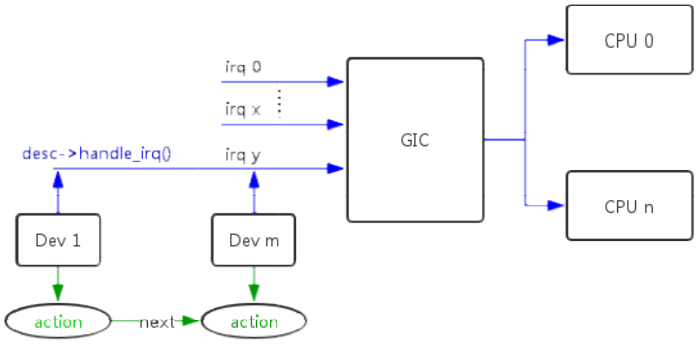
现在打开看下 request_irq 到底是怎么注册 action 的 handler 的：

```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
        const char *name, void *dev)
{
    return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}

int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                 irq_handler_t thread_fn, unsigned long irqflags,
                 const char *devname, void *dev_id)
{
    struct irqaction *action;
    struct irq_desc *desc;
    int retval;

    if (irq == IRQ_NOTCONNECTED)
        return -ENOTCONN;

    /*
     * Sanity-check: shared interrupts must pass in a real dev-ID,
     * otherwise we'll have trouble later trying to figure out
     * which interrupt is which (messes up the interrupt freeing
     * logic etc).
     *
     * Also IRQF_COND_SUSPEND only makes sense for shared interrupts and
     * it cannot be set along with IRQF_NO_SUSPEND.
     */
    if (((irqflags & IRQF_SHARED) && !dev_id) ||
        (!(irqflags & IRQF_SHARED) && (irqflags & IRQF_COND_SUSPEND)) ||
        ((irqflags & IRQF_NO_SUSPEND) && (irqflags & IRQF_COND_SUSPEND)))
        return -EINVAL; 对于共享中断，必须由devid，否则无法判断将来要给谁

    desc = irq_to_desc(irq); 通过irq号来拿到struct irq_desc
    if (!desc)
        return -EINVAL;

    if (!irq_settings_can_request(desc) ||                  1.如_IRQ_NOREQUEST标志，系统预留，退出
        WARN_ON(irq_settings_is_per_cpu_devid(desc)))       2.如IRQF_PERCPU，退出，应该用request_percpu_irq
        return -EINVAL;

    if (!handler) {
        if (!thread_fn)
            return -EINVAL;
        handler = irq_default_primary_handler; 默认handler直接返回IRQ_WAKE_THREAD
    }

    action = kzalloc(sizeof(struct irqaction), GFP_KERNEL);
    if (!action)
        return -ENOMEM;

    action->handler = handler;
    action->thread_fn = thread_fn;
    action->flags = irqflags;
    action->name = devname;
    action->dev_id = dev_id;

    retval = __setup_irq(irq, desc, action);

    if (retval) {
        irq_chip_pm_put(&desc->irq_data);
        kfree(action->secondary);
        kfree(action);
    }

#ifdef CONFIG_DEBUG_SHIRQ_FIXME
    if (!retval && (irqflags & IRQF_SHARED)) {
        /*
         * It's a shared IRQ -- the driver ought to be prepared for it
         * to happen immediately, so let's make sure....
         * We disable the irq to make sure that a 'real' IRQ doesn't
         * run in parallel with our fake.
         */
        unsigned long flags;

        disable_irq(irq);
        local_irq_save(flags);

        handler(irq, dev_id);

        local_irq_restore(flags);
        enable_irq(irq);
    }
#endif
    return retval;
} ?end request_threaded_irq ?
```

可以看到，所有"用户自定义"的 handler，都是放在 action 里面的。包括中断的上半部分 action->handler 和下半部分 action->thread_fn
另外，同时一个中断源可以多个设备共享，所以一个 desc 可以挂载多个 action，由链表结构组织起来。



这些个 action，通过__setup_irq 安装到 irq_desc 中。
下面打开看下_setup_irq()里面都完成了哪些工作。这个函数流程很长：

```
static int
__setup_irq(unsigned int irq, struct irq_desc *desc, struct irqaction *new)
{
        struct irqaction *old, **old_ptr;
        unsigned long flags, thread_mask = 0;
        int ret, nested, shared = 0;
        cpumask_var_t mask;

        if (!desc)
                return -EINVAL;

        if (desc->irq_data.chip == &no_irq_chip) //  没有 irq_chip 表示还没初始化中断控制器，退出
                return -ENOSYS;
        if (!try_module_get(desc->owner))
                return -ENODEV;

        /*
         * Check whether the interrupt nests into another interrupt
         * thread.
         */
        nested = irq_settings_is_nested_thread(desc);
        // (4.1) 判断中断是否是支持嵌套
        if (nested) {
                if (!new->thread_fn) {
                        ret = -EINVAL;
                        goto out_mput;
                }
```

```
                /*
                 * Replace the primary handler which was provided from
                 * the driver for non nested interrupt handling by the
                 * dummy function which warns when called.
                 */
                new->handler = irq_nested_primary_handler;
        } else {
                // (4.2) 判断中断是否可以被线程化
                // 如果中断没有设置 _IRQ_NOTHREAD 标志 & 强制中断线程化标志被设置 (force_irqthreads=1)
                // 强制把中断线程化：
                // new->thread_fn = new->handler;new->handler = irq_default_primary_handler;
                if (irq_settings_can_thread(desc))
                        irq_setup_forced_threading(new); //后面展开
        }

        /*
         * Create a handler thread when a thread function is supplied
         * and the interrupt does not nest into another interrupt
         * thread.
         */
        // (4.3) 如果是线程化中断，创建线程化中断对应的线程
        if (new->thread_fn && !nested) {
                struct task_struct *t;
                static const struct sched_param param = {
                        .sched_priority = MAX_USER_RT_PRIO/2,
                };

                // 创建线程
                t = kthread_create(irq_thread, new, "irq/%d-%s", irq,
                                   new->name);
                if (IS_ERR(t)) {
                        ret = PTR_ERR(t);
                        goto out_mput;
                }
                // 设置调度方式是 SCHED_FIFO
                sched_setscheduler_nocheck(t, SCHED_FIFO, &param);

                /*
                 * We keep the reference to the task struct even if
                 * the thread dies to avoid that the interrupt code
                 * references an already freed task_struct.
                 */
                get_task_struct(t);
```

```
                        // 赋值给 ->thread 成员
                        new->thread = t;
                        /*
                         * Tell the thread to set its affinity. This is
                         * important for shared interrupt handlers as we do
                         * not invoke setup_affinity() for the secondary
                         * handlers as everything is already set up. Even for
                         * interrupts marked with IRQF_NO_BALANCE this is
                         * correct as we want the thread to move to the cpu(s)
                         * on which the requesting code placed the interrupt.
                         */
                        set_bit(IRQTF_AFFINITY, &new->thread_flags);
                }

                if (!alloc_cpumask_var(&mask, GFP_KERNEL)) {
                        ret = -ENOMEM;
                        goto out_thread;
                }

                /*
                 * Drivers are often written to work w/o knowledge about the
                 * underlying irq chip implementation, so a request for a
                 * threaded irq without a primary hard irq context handler
                 * requires the ONESHOT flag to be set. Some irq chips like
                 * MSI based interrupts are per se one shot safe. Check the
                 * chip flags, so we can avoid the unmask dance at the end of
                 * the threaded handler for those.
                 */
                if (desc->irq_data.chip->flags & IRQCHIP_ONESHOT_SAFE) //如果中断控制器只支持 one shot,
则删除中断的 IRQF_ONESHOT 标志
                        new->flags &= ~IRQF_ONESHOT;

                /*
                 * The following block of code has to be executed atomically
                 */
                // (4.4) 找到最后一个 action 结构（针对共享中断来说的，共享中断的 irqaction
是通过 next 连接成链表的，所以要顺着链表找到最后一个）
                raw_spin_lock_irqsave(&desc->lock, flags);
                old_ptr = &desc->action;
                old = *old_ptr;
                if (old) {
                        /*
                         * Can't share interrupts unless both agree to and are
                         * the same type (level, edge, polarity). So both flag
```

```
                 * fields must have IRQF_SHARED set and the bits which
                 * set the trigger type must match. Also all must
                 * agree on ONESHOT.
                 */
                if (!((old->flags & new->flags) & IRQF_SHARED) ||
                    ((old->flags ^ new->flags) & IRQF_TRIGGER_MASK) ||
                    ((old->flags ^ new->flags) & IRQF_ONESHOT))
                        goto mismatch;

                /* All handlers must agree on per-cpuness */
                if ((old->flags & IRQF_PERCPU) !=
                    (new->flags & IRQF_PERCPU))
                        goto mismatch;

                /* add new interrupt at end of irq queue */
                do {
                        /*
                         * Or all existing action->thread_mask bits,
                         * so we can find the next zero bit for this
                         * new action.
                         */
                        thread_mask |= old->thread_mask;
                        old_ptr = &old->next;
                        old = *old_ptr;
                } while (old);
                // 如果有多个 action，共享标志设为 1
                shared = 1;
        }

        /*
         * Setup the thread mask for this irqaction for ONESHOT. For
         * !ONESHOT irqs the thread mask is 0 so we can avoid a
         * conditional in irq_wake_thread().
         *// 同样针对共享中断，因为有不同的 irqaction，需要一个位图来维护，要等所有 irqaction 都完成以后，才能解除对中断源的屏蔽。
        if (new->flags & IRQF_ONESHOT) {
                /*
                 * Unlikely to have 32 resp 64 irqs sharing one line,
                 * but who knows.
                 */
                if (thread_mask == ~0UL) {
                        ret = -EBUSY;
                        goto out_mask;
                }
```

```
                        /*
                         * The thread_mask for the action is or'ed to
                         * desc->thread_active to indicate that the
                         * IRQF_ONESHOT thread handler has been woken, but not
                         * yet finished. The bit is cleared when a thread
                         * completes. When all threads of a shared interrupt
                         * line have completed desc->threads_active becomes
                         * zero and the interrupt line is unmasked. See
                         * handle.c:irq_wake_thread() for further information.
                         *
                         * If no thread is woken by primary (hard irq context)
                         * interrupt handlers, then desc->threads_active is
                         * also checked for zero to unmask the irq line in the
                         * affected hard irq flow handlers
                         * (handle_[fasteoi|level]_irq).
                         *
                         * The new action gets the first zero bit of
                         * thread_mask assigned. See the loop above which or's
                         * all existing action->thread_mask bits.
                         */
                        new->thread_mask = 1 << ffz(thread_mask);

                } else if (new->handler == irq_default_primary_handler &&
                           !(desc->irq_data.chip->flags & IRQCHIP_ONESHOT_SAFE)) {
                        /*
                         * The interrupt was requested with handler = NULL, so
                         * we use the default primary handler for it. But it
                         * does not have the oneshot flag set. In combination
                         * with level interrupts this is deadly, because the
                         * default primary handler just wakes the thread, then
                         * the irq lines is reenabled, but the device still
                         * has the level irq asserted. Rinse and repeat....
                         *
                         * While this works for edge type interrupts, we play
                         * it safe and reject unconditionally because we can't
                         * say for sure which type this interrupt really
                         * has. The type flags are unreliable as the
                         * underlying chip implementation can override them.
                         */// 如果是 IRQF_ONESHOT，又没有自己定制的 irq thread，极有可能因为没有被主动清中断（电平抬起之类），导致中断风暴。这种情况返回错误。一个例外是 irq_chip 中断控制器本身带了 ONESHOT 自动清掉的能力，则 irq_chip 里面会有 IRQCHIP_ONESHOT_SAFE，这样就不返回错误
                        pr_err("Threaded irq requested with handler=NULL and !ONESHOT for irq %d\n",
                               irq);
```

```
                ret = -EINVAL;
                goto out_mask;
        }

        // (4.5) 如果是第一个 action，做一些初始化工作，清除 IRQD_IRQ_INPROGRESS
        if (!shared) {
                ret = irq_request_resources(desc);
                if (ret) {
                        pr_err("Failed to request resources for %s (irq %d) on irqchip %s\n",
                               new->name, irq, desc->irq_data.chip->name);
                        goto out_mask;
                }

                init_waitqueue_head(&desc->wait_for_threads);

                /* Setup the type (level, edge polarity) if configured: */
                if (new->flags & IRQF_TRIGGER_MASK) {
                        ret = __irq_set_trigger(desc, irq,
                                        new->flags & IRQF_TRIGGER_MASK);

                        if (ret)
                                goto out_mask;
                }

                desc->istate &= ~(IRQS_AUTODETECT | IRQS_SPURIOUS_DISABLED | \
                                  IRQS_ONESHOT | IRQS_WAITING);
                irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);

                if (new->flags & IRQF_PERCPU) {
                        irqd_set(&desc->irq_data, IRQD_PER_CPU);
                        irq_settings_set_per_cpu(desc);
                }

                if (new->flags & IRQF_ONESHOT)
                        desc->istate |= IRQS_ONESHOT;

                if (irq_settings_can_autoenable(desc))
                        irq_startup(desc, true);
                else
                        /* Undo nested disables: */
                        desc->depth = 1;

                /* Exclude IRQ from balancing if requested */
                if (new->flags & IRQF_NOBALANCING) {
```

```
                        irq_settings_set_no_balancing(desc);
                        irqd_set(&desc->irq_data, IRQD_NO_BALANCING);
                }

                // 设置中断亲和力
                /* Set default affinity mask once everything is setup */
                setup_affinity(irq, desc, mask);
        } else if (new->flags & IRQF_TRIGGER_MASK) {
                unsigned int nmsk = new->flags & IRQF_TRIGGER_MASK;
                unsigned int omsk = irq_settings_get_trigger_mask(desc);

                if (nmsk != omsk)
                        /* hope the handler works with current  trigger mode */
                        pr_warning("irq %d uses trigger mode %u; requested %u\n",
                                   irq, nmsk, omsk);
        }

        // (4.6) 将新的 action 插入到 desc 链表中
        new->irq = irq;
        *old_ptr = new;

        irq_pm_install_action(desc, new);

        /* Reset broken irq detection when installing new handler */
        desc->irq_count = 0;
        desc->irqs_unhandled = 0;

        /*
         * Check whether we disabled the irq via the spurious handler
         * before. Reenable it and give it another chance.
         */
        // (4.7) 如果中断之前被虚假 disable 了，重新 enable 中断
        if (shared && (desc->istate & IRQS_SPURIOUS_DISABLED)) {
                desc->istate &= ~IRQS_SPURIOUS_DISABLED;
                __enable_irq(desc, irq);
        }

        raw_spin_unlock_irqrestore(&desc->lock, flags);

        /*
         * Strictly no need to wake it up, but hung_task complains
         * when no hard interrupt wakes the thread up.
         */
```

```
        // (4.8) 唤醒线程化中断对应的线程，每个中断对应一个线程，而不是每个 cpu 对应一个线程。
        if (new->thread)
                wake_up_process(new->thread);

        register_irq_proc(irq, desc);
        new->dir = NULL;
        register_handler_proc(irq, new);
        free_cpumask_var(mask);

        return 0;
```

整体上注意：

1.入参 irq 当然是虚拟 irq

2.主 handler 和 threaded_fn 不能都是 NULL

3.如果配置了电平触发，且主 handler=null,irq_chip 不支持 ONESHOT_SAFE 也就是硬件 oneshot 时，注册中断应该显式的设置 IRQF_ONESHOT 告诉内核没事，否则报错。

4.启用线程化以后，主 handler 需要返回 IRQ_WAKE_THREAD

这个过程中，创建了 irq thread。这里打开看下细节：

```
// 创建线程
t = kthread_create(irq_thread, new, "irq/%d-%s", irq, new->name);
```

1.不管是哪个中断的线程，其入口函数是统一的，都是 static int irq_thread(void *data)

2.线程的命名规则是: "irq/%d-%s"，也就是 irq/中断号-中断名。

3.线程的调度设置成了 SCHED_FIFO，实时的。

```
root@:/ # ps | grep "irq/"
root          171   2       0       0       irq_thread 0000000000 S irq/389-charger
root          239   2       0       0       irq_thread 0000000000 S irq/296-PS_int-
root          247   2       0       0       irq_thread 0000000000 S irq/297-1124000
root          1415  2       0       0       irq_thread 0000000000 S irq/293-goodix_
root@a0255:/ #
```

之前说了，在

```
irqreturn_t __handle_irq_event_percpu(struct irq_desc *desc, unsigned int *flags)
{
	irqreturn_t retval = IRQ_NONE;
	unsigned int irq = desc->irq_data.irq;
	struct irqaction *action;

	record_irq_time(desc);

	for_each_action_of_desc(desc, action) {
		irqreturn_t res;

		trace_irq_handler_entry(irq, action);
		res = action->handler(irq, action->dev_id);
		trace_irq_handler_exit(irq, action, res);

		if (WARN_ONCE(!irqs_disabled(),"irq %u handler %pS enabled interrupts\n",
				irq, action->handler))
			local_irq_disable();

		switch (res) {
		case IRQ_WAKE_THREAD:
			/*
			 * Catch drivers which return WAKE_THREAD but
			 * did not set up a thread function
			 */
			if (unlikely(!action->thread_fn)) {
				warn_no_thread(irq, action);
				break;
			}

			__irq_wake_thread(desc, action);

			/* Fall through - to add to randomness */
		case IRQ_HANDLED:
			*flags |= action->flags;
			break;

		default:
			break;
		} ?end switch res ?

		retval |= res;
	}

	return retval;
} ?end __handle_irq_event_percpu ?
```

这里会唤醒线程 irq_thread：

中断对应的线程："irq/%d-%s"
kernel/irq/manage.c:

```
static int irq_thread(void *data)
{
        struct callback_head on_exit_work;
        struct irqaction *action = data;
        struct irq_desc *desc = irq_to_desc(
        irqreturn_t (*handler_fn)(struct irq_c
                        struct irqaction *action);

        if (force_irqthreads &&
test_bit(IRQTF_FORCED_THREAD,
                                &action->thre
                handler_fn = irq_forced_thread
        else
                handler_fn = irq_thread_fn;

        init_task_work(&on_exit_work, irq_th
        task_work_add(current, &on_exit_wo

        irq_thread_check_affinity(desc, actio

        //(2.1)等待唤醒和IRQTF_RUNTHREAD
        //__irq_wake_thread能唤醒
        while (!irq_wait_for_interrupt(action)
                irqreturn_t action_ret;

                irq_thread_check_affinity(desc,

                //(2.2)处理具体的action
                action_ret = handler_fn(desc, a
                if (action_ret == IRQ_HANDLE
                        atomic_inc(&desc->threa

                //(2.3)唤醒需要和本线程同步的其
                wake_threads_waitq(desc);
        }


        task_work_cancel(current, irq_thread
        return 0;
```

这里就调用到了 thread_fn，这个是每个 action 中用户自定义的函数。

看下 irq_setup_forced_threading 强制中断线程化：

```
static int irq_setup_forced_threading(struct irqaction *new)
{
	if (!force_irqthreads)
		return 0;
	if (new->flags & (IRQF_NO_THREAD | IRQF_PERCPU | IRQF_ONESHOT))
		return 0; // 这三种情况不适合强制中断线程化

	/*
	 * No further action required for interrupts which are requested as
	 * threaded interrupts already
	 */
	if (new->handler == irq_default_primary_handler)
		return 0; // 已经启用 irq_default_primary_handler，表示早就配置线程化了，不用进一步处理。

	new->flags |= IRQF_ONESHOT;

	/*
	 * Handle the case where we have a real primary handler and a
	 * thread handler. We force thread them as well by creating a
	 * secondary action.
	 */
	if (new->handler && new->thread_fn) { 如果同时有主 handler 和 thread_fn，则把主 handler 线程化以后，创建一个 secondary 的线程执行 thread_fn
		/* Allocate the secondary action */
		new->secondary = kzalloc(sizeof(struct irqaction), GFP_KERNEL);
		if (!new->secondary)
			return -ENOMEM;
		new->secondary->handler = irq_forced_secondary_handler;
		new->secondary->thread_fn = new->thread_fn;
		new->secondary->dev_id = new->dev_id;
		new->secondary->irq = new->irq;
		new->secondary->name = new->name;
	}
	/* Deal with the primary handler */ 把主 handler 变成 thread_fn，而原来的主 handler 被默认的 irq_default_primary_handler 代替。也就是强制线程化了
	set_bit(IRQTF_FORCED_THREAD, &new->thread_flags);
	new->thread_fn = new->handler;
	new->handler = irq_default_primary_handler;
	return 0;
}
```

参考文档：

《Linux Interrupt - 魅族内核团队》
《linux kernel 的中断子系统之（七）》
《linux kernel 的中断子系统之（八）》
《linux kernel 的中断子系统之（九）》
《Fundamentals of ARMv8-A》
《linux 中断子系统 - GIC 驱动源码分析 - 知乎》
Linux 5.2.5 内核代码