Linux中断处理流程(基于Linux-5.10.4) 作者: 王利涛 视频配套地址: https://wanglitao.taobao.com vector_stub宏定义 中断处理,CPU硬件自动完成的部分 .macro□vector_stub, name, mode, correction=0 系统启动过程中,每个控制器(irq_domain)初始化,会建立硬件中断号(HW interrupt ID)和软中断号(IRQ number)之间的映射 存CPSR到SPSR irq寄存器 align□5 b置CPSR控制位, 让CPU进入ARM状态、IRQ模式 SR中的IRQ位置一,硬件自动关闭IRQ static int gic_irq_domain_map(struct irq_domain *d, unsigned int irq, irq_hw_number_t hw) vector_\name: 当前中断地址(返回地址)保存到LR irg寄存器 □.if \correction 设置PC指针PC=0x00000018,跳转到中断向量表执行 □sub□lr, lr, #\correction struct gic_chip_data *gic = d->host_data; struct irq_data *irqd = irq_desc_get_irq_data(irq_to_desc(irq)); □@ (parent CPSR) ARM 中断向量表 \square stmia \square sp, {r0, lr} \square @ save r0, lr □mrs□1r, spsr section .vectors, "ax", %progbits rq domain set info(d, irq, hw, &gic->chip, d->host data, handle percpu devid fasteoi ipi, NULL, NULL); □str□lr, [sp, #8]□@ save spsr vectors start: W(b) □vector_rst lirq_set_percpu_devid(irq); \square W(b) \square vector_und @ Prepare for SVC32 mode. IRQs remain disabled. \square W(1dr) \square pc, .L_vectors_start + 0x1000 irq_domain_set_info(d, irq, hw, &gic->chip, d->host_data, handle_percpu_devid_irq, NULL, NULL); □W(b)□vector_pabt □mrs□r0, cpsr \square W(b) \square vector_dabt □eor□r0, r0, #(\mode^SVC_MODE|PSR_ISETSTATE) \square W(b) \square vector_addrexcptn □msr□spsr_cxsf, r0 □irq_domain_set_info(d, irq, hw, &gic->chip, d->host_data, handle_fasteoi_irq, NULL, NULL);]W(b) □vector_irq lirq set probe(irq); $\square W(b) \square vector fiq$ \square and \square 1r, 1r, #0x0f □irqd_set_single_target(irqd) $THUMB(\square adr \square r0, 1f \square \square)$ ∃break; THUMB ($\Box 1 dr \Box 1r$, [r0, 1r, 1s1 #2] \Box) irq usr □mov□r0, sp]/* Prevents SW retriggers which mess up the ACK/EOI ordering */ $ARM(\Box 1dr \Box 1r, [pc, 1r, 1s1 #2] \Box)$ _irq_usr: □movs□pc, 1r@ branch to handler in SVC mode lirqd_set_handle_enforce_irqctx(irqd); □usr_entry ENDPROC(vector_\name) return 0; kuser_cmpxchg_check IRQ 中断向量表 □get thread info tsk arch/arm/kernel/entry-armv.S □mov□why, #0 □b□ret_to_user_from_irq /* Interrupt dispatcher */ $UNWIND(.fnend \square \square)$ vector_stub irq, IRQ_MODE, 4 ENDPROC(__irq_usr) long irq_usr @ 0 (USR_26 / USR_32)]. long□__irq_invalid□@ 1 (FIQ_26 / FIQ_32) _irq_do_set_handler(struct irq_desc *desc, irq_flow_handler_t □.long□_irq_invalid□@ 2 (IRQ 26 / IRQ 32) /* Interrupt handling */ ndle, int is_chained, const char *name) \square . long \square irg svc \square @ 3 (SVC 26 / SVC 32) \square . macro \square irg handler \square .long \square _irq_invalid \square @ 4 #ifdef CONFIG GENERIC IRQ MULTI HANDLER □.long□_irq_invalid□@ 5 □ldr□r1, =handle_arch_irq \square .long \square _irq_invalid \square @ 6 \square mov \square r0, sp $\square desc->name = name;$ \square .long \square _irq_invalid \square @ 7 □badr□1r, 9997f].long□__irq_invalid□@ 8 $\Box 1 dr \Box pc$, [r1] J.long□__irq_invalid□@ 9 \square .long \square _irq_invalid \square @ a].long□__irq_invalid□@ b].long□__irq_invalid□@ c .long□__irq_invalid□@ d 根据HW interrupt ID<mark>找到IRQ number</mark>,调用asm_do_IRQ irqreturn_t __handle_irq_event_percpu(struct irq_desc *desc, void handle_fasteoi_irq(struct irq_desc *desc) .long□__irq_invalid□@ e unsigned int *flags)].long□__irq_invalid□@ f Struct irq_chip *chip = desc->irq_data.chip; /* Interrupt handling. Preserves r7, r8, r9*/ lirqreturn_t retval = IRQ_NONE; □raw_spin_lock(&desc->lock); unsigned int irq = desc->irq_data.irq; □desc->istate &= ~(IRQS_REPLAY | IRQS_WAITING) .macro□arch irq handler default ch/arm/kernel/irq.c □kstat_incr_irqs_this_cpu(desc); □get irqnr preamble r6, lr asmlinkage void __exception_irq_entry if (desc->istate & IRQS ONESHOT) et_irqnr_and_base r0, r2, r6, lr _asm_do_IRQ(unsigned int(irq, struct pt_regs(*regs)) □movne□r1, sp □irqreturn_t res; \square handle_IRQ(irq, regs); Itrace_irq_handler_entry(irq, action); □ cond unmask eoi irq(desc, chip); \square @ routine called with r0 = irq number □raw_spin_unlock(&desc->lock); \square @ , r1 = struct pt_regs * Itrace /irq handler_exit(irq, action, res); handle_IRQ(unsigned int irq, struct pt_regs *regs) □retval |= res; if (!(chip->flags & IRQCHIP_EOI IF HANDLED)) _handle_domain_irq(NULL, irq, false, regs); ∃return retval; □chip->irq_eoi(&desc->irq_data); \Box raw_spin_unlock(&desc->lock); /* Architectures call this to let the generic IRQ layer handle an interrupt*/ static inline void generic_handle_irq_desc(struct irq_desc *desc) __handle_domain_irq(struct irq_domain *domain, unsigned int hwirq, □□bool lookup, struct pt_regs *regs) irgreturn_t handle_irg_event(struct irg_desc *desc) lstruct pt regs *old regs = set irq regs(regs); irqreturn_t ret; \sqcup int ret = 0; □desc->istate &= ~IRQS_PENDING; lirqd_set(&desc->irq_data, IRQD_IRQ_INPROGRESS); \square irq_enter(); □raw spin unlock(&desc->lock); #ifdef CONFIG IRQ DOMAIN = handle_irq_event_percpu(de if (lookup) $\square \square irq = irq_find_mapping(domain, hwirq);$ t request_threaded_irq<mark>(unsigned int irq, irq_handler_t handler</mark>, raw_spin_lock(&desc->lock); □irq_handler_t thread_fn, unsigned long irqflags, irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS); □if (unlikely(!irq || irq >= nr_irqs)) { □const char *devname, void *dev_id) $\square \square$ ack bad irg(irg); return ret; $\square \square ret = -EINVAL;$ $|\Box$ } else { $\square \square$ generic_handle_irq(irq); int retval; □irq_exit(); if (irq == IRQ_NOTCONNECTED) □return -ENOTCONN; \square set_irq_regs(old_regs); □return ret: if (((irgflags & IRQF SHARED) && !dev id) (!(irqflags & IRQF_SHARED) && (irqflags & IRQF_COND_SUSPEND)) || ((irqflags & IRQF NO SUSPEND) && int generic_handle_irq(unsigned int irq) (irqflags & IRQF_COND_SUSPEND))) irq_desc[NR_IRQS] Istruct irq_desc *desc = irq_to_desc(irq); □return -EINVAL; □struct irq_data *data; lstruct irq_common_data□ irq_common_data; if (!desc) \(\text{data} = \text{irq_desc_get_irq_data(desc)} \); ∃struct irq_data□□irq_data; ☐ if (WARN_ON_ONCE(!in_irq() && handle_enforce_irqctx(data))) □□return -EINVAL; lunsigned int __percpu□*kstat_irqs; □□return -EPERM; lirq_flow_handler_t□handle_irq; if (!irq_settings_can_request(desc) | lstruct irqaction $\square *$ action; $\square / *$ IRQ action list * /WARN_ON(irq_settings_is_per_cpu_devid(desc))) □return -EINVAL; Ireturn 0; \square atomic_t \square \square threads_handled; if (\text{\text{Mandler}}) { □if (!thread fn) □□return -EINVAL; □□handler = irq_default_primary_handler; static inline int __must_check **——** request_irq(unsigned int irq, irq_handler_t handler, unsigned long void□□□*dev_id; flags, const char *name, void *dev) □void __percpu□□*percpu_dev_id; Jaction =\kzalloc(sizeof(struct irqaction), GFP_KERNEL); □struct irqaction□*next; if (!action) return request_threaded_irq(irq, handler, NULL, flags, name, dev); □irq_handler_t□□thread_fn; **←** □□return -ENOMEM; □struct task_struct□*thread; □action->handler = handler; □action->thread_fn = thread_fn; □struct irqaction□*secondary; □unsigned int□□irq; □unsigned int□□flags; □action->flags = irqflags; \square unsigned long \square \square thread_flags; \square action->name = devname; \square unsigned long \square \square thread_mask; □action->dev_id = dev_id; static int __init rtc_init(void) □const char□□*name; □struct proc_dir_entry□*dir; □retval = irq_chip_pm_get(&desc->irq_data); \square if (retval < 0) { irqreturn_t ret = 0; $\square \square$ kfree(action); truct irq_data { □□return retval; regs = (rtc_reg_t *)ioremap(RTC_BASE, sizeof(rtc_reg_t)); lu32□□□mask; printk("rtc_init\n"); / □unsigned int□□irq; □unsigned long□□hwirq; Iretval = __setup_irq(irq, desc, action); set_rtc_alarm(regs); / □struct irq_common_data□*common; □struct irq_chip□□*chip; ∃if (retval) { ret = request_irq(39, ret = request_irq(39, <a □struct irq_domain□*domain; Dirq_chip_pm_put(&desc->irq_data); if (ret == -1) { □void□□□*chip_data; □kfree(action->secondary); printk("request_irq failed!\n"); □□kfree(action); return −1; truct irq_chip { □struct device□*parent_device; #ifdef CONFIG DEBUG SHIRQ FIXME return 0; □const char□*name; if (!retval && (irqflags & IRQF_SHARED)) { □unsigned int□(*irq_startup)(struct irq_data *data); □unsigned long flags; $lvoid \square \square$ (*irq_shutdown) (struct irq_data *data); $lvoid \square \square$ (*irq_enable) (struct irq_data *data); □□disable_irq(irq); void□□(*irq_disable)(struct_irq_data *data); $\square \square local_irq_save(flags);$ $void \square \square$ (*irq_ack) (struct irq_data *data); □ handler(irq, dev_id);

□local_irq_restore(flags)

 \square enable_irq(irq);

□return retval;

lvoid□□(*irq_mask)(struct irq_data *data);

□unsigned long□flags;

 $lvoid \square \square$ (*irq mask ack) (struct irq data *data);

lvoid□□(*irq_unmask)(struct_irq_data *data); □void□□(*irq_eoi)(struct irq_data *data);

tasklet的执行过程 el/softirq.c oid __init softirq_init(void) for <u>each_possible_cpu(cpu)</u> { □per_cpu(tasklet_vec, cpu).tail = &per_cpu(tasklet_vec, cpu).head; □per_cpu(tasklet_hi_vec, cpu).tail = &per_cpu(tasklet_hi_vec, cpu).head; □open softirg(TASKLET SOFTIRQ, tasklet action); Jopen_softirq(HI_SOFTIRQ, tasklet_hi_action); static __latent_entropy void tasklet_action(struct softirq_action *a) \square tasklet action common(a, this cpu ptr(&tasklet vec), TASKLET SOFTIRQ); id tasklet_action_common(struct softirq_action *a, struct tasklet_head *tl_head, □□ unsigned int softirq_nr)]struct tasklet_struct *list; llocal_irq_disable(); $lst = t1 head \rightarrow head;$ _head->head = NULL; 1_head->tail = &tl_head->head; local_irq_enable(); hile (list) { |struct tasklet_struct *t = list; list = list->next; if (tasklet_trylock(t)) if (!atomic_read(&t->count)) if (!test_and_clear_bit(TASKLET_STATE_SCHED,&t->state)) if (t->use_callback) $\exists \Box t \rightarrow callback(t);$ raise_softirq(unsigned int nr) else $\Box \Box t \rightarrow func(t \rightarrow data);$ nsigned long flags; □ tasklet_unlock(t) llocal irq save(flags); continue; □raise softirg irgoff(nr); □local_irq_restore(flags); □tasklet_unlock(t); llocal irg disable() id raise softirq irqoff(unsigned int nr) t->next = NULL; |*tl_head->tail = t; raise softirg irqoff(nr); tl_head->tail = &t->next; (!in_interrupt()) __raise_softirq_irqoff(softirq_nr); □ wakeup softirqd(); local_irq_enable(); __raise_softirq_irqoff(unsigned int nr) ckdep_assert_irqs_disabled(); ce_softirq_raise(nr); open_softirq(int nr, void (*action)(struct softirq_action *)) oftirq_vec[nr].action = action; 软中断的执行过程 nlinkage __visible void __softirq_entry __do_softirq(void) unsigned long end = jiffies + MAX_SOFTIRQ_TIME; unsigned long old_flags = current->flags; int max_restart = MAX_SOFTIRQ_RESTART; l irq_exit(void) struct softirq_action *h; ool in_hardirq; irq_exit_rcu(); _u32 pending; u ira exit(): nt softirq_bit; |lockdep_hardirq_exit(); current->flags &= ~PF_MEMALLOC; id irq_exit_rcu(void) 🛮 🛶 pending = local_softirq_pending(); account_irq_enter_time(current); _local_bh_disable_ip(_RET_IP_, SOFTIRQ_OFFSET); ockdep_hardirq_exit(); n_hardirq = lockdep_softirq_start(); catic inline void __irq_exit_rcu(void) 💆 set_softirq_pending(0); fndef __ARCH_IRQ_EXIT_IRQS_DISABLED ocal_irq_enable(); local_irq_disable(); lockdep_assert_irqs_disabled(); softirq_vec; account irg exit time(current); while ((softirq_bit = ffs(pending))) { preempt_count_sub(HARDIRQ_OFFSET); □unsigned int vec_nr; [(!in_interrupt() && local_softirq_pending()) lint prev_count; linvoke_softirq(); h += softirq_bit - 1 ck_irq_exit(); /ec/nr = h - softirq_vec;]prev_count = preempt_count(); tic inline void invoke_softirq(void) Zkstat_incr_softirgs_this_cpu(vec_nr); (ksoftirqd_running(local_softirq_pending())) ltrace_softirq_entry(vec_nr);]h/>action(h); Itrace_softirq_exit(vec_nr); ef CONFIG HAVE IRQ EXIT ON IRQ STACK __do_softirq(); □pr_err("huh, entered softirq %u %s %p with preempt_count %08x, exited with %08x?\n", do softirg own stack(); vec nr, softirg to name[vec nr], h->action, prev_count, preempt_count());]□preempt_count_set(prev_count); \square wakeup_softirqd(); pending >>= softirq_bit; (_this_cpu_read(ksoftirqd) == current) cu_softirq_qs(); .ocal_irq_disable(); pending = local_softirq_pending(); if (time_before(jiffies, end) && !need_resched() && --max_restart) □□goto restart; □wakeup_softirqd(); lockdep_softirq_end(in_hardirq); iccount_irq_exit_time(current); local bh enable (SOFTIRQ OFFSET); □WARN ON ONCE(in interrupt()); current_restore_flags(old_flags, PF_MEMALLOC);

workqueue工作队列工作流程

struct work_struct□unbound_release_work;

struct rcu_head□rcu;

schedule work (struct work_struct *work) turn queue work(system wq, work); queue work (struct workqueue_struct *wq, struct work_struct *work) urn queue work on (WORK CPU UNBOUND, wq, work); queue work on (int cpu, struct workqueue_struct *wq, struct work_struct *work) l ret = false; igned long flags; al_irq_save(flags); (!test and set bit(WORK STRUCT PENDING BIT, work data bits(work))) __queue_work(cpu, wq, work); ret = true; al_irq_restore(flags); urn ret; queue work (int cpu, struct workqueue_struct *wq, struct work_struct *work) ict pool_workqueue *pwq; ruct worker_pool *last_pool; ict list_head *worklist; igned int work_flags; igned int req_cpu = cpu; ekdep_assert_irqs_disabled() bug work activate(work); (unlikely(wq->flags & WQ DRAINING) && WARN_ON_ONCE(!is_chained_work(wq))) _read_lock(); (wq->flags & WQ_UNBOUND) { f (reg cpu == WORK CPU UNBOUND) $\exists cpu = wq select unbound cpu(raw smp processor id());$ lpwq = unbound_pwq_by_node(wq, cpu_to_node(cpu)); f (reg cpu == WORK CPU UNBOUND) $\exists cpu = raw_smp_processor_id();$ lpwq = per_cpu_ptr(wq->cpu_pwqs, cpu); st pool = get work pool(work); (last pool && last pool != pwq->pool) truct worker *worker; raw_spin_lock(&last_pool->lock); orker = find_worker_executing_work(last_pool, work); f (worker && worker->current_pwq->wq == wq) { law spin unlock(&last pool->lock); lraw spin lock(&pwq->pool->lock); aw_spin_lock(&pwq->pool->lock); f (wq->flags & WQ_UNBOUND) { langle = langle ∃goto retry; lWARN_ONCE(true, "workqueue: per-cpu pwq for %s on cpu%d has 0 wq->name, cpu); ce_workqueue_queue_work(req_cpu, pwq, work); owq->nr in flight[pwq->work color]++; vork_flags = work_color_to_flags(pwq->work_color); (likely(pwq->nr_active < pwq->max_active)) { race_workqueue_activate_work(work); vq->nr_active++; orklist = &pwq->pool->worklist; f (list empty(worklist))]pwq->pool->watchdog_ts = jiffies; rork_flags |= WORK_STRUCT_DELAYED; orklist = &pwq->delayed_works; ert_work(pwq, work, worklist, work_flags); aw_spin_unlock(&pwq->pool->lock); u read unlock(); _queue_work(int cpu, struct workqueue_struct *wq, struct work_struct *work) ruct worker_pool { raw_spinlock_t□□lock;□/* the pool lock */ ct pool_workqueue { $\operatorname{int} \square \square \square \operatorname{cpu}; \square / * I$: the associated cpu $* / \square$ struct worker_pool = *pool; = /* 1. the associated pool */ $int \square \square \square node; \square/* I: the associated node ID */$ struct workqueue_struct *wq;□ int□□□id;□/* I: pool\ID */ int□work_color; □/* L: current color */
int□flush_color;□/* L: flushing color */ unsigned int□□flags;□/* X: flags */ unsigned long□□watchdog_ts;/* L: watchdog timestamp */ int□refcnt;□□/* L: reference count */ int□nr_in_flight[WORK_NR_COLORS];/* L: nr of in_flight works */ struct list_head□worklist;□/* L: list of pending works */内核线程一直执行这里的 int□nr_active;□ /* L: nr of active works */ int□max_active; □ /* L: max active works */ struct list_head□delayed_works; /* L: delayed works */ int□□□nr_workers;□/* L: total number of workers */ struct list_head□pwqs_node;□ /* WR: node on wq->pwqs */ struct list_head□mayday_node; /* MD: node on wq->maydays */

int□□□nr_idle;□/* L: currently idle workers */ struct list_head□idle_list;□/* X: list of idle workers */ struct timer_list□idle_timer; □/* L: worker idle timeout */ struct timer_list□mayday_timer;□/* L: SOS timer for workers */ /* a workers is either on busy_hash or idle_list, or the manager */ DECLARE_HASHTABLE(busy_hash, BUSY_WORKER_HASH_ORDER); □□/* L: hash of busy workers */ struct worker□□*manager;□/* L: purely informational */ struct list_head□workers;□/* A: attached workers */ struct completion□*detach_completion; /* all workers detached */ struct ida□□worker_ida;□/* worker IDs for task name */ struct workqueue_attrs□*attrs;□/* I: worker attributes */ struct hlist_node□hash_node;□/* PL: unbound_pool_hash_node */ int□□□refcnt;□/* PL: refcnt for unbound pools */ atomic_t□□nr_running ____cacheline_aligned_in_smp; struct rcu_head□□rcu;

本文档是Linux内核编程04期:中断,的配套文档 结合视频教程、代码、PPT文档学习,效果更好。

视频地址: https://wanglitao.taobao.com



