

# 中断、异常与系统调用

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

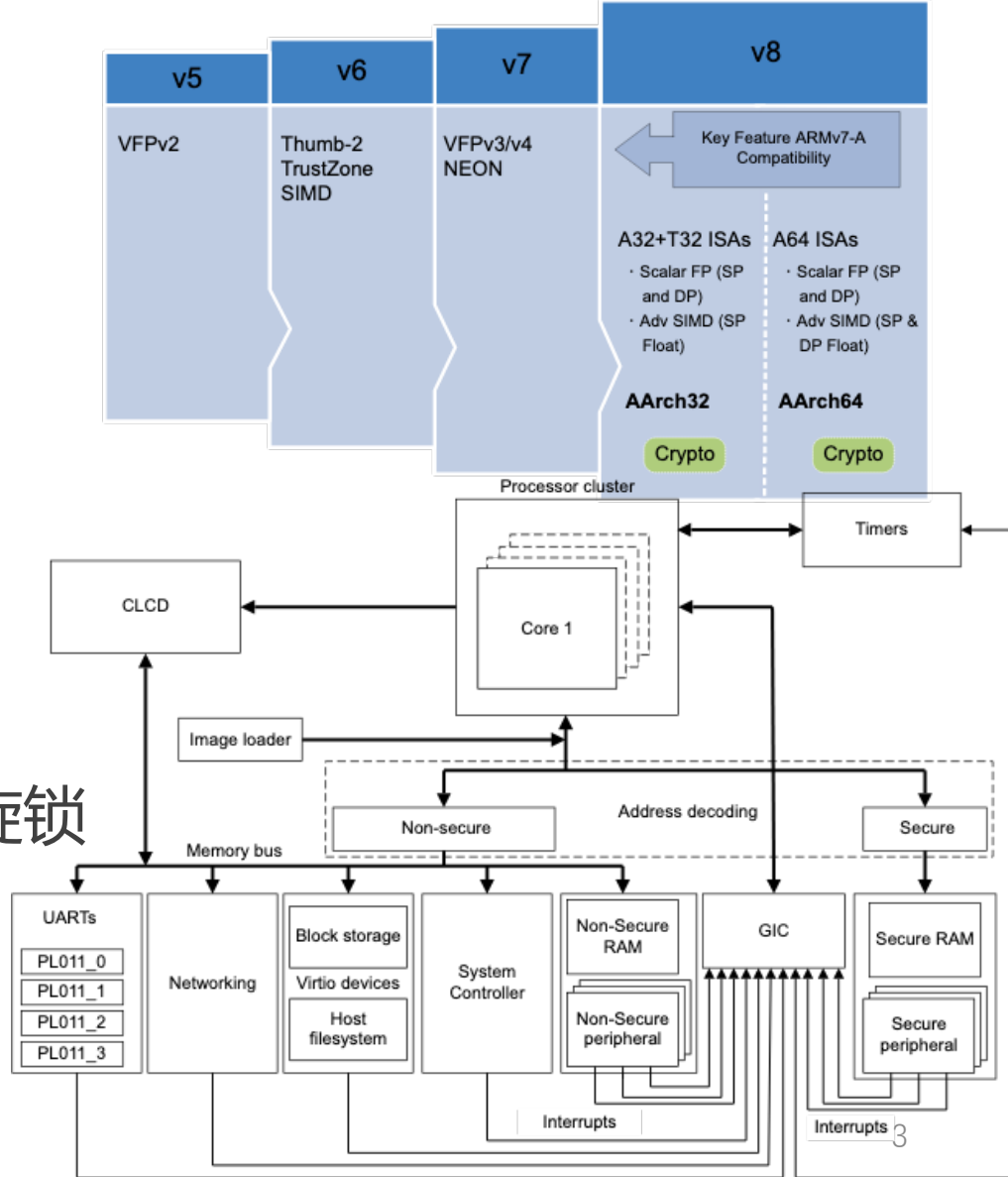
<https://ipads.se.sjtu.edu.cn>

# 版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源
  - 资料来自上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
  - 完整文本：<https://creativecommons.org/licenses/by/4.0/legalcode>

# 回顾：ARMv8

- 扩大物理寻址
  - 4GB以外的物理地址
- 64位虚拟地址
- 自动事件信号
  - 低功耗、高性能的自旋锁
- 硬件加速加密
- 新的异常模型



# 回顾：RISC vs CISC

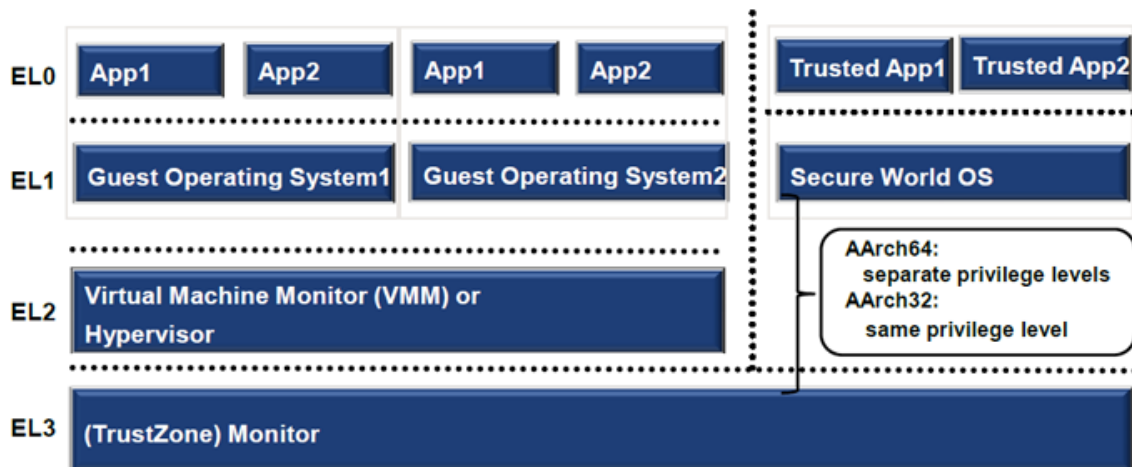
	RISC (Aarch64)	CISC (x86-64)
指令长度	定长	变长
寻址模式	寻址方式单一	多种寻址方式
内存操作	load/store	mov
实现	增加通用寄存器数量	微码
指令复杂度	简单	复杂
汇编复杂度	复杂	简单
中断响应	快	慢
功耗	低	高
处理器结构	简单	复杂

# 回顾：特权级/ARM (Exception Level)

低

- EL0: 用户态程序
- EL1: 内核
- EL2: hypervisor
- EL3: monitor

高



# 回顾：ChCore对异常级别的使用

低

- EL0 : ChCore中的用户程序
- EL1 : ChCore中的内核代码
- EL2 : 未使用
- EL3 : 未使用

高

# 回顾：设备上电后.....

- **OS如何启动**
  - 不同ARM设备启动OS的方式不同，例
  - Raspi 3: GPU进行初始化后从SD卡上加载
  - Hikey970: 通过BL链和UEFI加载
- **OS启动时进行硬件初始化工作，并开启页表**
- **进入内核**

# 问题：用户如何通过OS和外设交互的？

- 简单场景：用户是如何获得键盘输入？



# OS接受键盘输入

- 键盘等外设具有控制器和缓冲区，将输入存入缓冲区
- OS获取该输入的可能方法.....
  - 轮询：OS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

思考：就键盘而言，选用哪种方式更好？为什么？

思考：在何种场景下，使用“轮询”的效果更好？

# 用户从OS获取输入

- 为了给用户屏蔽硬件细节、管理硬件，用户一般不直接操控设备
- `getchar()` 中调用 `read()` 请求OS将获取的输入返回给用户
- 应用能否绕过OS直接控制硬件？
  - 用户态驱动 (但是也需要操作系统实现配置)

# 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、Watchdog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：用户程序 -> 操作系统
  - HVC：客户系统 -> 虚拟机管理器
  - SMC：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）**

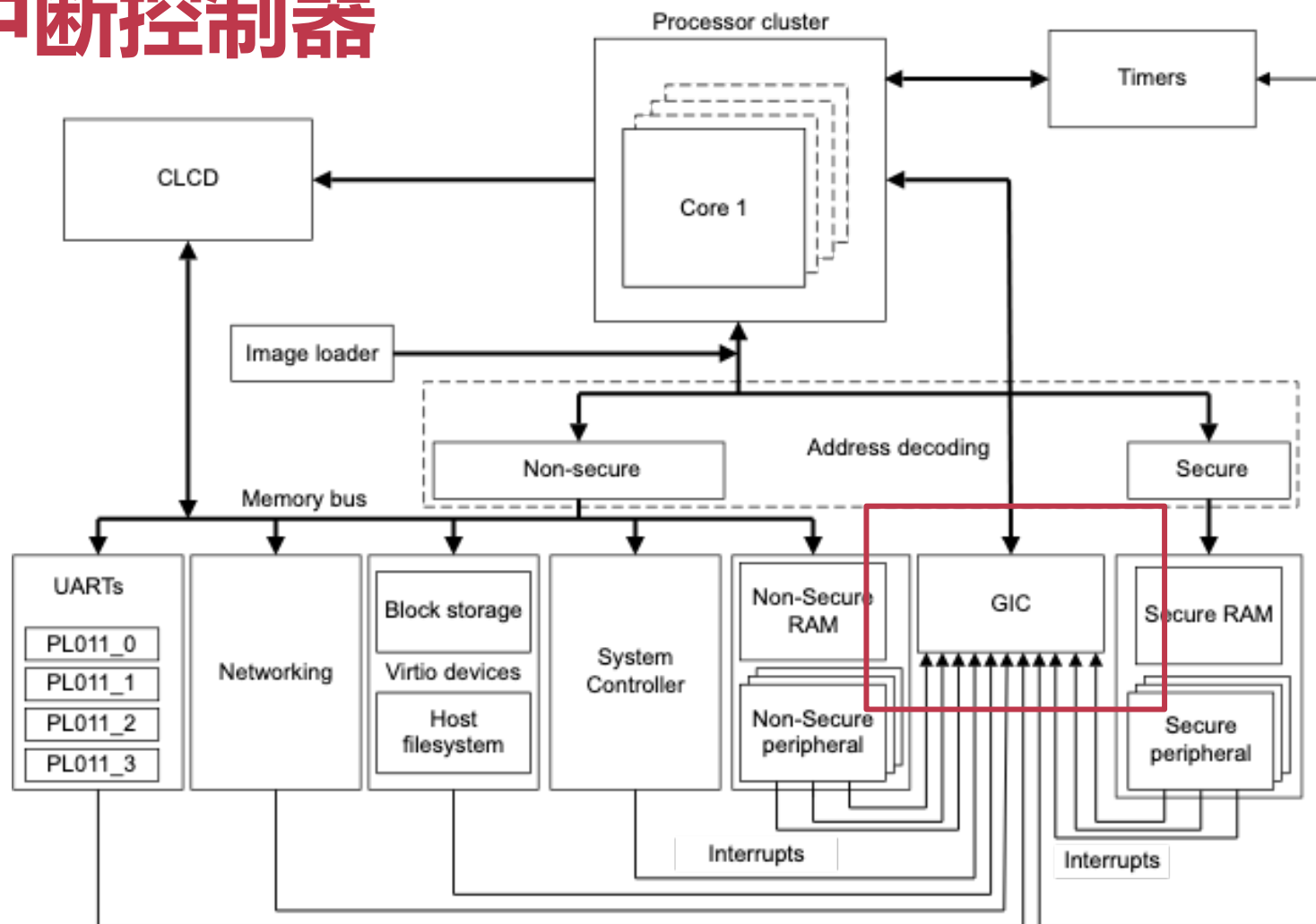
- **错误（Fault）**：如缺页异常（可恢复）、段错误（不可恢复）等
- **陷阱（Trap）**：无需恢复，如断点（int 3）、**系统调用**（int 80）
- **中止（Abort）**：严重的错误，不可恢复（机器检查）

AArch64的异步异常与x86-64的

# 中断的产生



# 中断控制器



# 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**

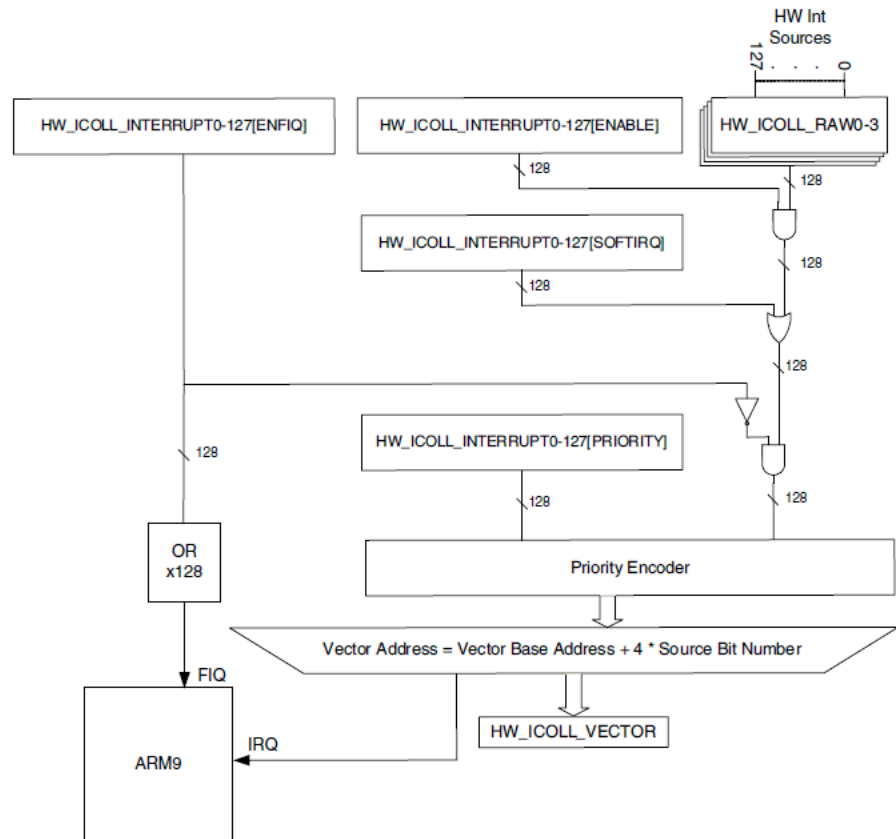
# AArch64中断的分类

- **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
  - **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
- 连接CPU的不同引脚
- 可在**中断控制器** ( Interrupt Controller ) 中配置

# 早期的ARM中断控制器

- 厂商指定模型

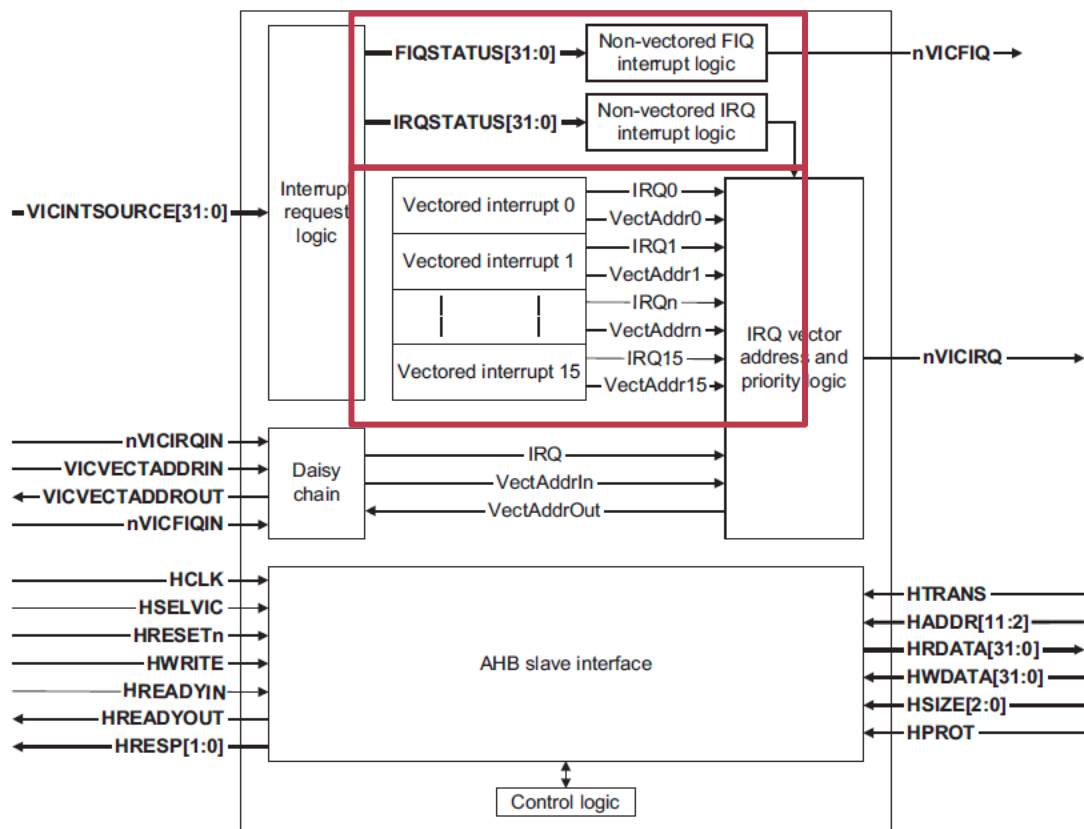
- 将中断路由给处理器的功能由SoC厂商实现
- FIQ直接发给处理器，IRQ根据优先级进行分发
- 如Freescale/NXP iMx233/iMx28系列中的ICOLL



# 早期的ARM中断控制器

- 向量中断控制器 VIC (Vector Interrupt Controller)
  - ARM提出
  - 放在AMBA (Advanced Microcontroller Bus Architecture)高速总线上
  - 32种非向量中断 (不同中断相同的处理入口)
  - 16种向量中断 (不同中断不同的处理入口)

# 向量中断与非向量中断



# GIC: 通用中断控制器

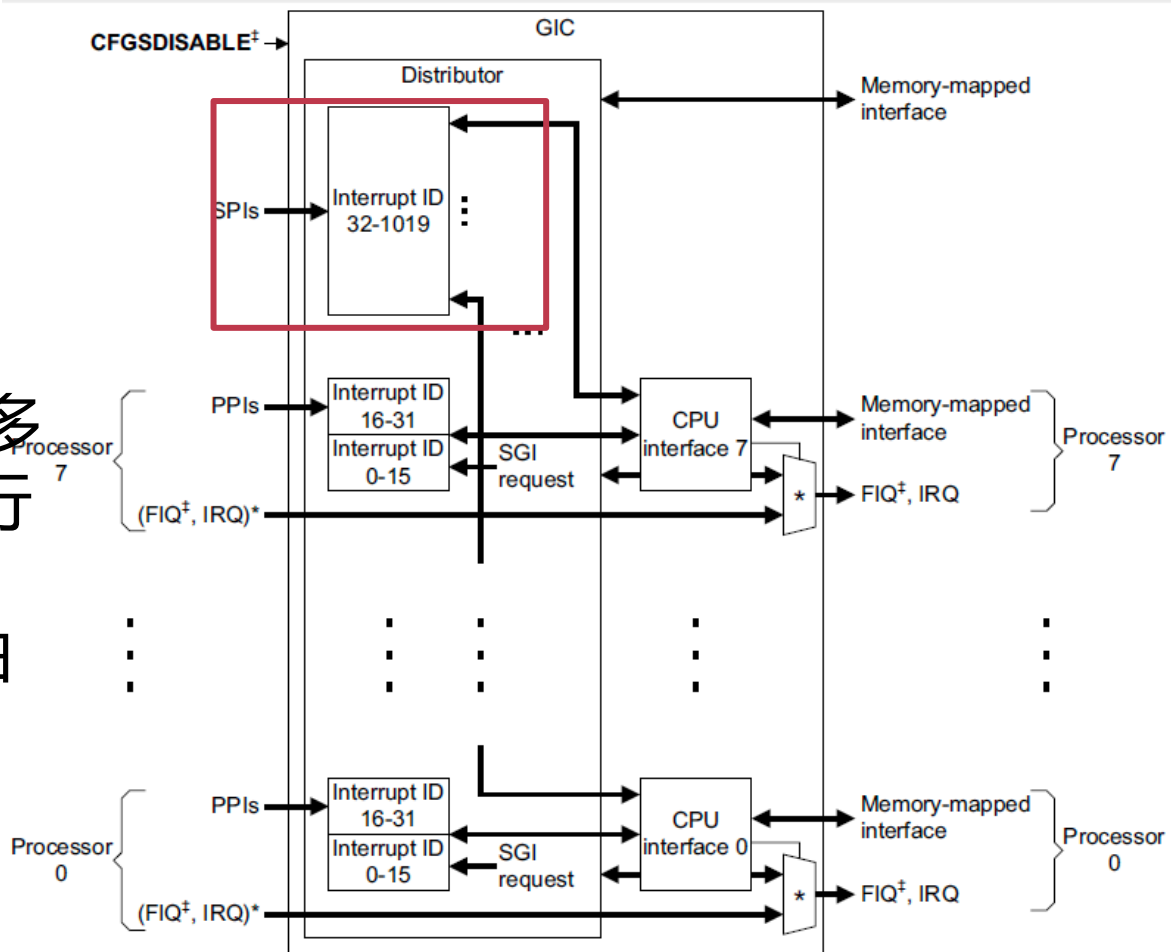
- 中断类型变多，将中断分发给不同的核（对称或非对称）进行处理
- 主要功能
  - 分发：管理所有中断、决定优先级、路由
  - CPU接口：给每个CPU核有对应的接口

# GIC中断来源

SPI：共享外围中断

- 可以被路由到一个或多个核，找到可用的核进行处理

- Distributor可配置路由
- 如UART中断
- 中断ID：32-1019





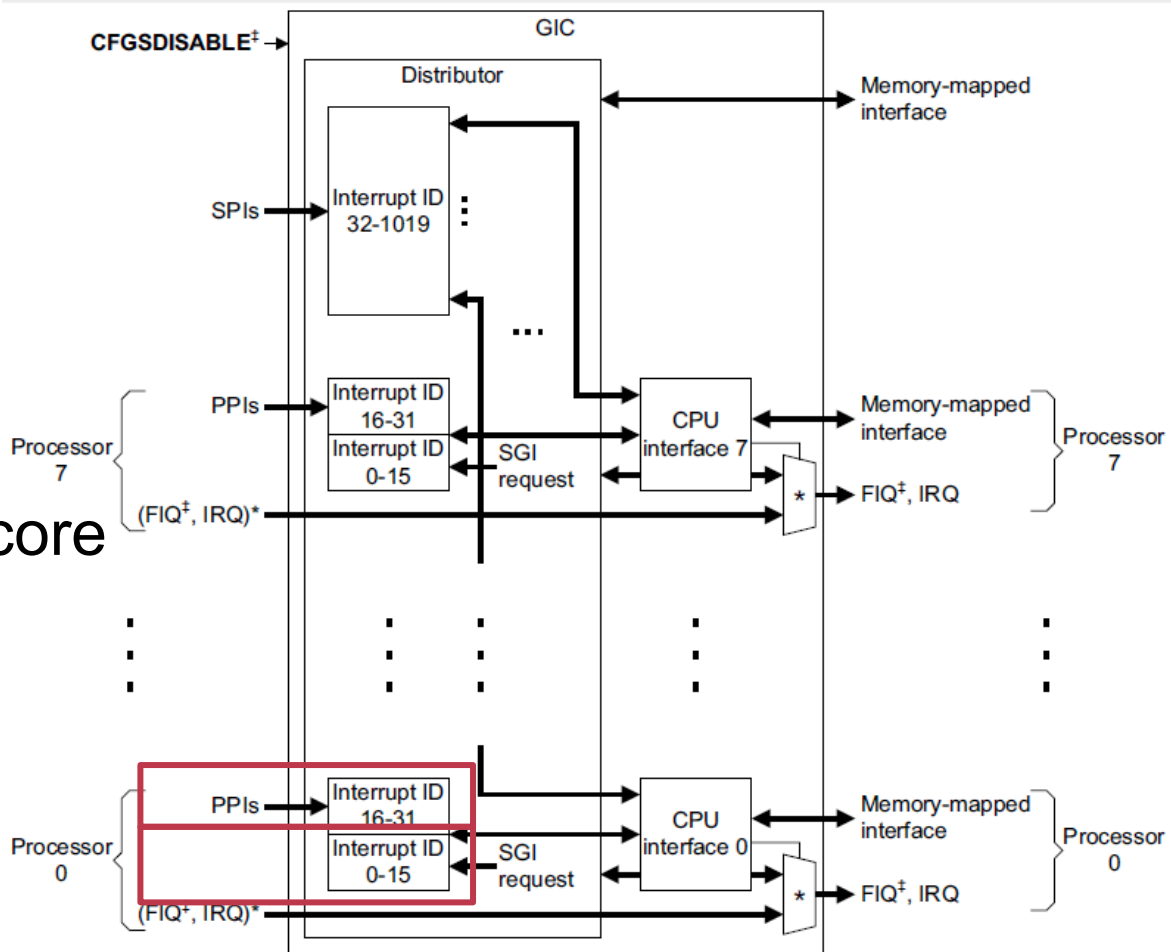
# GIC中断来源

PPI：私有外围中断

- 指定核处理
- Distributor可配置路由
- 如WatchingDog -> A5 core
- 中断ID：16-31

SPI：软件产生中断

- 核间通信
- 中断ID：0-15



# GIC的路由配置 – 以启用timer为例

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

使用MMIO，设置GIC中寄存器，启用timer

```
#define GICD_ISENBLER (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_ISENBLER, 0x08000000);
    timer_init();
}
```

# 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]);

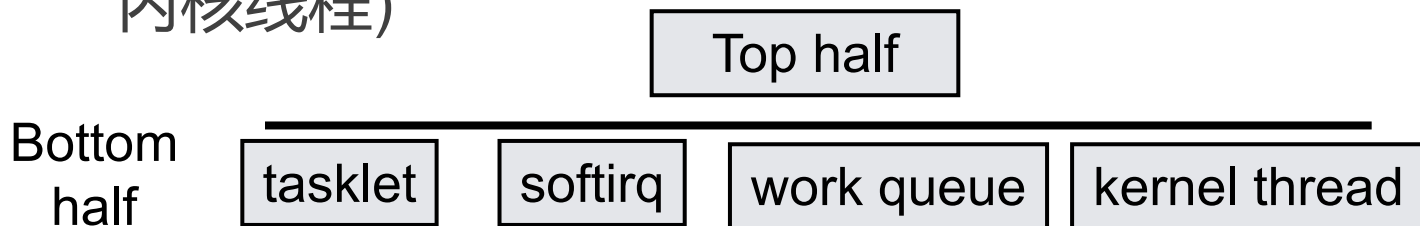
    irq = 1 << ctzl(irq_src);
    switch (irq) {
        case INT_SRC_TIMER3:
            handle_timer_irq();
            break;
        default:
            kinfo("Unsupported IRQ %d\n", irq);
    }
    return;
```

# 讨论：中断处理不能做太多的事情？

- 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)



# Top Half : 马上做

- **最小的、公共行为**
  - 保存寄存器、屏蔽其他中断
  - 恢复寄存器，返回原来场景
- **最重要：调用合适的由硬件驱动提供的中断处理handler**
- **因为中断被屏蔽，所以不要做太多事情（时间、空间）**
- **使用将请求放入队列，或者设置标志位将其他处理推迟到  
bottom half**

## Top Half : 找到handler

- 现代处理器中，多个I/O设备共享一个IRQ和中断向量
- 多个ISR (interrupt service routines)可以结合在一个向量上
- 调用每个设备对应该IRQ的ISR

## Bottom Half : 延迟完成

- 提供一些推迟完成任务的机制
  - softirqs
  - tasklets (建立在softirqs之上)
  - 工作队列
  - 内核线程
- 这些工作可以被中断



# 注意：中断处理没有进程上下文

为什么？

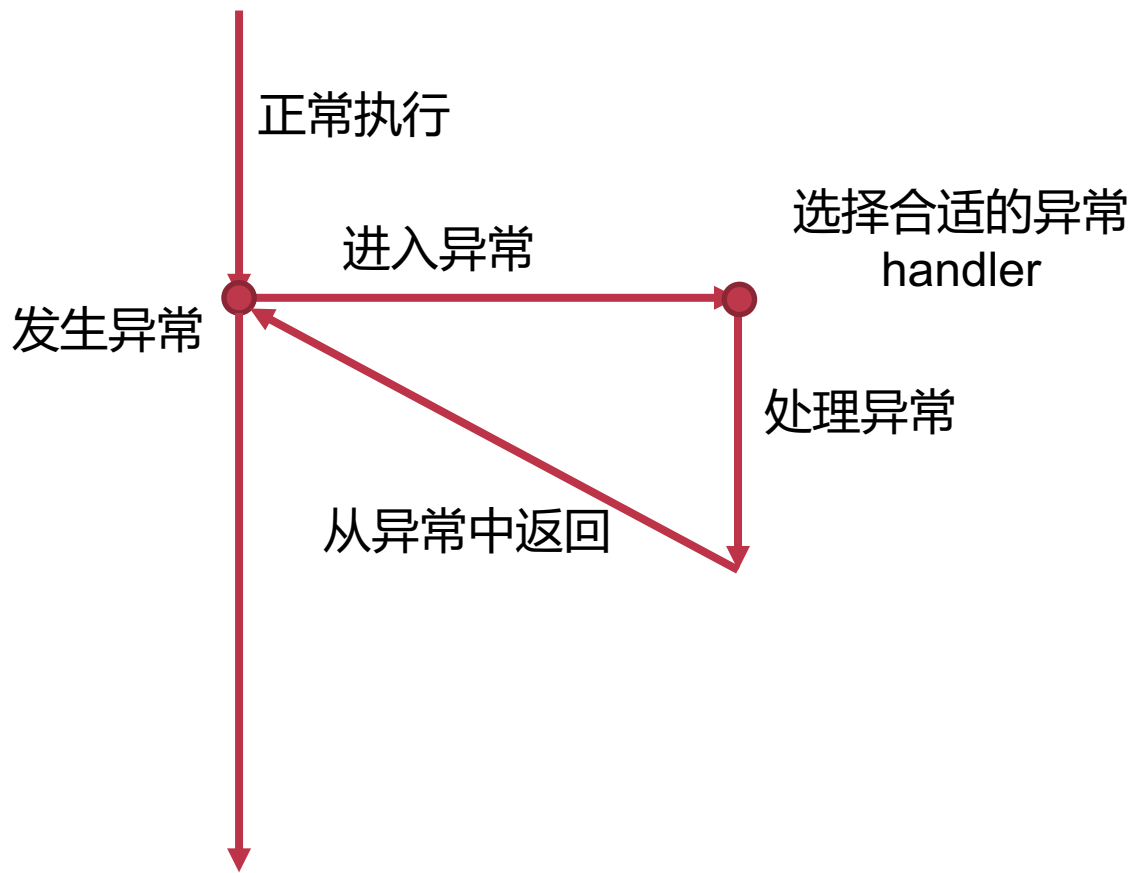
- 中断（和异常相比）和具体的某条指令无关
- 也和中断时正在跑的进程、用户程序无关
- 中断处理handler不能睡眠！

# 中断处理中的一些约束

- 不能睡眠
  - 或者调用可能会睡眠的任务
- 不能调用`schedule()` 调度
- 不能释放信号或调用可能睡眠的操作
- 不能和用户地址空间交换数据

## 异常的处理

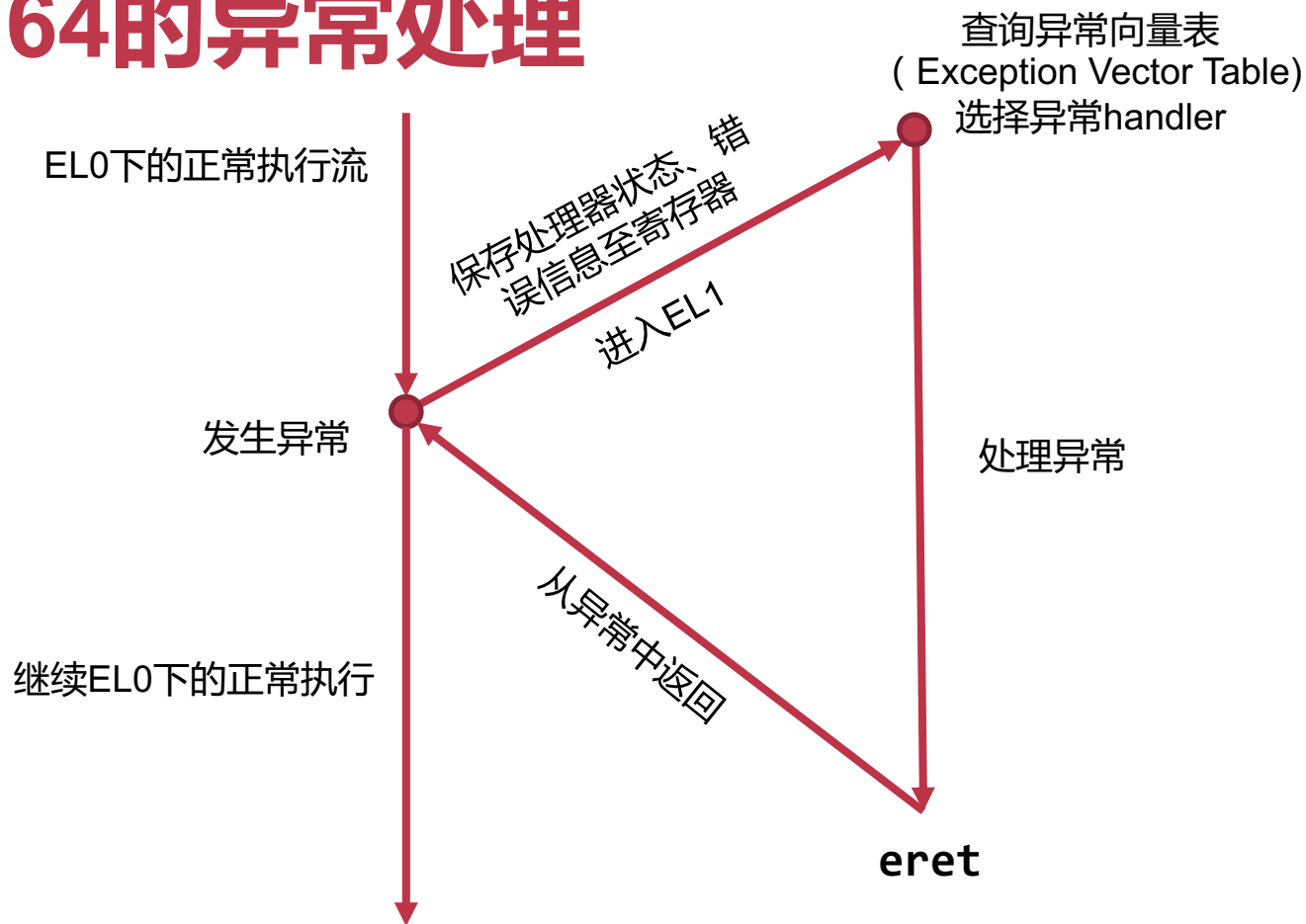
# 处理流程



# 异常处理必做事项

- **进入异常时**
  - 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
- **处理异常时**
  - 需获得关于异常的信息，如系统调用参数、错误原因等
- **从异常返回时**
  - 需恢复处理器状态，返回低特权级，继续正常执行流

# AArch64的异常处理



\* 以EL0 -> EL1为例，后同

# 进入异常 – 信息保存

- **异常发生后，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 ->
    - 1) Serror与异常：Exception Syndrome Register ( ESR\_EL1 )
    - 2) 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

# 进入异常 – 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 )

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	

地址	异常类型	异常发生时处理器状态	
+ 0x000	Synchronous	EL1 EL0作为SP	
AArch64状态下，EL0中发生同步异常： 内存访问权限错误			
+ 0x280	IRQ		
+ 0x300	FIQ		使用SP_EL1作为SP
+ 0x380	SError		
+ 0x400	Synchronous	EL0 运行于AArch64状态	
+ 0x480	IRQ		
+ 0x500	FIQ		
• 保存上下文、错误信息 • 切换至EL1 • PC <- VBAR_EL1 + 0x400 • 开始执行		EL0 运行于AArch32状态	
+ 0x580	SError		
+ 0x600	Synchronous		
+ 0x680	IRQ		
+ 0x700	FIQ		
+ 0x780	SError		

43

VBAR\_EL1



- 保存上下文、错误信息
- 切换至EL1
- $PC \leftarrow VBAR\_EL1 + 0x400$
- 开始执行

# 从异常中返回 ( 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

# ChCore异常向量表配置

```
.align 11
EXPORT(el1_vector)
    exception_entry sync_el1t
    exception_entry irq_el1t
    exception_entry fiq_el1t
    exception_entry error_el1t

    exception_entry sync_el1h
    exception_entry irq_el1h
    exception_entry fiq_el1h
    exception_entry error_el1h

    exception_entry sync_el0_64
    exception_entry irq_el0_64
    exception_entry fiq_el0_64
    exception_entry error_el0_64

    exception_entry sync_el0_32
    exception_entry irq_el0_32
    exception_entry fiq_el0_32
    exception_entry error_el0_32
```

```
BEGIN_FUNC(set_exception_vector)
    adr    x0, el1_vector
    msr    vbar_el1, x0
    ret
END_FUNC(set_exception_vector)
```

```
.macro    exception_entry label
    .align 7
    b      label
.endm
```

每个异常向量均为简单的跳转指令

# ChCore异常处理器

```
.align 11
EXPORT(el1_vector)

exception_entry sync_el1t
exception_entry irq_el1t
exception_entry fiq_el1t
exception_entry error_el1t

exception_entry sync_el1h
exception_entry irq_el1h
exception_entry fiq_el1h
exception_entry error_el1h

exception_entry sync_el0_64
exception_entry irq_el0_64
exception_entry fiq_el0_64
exception_entry error_el0_64

exception_entry sync_el0_32
exception_entry irq_el0_32
exception_entry fiq_el0_32
exception_entry error_el0_32
```

```
sync_el1t:
    handle_entry    1, SYNC_EL1t
```

```
.macro handle_entry el, type
    exception_enter    /* 保存内核软件需要的上下文 */
    mov x0, #\type     /* 准备参数 */
    mrs x1, esr_el1
    mrs x2, elr_el1
    bl handle_entry_c  /* 调用C语言异常处理器，实际处理异常 */
    exception_return    /* 恢复内核软件需要的上下文 */
.endm
```

# ChCore异常处理器

```
.macro handle_entry el, type
    exception_enter    /* 保存内核软件需要的上下文 */
    mov x0, #\type     /* 准备参数 */
    mrs x1, esr_el1
    mrs x2, elr_el1
    bl handle_entry_c /* 调用C语言异常处理器, 实际处理异常 */
    exception_return    /* 恢复内核软件需要的上下文 */
.endm
```

```
void handle_entry_c(int type, u64 esr, u64 elr)
{
    /* ec: 异常类型 (exception class) */
    u32 esr_ec = GET_ESR_EL1_EC(esr);

    switch (esr_ec) {
        case XXX:
            /* 处理异常XXX */
            /* ... */
        }
    }
}
```

```
.macro exception_enter
    /* 保存常规寄存器至内核栈 */
    sub sp, sp, #ARCH_EXEC_CONT_SIZE
    stp x0, x1, [sp, #16 * 0]
    stp x2, x3, [sp, #16 * 1]
    /* ... */
    stp x28, x29, [sp, #16 * 14]
    /* 保存一些特殊寄存器至内核栈 */
    mrs x21, sp_el0
    mrs x22, elr_el1
    mrs x23, spsr_el1
    stp x30, x21, [sp, #16 * 15]
    stp x22, x23, [sp, #16 * 16]
.endm
```

```
.macro exception_exit
    /* 从内核栈恢复特殊寄存器 */
    ldp x22, x23, [sp, #16 * 16]
    ldp x30, x21, [sp, #16 * 15]
    msr sp_el0, x21
    msr elr_el1, x22
    msr spsr_el1, x23
    /* 从内核栈恢复常规寄存器 */
    ldp x0, x1, [sp, #16 * 0]
    ldp x2, x3, [sp, #16 * 1]
    /* ... */
    ldp x28, x29, [sp, #16 * 14]
    add sp, sp, #ARCH_EXEC_CONT_SIZE
    /* 结束异常执行 */
    eret
.endm
```



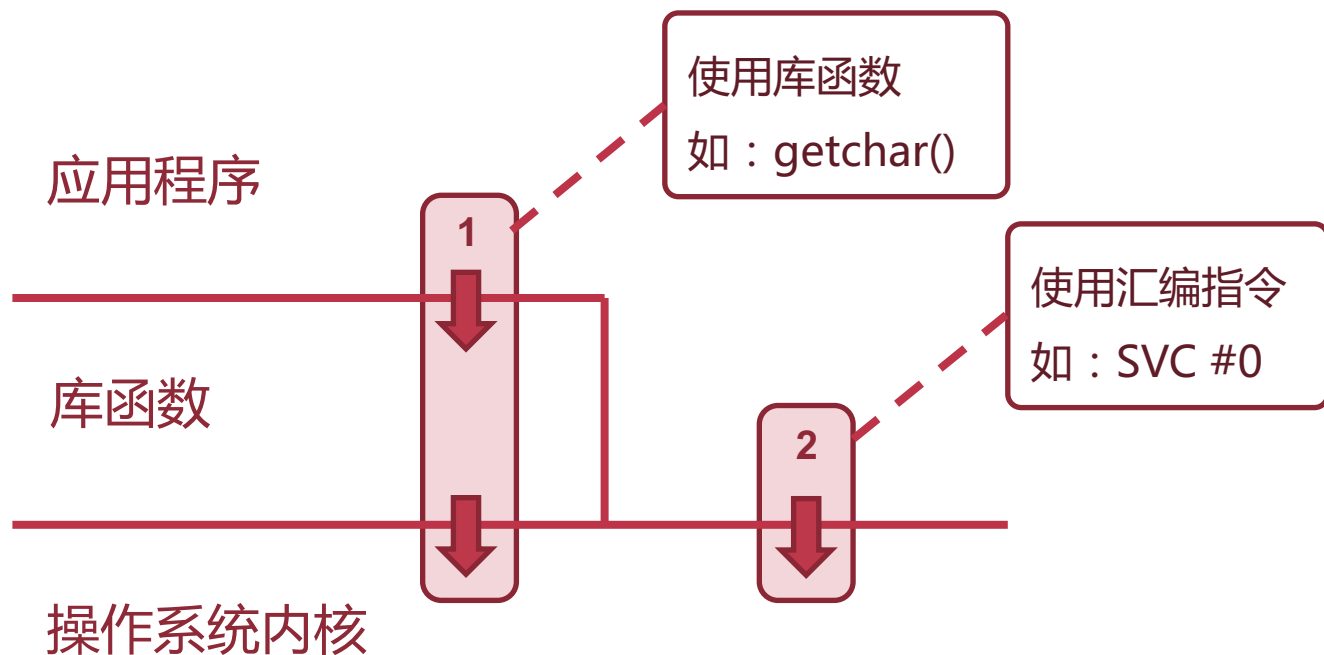


# 系统调用

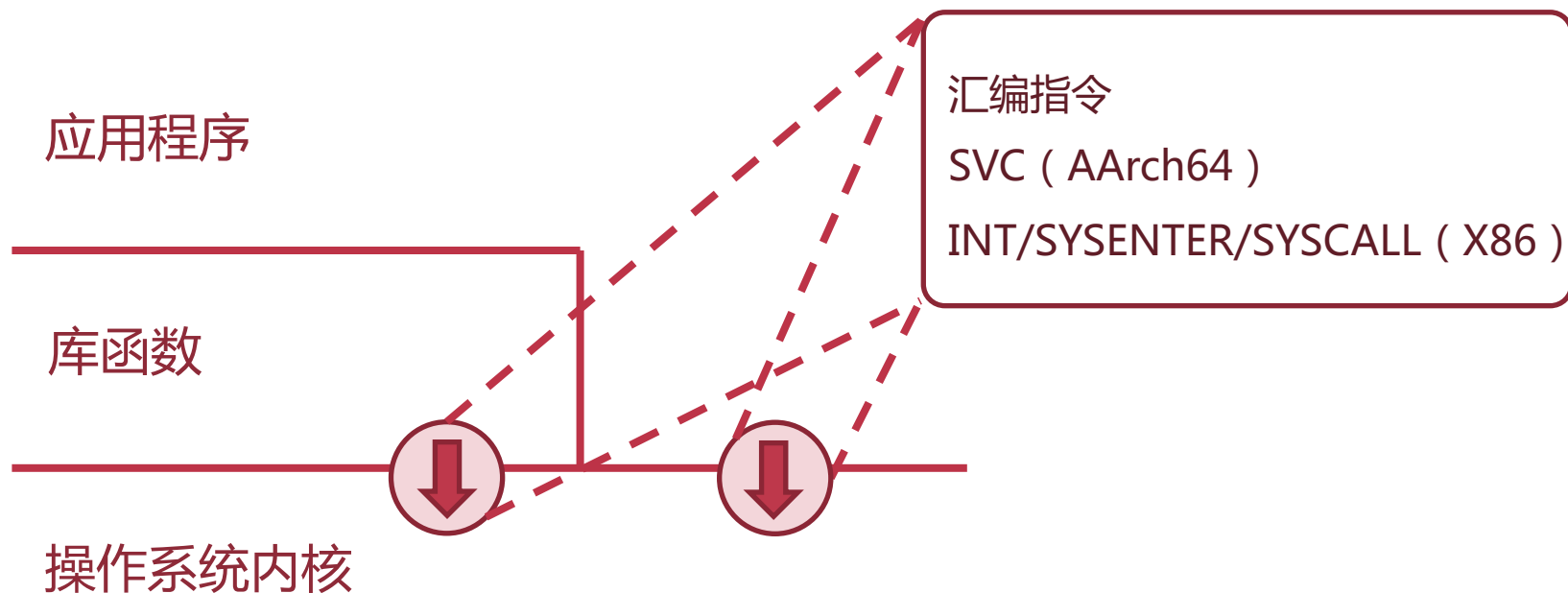
# 系统调用 ( Syscall )

- 指运行在用户空间的程序向操作系统内核请求需要更高权限运行的服务
- 系统调用提供用户程序与操作系统之间的接口

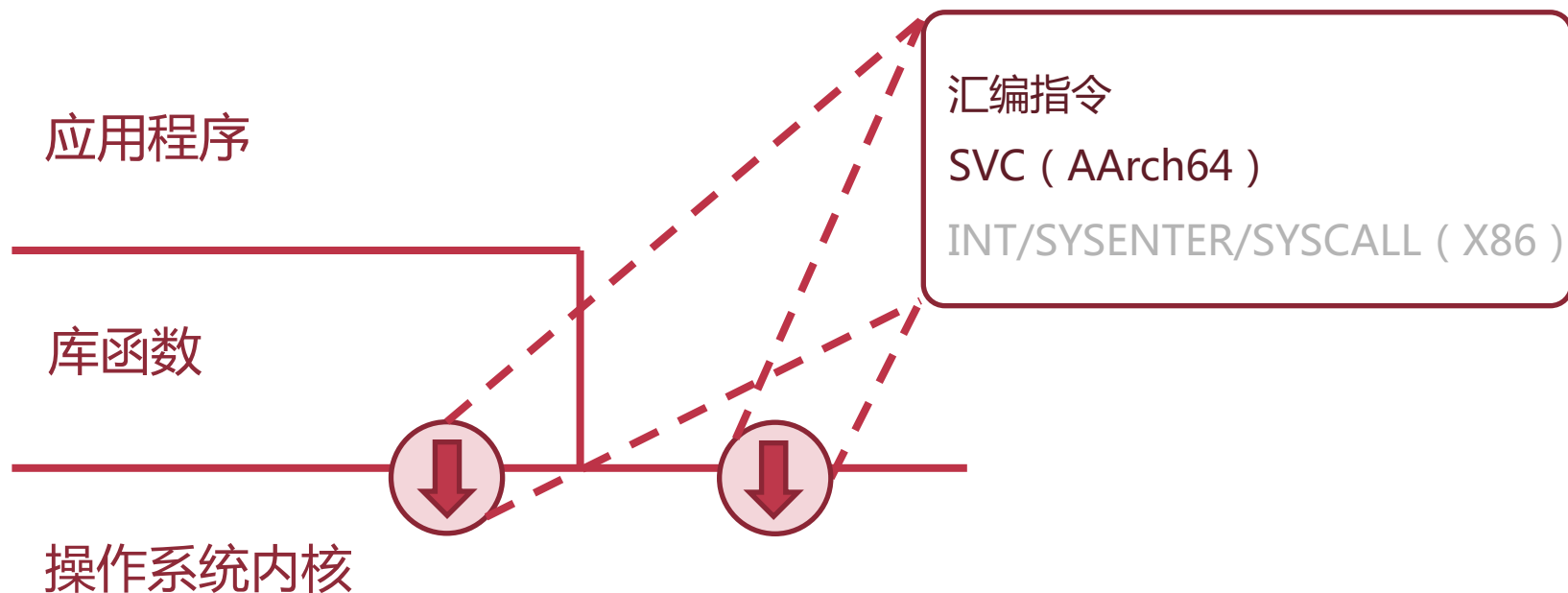
# 系统调用方式 – 程序员视角



# 系统调用方式 – 硬件视角



# 系统调用方式 – 硬件视角



# Linux的常用系统调用

Number	Name	Description	Number	Name	Description
1	exit	Terminate process	27	alarm	Set signal delivery alarm clock
2	fork	Create new process	29	pause	Suspend process until signal arrives
3	read	Read file	37	kill	Send signal to another process
4	write	Write file	48	signal	Install signal handler
5	open	Open file	63	dup2	Copy file descriptor
6	close	Close file	64	getppid	Get parent's process ID
7	waitpid	Wait for child to terminate	65	getpgrp	Get process group
11	execve	Load and run program	67	sigaction	Install portable signal handler
19	lseek	Go to file offset	90	mmap	Map memory page to file
20	getpid	Get process ID	106	stat	Get information about file

# Linux追踪系统调用

- 每当有系统调用产生时，Linux可打印发生的系统调用、系统调用的参数和系统调用的返回值
- `ptrace()`可追踪Linux中的系统调用情况
  - 广泛应用在各种debugger中
- 命令行中
  - `strace`追踪系统调用
  - `ltrace`追踪库函数的调用

# 使用strace

```
int main() {  
    write(1, "Hello world!\n", 13);  
}
```

```
$ strace -o hello.out ./hello
```

```
execve("./hello2", [ "./hello2" ], [ /* 59 vars */ ]) = 0  
uname({sys="Linux", node="kiwi", ...}) = 0  
brk(0) = 0xca9000  
brk(0xcaa1c0) = 0xcaa1c0  
arch_prctl(ARCH_SET_FS, 0xca9880) = 0  
brk(0xccb1c0) = 0xccb1c0  
brk(0xcccc000) = 0xcccc000  
write(1, "Hello world!\n", 13) = 13  
exit_group(13) = ?
```



# 案例分析：ChCore中的usys\_exit()

- **usys\_exit**：标明当前线程已返回

```
void main(int argc, char *argv[], char *envp[])
{
    printf("hello, world\n");
    usys_exit(0);
}
```

```
void usys_exit(int ret)
{
    syscall(SYS_exit, ret, 0, 0, 0, 0, 0, 0, 0, 0);
}
```

# 案例分析：ChCore中的usys\_exit()

```
.align 11
EXPORT(el1_vector)
    exception_entry sync_el1t
    exception_entry irq_el1t
    exception_entry fiq_el1t
    exception_entry error_el1t

    exception_entry sync_el1h
    exception_entry irq_el1h
    exception_entry fiq_el1h
    exception_entry error_el1h

    exception_entry sync_el0_64
    exception_entry irq_el0_64
    exception_entry fiq_el0_64
    exception_entry error_el0_64

    exception_entry sync_el0_32
    exception_entry irq_el0_32
    exception_entry fiq_el0_32
    exception_entry error_el0_32
```

```
syscall(u64 sys_no, u64 arg0, u64 arg1, u64 arg2, /* ... */)

u64 ret = 0;
asm volatile ("mov x8, %1\n"
              "mov x0, %2\n"
              "mov x1, %3\n"
              "mov x2, %4\n"
              /* ... */
              "svc #0\n"
              "mov %0, x0\n"
              : "=r" (ret)
              : "r"(sys_no), "r"(arg0), "r"(arg1), "r"(arg2),
              /* ... */
              : "x0", "x1", "x2", "x3", "x4", "x5", "x6", "x7", "x8"
              );
return ret;
```

# 系统调用与安全

- **AArch64使用寄存器传参，个数有限**
  - 如ChCore的系统调用支持使用寄存器X0-X7最多8个参数
- **若系统调用需要更多参数如何处理？**
  - 使用结构体打包参数，并将结构体的指针作为参数
- **问题：内存安全性**
  - 作为参数的指针必须经过检测！
  - 指向NULL -> kernel crash
  - 指向内核内存 -> 安全漏洞

# 用户指针检测

- **完备的指针检测十分耗时**
  - 需要遍历用户进程的所有合法内存区域进行检测
- **Linux解决方法：非全面检查**
  - Linux仅**初步检测**用户指针是否属于对应进程的用户内存区域的**最大可能边界**
  - 即使通过初步检测，用户指针**仍然可能非法**（如指向尚未分配的栈空间等）
  - 直接将**非法的指针**交给内核使用会导致内核出现**页错误**，内核态的页错误通常以为着bug，内核会打印异常信息并中止用户进程
  - Linux采用了一些复杂机制来防止这一情况发生

# 处理用户指针问题

- **内核代码仅使用特定代码片段访问用户指针（如copy\_from\_user）**
  - 由访问用户指针而导致内核内存错误的代码段是确定的
- **当内核发生页异常（Page Fault）时，内核会检查异常发生的PC**
  - 若异常发生的PC属于访问用户指针的代码段，Linux尝试对其进行修复
  - 若不属于，则报告问题并终止用户程序
- **然而**
  - Linux中很多地方违反了这一规定，导致了許多安全漏洞

# 系统调用与性能

- 系统调用会造成大量性能开销
- 硬件优化：新的系统调用指令
  - X86提出了syscall/sysenter/sysexit来代替int进行系统调用
- 软件优化
  - 开放性问题

# 下次课内容

- 操作系统结构