中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

1924-2024 中山大學 世纪华诞 100th ANNIVERSARY SUN YAT-SEN UNIVERSITY

硬件环境与软件抽象：特权级模型与中断

SSE202/204: 操作系统原理

苏玉鑫

suyx35@mail.sysu.edu.cn

助教：龙玉丹 单诗雯 毛晨希 沈志轩 郑灿峰 胡伟峰

版权信息

- 部分内容来自：上海交通大学并行与分布式系统研究所操作系统课件
  - https://ipads.se.sjtu.edu.cn/courses/os/
- 其它参考资料：
  - 清华大学操作系统公开课
    - https://open.163.com/newview/movie/courseintro?newurl=ME1NSA351
    - 介绍标准内容，适合考研
  - 南京大学计算机软件研究所
    - http://jyywiki.cn/OS/2025/
    - https://space.bilibili.com/202224425/channel/collectiondetail?sid=192498
    - 比较有趣

大纲

- 特权级模型
- 设备与中断
  - 基本概念
  - 中断如何产生
  - 设备的内存映射
  - 异常处理

大纲

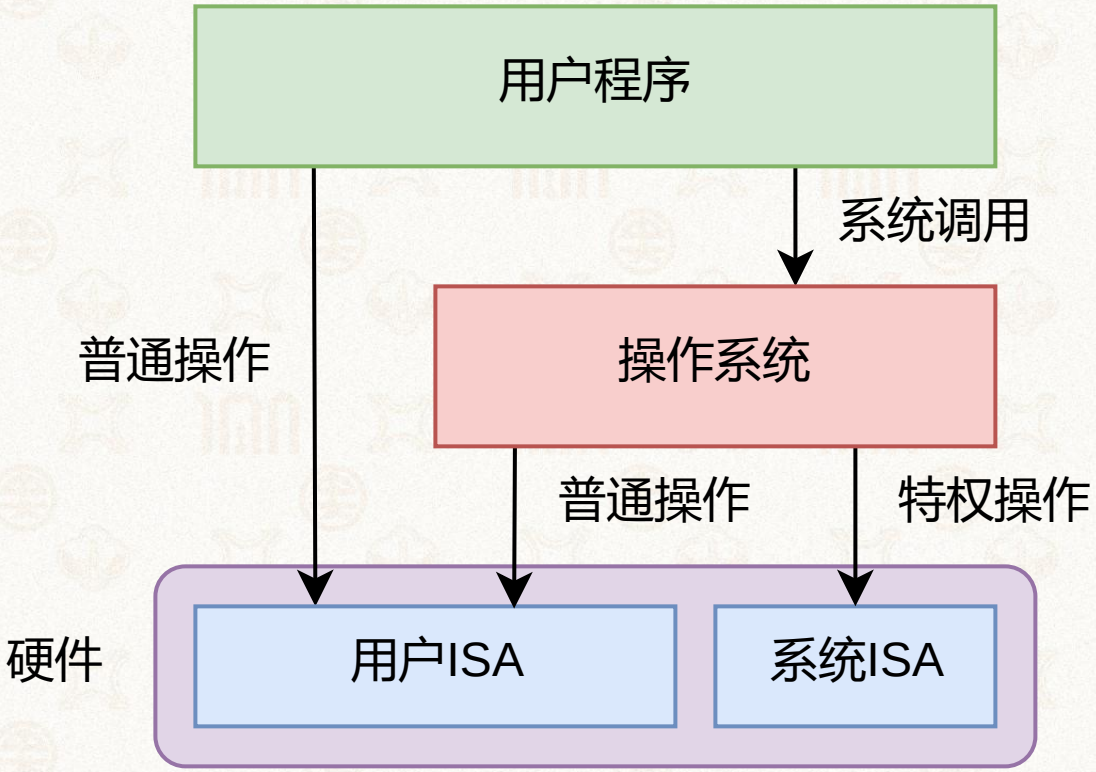
- 特权级模型

- 设备与中断
  - 基本概念
  - 中断如何产生
  - 设备的内存映射
  - 异常处理

应用程序的硬件执行环境

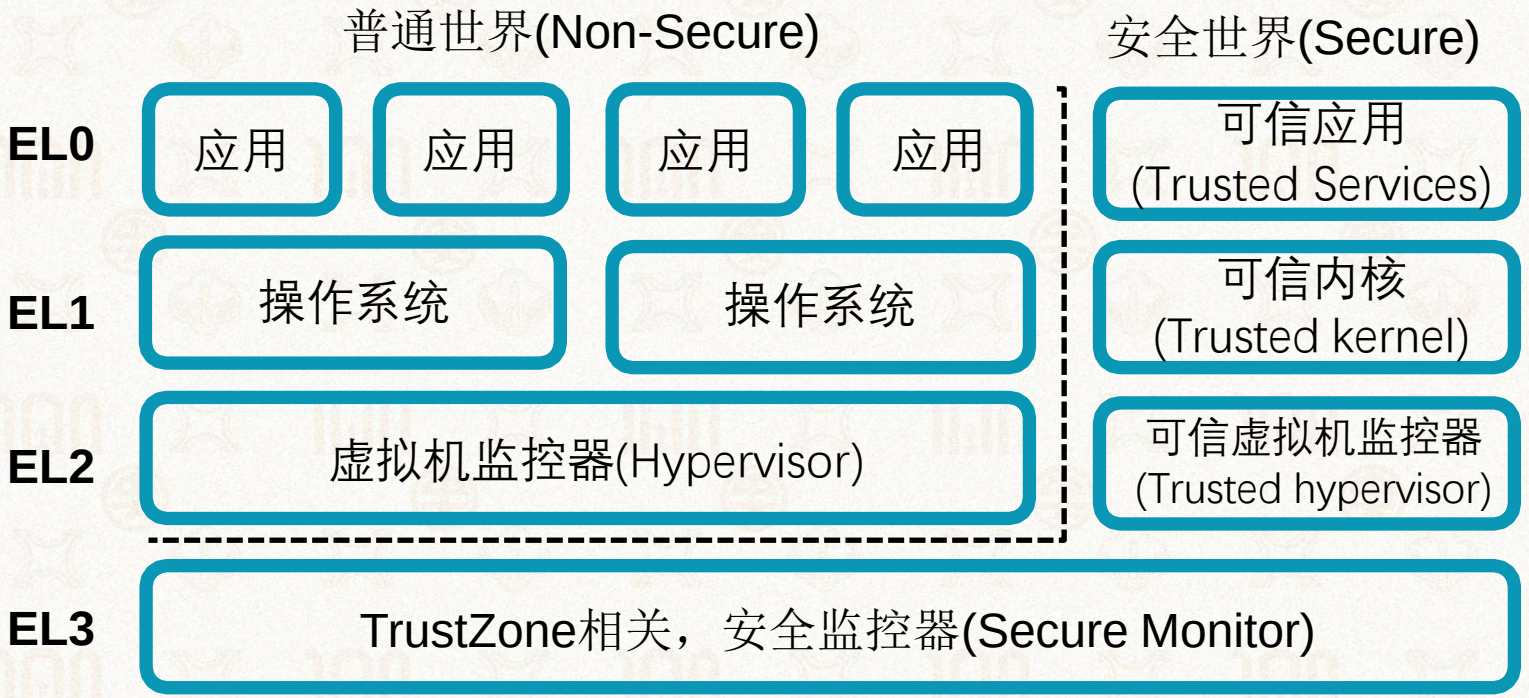
- 操作系统也是程序
  - 大量的CPU指令
  - 拥有一些特权指令

- 指令集架构(Instruction Set Architecture, ISA)
  - 与硬件绑定
  - 用户ISA示例
    - `mov x0, sp`
    - `add x0, x0, #1`
  - 系统ISA示例
    - `msr vbar_el1, x0`



特权级模型

- 4个异常级别(Exception Level, EL)，2个和安全相关的
- EL0异常级别最低，EL3异常级别最高
- 如果不开启安全模式和虚拟化，就不用管EL2和EL3



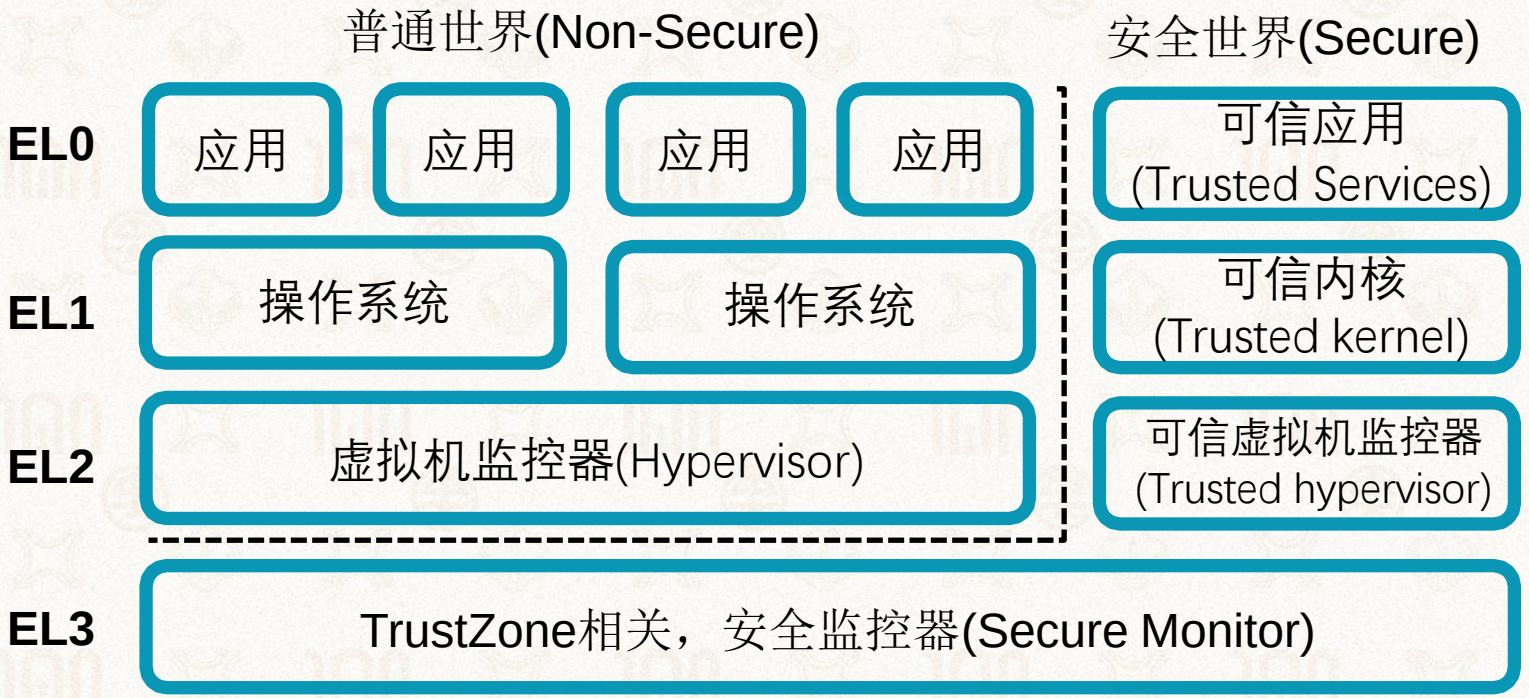
ARM中常用的寄存器

- 31个64位通用寄存器
  - X0-X30
- 1个PC寄存器
- 4个栈寄存器（切换时保存SP）
  - SP_EL0, SP_EL1, SP_EL2, SP_EL3
- 3个异常链接寄存器（保存异常的返回地址）
  - ELR_EL1, ELR_EL2, ELR_EL3
- 3个程序状态寄存器（切换时保存PSTATE）
  - SPSR_EL1, SPSR_EL2, SPSR_EL3

寄存器		EL0	EL1	描述
通用寄存器	$X0 \sim X30$	√	√	
特殊寄存器	PC	√	√	程序计数器
	SP_EL0	√	√	用户栈寄存器
	SP_EL1		√	内核栈寄存器
	$PSTATE$	√	√	状态寄存器
系统寄存器	ELR_EL1		√	异常链接寄存器
	$SPSR\_EL1$		√	保存的状态寄存器
	$VBAR\_EL1$		√	异常向量表基地址
	ESR_EL1		√	异常症状寄存器

特权级模型

- 处理器何时会从EL0进入到EL1?
  - 执行系统调用，具体指令为特权调用：svc (supervisor call)
  - 应用执行的指令触发了异常
  - CPU收到了外设发来的中断信息



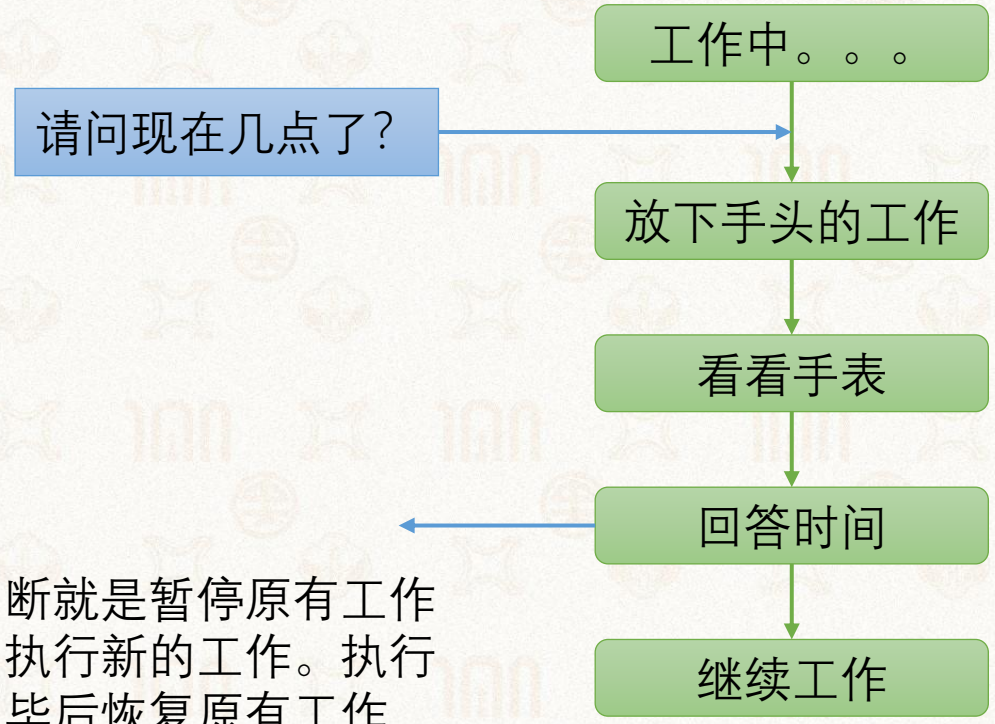
大纲

- 特权级模型

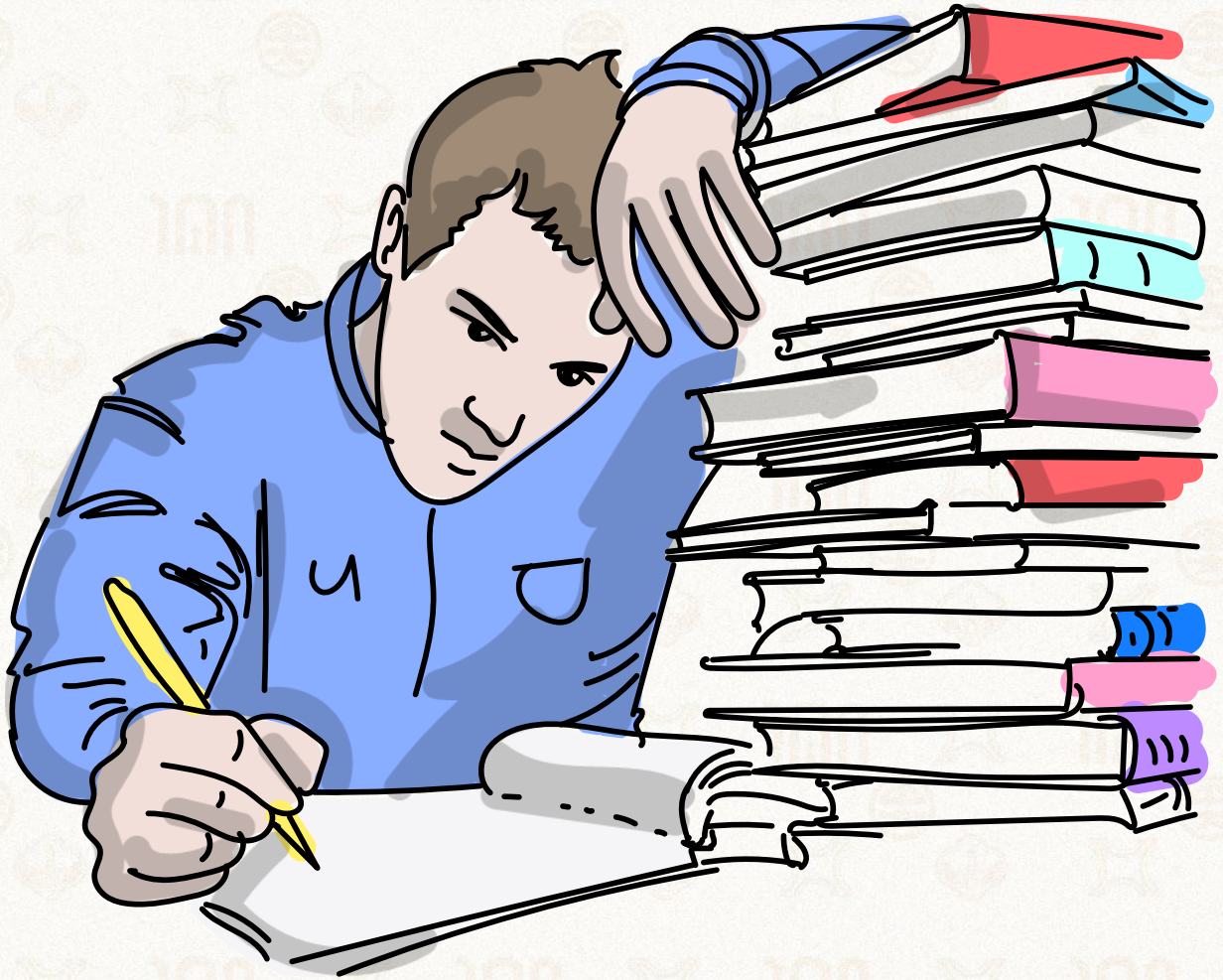
- 设备与中断
  - 基本概念
  - 中断如何产生
  - 设备的内存映射
  - 异常处理

同学你好，请问现在几点了？

- 正在奋笔疾书时，问人家时间
- 如果态度好，会回答你。问答的过程是怎样的？

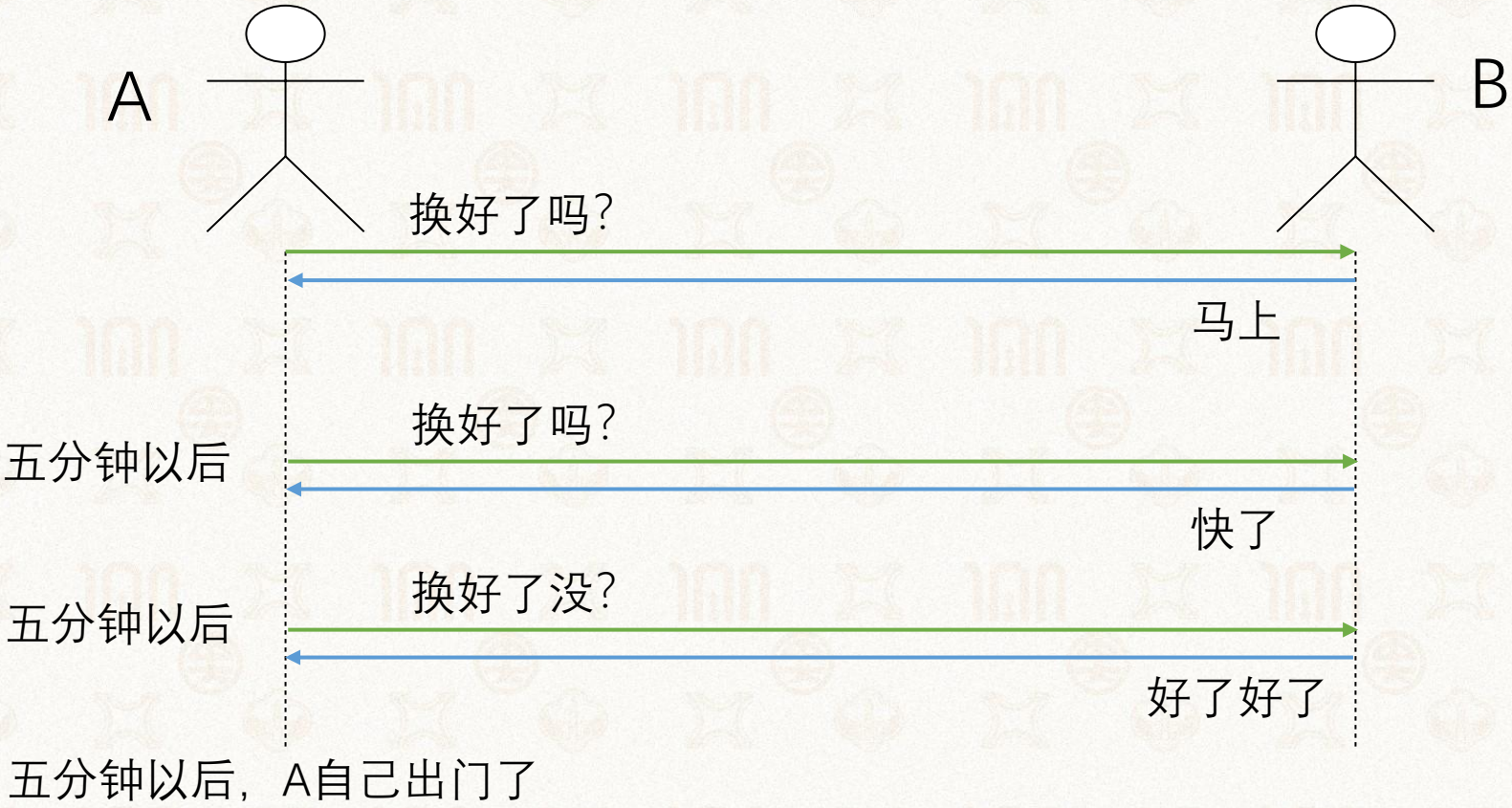


中断就是暂停原有工作，执行新的工作。执行完毕后恢复原有工作



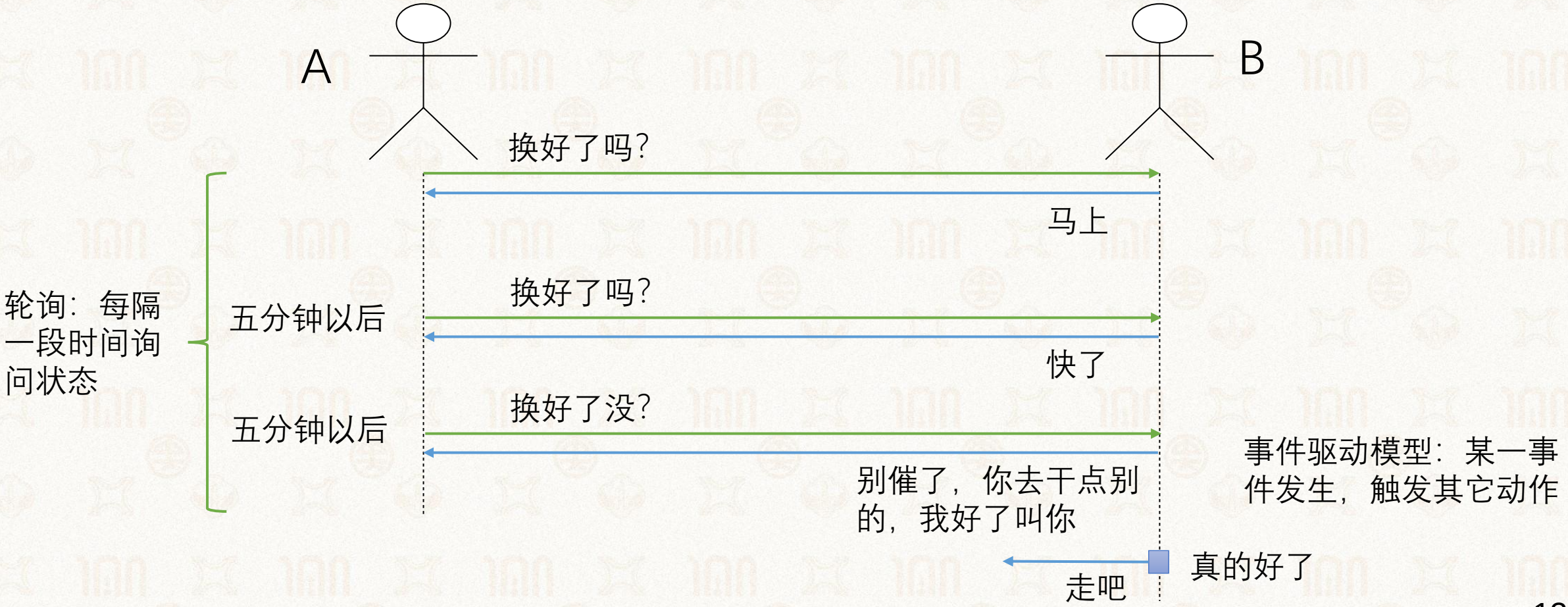
两人相约出门逛街

- 同学A与同学B相约出门逛街
- A已换好衣服，B还在挑选出门的衣服

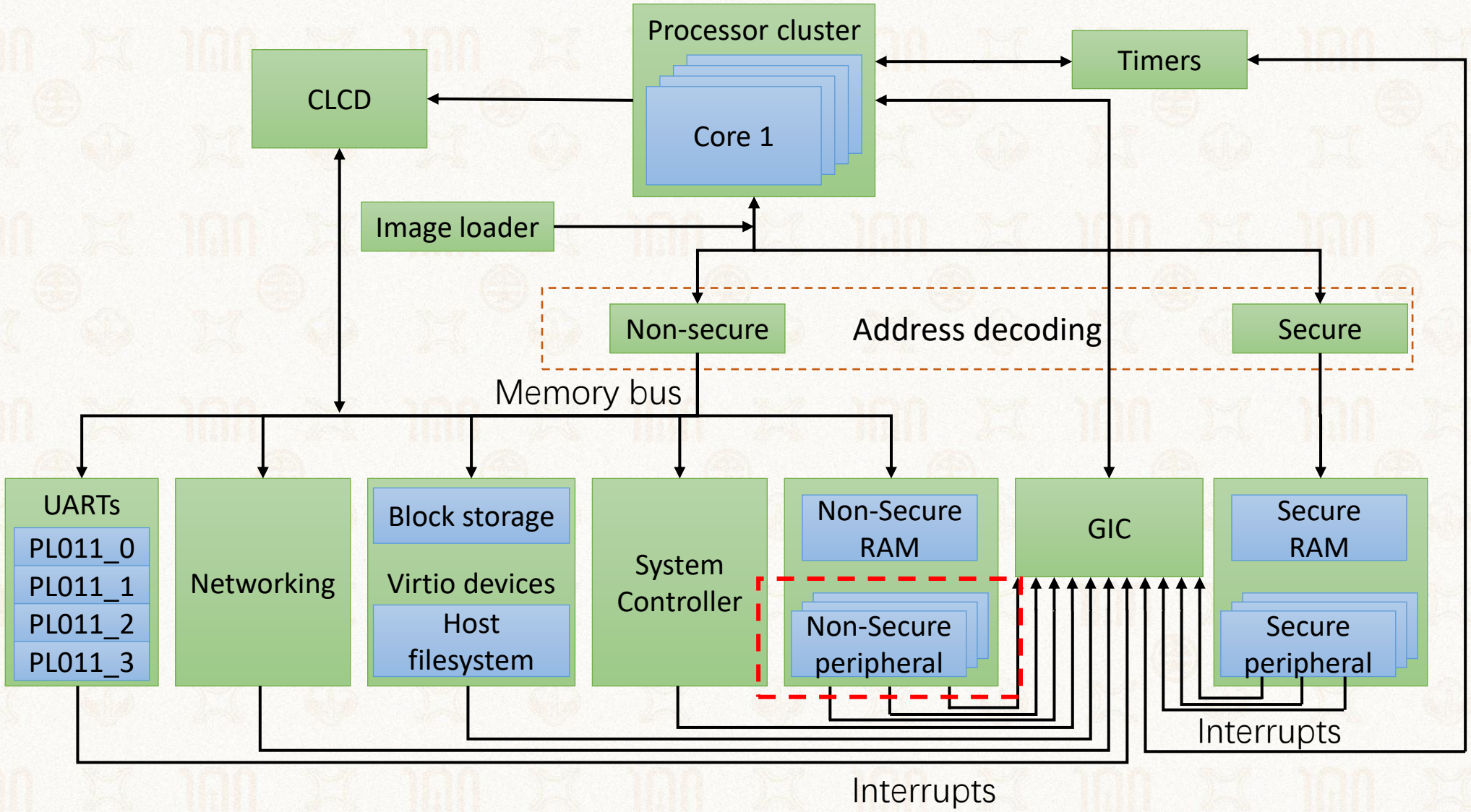


两人相约出门逛街

- 同学A与同学B相约出门逛街
- A已换好衣服，B还在挑选出门的衣服



从键盘说起

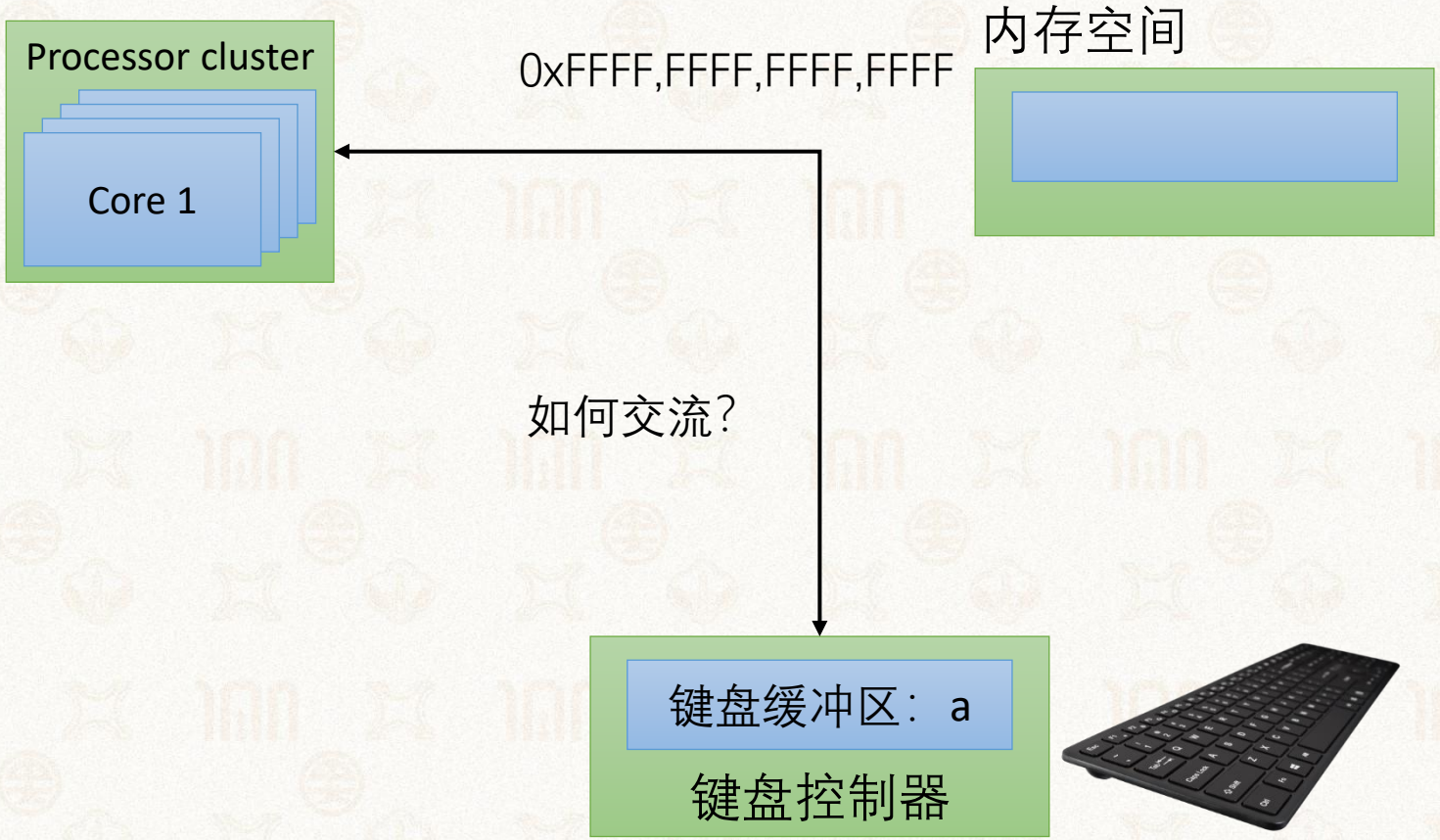

Processor cluster
Core 1
CLCD
Timers
Image loader
Non-secure
Address decoding
Secure
Memory bus
UARTs
PL011_0
PL011_1
PL011_2
PL011_3
Networking
Block storage
Virtio devices
Host filesystem
System Controller
Non-Secure RAM
Non-Secure peripheral
GIC
Secure RAM
Secure peripheral
Interrupts
Interrupts

从键盘说起

- CPU如何得知键盘输入的字符?
- 键盘的字符直接写入内存中?

从键盘说起

- CPU如何得知键盘输入的字符?
- 键盘等外设具有控制器和缓冲区，将输入存入缓冲区
- CPU获取该输入的可能方法:
  - 轮询：CPU不断去读该缓冲区中的值
  - 中断：当控制器接收到输入后，打断CPU正常执行，OS进行处理



哪种形式更好一些?
为什么?

通用概念

- 中断（Interrupt）
  - 外部硬件设备所产生的信号
  - 异步：产生原因和当前执行指令无关，如程序被磁盘读打断

- 异常（Exception）
  - 软件的程序执行而产生的事件
  - 包括系统调用（System Call）
    - 用户程序请求操作系统提供服务
  - 同步：产生和当前执行或试图执行的指令相关

不同体系结构术语的对应关系

通用概念	产生原因	AArch64		x86-64
中断	硬件异步	异常	异步异常（重置/中断）	中断（可屏蔽/不可屏蔽）
异常	软件同步		同步异常（终止/异常指令）	异常（Fault/Trap/Abort）

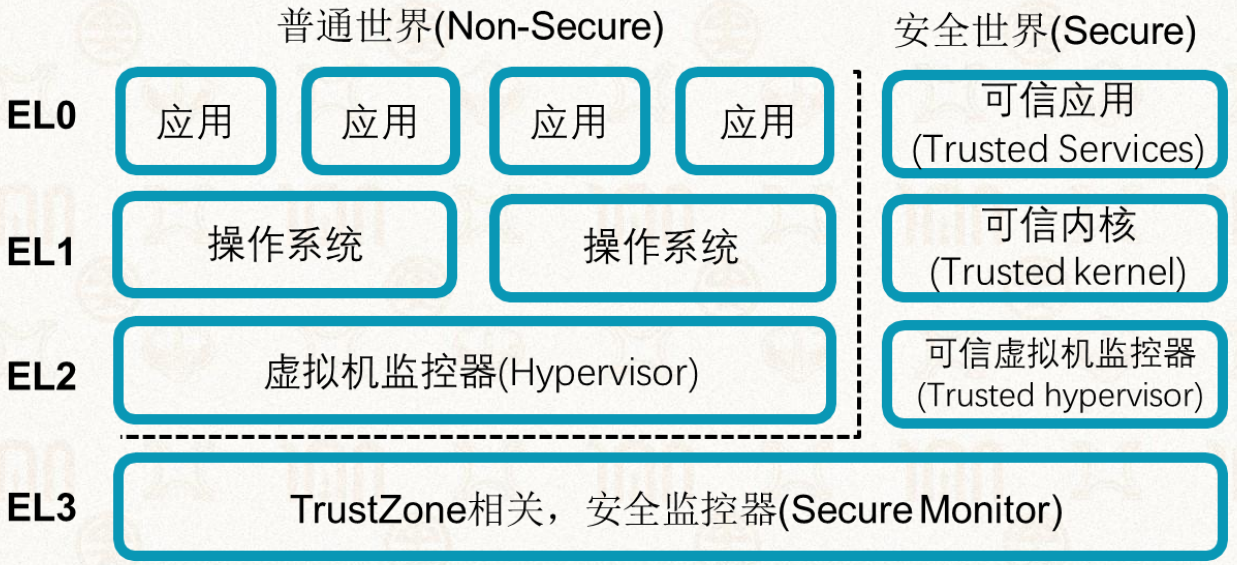
- 之后提到的“中断”、“异常”均为通用概念意义

AArch64的中断（异步异常）

- 重置（Reset）
  - 最高级别的异常，用以执行代码初始化CPU核心
  - 由系统首次上电或控制软件、看门狗等触发
- 中断（Interrupt）
  - CPU外部的信号触发，打断当前执行
  - 如计时器中断、键盘中断等

AArch64的（同步）异常

- 中止（Abort）
  - 失败的指令获取或数据访问
  - 如访问不可读的内存地址等
- 异常产生指令（Exception generating instructions）
  - SVC (Supervisor Call)：用户程序 -> 操作系统
  - HVC (Hypervisor Call)：客户系统 -> 虚拟机管理器
  - SMC (Secure Mode Call)：Normal World -> Secure World



x86-64术语

- 中断（设备产生、异步）
  - 可屏蔽：设备产生的信号，通过中断控制器与处理器相连，可被暂时屏蔽（如，键盘、网络事件）
  - 不可屏蔽：一些关键硬件的崩溃（如，内存校验错误）

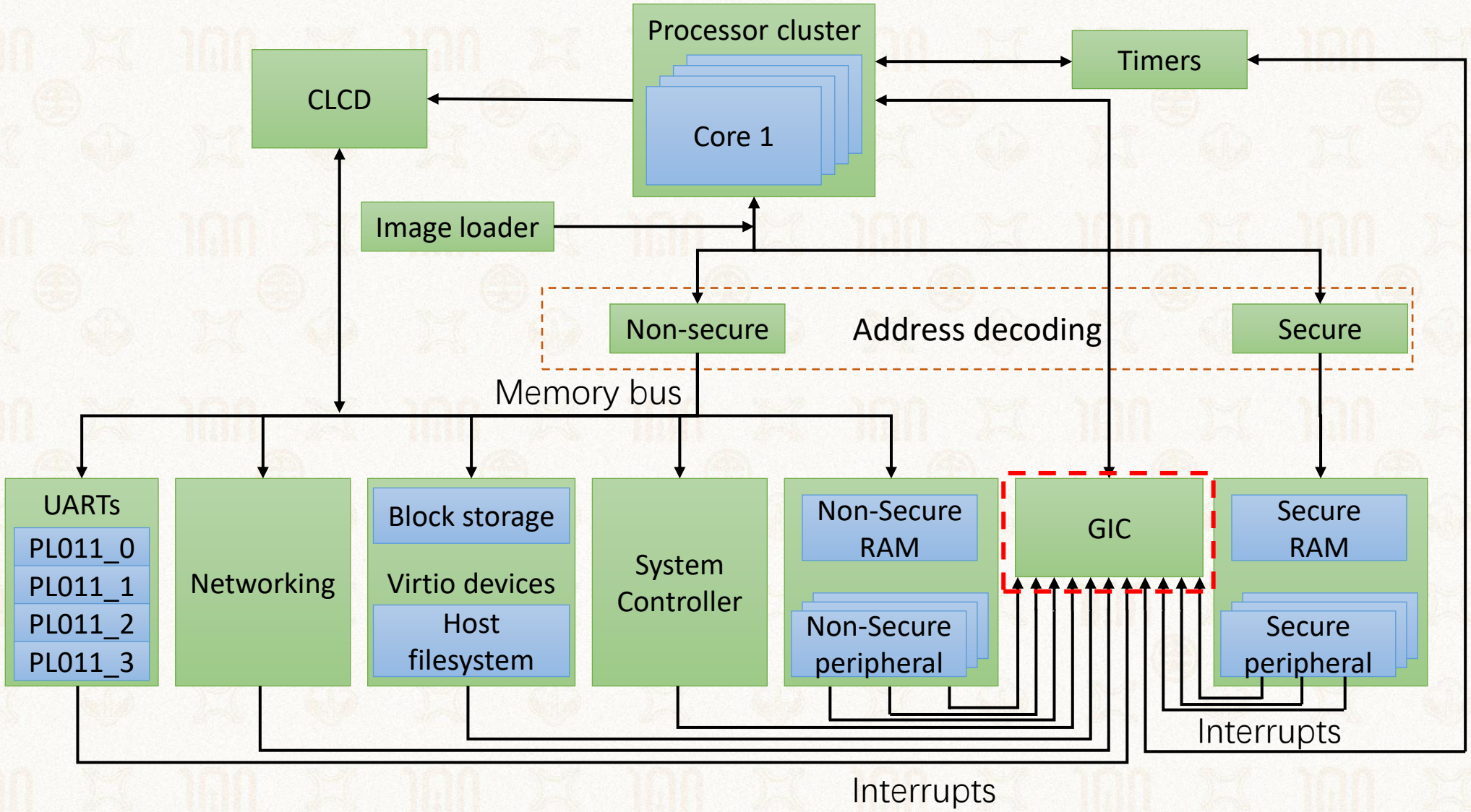
- 异常（软件产生、同步）
  - 错误（Fault）：如缺页异常（可恢复）、段错误（不可恢复）等
  - 陷阱（Trap）：无需恢复，如断点（int 3）、系统调用（int 80）
  - 中止（Abort）：严重的错误，不可恢复（机器检查）

大纲

- 特权级模型

- 设备与中断
  - 基本概念
  - 中断如何产生
  - 设备的内存映射
  - 异常处理

中断控制器


Processor cluster
Core 1
CLCD
Timers
Image loader
Non-secure
Address decoding
Secure
Memory bus
UARTs
PL011_0
PL011_1
PL011_2
PL011_3
Networking
Block storage
Virtio devices
Host filesystem
System Controller
Non-Secure RAM
Non-Secure peripheral
GIC
Secure RAM
Secure peripheral
Interrupts
Interrupts

AArch64中断的分类

- IRQ（Interrupt Request）
  - 普通中断，优先级低，处理慢
- FIQ（Fast Interrupt Request）
  - 一次只能有一个FIQ
  - 快速中断，优先级高，处理快
  - 常为可信任的中断源预留

连接CPU的不同针脚

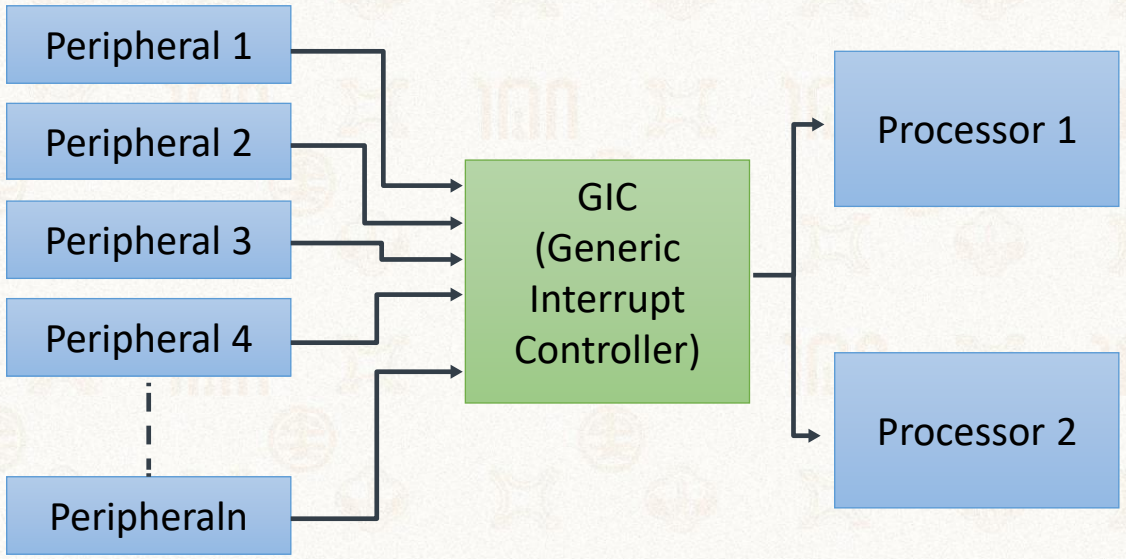
可在中断控制器中配置

- SError（System Error）
  - 原因难以定位、较难处理的异常，多由异步中止（Abort）导致
  - 如从缓存行（Cacheline）写回至内存时发生的异常

中断控制器需要考虑的问题

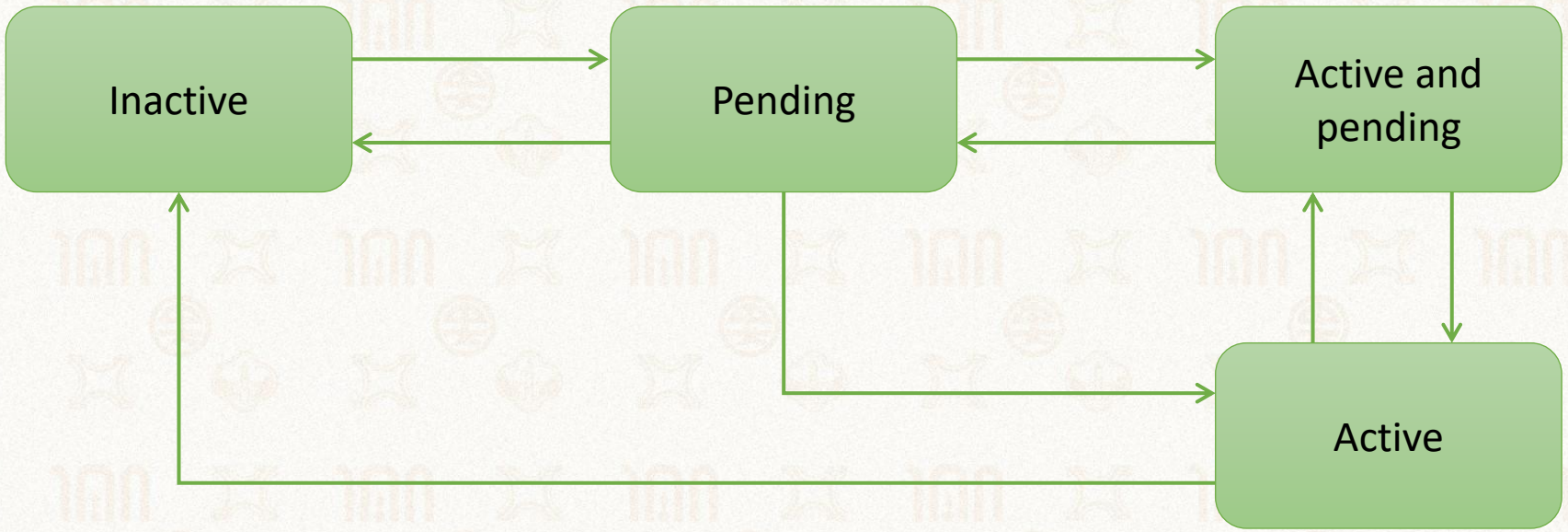
- 如何指定不同中断的优先级
  - 低优先级中断处理中，出现了高优先级的中断
  - 嵌套中断
- 中断交给谁处理
- 如何与软件协同

- 主要功能
  - 分发：管理所有中断、决定优先级、路由
  - CPU接口：给每个CPU核有对应的接口



GIC中断状态

- 确定有限状态:
  - Inactive: 中断源没有被触发
  - Pending: 中断源已经被触发，但CPU还没来得处理
  - Active:中断源已经被触发，CPU正在处理
  - Active and pending: 同一种中断源，其中一个实例正在被处理，而下一个实例已来，处于等待中



触发中断

➢ 电平触发中断

由外设到中断控制器的中断信号电位

由中断控制器到CPU的中断信号电位

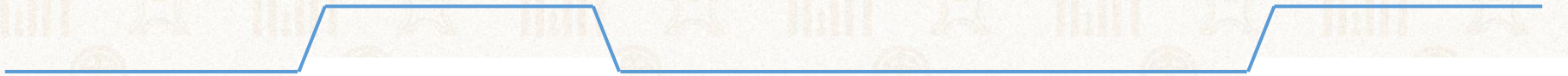
GIC中断状态模式

触发中断

边沿触发中断

由外设到中断控制器的中断信号电位

由中断控制器到CPU的中断信号电位



GIC中断状态模式

大纲

- 特权级模型

- 设备与中断
  - 基本概念
  - 中断如何产生
  - 设备的内存映射
  - 异常处理

统一内存映射

- 虚拟地址空间
- 前2^{48}字节的地址空间给普通应用
- 操作系统、外设的地址均为0xFFFF开头

内存映射的输入输出(Memory-Mapped I/O)

```c
unsigned int early_uart_recv(void)
{
    while (1) {
        if (early_uart_lsr() & 0x01)
            break;
    }
    return early_get32(AUX_MU_IO_REG) & 0xFF;
}

void early_uart_send(unsigned int c)
{
    while (1) {
        if (early_uart_lsr() & 0x20)
            break;
    }
    early_put32(AUX_MU_IO_REG, c);
}
```

```
BEGIN_FUNC(early_get32)
    ldr w0,[x0]
    ret
END_FUNC(early_get32)
```

- **MMIO: 复用ldr和str指令**
  - 映射到物理内存的特殊地址段

```
BEGIN_FUNC(early_put32)
    str w1,[x0]
    ret
END_FUNC(early_put32)
```

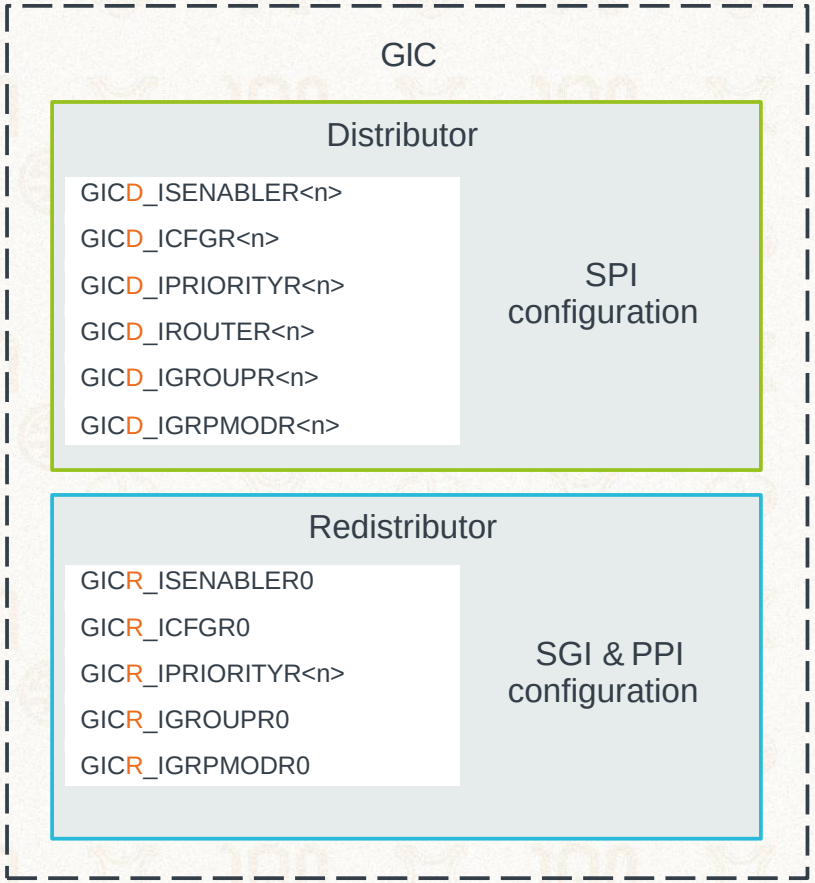
GIC的路由配置 – 以启用timer为例

- 使用MMIO，设置GIC中寄存器，启用timer

```
BEGIN_FUNC(put32)
    str w1,[x0]
    ret
END_FUNC(put32)
```

```
#define GICD_ISENABLER  (KBASE+0xE82B1100)

void plat_interrupt_init(void)
{
    u32 cpuid = smp_get_cpu_id();

    if (cpuid == 0)
        gicv2_dist_init();

    gicv2_cpu_init();

    /* 启用timer */
    put32(GICD_ISENABLER, 0x08000000);
    timer_init();

}
```

GIC的配置信息：



GIC中断信息获取

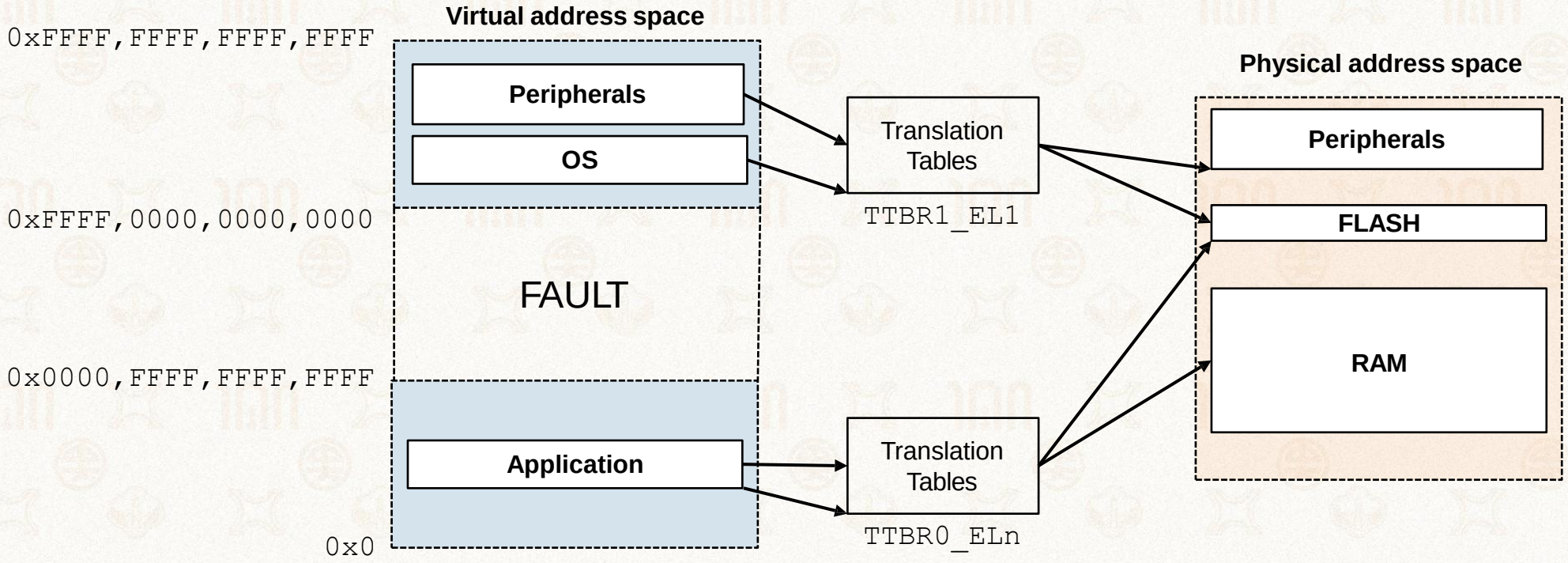
- 使用MMIO，从GIC中的寄存 器里获得中断信息

```
#define CORE_IRQ_BASE           (KBASE + 0x40000060)
#define CORE0_IRQ_SOURCE        (CORE_IRQ_BASE + 0x0)
#define CORE1_IRQ_SOURCE        (CORE_IRQ_BASE + 0x4)
#define CORE2_IRQ_SOURCE        (CORE_IRQ_BASE + 0x8)
#define CORE3_IRQ_SOURCE        (CORE_IRQ_BASE + 0xc)

u64 core_irq_source[PLAT_CPU_NUM] = {
    CORE0_IRQ_SOURCE,
    CORE1_IRQ_SOURCE,
    CORE2_IRQ_SOURCE,
    CORE3_IRQ_SOURCE
};
```

```
void plat_handle_irq(void)
{
    u32 cpuid = 0;
    unsigned int irq_src, irq;

    cpuid = smp_get_cpu_id();
    irq_src = get32(core_irq_source[cpuid]);

    irq = 1 << ctzl(irq_src);
    switch (irq) {
    case INT_SRC_TIMER3:
        handle_timer_irq();
        break;
    default:
        kinfo("Unsupported IRQ %d\n", irq);
    }
    return;
}
```

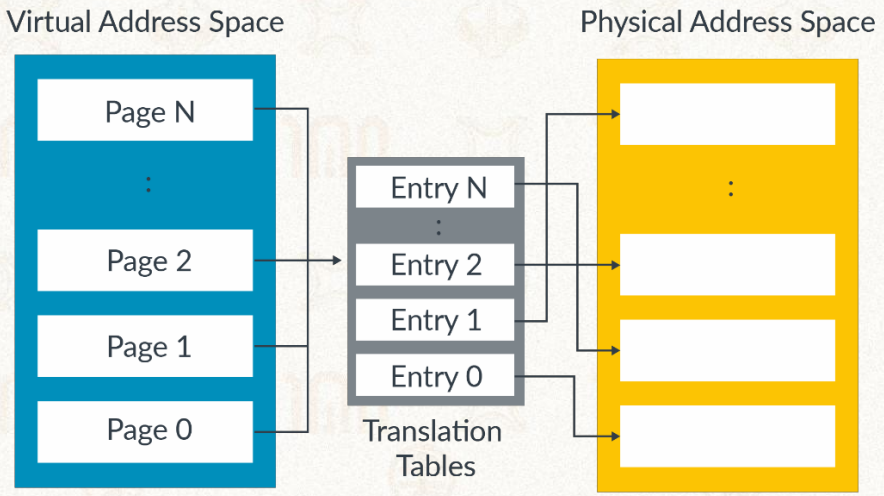
虚拟地址与物理地址的映射

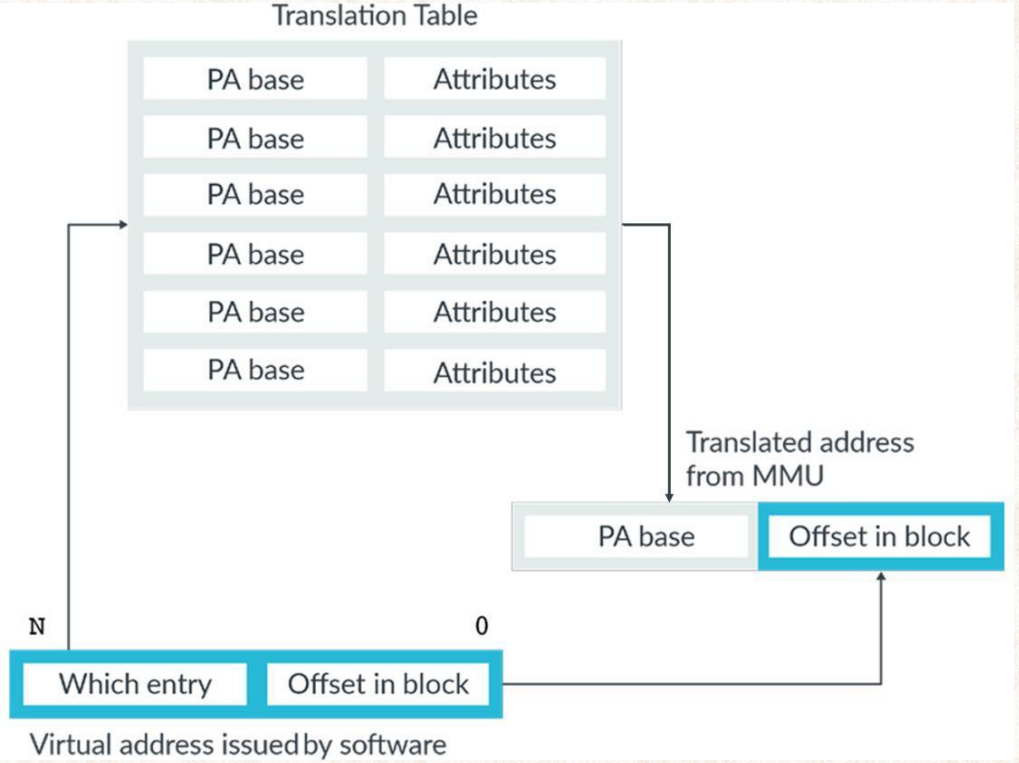
- 实现虚拟地址到物理地址的翻译



翻译过程

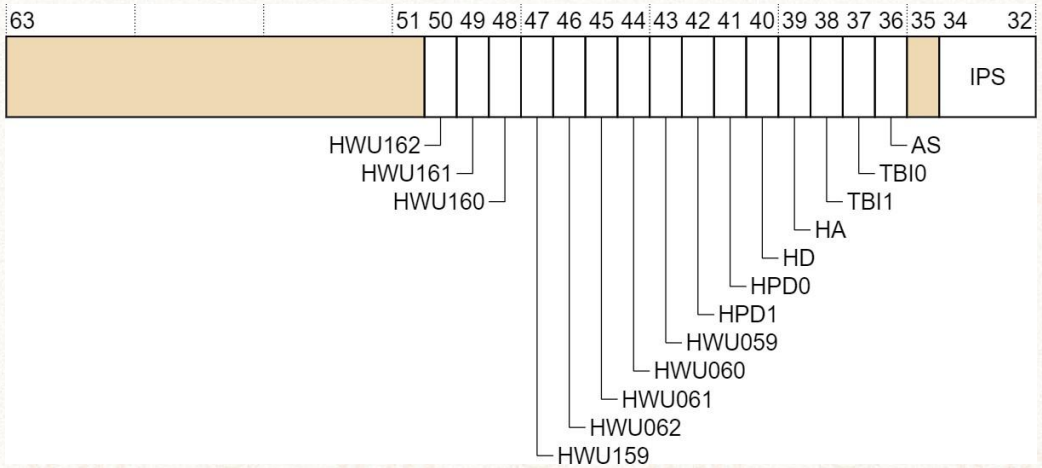
- 以页表的形式：
  - 找到虚拟和物理地址基地址的映射关系

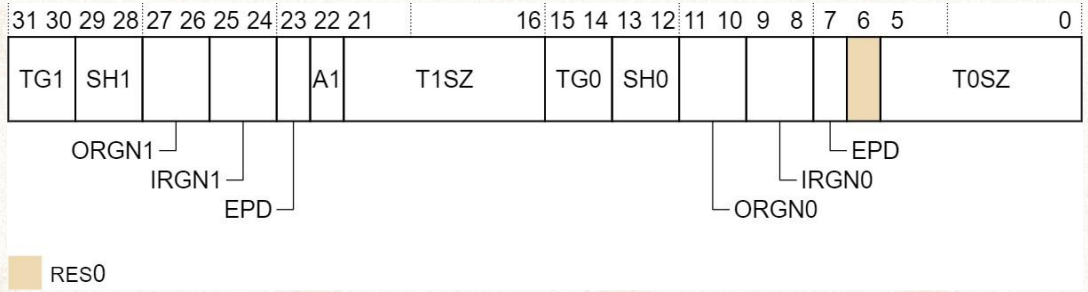




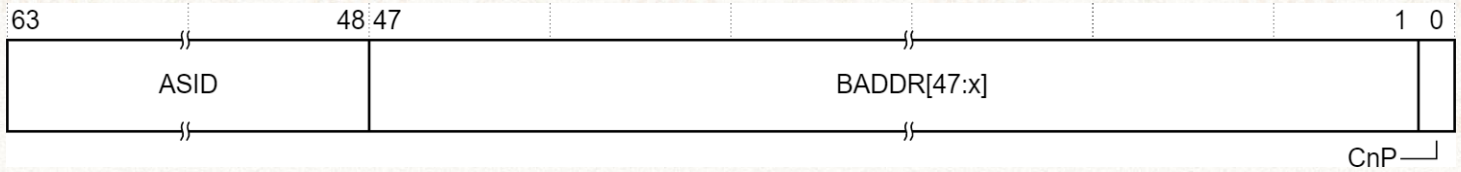
内存系统相关寄存器

- Translation Control Register (TCR)

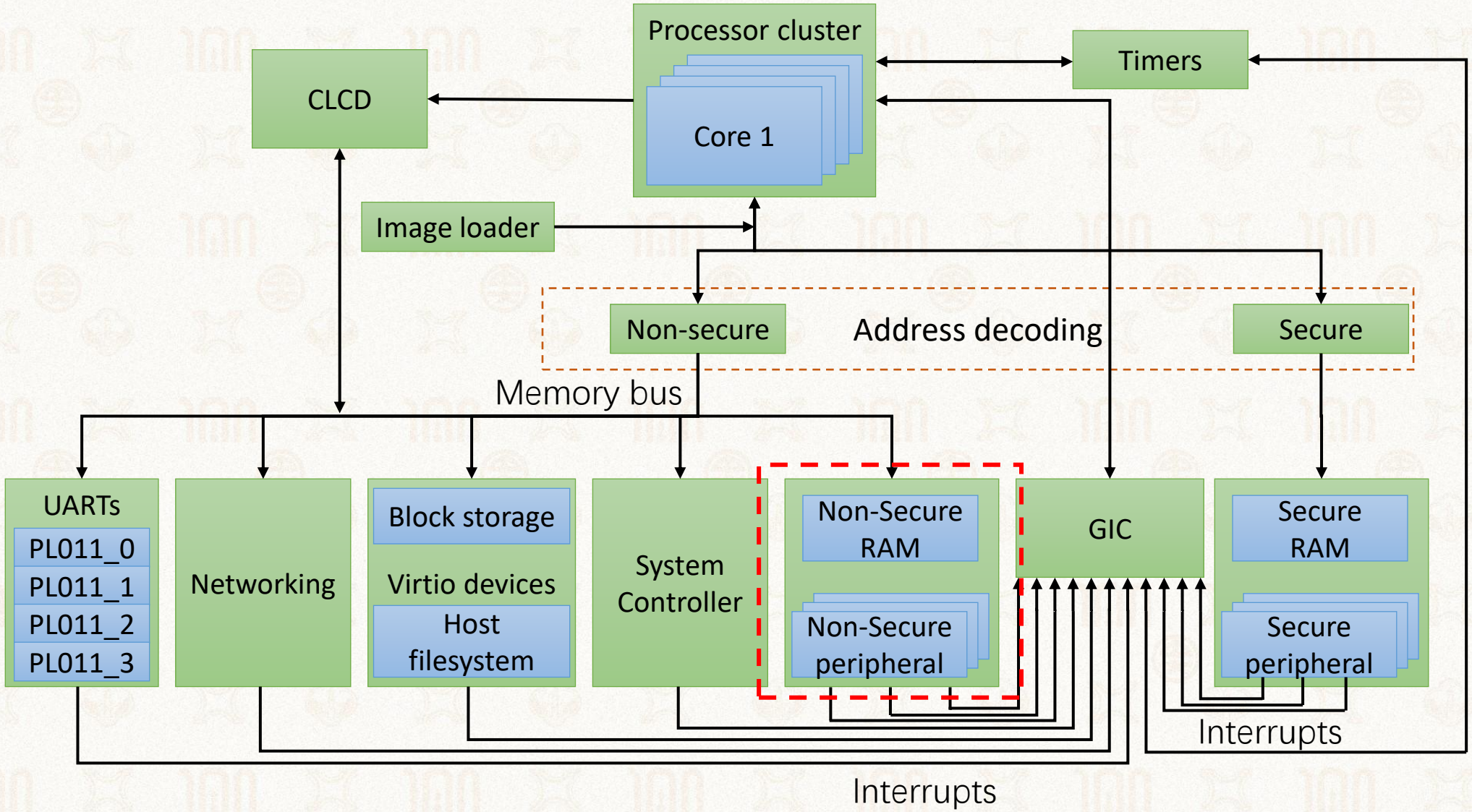




- Translation Table Base Register (TTBR)



ARMv8基础平台



GIC: Generic Interrupt Controller UART: Universal Asynchronous Receiver/Transmitters CLCD: Color LCD

ARMv8基础平台的内存映射

Start address	End address	Foundation v1 peripheral	Foundation v2 and v9 peripherals	Size	Security (v2 and v9 only)
0x00_0000_0000	0x00_03FF_FFFF	RAM	Trusted Boot ROM, secureflash	64MB	S
0x00_0400_0000	0x00_0403_FFFF	RAM	Trusted SRAM	256KB	S
0x00_0600_0000	0x00_07FF_FFFF	RAM	Trusted DRAM	32MB	S
0x00_0800_0000	0x00_0BFF_FFFF	-	NOR flash, flash0	64MB	S/NS
0x00_0C00_0000	0x00_0FFF_FFFF	-	NOR flash, flash1	64MB	S/NS
0x00_1800_0000	0x00_19FF_FFFF	-	VRAM	32MB [a]	-
0x00_1A00_0000	0x00_1AFF_FFFF	Ethernet, SMSC 91C111	Ethernet, SMSC 91C111	16MB	S/NS
0x00_1C01_0000	0x00_1C01_FFFF	System Registers	System Registers	64KB	S/NS
0x00_1C02_0000	0x00_1C02_FFFF	-	System Controller, SP810	64KB	S/NS
0x00_1C04_0000	0x00_1C07_FFFF	-	Warning + RAZ/WI	-	-
0x00_1C09_0000	0x00_1C09_FFFF	UART0, PL011	UART0, PL011	64KB	S/NS
0x00_1C0A_0000	0x00_1C0A_FFFF	UART1, PL011	UART1, PL011	64KB	S/NS
0x00_1C0B_0000	0x00_1C0B_FFFF	UART2, PL011	UART2, PL011	64KB	S/NS
0x00_1C0C_0000	0x00_1C0C_FFFF	UART3, PL011	UART3, PL011	64KB	S/NS
0x00_1C0D_0000	0x00_1C0D_FFFF	-	Warning + RAZ/WI	-	-
0x00_1C0F_0000	0x00_1C0F_FFFF	-	Watchdog, SP805	64KB	S/NS
0x00_1C10_0000	0x00_1C10_FFFF	-	Base Platform Power Controller	64KB	S/NS
0x00_1C11_0000	0x00_1C11_FFFF	-	Dual-Timer 0, SP804	64KB	S/NS
0x00_1C12_0000	0x00_1C12_FFFF	-	Dual-Timer 1, SP804	64KB	S/NS
0x00_1C13_0000	0x00_1C13_FFFF	Virtio block device	Virtio block device	64KB	S/NS
0x00_1C14_0000	0x00_1C16_FFFF	-	Virtio Plan 9 for v9, Warning + RAZ/W for v2.1	-	-
0x00_1C17_0000	0x00_1C17_FFFF	-	Realtime Clock, PL031	64KB	S/NS

ARMv8基础平台的内存映射

Start address	End address	Foundation v1 peripheral	Foundation v2 and v9 peripherals	Size	Security (v2 and v9 only)
0x00_1C0B_0000	0x00_1C0B_FFFF	UART2, PL011	UART2, PL011	64KB	S/NS
0x00_1C0C_0000	0x00_1C0C_FFFF	UART3, PL011	UART3, PL011	64KB	S/NS
0x00_1C0D_0000	0x00_1C0D_FFFF	-	Warning + RAZ/WI	-	-
0x00_1C0F_0000	0x00_1C0F_FFFF	-	Watchdog, SP805	64KB	S/NS
0x00_1C10_0000	0x00_1C10_FFFF	-	Base Platform Power Controller	64KB	S/NS
0x00_1C11_0000	0x00_1C11_FFFF	-	Dual-Timer 0, SP804	64KB	S/NS
0x00_1C12_0000	0x00_1C12_FFFF	-	Dual-Timer 1, SP804	64KB	S/NS
0x00_1C13_0000	0x00_1C13_FFFF	Virtio block device	Virtio block device	64KB	S/NS
0x00_1C14_0000	0x00_1C16_FFFF	-	Virtio Plan 9 for v9, Warning + RAZ/W for v2.1	-	-
0x00_1C17_0000	0x00_1C17_FFFF	-	Realtime Clock, PL031	64KB	S/NS
0x00_1C1A_0000	0x00_1FFF_FFFF	-	Warning + RAZ/W	-	-
0x00_1F00_0000	0x00_1F00_0FFF	-	Non-trusted ROM	4KB	S/NS
0x00_2A43_0000	0x00_2A43_FFFF	-	REFCLK CNTControl, Generic Timer	64KB	S
0x00_2A44_0000	0x00_2A44_FFFF	-	EL2 Generic Watchdog Control	64KB	S/NS
0x00_2A45_0000	0x00_2A45_FFFF	-	EL2 Generic Watchdog Refresh	64KB	S/NS
0x00_2A49_0000	0x00_2A49_FFFF	-	Trusted Watchdog, SP805	64KB	S
0x00_2A4A_0000	0x00_2A4A_FFFF	-	Warning + RAZ/W	-	-
0x00_2A80_0000	0x00_2A80_FFFF	-	REFCLK CNTRead, Generic Timer	64KB	S/NS
0x00_2A81_0000	0x00_2A81_FFFF	-	AP_REFCLK CNTCTL, Generic Timer	64KB	S/NS
0x00_2A82_0000	0x00_2A82_FFFF	-	AP_REFCLK CNTBase0, Generic Timer	64KB	S

ARMv8基础平台的内存映射

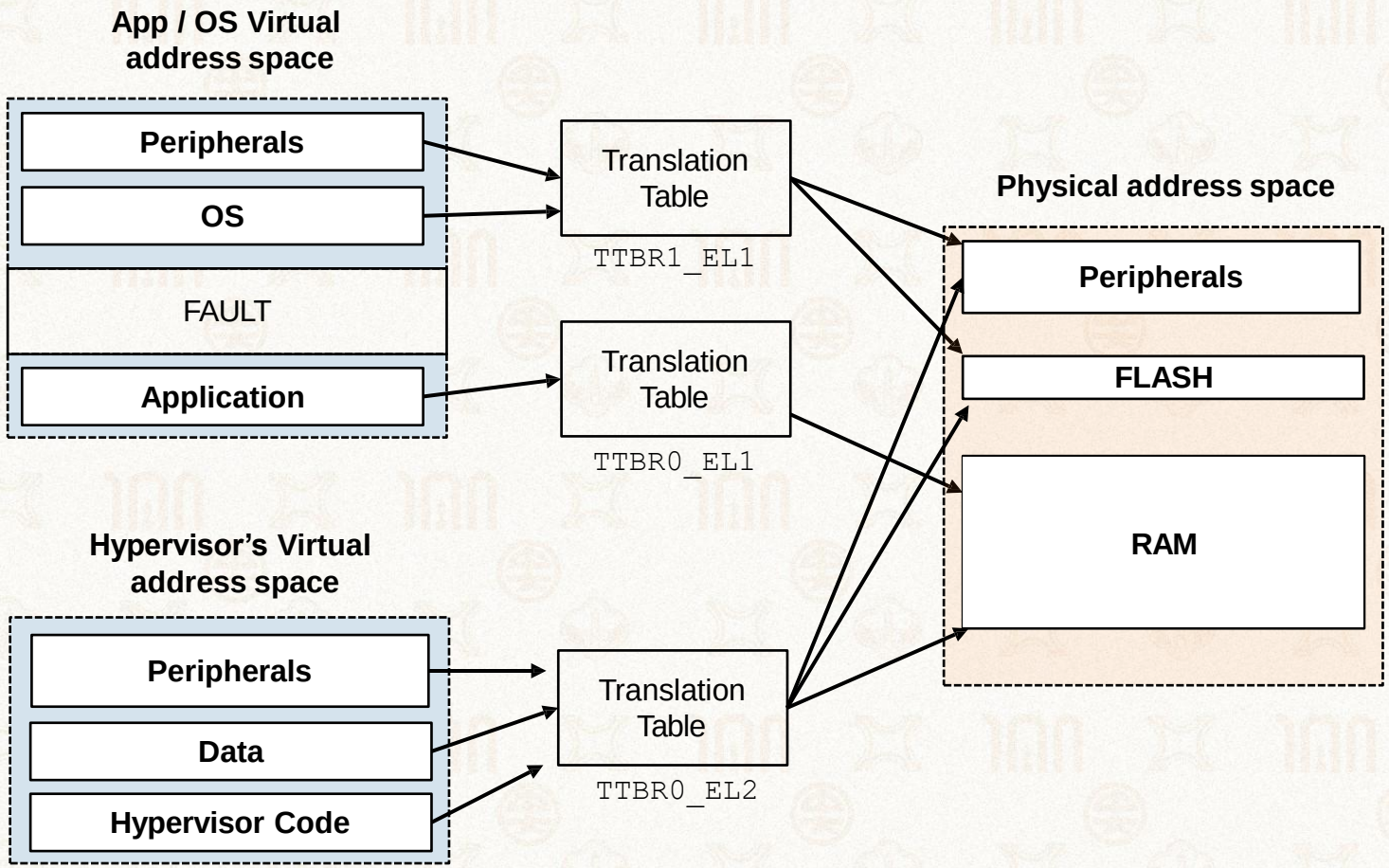
Start address	End address	Foundation v1 peripheral	Foundation v2 and v9 peripherals	Size	Security (v2 and v9 only)
0x00_2A83_0000	0x00_2A83_FFFF	-	AP_REFCLK CNTBase1, Generic Timer	64KB	S/NS
0x00_2C00_0000	0x00_2C00_1FFF	-	GIC Physical CPU interface, GICC[b]	8KB	S/NS
0x00_2C00_1000	0x00_2C00_1FFF	GIC Distributor	GIC Distributor[c]	4KB	-
0x00_2C00_2000	0x00_2C00_2FFF	GIC Processor Interface	GIC Processor Interface[c]	4KB	-
0x00_2C00_4000	0x00_2C00_4FFF	GIC Processor Hyp Interface	GIC Processor Hyp Interface[c]	4KB	-
0x00_2C00_5000	0x00_2C00_5FFF	GIC Hyp Interface	GIC Hyp Interface[c]	4KB	-
0x00_2C00_6000	0x00_2C00_7FFF	GIC Virtual CPU Interface	GIC Virtual CPU Interface[c]	8KB	-
0x00_2C01_0000	0x00_2C01_0FFF	-	GIC Virtual Interface Control, GICH	4KB	S/NS
0x00_2C02_F000	0x00_2C03_0FFF	-	GIC Virtual CPU Interface, GICV	8KB	S/NS
0x00_2C09_0000	0x00_2C09_FFFF	-	Warning + RAZ/W	-	-
0x00_2E00_0000	0x00_2E00_FFFF	-	Non-trusted SRAM	64KB	S/NS
0x00_2F00_0000	0x00_2F00_FFFF	-	GICv3 Distributor GICD[b]	64KB	S/NS
0x00_2F10_0000	0x00_2F1F_FFFF	-	GICv3 Distributor GICR	1MB	S/NS
0x00_7FE6_0000	0x00_7FE6_0FFF	-	Trusted Random Number Generator	4KB	S
0x00_7FE7_0000	0x00_7FE7_0FFF	-	Trusted Non-volatile counters	4KB	S
0x00_7FE8_0000	0x00_7FE8_0FFF	-	Trusted Root-Key Storage	4KB	S
0x00_8000_0000	0x00_FFFF_FFFF	DRAM (0GB - 2GB)	DRAM (0GB - 2GB)	2GB	S/NS
0x08_8000_0000	0x09_FFFF_FFFF	DRAM (2GB - 8GB)	DRAM (2GB - 8GB)	6GB	S/NS

在一个成熟的系统中，外设非常多，很难管理

http://caxapa.ru/thumbs/845437/DUI0677N_armv8_a_fp_ug.pdf

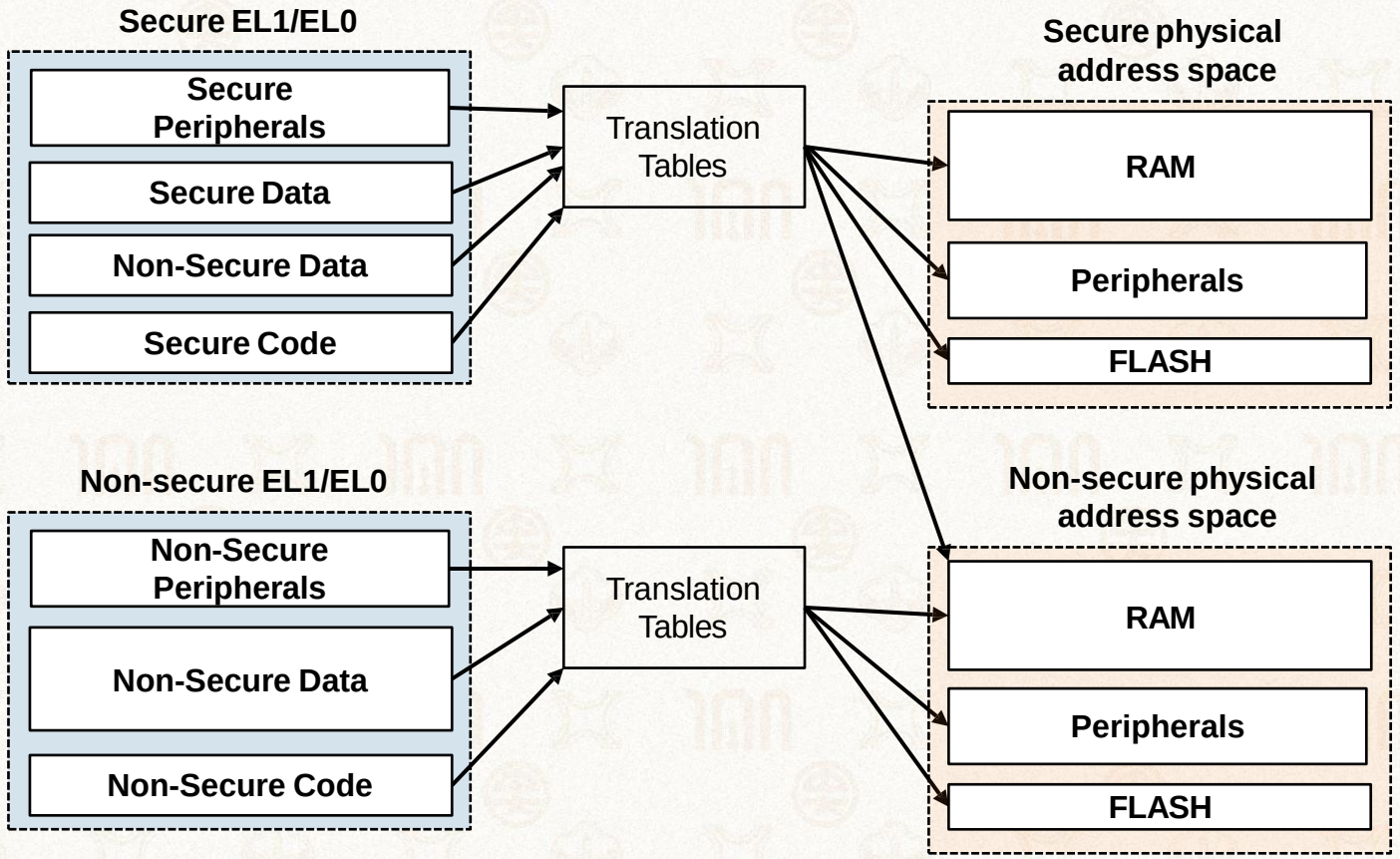
多个虚拟内存空间

- 不同特权级可以定义自己的内存转换表:
- EL0/1:
  - TTBR0_EL1
  - TTBR1_EL1
  - TCR_EL1
- EL2: 虚拟机监控器
  - TTBR0_EL2
  - TCR_EL2
- EL3: 安全监控
  - TTBR0_EL3
  - TCR_EL3

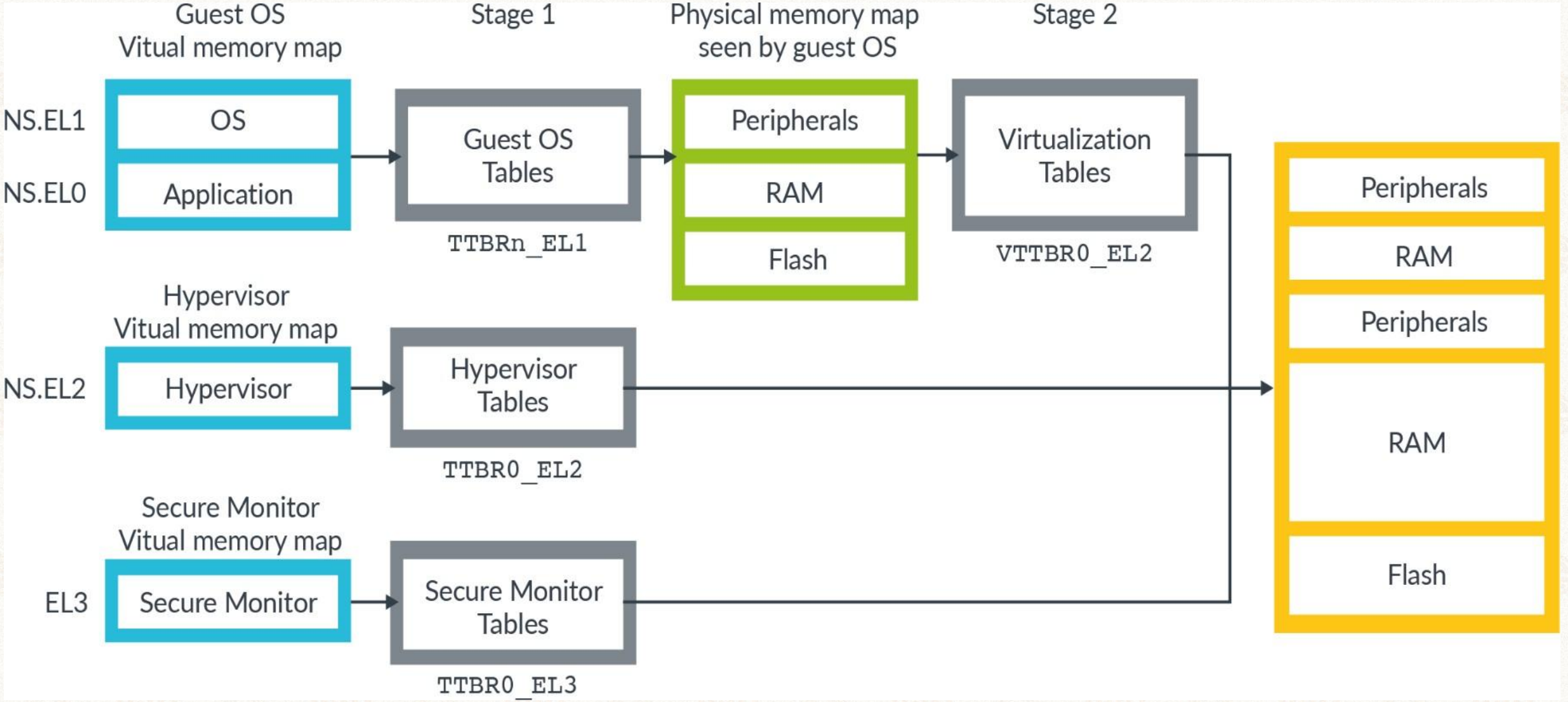


不同安全模式下的地址翻译

- 物理空间也要分安全、普通世界
- 普通世界只能访问普通世界的地址
- 安全世界可以访问所有地址



不同特权级的地址翻译


Guest OS
Vitual memory map
Stage 1
Physical memory map
seen by guest OS
Stage 2
NS.EL1
OS
NS.EL0
Application
Guest OS
Tables
TTBRn_EL1
Peripherals
RAM
Flash
Virtualization
Tables
VTTBR0_EL2
Hypervisor
Vitual memory map
NS.EL2
Hypervisor
Hypervisor
Tables
TTBR0_EL2
Secure Monitor
Vitual memory map
EL3
Secure Monitor
Secure Monitor
Tables
TTBR0_EL3
Peripherals
RAM
Peripherals
RAM
Flash

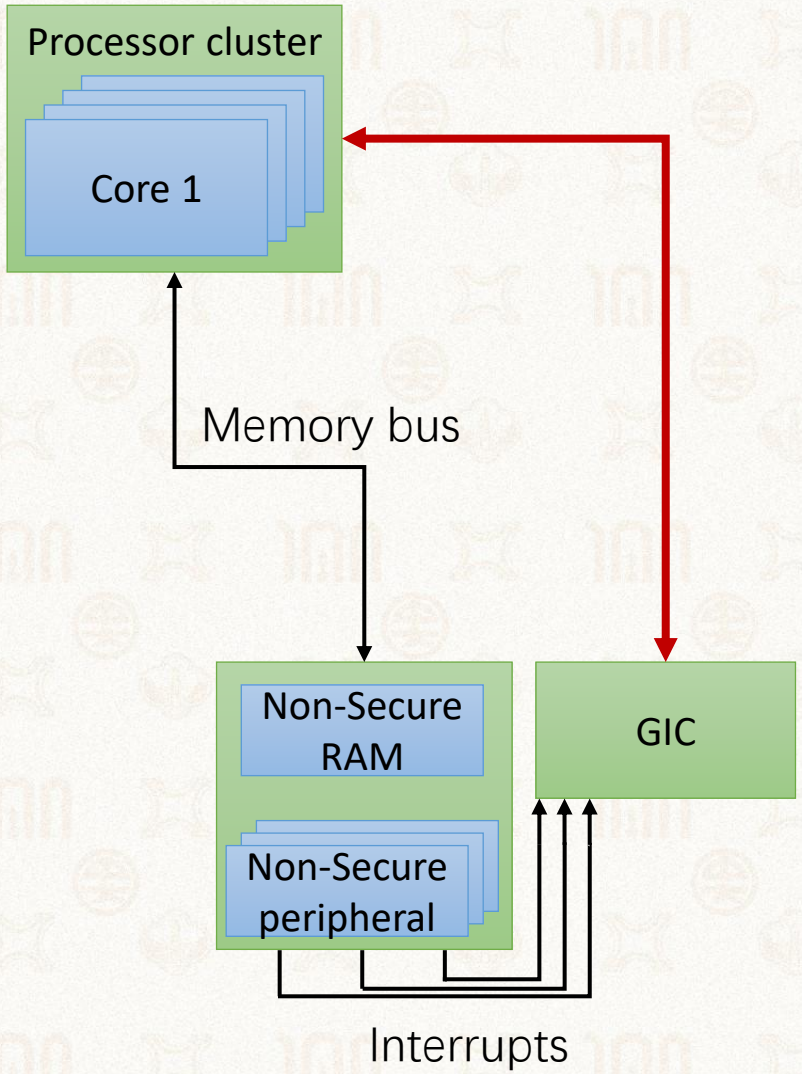
大纲

- 特权级模型

- 设备与中断
  - 基本概念
  - 中断如何产生
  - 设备的内存映射
  - 异常处理

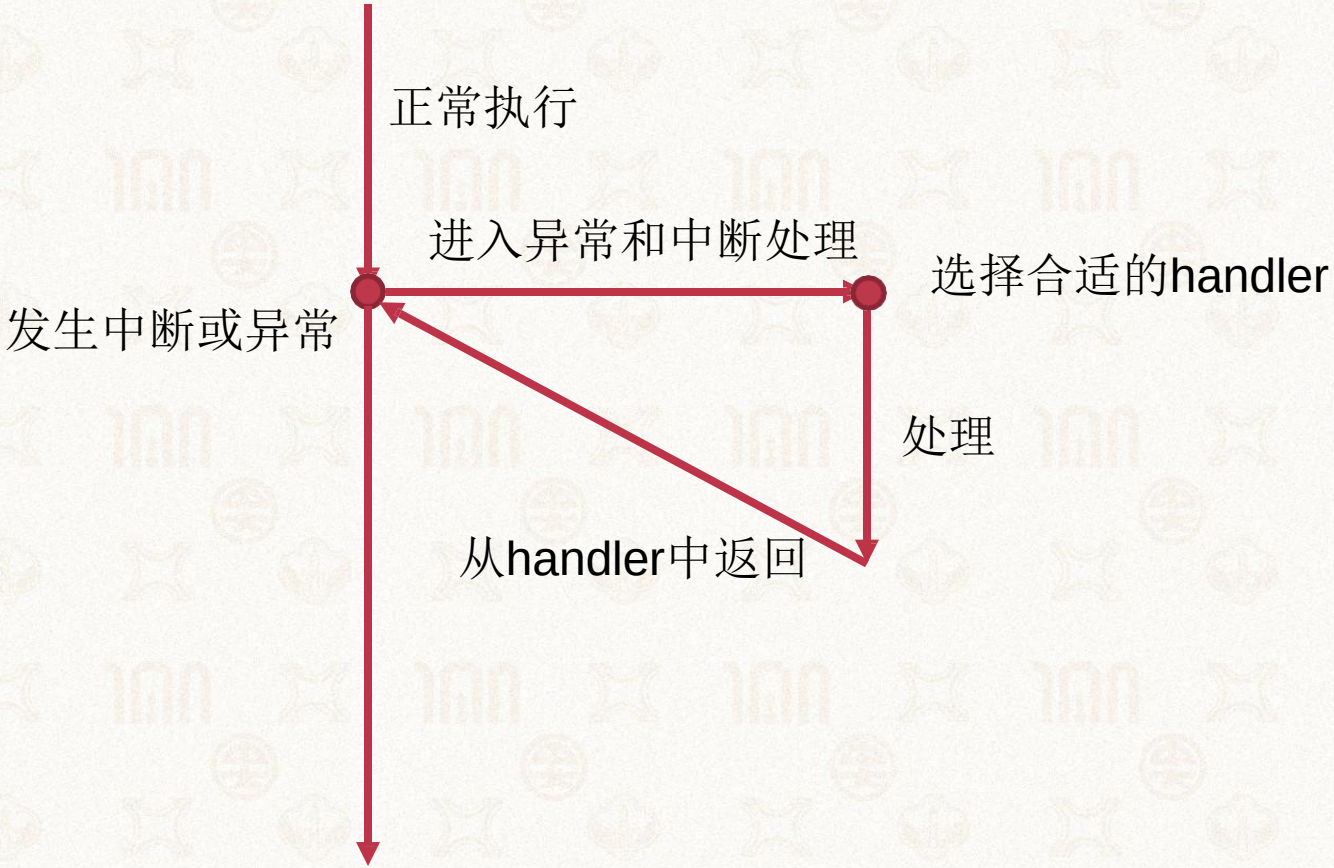
中断和异常的处理

- 中断控制器将IRQ, FIQ信号发送给CPU
- 由CPU处理后续



基本处理流程

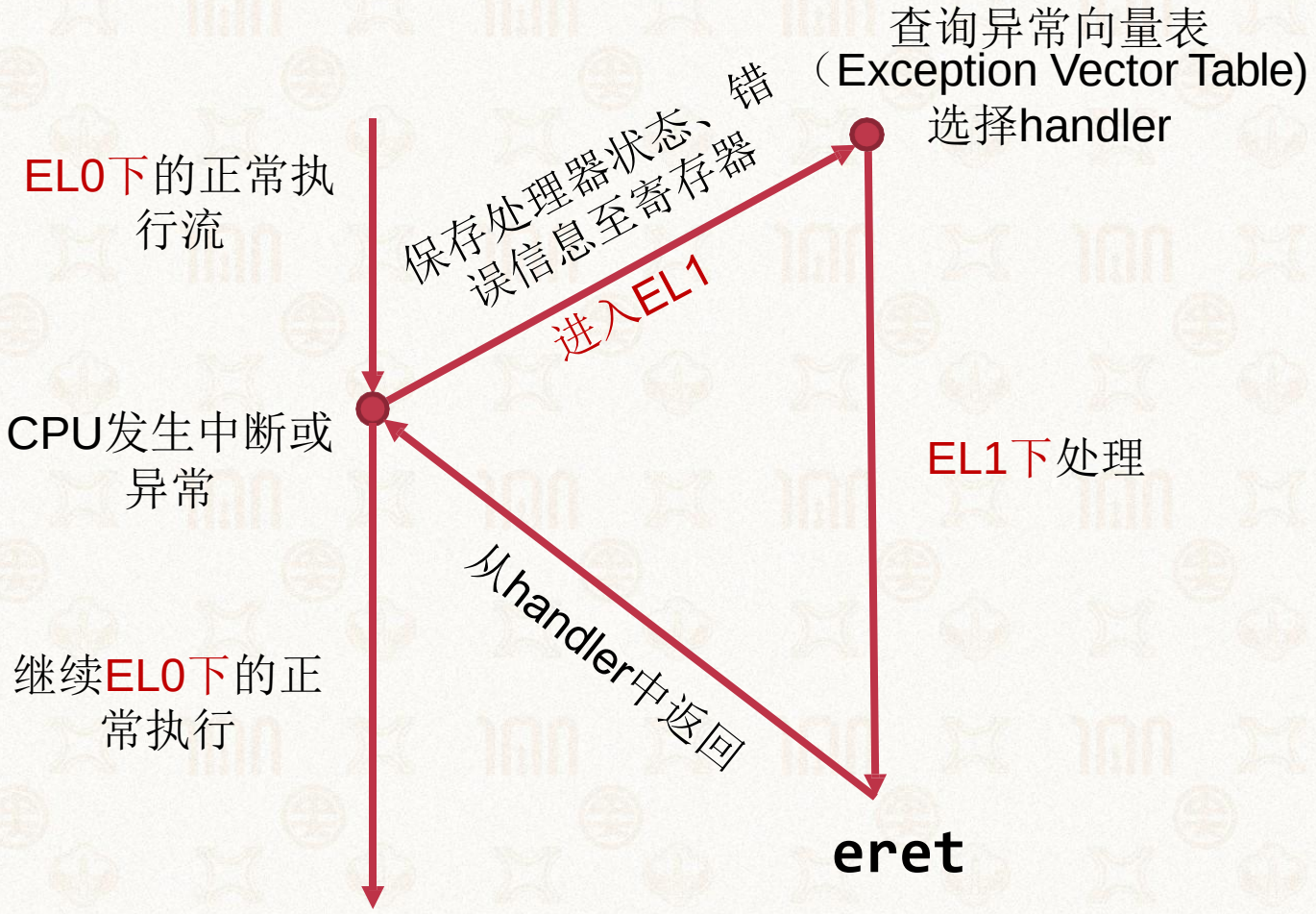
- 中断与异常的处理使用同一套机制，差异仅在选择handler中提现



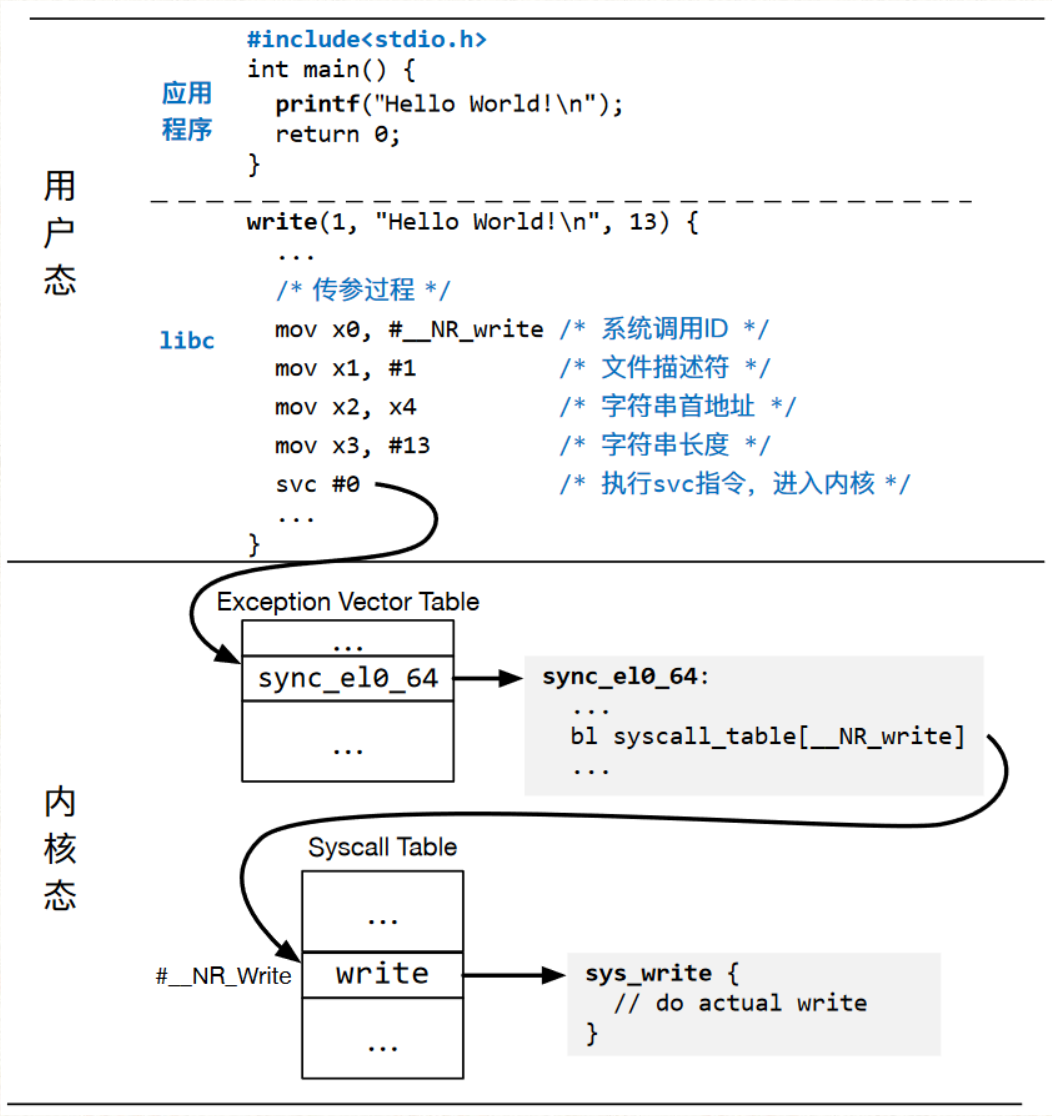
中断和异常处理必做事项

- 进入中断或异常时
  - 需保存处理器状态，方便之后恢复执行
  - 需准备好在高特权级下进行执行的环境
  - 需选择合适的异常处理器代码进行执行
  - 需保证用户态和内核态之间的隔离
- 处理时
  - 需获得关于异常的信息，如系统调用参数、错误原因等
- 返回时
  - 需恢复处理器状态，返回低特权级，继续正常执行流

AArch64的中断和异常处理



SVC 系统调用：使用异常向量表



发生 – 信息保存

- 异常或中断发生后，硬件会将错误码和部分上下文信息存储在寄存器中
  - 处理器状态（PSTATE）-> Saved Program Status Register（SPSR_EL1）
  - 当前指令地址（PC）-> Exception Link Register（ELR_EL1）
  - 异常发生原因 ->
    - Serror与异常：Exception Syndrome Register（ESR_EL1）
    - 中断：GIC中的寄存器（使用MMIO读取）

- 安全性问题
  - 上述寄存器均不可在用户态（EL0）中访问

发生 – 进入EL1

- 硬件会适当修改处理器状态（PSTATE），进入EL1执行
- 问题：栈内存的安全性
  - 进入EL1级别后，栈指针（SP）会自动换用SP_EL1
  - 从而实现用户栈->内核栈
  - 如需在EL1下使用SP_EL0作为栈指针，可配置SPSel寄存器

寻找handler的代码

- 使用异常向量表（Exception Vector Table）
  - 每个异常级别存在独立的异常向量表
  - 表项为异常向量（Exception Vector），是处理异常或跳转到异常handler的小段汇编代码
  - 地址位于VBAR_EL1寄存器中
  - 选择表项取决于
    - 异常类型（同步、IRQ、FIQ、Serror）
    - 异常发生的特权级
    - 异常发生时的处理器状态（使用的栈指针/运行状态）

异常向量表

VBAR_EL1

地址	异常类型	异常发生时处理器状态
+ 0x000	Synchronous	EL1 使用SP_EL0作为SP
+ 0x080	IRQ	
+ 0x100	FIQ	
+ 0x180	SError	
+ 0x200	Synchronous	EL1 使用SP_EL1作为SP
+ 0x280	IRQ	
+ 0x300	FIQ	
+ 0x380	SError	
+ 0x400	Synchronous	EL0 运行于AArch64状态
+ 0x480	IRQ	
+ 0x500	FIQ	
+ 0x580	SError	
+ 0x600	Synchronous	EL0 运行于AArch32状态
+ 0x680	IRQ	
+ 0x700	FIQ	
+ 0x780	SError	

异常向量表

VBAR_EL1

地址	异常类型	异常发生时处理器状态
+ 0x000	Synchronous	EL1 使用SP_EL0作为SP
+ 0x080	IRQ	
+ 0x100	FIQ	
+ 0x180	SError	
+ 0x200	Synchronous	EL1 使用SP_EL1作为SP
+ 0x280	IRQ	
+ 0x300	FIQ	
+ 0x380	SError	
+ 0x400	Synchronous	EL0 运行于AArch64状态
+ 0x480	IRQ	
+ 0x500	FIQ	
+ 0x580	SError	
+ 0x600	Synchronous	EL0 运行于AArch32状态
+ 0x680	IRQ	
+ 0x700	FIQ	
+ 0x780	SError	

AArch64状态下，EL0中发生同步异常：
内存访问权限错误

- 保存上下文、错误信息
- 切换至EL1
- PC <- VBAR_EL1 + 0x400
- 开始执行

异常向量表

```
_start:
mrs x8, mpidr_el1
and x8, x8, #0xFF
cbz x8, primary

/* Code for secondary core */ 7
...
/* Init exception vector */
bl set_exception_vector
...

primary:
/* Code for primary core */
/* init UART, Virtual Memory mapping in C */
...
/* Init exception vector */
adr x0, el1_vector
msr vbar_el1, x0
...
```

```
/* Exception Table */
el1_vector:
/* entry for other type of exception */
...
/* entry for synchronous exception from EL0 */
.align 7 // 128 bytes for each entry
b sync_el0_64
/* entry for interrupt from EL0 */
.align 7 // 128 bytes for each entry
b irq_el0_64
/* entry for other type of exception */
...
```

返回（Exception Return）

- eret 指令
  - ELR_EL1 -> PC，恢复PC状态
  - SPSR_EL1 -> PSTATE，恢复处理器状态
  - 降至EL0，硬件自动使用SP_EL0作为栈指针
  - 恢复执行

x86-64的中断和异常处理

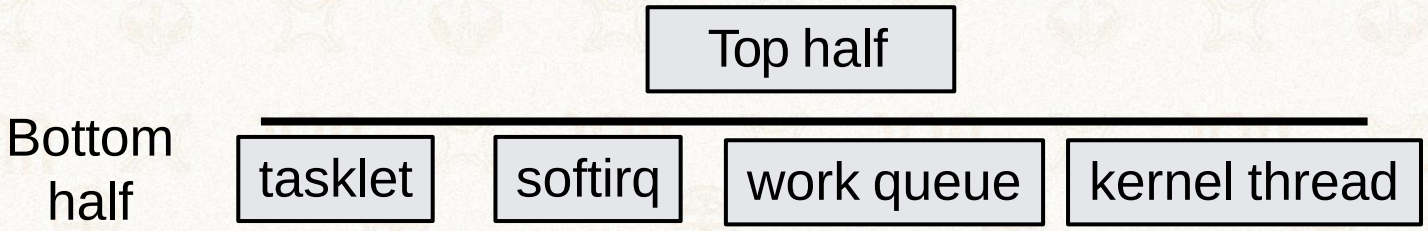
- 进入异常
  - 硬件会将上下文信息和错误码存储在内核栈上
- 用异常向量表寻找handler
  - 不分级
  - 异常向量表中存handler的地址
- iret返回
  - 恢复程序上下文
  - 从内核态返回用户态
  - 继续执行用户程序

中断处理是不能做太多事情的

- 怎么办?

案例：Linux的中断处理理念

- 在中断处理中做尽量少的事
- 推迟非关键行为
- 结构：Top half & Bottom half
  - Top half : 做最少的工作后返回
  - Bottom half： 推迟处理 (softirq, tasklets, 工作队列， 内核线程)



可选为实验课专题报告题目

大纲

- 特权级模型

- 设备与中断
  - 基本概念
  - 中断如何产生
  - 设备的内存映射
  - 异常处理

谢谢

微信：suyuxin
钉钉：苏玉鑫
B站：https://space.bilibili.com/502854403
软工集市课程专区：https://ssemarket.cn/new/course