The x86 kvm shadow mmu

The mmu (in arch/x86/kvm, files mmu.[ch] and paging_tmpl.h) is responsible for presenting a standard x86 mmu to the guest, while translating guest physical addresses to host physical addresses.

The mmu code attempts to satisfy the following requirements:

- correctness:
  the guest should not be able to determine that it is running on an emulated mmu except for timing (we attempt to comply with the specification, not emulate the characteristics of a particular implementation such as tlb size)
- security:
  the guest must not be able to touch host memory not assigned to it
- performance:
  minimize the performance penalty imposed by the mmu
- scaling:
  need to scale to large memory and large vcpu guests
- hardware:
  support the full range of x86 virtualization hardware
- integration:
  Linux memory management code must be in control of guest memory so that swapping, page migration, page merging, transparent hugepages, and similar features work without change
- dirty tracking:
  report writes to guest memory to enable live migration and framebuffer-based displays
- footprint:
  keep the amount of pinned kernel memory low (most memory should be shrinkable)
- reliability:
  avoid multipage or GFP_ATOMIC allocations

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)

Virtual and real hardware supported

The mmu supports first-generation mmu hardware, which allows an atomic switch of the current paging mode and cr3 during guest entry, as well as two-dimensional paging (AMD's NPT and Intel's EPT). The emulated hardware it exposes is the traditional 2/3/4 level x86 mmu, with support for global pages, pae, pse, pse36, cr0.wp, and 1GB pages. Emulated hardware also able to expose NPT capable hardware on NPT capable hosts.

Translation

The primary job of the mmu is to program the processor's mmu to translate addresses for the guest. Different translations are required at different times:

- when guest paging is disabled, we translate guest physical addresses to host physical addresses (gpa->hpa)
- when guest paging is enabled, we translate guest virtual addresses, to guest physical addresses, to host physical addresses (gva->gpa->hpa)
- when the guest launches a guest of its own, we translate nested guest virtual addresses, to nested guest physical addresses, to guest physical addresses, to host physical addresses (ngva->ngpa->gpa->hpa)

The primary challenge is to encode between 1 and 3 translations into hardware that support only 1 (traditional) and 2 (tdp) translations. When the number of required translations matches the hardware, the mmu operates in direct mode; otherwise it operates in shadow mode (see below).

Memory

Guest memory (gpa) is part of the user address space of the process that is using kvm. Userspace defines the translation between guest addresses and user addresses (gpa->hva); note that two gpas may alias to the same hva, but not vice versa.

These hvas may be backed using any method available to the host: anonymous memory, file backed memory, and device memory. Memory might be paged by the host at any time.

Events

The mmu is driven by events, some from the guest, some from the host.

Guest generated events:

- writes to control registers (especially cr3)
- invlpg/invlpga instruction execution
- access to missing or protected translations

Host generated events:

- changes in the gpa->hpa translation (either through gpa->hva changes or through hva->hpa changes)
- memory pressure (the shrinker)

Shadow pages

The principal data structure is the shadow page, 'struct kvm_mmu_page'. A shadow page contains 512 sptes, which can be either leaf or nonleaf sptes. A shadow page may contain a mix of leaf and nonleaf sptes.

A nonleaf spte allows the hardware mmu to reach the leaf pages and is not related to a translation directly. It points to other shadow pages.

A leaf spte corresponds to either one or two translations encoded into one paging structure entry. These are always the lowest level of the translation stack, with optional higher level translations left to NPT/EPT. Leaf ptes point at guest pages.

The following table shows translations encoded by leaf ptes, with higher-level translations in parentheses:

Non-nested guests:

```
nonpaging:      gpa->hpa
paging:         gva->gpa->hpa
paging, tdp:    (gva->)gpa->hpa
```

Nested guests:

```
non-tdp:        ngva->gpa->hpa  (*)
tdp:            (ngva->)ngpa->gpa->hpa

(*) the guest hypervisor will encode the ngva->gpa translation into its page
    tables if npt is not present
```

Shadow pages contain the following information:

role.level:
: The level in the shadow paging hierarchy that this shadow page belongs to. 1=4k sptes, 2=2M sptes, 3=1G sptes, etc.

role.direct:
: If set, leaf sptes reachable from this page are for a linear range. Examples include real mode translation, large guest pages backed by small host pages, and gpa->hpa translations when NPT or EPT is active. The linear range starts at (gfn << PAGE_SHIFT) and its size is determined by role.level (2MB for first level, 1GB for second level, 0.5TB for third level, 256TB for fourth level) If clear, this page corresponds to a guest page table denoted by the gfn field.

role.quadrant:
: When role.has_4_byte_gpte=1, the guest uses 32-bit gptes while the host uses 64-bit sptes. That means a guest page table contains more ptes than the host, so multiple shadow pages are needed to shadow one guest page. For first-level shadow pages, role.quadrant can be 0 or 1 and denotes the first or second 512-gpte block in the guest page table. For second-level page tables, each 32-bit gpte is converted to two 64-bit sptes (since each first-level guest page is shadowed by two first-level shadow pages) so role.quadrant takes values in the range 0..3. Each quadrant maps 1GB virtual address space.

role.access:

Inherited guest access permissions from the parent ptes in the form uwx. Note execute permission is positive, not negative.

role.invalid:
: The page is invalid and should not be used. It is a root page that is currently pinned (by a cpu hardware register pointing to it); once it is unpinned it will be destroyed.

role.has_4_byte_gpte:
: Reflects the size of the guest PTE for which the page is valid, i.e. '0' if direct map or 64-bit gptes are in use, '1' if 32-bit gptes are in use.

role.efer_nx:
: Contains the value of efer.nx for which the page is valid.

role.cr0_wp:
: Contains the value of cr0.wp for which the page is valid.

role.smep_andnot_wp:
: Contains the value of cr4.smep && !cr0.wp for which the page is valid (pages for which this is true are different from other pages; see the treatment of cr0.wp=0 below).

role.smap_andnot_wp:
: Contains the value of cr4.smap && !cr0.wp for which the page is valid (pages for which this is true are different from other pages; see the treatment of cr0.wp=0 below).

role.smm:
: Is 1 if the page is valid in system management mode. This field determines which of the kvm_memslots array was used to build this shadow page; it is also used to go back from a struct kvm_mmu_page to a memslot, through the kvm_memslots_for_spte_role macro and __gfn_to_memslot.

role.ad_disabled:
: Is 1 if the MMU instance cannot use A/D bits. EPT did not have A/D bits before Haswell; shadow EPT page tables also cannot use A/D bits if the L1 hypervisor does not enable them.

gfn:
: Either the guest page table containing the translations shadowed by this page, or the base page frame for linear translations. See role.direct.

spt:
: A pageful of 64-bit sptes containing the translations for this page. Accessed by both kvm and hardware. The page pointed to by spt will have its page->private pointing back at the shadow page structure. sptes in spt point either at guest pages, or at lower-level shadow pages. Specifically, if sp1 and sp2 are shadow pages, then sp1->spt[n] may point at __pa(sp2->spt). sp2 will point back at sp1 through parent_pte. The spt array forms a DAG structure with the shadow page as a node, and guest pages as leaves.

gfns:
: An array of 512 guest frame numbers, one for each present pte. Used to perform a reverse map from a pte to a gfn. When role.direct is set, any element of this array can be calculated from the gfn field when used, in this case, the array of gfns is not allocated. See role.direct and gfn.

root_count:
: A counter keeping track of how many hardware registers (guest cr3 or pdptrs) are now pointing at the page. While this counter is nonzero, the page cannot be destroyed. See role.invalid.

parent_ptes:
: The reverse mapping for the pte/ptes pointing at this page's spt. If parent_ptes bit 0 is zero, only one spte points at this page and parent_ptes points at this single spte, otherwise, there exists multiple sptes pointing at this page and (parent_ptes & ~0x1) points at a data structure with a list of parent sptes.

unsync:
: If true, then the translations in this page may not match the guest's translation. This is equivalent to the state of the tlb when a pte is changed but before the tlb entry is flushed. Accordingly, unsync ptes are synchronized when the guest executes invlpg or flushes its tlb by other means. Valid for leaf pages.

unsync_children:
: How many sptes in the page point at pages that are unsync (or have unsynchronized children).

unsync_child_bitmap:
: A bitmap indicating which sptes in spt point (directly or indirectly) at pages that may be unsynchronized. Used to quickly locate all unsychronized pages reachable from a given page.

clear_spte_count:
: Only present on 32-bit hosts, where a 64-bit spte cannot be written atomically. The reader uses this while running out of the MMU lock to detect in-progress updates and retry them until the writer has finished the write.

write_flooding_count:
: A guest may write to a page table many times, causing a lot of emulations if the page needs to be write-protected (see "Synchronized and unsynchronized pages" below). Leaf pages can be unsynchronized so that they do not trigger frequent emulation, but this is not possible for non-leafs. This field counts the number of emulations since the last time the page table was actually used; if emulation is triggered too frequently on this page, KVM will unmap the page to avoid emulation in the future.

Reverse map

The mmu maintains a reverse mapping whereby all ptes mapping a page can be reached given its gfn. This is used, for example, when swapping out a page.

Synchronized and unsynchronized pages

The guest uses two events to synchronize its tlb and page tables: tlb flushes and page invalidations (invlpg).

A tlb flush means that we need to synchronize all sptes reachable from the guest's cr3. This is expensive, so we keep all guest page tables write protected, and synchronize sptes to gptes when a gpte is written.

A special case is when a guest page table is reachable from the current guest cr3. In this case, the guest is obliged to issue an invlpg instruction before using the translation. We take advantage of that by removing write protection from the guest page, and allowing the guest to modify it freely. We synchronize modified gptes when the guest invokes invlpg. This reduces the amount of emulation we have to do when the guest modifies multiple gptes, or when the a guest page is no longer used as a page table and is used for random guest data.

As a side effect we have to resynchronize all reachable unsynchronized shadow pages on a tlb flush.

Reaction to events

- guest page fault (or npt page fault, or ept violation)

This is the most complicated event. The cause of a page fault can be:

- a true guest fault (the guest translation won't allow the access) (*)
- access to a missing translation
- access to a protected translation - when logging dirty pages, memory is write protected - synchronized shadow pages are write protected (*)
- access to untranslatable memory (mmio)

(*) not applicable in direct mode

Handling a page fault is performed as follows:

- if the RSV bit of the error code is set, the page fault is caused by guest accessing MMIO and cached MMIO information is available.
  - walk shadow page table
  - check for valid generation number in the spte (see "Fast invalidation of MMIO sptes" below)
  - cache the information to vcpu->arch.mmio_gva, vcpu->arch.mmio_access and vcpu->arch.mmio_gfn, and call the emulator
- If both P bit and R/W bit of error code are set, this could possibly be handled as a "fast page fault" (fixed without taking the MMU lock). See the description in Documentation/virt/kvm/locking.rst.
- if needed, walk the guest page tables to determine the guest translation (gva->gpa or ngpa->gpa)
  - if permissions are insufficient, reflect the fault back to the guest
- determine the host page
  - if this is an mmio request, there is no host page; cache the info to vcpu->arch.mmio_gva, vcpu->arch.mmio_access and vcpu->arch.mmio_gfn
- walk the shadow page table to find the spte for the translation, instantiating missing intermediate page tables as necessary
  - If this is an mmio request, cache the mmio info to the spte and set some reserved bit on the spte (see callers of kvm_mmu_set_mmio_spte_mask)
- try to unsynchronize the page
  - if successful, we can let the guest continue and modify the gpte
- emulate the instruction
  - if failed, unshadow the page and let the guest continue
- update any translations that were modified by the instruction

invlpg handling:

- walk the shadow page hierarchy and drop affected translations
- try to reinstantiate the indicated translation in the hope that the guest will use it in the near future

Guest control register updates:

- mov to cr3
  - look up new shadow roots
  - synchronize newly reachable shadow pages
- mov to cr0/cr4/efer
  - set up mmu context for new paging mode

- look up new shadow roots
- synchronize newly reachable shadow pages

Host translation updates:

- mmu notifier called with updated hva
- look up affected sptes through reverse map
- drop (or update) translations

Emulating cr0.wp

If tdp is not enabled, the host must keep cr0.wp=1 so page write protection works for the guest kernel, not guest guest userspace. When the guest cr0.wp=1, this does not present a problem. However when the guest cr0.wp=0, we cannot map the permissions for gpte.u=1, gpte.w=0 to any spte (the semantics require allowing any guest kernel access plus user read access).

We handle this by mapping the permissions to two possible sptes, depending on fault type:

- kernel write fault: spte.u=0, spte.w=1 (allows full kernel access, disallows user access)
- read fault: spte.u=1, spte.w=0 (allows full read access, disallows kernel write access)

(user write faults generate a #PF)

In the first case there are two additional complications:

- if CR4.SMEP is enabled: since we've turned the page into a kernel page, the kernel may now execute it. We handle this by also setting spte.nx. If we get a user fetch or read fault, we'll change spte.u=1 and spte.nx=gpte.nx back. For this to work, KVM forces EFER.NX to 1 when shadow paging is in use.
- if CR4.SMAP is disabled: since the page has been changed to a kernel page, it can not be reused when CR4.SMAP is enabled. We set CR4.SMAP && !CR0.WP into shadow page's role to avoid this case. Note, here we do not care the case that CR4.SMAP is enabled since KVM will directly inject #PF to guest due to failed permission check.

To prevent an spte that was converted into a kernel page with cr0.wp=0 from being written by the kernel after cr0.wp has changed to 1, we make the value of cr0.wp part of the page role. This means that an spte created with one value of cr0.wp cannot be used when cr0.wp has a different value - it will simply be missed by the shadow page lookup code. A similar issue exists when an spte created with cr0.wp=0 and cr4.smep=0 is used after changing cr4.smep to 1. To avoid this, the value of !cr0.wp && cr4.smep is also made a part of the page role.

Large pages

The mmu supports all combinations of large and small guest and host pages. Supported page sizes include 4k, 2M, 4M, and 1G. 4M pages are treated as two separate 2M pages, on both guest and host, since the mmu always uses PAE paging.

To instantiate a large spte, four constraints must be satisfied:

- the spte must point to a large host page
- the guest pte must be a large pte of at least equivalent size (if tdp is enabled, there is no guest pte and this condition is satisfied)
- if the spte will be writeable, the large page frame may not overlap any write-protected pages
- the guest page must be wholly contained by a single memory slot

To check the last two conditions, the mmu maintains a ->disallow_lpage set of arrays for each memory slot and large page size. Every write protected page causes its disallow_lpage to be incremented, thus preventing instantiation of a large spte. The frames at the end of an unaligned memory slot have artificially inflated ->disallow_lpages so they can never be instantiated.

Fast invalidation of MMIO sptes

As mentioned in "Reaction to events" above, kvm will cache MMIO information in leaf sptes. When a new memslot is added or an existing memslot is changed, this information may become stale and needs to be invalidated. This also needs to hold the MMU lock while walking all shadow pages, and is made more scalable with a similar technique.

MMIO sptes have a few spare bits, which are used to store a generation number. The global generation number is stored in kvm_memslots(kvm)->generation, and increased whenever guest memory info changes.

When KVM finds an MMIO spte, it checks the generation number of the spte. If the generation number of the spte does not equal the global generation number, it will ignore the cached MMIO information and handle the page fault through the slow path.

Since only 18 bits are used to store generation-number on mmio spte, all pages are zapped when there is an overflow.

Unfortunately, a single memory access might access kvm_memslots(kvm) multiple times, the last one happening when the generation number is retrieved and stored into the MMIO spte. Thus, the MMIO spte might be created based on out-of-date information, but with an up-to-date generation number.

To avoid this, the generation number is incremented again after synchronize_srcu returns; thus, bit 63 of kvm_memslots(kvm)->generation set to 1 only during a memslot update, while some SRCU readers might be using the old copy. We do not want to use an MMIO sptes created with an odd generation number, and we can do this without losing a bit in the MMIO spte. The "update in-

progress" bit of the generation is not stored in MMIO spte, and is so is implicitly zero when the generation is extracted out of the spte. If KVM is unlucky and creates an MMIO spte while an update is in-progress, the next access to the spte will always be a cache miss. For example, a subsequent access during the update window will miss due to the in-progress flag diverging, while an access after the update window closes will have a higher generation number (as compared to the spte).

Further reading

- NPT presentation from KVM Forum 2008 https://www.linux-kvm.org/images/c/c8/KvmForum2008%24kdf2008_21.pdf