Linux内核编程：中断

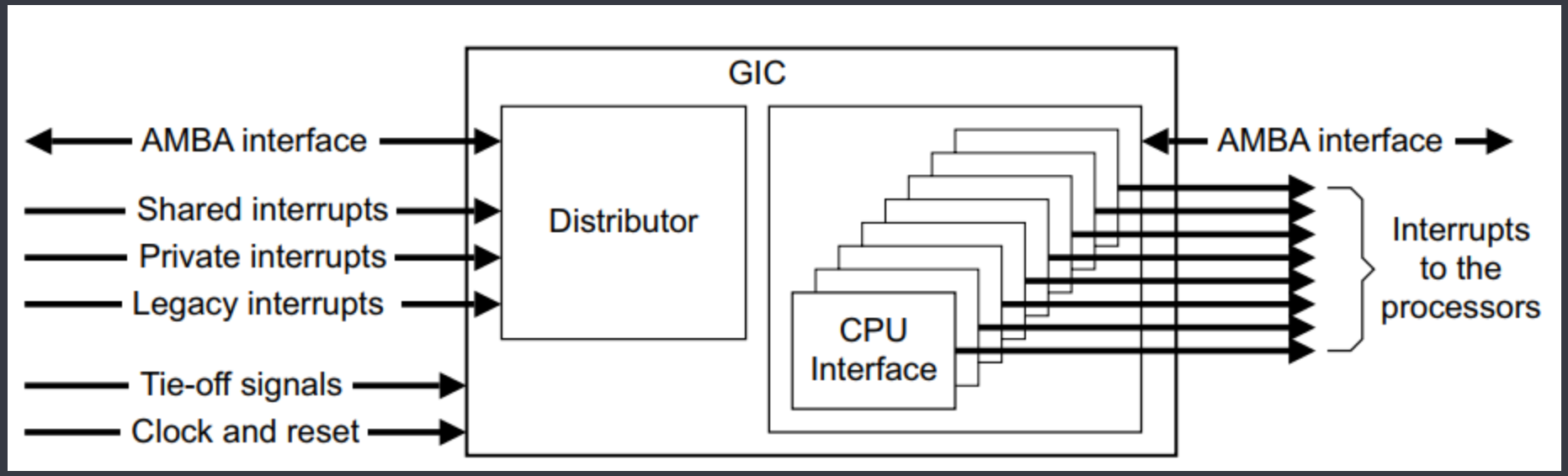
主讲：王利涛

- 中断的基本概念
  - 什么是轮询，什么是中断？
  - 异常、中断
  - 为什么处理器需要中断？

- 本期课程主要内容
  - 中断子系统：CPU、中断控制器、外设、软件
  - 中断控制器
  - 如何编写裸机中断程序
  - 如何在Linux内核驱动中编写中断处理程序
  - 软中断：softirq
  - 中断下半部：tasklet
  - 工作队列：workqueue
  - 延迟队列
  - 中断线程化：request_threaded_irq

01 中断子系统框架

- 中断子系统框架
  - CPU
  - 中断控制器
  - 外设
  - 中断向量表
  - 中断号
  - Linux内核中断子系统
  - 中断编程接口



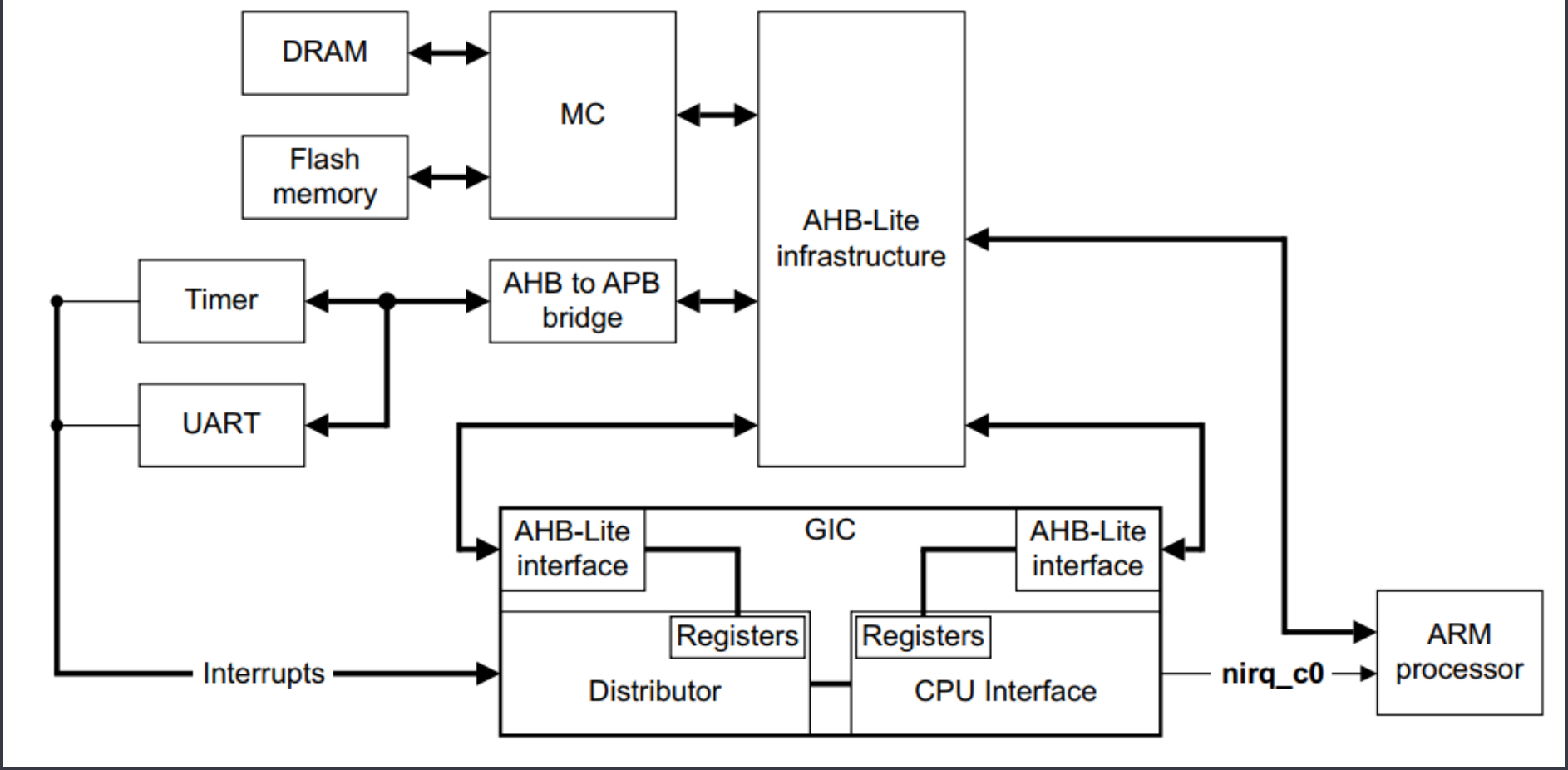
- 中断发生后…
  - 当前正在执行的程序
  - 中断向量表
  - 中断服务程序
  - 回到被打断的程序，继续执行

- 其实没这么简单...
  - 当前正在执行的程序
  - 保存被打断的上下文
  - 中断向量表
  - 找到发生中断的设备
  - 中断服务程序
  - 退出中断，调度程序运行
    - 恢复被打断的上下文
    - 回到被打断的程序，继续执行...
    - 恢复高优先级的进程的上下文
    - 切换到高优先级的程序执行...

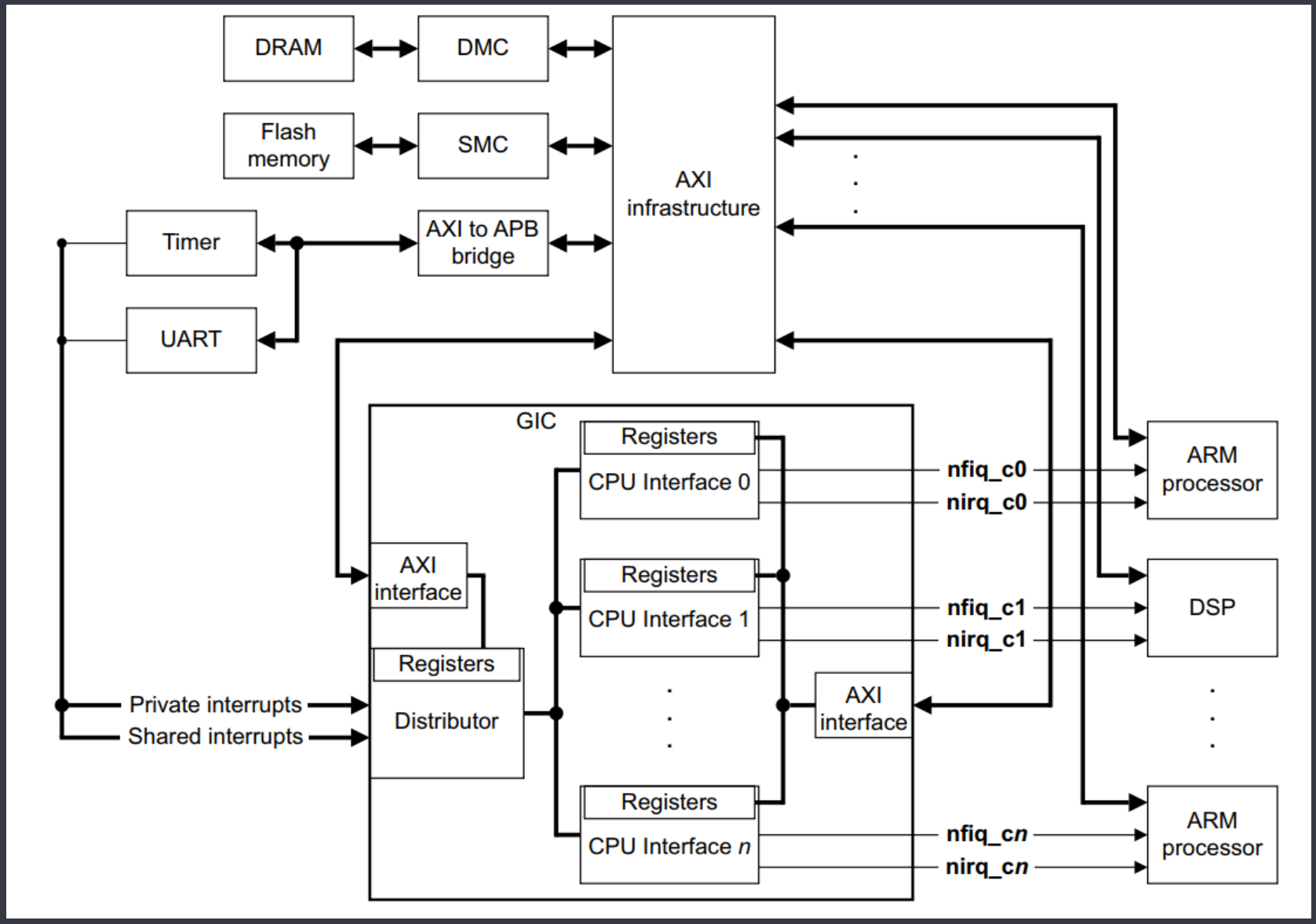
02 中断控制器：GIC

- 中断控制器的概念
  - 51单片机的中断
  - ARM SoC处理器的中断

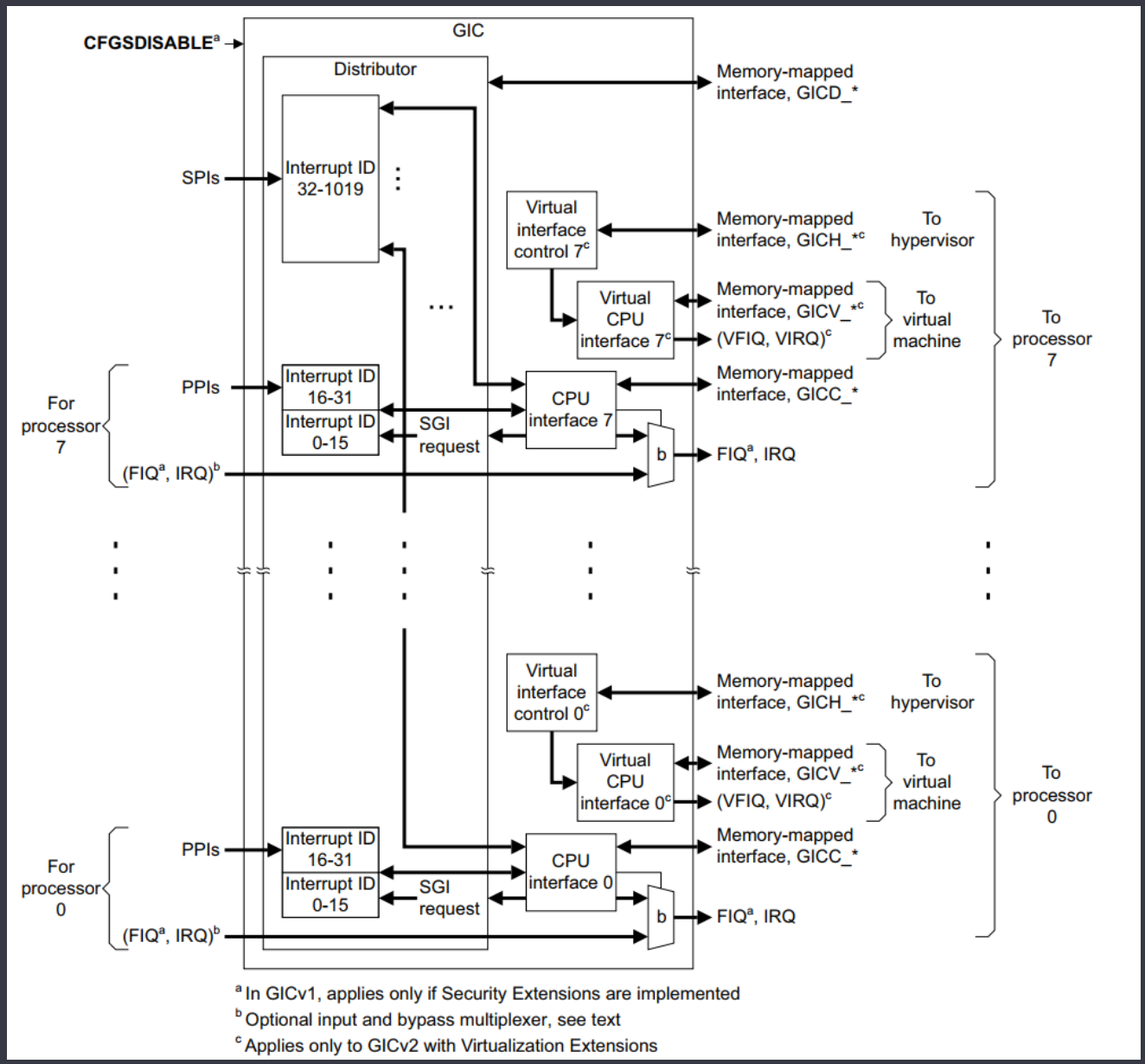
- 中断控制器的作用
  - 负责处理各种中断
  - 优先级、屏蔽、使能



• 多核下的中断控制器


DRAM
DMC
Flash memory
SMC
AXI infrastructure
Timer
AXI to APB bridge
UART
GIC
Registers
CPU Interface 0
nfiq_c0
nirq_c0
ARM processor
AXI interface
Registers
CPU Interface 1
nfiq_c1
nirq_c1
DSP
Registers
Distributor
Private interrupts
Shared interrupts
AXI interface
Registers
CPU Interface n
nfiq_cn
nirq_cn
ARM processor

• 中断分类


GIC
CFGSDISABLE[a]
Distributor
Memory-mapped interface, GICD_*
SPIs
Interrupt ID 32-1019
Virtual interface control 7[c]
Memory-mapped interface, GICH_*[c]
To hypervisor
Virtual CPU interface 7[c]
Memory-mapped interface, GICV_*[c]
(VFIQ, VIRQ)[c]
To virtual machine
To processor 7
PPIs
Interrupt ID 16-31
Interrupt ID 0-15
SGI request
CPU interface 7
Memory-mapped interface, GICC_*
b
FIQ[a], IRQ
For processor 7
(FIQ[a], IRQ)[b]
Virtual interface control 0[c]
Virtual CPU interface 0[c]
CPU interface 0
For processor 0
[a] In GICv1, applies only if Security Extensions are implemented
[b] Optional input and bypass multiplexer, see text
[c] Applies only to GICv2 with Virtualization Extensions

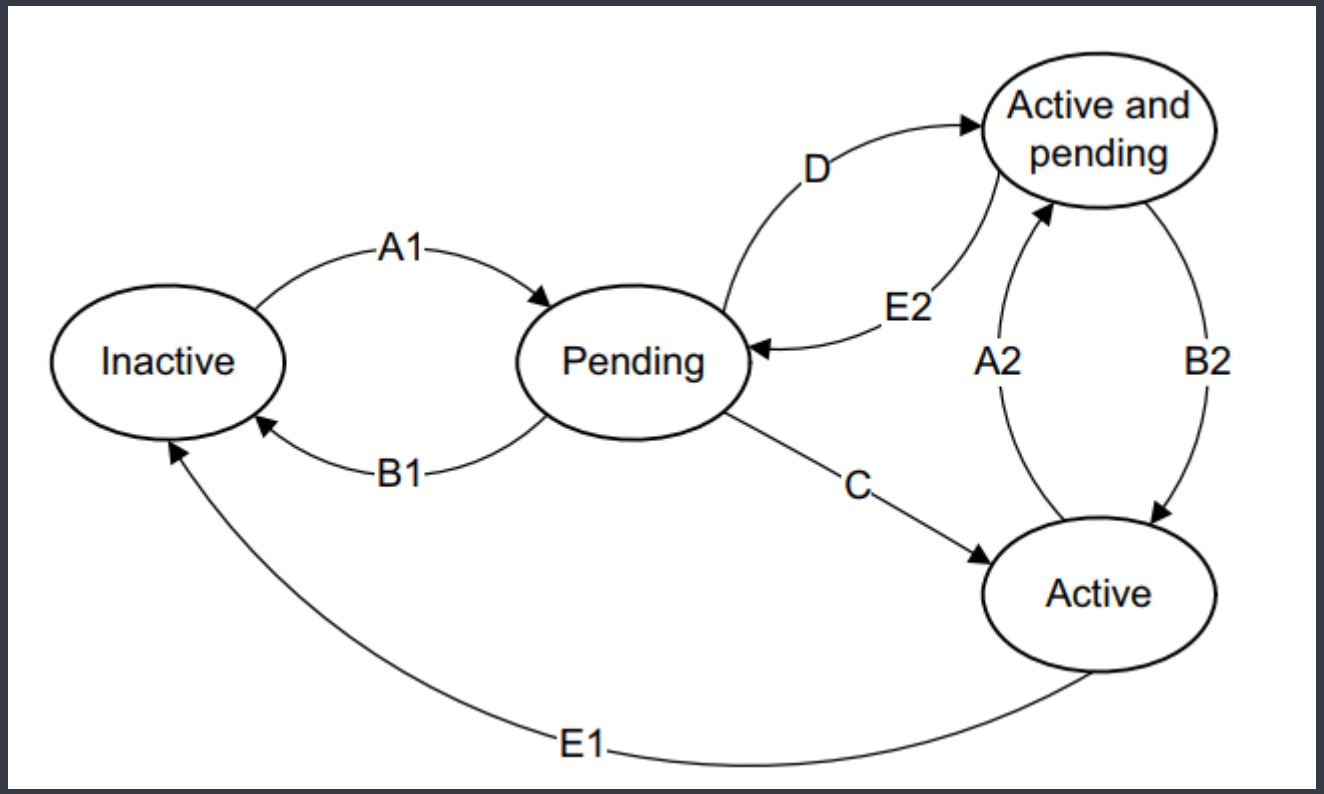
- 中断分类
  - SGI： 16 *Software Generated Interrupts*
    - 中断号ID0~ID15，用于多核之间通讯
  - PPI： 16 external *Private Peripheral Interrupts*
    - 每个core私有的中断，如本地时钟，ID16~ID31
  - SPI： *Shared Peripheral Interrupt*
    - 所有core共享的中断，可以在多个core上运行
    - 支持范围可配置：32~1019，步进32，从ID32开始
- 中断触发类型
  - edge-triggered
  - level-sensitive

- 中断号
  - HW interrupt ID
  - IRQ number
  - IRQ_domain

03 GIC控制器中断处理流程

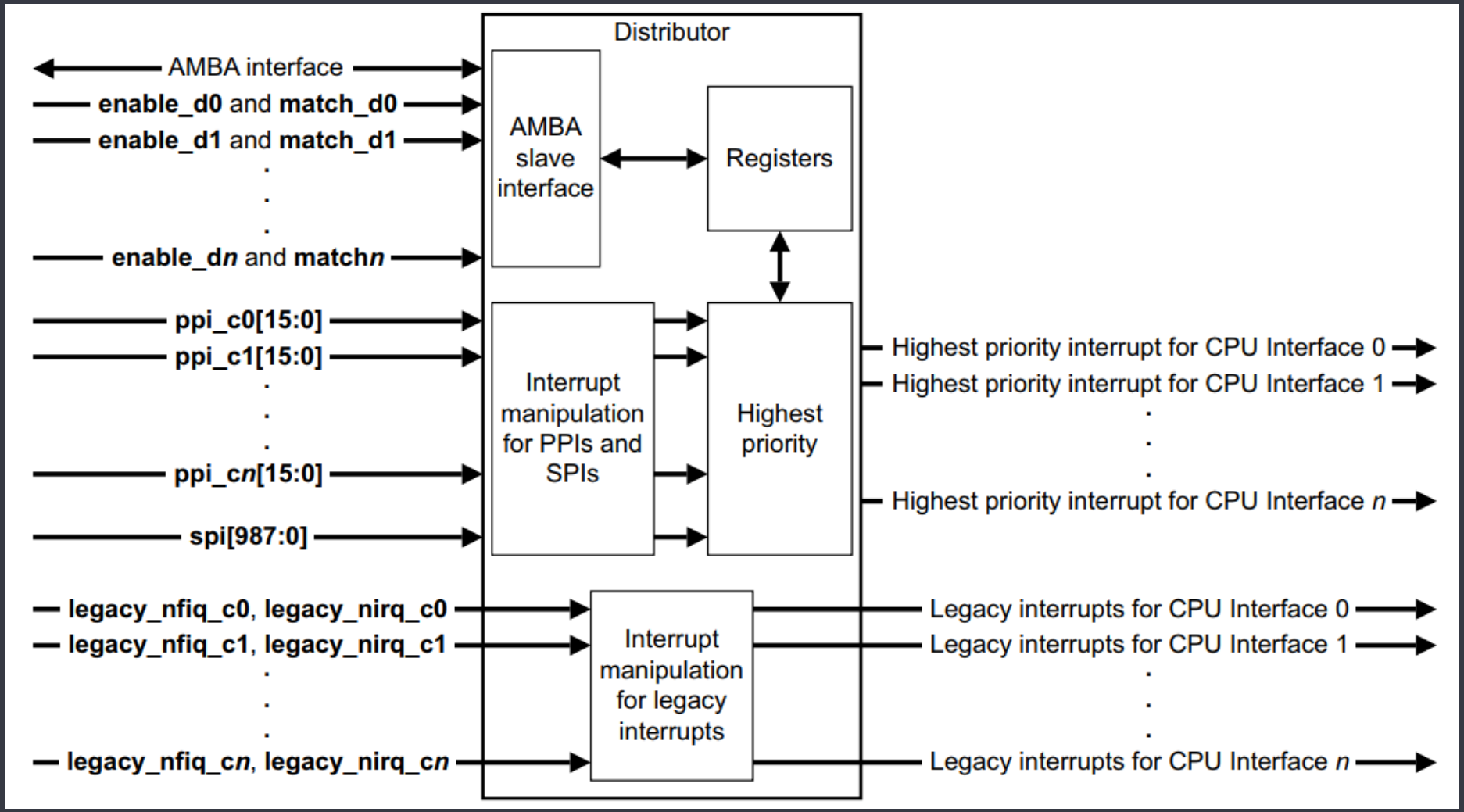
- Interrupt states
  - Inactive
  - Pending
  - Active
  - Active and pending

- Models for handling interrupts
  - 1-N model
  - N-N model



• 中断控制器内部结构

• Distributor



- 中断控制器内部结构
  - CPU interfaces
  - interrupt priority mask

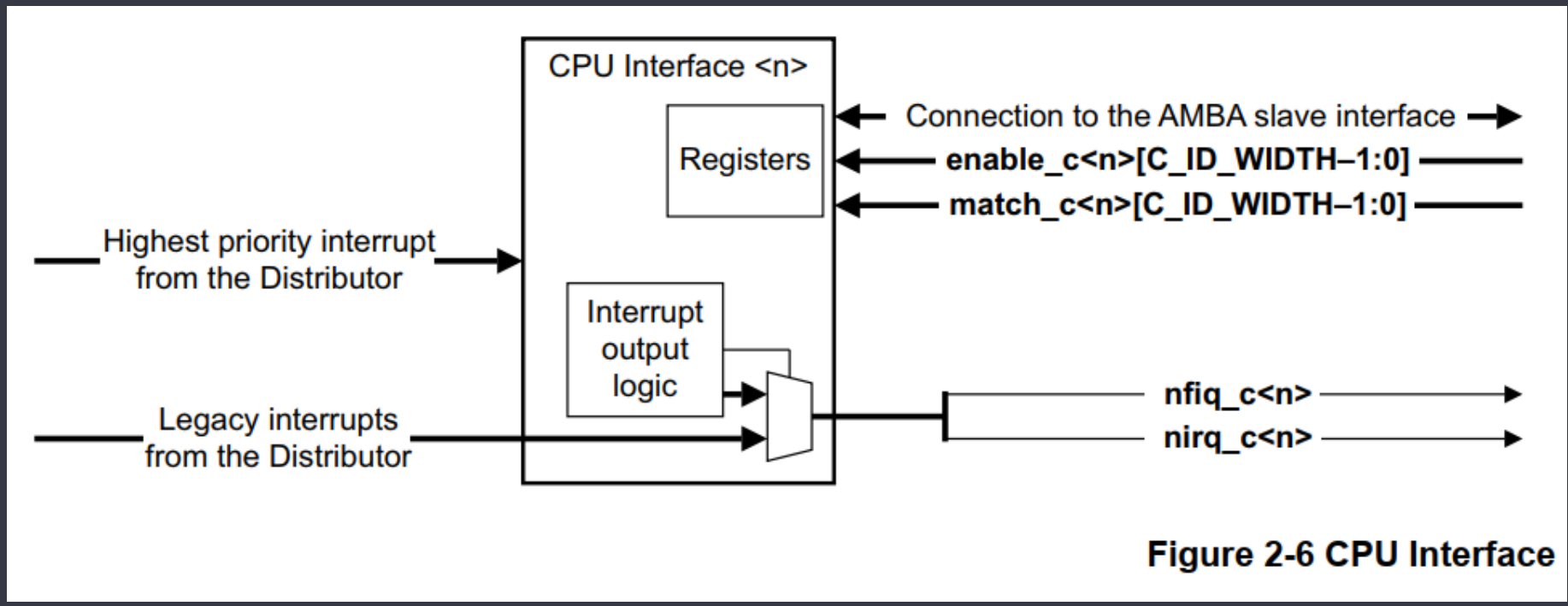


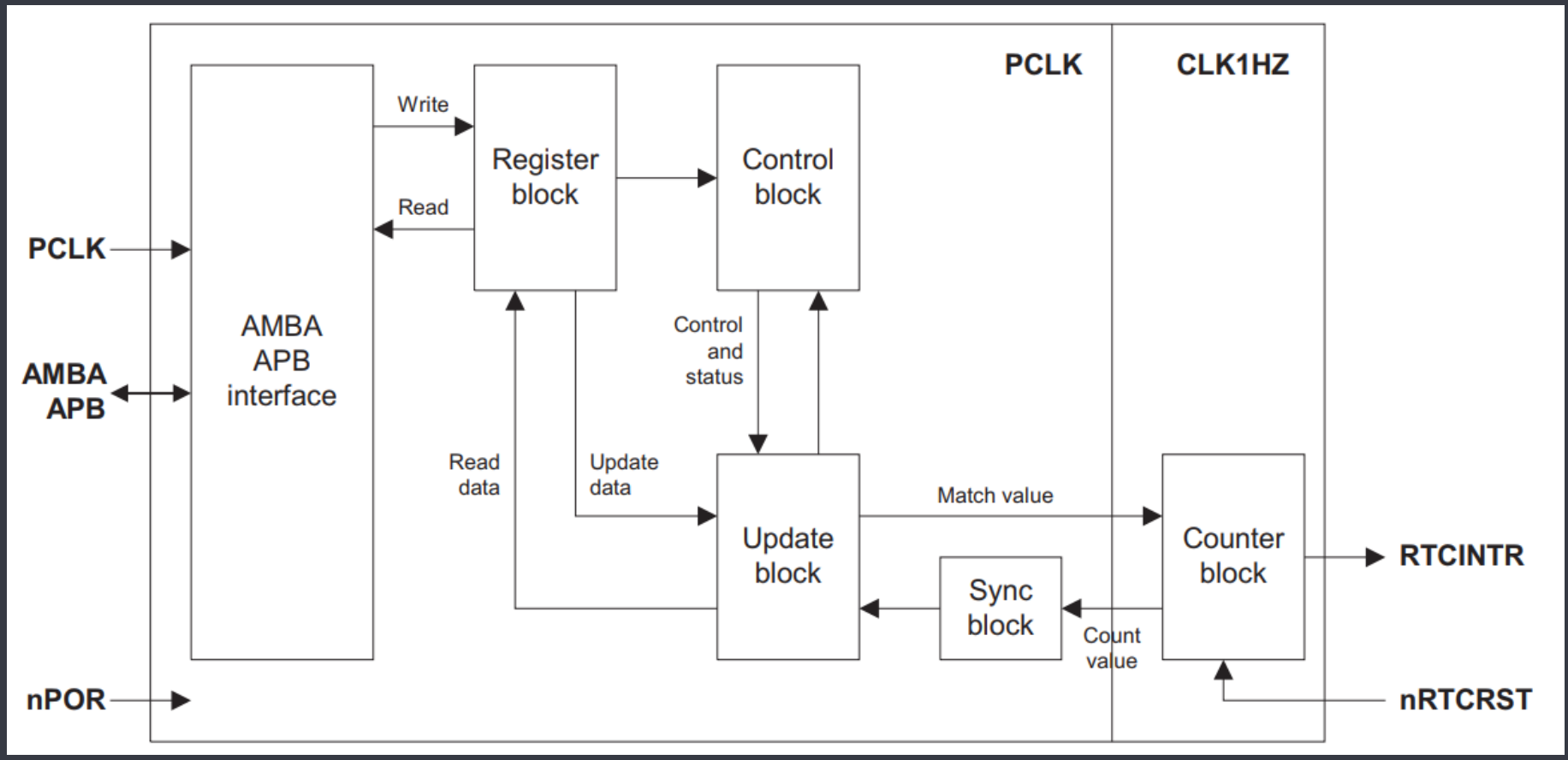
Figure 2-6 CPU Interface

- GIC 处理中断流程
  - GIC检测到使能的中断发生，将中断状态设为pending
  - GIC的仲裁器将最高优先级的pending中断发送到指定的CPU interface
  - CPU interface根据配置，将中断信号发送到CPU
  - CPU应答该中断，读取寄存器获取interrupt ID，GIC更新中断状态为active
    - » Pending → active
    - » Pending --> active and pending 中断重新产生
    - » Active → active and pending 若中断状态为active
  - CPU处理完中断后，发送EOI信号给GIC

- 中断实例
  - vexpress ARM A9开发板
  - GIC： PL390
  - timer： SP804
  - 代码： https://gitee.com/QQxiaoming/vexpress_v2p_ca9

04 编写RTC裸机中断程序

• RTC控制器介绍



05 中断函数的编写规则

- 编写中断函数需要注意的几个地方
  - 中断函数的特点
  - 无传参
  - 无返回值
  - 短小精悍

06 在Linux下编写RTC驱动中断程序

- Linux内核中断接口
  - request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev)
    - IRQF_SHARED：多设备共享一个中断号
    - IRQF_TIMER：
    - IRQF_PERCPU：
    - IRQF_NOBALANCING：
    - IRQF_IRQPOLL：
    - IRQF_ONESHOT：禁止中断嵌套
    - IRQF_NO_SUSPEND：
    - IRQF_FORCE_RESUME：
    - IRQF_NO_THREAD：
    - IRQF_EARLY_RESUME：

- Linux内核中断接口
  - request_irq()
  - free_irq()
  - HW interrupt ID和中断号
  - 物理地址与虚拟地址

07 Linux中断处理流程

- Linux中断处理流程
  - CPU硬件自动完成的
  - GIC驱动
  - Linux内核完成的
  - 用户编写的中断服务程序

08 中断上半部和下半部

- 为什么要分上半部和下半部？
  - 关中断、实时性
  - 上半部：响应中断，硬件配置，发送EOI给GIC
  - 下半部：数据复制、数据包封装转发、编解码…

- 中断下半部
  - 软中断
  - tasklet
  - 工作队列：workqueue
  - 中断线程化

09 SoftIRQ：软中断

- 软中断的相关知识
  - 编程接口
    - void open_softirq(int nr, void (*action)(struct softirq_action *));
    - void raise_softirq(unsigned int nr);
    - void raise_softirq_irqoff(unsigned int nr);
  - 编程实验：给内核添加一个软中断

```
enum
{
        HI_SOFTIRQ=0,
        TIMER_SOFTIRQ,
        NET_TX_SOFTIRQ,
        NET_RX_SOFTIRQ,
        BLOCK_SOFTIRQ,
        IRQ_POLL_SOFTIRQ,
        TASKLET_SOFTIRQ,
        SCHED_SOFTIRQ,
        HRTIMER_SOFTIRQ,
        RCU_SOFTIRQ,
        NR_SOFTIRQS
};
```

10 软中断的运行

- 软中断的实现
  - softirq 的数据结构： struct softirq_action
  - 软中断描述符数组： struct softirq_action softirq_vec[]
- 软中断的运行
  - 运行时机：中断退出后的某个时机
  - 开中断，可以被打断，不允许嵌套
  - 中断延后到线程执行： ksoftirqd

```
普通进程1正在运行，一个IRQ中断产生
    CPU自动执行部分：关闭IRQ中断、寄存器备份
    跳转到中断向量表跳执行流程：vector_irq
    irq_handler
        asm_do_IRQ->handle_IRQ->__hangle_domain_irq
        irq_enter
        generic_handle_irq：具体的外设IRQ中断处理
        irq_exit
            检测是否有pending的软中断,有则执行
            软中断是否频繁执行，有则放到softirqd执行
    退出中断，检测是否有更高优先级的进程
无，恢复被打断进程1的上下文，继续执行进程1...
有，调度更高优先级的进程2执行...
```

11 中断下半部：tasklet

- tasklet编程实战
  - 编程接口
    - DECLARE_TASKLET(name, _callback);
    - void tasklet_init(struct tasklet_struct *t,
      void (*func)(unsigned long), unsigned long data);
    - void tasklet_schedule(struct tasklet_struct *t);
    - void tasklet_hi_schedule(struct tasklet_struct *t);
    - void tasklet_kill(struct tasklet_struct *t);
  - 实验：将RTC驱动的下半部改为tasklet来实现

```
struct tasklet_struct
{
        struct tasklet_struct *next;
        unsigned long state;
        atomic_t count;
        bool use_callback;
        union {
                void (*func)(unsigned long data);
                void (*callback)(struct tasklet_struct *t);
        };
        unsigned long data;
};
```

12 tasklet的运行

- tasklet的运行
  - tasklet的实现：基于软中断
    - tasklet_vec 链表：软中断优先级6，TASKLET_SOFTIRQ
    - tasklet_hi_vec：使用软中断优先级0，HI_SOFTIRQ
  - 开中断，在中断上下文中执行
  - tasklet负载过重时，在进程上下文中执行

```
普通进程1正在运行，一个IRQ中断产生
    CPU自动执行部分：关闭IRQ中断、寄存器备份
    跳转到中断向量表跳执行流程：vector_irq
    irq_handler
        asm_do_IRQ->handle_IRQ->__hangle_domain_irq
        irq_enter
        generic_handle_irq：具体的外设IRQ中断处理
        irq_exit
            检测是否有pending的软中断,有则执行
            执行tasklet
            tasklet是否频繁执行，有则放到softirqd执行
    退出中断，检测是否有更高优先级的进程
无，恢复被打断进程1的上下文，继续执行进程1...
有，调度更高优先级的进程2执行...
```

13 中断下半部：workqueue

- 工作队列编程实战
  - 编程接口
    - DECLARE_WORK(n, f)
    - INIT_WORK(_work, _func)
    - schedule_work (struct work_struct *work);
    - cancel_work_sync (struct work_struct *work);
  - 实验：将RTC驱动的中断下半部改用工作队列实现

```
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;
#ifdef CONFIG_LOCKDEP
        struct lockdep_map lockdep_map;
#endif
};
typedef void (*work_func_t)(struct work_struct *work);
```

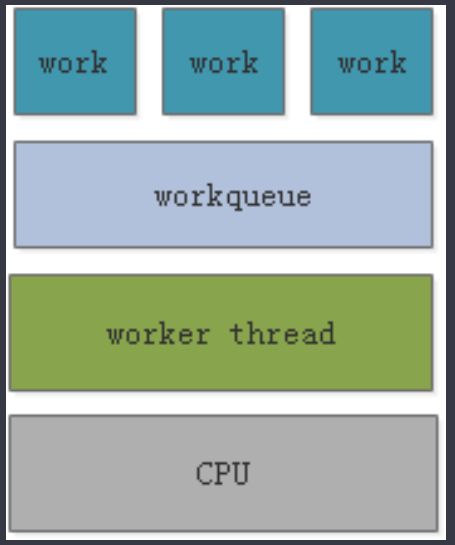
14 延迟工作队列

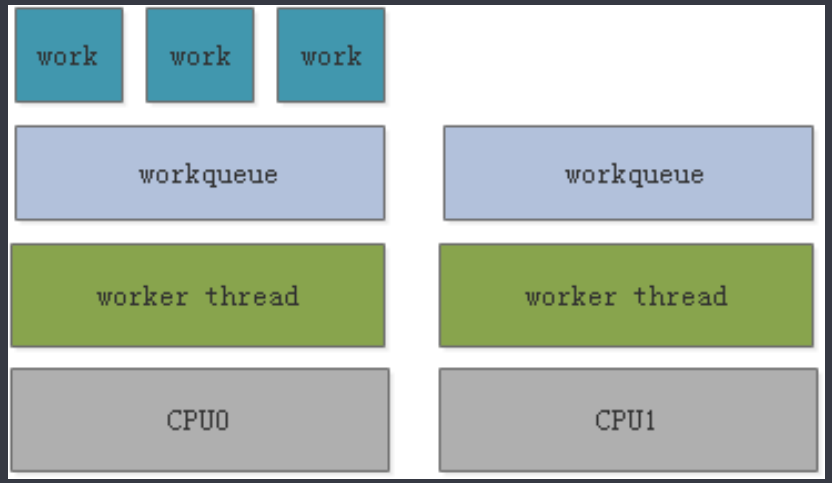
- 延迟工作队列编程实战
  - 编程接口
    - DECLARE_DELAYED_WORK(n, f)
    - INIT_DELAYED_WORK(_work, _func)
    - schedule_delayed_work(delayed_work, jiffies);
    - bool flush_delayed_work (struct delayed_work *dwork);
    - cancel_delayed_work_sync (struct work_struct *work);
  - 实验：将RTC驱动的下半部改用延迟工作队列实现

```
struct delayed_work {
        struct work_struct work;
        struct timer_list timer;
       /* target workqueue and CPU ->timer uses to queue ->work */
        struct workqueue_struct *wq;
        int cpu;
};
typedef void (*work_func_t)(struct work_struct *work);
```

15 工作队列的运行

- 工作队列的实现
  - 相关概念
    - 先struct一个work(内含中断下半部的处理函数)
    - schedule_work：将work添加workqueue
    - 执行workqueue的线程：worker thread
- 工作队列的运行
  - 单线程式
  - 多线程式





• workqueue队列的弊端

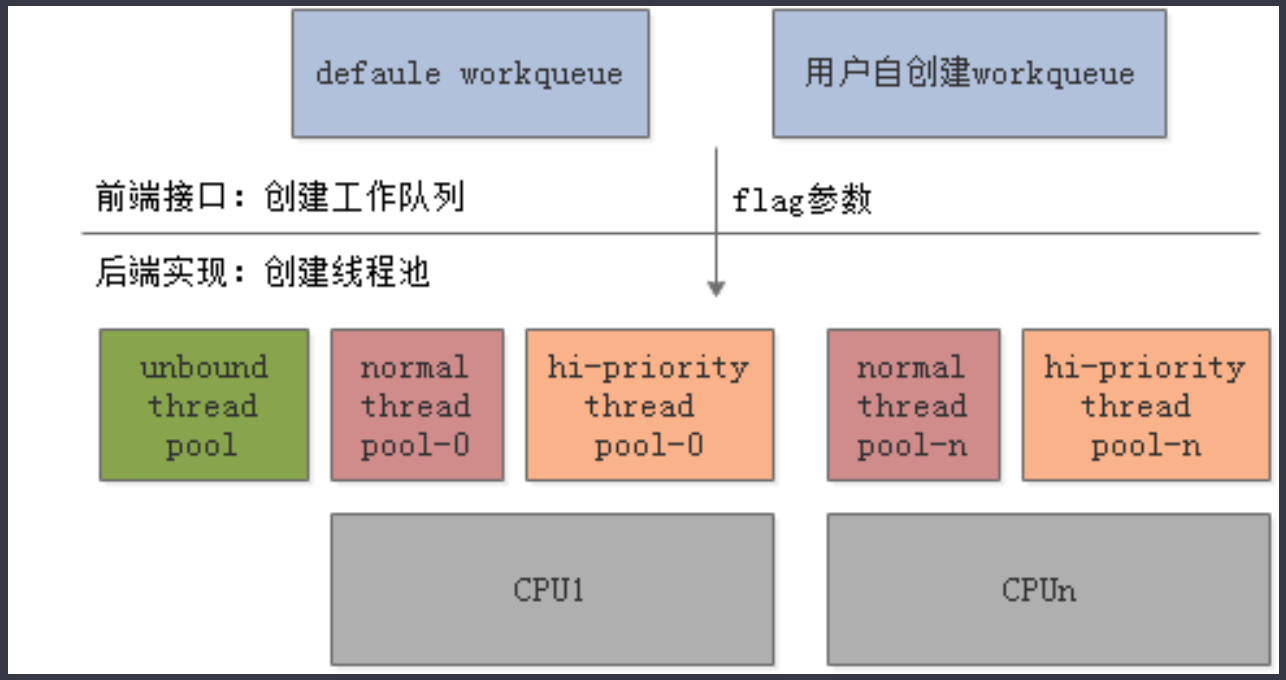
Work items w0, w1, w2 are queued to a bound wq q0 on the same CPU. w0 burns CPU for 5ms then sleeps for 10ms then burns CPU for 5ms again before finishing. w1 and w2 burn CPU for 5ms then sleep for 10ms.

TIME	EVENT
0	w0 starts and burns CPU
5	w0 sleeps
15	w0 wakes up and burns CPU
20	w0 finishes
20	w1 starts and burns CPU
25	w1 sleeps
35	w1 wakes up and finishes
35	w2 starts and burns CPU
40	w2 sleeps
50	w2 wakes up and finishes

16 CMWQ工作队列

CMWQ并发管理工作队列

- concurrency Managed Workqueue
  - 工作项：work item，添加到工作队列(workqueue)中
  - 提供两种worker线程池：bound和unbound
  - workqueue可通过flag参数绑定到指定类型的线程池执行
  - max_active：工作队列在每个CPU上并发处理的work个数



- CMWQ工作队列编程实战
  - 实验：将RTC驱动的中断下半部改用CMWQ实现
  - 编程接口

```
struct workqueue_struct * alloc_workqueue(const char *fmt, unsigned int flags,
                                          int max_active, ...);
    WQ_UNBOUND
    WQ_FREEZABLE
    WQ_MEM_RECLAIM
    WQ_HIGHPRI
    WQ_CPU_INTENSIVE
INIT_WORK (_work, _func)
bool queue_work (struct workqueue_struct *wq,struct work_struct *work);
flush_workqueue()
```

17 CMWQ工作队列的运行

• CMWQ工作队列的运行

Work items w0, w1, w2 are queued to a bound wq q0 on the same CPU. w0 burns CPU for 5ms then sleeps for 10ms then burns CPU for 5ms again before finishing. w1 and w2 burn CPU for 5ms then sleep for 10ms.

simple FIFO scheduling

TIME	EVENT
0	w0 starts and burns CPU
5	w0 sleeps
15	w0 wakes up and burns CPU
20	w0 finishes
20	w1 starts and burns CPU
25	w1 sleeps
35	w1 wakes up and finishes
35	w2 starts and burns CPU
40	w2 sleeps
50	w2 wakes up and finishes

max_active >= 3

TIME	EVENT
0	w0 starts and burns CPU
5	w0 sleeps
5	w1 starts and burns CPU
10	w1 sleeps
10	w2 starts and burns CPU
15	w2 sleeps
15	w0 wakes up and burns CPU
20	w0 finishes
20	w1 wakes up and finishes
25	w2 wakes up and finishes

• CMWQ工作队列的运行

Work items w0, w1, w2 are queued to a bound wq q0 on the same CPU. w0 burns CPU for 5ms then sleeps for 10ms then burns CPU for 5ms again before finishing. w1 and w2 burn CPU for 5ms then sleep for 10ms.

max_active = 2

TIME	EVENT
0	w0 starts and burns CPU
5	w0 sleeps
5	w1 starts and burns CPU
10	w1 sleeps
15	w0 wakes up and burns CPU
20	w0 finishes
20	w1 wakes up and finishes
20	w2 starts and burns CPU
25	w2 sleeps
35	w2 wakes up and finishes

w1 and w2 are queued to wq q1

TIME	EVENT
0	w0 starts and burns CPU
5	w0 sleeps
5	w1 and w2 start and burn CPU
10	w1 sleeps
15	w2 sleeps
15	w0 wakes up and burns CPU
20	w0 finishes
20	w1 wakes up and finishes
25	w2 wakes up and finishes

18 中断线程化：request_threaded_irq

- 中断线程化编程实战
  - 实验：将RTC驱动的中断下半部改为线程化执行
  - 编程接口

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                         irq_handler_t thread_fn,
                         unsigned long flags,
                         const char *name, void *dev);
```