

University of Southern California

Viterbi School of Engineering

EE352

Computer Organization and Architecture

Virtual Memory

References:

- 1) Textbook
- 2) Mark Redekopp's slide series

Shahin Nazarian

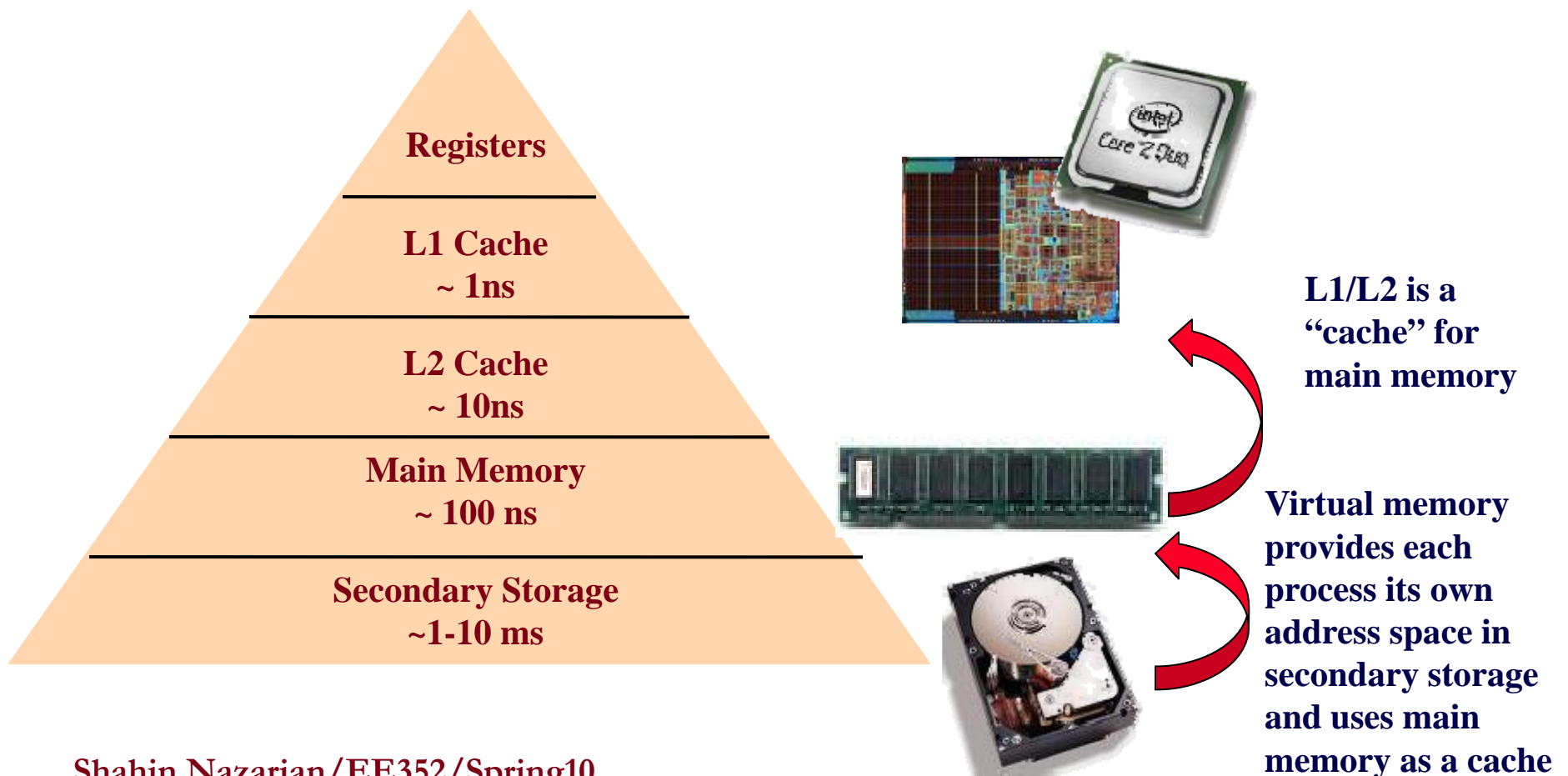
Spring 2010

Virtual Memory Concept

- A mechanism for hiding the details of how much physical memory exists and how it's being shared
- Provides the illusion of
 - Each program (process) having its own private control of the entire memory (address space)
 - A larger (or smaller) amount of memory than is physically present
- Use MM as a cache for multiple programs and their data as they run using secondary storage (hard-drive) as the home location

Memory Hierarchy & Caching

- Lower levels act as a cache for upper levels



Virtual Memory Motivation

- Allow a simple mechanism for multiple programs to run simultaneously and provide protection
- Make main memory look larger than it really is (or smaller if desired)
 - The smaller can occur when the processor address size is small relative to the state of the memory technology. No Single program can benefit, but a collection of programs running at the same time can benefit from not having to be swapped to memory

Virtual Memory Idea

- Suppose we want to build a caller-ID mechanism for your contacts on your cell phone
 - Let us assume 1000 contacts represented by a 3-digit integer (0-999) by the cell phone (this ID can be used to look up their names)
 - We want to use a simple **Look-Up Table (LUT)** to translate phone numbers to contact IDs, how shall we organize/index our LUT?

① LUT indexed w/ contact ID

000	213-745-9823
001	626-454-9985
002	818-329-1980
...	...
999	323-823-7104

Doesn't Work
 We are given phone # and need to translate to ID
 (1000 accesses)

② Sorted LUT indexed w/ used phone #'s

213-730-2198	436
213-745-9823	000
323-823-7104	999
...	...
818-329-1980	002

Could Work
 Since its in sorted order we could use a binary search
 ($\log_2 1000$ accesses)

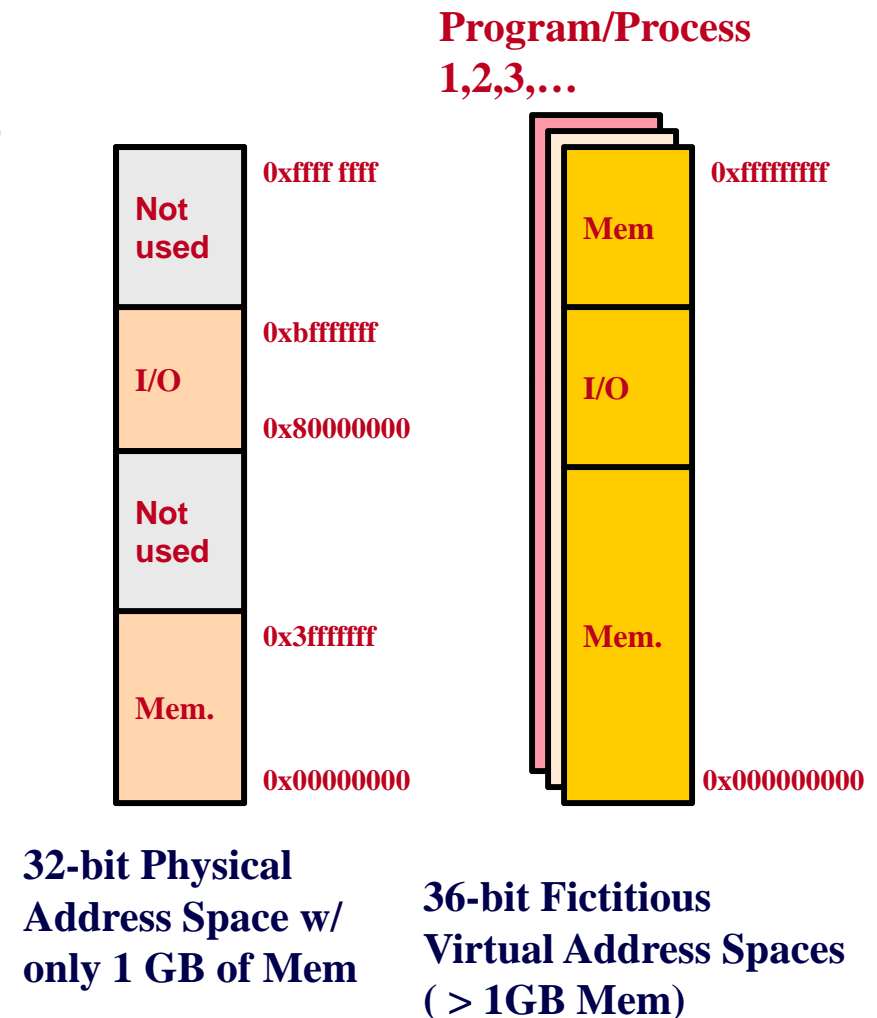
③ LUT indexed w/ all possible phone #'s

000-000-0000	null
...	..
213-745-9823	000
...	...
999-999-9999	null

Could Work
 Easy to index & find but **LARGE***
 (1 access)

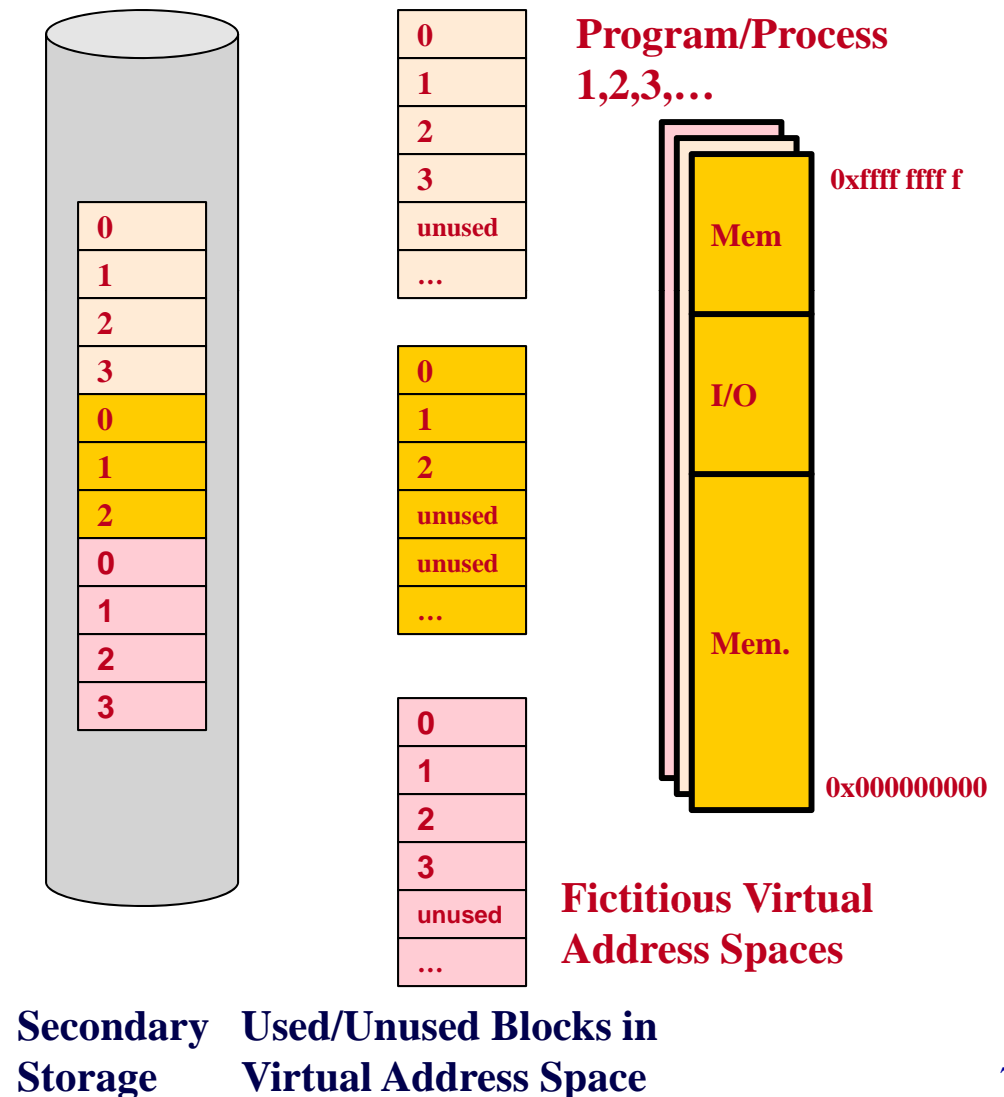
Address Spaces

- Physical address space corresponds to the actual system address bus width and range (i.e. main memory and I/O)
- Each process/program runs in its own private “virtual” address space
 - Virtual address space can be larger (or smaller) than physical memory
 - Virtual address spaces are protected from each other



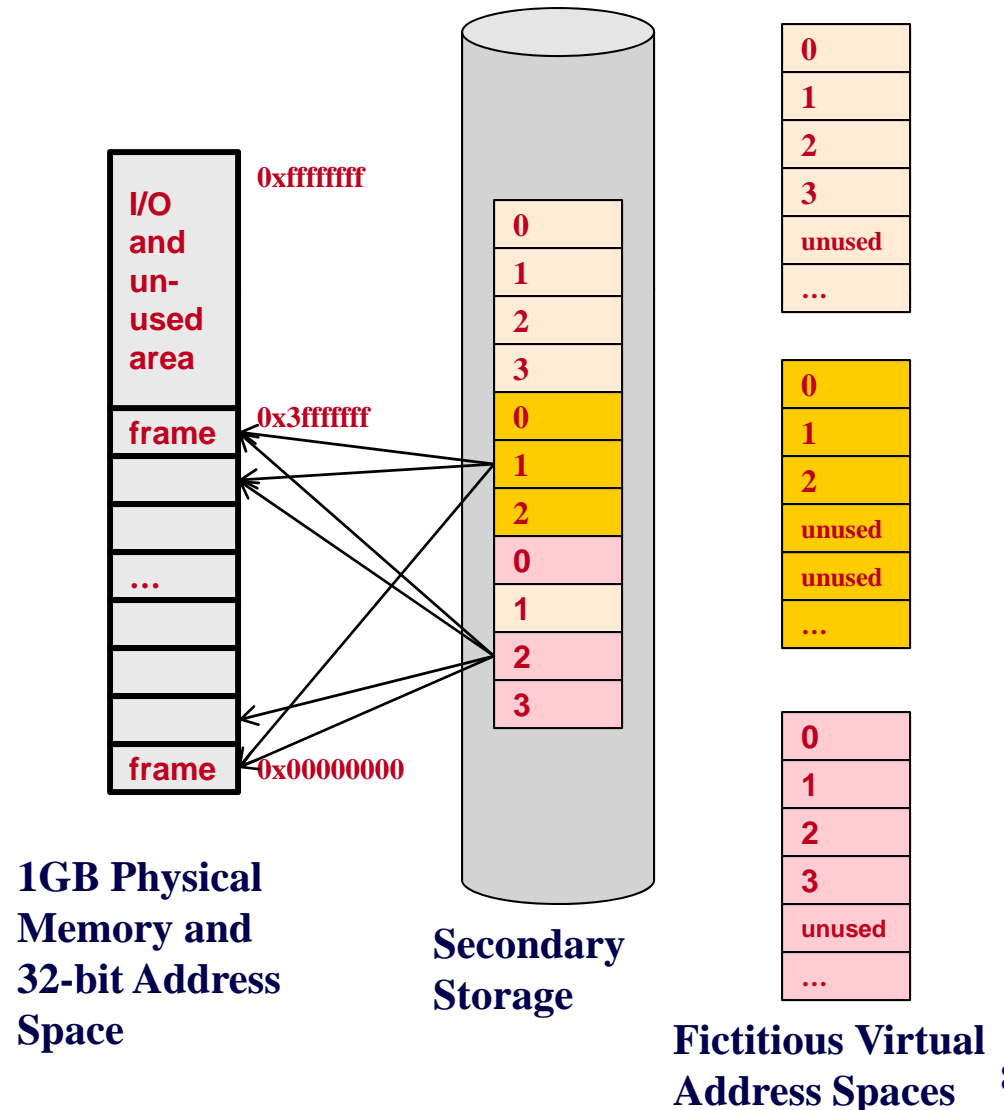
Virtual Address Spaces

- Virtual address spaces are broken into blocks called “pages”
- Depending on the program, much of the virtual address space will not be used
- All used pages are “housed” in secondary storage (hard drive)



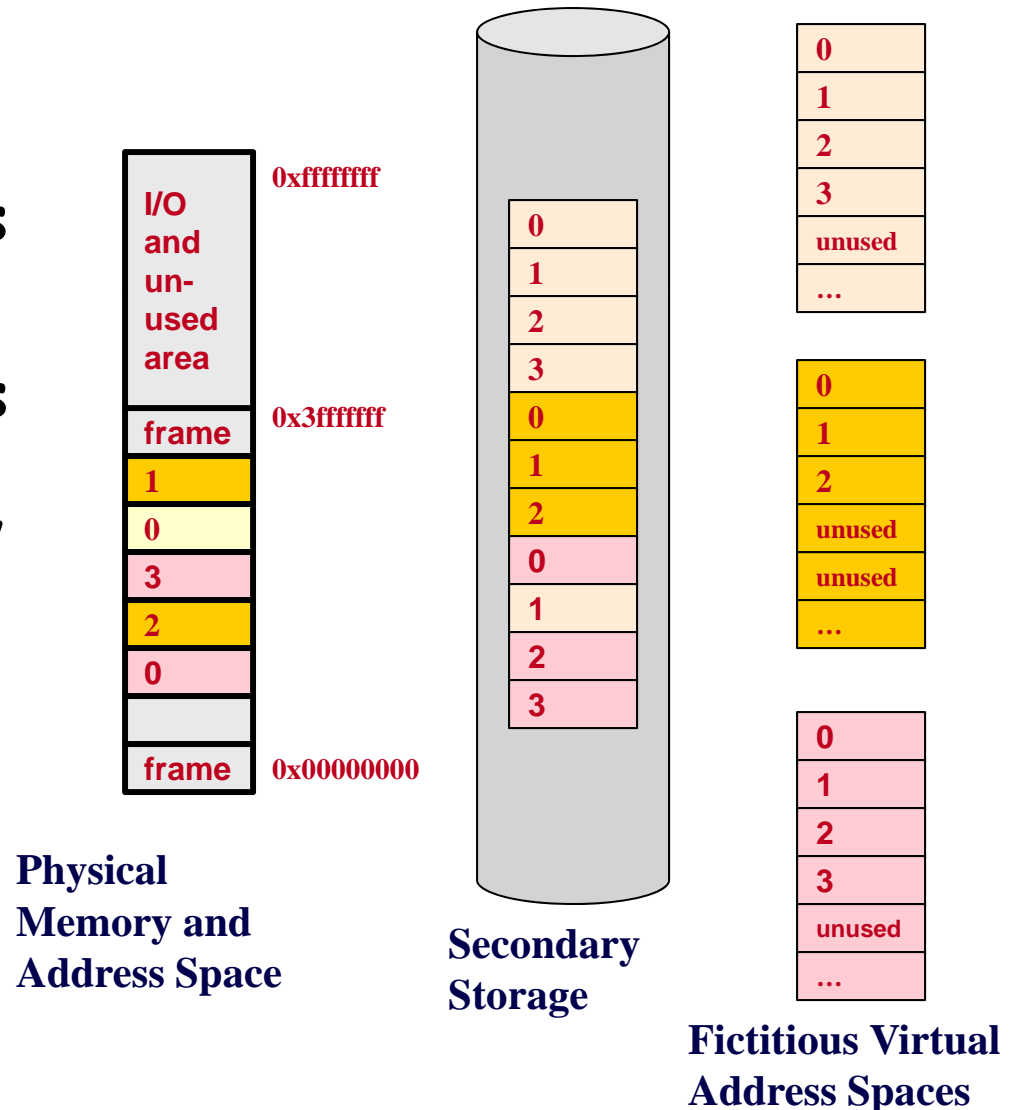
Physical Address Space

- Physical memory is broken into page-size blocks called "page frames"
- Multiple programs are run via time-slicing and their pages (code & data) need to reside in physical memory
- Physical memory acts as a cache for pages from the secondary storage as each program executes



Physical Memory Usage

- Hardware will translate the virtual addresses used by the program to the physical address where that page resides
- If an attempt is made to access a page that is not in physical memory, a “page fault exception” is declared and the OS brings in the page to physical memory (possibly evicting another page)



Page Fault

- Many design choices in virtual memory systems are motivated by the high cost of a **page fault** (miss in virtual memory)
- A page fault takes millions of clock cycles to process (MM latency is 100,000 times lower than disk,) so
 - Pages should be large enough (could be from 1KB to 64KB)
 - Page faults can be handled in sw, cause overhead will be small compared to disk access time
 - Write-through won't work for virtual memory, since writes take too long. Instead write-back should be used

Memory Technology	Typical Access Times	\$ per GB in 2008
SRAM	0.5-2.5ns	\$2000-\$5000
DRAM	50-70ns	\$20-\$75
Magnetic Disk	5,000,000-20,000,000ns	\$0.2-\$2

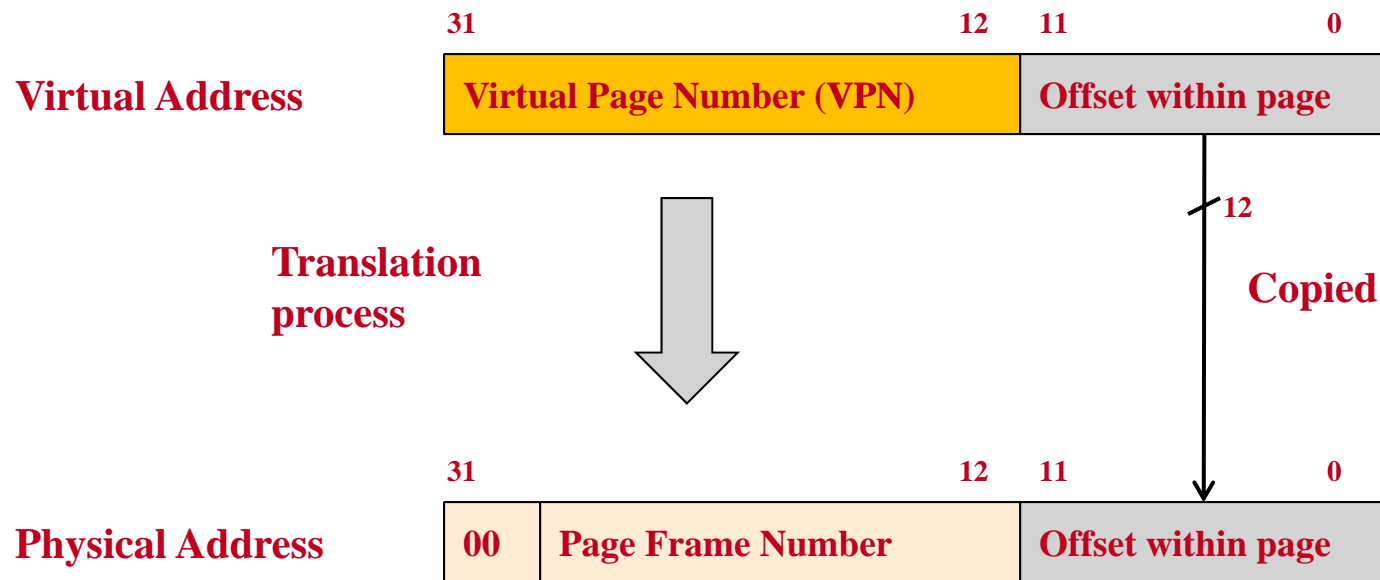
Source: H&P, “CO&D”, 4th ed.
Shahin Nazarian/EE352/Spring10

Virtual Address Translation

- Compiler (i.e. assembly) uses virtual addresses
- HW with the help of OS, takes the virtual address and produces either:
 - A translation of the virtual address to the corresponding MM physical address if the page is in MM, or:
 - Determines the page is not in MM and generates a page fault exception to allow the OS to bring in the desired data from the backing store
- Done on a per-process basis (i.e. each process needs its own set of translations) because a virtual address is not unique (each process may have data at its own virtual address 0x0)

Page Size and Address Translation

- Usually pages are several KB in size
- Example: 32-bit virtual & physical address, 1GB physical memory, 4KB pages
- Virtual page number to physical page frame translation performed by HW unit = **MMU (Memory Management Unit)**



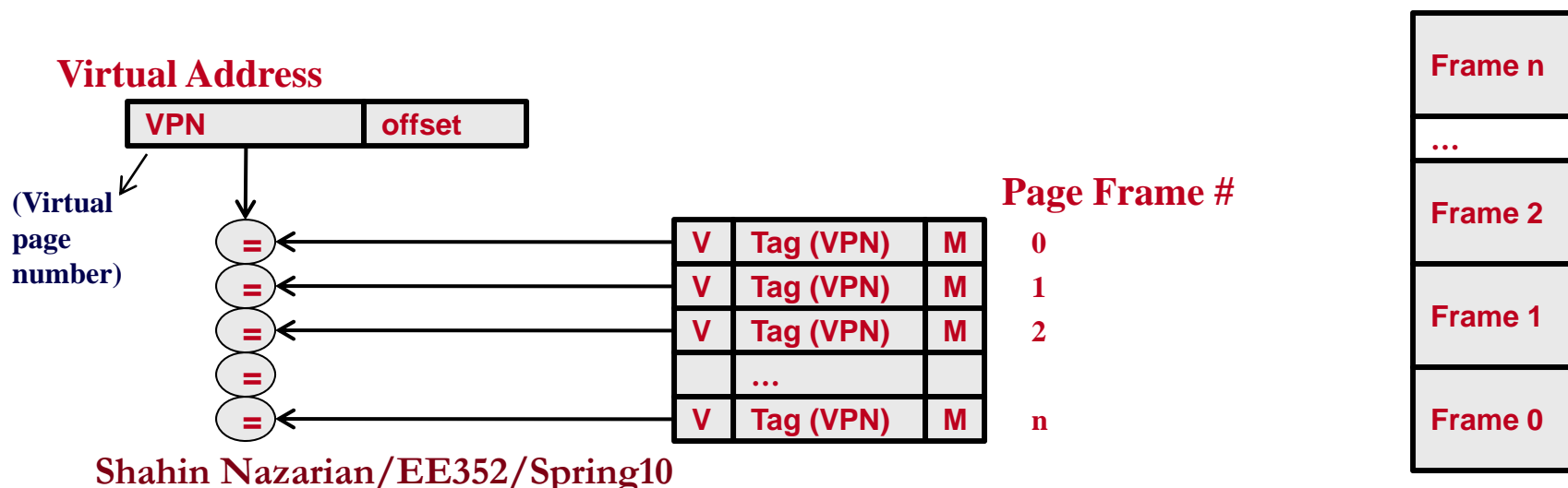
VM Design Implications - A Review

SLOW secondary storage access on page faults, e.g. 10ms

- Implies page size should be fairly large (i.e. once we've taken the time to find data on disk, make it worthwhile by accessing a reasonably large amount of data)
- Implies the placement of pages in main memory should be fully associative to reduce conflicts and maximize page hit rates
- Implies a "page fault" is going to take so much time to even access the data that we can handle them in software (via an exception) rather than using HW like typical cache misses
- Implies eviction algorithms like LRU can be used since reducing page miss rates will pay off greatly
- Implies write-back (write-through would be too expensive)

Address Translation Issues

- To implement full associativity we could try to store tags (**VPNs**) associated with each page frame
- This is impractical since if we have 1GB of physical memory and 4KB pages, it would require a search of 256K tags (too expensive even in SW)
- Instead, most systems implement full associativity using a look-up table = **PAGE TABLE**



Analogy for Page Tables

- Suppose we want to build a caller-ID mechanism for your contacts on your cell phone
 - Let us assume 1000 contacts represented by a 3-digit integer (0-999) by the cell phone (this ID can be used to look up their names)
 - We want to use a simple **Look-Up Table (LUT)** to translate phone numbers to contact IDs, how shall we organize/index our LUT?

① LUT indexed w/ contact ID

000	213-745-9823
001	626-454-9985
002	818-329-1980
...	...
999	323-823-7104

Doesn't Work
 We are given phone # and need to translate to ID
 (1000 accesses)

Shahin Nazarian/EE352/Spring10

② Sorted LUT indexed w/ used phone #'s

213-730-2198	436
213-745-9823	000
323-823-7104	999
...	...
818-329-1980	002

Could Work
 Since its in sorted order we could use a binary search
 ($\log_2 1000$ accesses)

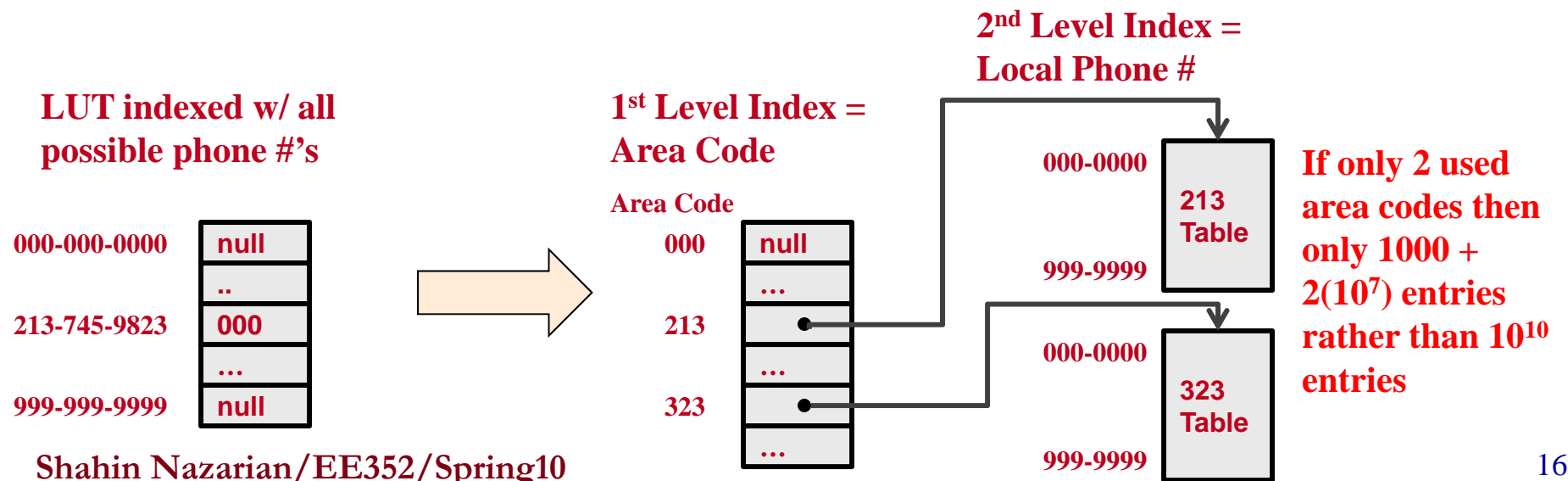
③ LUT indexed w/ all possible phone #'s

000-000-0000	null
...	..
213-745-9823	000
...	...
999-999-9999	null

Could Work
 Easy to index & find but **LARGE**
 (1 access)

Analogy for Page Tables

- Can we use the table indexed using all possible phone numbers (because it only requires 1 access) but somehow reduce the size especially since much of it is unused?
- Use a 2-level organization
 - 1st level LUT is indexed on area code and contains pointers to 2nd level tables
 - 2nd level LUTs indexed on local phone numbers and contains contact ID entries



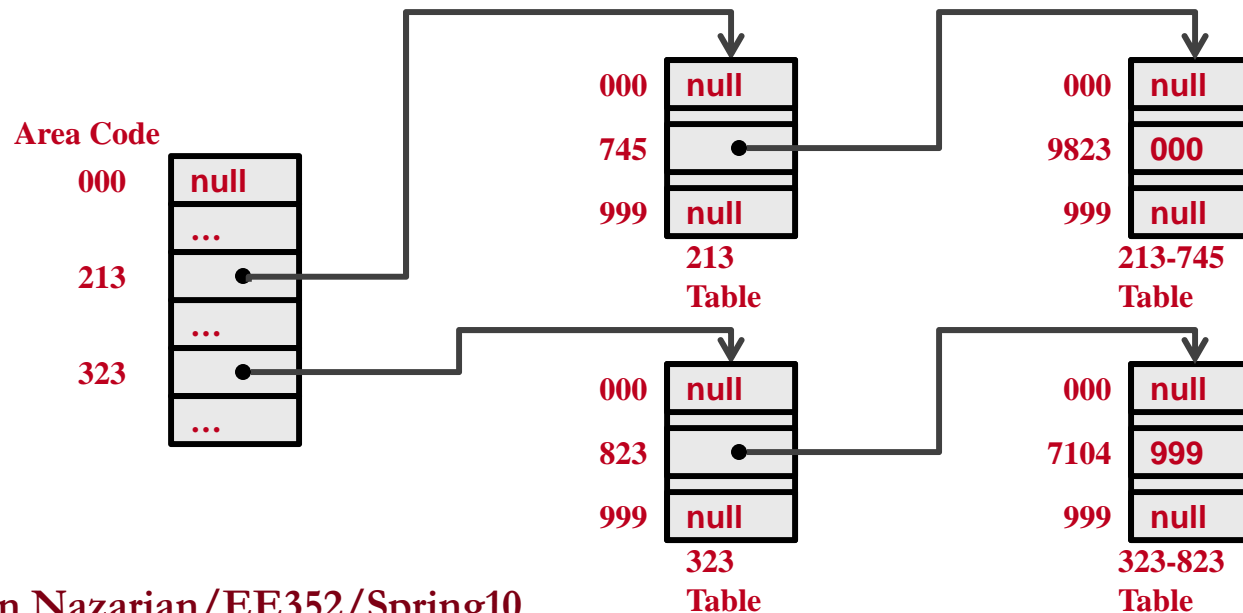
Analogy for Page Tables

- Could extend to 3 levels if desired
 - 1st Level = Area code and pointers to 2nd level tables
 - 2nd Level = First 3-digits of local phone and pointers to 3rd level tables
 - 3rd Level = Contact IDs

1st Level Index =
Area Code

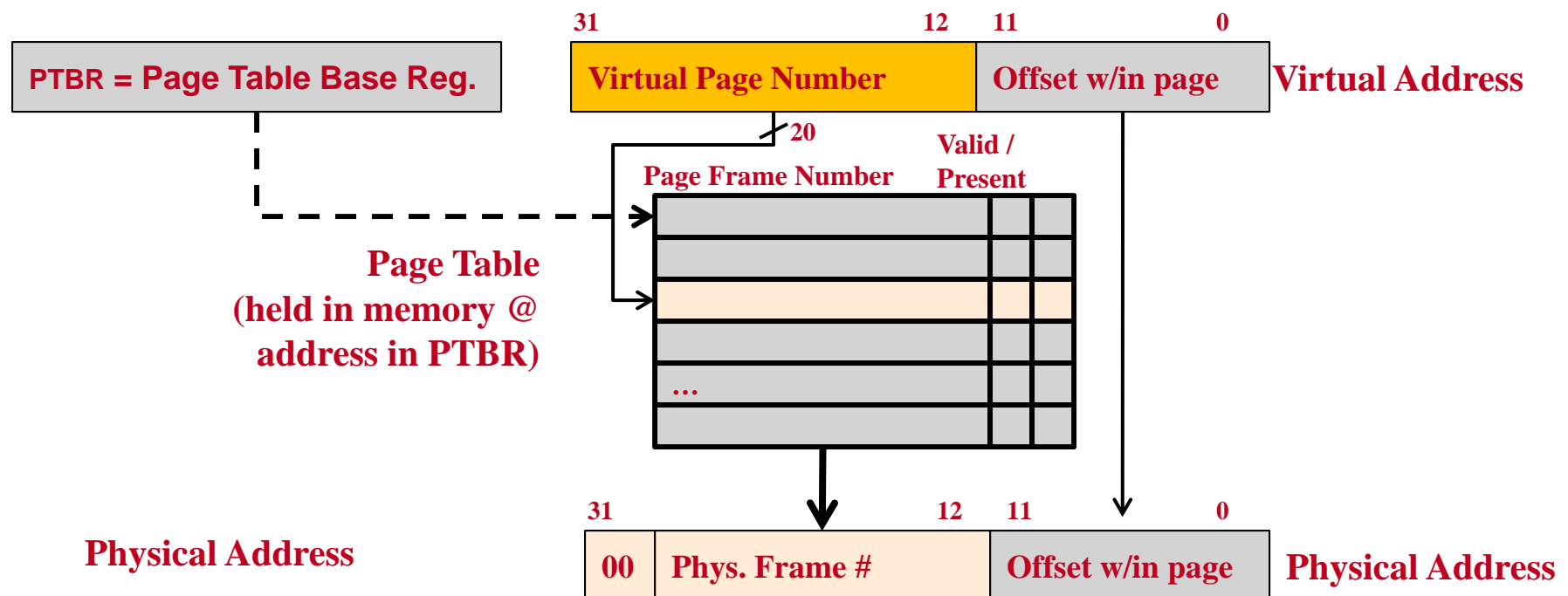
2nd Level Index =
Local Phone #

3rd Level Index =
Local Phone #



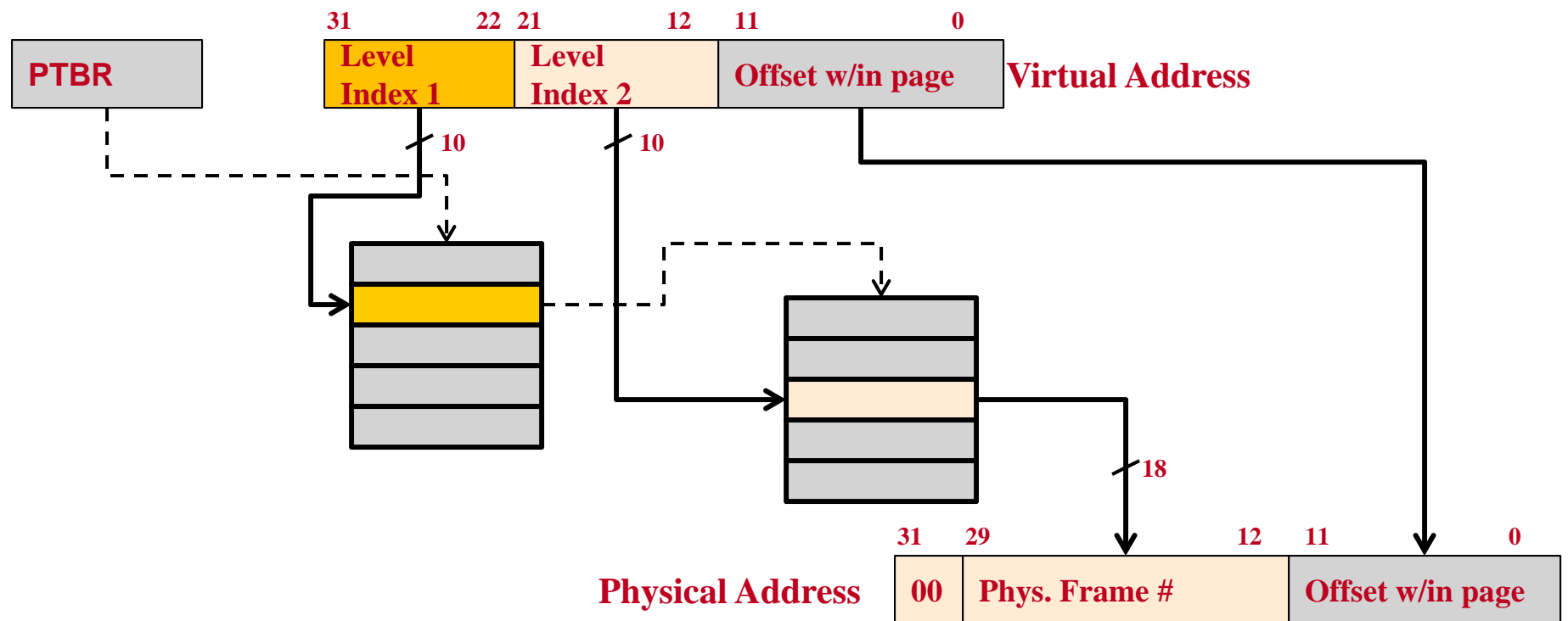
Page Tables

- Used to perform virtual to physical address translation
 - Allows virtual memory page placement to be fully associative in physical memory
- Page table (one per process) is a LUT held in main memory, indexed on virtual address



Multi-Level Page Table

- VPN is broken into fields to index each level of the multi-level page table

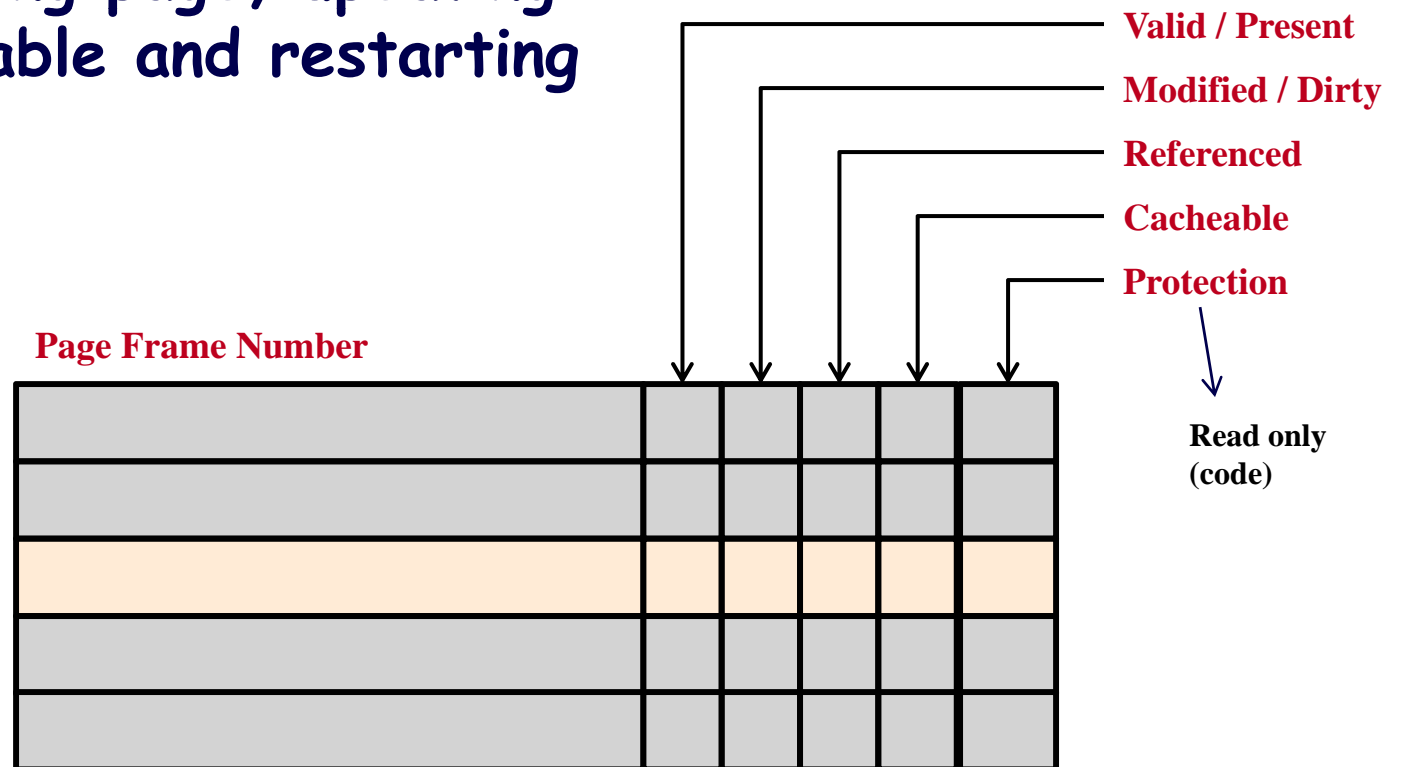


State

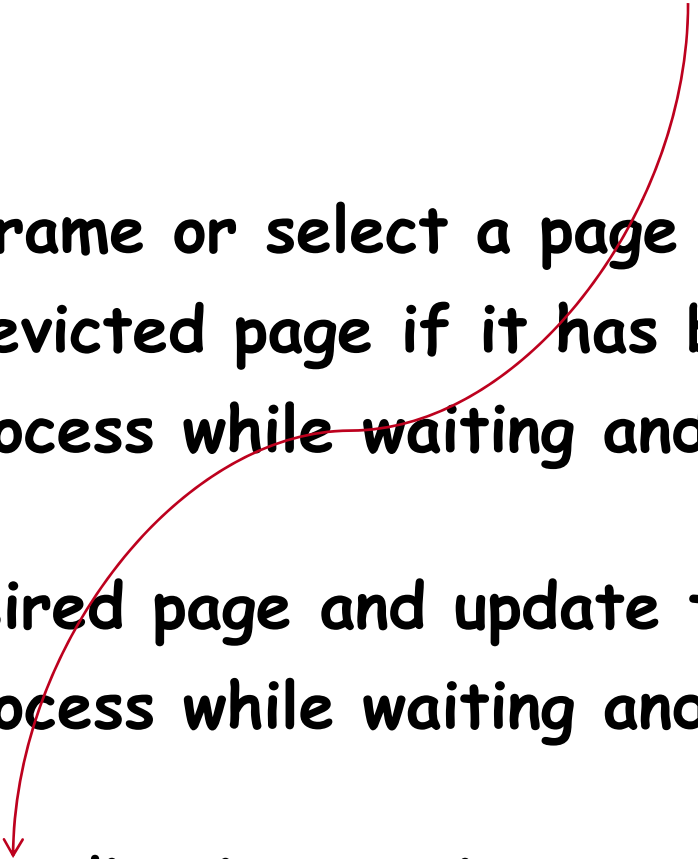
- The page table together with the program counter and the registers specifies the **state** of a program
- If we want to allow another program to use the processor, we must save this state. Later after restoring this state, the program can continue execution. We often refer to this state as a process. The process is active if it is in possession of the processor
- The process' address space, hence all the data it can access in memory is defined by its page table which resides in memory
- Rather save the entire page table, the OS simply loads the page table register to point to the page table of the process it wants to make active
- The OS is responsible for allocating the physical memory and updating the page tables so that virtual address spaces of different processes do not collide
- The use of separate page tables also provides protection of one process from another

Handling Page Faults

- Valid bit (1 = desired page in memory / 0 = page not present / page fault)
- On page fault, an exception will be posted and the OS will be responsible for bringing in the missing page, updating the page table and restarting execution



Page Fault Steps

- HW will...
 - Record the offending address, the EPC, and cause (page fault)
 - SW will...
 - Pick an empty frame or select a page to evict
 - Writeback the evicted page if it has been modified
 - May block process while waiting and yield processor
 - Bring in the desired page and update the page table
 - May block process while waiting and yield processor
 - Restart the offending instruction
- 

Page Replacement Policies

- Possible algorithms: LRU, FIFO, Random
- Since page misses are so costly (slow) we can afford to spend sometime keeping statistics to implement LRU
- Implementing exact LRU would require updating statistics every access (using some kind of timestamp). This is too much to do in HW and we don't want to use SW when we have hits
- HW will implement simple mechanism that allows SW to implement a pseudo-LRU algorithm
 - HW will set the “Referenced” bit when a page is used
 - At certain intervals, SW will use these reference bits to keep statistics on which pages have been used in that interval and then clear the reference bits
 - On replacement, these statistics can be used to find the pseudo-LRU page

Cache & VM Comparison

	Cache	Virtual Memory
Block Size	16-64B	4 KB – 64 MB
Mapping Schemes	Direct or Set Associative	Fully Associative
Miss handling and replacement	HW	SW
Replacement Policy	Full LRU if low associativity / Random is also used	Pseudo-LRU can be implemented

Performance Issues

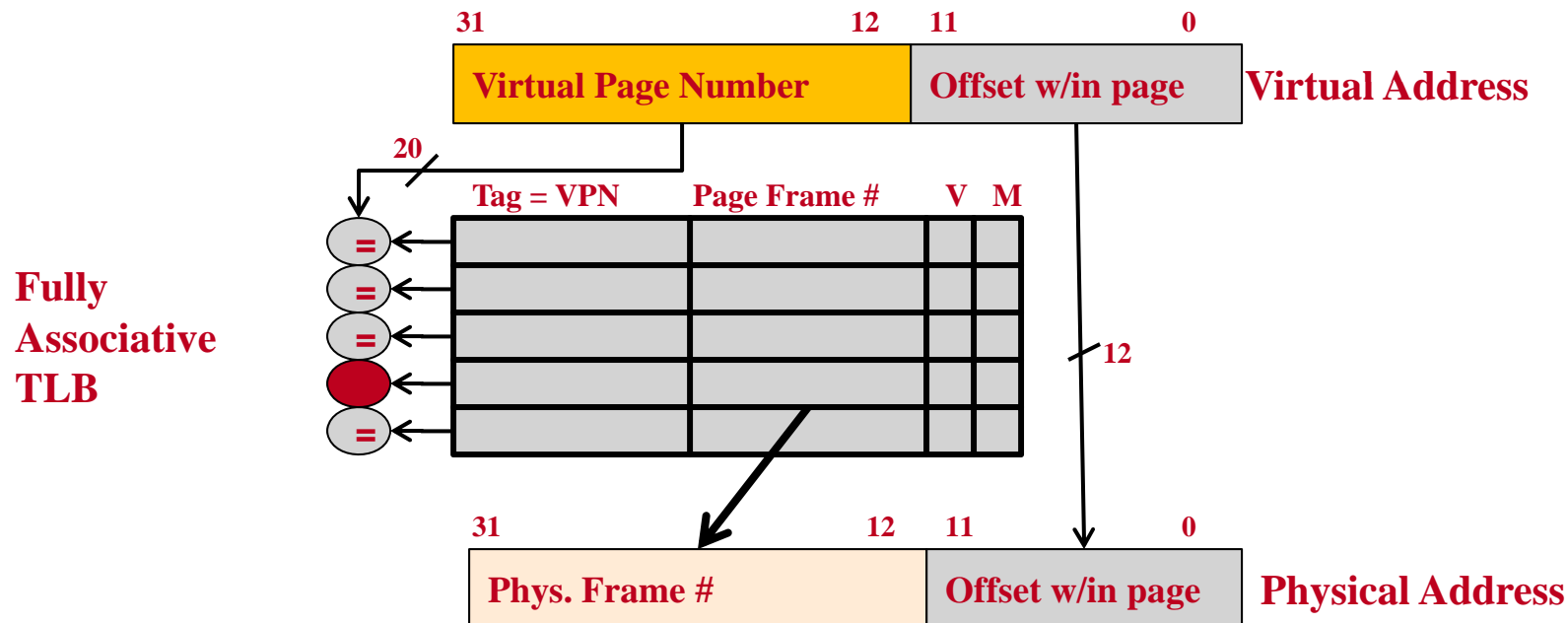
- Let cache hits = 10ns, memory accesses=100ns
- Assume a program makes an access to data located in cache...
 - Without VM, only requires 10ns cache access time
 - With VM, it must first be translated via the page table
 - If a single-level, one access to the page table (MM) = 100ns
 - If two-levels, two access to the page tables = 200ns
 - Finally, physical address can access cache = 10ns (if hit)
 - Total time equals $100 * L + 10$ (where $L = \#$ of Level of Page Table)
- Translation is extremely costly as currently implemented!!!

TLB - Motivation

- Since the page tables are stored in main memory, every memory access by a program can take at least twice as long; one to obtain the physical address and a second to get the data
- The key factor here is the locality, i.e., when a translation for a virtual page number is used, it will probably be needed again in the near future, because the references to the words on that page have both temporal and spatial locality
- Therefore modern processors include a special cache that keeps track of recently used translations. This special translation cache is referred to as a **translation-lookaside buffer (TLB)**
- In case of a TLB miss, either the page exists in MM, then TLB miss indicates that only the translation is missing, but if the page is not in MM, the TLB miss indicates a true page fault. Note that TLB misses will be much more frequent than page faults (cause TLB size is a lot smaller than # of pages in MM)
- TLB misses can be handled either in HW or SW with a little performance difference between the two approaches

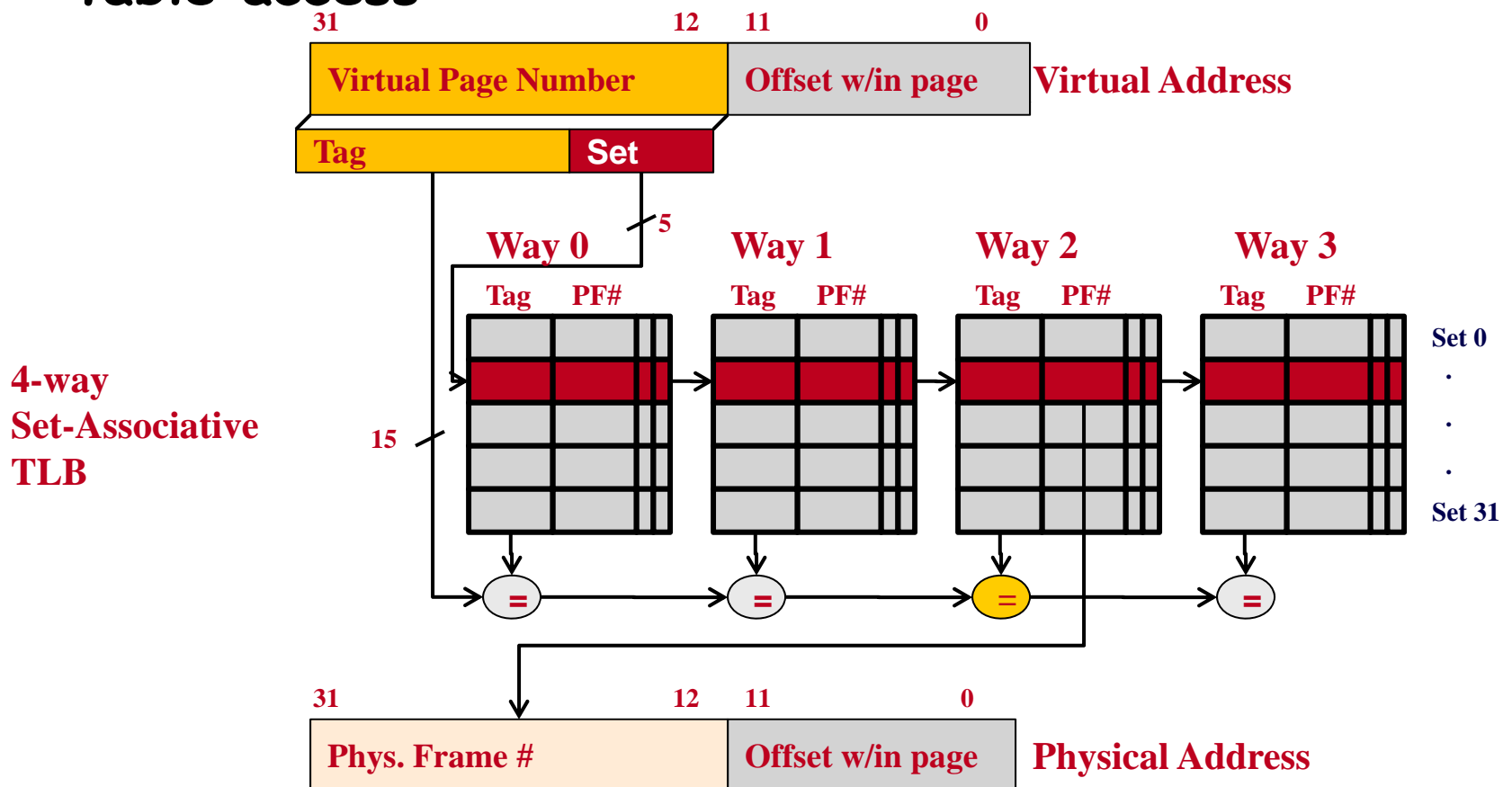
Translation Lookaside Buffer (TLB)

- Let's create a cache for translations = TLB
 - Needs to be small (64-128 entries) with high degree of associativity (at least 4-way and many times fully associative)



A 4-Way Set Associative TLB

- 128 entry 4-way SA TLB (set field indexes each "way"). $128/4=32$ sets
 - On hit, page frame # supplied quickly w/o page table access



TLB Issues

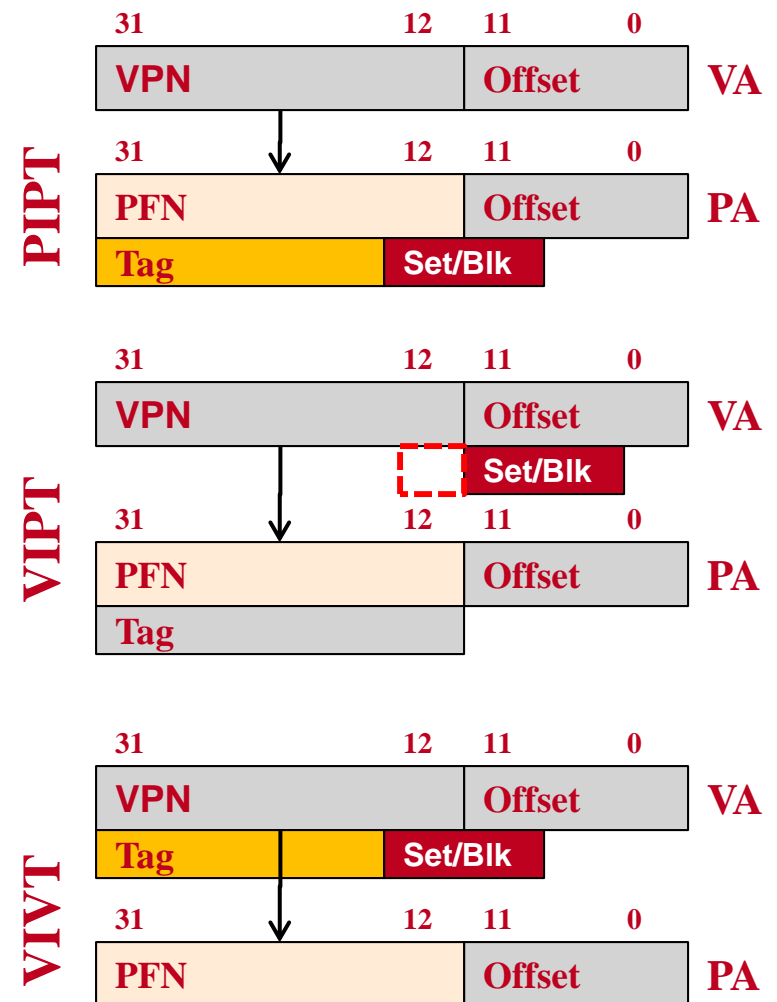
- Because of high degree of associativity and limited working set of pages (usually) we can get VERY HIGH hit rates for the TLB
 - Variable page size settable by OS to allow for different working set sizes
 - Example: 64 TLB entries and 4 KB pages = 256KB
- Often times, separate TLBs for instruction and data address translation
- On context switch (change of process), what needs to be done to TLB?
 - Invalidate all entries (Virtual addresses are non-unique)
 - Append Process ID to tags

[Optional] Cache Addressing with VM

- **Cache review**
 - Set or block field indexes LUT holding tags
 - 2 steps to determine hit:
 - Index (lookup) to find tags (using block or set bits)
 - Compare tags to determine hit
 - Sequential connection b/n indexing and tag comparison
- Rather than waiting for address translation and then performing this two step hit process, can we overlap the translation and portions of the hit sequence?
 - Yes if we choose page size, block size, and set/direct mapping carefully

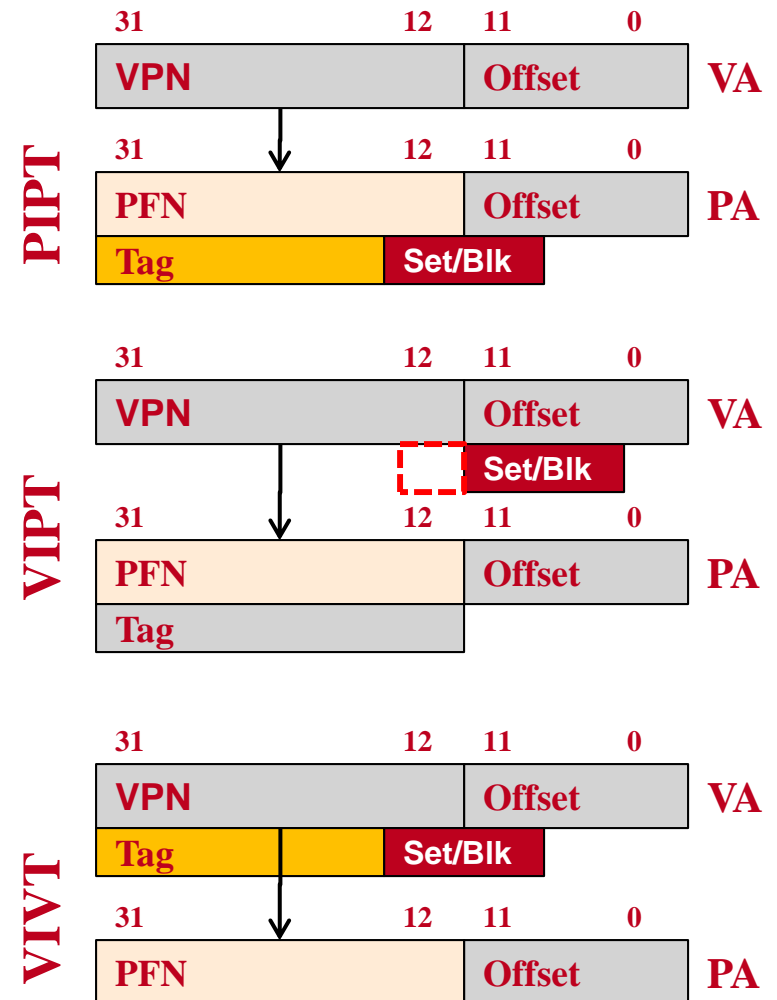
[Optional] Cache Index/Tag Options

- **Physically indexed, physically tagged (PIPT)**
 - Wait for full address translation
 - Then use physical address for both indexing and tag comparison
- **Virtually indexed, physically tagged (VIPT)**
 - Use portion of the virtual address for indexing then wait for address translation and use physical address for tag comparisons
 - Easiest when index portion of virtual address w/in offset (page size) address bits, otherwise aliasing may occur



[Optional] Cache Index/Tag Options (Cont.)

- **Virtually indexed, virtually tagged (VIVT)**
 - Use virtual address for both indexing and tagging...No TLB access unless cache miss
 - Requires invalidation of cache lines on context switch or use of process ID as part of tags

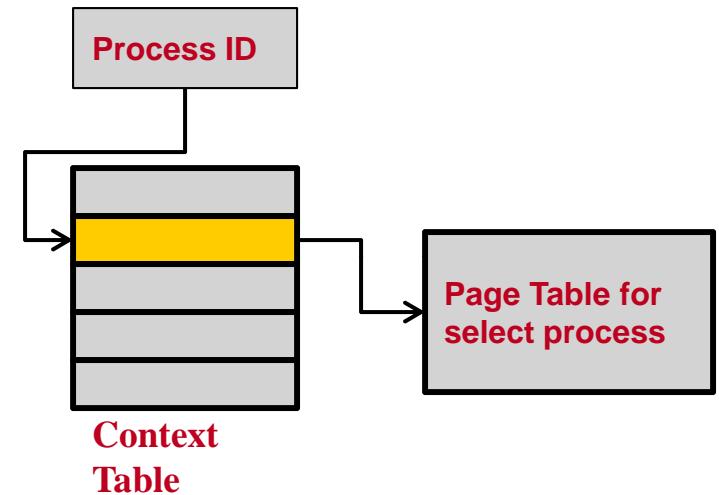


Cache, VM, and Main Memory

TLB	VM	Cache	Possible Y/N & Description
Hit	Hit	Hit	Possible: best-case scenario
Hit	Hit	Miss	Possible: TLB hits (hit in VM is implied), then cache miss
Miss	Hit	Hit	TLB misses, then hits in page table, then cache hit
Miss	Hit	Miss	TLB misses, then hits in page table, then cache miss
Miss	Miss	Miss	TLB misses, then page fault, then miss in cache
Hit	Miss	Miss	Impossible: cannot hit in TLB if page not present
Hit	Miss	Hit	Impossible: cannot hit in TLB if page not present
Miss	Miss	Hit	Impossible: data cannot be in cache if page not present

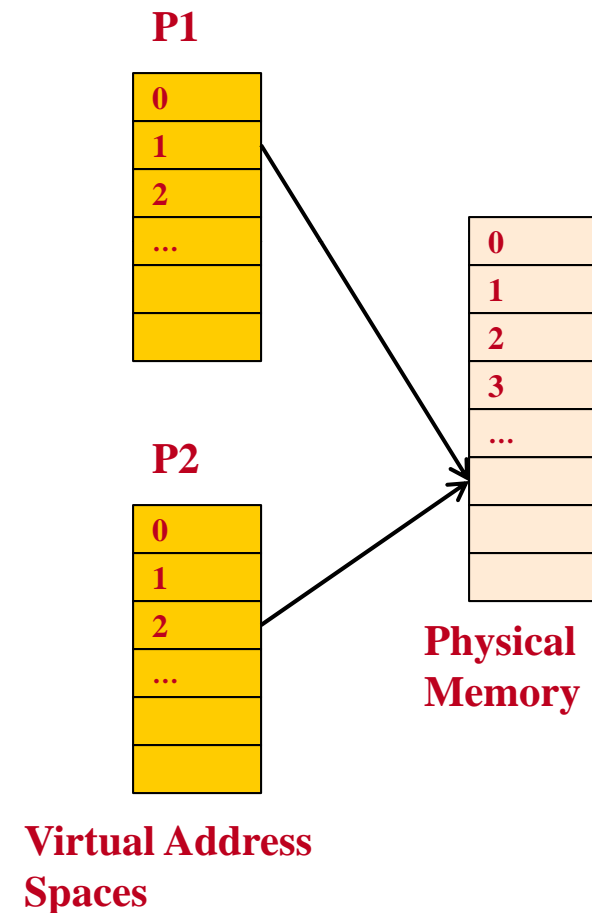
Protection

- To ensure each program/process can access only its page table we can
 - Ensure PTBR is changed on each context switch
 - Replace the single PTBR with a context table of pointers to process' page tables and use a Process ID as index
- We also have "protection" bits for each page (read/write/execute) for greater access control
 - Execute = Text/Code Sections



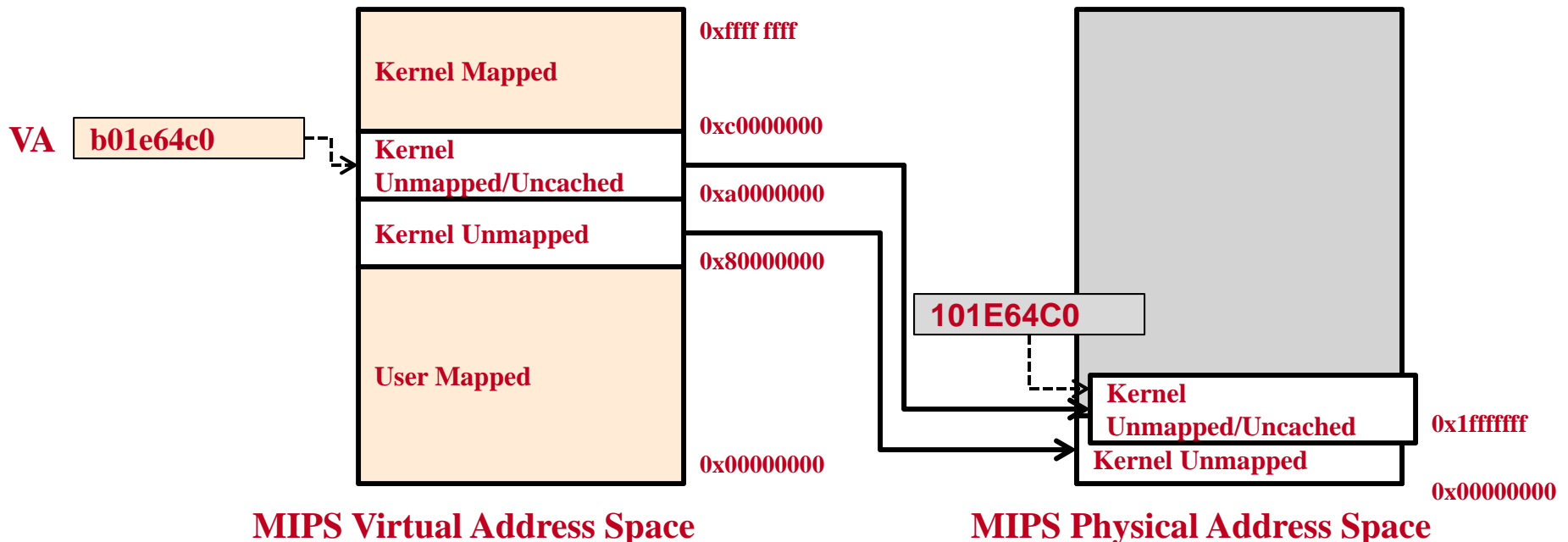
Shared Memory

- In current system, all memory is private to each process
- To share memory between two processes, the OS can allocate an entry in each process' page table to point to the same physical page
- Can use different protection bits for each page table entry (e.g. P1 can be R/W while P2 can be read only)



Unmapped Regions of Memory

- Certain areas of the virtual address space may be unmapped, meaning they won't be translated
 - In MIPS, these areas are windows to physical area @ 0x0 and up
 - 0x80000000-0x9fffffff & 0xa0000000 - 0xbfffffff map to 0x0-0x1fffffff [MS 3-bits=0]
 - Uncached = accesses to those areas will bypass cache hierarchy



Virtual Memory System Examples

Microprocessor	AMD Opteron	P4	PPC 7447a
Virtual Address	48-bit	32- or 48-bit	52-bit
Physical Address	40-bit	36-bit	32- or 36-bit
TLB Entries (I/D/L2 TLB)	L1: 40/40 L2: 512/512	L1: 128/128	L1: 128 / 128
TLB Mapping	L1: Fully L2: 4-way SA	Fully (4-way)	2-way set associative
Min. Page Size	4 KB	4 KB	4 KB

Notes: Large VAs include ASID (process IDs) and other segment information

Sources: H&P, “CO&D”, 3rd ed., Freescale.com