Linux源码分析之内核启动流程

main.c start_kernel 函数开始的，这个文件在linux源码里面的init文件夹

```
asmlinkage __visible void __init start_kernel(void)
{
    char *command_line;
    char *after_dashes;

    set_task_stack_end_magic(&amp;init_task); //1
    smp_setup_processor_id();
    debug_objects_early_init();

    cgroup_init_early();

    local_irq_disable();
    early_boot_irqs_disabled = true;

    /*
     * Interrupts are still disabled. Do necessary setups, then
     * enable them.
     */
    boot_cpu_init();
    page_address_init();
    pr_notice("%s", linux_banner);
    setup_arch(&amp;command_line);
    /*
     * Set up the the initial canary and entropy after arch
     * and after adding latent and command line entropy.
     */
    add_latent_entropy();
    add_device_randomness(command_line, strlen(command_line));
    boot_init_stack_canary();
    mm_init_cpumask(&amp;init_mm);
    setup_command_line(command_line);
    setup_nr_cpu_ids();
    setup_per_cpu_areas();
    smp_prepare_boot_cpu();    /* arch-specific boot-cpu hooks */
```

```
boot_cpu_hotplug_init();

build_all_zonelists(NULL);
page_alloc_init();

pr_notice("Kernel command line: %s\n", boot_command_line);
parse_early_param();
after_dashes = parse_args("Booting kernel",
              static_command_line, __start___param,
              __stop___param - __start___param,
              -1, -1, NULL, &amp;unknown_bootoption);
if (!IS_ERR_OR_NULL(after_dashes))
    parse_args("Setting init args", after_dashes, NULL, 0, -1, -1,
           NULL, set_init_arg);

jump_label_init();

/*
 * These use large bootmem allocations and must precede
 * kmem_cache_init()
 */
setup_log_buf(0);
vfs_caches_init_early();
sort_main_extable();
trap_init();
mm_init();

ftrace_init();

/* trace_printk can be enabled here */
early_trace_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();
/*
```

```c
	 * Disable preemption - early bootup scheduling is extremely
	 * fragile until we cpu_idle() for the first time.
	 */
	preempt_disable();
	if (WARN(!irqs_disabled(),
		 "Interrupts were enabled *very* early, fixing it\n"))
		local_irq_disable();
	radix_tree_init();

	/*
	 * Set up housekeeping before setting up workqueues to allow the unbound
	 * workqueue to take non-housekeeping into account.
	 */
	housekeeping_init();

	/*
	 * Allow workqueue creation and work item queueing/cancelling
	 * early.  Work item execution depends on kthreads and starts after
	 * workqueue_init().
	 */
	workqueue_init_early();

	rcu_init();

	/* Trace events are available after this */
	trace_init();

	if (initcall_debug)
		initcall_debug_enable();

	context_tracking_init();
	/* init some links before init_ISA_irqs() */
	early_irq_init();
	init_IRQ();
	tick_init();
	rcu_init_nohz();
	init_timers();
	hrtimers_init();
	softirq_init();
```

```c
timekeeping_init();
time_init();
printk_safe_init();
perf_event_init();
profile_init();
call_function_init();
WARN(!irqs_disabled(), "Interrupts were enabled early\n");

early_boot_irqs_disabled = false;
local_irq_enable();

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("Too many boot %s vars at `%s'", panic_later,
          panic_param);

lockdep_init();

/*
 * 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();

/*
 * This needs to be called before any devices perform DMA
 * operations that might use the SWIOTLB bounce buffers. It will
 * mark the bounce buffers as decrypted so that their usage will
 * not cause "plain-text" data to be decrypted when accessed.
 */
mem_encrypt_init();
```

```
#ifdef CONFIG_BLK_DEV_INITRD
    if (initrd_start &amp;&amp; !initrd_below_start_ok &amp;&amp;
        page_to_pfn(virt_to_page((void *)initrd_start)) &lt; min_low_pfn) {
        pr_crit("initrd overwritten (0x%08lx &lt; 0x%08lx) - disabling it.\n",
            page_to_pfn(virt_to_page((void *)initrd_start)),
            min_low_pfn);
        initrd_start = 0;
    }
#endif
    kmemleak_init();
    setup_per_cpu_pageset();
    numa_policy_init();
    acpi_early_init();
    if (late_time_init)
        late_time_init();
    sched_clock_init();
    calibrate_delay();
    pid_idr_init();
    anon_vma_init();
#ifdef CONFIG_X86
    if (efi_enabled(EFI_RUNTIME_SERVICES))
        efi_enter_virtual_mode();
#endif
    thread_stack_cache_init();
    cred_init();
    fork_init();
    proc_caches_init();
    uts_ns_init();
    buffer_init();
    key_init();
    security_init();
    dbg_late_init();
    vfs_caches_init();
    pagecache_init();
    signals_init();
    seq_file_init();
```

```
    proc_root_init();
    nsfs_init();
    cpuset_init();
    cgroup_init();
    taskstats_init_early();
    delayacct_init();

    check_bugs();

    acpi_subsystem_init();
    arch_post_acpi_subsys_init();
    sfi_init_late();

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

首先来看下这个函数set_task_stack_end_magic(&init_task);
在linux里面所有的进程都是由父进程创建而来，所以说在启动内核的时候需要有个祖先进程，这个进程是系统创建的
第一个进程，我们称为0号进程，它是唯一一个没有通过fork或者kernel_thread的进程

2. 初始化系统调用，对应的函数就是trap_init();这里面设置了很多中断门，用于处理各种中断
系统调用也是通过发送中断的方式进行的。
3. 内存管理模块的初始化，对应的函数是mm_init()；

4. 初始化任务调度，对应的函数就是sched_init();

5. preempt_disable();

6. tick_init();这个函数是时钟初始化

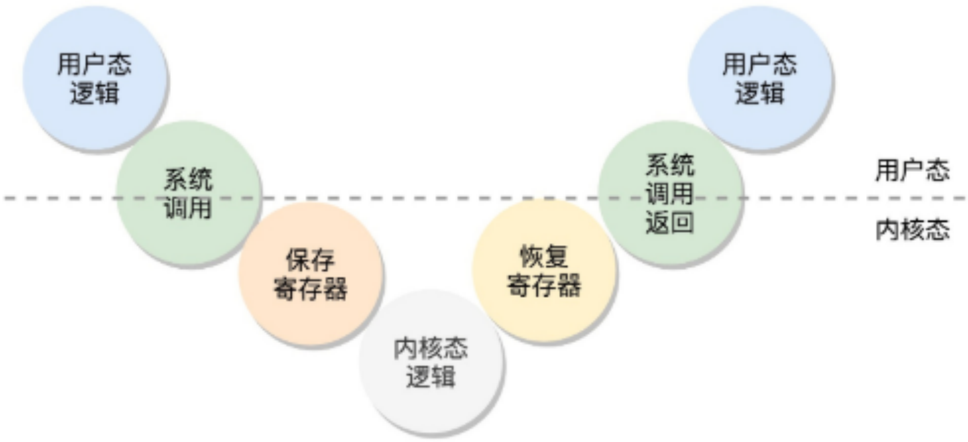
最后start_kernel()调用的是rest_init()

```
noinline void __ref rest_init(void)
{
    struct task_struct *tsk;
```

```
int pid;

rcu_scheduler_starting();
/*
 * We need to spawn init first so that it obtains pid 1, however
 * the init task will end up wanting to create kthreads, which, if
 * we schedule it before we create kthreadd, will OOPS.
 */
pid = kernel_thread(kernel_init, NULL, CLONE_FS);//2
/*
 * Pin init on the boot CPU. Task migration is not properly working
 * until sched_init_smp() has been run. It will set the allowed
 * CPUs for init to the non isolated CPUs.
 */
rcu_read_lock();
tsk = find_task_by_pid_ns(pid, &amp;init_pid_ns);
set_cpus_allowed_ptr(tsk, cpumask_of(smp_processor_id()));
rcu_read_unlock();

numa_default_policy();
pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);//3
rcu_read_lock();
kthreadd_task = find_task_by_pid_ns(pid, &amp;init_pid_ns);
rcu_read_unlock();

/*
 * Enable might_sleep() and smp_processor_id() checks.
 * They cannot be enabled earlier because with CONFIG_PREEMPT=y
 * kernel_thread() would trigger might_sleep() splats. With
 * CONFIG_PREEMPT_VOLUNTARY=y the init task might have scheduled
 * already, but it's stuck on the kthreadd_done completion.
 */
system_state = SYSTEM_SCHEDULING;

complete(&amp;kthreadd_done);

/*
 * The boot idle thread must execute schedule()
 * at least once to get things moving:
```

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

首先调用kernel_thread() 函数，用来创建用户态的第一个进程，这个进程是所有用户态进程的祖先进程，我们称为1号进程.
这个过程就是这样的，用户态-》系统调用-》保存寄存器-》内核态执行系统调用-》恢复寄存器-》返回用户态 接着运行



 然后接着说这个一号进程启动过程，现在这个进程还是在内核态的，那么要怎么把它搞到用户态里面的，
一般都是从用户态到内核态在返回到用户态，很少见过直接从内核态开始然后到用户态的
看下下面这个代码

```
void
start_thread(struct pt_regs *regs, unsigned long new_ip, unsigned long new_sp)
{
set_user_gs(regs, 0);
regs-&gt;fs    = 0;
regs-&gt;ds    = __USER_DS;
regs-&gt;es    = __USER_DS;
regs-&gt;ss    = __USER_DS;
```

```
regs-&gt;cs    = __USER_CS;
regs-&gt;ip    = new_ip;
regs-&gt;sp    = new_sp;
regs-&gt;flags    = X86_EFLAGS_IF;
force_iret();
}
EXPORT_SYMBOL_GPL(start_thread);
```

。

用户态的祖先进程创建完了，那么内核态有没有一个祖先进程呢?
有的，rest_init第二大事情就是第三个进程，也就是2号进程。