Operating Systems
(1DT020 & 1TT802)

Lecture 9
Memory Management :
Demand paging & page replacement

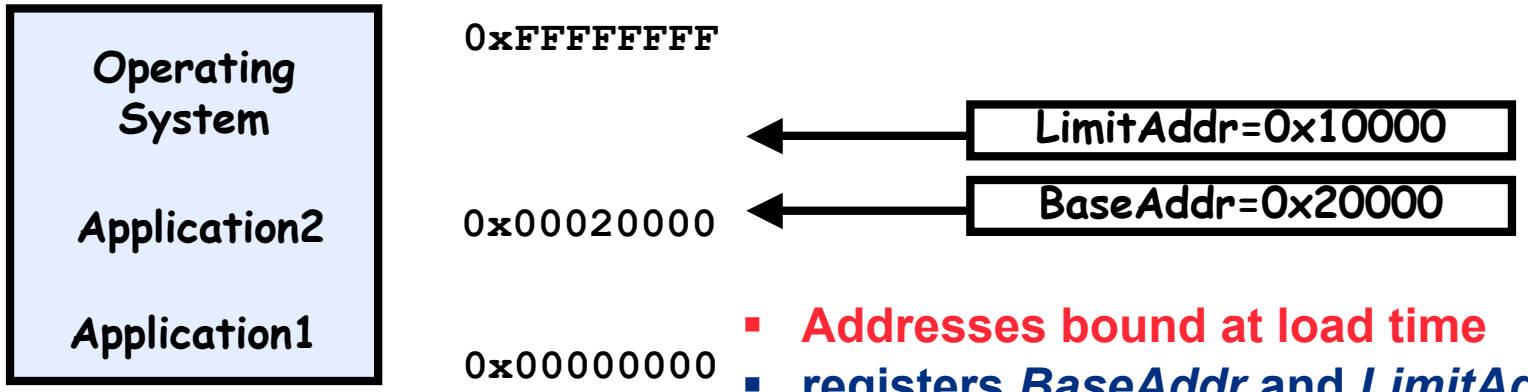
May 05, 2008

Léon Mugwaneza

http://www.it.uu.se/edu/course/homepage/os/vt08

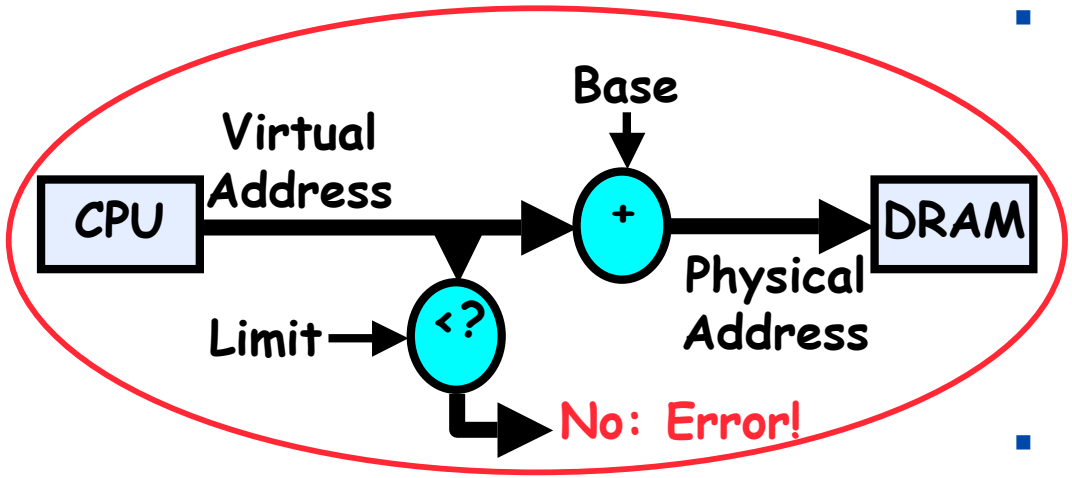
Review: Multiprogramming (with Protection)

- Base and Limit registers



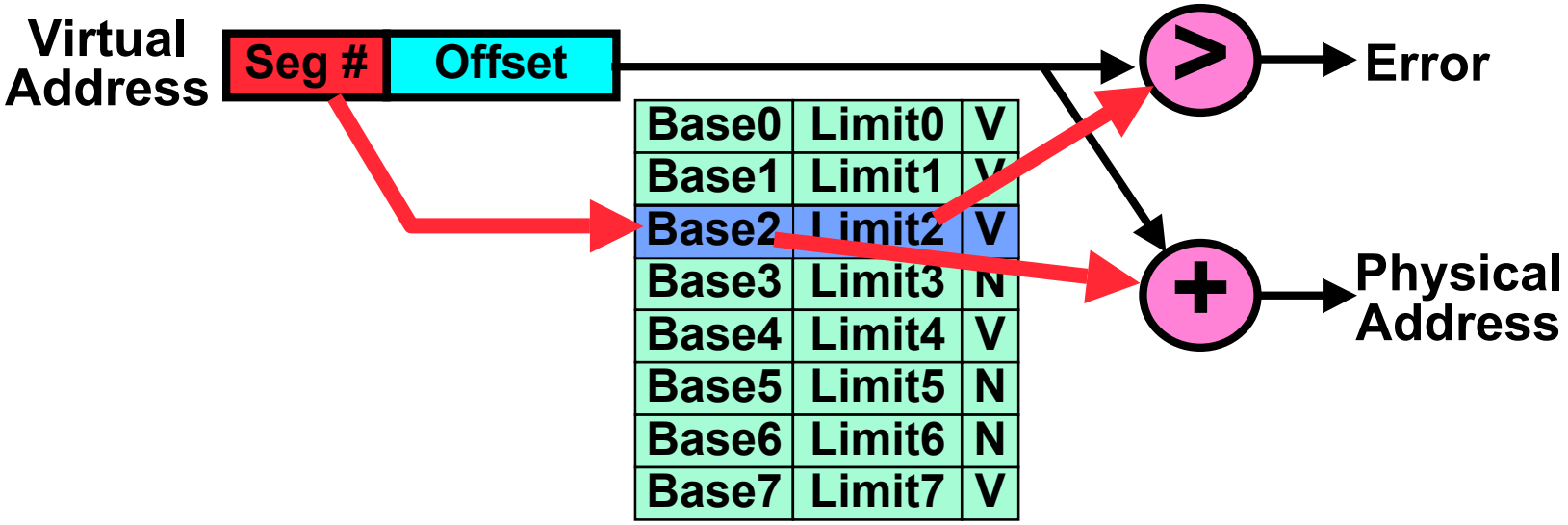
  - Addresses bound at load time
  - registers *BaseAddr* and *LimitAddr* to prevent user from straying outside designated area
  - Only OS can modify Base and limit

- Segmentation : Address translation (virtual memory)



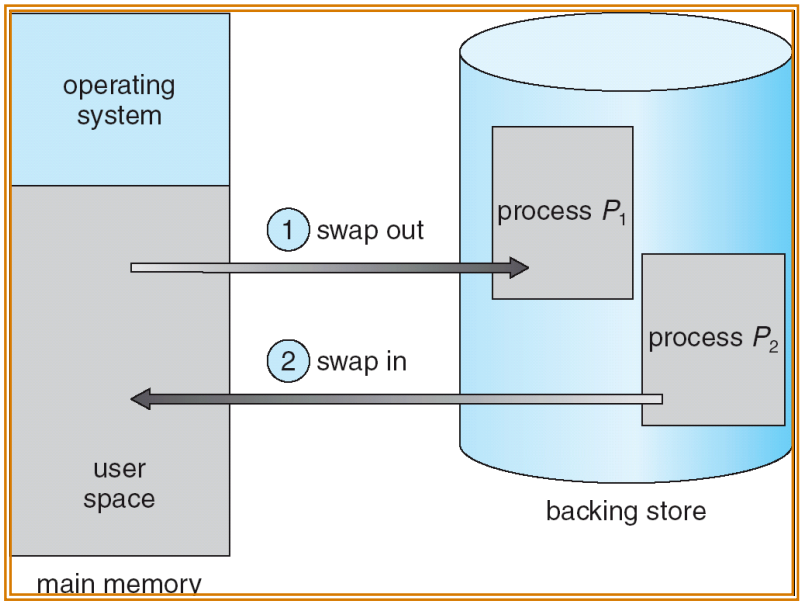
  - Addresses bound at link time
  - Program thinks it is alone in memory
    - Base register added to addresses
    - Accesses outside area checked using Limit register
    - Program can have multiple separate segments
  - Only OS can modify Base and limit

Review: Implementation of Multi-Segment Model



- **Virtual address space has holes**
  - Multiple segments efficient for sparse address spaces
  - If a program addresses gaps, trap to kernel and dump core or extend area
- **Need protection mode in segment table**
  - For example, code segment would be read-only, data and stack would be read-write, etc.
- **What must be saved/restored on context switch?**
  - Segment table stored in CPU, not in memory (small)
  - Might store all of processes memory onto disk when switched (called "swapping")

Review: Schematic View of Swapping



- **Extreme form of Context Switch: Swapping**
  - **In order to make room for next process, some or all of the previous process is moved to disk**
    - **» Likely need to send out complete segments**
  - **This greatly increases the cost of context-switching**
- **Desirable alternative?**
  - **Some way to keep only active portions of a process in memory at any one time**
  - **Need finer granularity control over physical memory**

Goals for Today

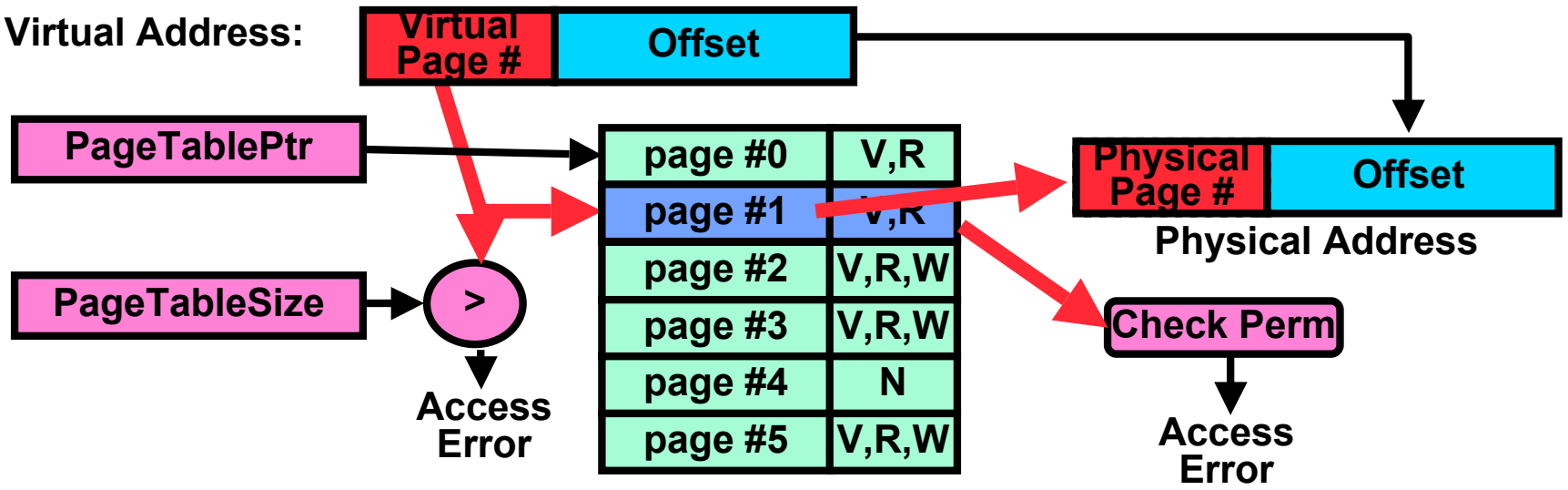
- Paging
- Concept of paging to disk (Demand Paging)
- Page replacement policies

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne, others from Kubiatowicz - CS162 ©UCB Fall 2007 (University of California at Berkeley)

Paging: Physical Memory in Fixed Size Chunks

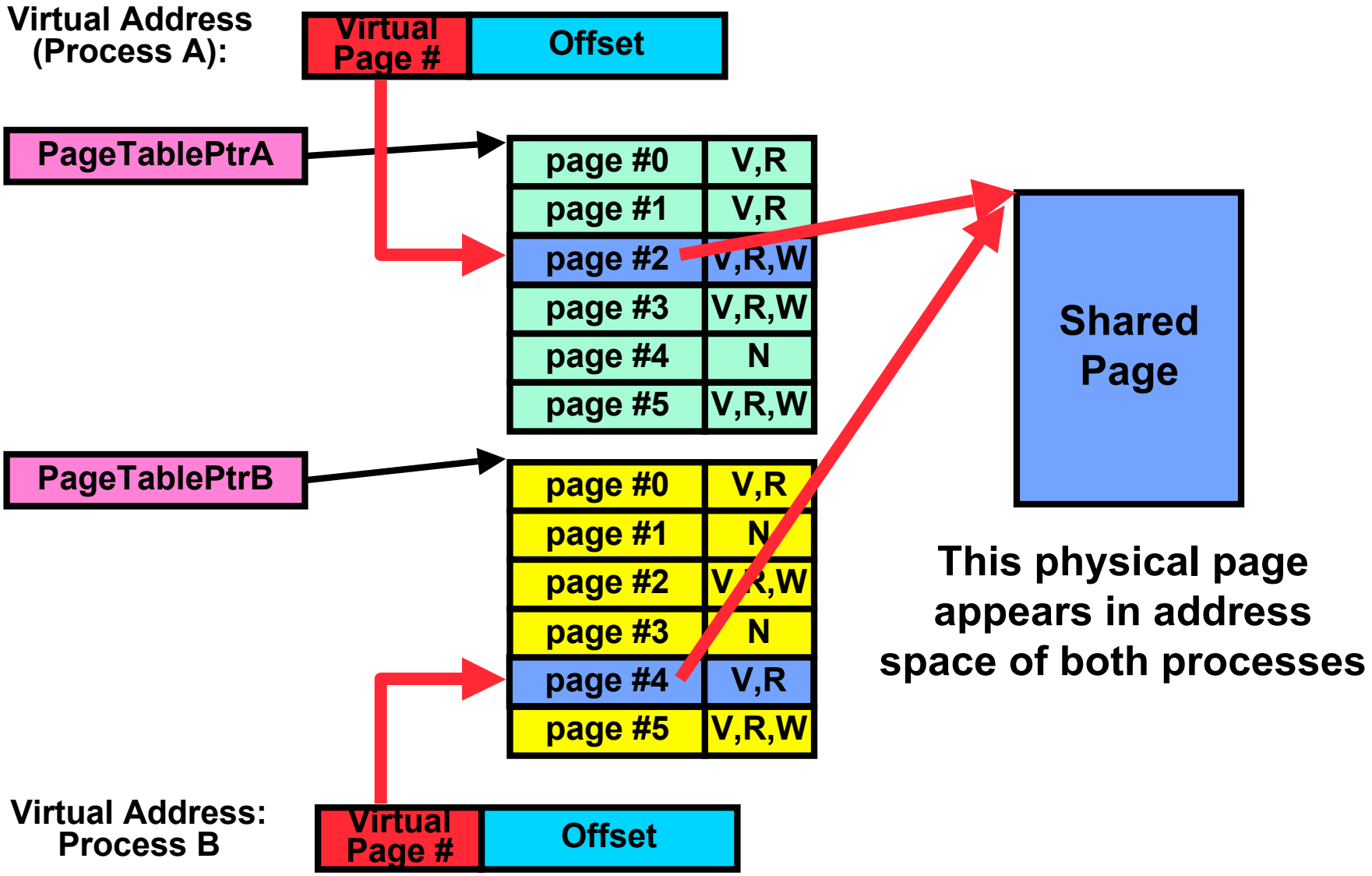
- Problems with segmentation?
  - Must fit variable-sized chunks into physical memory
  - May move processes multiple times to fit everything
  - Limited options for swapping to disk
- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don't need all memory within allocated chunks
- Solution to fragmentation from segments?
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
    - » Can use simple vector of bits to handle allocation:
      0011000111000110 1 … 110010
    - » Each bit represents page of physical memory
      1⇒allocated, 0⇒free
- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    - » Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment

How to Implement Paging?

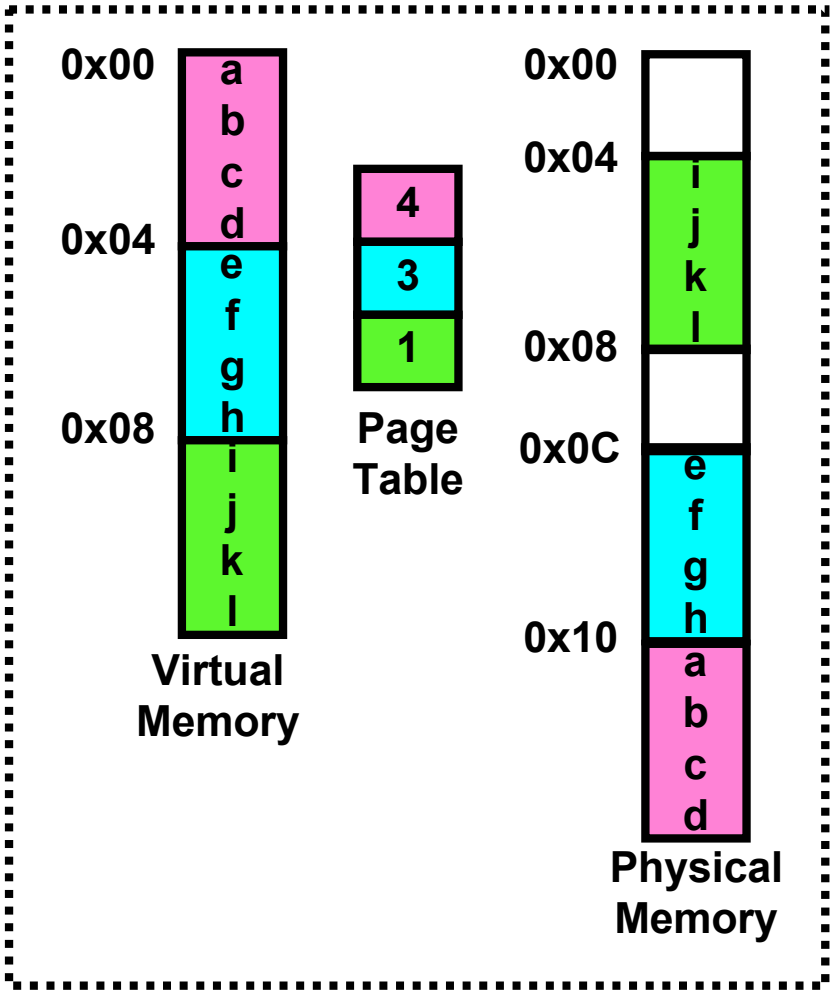


- **Page Table (One per process)**
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - » Permissions include: Valid bits, Read, Write, etc
- **Virtual address mapping**
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

What about Sharing?



Simple Page Table Discussion

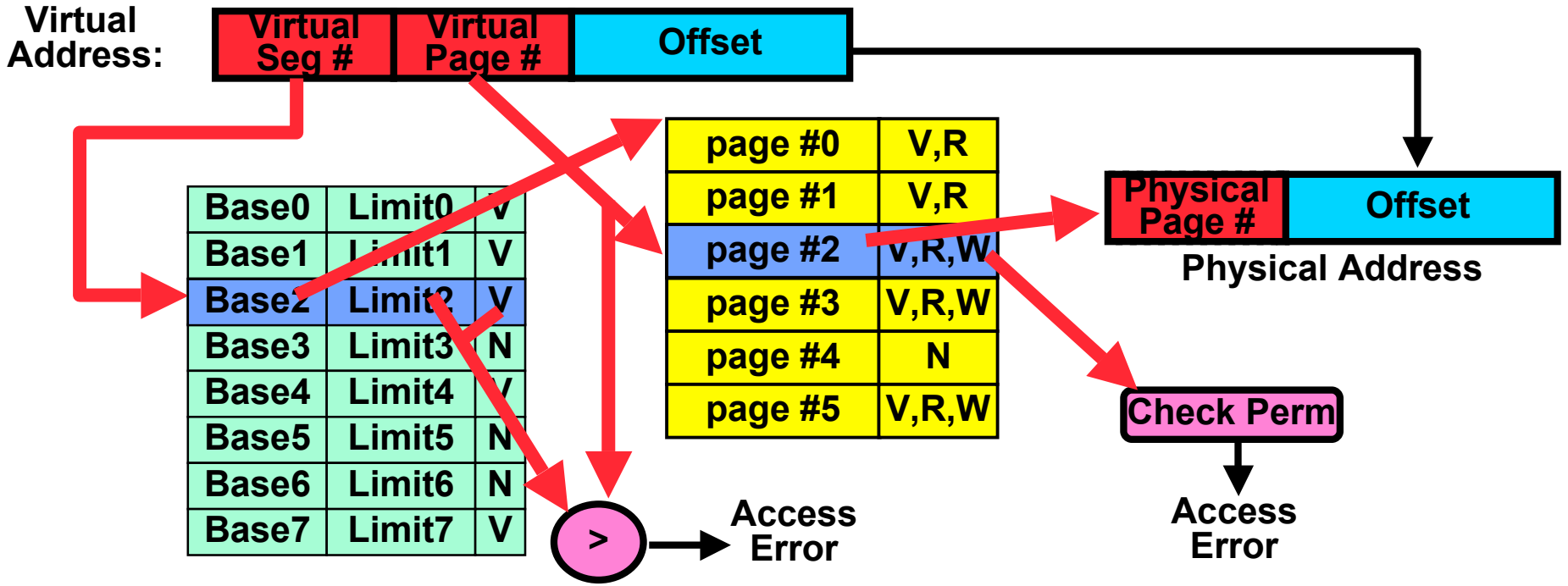


Example (4 byte pages)

- What needs to be saved on a context switch?
  - Page table pointer and limit
- Analysis
  - Pros
    - » Simple memory allocation
    - » Easy to Share
  - Con: What if address space is sparse?
    - » E.g. on UNIX, code starts at 0, stack starts at (2^{31}-1).
    - » With 1K pages, need 4 million page table entries!
  - Con: What if table really big?
    - » Not all pages used all the time ⇒ would be nice to have working set of page table in memory
- How about combining paging and segmentation?

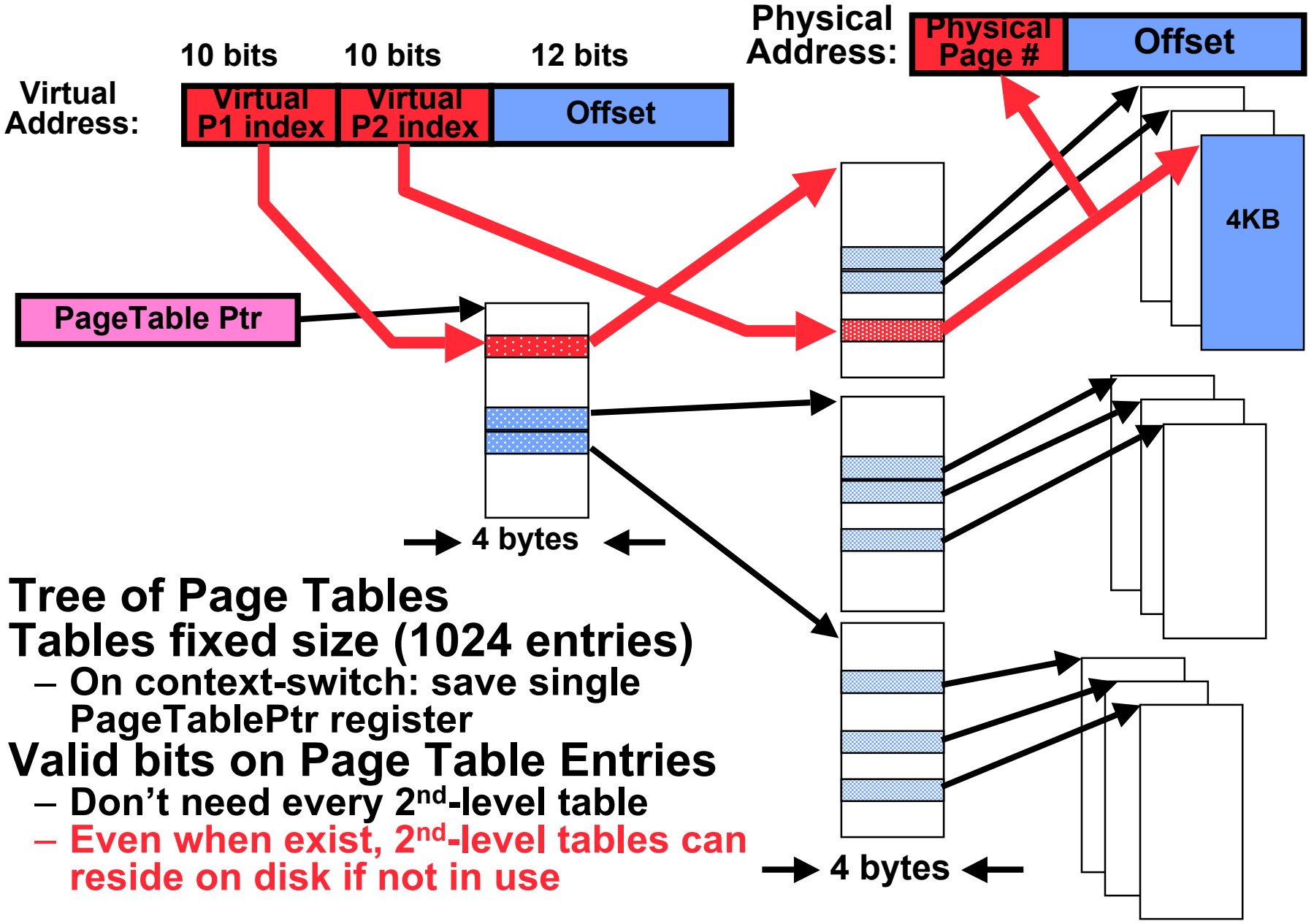
Multi-level Translation

- What about a tree of tables?
  - Lowest level page table⇒memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):



- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

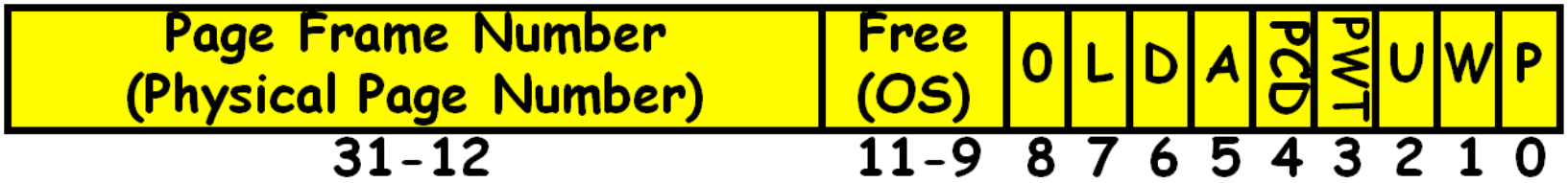
Another common example: two-level page table



- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

What is in a PTE?

- **What is in a Page Table Entry (or PTE)?**
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- **Example: Intel x86 architecture PTE:**
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"



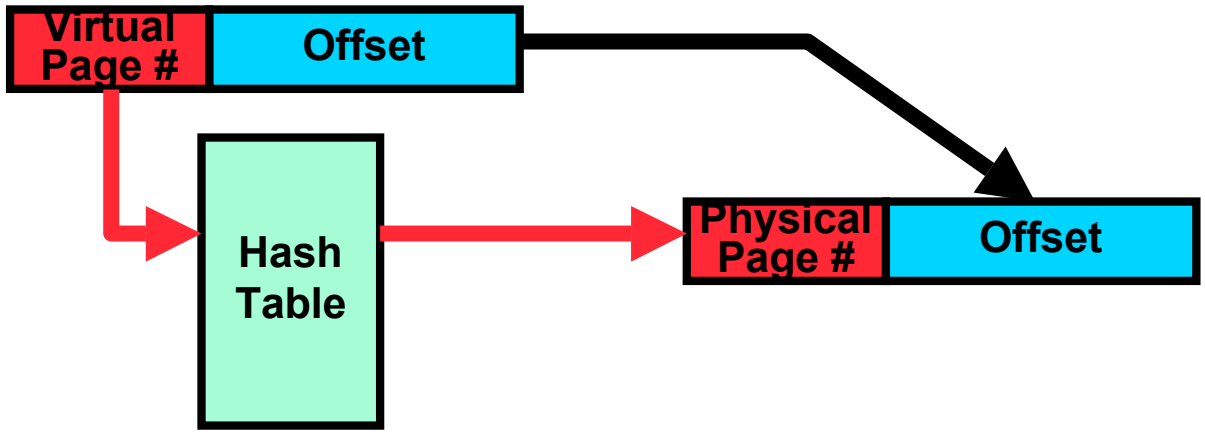
P: Present (same as "valid" bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

Multi-level Translation Analysis

- **Pros:**
  - Only need to allocate as many page table entries as we need for application
    - » In other words, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - » Share at segment or page level (need additional reference counting)
- **Cons:**
  - One pointer per page (typically 4K – 16K pages today)
  - Page tables need to be contiguous
    - » However, previous example keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    - » Seems very expensive!

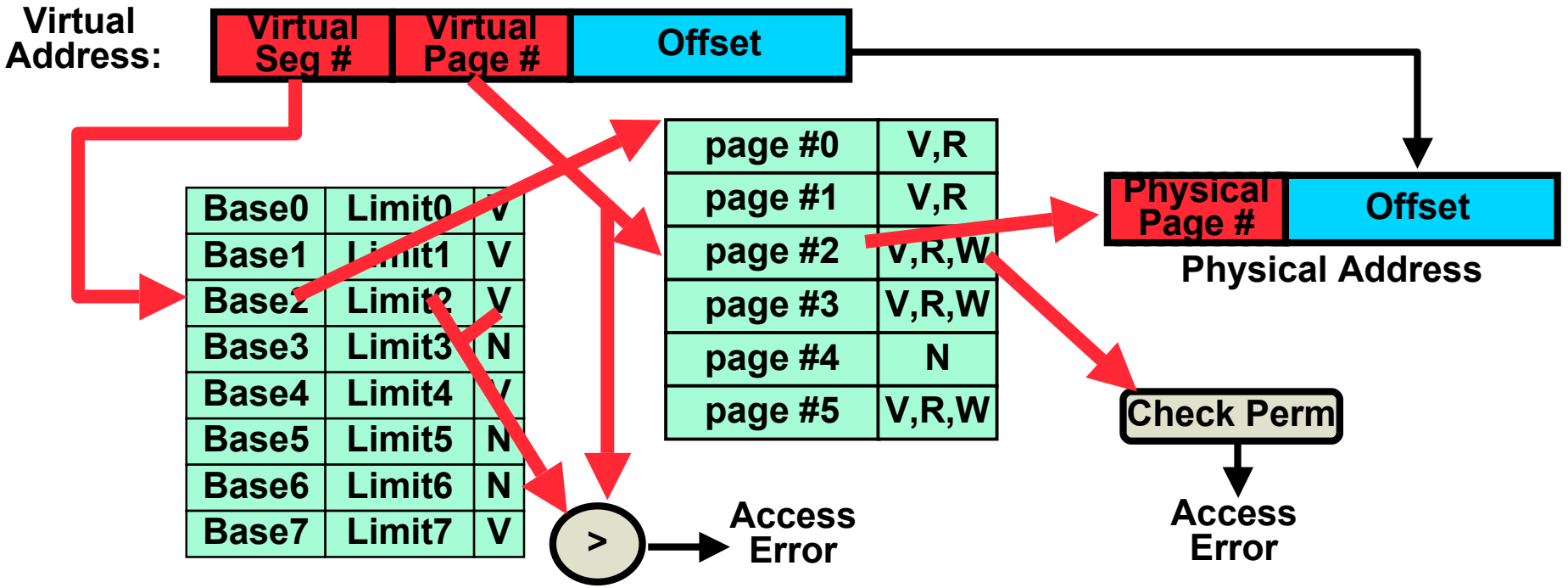
Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    - » Much of process space may be out on disk or not in use



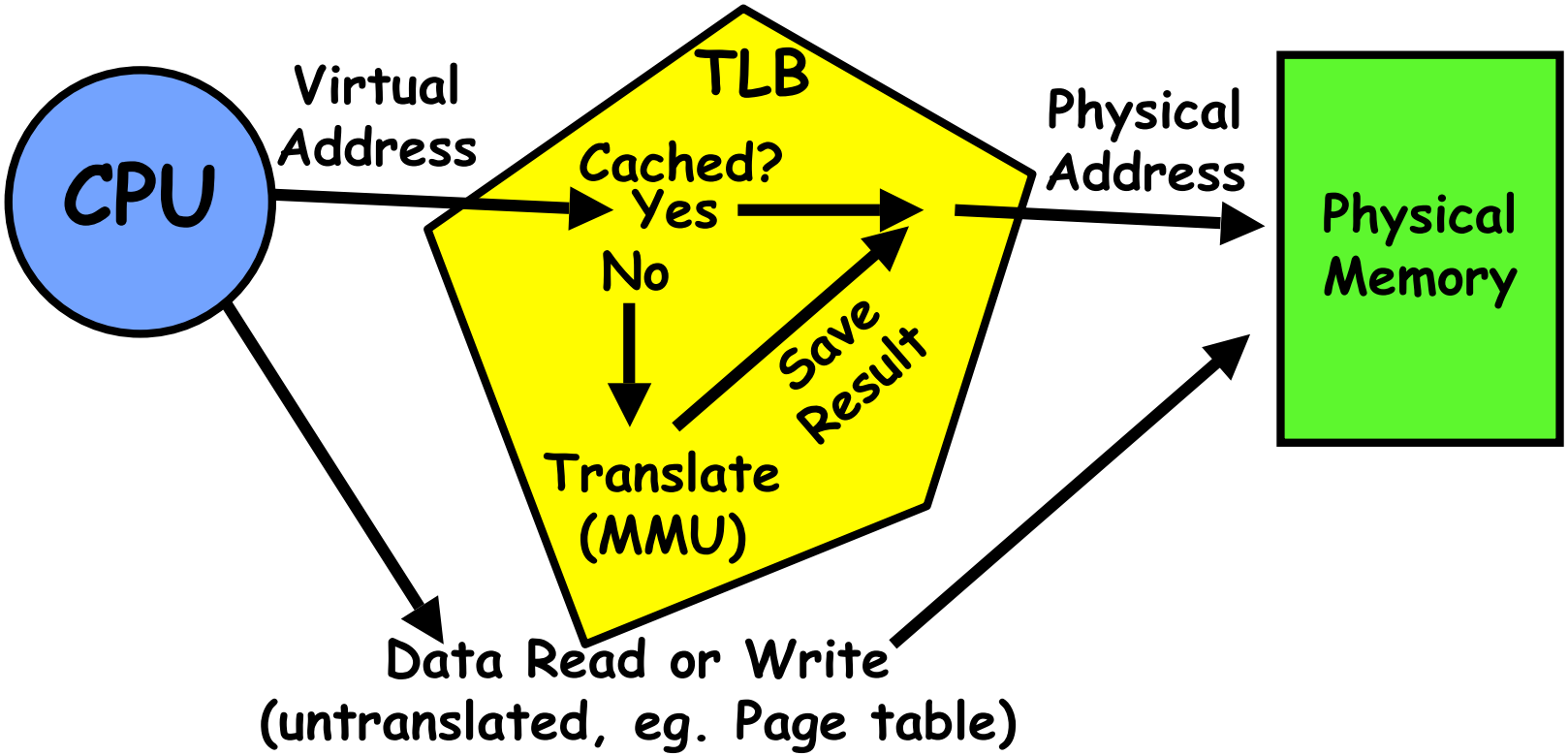
- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
- Cons: Complexity of managing hash changes
  - Often in hardware!

How long does Address translation take ?



- Cannot afford to translate on every access
  - At least 2 DRAM accesses per actual DRAM access
  - or : perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access???
- Solution? Cache translations!
  - Translation Cache: TLB ("Translation Lookaside Buffer")

Caching Applied to Address Translation



- **Question is one of page locality: does it exist?**
  - **Instruction accesses spend a lot of time on the same page (since accesses sequential)**
  - **Stack accesses have definite locality of reference**
  - **Data accesses have less page locality, but still some…**

What Actually Happens on a TLB Miss?

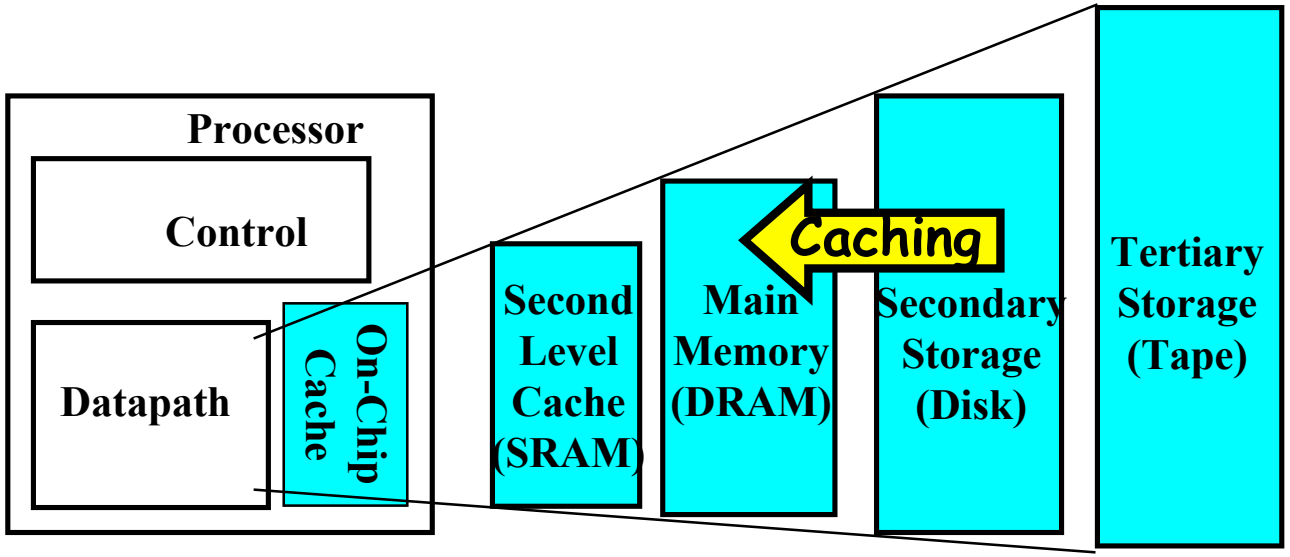
- **Hardware traversed page tables:**
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - » If PTE valid, hardware fills TLB and processor never knows
    - » If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards
- **Software traversed Page tables (like MIPS)**
  - On TLB miss, processor receives TLB fault
  - Kernel traverses page table to find PTE
    - » If PTE valid, fills TLB and returns from fault
    - » If PTE marked as invalid, internally calls Page Fault handler
- **Most chip sets provide hardware traversal**
  - Modern operating systems tend to have more TLB faults since they use translation for many things
  - Examples:
    - » shared segments
    - » user-level portions of an operating system

What happens on a Context Switch?

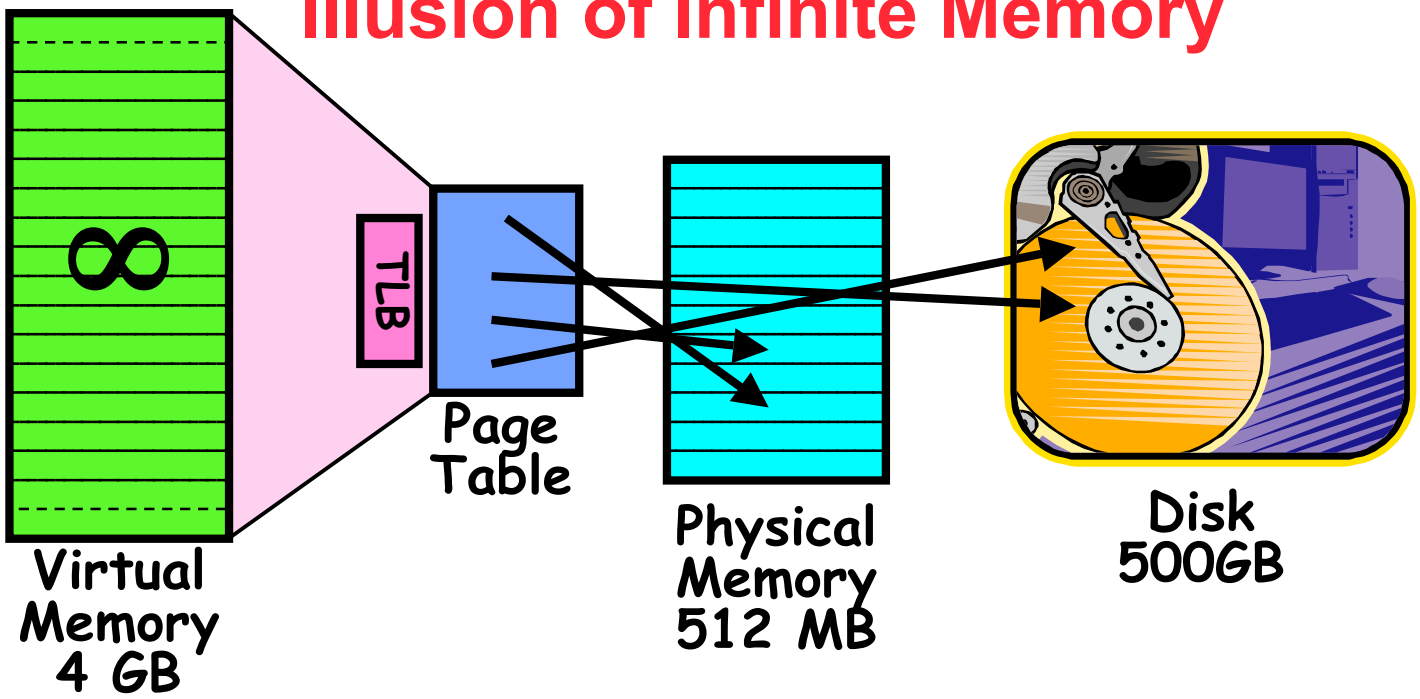
- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate TLB: simple but might be expensive
    - » What if switching frequently between processes?
  - Include ProcessID in TLB
    - » This is an architectural solution: needs hardware
- What if translation tables change?
  - For example, to move page from memory to disk or vice versa…
  - Must invalidate TLB entry!
    - » Otherwise, might think that page is still in memory!
- How big does TLB actually have to be?
  - Usually small: 128-512 entries (remember each entry corresponds to a whole page)

Demand Paging

- **Modern programs require a lot of physical memory**
  - Memory per system growing faster than 25%-30%/year
- **But they don't use all their memory all of the time**
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- **Solution: use main memory as cache for disk**



Illusion of Infinite Memory



- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - » More programs fit into memory, allowing more concurrency
- Principle: Transparent Level of Indirection (page table)
  - Supports flexible placement of physical data
    - » Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    - » Performance issue, not correctness issue

Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap is a "Page Fault"
  - What does OS do on a Page Fault?:
    - » Choose an old page to replace
    - » If old page modified ("D=1"), write contents back to disk
    - » Change its PTE and any cached TLB to be invalid
    - » Load new page into memory from disk
    - » Update page table entry, invalidate TLB for new entry
    - » Continue thread from original faulting location
  - TLB for new page will be loaded when thread continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    - » Suspended process sits on wait queue
- What if an instruction has side-effects?
  - Unwind side-effects (easy to restart) or Finish off side-effects (messy!)
  - Example 1: `mov (sp)+,10`.
    - » What if page fault occurs when write to stack pointer?
    - » Did `sp` get incremented before or after the page fault?

Demand Paging Example

- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
  - EAT = Hit Rate x Hit Time + Miss Rate x Miss Time
- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose p = Probability of miss, 1-p = Probably of hit
  - Then, we can compute EAT as follows:

$$\begin{aligned} EAT &= (1 - p) \times 200ns + p \times 8\ ms \\ &= (1 - p) \times 200ns + p \times 8{,}000{,}000ns \\ &= 200ns + p \times 7{,}999{,}800ns \end{aligned}$$

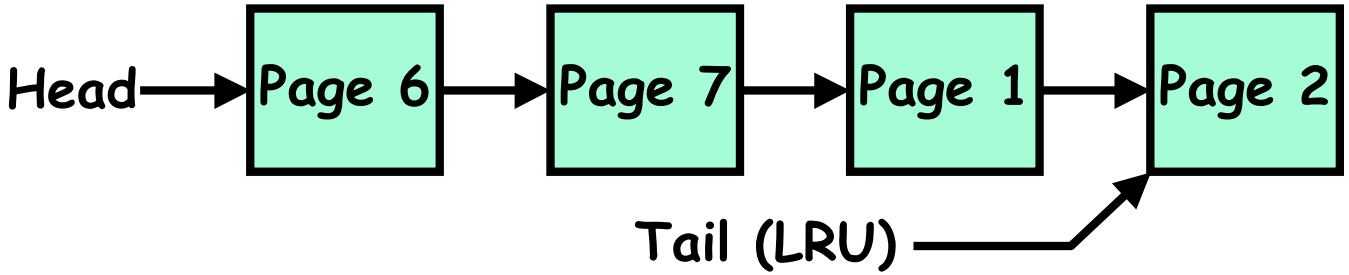
- If one access out of 1,000 causes a page fault, then EAT = 8.2 µs:
  - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
  - $200ns \times 1.1 < EAT \Rightarrow p < 2.5 \times 10^{-6}$
  - This is about 1 page fault in 400000!

Page Replacement Policies

- **Why do we care about Replacement Policy?**
  - Replacement is an issue with any cache
  - Particularly important with pages
    - » The cost of being wrong is high: must go to disk
    - » Must keep important pages in memory, not toss them out
- **FIFO (First In, First Out)**
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- **MIN (Minimum):**
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- **RANDOM:**
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Pretty unpredictable – makes it hard to make real-time guarantees

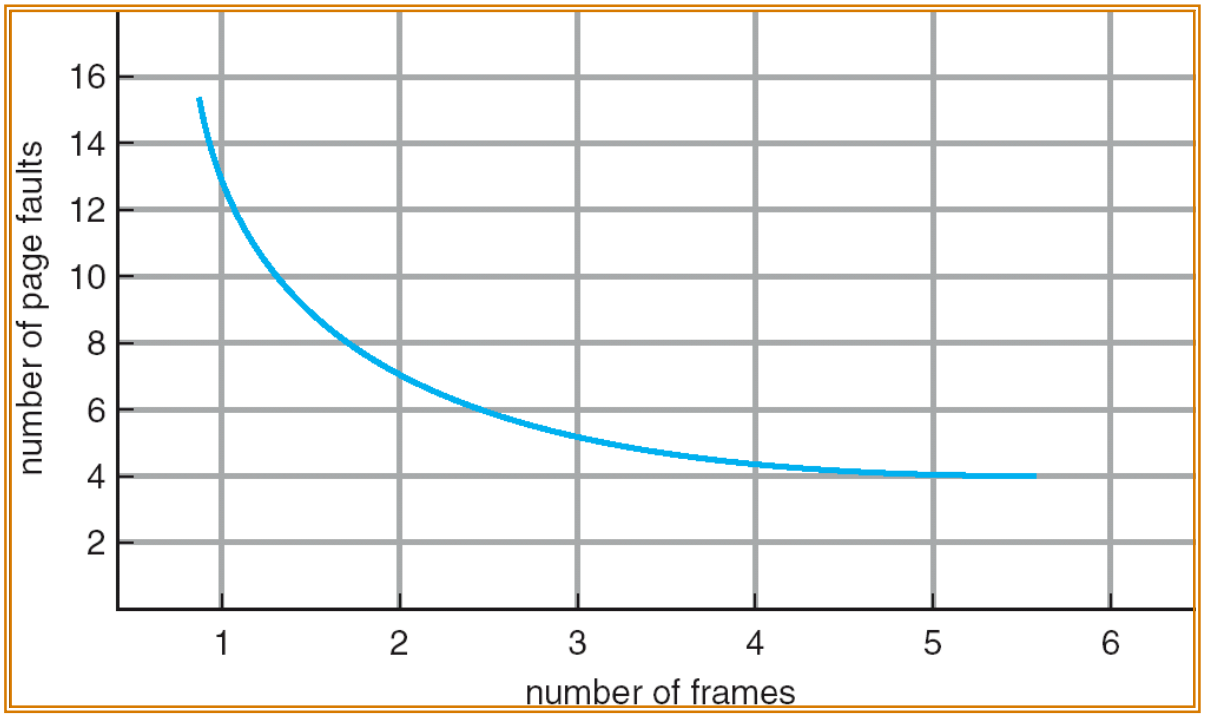
Replacement Policies (Con't)

- **LRU (Least Recently Used):**
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- **How to implement LRU? Use a list!**



  - On each use, remove page from list and place at head
  - LRU page is at tail
- **Problems with this scheme for paging?**
  - Need to know immediately when each page used so that can change position in list…
  - Many instructions for each hardware access
- **In practice, people approximate LRU (more later)**

Graph of Page Faults Versus The Number of Frames



- **One desirable property: When you add memory the miss rate goes down**
  - **Does this always happen?**
  - **Seems like it should, right?**
- **No: BeLady's anomaly**
  - **Certain replacement algorithms (FIFO) don't have this obvious property!**

Adding Memory Doesn't Always Help Fault Rate

- **Does adding memory reduce number of page faults?**
  - **Yes for LRU and MIN**
  - **Not necessarily for FIFO! (Called Belady's anomaly)**

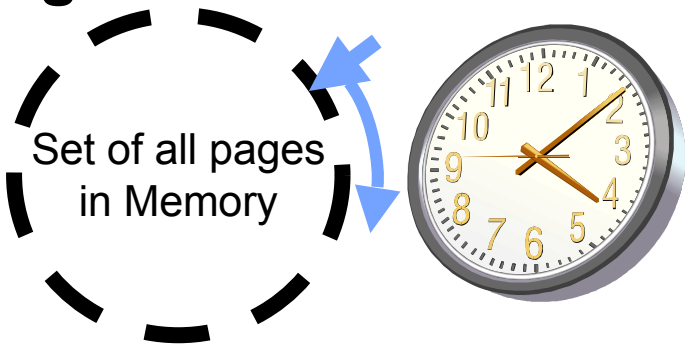
Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

- **After adding memory:**
  - **With FIFO, number of fault increased (10 for 4 frames vs 9 for 3 frames)**
  - **In contrast, with LRU or MIN, set of pages in memory with X frames is a subset of set of pages in memory with X+1 frames**

Implementing LRU

- **Perfect:**
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality for many reasons
- **Clock Algorithm: Arrange physical pages in circle with single clock hand**
  - Approximate LRU (approx to approx to MIN)
  - Replace an old page, not the oldest page

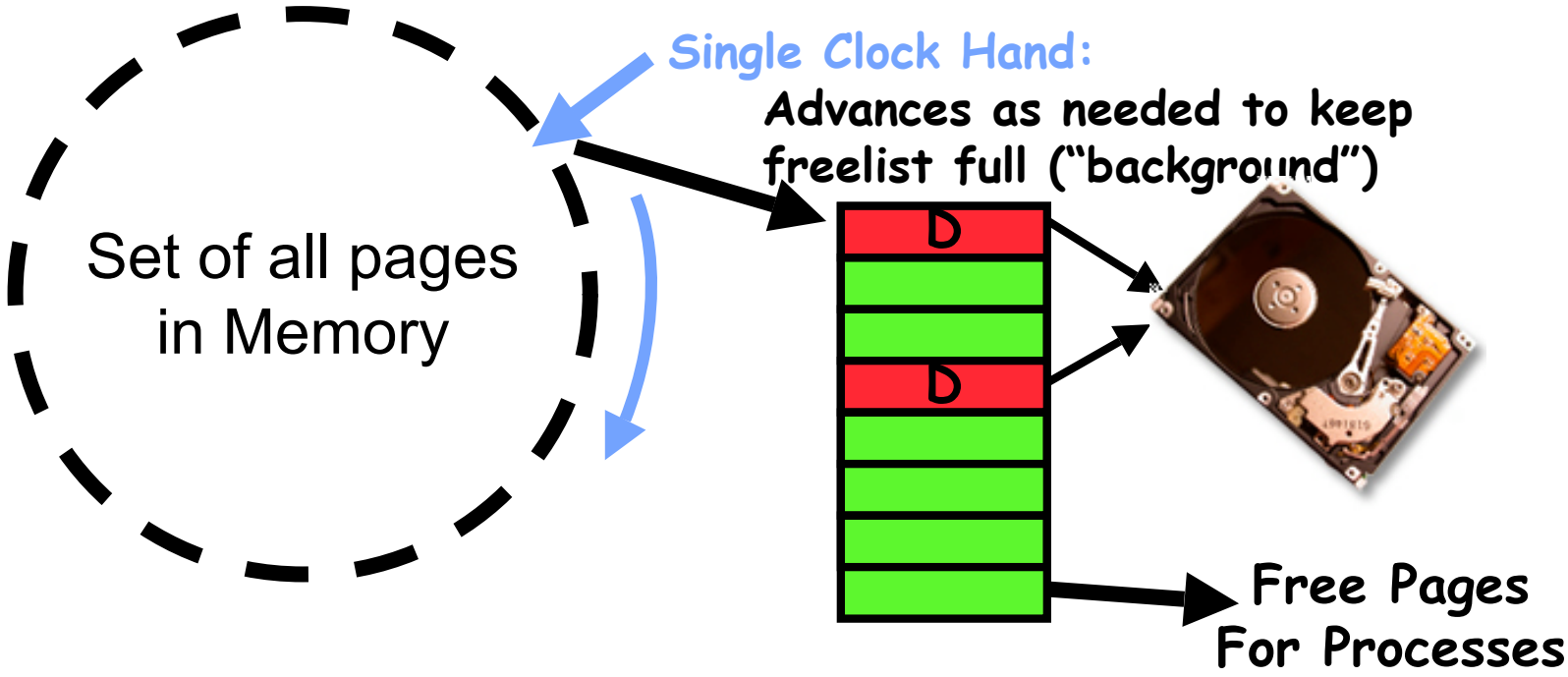


- **Details:**
  - Hardware "use" bit per physical page:
    - » Hardware sets use bit on each reference
    - » If use bit isn't set, means not referenced in a long time
    - » hardware sets use bit in the TLB; use bit copied back to page table when TLB entry gets replaced
  - On page fault:
    - » Advance clock hand (not real time)
    - » Check use bit: 1→used recently; clear and leave alone
      0→selected candidate for replacement
  - Will always find a page or loop forever?
    - » Even if all use bits set, will eventually loop around⇒FIFO
- **One way to view clock algorithm:**
  - Crude partitioning of pages into two groups: young and old
  - Why not partition into more than 2 groups?

N^{th} Chance version of Clock Algorithm

- **N^{th} chance algorithm: Give page N chances**
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - » 1⇒clear use and also clear counter (used in last sweep)
    - » 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- **How do we pick N?**
  - Why pick large N? Better approx to LRU
    - » If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - » Otherwise might have to look a long way to find free page
- **What about dirty pages?**
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - » Clean pages, use N=1
    - » Dirty pages, use N=2 (and write back to disk when N=1)

Free List



- **Keep set of free pages ready for use in demand paging**
  - Free list filled in background by Clock algorithm or other technique ("Pageout demon")
  - Dirty pages start copying back to disk when enter list
  - If page needed before reused, just return to active set
- **Advantage: Faster for page fault**
  - Can always use page (or pages) immediately on fault

Summary

- Paging : Memory divided into fixed-sized chunks (pages) of memory
  - Virtual page number from virtual address mapped through page table to physical page number. Offset of virtual address same as physical address
  - Changing of page tables only available to kernel
  - Every Access translated through page table
    - » Translation speeded up using a TLB (cache for recent translations)
  - Multi-Level Tables Permit sparse population of address space
- Demand paging: main memory used as cache for disk
- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past
- Clock Algorithm: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not "in use"
  - If page not "in use" for one pass, than can replace
- N^{th}-chance clock algorithm: Another approx LRU
  - Give pages multiple passes of clock hand before replacing
- List of free page frames makes page fault handling faster
  - Filled in back ground by pageout demon