KVM 虚拟机源代码分析

1, KVM 结构及工作原理

1.1KVM 结构

KVM 基本结构有两部分组成。一个是 KVM Driver ，已经成为 Linux 内核的一个模块。负责虚拟机的创建，虚拟内存的分配，虚拟 CPU 寄存器的读写以及虚拟 CPU 的运行等。另外一个是稍微修改过的 Qemu，用于模拟 PC 硬件的用户空间组件，提供 I/O 设备模型以及访问外设的途径。

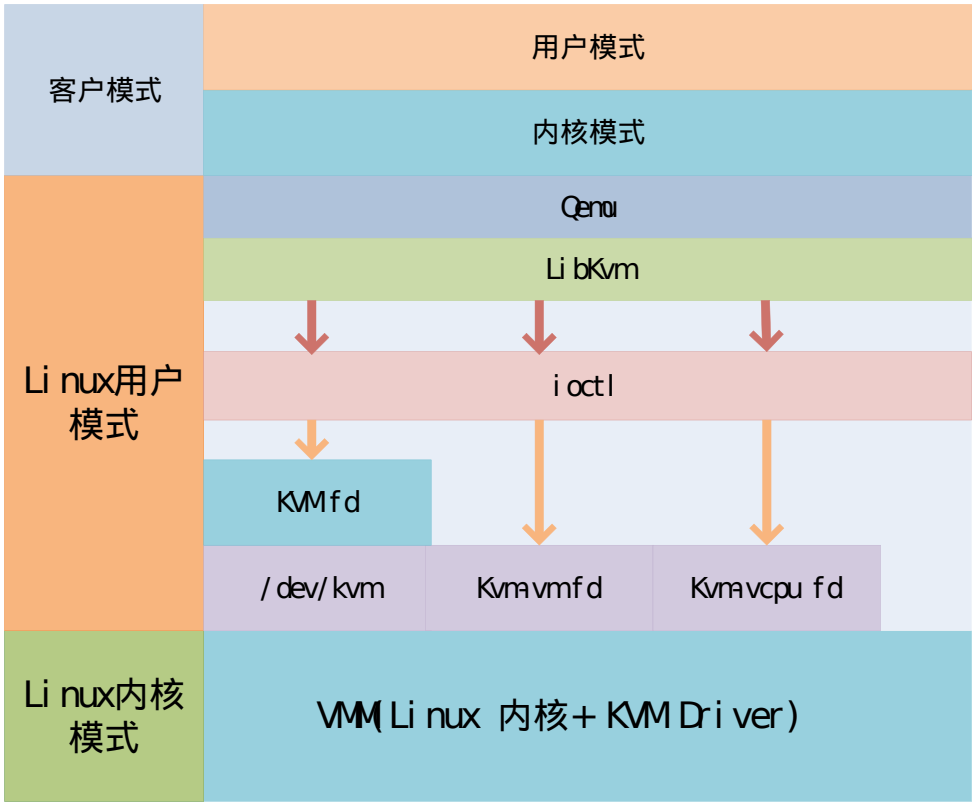


图 1KVM 基本结构

KVM 基本结构如图 1 所示。其中 KVM 加入到标准的 Linux 内核中，被组织成 Linux 中标准的字符设备(/dev/kvm)。Qemu 通 KVM 提供的 LibKvm 应用程序接口，通过 ioctl 系统调用创建和运行虚拟机。KVM Driver 使得整个 Linux 成为一个虚拟机监控器。并且在原有的 Linux 两种执行模式(内核模式和用户模式)的基础上，新增加了客户模式，客户模式拥有自己的内核模式和用户模式。在虚拟机运行下，三种模式的分工如下：

客户模式：执行非 I/O 的客户代码。虚拟机运行在客户模式下。

内核模式：实现到客户模式的切换。处理因为 I/O 或者其它指令引起的从客户模式的退出。KVM Driver 工作在这种模式下。

用户模式：代表客户执行 I/O 指令 Qemu 运行在这种模式下。

在 KVM 模型中，每一个 Guest OS 都作为一个标准的 Linux 进程，可以使用 Linux 的进程管理指令管理。

在图 1 中./dev/kvm 在内核中创建的标准字符设备，通过 ioctl 系统调用来访问内核虚拟机，进行虚拟机的创建和初始化；kvm_vm fd 是创建的指向特定虚拟机实例的文件描述符，通过这个文件描述符对特定虚拟机进行访问控制；kvm_vcpu fd 指向为虚拟机创建的虚拟处理器的文件描述符，通过该描述符使用 ioctl 系统调用设置和调度虚拟处理器的运行。

1.2KVM 工作原理

KVM 的基本工作原理：用户模式的 Qemu 利用接口 libkvm 通过 ioctl 系统调用进入内核模式。KVM Driver 为虚拟机创建虚拟内存和虚拟 CPU 后执行 VMLAUCH 指令进入客户模式。装载 Guest OS 执行。如果 Guest OS 发生外部中断或者影子页表缺页之类的事件，暂停 Guest OS 的执行，退出客户模式进行一些必要的处理。然后重新进入客户模式，执行客户代码。如果发生 I/O 事件或者信号队列中有信号到达，就会进入用户模式处理。KVM 采用全虚拟化技术。客户机不用修改就可以运行。

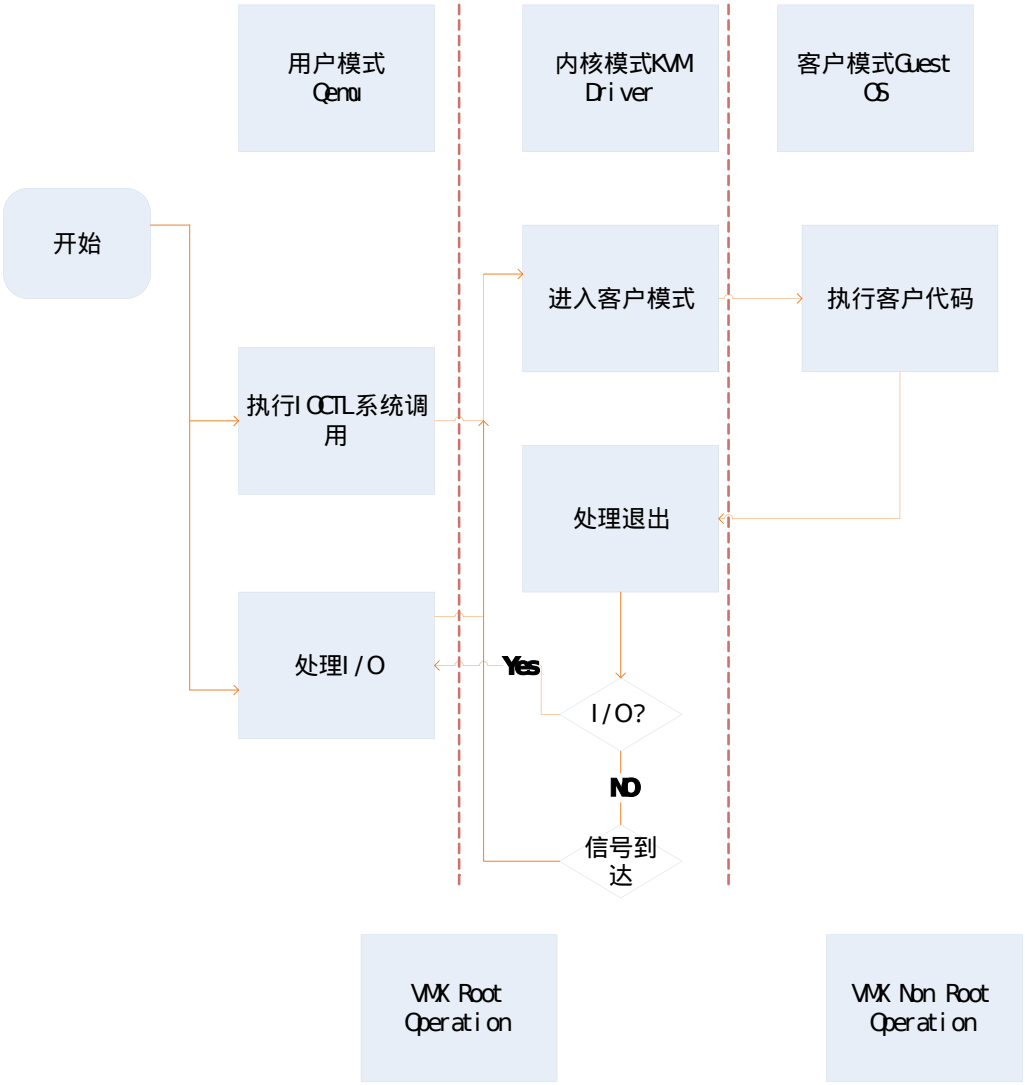


图 2 KVM 工作基本原理

2 ，相关技术-处理器管理和硬件辅助虚拟化技术

Intel 在 2006 年发布了硬件虚拟化技术。其中支持 X86 体系结构的称为 Intel VT-x 技术。ADM 称为 SVM 技术。

VT-x 引入了一种新的处理器操作，叫做 VMX（Virtual Machine Extension），提供了两种处理器的工作环境。VMCS 结构实现两种环境之间的切换。VM Entry 使虚拟机进去客户模式，VM Exit 使虚拟机退出客户模式。

2.1KVM 中 Guest OS 的调度执行

VMM 调度 Guest OS 执行时，Qemu 通过 ioctl 系统调用进入内核模式，在 KVM Driver 中通过 get_cpu 获得当前物理 CPU 的引用。之后将 Guest 状态从 VMCS 中读出。并装入物理 CPU 中。执行 VMLAUCH 指令使得物理处理器进入非根操作环境，运行客户代码。

当 Guest OS 执行一些特权指令或者外部事件时，比如 I/O 访问，对控制寄存器的操作，MSR 的读写数据包到达等。都会导致物理 CPU 发生 VMExit，停止运行 Guest OS。将 Guest OS 保存到 VMCS 中，Host 状态装入物理处理器中，处理器进入根操作环境，KVM 取得控制权，通过读取 VMCS 中 VM_EXIT_REASON 字段得到引起 VM Exit 的原因。从而调用 kvm_exit_handler 处理函数。如果由于 I/O 获得信号到达，则退出到用户模式的 Qemu 处理。处理完毕后，重新进入客户模式运行虚拟 CPU。如果是因为外部中断，则在 Lib KVM 中做一些必要的处理，重新进入客户模式执行客户代码。

2.2KVM 中内存管理

KVM 使用影子页表实现客户物理地址到主机物理地址的转换。初始为空，随着虚拟机页访问实效的增加，影子页表被逐渐建立起来，并随着客户机页表的更新而更新。在 KVM 中提供了一个哈希列表和哈希函数，以客户机页表项中的虚拟页号和该页表项所在页表的级别作为键值，通过该键值查询，如不为空，则表示该对应的影子页表项中的物理页号已经存在并且所指向的影子页表已经生成。如为空，则需新生成一张影子页表，KVM 将获取指向该影子页表的主机物理页号填充到相应的影子页表项的内容中，同时以客户机页表虚拟页号和表所在的级别生成键值，在代表该键值的哈希桶中填入主机物理页号，以备查询。但是一旦 Guest OS 中出现进程切换，会把整个影子页表全部删除重建，而刚被删掉的页表可能很快又被客户机使用，如果只更新相应的影子页表的表项，旧的影子页表就可以重用。因此在 KVM 中采用将影子页表中对应主机物理页的客户虚拟页写保护并且维护一张影子页表的逆向映射表，即从主机物理地址到客户虚拟地址之间的转换表，这样 VM 对页表或页目录的修改就可以触发一个缺页异常，从而被 KVM 捕获，对客户页表或页目录项的修改就可以同样作用于影子页表，通过这种方式实现影子页表与客户机页表保持同步。

2.3KVM 中设备管理

一个机器只有一套 I/o 地址和设备。设备的管理和访问是操作系统中的突出问题、同样也是虚拟机实现的难题，另外还要提供虚拟设备供各个 VM 使用。在 KVM 中通过移植 Qemu 中的设备模型(Device Model)进行设备的管理和访问。操作系统中，软件使用可编程 I/O（PIO）和内存映射 I/O（MMIO）与硬件交互。而且硬件可以发出中断请求，由操作系统处理。在有虚拟机的情况下，虚拟机必须要捕获并且模拟 PIO 和 MMIO 的请求，模拟虚拟硬件中断。

捕获 PIO：由硬件直接提供。当 VM 发出 PIO 指令时，导致 VM Exit 然后硬件会将 VM Exit 原因及对应的指令写入 VMCS 控制结构中，这样 KVM 就会模拟 PIO 指令。MMIO 捕获：对 MMIO 页的访问导致缺页异常，被 KVM 捕获，通过 X86 模拟器模拟执行 MMIO 指令。KVM 中的 I/O 虚拟化都是用户空间的 Qemu 实现的。所有 PIO 和 MMIO 的访问都是被转发到 Qemu 的。Qemu 模拟硬件设备提供给虚拟机使用。KVM 通过异步通知机制以及 I/O 指令的模拟来完成设备访问，这些通知包括：虚拟中断请求，信号驱动机制以及 VM 间的通信。

以虚拟机接收数据包来说明虚拟机和设备的交互。

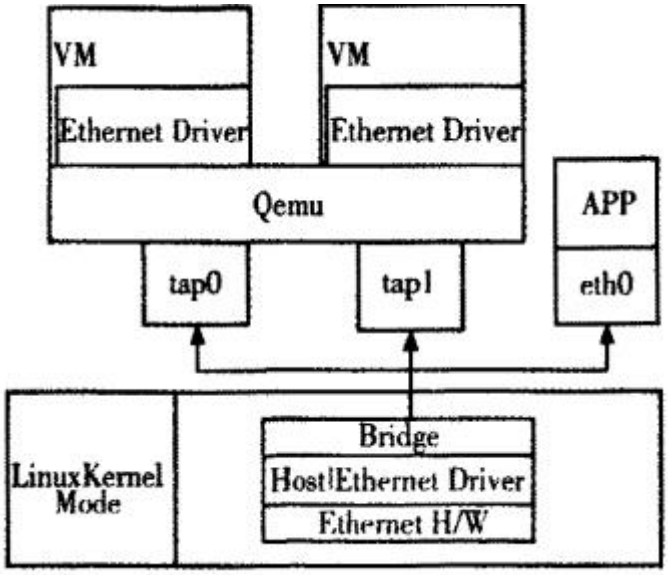


图 3 I/O 分析

(1)当数据包到达主机的物理网卡后，调用物理网卡的驱动程序，在其中利用 Linux 内核中的软件网桥，实现数据的转发。

(2)在软件网挢这一层，会判断数据包是发往那个设备的，同时调用网桥的发送函数，向对应的端口发送数据包。

(3)若数据包是发往虚拟机的，则要通过 tap 设备进行转发，tap 设备由两部分组成，网络设备和字符设备。网络设备负责接收和发送数据包，字符设备负责将数据包往内核空间和用户空间进行转发。Tap 网络部分收到数据包后，将其设备文件符置位，同时向正在运行 VM 的进程发出 I/O 可用信号，引起 VM Exit，停止 VM 运行，进入根操作状态。KVM 根据 KVM_EXIT_REASON 判断原因，模拟 I/O 指令的执行，将中断注入到 VM 的中断向量表中。

(4)返回用户模式的 Qemu 中，执行设备模型。返回到 kvm_main loop 中，执行 Kvm—main—loop—wait，之后进入 main_loop wait 中，在这个函数里收集对应设备的设备文件描述符的状态，此时 tap 设备文件描述符的状态同样被集到 fd set。

(5)Kvm main—loop 不停地循环，通过 select 系统调用判断哪嫂文件描述符的状态发生变化，相应的调用对应的处理函数。对予 tap 来说，就会通过 Qemu—send_packet 将数据发往 rtl8139 一 do—receiver，在这个函数中完成相当于硬件 RTL8139 网卡的逻辑操作。KVM 通过模拟 I／O 指令操作虚拟 RTL8139 将数据拷贝到用户地址空间，放在相应的 I／O 地址。用户模式处理完毕后返回内核模式，而后进入客户模式，VM 被再次执行，继续收发数据包。

3，KVM 源代码分析-虚拟机创建和运行流程代码分析

3.1 KVM 创建和运行虚拟机流程

KVM 虚拟机创建和运行虚拟机分为用户态和核心态两个部分，用户态主要提供应用程序接口，为虚拟机创建虚拟机上下文环境，在 libkvm 中提供访问内核字符设备/dev/kvm 的接口；内核态为添加到内核中的字符设备/dev/kvm，模块加载进内核后即可进行接口用户空间调用创建虚拟机。在创建虚拟机过程中，kvm 字符设备主要为客户机创建 kvm 数据机构，创建该虚拟机的虚拟机文件描述符及其相应的数据结构以及创建虚拟处理器及其相应的数据结构。Kvm 创建虚拟机的流程如图 4 所示。

首先申明一个 kvm_context_t 变量用以描述用户态虚拟机上下文信息，然后调用 kvm_init()函数初始化虚拟机上下文信息；函数 kvm_create()创建虚拟机实例，该函数通过 ioctl 系统调用创建虚拟机相关的内核数据结构并且返回虚拟机文件描述符给用户态 kvm_context_t 数据结构；创建完内核虚拟机数据结构后，再创建内核 pit 以及 mmio 等基本外设模拟设备，然后调用 kvm_create_vcpu()函数来创建虚拟处理器，kvm_create_vcpu()函数通过 ioctl()系统调用向由 vm_fd 文件描述符指向的虚拟文件调用创建虚拟处理器，并将虚拟处理器的文件描述符返回给用户态程序，用以以后的调度使用；创建完虚拟处理器后，由用户态的 QEMU 程序申请客户机用户空间，用以加载和运行客户机代码；为了使得客户虚拟机正确执行，必须要在内核中为客户机建立正确的内存映射关系，即影子页表信息。因此，申请客户机内存地址空间后，调用函数 kvm_create_phys_mem()创建客户机内存映射关系，该函数主要通过 ioctl 系统调用向 vm_fd 指向的虚拟文件调用设置内核数据结构中客户机内存域相关信息，主要建立影子页表信息；当创建好虚拟处理器和影子页表后，即可读取客户机到指定分配的空间中，然后调度虚拟处理器运行。调度虚拟机的函数为 kvm_run()，该函数通过 ioctl 系统调用调用由虚拟处理器文件描述符指向的虚拟文件调度处理函数 kvm_run()调度虚拟处理器的执行，该系统调用将虚拟处理器 vcpu 信息加载到物理处理器中，通过 vm_entry 执行进入客户机执行。在客户机正常运行期间 kvm_run()函数不返回，只有发生以下两种情况时，函数返回：1，发生了 I/O 事件，如客户机发出读写 I/O 的指令；2，产生了客户机和内核 KVM 都无法处理的异常。I/O 事件处理完毕后，通过重新调用 KVM_RUN()函数继续调度客户机的执行。

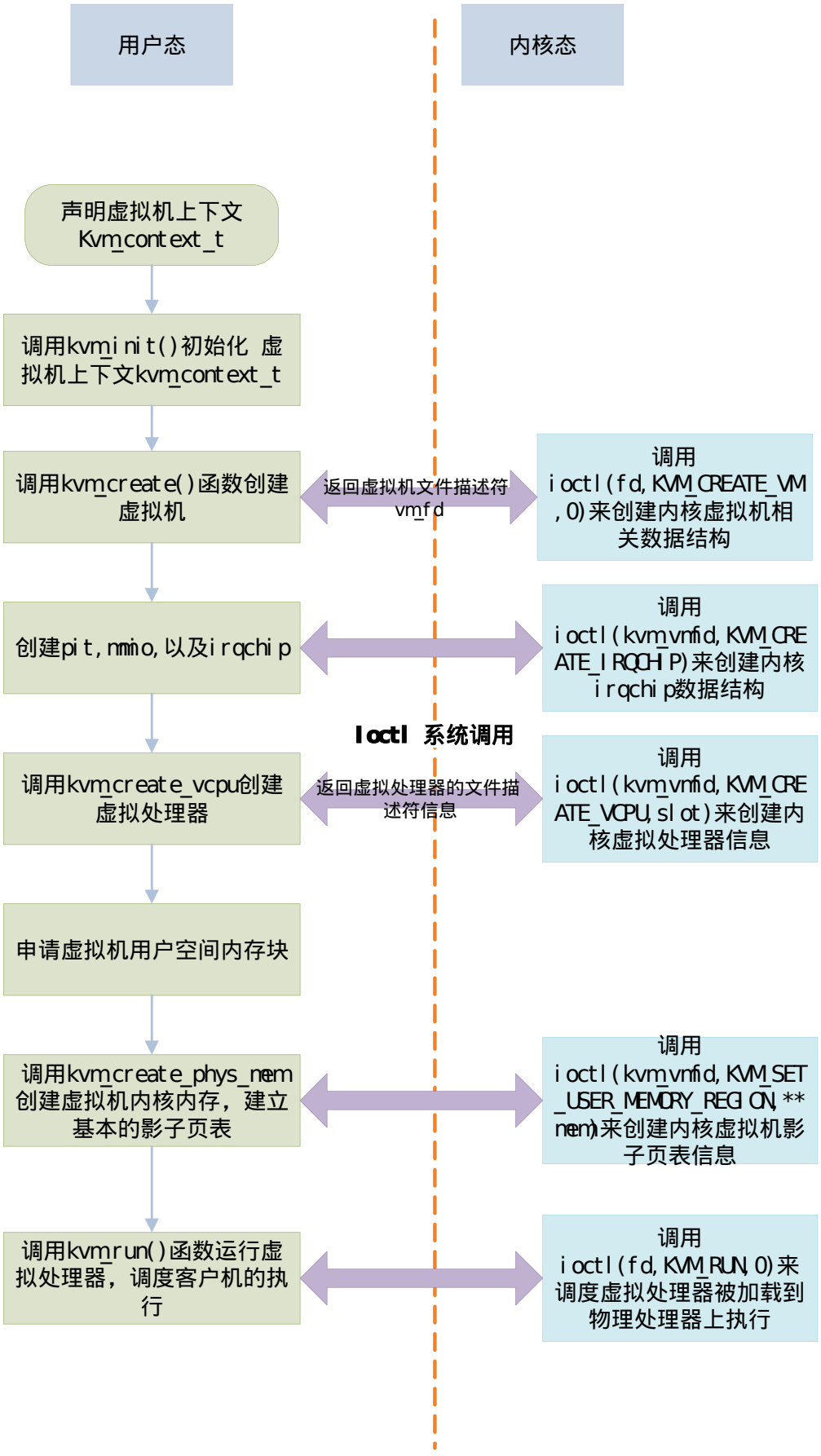


图 4 KVM 虚拟机创建流程

3.2 虚拟机创建和运行主要函数分析

1，函数 kvm_init()：该函数在用户态创建一个虚拟机上下文，用以在用户态保存基本的虚拟机信息，这个函数是创建虚拟机第一个需要调用的函数，函数返回一个 kvm_context_t 结构体。该函数原型为：

Kvm_context_t kvm_init(struct kvm_callbacks *callbacks,void *opaque);

参数：callbacks 为结构体 kvm_callbacks 变量，该结构体包含指向函数的一组指针，用于在客户机执行过程中因为 I/O 事件退出到用户态的时候处理的回调函数(后面会分析)。参数 opaque 一般未使用。

函数执行基本过程：打开字符设备 dev/kvm,申请虚拟机上下文变量 kvm_context_t 空间，初始化上下文的基本信息：设置 fd 文件描述符指向/dev/kvm、禁用虚拟机文件描述符 vm_fd(-1)

设置 I/O 事件回调函数结构体，设置 IRQ 和 PIT 的标志位以及内存页面记录的标志位。

主要相关数据结构：

虚拟机上下文 structkvm_context_t，用户态数据结构，用以描述虚拟机实例的用户态上下文信息。

Struct kvm_context： 该结构体用于表示一个虚拟机上下文。主要包含的数据域为：

Int fd :指向内核标准字符设备/dev/kvm 的文件描述符。

Int vm_fd:指向所创建的内核虚拟机数据结构相关文件的文件描述符。

Int vcpu_fd[MAX_VCPUS]:指向虚拟机所有的虚拟处理器的文件描述符数组。

Struct kvm_run *run[MAX_VCPUS]：指向虚拟机运行环境上下文的指针数组。

Struct kvm_callbacks *call_backs: 回调函数结构体指针，该结构体用于处理用户态 I/O 事件。

Void *opaque:指针(还未弄清楚)

Int dirty_page_log_all:设置是否记录脏页面的标志。

Int no_ira_creation: 用于设置是否再 kernel 里设置 irq 芯片。

Int_irqchip_in_kernel: 内核中 irqchip 的状态

Int irqchip_inject_ioctl:用于拦截中断的 iotcl 系统调用

Int no_pit_creation: 设置是否再内核中设置陷阱

int pit_in_kernel：PIT 状态

int coalesced_mmio：kernel 中 mmio

struct kvm_irq_routing *irq_routes：KVM 中中断处理的路由结构指针。

int nr_allocated_irq_routes：分配给该虚拟机的中断路由数目。

int max_used_gsi：使用的最大的 gsi(?)。

用户态 I/O 处理函数结构体 struct kvm_callbacks，该结构体指向一组函数，主要用于处理客户机因为 I/O 事件退出而执行的过程。

struct kvm_callbacks :该结构体用于在用户态中处理 I/O 事件，在 KVM 中调用 KVM_QEMU 实现。主要包含的数据域为：

int (*inb)(void *opaque, uint16_t addr, uint8_t *data)：用于模拟客户机执行 8 位的 inb 指令。

int (*inw)(void *opaque, uint16_t addr, uint16_t *data)：用于模拟客户机执行 16 位的 inw 指令。

int (*inl)(void *opaque, uint16_t addr, uint32_t *data)：用于模拟客户机执行 32 位的 inl 指令。

int (*outb)(void *opaque, uint16_t addr, uint8_t data)：用于模拟客户机执行 8 位的 outb 指令。

int (*outw)(void *opaque, uint16_t addr, uint16_t data)：用于模拟客户机执行 16 位的 outw 指令。
int (*outl)(void *opaque, uint16_t addr, uint32_t data)：用于模拟客户机执行 32 位的 outl 指令。
int (*mmio_read)(void *opaque, uint64_t addr, uint8_t *data,int len)：用于模拟客户机执行 mmio 读指令。
int (*mmio_write)(void *opaque, uint64_t addr, uint8_t *data,int len)：用于模拟客户机执行 mmio 写指令。
int (*debug)(void *opaque, void *env, struct kvm_debug_exit_arch *arch_info)：用户客户机调试的回调函数。
int (*halt)(void *opaque, int vcpu)：用于客户机执行 halt 指令的响应。
int (*shutdown)(void *opaque, void *env)：用于客户机执行 shutdown 指令的响应。
int (*io_window)(void *opaque)：用于获得客户机 io_windows。
int (*try_push_interrupts)(void *opaque)：用于注入中断的回调函数。
void (*push_nmi)(void *opaque)：用于注入 nmi 中断的函数。
void (*post_kvm_run)(void *opaque, void *env);用户得到 kvm 运行状态函数。
int (*pre_kvm_run)(void *opaque, void *env);用于获得 kvm 之前运行状态的函数
int (*tpr_access)(void *opaque, int vcpu, uint64_t rip, int is_write);获得 tpr 访问处理函数
int (*powerpc_dcr_read)(int vcpu, uint32_t dcrn, uint32_t *data);用于 powerpc 的 dcr 读操作
nt (*powerpc_dcr_write)(int vcpu, uint32_t dcrn, uint32_t data);用于 powerpc 的 dcr 写操作
int (*s390_handle_intercept)(kvm_context_t context, int vcpu,struct kvm_run *run);用于 s390 的中断处理。
int (*s390_handle_reset)(kvm_context_t context, int vcpu,struct kvm_run *run);用于 s390 的重设处理。
}

当客户机执行 I/O 事件或者停机操作等事件时，KVM 会交给用户态的 QEMU 模拟外部 I/O 事件，调用这个结构体指向的相关的函数进行处理。
Struct kvm_run: 用于 KVM 运行时一些的一些状态信息。主要包含的数据域为：
__u8 request_interrupt_window;
__u8 padding1[7];
__u32 exit_reason;
__u8 ready_for_interrupt_injection;
__u8 if_flag;
__u8 padding2[2];
/* in (pre_kvm_run), out (post_kvm_run) */
__u64 cr8;
__u64 apic_base;
union {
/* KVM_EXIT_UNKNOWN */
struct {
__u64 hardware_exit_reason; 记录退出原因
} hw;
/* KVM_EXIT_FAIL_ENTRY */ 客户机执行过程中执行 VM_ENTRY 失败。

```
struct {
        __u64 hardware_entry_failure_reason;
} fail_entry;
/* KVM_EXIT_EXCEPTION */   客户机因为异常退出
struct {
        __u32 exception;
        __u32 error_code;
} ex;
/* KVM_EXIT_IO */     客户机因为 IO 事件退出。
struct kvm_io {
#define KVM_EXIT_IO_IN    0
#define KVM_EXIT_IO_OUT 1
__u8 direction;
__u8 size; /* bytes */
__u16 port;
__u32 count;
__u64 data_offset; /* relative to kvm_run start */
} io;
struct {
struct kvm_debug_exit_arch arch;
} debug;
/* KVM_EXIT_MMIO */ 客户机因为 MMIO 退出
struct {
__u64 phys_addr;
__u8    data[8];
__u32 len;
__u8    is_write;
} mmio;
/* KVM_EXIT_HYPERCALL */ 客户机退出的超调用参数。
struct {
__u64 nr;
__u64 args[6];
__u64 ret;
__u32 longmode;
__u32 pad;
} hypercall;
            /* KVM_EXIT_TPR_ACCESS */ 客户机退出访问 TPR 参数
struct {
        __u64 rip;
        __u32 is_write;
        __u32 pad;
} tpr_access;
/* KVM_EXIT_S390_SIEIC */   和 S390 相关数据
struct {
```

```
__u8 icptcode;
__u64 mask; /* psw upper half */
__u64 addr; /* psw lower half */
__u16 ipa;
__u32 ipb;
} s390_sieic;
/* KVM_EXIT_S390_RESET */
#define KVM_S390_RESET_POR           1
#define KVM_S390_RESET_CLEAR         2
#define KVM_S390_RESET_SUBSYSTEM 4
#define KVM_S390_RESET_CPU_INIT    8
#define KVM_S390_RESET_IPL           16
__u64 s390_reset_flags;
/* KVM_EXIT_DCR */
struct {
        __u32 dcrn;
        __u32 data;
        __u8    is_write;
} dcr;
/* Fix the size of the union. */
char padding[256];
```

2， 函数 kvm_create()：该函数主要用于创建一个虚拟机内核环境。该函数原型为：

int kvm_create(kvm_context_t kvm,unsigned long phys_mem_bytes, void **phys_mem);

参数：kvm_context_t 表示传递的用户态虚拟机上下文环境，phys_mem_bytes 表示需要创建的物理内存的大小，phys_mem 表示创建虚拟机的首地址。这个函数首先调用 kvm_create_vm()分配 IRQ 并且初始化为 0,设置 vcpu[0]的值为-1，即不允许调度虚拟机执行。然后调用 ioctl 系统调用 ioctl(fd,KVM_CREATE_VM,0)来创建虚拟机内核数据结构 struct kvm。

3,系统调用函数 ioctl(fd,KVM_CREATE_VM,0)，用于在内核中创建和虚拟机相关的数据结构。该函数原型为：

Static long kvm_dev_ioctl(struct file *filp，unsigned int ioctl， unsigned long arg);其中 ioctl 表示命令。这个函数调用 kvm_dev_ioctl_create_vm()创建虚拟机实例内核相关数据结构。该函数首先通过内核中 kvm_create_vm()函数创建内核中 kvm 上下文 struct kvm，然后通过函数 Anno_inode_getfd("kvm_vm",&kvm_vm_fops,kvm,0)返回该虚拟机的文件描述符，返回给用户调用函数，由 2 中描述的函数赋值给用户态虚拟机上下文变量中的虚拟机描述符 kvm_vm_fd。

4，内核创建虚拟机 kvm 对象后，接着调用 kvm_arch_create 函数用于创建一些体系结构相关的信息，主要包括 kvm_init_tss、kvm_create_pit 以及 kvm_init_coalsced_mmio 等信息。然后调用 kvm_create_phys_mem 创建物理内存，函数 kvm_create_irqchip 用于创建内核 irq 信息，通过系统调用 ioctl(kvm->vm_fd,KVM_CREATE_IRQCHIP)。

5，函数 kvm_create_vcpu()：用于创建虚拟处理器。该函数原型为：

int kvm_create_vcpu(kvm_context_t kvm, int slot);

参数：kvm 表示对应用户态虚拟机上下文，slot 表示需要创建的虚拟处理器的个数。

该函数通过 ioctl 系统调用 ioctl(kvm->vm_fd,KVM_CREATE_VCPU，slot)创建属于该虚拟机

的虚拟处理器。该系统调用函数：
Static init kvm_vm_ioctl_create_vcpu(struct *kvm, n) 参数 kvm 为内核虚拟机实例数据结构，n 为创建的虚拟 CPU 的数目。
6，函数 kvm_create_phys_mem()用于创建虚拟机内存空间，该函数原型：
Void * kvm_create_phys_mem(kvm_context_t kvm,unsigned long phys_start,unsigned len,int log,int writable);
参数：kvm 表示用户态虚拟机上下文信息，phys_start 为分配给该虚拟机的物理起始地址，len 表示内存大小，log 表示是否记录脏页面，writable 表示该段内存对应的页表是否可写。该函数首先申请一个结构体 kvm_userspace_memory_region 然后通过系统调用 KVM_SET_USER_MEMORY_REGION 来设置内核中对应的内存的属性。该系统调用函数原型：
Ioctl(int kvm->vm_fd，KVM_SET_USER_MEMORY_REGION，&memory);
参数：第一个参数 vm_fd 为指向内核虚拟机实例对象的文件描述符，第二个参数 KVM_SET_USER_MEMORY_REGION 为系统调用命令参数，表示该系统调用为创建内核客户机映射，即影子页表。第三个参数 memory 表示指向该虚拟机的内存空间地址。系统调用首先通过参数 memory 通过函数 copy_from_user 从用户空间复制 struct_user_momory_region 变量，然后通过 kvm_vm_ioctl_set_memory_region 函数设置内核中对应的内存域。该函数原型：
Int kvm_vm_ioctl_set_memory_region(struct *kvm,struct kvm_usersapce_memory_region *mem,int user_alloc);该函数再调用函数 kvm_set_memory_resgion()设置影子页表。当这一切都准备完毕后，调用 kvm_run()函数即可调度执行虚拟处理器。
7，函数 kvm_run()：用于调度运行虚拟处理器。该函数原型为：
Int kvm_run(kvm_context_t kvm,int vcpu, void *env) 该函数首先得到 vcpu 的描述符，然后调用系统调用 ioctl(fd,kvm_run,0)调度运行虚拟处理器。Kvm_run 函数在正常运行情况下并不返回，除非发生以下事件之一：一是发生了 I/O 事件，I/O 事件由用户态的 QEMU 处理；一个是发生了客户机和 KVM 都无法处理的异常事件。KVM_RUN()中返回截获的事件，主要是 I/O 以及停机等事件。

4, KVM 客户机异常处理机制和代码流程

KVM 保证客户机正确执行的基本手段就是当客户机执行 I/O 指令或者其它特权指令时，引发处理器异常，从而陷入到根操作模式，由 KVM Driver 模拟执行，可以说，虚拟化保证客户机正确执行的基本手段就是异常处理机制。由于 KVM 采取了硬件辅助虚拟化技术，因此，和异常处理机制相关的一个重要的数据结构就是虚拟机控制结构 VMCS。

VMCS 控制结构分为三个部分，一个是版本信息，一个是中止标识符，最后一个是 VMCS 数据域。VMCS 数据域包含了六类信息：客户机状态域，宿主机状态域，VM-Entry 控制域，VM-Execution 控制域 VM-Exit 控制域以及 VM-Exit 信息域。其中 VM-Execution 控制域可以设置一些可选的标志位使得客户机可以引发一定的异常的指令。宿主机状态域，则保持了基本的寄存器信息，其中 CS:RIP 指向 KVM 中异常处理程序的地址。是客户机异常处理的总入口，而异常处理程序则根据 VM-Exit 信息域来判断客户机异常的根本原因，选择正确的处理逻辑来进行处理。

vmx.c 文件是和 Intel VT-x 体系结构相关的代码文件，用于处理内核态相关的硬件逻辑代码。在 VCPU 初始化中(vmx_vcpu_create)，将 kvm 中的对应的异常退出处理函数赋值给 CS:EIP 中，在客户机运行过程中，产生客户机异常时，CPU 根据 VMCS 中的客户机状态域

装载 CS:EIP 的值，从而退出到内核执行异常处理。在 KVM 内核中，异常处理函数的总入口为：

static int vmx_handle_exit(struct kvm_run *kvm_run, struct kvm_vcpu *vcpu);

参数：kvm_run,当前虚拟机实例的运行状态信息，vcpu,对应的虚拟 cpu。

这个函数首先从客户机 VM-Exit 信息域中读取 exit_reason 字段信息，然后进行一些必要的处理后，调用对应于函数指针数组中对应退出原因字段的处理函数进行处理。函数指针数组定义信息为：

```
static int (*kvm_vmx_exit_handlers[])(struct kvm_vcpu *vcpu,
                                      struct kvm_run *kvm_run) = {
    [EXIT_REASON_EXCEPTION_NMI]           =handle_exception,
    [EXIT_REASON_EXTERNAL_INTERRUPT]      =handle_external_interrupt,
    [EXIT_REASON_TRIPLE_FAULT]            =handle_triple_fault,
    [EXIT_REASON_NMI_WINDOW]              =handle_nmi_window,
    [EXIT_REASON_IO_INSTRUCTION]          =handle_io,
    [EXIT_REASON_CR_ACCESS]               =handle_cr,
    [EXIT_REASON_DR_ACCESS]               =handle_dr,
    [EXIT_REASON_CPUID]                   =handle_cpuid,
    [EXIT_REASON_MSR_READ]                =handle_rdmsr,
    [EXIT_REASON_MSR_WRITE]               =handle_wrmsr,
    [EXIT_REASON_PENDING_INTERRUPT]       =handle_interrupt_window,
    [EXIT_REASON_HLT]                     =handle_halt,
    [EXIT_REASON_INVLPG]                  =handle_invlpg,
    [EXIT_REASON_VMCALL]                  =handle_vmcall,
    [EXIT_REASON_TPR_BELOW_THRESHOLD]     =handle_tpr_below_threshold,
    [EXIT_REASON_APIC_ACCESS]             =handle_apic_access,
    [EXIT_REASON_WBINVD]                  =handle_wbinvd,
    [EXIT_REASON_TASK_SWITCH]             =handle_task_switch,
    [EXIT_REASON_EPT_VIOLATION]           =handle_ept_violation,
};
```

这是一组指针数组，用于处理客户机引发异常时候，根据对应的退出字段选择处理函数进行处理。例如 EXIT_REASON_EXCEPTION_NMI 对应的 handle_exception 处理函数用于处理 NMI 引脚异常，而 EXIT_REASON_EPT_VIOLATION 对应的 handle_ept_violation 处理函数用于处理缺页异常。

5, KVM 客户机影子页表机制和缺页处理机制及其代码流程

5.1 KVM 影子页表基本原理和缺页处理流程

KVM 给客户机呈现了一个从 0 开始的连续的物理地址空间，成为客户机物理地址空间，在宿主机中，只是由 KVM 分配给客户机的宿主机物理内存的一部分。客户机中为应

用程序建立的客户机页表，只是实现由客户机虚拟地址到客户机物理地址的转换（GVA-GPA），为了正确的访问内存，需要进行二次转换，即客户机物理地址到宿主机物理地址的转换（GPA-HPA），两个地址转换，前者由客户机负责完成，后者由 VMM 负责完成。为了实现客户机物理地址到主机物理地址的地址翻译，VMM 为每个虚拟机动态维护了一张从客户机物理地址到宿主机物理地址的映射关系表。客户机OS所维护的页表只是负责传统的客户机虚拟地址到客户机物理地址的转换。如果 MMU 直接装在客户机操作系统所维护的页表进行内存访问，硬件则无法争取的实现地址翻译。

影子页表是一个有效的解决办法。一份影子页表与一份客户机操作系统的页表相对应，作用是实现从客户虚拟机地址到宿主机物理地址的直接翻译。这样，客户机所能看到和操作的都是虚拟 MMU，而真正载入到屋里 MMU 的是影子页表。如图 5所示。

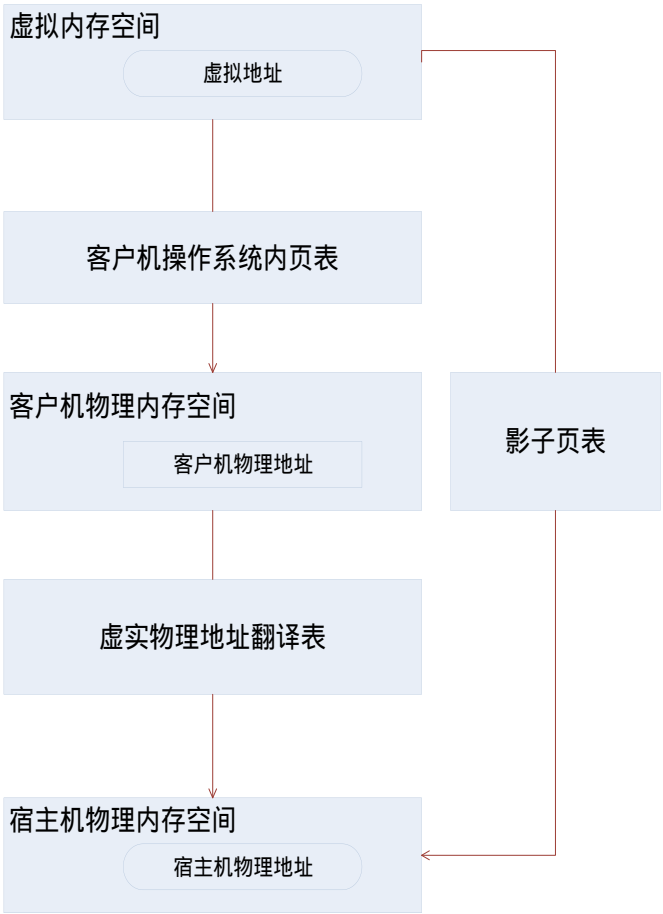


图 5-影子页表

由于客户机维护的页表真正体现在影子页表上，因此，客户机对客户机页表的操作实质上反映到影子页表中，并且由 VMM 控制，因此，影子页表中，对于页表页的访问权限是只读的。一旦客户机对客户机页表进行修改，则产生页面异常，由 VMM 处理。

影子页表的结构：以 x86 的两级页表为例。客户机物理帧号 GFN 和宿主机物理帧号 MFN 是一一对应的，而宿主机必须要另外负责创建一个页用于 CR3 寄存器指向的影子页表 SMFN。通常通过 hash 表实现 MFN 和 SMFN 之间的计算。影子页表的建立和修改过程交错在客户机对客户机页表的创建过程中。

影子页表的缺页异常处理：当发生缺页异常时，产生异常条件并被 VMM 捕获，让 VMM 发现客户机操作系统尚未给客户机虚拟机地址分配客户机物理页，那么 VMM 首先将缺页异常传递给客户机操作系统，由客户机操作系统为这个客户机虚拟地址分配客户机物

理页，由于客户机操作系统分配客户机物理页需要修改其页表，因为又被 VMM 捕获，VMM 更新影子页表中相应的目录和页表项，增加这个客户机虚拟地址到分配的宿主页的映射。如果产生异常时，VMM 发现客户机操作系统已经分配给了相应的客户机物理页，只是写权限造成写异常，则 VMM 更新影子页表中的页目录和页表项，重定向客户机虚拟地址页到宿主机页的映射。客户机缺页异常处理逻辑流程如图 6 所示。

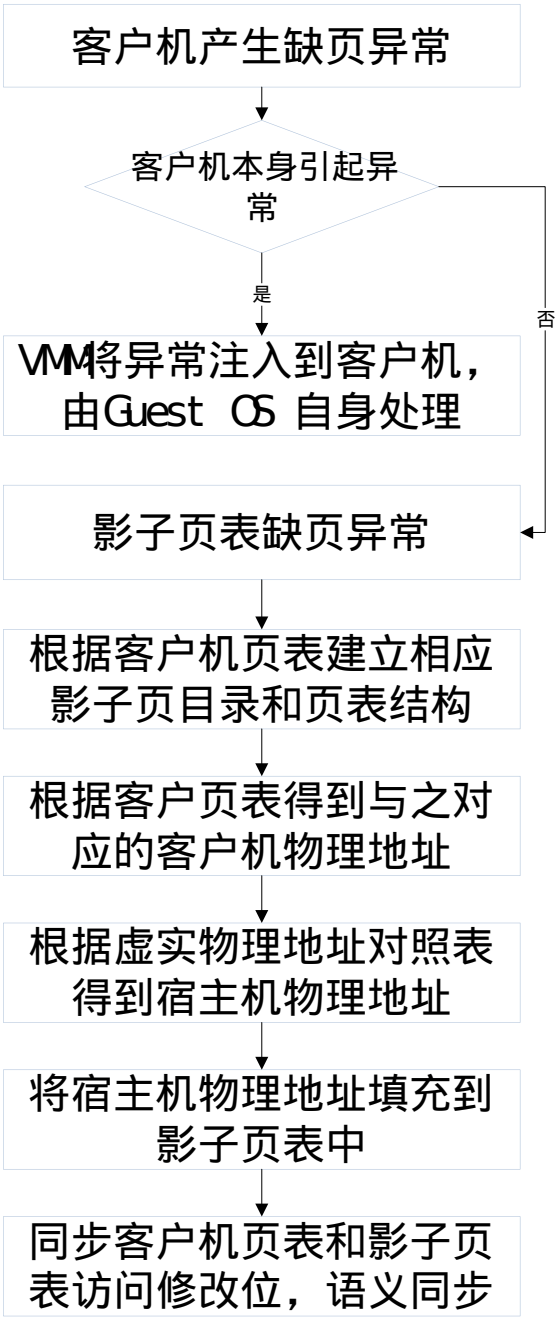


图 6– 缺页异常处理流程

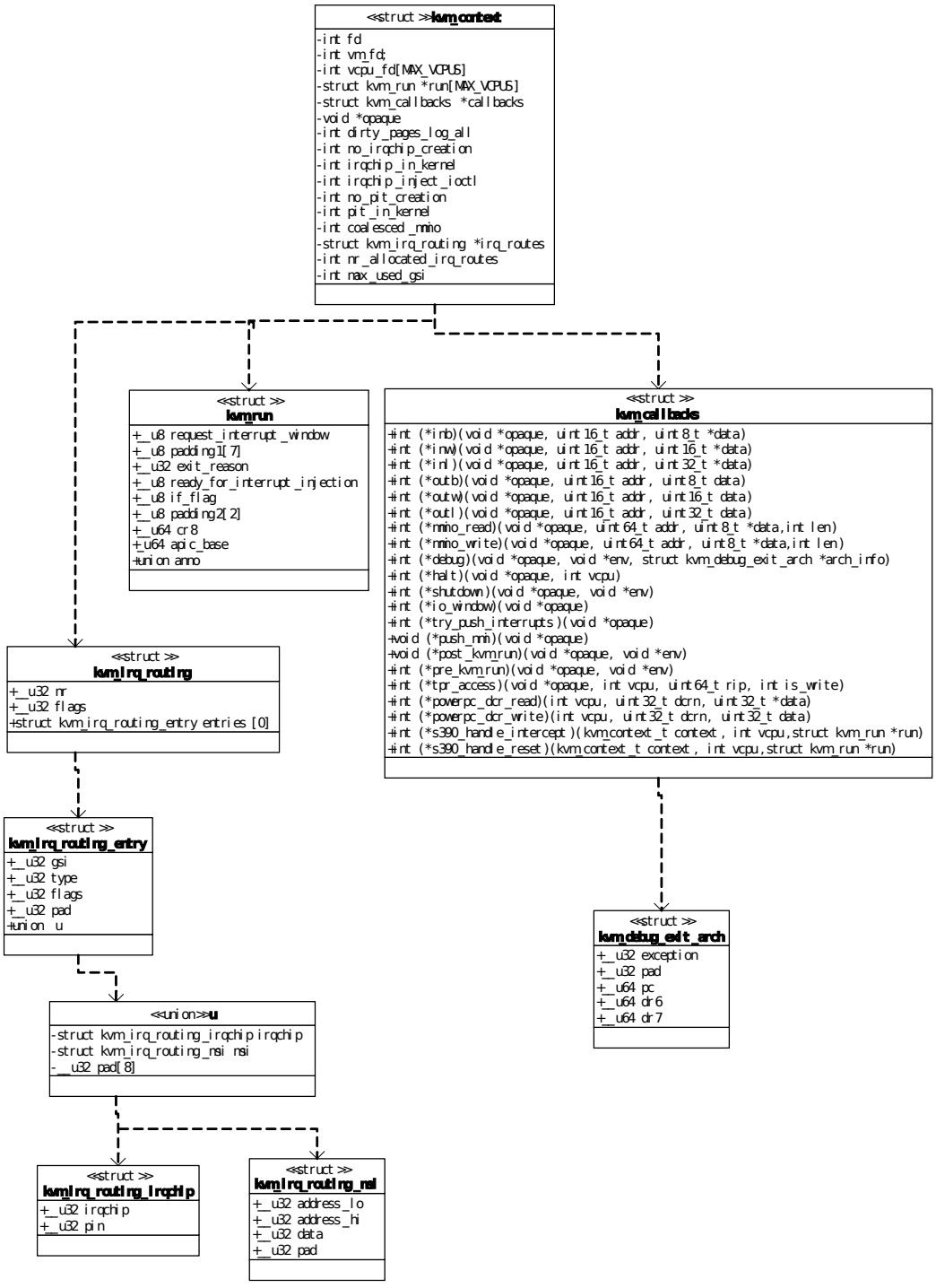
5.2 KVM 中缺页异常的处理代码流程

由 4 中说明的异常处理逻辑中，vmx_handle_exit 函数用于处理客户机异常的总入口程序，KVM 根据退出域中的退出原因字段，调用函数指针数据中对应的处理函数进行处理，

在缺页异常中，处理函数为 handle_ept_violation。该函数调用 kvm_mmu_page_fault 函数进行缺页异常处理。

6, KVM 中主要数据结构以及相互关系

6.1 用户态关键数据机构



KVM 用户态重要数据结构

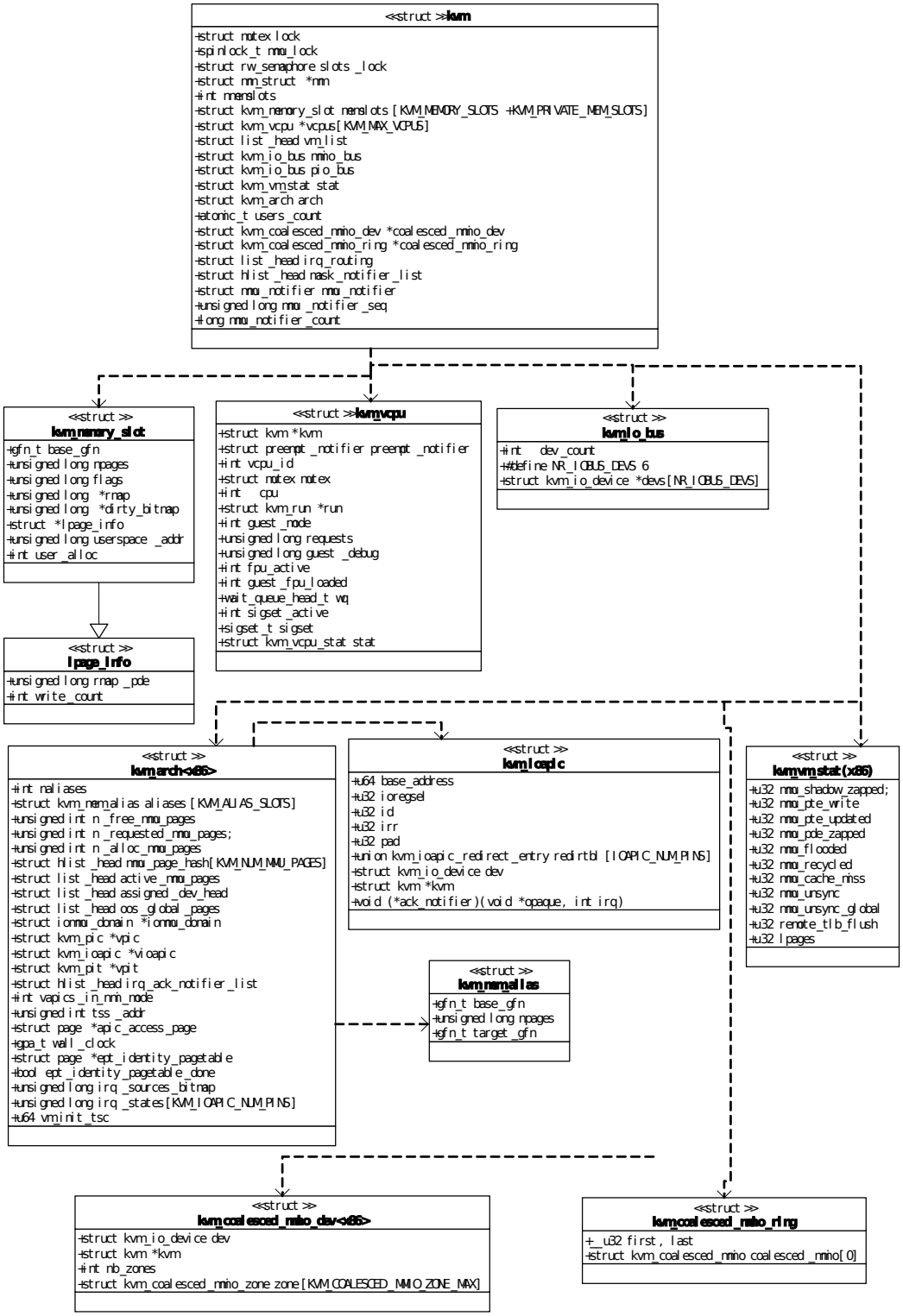
KVM-CONTEXT-T 为虚拟机上下文

KVM-run 为虚拟机运行状态

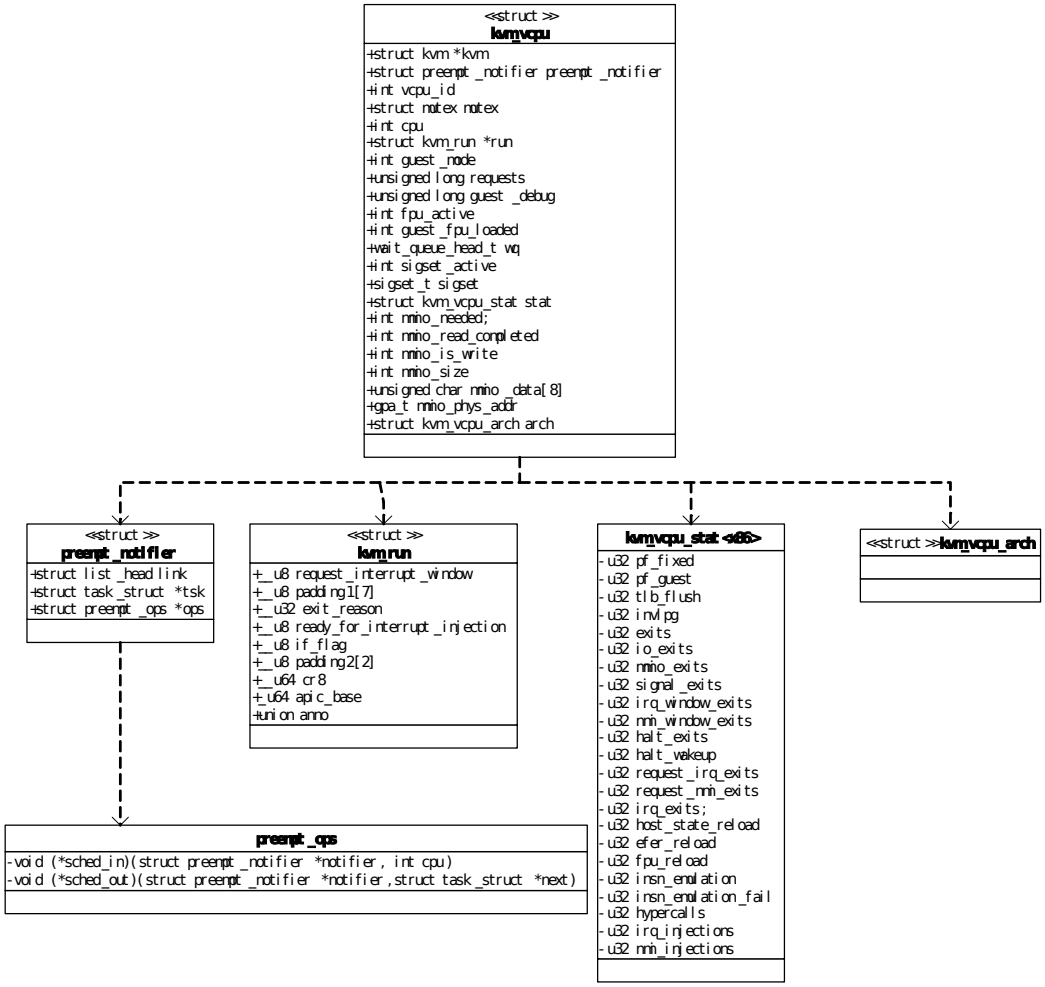
Kvm-callbacks 为虚拟机产生 I/O 事件处理的回调函数

Kvm-irq-routing 为中断处理相关数据结构

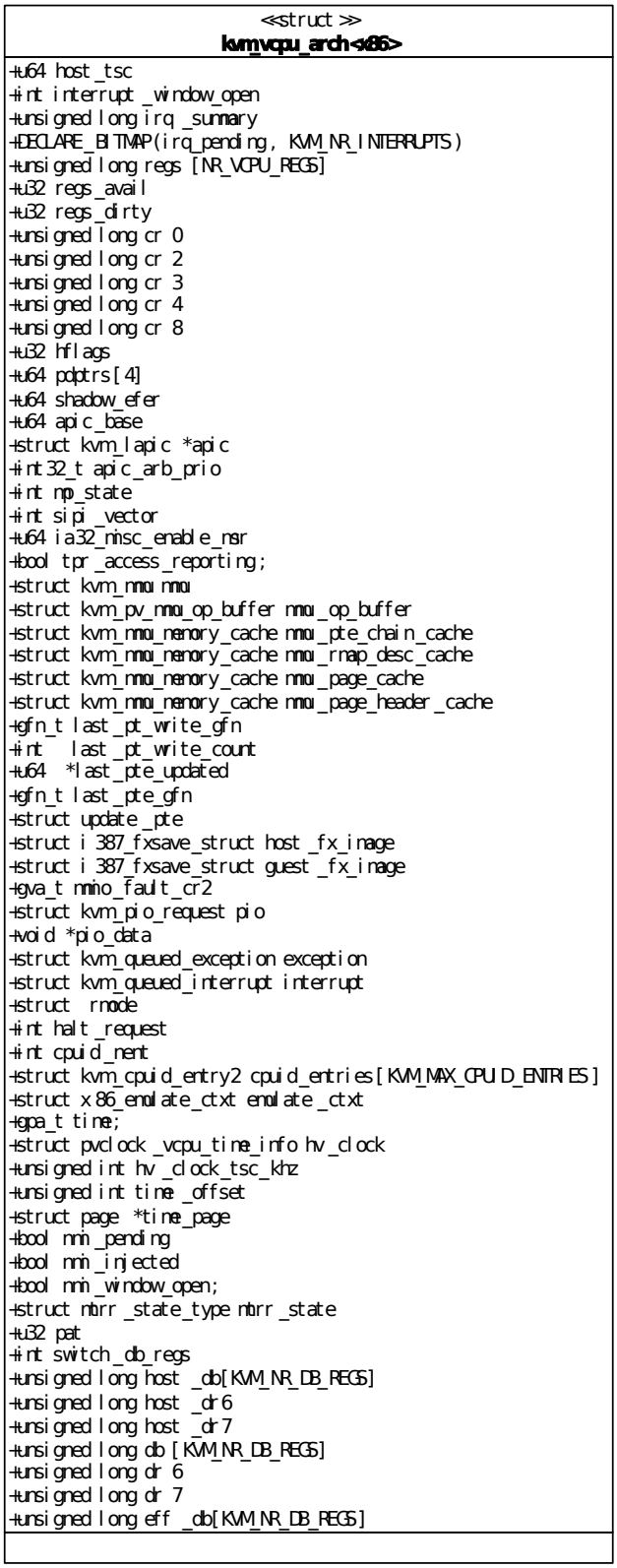
6.2 内核态关键数据机构



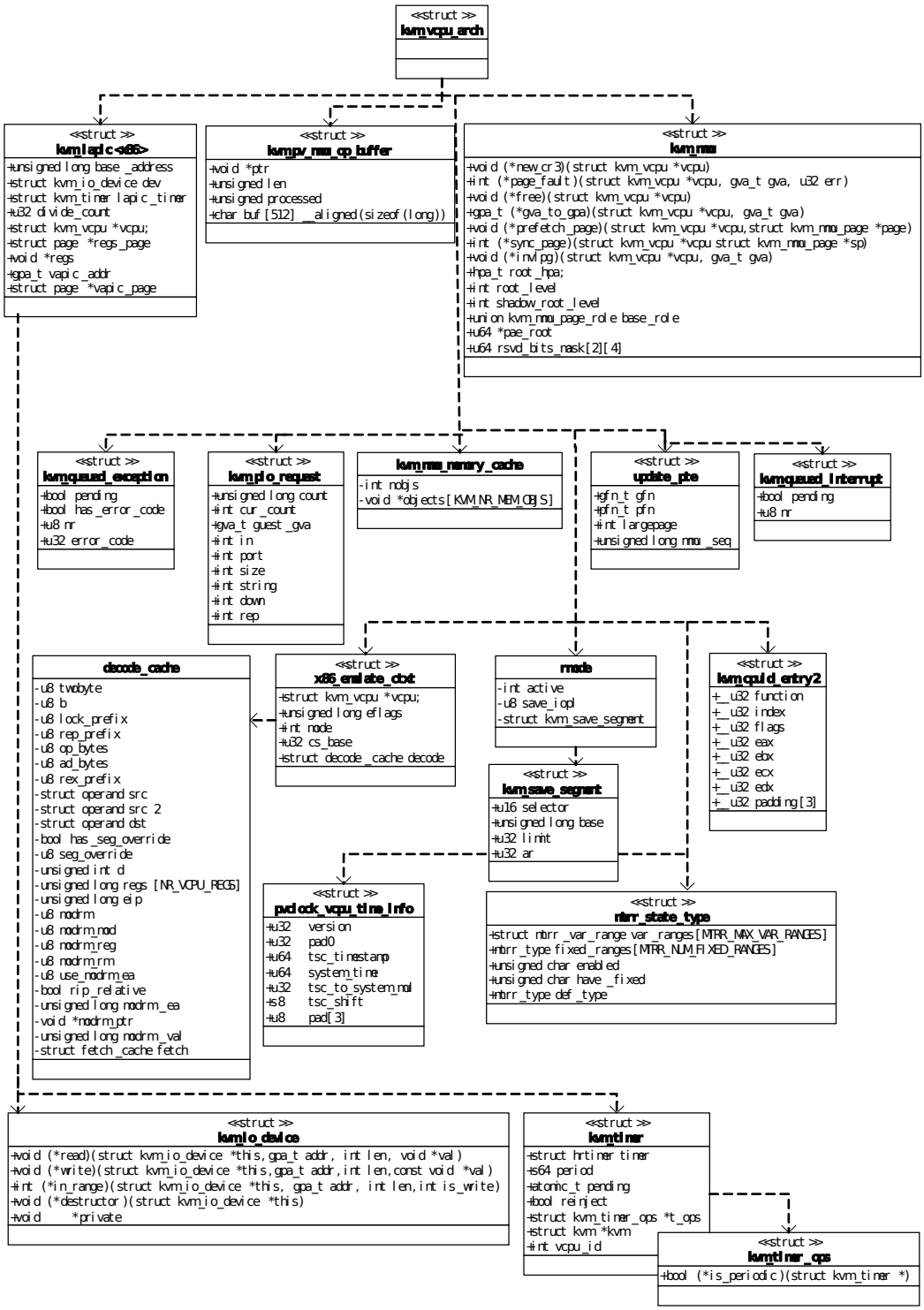
KVM 结构体定义


«struct»
kvm_vcpu
+struct kvm *kvm
+struct preempt_notifier preempt_notifier
+int vcpu_id
+struct mutex mutex
+int cpu
+struct kvm_run *run
+int guest_mode
+unsigned long requests
+unsigned long guest_debug
+int fpu_active
+int guest_fpu_loaded
+wait_queue_head_t wq
+int sigset_active
+sigset_t sigset
+struct kvm_vcpu_stat stat
+int mmio_needed;
+int mmio_read_completed
+int mmio_is_write
+int mmio_size
+unsigned char mmio_data[8]
+gpa_t mmio_phys_addr
+struct kvm_vcpu_arch arch
«struct»
preempt_notifier
+struct list_head link
+struct task_struct *tsk
+struct preempt_ops *ops
«struct»
kvm_run
+__u8 request_interrupt_window
+__u8 padding1[7]
+__u32 exit_reason
+__u8 ready_for_interrupt_injection
+__u8 if_flag
+__u8 padding2[2]
+__u64 cr8
+__u64 apic_base
+union anno
kvm_vcpu_stat: x86
-u32 pf_fixed
-u32 pf_guest
-u32 tlb_flush
-u32 invlpg
-u32 exits
-u32 io_exits
-u32 mmio_exits
-u32 signal_exits
-u32 irq_window_exits
-u32 nmi_window_exits
-u32 halt_exits
-u32 halt_wakeup
-u32 request_irq_exits
-u32 request_nmi_exits
-u32 irq_exits;
-u32 host_state_reload
-u32 efer_reload
-u32 fpu_reload
-u32 insn_emulation
-u32 insn_emulation_fail
-u32 hypercalls
-u32 irq_injections
-u32 nmi_injections
«struct»kvm_vcpu_arch
preempt_ops
-void (*sched_in)(struct preempt_notifier *notifier, int cpu)
-void (*sched_out)(struct preempt_notifier *notifier,struct task_struct *next)

VCPU 结构体定义



kvm_vcpu_arch 结构



Kvm-vcpu-arch 包含数据结构