18-447 Lecture 17:
VM Concepts: Address Translation

James C. Hoe

Department of ECE

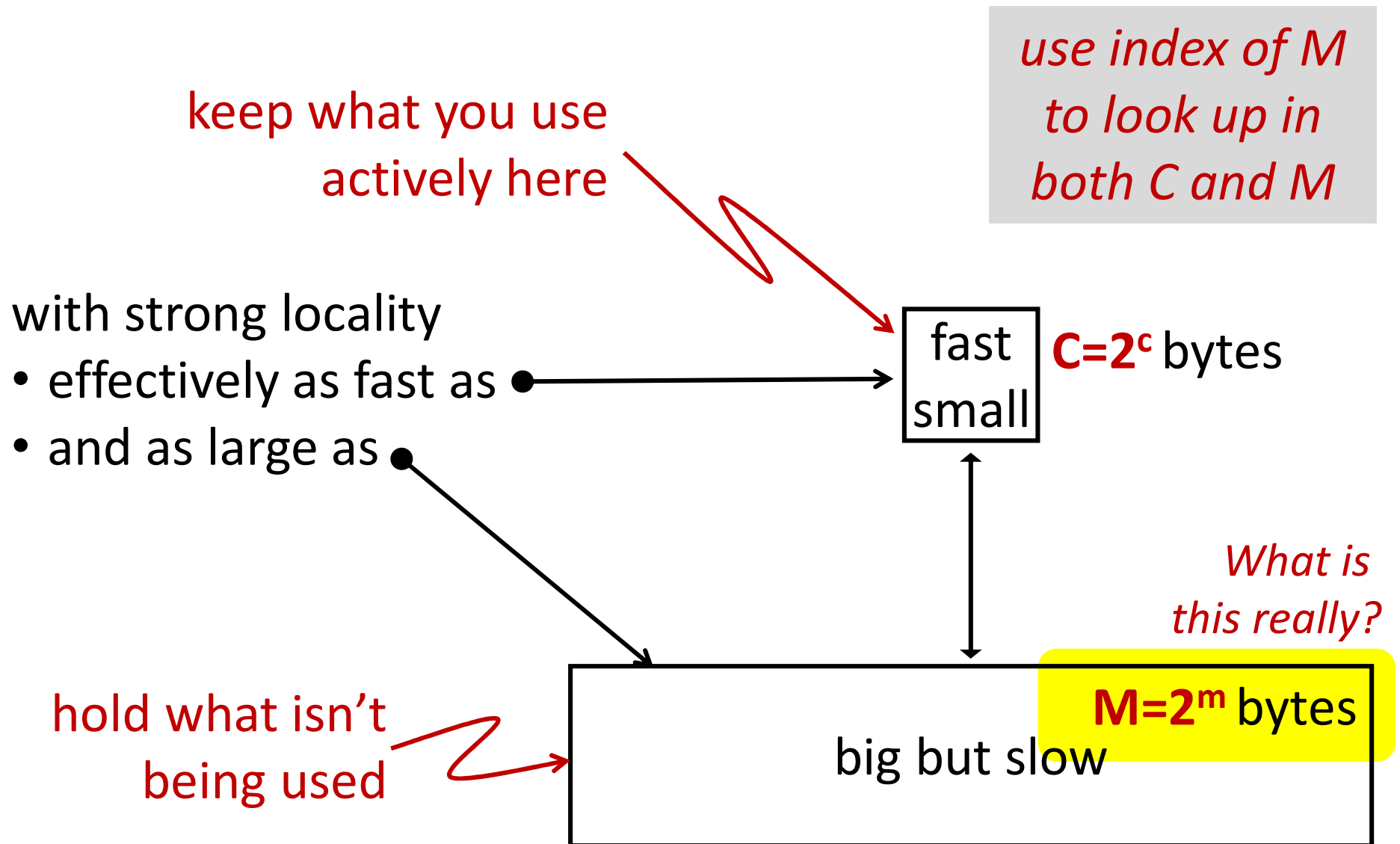Carnegie Mellon University

Format of the Midterm

- Covers lectures (L11~L18), HW, labs, assigned readings (from textbooks and papers)
- Types of questions
  - freebies: remember the materials
  - **>> probing: understand the materials <<**
  - applied: apply the materials in original interpretation
- ****70 minutes, 70 points****
  - point values calibrated to time needed
  - closed-book, one 8½x11-in^2 hand-written cribsheet
  - no electronics
  - use pencil or black/blue ink only
  - ****new rule** no questions in the final ~~20~~ 10 min**

Housekeeping

- Your goal today
  - see "Virtual Memory" in easy to digest pieces
- Notices
  - HW 5, due 4/4
  - Lab 3, due week 10
  - Midterm 2, Wed, 4/6
- Readings
  - P&H Ch 5

Cache Hierarchy

use index of M to look up in both C and M

keep what you use actively here

with strong locality
- effectively as fast as
- and as large as

fast small $C=2^c$ bytes

What is this really?

$M=2^m$ bytes

big but slow

hold what isn't being used

2 Parts to Modern VM

- In a multi-tasking system, **virtual** memory supports the **illusion** of a <u>large</u>, <u>private</u>, and <u>uniform</u> memory space to each process

- Ingredient A: naming and protection
  - each process sees a large, contiguous address space without holes **(for convenience)**
  - each process's memory is private, i.e., protected from access by other processes **(for sharing)**

- Ingredient B: demand paging **(for hierarchy)**
  - capacity of secondary storage (swap space on disk)
  - speed of primary storage (DRAM)

The Common Denominator: Address Translation

- <u>Large</u>, <u>private</u>, and <u>uniform</u> abstraction achieved through address translation
    - user process operates on effective address (**EA**)
    - HW translates from **EA** to physical address (**PA**) on every memory reference
- Through address translation
    - control which physical locations (DRAM and/or swap disk) can be referred to by a process
    - allow dynamic allocation and relocation of physical backing store (where in DRAM and/or swap disk)
- Address translation HW and policies controlled by the OS and protected from user

Beginnings of Memory Protection

- No need for protection or translation early on
  - single process, single user at a time
  - access all locations directly with **PA**
- Multitasking 101
  - each process limited to a non-overlapping, contiguous physical memory region (space doesn't start from addr 0 . . .)
  - everything must fit in the region
  - how to keep one process from reading or trashing another process's code and data? *(see corewars.org)*

active process's region
another process's region
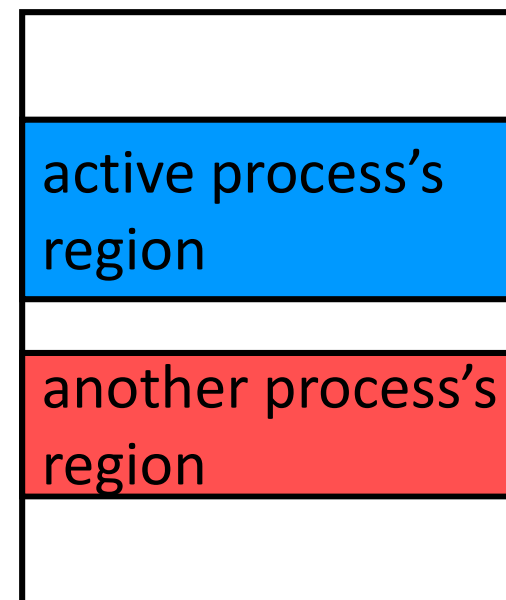
Base and Bound

- A process's private memory region defined by
  - **base**: starting address of region
  - **bound**: size of region

- User process issue "effective" address (**EA**) between 0 and the size of its allocated region (private and uniform)

privileged control registers
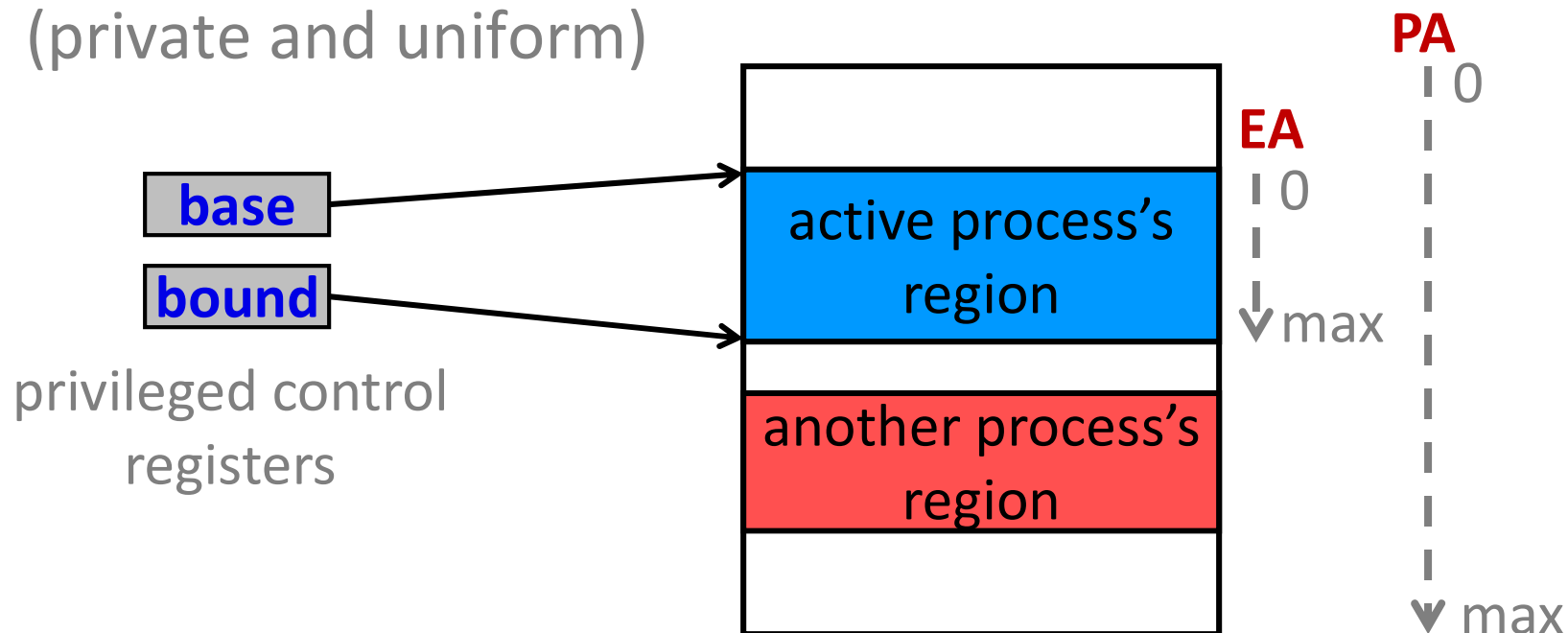
Base and Bound Registers

- Translation and protection check <u>in hardware</u> on <u>every</u> user memory reference

    - **PA** = **EA** + **base**

    - if (**EA** < **bound**) then okay else violation

- When switching user processes, OS sets **base** and **bound** registers

- User processes cannot be allowed to modify **base** and **bound** registers themselves

Requires at least 2 privilege levels with protected instruction and state for OS only

Segmented Memory

- Limitations of single base-and-bound region
  - hard to find large contiguous space after a while—free space become fragmented
  - can two processes shared some memory regions but not others?
- A "base-and-bound" pair is a unit of protection

  $\Rightarrow$ give user multiple memory "segments"
  - each segment is a contiguous memory region
  - each segment is defined by a **base** and **bound** pair
- Earliest use, separate code and data segments
  - 2 sets of **base**/**bound** for code vs data
  - forked processes can share code segments

more elaborate later: code, data, stack, etc.

Segmented Address

a few large segments good for managing separation
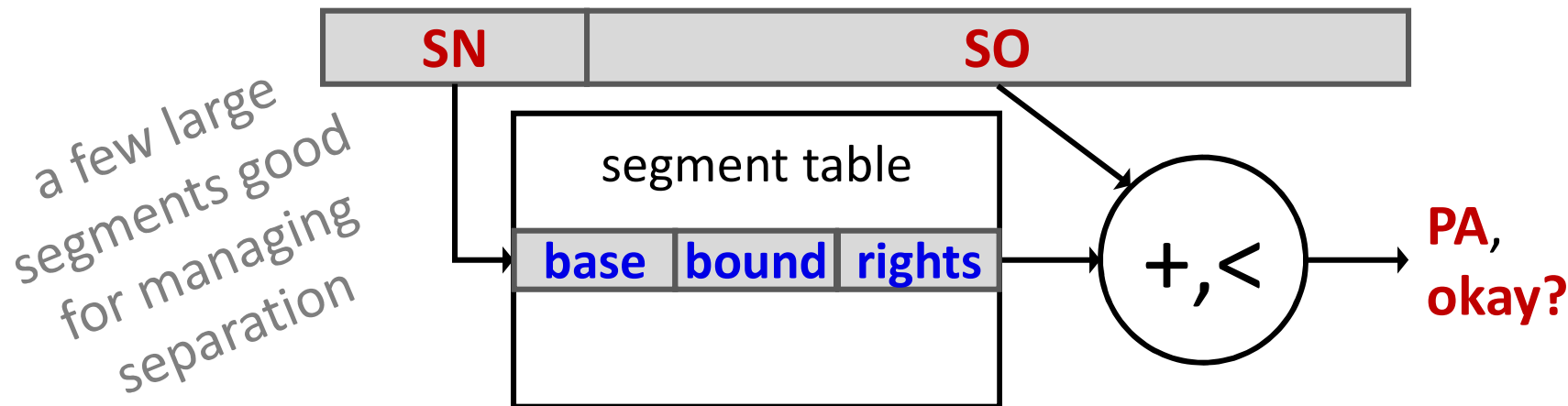
- **EA** partitioned into segment number (**SN**) and segment offset (**SO**)
  - – max segment size limited by the range of **SO**
  - – active segment size set by **bound**
- Per-process segment translation table
  - – map **SN** to corresponding **base** and **bound**
  - – separate mapping for each process
  - – privileged structure if used to enforce protection

Access Protection

- Per-segment access rights can be specified as protection bits in segment table entries
- Generic options include
  - **<u>R</u>eadable?**
  - **<u>W</u>riteable?**
  - **<u>E</u>xecutable?**

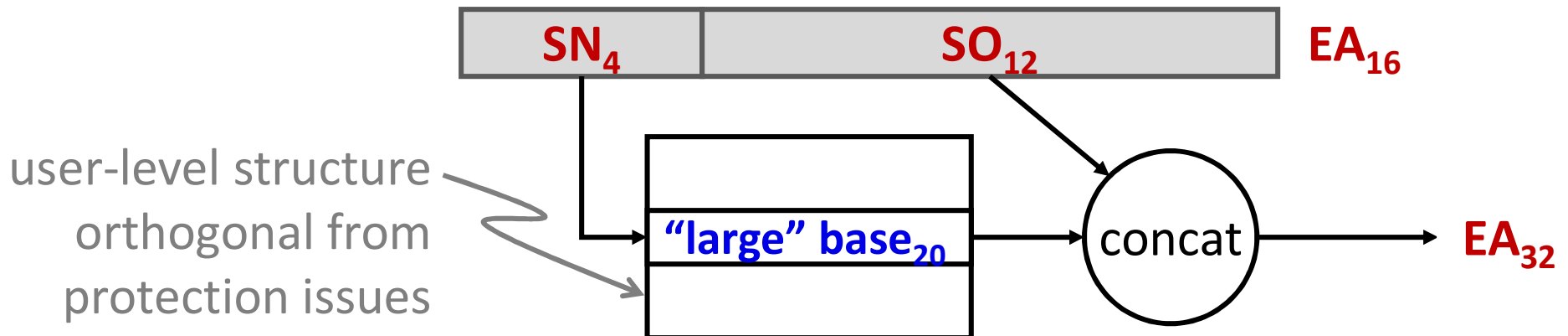 *also misc. options, such as "cacheable"*

- For example
  - normal data segment $\Rightarrow$ **RW**(!**E**)
  - static shared data segment $\Rightarrow$ **R**(!**W**)(!**E**)
  - code segment $\Rightarrow$ **R**(!**W**)**E**   *self modifying code?*
  - illegal segment $\Rightarrow$ (!**R**)(!**W**)(!**E**)   *what for?*

  Access violation exception brings OS into play

Aside: Another (ab)use of segments

- Extend old ISA to give new applications a large address space while stay compatible with old

- "User-managed" segmented addressing $SA \equiv EA_{small}$
  - old application use identity mapping in table; old applications unaware of segments
  - new application reloads table at run time to access different regions in EA_{large}; unequal access to active vs inactive regions

| SN_4 | SO_{12} | EA_{16} |

user-level structure orthogonal from protection issues

"large" base$_{20}$

concat → EA_{32}

Paged Address Space

- Divide **PA** and **EA** space into equal, fixed size segments known as "page frames"

historically 4KByte pages

- **EA** and **PA** are interpreted as page number (**PN**) and page offset (**PO**)
  - page table translates **EPN** to **PPN**; **EPO**=**PPO**
  - **PA**={**PPN**,**PO**}

EPN	PO

many small pages good for managing allocation

page table → **PPN rights?** → concat → **PA**
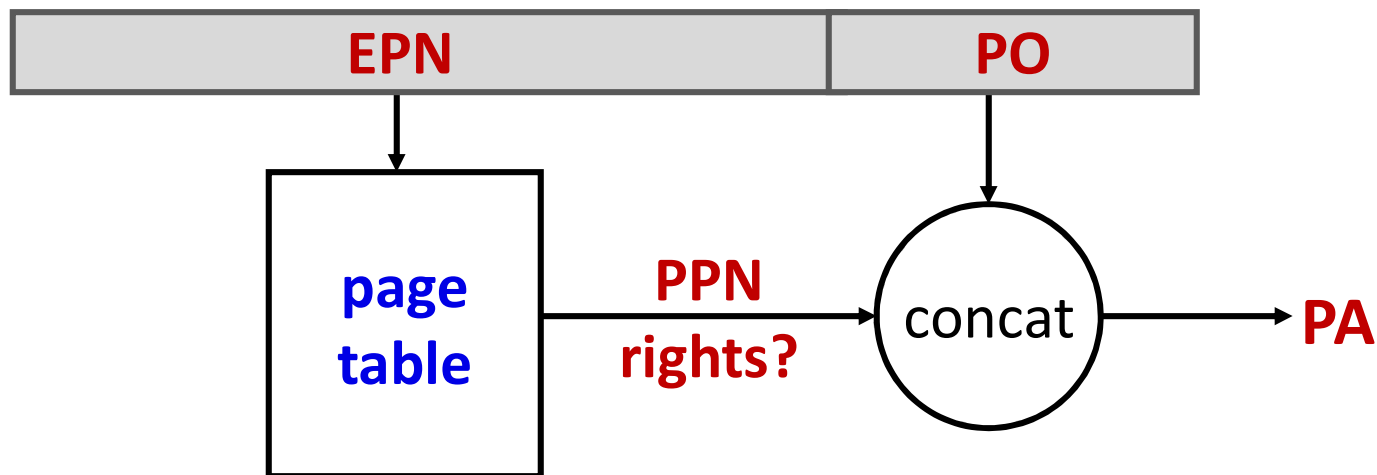
Fragmentation

- External fragmentation by segments
  - plenty of unallocated DRAM but none in contiguous region of a sufficient size
  - paged memory eliminates external fragmentation
- Internal fragmentation of pages
  - entire page (4KByte) is allocated; unused bytes go to waste
  - smaller page size reduces internal fragmentation
  - modern ISA moving to larger page sizes (MBytes) in addition to 4KBytes

Segments and pages not meant for the same role

Demand Paging

- Use main memory and "swap" disk as <u>automatically managed</u> memory hierarchy levels

  analogous to cache vs. main memory

- Early attempts
  - von Neumann already described <u>manual</u> memory hierarchies
  - Brookner's interpretive coding, 1960:

    program interpreter managed paging between a 40KByte main memory and a 640KByte drum

  - Atlas, 1962:

    hardware managed paging between 32-page core memory and 192-page drum (512 word/page)

Demand Paging: just like caching

- **M** bytes of storage (DRAM+Disk), keep most frequently used **C** bytes in DRAM where **C** < **M**

- Same basic issues as before

    (1) where to place a page in DRAM or disk?

    (2) how to find a page in DRAM or disk?

    (3) when to bring a page into DRAM from disk?

    (4) which page to evict from DRAM to disk to free-up DRAM for new pages?

- Key conceptual difference: swap vs. cache

    – DRAM doesn't hold "copies" of what is on disk

    – a page in **M** either in DRAM or disk (or non-existent) *"virtual"*

    – address not bound to 1 location for all time
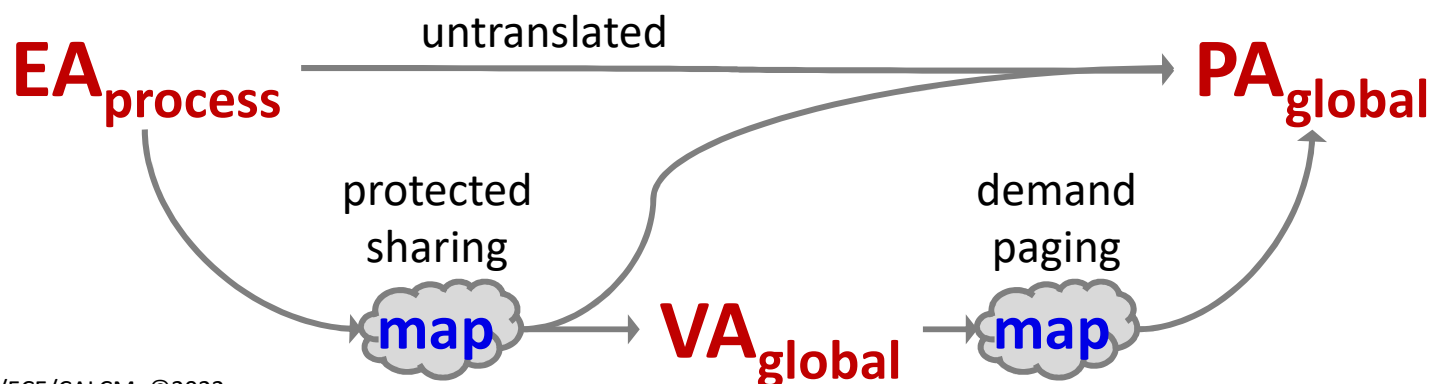
Demand Paging: not at all like caching

- Drastically different size and time scale leads to drastically different implementation choices

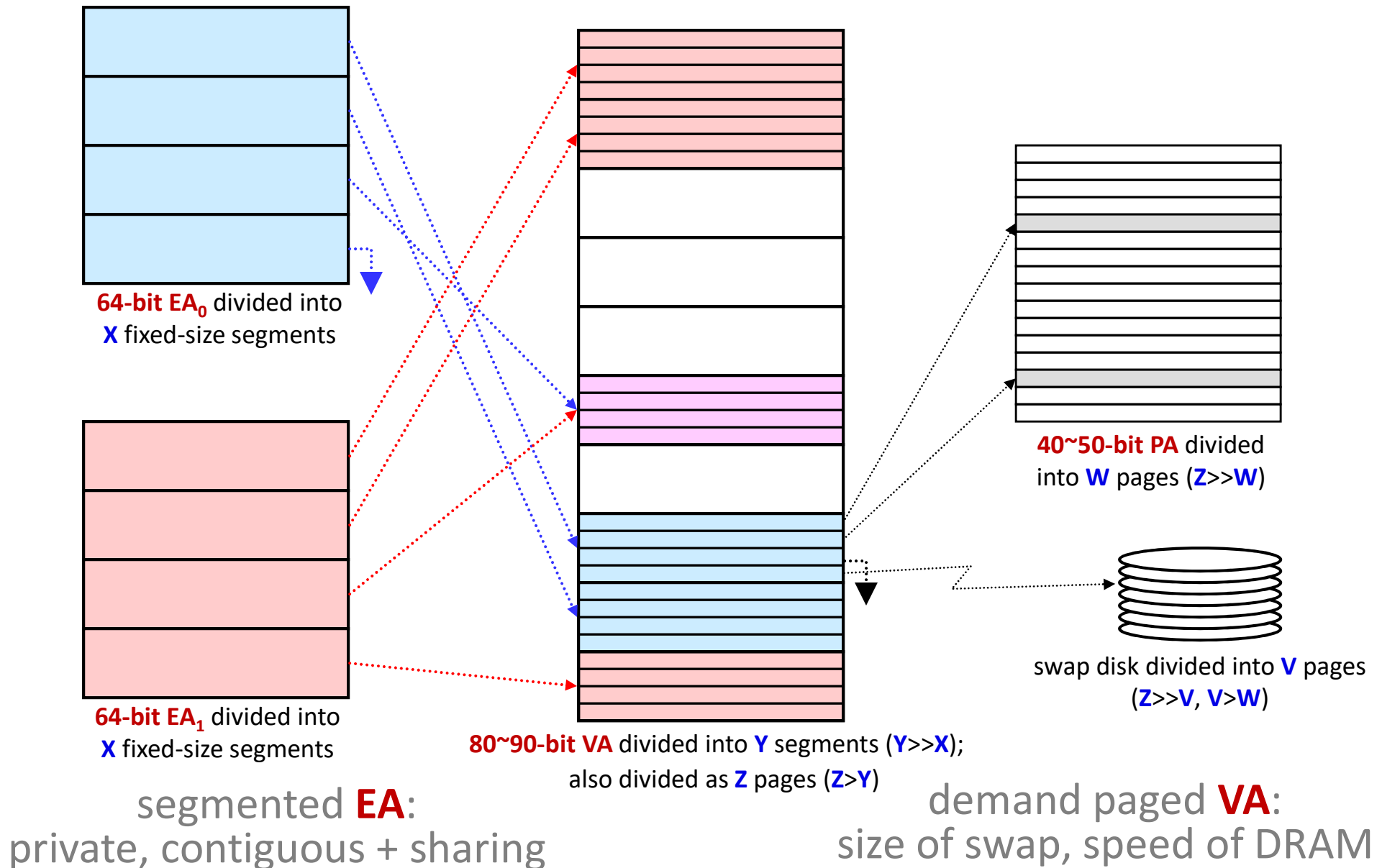|	**L1 Cache**	**L2 Cache**	**Demand Paging**
capacity	10s KByte	MByte	GByte
block size	10s Byte	$\geq$ L1	4K~4M Byte
hit time	few cyc	few 10s cyc	few 100s cyc
miss penalty	few 10s cyc	few 100s cyc	**10 msec**
miss rate	0.1~10%	<<0.1%	**0.00001~0.001%**
		(per mem reference not per cache access)	
hit handling	HW	HW	HW
miss handling	HW	HW	**SW**

Hit time, miss penalty, miss rate not independent!!

Don't use "VM" to mean everything

- Effective Address (**EA**): emitted by user instructions in a **_per-process_** space (protection)

- Physical Address (**PA**): corresponds to actual storage locations on DRAM or on swap disk

- Virtual Address (**VA**): refers to locations in a **_single system-wide_**, large, linear address space; not all locations in **VA** space have physical backing (demand paging)

$$EA_{process} \xrightarrow{\text{untranslated}} PA_{global}$$

protected sharing → **map** → VA_{global} → demand paging → **map** → PA_{global}

EA, VA and PA (IBM Power view)

64-bit EA_0 divided into X fixed-size segments

64-bit EA_1 divided into X fixed-size segments

80~90-bit VA divided into **Y** segments (**Y>>X**); also divided as **Z** pages (**Z>Y**)

40~50-bit PA divided into **W** pages (**Z>>W**)

swap disk divided into **V** pages (**Z>>V, V>W**)

segmented **EA**:
private, contiguous + sharing

demand paged **VA**:
size of swap, speed of DRAM

EA, VA and PA (almost everyone else)

EA$_0$
with unique **ASID**=0

EA$_i$
with unique **ASID**=i

Easy to blur **EA** and **VA** colloquially but full **VA** is **{ASID, EA}**!!!

VA divided into **N** "address spaces" indexed by **ASID**; also divided as **Z** pages (**Z>>N**)

PA divided into **W** pages (**Z>>W**)

swap disk divided into **V** pages (**Z>>V**, **V>>W**)

how do processes share pages?
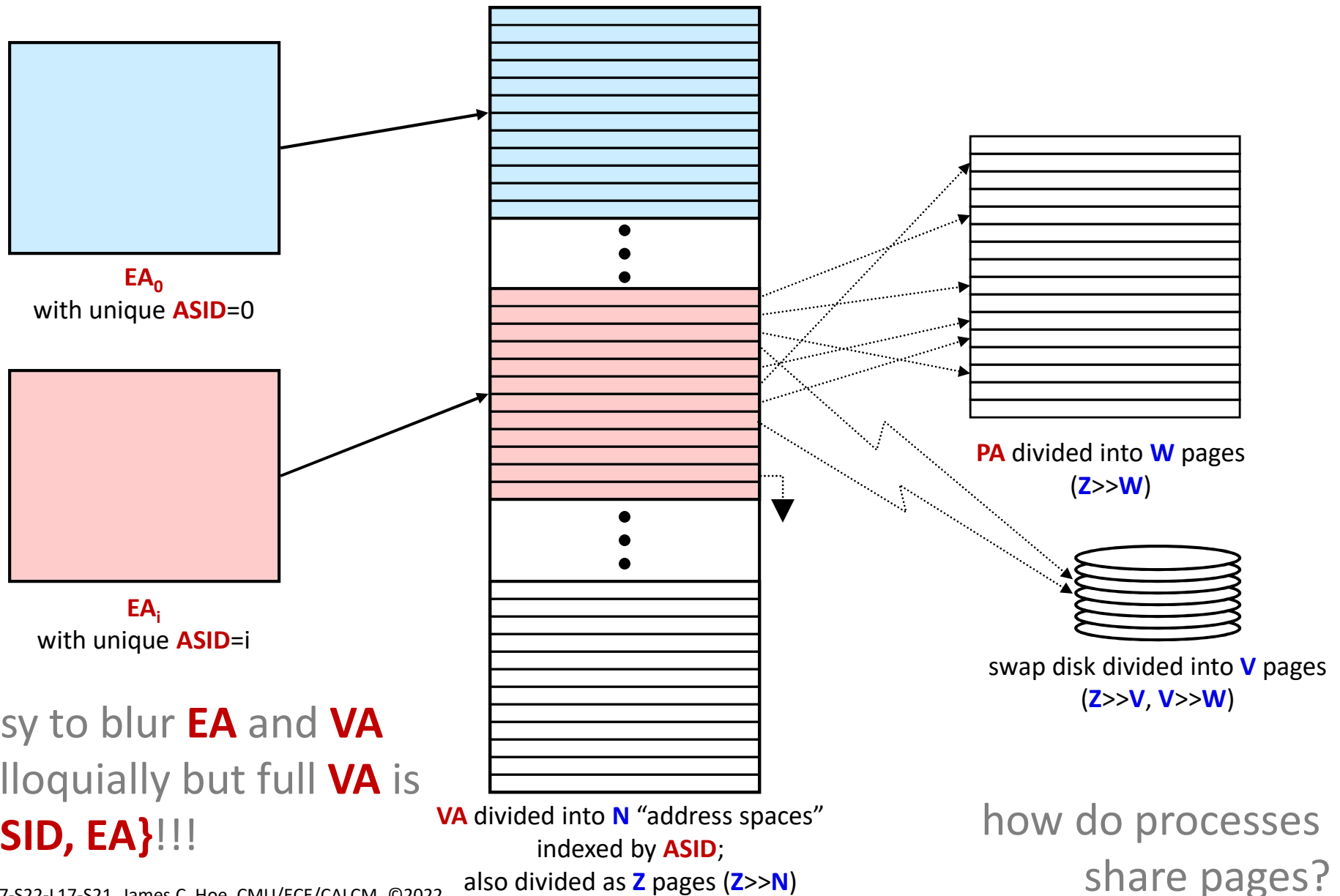
Just one more thing:
How large is the page table?
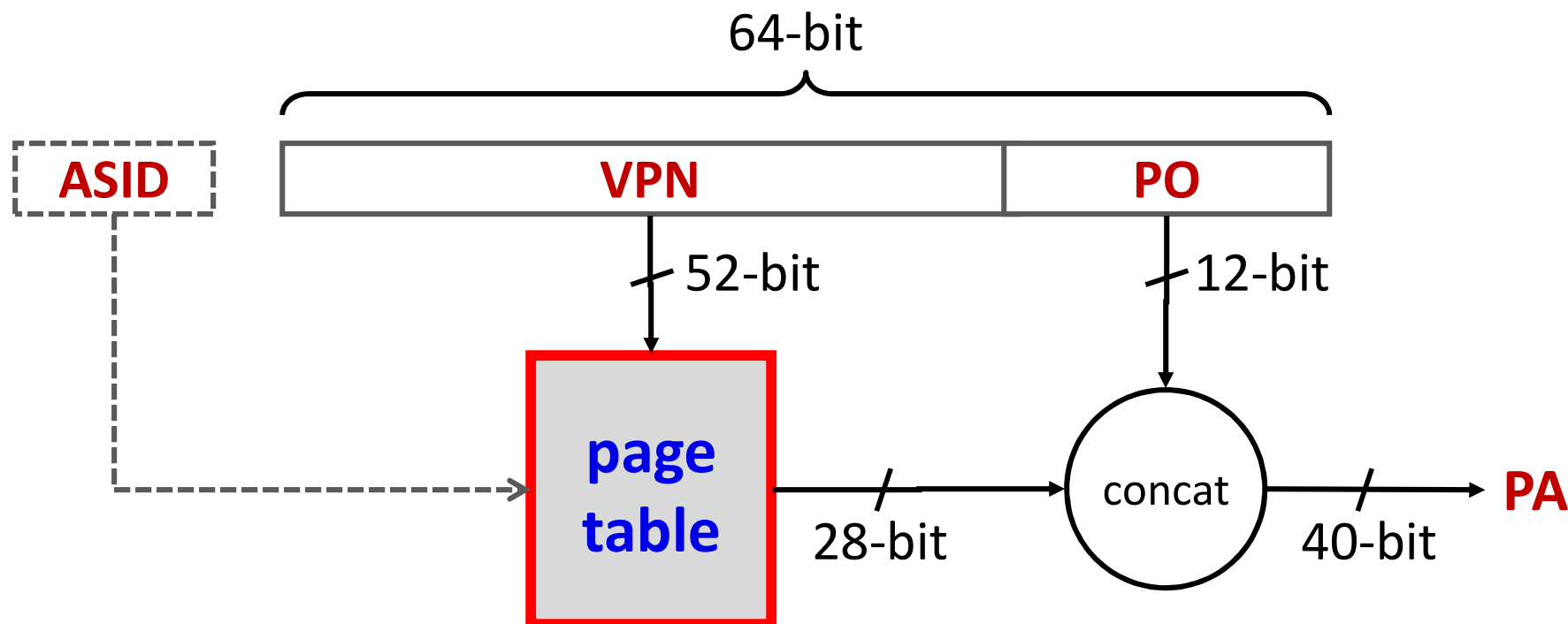
- A page table holds mapping from **VPN** to **PPN**
- Suppose 64-bit **VA** and 40-bit **PA**, how large is the page table? 2^{52} entries x ~4 bytes $\approx$ 16x10^{15} Bytes

And that is for just one process!!?

How large should it be?

- Don't need to track entire **VA** space
  - total allocated **VA** *space* is 2^{64} bytes x # processes, but most of which not backed by *storage*
  - can't use more memory locations than physically exist (DRAM and swap disk)
- A clever page table should scale linearly with physical *storage* size and not **VA** *space* size
- Table cannot be too convoluted
  - a page table is accessed not infrequently
  - a page table should be "walkable" quickly in HW

Two dominant schemes in use today:
hierarchical page table and *hashed page table*