Xen 分析

余上

2008-02-13

内容目录

第一章 总体结构

Xen是一个开源的虚拟化管理软件,用于将硬件虚拟化,呈现给上层系统一个和真实处理器一样的"软"处理器,也即虚拟机.可以创建的虚拟机个数理论上是无限的,因此,使用Xen能够很好的利用底层硬件的计算能力.例如,可以在一个硬件平台上同时运行多个操作系统,用户可以控制每个操作系统分配的资源,处理器时间等.下面是Xen总体结构的一个示意图:

<table>
<tr><td>客户系统</td><td colspan="2">Linux,Windows...</td></tr>
<tr><td rowspan="2">Xen</td><td>paravirtulization</td><td>hvm</td></tr>
<tr><td colspan="2">Xen core</td></tr>
<tr><td>硬件层</td><td colspan="2">x86,powerpc...</td></tr>
</table>

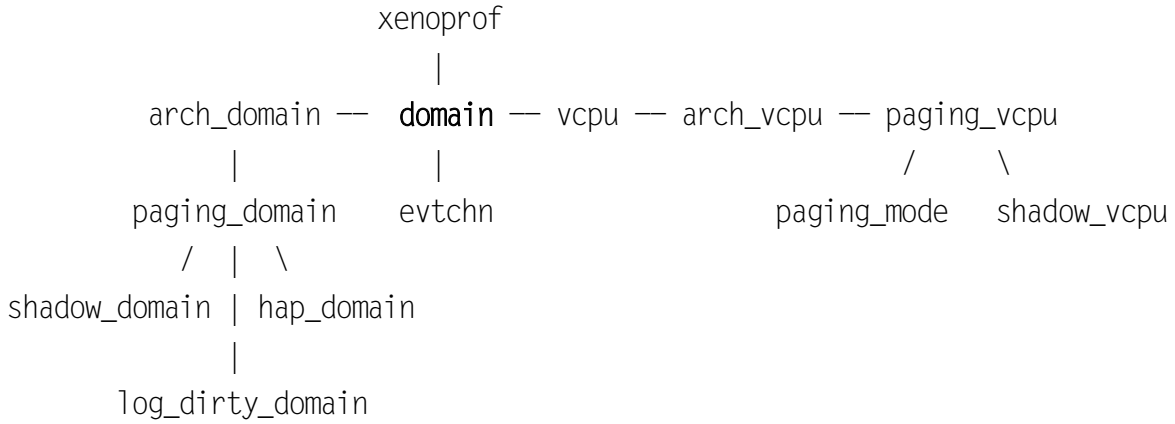
如上图所示,Xen支持两种虚拟化方式,一种叫作paravirtulization,这种方式下,Xen通过提供一套称之为hypercall的系统调用接口来实现虚拟.这套接口是非常底层的面向处理器的接口.现有的系统,例如Linux等,要进行必要的移植才能运行在paravirt环境中.而运行在操作系统上的应用软件无需修改.

另一种方式称之为hvm(hardware virtual machine),这是基于x86架构下的VMX(INTEL)或者SVM(AMD)技术的一种虚拟化,hvm利用了处理器内建的虚拟化支持.运行在hvm上的客户系统无需任何修改就可以运行在Xen上,也就是说hvm对客户系统是完全透明的.

客户系统,是指运行在虚拟机上的代码,一般都是操作系统级的软件.Xen在初始化完毕后会运行一个指定的客户系统,这是Xen的第一个客户系统,一般称之为dom0.其它的客户系统统称为domU.

安全模型,由于Xen是运行在硬件上的一个管理层软件,必须拥有硬件的全部控制权,因此Xen运行在x86的ring0层,dom0可以选择运行在ring0或者ring1,domU只能运行在ring3中.

第一节 主要对象



1)domain

domain,域,是Xen的中心概念,一个域可以看作为物理计算机系统的虚拟,具有如下的属性:

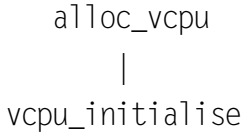
- ✔ 每个域都有一个ID,下面ID是特殊的域

ID	说明
0	Xen初始化后默认运行的域,一般称之为dom0,使用paravirt虚拟技术.运行其中的系统可以用于提供一个存取Xen功能的介面
IDLE_DOMAIN_ID	空闲域

- ✔ 是否hvm使能,这个属性决定了运行在域中的客户系统将采用何种虚拟化,如果是hvm使能,意味着使用hvm虚拟,否则使用paravirt虚拟.

  域是否hvm使能由创建函数domain_create参数:domcr_flags决定,目前这个参数的值为0或者DOMCRF_hvm,使用后者将创建hvm使能的域

- ✔ 每个域可以管理最多MAX_VIRT_CPUS个虚拟机.每个虚拟机需要调用alloc_vcpu来初始化,vcpu的初始化要经过几个层次:



/ \
hvm vcpu初始化 paravirt vcpu初始化

一个域实际分配的vcpu应该等于物理处理器的个数,一个多vcpu的域可以看作为对smp系统的虚拟,单vcpu的域可以看作为up系统的虚拟

2)vcpu

虚拟机,Xen用vcpu结构来代表一个虚拟机.vcpu是物理处理器的虚拟,其中最重要的部份就是相当于物理处理器的寄存器的虚拟机的状态了.例如,libxc中在创建好一个domU之后,就会调用hypercall设置虚拟机状态,包括用户寄存器值,页表寄存器值,domU入口地址等等.通过libxc创建的domU都运行在vcpu0上.

用户在配置domU时可以设定该domU中的vcpu个数,dom0的vcpu数由dom0_max_vcpus命令行参数决定.

idle_vcpu数组,这是一个NR_CPUS大小的vcpu数组,每个物理处理器在初始化时会分配一个空闲vcpu,用于运行空闲进程.所有的空闲vcpu都属于空闲域.

3)arch_vcpu

arch_vcpu提供vcpu体系相关部份的接口,其中非常重要的三个接口函数是ctxt_switch_from和ctxt_switch_to以及schedule_tail,这三个接口用于在VM切换时完成体系相关的上下文保存,恢复和最终的切换.目前系统支持paravirt,hvm两种体系,因此也有三个(hvm下两个子体系)这样的接口:

体系	子体系	ctxt_switch_from	ctxt_switch_to	schedule_tail
paravirt		paravirt_ctxt_switch_from	paravirt_ctxt_switch_to	continue_nonidle_domain, continue_idle_domain
hvm	vmx	vmx_ctxt_switch_from	vmx_ctxt_switch_to	vmx_do_resmue
	svm	svm_ctxt_switch_from	svm_ctxt_switch_to	svm_do_resume

这样VM切换的基本逻辑为:

1. ctxt_switch_from(p)
2. ctxt_switch_to(n)
3. schedule_tail(n)

第二章 初始化

第一节 第一部份

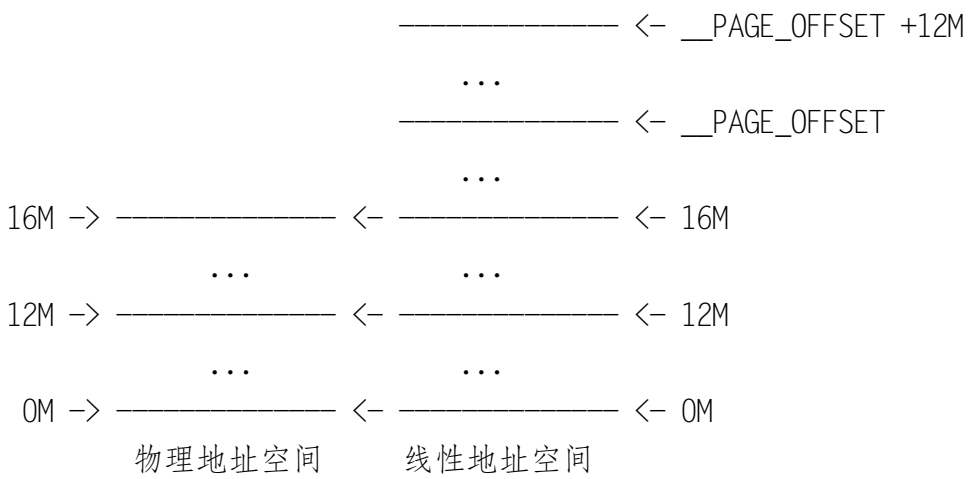
这部份是指进入C部份的__start_xen函数之前的初始化过程,主要由head.S, trampoline.S, x86_32.S几部份组成:

一,head.S,主要工作为

- 装入GDT(trampoline_gdt)

GDT项	说明	段选择子
1	ring0,code,32-bit mode	BOOT_CS32(0x0008)
2	ring0,code,64-bit mode	BOOT_CS64(0x0010)
3	ring0,data	BOOT_DS(0x0018)
4	real-mode code	BOOT_PSEUDORM_CS(0x0020)
5	real-mode data	BOOT_PSEUDORM_DS(0x0028)

- 进入保护模式
- 初始化页表,将线性空间的0-12M和__PAGE_OFFSET-__PAGE_OFFSET+12M都映射到物理地址的0-12M;而将线性空间的12M-16M映射到物理地址的12M-16M(注意,这时并没有启用分页机制):



- 解析早期命令行参数
- 调整trampoline.S代码的内存位置,移动到BOOT_TRAMPOLINE(0x8c00处)
- 跳转到trampoline_boot_cpu_entry

二,trampoline.S,主要工作为

- 进入实模式,读取内存,磁盘,视频信息

- 再次进入保护模式

  装入新的GDT(gdt_table)

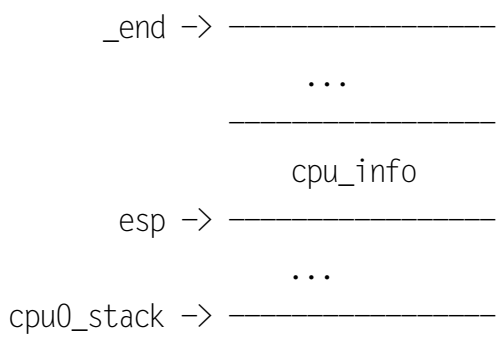
GDT项	说明	段选择子
0xe001	ring0,code,base 0,limit 4G	__HYPERVISOR_CS(0xe008)
0xe002	ring0,data,base 0,limit 4G	__HYPERVISOR_DS(0xe010)
0xe003	ring1,code,base 0,limit 3.xxG	FLAT_RING1_CS(0xe019) FLAT_KERNEL_CS
0xe004	ring1,data,base 0,limit 3.xxG	FLAT_RING1_DS(0xe021) FLAT_RING1_SS FLAT_KERNEL_DS FLAT_KERNEL_SS
0xe005	ring3,code,base 0,limit 3.xxG	FLAT_RING3_CS(0xe02b) FLAT_USER_CS
0xe006	ring3,data,base 0,limit 3.xxG	FLAT_RING3_DS(0xe033) FLAT_RING3_SS FLAT_USER_DS FLAT_USER_SS

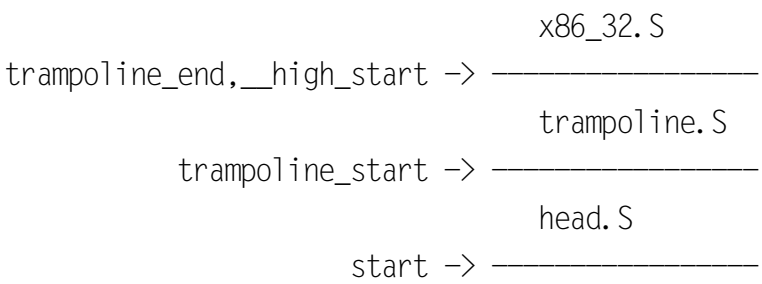
  加载前面初始化了的页表,启用分页机制,跳转到__high_start

三,x86_32.S,这个文件提供__high_start入口,主要工作为

- 装入堆栈指针,注意,Xen会在栈顶分配一个cpu_info结构(参见下图),这个结构包含很多重要的成员:1)客户系统的切换上下文2)当前运行的vcpu指针3)物理处理器编号
- IDT的处理,整个idt_table的向量入口都初始化ignore_int,这个中断处理函数打印"Unknown interrupt (cr2=XXXXXXXX)"信息后系统进入循环
- 如果是BSP,跳转到__start_xen否则,跳转到start_secondary

四,内存映像(用虚拟地址标注)





第二节 __start_xen

这个函数进一步初始化 Xen 系统,主要逻辑如下:

```
void __init __start_xen(multiboot_info_t *mbi)
{
  //注意,默认的情况下,参数 mbi 将从堆栈传递,这个值是前面汇编代码中的 ebx 值
  //解析命令行
  //初始化 console
  //整理内存信息
  //为 dom0 模块保留内存,并更新页表
  //初始化 boot allocator.这是一个内存分配器.在这之后,end_boot_allocator 调用之前的初始化都可以使用这个分配器来分配内存
    //初始化堆分配器
  //end_boot_allocator,结束 boot allocator
  //early_boot = 0; 标志"早期"初始化结束
  //trap_init,初始化 IDT
  //smp_prepare_cpus,AP 的初始化
  //do_initcalls
  //创建 dom0
  //domain_unpause_by_systemcontroller(dom0),调度 dom0
  //reset_stack_and_jump(init_done),Xen 进入 idle 循环
}
```

第三节 AP 初始化

__start_xen 通过 smp_prepare_cpus 函数开始 AP 的初始化过程:

```
void __init smp_prepare_cpus(unsigned int max_cpus)
```

```
{
    for every APs
    {
       do_boot_cpu
       {
            prepare_idle_vcpu //为 AP 准备 vcpu
            wakeup_secondary_cpu(apicid, start_eip)
                                    //start_eip:trampoline_realmode_entry
       }
    }
}
```

AP 在接收到 SIPI 消息后从 trampoline.S 的 trampoline_realmode_entry 开始执行,最后进入 start_secondary:

```
void __devinit start_secondary(void *unused)
{
    unsigned int cpu = booting_cpu;
    set_processor_id(cpu);
    set_current(idle_vcpu[cpu]);
    this_cpu(curr_vcpu) = idle_vcpu[cpu];
    ...
    startup_cpu_idle_loop();
}
```

第三章 调度

Xen是一个VMM,即虚拟机监视器,它的一个重要功能就是调度不同的虚拟机到处理器上运行.Xen调度的基本方法是虚拟机按时间片运行.

调度器接口决定下一个可调度的虚拟机以及运行的时间片大小的功能,Xen把这部份功能抽象为调度器.用户可以根据自己的需要实现不同的调度器,从而实现不同的调度策略.

第一节 调度器接口

Xen用调度器接口struct scheduler来代表这个调度器,每个调度器都需要填充这样一个结构并向Xen注册,结构的重要成员如下:

成员	说明
name	调度器名称,目前系统中实现了两个调度器:sedf和credit,前者已经过时
init	调度器初始化
init_domain	在域创建时(sched_init_domain),这个回调函数负责设置域中调度器相关的数据结构
destroy_domain	在域析构时(sched_destroy_domain)
init_vcpu	在创建一个新的vcpu时(sched_init_vcpu),这个函数负责设置vcpu中调度器相关的数据结构
destroy_vcpu	在vcpu析构时(sched_destroy_vcpu)
sleep	在sched_sleep_nosync中调用
wake	在vcpu_wake中调用,这个函数一般会调用schedule
do_schedule	在schedule中调用
pick_cpu	在vcpu_migrate中调用
adjust	在sched_adjust中调用
dump_setting	在dump_runq中调用
dump_cpu_state	在dump_runq中调用

Xen并没有提供一个调度器注册函数,要添加新的调度器只能修改schedulers数组,目前这个数组中有如下成员:

```
static struct scheduler *schedulers[] = {
    &sched_sedf_def,
    &sched_credit_def,
    NULL
}
```

task_slice,调度器用这个结构和调度核心接口,包括了下一个运行的 VM 以及运行的时间片大小

```
struct task_slice {
    struct vcpu *task;
    s_time_t     time;
}
```

当调度核心需要调度一个虚拟机时就会咨询当前的调度器,调度器通过返回这样的一个结构来确定下一个调度运行的虚拟机

第二节 调度核心

Xen 的调度器核心为 schedule 函数,这个函数实现 VM 切换操作,主要逻辑如下:

```
static void schedule(void)
{
    struct vcpu *prev = current;
    ...
    next_slice = ops.do_schedule(now);//咨询调度器,得到下一个需要调度的 VM
    ...
    r_time = next_slice.time;//运行时间片
    next = next_slice.task;//下一个 VM
    ...
    set_timer(&sd->s_timer, now + r_time);//定时
    ...
    context_switch(prev, next);
        ...
        set_current(next);
        ...
        __context_switch();
            ...
            schedule_tail(next);//这一步完成切换
}
```

```
static void __context_switch(void)
{
      struct cpu_user_regs *stack_regs = guest_cpu_user_regs();
      ...
      if ( !is_idle_vcpu(p) )
      {
            memcpy(&p->arch.guest_context.user_regs,
                  stack_regs,
                  CTXT_SWITCH_STACK_BYTES);//保存现场
            ...
            p->arch.ctxt_switch_from(p);
      }
      if ( !is_idle_vcpu(n) )
      {
            memcpy(stack_regs,
                  &n->arch.guest_context.user_regs,
                  CTXT_SWITCH_STACK_BYTES);//恢复现场
            n->arch.ctxt_switch_to(n);
      }
      ...
}
```

schedule_tail 函数完成最后的切换,不同的体系下,这个函数有不同的实现,但是都要调用 reset_stack_and_jump

- ✔ reset_stack_and_jump

  ```
  #define reset_stack_and_jump(__fn)              \
    __asm__ __volatile__ (                        \
    "mov %0,%%"__OP"sp; jmp "STR(__fn)      \
    : : "r" (guest_cpu_user_regs()) : "memory" )
  ```

  这段代码进行栈帧的切换,最后根据__fn 的不同将实现不同的功能:有的__fn 会调整栈帧,最后利用 ret 指令实现向栈帧中地址的跳转,例如在 paravirt 下;有的__fn 则读取栈帧中数据写入到另外的控制结构中,然后用别的方式实现切换,例如在 hvm 下;有的__fn 则是一个死循环,例如 reset_stack_and_jump(idle_loop)中的 idle_loop

- ✔ paravirt,这个体系下有 2 个切换函数:

  ```
  static void continue_idle_domain(struct vcpu *v)
  {
  ```

```
        reset_stack_and_jump(idle_loop);
}
static void continue_nonidle_domain(struct vcpu *v)
{
        reset_stack_and_jump(ret_from_intr);
}
```

- ✔ hvm 下分别是 vmx_do_resume 和 svm_do_resume,类似的这两个函数也会在最后调用 reset_stack_and_jump. 例如,vmx 是调用 reset_stack_and_jump(vmx_asm_do_vmentry),vmx_asm_do_vmentry 会把栈帧中的 eip,esp,eflags 写入到 vmcs 区域,最后用 vmlaunch 或 vmresume 指令进入 VM. 所以 hvm 下对栈帧的利用和在 paravirt 下还是不一样的,后者的切换方式和 linux 的任务切换是类似的.

第三节 时钟中断

Xen 利用时钟中断来实现虚拟机按时间片运行的功能. 当有时钟中断发生时,会调用下面函数:

```
fastcall void smp_apic_timer_interrupt(struct cpu_user_regs * regs)
```

这个函数设置定时器软中断标志后就返回了,紧接着会执行下面的代码序列:

```
ENTRY(ret_from_intr)
        GET_CURRENT(%ebx)
        movl  UREGS_eflags(%esp),%eax
        movb  UREGS_cs(%esp),%al
        //根据这个来判断时钟中断进入的是客户系统(ring3)还是 Xen(ring0)
        testl $(3|X86_EFLAGS_VM),%eax
        //如果是客户系统,则执行所有的软中断,例如,如果有调度定时器到期,
        //系统就会调用 schedule 函数,切换掉当前的虚拟机,需要注意的是当进入
        //schedule 函数时,堆栈指针刚好指向一个 cpu_user_regs 结构,这个结构会
        //在__context_switch 中保存到 vcpu 的状态中
        jnz   test_all_events
        //否则返回 Xen
        jmp   restore_all_xen
```

第四章 内存管理

Xen在初始化的过程中在不同的阶段会使用不同的内存管理方法(内存分配器),这些方法依次是:e820内存分配器,boot内存分配器,堆分配器.堆分配器是Xen正常运行下的主内存分配器,所有的客户系统的内存都是从这个分配器上分配.

第一节 初始内存分配

要点如下:

- 内存信息来源

mmap_type	说明
Xen-e820	内存信息由mem.S使用e820获取
Xen-e801	内存信息由mem.S使用e801获取
Multiboot-e820	内存信息由bootloader使用e820获取
Multiboot-e801	内存信息由bootloader使用e801获取

- 不管使用哪种方法,xen都使用e820_raw数组来记录系统最初的内存信息,这个数组的大小为e820_raw_nr
- 整理为e820,这是一个e820map类型的变量,xen通过下面调用将e820_raw转化为e820

```
max_page = init_e820(memmap_type, e820_raw, &e820_raw_nr)
```

max_page为内存上限的页框数.之后,e820就成为xen内存管理的主要数据结构,所有的内存分配操作都是围绕这个结构来进行的

- e820内存分配

```
int __init reserve_e820_ram(struct e820map *e820, uint64_t s, uint64_t e)
```

这个函数从e820上分配地址从s到e的内存

- 页表的更新1:e820中的内存必须映射到页表中才能使用:

```
for ( i = boot_e820.nr_map-1; i >= 0; i-- )
{
        ...
        map_pages_to_xen(
                (unsigned long)maddr_to_bootstrap_virt(s),
                s >> PAGE_SHIFT, (e-s) >> PAGE_SHIFT, PAGE_HYPERVISOR);
        ...
}
```

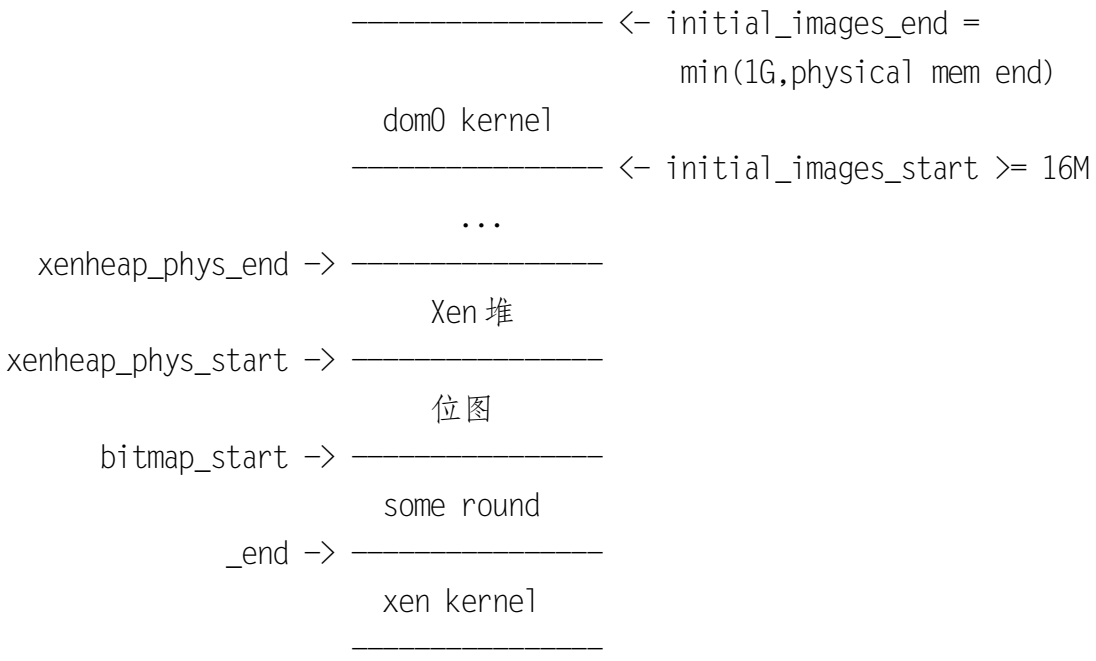
Xen的页表使用的是2M页框和4K页框的混合模式,当需要映射的内存满足一定的条件

时,将按照 2M 页框来管理,否则按 4K 页框管理.上面代码仅映射 16M-BOOTSTRAP_DIRECTMAP_END(x86_32 下是 1G)间并且是 Superpage 对齐的内存,另外 x86_32 下,线性地址和物理地址是一致的,参见下面宏定义:

```
#if defined(CONFIG_X86_64)
#define BOOTSTRAP_DIRECTMAP_END (1UL << 32) /* 4GB */
#define maddr_to_bootstrap_virt(m) maddr_to_virt(m)
#else
#define BOOTSTRAP_DIRECTMAP_END (1UL << 30) /* 1GB */
#define maddr_to_bootstrap_virt(m) ((void *)(long)(m))
#endif
```

注:这一步映射是为了满足移动 module 的需要,参见下图.

第二节 boot 分配器



内存管理初始化示意图

boot 内存分配器是一个暂时性的内存管理器.boot 分配器的建立分为两步:

- ✔ init_boot_allocator,在_end 后面(可能会有一些对齐)建立页框位图,位图大小取决于物理内存的多少
- ✔ 历遍 e820 数组,将 xenheap_phys_end 开始的内存登记到 boot 分配器中;对于大于 16M-1G 范围的内存还要映射到 Xen 的页表中(注 1),这时的映射为 1:1 的映射,也就是说物理地址和线性地址一致.

注 1:我们知道 0-16M 的空间在更早的初始化过程中已经映射到 xen 的页表中了,到这一步时,xen 的线性空间 0-1G 和__PAGE_OFFSET- +12M 范围已经映射

boot 内存的使用,在 Xen 调用 end_boot_allocator 之前,可以用 alloc_boot_pages 函数来分配 boot 内存.例如,init_frametable 就分配了 boot 内存.

```
unsigned long __init alloc_boot_pages(
                unsigned long nr_pfns, unsigned long pfn_align)
```

用这个函数分配 nr_pfns 个连续页框,返回的是起始页框的页框号,pfn_align 为起始页框的对齐参数,必须是 2 的幂,例如,可以要求起始页框在 1,2,4 等处对齐

第三节 堆分配器

堆分配器是 xen 的主内存分配器,这是一个和 linux 的内存分配器类似的分配器,使用(结点,区,order)的三元组来刻画内存,一个空闲的页框属于哪个区是由下面宏来计算的:

```
#define pfn_dom_zone_type(_pfn) (fls(_pfn) - 1)
```

例如,页框 1 属于 0 区,2,3 属于 1 区,4,5...7 属于 2 区,8,9...15 属于 3 区等等.即,一个区的内存范围为[2^n,2^(n+1)-1],n 为区号.在每个区中则按最大可用块的原则进行管理.

堆内存的建立分为两步:

- ✔ Xen 堆,即 xenheap_phys_start 和 xenheap_phys_end 之间的内存.这是一个特殊的堆,这个堆上的内存映射到 Xen 线性空间 DIRECTMAP_VIRT_START/DIRECTMAP_VIRT_END.参见 alloc_xenheap_pages 函数说明.Xen 堆使用 init_xenheap_pages 函数来初始化
- ✔ Dom 堆,管理除 Xen 堆之外的所有内存.用于 Domain 内存分配.Dom 堆在 boot 分配器结束后初始化

堆分配器的分配,堆分配器进行内存分配的核心函数为 alloc_heap_pages,这个函数原型如下:

```
static struct page_info *alloc_heap_pages(
    unsigned int zone_lo,
    unsigned int zone_hi,
    unsigned int cpu,
    unsigned int order)
```

zone_lo 和 zone_hi 为堆内存区范围,cpu 用来计算需要从中分配的结点,order 指明分配 2^order 个连续页框.

Xen 堆内存使用 alloc_xenheap_pages 来分配:

```
void *alloc_xenheap_pages(unsigned int order)
```

这个函数分配 2^order 个连续页框,返回的是线性空间 DIRECTMAP_VIRT_START/DIRECTMAP_VIRT_END 中的地址:我们知道 0-12M 间的内存是映射到__PAGE_OFFSET 开始的线性空间的,而__PAGE_OFFSET 和 DIRECTMAP_VIRT_START 是相等的,所以分配的线性地址是可以存

取的.

Dom堆使用alloc_domheap_pages函数来分配,这个函数除了从堆中分配页框外,还会将分配的内存记录到域中:

```
struct page_info *alloc_domheap_pages(
    struct domain *d,
    unsigned int order,
    unsigned int flags)
```

第四节 页框管理

1)页框管理结构

对于每个页框,Xen都分配一个page_info结构,称之为页框的管理结构.

```
struct page_info
{
    struct list_head list;
    u32 count_info;
    union {
        struct {
            u32 _domain;
            unsigned long type_info;
        } __attribute__ ((packed)) inuse;
        struct {
            u32 order;
            cpumask_t cpumask;
        } __attribute__ ((packed)) free;
    } u;
    union {
        u32 tlbflush_timestamp;
        unsigned long shadow_flags;
    };
};
```

页框管理结构记录了页框的使用情况,重要的成员如下:

- ✔ count_info这是一个32位的整数,用作页框的引用计数,各个位定义如下

位	说明
0-25	引用计数
26-28	3-bit PAT/PCD/PWT cache-attribute hint
29	PGC_page_table,当页框用作页表时
30	PGC_out_of_sync,当标记为和 Shadow 页框不同步时
31	PGC_allocated,当页框分配给客户系统后

- ✔ type_info

位	说明
0-25	类型引用计数
26	PGT_pae_xen_l2,仅 PAE,页框是一个包含 Xen 私有映射的二级页目录
27	PGT_validated,页框的当前类型已验证
28	PGT_pinned,客户系统锁定了页框的当前类型
29-31	页框类型,参见下表

- ✔ 页框类型(这些用途是互斥的)

值	类型	说明
000b	PGT_none	无特殊用途
001b	PGT_l1_page_table	1 级页表
010b	PGT_l2_page_table	2 级页表
011b	PGT_l3_page_table	3 级页表
100b	PGT_l4_page_table	4 级页表
101b	PGT_gdt_page	GDT
110b	PGT_ldt_page	LDT
111b	PGT_writable_page	可写页

- ✔ shadow_flags,当这个页框被 Shadow 时,这个成员会被设置,

位	名称	说明
0	SH_type_none	
1	SH_type_min_shadow SH_type_l1_32_shadow	

位	名称	说明
2	SH_type_fl1_32_shadow	
3	SH_type_l2_32_shadow	
4	SH_type_l1_pae_shadow	
5	SH_type_fl1_pae_shadow	
6	SH_type_l2_pae_shadow	
7	SH_type_l2h_pae_shadow	
8	SH_type_l1_64_shadow	
9	SH_type_fl1_64_shadow	
10	SH_type_l2_64_shadow	
11	SH_type_l2h_64_shadow	
12	SH_type_l3_64_shadow	
13	SH_type_l4_64_shadow SH_type_max_shadow	
14	SH_type_p2m_table	
15	SH_type_monitor_table	
16	SH_type_unused	

当需要设定一个页框的用途时,调用下面函数:

```
int get_page_type(struct page_info *page, unsigned long type)
```

page 为需要设定用途的页框,type 必须是上面”页框类型”表”类型”列中的值之一或者这些值之一与 PGT_pae_xen_l2 的或值.这个函数会:

- ✔ 增加 type_info 的引用计数
- ✔ 如果 type 与 page 现有的用途不一致,则 page 会被标记为未验证(清除 PGT_validated 位),但是 PGT_writable_page 例外,因为这个用途无需额外验证
- ✔ 如果 page 被标记为未验证,调用 alloc_page_type 验证之,如果成功,标记 page 为已验证

当需要引用一个页框时,调用下面函数:

```
int get_page(struct page_info *page, struct domain *domain)
```

这个函数增加 page 的 count_info 引用计数

```
static inline int get_page_and_type(struct page_info *page,
                                    struct domain *domain,
                                    unsigned long type)
```

这个函数则先调用 get_page 然后调用 get_page_type

2)页框号的管理

Xen 将系统内存按照每 4K 一个单位进行编号,称之为页框号,整个页框空间为 0-max_page,这个空间中的页框号也叫机器页框号,或 mfn,这个页框号是处理器可以识别的;当内存分配到客户系统中后,由于客户系统一般都要求连续的内存空间,因此,这些页框被重新编号,一般从 0 开始,这个编号叫客户物理页框号,或 gpfn,客户系统工作在这个编号上.两者常常需要相互转换,因此 Xen 维护两个表:

一个是 mfn 到 gpfn 的映射表,这个表可以通过下列途径进行更新:

- ✔ 直接更新(使用 set_gpfn_from_mfn 函数),dom0 的创建过程中就是使用这个方法
- ✔ 通过 hypercall:XENMEM_populate_physmap,这个调用向 Xen 申请一定数量的内存,调用者需要传入一个 gpfn 的列表,Xen 会为每个 gpfn 分配内存,并在映射表中建立关联.这个调用会返回一个客户系统分配到的 mfn 的列表,客户系统使用这个列表建立 p2m 映射
- ✔ 通过 hypercall:MMU_MACHPHYS_UPDATE,调用者传入一个(mfn,gpfn)对,Xen 将会更新 mfn 的对应到新的 gpfn

另一个是 gpfn 到 mfn 的映射表,这个表通过下列途径进行更新:

- ✔ 直接更新,例如 dom0 就是直接写入 vphysmap_start 数组
- ✔ 没启用 PG_translate 时,利用 hypercall:XENMEM_populate_physmap 的返回值(参见上面说明).例如,libxc 在为 domU 申请内存后,就会将这个返回值暂存到 p2m_host 中,并最终通过 mfn_list 传递给 domU.对于 domU 来说,其启动结构(start_info)成员 mfn_list 指向这个映射表, domU 在调用某些要求 mfn 的 hypercall 时可以利用这个表来查找对应的 mfn.

  在启用 PG_translate 的情况下,hypercall:XENMEM_populate_physmap 不会返回 mfn 列表,客户系统不需要自己维护这样的一个列表,Xen 会自动维护.

一些有用的宏/函数:

```
gfn_to_mfn(d, g, t)
```

返回 g 对应的 mfn

```
static inline unsigned long gmfn_to_mfn(struct domain *d, unsigned long gpfn)
```

同 gfn_to_mfn

```
static inline unsigned long mfn_to_gfn(struct domain *d, mfn_t mfn)
```

返回 mfn 对应的 gpfn

第五章 页表管理

x86体系下,代码是通过线性地址空间来存取内存的,而处理器在执行内存操作时使用的是物理地址,因此需要一个叫页表的数据结构实现两者的转换.x86提供了多种页表映射模式,Xen只使用了其中的几种模式,参见第一节.页表可以是”稀疏”的,也就是说线性空间的某个部份在页表中可能并没有建立映射,当代码存取这部份空间时就会发生”缺页”,这时处理器会引发一个缺页中断,这个中断负责建立缺页地址的页表项.

Xen在初始化完毕后会建立自己的缺页中断处理函数,发生缺页的情况后,Xen会首先获得控制权,进行必要的处理,如果需要再将控制转移到客户系统的缺页管理函数.

需要澄清的是Xen中有两种类型的页表:

- ✔ Xen自身的页表,Xen运行在线性空间中,因此也需要页表.以x86_32为例,初始化完毕后Xen页表的映射关系如下(从高到低):

线性地址空间	说明
IOREMAP_VIRT_START至IOREMAP_VIRT_END	ioremap()/map_domain_page_global()使用
DIRECTMAP_VIRT_START至DIRECTMAP_VIRT_END	Xen堆分配空间.映射0-12M内存范围
RDWR_MPT_VIRT_START至RDWR_MPT_VIRT_END	mfn->gpfn映射表.
FRAMETABLE_VIRT_START至FRAMETABLE_VIRT_END	page_info数组
16M-1G	Dom堆分配空间.映射16M-1G内存范围(注)
12M-16M	映射12M-16M内存范围
0M-12M	映射0M-12M内存范围,Xen映像

注:取决于1G和实际内存中的小者

- ✔ 域页表

线性地址空间	说明
PERDOMAIN_VIRT_START至PERDOMAIN_VIRT_END	map_domain_page()
LINEAR_PT_VIRT_START至LINEAR_PT_VIRT_END	domain页表空间

线性地址空间	说明
RO_MPT_VIRT_START 至 RO_MPT_VIRT_END	domain 的 p2m 表
v_start 至 v_end	domain 映像

第一节 页表模式

Xen 在 x86_32 配置下支持两种页表配置,每种配置下,Xen kernel 和客户系统分别使用不同页框大小页表模式:

一,PAE 模式

- ✔ 客户系统页表模式,线性地址被分解为 4 部分,对应 3 级页表,参见下表:

线性空间划分				页框大小
31-30	29-21	20-12	11-0	4K
l3 页表	l2 页表	l1 页表	---	

- ✔ Xen 页表模式,线性地址被分解为 3 部分,对应 2 级页表,参见下表:

线性空间划分			页框大小
31-30	29-21	20-0	2M
idle_pg_table	idle_pg_table_l2	---	

实际上 Xen kernel 使用的是混合页表模式,既有 2M 的页框,也有 4K 的页框

二,非 PAE 模式

- ✔ 客户系统页表模式,线性地址被分解为 3 部分,对应 2 级页表,参见下表:

线性空间划分			页框大小
31-22	21-12	11-0	4K
l2 页表	l1 页表	---	

- ✔ Xen kernel 页表模式,线性地址被分解为 2 部分对应 1 级页表,参见下表:

线性空间划分		页框大小
31-22	21-0	4M
idle_pg_table	---	

第二节 dom0 页表的构建

从前面可知,dom0 映像会从 Dom 堆上分配内存,这部份内存就包括了 dom0 页表所需空间,具体的说从 vpt_start 到 vpt_end 之间的空间用来构建 dom0 的页表.根据是否启用了 PAE 模式,这段空间的分配如下:

偏移(页框)	用途	page_info	说明
PAE 模式			
0	第 3 级页表	PGT_l3_page_table PGT_pinned	实际只使用 4 个表项
1-3	第 2 级页表	PGT_l2_page_table	初始化为 idle_pg_table_l2
4	第 2 级页表	PGT_l2_page_table| PGT_pae_xen_l2	
5-	第 1 级页表	PGT_l1_page_table	v_start 至 v_end 空间所需第 1 级页表从这里分配
非 PAE 模式			
0	第 2 级页表	PGT_l2_page_table PGT_pinned	初始化为 idle_pg_table
1-	第 1 级页表	PGT_l1_page_table	v_start 至 v_end 空间所需第 1 级页表从这里分配

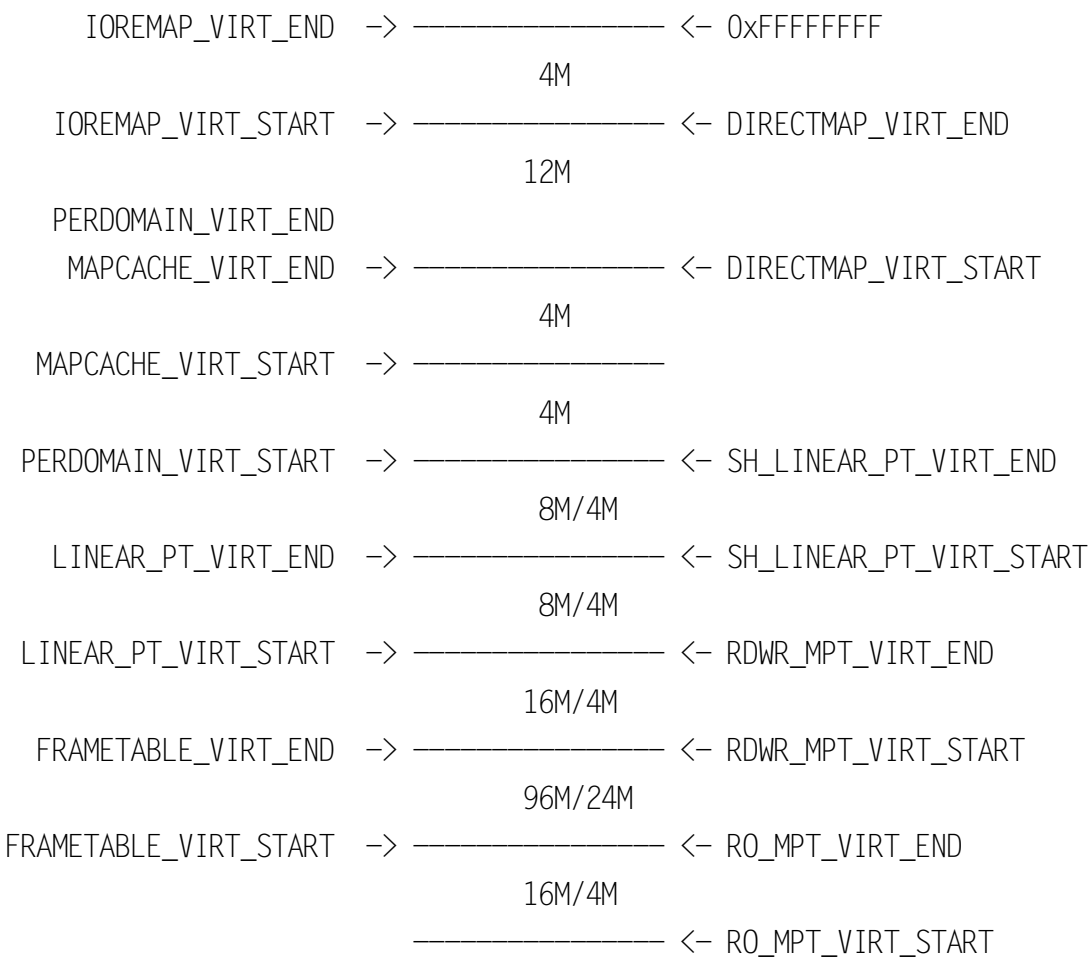
第三节 domU 页表的构建

domU 用 libxc 中的提供的函数来创建,当然 libxc 最终调用的是 hypercall 来完成相关功能.主要流程如下:

- ✔ 根据 domU 映像大小计算所需页表
- ✔ 分配页表所需内存
- ✔ 映射页表内存到 dom0 线性空间中以便读写
- ✔ 设置页表:将 domU 映像映射入线性空间,包括页表自身
- ✔ 设置 vcpu context 中的 cr3

第四节 Xen 线性空间

这部份线性空间驻留在 Xen 和 domain 线性空间的高端,并不是所有的空间范围都有效,例如,FRAMETABLE_VIRT_START/ FRAMETABLE_VIRT_END 就仅对 Xen 有效,



- ✔ FRAMETABLE_VIRT_START/FRAMETABLE_VIRT_END,页框管理结构 struct page_info 数组.变量 frame_table 指向这个数组的开头.init_frametable 分配并安装这个区间所需页框.和这个数组相关的一些宏如下:

宏	说明
mfn_to_page(mfn)	机器页框<->page_info 结构
page_to_mfn(pg)	
virt_to_page(va)	Xen 堆线性地址<->page_info 结构
page_to_virt(pg)	
maddr_to_page(ma)	机器地址<->page_info 结构
page_to_maddr(pg)	

- ✔ RDWR_MPT_VIRT_START/RDWR_MPT_VIRT_END,这段空间是 mfn 到 gpfn 的映射表,所需页

框(注意,这是 2M 大小的页框)在 paging_init 中分配并安装,与之相关的宏如下:

宏	说明
machine_to_phys_mapping	映射表的线性地址
set_gpfn_from_mfn(mfn, pfn)	设置 mfn->pfn 映射
get_gpfn_from_mfn(mfn)	获取 mfn 对应的 pfn

- RO_MPT_VIRT_START 和 RO_MPT_VIRT_END,对于 paging_mode_translate 客户系统,这部份空间是 gpfn 到 mfn 的映射表(p2m 表),所需内存由 domU 的创建者分配,例如,在 libxc 中 alloc_magic_pages(xc_dom_x86.c)函数就负责计算并分配 p2m 表所需页框.分配的页框通过 start_info.mfn_list 传递给客户系统;对于 Xen,这部份空间和 RDWR_MPT_VIRT_START-RDWR_MPT_VIRT_END 空间对应的都是一样的页框,即,是 mfn 到 gpfn 的映射,只不过这部份空间为只读映射,详细情况参见 paging_init 函数
- LINEAR_PT_VIRT_START/LINEAR_PT_VIRT_END,这部份空间用于存取 domain 的页表
- PERDOMAIN_VIRT_START/PERDOMAIN_VIRT_END,这部份空间所需页表由 domain.arch.mm_perdomain_pt 提供,这部份页表在 arch_domain_create 中计算并分配,对于 dom0 在 construct_dom0 中安装,对于 domU 是在 create_pae_xen_mappings 中安装.初始化情况如下:

偏移(l1_pgentry_t 个数)	名称	说明
14	FIRST_RESERVED_GDT_PAGE	从这里开始,vcpuid <<GDT_LDT_VCPU_SHIFT 处为一个指向 gdt_table 的 l1 表项,共 MAX_VIRT_CPUS 个
fixme	domain.arch.mapcache.l1tab	MAPCACHE_VIRT_START/MAPCACHE_VIRT_END 空间所需页表,共 1024 个表项,刚好映射,这个空间段:4M

- MAPCACHE_VIRT_START/MAPCACHE_VIRT_END,这部份空间用于临时映射一些页框,以便 Xen 对页框的读写,参见下面函数:

函数	说明
void *map_domain_page(unsigned long mfn)	映射 mfn 到上述空间中
void unmap_domain_page(void *va)	取消 va 处的映射

注:这部份空间所需页表来自 d->arch.mapcache.l1tab,参见 mapcache_domain_init

函数

- ✔ DIRECTMAP_VIRT_START/DIRECTMAP_VIRT_END,Xen 堆线性空间,即,所有从 Xen 堆分配的内存,其线性地址都落入这个范围,相关的宏如下:

宏	说明
virt_to_maddr(va)/__pa(x)	Xen 堆线性地址<->机器地址
maddr_to_virt(ma)/__va(x)	

- ✔ IOREMAP_VIRT_START/IOREMAP_VIRT_END,这部份空间也是用于临时映射一些页框,以便 Xen 对页框的读写,所需页表在 paging_init 中分配并安装.使用下面函数来映射页框到这个空间:

函数	说明
void *map_domain_page_global(unsigned long mfn)	映射 mfn 到上述空间中
void unmap_domain_page_global(void *va)	取消 va 处的映射

第五节 缺页中断

我们知道,Xen 对 MMU 也进行了虚拟化,当发生缺页中断时,Xen 的中断逻辑将首先被执行:

```
asmlinkage void do_page_fault(struct cpu_user_regs *regs)
{
    ...
    if ( unlikely(fixup_page_fault(addr, regs) != 0) )
        return;
    if ( unlikely(!guest_mode(regs)) )
    {
        if ( spurious_page_fault(addr, regs) )
            return;
        if ( likely((fixup = search_exception_table(regs->eip)) != 0) )
        {
            ...
            regs->eip = fixup;
            return;
        }
        ...
```

```
    }
    propagate_page_fault(addr, regs->error_code);//调用客户系统的缺页逻辑
}
static int fixup_page_fault(unsigned long addr, struct cpu_user_regs *regs)
{
    if ( unlikely(IN_HYPERVISOR_RANGE(addr)) )//addr>= HYPERVISOR_VIRT_START
    {
        if ( paging_mode_external(d) && guest_mode(regs) )
        {
            int ret = paging_fault(addr, regs);
            if ( ret == EXCRET_fault_fixed )
                trace_trap_two_addr(TRC_PV_PAGING_FIXUP, regs->eip, addr);
            return ret;
        }
        if ( (addr >= GDT_LDT_VIRT_START) && (addr < GDT_LDT_VIRT_END) )
            return handle_gdt_ldt_mapping_fault(
                addr - GDT_LDT_VIRT_START, regs);
        return 0;
    }
    if ( VM_ASSIST(d, VMASST_TYPE_writable_pagetables) &&
         guest_kernel_mode(v, regs) &&
         ((regs->error_code & PFEC_write_access) == PFEC_write_access) &&
         ptwr_do_page_fault(v, addr, regs) )
        return EXCRET_fault_fixed;
    if ( paging_mode_enabled(d) )//启用了页表助手
    {
        int ret = paging_fault(addr, regs);//调用页表助手缺页逻辑
        if ( ret == EXCRET_fault_fixed )
            trace_trap_two_addr(TRC_PV_PAGING_FIXUP, regs->eip, addr);
        return ret;
    }
}
```

从上面可知,在没有启用页表助手的情况下,这个缺页逻辑基本上等同于客户系统的缺页逻辑.

第六节 页表助手

页表助手是指domain.arch.paging.mode成员的值,这些值直接影响着Xen页表管理逻辑,参见下表:

位	名称	说明	相关宏
10-12	修饰符	001b:PG_refcounts	paging_mode_refcounts
		010b:PG_log_dirty	paging_mode_log_dirty
		011b:PG_translate	paging_mode_translate
		100b:PG_external	paging_mode_external
20	PG_SH_enable	启用Shadow助手	paging_mode_shadow shadow_mode_enabled shadow_mode_refsounts shadow_mode_log_dirty shadow_mode_translate shadow_mode_external
21	PG_HAP_enable	启用HAP助手	paging_mode_hap

另外paging_mode_enabled定义为:

```
#define paging_mode_enabled(_d)    ((_d)->arch.paging.mode)
```

含义是是否启用了页表助手

页表助手接口,用于实现特定的页表助手,Xen的缺页逻辑在适当的时候调用这个接口,参见第五节.目前,Xen实现了两个页表助手:Shadow和HAP,因此也存在两个这样的接口,这个接口用结构paging_mode来代表,其成员如下:

成员	说明
page_fault	缺页处理函数
invlpg	
gva_to_gfn	
update_cr3	如果domain是paging_mode_enabled,这个成员将会在update_cr3函数中被调用
update_paging_modes	
write_p2m_entry	

成员	说明
write_guest_entry	
cmpxchg_guest_entry	
guest_map_l1e	
guest_get_eff_l1e	

第七节 Shadow 页表

Shadow 页表是和客户页表一样的一套页表,在启用 Shadow 助手时,Xen 会用这套页表替换客户页表.要点如下:

- ✔ 更新 cr3 时 Shadow 将拷贝客户的顶级页表(下面代码是以 x86_32pae 为例):

```
//得到客户页表页框
gmfn = pagetable_get_mfn(v->arch.guest_table);
//映射到线性空间中,这里 guest_idx =0
gl3e = ((guest_l3e_t *)sh_map_domain_page(gmfn)) + guest_idx;
//拷贝
for ( i = 0; i < 4 ; i++ )
   v->arch.paging.shadow.gl3e[i] = gl3e[i];
for ( i = 0; i < 4; i++ )
{
   //创建 Shadow,Shadow 页框将记录在 arch_vcpu.shadow_table 数组中
   sh_set_toplevel_shadow(...);
}
//创建 l3table
for ( i = 0; i < 4; i++ )
{
    smfn = pagetable_get_mfn(v->arch.shadow_table[i]);
    v->arch.paging.shadow.l3table[i] =
              (mfn_x(smfn) == 0)
              ? shadow_l3e_empty()
              : shadow_l3e_from_mfn(smfn, _PAGE_PRESENT);
}
//最后更新 cr3 指向 Shadow 页表
v->arch.cr3 = virt_to_maddr(&v->arch.paging.shadow.l3table);
```

- 其它层级的Shadow页表则是在缺页中断处理函数中建立的

```
static int sh_page_fault(struct vcpu *v,
                         unsigned long va,
                         struct cpu_user_regs *regs)
{
    ...
    ptr_sl1e = shadow_get_and_create_l1e(v, &gw, &sl1mfn, ft);
    ...
    //生成sl1e表项
    l1e_propagate_from_guest(v, gw.l1e, gmfn, &sl1e, ft, p2mt);
    //写入之
    r = shadow_set_l1e(v, ptr_sl1e, sl1e, sl1mfn);
}
```

当启用Shadow时,Xen会从domain堆上分配一些Shadow功能所需的页表,这些页表记录在domain.arch.paging.shadow.freelists中,这是一个不同大小内存块的数组,相同大小的内存块链接在一起,最大的内存块为2^(SHADOW_MAX_ORDER + 1).当需要从这个内存池分配内存时,使用下面函数:

```
mfn_t shadow_alloc(struct domain *d,
                   u32 shadow_type,
                   unsigned long backpointer)
```

这个函数返回与shadow_type对应大小的内存,当某个客户页表需要Shadow时,就可以调用这个函数来分配一个Shadow页.Shadow助手维护一个Shadow页页框号和客户页页框号对应关系的高速缓存(shadow hash),每当Shadow一个客户页表时就在hash表中创建一个条目.

第六章 事件管道

事件管道(event channel)是向域发送消息的一种通信机制.管道的一端连接一个域,另一端连接一个事件源,这些事件源可以是其它的域,物理中断,虚拟中断和IPI消息,当连接的是两个域时,管道可以用于域间的双向通信.每个这样的管道都有一个编号,称之为端口.一个域可以打开的端口数为 NR_EVENT_CHANNELS(1024)个.下面是事件管道用途的列表:

用途	说明
ECS_FREE	管道可供分配
ECS_RESERVED	管道被保留
ECS_UNBOUND	等待远端接入

用途	说明
ECS_INTERDOMAIN	远端已接入或已接入到远端
ECS_PIRQ	物理 IRQ 管道
ECS_VIRQ	虚拟 IRQ 管道
ECS_IPI	IPI 消息管道

第一节 事件的处理

是指 Xen 是如何实现管道机制的,要点如下:

- ✔ 当某个管道有事件发生时,domain.shared_info->evtchn_pending 的对应位会被置位.随后 Xen 会调用 vcpu_mark_events_pending,这个函数会设置 vcpu.vcpu_info->evtchn_upcall_pending 成员
- ✔ 在每次从中断中返回时,evtchn_upcall_pending 会被检查,如果这个成员值不为 0,Xen 会进入事件处理流程(详情参见 entry.S 中的 test_all_events)
- ✔ 一般来说客户系统会调用 do_set_callbacks 或者 do_callback_op hypercall 向 Xen 注册自己的事件回调函数(前者是后者的简化版).例如,mini-os 调用前者设置事件处理函数为 hypervisor_callback.客户系统在这个回调函数中实现自己的事件处理逻辑.

第二节 事件管道 hypercall

客户系统用这些调用来使用 Xen 的事件管道机制,列表如下:

调用	说明
EVTCHNOP_alloc_unbound	从域中分配一个等待某个远端域连接的事件管道.注,只有特权域,例如 dom0 才能从任意域中分配端口,普通域只能本地分配端口.调用返回新分配的本地端口.这种类型的管道是留做 inerdomain 类型管道的接入的
EVTCHNOP_bind_interdomain	建立一个当前域(**注 1**)到远端域的管道.这个调用需要指定远端域和远端端口号,远端域的端口必须是预先用 alloc_unbound 为本域分配的.调用会返回管道的本地端口号.注,这是一个双向管道,即,管道的两端都可以向对方发送信号
EVTCHNOP_bind_virq	建立虚拟中断(virq)事件管道.这个调用对当前域进行操

调用	说明
	作,并且要指定域内的通知 vcpu(**注 2**),如果 virq 为全局 virq(**注 3**),通知 vcpu 只能是 0.当需要引发 virq 时,调用 send_guest_vcpu_virq 或 send_guest_global_virq 函数
EVTCHNOP_bind_ipi	建立 IPI 消息的事件管道.这个调用对当前域进行操作,并且要指定域内 IPI 信号的通知 vcpu.用 EVTCHNOP_send 调用发送 IPI 信号
EVTCHNOP_bind_pirq	建立物理中断(pirq)事件管道.这个调用对当前域进行操作.这个调用会把当前域加入到 pirq 处理函数的客户系统列表中,这样在发生 pirq 时,客户 系统就会收到中断.注:pirq 管道是需要一个通知 vcpu 参数的,但是在这个调用中没有设置,需要再调用一次 EVTCHNOP_bind_vcpu 设置
EVTCHNOP_close	关闭指定域中的指定的管道.关闭后,管道状态被设置为 ECS_FREE,如果是 interdoamin 管道,远端管道被设置为 ECS_UNBOUND
EVTCHNOP_send	用于向事件管道发送一个消息.这个调用对当前域进行操作.仅支持 IPI 和 INTERDOMAIN 管道,对于前者,相当于给通知 vcpu 发送一个 IPI 信号,对于后者则是向远端域的通知 vcpu 发送信号
EVTCHNOP_status	读取指定域的指定管道状态.注,只有特权域,例如 dom0 才能读取任意域的管道状态,否则只能对自己进行此操作
EVTCHNOP_bind_vcpu	设置管道的通知 vcpu,仅支持 virq(仅当 virq 为全局时),unbound,interdomain,pirq 管道
EVTCHNOP_unmask	启用通知 vcpu 上的指定管道.这个调用对当前域进行操作
EVTCHNOP_reset	对指定域上的所有管道调用 EVTCHNOP_close.注,只有特权域才能对其它域进行此操作,普通域只能对自己进行此操作

注 1:当前 vcpu 所属域

注 2:即 evtchn 的 notify_vcpu_id 成员,下面是 evtchn 的成员,为了便于对上述调用的理解,将 evtchn 结构列示在下面:

```
struct evtchn
{
```

```
    u8  state;              /* ECS_* */
    u8  consumer_is_xen;    /* Consumed by Xen or by guest? */
    u16 notify_vcpu_id;     /* VCPU for local delivery notification */
    union {
        struct {
            domid_t remote_domid;
        } unbound;      /* state == ECS_UNBOUND */
        struct {
            u16             remote_port;
            struct domain *remote_dom;
        } interdomain; /* state == ECS_INTERDOMAIN */
        u16 pirq;       /* state == ECS_PIRQ */
        u16 virq;       /* state == ECS_VIRQ */
    } u;
#ifdef FLASK_ENABLE
    void *ssid;
#endif
};
```

注 3:即 virq_is_global(virq),全部的 virq 列表如下:

virq	名称	全局	说明
0	VIRQ_TIMER	否	vcpu 定时器中断.可以通过 set_periodic_timer hypercall 来设置定时器间隔
1	VIRQ_DEBUG	否	要求客户系统生成调试信息
2	VIRQ_CONSOLE	是	由紧急 console 驱动产生,通知 dom0 接收到新字符
3	VIRQ_DOM_EXC	是	域异常中断(dom0).在域关闭时产生该中断
4	VIRQ_TBUF	是	跟踪缓冲区有记录产生(dom0)
6	VIRQ_DEBUGGER	是	当有客户系统因调试而暂停时(dom0)
7	VIRQ_XENOPROF	否	xenprofile 有新的数据时
8	VIRQ_CON_RING	是	由 console 驱动产生,通知 dom0 接收到

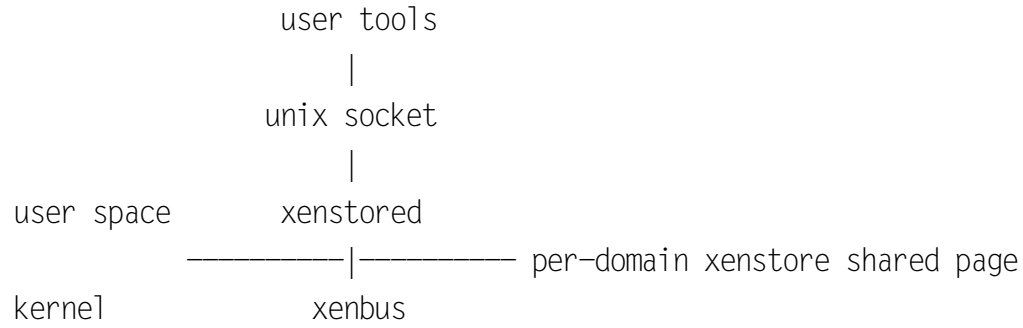
virq	名称	全局	说明
			新字符
16-23	VIRQ_ARCH_0-VIRQ_ARCH_7	取决于体系	

第三节 事件管道设备

这个设备是dom0-linux下用户空间和Xen事件系统间的接口.设备路径为/dev/xen/evtchn.要点如下:

- ✔ 每次打开这个设备时,Xen将会分配一个页面作为环形读写区和一个管理结构.读写的基本元素为管道端口号.
- ✔ 读操作,当没有事件发生时,读操作会休眠,如果有事件发生,读入的是一个管道端口号的数组,该数组中的所有端口都有事件发生.
- ✔ 写操作,这个操作开启指定的事件管道.
- ✔ 可以通过ioctl来进行上面的事件管道操作.

第七章 设备模型



如上图所示,Xen使用一个叫xenstore的中心数据库来存储设备信息,这是一个按树状结构组织的数据库,例如,每个域在/local/domain/目录下有一个对应目录,目录名就是域ID号,域目录下最重要的两个和设备配置有关的子目录是backend和device,分别存放后端和前端设备的配置参数.

xenstore由xenstored守护者进程来管理,这个进程通过unix socket响应用户空间的请求.例如,用户空间工具可以配置某个域中的前后端设备.xenstored可以响应的消息如下表所示:

消息	说明
XS_DIRECTORY	列出指定目录下的内容

消息	说明
XS_READ	读取某个结点值
XS_WRITE	写入某个结点值
XS_MKDIR	创建一个目录
XS_RM	移除一个目录
XS_GET_PERMS	读取某个目录的操作权限
XS_SET_PERMS	设置某个目录的操作权限
XS_DEBUG	
XS_WATCH	监听某个路径,当路径发生改变时将调用设定的回调函数
XS_UNWATCH	解除路径监听
XS_TRANSACTION_START	开始一个事务
XS_TRANSACTION_END	结束一个事务
XS_INTRODUCE	引入一个域,建立起 xenstored 守护者进程和 xenbus 之间的通信区
XS_IS_DOMAIN_INTRODUCED	域是否已引入
XS_RELEASE	释放 XS_INTRODUCE 时创建的 xenstored 和 xenbus 之间的连接
XS_GET_DOMAIN_PATH	返回指定域的路径,即,/local/domain/<domid>
XS_RESUME	

另外每个域在创建时都会分配一个 xenstore page,这个页框将作为 xenbus 和 xenstored 之间的通信区.当设备驱动需要读写配置信息时,就通过 xenbus 来存取.

第一节 设备模型

Xen 的设备模型将设备分为两部份,一部份叫后端,一部份叫前端.后端一般运行在 dom0 中,负责和真正的设备打交道,而前端运行在 domU 中,为 domU 中的客户系统提供设备.后端和前端之间通过事件管道进行通信.下面以块设备为例说明 Xen 设备模型的基本工作流程:

上图是块设备模型的一个总体结构,blkfront 和 blkback 分别是块设备的前端和后端,在 xenstore 中的目录为 device/vbd 和 backend/vbd.blkif 这个对象代表的是后端的真实设备,由它向物理设备派发请求.gendisk 则是呈现给 domU 的设备.所有的块设备请求通过 blkfront 传递给 blkback 再由后者派发到物理设备.整个系统初始化的要点如下:

1,blkfront 设备

1,blkfront_probe,这是 blkfront 设备驱动的探测函数

1,读取 xs:virtual-device 参数

2,分配并填写一个 blkfront_info 结构,并用这个结构调用 talk_to_backend

2,talk_to_backend,setup_blkring

1,分配 ring 缓冲区,这是前后端的主通信区,并将缓冲区授权给后端存取

2,分配一个 unbound 的事件管道,该管道通过一个 irq 连接 blkif_int 函数,该函数处理后端信号:更新请求的执行情况

3,将第一步中缓冲区的授权号写入 xs:ring-ref 参数;将第二步中分配的端口号写入 xs:evtchn-channel 参数;设置 xs:protocol 参数为 XEN_IO_PROTO_ABI_NATIVE

4,切换状态为 XenbusStateInitialised,这个状态会触发后端的 connect_ring 和 update_blkif_status

3,connect

1,读取后端设备的 sectors,info,sector-size 参数

2,添加 vbd 设备:xlvbd_add

3,切换状态为 XenbusStateConnected

4,添加一个 gendisk

2,blkback 设备

1,blkback_probe,这是 blkback 设备驱动的探测函数

1,分配一个 blkif 结构

2,监听 xs:physical-device 参数,这个参数是 MAJOR:MINOR 的格式,监听处理函数会根据这个物理设备号创建相关结构,核心是得到一个 block_device 结构

3,切换状态为 XenbusStateInitWait

2,connect_ring

1,读取前端设备的 ring-ref,event-channel,protocol 参数

2,映射 blkif

1,分配 blk_ring_area

2,映射 ring 缓冲区到 blk_ring_area

3,初始化 blk_rings,实际也是指向了 ring 缓冲区

4,interdomain 连接到前端的事件管道,并连接到 blkif_be_int 函数,该函数处理前端信号:唤醒 blkif_schedule 线程处理块设备请求

3,update_blkif_status

1,在 xs 中生成 sectors,info,sector-size 块设备参数

2,切换状态为 XenbusStateConnected,这个状态会触发前端的 connect

3,创建一个跑 blkif_schedule 的线程,线程的名称为 blkback.<domid>.<name>,其中 name 根据 xs 的 dev 参数生成.这个线程负责执行 ring 缓冲区中的块设备请求

第二节 授权表

xen <- shared[] -> domU

授权表(grant table),用于实现不同客户系统间共享页框的功能,如上图所示,Xen 和域之间通过一个共享数组 shared 来交换授权表信息.当一个域需要授权某个域访问其页框时,就在 shared 数组中填写相关的授权信息.当需要时,Xen 读取该信息将共享页框映射到被授权域.因为授权表用况大量出现在设备驱动中(例如,块设备驱动中,客户系统的 IO 页框会授权给后端设备读写),所以放在本章讲解.Xen 用域的 grant_table 成员来管理授权表系统,重要成员如下:

- shared,这是一个 struct grant_entry 类型的数组,Xen 会和客户系统共享 shared 数组,因此,客户系统可以通过读写相应的 grant_entry 条目的方式来授权某个页框(frame)到某个域(domid)

```
struct grant_entry {
    uint16_t flags;
    domid_t  domid;
    uint32_t frame;
}
```

下面这个宏用于存取授权表 e 对应的 shared 条目,t 是域的 grant_table 成员

```
shared_entry(t, e)
```

- active,这是一个 struct active_grant_entry 类型的数组,当域需要访问某个授权表时,调用 map_grant_ref hypercall 来映射授权表到线性空间中供存取.这时 Xen 会使用一个 active 条目来记录这个映射.frame 从 shared 数组条目复制过来

```
struct active_grant_entry {
    u32           pin;
    domid_t       domid;
```

```
    unsigned long frame;
}
```

- ✔ 下面这个宏用于存取授权表 e 对应的 active 条目,t 是域的 grant_table 成员
  active_entry(t, e)

第九章 hypercall

hypercall 是 Xen 提供给客户系统的交互介面,客户系统通过 hypercall 存取硬件资源. 根据 Xen 的安全模型,Xen 运行在 ring0 层,而 dom0 可以运行在 ring0 或 ring1 层,domU 运行在 ring3 层,运行在不同层的客户系统,其 hypercall 有不同的调用方法.

第一节 hypercall 初始化

为了能够使用 hypercall,客户系统必须分配一个 hypercall 页框,并将页框地址告诉 Xen(通过 HYPERCALL_PAGE 标签),Xen 在加载客户系统时就会在这个页框中创建 hypercall 调用,具体的说就是在页框中植入一些特定的代码,根据客户系统安全层次/体系的不同,将有不同的代码模版,参见下表:

体系	子体系/安全层次	指令模版

paravirt	ring0	pushf
		cli
		mov i,eax
		lcall __HYPERVISOR_CS,&hypercall
		ret
		对于 iret 为:
		push %eax
		pushf
		cli
		mov i,%eax
		lcall __HYPERVISOR_CS,&hypercall

	ring1,ring3	mov i,eax int 0x82 ret 对于 iret 为:

体系	子体系/安全层次	指令模版
		push %eax mov __HYPERCALL_iret,eax int 0x82
hvm	vmx	mov i,%eax vmcall ret
	svm	略

i 为 hypercall 的调用号.最终形成如下的调用页框(假设 ring3 的情况)

```
hypercall_page:
    mov 0,eax
    int 0x82
    ret
    ...
    mov 1,eax
    int 0x82
    ret
    ...
    ...
    mov n,eax
    int 0x82
    ret
```

上面每个代码模版会相差 32 个字节,因此,客户系统的 hypercall 调用模版为:

```
call hypercall_page + nr * 32
```

其中 nr 为 hypercall 调用号,hypercall 的参数则依次用 eax,ebx,ecx,edx,esi,edi 来传递

xen 支持的 hypercall 入口在 entry.S 中(hypercall_table),列表如下(按调用号排序):

1. do_set_trap_table /* 0 */
2. do_mmu_update
3. do_set_gdt
4. do_stack_switch
5. do_set_callbacks
6. do_fpu_taskswitch /* 5 */
7. do_sched_op_compat
8. do_platform_op

9. do_set_debugreg
10. do_get_debugreg
11. do_update_descriptor /* 10 */
12. do_ni_hypercall
13. do_memory_op
14. do_multicall
15. do_update_va_mapping
16. do_set_timer_op /* 15 */
17. do_event_channel_op_compat
18. do_xen_version
19. do_console_io
20. do_physdev_op_compat
21. do_grant_table_op /* 20 */
22. do_vm_assist
23. do_update_va_mapping_otherdomain
24. do_iret
25. do_vcpu_op
26. do_ni_hypercall /* 25 */
27. do_mmuext_op
28. do_xsm_op
29. do_nmi_op
30. do_sched_op
31. do_callback_op /* 30 */
32. do_xenoprof_op
33. do_event_channel_op
34. do_physdev_op
35. do_hvm_op
36. do_sysctl /* 35 */
37. do_domctl
38. do_kexec_op

本文不对这些 hypercall 作详细讲解.