六、 NewBluePill 程序逻辑

1) Nbp 的初始化过程

1. 总体概述

Nbp 驱动入口在 Newbp.C 文件中，采用标准 Windows 驱动接口，入口函数为：

NTSTATUS DriverEntry (PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)

在加载该驱动后，nbp 的初始化过程如下（见图 6.1 和图 6.2）

在这个过程中，值得注意的是在 CmSubvert 方法上，它存储了当前操作系统的所有寄存器的值，并在给 HvmSubvertCpu()函数创建好传参堆栈后把 rsp 的值当作传参传入 HvmSubvertCpu()函数。这个函数的返回确实在另一个 VM 环境下，当调用了 CmSlipIntoMatrix()函数时，原来 CmSubvert 保护的值被一个一个恢复，最后一个 ret 返回的地址使用的正是 CmSubvert()函数的返回地址，关于这一点可以参考 64 位平台的函数调用规范。

2. 具体描述

通过上述简单描述，接下来让我们具体看下 nbp 的初始化过程。内存部分的初始化过程具体描述会在后面 nbp 的内存部分讲到，因此在此处暂时跳过。

1. Nbp 进入驱动入口后首先做的是获得一个全局唯一的 nbp ID 标示，通过调用 ComInit()方法，读取处理器的 64-bit time-stamp counter，取其低八位作为 g_BpId,也就是全局唯一的 nbp ID。
2. 接着 nbp 开始了对符合 Long Mode 地址翻译的页表构建工作，这一步是通过 MmInitManager()函数完成。
3. 接下来 nbp 调用 DbgRegisterWindow()函数来初始化调试窗口，主要工作是将一段调试信息内存注册为一个设备，这种做法也称为创建调试窗口，其目的是开启一段共享内存，在后面 Dbgclient 驱动可以与这个窗口互相通信进而打印调试信息。
4. 然后 nbp 会利用 MmMapGuestKernelPages()函数遍历 Windows 内核地址空间，并挂载到自己的页表上，这一步为以后使用 Windows API 提供了保证。当然在这之前也有一些对于 Intel VMX 架构的特殊处理操作，但基本功能实现目标还是一样的。
5. 接下来在 newbp.c 第 116 行，程序会调用 HvmInit()函数，从此步开始，辅助初始化工作结束，正式进入虚拟机初始化过程。HvmInit 函数的作用是确定当前系统架构是否支持 HVM 并且确定 BluePill 所运行的系统架构,并且将相应的处理函数组和平台信息捆绑在 hvm 结构体上。
6. 接下来从 HvmInit()函数返回，程序运行到 newbp.c 第 125 行，调用 HvmSwallowBluepill()函数。这个函数的作用是给每个 CPU 安装 BluePill(这里的 CPU 是指逻辑 CPU，也就是每个核)，可以说这个函数也是总的 BluePill 初始化

入口。

HvmSwallowBluepill()首先等待拿到 g_HvmMutex 互斥锁，从而确保自己在后面的操作中可以几乎不被打断的进行。然后进入一个 for loop 中为每个逻辑 CPU 安装 nbp。具体的安装过程是通过 CmDeliverToProcessor()函数完成的。需要注意的是如果在任一 CPU 上 nbp 安装失败，那么 nbp 将调用 HvmSpitOutBluepill()函数卸载所有 CPU 上的 nbp

7.

2) DbgClient 的初始化过程

DbgClient 是 BluePill Debug 信息的消费者，以注册驱动的方式出现，Debug 信息的生产者，也就是 nbp 驱动，会把信息写到\\Device\\itldbgclient 这个设备的共享内存空间上。

由于 DbgClient 同样是一个驱动，因此其入口函数同样是 DriverEntry，初始化过程如下：

1．声明”\\Device\\itldbgclient”和”\\DosDevices\\itldbgclient”两个设备，前者将是 nbp 驱动发送调试信息的目标设备。

2．自定义了一个名为 g_ShutdownEvent 的 Event,而且是 NotificationEvent。

3．开启一个系统内核线程，其作用是周期性打印输出信息，这个线程执行的函数是 ScanWindowsThread()。

4．将驱动处理函数 DriverDispather 捆绑在 IRP_MJ_CREATE，IRP_MJ_CLOSE，IRP_MJ_DEVICE_CONTROL 三个处理事件上。

3) Nbp 的卸载过程

4) DbgClient 的卸载过程

5) Bpknock 的作用

Bpknock 程序是 nbp 的演示程序，其作用是通过调用 cpuid 这条汇编指令触发

VM Exit 条件，使得 nbp 进入 VMM 层处理异常，最后将自己定义的返回结果赋值给相应寄存器，再回到 OS 中读出该修改过的返回结果，从而证明 nbp 确实起作用了。

```
ULONG32 __declspec(naked) NBPCall (ULONG32 knock) {//自定义出入栈顺序
    __asm {
    push    ebp
    mov ebp, esp
    pushebx
    pushecx
    pushedx
    cpuid    ;要用 cpuid 触发异常陷入 VMM
    pop  edx
    pop  ecx
    pop  ebx
    mov esp, ebp
    pop  ebp
    ret
    }
}
```

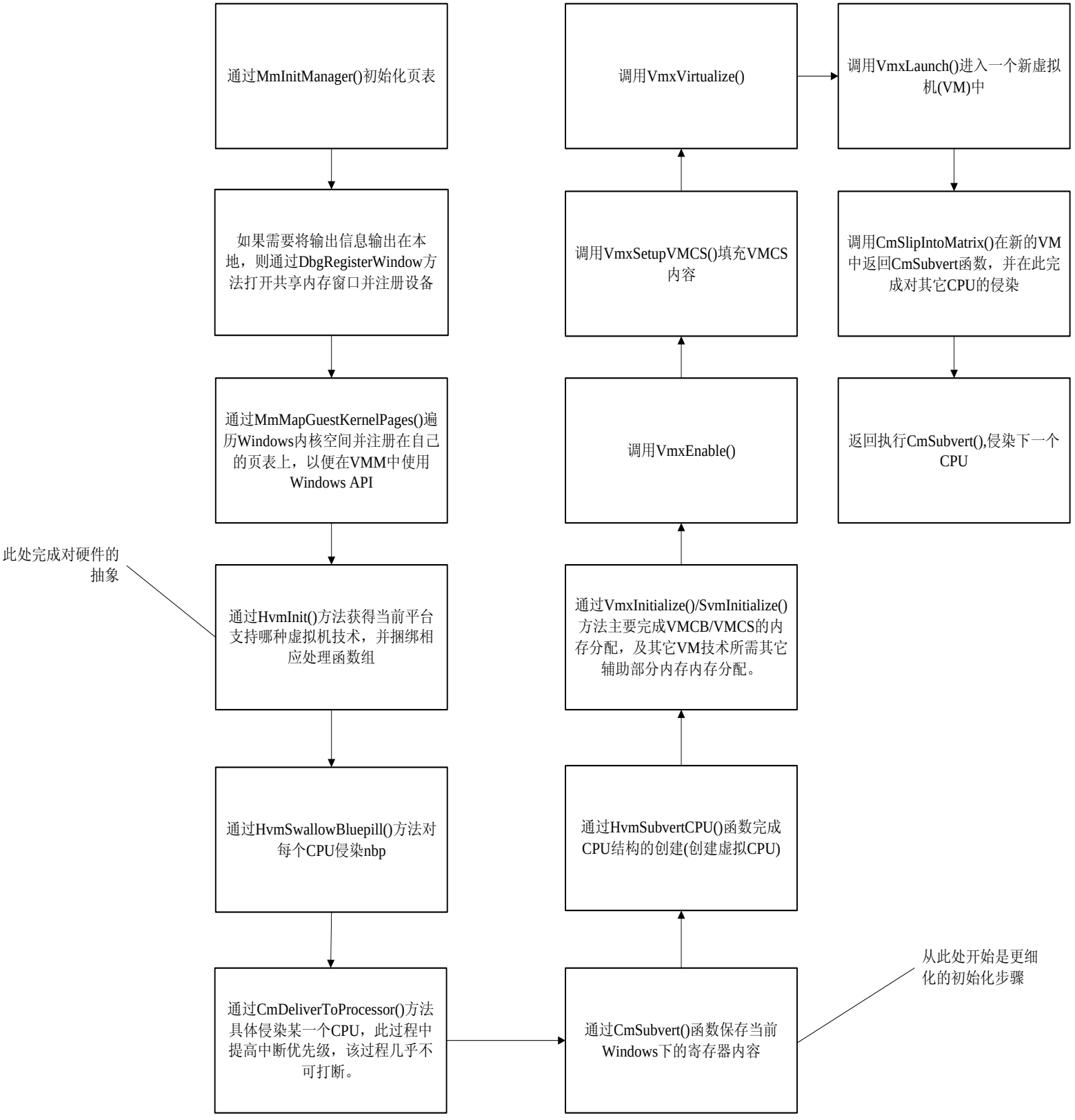

图 6.1 nbp 驱动初始化流程图

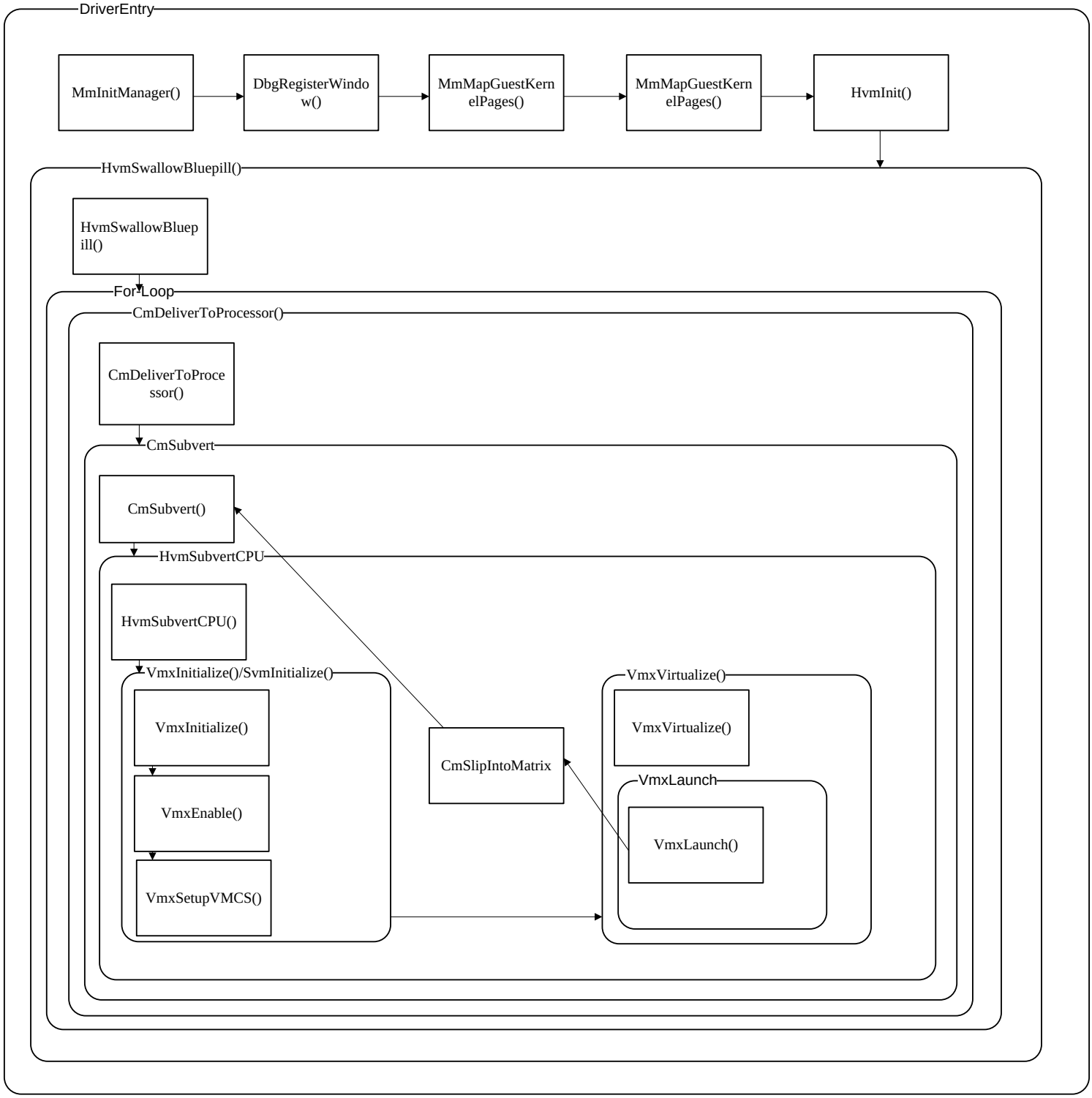

图 6.2 nbp 初始化过程函数调用图

七、 NewBluePill 硬件相关层

八、 NewBluePill 内存系统

1）相关文件：

nbp-0.32-public\common\Paging.c

nbp-0.32-public\common\Paging.h

2）技术背景：

分页机制是整个 nbp 的核心之一，称其核心是因为 nbp 所使用的很多其它部分都需要分页机制的支持，比如调试部分中调试信息的存放，还有整个 nbp 的代码段，以及 nbp 安装到每个 CPU 上后为了运行 nbp 所分配的堆栈区，这些内存（页）全部需要映射到 nbp 的分页机制上。

nbp 的分页机制是利用了 64 位系统特有的四级页表分页机制。AMD 和 Intel 对应的硬件实现基本相似。不过现在仍需定位 nbp 在哪里赋值给 EFER 和 CR4 寄存器，以及它赋的什么值，这影响到 nbp 如何启用相应的地址翻译机制。对于 AMD 的 CPU 来说，其各级页表寻址翻译总体过程如图 8.1 所示，每级 Virtual address 的地址结构要求可参考 AMD64 Architecture Programmer's Manual, Volume 2: System Programming 的第 131 页的四张图。Long Mode 的地址翻译机制还需要注意 CR3 的寄存器内容，在长模式下(Long Mode)，CR3 的寄存器内容包含如下部分（见图 8.2）：

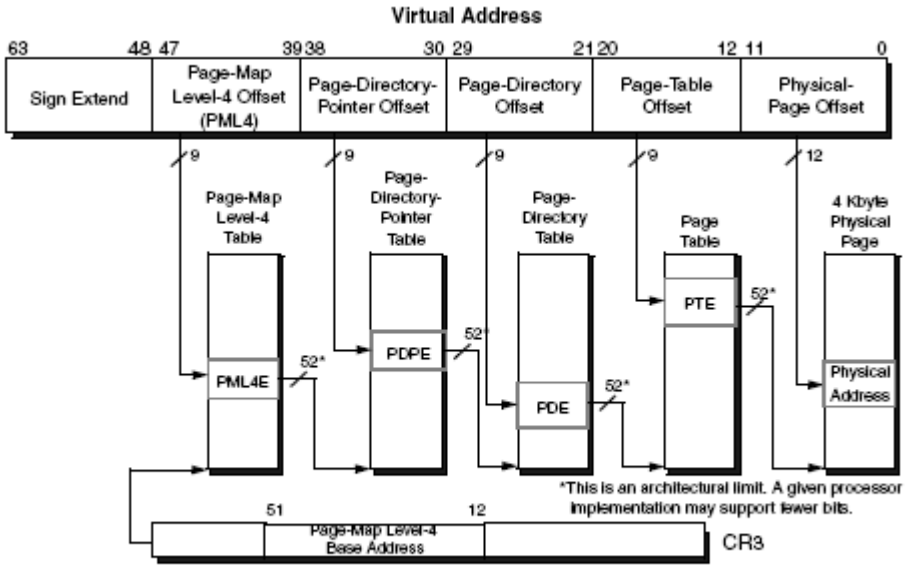

Figure 5-17. 4-Kbyte Page Translation—Long Mode

图 8.1 AMD 长模式(Long Mode)地址——4K 页翻译图

1. 页表基址域(Table Base Address Field)，Bit 51-12，这 40 位指向 PML4 页表基地

址，PML4 页表必须是页对齐的。

2. 高页写穿位（Page-Level Writethrough (PWT) Bit），Bit 3，表明最高级别的页表使用写回(Writeback)还是写穿(Writethough)策略
3. 高页可缓存位(Page-Level Cache Disable (PCD) Bit)，Bit 4，表明最高级别的页表是否可缓存。
4. 保留位(Reserved Bits)，保留位在写 CR3 寄存器时候应该清零。

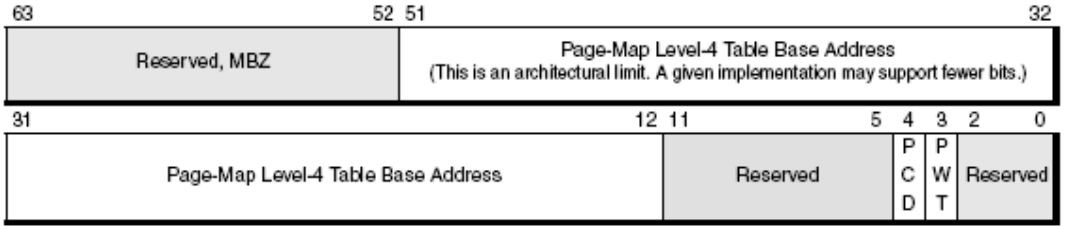

Figure 5-16. Control Register 3 (CR3)—Long Mode

图 8.2 AMD 长模式(Long Mode)中 CR3 寄存器内容

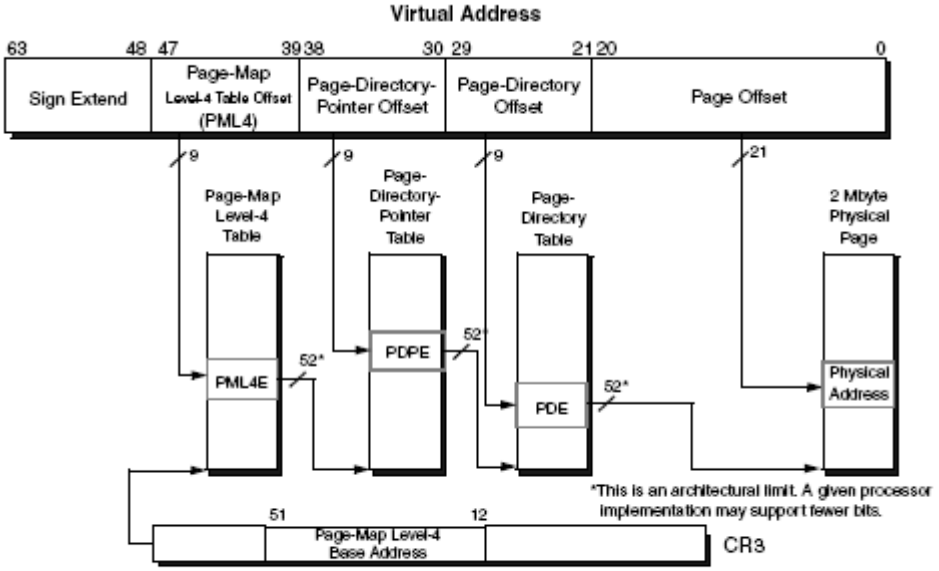

Figure 5-22. 2-Mbyte Page Translation—Long Mode

图 8.3 AMD 长模式(Long Mode)地址——2M 页翻译图

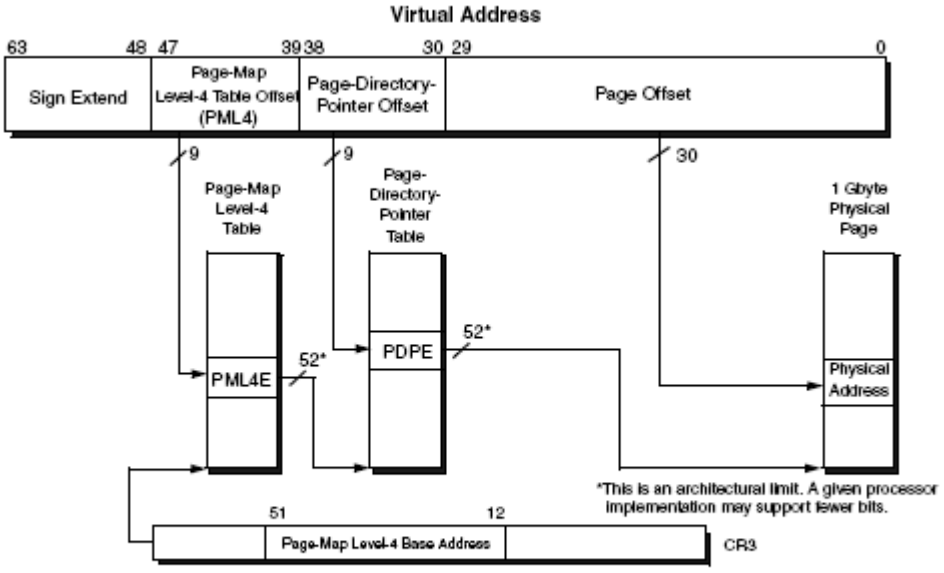

Figure 5-26. 1-Gbyte Page Translation—Long Mode

图 8.4 AMD 长模式(Long Mode)地址——1G 页翻译图

同时 AMD 也支持 2M 和 1G 页，通过分别挂载到 PD 和 PDP 即可实现，分别为 21 位寻址和 30 位寻址，比较简单因此不再叙述。（见图 8.3 和 8.4）

而对于 Intel 的 CPU 来说，这个翻译过程被称作 IA-32e Mode Linear Address Translation （IA-32e 模式地址翻译），各级页表的名称和翻译原理均十分相像，故在此仅列出 4K 页的映射过程(见图 8.5)，具体原理可参考 Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A 中第 3-42Vol3（第 126 页 Protect-Mode Memory Management）内容

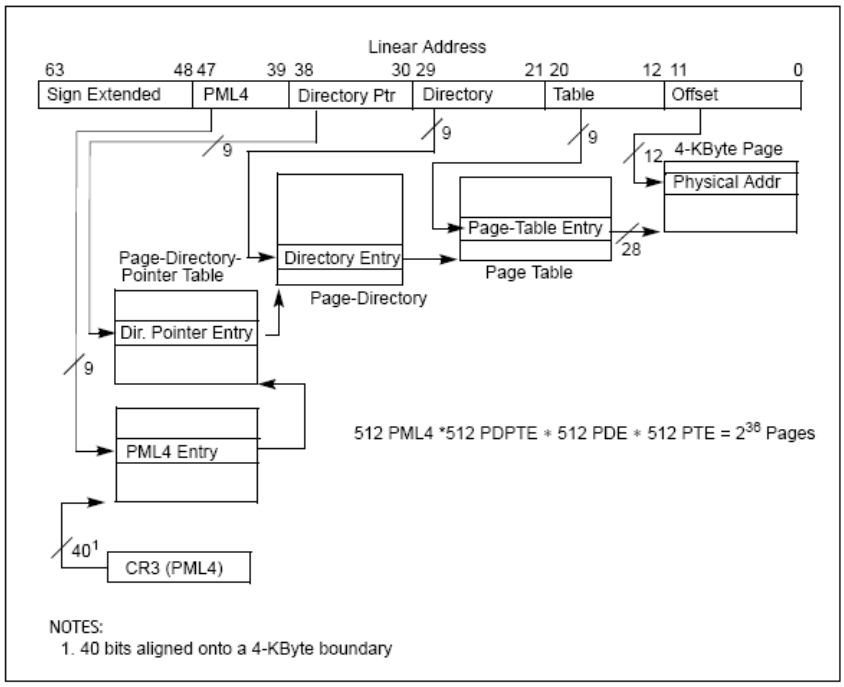

Figure 3-24. IA-32e Mode Paging Structures (4-KByte Pages)

图 8.5 Intel IA-32e 模式地址——4K 页翻译图

3）总体功能介绍：

Nbp 的分页系统要完成的总体任务：对 Windows 的内核地址分析，并按照上面所提到的虚拟机地址翻译方法创建页表，从而使得 nbp 可以在 VMM 层中仍旧可以使用 Windows API。

另一个目标是在 nbp 运行时分配自己的内存，尽管分配的方法用到了 Windows API，但是如何把新分配到的内存挂载到符合虚拟机地址翻译（Address-Translation）的页表上却是由 nbp 自己要负责的。

另外一点就是对于 Nbp 中自己产生的页表的 Host Address 和它在 Windows 中的 Virtual Address 并不相同，这是由它要创建符合虚拟机地址翻译（Address-Translation）的页表造成的，Nbp 中自己产生的页的 Host Address 确是和 Windows 中的 Virtual Address 一样，这个可能只是为了省事而已。

关于内存保护，nbp 的内存保护依赖于 Windows 操作系统。通过 Windows API 分配内存的方法使得各驱动间几乎无法访问同一块物理内存，所以 nbp 无需再实现自己的内存保护方案。

4）实现过程：

在谈了那么多背景知识后，我们看一看 nbp 是去使用页表地址翻译的。nbp 利用了硬件提供的页表做地址翻译，所以 nbp 在内存管理上主要做两件事情：

1. 根据硬件要求构建页表结构，并将需要挂载的页挂载到这个页表结构上。
2. 在需要的时候改变 CR3 寄存器内容，从而使 CPU 使用 nbp 的页表做地址翻译。（这个时间点仍需确定）（CR3 更新内容后怎么找到 nbp 各段地址以继续运行？）

MmInitManager()方法

从 Newbp.c 的 DriverEntry 方法进入，可以看到在通过 ComInit()方法设置当前运行的 bluepill 唯一标示 ID 后，立即调用了 MmInitManager()方法来构建页表，这个函数也是 nbp 内存管理部分的入口函数，其作用是确定 PML4 页表的基地址（第 621 行）并初始化从 PML4 到 PT 每级一个页表，当然此时也初始化好了他们之间的映射关系，后者过程是通过 MmCreateMapping()方法和 MmUpdatePageTable()方法递归完成的。

可以观察到在 MmInitManager 中出现了两组地址翻译的操作，一个是从 OS 的虚拟地址到内存物理地址的映射，另一个是 VM 的虚拟地址（或者说是 nbp 的虚拟地址）到内存物理地址的映射。在 nbp 中，HostAddress 指 nbp 中 VM 寻址用的虚拟地址，PhysicalAddress 指物理内存地址，而 GuestAddress 是指 OS 寻址用的虚拟地址。在后面我们可以看到，nbp 为了实现方便，把页的 GuestAddress 作为 HostAddress 直接挂在了 nbp 的页表上。

此外，nbp 为了方便管理这套页表（主要是为了方便查找和释放某页），利用 Windows 提供的双向链表将 nbp 自己所有的页表信息和页信息保存了起来并且顺序挂在这个双向链表上，实现这个过程的方法是 MmSavePage(),

这里有一个细节就是每级页表所对应的 HostAddress，依次定义如下：（定义在

\nbp-0.32-public\commoncommon.h 的第 122 行）

```
#define   PML4_BASE    0xFFFFF6FB7DBED000
#define   PDP_BASE     0xFFFFF6FB7DA00000
#define   PD_BASE      0xFFFFF6FB40000000
#define   PT_BASE      0xFFFFF68000000000
```

此处的几个地址是固定的地址，这几个地址的计算可参考参考书目一章中的[4][5][6]，此处仅列出 Windows x64 address space layout（图 8.6，512GB 4 level Page table map 行使我们所关心的，这四个地址全在那段中）

当然更进一步的搜索发现在 xen 中也有完全相同的上面四个数字，因此如果我们是开发自己的系统，可以直接拿这几个 base address 来用，常量不变。

MmSavePage ()方法

作用是将 nbp 分配出来的某个页表/页的信息保存起来并顺序挂在 g_PageTableList 这个双向链表上，这个信息被存在 ALLOCATED_PAGE 结构体中，主要记录了一个页表/页的物理地址，在 OS 中的虚拟地址（Guest Address,某些时候使用以及释放该页时要借助于 OS 的帮忙），在 nbp 内部的虚拟地址（Host Address,对于页来说这个地址就是其在 OS 中的虚拟地址），分配类型(Allocation Type),页数量（uNumberOfPages，nbp 只支持整页分配，因此这项必为整数），标志位（Flags，标记了关于该页的其它信息）

这里关于 PhysicalAddress 只取中间的 40 位，而 HostAddress 要取高 52 位的一个原因是对于 PhysicalAddress 现在的内存远远不需要用 64 位来表示，此处最细粒度是 4K 页，因此低 12 位在此处是不用存的，因为 CR3 中要求接受物理内存地址的中间 40 位，

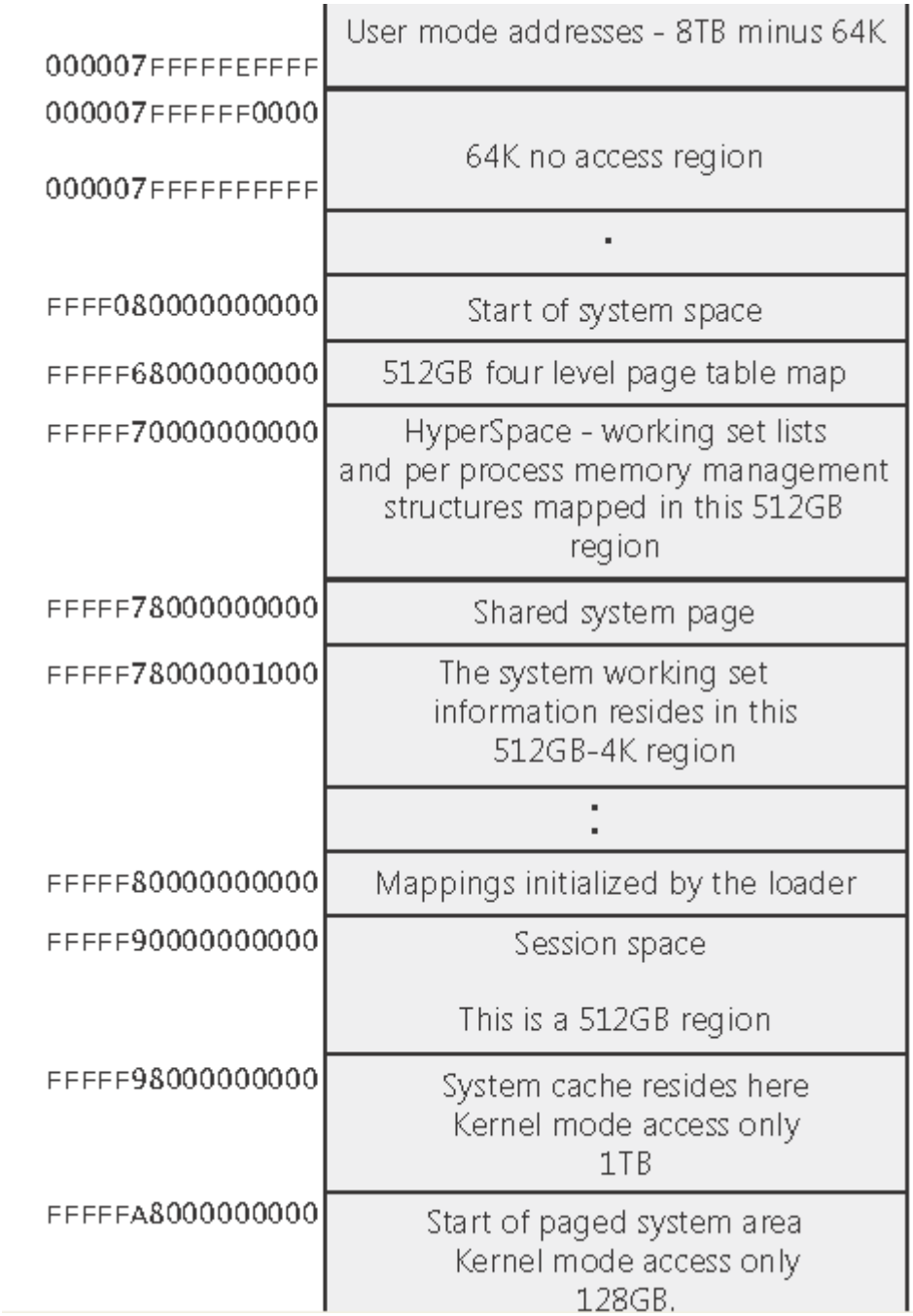

图 8.6 Windows x64 结构地址空间图(x64 address space layout)

因此只需满足 CR3 的要求即可，当然如果为了以后 nbp 也能跑起来，CR3 高端部分开几位这里也就跟着开几位就可以了。

HostAddress 高 12 位必须保留，因为规范中规定这部分是 Sign extend 第 47 位得来的

关于分配类型(AllocationType)，在 nbp 中一共有三种内存分配类型：

- PAT_POOL 同 Windows 中的 a block of pool memory.
- PAT_CONTIGUOUS 同 Windows 中的 Contiguous Memory
- PAT_DONT_FREE 分配出来的多页内存，这种表示除了第一页外其它页的类型：此空间已被使用。（没什么别的含义）

关于标志（Flags）,在 nbp 中对页表的标志有 5 个：

- AP_PAGETABLE：表明该信息指一个页表的信息
- AP_PT：该页表是 PT 级页表
- AP_PD：该页表是 PD 级页表
- AP_PDP：该页表是 PDP 级页表
- AP_PML4：该页表是 PML4 级页表

◆ MmCreateMapping ()方法

这个方法主要作用是创建 PhysicalAddress 和 VirtualAddress（也就是 HostAddress）之间的映射，同时构造在这个翻译过程中所需要的每级页表。实际上构造完的结果可以使得用 VirtualAddress 去寻址 PhysicalAddress。这也是整个系统在分配页时所使用的方法，比如 MmAllocatePages(), MmAllocateContiguousPages(), MmMapGuestPages()等都调用该方法创建 VA 到 PA（虚拟地址到物理地址）映射。

MmCreateMapping ()方法参数除了物理地址(Physical Address)和虚拟地址(Virtual Address)外，还有一个 bLargePage 的布尔类型参数，这个参数用于表明这个映射是否对应于一个大页，在 nbp 中，只有 2M 页和 4K 页两种，当 bLargePage 为 true 时，表明此时映射的是一个 2M 的大页，一般情况下为 false，表明映射 4K 页。

进入函数首先会调用 MmFindPageByPA()方法，这个方法在此处的作用是获得每个 nbp 唯一的 PML4 页表信息，其实最主要的是要获得其 OS 中的虚拟地址，因为接下来会通过这个虚拟地址借助于 OS 的帮助构建后三级页表。

在这个函数中最主要的方法是 MmUpdatePageTable()方法，它会根据虚拟地址用递归的方法创建完整的从 PML4 到 PT 各级页表，具体实现方法会在 MmUpdatePageTable()方法中介绍。同时 PhysicalAddress 和 VirtualAddress 仍各取中间的 40 位和高 52 位，此处这只是一种保证措施，具体原因可参照上文。

MmSavePage ()方法

MmSavePage ()方法