Memory Virtualization

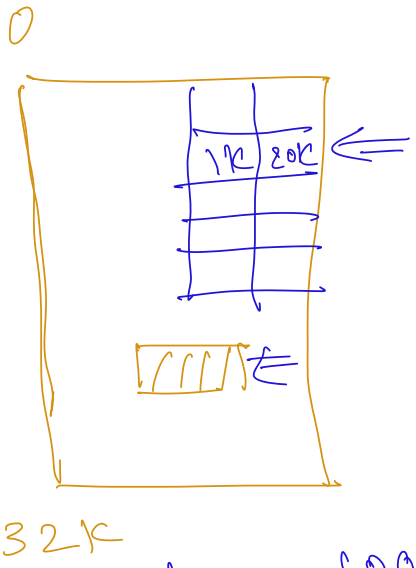
- Review how mem. virtualization works in normal O.S. without V.Ms.

Normal O.S.

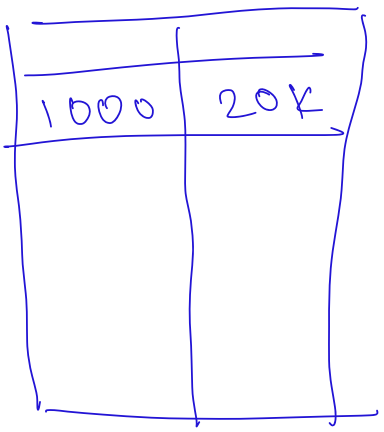
- virtual memory, each process gets own address space
- Every address that process sees is fake
- e.g. read 1000 (not actual 1000)
- All addresses of processes are virtual addr.

VIRT

PHY



- Goal: translate virt address 1000 to physical address 20K
- Done via page tables.

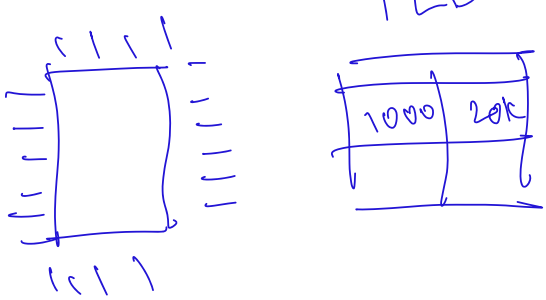


← This table also in physical memory not in special hardware.

Logical

- To translate 1,000, we would need atleast 2 memory accesses.
- First read page table, then actual data.
- Already 100% overhead.
- To get rid of this: special cache near chip.

TLB



← very fast, store page table entry.

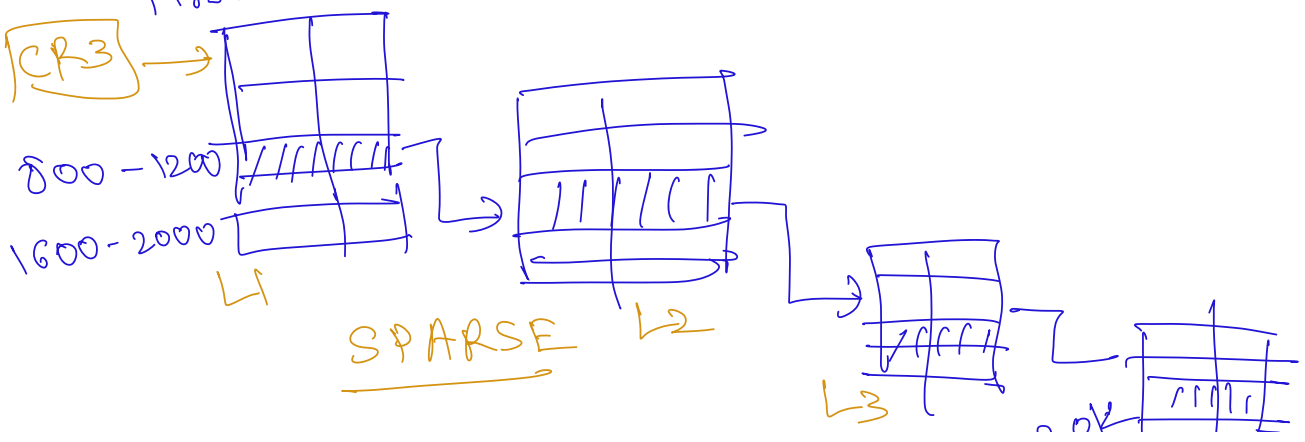
Now, when we want to read 1000, don't go to memory, go to TLB, which is very fast.

cost = cost to read TLB + cost to read from mem.

- When is TLB populated?
  : Very first time we try to translate using page tables, we populate TLB.
  - Next time: direct from TLB.

→ Problem: page tables can be very big.
processes: 64-bit address spaces
imagine 1 entry for every virt. page to phy page: lot of entries:
32-bit address space: 1 GB PAGE TABLE
- Lot of memory !!!, need it contiguous.

∴ Hierarchical page tables.
- break one giant page table into a hierarchy.
- we have a tree.
first few bits, index into first table.



- only allocate on-demand.

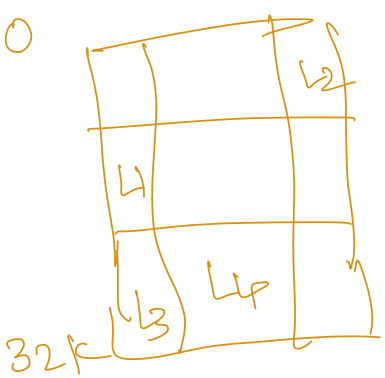
e.g. 1600–2000 process never used virt. addr in this range.

Never allocate all tables.

+ saves lot of memory. (GB page table shrinks significantly

+ Each chunk 4KB in size.

- Divide physical mem. into 4KB chunks & allocate 4KB at a time



- Increased overhead: 400% overhead
- Read 4 locations to get address

∴ TLB very important.

- computers very slow.

- Lot of research in how big it should be
- only so many entries ∴ how to increase range, etc

- Recap: page tables: 5 levels,

TLB

page table walk: hardware knows how to do this.

- There is agreement with software & hardware as to how a page table should look like.

∴ on read 1000, hw first looks at TLB. If miss, hw knows how to walk page tables, get the translation, put in TLB, read from mem. & return to application.

Register called CR3 that points to L1.

how many entries in TLB?

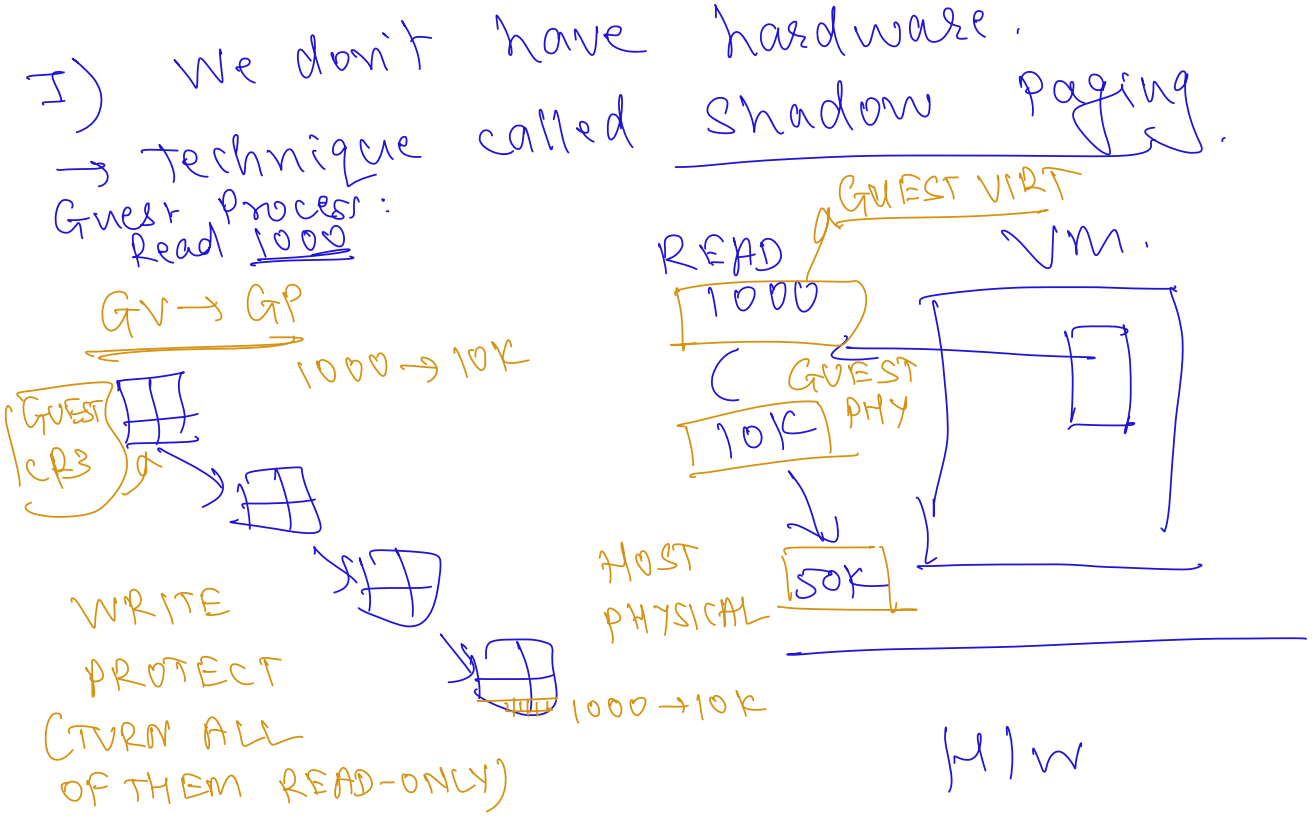
Software TLB?

* Page fault what happens?
* malloc() what happens?

How to virtualize everything

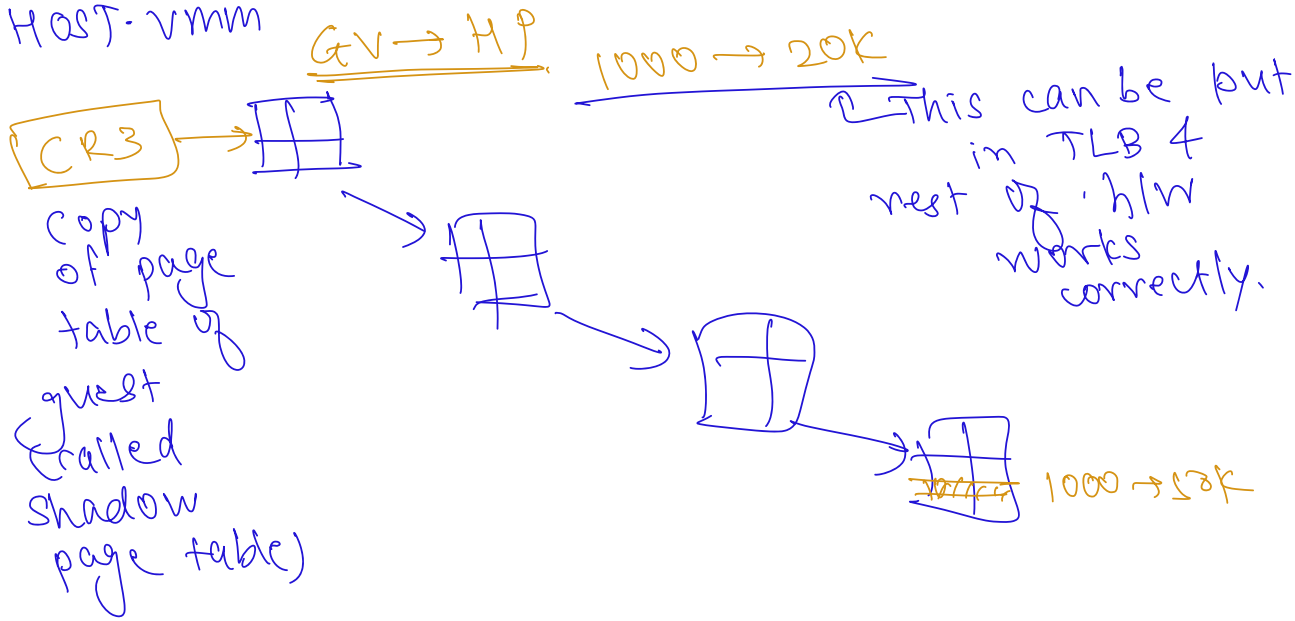
I) We don't have hardware.

→ Technique called Shadow paging.



GUEST VM

HOST - VMM



- VM has its own view of hardware: it thinks it is running on hardware.
- Guest physical: Physical address as seen by the guest.
- Process inside guest sees 1,000.
- O.S. inside guest sees 10K ← FAKE
- Actual h/w sees 20K
- To read 1,000 from physical h/w, we need to translate 1,000 to 20K.

∴ GUEST VIRT → HOST PHY

- Every time guest tries to write to page tables (on a page fault), a trap will be generated & VMM will ensure that its page tables are in sync with guest's page tables.
- We are able to intercept each change to page table in VM, & add entries in shadow page tables.

- e.g. Guest Process: Read 1000: h/w will try to read TLB & miss.

- CR3 points to shadow page table.
- Inside GUEST, there is GUEST CR3
- Not able to find entry: FAULT

VMM sends PAGE FAULT EXCEPTION to guest.

→ Guest thinks that it tried to walk its own page tables & didn't find translation.

→ Guest wants to add translation.

→ tries to map 1000→10K, & traps. Because it is write protected.
- cannot write.

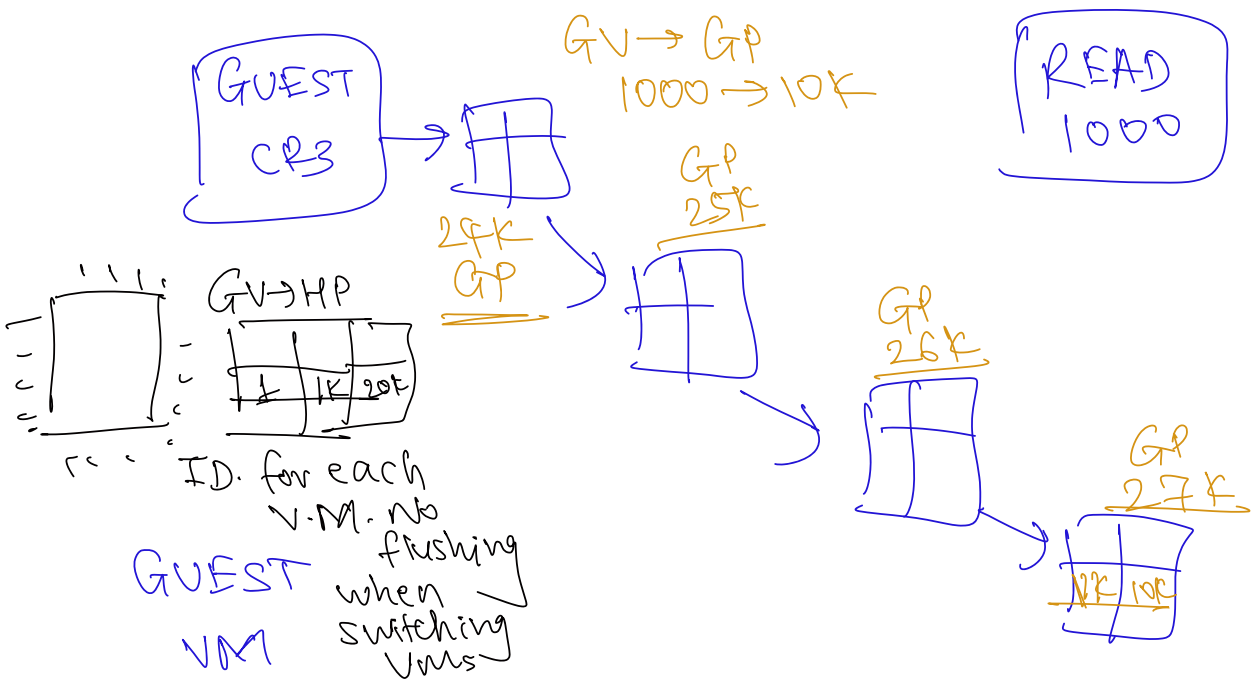
→ VMM gets trap, expected. Allocates actual page, adds 1000→20K in PTE & then add 1000→10K in GUEST PTE

→ To the GUEST: entry is present, succeeded.
re-execute read 1000, works!

Translation betn
GUEST PHY → HOST PHY somewhere else.

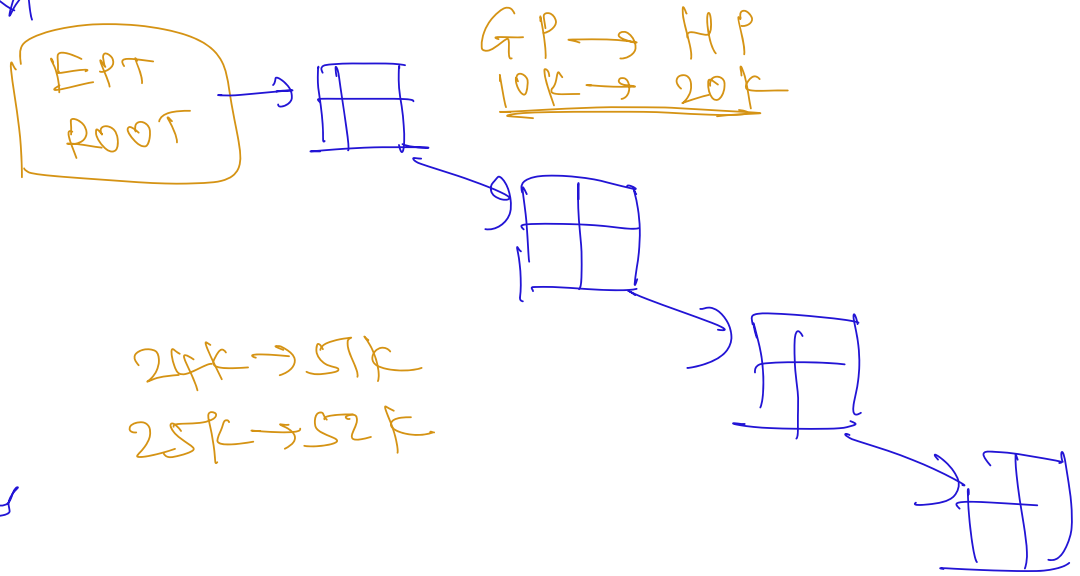
HARDWARE SUPPORT

Extended Page Tables



VMM

New reg.: EPT ROOT. This reg. is what hlw assisted virt. give us.



- New Register which points to root of extended page tables
  - Case 1: Hit in TLB: No problem, prog. can continue.

- Case 2: Miss in TLB. Interesting case
  - Go to guest CR3
  - Do hardware page walk at guest.
  - Every address is guest PHY. (we cannot directly access its memory locations.)
  - Hardware knows what to do. It will look for translation of GP address.
  - Goes to VMM, EPT ROOT, do a hardware walk for 24K
  - gets data, reads it & that data has the root GP addr of next level's page dir. root.
  - etc. figure.

- More work than shadow page tables
- For every entry in Guest P.T. hierarchy, full hardware walk in VMM.
- e.g. n levels in guest P.T. hierarchy
  n levels in VMM P.T. hierarchy.

$$n \times (n+1) + n$$

↑ # of memory accesses.

- More memory accesses, but simpler for software
- No trap required.
- O.S. in shadow P.T. requires trap.
- When shadow P.T. are set up, shadow page tables faster.
- cost to fill TLB is costlier in EPT.

** only one EPT per guest VM. (since it maps GP → HP)

e.g. 4 VMs with 10 processes each:

80 page tables for Shadow P.T.
44 page tables for EPT case.

- EPT: Reducing memory requirement at the cost of more computation/ more memory accesses. & for simplicity of software.