Memory: Address Translation, Paging, Caching, and TLBs

Lecture 14

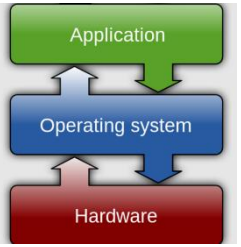
Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4103/

Address Translation

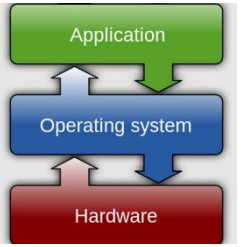
Next Objective

- Dive deeper into the concepts and mechanisms of memory sharing and address translation
- Enabler of many key aspects of operating systems
  - Protection
  - Multi-programming
  - Isolation
  - Memory resource management
  - I/O efficiency
  - Sharing
  - Inter-process communication
  - Debugging
  - Demand paging



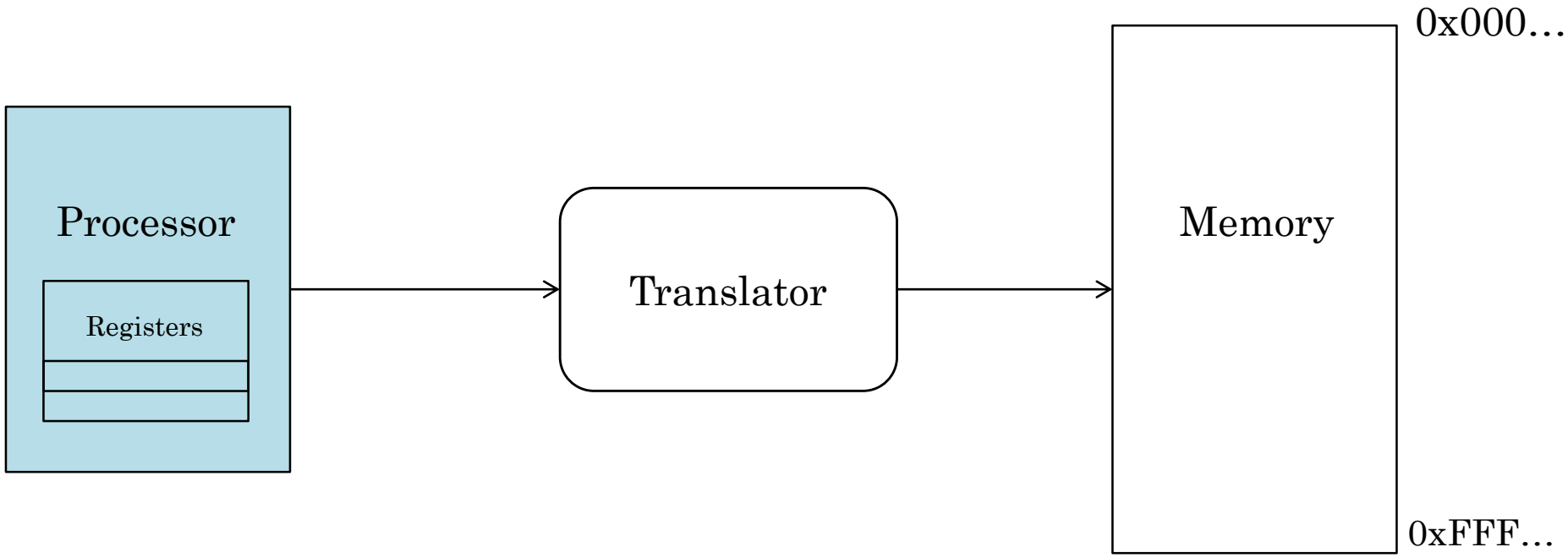
Recall: Four Fundamental OS Concepts

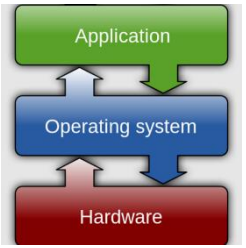
- Thread: Execution Context
  - Program Counter, Registers, Execution Flags, Stack
- Address Space (with Translation)
  - Program's view of memory is distinct from physical machine
- Process: Instance of a Running Program
  - Address space + one or more threads + …
- Dual-Mode Operation and Protection
  - Only the "system" can access certain resources
  - Combined with translation, isolates programs from each other



Key OS Concept: Address Space

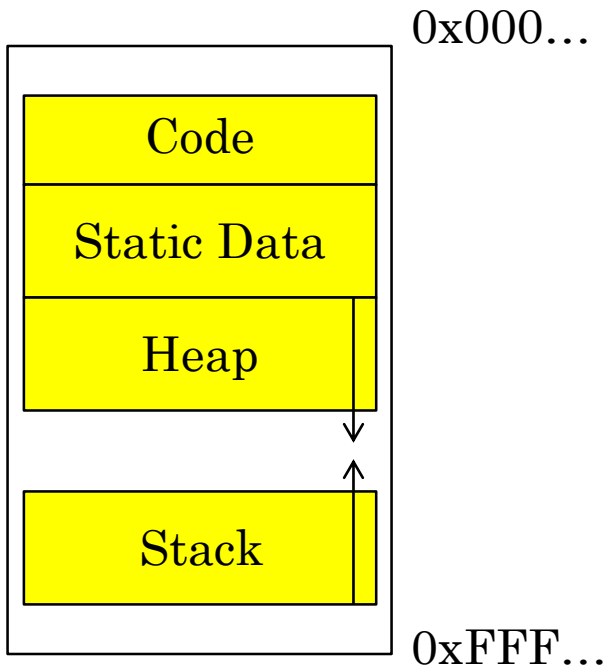
- Program operates in an address space that is distinct from the physical memory space of the machine

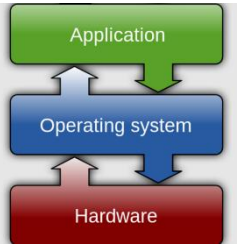




Recall: Address Space

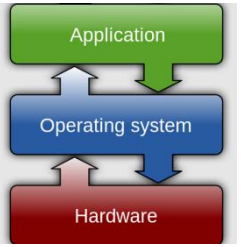
- Definition: Set of accessible addresses and the state associated with them
  - 2^{32} = ~4 billion on a 32-bit machine
- What happens when you read or write to an address?
  - Perhaps acts like regular memory
  - Perhaps causes I/O operation
    - (Memory-mapped I/O)
  - Causes program to abort (segfault)?
  - Communicate with another program
  - …





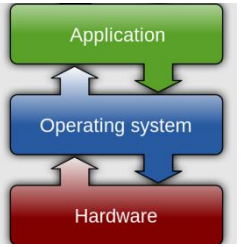
Recall: Interposing on Process Behavior

- OS interposes on process' I/O operations
  - How? All I/O happens via syscalls.
- OS interposes on process' CPU usage
  - How? Interrupt lets OS preempt current thread
- Question: How can the OS interpose on process' memory accesses?
  - Too slow for the OS to interpose every memory access
  - Translation: hardware support to accelerate the common case
  - Page fault: uncommon cases trap to the OS to handle

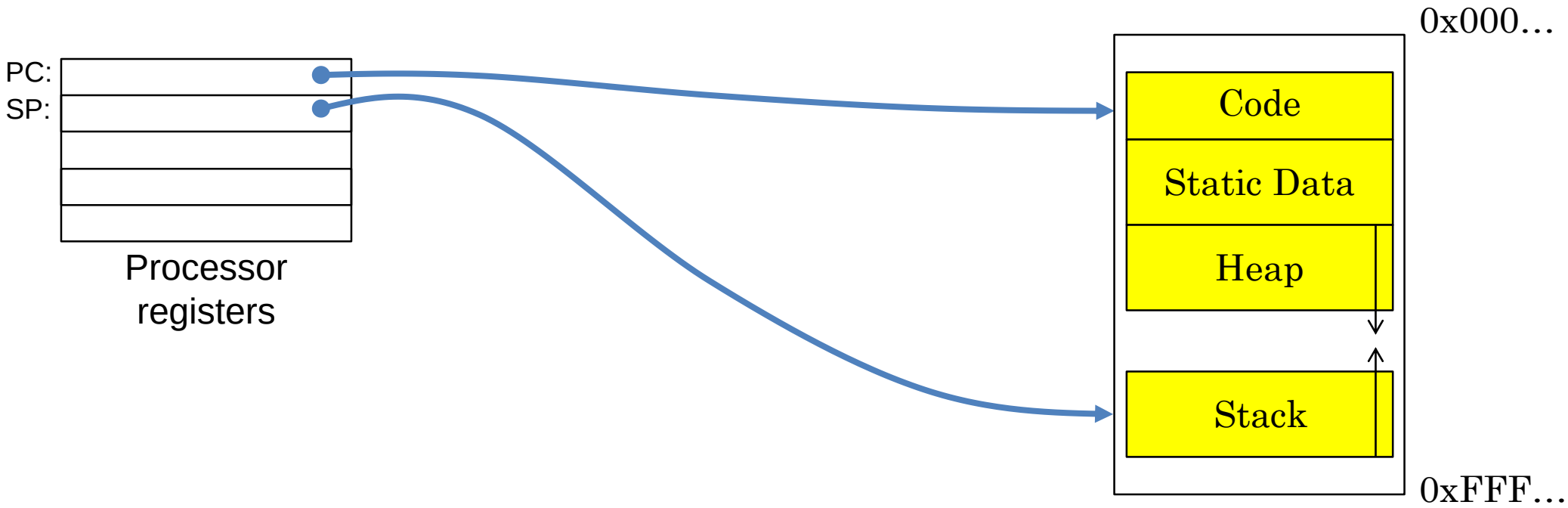


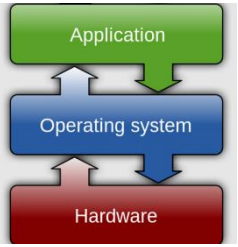
Recall: Important Aspects of Memory Translation

- Protection
  - Prevent access to private memory of other process or kernel
- Translation
  - Gives uniform view of memory to programs
  - Allows for efficient "tricks"
    - E.g., in implementation of fork()
- Controlled Overlap
  - Read-only data, execute-only shared libraries
  - Inter-process communication

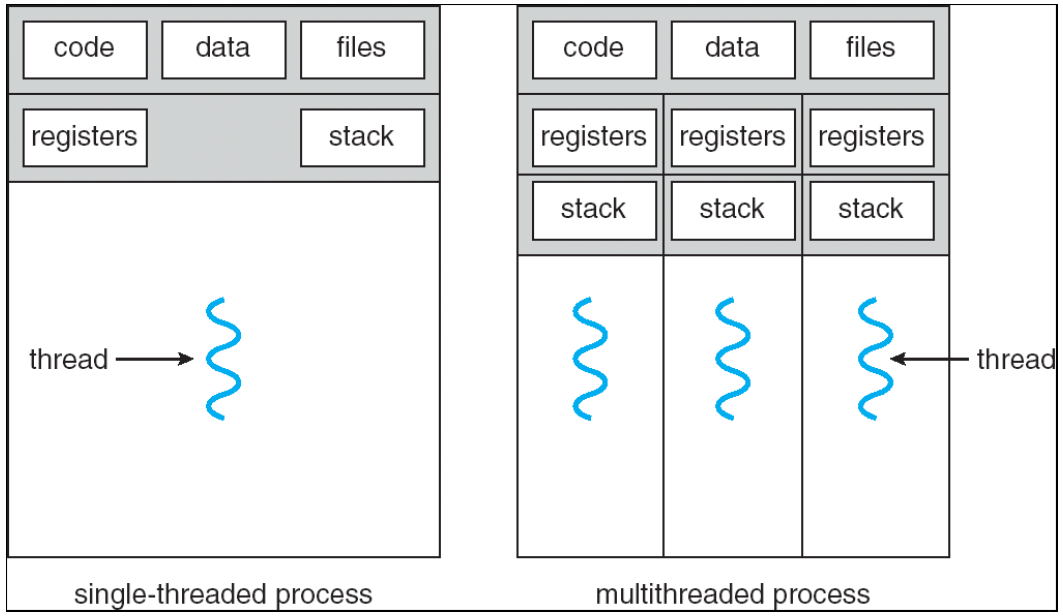


Recall: Typical Address Space Structure

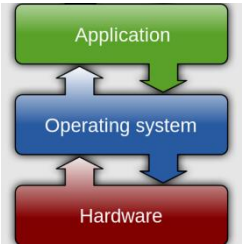




Recall: Single and Multithreaded Processes

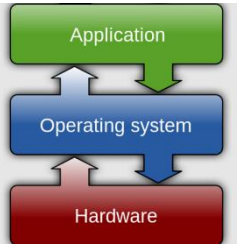


- Threads encapsulate concurrency
  - "Active" component
- Address space encapsulate protection:
  - "Passive" component
  - Keeps bugs from crashing the entire system
- Why have multiple threads per address space?



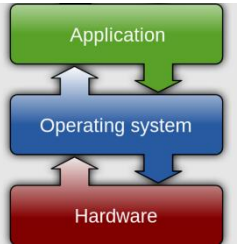
Important Aspects of Memory Multiplexing

- Protection
  - Prevent access to private memory of other process or kernel
- Translation
  - Gives uniform view of memory to programs
  - Allows for efficient "tricks"
    - E.g., in implementation of fork()
- Controlled Overlap
  - Read-only data, execute-only shared libraries
  - Inter-process communication



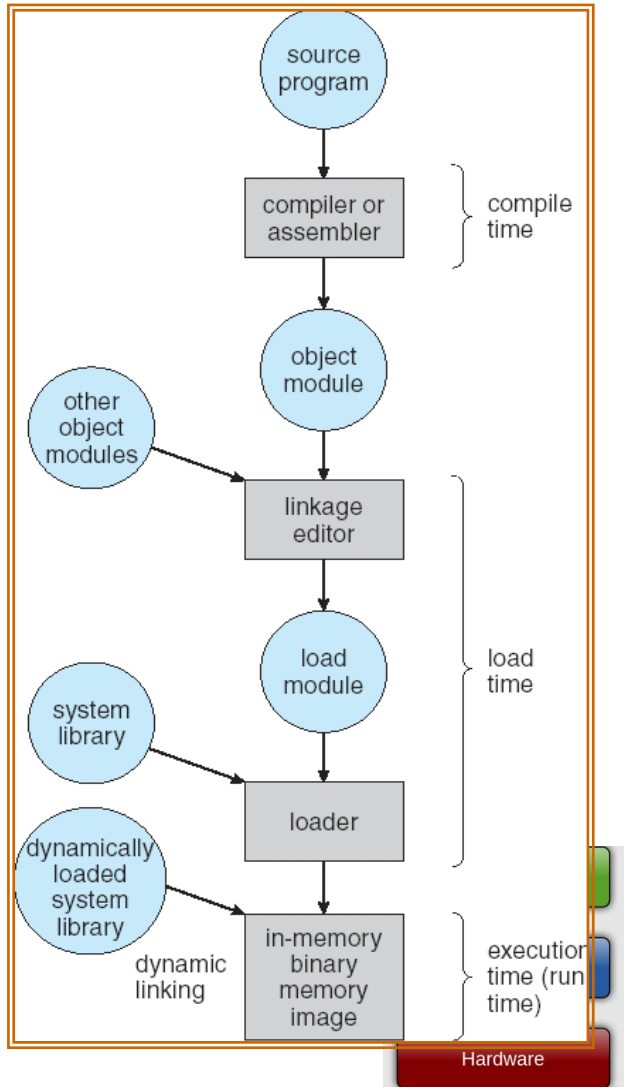
Alternative View: Interposing on Process Behavior

- OS interposes on process' I/O operations
  - How? All I/O happens via syscalls.
- OS interposes on process' CPU usage
  - How? Interrupt lets OS preempt current thread
- Question: How can the OS interpose on process' memory accesses?
  - Too slow for the OS to interpose every memory access
  - Translation: hardware support to accelerate the common case
  - Page fault: uncommon cases trap to the OS to handle



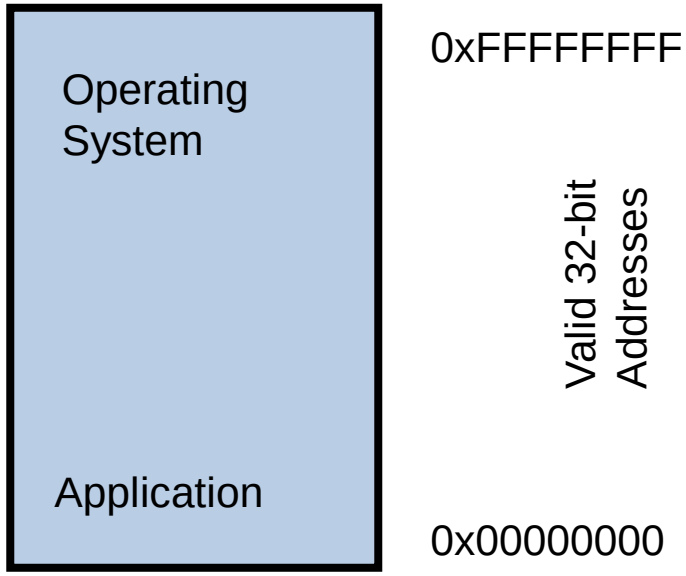
From Program to Process

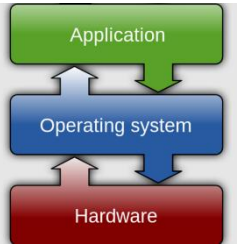
- Preparation of a program for execution involves components at:
  - Compile time (i.e., "gcc")
  - Link/Load time (UNIX "ld" does link)
  - Execution time (e.g., dynamic libs)
- Addresses can be bound to final values anywhere in this path
  - Depends on hardware support
  - Also depends on operating system
- Dynamic Libraries
  - Linking postponed until execution
  - Small piece of code (i.e. the stub), locates appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine



Uniprogramming: One Process at a Time

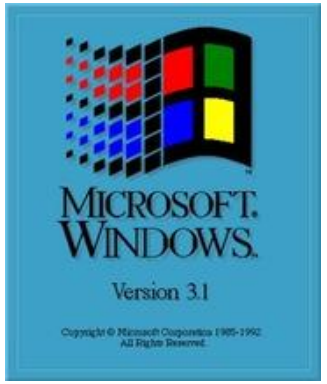
- No Translation or Protection
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address
  - Application given illusion of dedicated machine by giving it reality of a dedicated machine

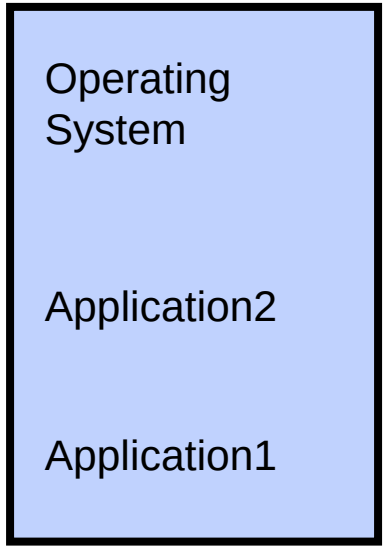




Primitive Multiprogramming

- Multiprogramming without Translation or Protection
- Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
  - Everything adjusted to memory location where OS put program
  - Translation done by a linker-loader (relocation)
- No protection!

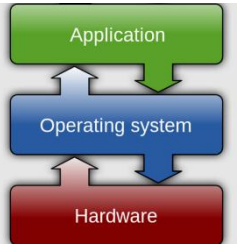




0xFFFFFFFF

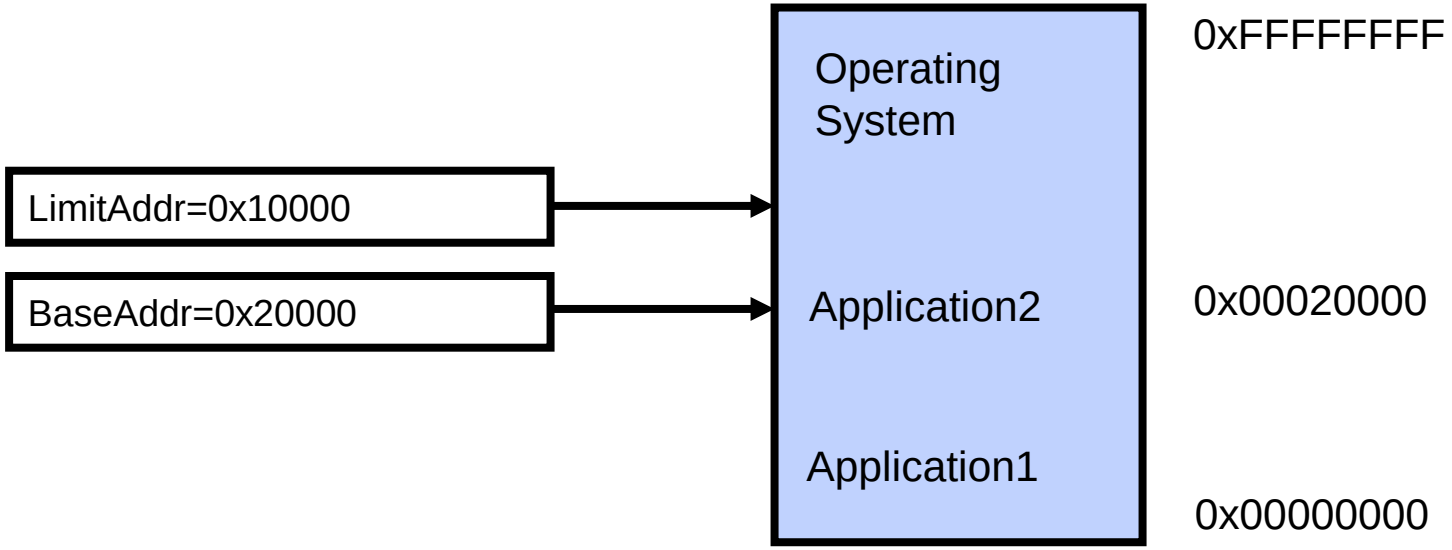
0x00020000

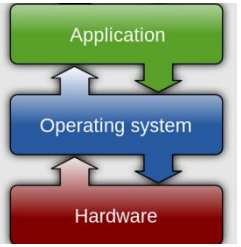
0x00000000



Multiprogramming with Protection

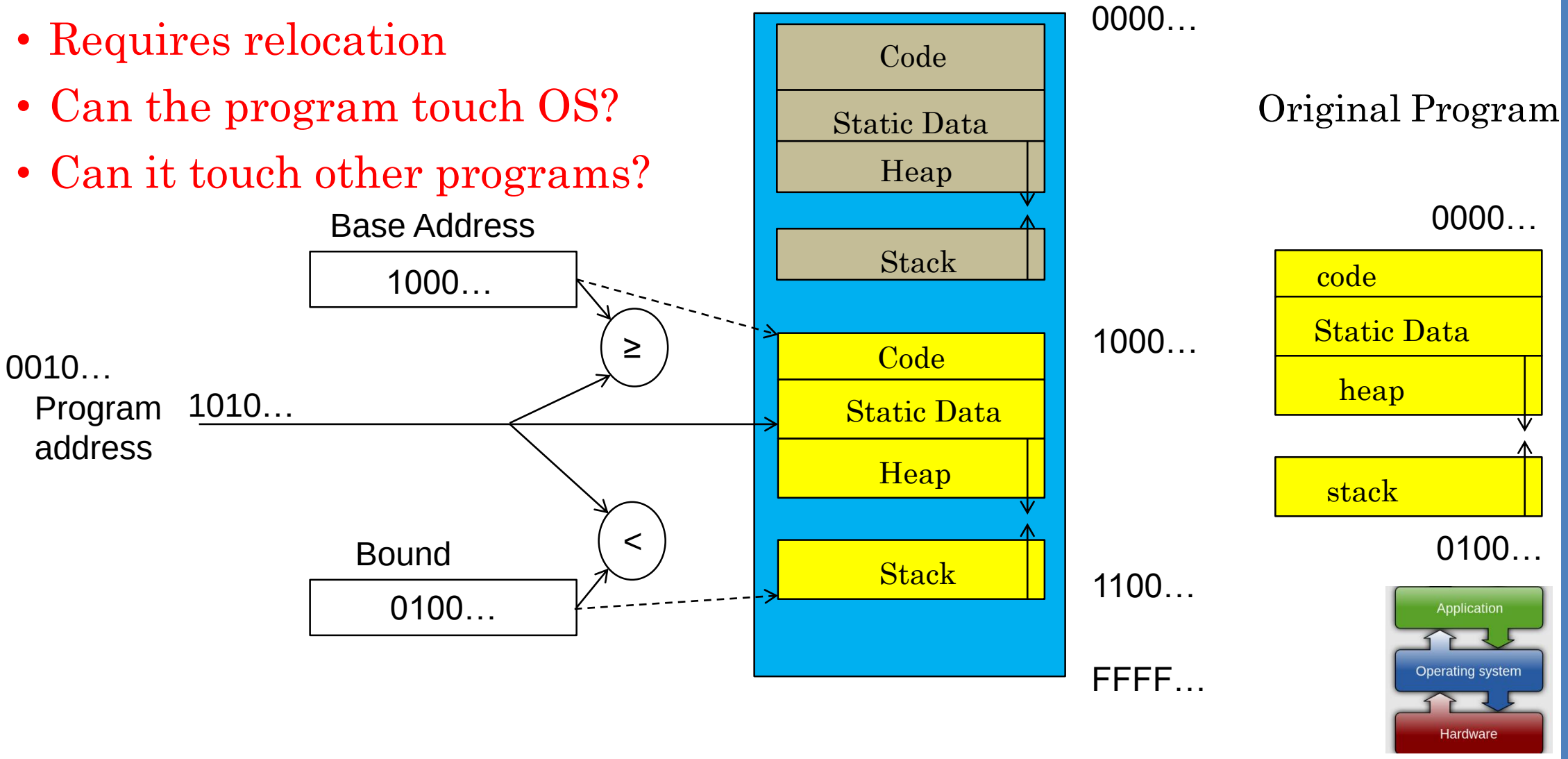
- Can we protect programs from each other without translation?
  - Yes: Base and Bound!
  - Used by, e.g., Cray-1 supercomputer



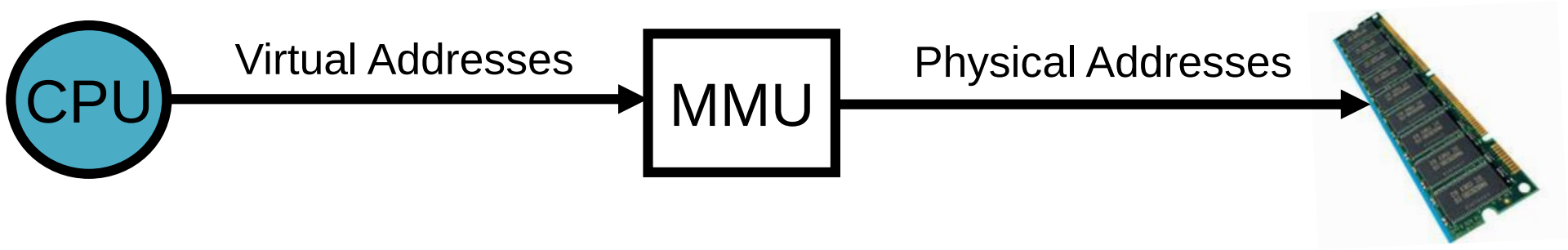


Recall: Base and Bound (no Translation)

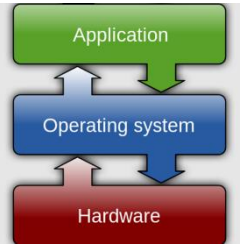
- Requires relocation
- Can the program touch OS?
- Can it touch other programs?



General Translation

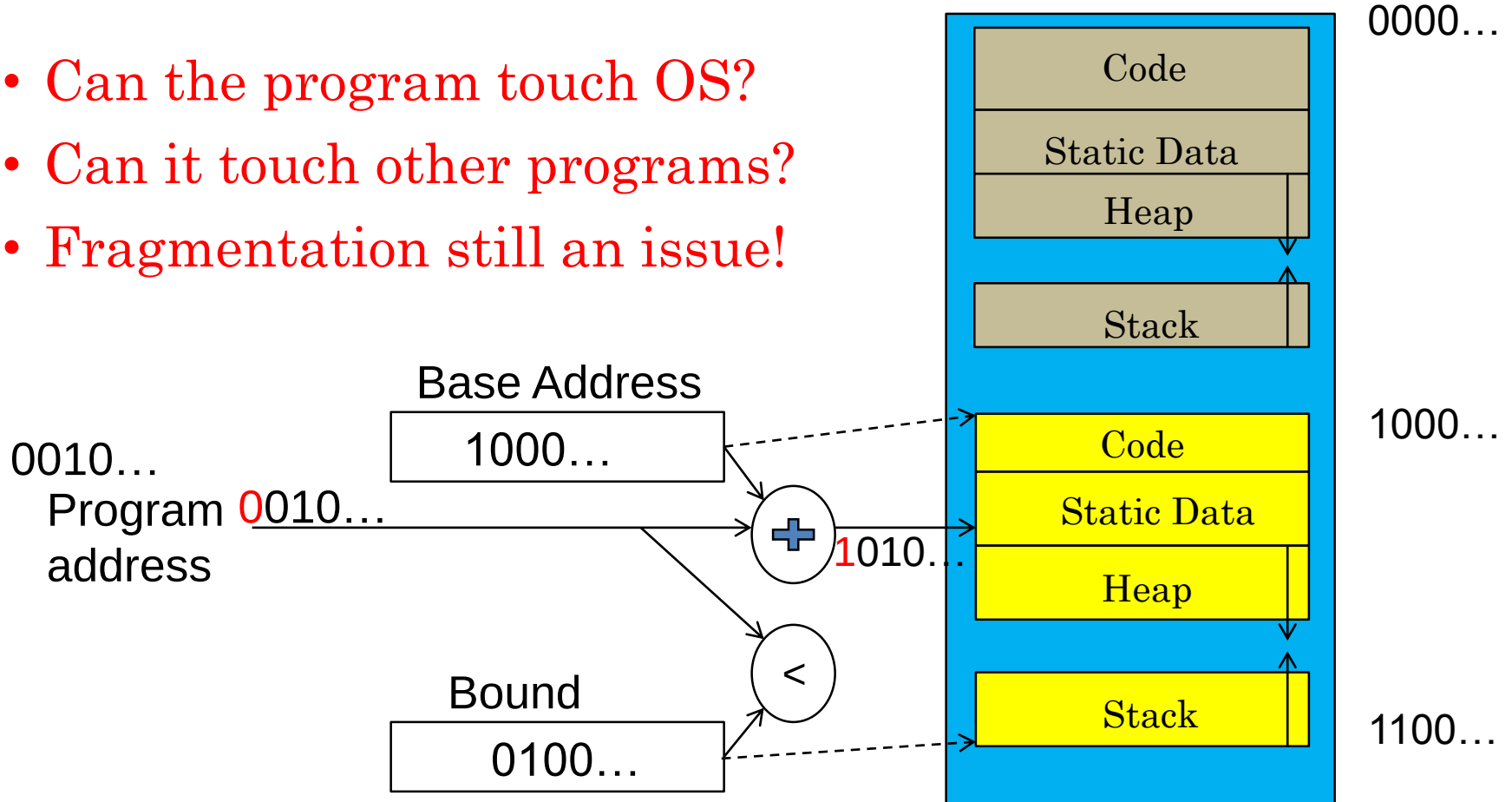


- Two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Hardware translator (Memory Management Unit or MMU) converts between the two views
- With translation, every program can be linked/loaded into same region of user address space

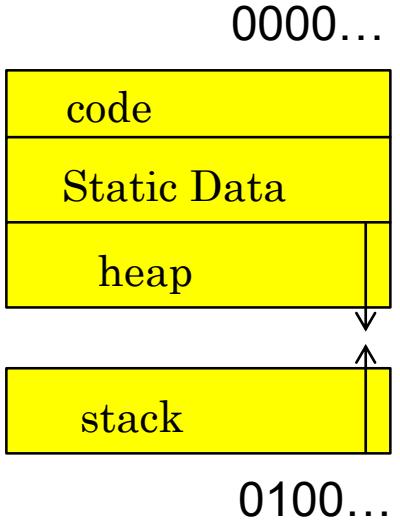


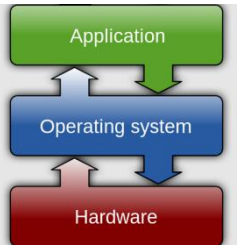
Recall: Base and Bound (with Translation)

- Can the program touch OS?
- Can it touch other programs?
- Fragmentation still an issue!

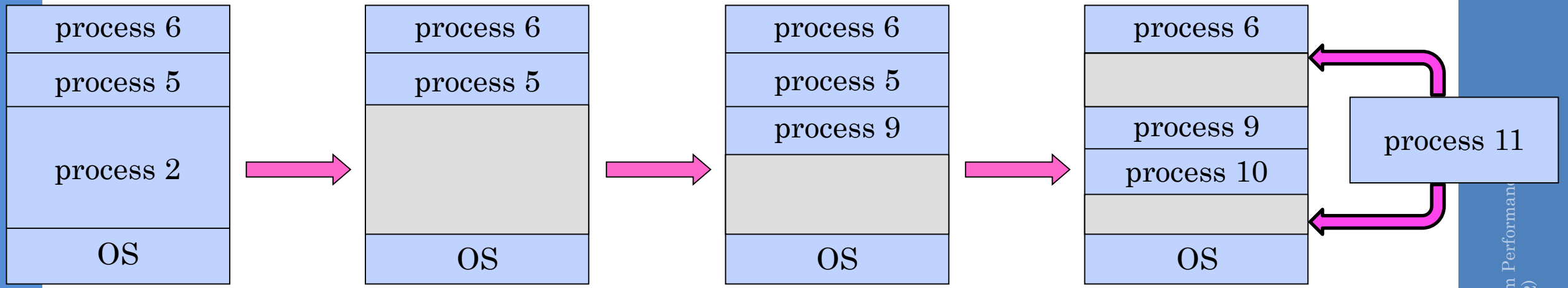


Original Program

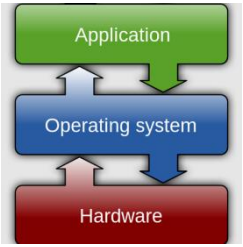




Issues with Simple Base and Bound

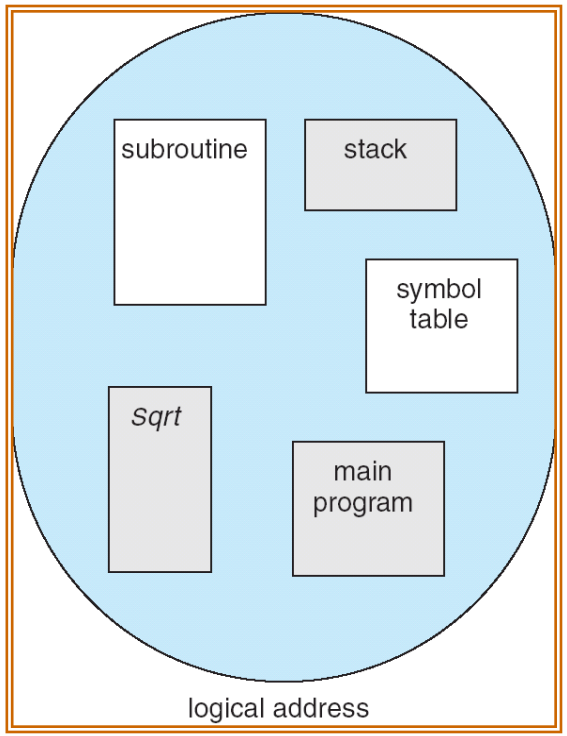


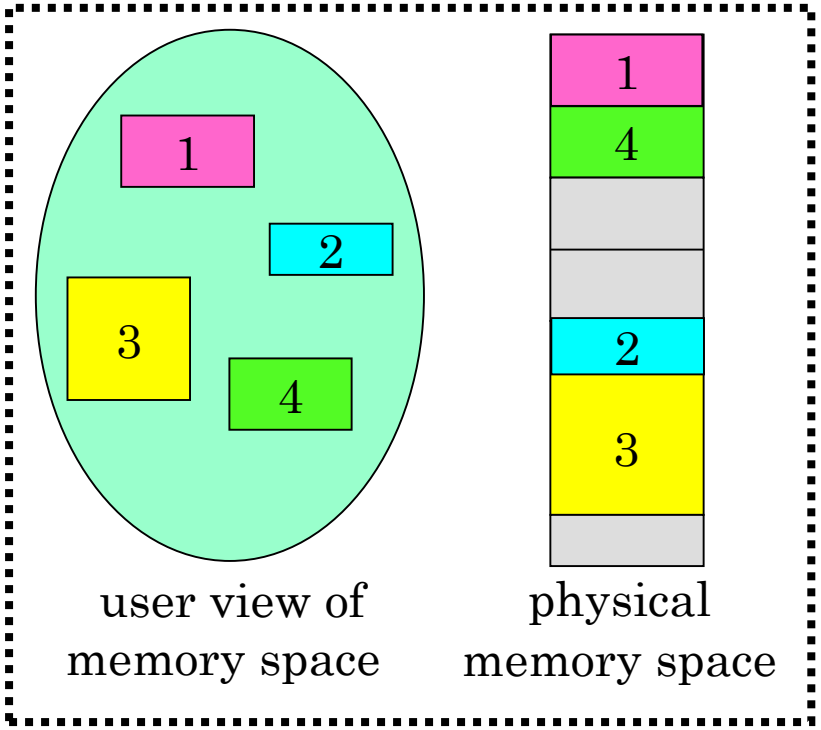
- Fragmentation problem over time
- No support for sparse address space
- Hard to do interprocess sharing
  - E.g., to share code



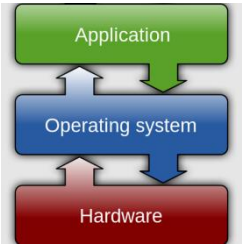
Segmentation

Segmentation

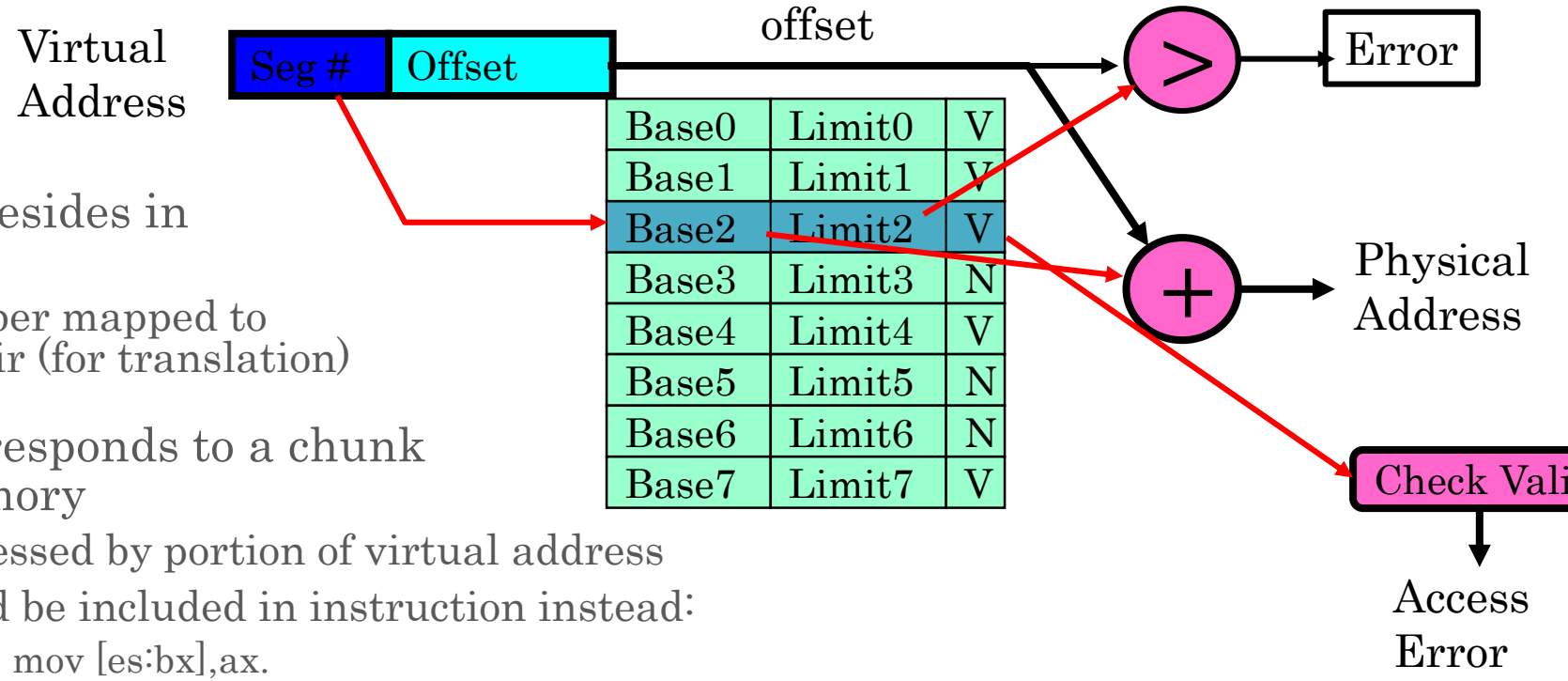




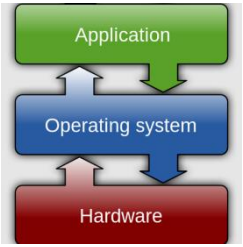
- Program's view of memory: multiple separate segments
- Each segment is given a region of contiguous memory
  - Has a base and limit
- Memory address consists of segment ID and offset



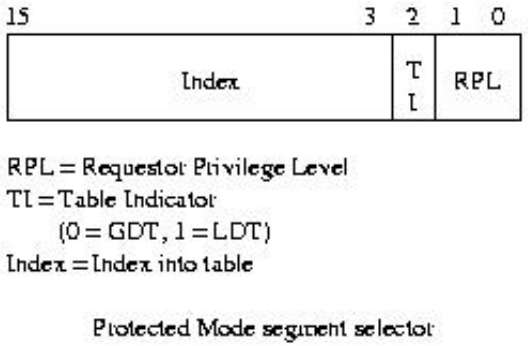
Hardware Support for Segmentation



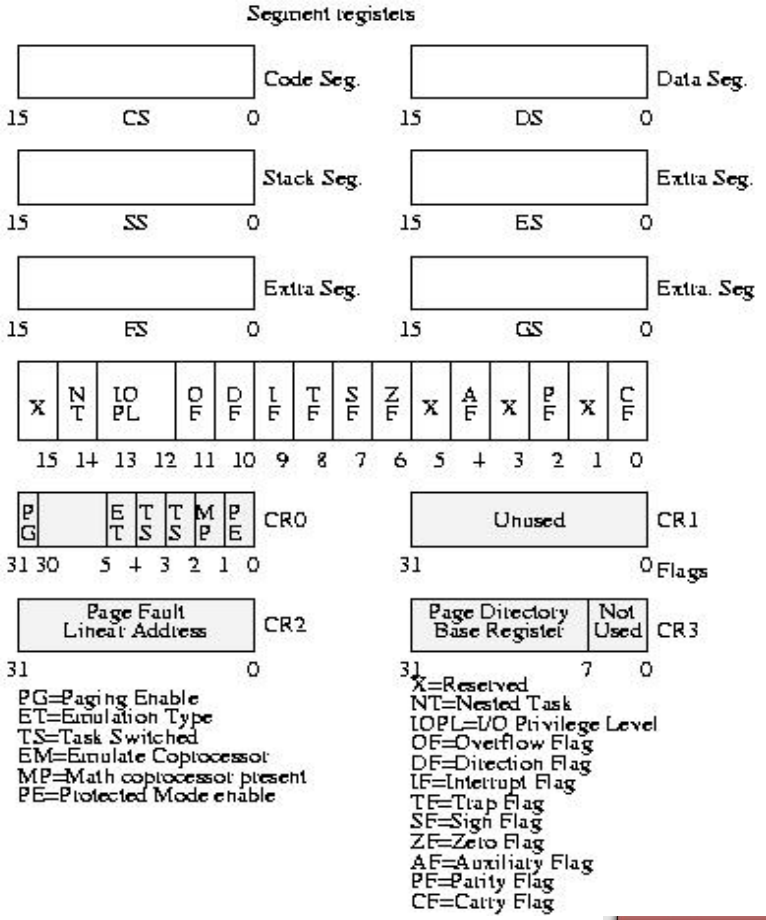
- Segment map resides in processor
  - Segment number mapped to base/bound pair (for translation)
- Each entry corresponds to a chunk of physical memory
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - x86 Example: mov [es:bx],ax.
- What is "V/N" (valid / not valid)?
  - Can mark segments as invalid; requires check as well



Intel x86 Special Registers



80386 Special Registers

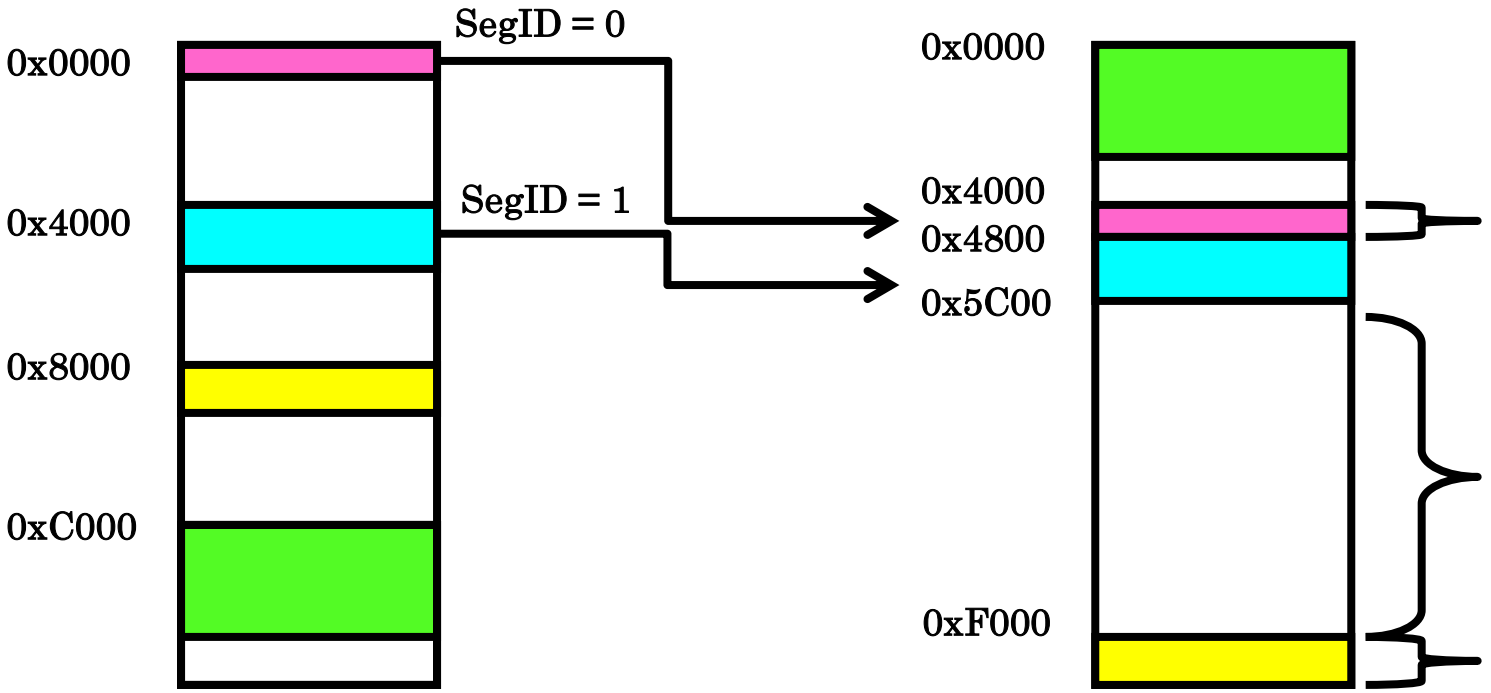


- Segmentation can't be just "turned off"
- What if we just want to use paging?
  - Set base and bound to all of memory, in all segments

Example: Four Segments

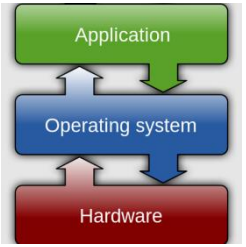
Virtual Address Format:

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



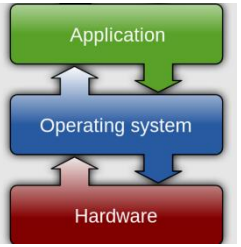
Virtual Address Space

Physical Address Space

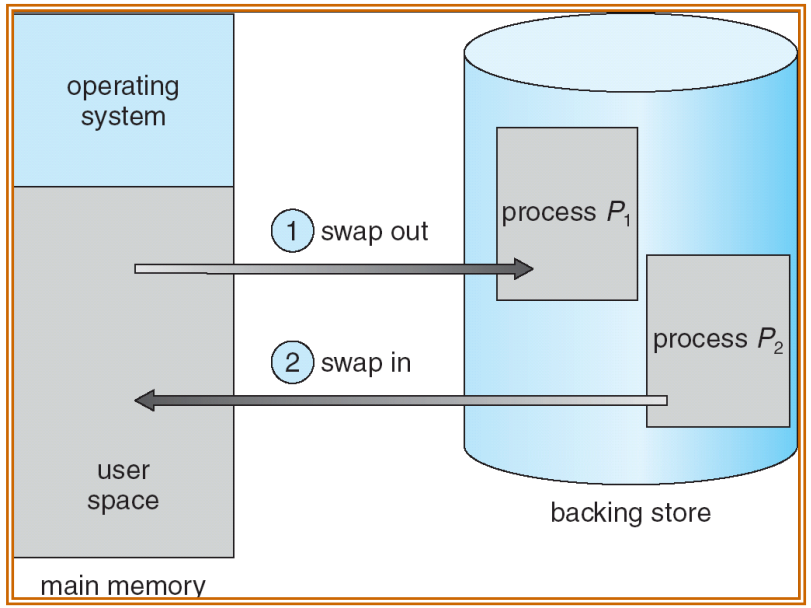


Observations about Segmentation

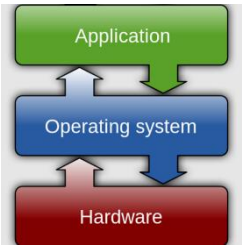
- Translation on every instruction fetch, load or store
- Virtual address space has holes
  - Segmentation efficient for sparse address spaces
- When it is OK to address outside valid range?
  - This is how the stack (and heap?) is allowed to grow
  - For instance, stack takes fault, system automatically increases size of stack
- Need protection mode in segment table
  - For example, code segment would be read-only
  - Data and stack would be read-write (stores allowed)
- 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")



What if not all segments fit in memory?

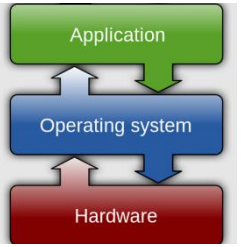


- Extreme form of Context Switch: Swapping
  - In order to make room for next process, some or all of the previous process is moved to disk
    - Likely need to send out complete segments
  - This greatly increases the cost of context-switching
- What might be a desirable alternative?
  - Some way to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



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

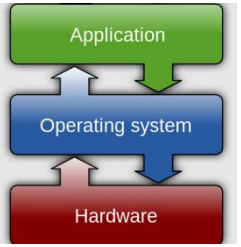


Paging

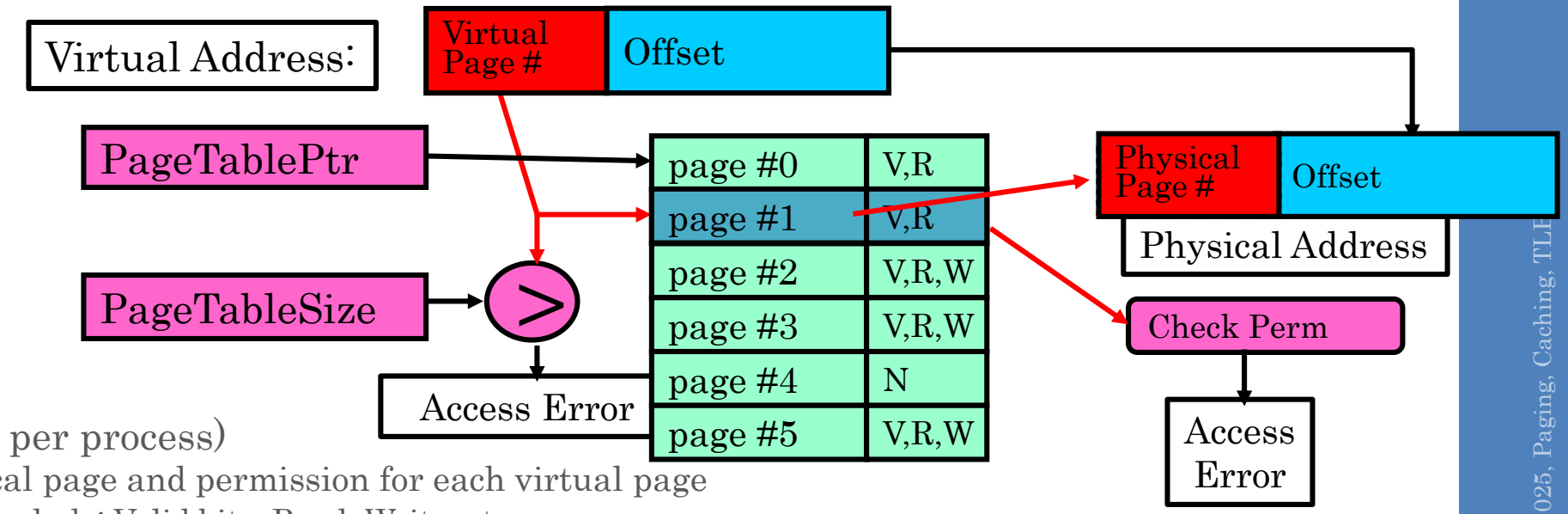
Paging: Fixed-Size Chunks of Memory

- Divide up physical memory into equal-size chunks called page frames
- Divide up virtual memory into equal-size chunks called pages
- Key idea: each physical page frame can contain any page

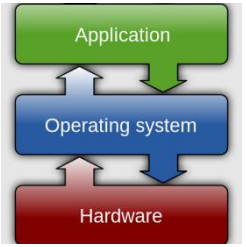
- No external fragmentation!
- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    - Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment



Hardware Support for Paging

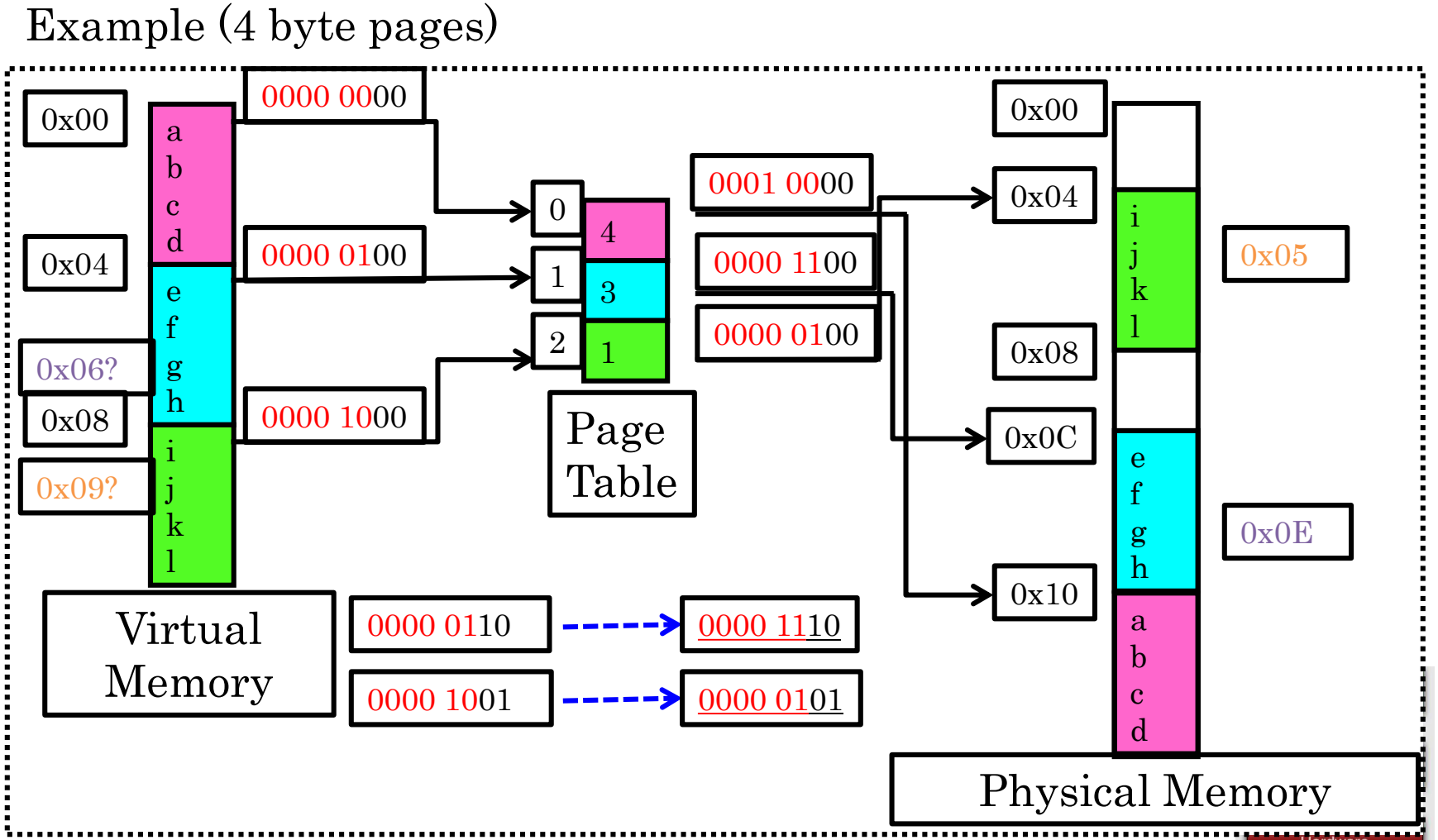


- Page Table (One per process)
  - Contains physical page and permission for each virtual page
    - Permissions include: Valid bits, Read, Write, etc.
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
  - Virtual page # is the index into the page table
    - Physical page # copied from table into physical address

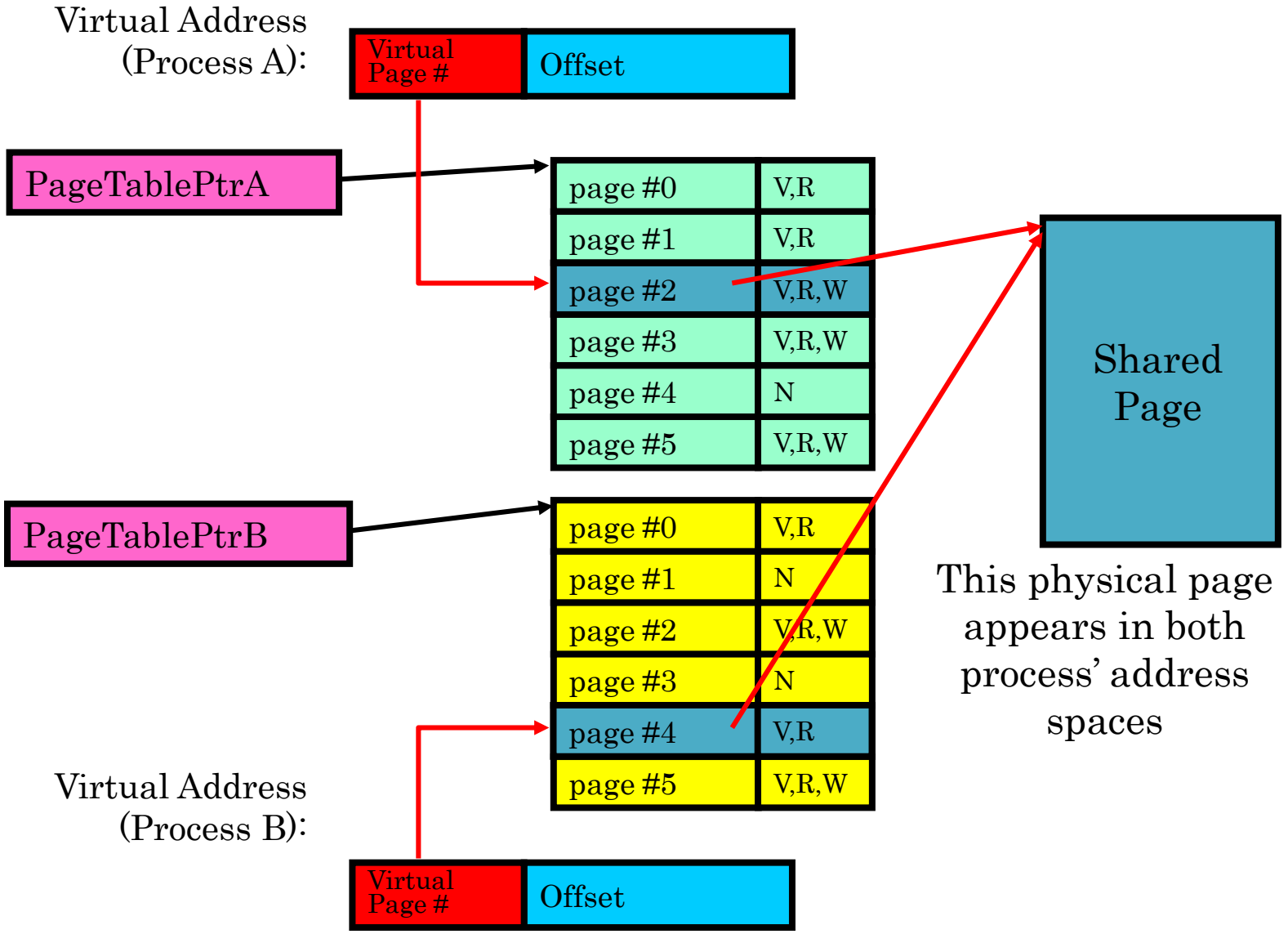


Simple Page Table Example

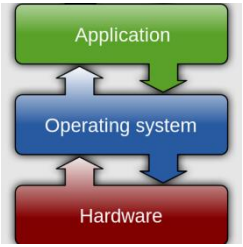
- What is the physical address for
  - Virtual address 0x06?
  - Virtual address 0x09?



What About Sharing?



- Where is page sharing used?
  - Kernel data mapped into each process
  - Different processes running the same binary
  - User-level system libraries
  - Shared pages as IPC



Example: Memory Layout for Linux 32-bit*

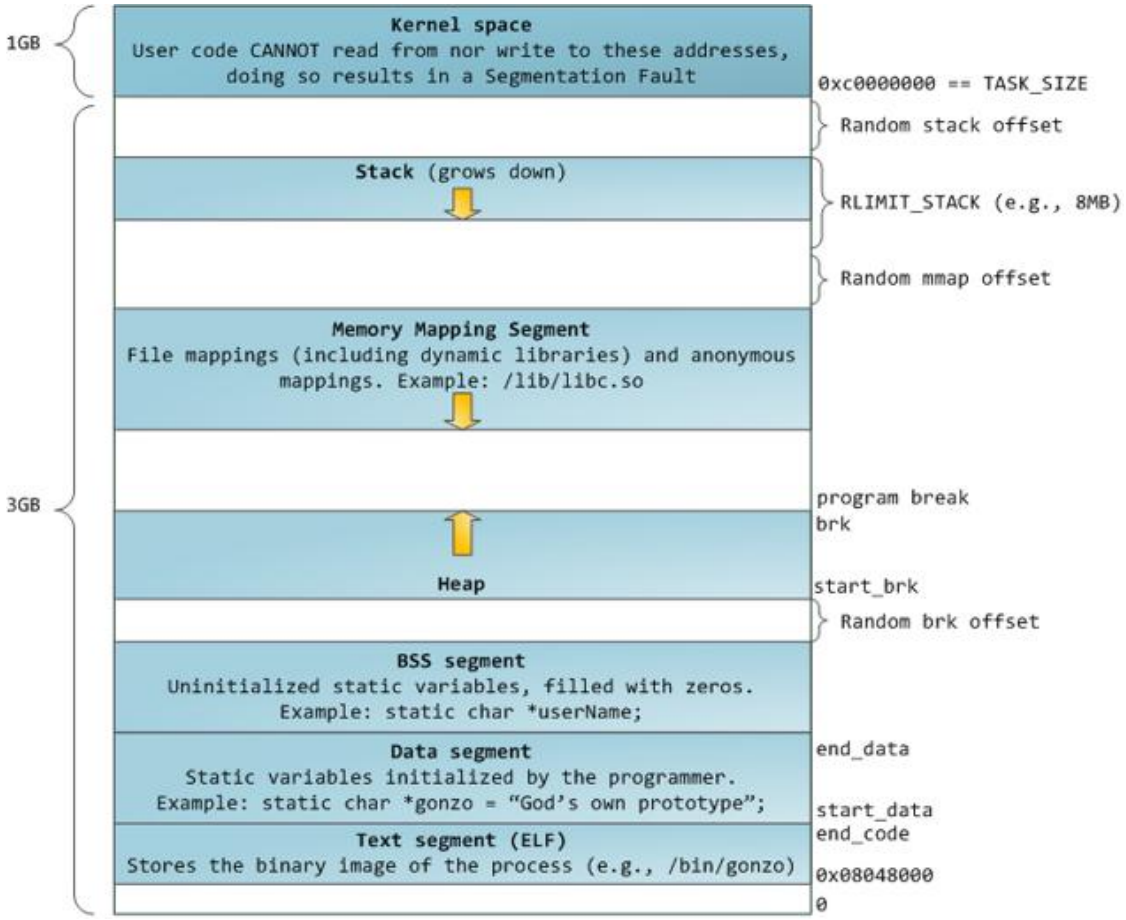
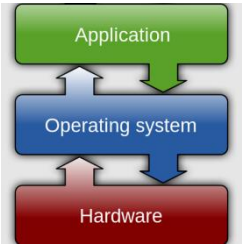
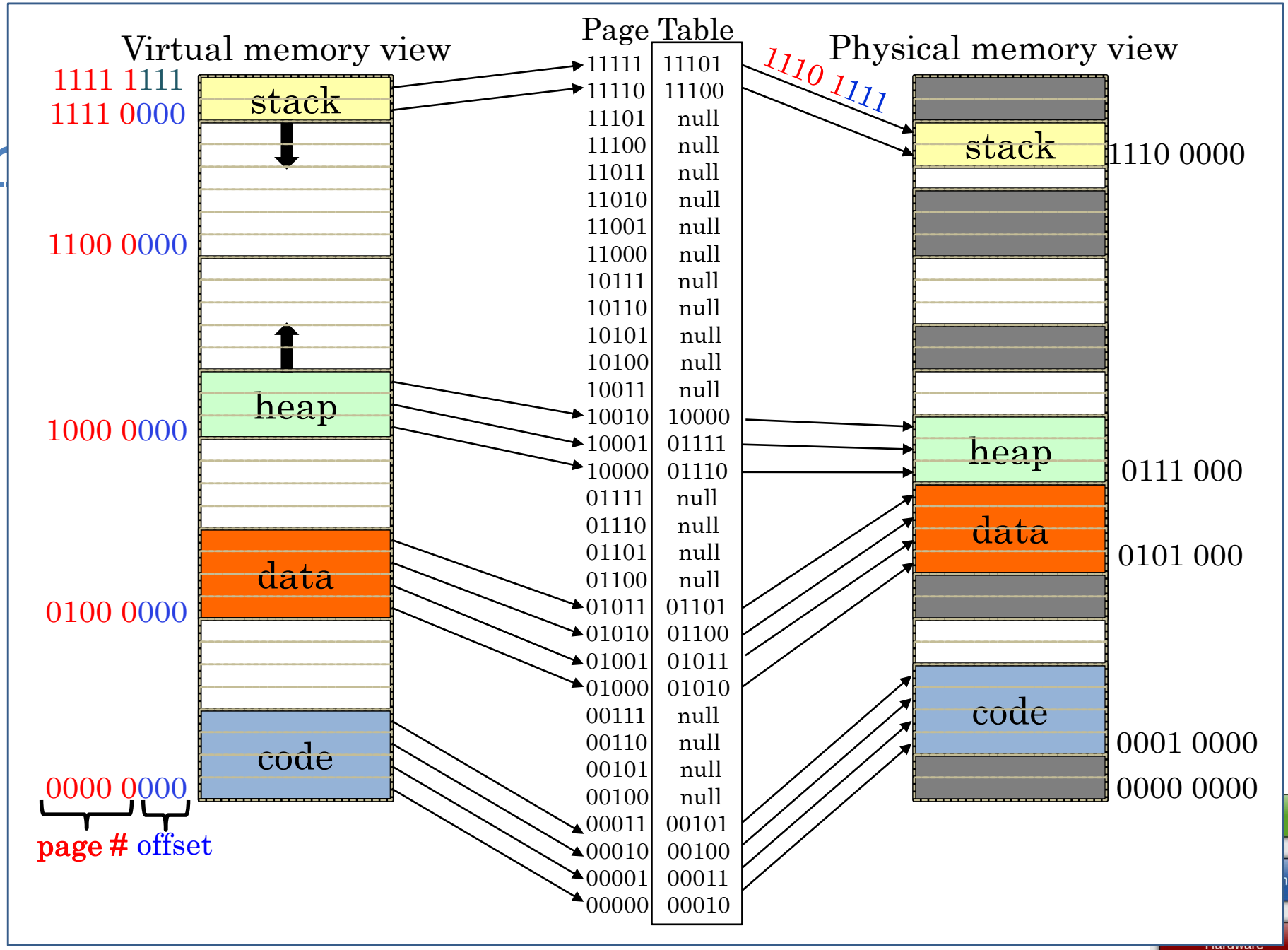


Diagram source: http://static.duartes.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png

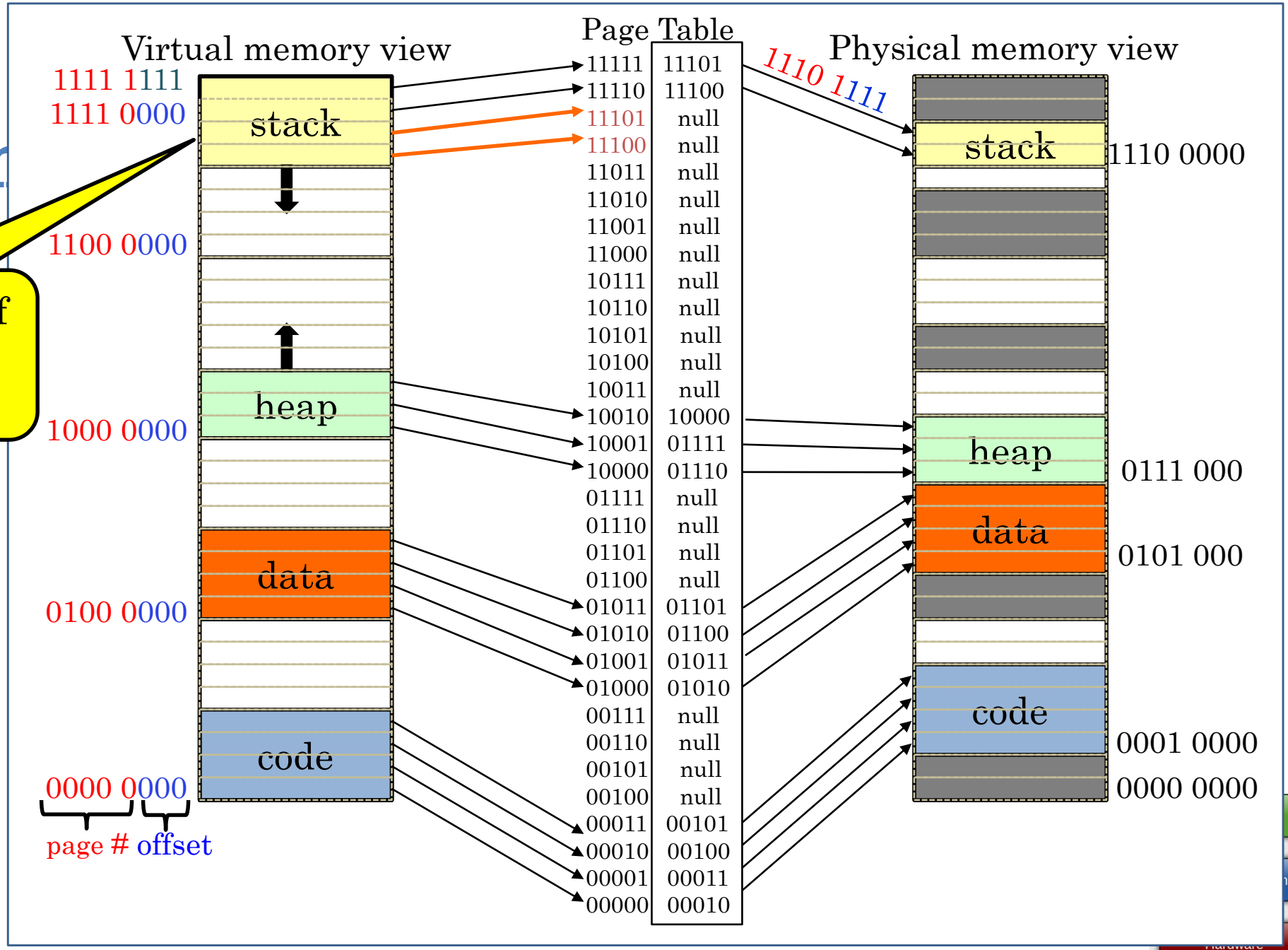


Sum

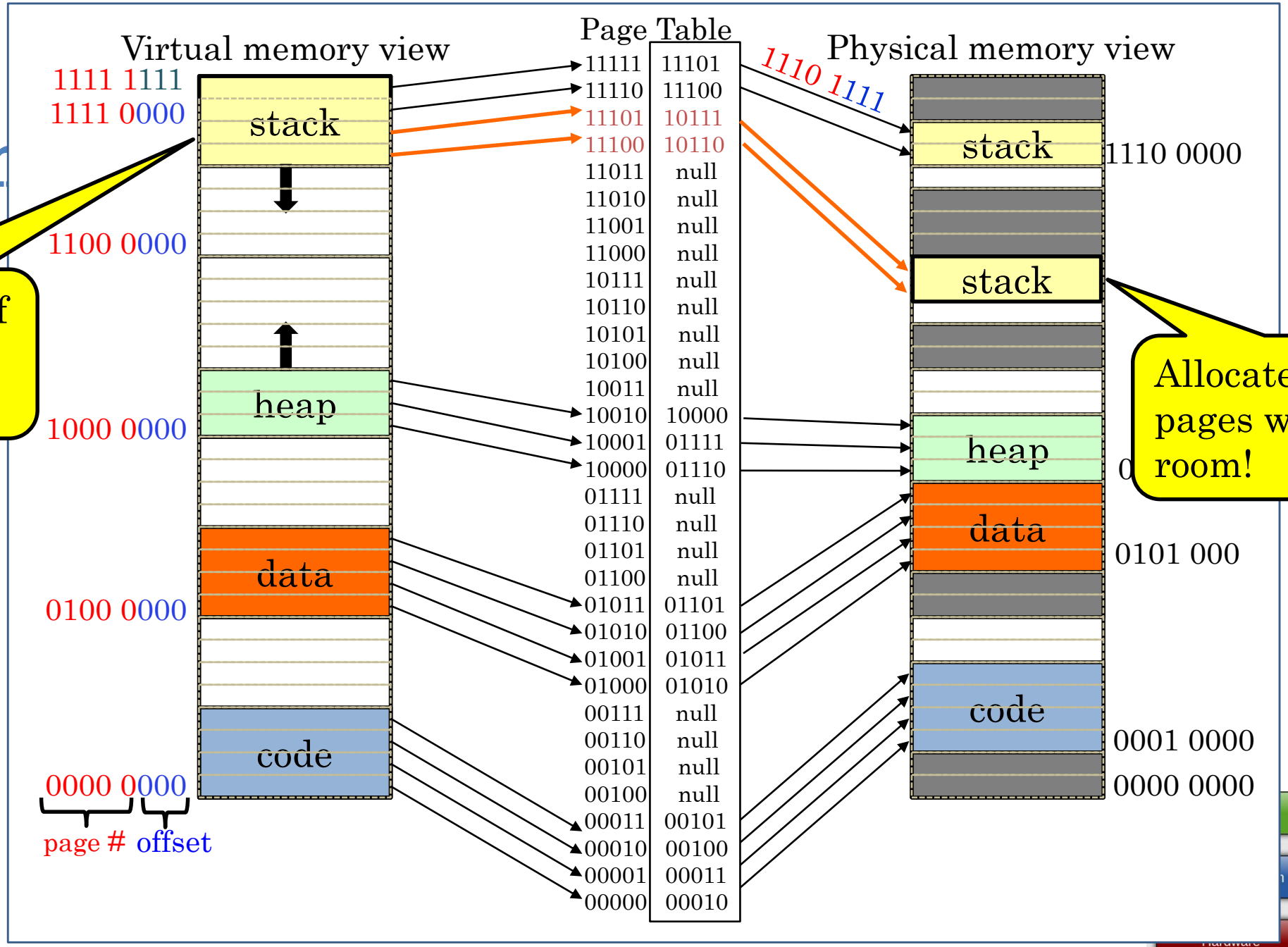

Virtual memory view
Page Table
Physical memory view
1111 1111
1111 0000
1100 0000
1000 0000
0100 0000
0000 0000
stack
heap
data
code
page # offset
11111 11101
11110 11100
11101 null
11100 null
11011 null
11010 null
11001 null
11000 null
10111 null
10110 null
10101 null
10100 null
10011 null
10010 10000
10001 01111
10000 01110
01111 null
01110 null
01101 null
01100 null
01011 01101
01010 01100
01001 01011
01000 01010
00111 null
00110 null
00101 null
00100 null
00011 00101
00010 00100
00001 00011
00000 00010
1110 1111
stack
heap
data
code
1110 0000
0111 000
0101 000
0001 0000
0000 0000

Sum

What happens if stack grows to 1110 0000?



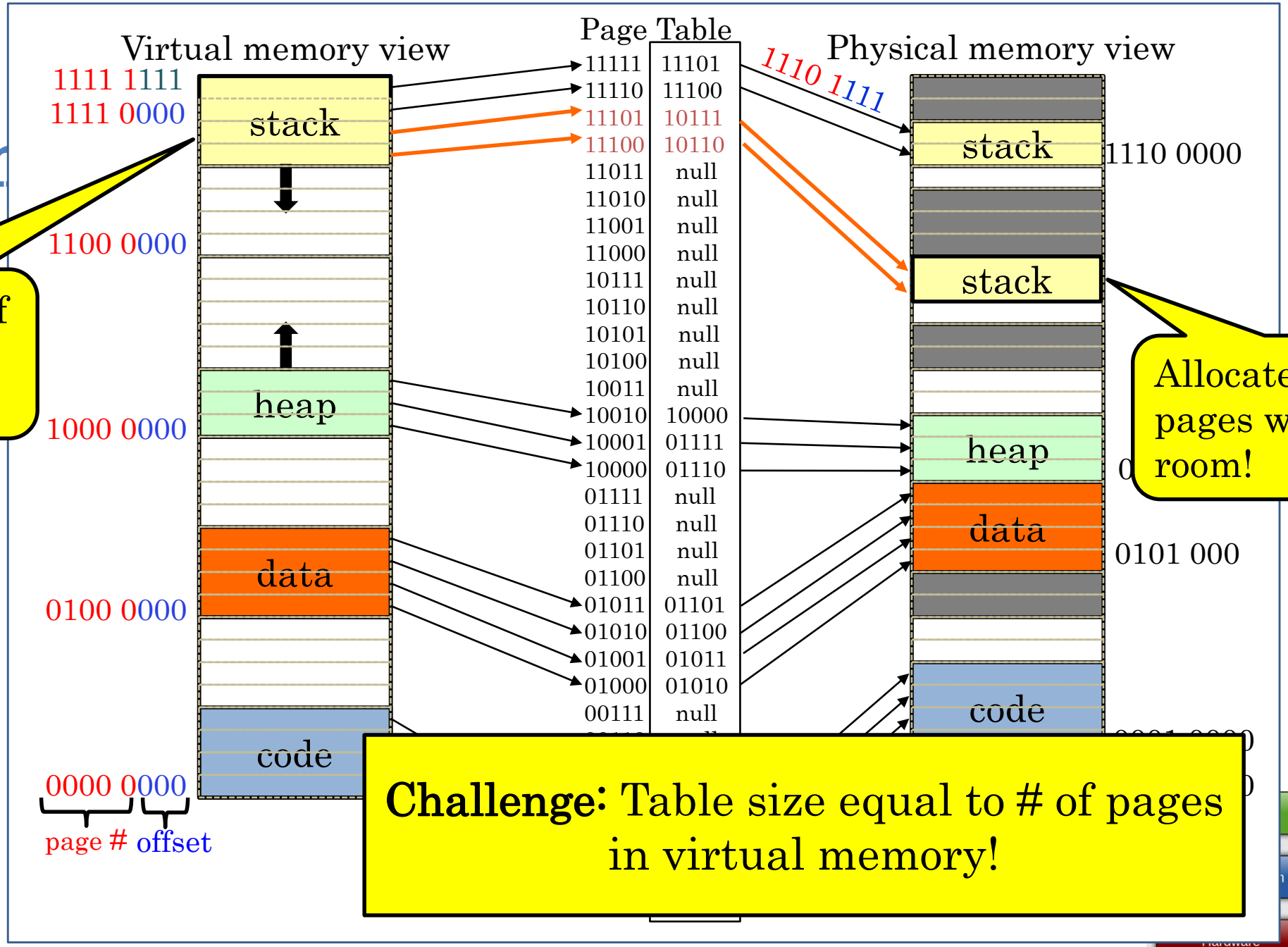
Sum



What happens if stack grows to 1110 0000?

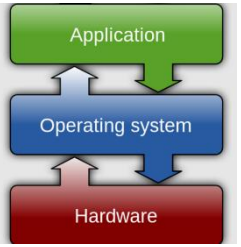
Allocate new pages where room!

Sum


Virtual memory view
Page Table
Physical memory view
1111 1111
1111 0000
1100 0000
1000 0000
0100 0000
0000 0000
page # offset
stack
heap
data
code
11111 11101
11110 11100
11101 10111
11100 10110
11011 null
11010 null
11001 null
11000 null
10111 null
10110 null
10101 null
10100 null
10011 null
10010 10000
10001 01111
10000 01110
01111 null
01110 null
01101 null
01100 null
01011 01101
01010 01100
01001 01011
01000 01010
00111 null
1110 1111
stack
1110 0000
stack
heap
data
0101 000
code
What happens if stack grows to 1110 0000?
Allocate new pages where room!
Challenge: Table size equal to # of pages in virtual memory!

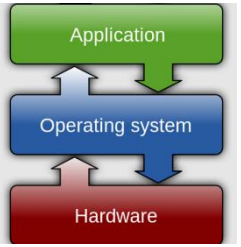
How Big is the Page Table?

- Typical page size: 4 KiB
  - How many bits of the address is that? (remember $2^{10} = 1024$)
  - Ans: 4KiB = $4 \times 2^{10} = 2^{12} \Rightarrow$ 12 bits of the address
- So how big is the simple page table for each process?
  - $2^{32}/2^{12} = 2^{20}$ (that's about a million entries) x 4 bytes each => 4 MiB
  - When 32-bit machines got started (vax 11/780, intel 80386), this was a lot of memory
- How big is a simple page table on a 64-bit processor (x86_64)?
  - $2^{64}/2^{12} = 2^{52}$ (that's 4.5×10^{15} or 4.5 exa-entries)×8 bytes each = 36×10^{15} bytes or 36 exa-bytes!!!! This is a ridiculous amount of memory!
- Mostly, the address space is sparse, i.e. has holes in it that are not mapped to physical memory
  - So, most of this space is taken up by page tables mapped to nothing



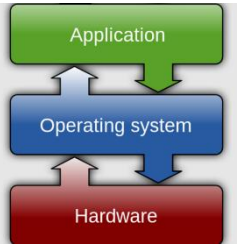
Page Table Discussion

- What provides protection here?
  - Translation (per process) and dual-mode operation!
  - Can't let process alter its own page table!
- 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 4K pages, need 1 million page table entries (256 pages for the page table)!
  - 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
- Simple Page table is way too big!
  - Does it all need to be in memory?
  - How about multi-level paging?
  - or combining paging and segmentation

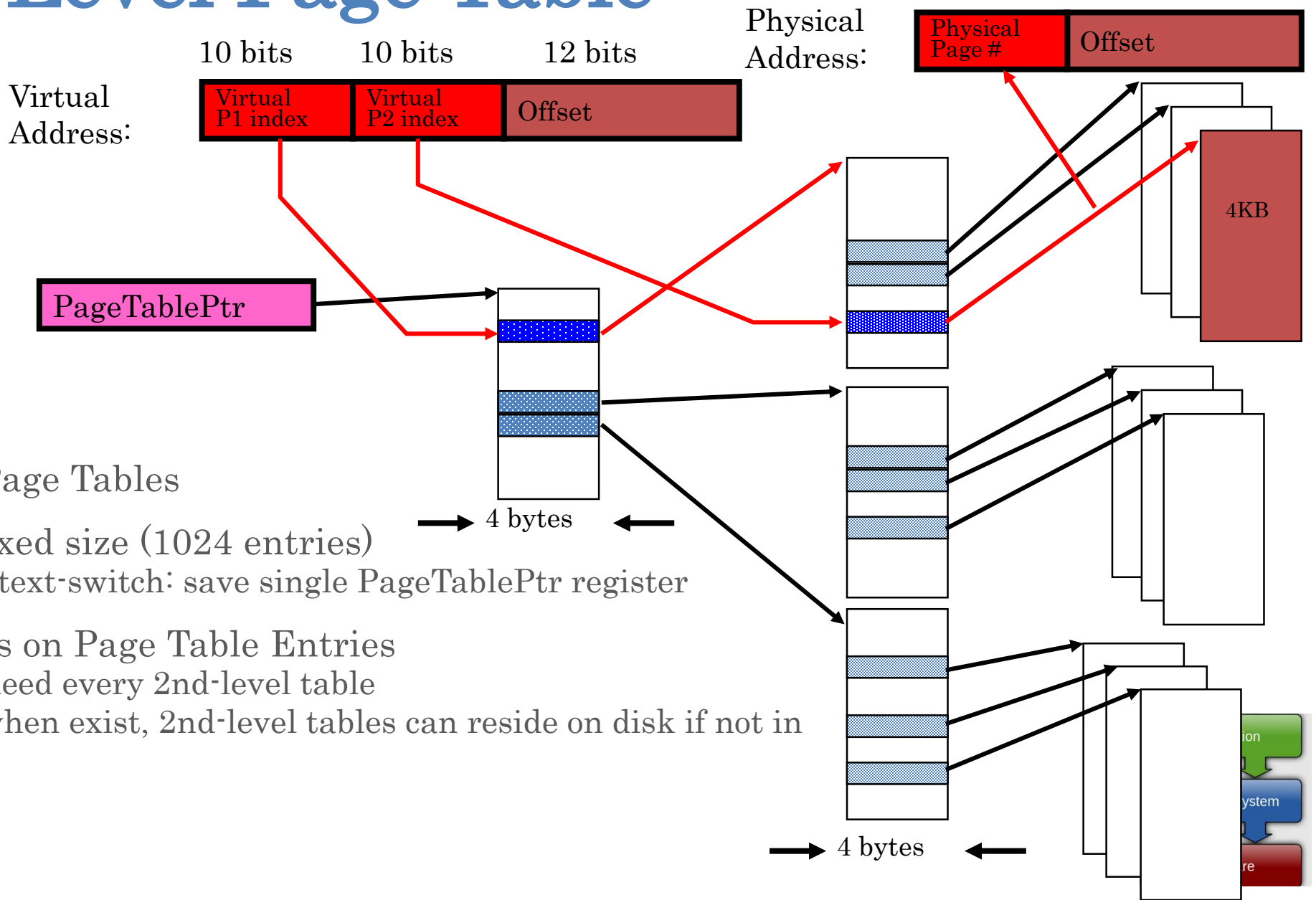


How to Structure a Page Table

- Page Table is a map from VPN to PPN
- Simple page table corresponds to a sparse array
  - VPN is index into table, each entry contains PPN
- What other map structures can you think of?
  - Trees?
  - Hash Tables?

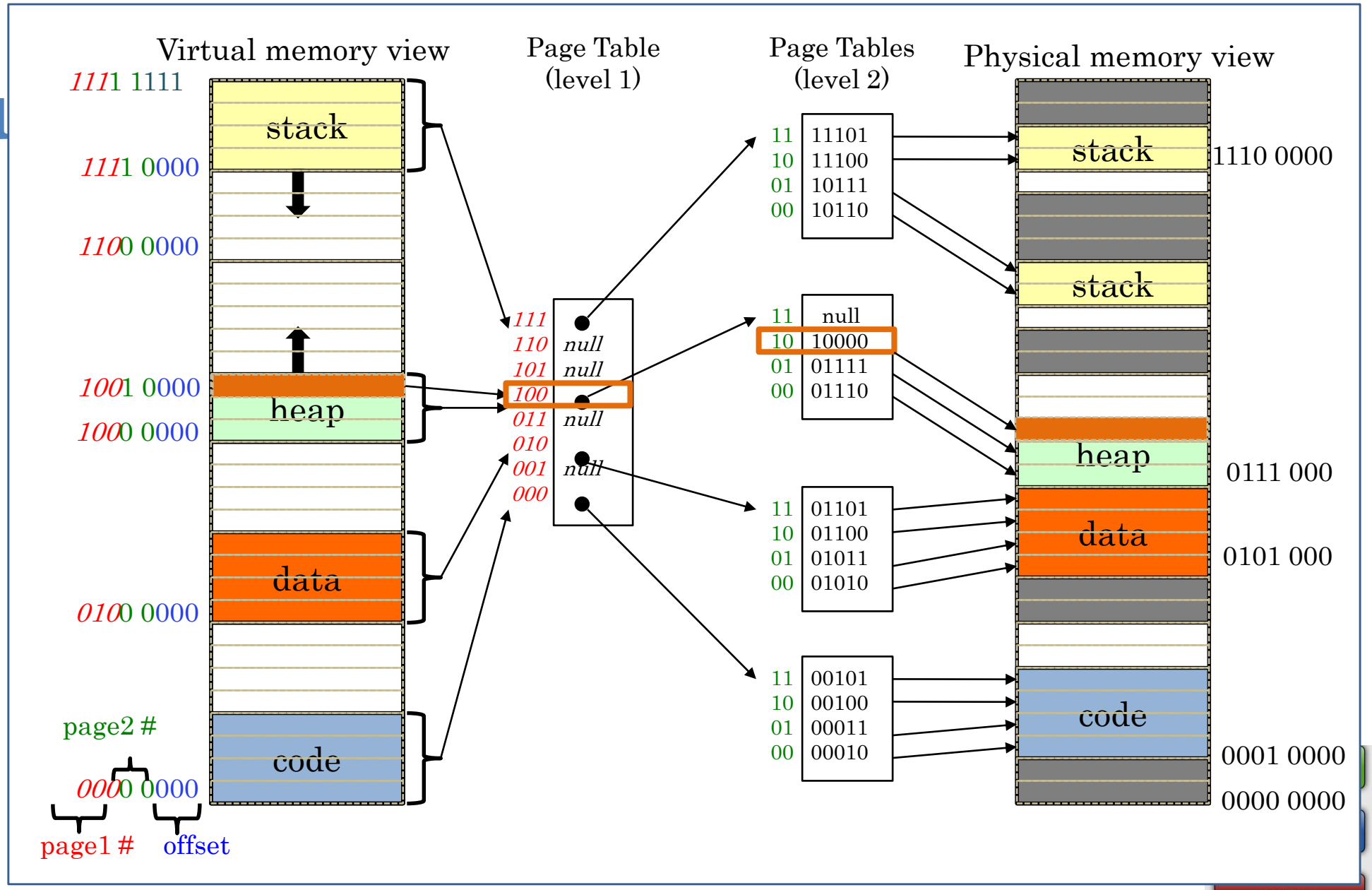


Two-Level Page Table



- 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

Su


Virtual memory view
Page Table (level 1)
Page Tables (level 2)
Physical memory view
1111 1111
1111 0000
1100 0000
1001 0000
1000 0000
0100 0000
0000 0000
page2 #
page1 #
offset
stack
heap
data
code
111
110 null
101 null
100
011 null
010
001 null
000
11 11101
10 11100
01 10111
00 10110
11 null
10 10000
01 01111
00 01110
11 01101
10 01100
01 01011
00 01010
11 00101
10 00100
01 00011
00 00010
stack
stack
heap
data
code
1110 0000
0111 000
0101 000
0001 0000
0000 0000

x86 Classic 32-bit Address Translation

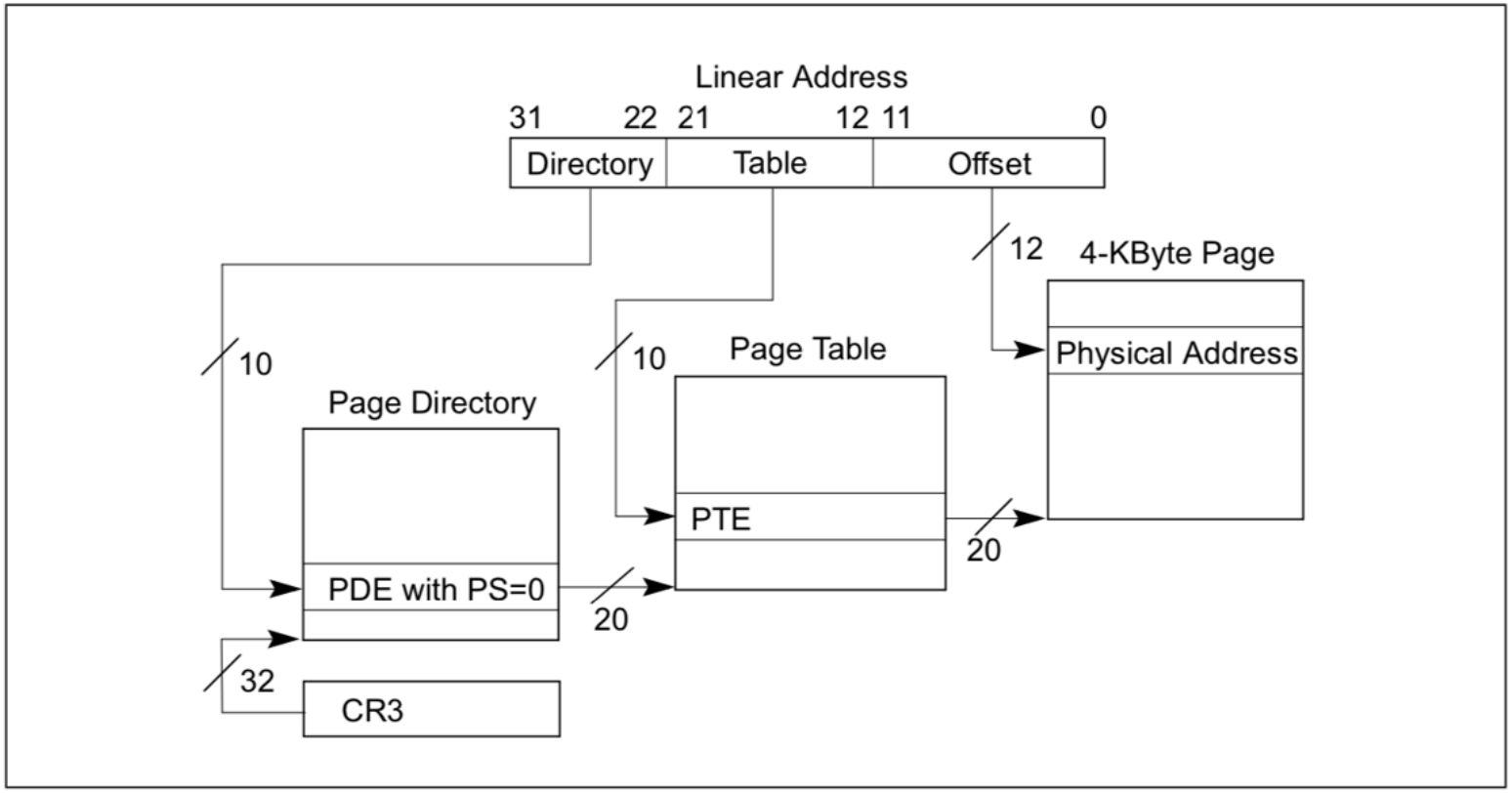
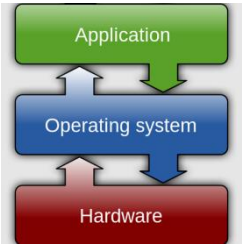
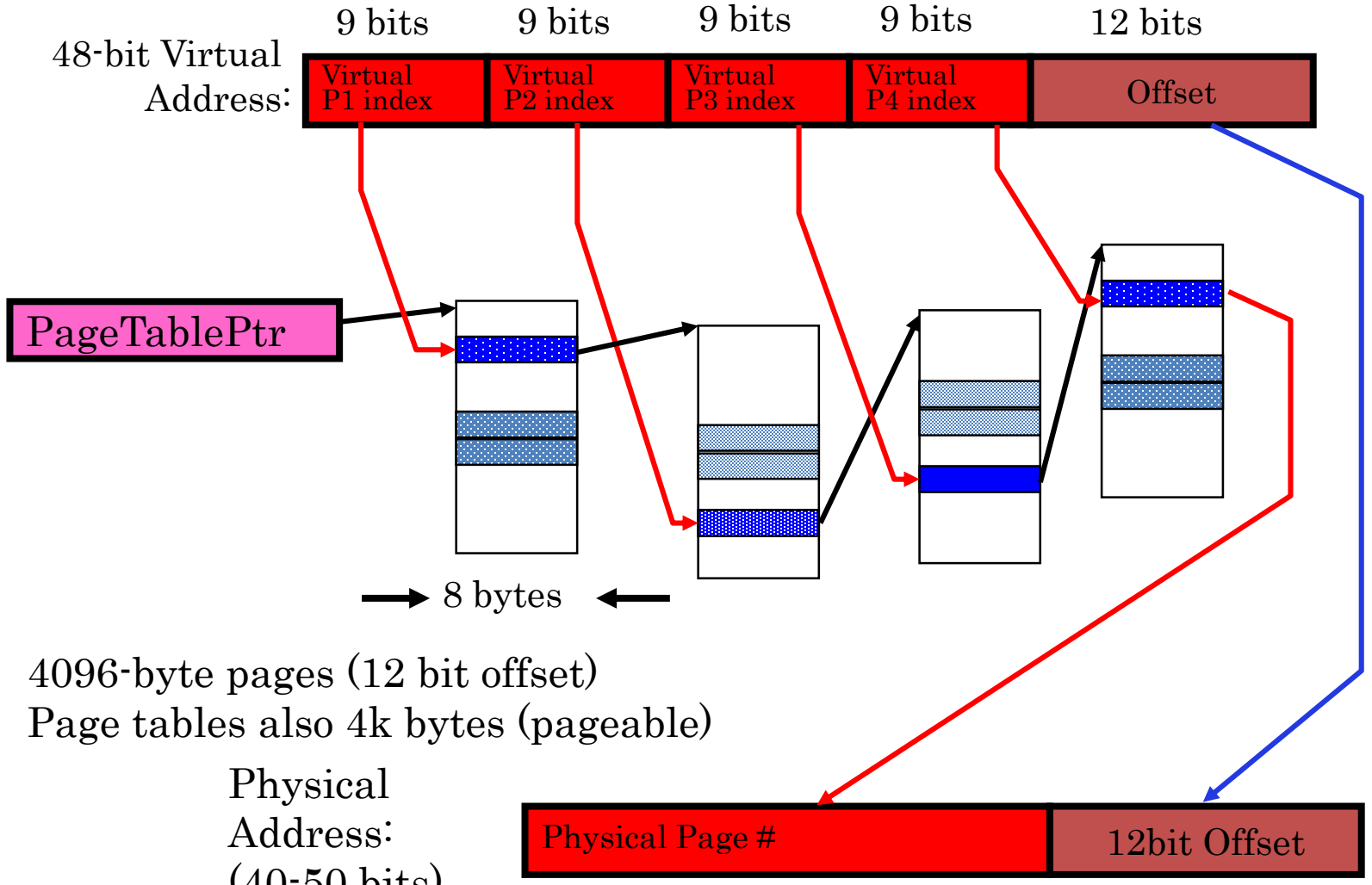


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

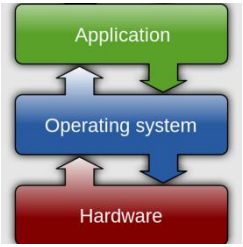


x86-64: Four-Level Page Table!



4096-byte pages (12 bit offset)
Page tables also 4k bytes (pageable)

Physical Address: (40-50 bits)



Large 64-bit Address Space

- All current x86-64 processors support 64-bit operations
- 64-bit words but 48-bit addresses

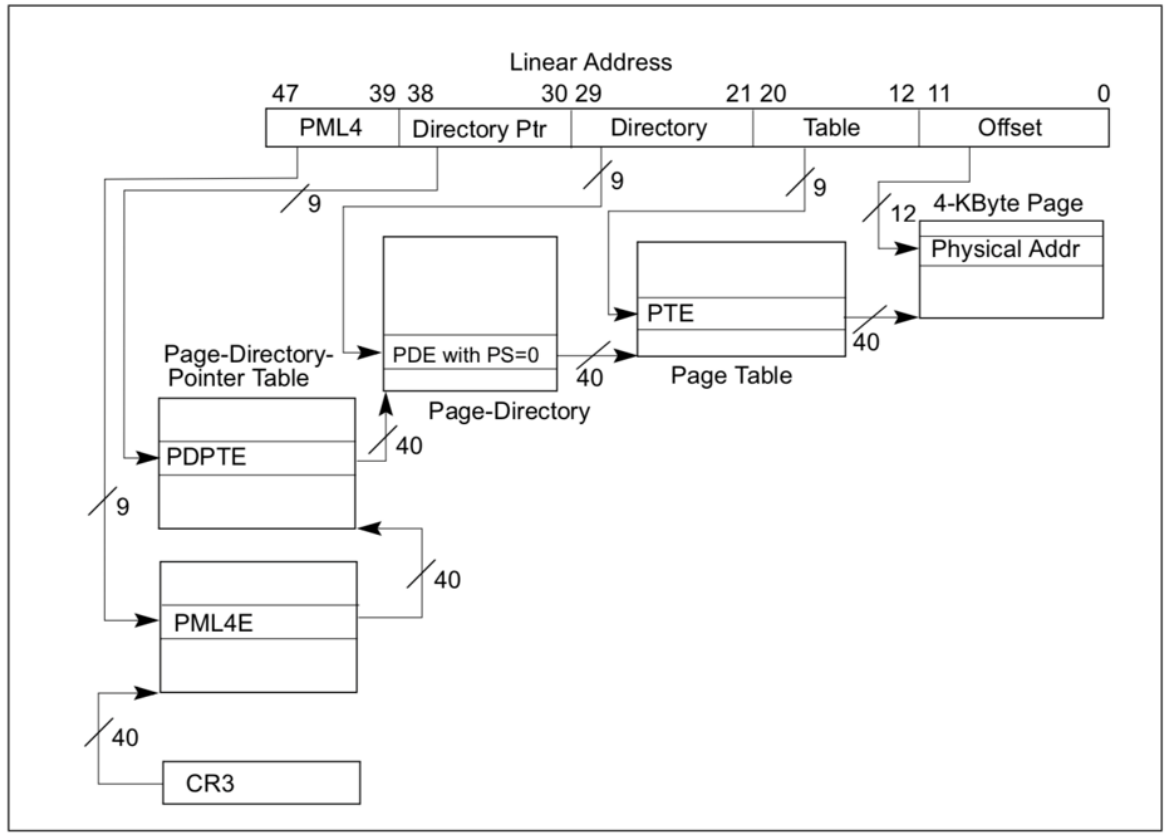
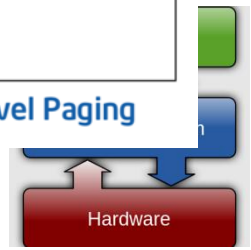


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging



"Huge Pages" Supported as Well

- Memory is now cheap…

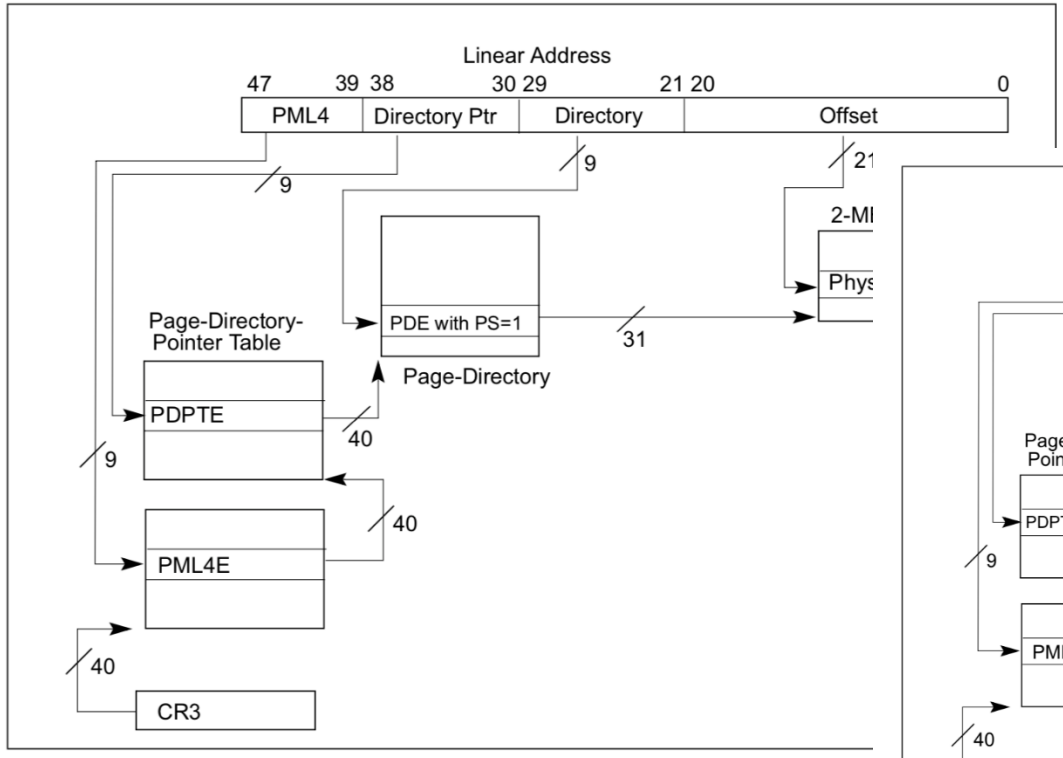


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-l

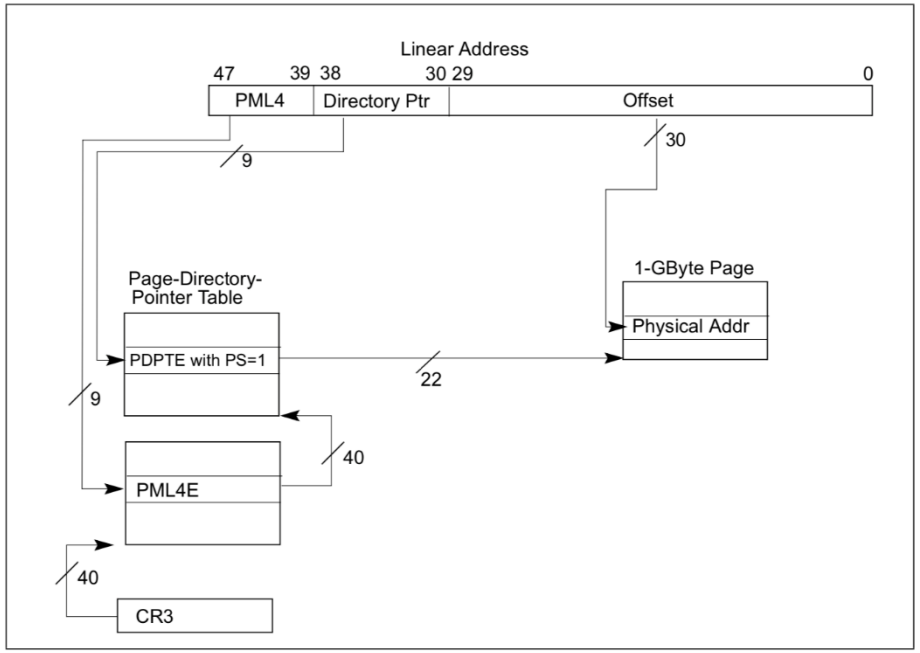
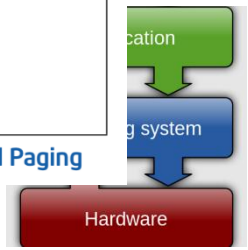


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging



Intel Ice Lake (2019): One More Layer

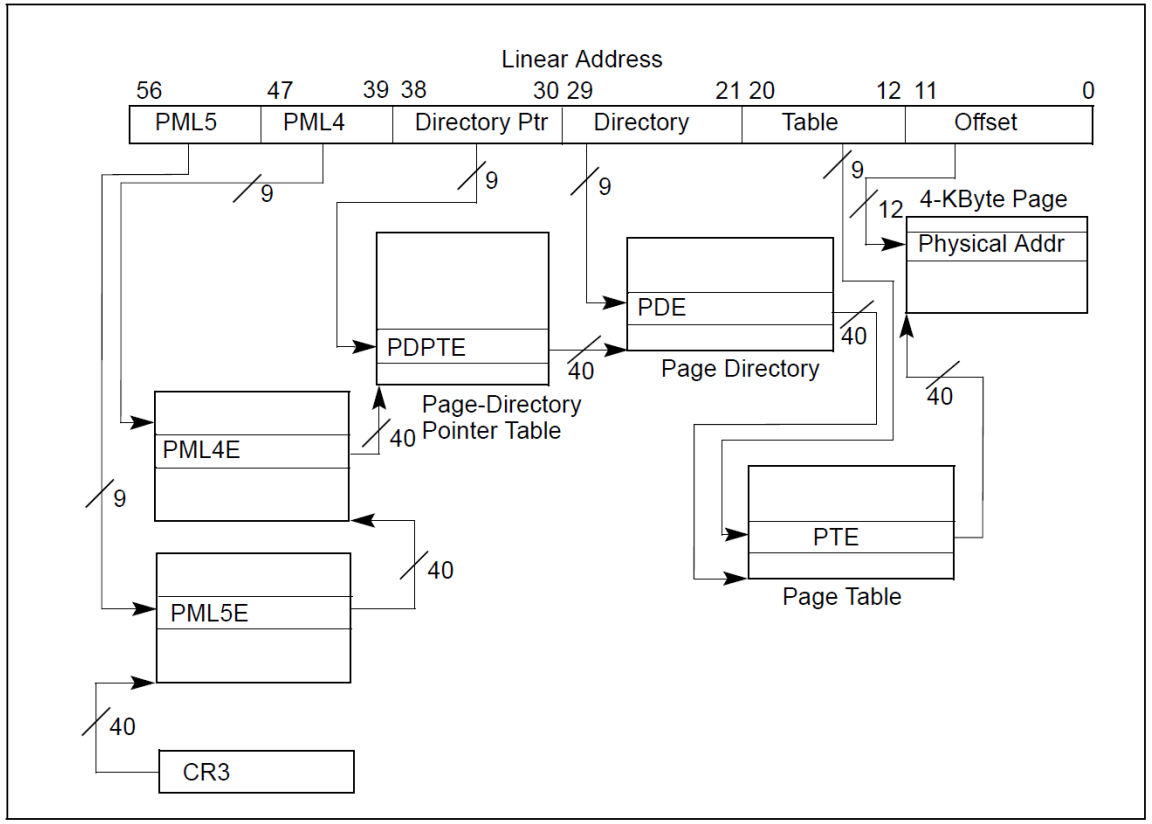
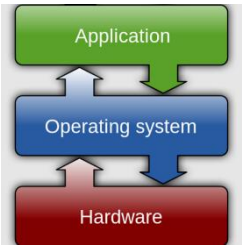
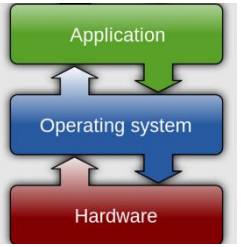


Figure 2-1. Linear-Address Translation Using 5-Level Paging

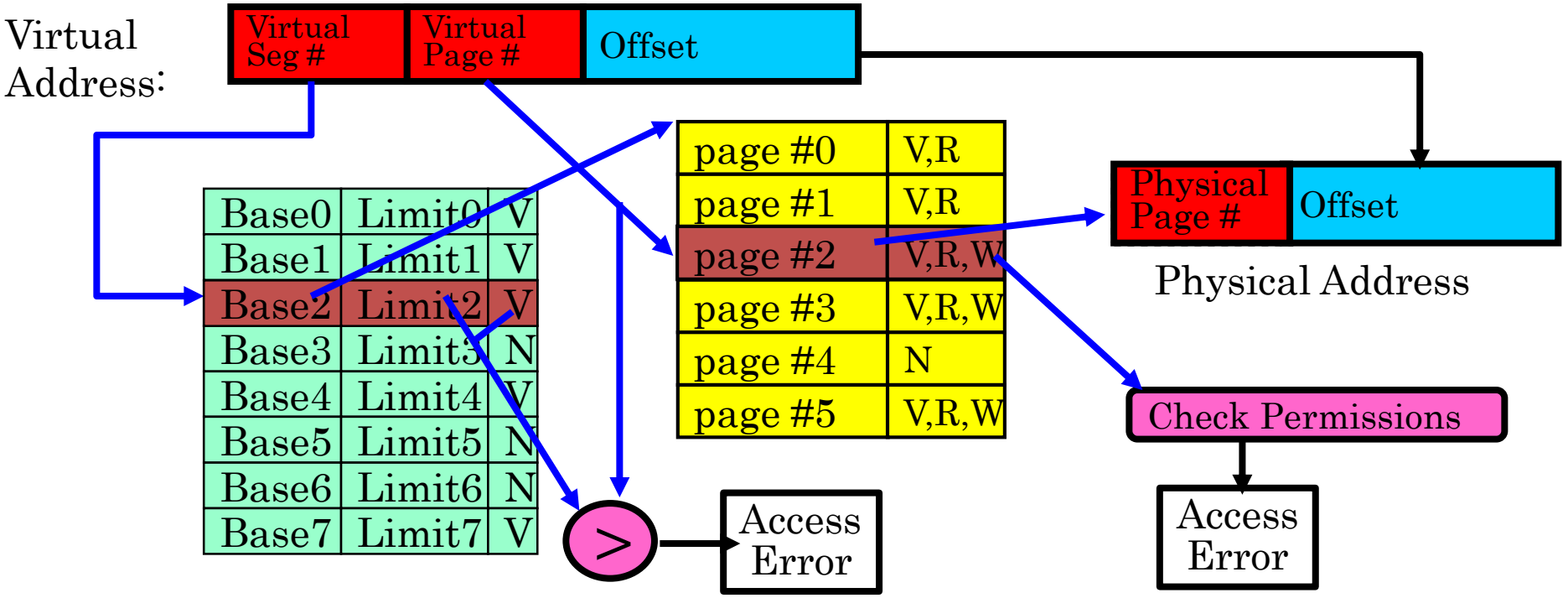


Multi-Level Translation Analysis

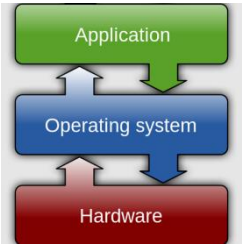
- Pros:
  - Only need to allocate as many page table entries as we need for application
    - In other words, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - Share at segment or page level (need additional reference counting)
- Cons:
  - One pointer per page (typically 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!



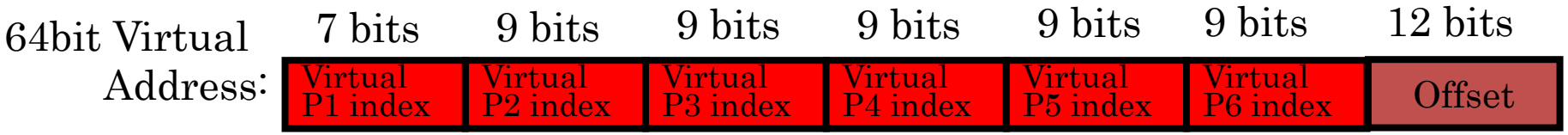
Aside: Segments + Pages



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

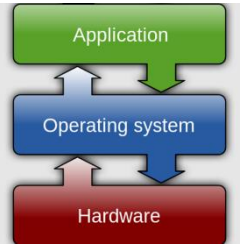


IA-64: 64-bit Address: Six Levels???



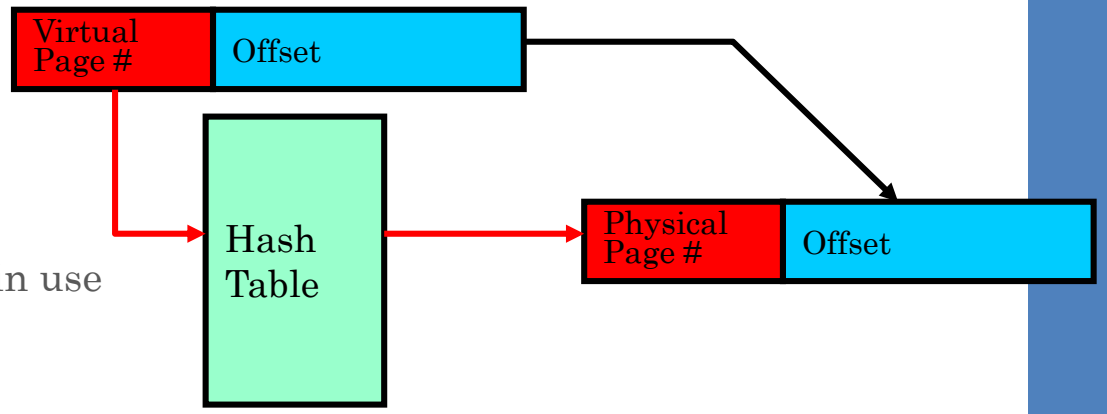
Too slow

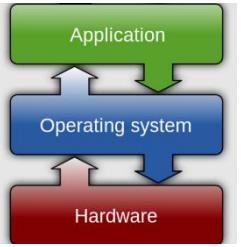
Too many almost-empty tables



Alternative: Inverted Page Table

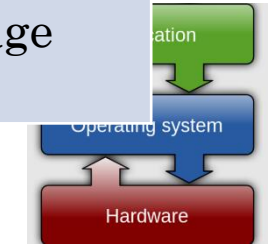
- With all previous examples ("Forward Page Tables")
  - Size of page table is at least proportional to amount of virtual memory allocated to processes
  - Physical memory may be much less
    - Much of process space may be out on disk or not in use
- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
    - PowerPC, UltraSPARC, IA64
- Cons:
  - Complexity of managing hash chains: Often in hardware!
  - Poor cache locality of page table





Address Translation Comparison

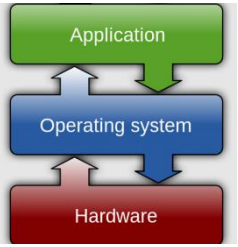
	Advantages	Disadvantages
Simple Segmentation	Fast context switching (segment map maintained by CPU)	External fragmentation
Paging (Single-Level)	No external fragmentation Fast and easy allocation	Large table size (~ virtual memory) Internal fragmentation
Paged Segmentation	Table size ~ # of pages in virtual memory Fast and easy allocation	Multiple memory references per page access
Multi-Level Paging		
Inverted Page Table	Table size ~ # of pages in physical memory	Hash function more complex No cache locality of page table



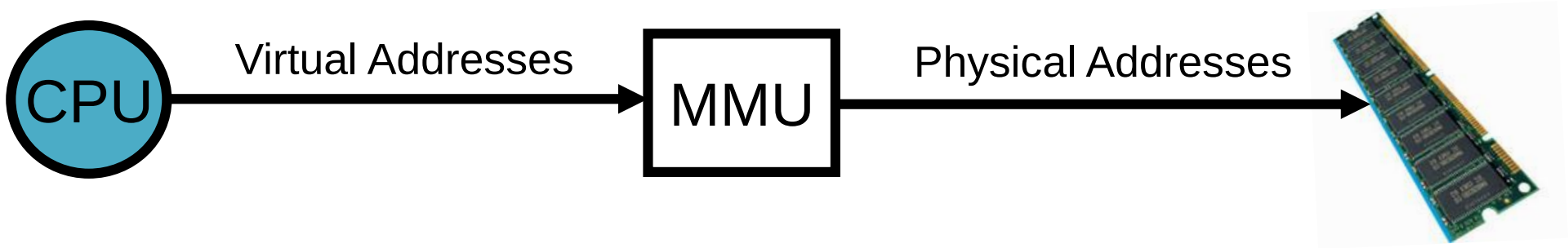
Announcements

- Assignment 2 (extended) deadline today
- Project 1 (extended) deadline Monday, April 21
  - Questionnaire for project 1 will be posted on Moodle soon
- Assignment 3 deadline Friday May 2

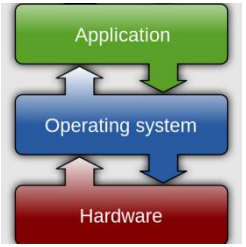
- No lectures next week (April 21 and April 23)



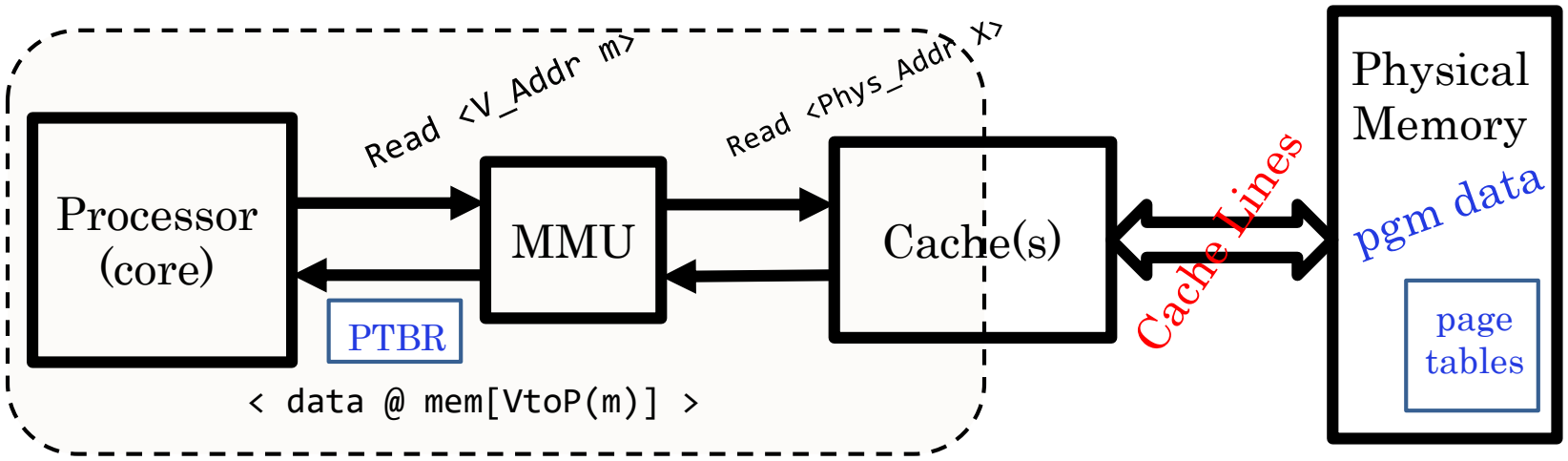
How to Translate Addresses Fast Enough?



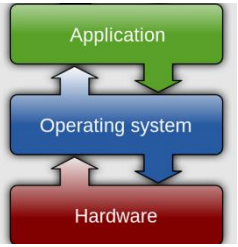
- The MMU must translate virtual address to physical address on:
  - Every instruction fetch
  - Every load
  - Every store
- More than one translation for EVERY instruction
  - Each one requires a page table tree traversal (!)
  - How to simplify this???



Where and What is the MMU?

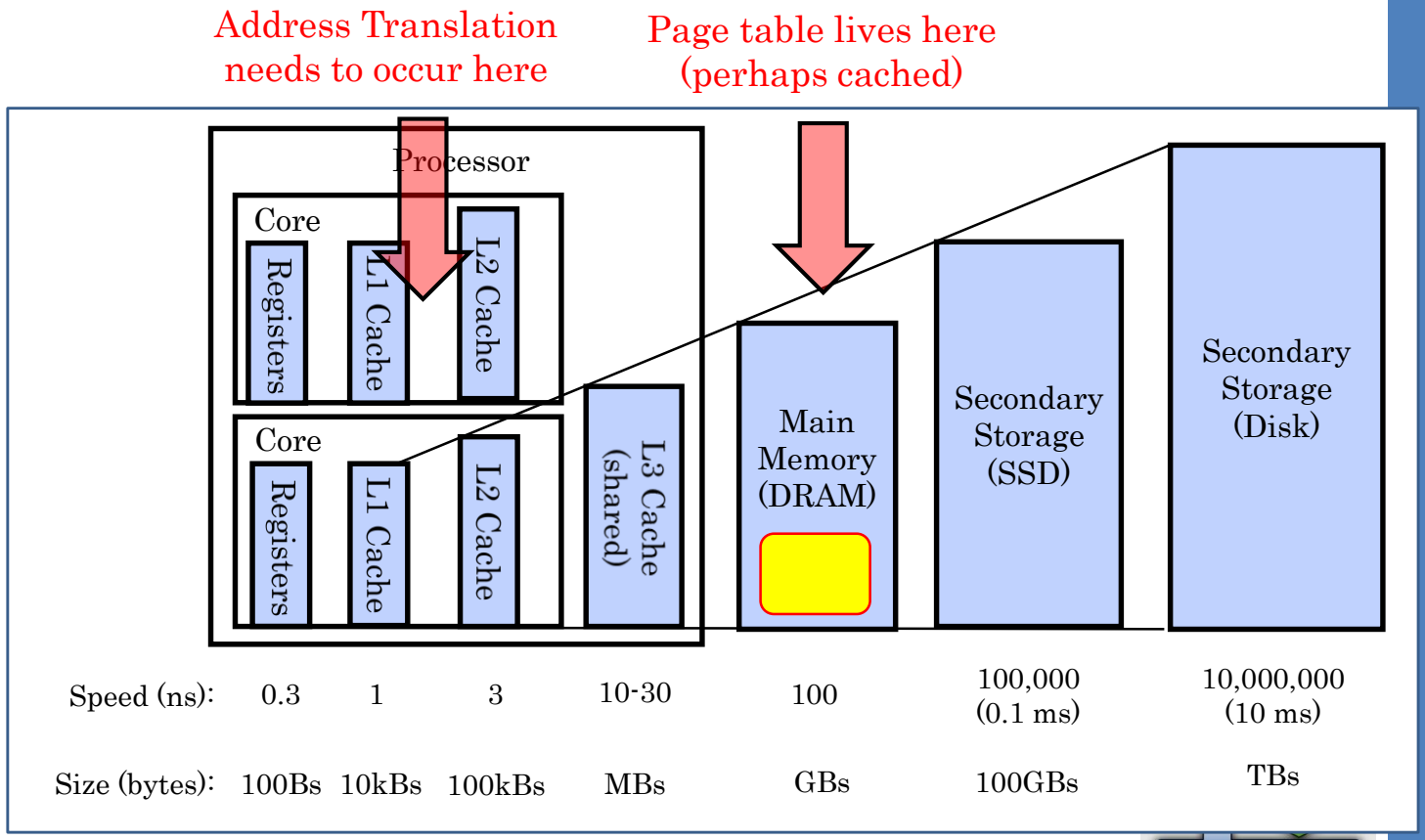


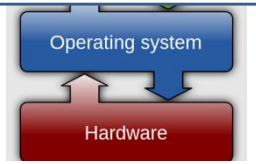
- On every memory reference (I-fetch, Load, Store), MMU reads (multiple levels of) page table entries to get physical frame or FAULT
  - Through the caches to the memory
  - Then read/write the physical location



Recall: Memory Hierarchy

- Large memories are slow
- Small memories are fast

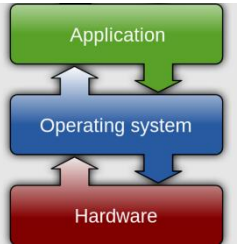




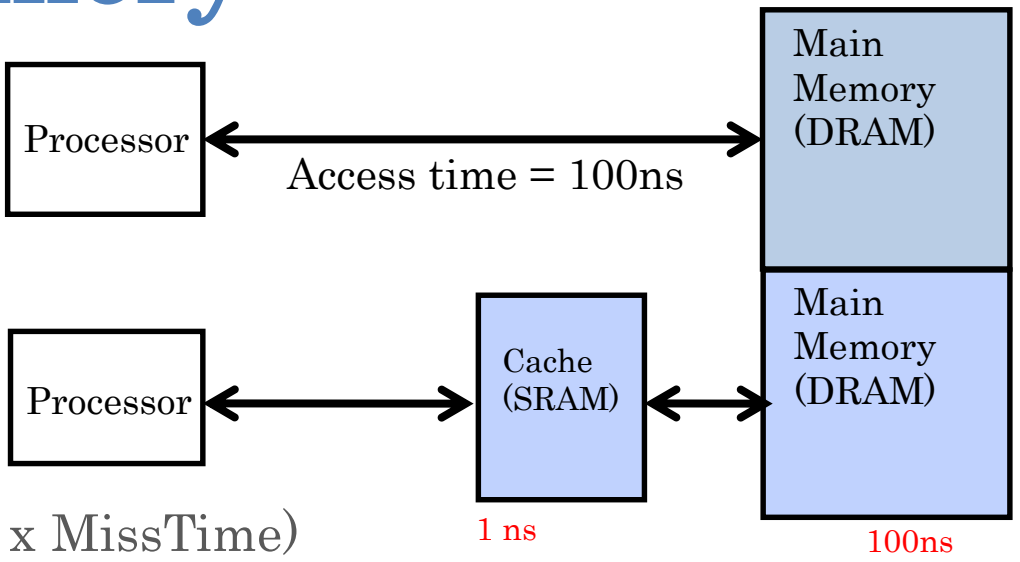
Recall: Caches

- Cache: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many techniques used to make computers fast
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc…
- Key measure:

Average Access time =
(Hit Rate x Hit Time) + (Miss Rate x Miss Time)



Recall: Caching Memory

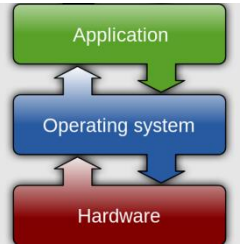


Average Memory Access Time (AMAT)

= (Hit Rate x HitTime) + (Miss Rate x MissTime)

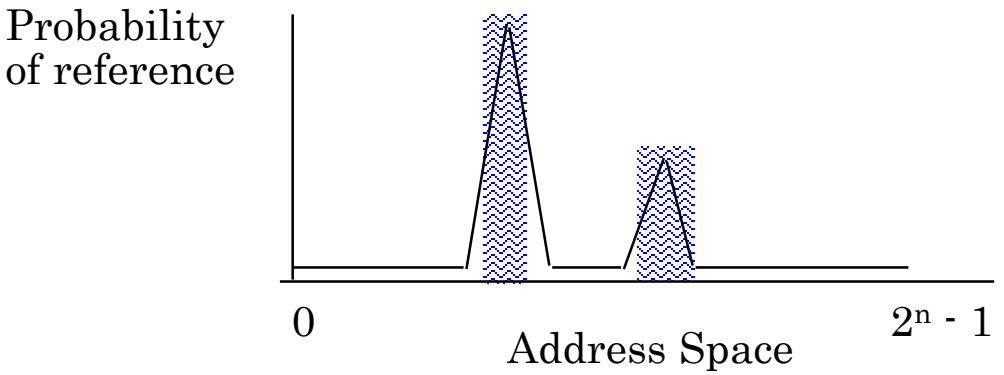
Where: HitRate + MissRate = 1

- HitRate = 90% => AMAT = (0.9 x 1) + (0.1 x (100+1)) = 11 ns
- HitRate = 99% => AMAT = (0.99 x 1) + (0.01 x (100+1)) = 2 ns
- MissTime_{L1} includes HitTime_{L1}+MissPenalty_{L1} ≡ HitTime_{L1} +AMAT_{L2}

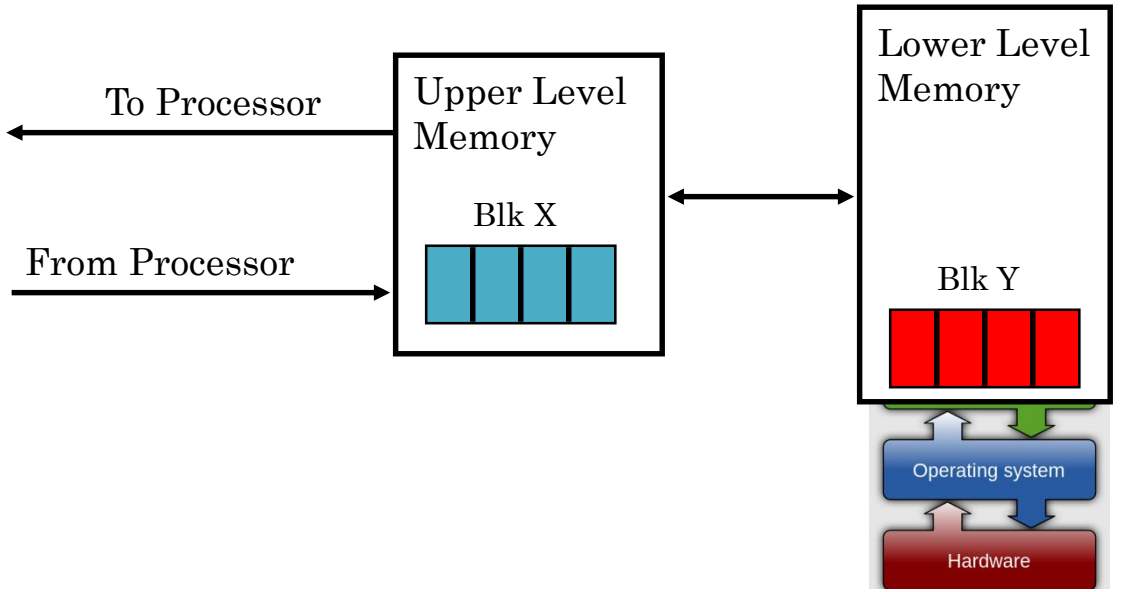


Why Does Caching Help? Locality!

- Temporal Locality (Locality in Time)
  - Keep recently accessed data items closer to processor

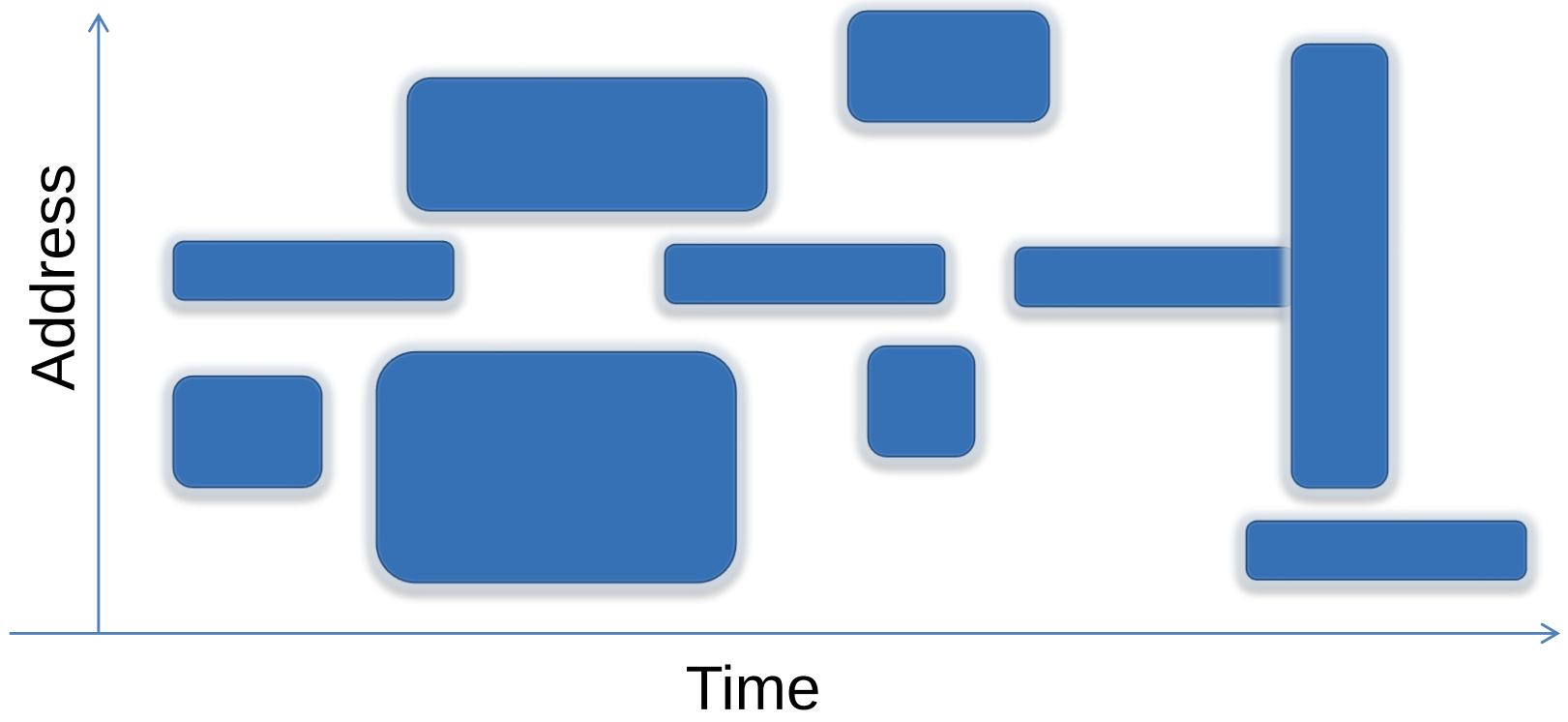


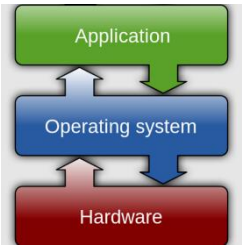
- Spatial Locality (Locality in Space)
  - Move contiguous blocks to the upper levels



Recall: Working Set Model

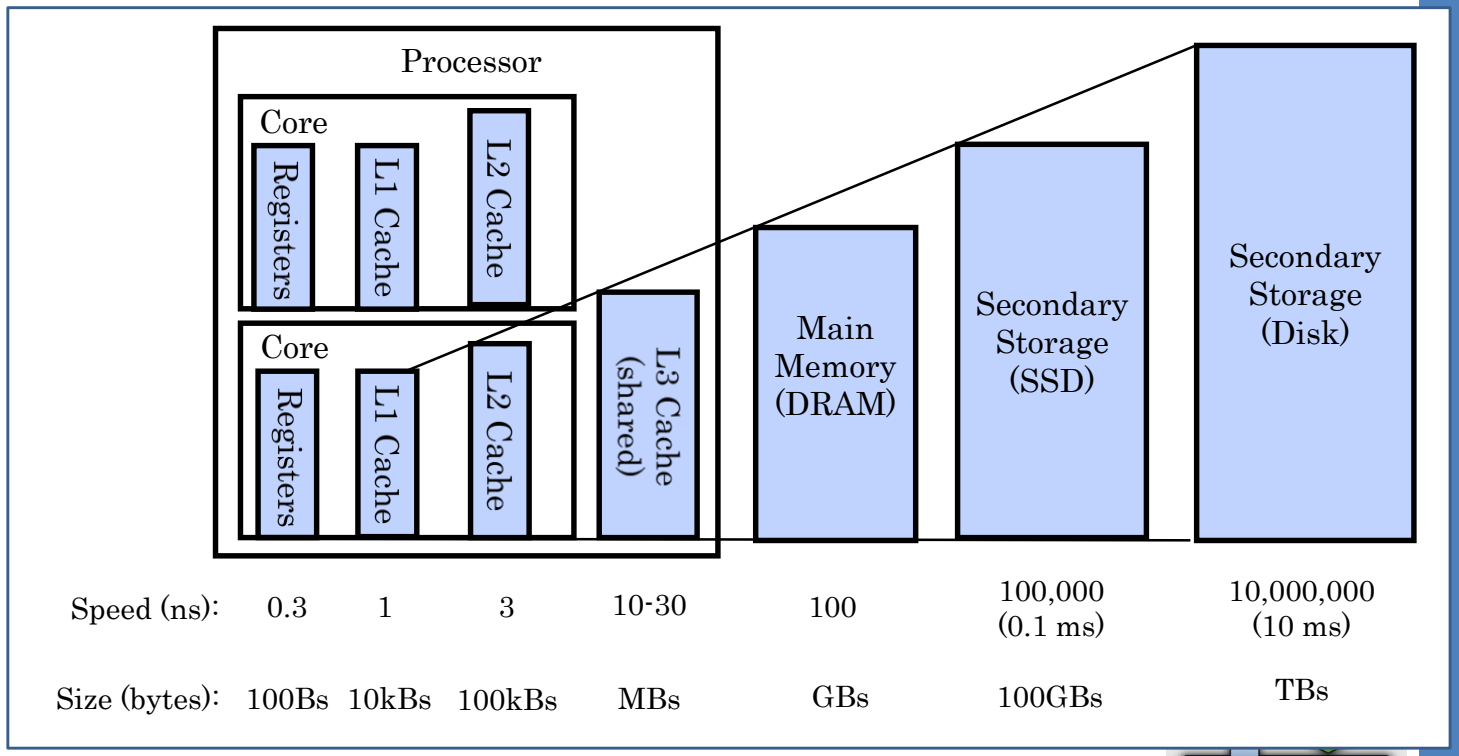
- As a program executes it transitions through a sequence of "working sets" consisting of varying sized subsets of the address space

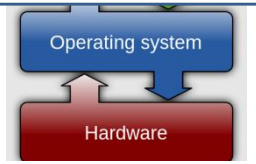




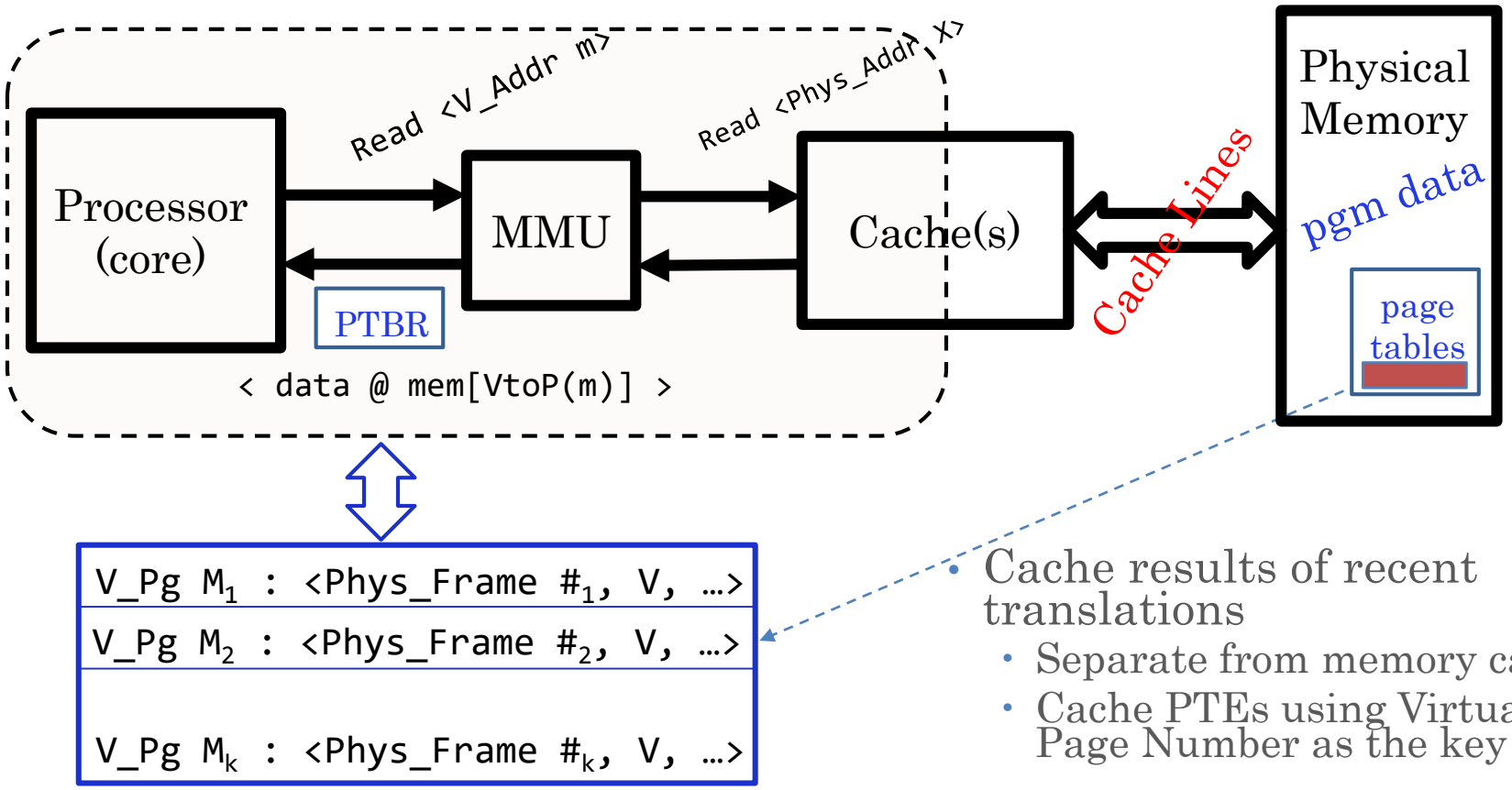
Recall: Memory Hierarchy

- Take advantage of the principle of locality to:
  - Present as much memory in the cheapest technology
  - Provide access at speed offered by the fastest technology

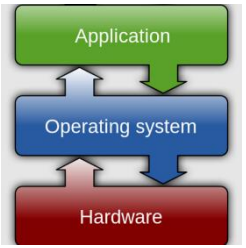




Making Address Translation Fast

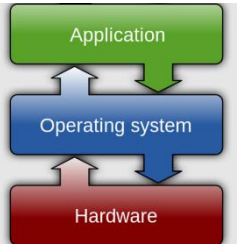


- Cache results of recent translations
  - Separate from memory cache
  - Cache PTEs using Virtual Page Number as the key

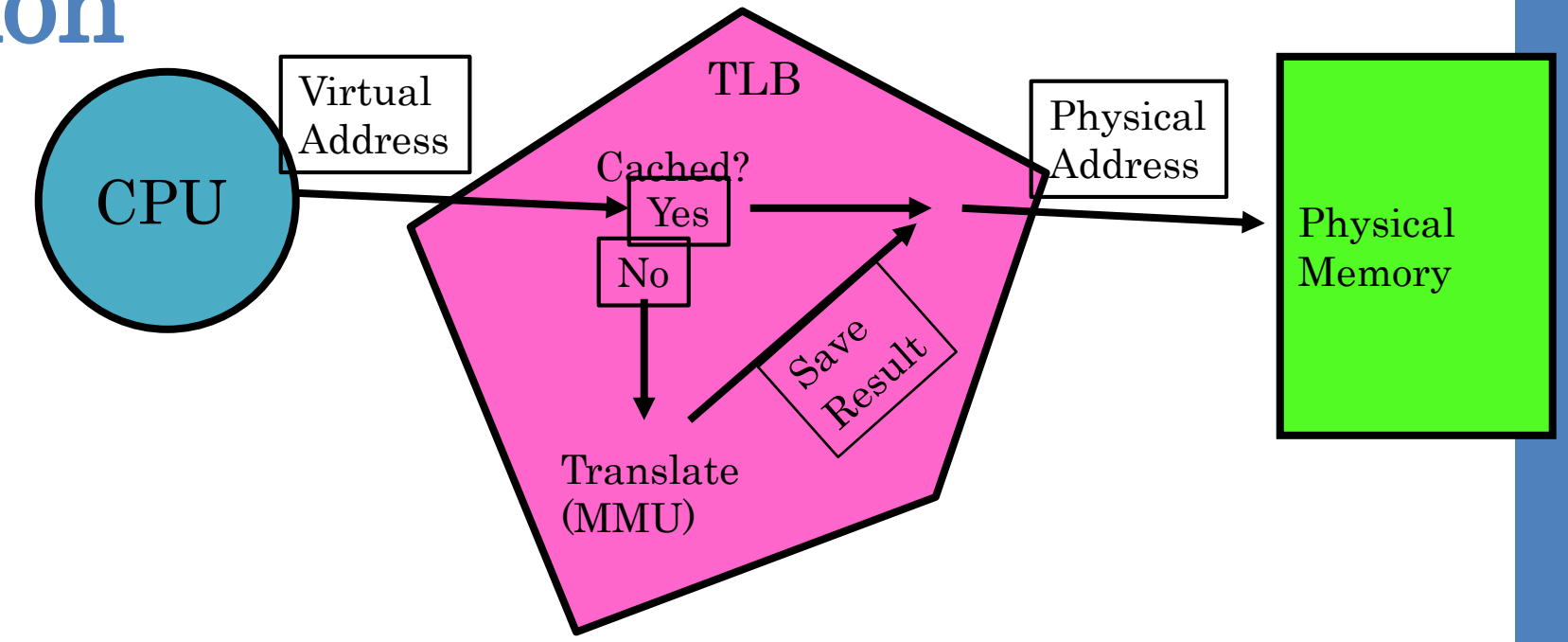


Translation Lookaside Buffer (TLB)

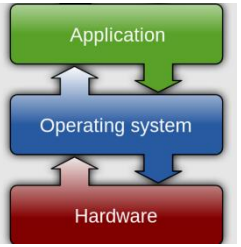
- Record recent Virtual Page # to Physical Frame # translations
- If present in the TLB, translate address without reading page table
  - Caches the end-to-end result, even if the translation involved multiple levels
  - Was invented by Sir Maurice Wilkes – prior to caches
  - People realized "if it's good for page tables, why not the rest of the data in memory?"
- On a TLB miss, the page tables may be cached, so only go to memory when both miss



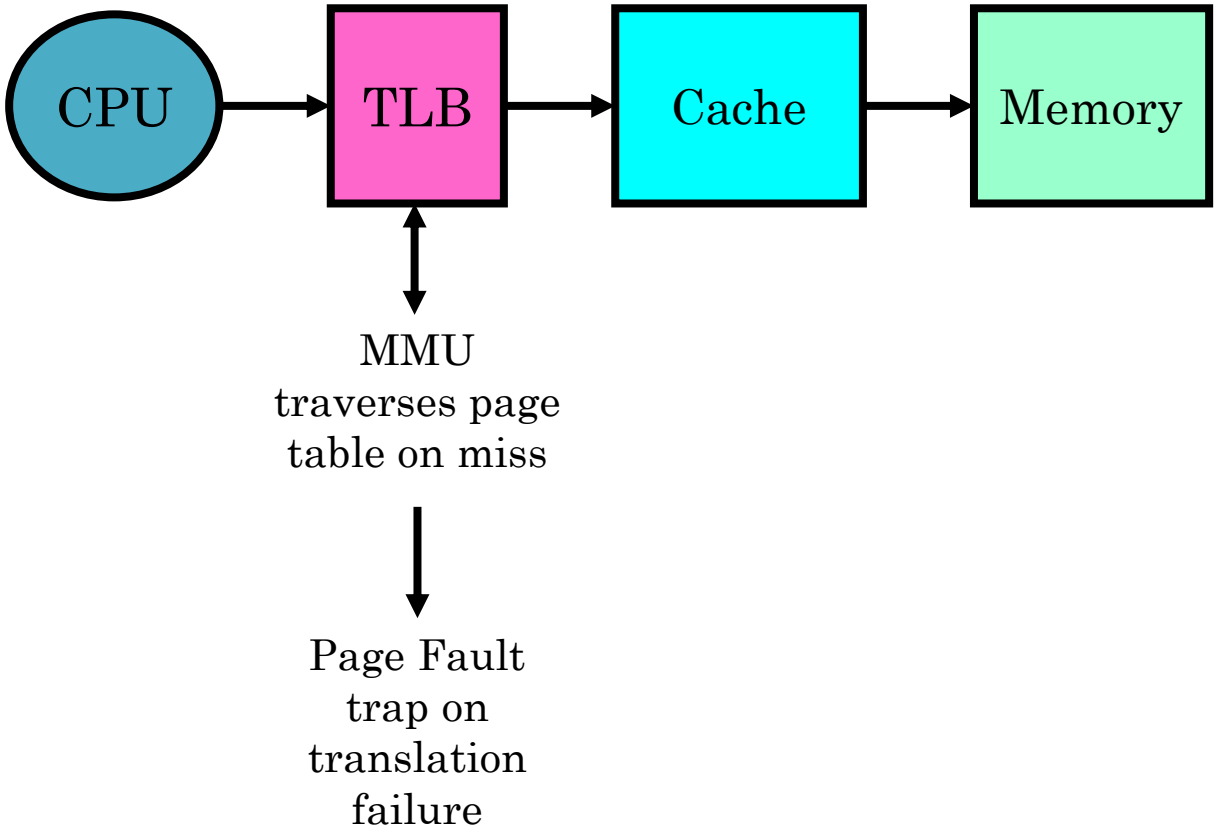
Caching Applied to Address Translation

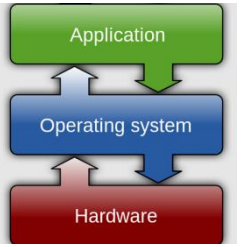


- This relies on locality of PTE accesses
  - Instruction accesses spend a lot of time on the same page
  - Stack accesses have locality
  - Data accesses???



The Big Picture

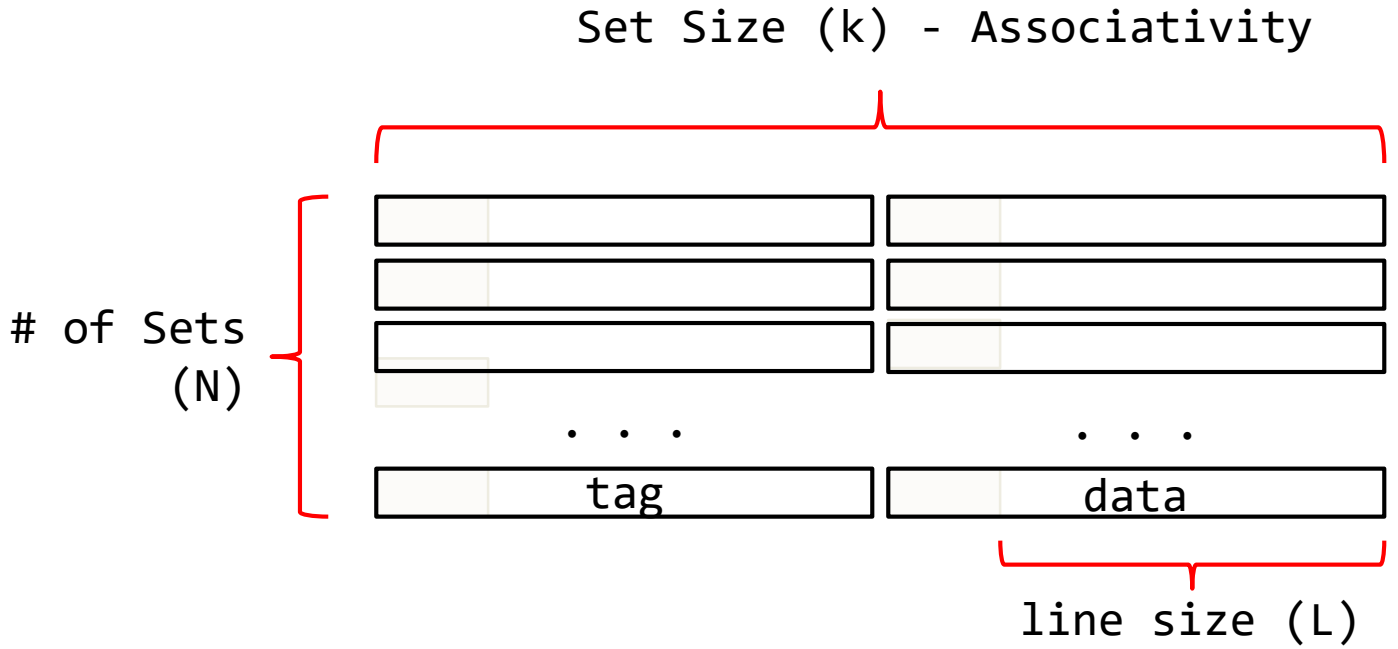




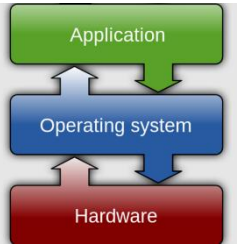
How might organization of a TLB differ from that of a conventional instruction or data cache?

Let's do some review…

What Kind of Cache for TLB?

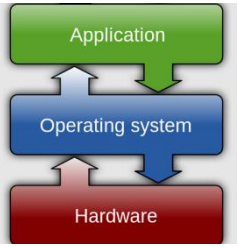


- Remember all those cache design parameters and trade-offs?
- Amount of Data = N * L * K
- Write Policy (write-thru, write-back), Eviction Policy (LRU, …)

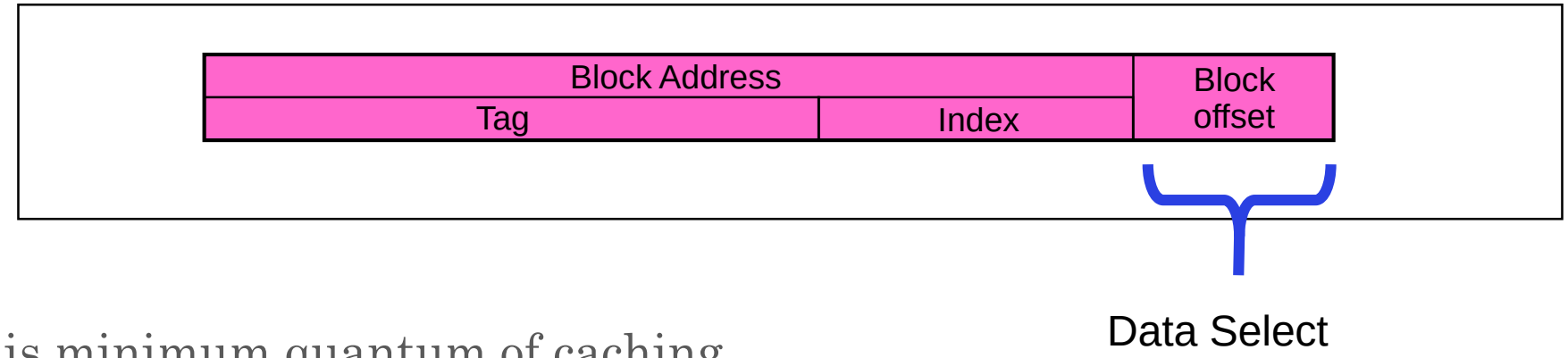


Sources of Cache Misses

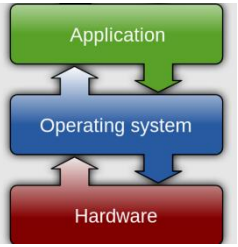
- Compulsory (cold start or first reference): first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: If you are going to run "billions" of instruction, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- Coherence (Invalidation): other process (e.g., I/O) updates memory



Finding a Block in a Cache?



- Block is minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don't have data select field
- Index Used to Lookup Candidates in Cache
  - Index identifies the set of possibilities (check tag)
- Tag used to identify actual the block (what address?)
  - If no candidates match, then declare cache miss

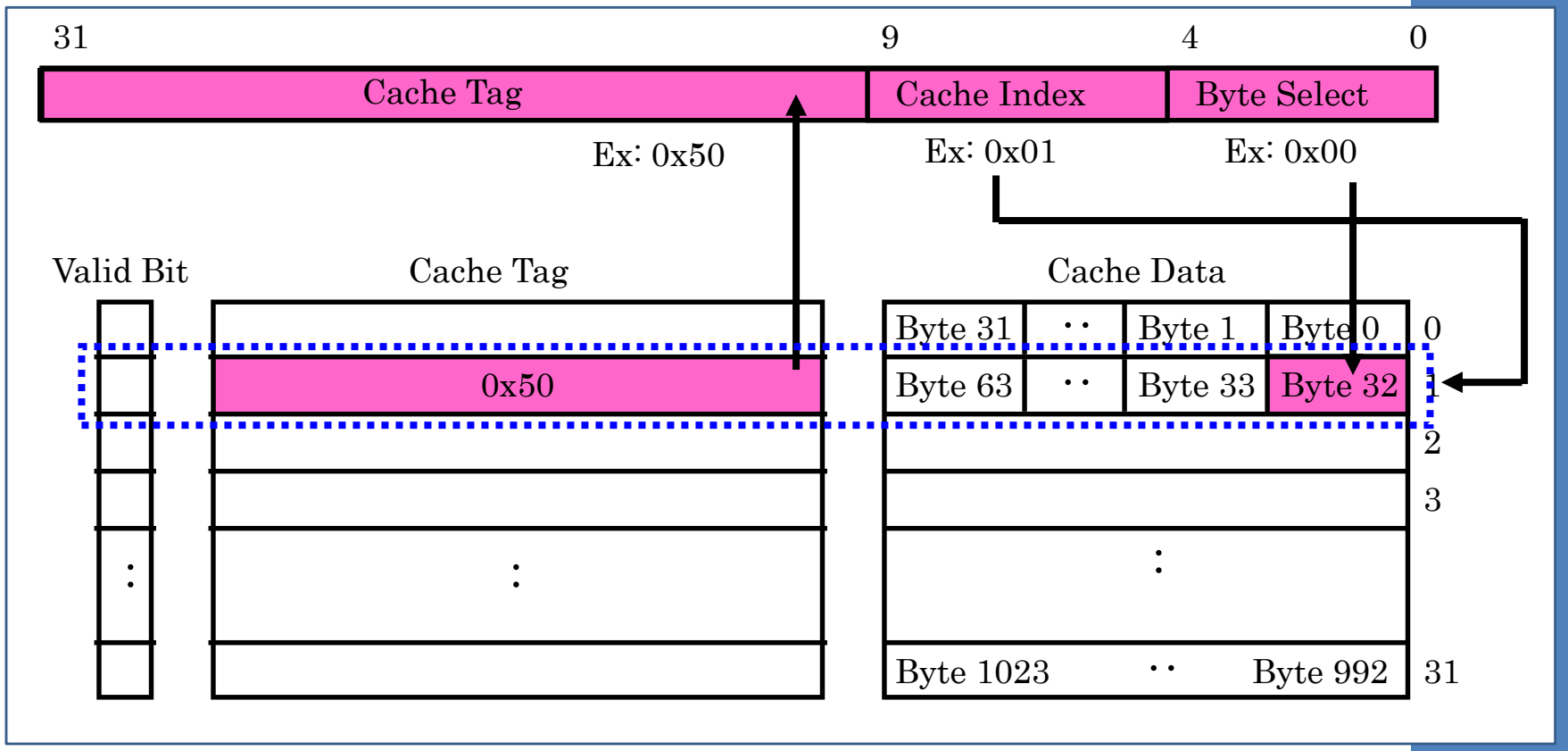


Direct-Mapped Cache

- Direct Mapped 2^N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2^M)

Example: 1 KB Direct Mapped Cache with 32B Blocks

- Index chooses potential block
- Tag checked to verify block
- Byte select chooses byte within block

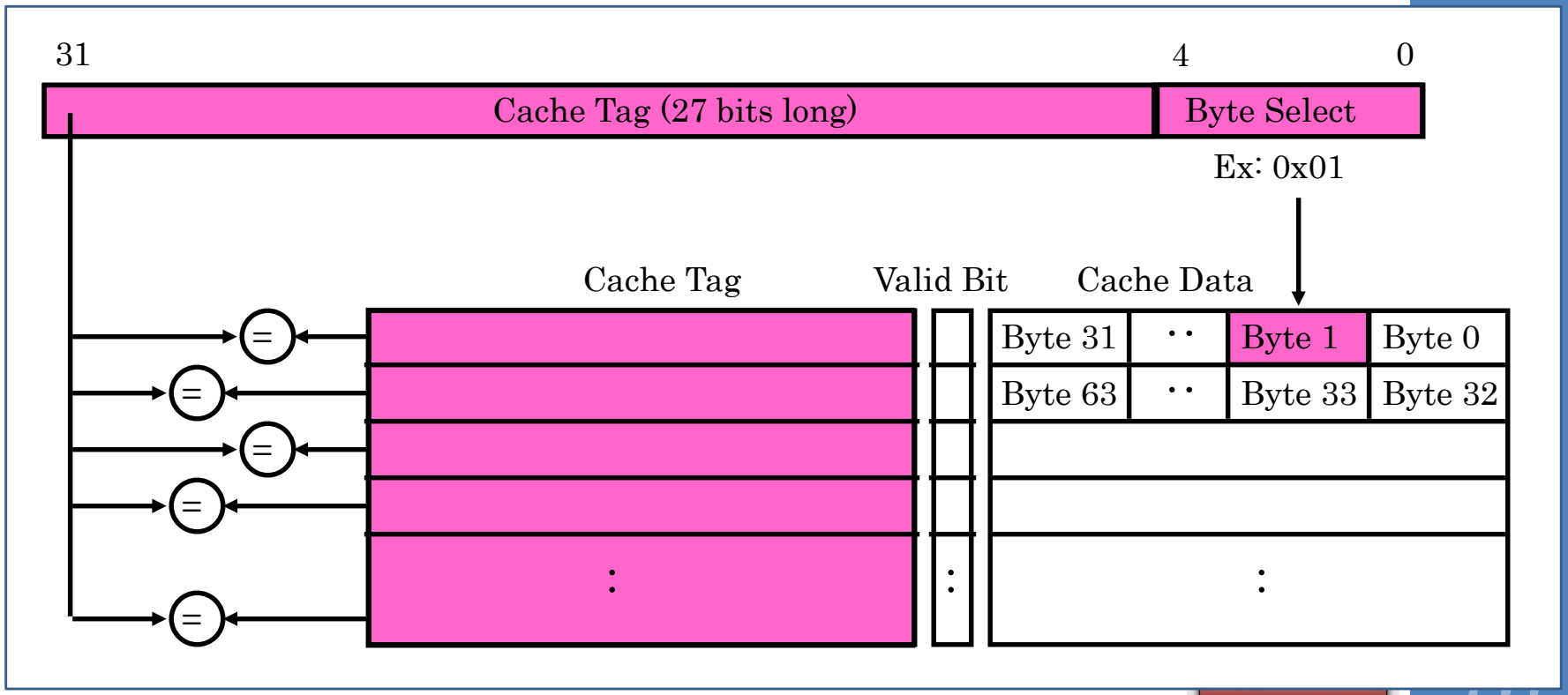


Fully Associative Cache

- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare tags of all cache entries in parallel

Example: Block Size=32B blocks

- We need N 27-bit comparators
- Still have byte select to choose from within block

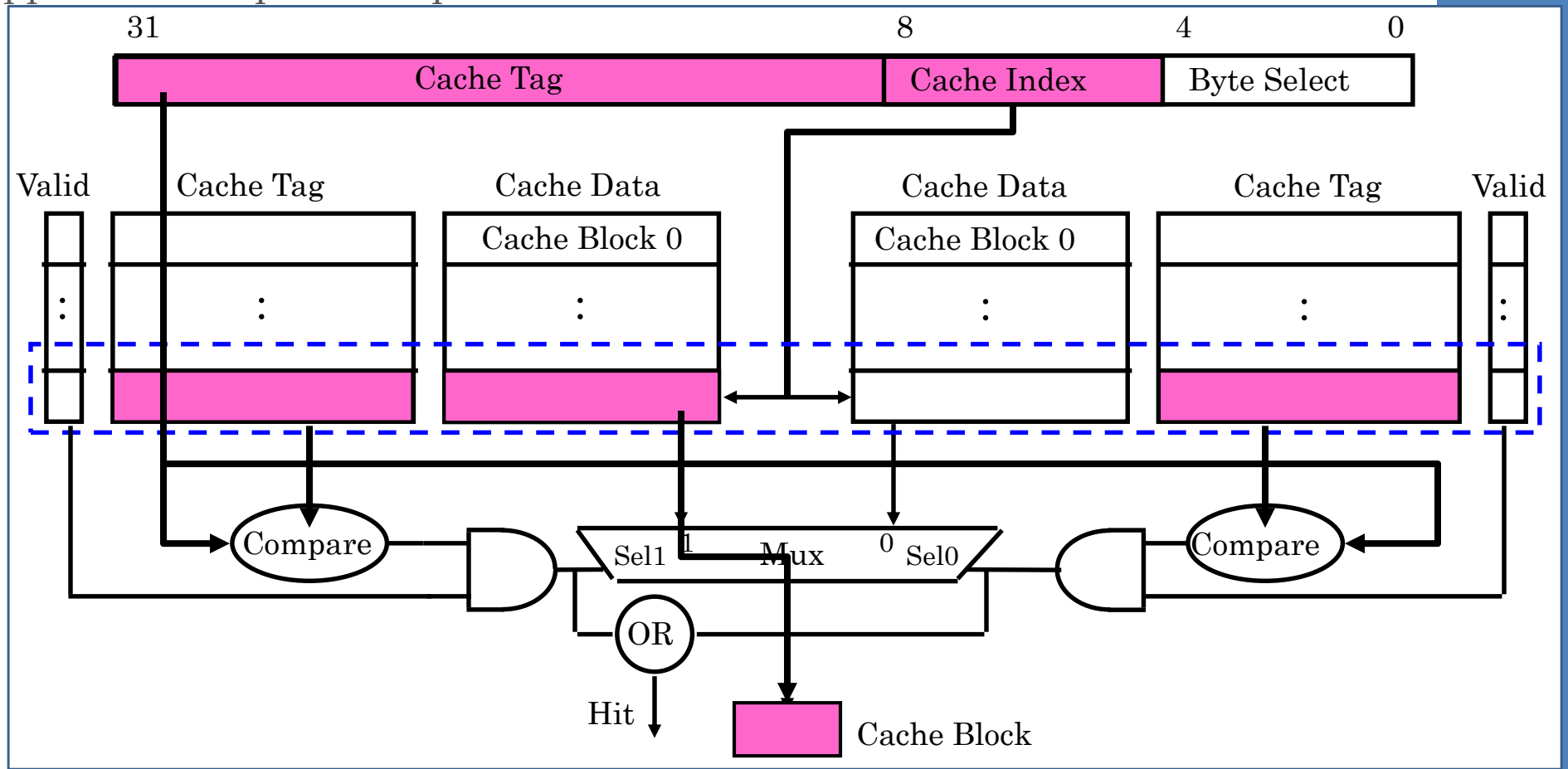


Set-Associative Cache

- N-way set associative: N entries per Cache Index
  - N direct mapped caches operate in parallel

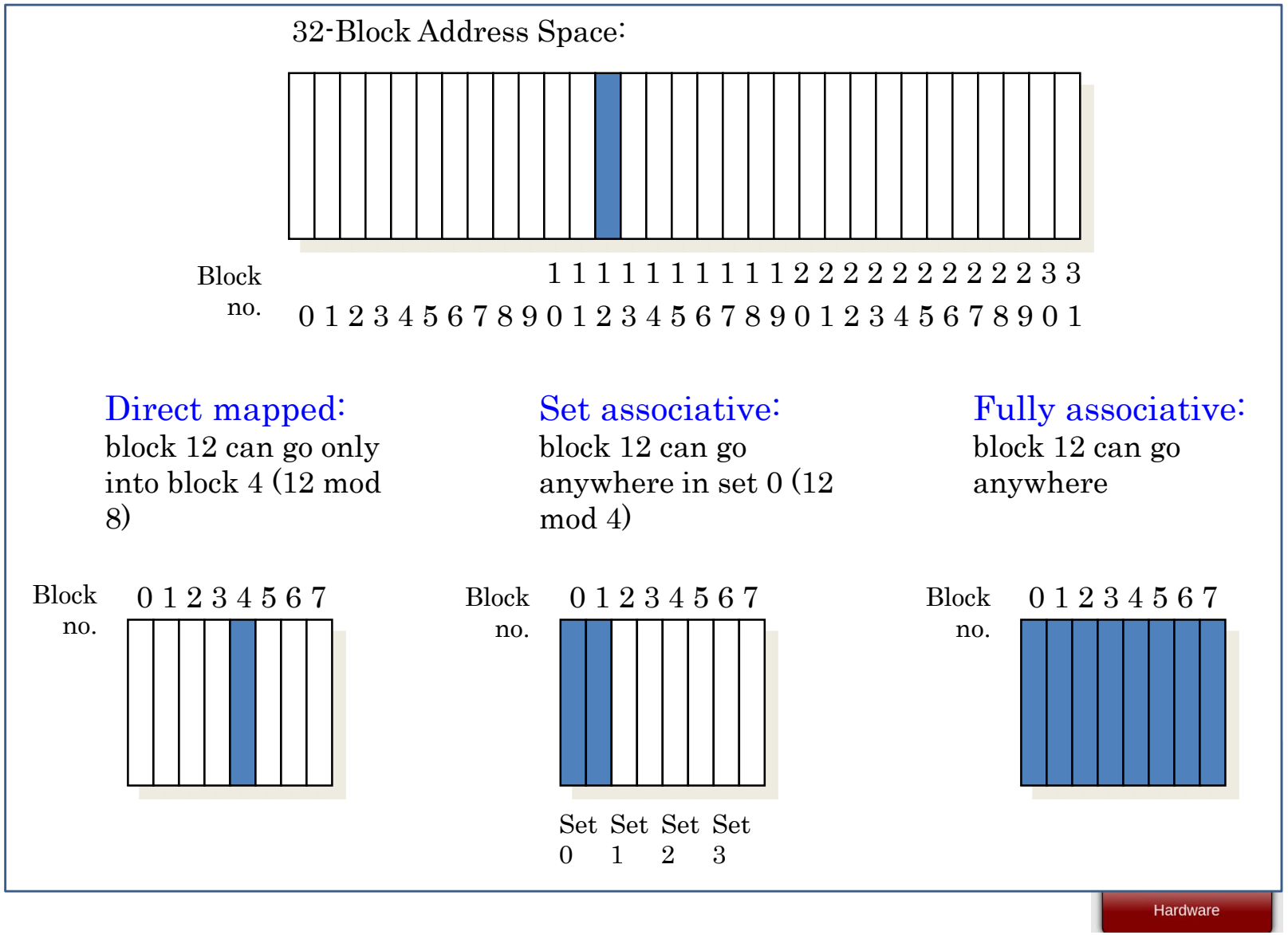
Example: Two-way set associative cache

- Cache Index selects a "set" from the cache
- Two tags in the set are compared to input in parallel
- Data is selected based on the tag result



Where Does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8-block cache
  - Address space has 32 blocks

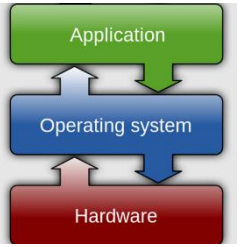


Which Block to Replace on a Miss?

- Easy for Direct Mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

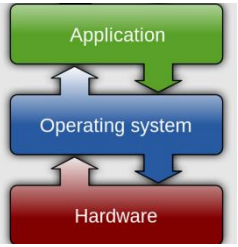
- Miss rates for a workload:

	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%



Recall: What Happens on a Write?

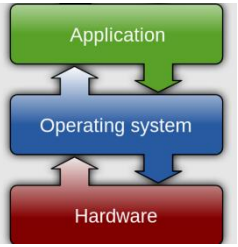
- Write through: The information is written to both the block in the cache and to the block in the lower-level memory
- Write back: The information is written only to the block in the cache
  - Modified cache block is written to main memory only when it is replaced
  - Question is block clean or dirty?
- Pros and Cons of each?
  - WT:
    - PRO: read misses cannot result in writes
    - CON: Processor held up on writes unless writes buffered
  - WB:
    - PRO: repeated writes not sent to DRAM
      processor not held up on writes
    - CON: More complex
      Read miss may require writeback of dirty data



What does our understanding of caches tell us about TLB design?

What TLB Organization Makes Sense?

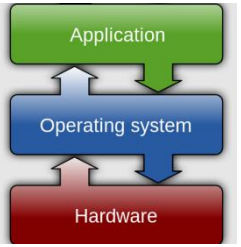
- Needs to be really fast
  - Seems to argue for Direct Mapped or Low Associativity
- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high! (PT traversal)
  - Cost of Conflict (Miss Time) is high
  - Hit Time – dictated by clock cycle
- Thrashing: continuous conflicts between accesses
  - What if use low order bits of page as index into TLB?
    - First page of code, data, stack may map to same entry
    - Need 3-way associativity at least?
  - What if use high order bits as index?
    - TLB mostly unused for small programs



TLB Organization: Include Protection

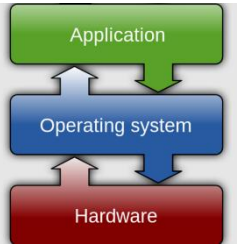
- How big does TLB actually have to be?
  - Usually fewer entries than the cache (why?)
  - Not very big, can support higher associativity
- Small TLBs usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info
- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a "TLB Slice"
- Example for MIPS R3000:

Virtual Address	Physical Address	Dirty	Ref	Valid	Access	ASID
0xFA00	0x0003	Y	N	Y	R/W	34
0x0040	0x0010	N	Y	Y	R	0
0x0041	0x0011	N	Y	Y	R	0



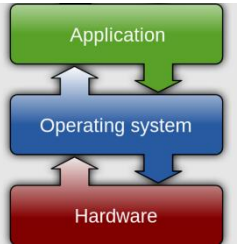
Example: Pentium-M TLBs (2003)

- Four different TLBs
  - Instruction TLB for 4K pages
    - 128 entries, 4-way set associative
  - Instruction TLB for large pages
    - 2 entries, fully associative
  - Data TLB for 4K pages
    - 128 entries, 4-way set associative
  - Data TLB for large pages
    - 8 entries, 4-way set associative
- All TLBs use LRU replacement policy
- Why different TLBs for instruction, data, and page sizes?

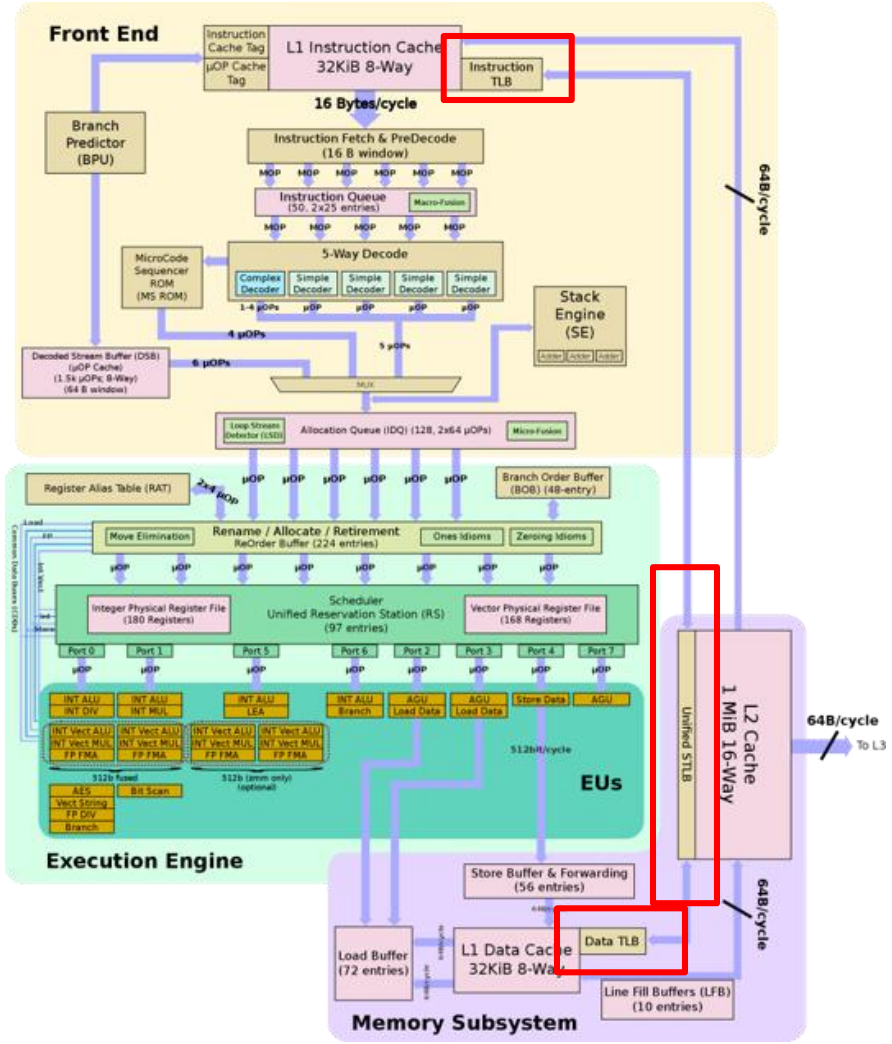


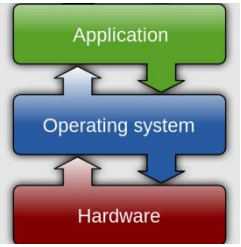
Example: Intel Nehalem (2008)

- L1 DTLB
  - 64 entries for 4 K pages and
  - 32 entries for 2/4 M pages,
- L1 ITLB
  - 128 entries for 4 K pages using 4-way associativity and
  - 14 fully associative entries for 2/4 MiB pages
- unified 512-entry L2 TLB for 4 KiB pages, 4-way associative



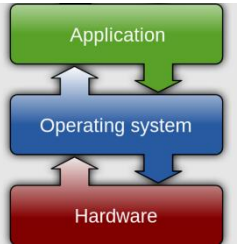
Example: Skylake, Cascade Lake





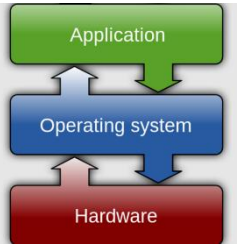
Current Example: Memory Hierarchy

- Caches (all 64 B line size)
  - L1 I-Cache: 32 KB/core, 8-way set assoc.
  - L1 D Cache: 32 KB/core, 8-way set assoc., 4-5 cycles load-to-use, Write-back policy
  - L2 Cache: 1 MB/core, 16-way set assoc., Inclusive, Write-back policy, 14 cycles latency
  - L3 Cache: 1.375 MB/core, 11-way set assoc., shared across cores, Non-inclusive victim cache, Write-back policy, 50-70 cycles latency
- TLB
  - L1 ITLB, 128 entries; 8-way set assoc. for 4 KB pages
    - 8 entries per thread; fully associative, for 2 MB / 4 MB page
  - L1 DTLB 64 entries; 4-way set associative for 4 KB pages
    - 32 entries; 4-way set associative, 2 MB / 4 MB page translations
    - 4 entries; 4-way associative, 1G page translations
  - L2 STLB: 1536 entries; 12-way set assoc. 4 KiB + 2 MB pages
    - 16 entries; 4-way set associative, 1 GB page translations



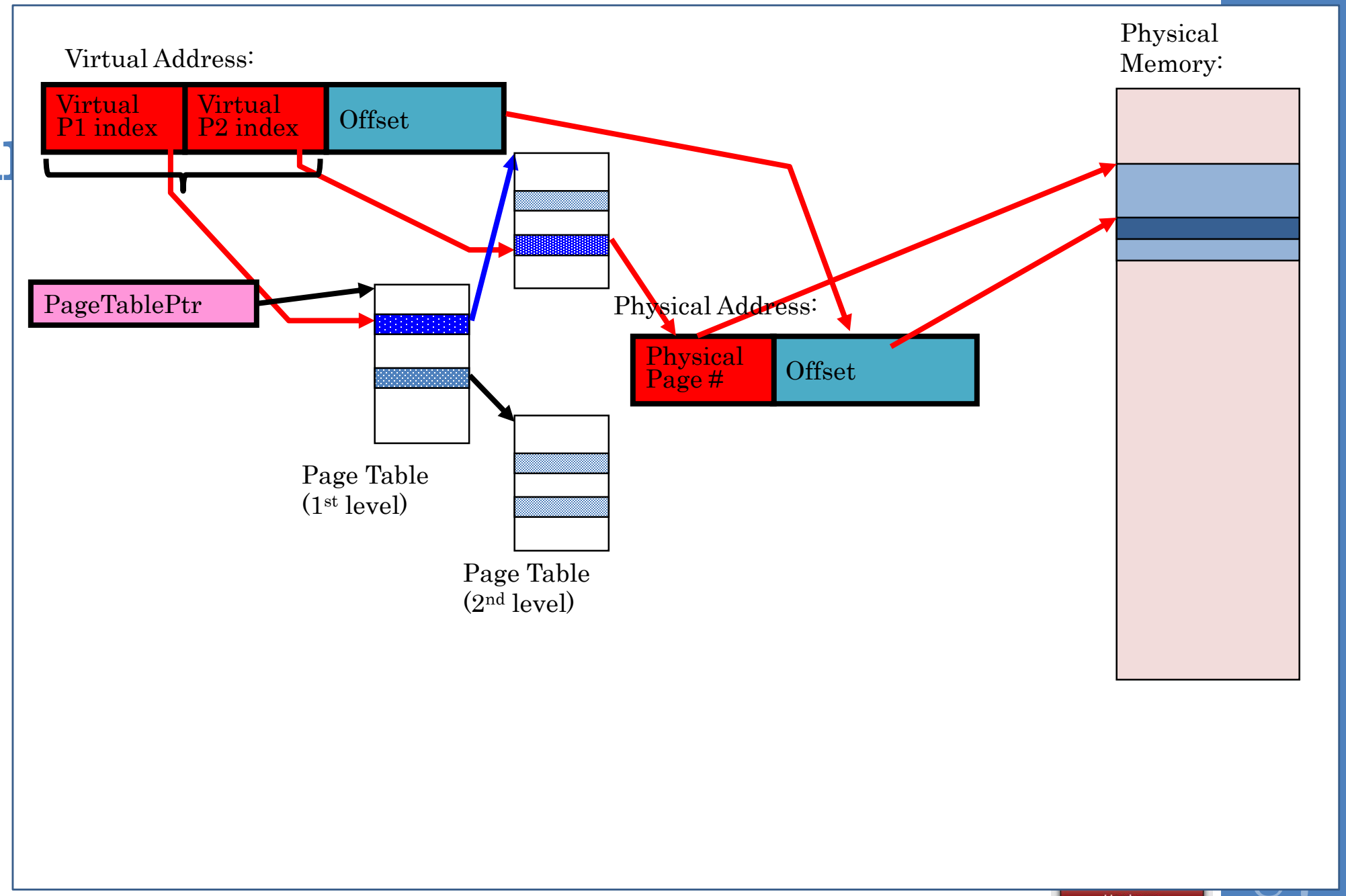
What Happens on a Context Switch?

- Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate TLB (simple but expensive)
  - Include ASID (address space identifier) in TLB
- What if the OS changes the page table?
  - Must invalidate TLB entry!
  - Called “TLB Consistency”



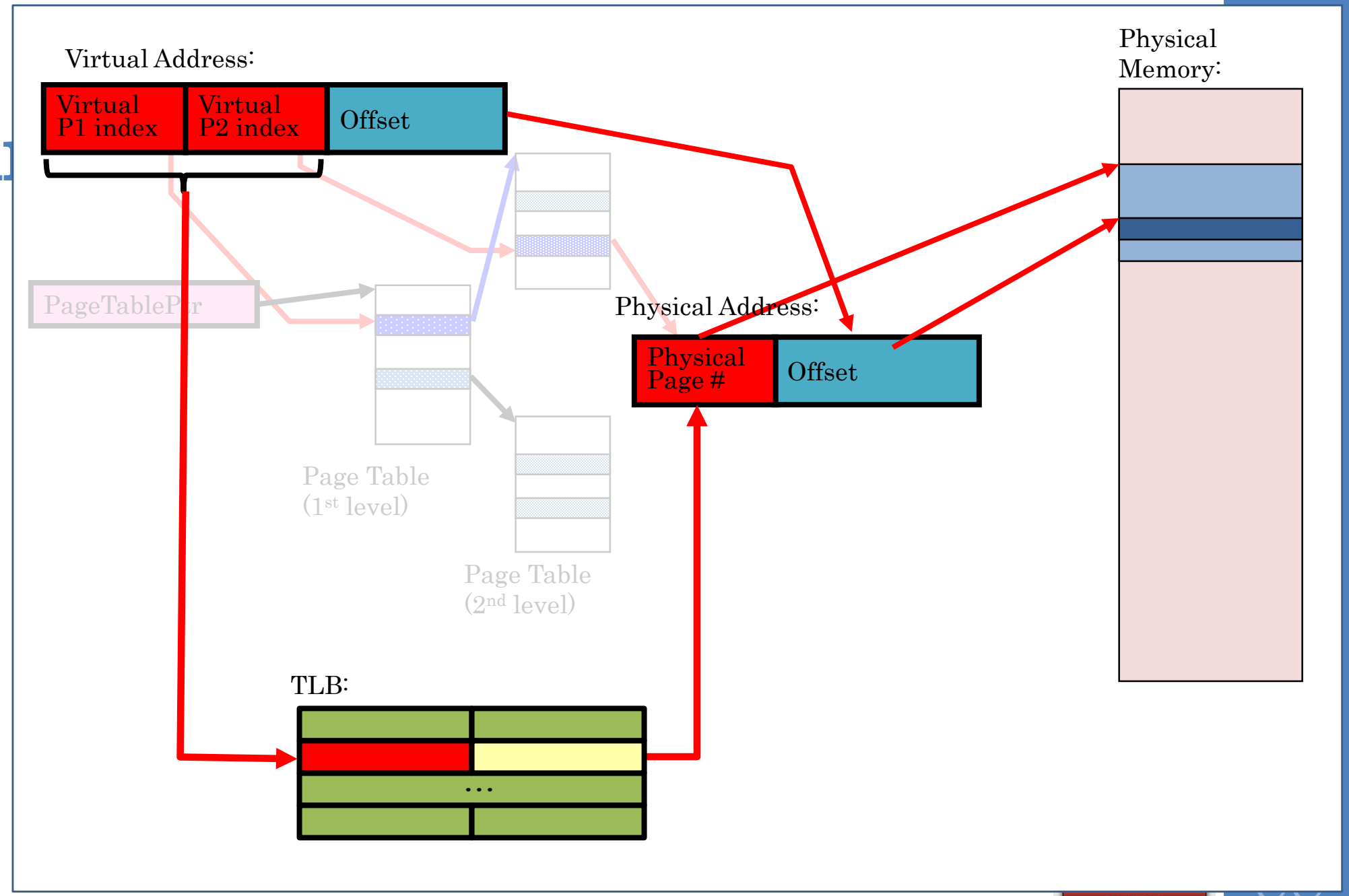
Putti

Address Translation



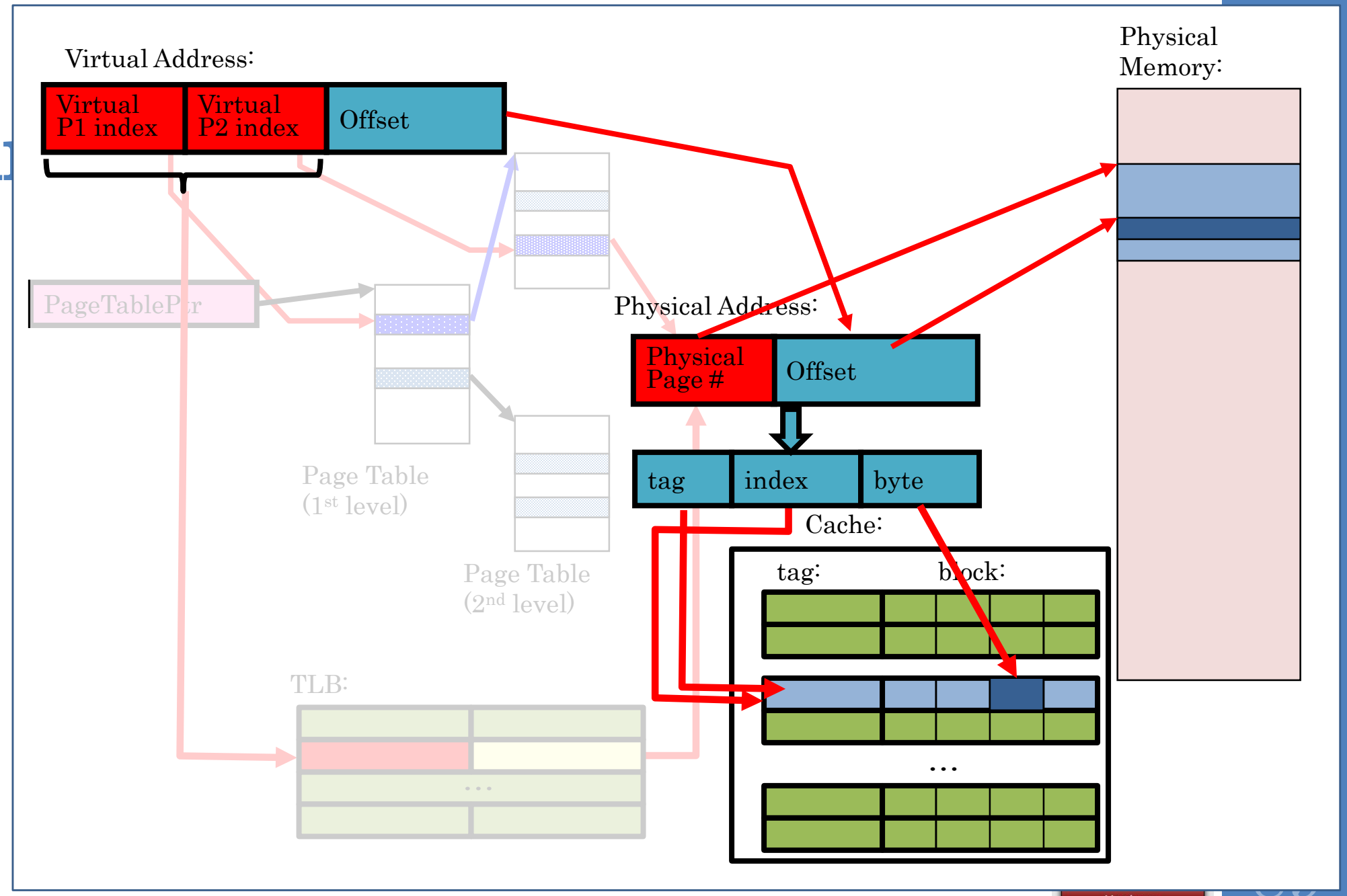
Putti

TLB

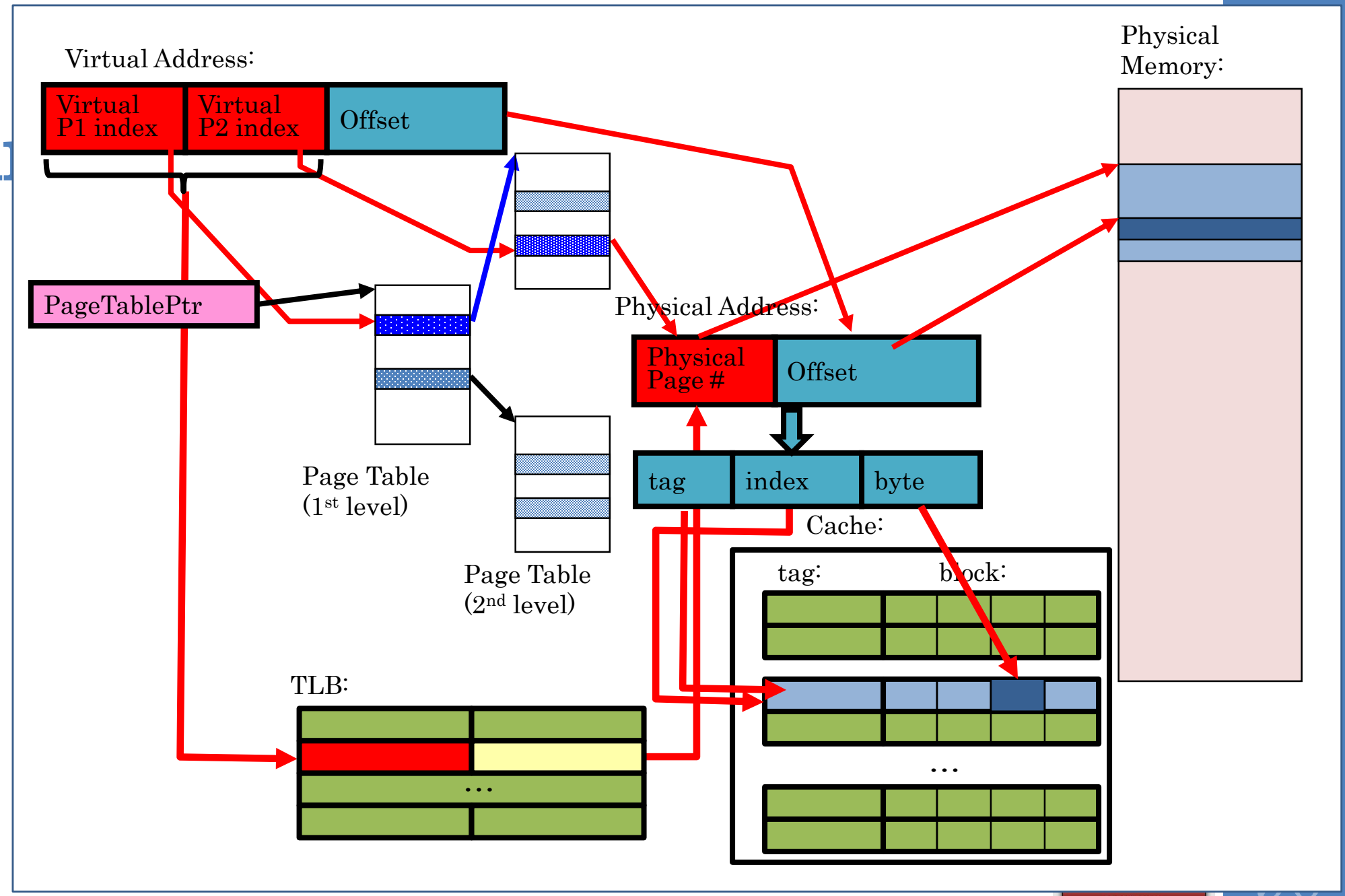

Virtual Address:
Virtual P1 index
Virtual P2 index
Offset
PageTablePtr
Page Table (1st level)
Page Table (2nd level)
Physical Address:
Physical Page #
Offset
Physical Memory:
TLB:
...

Putti

Cache

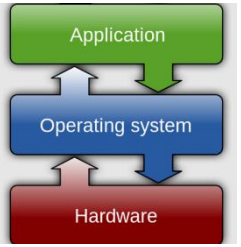

Virtual Address:
Virtual P1 index
Virtual P2 index
Offset
Physical Memory:
PageTablePtr
Page Table (1st level)
Page Table (2nd level)
Physical Address:
Physical Page #
Offset
tag
index
byte
Cache:
tag:
block:
TLB:

Putti


Virtual Address:
Virtual P1 index
Virtual P2 index
Offset
PageTablePtr
Page Table (1st level)
Page Table (2nd level)
Physical Address:
Physical Page #
Offset
tag
index
byte
Physical Memory:
Cache:
tag:
block:
TLB:

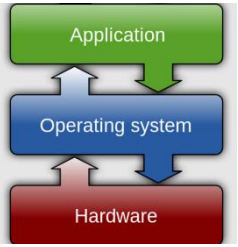
Summary: Page Tables

- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical page number (base address of physical page)
  - Offset of virtual address same as offset of physical address
  - Large page tables can be placed into virtual memory
- Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- Inverted Page Table
  - Use of hash-table to hold translation entries
  - Size of page table ~ size of physical memory rather than size of virtual memory



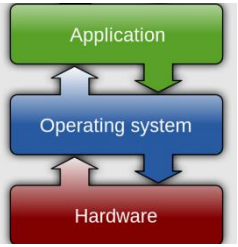
Summary: Caching

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- Three (+1) Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity
  - Capacity Misses: increase cache size
  - Coherence Misses: Caused by external processors or I/O devices
- Cache Organizations:
  - Direct Mapped: single block per set
  - Set associative: more than one block per set
  - Fully associative: all entries equivalent



Summary: TLBs

- "Translation Lookaside Buffer" (TLB)
  - Small number of PTEs and optional process IDs (< 512)
  - Fully Associative (Since conflict misses expensive)
  - On TLB miss, page table must be traversed and if located PTE is invalid, cause Page Fault
  - On change in page table, TLB entries must be invalidated
  - TLB is logically in front of cache (need to overlap with cache access)
- On Page Fault, OS can take actions to resolve the situation
  - Demand paging, automatic memory management
  - Make copy of existing page for process
  - On process start, don't have to load much of executable into memory
  - Rarely used code and data may never get paged in
- Need to handle the exception carefully



LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP