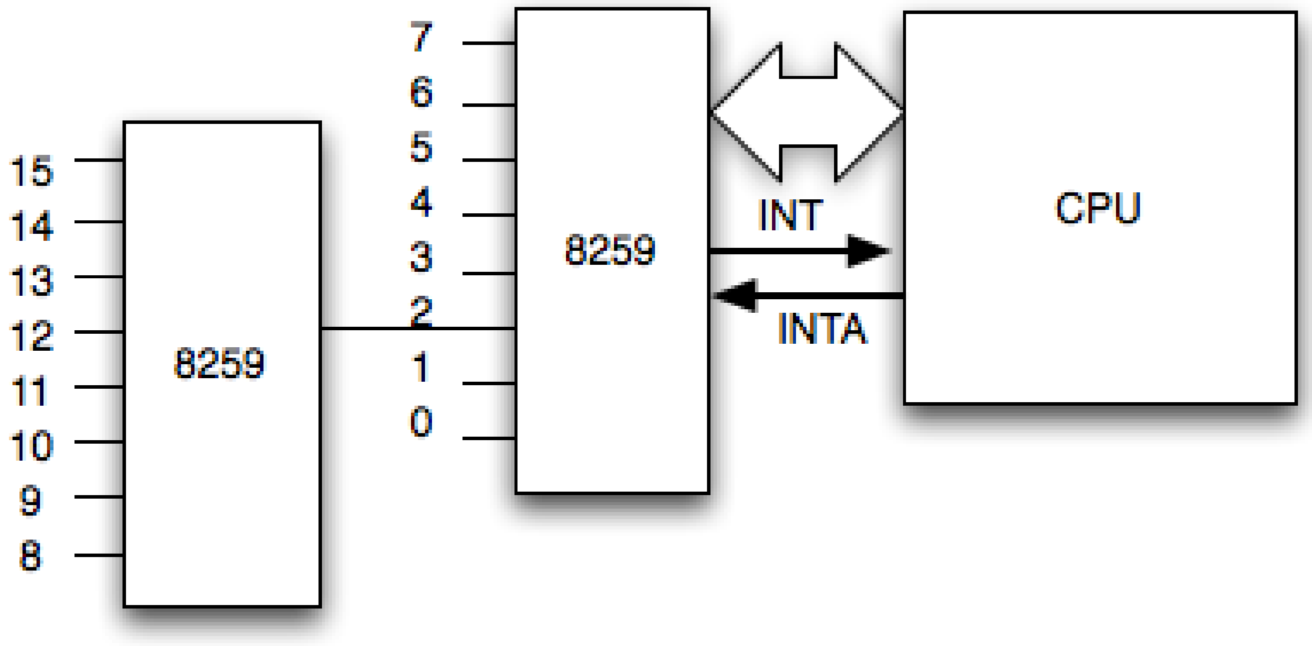
2021.09.23

彭天祥

Interrupt Controller

Interrupt Controller

Uniprocessor: Intel 8259A Interrupt Controller



Interrupt Controller

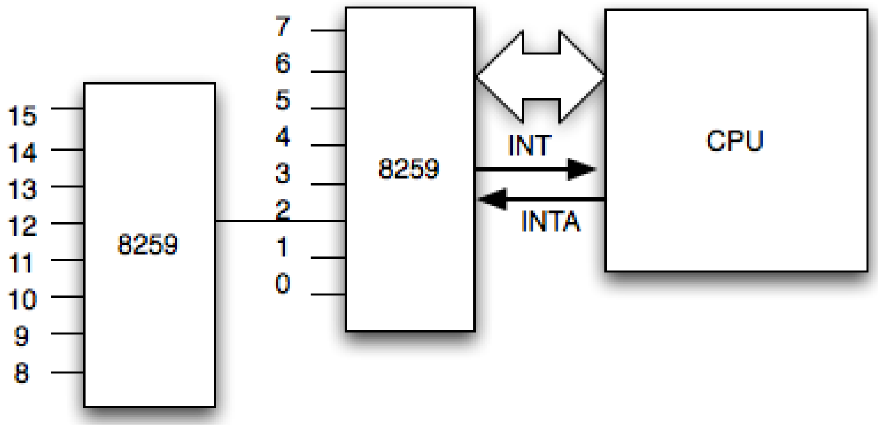
Uniprocessor: Intel 8259A Interrupt Controller

CPU side:

- INT向CPU assert中断
- 通过*cli*屏蔽所有中断信号

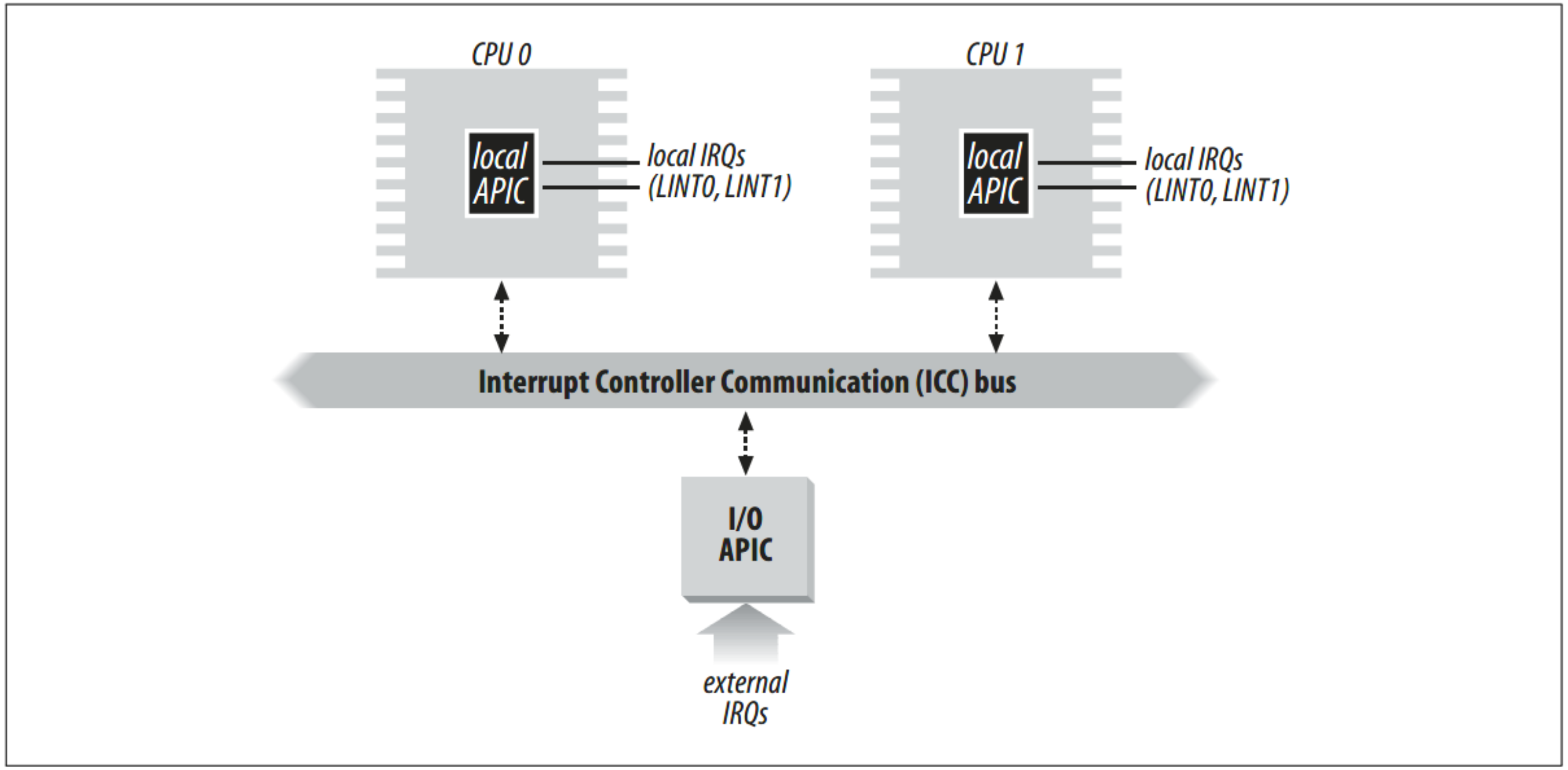
Device side:

- 多个IRQ pin
- 接收设备IRQ，映射为Linux中的IRQ(+32)，向CPU side发送中断信号
- mask特定的IRQ line来仅屏蔽一条IRQ line上的所有中断



Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)



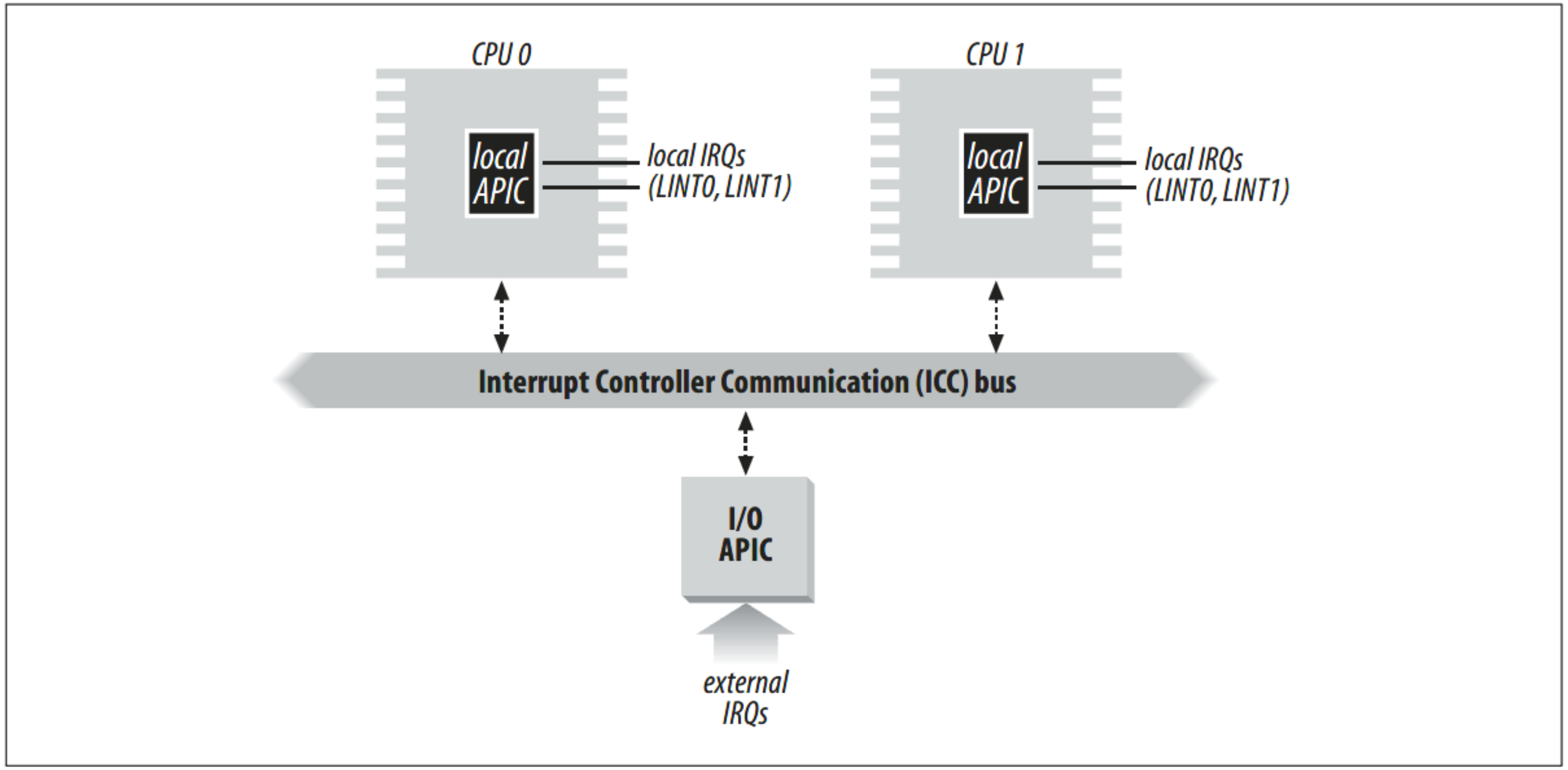
Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)

- 面向多核，每个CPU都有自己的local APIC

Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)



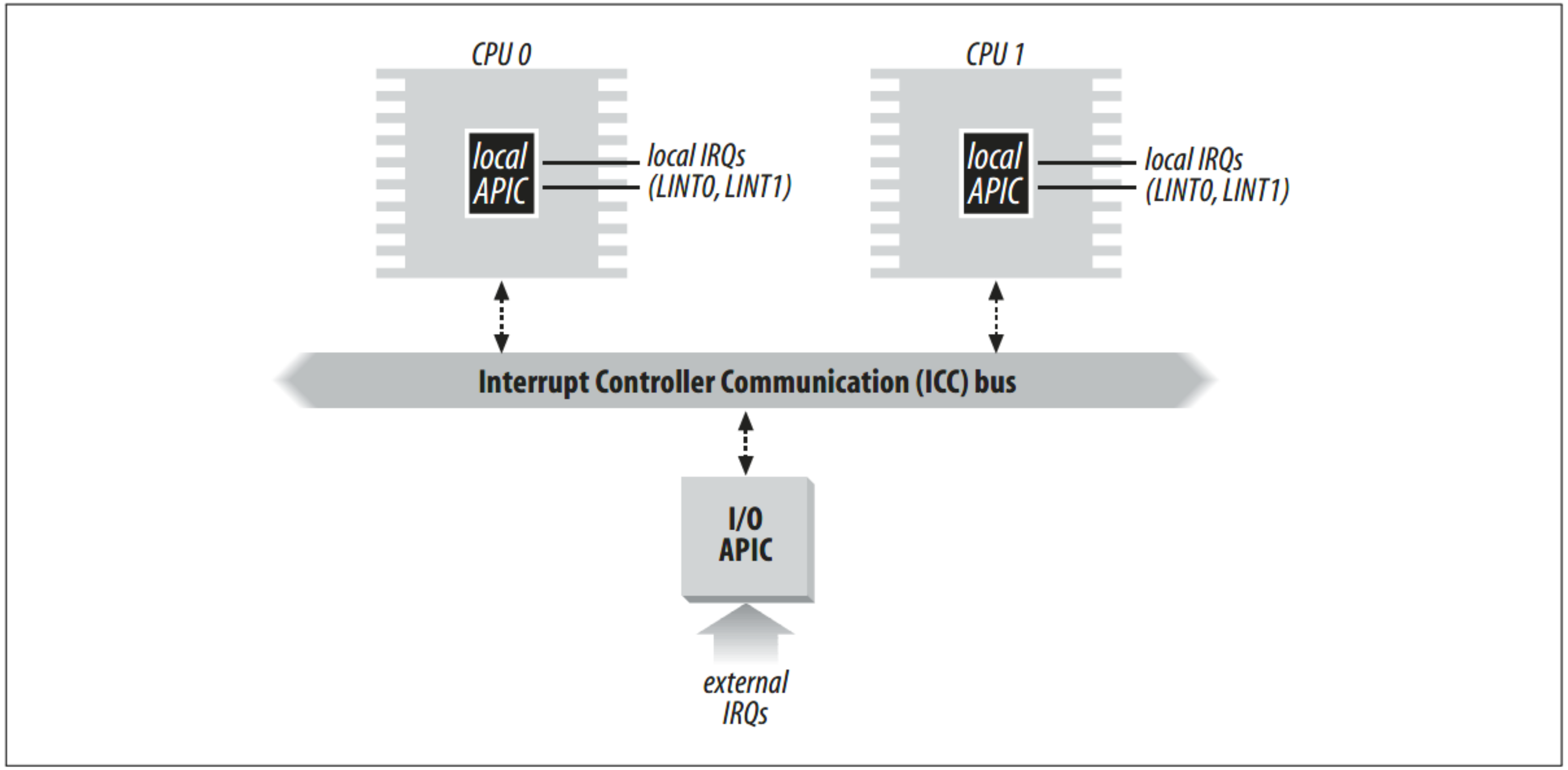
Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)

- 面向多核，每个CPU都有自己的local APIC
- **有一个I/O APIC负责接收所有设备信号，通过总线连接到所有核心的local APIC**

Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)



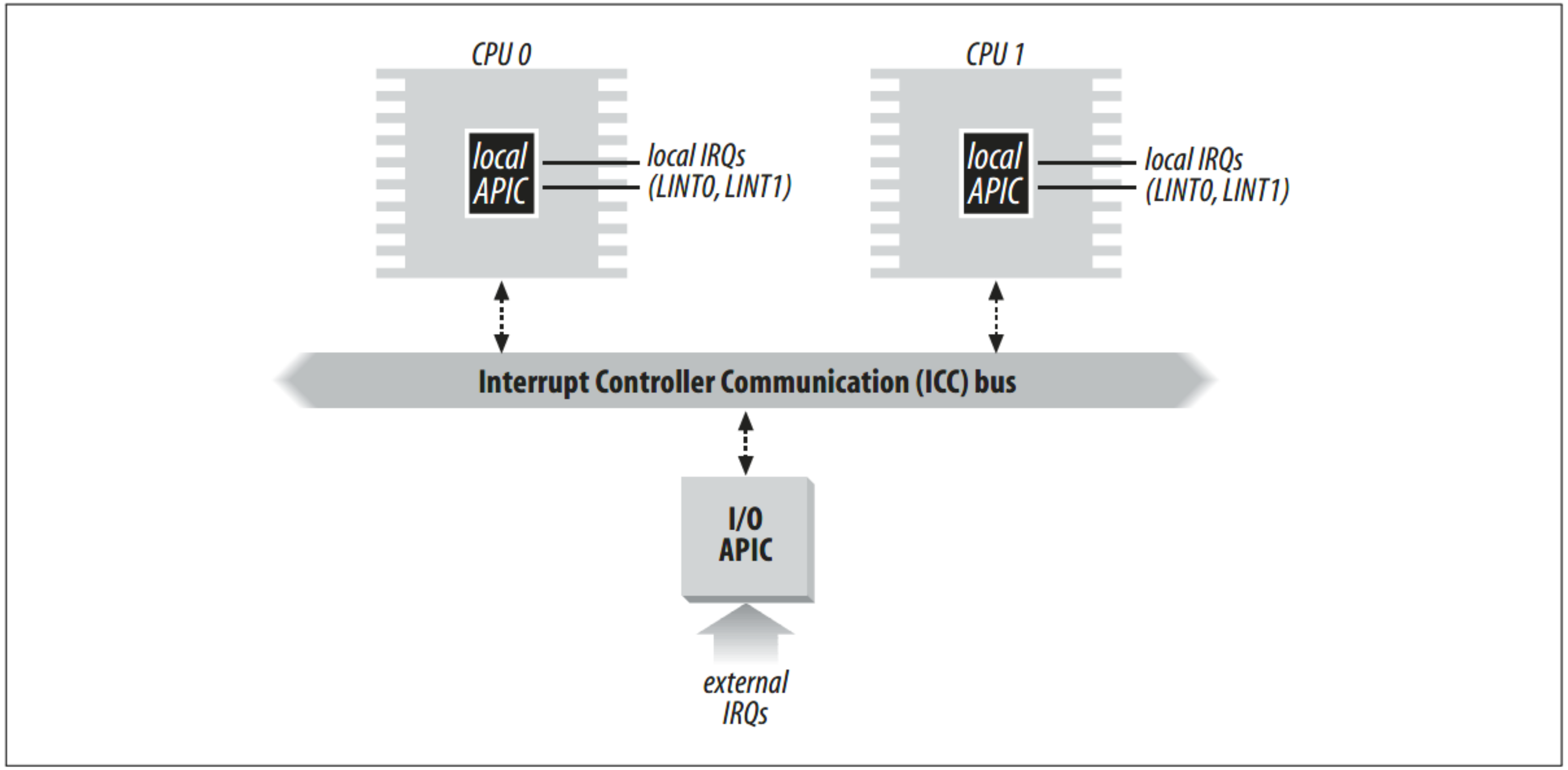
Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)

- 面向多核，每个CPU都有自己的local APIC
- 有一个I/O APIC负责接收所有设备信号，通过总线连接到所有核心的local APIC
    - **包含24个IRQ pin，以及一个24项的Interrupt Redirection Table**
        - **优先级**
        - **目标CPU**
        - **hardware IRQ到software IRQ的映射**
        - **选择目标CPU的策略（动态/静态）**

Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)



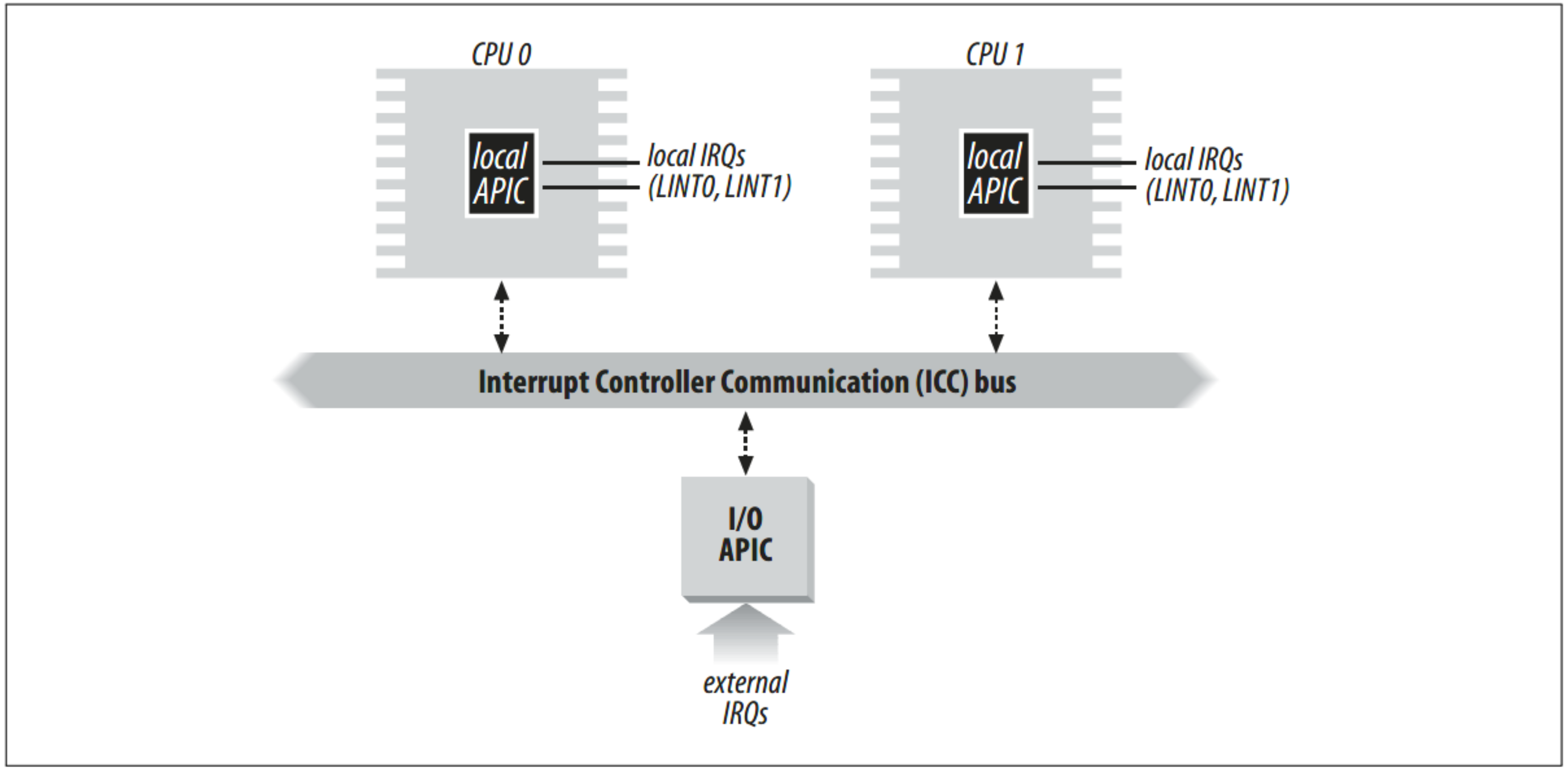
Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)

- 面向多核，每个CPU都有自己的local APIC
- 有一个I/O APIC负责接收所有设备信号，通过总线连接到所有核心的local APIC
    - 包含24个IRQ pin，以及一个24项的Interrupt Redirection Table
        - 优先级
        - 目标CPU
        - hardware IRQ到software IRQ的映射
        - 选择目标CPU的策略（动态/静态）
- 转发策略：
    - **静态：根据Redirection Table，可以一个/一些/广播**

Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)



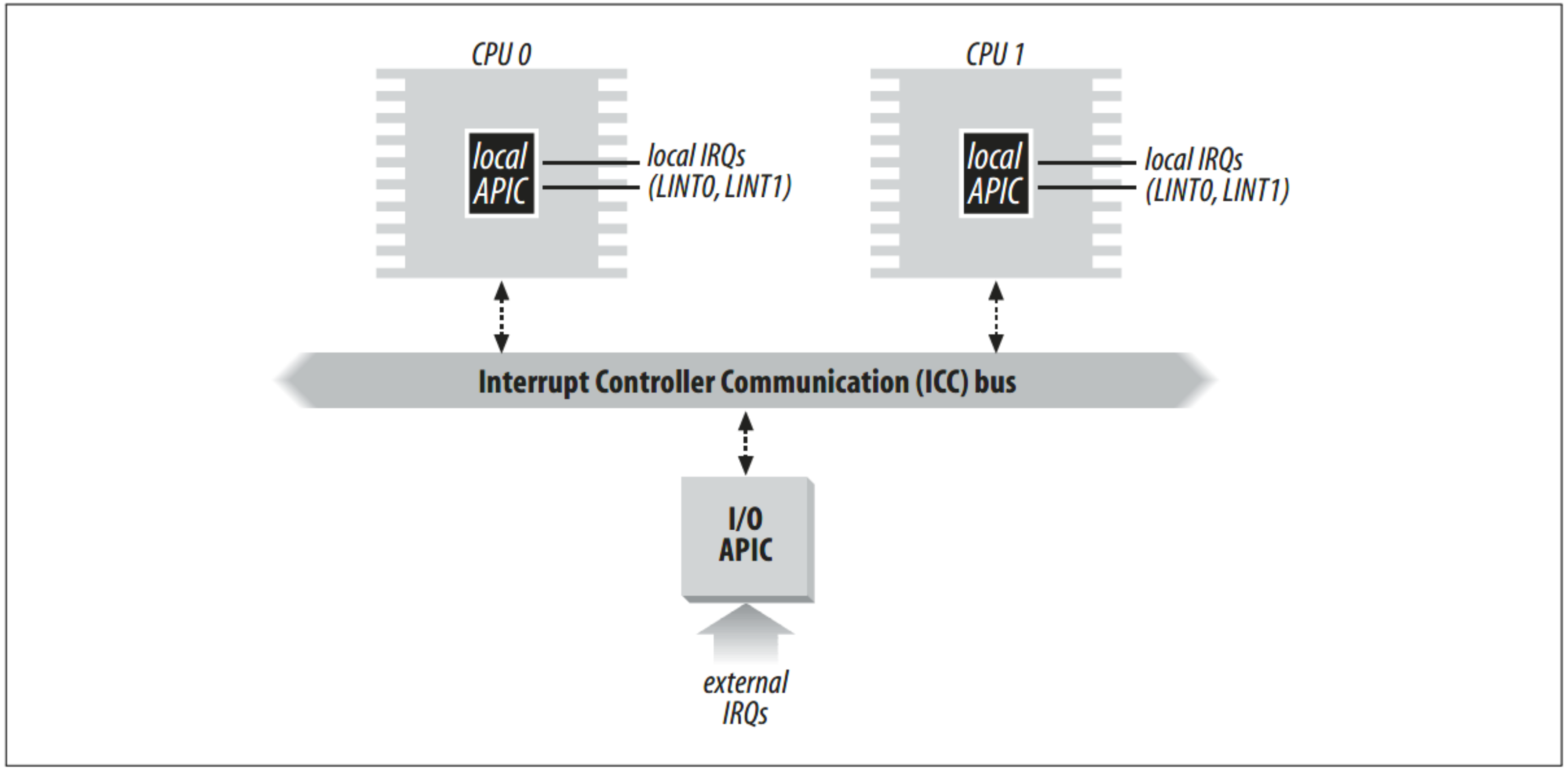
Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)

- 面向多核，每个CPU都有自己的local APIC
- 有一个I/O APIC负责接收所有设备信号，通过总线连接到所有核心的local APIC
    - 包含24个IRQ pin，以及一个24项的Interrupt Redirection Table
        - 优先级
        - 目标CPU
        - hardware IRQ到software IRQ的映射
        - 选择目标CPU的策略（动态/静态）
- 转发策略：
    - 静态：根据Redirection Table，可以一个/一些/广播
    - **动态：每个Local APIC有一个硬件task priority register，希望os每次reschedule的时候更新，这样每个dynamic IRQ到来时分发到最低优先级的CPU，优先级相同时RR**

Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)



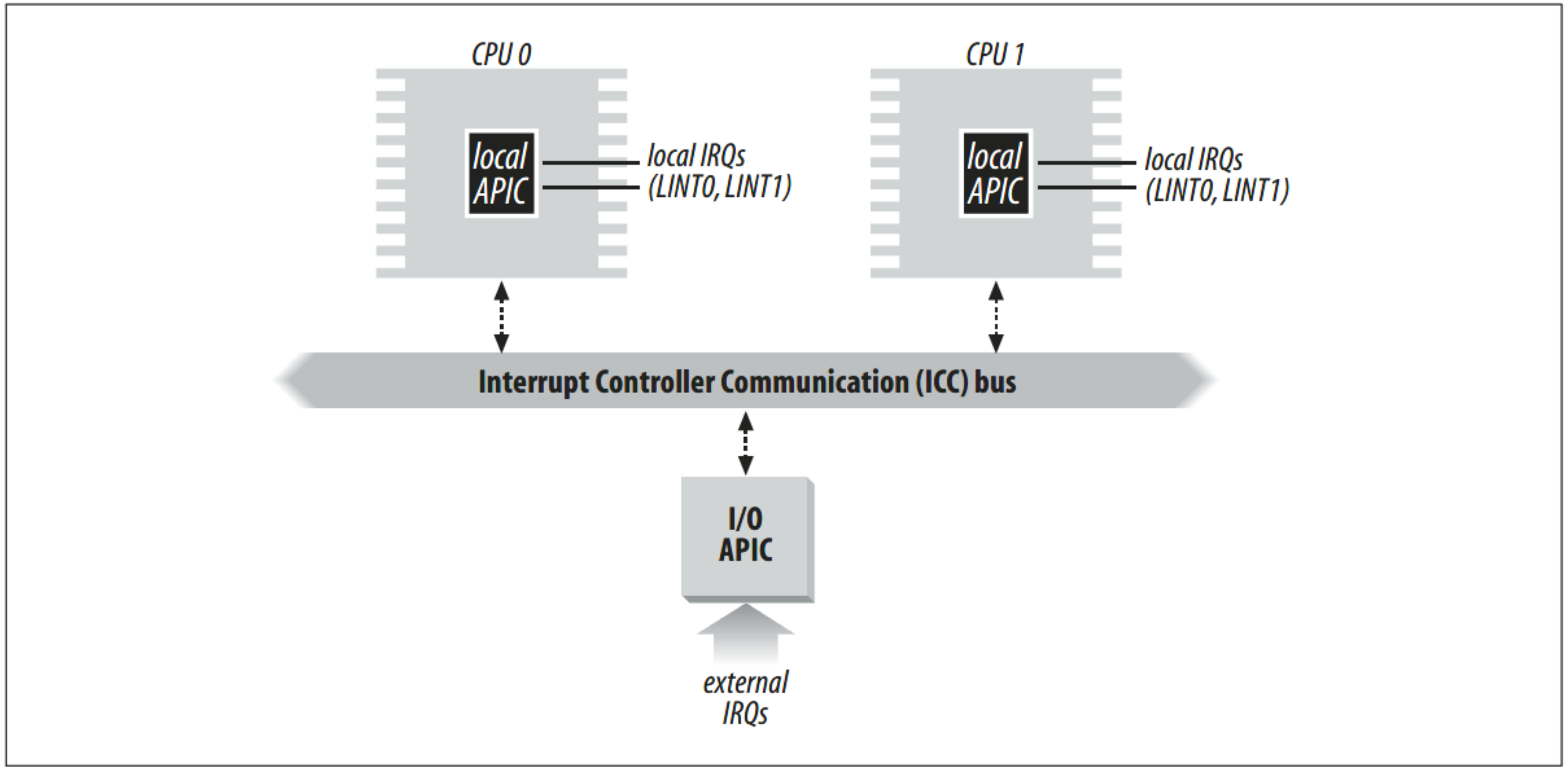
Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)

- 面向多核，每个CPU都有自己的local APIC
- 有一个I/O APIC负责接收所有设备信号，通过总线连接到所有核心的local APIC
    - 包含24个IRQ pin，以及一个24项的Interrupt Redirection Table
        - 优先级
        - 目标CPU
        - hardware IRQ到software IRQ的映射
        - 选择目标CPU的策略（动态/静态）
- 转发策略：
    - 静态：根据Redirection Table，可以一个/一些/广播
    - 动态：每个Local APIC有一个硬件task priority register，希望os每次reschedule的时候更新，这样每个dynamic IRQ到来时分发到最低优先级的CPU，优先级相同时RR
- **cli时仅屏蔽local APIC中断**

Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)



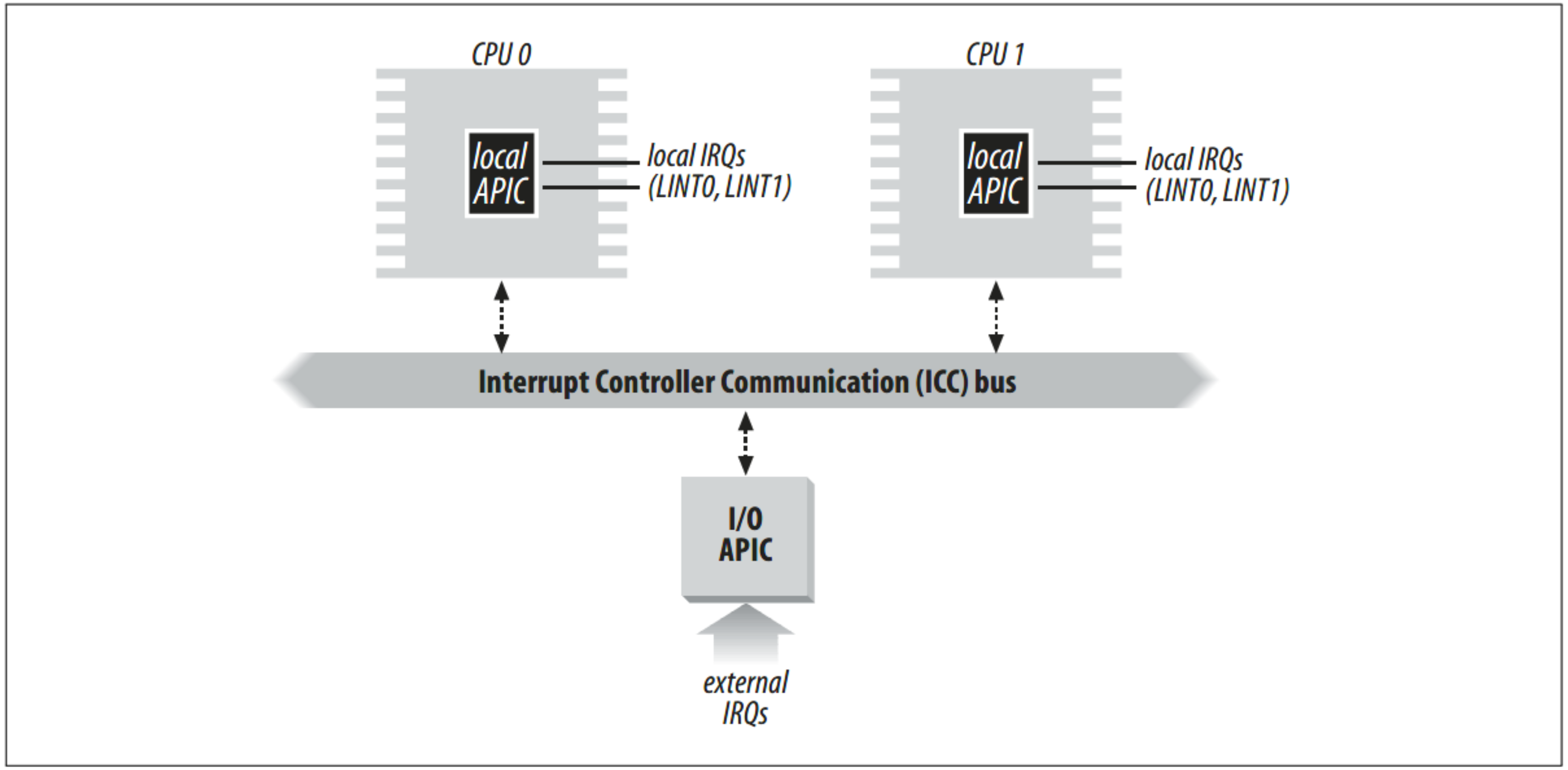
Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)

- 面向多核，每个CPU都有自己的local APIC
- 有一个I/O APIC负责接收所有设备信号，通过总线连接到所有核心的local APIC
    - 包含24个IRQ pin，以及一个24项的Interrupt Redirection Table
        - 优先级
        - 目标CPU
        - hardware IRQ到software IRQ的映射
        - 选择目标CPU的策略（动态/静态）
- 转发策略：
    - 静态：根据Redirection Table，可以一个/一些/广播
    - 动态：每个Local APIC有一个硬件task priority register，希望os每次reschedule的时候更新，这样每个dynamic IRQ到来时分发到最低优先级的CPU，优先级相同时RR
- cli时仅屏蔽local APIC中断
- **mask一条IRQ line时会使I/O APIC不再响应这个irq，从而对所有CPU屏蔽掉一条IRQ**

Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)



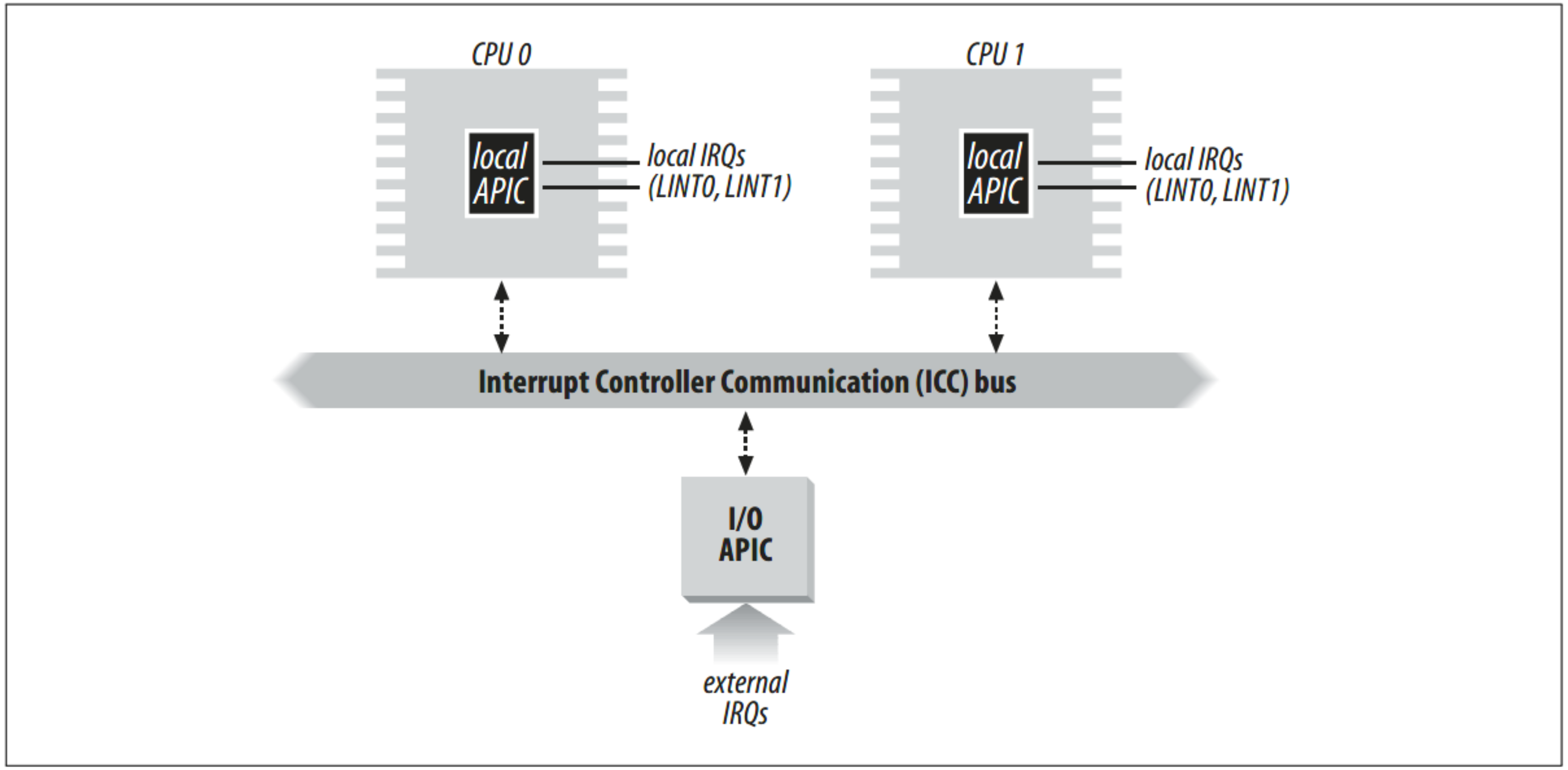
Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)

- 面向多核，每个CPU都有自己的local APIC
- 有一个I/O APIC负责接收所有设备信号，通过总线连接到所有核心的local APIC
    - 包含24个IRQ pin，以及一个24项的Interrupt Redirection Table
        - 优先级
        - 目标CPU
        - hardware IRQ到software IRQ的映射
        - 选择目标CPU的策略（动态/静态）
- 转发策略：
    - 静态：根据Redirection Table，可以一个/一些/广播
    - 动态：每个Local APIC有一个硬件task priority register，希望os每次reschedule的时候更新，这样每个dynamic IRQ到来时分发到最低优先级的CPU，优先级相同时RR
- cli时仅屏蔽local APIC中断
- mask一条IRQ line时会使I/O APIC不再响应这个irq，从而对所有CPU屏蔽掉一条IRQ
- **发送IPI：CPU将中断向量与目标Local APIC的标识符一起写入到自己Local APIC的Interrupt Command Register，从而触发IPI**

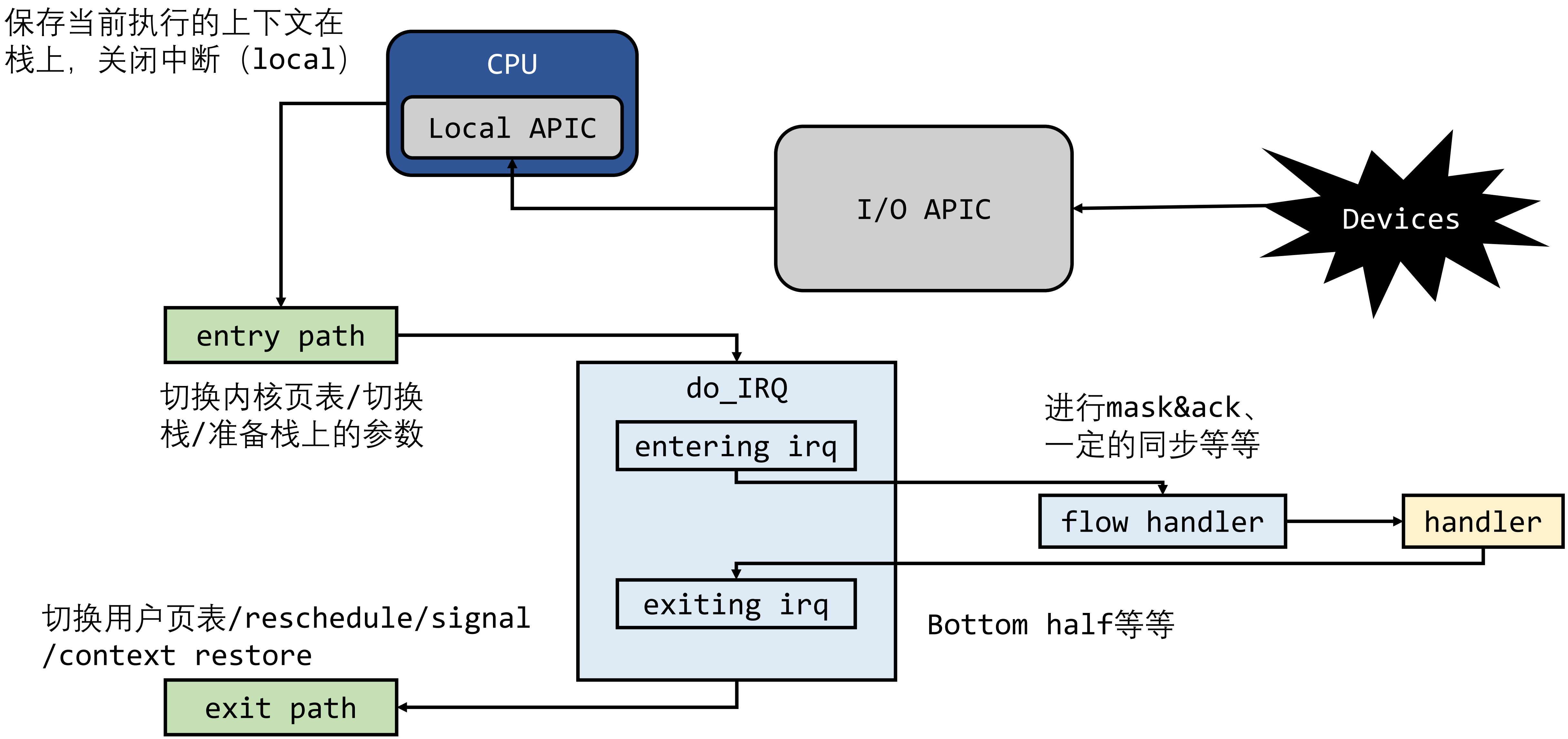
Interrupt Controller

Intel APIC(Advanced Programmable Interrupt Controller)



Handling Overview

Handling Overview



Interrupt Context

Interrupt Context

- 发生中断时我们可能
  - 打断了其他用户程序
  - 打断了其他用户程序的内核path
  - 不管怎么样，都会有一个current进程，但这个current进程与我们无关

- 处于interrupt context时我们不能sleep或者阻塞，为什么？
  - interrupt disabled
  - interrupt enabled
    - 无辜的current
    - irq masked

- 这意味着
  - 我们甚至不能使用普通的内存分配（可能sleep）
  - 将不必要的内容推后到bottom half或者interrupt thread中

Interrupt Context

- handler要保证线程安全吗?
    - 如果handler只用于单一irq
        - 不需要，因为执行handler时irq line会被mask，其他核不会执行handler
    - 如果同一handler用于多个irq
        - 需要

- handler要保证可重入吗?
    - 如果handler只用于单一irq
        - 不需要，因为执行handler时irq line会被mask，当前核不会再重新进入handler
    - 如果同一handler用于多个irq
        - 执行handler时disable local interrupt
            - 不需要，因为不会被打断，没有机会执行重新进入irq handler
        - 执行handler时enable local interrupt
            - 需要

Handler Management

Handler Management

如何在某个irq上添加/移除我们的自定义handler呢?

Kernel API:
- request_irq/free_irq

```
int request_irq(
        unsigned int irq,
        irq_handler_t handler, // typedef int (*irq_handler_t)(irq, dev_id)
        unsigned long irqflags, // 是否容忍共享irq line等等
        const char *devname, // 会显示在/proc/interrupts下
        void *dev_id); // 唯一标识符, 一般传dev结构体指针

const void *free_irq(
        unsigned int irq,
        void *dev_id);
```

Handler Management

```
struct irq_desc {
        ...
        struct irq_common_data {
                ...
                void *handler_data;
                ...
        } irq_common_data;

        struct irq_data {
                ...
                struct irq_chip *chip;

                void *chip_data;
                ...
        } irq_data;

        irq_flow_handler_t handle_irq;

        struct irq_action *action;

        unsigned int istate;

        struct depth;

        cosnt char *name;
        ...
}
```

```
struct irq_chip {
        ...
        void (*irq_ack)(struct irq_data *data);

        void (*irq_mask)(struct irq_data *data);
        ...
}
```

```
struct irqaction {
        irq_handler_t handler;

        void *dev_id;

        ...
        struct irqaction *next;
        ...

        unsigned int irq;

        unsigned int flags;

        ...
        const char *name;
        ...
}
```

Handler Management

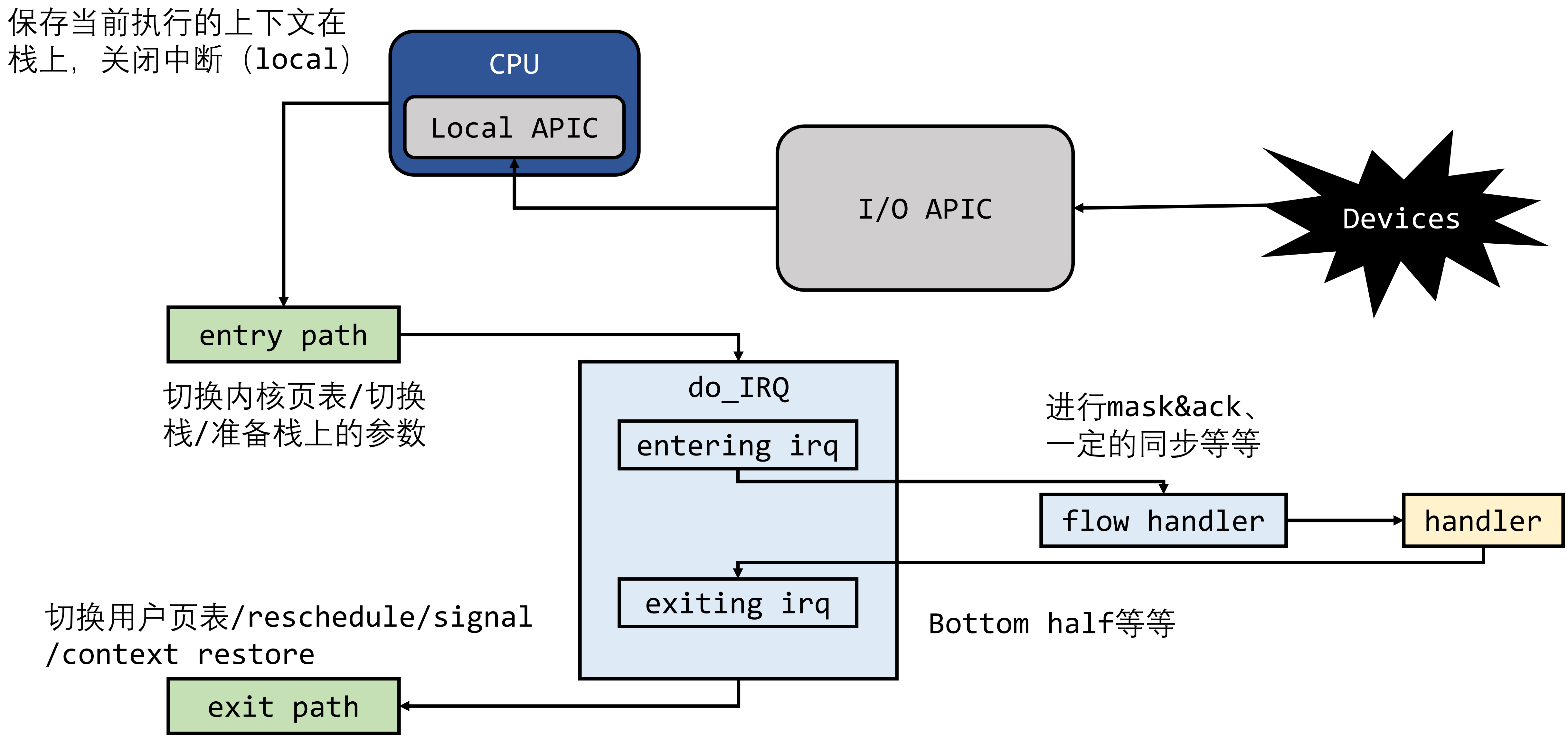
```
struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {
    [0 ... NR_IRQS-1] = {
        .handle_irq = handle_bad_irq,
        .depth      = 1,
        .lock       = __RAW_SPIN_LOCK_UNLOCKED(irq_desc->lock),
    }
};
```

```
typedef struct irq_desc* vector_irq_t[NR_VECTORS];
DECLARE_PER_CPU(vector_irq_t, vector_irq);
```

```
DEFINE_PER_CPU(vector_irq_t, vector_irq) = {
    [0 ... NR_VECTORS - 1] = VECTOR_UNUSED,
};
```

Handling Overview

do_IRQ


保存当前执行的上下文在
栈上，关闭中断（local）
CPU
Local APIC
I/O APIC
Devices
entry path
切换内核页表/切换
栈/准备栈上的参数
do_IRQ
entering irq
进行mask&ack、
一定的同步等等
flow handler
handler
exiting irq
Bottom half等等
切换用户页表/reschedule/signal
/context restore
exit path

do_IRQ

```c
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
        struct pt_regs *old_regs = set_irq_regs(regs);
        struct irq_desc * desc;
        /* high bit used in ret_from_ code  */
        unsigned vector = ~regs->orig_ax;

        entering_irq();

        /* entering_irq() tells RCU that we're not quiescent.  Check it. */
        RCU_LOCKDEP_WARN(!rcu_is_watching(), "IRQ failed to wake up RCU");

        desc = __this_cpu_read(vector_irq[vector]);
        if (likely(!IS_ERR_OR_NULL(desc))) {
                if (IS_ENABLED(CONFIG_X86_32))
                        handle_irq(desc, regs);
                else
                        generic_handle_irq_desc(desc);
        } else {
                ack_APIC_irq();

                if (desc == VECTOR_UNUSED) {
                        pr_emerg_ratelimited("%s: %d.%d No irq handler for vector\n",
                                             __func__, smp_processor_id(),
                                             vector);
                } else {
                        __this_cpu_write(vector_irq[vector], VECTOR_UNUSED);
                }
        }

        exiting_irq();

        set_irq_regs(old_regs);
        return 1;
}
```

```c
/*
 * Interrupt entry helper function.
 *
 * Entry runs with interrupts off. Stack layout at entry:
 * +----------------------------------------------------+
 * | regs->ss
 * | regs->rsp
 * | regs->eflags                                       |
 * | regs->cs
 * | regs->ip
 * +----------------------------------------------------+
 * | regs->orig_ax = ~(interrupt number)                |
 * +----------------------------------------------------+
 * | return address
 * +----------------------------------------------------+
 */
```

do_IRQ

```c
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
	struct pt_regs *old_regs = set_irq_regs(regs);
	struct irq_desc * desc;
	/* high bit used in ret_from_ code  */
	unsigned vector = ~regs->orig_ax;

	entering_irq();

	/* entering_irq() tells RCU that we're not quiescent.  Check it. */
	RCU_LOCKDEP_WARN(!rcu_is_watching(), "IRQ failed to wake up RCU");

	desc = __this_cpu_read(vector_irq[vector]);
	if (likely(!IS_ERR_OR_NULL(desc))) {
		if (IS_ENABLED(CONFIG_X86_32))
			handle_irq(desc, regs);
		else
			generic_handle_irq_desc(desc);
	} else {
		ack_APIC_irq();

		if (desc == VECTOR_UNUSED) {
			pr_emerg_ratelimited("%s: %d.%d No irq handler for vector\n",
					     __func__, smp_processor_id(),
					     vector);
		} else {
			__this_cpu_write(vector_irq[vector], VECTOR_UNUSED);
		}
	}

	exiting_irq();

	set_irq_regs(old_regs);
	return 1;
}
```

1. save/restore irq context

do_IRQ

2. 获取中断描述符

```
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
	struct pt_regs *old_regs = set_irq_regs(regs);
	struct irq_desc * desc;
	/* high bit used in ret_from_ code  */
	unsigned vector = ~regs->orig_ax;

	entering_irq();

	/* entering_irq() tells RCU that we're not quiescent.  Check it. */
	RCU_LOCKDEP_WARN(!rcu_is_watching(), "IRQ failed to wake up RCU");

	desc = __this_cpu_read(vector_irq[vector]);
	if (likely(!IS_ERR_OR_NULL(desc))) {
		if (IS_ENABLED(CONFIG_X86_32))
			handle_irq(desc, regs);
		else
			generic_handle_irq_desc(desc);
	} else {
		ack_APIC_irq();

		if (desc == VECTOR_UNUSED) {
			pr_emerg_ratelimited("%s: %d.%d No irq handler for vector\n",
					     __func__, smp_processor_id(),
					     vector);
		} else {
			__this_cpu_write(vector_irq[vector], VECTOR_UNUSED);
		}
	}

	exiting_irq();

	set_irq_regs(old_regs);
	return 1;
}
```

do_IRQ

```
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
        struct pt_regs *old_regs = set_irq_regs(regs);
        struct irq_desc * desc;
        /* high bit used in ret_from_ code  */
        unsigned vector = ~regs->orig_ax;

        entering_irq();

        /* entering_irq() tells RCU that we're not quiescent.  Check it. */
        RCU_LOCKDEP_WARN(!rcu_is_watching(), "IRQ failed to wake up RCU");

        desc = __this_cpu_read(vector_irq[vector]);
        if (likely(!IS_ERR_OR_NULL(desc))) {
                if (IS_ENABLED(CONFIG_X86_32))
                        handle_irq(desc, regs);
                else
                        generic_handle_irq_desc(desc);
        } else {
                ack_APIC_irq();

                if (desc == VECTOR_UNUSED) {
                        pr_emerg_ratelimited("%s: %d.%d No irq handler for vector\n",
                                             __func__, smp_processor_id(),
                                             vector);
                } else {
                        __this_cpu_write(vector_irq[vector], VECTOR_UNUSED);
                }
        }

        exiting_irq();

        set_irq_regs(old_regs);
        return 1;
}
```

3. 32/64-bit handling

```
/*
 * Architectures call this to let the generic IRQ layer
 * handle an interrupt.
 */
static inline void generic_handle_irq_desc(struct irq_desc *desc)
{
    desc->handle_irq(desc);
}
```

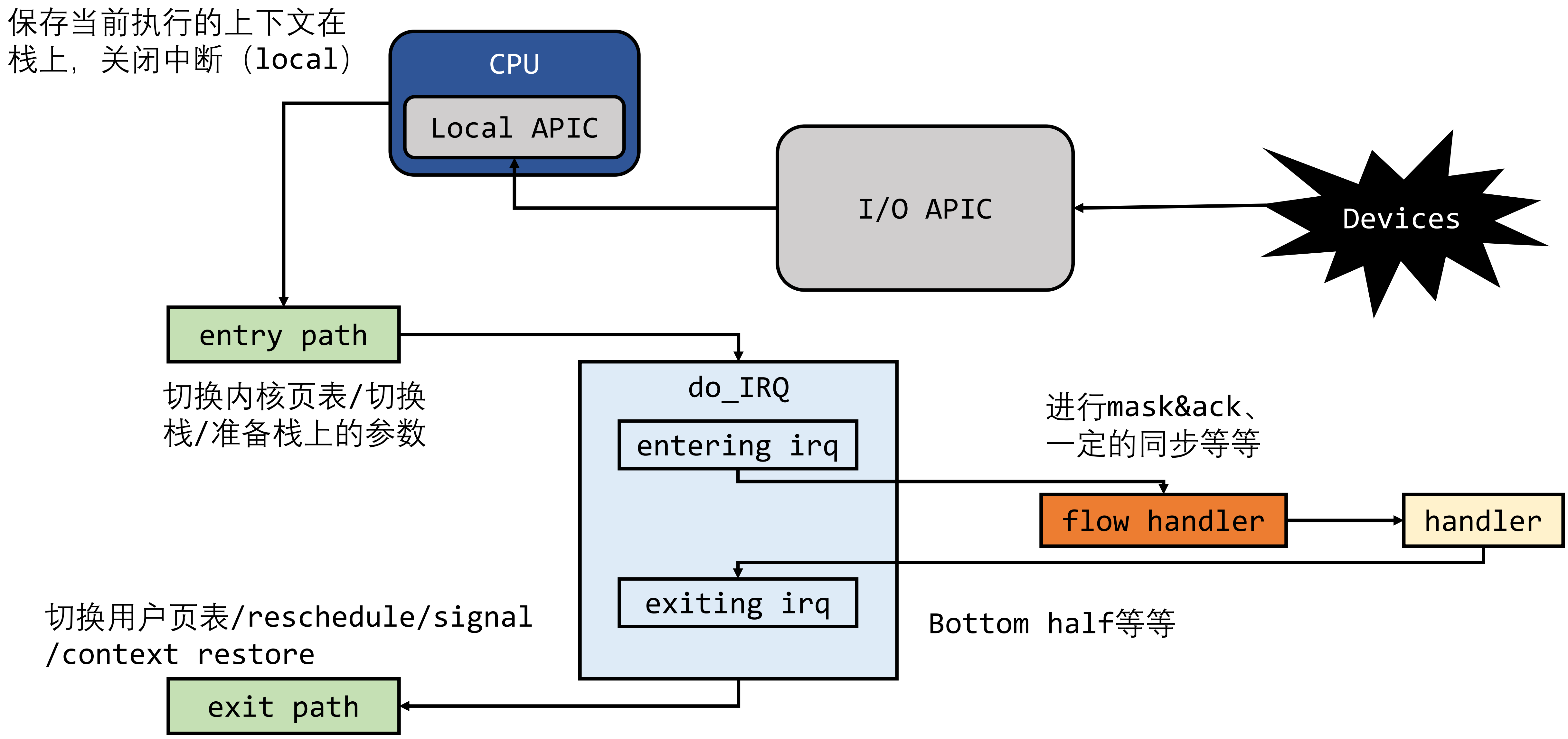
irq_flow_handler_t

```
struct irq_desc {
        ...
        struct irq_common_data {
                ...
                void *handler_data;
                ...
        } irq_common_data;

        struct irq_data {
                ...
                struct irq_chip *chip;

                void *chip_data;
                ...
        } irq_data;

        irq_flow_handler_t handle_irq;

        struct irq_action *action;

        unsigned int istate;

        struct depth;

        cosnt char *name;
        ...
}
```

```
struct irq_chip {
        ...
        void (*irq_ack)(struct irq_data *data);

        void (*irq_mask)(struct irq_data *data);
        ...
}
```

```
struct irqaction {
        irq_handler_t handler;

        void *dev_id;

        ...
        struct irqaction *next;
        ...

        unsigned int irq;

        unsigned int flags;

        ...
        const char *name;
        ...
}
```

irq_flow_handler_t



irq_flow_handler_t

```
interrupt handler types:
- Level type
- Edge type
- Simple type
- Fast EOI type
- Per CPU type
```

handle_level_irq

```c
void handle_level_irq(struct irq_desc *desc)
{
    raw_spin_lock(&desc->lock);
    mask_ack_irq(desc);

    if (!irq_may_run(desc))
        goto out_unlock;

    ...

    handle_irq_event(desc);

    ...

out_unlock:
    raw_spin_unlock(&desc->lock);
}
```

handle_level_irq

```c
irqreturn_t handle_irq_event(struct irq_desc *desc)
{
    irqreturn_t ret;

    desc->istate &= ~IRQS_PENDING;
    irqd_set(&desc->irq_data, IRQD_IRQ_INPROGRESS);
    raw_spin_unlock(&desc->lock);

    ret = handle_irq_event_percpu(desc);

    raw_spin_lock(&desc->lock);
    irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);
    return ret;
}
```

- 标志自己正在处理，release锁
- handle irq
- 重新hold锁，抹去自己正在处理的标志

handle_level_irq

```c
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;
		}

		retval |= res;
	}
```

遍历所有action（也即所有的shared handler）尝试响应irq

通过检查返回值可知中断是否有handler响应

handle_level_irq

```
void handle_level_irq(struct irq_desc *desc)
{
    raw_spin_lock(&desc->lock);
    mask_ack_irq(desc);

    if (!irq_may_run(desc))
        goto out_unlock;

    ...

    handle_irq_event(desc);

    ...

out_unlock:
    raw_spin_unlock(&desc->lock);
}
```

```
static bool irq_may_run(struct irq_desc *desc)
{
        unsigned int mask = IRQD_IRQ_INPROGRESS | IRQD_WAKEUP_ARMED;

        if (!irqd_has_set(&desc->irq_data, mask))
                return true;

        ...

}
```

```
irqreturn_t handle_irq_event(struct irq_desc *desc)
{
    irqreturn_t ret;

    desc->istate &= ~IRQS_PENDING;
    irqd_set(&desc->irq_data, IRQD_IRQ_INPROGRESS);
    raw_spin_unlock(&desc->lock);

    ret = handle_irq_event_percpu(desc);

    raw_spin_lock(&desc->lock);
    irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);
    return ret;
}
```

handle_edge_irq

```
void handle_edge_irq(struct irq_desc *desc)
{
    raw_spin_lock(&desc->lock);

    ...

    if (!irq_may_run(desc)) {
        desc->istate |= IRQS_PENDING;
        mask_ack_irq(desc);
        goto out_unlock;
    }

    ...

    /* Start handling the irq */
    desc->irq_data.chip->irq_ack(&desc->irq_data);

    do {

        ...

        if (unlikely(desc->istate & IRQS_PENDING)) {
            if (!irqd_irq_disabled(&desc->irq_data) &&
                irqd_irq_masked(&desc->irq_data))
                unmask_irq(desc);
        }

        handle_irq_event(desc);

    } while ((desc->istate & IRQS_PENDING) &&
         !irqd_irq_disabled(&desc->irq_data));

out_unlock:
    raw_spin_unlock(&desc->lock);
}
```

handle_edge_irq

core1

```
void handle_edge_irq(struct irq_desc *desc)
{
    raw_spin_lock(&desc->lock);

    ...

    /* Start handling the irq */
    desc->irq_data.chip->irq_ack(&desc->irq_data);

    do {

        ...

        if (unlikely(desc->istate & IRQS_PENDING)) {
            if (!irqd_irq_disabled(&desc->irq_data) &&
                irqd_irq_masked(&desc->irq_data))
                unmask_irq(desc);
        }

        handle_irq_event(desc);

    } while ((desc->istate & IRQS_PENDING) &&
         !irqd_irq_disabled(&desc->irq_data));

out_unlock:
    raw_spin_unlock(&desc->lock);
}
```

1. ack(而不mask)

```
irqreturn_t handle_irq_event(struct irq_desc *desc)
{
    irqreturn_t ret;

    desc->istate &= ~IRQS_PENDING;
    irqd_set(&desc->irq_data, IRQD_IRQ_INPROGRESS);
    raw_spin_unlock(&desc->lock);

    ret = handle_irq_event_percpu(desc);

    raw_spin_lock(&desc->lock);
    irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);
    return ret;
}
```

2. handle irq

3. 检查pending，处理新来的edge中断（因为中间release了锁，所以可能其他核会重入这部分代码设置pending）

handle_edge_irq

core2

```
void handle_edge_irq(struct irq_desc *desc)
{
    raw_spin_lock(&desc->lock);

    ...

    if (!irq_may_run(desc)) {
        desc->istate |= IRQS_PENDING;
        mask_ack_irq(desc);
        goto out_unlock;
    }

out_unlock:
    raw_spin_unlock(&desc->lock);
}
```

```
static bool irq_may_run(struct irq_desc *desc)
{
        unsigned int mask = IRQD_IRQ_INPROGRESS | IRQD_WAKEUP_ARMED;

        if (!irqd_has_set(&desc->irq_data, mask))
                return true;

        ...

}
```

标记pending（用于core 1的loop检测，告诉他处理新来的edge中断）

mask & ack：所有核不再接收这条irq上的中断信号，最多只能pending一个edge中断

handle_edge_irq

core1

```c
void handle_edge_irq(struct irq_desc *desc)
{
    raw_spin_lock(&desc->lock);

    ...

    /* Start handling the irq */
    desc->irq_data.chip->irq_ack(&desc->irq_data);

    do {

        ...

        if (unlikely(desc->istate & IRQS_PENDING)) {
            if (!irqd_irq_disabled(&desc->irq_data) &&
                irqd_irq_masked(&desc->irq_data))
                unmask_irq(desc);
        }

        handle_irq_event(desc);

    } while ((desc->istate & IRQS_PENDING) &&
         !irqd_irq_disabled(&desc->irq_data));

out_unlock:
    raw_spin_unlock(&desc->lock);
}
```

1. ack(而不mask)

4. 取出pending，重新unmask

2. handle irq

3. 检查pending，处理新来的edge中断（因为中间release了锁，所以可能其他核会重入这部分代码设置pending）

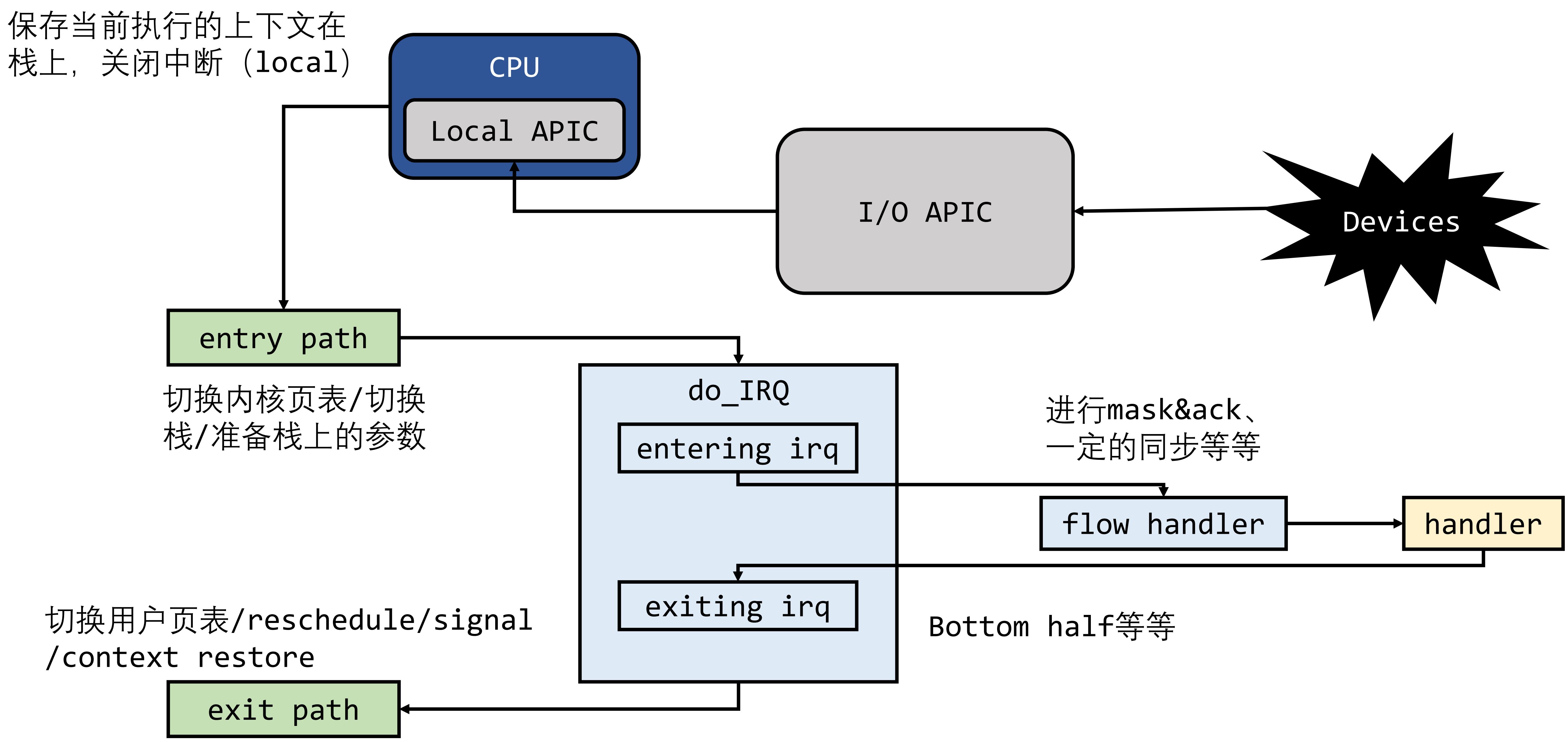
```c
irqreturn_t handle_irq_event(struct irq_desc *desc)
{
    irqreturn_t ret;

    desc->istate &= ~IRQS_PENDING;
    irqd_set(&desc->irq_data, IRQD_IRQ_INPROGRESS);
    raw_spin_unlock(&desc->lock);

    ret = handle_irq_event_percpu(desc);

    raw_spin_lock(&desc->lock);
    irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);
    return ret;
}
```

handle_edge_irq

```c
void handle_edge_irq(struct irq_desc *desc)
{
    raw_spin_lock(&desc->lock);

    ...

    if (!irq_may_run(desc)) {
        desc->istate |= IRQS_PENDING;
        mask_ack_irq(desc);
        goto out_unlock;
    }

    ...

    /* Start handling the irq */
    desc->irq_data.chip->irq_ack(&desc->irq_data);

    do {

        ...

        if (unlikely(desc->istate & IRQS_PENDING)) {
            if (!irqd_irq_disabled(&desc->irq_data) &&
                irqd_irq_masked(&desc->irq_data))
                unmask_irq(desc);
        }

        handle_irq_event(desc);

    } while ((desc->istate & IRQS_PENDING) &&
         !irqd_irq_disabled(&desc->irq_data));

out_unlock:
    raw_spin_unlock(&desc->lock);
}
```

```c
void handle_level_irq(struct irq_desc *desc)
{
    raw_spin_lock(&desc->lock);
    mask_ack_irq(desc);

    if (!irq_may_run(desc))
        goto out_unlock;

    ...

    handle_irq_event(desc);

    ...

out_unlock:
    raw_spin_unlock(&desc->lock);
}
```

irq_flow_handler_t



do_IRQ

```c
__visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
{
	struct pt_regs *old_regs = set_irq_regs(regs);
	struct irq_desc * desc;
	/* high bit used in ret_from_ code  */
	unsigned vector = ~regs->orig_ax;

	entering_irq();

	/* entering_irq() tells RCU that we're not quiescent.  Check it. */
	RCU_LOCKDEP_WARN(!rcu_is_watching(), "IRQ failed to wake up RCU");

	desc = __this_cpu_read(vector_irq[vector]);
	if (likely(!IS_ERR_OR_NULL(desc))) {
		if (IS_ENABLED(CONFIG_X86_32))
			handle_irq(desc, regs);
		else
			generic_handle_irq_desc(desc);
	} else {
		ack_APIC_irq();

		if (desc == VECTOR_UNUSED) {
			pr_emerg_ratelimited("%s: %d.%d No irq handler for vector\n",
					     __func__, smp_processor_id(),
					     vector);
		} else {
			__this_cpu_write(vector_irq[vector], VECTOR_UNUSED);
		}
	}

	exiting_irq();

	set_irq_regs(old_regs);
	return 1;
}
```

3. 32/64-bit handling

```c
/*
 * Architectures call this to let the generic IRQ layer
 * handle an interrupt.
 */
static inline void generic_handle_irq_desc(struct irq_desc *desc)
{
	desc->handle_irq(desc);
}
```

handle_irq

```c
void handle_irq(struct irq_desc *desc, struct pt_regs *regs)
{
    int overflow = check_stack_overflow();

    if (user_mode(regs) || !execute_on_irq_stack(overflow, desc)) {
        if (unlikely(overflow))
            print_stack_overflow();
        generic_handle_irq_desc(desc);
    }
}
```

```c
static inline int execute_on_irq_stack(int overflow, struct irq_desc *desc)
{
    struct irq_stack *curstk, *irqstk;
    u32 *isp, *prev_esp, arg1;

    curstk = (struct irq_stack *) current_stack();
    irqstk = __this_cpu_read(hardirq_stack_ptr);

    /*
     * this is where we switch to the IRQ stack. However, if we are
     * already using the IRQ stack (because we interrupted a hardirq
     * handler) we can't do that and just have to keep using the
     * current stack (which is the irq stack already after all)
     */
    if (unlikely(curstk == irqstk))
        return 0;

    isp = (u32 *) ((char *)irqstk + sizeof(*irqstk));

    /* Save the next esp at the bottom of the stack */
    prev_esp = (u32 *)irqstk;
    *prev_esp = current_stack_pointer;

    if (unlikely(overflow))
        call_on_stack(print_stack_overflow, isp);

    asm volatile("xchgl %%ebx,%%esp \n"
             CALL_NOSPEC
             "movl  %%ebx,%%esp \n"
             : "=a" (arg1), "=b" (isp)
             :  "0" (desc),   "1" (isp),
            [thunk_target] "D" (desc->handle_irq)
             : "memory", "cc", "ecx");
    return 1;
}
```

Nested Interrupt

IRQF_DISABLED

```c
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;
		}

		retval |= res;
	}
```

遍历所有action（也即所有的shared handler）尝试响应irq

```c
if (!(action->flags & IRQF_DISABLED))
    local_irq_enable_in_hard_irq();
```

通过检查返回值可知中断是否有handler响应

Nested Interrupt

移除IRQF_DISABLED

- 世界不是完美的，我们当然希望所有的irq handler都能够快速执行critical任务with interrupt disabled，但限于有些老旧的硬件以及开发handler人员的水平，很多handler执行是比较慢的。

- 举个例子，IDE硬盘的数据传输相比串口的UART就是慢的，由于UART只能保存一个字符，我们应该尽可能地让UART的先执行，避免过多的字符丢失。从程序上体现就是嵌套中断。

Nested Interrupt

那么为什么又要重新提出（默认）关闭嵌套中断呢?

- 设备硬件、cpu的性能都比以前高了很多（所以所有handler的执行、数据传输都更快了）

- 内核的bottom half机制日趋完善、稳定，很多工作都可以推迟到bottom half中，慢中断（允许嵌套中断）与快中断之间的界限没那么明显了

- 而默认允许嵌套中断也带来了一系列问题

    - 发生嵌套时对cache不友好

    - handler的运行时间不稳定（被嵌套打断）

    - 中断栈的溢出（即使切换到中断栈上仍然可能会溢出）

Nested Interrupt

导火索：针对中断栈溢出的一个patch

- commit的人对内核做出修改，当执行嵌套中断的时候如果发现中断栈被用了超过一半，就不管是否允许中断打开而直接关着中断执行handler

- 但是这个人也表示，这其实治标不治本，核心问题还是默认的嵌套中断：

如果一个handler写得足够好，很快就能执行完，那么执行它的时候非要开着中断（别的中断会打断、嵌套）有啥意义呢？

Nested Interrupt

一开始Linus并不赞同取消默认的嵌套中断，一些例子：

- 一些handler注定要执行很多工作，而且这是很难改变的，比如很多挑剔的嵌入式硬件以及它们孱弱的处理器

- 有些handler不开着中断没法正常运行，比如有的handler可能要循环等待jiffies被增加多少（由timer interrupt更新），关掉中断就成死循环了

- 还有一些handler可能对延迟容忍度很低，必须立马得到执行（以嵌套的方式）而不是等着前面的handler执行完（重新打开中断）

Nested Interrupt

所以一个问题是，如果我们取消了默认的嵌套中断机制，系统还能正常运行吗？

- 一个很有趣的现象是，很多内核开着lockdep checker已经运行很多年了，而它会使得handler运行在interrupt disabled状态下

- 还有就是动态tick部分的代码会在系统空闲的时候直接disable时钟中断，这也导致jiffes无法得到更新，因此很多驱动针对jiffes不更新已经做出了改善

- 另外就是realtime tree的开发者花很多精力在中断处理的延迟问题上，超时是其中最不可接收的问题之一，所以它们的代码也会针对中断关闭的问题及时做出更新

- threaded interrupt handler模块的支持使得现有的驱动代码可以很好地迁移，直接移动到handler的thread中去即可（thread中中断是打开的）

Nested Interrupt

当然，仍然还有问题等待未来的解决

- 对嵌入系统来说，handler的thread带来的响应延迟是很难接受的

- 对于实在需要打开中断的，可以借助local_irq_enable_in_hardirq在handler内部打开中断

https://lwn.net/Articles/380931/

感谢观看

彭天祥