

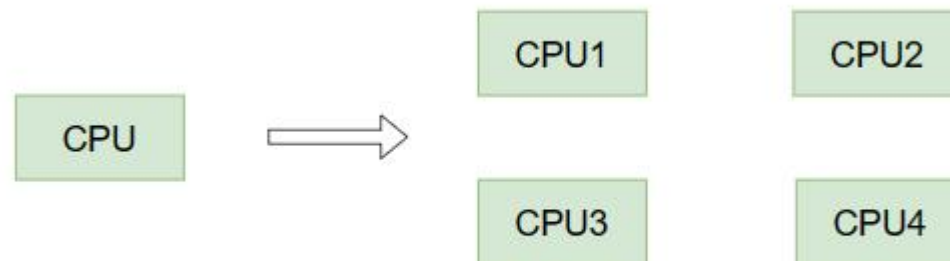
# RROS SMP多核机制

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: timekeeping时钟同步从而引起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

```
156 pub struct RrosClock {
157     resolution: KtimeT,    clock的精度
158     gravity: RrosClockGravity, 三个超参数, 系统响应时钟事件的基本时延
159     name: &'static CStr,
160     flags: i32,
161     ops: RrosClockOps,    operations
162     timerdata: *mut RrosTimerbase, clock通过alloc_percpu分配RrosTimerbase得到的地址
163     master: *mut RrosClock, 当前clock所依赖的时钟: realtime依赖于mono clock
164     offset: KtimeT, 当前clock和master clock之间的时间差值
165     next: *mut ListHead, 通过该节点连接在全局clock_list链表之上
166     element: Option<Rc<RefCell<RrosElement>>>,
167     dispose: Option<fn(&mut RrosClock)>,
168     #[cfg(CONFIG_SMP)]
169     pub affinity: Option<cpumask::CpumaskT>, CPU亲和性
170 }
```

Li Hongyu, 5个月前 | 1 author (Li Hongyu) | 1 implementation

```
86 pub struct RrosClockGravity {
87     irq: KtimeT,    系统响应时钟事件的基本时延
88     kernel: KtimeT,
89     user: KtimeT,
90 }
```

Li Hongyu, 5个月前 | 1 author (Li Hongyu) | 0 implementations

```
22 pub struct RrosTimerbase {    每个CPU上都被alloc的定时器管理链表
23     pub lock: SpinLock<i32>,
24     pub q: List<Arc<SpinLock<RrosTimer>>>,
25 }
```

# 时钟子系统相关的RROS data structures

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

```
35 pub struct RrosTimer {
36     clock: *mut RrosClock, timer所属的clock
37     date: KtimeT, timer下次需要被触发的时间
38     //adjlink: list_head, // Used when adjusting
39     status: i32,
40     pub interval: KtimeT, /* 0 == oneshot */ 当定时器是periodic时, 表示两个tick之间的间隔时间
41     pub start_date: KtimeT, 定时器开始触发的时间点
42     pexpect_ticks: u64, /* periodic release date */ 定时器本应该被触发的次数
43     periodic_ticks: u64, 定时器实际上被触发的次数
44     base: *mut RrosTimerbase, 定时器所在CPU上的定时器管理链表
45     handler: fn(*mut RrosTimer), 定时器handler函数, 等于timerfd_handler
46     name: &'static CStr,
47     #[cfg(CONFIG_RROS_RUNSTATS)]
48     scheduled: RrosCounter,
49     #[cfg(CONFIG_RROS_RUNSTATS)]
50     fired: RrosCounter,
51     #[cfg(CONFIG_SMP)]
52     rq: *mut RrosRq, 定时器所属的CPU的运行队列
53     pub thread: Option<Arc<SpinLock<RrosThread>>>,
54 }
```

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

```
541 pub struct RrosTimerFd {
542     timer: Arc<SpinLock<RrosTimer>>, 定时器本身
543     readers: RrosWaitQueue, timer未被触发时, thread调用timerfd_oob_read就会被放到该链表上
544     poll_head: RrosPollHead, 通过调用poll被阻塞的thread
545     efile: RrosFile, 将定时器和文件关联
546     ticked: bool, 标志定时器是否已经被触发
547 }
```



# 时钟初始化: rros\_clock\_init

RROS中含有两个clock:

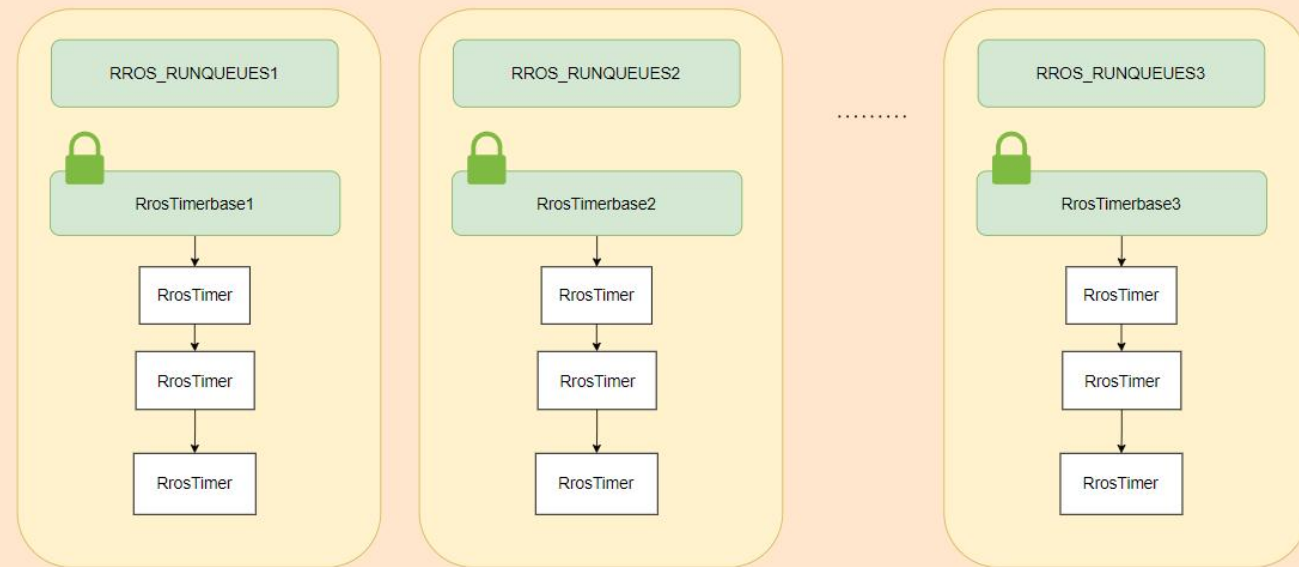
1. RROS\_MONO\_CLOCK
2. RROS\_REALTIME\_CLOCK: 附属在RROS\_MONO\_CLOCK上

```
1234 pub fn rros_clock_init() -> Result<usize> {
1235     let pinned: Pin<&mut SpinLock<i32>> = unsafe { Pin::new_unchecked(pointer: &mut CLOCKLIST_LOCK) };
1236     spinlock_init!(pinned, "CLOCKLIST_LOCK");
1237     unsafe {
1238         RROS_MONO_CLOCK.reset_gravity();
1239         RROS_REALTIME_CLOCK.reset_gravity();
1240
1241         let mut element: RrosElement = RrosElement::new()?;
1242         element.pointer = &mut RROS_MONO_CLOCK as *mut _ as *mut u8;
1243         RROS_MONO_CLOCK.element = Some(Rc::try_new(RefCell::new(element)).unwrap());
1244         let mut element: RrosElement = RrosElement::new()?;
1245         element.pointer = &mut RROS_REALTIME_CLOCK as *mut _ as *mut u8;
1246         RROS_REALTIME_CLOCK.element = Some(Rc::try_new(RefCell::new(element)).unwrap());
1247
1248         rros_init_clock(&mut RROS_MONO_CLOCK, affinity: &RROS_OOB_CPUS)?;
1249     }
1250     let ret: Result<usize, Error> = unsafe { rros_init_slave_clock(&mut RROS_REALTIME_CLOCK, master: &mut RROS_MONO_CLOCK) };
1251     if let Err(_) = ret {
1252         //rros_put_element(&rros_mono_clock.element);
1253     }
1254     // pr_debug!("clock init success!");
1255     ok(0)
1256 }
```

# 时钟初始化: rros\_init\_clock

```
1190 fn rros_init_clock(clock: &mut RrosClock, affinity: &cpumask::CpumaskT) -> Result<usize> {
1191     preempt::running_inband()?;
1192
1193     #[cfg(CONFIG_SMP)]
1194     {
1195         if clock.affinity.is_none() {
1196             clock.affinity = Some(cpumask::CpumaskT::from_int(0));
1197         }
1198         let clock_affinity: &mut CpumaskT = clock.affinity.as_mut().unwrap();
1199         if affinity.cpumask_empty().is_ok() {
1200             clock_affinity.cpumask_clear();
1201             clock_affinity.cpumask_set_cpu(unsafe { RROS_OOB_CPUS.cpumask_first() as u32 });
1202         } else {
1203             clock_affinity.cpumask_and(src1: affinity, src2: unsafe { &RROS_OOB_CPUS });
1204             if clock_affinity.cpumask_empty().is_ok() {
1205                 return Err(Error::EINVAL);
1206             }
1207         }
1208     }
1209
1210     // 8 byte alignment
1211     let tmb: *mut RrosTimerbase = percpu::alloc_per_cpu(
1212         size: size_of::<RrosTimerbase>() as usize,
1213         align: align_of::<RrosTimerbase>() as usize,
1214     ) as *mut RrosTimerbase;
1215     if tmb == 0 as *mut RrosTimerbase {
1216         return Err(kernel::Error::ENOMEM);
1217     }
1218     clock.timerdata = tmb;
1219
1220     for cpu: <OnlineCpusIndexIter as IntoIterator>::Iter in online_cpus() {
1221         let mut tmb: *mut RrosTimerbase = rros_percpu_timers(clock, cpu as i32);
1222         unsafe { raw_spin_lock_init(&mut (*tmb).lock); }
1223     }
1224
1225     clock.offset = 0;
1226     let ret: Result<usize, Error> = init_clock(clock as *mut RrosClock, master: clock as *mut RrosClock);
1227     if let Err(_) = ret {
1228         percpu::free_per_cpu(pdata: clock.get_timerdata_addr() as *mut u8);
1229         return ret;
1230     }
1231     Ok(0)
1232 } fn rros_init_clock
```

每个CPU上都含有一个timerbase链表



# 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

```
21 static unsigned int proxy_tick_irq;
22
23 static DEFINE_MUTEX(proxy_mutex);
24
25 static DEFINE_PER_CPU(struct clock_proxy_device, proxy_tick_device);
361
362 DECLARE_PER_CPU(struct tick_device, tick_cpu_device);
363
```

```
12 struct tick_device {
13     struct clock_event_device *evtdev;
14     enum tick_device_mode mode;
15 };
16
```

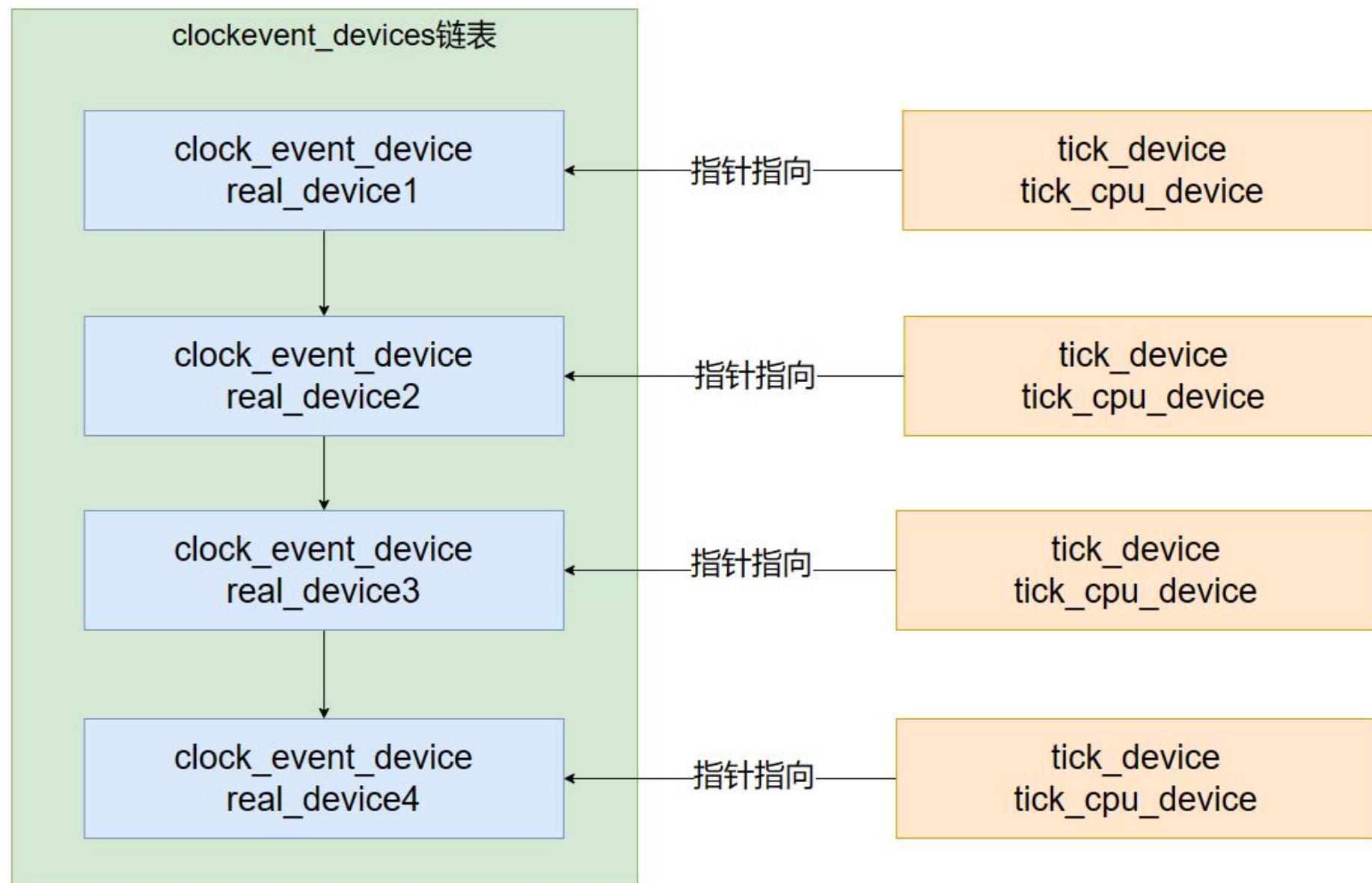
```
19 /* The registered clock event devices */
20 static LIST_HEAD(clockevent_devices);
21 static LIST_HEAD(clockevents_released);
22 /* Protection for the above */
23 static DEFINE_RAW_SPINLOCK(clockevents_lock);
24 /* Protection for unbind operations */
25 static DEFINE_MUTEX(clockevents_mutex);
```

```
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);
};
```

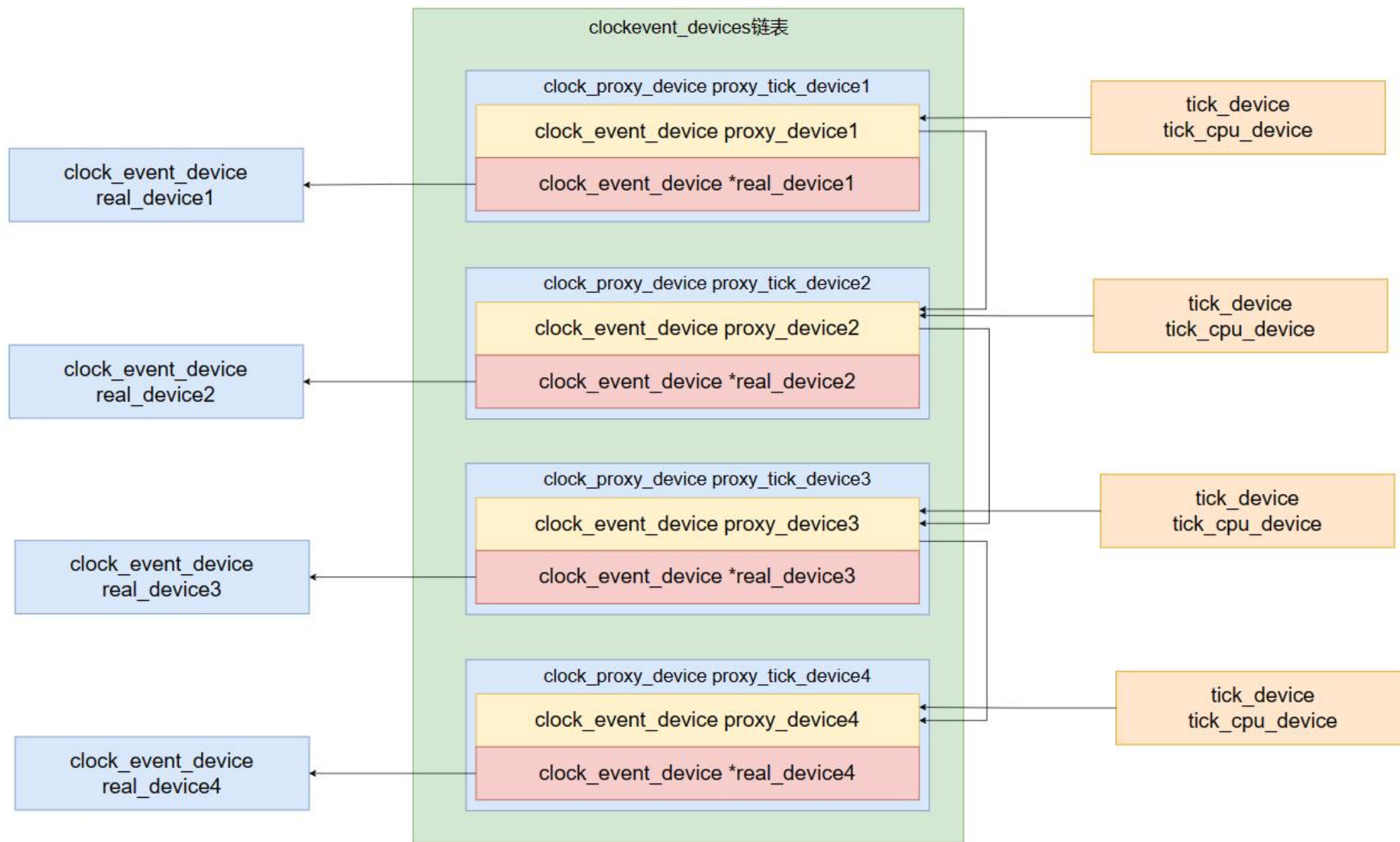
```
24 static DEFINE_PER_CPU(struct clock_proxy_device *, proxy_device);
25
```



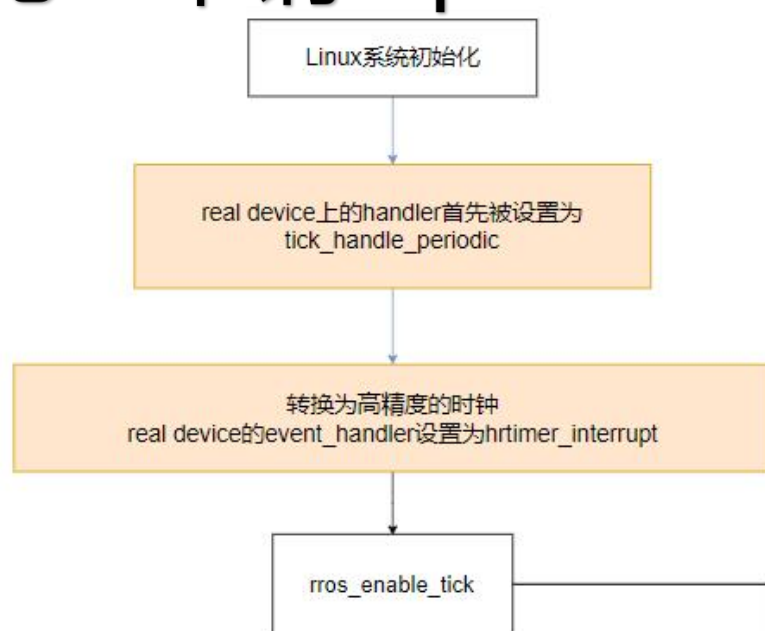
# tick初始化: 原始的real device



# tick初始化: real device和proxy device的关系



# tick初始化: 申请irq



Linux

```
83 #ifdef CONFIG_SMP
84 static irqreturn_t clock_ipi_handler(int irq, void *dev_id)
85 {
86     evl_core_tick(NULL);
87     return IRQ_HANDLED;
88 }
89 #else
90 #define clock_ipi_handler NULL
91 #endif
```

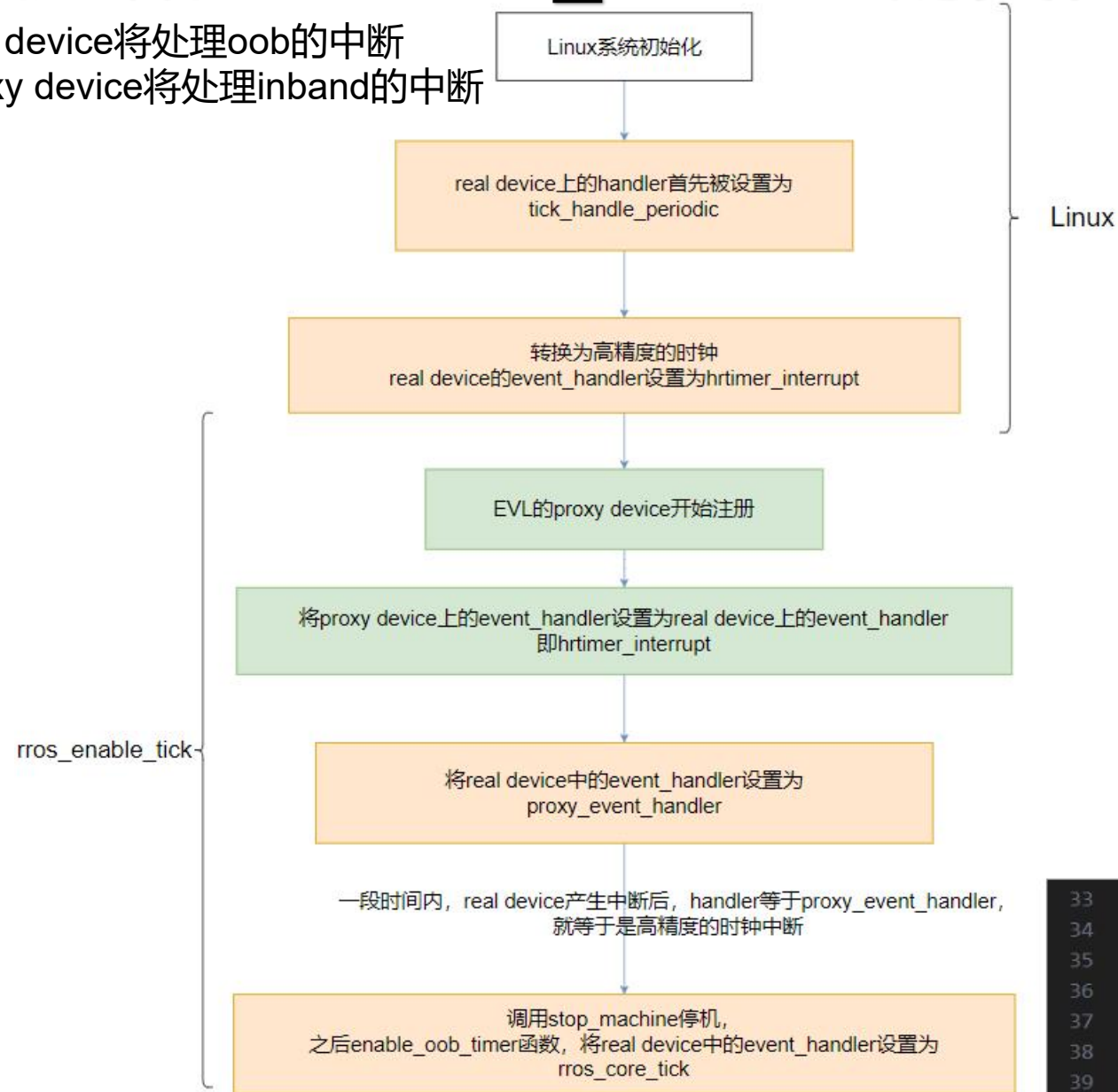
```
111 /*
112  * We may be running a SMP kernel on a uniprocessor machine
113  * whose interrupt controller provides no IPI: attempt to hook
114  * the timer IPI only if the hardware can support multiple
115  * CPUs.
116  */
117 if (IS_ENABLED(CONFIG_SMP) && num_possible_cpus() > 1) {
118     ret = __request_percpu_irq(TIMER_OOB_IPI,
119                               clock_ipi_handler,
120                               IRQF_OOB, "EVL timer IPI",
121                               &evl_machine_cpudata);
122     if (ret)
123         return ret;
124 }
```

```
317 int tick_install_proxy(void (*setup_proxy)(struct clock_proxy_device *dev),
318                        const struct cpumask *cpumask)
319 {
320     struct proxy_install_arg arg;
321     int ret, sirq;
322
323     mutex_lock(&proxy_mutex);
324
325     ret = -EAGAIN;
326     if (proxy_tick_irq)
327         goto out;
328
329     sirq = irq_create_direct_mapping(synthetic_irq_domain);
330     if (WARN_ON(sirq == 0))
331         goto out;
332
333     ret = __request_percpu_irq(sirq, proxy_irq_handler,
334                               IRQF_NO_THREAD, /* no IRQF_TIMER here. */
335                               "proxy tick",
336                               &proxy_tick_device);
337     if (WARN_ON(ret)) {
338         irq_dispose_mapping(sirq);
339         goto out;
340     }
341
342     proxy_tick_irq = sirq;
343     barrier();
```



# 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

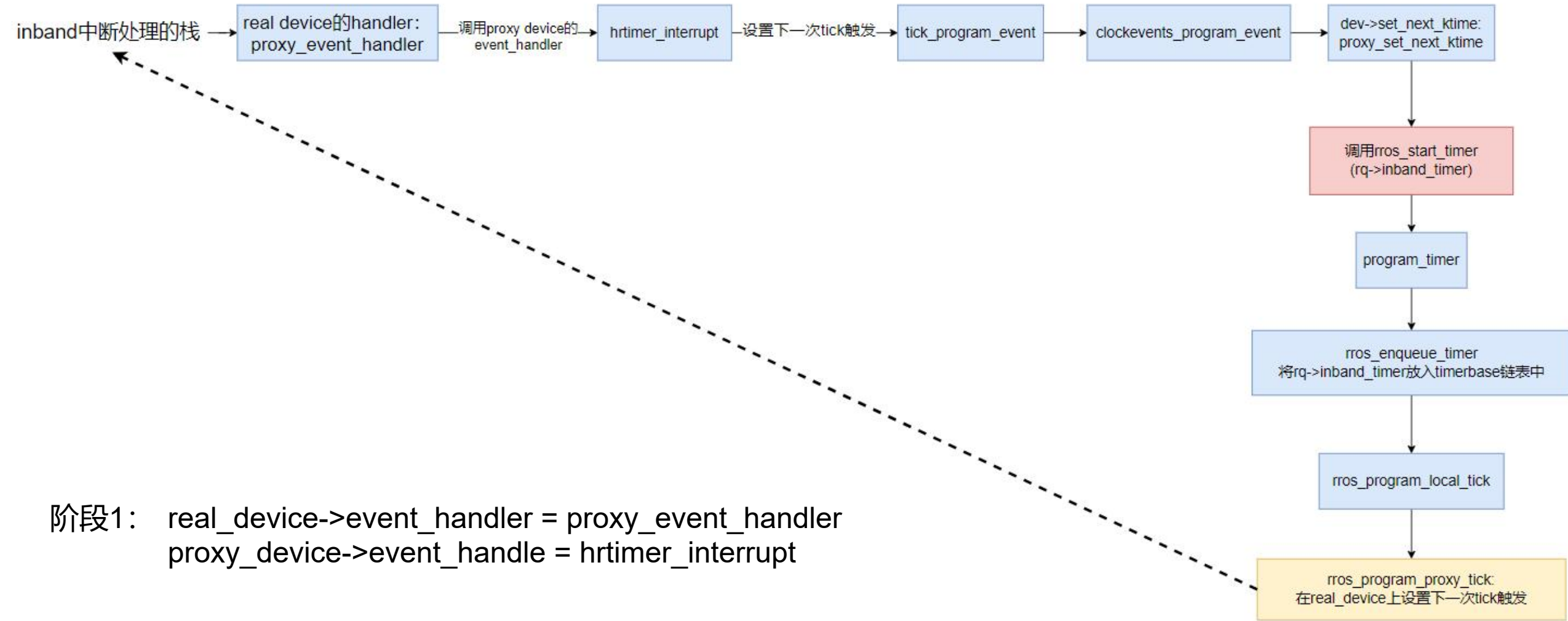


```
165 static irqreturn_t proxy_irq_handler(int irq, void *dev_id)
166 {
167     struct clock_event_device *evt;
168
169     /*
170      * Tricky: we may end up running this in-band IRQ handler
171      * because tick_notify_proxy() was posted either:
172      *
173      * - from the out-of-band stage via ->handle_oob_event() for
174      *   emulating an in-band tick. In this case, the active tick
175      *   device for the in-band timing core is the proxy device,
176      *   whose event handler is still the same than the real tick
177      *   device's.
178      *
179      * - directly by the clock chip driver on the local CPU via
180      *   clockevents_handle_event(), for propagating a tick to the
181      *   in-band stage nobody from the out-of-band stage is
182      *   interested on i.e. no proxy device was registered on the
183      *   receiving CPU, which was excluded from @cpumask in the call
184      *   to tick_install_proxy(). In this case, the active tick
185      *   device for the in-band timing core is a real clock event
186      *   device.
187      *
188      * In both cases, we are running on the in-band stage, and we
189      * should fire the event handler of the currently active tick
190      * device for the in-band timing core.
191      */
192     evt = raw_cpu_ptr(&tick_cpu_device)->evtdev;
193     evt->event_handler(evt);
194
195     return IRQ_HANDLED;
196 }
```

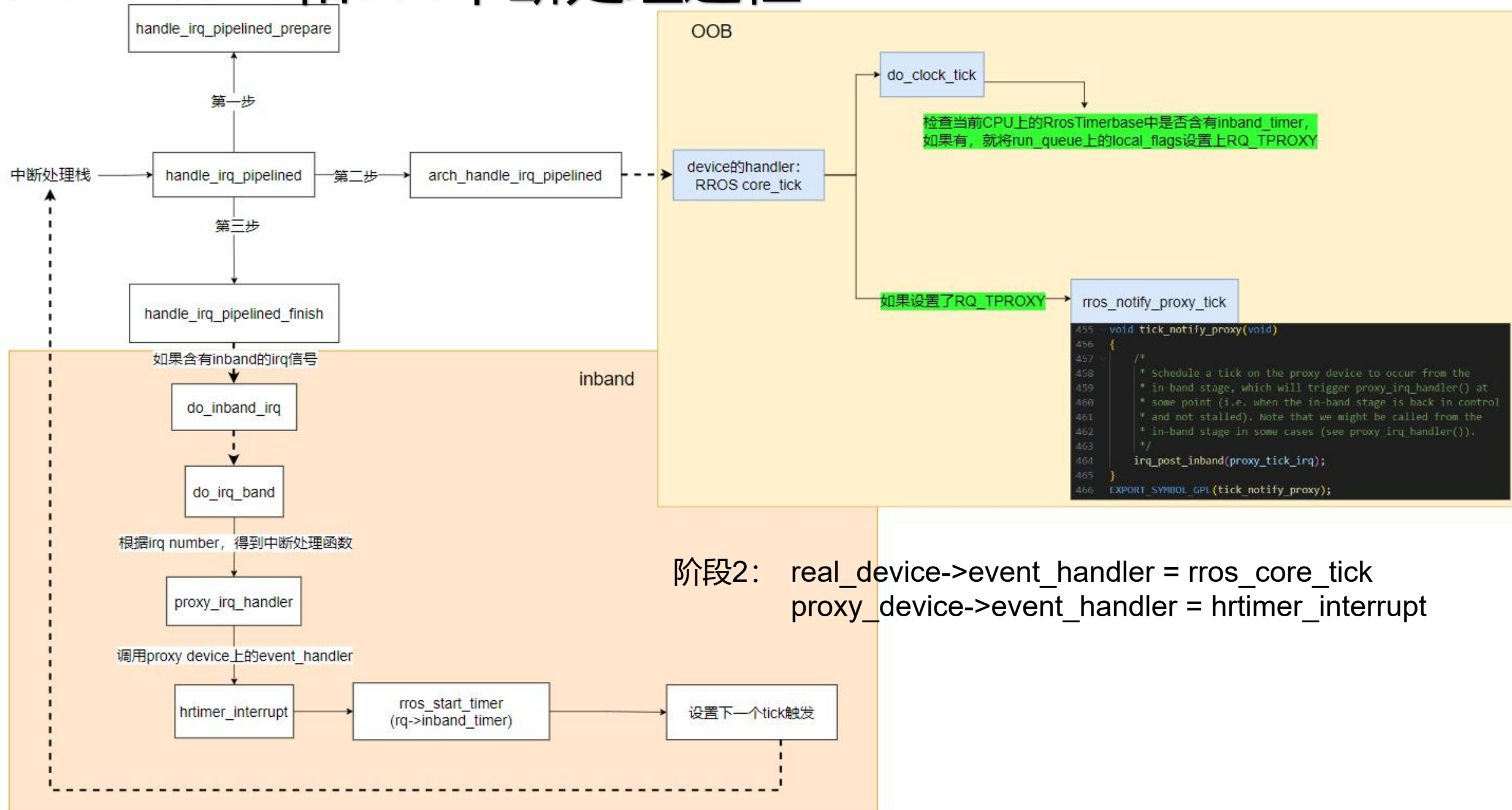
```
33 static void proxy_event_handler(struct clock_event_device *real_dev)
34 {
35     struct clock_proxy_device *dev = raw_cpu_ptr(&proxy_tick_device);
36     struct clock_event_device *proxy_dev = &dev->proxy_device;
37
38     proxy_dev->event_handler(proxy_dev);
39 }
```



# tick: inband和oob中断处理过程



# tick: inband和oob中断处理过程



# 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
541 pub struct RrosTimerFd {
542     timer: Arc<SpinLock<RrosTimer>>, 定时器本身
543     readers: RrosWaitQueue, timer未被触发时, thread调用timerfd_oob_read就会被放到该链表上
544     poll_head: RrosPollHead, 通过调用poll被阻塞的thread
545     efile: RrosFile, 将定时器和文件关联
546     ticked: bool, 标志定时器是否已经被触发
547 }
```

# 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`

```
279 #define evl_init_timer_on_rq(__timer, __clock, __handler, __rq, __flags) \
280     __evl_init_timer(__timer, __clock, __handler, \
281         __rq, #__handler, __flags)
282
289 void __evl_init_timer(struct evl_timer *timer,
290     struct evl_clock *clock,
291     void (*handler)(struct evl_timer *timer),
292     struct evl_rq *rq,
293     const char *name,
294     int flags)
295 {
296     int cpu;
297
298     timer->clock = clock;
299     evl_tdate(timer) = EVL_INFINITE;
300     timer->status = EVL_TIMER_DEQUEUED | (flags & EVL_TIMER_INIT_MASK);
301     timer->handler = handler;
302     timer->interval = EVL_INFINITE;
303
304     /*
305      * Set the timer affinity to the CPU rq is on if given, or the
306      * first CPU which may run EVL threads otherwise.
307      */
308     cpu = rq ?
309         get_clock_cpu(clock->master, evl_rq_cpu(rq)) :
310         cpumask_first(&evl_cpu_affinity);
311     #ifdef CONFIG_SMP
312     timer->rq = evl_cpu_rq(cpu);
313     #endif
314     timer->base = evl_percpu_timers(clock, cpu);
315     timer->clock = clock;
316     timer->name = name ? "<timer>";
317     evl_reset_timer_stats(timer);
318 }
```

传入的rq=NULL,  
所以创建在第一个  
oob CPU上

```
739 static int new_timerfd(struct evl_clock *clock)
740 {
741     struct evl_timerfd *timerfd;
742     struct file *filp;
743     int ret, fd;
744
745     timerfd = kzalloc(sizeof(*timerfd), GFP_KERNEL);
746     if (timerfd == NULL)
747         return -ENOMEM;
748
749     filp = anon_inode_getfile("[evl-timerfd]", &timerfd_fops,
750         timerfd, O_RDWR|O_CLOEXEC);
751     if (IS_ERR(filp)) {
752         kfree(timerfd);
753         return PTR_ERR(filp);
754     }
755
756     /*
757      * From that point, timerfd_release() might be called for
758      * cleaning up on error via filp_close(). So initialize
759      * everything we need for a graceful cleanup.
760      */
761     evl_get_element(&clock->element);
762     evl_init_timer_on_rq(&timerfd->timer, clock, timerfd_handler,
763         NULL, EVL_TIMER_UGRAVITY);
764     evl_init_wait(&timerfd->readers, clock, EVL_WAIT_PRIO);
765     evl_init_poll_head(&timerfd->poll_head);
766
767     ret = evl_open_file(&timerfd->efile, filp);
768     if (ret)
769         goto fail_open;
770
771     fd = get_unused_fd_flags(O_RDWR|O_CLOEXEC);
772     if (fd < 0) {
773         ret = fd;
774         goto fail_getfd;
775     }
776
777     fd_install(fd, filp);
778
779     return fd;
```



# tick: 1. 进程设置定时器触发时间, 引发remote tick

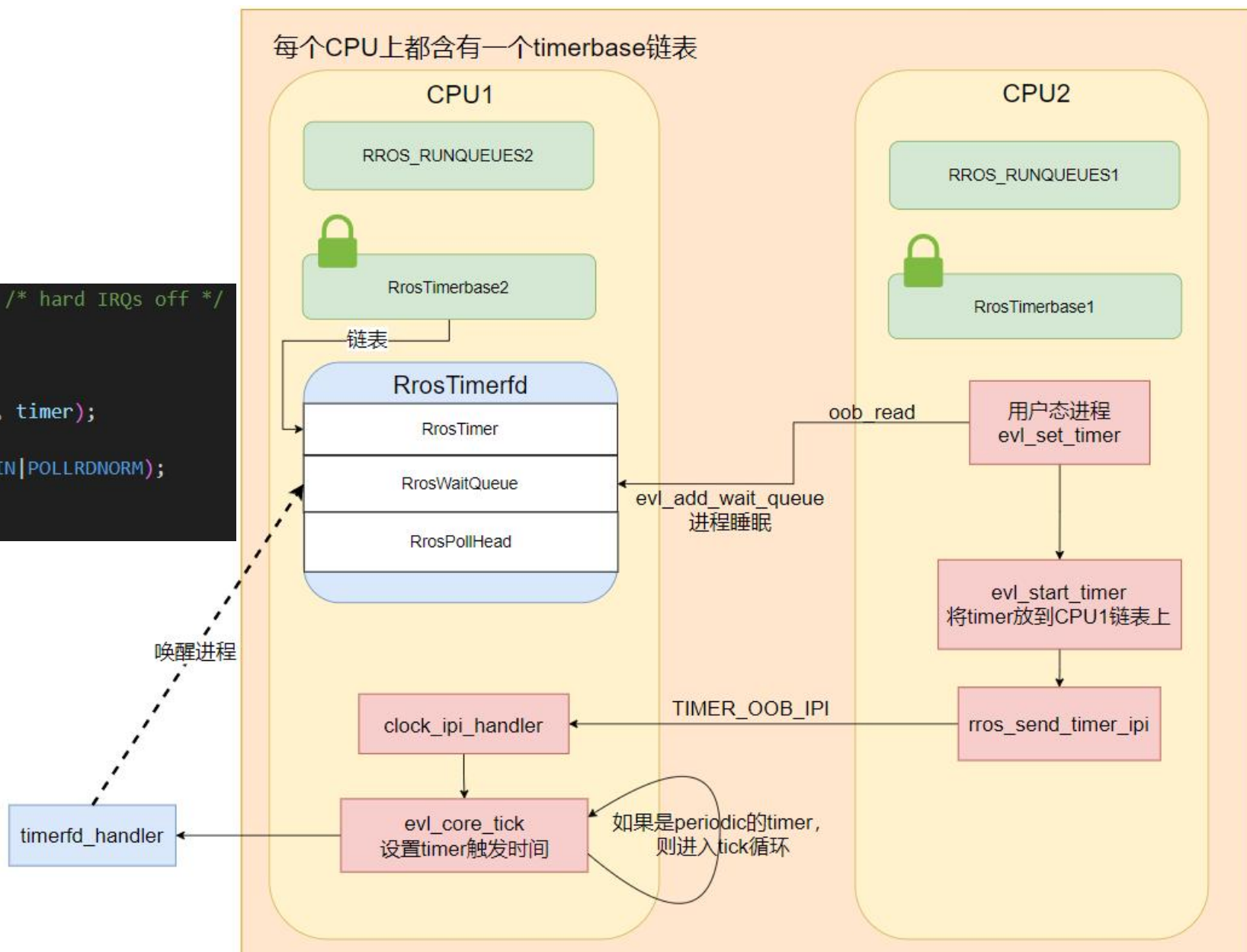
- 用户态: evl\_set\_timer
  - 用户态: \_\_evl\_common\_ioctl(EVL\_TFDIOC\_SET)
    - timerfd\_common\_ioctl
      - set\_timerfd
        - set\_timer\_value
          - rros\_start\_timer
            - program\_timer
              - 如果timer在当前CPU上, 则rros\_program\_local\_tick
              - 如果timer不在当前CPU上, 则rros\_program\_remote\_tick
                - 调用rros\_send\_timer\_ipi

```
83  #ifdef CONFIG_SMP
84  static irqreturn_t clock_ipi_handler(int irq, void *dev_id)
85  {
86      evl_core_tick(NULL);
87
88      return IRQ_HANDLED;
89  }
90  #else
91  #define clock_ipi_handler NULL
92  #endif
```

```
278  #[cfg(CONFIG_SMP)]
279  pub fn rros_send_timer_ipi(_clock: &RrosClock, rq: *mut RrosRq) {
280      |   irq_send_oob_ipi(ipi: irq_get_timer_oob_ipi(), cpumask: cpumask::CpumaskT::cpumask_of(cpu: rros_rq_cpu(rq) as u32));
281      |   }
282      |   TIMER_OOB_IPI
```

# tick: 1. 进程设置定时器触发时间

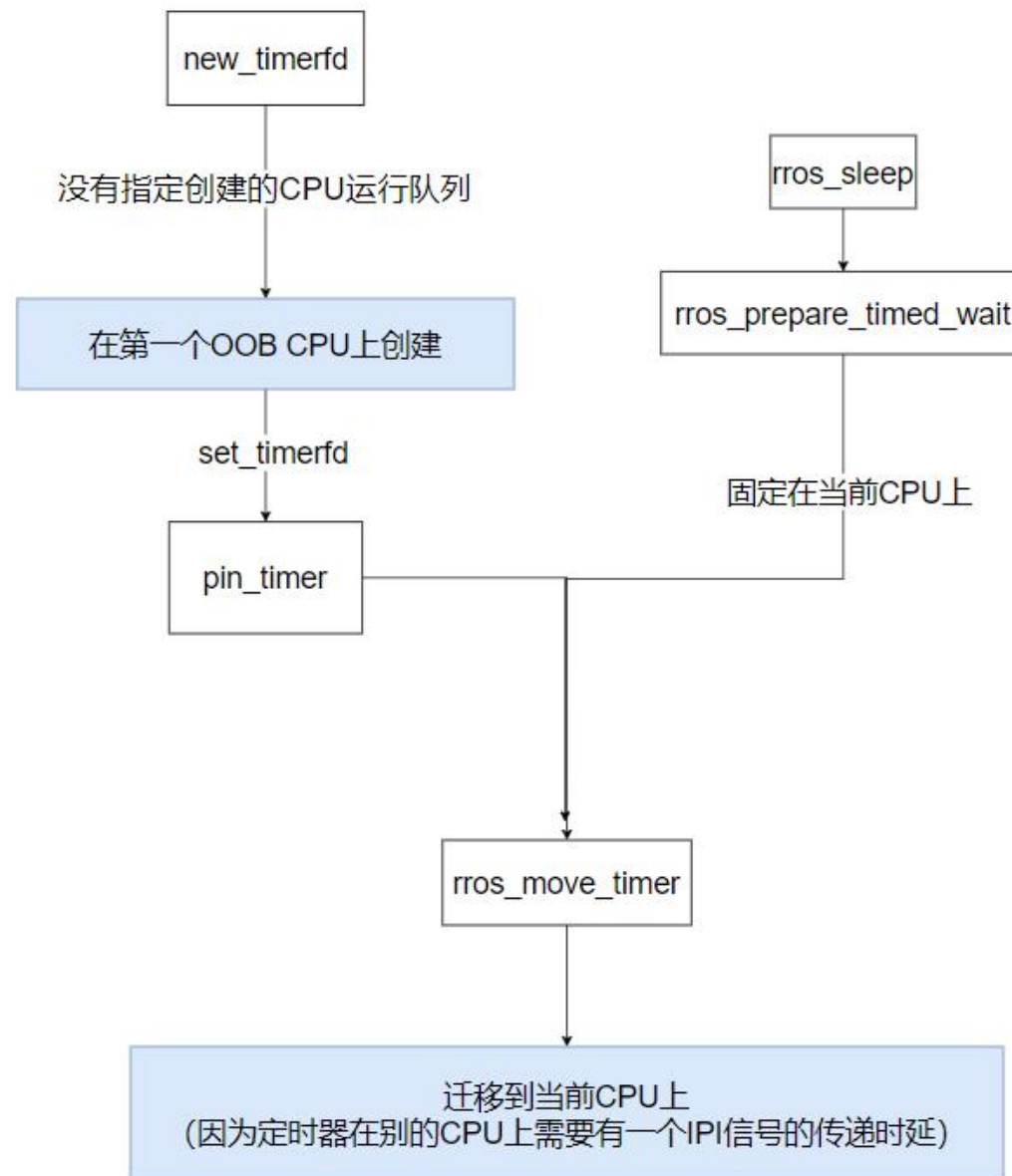
```
604 static void timerfd_handler(struct evl_timer *timer) /* hard IRQs off */
605 {
606     struct evl_timerfd *timerfd;
607
608     timerfd = container_of(timer, struct evl_timerfd, timer);
609     timerfd->ticked = true;
610     evl_signal_poll_events(&timerfd->poll_head, POLLIN|POLLRDNORM);
611     evl_flush_wait(&timerfd->readers, 0);
612 }
```



# tick: 1. pin\_timer

由于有pin\_timer和rros\_move\_timer, 所以一般不会引发remote tick

```
675 fn set_timerfd(  
676     timerfd: &RrosTimerFd,  
677     value: &mut Itimerspec64,  
678     ovalue: Itimerspec64,  
679 ) -> Result<i32> {  
680     get_timer_value(timer: timerfd.timer.clone(), value);  
681  
682     if rros_current() != core::ptr::null_mut() {  
683         pin_timer(timerfd.timer.clone());  
684     }  
685  
686     set_timer_value(timer: timerfd.timer.clone(), value)  
687 }
```





# tick: 2. timekeeping时钟同步引发remote tick

timekeeping模块会按照自己的节奏来更新系统时间

RROS还没有实现，EVL中是：

```
(gdb) bt
#0  inband_clock_was_set () at kernel/dovetail.c:398
#1  0xffffffffc010173990 in clock_was_set () at kernel/time/hrtimer.c:876
#2  0xffffffffc0101777bc in do_settimeofday64 (ts=0xffffffffc0116dbdd8)
    at kernel/time/timekeeping.c:1327
#3  0xffffffffc01016dbdc in do_sys_settimeofday64 (tv=0xffffffffc0116dbdd8, tz=0x0)
    at kernel/time/time.c:195
#4  0xffffffffc01018115c in posix_clock_realtime_set (which_clock=<optimized out>,
    tp=0x169169e265d18f20) at kernel/time/posix-timers.c:185
#5  0xffffffffc01017fbc0 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  0xffffffffc01002a264 in __invoke_syscall (regs=0xffffffffc0116dbeb0,
    syscall_fn=0xffffffffc0115c7bd8 <tk_core+152>) at arch/arm64/kernel/syscall.c:39
#9  invoke_syscall (regs=0xffffffffc0116dbeb0, scno=0, sc_nr=447, syscall_table=0x100)
    at arch/arm64/kernel/syscall.c:53
#10 0xffffffffc01002a170 in el0_svc_common (regs=0xffffffffc0116dbeb0, scno=112, sc_nr=447,
    syscall_table=0xffffffffc010cd0ae8 <sys_call_table>) at arch/arm64/kernel/syscall.c:153
#11 0xffffffffc01002a058 in do_el0_svc (regs=0x169169e265d18f20)
    at arch/arm64/kernel/syscall.c:193
#12 0xffffffffc010ca4494 in el0_svc (regs=0xffffffffc0116dbeb0)
    at arch/arm64/kernel/entry-common.c:528
#13 0xffffffffc010ca4408 in el0_sync_handler (regs=0x169169e265d18f20)
    at arch/arm64/kernel/entry-common.c:544
#14 0xffffffffc010011ed8 in el0_sync () at arch/arm64/kernel/entry.S:777
Backtrace stopped: Cannot access memory at address 0xffffffffc0116dc0c8
(gdb) █
```

```
140 void inband_clock_was_set(void)
141 {
142     struct evl_clock *clock;
143
144     if (!evl_is_enabled())
145         return;
146
147     mutex_lock(&clocklist_lock);
148
149     list_for_each_entry(clock, &clock_list, next) {
150         if (clock->ops.adjust)
151             clock->ops.adjust(clock);
152     }
153
154     mutex_unlock(&clocklist_lock);
155 }
```



# tick: 2. timekeeping时钟同步引发remote tick

```
96 void evl_adjust_timers(struct evl_clock *clock, ktime_t delta)
97 {
98     struct evl_timer *timer, *tmp;
99     struct evl_timerbase *tmb;
100     struct evl_queue *tq;
101     struct evl_tnode *tn;
102     struct list_head adjq;
103     struct evl_rq *rq;
104     unsigned long flags;
105     int cpu;
106
107     INIT_LIST_HEAD(&adjq);
108
109     for_each_online_cpu(cpu) {
110         rq = evl_cpu_rq(cpu);
111         tmb = evl_percpu_timers(clock, cpu);
112         tq = &tmb->q;
113         raw_spin_lock_irqsave(&tmb->lock, flags);
114
115         for_each_evl_tnode(tn, tq) {
116             timer = container_of(tn, struct evl_timer, node);
117             if (timer->clock == clock)
118                 list_add_tail(&timer->adjlink, &adjq);
119         }
120
121         if (list_empty(&adjq))
122             goto next;
123
124         list_for_each_entry_safe(timer, tmp, &adjq, adjlink) {
125             list_del(&timer->adjlink);
126             evl_dequeue_timer(timer, tq);
127             adjust_timer(clock, timer, tq, delta);
128         }
129
130         if (rq != this_evl_rq())
131             evl_program_remote_tick(clock, rq);
132         else
133             evl_program_local_tick(clock);
134     next:
135         raw_spin_unlock_irqrestore(&tmb->lock, flags);
136     }
137 }
```

# 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

```
519 |         kthread_run(  
520 |             threadfn: Some(kthread_trampoline),  
521 |             data,  
522 |             namefmt: c_str!("{}", thread.locked_data().get().name),  
523 |             msg: format_args!("{}", (*thread.locked_data().get()).name),  
524 |         )
```

```
6827 | /*  
6828 |  * select_task_rq_fair: Select target runqueue for the waking task in domains  
6829 |  * that have the relevant SD flag set. In practice, this is SD_BALANCE_WAKE,  
6830 |  * SD_BALANCE_FORK, or SD_BALANCE_EXEC.  
6831 |  *  
6832 |  * Balances load by selecting the idlest CPU in the idlest group, or under  
6833 |  * certain conditions an idle sibling CPU if the domain has SD_WAKE_AFFINE set.  
6834 |  *  
6835 |  * Returns the target CPU number.  
6836 |  *  
6837 |  * preempt must be disabled.  
6838 |  */  
6839 | static int  
6840 | select_task_rq_fair(struct task_struct *p, int prev_cpu, int wake_flags)
```

# 调度子系统：申请RESCHEDULE\_OOB\_IPI

```
1342 int __init evl_init_sched(void)
1343 {
1344     struct evl_rq *rq;
1345     int ret, cpu;
1346
1347     register_classes();
1348
1349     for_each_online_cpu(cpu) {
1350         rq = &per_cpu(evl_runqueues, cpu);
1351         init_rq(rq, cpu);
1352     }
1353
1354     /* See comment about hooking TIMER_OOB_IPI. */
1355     if (IS_ENABLED(CONFIG_SMP) && num_possible_cpus() > 1) {
1356         ret = __request_percpu_irq(RESCHEDULE_OOB_IPI,
1357                                   oob_reschedule_interrupt,
1358                                   IRQF_OOB,
1359                                   "EVL reschedule",
1360                                   &evl_machine_cpudata);
1361         if (ret)
1362             goto cleanup_rq;
1363     }
1364
1365     return 0;
1366
1367 cleanup_rq:
1368     for_each_online_cpu(cpu) {
1369         rq = evl_cpu_rq(cpu);
1370         destroy_rq(rq);
1371     }
1372
1373     return ret;
1374 }
```

```
722 #ifdef CONFIG_SMP
723 /* oob stalled. */
724 static irqreturn_t oob_reschedule_interrupt(int irq, void *dev_id)
725 {
726     trace_evl_reschedule_ipi(this_evl_rq());
727
728     /* Will reschedule from evl_exit_irq(). */
729
730     return IRQ_HANDLED;
731 }
732 #else
733 #define oob_reschedule_interrupt NULL
734 #endif
```



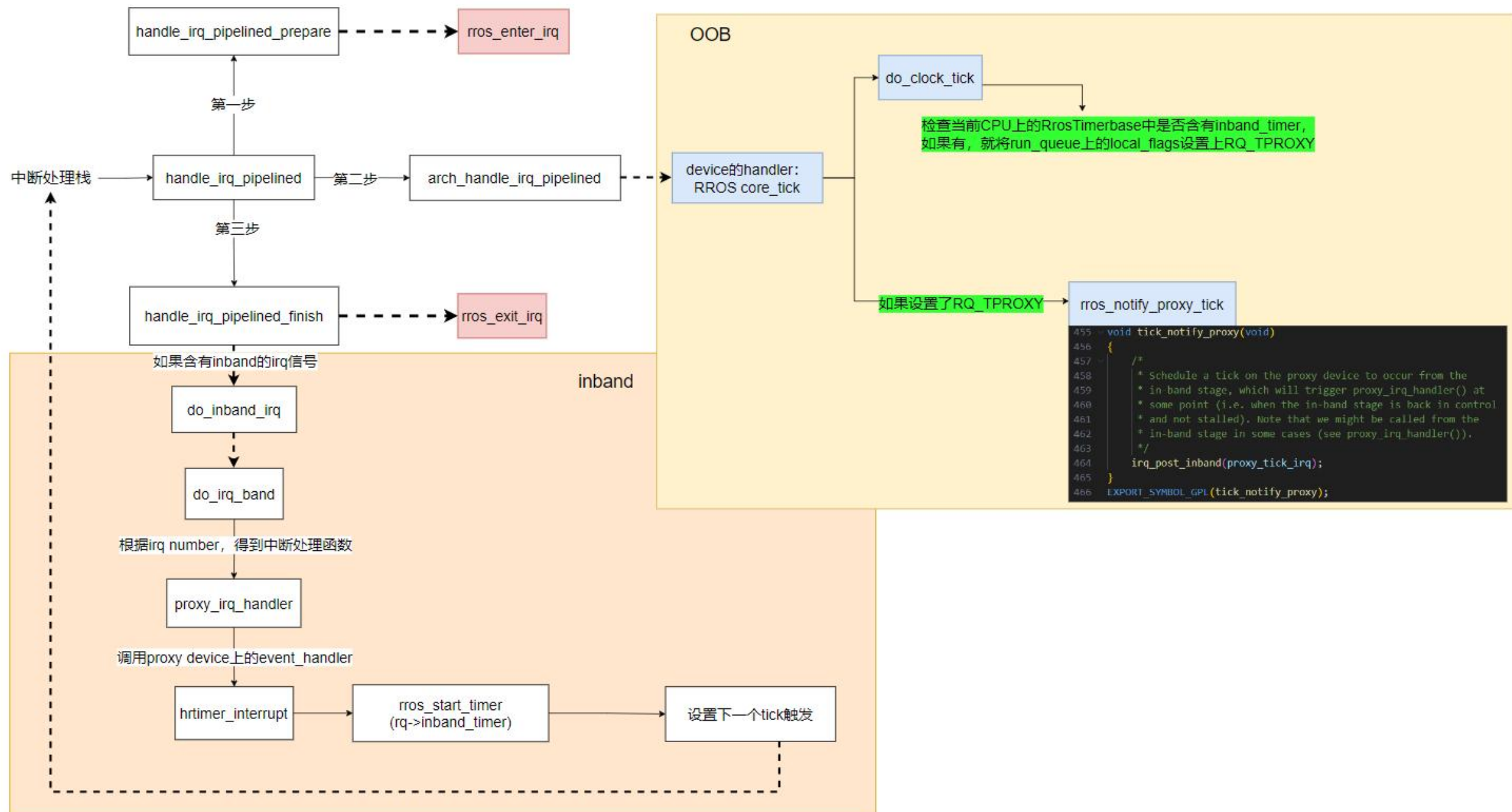
# 调度子系统: RESCHEDULE\_OOB\_IPI

- 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

```
219 static inline void evl_set_resched(struct evl_rq *rq)
220 {
221     struct evl_rq *this_rq = this_evl_rq();
222
223     assert_hard_lock(&rq->lock); /* Implies hard irqs are off. */
224
225     if (this_rq == rq) {
226         this_rq->flags |= RQ_SCHED;
227     } else if (!evl_need_resched(rq)) {
228         rq->flags |= RQ_SCHED;
229         /*
230          * The following updates change CPU-local data and
231          * hard irqs are off on the current CPU, so this is
232          * safe despite that we don't hold this_rq->lock.
233          *
234          * NOTE: raising RQ_SCHED in the local_flags too
235          * ensures that the current CPU will pass through
236          * evl_schedule() to __evl_schedule() at the next
237          * opportunity for sending the resched IPIs (see
238          * test_resched()).
239          */
240         this_rq->local_flags |= RQ_SCHED;
241         cpumask_set_cpu(evl_rq_cpu(rq), &this_rq->resched_cpus);
242     }
243 }
```

```
886 /* hard irqs off. */
887 static __always_inline bool test_resched(struct evl_rq *this_rq)
888 {
889     bool need_resched = evl_need_resched(this_rq);
890
891     #ifdef CONFIG_SMP
892         /* Send resched IPI to remote CPU(s). */
893         if (unlikely(!cpumask_empty(&this_rq->resched_cpus))) {
894             irq_send_oob_ipi(RESCHEDULE_OOB_IPI, &this_rq->resched_cpus);
895             cpumask_clear(&this_rq->resched_cpus);
896             this_rq->local_flags &= ~RQ_SCHED;
897         }
898     #endif
899     if (need_resched)
900         this_rq->flags &= ~RQ_SCHED;
901
902     return need_resched;
903 }
```

# 调度子系统: RESCHEDULE\_OOB\_IPI



# 调度子系统: RESCHEDULE\_OOB\_IPI

```
12  /* hard irqs off. */
13  static inline void evl_enter_irq(void)
14  {
15      struct evl_rq *rq = this_evl_rq();
16
17      rq->local_flags |= RQ_IRQ;
18  }
19
20  /* hard irqs off. */
21  static inline void evl_exit_irq(void)
22  {
23      struct evl_rq *this_rq = this_evl_rq();
24
25      this_rq->local_flags &= ~RQ_IRQ;
26
27      /*
28       * CAUTION: Switching stages as a result of rescheduling may
29       * re-enable irqs, shut them off before returning if so.
30       */
31      if ((this_rq->flags | this_rq->local_flags) & RQ_SCHED) {
32          evl_schedule();
33          if (!hard_irqs_disabled())
34              hard_local_irq_disable();
35      }
36  }
```

# 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

```
73  #define __evl_spin_lock(__lock) \
74      do { \
75          evl_disable_preempt(); \
76          raw_spin_lock(&(__lock)->_lock); \
77      } while (0)
```

关抢占、上锁

```
328 static inline void __evl_disable_preempt(void)
329 {
330     dovetail_current_state()->preempt_count++;
331 }
```

但是在EVL中调用raw\_spin\_lock之前也都会关中断

```
234 #define raw_spin_lock(lock) \
235     LOCK_ALTERNATIVES(lock, spin_lock, _raw_spin_lock(__RAWLOCK(lock)))
236
```

```
149 void __lockfunc _raw_spin_lock(raw_spinlock_t *lock)
150 {
151     __raw_spin_lock(lock);
152 }
153 EXPORT_SYMBOL(_raw_spin_lock);
```

```
139 static inline void __raw_spin_lock(raw_spinlock_t *lock)
140 {
141     preempt_disable();
142     spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
143     LOCK_CONTENTED(lock, do_raw_spin_trylock, do_raw_spin_lock);
144 }
```

# evl\_spin\_lock\_irqsave & raw\_spin\_lock\_irqsave

```
134 #define evl_spin_lock_irqsave(__lock, __flags) \
135     do { \
136         evl_disable_preempt(); \
137         raw_spin_lock_irqsave(&(__lock)->_lock, __flags); \
138     } while (0)
```

关本地中断、关抢占、上锁

```
328 static inline void __evl_disable_preempt(void)
329 {
330     dovetail_current_state()->preempt_count++;
331 }
```

```
262 #define raw_spin_lock_irqsave(lock, flags) \
263     LOCK_ALTERNATIVES(lock, spin_lock_irqsave, \
264     do { \
265         typecheck(unsigned long, flags); \
266         flags = _raw_spin_lock_irqsave(__RAWLOCK(lock)); \
267     } while (0), flags)
```

```
86 #define raw_spin_lock_irqsave(lock, flags) __LOCK_IRQSAVE(lock, flags)
```

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

```
30 #define __LOCK(lock) \
31     do { preempt_disable(); __LOCK(lock); } while (0)
```

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

```
1640 flags = hard_local_irq_save();
```

# hard\_local\_irq\_save

```
25  #define hard_local_irq_save()      native_irq_save()

47  static __always_inline void native_irq_disable(void)
48  {
49      asm volatile("cli": : : "memory");
50  }
```

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](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](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/>