# Virtual Memory (and Indirection)

Monday – Virtual Memory and Disks
Wednesday – SIMD1
Friday – SIMD2
Monday – NO CLASS for exam 6

## A Real Problem

- What if you wanted to run a program that needs more memory than you have?

$$LW \underbrace{\hspace{6cm}}_{\text{32-bit address}}$$

## Today's lecture

- Virtual Memory
  - Motivations for virtual memory
  - Basic implementation details of virtual memory
  - How to make virtual memory fast: Translation Lookaside Buffers (TLBs) (i.e., caches!)

## More Real Problems

- Running multiple programs at the same time brings up more problems.

1. Even if each program fits in memory, running 10 programs might not.

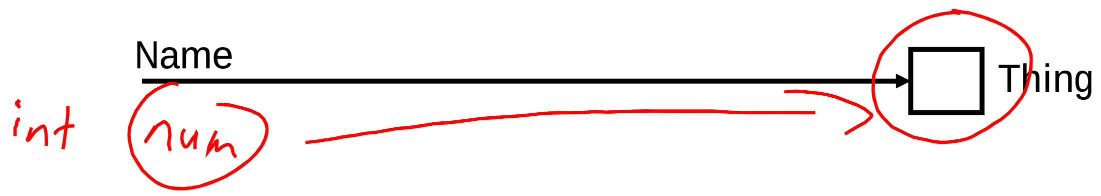
2. Multiple programs may want to store something at the same address.

3. How do we protect one program’s data from being read or written by another program?

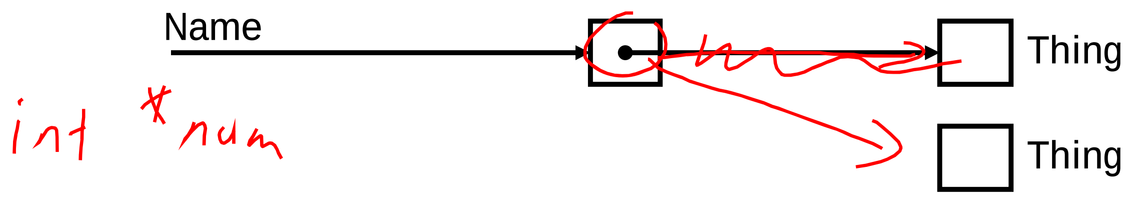
## Use indirection to keep track of where things are stored and not the actual things

- "Any problem in CS can be solved by adding a level of indirection"

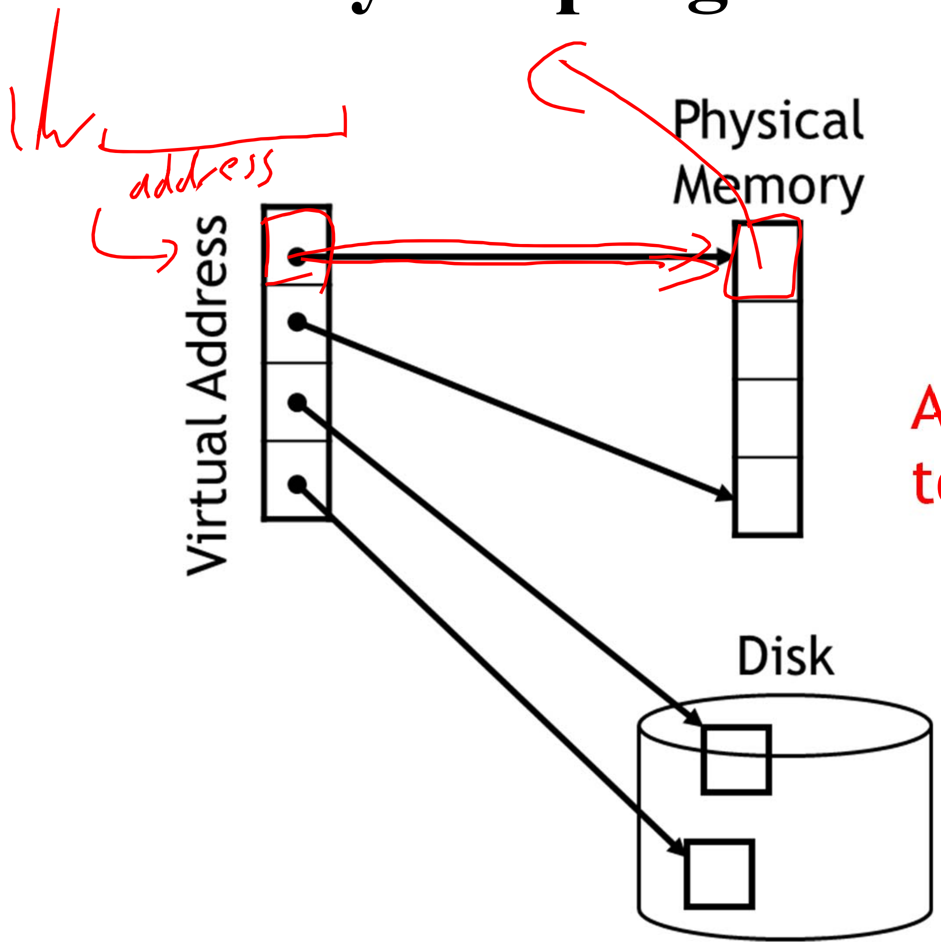
- Without Indirection



- With Indirection



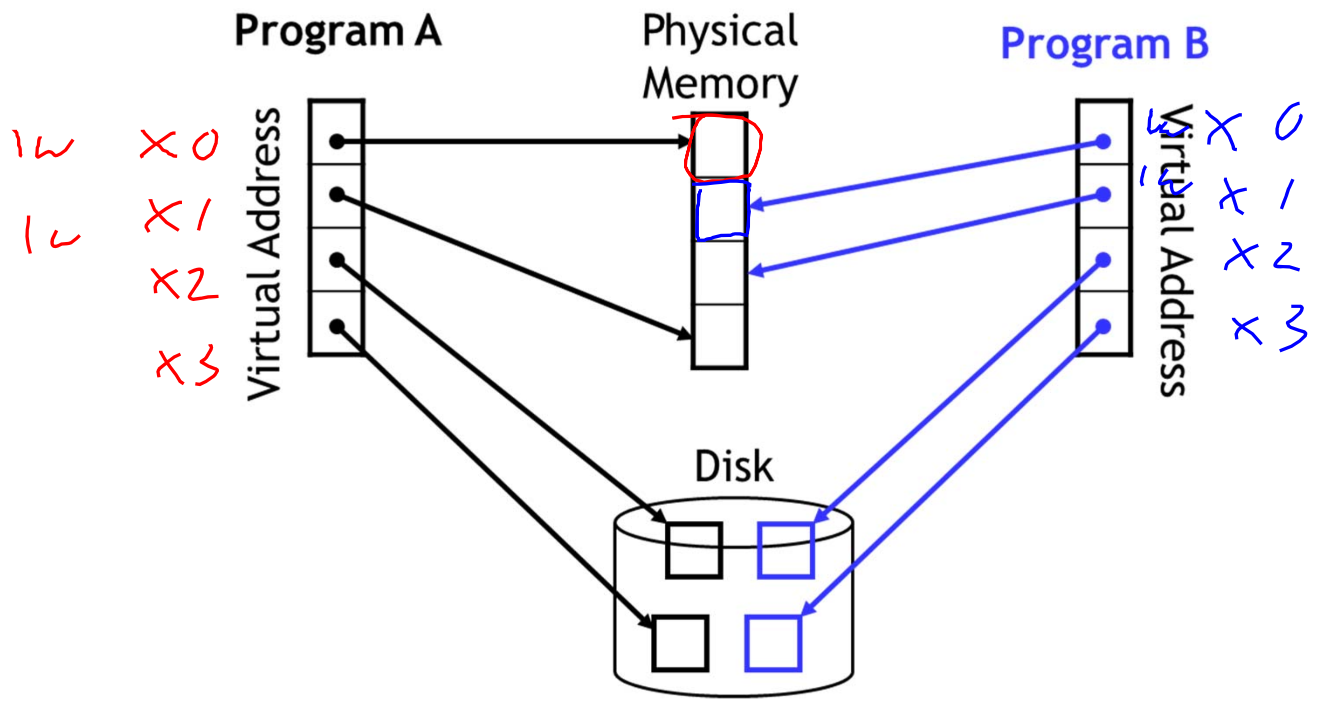
## Virtual Memory translates "virtual addresses" used by the program into "physical addresses"



A virtual address can be mapped to either physical memory or disk.

## All programs write to the same “virtual addresses” but to different “physical addresses”

- By allocating distinct regions of physical memory to A and B, they are prevented from reading/writing each others data.



## Virtual memory treats memory as a cache for disk

- Once the translation infrastructure is in place, the problem boils down to caching.
    - We want the size of disk, but the performance of memory.

- The design of virtual memory systems is really motivated by the high cost of accessing disk.
    - While memory latency is ~100 times that of cache, disk latency is ~100,000 times that of memory.
        - i.e., the miss penalty is a real whopper.

block

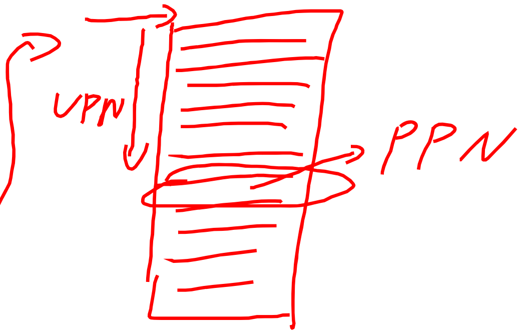
- Hence, we try to minimize the miss rate:
    - VM "pages" are much larger than cache blocks. Why?
    - A fully associative policy is used.
        - With approximate LRU

## Finding the right page

- If it is fully associative, how do we find the right page without scanning all of memory?

## Finding the right page

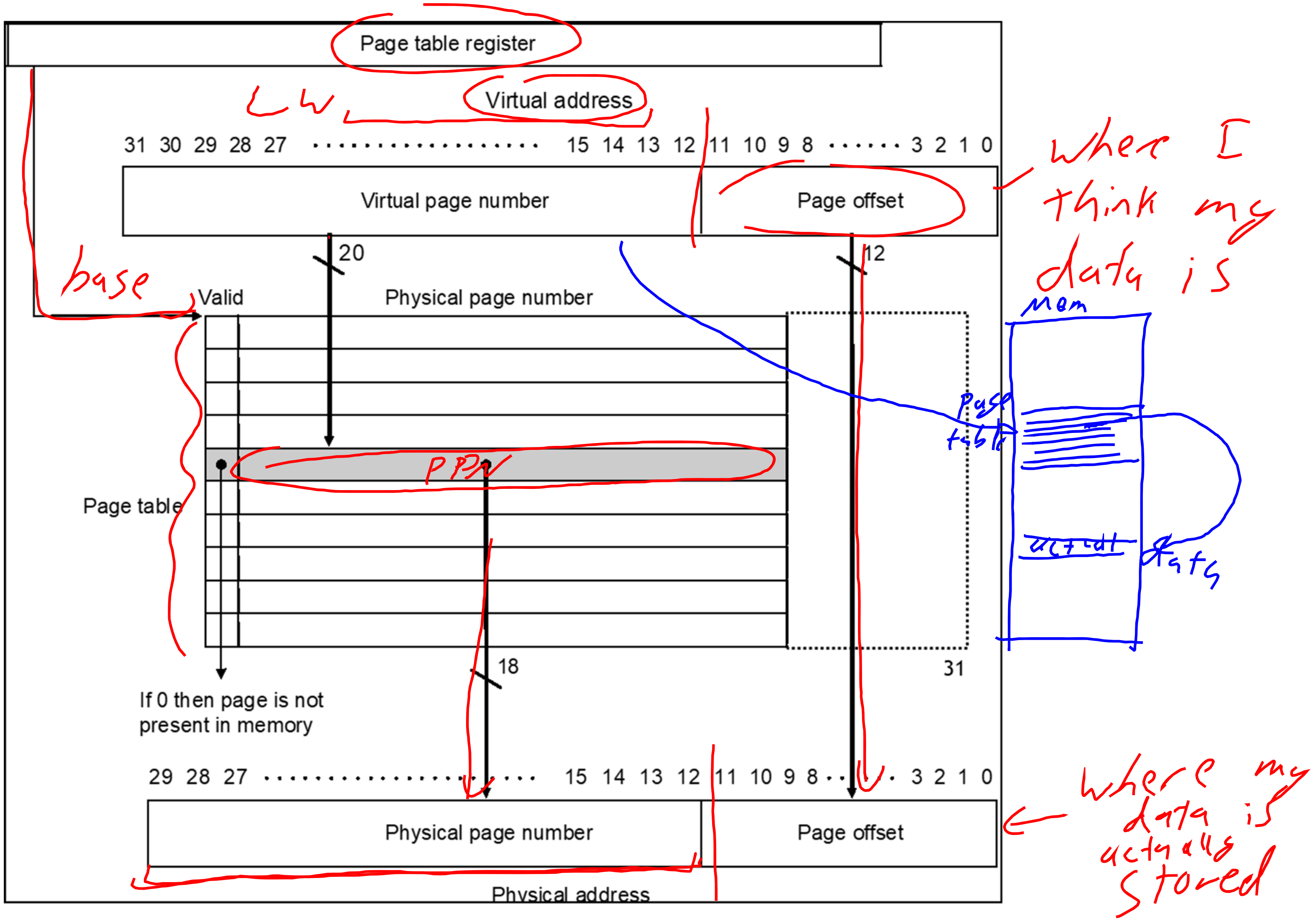
- If it is fully associative, how do we find the right page without scanning all of memory?
  - Use an index, just like you would for a book.

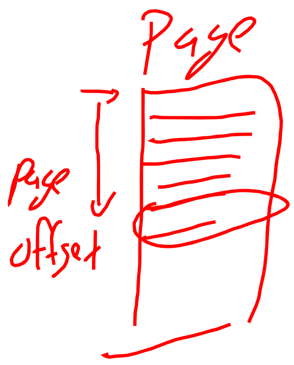


- Our index happens to be called the page table:
  - Each process has a separate page table
    - A "page table base register" points to the starting address of the process's page table
  - The page table is indexed with the virtual page number (VPN)
    - The VPN is all of the bits that aren't part of the page offset.
  - Each entry contains a valid bit, and a physical page number (PPN)
    - The PPN is concatenated with the page offset to get the physical address
  - No tag is needed because the index is the full VPN.

# Page Table

per program →





# Clicker Question

- Consider the following page design
  - 32-bit virtual addresses
  - Page uses 14 bits for the page offset
  - Cache uses 32B cache blocks

How many cache blocks will fit in a page

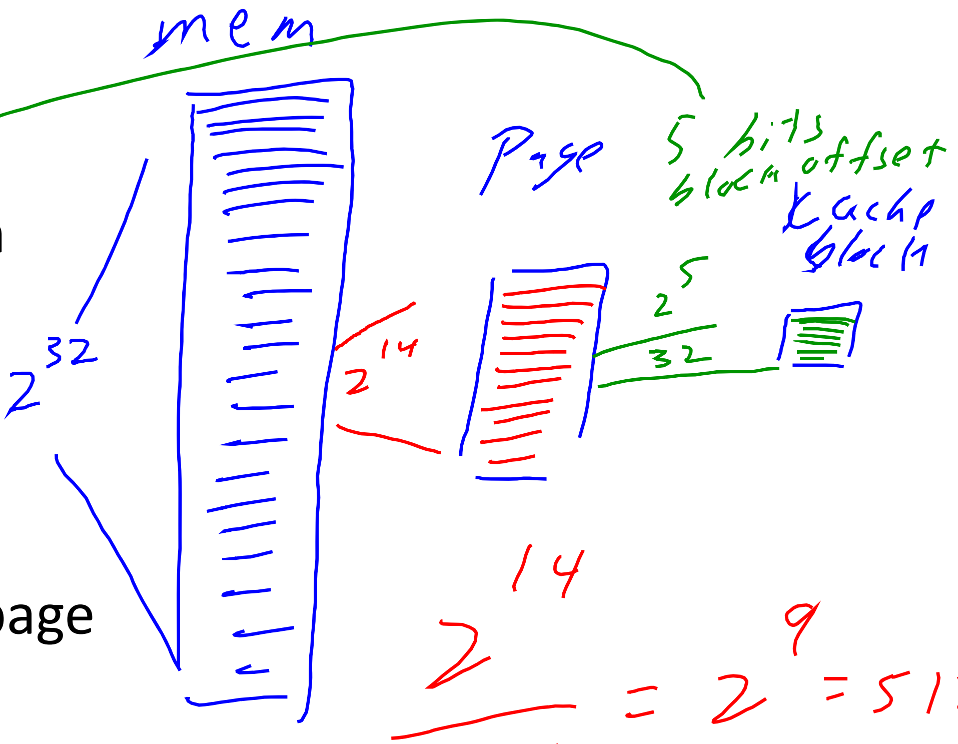
a) 128

b) 256

c) 512

d) 1024

e) $2^{18}$



$$\frac{2^{14}}{2^5} = 2^9 = 512$$

## How big is the page table?

- From the previous slide:
  - Virtual page number is 20 bits.
  - Physical page number is 18 bits + valid bit -> round up to 32 bits. // 4 Bytes

$2^{20}$ elements in page $\cdot$ $2^{2}$ bytes per element

$\simeq 2^{22} = 4$ MB data for page table

- How about for a 64b architecture?

42 bits VA

30 bits for VPN

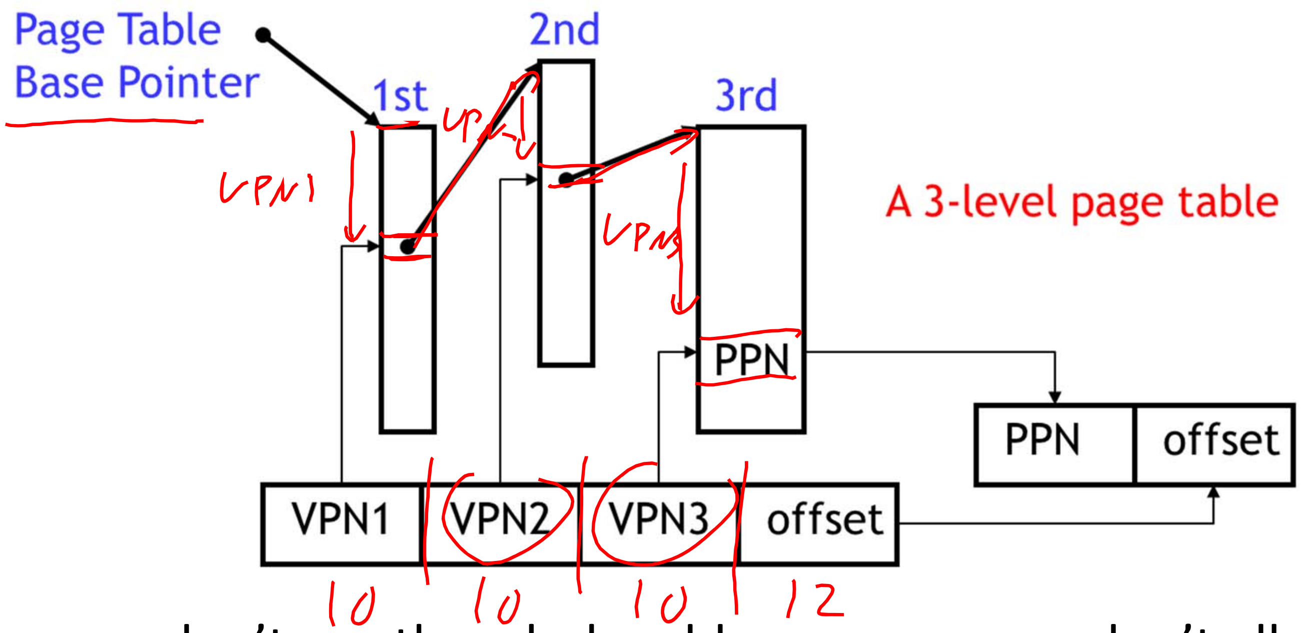
12 bits page offse

$2^{30}$

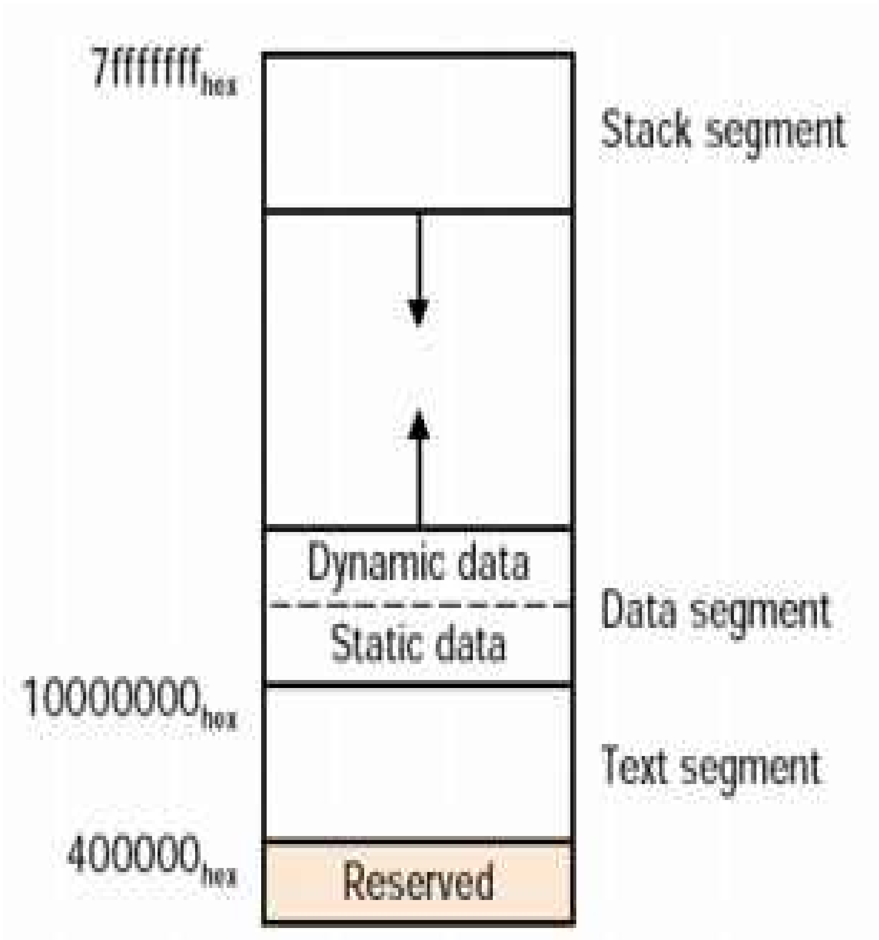
$\rightarrow$ 4 GB

## Dealing with large page tables

- Multi-level page tables
    - “Any problem in CS can be solved by adding a level of indirection”
        - or two…



- Since most processes don’t use the whole address space, you don’t allocate the tables that aren’t needed
    - Also, the 2nd and 3rd level page tables can be “paged” to disk.


7fffffff$_{hex}$
Stack segment
Dynamic data
Static data
Data segment
10000000$_{hex}$
Text segment
400000$_{hex}$
Reserved

## Waitaminute!

- We've just replaced every memory access MEM[addr] with:

  MEM[MEM[MEM[MEM[PTBR + VPN1<<2] + VPN2<<2] + VPN3<<2] + offset]

  - *i.e.*, 4 memory accesses

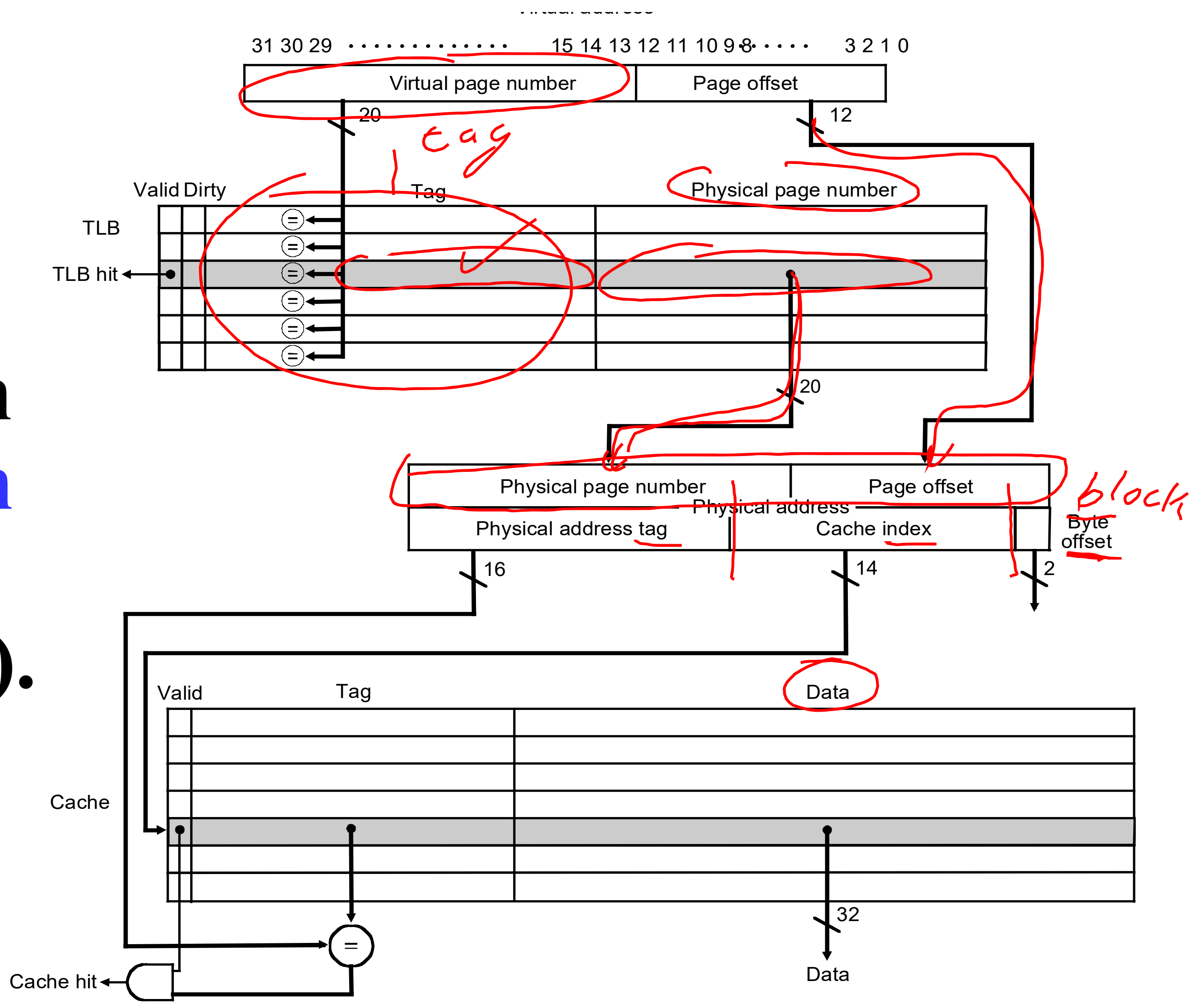
- And we haven't talked about the bad case yet (*i.e.*, page faults)…

"Any problem in CS can be solved by adding a level of indirection"

- except too many levels of indirection…

- How do we deal with too many levels of indirection?

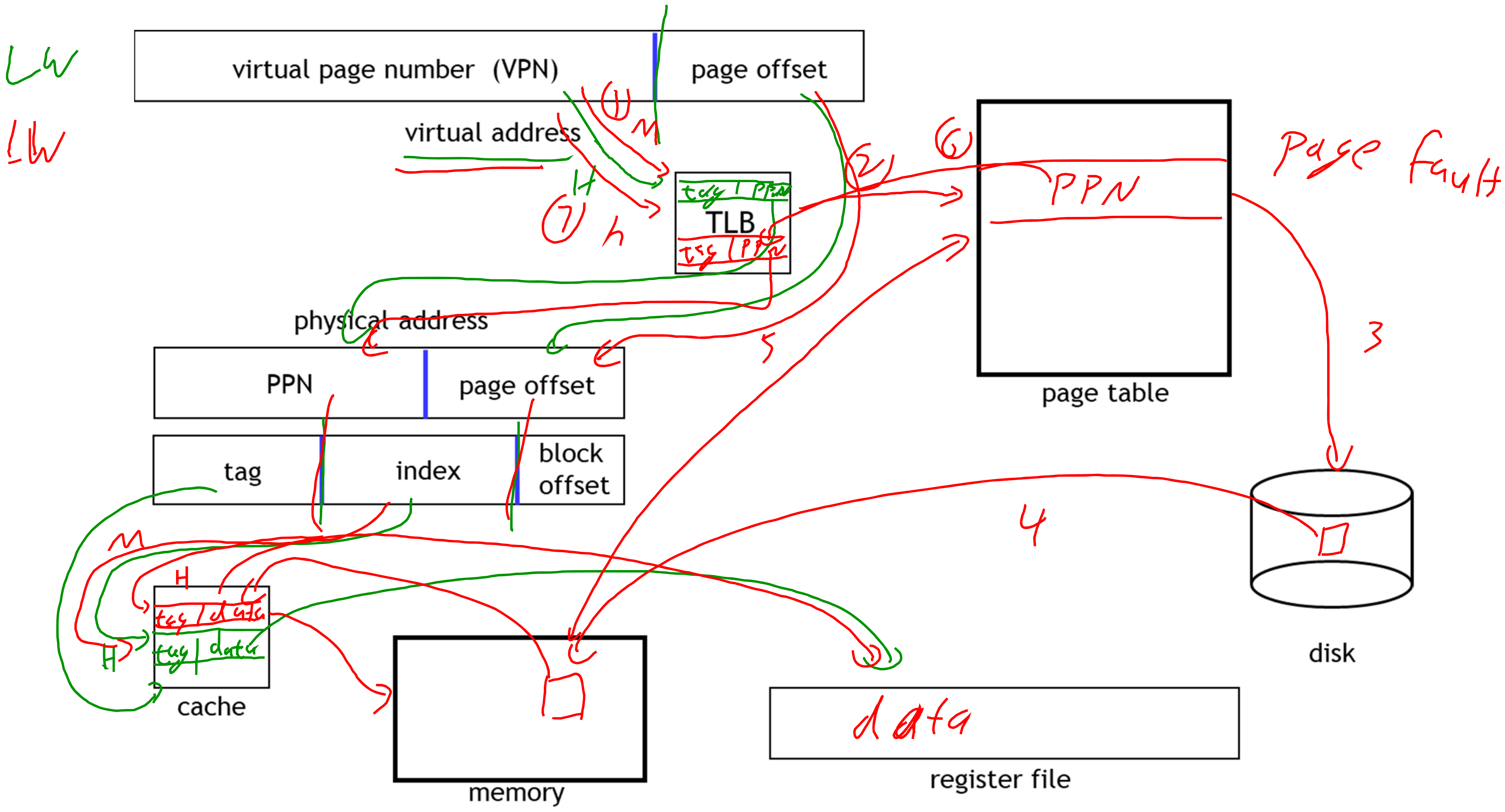
## Virtual to Physical translations are cached in a Translation Lookaside Buffer (TLB).



## What about a TLB miss?

- If we miss in the TLB, we need to “walk the page table”
  - In MIPS, an exception is raised and software fills the TLB
    - MIPS has “TLB_write” instructions
  - In x86, a “hardware page table walker” fills the TLB

- What if the page is not in memory?
  - This situation is called a **page fault**.
  - The operating system will have to request the page from disk.
  - It will need to select a page to replace.
    - The O/S tries to approximate LRU (see CS241/CS423)
  - The replaced page will need to be written back if dirty.


LW
LW
virtual page number (VPN)
page offset
virtual address
TLB
tag | PPN
tag | PPN
page table
PPN
Page fault
physical address
PPN
page offset
tag
index
block offset
cache
tag | data
tag | data
memory
register file
data
disk
1
2
3
4
5
6
7
M
H
h

## Virtual Memory & Prefetching

- Don’t want to cause page faults by prefetching

- Prefetches typically dropped when they miss in the TLB
  - Don’t want to disrupt program’s execution for a prefetch.
  - May cause a hardware TLB fill on x86 platforms.

- HW prefetchers don’t cross page boundaries.
  - They use physical addresses
    - Don’t use the TLB.
  - After page boundary don’t know where next page lies
  - Sequential stream will have a few misses @ beginning of each page

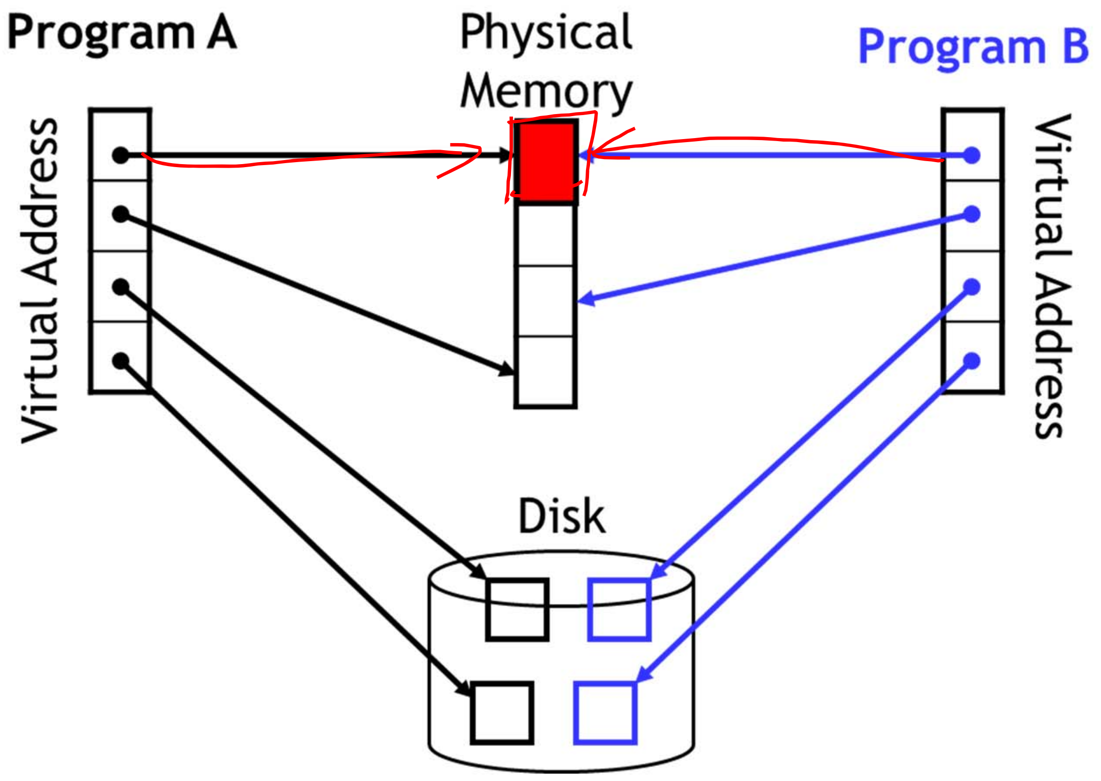
## Memory Protection

- In order to prevent one process from reading/writing another process's memory, we must ensure that a process cannot change its virtual-to-physical translations.
- Typically, this is done by:
  - Having two processor modes: user & kernel.
    - Only the O/S runs in kernel mode
  - Only allowing kernel mode to write to the virtual memory state, *e.g.*,
    - The page table
    - The page table base pointer
    - The TLB

## Pages can be shared between programs

- For example, if you run two copies of a program, the O/S will share the code pages between the programs.

stdio.h



## Summary

- Virtual memory is pure manna from heaven:
  - It means that we don’t have to manage our own memory.
  - It allows different programs to use the same (virtual) addresses.
  - It provides protect between different processes.
  - It allows controlled sharing between processes (albeit somewhat inflexibly).
- The key technique is **indirection**:
  - Yet another classic CS trick you’ve seen in this class.
  - Many problems can be solved with indirection.
- Caching made a few cameo appearances, too:
  - Virtual memory enables using physical memory as a cache for disk.
  - We used caching (in the form of the Translation Lookaside Buffer) to make Virtual Memory’s indirection fast.