Qemu and KVM Internals

QEMU = Quick EMUlator

Source

KVM (Kernel Virtual Machine) is a Linux kernel module that allows a user space program << like QEMU >> to utilize the hardware virtualization features of various processors. Today, it supports recent Intel and AMD processors (x86 and x86_64), PPC 440, PPC 970, S/390, ARM (Cortex A15, AArch64), and MIPS32 processors.

QEMU can make use of KVM when running a target architecture that is the same as the host architecture. For instance, when running *qemu-system-x86* on an x86 compatible processor, you can take advantage of the KVM acceleration - giving you benefit for your host and your guest system.

Put another way: KVM is a virtualization feature in the Linux kernel that lets a program like qemu safely execute guest code directly on the host CPU. This is only possible when the target architecture is supported by the host CPU; today that means x86-on-x86 virtualization only.

The reality is that QEMU is a standalone full virtualization software application, capable of emulating a hardware platform in it's entirety *without* the requirement of a host OS-level hypervisor (pure emulation mode).

If a host hypervisor (like kvm) is present, and the guest and host are the same architecture (cpu family), then QEMU has the capability to take advantage of the hypervisor to (greatly!) accelerate performance.

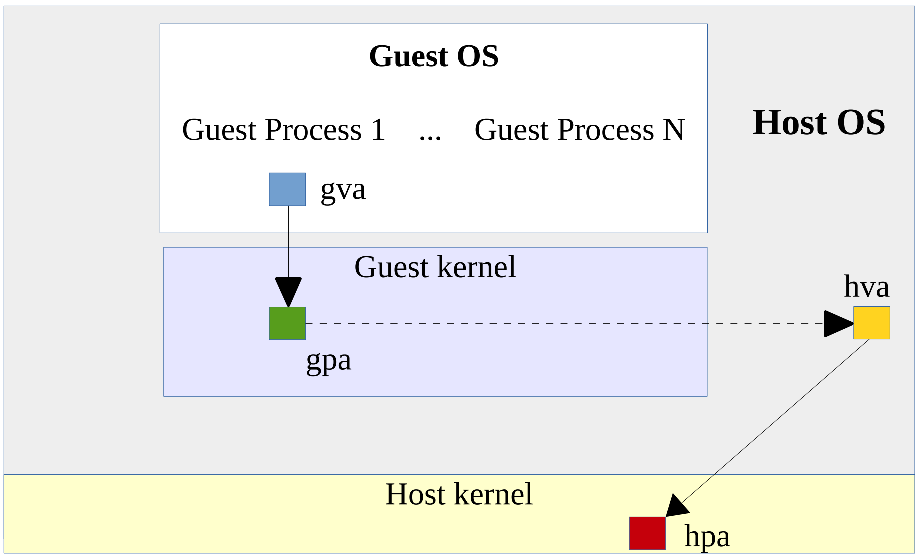
Full virtualization: Qemu, VMware, VirtualBox, Xen, MS Hyper-V, etc
Paravirtualization: Qemu, Xen, UML (User-Mode Linux), etc

Resources

KVM Internals brief: Kernel Virtual Machine (KVM), Shashank Rachamalla

Virtualization and Memory

- On a regular host system, all modern OS's use virtual memory (*vm*)
- Essentially, *vm* sets up an *illusion* for each process: that it has all potentially available memory available to it (so, on a 32-bit cpu, each process is fooled into thinking that it has 2^{32} = 4 GB memory available)
- This illusion is achieved by *mapping* process virtual addresses to host (DRAM) physical addresses and/or the swap partition (that's more than a bit simplistic; but now's not the time or place to delve into the black magic of virtual memory!)
- So every time a process accesses memory (for reading or writing, which is usually pretty much almost all the time), we have to *translate* the virtual address to a physical address
- Doing this in OS software is possible, but far too slow
- Hence, we use hardware to do it at runtime – the MMU (Memory Management Unit) on the CPU. (Not only that, for further efficiency, all modern CPUs set aside some high-speed cpu memory as a *cache* for address translations; this is called the TLB (Translation Lookaside Buffer))
- *vm* work is split between software and hardware
  - The OS's (software) job is to setup and maintain *paging tables* on a per process basis
  - The MMU's (and TLB's, hardware) job is to perform runtime address translation by looking up the TLB and/or paging tables
  - this makes the whole scheme feasible in practice
- Now, think about it: what if we insert a layer between the process and the OS – a Virtual Machine (VM), along with a hypervisor to service it! It failry boggles the mind, as now:
  - the entire VM is nothing but a usermode host process
    - but internally, it's an entire OS itself – the guest OS, hosting guest processes
      - so, each guest process uses virtual addresses – these we call *gva*'s (guest virtual addresses),
        - upon access, they must be translated to guest physical addresses (*gpa*'s),
          - which in turn are nothing but host virtual addresses (*hva*'s),
            - ➢ which in turn must be translated to host physical addresses (*hpa*'s)!



From https://www.kernel.org/doc/Documentation/virtual/kvm/mmu.txt

Acronyms
========

pfn	host page frame number
hpa	host physical address
hva	host virtual address
gfn	guest frame number
gpa	guest physical address
gva	guest virtual address
ngpa	nested guest physical address
ngva	nested guest virtual address
pte	page table entry (used also to refer generically to paging structure entries)
gpte	guest pte (referring to gfns)
spte	shadow pte (referring to pfns)
tdp	two dimensional paging (vendor neutral term for NPT and EPT)

VMware Terminology:
LPN : logical page number; equivalent to 'gfn' (and therefore 'gva') above
PPN : physical page number; equivalent to 'gpa' (and therefore 'hva') above
MPN : machine page number; equivalent to 'pfn' (and therefore 'hpa') above

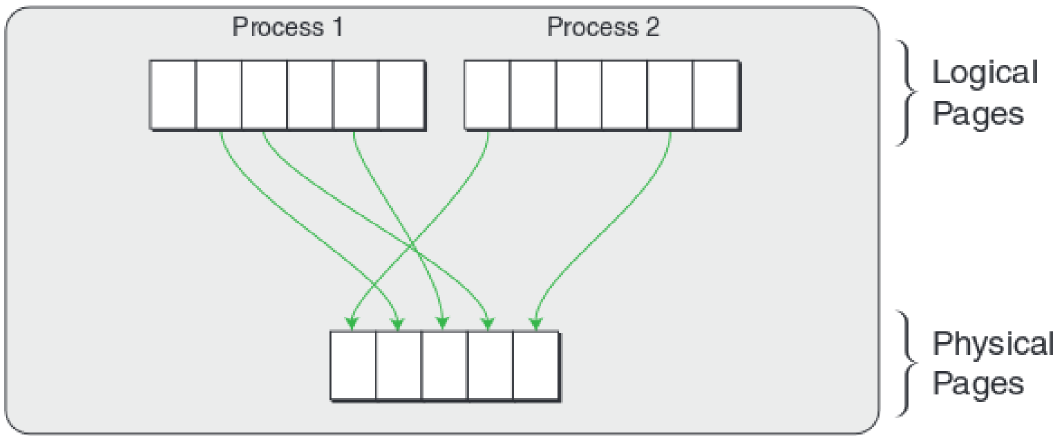
Broad Approaches to Address Translation on a Virtual Machine

- Naive approach
- Shadow Page Tables (SPT)
- Hardware Assist solutions (Intel EPT / AMD NPT)

Naive Approach

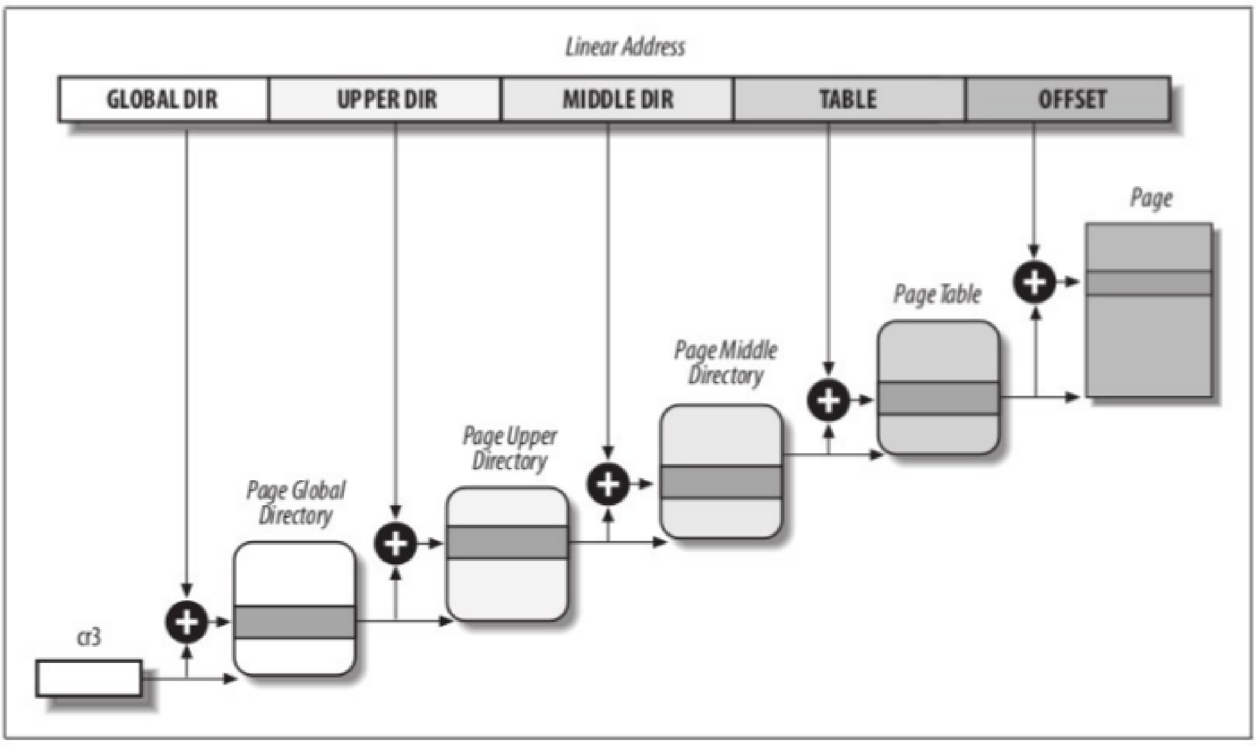
- On the host, every virtual address (va) accessed by a process goes through the TLB, and if we have a TLB miss, through the MMU which performs an n-level (n=4 on Linux) page table "walk" to translate the va to a physical address (pa)

Figure 1. Native System Memory Management Unit Diagram



Source

- On a Virtual Machine, for every guest virtual address (gva) accessed by a guest process, the gva goes to the *guest paging tables* and the page table walk has to be undertaken [gva -> hva]
- Lets assume a 64-bit guest OS running on a 64-bit host
- x86_64 uses 4-level paging (so does the arch-independent Linux OS code); thus each va to pa translation involves 4 memory lookups (in case of a TLB miss) [see detailed diagrams below]



4-Level Paging
Source: Understanding the Linux Kernel, Bovet & Cesati, O'Reilly

```
                                va -> pa
                                   |
                                   v
                             Assume TLB miss
                                   |
                                   v
                             Page Table Walk
                         [--48-bit virtual address--]
         (CR3 base address) PGD -> PUD -> PMD -> PTE -> pf (page frame)
                                   ML     ML     ML     ML
ML = Memory Lookup
```

Address Translation via page table walking on an x86_64 requires **4 DRAM lookups**.

- So, for each gva looked up, it will take at least 4 memory lookups to translate the gva to a hva; IOW, *each* of these memory lookups (denoted by the "ML" in the illustration above) is actually a host virtual address (hva) which must subsequently be translated to a host physical address (hpa) [hva -> hpa]

- Translating hva -> hpa again requires 4 DRAM lookups *per* Virtual Memory lookup in the guest paging tables
- Thus, we require 4x4 = 16 memory (DRAM) lookups to translate a single gva to hpa!

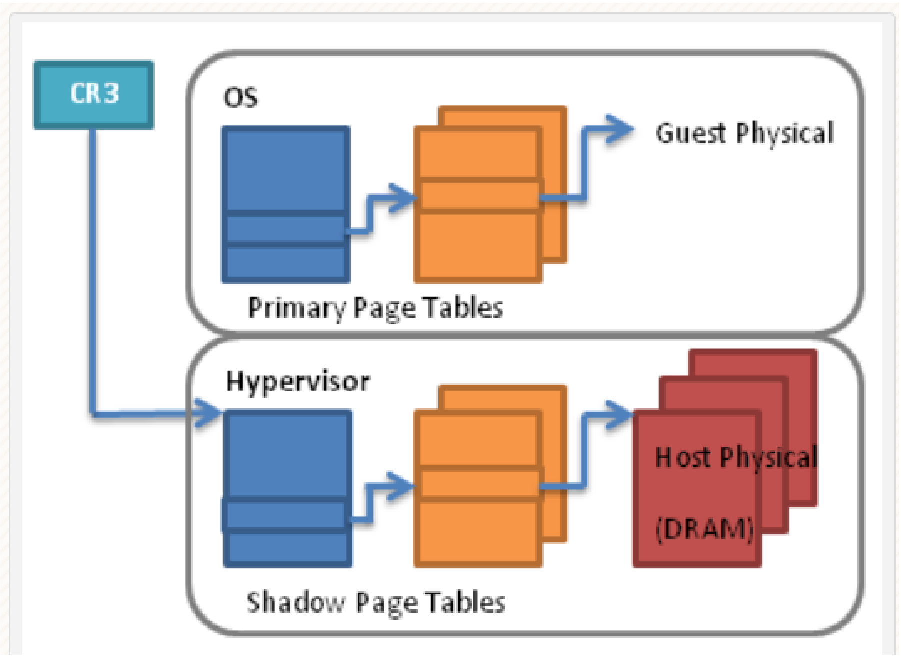
Host kernel

- Conceptually straightforward, but extremely high overhead!

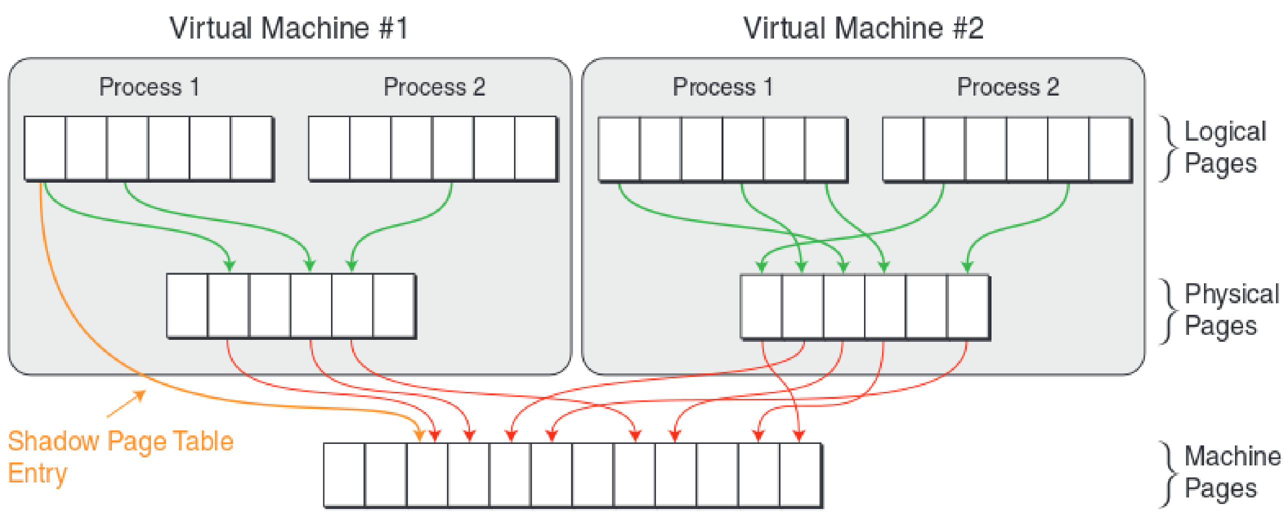
Shadow Page Tables or Softmmu

Source

In the absence of nested (hw assist, seen next) page table support, there is another cheat that's used by virtual machines. The idea is to create a second copy of page tables called shadow page tables that map (guest) virtual addresses directly to host physical addresses (actual DRAM), and let the processor use them for address translation (so the CR3 register points to those, rather than to primary (guest) page tables).



The hypervisor keeps separate shadow page tables for each process. When a context switch occurs, the OS tries to modify the CR3 register to point to the primary page directory for the new process. CR3 modification is a protected instruction and it traps into the hypervisor and gives it the opportunity to switch the shadow tables and point the CR3 at them. The hypervisor might also chose to discard shadow page tables upon context switch, and later refill them on demand using hidden page faults.



[Above pic Source]

<<
Shadow paging tables : the hypervisor maintains one shadow paging table per guest process. Shadow paging is sometimes also referred to as "software MMU" or "softmmu" mode.
>>

Shadow Paging - Details

The structure of (guest) shadow page tables that are kept by the hypervisor reflects the structure of primary page tables that are kept by the operating system. There are several approaches to keeping those two structures in sync.

<<
Q> What operation(s) could change the guest paging tables?

A> Why, several! Here's some:

- any kind of page fault (includes several of the cases below)
- fork() (COW-setup, etc)
- exit() (teardown)
- accessing a guest virtual page (malloc'ed) that has no backing physical page (in effect, a minor page fault)
- swapping out a guest page
- paging in a guest page (major page fault)
- kernel operations / optimisations:
  - THP (merging 4 KB-to-2 MB or vice-versa (splitting))
  - KSM (kernel samepage merging)
  - Compaction
  - Buddy-system page migration code
  - etc

Every single change in the guest pte's has to be synchronized with the host paging tables!
>>

The naive approach would be to let the OS freely modify its page tables, e.g., in response to a page fault. The hypervisor would only intercept the privileged instruction INVLPG that the OS uses to invalidate a TLB entry. At that point the hypervisor would make a corresponding change to its shadow page tables, substituting the guest physical address with the corresponding host physical address (possibly allocating a physical page from its own pool). Unfortunately, this doesn't work because some operating systems (I won't be pointing fingers) occasionally forget to invalidate TLB entries.

The fallback approach is for the hypervisor to write-protect all guest page tables. Whenever the (guest) OS tries to modify them, a page fault occurs — the so called *tracing fault* — which is vectored into the hypervisor. That gives the hypervisor the opportunity to immediately update its shadow page tables. In this scheme the two structures always mirror each other (modulo physical addresses) but at the cost of a large number of page faults.

A common optimization is to allow shadow page tables to be lazily updated. << *IOW don't immediately update shadow page tables when the host pte's change; do it later when the host faults upon trying to access a non-existant guest pte* >>
This is possible if not all primary page tables are write protected. If an entry is present in the primary page table but is missing from the shadow page table, a page fault will occur when the hardware walks it for the first time. This is a so called hidden page fault and it's serviced by the hypervisor, which at that point creates an entry in its shadow page tables based on the entry in the primary page tables.

Since the hypervisor must be involved in the processing of some of the page faults, it must install its own page handler. That makes all faults, real and hidden, vector into the hypervisor. A real page fault happens when the entry is missing (or is otherwise protected) in the primary (host) page tables. Such faults are reflected back to the (host) OS — it knows what to do with them. Faults that are caused by the (host) OS manipulating page tables, or by the absence of an entry in the shadow page tables are serviced by the hypervisor.

An additional complication is related to the A/D (accessed and dirty) bits stored in page tables. Normally, whenever a page of memory is accessed, the processor sets the A bit in the corresponding PTE (Page Table Entry); and when the page is written, it also sets the D bit. These bits are used by the OS in its page replacement policy. For instance, if a page hasn't been dirtied, it doesn't have to be written back to disk when it's unmapped. In a virtualized system, the processor will of course mark the shadow PTEs << as the *virtualized* CR3 register will point to the guest base paging table for that process context! >>, and those bits have to somehow propagate to the primary page tables for the use by the OS.

This can be accomplished by always starting host physical pages in the read-only mode. The hypervisor allocates a host physical page but it sets the read-only bit in its shadow PTE, which is used by the processor for address translation. Reading from this page proceeds normally, but as soon as the processor tries to write to it, there is a protection fault. The hypervisor intercepts this fault: It checks the guest page tables to make sure the page was supposed to be writable; if so, it sets the dirty bit in the guest PTE, and removes the write protection from the host PTE. After that, write access to this page may proceed without further faults.
...

Additional Ref: *Address Translation for Virtual Machines*

Hardware Assist [Nested Paging Tables]

Source: Performance Evaluation of Intel EPT Hardware Assist, VMware, Mar 2009

... In some of their recent x86 processors AMD and Intel have begun to provide hardware extensions to help bridge this performance gap. In 2006, both vendors introduced their first-generation hardware support for x86 virtualization with AMD-Virtualization ™ (AMD-V ™) and Intel ® VT-x technologies. Recently Intel introduced its second generation of hardware support that incorporates MMU virtualization, called Extended Page Tables (EPT).

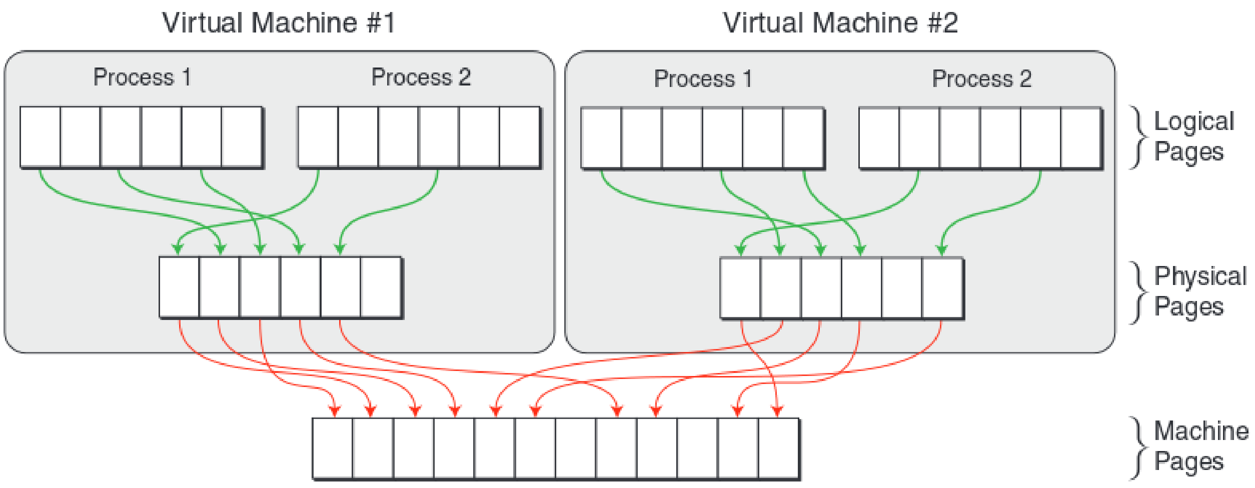
<<
The solution to serious performance issues with shadow paging is Hardware Support: Intel's EPT and AMD's NPT:

Eg. on an Intel host:
```
# lsmod |grep kvm
kvm_intel             151552  0
kvm                   483328  1 kvm_intel
#
```

kvm_intel is the kernel module providing Intel EPT (hardware-layer) support for nested paging; it uses (functions/data) of the kvm kernel module. (On an AMD machine, the kernel module would be called kvm_amd).
>>

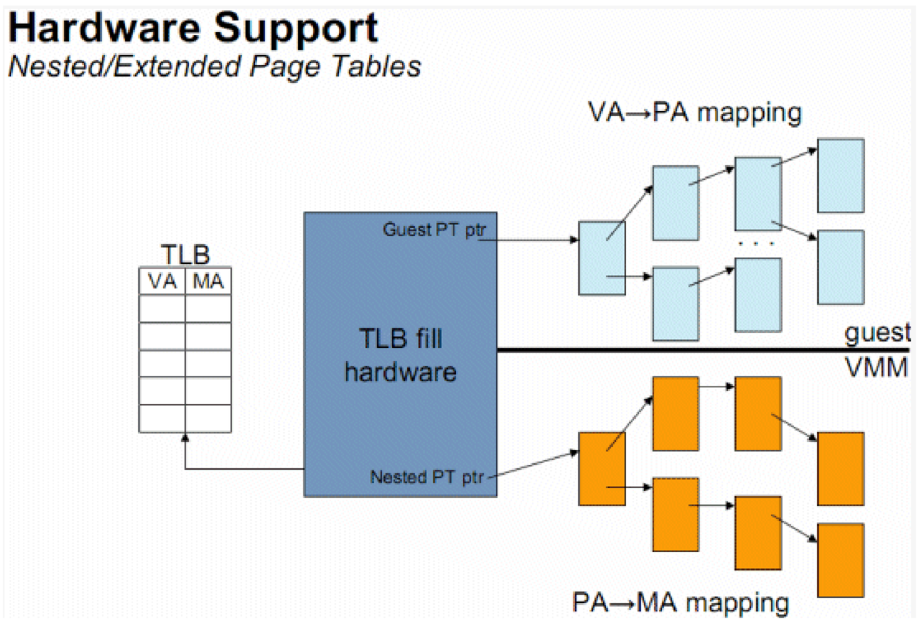
In this case both the guest page tables and the nested page tables are exposed to the hardware.

Figure 3. Hardware Memory Management Unit Diagram



Source

When a logical address is accessed, the hardware (MMU) walks the guest page tables as in the case of native execution, but for every PPN << gpa == hva >> accessed during the guest page table walk, the hardware also walks the nested page tables to determine the corresponding MPN (hpa), thus eliminating "shadow" paging << *seen earlier:* usage of a soft-mmu, and the frequent update of the shadow PTEs >>, instead using only the *hardware* MMU for both translations!



<<

From above diagram:

- *the TLB attempts to cache* all *translations: VA -> MA, i.e., both*
  - *gva -> hpa, and*
  - *hva -> hpa*
- *VA -> PA :: gva -> hva*

- *PA -> MA :: hva -> hpa*
- *Key point: with EPT/NPT, the hardware TLB(s) on the host core(s) now cache **both** guest-to-host and host-to-machine translations, making address translation efficient. All without the need for (performance-draining) shadow paging.*

>>

This composite translation eliminates the need to maintain shadow page tables and synchronize them with the guest page tables.

However the extra operation also increases the cost of a page walk, thereby impacting the performance of applications that stress the TLB. This cost can be reduced by using large pages *<< recall THP (Transparent Huge Pages) and KSM (Kernel Samepage Merging) - Linux OS MM technologies >>*, thus reducing the stress on the TLB for applications with good spatial locality.

For optimal performance the ESX VMM and VMkernel aggressively try to use large pages for their own memory when EPT is used.

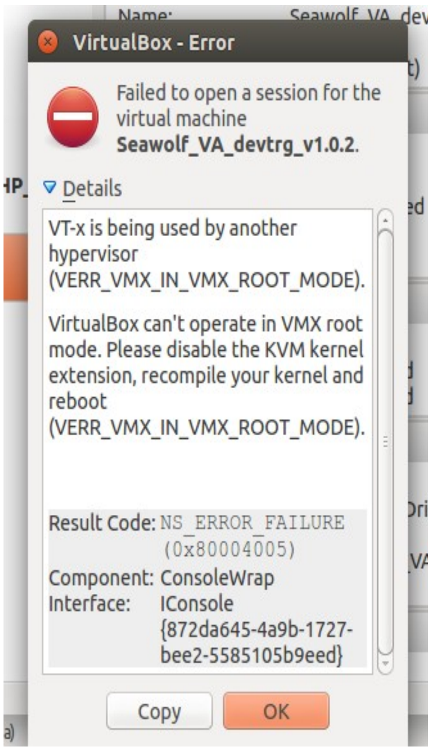
<< Also see SIDEBAR below on Performance Evaluation, VMware >>.

<<
Note!

Only ONE hypervisor can make use of the processor hardware virtualization extensions (VT-x on Intel) at a time.

See the error thrown up by VirtualBox when attempting to launch a guest when another QEMU guest is already executing via the hypervisor (and thus making use of VT-x on a system that has the feature enabled):
"VT-x is being used by another hypervisor ..."

>>



Failed to open a session for the virtual machine **Seawolf_VA_devtrg_v1.0.2.**

VT-x is being used by another hypervisor (VERR_VMX_IN_VMX_ROOT_MODE).

VirtualBox can't operate in VMX root mode. Please disable the KVM kernel extension, recompile your kernel and reboot (VERR_VMX_IN_VMX_ROOT_MODE).

Result Code: NS_ERROR_FAILURE (0x80004005)
Component: ConsoleWrap
Interface: IConsole {872da645-4a9b-1727-bee2-5585105b9eed}

Source: Virtual Machines: Virtualizing Virtual Memory
...

Address translation on the x86 is implemented in hardware, with page tables stored in regular memory that is directly accessible to the OS kernel. It's not easy to hook an additional level of indirection into this machinery. In recent years Intel and AMD started producing chips that provide hardware support for the second layer of virtualization. AMD calls its technology nested page tables (NPT), and Intel calls theirs extended page tables (EPT). Unfortunately those two differ in some important details — e.g., EPT doesn't support accessed/dirty bits.

The usual page table walk, which happens when the mapping is not present in the TLB, goes through the guest page tables and continues into host page tables, until the physical address is found. This address then goes into the TLB to speed up future lookups. This happens in hardware without any intervention from the hypervisor.

The hypervisor has to intervene only when a page fault happens. It's a two stage process: First the (guest) operating system processes the fault. It allocates guest physical memory (*gpa* == *hva*) from its (fake) pool and stores it in the guest page table. Remember, the (guest) OS knows nothing about the second level of indirection — it's just your regular Linux or Windows.

<<
This hardware-support paging feature is called "tdp" by KVM – "two-dimensional paging". The two-dimensional part comes in because the *page tables of the guest must *themselves* go through the page tables of the host*.
See Appendix B : Nested Paging – Overheads.
>>

When the user instruction that caused the fault is restarted, address translation progresses half way, and faults again when it can't find the guest-to-host mapping. This time the fault is vectored into the hypervisor, which allocates actual physical memory and patches the rest of the page tables.

Old generations chips don't support NTP so a different set of software tricks is used. ...

<<

If direct hardware-assisted page table translation guest-virtual to host-physical (nested paging / EPT / NPT) is *not* available on the CPU, hypervisors fallback to a software-only translation mode (called "softmmu" or shadow paging).

Linux KVM

Linux kvm by default assumes that we will not have the hardware nested paging support available (Intel EPT or AMD NPT).

Upon initialisation, kvm will check whether hardware mmu support is available; if so, it will switch to using it, if not, it will use Shadow Paging (the "soft-mmu" mode, default, which is of course more expensive under several (MMU-intensive) workloads in performance terms).

arch/x86/kvm/mmu.c

```
...
/*
 * When setting this variable to true it enables Two-Dimensional-Paging
 * where the hardware walks 2 page tables:
 * 1. the guest-virtual to guest-physical
 * 2. while doing 1. it walks guest-physical to host-physical
 * If the hardware supports that we don't need to do shadow paging.
 */
bool tdp_enabled = false;
```

However:

```
...
     else if (tdp_enabled)
         init_kvm_tdp_mmu(vcpu);
     else
         init_kvm_softmmu(vcpu);  << if tdp_enabled is false (the default) ;
                                     softmmu => Shadow Paging >>
...
```

Where is TDP enabled?

```
...
void kvm_enable_tdp(void)
{
    tdp_enabled = true;
}
EXPORT_SYMBOL_GPL(kvm_enable_tdp);

void kvm_disable_tdp(void)
{
    tdp_enabled = false;
}
EXPORT_SYMBOL_GPL(kvm_disable_tdp);
...
```

```
arch/x86/kvm/vmx.c    << for Intel EPT >>
...
 if (enable_ept) {
        kvm_mmu_set_mask_ptes(0ull,
            (enable_ept_ad_bits) ? VMX_EPT_ACCESS_BIT : 0ull,
            (enable_ept_ad_bits) ? VMX_EPT_DIRTY_BIT : 0ull,
            0ull, VMX_EPT_EXECUTABLE_MASK);
        ept_set_mmio_spte_mask();
        kvm_enable_tdp();
    } else
        kvm_disable_tdp();
...
```

and for AMD:

```
arch/x86/kvm/svm.c
...
   if (npt_enabled) {     << for AMD-Virtualization >>
        printk(KERN_INFO "kvm: Nested Paging enabled\n");
        kvm_enable_tdp();
    } else
        kvm_disable_tdp();
...
```

>>

An excellent resource:
“System Virtualization: Memory Virtualization” [PPTX]

SIDEBAR : Virtualization Extensions on the ARMv-7

Src: ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition

"The ARMv7-A memory system incorporates a Memory Management Unit (MMU), controlled by CP15 registers. The memory system supports virtual addressing, with the MMU performing virtual to physical address translation, in hardware, as part of program execution.

An ARMv7-A processor that implements the Virtualization Extensions provides two stages of address translation for processes running at the Application level:
• The operating system defines the mappings from virtual addresses to intermediate physical addresses (IPAs). When it does this, it believes it is mapping virtual addresses to physical addresses.
• The hypervisor defines the mappings from IPAs to physical addresses. These translations are invisible to the operating system.
For more information see About address translation on page B3-1311.
..."

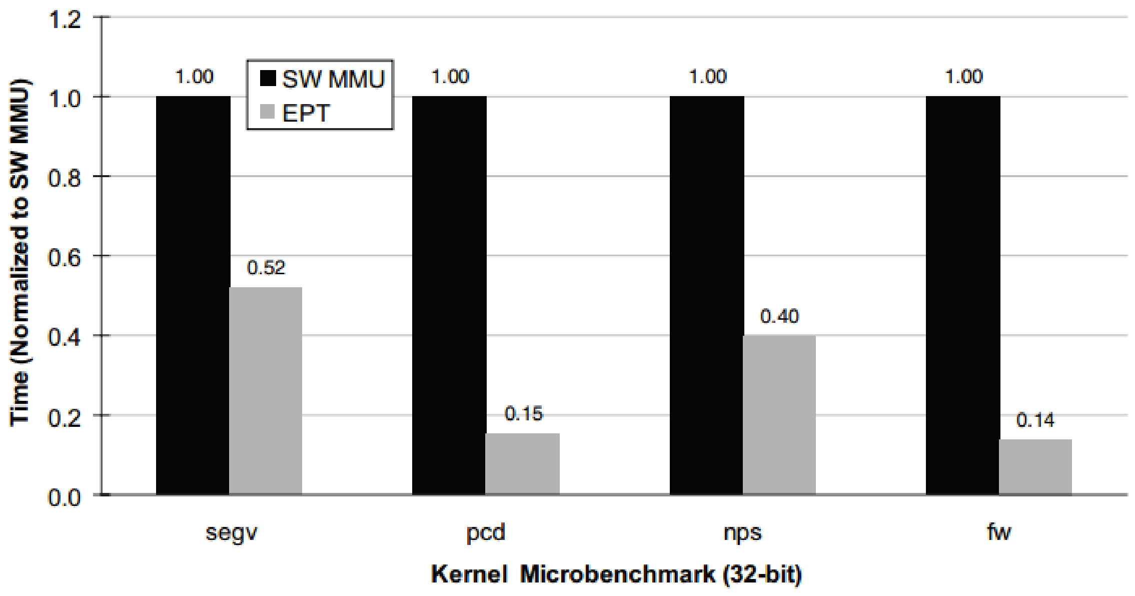
Performance

An interesting point: hardware-assisted nested paging techniques (NPT/EPT) do not always yield superior performance!

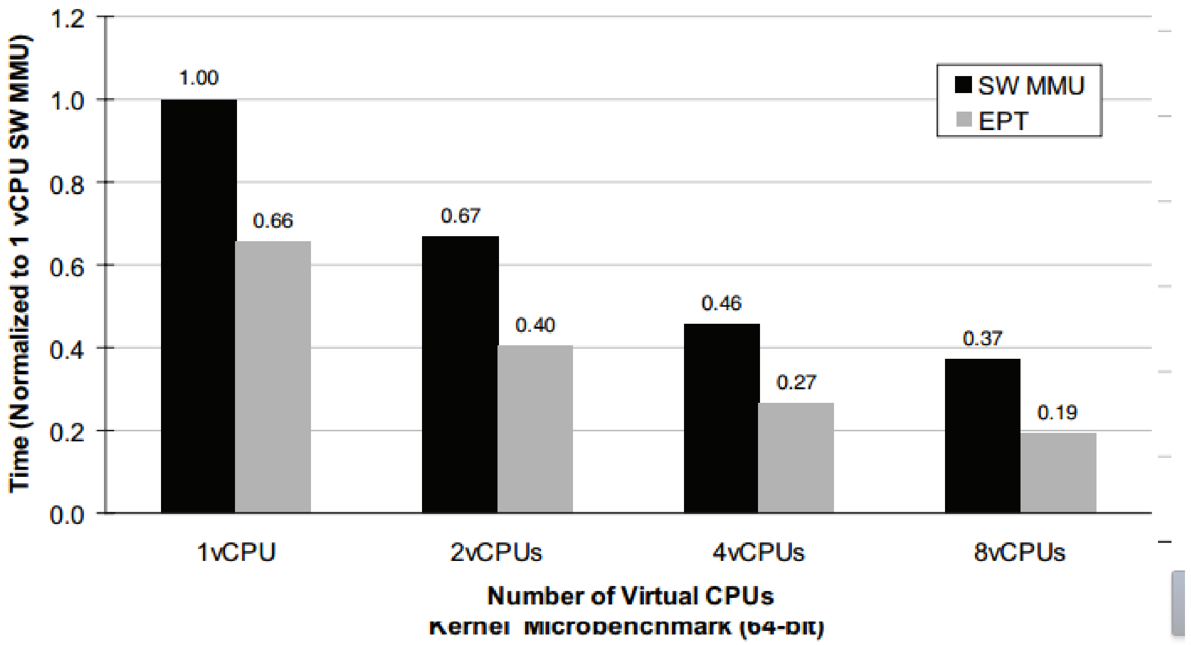
See VMware's study here: ***Performance Evaluation of Intel EPT Hardware Assist, VMware, Mar 2009***

Excerpts:
...

Figure 4. 32-bit MMU-Intensive Kernel Microbenchmark Results (Lower is Better)



F **Figure 6.** 32-bit Apache Compile Time (Lower is Better)



…

SPECjbb2005 is an industry-standard server-side Java benchmark. It has little MMU activity but exhibits high TLB miss activity due to Java's usage of the heap and associated garbage collection. Modern x86 operating system vendors provide large page support to enhance the performance of such TLB-intensive workloads. Because EPT further increases the TLB miss latency (due to additional paging levels), large page usage in the guest operating system is imperative for high performance of such applications in an EPT-enabled virtual machine, as shown for 32-bit and 64-bit guests in Figure 8 and Figure 9, respectively.

Figure 8. 32-bit SPECjbb2005 Results (Higher is Better)

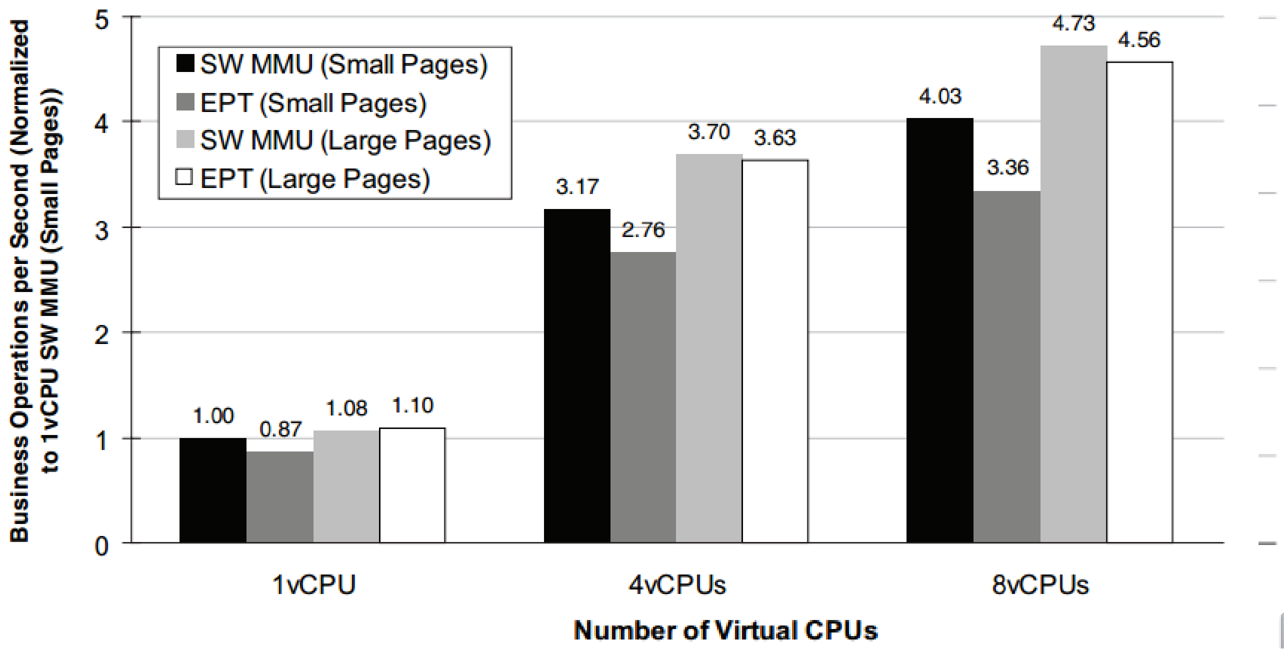
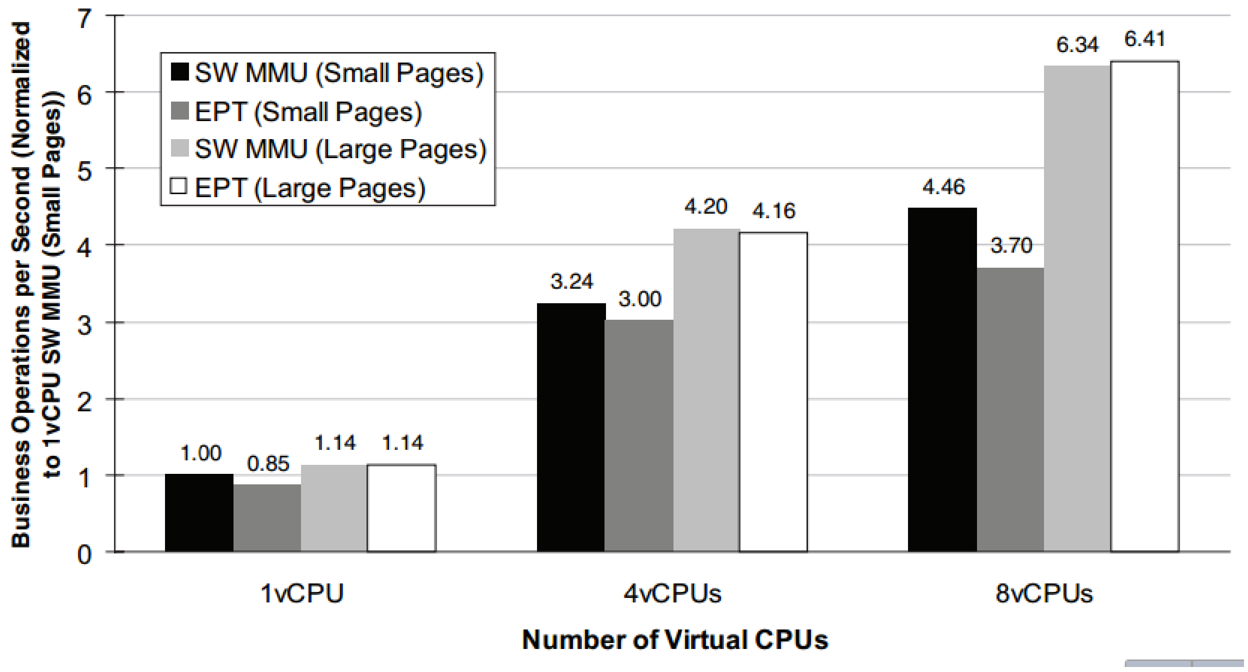


Figure 9. 64-bit SPECjbb2005 Results (Higher is Better)



See the above cases!

…

SPECjbb2005 is an industry-standard server-side Java benchmark. It has little MMU activity but exhibits high TLB miss activity due to Java's usage of the heap and associated garbage collection.

Modern x86 operating system vendors provide large page support to enhance the performance of such TLB-intensive workloads. Because EPT further increases the TLB miss latency (due to additional paging levels), **large page usage in the guest operating system is imperative for high performance** of such applications in an EPT-enabled virtual machine, as shown for 32-bit and 64-bit guests in Figure 8 and Figure 9, respectively.

…

Conclusion

Intel EPT-enabled CPUs offload a significant part of the VMM's MMU virtualization responsibilities to the hardware, resulting in higher performance. Results of experiments done on this platform indicate that the current VMware VMM leverages these features quite well, resulting in performance gains of up to 48% for MMU-intensive benchmarks and up to 600% for MMU-intensive microbenchmarks. We recommend that TLB-intensive workloads make extensive use of large pages to mitigate the higher cost of a TLB miss.

Also see Appendix B : Nested Paging – Overheads.

Article – KVM Memory

<< Note- Here, by "KVM" we really mean "Qemu" >>

The qemu/kvm process runs mostly like a normal Linux program. It allocates its memory with normal malloc() or mmap() calls. If a guest is going to have 1GB of physical memory, qemu/kvm will effectively do a malloc(1<<30), allocating 1GB of host virtual space. However, just like a normal program doing a malloc(), there is no actual physical memory allocated at the time of the malloc(). It will not be actually allocated until the first time it is touched.

Once the guest is running, it sees that malloc()'d memory area as being its physical memory. If the guest's kernel were to access what it sees as physical address 0x0, it will see the first page of that malloc() done by the qemu/kvm process.

It used to be that every time a KVM guest changed its page tables, the host had to be involved.

<< How does the host know the guest has changed paging table entries? By "trapping" into certain privileged instructions like the CR3 set (VM-x/AMD-V allows this) , TLB/page invalidate, etc. By setting mmu permissions on PTEs to be read-only so that it can trap into any write.. >>

The host would validate that the entries the guest put in its page tables were valid and that they did not access any memory which was not allowed. It did this with two mechanisms:

- *[Shadow Paging/softmmu]* : One was that the actual set of page tables being used by the virtualization hardware are separate from the page tables that the guest *thought* were being used. The guest first makes a change in its page tables. Later, the host notices this change, verifies it, and then makes a real page table which is accessed by the hardware. The guest software is not allowed to directly manipulate the page tables accessed by the hardware *<< it cannot as it's running in unprivileged mode (ring 3) >>* . This concept is called shadow page tables and it is a very common technique in virtualization.

- *[1st generation Hardware Virt support (VMx/AMD-V)]* : The second part was that the VMx/AMD-V extensions allowed the host to trap whenever the guest tried to set the register pointing to the base page table (CR3).

This technique works fine. But, it has some serious performance implications. A single access to a guest page can take up to 25 memory accesses to complete, which gets very costly. See this paper: http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf for more information.

The basic problem is that every access to memory must go through both the page tables of the guest and then the page tables of the host. The two-dimensional part comes in because the *page tables of* *the guest must *themselves* go through the page tables of the host*.

It can also be very costly for the host (hypervisor) to verify and maintain the shadow page tables.

Both AMD and Intel sought solutions to these problems and came up with similar answers called EPT and NPT. They specify a set of structures recognized by the hardware which can quickly translate guest physical addresses to host physical addresses *without* going through the host page tables. This shortcut removes the costly two-dimensional page table walks.
The problem with this is that the host page tables are what we use to enforce things like process separation. If a page was to be unmapped from the host (when it is swapped, for instance), it then we *must* coordinate that change with these new hardware EPT/NPT structures.

The solution in software is something Linux calls *mmu_notifiers*. Since the qemu/kvm memory is normal Linux memory (from the host Linux kernel's perspective) the (host) kernel may try to swap it, replace it, or even free it just like normal memory.

<< Good ref: [Memory Management Notifiers, LWN](), Jon Corbet, June 2008 >>

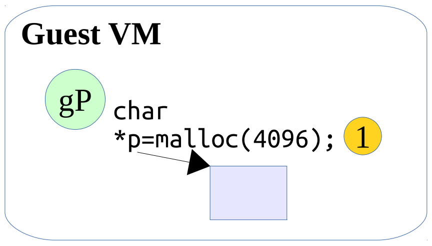
But, before the pages are actually given back to the host kernel for other use, the kvm/qemu guest is notified of the host's intentions. The kvm/qemu guest can then remove the page from the shadow page tables or the NPT/EPT structures. After the kvm/qemu guest has done this, the host kernel is then free to do what it wishes with the page.

[P.T.O. -->]

A day in the life of a KVM guest physical page: *<< diagrams added >>*

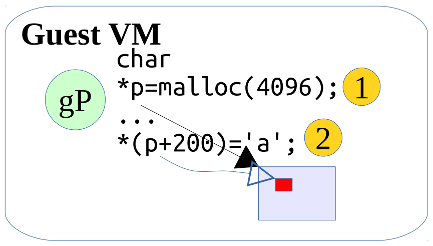
Fault-in path

1. QEMU calls malloc() and allocates virtual space for the page, but no backing physical page

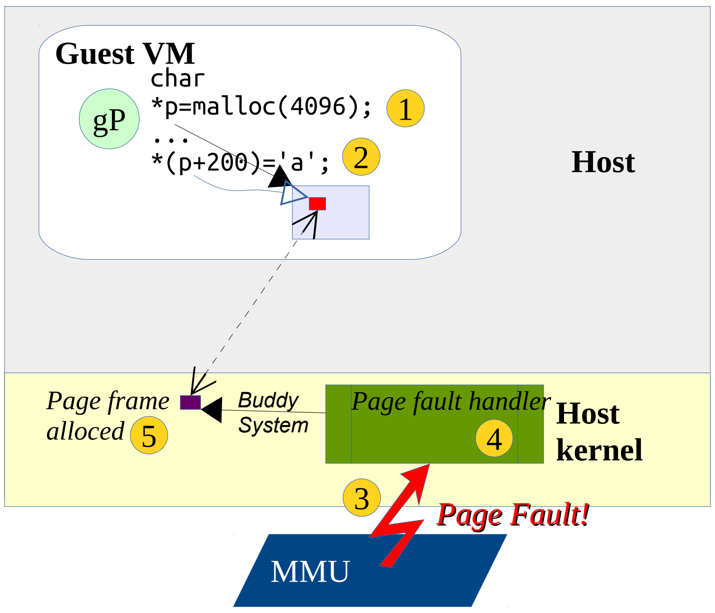


gP : guest process

2. The guest process touches what it thinks is a physical address, but this traps into the host since the memory is unallocated



3. The host kernel sees a page fault, calls do_page_fault() in the area that was malloc()'d, and if all goes well, allocates some memory to back it.



4. The host kernel creates a pte_t to connect the malloc()'d virtual address to a host physical address, makes rmap entries, puts it on the LRU, invalidates the TLB if requied (INVLPG instruction issued!), etc...

5. Appropriate hook functions get invoked:
    1. MMU-notifier hook function (*virt/kvm/kvm_main.c:kvm_mmu_notifier_change_pte()*) is called, which allows KVM to create an NPT/EPT entry for the new page. (and an spte entry?? *(yes, only if shadow paging is enabled, i.e., if tdp_enabled == False)*)
    2. If setup and occurs in host, the INVLPG hook in the hypervisor.

6. Host returns from page fault, guest execution resumes.

Swap-out path

Now, let's say the host is under memory pressure. The (guest) page from above has gone through the Linux LRU and has found itself on the inactive list. The kernel decides that it wants the page back:

1. The host kernel uses rmap structures to find out in which VMA(s) (vm_area_struct) the page is mapped.
2. The host kernel looks up the mm_struct associated with that VMA(s), and (from it's pgd entry,) walks down the Linux page tables to find the host hardware page table entry (pte_t) for the page.
3. The host kernel swaps out the page and clears out the pte_t (let's assume that this page was only used in a single place). But, before freeing the page:
4. The host kernel calls the mmu_notifier invalidate_page(). This looks up the page's entry in the NPT/EPT structures and removes it.
5. Now, any subsequent access to the page will trap into the host ((2) in the fault-in path above).

Memory Overcommit

Given all of the above, it should be apparent that just like normal processes on linux, the host memory allocated to the host processes representing kvm guests may be overcommitted. One distro's discussion of this appears here.

Also see:
Appendix B : Nested Paging – Overheads
Resource: Memory management for virtualization, LWN, Jon Corbet, April 2010.

In a nutshell, technologies that help mitigate serious virtualization performance issues:

- Hardware (CPU)
  - hardware-assist
    - nested paging (Intel EPT, AMD NPT)
    - running guest code directly on host CPU in guest/non-root mode
  - having large (huge) TLBs on the processor(s)
- OS
  - KSM (Kernel Samepage Merging)
  - THP (Transparent Huge Pages)
  - Tmem (Transcendent Memory)
  - Ballooning

Source

Terminology-
Hardware Virtualization Extensions from Intel

- **vmx** : virtualization extensions; instructions; instruction traps (for hypervisor), etc
- **vpid** : eliminates expensive TLB flushes when context switching between VM's
- **ept** : nested paging support; accelerates guest virtual to host physical address translation mechanism
- **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
- **vnmi**: Intel Virtual NMI feature which helps with certain sorts of interrupt events in guests.

ASIDE:
VirtualBox is a powerful opensource desktop virtualization solution that runs on Windows, Linux and Mac hosts.

From the VirtualBox User Manual (ver 5.0.10)

...

--paravirtprovider none|default|legacy|minimal|hyperv|kvm : This setting specifies which paravirtualization interface to provide to the guest operating system. Specifying none explicitly turns off exposing any paravirtualization interface. The option default , will pick an appropriate interface depending on the guest OS type while starting the VM. This is the default option chosen while creating new VMs. The legacy option is chosen for VMs which were created with older VirtualBox versions and will pick a paravirtualization interface while starting the VM with VirtualBox 5.0 and newer. The minimal provider is mandatory for Mac OS X guests, while kvm and hyperv are recommended for Linux and Windows guests respectively. These options are explained in detail under chapter 10.4, Paravirtualization providers, page 222.

• **--nestedpaging on|off :** If hardware virtualization is enabled, this additional setting enables or disables the use of the nested paging feature in the processor of your host system; see chapter 10.3, Hardware vs. software virtualization, page 221.

• **--largepages on|off :** If hardware virtualization and nested paging are enabled, for Intel VT-x only, an additional performance improvement of up to 5% can be obtained by enabling this setting. This causes the hypervisor to use large pages to reduce TLB use and overhead.

• **--vtxvpid on|off :** If hardware virtualization is enabled, for Intel VT-x only, this additional setting enables or disables the use of the tagged TLB (VPID) feature in the processor of your host system; see chapter 10.3, Hardware vs. software virtualization, page 221.

• **--vtxux on|off :** If hardware virtualization is enabled, for Intel VT-x only, this setting enables or disables the use of the unrestricted guest mode feature for executing your guest.

• **--accelerate3d on|off :** This enables, if the Guest Additions are installed, whether hardware 3D acceleration should be available; see chapter 4.5.1, Hardware 3D acceleration (OpenGL and Direct3D 8/9), page 75.

• **--accelerate2dvideo on|off :** This enables, if the Guest Additions are installed, whether 2D video acceleration should be available; see chapter 4.5.2, Hardware 2D video acceleration for Windows guests, page 77.

...

Internals

Guest Mode

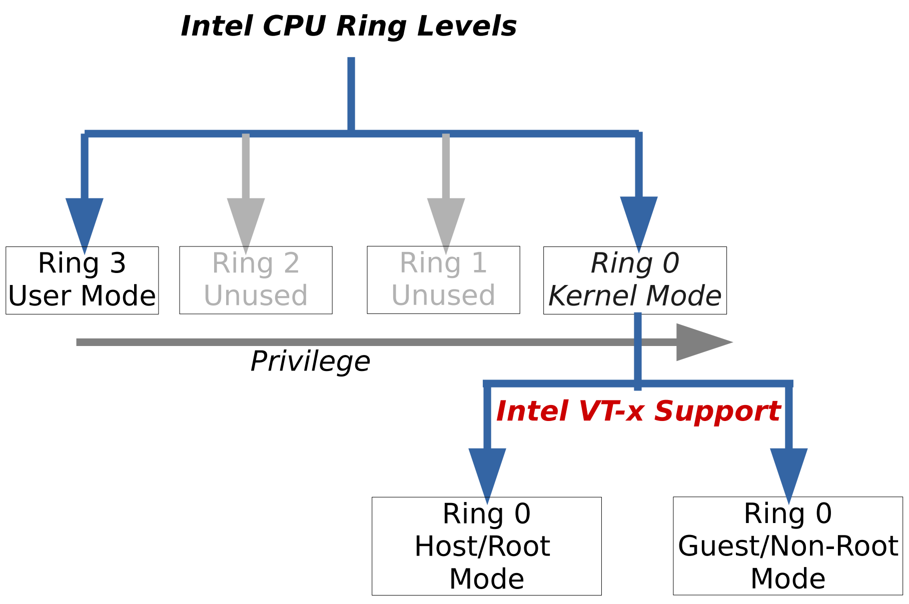
In order to execute a guest VM's code with high performance, we need to minimize "inteference" - the hypervisor continually butting in to "fixup" various things (memory translation with shadow paging, hooking into the host fault handling, interrupts, IO, etc etc).

Traditional Intel/AMD processors supported code execution in four modes, or "ring levels", of which only two are actually used in practice:

- Unprivileged mode (ring 3) : usermode processes/threads
- Privileged or Root mode (ring 0) : kernelmode code/threads

With the advent of hardware virtualization support, modern (x86) processors allow a third execution mode. This is achieved essentially by "splitting" Ring 0 into two modes:

- Ring 0 Host or "Root" mode : host kernel code/threads
- Ring 0 Guest or "Non-Root" mode (or the so-called "-1" privilege level) : guest code / threads



Executing guest code directly on the hardware cpu without (binary or other) transalation will yield major performance gains. To do this, we need a system of entering / transiting from host kernel mode to guest kernel mode and vice-versa. These transitions are called "VM Entry" and "VM Exit":

```
VM ENTER :: Mode Transitions:
 Ring 3 (User Mode) -> Ring 0 (kernel Root) -> Ring 0 (kernel Non-Root/Guest)

VM EXIT :: Mode Transitions:
 Ring 0 (kernel Non-Root/Guest) -> Ring 0 (kernel Root) -> Ring 3 (User Mode)
```

(In code terms, KVM effects VM Entry via the KVM_RUN ioctl (which Qemu invokes)).

VM Entry and Exits

Intel processors with virtualization extensions (VTx) support an addtional execution mode called Guest mode *[or Non-Root 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.

Modes:

- Unprivileged mode (ring 3) : usermode processes/threads
- Privileged or Root mode (ring 0) : kernelmode code/threads
- Non-Root or Guest mode (ring "-1") : VM guest code entry/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 (kvm-qemu) invoking the KVM RUN API.

VM Exits

Source: Best Practices for Paravirtualization Enhancements from Intel® Virtualization Technology: EPT and VT-d
...
As mentioned above, performance benefits from hardware page virtualization are tied to the prevalence of VT-x exit transitions. Specifically, higher levels of these events in an application workload approximately suggest higher likelihood for benefit from hardware page table virtualization.

VT-x exit transitions from the guest to monitor occur when guest code accesses or modifies privileged virtualized state, executes privileged instructions, or handles certain external events (such as external interrupts). Frequent exits are caused by page faults, external interrupts, control register reads/writes, and I/O instructions. Page fault exits include guest page table faults, MMU faults, APIC (Advanced Programmable Interrupt Controller) reads and writes, and device MMIO (memory-mapped I/O) reads and writes.
...

Virtualization Blog, Bartosz Milewski

- Virtual Machines: Virtualizing Virtual Memory

- Virtual Machines: The Traps

FYI:

How does the Hypervisor Trap into the OS?

Source: Virtual Machines: The Traps, Bartosz Milewski

In my previous blog I explained how a virtual machine (VM) manages memory behind the back of the operating system. Let's face it, it's all based on a lie perpetrated by the hypervisor. Every time the operating system is on the verge of discovering the lie, it is knocked unconscious by the hypervisor who then fixes up the OS's version of reality. There is a scene in the movie *Barnyard* that graphically illustrates this principle: The farmer is about to discover that farm animals are intelligent when he's kicked in the head by the mule.

To implement that mechanism, some help from the hardware is needed. The idea is that the execution of certain instructions or access to certain resources will cause a hardware trap that can be processed by the hypervisor. It's the same principle that is used by the OS to create virtual reality for user processes.

Rings of Protection

The x86 chip if full of vestigial organs, relics of some long forgotten trends in operating system design. One of them is the segmented memory architecture, another is the four rings of protection.

Three of the rings, 1, 2 and 3, correspond to user mode, while the OS kernel lives in ring 0. In practice, operating systems use just two rings, usually rings 3 and 0, to separate user processes from the kernel and from each other.

The hardware makes this illusion possible by making some processor instructions available only in ring 0. An attempt to execute them in user mode will result in a fault. Protected instructions include those that modify control registers, in particular the page directory register CR3 and the Interrupt Descriptor Table Register, IDTR. (There are also sensitive instructions that can only be executed with the appropriate I/O privilege level — those deal with I/O and interrupt flags.)

The OS, having exclusive access to privileged instructions can set up page tables and interrupt handlers — including the page-fault handler. With those, it can hide its own data structures (including said page tables and the IDT) and its code from user processes.

The OS switches from kernel mode to user mode by setting the lowest bit in the CR0 register. But since access to control registers doesn't work in user mode, switching back to kernel can only be done through a fault or an interrupt.

Hardware Virtualization

With all these rings of protection you'd think that virtualization of the x86 should be a piece of cake. Just put the hypervisor in ring 0, the OS in ring 1, and user processes in ring 3. Unfortunately this doesn't work, and for many years the x86 was considered non-virtualizable. I'll describe later how VMware was able to make this work using some pretty amazing tricks. Fortunately, in around 2005, both Intel, with its VT-x, and AMD, with its AMD-V, introduced hardware support for virtualization. The idea was to further split ring 0 into two layers, host mode and guest mode — the hypervisor (along with the host OS) running in the host mode and the (guest) OS in the guest mode. (The two modes are also called root and non-root.)

In this model, the hypervisor has complete control over hardware and memory. Before transiting into guest mode to give control to the OS, it prepares a special Virtual Machine Control Block (VMCB), *<< Update: this data structure is now called the Virtual Machine Control Structure – VMCS*[1] *>>* which will set the state of the guest. Besides storing copies of guest system registers, VMCS also includes control bits that specify conditions under which the host will trap (exit) into the hypervisor.

For instance, the hypervisor usually wants to intercept: reads or writes to control registers, specific exceptions, I/O operations, etc. It may also instruct the processor to use nested page tables. Once the VMCS is ready, the hypervisor executes *vmrun* *<< actually becomes the VMLAUNCH machine instruction; seen later in the KVM codebase >>* and lets the (guest) OS take control. The guest OS runs in ring 0 (guest mode) *<< non-root mode >>*.

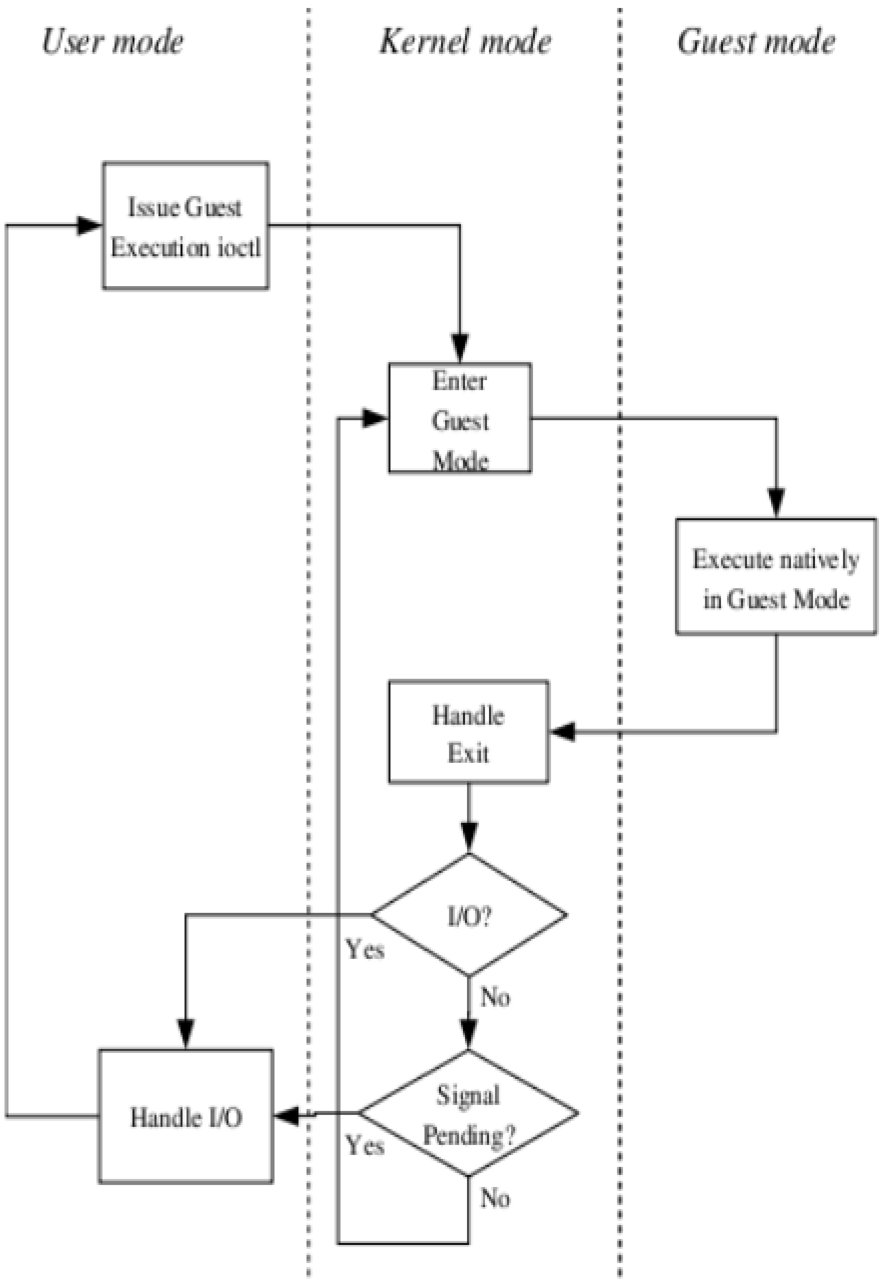
When any of the exit conditions that were specified in the VMCS are met, the processor exits into the hypervisor (host mode), storing the current state of the processor and the reason for the exit back into the VMCS.

The beauty of this system is that the hypervisor's intervention is (almost) totally transparent to the OS. In fact a thin hypervisor was used by Joanna Rutkowska in a successful attack on Windows Vista that stirred a lot of controversy at the 2006 Black Hat conference in Las Vegas. Borrowing from The Matrix, Rutkowska called her rootkit Blue Pill. I'll talk more about thin hypervisors in my next post.

... 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.

(See flow diagram on next page).

1 VMCS: Virtual Machine Control Structures. See Intel Manual Vol 3 Ch 24 for complete details.



Source
>>

SIDEBAR :: VMCS structure

See Kernel Virtual Machine (KVM), Shashank Rachamalla
…
TODO

arch/x86/kvm/vmx.c
```
/*
 * 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 (*const 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,
...
```

Software Virtualization

Before there was hadware support for virtualization in the x86 architecture, VMware created a tour de force software implementation of VM. The idea was to use the protection ring system of the x86 and run the operating system in ring 1 (or sometimes 3) instead of ring 0. Since ring 1 doesn't have kernel privileges, most of the protected things the OS tries to do would cause faults that could be vectored into the hypervisor.

Unfortunately there are some kernel instructions that, instead of faulting in user mode, fail quietly. The notorious example is the popf instruction that loads processor flags from the stack. The x86 flag register is a kitchen sink of bits, some of them used in arithmetic (sign flag, carry flag, etc.), others controlling the system (like I/O Privilege Level, IOPL). Only kernel code should be able to modify IOPL. However, when popf is called from user mode, it doesn't trap — the IOPL bits are quietly ignored. Obviously this wreaks havoc on the operation of the OS when it's run in user mode.

The solution is to modify the OS code, replacing all occurrences of popf with hypervisor calls. That's what VMWere did — sort of. Since they didn't have the source code to all operating systems, they had to do the translation in binary and on the fly. Instead of executing the OS binaries directly, they redirected the stream of instructions to their binary translator, which methodically scanned it for the likes of popf, and produced a new stream of instructions with appropriate traps, which was then sent to the processor. Of course, once you get on the path of binary translation, you have to worry about things like the targets of all jumps and branches having to be adjusted on the fly. You have to divide code into basic blocks and then patch all the jumps and so on.

The amazing thing is that all this was done with very small slowdown. In fact when VMware started replacing this system with the newly available VT-x and AMD-V, the binary translation initially performed better than hardware virtualization (this has changed since).

Future Directions

I've been describing machine virtualization as a natural extension of process virtualization. But why stop at just two levels? Is it possible to virtualize a virtual machine? Not only is it possible, but in some cases it's highly desirable. Take the case of Windows 7 and its emulation of Windows XP that uses Virtual PC. It should be possible to run this combo inside a virtual box alongside, say, Linux.

There is also a trend to use specialized thin hypervisors for debugging (e.g., Corensic's Jinx) or security (McAfee's DeepSAFE). Such programs should be able to work together. Although there are techniques for implementing nested virtualization on the x86, there is still no agreement on the protocol that would allow different hypervisors to cooperate.

In the next installment I'll talk about thin hypervisors.

Bibliography

1. Gerald J. Popek, Robert P. Goldberg, Formal Requirements for Virtualizable Third Generation Architectures. A set of formal requirements that make virtualization possible.
2. James Smith, Ravi Nair, The Architecture of Virtual Machines
3. Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, Pratap Subrahmanyam, The Evolution of an x86 Virtual Machine Monitor
4. Keith Adams, Ole Agesen, A Comparison of Software and Hardware Techniques for x86 Virtualization
5. Paul Barham et al., Xen and the Art of Virtualization.
6. Muli Ben-Yehuda et al., The Turtles Project: Design and Implementation of Nested Virtualization.

Ref: http://blog.vmsplice.net/2011/03/qemu-internals-big-picture-overview.html

KVM virtualization

KVM is a virtualization feature in the Linux kernel that lets a program like qemu safely execute guest code directly on the host CPU. This is only possible when the target architecture is supported by the host CPU; today that means x86-on-x86 virtualization only.

In order to execute guest code using KVM, the qemu process opens */dev/kvm* and issues the KVM_RUN ioctl. The KVM kernel module uses hardware virtualization extensions found on modern Intel and AMD CPUs to directly execute guest code.

When the guest accesses a hardware device register, halts the guest CPU, or performs other special operations, KVM exits back to qemu. At that point qemu can emulate the desired outcome of the operation or simply wait for the next guest interrupt in the case of a halted guest CPU.

The basic flow of a guest CPU is as follows (QEMU code paths):

```
open("/dev/kvm")
ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)
for (;;) {
     ioctl(KVM_RUN)
     switch (exit_reason) {
     case KVM_EXIT_IO:  /* [...] */
     case KVM_EXIT_HLT: /* [...] */
     }
}
```

[...]

Since qemu system emulation provides a full virtual machine inside the qemu userspace process, the details of what processes are running inside the guest are not directly visible from the host.

One way of understanding this is that qemu provides:
- a slab of guest RAM,
- the ability to execute guest code, and
- emulated hardware devices;

therefore any operating system (or no operating system at all) can run inside the guest. There is no ability for the host to peek inside an arbitrary guest.

[...]

SIDEBAR :: QEMU Monitor Mode

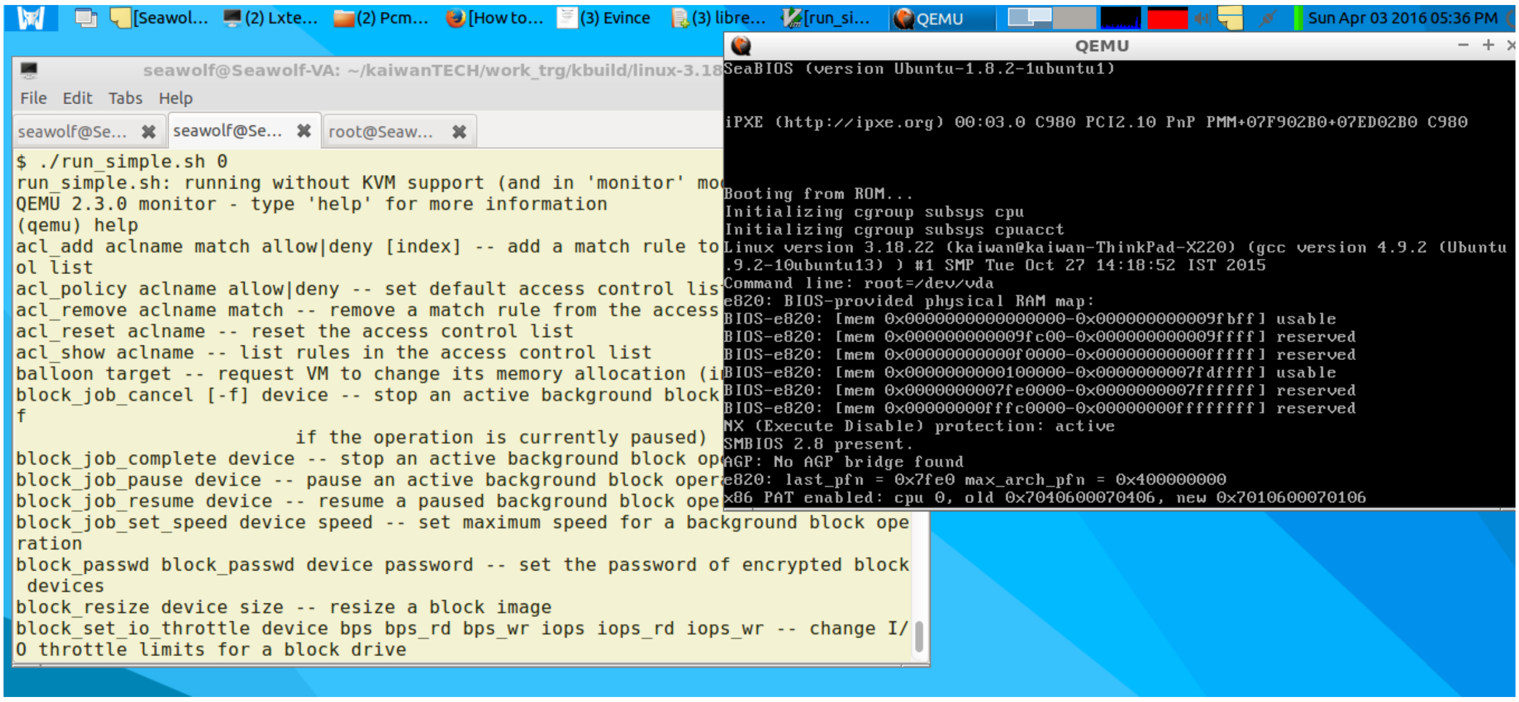
Well, it *is* possible to peek inside a guest VM when running Qemu: run it with “monitor-mode” enabled.
See the QEMU/Monitor Wiki page for full details.

<<
Example:

Assumptions:
Using Debian debootstrap, we download a ready-to-use Debian root filesystem (“wheezy” Debian v7); we also have (or have built) a Linux kernel image (bzImage) for x86_64.

Emulate the guest OS with Qemu like this:

```
$ sudo modprobe kvm
$ sudo qemu-system-x86_64 --enable-kvm -kernel <path/to/bzImage> \
  -drive file=<path/to/deb-rootfs>,if=virtio,format=raw -append root=/dev/vda \
  -device virtio-serial-pci \
  -monitor stdio    # enables Qemu “monitor” mode
```



```
$ ./run_simple.sh 0
run_simple.sh: running without KVM support (and in 'monitor' mo
QEMU 2.3.0 monitor - type 'help' for more information
(qemu) help
acl_add aclname match allow|deny [index] -- add a match rule to
ol list
acl_policy aclname allow|deny -- set default access control lis
acl_remove aclname match -- remove a match rule from the access
acl_reset aclname -- reset the access control list
acl_show aclname -- list rules in the access control list
balloon target -- request VM to change its memory allocation (i
block_job_cancel [-f] device -- stop an active background block
f
                             if the operation is currently paused)
block_job_complete device -- stop an active background block op
block_job_pause device -- pause an active background block oper
block_job_resume device -- resume a paused background block ope
block_job_set_speed device speed -- set maximum speed for a background block ope
ration
block_passwd block_passwd device password -- set the password of encrypted block
 devices
block_resize device size -- resize a block image
block_set_io_throttle device bps bps_rd bps_wr iops iops_rd iops_wr -- change I/
O throttle limits for a block drive
```

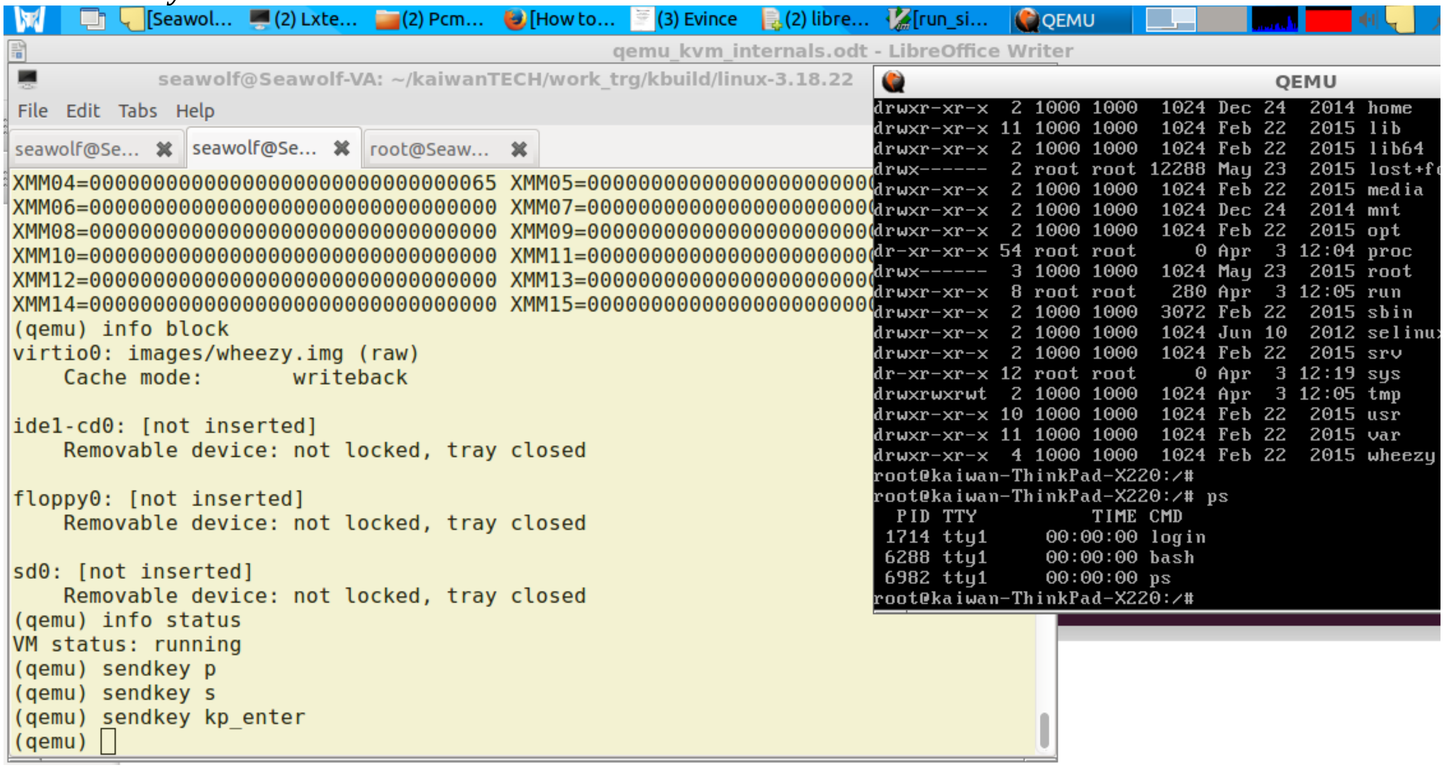
```
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F902B0+07ED02B0 C980



Booting from ROM...
Initializing cgroup subsys cpu
Initializing cgroup subsys cpuacct
Linux version 3.18.22 (kaiwan@kaiwan-ThinkPad-X220) (gcc version 4.9.2 (Ubuntu
.9.2-10ubuntu13) ) #1 SMP Tue Oct 27 14:18:52 IST 2015
Command line: root=/dev/vda
e820: BIOS-provided physical RAM map:
BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable
BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff] reserved
BIOS-e820: [mem 0x00000000000f0000-0x00000000000fffff] reserved
BIOS-e820: [mem 0x0000000000100000-0x0000000007fdffff] usable
BIOS-e820: [mem 0x0000000007fe0000-0x0000000007ffffff] reserved
BIOS-e820: [mem 0x00000000fffc0000-0x00000000ffffffff] reserved
NX (Execute Disable) protection: active
SMBIOS 2.8 present.
AGP: No AGP bridge found
e820: last_pfn = 0x7fe0 max_arch_pfn = 0x400000000
x86 PAT enabled: cpu 0, old 0x7040600070406, new 0x7010600070106
```

```
...
(qemu)
(qemu) info status
VM status: running
(qemu) info roms
fw=genroms/kvmvapic.bin size=0x002400 name="kvmvapic.bin"
fw=genroms/linuxboot.bin size=0x000400 name="linuxboot.bin"
addr=00000000fffc0000 size=0x040000 mem=rom name="bios-256k.bin"
/rom@etc/acpi/tables size=0x020000 name="etc/acpi/tables"
/rom@etc/table-loader size=0x001000 name="etc/table-loader"
(qemu) info uuid
00000000-0000-0000-0000-000000000000
(qemu) info block
virtio0: images/wheezy.img (raw)

ide1-cd0: [not inserted]
    Removable device: not locked, tray closed

floppy0: [not inserted]
    Removable device: not locked, tray closed

sd0: [not inserted]
    Removable device: not locked, tray closed
(qemu) info tlb                      << show virtual to physical memory mappings >>
<<Virtual Page>>: <<Physical Page>>
0000000000400000: 0000000076c3000 ----A--U-
0000000000401000: 0000000007e0f000 ----A--U-
0000000000402000: 0000000007693000 ----A--U-
0000000000403000: 000000000769a000 ----A--U-
0000000000404000: 00000000076a3000 ----A--U-
0000000000405000: 0000000007997000 ----A--U-
0000000000406000: 0000000001bc9000 ----A--U-
0000000000606000: 000000000613a000 X--DA--U-
0000000000607000: 0000000007f6f000 X—DA—UW
00007f6104f9a000: 00000000017c8000 ----A--U-
00007f6104f9b000: 00000000017c9000 ----A--U-
```

```
00007f6104f9c000: 00000000017ca000 ----A--U-
...
00007ffd303d1000: 00000000fed00000 X---AC-U-  << Wow! See below the transition from
                                               user-to-kernel virtual address space! >>
00007ffd303d2000: 0000000001805000 ----A--U-
ffff880000000000: 0000000000000000 XG-DA---W
ffff880000001000: 0000000000001000 XG-DA---W
...
ffffffffff5fd000: 00000000fee00000 XG-DACT-W
ffffffffff600000: 0000000001804000 XG-DA--U-
(qemu)
```

[P.T.O.]

sendkey demo



```
XMM04=00000000000000000000000000000065 XMM05=0000000000000000000000
XMM06=00000000000000000000000000000000 XMM07=0000000000000000000000
XMM08=00000000000000000000000000000000 XMM09=0000000000000000000000
XMM10=00000000000000000000000000000000 XMM11=0000000000000000000000
XMM12=00000000000000000000000000000000 XMM13=0000000000000000000000
XMM14=00000000000000000000000000000000 XMM15=0000000000000000000000
(qemu) info block
virtio0: images/wheezy.img (raw)
    Cache mode:       writeback

ide1-cd0: [not inserted]
    Removable device: not locked, tray closed

floppy0: [not inserted]
    Removable device: not locked, tray closed

sd0: [not inserted]
    Removable device: not locked, tray closed
(qemu) info status
VM status: running
(qemu) sendkey p
(qemu) sendkey s
(qemu) sendkey kp_enter
(qemu) 
```

```
drwxr-xr-x  2 1000 1000  1024 Dec 24  2014 home
drwxr-xr-x 11 1000 1000  1024 Feb 22  2015 lib
drwxr-xr-x  2 1000 1000  1024 Feb 22  2015 lib64
drwx------  2 root root 12288 May 23  2015 lost+f
drwxr-xr-x  2 1000 1000  1024 Feb 22  2015 media
drwxr-xr-x  2 1000 1000  1024 Dec 24  2014 mnt
drwxr-xr-x  2 1000 1000  1024 Feb 22  2015 opt
dr-xr-xr-x 54 root root     0 Apr  3 12:04 proc
drwx------  3 1000 1000  1024 May 23  2015 root
drwxr-xr-x  8 root root   280 Apr  3 12:05 run
drwxr-xr-x  2 1000 1000  3072 Feb 22  2015 sbin
drwxr-xr-x  2 1000 1000  1024 Jun 10  2012 selinu
drwxr-xr-x  2 1000 1000  1024 Feb 22  2015 srv
dr-xr-xr-x 12 root root     0 Apr  3 12:19 sys
drwxrwxrwt  2 1000 1000  1024 Apr  3 12:05 tmp
drwxr-xr-x 10 1000 1000  1024 Feb 22  2015 usr
drwxr-xr-x 11 1000 1000  1024 Feb 22  2015 var
drwxr-xr-x  4 1000 1000  1024 Feb 22  2015 wheezy
root@kaiwan-ThinkPad-X220:/#
root@kaiwan-ThinkPad-X220:/# ps
  PID TTY          TIME CMD
 1714 tty1     00:00:00 login
 6288 tty1     00:00:00 bash
 6982 tty1     00:00:00 ps
root@kaiwan-ThinkPad-X220:/#
```

>>

Code View

Source: QEMU codebase
Ver 2.4.50, Aug 2015 (downloaded zip from https://github.com/qemu/qemu)

FYI see include/sysemu/kvm.h : QEMU KVM support.

kvm-all.c

```
...
static void kvm_accel_class_init(ObjectClass *oc, void *data)
{
    AccelClass *ac = ACCEL_CLASS(oc);
    ac->name = "KVM";
    ac->init_machine = kvm_init;
    ac->allowed = &kvm_allowed;
}

...
static int kvm_init(MachineState *ms)
{
    MachineClass *mc = MACHINE_GET_CLASS(ms);
    static const char upgrade_note[] =
        "Please upgrade to at least kernel 2.6.29 or recent kvm-kmod\n"
        "(see http://sourceforge.net/projects/kvm).\n";
...
    KVMState *s;
...

  s->fd = qemu_open("/dev/kvm", O_RDWR);    << Qemu: if /dev/kvm is present (and other
checks concur), Qemu ‘understands’ that it’s running with hardware-assist and will take
advantage of it! >>

...
<< Validity checks, etc >>
...

do {                             << Qemu: Create the VM ! This latches into the KVM kernel
                                      module's device ioctl code path (seen later) >>
        ret = kvm_ioctl(s, KVM_CREATE_VM, type);
    } while (ret == -EINTR);

...
```

<<
While a VM was running on the host (with VirtualBox as the emulator), I attempted to run *qemu-system-x86_64* on the host to emulate a simple Debian system. It does work, however, without acceleration (i.e. without kvm kernel support).
See the output below: apparently the KVM_CREATE_VM ioctl(2) fails with an EBUSY error! So qemu falls back to the slower 'tcg' mode...

```
+ sudo qemu-system-x86_64 --enable-kvm -kernel images/bzImage -drive
file=images/wheezy.img,if=virtio,format=raw -append root=/dev/vda -device virtio-
serial-pci -monitor stdio
ioctl(KVM_CREATE_VM) failed: 16 Device or resource busy    << fails with EBUSY >>
failed to initialize KVM: Device or resource busy
```

```
+ sudo qemu-system-x86_64 -kernel images/bzImage -drive
file=images/wheezy.img,if=virtio,format=raw -append root=/dev/vda -display curses
                                                         << succeeds >>
>>

    ret = kvm_arch_init(ms, s);

...

    cpu_interrupt_handler = kvm_handle_interrupt;

    return 0;
...
}




void qemu_init_vcpu(CPUState *cpu)
{
    cpu->nr_cores = smp_cores;
    cpu->nr_threads = smp_threads;
    cpu->stopped = true;
    if (kvm_enabled()) {          << Qemu tied to kvm hypervisor ?? this in turn
                                      depends on the value of CONFIG_KVM >>
        qemu_kvm_start_vcpu(cpu);
    } else if (tcg_enabled()) {   << no? Use tcg - tiny code generator - module;
                                      tcg is QEMU's pure usermode emulation mode;
                                      it works of course, but is definitely slower >>
        qemu_tcg_init_vcpu(cpu);
    } else {
        qemu_dummy_start_vcpu(cpu);
    }
}

[...]

cpus.c
[...]
    qemu_cond_init(cpu->halt_cond);
    snprintf(thread_name, VCPU_THREAD_NAME_SIZE, "CPU %d/KVM",
             cpu->cpu_index);
    qemu_thread_create(cpu->thread, thread_name, qemu_kvm_cpu_thread_fn,
                       cpu, QEMU_THREAD_JOINABLE); << wrapper over pthread_create() and
                                                      related APIs >>
    while (!cpu->created) {
        qemu_cond_wait(&qemu_cpu_cond, &qemu_global_mutex);
    }
[...]


<< The QEMU/kvm thread representing a vcpu >>
static void *qemu_kvm_cpu_thread_fn(void *arg)
{
    CPUState *cpu = arg;
    int r;

    rcu_register_thread();

    qemu_mutex_lock_iothread();
    qemu_thread_get_self(cpu->thread);
    cpu->thread_id = qemu_get_thread_id();
    cpu->can_do_io = 1;
    current_cpu = cpu;

    r = kvm_init_vcpu(cpu);   << both arch-independent and arch-dependant init >>
    if (r < 0) {
        fprintf(stderr, "kvm_init_vcpu failed: %s\n", strerror(-r));
        exit(1);
    }

    qemu_kvm_init_cpu_signals(cpu);

    /* signal CPU creation */
    cpu->created = true;
    qemu_cond_signal(&qemu_cpu_cond);

    while (1) {                  << Qemu event loop ;
                                see detailed comments in include/qemu/main-loop.h  >>
        if (cpu_can_run(cpu)) {
            r = kvm_cpu_exec(cpu);   << see code below >>
            if (r == EXCP_DEBUG) {
                cpu_handle_guest_debug(cpu);
            }
        }
        qemu_kvm_wait_io_event(cpu);  << see code below >>
    }

    return NULL;
}
[...]
```

```
kvm_all.c
[...]
int kvm_cpu_exec(CPUState *cpu)
{
    struct kvm_run *run = cpu->kvm_run;
    int ret, run_ret;

    DPRINTF("kvm_cpu_exec()\n");

   [...]

    do {
        MemTxAttrs attrs;

        if (cpu->kvm_vcpu_dirty) {
            kvm_arch_put_registers(cpu, KVM_PUT_RUNTIME_STATE);
            cpu->kvm_vcpu_dirty = false;
        }

        kvm_arch_pre_run(cpu, run);
        if (cpu->exit_request) {
            DPRINTF("interrupt exit requested\n");
            /*
             * KVM requires us to reenter the kernel after IO exits to complete
             * instruction emulation. This self-signal will ensure that we
             * leave ASAP again.
```

```
                 */
                qemu_cpu_kick_self();
        }
```

<<

VM Entry and VM Exit:

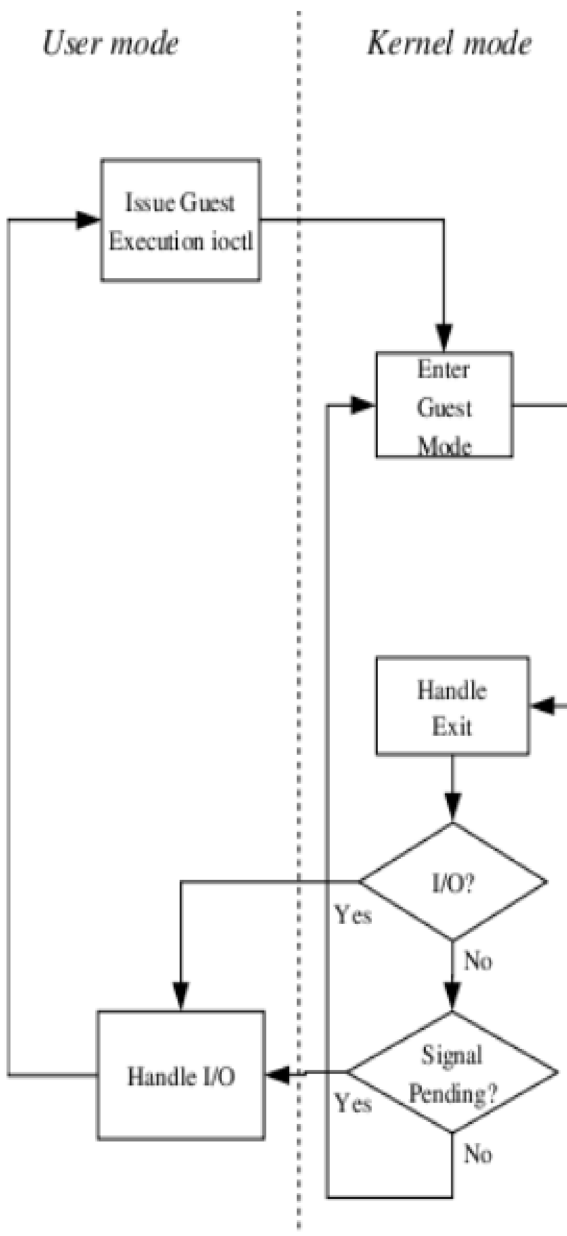
Qemu issues the ioctl KVM_RUN; this tells the hypervisor to allow the guest VM code to run unaffected on the host cpu (in “Guest” or “Non-Root” Ring 0 << *also called “ring -1”* >> kernel mode). If for any reason the host must intervene, the guest code will cause a trap into the hypervisor and we will “exit” the Non-Root Guest mode – this is called a “VM Exit”. The guest VM “exits” to the hypervisor (kvm), which processes the trap, possibly returning back to the usermode emulator (Qemu), then back to the hypervisor and finally back to Guest Mode “entering” it again – called a “VM Entry”.

[A brief article “Virtualization and Performance: Understanding VM Exits”, David Ott, Intel].

The possible exit “reasons” are encoded into the VMCS, and upon VM exit (to Qemu), the exit reason will be returned here (see code below).

[P.T.O.]

KVM Flow diagram- reproduced below for clarity:



```
>>
        run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);
<<
Becomes an ioctl() to KVM ! ioctl(2) is a system call; we switch to kernel "Root" mode
ring 0 and then hit the VM ENTER :: Qemu guest code now runs in "Guest / Non-Root"
kernel mode directly on the processor!
Mode Transitions:
 Ring 3 (User Mode) -> Ring 0 (kernel Root) -> Ring 0/"-1" (Non-Root/Guest)

VM EXIT :: Back from kernel Guest mode to normal Usermode Ring 3 execution of Qemu now.
Mode Transitions:
 Ring 0/"-1" (Non-Root/Guest) -> Ring 0 (kernel Root) -> Ring 3 (User Mode)
>>
        attrs = kvm_arch_post_run(cpu, run);
```

```
[...]

        trace_kvm_run_exit(cpu->cpu_index, run->exit_reason);
        switch (run->exit_reason) {
        case KVM_EXIT_IO:
            DPRINTF("handle_io\n");
            /* Called outside BQL */
            kvm_handle_io(run->io.port, attrs,
                          (uint8_t *)run + run->io.data_offset,
                          run->io.direction,
                          run->io.size,
                          run->io.count);
            ret = 0;
            break;
        case KVM_EXIT_MMIO:
            DPRINTF("handle_mmio\n");
            /* Called outside BQL */
            address_space_rw(&address_space_memory,
                             run->mmio.phys_addr, attrs,
                             run->mmio.data,
                             run->mmio.len,
                             run->mmio.is_write);
            ret = 0;
            break;
        case KVM_EXIT_IRQ_WINDOW_OPEN:
            DPRINTF("irq_window_open\n");
            ret = EXCP_INTERRUPT;
            break;
[...]
     default:
            DPRINTF("kvm_arch_handle_exit\n");
            ret = kvm_arch_handle_exit(cpu, run);
            break;
        }
    } while (ret == 0);
[...]


static void qemu_kvm_wait_io_event(CPUState *cpu)
{
    while (cpu_thread_is_idle(cpu)) {
        qemu_cond_wait(cpu->halt_cond, &qemu_global_mutex); << blocks on “io” becoming
available >>
    }

    qemu_kvm_eat_signals(cpu);      << process all pending signals >>
    qemu_wait_io_event_common(cpu);
          <<  consumer: calls:
              flush_queued_work(cpu);
                 which calls :
             ...
                 while ((wi = cpu->queued_work_first)) {
                    cpu->queued_work_first = wi->next;
                    wi->func(wi->data);
                    wi->done = true;
                    if (wi->free) {
                        g_free(wi);
                    }
                 }
             ...
              >>
```

```
}
```

KVM (Hypervisor) Code View

The kvm hypervisor is encoded as a kernel module (an LKM). In mainline from 2.6.20.

Linux kernel ver: 4.1.0-rc7 (July 2015)

virt/kvm/kvm_main.c

```
/*
 * Kernel-based Virtual Machine driver for Linux
 *
 * This module enables machines with Intel VT-x extensions to run virtual
 * machines without emulation or binary translation.
 *
 * Copyright (C) 2006 Qumranet, Inc.
 * Copyright 2010 Red Hat, Inc. and/or its affiliates.
 *
 * Authors:
 *   Avi Kivity   <avi@qumranet.com>
 *   Yaniv Kamay  <yaniv@qumranet.com>
 *
 * This work is licensed under the terms of the GNU GPL, version 2.  See
 * the COPYING file in the top-level directory.
 *
 */
[...]
```

KVM kernel module entry point *(vmx = Virtual Machine eXtensions, Intel)*:

arch/x86/kvm/vmx.c

```
static int __init vmx_init(void)
{
    int r = kvm_init(&vmx_x86_ops, sizeof(struct vcpu_vmx),
                     __alignof__(struct vcpu_vmx), THIS_MODULE);
    if (r)
        return r;
[...]

}

module_init(vmx_init);

...

static struct kvm_x86_ops vmx_x86_ops = {
    .cpu_has_kvm_support = cpu_has_kvm_support,
    .disabled_by_bios = vmx_disabled_by_bios,
    .hardware_setup = hardware_setup,
    .hardware_unsetup = hardware_unsetup,
    .check_processor_compatibility = vmx_check_processor_compat,
    .hardware_enable = hardware_enable,
    .hardware_disable = hardware_disable,
    .cpu_has_accelerated_tpr = report_flexpriority,

    .vcpu_create = vmx_create_vcpu,
    .vcpu_free = vmx_free_vcpu,
    .vcpu_reset = vmx_vcpu_reset,

    .prepare_guest_switch = vmx_save_host_state,
    .vcpu_load = vmx_vcpu_load,
    .vcpu_put = vmx_vcpu_put,

    .update_db_bp_intercept = update_exception_bitmap,
    .get_msr = vmx_get_msr,
    .set_msr = vmx_set_msr,
    .get_segment_base = vmx_get_segment_base,
    .get_segment = vmx_get_segment,
    .set_segment = vmx_set_segment,
    .get_cpl = vmx_get_cpl,
    .get_cs_db_l_bits = vmx_get_cs_db_l_bits,
    .decache_cr0_guest_bits = vmx_decache_cr0_guest_bits,
    .decache_cr3 = vmx_decache_cr3,
    .decache_cr4_guest_bits = vmx_decache_cr4_guest_bits,
    .set_cr0 = vmx_set_cr0,
    .set_cr3 = vmx_set_cr3,
    .set_cr4 = vmx_set_cr4,
    .set_efer = vmx_set_efer,
    .get_idt = vmx_get_idt,
    .set_idt = vmx_set_idt,
    .get_gdt = vmx_get_gdt,
    .set_gdt = vmx_set_gdt,
    .get_dr6 = vmx_get_dr6,
    .set_dr6 = vmx_set_dr6,
    .set_dr7 = vmx_set_dr7,
    .sync_dirty_debug_regs = vmx_sync_dirty_debug_regs,
    .cache_reg = vmx_cache_reg,
    .get_rflags = vmx_get_rflags,
    .set_rflags = vmx_set_rflags,
    .fpu_activate = vmx_fpu_activate,
    .fpu_deactivate = vmx_fpu_deactivate,

    .tlb_flush = vmx_flush_tlb,

    .run = vmx_vcpu_run,
    .handle_exit = vmx_handle_exit,
    .skip_emulated_instruction = skip_emulated_instruction,
    .set_interrupt_shadow = vmx_set_interrupt_shadow,
    .get_interrupt_shadow = vmx_get_interrupt_shadow,
    .patch_hypercall = vmx_patch_hypercall,
    .set_irq = vmx_inject_irq,
    .set_nmi = vmx_inject_nmi,
    .queue_exception = vmx_queue_exception,
    .cancel_injection = vmx_cancel_injection,
    .interrupt_allowed = vmx_interrupt_allowed,
    .nmi_allowed = vmx_nmi_allowed,
    .get_nmi_mask = vmx_get_nmi_mask,
    .set_nmi_mask = vmx_set_nmi_mask,
    .enable_nmi_window = enable_nmi_window,
    .enable_irq_window = enable_irq_window,
    .update_cr8_intercept = update_cr8_intercept,
    .set_virtual_x2apic_mode = vmx_set_virtual_x2apic_mode,
    .set_apic_access_page_addr = vmx_set_apic_access_page_addr,
    .vm_has_apicv = vmx_vm_has_apicv,
    .load_eoi_exitmap = vmx_load_eoi_exitmap,
    .hwapic_irr_update = vmx_hwapic_irr_update,
    .hwapic_isr_update = vmx_hwapic_isr_update,
    .sync_pir_to_irr = vmx_sync_pir_to_irr,
    .deliver_posted_interrupt = vmx_deliver_posted_interrupt,

    .set_tss_addr = vmx_set_tss_addr,
    .get_tdp_level = get_ept_level,
    .get_mt_mask = vmx_get_mt_mask,

    .get_exit_info = vmx_get_exit_info,

    .get_lpage_level = vmx_get_lpage_level,

    .cpuid_update = vmx_cpuid_update,

    .rdtscp_supported = vmx_rdtscp_supported,
    .invpcid_supported = vmx_invpcid_supported,

    .set_supported_cpuid = vmx_set_supported_cpuid,

    .has_wbinvd_exit = cpu_has_vmx_wbinvd_exit,

    .set_tsc_khz = vmx_set_tsc_khz,
    .read_tsc_offset = vmx_read_tsc_offset,
    .write_tsc_offset = vmx_write_tsc_offset,
    .adjust_tsc_offset = vmx_adjust_tsc_offset,
    .compute_tsc_offset = vmx_compute_tsc_offset,
    .read_l1_tsc = vmx_read_l1_tsc,

    .set_tdp_cr3 = vmx_set_cr3,

    .check_intercept = vmx_check_intercept,
    .handle_external_intr = vmx_handle_external_intr,
    .mpx_supported = vmx_mpx_supported,
    .xsaves_supported = vmx_xsaves_supported,

    .check_nested_events = vmx_check_nested_events,

    .sched_in = vmx_sched_in,

    .slot_enable_log_dirty = vmx_slot_enable_log_dirty,
    .slot_disable_log_dirty = vmx_slot_disable_log_dirty,
    .flush_log_dirty = vmx_flush_log_dirty,
    .enable_log_dirty_pt_masked = vmx_enable_log_dirty_pt_masked,
};
...
```

virt/kvm/kvm_main.c

```
int kvm_init(void *opaque, unsigned vcpu_size, unsigned vcpu_align,
          struct module *module)
{
    int r;
    int cpu;
...
    kvm_arch_init(opaque);
```

```
...
    kvm_irqfd_init();
...
    kvm_arch_hardware_setup();

...
    register_cpu_notifier(&kvm_cpu_notifier);

...
    register_reboot_notifier(&kvm_reboot_notifier);

    /* A kmem cache lets us meet the alignment requirements of fx_save. */
    if (!vcpu_align)
        vcpu_align = __alignof__(struct kvm_vcpu);
    kvm_vcpu_cache = kmem_cache_create("kvm_vcpu", vcpu_size, vcpu_align,
                                       0, NULL);

...
    kvm_async_pf_init();  << Async page fault (slab cache) setup >>

...
    kvm_chardev_ops.owner = module;
    kvm_vm_fops.owner = module;
    kvm_vcpu_fops.owner = module;

    r = misc_register(&kvm_dev);

...
    register_syscore_ops(&kvm_syscore_ops);

    kvm_preempt_ops.sched_in = kvm_sched_in;
    kvm_preempt_ops.sched_out = kvm_sched_out;

...
  r = kvm_init_debug();
   <<
   static int kvm_init_debug(void)
   {
     int r = -EEXIST;
     struct kvm_stats_debugfs_item *p;

     kvm_debugfs_dir = debugfs_create_dir("kvm", NULL);   << Debugfs >>
     if (kvm_debugfs_dir == NULL)
        goto out;

     for (p = debugfs_entries; p->name; ++p) {
        p->dentry = debugfs_create_file(p->name, 0444, kvm_debugfs_dir,
                        (void *)(long)p->offset,
                        stat_fops[p->kind]);

    >>

<<------------------------------
```

Debugfs entries for kvm: an example of runtime usage:
<< xplore_fs.sh : convenience script to find files, folders & show content >>

```
# xplore_fs.sh /sys/kernel/debug/kvm/
===================== SUMMARY LIST of Files ==========================

Note: Max Depth is 4.

/sys/kernel/debug/kvm/
/sys/kernel/debug/kvm/largepages
/sys/kernel/debug/kvm/remote_tlb_flush
/sys/kernel/debug/kvm/mmu_unsync
/sys/kernel/debug/kvm/mmu_cache_miss
/sys/kernel/debug/kvm/mmu_recycled
/sys/kernel/debug/kvm/mmu_flooded
/sys/kernel/debug/kvm/mmu_pde_zapped
/sys/kernel/debug/kvm/mmu_pte_updated
/sys/kernel/debug/kvm/mmu_pte_write
/sys/kernel/debug/kvm/mmu_shadow_zapped
/sys/kernel/debug/kvm/nmi_injections
/sys/kernel/debug/kvm/irq_injections
/sys/kernel/debug/kvm/insn_emulation_fail
/sys/kernel/debug/kvm/insn_emulation
/sys/kernel/debug/kvm/fpu_reload
/sys/kernel/debug/kvm/efer_reload
/sys/kernel/debug/kvm/host_state_reload
/sys/kernel/debug/kvm/irq_exits
/sys/kernel/debug/kvm/request_irq
/sys/kernel/debug/kvm/hypercalls
/sys/kernel/debug/kvm/halt_wakeup
/sys/kernel/debug/kvm/halt_exits
/sys/kernel/debug/kvm/nmi_window
/sys/kernel/debug/kvm/irq_window
/sys/kernel/debug/kvm/signal_exits
/sys/kernel/debug/kvm/mmio_exits
/sys/kernel/debug/kvm/io_exits
/sys/kernel/debug/kvm/exits
/sys/kernel/debug/kvm/invlpg
/sys/kernel/debug/kvm/tlb_flush
/sys/kernel/debug/kvm/pf_guest
/sys/kernel/debug/kvm/pf_fixed


 -----------------------------------------------------------------------
/sys/kernel/debug/kvm/                                      : <dir>
/sys/kernel/debug/kvm/largepages                            :           25
/sys/kernel/debug/kvm/remote_tlb_flush                      :            0
/sys/kernel/debug/kvm/mmu_unsync                            :            0
/sys/kernel/debug/kvm/mmu_cache_miss                        :          235
/sys/kernel/debug/kvm/mmu_recycled                          :            0
/sys/kernel/debug/kvm/mmu_flooded                           :            0
/sys/kernel/debug/kvm/mmu_pde_zapped                        :            0
/sys/kernel/debug/kvm/mmu_pte_updated                       :            0
/sys/kernel/debug/kvm/mmu_pte_write                         :            0
/sys/kernel/debug/kvm/mmu_shadow_zapped                     :          215
/sys/kernel/debug/kvm/nmi_injections                        :            0
/sys/kernel/debug/kvm/irq_injections                        :         1264 (      1 K)
/sys/kernel/debug/kvm/insn_emulation_fail                   :            0
/sys/kernel/debug/kvm/insn_emulation                        :       129774 (    126 K)
```

```
/sys/kernel/debug/kvm/fpu_reload                              :        488
/sys/kernel/debug/kvm/efer_reload                             :          0
/sys/kernel/debug/kvm/host_state_reload                       :     115851 (    113 K)
/sys/kernel/debug/kvm/irq_exits                               :        978
/sys/kernel/debug/kvm/request_irq                             :          0
/sys/kernel/debug/kvm/hypercalls                              :          0
/sys/kernel/debug/kvm/halt_wakeup                             :        356
/sys/kernel/debug/kvm/halt_exits                              :        565
/sys/kernel/debug/kvm/nmi_window                              :          0
/sys/kernel/debug/kvm/irq_window                              :        204
/sys/kernel/debug/kvm/signal_exits                            :          0
/sys/kernel/debug/kvm/mmio_exits                              :         19
/sys/kernel/debug/kvm/io_exits                                :     105261 (    102 K)
/sys/kernel/debug/kvm/exits                                   :     291774 (    284 K)
/sys/kernel/debug/kvm/invlpg                                  :          0
/sys/kernel/debug/kvm/tlb_flush                               :        100
/sys/kernel/debug/kvm/pf_guest                                :          1
/sys/kernel/debug/kvm/pf_fixed                                :      36073 (     35 K)
#
```

------------------------------------>>

```
...
} << end func kvm_init() >>

...
```

The various operation vector structures:

Source: Documentation/virtual/kvm/api.txt

The kvm API is (built around) a set of ioctls that are issued to control various aspects of a virtual machine. The ioctls belong to three classes:

- System ioctls: These query and set global attributes which affect the whole kvm subsystem. In addition a system ioctl is used to create virtual machines

```
static struct file_operations kvm_chardev_ops = {
    .unlocked_ioctl = kvm_dev_ioctl,
    .compat_ioctl   = kvm_dev_ioctl,
    .llseek     = noop_llseek,
};

static struct miscdevice kvm_dev = {
    KVM_MINOR,
    "kvm",        << Becomes /dev/kvm (regd as a “misc” device);
                     So an ioctl(2) on this becomes the kvm_dev_ioctl() entry point
                     (see code below) >>
    &kvm_chardev_ops,
};
```

```
...
```

- VM ioctls: These query and set attributes that affect an entire virtual machine, for example memory layout. In addition a VM ioctl is used to create virtual cpus (vcpus).

  Only run VM ioctls from the same process (address space) that was used to create the VM.

```
static struct file_operations kvm_vm_fops = {
    .release        = kvm_vm_release,
    .unlocked_ioctl = kvm_vm_ioctl,
#ifdef CONFIG_KVM_COMPAT
    .compat_ioctl   = kvm_vm_compat_ioctl,
#endif
    .llseek     = noop_llseek,
};

...
```

- vcpu ioctls: These query and set attributes that control the operation of a single virtual cpu.

  Only run vcpu ioctls from the same thread that was used to create the vcpu.

```
static struct file_operations kvm_vcpu_fops = {
    .release        = kvm_vcpu_release,
    .unlocked_ioctl = kvm_vcpu_ioctl,
#ifdef CONFIG_KVM_COMPAT
    .compat_ioctl   = kvm_vcpu_compat_ioctl,
#endif
    .mmap           = kvm_vcpu_mmap,
    .llseek     = noop_llseek,
};

...
```

<< --

Q. How exactly is kvm 'tdp' (two dimensional paging, i.e., hardware mmu nested paging support), enabled?

A. Upon guest launch, Qemu invokes an ioctl(2) syscall to create a VCPU:

```
ret = kvm_ioctl(s, KVM_CREATE_VM, type);
```

Resulting kvm code path (call graph):

```
kvm_vm_ioctl
 kvm_vm_ioctl_create_vcpu
  kvm_arch_vcpu_setup
    kvm_vcpu_reset
      kvm_x86_ops->vcpu_reset(vcpu);
        vmx_vcpu_reset
```

```
          << here the vcpu is setup; segment registers, VMCS[2] regs, msr's, apicv,
control regs (cr0, cr4), etc etc >>
          vmx_x86_ops->set_cr0
            vmx_set_cr0
              enter_rmode
                kvm_mmu_reset_context
                  init_kvm_mmu
                    init_kvm_tdp_mmu(vcpu);
                         << here's where nested paging mmu setup is done! >>

---------------------------------------------------------- >>
```

How does QEMU hook into the kvm hypervisor?

- Synchronous
  - ioctl(2) 's from Qemu
- Asynchronous
  - Async PF (page fault) handler
  - Memory notification mechanism (mm_notifier)
  - Interrupt recognition and processing
- IO

The main kvm ioctl path

```
virt/kvm/kvm_main.c
static long kvm_dev_ioctl(struct file *filp,                    << 'system' icotl >>
              unsigned int ioctl, unsigned long arg)
{
    long r = -EINVAL;

    switch (ioctl) {
    case KVM_GET_API_VERSION:
        if (arg)
            goto out;
        r = KVM_API_VERSION;
        break;
    case KVM_CREATE_VM:
        r = kvm_dev_ioctl_create_vm(arg);
        break;
    case KVM_CHECK_EXTENSION:
        r = kvm_vm_ioctl_check_extension_generic(NULL, arg);
        break;
    case KVM_GET_VCPU_MMAP_SIZE:
        if (arg)
            goto out;
        r = PAGE_SIZE;     /* struct kvm_run */
#ifdef CONFIG_X86
        r += PAGE_SIZE;    /* pio data page */
#endif
#ifdef KVM_COALESCED_MMIO_PAGE_OFFSET
        r += PAGE_SIZE;    /* coalesced mmio ring page */
```

2 VMCS: Virtual Machine Control Structures. See Intel Manual Vol 3 Ch 24 for complete details.

```
#endif
        break;
    case KVM_TRACE_ENABLE:
    case KVM_TRACE_PAUSE:
    case KVM_TRACE_DISABLE:
        r = -EOPNOTSUPP;
        break;
    default:
        return kvm_arch_dev_ioctl(filp, ioctl, arg);
    }
out:
    return r;
}
```

<<------------------------ *QEMU [userland]*
Here, repeated for clarity, is the Qemu code-flow that invokes the KVM_CREATE_VM ioctl !

kvm-all.c

```
static int kvm_init(MachineState *ms)
{
    MachineClass *mc = MACHINE_GET_CLASS(ms);
...
 KVMState *s;
...
  s->fd = qemu_open("/dev/kvm", O_RDWR);

...
<< Validity checks, etc >>
...

do {                             << Qemu: Create the VM ! This latches into the KVM kernel
                                      module's device ioctl code path >>
        ret = kvm_ioctl(s, KVM_CREATE_VM, type);
    } while (ret == -EINTR);
...
```

-------------------------->>

Back to kvm

virt/kvm/kvm_main.c

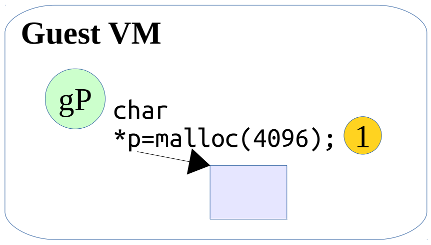
```
static int kvm_dev_ioctl_create_vm(unsigned long type)
{
    int r;
    struct kvm *kvm;

    kvm = kvm_create_vm(type);
    if (IS_ERR(kvm))
        return PTR_ERR(kvm);
#ifdef KVM_COALESCED_MMIO_PAGE_OFFSET
    r = kvm_coalesced_mmio_init(kvm);
    if (r < 0) {
        kvm_put_kvm(kvm);
        return r;
    }
#endif
    r = anon_inode_getfd("kvm-vm", &kvm_vm_fops, kvm, O_RDWR | O_CLOEXEC);
    if (r < 0)
```

```
        kvm_put_kvm(kvm);

    return r;
}


virt/kvm/kvm_main.c
static struct kvm *kvm_create_vm(unsigned long type)
{
    int r, i;
    struct kvm *kvm = kvm_arch_alloc_vm();

...
    kvm_arch_init_vm(kvm, type);
...
    hardware_enable_all();
...
    kvm_init_memslots_id(kvm);
...

    spin_lock_init(&kvm->mmu_lock);
    kvm->mm = current->mm;         << process context 'current' will usually be Qemu >>
    atomic_inc(&kvm->mm->mm_count);
    kvm_eventfd_init(kvm);
    mutex_init(&kvm->lock);
    mutex_init(&kvm->irq_lock);
    mutex_init(&kvm->slots_lock);
    atomic_set(&kvm->users_count, 1);
    INIT_LIST_HEAD(&kvm->devices);

    r = kvm_init_mmu_notifier(kvm);

...

    spin_lock(&kvm_lock);
    list_add(&kvm->vm_list, &vm_list);
    spin_unlock(&kvm_lock);

    return kvm;
}
```

<<
Recall (covered earlier in this module):

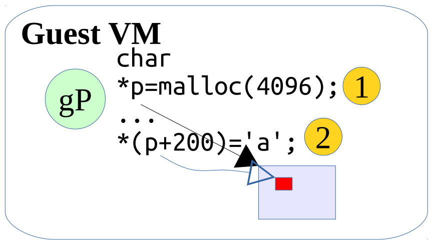
Fault-in path

1. QEMU calls malloc() and allocates virtual space for the page, but no backing physical page

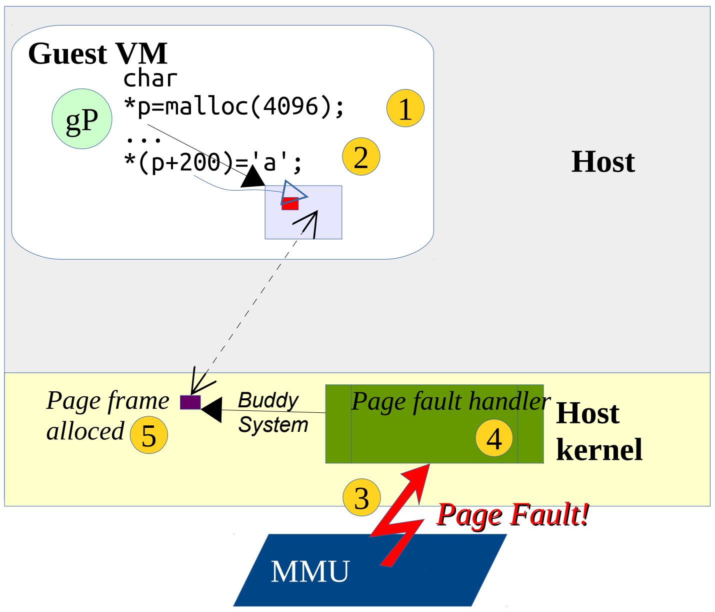


gP : guest process

2. The guest process touches what it thinks is a physical address, but this traps into the host since the memory is unallocated



3. The host kernel sees a page fault, calls do_page_fault() in the area that was malloc()'d, and if all goes well, allocates some memory to back it.



4. The host kernel creates a pte_t to connect the malloc()'d virtual address to a host physical

address, makes rmap entries, puts it on the LRU, invalidates the TLB if requied (INVLPG instruction issued!), etc...

5. The appropriate MMU-notifier hook function (~~mmu_notifier change_pte()??~~) (*virt/kvm/kvm_main.c:kvm_mmu_notifier_change_pte()*) is called, which allows KVM to create an NPT/EPT entry for the new page. (and an spte entry?? *(only if shadow paging is enabled, i.e., if tdp_enabled == False)*)

6. Host returns from page fault, guest execution resumes.

>>

```
...

static const struct mmu_notifier_ops kvm_mmu_notifier_ops = {
    .invalidate_page     = kvm_mmu_notifier_invalidate_page,
    .invalidate_range_start = kvm_mmu_notifier_invalidate_range_start,
    .invalidate_range_end   = kvm_mmu_notifier_invalidate_range_end,
    .clear_flush_young  = kvm_mmu_notifier_clear_flush_young,
    .test_young      = kvm_mmu_notifier_test_young,
    .change_pte      = kvm_mmu_notifier_change_pte,
    .release         = kvm_mmu_notifier_release,
};

static int kvm_init_mmu_notifier(struct kvm *kvm)
{
    kvm->mmu_notifier.ops = &kvm_mmu_notifier_ops;
    return mmu_notifier_register(&kvm->mmu_notifier, current->mm);
}

...
virt/kvm/kvm_main.c
static void kvm_mmu_notifier_change_pte(struct mmu_notifier *mn,
                    struct mm_struct *mm,
                    unsigned long address,
                    pte_t pte)
{
    struct kvm *kvm = mmu_notifier_to_kvm(mn);
    int idx;

    idx = srcu_read_lock(&kvm->srcu);
    spin_lock(&kvm->mmu_lock);
    kvm->mmu_notifier_seq++;
    kvm_set_spte_hva(kvm, address, pte); << kvm sets up the shadow pte; meat of the
                                            code is in kvm_handle_hva_range() >>
    spin_unlock(&kvm->mmu_lock);
    srcu_read_unlock(&kvm->srcu, idx);
}
...
```

Qemu Processing Flow within the host hypervisor (kvm) Event Loop(?)

To see first-hand (at the code level) how Qemu/KVM works, we *ftrace* Qemu (implies a kernel-level trace of the host hypervisor. See the ftrace script provided to see exactly with what ftrace options are used, etc).

See the annotated ftrace output snippets below:

```
...
 2) qemu-sy-25172  |  ....             |  SyS_ioctl() {  << Qemu has issued an ioctl(<fd>, KVM_RUN, ...);
                                                          >>
 2) qemu-sy-25172  |  ....             |    __fdget() {
 2) qemu-sy-25172  |  ....             |      __fget_light() {
 2) qemu-sy-25172  |  ....  0.055 us   |        __fget();
 2) qemu-sy-25172  |  ....  0.328 us   |      }
 2) qemu-sy-25172  |  ....  0.602 us   |    }
 2) qemu-sy-25172  |  ....             |    security_file_ioctl() {
 2) qemu-sy-25172  |  ....  0.033 us   |      cap_file_ioctl();
 2) qemu-sy-25172  |  ....  0.308 us   |    }
 2) qemu-sy-25172  |  ....             |    do_vfs_ioctl() {
 2) qemu-sy-25172  |  ....             |      kvm_vcpu_ioctl [kvm]() {   << "[kvm]" => in LKM kvm.ko >>
 2) qemu-sy-25172  |  ....             |        vcpu_load [kvm]() {
 2) qemu-sy-25172  |  ....             |          mutex_lock_killable() {
 2) qemu-sy-25172  |  ....  0.032 us   |            _cond_resched();
 2) qemu-sy-25172  |  ....  0.341 us   |          }
 2) qemu-sy-25172  |  ....  0.040 us   |          preempt_notifier_register();
 2) qemu-sy-25172  |  ....             |          kvm_arch_vcpu_load [kvm]() {
 2) qemu-sy-25172  |  ....  0.055 us   |            vmx_vcpu_load [kvm_intel]();
 2) qemu-sy-25172  |  ....  0.365 us   |          }
 2) qemu-sy-25172  |  ....  1.568 us   |        }
 2) qemu-sy-25172  |  ....             |        kvm_arch_vcpu_ioctl_run [kvm]() { << corresponds to
                                                                                 KVM_RUN >>
 2) qemu-sy-25172  |  ....             |          sigprocmask() {
 2) qemu-sy-25172  |  ....             |            __set_current_blocked() {
 2) qemu-sy-25172  |  ....  0.035 us   |              _raw_spin_lock_irq();
 2) qemu-sy-25172  |  d...             |              __set_task_blocked() {
 2) qemu-sy-25172  |  d...             |                recalc_sigpending() {
 2) qemu-sy-25172  |  d...  0.034 us   |                  recalc_sigpending_tsk();
 2) qemu-sy-25172  |  d...  0.316 us   |                }
 2) qemu-sy-25172  |  d...  0.612 us   |              }
 2) qemu-sy-25172  |  ....  1.195 us   |            }
 2) qemu-sy-25172  |  ....  1.496 us   |          }
 2) qemu-sy-25172  |  ....  0.059 us   |          __srcu_read_lock();
 2) qemu-sy-25172  |  ....             |          vcpu_enter_guest [kvm]() { << "enter" guest mode >>

<<
vcpu_enter_guest()
  < ... checks & processes guest request(s) ... >
  kvm_mmu_reload
  kvm_x86_ops->prepare_guest_switch(vcpu);   << virt fn >>
    vmx_save_host_state
    kvm_load_guest_xcr0(vcpu);
    vcpu->mode = IN_GUEST_MODE;
   <... if need to exit guest mode or signal or schedule pending, then cancel ...>
   kvm_guest_enter();
      guest_enter(void);
        current->flags |= PF_VCPU;   << Qemu (or whatever) marked as "I'm a virtual CPU" >>

   kvm_x86_ops->run(vcpu);    << virt fn >>
```

```
     vmx_vcpu_run();     << This is where the VM Entry, from host-to-guest (Ring 0 Root -> Ring 0 Non-Root
mode), and, then back again (VM Exit), is done! Hand-coded assembly implements the actual context switch!
>>
arch/x86/kvm/vmx.c
        ...
         asm(
        /* Store host registers */
        "push %%" _ASM_DX "; push %%" _ASM_BP ";"
        "push %%" _ASM_CX " \n\t" /* placeholder for guest rcx */
        "push %%" _ASM_CX " \n\t"
       ...
       /* Load guest registers. .. */
       ...
       /* Enter guest mode */
        "jne 1f \n\t"
        __ex(ASM_VMX_VMLAUNCH) "\n\t"     << From Intel's manual Vol 3:
"Using VM entries, a VMM can then enter guests into virtual machines (one at a time). The VMM effects a
VM entry using instructions VMLAUNCH and VMRESUME; it regains control using VM exits." >>
        "jmp 2f \n\t"
        "1: " __ex(ASM_VMX_VMRESUME) "\n\t"
        "2: "
        /* Save guest registers, load host registers, keep flags */
        ...
        );

   kvm_x86_ops->handle_external_intr(vcpu);    << virt fn >>
     vmx_handle_external_intr();
   kvm_guest_exit();
   kvm_x86_ops->handle_exit(vcpu);   << virt fn >>
       vmx_handle_exit();
>>


 2) qemu-sy-25172  |  ....  0.049 us    |            kvm_read_guest_cached [kvm]();
 2) qemu-sy-25172  |  ....              |            kvm_write_guest_cached [kvm]() {
 2) qemu-sy-25172  |  ....  0.034 us    |              mark_page_dirty_in_slot.isra.19 [kvm]();
 2) qemu-sy-25172  |  ....  0.319 us    |            }
 2) qemu-sy-25172  |  ....              |            vmx_save_host_state [kvm_intel]() {
 2) qemu-sy-25172  |  ....  0.033 us    |              vmcs_writel [kvm_intel](); << VMCS: Virtual Machine
Control Structures. See Intel Manual Vol 3 Ch 24 for complete details. >>
 2) qemu-sy-25172  |  ....  0.034 us    |              vmcs_writel [kvm_intel]();
 2) qemu-sy-25172  |  ....  0.033 us    |              vmcs_writel [kvm_intel]();
 2) qemu-sy-25172  |  ....  0.033 us    |              vmcs_writel [kvm_intel]();
 2) qemu-sy-25172  |  ....  0.036 us    |              kvm_set_shared_msr [kvm]();
 2) qemu-sy-25172  |  ....              |              kvm_set_shared_msr [kvm]() {
 2) qemu-sy-25172  |  ....  0.036 us    |                user_return_notifier_register();
 2) qemu-sy-25172  |  ....  0.379 us    |              }
 2) qemu-sy-25172  |  ....  0.087 us    |              kvm_set_shared_msr [kvm]();
 2) qemu-sy-25172  |  ....  0.049 us    |              kvm_set_shared_msr [kvm]();
 2) qemu-sy-25172  |  ....  3.093 us    |            }
 2) qemu-sy-25172  |  ....  0.054 us    |            __srcu_read_unlock();
 2) qemu-sy-25172  |  d...  0.037 us    |            rcu_note_context_switch();
 2) qemu-sy-25172  |  d...              |            vmx_vcpu_run [kvm_intel]() { << sw again to guest
                                                                        code >>
 2) qemu-sy-25172  |  d...  0.033 us    |              vmcs_writel [kvm_intel]();
 2) qemu-sy-25172  |  d...              |              perf_guest_get_msrs() {
 2) qemu-sy-25172  |  d...  0.035 us    |                intel_guest_get_msrs();
 2) qemu-sy-25172  |  d...  0.346 us    |              }
 2) qemu-sy-25172  |  d...  0.039 us    |              clear_atomic_switch_msr [kvm_intel]();
 2) qemu-sy-25172  |  d...  0.036 us    |              clear_atomic_switch_msr [kvm_intel]();
 2) qemu-sy-25172  |  d...  2.105 us    |            }
 2) qemu-sy-25172  |  d...  0.032 us    |            vmx_read_l1_tsc [kvm_intel]();
 2) qemu-sy-25172  |  d...  0.034 us    |            vmx_handle_external_intr [kvm_intel]();
 2) qemu-sy-25172  |  ....  0.074 us    |            __srcu_read_lock();
 2) qemu-sy-25172  |  ....              |            vmx_handle_exit [kvm_intel]() {
 2) qemu-sy-25172  |  ....              |              handle_io [kvm_intel]() {
 2) qemu-sy-25172  |  ....              |                skip_emulated_instruction [kvm_intel]() {
 2) qemu-sy-25172  |  ....  0.035 us    |                  vmx_cache_reg [kvm_intel]();
 2) qemu-sy-25172  |  ....  0.034 us    |                  vmx_set_interrupt_shadow [kvm_intel]();
 2) qemu-sy-25172  |  ....  0.619 us    |                }
 2) qemu-sy-25172  |  ....              |                kvm_fast_pio_out [kvm]() {
 2) qemu-sy-25172  |  ....              |                  emulator_pio_out_emulated [kvm]() {
 2) qemu-sy-25172  |  ....              |                    emulator_pio_in_out [kvm]() {
 2) qemu-sy-25172  |  ....              |                      kvm_io_bus_write [kvm]() {
 2) qemu-sy-25172  |  ....              |                        __kvm_io_bus_write [kvm]() {
 2) qemu-sy-25172  |  ....              |                          kvm_io_bus_get_first_dev [kvm]() {
 2) qemu-sy-25172  |  ....  0.036 us    |                            kvm_io_bus_sort_cmp [kvm]();
 2) qemu-sy-25172  |  ....  0.036 us    |                            kvm_io_bus_sort_cmp [kvm]();
 2) qemu-sy-25172  |  ....  0.034 us    |                            kvm_io_bus_sort_cmp [kvm]();
 2) qemu-sy-25172  |  ....  0.036 us    |                            kvm_io_bus_sort_cmp [kvm]();
 2) qemu-sy-25172  |  ....  0.045 us    |                            kvm_io_bus_sort_cmp [kvm]();
 2) qemu-sy-25172  |  ....  0.035 us    |                            kvm_io_bus_sort_cmp [kvm]();
 2) qemu-sy-25172  |  ....  1.681 us    |                          }
 2) qemu-sy-25172  |  ....  1.957 us    |                        }
 2) qemu-sy-25172  |  ....  2.236 us    |                      }
 2) qemu-sy-25172  |  ....  2.519 us    |                    }
 2) qemu-sy-25172  |  ....  2.815 us    |                  }
 2) qemu-sy-25172  |  ....  3.098 us    |                }
 2) qemu-sy-25172  |  ....  4.232 us    |              }
 2) qemu-sy-25172  |  ....  4.519 us    |            }
 2) qemu-sy-25172  |  ....+ 12.995 us   |          }
 2) qemu-sy-25172  |  ....  0.052 us    |          __srcu_read_unlock();
 2) qemu-sy-25172  |  ....  0.033 us    |          vmx_get_rflags [kvm_intel]();
 2) qemu-sy-25172  |  ....  0.034 us    |          kvm_lapic_get_cr8 [kvm]();
 2) qemu-sy-25172  |  ....              |          sigprocmask() {
 2) qemu-sy-25172  |  ....              |            __set_current_blocked() {
 2) qemu-sy-25172  |  ....  0.035 us    |              _raw_spin_lock_irq();
 2) qemu-sy-25172  |  d...              |              __set_task_blocked() {
 2) qemu-sy-25172  |  d...              |                recalc_sigpending() {
 2) qemu-sy-25172  |  d...  0.034 us    |                  recalc_sigpending_tsk();
 2) qemu-sy-25172  |  d...  0.342 us    |                }
 2) qemu-sy-25172  |  d...  0.615 us    |              }
 2) qemu-sy-25172  |  ....  1.167 us    |            }
 2) qemu-sy-25172  |  ....  1.440 us    |          }
 2) qemu-sy-25172  |  ....+ 18.025 us   |        }
 2) qemu-sy-25172  |  ....              |        vcpu_put [kvm]() {
 2) qemu-sy-25172  |  ....              |          kvm_arch_vcpu_put [kvm]() {
 2) qemu-sy-25172  |  ....              |            vmx_vcpu_put [kvm_intel]() {
 2) qemu-sy-25172  |  ....  0.248 us    |              __vmx_load_host_state.part.46 [kvm_intel]();
 2) qemu-sy-25172  |  ....  0.556 us    |            }
 2) qemu-sy-25172  |  ....  0.034 us    |            kvm_put_guest_fpu [kvm]();
 2) qemu-sy-25172  |  ....  1.149 us    |          }
 2) qemu-sy-25172  |  ....  0.035 us    |          preempt_notifier_unregister();
 2) qemu-sy-25172  |  ....  0.039 us    |          mutex_unlock();
 2) qemu-sy-25172  |  ....  2.007 us    |        }
 2) qemu-sy-25172  |  ....  0.036 us    |        kfree();
 2) qemu-sy-25172  |  ....  0.036 us    |        kfree();
 2) qemu-sy-25172  |  ....+ 22.979 us   |      }
 2) qemu-sy-25172  |  ....+ 23.263 us   |    }
 2) qemu-sy-25172  |  ....  0.037 us    |    fput();
 2) qemu-sy-25172  |  ....+ 25.541 us   |  }        << SyS_ioctl() above ends >>
...
```

virt/kvm/kvm_main.c

```
static long kvm_vcpu_ioctl(struct file *filp,
               unsigned int ioctl, unsigned long arg)
{
    struct kvm_vcpu *vcpu = filp->private_data;
...

    vcpu_load(vcpu);

...
```

Qemu issues the ioctl(..KVM_RUN..); this tells the hypervisor to allow the guest VM code to run unaffected on the host cpu. If for any reason the host must intervene, the guest code will cause a trap into the hypervisor and we will “enter” kvm, process the trap and “exit” kvm, back into the guest run state.

```
...
switch (ioctl) {
    case KVM_RUN:
        r = -EINVAL;
        if (arg)
            goto out;
        if (unlikely(vcpu->pid != current->pids[PIDTYPE_PID].pid)) {
            /* The thread running this VCPU changed. */
            struct pid *oldpid = vcpu->pid;
            struct pid *newpid = get_task_pid(current, PIDTYPE_PID);

            rcu_assign_pointer(vcpu->pid, newpid);
            if (oldpid)
                synchronize_rcu();
            put_pid(oldpid);
        }
        r = kvm_arch_vcpu_ioctl_run(vcpu, vcpu->run);
        trace_kvm_userspace_exit(vcpu->run->exit_reason, r);
        break;
    case KVM_GET_REGS: {
        struct kvm_regs *kvm_regs;

        r = -ENOMEM;
        kvm_regs = kzalloc(sizeof(struct kvm_regs), GFP_KERNEL);
        if (!kvm_regs)
            goto out;
        r = kvm_arch_vcpu_ioctl_get_regs(vcpu, kvm_regs);
...
```

Suggested Mini Project
- Sync time in L0 and L1 (host & guest) - Ftrace host (kvm) with absolute timestamps - Simultaneously trace guest execution with absolute timestamps - Compare and interpret trace data.

To visualize the trace data as a histogram:

```
# trace-cmd hist
  %2.93  (32506) <...>                    seq_printf #25921
         |
         --- seq_printf
            |
            |--%78.39-- render_sigset_t  # 20320
            |          proc_pid_status
            |          proc_single_show

...
                                                    << This routine ran 22,505 times >>
  %2.55  (31854) qemu-system-x86                      vmcs_writel #22505
         |
         --- vmcs_writel
            |
            |--%36.81-- vmx_save_host_state  # 8284
            |            |
            |            |--%99.95-- vcpu_enter_guest  # 8280
            |            |            |
            |            |            |--%99.76-- kvm_arch_vcpu_ioctl_run  # 8260
            |            |            |          kvm_vcpu_ioctl
            |            |            |          do_vfs_ioctl
            |            |            |          SyS_ioctl
            |            |            |
            |            |            |--%0.19-- sigprocmask  # 16
            |            |            |          kvm_vcpu_ioctl
            |            |            |          do_vfs_ioctl
            |            |            |          SyS_ioctl
            |            |            |
            |            |            |--%0.05-- rcu_irq_enter  # 4
            |            |                       irq_enter
            |            |                       __irqentry_text_start
            |            |                       finish_task_switch
            |            |
            |            |--%0.05-- kvm_read_guest_cached  # 4
            |                       kvm_arch_vcpu_ioctl_run
            |                       kvm_vcpu_ioctl
            |                       do_vfs_ioctl
            |                       SyS_ioctl
            |
            |--%16.94-- vmx_vcpu_run  # 3813
            |          vcpu_enter_guest
            |            |
            |            |--%99.87-- kvm_arch_vcpu_ioctl_run  # 3808
            |            |          kvm_vcpu_ioctl
            |            |          do_vfs_ioctl
            |            |          SyS_ioctl
            |            |
            |            |--%0.08-- rcu_irq_enter  # 3
            |            |          irq_enter
            |            |          __irqentry_text_start
            |            |          finish_task_switch
            |            |
            |            |--%0.05-- sigprocmask  # 2
            |                       kvm_vcpu_ioctl
            |                       do_vfs_ioctl
            |                       SyS_ioctl
            |
            |--%7.49-- vmx_fpu_activate.part.60  # 1685
            |             |
            |             |--%99.94-- handle_exception  # 1684
            |             |          vmx_handle_exit
            |             |          vcpu_enter_guest
            |             |          kvm_arch_vcpu_ioctl_run
            |             |          kvm_vcpu_ioctl
            |             |          do_vfs_ioctl
            |             |          SyS_ioctl

...
```

Appendix A :: Misc / FYI Information on KVM / Qemu / Virtualization

- VM Hierarchy
```
 L0   (host)
   |_  L1  (guest level 1)
         |_  L2  (nested guest level 2)
```
- One guest VM per cpu core (?)
- One nested guest possible - guest VM (L2) within a guest VM (L1); maintained in the vmcs12 and nested_vmx structures (arch/x86/kvm/vmx.c)
- **Communication** between a guest VM and other guest VM's or a host:
  - Efficient shared memory message passing for inter-VM communications

KVM Fault Handling

The "async_page_fault" label becomes the do_async_page_fault function in both IA-32 and Intel-64 architectures.

./arch/x86/kernel/entry_64.S

```
#ifdef CONFIG_KVM_GUEST
idtentry async_page_fault do_async_page_fault has_error_code=1
#endif
```

arch/x86/kernel/entry_32.S

```
#ifdef CONFIG_KVM_GUEST
ENTRY(async_page_fault)
    RING0_EC_FRAME
    ASM_CLAC
    pushl_cfi $do_async_page_fault
    jmp error_code
    CFI_ENDPROC
END(async_page_fault)
#endif
```

arch/x86/kernel/kvm.c

```
dotraplinkage void
do_async_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    enum ctx_state prev_state;

    switch (kvm_read_and_reset_pf_reason()) {
```

```
    default:
        trace_do_page_fault(regs, error_code); << if here, it's an OS fault; reflect it
                                                back to the OS; this code invokes __do_page_fault() -
                                                the “normal” page fault handler>>
        break;
    case KVM_PV_REASON_PAGE_NOT_PRESENT:
        /* page is swapped out by the host. */
        prev_state = exception_enter();
        exit_idle();
        kvm_async_pf_task_wait((u32)read_cr2());
        exception_exit(prev_state);
        break;
    case KVM_PV_REASON_PAGE_READY:
        rcu_irq_enter();
        exit_idle();
        kvm_async_pf_task_wake((u32)read_cr2());
        rcu_irq_exit();
        break;
    }
}
NOKPROBE_SYMBOL(do_async_page_fault);
```

Resources

http://wiki.qemu.org/Manual

http://qemu.weilnetz.de/qemu-tech.html

http://daehee87.tistory.com/328 <-- Xlate to English..

Virtualization Techniques

- De-privileging
  Guest does not run in OS privilege mode on the cpu; typically, it runs as a “normal” user application (at Ring 3). This allows the hypervisor to control guests and disallows guests from taking over the system.
- Trapping into the host (see discussion below)
- Binary Translation (see discussion below)
- Dynamic Translation (see discussion below)
-

Source

Trapping Into the Host

The grandfathers of virtualization, such as the IBM S/370, used a very robust system to allow the hypervisor to control the virtual machines. Every privileged instruction by a virtual machine caused a "trap", an error, as it was trying to execute a "resource management" instruction while running in a less privileged ring. The VMM intercepts all those traps and emulates the instruction, without jeopardizing the integrity of the other guests. In order to improve performance, the developers of the guest OS and VMM (both at IBM) tried to minimize the number of traps and reduce the time required to take care of the various traps.

This kind of virtualization was not possible on x86 as the 32/64-bit Intel ISA does not trap every incident that should lead to VMM intervention.

One example is the POPF instruction that disables and enables interrupts. The problem is that if this instruction is executed by a guest OS in ring 1, an x86 CPU does not make a fuss about it, but simply ignores it. The result is that if a guest OS is trying to disable an interrupt, the interrupt is not disabled at all, and the VMM has no way of knowing that this is happening. As always, the good old x86 ISA is a bit chaotic: it has 17 of these "non-interceptable, cloaked for the VMM" instructions [1]. The conclusion is that x86 cannot be virtualized the way that the old mainframes were virtualized.
Incidentally, the PowerPC and Alpha ISA's are clean enough to be virtualized in the classic manner.

Binary Translation

VMware didn't wait for Intel or AMD to solve the "x86 stealth instructions" problem and launched their solution at the end of the previous century (1999). To uncloak the stealthy x86 instructions, VMware used Binary translation (unfortunately, a Tachyon detection grid proved too expensive) . VMware's Binary Translation is a lot lighter than the binary translation technology that the Intel Itanium (x86 to IA64), Transmeta (x86 to VLIW), Digital FX!32 (Alpha to x86), or Rosetta software use. It doesn't have to translate from one Instruction Set Architecture (ISA) to another but it is based on an x86 to x86 translator. In fact, in some cases it just makes an exact copy of the original instruction.

VMware translates the binary code that the kernel of a guest OS wants to execute on the fly and stores the adapted x86 code in a Translator Cache (TC). User applications will not be touched by VMware's Binary Translator (BT) as it knows/assumes that user code is safe. User mode applications are executed directly as if they were running natively.

[[...]]

It is the kernel code that has go through the "x86 to slightly longer x86" code translation. You could say that the kernel of the guest OS is no longer running. The kernel code in the memory is nothing more than an input for the BT; it is the BT translated kernel that will run in ring 1.

In many cases, the translated kernel code will be an exact copy. However, there are several cases where BT must make the "translated" kernel code a bit longer than the original code. If the kernel of the guest OS has to run a privileged instruction, the BT will change this kind of code into "safer" user mode code. If the kernel needs to get control of the physical hardware, the BT will replace that binary code with code that manipulates the virtual hardware.

Binary translation is all about scanning the code that the kernel of the guest OS should execute at a certain moment in time and replacing it with something safe (virtualized) on the fly. With "safe", we mean safe for the other guest OSes and the VMM. VMware also keeps the overhead of the translation as low as possible. The BT does not optimize the binary instruction stream, and an instruction stream that has been translated is kept in a cache. In case of a loop, this means that the translation is done only once.
[[...]]
It is clear that replacing code with "safer" code is a lot less costly than letting privileged instructions result in traps and then handling those traps afterwards. Nevertheless, that doesn't mean that the overhead of this kind of virtualization is always low. The "Translator overhead" is rather low, and its impact gets lower and lower over time, courtesy of the Translator cache. However, BT cannot completely crack several hard nuts:

1. System Calls
2. Accesses to chipset and I/O, interrupts and DMA
3. Memory management
4. "Weird and complex code" (Self-modifying, indirect control flows, etc.)

Especially the first three are interesting. The last one is hard in an OS running in "native mode" too, so it is only normal that this doesn't get any better if you run more than one OS.

<<
Qemu Internals: ***Source***

[[...]]

2.2 Portable dynamic translation

QEMU is a dynamic translator. When it first encounters a piece of code, it converts it to the host instruction set. Usually dynamic translators are very complicated and highly CPU dependent. QEMU uses some tricks which make it relatively easily portable and simple while achieving good performances.

After the release of version 0.9.1, QEMU switched to a new method of generating code, Tiny Code Generator or TCG. TCG relaxes the dependency on the exact version of the compiler used. The basic idea is to split every target instruction into a couple of RISC-like TCG ops (see target-i386/translate.c). Some optimizations can be performed at this stage, including liveness analysis and trivial constant expression evaluation. TCG ops are then implemented in the host CPU back end, also known as TCG target (see tcg/i386/tcg-target.c). For more information, please take a look at tcg/README.

[[...]]

2.5 Translation cache

A 32 MByte cache holds the most recently used translations. For simplicity, it is completely flushed when it is full. A translation unit contains just a single basic block (a block of x86 instructions terminated by a jump or by a virtual CPU state change which the translator cannot deduce statically).

[...]

2.8 Exception support

longjmp() is used when an exception such as division by zero is encountered.

The host SIGSEGV and SIGBUS signal handlers are used to get invalid memory accesses. The simulated program counter is found by retranslating the corresponding basic block and by looking where the host program counter was at the exception point.

The virtual CPU cannot retrieve the exact EFLAGS register because in some cases it is not computed because of condition code optimisations. It is not a big concern because the emulated code can still be restarted in any cases.

2.9 MMU emulation

For system emulation QEMU supports a soft MMU. In that mode, the MMU virtual to physical address translation is done at every memory access. QEMU uses an address translation cache to speed up the translation.

In order to avoid flushing the translated code each time the MMU mappings change, QEMU uses a physically indexed translation cache. It means that each basic block is indexed with its physical address.

When MMU mappings change, only the chaining of the basic blocks is reset (i.e. a basic block can no longer jump directly to another one).

2.10 Device emulation

Systems emulated by QEMU are organized by boards. At initialization phase, each board instantiates a number of CPUs, devices, RAM and ROM. Each device in turn can assign I/O ports or memory areas (for MMIO) to its handlers. When the emulation starts, an access to the ports or

MMIO memory areas assigned to the device causes the corresponding handler to be called.

RAM and ROM are handled more optimally, only the offset to the host memory needs to be added to the guest address.

The video RAM of VGA and other display cards is special: it can be read or written directly like RAM, but write accesses cause the memory to be marked with VGA_DIRTY flag as well.

QEMU supports some device classes like serial and parallel ports, USB, drives and network devices, by providing APIs for easier connection to the generic, higher level implementations.

The API hides the implementation details from the devices, like native device use or advanced block device formats like QCOW.

Usually the devices implement a reset method and register support for saving and loading of the device state. The devices can also use timers, especially together with the use of bottom halves (BHs).

2.11 Hardware interrupts

In order to be faster, QEMU does not check at every basic block if a hardware interrupt is pending. Instead, the user must asynchronously call a specific function to tell that an interrupt is pending. This function resets the chaining of the currently executing basic block. It ensures that the execution will return soon in the main loop of the CPU emulator. Then the main loop can test if the interrupt is pending and handle it.

[...]

2.12.4 Self-virtualization

QEMU was conceived so that ultimately it can emulate itself. Although it is not very useful, it is an important test to show the power of the emulator.

Achieving self-virtualization is not easy because there may be address space conflicts. QEMU user emulators solve this problem by being an executable ELF shared object as the ld-linux.so ELF interpreter. That way, it can be relocated at load time.

>>

KVM

http://www.dedoimedo.com/computers/kvm-intro.html
[This tutorial is good for user-level management of VM's with qemu/kvm with GUIs as well as the 'virsh' CLI]

KVM functions by utilizing the CPU virtualization technology extensions on modern Intel and AMD processors, known as Intel-VT and AMD-V. Using a kernel module loaded into memory, KVM utilizes the processor and, via user-mode driver based on modified QEMU, it emulates a hardware layer upon which virtual machines can be created and run. KVM can also be executed without the CPU extensions, but then, it will run in a pure emulation mode using QEMU, resulting in a significant performance penalty.

KVM can be managed either via a graphical management tool, similar to VMware products or VirtualBox, or via command line using several methods.

The most popular GUI is called Virtual Machine Manager (VMM), developed by RedHat. The tool is also know by its generic package name virt-manager. It comes with a number of supporting tools, including virt-install, virt-clone, virt-image, and virt-viewer, which are used to provision, clone, install, and view virtual machines, respectively. VMM also supports Xen machines.

The generic KVM command interface is provided by virsh. Specifically, you can use the supporting tools, like virt-install for creating your virtual machines. On Ubuntu, there's a special ubuntu-vm-builder tool that can be used for provisioning Ubuntu builds, developed by Canonical.

[...]
To sum it up, KVM is good for you if you are looking for a free modern virtualization solution with an unlimited usage mode and without additional licensing fees or feature tiering, a powerful command line interface and you're not afraid to dirty your hands.

KVM is not good for you if your CPU does not have virtualization extensions, you're afraid of writing scripts, have no desire to dabble in code, and prefer administratively simpler solutions like VMware Server, ESXi or Virtualbox.
[...]

-> Check that your CPU has virtualization extensions:

```
# egrep -c '(vmx|svm)' /proc/cpuinfo
4
#
```

A non-zero result implies you're good.
[vmx for Intel, svm for AMD].

-> Also verify that CPU virtualization extensions are enabled in the system BIOS.

Resources
Virtualization Blog, Bartosz Milewski
- Virtual Machines: Virtualizing Virtual Memory
- Virtual Machines: The Traps

EPT

https://en.wikipedia.org/wiki/Second_Level_Address_Translation

Second Level Address Translation (**SLAT**), also known as **nested paging**, is a hardware-assisted virtualization technology which makes it possible to avoid the overhead associated with software-managed shadow page tables.

Intel's implementation of SLAT, known as Extended Page Table (EPT), was introduced in the Nehalem microarchitecture found in certain Core i7, Core i5, andCore i3 processors. AMD supports SLAT through the Rapid Virtualization Indexing (RVI) technology since the introduction of its third-generation Opteronprocessors (code name Barcelona).

ARM's virtualization extensions support SLAT, known as Stage-2 page-tables provided by a Stage-2 MMU. The guest uses the Stage-1 MMU. Support was added as optional in the ARMv7ve architecture and is also supported in the ARMv8 (32-bit and 64-bit) architectures.

…

Extended Page Tables (EPT) is an Intel second-generation x86 virtualization technology for the memory management unit (MMU). EPT support is found in Intel'sCore i3, Core i5 and Core i7 CPUs, among others.[1]

EPT is required in order to launch a logical processor directly in real mode, a feature called "unrestricted guest" in Intel's jargon, and introduced in the Westmere microarchitecture.[2][3]

According to a VMware evaluation paper: "EPT provides performance gains of up to 48% for MMU-intensive benchmarks and up to 600% for MMU-intensive microbenchmarks", although it can actually cause code to run slower than a software implementation in some corner cases.[4]

…

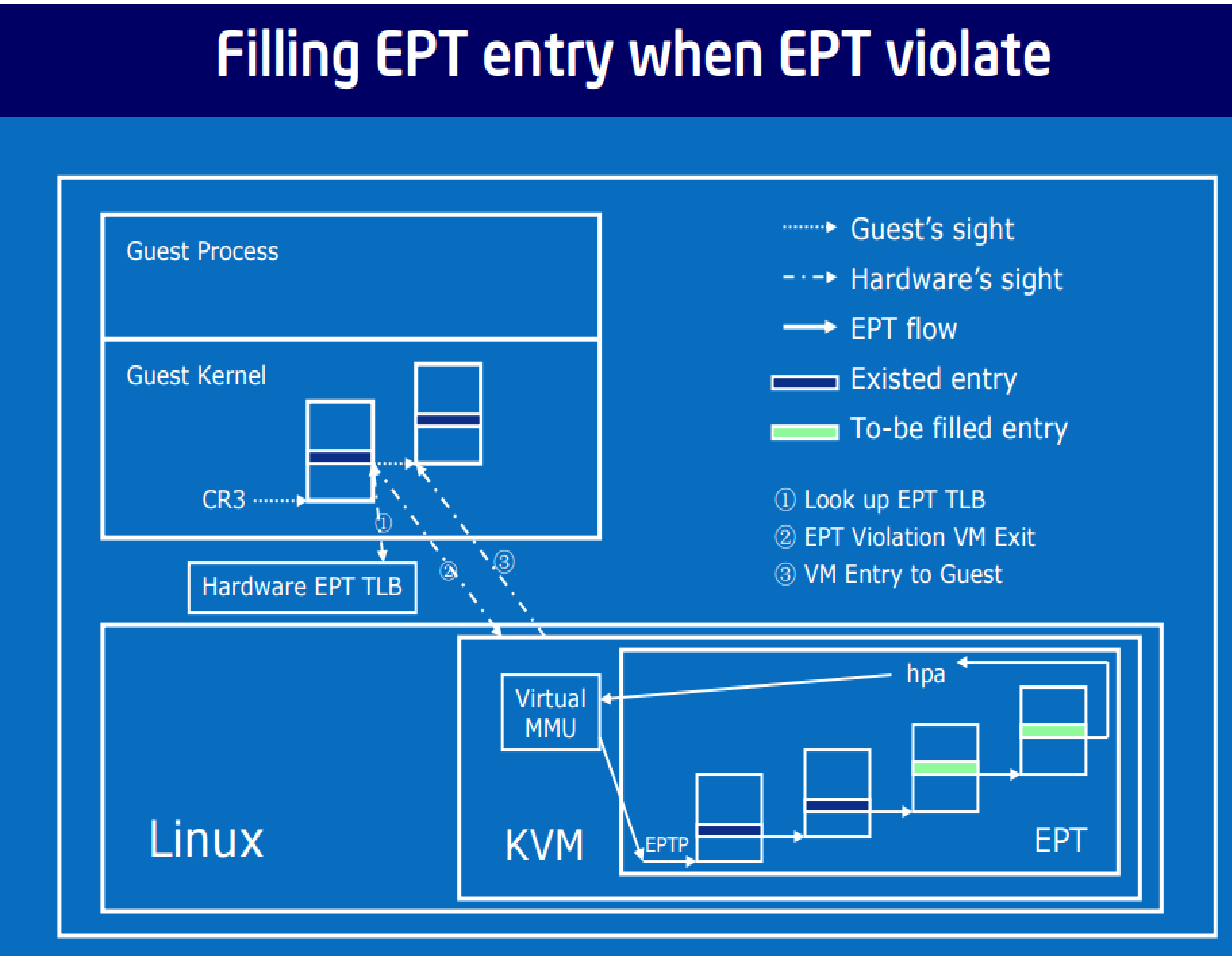
From Intel Vol 3 : Ch 28 : "VMX SUPPORT FOR ADDRESS TRANSLATION"

The architecture for VMX operation includes two features that support address translation: virtual-processor identifiers (VPIDs) and the extended page-table mechanism (EPT). VPIDs are a mechanism for managing translations of linear addresses. EPT defines a layer of address translation that augments the translation of linear addresses.

…

Filling EPT

Source: http://www.linux-kvm.org/images/c/c7/KvmForum2008$kdf2008_11.pdf


Filling EPT entry when EPT violate
Guest Process
Guest Kernel
CR3
Hardware EPT TLB
Guest's sight
Hardware's sight
EPT flow
Existed entry
To-be filled entry
① Look up EPT TLB
② EPT Violation VM Exit
③ VM Entry to Guest
①
②
③
Linux
KVM
Virtual MMU
hpa
EPTP
EPT