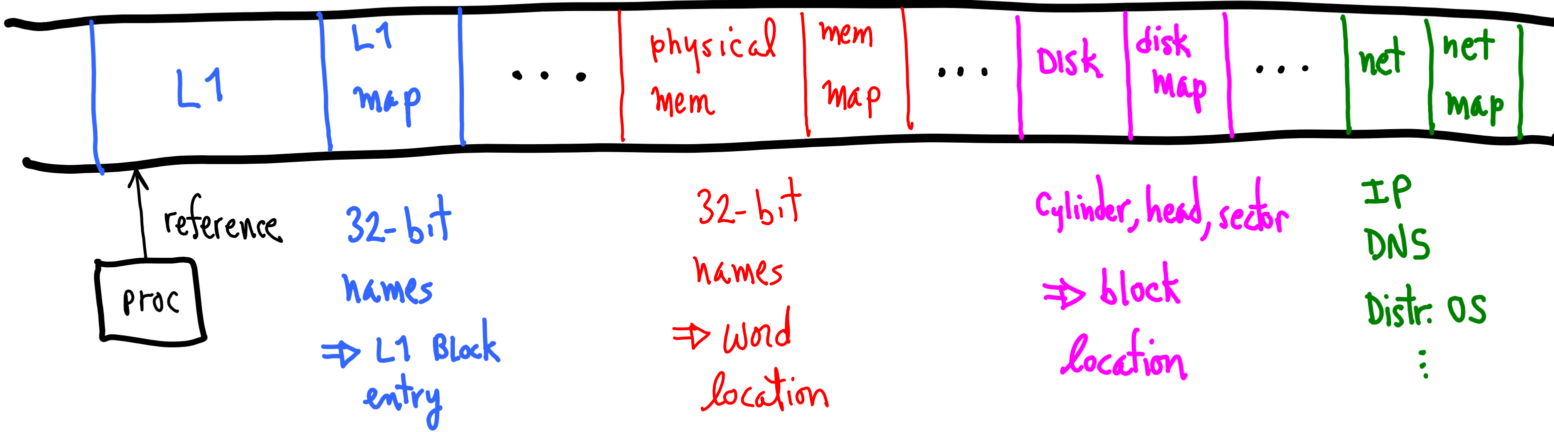
Finding things by name, generally

Fast access
===> use map to find object

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

Extend range
===> longer, hierarchical names

It's all TM Tape (move L, move R)



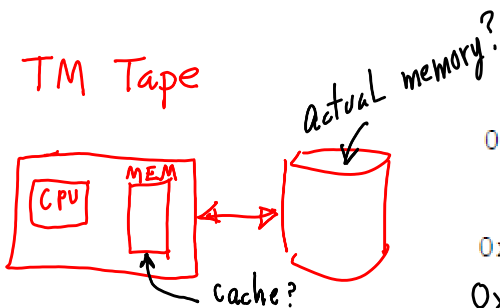
How is map embodied:
--- L1?
--- Memory?

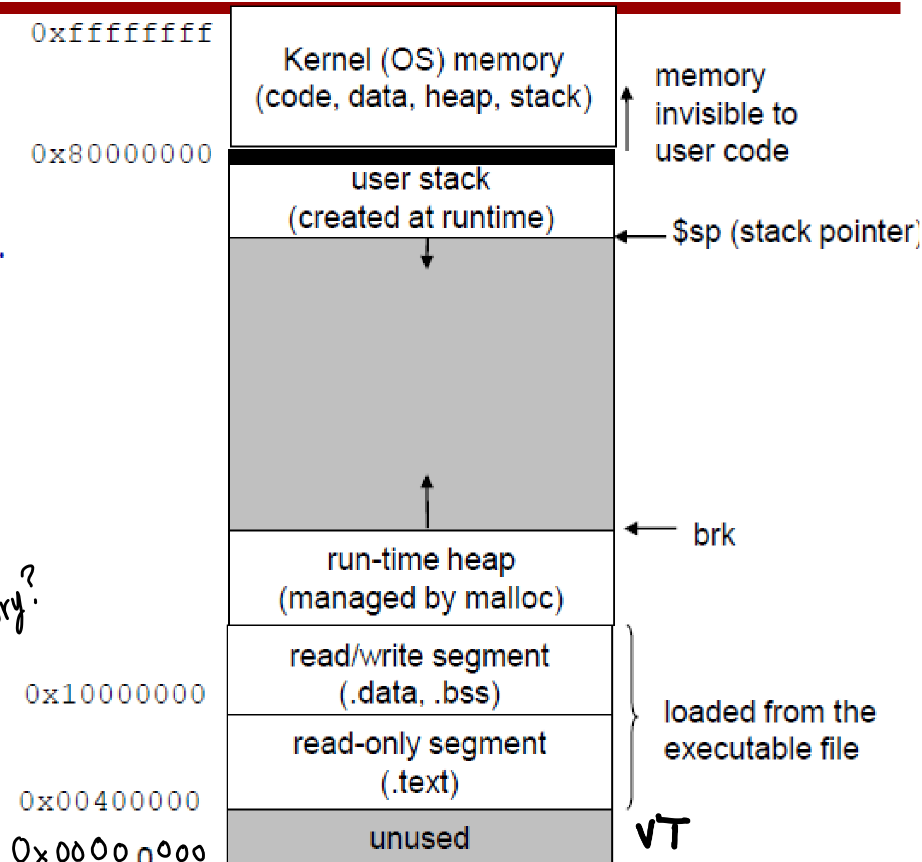
Virtual Memory

Motivation #1: Large Address Space for Each Executing Program

- 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

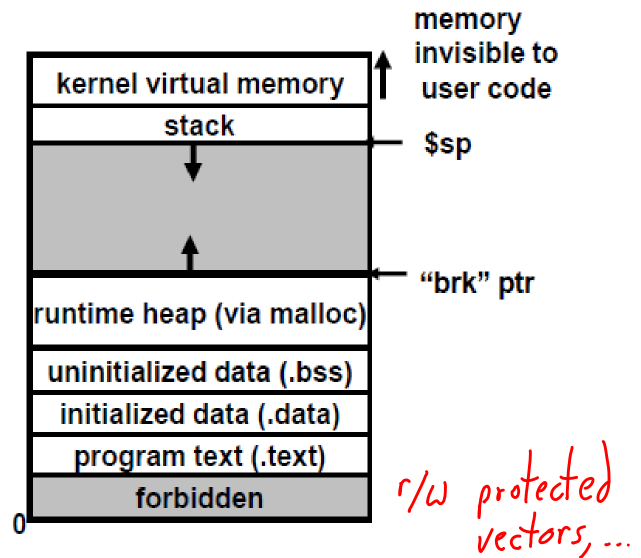
MEM = TM Tape





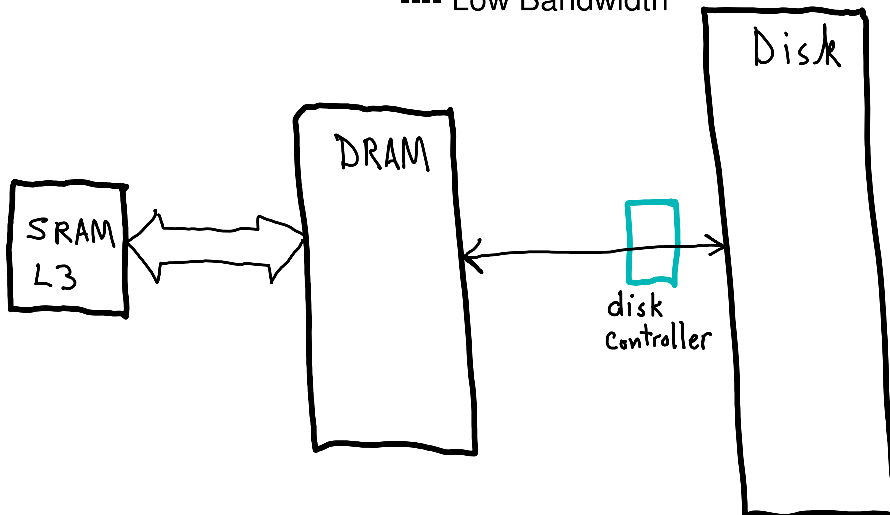
Motivation #2: Memory Management for Multiple Programs

- 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



The Environment

---- Long Latency
---- Low Bandwidth



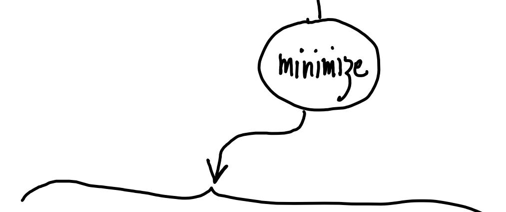
Latency 500 cycles
Bandwidth 10 GB/s

Latency 10^6 cycles
Bandwidth 300 MB/s

Performance

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

huge



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

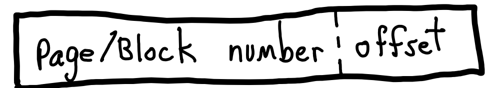
- New terms
  - VM block is called a "page"
    - The unit of data moving between disk and DRAM
    - It is larger than a cache block (e.g., 4KB or 16KB)
    - Virtual and physical address spaces are divided into virtual pages and physical pages (e.g., contiguous chunks of 4KB)
  - VM miss is called a "page fault"
    - More on this later

Just like cache blocks
But, much bigger offset

64 B (16 32-bit words)

6-bit

MAR

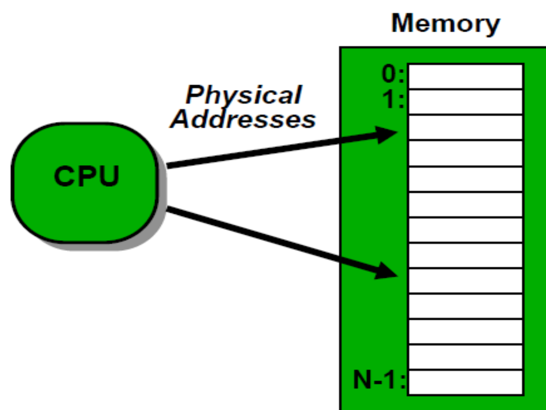


12-bit

4kB (1k 32-bit words)

A System with Physical Memory Only

- Examples:
  - most Cray machines, early PCs, nearly all embedded systems, etc.



In a general setting (TM), it's all tape (move L,R): more time for more remote accesses. Generally, how do we "address" something? What are "names"?

Addresses generated by the CPU point directly to bytes in physical memory

A System with Virtual Memory

(Physical) ⇒ A Fully Assoc. Cache for Disk Blocks

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

MAR ≡ Virtual Addresses

Page Table

Physical Addresses

Memory

0:
1:
P-1:

0:
1:
N-1:

CPU

look up by tag

disk addr

Disk

physical memory

MAR

Tag	block offset

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

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

FA Cache

TAG	data

separate into two parts ⇒

Cache Tags — TAG — pointer to data → Cache Data — WORD } Cache Block

Mem — Block — WORD

MAR

Block Address	Word offset	B

(Byte offset)

⇑ In cache

⇐ not in cache

Expand tag storage:
entry for every possible tag

tag == block address, aka block number, is redundant, eliminate

Block ==> Page
Tag storage ==> Page Table
Cache Data ==> Memory pages

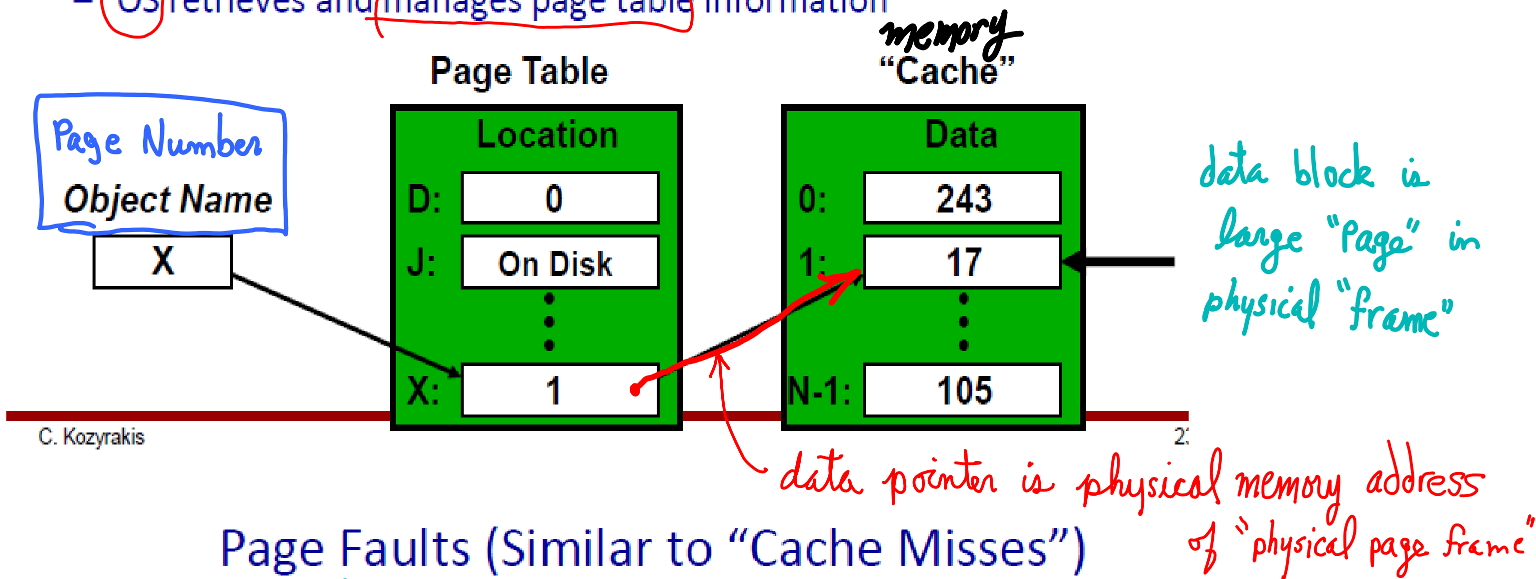
Tag: 0, 1, 2, 3, 4, 5, ..., n-1

Cache Data — WORD } Cache Block

only some tags have pointers to data

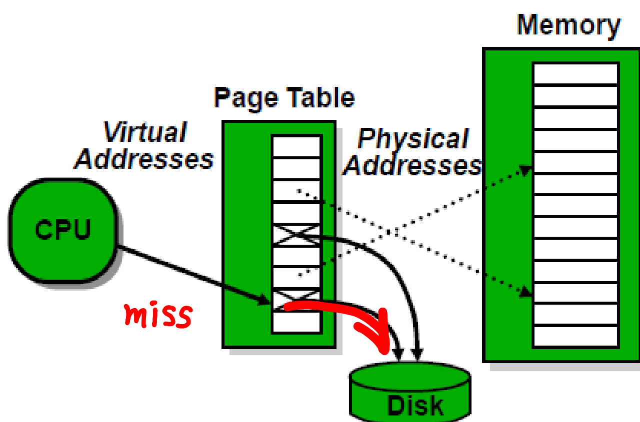
Locating an Object in a "Cache" (cont.)

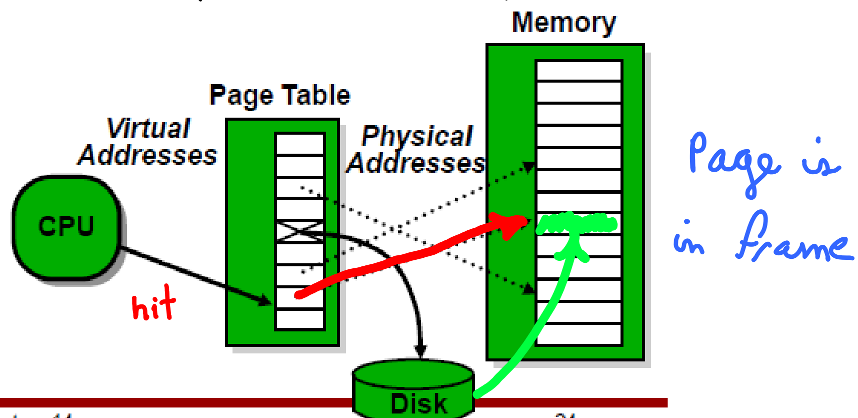
- DRAM Cache (virtual memory)
  - Each page of virtual memory has entry in page table
  - Mapping from virtual pages to physical pages
    - One entry per page in the virtual address space
  - Page table entry even if page not in memory
    - Specifies disk address
  - OS retrieves and manages page table information



Page Faults (Similar to "Cache Misses")

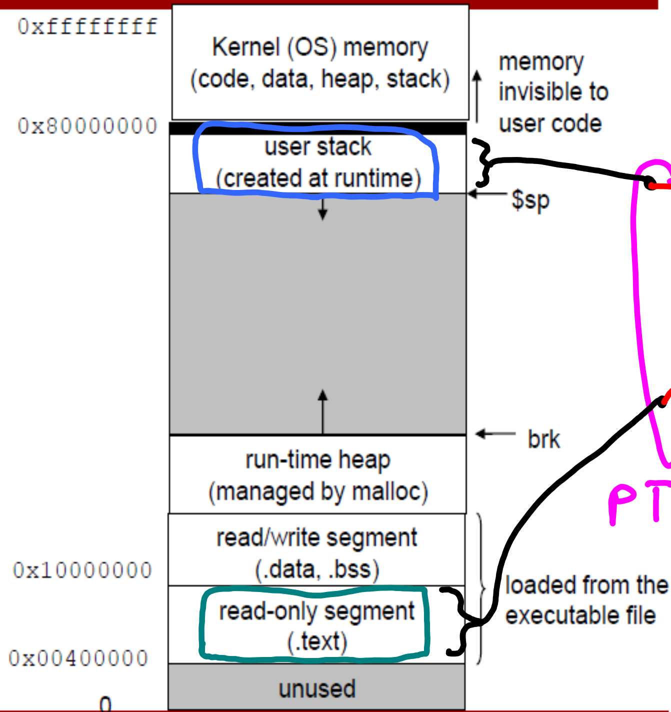
- What if an object is on disk rather than in memory?
  - Page table entry indicates virtual address not in memory — Valid bit + disk address
  - OS exception handler invoked to move data from disk into memory
    - OS has full control over placement
    - Full-associativity to minimize future misses — any memory "frame" holds any "page"

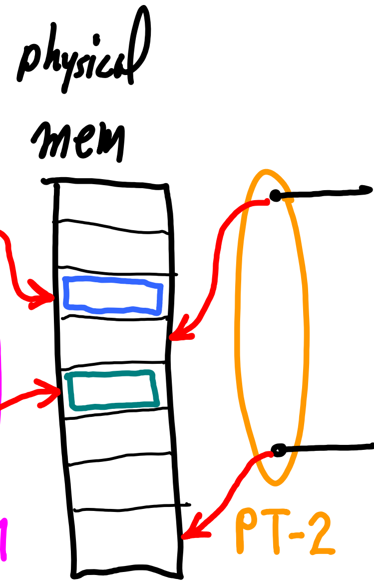




Does VM Satisfy Original Motivations?

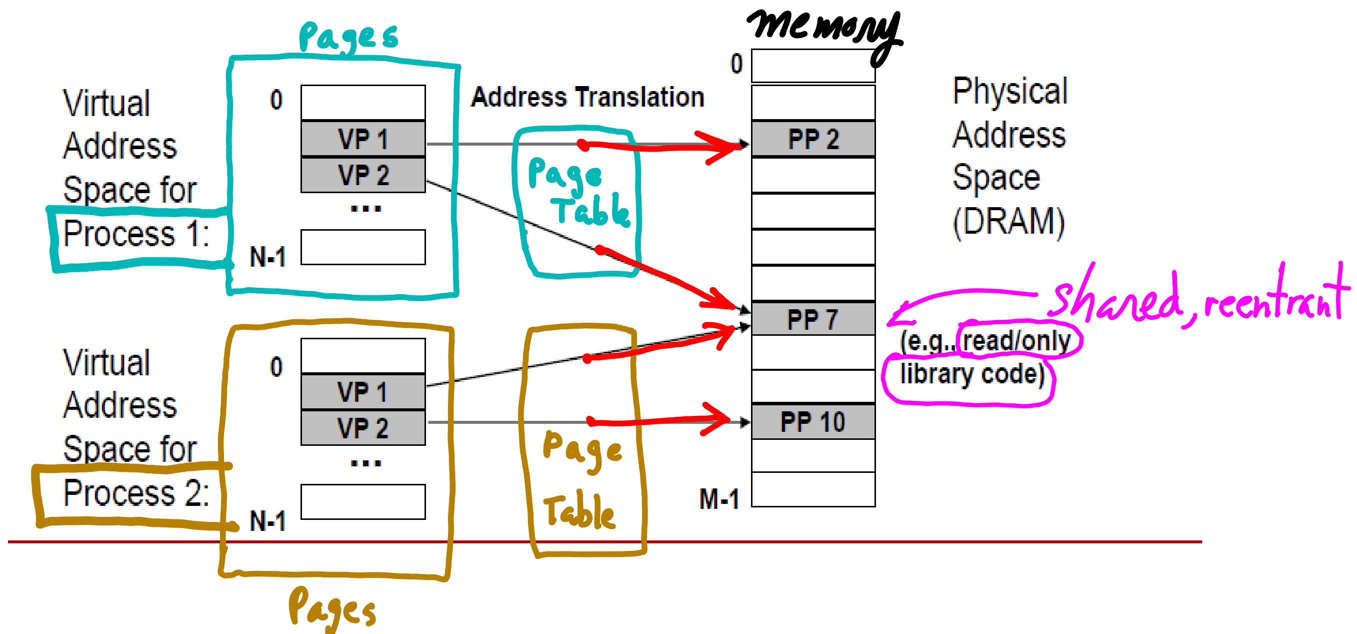
- Multiple active programs can share physical address space
- Address conflicts are resolved
  - All programs think their code is at 0x400000...
- Data from different programs can be protected how?
- Programs can share data or code when desired how?





Answer: Yes using 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



I'v got page table ***issues***

--- **Where** are the page tables, physically?

===> **memory? SRAM? Hardware?**

--- If in memory, **how many memory accesses** to read one data item (ignore cache)?

--- If page tables are **read/write**

===> **Can my program rewrite your page table (or my own, accidentally)?**

--- If page tables are **not read/write**, how do they get pointer values?

===> **Need protection bits per page**: **Kernel Mode 0:** **R/W**, **User Mode 1:** **no R/W**
===> Where do protection bits go? How are they accessed?

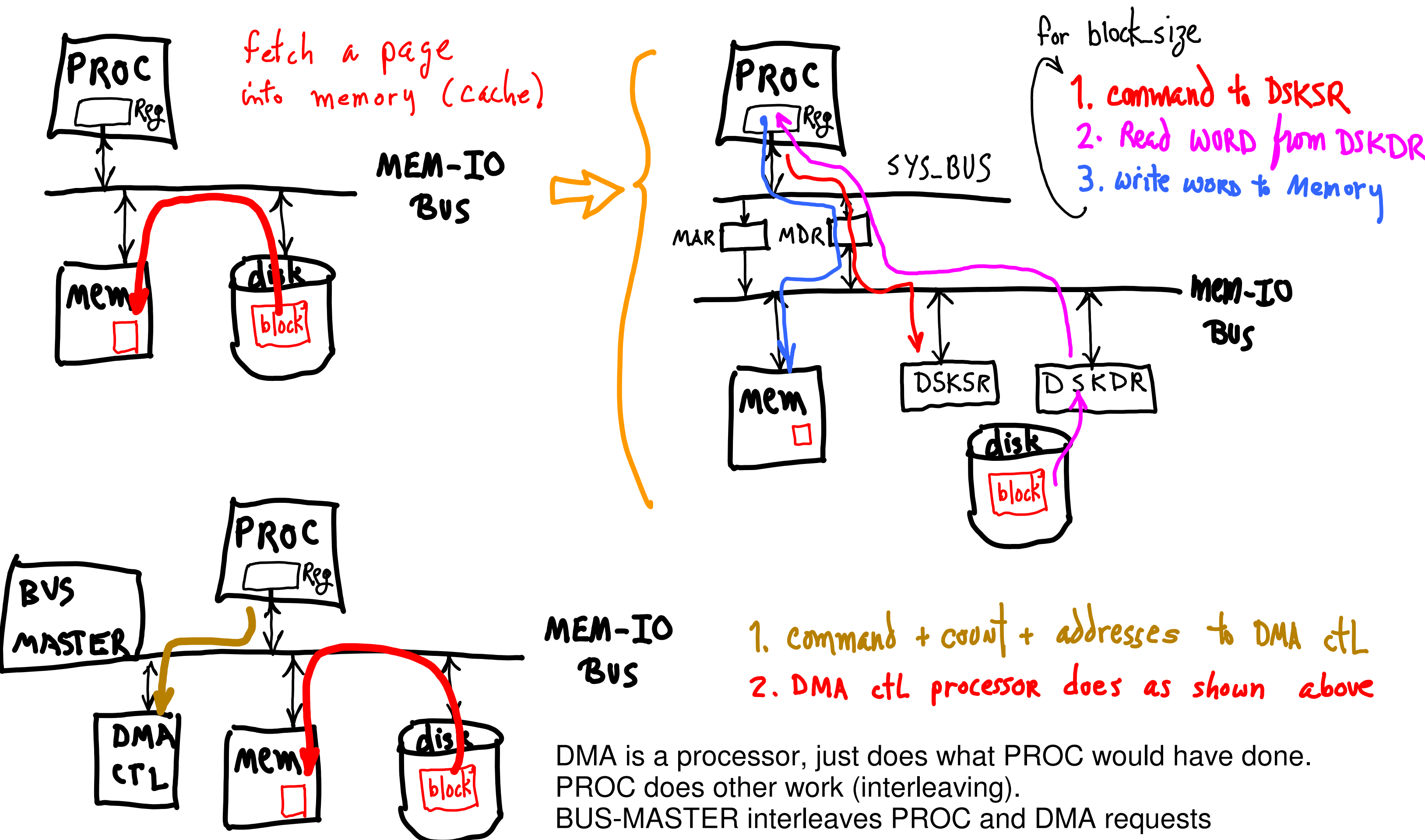
--- It's nice to **share** memory, but **why bother**?

===> **Principle of interleaving**: long latency task? Go find other work to do.
===> OS has work to do, too.

--- What about I/O?

===> Is that done using virtual addresses? **Memory mapped I/O device registers**?

--- Speaking of I/O, what about long, slow **I/O for disk blocks (pages)?**

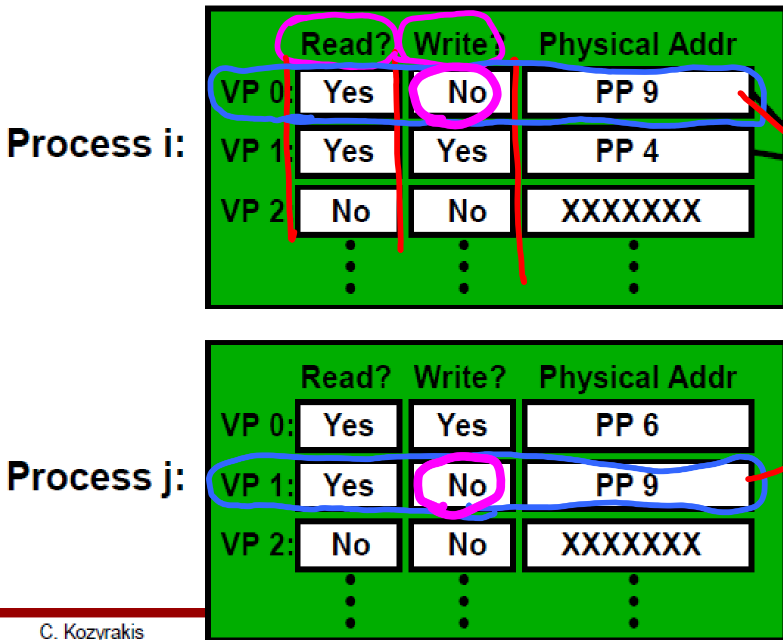


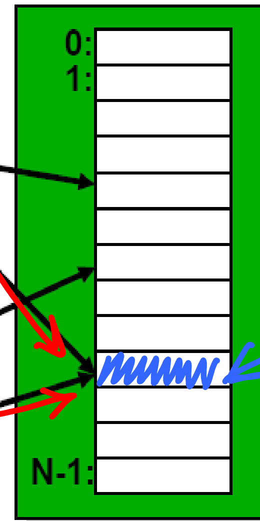
DMA is a processor, just does what PROC would have done.
PROC does other work (interleaving).
BUS-MASTER interleaves PROC and DMA requests

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



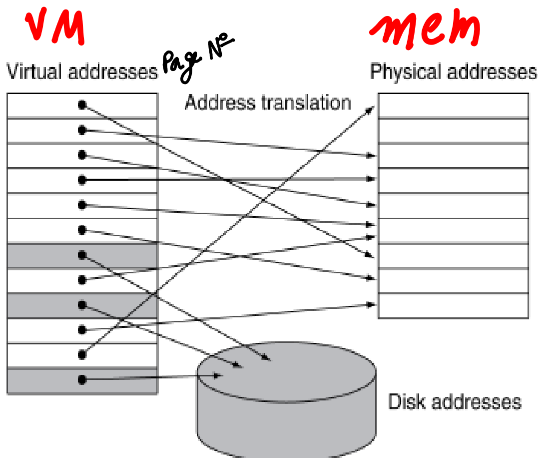


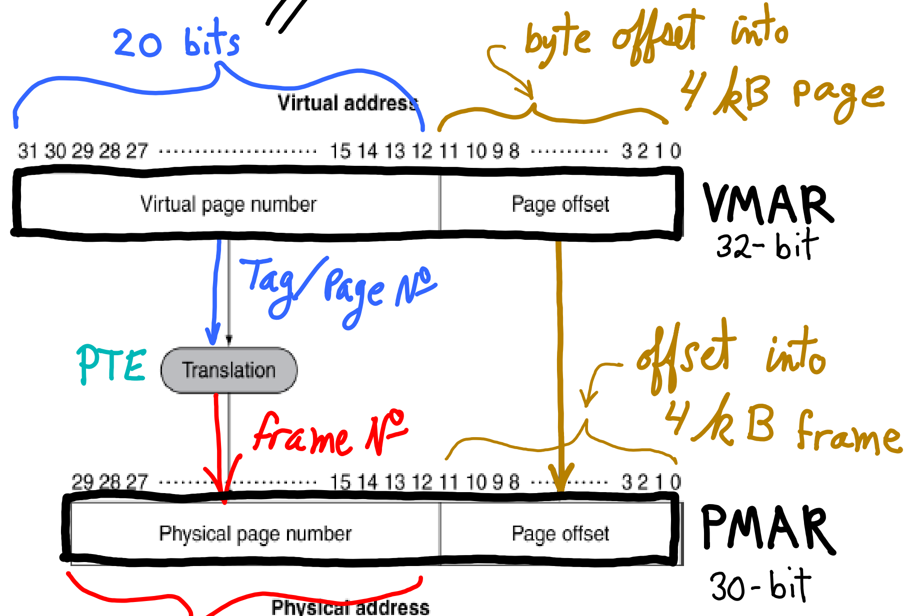
e.g., shared code not writeable

Translation: High-level View

2^{20} pages @ 4kB

- Fixed-size pages (e.g., 4K)





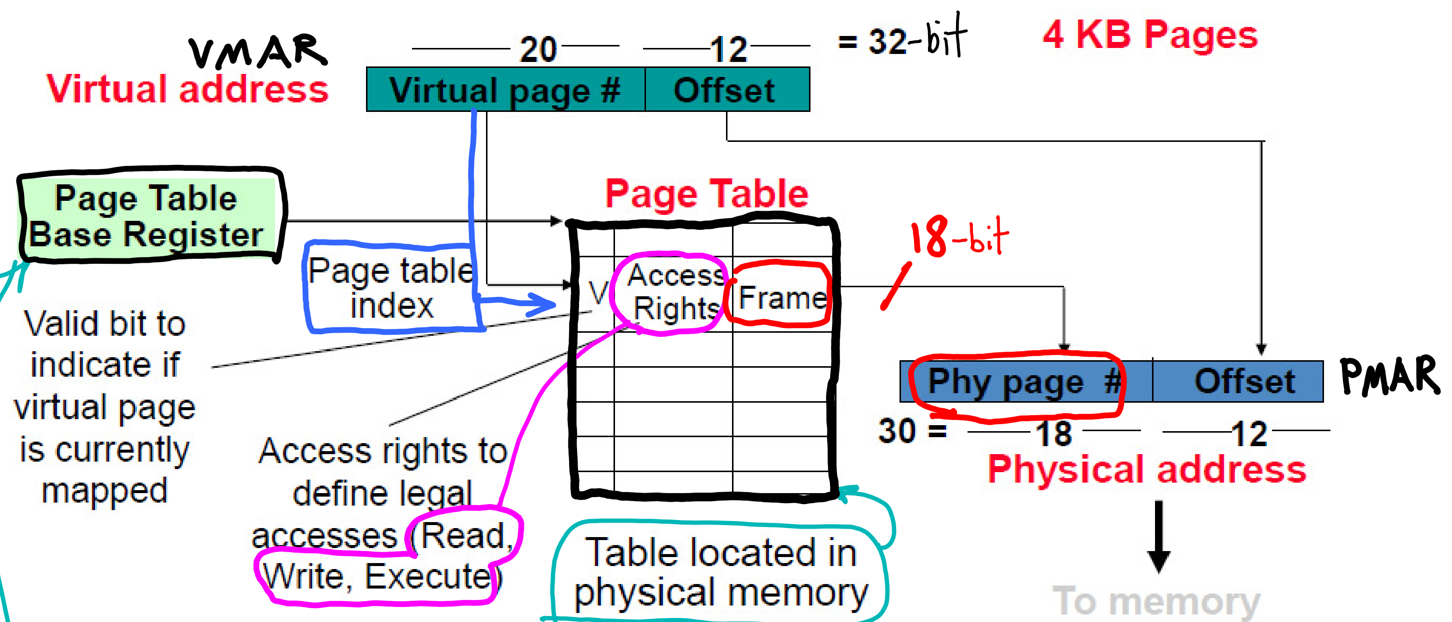
18-bit Physical frame number

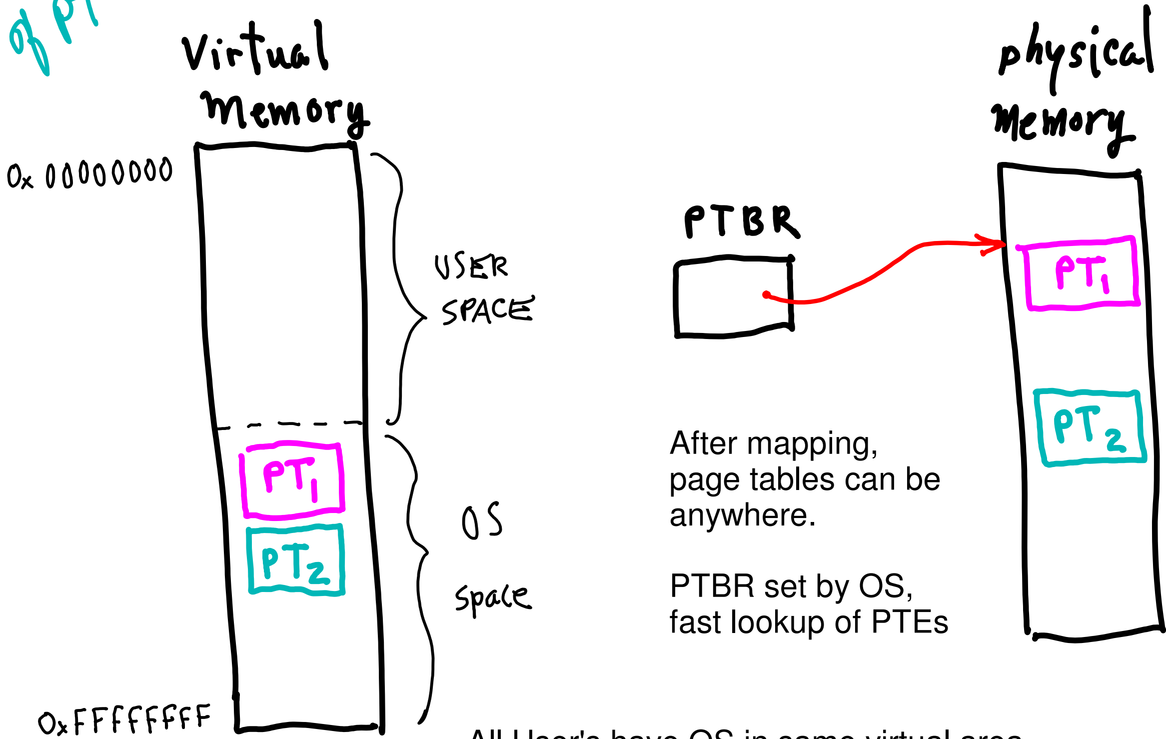
- Physical page sometimes called a frame

Bigger space? 64-bit ⇒ $2^{64} = 2^{30} \cdot 2^{30} \cdot 16 = (16G)GB$

2^{18} frames @ 4kB

Translation: Process





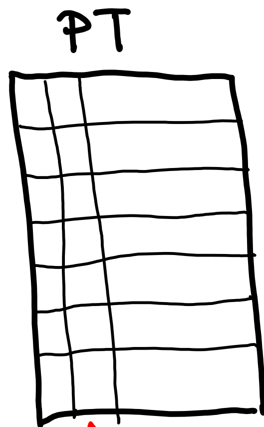
All User's have OS in same virtual area.

All virtual OS space is mapped identically for all users.

OS can turn off virtual addressing to access physical memory.

PTBR holds physical address of PT for fast access.

Replacement Policy | LRU approximation, OS



Set dirty bit on write.

Set accessed bit on read or write.

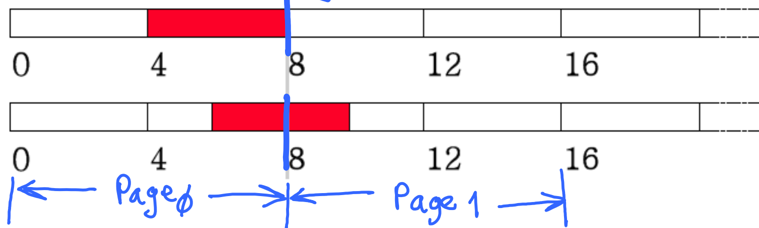
Clear all accessed bits every k ticks.

Page Miss:
--- evict page (ordered by 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

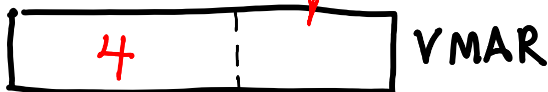
- What happens if unaligned address access straddles a page boundary?
  - What if one page is present and the other is not?
  - Or, what if neither is present?
- MIPS architecture disallows unaligned memory access
- Interesting legacy problem on 80x86 which does support unaligned access

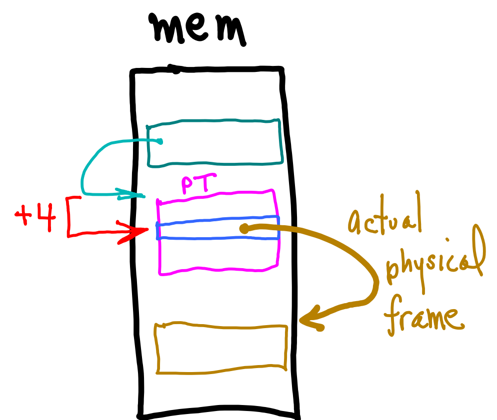
---

How many memory references

lw $7, 0x00040000

16-bit offset (64 kB page)





lw $1, Page-Table-Location-Pointer — get addr of PT

lw $2, 4 ($1) — get PTE

lw $7, ($2) — get data

(NOTE: operations in hardware, not instruction execution.)

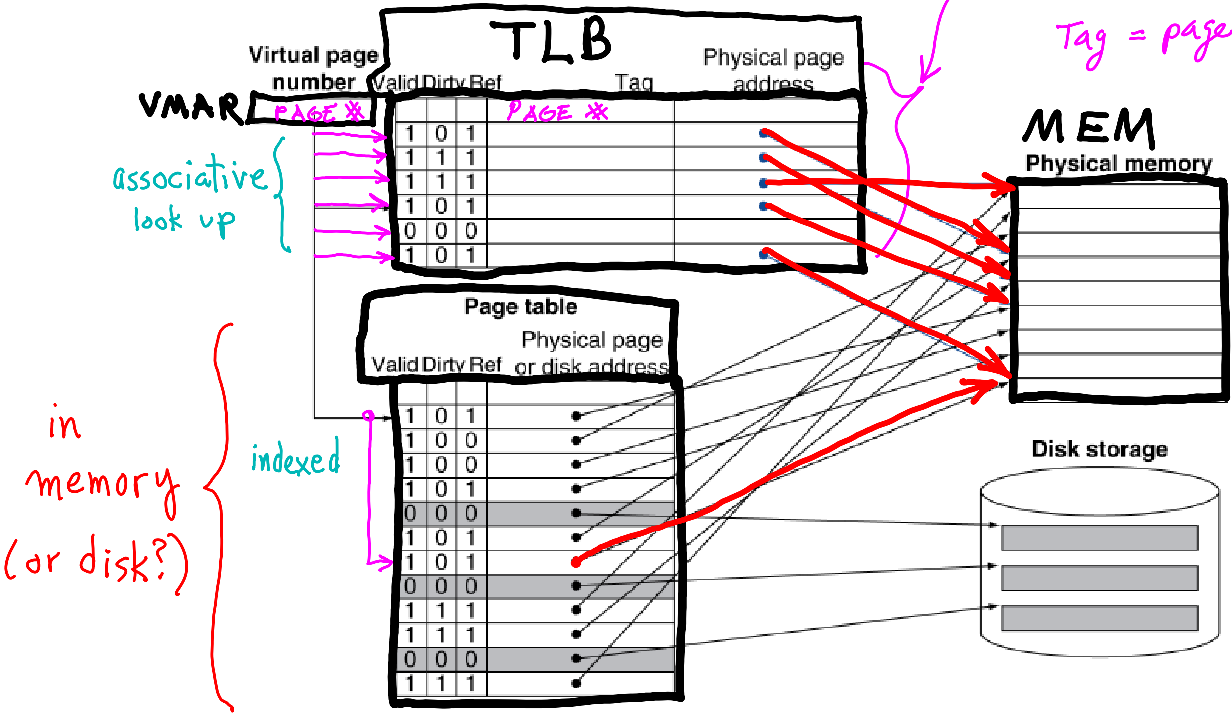
Speed it up:
1. PTBR <== **Page-table-location-pointer**
Do this once at program startup

2. Cache PTEs!

Fast Translation Using a TLB

fully assoc. cache for PTEs

Tag = page #



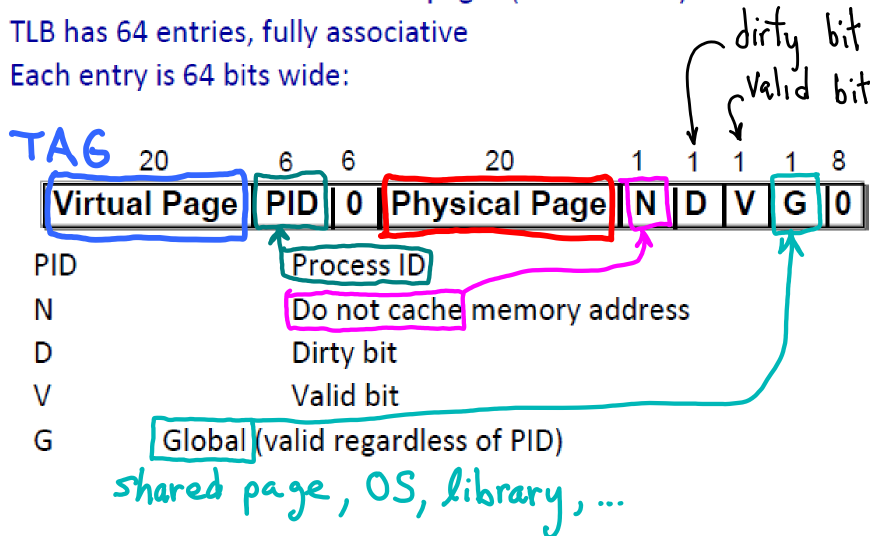
TLB Entries

- The TLB is a cache for page table entries (PTE)
- The data for a TLB entry (== a PTE entry)
  - Physical page number (frame #)
  - + Access rights (R/W bits)
  - + Any other PTE information (dirty bit, LRU info, etc)
- The tags for a TLB entry
  - Virtual page number
    - Portion of it not used for indexing into the TLB { N-way Set Associative
  - Valid bit
  - LRU bits
    - If TLB is associative and LRU replacement is used

TLB entry replacement info

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:



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

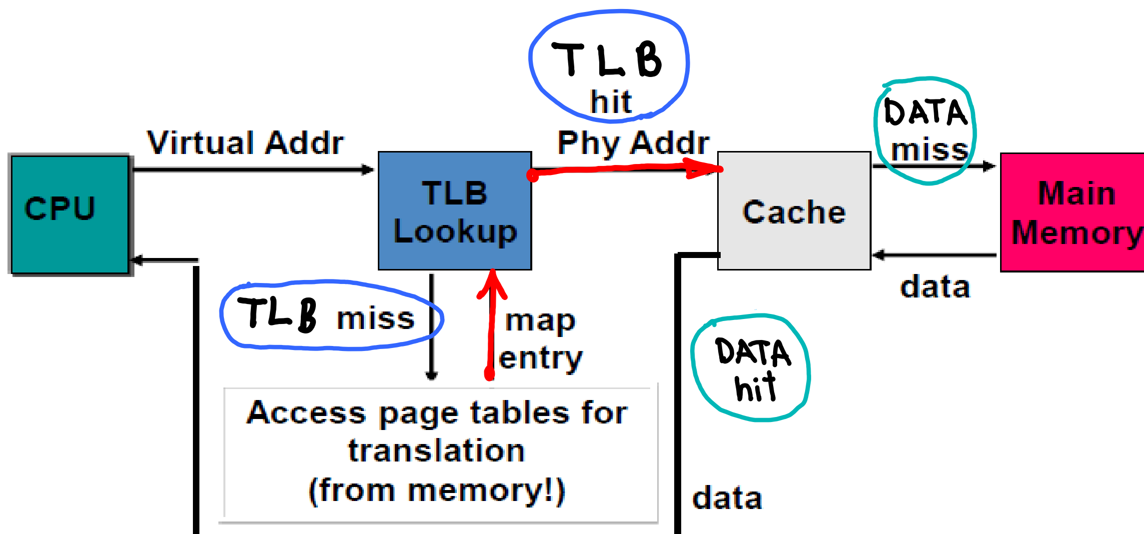
TLB Misses → TLB exception handler

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 Load PTE to TLB
  - Then restart the faulting instruction

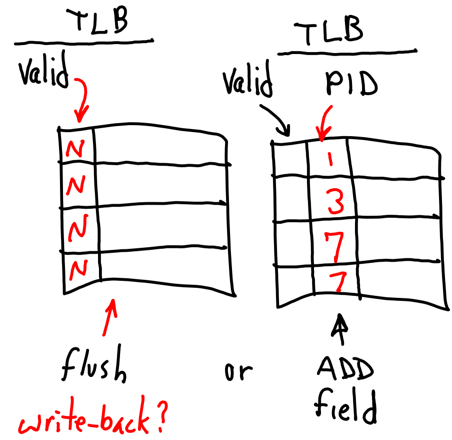
TLB & Memory Hierarchies

- Once address is translated, it used to access memory hierarchy
  - A hierarchy of caches (L1, L2, etc)



Virtual address
31 30 29 .... 14 13 12 11 10 9 .... 3 2 1 0
VMAR
Virtual page number
Page offset
20
12
TLB
Valid Dirty
Tag
Physical page number
TLB
TLB hit
PAGE #
FRAME #
20
physical Tag
CMAR
Physical page number
Page offset
Physical address
Physical address tag
Cache index
Block offset
Byte offset
MAR
18
8
4
2
4
4
Cache index
8
TAG
Valid
Tag
Data
4
Cache
TAG
Cache hit
1 cache block, 16 words
32
Data

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
  - 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
  - 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

Page Size Tradeoff

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

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

Multiple Page Sizes

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

OS sets MMU

Final Page Table Problem: Its Size

- Page table size is proportional to size of address space
- Example: Intel 80x86 Page Tables
  - Virtual addresses are 32 bits, pages are 4 KB
  - Total number of pages: $2^{32} / 2^{12}$ = 1 Million
  - 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!
- Why is this a problem?
  - The page table must be resident in memory (why?)
  - What happens for the 64-bit version of x86?
  - What about running multiple programs?

$2^N \rightarrow$ N-bit address

$2^m \rightarrow 2^m$ B page

$= 2^{N-m}$ entries

$m = 12$

$2^{32-12} = 2^{20}$

who uses all 2^{32} addresses?

map to disk, valid, …

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

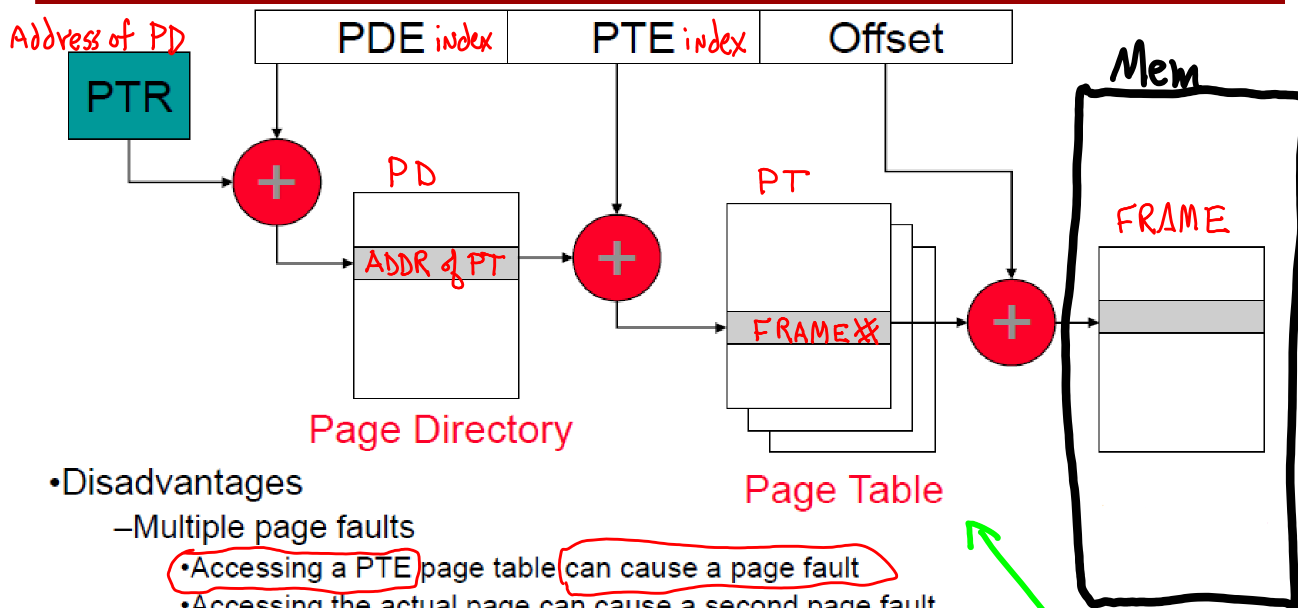
$(2^{20})(2^{30}) = (M)(G)$ entries!

Solution: Multi-Level Page Tables

- Use a hierarchical page table structure
  - Two levels are typically sufficient
    - First level: directory entries
    - Second level: actual page table entries
  - Only top level must be resident in memory
  - Remaining levels can be in memory, on disk, or unallocated
    - Unallocated if the corresponding ranges of the virtual address space are not used

? 64-bit, 128-bit address space? + more levels? ⇒ inverted page table

indexing ⇒ hashing +chaining



- 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

frame 0 1 2 3 | Page №

collisions? slow? does 'it matter? where is page table anyway?

Possibly unallocated pages

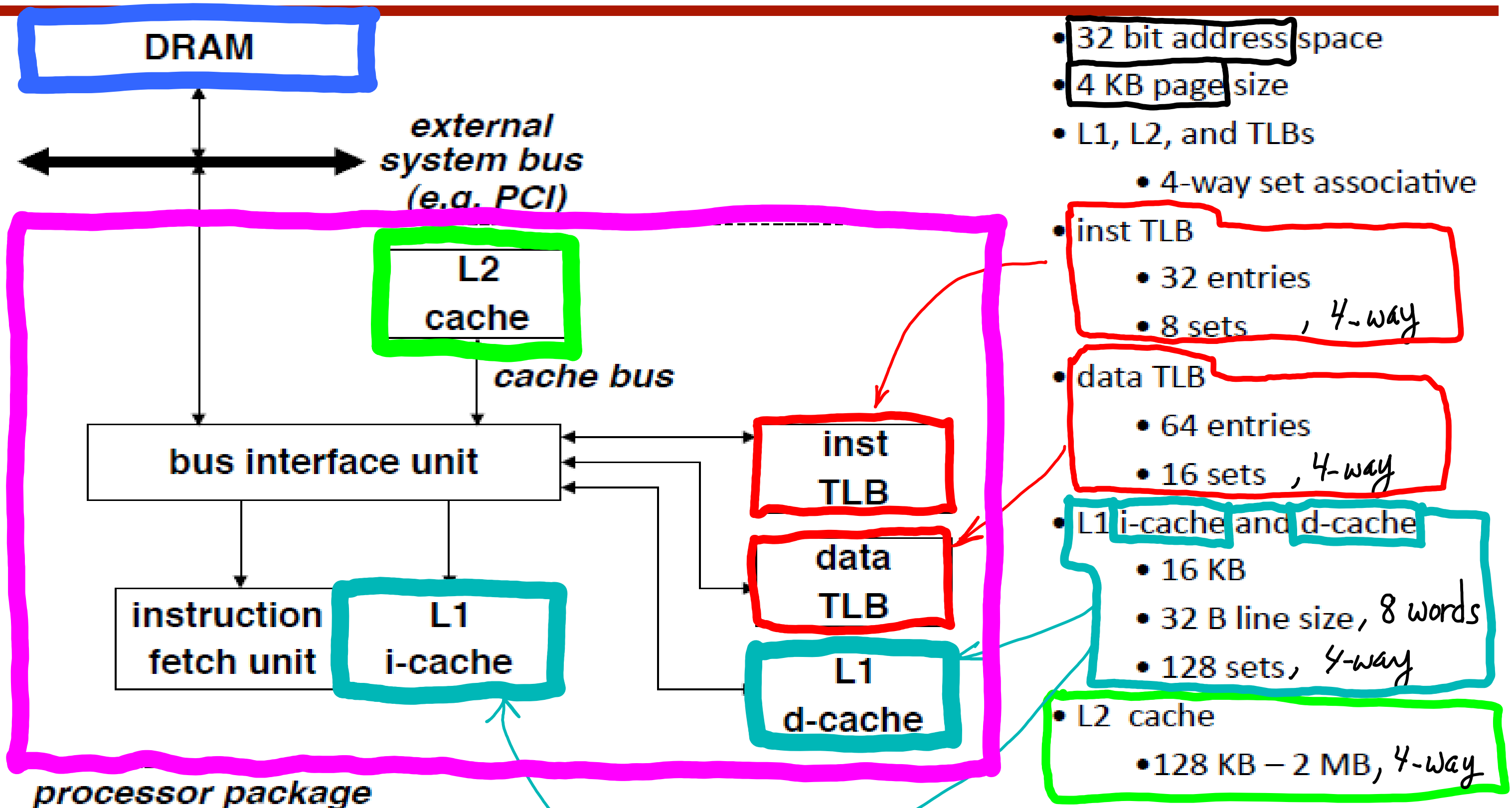
Real Example: Intel P6

- 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

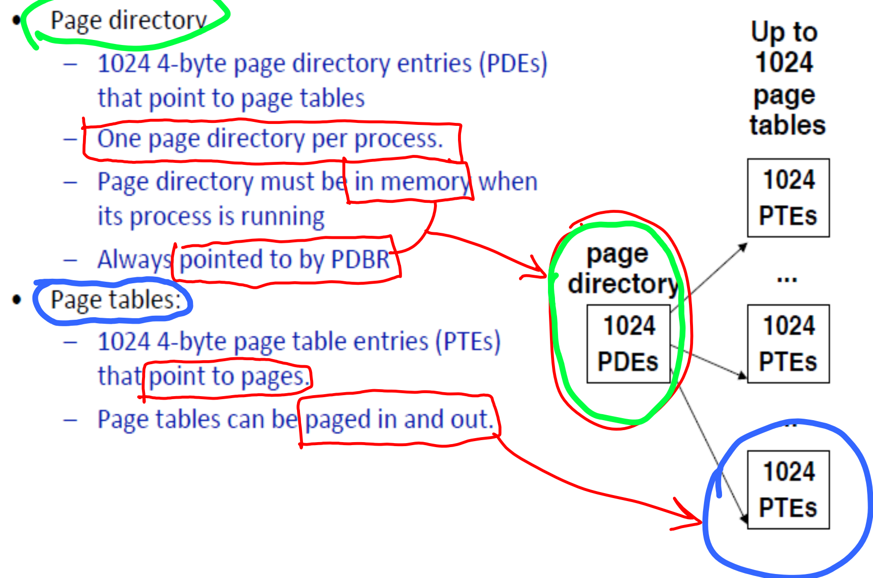
P6 memory system



- Components of the virtual address (VA)
  - TLBI: TLB index
  - TLBT: TLB tag
  - VPO: virtual page offset
  - VPN: virtual page number
- Components of the physical address (PA)
  - PPO: physical page offset (same as VPO)
  - PPN: physical page number
  - CO: byte offset within cache line
  - CI: cache index
  - CT: cache tag

} for set-assoc. TLB

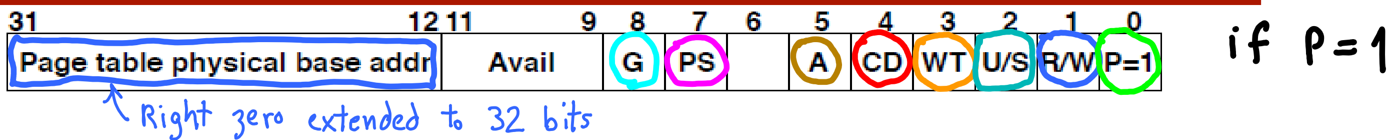
P6 2-level page table structure



Overview of P6 address translation

CPU
Data / instruction
32
result
Data
L2 and DRAM
VMAR
20 VPN 12 VPO
virtual address (VA)
16 TLBT 4 TLBI
index
4-way SA TAG
L1 hit
Data
L1 miss
L1 (128 sets, 4 lines/set)
TLB miss
TLB hit
...
TLB (16 sets, 4 entries/set)
cache index
Directory index
10 VPN1 10 VPN2
PT index
Directory
PAGE TABLE
PDE
PTE
TAG
20 PPN 12 PPO
20 CT 7 CI 5 CO
physical address (PA)
PDBR
Page tables
IN MEMORY

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



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)

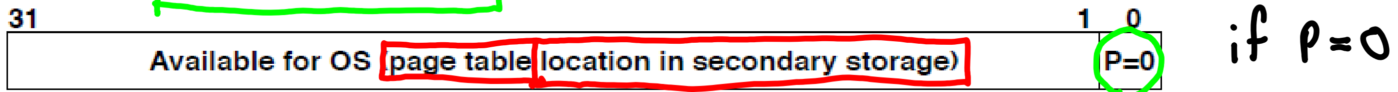
CD: cache disabled (1) or enabled (0)

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

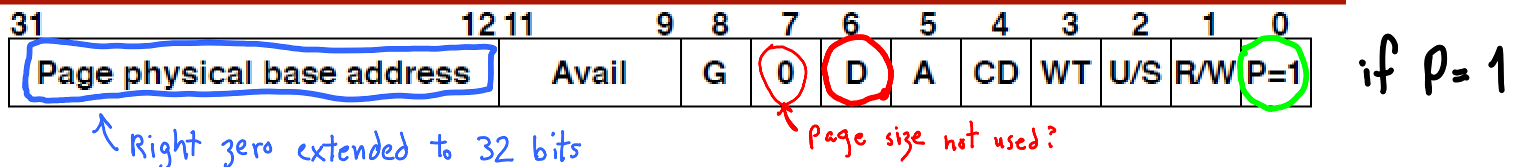
U/S: user or supervisor mode access

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

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



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



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)

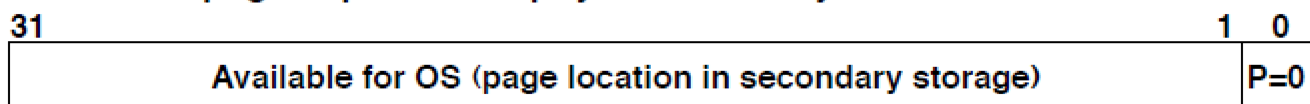
CD: cache disabled or enabled

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

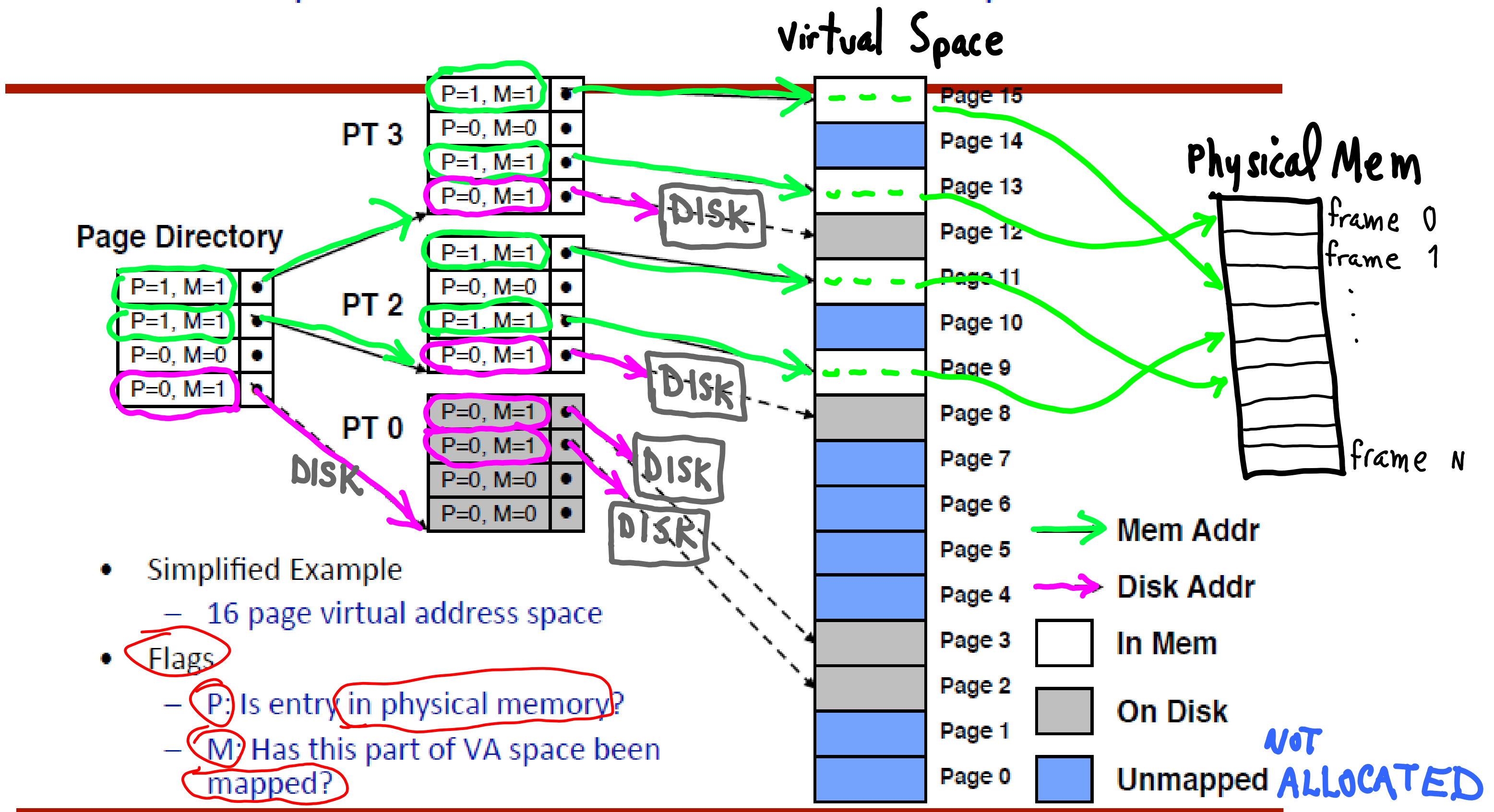
U/S: user/supervisor

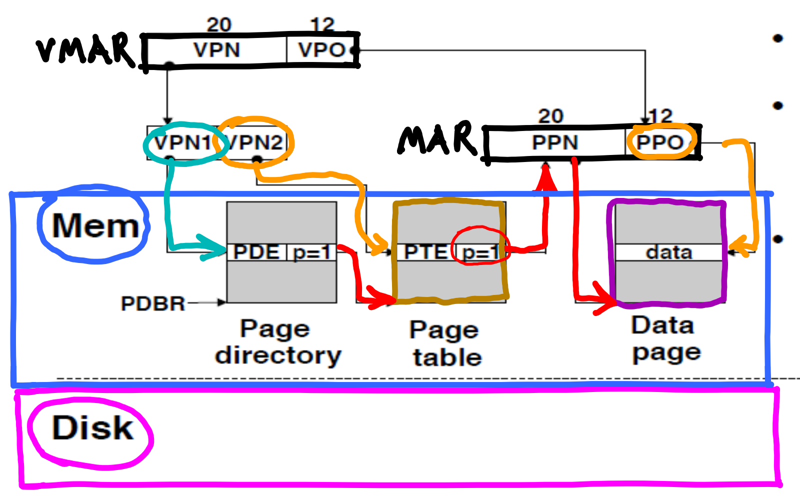
R/W: read/write

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



Mapped vs UnMapped pages



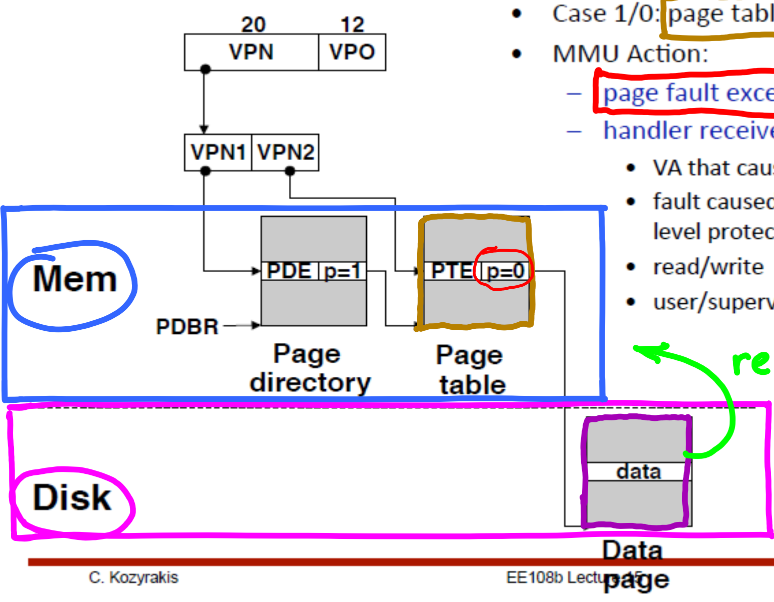


- Case 1/1: page table and page present.
- MMU Action:
  - MMU build physical address and fetch data word.
- OS action
  - none

Case

Page Table page in memory

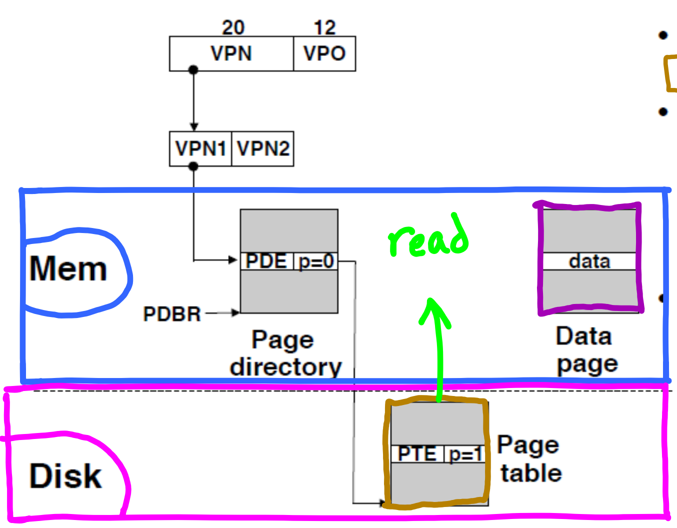
Data page in memory



- 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

data

- 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.

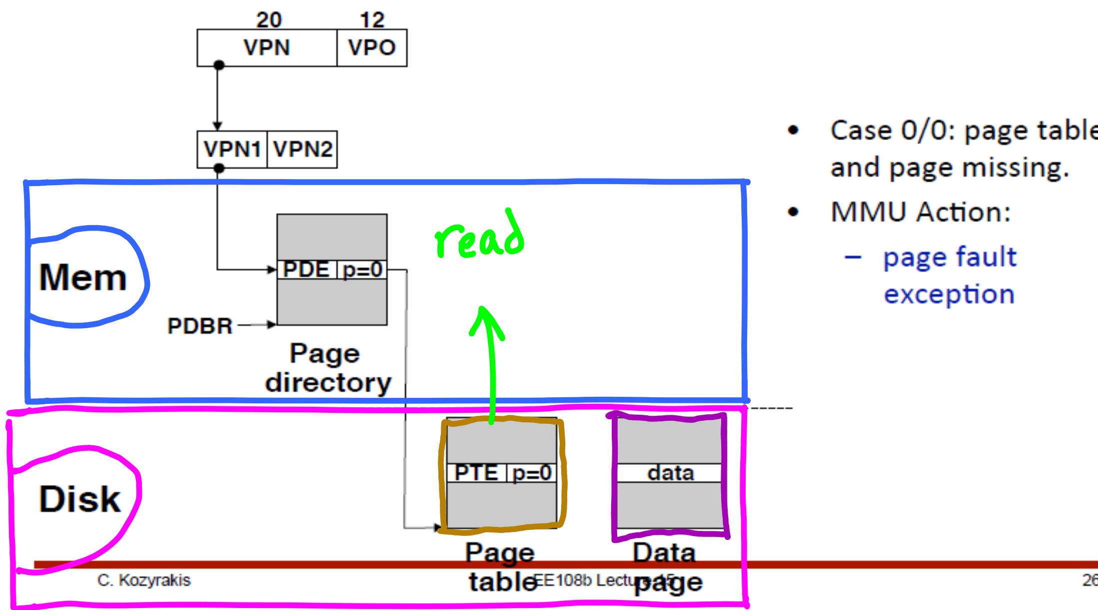


- Case 0/1: page table missing but
- Introduces consistency issue.
  - potentially every page out requires update of disk page table.
- Linux disallows this
  - if a page table is swapped out, then swap out its data pages too.

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.

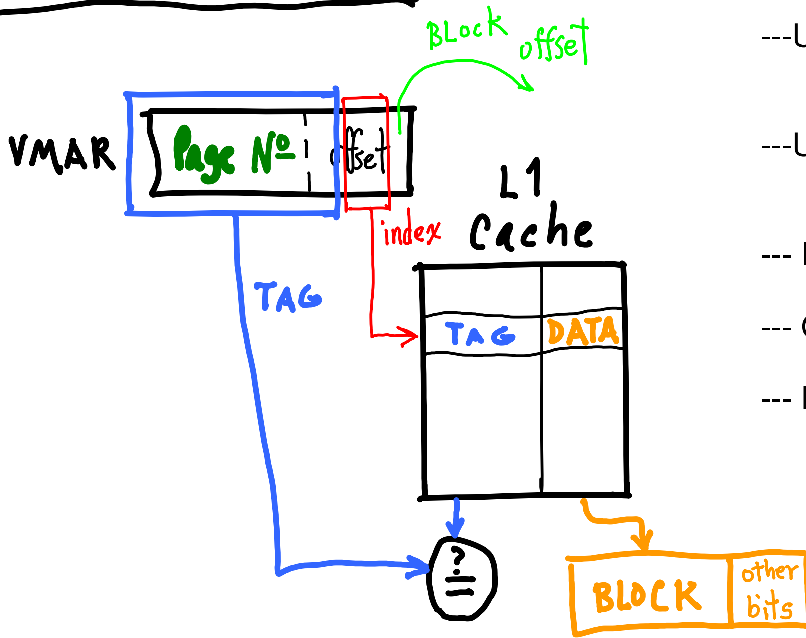
Read PDE, find PT disk address; Read PT page from disk; Restart;
(after restart: becomes Case 1/1)



- 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

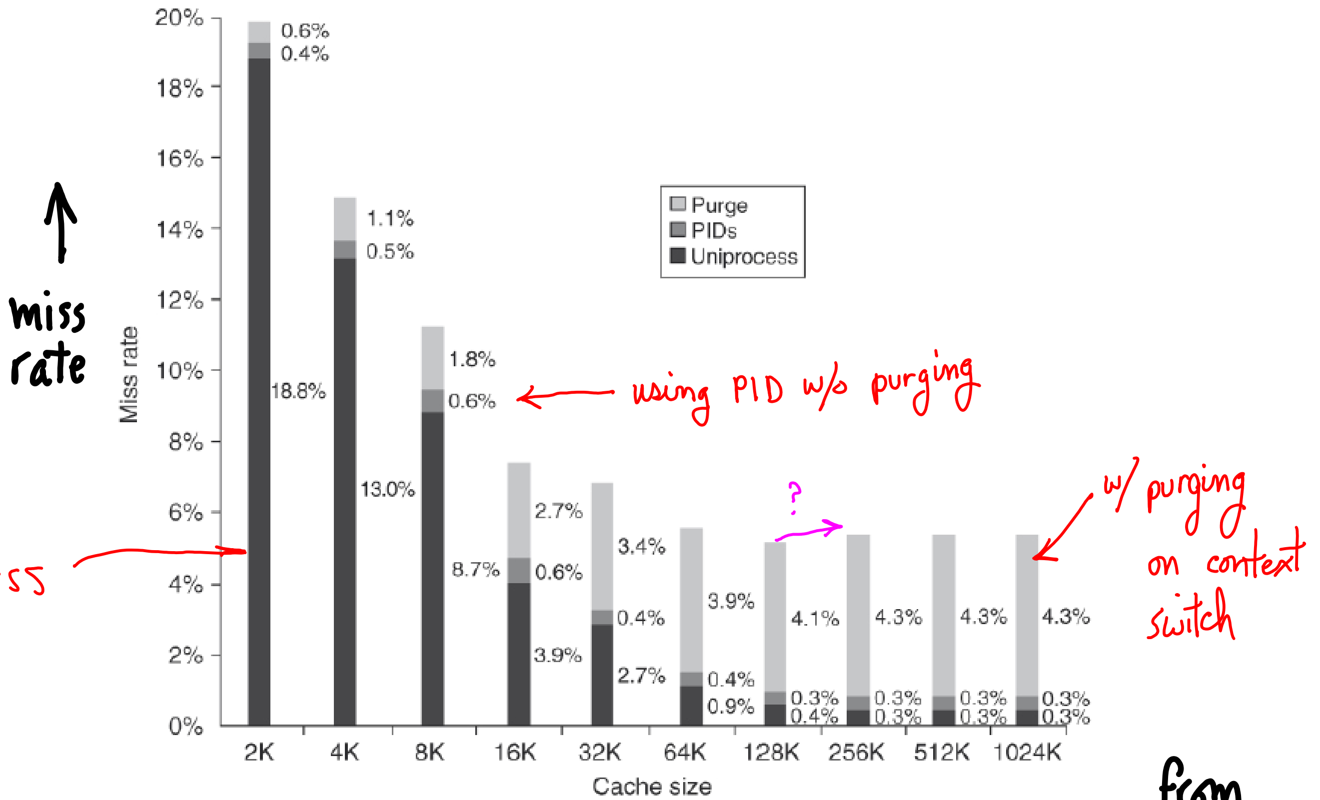
---Use part of virtual address as **tag** **(Page No. + or - some bits)**

---Use other bits for **index** into cache **(remainder is block offset)**

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

--- Only **translate on misses**

--- L2 is a physical cache



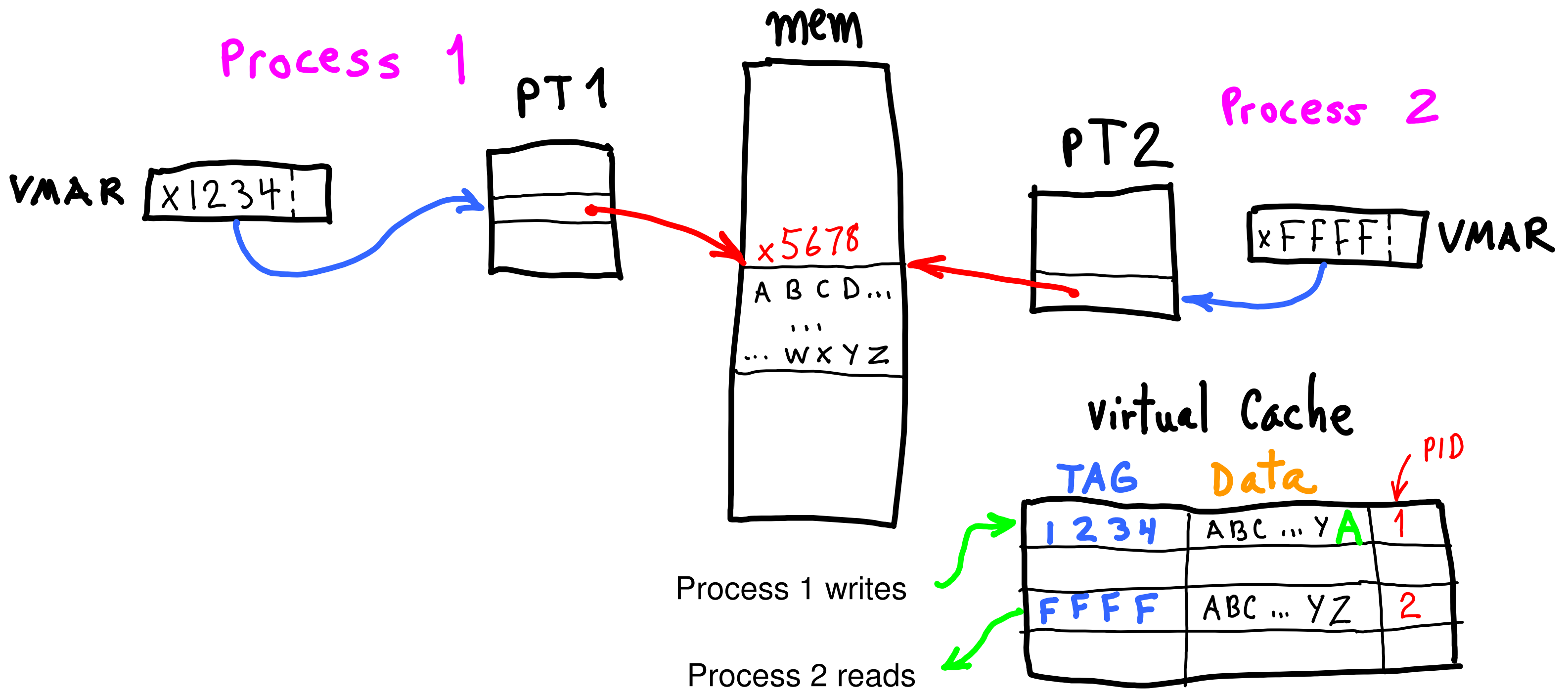
16 B blocks, DM Cache Size →

from H&P

Want PID

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)

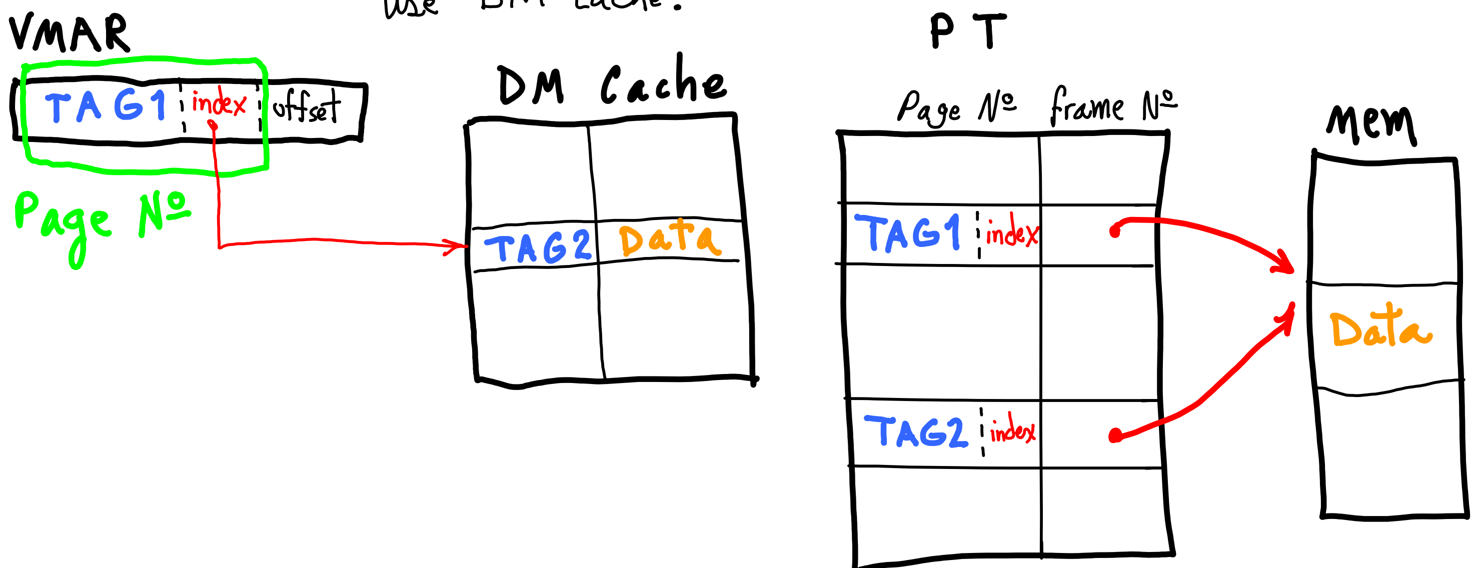


Goal Invariant:
--- Never allow this to happen :-)
--- Shared data only appears once, at most.

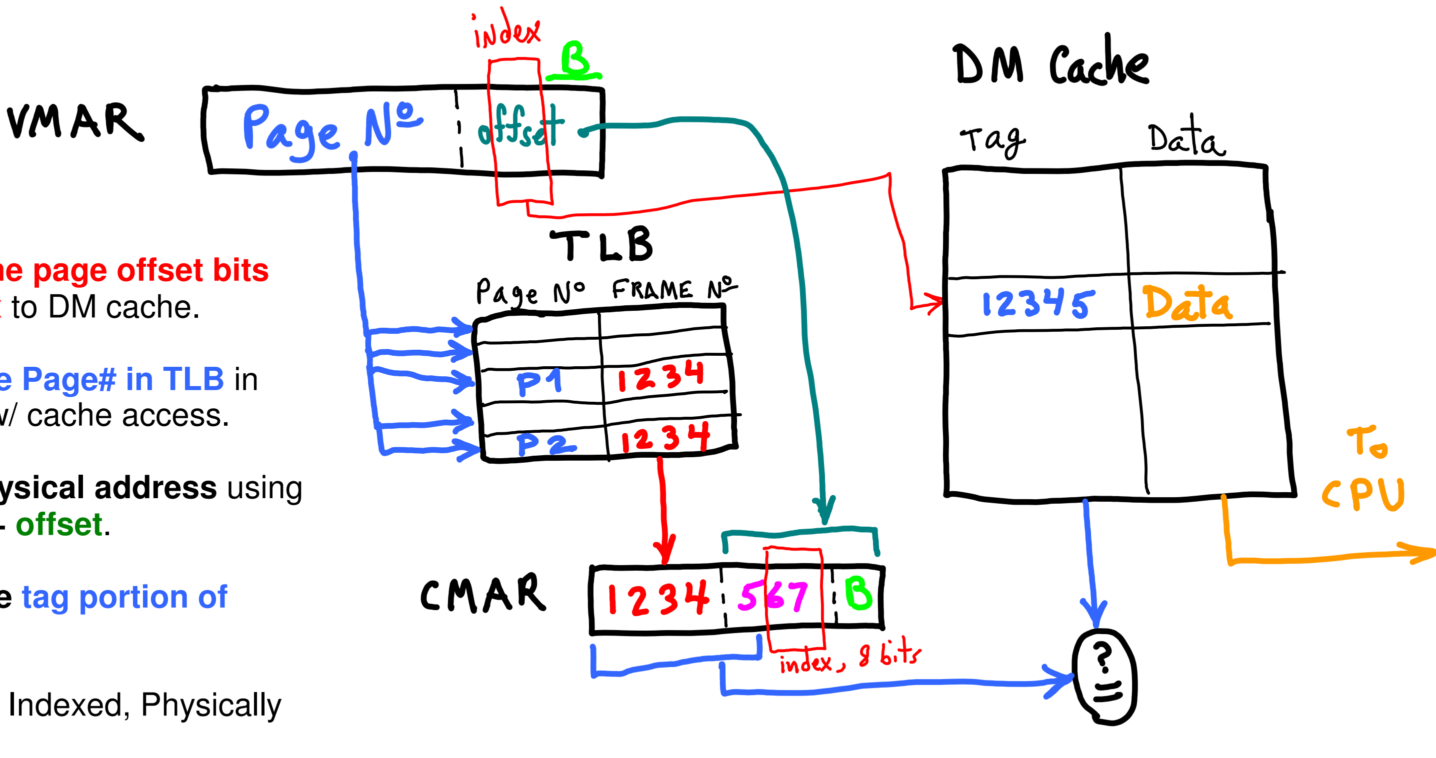
(Could also be multiple mappings in same page table.)

Solution 1

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



Solution 2a



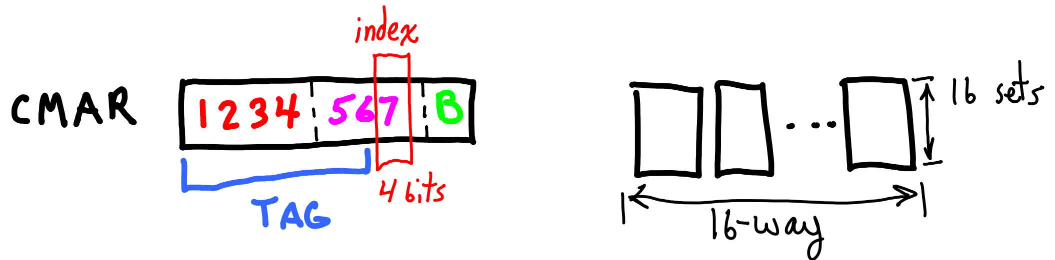
Use **some page offset bits as index** to DM cache.

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

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

Compare tag portion of CMAR.

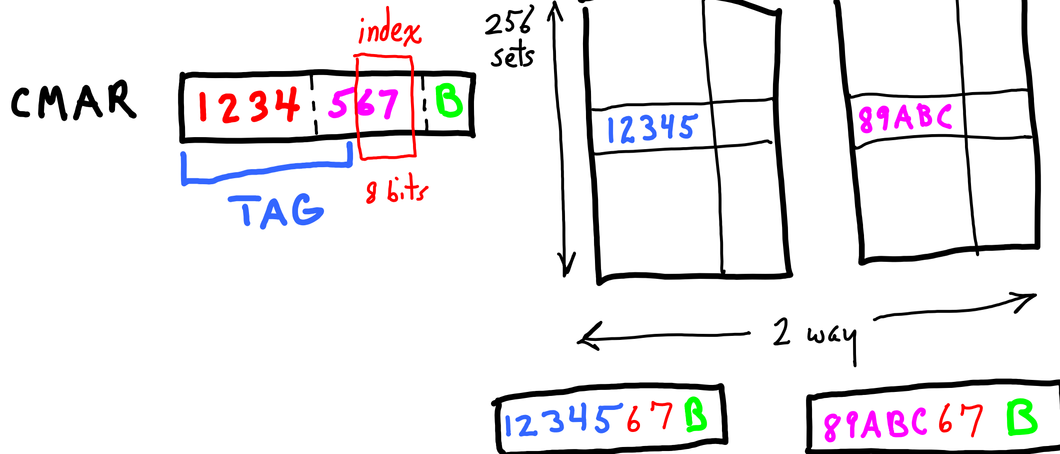
"Virtually Indexed, Physically Tagged"



INCREASE ASSOCIATIVITY

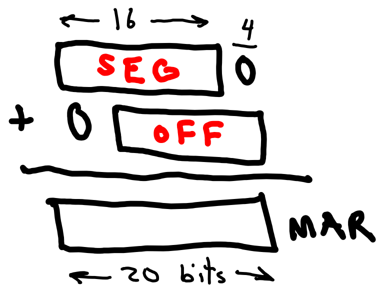
1. Fixed Cache Size
 ---- fewer index bits
 ---- more tag bits

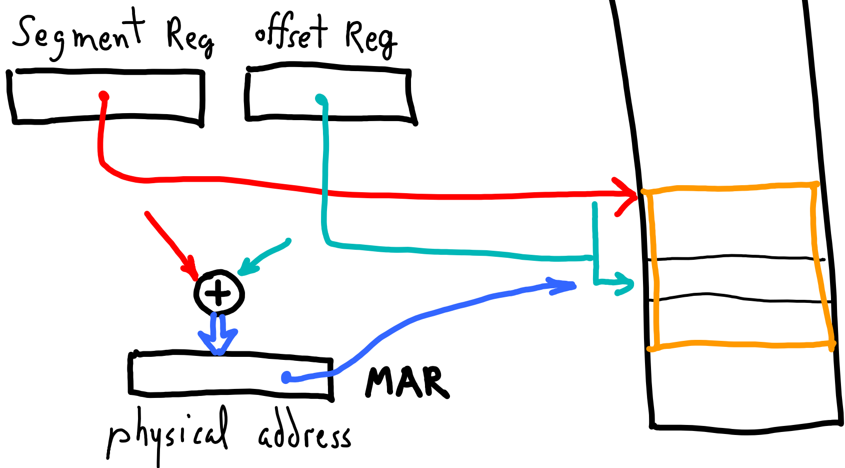
2. Increase Cache Size
 ---- same index bits
 ---- same tag bits



Segmented Memory

IA-32 / x86





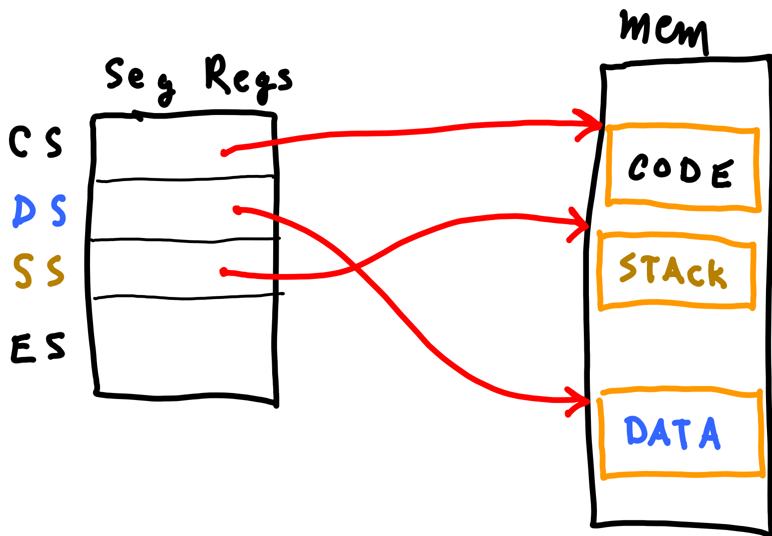
Originally

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

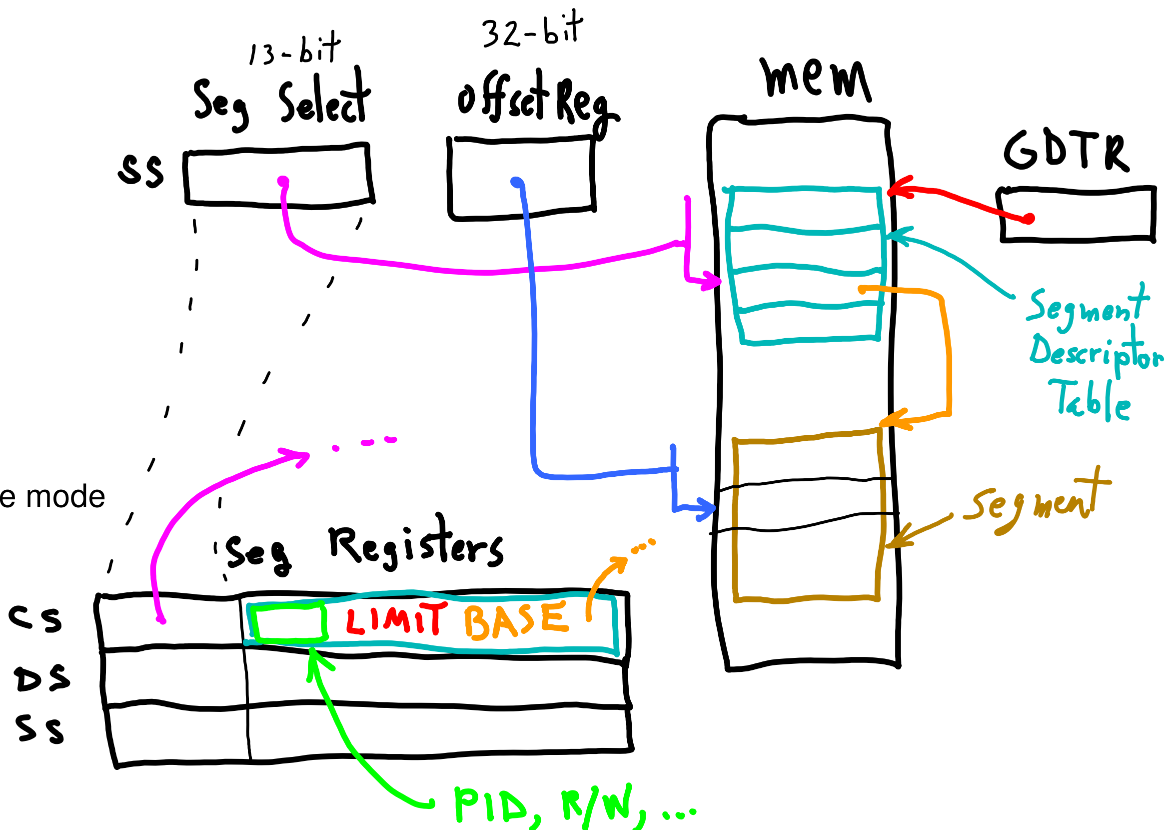
Programmer's perspective:
---- Segments address from 0
---- Offset is address



Next Level

Too Slow:
---- extend Seg Regs
---- cache Descriptor
in Seg Reg
---- Check limit and 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 CS, DS, SS can be written (change segments like original).
Descriptor table is OS controlled.

Also available in IA-32 (x86)

---- Paging mode (2-level and 3-level)
---- "Real" mode (acts like original)
---- Paged Segments (paging + segmentation: Segment Descriptor points to Page Directory)

Virtual Machines

Approach I
General TM Simulation

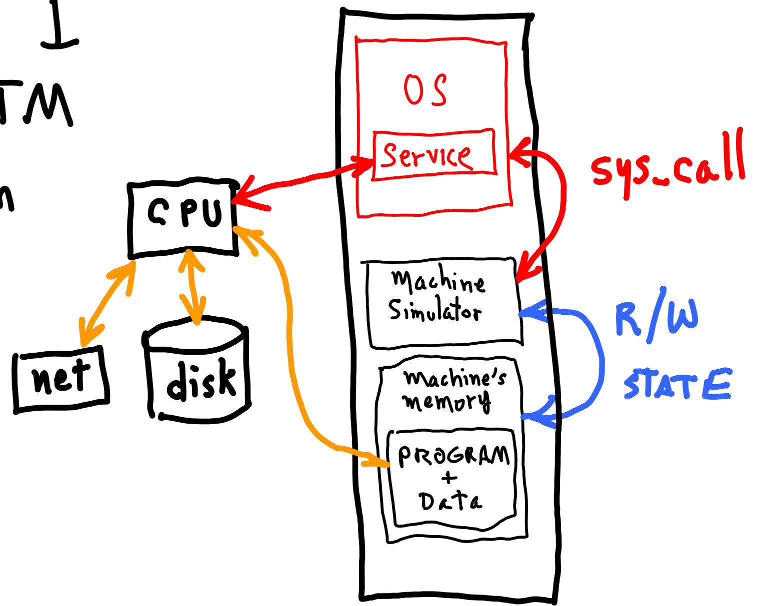
---- Simulated machine is arbitrary (HW, ISA)

--- Virtual Machine (VM) is defined by
simulation program.

--- VM's resources are in
simulator program's data structures.

--- Interaction with host is through
simulator's actions.

--- OS provides
--- IO services
--- isolation, protection



Just an ordinary Process

Is OS able to isolate/protect?
- complexity, bugs

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

Approach II

minimal "OS"

--- Small monitor provides
Mapping to partitioned resources

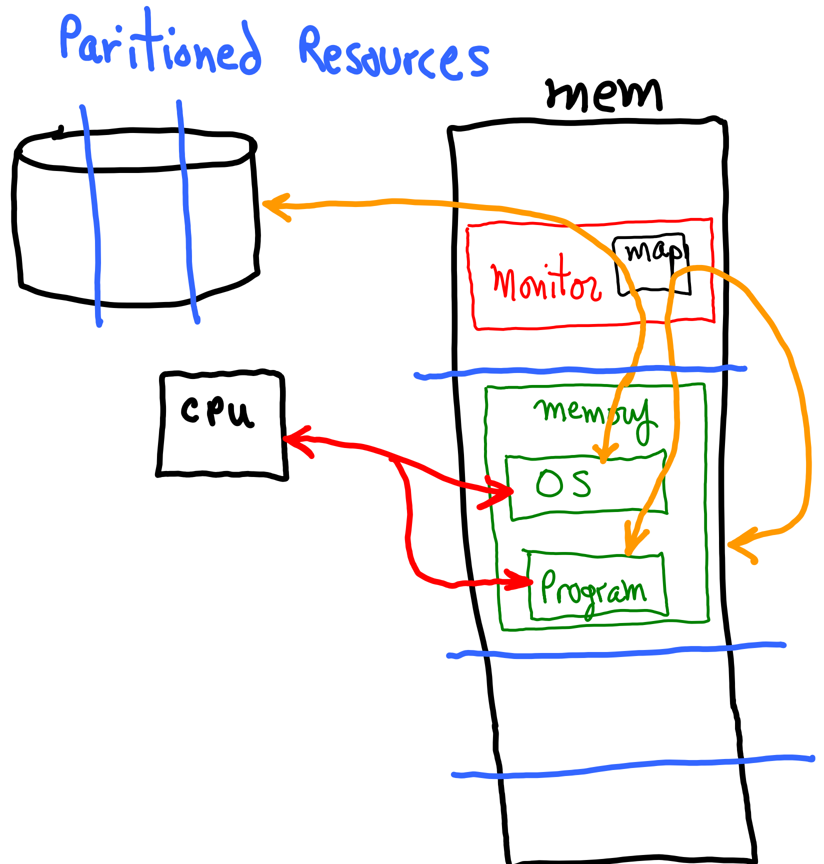
--- Monitor is small, simple, reliable

--- Each guest runs in its own VM

--- VM is only virtual in the mapping
Guest instructions run w/o emulation

--- Guest has same ISA as HW

--- Each VM has its own OS
manages its own resources



Some advantages

--- Monitor-1 and Monitor-2 present identical virtual machines to guests

--- Guest migration is possible: uptime, bulk efficiencies

--- Multiple guests share pool of computing resources

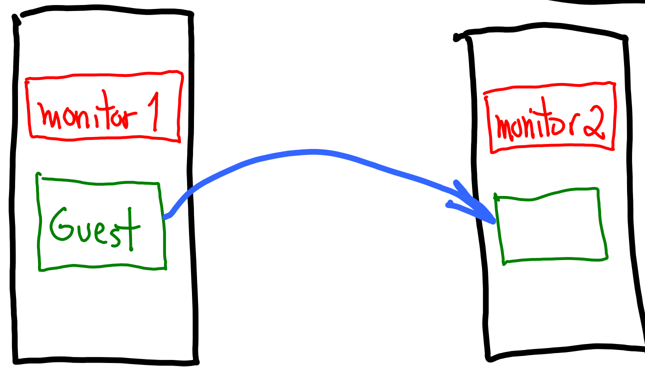
--- Isolation between guests (?)

--- HW architecture can be different between hosts (degree?)

--- Run legacy apps on legacy VM.

--- Guest OS configuration specific to guest's apps.

HW Platform 1 ≠ HW Platform 1



Monitor runs in kernel mode.

Guest runs in User mode:

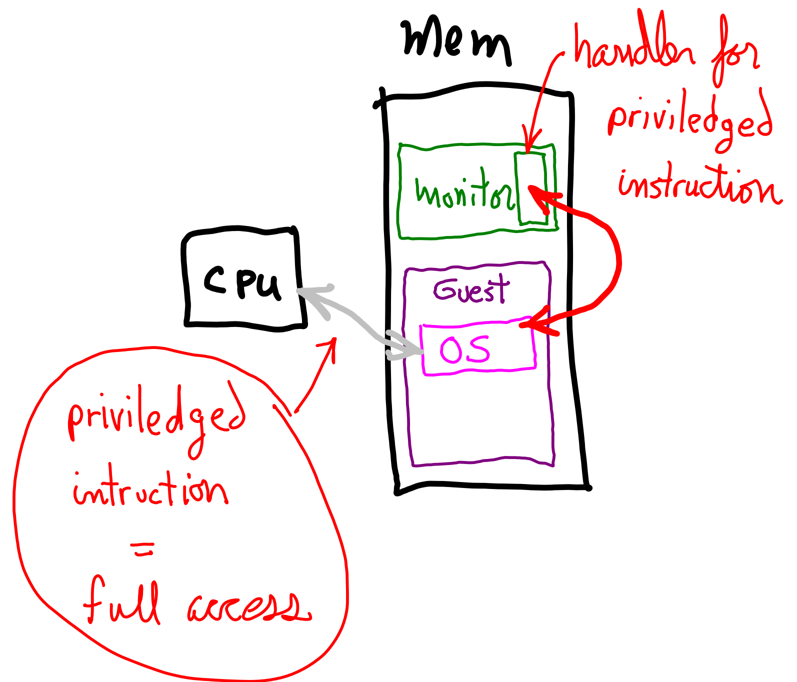
all priviledged instructions trap to monitor

OR

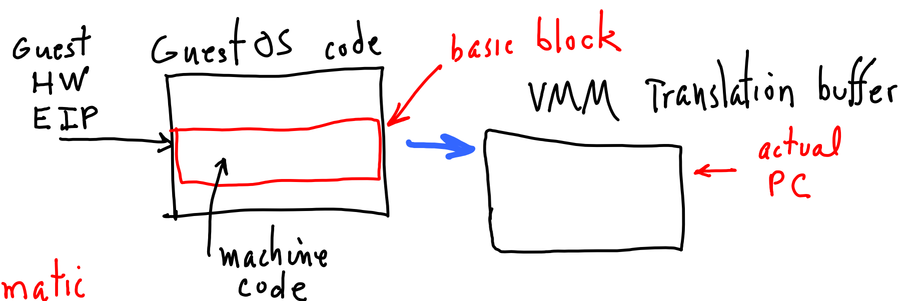
Binary translation (static or runtime):
--- Replace problematic instructions

OR

New hardware modes of execution.



Dynamic Translation



check for problematic instructions, replace them

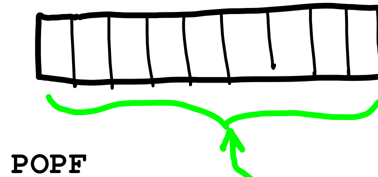
What problem instructions, can't we trap all of them?

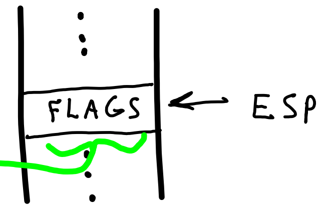
x86 virtualization!

x86 EFLAGS register, a larger version of our LC3's CC codes:

--- N, Z, P
--- Overflow
--- Carry
...
--- **I**nterrupt **E**nable
...

EFLAGS





POPF

Kernel mode
All 32-bits written

User mode
IE bit not written → x86: only kernel can alter system flags

Guest OS is running USER mode!
→ Does not cause a trap
→ Does not work correctly

METHODS

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

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

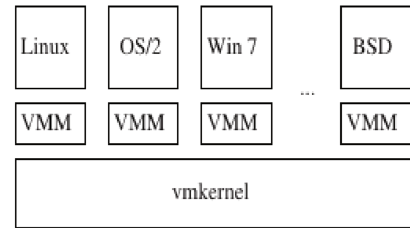


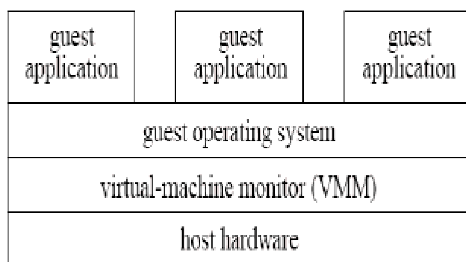
Figure 1: The ESX hypervisor: one vmkernel per host, and one VMM per virtual machine.

from **The Evolution of an x86 Virtual Machine Monitor**

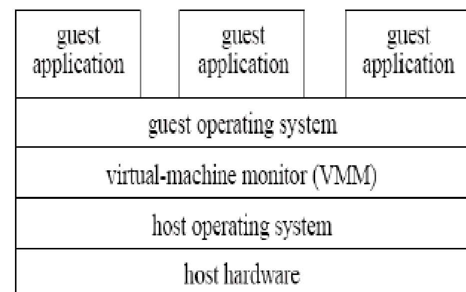
Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, Pratap Subrahmanyam

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.



Type I VMM

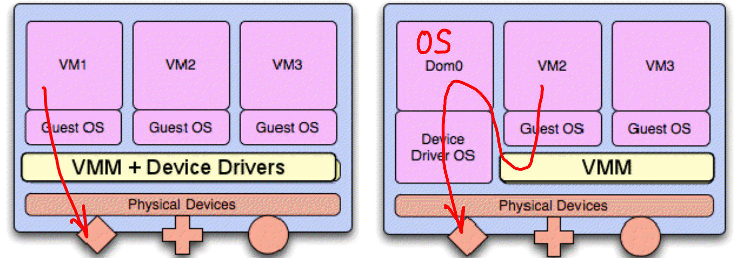


Type II VMM

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

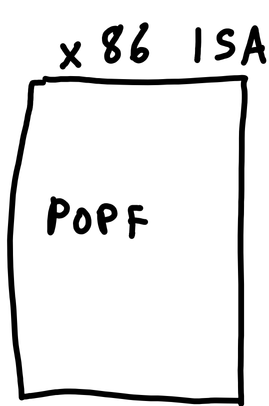
I/O Virtualization

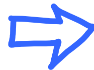
- Issue: lots of I/O devices
- Problem: Writing device drivers for all I/O device in the VMM layer is not a feasible option
- Insight: Device driver already written for popular Operating Systems
- Solution: Present *virtual* I/O devices to *guest* VMs and channel I/O requests to a trusted *host* VM running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

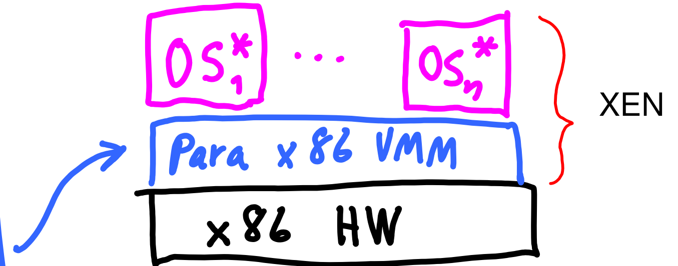


Para-Virtualization

Change the x86!







Have to rewrite the OS
--- use new ISA
===> more runs w/ VMM
===> faster

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

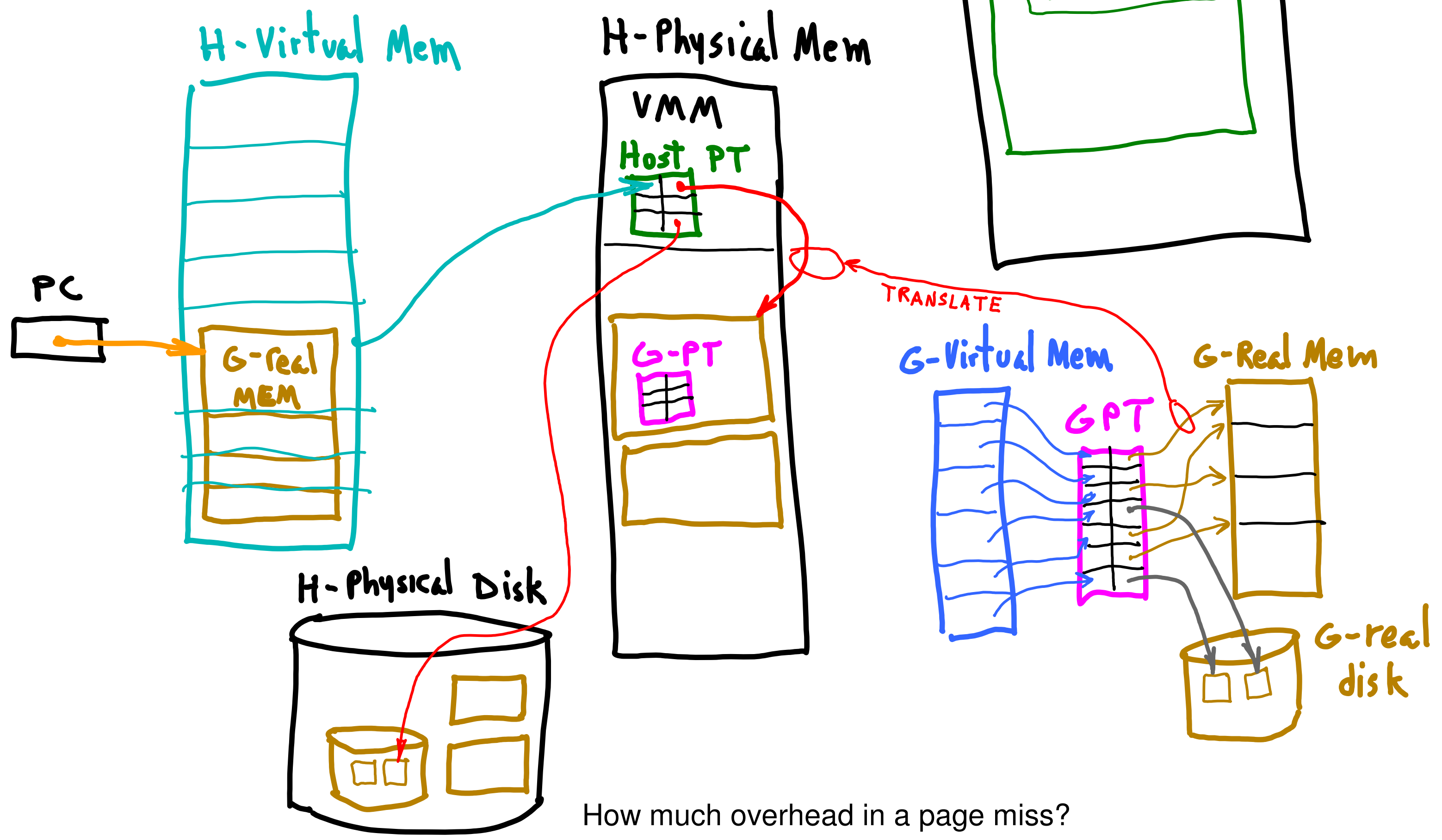
Virtual Memory & VMM

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

Adjust physical frame number
  --- use shadow page table

OR

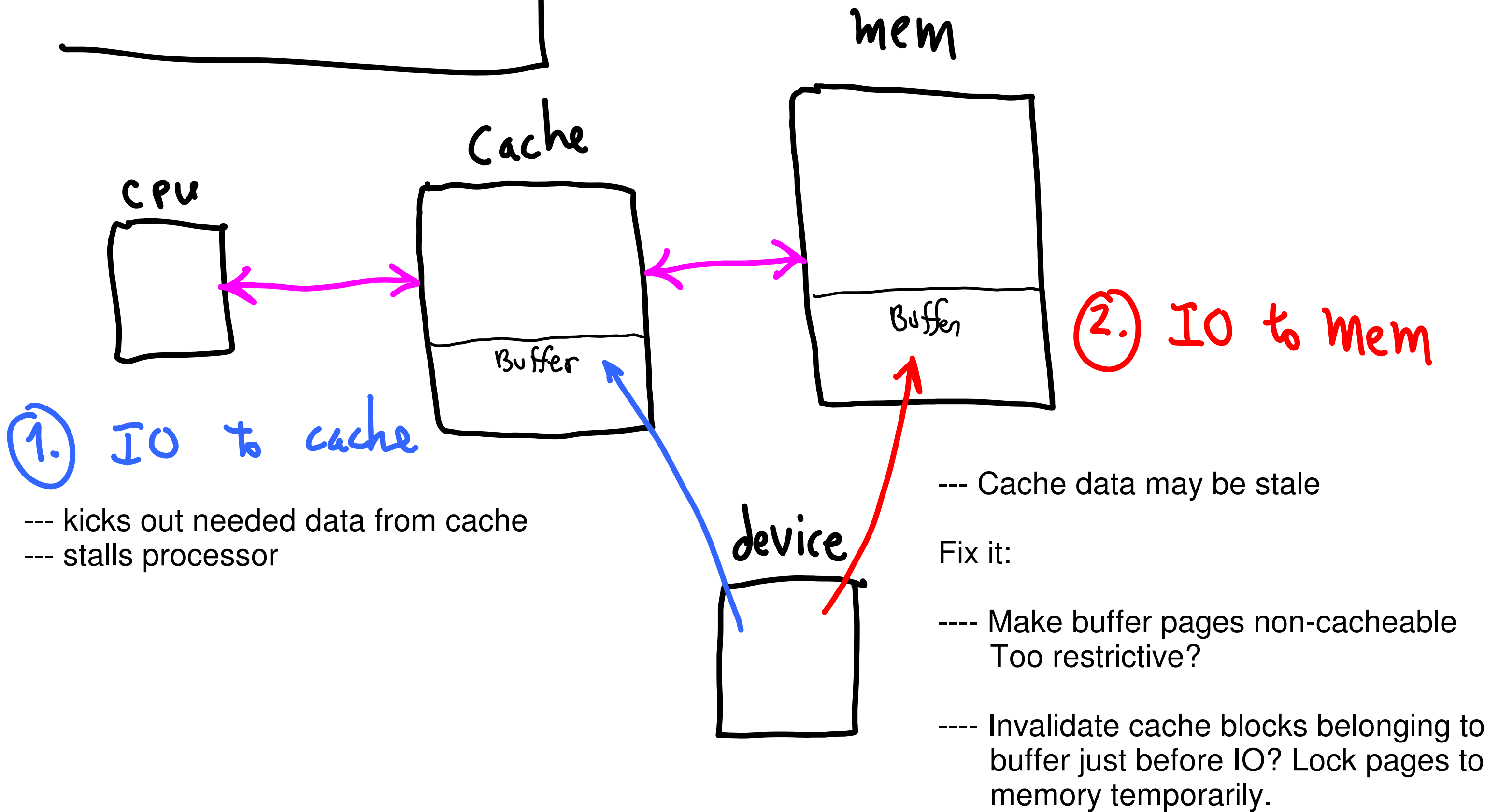
Add HW second layer translation
(1960's IBM 370 solution)



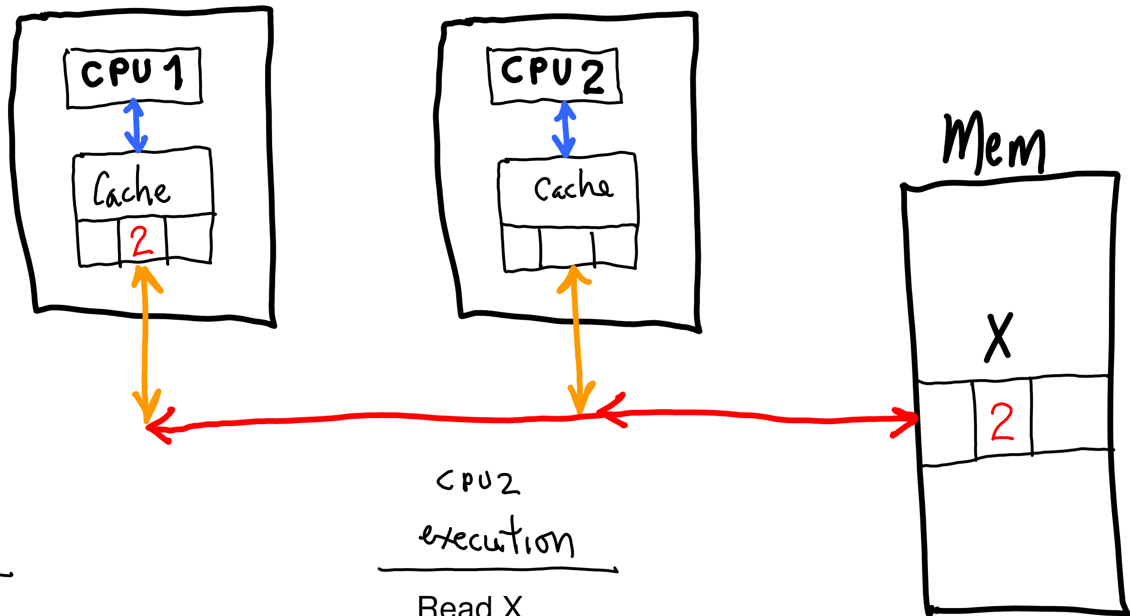
How much overhead in a page miss?

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
mem
Cache
CPU
Buffer
Buffer
2. IO to mem
1. IO to cache
device
--- Cache data may be stale
--- kicks out needed data from cache
--- stalls processor
Fix it:
---- Make buffer pages non-cacheable
Too restrictive?
---- Invalidate cache blocks belonging to buffer just before IO? Lock pages to 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	CPU-2
cache <== 2	
cache <== 7	
X <== 7	
	cache <== 7
	cache <== 10
	X <== 10

This OK.

Reverse is OK.

Interleaved?

what's supposed to happen?