Xen 内存管理

一 .Xen 如何限制 Guest OS 的内存访问

二 . 地址转换

(PV) 直接分配

(HVM) Shadow Page Table

(HVM) EPT + VPID

三 . Xen 的实现

四 . 附录

Xen 如何限制 Guest OS 的内存访问

Xen 拥有自己独占的内存，该内存映射到 Guest OS 的线性地址空间上

- 好处：如果不在 ==》 VMM 进入和退出时，都要更新页表，刷新 TLB ==》开销太大
- X86_32(PAE) ➔ 利用分段机制
  - quad 0x00cf9a000000ffff /* 0xe008 ring 0 0→4.00GB code */
  - quad 0x00cf92000000ffff /* 0xe010 ring 0 0→4.00GB data */
  - quad 0x00cfba00000067ff /* 0xe019 ring 1 0→4GB-152M code */
  - quad 0x00cfb200000067ff /* 0xe021 ring 1 0→4GB-152M data */
  - quad 0x00cffa00000067ff /* 0xe02b ring 3 0→4GB-152M code */
  - quad 0x00cff200000067ff /* 0xe033 ring 3 0→4GB-152M data */

- Memory Layout (168M)
  - I/O remapping area (4MB)
  - Direct-map (1:1) area [Xen code/data/heap] (12MB)

  ----------__PAGE_OFFSET (0xFF000000)

  4G-16M
  - Per-domain mappings (inc. 4MB map_domain_page cache) (8MB)
  - Shadow linear pagetable (8MB)
  - Guest linear pagetable (8MB)
  - Machine-to-physical translation table [writable] (16MB)
  - Frame-info table (96MB)
  - Machine-to-physical translation table [read-only] (16MB)

CloudEx

X86_64

- [128TB, 2^47 bytes, PML4:0-255] (User).
  - 0x0000000000000000 - 0x00000000f57fffff [3928MB] Guest-defined use.
  - 0x00000000f5800000 - 0x00000000ffffffff [168MB] Read-only machine-to-phys translation table
  - 0x0000000100000000 - 0x0000007fffffffff [508GB] Unused.
  - 0x0000008000000000 - 0x000000ffffffffff [512GB PML4:1] Hypercall argument translation area.
  - 0x0000010000000000 - 0x00007fffffffffff [127TB PML4:2-255] Reserved for future use.

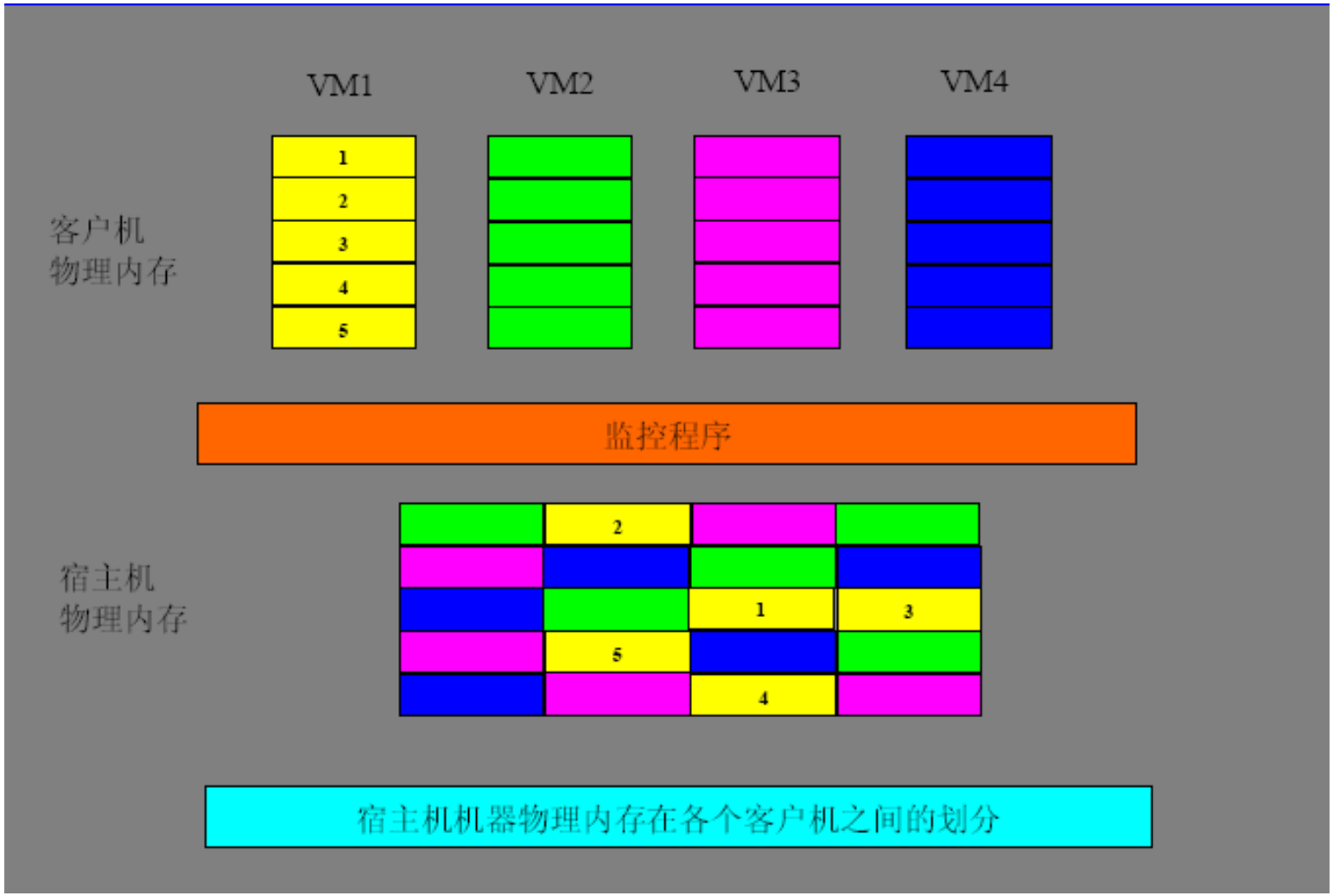
- 0x0000800000000000 - 0xffff7fffffffffff [16EB] Inaccessible: current arch only supports 48-bit sign-extended VAs.

- [128TB, 2^47 bytes, PML4:256-511] (Kernel).
  - [512GB, PML4:256]
    0xffff800000000000 - 0xffff803fffffffff Read-only machine-to-phys translation table
    0xffff804000000000 - 0xffff807fffffffff Reserved for future shared info with the guest OS
  - [512GB, PML4:257] ioremap for PCI mmconfig space
  - [512GB, PML4:258] Guest linear page table.
  - [512GB, PML4:259] Shadow linear page table.
  - [512GB, PML4:260] Per-domain mappings (e.g., GDT, LDT).
  - [512GB, PML4:261]
    0xffff828000000000 - 0xffff82bfffffffff Machine-to-phys translation table.
    0xffff82c000000000 - 0xffff82c3ffffffff ioremap()/fixmap area.
    0xffff82c400000000 - 0xffff82c43fffffff Compatibility machine-to-phys translation table.
    0xffff82c440000000 - 0xffff82c47fffffff High read-only compatibility machine-to-phys translation table.
    0xffff82c480000000 - 0xffff82c4bfffffff Xen text, static data, bss.
    0xffff82c4c0000000 - 0xffff82f5ffffffff Reserved for future use.
    0xffff82f600000000 - 0xffff82ffffffffff Page-frame information array.

---------- __PAGE_OFFSET
PML4_ADDR(262)

  - [5TB, PML4:262-271] 1:1 direct mapping of all physical memory.
  - [120TB, PML4:272-511] Guest-defined use.

地址转换

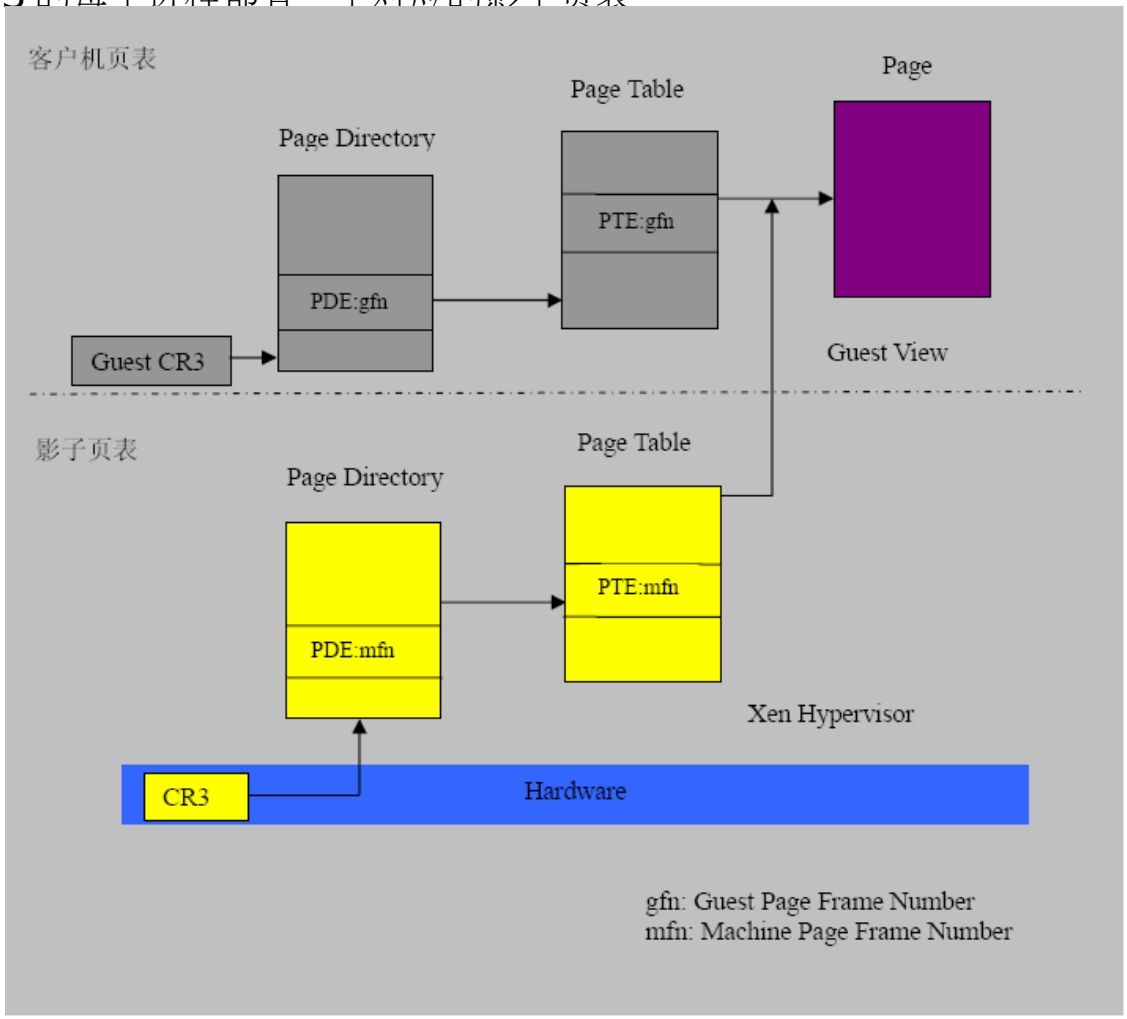

VM1
VM2
VM3
VM4
客户机
物理内存
1
2
3
4
5
监控程序
宿主机
物理内存
2
1
3
5
4
宿主机机器物理内存在各个客户机之间的划分

直接模式 (PV)

- 直接模式 (PV)
  - 页表里的内容为 HPA
  - 页表项 Guest OS 只可读，普通的页 Guest OS 可直接读写。一旦更新页表的内容引起 Page 异常。
  - 想要更新页表内容，调用相应的 Hypercall
- Xen 的实现

影子模式

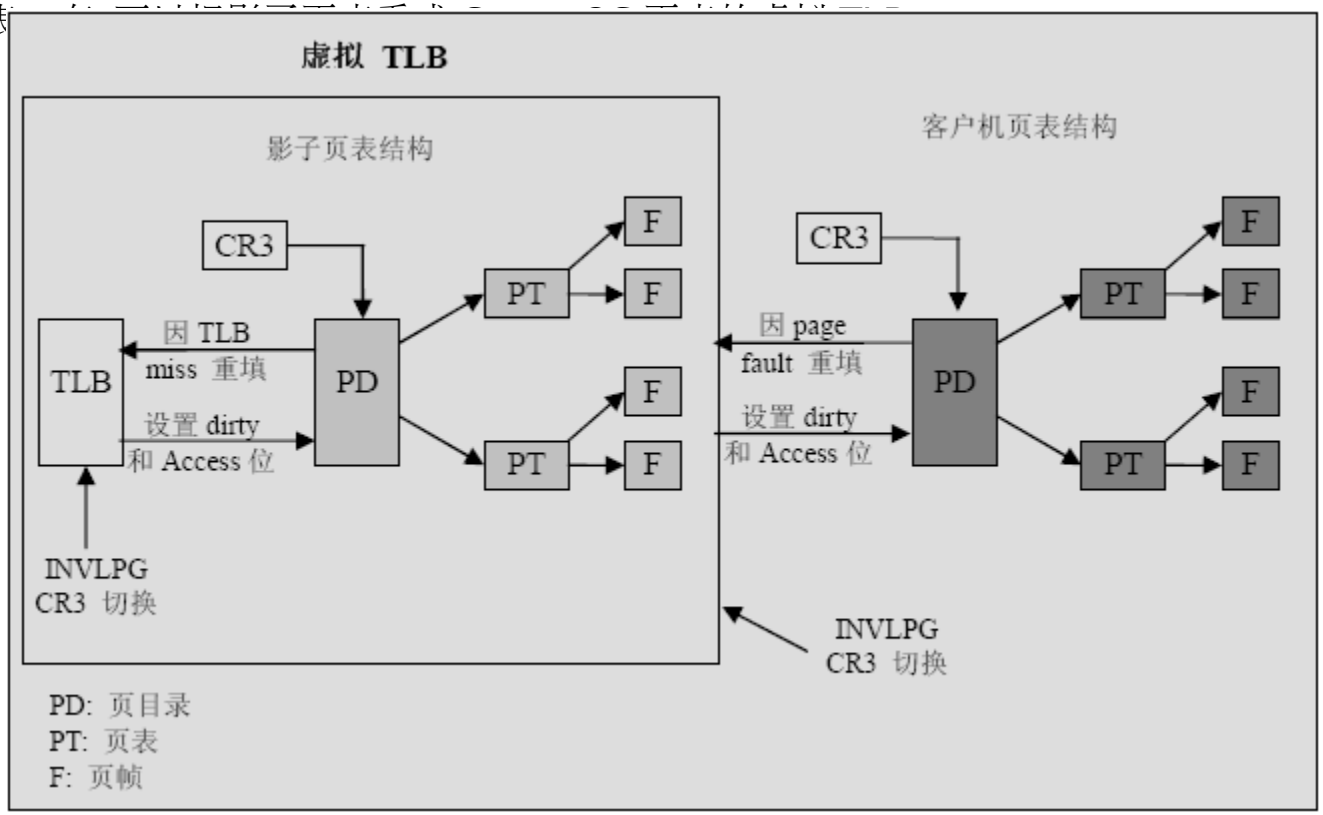
- 什么是影子页表 (DomU)
  - 实际装入物理 MMU 中的页表
  - 转换的是：从 Guest OS 的线性地址到 Host OS 的物理地址的转换
  - Guest OS 的每个进程都有一个对应的影子页表



影子页表如何工作 1

- 用什么数据建立起来的
  - P2M 表
  - Monitor Table （公用的 Xen 的地址映射的信息）
  - Hash Table(影子页表基地址 / 页目录本身地址 = hash(MFN,type)) ➔ 下次就可以直接使用了

    why MFN, not GFN?
- 什么时候刷新影子页表
  - CR3 切换
  - INVLPG 指令
  - 影子页表缺页



影子页表如何工作 2

- 捕获 GuestOS 的页表操作
  - CR3 切换时 ▪ 切换影子页表
  - INVLPG 指令时 ▪ 把该影子页删除，这样访问该页时发生影子缺页
  - 修改页表内容 ▪ 影子页表中页表页都是只读的，捕获后更新相应影子页表的内容
- 影子页表缺页处理 (建立也是通过缺页来处理的)
  - 判断是不是 GuestOS 本身引起的 存直接返回
  - 更新影子页表 (利用 P2M 表，建立 HashTable ，填充影子页表的内容)
  - 设置影子页表的访问位和修改位

Guest OS 条件			影子页表的结果			
Presen	Acces	Dirty	Presen	Access	Dirty	R/W
0	x	x	删除该项			
1	0	x	0 → Guest OS 读写该页都会被捕获，有机会设置 A			
	1	0	1	1	0	0 → Guest OS 写入该页时被捕获，有机会设置 D
	1	1	1	1	1	如果是页目录， =0 ；否则， =1

➔ 客户机页表的权限位被修改时，因为影子页表的权限 < Guest OS 页表的权限，所以能捕获该操作，有机会设置影子页表的权限位

➔ 客户机页表的内容被修改时，因为页表页在影子页表中的 R/W=0 ，当 Guest OS 更新页表时有机会同时更新影子页表

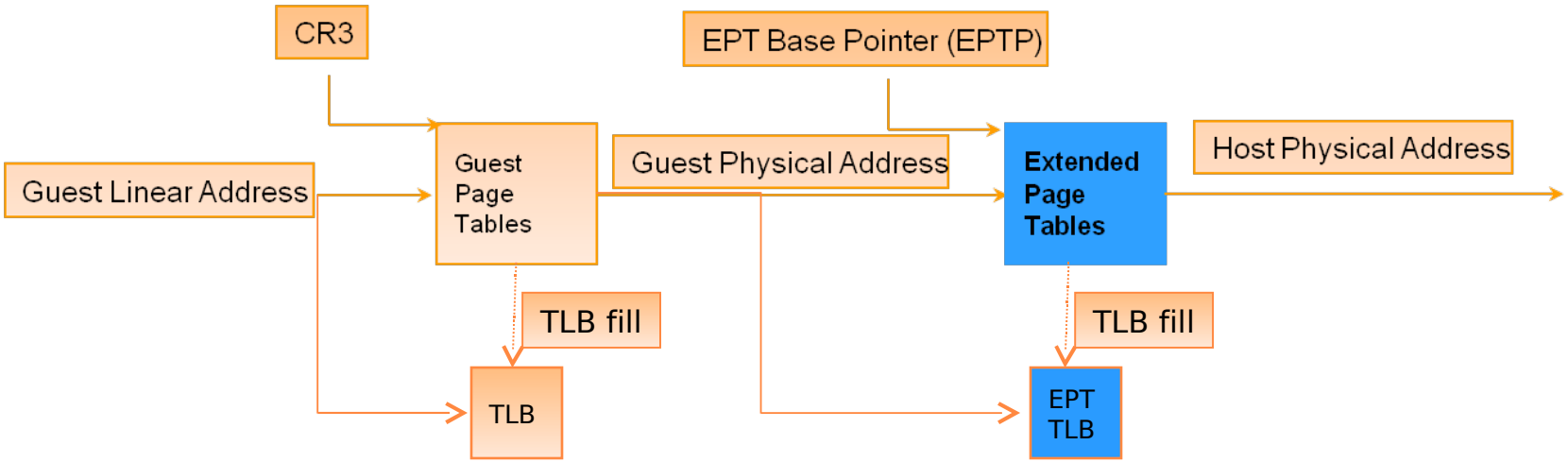
CloudEx

影子页表如何工作 3

- 如何优化
  - 问题描述：客户机切换 CR3 时要求 TLB 全部刷新 ==》影子页表全部删掉重建。
    但是，如果每次都把影子页表全部删掉重建，就会有不小的开销，而实际上刚被删掉的页表可能很快又被客户机使用到。
  - 解决方案：如果监控程序此时知道客户机页表中哪些表项被更新了，就可以只更新相应的影子页表中的表项，旧的影子页表就可以重用。这样就可以尽量避免 CR3 切换时的开销。
  - 具体方法：
    - 用 out-of-sync 链表记录当前影子页表：哪些项，最后一个更新的是“谁”
    - 不访问 out-of-sync 链表中的那些项，更新影子页表，并就把（当前要访问项，“我”）加到 out-of-sync 中
    - 要访问 out-of-sync 链表中的那些项
      - 如果发现那个“谁”正好是“我”自己，就不要刷新了 ==》 达到复用的目的
      - 如果发现这个“谁”不是“我”时，需要查看影子页表指向的地址
        - 刚好就是 “我”所求，不刷新 ==》 达到复用的目的
        - 和“我”的地址不同，刷新页表，并修改 out-of-sync 链表中的“谁” ->“我”

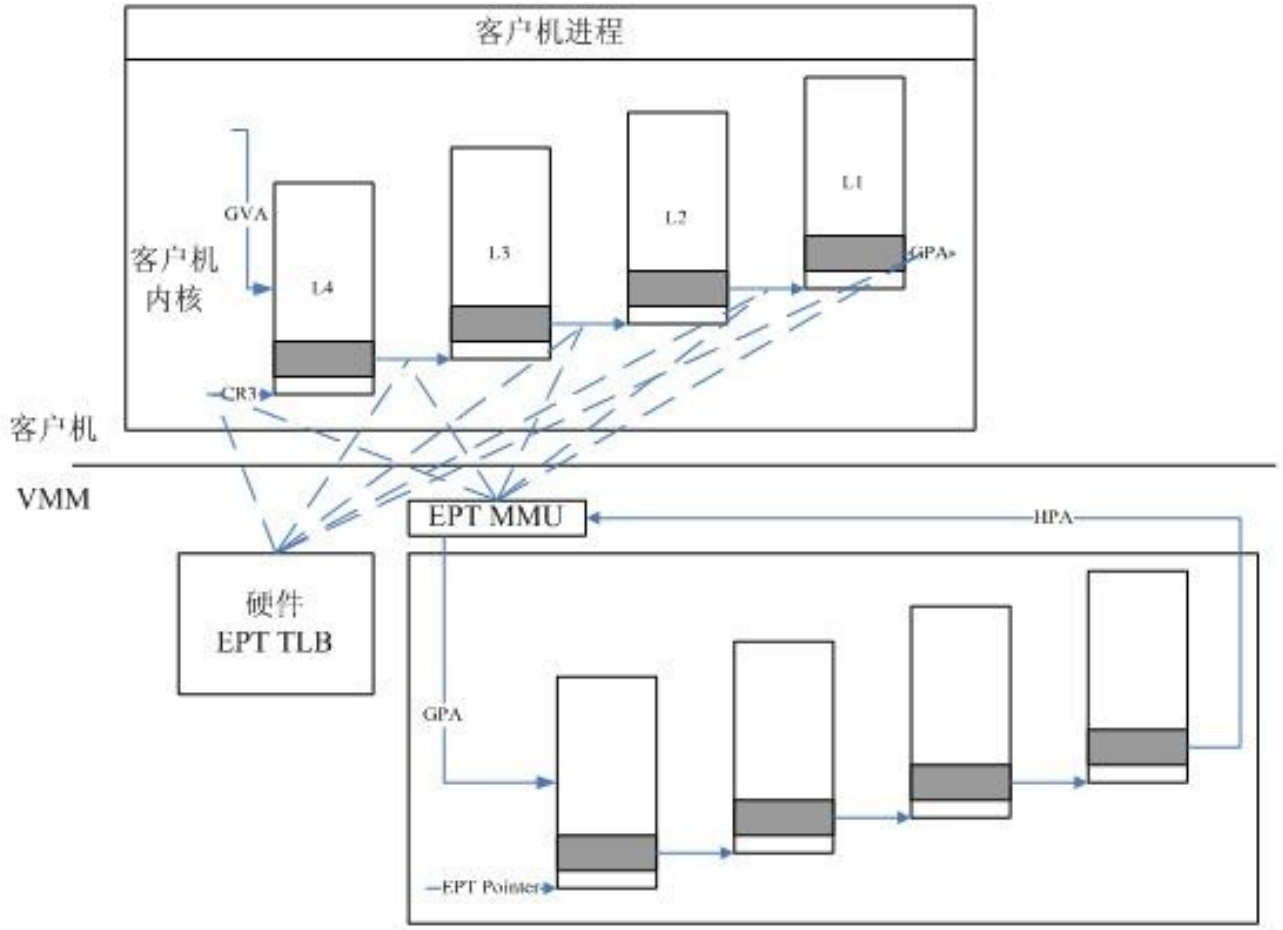
EPT + VPIDS(Intel 硬件辅助模式)

- 什么是 EPT
  - 干了什么事
    通过在内存中 VMM 为 GuestOS 维护的 EPT， 实现 GPA -> HPA 的转换
    EPT MMU 硬件完成转换
  - 解决了什么问题
    实现了 GVA -> GPA -> HPA 两次转换相互分离 ==》减轻 VMM 实现负担
  - 和影子页表比较的优缺点
    优点：客户机 Page Fault/CR3 change/ INVLPG 不会导致 VMM 陷入
    内存开销比较小（每 Guest OS 一个表）
    缺点： adds overhead to the page walk and TLB fill processes



EPT 如何工作 1

- 页表转换时 All guest-physical addresses (included CR3)，都经由 EPT MMU 自动转换
- EPT activated on VM entry(EPTP saved on VMCS)/EPT deactivated on VM exit
- 如何 Enable：
  - readmsr(curr_vcpu->arch.hvm_vmx.secondary_exec_control |= SECONDARY_EXEC_ENABLE_EPT)
  - __vmwrite(EPT_POINTER, d->arch.hvm_domain.vmx.ept_control.eptp)

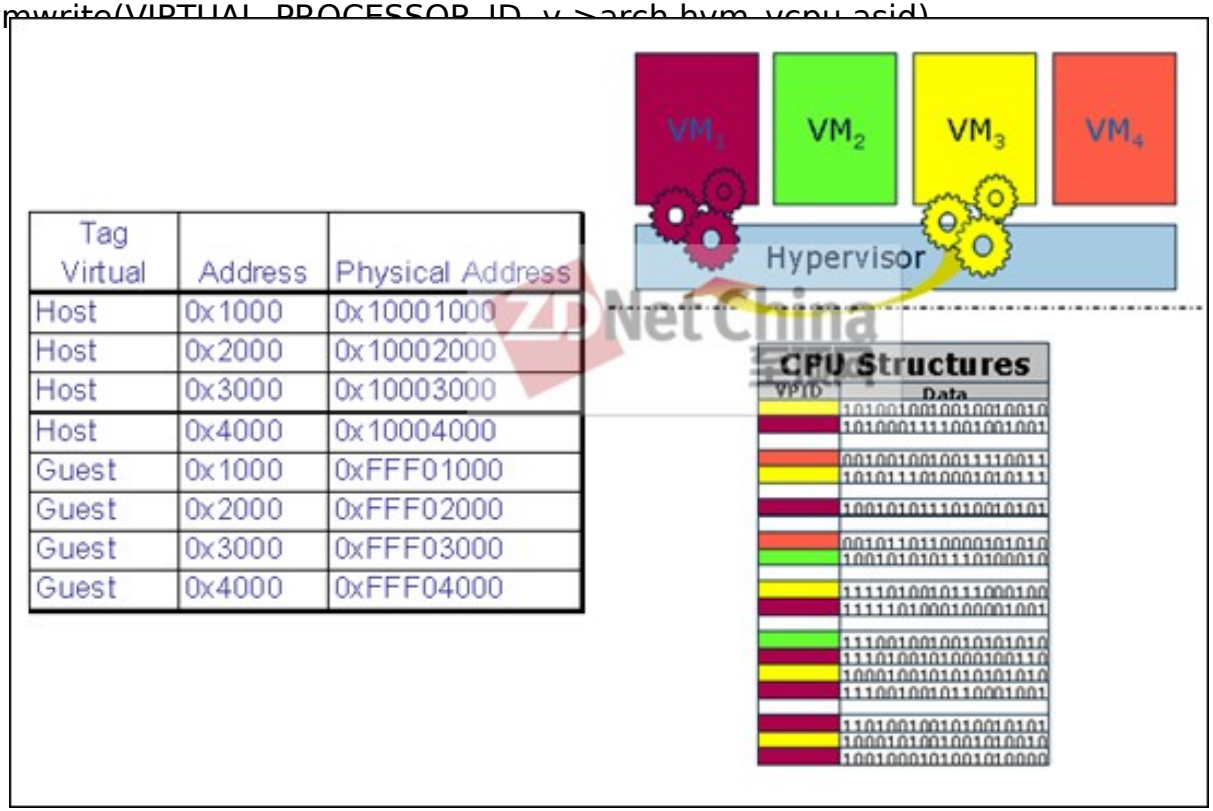


EPT 如何工作 2

- 用什么数据建立起来的
  ept 表就是 p2m 表
- 如何建立起来的
  采用懒惰算法， Violation 时自动建立
- 处理 EPT Violation == ept_handle_violation
  vmx_vmexit_handler ==》 EPT Violation 发生在 VM-Exit 时
  |--...
  |--case EXIT_REASON_EPT_VIOLATION:
  |--ept_handle_violation()
  | |--if(qualification & EPT_GLA_VALID){
  | |--hvm_hap_nested_page_fault(gfn)}
  | | |--mfn = gfn_to_mfn_type_current(gfn)
  | | |--1.If this GFN is emulated MMIO or marked as read-only,
  | | |-- pass the fault to the mmio handler
  | | |--2.Check if the page has been paged out
  | | |-- ==>p2m_mem_paging_populate
  | | |--3.Mem sharing: unshare the page and try again

什么是 VPIDS(Virtual Processor identifiers)

- 什么是 VPIDS： TLB 中使用的 VCPU 的 Indentify
- 有什么好处： Without VPIDS, VM-Entry/VM-Exit 时下次可能使用的“页表转换结果”被 refresh 了
- 同一个 Domain 的 VPID 是相同的
- VMM 使用步骤:
  - 设置启动选项 vpid
  - readmsr(curr_vcpu->arch.hvm_vmx.secondary_exec_control |= SECONDARY_EXEC_ENABLE_VPID)
  - __vmwrite(VIRTUAL_PROCESSOR_ID, v->arch.hvm_vcpu.asid)



CloudEx

Xen 如何工作 (EPT/Shadow Page Table)

- Domain 内存管理
  - 表示
    - domain-> tot_pages ➔ 现在使用了多少内存 (GuestOS 使用的 + Xen 使用的)
    - domain-> max_pages ➔ 最多可以使用多少内存 xc_domain_setmaxmem()
    - Pool ➔ can be called xc_shadow_control(set_alloc)， Xen 使用的内存
      - d->arch.paging.xxx.total_pages ++
      - d->arch.paging.xxx.free_pages++
    - p2m 表 ▪ 内容是所有 Guest OS 可见的内存，但表本身的 page 从 Pool 来
  - populate
    - Guest OS 在运行过程中申请内存
    - populate_physmap() = alloc_domheap_pages + set_p2m_entry
    - Xen 在运行过程中申请内存 (比如建立 Shodow Pool/EPT Pool) → alloc_domheap_pages()
    - 见附录 1
- P2M 表
  - 基地址 ▪ d->arch.phys_table
  - allocate → p2m_alloc_table
  - set_entry → set_p2m_entry
  - get_entry →
  - 见附录 2 、附录 3

CloudEx

Xen 如何工作 2

- EPT Pool
  - 如何建立
    - hap_set_allocation
  - 有什么用途
    - hap_make_monitor_table
    - create p2m
- EPT 的 P2M 表
  - 提供 P→M 的转换接口
  - 支持 >4K 的页 (super_page)
  - 页表页是 EPT Pool 里的 page

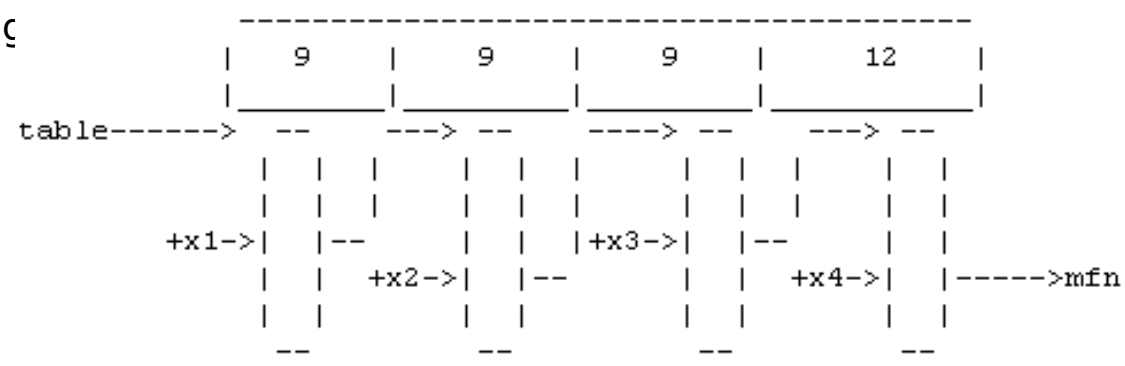
  hap_alloc_p2m_page

  |--d->arch.paging.hap.free_pag

  |--d->arch.paging.hap.total_pa

  |--d->arch.paging.hap.p2m_pa

```
typedef union {
    struct {
        r           :   1,
        w           :   1,
        x           :   1,
        emt         :   3, /* EPT Memory type */
        ipat        :   1, /* Ignore PAT memory type */
        sp_avail    :   1, /* Is this a superpage? */
        avail1      :   4,
        mfn         :   40,
        avail2      :   12;
    };
    u64 epte;
} ept_entry_t;
```



CloudEx

Xen 如何工作 3

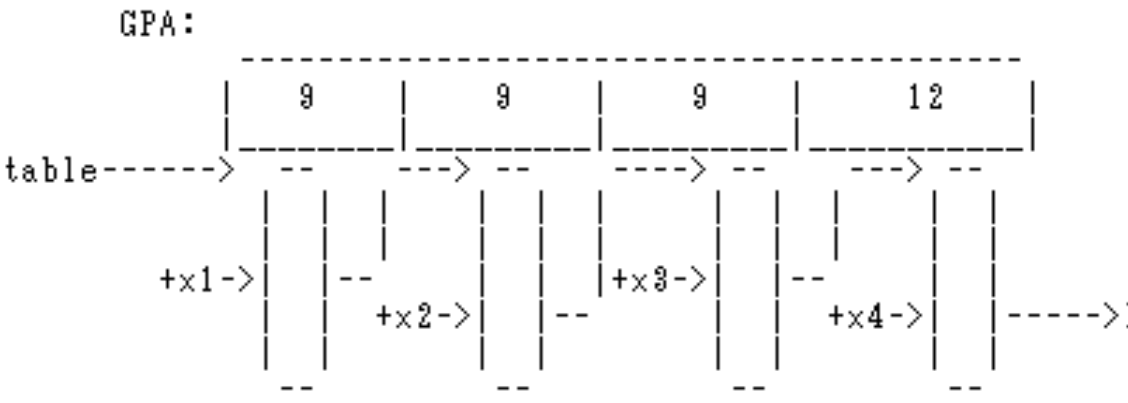
- ST 的 shadow pool
  - 如何建立
    - sh_set_allocation
  - 有什么用途
    - #define SH_type_xxx_shadc
    - #define SH_type_p2m_table
    - #define SH_type_monitor_ta
    - #define SH_type_oos_snaps
- ST 的 P2M 表
  - 提供 P→M 的转换接口
  - 辅助影子页表的工作
- ST 的 Hash table

  table = xmalloc_array(struct page_info *, 251);

  d->arch.paging.shadow.hash_table = table;
- ST 的 shadow page table

  见 Page8

```
typedef u64 intpte_t;
typedef struct { intpte_t l1; } l1_pgentry_t;
typedef struct { intpte_t l2; } l2_pgentry_t;
typedef struct { intpte_t l3; } l3_pgentry_t;
typedef struct { intpte_t l4; } l4_pgentry_t;
typedef l4_pgentry_t root_pgentry_t;
```



CloudEx

Xen 如何工作 4

- Live Migration
  - populate_physmap

    申请内存 动按照自己的要求建立 P2M 表
  - log_dirty_domain

    hap_enable_log_dirty

    |--d->arch.paging.mode |= PG_log_dirty

    |--p2m_change_entry_type_global(d, p2m_ram_rw, p2m_ram_logdirty)

    |--|--entry->r = entry->x = 1;entry->w = 0;

    shadow_enable_log_dirty

    |--if(shadow_mode_enabled(d))

    |--| shadow_blow_tables(d)

    | | |--unpin all pinned pages

    | | |--unhook entries of in-use shadows --> set l4 entry empty

    |--}

    |--shadow_one_bit_enable(d, PG_log_dirty)

    |--...

    |--update_cr3

CloudEx

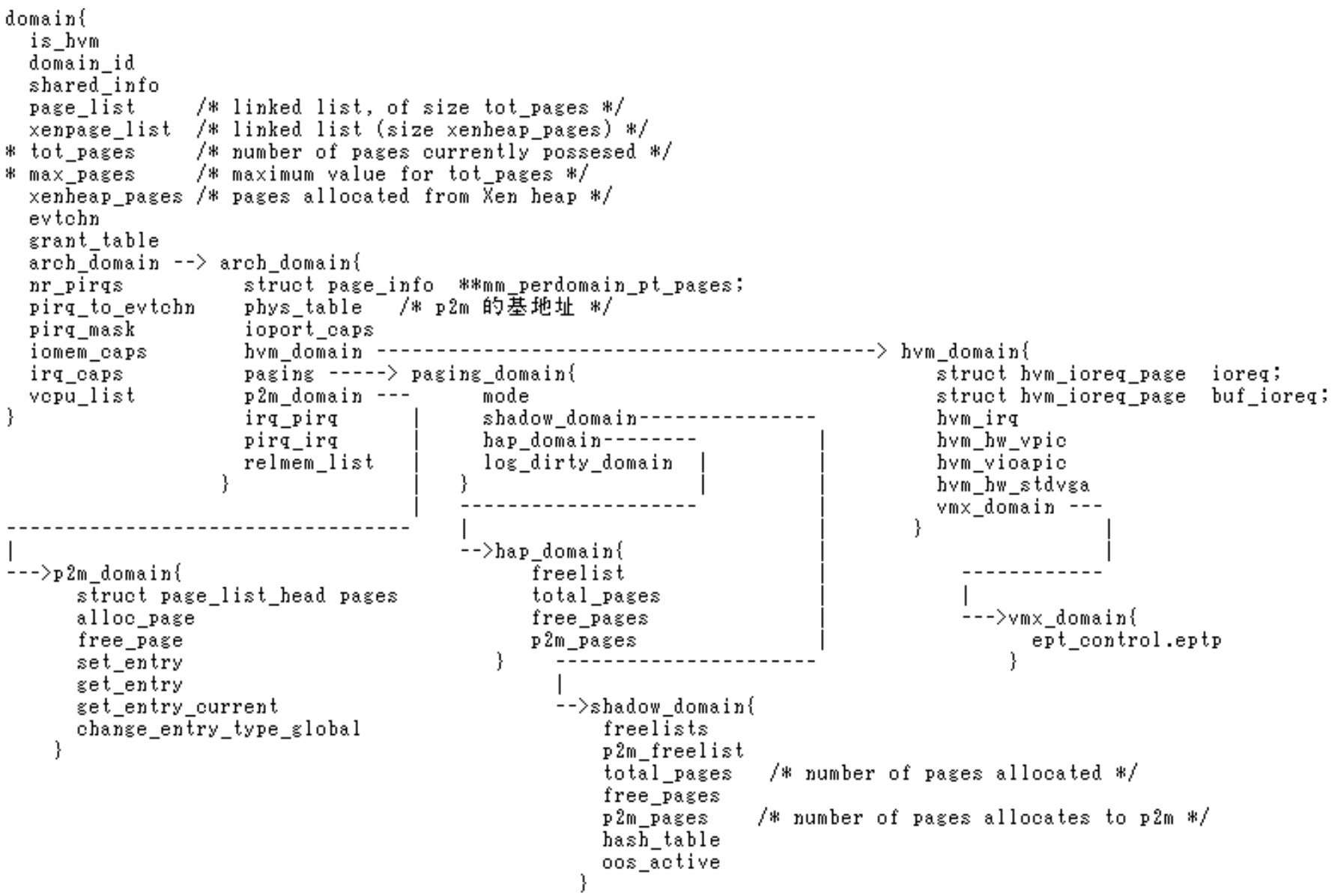
附录 1-- 内存管理

XC 分配内存

```
xc_domain_memory_populate_physmap
|--populate_physmap
|  |--page = alloc_domheap_pages()
|  |--guest_physmap_add_page() -- guest_physmap_add_entry
|  |  |--for(){
|  |  |--set_p2m_entry(j)
|  |  |  |--d->arch.p2m->set_entry() -->  建立 p2m 表项
|  |  |--}
```

```
alloc_domheap_pages  -->  真正的给 domain 分配内存的函数
|--pg = alloc_heap_pages(zone,node,num)
|--assign_pages
|  |--d->tot_pages += num
|  |--for(i-->num){
|  |--page_set_owner()
|  |--page_list_add_tail(&pg[i], &d->page_list)
|  |--}
```

附录 2-- 数据结构



CloudEx

附录 3—create HVM on VMM

```
do_domctl(HVM)
|--domain_create
|   |--XSM_create
|   |--evtchn_init
|   |--grant_table_create                          (create Per-domain grant table)
|   |   |--Active grant table                      used by xen
|   |   |--Tracking of mapped foreign frames table  other's pages
|   |   |--Shared grant table                      my pages
|   |--arch_domain_create
|   |   |--d->arch.hvm_domain.hap_enabled = is_hvm_domain(d) && hvm_funcs.hap_supported && (domcr_flags & DOMCRF_hap);
|   |   |--paging_domain_init
|   |   |   |--p2m_init
|   |   |   |   |--d->arch.p2m->set_entry = p2m_set_entry;
|   |   |   |   |--d->arch.p2m->get_entry = p2m_gfn_to_mfn;
|   |   |   |   |--if ept_p2m_init
|   |   |   |   |   |--d->arch.p2m->set_entry = ept_set_entry;
|   |   |   |   |   |--d->arch.p2m->get_entry = ept_get_entry;
|   |   |   |--shadow_domain_init
|   |   |   |   |--paging_log_dirty_init
|   |   |   |--hap_domain_init
|   |   |   |   |--d->arch.paging.hap.freelist
|   |   |--d->arch.ioport_caps = rangeset_new(d, "I/O Ports")
|   |   |--d->shared_info = alloc_xenheap_pages()
|   |   |--share_xen_page_with_guest()
|   |   |--d->arch.pirq_irq = xmalloc_array()
|   |   |--d->arch.irq_pirq = xmalloc_array()
|   |   |--for(i=1;platform_legacy_irq(i);++i)IO_APIC_IRQ(i);
|   |   |--hvm_domain_initialise
|   |   |   |--paging_enable
|   |   |   |   |--hap_enable
|   |   |   |   |--or
|   |   |   |   |--shadow_enable
|   |   |   |--vpic_init
|   |   |   |--vioapic_init
|   |   |   |--stdvga_init
|   |   |   |--rtc_init
|   |   |   |--hvm_init_ioreq_page(d, &d->arch.hvm_domain.ioreq)
|   |   |   |--hvm_init_ioreq_page(d, &d->arch.hvm_domain.buf_ioreq)
|   |   |   |--register_portio_handler(d, 0xe9, 1, hvm_print_line)
|   |   |   |--hvm_funcs.domain_initialise
|   |--d->iomem_caps = rangeset_new(d, "I/O Memory")
|   |--d->irq_caps   = rangeset_new(d, "Interrupts")
|   |--sched_init_domain
```

thank you grazie **merci** danke **grazias** 謝謝 спасибо
gracias **obrigado** ありがとう **dank** takk **bedankt** dakujem