CSE 140
Computer Architecture

Lecture 9 – Virtual Memory (2)

Announcement

- HW #3a
  - Due in 2 weeks (same as HW #3 – out next week)
- Project #1
  - Due 10/11 (Friday) at 11:59pm
- Reading assignment #4
  - Chapter 5.7 – 5.8
    - Do all Participation Activities in each section
    - Access through CatCourses
    - Due Thursday (9/26) at 11:59pm
  - Basic idea of using cache (at CatCourses, under Files)
    - ***How L1 and L2 CPU Caches Work.pdf***

Virtual memory revisited

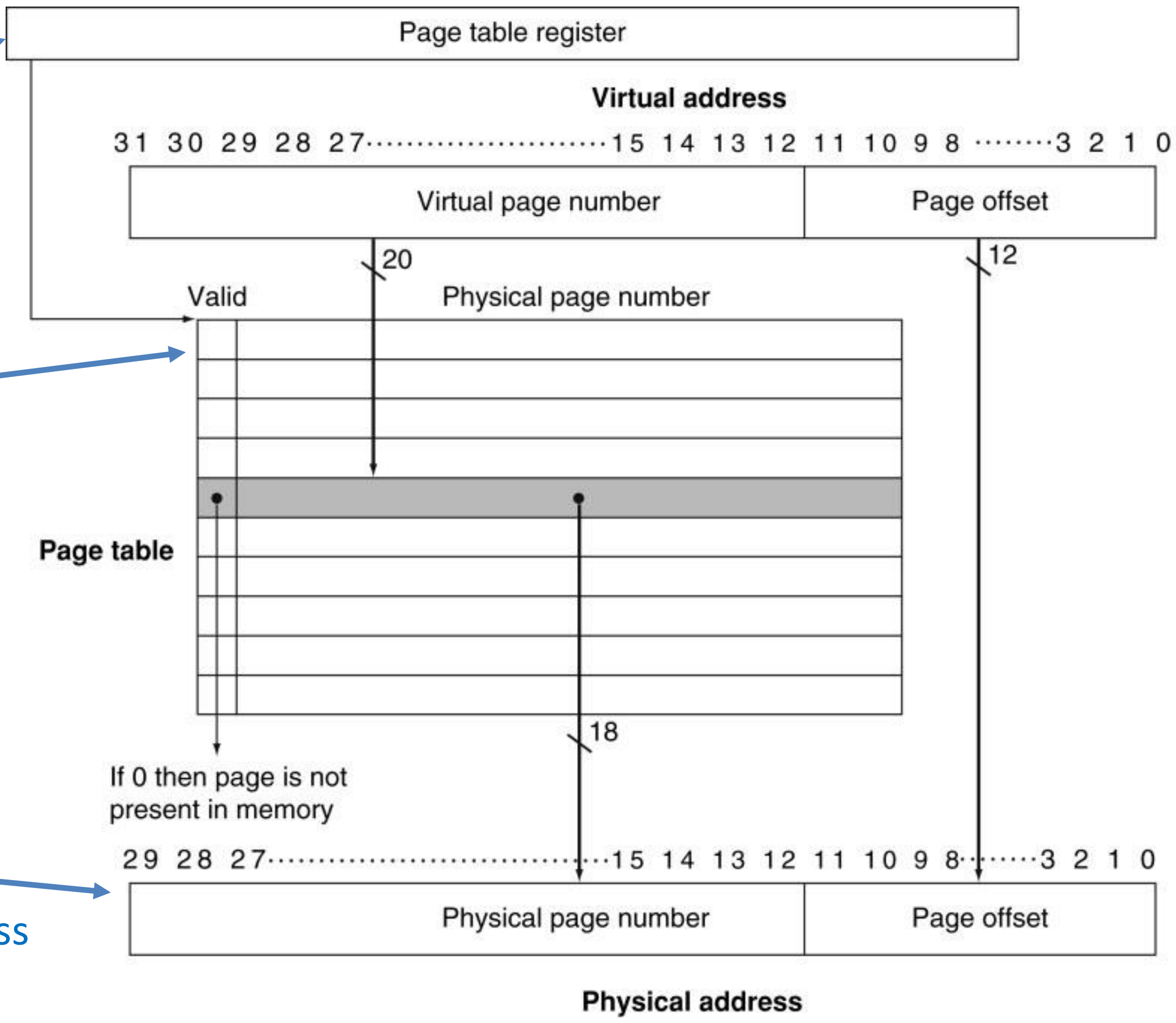
- Remember the motivation for VM:
  - Sharing memory with protection
    - Different physical pages can be allocated to different processes (sharing)
    - A process can only touch pages in its own page table (protection)
  - Separate address spaces
    - Since programs work only with virtual addresses, different programs can have different data/code at the same address!

Address Mapping: Page Table

This register tells you where to find the page table

- Page Table is located in physical memory
- May contain extra entries about the page
  - Dirty bit, ref bit, access rights

Once the page is in the memory, form the physical memory address

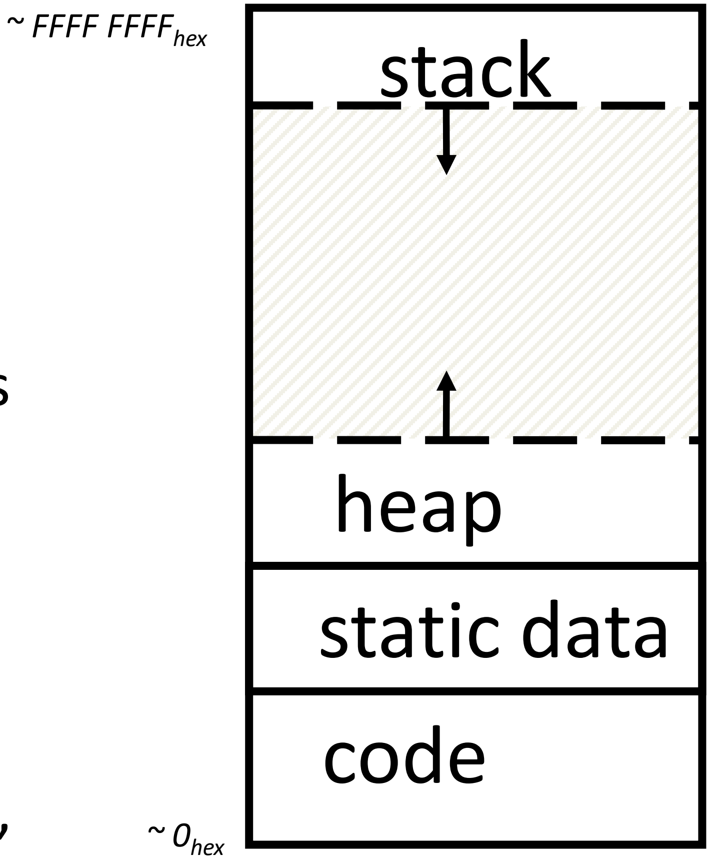


Notes on Page Table

- Solves Fragmentation problem: all chunks same size, so all holes can be used (like slab allocator)
- 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

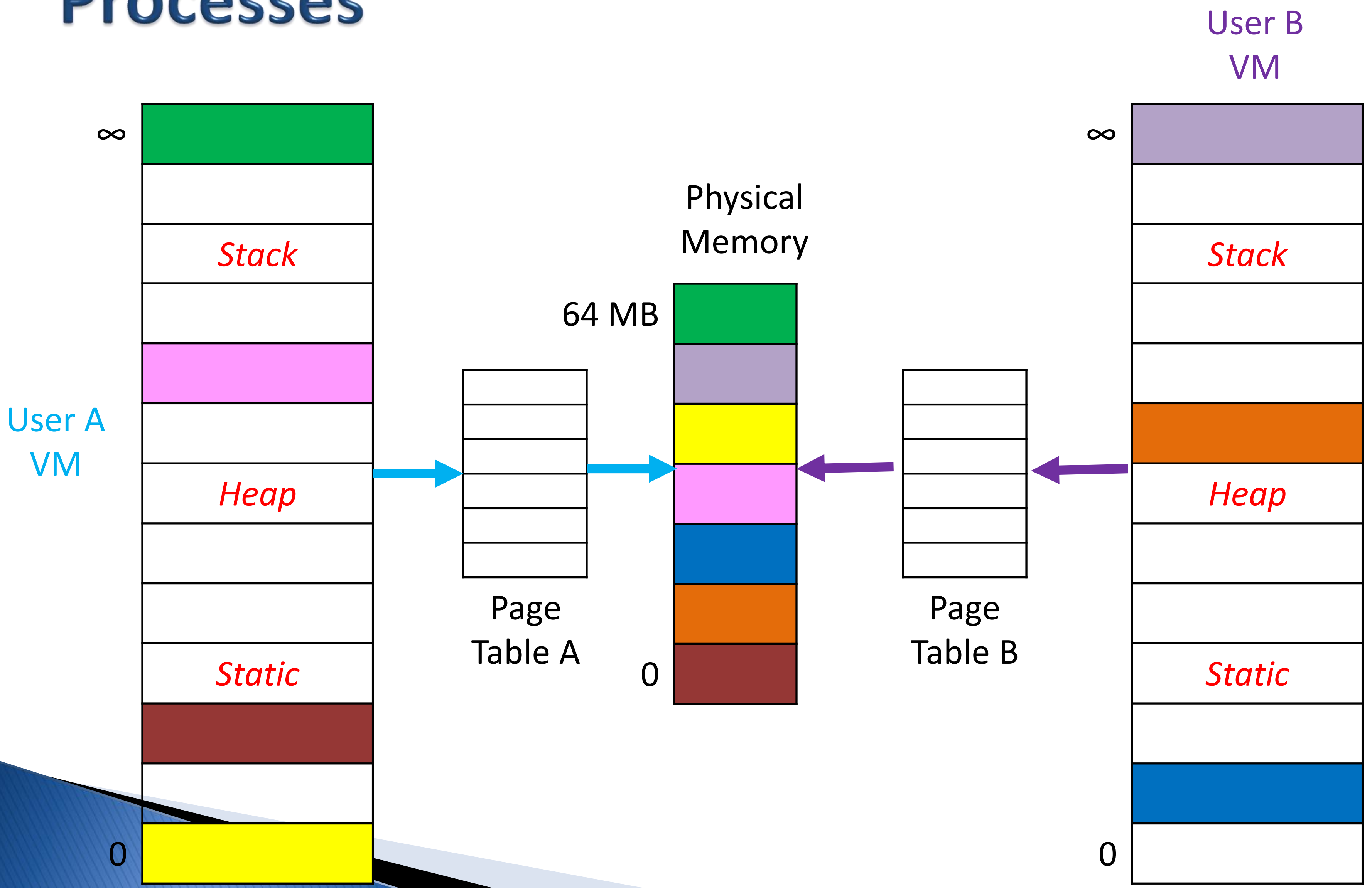
Why would a process need to "grow"?

- A program's address space contains 4 regions:
  - stack: local variables, grows downward
  - heap: space requested for pointers via malloc() ; resizes dynamically, grows upward
  - static data: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change



For now, OS somehow prevents accesses between stack and heap (gray hash lines).

Paging/Virtual Memory Multiple Processes



Comparing the 2 levels of hierarchy

- Cache version
  - Block
  - Miss
  - Block Size: 32-64B
  - Placement:
    - Direct Mapped
    - N-way Set Associative
  - Replacement:
    - LRU or Random
  - Write Thru or Back

- Virtual Memory version
  - Page
  - Page Fault
  - Page Size: 4K-8KB
  - Placement:
    - Fully Associative
  - Replacement:
    - LRU
  - Write Back

Virtual Memory Problem

- Map every address → 1 indirection via Page Table in memory per virtual address
  - For every virtual memory access
    - 2 physical memory accesses (1 for page table, 1 for actual access)
- 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

Translation Look-Aside Buffers (TLBs)

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

Why Translation Lookaside Buffer (TLB)?

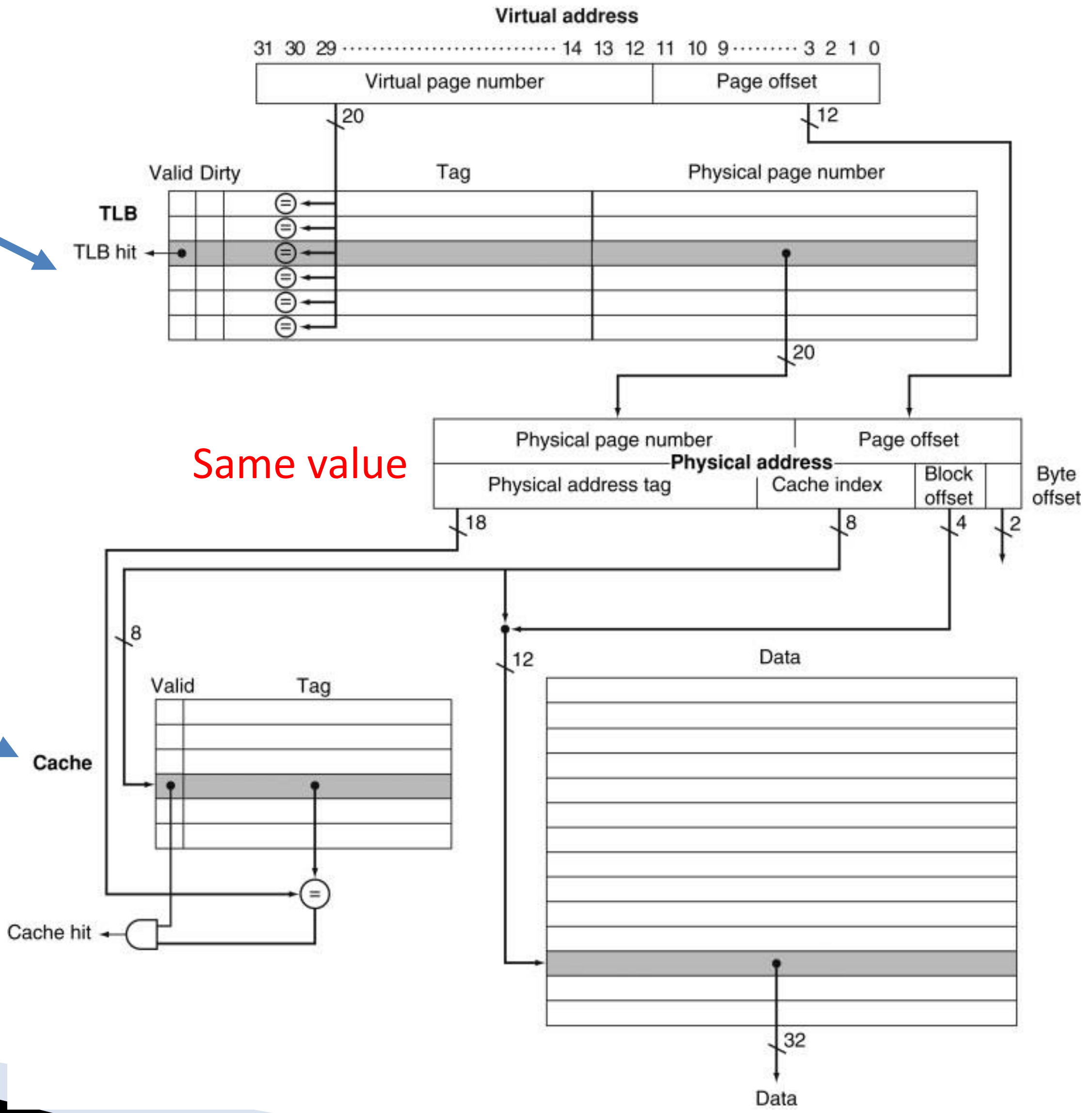
- Paging is most popular implementation of virtual memory (vs. base/bounds)
- Every paged virtual memory access must be checked against Entry of Page Table in memory to provide protection / indirection
- Cache of Page Table Entries (TLB) makes address translation possible without memory access in common case to make memory accesses fast

The big picture

If virtual page number is not found in TLB (miss), go to page table to look for it.

If virtual page number is found in TLB (hit), proceed to cache access

Same value



Three Advantages of Virtual Memory

1. Translation:
    - Program can be given consistent view of memory, even though physical memory is scrambled
    - Makes multiple processes reasonable
    - Only the most important part of program (“Working Set”) is in the physical memory
    - Contiguous structures (like stacks) use only as much physical memory as necessary yet still allow to grow later

Three Advantages of Virtual Memory

2. Protection:
   - Different processes protected from each other
   - Different pages can be given special behavior
     - Read Only, Invisible to user programs, etc.
   - Kernel (OS) data protected from User programs
   - Very important for protection from malicious programs
     - Far more “viruses” under Microsoft Windows (was built for single user)
   - Special Mode in processor (“Kernel mode”) allows processor to change page table/TLB
3. Sharing:
   - Can map same physical page to multiple users (“Shared memory”)

Virtual Memory Overview (1/2)

- User program view of memory:
  - Contiguous
  - Start from same set address
  - Infinitely large (depends on disk space)
  - Is the only running program
- Reality:
  - Non-contiguous
  - Start wherever available memory is
  - Finite size
  - Many programs running at a time

Virtual Memory Overview (2/2)

- Virtual memory provides:
  - illusion of contiguous memory
  - all programs starting at same set address
  - illusion of ~ infinite memory
    - (2^{32} or 2^{64} bytes)
  - Protection
- Implementation:
  - Divide memory into “chunks” (pages)
  - Operating system controls page table that maps virtual addresses into physical addresses
  - Think of memory as a cache for disk
  - TLB is a cache for the page table
  - Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well

4 Qs for any Memory Hierarchy

- Q1: Where can a block be placed (in cache)?
  - One place (direct mapped)
  - A few places (set associative)
  - Any place (fully associative)
- Q2: How is a block found?
  - Indexing (as in a direct-mapped cache)
  - Limited search (as in a set-associative cache)
  - Full search (as in a fully associative cache)
  - Separate lookup table (as in a page table, a block is in a page)
- Q3: Which block is replaced on a miss?
  - Least recently used (LRU)
  - Random
- Q4: How are writes handled?
  - Write through (Level never inconsistent w/lower)
  - Write back (Could be “dirty”, must have dirty bit)

Fetching data on a memory read

- What are the steps to perform LW instruction?
- Check TLB (input: VPN, output: PPN)
  - hit: fetch translation (goto check cache)
  - miss: check page table (in memory)
    - Page table hit: fetch translation, update translation to TLB
    - Page table miss: page fault, fetch page from disk to memory, update translation to TLB and page table
- Check cache (input: physical memory address, output: data)
  - hit: return value
  - miss: fetch value from memory, copy it in cache, return value

What if 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
  - MIPS follows Option 2: Hardware knows nothing about page table (no logic hardware between TLB and page table)
  - A trap is a synchronous exception in a user process, often resulting in the OS taking over and performing some action before returning to the program.
    - More about exceptions in future lecture

What if the data is on disk (page fault)?

- We load the page off the disk into a free block of memory, using a DMA transfer (Direct Memory Access – special hardware support to avoid using processor)
  - Meantime we switch to some other process waiting to be run
- When the DMA is complete, we get an interrupt and 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?

- We chose some other page belonging to a program and transfer it onto the disk if it is dirty
  - If clean (disk copy is up-to-date), just overwrite that data in memory
  - We chose the page to evict based on replacement policy (e.g., LRU)
- Update that program's page table to reflect the fact that its memory moved somewhere else
- If continuously swap between disk and memory, called Thrashing

Typical TLB Format

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

Tag	Physical Page #	Dirty	Reference	Valid	Access Right

Question (1/3)

Why are there more bits in VA?

- 40-bit virtual address, 16 KB page

Virtual Page Number (? bits)	Page Offset (? bits)

- 32-bit physical address

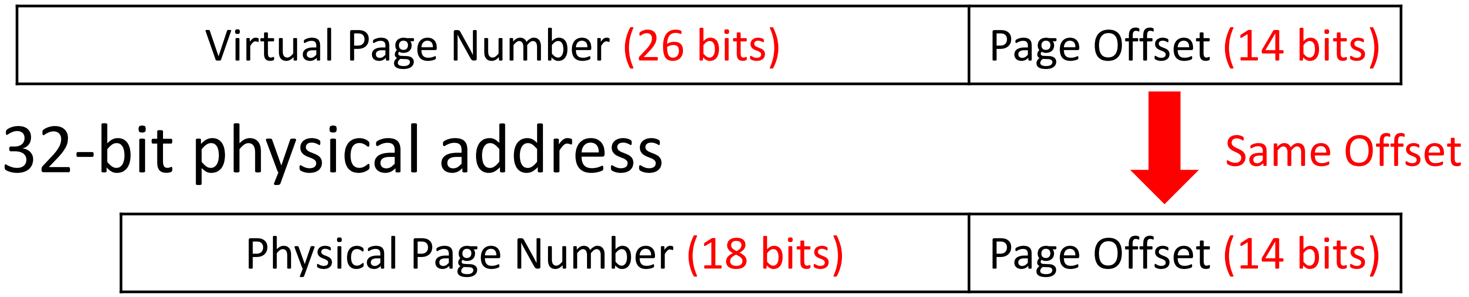
PhysicalPage Number (? bits)	Page Offset (? bits)

- Number of bits in
  - Virtual Page Number/Page offset?
  - Physical Page Number/Page offset?

1: 22/18 (VPN/PO), 18/14 (PPN/PO)
2: 24/16, 16/16
3: 26/14, 18/14
4: 26/14, 22/10
5: 28/12, 20/12

Question (1/3)

- 40-bit virtual address, 16 KB page
- 32-bit physical address



- Number of bits in
  - Virtual Page Number/Page offset?
  - Physical Page Number/Page offset?

1: 22/18 (VPN/PO), 18/14 (PPN/PO)
2: 24/16, 16/16
3: 26/14, 18/14
4: 26/14, 22/10
5: 28/12, 20/12

Question (2/3) 40b VA, 32b PA

- 2-way set-assoc. TLB, 256 “sets”, 2 TLB entries per set

TLB Tag (? bits)	TLB Index (? bits)	Page Offset (14 bits)

$\underbrace{\text{TLB Tag (? bits)} \quad \text{TLB Index (? bits)}}_{\text{Virtual Page Number (26 bits)}}$

- TLB Entry: Valid bit, Dirty bit, Access Control (2 bits), Virtual Page Number, Physical Page Number

V	D	Access (2 bits)	TLB Tag (? bits)	Physical Page # (18 bits)

1: 12 / 14 / 34 (TLB Tag / Index / Entry)
2: 14 / 12 / 36
3: 18 / 8 / 40
4: 18 / 8 / 54

Question (2/3) 40b VA, 32b PA

- 2-way set-assoc. TLB, 256 (2^8) "sets", 2 TLB entries per set => 8 bit index

TLB Tag (18 bits)	TLB Index (8 bits)	Page Offset (14 bits)

Virtual Page Number (26 bits)

- TLB Entry: Valid bit, Dirty bit, Access Control (2 bits), Virtual Page Number, Physical Page Number

V	D	Access (2 bits)	TLB Tag (18 bits)	Physical Page # (18 bits)

1: 12 / 14 / 34 (TLB Tag / Index / Entry)
2: 14 / 12 / 36
3: 18 / 8 / 40
4: 18 / 8 / 54

Question (3/3)

- 2-way set-assoc, 64KB data cache, 64B block

Cache Tag (? bits)	Cache Index (? bits)	Block Offset (? bits)

Physical Page Address (32 bits)

- Data Cache Entry: Valid bit, Dirty bit, Cache tag, ? bits of Data

V	D	Cache Tag (? Bits)	Cache Data (? Bits)

- Number of bits in Data cache Tag / Index / Offset / Cache Entry?

1: 12 / 10 / 14 / 87 (Tag/Index/Offset/Cache Entry)
2: 16 / 10 / 6 / 86
3: 16/ 10 / 6 / 530
4: 17/ 9 / 6 / 87
5: 17/ 9 / 6 / 531

Question (3/3)

- 2-way set-assoc data cache, 64KB/64B = 1K (2^{10}) blocks, 2 blocks per set => 9 bit index

Cache Tag (17 bits)	Cache Index (9 bits)	Block Offset (6 bits)

Physical Page Address (32 bits)

- Data Cache Entry: Valid bit, Dirty bit, Cache tag + 64 Bytes of Data

V	D	Cache Tag (17 Bits)	Cache Data (64B = 512 Bits)

1: 12 / 10 / 14 / 87 (Tag/Index/Offset/Entry)
2: 16 / 10 / 6 / 86
3: 16/ 10 / 6 / 530
4: 17/ 9 / 6 / 87
5: 17/ 9 / 6 / 531

Summary

- Virtual memory to Physical Memory Translation too slow?
    - Add a cache of Virtual to Physical Address Translations, called a TLB
- Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well
- Virtual Memory allows protected sharing of memory between processes with less swapping to disk