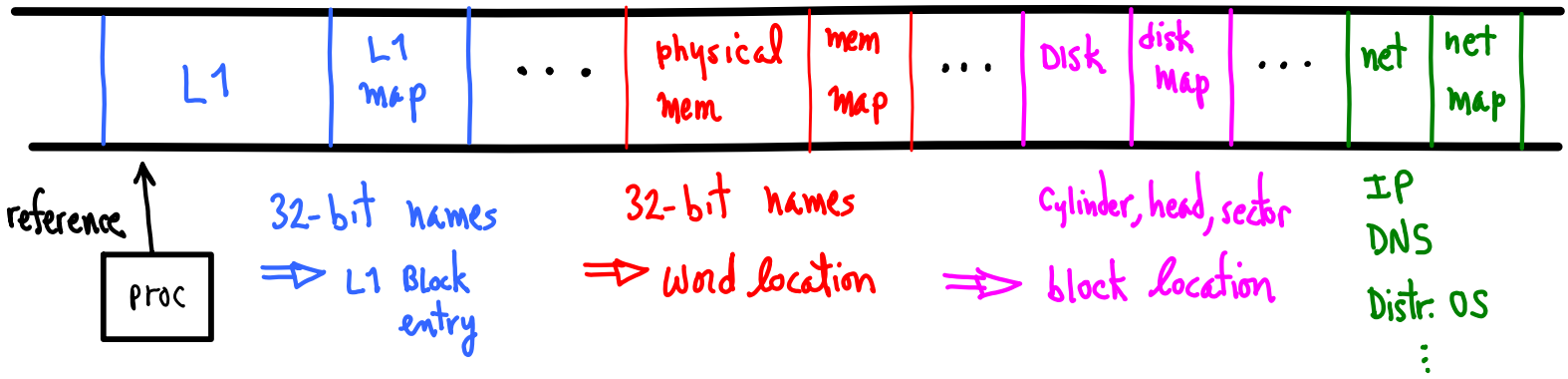
Finding things by name, generally

It's all TM Tape (move L, move R)
⇒ We can always search

Fast access
===> use map to find object

HW == SW
===> map is in HW or SW or combo

Extend range
===> longer, hierarchical names



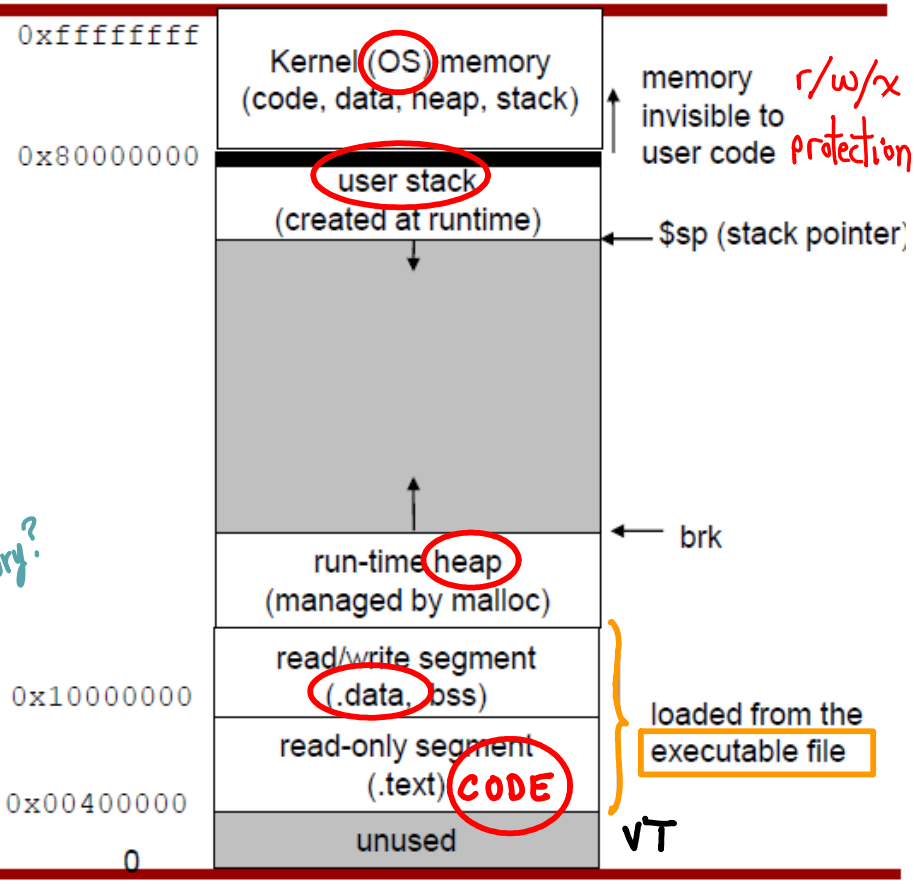
How is map embodied:
--- L1?
--- Memory?

Virtual Memory

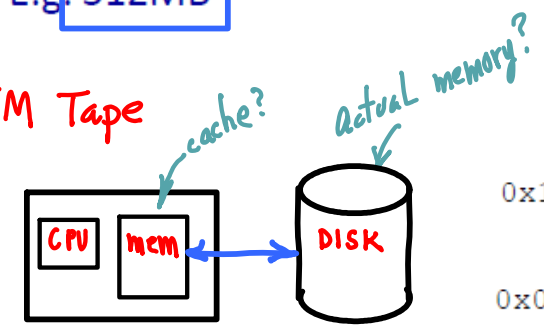
Motivation #1: Large Address Space for Each Executing Program

32-bit Addresses

- Each program thinks it has a ~2^{32}-byte address space of its own 4GB
  - May not use it all though...
- Available main memory may be much smaller
  - E.g. 512MB

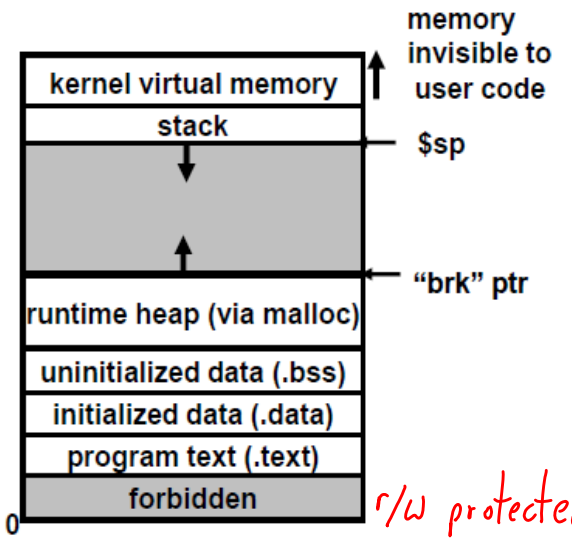


MEM = TM Tape

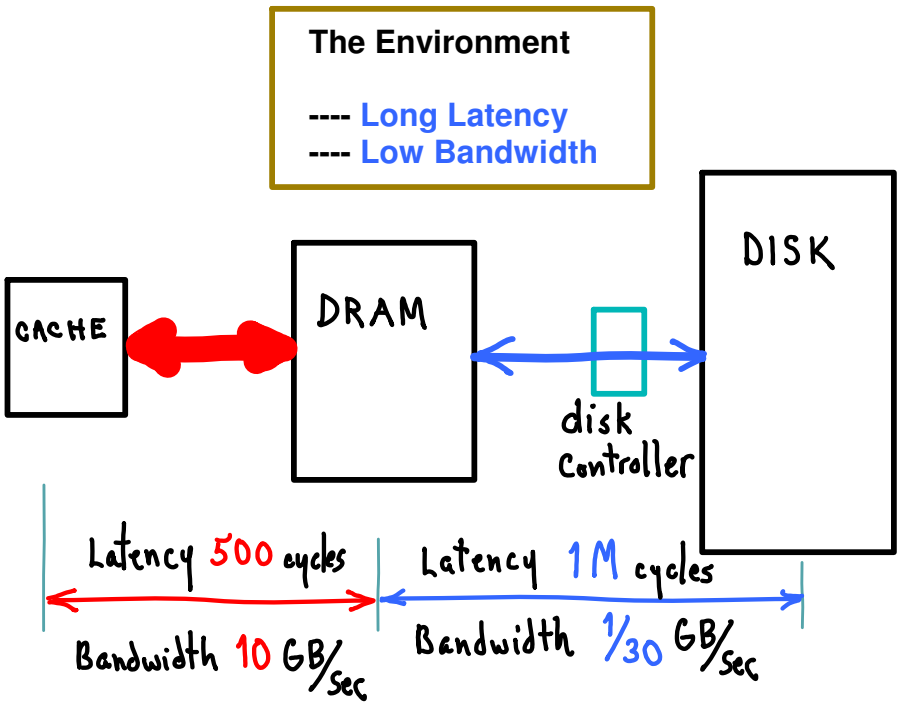


Motivation #2: Memory Management for Multiple Programs

- At any point in time, a computer may be running multiple programs — processes / jobs / tasks
  - E.g., Firefox + Thunderbird
  - See discussion on processes in following lectures
- Questions:
  - How do we avoid address conflicts?
  - How do we protect programs from each other?
  - How do we share memory between multiple programs?
    - Isolation and selective sharing



r/w protected vectors ...



Performance

$T_{avg} = T_{hit} + (\text{miss rate}) \times T_{penalty}$

huge

minimize

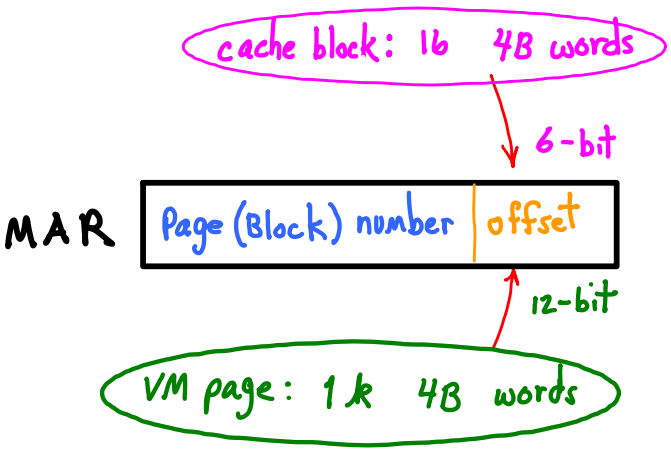
Big blocks	:	spatial locality
Big cache	:	lower miss rate
Associative	:	lower miss rate
Write back	:	less bandwidth
Multiple levels	:	lower avg penalty

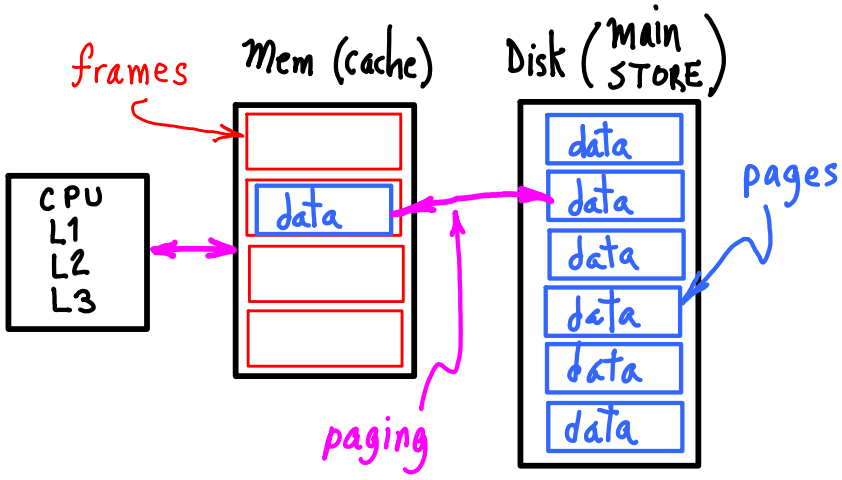
disk controller

- pre fetch + write buffer
- cache disk blocks
- schedule requests

VM: names & objects

Names: memory addresses (as before) and disk addresses;
Objects: data/instruction "pages" (the same, but bigger than cache blocks)





A name is not a place

A name is a key to look up its location,
Mapping: key ===> location

Why Mapping/Translation?

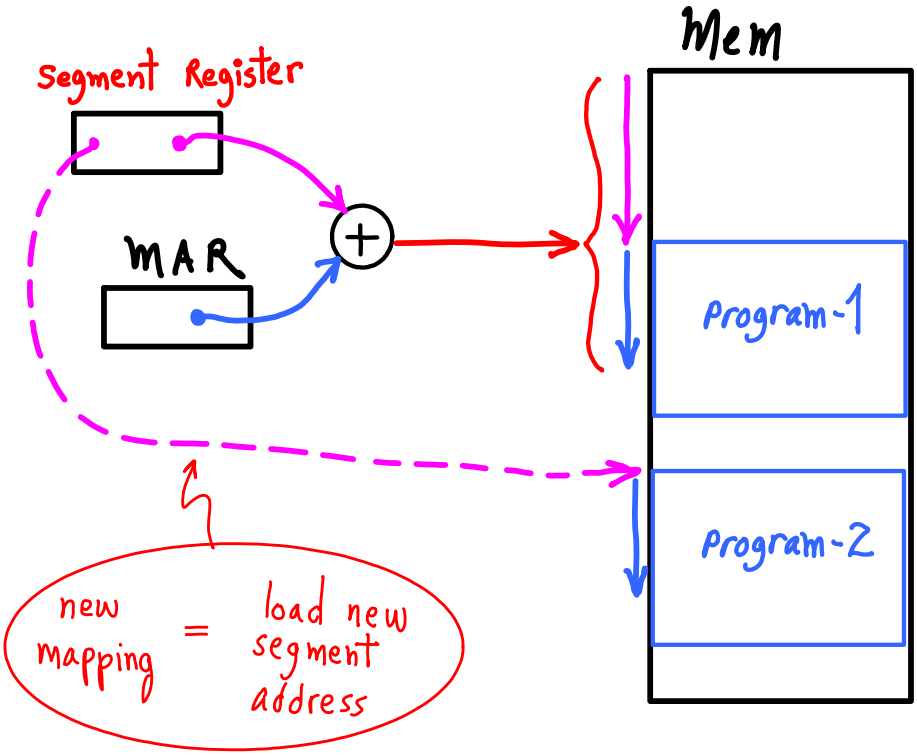
--- Multiple Programs

--- Program written w/o knowing where it will be in memory

--- OS moves programs around

--- Problem w/ physical memory: relocation == load editing?

Simple Translation



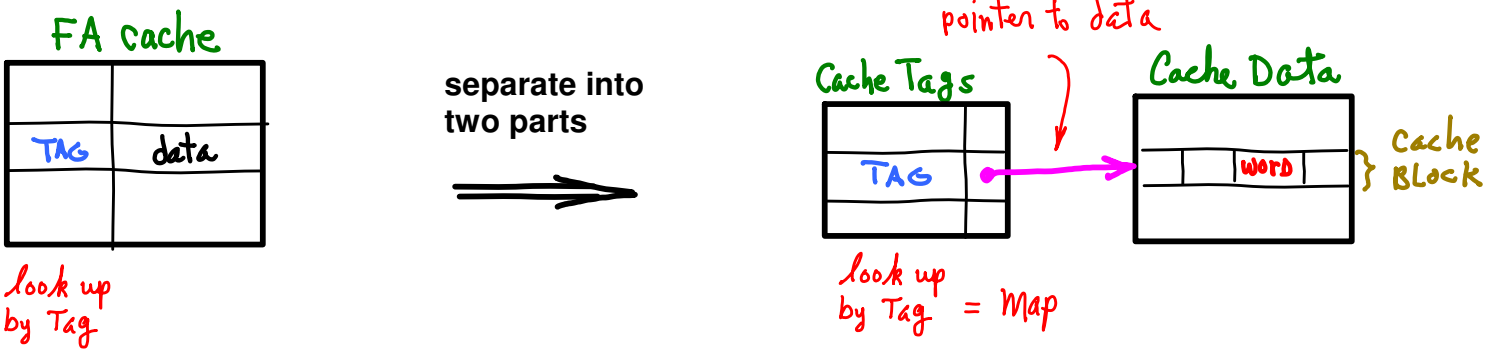
Segment register could point anywhere in memory

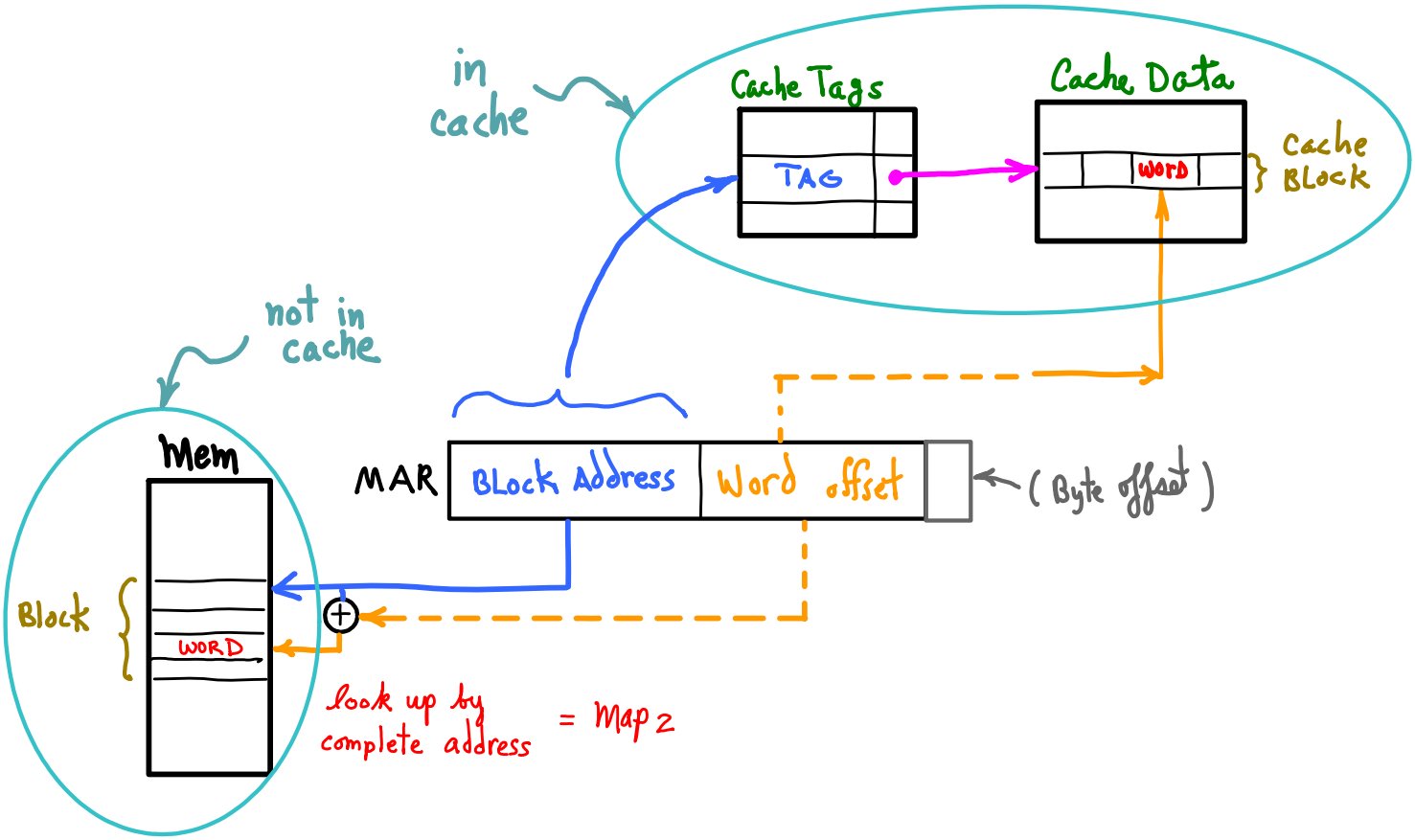
MAR content is address relative to Segment register's pointer.

Swap Segment register's value ===> Switch to new memory area

Programs both "live" in own space, but each "thinks" its space starts at address 0.

It's all caching (review, new view of old stew)



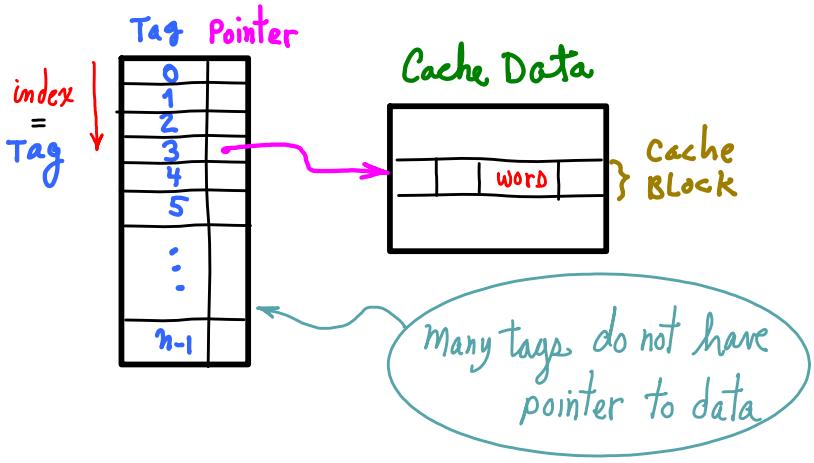


A NEW WAY (equivalent to above)

TAG/pointer table:
an entry for every possible tag

TAG == block address == index
===> eliminate TAG column!
===> no TAG search!

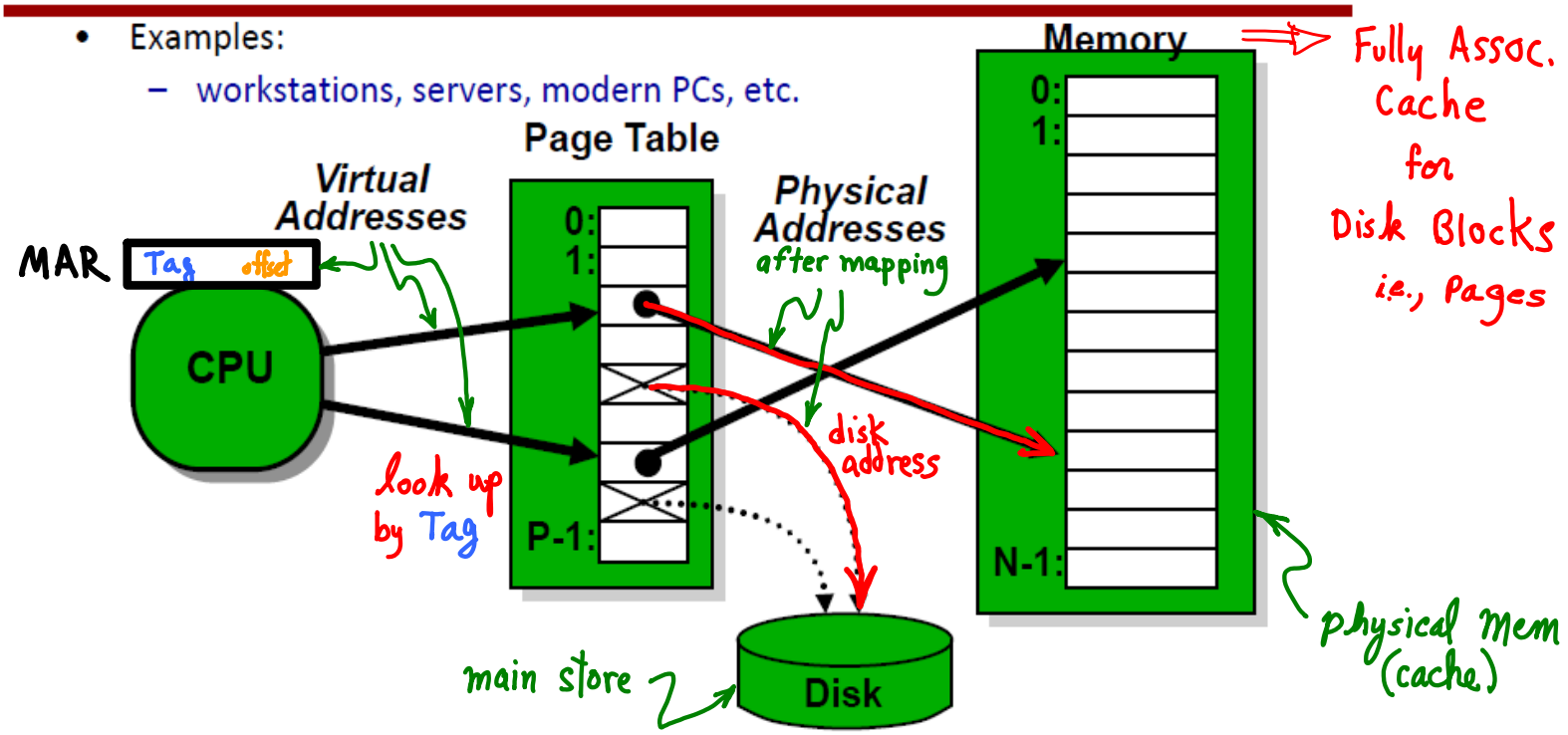
cache block == VM Page
pointer table == Page Table
cache data == Disk/Memory Page



A System with Virtual Memory

Address Translation

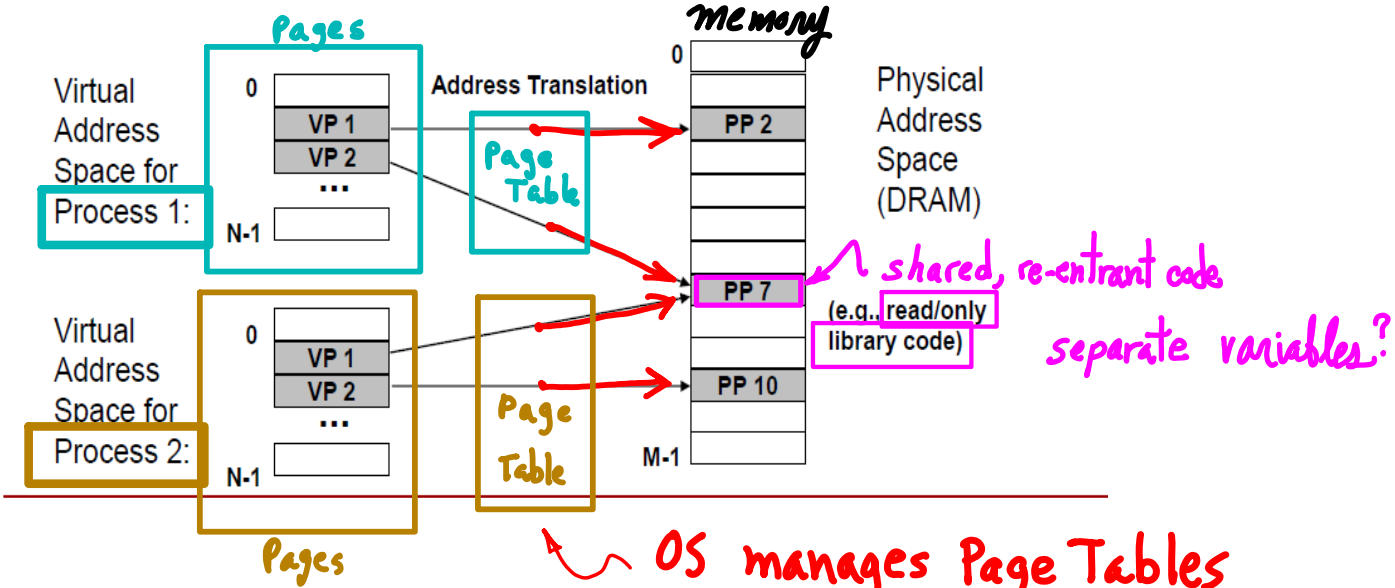
- Examples:
  - workstations, servers, modern PCs, etc.



Address Translation: Hardware converts *virtual addresses* to *physical addresses* via an OS-managed lookup table (*page table*)

Separate Address Spaces Per Program

- Each program has its own virtual address space and own page table
  - Addresses 0x400000 from different programs can map to different locations or same location as desired
  - OS control how virtual pages as assigned to physical memory



Page Table *issues*

--- Where, physically?

memory? SRAM? Hardware?

table in memory?
how many memory accesses to read a word? (ignoring L1, L2, ...)

--- Page Tables R/W?

program rewrites page table (accidentally)?

not R/W? how to store pointer values?

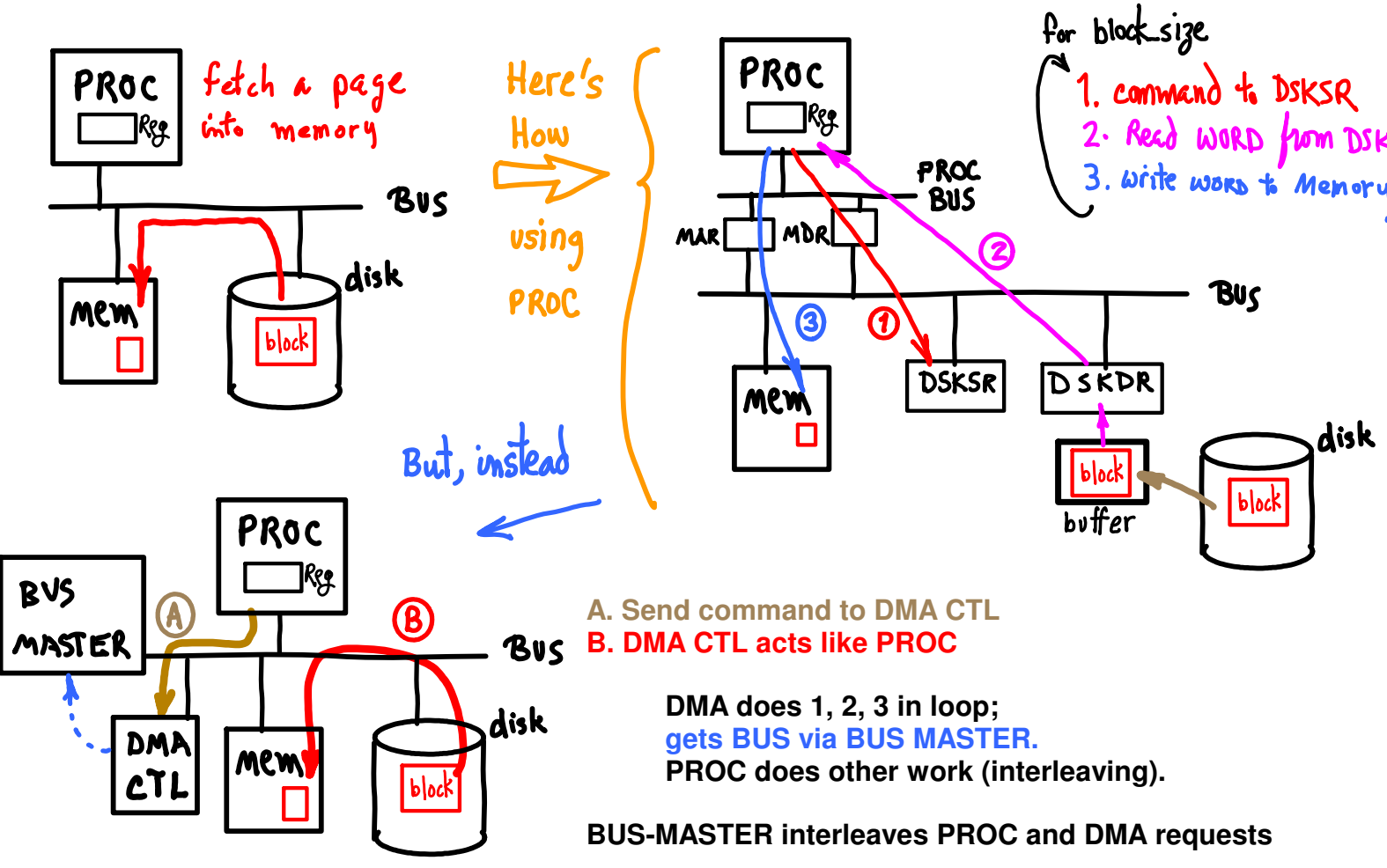
===> protection bits per page: Kernel Mode: R/W, User Mode: not R/W
===> Where are bits? How accessed?

--- Share physical memory?

interleaving: **long latency, do other work.**
OS has work to do, too.

--- I/O?

use virtual addresses? Memory mapped **device registers?**
long, slow **I/O for disk blocks (pages)?**



A. Send command to DMA CTL
B. DMA CTL acts like PROC

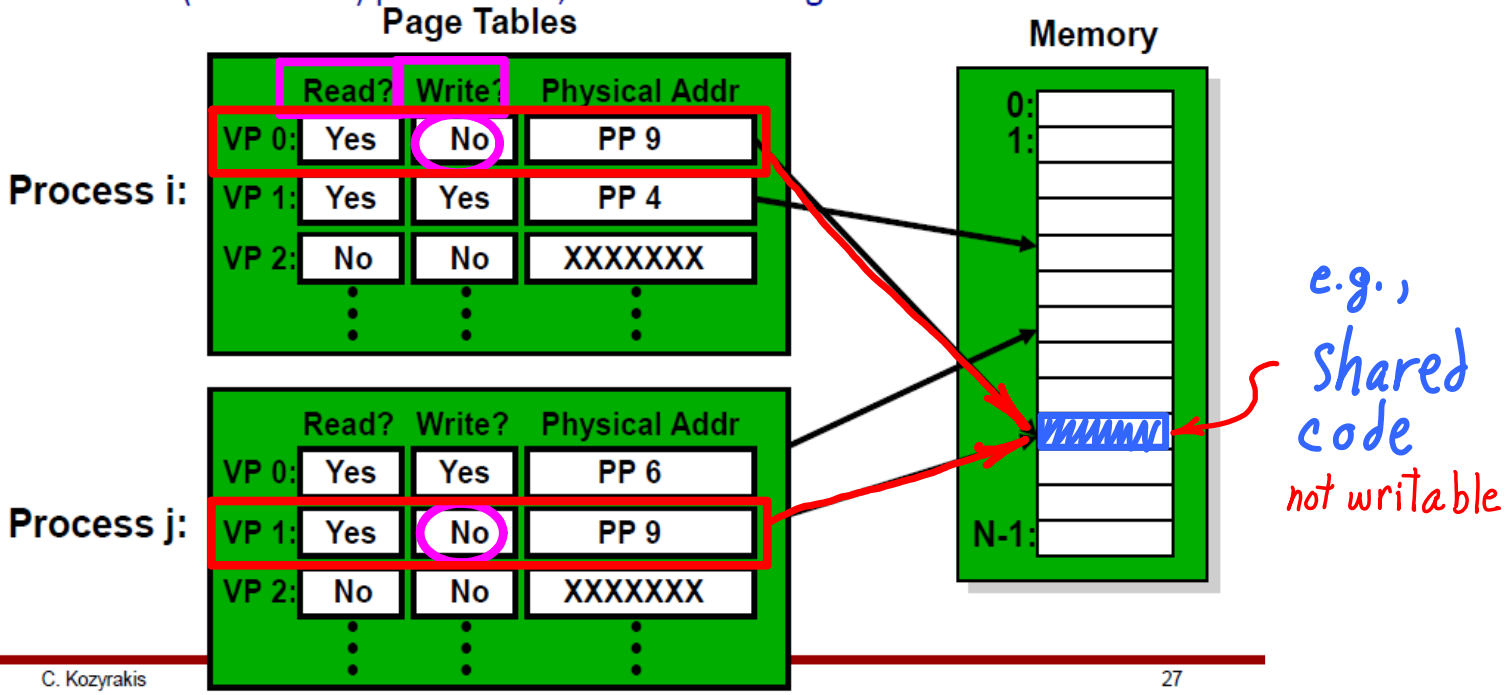
DMA does 1, 2, 3 in loop;
gets BUS via BUS MASTER.
PROC does other work (interleaving).

BUS-MASTER interleaves PROC and DMA requests

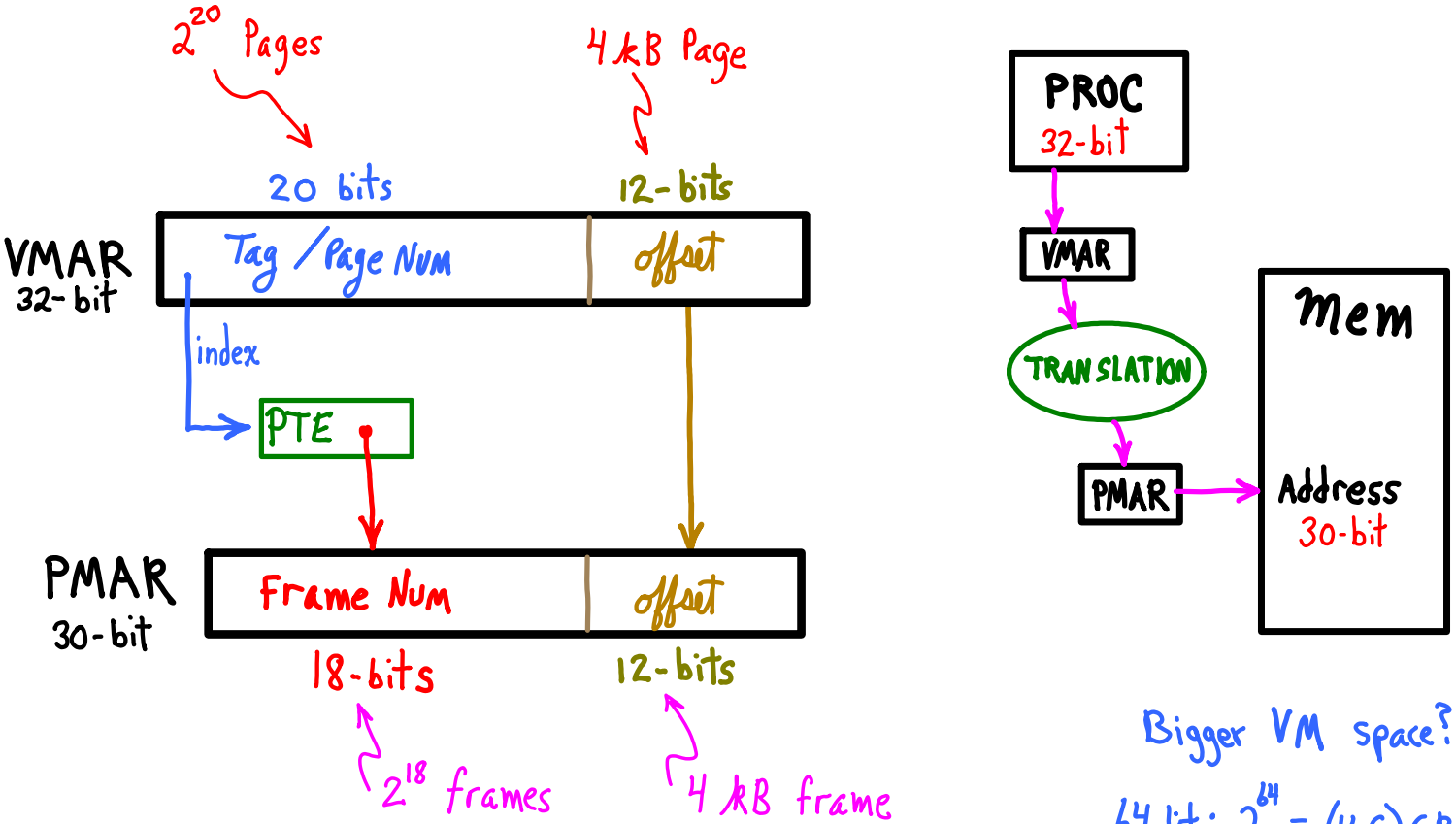
Protection through Access Permissions

Add more bits to Page Table Entry (PTE)

- Page table entry contains access rights information
  - RW (read-write) permissions, enforced during translation



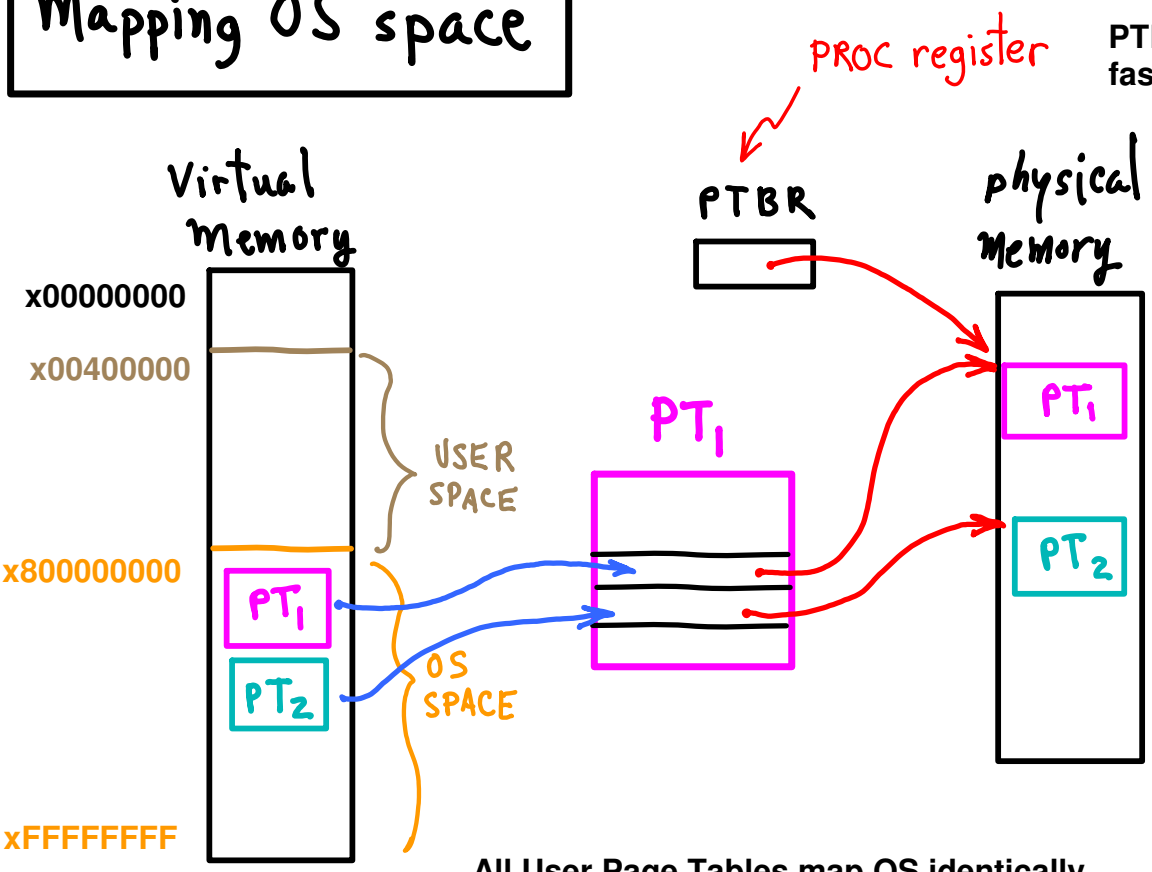
Translation: VM address → Physical address



Bigger VM space?

64-bit: 2^{64} = (16G)GB

Mapping OS space

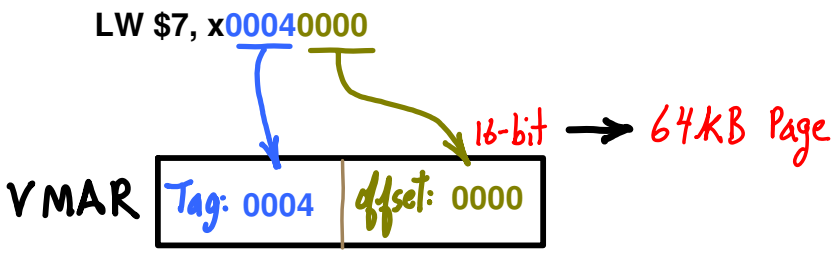


All User Page Tables map OS identically.

OS turns off VM translation to directly access physical memory.

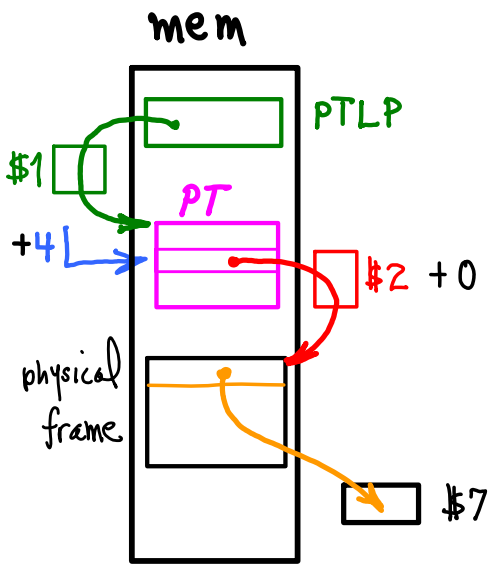
How many memory references?

LW $7, x00040000




```
LW $1, Page-Table-Location-Pointer  ;-- get addr of PT
LW $2,  4( $1 )                     ;-- get PTE
LW $7,  0( $2 )                     ;-- get data
```

effectively, the actions done in HW

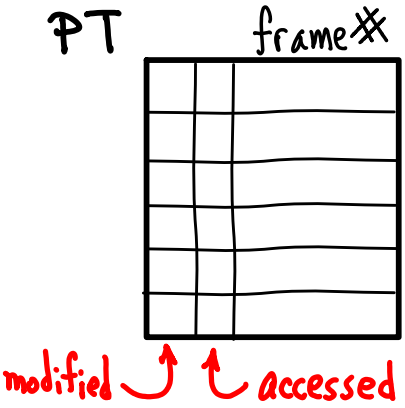


Speed it up:

1. **PTBR** <== **Page-table-location-pointer**
   Do this once at program startup

2. **Cache PTEs!**

Replacement Policy



WRITE-BACK
LRU approximation

W: modified <=== 1

R/W: accessed <=== 1

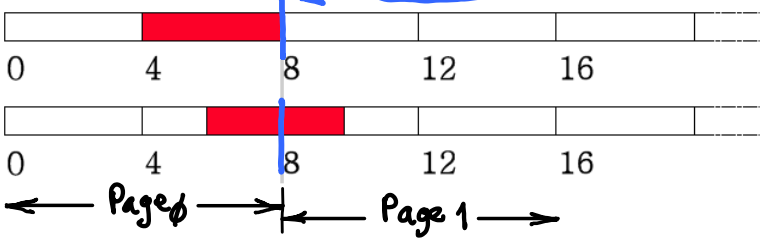
k ticks: accessed <=== 0

Page Miss:
--- evicted page (order of preference):
---- 1. dirty == 0, accessed == 0
---- 2. dirty == 0, accessed == 1
---- 3. dirty == 1, accessed == 0
---- 4. dirty == 1, accessed == 1

VM: Issues with Unaligned Accesses

Page boundary

- Memory access might be aligned or unaligned



Aligned 4B Word

Un-aligned 4B Word

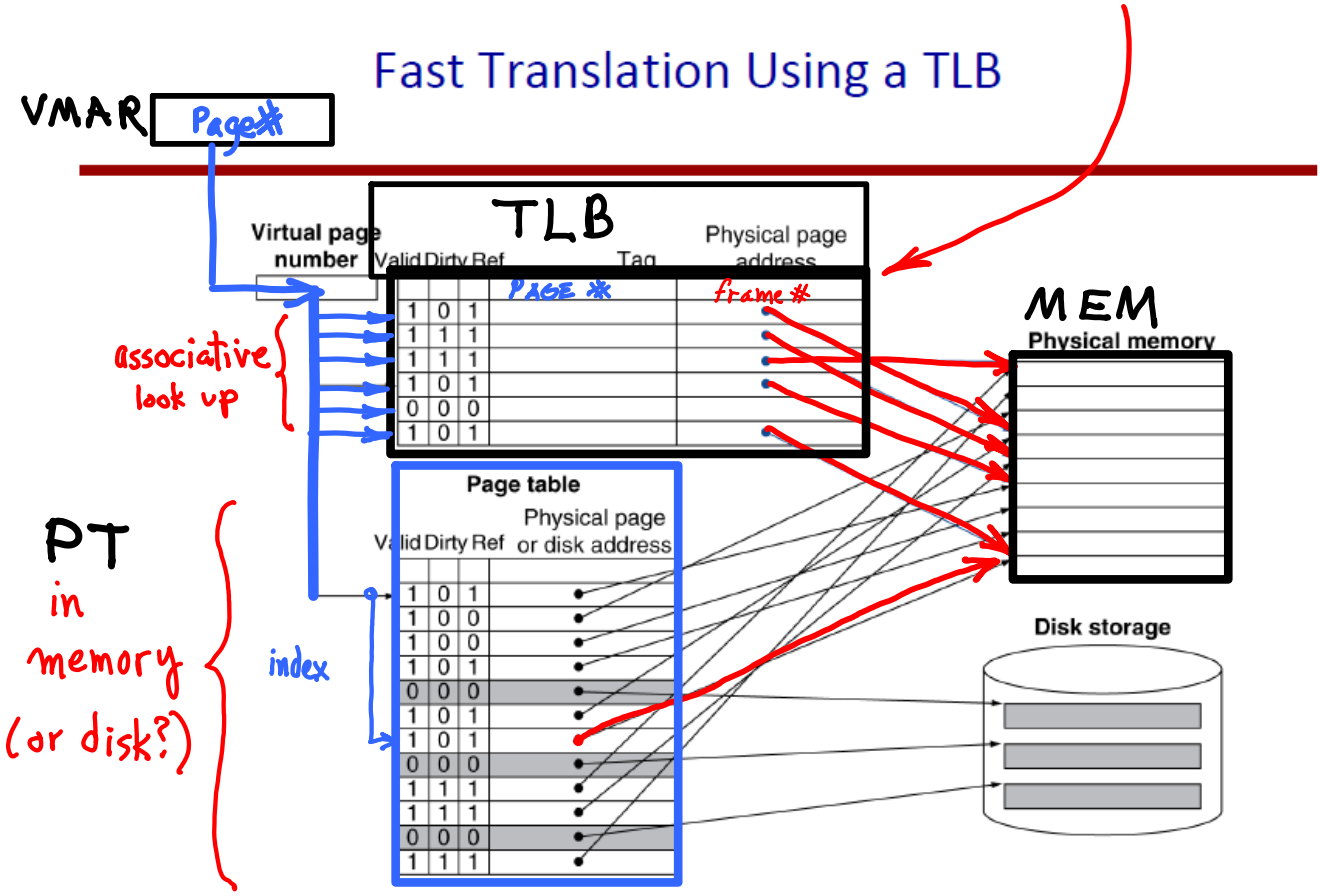
- What happens if unaligned address access straddles a page boundary?
  - What if one page is present and the other is not?
  - Or, what if neither is present?
- MIPS architecture disallows unaligned memory access
- Interesting legacy problem on 80x86 which does support unaligned access

what is the problem? How do we complete the access?

Caching PTEs

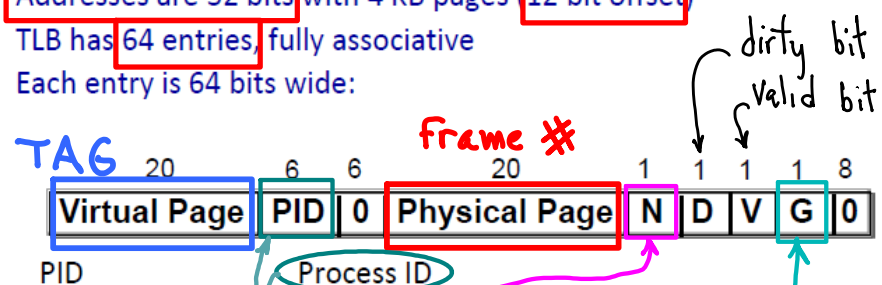
Fully Associative Cache for PTEs
Page # == TAG
Data == Physical Frame #
valid bit
LRU bits, ...

Fast Translation Using a TLB



TLB Case Study: MIPS R2000/R3000

- Consider the MIPS R2000/R3000 processors
  - Addresses are 32 bits with 4 KB pages (12 bit offset)
  - TLB has 64 entries, fully associative
  - Each entry is 64 bits wide:



PID	Process ID
N	Do not cache memory address
D	Dirty bit
V	Valid bit
G	Global (valid regardless of PID)

shared page, OS, library, ...

memory mapped I/O: always go to mem-io bus, not cache.

TLB Misses → TLB exception handler

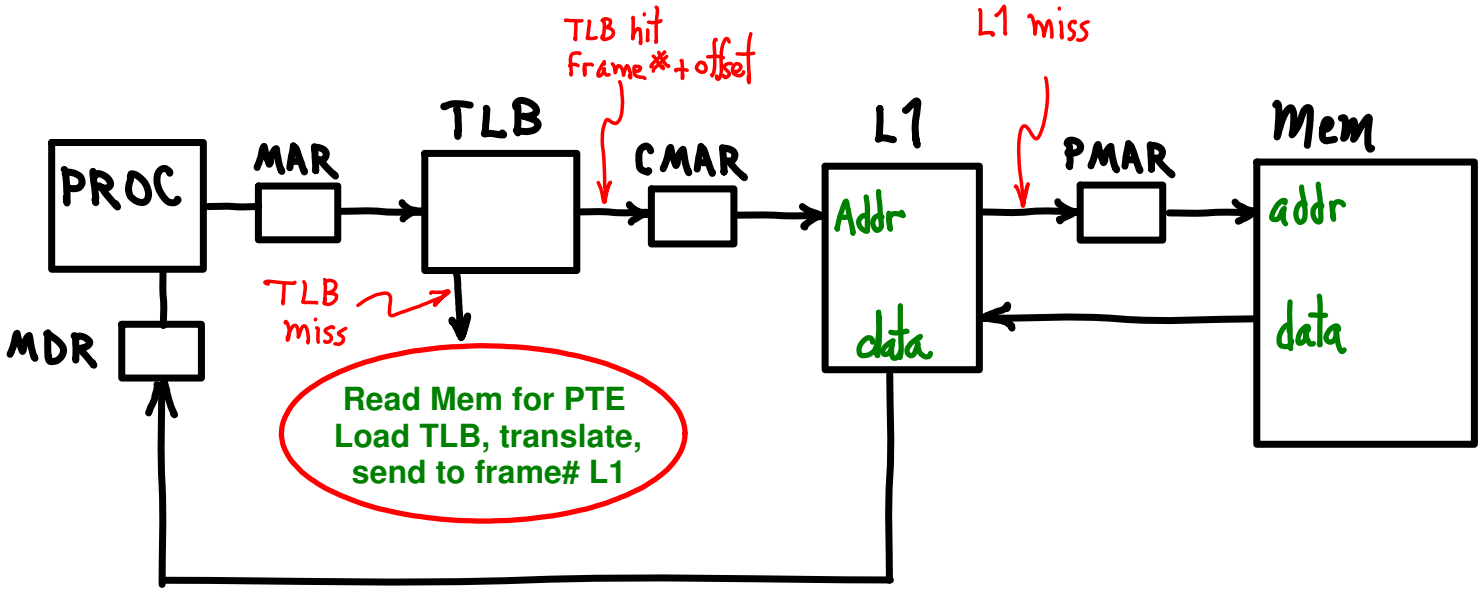
Read PT, get PTE

- If page is in memory
  - Load the PTE to TLB and retry instruction
  - Could be handled in hardware ?
    - Can get complex for more complicated page table structures ?
  - Or in software
    - Raise a special exception, with optimized handler
    - This is what MIPS does using a special vectored interrupt

- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction

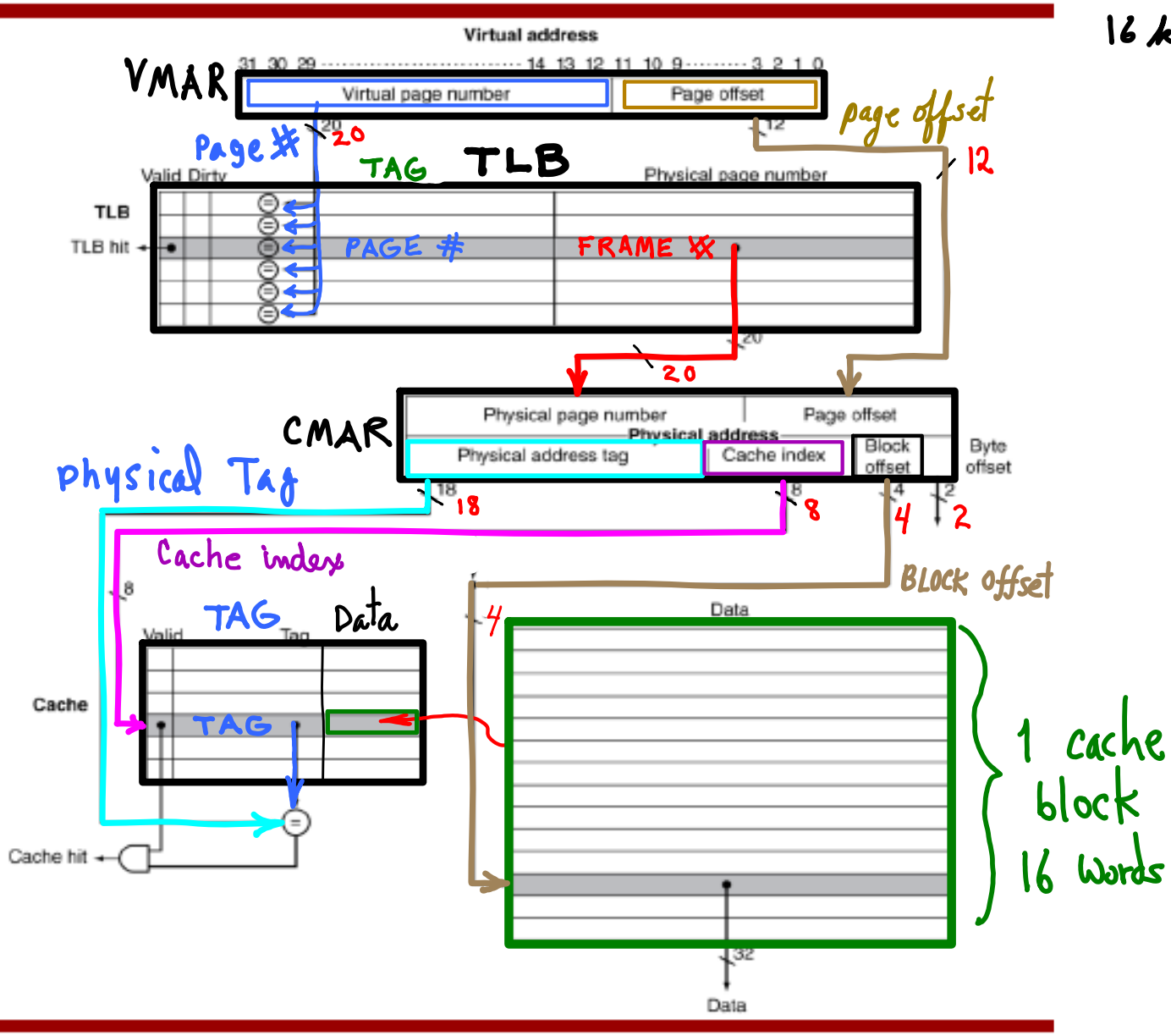
Load PTE to TLB

TLB + Cache

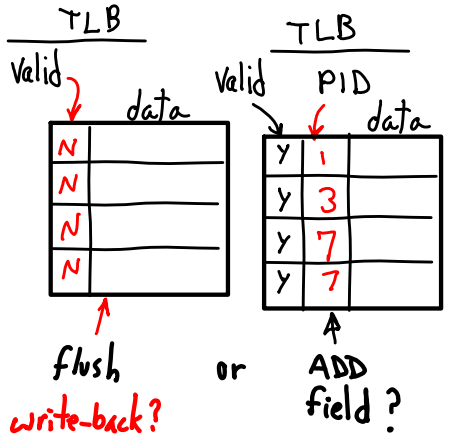


32-bit example | 256-block DM cache, 16 4-B words/block

16 kB



TLB Caveats



- What happens to the TLB when switching between programs
  - The OS must flush the entries in the TLB
    - Large number of TLB misses after every switch ← OR
  - Alternatively, use PIDs (process ID) in each TLB entry ↓
    - Allows entries from multiple programs to co-exist
    - Gradual replacement

- Limited reach
  - 64 entry TLB with 8KB pages maps 0.5 MB of address space
  - Smaller than many L2 caches in most systems
  - TLB miss rate > L2 miss rate!
  - Potential solutions
    - Multilevel TLBs (just like multi-level caches) ?
    - Larger pages ?

TLB is small ⇒ misses

⟹ Bigger pages ?

Page Size Tradeoff

- Larger Pages
  - Advantages
    - Smaller page tables
    - Fewer page faults and more efficient transfer with larger applications
    - Improved TLB coverage
  - BUT – Disadvantages
    - Higher internal fragmentation →

big page

unused, not accessed
= wasted physical mem

- Smaller Pages
  - Advantages
    - Improved time to start up small processes with fewer pages
    - Internal fragmentation is low (important for small programs)
  - BUT – Disadvantages
    - High overhead in large page tables
- General trend toward larger pages
  - 1978: 512 B, 1984: 4 KB, 1990: 16 KB, 2000: 64 KB → GFS Google File System 64 MB!

Multiple Page Sizes — OS chooses

- Many machines support multiple page sizes
  - SPARC: 8KB, 64KB, 1 MB, 4MB
  - MIPS R4000: 4KB – 16 MB

- Page size dependent upon application
  - OS kernel uses large pages
  - User applications use smaller pages

} OS sets MMU

BUT

- Issues
  - Software complexity
  - TLB complexity
    - How do you do match if not sure about the page size?

Final Page Table Problem: Its Size

- Page table size is proportional to size of address space

$\frac{2^N}{2^m}$ = #(N-bit Byte addr) / #(Bytes / page) = 2^{N-m} entries

- Example: Intel 80x86 Page Tables
  - Virtual addresses are 32 bits, pages are 4 KB $m = 12$
  - Total number of pages: $2^{32} / 2^{12}$ = 1 Million $2^{32-12} = 2^{20}$
  - @ Page Table Entry (PTE) are 4B
    - 20 bit Frame address, dirty bit, accessed bit, valid bit, access bits...
  - → Total page table size is therefore $2^{20} \times 4$ bytes = 4 MB
    - BUT, only a small fraction of those pages are actually used!

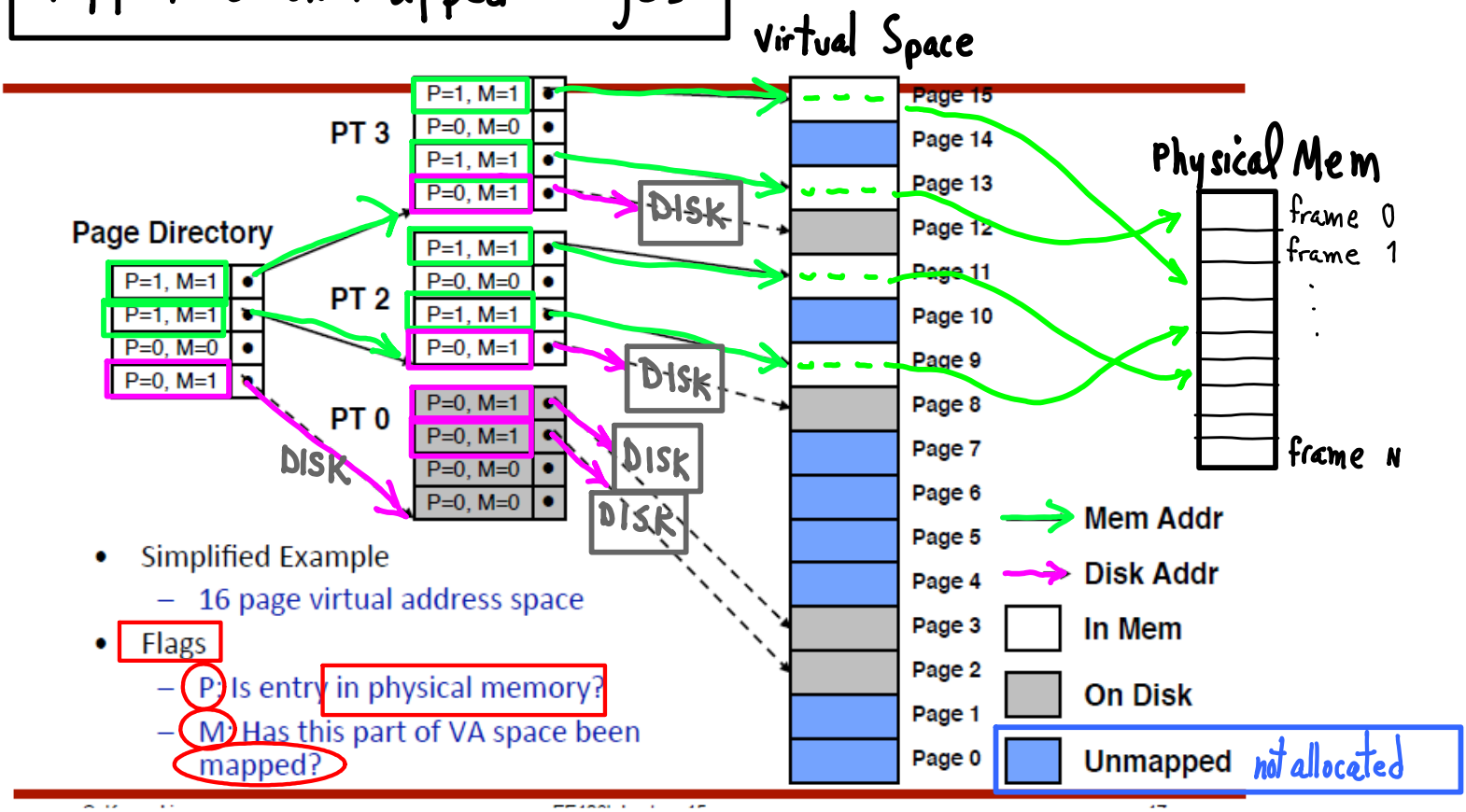
who uses all 2^{32} addresses?

- Why is this a problem?
  - The page table must be resident in memory (why?) ← map to disk, valid, ...
  - What happens for the 64-bit version of x86?
  - What about running multiple programs?

$2^{N-m} = 2^{64-12} = 2^{52}$ entries

$(2^{20})(2^{30})$ = (M)(G) entries! per table

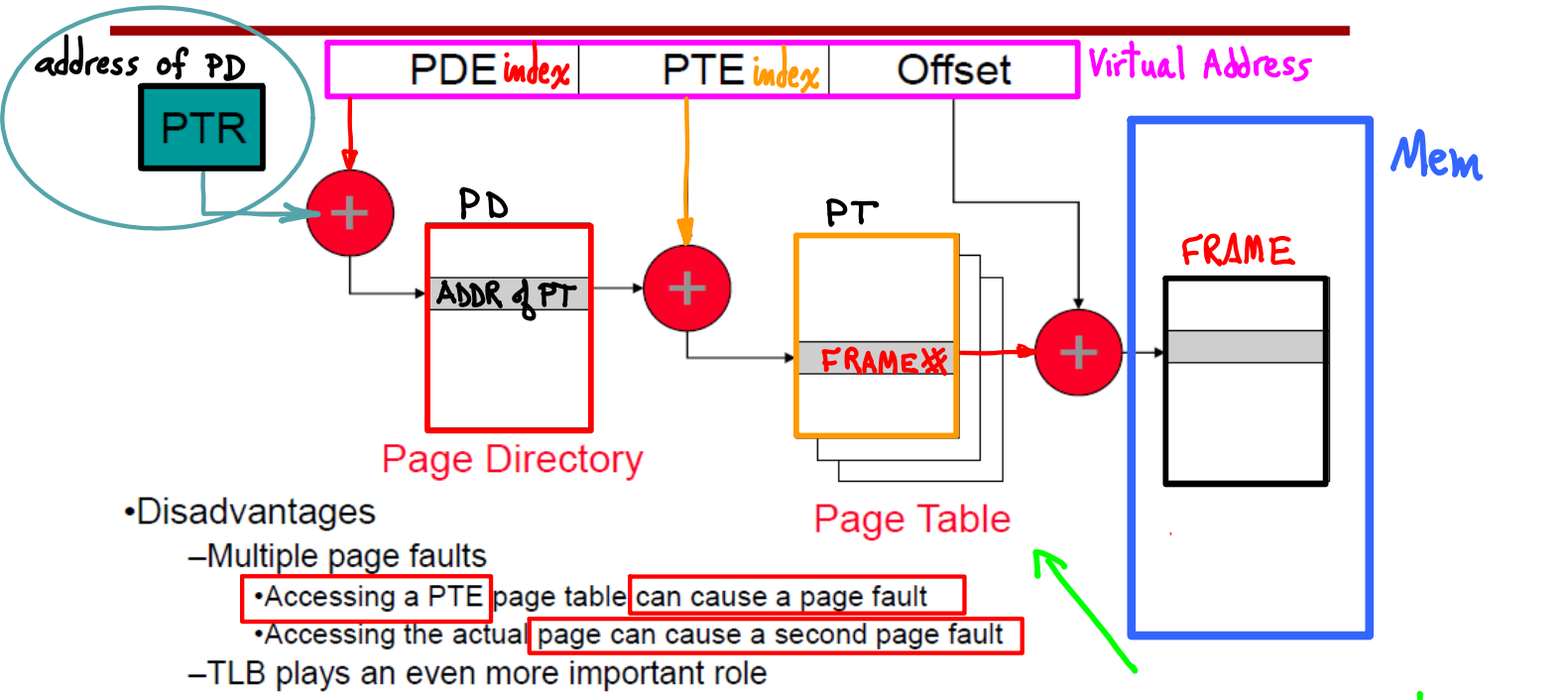
Mapped vs un-Mapped Pages



- Simplified Example
  - 16 page virtual address space
- Flags
  - P: Is entry in physical memory?
  - M: Has this part of VA space been mapped?

multi-level Page Tables

Is PT same size as before?
How do we save space with this?

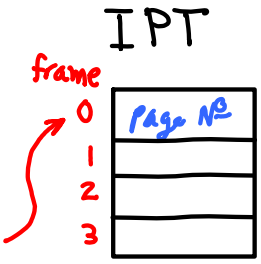


- Disadvantages
  - Multiple page faults
    - Accessing a PTE page table can cause a page fault
    - Accessing the actual page can cause a second page fault
  - TLB plays an even more important role

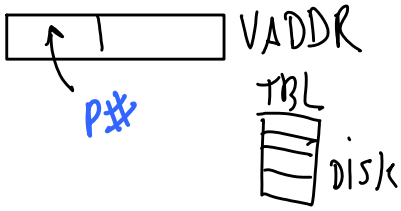
unmapped pages? → Don't allocate PT.

64-bit address space? 128-bit? Add more levels?

⇒ inverted page table

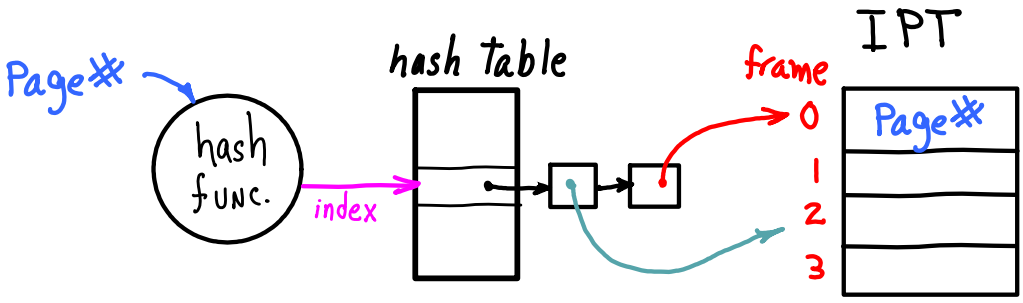


Inverted table shows which page is in the frame. 1 entry per frame.



Given a P#, how to find frame? Search table linearly?

Use chained hashing: entries contain pointers to inverted table entry.



Do collisions bother us? Typically, how many?

Is this scheme to slow?

Why? When does this happen?

How many instructions are involved anyway?

Where is the real Page Table anyway?

Do we need one?

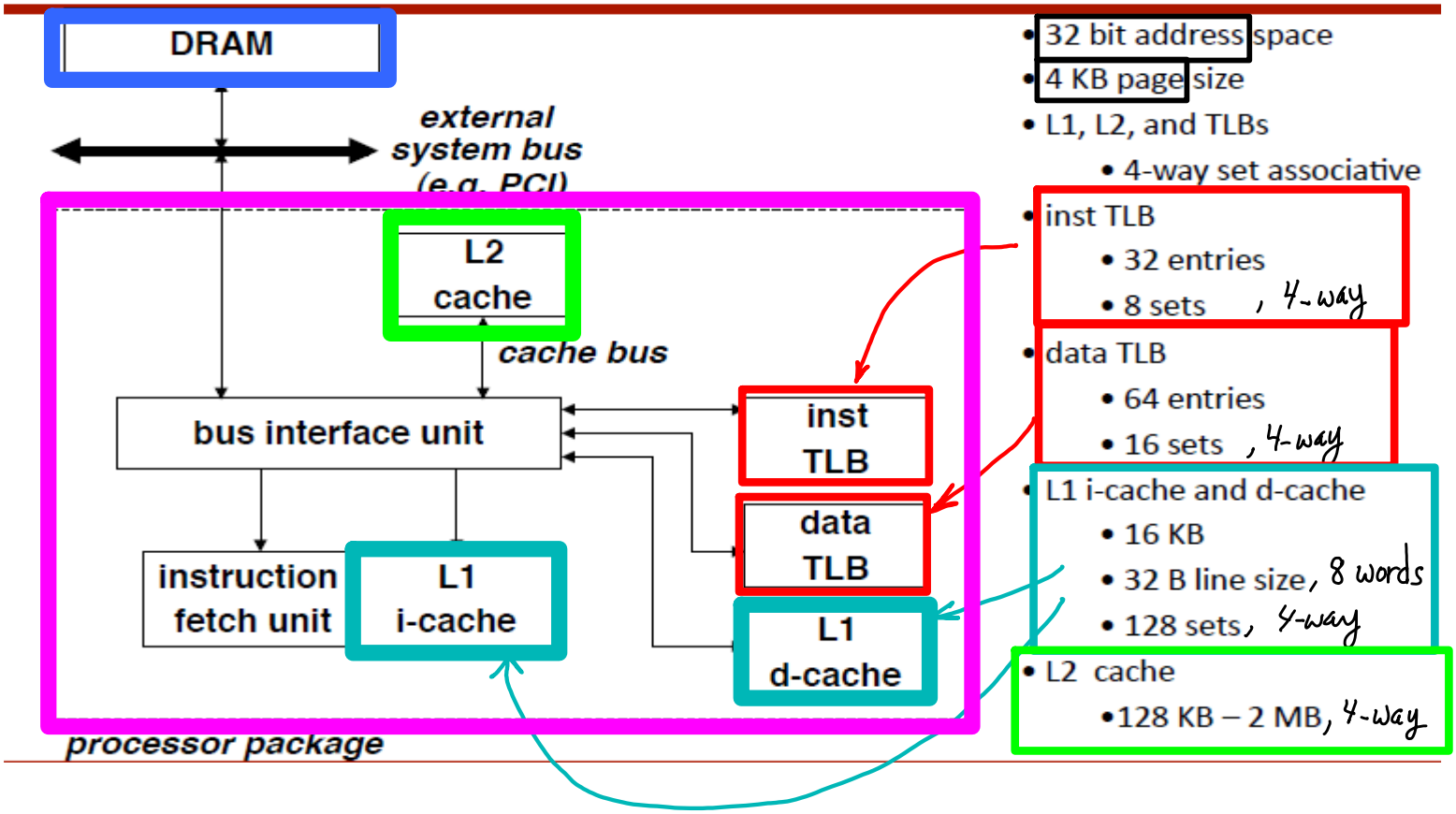
Real Example: Intel P6

- Internal Designation for Successor to Pentium
  - Which had internal designation P5
- Fundamentally Different from Pentium
  - Out-of-order, superscalar operation
  - Designed to handle server applications
    - Requires high performance memory system
- Resulting Processors
  - PentiumPro 200 MHz (1996)
  - Pentium II (1997)
    - Incorporated MMX instructions
    - L2 cache on same chip
  - Pentium III (1999)
    - Incorporated Streaming SIMD Extensions
  - Pentium M 1.6 GHz (2003)
    - Low power for mobile
  - The base for Intel Core and Core 2

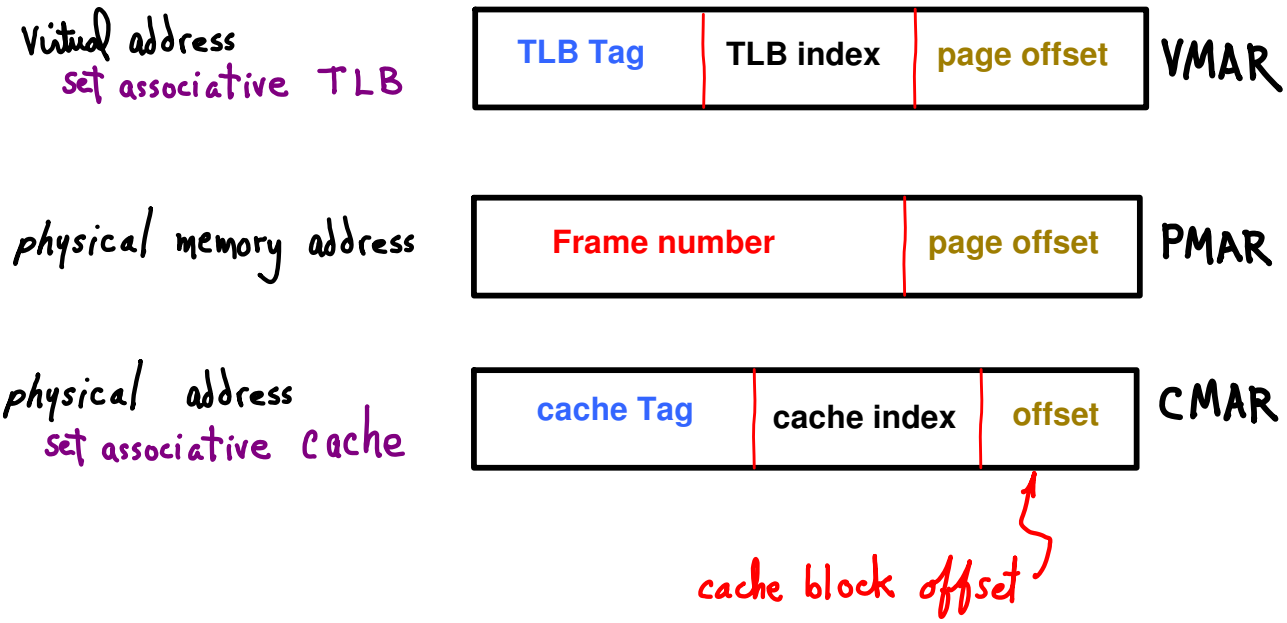
Adapted from Computer Systems: APP

Bryant and O'Halloraon

P6 memory system

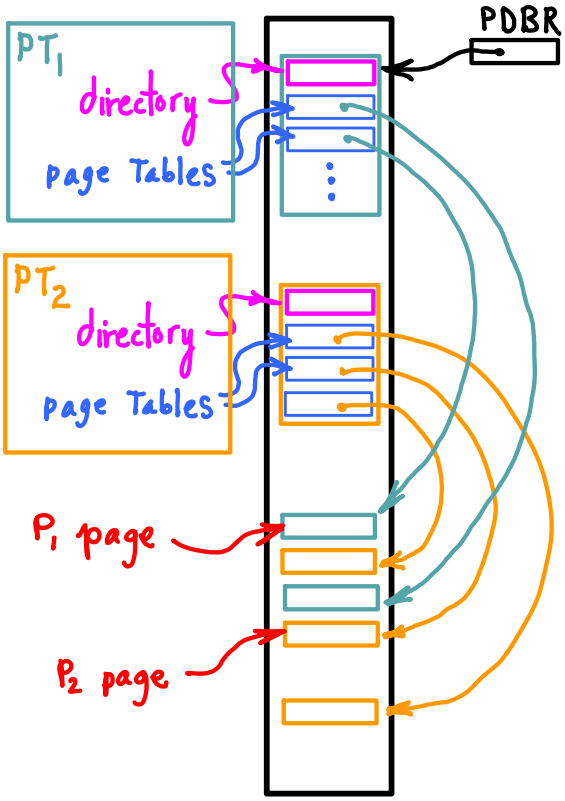


P6 address bit usage



P6 2-level Page Table,

Programs P1 and P2:



} hardware register, points to current directory

directory

} 1k 32-bit addresses, point to page tables

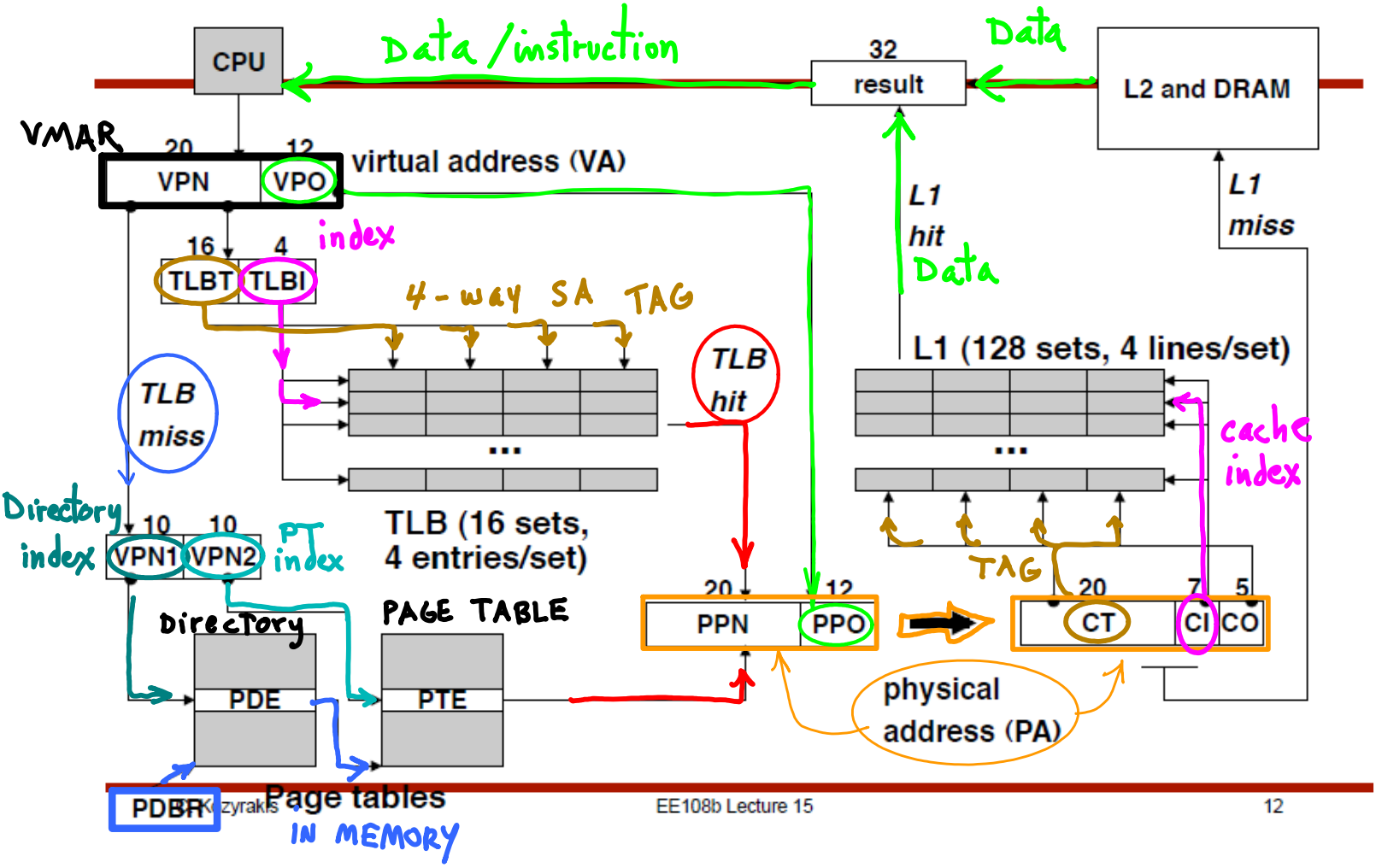
page table

} 1k 32-bit addresses, point to pages

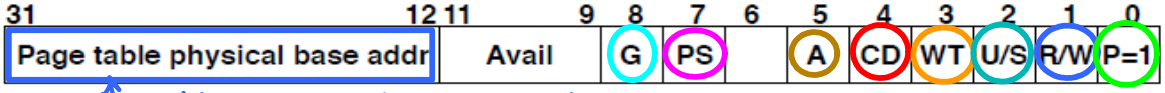
Page

} 1k 32-bit words, data/instructions/stack

Overview of P6 address translation



P6 page directory entry (PDE) one 32-bit word



if P=1

Right zero extended to 32 bits

Page table physical base address: 20 most significant bits of physical page table address (forces page tables to be 4KB aligned)

Avail: available for system programmers

G: global page (don't evict from TLB on task switch)

PS: page size 4K (0) or 4M (1)

A: accessed (set by MMU on reads and writes, cleared by software)

CD: cache disabled (1) or enabled (0)

WT: write-through or write-back cache policy for this page table

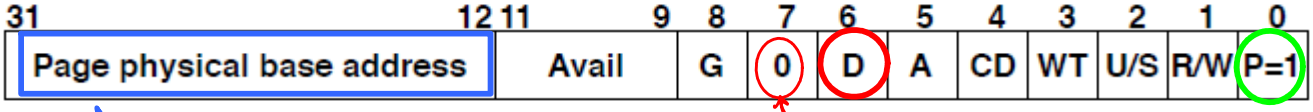
U/S: user or supervisor mode access

R/W: read-only or read-write access

P: page table is present in memory (1) or not (0)

if P=0

P6 page table entry (PTE) one 32-bit word



if P=1

Right zero extended to 32 bits

Page size not used?

Page base address: 20 most significant bits of physical page address (forces pages to be 4 KB aligned)

Avail: available for system programmers

G: global page (don't evict from TLB on task switch)

D: dirty (set by MMU on writes)

A: accessed (set by MMU on reads and writes)

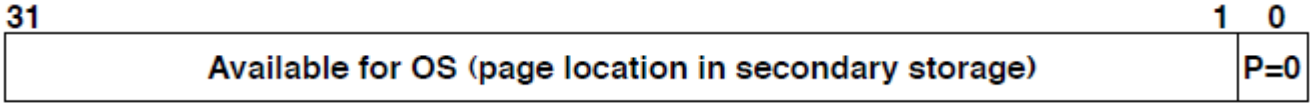
CD: cache disabled or enabled

WT: write-through or write-back cache policy for this page

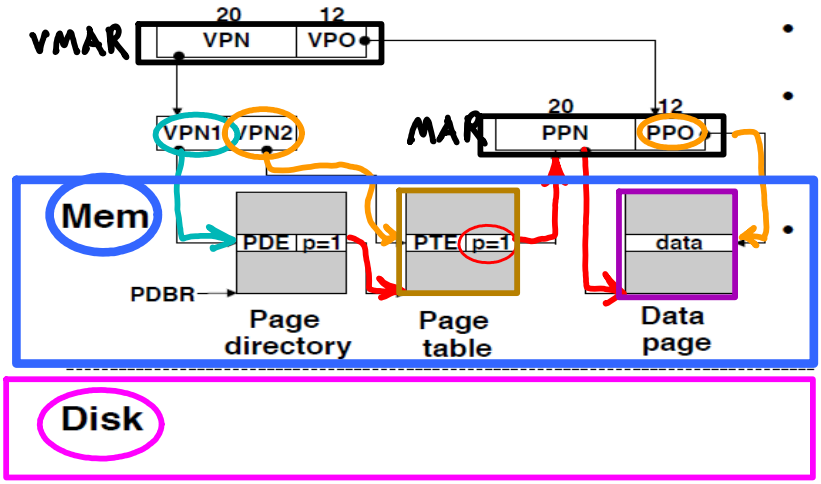
U/S: user/supervisor

R/W: read/write

P: page is present in physical memory (1) or not (0)



Possible Scenarios for Locations of Pages containing PD, PT, Data



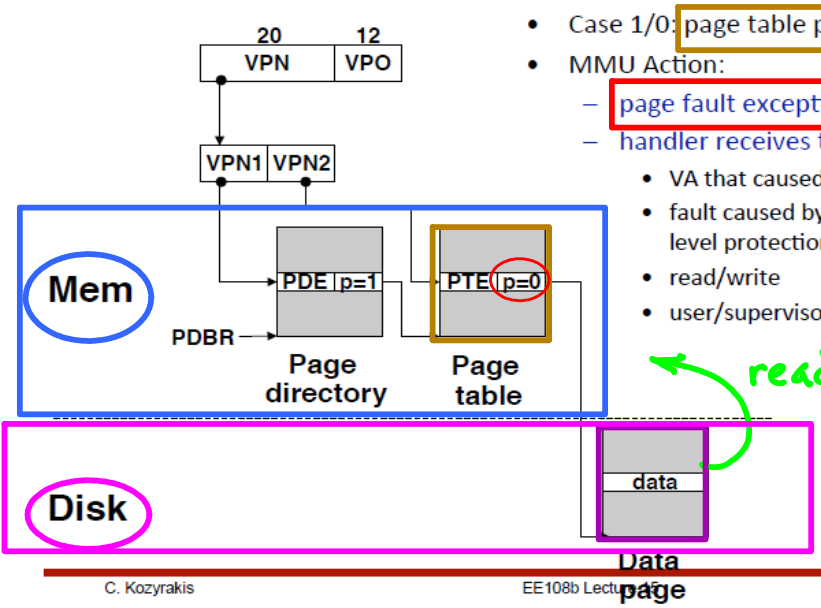
- Case 1/1: page table and page present.
- MMU Action:
  - MMU build physical address and fetch data word.
- OS action
  - none

Case 1/1

Page Table page in memory

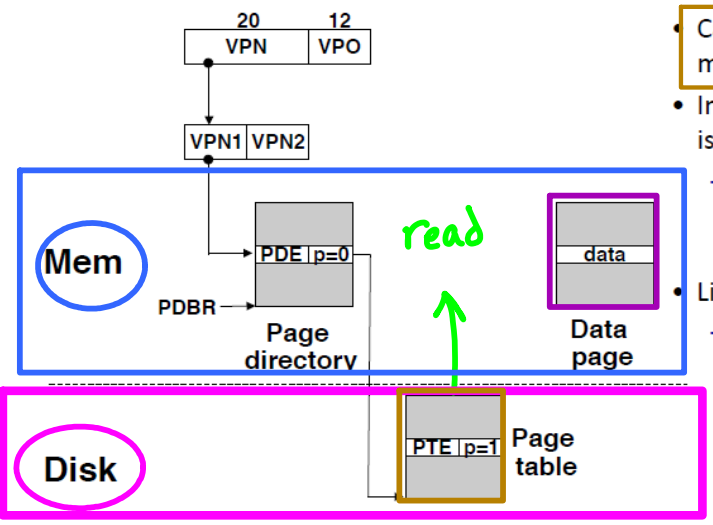
Data page in memory

Case 1/0



- Case 1/0: page table present but page missing.
- MMU Action:
  - page fault exception
  - handler receives the following args:
    - VA that caused fault
    - fault caused by non-present page or page-level protection violation
    - read/write
    - user/supervisor
- OS Action:
  - Check for a legal virtual address.
  - Read PTE through PDE.
  - Find free physical page (swapping out current page if necessary)
  - Read virtual page from disk and copy to physical page
  - Restart faulting instruction by returning from exception handler.

Case 0/1



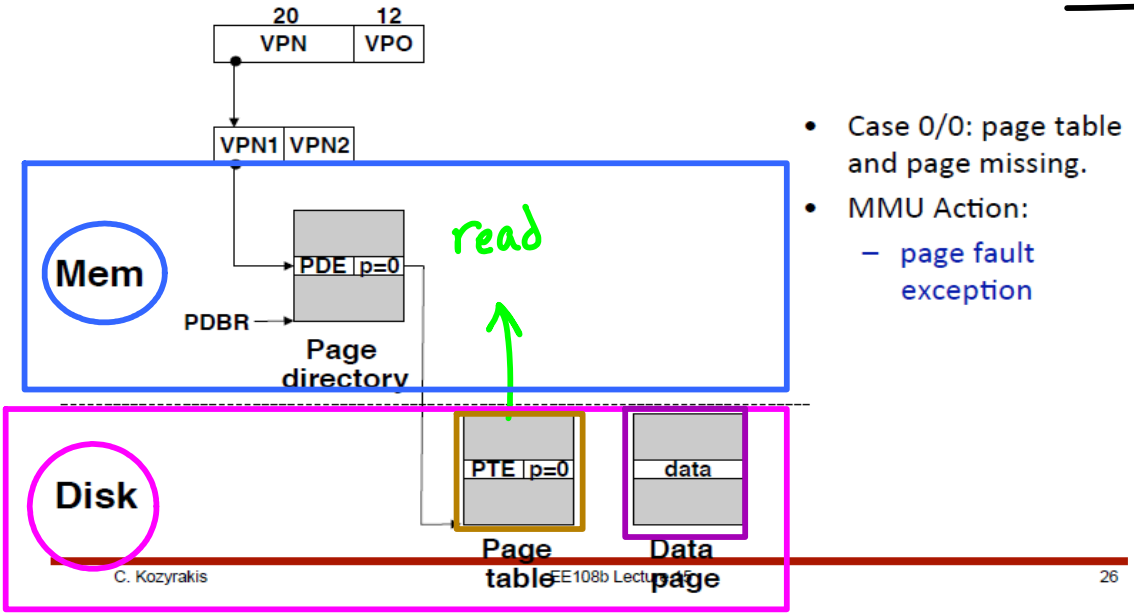
- Case 0/1: page table missing but [redacted]
- Introduces consistency issue.
  - potentially every page out requires update of disk page table.
- Linux disallows this
  - if a page table is swapped out, then swap out its data pages too.

OS Action:

- Check for a legal virtual address.
- Read PTE through PDE.
- Find free physical page (swapping out current page if necessary)
- Read virtual page from disk and copy to physical page
- Restart faulting instruction by returning from exception handler.

Read PDE, find PT disk address; Read PT page from disk; Restart; (after restart: Case 1/1)

Case 0/0

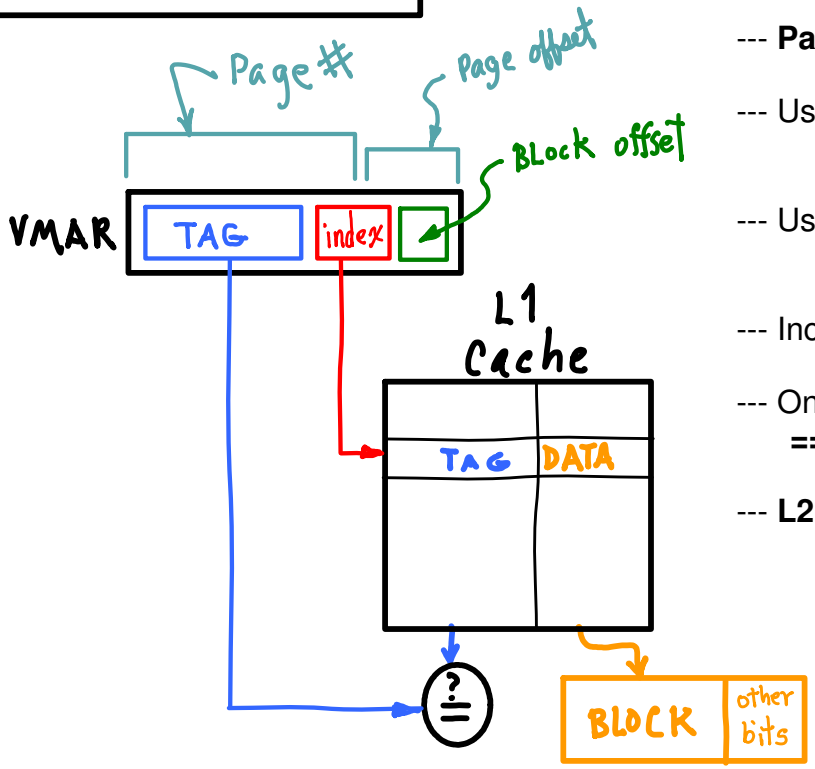


- Case 0/0: page table and page missing.
- MMU Action:
  - page fault exception

- OS action:
  - swap in page table.
  - restart faulting instruction by returning from handler.
- Like case 0/1 from here on.

Page fault for PT as in case 0/1; Restart; (after restart, becomes Case 1/0)

Virtual Caches



E.G. Simple DM cache **(virtually tagged and indexed)**

--- **Page-offset** is **untranslated**, **PAGE#** is **translated**

--- Use **part of virtual address** as **tag**
(Page No. + or - some bits)

--- Use **some translated VM bits** for **index**
(remainder is block offset)

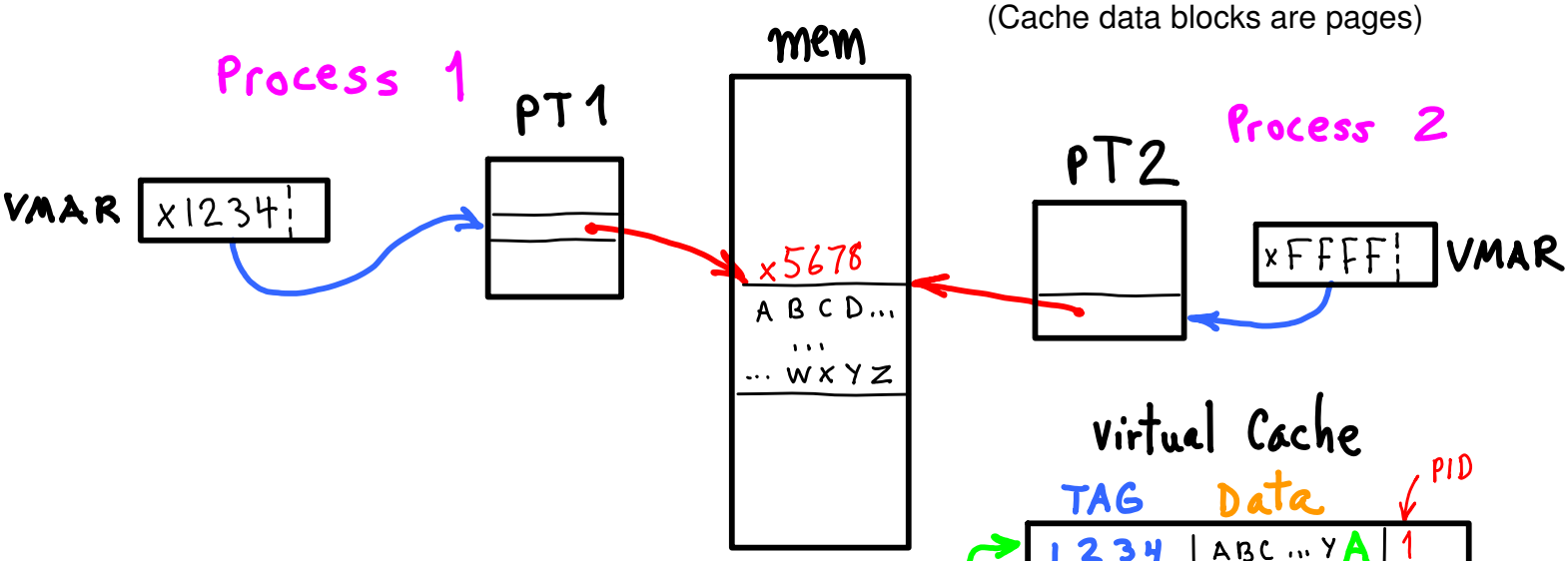
--- Include **PID**, Accessed and Dirty bits, etc., in cache

--- Only **translate on misses**
===> **cuts TLB from critical path**

--- **L2 is a physically** indexed and tagged cache

Aliases, Synonyms

Shared page
PT1: **Mapped from V-Page x1234**
PT2: **Mapped from V-Page xFFFF**
Both Map to frame x5678
(Cache data blocks are pages)

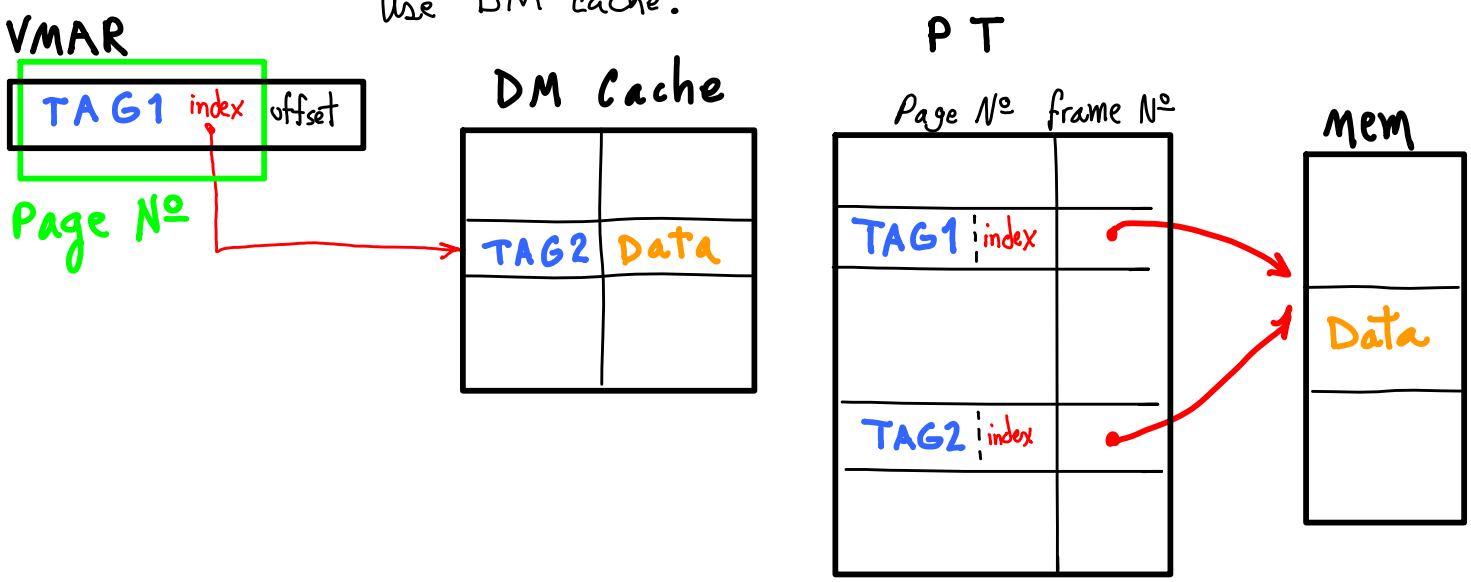


(Could also be multiple mappings in same page table.)

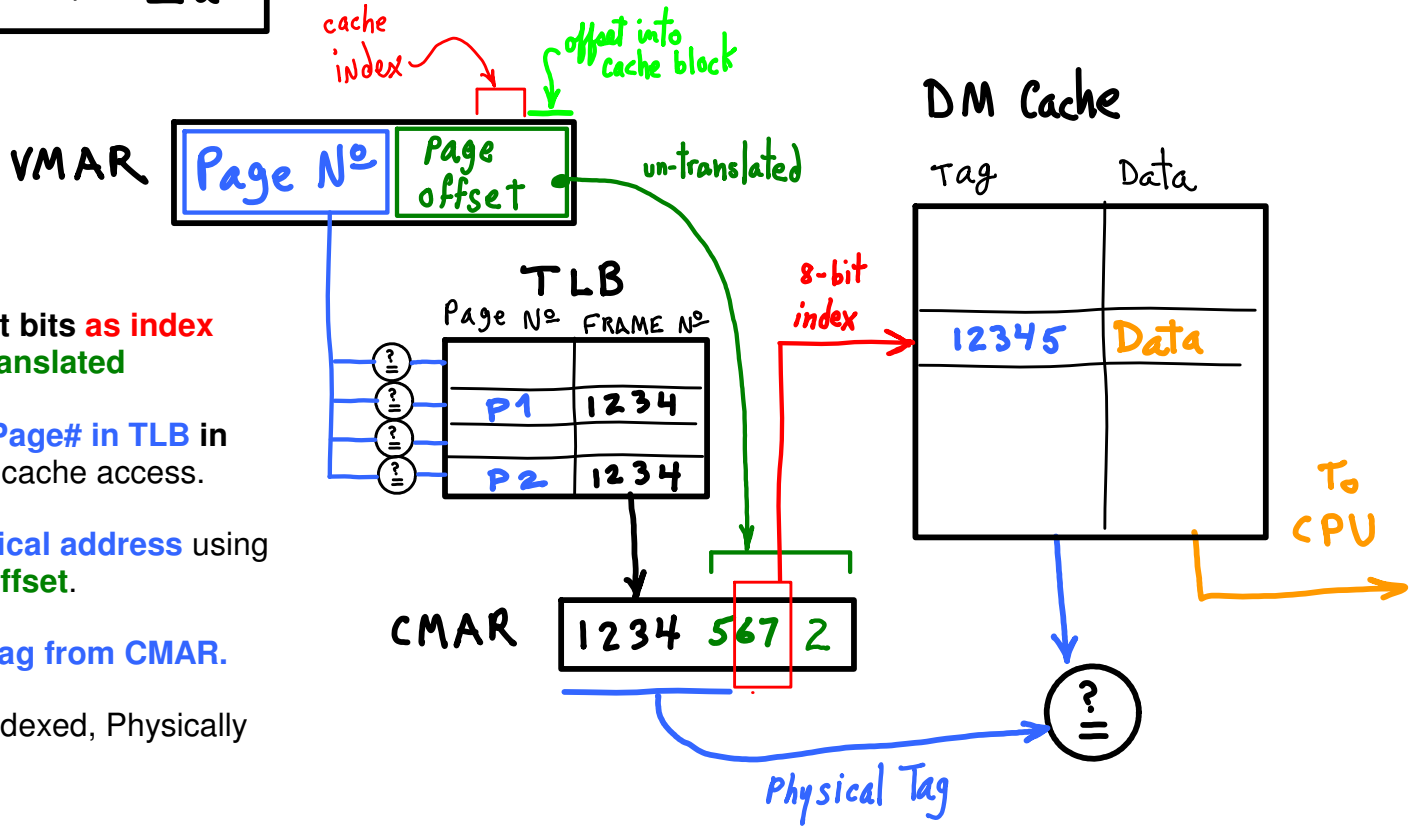
Goal: enforce an invariant so that,
--- Shared data is present once, at most.

Solution 1

force collision : Software insures all shared pages have same index. Use DM cache.



Solution 2a



page offset bits as index ===> un-translated

Compare Page# in TLB in parallel w/ cache access.

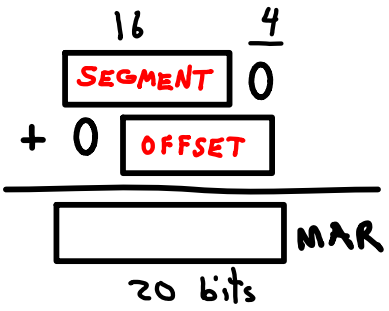
Form **physical address** using **frame# + offset**.

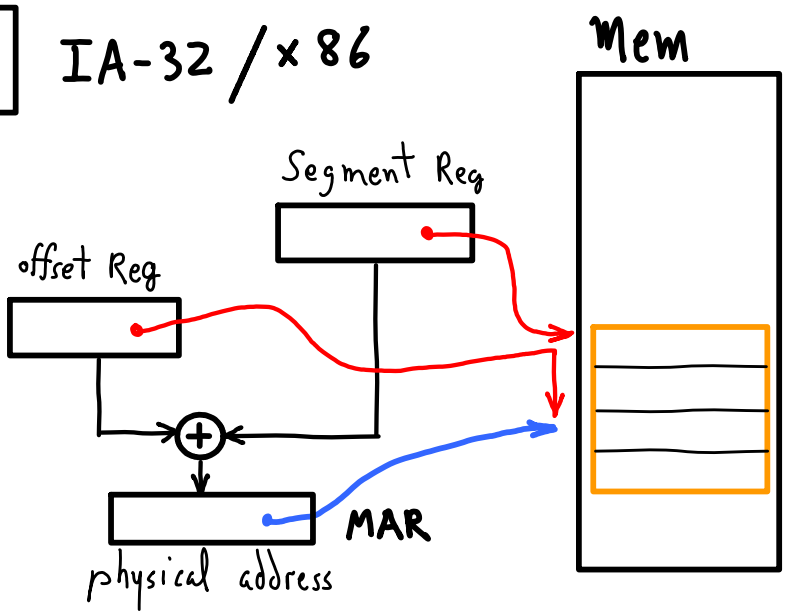
Compare tag from CMAR.

"Virtually Indexed, Physically Tagged"

Segmented Memory

IA-32 / x86





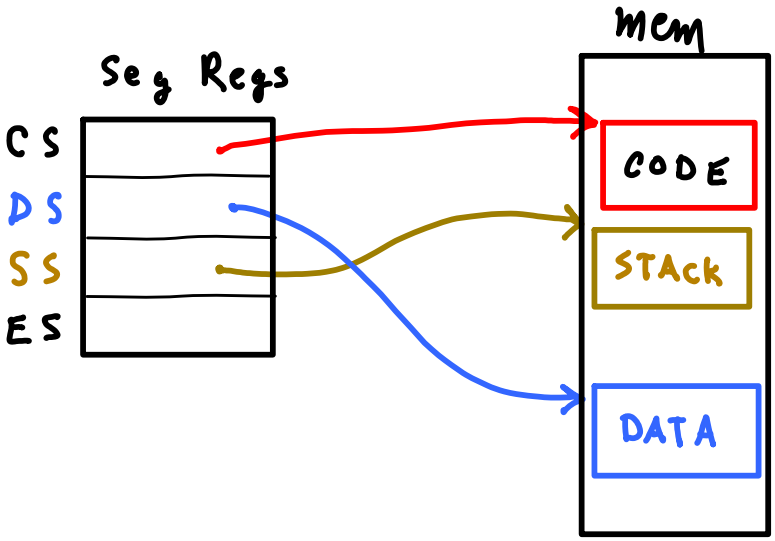
Originally

No limit checking
---- can overrun segment

No protection
----- can write segment registers

Segment registers implicit
----- instruction fetch: uses CS
----- data access: uses DS
----- stack operation: uses SS

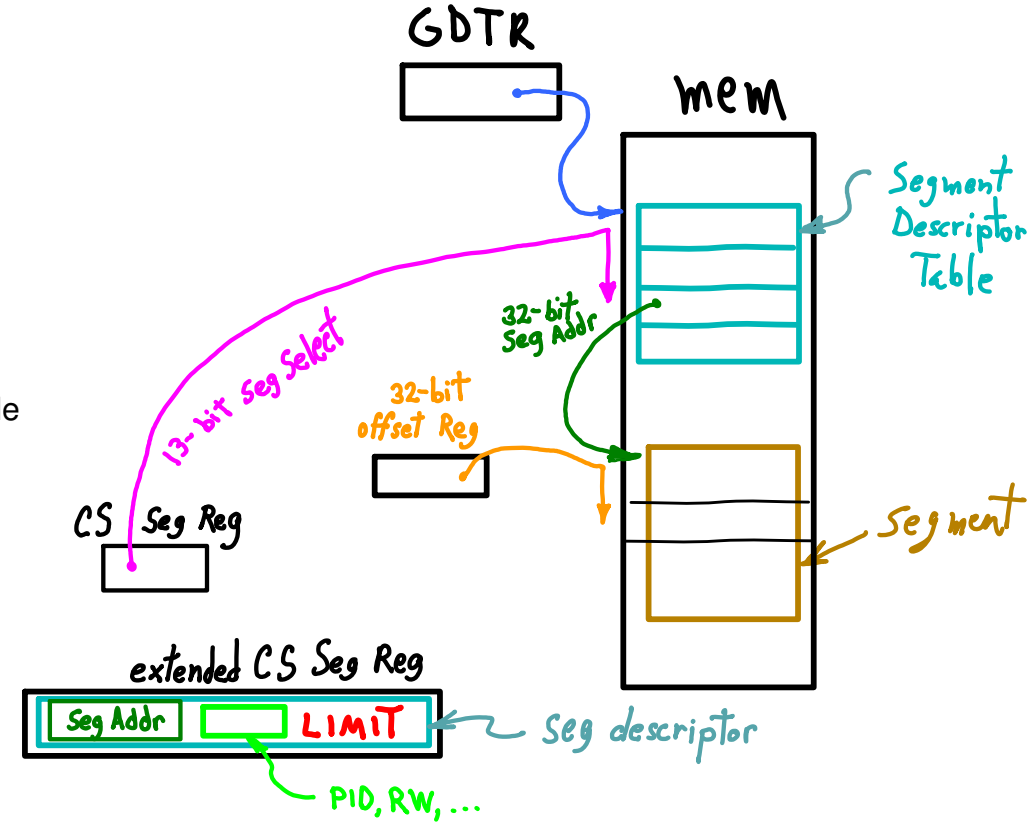
Programmer's perspective:
---- Segment's address is 0
---- Offset is address



Next Level

Seg Reg is offset into table
---- Entries are descriptors
Too Slow? Fix it:
---- extend Seg Regs
---- cache Descriptor
in extended Seg Reg
---- Check limits, PID, R/W, ...
Also:
---- Special Segs for Calls
---- "conforming" ==> change mode
---- 8k segments @ 4GB

Flat Addressing:
---- set all Descriptors:
---- BASE == x000000000
---- LIMIT == xFFFFFFFF
---- 1-to-1 w/ 32-bit **MAR**



---- Seg Selects can be written

writing CS, DS, SS: changes segments
But, via selecting different segment descriptors in table

Descriptor table is OS controlled.

---- Also available in IA-32 (x86)

Paging mode (2-level and 3-level)

"Real" mode (all physical 20-bit address w/ 16-bit segment + 16-bit offsets)

Paged Segments paging + segmentation:

Segment Descriptor points to Page Directory

Reference:
http://pdos.csail.mit.edu/6.828/2005/readings/i386/s05_01.htm

Virtual Machines

Approach I

General TM simulation

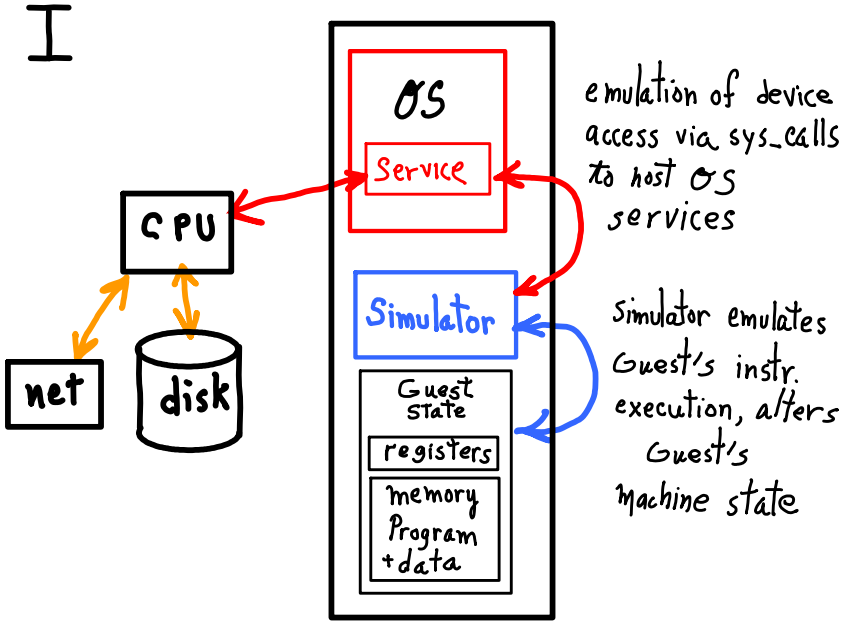
---- **Simulated machine is arbitrary**
Just write a program to model virtual machine's hardware/ISA

--- **Virtual Machine** (VMa or Guest) defined by **simulator program**.

--- **VMa's resources** are **simulator's data structures**.

--- VMa **interacts with external devices** through **simulator's actions**.

--- **Host OS provides**
--- **services: I/0, cpu time, ...**
--- isolation, **protection**
--- **simulator is a user process**



Just an ordinary Process

Can Host OS isolate/protect?
- Host OS complexity, bugs

Is simulation fast enough?
- Each simulated instruction requires many host instructions.

Approach II

minimal "OS"

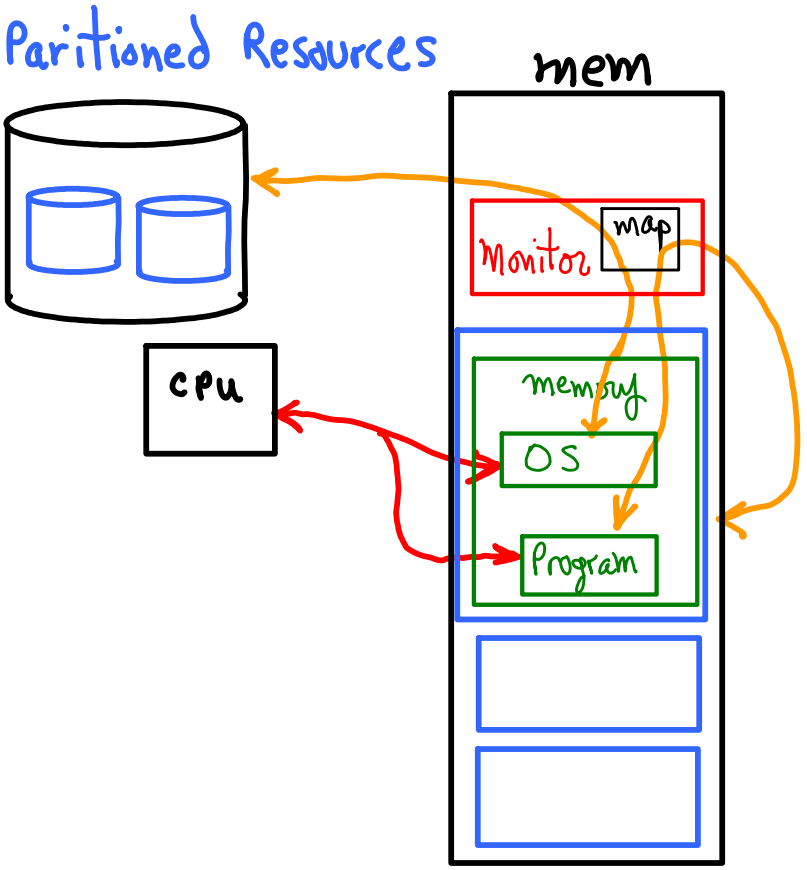
--- **Monitor provides Mapping Virtual Machine (VMa) ===> partitioned resources**

--- **Monitor is small, simple, reliable**

--- **Each guest** runs in **its own VMa**

--- **Guest instructions run w/o emulation**
Guest code is in HW ISA

--- **Each VMa** has **its own OS**
manages resources separately
--- processes
--- memory
--- disk space
--- cpu scheduling



Advantages

--- Monitor-1, Monitor-2
identical virtual machines
Host HW can be different (degree?)

--- Guests Isolated
VMM is safer than OS
Bugs/Attacks limited to one VMa

--- Guest migration, Multiple guests
===> bulk efficiencies:
shared computing resources
uptime
load balancing

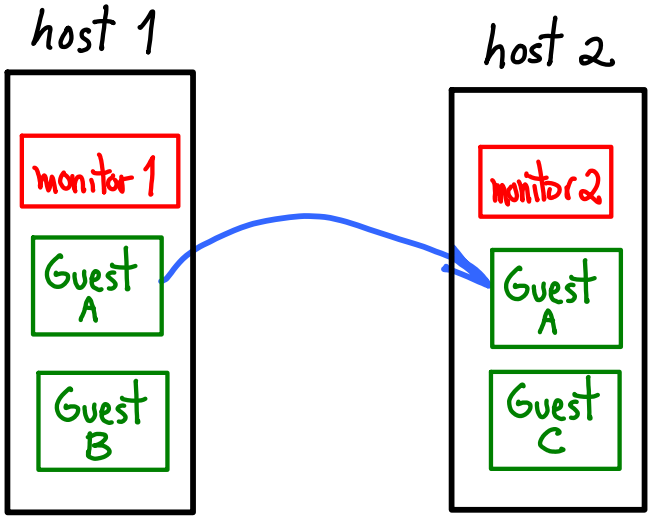
--- Legacy apps ===> Legacy VMa.

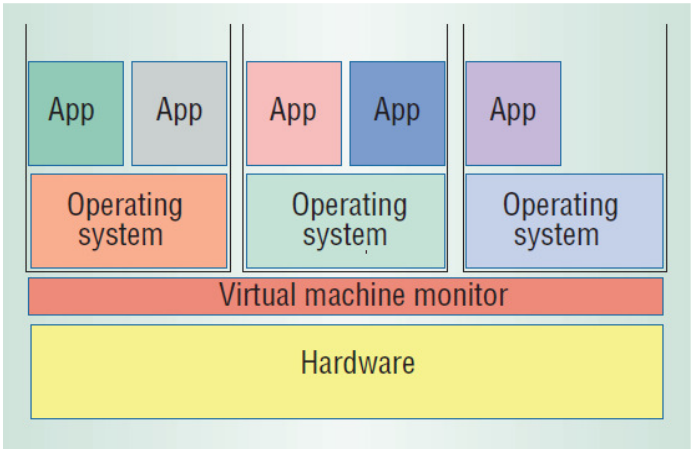
--- Guest OS configuration
matches guest's apps

--- Different OS per Guest

--- Checkpointing
Suspend/restart/rollback

HW Platform 1 ≠ HW Platform 2





--- Virtualizable if:
1. Can execute directly on HW
2. VMM controls resources

--- Monitor runs in kernel mode

--- Guest runs in user mode

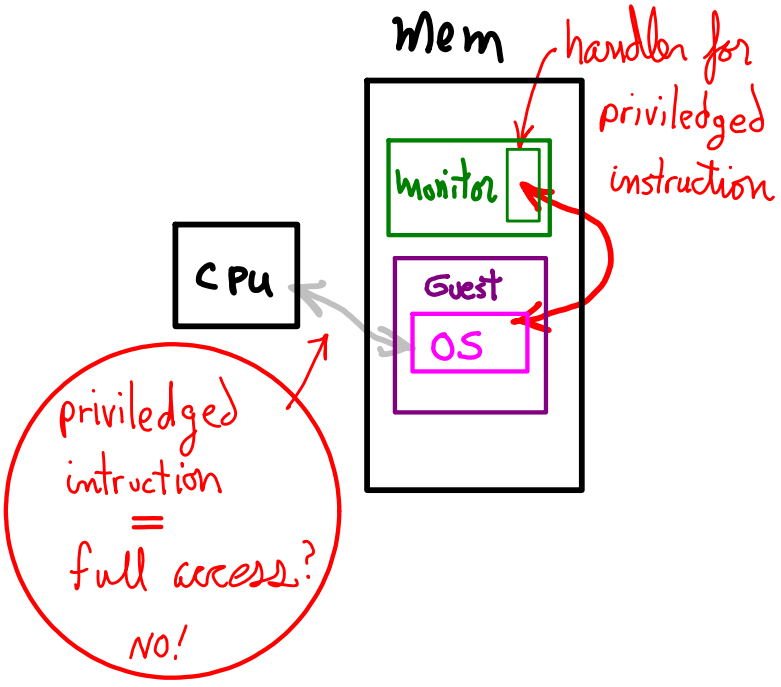
===> privileged instructions
trap to monitor's handler

OR

---- Binary translation (static or runtime):
Replace problematic instructions

OR

---- Add new hardware modes.

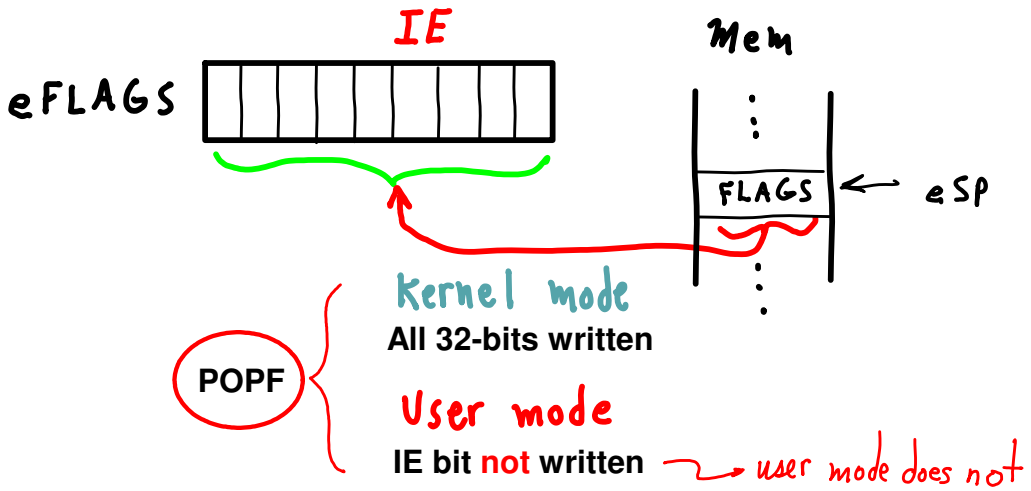


What problem instructions, can't we trap all of them?

if so → x86 virtualization!

x86 eFLAGS register, like LC3's PSR:

--- N, Z, P
--- Overflow
--- Carry
...
--- Interrupt Enable
...

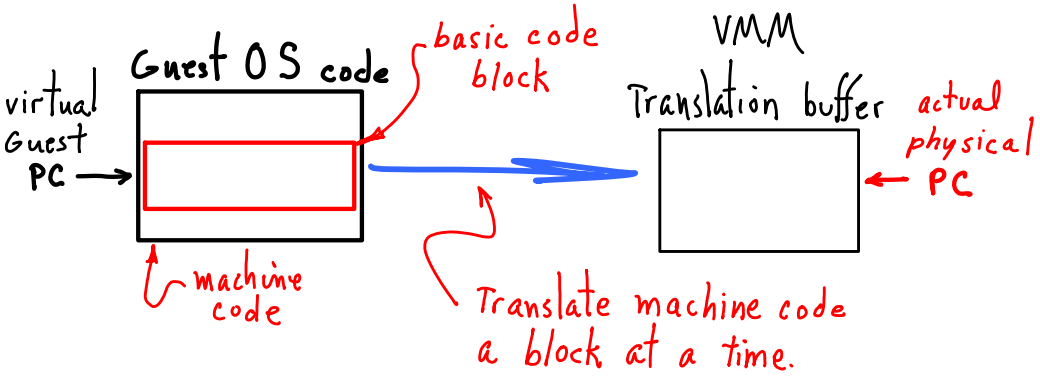


Guest OS is running USER mode!

→ Does not cause a trap
→ Does not work correctly

Dynamic Translation

check for problematic instructions, replace them



VM Ware METHODS

vmkernel:
--- boot loader
--- x86 abstraction
--- IO stacks (storage, network)
--- memory scheduler
--- cpu scheduler

VMM (vmkernel priviledged process):
--- Trapping, translation
--- one per VM

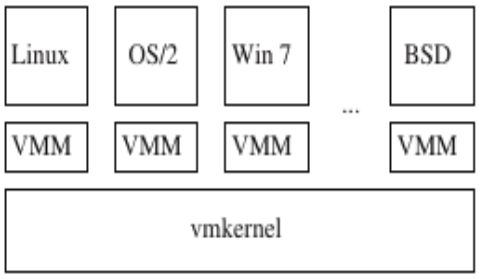


Figure 1: The ESX hypervisor: one vmkernel per host, and one VMM per virtual machine.

The Evolution of an x86 Virtual Machine Monitor

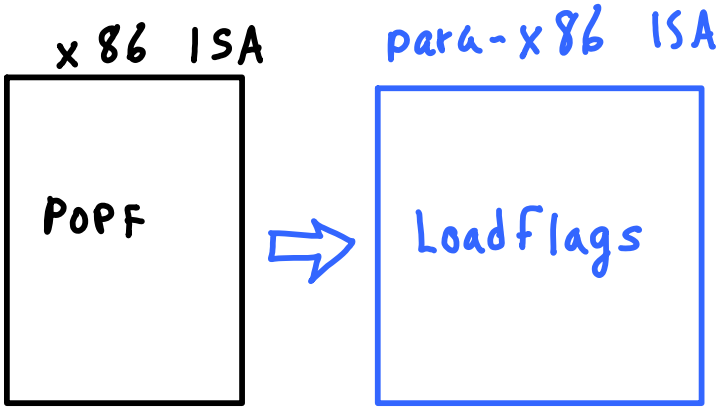
Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, Pratap Subrahmanyam
{agesen, alexg, jeffshel, pratap}@vmware.com

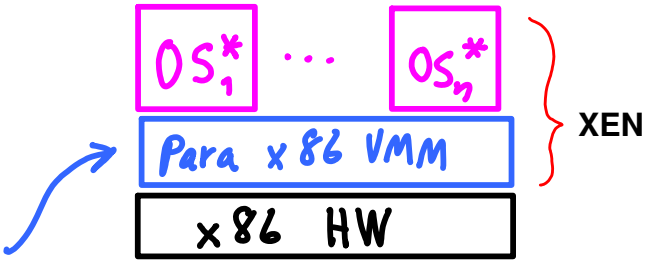
from VMWare – Server Version

For example, VMware's vSphere ESX hypervisor is comprised of the *vmkernel* and a VMM. The vmkernel contains a boot loader, an x86 hardware abstraction layer, I/O stacks for storage and networking, as well as memory and CPU schedulers. To run a VM, the vmkernel loads the VMM, which encapsulates the details of virtualizing the x86 architecture, including all 16 and 32 bit legacy modes as well as 64 bit long mode. The VM executes directly on top of the VMM, touching the hypervisor only through the VMM surface area.

Para-Virtualization

Change the x86 ISA !





New ISA ===> Rewrite the OS*

---- more instructions run w/ VMM
===> faster, less trapping
===> no dynamic translation

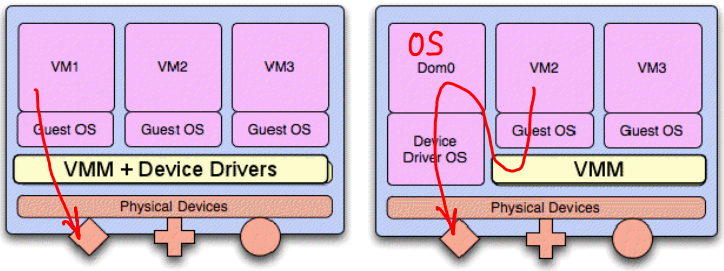
But

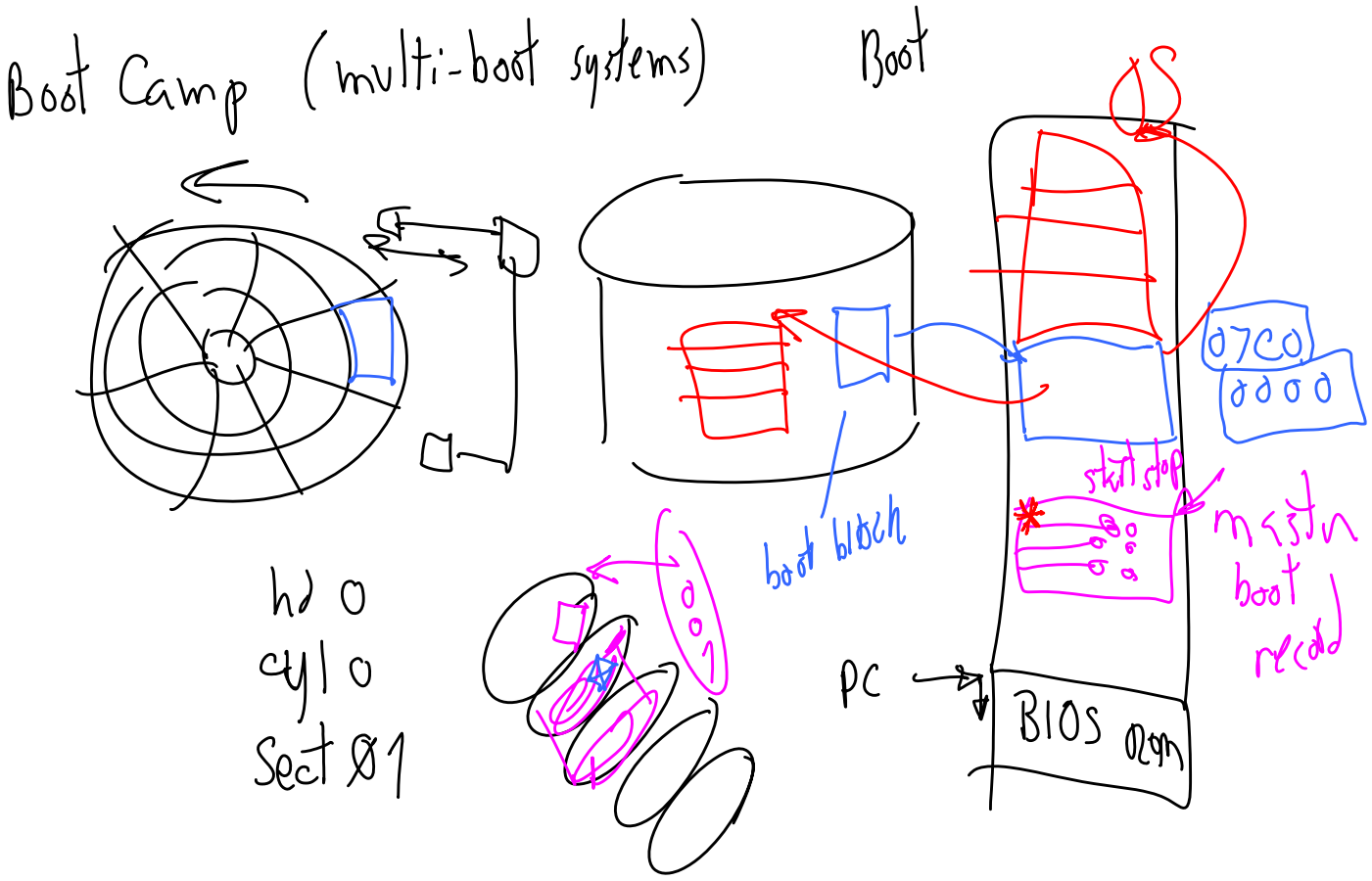
--- can't run original OS binaries
===> keeping up w/ the Joneses?

	Type I	Type II
Fully-virtualized	VMware ESX	VMware Workstation
Para-virtualized	Xen	User Mode Linux

I/O Virtualization

- Issue: lots of I/O devices
- Problem: Writing device drivers for all I/O device in the VMM layer is not a feasible option
- Insight: Device driver already written for popular Operating Systems
- Solution: Present *virtual* I/O devices to *guest* VMs and channel I/O requests to a trusted *host* VM running popular OS





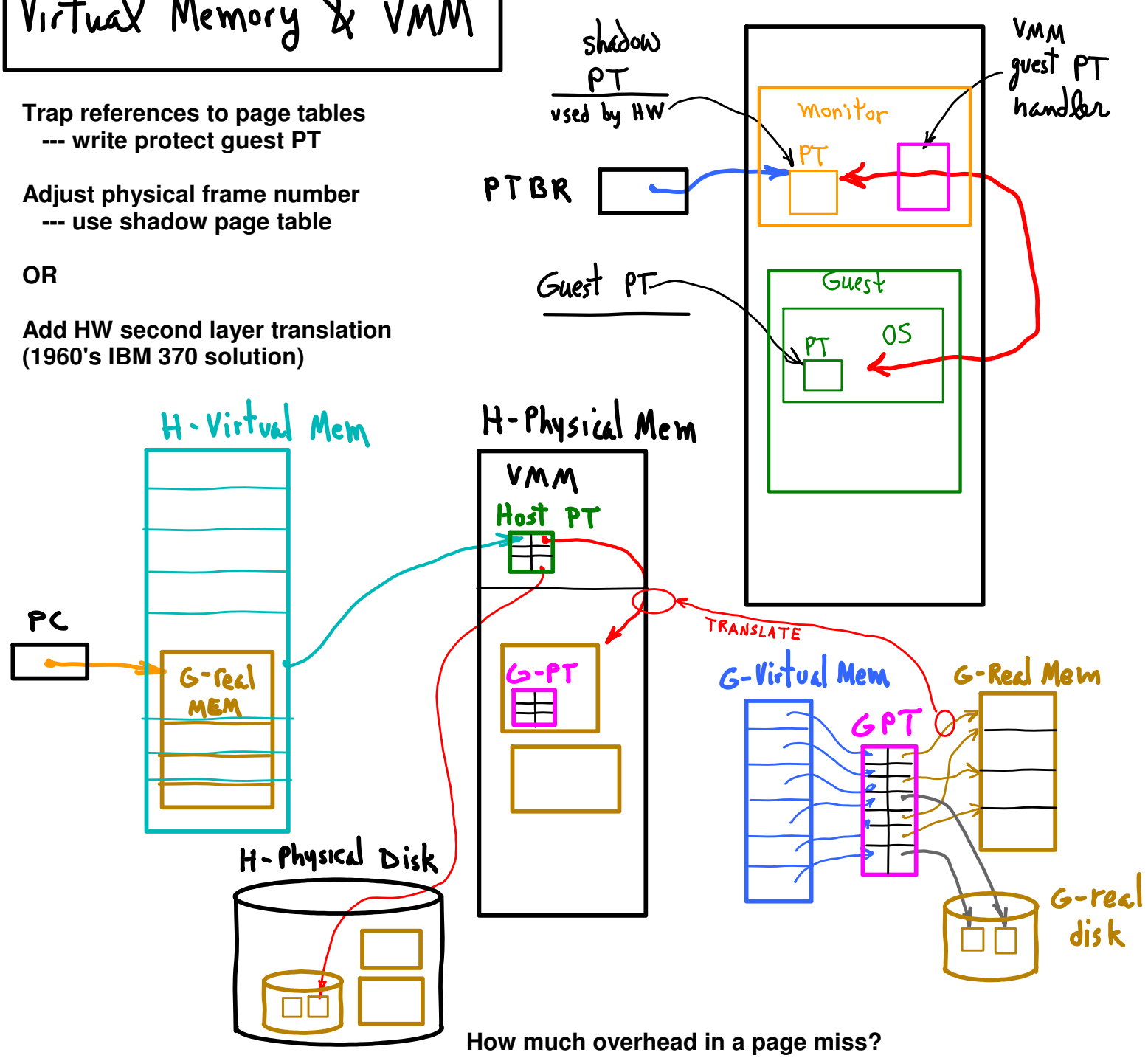
Virtual Memory & VMM

Trap references to page tables
--- write protect guest PT

Adjust physical frame number
--- use shadow page table

OR

Add HW second layer translation
(1960's IBM 370 solution)

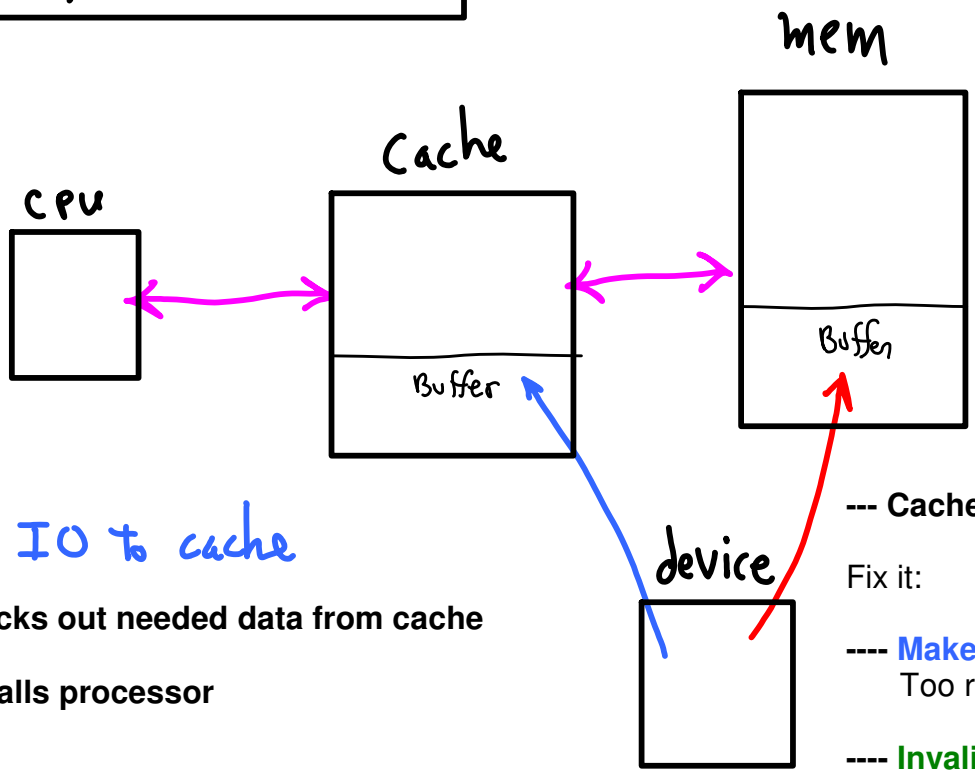


How much overhead in a page miss?

G-OS: page fault, context switches, 100s of cycles
VMM: examine G-PT (find G-PA), 100s of cycles
VMM: find H-Phys-Addr, 100s of cycles
VMM: allocate/fill shadow PT 100s of cycles

we must examine what happens when the guest accesses a particular gVA. First, the memory access causes a page fault (several hundred cycles in the circa 2002 processors). Then, the VMM walks the guest's page tables in software to determine the gPA backing that gVA (again costing a few hundred cycles). Next, the VMM determines the hPA that backs that gPA. Often, this step is fast, but upon first touch it requires the host OS to allocate a backing page. Finally, the VMM allocates a shadow page table for the mapping and wires it into the shadow page table tree. The page fault and the subsequent shadow page table update are analogous to a normal TLB fill in that they are invisible to the guest,

I/O + caches



1. IO to cache

--- kicks out needed data from cache

--- stalls processor

2. IO to mem

--- Cache data may be stale

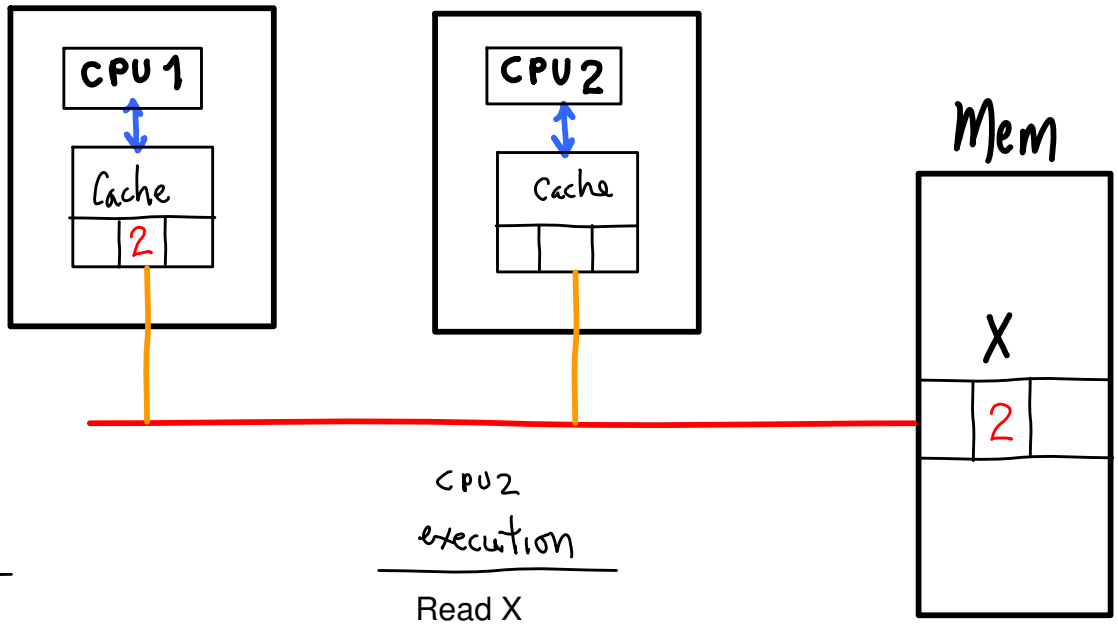
Fix it:

---- **Make buffer pages non-cacheable**
Too restrictive?

---- **Invalidate cache blocks** belonging to buffer just before IO?
Lock pages in memory temporarily.

Cache Coherency

multi-core, multi-processor, distributed



CPU1 execution

Read X
X <== X + 5
Write X

CPU2 execution

Read X
X <== X + 3
Write X

CPU-1

cache <== 2
cache <== 7
X <== 7

CPU-2

cache <== 7
cache <== 10
X <== 10

This OK.
Reverse is OK.
Interleaved?
what's supposed to happen?