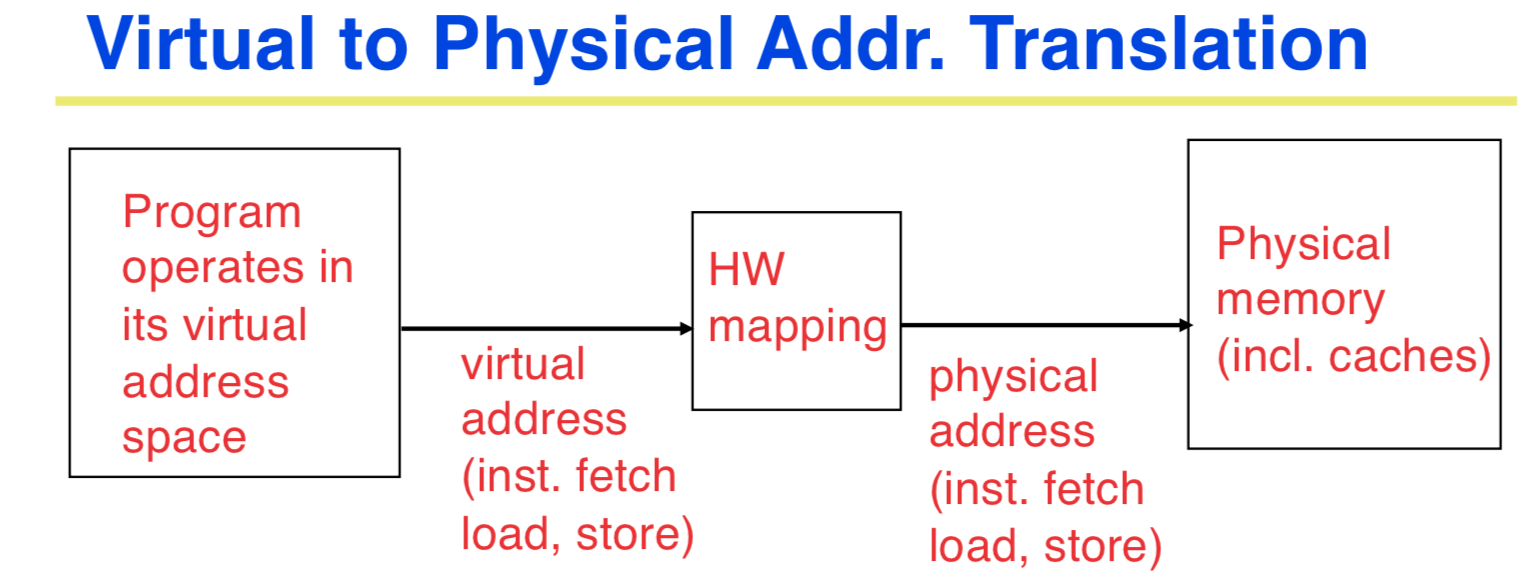
**Virtual Memory**

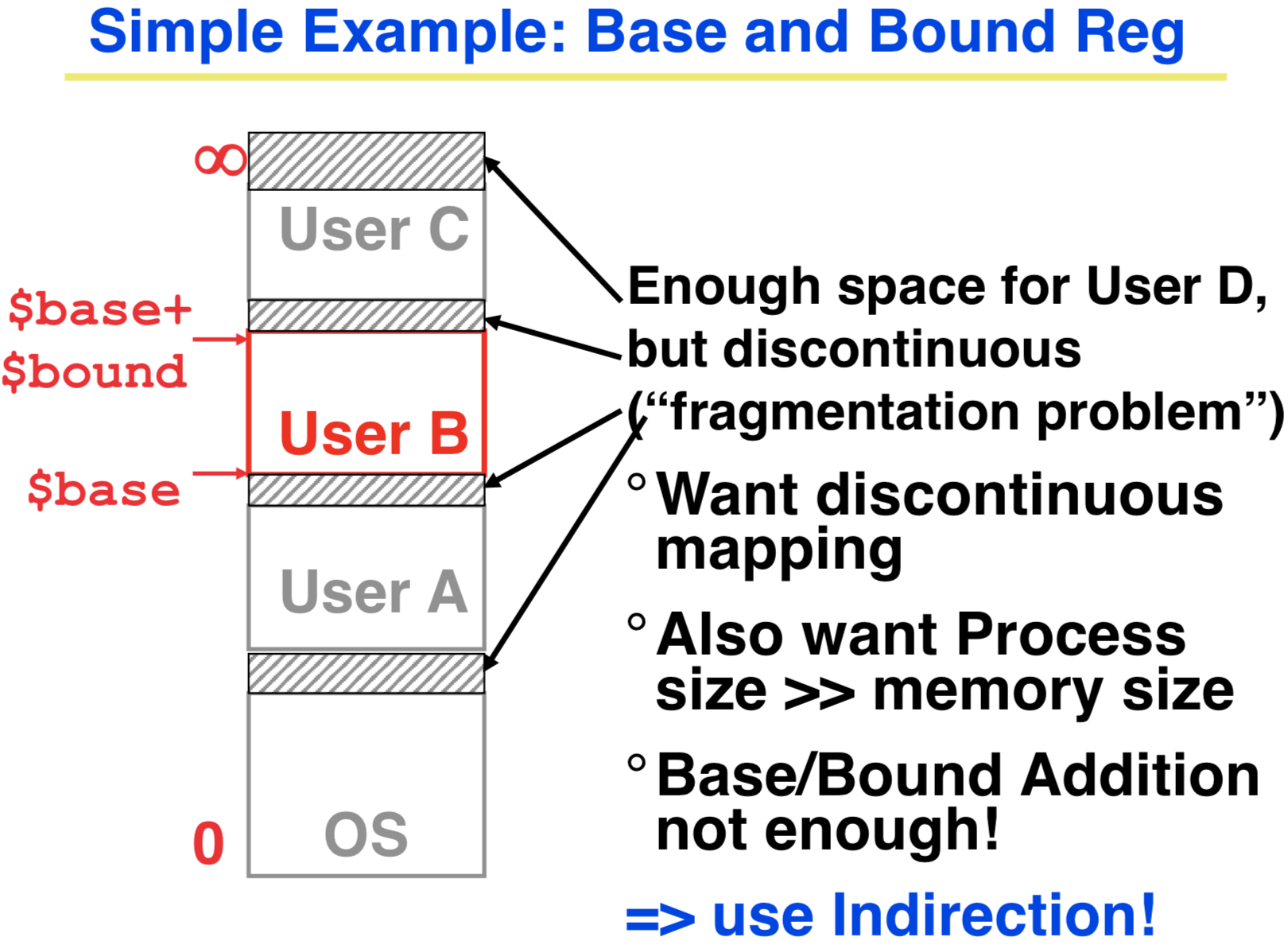
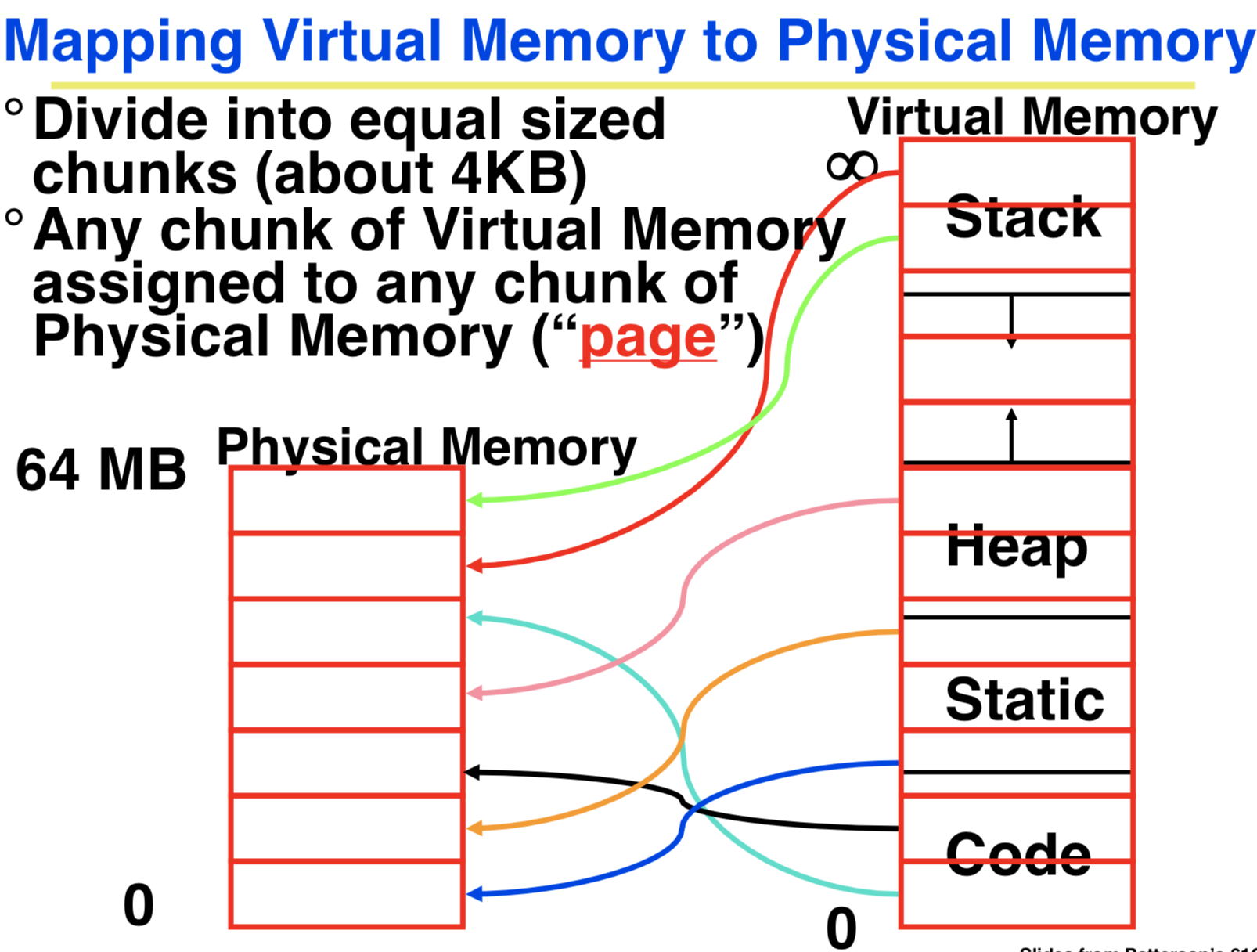


Each program operates in its own virtual address space; ~only program running

Each “process” is protected from the other

OS can decide where each goes in memory

Hardware (HW) provides virtual 🡪 physical mapping

**[Virtual Memory Mapping 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

* Physical Offset = Virtual Offset
* Physical Page Number = Page Table[Virtual Page Number]

(P.P.N. also called “Page Frame”)

**[Page Table]**

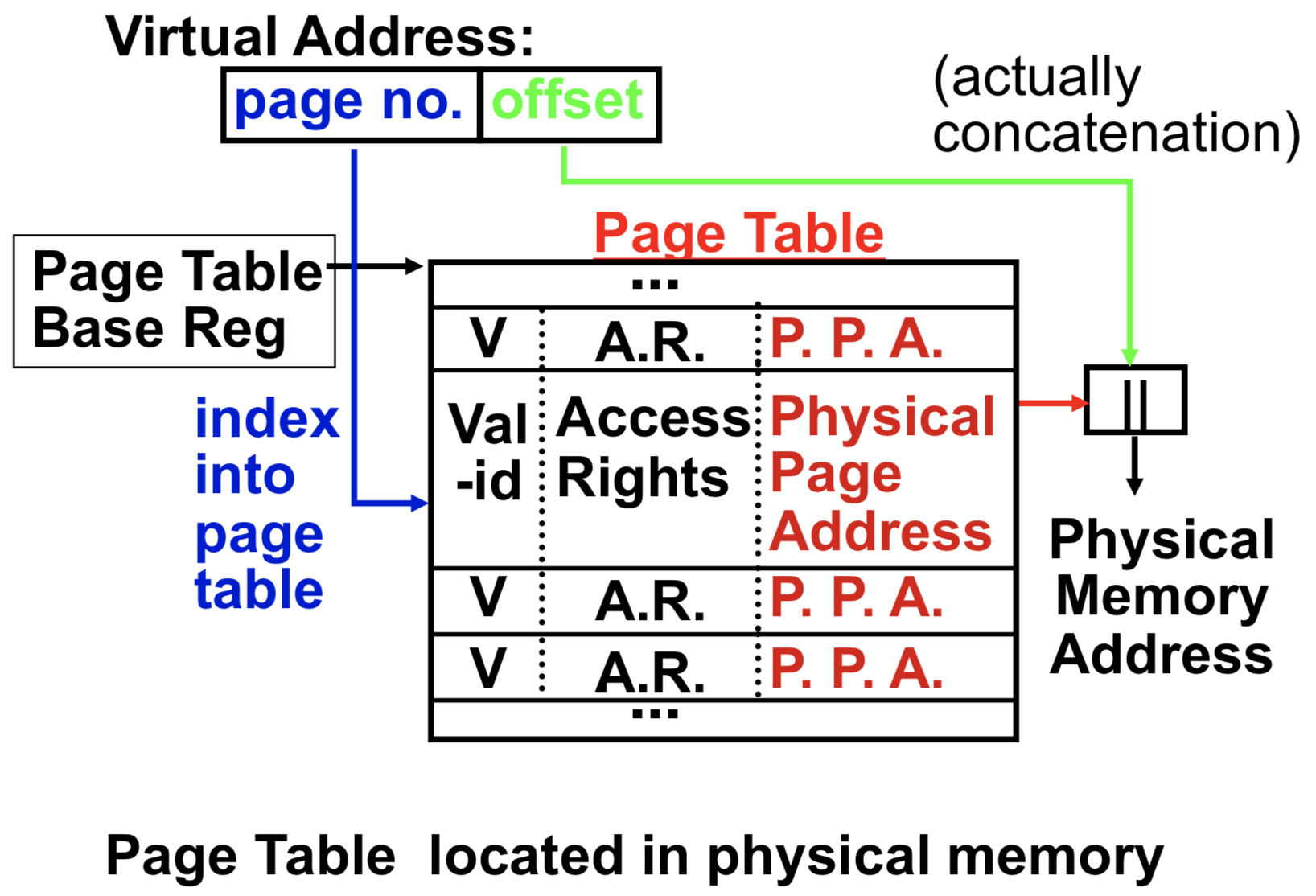
*A page table is an operating system structure which contains the mapping of virtual addresses to physical locations*

• 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

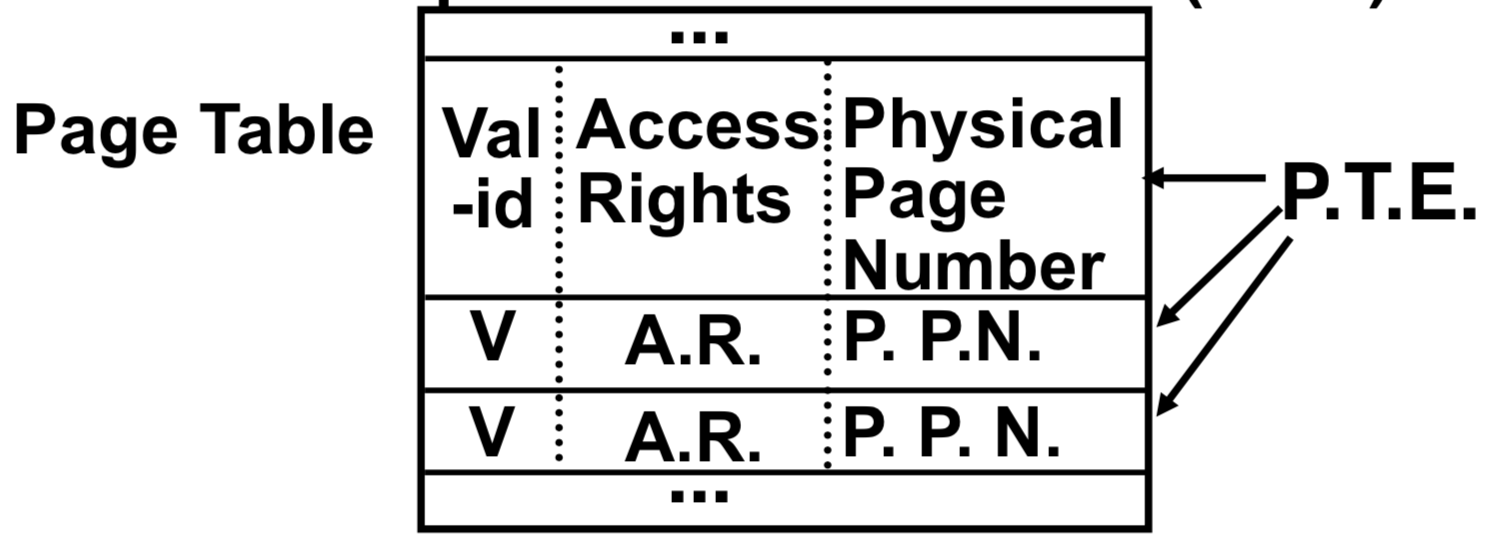
• “State” of process is PC, all registers, plus page table

• OS changes page tables by changing contents of Page Table Base Register



**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, also check if have 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

Analogy

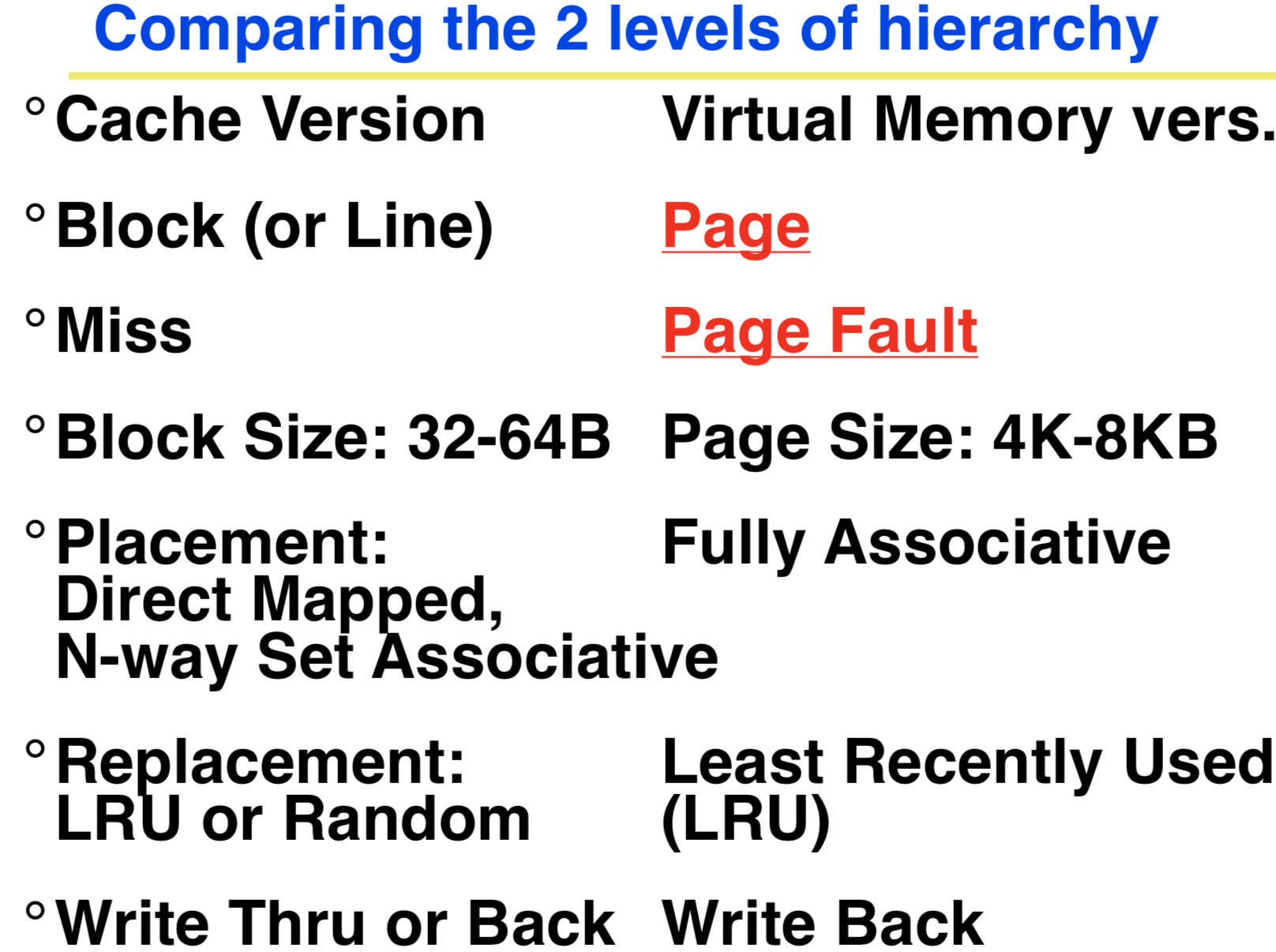
Book title-----virtual address

Library of Congress call number-----physical address

Card catalogue-----page table, mapping from book title to call number

On card for book, in local library vs. in another branch----valid bit indicating in main memory vs. on disk

On card, available for 2-hour in library use (vs. 2-week checkout)----access rights



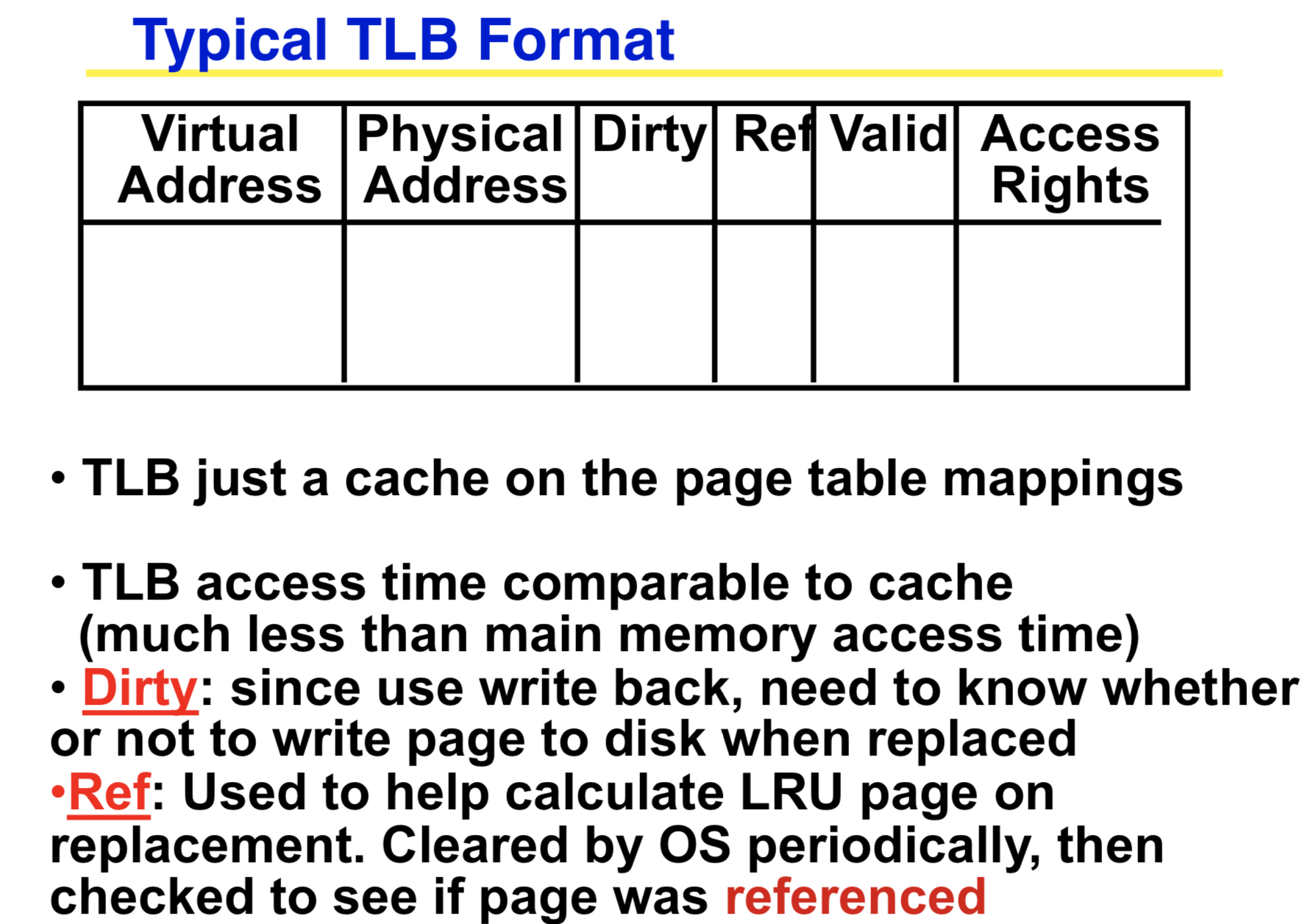
**Virtual Memory Problem 1**

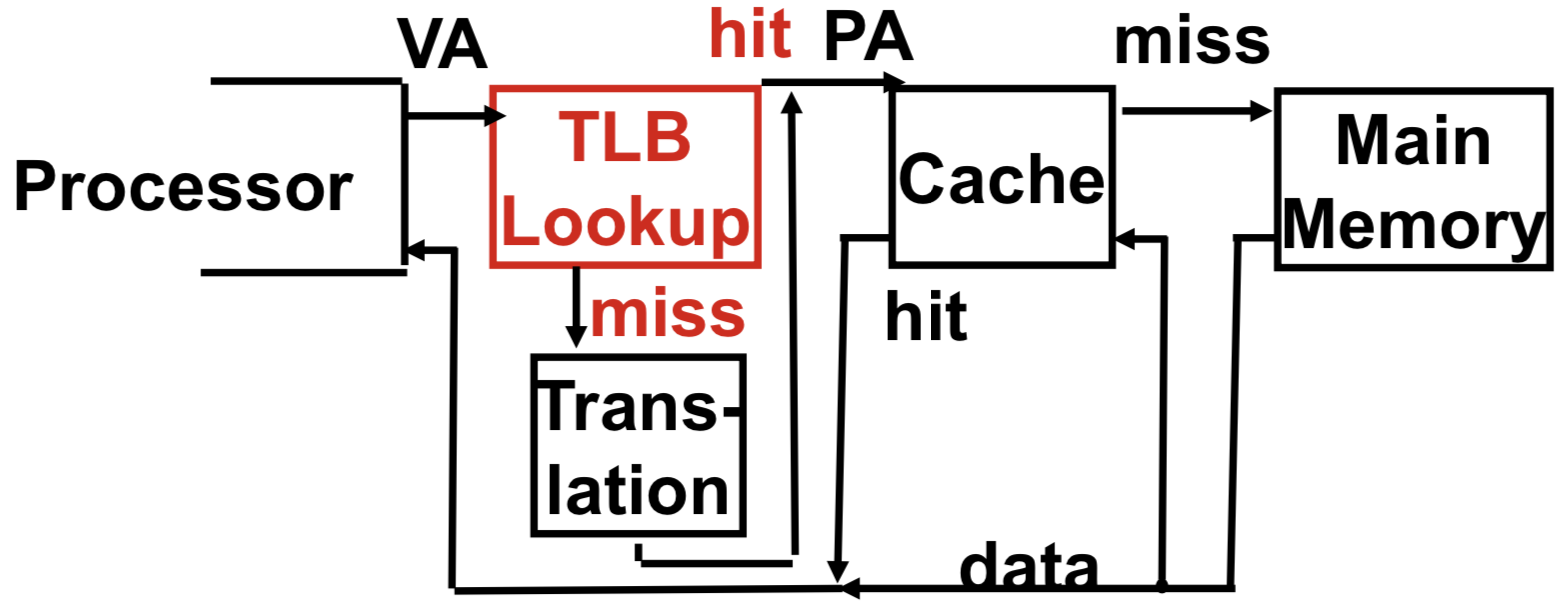
Map every address 🡪 1 indirection via Page Table in memory per virtual address 🡪 1 virtual memory access = 2 physical memory accesses 🡪 SLOW!

Observation: since locality in pages of data, there must be locality in **virtual address translations** of those pages

Since small is fast, use a small cache (***Translation Lookaside Buffer, or TLB***) of virtual to physical address translations to make translation fast.

TLBs usually small, typically 128 - 256 entries

Like any other cache, the TLB can be direct mapped, set associative, or fully associative



TLB Miss (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

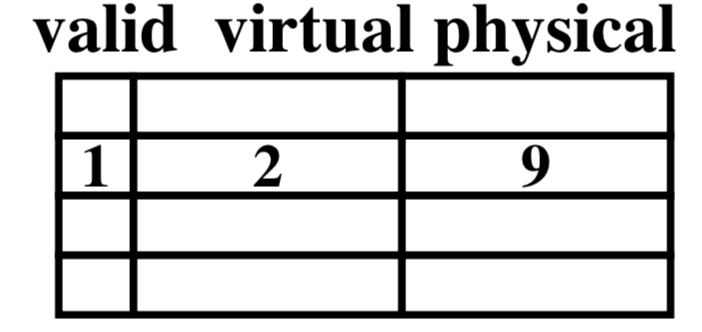
MIPS follows Option 2: Hardware knows nothing about page table

**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)

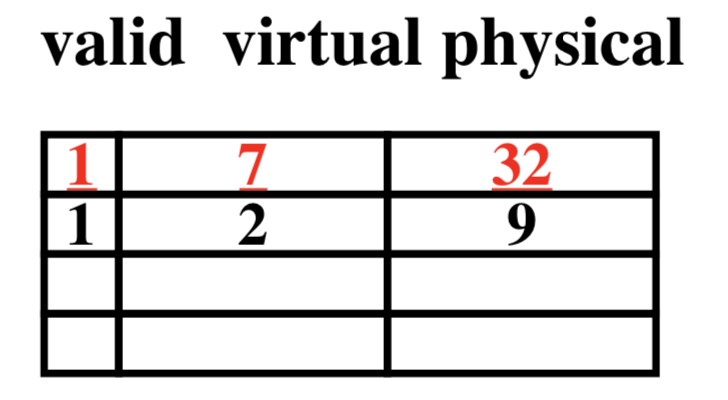
The operating system knows which program caused the TLB fault, page fault, and knows what virtual address was requested

* So we look the data up in the page table



* **If the data is in memory**

We simply add the entry to the TLB, evicting an old entry from the TLB



* **If the data is in 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 update the process's page table

So when we switch back to the task, the desired data will be in memory

* **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)

And update that program's page table to reflect the fact that its memory moved somewhere else

**Virtual Memory Problem 2**

* **Not enough physical memory!**
* Only, say, 64 MB of physical memory
* N processes, each 4 GB (232 B) of virtual memory!
* Could have 1K virtual pages/physical page!
* **Spatial Locality to the rescue**
* Each page is 4 KB, lots of nearby references
* **No matter how big program is, at any time only accessing a few pages**
* ***“Working Set”:*** *recently used pages*

**Virtual Memory Problem 3**

* **Page Table too big!**

4GB Virtual Memory ÷ 4 KB page

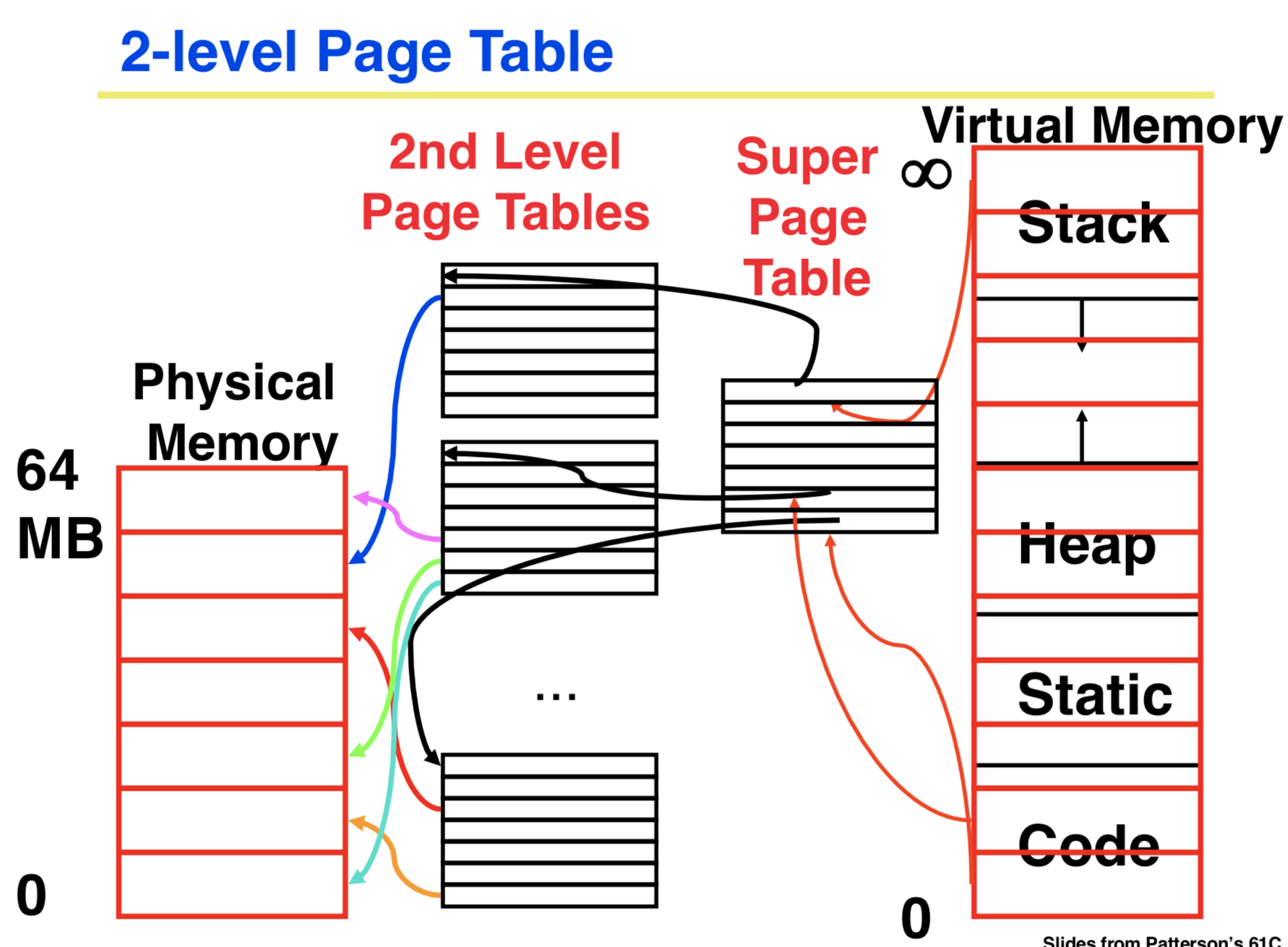
🡪 ~ 1 million Page Table Entries

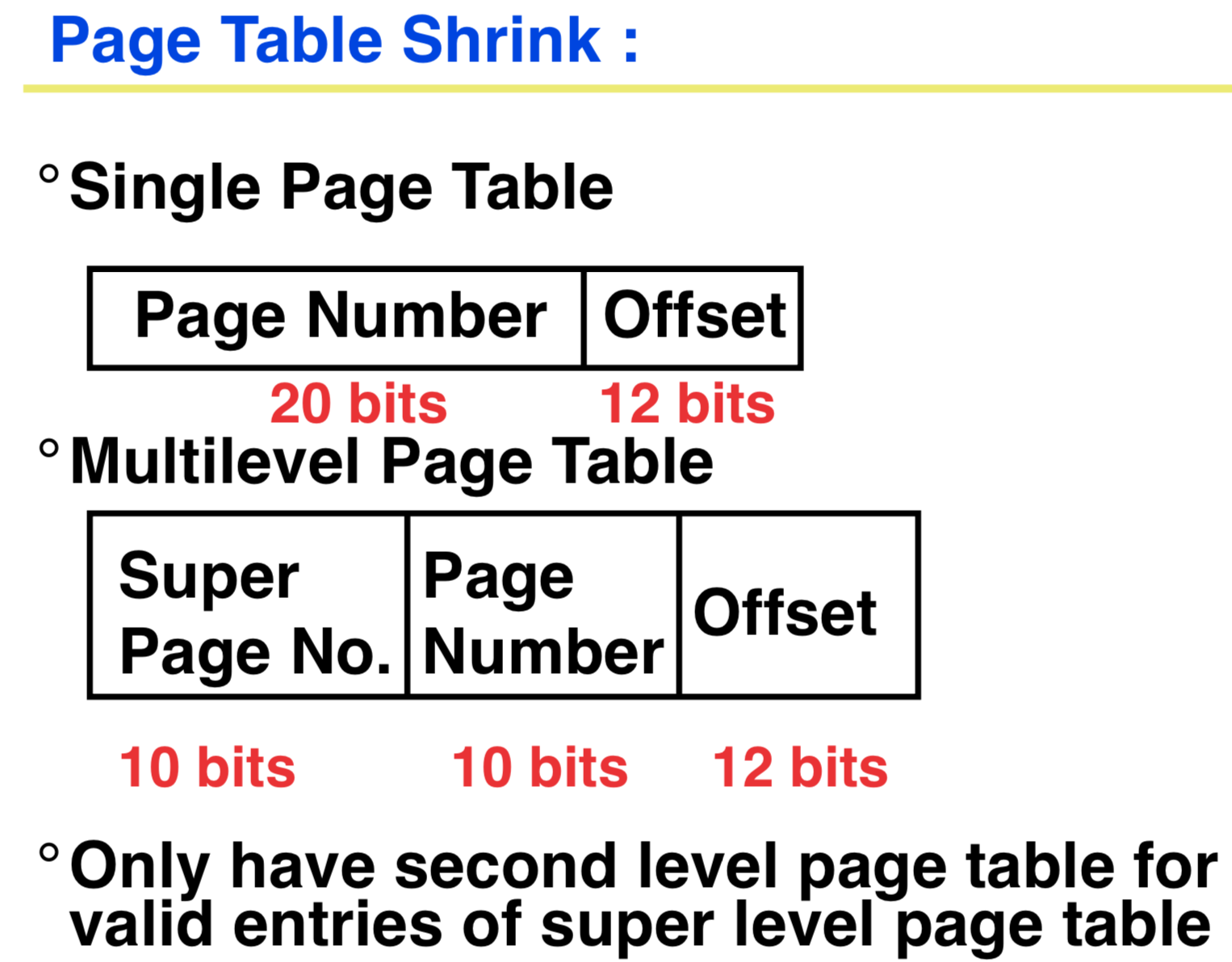
🡪 4 MB just for Page Table for 1 process

🡪 25 processes 100 MB for Page Tables!

* **Variety of solutions to tradeoff memory size of mapping function for slower when miss TLB**

Make TLB large enough, highly associative so rarely miss on address translation





Space Savings for Multi-Level Page Table

If only 10% of entries of Super Page Table have valid entries, then total mapping size is roughly 1/10-th of single level page table