COMP-273
Virtual Memory

Kaleem Siddiqi

Slides from Patterson's 61C

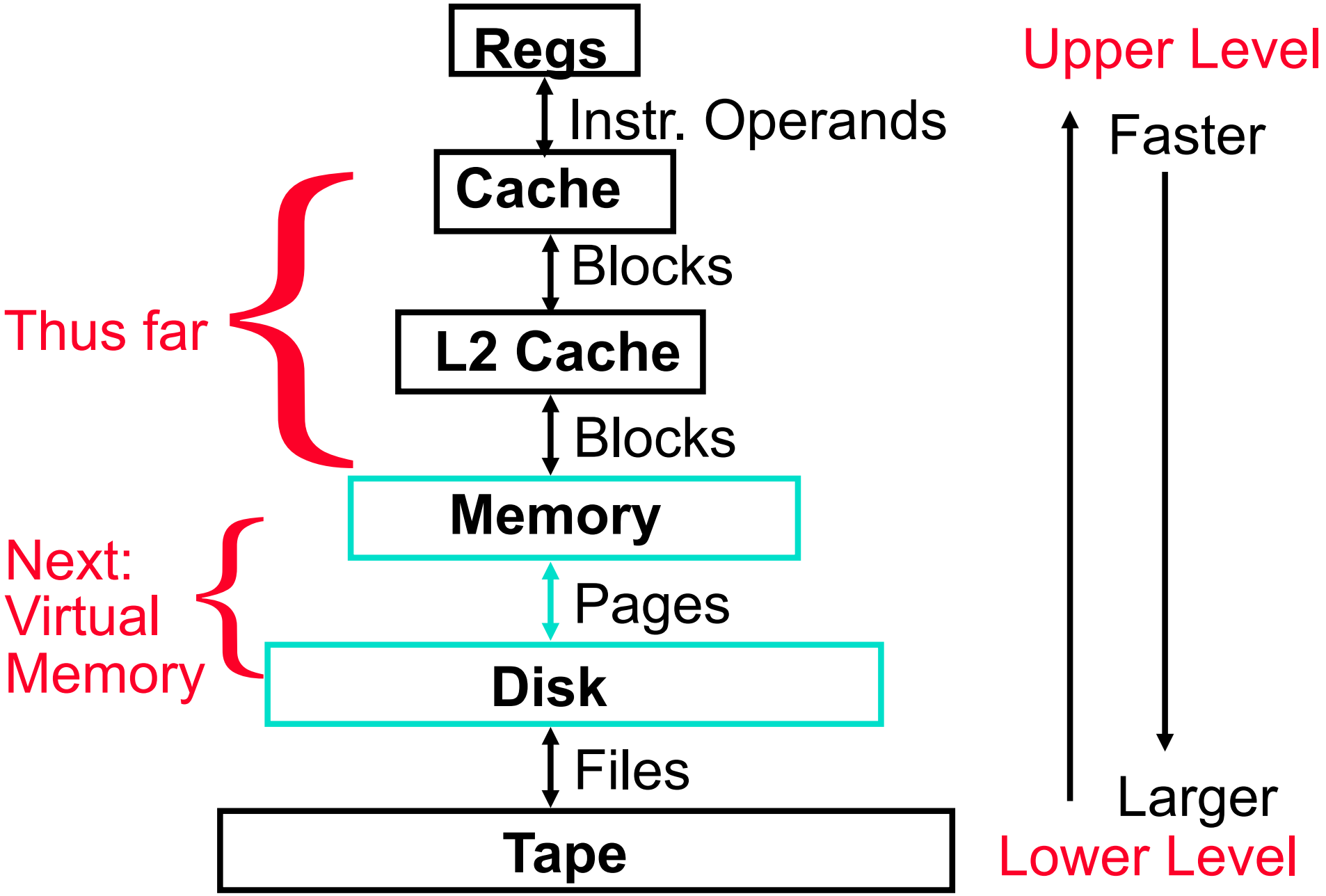
Review (1/2)

- 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 higher 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



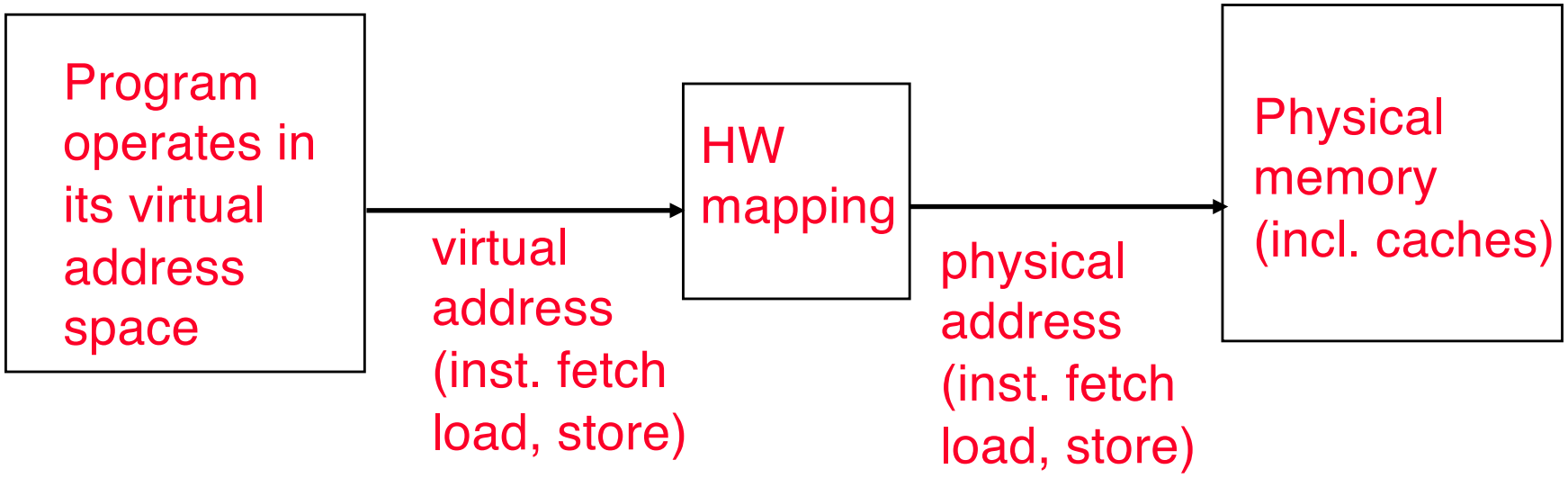
Slides from Patterson's 61C

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?
- 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
  - Historically, it predates caches

Slides from Patterson's 61C

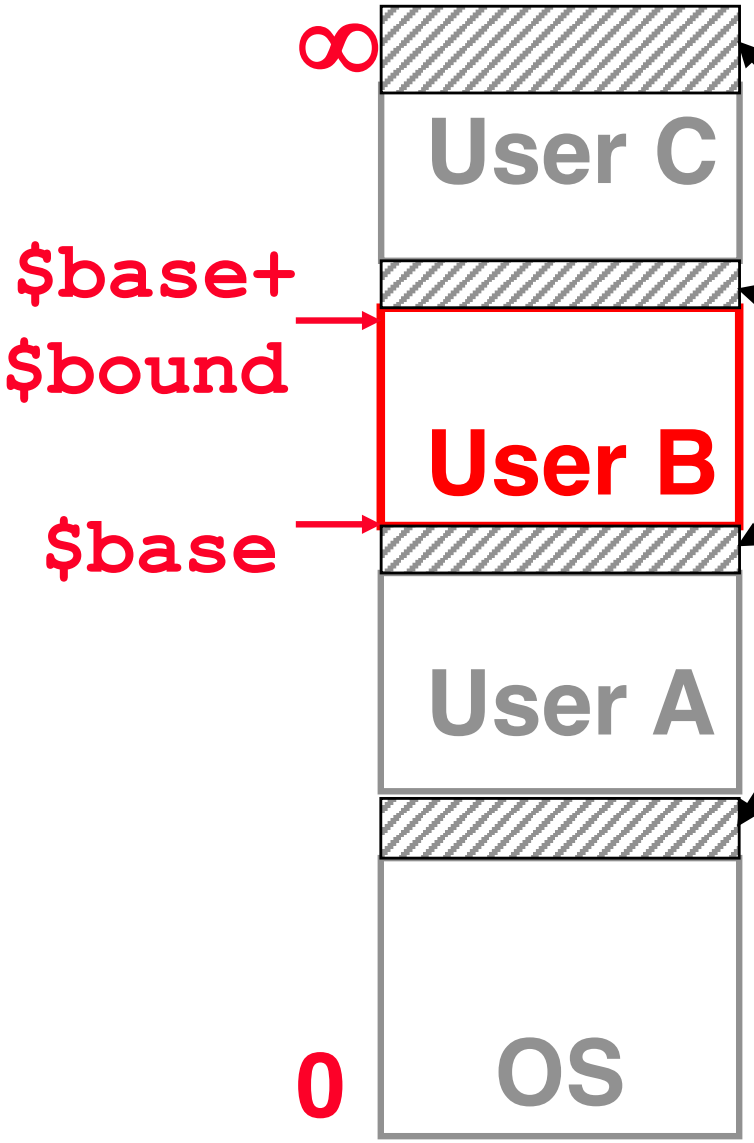
Virtual to Physical Addr. Translation



- 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

Slides from Patterson's 61C

Simple Example: Base and Bound Reg



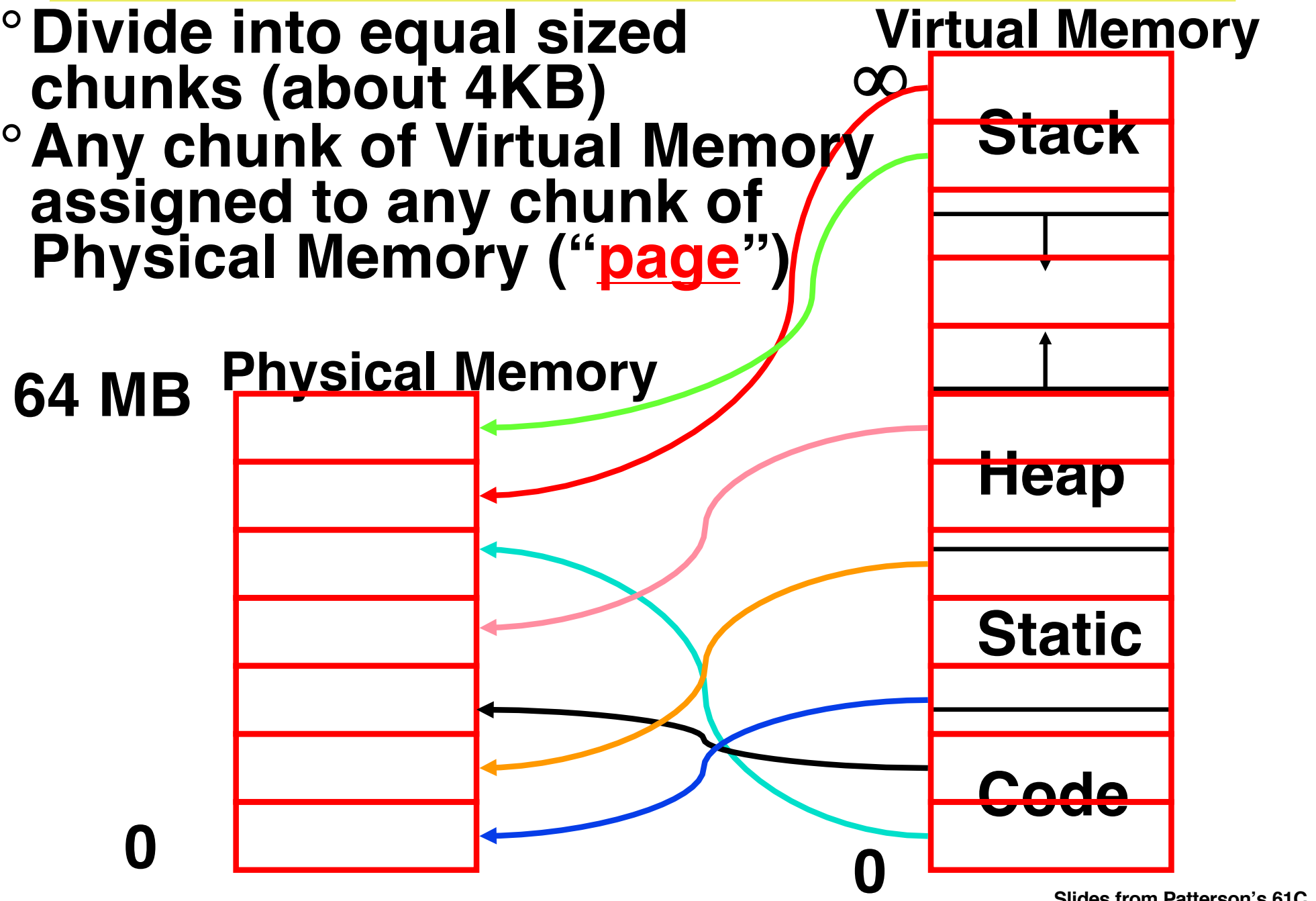
- Enough space for User D, but discontinuous ("fragmentation problem")
  - Want discontinuous mapping
  - Also want Process size >> memory size
  - Base/Bound Addition not enough!

=> use Indirection!

Slides from Patterson's 61C

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")

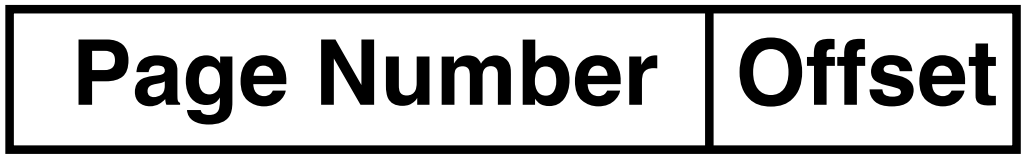


Slides from Patterson's 61C

Virtual Memory Mapping Function

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

Virtual address:



- 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")

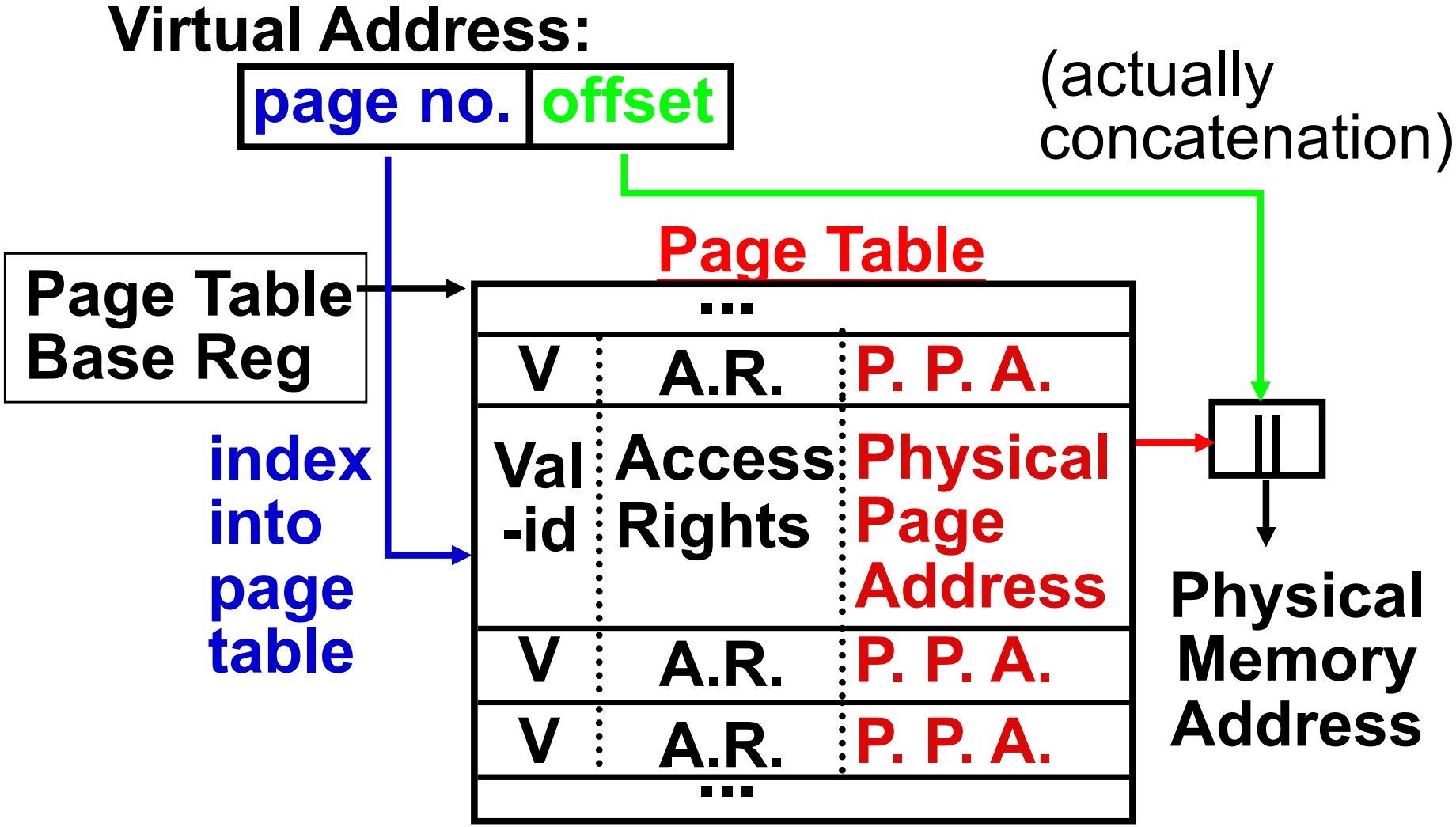
Slides from Patterson's 61C

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

Slides from Patterson's 61C

Address Mapping: Page Table

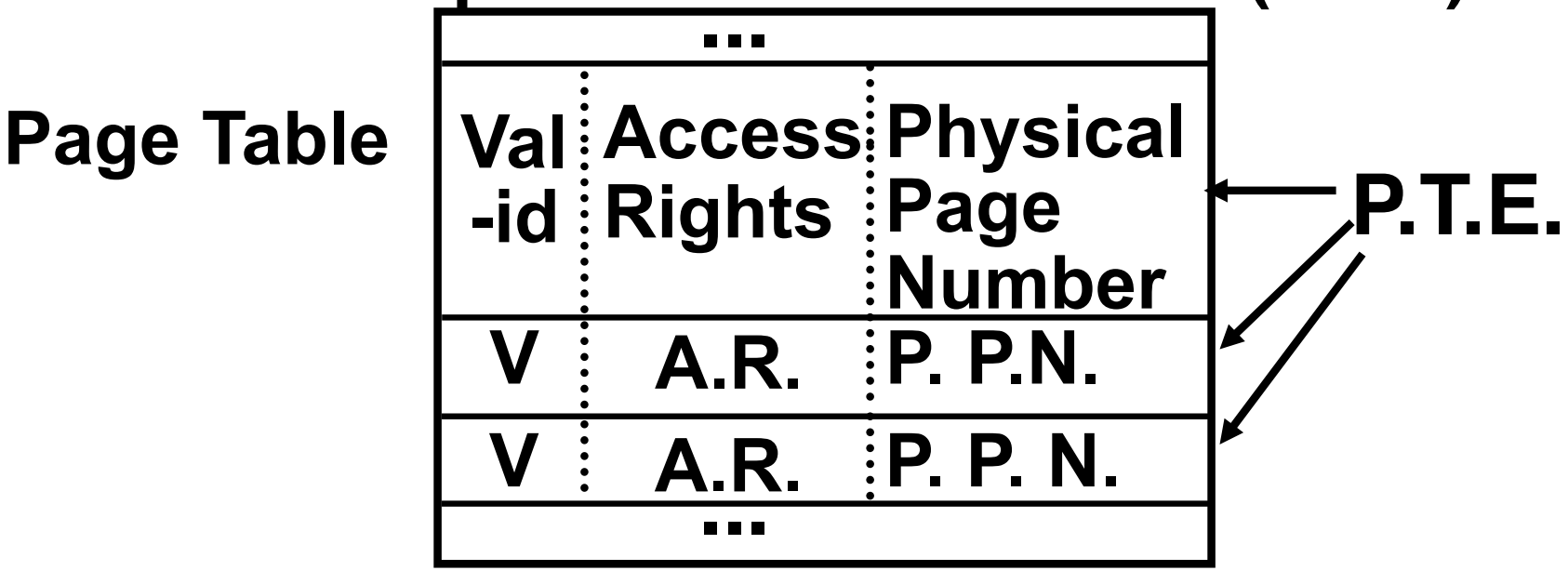


Page Table located in physical memory

Slides from Patterson's 61C

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, 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 like virtual address
- Library of Congress call number like physical address
- Card catalogue like page table, mapping from book title to call number
- On card for book, in local library vs. in another branch like valid bit indicating in main memory vs. on disk
- On card, available for 2-hour in library use (vs. 2-week checkout) like access rights

Comparing the 2 levels of hierarchy

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

Virtual Memory Problem #1

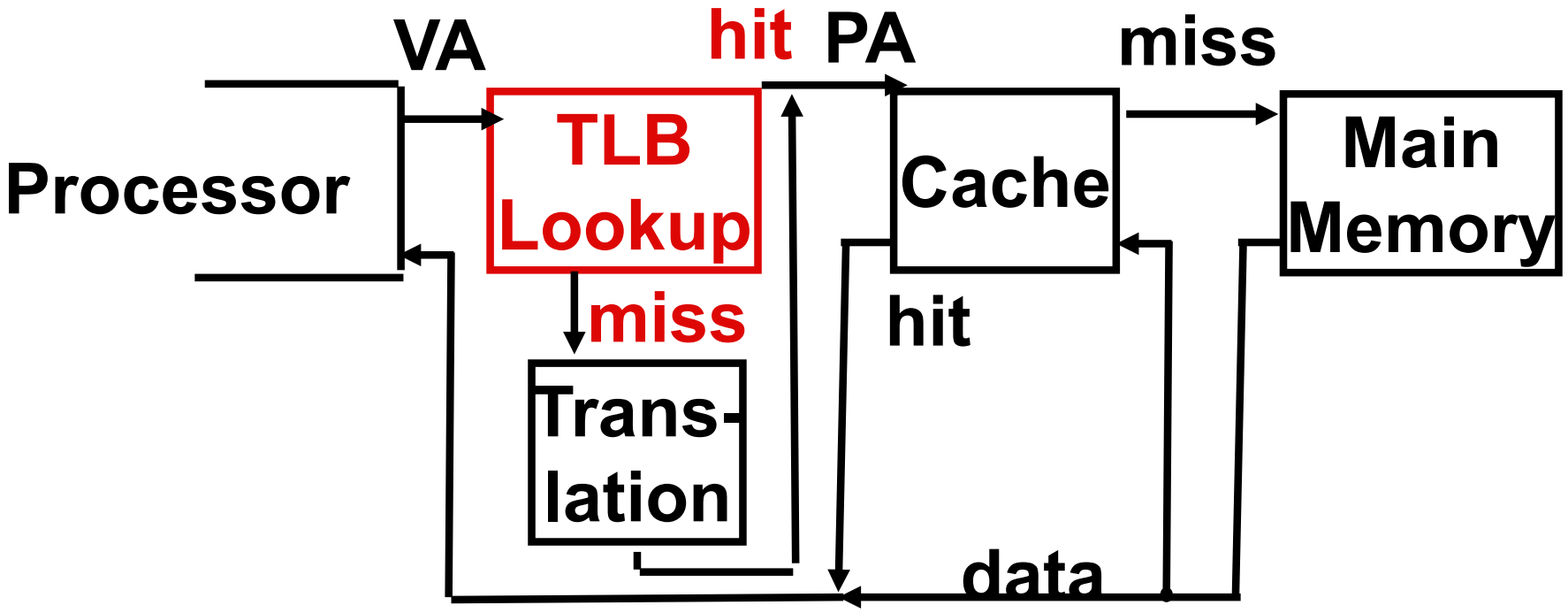
- Map every address ⇒ 1 indirection via Page Table in memory per virtual address ⇒ 1 virtual memory access = 2 physical memory accesses ⇒ SLOW!
- Observation: since locality in pages of data, there must be locality in virtual address translations of those pages
- Since small is fast, why not use a small cache of virtual to physical address translations to make translation fast?
- For historical reasons, cache is called a Translation Lookaside Buffer, or TLB

Slides from Patterson's 61C

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



Slides from Patterson's 61C

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 use write back, need to know whether or not to write page to disk when replaced
- Ref: Used to help calculate LRU page on replacement. Cleared by OS periodically, then checked to see if page was referenced

Slides from Patterson's 61C

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

TLB Miss (simplified format)

- 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

valid	virtual	physical
1	2	9

If the data is in memory

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

valid	virtual	physical
1	7	32
1	2	9

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

Virtual Memory Problem #2

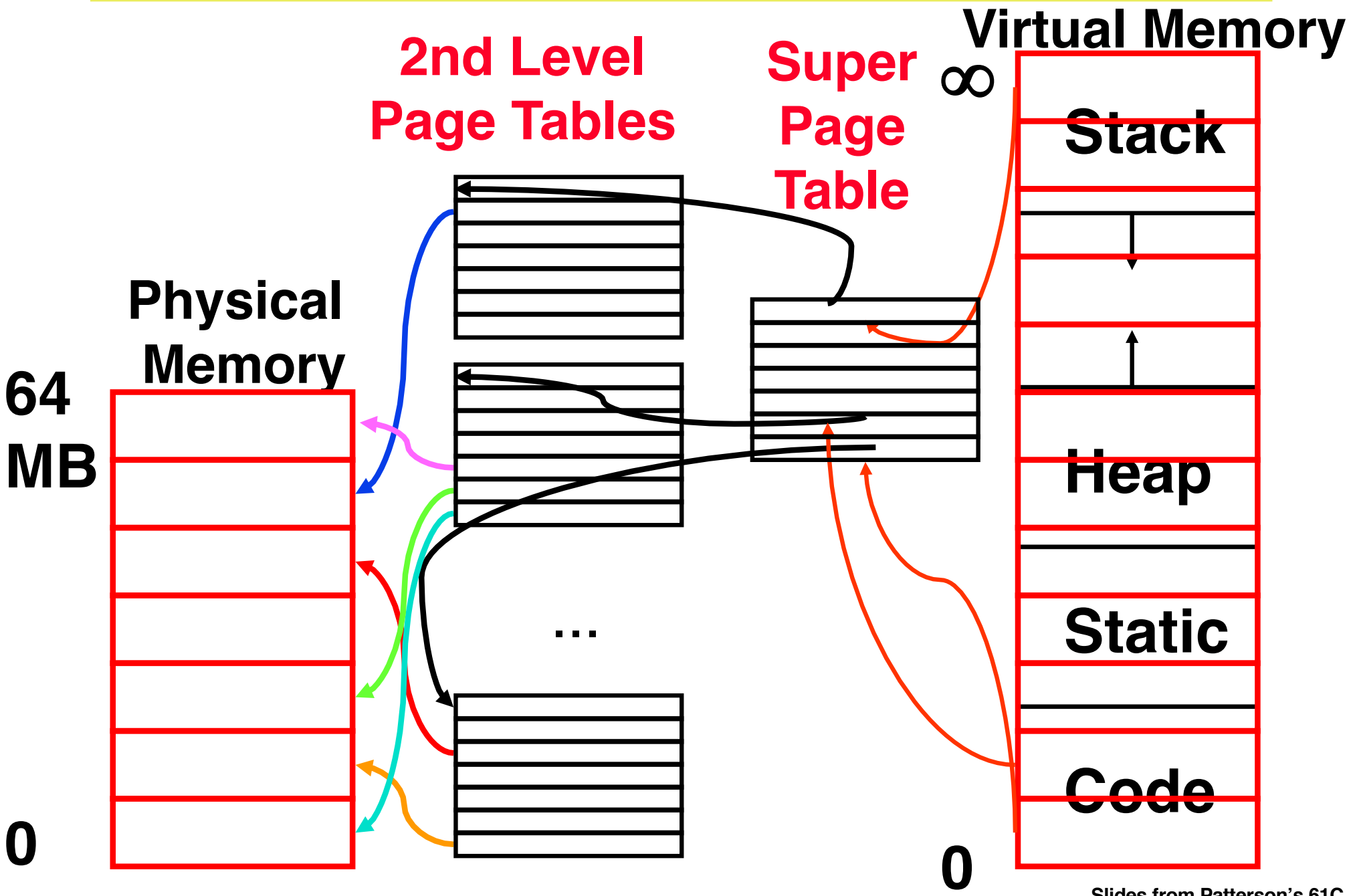
- **Not enough physical memory!**
  - Only, say, 64 MB of physical memory
  - N processes, each 4 GB (2^{32} B) of virtual memory!
  - Could have 1K virtual pages/physical page!
- Spatial Locality to the rescue
  - Each page is 4 KB, lots of nearby references
- No matter how big program is, at any time only accessing a few pages
  - "Working Set": recently used pages

Slides from Patterson's 61C

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

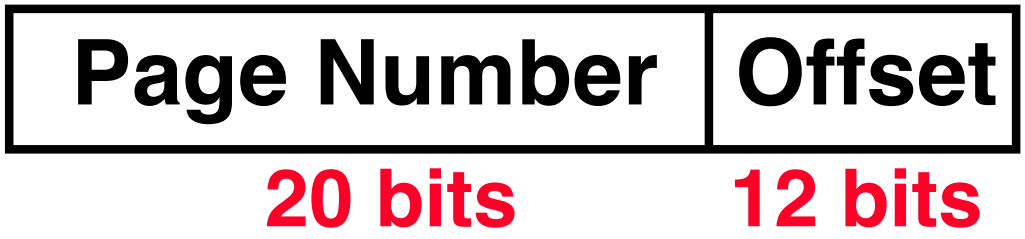
2-level Page Table



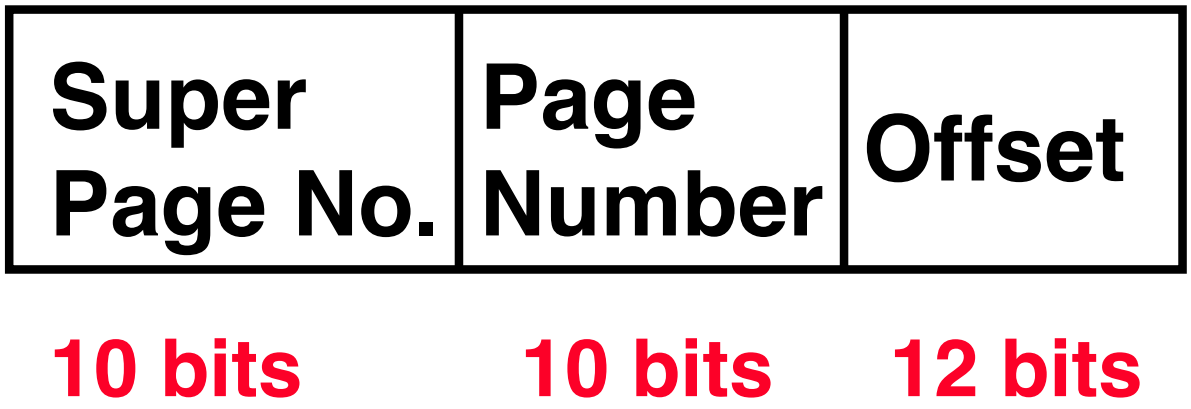
Slides from Patterson's 61C

Page Table Shrink :

- Single Page Table



- Multilevel Page Table



- Only have second level page table for valid entries of super level page table

Slides from Patterson's 61C

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
  - Exercise 7.35 explores exact size

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 more compact representation to reduce memory size cost of simple 1-level page table (especially when 32-bit address $\Rightarrow$ 64-bit address) . Hence, a 2-level page table