Improving KVM x86 Nested Virtualization

Liran Alon

Who am I?

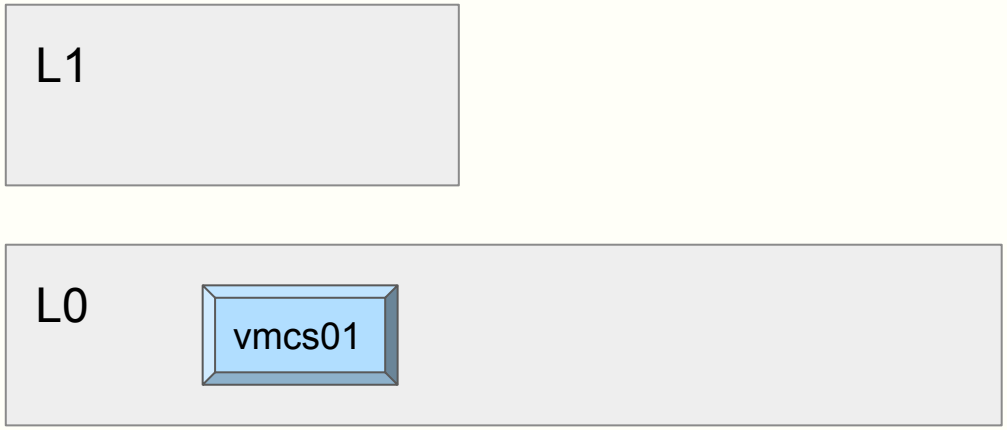
- Architect at OCI (Oracle Cloud Infrastructure)
- ~4 years of Virtualization, SDN and Cloud Computing
- ~8 years of cyber R&D in PMO & IDF
- Active KVM nVMX contributor
- Interests: Anything low-level
  - CPU, OS internals, networking, vulnerabilities, exploits, virtualization and etc.
- Twitter: @Liran_Alon

Outline

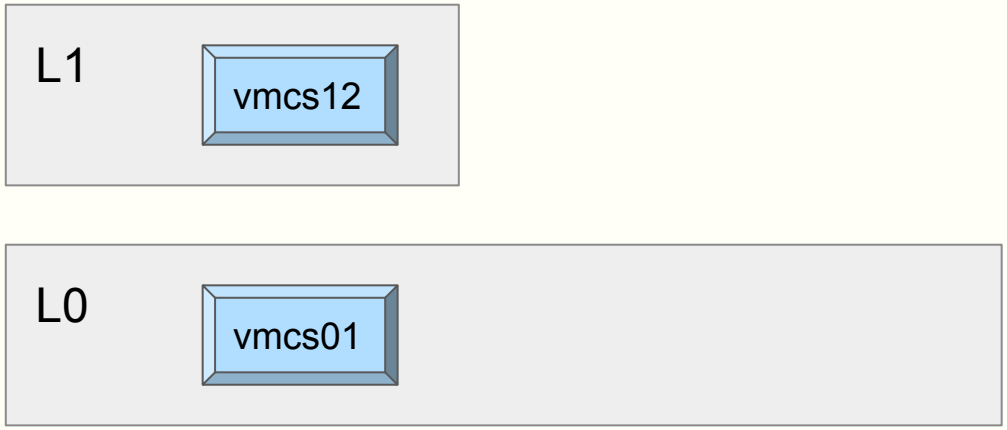
- Focus only on nVMX (Sorry AMD…)
- Deep-dive into one nVMX mechanism which had many issues
  - First documentation of mechanism outside code
  - Maybe relevant for other architectures nested support
- Present recent nVMX improvements in high-level
- Highlight nVMX open issues
- Suggest possible nVMX future directions

How does nVMX works?

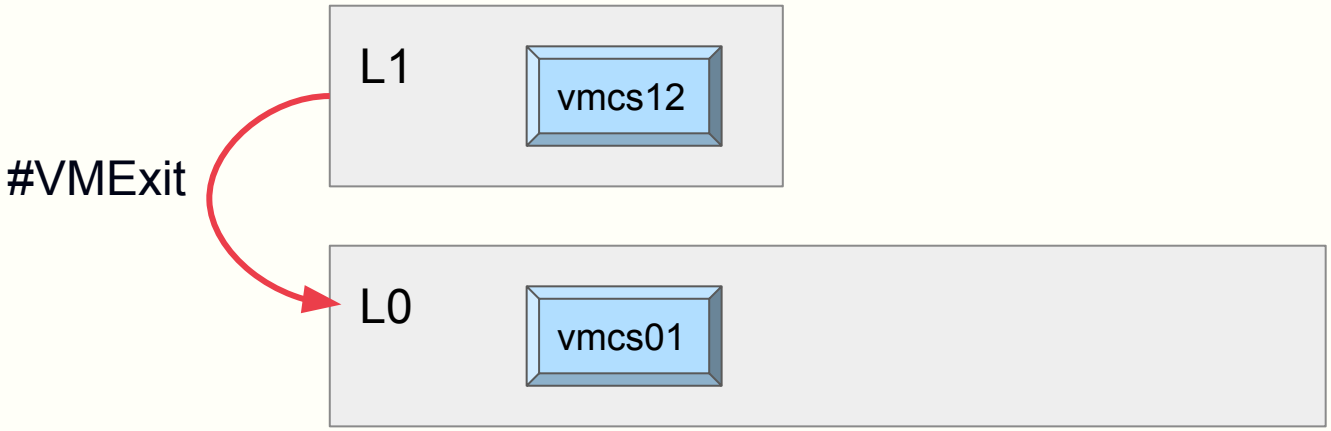
L0 runs L1 with vmcs01



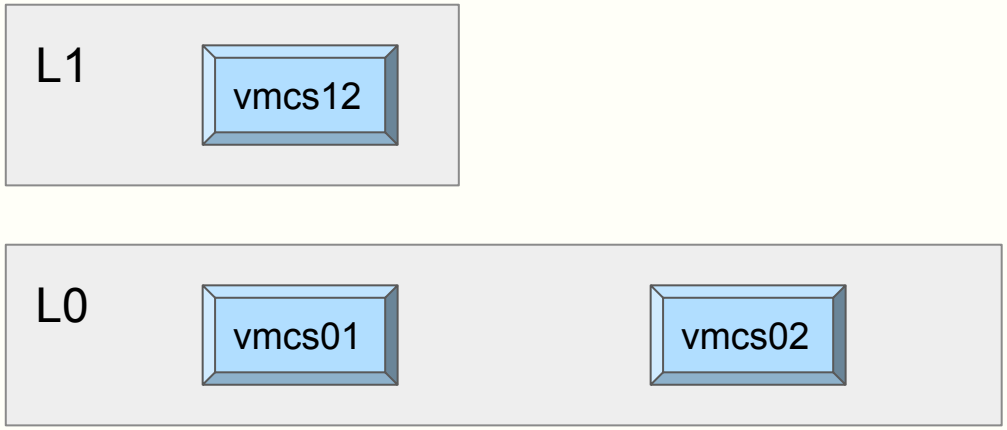
L1 creates vmcs12 for running L2



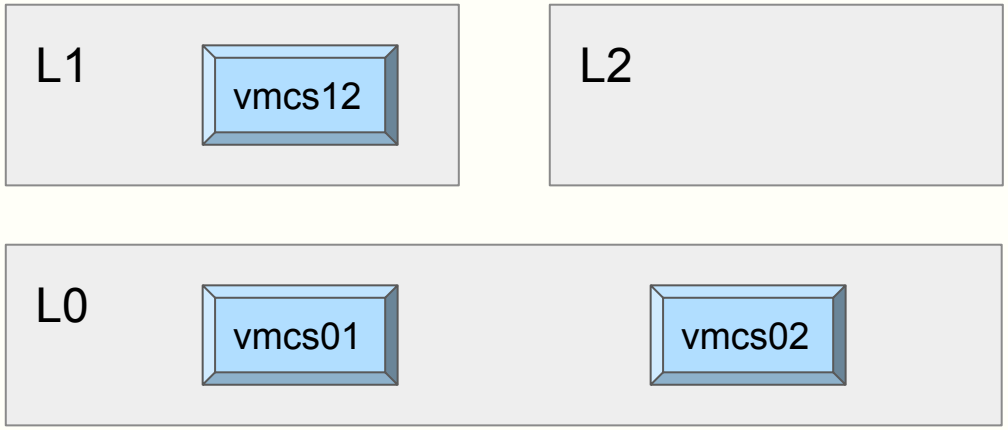
L1’s VMRESUME triggers VMExit to L0



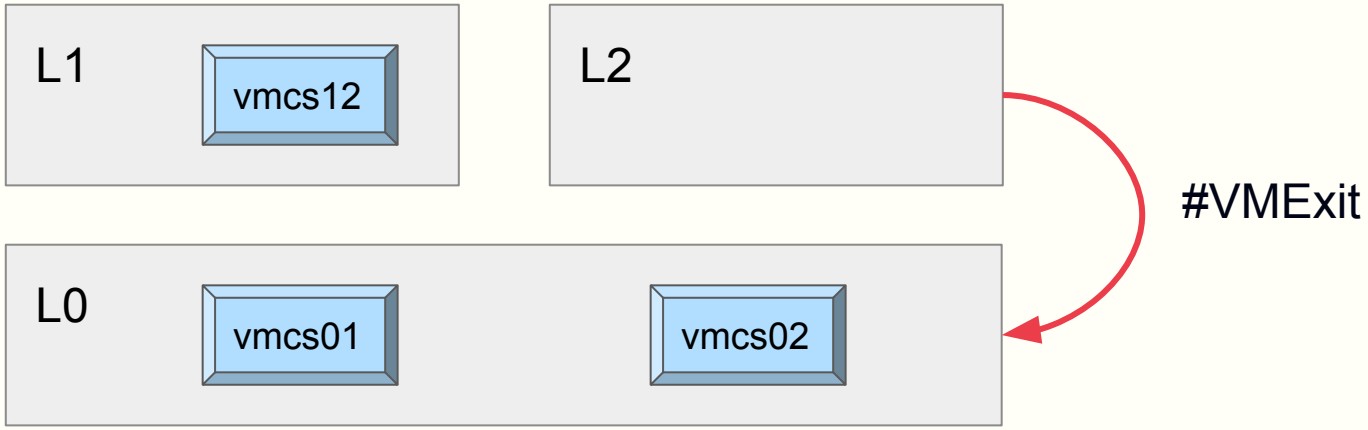
L0 merges vmcs01 & vmcs12 to vmcs02



L0 runs L2 with vmcs02

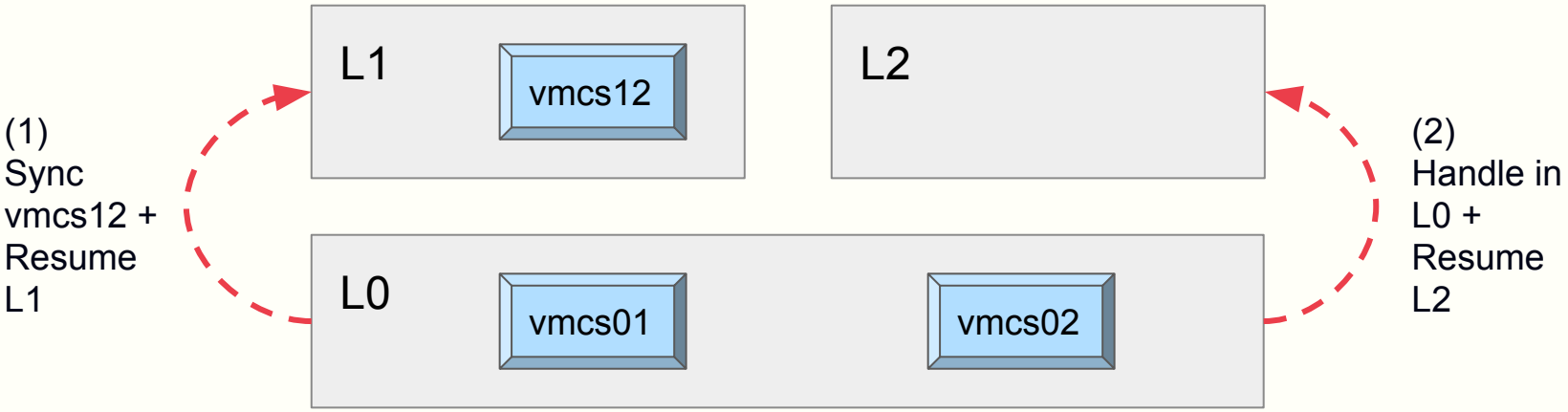


L2 runs until VMExit triggered because of vmcs02



L0 decides if to handle VMExit itself

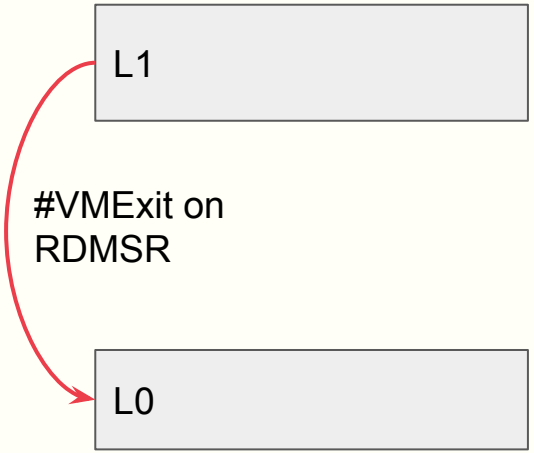
or reflect VMExit to L1



nVMX event injection

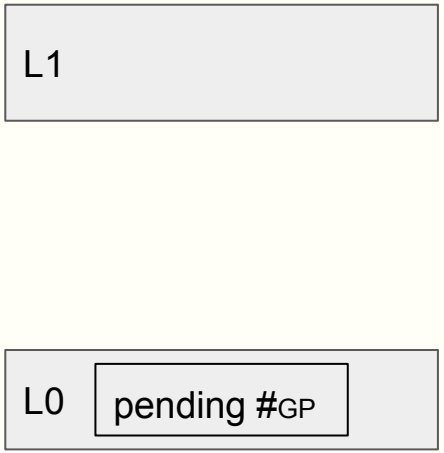
Basic KVM event-injection

1. L1 RDMSR bad_msr
2. RDMSR exits to L0

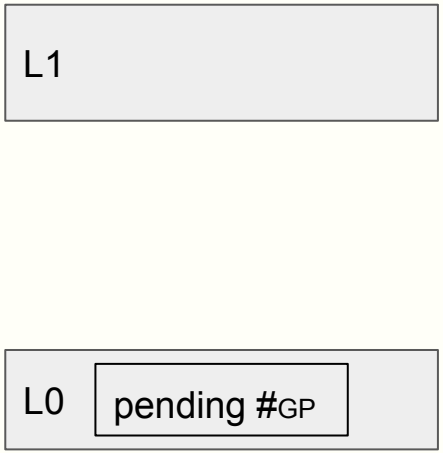


Basic KVM event-injection

1. L1 RDMSR bad_msr
2. RDMSR exits to L0
3. L0 emulates RDMSR and queues #GP:
   (a) Save pending exception in struct kvm_vcpu_arch
   (b) Set KVM_REQ_EVENT

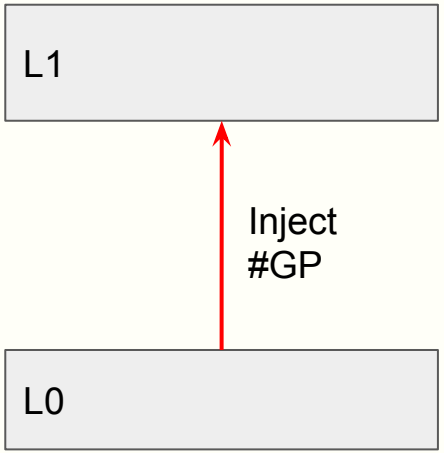


Basic KVM event-injection



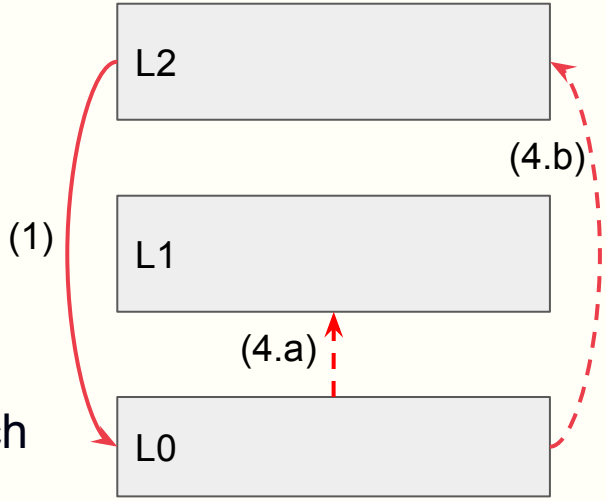
1. L1 RDMSR bad_msr
2. RDMSR exits to L0
3. L0 emulates RDMSR and queues #GP:
   (a) Save pending exception in struct kvm_vcpu_arch
   (b) Set KVM_REQ_EVENT
4. Before host entry to guest, KVM_REQ_EVENT evaluates queued events:

Basic KVM event-injection



1. L1 RDMSR bad_msr
2. RDMSR exits to L0
3. L0 emulates RDMSR and queues #GP:
   (a) Save pending exception in struct kvm_vcpu_arch
   (b) Set KVM_REQ_EVENT
4. Before host entry to guest, KVM_REQ_EVENT evaluates queued events:
   Inject pending #GP to guest by VMCS

Basic KVM nVMX event-injection



1. L2 RDMSR bad_msr
2. RDMSR exits to L0
3. L0 emulates RDMSR and queues #GP:
   (a) Save pending exception in struct kvm_vcpu_arch
   (b) Set KVM_REQ_EVENT
4. Before host entry to guest, KVM_REQ_EVENT evaluates queued events:
   (a) If (vCPU in guest-mode && Has event which should L2→L1)
   ⇒ Emulate L2→L1
   (b) Otherwise ⇒ Inject pending #GP to guest by VMCS

Exit during event-delivery

- A VMExit can occur during event-delivery
- Example: Write of exception frame to stack triggers EPT_VIOLATION

Exit during event-delivery

- A VMExit can occur during event-delivery
- Example: Write of exception frame to stack triggers EPT_VIOLATION
- CPU saves the event which attempted to deliver in vmcs→idt_vectoring_info

Exit during event-delivery

- A VMExit can occur during event-delivery
- Example: Write of exception frame to stack triggers EPT_VIOLATION
- CPU saves the event which attempted to deliver in vmcs→idt_vectoring_info
- On guest→host:
  1. KVM checks if vmcs→idt_vectoring_info valid
  2. If valid, queue injected event in struct kvm_vcpu_arch
     and set KVM_REQ_EVENT

Exit during event-delivery

- A VMExit can occur during event-delivery
- Example: Write of exception frame to stack triggers EPT_VIOLATION
- CPU saves the event which attempted to deliver in vmcs→idt_vectoring_info
- On guest→host:
  1. KVM checks if vmcs→idt_vectoring_info valid
  2. If valid, queue injected event in struct kvm_vcpu_arch
     and set KVM_REQ_EVENT
- KVM_REQ_EVENT will evaluate injected event on next entry to guest

nVMX Exit during event-delivery

- What if VMExit occurs during event-delivery to L2?

nVMX Exit during event-delivery

- What if VMExit occurs during event-delivery to L2?
- If exit reflected to L1 ⇒ Emulate exit during event-delivery to L1
  - Emulate L2→L1 VMExit with injected event in vmcs12→idt_vectoring_info

nVMX Exit during event-delivery

- What if VMExit occurs during event-delivery to L2?
- If exit reflected to L1 ⇒ Emulate exit during event-delivery to L1
  - Emulate L2→L1 VMExit with injected event in vmcs12→idt_vectoring_info
- Otherwise ⇒ Inject event directly to L2
  - L1 **should not** intercept event!
  - L1 should not be aware L0 had exit during attempt to deliver event

nVMX Exit during event-delivery

- What if VMExit occurs during event-delivery to L2?
- If exit reflected to L1 ⇒ Emulate exit during event-delivery to L1
  - Emulate L2→L1 VMExit with injected event in vmcs12→idt_vectoring_info
- Otherwise ⇒ Inject event directly to L2
  - L1 **should not** intercept event!
  - L1 should not be aware L0 had exit during attempt to deliver event
- How L0 knows L1 cannot intercept event on KVM_REQ_EVENT handler?

nVMX Exit during event-delivery

- What if VMExit occurs during event-delivery to L2?
- If exit reflected to L1 ⇒ Emulate exit during event-delivery to L1
  - Emulate L2→L1 VMExit with injected event in vmcs12→idt_vectoring_info
- Otherwise ⇒ Inject event directly to L2
  - L1 **should not** intercept event!
  - L1 should not be aware L0 had exit during attempt to deliver event
- How L0 knows L1 cannot intercept event on KVM_REQ_EVENT handler?
- **nVMX requires clear separation between pending vs. injected!**
  - A pending event **can** be intercepted by L1
  - An injected event **cannot** be intercepted by L1

Example: nVMX event handling issue

- Occasionally, L1 stuck after running L2 guests

Example: nVMX event handling issue

- Occasionally, L1 stuck after running L2 guests
- L1 dmesg reveals some hints on the issue
- All L1 CPUs but one waiting on KVM's mmu_lock:
  - **_raw_spin_lock**+0x20/0x30
    tdp_page_fault+0x1b1/0x260 [kvm]
    ? __remove_hrtimer+0x3c/0x90
    kvm_mmu_page_fault+0x65/0x130 [kvm]
    handle_ept_violation+0xaa/0x1a0 [kvm_intel]

Example: nVMX event handling issue

- Occasionally, L1 stuck after running L2 guests
- L1 dmesg reveals some hints on the issue
- All L1 CPUs but one waiting on KVM's mmu_lock:
  - **_raw_spin_lock**+0x20/0x30
    tdp_page_fault+0x1b1/0x260 [kvm]
    ? __remove_hrtimer+0x3c/0x90
    kvm_mmu_page_fault+0x65/0x130 [kvm]
    handle_ept_violation+0xaa/0x1a0 [kvm_intel]
- One L1 CPU is holding KVM's mmu_lock while waiting for IPI ACK:
  - ? **smp_call_function_many**+0x1c7/0x250
    kvm_make_all_cpus_request+0xbb/0xd0 [kvm]
    kvm_flush_remote_tlbs+0x1d/0x40 [kvm]
    kvm_mmu_commit_zap_page+0x22/0xf0 [kvm]
    mmu_free_roots+0x13c/0x150 [kvm]

L0 KVM event trace

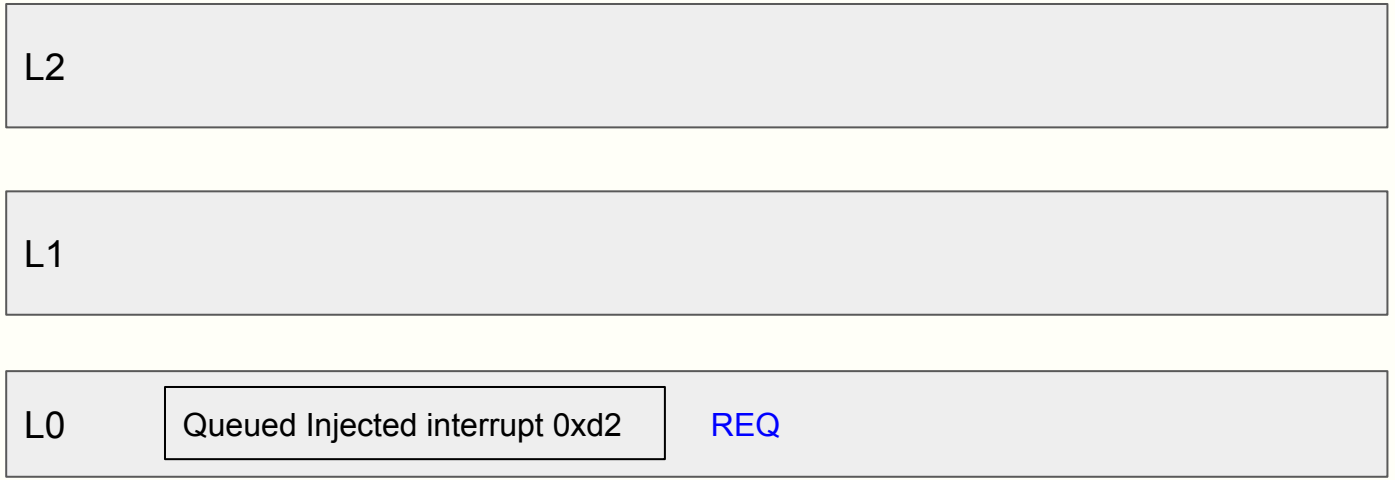
```
qemu-system-x86-19066 [030] kvm_nested_vmexit: rip: ⇦ Exit L2 to L0
0xffffff802c5dca82f reason: EPT_VIOLATION ext_inf1: 0x0000000000000182
ext_inf2: 0x00000000800000d2 ext_int: 0x00000000 ext_int_err: 0x00000000
```

⇒ L2→L0 exit during event-delivery of interrupt 0xd2

L0 KVM event trace

```
qemu-system-x86-19066 [030] kvm_nested_vmexit: rip: ⇐ Exit L2 to L0
0xffffff802c5dca82f reason: EPT_VIOLATION ext_inf1: 0x0000000000000182
ext_inf2: 0x00000000800000d2 ext_int: 0x00000000 ext_int_err: 0x00000000
```

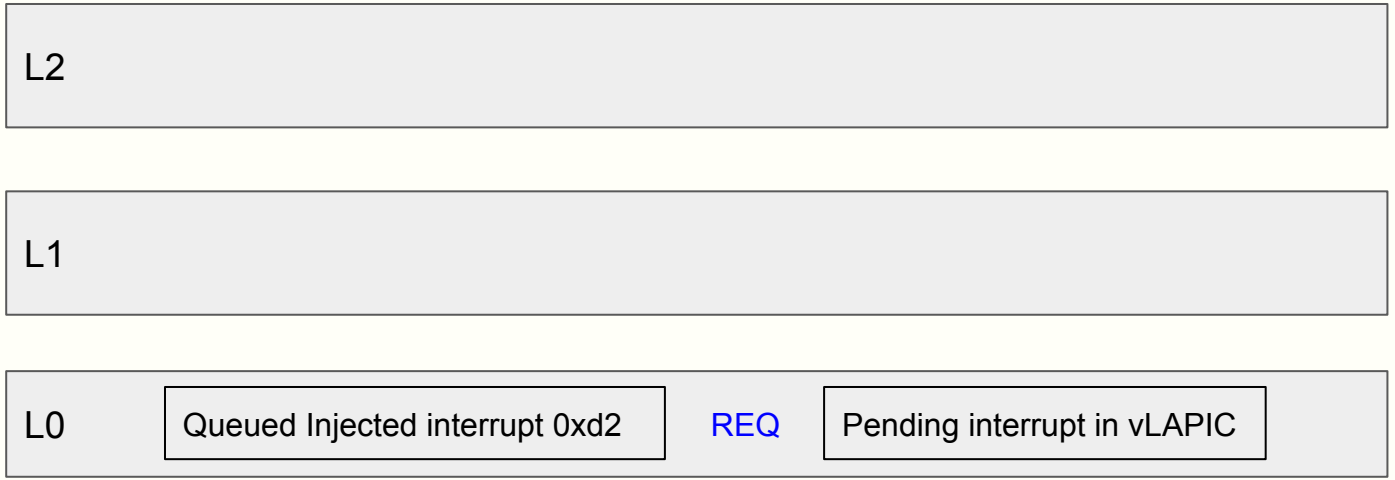
⇒ L2→L0 exit during event-delivery of interrupt 0xd2
⇒ L0 queues event for injection + Set KVM_REQ_EVENT



L0 KVM event trace

```
qemu-system-x86-19054 [028] kvm_apic_accept_irq: apicid f vec 252 (Fixed|edge)
```

⇒ Received IPI queued pending interrupt 252 in L1 vLAPIC



L0 KVM event trace

```
qemu-system-x86-19066 [030] kvm_inj_virq: irq 210 ⇇ re-inject interrupt to L2

qemu-system-x86-19066 [030] kvm_entry: vcpu 15 ⇇ Resume L2
```

⇒ KVM_REQ_EVENT re-inject queued injected interrupt to L2

L0 KVM event trace

```
qemu-system-x86-19066 [030] kvm_nested_vmexit: rip: ⇦ Exit L2 to L0
0xffffe00069202690 reason: EPT_VIOLATION ext_inf1: 0x0000000000000083
ext_inf2: 0x0000000000000000 ext_int: 0x00000000 ext_int_err: 0x00000000
```

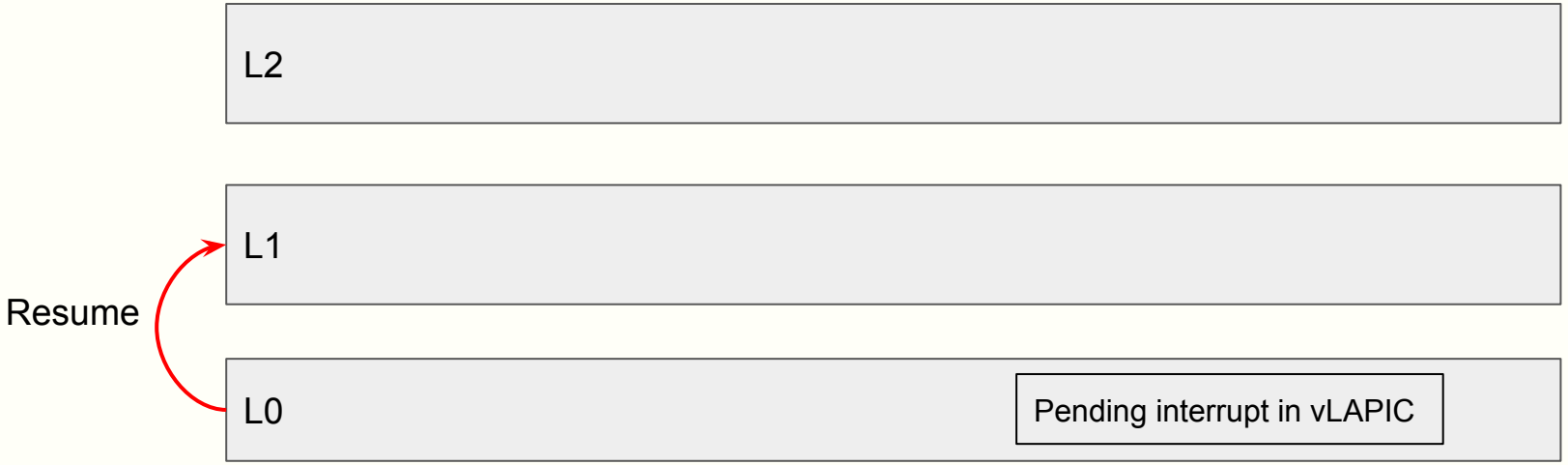
⇒ L2→L0 on EPT_VIOLATION (not during event-delivery)

L0 KVM event trace

```
qemu-system-x86-19066 [030] kvm_nested_vmexit_inject: reason:
EPT_VIOLATION ext_inf1: 0x0000000000000083 ext_inf2: 0x0000000000000000
ext_int: 0x00000000 ext_int_err: 0x00000000
⇇ Emulate exit from L2 to L1

qemu-system-x86-19066 [030] kvm_entry: vcpu 15 ⇇ Resume L1
```

⇒ L0 resumes L1



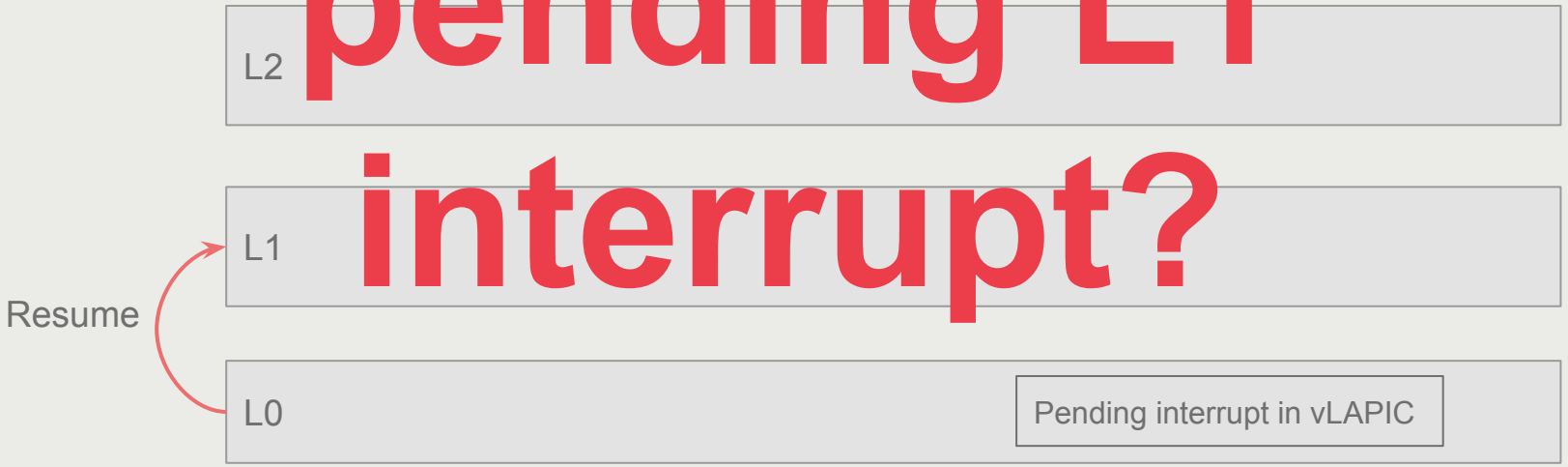
L0 KVM event trace

```
qemu-system-x86-19066 [030] kvm_nested_vmexit_inject: reason:
EPT_VIOLATION ext_inf1: 0x0000000000000083 ext_inf2: 0x0000000000000000
ext_int: 0x00000000 ext_int_err: 0x00000000
⇐ Emulate ext from L2 to L1

qemu-system-x86-19066 [030] kvm_entry: vcpu 15 ⇐ Resume L1
```

⇒ L0 resumes L1

What about the pending L1 interrupt?

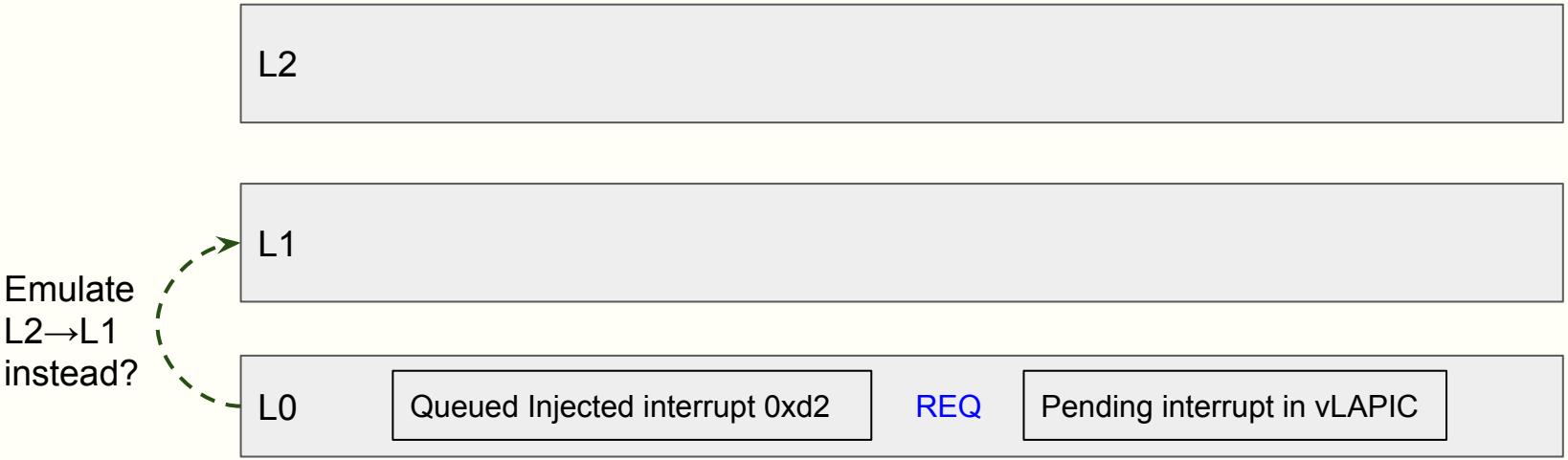


Example: nVMX event handling issue

- L0 didn’t re-evaluate pending L1 event because KVM_REQ_EVENT not set
  - IPI received when KVM_REQ_EVENT already set (Before re-injection to L2)

Example: nVMX event handling issue

- L0 didn’t re-evaluate pending L1 event because KVM_REQ_EVENT not set
  - IPI received when KVM_REQ_EVENT already set (Before re-injection to L2)
- What if L0 will L2→L1 before resuming L2 with re-injection?



Example: nVMX event handling issue

- L0 didn’t re-evaluate pending L1 event because KVM_REQ_EVENT not set
  - IPI received when KVM_REQ_EVENT already set (Before re-injection to L2)
- What if L0 will L2→L1 before resuming L2 with re-injection?
  ⇒ Queued injected event will be written to vmcs12->idt_vectoring_info

Example: nVMX event handling issue

- L0 didn’t re-evaluate pending L1 event because KVM_REQ_EVENT not set
  - IPI received when KVM_REQ_EVENT already set (Before re-injection to L2)
- What if L0 will L2→L1 before resuming L2 with re-injection?

  ⇒ Queued injected event will be written to vmcs12->idt_vectoring_info

  ⇒ Bug: L1 will see exit on EXTERNAL_INTERRUPT during event-delivery

Example: nVMX event handling issue

- L0 didn’t re-evaluate pending L1 event because KVM_REQ_EVENT not set
  - IPI received when KVM_REQ_EVENT already set (Before re-injection to L2)
- What if L0 will L2→L1 before resuming L2 with re-injection?
  ⇒ Queued injected event will be written to vmcs12->idt_vectoring_info
  ⇒ Bug: L1 will see exit on EXTERNAL_INTERRUPT during event-delivery
- We wish to inject to L2 **and** immediately after re-evaluate L1 pending event

“Immediate exit” mechanism

- Sometimes KVM is blocked from exiting from L2 to L1:
  - E.g. There is an injected event for L2

“Immediate exit” mechanism

- Sometimes KVM is blocked from exiting from L2 to L1:
  - E.g. There is an injected event for L2
- In these cases, we can request an “immediate-exit” from L2 to L0

“Immediate exit” mechanism

- Sometimes KVM is blocked from exiting from L2 to L1:
  - E.g. There is an injected event for L2
- In these cases, we can request an “immediate-exit” from L2 to L0
- “Immediate-exit” requests CPU to exit guest immediately after entering it
  - Set KVM_REQ_EVENT
  - Disable interrupts + self-IPI, just before entering the guest

“Immediate exit” mechanism

- Sometimes KVM is blocked from exiting from L2 to L1:
  - E.g. There is an injected event for L2
- In these cases, we can request an “immediate-exit” from L2 to L0
- “Immediate-exit” requests CPU to exit guest immediately after entering it
  - Set KVM_REQ_EVENT
  - Disable interrupts + self-IPI, just before entering the guest
- CPU will inject event and then immediately exit on EXTERNAL_INTERRUPT
- When exit back to L0, re-evaluate L1 pending event

Example: nVMX event handling issue

- Miss of L1 IPI when on L2→L0 exit during interrupt-delivery
  - L1 stuck as a result of losing an IPI while holding KVM mmu_lock
- Root-cause: Reinjection of L2 events blocked evaluation of L1 pending events
- 1a680e355c94 (“KVM: nVMX: Require immediate-exit when event reinjected to L2 and L1 event pending”)

Recent nVMX improvements

Mechanisms fixes

- nVMX event-injection fixes
  - Missing L1 events, SMI while in guest-mode, handling L1 not intercepting interrupts
  - TODO: Ability to get/set vCPU events considering pending/injected
  - TODO: Keep CR2/DR6 unmodified if #PF/#DB intercepted by L1
    - Jim Mattson series to handle all mentioned above: https://patchwork.kernel.org/project/kvm/list/?series=31593
  - TODO: L2 pending trap exceptions can still be lost because of L2→L1 transition
- Nested APICv fixes
  - Nested posted-interrupts race-condition and EOI-exitmap corruption
  - Enabled running **ESXi as L1 hypervisor** with APICv enabled
  - TODO: Use hardware for re-evaluation of missed nested posted-interrupts
    - https://patchwork.kernel.org/patch/10132081/
    - https://patchwork.kernel.org/patch/10132083/

Mechanisms fixes

- Nested VPID fixes & optimizations
  - Invalidation of wrong TLB mappings and unnecessary TLB flushes
- Nested MMU optimizations
  - L1<->L2 transitions avoid MMU unload, fast switch EPTP and L1/L2 separate MMU contextes
- L1→L2 VMEntry optimizations
  - Dirty-track non-shadowed VMCS fields, faster build of vmcs02 MSR bitmap, optimize shadow VMCS copying
- Exposure of VMX features to guest fixes
  - Affected by CPU features, KVM module parameters and guest CPUID!
- More L1→L2 VMEntry checks

New mechanism: nVMX Migration support

- KVM holds internal CPU state for running L2
  - VMXON region address, active vmcs12 address
  - Cached vmcs12 & cached shadow vmcs12
  - Internal nested flags (E.g. nested_run_pending)
- Required IOCTLs to save/restore this state for migration
  - KVM_{GET,SET}_NESTED_STATE
- Required ability to set VMX MSRs from userspace
- TODO: QEMU patches for supporting this
  - https://patchwork.kernel.org/patch/10601689/

New mechanism: VMCS Shadowing virtualization

- Use-case: Triple-Virtualization!
- Accelerate L2 VMREADs/VMWRITEs

New mechanism: VMCS Shadowing virtualization

- Use-case: Triple-Virtualization!
- Accelerate L2 VMREADs/VMWRITEs
- However, L3 performance still insufficient for Oracle's production workloads
  - See appendix slides for details
- TODO: Optimizations for VMCS Shadowing virtualization
  - Avoid building vmcs02→{vmread,vmwrite}_bitmap when vmcs12 bitmaps unchanged
  - Cache CPU unsupported VMCS fields on KVM boot-time
  - Get rid of cached shadow vmcs12

Recent improvements have led to...

kvm-intel.nested=1 default on kernel 4.20!

https://patchwork.kernel.org/patch/10644311/

Future directions of nVMX

- Microsoft Hyper-V improved nested perf by PV interface (eVMCS)

Future directions of nVMX

- Microsoft Hyper-V improved nested perf by PV interface (eVMCS)
- KVM should have it’s own PV interface for nested
  - eVMCS too coupled with Hyper-V PV interface

Future directions of nVMX

- Microsoft Hyper-V improved nested perf by PV interface (eVMCS)
- KVM should have it’s own PV interface for nested
  - eVMCS too coupled with Hyper-V PV interface
- Future of supporting all combinations of L0/L1 hypervisors PV interfaces?
  - E.g. KVM was recently enhanced to be able to both use and expose Hyper-V eVMCS
- Should there be cross-hypervisor PV standard for nested-virtualization?

Conclusion

Conclusion

- Many nVMX advancements over past year which we don’t have time for…
  - **Appendix slides contain deep-dive to some of those mechanisms for reference!**
- nVMX semantics stabilized very well over the past year
  - Thanks to multiple contributes: Google, AWS, Intel, RedHat, Oracle and more
  - Most semantic issues discovered by running various hypervisors as L1
- kvm-unit-tests VMX tests cover mainly edge cases and regression tests
- Challenges ahead: PV for nested & Triple-Virtualization

Questions?

Thank you!

Appendix

Things I wish I had time to present… :)

Appendix Outline

- VMCS Shadowing
- Triple-Virtualization
- Nested APICv
- nVMX event injection

VMCS Shadowing

VMCS cache

- On VMPTRLD, VMCS loaded to CPU is cached in per-CPU VMCS cache
- Access to VMCS done by dedicated instructions: VMREAD/VMWRITE

VMCS cache

- On VMPTRLD, VMCS loaded to CPU is cached in per-CPU VMCS cache
- Access to VMCS done by dedicated instructions: VMREAD/VMWRITE

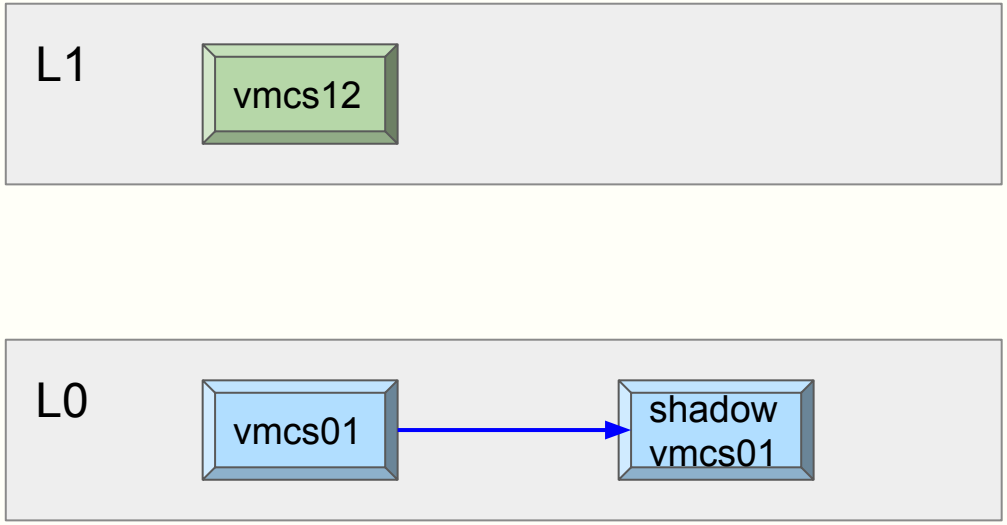
nVMX implementation:

- L0 VMPTRLD emulation copy in-memory vmcs12 into software cache
- L0 VMREAD/VMWRITE emulation read/write from/to cached vmcs12
  ⇒ Exits on L1's VMREAD/VMWRITE are significant performance hit

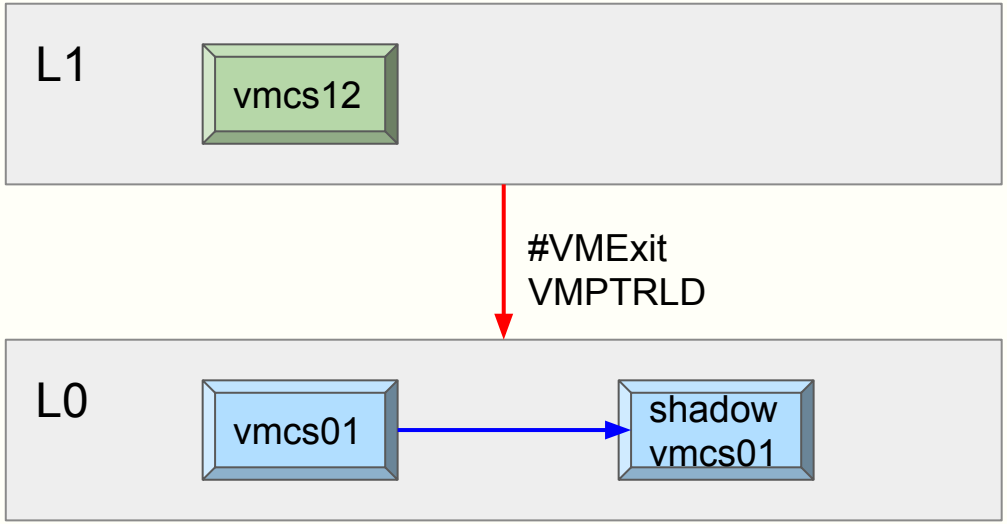
VMCS Shadowing

- Hardware VMX feature to improve nVMX performance
- Reduce #VMExits on L1 VMREAD/VMWRITE
- VMCS->vmcs_link_ptr points to “shadow vmcs”
- L1 VMREAD/VMWRITE directed to “shadow vmcs”
  - According to VMCS->{vmread,vmwrite}_bitmap

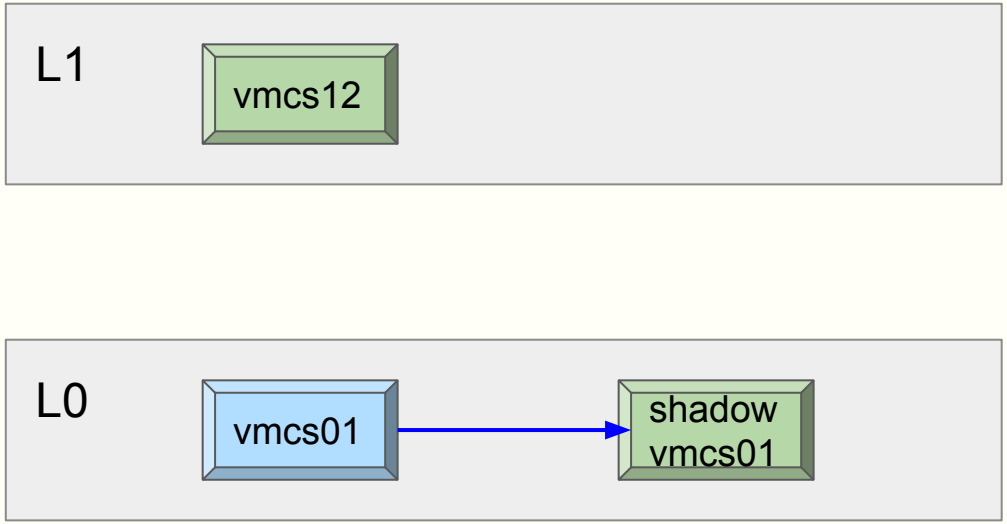
VMCS Shadowing usage



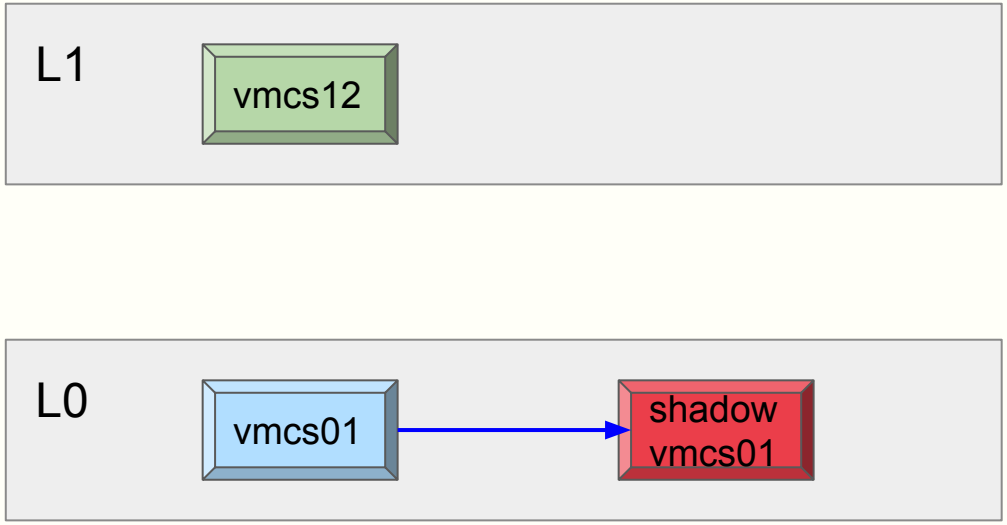
L1 VMPTRLD vmcs12



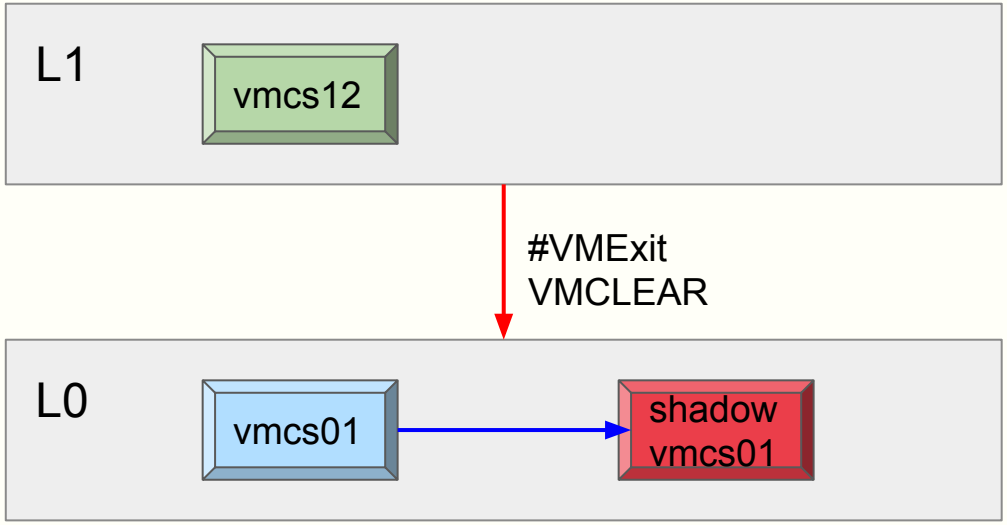
L0 copies vmcs12 to shadow vmcs01



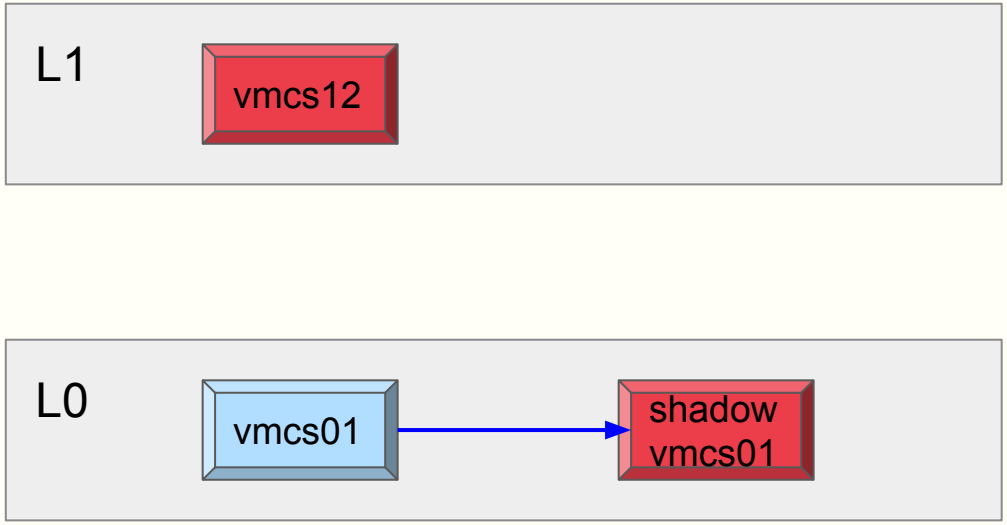
L1 VMREAD/VMWRITE read/write shadow vmcs01



L0 VMCLEAR vmcs12



L0 copies shadow vmcs01 to vmcs12



VMCS Shadowing usage (Many copies…)

- On L1 VMPTRLD, L0 copies cached vmcs12 to shadow vmcs01
  - L1 will read values using VMREAD from shadow vmcs01
- On L1 VMCLEAR, L0 copies shadow vmcs01 to cached vmcs12
  - L1 may have modified shadow vmcs01 using VMWRITE
- On L1→L2 transition, L0 copies shadow vmcs01 to cached vmcs12
  - L1 has written values using VMWRITE to shadow vmcs01
- On L2→L1 transition, L0 copies cached vmcs12 to shadow vmcs01
  - L1 will read values using VMREAD from shadow vmcs01

Triple-Virtualization!

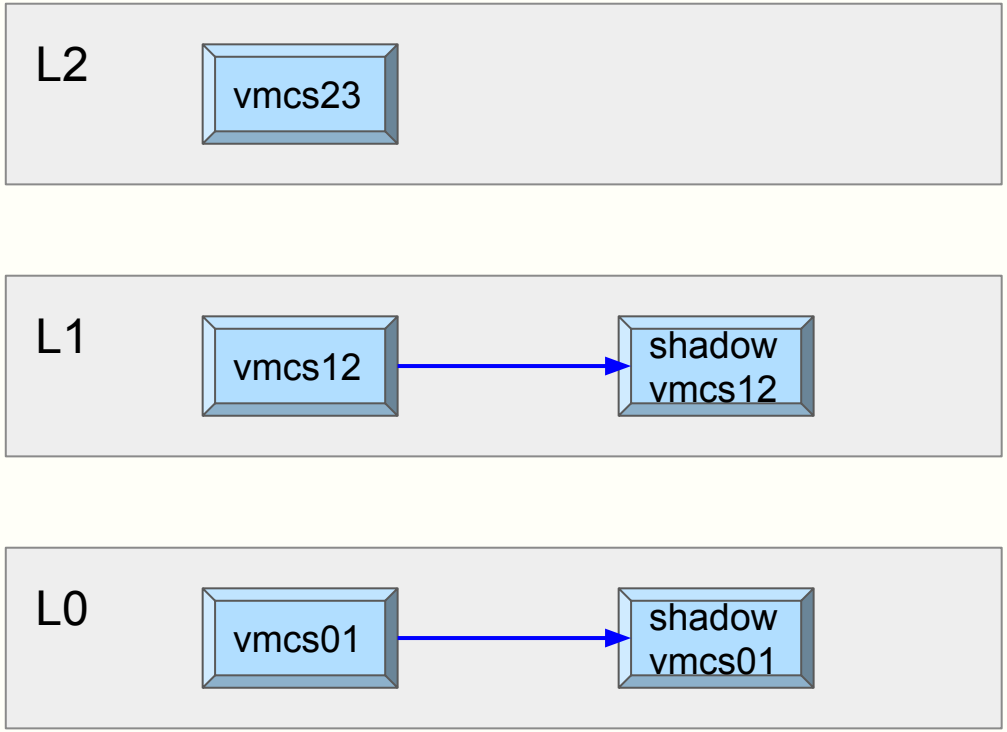
Triple-Virtualization: Why?!

- Some of Oracle Ravello guests are hypervisors themselves…
  - ESXi, KVM, Xen, Hyper-V
- Therefore, setup is:
  - L0 = Public cloud provider hypervisor
  - L1 = Ravello's hypervisor (KVM based)
  - L2 = Ravello guest which is a hypervisor (e.g. ESXi)
  - L3 = L2 guests
- We are dealing with a Triple-Virtualization scenario!

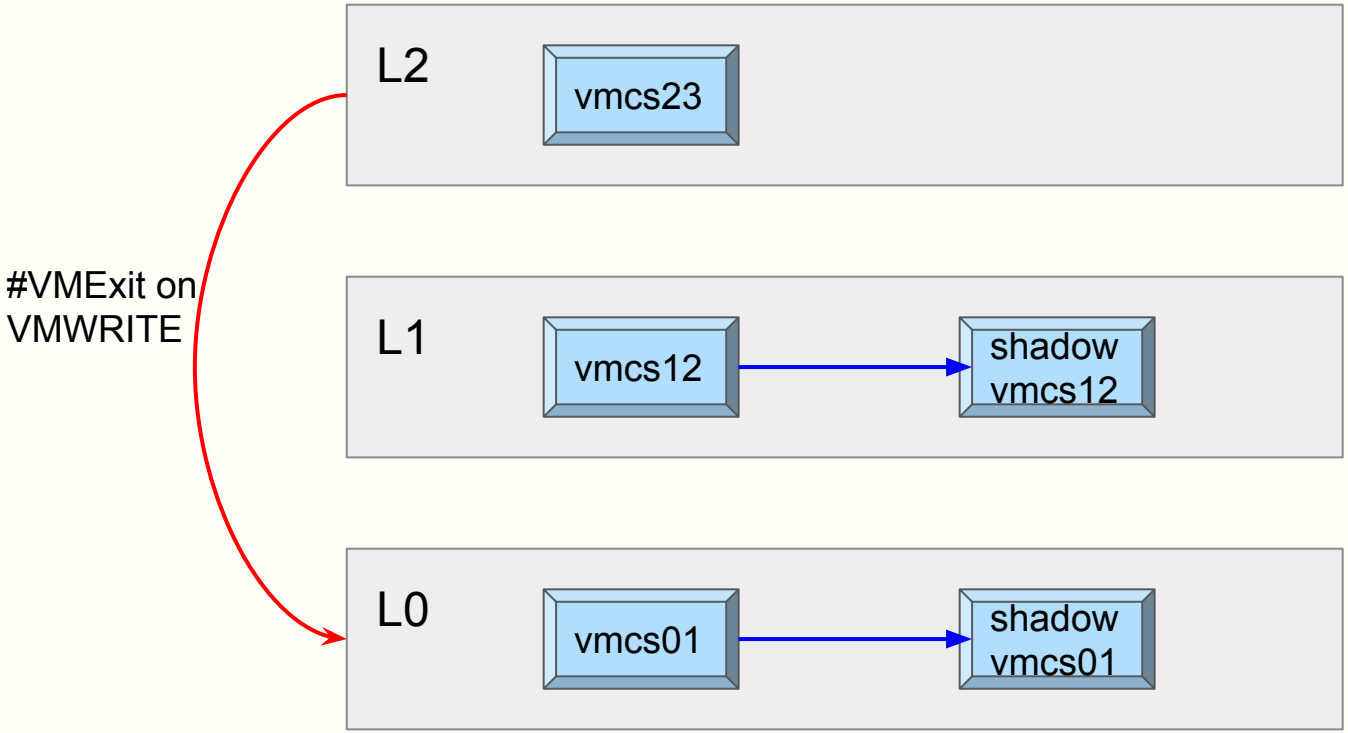
Triple-Virtualization: VMCS Shadowing Virtualization

- Triple-Virtualization works but...
- L2 will execute VMREADs / VMWRITEs
- They will perform extremely poorly unless L1 is utilizing VMCS Shadowing
  ⇒ We need L0 to support VMCS Shadowing Virtualization!
- Lead us to contact Jim Mattson to implement this in GCE L0 KVM
- Jim developed the patches and I have further modified them for upstream
- https://www.spinics.net/lists/kvm/msg170724.html

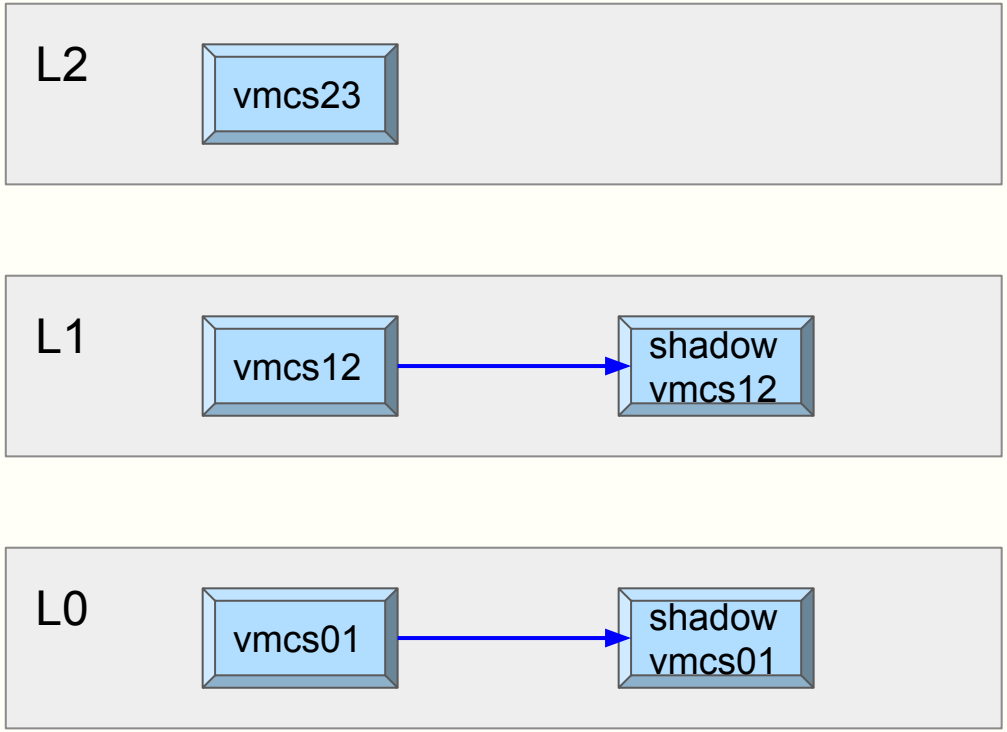
VMCS Shadowing Emulation



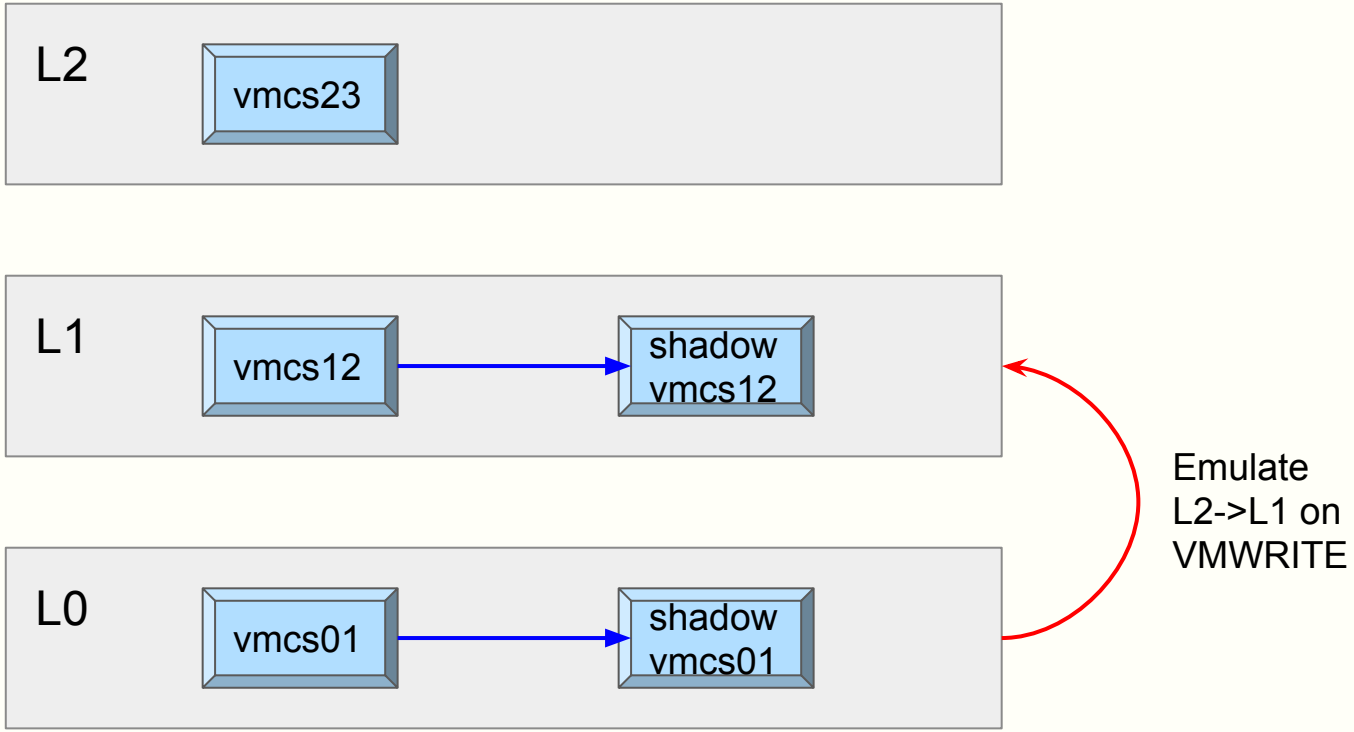
L2 executes VMWRITE



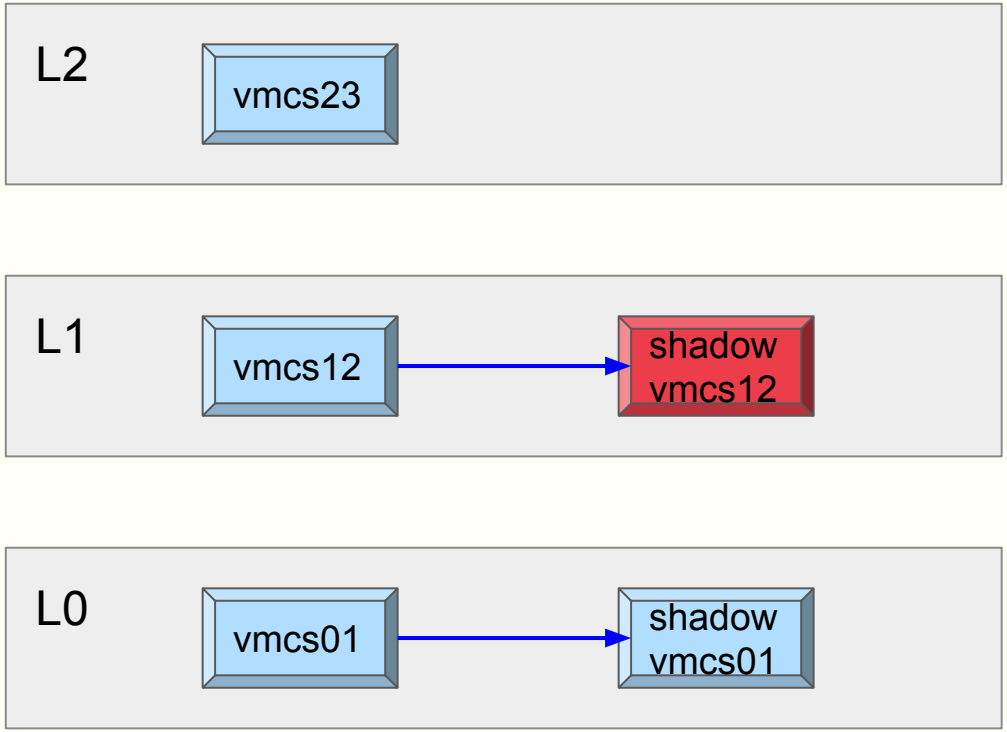
L0 examins vmcs12→vmwrite_bitmap



Option 1: VMWRITE intercepted by L1 by bitmap



Option 2: L0 emulates VMWRITE on shadow vmcs12

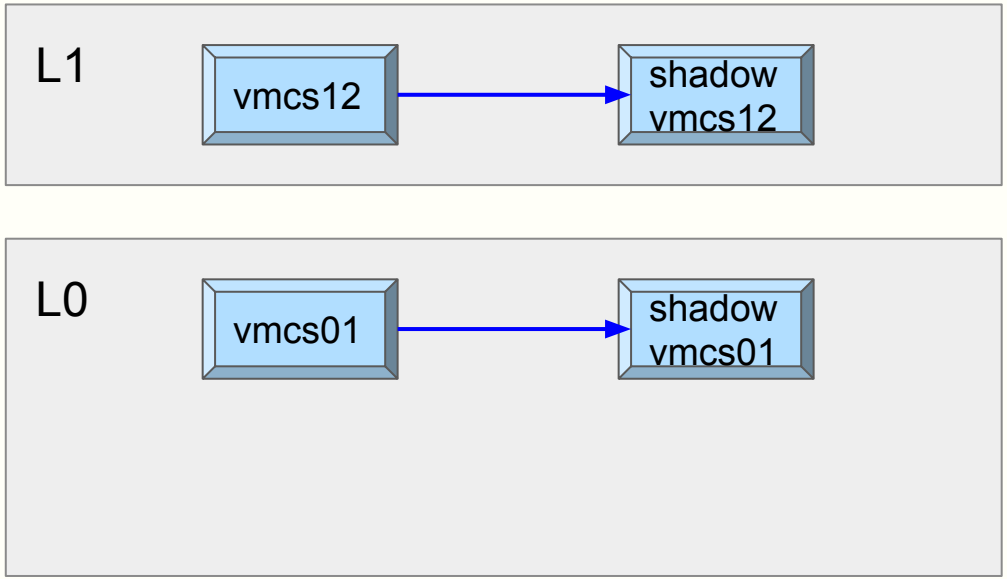


VMCS Shadowing Emulation

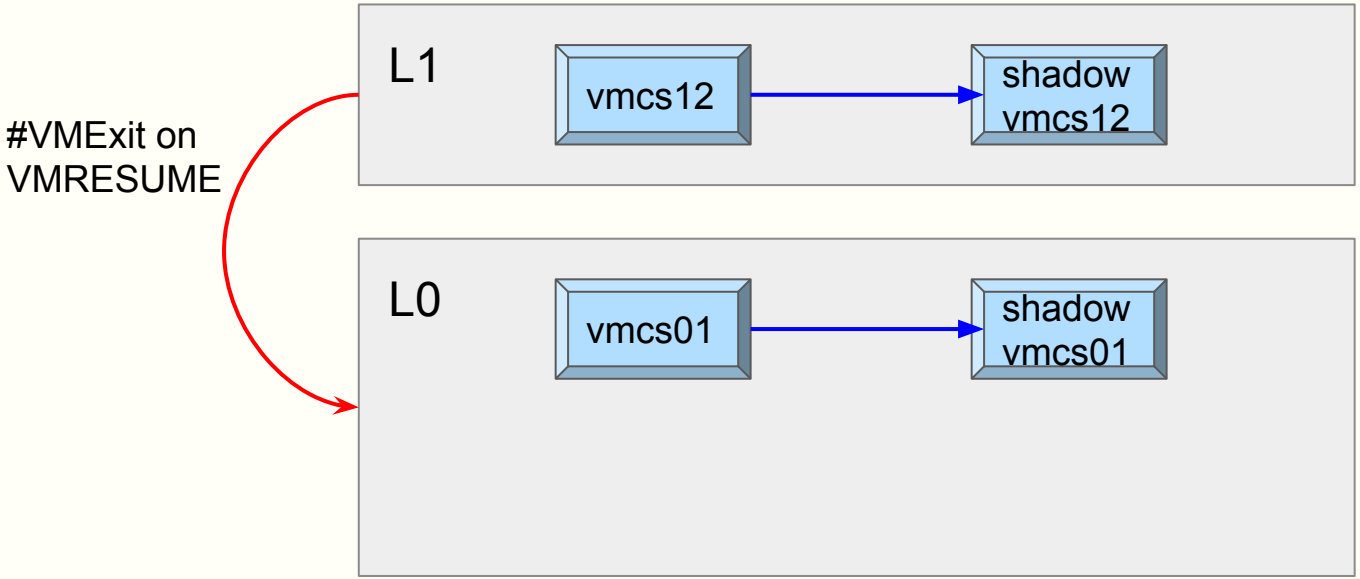
- All L2 VMREAD/VMWRITE still exit to L0
- Reflect exit on VMREAD/VMWRITE to L1 based on vmcs12→ {vmread,vmwrite}_bitmap
- Modify L0 VMREAD/VMWRITE exit handlers to write to cached shadow vmcs12 instead of cached vmcs12 if vCPU in guest-mode
- Cache shadow vmcs12
  - L1→L2: Copy from vmcs12→vmcs_link_ptr to shadow VMCS12 cache
  - L2→L1: Flush shadow vmcs12 cache to guest vmcs12→vmcs_link_ptr
- 32c7acf04487 (“KVM: nVMX: Expose VMCS shadowing to L1 guest”)

⇒ Saves exits to L1 on L2’s VMREADs/VMWRITEs!

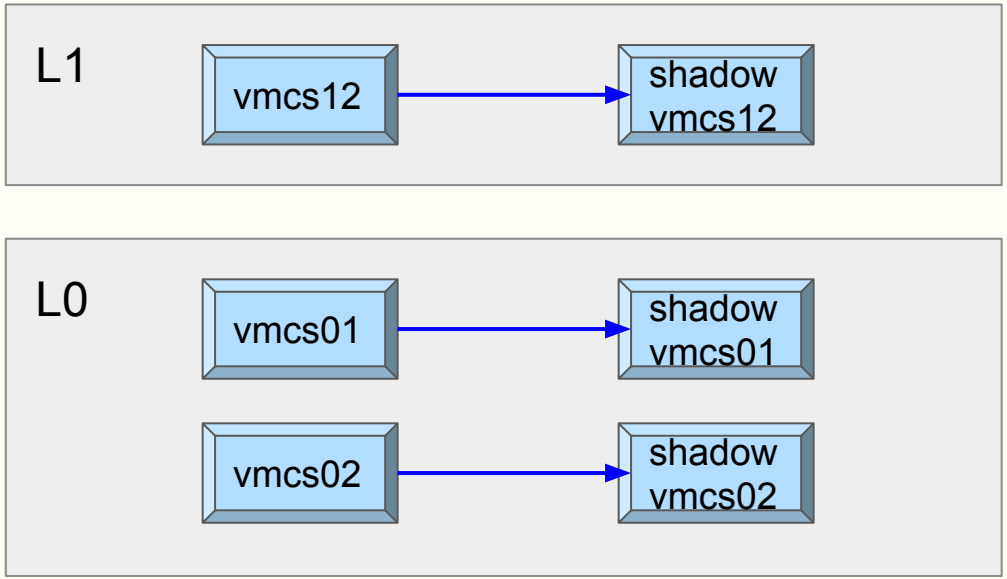
VMCS Shadowing Virtualization



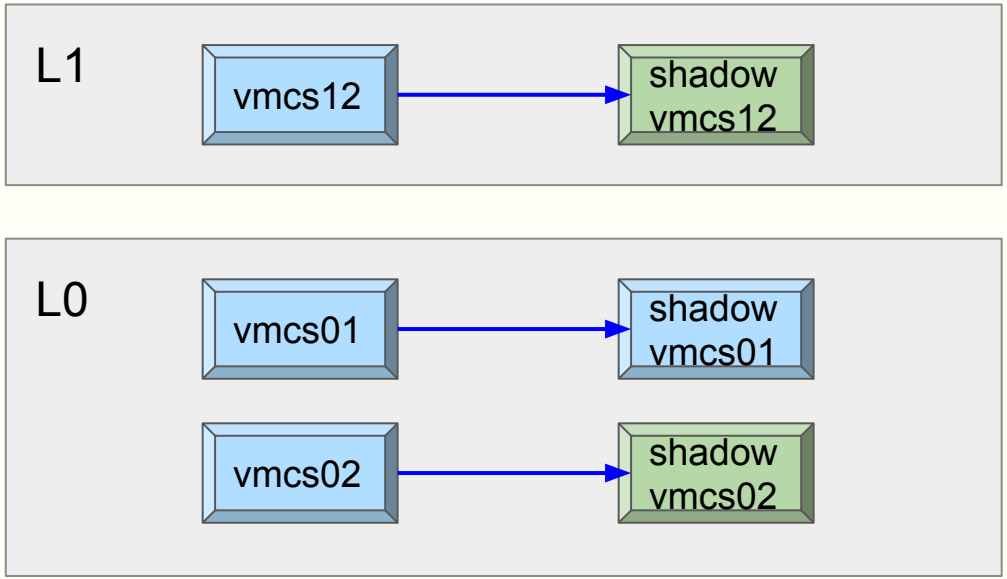
L1 executes VMRESUME



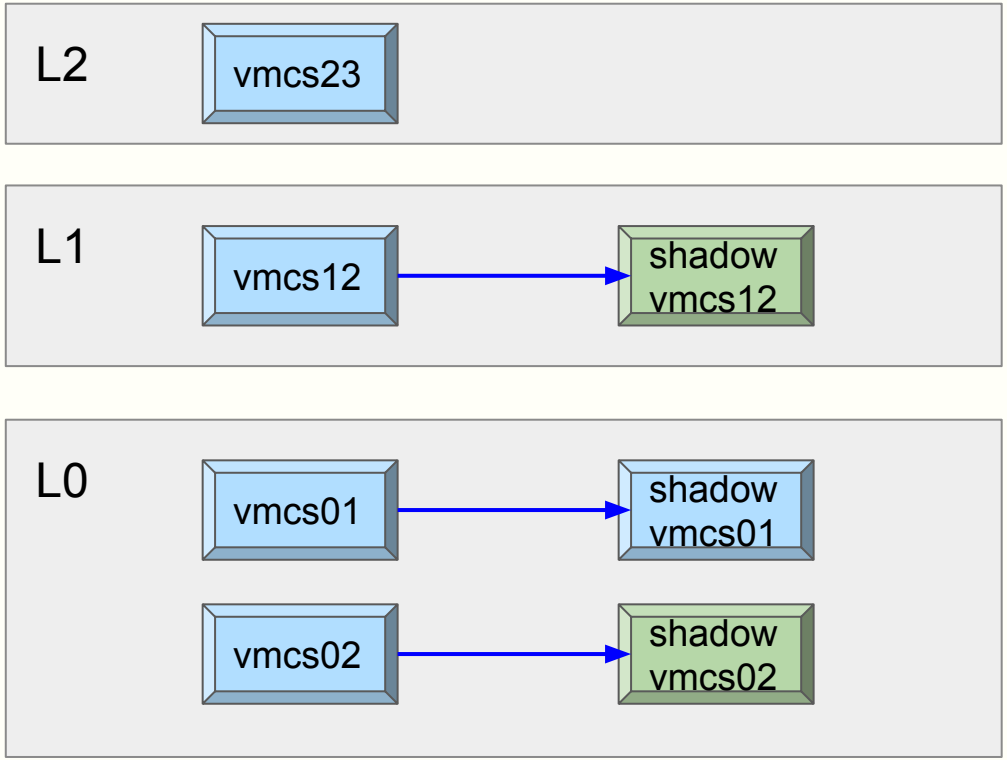
L0 creates vmcs02 with shadow vmcs



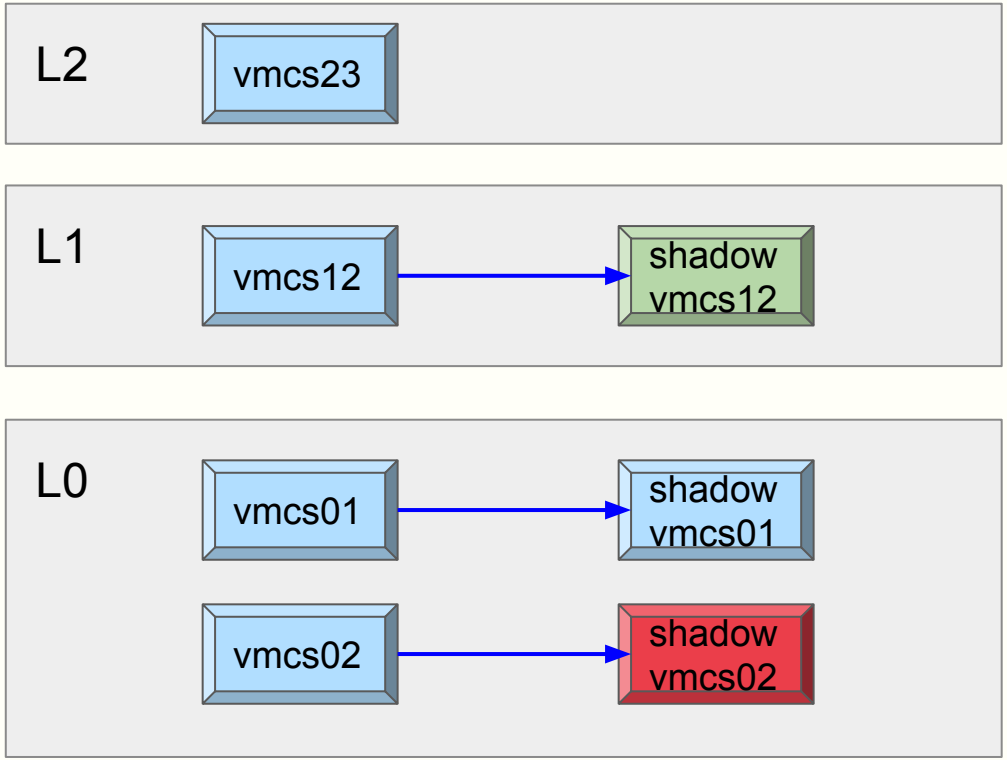
L0 copies shadow vmcs12 to shadow vmcs02



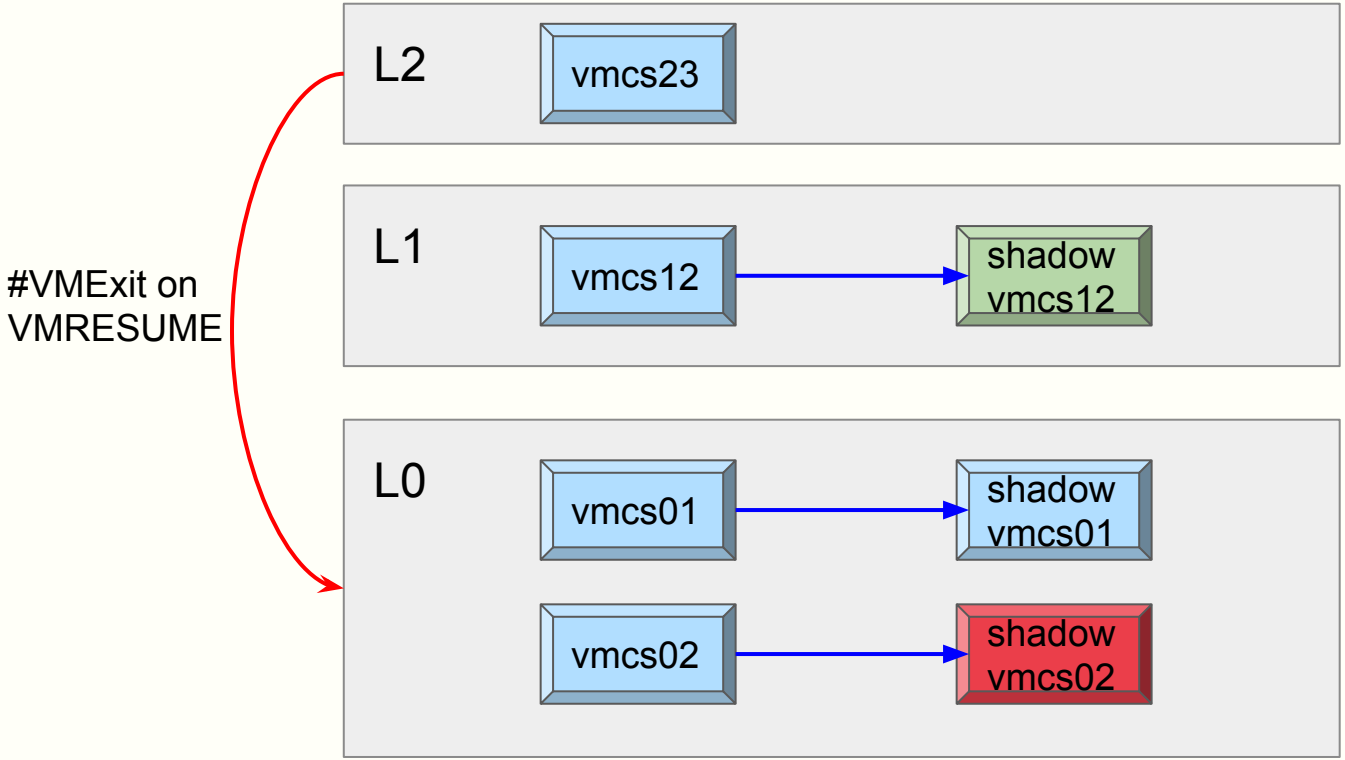
L0 executes L2 with vmcs02



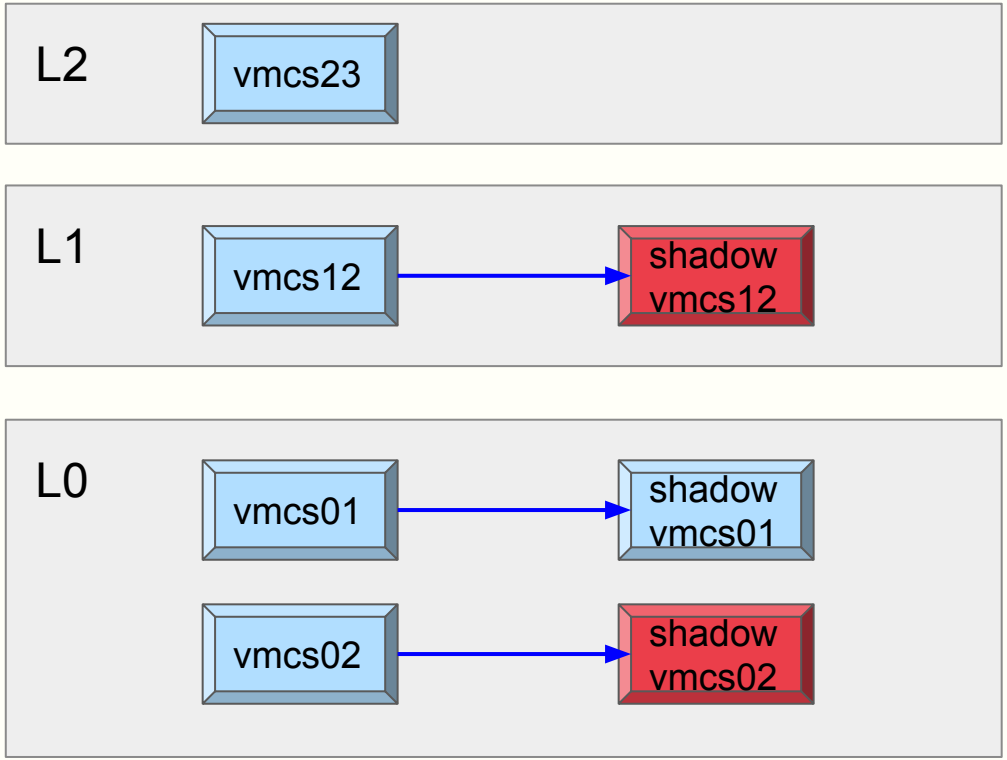
L2 executes VMWRITE writes to shadow vmcs02



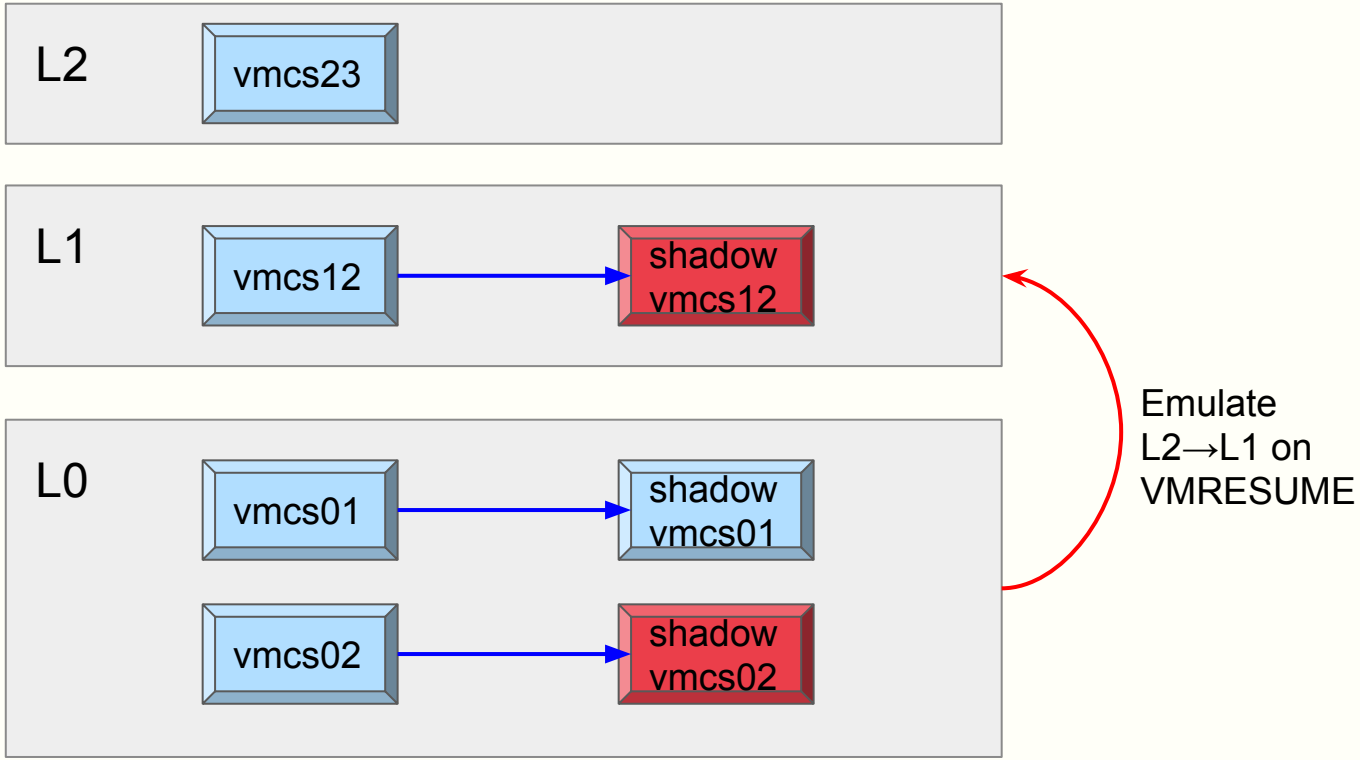
L2 executes VMRESUME



L0 copies shadow vmcs02 to shadow vmcs12



L0 forward exit on VMRESUME to L1



VMCS Shadowing Virtualization

- Allocate shadow VMCS and {vmread,vmwrite}_bitmap for vmcs02
- vmcs02→{vmread,vmwrite}_bitmap based on vmcs12 bitmaps
  - Not identical as unsupported VMCS fields by L0 are still intercepted
- On L1→L2, copy cached shadow vmcs12 to shadow vmcs02
- On L2→L1, copy shadow vmcs02 to cached shadow vmcs12
- Not applied yet. v1 of patch series: https://www.spinics.net/lists/kvm/msg170724.html

⇒ Saves exits to both L0 & L1 on L2's VMREADs/VMWRITEs!

VMCS Shadowing usage: Many copies…

- On L1→L2 transition, L0 copies shadow vmcs01 to cached vmcs12
- On L2→L1 transition, L0 copies cached vmcs12 to shadow vmcs01
- On L1 VMPTRLD, L0 copies cached vmcs12 to shadow vmcs01
- On L1 VMCLEAR, L0 copies shadow vmcs01 to cached vmcs12

VMCS Shadowing Emulation: More copies...

- On L1→L2 transition, L0 copies shadow vmcs01 to cached vmcs12
  And cache shadow vmcs12
- On L2→L1 transition, L0 copies cached vmcs12 to shadow vmcs01
  And flush cached shadow vmcs12 to shadow vmcs12
- On L1 VMPTRLD, L0 copies cached vmcs12 to shadow vmcs01
- On L1 VMCLEAR, L0 copies shadow vmcs01 to cached vmcs12

VMCS Shadowing Virtualization: Even more copies!

- On L1→L2 transition, L0 copies shadow vmcs01 to cached vmcs12
  And cache shadow vmcs12
  And build vmcs02→{vmread,vmwrite}_bitmap from vmcs12 bitmaps
  And copy cached shadow vmcs12 to shadow vmcs02
- On L2→L1 transition, L0 copies cached vmcs12 to shadow vmcs01
  And copy shadow vmcs02 to cached shadow vmcs12
  And flush cached shadow vmcs12 to shadow vmcs12
- On L1 VMPTRLD, L0 copies cached vmcs12 to shadow vmcs01
- On L1 VMCLEAR, L0 copies shadow vmcs01 to cached vmcs12

Triple-Virtualization insufficient performance

- L3→L2 results in a total of **15 copies!**
  - Of shadow vmcs to cached vmcs (/ cached shadow vmcs) and vice-versa

Triple-Virtualization insufficient performance

- L3→L2 results in a total of **15 copies!**
  - Of shadow vmcs to cached vmcs (/ cached shadow vmcs) and vice-versa

- **Key Observations:**
  1. All transitions from/to Lx involves all underlying layers
  2. Lx<->Ly involve copies of shadow vmcs to cached vmcs and vice-versa
  3. Copies of shadow VMCS requires VMPTRLD which incur #VMExit

Triple-Virtualization possible solutions

- Should create a KVM PV interface for nested? Similar to Hyper-V?
- Should there be cross-hypervisor PV standard for nested-virtualization?
  - Supporting all combinations of L0/L1 hypervisors PV interfaces is complex…
  - KVM was recently enhanced to be able to both use and expose Hyper-V eVMCS
- Can we suggest a new VMX feature for Intel to improve triple-virtualization?
  - Ability to read/write from/to shadow VMCS without VMPTRLD to make it active?

* Note: Deeper perf analysis wasn't performed yet

Triple-Virtualization!

Technical details

VMCS Shadowing Virtualization: L3→L2

- L3 exits to L0 which decides to forward exit to L1
  - L0 copy cached shadow vmcs12 to shadow vmcs12
  - L0 copy cached vmcs12 to shadow vmcs01
- L0 resume into L1 which decides to forward exit to L2
  - L1 copy cached shadow vmcs23 to shadow vmcs23
  - L1 VMPTRLD vmcs12 which exit to L0
    - L0 copy shadow vmcs01 to cached vmcs12
    - L0 copy vmcs12 to cached vmcs12
    - L0 copy vmcs12 to shadow vmcs02
  - L1 copy cached vmcs23 to shadow vmcs12
    - L1 VMPTRLD shadow vmcs12 which exit to L0
      - L0 does 3 copies…
    - L1 copies VMCS fields
    - L1 VMCLEAR shadow vmcs12 which exit to L0
      - L0 copies shadow vmcs01 to cached vmcs12
    - L1 VMPTRLD vmcs12
      - L0 does 3 copies...

VMCS Shadowing Virtualization: L3→L2 (continue..)

- L1 resume into L2 exit to L0
  - L0 copy shadow vmcs01 to cached vmcs12
  - L0 copy shadow vmcs12 into cached shadow vmcs12
- And that's it! :)

Total of 15 copies!

(And we haven't counted L2→L3…)

Triple-Virtualization using Binary-Translation

- HVX == Oracle Ravello binary-translation hypervisor
- **L1 binary translation results in L3→L2 not involving L0**
- Performance test setup:
  - L0 = GCE_KVM
  - L1 = KVM / HVX (Haswell, 4 vCPUs, 26GB memory)
  - L2 = KVM (8 vCPUs, 16GB memory)
  - L2 is running 2 Ubuntu 16.04 guests as L3 (4 vCPUs, 8GB memory)
- netperf between L3 guests:
  - HVX performs **~4x** better than KVM both in throughput and latency
- Sysbench:
  - HVX performs **~2x** better then KVM

Triple-Virtualization using eVMCS

- We can avoid shadow VMCS performance hit with PV interfaces
- Hyper-V eVMCS mechanism helps
- L0 emulate Hyper-V PV interface with eVMCS and L1 will consume it
- Probably was first one to run such a setup…
  - Created patches for L0 QEMU to expose eVMCS
  - Fixed bug: 2307af1c4b2e (“KVM: VMX: Mark VMXArea with revision_id of physical CPU even when eVMCS enabled”)
- Not a real solution for the general case
  - Works only if L1 knows how to use eVMCS...
  - We don’t really want to expose Hyper-V PV interface for L1
- TODO: Collect concrete perf numbers

Nested APICv

How APICv works?

- Generic name for combination of APIC related VMX features
- Aim to reduce #VMExits because of APIC and interrupts virtualization

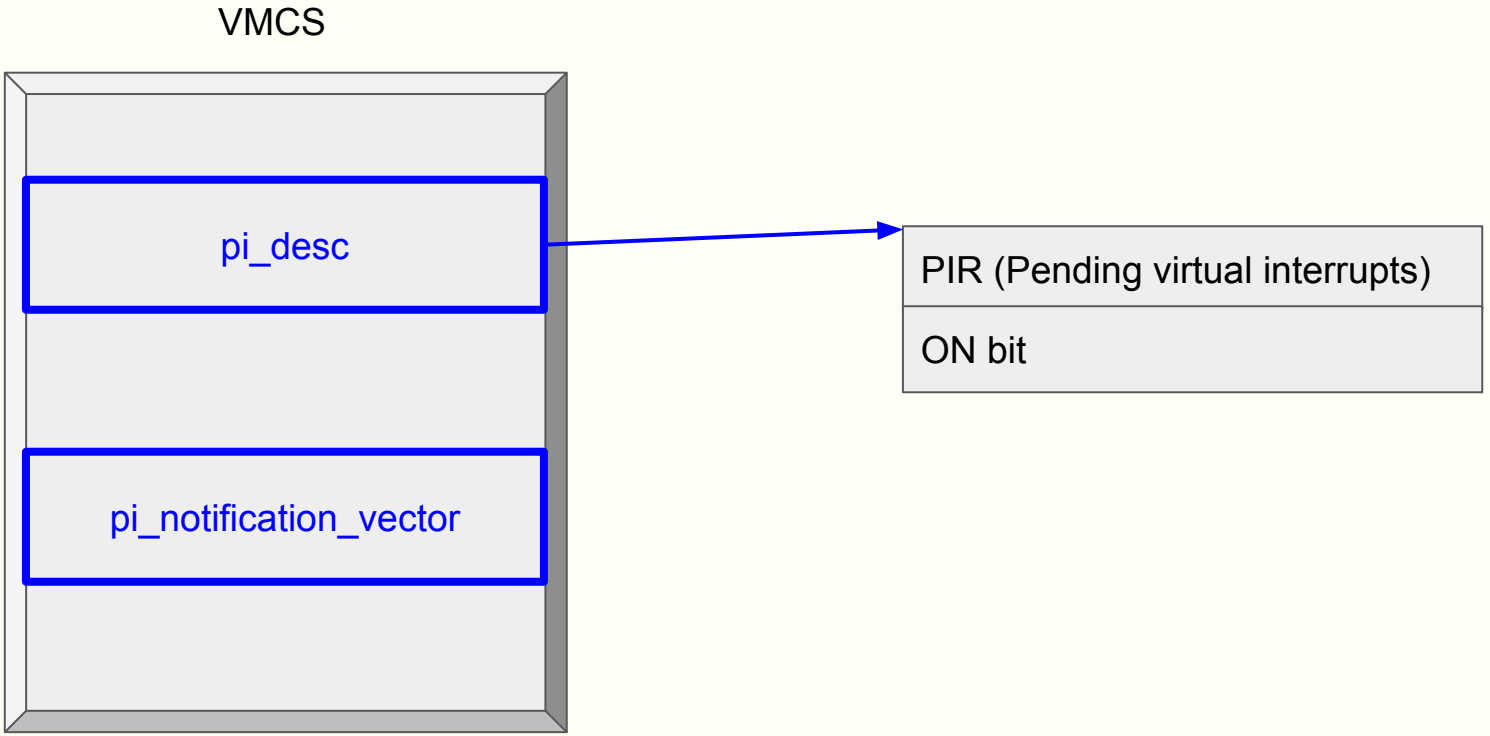
How APICv works?

- APIC {access,register} Virtualization:
  CPU emulates read/write from/to vLAPIC without #VMExit

- Virtual interrupt delivery:
  CPU emulates LAPIC interrupt evaluation and delivery without #VMExit

- Posted-Interrupts:
  Post interrupt to another CPU without #VMExit target CPU

How posted-interrupts work?



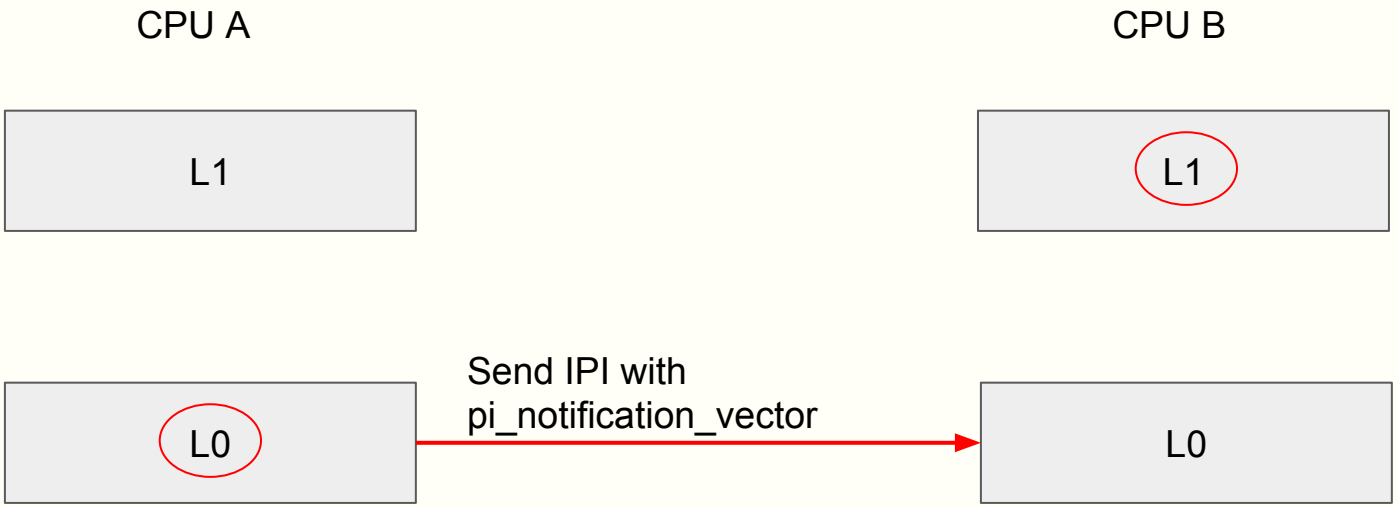
How posted-interrupts work?

How posted-interrupts work?

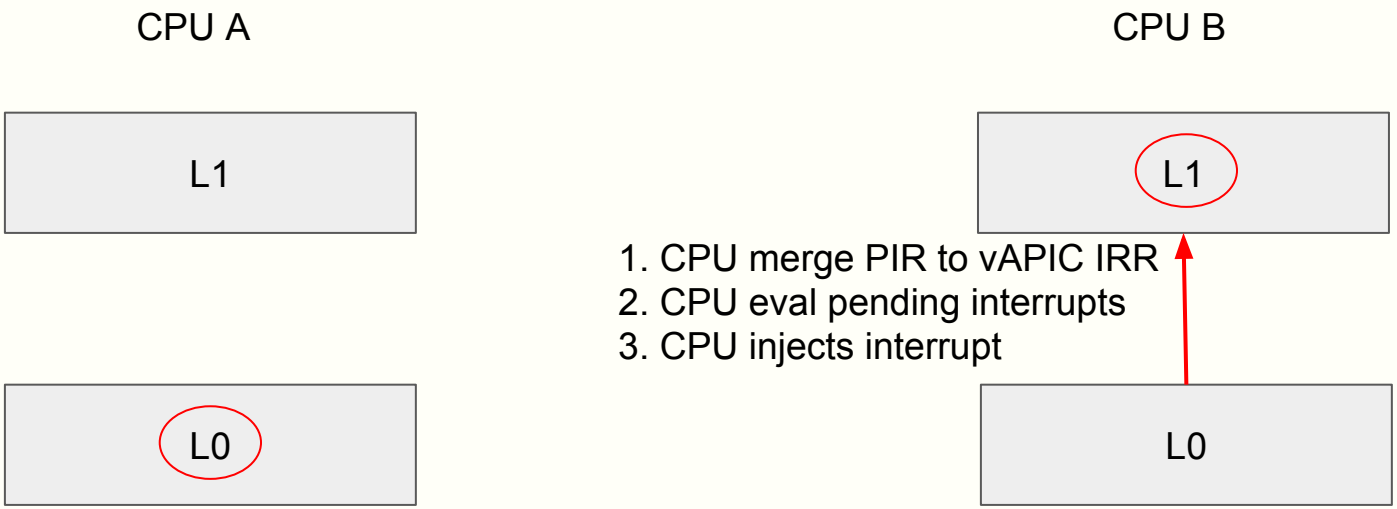
How posted-interrupts work?

1. Set bit in pi_desc→PIR
2. Set pi_desc→ON

How posted-interrupts work?



How posted-interrupts work?



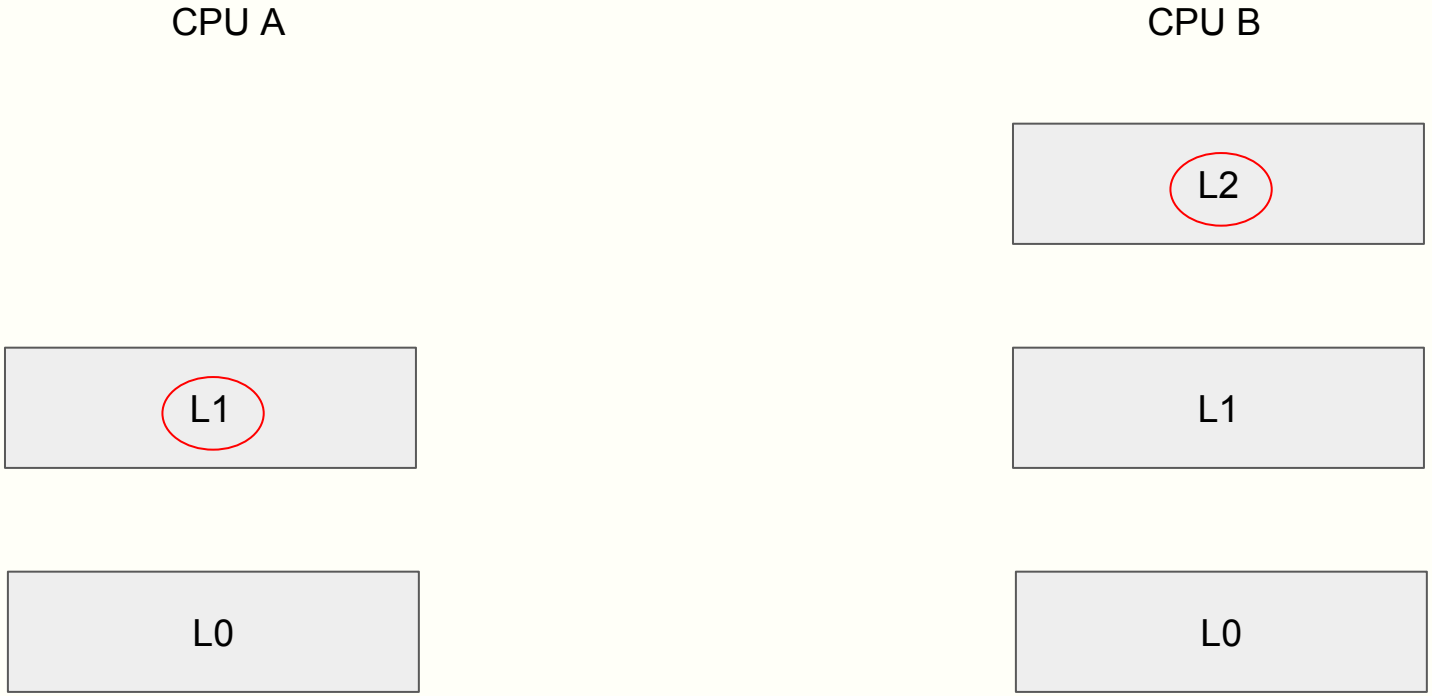
How posted-interrupts works?

- What if target CPU currently at L0?
- L0 needs to evaluate pending posted-interrupts in software

How posted-interrupts works?

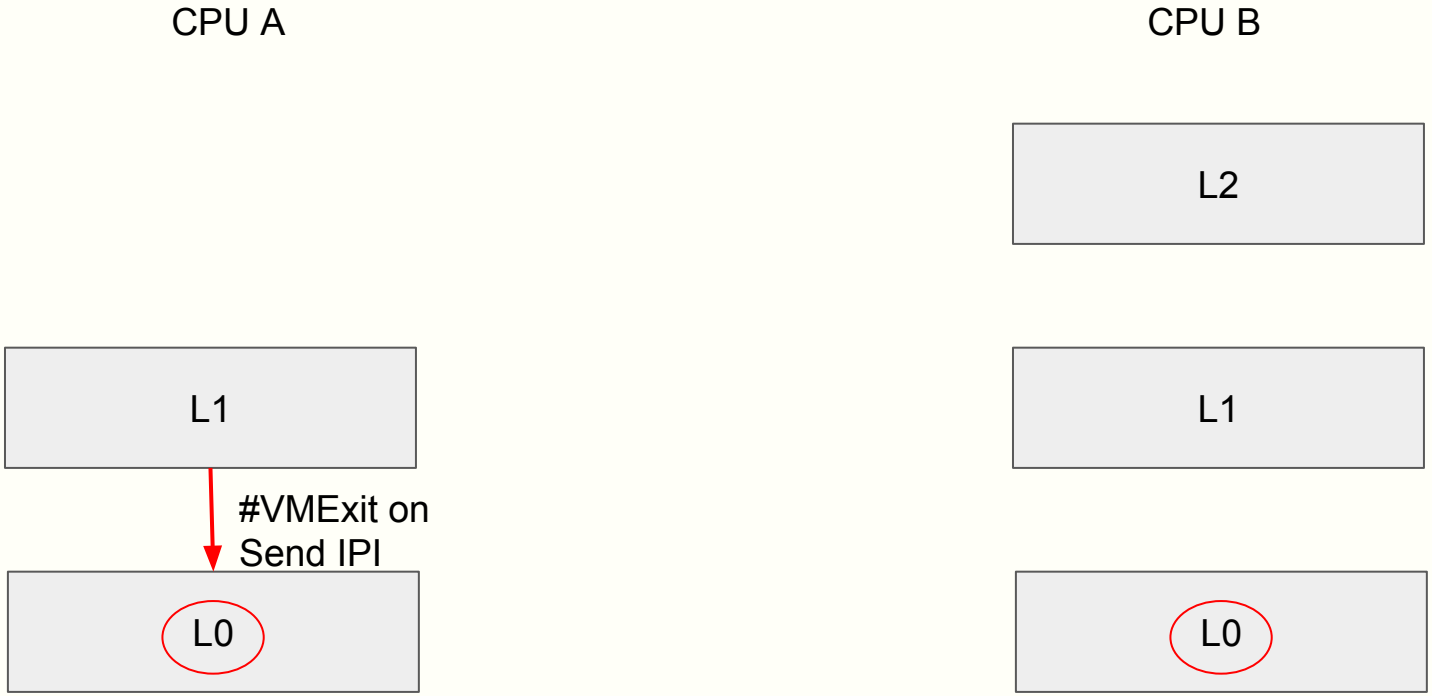
- What if target CPU currently at L0?
- L0 needs to evaluate pending posted-interrupts in software
  ⇒ Before each entry to guest, sync PIR to LAPIC IRR

Example: nVMX posted-interrupt issue

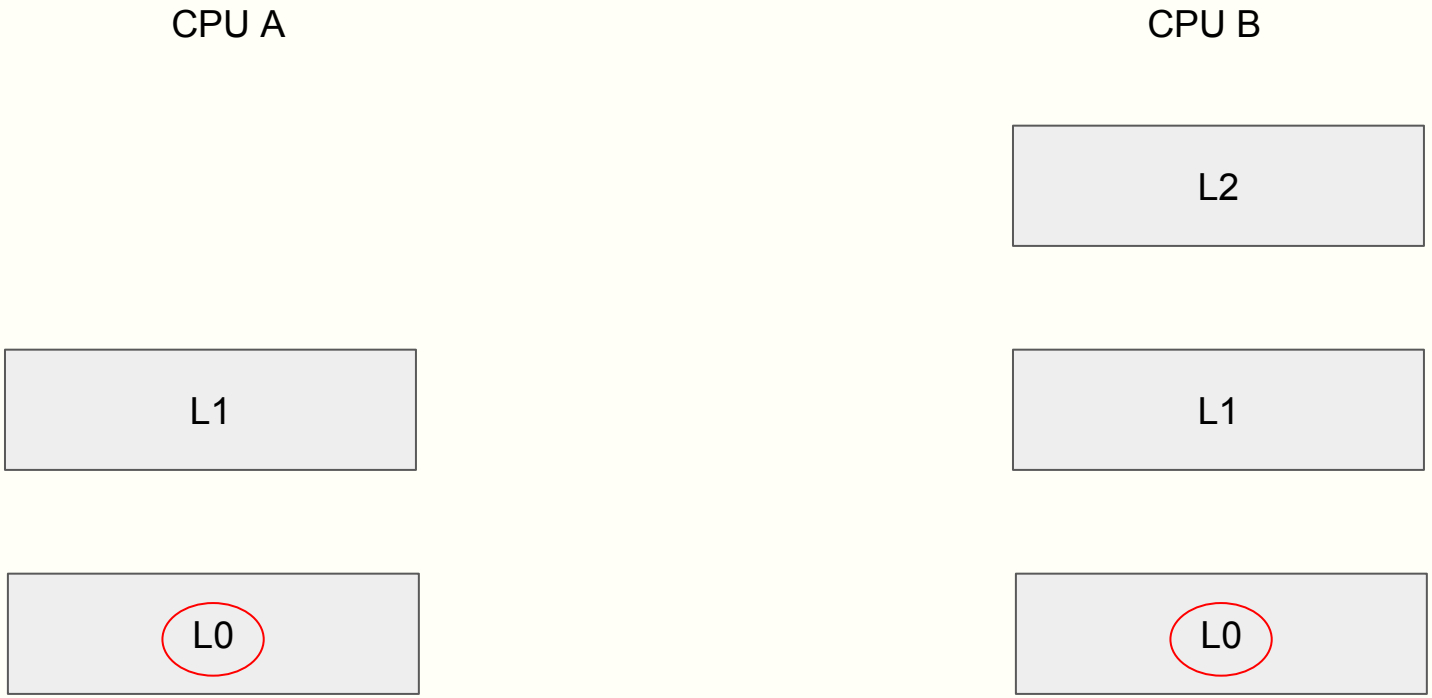


Example: nVMX posted-interrupt issue

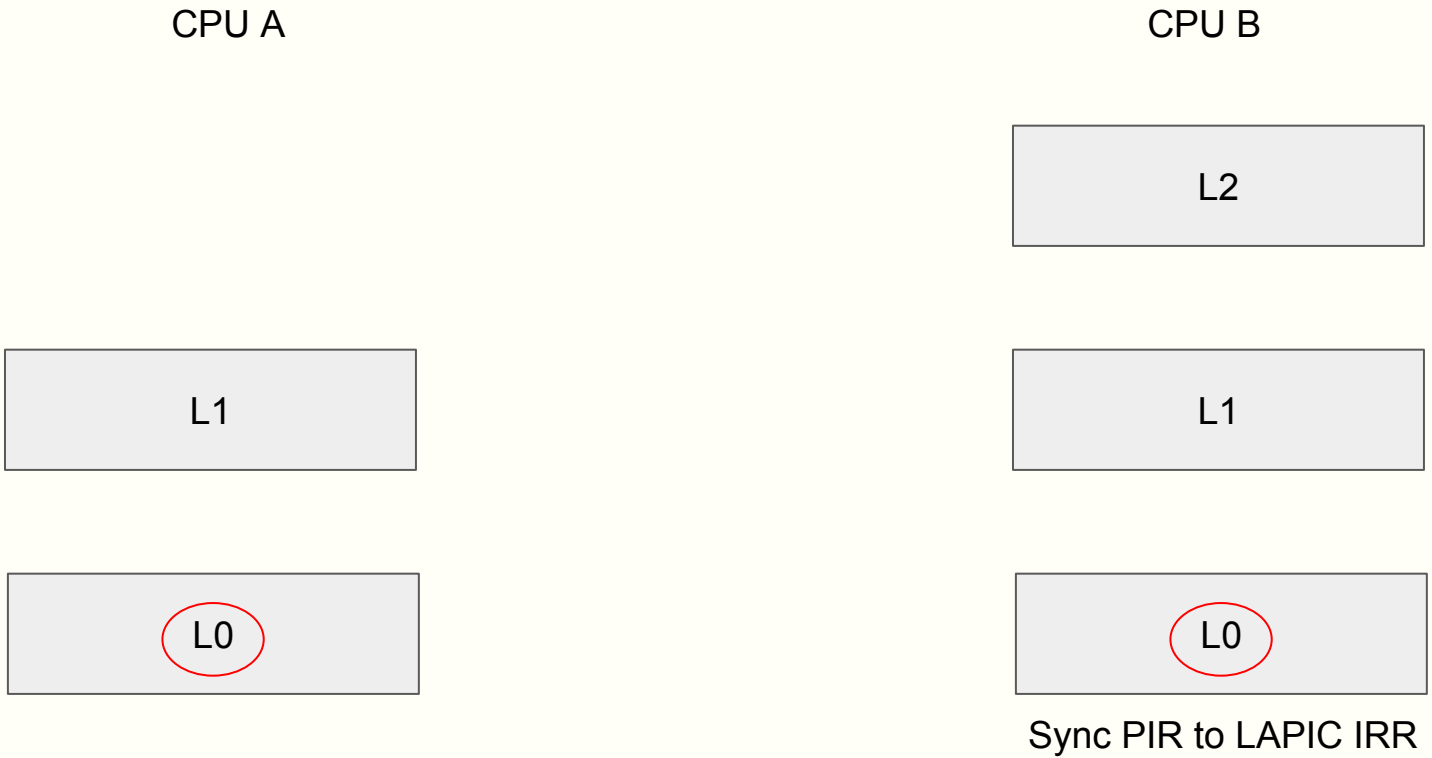
Example: nVMX posted-interrupt issue



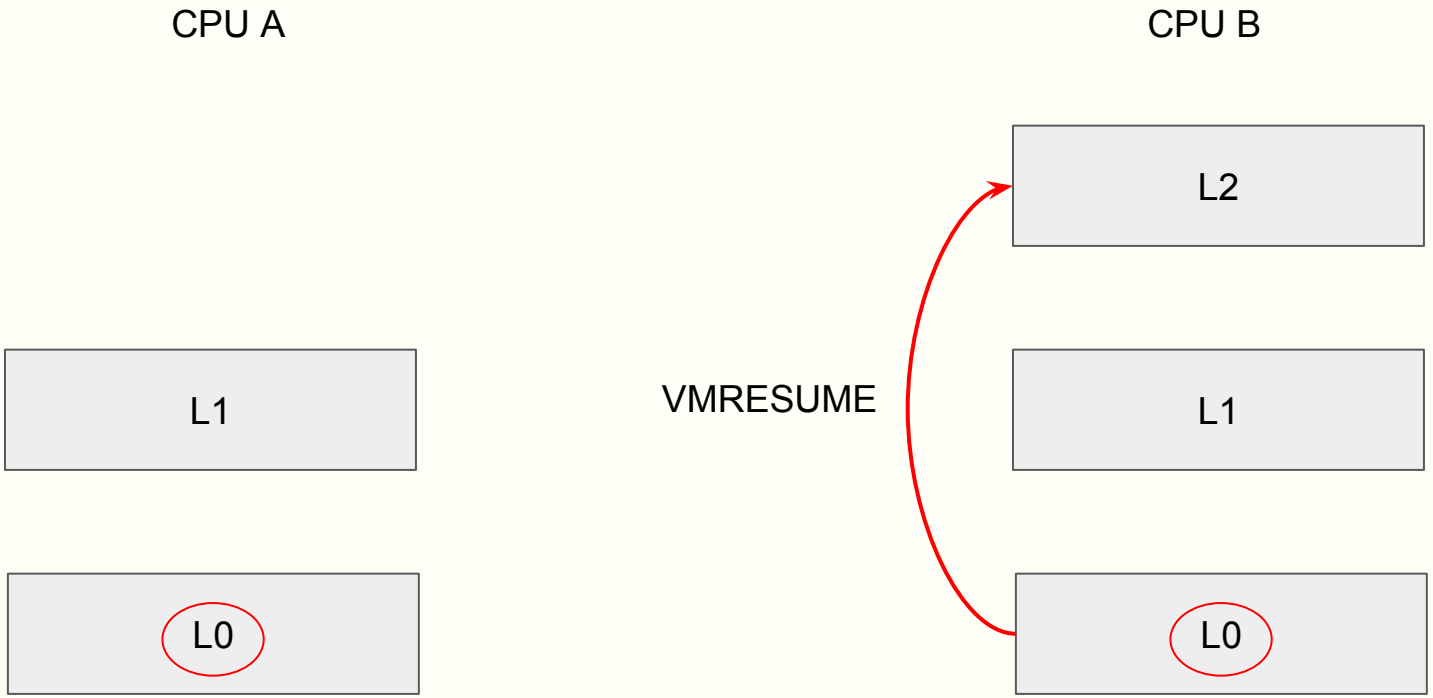
Example: nVMX posted-interrupt issue



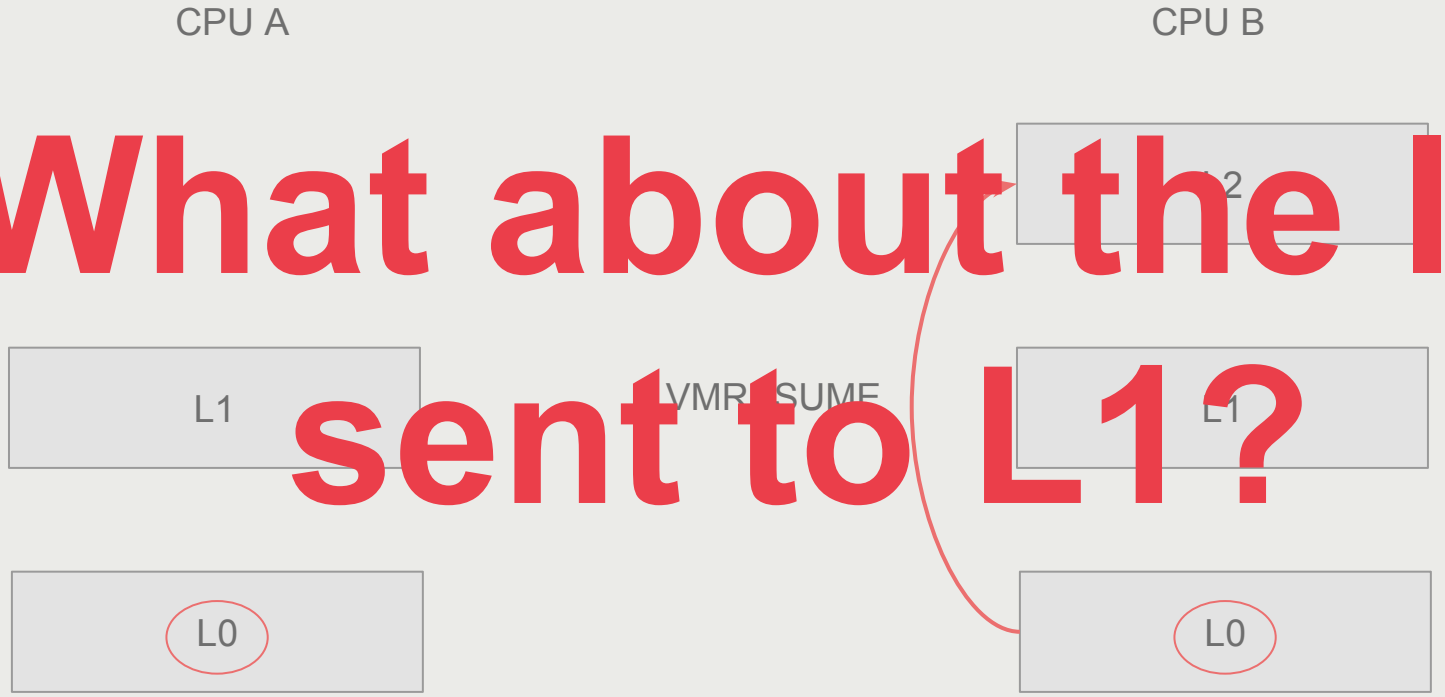
Example: nVMX posted-interrupt issue



Example: nVMX posted-interrupt issue



Example: nVMX posted-interrupt issue



What about the IPI sent to L1?

Example: nVMX posted-interrupt issue

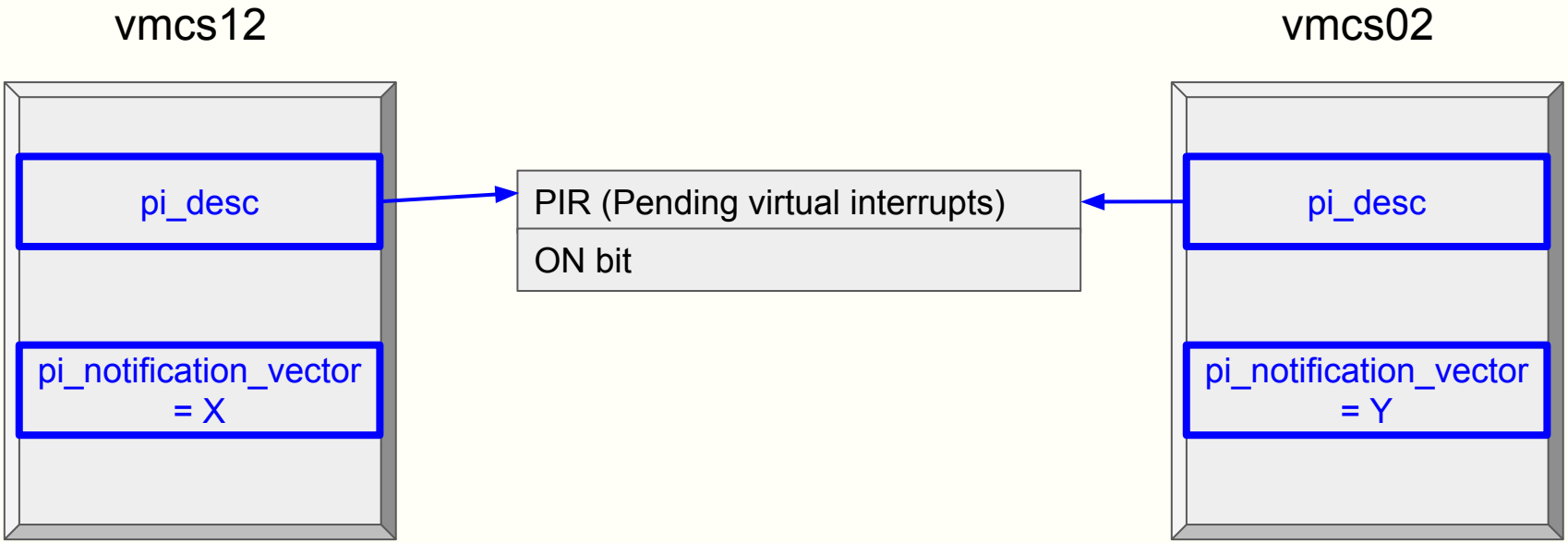
- L1 pending posted-interrupt needs to be evaluated before entry to L2!
  - L1 intercepts external-interrupts ⇒ L0 should L2→L1
  - Otherwise, inject interrupt directly to L2
- f27a85c4988d (“KVM: nVMX: Re-evaluate L1 pending events when running L2 and L1 got posted-interrupt”)

Example: nVMX posted-interrupt issue

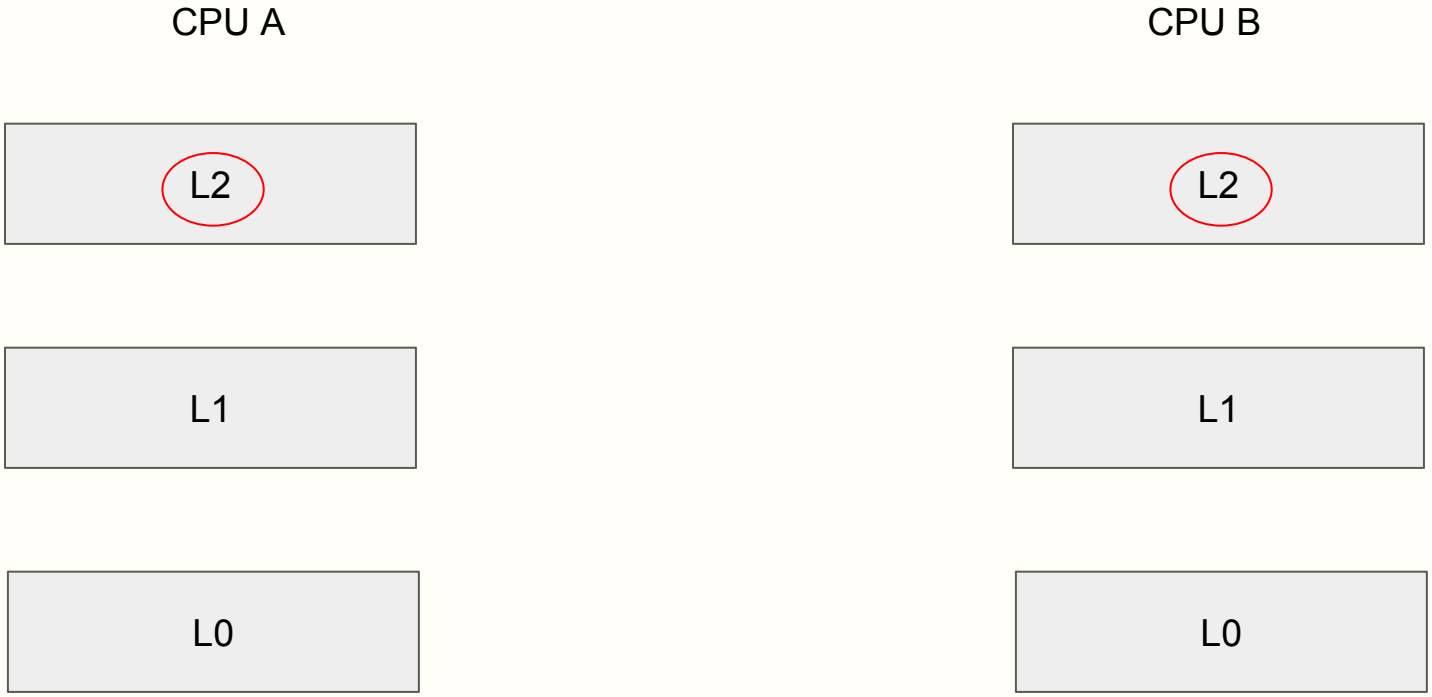
```
diff --git a/arch/x86/kvm/vmx.c b/arch/x86/kvm/vmx.c
index 5ea482bb1b9c..5fe94e375d2d 100644
--- a/arch/x86/kvm/vmx.c
+++ b/arch/x86/kvm/vmx.c
@@ -8978,6 +8978,7 @@ static int vmx_sync_pir_to_irr(struct kvm_vcpu *vcpu)
 {
        struct vcpu_vmx *vmx = to_vmx(vcpu);
        int max_irr;
+       bool max_irr_updated;

        WARN_ON(!vcpu->arch.apicv_active);
        if (pi_test_on(&vmx->pi_desc)) {
@@ -8987,7 +8988,16 @@ static int vmx_sync_pir_to_irr(struct kvm_vcpu *vcpu)
                 * But on x86 this is just a compiler barrier anyway.
                 */
                smp_mb__after_atomic();
-               kvm_apic_update_irr(vcpu, vmx->pi_desc.pir, &max_irr);
+               max_irr_updated =
+                       kvm_apic_update_irr(vcpu, vmx->pi_desc.pir, &max_irr);
+
+               /*
+                * If we are running L2 and L1 has a new pending interrupt
+                * which can be injected, we should re-evaluate
+                * what should be done with this new L1 interrupt.
+                */
+               if (is_guest_mode(vcpu) && max_irr_updated)
+                       kvm_vcpu_exiting_guest_mode(vcpu);
        } else {
                max_irr = kvm_lapic_find_highest_irr(vcpu);
        }
```

How nested posted-interrupts work?



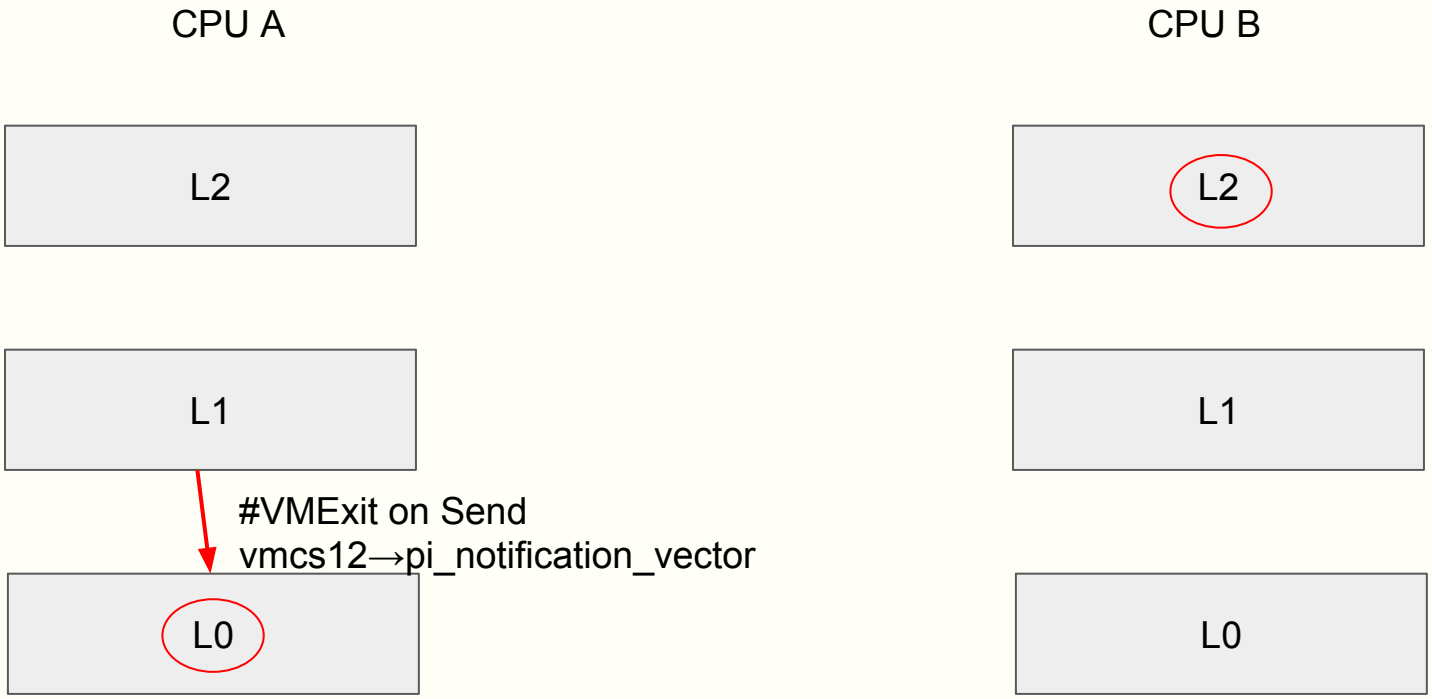
How nested posted-interrupts work?



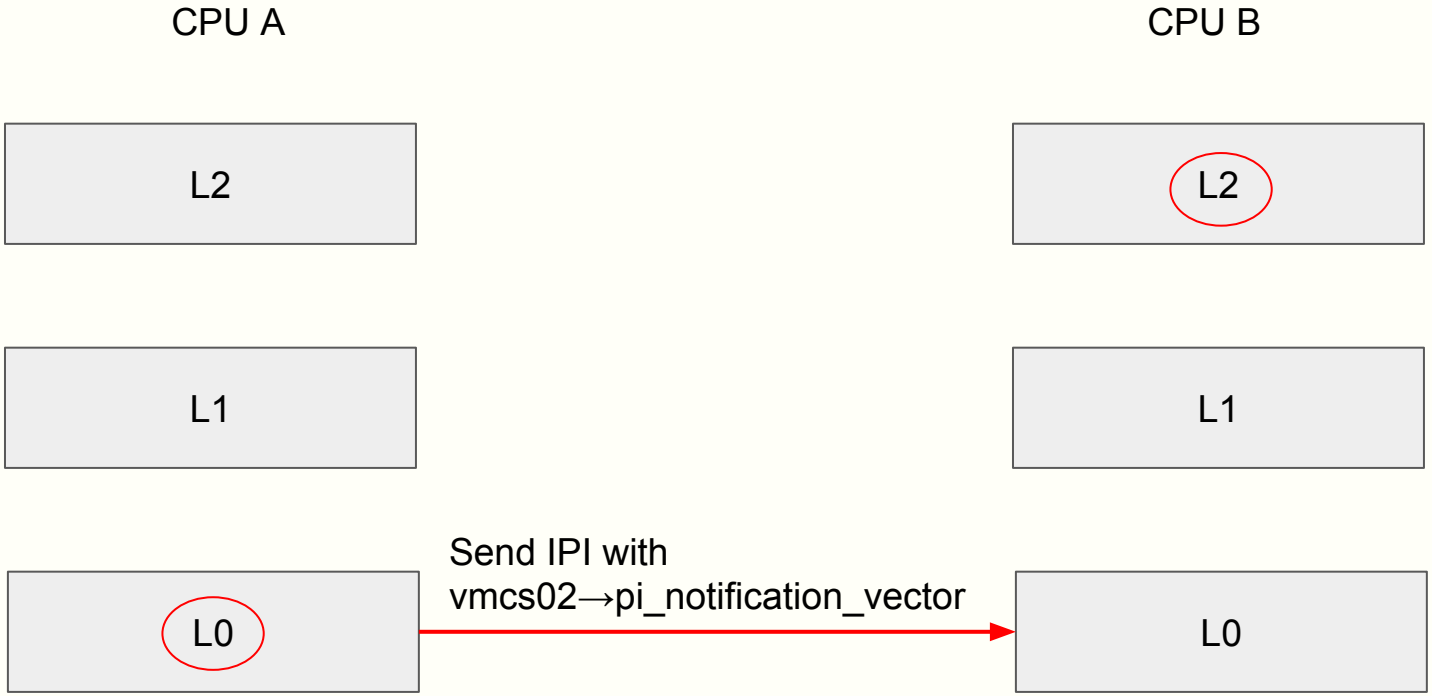
How nested posted-interrupts work?

How nested posted-interrupts work?

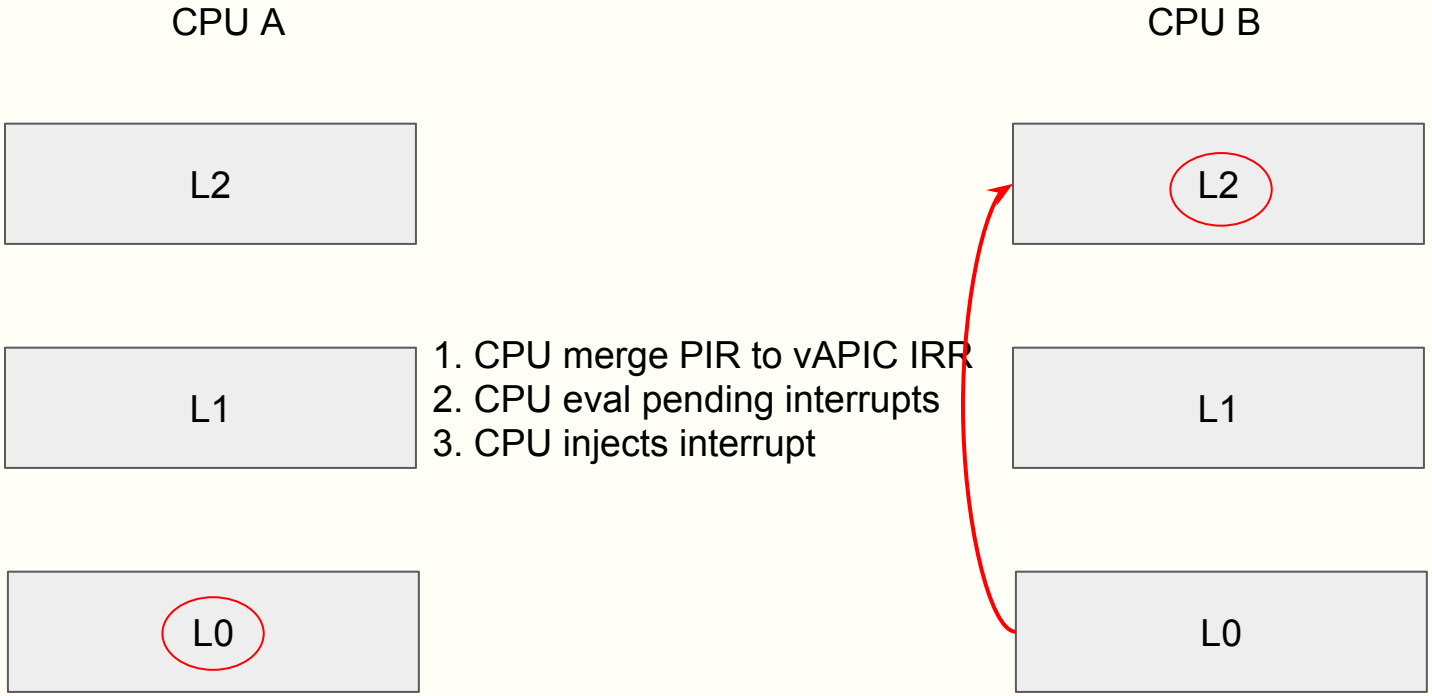
How nested posted-interrupts work?



How nested posted-interrupts work?



How nested posted-interrupts work?



How nested posted-interrupts works?

- What if target CPU currently at L0?
- Need to request L0 to evaluate pending nested posted-interrupts

How nested posted-interrupts works?

- What if target CPU currently at L0?
- Need to request L0 to evaluate pending nested posted-interrupts
  ⇒ Signal pending nested-posted-interrupt and set KVM_REQ_EVENT

How nested posted-interrupts works?

- What if target CPU currently at L0?
- Need to request L0 to evaluate pending nested posted-interrupts
  ⇒ Signal pending nested-posted-interrupt and set KVM_REQ_EVENT
- KVM_REQ_EVENT **emulate nested-posted-interrupt delivery in software!**
  - Clear pi_desc ON bit
  - Sync pi_desc→pir to L1 vLAPIC page
  - Update vmcs02→guest_intr_status (RVI/SVI) accordingly

HW-assisted nested posted-interrupt

- Software emulation error prone and less efficient
  - Could mistakenly diverge from hardware implementation
  - TODO: Bug: If target vCPU exits to L1 after sender sets pi_pending, than notification-vector interrupt is not raised to L1!
- TODO: Get rid of pi_pending and instead use L1 LAPIC IRR as CPU does
- TODO: Install host handler for vmcs02→pi_notification_vector to avoid missing pending interrupt
- TODO: Trigger CPU posted-interrupt logic by self-IPI in case of pending nested posted interrupt

HW-assisted nested posted-interrupt

- https://patchwork.kernel.org/patch/10132081/
- https://patchwork.kernel.org/patch/10132083/

Example 2: Nested posted-interrupt issue

- Race-Condition in delivering nested Posted-Interrupts
- Root-cause: Delivering event in non-standard way
  - Should use kvm_make_request() + kvm_vcpu_kick()
- 6b6977117f50 (“KVM: nVMX: Fix races when sending nested PI while dest enters/leaves L2”)

How virtual interrupt delivery works?

- VMCS→guest_intr_status specifies:
  1. RVI: Holds pending virtual interrupt vector
  2. SVI: Holds in-service virtual interrupt vector
- Certain actions cause evaluation of pending virtual interrupts
  - VMEntry, Write to TPR, Write to EOI, Self-IPI and posted-interrupts processing

How virtual interrupt delivery works?

- VMCS→guest_intr_status specifies:
  1. RVI: Holds pending virtual interrupt vector
  2. SVI: Holds in-service virtual interrupt vector
- Certain actions cause evaluation of pending virtual interrupts
  - VMEntry, Write to TPR, Write to EOI, Self-IPI and posted-interrupts processing
- VMCS→eoi_exitmap defines vectors on which EOI will cause VMExit in addition to EOI virtualization
  - In order to emulate LAPIC EOI broadcast to IOAPIC EOI

How KVM use virtual interrupt delivery?

- On entry to guest, set RVI to highest vector set in vLAPIC IRR

How KVM use virtual interrupt delivery?

- On entry to guest, set RVI to highest vector set in vLAPIC IRR
- Write to IOAPIC redir-table request KVM_REQ_SCAN_IOAPIC on all vCPUs
  - Configure VMCS->eoi_exitmap according to vectors IOAPIC require EOI broadcast on

How KVM use virtual interrupt delivery?

- On entry to guest, set RVI to highest vector set in vLAPIC IRR
- Write to IOAPIC redir-table request KVM_REQ_SCAN_IOAPIC on all vCPUs
  - Configure VMCS->eoi_exitmap according to vectors IOAPIC require EOI broadcast on
- Nested virtual interrupt delivery is “trivial”
  - vmcs02→guest_intr_status = vmcs12→guest_intr_status
  - vmcs02→eoi_exitmap = vmcs12→eoi_exitmap
  - Disable WRMSR intercept on LAPIC EOI and SELF_IPI

Example: Nested virtual-interrupt-delivery issue

- ESXi running as L1 which runs L2 guests lose network connectivity

Example: Nested virtual-interrupt-delivery issue

- ESXi running as L1 which runs L2 guests lose network connectivity
- Setting enable_apicv=0 seems to make problem disappear

Example: Nested virtual-interrupt-delivery issue

- ESXi running as L1 which runs L2 guests lose network connectivity
- Setting enable_apicv=0 seems to make problem disappear
- Analysis shows IOAPIC never got EOI for previous NIC IRQ

Example: Nested virtual-interrupt-delivery issue

- ESXi running as L1 which runs L2 guests lose network connectivity
- Setting enable_apicv=0 seems to make problem disappear
- Analysis shows IOAPIC never got EOI for previous NIC IRQ
- L0 KVM event trace shows:

Example: Nested virtual-interrupt-delivery issue

- ESXi running as L1 which runs L2 guests lose network connectivity
- Setting enable_apicv=0 seems to make problem disappear
- Analysis shows IOAPIC never got EOI for previous NIC IRQ
- L0 KVM event trace shows:
  1. ESXi kernel modifies IOAPIC redir-table (IOAPIC Steering)

Example: Nested virtual-interrupt-delivery issue

- ESXi running as L1 which runs L2 guests lose network connectivity
- Setting enable_apicv=0 seems to make problem disappear
- Analysis shows IOAPIC never got EOI for previous NIC IRQ
- L0 KVM event trace shows:
  1. ESXi kernel modifies IOAPIC redir-table (IOAPIC Steering)
  2. Write to IOAPIC requests KVM_REQ_SCAN_IOAPIC on all L1 vCPUs

Example: Nested virtual-interrupt-delivery issue

- ESXi running as L1 which runs L2 guests lose network connectivity
- Setting enable_apicv=0 seems to make problem disappear
- Analysis shows IOAPIC never got EOI for previous NIC IRQ
- L0 KVM event trace shows:
  1. ESXi kernel modifies IOAPIC redir-table (IOAPIC Steering)
  2. Write to IOAPIC requests KVM_REQ_SCAN_IOAPIC on all L1 vCPUs
  3. One CPU runs SCAN_IOAPIC handler while vCPU in guest-mode!

Example: Nested virtual-interrupt-delivery issue

- ESXi running as L1 which runs L2 guests lose network connectivity
- Setting enable_apicv=0 seems to make problem disappear
- Analysis shows IOAPIC never got EOI for previous NIC IRQ
- L0 KVM event trace shows:
  1. ESXi kernel modifies IOAPIC redir-table (IOAPIC Steering)
  2. Write to IOAPIC requests KVM_REQ_SCAN_IOAPIC on all L1 vCPUs
  3. One CPU runs SCAN_IOAPIC handler while vCPU in guest-mode!

  ⇒ Will update eoi_exitmap of vmcs02 instead of vmcs01!

  4. L1 NIC IRQ EOI will not exit to L0 and thus won't propagate to IOAPIC

Example: Nested virtual-interrupt-delivery issue

- IOAPIC never got EOI for previous NIC IRQ
- Issue found only when running ESXi as L1
  - Many issues caused by ESXi IOAPIC steering mechanism...
- Root-cause: IOAPIC EOI-exitmap code not adjusted to nested case
  - Update of EOI-exitmap should be delayed to when vCPU is running L1
  - Handle case LAPIC & IOAPIC are pass-through by updating vcpu->arch.ioapic_handled_vectors and only delay update of EOI-exitmap
- e40ff1d6608d (“KVM: nVMX: Do not load EOI-exitmap while running L2”)

Example: Nested virtual-interrupt-delivery issue

```
@@ -7124,6 +7122,20 @@ static void vcpu_scan_ioapic(struct kvm_vcpu *vcpu)
                        kvm_x86_ops->sync_pir_to_irr(vcpu);
                kvm_ioapic_scan_entry(vcpu, vcpu->arch.ioapic_handled_vectors);
        }
+
+       if (is_guest_mode(vcpu))
+               vcpu->arch.load_eoi_exitmap_pending = true;
+       else
+               kvm_make_request(KVM_REQ_LOAD_EOI_EXITMAP, vcpu);
+}
+
+static void vcpu_load_eoi_exitmap(struct kvm_vcpu *vcpu)
+{
+       u64 eoi_exit_bitmap[4];
+
+       if (!kvm_apic_hw_enabled(vcpu->arch.apic))
+               return;
+
        bitmap_or((ulong *)eoi_exit_bitmap, vcpu->arch.ioapic_handled_vectors,
                  vcpu_to_synic(vcpu)->vec_bitmap, 256);
        kvm_x86_ops->load_eoi_exitmap(vcpu, eoi_exit_bitmap);
@@ -7238,6 +7250,8 @@ static int vcpu_enter_guest(struct kvm_vcpu *vcpu)
                }
                if (kvm_check_request(KVM_REQ_SCAN_IOAPIC, vcpu))
                        vcpu_scan_ioapic(vcpu);
+               if (kvm_check_request(KVM_REQ_LOAD_EOI_EXITMAP, vcpu))
+                       vcpu_load_eoi_exitmap(vcpu);
```

Example 2: Nested virtual-interrupt-delivery issue

- vCPU should not halt when L1 is injecting events to L2
- Root-Cause: Not checking if VMEntry is vectoring when guest activity state is set to HLT
- 135a06c3a515 (“KVM: nVMX: Don't halt vcpu when L1 is injecting events to L2”)
- Should also wake blocked vCPU while in guest-mode if pending RVI
  - Evaluating pending vCPU events should include check if RVI[7:4] > vPPR[7:4]
  - e6c67d8cf117 (“KVM: nVMX: Wake blocked vCPU in guest-mode if pending interrupt in virtual APICv”)

Example 3: Nested virtual-interrupt-delivery issue

- Direct interrupt injection to L2 don’t update L1 LAPIC IRR and ISR and doesn’t consider PPR
- Root-cause: Not using standard inject_pending_event() event injection framework for injecting interrupt directly to L2
- 851c1a18c541 (“KVM: nVMX: Fix injection to L2 when L1 don't intercept external-interrupts”)

nVMX event injection

More issues...

Example: nVMX event-injection issue

1. L2 RDMSR bad_msr which exits to L0

Example: nVMX event-injection issue

1. L2 RDMSR bad_msr which exits to L0
2. L1 doesn’t intercept MSR and thus RDMSR emulated by L0

Example: nVMX event-injection issue

1. L2 RDMSR bad_msr which exits to L0
2. L1 doesn’t intercept MSR and thus RDMSR emulated by L0
3. L0 queues a pending #GP exception

Example: nVMX event-injection issue

1. L2 RDMSR bad_msr which exits to L0
2. L1 doesn’t intercept MSR and thus RDMSR emulated by L0
3. L0 queues a pending #GP exception
4. L0 KVM_REQ_EVENT evaluates what should be done with queued events:
    a. L1 doesn’t intercept #GP
    b. L1 has pending interrupt in LAPIC (Other L1 CPU sent IPI)

Example: nVMX event-injection issue

1. L2 RDMSR bad_msr which exits to L0
2. L1 doesn’t intercept MSR and thus RDMSR emulated by L0
3. L0 queues a pending #GP exception
4. L0 KVM_REQ_EVENT evaluates what should be done with queued events:
    a. L1 doesn’t intercept #GP
    b. L1 has pending interrupt in LAPIC (Other L1 CPU sent IPI)
5. L0 emulates L2→L1 on EXTERNAL_INTERRUPT

Example: nVMX event-injection issue

1. L2 RDMSR bad_msr which exits to L0
2. L1 doesn’t intercept MSR and thus RDMSR emulated by L0
3. L0 queues a pending #GP exception
4. L0 KVM_REQ_EVENT evaluates what should be done with queued events:
   a. L1 doesn’t intercept #GP
   b. L1 has pending interrupt in LAPIC (Other L1 CPU sent IPI)
5. L0 emulates L2→L1 on EXTERNAL_INTERRUPT
6. Exception still pending in struct kvm_vcpu_arch
   ⇒ Will be injected to L1 on next KVM_REQ_EVENT!

Example: nVMX event-injection issue

- L2 exception injected into L1!
- Root-cause: Not clearing exception.pending on L2→L1 transition
  - Bug mistakenly introduced when exception.injected was added
- Fix: Clear pending exception on L2→L1
  - 5c7d4f9ad39d (“KVM: nVMX: Fix bug of injecting L2 exception into L1”)

Example: nVMX event-injection issue

- L2 exception injected into L1!
- Root-cause: Not clearing exception.pending on L2→L1 transition
  - Bug mistakenly introduced when exception.injected was added
- Fix: Clear pending exception on L2→L1
  - 5c7d4f9ad39d (“KVM: nVMX: Fix bug of injecting L2 exception into L1”)
- OK to clear exception.pending?
  - A pending exception will be **re-triggered*** on next resume of L2

* TODO: L2 pending trap exceptions can still be lost…