VM Exit 的处理过程

一、 相关信息：

1. VMCS 包含一个只读域提供 VM exit 的信息。 （手册 20.9）
   Exit reason（32 bits）其中 15：0 为 basic exit reason

2. VM Exit 进程包括从 Guest 向 VMM 传输指令或者数据，进入到 Root 模式，在 VMCS 保存 Guest 状态并且重新载入 Guest 状态
3. （手册 25.7） 处理 VM EXITs
   首先使用 VMREAD 指令获取 exit-reason（在 VMCS 里）
   VMRead exit-qualification 提供辅助信息
   根据 exit reason 获取其他相关的 VMCS 内的信息
   处理 VM-exit
   重新进入 vm 继续执行

二、 DebugView 中打印出的一段信息

以下共包含 2 次 VM Exit 的处理，分别对应 vmlauch 和 CR_ACCESS

```
00010052    0.60838979    <98>:   VmxDispatchEvent(): exitcode = 20//vmlaunch
00010053    0.60839295    <98>:   VmxHandleInterception(): Exitcode 20
00010054    0.60839611    <98>:   VmxDispatchEvent(): exitcode = 1c//CR_ACCESS
00010055    0.60839927    <98>:   VmxHandleInterception(): Exitcode 1c
00010056    0.60840195    <98>:   VmxDispatchCrAccess()
00010057    0.60840583    <98>:   VmxDispatchCrAccess(): gp: 0x2 cr: 0x3 exit_qualification: 0x203
00010058    0.60840923    <98>:   VmxDispatchCrAccess(): TYPE_MOV_TO_CR cr3:0x3d09d000
```

三、 NBP 对于 VM Exit 处理过程详解

1. **首先来看 VM Exit 引发后执行的函数是如何调用到的：**
   由于 VM Exit 会引发许多寄存器值刷新，这里主要关系到的是 RIP 和 RSP。在 VM Exit 触发后，RIP 和 RSP 会被 VMCS 中 Host RIP 和 Host RSP 域的值替换。
   在 Vmx.c 中的 VmxSetupVMCS 中

   VmxWrite (HOST_RIP, (ULONG64) VmxVmexitHandler);

   这句的把 VmxVmexitHandler 的函数指针写到了 HOST_RIP（0x00006c16）中，这样在 VM Exit 被触发并替换 RIP 的值后，就会自动执行 VmxVmexitHandler 这个方法。

   另外 *VmxWrite (HOST_RSP, (ULONG64) Cpu);*表示 VM Exit 触发后，RSP 的值会指向 Cpu 结构体。所以在执行 Trap 过程的堆栈在内存里的位置应该是和 Cpu 结构体的位置相关的。同时在调用 VmxVmexitHandler 时的第一个参数 PCPU Cpu，也就可以通过 rsp 的值来获得了。

2. **VmxVmexitHandler（Vmx-asm.asm）**

```
VmxVmexitHandler PROC

        HVM_SAVE_ALL_NOSEGREGS

        mov     rcx, [rsp + 80h] ;PCPU//由于之前保存了16个寄存器，所以这里移位值为16*8
                                 ;之后rcx指向栈中参数Cpu的位置
        mov     rdx, rsp                ;GuestRegs//栈中保存寄存器的位置
        mov     r8, 0           ;TSC
        ;调用规范:参数在寄存器 RCX、RDX、R8 和 R9 中传递
        sub     rsp, 28h        ;//指向rbp?

        ;rdtsc

        call    HvmEventCallback
        add     rsp, 28h
        HVM_RESTORE_ALL_NOSEGREGS
        vmx_resume
        ret

VmxVmexitHandler ENDP
```

前期的准备是为调用 HvmEventCallback 而做的，由于 HvmEventCallback 的参数为：PCPU Cpu, PGUEST_REGS GuestRegs。在 64 位系统中，由调用规范，可以看到，Cpu 指针被赋给了 rcx，GuestRegs 指针被赋给了 rdx（这里我的理解是，VmxVmexitHandler 首先执行了 HVM_SAVE_ALL_NOSEGREGS，此后 rsp 指向栈中保存的所有寄存器值的首地址，这样栈内的参数就等于自动与 GuestRegs 中的变量绑定，然后在 trap 方法中通过修改 GuestRegs 中相应变量的值，就相当于修改了栈中对应保存的寄存器的值，然后退出 VmxVmexitHandler 之前会调用 HVM_RESTORE_ALL_NOSEGREGS 来恢复寄存器，此时用来恢复的值已经由于保存在栈中已经被修改，也就达到了对输出结果的修改。

sub rsp,28h，这句根据 x64 调用规范，为调用函数为被调用函数分配参数在栈中的空间，然后被调用函数会在调用时把参数放到之前保留的空间中。由于必须至少保留 4 个寄存器参数的空间，因此至少要空出 20H 的空间（经测试，改为 20H 也可跑通）。

执行完 HvmEventCallback 后，把之前分配的空间取消也就是 add rsp, 28h。然后恢复寄存器（HVM_RESTORE_ALL_NOSEGREGS），然后通过 vmx_resume 把控制权交还给 VM。

3. HvmEventCallback（Hvm.c）

if (Hvm->Architecture == ARCH_VMX)

GuestRegs->rsp = VmxRead (GUEST_RSP);

RSP 在这里被 VmxRead 方法赋值，不过这个方法只是与 Intel 架构相关，GUEST_RSP 是指示存储在 VMCS 中的物理地址，从这里读到的值就是 Guest 的 rsp 的值。

//GUEST_RSP = 0x0000681c,

if (Hvm->ArchIsNestedEvent (Cpu, GuestRegs)) {

……

}

这里的 if 语句在 Intel 环境下不会进入，由于没有实现嵌套。

因此关键的语句是 Hvm->ArchDispatchEvent (Cpu, GuestRegs);

if (Hvm->Architecture == ARCH_VMX)

VmxWrite (GUEST_RSP, GuestRegs->rsp);

这两句与之前的 read rsp 的值对应。其实由于进入 vmm，之前提到的保存寄存器的动作执行时，有些寄存器的值是已经被修改了的，比如 rsp（此时已经是指向 vmm 的栈了），这时保存的值是不能和 GuestRegs 的 rsp 对应上的，也因此在做保存寄存器的值时，push 的是 rbp 不是 rsp：

```
push rbp        ; rsp
```

注释为 rsp，因为在 GuestRegs 中对应的位置放的应该是 rsp 值

对 rsp 的保存和恢复是必须通过 VMCS 中的 GUEST_RSP 域的赋值来实现的，这里通过 *GuestRegs->rsp = VmxRead (GUEST_RSP);*来获取 rsp 的值写到 GuestRegs 的对应变量中。

4. **VmxDispatchEvent （Vmx.c ArchDispatchEvent 对应的 intel 架构函数指针）**

向下函数调用 *VmxHandleInterception (Cpu, GuestRegs, FALSE*

/ this intercept will not be handled by guest hv */*

);

5. **VmxHandleInterception （Vmx.c）**

Exitcode = VmxRead (VM_EXIT_REASON);

读取 VMCS 中的 EXIT_REASON，Exitcode 的值与含义的对应关系可在 Vmx.h 中找到，均有芯片固定提供。

Status = TrFindRegisteredTrap (Cpu, GuestRegs, Exitcode, &Trap);

在开始绑定的处理函数与 Exit_reason 的 Trap 的链表中找对应的 Trap 附给 Trap 变量，找到的话返回 STATUS_SUCCESS，没找到返回 STATUS_NOT_FOUND

```
if (!NT_SUCCESS (Status = TrExecuteGeneralTrapHandler (Cpu, GuestRegs, Trap, WillBeAlsoHandledByGuestHv))) {
  _KdPrint (("VmxHandleInterception(): HvmExecuteGeneralTrapHandler() failed with status 0x%08hX\n", Status));
}
```

执行 TrExecuteGeneralTrapHandler。

6. **TrExecuteGeneralTrapHandler（Traps.c）**

if (Trap->TrapCallback (Cpu, GuestRegs, Trap, WillBeAlsoHandledByGuestHv)) {

// trap handler wants us to adjust guest's RIP

Hvm->ArchAdjustRip (Cpu, GuestRegs, Trap->General.RipDelta);

}

Trap->TrapCallback (Cpu, GuestRegs, Trap, WillBeAlsoHandledByGuestHv)这句相当于调用了对应的处理函数。以 CPUID 为例，调用的就是 VmxDispatchCpuid（VmxTraps.c）。

Hvm->ArchAdjustRip (Cpu, GuestRegs, Trap->General.RipDelta);

内部执行的是 VmxWrite (GUEST_RIP, VmxRead (GUEST_RIP) + Delta);

可以理解为由于对某条指令执行了 trap，在把控制权返还 VM 时，需要告诉 VM 跳过 trap 的那条指令。

四、 添加 Trap

Hvm.c 的 Status = Hvm->ArchRegisterTraps (Cpu);

会调用到 Vmxtraps.c 中的 VmxRegisterTraps 方法

这个方法初始化了目前虚拟机 trap 的相关内容，对每一种 trap 调用

TrInitializeGeneralTrap 或 TrInitializeMsrTrap（intel 部分未使用）或 TrInitializeIoTrap（均未使用）

来绑定相应的处理函数和属性，再调用 TrRegisterTrap 把初始完的一个 trap 添加到对应的链表里。

```
if (!NT_SUCCESS (Status = TrInitializeGeneralTrap (Cpu, EXIT_REASON_CPUID, 0, // length of the instruction, 0 mea
                                                   VmxDispatchCpuid, &Trap))) {
  _KdPrint (("VmxRegisterTraps(): Failed to register VmxDispatchCpuid with status 0x%08hX\n", Status));
  return Status;
}
TrRegisterTrap (Cpu, Trap);
```

简单地说就是通过上图的类似方法添加即可。

目前看来，这样的添加只对某些开启的会引发 VM Exit 的一些指令相关（如何开启时通过在 VMCS 相应位

置设定值来定义的）。