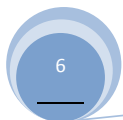


# 目录

前言	7
鸣谢	8
关于作者.....	9
本书简介.....	10
一、 概述.....	12
Hypervisor 概述.....	12
虚拟化的历史.....	12
硬件虚拟化技术（HEV）.....	12
HEV 技术最佳实践.....	13
已有的 HEV 技术平台介绍.....	15
SVM.....	16
概述.....	16
新的地址翻译机制.....	17
相关结构和汇编指令.....	18
基于 SVM 的 Hypervisor 开发逻辑.....	18
Intel-VTx.....	19
概述.....	19
新的地址翻译机制.....	19
相关结构和汇编指令.....	20
基于 VT 的 Hypervisor 开发逻辑.....	22
Intel-VTd(如果时间充裕则写).....	22
NewBluePill 项目介绍.....	23
PART1 HEV 技术相关知识.....	25
二、 深入 HEV 技术细节.....	25
HEV 下虚拟机启动过程.....	25
启动过程模型.....	25
VT 技术下开启虚拟机的过程.....	28
SVM 技术下开启虚拟机的过程.....	29
HEV 下虚拟机关键结构体.....	29
VT 技术下的 VMCS 结构体.....	30
虚拟机状态保存区（Guest-State Area）.....	31
宿主机（Hypervisor）状态保存区（Host-State Area）.....	33
虚拟机运行控制域（VM-Execution Control Fields）.....	33
VMEntry 行为控制域（VM-Entry Control Fields）.....	37
VMEXIT 行为控制域（VM-Exit Control Fields）.....	38
VMEXIT 相关信息域（VM Exit Information Fields）.....	39
SVM 技术下的 VMCB 结构体.....	39
HEV 下虚拟机关闭过程.....	39
VT 技术下关闭 Hypervisor 和虚拟机的过程.....	39
SVM 技术下关闭 Hypervisor 和虚拟机的过程.....	39
PART2 深入研究 NewBluePill.....	42
三、 体验 NewBluePill.....	42

编译 NewBluePill .....	42
演示 NewBluePill .....	44
调试 NewBluePill .....	47
四、NewBluePill 的启动和卸载 .....	50
NewBluePill 驱动的启动过程 .....	50
构建私有页表 .....	50
初始化调试系统 .....	51
构建 Hypervisor 并将操作系统放入虚拟机 .....	51
进入 NewBluePill 的世界 .....	52
阶段 1 初始化 .....	52
阶段 2 初始化 .....	60
NewBluePill 驱动的卸载过程 .....	61
2. 具体描述 .....	62
1) Nbp 的卸载过程 .....	62
2) Bpknock 的作用 .....	63
五、NewBluePill 内存系统 .....	64
1) 相关文件: .....	64
2) 技术背景: .....	64
3) 总体功能介绍: .....	67
4) 实现过程: .....	67
MmInitManager() 方法 .....	67
MmSavePage () 方法 .....	68
MmSavePage () 方法 .....	70
MmSavePage () 方法 .....	70
六、NewBluePill 陷入事件管理系统 .....	71
1) 注册机制 .....	71
2) 触发机制 .....	71
3) 各处理函数功能和实现 .....	71
七、NewBluePill 反探测系统 .....	72
八、NewBluePill 其它系统 .....	73
3) DbgClient 的初始化过程 .....	73
4) DbgClient 的卸载过程 .....	73
PART3 实验部分 .....	75
九、动手写自己的第一个 HVM 程序 .....	75
实验目的 .....	75
实验概述 .....	75
实验过程 .....	75
十、移植 NBP 到 32 位系统 .....	76
十一、开发自己的序列号验证器 .....	77
A. 其它有关 HVM 技术的项目 .....	78
B. 其它安全技术 .....	79
C. 相关软件和参考文档 .....	81



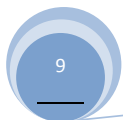
## 前言



鸣谢



## 关于作者



## 本书简介

好的数据结构是程序的灵魂，一个定义优秀的数据结构能够使人立刻读懂其中的算法。所以我们更着重于介绍其中所涉及的数据结构，力求用最短的篇幅说明 NewBluePill 的世界。

~~本书假定读者仅理解最基本的术语，不具有嵌入式开发经验，当然如果有相关经验的话就可以更快的理解书中内容。~~

~~对于没有太多基础的读者，建议先看第三章 一些背景知识，然后再看第二章和后续章节。~~  
本书假定读者具有一定嵌入式软件开发经验，对于没有太多基础的读者，建议先阅读 *Windows Internals, 4<sup>th</sup> Edition* 和一些 x86 平台相关开发书籍。

写这本书的目的只是去引导读者阅读代码，因此并不具体到代码中涉及的每个角落。

本书涉及的表示方法：

<i>Windows Internals, 4<sup>th</sup> Edition</i>	书名
Hypervisor	英文名词
<a href="http://www.bluepillproject.org/">http://www.bluepillproject.org/</a>	超级链接
HvmSubvertCpu()	书中代码及函数名

(This page is intended to be blank)



## 一、概述

在这一章中，我们先介绍一些贯穿全书的概念，比如 Hypervisor, VT-x, VT-d, SVM 等。然后我们会简略介绍下 NewBluePill 项目背景及其所采用的硬件虚拟化技术。

这一章只是介绍这些技术大致的轮廓，详细内容会在后面各章节中逐一介绍。

## Hypervisor 概述

### 虚拟化的历史

在讨论 Hypervisor 之前首先谈谈虚拟，虚拟 (virtualization) 指对计算机资源的抽象，一种常用的定义是“虚拟就是这样的一种技术，它隐藏掉了系统，应用和终端用户赖以交互的计算机资源的物理性的一面，最常做的方法就是把单一的物理资源转化为多个逻辑资源，当然也可以把多个物理资源转化为一个逻辑资源（这在存储设备和服务器上很常见）”

实际上，虚拟技术早在 20 世纪 60 年代就已出现，最早由 IBM 提出，并且应用于计算技术的许多领域，模拟的对象也多种多样，从整台主机到一个组件，其实打印机就可以看成是一直在使用虚拟化技术的，总是有一个打印机守护进程运行在系统中，在操作系统看来，它就是一个虚拟的打印机，任何打印任务都是与它交互，而只有这个进程才知道如何与真正的物理打印机正确通信，并进行正确的打印管理，保证每个 job 按序完成。

长久以来，用户常见的都是进程虚拟机，也就是作为已有操作系统的一个进程，完全通过软件的手段去模拟硬件，软件再翻译内存地址的方法实现物理机器的模拟，比如较老版本的 VMWare, VirtualPC 软件都属于这种。

在 2005 年和 2006 年，Intel 和 AMD 都开发出了支持硬件虚拟技术的 CPU，也就是在这时，x86 平台才真正有可能实现完全虚拟化<sup>1</sup>。在 2007 年初的时候，Intel 还进一步的发布了 VT-d 技术规范，从而在硬件上支持 I/O 操作的虚拟化。随着硬件虚拟化技术越来越广泛的采用，开发者也开始虚拟技术来做一些其他的事情：当前 HVM 已经在虚拟机，安全，加密等领域上有所应用，例如 VMware Fusion, Parallels Desktop for Mac, Parallels Workstation 和 DNGuard HVM，随着虚拟化办公和应用的兴起，相信虚拟化技术也会在未来得到不断发展。

### 硬件虚拟化技术 (HEV)

有了虚拟技术的基本概念，下面我们谈谈硬件虚拟化技术。硬件虚拟化技术 (Hardware Enabled Virtualization, 本书中简称 HEV)，也就是在硬件层面上，更确切的说是在 CPU 里 (VT-d 技术是在主板上北桥芯片支持)，对虚拟技术提供直接支持。在硬件虚拟化技术诞生前，编写虚拟机过程中，为了实现多个虚拟机上的真实物理地址隔离，需要编程实现把客户机的物理地址翻译为真实机器的物理地址。同时也需要给不同的客户机操作系统编写不同的虚拟设备驱动程序，使之能够共享同一真实硬件资源。硬件虚拟化技术则在硬件上实现了内存地址甚至于 I/O 设备的映射，因此大大简化了编写虚拟机的过程。而其硬件直接支持二次寻址和 I/O 映射的特性也提升了虚拟机在运行时的性能。<sup>2</sup>

<sup>1</sup> 完全虚拟化(Full Virtualization)，完整虚拟底层硬件，这就使得能运行在该底层硬件上的所有操作系统和它的应用程序，也都能运行在这个虚拟机上。

<sup>2</sup> 一些优化技术也在硬件中被采用，比如专门用于二次寻址的 TLB，详细信息可以参考 Intel 和 AMD 的手册

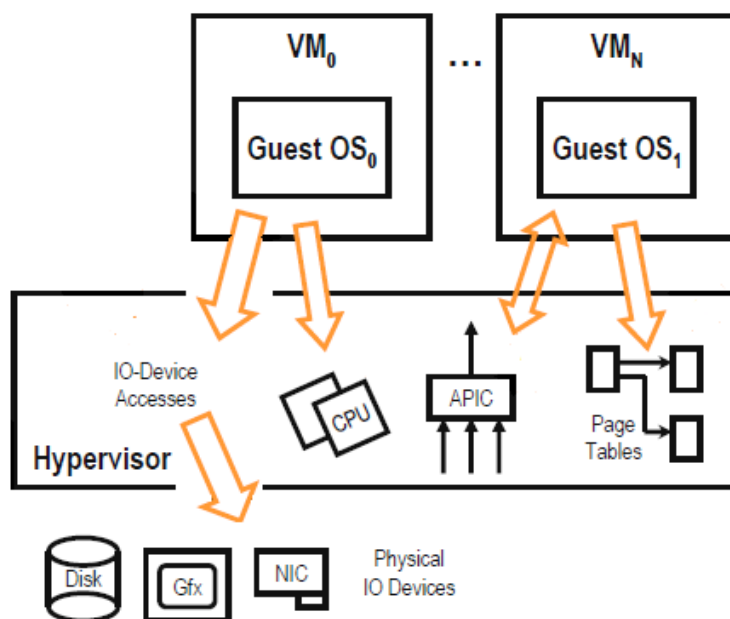


图 1.1 硬件虚拟化技术架构示意图

在硬件虚拟化技术中，一个重要的概念就是 HVM。HVM, Hypervisor Virtual Machine 的缩写（在本书中简称为 Hypervisor），是在使用硬件虚拟化技术时创建出来的特权层，该层提供给虚拟机开发者，用来实现虚拟硬件与真实硬件的通信和一些事件处理操作（如图 1.1），因此 Hypervisor 的权限级别要高于等于操作系统权限。

## HEV 技术最佳实践

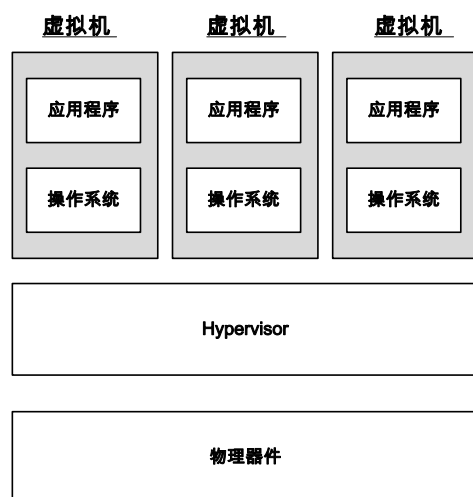


图 1.2 Hypervisor 使用架构图

前文中已经提过，Hypervisor 层的权限要高于等于操作系统的权限。操作系统的内核态已经处在了 Ring0 特权级上，因此 Hypervisor 层实际上要运行在一个新的特权级别上，我们称

之为“Ring -1”特权级。同时需要新的指令，寄存器以及标志位去实现这个新增特权级的功能。

作为一种最佳实践方案，一般 Hypervisor 层的实现都是越简单越好。一方面，简单的实现能够尽量降低花在 Hypervisor 上的开销<sup>1</sup>，毕竟大多数这些开销在原先的操作系统上是不存在的。另一方面，复杂的程序实现容易引入程序漏洞，Hypervisor 也是如此，且一旦 Hypervisor 中的漏洞被恶意使用，由于其所处特权级高于操作系统，将使隐藏在其中的病毒、恶意程序很难被查出。

批注 [S1]: 后面要描述 Hypervisor 的开销问题

Virtualization.pdf 10

## HEV 技术所带来的性能损耗

新技术在使得开发人员的世界变得更加美好的同时，也不可避免的带来性能上的冲击。

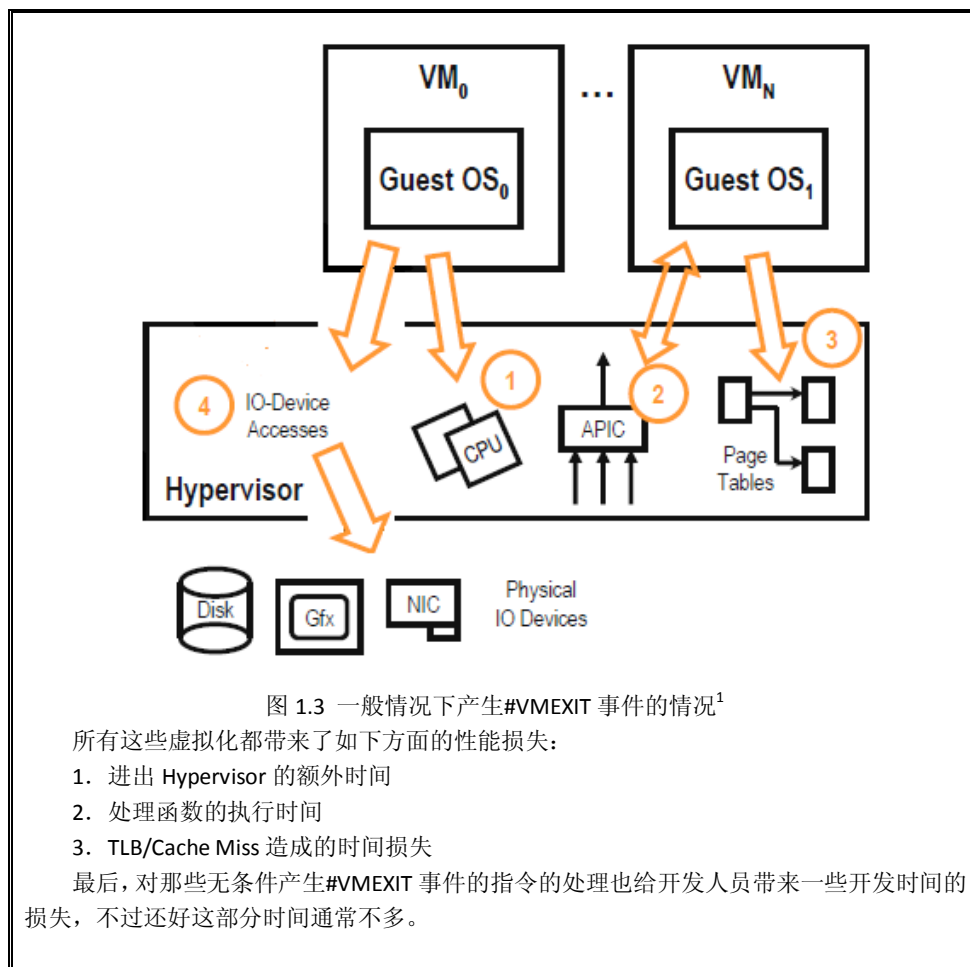
HEV 技术中，性能损耗最大的地方在于 Hypervisor 的引入及其所造成的需要进出 Hypervisor。一个最简单的例子，在普通的 x86 保护模式下，运行时刻执行到 CPUID 指令时，处理器<sup>2</sup>会根据 EAX (RAX) 寄存器的值直接读取 MSR 寄存器，并把结果写到 EAX~EDX (RAX~RDX) 寄存器。但是在 Guest 模式下并且设置对 CPUID 指令进行拦截，那么每当 Guest OS 执行到 CPUID 指令时，处理器都会产生 #VMEXIT 事件，从而陷入 Hypervisor 中对该指令进行相应处理，这个过程中涉及到 Guest 模式寄存器的保存，Host 模式寄存器的恢复，填充 VMCS/VMCB 中相应内容（都是一系列处理器自动完成的内存操作），然后 Hypervisor 中不可缺少的有 CPUID 指令陷入的处理，最后在 Hypervisor 处理完后，处理器要回到 Guest 模式，这又涉及到 Host 当前寄存器的保存，Guest 模式寄存器的恢复，以及 VMCS/VMCB 中相应内容的填充。显然，花费在这上面的指令周期数将是保护模式下 CPUID 一条汇编指令所消耗的指令周期数的成千上万倍以上。

而在更一般的情况下，下列四种产生 #VMEXIT 事件的情况都是需要处理的：

1. 访问特权级别的 CPU 的状态（Access to Privileged CPU State）
2. 中断虚拟化（Interrupt Virtualization）
3. 页表虚拟化（Page-Table Virtualization）
4. IO 设备虚拟化（IO-device virtualization）

<sup>1</sup> 关于 Hypervisor 的开销问题，后面的章节会有介绍

<sup>2</sup> 注意：本书中强调处理器（Processor）和 CPU 的差别，文中若无特别说明，处理器指逻辑处理器（Logical Processor）。在多核时代，一个 CPU 上可能会有多个核，而在操作系统视角中，一个核才是一个逻辑处理器，因此通过操作系统查看的逻辑处理器数量往往大于真实 CPU 的数量，并且逻辑处理器才是能够运行 Hypervisor 的基础。



## 已有的 HEV 技术平台介绍

现今两大主要硬件厂商 Intel 和 AMD 均以推出了支持硬件虚拟化技术的产品，两者大体功能和实现方法近似（意料之中，因为两家公司在过去你死我活的市场拼斗中，每次也都是实现功能和方法类似，只不过名字不同罢了）。下面我们简略介绍下这两家公司的支持 HEV 技术的平台，读者可以首先对这两种平台有概念。后面的章节中会有对这两个平台技术细节的更详细描述。

<sup>1</sup> 此图摘自 *Intel® Virtualization Technology Processor Virtualization Extensions and Intel® Trusted execution Technology*, Gideon Gerzon

## SVM

### 概述

AMD 芯片支持硬件虚拟化的技术被称作 AMD-V (在技术文档中也被称为 SVM,其全称是 AMD Secure Virtual Machine, 在本书中我们仍称其为 SVM)。其主要是通过一组能够影响到 Hypervisor 和 Guest Machine (客户机, 下文简称 Guest) 的中断实现的。AMD-V 技术设计目标如下:<sup>[2]</sup>

- 引入客户机模式 (Guest Mode)<sup>1</sup>
- Hypervisor 和 Guest 之间的快速切换
- 中断 Guest 中特定的指令或事件(events)
- DMA 访问保护
- 中断处理上的辅助并对虚拟中断 (virtual interrupt) 提供支持
- 新的嵌套页表用来实现地址翻译
- 一个新的 TLB (其实就是一个 Cache) 来减少虚拟化造成的性能下降。
- 对系统安全的支持

**新的客户机模式** 通过 VMRUN 指令即可进入这种新的处理器模式, 当进入客户机模式后, 为了辅助虚拟化过程, 一些 x86 汇编指令的语义会发生变化。

**外部访问保护** 过去客户机 (Guest) 可以直接访问选定的 I/O 设备。现在硬件上已经实现这样的安全功能, 能够阻止某个虚拟机拥有的某个设备访问其它虚拟机的内存。

**中断上的支持** 为了辅助中断的虚拟化, 下列各项现在已经得到硬件支持, 并且可以通过配置 VMCB 结构体<sup>2</sup>的方法使用

- 1) 拦截物理中断分发 (Intercepting physical interrupt delivery) 发生在物理硬件上的中断能够让虚拟机发生一个中断, 陷入 Hypervisor, 从而使得 Hypervisor 可以首先处理这个中断
- 2) 虚中断 (Virtual Interrupts) Hypervisor 能够将提供给客户机 (Guest) 一套虚拟的中断机制。它是这样实现的, Hypervisor 会给这个客户机复制出来一份 EFLAGS.IF 用做中断屏蔽位 (Interrupt Mask Bit), 同时复制 APIC<sup>3</sup>中的中断优先级寄存器提供给客户机, 从而客户机就会去操纵这套假的中断机制, 而不是直接去操纵物理中断。
- 3) 共享物理 APIC AMD 的 SVM 技术能够允许多个 Guest 共享同一物理 APIC, 同时又能保护这个 APIC 以免某个客户机不慎或恶意的在未经其它客户机许可的情况下, 将可接收中断优先级设置为高优先级, 从而清空了所有其它 Guest 的中断。

**被标记的 TLB (Tagged TLB)** 为了降低 Guest 模式和 VMM 模式切换开销, TLB 上新加了一个 ASID 标记 (Address Space Identifier), 这个标记可以区分 TLB 上的一块地址是

<sup>1</sup> x86 上原有处理器模式包括保护模式(Protected Mode), 管理模式(SMM), 实模式(Real Mode)

<sup>2</sup> VMCB 结构体, Virtual Machine Control Block, 也称 VMCB 控制块, Intel 的相应结构体名称为 Virtual Machine Control Sector, VMCS。这个控制块用于通知物理 Processor 要拦截的事件, 以及在进出 Hypervisor 上下文切换时保存 Hypervisor 和 Guest 的各项寄存器, 后面的章节中会有对这个结构体的详细介绍

<sup>3</sup> APIC, Advanced Programmable Interrupt Controller, 高级可编程中断控制器, 第三章有关于此主题内容。

**批注 [S2]:** 后面的章节要详细介绍 VMCB 结构体

Hypervisor 范围内的地址还是 Guest 的地址，从而加速了地址翻译。

**安全方面的支持** 现在提供的安全方面的支持主要是利用和 TPM 模块（Trusted Platform Module）<sup>1</sup>的交互，基于与安全 Hash 值的比较。

批注 [S3]: 第 17 章 其它安全技术写些有关 TPM 技术的介绍

## 新的地址翻译机制

AMD 引入了新的地址翻译技术——嵌套页表翻译（Nested Page Table, NPT），用于支持两级地址翻译，这样就使得虚拟机管理器不用再自己软件维护一套影子页表<sup>2</sup>

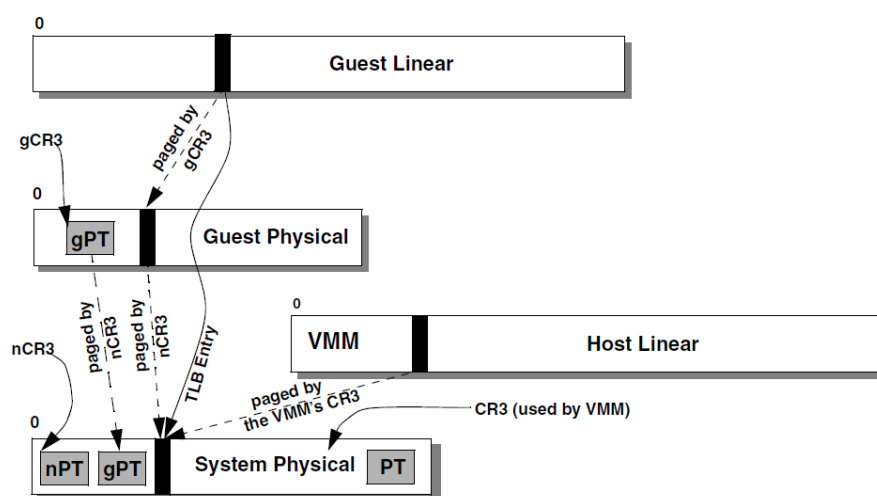


图 1.4 嵌套页表翻译地址过程<sup>3</sup>

嵌套页表翻译地址的过程如图 1.4 所示，这套机制的实现允许了从 Guest 线性地址到真实物理地址的翻译，也允许了在 Hypervisor 范围内的 Host 线性地址到真实物理地址的翻译。同时专门附加了一个 TLB 寄存器，用于缓存从 Guest 线性地址到真实物理地址的映射，从而提升了虚拟机的运行性能。

**Note** 关于嵌套页表技术的详细解释会在“第四章 深入 HEV 技术细节”一章中介绍，完整的描述请参考 *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*

批注 [S4]: 记得所有书名用斜体

<sup>1</sup> TPM 技术会在“第 17 章 其它安全技术”一章中介绍。

<sup>2</sup> 影子页表 (Shadow Page Tables)，常见于过去传统的虚拟机管理器中，由于存在从 Guest 线性地址到 Guest 物理地址，从 Guest 物理地址到真实物理地址两层地址翻译，所以在过去一般是要虚拟机软件自己维护两套页表去做这样的地址翻译。

<sup>3</sup> 此图摘自 AMD64 Architecture Programmer's Manual, Volume 2: System Programming

## 相关结构和汇编指令

在 SVM 中，VM 控制块被称为 VMCB (Virtual Machine Control Blocks)，其信息主要分为两块，第一块是控制信息存储部分，同时也包含是否允许拦截某特定异常的遮罩(interception enable mask)，Guest 中不同的指令和事件都能以修改 VMCB 中相应控制位的方法拦截，SVM 支持的两类主要的拦截是异常拦截和指令拦截，第二部分则是 guest 的状态信息保存，这里会保存段寄存器以及大部分的虚拟内存的入口控制寄存器，不过浮点寄存器信息不会被保存。需要注意的是 VMCB 在不同的处理器间不共享，并且 VMCB 一定要保证是在 4K 页对齐的连续物理内存空间中。

SVM 中主要的指令有以下这些：

- VMLOAD 从VMCB加载guest的状态，VMCB与guest是有对应关系的。
- VMMCALL 通过该方法guest可以与VMM显式的交流，方法是利用生成#VMEXIT从guest层退到VM层。
- VMRUN 加载VMCB，并开始执行guest层的指令，VMCB的物理地址将通过RAX获得，这个VMCB对应于要执行的guest
- VMSAVE 存储处理器状态的子集到VMCB中，这个VMCB的物理地址由RAX寄存器给出。
- STGI 用于设置全局中断标志（Global Interruption Flag）为1，这个指令属于Secure Virtual Machine。
- CLGI 用于设置全局中断标志（Global Interruption Flag）为0，同样这个指令属于Secure Virtual Machine。
- INVLPGA 使得TLB上一个ASID和一个虚拟页(Virtual Page)之间的映射关系无效，这个指令属于Secure Virtual Machine。
- SKINIT 安全的重新初始化CPU，使得CPU可以开始执行一段受信任的程序（trusted software）其方法是将该代码进行安全的哈希比较(secure hash comparison)。这也就是的开发者可以开发一个更安全的VMM loader。这种安全手段可以在TPM的帮助下发挥更大作用
- 改进的MOV指令 现在的MOV指令可以直接读写CR8寄存器（任务优先级寄存器 Task Priority Register），因此可以用来提高SVM应用的性能。

## 基于 SVM 的 Hypervisor 开发逻辑

其实由上文的描述可以看出，开发基于SVM的Hypervisor最主要是编写一个循环，这个循环要包含VMRUN命令以便从VM层启动一个Guest虚拟机，也要包含一段程序用于处理当#VMEXIT发生后的异常情况，这其中可能要手动做一些必要的保存现场和恢复现场的工作，具体造成异常的起因等均可通过读取VMCB中的数据获得。不过SVM没有提供一个显示终止Hypervisor的指令，因此若有需要，则要用其它方法关闭Hypervisor。NewBluePill中对SVM的支持就是这样实现的，我们会在深入探究NewBluePill的章节中详细展示怎样使用这些指令。

批注 [55]: 后面要介绍 NBP 中关于 SVM 技术的运用



## Intel-VT<sub>x</sub>

### 概述

Intel 芯片支持硬件虚拟化的技术被称为 Intel VT 技术（Intel® Virtualization Technology）。与 SVM 一样，其主要也是通过一组能够影响到 Hypervisor 和 Guest Machine 的中断实现的。

在 VT 技术中，与 SVM 类似的，设计架构上同样存在两种角色——虚拟机管理器（Virtual Machine Monitors, VMM）和客户机（Guest），两者分处在 VMX root 模式和 VMX non-root 两种模式下。VT 技术的设计目标是：

对于 VMM 层：（进入此层则代表进入了 VMX root 模式）

- 为每个虚拟机提供虚拟处理器，并且可以在恰当的时候把它放在真正的物理处理器上，从而使得这个虚拟处理器可以处理指令。
- VMM 层可以控制处理器资源，物理内存，管理中断和 I/O 操作

对于 Guest Machine：（进入此层则代表进入了 VMX non-root 模式）

- 每个虚拟机使用相同的接口来使用虚拟处理器，内存，存储设备等资源
- 每个虚拟机可以独立的不受干扰的运行，虚拟机间都是相互独立的
- 对于虚拟机来说，VMM 层像是完全透明的。

在 VT 技术下的 Hypervisor 生命周期如图 1.5 所示：

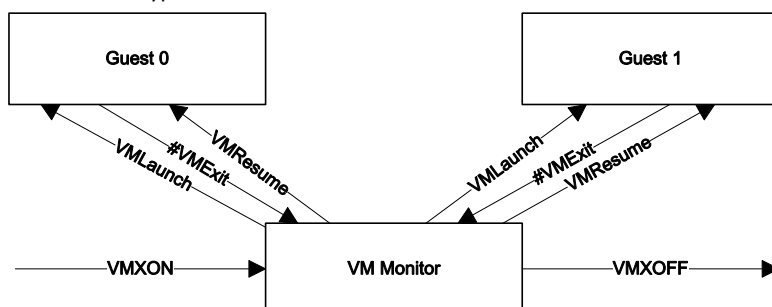


图 1.5 VT 技术中 Hypervisor 的生命周期

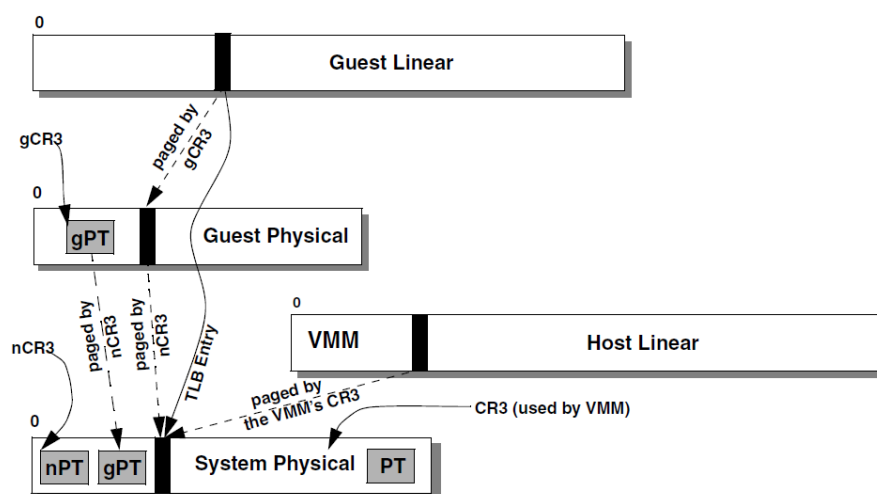
图示表明，软件通过执行 VMXON 指令进入 VMX Root 模式下，开启了虚拟机管理器的运行环境。然后通过使用 VMLaunch 指令使得目标系统正式运行在虚拟机中。当某条指令产生了 #VMEXIT 事件后，会陷入虚拟机管理器中，待其处理完这个事件，可以通过 VMXResume 指令将控制权移交回发生 #VMEXIT 事件的虚拟机。直到某个时刻，在 Hypervisor 中显示的调用了 VMXOFF 指令，Hypervisor 才会被关闭。

### 新的地址翻译机制

Intel 同样引入了新的地址翻译技术——扩展页表翻译（Extended Page Table, EPT），用于支持两级地址翻译。

批注 [S6]: 仔细看完 EPT 技术后补充这一部分



图 1.5 嵌套页表翻译地址过程<sup>1</sup>

嵌套页表翻译地址的过程如图 1.4 所示，这套机制的实现允许了从 Guest 线性地址到真实物理地址的翻译，也允许了在 Hypervisor 范围内的 Host 线性地址到真实物理地址的翻译。同时专门附加了一个 TLB 寄存器，用于缓存从 Guest 线性地址到真实物理地址的映射，从而提升了虚拟机的运行性能。

**Note** 关于扩展页表技术的详细解释会在“第四章 深入 HEV 技术细节”一章中介绍，完整的描述请参考 Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B, Chapter 24. Support for Address Translation

## 相关结构和汇编指令

在 VT 技术中，VM 控制块被称为 VMCS (Virtual Machine Control Structure)。VMCS 包括三个组成部分：

表 1.1 VMCS 区域的组成部分

Byte 偏移量	内容
0	VMCS 版本标志 (Revision Identifier)
4	VMX 退出原因指示器 (VMX-abort indicator) <sup>2</sup>
8	VMCS 数据区

批注 [S7]: 后文会详细介绍 VMX Abort

如表 1.1 所示，VMCS 区域的前四个字节用于 VMCS 版本标志，不同的 VMCS 格式对应的版本号也不同，而这个物理处理器可以加载的 VMCS 结构体的版本号会存储在 MSR 寄存器中，因此这样的设计也就给未来发展留下了空间。

<sup>1</sup> 此图摘自 AMD64 Architecture Programmer's Manual, Volume 2: System Programming

<sup>2</sup> 如果在 VM Exit 的时候遇到问题，就会发生 VMX Abort，一旦发生，那么这个逻辑处理器会进入关闭状态 (Shutdown State)

在 VMCS 数据区中，主要有如下几个组成部分：

表 2.2 VMCS 数据区主要组成部分<sup>[3]</sup>

名称	作用
虚拟机状态保存区 (Guest State Area)	当发生了#VMEXIT 事件时虚拟机当前状态保存于此，在重新进入虚拟机的时候再利用此处的数据恢复虚拟机的状态
宿主机状态保存区 (Host State Area)	当发生了#VMEXIT 事件时宿主机的状态利用此处数据恢复
虚拟机运行控制域 (VM Execution Control Fields)	此处数据定义了虚拟机在什么情况下发生#VMEXIT 事件，对 VMX non-root 模式有影响
VMEXIT 行为控制域 (VM Exit Control Fields)	此处数据定义了#VMEXIT 事件发生时要做的附加工作(比如保存调试寄存器，加载全局性能控制寄存器等等这些工作)
VMEntry 行为控制域 (VM Entry Control Fields)	此处数据定义了#VMEntry 事件时(通常是因为调用了 VMResume 汇编指令)要做的附加工作。
VMEXIT 相关信息域 (VM Exit Information Fields)	此处数据在发生#VMEXIT 事件时自动记录了发生原因和该事件的具体种类。这个域是只读的

VMX Abort 和 VMCS 数据区结构和用法会在后续章节中详细介绍。

VT 技术在设计时注明，没有任何标志位用于区分一个逻辑处理器 (Logical Processor) 当前正在执行 VMX root 模式下的指令还是执行的 VMX non-root 模式下的指令，这就确保了 Hypervisor 对虚拟机完全透明——因为虚拟机无从判断它当前是否运行在一个虚拟机下。最后要注意的是 VMCS 同样要求保证是在 4K 页对齐的连续物理内存空间中。

VT 中主要的指令有如下这些：

维护 VMCS 结构体的指令

- VMPTRLD 参数为 VMCS 块的物理地址。该指令用来激活一块 VMCS。修改该处理器的当前 VMCS 指针 (Current-VMCS Pointer) 指向传入的 VMCS 物理地址，并且激活该 VMCS，如果要维护一块 VMCS 则必须先激活该 VMCS。(否则不能用这些指令来维护)
- VMPTRST 用来存储当前 VMCS 指针 (VMCS 块物理地址) 到指定位置。
- VMCLEAR 该指令用来使一块 VMCS 变为不活跃状态。该指令将标记为已启动状态 (Launch State) 的 VMCS 设置为不活跃状态 (Inactive State/Clear State) 并且更新该 VMCS 块所有区域信息并确保写入 VMCS 块内存中 (这也就把对应虚拟机和 Hypervisor 的最新信息同时写入到 VMCS 块中)，如果带操作的 VMCS 块就是当前 VMCS 指针所指向的 VMCS 块，那么该指针会被设置为无效地址
- VMREAD 通过指定的 VMCS Encoding 从当前 VMCS 块中读取一个参数。
- VMWRITE 通过指定的 VMCS Encoding 从当前 VMCS 块中写入一个参数。

与虚拟机管理器有关的指令

- VMCALL 这条指令用于 Guest 和 Hypervisor 进行通信。执行该汇编指令会产生一个 #VMEXIT 事件，从而使得可以陷入 Hypervisor 中。
- VMLAUNCH 这条指令用于启动当前 VMCS 指针所指的一个虚拟机，并且移交控制权给 Guest。
- VMRESUME 这条指令用于从 Hypervisor 中恢复虚拟机的执行，并且移交控制权给 Guest。
- VMXOFF 这条指令用于关闭 Hypervisor。在下次执行 VMXON 开启 Hypervisor 前不

批注 [S8]: VMX Abort 和 VMCS 数据区结构和用法会在后续章节中详细介绍

得执行虚拟机相关汇编指令。

- **VMXON** 这条指令用于处理器进入VMX模式下，执行该指令后也就可以运行 Hypervisor。传入的参数必须是4K页对齐的物理地址，这段内存用于支持后续VMX相关的操作。

## VMLAUNCH 和 VMRESUME 指令的异同

VMLAUNCH 和 VMRESUME 命令都是将控制权移交到虚拟机，那么两者的区别呢？

两者运行的时机不同！

1. VMLAUNCH 指令会检查当前 VMCS 的启动状态是不是不活跃状态（相应标记位清空）。成功运行结果是该 VMCS 被标记为已启动状态。
2. VMRESUME 指令会检查当前 VMCS 的启动状态是不是已启动状态

所以，必须利用 VMLAUNCH 指令启动一个虚拟机。以后的某个时候，因为 VMEXIT 事件而陷入 Hypervisor 中，这个时候要恢复虚拟机的运行则要利用 VMRESUME 指令，正如图 1.5 所示。

管理VT相关的TLB的控制指令

- **INVEPT** 这条指令用于EPT地址翻译中，使TLB中缓存的地址映射失效
- **INVVPID** 这条指令用于在TLB中使某个VPID对应的地址映射失效

## 基于 VT 的 Hypervisor 开发逻辑

利用 VT 技术开发 Hypervisor 的过程不同于利用 SVM 技术的开发过程，最主要的差别是在 VT 技术中，Guest 和 Hypervisor 下面要运行的 IP 地址是可以在 VMCS 中设置的，同时 Hypervisor 就是用于处理 VMEXIT 事件，因此就像现代操作系统为系统调用设置一个统一入口，并将入口地址存入 MSR 寄存器一样，在 VT 中，通常也将 Hypervisor 的这个入口 IP 设置为事件处理函数入口地址（Event Dispatcher Address）。在事件处理的最后，又通过一个 VMXRESUME 指令统一的返回到 Guest 的指令执行流程中。对于事件发生信息，同样可以通过读取 VMCB 中相应数据获得。NewBluePill 中也有对 VT 技术的支持，我们同样会在深入探究 NewBluePill 的章节中详细展示怎样使用这些指令。

批注 [S9]: 后面要介绍 NBP 中关于 VT 技术的运用

## Intel-VTd(如果时间充裕则写)

### SVM 和 VT 不同之处和使用时应注意的地方

通过前文的描述，看上去 SVM 技术和 VT 技术十分相似，但是实际上两者还是有一些不同之处。在开发过程中必须注意到这些不同之处，它们是正确并且高效实现 Hypervisor 的关键。

SVM 的开发逻辑中，VMRUN 和事件处理程序要处于同一循环中，这是因为 Hypervisor

的事件处理程序入口在 VMRUN 的下一条地址上，而在 VT 技术中，由于可以自由指定这个入口地址，因此可以在 VMCS 块中指定一个函数作为事件处理入口函数。

SVM 采用 ASID 作为 TLB 中 Guest 和 Hypervisor 地址的标记，而 VT 采用 VPIDs (Virtual-Processor Identifiers) 作为 TLB 中不同虚拟机地址翻译的缓存标记，因此 VT 技术的缓存策略更精细所以更好些。

虽然 SVM 技术和 VT 技术都可以管理中断，管理资源，访问控制，但两者具体处理行为有一些差别，VT 技术将指令分为三种：无条件陷入的指令，有条件陷入的指令和不产生陷入的指令/事件。SVM 技术则分为了异常拦截和指令拦截，其中一些异常虽然会造成陷入，但是同时也会自动标记相应的异常寄存器 (Exception Specific Registers)。应用的时候一定要根据手册上的描述给出相应的实现。

SVM 和 VT 技术在使用时一定要注意，#VMEXIT 事件的产生来源于异常而不是行为，比如用户可以拦截 RDMSR 指令，但是发生的 sysenter 指令却不能拦截到，是因为在这种情况下，虽然 sysenter 有读取 MSR 寄存器的操作，但是因为没有提前在 VMCS/VMCB 中设定处理器遇到 sysenter 产生异常，所以处理器执行到 sysenter 指令当然也就不会产生 #VMEXIT 事件。

## NewBluePill 项目介绍

NewBluePill 项目 (<http://www.bluepillproject.org/>) 诞生于 2007 年，由 Invisible Things Lab 开发，现在对外公开的版本是 nbp-0.32-public 版本。该项目从 2007 年第三季度以来得到了 Phoenix 公司的大力支持。<sup>1</sup>

该项目目的是开发这样一种恶意软件：

- 不用已有的方法的隐藏自己
- 即使它的隐藏方法众所周知，其它软件也不能探测到它
- 即使它的实现代码众所周知，其它软件也不能探测到它

可以看出，该目的与非对称密码的设计目的有异曲同工之妙。该项目通过发掘 VT 技术和 SVM 技术平台漏洞来实现，当前在公开版本上已经实现的功能包括：

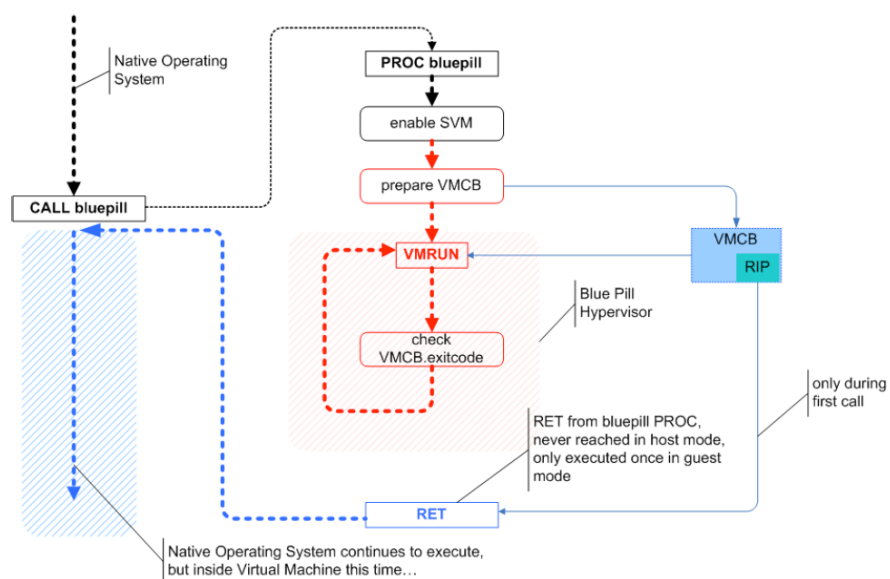
- 支持 SVM 和 VT-x 技术构建 Hypervisor
- 在操作系统运行时时刻动态加载和卸载，因此在操作系统完全不知情的情况下将操作系统放入了虚拟机中继续运行。
- 在 AMD 平台上支持嵌套 Hypervisor
- 一套自己的页表，用于实现内存隐藏，因此在操作系统中无法访问到 NewBluePill 的内存
- 反 Hypervisor 探测技术 (Anti-Hypervisor Detector)：RDTSC 欺骗
- 反 Hypervisor 探测技术 (Anti-Hypervisor Detector)：组织可信时间源的检测 (Blue-Chicken 技术)

在其未公开的版本上，实现的功能包括：

- 在 Intel VT-x 平台上实现嵌套 Hypervisor

这些特性最终使得即使 NewBluePill 后于操作系统启动，操作系统却完全无法感知到 NewBluePill 的存在，这也就是 NewBluePill 的主要设计目标 (如图 1.6 所示)。

<sup>1</sup> Phoenix 公司的虚拟化技术产品 HyperSpace，具体信息可以参考网上资料。与之类似的还有华硕公司 (Asus Inc.) 的 Instant On 技术

图 1.6 NewBluePill 的实现目标及思路<sup>1</sup>

## 版权信息

本书引用 NewBluePill 代码版权归属于 Invisible Things Lab

/\*

\* Copyright holder: Invisible Things Lab

\*

\* This software is protected by domestic and International

\* copyright laws. Any unauthorized use (including publishing and

\* distribution) of this software requires a valid license \* from the copyright holder.

\*

\* This software has been provided for the educational use

\* only during the Black Hat training and conference. This

\* software should not be used on production systems.

\*

\*/

<sup>1</sup> 本图来源 *Subverting Vista™ Kernel For Fun And Profit*, Joanna Rutkowska

## PART1 HEV 技术相关知识

### 二、深入 HEV 技术细节

在前一章中，我们简单介绍了 SVM 和 VT 技术，但是他们是如何被具体使用的呢？在本章中我们将详细介绍这些技术的细节：

- HEV 下虚拟机的启动过程
- VMEXIT 事件的陷入和处理
- 拆除 Hypervisor 和虚拟机
- EPT/NPT 翻译地址过程

直接阅读本章，可能会觉得理解其中内容却印象不深，推荐在阅读完全书后再次阅读本章——在理解了 NewBluePill 代码后，对本章内容自然会有更深的认识。

批注 [S10]: 本章写好后对这个列表更新

### HEV 下虚拟机启动过程

“物有本末,事有终始,知所先后,则近道矣”——《大学》

想要了解 HEV 技术的本质，则要了解 HEV 要解决的问题和怎样解决这些问题。要熟悉这些，就要沿着虚拟机开启——运行——关闭的过程，看 HEV 技术是怎样融入其中的。所以我们首先就来看看在 HEV 技术的帮助下，虚拟机是怎样启动的。

### 启动过程模型

首先介绍下有了 HEV 技术后，启动虚拟机的方式。使用了硬件虚拟化技术的虚拟机可以有三种引导 Guest 操作系统的方式：

1. 存在特殊 OS/Host OS，后启动 Hypervisor 的虚拟机启动过程
  2. 存在特殊 OS/Host OS，先启动 Hypervisor 的虚拟机启动过程
  3. 不存在特殊 OS/Host OS，先启动 Hypervisor 的虚拟机启动过程
- 存在特殊 OS/Host OS，后启动 Hypervisor 的虚拟机启动过程。采用这种启动过程的虚拟机代表是 KVM，其启动过程如下：
    - a) 先启动宿主 Linux 操作系统
    - b) 在 Linux 中启动 KVM 设备，从而启动了 Hypervisor
    - c) 启动虚拟机，作为 Linux 进程运行

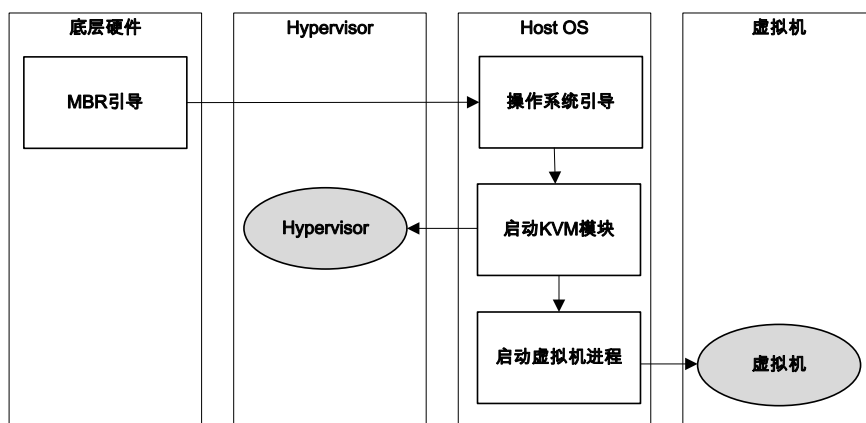


图 2.1 KVM 中虚拟机的启动过程

启动过程如图 2.1，可以看出，KVM 启动虚拟机的模式说明它不想脱离进程级虚拟机的本质，但是它要利用虚拟化技术进行加速。这样做的缺点在于需要一个 Host OS 充当载体。除 KVM 外，VMWare6.5 以上版本也是采用类似的架构，使用支持 HEV 技术的 CPU 进行加速。但是它们都需要再另外安装相应 Guest OS 上的驱动。

- 存在特殊 OS/Host OS，先启动 Hypervisor 的虚拟机启动过程。采用这种启动过程的虚拟机代表是 Xen，其启动过程如下：
  - a) 先创建并启动 Hypervisor
  - b) 引导 Dom0
  - c) 由 Hypervisor 和 Dom0 一起协作创建虚拟机
  - d) 启动该虚拟机<sup>1</sup>

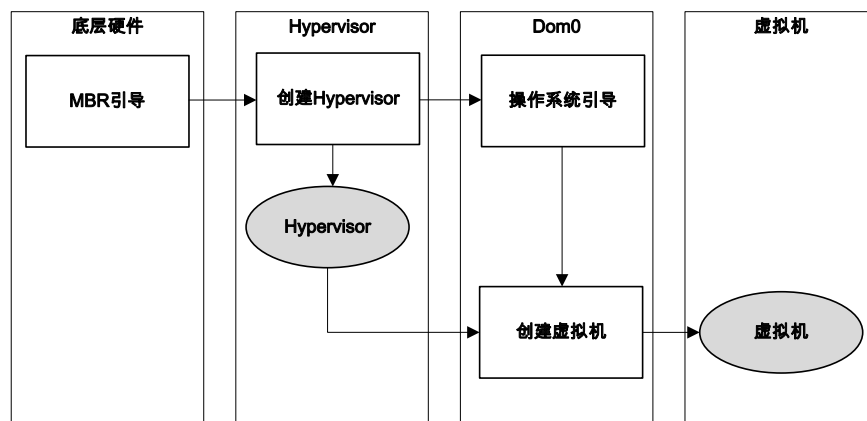


图 2.2 Xen 中虚拟机的启动过程

启动过程如图 2.2，可以看出，Xen 中仍存在 Dom0 是因为它要适应过去未出现 HEV 技术时的架构，所以无论是 Dom0 还是 Hypervisor 的实现都比较笨重，并且安装和配置也比较麻烦，同样需要另外安装相应 Guest OS 上的驱动。但是不可忽视的是

<sup>1</sup> Xen 中具体创建和启动虚拟机的过程会在“第 14 章 其它有关 HEV 项目”中介绍

Xen 的虚拟化效率最高。

- 不存在特殊 OS/Host OS，先启动 Hypervisor 的虚拟机启动过程。当前暂时没有采用这种启动过程的虚拟机软件（暂时称之为 UVM, Unknown Virtual Machine），其启动过程如下：

- a) 先创建并启动 Hypervisor
- b) 从 Hypervisor 中创建并启动虚拟机

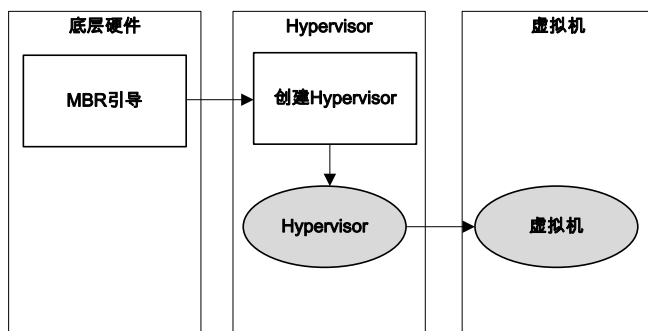


图 2.3 UVM 中虚拟机的启动过程

启动过程如图 2.3，这样的启动过程包括了如下组件：

表 2.1 UVM 模型下启动过程主要组件

组件	运行模式	作用
主引导扇区代码 (MBR)	16 位实保护模式	读取并加载活动分区启动扇区 (Active Partition's Boot Sector)
启动扇区 (Boot Sector)	16 位实保护模式	读取并运行磁盘上的 Hypervisor 创建程序
Hypervisor 创建程序	16 位实保护模式，虚拟机模式	创建并初始化 Hypervisor，并创建至少一个虚拟机
虚拟机引导程序	虚拟机模式	任何已有的 BIOS 初始化程序或操作系统的 MBR 引导程序，目的是初始化虚拟机

在于：不需要在 Guest OS 中安装任何支持驱动。换句话说，Hypervisor 对于 Guest OS 完全透明，从而实现完全虚拟化 (Full Virtualization)。这种方式的缺点是：Hypervisor 可能实现会很笨重，因而虚拟化效率不高，也会影响到系统安全，虚拟机的配置和管理可能也不易呈现给用户。

### 实验：阅读 Xen 和 KVM 的初始化部分代码

在 Xen 和 KVM 中，Hypervisor 的初始化代码都是用 C 编写的。阅读 Hypervisor 的初始化代码，对照图 2.1 和 2.2 体会其初始化的过程。



## VT 技术下开启虚拟机的过程

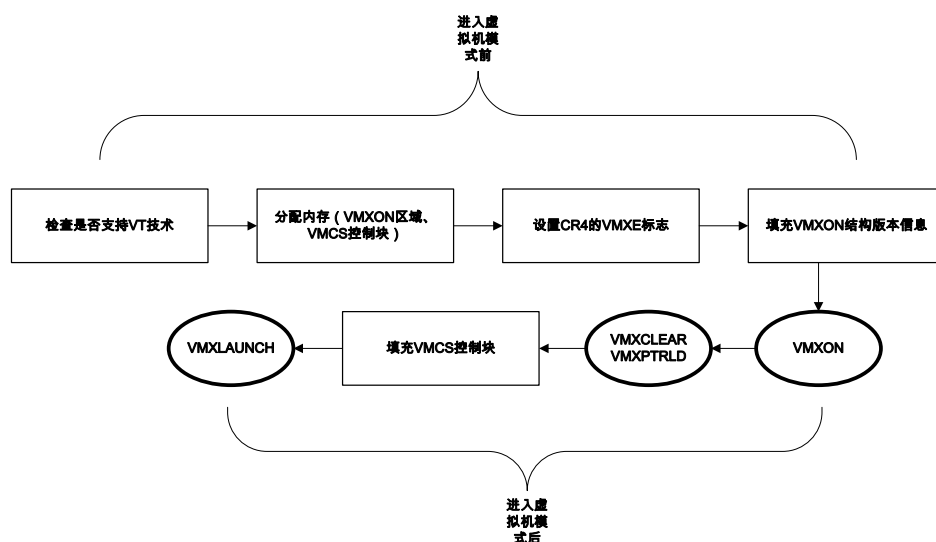


图 2.4 VT 技术下开启虚拟机的过程

在 VT 技术下进入虚拟机模式，开启虚拟机的全过程如图 2.4 所示。

首先检查当前 CPU 是否支持硬件虚拟化技术，这可以通过使用 `CPUID` 指令检查是否 `CPUID.1:ECX.VMX[bit 5]=1`（这句话表示操作数为 1 执行 `CPUID` 指令，检查返回的 `ECX` 寄存器 bit 5 位，也就是 `VMX` 位，查看结果是否为 1，下文均用这种表示法）。

开启虚拟机前，必须设置 `CR4.VMXE[bit 13]=1`，并且在内存中分配出来 `VMXON` 区域（`VMXON Region`）和 `VMCS` 控制块，后者也可以在进入虚拟机模式后再分配。需要注意的是，他们两者都必须分配在 4K 页对齐的内存区域上。

然后还需要初始化 `VMXON` 区域，需要把 `VMCS` 的版本号表示符（`VMCS Revision Identifier`）写入 `VMXON` 区域当中，该版本号可以通过访问 `IA32_VMX_BASIC` MSR 寄存器获得。除此以外不需要任何其它操作。只是要注意的是，在 `VMXON` 和 `VMXOFF` 指令之间的代码区中不要访问或者修改这个 `VMXON` 区域。

这之后通过执行 `VMXON` 指令即可以进入虚拟机模式。要注意的是，如果在执行 `VMXON` 指令时发现 `CR4.VMXE=0`，那么 `VMXON` 指令会发生无效操作数的异常（`#UD`<sup>1</sup>）。最后，一旦进入虚拟机模式，`CR4` 和 `CR0` 寄存器中的一些与虚拟机模式相关的位将无法设置。

### 关于 `VMXON` 指令（`VMXON Instruction`）

`VMXON` 指令能否执行，同样也受 `IA32_FEATURE_CONTROL_MSR` 寄存器控制：

■ Bit 0 是置锁位（Lock Bit） 如果该位为 0，那么 `VMXON` 指令不能执行；如果该位为 1，那么 `WRMSR`（写 MSR 寄存器指令）不能去写这个寄存器。该位在系统上电后便不能

<sup>1</sup> 关于异常的说明，请参考 *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1* 中 Chapter 6.4 Interrupts and Exceptions 一节。

修改。BIOS 通过修改这个寄存器来设置是否支持虚拟化操作。在支持虚拟化操作的情况下，BIOS 还要设置 Bit 1 和 Bit 2

■ Bit 1 指示 VMXON 指令能否在 SMX<sup>1</sup>操作环境中执行 如果这一位为 0，那么 VMXON 指令不能在 SMX 操作环境中执行

■ Bit 2 指示 VMXON 指令能否在 SMX<sup>2</sup>操作环境外执行 如果这一位为 0，那么 VMXON 指令不能在 SMX 操作系统外执行

如图 2.4 所指出的那样，执行 VMXON 指令后，我们需要执行 VMCLEAR 指令来初始化 VMCS 控制块。在第一章中我们介绍了 VMCS 区域的组成部分，在使用中，处理器会使用 VMCS 数据区来维护 VMCS 版本特定信息（implementation-specific information），VMCLEAR 指令会初始化这些信息，VT 技术推荐先执行 VMCLEAR 指令从而设置 VMCS 启动状态为空状态（clear），再执行 VMPTRLD 指令激活该 VMCS 块。

### 关于 VMCS 的启动状态

在第一章中我们在介绍了 VMLAUNCH 指令和 VMRESUME 指令的区别，里面同样涉及到 VMCS 的启动状态。

实际上，一个 VMCS 块的启动状态包含空状态（Clear）和已启动状态（Launched）两种状态，VMCLEAR 指令会把指定的 VMCS 块置为空状态，而 VMXRESUME 和 VMXLAUNCH 两个指令会根据该状态的进行相应的操作。

要注意的是，通过 VMWRITE 指令是不能修改这个状态的。而且在转移一个 VMCS 块到另一个逻辑处理器上时，需要先使用 VMCLEAR 指令设置启动状态为空状态，再用 VMLAUNCH 启动该 VMCS 块所代表的虚拟机。因此尽量避免在不同逻辑处理器间转移执行 VMCS 块的操作。

利用 VMPTRLD 加载 VMCS 块之后，需要开发者按照需要配置 VMCS 块具体内容，VMCS 相关具体内容，会在本章稍后详加解释。

当这一切都设置好以后，利用 VMXLAUNCH 指令启动该 VMCS 块所代表虚拟机，至此虚拟机和 Hypervisor 均已初始化完毕，并能够成功开启虚拟机。

### SVM 技术下开启虚拟机的过程

批注 [S11]: 阅读 AMD 手册补充这部分

### HEV 下虚拟机关键结构体

VT 下的 VMCS 结构体和 SVM 下的 VMCB 结构体在开启虚拟机的过程中起着至关重要的作用，这一节我们就将深入探索 VMCS、VMCB 结构体的细节。

<sup>1</sup> SMX（安全扩展模式，Safer Mode Extensions）

<sup>2</sup> SMX（安全扩展模式，Safer Mode Extensions）

## VT 技术下的 VMCS 结构体

在上一章中我们提到，VMCS 区域由 VMCS 版本标志（Revision Identifier）、VMX 退出原因指示器（VMX-abort indicator）、VMCS 数据区三部分构成。

在使用 VMCS 区域前，应当首先设置 VMCS 版本标志，因为文档中说明，这个域是由软件负责设置的，并且 VMPTRLD 指令在执行时会检查该 VMCS 块设置的版本标志与目标处理器能够接受的 VMCS 版本是否相符，不相符则会抛出异常。通常软件通过读取 IA32\_VMX\_BASIC MSR 寄存器来设置 VMCS 版本标志，因为这个 MSR 寄存器存有该处理器能够接受的 VMCS 版本标志。

VMX 退出原因指示器的产生是因为在 VM Exit 事件的时候如果遇到问题，系统就会发生 VMX Abort 事件，这会导致该逻辑处理器进入关闭状态。对于一个激活了的 VMCS，一个 VMX Abort 事件的发生并不会修改 VMCS 数据区，因此我们需要一种机制去判断是什么原因导致了 VMX Abort 事件的发生。通常，造成 VMX Abort 事件的原因包括：保存客户虚拟机 MSR 寄存器失败、当前 VMCS 区域损坏、加载 Hypervisor 的 MSR 寄存器失败等等。

VMCS 数据区构成了 VMCS 区域的主体，第一章中我们介绍了 VMCS 数据区的六个主要组成部分，下面我们将逐一介绍这六个主要部分又是怎样构成的。

批注 [S12]: 该图在本节写好后会被替换

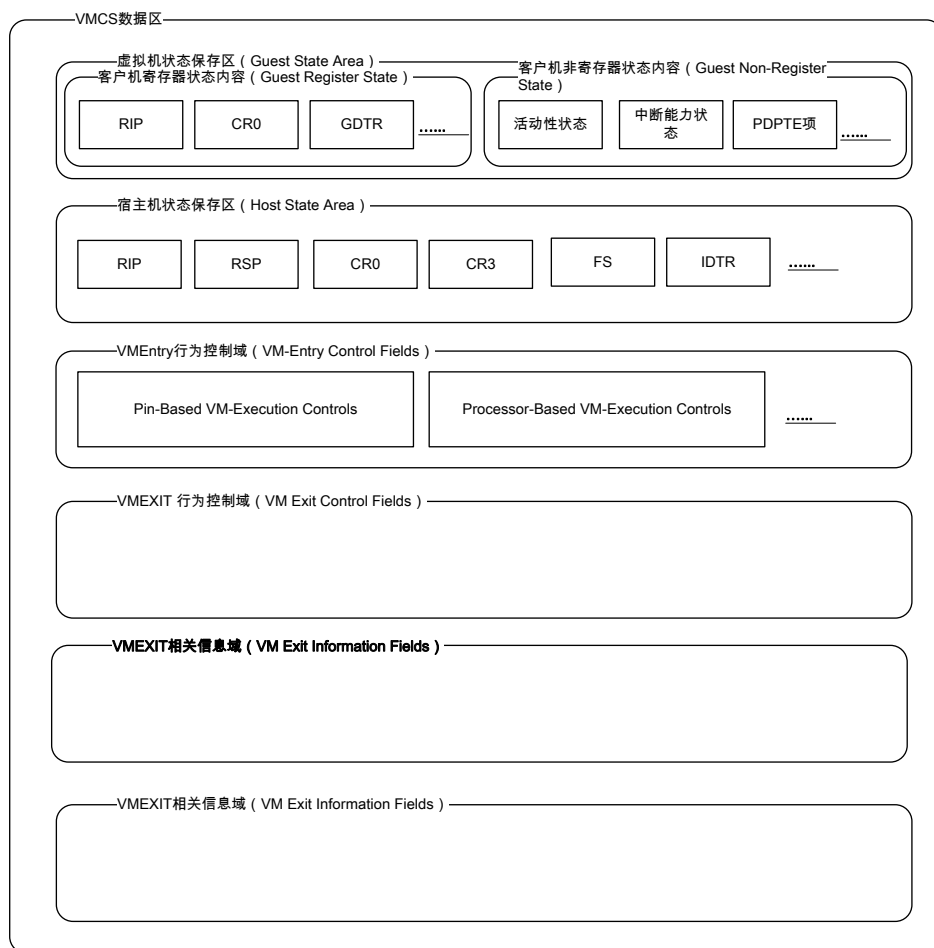


图 2.5 VMCS 数据区结构图

## 虚拟机状态保存区 (Guest-State Area)

这个区域保存了用于描述客户虚拟机状态的寄存器，当发生#VMEXIT 事件的时候，虚拟机的状态会自动保存到这些域当中，而当发生#VMENTRY 事件的时候，这些域中的值又被用来恢复虚拟机的运行状态。

这个区域可以保存的内容分为客户机寄存器状态内容 (Guest Register State) 和客户机非寄存器状态内容 (Guest Non-Register State) 两部分。客户机寄存器状态内容 (x86 上分别对应各自 x86 寄存器)：

- 1) 控制寄存器 CR0, CR3, CR4
- 2) 调试寄存器 DR7
- 3) RSP, RIP 和状态寄存器 RFLAGS
- 4) CS, SS, DS, ES, FS, GS, LDTR 和 TR 寄存器的下面各项

- 选择子 (Selector)
  - 基址 (Base Address)
  - 段长 (Segment limit)
  - 访问权限 (Access Rights)
- 5) GDTR 和 IDTR 信息
- 基址 (Base Address)
  - 段长 (Segment limit)
- 6) 一些 MSR 寄存器, 包括 IA32\_DEBUGCTL, IA32\_SYSENTER\_CS, IA32\_SYSENTER\_ESP, IA32\_SYSENTER\_EIP, IA32\_PERF\_GLOBAL\_CTRL, IA32\_PAT, IA32\_EFER

客户非寄存器状态内容包括:

- 1) 活动性状态 (Activity State) 这项表明了当前逻辑处理器的活动性, 也就是能否正常处理客户机程序指令, 包含 4 个状态:
  - a) 活跃的 (Active) 说明当前逻辑处理器可以执行客户机指令
  - b) 中断的 (HLT) 当前逻辑处理器不能执行客户机指令, 因为执行了一条 HLT 指令。
  - c) 关闭的 (Shutdown) 当前逻辑处理器不能执行客户机指令, 因为发生了严重的错误。
  - d) 等待 SIPI 中断 (Wait-for-SIPI) 当前逻辑处理器不能执行客户机指令, 因为在等待 Startup-IPI 中断<sup>1</sup>
- 2) 中断能力状态 (Interruptibility State) x86 架构支持某些事件能被阻塞一段时间的特性, 包括被 STI 开中断指令阻塞, 被 MOV SS 阻塞, 被 SMI 中断阻塞和被 NMI 中断阻塞。这个域就包含了对应的描述信息。
- 3) 推迟调试的异常 (Pending Debug Exceptions) x86 支持延迟发送一些调试异常, 如某些情况下的单步调试。这个域就包含了对应的描述信息。
- 4) VMCS 连接指针 (VMCS Link Pointer) 该域保留, 用于未来扩展。
- 5) VMX 抢占计时器值 (VMX-Preemption Timer Value) 该域保存了虚拟机的 VMX 抢占计时器计数值。
- 6) PDPTE 项 (Page Directory Pointer Table Entries) 虚拟机状态保存区只有在虚拟机运行控制域中开启 EPT 模式才会用到此域, 其中包括 4 个 64 位数据: PDPTE0~PDPTE3。

### 关于 VMX 抢占计时器 (VMX-Preemption Timer)

VMX 抢占计时器是 VT 技术中这样的一种特性, 如果在 #VMEntry 事件发生后, 处理器硬件发现在虚拟机运行控制域中“启用 VMX 抢占计时 (Activate VMX-Preemption Timer)”被设置, 那么交由虚拟机执行指令时, 将启用一个 **VMX 抢占计时器**, 该计时器会倒数, 当倒数为 0 时会触发一个 #VMEXIT 事件陷入 Hypervisor 中。

如果“启用 VMX 抢占计时 (Activate VMX-Preemption Timer)”被设置, 同时 VMX 抢占计时器值为 0, 那么 #VMEXIT 事件会在 #VMENTRY 事件发生后执行任何指令前发生 (但是如果有推迟调试的异常 (Pending Debug Exceptions), 它们将先于 #VMEXIT 事件得到处理)。

VMX 抢占计时器的计时间隔 (Timer Interval) 是可以通过 IA32\_VMX\_MISC 寄存器设置

<sup>1</sup> SIPI (Startup Inter-Processor Interrupt), 用于多核平台初始化其它处理器。具体可参考 *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B* 手册

的 (0~31)，且为 2 的整数次幂于 TSC 寄存器值的变化。比如，如果我们设置该寄存器内容为 5，那么表示 TSC 寄存器中的值每增加 32，VMX 抢占计时器计数减 1。

但是要注意，由于节能模式的关系，所以当逻辑处理器在 C-states 的 C0, C1, C2 之外的其它状态时，VMX 抢占计时器不会产生 #VMEXIT 事件<sup>1</sup>

对于系统管理中断 (SMIs, System Management Interrupts) 和系统管理模式 (SMM, System Management Mode)，VMX 抢占计时器的行为依赖于 Hypervisor 是否要处理这个模式：

- 默认情况下，VMX 抢占计时器在虚拟机处理系统管理中断时，也会继续倒计时时，只不过在倒数为 0 的情况下，VMX 抢占计时器要继续等到虚拟机脱离系统管理模式后再产生 #VMEXIT 事件。

- 如果 Hypervisor 的设定指出同样要监管系统管理中断和系统管理模式时，这个时候 VMX 抢占计时器的行为和在处理一般 VMEXIT 事件、VMEntry 事件时一样。

## 宿主机 (Hypervisor) 状态保存区 (Host-State Area)

这个区域记录了所有有关 Hypervisor 的状态信息，正如第一章中所提到的，这个区域保存的内容会在每次发生 #VMEXIT 事件时恢复到相应的寄存器中，以恢复 Hypervisor 的执行环境。

与虚拟机状态保存区不同，宿主机状态保存区只能存储有关寄存器的信息：

- 1) 控制寄存器 CR0, CR3, CR4
- 2) RSP, RIP
- 3) CS, SS, DS, ES, FS, GS 和 TR 寄存器的下面各项
  - 选择子 (Selector)
- 4) FS, GS, TR, GDTR 和 IDTR 信息
  - 基址 (Base Address)
- 5) 一些 MSR 寄存器，包括 IA32\_SYSENTER\_CS, IA32\_SYSENTER\_ESP, IA32\_SYSENTER\_EIP, IA32\_PERF\_GLOBAL\_CTRL, IA32\_PAT, IA32\_EFER

## 虚拟机运行控制域 (VM-Execution Control Fields)

虚拟机运行控制域管理着虚拟机的运行，包含下列域：

- 1) 基于针脚的虚拟机执行控制 (Pin-Based VM-Execution Controls)

这个域用来管理中断等异步事件 (Asynchronous Event)<sup>2</sup>，如“启用抢占计时”就是在这个域中设置的。对于其中保留位的设置，软件要参考 IA32\_VMX\_PINBASED\_CTLs 和 IA32\_VMX\_TRUE\_PINBASED\_CTLs 两个 MSR 寄存器的内容。

- 2) 基于处理器的虚拟机执行控制 (Processor-Based VM-Execution Controls)

该域包含两个 4 字节值，用于管理执行特定指令而产生的同步事件 (Synchronous Event)。这个控制域分为主要基于处理器的虚拟机执行控制和次要基于处理器的虚拟机执行控制两个，前者包括对 HLT、INVLPG、MWAIT、RDPMSR、RDTSC、CR3 读取/存储、

<sup>1</sup> 有关节能模式和 C-states 的详细信息，可以参考网上相关资料

<sup>2</sup> 有些中断不受该域控制，触发 VMEXIT 事件

使用 I/O Bitmap 等等这些产生 VMEXIT 事件的控制；后者包括对开启 EPT 地址翻译、开启 VPID<sup>1</sup>、开启虚拟 APIC（高级可编程中断控制器，关于此部分的介绍，可参考随后“关于可编程中断控制器”部分）等等的控制。对于其中保留位的设置，前者要参考 IA32\_VMX\_PROCBASED\_CTLs 和 IA32\_VMX\_TRUE\_PROCBASED\_CTLs 两个 MSR 寄存器的内容，后者要参考 IA32\_VMX\_PROCBASED\_CTLs2 MSR 寄存器。

### 3) 异常位图（Exception Bitmap）

该域仅有 32 位长，每一位代表当某种异常发生时，硬件会自动产生 VMEXIT 事件；如果某一位为 0，则表示这个异常会通过 IDT 表正常处理。这其中要注意的是缺页异常的处理略有不同，需要借助于 VMCS 中另外的两个域（Page Fault Error Code Mask 和 Page Fault Error Code Match）来处理。

### 4) I/O 位图地址（I/O Bitmap Addresses）

该域包含两个 64 位长的物理地址，指向两块 I/O 位图，只有在 Primary Processor-Based VM-Execution Controls.Use I/O Bitmaps[bit 25]=1 的情况下才会被使用。逻辑处理器在处理 I/O 指令时，会根据这些 I/O 位图而在访问相应地址时产生 VMEXIT 事件。VT 技术要求这些位图必须在 4K 页对齐的位置上。

### 5) 时间戳寄存器偏移值（Time-Stamp Counter Offset）

虚拟机运行控制域还包括一个时间戳寄存器偏移值域，这个域在 Primary Processor-Based VM-Execution Controls.RDTSC Exiting[bit 12]=0 且 Primary Processor-Based VM-Execution Controls.Use TSC Offsetting[bit 3]=1 时起作用。当客户机利用 RDTSC、RDTSCP 指令或者访问 IA32\_TIME\_STAMP\_COUNTER MSR 寄存器的时候，得到的结果将是真实的值加上这个偏移量的和。

### 6) 虚拟机/Hypervisor 屏蔽和 CR0/CR4 访问隐藏设置（Guest/Host Masks and Read Shadows for CR0 and CR4）

这个域主要对 CR0 和 CR4 寄存器进行保护。虚拟机/Hypervisor 屏蔽中置位 1 的位说明 CR0 和 CR4 寄存器的相应位只能由 Hypervisor 修改，否则产生 VMEXIT 事件，置位 0 的部分说明 CR0 和 CR4 的相应位可以在虚拟机中修改。

### 7) CR3 访问目标控制（CR3-Targeting Controls）

包含最多 4 个 CR3 目标值<sup>2</sup>，当在虚拟机中执行 CR3 的赋值操作时，如果赋予的值是这些目标值中任意一个，那么硬件不会产生 VMEXIT 事件。

### 8) APIC 访问控制（Controls for APIC Accesses）

访问本地 APIC 的寄存器有三种方法：通过 xAPIC 模式访问、通过 x2APIC 模式访问、在 64 位模式下通过 mov CR8 指令来访问任务优先级寄存器（Task Priority Register, TPR）<sup>3</sup>。APIC 的访问控制实际上也是 VT 技术对 APIC 的虚拟化，在这个域中，通过配置“使用影子 TPR（Use TPR Shadow）”、“虚拟化 APIC 访问（Virtualize APIC Accesses）”、“虚拟

<sup>1</sup> 开启 VPID (Virtual-Processor Identifier)意味着缓存在翻译线性地址时会根据 VPID 进行翻译，其优点已在第一章中介绍。

<sup>2</sup> 未来可能在此处容纳更多的 CR3 目标值，因此在使用前参考 IA32\_VMX\_MISC MSR 寄存器来查看当前处理器详细信息。

<sup>3</sup> 请参考 Intel 手册相关部分以获得有关三种 APIC 访问方法的详细资料。

化 x2APIC 模式 (Virtualize x2APIC Mode)”对 APIC 进行虚拟访问。

#### 9) MSR 位图地址 (MSR-Bitmap Address)

该域包含一个指向 MSR 位图区域的物理地址, 当 Primary Processor-Based VM-Execution Controls.Use MSR Bitmaps[bit 28]=1 时会被使用。MSR 位图区域占据 4K 大小内存, 分为 4 个连续区域: 读低地址/高地址 MSR、写低地址/高地址 MSR。根据配置, 当相应的 MSR 访问执行时, 会发生 VMEXIT 事件。

#### 10) 执行体 VMCS 指针 (Executive-VMCS Pointer)

该域 64 位长, 用于 VT 技术对系统管理中断 (SMI) 和系统管理模式 (SMM) 进行监管 (也称 Dual-Monitor Treatment)

#### 11) EPT 指针 (Extended Page Table Pointer)

该域包含了 EPT 页表的基地址 (也就是指向 EPML4 级页表的物理地址), 以及一些 EPT 页表的配置信息, 当 Secondary Processor-Based VM-Execution Controls.Enable EPT[bit 1]=1 时启用。

#### 12) 虚拟机标示符 (Virtual Processor Identifier, VPID)

虚拟机表示符定义为 16 位长, 当 Secondary Processor-Based VM-Execution Controls.Enable EPT[bit 5]=1 时启用。

### 关于可编程中断控制器

#### ■ x86 上的中断控制器

在大多数很古老的 x86 系统上都有一个 i8259A 可编程中断控制器 (Programmable Interrupt Controller, PIC), 或者新一些的, i82489 高级可编程中断控制器 (Advanced Programmable Interrupt Controller, APIC)。APIC 兼容 PIC, 前者拥有 256 条中断线, 而后者仅拥有 15 条中断线; 前者支持多核技术, 而后者并不支持。APIC 架构图如图 2.6 所示

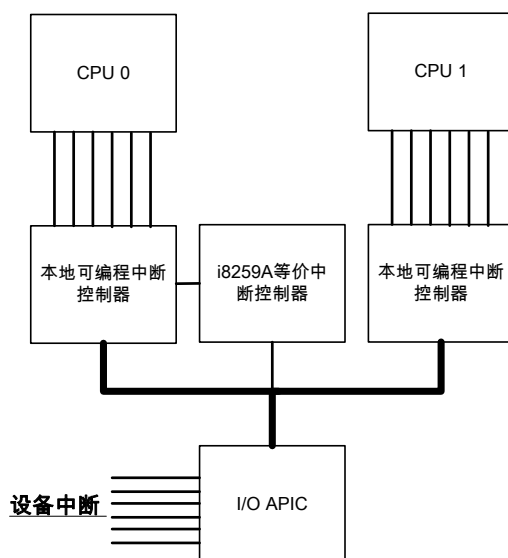




图 2.6 可编程中断控制器 (APIC) 架构图

APIC 包括下列组成部分:

- ◆ I/O APIC 用于从设备接收中断
- ◆ 本地可编程中断控制器 (Local APIC) 本地中断控制器利用一条私有总线, 从 I/O APIC 上接收中断, 然后向与其关联的 CPU 发送中断。
- ◆ i8259A 等价中断控制器 (i8259A Equivalent PIC) 负责把 APIC 的输入信号翻译成 PIC 等价的信号, 用于实现在 APIC 上对 PIC 的兼容

APIC 私有总线也要负责决定要把该中断信号发送到哪个 Local APIC 上 (Windows 有权决定是否要利用 I/O APIC 的这个算法), 这样做可以更好的利用处理器本地性。

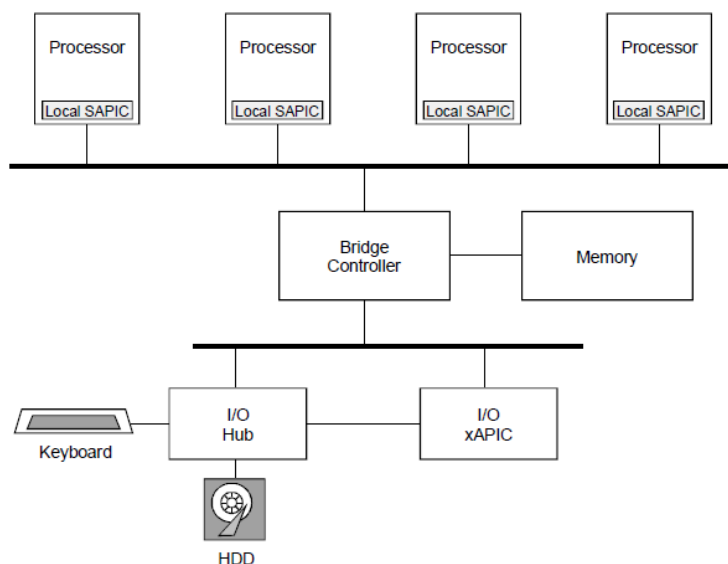
Local APIC 的另外一个功能是发送处理器间中断 (Inter-Processor Interrupts, IPI) 到其它的 Local APIC 上, 考虑下列情况: 每当一个时钟中断发生的时候, 在多核平台上, 显然只应该有一个处理器去负责更新系统时间, 否则系统时间将会在一次时钟中断下被多次更新。这个时候就需要 IPI 的帮忙了。其它情况比如更新 TLB Cache、在 DISPATCH\_LEVEL 中断级别下调度一个线程、系统崩溃、系统关闭时均需要 IPI 中断的帮忙。

#### ■ x64 上的中断控制器

因为 x64 平台要兼容于 x86 平台, 所以 x64 平台上的中断控制器与 x86 平台的相同。但是 x64 平台上的 Windows 要求一定要有 APIC 作为中断控制器。

#### ■ IA64 上的中断控制器

IA64 平台使用 SAPIC (Streamlined Advanced Programmable Interrupt Controller) 作为其中断控制器。SAPIC 的 I/O APIC 不再使用一条私有总线传输中断, 而是利用系统总线, 进而通过北桥控制器来传输中断 (见图 2.7)。这样做的好处是能够更快的传输中断。

图 2.7 SAPIC 架构图<sup>1</sup>

<sup>1</sup>此图摘自 Intel Itanium Processor Family Interrupt Architecture Guide

另一个不同之处在于中断分发的路由算法上，由于 SAPIC 没有了私有 APIC 总线，所以中断路由算法就要写入固件（Firmware）当中。与 x86 中的情况类似，Windows 可以决定是否要使用这个算法。

### 实验：查看本机上的 APIC 配置

在多核机器上，可以在 windbg 中利用 !apic 命令查看本机 APIC 配置，“0: kd”表明当前运行在 CPU 0 上，m 表明当前中断被屏蔽。

```
0: kd> !apic
Apic @ fffe0000 ID:0 (40011) LogDesc:01000000 DestFmt:ffffff TPR FF
TimeCnt: 03f07410clk SpurVec:1f FaultVec:e3 error:0
Ipi Cmd: 02000000`000008e1 Vec:E1 FixedDel Lg:02000000 edg high
Timer..: 00000000`000300fd Vec:FD FixedDel Dest=Self edg high m
Linti0.: 00000000`0001001f Vec:1F FixedDel Dest=Self edg high m
Linti1.: 00000000`000004ff Vec:FF NMI Dest=Self edg high
TMR: 63, 83, B1
IRR: B1, D1
ISR: D1
```

**Note** 阅读 *Intel Itanium Processor Family Interrupt Architecture Guide* 文档可以获得关于 IA64 SAPIC 的更详细的指导信息

## VMEntry 行为控制域（VM-Entry Control Fields）

VMEntry 行为控制域定义了 VMEntry 事件发生后硬件要立即做的事情，主要包括三部分：VMEntry 基本操作控制设置（VM Entry Controls）、VMEntry MSR 寄存器操作控制设置（VM Entry Controls For MSRs）和 VMEntry 注入事件控制设置（VM Entry Controls for Event Injection）。

VMEntry 基本操作控制设置包括：

- 1) 加载调试寄存器内容
  - DR7
  - IA32\_DEBUGCTL MSR 寄存器
- 2) 虚拟机是否进入 x64 支持模式（x86 架构上永远为 0）
- 3) 进入系统管理模式（SMM）
- 4) 关闭 Dual-Monitor Treatment
- 5) 加载 IA32\_PERF\_GLOBAL\_CTRL MSR 寄存器
- 6) 加载 IA32\_PAT MSR 寄存器
- 7) 加载 IA32\_EFER MSR 寄存器

对于其它保留位的设置，软件必须根据 IA32\_VMX\_ENTRY\_CTLS MSR 寄存器和 IA32\_VMX\_TRUE\_ENTRY\_CTLS MSR 寄存器内容设置。

对于 VMEntry MSR 寄存器操作控制设置，细心的读者可能发现，前面我们在描述客户机状态域时提到过，硬件会在发生 VMEntry 事件后自动加载一些有关虚拟机状态 MSR 寄存器。这里 VMEntry MSR 寄存器操作控制设置允许开发人员恢复更多的 MSR 寄存器，主要通过下面两个域配置：

- 1) VMEntry MSR 寄存器加载数量 (VMEntry MSR-Load Count)
- 2) VMEntry MSR 寄存器加载地址 (VMEntry MSR-Load Address)

VMEntry 事件可以在所有虚拟机状态恢复完毕后，通过客户机 IDT 表触发一个中断。VMEntry 注入事件控制设置就是用来配置这个特性的，它主要有如下三个可配置部分：

- 1) VMEntry 中断信息域 (VM Entry Interruption Information Field)
- 2) VMEntry 异常错误码 (VM Entry Exception Error Code)
- 3) VMEntry 指令长度 (VMEntry Instruction Code)<sup>1</sup>

## VMEXIT 行为控制域 (VM-Exit Control Fields)

VMEXIT 行为控制域定义了 VMEXIT 事件发生后硬件要立即做的事情，主要包括两部分：VMEXIT 基本操作控制设置 (VM Exit Controls) 和 VMEXIT MSR 寄存器操作控制设置 (VM Exit Controls For MSRs)。

VMEXIT 基本操作控制设置包括：

- 1) 保存调试寄存器内容
  - DR7
  - IA32\_DEBUGCTL MSR 寄存器
- 2) Hypervisor 地址空间大小 (x86 架构上永远为 0)
- 3) 加载 IA32\_PERF\_GLOBAL\_CTRL MSR 寄存器
- 4) VMEXIT 后发出中断 (Acknowledge Interrupt on Exit)
- 5) 保存 IA32\_PAT MSR 寄存器
- 6) 加载 IA32\_PAT MSR 寄存器
- 7) 保存 IA32\_EFER MSR 寄存器
- 8) 加载 IA32\_EFER MSR 寄存器
- 9) 保存 VMX 抢占计时器值

VMEXIT MSR 寄存器操作控制设置类似于上文中的 VMEntry MSR 寄存器操作控制设置，它允许开发人员在 VMEXIT 事件发生后恢复更多有关 Hypervisor 状态的 MSR 寄存器，主要通过下面两个域实现：

- 1) VMEXIT MSR 寄存器加载数量 (VM-Exit MSR-Load Count)
- 2) VMEXIT MSR 寄存器加载地址 (VM-Exit MSR-Load Address)

<sup>1</sup> 对于软中断 (Software Interrupt)、软件异常 (Software Exception) 和特权软件异常 (Privileged Software Exception)，这个域用来决定填充到异常堆栈上的 RIP 地址指针值。

## VMEXIT 相关信息域 (VM Exit Information Fields)

### SVM 技术下的 VMCB 结构体

VMEXIT 事件的陷入和处理

什么拦截的到什么拦截不到

有条件陷入、无条件陷入

SMM 和 Hypervisor

### HEV 下虚拟机关闭过程

### VT 技术下关闭 Hypervisor 和虚拟机的过程

通过在 Hypervisor 下执行 VMXOFF 指令来关闭虚拟机模式，然后可以清除 CR4.VMXE 的值。

### SVM 技术下关闭 Hypervisor 和虚拟机的过程

怎样拆除 Hypervisor 和虚拟机

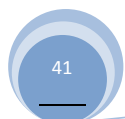
怎样将正在运行的虚拟机退回到保护模式

EPT/NPT (Chapter 24)

Vmxon Region, VMCS, I/O 位图, Msr 位图

AMD Nested Page Table 详细技术细节 AMD 手册 P406

**Note** 关于此章内容更详细的信息，请参考 *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B* 和 *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*, Chapter 15 Secure Virtual Machine



## PART2 深入研究 NewBluePill

### 三、 体验 NewBluePill

首先介绍下我的平台，在整个项目中我用了两台计算机

PC1（调试机）：Intel Core 2 6300, 1G RAM, XP SP2(X32)+windbg+WDK6001.18001

PC2(被调试机)：Intel Core 2 6300, 1G RAM, Windows Server 2008 Beta 1(X64), NewBluePill  
只能运行于这台机器上。

### 编译 NewBluePill

了解了以上那么多，是不是很想亲自动手尝试下呢？不过先别急，还是先把工具准备好再说。

工具一共有下面几个：

1. Windbg
2. DebugView<sup>1</sup>
3. InstDrv<sup>2</sup>
4. Windows Driver Kits (WDK 6001.18001)

总体来说编译 NewBluePill 的过程很简单。

步骤 1. 首先确保手上有 `nbp-0.32-public.zip` 这个代码<sup>3</sup>。然后解压缩到一个根目录，在这里我们假设是 D 盘。目录结构应该是这样的：

<sup>1</sup> DebugView，可以到 <http://download.sysinternals.com/Files/DebugView.zip> 下载

<sup>2</sup> InstDrv，可以到 <http://dl2.csdn.net/fd.php?i=23314208212665&s=0affa2ecb56fc0dcc14cff07345a388e> 下载

<sup>3</sup> NewBluePill 项目源代码可以从 <http://www.bluepillproject.org/> 下载

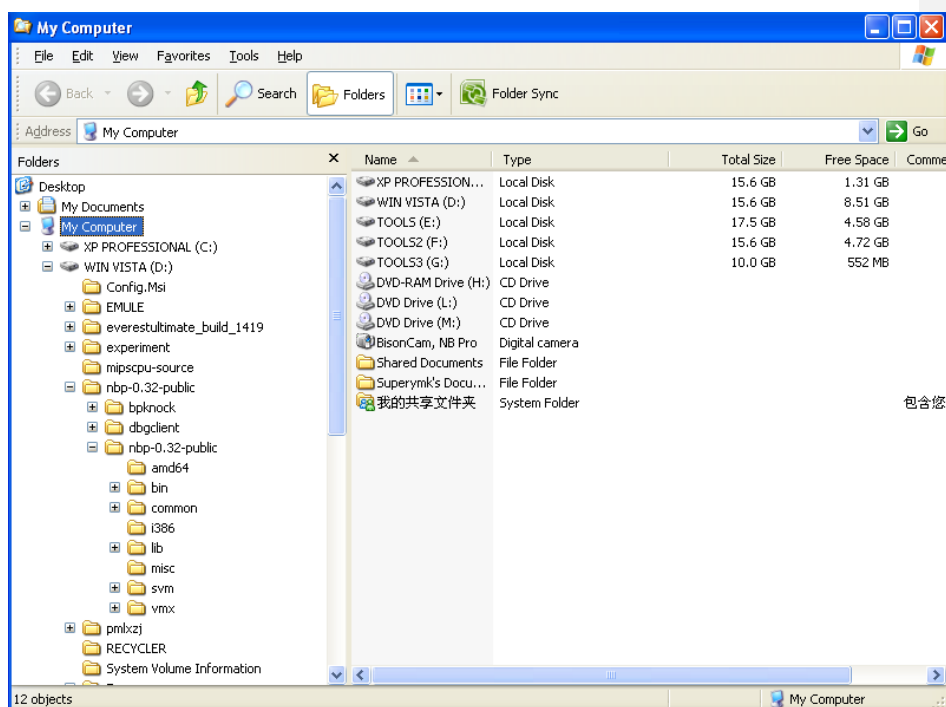


图 3.1 NewBluePill 项目目录结构

步骤 2. 然后打开 Launch Windows Vista and Windows Server 2008 x64 Checked Build Environment 编译环境:

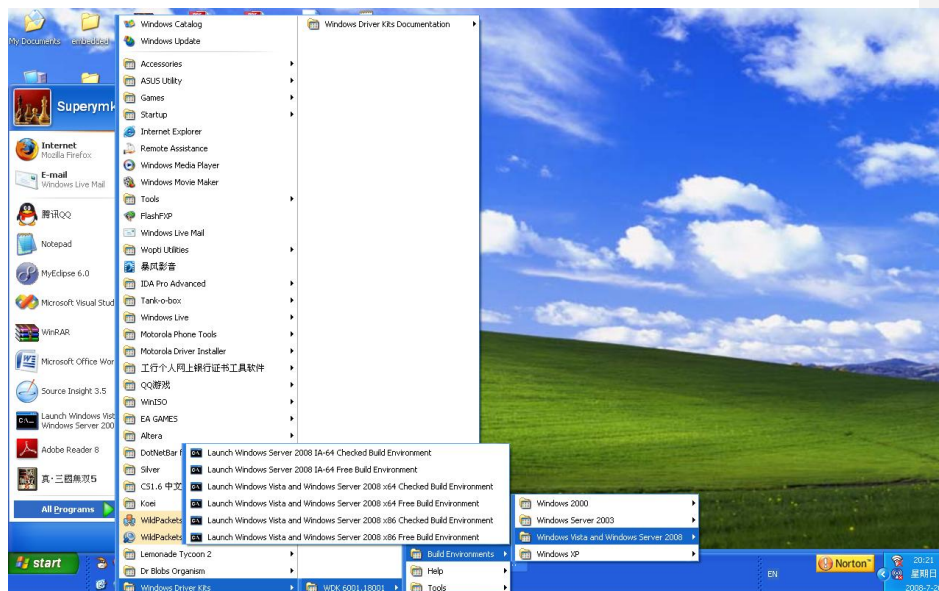


图 3.2 WinDDK 编译环境快捷方式的位置

步骤 3. 在该编译环境中执行 `nbp-0.32-public\NewBluePill-0.32-public\build_code.cmd`,



如果编译成功则会出现以下窗口：

```

1>Compiling - vmx\vmxdebug.c
2>Building Library - lib\amd64\svm.lib
1>Building Library - lib\amd64\vmx.lib
1>BUILD: Compiling and Linking d:\nbp-0.32-public\nbp-0.32-public\common directory
1>Assembling - amd64\msr.asm
1>Assembling - amd64\svm-asm.asm
1>Assembling - amd64\vmx-asm.asm
1>Assembling - amd64\common-asm.asm
1>Assembling - amd64\regs.asm
1>Assembling - amd64\cpuid.asm
1>Assembling - amd64\intstubs.asm
1>Compiling - common\newbp.c
1>Compiling - common\hvm.c
1>Compiling - common\portio.c
1>Compiling - common\comprint.c
1>Compiling - common\hypercalls.c
1>Compiling - common\traps.c
1>warnings in directory d:\nbp-0.32-public\nbp-0.32-public\common
1>d:\nbp-0.32-public\nbp-0.32-public\common\traps.c : warning C4819: The file contains a character that cannot be represented in the current code page (936). Save the file in Unicode format to prevent data loss
1>Compiling - common\interrupts.c
1>Compiling - common\common.c
1>Compiling - common\paging.c
1>Compiling - common\snprintf.c
1>Compiling - common\chicken.c
1>Compiling - common\dbgclient.c
1>Linking Executable - bin\amd64\newbp.sys
BUILD: Finish time: Sun Jul 20 20:22:39 2008
BUILD: Done

30 files compiled - 2 Warnings
2 libraries built
1 executable built

D:\nbp-0.32-public\nbp-0.32-public>ctags -R
'ctags' is not recognized as an internal or external command,
operable program or batch file.

D:\nbp-0.32-public\nbp-0.32-public>

```

图 3.3 显示编译成功信息的 WinDDK 控制台

如果看到这个提示，恭喜你，编译成功了！

## 演示 NewBluePill

运行 NewBluePill 就有一定要求了，首先要求必须运行在支持虚拟技术（HVM）的 CPU 上，并且推荐在 64 位或者支持虚拟 64 位技术的 CPU 上运行，原因是虽然 NewBluePill 程序中附带了支持 32 位 CPU 的代码，但是有几个函数在编译时（Vista x86 Checked Mode）会出现问题<sup>1</sup>，而且有几个函数是未实现的，所以还是在 x64 上去跑吧。

下面是详细步骤：

步骤 1：重启被调试机，按 F8，然后选择 Disable Driver Signature Enforcement(切记一定要用这个模式启动，否则不能加载未签名的驱动程序)

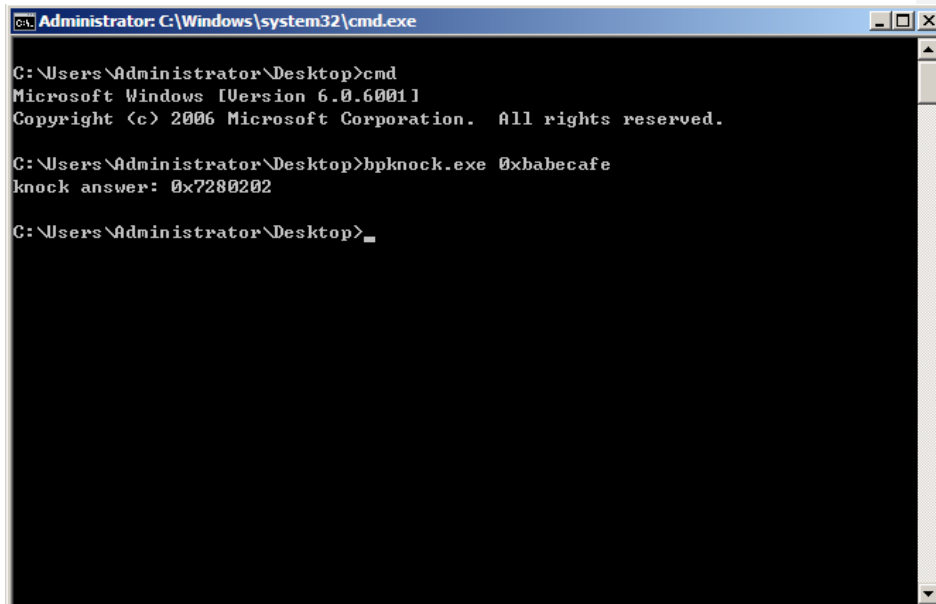
步骤 2：去 nbp-0.32-public 主目录及其子目录下找到下面几个编译生成的二进制文件：

<sup>1</sup> 这个问题会作为实验内容“第十二章 移植 NewBluePill 到 32 位系统”留给读者解决

批注 [S13]: 此处出现“第十二章 移植 NewBluePill 到 32 位系统”章号

bpknock.exe, dbgclient.sys, newbp.sys

步骤 3: 运行下 bpknock 0xbabecafe 看下没运行 NewBluePill 的输出结果。



```
Administrator: C:\Windows\system32\cmd.exe

C:\Users\Administrator\Desktop>cmd
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\Users\Administrator\Desktop>bpknock.exe 0xbabecafe
knock answer: 0x7280202

C:\Users\Administrator\Desktop>
```

图 3.4 未加载 newbp 驱动的 bpknock 程序输出结果

步骤 4: 打开 DebugView, 在 DebugView 中的 Capture 菜单中选中下列项:

Capture Global Win32

Capture Kernel

Enable Verbose Kernel Output(这个一定要选中)

Pass-Through

Capture Events

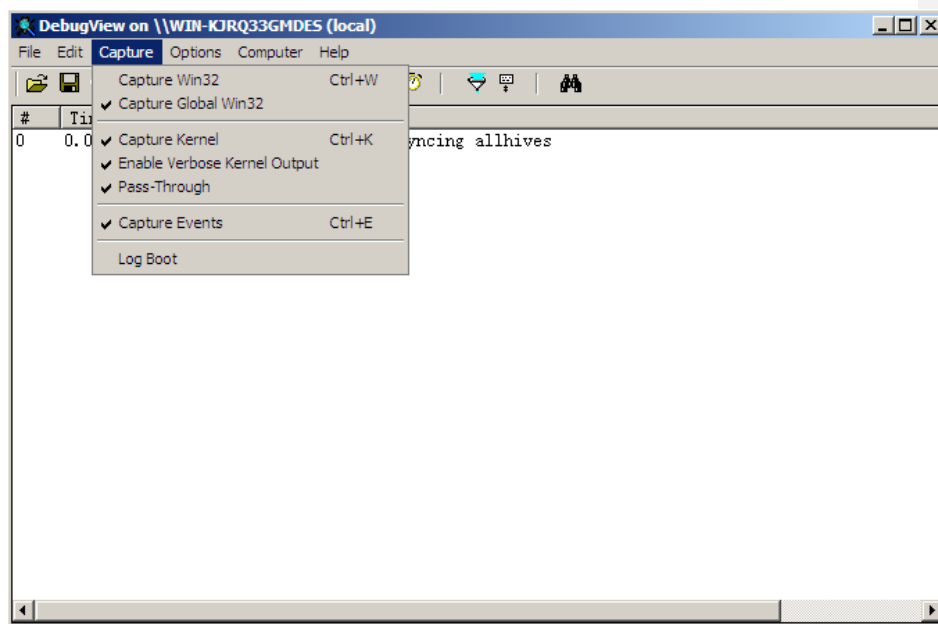
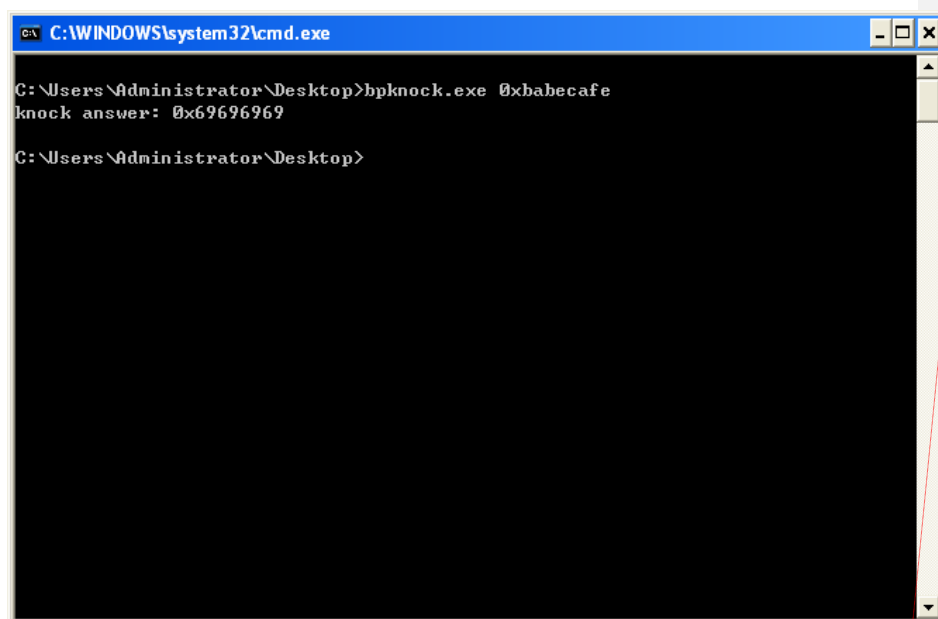


图 3.5 配置 DebugView

然后打开 InstDrv，先后加载并启动 dbgclient.sys 驱动和 newbp.sys 驱动<sup>1</sup>

步骤 6：再运行下 bpknock 0xbabecafe 看下运行了 nbp 的输出结果。（如图 3.6 所示）



批注 [S14]: 以后再把这个换成 win2k8 下面的

图 3.6 加载 newbp 驱动的 bpknock 程序输出结果

<sup>1</sup> 这两个驱动分别在各自文件夹的 bin 子目录下

## 调试 NewBluePill

调试 NewBluePill 需要用到 WinDbg, 主要过程如下:

步骤 1. 为了调试过程中可以下断点 (切记做这一步只是为了以后能够调试, 并且使得 NewBluePill 驱动只能运行在操作系统的 debug 模式下), 修改 common 目录下的 newbp.c 文件, 在 DriverEntry 方法的一开始添加 CmDebugBreak() 方法调用<sup>1</sup> (如图 3.7 所示), 重新编译。修改后的代码如下:

```
00046: NTSTATUS DriverEntry (  
00047:     PDRIVER_OBJECT DriverObject,  
00048:     PUNICODE_STRING RegistryPath  
00049: )  
00050: {  
00051:     NTSTATUS Status;  
00052:     CmDebugBreak();  
00053:     #ifdef USE_COM_PRINTS  
00054:     PciInit ((PUCHAR) COM_PORT_ADDRESS);  
00055:     #endif  
00056:     ComInit ();  
00057:     Status = MmInitManager ();  
00058:     return Status;  
}
```

图 3.7 添加 CmDebugBreak() 方法的位置

步骤 2: 参考 Debugging Windows Vista<sup>2</sup> 修改被调试机启动项和调试项 (这一步只需做这一次就可以)

步骤 3: 重启被调试机, 可以看到启动项中多了一个 DebugEntry [debugger enabled] 项, 选中它按 F8, 然后选择 Disable Driver Signature Enforcement 项启动

步骤 4: 调试机上设置 \_NT\_SYMBOL\_PATH 环境变量, 指向 newbp.pdb 所在的目录, 用于链接符号表。

步骤 5: 调试机上启动 WinDbg, 单击 File 菜单选择 Kernel Debugging, 在弹出的对话框输入 Baud Rate 为 115200, Port 用 com1。<sup>3</sup>这是由于刚才在演示过程的第一步我们用的是默认配置, 如果调试端口发生相应改变, 这里也要改。

<sup>1</sup> CmDebugBreak() 函数实际上是一个 int 3 调用, 在非调试模式的 Windows 下, 这时这个中断的处理程序未注册, 因此执行 int 3 指令会死机。

<sup>2</sup> 文章来源: [http://www.microsoft.com/whdc/driver/tips/debug\\_vista.mspx](http://www.microsoft.com/whdc/driver/tips/debug_vista.mspx)

<sup>3</sup> 除了利用串口线调试外, 也可以利用 1394 线进行调试, 这样做的好处是数据传输速度更快。具体方法可以参考网上相关资料。

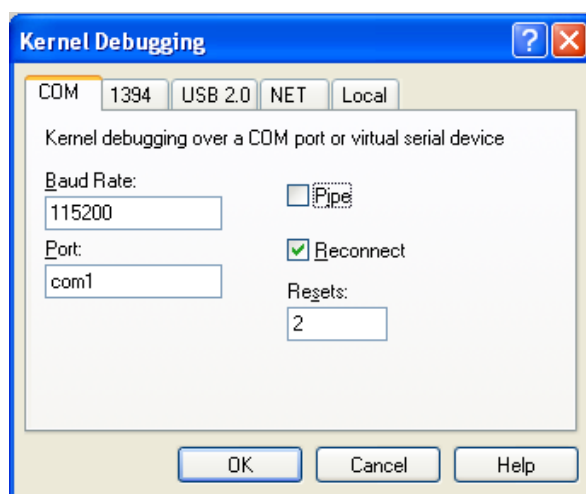


图 3.8 配置 WinDbg

步骤 6: 被调试机上先后加载并启动 dbgclient.sys 和 newbp.sys 两个驱动, 运行 bpknock 程序, 开始调试。

如果出现 symbol 不能被加载的情况可以试试 WinDbg 中的.reload 命令, 如果不行可以试试用.sympath 在 WinDbg 运行时设定 symbol 路径, 然后.reload 重新加载符号表。

成功情况下的截图:

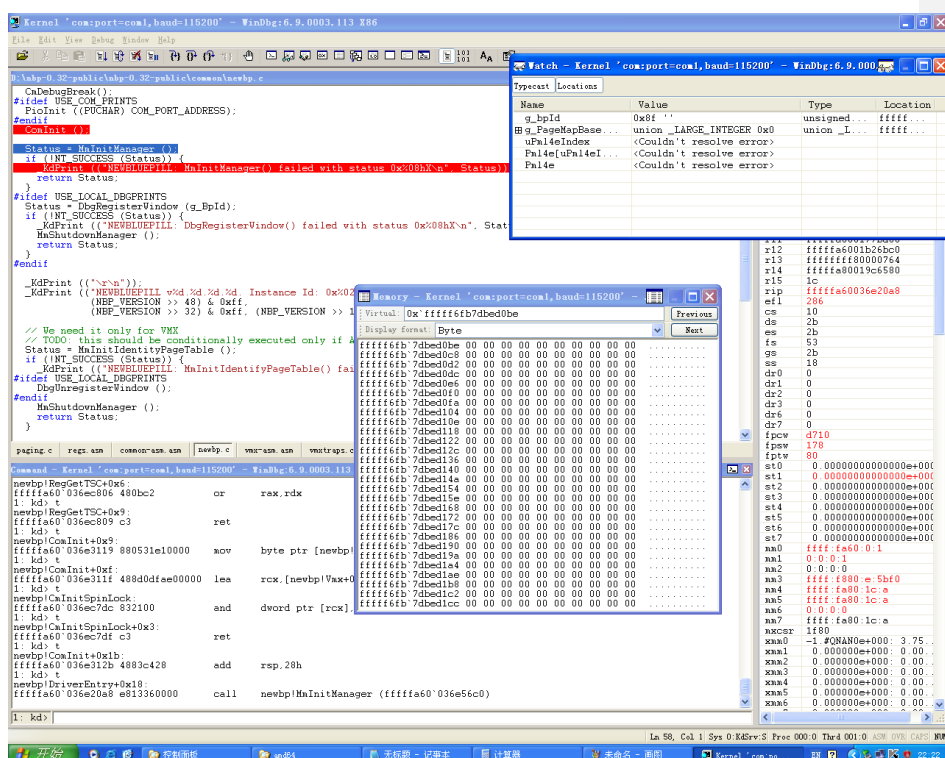


图 3.9 成功搭建调试平台

Ok, 有了调试平台, 我们就可以揭开 NewBluePill 的层层面纱了。

## 四、NewBluePill 的启动和卸载

在这一章中，我们将探究 NewBluePill 驱动的启动和关闭过程，从而将 NewBluePill 各组件串接起来（本章不涉及 dbgclient.sys 的启动过程，“第八章 NewBluePill 其它系统”会对其加以说明）。启动和卸载过程在 NewBluePill 中占据很大的比重，这一点可以从相关代码所占总代码量比重上看出：约 30% 的源文件均与启动和卸载过程有关。所以了解 NewBluePill 的启动和卸载，将对了解其功能实现有极大帮助。

在后续章节中，我们将逐一探索每个组件是如何完成其功能的。

### NewBluePill 驱动的启动过程

NewBluePill 驱动入口在 common\newbp.c 文件中，入口函数为 DriverEntry 函数（Newbp.c+46 行）。这个函数流程图如图 4.1 所示：

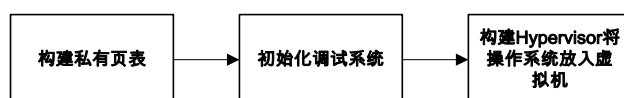


图 4.1 NewBluePill 启动流程图

下面我们将按照图 4.1 逐一介绍每部分运行过程。

### 构建私有页表

在 DriverEntry 函数中，从 58 行到 62 行，以及从 79 行到 114 行，都是在完成私有页表的构建。（对于内存相关部分，本章中我们只是列举出被调用的函数，每个函数的详细作用我们会在“第五章 NewBluePill 内存系统”中阐述）

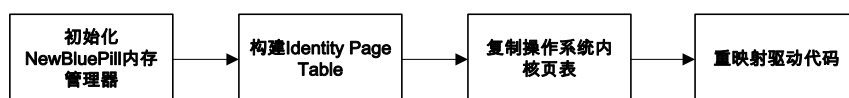


图 4.2 NewBluePill 初始化过程中构建私有页表的流程

- **初始化 NewBluePill 内存管理器** Newbp.c 中第 58 行到第 62 行，该代码调用函数 MmInitManager(), 该函数会在内存中分配新空间作为 NewBluePill 自己的页表，并按照 x64 地址翻译机制构建出页表结构。
- **构建 Identity Page Table** Newbp.c 中第 79 行到第 87 行，不知道这个函数干什么用的
- **复制操作系统内核页表** Newbp.c 中第 89 行到第 97 行，该代码调用函数 MmMapGuestKernelPages(), 该函数会根据当前 Windows 操作系统的内核页表内容，填充 NewBluePill 自己的页表。
- **重映射驱动代码** Newbp.c 中第 98 行到第 114 行，调用函数

批注 [S15]: 第八章 NewBluePill 其它系统  
章号

批注 [S16]: “第五章 NewBluePill 内存系统”章号

批注 [S17]: 待调查后补充  
TODO

MmMapGuestPages()。NewBluePill 作为驱动，必定要占据内核空间，此处调用该函数，就是要把自己占用的操作系统内核页面空间的页表信息复制到自己的页表中，从而为以后实现页表隐藏打下基础。

## 初始化调试系统

DriverEntry 函数的 63 行到 75 行在初始化 NewBluePill 的调试系统。（对于调试系统部分，本章中我们只是列举出被调用的函数，每个函数的详细作用我们会在“第八章 NewBluePill 其它系统”中阐述）

NewBluePill 初始化调试系统，是通过调用函数 DbgRegisterWindow()实现的，这个函数主要作用是根据当前 NewBluePill 驱动实例的唯一 ID（驱动运行时读取处理器时间寄存器（Time Stamp Counter，TSC），并将低八位作为这个 ID），分配一段共享内存，使得可以将打印信息全部保存在这段内存上，从而使得本机调试变得方便。（dbgclient.sys 会读取这段共享内存的内容并发送到调试机上，其实也可以调整下让它将这些内容保存在磁盘上）

NewBluePill 也直接支持利用串口发送调试信息到调试机上（不需 dbgclient.sys 的帮助）

## 构建 Hypervisor 并将操作系统放入虚拟机

构建 Hypervisor，并将操作系统放入虚拟机的工作，是在 DriverEntry 函数中的 116 行到 132 行完成的，主要调用的函数有两个：

- HvmInit()函数
- HvmSwallowBluepill()函数

HvmInit()函数的作用是：确定当前系统架构是否支持 HEV 技术，并确定 NewBluePill 支持哪种 HEV 技术（Intel VT/AMD SVM）。最后根据获得的信息，将相应的处理函数组 and 平台信息捆绑在 Hvm 结构体上，该结构体可以通过在 windbg 下输入 dt Hvm 命令实现。

### 实验：查看 Hvm 结构体

在 NewBluePill 运行时，您可以在 windbg 下使用 dt Hvm 命令查看当前平台对应的 Hvm 结构体内容：

Lkd

批注 [S18]: 给出该实验的实验结果

实际上，检查是否支持 HEV 技术，和确定平台的函数（两者都由 Hvm->ArchIsHvmImplemented() 实现）在后面的 HvmSubvertCpu() 函数中（被 HvmSwallowBluepill()调用，后面会讲到）再次出现，我们认为重复检查是不必要的。

HvmSwallowBluepill()函数的作用是：该函数及其子函数给每个处理器（Processor）安装 NewBluePill 的 Hypervisor，这个函数也是 NewBluePill 主要逻辑的初始化入口。下面，我们就将进入这个入口背后的世界。



## 进入 NewBluePill 的世界

当我们进入了 `HvmSwallowBluePill()` 函数,也就踏入了 NewBluePill 的世界,这个世界层峦叠嶂,险象环生,既涉及到嵌入式编程中的精细控制,又涉及到两个模式的相生相存。程序直到运行到最后一步,一切才能正常工作,少了中间任何一步,一切便陷入混乱之中,正所谓“如临深渊,如履薄冰”。

为了便于理解,我们人为的把启动过程分为两个阶段,进入虚拟机模式前的初始化部分称为阶段 1 初始化(Phase 1),进入虚拟机模式后的初始化部分称为阶段 2 初始化(Phase 2)。

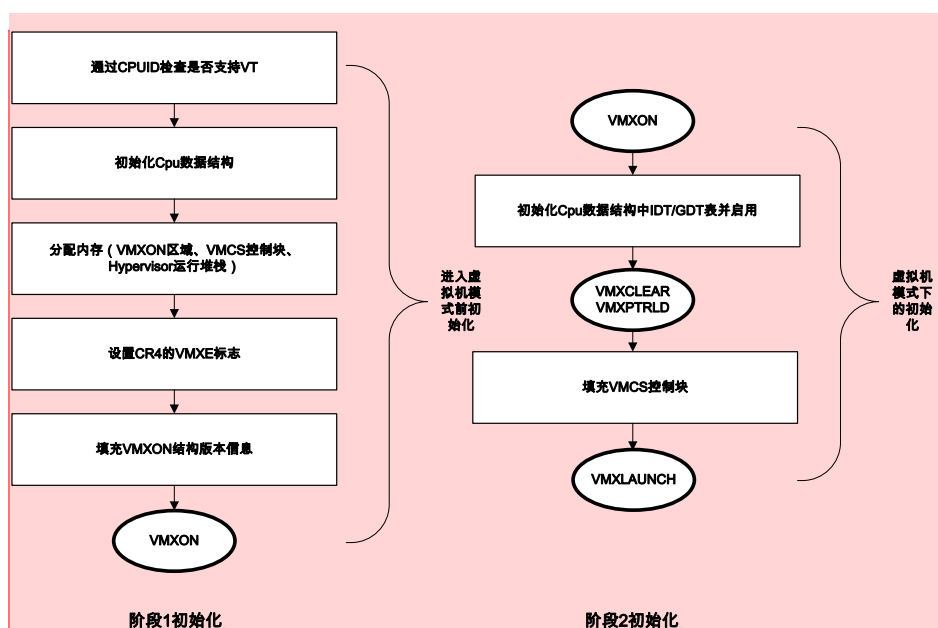


图 4.3 阶段 1 和阶段 2 初始化流程图

批注 [S19]: 此图应该分为 VT 和 SVM 两者  
分别的初始化过程图

### 阶段 1 初始化

阶段 1 初始化主要任务是为每个处理器开启虚拟机模式。

`HvmSwallowBluePill()` 函数首先会在操作系统范围内申请一个互斥锁 `g_HvmMutex` (`common\Hvm.c` 第 556 行),当多个 NewBluePill 驱动同时启动/卸载时,该锁能够保证每个时刻只能有一个 NewBluePill 实例运行,以避免嵌套设置失败。随后,`HvmSwallowBluePill()` 函数遍历每个处理器 (`common\Hvm.c` 第 558 行~581 行)调用 `CmDeliverToProcessor()` 函数,进而通过回调手段调用 `HvmSubvertCpu()` 函数,后者用来指导对 VT/SVM 两种平台执行同样的侵染过程。

`HvmSwallowBluePill()` 函数中对每个处理器遍历的代码如图 4.4 所示,注意这个模式所采用的手法,遍历内部传入了回调函数,并且既要处理 `CmDeliverToProcessor()`

的运行结果，又要处理回调函数的运行结果。该模式在安装和卸载过程中均有出现。<sup>1</sup>

```

00558: for (cProcessorNumber = 0; cProcessorNumber < KeNumberProcessors; cProcessorNumber++) {
00559:
00560:     _KdPrint (("HvmSwallowBluepill(): Subverting processor #d\n", cProcessorNumber));
00561:
00562:     Status = CmDeliverToProcessor (cProcessorNumber, CmSubvert, NULL, &CallbackStatus);
00563:
00564:     if (!NT_SUCCESS (Status)) {
00565:         _KdPrint (("HvmSwallowBluepill(): CmDeliverToProcessor() failed with status 0x%08hX\n", Status));
00566:         KeReleaseMutex (&g_HvmMutex, FALSE);
00567:         HvmSpitOutBluepill ();
00568:         return Status;
00569:     }
00570:
00571:     if (!NT_SUCCESS (CallbackStatus)) {
00572:         _KdPrint (("HvmSwallowBluepill(): HvmSubvertCpu() failed with status 0x%08hX\n", CallbackStatus));
00573:         KeReleaseMutex (&g_HvmMutex, FALSE);
00574:         HvmSpitOutBluepill ();
00575:         return CallbackStatus;
00576:     }
00577: }
00578:
00579: KeReleaseMutex (&g_HvmMutex, FALSE);
00580:
00581: if (KeNumberProcessors != g_uSubvertedCPUs) {
00582:     HvmSpitOutBluepill ();
00583:     return STATUS_UNSUCCESSFUL;
00584: }
00585:
00586: }

```

图 4.4 HvmSwallowBluePill() 函数中对每个处理器遍历的代码

CmDeliverToProcessor() 函数用于提高中断优先级 (DISPATCH\_LEVEL IRQL)，并在指定的处理器上执行一个函数，从而可以不被打断<sup>2</sup>的在该处理器上执行这个函数，同时也在执行最后会恢复到原先执行该指令的处理器组设置上（也称亲核性，CPU Affinity，参见 Common.c 代码第 359 行）。在安装过程中，通过它来执行 HvmSubvertCpu() 函数，从而保证 HvmSubvertCpu() 函数的运行都作用于指定的处理器。

**important** 在操作系统中断优先级高于等于 DISPATCH\_LEVEL 的情况下，不能使用可分页的内存，因为访问一个已换出页的内存地址会发生一个内存页换入 (swap-in) 操作，而该操作是在较低的中断优先级（确切的说：APC\_LEVEL）进行的。这种情况下，操作系统会蓝屏。

正确的做法是：如果确实要在 DISPATCH\_LEVEL 或更高的中断优先级下分配内存，则要从操作系统的不分页内存池 (Nonpaged pool) 中分配内存，该段空间在操作系统存活期间内驻留在内存。

我们可以通过研究 MmAllocatePages() 函数 (common\Paging.c 第 275 行) 和其调用者出现的逻辑位置看到 NewBluePill 驱动是怎样注意这个问题的。

HvmSubvertCpu() 函数主要做了两件事情：

- 1) 创建并初始化 CPU 数据结构 (Hvm.c 中第 365 行到 418 行，453 行到 459 行，后者用于构建 NewBluePill 自己的 GDT、IDT 表，并在虚拟机模式中启用)。
- 2) 指导怎样侵染一个核。(Hvm.c 中第 420 行到 465 行，不包括 453 行到 459 行)

CPU 数据结构定义在 Hvm.h 文件的第 53 行，该数据结构在阶段 1 中初始化，阶段 2 中被使用。查找陷入处理函数、页表隐藏以及实现 Blue Chicken 反虚拟机探测技术都需要这个

<sup>1</sup> 在这段代码中一个小的 Bug 是 for 循环应该使用 KeQueryActiveProcessors() 函数而不是 KeNumberProcessors 变量来获得当前处理器个数，这个 bug 会影响到支持热插拔 CPU 的系统，后果是可能导致在这种系统上某些处理器未被侵染，或重复侵染而导致内存泄露、系统异常等其他问题。

<sup>2</sup> 确切地说还是会被打断的，比如时钟中断就可以打断 NewBluePill 的运行，但是这些打断不会干扰到 NewBluePill 的运行。

结构体的帮助，可以说，这个结构体是继 VMCS/VMCB 之后第二重要的结构体。下面先详细解释下这个数据结构

```
typedef struct _CPU
{
    PCPU      SelfPointer;      /*必须放在第一个，与处理事件逻辑有关*/

    union
    {
        SVM      Svm;
        VMX      Vmx;
    };                          /*标示所运行在何种平台上 SVM/VT,里面含有构建该平台下 Hypervisor 和虚拟机的关键信息*/

    ULONG      ProcessorNumber; /*该处理器在操作系统中的处理器序号*/
    ULONG64     TotalTscOffset; /* */

    LARGE_INTEGER LpicBaseMsr; /*实际上未被用到*/
    PHYSICAL_ADDRESS LpicPhysicalBase; /*实际上未被用到*/
    PCHAR        LpicVirtualBase; /*实际上未被用到*/

    LIST_ENTRY   GeneralTrapsList; /*该处理函数列表用于处理一般原因陷入的 VMEXIT 事件，比如对 cr0-cr4 寄存器的操作*/
    LIST_ENTRY   MsrTrapsList; /*该处理函数列表用于处理操作 MSR 寄存器而产生的 VMEXIT 事件，比如 rdmsr 指令和 wrmsr 指令*/
    LIST_ENTRY   IoTrapsList; /*该处理函数列表用于处理因 I/O 操作而产生的 VMEXIT 事件，实际上没有用到1*/

    PVOID        SparePage; /*存储一个空页面的引用，用于后面的页表隐藏操作*/
    PHYSICAL_ADDRESS SparePagePA; /*SparePage 原来的物理地址*/
    PULONG64     SparePagePTE;

    PSEGMENT_DESCRIPTOR GdtArea; /*NewBluePill 自己的 GDT 表空间*/
    PVOID          IdtArea; /*NewBluePill 自己的 IDT 表空间*/

    PVOID          HostStack; /*NewBluePill Hypervisor 在该处理器上的运行时堆栈空间，16 页大小，这段空间顶端放置了 CPU 结构体*/
    BOOLEAN        Nested; /*是否是嵌套 NewBluePill*/

#ifdef INTERCEPT_RDTSCs
```

批注 [S20]: 调查时间迷惑后补充

批注 [S21]: 调查后补充

<sup>1</sup> 可能算一个 bug，当前确实有 I/O 造成的 VMEXIT 事件的处理函数，但是相关信息却并未放到 IoTrapsList 链表中。不过影响不大。MsrTrapsList 也有相似的问题，只有 NewBluePill 对 SVM 技术支持的实现中用到了这个链表。

```

// variables for RDTSC tracing and cheating
ULONG64 Tsc;
ULONG64 LastTsc;
ULONG64 EmulatedCycles;
int Tracing; // we trace instructions
until Tracing = 0
int NoOfRecordedInstructions;

//currently not implemented:
//int NextInstrOffsetinBuffer;
//PVOID RecordedInstructions[INSTR_TRACE_MAX * 16];

#endif
#ifdef BLUE_CHICKEN

int ChickenQueueSize;
ULONG64 ChickenQueueTable[CHICKEN_QUEUE_SZ];
int ChickenQueueHead, ChickenQueueTail;

UCHAR OriginalTrampoline[0x600];

#endif

ULONG64 ComPrintLastTsc; /*上次 Com 口输出时间，调试系统会利用这个值*/

} CPU,
*PCPU;

```

批注 [S22]: 搞清楚时间欺骗后更新此处

批注 [S23]: 搞清楚 Blue Chicken 机制后更新此处

## 实验：查看 Cpu 结构体

在 NewBluePill 调试状态下，您可以在 windbg 下，通过 bp 命令设置断点，使得被调试机系统停在 NewBluePill 包含 Cpu 结构的函数内，然后使用 dt Cpu 命令查看 Cpu 结构体内容。比如加载 NewBluePill 驱动后，当驱动停在 CmDebugBreak() 后，在 windbg 中输入 bp VmxDispatchCpuid，然后按 F5。当驱动停在 VmxDispatchCpuid() 后，输入 dt Cpu:

Lkd

批注 [S24]: 给出该实验的实验结果

有一点需要注意，那就是在 HvmSubvertCpu() 函数中处理 SparePage 的地方 (hvm.c 第 414 行)，在为 Cpu 结构体的 SparePagePTE 赋值的时候：

```
Cpu->SparePagePTE = (PULONG64)((((ULONG64)(Cpu->SparePage) >> 9) & 0x7fffffff8)+PT_BASE);
```

这么做是因为，

批注 [S25]: Cpu-&gt;SparePagePTE 的问题

## 关于页表项状态位

在 Hvm.c 的 417 行:

```
*Cpu->SparePagePTE |= (1 << 4);
```

这条语句是在设置其所指向的 PTE 是否禁用 Cache。关于硬件页表项各状态位的含义,可以参考 *Windows Internals, 4<sup>th</sup> Edition* 的 Chapter 7. Memory Management 中的 Address Translation 部分中的相关内容,或者 Intel/AMD 手册中的相关部分。

HvmSubvertCpu() 函数还指导了 NewBluePill 如何侵染一个核。过程按先后顺序分为三步:

- a) Hvm->ArchRegisterTraps(Cpu) 注册 VMEXIT 事件处理函数
- b) Hvm->ArchInitialize(Cpu, CmSlipIntoMatrix, GuestRsp)  
开启虚拟机模式, 成为 Hypervisor 并构建虚拟机以装入原来的操作系统
- c) Hvm->ArchVirtualize(Cpu) 开启虚拟机

可以看到,实际上 Hvm 结构体起到了硬件抽象的作用,对上层屏蔽了 SVM 和 VT 技术两者的差异,提高了代码的复用性。下面,我们将深入 NewBluePill 适应于每个技术的具体实现,查看阶段 1 是如何完成的。

## SVM 技术下阶段 1 的初始化

在 NewBluePill 对 SVM 技术的支持中, HvmSubvertCpu() 函数中 Hvm 结构体的三个函数对应关系如下:

表 4.1 Hvm 结构体中三函数对应关系 (AMD 平台)

Hvm 结构体中函数	映射函数 (svm\Svm.c 和 svm\Svmtraps.c)
Hvm->ArchRegisterTraps()	SvmRegisterTraps()
Hvm->ArchInitialize()	SvmInitialize()
Hvm->ArchVirtualize()	SvmVirtualize()

首先是 SvmRegisterTraps() 函数

批注 [S26]: 对 SVM 技术的支持随后再写

## VT 技术下阶段 1 的初始化

在 NewBluePill 对 VT 技术的支持中, HvmSubvertCpu() 函数中 Hvm 结构体的三个函数对应关系如下:

表 4.2 Hvm 结构体中三函数对应关系 (AMD 平台)

Hvm 结构体中函数	映射函数 (vmx\Vmx.c 和 vmx\Vmxtraps.c)
Hvm->ArchRegisterTraps()	VmxRegisterTraps()
Hvm->ArchInitialize()	VmxInitialize()
Hvm->ArchVirtualize()	VmxVirtualize()

## VmxRegisterTraps 函数

首先是 `VmxRegisterTraps()` 函数，该函数通过调用 `TrInitializeGeneralTrap()` 函数将各事件与处理函数捆绑起来成为一个新的 `Trap` 元素，并通过调用 `TrRegisterTrap()` 函数将这个 `Trap` 存入 `Cpu` 结构的 `GeneralTrapsList` 链表中<sup>1</sup>。同时，由于 `MSR` 操作、`I/O` 操作、和其他通用操作间的不同，所以在 `Trap` 元素中又使用了另外的类型来存储三者 `Trap` 处理特定的信息。`Trap` 元素及其相关 `NBP_TRAP_DATA_GENERAL`、`NBP_TRAP_DATA_MSR`、`NBP_TRAP_DATA_IO` 结构体定义如下：

```
typedef struct _NBP_TRAP
{
    /*_NBP_TRAP 结构体存储了 Trap 类型, Trap 发生原因, Trap 处理函数等一系列的信息,
    用于在发生该类型 Trap 时搜索适当的处理函数*/
    LIST_ENTRY    le;          /*Windows 系统链表元素必备域, 且必须在头部*/

    TRAP_TYPE     TrapType;    /*记录该 Trap 的类型, 可以是 TRAP_DISABLED、
    TRAP_GENERAL、TRAP_MSR、TRAP_IO 之一*/
    TRAP_TYPE     SavedTrapType; /*用于在该 Trap 元素被 Disable 时, 备份
    TrapType, 以供在该元素被 Enable 时恢复*/

    union
    {
        NBP_TRAP_DATA_GENERAL    General;
        NBP_TRAP_DATA_MSR        Msr;
        NBP_TRAP_DATA_IO         Io;
    };                          /*用于该 Trap 元素记录附加信息, 详细内容参见这三个
    结构体注释*/

    NBP_TRAP_CALLBACK    TrapCallback;    /*记录若该 Trap 元素被选中,
    那么应该调用的事件处理函数*/
    BOOLEAN               bForwardTrapToGuest; /*记录是否应该把这个事件继续上
    传, 在处理 MSR 造成的 VMEXIT 时会用到*/

} NBP_TRAP,
*PNBP_TRAP;
```

Trap 结构体定义 (Traps.h 中第 67 行到 87 行)

```
typedef struct _NBP_TRAP_DATA_GENERAL
{
    ULONG           TrappedVmExit;          /*记录该 Trap 元素对于什么原因造成
    的 VMEXIT 事件进行处理*/
```

<sup>1</sup> NewBluePill 对 VT 技术的支持实现中，所有的事件处理的 `Trap` 元素都放到了 `Cpu->GeneralTrapsList` 链表中，虽然与 VT 技术特性有关，但也应该算是一个 bug，不过不影响当前运行。

```

        ULONG64    RipDelta;                /*这个值用于 NewBluePill Hypervisor
处理完 VMEXIT 事件返回虚拟机后, Guest_Rip 上要加的字节数以跳过触发 VMEXIT 事件
的指令, 如果为 0 则说明需要在陷入 Hypervisor 后, 从 VMCS 中获得该值*/
    } NBP_TRAP_DATA_GENERAL,
    *PNBP_TRAP_DATA_GENERAL;

```

NBP\_TRAP\_DATA\_GENERAL 结构体定义 (Traps.h 中第 36 行到 41 行)

```

typedef struct _NBP_TRAP_DATA_MSR
{
    ULONG32    TrappedMsr;
    UCHAR      TrappedMsrAccess;
    UCHAR      GuestTrappedMsrAccess;
} NBP_TRAP_DATA_MSR,
*PNBP_TRAP_DATA_MSR;

```

NBP\_TRAP\_DATA\_MSR 结构体定义 (Traps.h 中第 43 行到 49 行)

```

typedef struct _NBP_TRAP_DATA_IO
{
    ULONG      TrappedPort;
} NBP_TRAP_DATA_IO,
*PNBP_TRAP_DATA_IO;

```

NBP\_TRAP\_DATA\_IO 结构体定义 (Traps.h 中第 51 行到 55 行)

结合“第二章 深入 HEV 技术细节”中的内容我们可以看到, 在 `VmxRegisterTraps()` 函数中, NewBluePill Hypervisor 对所有的无条件产生 VMEXIT 事件的汇编指令都添加了相应的处理函数, 如表 4.3 所示。

批注 [S27]: 看完 AMD 后补充

批注 [S28]: “第二章 深入 HEV 技术细节” 章号

表 4.3 VT 下 NewBluePill Hypervisor 对无条件陷入绑定的处理函数

造成无条件陷入的指令	对应硬件 VMEXIT 原因标记 <sup>1</sup>	处理函数 (vmx\Vmxtaps.c)
VMCALL	EXIT_REASON_VMCALL	VmxDispatchVmxInstrDummy
VMCLEAR	EXIT_REASON_VMCLEAR <sup>2</sup>	VmxDispatchVmxInstrDummy
VMLAUNCH	EXIT_REASON_VMLAUNCH	VmxDispatchVmxInstrDummy
VMRESUME	EXIT_REASON_VMRESUME	VmxDispatchVmxInstrDummy
VMPTRLD	EXIT_REASON_VMPTRLD	VmxDispatchVmxInstrDummy
VMPTRST	EXIT_REASON_VM PTRST	VmxDispatchVmxInstrDummy
VMREAD	EXIT_REASON_VMREAD	VmxDispatchVmxInstrDummy
VMWRITE	EXIT_REASON_VMWRITE	VmxDispatchVmxInstrDummy
VMXON	EXIT_REASON_VMXON	VmxDispatchVmxInstrDummy

<sup>1</sup> 所有这些标记定义 Vmx.h 中, 具体这些定义为什么这样取值, 请参考 *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B* 手册 Appendix I: VMX Basic Exit Reasons

<sup>2</sup> 在源代码中有一个 bug, 那就是在 Vmxtaps.c 的第 518 行和 519 行两个都是 EXIT\_REASON\_VMCALL, 其中一个应该是 EXIT\_REASON\_VMCLEAR。

VMXOFF	EXIT_REASON_VMXOFF	VmxDispatchVmxInstrDummy
CPUID	EXIT_REASON_CPUID	VmxDispatchCpuid
INVD	EXIT_REASON_INVD	VmxDispatchINVD

此外，也对有条件陷入的指令添加了处理函数，表 4.4 列举了它们：

表 4.4 VT 下 NewBluePill Hypervisor 对有条件陷入绑定的处理函数

造成有条件陷入的指令	对应硬件 VMEXIT 原因标记 <sup>1</sup>	处理函数 (vmx\Vmxttraps.c)
RDMSR	EXIT_REASON_MSR_READ	VmxDispatchMsrRead
WRMSR	EXIT_REASON_MSR_WRITE	VmxDispatchMsrWrite
CR Ops	EXIT_REASON_CR_ACCESS	VmxDispatchCrAccess
NMI 中断	EXIT_REASON_EXCEPTION_NMI	VmxDispatchException
RDTSR	EXIT_REASON_RDTSR	VmxDispatchRdtsc
I/O 操作 <sup>2</sup>	EXIT_REASON_IO_INSTRUCTION	VmxDispatchIoAccess

关于这些处理函数的功能，我们会在“第六章 NewBluePill 陷入事件管理系统”进行详细讨论。

批注 [S29]: “第六章 NewBluePill 陷入事件管理系统”章号

## VmxInitialize 函数

在注册完各陷入事件处理函数后，NewBluePill 会调用 VmxInitialize() 函数进行后续的阶段 1 初始化工作。

```
typedef struct _VMX
{
    PHYSICAL_ADDRESS    VmcsToContinuePA;    // MUST go first in the
    structure; refer to SvmVmrn() for details
    PVOID                _2mbVmcbMap;

    PHYSICAL_ADDRESS    OriginalVmcsPA;      /*VT 技术中用于控制虚拟机的
    VMCS 结构体物理地址*/
    PVOID                OriginalVmcs;       /*VMCS 结构体指针*/
    PHYSICAL_ADDRESS    OriginalVmxonRPA;    /*为了开启虚拟机环境所需的
    VMXON 区域的物理地址*/
    PVOID                OriginaVmxonR;     /*该指针指向 VMXON 区域*/

    PHYSICAL_ADDRESS    IOBitmapAPA; // points to IOBitMapA.
    PVOID                IOBitmapA;
```

批注 [S30]: 未注释完全，待完全写好启动过程后看看有没有什么新发现

<sup>1</sup> 所有这些标记定义 Vmx.h 中，具体这些定义为什么这样取值，请参考 *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B* 手册 Appendix I: VMX Basic Exit Reasons

<sup>2</sup> 注意到在 Vmxttrap.c 第 589 行，只有在头文件中预先定义 VMX\_ENABLE\_PS2\_KBD\_SNIFFER，NewBluePill Hypervisor 才会处理访问 I/O 造成的陷入。



```

    PHYSICAL_ADDRESS    IOBitmapBPA; // points to IOBitMapB
    PVOID                IOBitmapB;

    PHYSICAL_ADDRESS    MSRBitmapPA; // points to MsrBitMap
    PVOID                MSRBitmap;

    ULONG64              GuestCR0;          //Guest's CR0.
    ULONG64              GuestCR3;          //Guest's CR3. for
storing guest cr3 when guest diasble paging.
    ULONG64              GuestCR4;          //Guest's CR4.
    ULONG64              GuestEFER;
    UCHAR                GuestStateBeforeInterrupt[0xc00];

} VMX,
*PVMX;

```

VMX 结构体定义 (vmx\Vmx.h 中第 205 行到 231 行)

VmxInitialize() 函数主要职责是初始化 Cpu 结构体中的 Cpu->Vmx 项，这之中最重要的莫过于分配 Vmxon 空间 (Vmxon Regions) (Vmx.c 文件第 519 行)、VMCS 空间 (Vmx.c 文件第 530 行)、IOBitmap (I/O 位图, Vmx.c 文件第 544 行和 554 行) 和 MsrBitmap (Msr 位图, Vmx.c 文件第 564 行)。具体这四个空间的作用可参考“第二章 深入 HEV 技术细节”中相关内容描述。

当这些结构体都已经被分配出来后，阶段 1 的初始化工作也将要结束。在 VmxInitialize() 函数的第 573 行，程序调用 VmxEnable() 函数。

VmxEnable() 函数中每个有关设置的步骤都很关键。首先设置 CR4 寄存器第 13 位为 1 (该位也称 CR4.VMXE，用于执行 VMXON 开启虚拟机模式的前提) (Vmx.c 文件第 46 行)，然后为了保险起见，检查 CR4 和 IA32\_FEATURE\_CONTROL，确保当前系统支持开启虚拟机模式。接下来在 Vmxon 区域中填充 VMCS 版本号 (Vmx.c 文件第 58 行到第 59 行)。最后通过调用 VmxTurnOn() 函数 (实际就是 vmxon 汇编指令) 开启虚拟机模式。

至此 VT 技术下阶段 1 的初始化工作全部完成。

批注 [531]: “第二章 深入 HEV 技术细节” 章号

## 阶段 2 初始化

### VmxVirtualize 函数