# Virtual Memory

COMP273

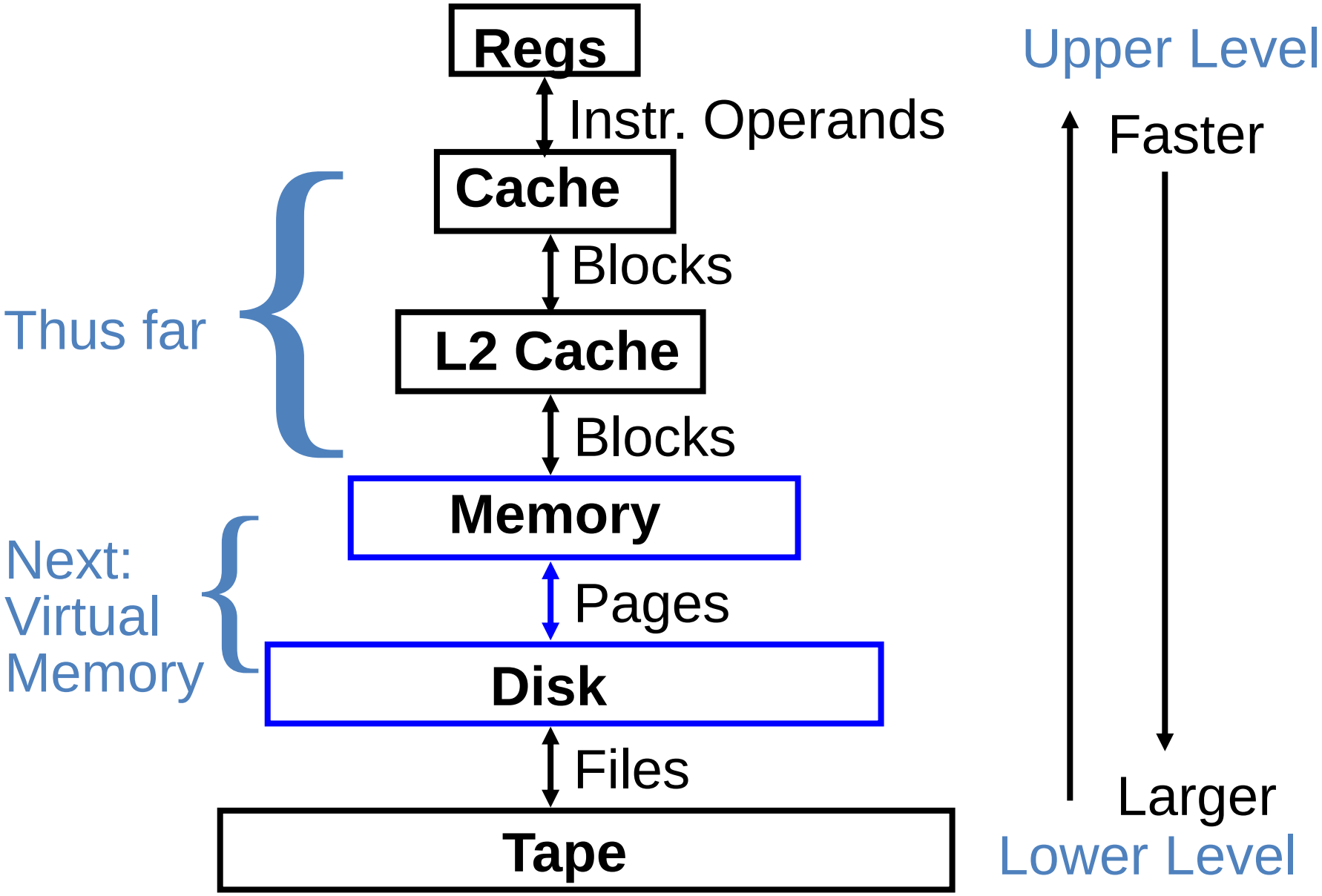
# Review (1/2)

- Caches are NOT mandatory:
  - Processor performs arithmetic
  - Memory stores data
  - Caches simply make things go *faster*
- Each level of memory hierarchy is just a subset of next lower level
- Caches speed up due to temporal locality: store data used recently
- Block size > 1 word speeds up due to spatial locality: store words adjacent to the ones used recently

# Review (2/2)

- Cache design choices:
  - size of cache: speed v. capacity
  - direct-mapped v. associative
  - for N-way set assoc: choice of N
  - block replacement policy
  - 2nd level cache?
  - Write through v. write back?
- Use performance model to pick between choices, depending on programs, technology, budget, ...

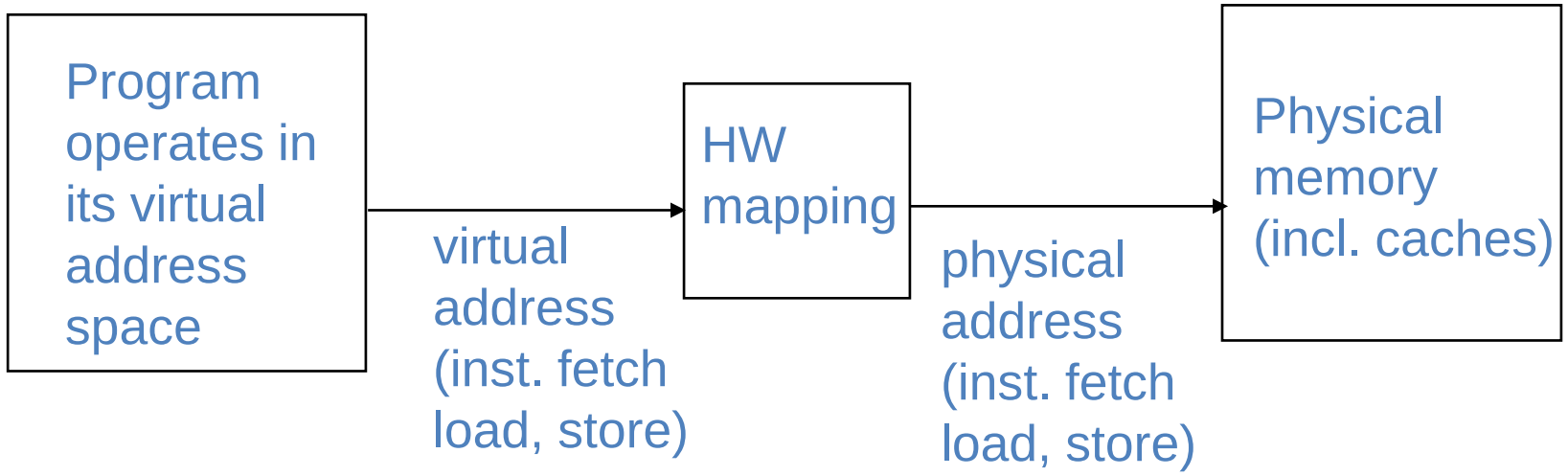
# Another View of the Memory Hierarchy



# Virtual Memory

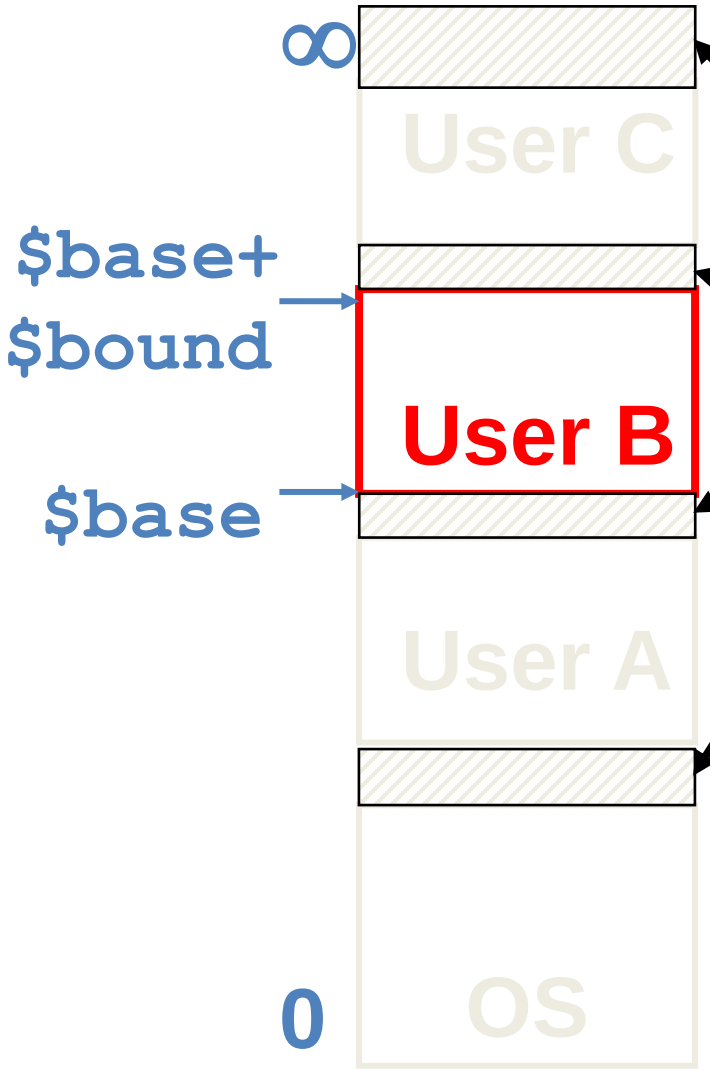
- If Principle of Locality allows caches to offer (usually) speed of cache memory with size of DRAM memory,
  then recursively why not use at next level to give speed of DRAM memory, size of Disk memory?
- Can think of memory as a cache for disk
- Called "Virtual Memory"
  - Also allows OS to share memory, protect programs from each other
  - Today, more important for protection vs. just another level of memory hierarchy
  - Isolates OS (kernel) process from a user process.
  - Historically, it predates caches

# Virtual to Physical Addr. Translation




- Each program operates in its own virtual address space;  as if it were the only program running
- Each "process" is protected from the other
- OS can decide where each goes in memory
- Hardware (HW) provides virtual -> physical mapping

# Simple Example: Base and Bound Reg



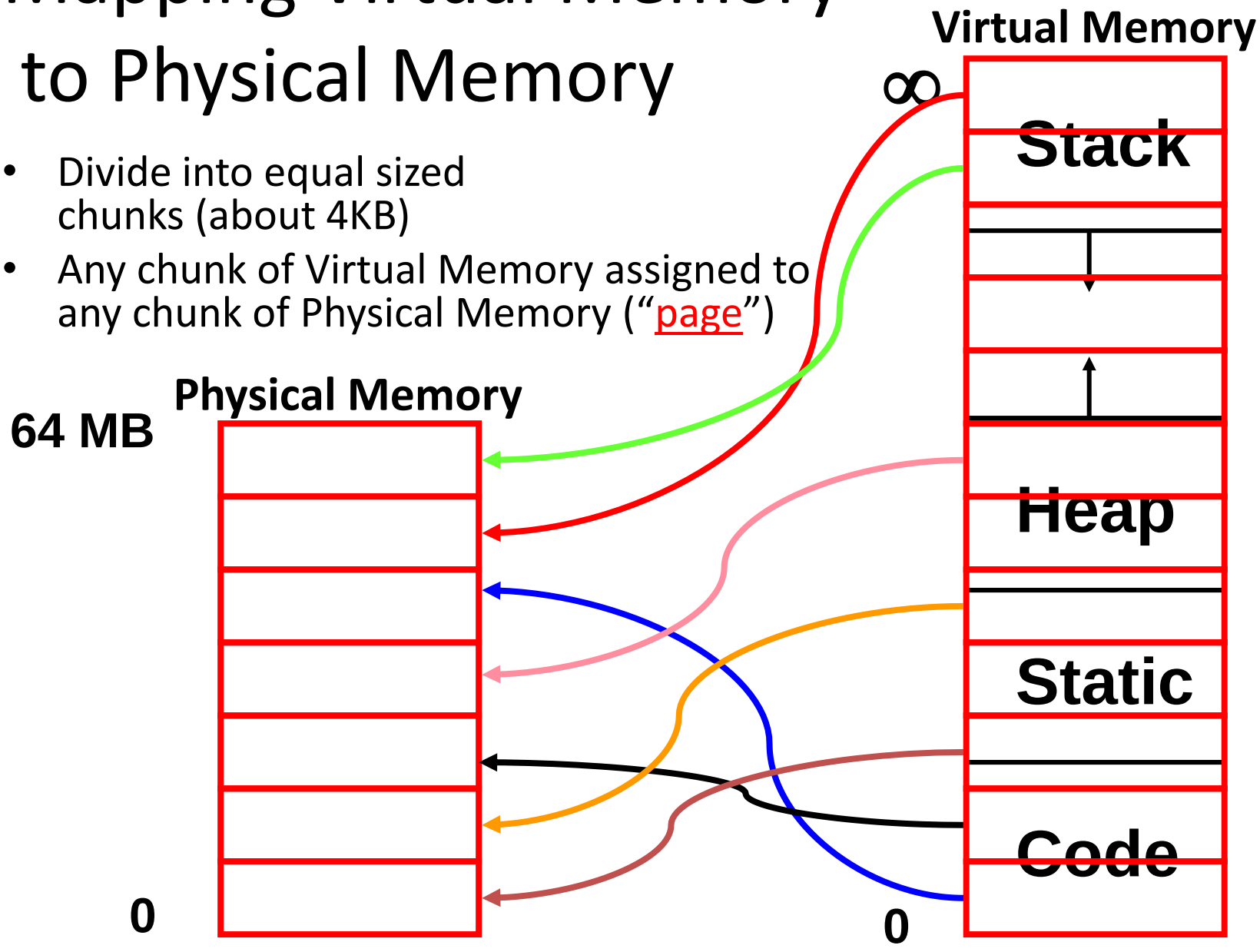

**Enough space for User D, but discontinuous ("fragmentation problem")**

- **Want discontinuous mapping**
- **Also want Process size >> memory size**
- **Base/Bound trick is not enough!**

**=> use Indirection!**

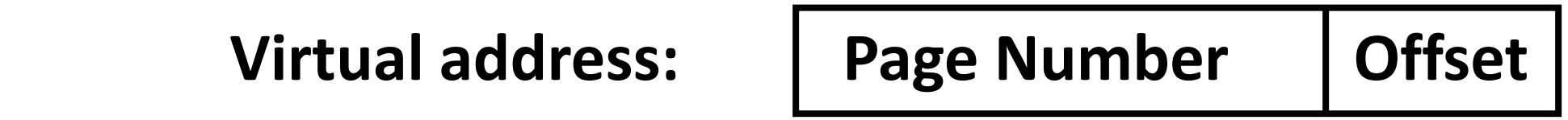
# Mapping Virtual Memory to Physical Memory

- Divide into equal sized chunks (about 4KB)
- Any chunk of Virtual Memory assigned to any chunk of Physical Memory ("page")



# Virtual Memory Mapping/translation Function

- Cannot have simple function to predict arbitrary mapping
- Use table lookup of mappings

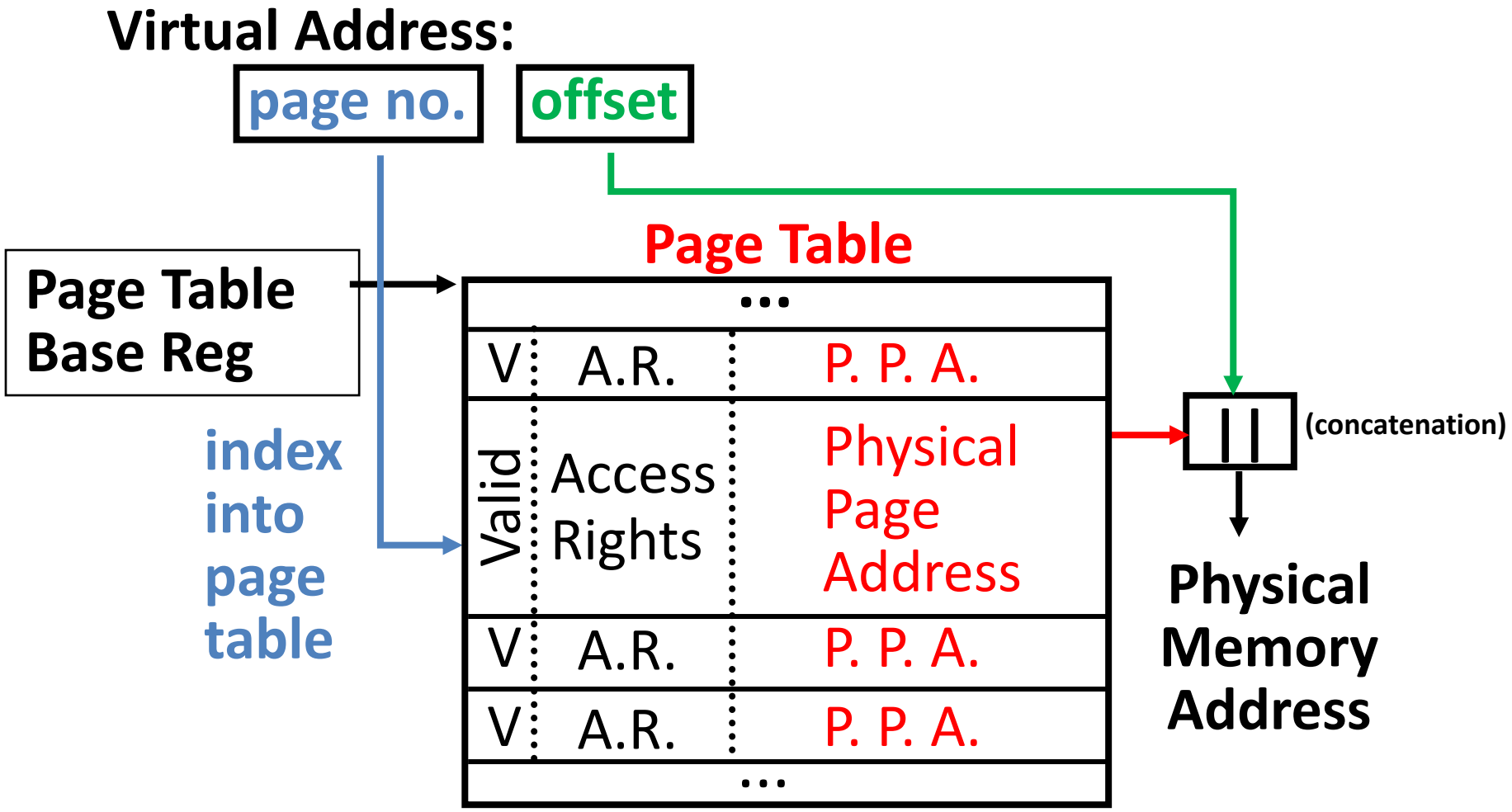



- Use table lookup ("Page Table") for mappings:
  - Page number is index
- Virtual Memory Mapping Function
  - Physical Offset = Virtual Offset
  - Physical Page Number
    = PageTable[Virtual Page Number]
  - (P.P.N. also called "Page Frame")

# Page Table

- A page table is an operating system structure which contains the mapping of virtual addresses to physical locations
  - There are several different ways, all up to the operating system, to keep this data around
- Each process running in the operating system has its own page table
  - "State" of process is PC, all registers, plus page table
  - OS changes page tables by changing contents of Page Table Base Register

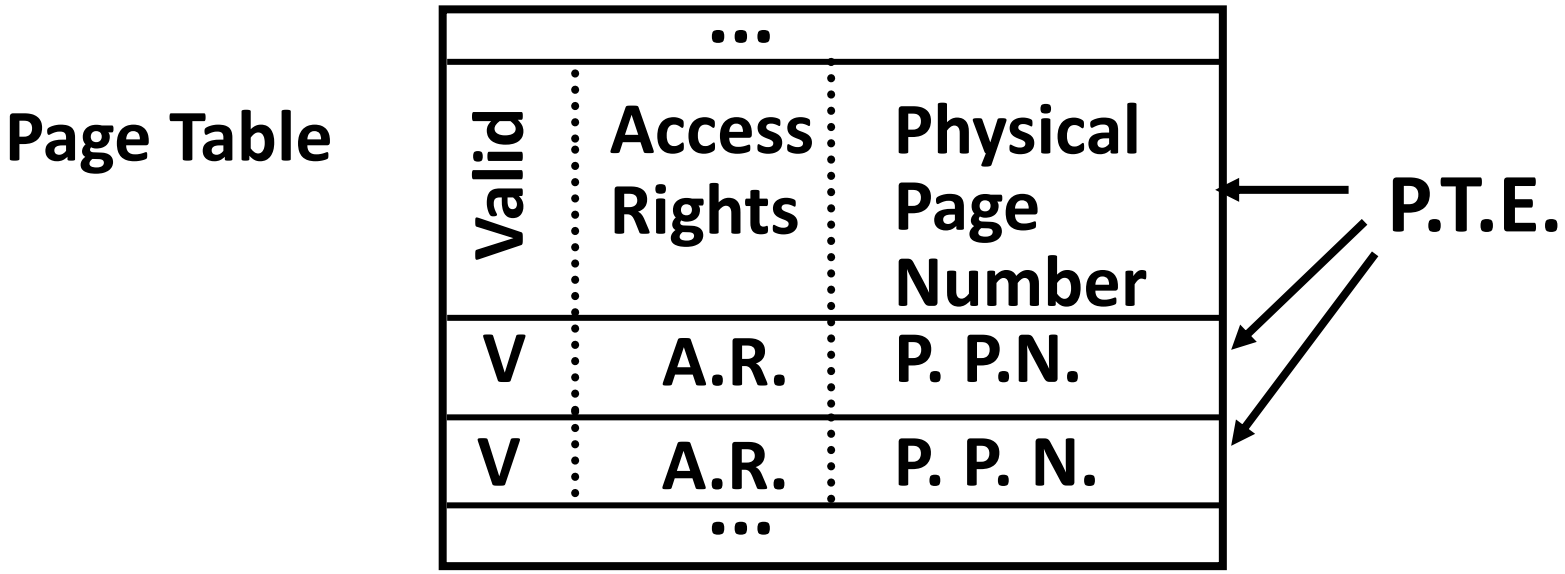
# Address Mapping: Page Table




**Page Table located in physical memory**

# Page Table Continued

- Page Table Entry (PTE) Contains either Physical Page Number or indication not in Main Memory (Valid = 0)
  - OS maps to disk if Not Valid (V = 0)




- If valid, check permission to use page: **Access Rights** (A.R.) may be Read Only, Read/Write, Executable

# Notes on Page Table

- Solves Fragmentation problem: all chunks same size, so all holes can be used
- OS must reserve "***Swap Space***" on disk for each process
- To grow a process, ask Operating System
  - If unused pages, OS uses them first
  - If not, OS swaps some old pages to disk
  - (Least Recently Used to pick pages to swap)
- Each process has own Page Table
- Will add details, but Page Table is essence of Virtual Memory

# Comparing the 2 levels of hierarchy

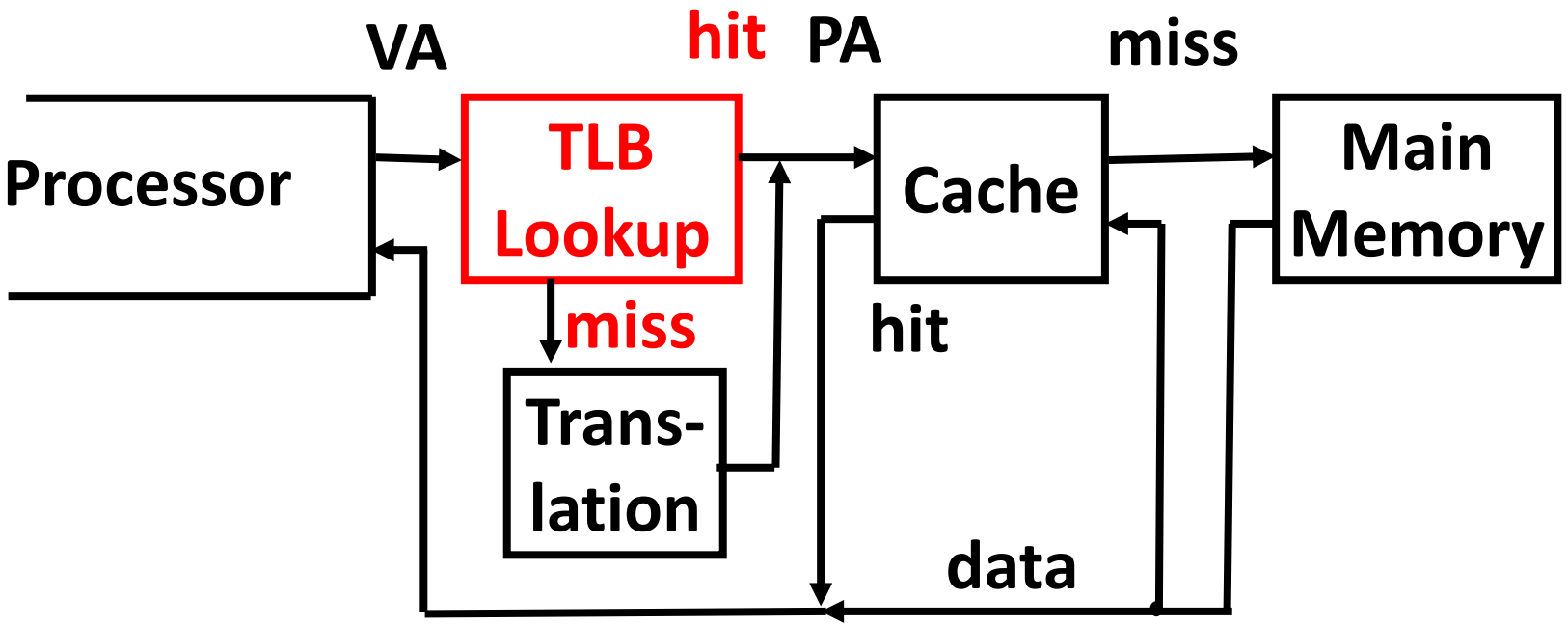
| **Cache Version** | **Virtual Memory Version** |
| --- | --- |
| Block (or Line) | Page |
| Miss | Page Fault |
| Block Size: 32-64B | Page Size: 4K-8KB |
| Placement:<br>Direct Mapped,<br>N-way Set Associative | Fully Associative |
| Replacement:<br>LRU or Random or… | Least Recently Used<br>(LRU) |
| Write Thru or Back | Write Back |

# Virtual Memory Problem #1

- Map every address ⇒ every virtual memory access must be checked with the corresponding entry of the Page Table (which is stored in physical memory for address translation) and another access to get the data
  - This implies 2 physical memory accesses per virtual address ⇒ **SLOW!**
- Observation: since locality in pages of data, there must be locality in ***virtual address translations*** of those pages
- Since small is fast, why not use a small cache of virtual to physical address translations to make translation fast?
- For historical reasons, this cache is called a ***Translation Lookaside Buffer***, or ***TLB***
- TLB makes address translation possible without memory access (to read page table)

# Translation Look-Aside Buffers

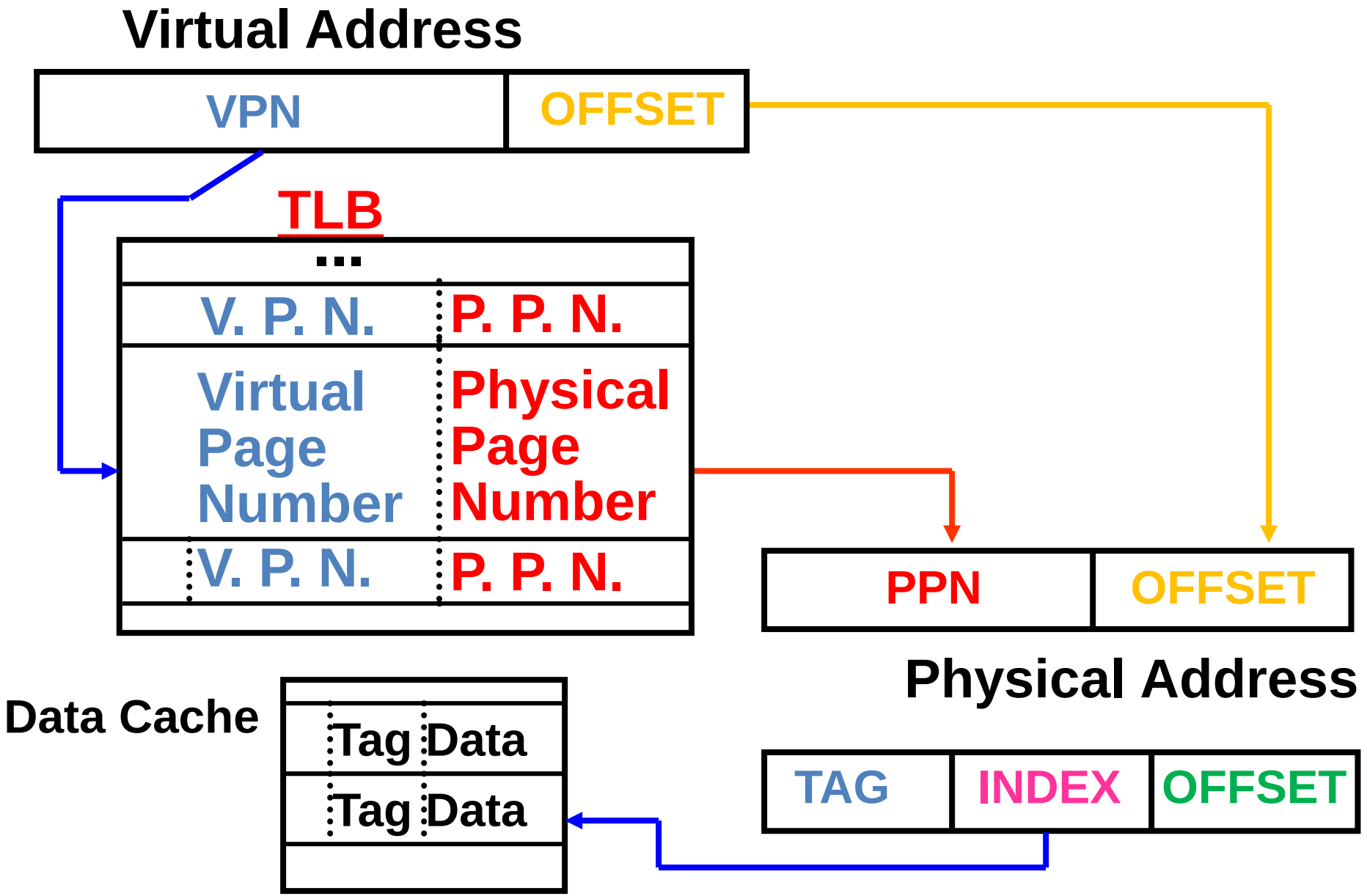
- TLBs usually small, typically 128 - 256 entries
- Like any other cache, the TLB can be direct mapped, set associative, or fully associative



# Load data example

- Suppose we are fetching (loading) some data:
  - Check TLB (input: VPN, output: PPN)
    - hit: fetch translation
    - miss: check page table (in memory)
      - **Page table hit: fetch translation**
      - **Page table miss: page fault, fetch page from disk to memory, return translation to TLB**
  - Check cache (input: PA, output: data)
    - hit: return value
    - miss: fetch value from memory

# Address Translation & Cache Lookup



# Typical TLB Format

| Virtual Address | Physical Address | Dirty | Ref | Valid | Access Rights |
|---|---|---|---|---|---|
| | | | | | |

- TLB just a cache on the page table mappings
- TLB access time comparable to cache (much less than main memory access time)
- Dirty: since we use "write back" policy, need to know whether or not to write page to disk when replaced
- Ref: Used to help calculate LRU on replacement
  - Can be cleared periodically by OS, then checked later to see if page was **referenced**

# What if page not in TLB?

- Option 1: Hardware checks page table and loads new Page Table Entry into TLB
- Option 2: Hardware traps to OS, up to OS to decide what to do
  - This is a simple and flexible strategy!
- MIPS follows Option 2: Hardware knows nothing about page table
  - That is, there is no “page table base register” and instead there is only the TLB

# TLB Miss (simple strategy)

- If the address is not in the TLB,
  MIPS traps to the operating system
  - When in the operating system, we don't do translation (turn off virtual memory)
- The operating system knows which program caused the TLB fault, page fault, and knows what the virtual address desired was requested
  - So we look the entry up in the page table
  - Then we add the entry to the TLB , using empty slot, or evicting old entry.
  - Then we resume the program again at the instruction that failed, and it will succeed this time.

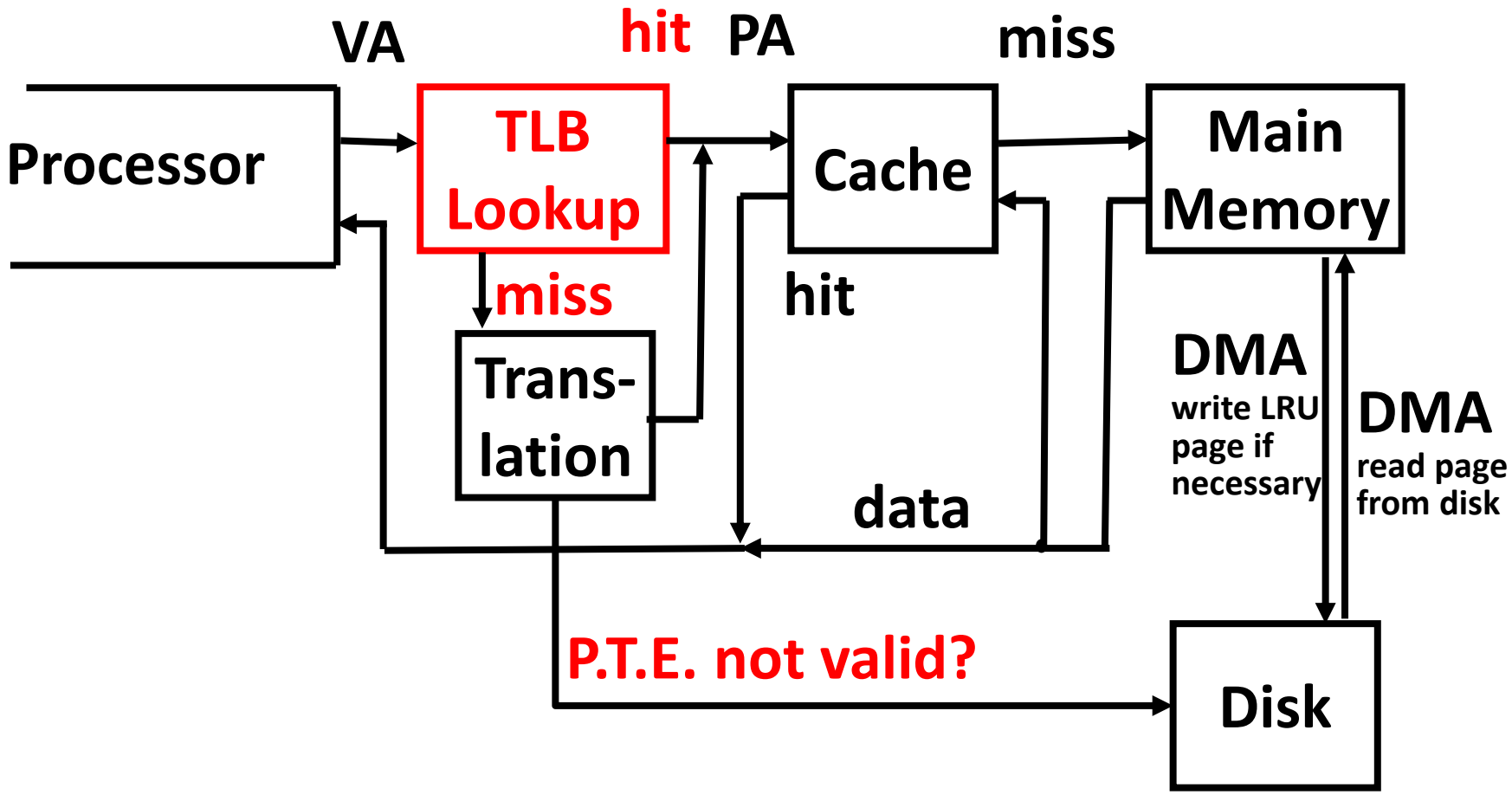
# What if the data is on disk?

- We load the page off the disk into a free block of memory, using a ***DMA transfer***
  - Meantime we switch to some other process waiting to be run
- When the DMA is complete, we get an interrupt, and can then update the process's page table
  - So when we switch back to the task, the desired data will be in memory

# What if we don't have enough memory?

- Chose some physical page belonging to a program,
  - We chose the physical page to evict based on replacement policy (e.g., LRU)
  - If chosen page is dirty, transfer it onto the disk
  - If chosen page is clean (disk copy is up-to-date), then we can simply overwrite that data in memory
- The program previously using the chosen page must have its page table updated to reflect the fact that its memory moved somewhere else.
- Finally, the OS can update our program's page table to use this physical page

# Data on Disk? (1/2)



# Data on Disk? (2/2)

- Virtual memory and cache work together
- Hierarchy must be preserved
  - When a page is migrated to disk, the OS will flush the contents of the page from the cache
  - Also modifies page table and TLB so that attempts to access data on migrated page will produce a fault.

# Question

- A memory reference can encounter three different types of misses:
  - TLB miss, page fault, cache miss
- Consider all combinations of these events with one or more occurring (7 possibilities).
- State if each event can actually occur and under what circumstances

# Answer

| TLB | PAGE TABLE | CACHE | POSSIBLE? HOW? |
|---|---|---|---|
| Hit | Hit | Miss | Possible, though page table not checked if TLB hits |
| Miss | Hit | Hit | TLB misses, but entry found in page table; after retry, data is found in cache |
| Miss | Hit | Miss | TLB misses, but entry found in page table; after retry, data misses in cache |
| Miss | Miss | Miss | TLB misses and is followed by a page fault; after retry, data ***must*** miss cache |
| Hit | Miss | Miss | impossible: cannot have a translation in TLB if page is not present in memory |
| Hit | Miss | Hit | impossible: cannot have a translation in TLB if page is not present in memory |
| Miss | Miss | Hit | impossible: data not allowed in cache if the page is not in memory |

# Virtual Memory Problem #2

- Not enough physical memory! Suppose 4KB pages and…
  - Have only 64 MB ($2^{26}$ B) of physical memory
  - N processes, each given 4 GB ($2^{32}$ B) of virtual memory
  - How many virtual pages per physical page for N=1 process?
  - For what N will we have 1024 virtual pages/physical page?
- Spatial Locality to the rescue
  - Each page is 4 KB, lots of nearby references
- Even for huge programs, typically only accessing a few pages at any given time
  - The "Working Set" of recently used pages

# Virtual Memory Problem #3

- Page Table too big!
  - 4 GB Virtual Memory ÷ 4 KB page
    ⇒ approximately 1 million Page Table Entries,
    each taking up 1 word of memory
    ⇒ 4 MB just for Page Table for 1 process,
    25 processes **will need 100 MB** for Page Tables!
- *Don't give entire virtual address space to an individual process.*
  - *tradeoff memory size of mapping function for slower TLB misses*
- *Alternative mapping functions in textbook but are not in the scope of this course*
  - *Hint: Multi level page table.*

# Things to Remember 1/2

- Apply Principle of Locality Recursively
- Reduce Miss Penalty? add a (L2) cache
- Manage memory to disk? Treat as cache
  - Originally included protection as bonus, now protection is critical
  - Use Page Table of mappings vs. tag/data in cache
- Virtual memory to Physical Memory Translation too slow?
  - Add a cache of Virtual to Physical Address Translations, called a TLB

# Things to Remember 2/2

- Virtual Memory allows protected sharing of memory between processes with less swapping to disk, less fragmentation than always-swap, or base/bound
- Spatial and Temporal Locality means Working Set of Pages is all that must be in memory for process to run fairly well
- TLB to reduce performance cost of VM
- Need a more compact representation to reduce memory size cost of simple 1-level page table (*especially when using a 64-bit address space*)

# Review and More Information

- Textbook 5th edition 5.7, Virtual Memory
- See also 5.8 – common framework for memory hierarchies