4023        if (i >= MAX_ACTIVE_REGIONS) {
4024            printk(KERN_CRIT "More than %d memory regions,
truncating\n",
4025                                    MAX_ACTIVE_REGIONS);
4026            return;
4027        }
4028
4029        early_node_map[i].nid = nid;
4030        early_node_map[i].start_pfn = start_pfn;
4031        early_node_map[i].end_pfn = end_pfn;
4032        nr_nodemap_entries = i + 1;
```

4029-4032：如果无法合并，这建立一个新的项，同时将 nr_nodemap_entries 加一

```
4033 }
```

有一个小小的疑问：为啥不考虑

start_pfn <= early_node_map[i].start_pfn &&

end_pfn > early_node_map[i].end_pfn 的情况？

4.1.8.2 sparse_memory_present_with_active_regions

该函数所在文件为 mm/page_alloc.c 中：

```
3463 /**
3464  * sparse_memory_present_with_active_regions - Call
memory_present for each active range
```

```
 * @nid: The node to call memory_present for. If MAX_NUMNODES, all nodes will be used.
 *
 * If an architecture guarantees that all ranges registered with
 * add_active_ranges() contain no holes and may be freed, this
 * function may be used instead of calling memory_present() manually.
 */
void __init sparse_memory_present_with_active_regions(int nid)
{
	int i;

	for_each_active_range_index_in_nid(i, nid)
		memory_present(early_node_map[i].nid,
				early_node_map[i].start_pfn,
				early_node_map[i].end_pfn);
}
```

该函数使用 for_each_active_range_index_in_nid(i, nid)宏（具体定义见下），探测 early_node_map 中节点号为 nid 的所有内存区域。memory_present 函数就不细看了。

```
/* Basic iterator support to walk early_node_map[] */
#define for_each_active_range_index_in_nid(i, nid) \
	for (i = first_active_region_index_in_nid(nid); i != -1; \
				i = next_active_region_index_in_nid(i, nid))

```

```
 * Basic iterator support. Return the first range of PFNs for a node
 * Note: nid == MAX_NUMNODES returns first region regardless of node
 */
static int __meminit first_active_region_index_in_nid(int nid)
```

```
3291 {
3292         int i;
3293
3294         for (i = 0; i < nr_nodemap_entries; i++)
3295                 if (nid == MAX_NUMNODES || early_node_map[i].nid == nid)
3296                         return i;
```

3295：如果 nid 不是用来指定 cpu 节点号，则直接返回 0，如果 early_node_map 中有 nid 的内存区域，则返回第一个匹配的内存区域号。

```
3297
3298         return -1;
3299 }
```

```
3302    * Basic iterator support. Return the next active range of PFNs for a node
3303    * Note: nid == MAX_NUMNODES returns next region regardless of node
3304    */
3305 static int __meminit next_active_region_index_in_nid(int index, int nid)
3306 {
3307         for (index = index + 1; index < nr_nodemap_entries; index++)
3308                 if (nid == MAX_NUMNODES || early_node_map[index].nid == nid)
3309                         return index;
```

该函数的基本功能同上，只不过是查找的起始号从传入的参数 index 开始。

```
3310
3311         return -1;
3312 }
```

4.1.8.3 setup_bootmem_allocator()

```
void __init setup_bootmem_allocator(void)
{
#ifndef CONFIG_NO_BOOTMEM
	int nodeid;
	unsigned long bootmap_size, bootmap;
	/*
	 * Initialize the boot-time allocator (with low memory only):
	 */
	bootmap_size = bootmem_bootmap_pages(max_low_pfn)<<PAGE_SHIFT;
	bootmap = find_e820_area(0, max_pfn_mapped<<PAGE_SHIFT, bootmap_size,
				 PAGE_SIZE);
	if (bootmap == -1L)
		panic("Cannot find bootmem map of size %ld\n", bootmap_size);
	reserve_early(bootmap, bootmap + bootmap_size, "BOOTMAP");
#endif

	printk(KERN_INFO "  mapped low ram: 0 - %08lx\n",
		 max_pfn_mapped<<PAGE_SHIFT);
	printk(KERN_INFO "  low ram: 0 - %08lx\n", max_low_pfn<<PAGE_SHIFT);

#ifndef CONFIG_NO_BOOTMEM
	for_each_online_node(nodeid) {
		 unsigned long start_pfn, end_pfn;

#ifdef CONFIG_NEED_MULTIPLE_NODES
```

```
		start_pfn = node_start_pfn[nodeid];
		end_pfn = node_end_pfn[nodeid];
		if (start_pfn > max_low_pfn)
			continue;
		if (end_pfn > max_low_pfn)
			end_pfn = max_low_pfn;
#else
		start_pfn = 0;
		end_pfn = max_low_pfn;
#endif
		bootmap = setup_node_bootmem(nodeid, start_pfn, end_pfn,
						bootmap);
	}
#endif

	after_bootmem = 1;
}
```

该函数在设置了 CONFIG_NO_BOOTMEM 后，基本上啥事也不敢，只是将 after_bootmem 设置为 1。现在一般弃用 BOOTMEM，所以没有设置 CONFIG_NO_BOOTMEM 的过程这里不讨论了。

二 、NUMA 版

NUMA 版的函数在 arch/x86/mm/numa_32.c 中：

```
void __init initmem_init(unsigned long start_pfn, unsigned long end_pfn,
				int acpi, int k8)
{
	int nid;
	long kva_target_pfn;

	/*
```

```
	 * When mapping a NUMA machine we allocate the node_mem_map arrays
	 * from node local memory.  They are then mapped directly into KVA
	 * between zone normal and vmalloc space.  Calculate the size of
	 * this space and use it to adjust the boundary between ZONE_NORMAL
	 * and ZONE_HIGHMEM.
	 */

	get_memcfg_numa();

	kva_pages = roundup(calculate_numa_remap_pages(), PTRS_PER_PTE);

	kva_target_pfn = round_down(max_low_pfn - kva_pages, PTRS_PER_PTE);
	do {
		kva_start_pfn = find_e820_area(kva_target_pfn<<PAGE_SHIFT,
					max_low_pfn<<PAGE_SHIFT,
					kva_pages<<PAGE_SHIFT,
					PTRS_PER_PTE<<PAGE_SHIFT) >> PAGE_SHIFT;
		kva_target_pfn -= PTRS_PER_PTE;
	} while (kva_start_pfn == -1UL && kva_target_pfn > min_low_pfn);

	if (kva_start_pfn == -1UL)
		panic("Can not get kva space\n");
```

```
	printk(KERN_INFO "kva_start_pfn ~ %lx max_low_pfn ~ %lx\n",
		kva_start_pfn, max_low_pfn);
	printk(KERN_INFO "max_pfn = %lx\n", max_pfn);

	/* avoid clash with initrd */
	reserve_early(kva_start_pfn<<PAGE_SHIFT,
			(kva_start_pfn + kva_pages)<<PAGE_SHIFT,
			"KVA PG");
#ifdef CONFIG_HIGHMEM
	highstart_pfn = highend_pfn = max_pfn;
	if (max_pfn > max_low_pfn)
		highstart_pfn = max_low_pfn;
	printk(KERN_NOTICE "%ldMB HIGHMEM available.\n",
			pages_to_mb(highend_pfn - highstart_pfn));
	num_physpages = highend_pfn;
	high_memory = (void *) __va(highstart_pfn * PAGE_SIZE - 1) + 1;
#else
	num_physpages = max_low_pfn;
	high_memory = (void *) __va(max_low_pfn * PAGE_SIZE - 1) + 1;
#endif
	printk(KERN_NOTICE "%ldMB LOWMEM available.\n",
			pages_to_mb(max_low_pfn));
	printk(KERN_DEBUG "max_low_pfn = %lx, highstart_pfn = %lx\n",
			max_low_pfn, highstart_pfn);

	printk(KERN_DEBUG "Low memory ends at vaddr %08lx\n",
			(ulong) pfn_to_kaddr(max_low_pfn));
	for_each_online_node(nid) {
		init_remap_allocator(nid);
```

```

                allocate_pgdat(nid);
        }
        remap_numa_kva();

        printk(KERN_DEBUG "High memory starts at vaddr %08lx\n",
                        (ulong) pfn_to_kaddr(highstart_pfn));
        for_each_online_node(nid)
                propagate_e820_map_node(nid);

        for_each_online_node(nid) {
                memset(NODE_DATA(nid), 0, sizeof(struct pglist_data));
                NODE_DATA(nid)->node_id = nid;
#ifndef CONFIG_NO_BOOTMEM
                NODE_DATA(nid)->bdata = &bootmem_node_data[nid];
#endif
        }

        setup_bootmem_allocator();
}
```

4.1.8.4 acpi_numa_init

要查看该函数，需要回去看看 Setup_arch 函数中的第 976 行的 acpi_numa_init 函数。

```
972 #ifdef CONFIG_ACPI_NUMA
973         /*
974          * Parse SRAT to discover nodes.
975          */
976         acpi = acpi_numa_init();
977 #endif
```

该函数所在的文件为 drivers/acpi/numa.c 中：

```
int __init acpi_numa_init(void)
{
	int ret = 0;

	/* SRAT: Static Resource Affinity Table */
	if (!acpi_table_parse(ACPI_SIG_SRAT, acpi_parse_srat)) {
		acpi_table_parse_srat(ACPI_SRAT_TYPE_X2APIC_CPU_AFFINITY,
				     acpi_parse_x2apic_affinity, nr_cpu_ids);
		acpi_table_parse_srat(ACPI_SRAT_TYPE_CPU_AFFINITY,
				     acpi_parse_processor_affinity, nr_cpu_ids);
		ret = acpi_table_parse_srat(ACPI_SRAT_TYPE_MEMORY_AFFINITY,
					    acpi_parse_memory_affinity,
					    NR_NODE_MEMBLKS);
	}
```

该函数使用 ACPI 的 SRAT 来处理和内存与处理器相关的函数，这里只关心 acpi_parse_memory_affinity 函数。

```
static int __init
acpi_parse_memory_affinity(struct acpi_subtable_header * header,
			   const unsigned long end)
{
	struct acpi_srat_mem_affinity *memory_affinity;

	memory_affinity = (struct acpi_srat_mem_affinity *)header;
	if (!memory_affinity)
		return -EINVAL;

	acpi_table_print_srat_entry(header);

```

```
        /* let architecture-dependent part to do it */
        acpi_numa_memory_affinity_init(memory_affinity);

        return 0;
}
```

该函数调用 acpi_numa_memory_affinity_init 函数，函数所在文件 arch/x86/mm/srat_32.c

该函数是供 ACPI 驱动调用的。用来设置 node_memory_chunk 全局变量。该变量的定义如下：

```
47 #define MAX_CHUNKS_PER_NODE 3
48 #define MAXCHUNKS           (MAX_CHUNKS_PER_NODE * MAX_NUMNODES)
49 struct node_memory_chunk_s {
50         unsigned long   start_pfn; //起始页面号
51         unsigned long   end_pfn; //终止页面号
52         u8   pxm;           // proximity domain of node 邻近域的节点
53         u8   nid;           // which cnode contains this chunk?
54         u8   bank;          // which mem bank on this node
55 };
56 static struct node_memory_chunk_s __initdata node_memory_chunk[MAXCHUNKS];
```

56 行：全局变量 node_memory_chunk，记录每个内存 chunk 的数组

```
57
58 static int __initdata num_memory_chunks; /* total number of memory chunks */
```

58 行：全局变量 num_memory_chunks，记录总的内存 chunk 的数目。

下面看函数的具体代码：

```
acpi_numa_memory_affinity_init(struct acpi_srat_mem_affinity *memory_affinity)
{
        unsigned long long paddr, size;
```

```
	unsigned long start_pfn, end_pfn;
	u8 pxm;
	struct node_memory_chunk_s *p, *q, *pend;

	if (srat_disabled())
		return;
	if (memory_affinity->header.length !=
	     sizeof(struct acpi_srat_mem_affinity)) {
		bad_srat();
		return;
	}

	if ((memory_affinity->flags & ACPI_SRAT_MEM_ENABLED) == 0)
		return;		/* empty entry */

	pxm = memory_affinity->proximity_domain & 0xff;

	/* mark this node as "seen" in node bitmap */
	BMAP_SET(pxm_bitmap, pxm);

	/* calculate info for memory chunk structure */
	paddr = memory_affinity->base_address;
	size = memory_affinity->length;

	start_pfn = paddr >> PAGE_SHIFT;
	end_pfn = (paddr + size) >> PAGE_SHIFT;


	if (num_memory_chunks >= MAXCHUNKS) {
		printk(KERN_WARNING "Too many mem chunks in SRAT."
			" Ignoring %lld MBytes at %llx\n",
```

```
139                 size/(1024*1024), paddr);
140             return;
141         }
```

136-141：如果 num_memory_chunks 超过内核设置上限，则退出函数

```
142
143         /* Insertion sort based on base address */
144         pend = &node_memory_chunk[num_memory_chunks];
145         for (p = &node_memory_chunk[0]; p < pend; p++) {
146             if (start_pfn < p->start_pfn)
147                 break;
148         }
149         if (p < pend) {
150             for (q = pend; q >= p; q--)
151                 *(q + 1) = *q;
152         }
153         p->start_pfn = start_pfn;
154         p->end_pfn = end_pfn;
155         p->pxm = pxm;
156
157         num_memory_chunks++;
```

153-157 行：设置 chunk 的各个变量。

```
158
159         printk(KERN_DEBUG "Memory range %08lx to %08lx"
160                   " in proximity domain %02x %s\n",
161             start_pfn, end_pfn,
162             pxm,
163             ((memory_affinity->flags &
ACPI_SRAT_MEM_HOT_PLUGGABLE) ?
164              "enabled and removable" : "enabled" ) );
165 }
```

4.1.8.5 get_memcfg_numa()

在 acpi_numa_init 中设置了 node_memory_chunk 后，get_memcfg_numa 将使用改变了来初始化 node_early_map 变量。

该函数所在的文件为：arch/x86/include/asm/mmzone_32.h

```
 * This allows any one NUMA architecture to be compiled
 * for, and still fall back to the flat function if it
 * fails.
 */
static inline void get_memcfg_numa(void)
{

	if (get_memcfg_numaq()) //IBM NUMA-Q
		return;
	if (get_memcfg_from_srat()) //intel srat
		return;
	get_memcfg_numa_flat();
}
```

INTEL 处理器通过 ACPI 规范对硬件资源进行管理。其中 SRAT 给出了全系统中的 CPU 和 MEM 的分布情况。上面 get_memcfg_from_srat()就是该过程的实现。下面只分析 get_memcfg_from_srat()函数的实现：

该函数所在的文件为 arch/x86/mm/srat_32.c

```
209 int __init get_memcfg_from_srat(void)
210 {
211 	int i, j, nid;
212
213
214 	if (srat_disabled())
215 		goto out_fail;
```

214-215：检测是否支持 srat。

```
216
```

```
217         if (num_memory_chunks == 0) {
218                 printk(KERN_DEBUG
219                          "could not find any ACPI SRAT memory areas.\n");
220                 goto out_fail;
221         }
```

217-221：如果 srat 内存区域为 0，则跳转到 out_fail

```
222
223         /* Calculate total number of nodes in system from PXM bitmap and create
224          * a set of sequential node IDs starting at zero.  (ACPI doesn't seem
225          * to specify the range of _PXM values.)
226          */
227         /*
228          * MCD - we no longer HAVE to number nodes sequentially.  PXM domain
229          * numbers could go as high as 256, and MAX_NUMNODES for i386 is typically
230          * 32, so we will continue numbering them in this manner until MAX_NUMNODES
231          * approaches MAX_PXM_DOMAINS for i386.
232          */
233         nodes_clear(node_online_map);
234         for (i = 0; i < MAX_PXM_DOMAINS; i++) {
235                 if (BMAP_TEST(pxm_bitmap, i)) {
236                         int nid = acpi_map_pxm_to_node(i);
237                         node_set_online(nid);
238                 }
239         }
240         BUG_ON(num_online_nodes() == 0);
241
242         /* set cnode id in memory chunk structure */
```

```
243 	for (i = 0; i < num_memory_chunks; i++)
244 		node_memory_chunk[i].nid = pxm_to_node(node_memory_chunk[i].pxm);
245 
printk(KERN_DEBUG "pxm bitmap: ");
247 	for (i = 0; i < sizeof(pxm_bitmap); i++) {
248 		printk(KERN_CONT "%02x ", pxm_bitmap[i]);
249 	}
250 	printk(KERN_CONT "\n");
251 	printk(KERN_DEBUG "Number of logical nodes in system = %d\n",
252 			num_online_nodes());
253 	printk(KERN_DEBUG "Number of memory chunks in system = %d\n",
254 			num_memory_chunks);
255 
256 	for (i = 0; i < MAX_APICID; i++)
257 		apicid_2_node[i] = pxm_to_node(apicid_to_pxm[i]);
```

233-257 行：初始化一系列变量后，来到 259 行

```
258 
259 	for (j = 0; j < num_memory_chunks; j++){
260 		struct node_memory_chunk_s * chunk = &node_memory_chunk[j];
261 		printk(KERN_DEBUG
262 			"chunk %d nid %d start_pfn %08lx end_pfn %08lx\n",
263 			 j, chunk->nid, chunk->start_pfn, chunk->end_pfn);
264 		if (node_read_chunk(chunk->nid, chunk))
265 			continue;
```

264-265 行：验证 chunk 的有效性，有需要的话修改节点 node 的 node_start_pfn 变量。

```
266 
```

```
267             e820_register_active_regions(chunk->nid,
chunk->start_pfn,
268                                     min(chunk->end_pfn, max_pfn));
269         }
```

267-269 行：根据 chunk 初始化 early_node_map 全局变量，该函数在前面已经分析过。

```
270         /* for out of order entries in SRAT */
271         sort_node_map();
272
273         for_each_online_node(nid) {
274             unsigned long start = node_start_pfn[nid];
275             unsigned long end = min(node_end_pfn[nid], max_pfn);
276
277             memory_present(nid, start, end);
278             node_remap_size[nid] = node_memmap_size_bytes(nid,
start, end);
279         }
280         return 1;
281 out_fail:
282         printk(KERN_DEBUG "failed to get NUMA memory information
from SRAT"
283                 " table\n");
284         return 0;
285 }
```

NUMA 函数版的 initmem_init 深入的函数在这就不在分析了。

对比一下 UMA 和 NUMA 的 initmem_init 函数：

UMA 的使用 E820 建立的数据结构，来初始化 early_node_map 全局变量。

NUMA 使用 ACPI 的 SRAT 建立的数据结构，来初始化 early_node_map 全局变量。

当记录内存区域的变量 early_node_map 建立好后，在其基础上调用 paging_init 建立物理内存管理的相关数据结构。

4.1.9 paging_init

该函数所在的文件为 arch/x86/mm/init_32.c

```
/*
 * paging_init() sets up the page tables - note that the first 8MB are
 * already mapped by head.S.
 *
 * This routines also unmaps the page at virtual kernel address 0, so
 * that we can trap those pesky NULL-reference errors in the kernel.
 */
void __init paging_init(void)
{
        pagetable_init();

        __flush_tlb_all();

        kmap_init();

        /*
         * NOTE: at this point the bootmem allocator is fully available.
         */
        sparse_init();
        zone_sizes_init();
}
```

该函数分别调用：

Pagetable_init：初始化永久内核页表

Kmap_init：临时内核映射

Sparse_init 初始化

zone_sizes_init：内存区域数据结构初始化。

4.1.9.1 永久内核页表分配

pagetable_init：初始化永久内核页表。分配一个页框存储页表，但页表中的页表项并没有指向相应的页框。

```
544 static void __init pagetable_init(void)
545 {
546         pgd_t *pgd_base = swapper_pg_dir; //pgd_base 指向页目录
547
548         permanent_kmaps_init(pgd_base);
549 }
```

546 行：pgd_base 指向全局页目录表

548 行：调用 permanent_kmaps_init 执行永久内核区域的初始化

```
#ifdef CONFIG_HIGHMEM
static void __init permanent_kmaps_init(pgd_t *pgd_base)
{
        unsigned long vaddr;
        pgd_t *pgd;
        pud_t *pud;
        pmd_t *pmd;
        pte_t *pte;

        #define PKMAP_BASE ((FIXADDR_BOOT_START - PAGE_SIZE * (LAST_PKMAP + 1))        & PMD_MASK)
        vaddr = PKMAP_BASE;//永久内核映射虚拟地址开始区
        //LAST_PKMAP=1024 PAGE_SIZE=4K
        page_table_range_init(vaddr, vaddr + PAGE_SIZE*LAST_PKMAP, pgd_base);

        pgd = swapper_pg_dir + pgd_index(vaddr);
        pud = pud_offset(pgd, vaddr);
        pmd = pmd_offset(pud, vaddr);
```

```
413         pte = pte_offset_kernel(pmd, vaddr);
414         pkmap_page_table = pte;
415 }
```

407 行：vaddr 是永久内核映射的起始地址。被赋值为 PKMAP_BASE。PKMAP_BASE 的定义如下：

#define PKMAP_BASE ((FIXADDR_BOOT_START - PAGE_SIZE * (LAST_PKMAP + 1))   & PMD_MASK)

其中，FIXADDR_BOOT_START 为固定映射的起始虚拟地址，LASK_PKMAP 为 1024，整个表达式的含义是 永久内核映射的起始地址 为：固定映射的起始地址 -4M-4K，并且地址为 PMD_MASK 对齐。之所以在永久内核映射与固定映射之间有 4K 的空隙，适用于内核捕获内存访问异常使用的。

408 行：调用 page_table_range_init 函数，为虚拟地址[vaddr,vaddr+4M]在全局页目录表中设置页目录项，并且将页目录项指向分配的页表。

410-414 行：在全局页目录表中设置好页目录项后，通过 pud_offset、pmd_offset、 pte_offset_kernel 函数的到分配的页表的虚拟地址，并且将虚拟地址赋值给全局变量 pkmap_page_table。

这里需要重点讲解一下 pte_offset_kernel 函数。

我们知道，在页目录中的页目录项、页表中的页表项存放的都是对应页框的物理地址，pte_offset_kernel 函数如下：

```
static inline pte_t *pte_offset_kernel(pmd_t *pmd, unsigned long address)
{
        return (pte_t *)pmd_page_vaddr(*pmd) + pte_index(address);
}
```

(pte_t *)pmd_page_vaddr(*pmd)将也中间目录项的物理地址转换为虚拟地址，pte_index(address)的到虚拟地址 address 在 pmd 指向的页表中的偏移。

整个函数的作用：虚拟地址 address 在 pmd（物理地址）指向的页表中有一个对应的表项，有一个指向这个表项的指针 p_pte，pte_offset_kernel 的作用就是将 p_pte(物理地址)转换为虚拟地址。

page_table_range_init 为虚拟地址[vaddr,vaddr+4M]在全局页目录表中设置页目录项，并且将页目录项指向分配的页表。

```
/*
 * This function initializes a certain range of kernel virtual memory
```

```
 * with new bootmem page tables, everywhere page tables are missing in
 * the given range.
 *
 * NOTE: The pagetables are allocated contiguous on the physical space
 * so we can cache the place of the first one and move around without
 * checking the pgd every time.
 */
static void __init
page_table_range_init(unsigned long start, unsigned long end, pgd_t *pgd_base)
{
        int pgd_idx, pmd_idx;
        unsigned long vaddr;
        pgd_t *pgd;
        pmd_t *pmd;
        pte_t *pte = NULL;

        vaddr = start;
        pgd_idx = pgd_index(vaddr);    //页目录索引
        pmd_idx = pmd_index(vaddr);//中间页表索引
        pgd = pgd_base + pgd_idx;
//内外 for 循环都指执行一次
        for ( ; (pgd_idx < PTRS_PER_PGD) && (vaddr != end); pgd++, pgd_idx++) {
                pmd = one_md_table_init(pgd);//pmd  指向中间页表
                pmd = pmd + pmd_index(vaddr);//中间页表项
                for (; (pmd_idx < PTRS_PER_PMD) && (vaddr != end);
         pmd++, pmd_idx++) {
           pte=page_table_kmap_check(one_page_table_init(pmd),
```

```
218                                                 pmd, vaddr, pte);
219
220                 vaddr += PMD_SIZE;
221         }
222         pmd_idx = 0;
223     }
224 }
```

212 215 内外 for 循环都只是执行一次。

213 行：one_md_table_init 函数在没有启用物理地址扩展选项，并且内核采用二级页目录时，只是将 pgd 赋值给 pmd。

217 行：one_page_table_init(pmd)：分配一个页框，并将页框的物理地址赋值给 pmd 指向的页目录项（二级页面寻址）。

```
 * Create a page table and place a pointer to it in a middle page
 * directory entry:
 */
static pte_t * __init one_page_table_init(pmd_t *pmd)
{
    if (!(pmd_val(*pmd) & _PAGE_PRESENT)) {
        pte_t *page_table = NULL;

        if (after_bootmem) {
#if defined(CONFIG_DEBUG_PAGEALLOC) || defined(CONFIG_KMEMCHECK)
            page_table = (pte_t *) alloc_bootmem_pages(PAGE_SIZE);
#endif
            if (!page_table)
                page_table =
                (pte_t *)alloc_bootmem_pages(PAGE_SIZE);
        } else
```

```
            page_table = (pte_t *)alloc_low_page();//分配一个页框，用来作为页表

        paravirt_alloc_pte(&init_mm, __pa(page_table) >> PAGE_SHIFT);
        set_pmd(pmd, __pmd(__pa(page_table) | _PAGE_TABLE));
        BUG_ON(page_table != pte_offset_kernel(pmd, 0));
    }

    return pte_offset_kernel(pmd, 0);
}
```

4.1.9.2 临时内核映射初始化

kmap_init()初始化临时内核映射。

```
385 static void __init kmap_init(void)
386 {
387         unsigned long kmap_vstart;
388 
389         /*
390          * Cache the first kmap pte:
391          */
392         kmap_vstart = __fix_to_virt(FIX_KMAP_BEGIN);
393         kmap_pte = kmap_get_fixmap_pte(kmap_vstart);
394 
395         kmap_prot = PAGE_KERNEL;
396 }
```

392 行：获得临时内核映射虚拟地址的起始地址。

393 行：将临时内核映射的第一个页表的虚拟地址保存在全局变量 kmap_pte 中。

4.1.9.3 内存区域初始化

4.1.9.3.1 zone_size_init

Linux 内核采用一个三级的结构来管理内存，从节点->区域->页面。
Zone_size_init 用来初始化节点、区域中的部分数据结构。

```
#define MAX_DMA_ADDRESS         (PAGE_OFFSET + 0x1000000)
739 static void __init zone_sizes_init(void)
740 {
741         unsigned long max_zone_pfns[MAX_NR_ZONES];
742         memset(max_zone_pfns, 0, sizeof(max_zone_pfns));
743         max_zone_pfns[ZONE_DMA] =
744                 virt_to_phys((char *)MAX_DMA_ADDRESS) >> PAGE_SHIFT;
745         max_zone_pfns[ZONE_NORMAL] = max_low_pfn;
746 #ifdef CONFIG_HIGHMEM
747         max_zone_pfns[ZONE_HIGHMEM] = highend_pfn;
748 #endif
749
750         free_area_init_nodes(max_zone_pfns);
751 }
```

Zone_sizes_init 的主要作用是初始化 max_zone_pfns 数组，然后将 max_zone_pfns 作为参数调用 free_area_init_nodes 函数。

743、745、747 行分别设置 DMA 区、NORMAL 区、HIGHMEM 的边界。

DMA 区的边界为：0-16M。

NORMAL 的边界为:16M-max_low_pfn*4k，max_low_pfn 是在前面内核初始化过程中计算出来的，如果有高端内存，则 max_low_pfs*4k 为 896M。

HIGHMEM 的边界为：896M-高端内存的结尾（小于 4G）。

4.1.9.3.2 free_area_init_ndoes

free_area_init_nodes 函数设置必要的变量后，初始化每一个节点。

```
Mm/page_alloc. C
```

```
4346  * free_area_init_nodes - Initialise all pg_data_t and zone data
4347  * @max_zone_pfn: an array of max PFNs for each zone
4348  *
4349  * This will call free_area_init_node() for each active node in the system.
4350  * Using the page ranges provided by add_active_range(), the size of each
4351  * zone in each node and their holes is calculated. If the maximum PFN
4352  * between two adjacent zones match, it is assumed that the zone is empty.
4353  * For example, if arch_max_dma_pfn == arch_max_dma32_pfn, it is assumed
4354  * that arch_max_dma32_pfn has no pages. It is also assumed that a zone
4355  * starts where the previous one ended. For example, ZONE_DMA32 starts
4356  * at arch_max_dma_pfn.
4357  */
4358 void __init free_area_init_nodes(unsigned long *max_zone_pfn)
4359 {
4360         unsigned long nid;
4361         int i;
4362
4363         /* Sort early_node_map as initialisation assumes it is sorted */
4364         sort_node_map();
4365
4366         /* Record where the zone boundaries are */
4367         memset(arch_zone_lowest_possible_pfn, 0,
4368                                 sizeof(arch_zone_lowest_possible_pfn));
4369         memset(arch_zone_highest_possible_pfn, 0,
4370                                 sizeof(arch_zone_highest_possible_pfn));
```

4367、4369 初始化静态数组 arch_zone_lowest_possible_pfn，arch_zone_highest_possible_pfn，两个数组同一索引出的元素分别记录某一个内存区（zone）的起始页面号和终止页面号。

```
static unsigned long __meminitdata
arch_zone_lowest_possible_pfn[MAX_NR_ZONES];
```

```
static unsigned long __meminitdata arch_zone_highest_possible_pfn[MAX_NR_ZONES];
```

```
	arch_zone_lowest_possible_pfn[0] = find_min_pfn_with_active_regions();
```

find_min_pfn_with_active_regions 函数调用 find_min_pfn_for_node 函数查找指定（该处为所有节点）节点中最小的页面号。

```
4136 /* Find the lowest pfn for a node */
4137 static unsigned long __init find_min_pfn_for_node(int nid)
4138 {
4139 	int i;
4140 	unsigned long min_pfn = ULONG_MAX;
4141 
4142 	/* Assuming a sorted map, the first range found has the starting pfn */
4143 	for_each_active_range_index_in_nid(i, nid)
4144 		min_pfn = min(min_pfn, early_node_map[i].start_pfn);
4145 
4146 	if (min_pfn == ULONG_MAX) {
4147 		printk(KERN_WARNING
4148 			"Could not find start_pfn for node %d\n", nid);
4149 		return 0;
4150 	}
4151 
4152 	return min_pfn;
4153 }
```

4143-4144 行：使用内核在之前初始化的数据结构 early_node_map，查找所有节点中页面号最小的页面。

```
	arch_zone_highest_possible_pfn[0] = max_zone_pfn[0];
	for (i = 1; i < MAX_NR_ZONES; i++) {
		if (i == ZONE_MOVABLE)
			continue;
```

```
4376            arch_zone_lowest_possible_pfn[i] =
4377                arch_zone_highest_possible_pfn[i-1];
4378            arch_zone_highest_possible_pfn[i] =
4379                max(max_zone_pfn[i], arch_zone_lowest_possible_pfn[i]);
4380        }
```

4372-4380：根据 max_zone_pfn 来计算各初始化数组。

```
4381        arch_zone_lowest_possible_pfn[ZONE_MOVABLE] = 0;
4382        arch_zone_highest_possible_pfn[ZONE_MOVABLE] = 0;
4383
4384        /* Find the PFNs that ZONE_MOVABLE begins at in each node */
4385        memset(zone_movable_pfn, 0, sizeof(zone_movable_pfn));
4386        find_zone_movable_pfns_for_nodes(zone_movable_pfn);
4387
4388        /* Print out the zone ranges */
4389        printk("Zone PFN ranges:\n");
4390        for (i = 0; i < MAX_NR_ZONES; i++) {
4391            if (i == ZONE_MOVABLE)
4392                continue;
4393            printk("    %-8s ", zone_names[i]);
4394            if (arch_zone_lowest_possible_pfn[i] ==
4395                    arch_zone_highest_possible_pfn[i])
4396                printk("empty\n");
4397            else
4398                printk("%0#10lx -> %0#10lx\n",
4399                    arch_zone_lowest_possible_pfn[i],
4400                    arch_zone_highest_possible_pfn[i]);
4401        }
4402
4403        /* Print out the PFNs ZONE_MOVABLE begins at in each node */
4404        printk("Movable zone start PFN for each node\n");
4405        for (i = 0; i < MAX_NUMNODES; i++) {
```

```
4406            if (zone_movable_pfn[i])
4407                printk("    Node %d: %lu\n", i, zone_movable_pfn[i]);
4408        }
4409
4410        /* Print out the early_node_map[] */
4411        printk("early_node_map[%d] active PFN ranges\n", nr_nodemap_entries);
4412        for (i = 0; i < nr_nodemap_entries; i++)
4413            printk("    %3d: %0#10lx -> %0#10lx\n", early_node_map[i].nid,
4414                                early_node_map[i].start_pfn,
4415                                early_node_map[i].end_pfn);
4416
4417        /* Initialise every node */
4418        mminit_verify_pageflags_layout();
4419        setup_nr_node_ids();
4420        for_each_online_node(nid) {
4421            pg_data_t *pgdat = NODE_DATA(nid);//节点数据
4422            free_area_init_node(nid, NULL,
4423                        find_min_pfn_for_node(nid), NULL);
4424
4425            /* Any memory on that node */
4426            if (pgdat->node_present_pages)
4427                node_set_state(nid, N_HIGH_MEMORY);
4428            check_for_regular_memory(pgdat);
4429        }
4430 }
```

4422 行调用 free_area_init_node 来初始化一个节点。

4.1.9.3.3 free_area_init_node

//初始化指定节点的相关数据结构

```
void __paginginit free_area_init_node(int nid, unsigned long *zones_size,
```

```
3933            unsigned long node_start_pfn, unsigned long *zholes_size)
3934 {
3935        pg_data_t *pgdat = NODE_DATA(nid);
3936
3937        pgdat->node_id = nid;
3938        pgdat->node_start_pfn = node_start_pfn;
3939        calculate_node_totalpages(pgdat, zones_size, zholes_size);
3940
3941        alloc_node_mem_map(pgdat);
3942 #ifdef CONFIG_FLAT_NODE_MEM_MAP
3943        printk(KERN_DEBUG "free_area_init_node: node %d, pgdat %08lx, node_mem_map %08lx\n",
3944                nid, (unsigned long)pgdat,
3945                (unsigned long)pgdat->node_mem_map);
3946 #endif
3947
3948        free_area_init_core(pgdat, zones_size, zholes_size);
3949 }
```

3937-3938 行：初始话节点的相关变量。

3939 行：计算节点中所有内存区总的节点数，空洞数。

3941 行：为节点分配表示页框的数据结构 page 所需要的连续内存。

3949 行：调用 free_area_init_core 初始化节点。

```
//计算节点中总的页面数以及空洞数
static void __meminit calculate_node_totalpages(struct pglist_data *pgdat,
            unsigned long *zones_size, unsigned long *zholes_size)
{
        unsigned long realtotalpages, totalpages = 0;
        enum zone_type i;

        for (i = 0; i < MAX_NR_ZONES; i++)
            totalpages += zone_spanned_pages_in_node(pgdat->node_id, i,
```

```
3707                                     zones_size);
3708        pgdat->node_spanned_pages = totalpages;
3709
3710        realtotalpages = totalpages;
3711        for (i = 0; i < MAX_NR_ZONES; i++)
3712                realtotalpages -=
3713                        zone_absent_pages_in_node(pgdat->node_id, i,
3714                                zholes_size);
3715        pgdat->node_present_pages = realtotalpages;
3716        printk(KERN_DEBUG "On node %d totalpages: %lu\n", pgdat->node_id,
3717                                        realtotalpages);
3718 }
```

3705-3708 行：计算制定节点中所有区域的中的页面数，并赋值给节点的 node_spanned_pages 变量。

3711-3714 行：计算节点中各个区域的空洞页数，用 totalpages 减去空洞页数，得到实际的页面数，赋值给节点的 node_present_pages 变量。

```
/*
 * Return the number of pages a zone spans in a node, including holes
 * present_pages = zone_spanned_pages_in_node() - zone_absent_pages_in_node()
 */
static unsigned long __meminit zone_spanned_pages_in_node(int nid,
                                unsigned long zone_type,
                                unsigned long *ignored)
{
        unsigned long node_start_pfn, node_end_pfn;
        unsigned long zone_start_pfn, zone_end_pfn;

        /* Get the start and end of the node and zone */
        get_pfn_range_for_nid(nid, &node_start_pfn, &node_end_pfn);
        zone_start_pfn = arch_zone_lowest_possible_pfn[zone_type];
```

```
3579        zone_end_pfn = arch_zone_highest_possible_pfn[zone_type];
3580        adjust_zone_range_for_zone_movable(nid, zone_type,
3581                        node_start_pfn, node_end_pfn,
3582                        &zone_start_pfn, &zone_end_pfn);
3583
3584        /* Check that this node has pages within the zone's required range */
3585        if (zone_end_pfn < node_start_pfn || zone_start_pfn > node_end_pfn)
3586            return 0;
3587
3588        /* Move the zone boundaries inside the node if necessary */
3589        zone_end_pfn = min(zone_end_pfn, node_end_pfn);
3590        zone_start_pfn = max(zone_start_pfn, node_start_pfn);
3591
3592        /* Return the spanned pages */
3593        return zone_end_pfn - zone_start_pfn;
3594 }
```

该函数通过节点的起始页面和终止页面与内存区域的起始页面和终止页面，计算内存区域包含的页面数。

计算的方法见 3589-3590 行：用内存区域与节点终止页面较小的一个减去 内存区域与节点起始页面较大的一个 得到内存区域总的页面数。

```
//计算一个节点中 hole 的数目
/* Return the number of page frames in holes in a zone on a node */
static unsigned long __meminit zone_absent_pages_in_node(int nid,
                        unsigned long zone_type,
                        unsigned long *ignored)
{
    unsigned long node_start_pfn, node_end_pfn;
    unsigned long zone_start_pfn, zone_end_pfn;

    get_pfn_range_for_nid(nid, &node_start_pfn, &node_end_pfn);
    zone_start_pfn = max(arch_zone_lowest_possible_pfn[zone_type],
```

```
3669                                    node_start_pfn);
3670        zone_end_pfn = min(arch_zone_highest_possible_pfn[zone_type],
3671                                    node_end_pfn);
3672
3673        adjust_zone_range_for_zone_movable(nid, zone_type,
3674                    node_start_pfn, node_end_pfn,
3675                    &zone_start_pfn, &zone_end_pfn);
3676        return __absent_pages_in_range(nid, zone_start_pfn, zone_end_pfn);
3677 }
```

3668-3671 行：得到区域起始和终止页面号。

3676:行：调用__absent_pages_in_range 计算起始和终止页面中是空洞的页面数目。

```
/*
 * Return the number of holes in a range on a node. If nid is MAX_NUMNODES,
 * then all holes in the requested range will be accounted for.
 */
unsigned long __meminit __absent_pages_in_range(int nid,
                        unsigned long range_start_pfn,
                        unsigned long range_end_pfn)
{
        int i = 0;
        unsigned long prev_end_pfn = 0, hole_pages = 0;
        unsigned long start_pfn;

        /* Find the end_pfn of the first active range of pfns in the node */
        i = first_active_region_index_in_nid(nid);
        if (i == -1)
                return 0;

        prev_end_pfn = min(early_node_map[i].start_pfn, range_end_pfn);

        /* Account for ranges before physical memory on this node */
```

```
	if (early_node_map[i].start_pfn > range_start_pfn)
		hole_pages = prev_end_pfn - range_start_pfn;

	/* Find all holes for the zone within the node */
	for (; i != -1; i = next_active_region_index_in_nid(i, nid)) {

		/* No need to continue if prev_end_pfn is outside the zone */
		if (prev_end_pfn >= range_end_pfn)
			break;

		/* Make sure the end of the zone is not within the hole */
		start_pfn = min(early_node_map[i].start_pfn, range_end_pfn);
		prev_end_pfn = max(prev_end_pfn, range_start_pfn);

		/* Update the hole size cound and move on */
		if (start_pfn > range_start_pfn) {
			BUG_ON(prev_end_pfn > start_pfn);
			hole_pages += start_pfn - prev_end_pfn;
		}
		prev_end_pfn = early_node_map[i].end_pfn;
	}

	/* Account for ranges past physical memory on this node */
	if (range_end_pfn > prev_end_pfn)
		hole_pages += range_end_pfn -
				max(range_start_pfn, prev_end_pfn);

	return hole_pages;
}
```

该函数通过计算[range_start_pfn, range_end_pfn]中包含的early_node_map内存区数目，如果这些内存区中存在间隙，则[range_start_pfn, range_end_pfn]存在空洞，并且可以计算空洞的数目。

//为节点中的叶框分配 page 数据结构所需的 连续内存。

```
static void __init_refok alloc_node_mem_map(struct pglist_data *pgdat)
{
	/* Skip empty nodes */
	if (!pgdat->node_spanned_pages)
		return;

#ifdef CONFIG_FLAT_NODE_MEM_MAP
	/* ia64 gets its own node_mem_map, before this, without bootmem */
	if (!pgdat->node_mem_map) {
		unsigned long size, start, end;
		struct page *map;

		/*
		 * The zone's endpoints aren't required to be MAX_ORDER
		 * aligned but the node_mem_map endpoints must be in order
		 * for the buddy allocator to function correctly.
		 */
		start = pgdat->node_start_pfn & ~(MAX_ORDER_NR_PAGES - 1);
		end = pgdat->node_start_pfn + pgdat->node_spanned_pages;
		end = ALIGN(end, MAX_ORDER_NR_PAGES);
		size =  (end - start) * sizeof(struct page);
		map = alloc_remap(pgdat->node_id, size);
		if (!map)
			map = alloc_bootmem_node(pgdat, size);
		pgdat->node_mem_map = map + (pgdat->node_start_pfn - start);
	}
#ifndef CONFIG_NEED_MULTIPLE_NODES
```

```
	/*
	 * With no DISCONTIG, the global mem_map is just set as node 0's
	 */
	if (pgdat == NODE_DATA(0)) {
		mem_map = NODE_DATA(0)->node_mem_map;
#ifdef CONFIG_ARCH_POPULATES_NODE_MAP
		if (page_to_pfn(mem_map) != pgdat->node_start_pfn)
			mem_map -= (pgdat->node_start_pfn - ARCH_PFN_OFFSET);
#endif /* CONFIG_ARCH_POPULATES_NODE_MAP */
	}
#endif
#endif /* CONFIG_FLAT_NODE_MEM_MAP */
}
```

该函数通过节点中页面数（node_spanned_pages，包含空洞页面）计算所需的 page 的数目，然后分配一块连续的物理内存来存放这些 page，并将 page 的起始地址保存在 pgdat->node_mem_map 中。

4.1.9.3.4 free_area_init_core

该函数初始化节点节点中的每一个内存区域,初始化的内容包括：

1、计算内存区域中包含的页面数，空洞页面数
2、设置内存区域中所有页面为保留
3、设置伙伴系统中所有的队列为空
4、清空内存位图

```
/*
 * Set up the zone data structures:
 *   - mark all pages reserved
 *   - mark all memory queues empty
 *   - clear the memory bitmaps
 */
static void __paginginit free_area_init_core(struct pglist_data *pgdat,
		unsigned long *zones_size, unsigned long *zholes_size)
```

```
{
	enum zone_type j;
	int nid = pgdat->node_id;
	unsigned long zone_start_pfn = pgdat->node_start_pfn;
	int ret;

	pgdat_resize_init(pgdat);
	pgdat->nr_zones = 0;
	init_waitqueue_head(&pgdat->kswapd_wait);
	pgdat->kswapd_max_order = 0;
	pgdat_page_cgroup_init(pgdat);
	//初始化节点这中的每个区
	for (j = 0; j < MAX_NR_ZONES; j++) {
		struct zone *zone = pgdat->node_zones + j;
		unsigned long size, realsize, memmap_pages;
		enum lru_list l;

		size = zone_spanned_pages_in_node(nid, j, zones_size);
		realsize = size - zone_absent_pages_in_node(nid, j,
								zholes_size);

		/*
		 * Adjust realsize so that it accounts for how much memory
		 * is used by this zone for memmap. This affects the watermark
		 * and per-cpu initialisations
		 */
		memmap_pages =
			PAGE_ALIGN(size * sizeof(struct page)) >> PAGE_SHIFT;
		if (realsize >= memmap_pages) {
			realsize -= memmap_pages;
			if (memmap_pages)
```

```
3832                    printk(KERN_DEBUG
3833                            "  %s zone: %lu pages used for memmap\n",
3834                            zone_names[j], memmap_pages);
3835            } else
3836                printk(KERN_WARNING
3837                    "  %s zone: %lu pages exceeds realsize %lu\n",
3838                    zone_names[j], memmap_pages, realsize);
3839
3840            /* Account for reserved pages */
3841            if (j == 0 && realsize > dma_reserve) {
3842                realsize -= dma_reserve;
3843                printk(KERN_DEBUG "  %s zone: %lu pages reserved\n",
3844                        zone_names[0], dma_reserve);
3845            }
3846
3847            if (!is_highmem_idx(j))
3848                nr_kernel_pages += realsize;
3849            nr_all_pages += realsize;
3850
3851            zone->spanned_pages = size;
3852            zone->present_pages = realsize;
```

3818-3852 行：计算内存区域中页面数目，空洞页面数目。

```
3853 #ifdef CONFIG_NUMA
3854            zone->node = nid;
3855            zone->min_unmapped_pages = (realsize*sysctl_min_unmapped_ratio)
3856                            / 100;
3857            zone->min_slab_pages = (realsize * sysctl_min_slab_ratio) / 100;
3858 #endif
3859            zone->name = zone_names[j];
3860            spin_lock_init(&zone->lock);
3861            spin_lock_init(&zone->lru_lock);
```

```
3862            zone_seqlock_init(zone);
3863            zone->zone_pgdat = pgdat;
3864
3865            zone->prev_priority = DEF_PRIORITY;
3866
3867            zone_pcp_init(zone);
3868            for_each_lru(l) {
3869                    INIT_LIST_HEAD(&zone->lru[l].list);
3870                    zone->reclaim_stat.nr_saved_scan[l] = 0;
3871            }
3872            zone->reclaim_stat.recent_rotated[0] = 0;
3873            zone->reclaim_stat.recent_rotated[1] = 0;
3874            zone->reclaim_stat.recent_scanned[0] = 0;
3875            zone->reclaim_stat.recent_scanned[1] = 0;
3876            zap_zone_vm_stats(zone);
3877            zone->flags = 0;
3878            if (!size)
3879                    continue;
3880
3881            set_pageblock_order(pageblock_default_order());
```

3853-3881 行：初始化内存区域 zone 中的各个变量。

```
3882            setup_usemap(pgdat, zone, size);
```

3882 行：清空内存位图

```
3883            ret = init_currently_empty_zone(zone, zone_start_pfn,
3884                                            size, MEMMAP_EARLY);
3885            BUG_ON(ret);
3886            memmap_init(size, nid, j, zone_start_pfn);
3887            zone_start_pfn += size;
3888        }
3889 }
```

3883 行：初始化伙伴系统

3886 行：设置页面为保留

```
//初始化和伙伴系统有关的数据结构
3260 __meminit int init_currently_empty_zone(struct zone *zone,
3261                                 unsigned long zone_start_pfn,
3262                                 unsigned long size,
3263                                 enum memmap_context context)
3264 {
3265         struct pglist_data *pgdat = zone->zone_pgdat;
3266         int ret;
3267         ret = zone_wait_table_init(zone, size);
3268         if (ret)
3269                 return ret;
3270         pgdat->nr_zones = zone_idx(zone) + 1;
3271
3272         zone->zone_start_pfn = zone_start_pfn;
3273
3274         mminit_dprintk(MMINIT_TRACE, "memmap_init",
3275                         "Initialising map node %d zone %lu pfns %lu -> %lu\n",
3276                         pgdat->node_id,
3277                         (unsigned long)zone_idx(zone),
3278                         zone_start_pfn, (zone_start_pfn + size));
3279
3280         zone_init_free_lists(zone);
3281
3282         return 0;
3283 }
```

3280 行：调用 zone_init_free_lists 函数初始化伙伴系统数据结构 free_list。

```
3053 static void __meminit zone_init_free_lists(struct zone *zone)
3054 {
```

```
	int order, t;
	for_each_migratetype_order(order, t) {
		INIT_LIST_HEAD(&zone->free_area[order].free_list[t]);
		zone->free_area[order].nr_free = 0;
	}
}
```

```
#ifndef __HAVE_ARCH_MEMMAP_INIT
#define memmap_init(size, nid, zone, start_pfn) \
	memmap_init_zone((size), (nid), (zone), (start_pfn), MEMMAP_EARLY)
#endif
```

```
/*
 * Initially all pages are reserved - free ones are freed
 * up by free_all_bootmem() once the early boot process is
 * done. Non-atomic initialization, single-pass.
 */
void __meminit memmap_init_zone(unsigned long size, int nid, unsigned long zone,
		unsigned long start_pfn, enum memmap_context context)
{
	struct page *page;
	unsigned long end_pfn = start_pfn + size;
	unsigned long pfn;
	struct zone *z;

	if (highest_memmap_pfn < end_pfn - 1)
		highest_memmap_pfn = end_pfn - 1;

	z = &NODE_DATA(nid)->node_zones[zone];
	for (pfn = start_pfn; pfn < end_pfn; pfn++) {
		/*
```

```
         * There can be holes in boot-time mem_map[]s
         * handed to this function.  They do not
         * exist on hotplugged memory.
         */
        if (context == MEMMAP_EARLY) {
                if (!early_pfn_valid(pfn))
                        continue;
                if (!early_pfn_in_nid(pfn, nid))
                        continue;
        }
        page = pfn_to_page(pfn);
        set_page_links(page, zone, nid, pfn);
        mminit_verify_page_links(page, zone, nid, pfn);
        init_page_count(page);
        reset_page_mapcount(page);
        SetPageReserved(page);
```

设置页面为保留。

```
        /*
         * Mark the block movable so that blocks are reserved for
         * movable at startup. This will force kernel allocations
         * to reserve their blocks rather than leaking throughout
         * the address space during boot when many long-lived
         * kernel allocations are made. Later some blocks near
         * the start are marked MIGRATE_RESERVE by
         * setup_zone_migrate_reserve()
         *
         * bitmap is created for zone's valid pfn range. but memmap
         * can be created for invalid pages (for alignment)
         * check here not to call set_pageblock_migratetype() against
         * pfn out of zone.
         */
```

```
        if ((z->zone_start_pfn <= pfn)
            && (pfn < z->zone_start_pfn + z->spanned_pages)
            && !(pfn & (pageblock_nr_pages - 1)))
            set_pageblock_migratetype(page, MIGRATE_MOVABLE);

        INIT_LIST_HEAD(&page->lru);
#ifdef WANT_PAGE_VIRTUAL
        /* The shift won't overflow because ZONE_NORMAL is below 4G. */
        if (!is_highmem_idx(zone))
            set_page_address(page, __va(pfn << PAGE_SHIFT));
#endif
    }
}
```

回到 start_kernel 中，调用 build_all_zonelists 建立节点备用列表。

5. 节点备用列表初始化

5.1 build_all_zonelists

初始化节点备用分配内存区域分配顺序

```
void build_all_zonelists(void)
{
    set_zonelist_order();

    if (system_state == SYSTEM_BOOTING) {
        __build_all_zonelists(NULL);
        mminit_verify_zonelist();
        cpuset_init_current_mems_allowed();
    } else {
        /* we have to stop all cpus to guarantee there is no user
```

```
                    of zonelist */
                stop_machine(__build_all_zonelists, NULL, NULL);
                /* cpuset refresh routine should be here */
        }
        vm_total_pages = nr_free_pagecache_pages();
        /*
         * Disable grouping by mobility if the number of pages in the
         * system is too low to allow the mechanism to work. It would be
         * more accurate, but expensive to check per-zone. This check is
         * made on memory-hotadd so a system can start with mobility
         * disabled and enable it later
         */
        if (vm_total_pages < (pageblock_nr_pages * MIGRATE_TYPES))
                page_group_by_mobility_disabled = 1;
        else
                page_group_by_mobility_disabled = 0;

        printk("Built %i zonelists in %s order, mobility grouping %s.    "
                "Total pages: %ld\n",
                        nr_online_nodes,
                        zonelist_order_name[current_zonelist_order],
                        page_group_by_mobility_disabled ? "off" : "on",
                        vm_total_pages);
#ifdef CONFIG_NUMA
        printk("Policy zone: %s\n", zone_names[policy_zone]);
#endif
}
```

该函数调用__build_all_zonelists 来完成大部分的工作。

5.2 __build_all_zonelists

```
/* return values int ....just for stop_machine() */
static int __build_all_zonelists(void *dummy)
{
        int nid;
        int cpu;

#ifdef CONFIG_NUMA
        memset(node_load, 0, sizeof(node_load));
#endif
        for_each_online_node(nid) {
                pg_data_t *pgdat = NODE_DATA(nid);

                build_zonelists(pgdat);
                build_zonelist_cache(pgdat);
        }

        /*
         * Initialize the boot_pagesets that are going to be used
         * for bootstrapping processors. The real pagesets for
         * each zone will be allocated later when the per cpu
         * allocator is available.
         *
         * boot_pagesets are used also for bootstrapping offline
         * cpus if the system is already booted because the pagesets
         * are needed to initialize allocators on a specific cpu too.
         * F.e. the percpu allocator needs the page allocator which
         * needs the percpu allocator in order to allocate its pagesets
         * (a chicken-egg dilemma).
         */
```

```
2809        for_each_possible_cpu(cpu)
2810            setup_pageset(&per_cpu(boot_pageset, cpu), 0);
2811
2812        return 0;
2813 }
```

2789-2794 行：对每个节点，调用 build_zonelists 函数建立备用列表。

5.3 build_zonelists

```
static void build_zonelists(pg_data_t *pgdat)
{
    int j, node, load;
    enum zone_type i;
    nodemask_t used_mask;
    int local_node, prev_node;
    struct zonelist *zonelist;
    int order = current_zonelist_order;

    /* initialize zonelists */
    for (i = 0; i < MAX_ZONELISTS; i++) {
        zonelist = pgdat->node_zonelists + i;
        zonelist->_zonerefs[0].zone = NULL;
        zonelist->_zonerefs[0].zone_idx = 0;
    }

    /* NUMA-aware ordering of nodes */
    local_node = pgdat->node_id;
    load = nr_online_nodes;
    prev_node = local_node;
    nodes_clear(used_mask);

```

```
	memset(node_order, 0, sizeof(node_order));
	j = 0;

	while ((node = find_next_best_node(local_node, &used_mask)) >= 0) {
		int distance = node_distance(local_node, node);

		/*
		 * If another node is sufficiently far away then it is better
		 * to reclaim pages in a zone before going off node.
		 */
		if (distance > RECLAIM_DISTANCE)
			zone_reclaim_mode = 1;

		/*
		 * We don't want to pressure a particular node.
		 * So adding penalty to the first node in same
		 * distance group to make it round-robin.
		 */
		if (distance != node_distance(local_node, prev_node))
			node_load[node] = load;

		prev_node = node;
		load--;
		if (order == ZONELIST_ORDER_NODE)
			build_zonelists_in_node_order(pgdat, node);
		else
			node_order[j++] = node; /* remember order */
	}

	if (order == ZONELIST_ORDER_ZONE) {
		/* calculate node order -- i.e., DMA last! */
```

```
                build_zonelists_in_zone_order(pgdat, j);
        }

        build_thisnode_zonelists(pgdat);
}
```

5.4 build_zonelists_in_node_order

```
/*
 * Build zonelists ordered by node and zones within node.
 * This results in maximum locality--normal zone overflows into local
 * DMA zone, if any--but risks exhausting DMA zone.
 */
static void build_zonelists_in_node_order(pg_data_t *pgdat, int node)
{
        int j;
        struct zonelist *zonelist;

        zonelist = &pgdat->node_zonelists[0];
        for (j = 0; zonelist->_zonerefs[j].zone != NULL; j++)
                ;
        j = build_zonelists_node(NODE_DATA(node), zonelist, j,
                                                        MAX_NR_ZONES - 1);
        zonelist->_zonerefs[j].zone = NULL;
        zonelist->_zonerefs[j].zone_idx = 0;
}
```

6. 利用early_res分配内存

在 slab 分配器还不存在的时候，只有前面刚刚建立好的初始化阶段的内存管理体系，如何分配一个内存空间来存放 pid 散列表。而 pid 散列表这么一个东西，它跟物理内存大小有关，也就是 1GB 的内存空间就可能有 16~4096 个散列项。不管怎样，调用 alloc_large_system_hash。

```
void __init pidhash_init(void)
{
        int i, pidhash_size;

        pid_hash = alloc_large_system_hash("PID", sizeof(*pid_hash), 0, 18,
                                           HASH_EARLY | HASH_SMALL,
                                           &pidhash_shift, NULL, 4096);
        pidhash_size = 1 << pidhash_shift;

        for (i = 0; i < pidhash_size; i++)
                INIT_HLIST_HEAD(&pid_hash[i]);
}
```

在这里重点分析 alloc_large_system_hash 函数。

```
/*
 * allocate a large system hash table from bootmem
 * - it is assumed that the hash table must contain an exact power-of-2
 *   quantity of entries
 * - limit is the number of hash buckets, not the total allocation size
 */
void *__init alloc_large_system_hash(const char *tablename,
                                     unsigned long bucketsize,
                                     unsigned long numentries,
                                     int scale,
                                     int flags,
```

```
					unsigned int *_hash_shift,
					unsigned int *_hash_mask,
					unsigned long limit)
{
	unsigned long long max = limit;
	unsigned long log2qty, size;
	void *table = NULL;

	/* allow the kernel cmdline to have a say */
	if (!numentries) {
		/* round applicable memory size up to nearest megabyte */
		numentries = nr_kernel_pages;
		numentries += (1UL << (20 - PAGE_SHIFT)) - 1;
		numentries >>= 20 - PAGE_SHIFT;
		numentries <<= 20 - PAGE_SHIFT;

		/* limit to 1 bucket per 2^scale bytes of low memory */
		if (scale > PAGE_SHIFT)
			numentries >>= (scale - PAGE_SHIFT);
		else
			numentries <<= (PAGE_SHIFT - scale);

		/* Make sure we've got at least a 0-order allocation.. */
		if (unlikely(flags & HASH_SMALL)) {
			/* Makes no sense without HASH_EARLY */
			WARN_ON(!(flags & HASH_EARLY));
			if (!(numentries >> *_hash_shift)) {
				numentries = 1UL << *_hash_shift;
				BUG_ON(!numentries);
			}
		} else if (unlikely((numentries * bucketsize) < PAGE_SIZE))
```

```
		numentries = PAGE_SIZE / bucketsize;
	}
	numentries = roundup_pow_of_two(numentries);

	/* limit allocation size to 1/16 total memory by default */
	if (max == 0) {
		max = ((unsigned long long)nr_all_pages << PAGE_SHIFT) >> 4;
		do_div(max, bucketsize);
	}

	if (numentries > max)
		numentries = max;

	log2qty = ilog2(numentries);

	do {
		size = bucketsize << log2qty;
		if (flags & HASH_EARLY)
			table = alloc_bootmem_nopanic(size);
		else if (hashdist)
			table = __vmalloc(size, GFP_ATOMIC, PAGE_KERNEL);
		else {
			/*
			 * If bucketsize is not a power-of-two, we may free
			 * some pages at the end of hash table which
			 * alloc_pages_exact() automatically does
			 */
			if (get_order(size) < MAX_ORDER) {
				table = alloc_pages_exact(size, GFP_ATOMIC);
				kmemleak_alloc(table, size, 1, GFP_ATOMIC);
			}
```

```
		}
	} while (!table && size > PAGE_SIZE && --log2qty);

	if (!table)
		panic("Failed to allocate %s hash table\n", tablename);

	printk(KERN_INFO "%s hash table entries: %d (order: %d, %lu bytes)\n",
	       tablename,
	       (1U << log2qty),
	       ilog2(size) - PAGE_SHIFT,
	       size);

	if (_hash_shift)
		*_hash_shift = log2qty;
	if (_hash_mask)
		*_hash_mask = (1 << log2qty) - 1;

	return table;
}
```

该函数具体内容不细看了，主要查看分配内存的函数。alloc_bootmem_nopanic函数最终会调用___alloc_bootmem_nopanic函数分配内存。

```
static void * __init ___alloc_bootmem_nopanic(unsigned long size,
					unsigned long align,
					unsigned long goal,
					unsigned long limit)
{
#ifdef CONFIG_NO_BOOTMEM
	void *ptr;

	if (WARN_ON_ONCE(slab_is_available()))
		return kzalloc(size, GFP_NOWAIT);
```

```

restart:

	ptr = __alloc_memory_core_early(MAX_NUMNODES, size, align, goal, limit);
				¢] 990L, 24400C
	if (ptr)
		return ptr;

	if (goal != 0) {
		goal = 0;
		goto restart;
	}

	return NULL;
#else
	bootmem_data_t *bdata;
	void *region;

restart:
	region = alloc_arch_preferred_bootmem(NULL, size, align, goal, limit);
	if (region)
		return region;

	list_for_each_entry(bdata, &bdata_list, list) {
		if (goal && bdata->node_low_pfn <= PFN_DOWN(goal))
			continue;
		if (limit && bdata->node_min_pfn >= PFN_DOWN(limit))
			break;

		region = alloc_bootmem_core(bdata, size, align, goal, limit);
```

```
		if (region)
			return region;
	}

	if (goal) {
		goal = 0;
		goto restart;
	}

	return NULL;
#endif
}

```

6.1 __alloc_memory_core_early

```
#ifdef CONFIG_NO_BOOTMEM
void * __init __alloc_memory_core_early(int nid, u64 size, u64 align,
					u64 goal, u64 limit)
{
	int i;
	void *ptr;

	/* need to go over early_node_map to find out good range for node */
	for_each_active_range_index_in_nid(i, nid) {
		u64 addr;
		u64 ei_start, ei_last;

		ei_last = early_node_map[i].end_pfn;
		ei_last <<= PAGE_SHIFT;
		ei_start = early_node_map[i].start_pfn;
```

```
                ei_start <<= PAGE_SHIFT;
                addr = find_early_area(ei_start, ei_last,
                                        goal, limit, size, align);

                if (addr == -1ULL)
                        continue;

#if 0
                printk(KERN_DEBUG "alloc (nid=%d %llx - %llx) (%llx - %llx) %llx %llx => %llx\n",
                                nid,
                                ei_start, ei_last, goal, limit, size,
                                align, addr);
#endif

                ptr = phys_to_virt(addr);
                memset(ptr, 0, size);
                reserve_early_without_check(addr, addr + size, "BOOTMEM");
                return ptr;
        }

        return NULL;
}
#endif
```

该函数调用 find_early_area 来分配函数，find_early_area 在之前的章节中已经分析过。

至此，针对初始化期间的内存管理就全介绍完了，不过我还有一点要说。初始化期间为各个数据结构分配物理页面都是通过_alloc_memory_core_early 函数。内核发展到现在，已经抛弃了以前的 BOOTMEM 分配体系， 而只是通过

_alloc_memory_core_early 函数调用 find_early_area 函数进行简单的分配，大幅提高了系统初始化期间的性能。

通过_alloc_memory_core_early 分配的页面是会释放的。只要当伙伴系统建立起来后，我们通过 alloc_page 函数分配的页面，其对应的释放函数是 free_page。而_alloc_memory_core_early 函数也有对应的 free 函数的，那就是 free_all_memory_core_early 函数，用于释放初始化期间占有的临时内存，后面一定会碰到。

7. 虚拟文件系统early初始化

使用 early_res 分配虚拟文件系统相关数据结构

```
fs/dcache.c
void __init vfs_caches_init_early(void)
{
	dcache_init_early();
	inode_init_early();
}
```

```
static void __init dcache_init_early(void)
{
	int loop;

	/* If hashes are distributed across NUMA nodes, defer
	 * hash allocation until vmalloc space is available.
	 */
	if (hashdist)
		return;

	dentry_hashtable =
		alloc_large_system_hash("Dentry cache",
					sizeof(struct hlist_head),
```

```
					dhash_entries,
					13,
					HASH_EARLY,
					&d_hash_shift,
					&d_hash_mask,
					0);

	for (loop = 0; loop < (1 << d_hash_shift); loop++)
		INIT_HLIST_HEAD(&dentry_hashtable[loop]);
}
```

```
/*
 * Initialize the waitqueues and inode hash table.
 */
void __init inode_init_early(void)
{
	int loop;

	/* If hashes are distributed across NUMA nodes, defer
	 * hash allocation until vmalloc space is available.
	 */
	if (hashdist)
		return;

	inode_hashtable =
		alloc_large_system_hash("Inode-cache",
					sizeof(struct hlist_head),
					ihash_entries,
					14,
					HASH_EARLY,
					&i_hash_shift,
```

```
					&i_hash_mask,
					0);

	for (loop = 0; loop < (1 << i_hash_shift); loop++)
		INIT_HLIST_HEAD(&inode_hashtable[loop]);
}
```

8. 中断向量表初始化

中断向量表初始化，该过程就不详细分析了，在分析中断时再讨论。

```
void __init trap_init(void)
{
	int i;

#ifdef CONFIG_EISA
	void __iomem *p = early_ioremap(0x0FFFD9, 4);

	if (readl(p) == 'E' + ('I'<<8) + ('S'<<16) + ('A'<<24))
		EISA_bus = 1;
	early_iounmap(p, 4);
#endif

	set_intr_gate(0, &divide_error);
	set_intr_gate_ist(1, &debug, DEBUG_STACK);
	set_intr_gate_ist(2, &nmi, NMI_STACK);
	/* int3 can be called from all */
	set_system_intr_gate_ist(3, &int3, DEBUG_STACK);
	/* int4 can be called from all */
	set_system_intr_gate(4, &overflow);
```

```
	set_intr_gate(5, &bounds);
	set_intr_gate(6, &invalid_op);
	set_intr_gate(7, &device_not_available);
#ifdef CONFIG_X86_32
	set_task_gate(8, GDT_ENTRY_DOUBLEFAULT_TSS);
#else
	set_intr_gate_ist(8, &double_fault, DOUBLEFAULT_STACK);
#endif
	set_intr_gate(9, &coprocessor_segment_overrun);
	set_intr_gate(10, &invalid_TSS);
	set_intr_gate(11, &segment_not_present);
	set_intr_gate_ist(12, &stack_segment, STACKFAULT_STACK);
	set_intr_gate(13, &general_protection);
	set_intr_gate(14, &page_fault);
	set_intr_gate(15, &spurious_interrupt_bug);
	set_intr_gate(16, &coprocessor_error);
	set_intr_gate(17, &alignment_check);
#ifdef CONFIG_X86_MCE
	set_intr_gate_ist(18, &machine_check, MCE_STACK);
#endif
	set_intr_gate(19, &simd_coprocessor_error);

	/* Reserve all the builtin and the syscall vector: */
	for (i = 0; i < FIRST_EXTERNAL_VECTOR; i++)
		set_bit(i, used_vectors);

#ifdef CONFIG_IA32_EMULATION
	set_system_intr_gate(IA32_SYSCALL_VECTOR, ia32_syscall);
	set_bit(IA32_SYSCALL_VECTOR, used_vectors);
#endif

```

```
#ifdef CONFIG_X86_32
	if (cpu_has_fxsr) {
		printk(KERN_INFO "Enabling fast FPU save and restore... ");
		set_in_cr4(X86_CR4_OSFXSR);
		printk("done.\n");
	}
	if (cpu_has_xmm) {
		printk(KERN_INFO
			"Enabling unmasked SIMD FPU exception support... ");
		set_in_cr4(X86_CR4_OSXMMEXCPT);
		printk("done.\n");
	}
	}
	set_system_trap_gate(SYSCALL_VECTOR, &system_call);
	set_bit(SYSCALL_VECTOR, used_vectors);
#endif

	/*
	 * Should be a barrier for any external CPU state:
	 */
	cpu_init();

	x86_init.irqs.trap_init();
}
```

9. 内存管理初始化

9.1 mm_init函数全貌

该函数始内核初始化中最后和内存管理相关的函数。

1、mem_init 初始化伙伴系统。
2、kemn_cache_init 初始化 slab 管理数据结构
3、vmalloc_init:初始化高端内存管理中的非连续内存分配结构

```
/*
 * Set up kernel memory allocators
 */
static void __init mm_init(void)
{
        /*
         * page_cgroup requires countinous pages as memmap
         * and it's bigger than MAX_ORDER unless SPARSEMEM.
         */
        page_cgroup_init_flatmem();
        mem_init();
        kmem_cache_init();
        pgtable_cache_init();
        vmalloc_init();
}
```

9.2 mem_init伙伴系统的建立

```
867 void __init mem_init(void)
868 {
869         int codesize, reservedpages, datasize, initsize;
870         int tmp;
871
872         pci_iommu_alloc();
873
874 #ifdef CONFIG_FLATMEM
875         BUG_ON(!mem_map);
876 #endif
877         /* this will put all low memory onto the freelists */
878         totalram_pages += free_all_bootmem();
```

878 行：释放低端内存到伙伴系统。

```
879
880         reservedpages = 0;
```

```
881        for (tmp = 0; tmp < max_low_pfn; tmp++)
882              /*
883               * Only count reserved RAM pages:
884               */
885              if (page_is_ram(tmp) && PageReserved(pfn_to_page(tmp)))
886                    reservedpages++;
```

881-886：计算低端内存中保留页面的数目

```
887
888        set_highmem_pages_init();
```

888 行：初始化高端内存到伙伴系统

```
889
890        codesize =   (unsigned long) &_etext - (unsigned long) &_text;
891        datasize =   (unsigned long) &_edata - (unsigned long) &_etext;
892        initsize =   (unsigned long) &__init_end - (unsigned long) &__init_begin;
893
894        printk(KERN_INFO "Memory: %luk/%luk available (%dk kernel code, "
895                    "%dk reserved, %dk data, %dk init, %ldk highmem)\n",
896              nr_free_pages() << (PAGE_SHIFT-10),
897              num_physpages << (PAGE_SHIFT-10),
898              codesize >> 10,
899              reservedpages << (PAGE_SHIFT-10),
900              datasize >> 10,
901              initsize >> 10,
902              totalhigh_pages << (PAGE_SHIFT-10));
903
904        printk(KERN_INFO "virtual kernel memory layout:\n"
905              "      fixmap   : 0x%08lx - 0x%08lx     (%4ld kB)\n"
906 #ifdef CONFIG_HIGHMEM
907              "      pkmap     : 0x%08lx - 0x%08lx     (%4ld kB)\n"
908 #endif
909              "      vmalloc : 0x%08lx - 0x%08lx     (%4ld MB)\n"
```

```
            "       lowmem   : 0x%08lx - 0x%08lx     (%4ld MB)\n"
            "          .init : 0x%08lx - 0x%08lx     (%4ld kB)\n"
            "          .data : 0x%08lx - 0x%08lx     (%4ld kB)\n"
            "          .text : 0x%08lx - 0x%08lx     (%4ld kB)\n",
            FIXADDR_START, FIXADDR_TOP,
            (FIXADDR_TOP - FIXADDR_START) >> 10,

#ifdef CONFIG_HIGHMEM
            PKMAP_BASE, PKMAP_BASE+LAST_PKMAP*PAGE_SIZE,
            (LAST_PKMAP*PAGE_SIZE) >> 10,
#endif

            VMALLOC_START, VMALLOC_END,
            (VMALLOC_END - VMALLOC_START) >> 20,

            (unsigned long)__va(0), (unsigned long)high_memory,
            ((unsigned long)high_memory - (unsigned long)__va(0)) >> 20,

            (unsigned long)&__init_begin, (unsigned long)&__init_end,
            ((unsigned long)&__init_end -
             (unsigned long)&__init_begin) >> 10,

            (unsigned long)&_etext, (unsigned long)&_edata,
            ((unsigned long)&_edata - (unsigned long)&_etext) >> 10,

            (unsigned long)&_text, (unsigned long)&_etext,
            ((unsigned long)&_etext - (unsigned long)&_text) >> 10);

    /*
     * Check boundaries twice: Some fundamental inconsistencies can
     * be detected at build time already.
```

```
941         */
942 #define __FIXADDR_TOP (-PAGE_SIZE)
943 #ifdef CONFIG_HIGHMEM
944         BUILD_BUG_ON(PKMAP_BASE + LAST_PKMAP*PAGE_SIZE    > FIXADDR_START);
945         BUILD_BUG_ON(VMALLOC_END                    > PKMAP_BASE);
946 #endif
947 #define high_memory (-128UL << 20)
948         BUILD_BUG_ON(VMALLOC_START                  >= VMALLOC_END);
949 #undef high_memory
950 #undef __FIXADDR_TOP
951
952 #ifdef CONFIG_HIGHMEM
953         BUG_ON(PKMAP_BASE + LAST_PKMAP*PAGE_SIZE       > FIXADDR_START);
954         BUG_ON(VMALLOC_END                         > PKMAP_BASE);
955 #endif
956         BUG_ON(VMALLOC_START                       >= VMALLOC_END);
957         BUG_ON((unsigned long)high_memory           > VMALLOC_START);
958
959         if (boot_cpu_data.wp_works_ok < 0)
960                 test_wp_bit();
```

889-960：输出内存布局相关信息

```
961
962         save_pg_dir();
```

962:备份页全局变量 swapper_pg_dir

```
963         zap_low_mappings(true);
```

963 行：

```
964 }
```

9.2.1 低端内存释放free_all_bootmem

释放空闲页面到伙伴系统分配器中

```
Mm/bootmem.c
/**
 * free_all_bootmem - release free pages to the buddy allocator
 *
 * Returns the number of pages actually released.
 */
unsigned long __init free_all_bootmem(void)
{
#ifdef CONFIG_NO_BOOTMEM
        /*
         * We need to use MAX_NUMNODES instead of NODE_DATA(0)->node_id
         *    because in some case like Node0 doesnt have RAM installed
         *    low ram will be on Node1
         * Use MAX_NUMNODES will make sure all ranges in early_node_map[]
         *    will be used instead of only Node0 related
         */
        return free_all_memory_core_early(MAX_NUMNODES);
#else
        unsigned long total_pages = 0;
        bootmem_data_t *bdata;

        list_for_each_entry(bdata, &bdata_list, list)
                total_pages += free_all_bootmem_core(bdata);

        return total_pages;
#endif
}
```

因为内核倾向不使用 BOOTMEM，所以在这分析 free_all_memory_core_early(MAX_NUMNODES)函数。

9.2.1.1 free_all_memory_core_early

该函数释放由 early_res 分配的内存。

```
200 unsigned long __init free_all_memory_core_early(int nodeid)
201 {
202         int i;
203         u64 start, end;
204         unsigned long count = 0;
205         struct range *range = NULL;
206         int nr_range;
207
208         nr_range = get_free_all_memory_range(&range, nodeid);
209
210         for (i = 0; i < nr_range; i++) {
211                 start = range[i].start;
212                 end = range[i].end;
213                 count += end - start;
214                 __free_pages_memory(start, end);
215         }
216
217         return count;
218 }
```

208 行：获得空闲 page 的区域。

210-215 行：根据 208 行获得的 page 区域，释放 page 到伙伴系统分配器。

9.2.1.2 get_free_all_memory_range

```
int __init get_free_all_memory_range(struct range **rangep, int nodeid)
{
        int i, count;
```

```
	u64 start = 0, end;
	u64 size;
	u64 mem;
	struct range *range;
	int nr_range;

	count   = 0;
	for (i = 0; i < max_early_res && early_res[i].end; i++)
		count++;

	count *= 2;

	size = sizeof(struct range) * count;
	end = get_max_mapped();//获得最大的物理页面号
#ifdef MAX_DMA32_PFN
	if (end > (MAX_DMA32_PFN << PAGE_SHIFT))
		start = MAX_DMA32_PFN << PAGE_SHIFT;
#endif
	mem = find_fw_memmap_area(start, end, size, sizeof(struct range));
```

从[start,end]分配一块连续的内存，存放 range 结构，用于存放空闲页面区间的数据结构。

Range 的类型如下：

```
struct range {
    u64    start;
    u64    end;
};
```

```
u64 __init find_fw_memmap_area(u64 start, u64 end, u64 size, u64 align)
{
	return find_e820_area(start, end, size, align);
}
```

```
415         if (mem == -1ULL)
416                 panic("can not find more space for range free");
417
418         range = __va(mem);
419         /* use early_node_map[] and early_res to get range array at first */
420         memset(range, 0, size);
421         nr_range = 0;
```

初始化 range 数组

```
422
423         /* need to go over early_node_map to find out good range for node */
424         nr_range = add_from_early_node_map(range, count, nr_range, nodeid);
425 #ifdef CONFIG_X86_32
426         subtract_range(range, count, max_low_pfn, -1ULL);
```

426 行：从 range 中去掉高端内存范围内的元素

```
427 #endif
428         subtract_early_res(range, count);
```

428 行：从 range 中去掉 early_res 中占用的页面区域

```
429         nr_range = clean_sort_range(range, count);
430
431         /* need to clear it ? */
432         if (nodeid == MAX_NUMNODES) {
433                 memset(&early_res[0], 0,
434                         sizeof(struct early_res) * max_early_res);
435                 early_res = NULL;
436                 max_early_res = 0;
437         }
```

431-437 行：清空 early_res 数组。

```
        *rangep = range;
        return nr_range;
}
```

9.2.1.3 add_from_early_node_map

```
3396 int __init add_from_early_node_map(struct range *range, int az,
3397                                     int nr_range, int nid)
3398 {
3399         int i;
3400         u64 start, end;
3401
3402         /* need to go over early_node_map to find out good range for node */
3403         for_each_active_range_index_in_nid(i, nid) {
3404                 start = early_node_map[i].start_pfn;
3405                 end = early_node_map[i].end_pfn;
3406                 nr_range = add_range(range, az, nr_range, start, end);
3407         }
3408         return nr_range;
3409 }
```

该函数将 early_node_map 中的页面添加到 range 数组中，不管页面是否保留（也就是非空闲的页面也会包含到 range 数组中。）

3404-3405 行：获得 early_node_map 索引 i 的起始和终止页面号

3406 行：将 start,end 添加到 range 数组中。

```
int add_range(struct range *range, int az, int nr_range, u64 start, u64 end)
{
        if (start >= end)
                return nr_range;//范围检查
```

```
    /* Out of slots: */
    if (nr_range >= az)
        return nr_range;//超出 range 数组的范围。
    range[nr_range].start = start;
    range[nr_range].end = end;
    nr_range++;
    return nr_range;
}
```

9.2.1.4 subtract_range

```
static void __init subtract_early_res(struct range *range, int az)
{
    int i, count;
    u64 final_start, final_end;
    int idx = 0;
    count   = 0;
    for (i = 0; i < max_early_res && early_res[i].end; i++)
        count++;
    /* need to skip first one ?*/
    if (early_res != early_res_x)
        idx = 1;
#define DEBUG_PRINT_EARLY_RES 1
```

```
375 #if DEBUG_PRINT_EARLY_RES
376         printk(KERN_INFO "Subtract (%d early reservations)\n", count);
377 #endif
378         for (i = idx; i < count; i++) {
379                 struct early_res *r = &early_res[i];
380 #if DEBUG_PRINT_EARLY_RES
381                 printk(KERN_INFO "    #%d [%010llx - %010llx] %15s\n", i,
382                         r->start, r->end, r->name);
383 #endif
384                 final_start = PFN_DOWN(r->start);
385                 final_end = PFN_UP(r->end);
386                 if (final_start >= final_end)
387                         continue;
388                 subtract_range(range, az, final_start, final_end);
```

378-379 行：去掉 early_res 中占用的 early_res。

```
389         }
```

```
64 void subtract_range(struct range *range, int az, u64 start, u64 end)
 65 {
 66         int i, j;
 67 
 68         if (start >= end)
 69                 return;
 70 
 71         for (j = 0; j < az; j++) {
 72                 if (!range[j].end)
 73                         continue;
 74 
 75                 if (start <= range[j].start && end >= range[j].end) {
 76                         range[j].start = 0;
 77                         range[j].end = 0;
```

```
78                  continue;
79              }
```

75-79 行：如果 range[i]落于[start,end]之间，则去掉。

```
80
81              if (start <= range[j].start && end < range[j].end &&
82                  range[j].start < end) {
83                  range[j].start = end;
84                  continue;
85              }
```

81-85 行：range[i]与[start,end]有交集，去掉交集部分

```
86
87
88              if (start > range[j].start && end >= range[j].end &&
89                  range[j].end > start) {
90                  range[j].end = start;
91                  continue;
92              }
```

88-92 行：range[i]与[start,end]有交集，去掉交集部分

```
93
94              if (start > range[j].start && end < range[j].end) {
95                  /* Find the new spare: */
96                  for (i = 0; i < az; i++) {
97                      if (range[i].end == 0)
98                          break;
99                  }
100                 if (i < az) {
101                     range[i].end = range[j].end;
102                     range[i].start = end;
103                 } else {
104                     printk(KERN_ERR "run of slot in ranges\n");
105                 }
```

```
106                 range[j].end = start;
107                 continue;
108             }
```

94-108 行：range[i]包含[start,end]，则拆分 range。

```
109         }
110 }
```

9.2.1.5 __free_pages_memory(start, end)

获得空闲的 range 后，在 free_all_memory_core_early 中调用__free_pages_memory 释放内存到伙伴系统。

```
174 static void __init __free_pages_memory(unsigned long start, unsigned long end)
175 {
176         int i;
177         unsigned long start_aligned, end_aligned;
178         int order = ilog2(BITS_PER_LONG);
179
180         start_aligned = (start + (BITS_PER_LONG - 1)) & ~(BITS_PER_LONG - 1);
181         end_aligned = end & ~(BITS_PER_LONG - 1);
182
183         if (end_aligned <= start_aligned) {
184                 for (i = start; i < end; i++)
185                         __free_pages_bootmem(pfn_to_page(i), 0);
186
187                 return;
188         }
189
190         for (i = start; i < start_aligned; i++)
191                 __free_pages_bootmem(pfn_to_page(i), 0);
```

```

	for (i = start_aligned; i < end_aligned; i += BITS_PER_LONG)
		__free_pages_bootmem(pfn_to_page(i), order);

	for (i = end_aligned; i < end; i++)
		__free_pages_bootmem(pfn_to_page(i), 0);
}
```

该函数调用__free_pages_bootmem 函数释放内存。

```
void __meminit __free_pages_bootmem(struct page *page, unsigned int order)
{
	if (order == 0) {
		__ClearPageReserved(page);
		set_page_count(page, 0);
		set_page_refcounted(page);
		__free_page(page);
	} else {
		int loop;

		prefetchw(page);
		for (loop = 0; loop < BITS_PER_LONG; loop++) {
			struct page *p = &page[loop];

			if (loop + 1 < BITS_PER_LONG)
				prefetchw(p + 1);
			__ClearPageReserved(p);
			set_page_count(p, 0);
		}

		set_page_refcounted(page);
		__free_pages(page, order);
```

```
}
```

该函数调用__free_page 释放内存。__free_page 始释放页面的最终接口，待会在分析该函数。

9.2.2 高端内存释放set_highmem_pages_init

arch/x86/mm/ highmem_32.c

```
void __init set_highmem_pages_init(void)
{
	struct zone *zone;
	int nid;

	for_each_zone(zone) {
		unsigned long zone_start_pfn, zone_end_pfn;

		if (!is_highmem(zone))
			continue;

		zone_start_pfn = zone->zone_start_pfn;
		zone_end_pfn = zone_start_pfn + zone->spanned_pages;

		nid = zone_to_nid(zone);
		printk(KERN_INFO "Initializing %s for node %d (%08lx:%08lx)\n",
				zone->name, nid, zone_start_pfn, zone_end_pfn);

		add_highpages_with_active_regions(nid, zone_start_pfn,
				zone_end_pfn);
	}
	totalram_pages += totalhigh_pages;
}
```

117-118 行：检查区域是否属于高端内存。

127 行：调用 add_highpages_with_active_regions 函数进一步处理。

```
void __init add_highpages_with_active_regions(int nid, unsigned long start_pfn,
                                      unsigned long end_pfn)
{
        struct add_highpages_data data;

        data.start_pfn = start_pfn;
        data.end_pfn = end_pfn;

        work_with_active_regions(nid, add_highpages_work_fn, &data);
}
```

add_highpages_with_active_regions 调用 work_with_active_regions 函数进一步处理，并传入函数 add_highpages_work_fn 作为参数。

```
451 void __init work_with_active_regions(int nid, work_fn_t work_fn, void *data)
3452 {
3453         int i;
3454         int ret;
3455 
3456         for_each_active_range_index_in_nid(i, nid) {
3457                 ret = work_fn(early_node_map[i].start_pfn,
3458                                         early_node_map[i].end_pfn, data);
3459                 if (ret)
3460                         break;
3461         }
3462 }
```

3457-3458 行：调用 add_highpages_work_fn 进一步处理。

```
static int __init add_highpages_work_fn(unsigned long start_pfn,
                                 unsigned long end_pfn, void *datax)
{
        int node_pfn;
        struct page *page;
        unsigned long final_start_pfn, final_end_pfn;
        struct add_highpages_data *data;

        data = (struct add_highpages_data *)datax;

        final_start_pfn = max(start_pfn, data->start_pfn);
        final_end_pfn = min(end_pfn, data->end_pfn);
        if (final_start_pfn >= final_end_pfn)
                return 0;

        for (node_pfn = final_start_pfn; node_pfn < final_end_pfn;
             node_pfn++) {
                if (!pfn_valid(node_pfn))
                        continue;
                page = pfn_to_page(node_pfn);
                add_one_highpage_init(page);
        }

        return 0;

}
```

调用 add_one_highpage_init 进一步处理。

```
static void __init add_one_highpage_init(struct page *page)
```

```
418 {
419         ClearPageReserved(page);
420         init_page_count(page);
421         __free_page(page);
422         totalhigh_pages++;
423 }
```

419 行：去掉页面保留标志

420 行：初始化引用计数

421 行：调用__free_page 释放页面到伙伴系统。

9.2.3 __free_pages释放页面

```
void __free_pages(struct page *page, unsigned int order)
{
        if (put_page_testzero(page)) {
                if (order == 0)
                        free_hot_cold_page(page, 0);
                else
                        __free_pages_ok(page, order);
        }
}
```

order=0，调用 free_hot_cold_page(page, 0);

9.2.3.1 free_hot_cold_page

```
 * Free a 0-order page
 * cold == 1 ? free a cold page : free a hot page
 */
void free_hot_cold_page(struct page *page, int cold)
{
        struct zone *zone = page_zone(page);
```

```
1105        struct per_cpu_pages *pcp;
1106        unsigned long flags;
1107        int migratetype;
1108        int wasMlocked = __TestClearPageMlocked(page);
1109
1110        trace_mm_page_free_direct(page, 0);
1111        kmemcheck_free_shadow(page, 0);
1112
1113        if (PageAnon(page))
1114            page->mapping = NULL;
1115        if (free_pages_check(page))
1116                return;
```

1115-1116 行：检测 page 是否可以被释放。

1、page mapcount 必须等于-1

2、page _count 等于 0

3、page 标志位不能含有如下的标志：

```
/*
 * Flags checked when a page is freed.  Pages being freed should not
have
 * these flags set.  It they are, there is a problem.
 */
#define PAGE_FLAGS_CHECK_AT_FREE \
    (1 << PG_lru      | 1 << PG_locked      | \
     1 << PG_private | 1 << PG_private_2 | \
     1 << PG_buddy      | 1 << PG_writeback | 1 << PG_reserved | \
     1 << PG_slab    | 1 << PG_swapcache | 1 << PG_active | \
     1 << PG_unevictable | __PG_MLOCKED | __PG_HWPOISON)
```

```
static inline int free_pages_check(struct page *page)
{
        if (unlikely(page_mapcount(page) |
```

```
        (page->mapping != NULL)   |
        (atomic_read(&page->_count) != 0) |
        (page->flags & PAGE_FLAGS_CHECK_AT_FREE))) {
        bad_page(page);
        return 1;
    }
    if (page->flags & PAGE_FLAGS_CHECK_AT_PREP)
        page->flags &= ~PAGE_FLAGS_CHECK_AT_PREP;
    return 0;
}
```

```

    if (!PageHighMem(page)) {
        debug_check_no_locks_freed(page_address(page), PAGE_SIZE);
        debug_check_no_obj_freed(page_address(page), PAGE_SIZE);
    }
    arch_free_page(page, 0);
    kernel_map_pages(page, 1, 0);

    migratetype = get_pageblock_migratetype(page);
    set_page_private(page, migratetype);
    local_irq_save(flags);
    if (unlikely(wasMlocked))
        free_page_mlock(page);
    __count_vm_event(PGFREE);

    /*
     * We only track unmovable, reclaimable and movable on pcp lists.
     * Free ISOLATE pages back to the allocator because they are being
     * offlined but treat RESERVE as movable pages so we can get those
     * areas back if necessary. Otherwise, we may have to free
```

```
	 * excessively into the page allocator
	 */
	if (migratetype >= MIGRATE_PCPTYPES) {
		if (unlikely(migratetype == MIGRATE_ISOLATE)) {
			free_one_page(zone, page, 0, migratetype);
			goto out;
		}
		migratetype = MIGRATE_MOVABLE;
	}

	pcp = &this_cpu_ptr(zone->pageset)->pcp;
	if (cold)
		list_add_tail(&page->lru, &pcp->lists[migratetype]);
	else
		list_add(&page->lru, &pcp->lists[migratetype]);
	pcp->count++;
	if (pcp->count >= pcp->high) {
		free_pcppages_bulk(zone, pcp->batch, pcp);
		pcp->count -= pcp->batch;
	}

out:
	local_irq_restore(flags);
}
```

9.2.3.2 free_one_page

```
static void free_one_page(struct zone *zone, struct page *page, int order,
				int migratetype)
{
```

```
        spin_lock(&zone->lock);
        zone->all_unreclaimable = 0;
        zone->pages_scanned = 0;

        __mod_zone_page_state(zone, NR_FREE_PAGES, 1 << order);
        __free_one_page(page, zone, order, migratetype);
        spin_unlock(&zone->lock);
}
```

9.2.3.3 __free_one_page 函数

```
/*
 * Freeing function for a buddy system allocator.
 *
 * The concept of a buddy system is to maintain direct-mapped table
 * (containing bit values) for memory blocks of various "orders".
 * The bottom level table contains the map for the smallest allocatable
 * units of memory (here, pages), and each level above it describes
 * pairs of units from the levels below, hence, "buddies".
 * At a high level, all that happens here is marking the table entry
 * at the bottom level available, and propagating the changes upward
 * as necessary, plus some accounting needed to play nicely with other
 * parts of the VM system.
 * At each level, we keep a list of pages, which are heads of continuous
 * free pages of length of (1 << order) and marked with PG_buddy. Page's
 * order is recorded in page_private(page) field.
 * So when we are allocating or freeing one, we can derive the state of the
 * other.   That is, if we allocate a small block, and both were
 * free, the remainder of the region must be split into blocks.
 * If a block is freed, and its buddy is also free, then this
```

```
468   * triggers coalescing into a block of larger size.
469   *
470   * -- wli
471   */
472
473 static inline void __free_one_page(struct page *page,
474               struct zone *zone, unsigned int order,
475               int migratetype)
476 {
477         unsigned long page_idx;
478
478
479         if (unlikely(PageCompound(page)))
480               if (unlikely(destroy_compound_page(page, order)))
481                     return;
482
483         VM_BUG_ON(migratetype == -1);
484
485         page_idx = page_to_pfn(page) & ((1 << MAX_ORDER) - 1);
```

485 行：page_idx 表示 page 在 mem_map 数组中的下标模 1023 的值。

```
486
487         VM_BUG_ON(page_idx & ((1 << order) - 1));
488         VM_BUG_ON(bad_range(zone, page));
489
490         while (order < MAX_ORDER-1) {
491               unsigned long combined_idx;
492               struct page *buddy;
493
494               buddy = __page_find_buddy(page, page_idx, order);
```

494 行：寻找 page 块的伙伴。

```
406 static inline struct page *
```

```
407 __page_find_buddy(struct page *page, unsigned long page_idx, unsigned int order)
408 {
409         unsigned long buddy_idx = page_idx ^ (1 << order);
410
411         return page + (buddy_idx - page_idx);
412 }
```

要看懂上述函数，必须清楚如下几点：

1、伙伴算法的块的分配的方法，大小为 2^x 个页框的块，他的起始地址一定是 2^x 的倍数

2、在把两个大小为 2^(x-1)的块合并为大小为 2^x 的块的时候，需要考虑目前回收的块是 buddy 里面的第一个还是第二个，判断的依据就是 page_idx 的 2^order 这一位，如果为 1，则必须向前寻找 buddy，如果为 0，则必须向后寻找 buddy，这也是异或的精妙之处，如果写成 if else 可能更好理解，但是有了奇技淫巧才显得 linux 源代码的高深。

3、当 order 固定时，一个 page_idx 在指数为 order 的 free_list 中只可能有一个伙伴。

```
495             if (!page_is_buddy(page, buddy, order))
496                     break;
```

495-496 行：判断 buddy 是否为 page 的伙伴。

420-427 行的注释很清楚的说明了伙伴的条件。

1、buddy 不能再空洞中。

2、伙伴在伙伴系统中。

3、伙伴和 page 需要有相同的阶数

4、page 和 buddy 在相同的内存区域中。

```
420 /*
421  * This function checks whether a page is free && is the buddy
422  * we can do coalesce a page and its buddy if
423  * (a) the buddy is not in a hole &&
424  * (b) the buddy is in the buddy system &&
425  * (c) a page and its buddy have the same order &&
426  * (d) a page and its buddy are in the same zone.
```

```
427  *
428  * For recording whether a page is in the buddy system, we use PG_buddy.
429  * Setting, clearing, and testing PG_buddy is serialized by zone->lock.
430  *
431  * For recording page's order, we use page_private(page).
432  */
433 static inline int page_is_buddy(struct page *page, struct page *buddy,
434                                        int order)
435 {
436      if (!pfn_valid_within(page_to_pfn(buddy)))
437            return 0;
438
439      if (page_zone_id(page) != page_zone_id(buddy))
440            return 0;
441
442      if (PageBuddy(buddy) && page_order(buddy) == order) {
443            VM_BUG_ON(page_count(buddy) != 0);
444            return 1;
445      }
446      return 0;
447 }
```

```
497
498            /* Our buddy is free, merge with it and move up one order. */
499            list_del(&buddy->lru);
500            zone->free_area[order].nr_free--;
```

499-500 行：将伙伴冲阶 order 中移除，同时减少阶 order 中 nr_free。

```
501            rmv_page_order(buddy);
```

```
502             combined_idx = __find_combined_index(page_idx, order);
```

502 行：计算伙伴合并后新的 idx

```
503             page = page + (combined_idx - page_idx);
504             page_idx = combined_idx;
505             order++;
```

503-505 行：计算伙伴的头 page，得到伙伴的 idx，order++，进行下一轮循环。

```
506     }
507     set_page_order(page, order);
```

507 行：设置 page 的阶数

```
508     list_add(&page->lru,
509             &zone->free_area[order].free_list[migratetype]);
510     zone->free_area[order].nr_free++;
```

508-510 行：将伙伴添加到对应的阶中，且该阶的 nr_free++。

```
511 }
```

9.2.4 save_pg_dir

备份页全局目录，原因注释中讲的很清楚。

```
#ifdef CONFIG_ACPI_SLEEP
/*
 * ACPI suspend needs this for resume, because things like the intel-agp
 * driver might have split up a kernel 4MB mapping.
 */
char swsusp_pg_dir[PAGE_SIZE]
        __attribute__ ((aligned(PAGE_SIZE)));

static inline void save_pg_dir(void)
{
        memcpy(swsusp_pg_dir, swapper_pg_dir, PAGE_SIZE);
```

```
}
```

9.2.5 zap_low_mappings

这个函数很简单，就是把前面我们在 arch/x86/kernel/head_32.S 中设置的页全局目录的前若干项清零。这若干项到底是多少项呢？我们看看 KERNEL_PGD_BOUNDARY 是什么东西：

#define KERNEL_PGD_BOUNDARY pgd_index(PAGE_OFFSET)
#define pgd_index(address) (((address) >> PGDIR_SHIFT) & (PTRS_PER_PGD - 1))
#define PGDIR_SHIFT 22
#define PTRS_PER_PGD 1024

不错，0xc0000000>>22 & 1023= 768，这些也全局目录项代表虚拟地址前 3G 的页面，也就是所谓的用户区，我们在这里把它全清零了。

```
void zap_low_mappings(bool early)
{
	int i;

	/*
	 * Zap initial low-memory mappings.
	 *
	 * Note that "pgd_clear()" doesn't do it for
	 * us, because pgd_clear() is a no-op on i386.
	 */
	for (i = 0; i < KERNEL_PGD_BOUNDARY; i++) {
#ifdef CONFIG_X86_PAE
		set_pgd(swapper_pg_dir+i, __pgd(1 + __pa(empty_zero_page)));
#else
		set_pgd(swapper_pg_dir+i, __pgd(0));
#endif
	}
```

```
        if (early)
                __flush_tlb();
        else
                flush_tlb_all();
}
```

9.3 kmem_cache_init slab分配器初始化

9.3.1 slab分配器相关数据结构

在 kmem_cache_init 函数之前，内核使用 early_res 机制分配内存，但 early_res 毕竟是一个临时的过渡方案，只考虑分配的效率没有考虑内存管理中内存脆片的问题。上节的伙伴系统按也来分配内存，这个分配单位太大。现在是时候初始化 slab 分配器聊。

slab 分配器中的一个重要概念是缓存。一个缓存包含一些列的 slab，每一个 slab 由几个连续的物理页面组成，这些页面包含相同类型的对象。

在内核中，缓存的结构如下：

Include/linux/slab_def.h

9.3.1.1 kmem_cache数据结构

```
19 /*
20   * struct kmem_cache
21   *
22   * manages a cache.
23   */
24
25 struct kmem_cache {
26 /* 1) per-cpu data, touched during every alloc/free */
27         struct array_cache *array[NR_CPUS];
```

27:每 cpu 数据，每次分配/释放时访问，加快分配速度。

array 是一个指针数组，数组的长度是系统中 CPU 数目的个数。每个数组项指向一个 array_cache 示例，表示系统中的一个 CPU。

```
28 /* 2) Cache tunables. Protected by cache_chain_mutex */
29     unsigned int batchcount;
30     unsigned int limit;
31     unsigned int shared;
32
33     unsigned int buffer_size;
34     u32 reciprocal_buffer_size;
```

28-34 行：可调整的缓存参数

batchcount：在 per-CPU 列表为空时，从缓存的 slab 中获取对象的数目。它还表示在缓存增长时分配的对象数目。

limit：指定了 per-CPU 列表中保存的对象的最大数目。

```
35 /* 3) touched by every alloc & free from the backend */
36
37     unsigned int flags;     /* constant flags */
38     unsigned int num;       /* # of objs per slab */
39
```

37-38 行：常数标志。

```
40 /* 4) cache_grow/shrink */
41     /* order of pgs per slab (2^n) */
42     unsigned int gfporder;
43
44     /* force GFP flags, e.g. GFP_DMA */
45     gfp_t gfpflags;
46
47     size_t colour;          /* cache colouring range */
48     unsigned int colour_off;    /* colour offset */
49     struct kmem_cache *slabp_cache;
50     unsigned int slab_size;
51     unsigned int dflags;        /* dynamic flags */
```

```
52                                            ¢] 223L, 5300C
53        /* constructor func */
54        void (*ctor)(void *obj);
55
```

41-55 行：缓存的增长/缩减变量

```
56 /* 5) cache creation/removal */
57        const char *name;
58        struct list_head next;
59
```

57-58 行：缓存创建/删去相关变量

```
60 /* 6) statistics */

61 #ifdef CONFIG_DEBUG_SLAB
62        unsigned long num_active;
63        unsigned long num_allocations;
64        unsigned long high_mark;
65        unsigned long grown;
66        unsigned long reaped;
67        unsigned long errors;
68        unsigned long max_freeable;
69        unsigned long node_allocs;
70        unsigned long node_frees;
71        unsigned long node_overflow;    ¢] 223L, 5300C
72        atomic_t allochit;
73        atomic_t allocmiss;
74        atomic_t freehit;
75        atomic_t freemiss;
76
77        /*
78         * If debugging is enabled, then the allocator can add additional
79         * fields and/or padding to every object. buffer_size contains the total
```

```
80        * object size including these internal fields, the following two
81        * variables contain the offset to the user object and its size.
82        */
83       int obj_offset;
84       int obj_size;
85 #endif /* CONFIG_DEBUG_SLAB */
```

61-85 行：统计量

```
86
87       /*
88        * We put nodelists[] at the end of kmem_cache, because we want to size
89        * this array to nr_node_ids slots instead of MAX_NUMNODES
90        * (see kmem_cache_init())
91        * We still use [MAX_NUMNODES] and not [1] or [0] because cache_cache
92        * is statically defined, so we reserve the max number of nodes.
93        */
94       struct kmem_list3 *nodelists[MAX_NUMNODES];
```

94 行：slab 链表表头数据结构。Nodelists 是一个数组，每个数组项对应系统中一个可能的节点。

```
95       /*
96        * Do not add fields after nodelists[]
97        */
98 };
```

9.3.1.2 数据结构array_cache

```
255 /*
256  * struct array_cache
257  *
258  * Purpose:
259  * - LIFO ordering, to hand out cache-warm objects from _alloc
260  * - reduce the number of linked list operations
```

```
 * - reduce spinlock operations
 *
 * The limit is stored in the per-cpu structure to reduce the data cache
 * footprint.
 *
 */
struct array_cache {
	unsigned int avail;
	unsigned int limit;
	unsigned int batchcount;
	unsigned int touched;
	spinlock_t lock;
	void *entry[];	/*
			 * Must have this definition in here for the proper
			 * alignment of array_cache. Also simplifies accessing
			 * the entries.
			 */
};
```

avail:保存当前可用对象的数目。

entry：一个伪数组，方便访问 array_cache 实例之后缓存中的各个对象。

9.3.1.3 kmem_list3 数据结构

```
/*
 * The slab lists for all objects.
 */
struct kmem_list3 {
	struct list_head slabs_partial; /* partial list first, better asm code */
	struct list_head slabs_full;
	struct list_head slabs_free;
```

部分、满、空闲 slab 的链表头

```
297         unsigned long free_objects;
298         unsigned int free_limit;
299         unsigned int colour_next;      /* Per-node cache coloring */
300         spinlock_t list_lock;
301         struct array_cache *shared; /* shared per node */
302         struct array_cache **alien; /* on other nodes */
303         unsigned long next_reap;        /* updated without locking */
304         int free_touched;            /* updated without locking */
305 };
```

297-305 行：和页面回收的相关数据结构

9.3.2 kmem_cache_init 函数

为初始化 slab 数据结构，内核若干小于 1 页的内存，这些内存最适合使用 kmalloc 调用，但在 slab 分配器建立之前，是不能使用 kmalloc 分配内存的。

内核使用编译时创建的静态数据，为 slab 的初始化提供内存。

Kmen_cache_init 的主要过程是：

1、创建系统中的第一个 slab 缓存，为 kmem_cache 实例提供内存。第一个 slab 缓存的实例使用编译时的静态数据结构 initarray_cache 用作 per-CPU 数组，缓存名称为 cache_cache。

2、初始化一般性的的缓存，为使用用 kmalloc 的做好准备。

3、

```
 * Initialisation.   Called after the page allocator have been initialised and
 * before smp_init().
 */
void __init kmem_cache_init(void)
{
        size_t left_over;
        struct cache_sizes *sizes;
        struct cache_names *names;
        int i;
        int order;
```

```
        int node;

        if (num_possible_nodes() == 1)
                use_alien_caches = 0;

```

```
        for (i = 0; i < NUM_INIT_LISTS; i++) {
                kmem_list3_init(&initkmem_list3[i]);
                if (i < MAX_NUMNODES)
                        cache_cache.nodelists[i] = NULL;
        }
```

初始化全局变量 initkmem_list3

```
/*
 * Need this for bootstrapping a per node allocator.
 */
#define NUM_INIT_LISTS (3 * MAX_NUMNODES)
struct kmem_list3 __initdata initkmem_list3[NUM_INIT_LISTS];
#define CACHE_CACHE 0
#define SIZE_AC MAX_NUMNODES
#define SIZE_L3 (2 * MAX_NUMNODES)
```

initkmem_list3 是一个数组，数组的每个元素为 kmem_list3,数组的个数为最大节点数的 3 倍。

```
        set_up_list3s(&cache_cache, CACHE_CACHE);
```

初始化全局变量 cache_cache.

```

        /*
         * Fragmentation resistance on low memory - only use bigger
         * page orders on machines with more than 32MB of memory.
         */
        if (totalram_pages > (32 << 20) >> PAGE_SHIFT)
                slab_break_gfp_order = BREAK_GFP_ORDER_HI;
```

```
/* Bootstrap is tricky, because several objects are allocated
 * from caches that do not exist yet:
 * 1) initialize the cache_cache cache: it contains the struct
 *    kmem_cache structures of all caches, except cache_cache itself:
 *    cache_cache is statically allocated.
 *    Initially an __init data area is used for the head array and the
 *    kmem_list3 structures, it's replaced with a kmalloc allocated
 *    array at the end of the bootstrap.
 * 2) Create the first kmalloc cache.
 *    The struct kmem_cache for the new cache is allocated normally.
 *    An __init data area is used for the head array.
 * 3) Create the remaining kmalloc caches, with minimally sized
 *    head arrays.
 * 4) Replace the __init data head arrays for cache_cache and the first
 *    kmalloc cache with kmalloc allocated arrays.
 * 5) Replace the __init data for kmem_list3 for cache_cache and
 *    the other cache's with kmalloc allocated memory.
 * 6) Resize the head arrays of the kmalloc caches to their final sizes.
 */

node = numa_node_id();

/* 1) create the cache_cache */
INIT_LIST_HEAD(&cache_chain);
list_add(&cache_cache.next, &cache_chain);
```

将 cache_cache 连接到链表头 cache_chain 中。

```
cache_cache.colour_off = cache_line_size();
cache_cache.array[smp_processor_id()] = &initarray_cache.cache;
```

使用静态全局变量作为 cache_cache 缓存的 per-CPU 实例。

```
static struct arraycache_init initarray_cache __initdata =
```

```
	{ {0, BOOT_CPUCACHE_ENTRIES, 1, 0} };
static struct arraycache_init initarray_generic =
	{ {0, BOOT_CPUCACHE_ENTRIES, 1, 0} };

/* internal cache of cache description objs */
static struct kmem_cache cache_cache = {
	.batchcount = 1,
	.limit = BOOT_CPUCACHE_ENTRIES,
	.shared = 1,
	.buffer_size = sizeof(struct kmem_cache),
	.name = "kmem_cache",
};
```

initarray_cache 是内型为 arraycache_init 结构体，它的元素包含一个 array_cache 的示例，以及一个指针数组。

```
#define BOOT_CPUCACHE_ENTRIES	1
struct arraycache_init {
	struct array_cache cache;
	void *entries[BOOT_CPUCACHE_ENTRIES];
};
```

```
	cache_cache.nodelists[node] = &initkmem_list3[CACHE_CACHE + node];

	/*
	 * struct kmem_cache size depends on nr_node_ids, which
	 * can be less than MAX_NUMNODES.
	 */
	cache_cache.buffer_size = offsetof(struct kmem_cache, nodelists) +
				 nr_node_ids * sizeof(struct kmem_list3 *);
#if DEBUG
	cache_cache.obj_size = cache_cache.buffer_size;
#endif
```

```
1439        cache_cache.buffer_size = ALIGN(cache_cache.buffer_size,
1440                                    cache_line_size());
1441        cache_cache.reciprocal_buffer_size =
1442            reciprocal_value(cache_cache.buffer_size);
1443
1444        for (order = 0; order < MAX_ORDER; order++) {
1445            cache_estimate(order, cache_cache.buffer_size,
1446                cache_line_size(), 0, &left_over, &cache_cache.num);
1447            if (cache_cache.num)
1448                break;
1449        }
1450        BUG_ON(!cache_cache.num);
1451        cache_cache.gfporder = order;
1452        cache_cache.colour = left_over / cache_cache.colour_off;
1453        cache_cache.slab_size = ALIGN(cache_cache.num * sizeof(kmem_bufctl_t) +
1454                                    sizeof(struct slab), cache_line_size());
1455
```

1426-1455 行：初始化 cache_cache 字段的相关变量

```
1456        /* 2+3) create the kmalloc caches */
1457        sizes = malloc_sizes;
1458        names = cache_names;
```

Malloc_sizes，cache_names 是两个指针数组，其中分别存放一般性缓存的大小和名称。

```
1459
1460        /*
1461         * Initialize the caches that provide memory for the array cache and the
1462         * kmem_list3 structures first.   Without this, further allocations will
1463         * bug.
1464         */
1465
```

```
	sizes[INDEX_AC].cs_cachep =
kmem_cache_create(names[INDEX_AC].name,
				sizes[INDEX_AC].cs_size,
				ARCH_KMALLOC_MINALIGN,
				ARCH_KMALLOC_FLAGS|SLAB_PANIC,
				NULL);
```

创建大小为(sizeof(struct arraycache_init)的普通缓存。

```
#define INDEX_AC index_of(sizeof(struct arraycache_init))
#define INDEX_L3 index_of(sizeof(struct kmem_list3))
```

```

	if (INDEX_AC != INDEX_L3) {
		sizes[INDEX_L3].cs_cachep =
			kmem_cache_create(names[INDEX_L3].name,
				sizes[INDEX_L3].cs_size,
				ARCH_KMALLOC_MINALIGN,
				ARCH_KMALLOC_FLAGS|SLAB_PANIC,
				NULL);
	}
```

如果 arraycache_init 和 kmem_list3 大小相等，则共用同一普通缓存，否则创建大小为 sizeof(struct kmem_list3)的缓存。

这两个普通缓存非常重要，因为它是下面其它缓存的基础。因为每个 kmem_cache 缓存都需要分配 arraycache_init 和 kmem_list3 大小的对象，而这些对象需要通过 kmalloc 来分配，而 kmalloc 的分配有是建立的上述两个普通缓存的基础上的。

这里有一个问题大家必须明白，即为分配 arraycache_init 和 kmem_list3 建立的缓存也是一个 kmem_cache 实例，它们也需要 arraycache_init 和 kmem_list3，所以也需要 kmalloc 的分配，但是 kmalloc 的分配在 arraycache_init 和 kmem_list3 的缓存还没建立起来是不可使用的，这里就存在一个鸡与蛋的问题，起始从上面的代码可以看出，内核的解决办法是：为分配 arraycache_init 和 kmem_list3 的而建立的缓存的 arraycache_init 和 kmem_list3 使用静态的全局变量。具体的实现就不深入进去了，等到讲解虚拟内存系统是在详细讨论。

```
1480
1481         slab_early_init = 0;
1482
1483         while (sizes->cs_size != ULONG_MAX) {
1484                 /*
1485                  * For performance, all the general caches are L1 aligned.
1486                  * This should be particularly beneficial on SMP boxes, as it
1487                  * eliminates "false sharing".
1488                  * Note for systems short on memory removing the alignment will
1489                  * allow tighter packing of the smaller caches.
1490                  */
1491                 if (!sizes->cs_cachep) {
1492                         sizes->cs_cachep = kmem_cache_create(names->name,
1493                                         sizes->cs_size,
1494                                         ARCH_KMALLOC_MINALIGN,
1495                                         ARCH_KMALLOC_FLAGS|SLAB_PANIC,
1496                                         NULL);
1497                 }
1498 #ifdef CONFIG_ZONE_DMA
1499                 sizes->cs_dmacachep = kmem_cache_create(
1500                                         names->name_dma,
1501                                         sizes->cs_size,
1502                                         ARCH_KMALLOC_MINALIGN,
1503                                         ARCH_KMALLOC_FLAGS|SLAB_CACHE_DMA|
1504                                                 SLAB_PANIC,
1505                                         NULL);
1506 #endif
1507                 sizes++;
1508                 names++;
1509         }
```

1456-1509 行：建立剩下的一般性缓存。

```
1510        /* 4) Replace the bootstrap head arrays */
1511        {
1512                struct array_cache *ptr;
1513
1514                ptr = kmalloc(sizeof(struct arraycache_init), GFP_NOWAIT);
```

前面的一般性缓存已经建立，现在可以使用 kmalloc 分配内存了。

```
1515
1516                BUG_ON(cpu_cache_get(&cache_cache) != &initarray_cache.cache);
```

1516 行：首先判断 cache_cache 的 per-CPU 变量是否是从静态全局变量 initarray_cache 中来的，如果不是这内核初始化肯定是出问题了。

```
1517                memcpy(ptr, cpu_cache_get(&cache_cache),
1518                       sizeof(struct arraycache_init));
1519                /*
1520                 * Do not assume that spinlocks can be initialized via memcpy:
1521                 */
1522                spin_lock_init(&ptr->lock);
1523
1524                cache_cache.array[smp_processor_id()] = ptr;
```

替换掉之前的静态 per-CPU 变量。

```
1525
1526                ptr = kmalloc(sizeof(struct arraycache_init), GFP_NOWAIT);
1527
1528                BUG_ON(cpu_cache_get(malloc_sizes[INDEX_AC].cs_cachep)
1529                       != &initarray_generic.cache);
1530                memcpy(ptr, cpu_cache_get(malloc_sizes[INDEX_AC].cs_cachep),
1531                       sizeof(struct arraycache_init));
1532                /*
1533                 * Do not assume that spinlocks can be initialized via memcpy:
1534                 */
1535                spin_lock_init(&ptr->lock);
```

```
1536
1537            malloc_sizes[INDEX_AC].cs_cachep->array[smp_processor_id()] =
1538                    ptr;
1539        }
```

1510-1539 行：替换 bootstrap

```
1540        /* 5) Replace the bootstrap kmem_list3's */
1541        {
1542            int nid;
1543
1544            for_each_online_node(nid) {
1545                    init_list(&cache_cache, &initkmem_list3[CACHE_CACHE + nid], nid);
1546
1547                    init_list(malloc_sizes[INDEX_AC].cs_cachep,
1548                              &initkmem_list3[SIZE_AC + nid], nid);
1549
1550                    if (INDEX_AC != INDEX_L3) {
1551                            init_list(malloc_sizes[INDEX_L3].cs_cachep,
1552                                      &initkmem_list3[SIZE_L3 + nid], nid);
1553                    }
```

1545-1553：替换掉缓存 cache_cache，arraycache_init，kmem_list3 的 kmem_list3 变量。

```
1554            }
1555        }
1556
1557        g_cpucache_up = EARLY;
```

设置 g_cpucache_up 为 EARLY。g_cpucache_up 的状态控制着建立缓存时 per-cache 的分配机制。

```
1558 }
```

9.4 vmalloc_init非连续内存区初始化

```
void __init vmalloc_init(void)
{
        struct vmap_area *va;
        struct vm_struct *tmp;
        int i;

        for_each_possible_cpu(i) {
                struct vmap_block_queue *vbq;

                vbq = &per_cpu(vmap_block_queue, i);
                spin_lock_init(&vbq->lock);
                INIT_LIST_HEAD(&vbq->free);
        }

        /* Import existing vmlist entries. */
        for (tmp = vmlist; tmp; tmp = tmp->next) {
                va = kzalloc(sizeof(struct vmap_area), GFP_NOWAIT);
                va->flags = tmp->flags | VM_VM_AREA;
                va->va_start = (unsigned long)tmp->addr;
                va->va_end = va->va_start + tmp->size;
                __insert_vmap_area(va);
```

将已经存在的 vmallco 区域添加到 vmap 中。

```
        }

        vmap_area_pcpu_hole = VMALLOC_END;

        vmap_initialized = true;
```

设置 vmap_initialized 全局变量，表示已经初始化。

```
}
```

10. 其它重要的初始化

下面这些内容的初始化非常重要，但是和内存管理关系不大，等到分析相关主题时再详细讨论，下面只是列出这些目录。

10.1 初始化调度程序

sched_init

10.2 初始化中断处理系统

init_IRQ

10.3 软中断初始化

softirq_init

10.4 定时器中断初始化

time_init

11. slab后续初始化 kmem_cache_init_late

```
void __init kmem_cache_init_late(void)
{
	struct kmem_cache *cachep;

	/* 6) resize the head arrays to their final sizes */
```

```
1565        mutex_lock(&cache_chain_mutex);
1566        list_for_each_entry(cachep, &cache_chain, next)
1567                if (enable_cpucache(cachep, GFP_NOWAIT))
```

1567 行：初始化阶段 local cache 的大小是固定的，现在要根据对象大小重新计算

```
1568                        BUG();
1569        mutex_unlock(&cache_chain_mutex);
1570
1571        /* Done! */
1572        g_cpucache_up = FULL;
```

到 1572 行：slab 分配器初始化全部完成。

```
1573
1574        /* Annotate slab for lockdep -- annotate the malloc caches */
1575        init_lock_keys();
1576
1577        /*
1578         * Register a cpu startup notifier callback that initializes
1579         * cpu_cache_get for all new cpus
1580         */
1581        register_cpu_notifier(&cpucache_notifier);
1582
1583        /*
1584         * The reap timers are started later, with a module init call: That part
1585         * of the kernel is not yet operational.
1586         */
1587 }
```

11.1 enable_cpucache

```
3947 static int enable_cpucache(struct kmem_cache *cachep, gfp_t gfp)
3948 {
```

```
	int err;
	int limit, shared;

	/*
	 * The head array serves three purposes:
	 * - create a LIFO ordering, i.e. return objects that are cache-warm
	 * - reduce the number of spinlock operations.
	 * - reduce the number of linked list operations on the slab and
	 *   bufctl chains: array operations are cheaper.
	 * The numbers are guessed, we should auto-tune as described by
	 * Bonwick.
	 */
	if (cachep->buffer_size > 131072)
		limit = 1;
	else if (cachep->buffer_size > PAGE_SIZE)
		limit = 8;
	else if (cachep->buffer_size > 1024)
		limit = 24;
	else if (cachep->buffer_size > 256)
		limit = 54;
	else
		limit = 120;
```

根据对象大小，计算本地 cpu 缓存中对象的数目。

```

	/*
	 * CPU bound tasks (e.g. network routing) can exhibit cpu bound
	 * allocation behaviour: Most allocs on one cpu, most free operations
	 * on another cpu. For these cases, an efficient object passing between
	 * cpus is necessary. This is provided by a shared array. The array
	 * replaces Bonwick's magazine layer.
	 * On uniprocessor, it's functionally equivalent (but less efficient)
```

```
3979          * to a larger limit. Thus disabled by default.
3980          */
3981         shared = 0;
3982         if (cachep->buffer_size <= PAGE_SIZE && num_possible_cpus() > 1)

3983                 shared = 8;
3984
3985 #if DEBUG
3986         /*
3987          * With debugging enabled, large batchcount lead to excessively long
3988          * periods with disabled local interrupts. Limit the batchcount
3989          */
3990         if (limit > 32)
3991                 limit = 32;
3992 #endif
3993         err = do_tune_cpucache(cachep, limit, (limit + 1) / 2, shared, gfp);
```

3993 行：配置 local cache

```
3994         if (err)
3995                 printk(KERN_ERR "enable_cpucache failed for %s, error %d.\n",
3996                                 cachep->name, -err);
3997         return err;
3998 }
```

11.2 do_tune_cpucache

```
static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
                                int batchcount, int shared, gfp_t gfp)
{
        struct ccupdate_struct *new;
        int i;

```

```
	new = kzalloc(sizeof(*new), gfp);
	if (!new)
		return -ENOMEM;

	for_each_online_cpu(i) {
		new->new[i] = alloc_arraycache(cpu_to_node(i), limit,
						batchcount, gfp);
		if (!new->new[i]) {
			for (i--; i >= 0; i--)
				kfree(new->new[i]);
			kfree(new);
			return -ENOMEM;
		}
	}
```

为每个 cpu 分配新的 struct array_cache 对象

```
	new->cachep = cachep;

	on_each_cpu(do_ccupdate_local, (void *)new, 1);
```

用新的 struct array_cache 对象替换旧的 struct array_cache 对象

```

	check_irq_on();
	cachep->batchcount = batchcount;
	cachep->limit = limit;
	cachep->shared = shared;

	for_each_online_cpu(i) {
		struct array_cache *ccold = new->new[i];
		if (!ccold)
			continue;
		spin_lock_irq(&cachep->nodelists[cpu_to_node(i)]->list_lock);
		free_block(cachep, ccold->entry, ccold->avail, cpu_to_node(i));
```

```
        spin_unlock_irq(&cachep->nodelists[cpu_to_node(i)]->list_lock);
        kfree(ccold);
    }
```

释放旧的 struct array_cache 对象

```
    kfree(new);
    return alloc_kmemlist(cachep, gfp);
```

初始化 shared local cache 和 slab 三链

```
}
```

```
static void do_ccupdate_local(void *info)
{
    struct ccupdate_struct *new = info;
    struct array_cache *old;

    check_irq_off();
    old = cpu_cache_get(new->cachep);

    new->cachep->array[smp_processor_id()] = new->new[smp_processor_id()];
```

指向新的 struct array_cache 对象

```
    new->new[smp_processor_id()] = old;
```

保存旧的 struct array_cache 对象*/

```
}
```

11.3 alloc_kmemlist

初始化 shared local cache 和 slab 三链，初始化完成后，slab 三链中没有任何 slab

```
 * This initializes kmem_list3 or resizes various caches for all nodes.
 */
```

```
static int alloc_kmemlist(struct kmem_cache *cachep, gfp_t gfp)
{
        int node;
        struct kmem_list3 *l3;
        struct array_cache *new_shared;
        struct array_cache **new_alien = NULL;

        for_each_online_node(node) {

                if (use_alien_caches) {
                        new_alien = alloc_alien_cache(node, cachep->limit, gfp);
                        if (!new_alien)
                                goto fail;
                }

                new_shared = NULL;
                if (cachep->shared) {
                        new_shared = alloc_arraycache(node,
                                cachep->shared*cachep->batchcount,
                                        0xbaadf00d, gfp);
```

/* 分配 shared local cache */

```
                        if (!new_shared) {
                                free_alien_cache(new_alien);
                                goto fail;
                        }
                }

```

获得旧的 slab 三链

```
                l3 = cachep->nodelists[node];
                if (l3) {
                        struct array_cache *shared = l3->shared;
```

```

                spin_lock_irq(&l3->list_lock);

                if (shared)
//旧 slab 三链指针不为空，需要先释放旧的资源
                        free_block(cachep, shared->entry,
                                        shared->avail, node);

//指向新的 shared local cache
                l3->shared = new_shared;
                if (!l3->alien) {
                        l3->alien = new_alien;
                        new_alien = NULL;
                }
//计算 cache 中空闲对象的上限
                l3->free_limit = (1 + nr_cpus_node(node)) *
                                cachep->batchcount + cachep->num;
                spin_unlock_irq(&l3->list_lock);
                kfree(shared);
                free_alien_cache(new_alien);
//释放旧 shared local cache 的 struct array_cache 对象
                continue;
        }
//如果没有旧的 l3，分配新的 slab 三链
        l3 = kmalloc_node(sizeof(struct kmem_list3), gfp, node);
        if (!l3) {
                free_alien_cache(new_alien);
                kfree(new_shared);
                goto fail;
        }

```

```
//初始化 slab 三链
                kmem_list3_init(l3);
                l3->next_reap = jiffies + REAPTIMEOUT_LIST3 +
                        ((unsigned long)cachep) % REAPTIMEOUT_LIST3;
                l3->shared = new_shared;
                l3->alien = new_alien;
                l3->free_limit = (1 + nr_cpus_node(node)) *
                                cachep->batchcount + cachep->num;
                cachep->nodelists[node] = l3;
        }
        return 0;

fail:
        if (!cachep->next.next) {
                /* Cache is not active yet. Roll back what we did */
                node--;
                while (node >= 0) {
                        if (cachep->nodelists[node]) {
                                l3 = cachep->nodelists[node];

                                kfree(l3->shared);
                                free_alien_cache(l3->alien);
                                kfree(l3);
                                cachep->nodelists[node] = NULL;
                        }
                        node--;
                }
        }
        return -ENOMEM;
}
```

12. 启动控制台输出

console_init

将保存在缓存中的内容输出到屏幕上。

13. fork_init

根据物理内存大小计算允许创建进程/线程的数量

```
void __init fork_init(unsigned long mempages)
{
#ifndef __HAVE_ARCH_TASK_STRUCT_ALLOCATOR
#ifndef ARCH_MIN_TASKALIGN
#define ARCH_MIN_TASKALIGN   L1_CACHE_BYTES
#endif
        /* create a slab on which task_structs can be allocated */
        task_struct_cachep =
                kmem_cache_create("task_struct", sizeof(struct task_struct),
                        ARCH_MIN_TASKALIGN, SLAB_PANIC | SLAB_NOTRACK, NULL);
#endif

        /* do the arch specific task caches init */
        arch_task_cache_init();

        /*
         * The default maximum number of threads is set to a safe
         * value: the thread structures can take up at most half
         * of memory.
         */
```

```
	max_threads = mempages / (8 * THREAD_SIZE / PAGE_SIZE);

	/*
	 * we need to allow at least 20 threads to boot a system
	 */
	if(max_threads < 20)
		max_threads = 20;

	init_task.signal->rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;
	init_task.signal->rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
	init_task.signal->rlim[RLIMIT_SIGPENDING] =
		init_task.signal->rlim[RLIMIT_NPROC];
}
```

14. proc_caches_init

与进程相关数据结构的缓存创建

```
void __init proc_caches_init(void)
{
	sighand_cachep = kmem_cache_create("sighand_cache",
			sizeof(struct sighand_struct), 0,
			SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_DESTROY_BY_RCU|
			SLAB_NOTRACK, sighand_ctor);
	signal_cachep = kmem_cache_create("signal_cache",
			sizeof(struct signal_struct), 0,
			SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_NOTRACK, NULL);
	files_cachep = kmem_cache_create("files_cache",
			sizeof(struct files_struct), 0,
```

```
                SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_NOTRACK, NULL);
        fs_cachep = kmem_cache_create("fs_cache",
                sizeof(struct fs_struct), 0,
                SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_NOTRACK, NULL);
        mm_cachep = kmem_cache_create("mm_struct",
                sizeof(struct mm_struct), ARCH_MIN_MMSTRUCT_ALIGN,
                SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_NOTRACK, NULL);
        vm_area_cachep = KMEM_CACHE(vm_area_struct, SLAB_PANIC);
        mmap_init();
}
```

15. 块缓存初始化buffer_init

```
void __init buffer_init(void)
{
        int nrpages;

        bh_cachep = kmem_cache_create("buffer_head",
                sizeof(struct buffer_head), 0,
                        (SLAB_RECLAIM_ACCOUNT|SLAB_PANIC|
                        SLAB_MEM_SPREAD),
                        NULL);

        /*
         * Limit the bh occupancy to 10% of ZONE_NORMAL
         */
        nrpages = (nr_free_buffer_pages() * 10) / 100;
```

```
        max_buffer_heads = nrpages * (PAGE_SIZE / sizeof(struct buffer_head));
        hotcpu_notifier(buffer_cpu_notify, 0);
}
```

16. 虚拟根文件系统安装vfs_caches_init

该函数对文件系统来说非常重要，等讲解文件系统时在详细分析。

```
void __init vfs_caches_init(unsigned long mempages)
{
        unsigned long reserve;

        /* Base hash sizes on available memory, with a reserve equal to
                   150% of current kernel size */

        reserve = min((mempages - nr_free_pages()) * 3/2, mempages - 1);
        mempages -= reserve;

        names_cachep = kmem_cache_create("names_cache", PATH_MAX, 0,
                        SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL);

        dcache_init();
        inode_init();
        files_init(mempages);
        mnt_init();
        bdev_cache_init();
        chrdev_init();
}
```

17. 安装PROC文件系统proc_root_init

```
void __init proc_root_init(void)
{
	int err;

	proc_init_inodecache();
	err = register_filesystem(&proc_fs_type);
	if (err)
		return;
	proc_mnt = kern_mount_data(&proc_fs_type, &init_pid_ns);
	err = PTR_ERR(proc_mnt);
	if (IS_ERR(proc_mnt)) {
		unregister_filesystem(&proc_fs_type);
		return;
	}

	proc_symlink("mounts", NULL, "self/mounts");

	proc_net_init();

#ifdef CONFIG_SYSVIPC
	proc_mkdir("sysvipc", NULL);
#endif
	proc_mkdir("fs", NULL);
	proc_mkdir("driver", NULL);
	proc_mkdir("fs/nfsd", NULL); /* somewhere for the nfsd filesystem to be mounted */
#if defined(CONFIG_SUN_OPENPROMFS) || defined(CONFIG_SUN_OPENPROMFS_MODULE)
	/* just give it a mountpoint */
```

```
        proc_mkdir("openprom", NULL);
#endif
        proc_tty_init();
#ifdef CONFIG_PROC_DEVICETREE
        proc_device_tree_init();
#endif
        proc_mkdir("bus", NULL);
        proc_sys_init();
}
```

18. 后start_kernel时代

start_kernel 最后调用 rest_init 函数进行初始化。

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

        rcu_scheduler_starting();
        kernel_thread(kernel_init, NULL, CLONE_FS | CLONE_SIGHAND);
        numa_default_policy();
        pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
        rcu_read_lock();
        kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
        rcu_read_unlock();
        unlock_kernel();

        /*
         * The boot idle thread must execute schedule()
```

```
	 * at least once to get things moving:
	 */
	init_idle_bootup_task(current);
	preempt_enable_no_resched();
	schedule();
	preempt_disable();

	/* Call into cpu_idle with preempt disabled */
	cpu_idle();
}
```

rest_init 使用 kernel_thread 创建内核线程 init(1 号进程)。

rest_init 使用 kernel_thread 创建内核线程 kthreadd。Kthreadd 根据内核需要，动态的创建其他内核线程。

0 还进程进入 cpu_idle 中，在系统中没有其他就绪进程时，就进入 0 号进程运行。

18.1 Kthreadd

```
int kthreadd(void *unused)
{
	struct task_struct *tsk = current;

	/* Setup a clean context for our children to inherit. */
	set_task_comm(tsk, "kthreadd");
	ignore_signals(tsk);
	set_cpus_allowed_ptr(tsk, cpu_all_mask);
	set_mems_allowed(node_states[N_HIGH_MEMORY]);

	current->flags |= PF_NOFREEZE | PF_FREEZER_NOSIG;

	for (;;) {
```

```
		set_current_state(TASK_INTERRUPTIBLE);
```

将本进程设置为可中断睡眠模式。

```
		if (list_empty(&kthread_create_list))
```

如果没有需要创建的内核线程，则让出 CPU。

```
			schedule();
		__set_current_state(TASK_RUNNING);

		spin_lock(&kthread_create_lock);
		while (!list_empty(&kthread_create_list)) {
			struct kthread_create_info *create;

			create = list_entry(kthread_create_list.next,
						struct kthread_create_info, list);
			list_del_init(&create->list);
			spin_unlock(&kthread_create_lock);

			create_kthread(create);
```

创建内核线程。

kthread_create_list 在内核运行的过程中会变化，接受创建线程的请求。

```

			spin_lock(&kthread_create_lock);
		}
		spin_unlock(&kthread_create_lock);
	}

	return 0;
}
```

18.2 cpu_idle

空转函数。

```
/*
 * The idle thread. There's no useful work to be
 * done, so just try to conserve power and have a
 * low exit latency (ie sit in a loop waiting for
 * somebody to say that they'd like to reschedule)
 */
void cpu_idle(void)
{
	int cpu = smp_processor_id();

	/*
	 * If we're the non-boot CPU, nothing set the stack canary up
	 * for us.   CPU0 already has it initialized but no harm in
	 * doing it again.   This is a good place for updating it, as
	 * we wont ever return from this function (so the invalid
	 * canaries already on the stack wont ever trigger).
	 */
	boot_init_stack_canary();

	current_thread_info()->status |= TS_POLLING;

	/* endless idle loop with no priority at all */
	while (1) {
		tick_nohz_stop_sched_tick(1);
		while (!need_resched()) {

			check_pgt_cache();
			rmb();

			if (cpu_is_offline(cpu))
				play_dead();
```

```
		local_irq_disable();
		/* Don't trace irqs off for idle */
		stop_critical_timings();
		pm_idle();
		start_critical_timings();
	}
	tick_nohz_restart_sched_tick();
	preempt_enable_no_resched();
	schedule();
	preempt_disable();
}
}
```

18.3 Kernel_init

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

	/*
	 * init can allocate pages on any node
	 */
	set_mems_allowed(node_states[N_HIGH_MEMORY]);
```

init 可以在所有节点上分配内存

```
	/*
	 * init can run on any cpu.
	 */
	set_cpus_allowed_ptr(current, cpu_all_mask);
```

init 可以在所有节点上运行。

```
	/*
```

```
	 * Tell the world that we're going to be the grim
	 * reaper of innocent orphaned children.
	 *
	 * We don't want people to have to make incorrect
	 * assumptions about where in the task array this
	 * can be found.
	 */
	init_pid_ns.child_reaper = current;//设置当前进程为接受孤儿进程的进程

	cad_pid = task_pid(current);

	smp_prepare_cpus(setup_max_cpus);

	do_pre_smp_initcalls();
	start_boot_trace();

	smp_init();
	sched_init_smp();

	do_basic_setup();

	/* Open the /dev/console on the rootfs, this should never fail */
	if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) < 0)
		printk(KERN_WARNING "Warning: unable to open an initial console.\n");

	(void) sys_dup(0);
	(void) sys_dup(0);
	/*
	 * check if there is an early userspace init.  If yes, let it do all
	 * the work
	 */
```

```

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

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

	/*
	 * Ok, we have completed the initial bootup, and
	 * we're essentially up and running. Get rid of the
	 * initmem segments and start the user-mode stuff..
	 */

	init_post();
	return 0;
}
```

18.3.1 子系统初始化

```
static void __init do_basic_setup(void)
{
	init_workqueues();//初始化工作队列 events
	cpuset_init_smp();//空函数
	usermodehelper_init();//初始化工作队列 khelper
	init_tmpfs();//初始化临时文件系统
	driver_init();//建立设备驱动模型
	init_irq_proc();//向/proc 文件系统中增加子目录 irq
	do_ctors();//初始化内核构造函数
```

```
        do_initcalls();//首先说一下，内核中有一个专门的节，用来存放初始化末尾要被调用的函数，改函数就是用来调用该节中的函数的。
}
```

18.3.2 启动SHELL

```
 * makes it inline to init() and it becomes part of init.text section
 */
static noinline int init_post(void)
        __releases(kernel_lock)
{
        /* need to finish all async __init code before freeing the memory */
        async_synchronize_full();
        free_initmem();
        unlock_kernel();
        mark_rodata_ro();
        system_state = SYSTEM_RUNNING;
        numa_default_policy();


        current->signal->flags |= SIGNAL_UNKILLABLE;

        if (ramdisk_execute_command) {
                run_init_process(ramdisk_execute_command);
                printk(KERN_WARNING "Failed to execute %s\n",
                                ramdisk_execute_command);
        }

        /*
         * We try each of these until one succeeds.
         *
```

```
     * The Bourne shell can be used instead of init if we are
     * trying to recover a really broken machine.
     */
    if (execute_command) {
        run_init_process(execute_command);
        printk(KERN_WARNING "Failed to execute %s.  Attempting "
                    "defaults...\n", execute_command);
    }
    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");

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

```
805 static void run_init_process(char *init_filename)
806 {
807         argv_init[0] = init_filename;
808         kernel_execve(init_filename, argv_init, envp_init);
809 }
```

808 行，从内核调用 execve 函数，进入 shell 进程，内核第一次进入用户空间（从内核空间进入用户空间）.

18.3.3 Initrd以及磁盘根文件系统的安

在Kernel_init的904 prepare_namespace()函数，会进行根文件系统的安装。

```
/*
 * Prepare the namespace - decide what/where to mount, load ramdisks, etc.
 */
```

```
void __init prepare_namespace(void)
{
	int is_floppy;

	if (root_delay) {
		printk(KERN_INFO "Waiting %dsec before mounting root device...\n",
		       root_delay);
		ssleep(root_delay);
	}

	/*
	 * wait for the known devices to complete their probing
	 *
	 * Note: this is a potential source of long boot delays.
	 * For example, it is not atypical to wait 5 seconds here
	 * for the touchpad of a laptop to initialize.
	 */
	wait_for_device_probe();

	md_run_setup();

	if (saved_root_name[0]) {
		root_device_name = saved_root_name;
		if (!strncmp(root_device_name, "mtd", 3) ||
		    !strncmp(root_device_name, "ubi", 3)) {
			mount_block_root(root_device_name, root_mountflags);
			goto out;

		}
		ROOT_DEV = name_to_dev_t(root_device_name);
		if (strncmp(root_device_name, "/dev/", 5) == 0)
```

```
                root_device_name += 5;
        }

        if (initrd_load())
                goto out;
```

根据内核启动参数的配置，决定是否加载 initrd（initrd 不是必须的）。

```

        /* wait for any asynchronous scanning to complete */
        if ((ROOT_DEV == 0) && root_wait) {
                printk(KERN_INFO "Waiting for root device %s...\n",
                        saved_root_name);
                while (driver_probe_done() != 0 ||
                        (ROOT_DEV = name_to_dev_t(saved_root_name)) == 0)
                        msleep(100);
                async_synchronize_full();
        }

        is_floppy = MAJOR(ROOT_DEV) == FLOPPY_MAJOR;

        if (is_floppy && rd_doload && rd_load_disk(0))
                ROOT_DEV = Root_RAM0;

        mount_root();
```

加载磁盘根文件系统

```
out:
        devtmpfs_mount("dev");
        sys_mount(".", "/", NULL, MS_MOVE, NULL);
        sys_chroot(".");
}
```

19. 参考书籍

[1] linux 内核 2.6.34 源码

[2] 深入了解 linux 内核 （第三版）

[3] linxu 内核源代码情景分析 毛德超

[4] 深入 linxu 内核架构

[5] http://blog.csdn.net/yunsongice(网络资源)