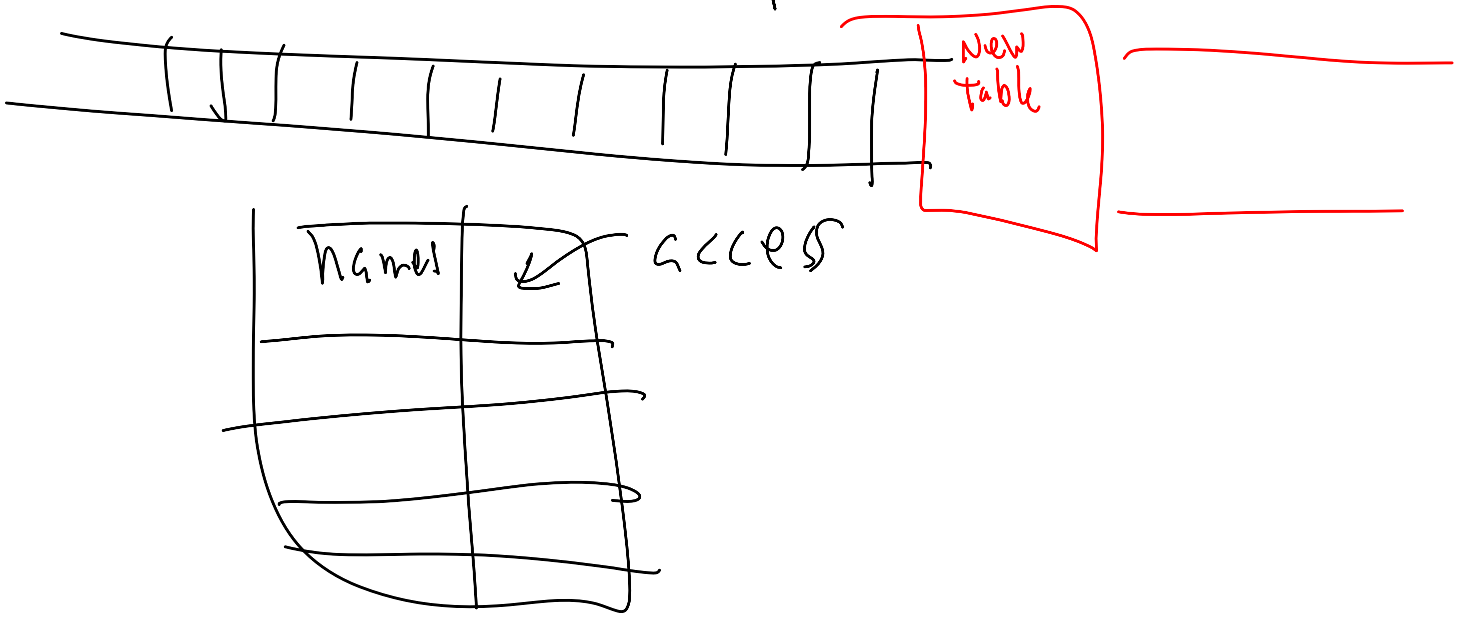
TM Tape

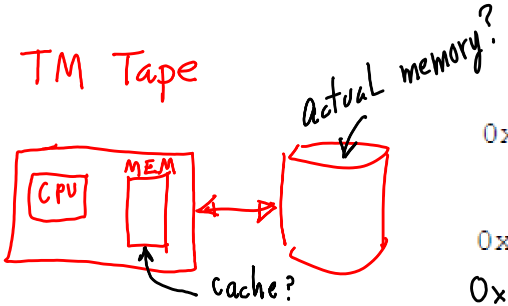


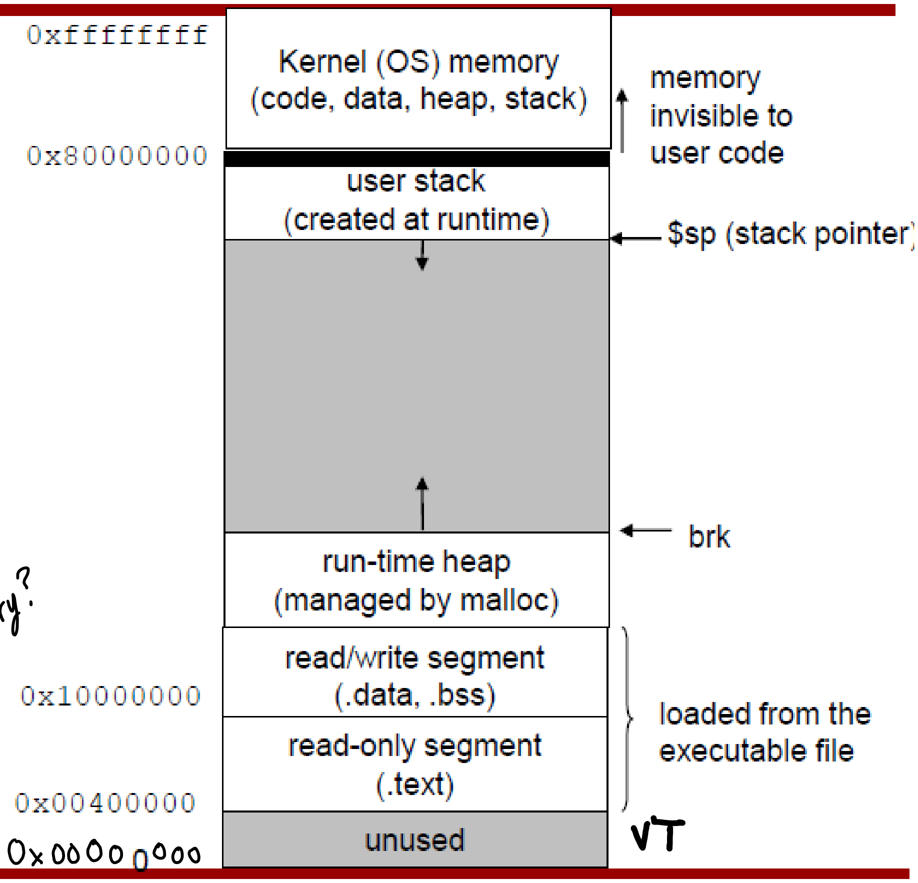
Virtual Memory

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





0xffffffff

Kernel (OS) memory (code, data, heap, stack)

memory invisible to user code

0x80000000

user stack (created at runtime)

$sp (stack pointer)

brk

run-time heap (managed by malloc)

0x10000000

read/write segment (.data, .bss)

read-only segment (.text)

loaded from the executable file

0x00400000

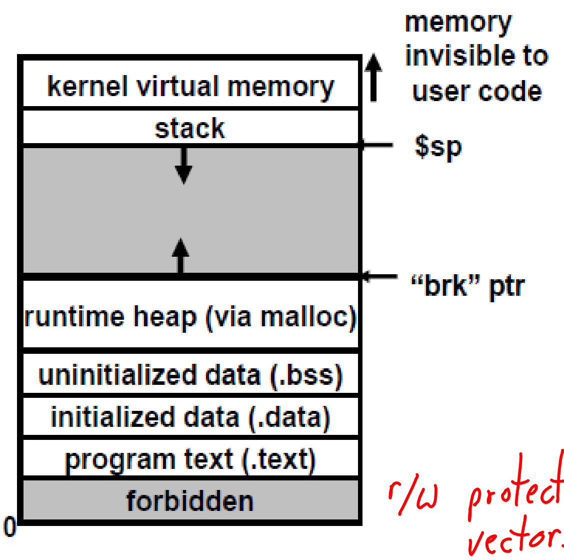
unused

0x00000000

VT

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

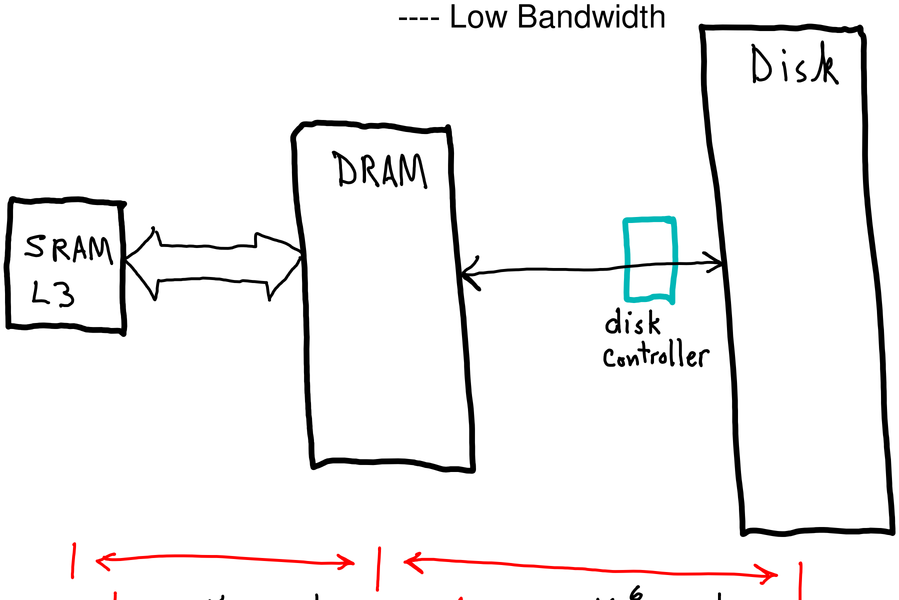


r/w protected vectors, ...

- 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

$$T_{avg} = T_{hit} + \underbrace{(\text{miss rate})}_{\text{minimize}} \underbrace{T_{penalty}}_{\text{huge}}$$

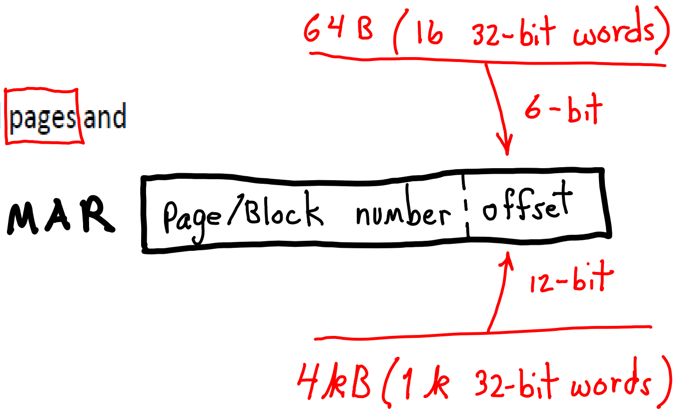
Big blocks	spatial locality
Big cache	lower miss rate
Associative	lower miss rate
Write back	less bandwidth
Multiple levels	lower avg penalty

disk controller

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

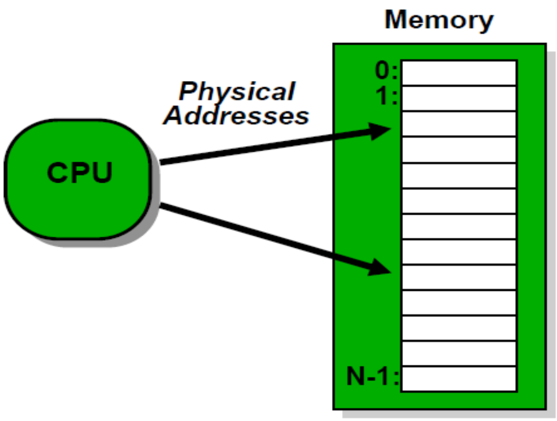
Just like cache blocks
But, much bigger offset

- 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



A System with Physical Memory Only

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

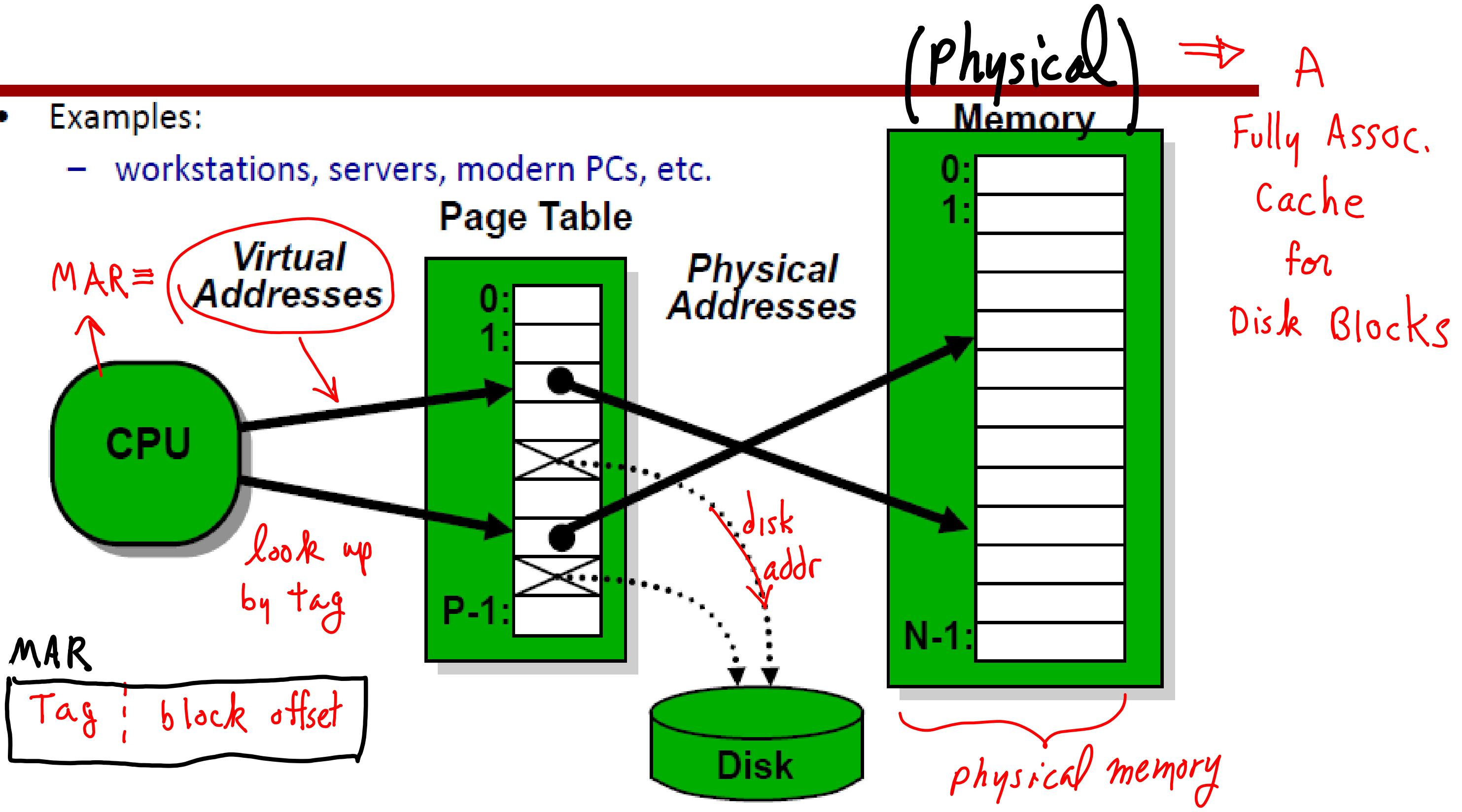


Addresses generated by the CPU point directly to bytes in physical memory

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"?

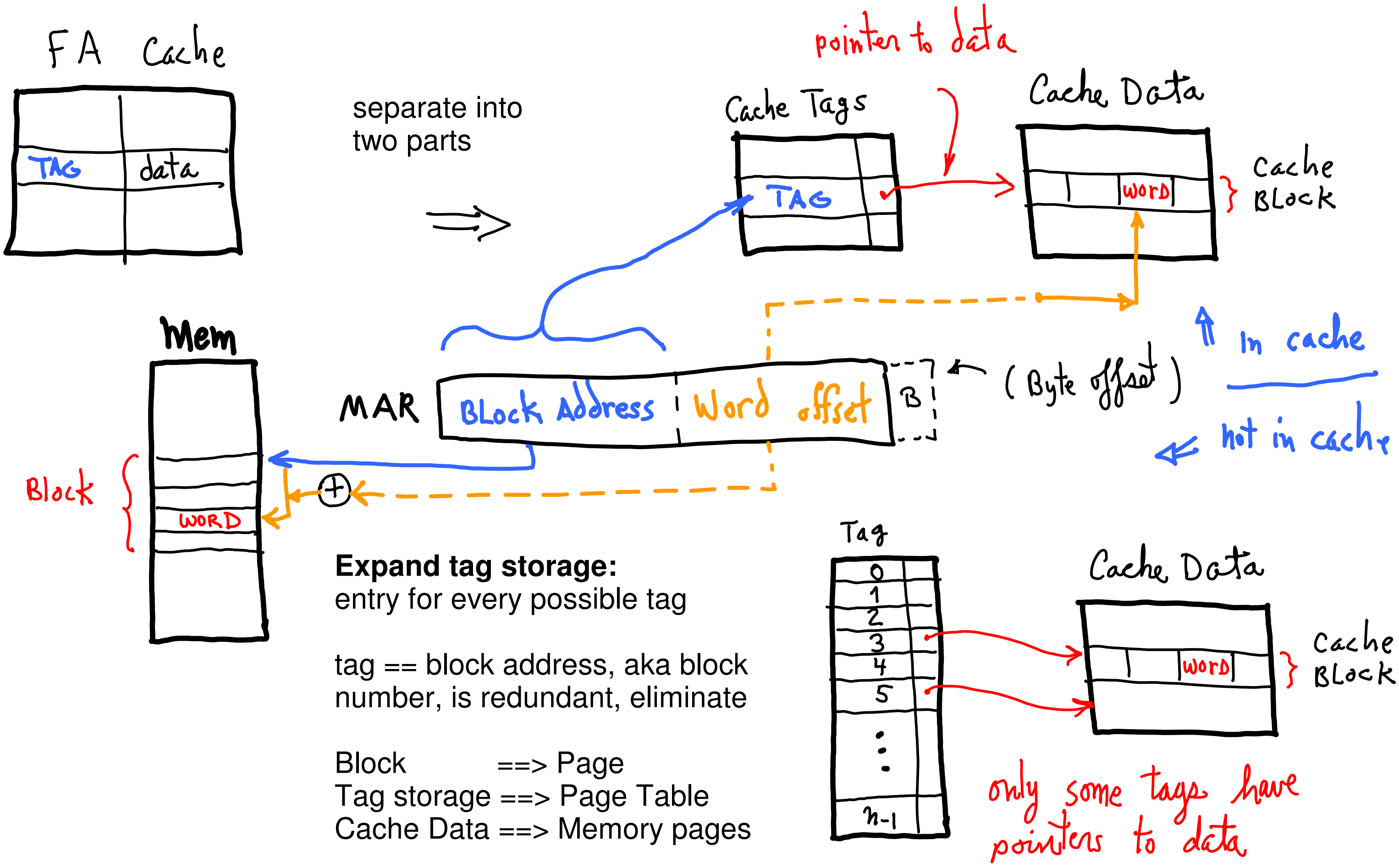
A System with Virtual Memory

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



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)



separate into
two parts

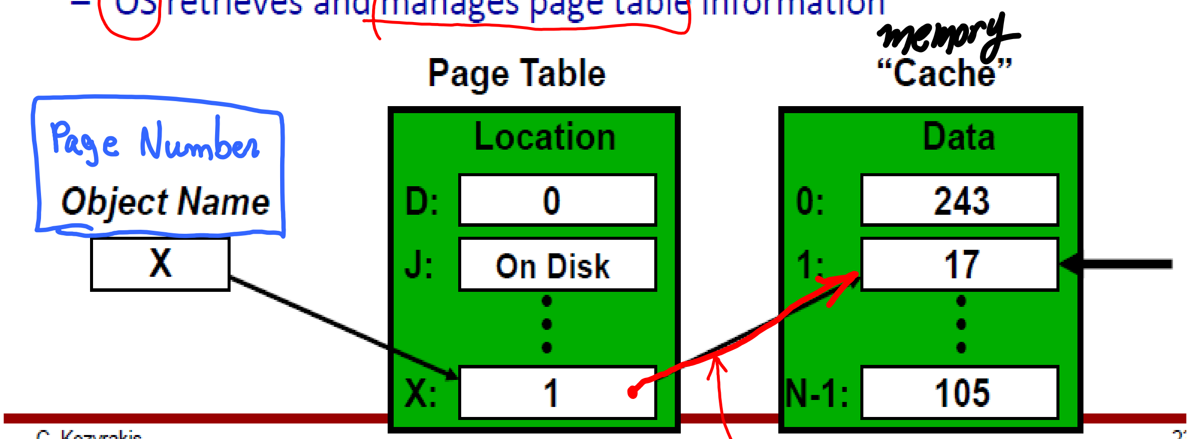
Expand tag storage:
entry for every possible tag

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

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

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 Table

	Location
D:	0
J:	On Disk
X:	1

memory “Cache”

	Data
0:	243
1:	17
N-1:	105

data block is large “page” in physical “frame”

data pointer is physical memory address of “physical page frame”

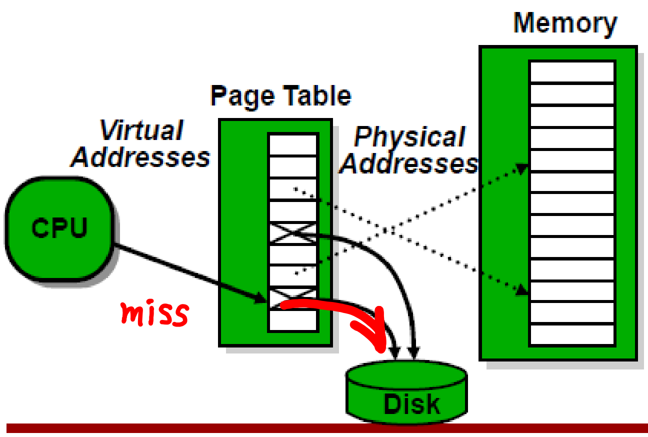
Page Faults (Similar to “Cache Misses”)

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

Valid bit + disk address

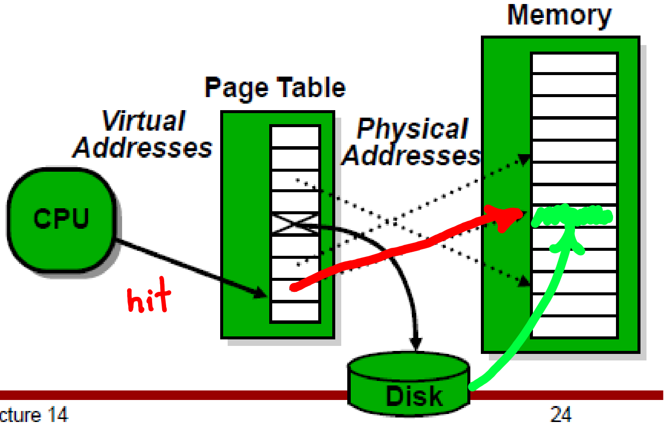
any memory “frame” holds any “page”

Before fault



After fault

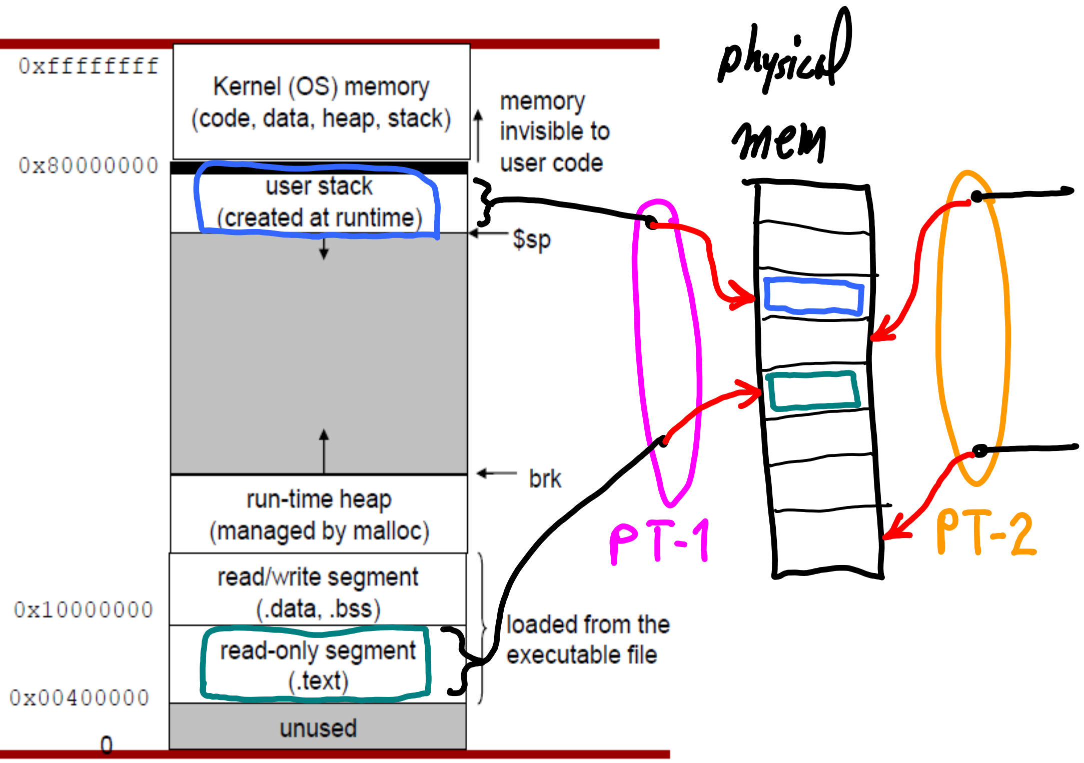
(restart instruction)



Page is in frame

Does VM Satisfy Original Motivations?

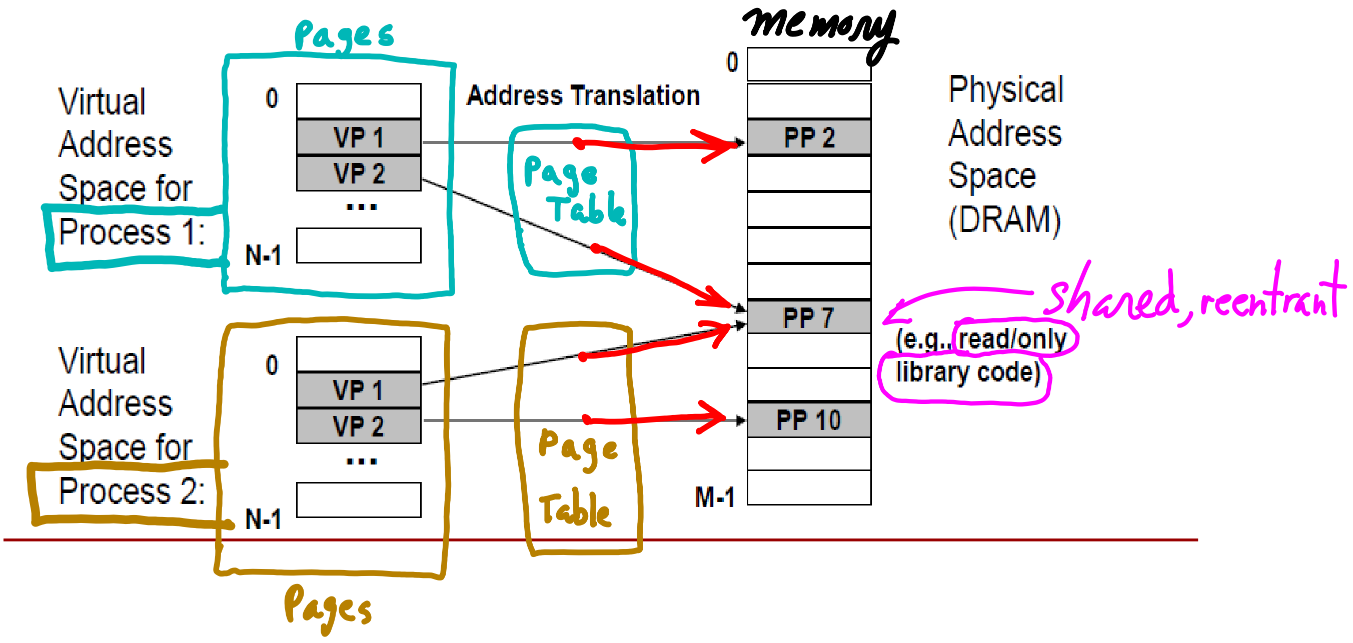
- Multiple active programs can share physical address space
- Address conflicts are resolved
  - All programs think their code is at 0x400000…
- Data from different programs can be protected how?
- Programs can share data or code when desired how?



0xffffffff
Kernel (OS) memory (code, data, heap, stack)
memory invisible to user code
0x80000000
user stack (created at runtime)
$sp
brk
run-time heap (managed by malloc)
read/write segment (.data, .bss)
0x10000000
loaded from the executable file
read-only segment (.text)
0x00400000
unused
0
physical mem
PT-1
PT-2

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



Pages
memory
Virtual Address Space for Process 1:
0
VP 1
VP 2
…
N-1
Address Translation
Page Table
0
PP 2
PP 7
PP 10
M-1
Physical Address Space (DRAM)
shared, reentrant
(e.g., read/only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
…
N-1
Page Table
Pages

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

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

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

--- 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?

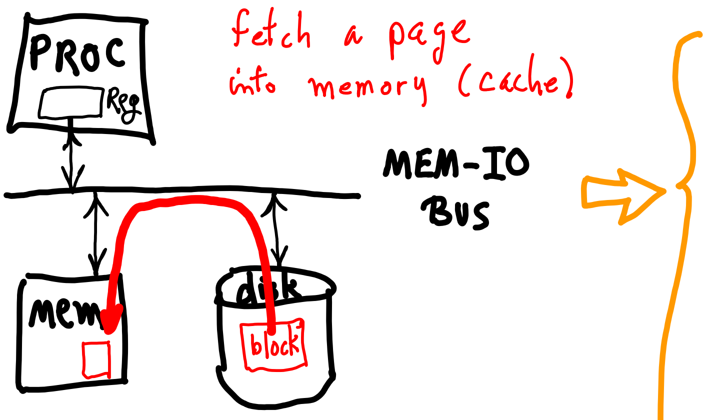
--- It's nice to **share** memory, but **why bother**?

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

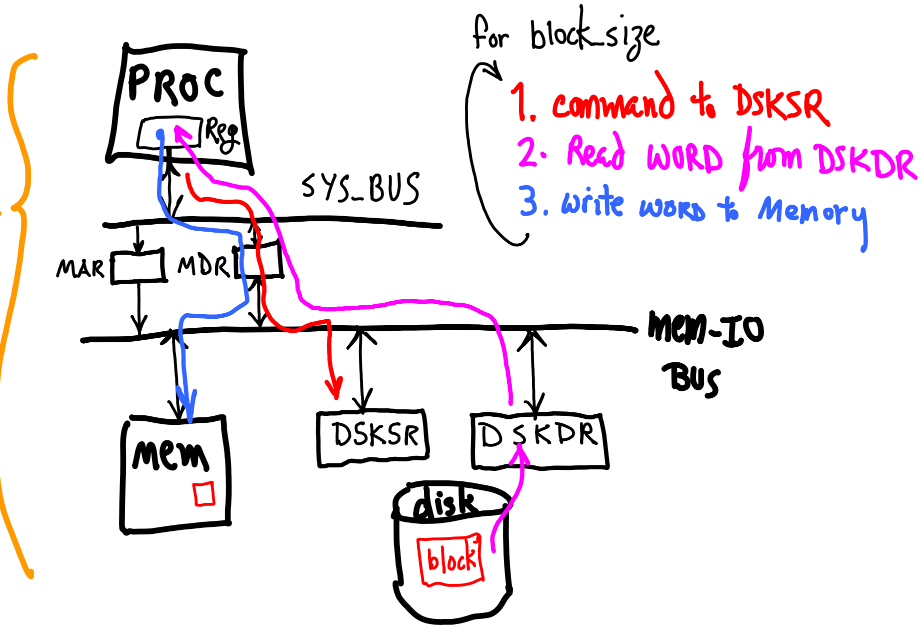
--- What about I/O?

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

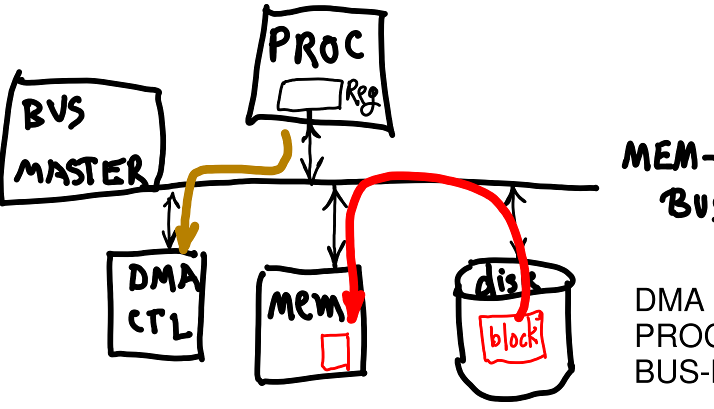
--- Speaking of I/O, what about long, slow **I/O for disk blocks (pages)?**



fetch a page
into memory (cache)



for block_size
1. command to DSKSR
2. Read WORD from DSKDR
3. write WORD to Memory



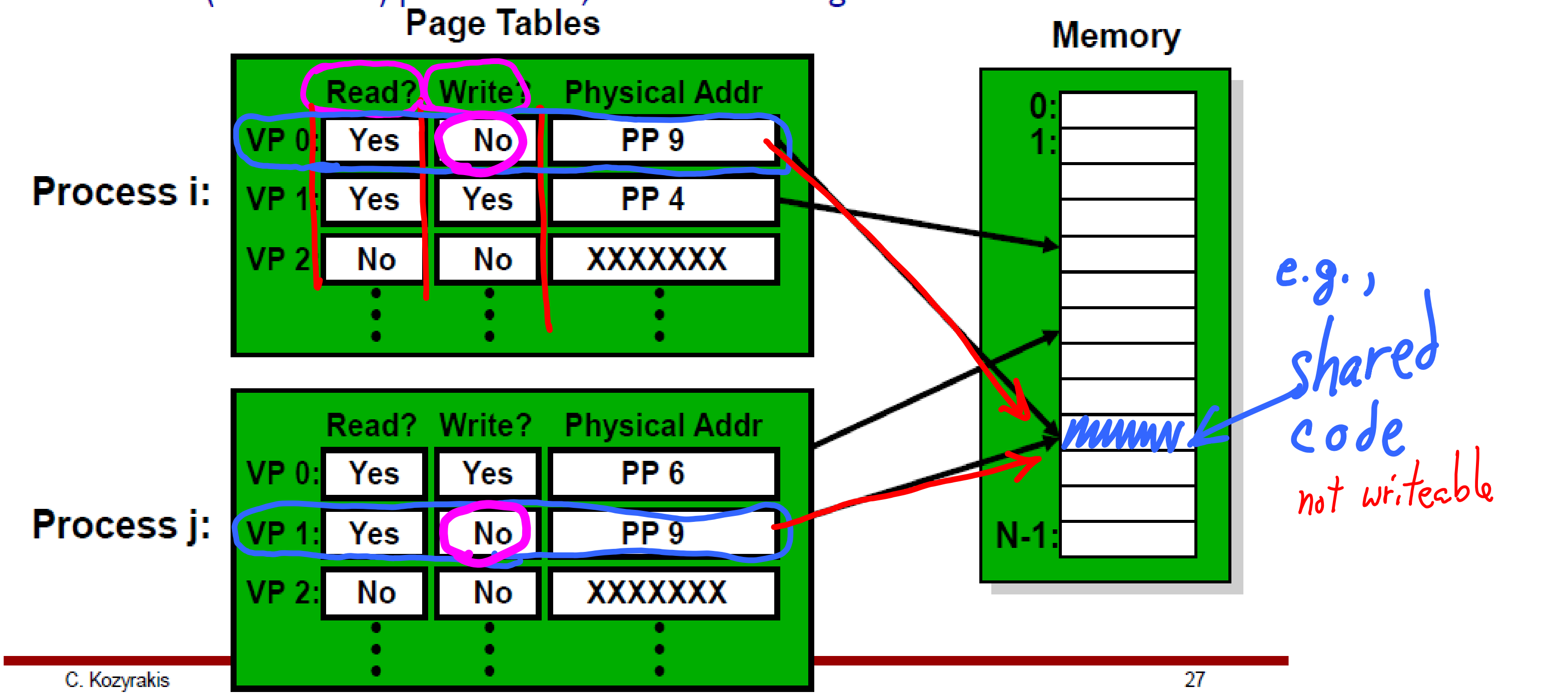
1. command + count + addresses to DMA ctl
2. DMA ctl processor does as shown above

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



Process i:

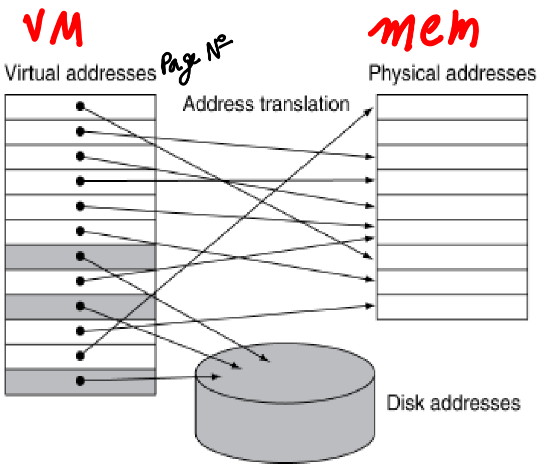
	Read?	Write?	Physical Addr
VP 0:	Yes	No	PP 9
VP 1:	Yes	Yes	PP 4
VP 2:	No	No	XXXXXXX
	⋮	⋮	⋮

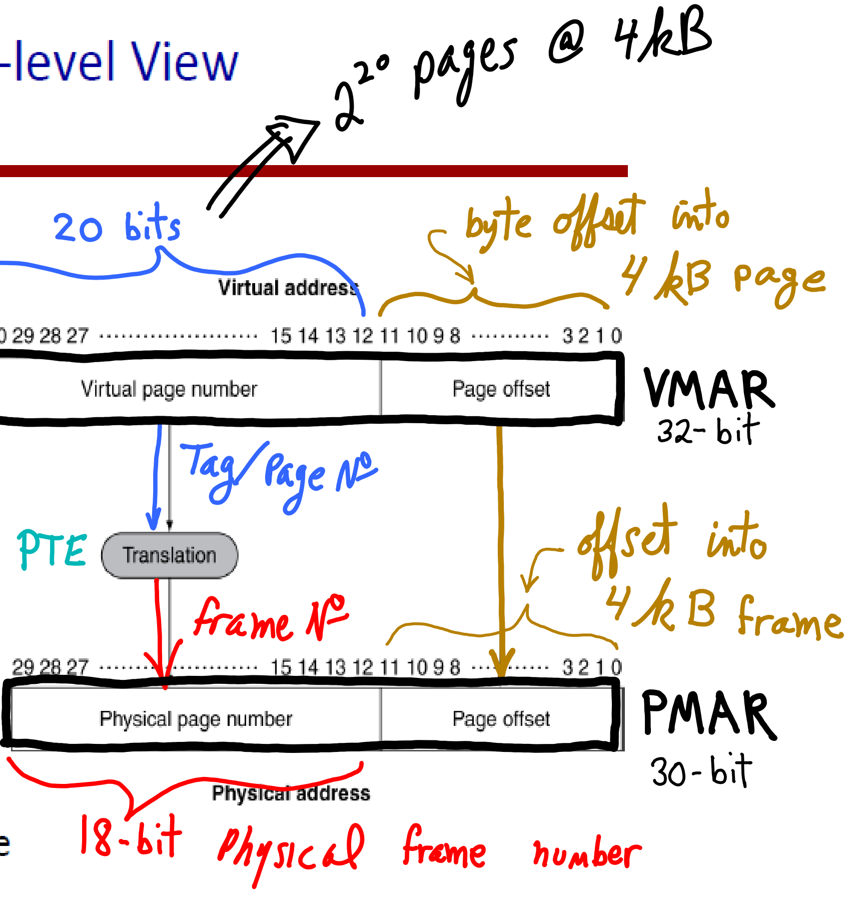
Process j:

	Read?	Write?	Physical Addr
VP 0:	Yes	Yes	PP 6
VP 1:	Yes	No	PP 9
VP 2:	No	No	XXXXXXX
	⋮	⋮	⋮

Translation: High-level View

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

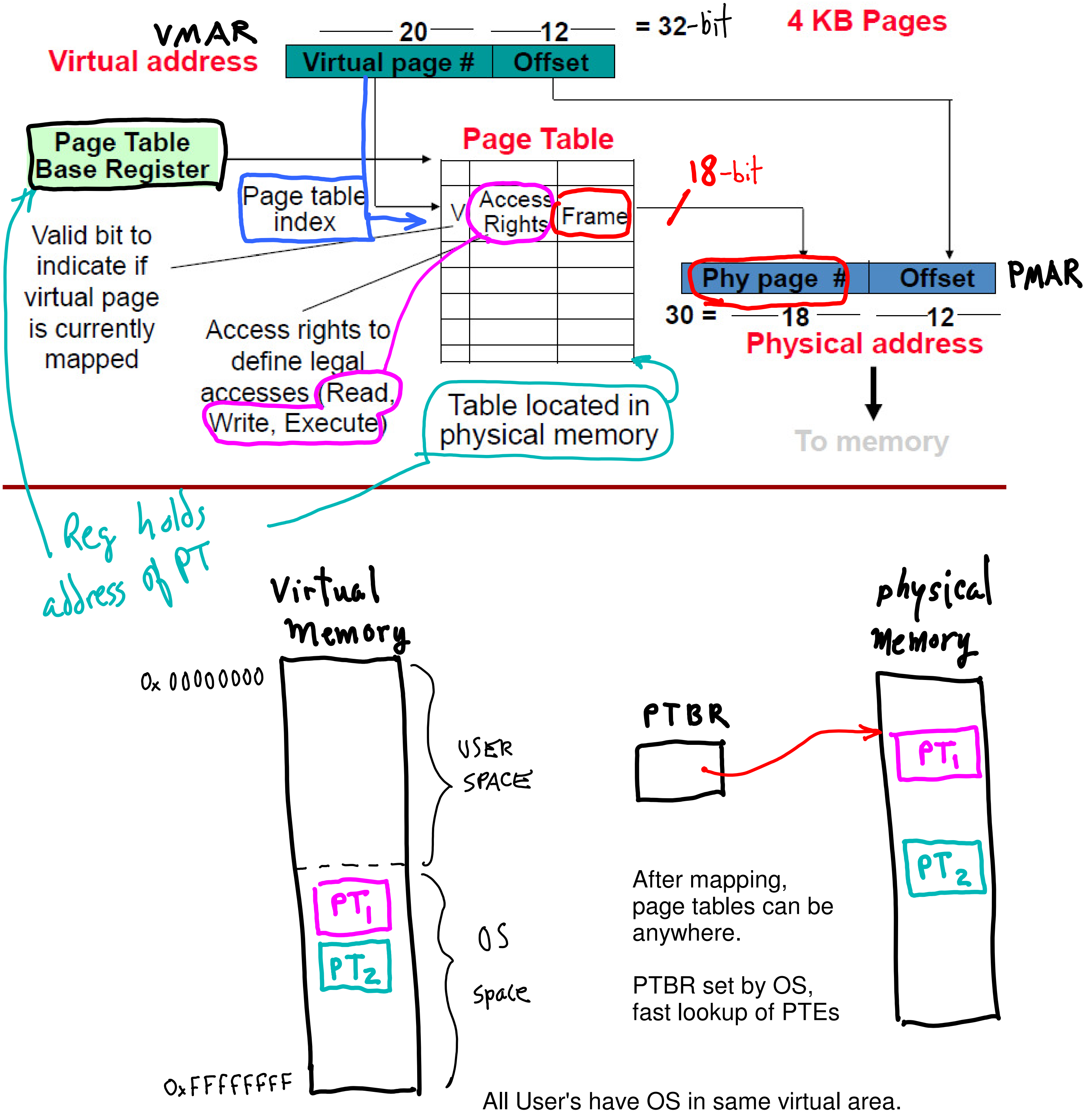




- Physical page sometimes called a frame

⇒ 2^{18} frames @ 4kB

Translation: Process



Valid bit to indicate if virtual page is currently mapped

Access rights to define legal accesses (Read, Write, Execute)

Table located in physical memory

After mapping, page tables can be anywhere.

PTBR set by OS, fast lookup of PTEs

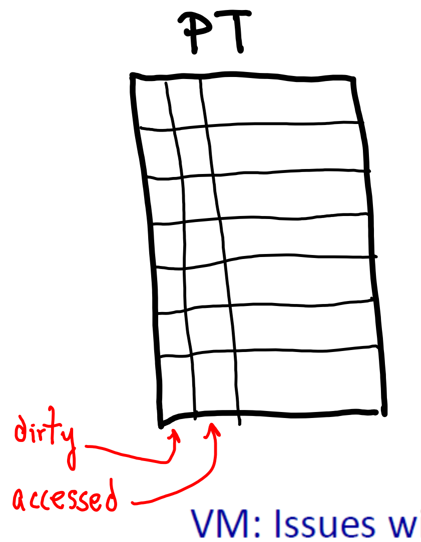
All User's have OS in same virtual area.

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.

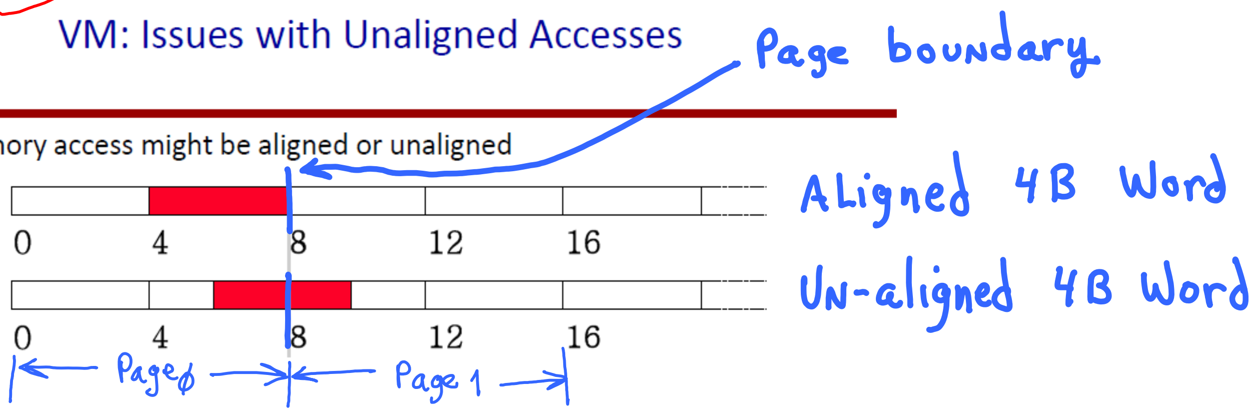
Set accessed bit on read or write.

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

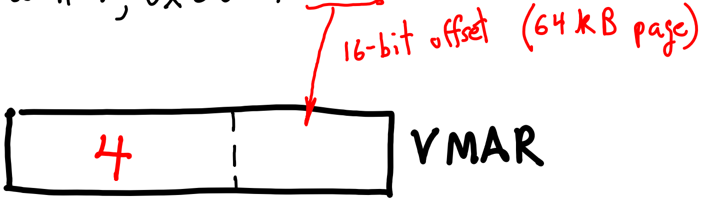
- Memory access might be aligned or unaligned

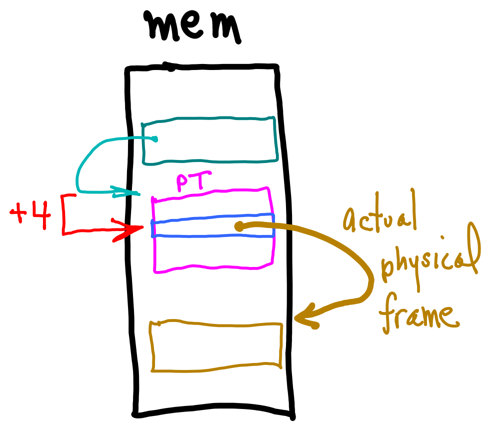


- What happens if unaligned address access straddles a page boundary?
  - What if one page is present and the other is not?
  - Or, what if neither is present?
- MIPS architecture disallows unaligned memory access
- Interesting legacy problem on 80x86 which does support unaligned access

How many memory references

lw $7, 0x00040000





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.)

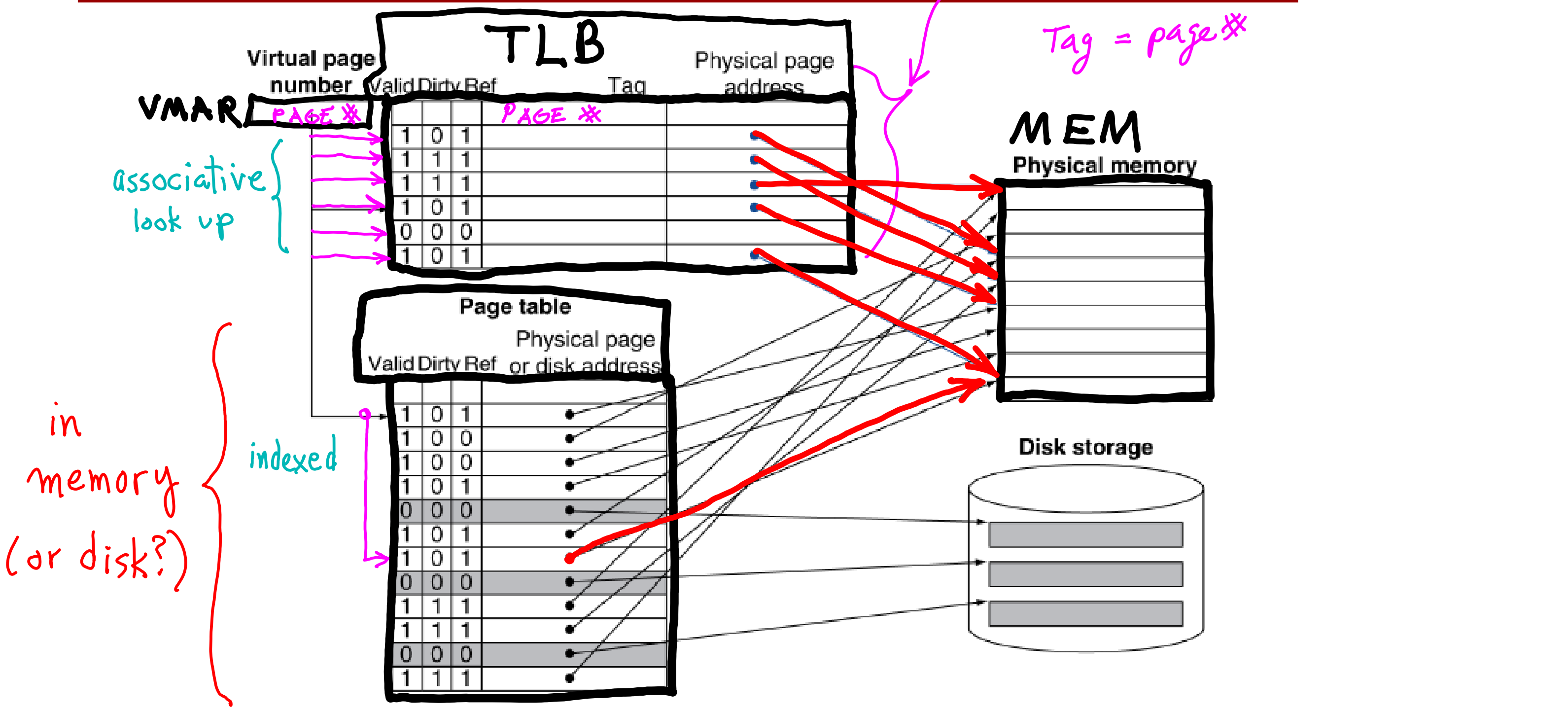
Speed it up:
1. PTBR <== **Page-table-location-pointer**
Do this once at program startup

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)

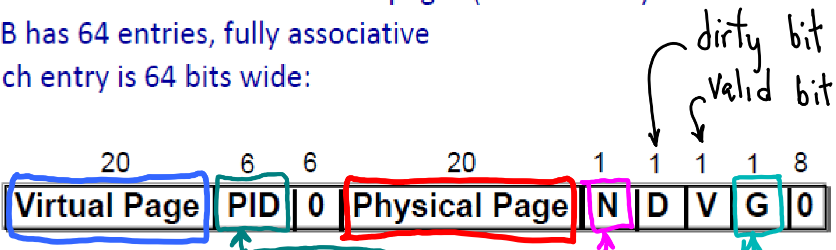
- The tags for a TLB entry
  - Virtual page number
    - Portion of it not used for indexing into the TLB { N-way Set Associative
  - Valid bit
  - LRU bits
    - If TLB is associative and LRU replacement is used

} 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:



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

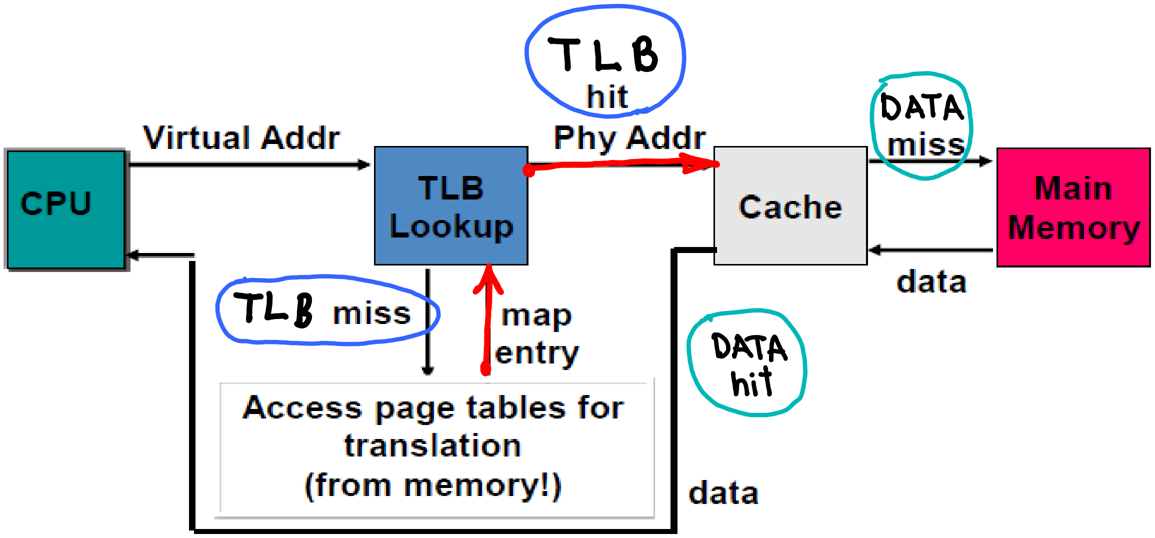
Read PT, get PTE

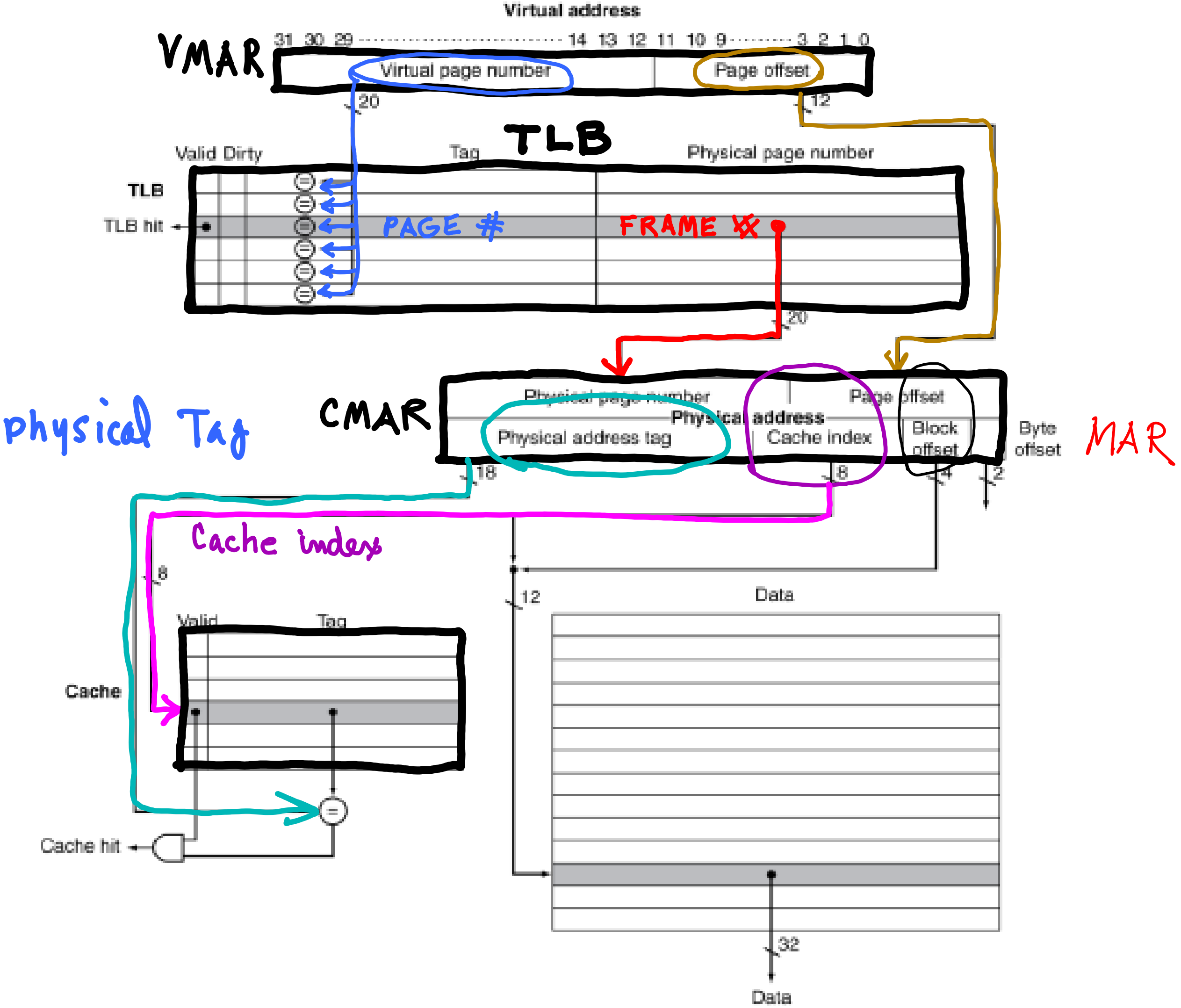
- If page is in memory
  - Load the PTE to TLB and retry instruction
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler
    - This is what MIPS does using a special vectored interrupt

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

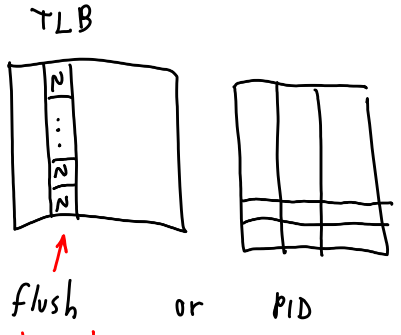
TLB & Memory Hierarchies

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





TLB Caveats



write-back?

- 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

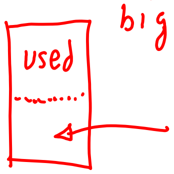
fully-assoc.

⇒ 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

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, 64MB! ?

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

OS sets MMU

- 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

2^N → N-bit address

2^m → 2^m B 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 32 bits wide
    - 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!

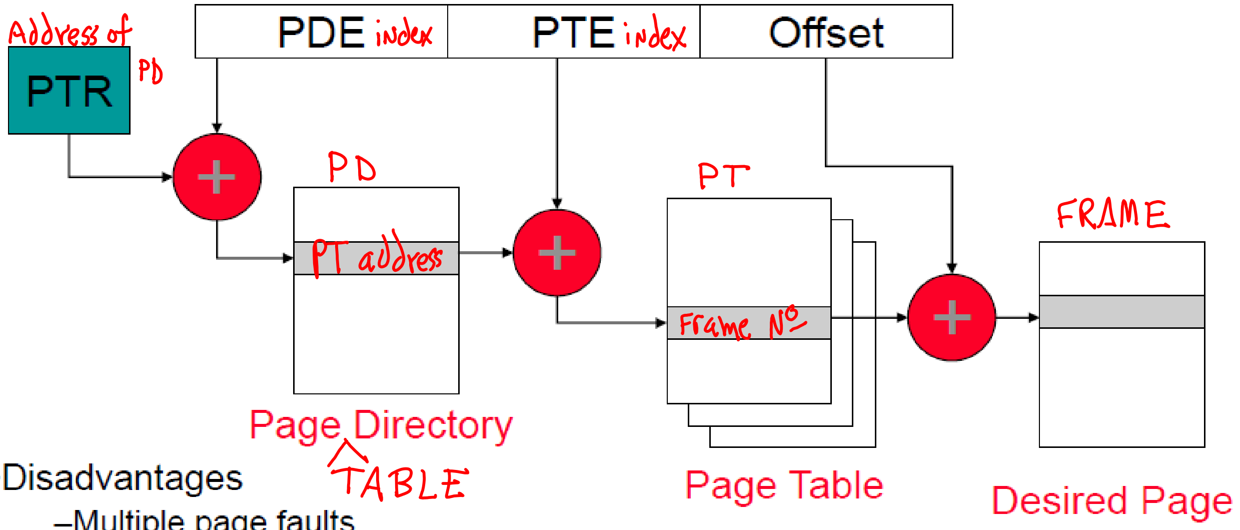
But, who uses all 2^{32} addresses?

- 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? → $2^{N-m} = 2^{64-12} = 2^{52}$ entries
  - What about running multiple programs?

$(2^{32} = 4G)$
$\times (2^{20} = 1M)$!?!

Solution: Multi-Level Page Tables

- Use a hierarchical page table structure
  - Two levels are typically sufficient ? 64-bit, 128-bit address space?
    - First level: directory entries
    - Second level: actual page table entries + more levels? ⇒ inverted page table
  - 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



- 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

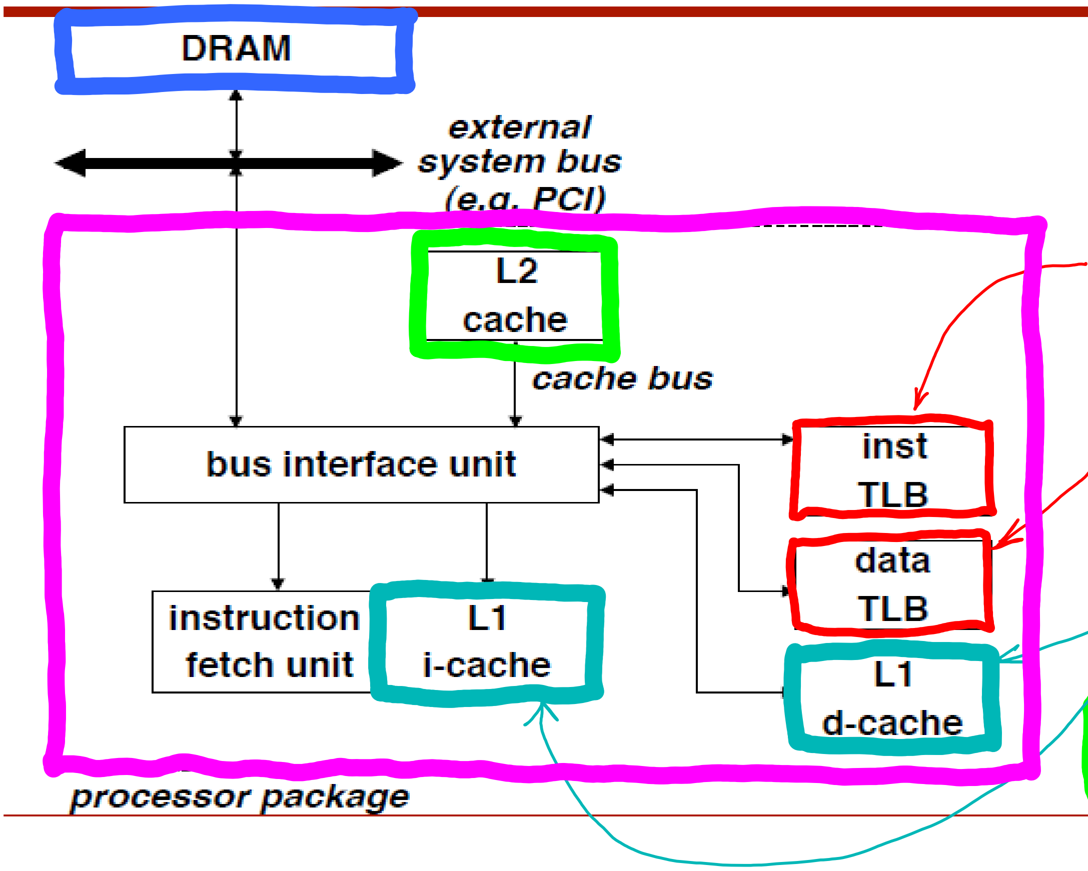
Real Example: Intel P6

- 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

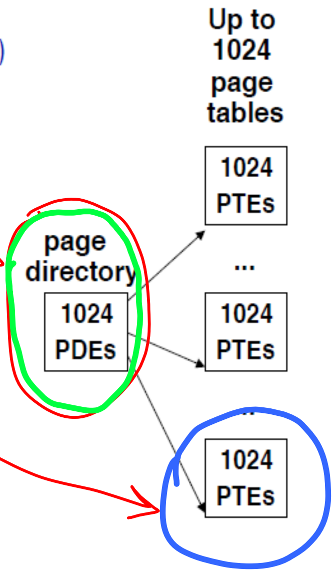


- 32 bit address space
- 4 KB page size
- L1, L2, and TLBs
  - 4-way set associative
- inst TLB
  - 32 entries
  - 8 sets , 4-way
- data TLB
  - 64 entries
  - 16 sets , 4-way
- L1 i-cache and d-cache
  - 16 KB
  - 32 B line size, 8 words
  - 128 sets, 4-way
- L2 cache
  - 128 KB – 2 MB, 4-way

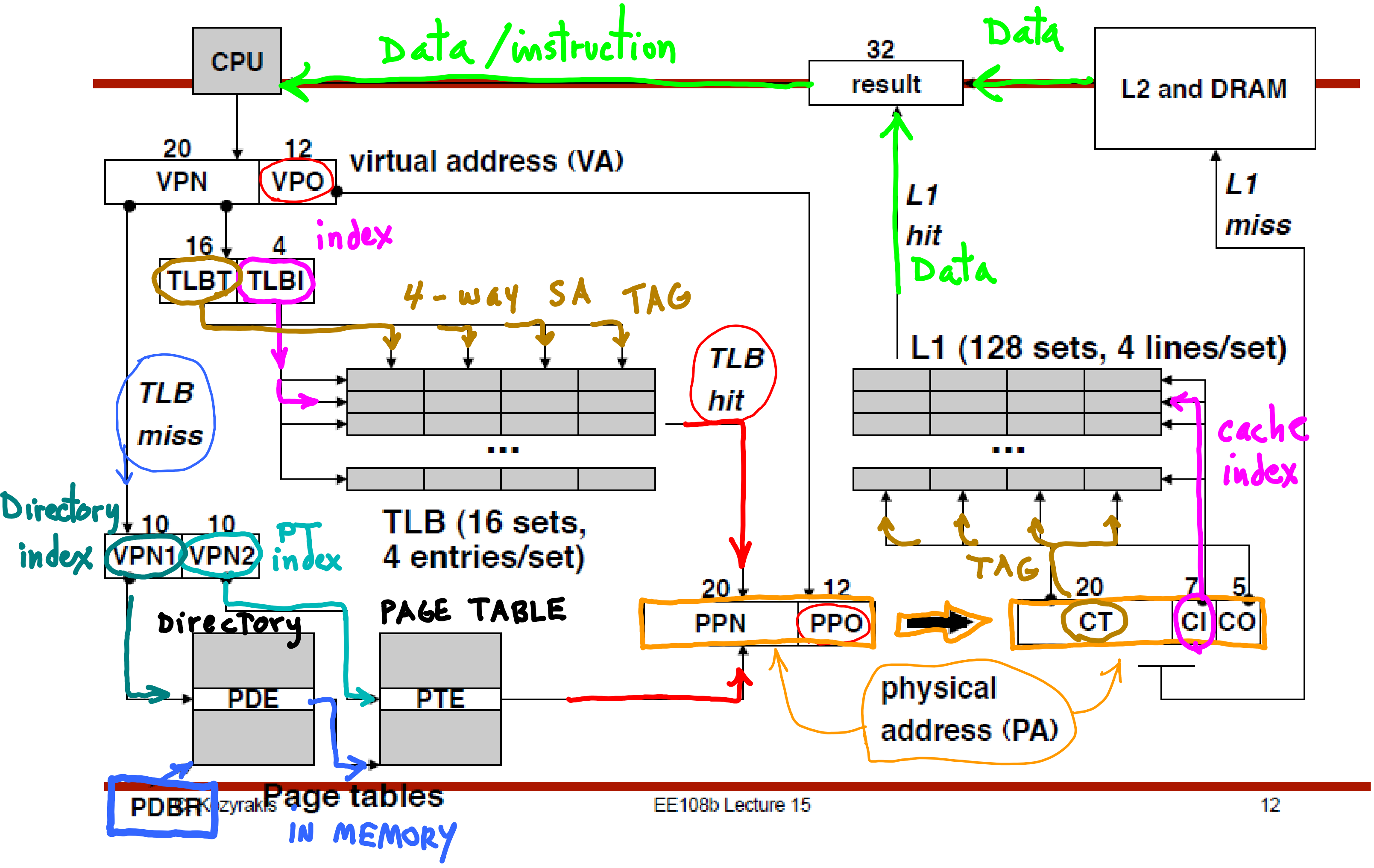
- Components of the virtual address (VA)
  - TLBI: TLB index } for set-assoc. TLB
  - TLBT: TLB tag
  - VPO: virtual page offset
  - VPN: virtual page number
- Components of the physical address (PA)
  - PPO: physical page offset (same as VPO)
  - PPN: physical page number
  - CO: byte offset within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

P6 2-level page table structure

- Page directory
  - 1024 4-byte page directory entries (PDEs) that point to page tables
  - One page directory per process.
  - Page directory must be in memory when its process is running
  - Always pointed to by PDBR
- Page tables:
  - 1024 4-byte page table entries (PTEs) that point to pages.
  - Page tables can be paged in and out.



Overview of P6 address translation



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

31 12	11 9	8	7	6	5	4	3	2	1	0
Page table physical base addr	Avail	G	PS		A	CD	WT	U/S	R/W	P=1

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)

31 1	0
Available for OS (page table location in secondary storage)	P=0

if P=0

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

31 12	11 9	8	7	6	5	4	3	2	1	0
Page physical base address	Avail	G	0	D	A	CD	WT	U/S	R/W	P=1

if P= 1

↑ Right zero extended to 32 bits

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

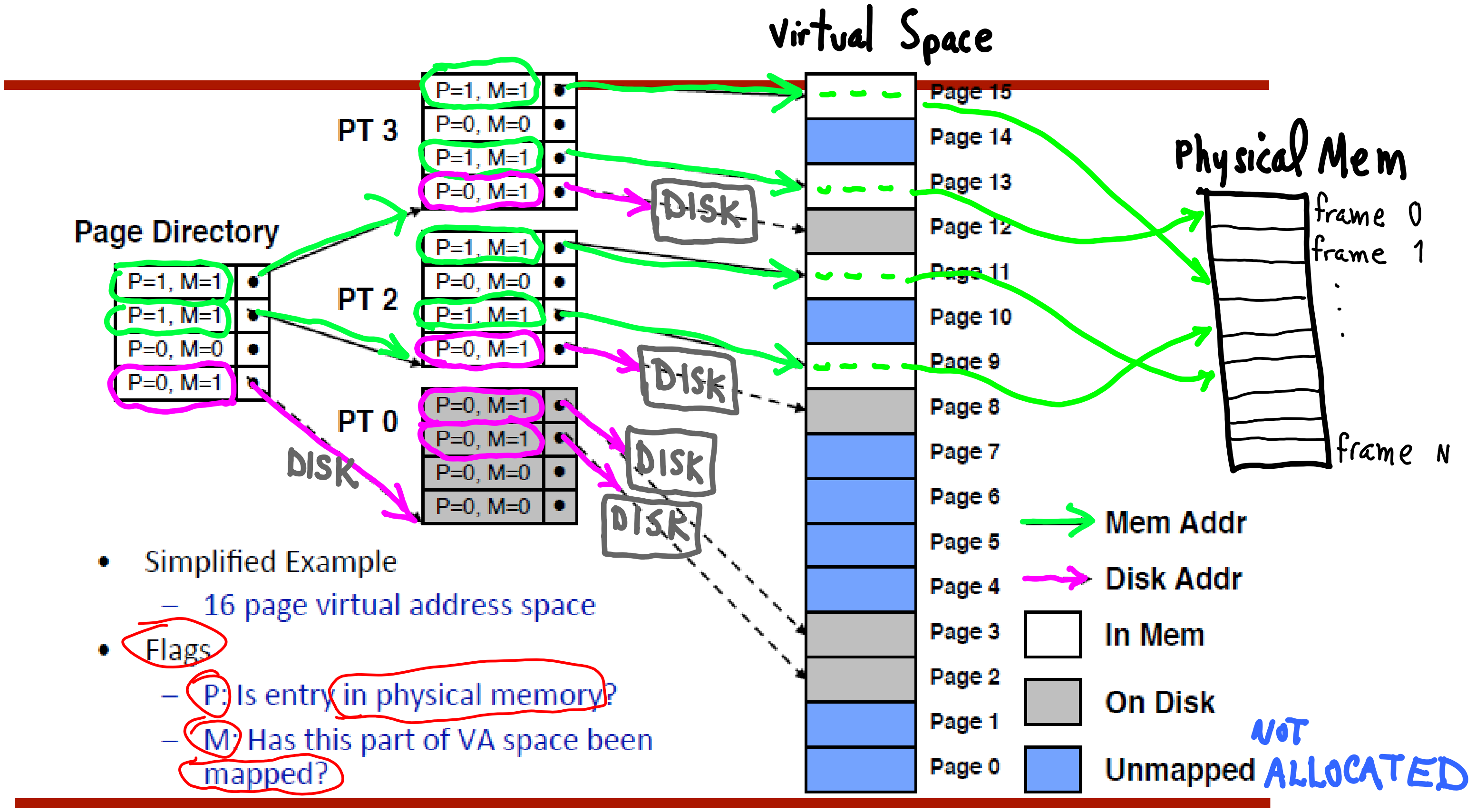
U/S: user/supervisor

R/W: read/write

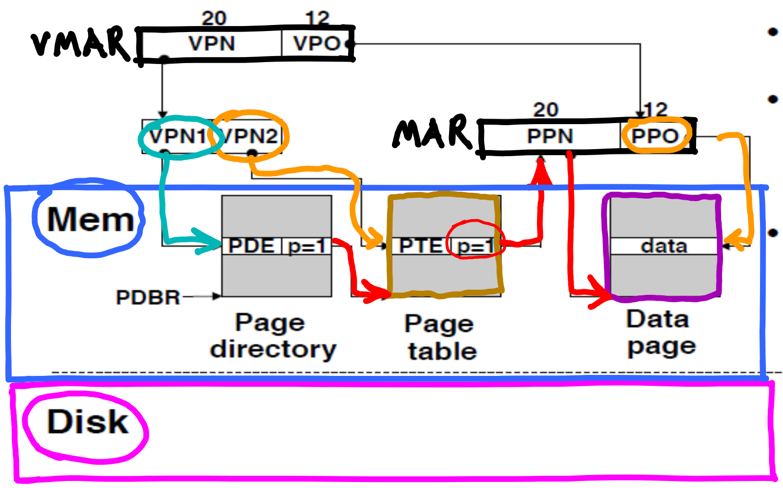
P: page is present in physical memory (1) or not (0)

31 1	0
Available for OS (page location in secondary storage)	P=0

Representation of Virtual Address Space



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

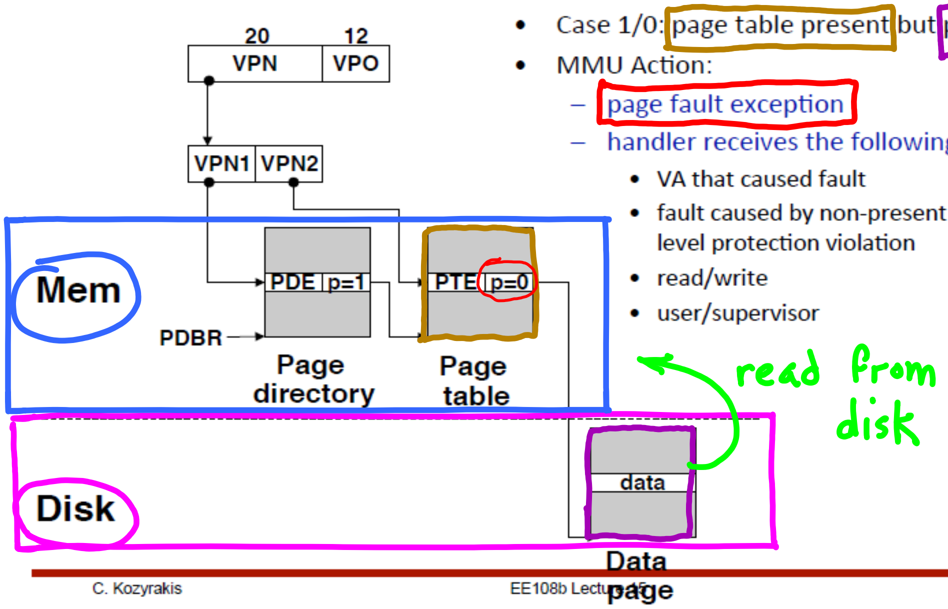


- Case 1/1: page table and page present.
- MMU Action:
  - MMU build physical address and fetch data word.
- OS action
  - none

Case

Page Table page in memory

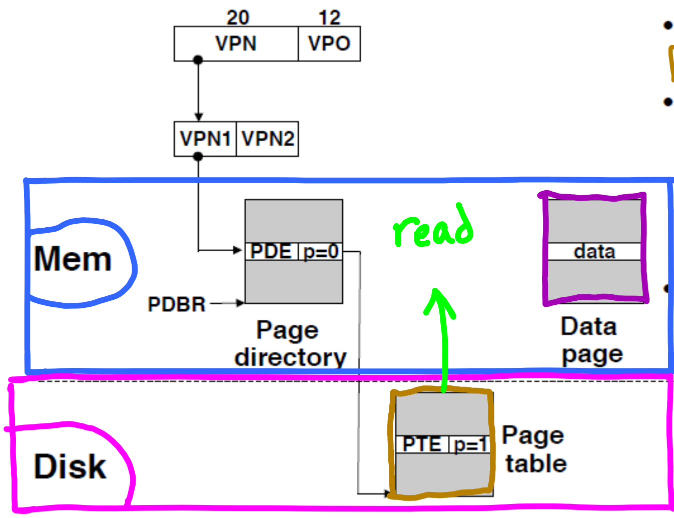
Data page in memory



data

- 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.

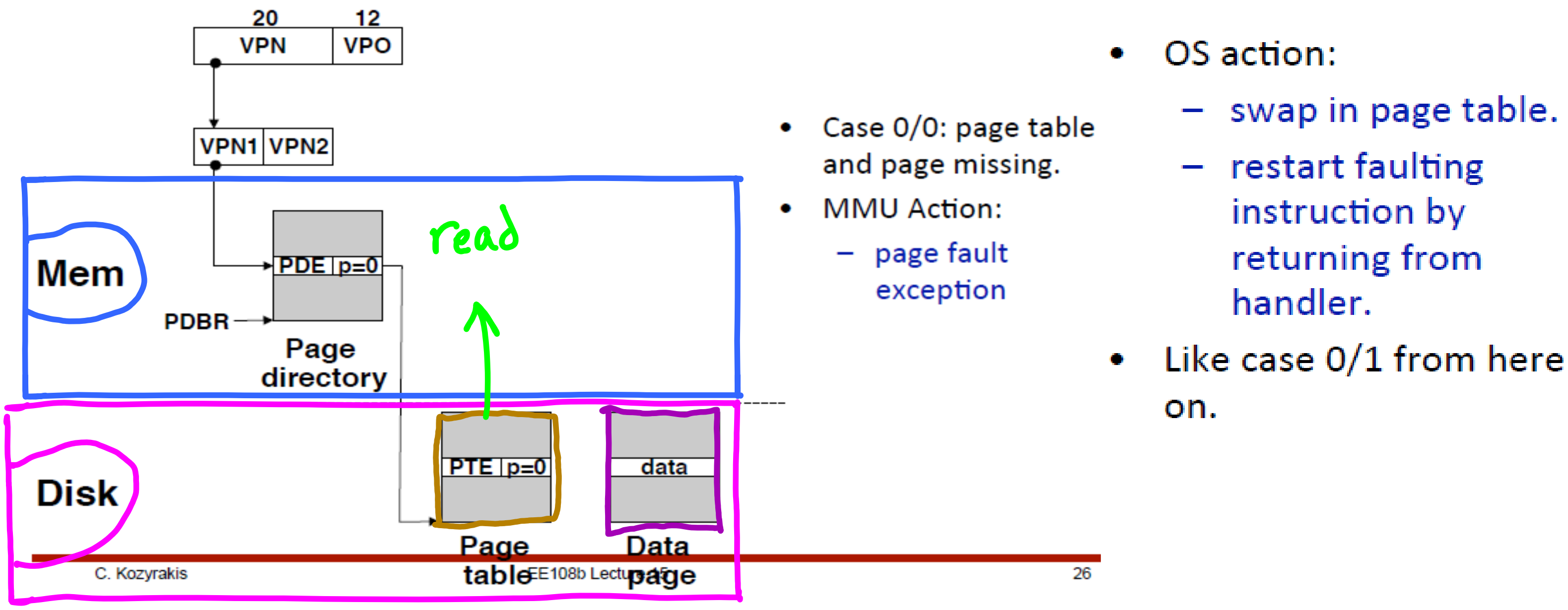


- Case 0/1: page table missing but [redacted]
- Introduces consistency issue.
  - potentially every page out requires update of disk page table.
- Linux disallows this
  - if a page table is swapped out, then swap out its data pages too.

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: becomes Case 1/1)



- 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)