The Shadowy Depths of the KVM MMU

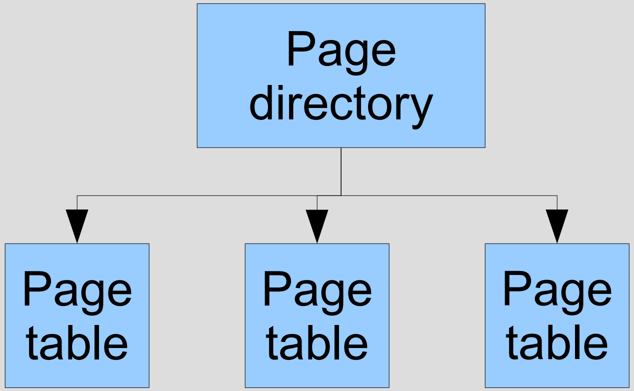
KVM Forum 2007

Agenda

- A quick recap of x86 paging
- Shadow paging in general
- Goals for the KVM MMU
- The gory details

Paging on x86

- Function: translate virtual addresses to physical addresses
- Looks a bit like a radix tree
- Read and modified by hardware
- Cached by on-chip structures (the Translation Look-aside Buffer, or TLB)

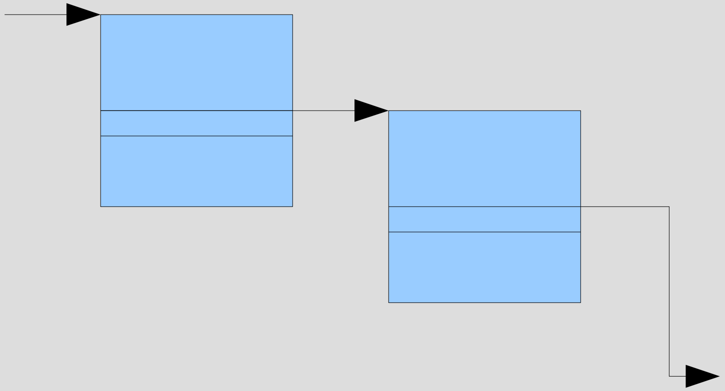


Page directory

Page table

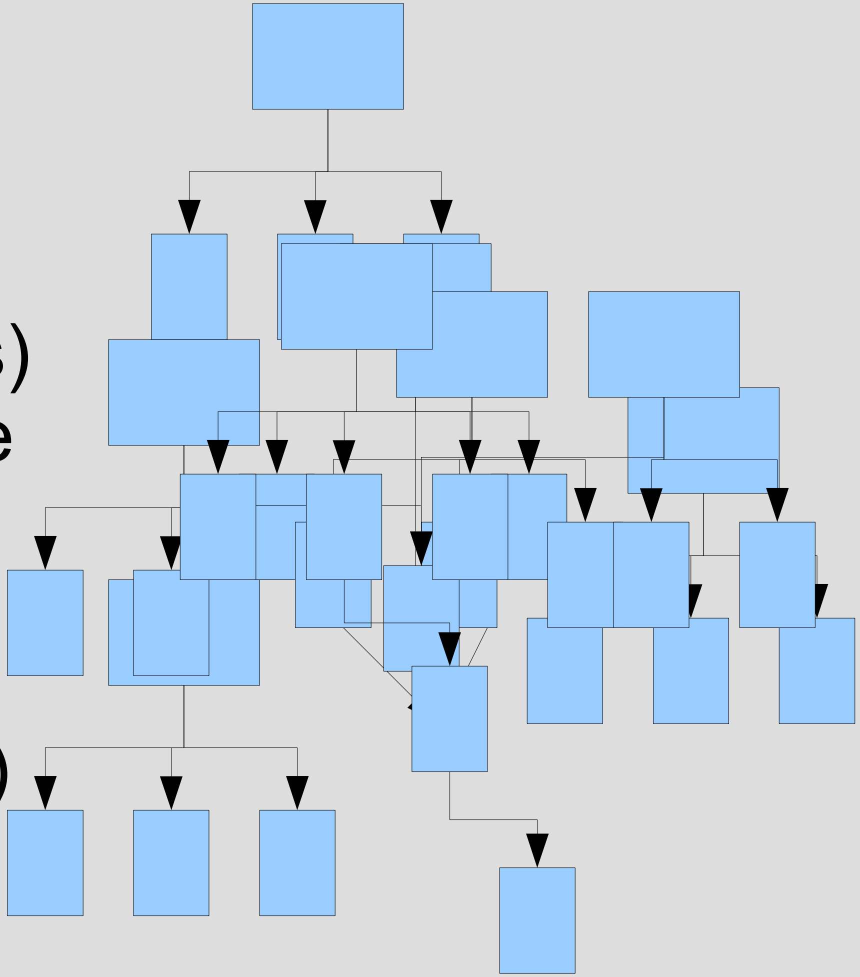
Page table

Page table



What does “a bit like a tree” mean?

- Okay, it isn't a tree
  - there are several trees in existence simultaneously (processes)
  - depth is not even (large pages)
  - multiple edges can point to the same leaves (page sharing, aliasing)
  - some nodes can be shared (shared kernel address space)
  - height can change (pae, long mode)
  - it may not exist (real mode)



Cache control

- invlpg
  - “this virtual address in the current address space is invalid”
- mov cr3
  - “everything you know is wrong”

- Too coarse!

Quick Intro to Shadow Paging

- Page tables/TLB specify guest virtual to guest physical mapping
- Hypervisor wants to translate guest physical addresses to host physical addresses
- Current processors can't do the additional translation in hardware

➔ The hypervisor computes the guest virtual to host physical mapping on the fly and stores it in a new set of page tables

Shadow paging challenges

- Handle all the complexities of x86 paging
- Keep the shadow page tables in sync with the guest page tables
- Do it fast

Goals for the KVM MMU

- Correctness
  - no assumptions about guest behavior
- Performance
- Acceptable worst case behavior
  - no nasty fallbacks if heuristics fail
- Code maintainability

Write protection

- We can't rely on invlpg and mov cr3 to tell us when we need to invalidate shadow page table entries
- So, we track guest page table modifications ourselves:
  - every shadowed guest page is write protected against guest modifications
  - if the guest tries to modify, we trap and emulate the modifying instruction
  - because we know the address, we can clear the associated shadow page table entry(ies)

Reverse mapping

- Write protection is not so easy
  - the same page may be already mapped by the guest in multiple locations
- So, we track writable mappings of **every** guest page
  - linked list hanging off page->private
  - each entry is a shadow pte
  - special case for one writable mapping
- When we shadow a guest page, we iterate over the reverse map and remove write access
- When adding write permission to a page, we check whether the page has a shadow

The shadow hash

- We sometimes need to find the shadows of a given page
  - when linking it into the shadow page table tree
  - to check if it can be made writable
- So we use the #1 data structure in computing: the hash table
- Key consists of
  - guest page frame number
  - this shadow's role in the universe
- We can have multiple shadow pages for a single guest page – one for each role!

Shadow page descriptor

- Every shadow page table has a descriptor
- The descriptor is reachable via page->private
  - Given a page table entry pointer, we can find out things about the page
- Contents
  - LRU link
  - Hash table link
  - Guest page number
  - Role
  - Slot bitmap
  - Parent pte list
  - Root ref count

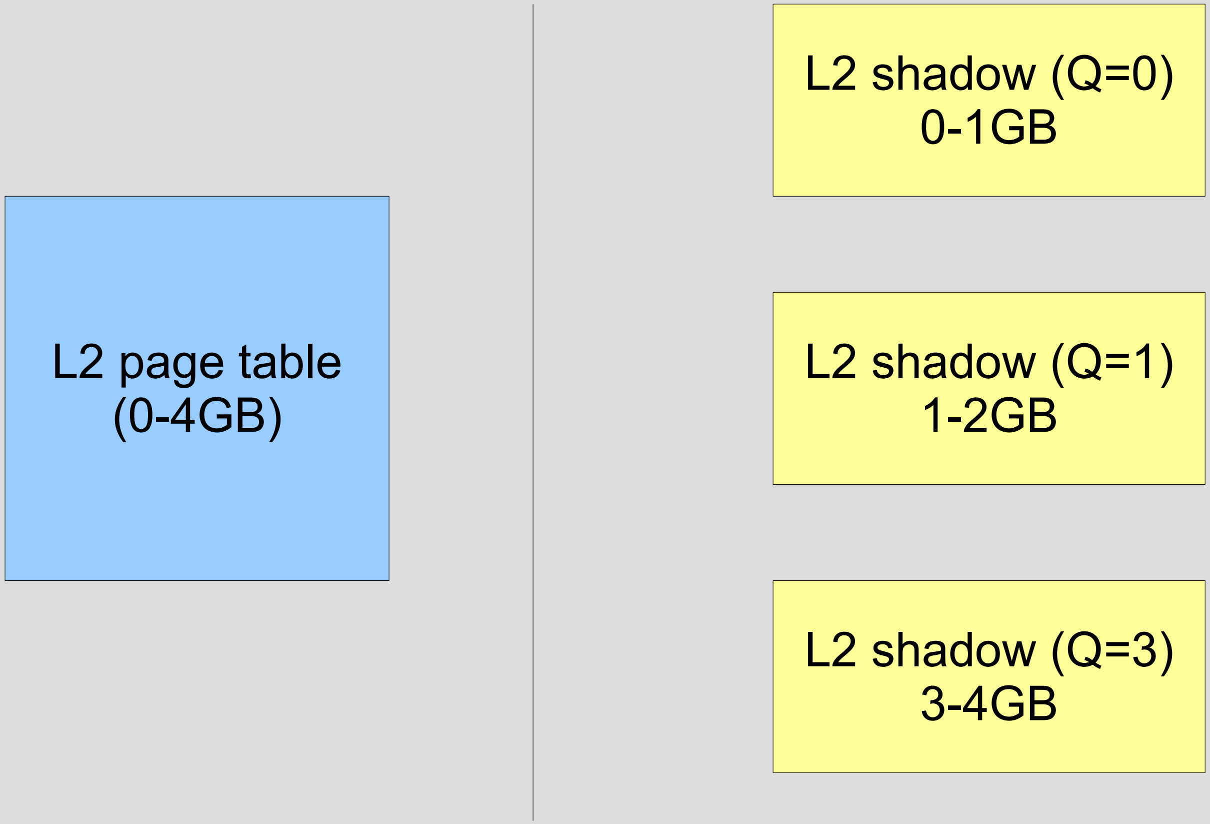
Shadow roles

- A word with many bitfields
  - what paging mode is in effect for this shadow (0/2/3/4)
  - what level in the paging hierarchy this page occupies (needed for cyclic paging structures)
  - inherited access rights
  - “metaphysical” bit
    - used for shadow pages which aren't directly related to guest pages: large page shadows and real mode shadows
    - metaphysical pages don't participate in some lookups
  - the quadrant
- Roles allow multiple operating modes to coexist

Quadrants

- The KVM MMU always uses 64-bit shadow page table entries
  - implies 512 page table entries per page
  - Level 1 page table maps 2MB
  - Level 2 page table maps 1GB
- But.
  - Guest can use 32-bit page table entries
  - Level 1 page table maps 4MB
  - Level 2 page table maps 4GB
- We need 2 or 4 shadows per guest page table!

Quadrants (cont'd)



L2 page table
(0-4GB)

L2 shadow (Q=0)
0-1GB

L2 shadow (Q=1)
1-2GB

L2 shadow (Q=3)
3-4GB

Shadows for different quadrants are completely independent

All those lists...

- rmap: from guest page to shadow ptes that map it
- Shadow hash: from guest page to its shadow
- Parent pte chain: from shaow page to upper-level shadow page
- Shadow pte: from shadow page to lower-level shadow page
- LRU: all active shadow pages

Dirty page tracking

- Maintains a bitmap of dirtied pages
- Used for framebuffer acceleration, live migration
- Activated per slot

- When mapping a page as writable, we mark it it as dirty
- Also, mark the shadow page table as having a pte mapping pages from that slot
- Later, when write protecting the slow, we scan all shadow pages, skipping those that don't map pages from the slot

SMP considerations

- All shadow code protected by a single lock
- When reducing permissions in a shadow pte (write->readonly, present->not present), we cause TLBs on all cpus that run a vcpu to be flushed
- One day, we may track which vcpus have accessed the shadow page

A day in the life of a page fault

- Page fault is intercepted by hardware
- Page fault handler invoked, calls guest page table walker
  - If it's a guest page fault, we're done
  - Walker remembers the guest page tables accessed
- Walk the shadow page table, instantiating page tables as necessary
  - Can involve an rmap walk and write protecting the guest page table
- Instantiate the new shadow pte

Page table updates

- The guest updates its page tables from time to time
  - Like, after every guest page fault
- We already need to zap the shadow pte on update, but in certain cases, we can do better
  - Instantiate the new pte immediately
  - Saves a vmexit

Flood protection

- The guest may issue a lot of writes to a guest page table (fork())
- This results in a lot of emulations
- Detect this, and unshadow the page

Shadow teardown

- If a guest page table is no longer a page table, write protection slows things down
- How to detect?
  - Misaligned writes
  - User-mode writes
  - Floods