

KVM Lock Overview

1. Acquisition Orders

The acquisition orders for mutexes are as follows:

- `kvm->lock` is taken outside `vcpu->mutex`
- `kvm->lock` is taken outside `kvm->slots_lock` and `kvm->irq_lock`
- `kvm->slots_lock` is taken outside `kvm->irq_lock`, though acquiring them together is quite rare.
- Unlike `kvm->slots_lock`, `kvm->slots_arch_lock` is released before `synchronize_srcu(&kvm->srcu)`. Therefore `kvm->slots_arch_lock` can be taken inside a `kvm->srcu` read-side critical section, while `kvm->slots_lock` cannot.
- `kvm->mm_active_invalidate_count` ensures that pairs of `invalidate_range_start()` and `invalidate_range_end()` callbacks use the same `memslots` array. `kvm->slots_lock` and `kvm->slots_arch_lock` are taken on the waiting side in `install_new_memslots`, so MMU notifiers must not take either `kvm->slots_lock` or `kvm->slots_arch_lock`.

On x86:

- `vcpu->mutex` is taken outside `kvm->arch.hyperv.hv_lock`
- `kvm->arch.mmu_lock` is an `rwlock`. `kvm->arch.tdp_mmu_pages_lock` and `kvm->arch.mmu_unsync_pages_lock` are taken inside `kvm->arch.mmu_lock`, and cannot be taken without already holding `kvm->arch.mmu_lock` (typically with `read_lock` for the TDP MMU, thus the need for additional spinlocks).

Everything else is a leaf: no other lock is taken inside the critical sections.

2. Exception

Fast page fault:

Fast page fault is the fast path which fixes the guest page fault out of the `mmu-lock` on x86. Currently, the page fault can be fast in one of the following two cases:

1. Access Tracking: The SPTE is not present, but it is marked for access tracking. That means we need to restore the saved R/X bits. This is described in more detail later below.
2. Write-Protection: The SPTE is present and the fault is caused by write-protect. That means we just need to change the W bit of the `spte`.

What we use to avoid all the race is the Host-writable bit and MMU-writable bit on the `spte`:

- Host-writable means the `gfn` is writable in the host kernel page tables and in its KVM memslot.
- MMU-writable means the `gfn` is writable in the guest's `mmu` and it is not write-protected by shadow page write-protection.

On fast page fault path, we will use `cmpxchg` to atomically set the `spte W` bit if `spte.HOST_WRITEABLE = 1` and `spte.WRITE_PROTECT = 1`, to restore the saved R/X bits if for an access-traced `spte`, or both. This is safe because whenever changing these bits can be detected by `cmpxchg`.

But we need carefully check these cases:

1. The mapping from `gfn` to `pfn`

The mapping from `gfn` to `pfn` may be changed since we can only ensure the `pfn` is not changed during `cmpxchg`. This is a ABA problem, for example, below case will happen:

At the beginning:	
<pre>gpte = gfn1 gfn1 is mapped to pfn1 on host spte is the shadow page table entry corresponding with gpte and spte = pfn1</pre>	
On fast page fault path:	
CPU 0:	CPU 1:
<pre>old_spte = *spte;</pre>	
	<pre>pfn1 is swapped out: spte = 0; pfn1 is re-allocated for gfn2. gpte is changed to point to gfn2 by the guest: spte = pfn1;</pre>

```

if (cmpxchg(spte, old_spte, old_spte+W)
    mark_page_dirty(vcpu->kvm, gfn1)
    OOPS!!!

```

We dirty-log for gfn1, that means gfn2 is lost in dirty-bitmap.

For direct sp, we can easily avoid it since the spte of direct sp is fixed to gfn. For indirect sp, we disabled fast page fault for simplicity.

A solution for indirect sp could be to pin the gfn, for example via `kvm_vcpu_gfn_to_pfn_atomic`, before the `cmpxchg`. After the pinning:

- We have held the refcount of pfn that means the pfn can not be freed and be reused for another gfn.
- The pfn is writable and therefore it cannot be shared between different gfn's by KSM.

Then, we can ensure the dirty bitmaps is correctly set for a gfn.

2. Dirty bit tracking

In the origin code, the spte can be fast updated (non-atomically) if the spte is read-only and the Accessed bit has already been set since the Accessed bit and Dirty bit can not be lost.

But it is not true after fast page fault since the spte can be marked writable between reading spte and updating spte. Like below case:

At the beginning:	
<pre> spte.W = 0 spte.Accessed = 1 </pre>	
CPU 0:	CPU 1:
In <code>mmu_spte_clear_track_bits()</code> :	
<pre> old_spte = *spte; /* 'if' condition is satisfied. */ if (old_spte.Accessed == 1 && old_spte.W == 0) spte = 0ull; </pre>	
	on fast page fault path: <pre> spte.W = 1 </pre> memory write on the spte: <pre> spte.Dirty = 1 </pre>
<pre> else old_spte = xchg(spte, 0ull) if (old_spte.Accessed == 1) kvm_set_pfn_accessed(spte.pfn); if (old_spte.Dirty == 1) kvm_set_pfn_dirty(spte.pfn); OOPS!!! </pre>	

The Dirty bit is lost in this case.

In order to avoid this kind of issue, we always treat the spte as "volatile" if it can be updated out of mmu-lock, see `spte_has_volatile_bits()`, it means, the spte is always atomically updated in this case.

3. flush tlbs due to spte updated

If the spte is updated from writable to readonly, we should flush all TLBs, otherwise `rmap_write_protect` will find a read-only spte, even though the writable spte might be cached on a CPU's TLB.

As mentioned before, the spte can be updated to writable out of mmu-lock on fast page fault path, in order to easily audit the path, we see if TLBs need be flushed caused by this reason in `mmu_spte_update()` since this is a common function to update spte (present -> present).

Since the spte is "volatile" if it can be updated out of mmu-lock, we always atomically update the spte, the race caused by fast page fault can be avoided, See the comments in `spte_has_volatile_bits()` and `mmu_spte_update()`.

Lockless Access Tracking:

This is used for Intel CPUs that are using EPT but do not support the EPT A/D bits. In this case, PTEs are tagged as A/D disabled (using ignored bits), and when the KVM MMU notifier is called to track accesses to a page (via `kvm_mmu_notifier_clear_flush_young`), it marks the PTE not-present in hardware by clearing the RWX bits in the PTE and storing the original R & X bits in more unused/ignored bits. When the VM tries to access the page later on, a fault is generated and the fast

page fault mechanism described above is used to atomically restore the PTE to a Present state. The W bit is not saved when the PTE is marked for access tracking and during restoration to the Present state, the W bit is set depending on whether or not it was a write access. If it wasn't, then the W bit will remain clear until a write access happens, at which time it will be set using the Dirty tracking mechanism described above.

3. Reference

`kvm_lock`

Type: mutex
Arch: any
Protects:

- `vm_list`

`kvm_count_lock`

Type: raw_spinlock_t
Arch: any
Protects:

- hardware virtualization enable/disable

Comment: 'raw' because hardware enabling/disabling must be atomic /wrt migration.

`kvm->mn_invalidate_lock`

Type: spinlock_t
Arch: any
Protects: `mn_active_invalidate_count`, `mn_memslots_update_rcuwait`

`kvm_arch::tsc_write_lock`

Type: raw_spinlock_t
Arch: x86
Protects:

- `kvm_arch::{last_tsc_write,last_tsc_nsec,last_tsc_offset}`
- tsc offset in `vmcb`

Comment: 'raw' because updating the tsc offsets must not be preempted.

`kvm->mmu_lock`

Type: spinlock_t or rwlock_t
Arch: any
Protects: -shadow page/shadow tlb entry
Comment: it is a spinlock since it is used in mmu notifier.

`kvm->srcu`

Type: srcu lock
Arch: any
Protects:

- `kvm->memslots`
- `kvm->buses`

Comment: The srcu read lock must be held while accessing memslots (e.g. when using `gfn_to_*` functions) and while accessing in-kernel MMIO/PIO address->device structure mapping (`kvm->buses`). The srcu index can be stored in `kvm_vcpu->srcu_idx` per vcpu if it is needed by multiple functions.

`kvm->slots_arch_lock`

Type: mutex
Arch: any (only needed on x86 though)
Protects: any arch-specific fields of memslots that have to be modified in a `kvm->srcu` read-side critical section.
Comment: must be held before reading the pointer to the current memslots, until after all changes to the memslots are complete

`wakeup_vcpus_on_cpu_lock`

Type: spinlock_t
Arch: x86
Protects: `wakeup_vcpus_on_cpu`
Comment: This is a per-CPU lock and it is used for VT-d posted-interrupts. When VT-d posted-interrupts is supported and the VM has assigned devices, we put the blocked vCPU on the list `blocked_vcpu_on_cpu` protected by `blocked_vcpu_on_cpu_lock`, when VT-d hardware issues wakeup notification event since external interrupts from the assigned devices happens, we will find the vCPU on the list to wakeup.