Contents

Implementations of Nested Virtualization

- Xen
- KVM

Have a look at the implementation pages for special information and performance data.

The idea

The idea behind nested virtualization is to run guests within guests.

```
                                                    +-------------+
L3                                                   | guest 2.2.1 |
                                                     +-------------+
                                                           |
      +-------------+  +-------------+ +-----------+ +-----------+
L2    | guest 1.1   |  | guest 1.2   | | guest 2.1 | | guest 2.2 |
      +-------------+  +-------------+ +-----------+ +-----------+
         |                   |               |             |
         +-------------------+               +-------------+
         |                                   |
      +-------------+                  +-----------+               +-----------+
L1    | guest 1     |                  | guest 2   |               | guest 3   |
      +-------------+                  +-----------+               +-----------+
         |                                   |                           |
         +-----------------------------------+---------------------------+
         |
      +------+
L0    | Host |
      +------+

                          Virtualization Levels
```

This pictures shows the virtualization levels. The host runs three guests. Two of them run another two guests in them. One of them run yet another guest in it.

The bird's eye

The guests at level N are the hosts for the guests at level N+1. The hosts of the guests at level N are at level N-1. The hardware is on level 0 (L0).

In theory, you can have as many levels as the user wants to have and on each level you can run as many guests as the user wants to, in case of level 0 that means you have multiple physical machines.

Paging Modes

This describes how the paging mechanism work with nested virtualization. Each guest has its own address space with virtual and physical addresses. Each guest maintains a pagetable to translate its virtual addresses into physical addresses.

In the picture below, it is assumed that nested guest PA is equal to guest VA and guest PA is equal to host VA.

```
                 +-------------+  nested guest VA
nested guest     | guest N+1   |
                 +-------------+  nested guest PA
                    |
                 +-------------+  guest VA
```

```
guest            | guest N     |
                 +-------------+  guest PA
                    |
                 +-------------+  host VA
host             | guest N-1   |
                 +-------------+  host PA
```

In order to run the guest, the host must translate guest VA into host PA. The host maintains a pagetable to translate guest addresses into host PA.
In order to run the nested guest, the gust must translate nested guest VA into guest PA. The guest maintains a pagetable to translate nested guest addresses into guest PA.

Shadow paging works with #PF intercepts. Since #PF report the failing VA, hence the page table translates guest VA into host PA or nested guest VA into guest PA, respectively.
Nested paging works with #NPF intercepts. #NPF report the failing PA, hence the page table translates guest PA into host PA or nested guest PA into guest PA, respectively.

There are now three ways to implement the page table maintenance:

Shadow-on-Shadow means both host and guest use shadow paging to run their guests.
Shadow-on-Nested means the host uses nested paging to run the guest and the guest uses shadow paging to run the nested guest.
Nested-on-Nested means both host and guest use nested paging to run their guests.

Shadow-on-Shadow

```
                 +--------------+  nested guest VA       +
L2               | nested guest |                        |
                 +--------------+  nested guest PA       |
                    |                                    |
                 +--------------+  guest VA         +    |
L1               | guest        |                   |    |
                 +--------------+  guest PA         |    |
                    |                               |    |
                 +--------------+  host VA          |    |
L0               | host         |                   |    |
                 +--------------+  host PA          V    V
                                                   PT10  PT20
```

This is a software only solution. The Host establishes a shadow page table which translates L2 guest virtual into host physical addresses (this is PT20 in above picture). On #PF intercept, the host walks the L1 guest shadow page table to translate the L2 guest virtual address into L1 guest virtual address. This is usually done by the hosts SoftMMU implementation.
If the walk fails then the L1 guest needs to fix its shadow page table. In this case, the host injects a VMEXIT (exitcode #PF intercept) into the L1 guest.
If the walk succeeds then the host fixes the shadow page table. Nested page fault (#NPF) intercepts never occur.

On VMRUN emulation the virtual cpu switches from PT10 to PT20 and on VMEXIT emulation the virtual cpu switches back from PT20 to PT10.

Shadow-on-Nested

```
                 +--------------+  nested guest VA        +
L2               | nested guest |                         |
                 +--------------+  nested guest PA        |
                    |                                     |
                 +--------------+  guest VA               |
L1               | guest        |                         |
                 +--------------+  guest PA          +    V
                    |                                |
                 +--------------+  host VA           |
L0               | host         |                    |
                 +--------------+  host PA           V
                                                  PT10  PT21
```

This is a combination of a hardware and software solution. This method is the easiest to implement into the hypervisor. The host already maintains the nested page table to run the guest (PT10 in the picture above). It is simply re-uses to run the nested guest. The host enables #PF intercept on VMRUN emulation and disables #PF intercept on VMEXIT emulation. If a #NPF intercept happens then the host fixes its host page table.
The guest already maintains its shadow page table (PT21 in the picture above) to run the nested guest. If a #PF intercept happens and the virtual cpu is in guest mode then the host injects a VMEXIT (exitcode #PF intercept) into the guest. The guest fixes its shadow page table.
A #PF intercept with virtual cpu in host mode should never occur.

Nested-on-Nested

```
                 +--------------+  nested guest VA
L2               | nested guest |
                 +--------------+  nested guest PA        +     +
                    |                                     |     |
                 +--------------+  guest VA               |     |
L1               | guest        |                         |     |
                 +--------------+  guest PA          +    |     V
                    |                                |    |
                 +--------------+  host VA           |    |
L0               | host         |                    |    |
                 +--------------+  host PA           V    V
                                                  PT10  PT20  PT21
```

This is a hardware only feature but needs software tricks to work. You need to deal with three page tables, the host page table, L1 guest page table and the L2 guest page table. The hardware can only deal with two page tables.

The software trick is to make the host creating a shadow page table which translates L2 guest physical addresses into host physical addresses (this is PT20 in the picture above). When a #NPF intercept happens and the virtual cpu is in host mode then the host needs to fix the page table used to run the L1 guest (PT10 in the picture above). When a #NPF intercept happens and the virtual cpu is in guest mode then the host walks the L1 guest page table (PT21 in the picture above) to translate the L2 guest physical address into L1 guest physical address. If the walk fails then the host injects a VMEXIT (exitcode #NPF) into the L1 guest. The L1 guest fixes its PT21 table. If the walk succeeds then the host fixes its shadow page table (PT20 in the picture above).

A #PF intercept should never occur.

On VMRUN emulation the virtual cpu switches from PT10 to PT20 and on VMEXIT emulation the virtual cpu switches back from PT20 to PT10.

General paging considerations

For Shadow-on-shadow and nested-on-nested paging modes, the host wants to know when the L1 guest (re-)starts a new or another L2 guest. There are several reasons for this:

- Reusing a shadow page table for another L2 guest results in wrong translations and causes a triple fault in the L1 guest.
- L1 guest runs a L2 SMP guest, each virtual cpu with its own n_cr3 or cr3 values. (Note, the L1 guest virtual cpus may schedule multiple L2 guest virtual cpus.)
- A L2 guest shutdown or crashed and host don't want to leak the shadow page table.

There are generally two ways to know when the L1 guest wants the host to flush its shadow page table.

- The L1 guest sets the tlb_control bit(s) in the vmcb
- The L1 guest changes the ASID in the vmcb

In nested-on-nested paging mode, it is additionally possible to share the shadow page table across multiple virtual cpus. In this case, the L1 guest writes the same n_cr3 addresses in the vmcb for multiple virtual cpus. This helps to reduce the number of #NPF intercepts.

Further, if the L1 guest does not change the ASID and n_cr3 and the tlb_control bit(s) are not set in the vmcb then the host can recycle the same shadow page table it has used on the prior VMRUN emulation. This also helps to reduce the number of #NPF intercepts.

The host has initially a list of several empty shadow page tables for each L1 guest. The host activates one of them in the VMRUN emulation and deactivates it in the VMEXIT emulation without flushing. Activation means the chosen shadow page table will be used for translation of L2 guest addresses into host physical addresses and will be filled up by the (nested) page fault handler. Flushing a shadow page table means, make it empty again.

On activation, the host shares, recycles or flushes the shadow page table depending on the vmcb conditions explained above. The activated one becomes the first entry in the list of shadow page tables no matter where it was before in the list. In the case of no vmcb condition matches it flushes and activates the shadow page table which is the last entry in the list and becomes the first entry in the list (= Last-Recent-Used mechanism).

The use of the LRU mechanism prevents leakage of shadow page tables when one or more L2 guests are started and shutdown in sequence.

Emulation

The guests can't run the instructions directly. They trigger an intercept. The L0 host receives the intercepts and has to emulate the behaviour of the hardware.

In the emulation, there are always two vmcb's involved: The guest on level N-1 sets up a vmcb to run guest on level N. The guest on level N sets up a vmcb to run guest on level N+1.

```
                 +-------------+
nested guest     | guest N+1   |
                 +-------------+
                    |
                 +-------------+----------+
guest            | guest N     | vmcb N+1 |
                 +-------------+----------+
                    |
                 +-------------+----------+
host             | guest N-1   | vmcb N   |
                 +-------------+----------+
```

That means, the cpu state of guest on level N is in the vmcb N.

On VMRUN, guest N-1 passes vmcb N to the cpu to run guest N. guest N passes vmcb N+1 to the cpu to run guest N+1.

The VMRUN/VMEXIT cycle when guest starts the nested guest:

```
 nested guest                          +-----+
                                       |     |
                                       |     |
 guest       +- VMRUN -+               |     |                 +- VMEXIT, VMRUN -+
             |         |               |     |                 |                 |
             |         |               |     |                 |                 |
 host VMRUN -+         +- VMRUN, VMRUN-+     +- VMEXIT, VMRUN -+                 +-> VMRUN,
                       (emulation)           (emulation)                          (emulation)
```

The VMRUN/VMEXIT cycle when nested guest is running and a special intercept (i.e. an interrupt) happens:

```
 nested guest --+                   +-->
                |                   |
                |                   |
 guest          |                   |
                |                   |
                |                   |
 host           +-- VMEXIT, VMRUN --+
```

Note, the normal VMEXIT handler runs, not the VMEXIT emulation.

The VMRUN/VMEXIT cycle in the same host #VMEXIT:

```
nested guest --+
               |
               |
guest          |                 +- VMEXIT, VMRUN -+                          +- VMEXIT, VMRUN -+
               |                 |                 |                          |                 |
               |                 |                 |                          |                 |
host           +- VMEXIT, VMRUN -+                 +- VMEXIT, VMEXIT, VMRUN -+                 +->
               (emulation)                         (emulation)
```

This scenario happens when a high priority interrupt occured (i.e. NMI, Machine Check Exception). This is not used for normal interrupts because too many interrupts let the nested guest become stuck. As a consequence operations causing interrupts wouldn't work due to time outs (i.e. copying a file from/to network), otherwise.

VMSAVE

The guest N expects to have a subset of the cpu state saved in the vmcb N+1. The pseudo-algorithm to emulate VMSAVE in the host (guest on level N-1) is as follows:

```
 1. Translate guest physical address guest N stored in rax into host virtual address.
 2. Map vmcb N+1 into host.
 3. VMSAVE(vmcb N)    // real cpu instruction. save real cpu state into vmcb N
 4. Copy vmcb fields vmcb N into vmcb N+1:
    fs, gs, tr, ldtr, kerngsbase, star, lstar, cstar, sfmask,
    sysenter_cs, sysenter_esp, sysenter_eip
 5. Unmap vmcb N+1.
```

VMLOAD

The guest N expects to have a subset of the cpu state loaded from the vmcb N+1. The pseudo-algorithm to emulate VMLOAD in the host (guest on level N-1) is as follows:

```
1. Translate guest physical address guest N stored in rax into host virtual address.
2. Map vmcb N+1 into host.
3. Copy vmcb fields vmcb N+1 into vmcb N:
   fs, gs, tr, ldtr, kerngsbase, star, lstar, cstar, sfmask,
   sysenter_cs, sysenter_esp, sysenter_eip
4. VMLOAD(vmcb N)    // real cpu instruction. load real cpu state from vmcb N
5. Unmap vmcb N+1.
```

VMMCALL

VMMCALL may be used by paravirtualized drivers to cause the cpu to exit the guest state. The VMMCALL needs no special handling for nested virtualization.

SKINIT

Not relevant for virtualization. Inject #GP into the guest.

STGI

The pseudo-algorithm is:

```
1. Set an emulated GIF
2. Enable events for paravirtualized drivers.
```

CLGI

The pseudo-algorithm is:

```
1. Clear the VINTR intercept     // clears an eventual open interrupt window
2. Clear the emulated GIF
3. Mask events for paravirtualized drivers
```

VMRUN

The pseudo-algorithm is this:

```
1. Remember the guest physcial address stored in rax        // Will be needed for emulated VMEXIT
2. Translate guest physical address guest N stored in rax into host virtual address.
3. Map vmcb N+1 into host.
4. Save the host state
5. vcpu enters guest state
6. Remember V_INTR_MASK state in the host
7. Load vmcb N+1
8. Load guest registers from vmcb N+1:
      rax, rip, rsp, rflags
9. Run STGI emulation                                 // Enables interrupts
10. Unmap vmcb N+1
```

Saving the hoststate

Step "4. Save the host state" in VMRUN in the detail:

```
1. Increase the rip value in the vmcb N by the length of the VMRUN instruction.
2. Save vmcb N into a temporary storage.
3. Remember the state of the interrupt flag in rflags field.
4. Save the shadowed values of the following registers into the temporary storage:
     efer, cr0, cr2, cr4
5. Check paging mode
5.a. on nested-nested and on nested-shadow, cr3 and n_cr3 already have been saved in step 2.
5.b. on shadow-shadow, save the shadowed value of cr3 into the temporary storage, clear
     n_cr3 to zero.
```

The shadowed values are managed by the hypervisor. The shadowed values are visible to the guest. Using the shadowed values is mandatory for the security. The guest could set its own values through the saving/restoring hoststate phase and break out, otherwise.

Loading VMCB N+1

Step "7. Load vmcb N+1" in VMRUN in the detail:

```
1. Remember the intercepts as set in vmcb N+1  // Needed for checking the intercepts on VMEXIT
2. Combine the intercepts in vmcb N with intercepts in vmcb N+1 with a binary 'or'
3. Virtualize MSR and IO permission bitmaps
4. Load TSC by adding tsc_offset from vmcb N+1 to the tsc_offset from vmcb N
5. Combine the tlb_control in vmcb N with tlb_control in vmcb N+1 with a binary 'or'
6.a. Copy the vintr field from vmcb N+1 into vmcb N
6.b. Set the intr_masking bit in the vintr field in vmcb N.
7. Remember the lbr_control as set in vmcb N+1     // Needed for checks on VMEXIT
8. Combine lbr_control in vmcb N with lbr_control in vmcb N+1 with a binary 'or'
9. Load the fields from the vmcb N+1:
     efer, cr4, cr0, cr2
10.a. On nested-on-nested, switch n_cr3 to the shadow nested page table, load cr3 as is
      from vmcb N+1, enable nested paging in vmcb N
10.b. On shadow-on-nested, load cr3 from vmcb N+1, load n_cr3 from vmcb N, enable #PF
      intercept, enable nested paging in vmcb N
10.c. On shadow-on-shadow, disable nested paging in vmcb N, clear n_cr3 in vmcb N,
      switch cr3 to the shadow shadow page table
11. Copy the fields from vmcb N+1 into vmcb N:
     pause_filter_count, interrupt_shadow, exitcode, exitinfo1, exitinfo2, exitintinfo,
     eventinj, nextrip, es, cs, ss, ds, gdtr, idtr, cpl, dr6, dr7, rflags, rip, rsp, rax,
     pdpe0, pdpe1, pdpe2, pdpe3, g_pat, debugctlmsr, lastbranchfromip, lastbranchtoip,
     lastintfromip, lastinttoip
12. Invalidate ASID
```

VMEXIT

The pseudo-algorithm is this:

```
1. Run CLGI emulation                   // Disables interrupts
2. Get guest physical address of vmcb N+1, saved in VMRUN emulation
3. Map vmcb N+1 into host
4. If vcpu is in host mode, proceed with step 11
5. Load vmcb N into temporary storage
6.a. Check special intercepts and handle them:
     #PF, #NPF, #INTR, #NMI, #MC, #MSR, #IOIO
6.b. Check if guest intercepts the intercept
6.c.   If guest does not intercept the intercept then run normal host vmexit handler.
       vcpu stays in guest mode. guest N+1 continues running.
       If guest intercepts the intercept, mark the intercept for injecting a #VMEXIT
       into the guest.
7. Prepare vmcb N+1 for VMEXIT injection
8. Restore host state
9. Switch vcpu into host mode
10. Inject VMEXIT into guest        // Make changes to vmcb N+1 in step 7 visible to the guest
11. Unmap vmcb N+1
```

Special intercepts

Special intercepts are those that need special handling. These are #INTR, #NMI, #MC, #MSR, #IOIO, #PF and #NPF.

For #INTR, #NMI and #MC run the normal host vmexit handler. vcpu stays in guest mode. guest N+1 continues running.

For #MSR and #IOIO check the virtualized permission bitmap if guest intercepts the MSR/IOIO operation or not. If it does, then inject a VMEXIT into the guest. If it does not, then run the normal host vmexit handler.

For #PF and #NPF check host and guest paging mode.

On Shadow-on-Shadow, a #NPF can never happen.
On Shadow-on-Shadow, a #PF is specially handled.
On Nested-on-Shadow, a #NPF is handled by the normal vmexit handler.
On Nested-on-Shadow, a #PF is always handled by injecting a VMEXIT into the guest.
On Nested-on-Nested, a #NPF is specially handled.
On Nested-on-Nested, a #PF should never happen, but if it happens then inject a VMEXIT into the guest.

See Paging Modes section for details.

Restore the hoststate

Step "8. Restore host state" in VMEXIT in the detail:

```
1. Load vmcb N from temporary storage, but keep register values handled
   by VMSAVE/VMLOAD unchanged.
2. Clear the VM bit in the rflags in the vmcb N
3. Load shadowed values from vmcb N into the hypervisor:
     * efer, cr4, cr0, cr2, cr3
     * On loading cr0, unconditionally set the PE bit.
4. On Shadow-Shadow, perform an MMU context reset
5. Restore guest registers from vmcb N:
     rax, rsp, rip, rflags
6. Clear the dr7 field in the vmcb N
7. Set cpl to 0 in the vmcb N
8. Clear the exitintinfo field in the vmcb N     // Prevents loop of re-injecting the
                                                 // same event again and again.
9. Invalidate ASID
```

Prepare VMEXIT

Step "7. Prepare vmcb N+1 for VMEXIT injection" in VMEXIT in the detail:

```
1. VMSAVE(vmcb N)                // real cpu instruction
2. Copy remembered intercepts back into vmcb N+1
3. Copy the fields from vmcb N into vmcb N+1:
    pause_filter_count, tsc_offset, interrupt shadow, exitcode,
    exitinfo1, exitinfo2, nextrip, es, cs, ss, ds, gdtr, idtr,
    cpl, efer, cr4, cr0, cr2, rflags, rip, rsp, rax,
    pdpe0, pdpe1, pdpe2, pdpe3, g_pat, debugctlmsr,
    lastbranchfromip, lastbranchtoip, lastintfromip,
    lastinttoip
4.a. If eventinj field in vmcb N is invalid then copy exitintinfo from vmcb N into vmcb N+1
4.b. If eventinj field is valid in vmcb N and the event needs to be
    reinjected then copy the eventinj field from vmcb N into
    the exitintinfo field from vmcb N+1
5. Clear tlb_control in vmcb N+1
6. Clear eventinj field in vmcb N+1
7.a. On nested-on-nested, switch n_cr3 to page table used to run L1 guest,
     copy the fields from vmcb N into vmcb N+1:
       np_enable, cr3
7.b. On shadow-on-nested, clear np_enable and n_cr3 in vmcb N+1,
     copy cr3 from vmcb N into vmcb N+1, disable #PF intercept
7.c. On shadow-on-shadow, clear np_enable and n_cr3 in vmcb N+1
8. Copy remembered lbr_control back into vmcb N+1
```

To clarify step 4, it may happen that the host emulates a VMRUN/VMEXIT in the same host #VMEXIT cycle. In this case, make sure to not lose injected events. Hence check eventinj and copy it to exitintinfo if it is valid and the event type needs re-injection. exitintinfo and eventinj can't be both valid because this case only happens on a VMRUN instruction intercept which has no valid exitintinfo set.

To clarify step 7, remember the values in vmcb N are shadowed. The original unshadowed values are still in vmcb N+1. Thus in 7.a. keep n_cr3 as is in vmcb N+1 and in 7.c. keep cr3 as is in vmcb N+1. The other modifications are needed to enforce correctness. Never trust the guest to never find a way to break out. In step 7.a. remember the n_cr3 from vmcb N+1 or in step 7.c. remember the cr3 from vmcb N+1, respectively. They are needed to walk the guest page walk tables.

Paravirtualized drivers / Invoking a hypercall

A guest can use paravirtualized drivers to improve IO performance. To make paravirtualized driver work, they need a way to cause the CPU to exit the guest. This can be achieved by using VMMCALL or using special MSRs which don't exist in hardware.