Kernel Virtual Machine

Shashank Rachamalla

Indian Institute of Technology
Dept. of Computer Science

November 24, 2011

Abstract

KVM (Kernel-based Virtual Machine) is a full virtualization solution for x86 hardware equipped with virtualization extensions (Intel VT or AMD-V). Using KVM, one can host multiple virtual machines running unmodified Linux or Windows images. These virtual machines have private virtualized hardware including a network card, disk, graphics adapter, etc. The purpose of this report is to clarify certain concepts required for better understanding of KVM code and thereby enable beginners to start contributing quickly. The focus will be on virtualization in the context of Intel processors (VTx)

Contents

1 KVM Kernel & Userspace

KVM is a Virtual Machine Manager (VMM) which fully exploits the Linux kernel for virtualization. To be precise, Linux kernel's process scheduling and memory management routines are used by KVM to manage the Guest software. On the other hand, KVM takes help from Qemu which runs in userspace for I/O emulation. Thus, Linux kernel and Qemu form the KVM Kernel and Userspace respectively.

1.1 KVM Module Initialization

KVM ships as a loadable kernel module, **kvm.ko**, that provides the core virtualization infrastructure and a processor specific module, **kvm-intel.ko** or **kvm-amd.ko** which together provide for all that is needed to get virtualized CPU support. The following figure captures the initialization phase of KVM.

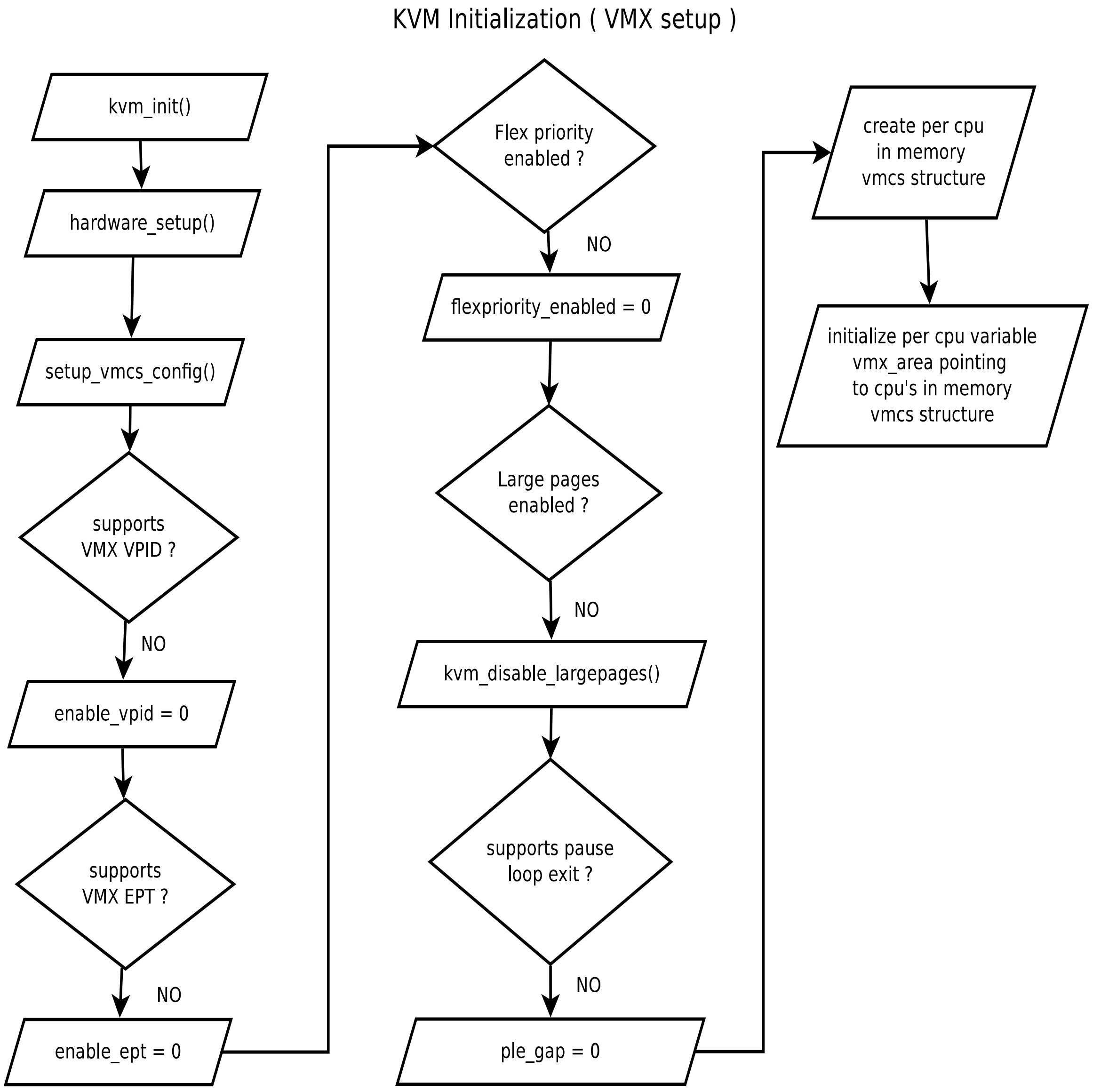


Figure 1: KVM Module Initialization

1.2 Intel Virtualization Extensions

The following is an explanation of different virtualization extensions being checked for during KVM initialization phase captured in above figure.

1. **vmx**: Intel VT-x, basic virtualization
2. **ept**: Extended Page Tables, an Intel feature to make emulation of guest page tables faster
3. **vpid**: Intel feature to make expensive TLB flushes unnecessary when context switching between guests
4. **tpr_shadow and flexpriority**: Intel feature that reduces calls into the hypervisor when accessing the Task Priority Register, which helps when running certain types of SMP guests
5. **vnmi**: Intel Virtual NMI feature which helps with certain sorts of interrupt events in guests

2 Intel VTx Processor Guest Mode

Intel processors with virtualization extensions (VTx) support an addtional execution mode called Guest mode along with the default privileged and unprivilged modes. Hence, at a given point in time the processor can be in any of these modes executing relevant instructions.

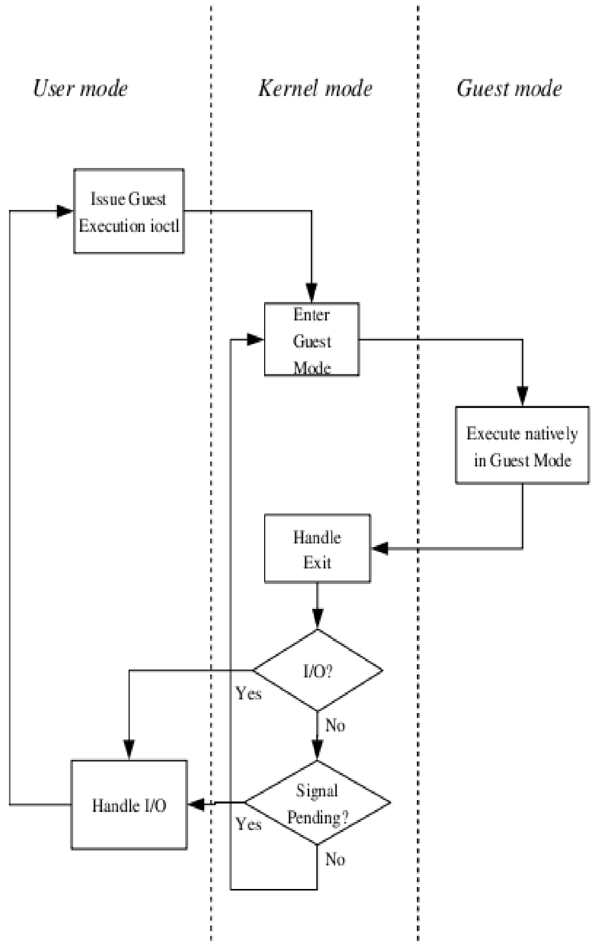


Figure 2: CPU Modes

3 KVM API

KVM API is the way to go for transitioning between user mode, kernel mode and guest mode. The API is defined as a set of ioctl commands to be issued to the KVM clone device **/dev/kvm** by opening the device as a file. The below code demonstrates the KVM_GET_API_VERSION call to fetch the version of KVM hypervisor.

```
#include<linux/kvm.h>
#include<stdio.h>
#include<linux/ioctl.h>
#include<fcntl.h>
int main() {

        // obtain fd of /dev/kvm
        int fd, version;
        fd = open("/dev/kvm", O_RDWR);
        if (fd == -1) {
                return -1;
        }
        version = ioctl(fd, KVM_GET_API_VERSION, 0);
        printf("Version:%d\n", version);
}
```

3.1 Classification

KVM API is broadly categorized into following classes.

1. **System ioctls**: These query and set global attributes which affect the whole kvm subsystem. A system ioctl is used to create virtual machines.
2. **VM ioctls**: These query and set attributes that affect an entire virtual machine, for example memory layout. A VM ioctl is used to create virtual cpus (vcpus).
3. **VCPU ioctls**: These query and set attributes that control the operation of a single virtual cpu.

4 Virtual Machine Entries & Exits

Virtual Machine Entry is defined as the transition from kernel mode to guest mode and Virtual Machine Exit is defined as the transition from guest mode to kernel mode. The KVM API call KVM_RUN when executed in user mode will cause a transition into kernel mode followed by a transition into guest mode. Also, it is very important to understand what causes a Virtual Machine Exit. In case of Intel processors with support for VTx, a data structure called VMCS (Virtual Machine Control Structure) defines the set of exit conditions. The important task of actually running the guest code is done by invoking the KVM_RUN API

5 VMCS(Virtual Machine Control Structure)

A virtual CPU uses VMCS while in guest mode. This data structure manages VM entries and exits. KVM uses a separate VMCS for each virtual CPU. Content in VMCS is organized into the following groups

1. **Guest-state area**: Processor state is saved into the guest-state area on VM exits and loaded from there on VM entries.
2. **Host-state area**: Processor state is loaded from the host-state area on VM exits.
3. **VM-execution control fields**: These fields control processor behavior in guest mode. They determine in part the causes of VM exits.
4. **VM-exit control fields**: These fields control VM exits.
5. **VM-entry control fields**: These fields control VM entries.
6. **VM-exit information fields**: These fields receive information on VM exits and describe the cause and the nature of VM exits. They are read-only.

5.1 VM Exit Scenarios

There can be several cases where VM Exits will be required. There are certain instructions which can cause VM Exits. Examples include CPUID, INVD, INS, OUTS etc. Other cases may include Exceptions, External Interrupts, Task Switches etc. VMCS specifies Exception and I/O bitmaps as part of VM-exit control fields. These bitmaps decide when an exception or I/O instruction causes an exit.

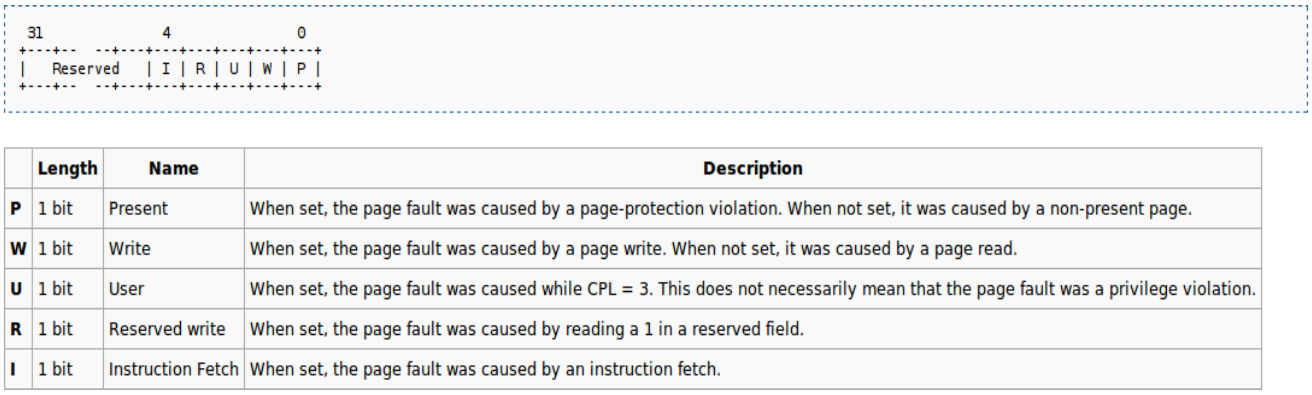
5.2 Exit handlers in KVM code

Below is a list of exit handlers defined in KVM code. Each exit handler is a function pointer.

```
/*
 * The exit handlers return 1 if the exit was handled fully and guest execution
 * may resume.  Otherwise they set the kvm_run parameter to indicate what needs
 * to be done to userspace and return 0.
 */
static int (*kvm_vmx_exit_handlers[])(struct kvm_vcpu *vcpu) = {
        [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_VMCLEAR]                 = handle_vmx_insn,
        [EXIT_REASON_VMLAUNCH]                = handle_vmx_insn,
        [EXIT_REASON_VMPTRLD]                 = handle_vmx_insn,
        [EXIT_REASON_VMPTRST]                 = handle_vmx_insn,
        [EXIT_REASON_VMREAD]                  = handle_vmx_insn,
        [EXIT_REASON_VMRESUME]                = handle_vmx_insn,
        [EXIT_REASON_VMWRITE]                 = handle_vmx_insn,
        [EXIT_REASON_VMOFF]                   = handle_vmx_insn,
        [EXIT_REASON_VMON]                    = handle_vmx_insn,
        [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_MCE_DURING_VMENTRY]      = handle_machine_check,
        [EXIT_REASON_EPT_VIOLATION]           = handle_ept_violation,
        [EXIT_REASON_EPT_MISCONFIG]           = handle_ept_misconfig,
        [EXIT_REASON_PAUSE_INSTRUCTION]       = handle_pause,
        [EXIT_REASON_MWAIT_INSTRUCTION]       = handle_invalid_op,
        [EXIT_REASON_MONITOR_INSTRUCTION]     = handle_invalid_op,
};
```

6 Handling Page Faults

When a page fault occurs, virtual CPU checks the bit 14 of the exception bitmap present in VMCS. A VM Exit happens if the bit is set. Also, PFEC (Page Fault Error Code) and the value of Guest Virtual Address (GVA) are written to VM-exit information fields for use by KVM.



```
 31              4              0
+---+--  --+---+---+---+---+---+
|   Reserved   | I | R | U | W | P |
+---+--  --+---+---+---+---+---+
```

	Length	Name	Description
P	1 bit	Present	When set, the page fault was caused by a page-protection violation. When not set, it was caused by a non-present page.
W	1 bit	Write	When set, the page fault was caused by a page write. When not set, it was caused by a page read.
U	1 bit	User	When set, the page fault was caused while CPL = 3. This does not necessarily mean that the page fault was a privilege violation.
R	1 bit	Reserved write	When set, the page fault was caused by reading a 1 in a reserved field.
I	1 bit	Instruction Fetch	When set, the page fault was caused by an instruction fetch.

Figure 3: PFEC

6.1 Terminology

1. GFN: Guest Frame Number
2. PFN: Host Page Frame Number
3. GPA: Guest Physical Address
4. HVA: Host Virtual Address
5. HPA: Host Physical Address
6. EPT: Extended Page Tables (Intel)
7. NPT: Nested Page Tables (AMD)
8. TDP: Two Dimensional Paging (A common term to EPT and NPT)
9. TSS - Task State Segment
10. TPR - Task priority register
11. MMIO - Memory mapped I/O
12. PIT - Programmable Interrupt Timer
13. IMP - Identity Map Page
14. KVM_CAP - KVM Capability

6.2 Soft MMU vs TDP MMU

Next generation Intel and AMD processors have support for Extended page tables (EPT) and Nested page tables (NPT) respectively. KVM code internally switches to TDP (Two dimensional paging) when it becomes aware of EPT/NPT. With EPT/NPT, page walking (translation) is done by processor instead of trapping into KVM. However, KVM uses Soft MMU (code based page walk) for processors without any support for EPT/NPT. The following figure shows the work flow involved in Virtual CPU creation.

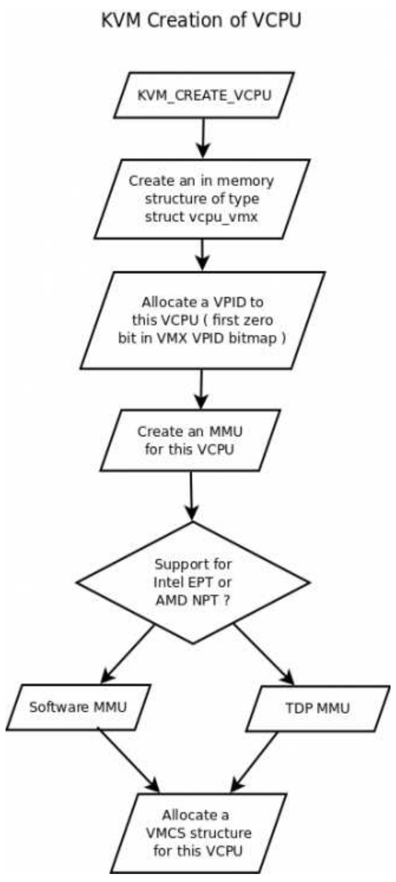
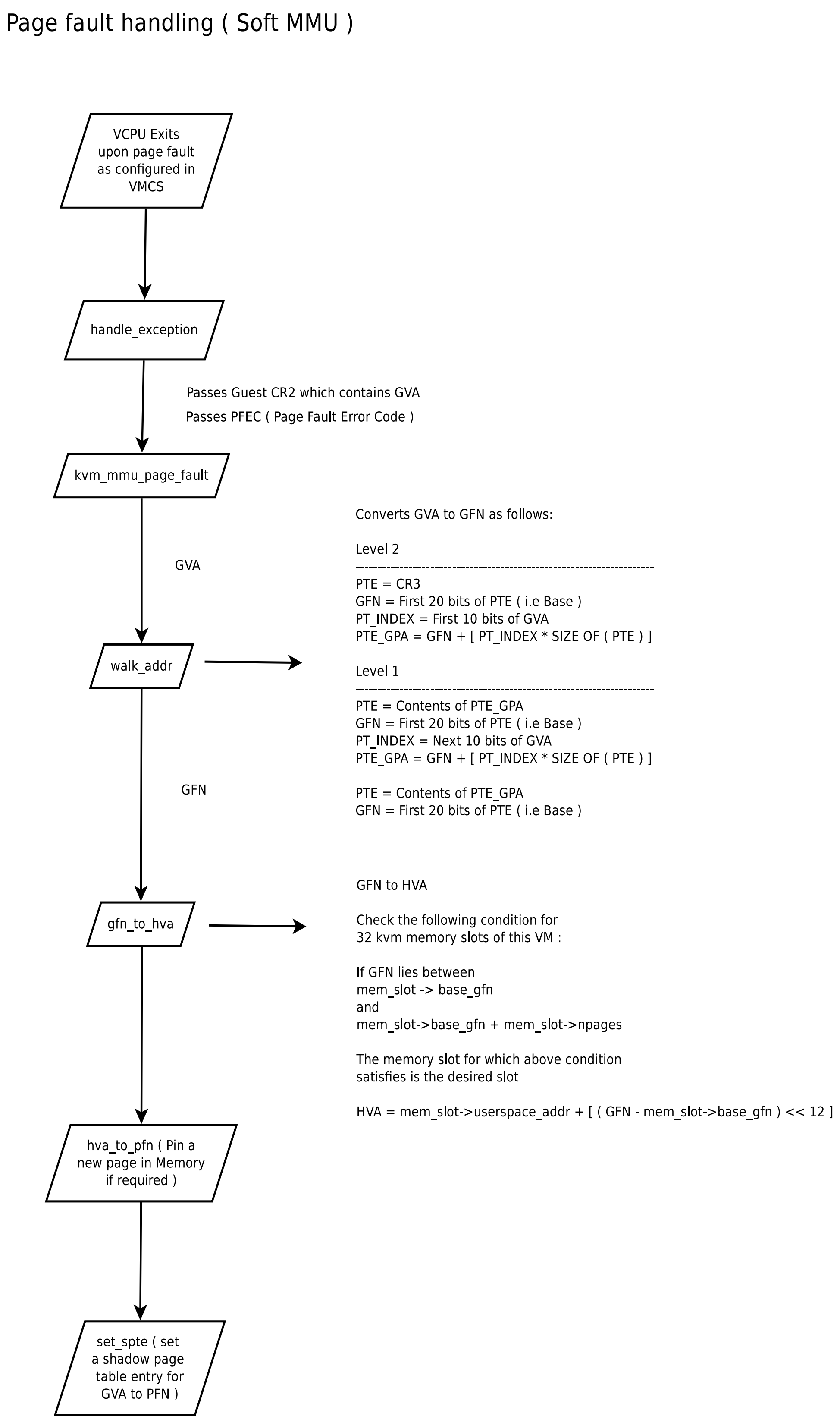

KVM Creation of VCPU
KVM_CREATE_VCPU
Create an in memory structure of type struct vcpu_vmx
Allocate a VPID to this VCPU (first zero bit in VMX VPID bitmap)
Create an MMU for this VCPU
Support for Intel EPT or AMD NPT ?
Software MMU
TDP MMU
Allocate a VMCS structure for this VCPU

Figure 4: VCPU Creation

6.3 Shadow Page Tables

KVM maintains a copy of Shadow Page Tables for each Virtual Machine it runs. Virtual CPUs refer to these page tables instead of Guest's page tables for all translations. In the above figure, if the Present bit is not set, the page fault is due to a missing page. KVM will bring a new physical page to memory and update the corresponding shadow page table entry before resuming control to Guest.

The below figure shows a translation from GVA to PFN when a Soft MMU is used. GVA is contained in the virtual CR2 register of the Virtual Machine and this register is available when Virtual Machine exits. The base address of Virtual Machine's page directory can be obtained from virtual CR3 register. With the base address of page directory and GVA, a two level page walk is performed to identify the corresponding GFN. The GFN is then converted to HVA by looking at the available 32 slots of memory allocated to this Virtual Machine. Finally, a page is loaded into the physical memory to backup this HVA.


Page fault handling (Soft MMU)
VCPU Exits upon page fault as configured in VMCS
handle_exception
kvm_mmu_page_fault
GVA
walk_addr
GFN
gfn_to_hva
hva_to_pfn (Pin a new page in Memory if required)
set_spte (set a shadow page table entry for GVA to PFN)

Passes Guest CR2 which contains GVA

Passes PFEC (Page Fault Error Code)

Converts GVA to GFN as follows:

Level 2

PTE = CR3
GFN = First 20 bits of PTE (i.e Base)
PT_INDEX = First 10 bits of GVA
PTE_GPA = GFN + [PT_INDEX * SIZE OF (PTE)]

Level 1

PTE = Contents of PTE_GPA
GFN = First 20 bits of PTE (i.e Base)
PT_INDEX = Next 10 bits of GVA
PTE_GPA = GFN + [PT_INDEX * SIZE OF (PTE)]

PTE = Contents of PTE_GPA
GFN = First 20 bits of PTE (i.e Base)

GFN to HVA

Check the following condition for
32 kvm memory slots of this VM :

If GFN lies between
mem_slot -> base_gfn
and
mem_slot->base_gfn + mem_slot->npages

The memory slot for which above condition
satisfies is the desired slot

HVA = mem_slot->userspace_addr + [(GFN - mem_slot->base_gfn) << 12]

Figure 5: GVA to PFN

7 Exercises

1. Identify the piece of code in **KVM Kernel space** where KVM exits to User space for emulating an I/O Instruction. Trace events of interest such as

   Time when I/O devices like keyboard and hard disk get accessed

   Frequency of access to a particular I/O device

2. Identify the piece of code in **KVM User space** (qemu-kvm) and trace events of interest such as

   Time taken to serve an I/O request

3. Watch out for interesting KVM related events in **debugfs** and identify trends.

4. Put the **KVM API** to use. KVM_GET_VCPU_EVENTS is a good place to start. Also, work out the flow involved when KVM_SET_USER_MEMORY_REGION is invoked.

8 Future Work

1. Page walk semantics with Intel EPT
2. Page fault during Dirty Bit updation
3. Memory Ballooning

9 References

1. Kernel documentation (kvm/api.txt)
2. Intel Software Developers Manual (Volume 3a - 3b)
3. KVM MMU (Avi Kivity - LWN.net)