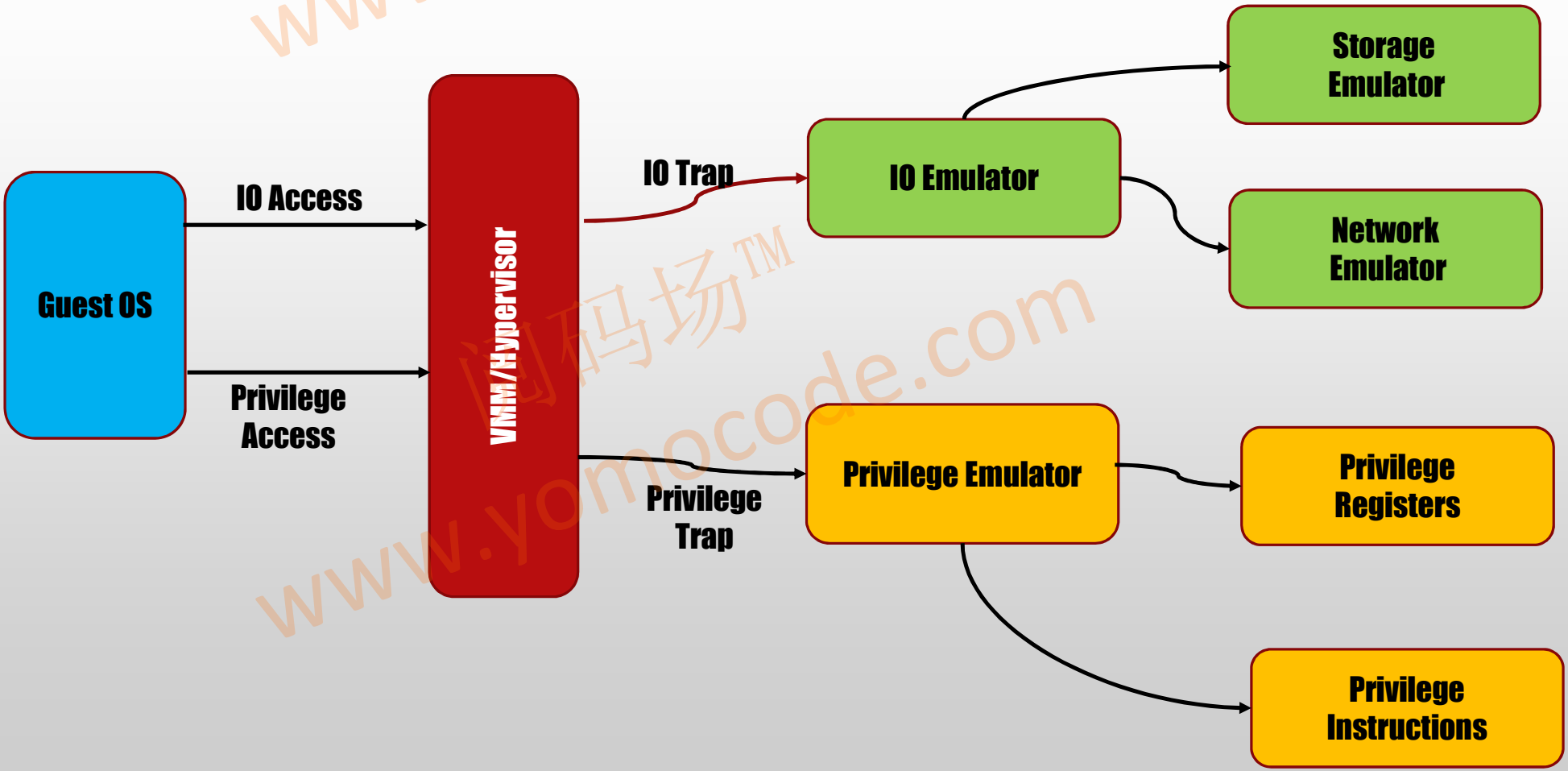
VIRTUALIZATION PRACTICE XEN, KVM...

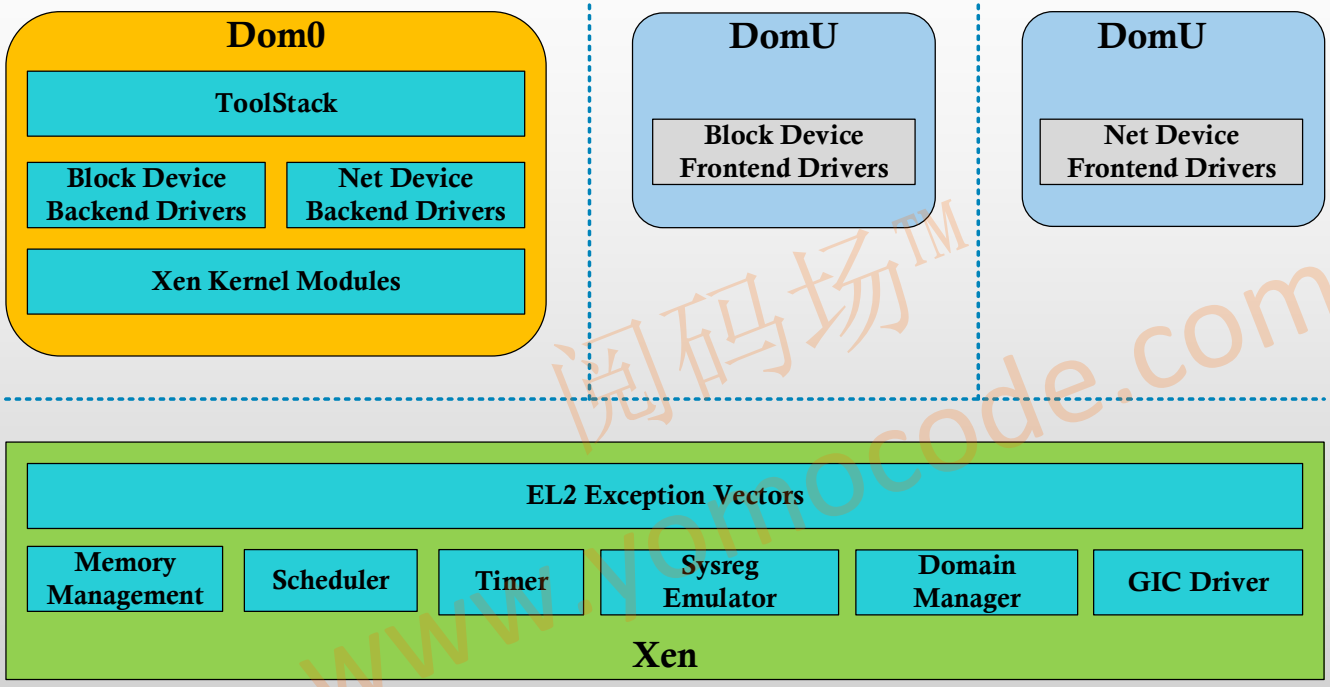
OPERATING PRINCIPLE OF HYPERVISOR

工作原理

TRAP AND EMULATOR



OVERVIEW OF XEN

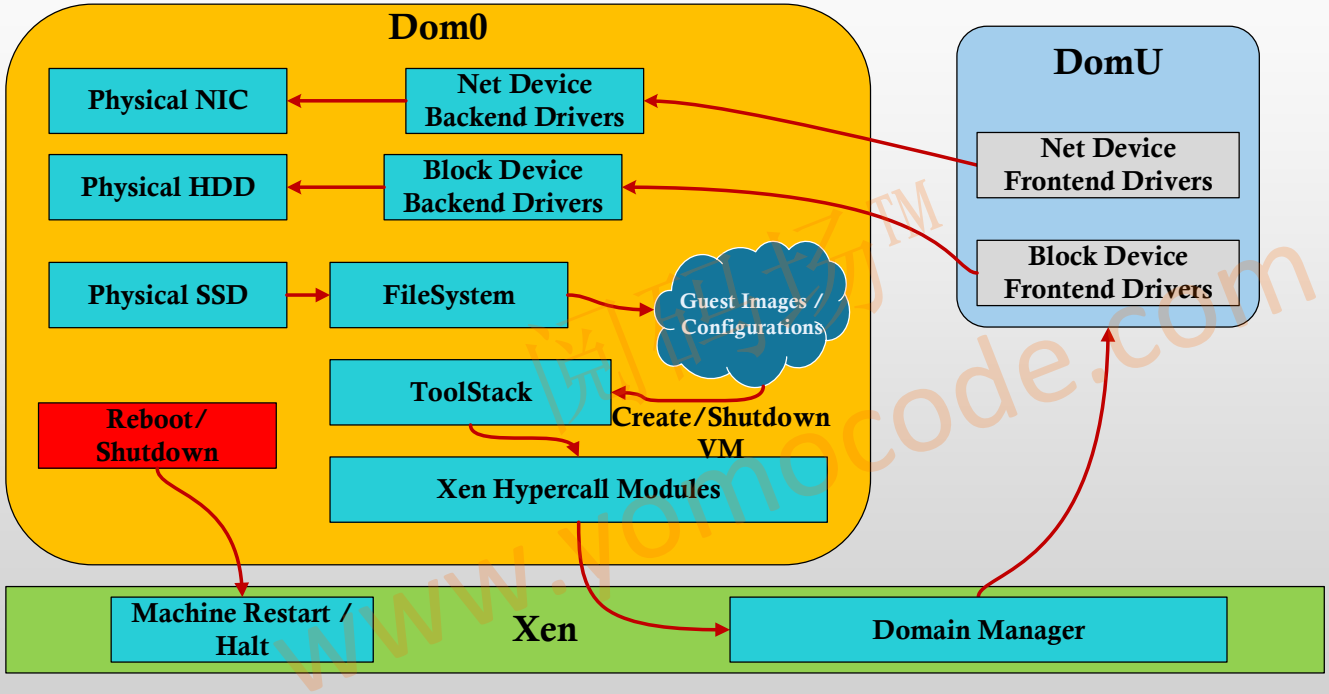


Xen Type1 Hypervisor控制所有的系统资源：

- 内存管理模块，
- 中断控制器驱动，
- 系统时钟驱动，
- 虚拟机调度器，
- 特权寄存器模块

但对**Linux**生态依赖比较严重的设备驱动和工具链，**Xen**将其放置于**Dom0**中，简化系统的实现，并提高兼容性。

DOMAIN#0 OF XEN



Domain0是**Xen**创建的一个拥有特殊权限的虚拟机：

- 它拥有大部分设备的访问权限，除非**Xen**做保留，
- 它能够访问**Xen**提供的所有**hypercall API**，用于支持工具链，
- 可以访问其它虚拟机的镜像和配置文件，
- 可以创建和删除其它虚拟机，
- 拥有重启和关闭整个物理主机的权限

INITIALIZATION OF HYPERVISOR

初始化

INITIALIZATION OF XEN

xen/arch/arm/arm64/head.S

```
        msr   DAIFSet, 0xf          /* Disable all interrupts */

        /* Save the bootloader arguments in less-clobberable registers */
        mov   x21, x0               /* x21 := DTB, physical address  */

        PRINT("- Boot CPU booting -\r\n")
……
        bl    check_cpu_mode
        bl    zero_bss
        bl    cpu_init
        bl    create_page_tables
        bl    enable_mmu

        /* We are still in the 1:1 mapping. Jump to the runtime Virtual Address. */
        ldr   x0, =primary_switched
        br    x0
primary_switched:
        bl    setup_fixmap

        b     launch
ENDPROC(real_start)
```

Xen遵循**Arm64 Linux Boot Protocol**，可以兼容支持**Arm64 Linux Image**格式的**Bootloader:**

- **X0**寄存器用于存放**Device Tree**的地址

* **x21**寄存器为**x0**的备份

INITIALIZATION OF XEN

xen/arch/arm/arm64/head.S

```
        msr   DAIFSet, 0xf          /* Disable all interrupts */

        /* Save the bootloader arguments in less-clobberable registers */
        mov   x21, x0               /* x21 := DTB, physical address  */

        PRINT("- Boot CPU booting -\r\n")
……
        bl    check_cpu_mode
        bl    zero_bss
        bl    cpu_init
        bl    create_page_tables
        bl    enable_mmu

        /* We are still in the 1:1 mapping. Jump to the runtime Virtual Address. */
        ldr   x0, =primary_switched
        br    x0
primary_switched:
        bl    setup_fixmap

        b     launch
ENDPROC(real_start)
```

1. 检查当前CPU所在的exception Level是否为EL2,
2. 在CPU_INIT中设置TCR和MAIR为建立页表做好准备,
3. Xen实际加载的地址和编译时指定的XEN_START_ADDR不一定移植，所以先建立一个简单的线性映射页表用于系统初始化，该页表后面会被替换,
4. 使能MMU

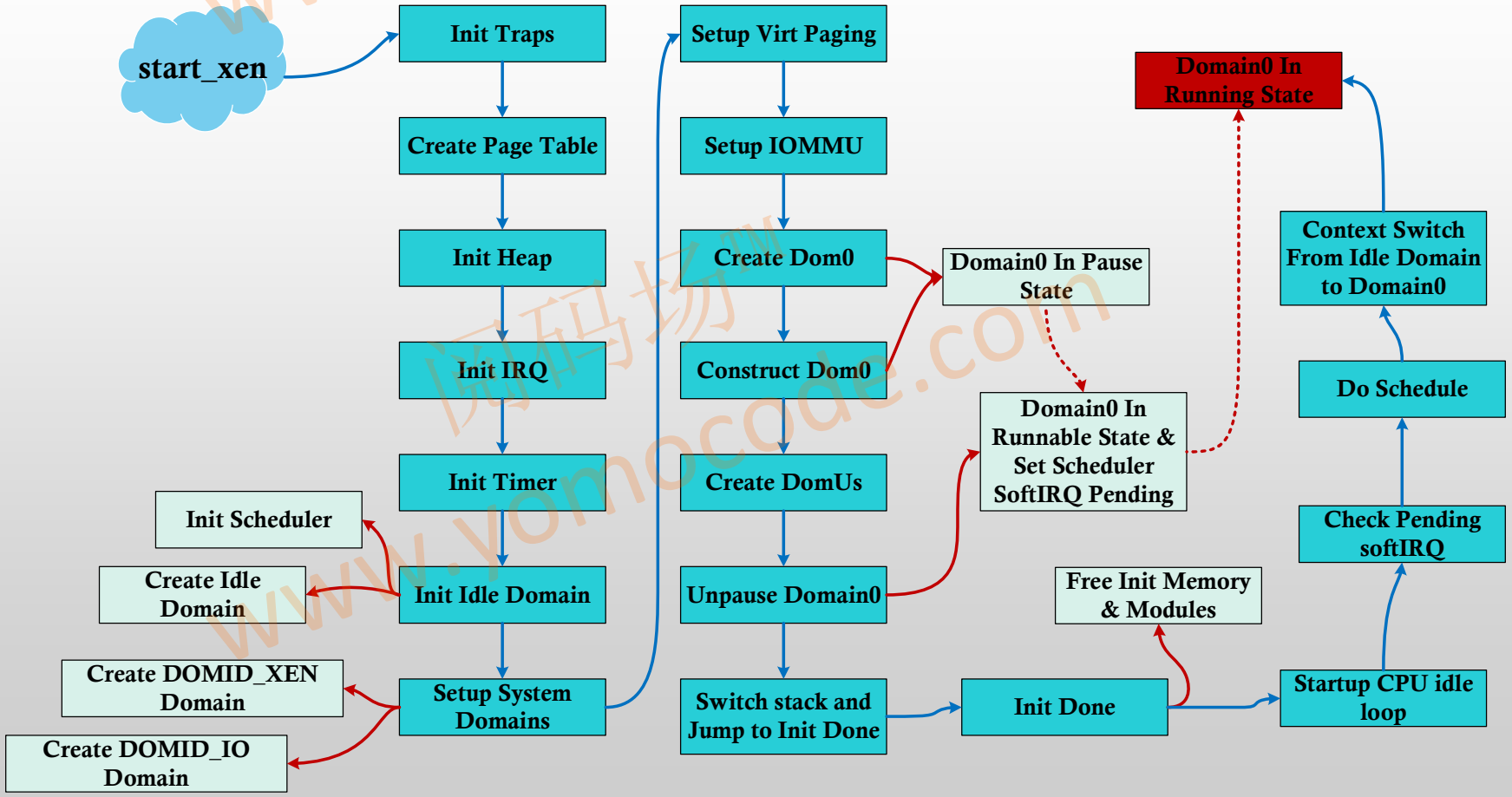
INITIALIZATION OF XEN

```
        msr   DAIFSet, 0xf          /* Disable all interrupts */

        /* Save the bootloader arguments in less-clobberable registers */
        mov   x21, x0               /* x21 := DTB, physical address  */

        PRINT("- Boot CPU booting -\r\n")
……
        bl    check_cpu_mode
        bl    zero_bss
        bl    cpu_init
        bl    create_page_tables
        bl    enable_mmu

        /* We are still in the 1:1 mapping. Jump to the runtime Virtual Address. */
        ldr   x0, =primary_switched
        br    x0
primary_switched:
        bl    setup_fixmap

        b     launch
ENDPROC(real_start)
```

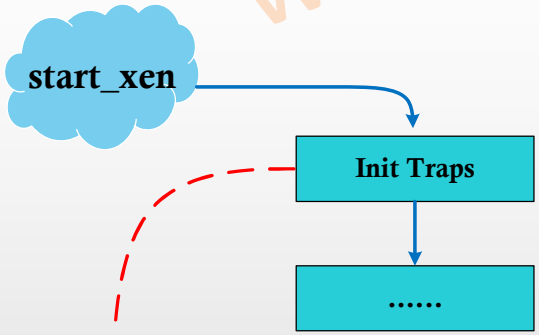
- **setup-fixmap**目前只为**early_console**做了固定映射，用于**debug**。
- **Launch**这边为调用**C**代码做准备，设置好**boot-time stack**我们就可以进入**start_xen**了

```
launch:
        PRINT("- Ready -\r\n")

        ldr   x0, =init_data
        add   x0, x0, #INITINFO_stack /* Find the boot-time stack */
        ldr   x0, [x0]
        add   x0, x0, #STACK_SIZE    /* (which grows down from the top). */
        sub   x0, x0, #CPUINFO_sizeof /* Make room for CPU save record */
        mov   sp, x0

        cbnz  x22, 1f

        mov   x0, x20               /* Marshal args: - phys_offset */
        mov   x1, x21               /*               - FDT */
        b     start_xen             /* and disappear into the land of C */
```

OVERVIEW OF START_XEN



INIT TRAPS




```
void init_traps(void)
{
    /*
     * Setup Hyp vector base. Note they might get updated with the
     * branch predictor hardening.
     */
    WRITE_SYSREG((vaddr_t)hyp_traps_vector, VBAR_EL2);

          ……

    /*
     * Configure HCR_EL2 with the bare minimum to run Xen until a guest
     * is scheduled. {A,I,F}MO bits are set to allow EL2 receiving
     * interrupts.
     */
    WRITE_SYSREG(HCR_AMO | HCR_FMO | HCR_IMO, HCR_EL2);
    isb();
}
```

```
/*
 * Exception vectors.
 */
ENTRY(hyp_traps_vector)
        ventry  hyp_sync_invalid        /* Synchronous EL2t */
        ventry  hyp_irq_invalid         /* IRQ EL2t */
        ventry  hyp_fiq_invalid         /* FIQ EL2t */
        ventry  hyp_error_invalid       /* Error EL2t */

        ventry  hyp_sync                /* Synchronous EL2h */
        ventry  hyp_irq                 /* IRQ EL2h */
        ventry  hyp_fiq_invalid         /* FIQ EL2h */
        ventry  hyp_error               /* Error EL2h */

        ventry  guest_sync              /* Synchronous 64-bit EL0/EL1 */
        ventry  guest_irq               /* IRQ 64-bit EL0/EL1 */
        ventry  guest_fiq_invalid       /* FIQ 64-bit EL0/EL1 */
        ventry  guest_error             /* Error 64-bit EL0/EL1 */

        ventry  guest_sync_compat       /* Synchronous 32-bit EL0/EL1 */
        ventry  guest_irq_compat        /* IRQ 32-bit EL0/EL1 */
        ventry  guest_fiq_invalid_compat /* FIQ 32-bit EL0/EL1 */
        ventry  guest_error_compat      /* Error 32-bit EL0/EL1 */
```

Xen不会使用**EL0**的**Stack**，所以**EL2t**的所有异常都是非法的

Xen的设计是以支持**Linux Guest**为目标的，所以不存在合法的**FIQ**

Xen在一开始使能异常向量表，用于检测和回溯在它初始化过程中遇到的：

发生在**EL2**的同步异常，中断和异步错误

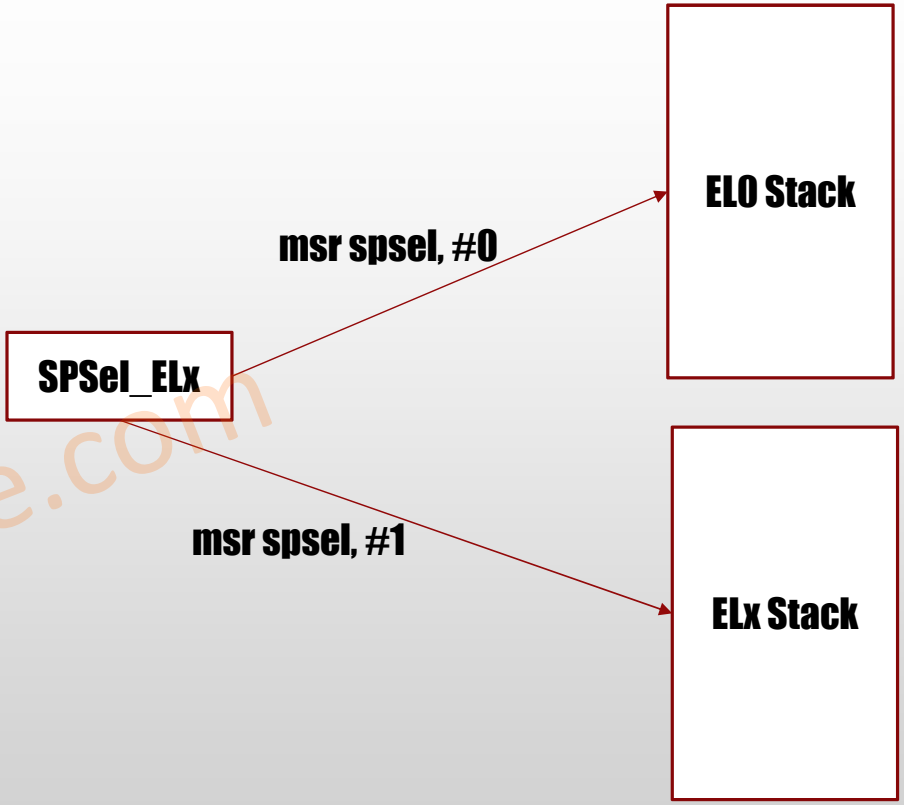
发生在**EL1 AArch64**模式的同步异常，中断和异步错误

发生在**EL1 AArch32**模式的同步异常，中断和异步错误

STACK SELECTOR

SPSel_ELx (x可以是**1**，**2**，**3)**，可以选择**EL1**，**EL2**和**EL3**使用独立的栈还是和**EL0**公用一个堆栈。当**SPSel_ELx**为**1**时表示**ELx**使用独立的栈。
举个例子：
Xen EL2执行**msr spsel, #1**就是选择在**EL2**使用独立的栈。

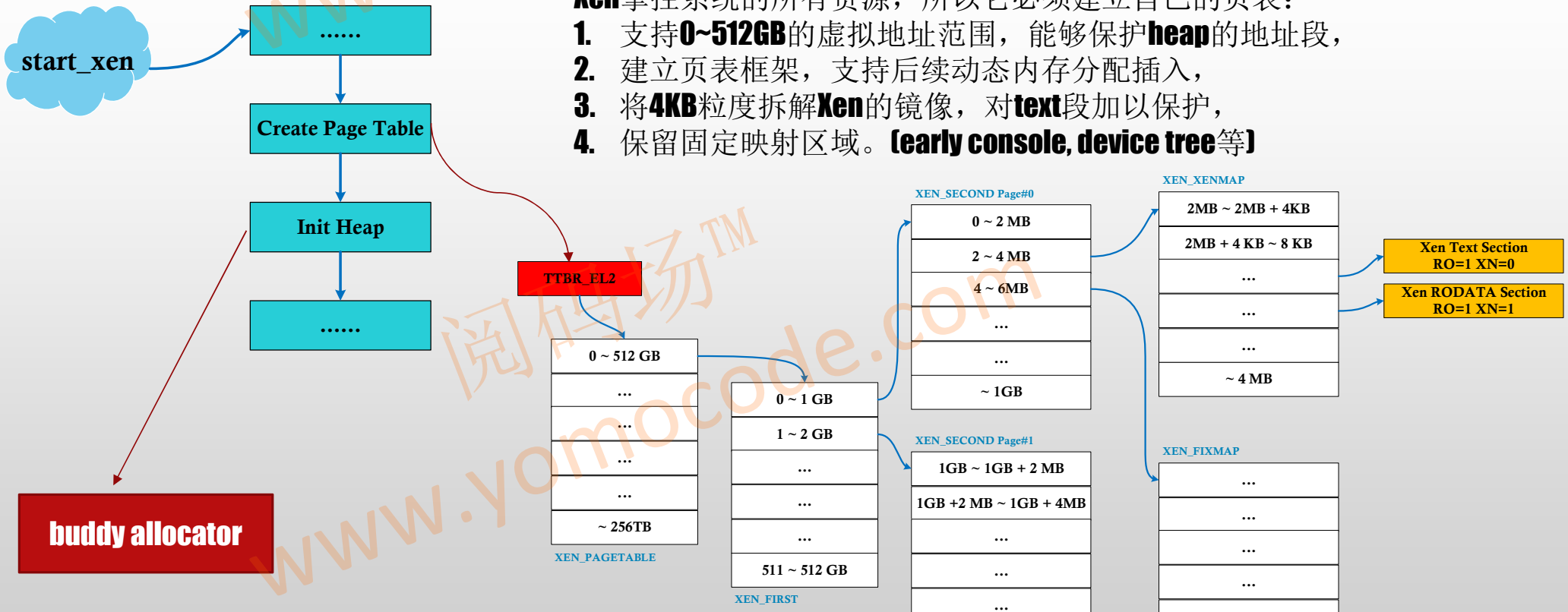
```
/*
 * Ensure that any exceptions encountered at EL2
 * are handled using the EL2 stack pointer, rather
 * than SP_EL0.
 */
msr spsel, #1
```



SETUP PAGE TABLE

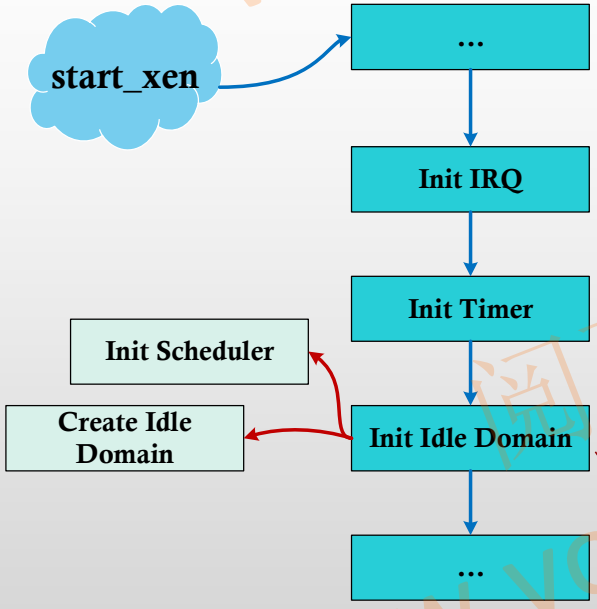
Xen掌控系统的所有资源，所以它必须建立自己的页表：

1. 支持**0~512GB**的虚拟地址范围，能够保护**heap**的地址段，
2. 建立页表框架，支持后续动态内存分配插入，
3. 将**4KB**粒度拆解**Xen**的镜像，对**text**段加以保护，
4. 保留固定映射区域。**(early console, device tree**等**)**



除了被页表、**DTB**、**Xen**镜像占用和使用的内存，和其它一些保留内存。剩下的所有内存都被**Xen**映射到**XENHEAP_VIRT_START**开始的堆。由内存分配器进行管理。**(**划分给虚拟机内存也从堆分配**)**

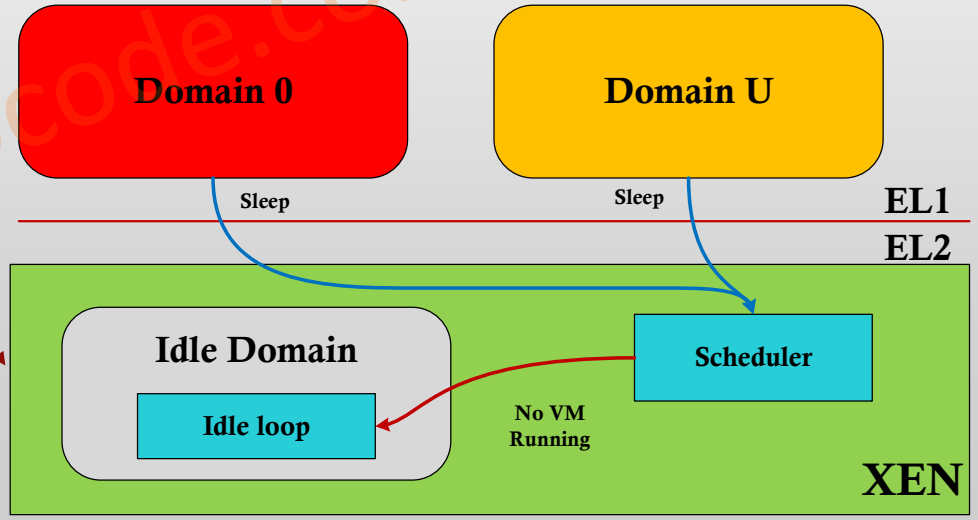
INIT SCHEDULER



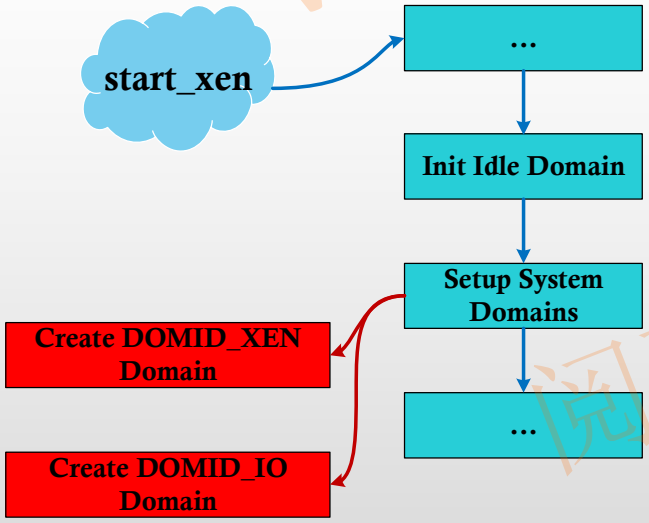
Xen内置一个调度器，负责虚拟机的调度。：

1. 初始化中断控制器，
2. 初始化时钟中断，
3. 初始化调度器，并创建**Idle Domain**

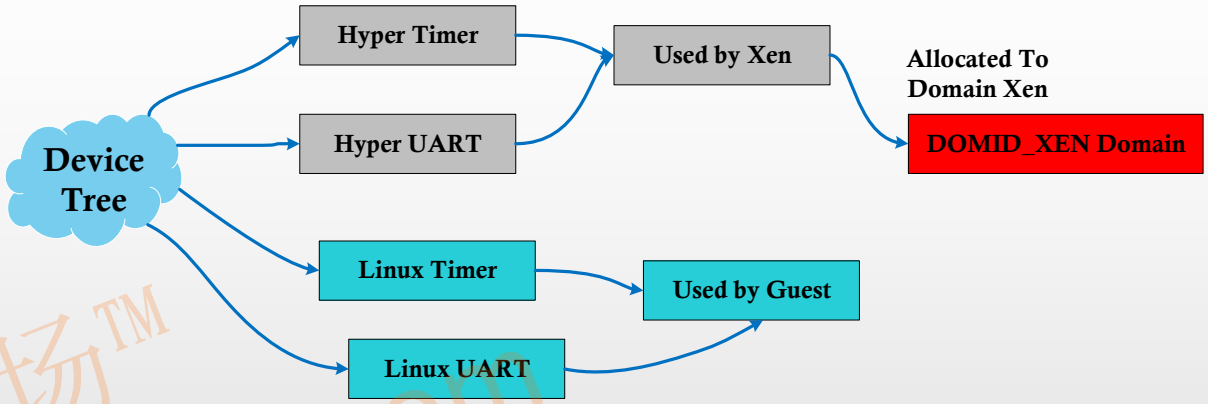
Xen目前支持**Credit, Credit2, RT**以及**ARINC653**三种调度算法，其中**ARINC653**为最新的实时调度算法。



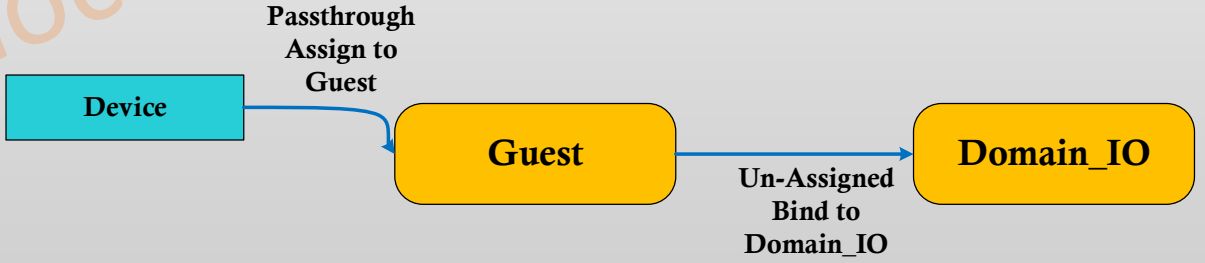
DOMAIN_XEN AND DOMAIN_IO



- **DOMAIN_XEN**用于绑定**Xen**使用的资源，比如串口，**hyper timer**等
- **DOMAIN_IO**用于绑定专门给**IO**使用的资源

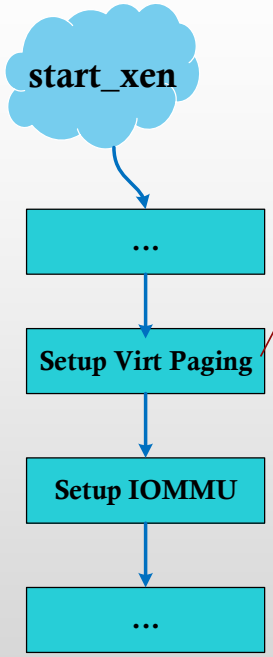


从**device tree**扫描设备，属于**Xen**的设备会绑定到**DOMAIN_XEN**然后从**Device tree**中屏蔽



将**Passthrough**到**Guest**设备进行回收，这个回收的设备就会绑定到**DOMAIN_IO.**

DETECT VIRTUAL MACHINE FEATURES




```
void __init setup_virt_paging(void)
{
  /* Setup Stage 2 address translation */
  ……
  {
    /* T0SZ minimum and SL0 maximum from ARM DDI 0487A.b Table D4-5 */
    /*  PA size, t0sz(min), root-order, sl0(max) */
    [0] = { 32,     32/*32*/,  0,        1 },
    [1] = { 36,     28/*28*/,  0,        1 },
    [2] = { 40,     24/*24*/,  1,        1 },
    [3] = { 42,     22/*22*/,  3,        1 },
    [4] = { 44,     20/*20*/,  0,        2 },
    [5] = { 48,     16/*16*/,  0,        2 },
  };
  ……
  ……
  val |= VTCR_TG0_4K;

  /* Set the VS bit only if 16 bit VMID is supported. */
  if ( MAX_VMID == MAX_VMID_16_BIT )
    val |= VTCR_VS;
  ……
  val |= VTCR_SL0(pa_range_info[pa_range].sl0);
  ……
  val |= VTCR_T0SZ(pa_range_info[pa_range].t0sz);

  ……
}
```

➢ 检测**hypervisor**的内存虚拟化能力：

1. 虚拟机支持多大的物理地址**(IPA)**范围

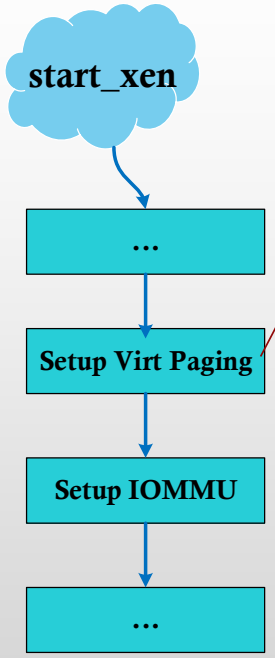
2. 虚拟机支持的**stage2**页表翻译粒度是**4KB**

3. 虚拟机支持多少个**VMID (8-bit or 16-bit)**

4. Stage 2页表的翻译级数最少可以是多少级**(4**级，**3**级？**)**

5. Stage 2实际需要负责翻译的虚拟机物理地址范围大小

DETECT VIRTUAL MACHINE FEATURES



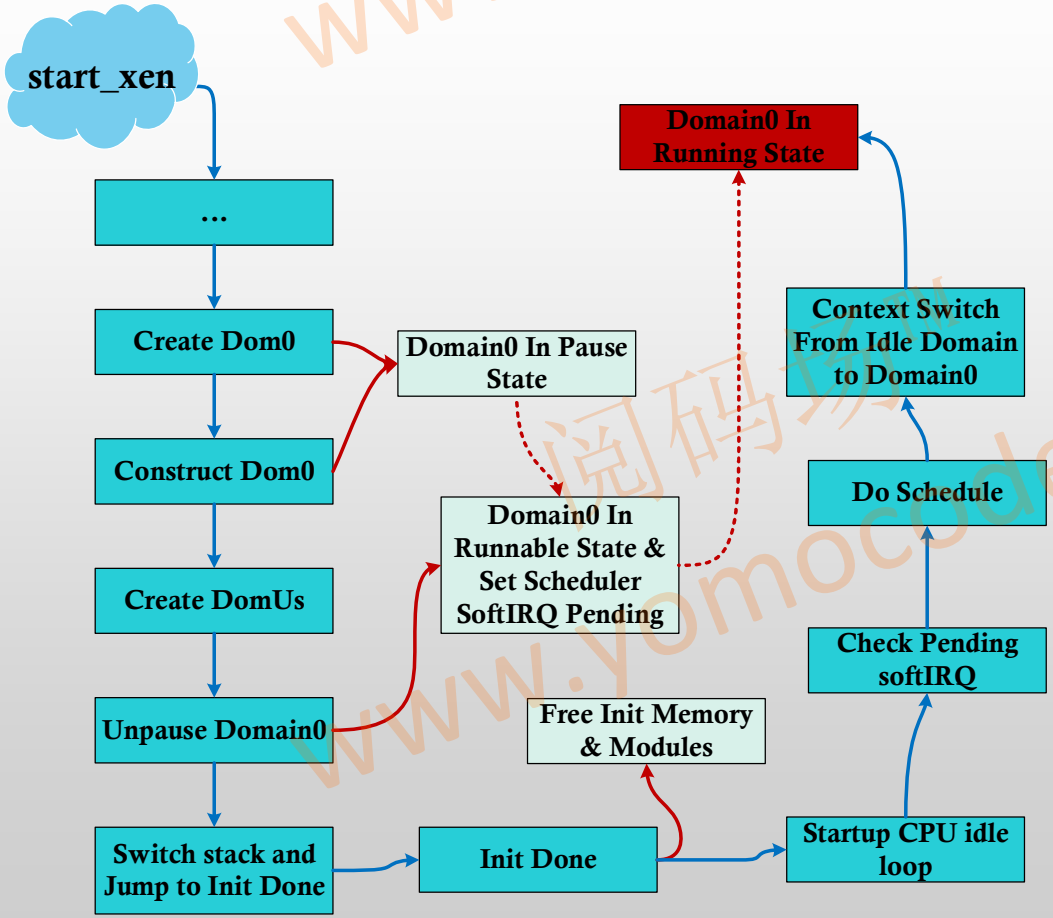

```
int __init iommu_setup(void)
{
    ……
    rc = iommu_hardware_setup();
    iommu_enabled = (rc == 0)

    ……

    if ( !iommu_intremap )
        iommu_intpost = 0;

    printk("I/O virtualisation %sabled\n", iommu_enabled ? "en" : "dis");
    ……
    {
        printk(" - Dom0 mode: %s\n",
               iommu_hwdom_passthrough ? "Passthrough" :
               iommu_hwdom_strict ? "Strict" : "Relaxed");
        printk("Interrupt remapping %sabled\n", iommu_intremap ? "en" : "dis");
    }
    ……
}
```

- 检测虚**hypervisor**的**I/O**虚拟化能力：
1. 是否支持**IOMMU**进行**DMA**内存地址转换。
2. **IOMMU**是否支持**interrupt remapping**

CREATE AND ENTER GUEST

创建和运行虚拟机

CREATE DOMAIN#0



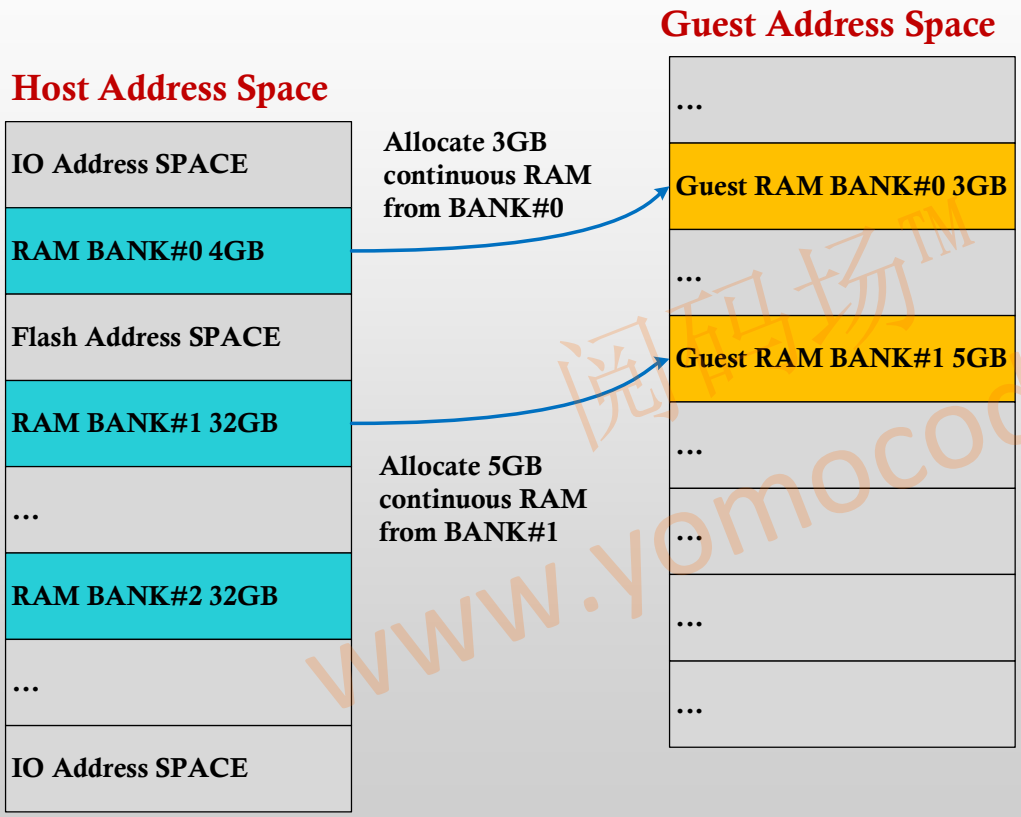
Domain 0是**Xen**功能的一个延伸，没有**Domain 0 Xen**无法正常工作。所以**Domain 0**是伴随着**Xen**的启动而启动的。只有**Domain 0**正常启动完成，**Xen**才算真正的初始化完成。

分配给**Domain 0**的内存和外设地址和实际的物理地址是**1:1**映射的**(PA = IPA)**。这是因为：

1. **Domain 0**本身可以被看做**Xen**的一部分，没有必要对它隐藏太多的细节。
2. **Domain 0**本身要负担各种硬件的驱动，这些驱动的信息都是描述在**DTB**或者**ACPI table**里。如果**Domain 0**内的地址空间做了重映射，那么就要按照**Domain 0**的地址空间重新生成一份**DTB**或者**ACPI table**。而不能像现在这样直接从主机的**DTB**或者**ACPI table**做少量的修改得到。

****allocate_memory11()**
****insert_11_bank()**

1:1 MAPPING MEMORY FOR DOMAIN#0

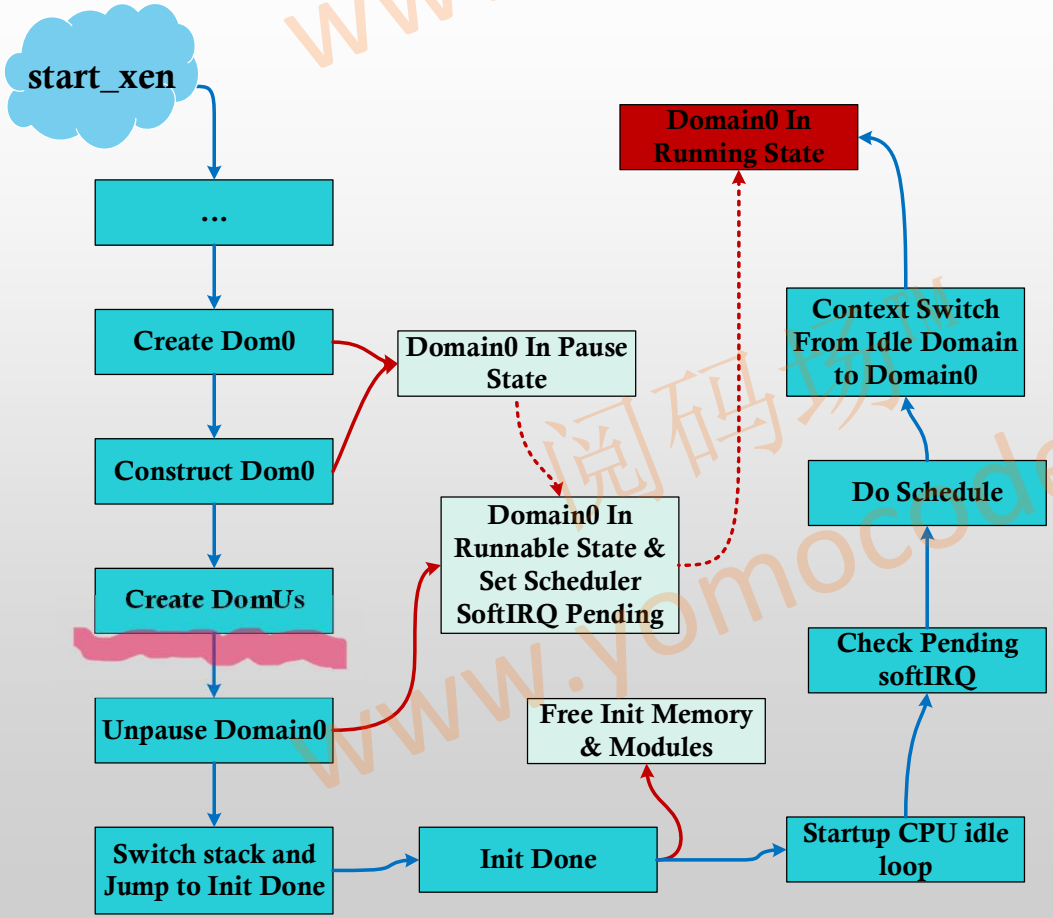


在保持**IPA=PA**的前提下，尝试从**Bank#0**开始分配连续的物理内存给**Domain#0**：

1. **Bank#0**的地址范围可能是**0~4GB**，可以兼容**32-bit no-LPAE Domain-0**系统和**32-bit DMA**设备。
2. **Linux** 希望讲内核镜像、**device tree**和**RAMDISK**在启动时都加载地址最低的同一块**bank**中。
3. **Linux zImage**启动方式要求内核被加载到内存的第一个**128MB**区间
4. 分配连续的内存可以让**Domain0**拥有少量大块的**bank**，而不是大量碎片化的**bank.**

如果只支持64-bit**的**Domain 0**，从**bank#0**开始分配内存就不是必须的。

CREATE DOMAIN#U

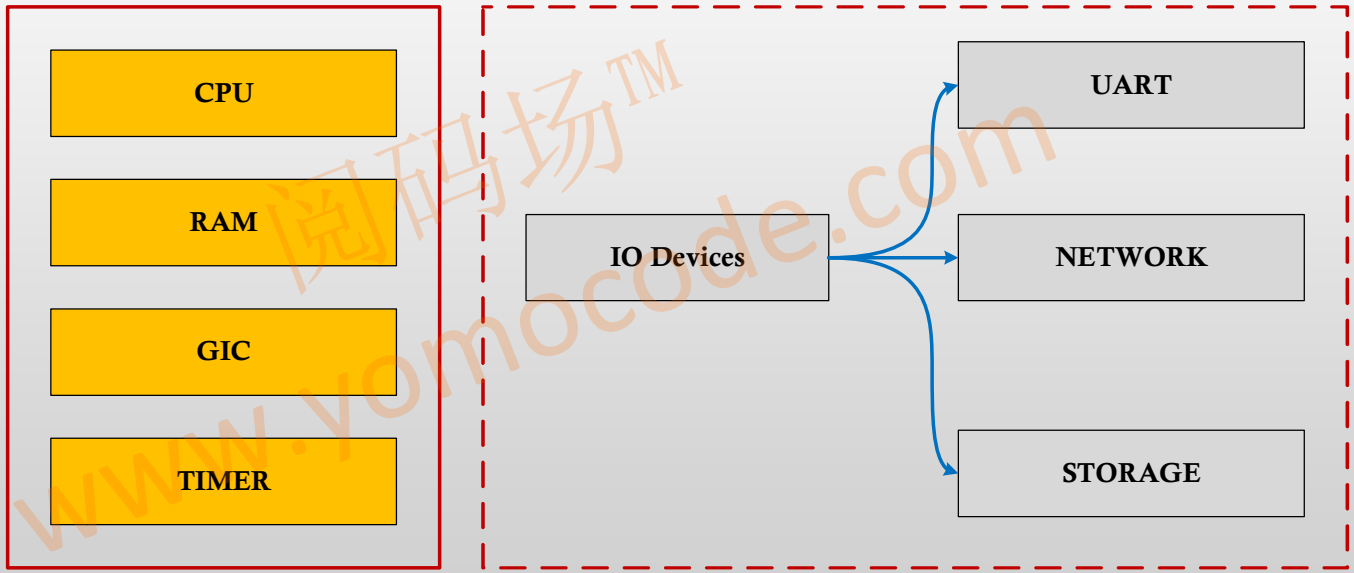


Create DomUs:
如果**Xen** 在启动时，除了**Domain 0**之外还指定了其它的**Guest**作为**Domain U**伴随启动。**Create DomUs**就会创建这些**Guest**. 可以不依赖于**Domain 0**里的工具链启动

HOW TO CREATE A VIRTUAL MACHINE

一个**ARM**虚拟机的构成:

1. 必要部分：支撑一个只有简单**idle task**的操作系统运行的最少组件
2. 可选部分：支撑一个支持网络和文件系统的操作系统需要的组件

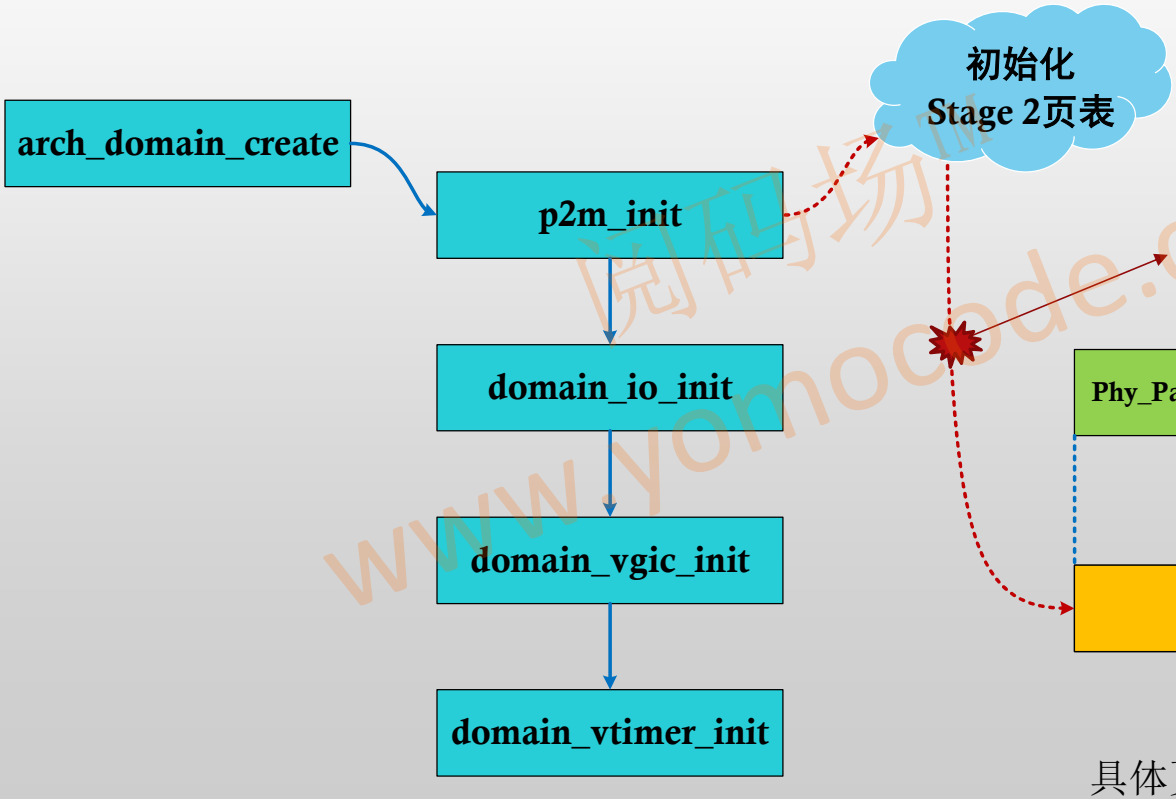


HOW TO CREATE A VIRTUAL MACHINE

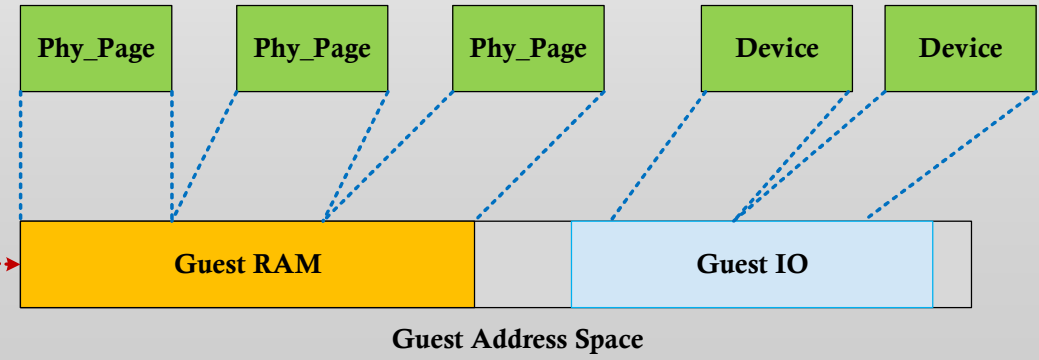
ARM虚拟机的创建分两部分：

1. arch_create_domain: 构建一个虚拟**SoC**，除了**CPU**(含紧耦合在**CPU**内部的**GICC, GICR**以及**ARCH_TIMER**)

2. arch_vcpu_create: 构建虚拟**CPU**，包含用于保存通用寄存器，浮点寄存器，**EL1**，**ELO**可以被修改的系统寄存器以及**GICC**，**GICR**和**ARCH_TIMER.**

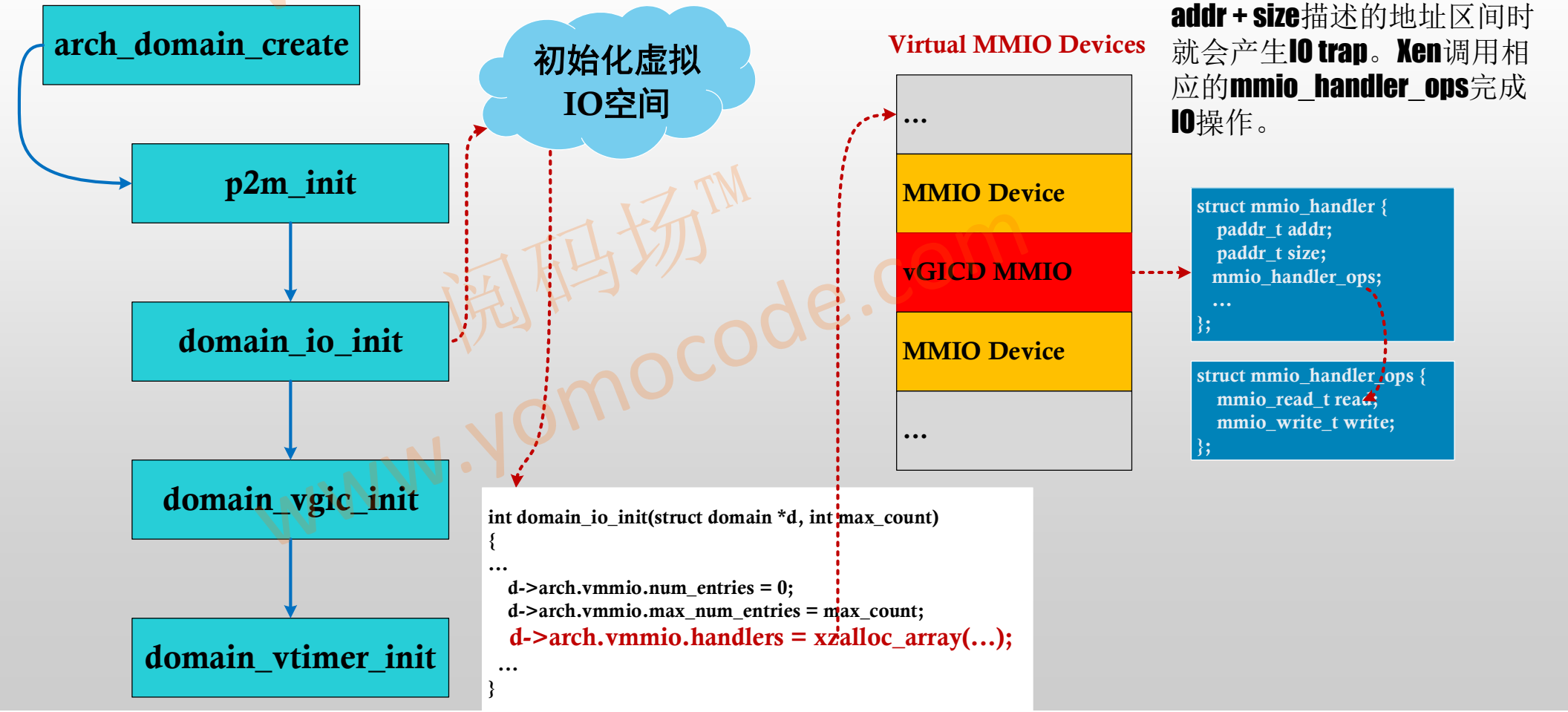


注：这个阶段并没有插入实际页面



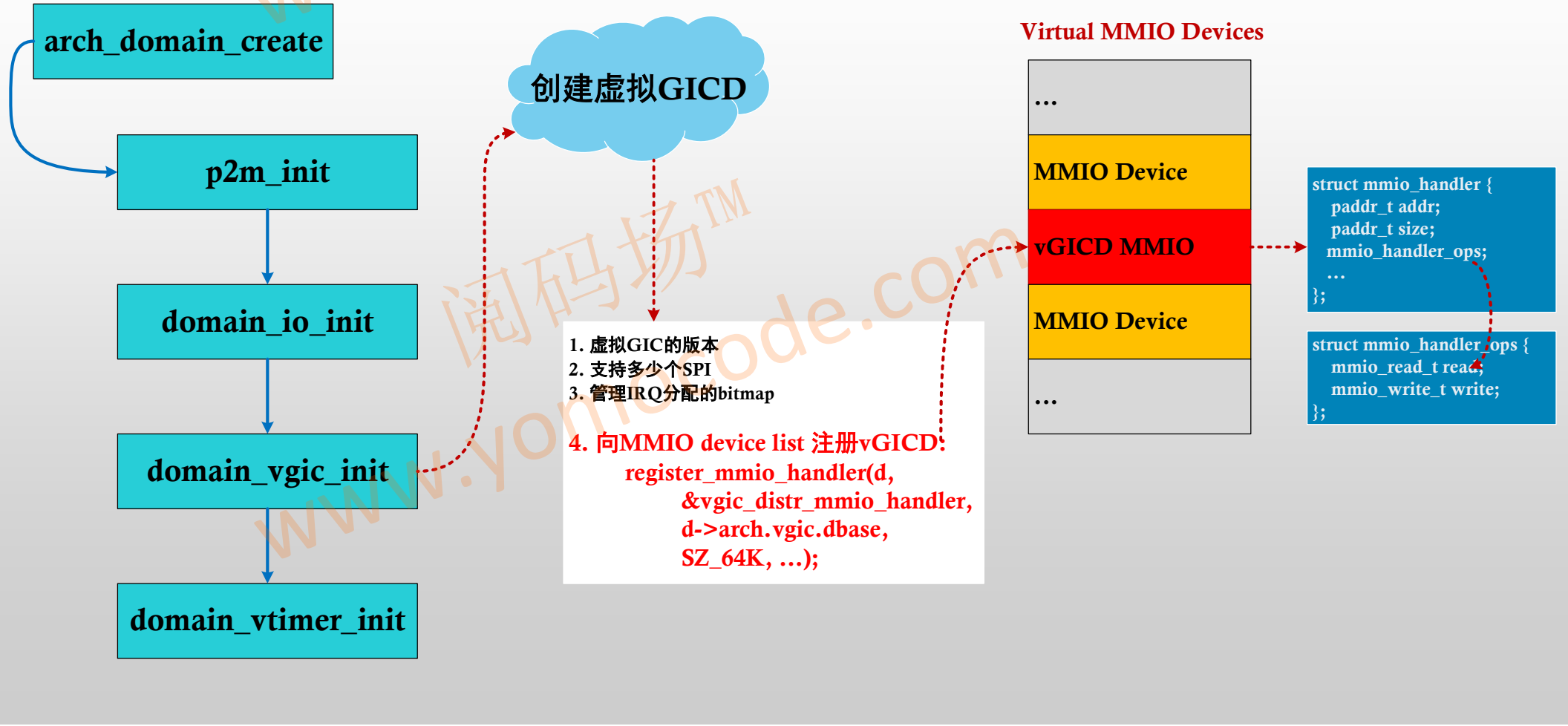
具体页面插入可参考：**p2m.c guest_physmap_add_entry**

HOW TO CREATE A VIRTUAL MACHINE

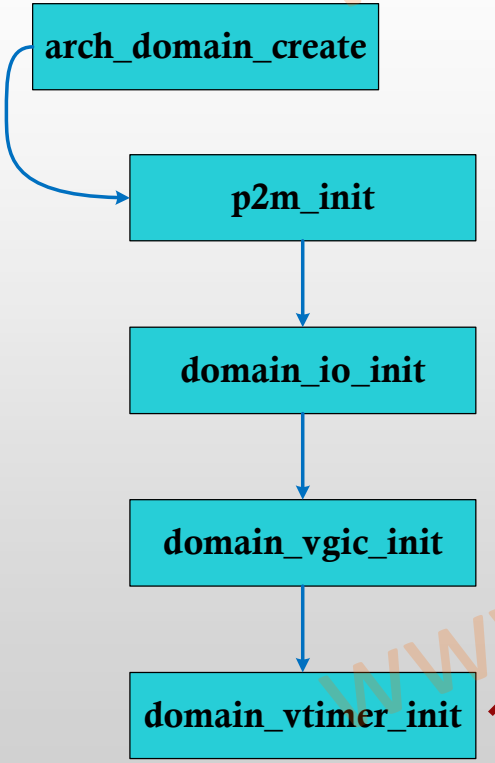


虚拟机访问到**MMIO addr**到**addr + size**描述的地址区间时就会产生**IO trap**。**Xen**调用相应的**mmio_handler_ops**完成**IO**操作。

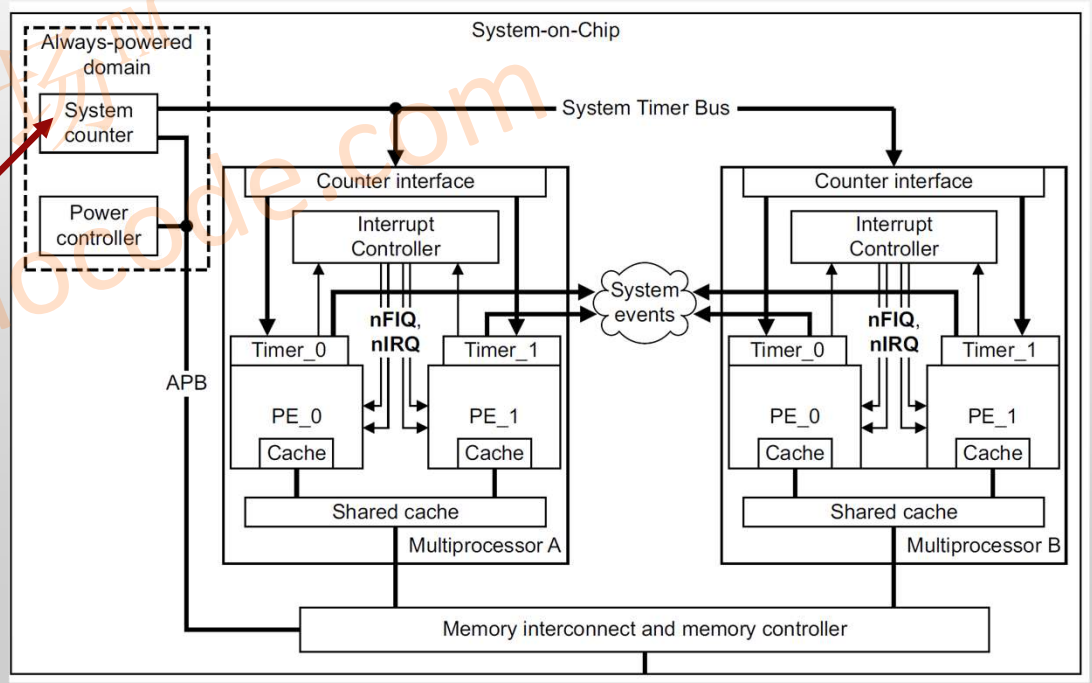
HOW TO CREATE A VIRTUAL MACHINE



HOW TO CREATE A VIRTUAL MACHINE

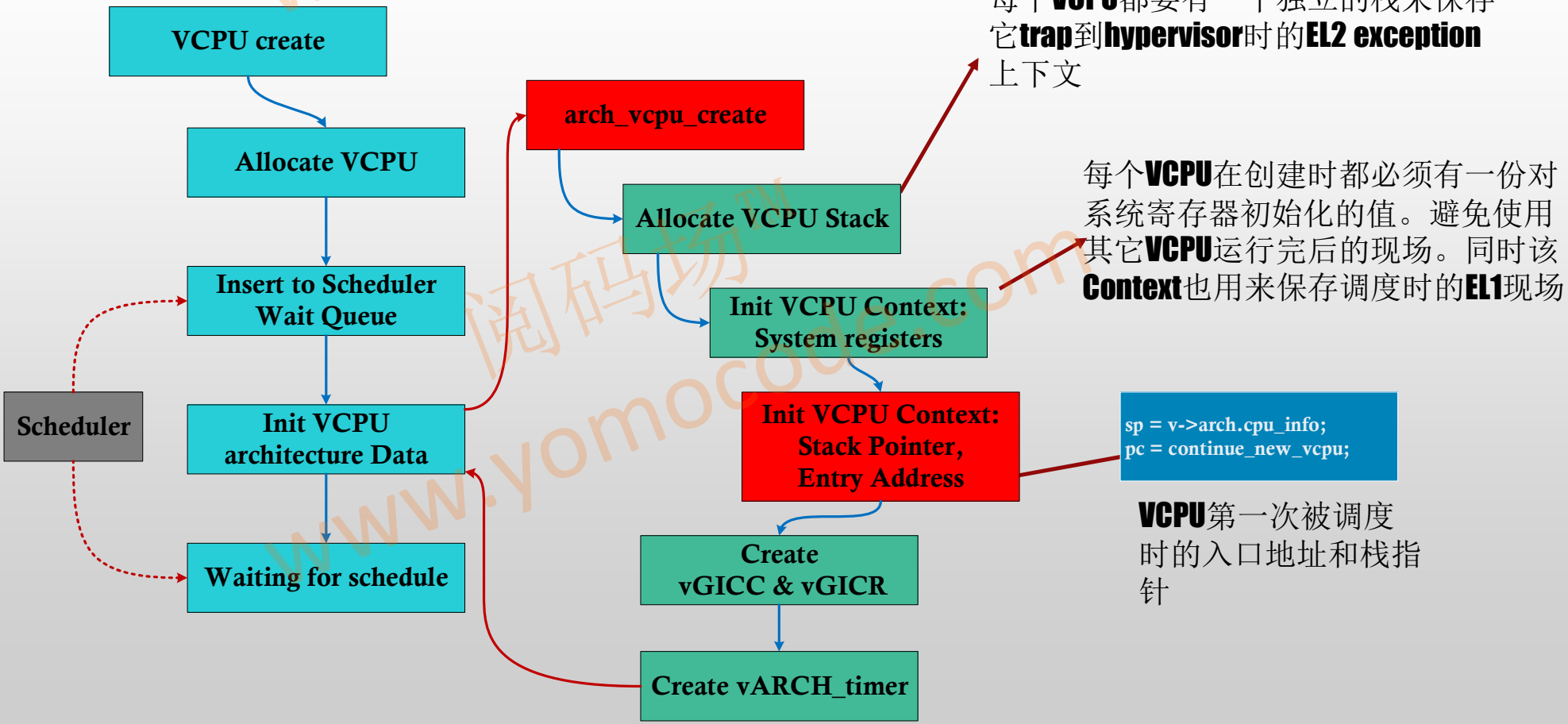


Arch timer属于**CPU**的范畴，**vtimer**也属**VCPU**的范畴。但是和所有**arch_timer**都源自于**sys_timer**一样。**Domain vtimer**也是类似的角色。**Domain_vtimer_init**保存**timer**的频率，和物理**timer**的偏移以及请求**vGIC**保留**vtimer**相关中断等工作。



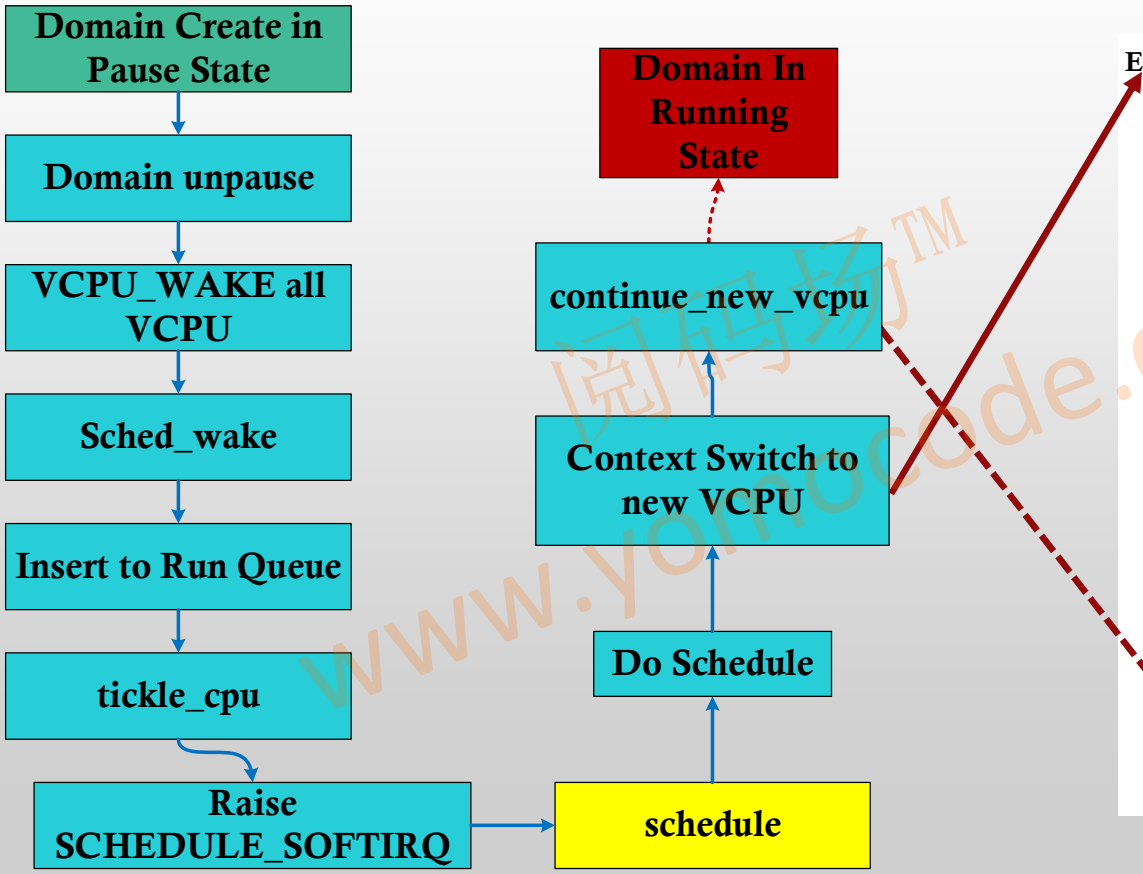
HOW TO CREATE A VIRTUAL MACHINE

vcpu_create: VCPU是Hypervisor的调度单元



VM RUN ON XEN

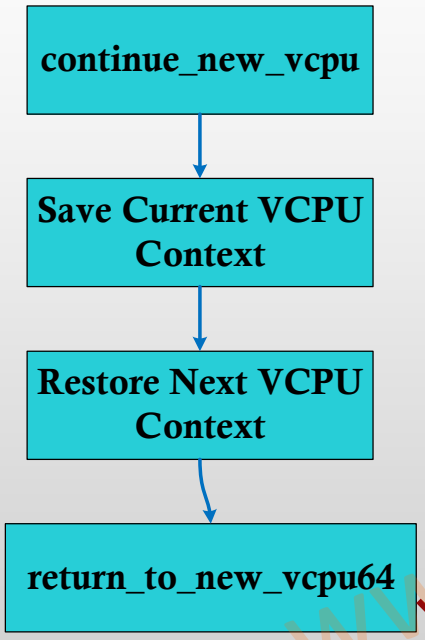
虚拟机的运行单元时**VCPU**，**VCPU**的执行是由**Xen**的调度器决定的。




```
ENTRY(__context_switch)
    add     x8, x0, #VCPU_arch_saved_context
    mov     x9, sp
    /* store callee-saved registers */
    stp     x19, x20, [x8], #16
    stp     x21, x22, [x8], #16
    stp     x23, x24, [x8], #16
    stp     x25, x26, [x8], #16
    stp     x27, x28, [x8], #16
    stp     x29, x9, [x8], #16
    str     lr, [x8]

    add     x8, x1, #VCPU_arch_saved_context
    /* restore callee-saved registers */
    ldp     x19, x20, [x8], #16
    ldp     x21, x22, [x8], #16
    ldp     x23, x24, [x8], #16
    ldp     x25, x26, [x8], #16
    ldp     x27, x28, [x8], #16
    ldp     x29, x9, [x8], #16
    ldr     lr, [x8]
    mov     sp, x9
    ret
```

LR指向:
continue_new_vcpu

GUEST RUN ON XEN




```
.macro  exit_guest

        add     x21, sp, #UREGS_SPSR_el1
        ldr     x23, [x21]
        msr     SPSR_el1, x23
        ......
        add     x21, sp, #UREGS_SP_el0
        ldr     x22, [x21]
        msr     SP_el0, x22

        add     x21, sp, #UREGS_SP_el1
        ldp     x22, x23, [x21]
        msr     SP_el1, x22
        msr     ELR_el1, x23
        ......
.endm

.macro  exit, hyp, compat
        bl      leave_hypervisor_tail
        exit_guest
        b       return_from_trap
 .endm

ENTRY(return_to_new_vcpu64)
        exit    hyp=0, compat=0
```

```
return_from_trap:
    msr     daifset, #2 /* Mask interrupts */

    ldr     x21, [sp, #UREGS_PC]            /* load ELR */
    ldr     w22, [sp, #UREGS_CPSR]          /* load SPSR */

    pop     x0, x1
    ......
    pop     x8, x9

    msr     elr_el2, x21 /* set up the return data */
    msr     spsr_el2, x22

    pop     x10, x11
    ......
    pop     x28, x29

    ldr     lr, [sp], #(UREGS_SPSR_el1 - UREGS_LR)

    eret
```

VCPU EL1入口地址

VCPU进入**EL1**，从**ELR_EL2**指定地址开始运行

#define PSR_GUEST64_INIT (PSR_ABT_MASK|PSR_FIQ_MASK|PSR_IRQ_MASK |PSR_MODE_EL1h) → SPSR_EL2决定**CPU**返回的模式

EXCEPTION TRAPS

异常处理

EXCEPTIONS

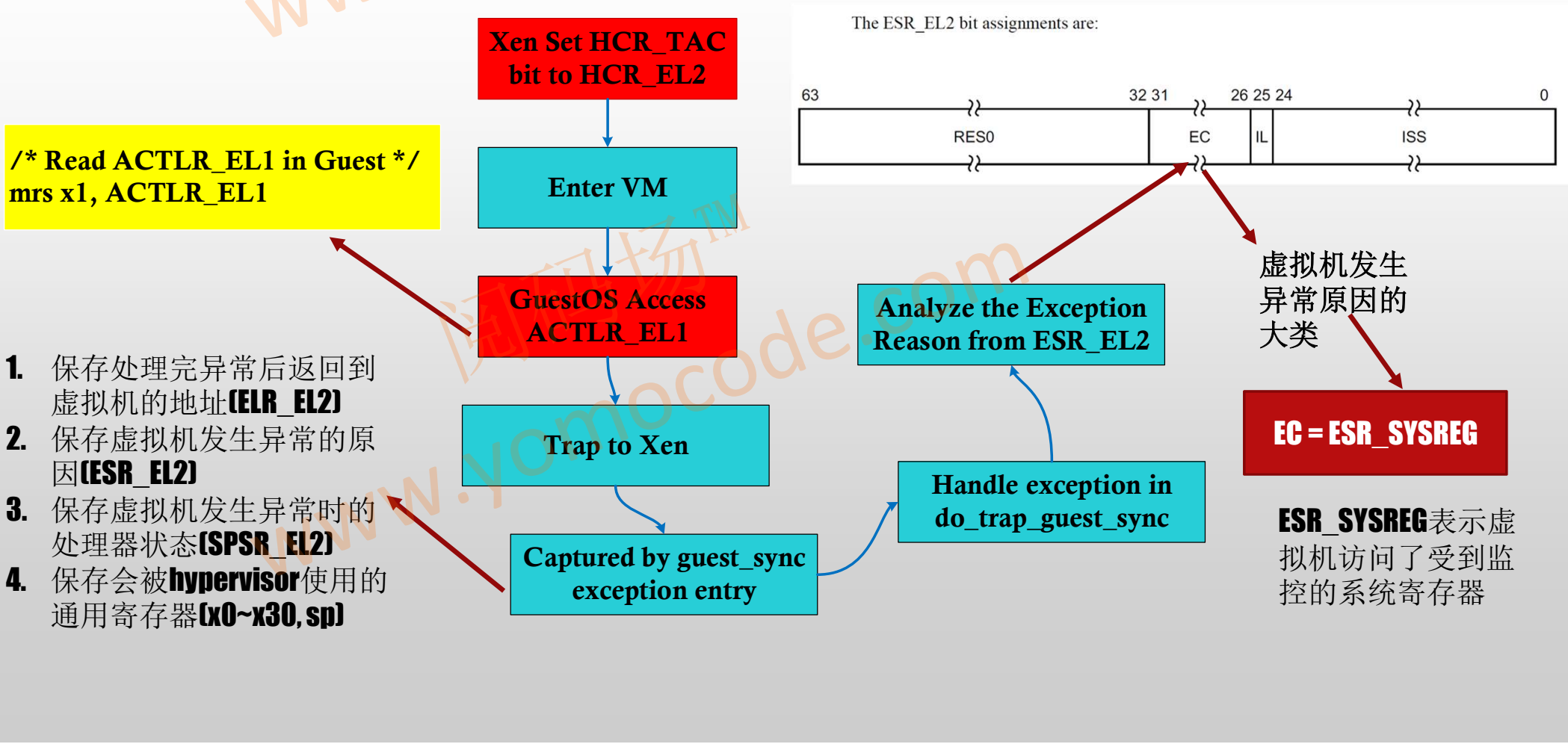
```
ENTRY(hyp_traps_vector)
    ventry  hyp_sync_invalid    /* Synchronous EL2t */
    ventry  hyp_irq_invalid     /* IRQ EL2t */
    ventry  hyp_fiq_invalid     /* FIQ EL2t */
    ventry  hyp_error_invalid   /* Error EL2t */

    ventry  hyp_sync            /* Synchronous EL2h */
    ventry  hyp_irq             /* IRQ EL2h */
    ventry  hyp_fiq_invalid     /* FIQ EL2h */
    ventry  hyp_error           /* Error EL2h */

    ventry  guest_sync          /* Synchronous 64-bit EL0/EL1 */
    ventry  guest_irq           /* IRQ 64-bit EL0/EL1 */
    ventry  guest_fiq_invalid   /* FIQ 64-bit EL0/EL1 */
    ventry  guest_error         /* Error 64-bit EL0/EL1 */

    ventry  guest_sync_compat       /* Synchronous 32-bit EL0/EL1 */
    ventry  guest_irq_compat        /* IRQ 32-bit EL0/EL1 */
    ventry  guest_fiq_invalid_compat    /* FIQ 32-bit EL0/EL1 */
    ventry  guest_error_compat      /* Error 32-bit EL0/EL1 */
```

捕获虚拟机运行时发生的异常，并不是所有虚拟机的异常都会**trap**到**EL2**，**Hypervisor**通过配置**HCR_EL2**寄存器来选择需要捕获的异常：

```
register_t get_default_hcr_flags(void)
{
    return  (HCR_PTW|HCR_BSU_INNER|
                HCR_AMO|HCR_IMO|HCR_FMO|HCR_VM|
                (vwfi != NATIVE ? (HCR_TWI|HCR_TWE) : 0) |
                HCR_TSC|HCR_TAC|HCR_SWIO|
                HCR_TIDCP|HCR_FB|HCR_TSW);
}
```

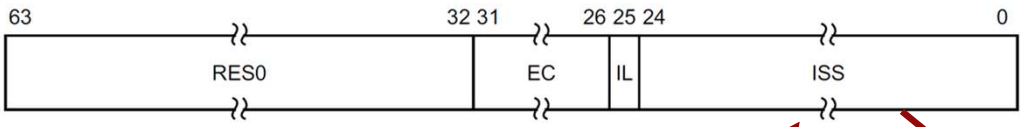
- 特权寄存器访问(**ACTLR_EL1**)
- 特权指令访问(**WFI,WFE,SMC**)
- 反正在虚拟机的**IRQ, FIQ**及**SERROR**
- 二级地址翻译异常
- **IO** 访问异常

PRIVILEGE REGISTERS ACCESS TRAP



PRIVILEGE REGISTERS ACCESS TRAP

The ESR_EL2 bit assignments are:



Xen Set HCR_TAC bit to HCR_EL2

Enter VM

GuestOS Access ACTLR_EL1

Trap to Xen

Captured by guest_sync exception entry

Handle exception in do_trap_guest_sync

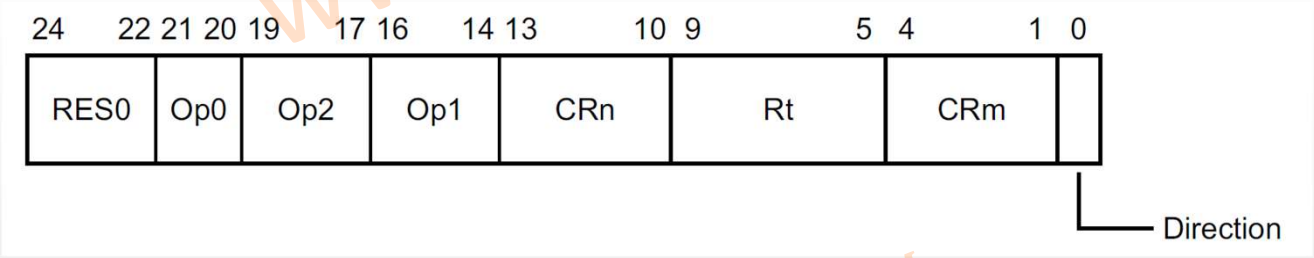
Analyze the Exception Reason from ESR_EL2

EC = ESR_SYSREG

Emulate system registers access in do_sysreg

分析**ESR_SYSREG**的具体原因：
哪个寄存器，
读写操作，
传递结果/参数的寄存器

PRIVILEGE REGISTERS ACCESS TRAP



- **Op0, OP2, OP1, CRn**和**CRm**是寄存器编码：
  HSR_SYSREG(3, 0, c1, c0, 1)代表 **ACTLR_EL1**
- **Direction**代表是读操作还是写操作
- **Rt**表示这次访问中采用哪个通用寄存器保存结果或者传递参数

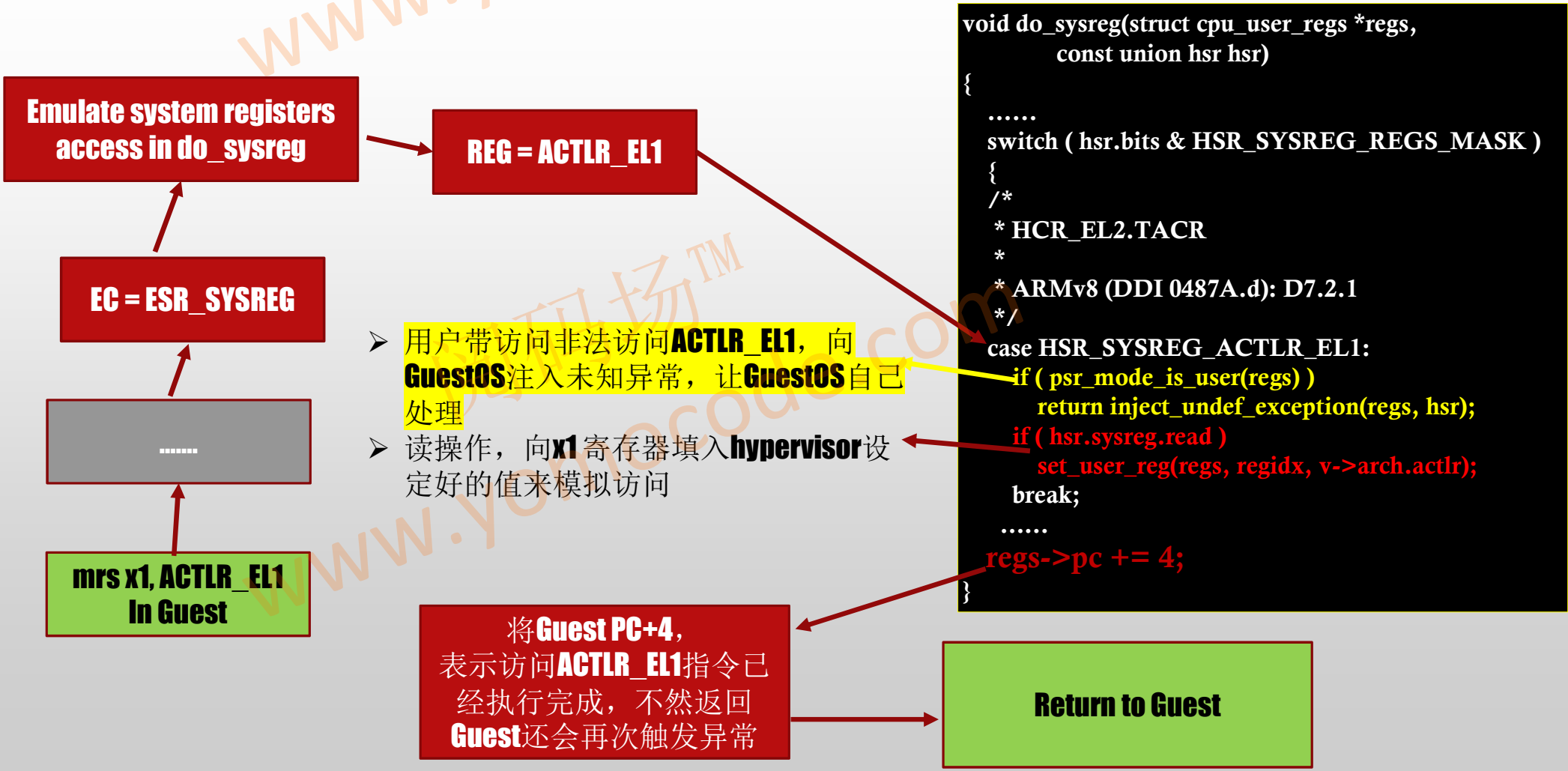
```
#define HSR_SYSREG_SCTLR_EL1      HSR_SYSREG(3,0,c1, c0,0)
#define HSR_SYSREG_ACTLR_EL1      HSR_SYSREG(3,0,c1, c0,1)
#define HSR_SYSREG_TTBR0_EL1      HSR_SYSREG(3,0,c2, c0,0)
#define HSR_SYSREG_TTBR1_EL1      HSR_SYSREG(3,0,c2, c0,1)
#define HSR_SYSREG_TCR_EL1        HSR_SYSREG(3,0,c2, c0,2)
#define HSR_SYSREG_AFSR0_EL1      HSR_SYSREG(3,0,c5, c1,0)
```

例如：

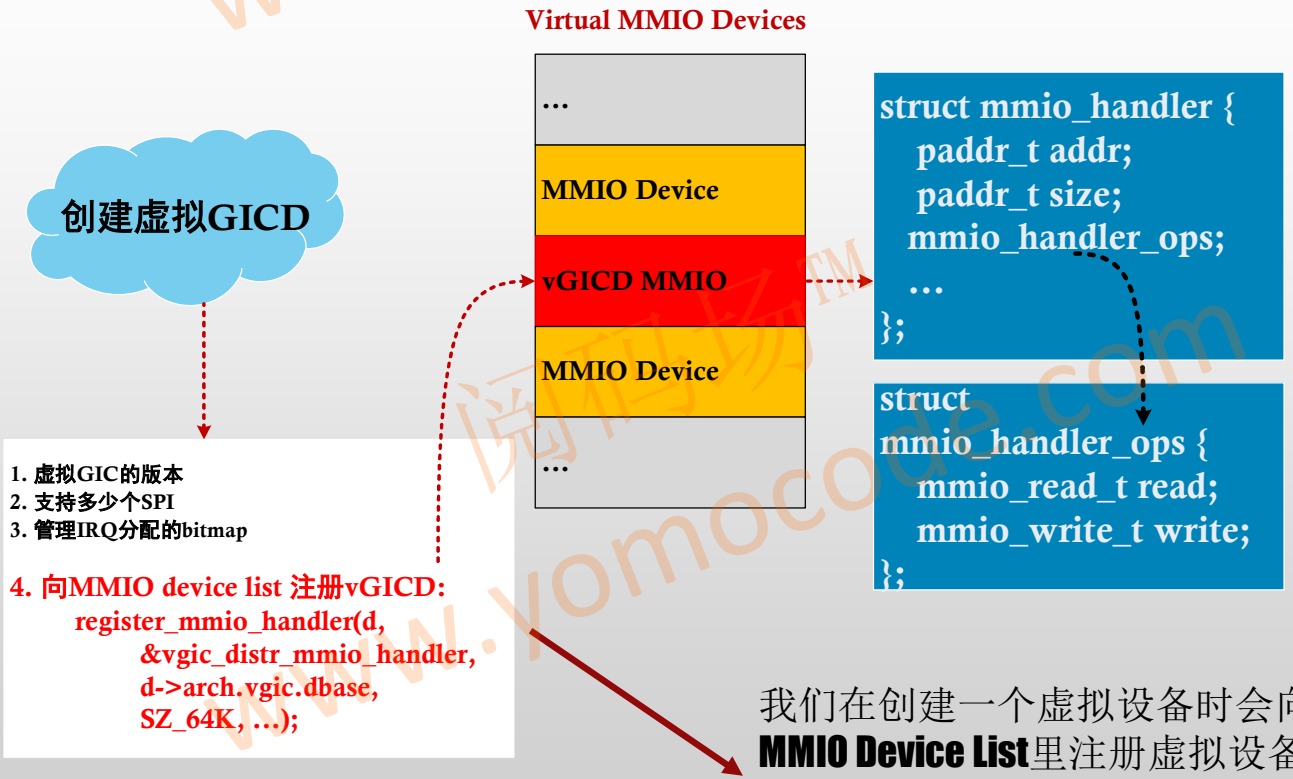
MRS x7 ACTLR_EL1 → 读取ACTLR_EL1: Direction = 1, Rt = 7

MSR x8 ACTLR_EL1 → 写入ACTLR_EL1: Direction = 0, Rt = 8

PRIVILEGE REGISTERS ACCESS TRAP

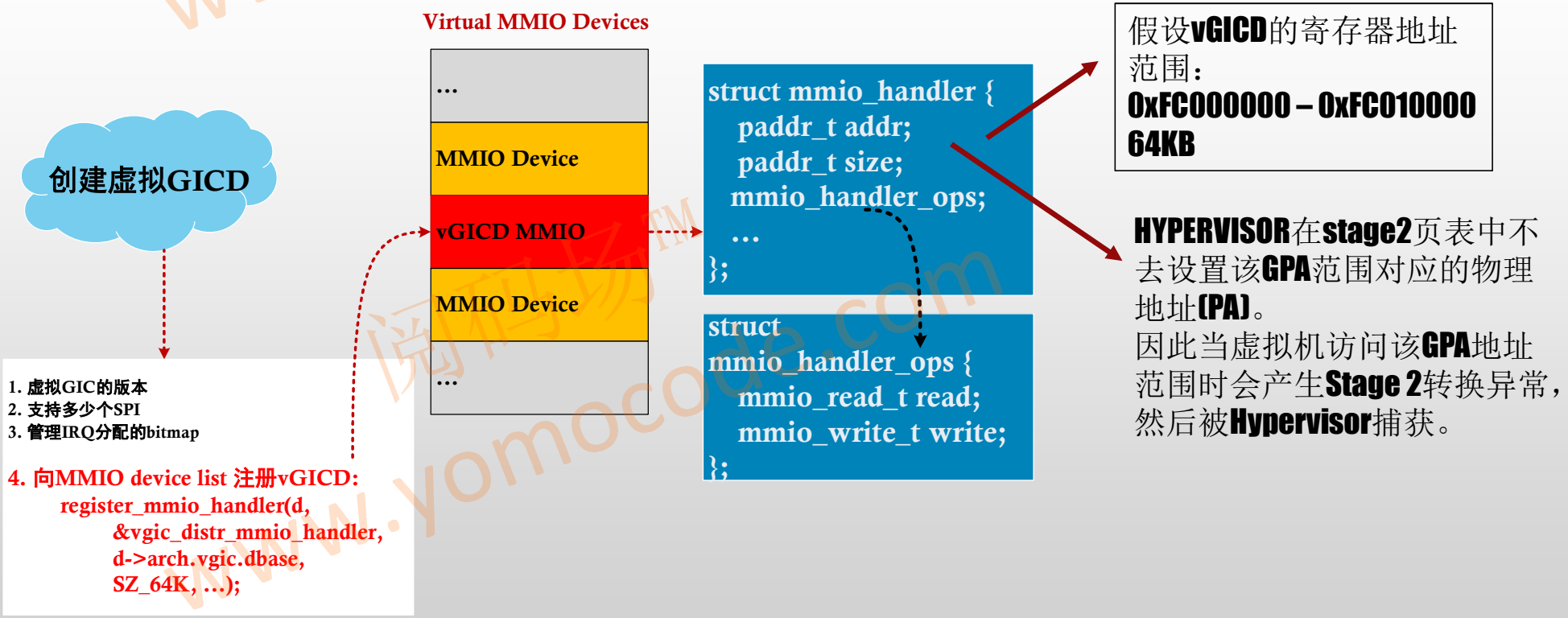


IO TRAP

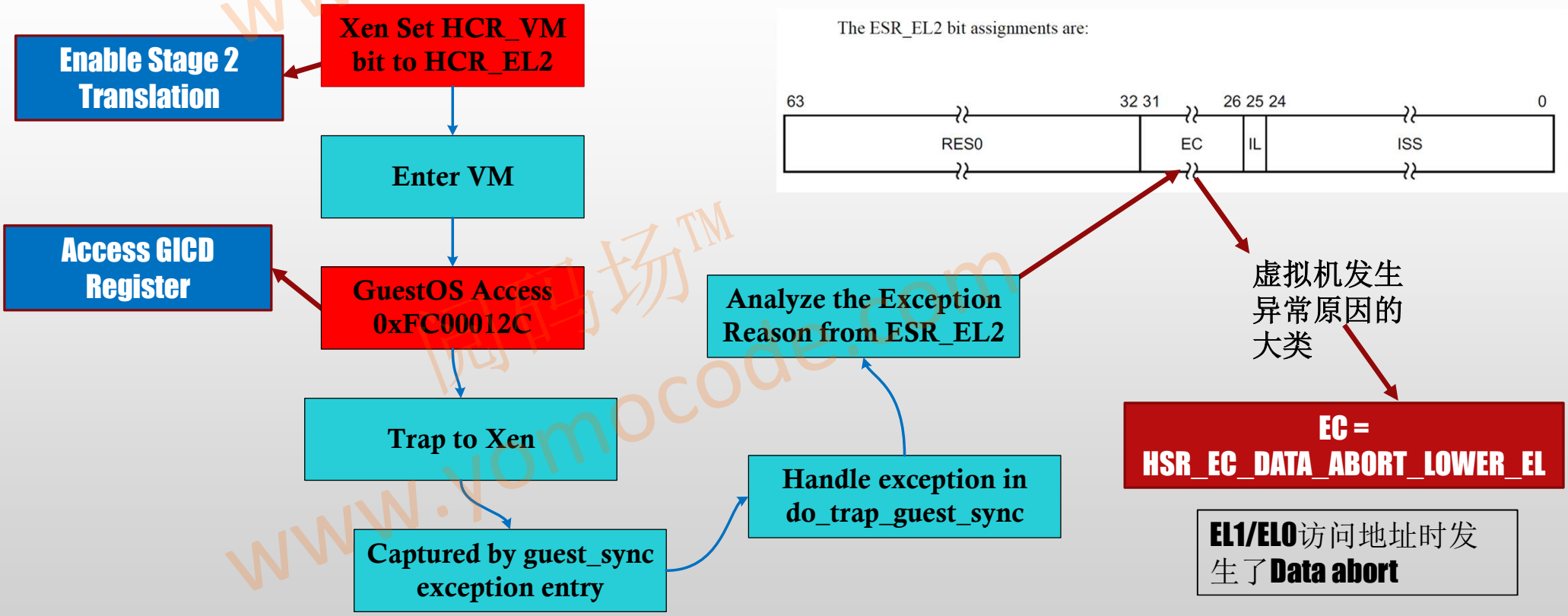


我们在创建一个虚拟设备时会向**Virtual MMIO Device List**里注册虚拟设备，包含该虚拟设备的寄存器地址范围，读写操作对应的回调函数。

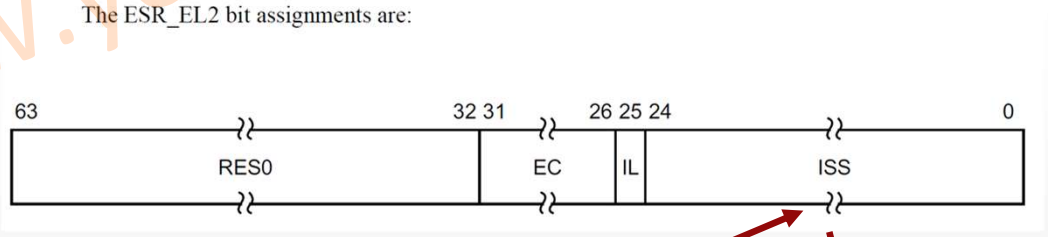
IO TRAP



IO TRAP



IO TRAP



Xen Set HCR_VM bit to HCR_EL2

Enter VM

GuestOS Access 0xFC00012C

Trap to Xen

Captured by guest_sync exception entry

Handle exception in do_trap_guest_sync

Analyze the Exception Reason from ESR_EL2

EC = HSR_EC_DATA_ABORT_LOWER_EL

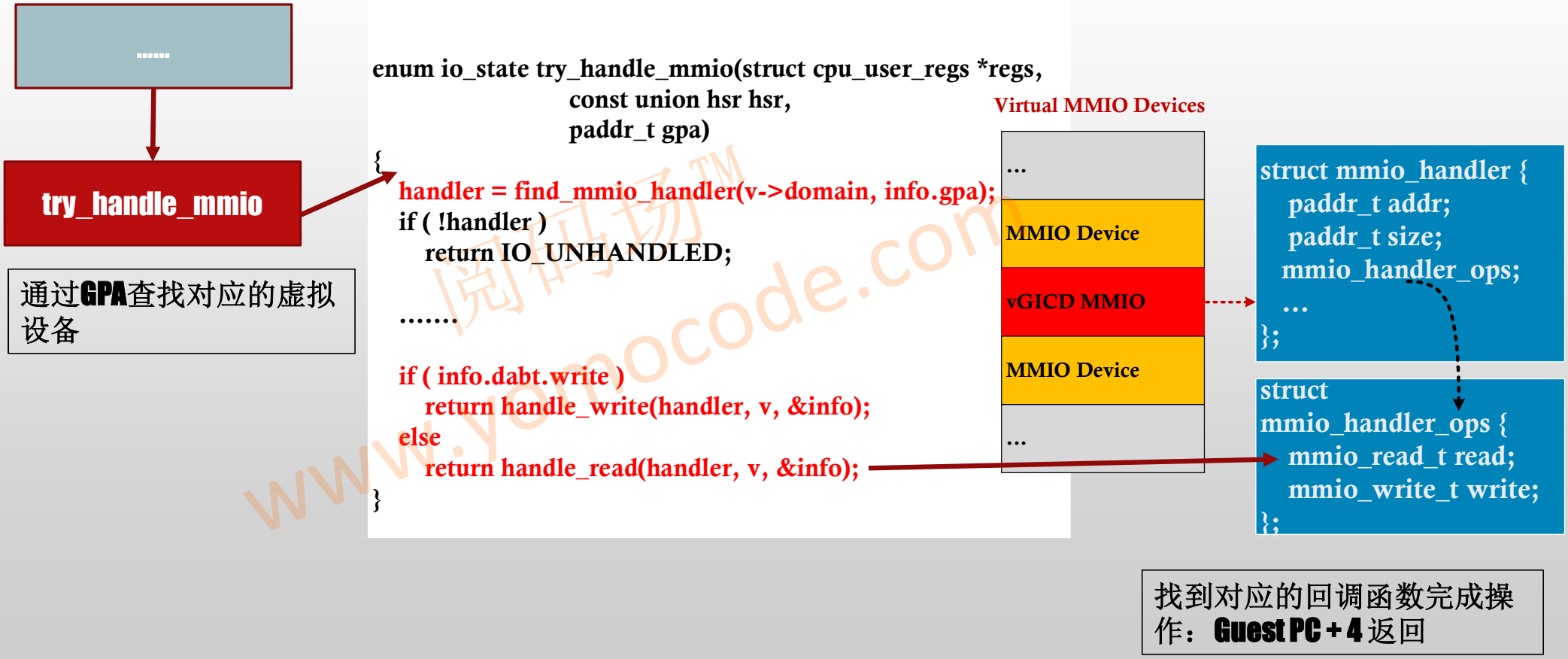
do_trap_stage2_abort_guest

EL1/EL0访问地址时发生了Stage2 转换异常

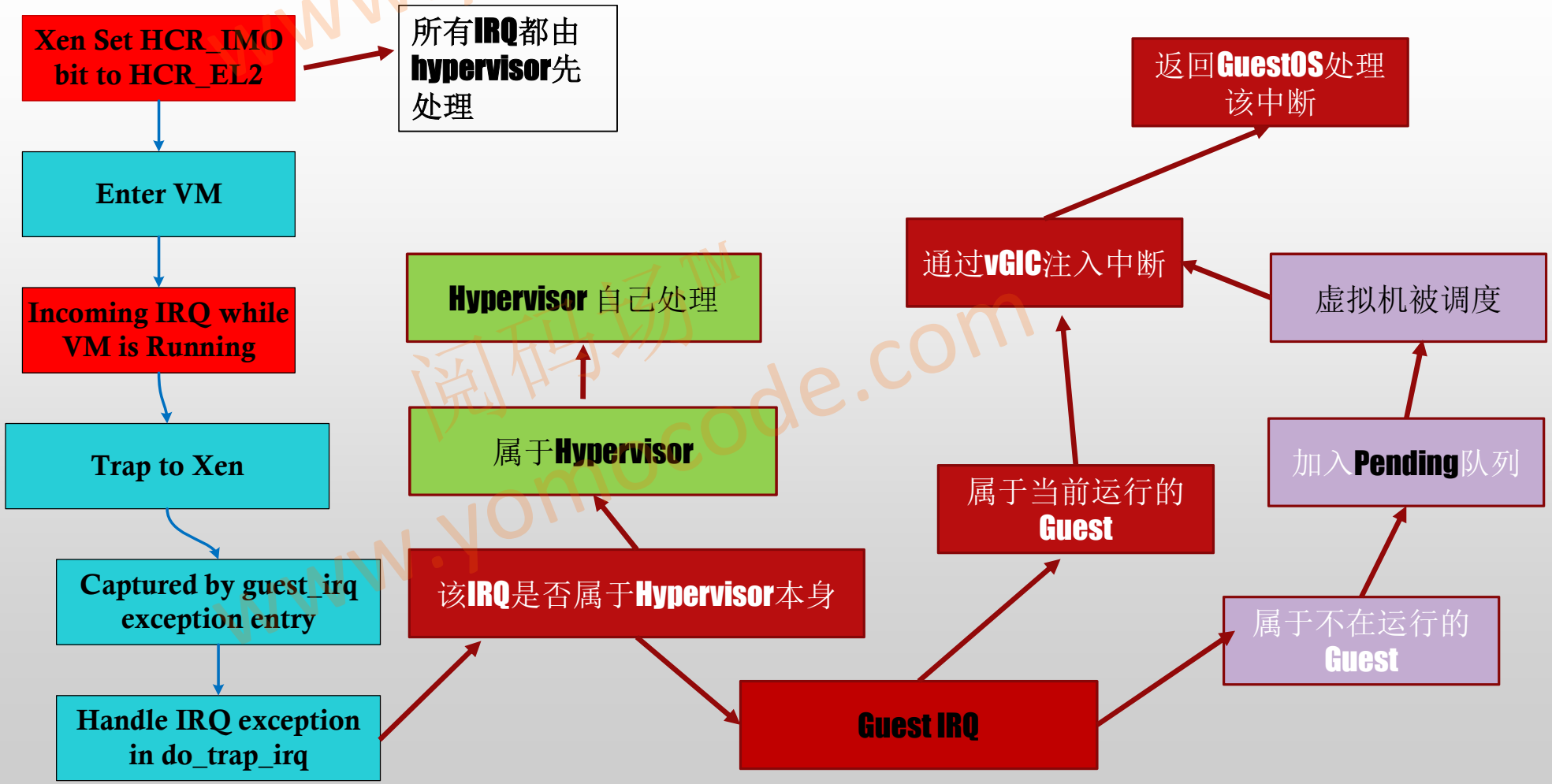
try_handle_mmio

通过GPA查找对应的虚拟设备

IO TRAP



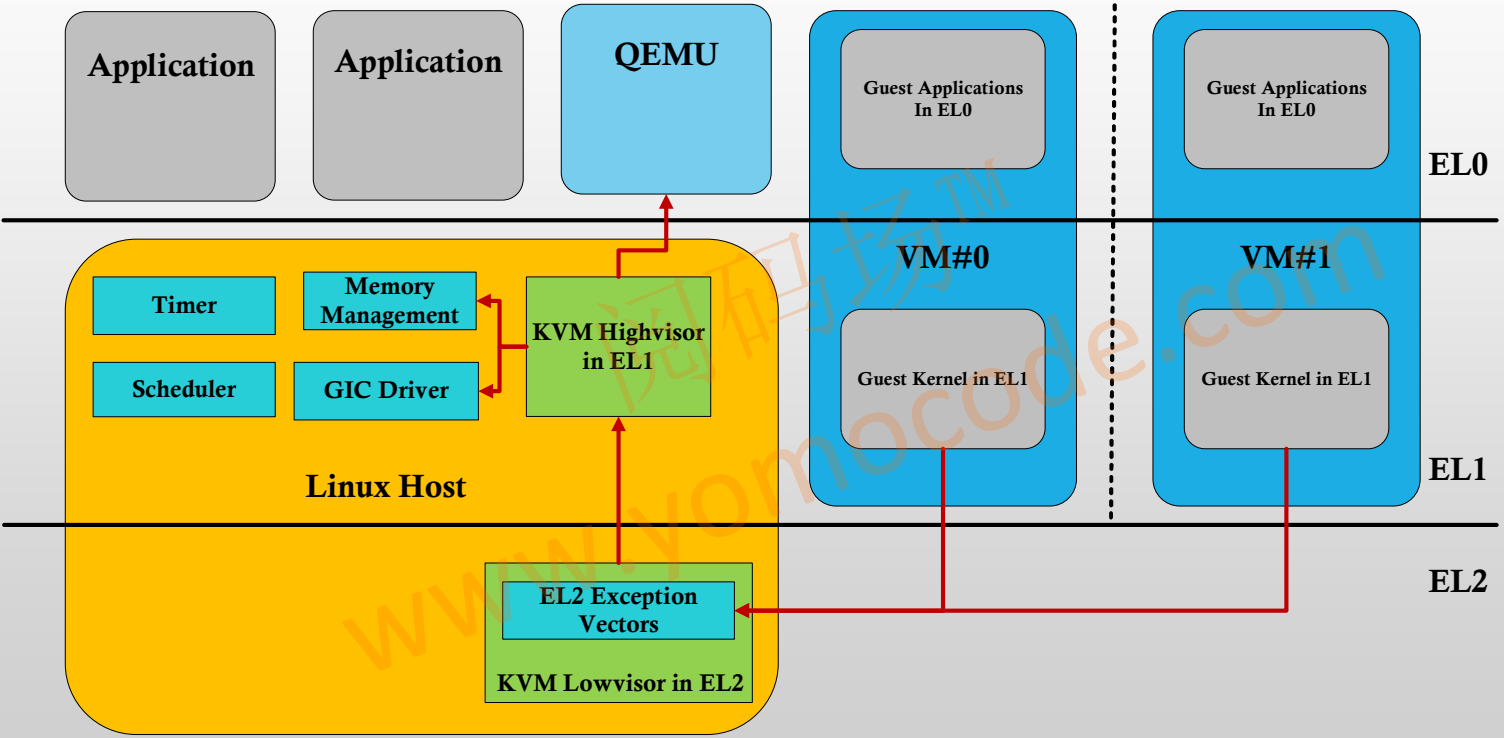
IRQ TRAP



DIFFERENCES OF KVM

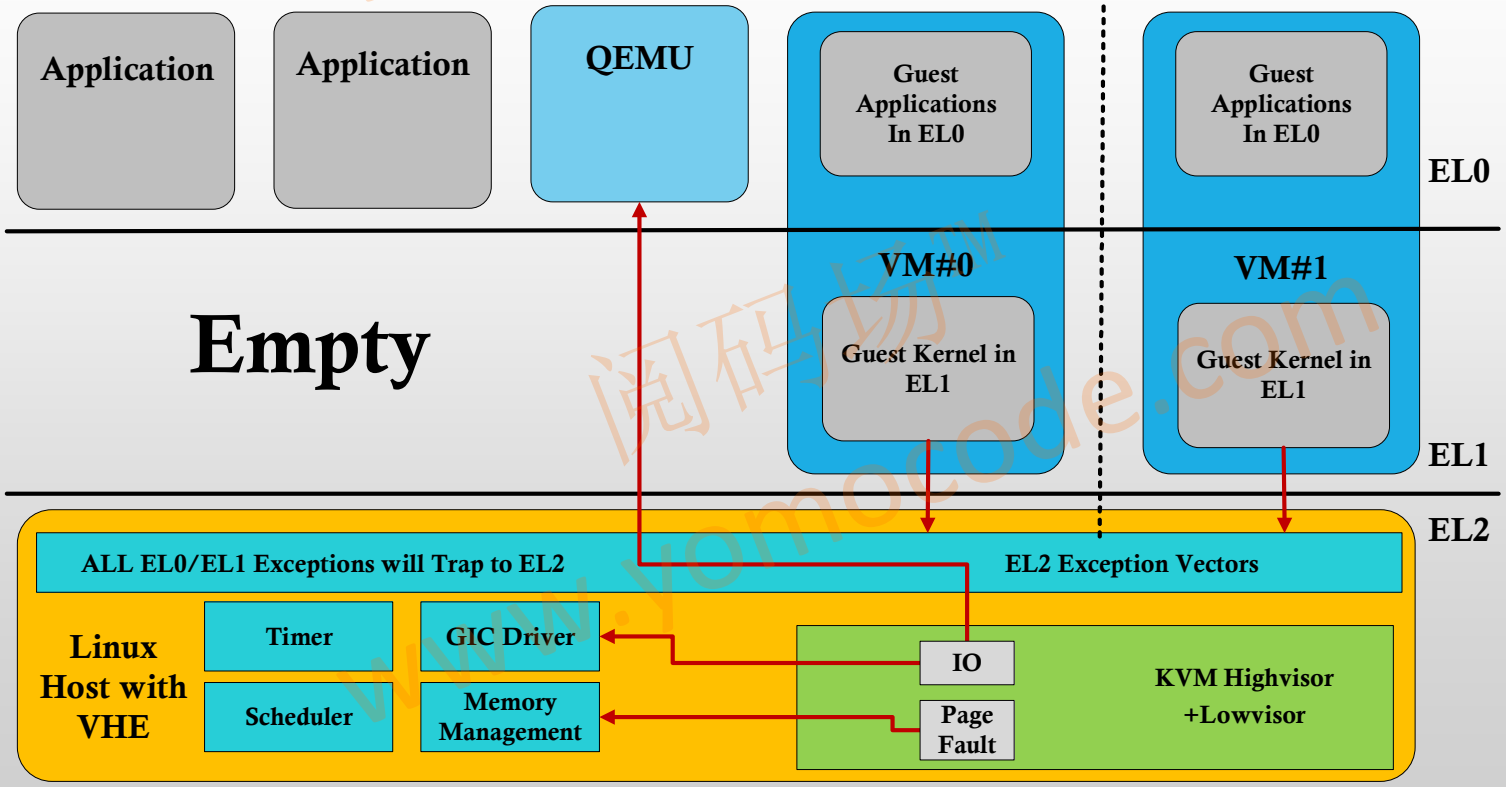
KVM和XEN的差异

OVERVIEW OF LINUX KVM



- 通过**Linux KVM**在**EL1**的**Highvisor**嵌入**EL2**调用**Lowvisor**访问只能在**EL2**访问的资源
- 使用**Linux host**内存管理器给虚拟机分配内存
- 虚拟机接受**Linux host**调度器的调度管理
- 中断控制器完全有**Linux host**掌控。**KVM Lowvisor**不处理中断。
- 设备虚拟化有内核和**QEMU**分别负责**(GIC,SMMU**由内核负责，**IO**操作交由**QEMU**完成**)**

OVERVIEW OF LINUX KVM



- **Linux Host** 整体下沉到**EL2, KVM Lowvisor**和**Highvisor**完成融合
- 消除了**Lowvisor**和**Highvisor**间的特权层切换**(EL2 ←→EL1)**
- **Linux Host** 的**EL1**层没有任何运行的代码


谢谢!
THANK YOU!