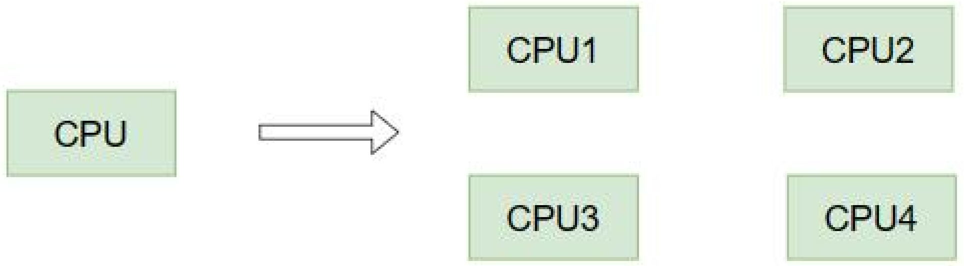
RROS SMP多核机制

Jiajun Du

1. 时钟子系统
    - 初始化：
        - rros_clock_init
        - rros_enable_tick
    - inband和oob中断处理过程
    - remote tick的触发原因
        - 定时：timer不在当前CPU上从而引起remote tick
        - 计时：clock_settime系统调用重置系统时间从而引起remote tick



2. 调度子系统
    - 创建进程：kthread_run 和 pin_to_initial_cpu
    - irq：RESCHEDULE_OOB_IPI
3. 临界区隔离方法：
    - 开关中断
    - 开关抢占
    - 锁

时钟子系统

时钟子系统相关的RROS data structures

```
You, 4个月前 | 3 authors (Li Hongyu and others) | 1 implementation
pub struct RrosClock {
    resolution: KtimeT,    clock的精度
    gravity: RrosClockGravity,    三个超参数，系统响应时钟事件的基本时延
    name: &'static CStr,
    flags: i32,
    ops: RrosClockOps,    operations
    timerdata: *mut RrosTimerbase, clock通过alloc_percpu分配RrosTimerbase得到的地址
    master: *mut RrosClock,    当前clock所依赖的时钟：realtime依赖于mono clock
    offset: KtimeT,    当前clock和master clock之间的时间差值
    next: *mut ListHead,    通过该节点连接在全局clock_list链表之上
    element: Option<Rc<RefCell<RrosElement>>>,
    dispose: Option<fn(&mut RrosClock)>,
    #[cfg(CONFIG_SMP)]
    pub affinity: Option<cpumask::CpumaskT>,    CPU亲和性
}
```

```
Li Hongyu, 5个月前 | 1 author (Li Hongyu) | 1 implementation
pub struct RrosClockGravity {
    irq: KtimeT,            系统响应时钟事件的基本时延
    kernel: KtimeT,
    user: KtimeT,
}
```

```
Li Hongyu, 5个月前 | 1 author (Li Hongyu) | 0 implementations
pub struct RrosTimerbase {    每个CPU上都被alloc的定时器管理链表
    pub lock: SpinLock<i32>,
    pub q: List<Arc<SpinLock<RrosTimer>>>,
}
```

时钟子系统相关的RROS data structures

You, 4个月前 | 3 authors (Li Hongyu and others) | 1 implementation

```
pub struct RrosTimer {
    clock: *mut RrosClock,    timer所属的clock
    date: KtimeT,    timer下次需要被触发的时间
    //adjlink: list_head,// Used when adjusting
    status: i32,
    pub interval: KtimeT, /* 0 == oneshot */当定时器是periodic时，表示两个tick之间的间隔时间
    pub start_date: KtimeT,    定时器开始触发的时间点
    pexpect_ticks: u64, /* periodic release date */    定时器本应该被触发的次数
    periodic_ticks: u64,    定时器实际上被触发的次数
    base: *mut RrosTimerbase,    定时器所在CPU上的定时器管理链表
    handler: fn(*mut RrosTimer),    定时器handler函数，等于timerfd_handler
    name: &'static CStr,
    #[cfg(CONFIG_RROS_RUNSTATS)]
    scheduled: RrosCounter,
    #[cfg(CONFIG_RROS_RUNSTATS)]
    fired: RrosCounter,
    #[cfg(CONFIG_SMP)]
    rq: *mut RrosRq,    定时器所属的CPU的运行队列
    pub thread: Option<Arc<SpinLock<RrosThread>>>,
}
```

Wang xinge, 5个月前 | 1 author (Wang xinge) | 1 implementation

```
pub struct RrosTimerFd {
    timer: Arc<SpinLock<RrosTimer>>,    定时器本身
    readers: RrosWaitQueue,    timer未被触发时，thread调用timerfd_oob_read就会被放到该链表上
    poll_head: RrosPollHead,    通过调用poll被阻塞的thread
    efile: RrosFile,    将定时器和文件关联
    ticked: bool,    标志定时器是否已经被触发
}
```

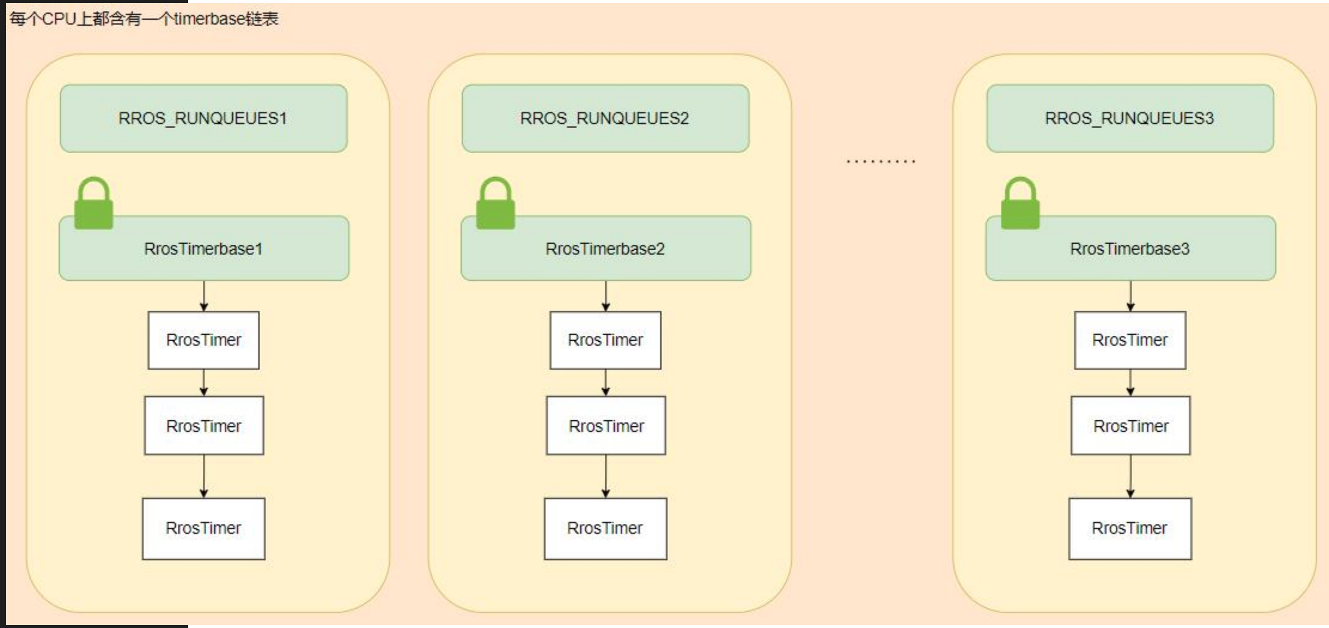
时钟初始化：rros_clock_init

RROS中含有两个clock：
1. RROS_MONO_CLOCK
2. RROS_REALTIME_CLOCK：附属在RROS_MONO_CLOCK上

```rust
pub fn rros_clock_init() -> Result<usize> {
    let pinned: Pin<&mut SpinLock<i32>> = unsafe { Pin::new_unchecked(pointer: &mut CLOCKLIST_LOCK) };
    spinlock_init!(pinned, "CLOCKLIST_LOCK");
    unsafe {
        RROS_MONO_CLOCK.reset_gravity();
        RROS_REALTIME_CLOCK.reset_gravity();

        let mut element: RrosElement = RrosElement::new()?;
        element.pointer = &mut RROS_MONO_CLOCK as *mut _ as *mut u8;
        RROS_MONO_CLOCK.element = Some(Rc::try_new(RefCell::new(element)).unwrap());
        let mut element: RrosElement = RrosElement::new()?;
        element.pointer = &mut RROS_REALTIME_CLOCK as *mut _ as *mut u8;
        RROS_REALTIME_CLOCK.element = Some(Rc::try_new(RefCell::new(element)).unwrap());

        rros_init_clock(&mut RROS_MONO_CLOCK, affinity: &RROS_OOB_CPUS)?;
    }
    let ret: Result<usize, Error> = unsafe { rros_init_slave_clock(&mut RROS_REALTIME_CLOCK, master: &mut RROS_MONO_CLOCK) };
    if let Err(_) = ret {
        //rros_put_element(&rros_mono_clock.element);
    }
    // pr_debug!("clock init success!");
    Ok(0)
}
```

时钟初始化：rros_init_clock

```rust
fn rros_init_clock(clock: &mut RrosClock, affinity: &cpumask::CpumaskT) -> Result<usize> {
    premmpt::running_inband()?;

    #[cfg(CONFIG_SMP)]
    {
        if clock.affinity.is_none() {
            clock.affinity = Some(cpumask::CpumaskT::from_int(0));
        }
        let clock_affinity: &mut CpumaskT = clock.affinity.as_mut().unwrap();
        if affinity.cpumask_empty().is_ok() {
            clock_affinity.cpumask_clear();
            clock_affinity.cpumask_set_cpu(unsafe { RROS_OOB_CPUS.cpumask_first() as u32 });
        } else {
            clock_affinity.cpumask_and(src1: affinity, src2: unsafe { &RROS_OOB_CPUS });
            if clock_affinity.cpumask_empty().is_ok() {
                return Err(Error::EINVAL);
            }
        }
    }

    // 8 byte alignment
    let tmb: *mut RrosTimerbase = percpu::alloc_per_cpu(
        size: size_of::<RrosTimerbase>() as usize,
        align: align_of::<RrosTimerbase>() as usize,
    ) as *mut RrosTimerbase;
    if tmb == 0 as *mut RrosTimerbase {
        return Err(kernel::Error::ENOMEM);
    }
    clock.timerdata = tmb;

    for cpu: <OnlineCpusIndexIter as IntoIterator>::It… in online_cpus() {
        let mut tmb: *mut RrosTimerbase = rros_percpu_timers(clock, cpu as i32);
        unsafe { raw_spin_lock_init(&mut (*tmb).lock); }
    }

    clock.offset = 0;
    let ret: Result<usize, Error> = init_clock(clock as *mut RrosClock, master: clock as *mut RrosClock);
    if let Err(_) = ret {
        percpu::free_per_cpu(pdata: clock.get_timerdata_addr() as *mut u8);
        return ret;
    }
    Ok(0)
} fn rros_init_clock
```



tick初始化

过程比较复杂，此处主要关注：

1. real device和proxy device上的event_handler函数
2. inband和oob两种irq信号分别是如何处理的
3. RROS申请得到的irq信号如何被触发、调用过程是怎样的、如何被处理

```c
static unsigned int proxy_tick_irq;

static DEFINE_MUTEX(proxy_mutex);

static DEFINE_PER_CPU(struct clock_proxy_device, proxy_tick_device);
```

```c
struct tick_device {
    struct clock_event_device *evtdev;
    enum tick_device_mode mode;
};
```

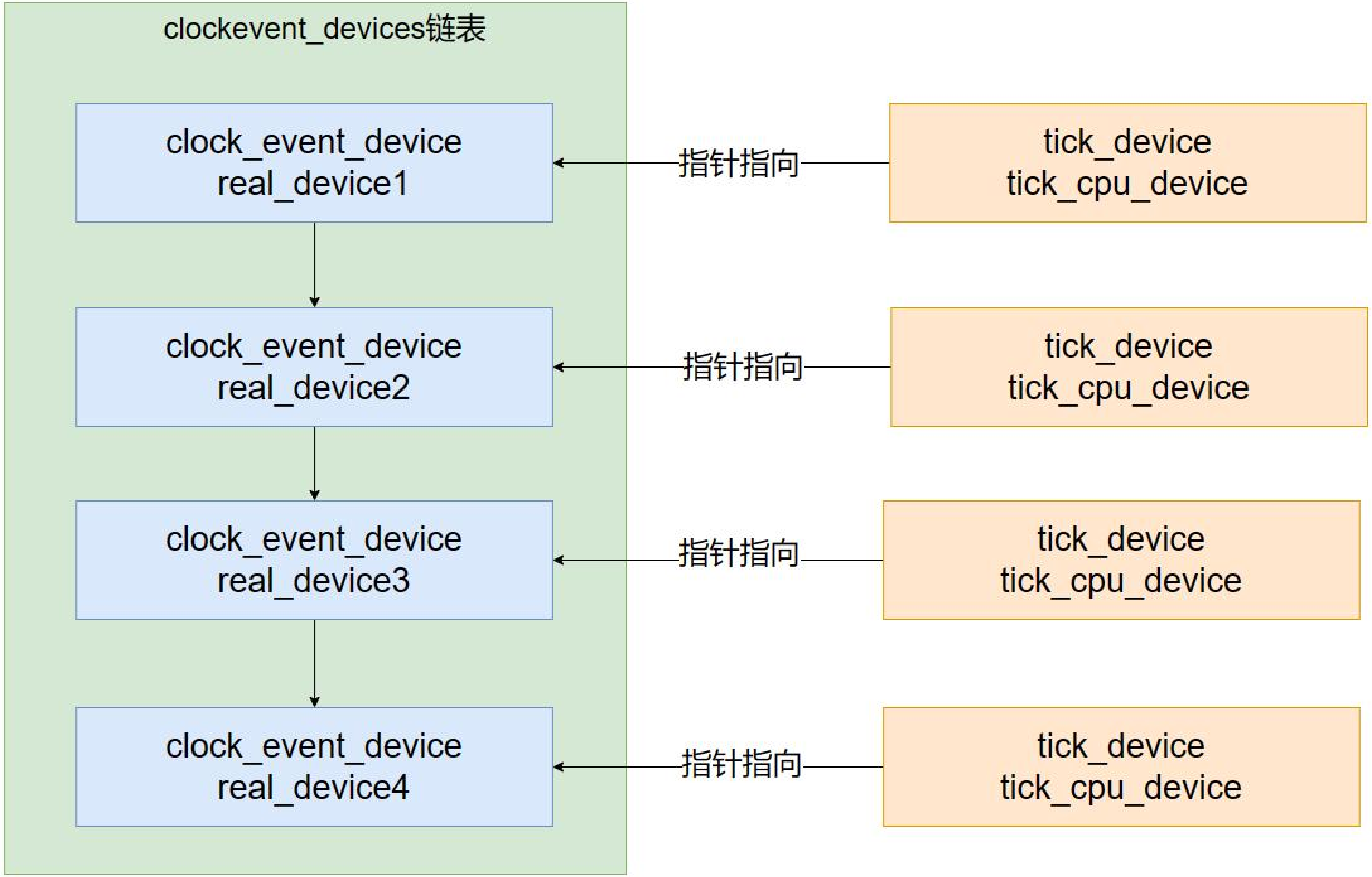
```c
DECLARE_PER_CPU(struct tick_device, tick_cpu_device);
```

```c
/* The registered clock event devices */
static LIST_HEAD(clockevent_devices);
static LIST_HEAD(clockevents_released);
/* Protection for the above */
static DEFINE_RAW_SPINLOCK(clockevents_lock);
/* Protection for unbind operations */
static DEFINE_MUTEX(clockevents_mutex);
```

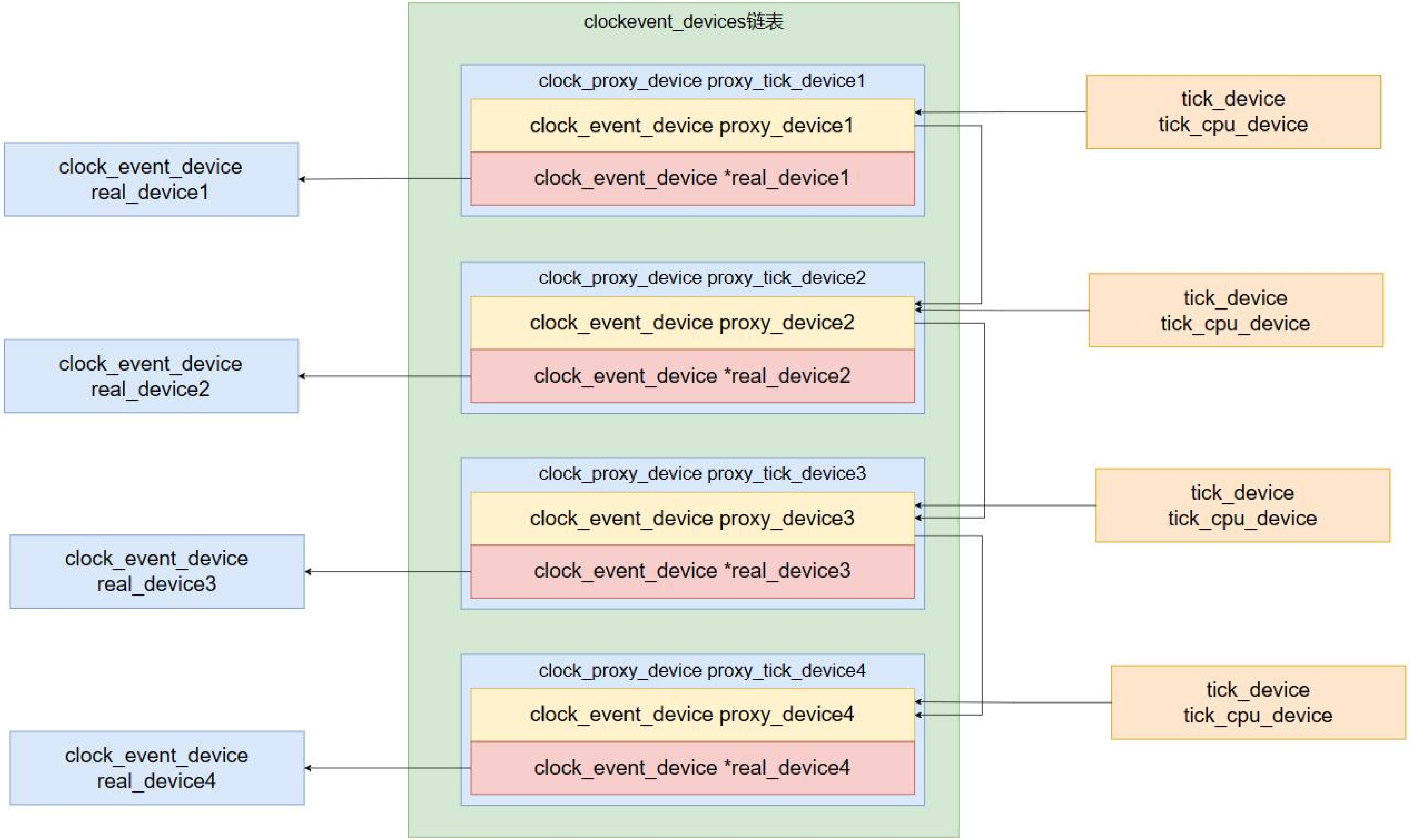
```c
struct clock_proxy_device {
        struct clock_event_device proxy_device;
        struct clock_event_device *real_device;
        void (*handle_oob_event)(struct clock_event_device *dev);
        /* Internal data - don't depend on this. */
        void (*__setup_handler)(struct clock_proxy_device *dev);
        void (*__original_handler)(struct clock_event_device *dev);
};
```

```c
static DEFINE_PER_CPU(struct clock_proxy_device *, proxy_device);
```

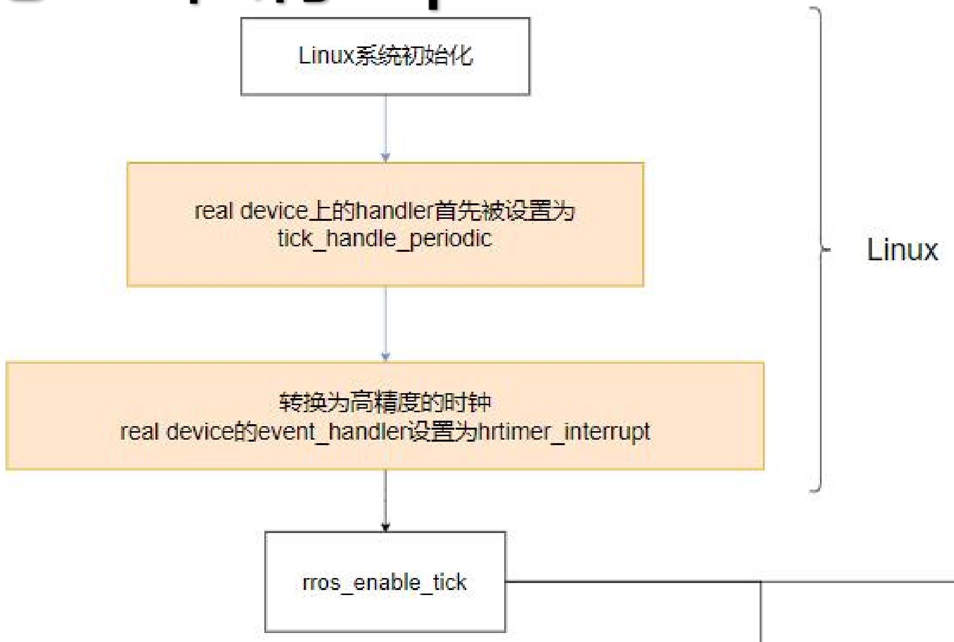
tick初始化：原始的real device



tick初始化：real device和proxy device的关系



tick初始化：申请irq



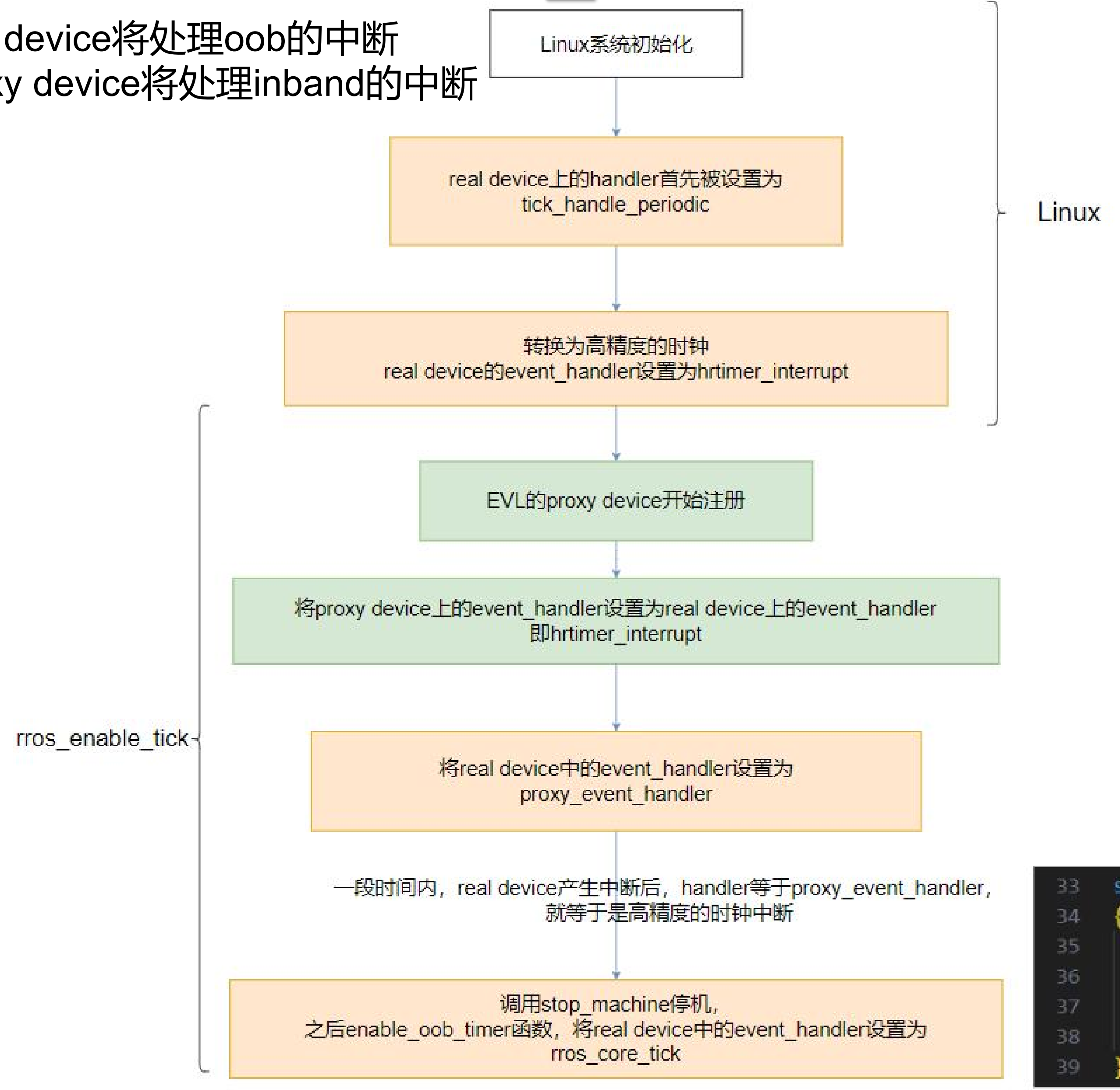

```c
	/*
	 * We may be running a SMP kernel on a uniprocessor machine
	 * whose interrupt controller provides no IPI: attempt to hook
	 * the timer IPI only if the hardware can support multiple
	 * CPUs.
	 */
	if (IS_ENABLED(CONFIG_SMP) && num_possible_cpus() > 1) {
		ret = __request_percpu_irq(TIMER_OOB_IPI,
					clock_ipi_handler,
					IRQF_OOB, "EVL timer IPI",
					&evl_machine_cpudata);
		if (ret)
			return ret;
	}
```

```c
int tick_install_proxy(void (*setup_proxy)(struct clock_proxy_device *dev),
		const struct cpumask *cpumask)
{
	struct proxy_install_arg arg;
	int ret, sirq;

	mutex_lock(&proxy_mutex);

	ret = -EAGAIN;
	if (proxy_tick_irq)
		goto out;

	sirq = irq_create_direct_mapping(synthetic_irq_domain);
	if (WARN_ON(sirq == 0))
		goto out;

	ret = __request_percpu_irq(sirq, proxy_irq_handler,
				IRQF_NO_THREAD, /* no IRQF_TIMER here. */
				"proxy tick",
				&proxy_tick_device);
	if (WARN_ON(ret)) {
		irq_dispose_mapping(sirq);
		goto out;
	}

	proxy_tick_irq = sirq;
	barrier();
```

```c
#ifdef CONFIG_SMP
static irqreturn_t clock_ipi_handler(int irq, void *dev_id)
{
	evl_core_tick(NULL);

	return IRQ_HANDLED;
}
#else
#define clock_ipi_handler	NULL
#endif
```

tick初始化：event_handler的变化

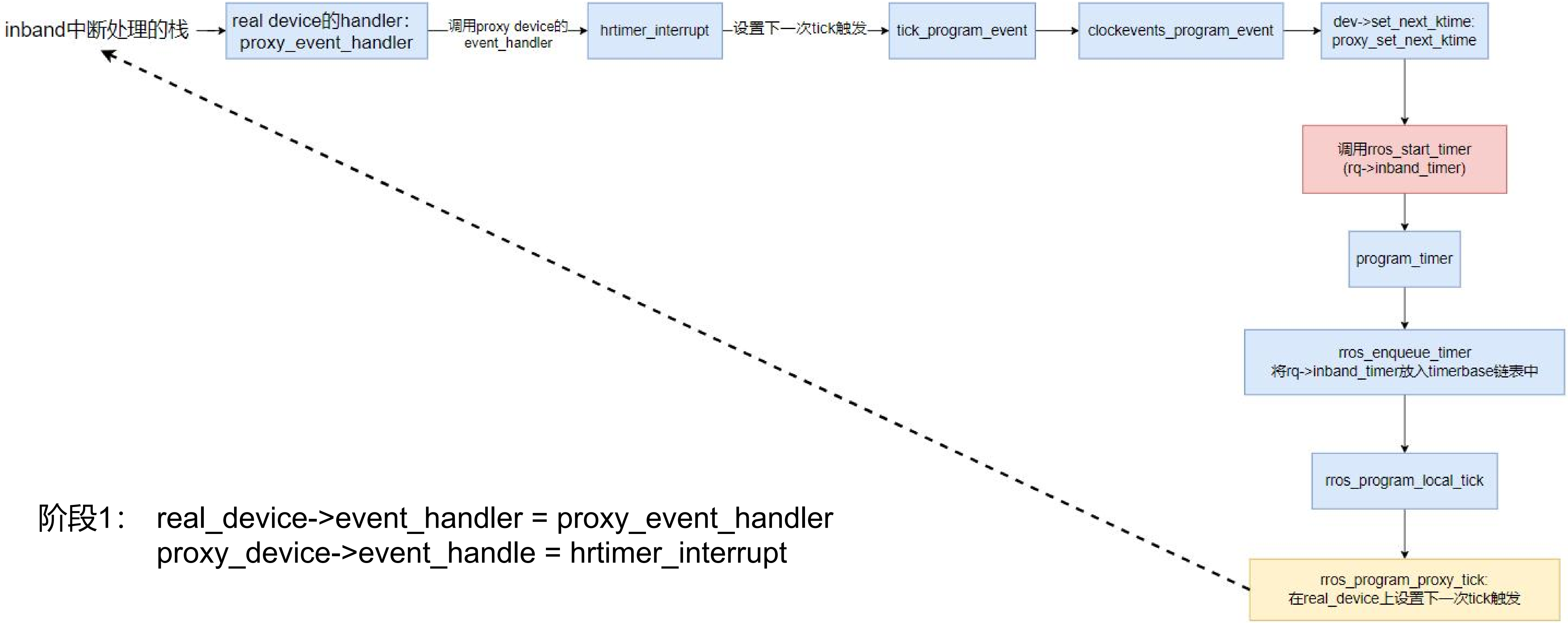
- real device将处理oob的中断
- proxy device将处理inband的中断




```c
static irqreturn_t proxy_irq_handler(int sirq, void *dev_id)
{
	struct clock_event_device *evt;

	/*
	 * Tricky: we may end up running this in-band IRQ handler
	 * because tick_notify_proxy() was posted either:
	 *
	 * - from the out-of-band stage via ->handle_oob_event() for
	 * emulating an in-band tick.  In this case, the active tick
	 * device for the in-band timing core is the proxy device,
	 * whose event handler is still the same than the real tick
	 * device's.
	 *
	 * - directly by the clock chip driver on the local CPU via
	 * clockevents_handle_event(), for propagating a tick to the
	 * in-band stage nobody from the out-of-band stage is
	 * interested on i.e. no proxy device was registered on the
	 * receiving CPU, which was excluded from @cpumask in the call
	 * to tick_install_proxy(). In this case, the active tick
	 * device for the in-band timing core is a real clock event
	 * device.
	 *
	 * In both cases, we are running on the in-band stage, and we
	 * should fire the event handler of the currently active tick
	 * device for the in-band timing core.
	 */
	evt = raw_cpu_ptr(&tick_cpu_device)->evtdev;
	evt->event_handler(evt);

	return IRQ_HANDLED;
}
```

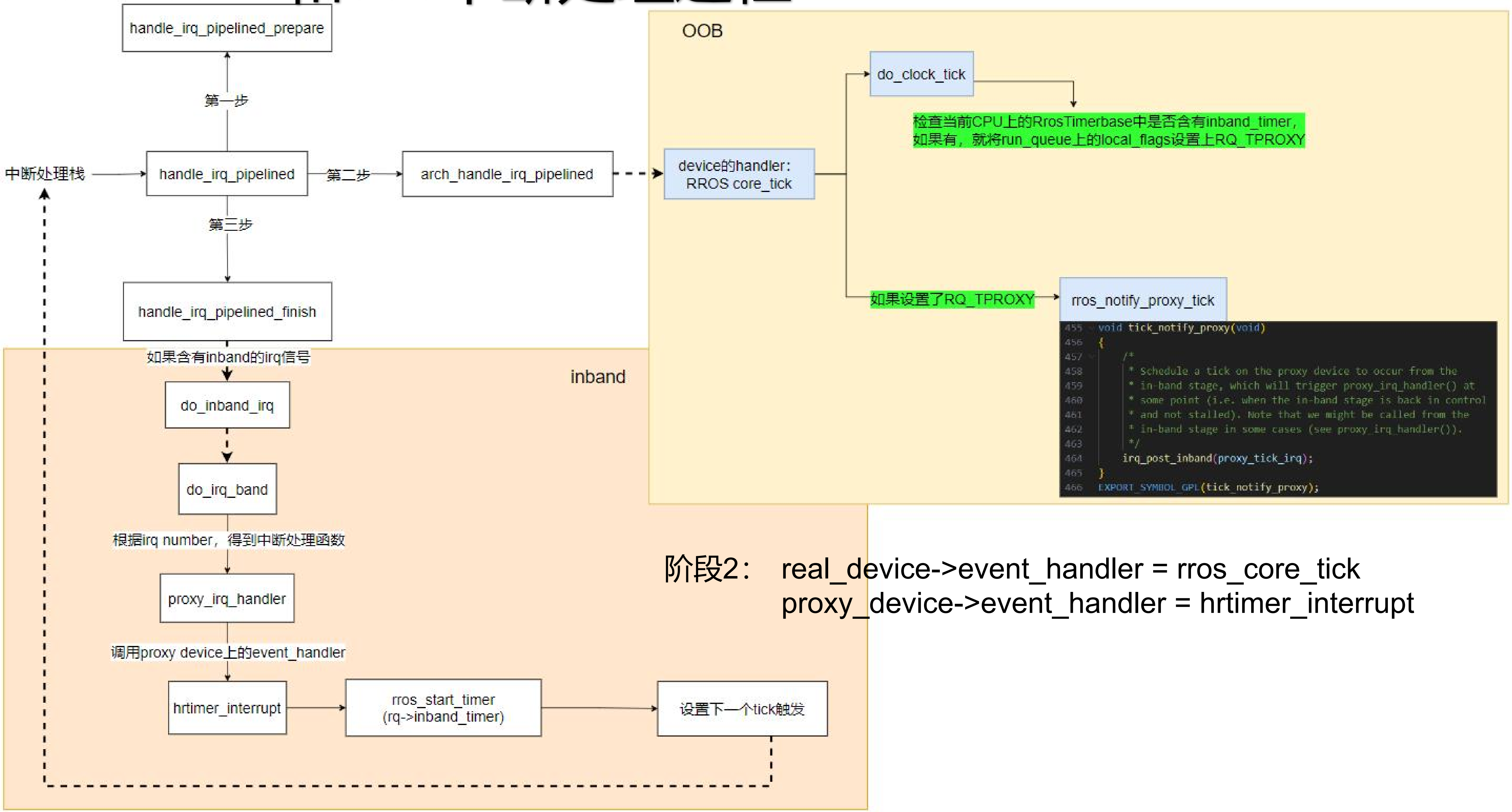
```c
static void proxy_event_handler(struct clock_event_device *real_dev)
{
	struct clock_proxy_device *dev = raw_cpu_ptr(&proxy_tick_device);
	struct clock_event_device *proxy_dev = &dev->proxy_device;

	proxy_dev->event_handler(proxy_dev);
}
```

tick：inband和oob中断处理过程



阶段1： real_device->event_handler = proxy_event_handler
proxy_device->event_handle = hrtimer_interrupt

tick：inband和oob中断处理过程



阶段2： real_device->event_handler = rros_core_tick
proxy_device->event_handler = hrtimer_interrupt

tick：remote tick信号的触发原因

- 根本原因：定时器不在当前CPU上
- 触发时机：
  1. 定时器不在当前CPU上，但是当前CPU上的进程需要等待定时器结束
  2. timekeeping时钟同步，从而调整该时钟上的定时器的下一个date

Wang xinge, 5个月前 | 1 author (Wang xinge) | 1 implementation

```
pub struct RrosTimerFd {
    timer: Arc<SpinLock<RrosTimer>>,   定时器本身
    readers: RrosWaitQueue,   timer未被触发时，thread调用timerfd_oob_read就会被放到该链表上
    poll_head: RrosPollHead,   通过调用poll被阻塞的thread
    efile: RrosFile,   将定时器和文件关联
    ticked: bool,   标志定时器是否已经被触发
}
```

tick：定时器不在当前CPU上的原因

- 用户态创建timer：evl_new_timer(EVL_CLOCK_MONOTONIC)
  - 用户态：ioctl(EVL_CLKIOC_NEW_TIMER)
    - clock_ioctl
      - new_timerfd

```c
#define evl_init_timer_on_rq(__timer, __clock, __handler, __rq, __flags) \
	__evl_init_timer(__timer, __clock, __handler,		\
			__rq, #__handler, __flags)
```

```c
void __evl_init_timer(struct evl_timer *timer,
		struct evl_clock *clock,
		void (*handler)(struct evl_timer *timer),
		struct evl_rq *rq,
		const char *name,
		int flags)
{
	int cpu;

	timer->clock = clock;
	evl_tdate(timer) = EVL_INFINITE;
	timer->status = EVL_TIMER_DEQUEUED|(flags & EVL_TIMER_INIT_MASK);
	timer->handler = handler;
	timer->interval = EVL_INFINITE;

	/*
	 * Set the timer affinity to the CPU rq is on if given, or the
	 * first CPU which may run EVL threads otherwise.
	 */
	cpu = rq ?
		get_clock_cpu(clock->master, evl_rq_cpu(rq)) :
		cpumask_first(&evl_cpu_affinity);
#ifdef CONFIG_SMP
	timer->rq = evl_cpu_rq(cpu);
#endif
	timer->base = evl_percpu_timers(clock, cpu);
	timer->clock = clock;
	timer->name = name ?: "<timer>";
	evl_reset_timer_stats(timer);
}
```

传入的rq=NULL，所以创建在第一个oob CPU上

```c
static int new_timerfd(struct evl_clock *clock)
{
	struct evl_timerfd *timerfd;
	struct file *filp;
	int ret, fd;

	timerfd = kzalloc(sizeof(*timerfd), GFP_KERNEL);
	if (timerfd == NULL)
		return -ENOMEM;

	filp = anon_inode_getfile("[evl-timerfd]", &timerfd_fops,
				timerfd, O_RDWR|O_CLOEXEC);
	if (IS_ERR(filp)) {
		kfree(timerfd);
		return PTR_ERR(filp);
	}

	/*
	 * From that point, timerfd_release() might be called for
	 * cleaning up on error via filp_close(). So initialize
	 * everything we need for a graceful cleanup.
	 */
	evl_get_element(&clock->element);
	evl_init_timer_on_rq(&timerfd->timer, clock, timerfd_handler,
			NULL, EVL_TIMER_UGRAVITY);
	evl_init_wait(&timerfd->readers, clock, EVL_WAIT_PRIO);
	evl_init_poll_head(&timerfd->poll_head);

	ret = evl_open_file(&timerfd->efile, filp);
	if (ret)
		goto fail_open;

	fd = get_unused_fd_flags(O_RDWR|O_CLOEXEC);
	if (fd < 0) {
		ret = fd;
		goto fail_getfd;
	}

	fd_install(fd, filp);

	return fd;
```

tick：1. 进程设置定时器触发时间，引发remote tick

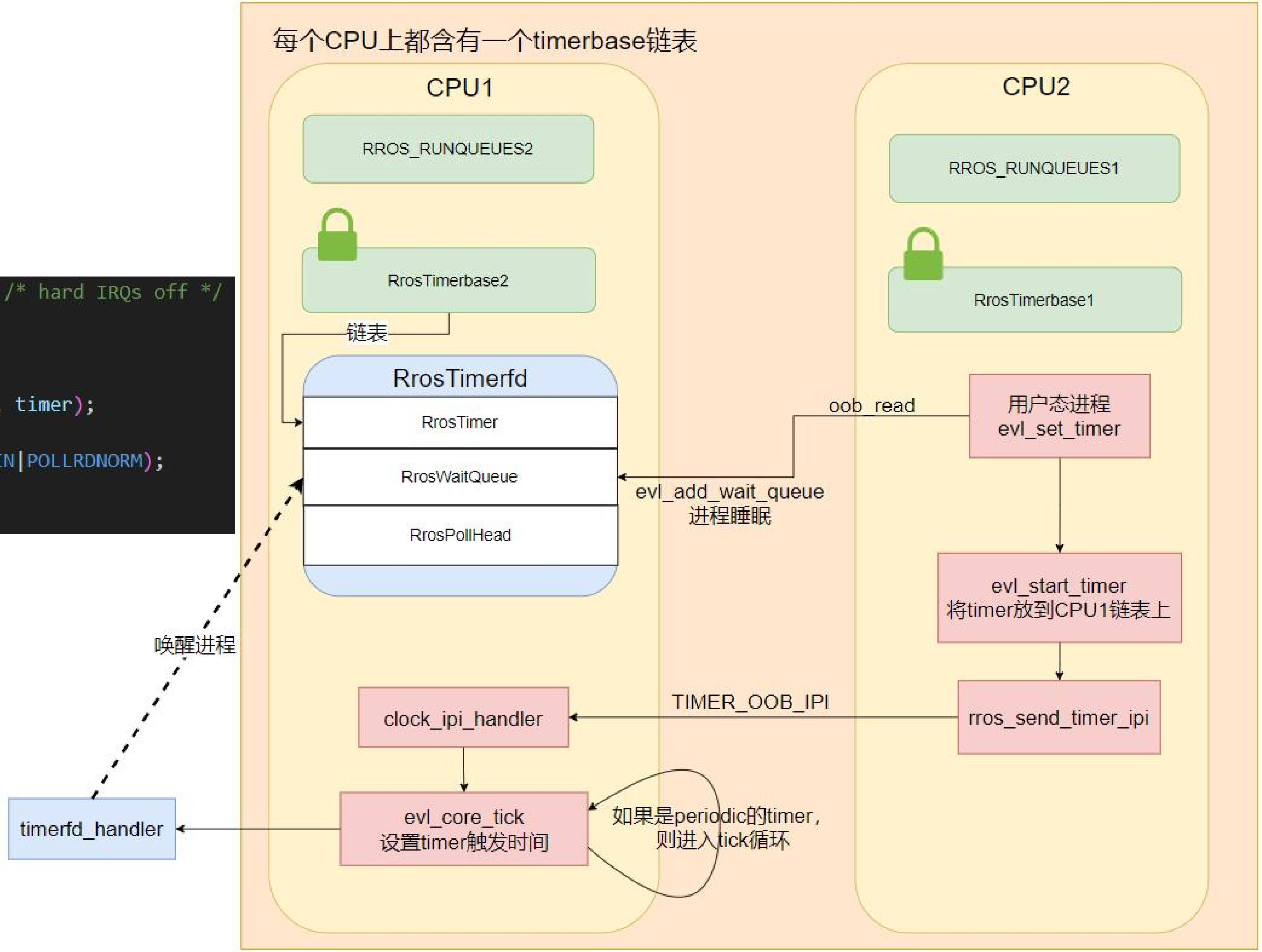
- 用户态：evl_set_timer
  - 用户态：__evl_common_ioctl(EVL_TFDIOC_SET)
    - timerfd_common_ioctl
      - set_timerfd
        - set_timer_value
          - rros_start_timer
            - program_timer
              - 如果timer在当前CPU上，则rros_program_local_tick
              - 如果timer不在当前CPU上，则rros_program_remote_tick
                - 调用rros_send_timer_ipi

```c
#ifdef CONFIG_SMP
static irqreturn_t clock_ipi_handler(int irq, void *dev_id)
{
    evl_core_tick(NULL);

    return IRQ_HANDLED;
}
#else
#define clock_ipi_handler  NULL
#endif
```

```rust
#[cfg(CONFIG_SMP)]
pub fn rros_send_timer_ipi(_clock: &RrosClock, rq: *mut RrosRq) {
    irq_send_oob_ipi(ipi: irq_get_timer_oob_ipi(), cpumask: cpumask::CpumaskT::cpumask_of(cpu: rros_rq_cpu(rq) as u32));
}                         TIMER_OOB_IPI

```

tick： 1. 进程设置定时器触发时间

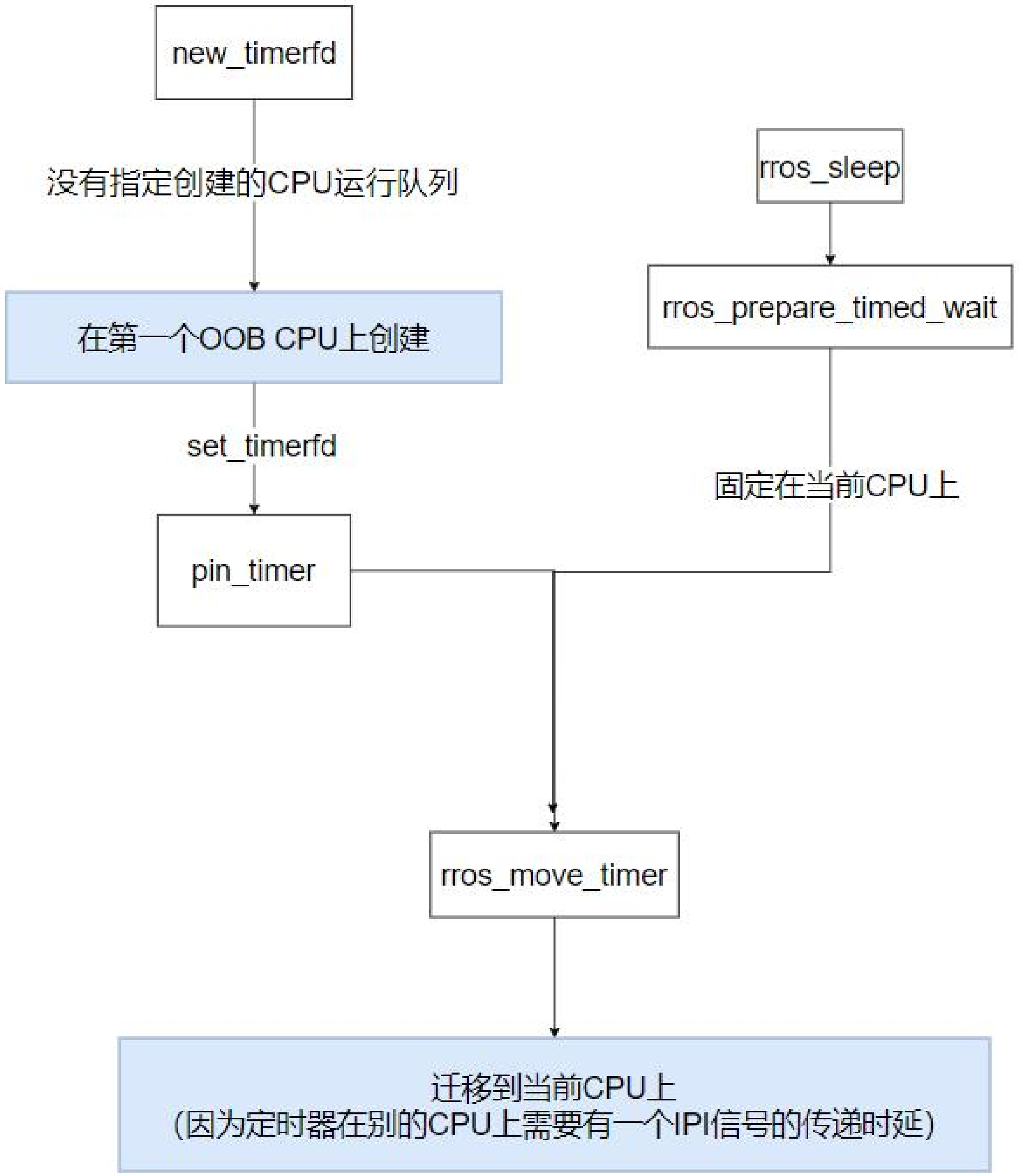
```c
static void timerfd_handler(struct evl_timer *timer) /* hard IRQs off */
{
    struct evl_timerfd *timerfd;

    timerfd = container_of(timer, struct evl_timerfd, timer);
    timerfd->ticked = true;
    evl_signal_poll_events(&timerfd->poll_head, POLLIN|POLLRDNORM);
    evl_flush_wait(&timerfd->readers, 0);
}
```



tick： 1. pin_timer

由于有pin_timer和rros_move_timer，所以一般不会引发remote tick

```rust
fn set_timerfd(
    timerfd: &RrosTimerFd,
    value: &mut Itimerspec64,
    ovalue: Itimerspec64,
) -> Result<i32> {
    get_timer_value(timer: timerfd.timer.clone(), value);

    if rros_current() != core::ptr::null_mut() {
        pin_timer(timerfd.timer.clone());
    }

    set_timer_value(timer: timerfd.timer.clone(), value)
}
```



tick：2. clock_settime系统调用引发remote tick

在用户态触发 clock_settime 系统调用更改系统时间时，每次更改结束都会重新将所有定时器的下一次触发时间进行修改， 修改后就会导致需要重新设置下一次时钟中断的触发，因此对于远程定时器来说也会通过 TIMER_OOB_IPI 信号完成通知。

RROS还没有实现，EVL中是：

```
(gdb) bt
#0  inband_clock_was_set () at kernel/dovetail.c:398
#1  0xffffffc010173990 in clock_was_set () at kernel/time/hrtimer.c:876
#2  0xffffffc0101777bc in do_settimeofday64 (ts=0xffffffc0116dbdd8)
    at kernel/time/timekeeping.c:1327
#3  0xffffffc01016dbdc in do_sys_settimeofday64 (tv=0xffffffc0116dbdd8, tz=0x0)
    at kernel/time/time.c:195
#4  0xffffffc01018115c in posix_clock_realtime_set (which_clock=<optimized out>,
    tp=0x169169e265d18f20) at kernel/time/posix-timers.c:185
#5  0xffffffc01017fbc0 in __do_sys_clock_settime (which_clock=0, tp=<optimized out>)
    at kernel/time/posix-timers.c:1079
#6  __se_sys_clock_settime (which_clock=0, tp=<optimized out>)
    at kernel/time/posix-timers.c:1067
#7  __arm64_sys_clock_settime (regs=<optimized out>) at kernel/time/posix-timers.c:1067
#8  0xffffffc01002a264 in __invoke_syscall (regs=0xffffffc0116dbeb0,
    syscall_fn=0xffffffc0115c7bd8 <tk_core+152>) at arch/arm64/kernel/syscall.c:39
#9  invoke_syscall (regs=0xffffffc0116dbeb0, scno=0, sc_nr=447, syscall_table=0x100)
    at arch/arm64/kernel/syscall.c:53
#10 0xffffffc01002a170 in el0_svc_common (regs=0xffffffc0116dbeb0, scno=112, sc_nr=447,
    syscall_table=0xffffffc010cd0ae8 <sys_call_table>) at arch/arm64/kernel/syscall.c:153
#11 0xffffffc01002a058 in do_el0_svc (regs=0x169169e265d18f20)
    at arch/arm64/kernel/syscall.c:193
#12 0xffffffc010ca4494 in el0_svc (regs=0xffffffc0116dbeb0)
    at arch/arm64/kernel/entry-common.c:528
#13 0xffffffc010ca4408 in el0_sync_handler (regs=0x169169e265d18f20)
    at arch/arm64/kernel/entry-common.c:544
#14 0xffffffc010011ed8 in el0_sync () at arch/arm64/kernel/entry.S:777
Backtrace stopped: Cannot access memory at address 0xffffffc0116dc0c8
(gdb)
```

```
void inband_clock_was_set(void)
{                                                   EVL
    struct evl_clock *clock;

    if (!evl_is_enabled())
        return;

    mutex_lock(&clocklist_lock);

    list_for_each_entry(clock, &clock_list, next) {
        if (clock->ops.adjust)
            clock->ops.adjust(clock);
    }

    mutex_unlock(&clocklist_lock);
}
```

tick：2. clock_settime系统调用引发remote tick

EVL

```
void evl_adjust_timers(struct evl_clock *clock, ktime_t delta)
{
    struct evl_timer *timer, *tmp;
    struct evl_timerbase *tmb;
    struct evl_tqueue *tq;
    struct evl_tnode *tn;
    struct list_head adjq;
    struct evl_rq *rq;
    unsigned long flags;
    int cpu;

    INIT_LIST_HEAD(&adjq);

    for_each_online_cpu(cpu) {
        rq = evl_cpu_rq(cpu);
        tmb = evl_percpu_timers(clock, cpu);
        tq = &tmb->q;
        raw_spin_lock_irqsave(&tmb->lock, flags);

        for_each_evl_tnode(tn, tq) {
            timer = container_of(tn, struct evl_timer, node);
            if (timer->clock == clock)
                list_add_tail(&timer->adjlink, &adjq);
        }

        if (list_empty(&adjq))
            goto next;

        list_for_each_entry_safe(timer, tmp, &adjq, adjlink) {
            list_del(&timer->adjlink);
            evl_dequeue_timer(timer, tq);
            adjust_timer(clock, timer, tq, delta);
        }

        if (rq != this_evl_rq())
            evl_program_remote_tick(clock, rq);
        else
            evl_program_local_tick(clock);
    next:
        raw_spin_unlock_irqrestore(&tmb->lock, flags);
    }
}
```

调度子系统

调度子系统：kthread_run 和 pin_to_initial_cpu

- 创建进程：kthread_run
  - kthread_create
  - wake_up_process
    - try_to_wake_up
      - cpu = select_task_rq
        - select_task_rq_fair：完成负载均衡
- 子进程：kthread_trampoline
  - map_kthread_self
    - pin_to_initial_cpu：为了保证实时性，固定在CPU上

```
            kthread_run(
                threadfn: Some(kthread_trampoline),
                data,
                namefmt: c_str!("%s").as_char_ptr(),
                msg: format_args!("{}", (*thread.locked_data().get()).name),
            )
```

```
/*
 * select_task_rq_fair: Select target runqueue for the waking task in domains
 * that have the relevant SD flag set. In practice, this is SD_BALANCE_WAKE,
 * SD_BALANCE_FORK, or SD_BALANCE_EXEC.
 *
 * Balances load by selecting the idlest CPU in the idlest group, or under
 * certain conditions an idle sibling CPU if the domain has SD_WAKE_AFFINE set.
 *
 * Returns the target CPU number.
 *
 * preempt must be disabled.
 */
static int
select_task_rq_fair(struct task_struct *p, int prev_cpu, int wake_flags)
```

调度子系统：申请RESCHEDULE_OOB_IPI

```c
int __init evl_init_sched(void)
{
	struct evl_rq *rq;
	int ret, cpu;

	register_classes();

	for_each_online_cpu(cpu) {
		rq = &per_cpu(evl_runqueues, cpu);
		init_rq(rq, cpu);
	}

	/* See comment about hooking TIMER_OOB_IPI. */
	if (IS_ENABLED(CONFIG_SMP) && num_possible_cpus() > 1) {
		ret = __request_percpu_irq(RESCHEDULE_OOB_IPI,
				oob_reschedule_interrupt,
				IRQF_OOB,
				"EVL reschedule",
				&evl_machine_cpudata);
		if (ret)
			goto cleanup_rq;
	}

	return 0;

cleanup_rq:
	for_each_online_cpu(cpu) {
		rq = evl_cpu_rq(cpu);
		destroy_rq(rq);
	}

	return ret;
}
```

```c
#ifdef CONFIG_SMP
/* oob stalled. */
static irqreturn_t oob_reschedule_interrupt(int irq, void *dev_id)
{
	trace_evl_reschedule_ipi(this_evl_rq());

	/* Will reschedule from evl_exit_irq(). */

	return IRQ_HANDLED;
}
#else
#define oob_reschedule_interrupt  NULL
#endif
```

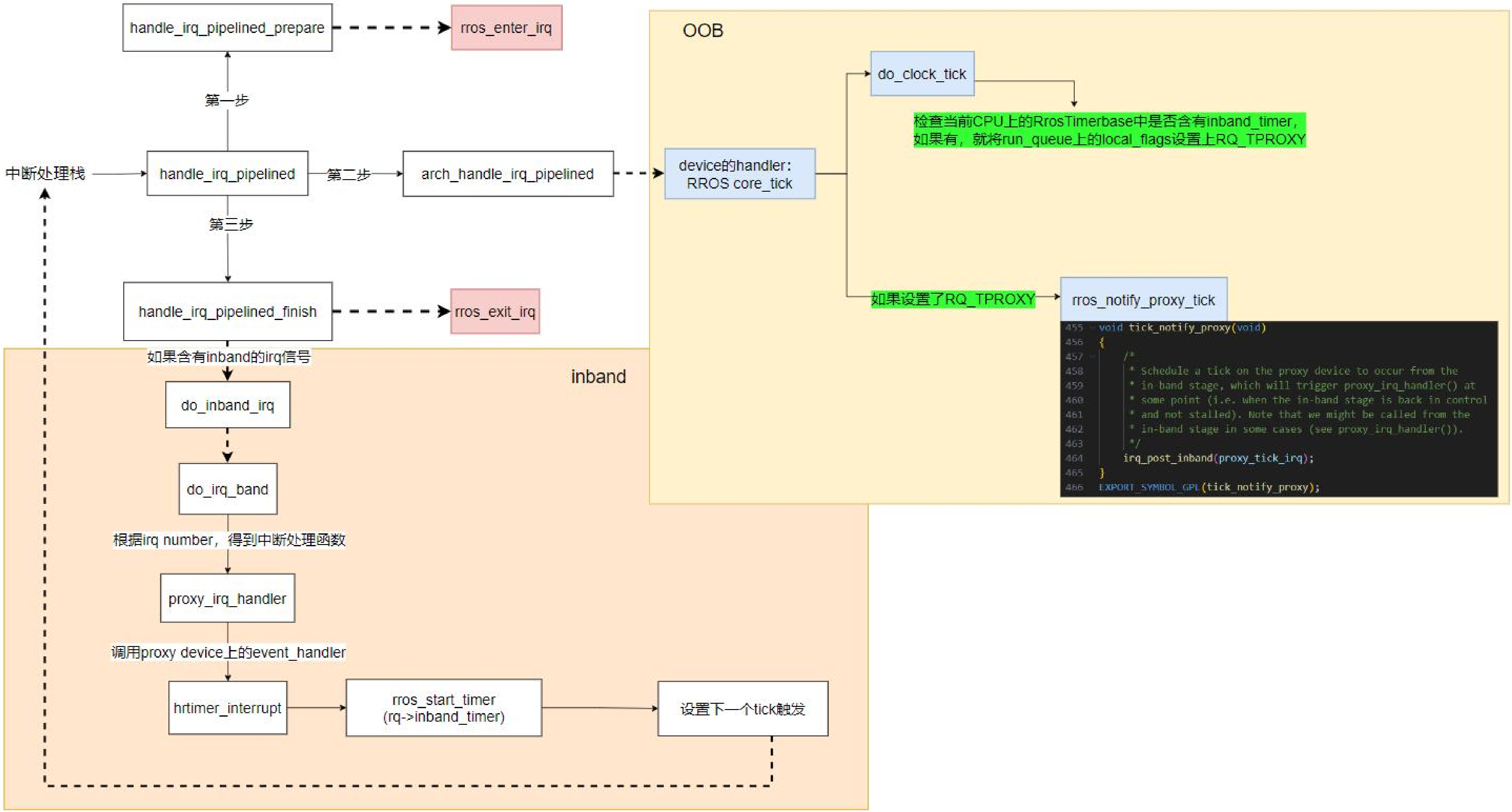
调度子系统：RESCHEDULE_OOB_IPI

- 如果CPU上的运行队列出现了变化，则会调用evl_set_resched设置一个flag，在下一次rros_schedule时就会调用test_resched函数检查是否需要重新调度；
- rq->local_flags中的RQ_SCHED表示其他CPU需要reschedule；rq->flag中的RQ_SCHED表示本CPU需要reschedule；
- 其他CPU的reschedule并不是在oob_reschedule_interrupt中断处理函数中进行的，而是在中断退出时调用rros_exit_irq函数完成的

```c
static inline void evl_set_resched(struct evl_rq *rq)
{
	struct evl_rq *this_rq = this_evl_rq();

	assert_hard_lock(&rq->lock); /* Implies hard irqs are off. */

	if (this_rq == rq) {
		this_rq->flags |= RQ_SCHED;
	} else if (!evl_need_resched(rq)) {
		rq->flags |= RQ_SCHED;
		/*
		 * The following updates change CPU-local data and
		 * hard irqs are off on the current CPU, so this is
		 * safe despite that we don't hold this_rq->lock.
		 *
		 * NOTE: raising RQ_SCHED in the local_flags too
		 * ensures that the current CPU will pass through
		 * evl_schedule() to __evl_schedule() at the next
		 * opportunity for sending the resched IPIs (see
		 * test_resched()).
		 */
		this_rq->local_flags |= RQ_SCHED;
		cpumask_set_cpu(evl_rq_cpu(rq), &this_rq->resched_cpus);
	}
}
```

```c
/* hard irqs off. */
static __always_inline bool test_resched(struct evl_rq *this_rq)
{
	bool need_resched = evl_need_resched(this_rq);

#ifdef CONFIG_SMP
	/* Send resched IPI to remote CPU(s). */
	if (unlikely(!cpumask_empty(&this_rq->resched_cpus))) {
		irq_send_oob_ipi(RESCHEDULE_OOB_IPI, &this_rq->resched_cpus);
		cpumask_clear(&this_rq->resched_cpus);
		this_rq->local_flags &= ~RQ_SCHED;
	}
#endif
	if (need_resched)
		this_rq->flags &= ~RQ_SCHED;

	return need_resched;
}
```

调度子系统：RESCHEDULE_OOB_IPI



调度子系统：RESCHEDULE_OOB_IPI

```c
/* hard irqs off. */
static inline void evl_enter_irq(void)
{
    struct evl_rq *rq = this_evl_rq();

    rq->local_flags |= RQ_IRQ;
}

/* hard irqs off. */
static inline void evl_exit_irq(void)
{
    struct evl_rq *this_rq = this_evl_rq();

    this_rq->local_flags &= ~RQ_IRQ;

    /*
     * CAUTION: Switching stages as a result of rescheduling may
     * re-enable irqs, shut them off before returning if so.
     */
    if ((this_rq->flags|this_rq->local_flags) & RQ_SCHED) {
        evl_schedule();
        if (!hard_irqs_disabled())
            hard_local_irq_disable();
    }
}
```

临界区隔离方法

中断、抢占、锁

- 抢占：允许更高优先级的任务打断当前任务
  - 内核抢占触发时机：
    1. 从中断上下文返回内核态；
    2. 内核代码直接调用schedule；
    3. 内核再次变成可抢占时，调用preempt_enable；
    4. 任务由于阻塞间接调用schedule；
  - 用户抢占触发时机：
    1. 从系统调用返回用户态时；
    2. 从中断上下文返回用户态时；

单CPU

- 单CPU上，自旋锁被简化为开关抢占；
- 单CPU、关闭内核抢占CONFIG时：
  - 隔离区只能使用开关中断实现同步；
- 单CPU、开启内核抢占CONFIG时：
  - 临界区隔离的方法中必须要有开关抢占的操作，可以通过自旋锁或者直接开关抢占实现（两者等价）；
  - 如果不开关抢占：
    - 进程1在内核态进入临界区时上A锁，然后产生一个中断，中断处理结束导致内核抢占，进程2执行，进程2进入临界区同样对A上锁，导致死锁；

evl_spin_lock & raw_spin_lock

```c
#define __evl_spin_lock(__lock)            \
    do {                                   \
        evl_disable_preempt();             \
        raw_spin_lock(&(__lock)->_lock);   \
    } while (0)
```

```c
static inline void __evl_disable_preempt(void)
{
    dovetail_current_state()->preempt_count++;
}
```

关抢占、上锁

但是在EVL中调用raw_spin_lock之前也都会关中断

```c
#define raw_spin_lock(lock) \
    LOCK_ALTERNATIVES(lock, spin_lock, _raw_spin_lock(__RAWLOCK(lock)))
```

```c
void __lockfunc _raw_spin_lock(raw_spinlock_t *lock)
{
    __raw_spin_lock(lock);
}
EXPORT_SYMBOL(_raw_spin_lock);
```

```c
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
    preempt_disable();
    spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
    LOCK_CONTENDED(lock, do_raw_spin_trylock, do_raw_spin_lock);
}
```

evl_spin_lock_irqsave & raw_spin_lock_irqsave

```c
#define evl_spin_lock_irqsave(__lock, __flags)              \
    do {                                                \
        evl_disable_preempt();                          \
        raw_spin_lock_irqsave(&(__lock)->_lock, __flags);   \
    } while (0)
```

```c
static inline void __evl_disable_preempt(void)
{
    dovetail_current_state()->preempt_count++;
}
```

关本地中断、关抢占、上锁

```c
#define raw_spin_lock_irqsave(lock, flags)              \
    LOCK_ALTERNATIVES(lock, spin_lock_irqsave,          \
    do {                                            \
        typecheck(unsigned long, flags);            \
        flags = _raw_spin_lock_irqsave(__RAWLOCK(lock));    \
    } while (0), flags)
```

```c
#define _raw_spin_lock_irqsave(lock, flags) __LOCK_IRQSAVE(lock, flags)
```

```c
#define __LOCK_IRQSAVE(lock, flags) \
  do { local_irq_save(flags); __LOCK(lock); } while (0)
```

```c
#define __LOCK(lock) \
  do { preempt_disable(); ___LOCK(lock); } while (0)
```

```c
#define local_irq_save(flags)   do { raw_local_irq_save(flags); } while (0)
```

```c
        flags = hard_local_irq_save();
```

hard_local_irq_save

```c
#define hard_local_irq_save()           native_irq_save()
```

```c
static __always_inline void native_irq_disable(void)
{
    asm volatile("cli": : :"memory");
}
```

关本地中断

多核

1. 如果在中断处理函数和当前进程共享数据时，此时需要关闭中断并且上锁，因为中断会打断当前进程的执行;
2. 嵌套中断的内核中，中断处理函数中使用自旋锁同步，为了避免本地CPU再次产生中断打断当前持有锁的内核代码，需要关闭本地中断;
3. 上自旋锁就需要关抢占（锁的操作内置）：
    - 单核不关抢占导致死锁（如前所述）；
    - 多核不关抢占导致CPU资源的浪费:
        - 进程A持有自旋锁访问临界区，产生中断，处理结束后发生内核抢占调度到进程B，进程B同样访问临界区，导致自旋，B的时间片耗尽，调度C，C同样访问...
        - 直到调度到A，释放锁，因此会造成CPU资源浪费;
        - 如果关了抢占，在中断返回之后就会检查是否可抢占，不可抢占则直接继续执行进程A;

参考资料

1. 临界区的隔离方法：https://github.com/freelancer-leon/notes/blob/master/kernel/lock/Lock-1.md
2. Timekeeping in the Linux Kernel：https://www.youtube.com/watch?v=E1uyXn0kBAc
3. Linux x86自旋锁实现：https://github.com/freelancer-leon/notes/blob/master/kernel/lock/Lock-2-Linux_x86_Spin_Lock.md
4. Linux中断、抢占、锁之间的关系：https://blog.guorongfei.com/2014/09/06/linux-interrupt-preemptive-lock/
5. Linux 中断（IRQ/softirq）基础：原理及内核实现（2022）：https://arthurchiao.art/blog/linux-irq-softirq-zh/
6. Linux Preemption - 1：https://oliveryang.net/2016/03/linux-scheduler-1/
7. Linux Preemption - 2：https://oliveryang.net/2016/03/linux-scheduler-2/
8. Linux中断处理：https://www.zhihu.com/question/271600057/answer/1593738076?utm_campaign=shareopn&utm_medium=social&utm_psn=1771915578544095232&utm_source=wechat_session
9. 中断与抢占：https://github.com/leon0625/blog/issues/10
10. Linux内核学习笔记之同步：https://hjk.life/posts/linux-kernel-synchronization/