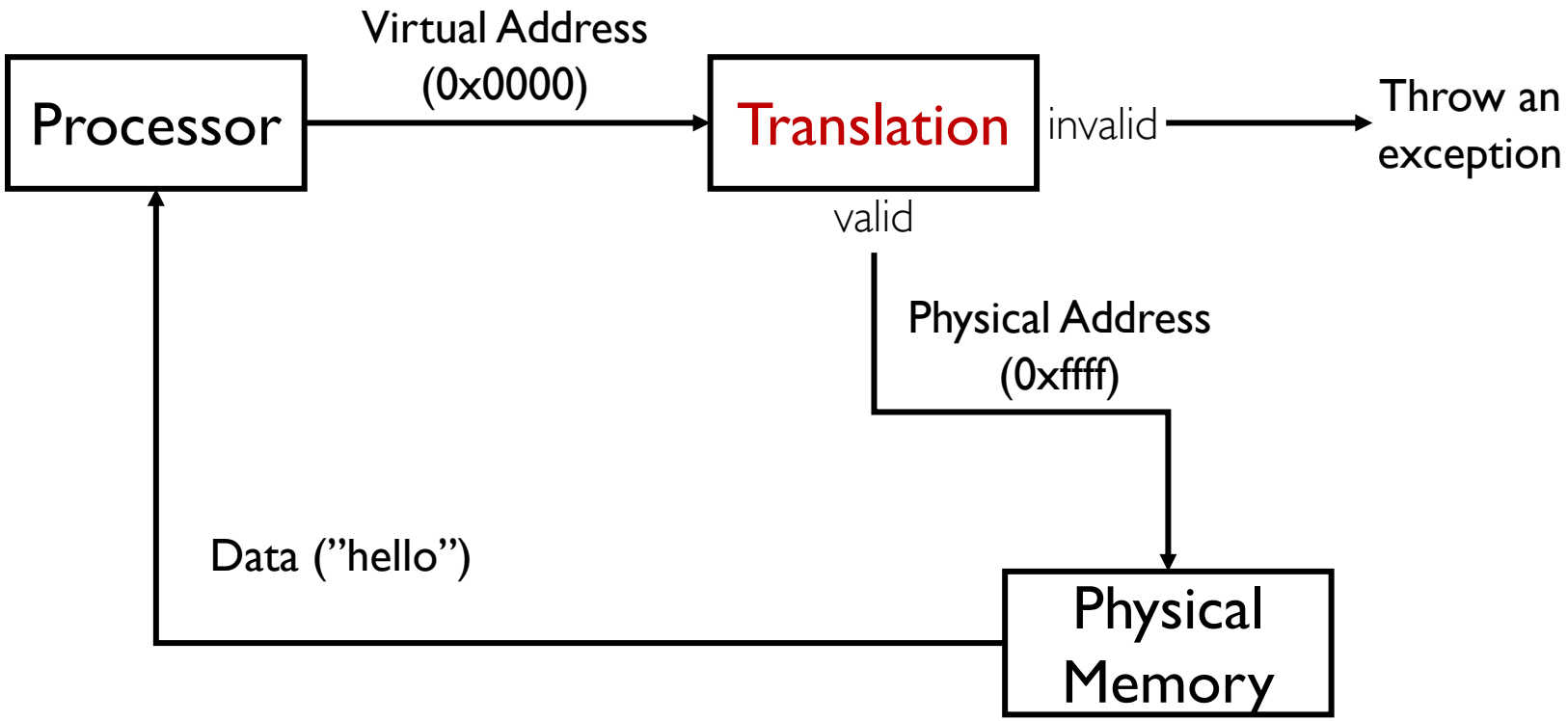
Operating Systems
Lecture 7

TLB and cache

Prof. Mengwei Xu

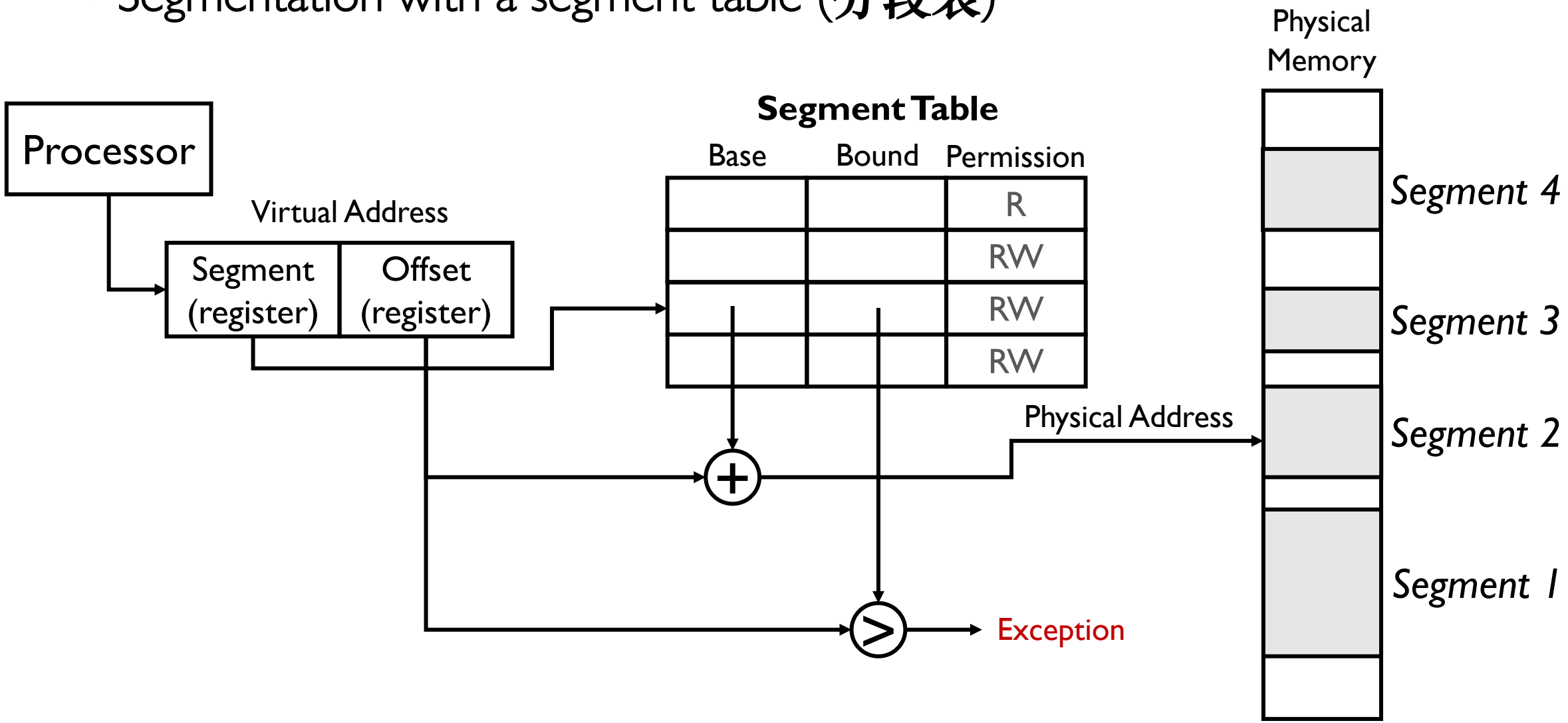
Recap: Address Translation

- From virtual memory address (虚拟内存地址) to physical memory address (物理内存地址)



Recap: Segmented Memory

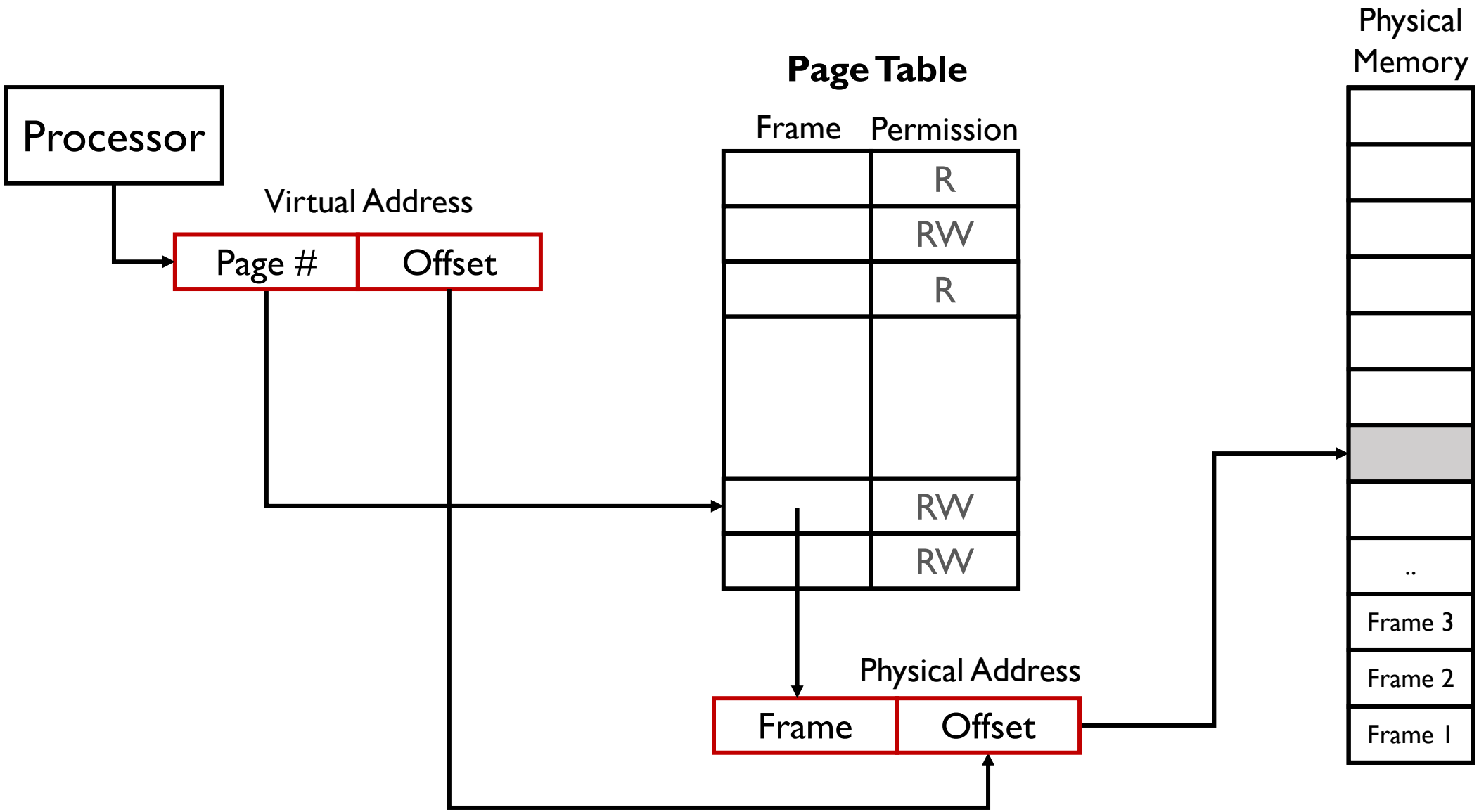
- Segmentation with a segment table (分段表)



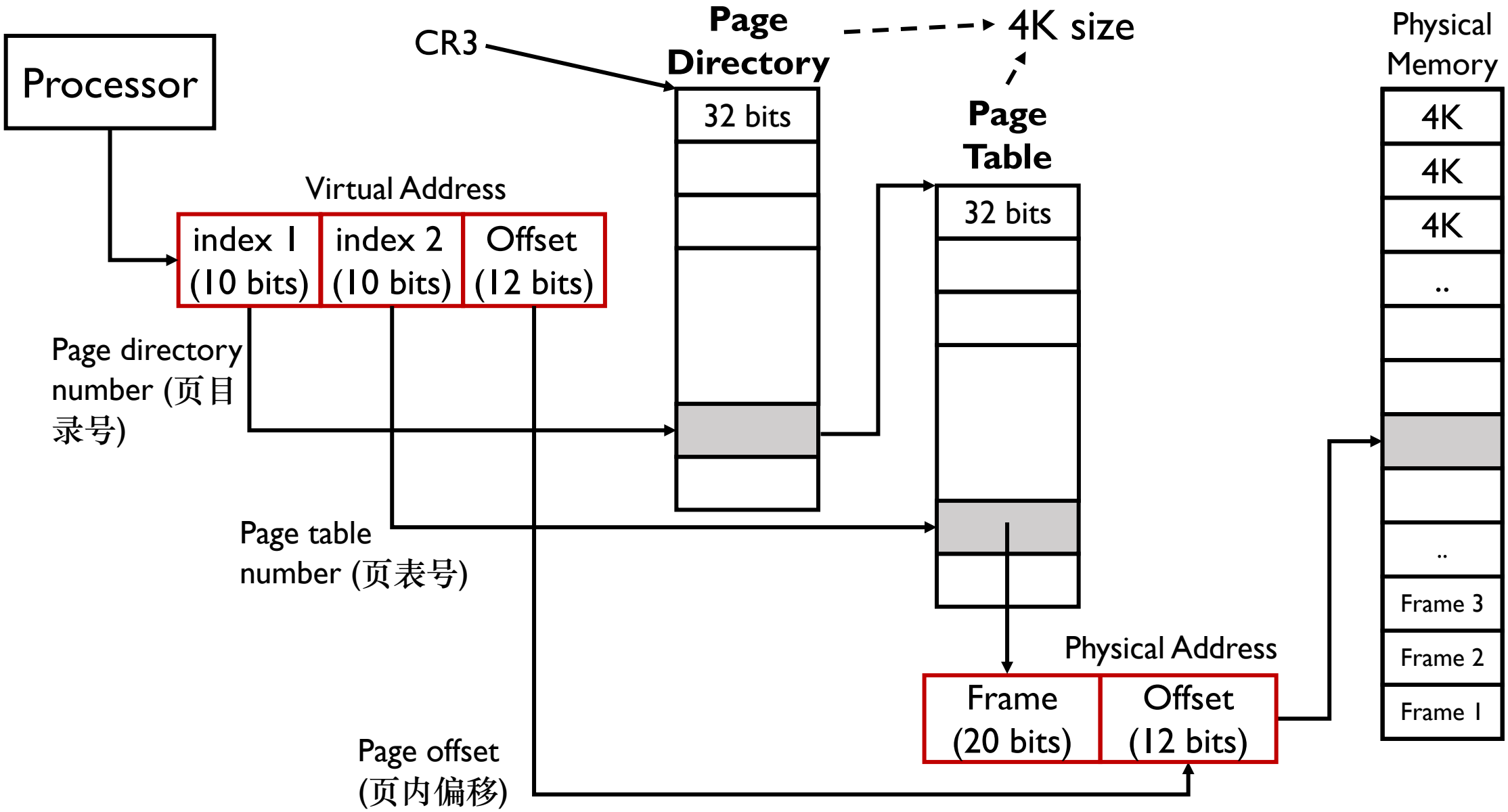
Recap: Paged Memory

- Paging (分页): allocating memory in fixed-sized chunks called page frames (页框)
- A page table (页表) stores for each process whose entries contain pointers to the page frames.
  - More compact than segment table because it does not need to store "bound"
- What's cool: the pages are scattered across physical memory regions
  - Yet within a page, the memory access is contiguous
  - For instance, a large matrix might span many pages
- Memory allocation becomes very simple: find a page frame.

Recap: Paged Memory

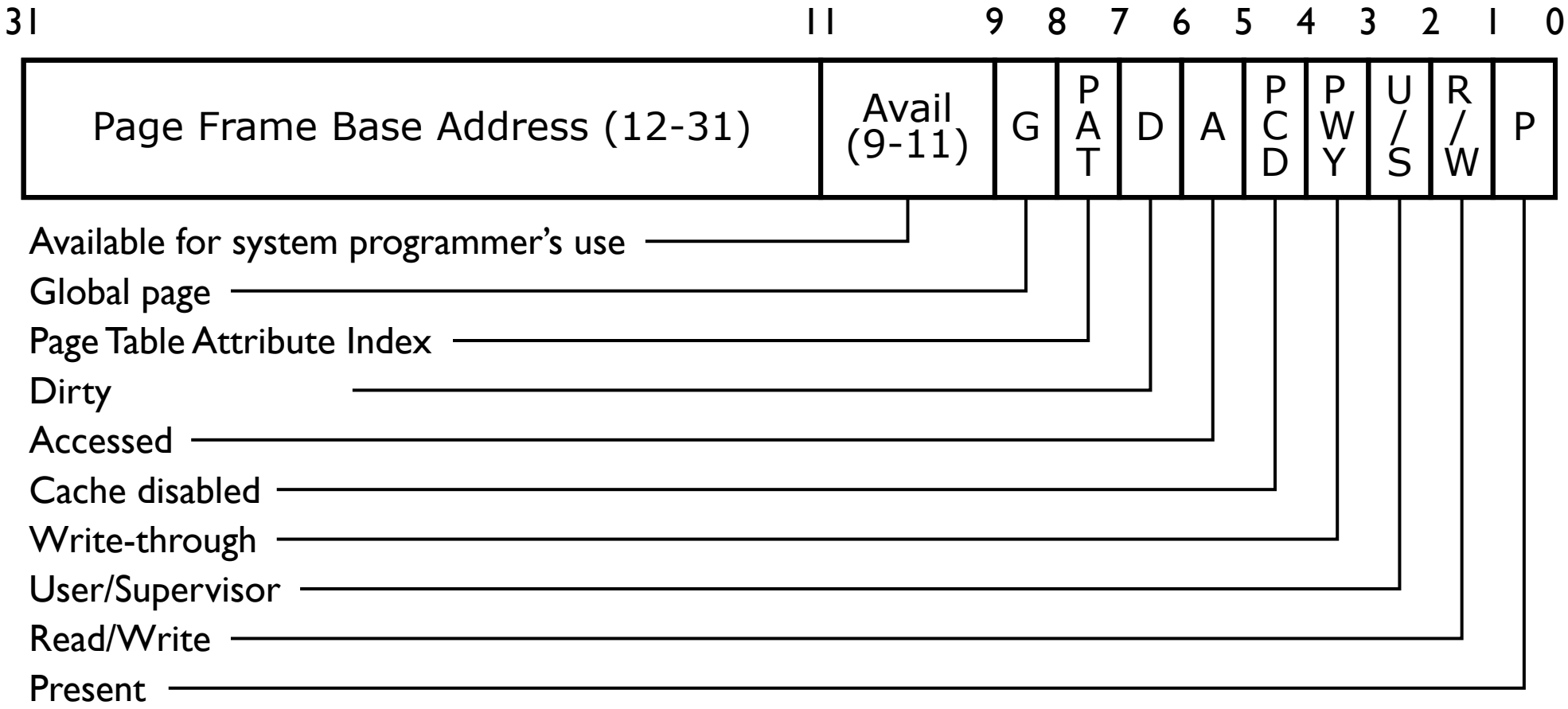


Recap: x86 Multi-level Paging



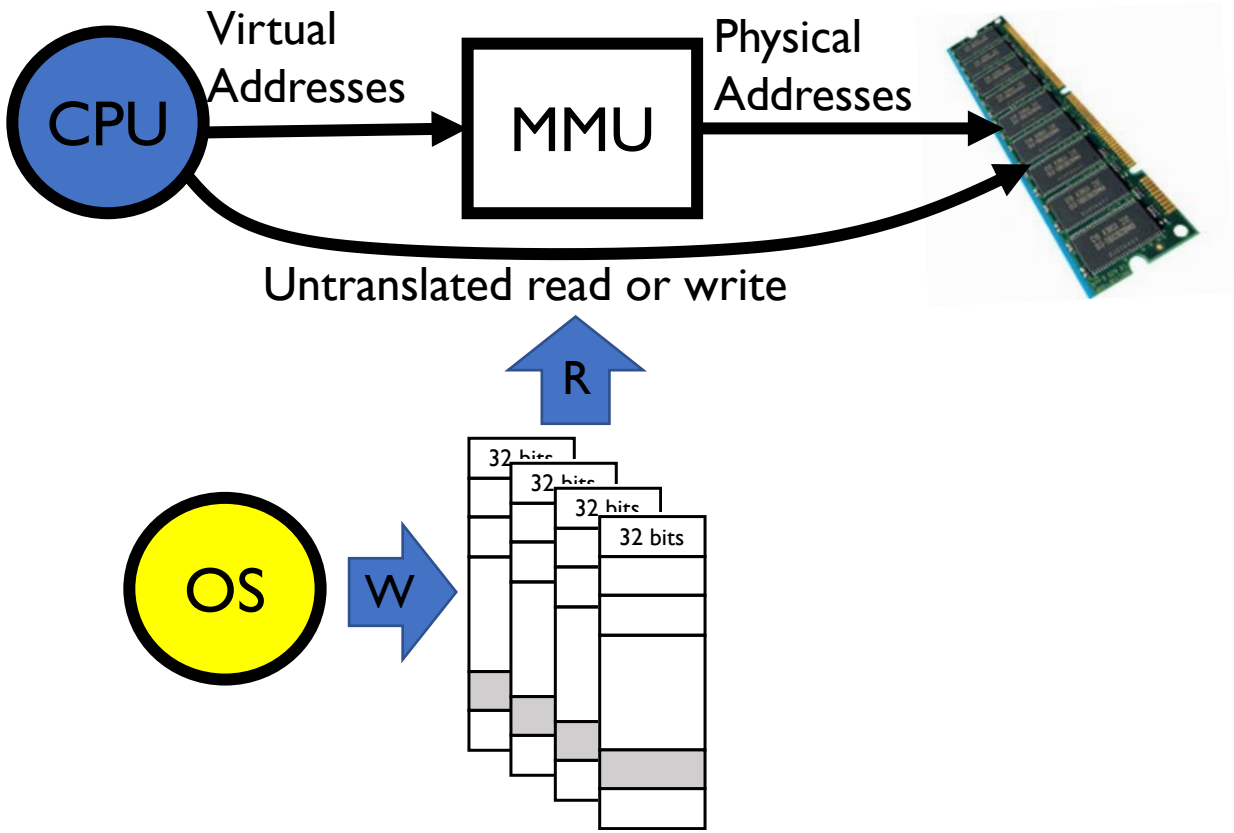
Recap: x86 Multi-level Paging

- Each page table entry (PTE, 页表项) is 32-bits long.



Recap: x86 Multi-level Paging

- Memory management unit (MMU, 分页内存管理单元): the hardware that actually does the translation
  - Usually located in CPU

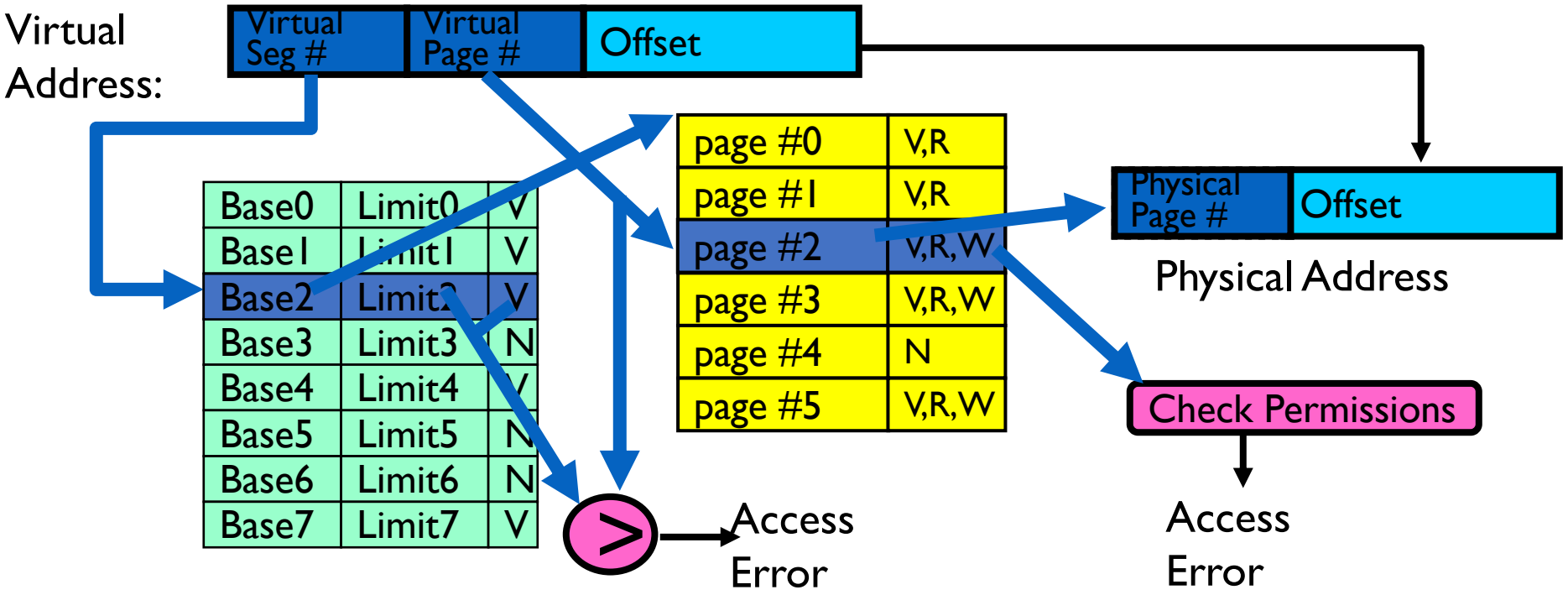


Recap: Multi-level Paging Summary

- Pros:
  - Only need to allocate as many page table entries as we need for application
    - In other wards, 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!

Segments + Paging

- 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):



Segmentation vs. Paging

- Intel x86 and Linux
  - 8086 era: segmentation and paging are both used
  - 80386 era: the segmentation is not really used
    - ❑ The processor provides 4 modes: none; paging only; segmentation only; both.
    - ❑ The CS is always set to 0 and the limit is 2^32.
  - x86_64 era: segmentation is considered as a legacy and not used in most OSes

- Now, everyone uses paging, few make any real use of segmentation.

https://softwareengineering.stackexchange.com/questions/100047/why-not-segmentation

Copy-on-Write (COW)

- How to implement an efficient fork()?
  - Do not copy all contents immediately, but mark the page/segment tables of both child and parent processes as "read-only"
  - When a write (from either child or parent) happens, it traps into kernel through page fault, and a private page is copied.

- A fork() followed immediately by a exec(), how many pages are really copied?

Goals for Today

- Cache Concepts
- TLB
- Memory Cache

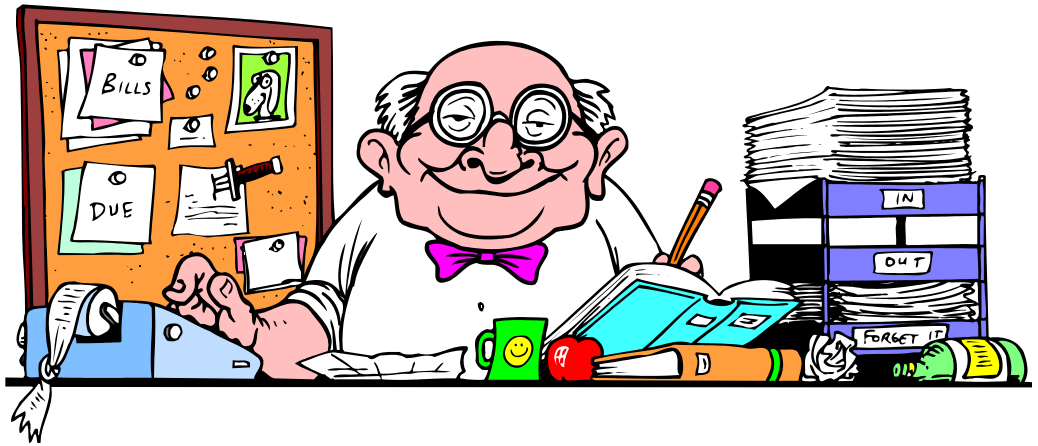
Goals for Today

- Cache Concepts
- TLB
- Memory Cache

Cache Concept

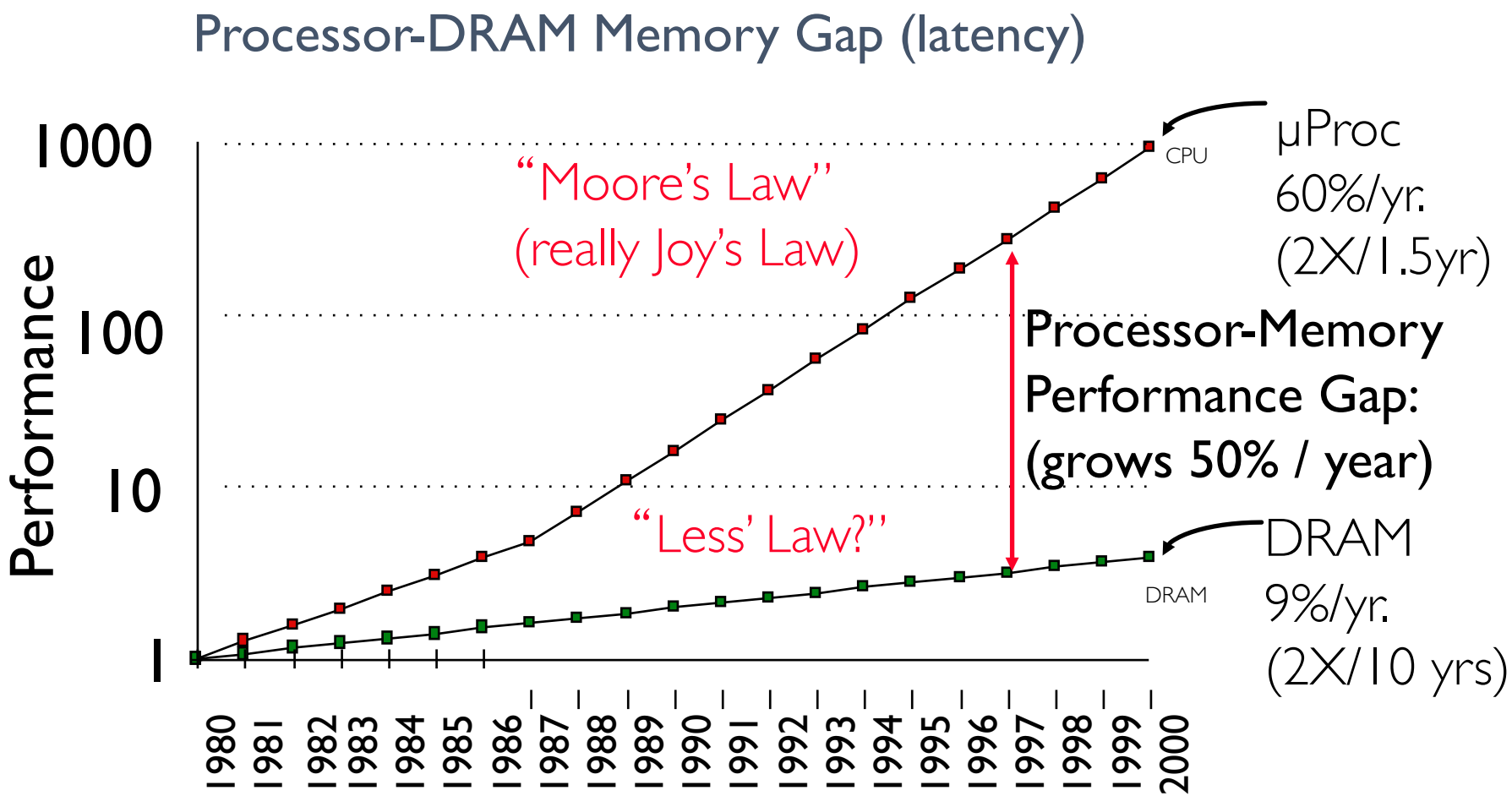
- Cache (缓存): a repository for copies that can be accessed more quickly than the original
  - One of the most widely adopted concept in computer systems: architecture, OS, distributed systems, network routes, etc..
  - Make frequent access fast!
  - Only works with high "cache hit"
- Average Access Time =

  (Hit Rate x Hit Time) + (Miss Rate x Miss Time)



Why Cache?

- Processing is often faster than I/O access

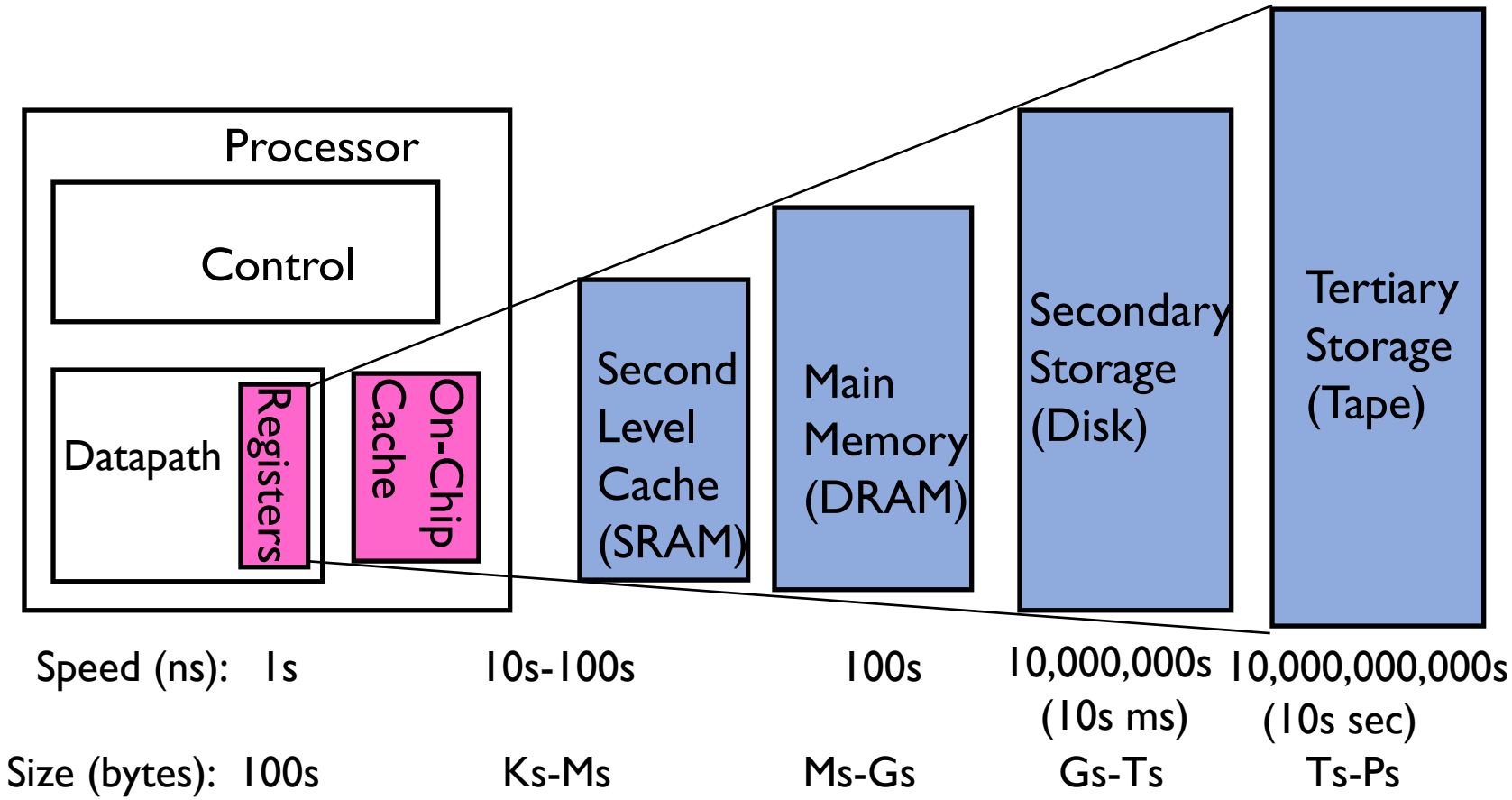


Locality: the Key to Cache Success

- Temporal locality (**时间局部性**): If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future.
  - To leverage: keep recently accessed data items closer to processor
- Spatial locality (**空间局部性**): if a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.
  - Move contiguous blocks to the upper levels

Memory Hierarchy

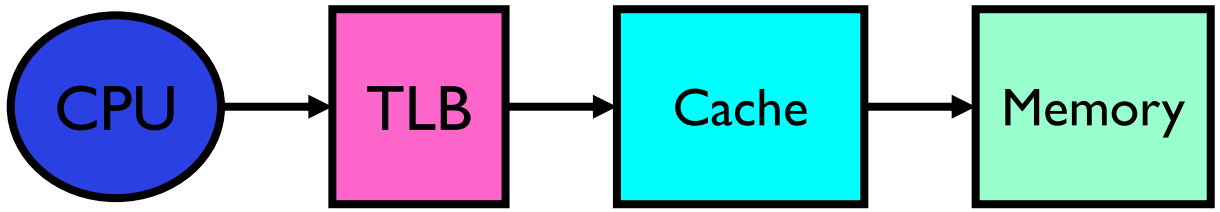
- Speed, Size, and Cost: take advantage of each level



Memory Hierarchy

- Speed, Size, and Cost: take advantage of each level
- Direct use of caching techniques
  - TLB (cache of PTEs)
  - Cache (cache of main memory, many levels)
  - Paged virtual memory (memory as cache for disk)
  - File systems (cache disk blocks in memory)
  - DNS (cache hostname => IP address translations)
  - Web proxies (cache recently accessed pages)

In this course:

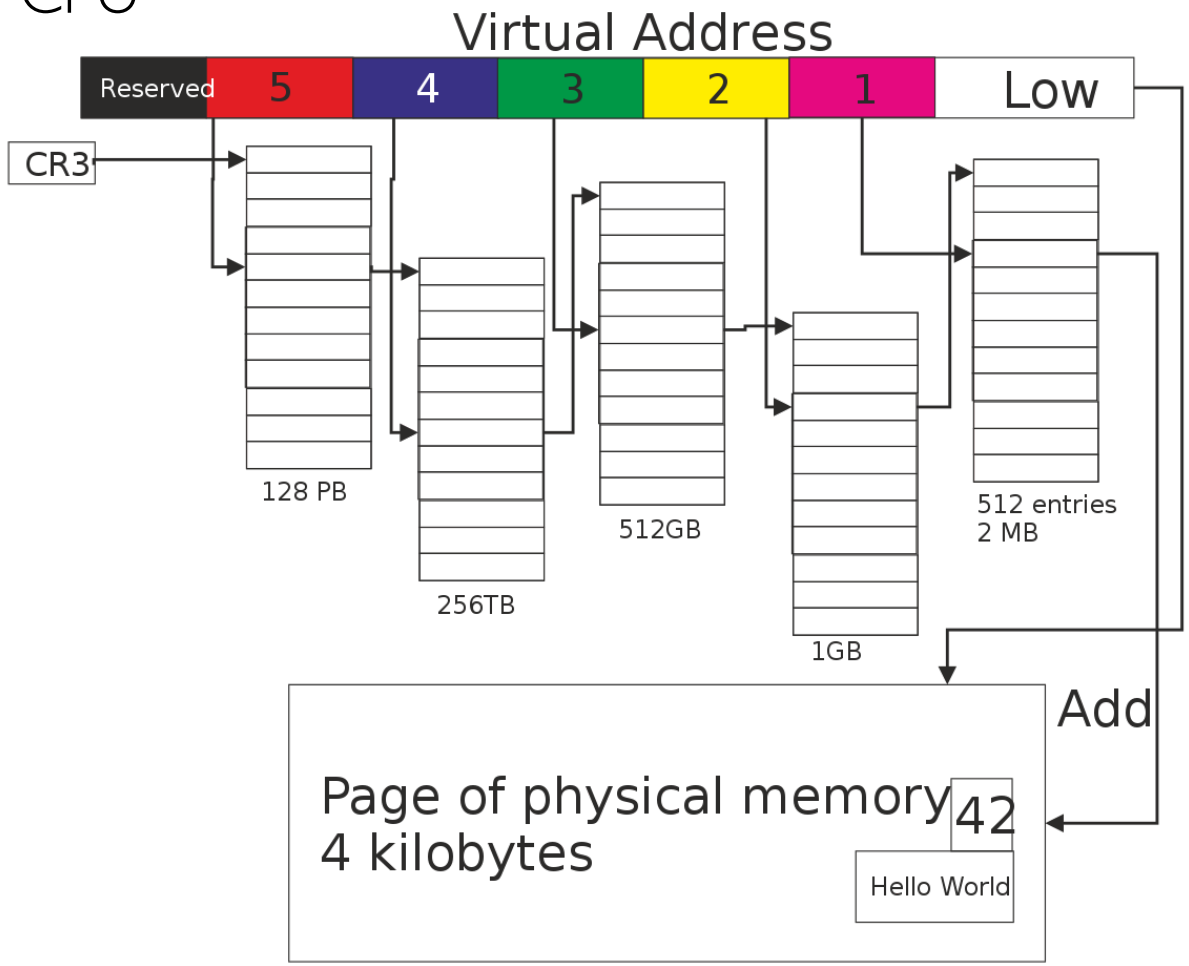


Goals for Today

- Cache Concepts
- TLB
- Memory Cache

Address Translation Problem

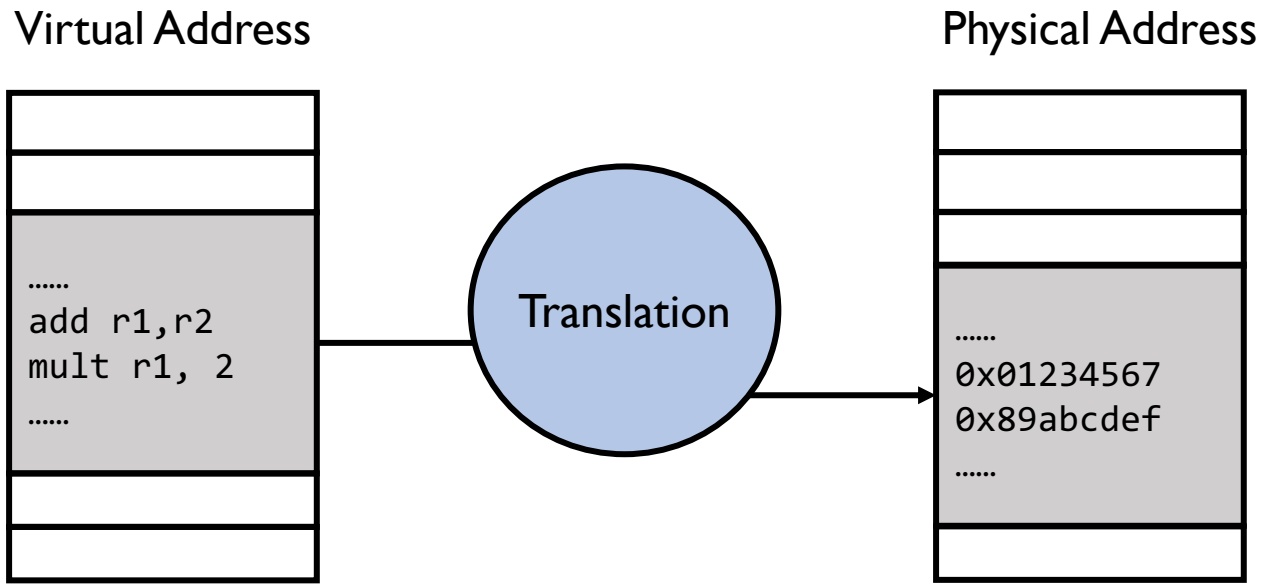
- It's too slow! Each memory access takes at least 2 extra memory access
  - Memory speed is often slower than CPU
  - With more levels of translation..



TLB as a Cache

- Translation Lookaside Buffers (TLB, 转换检测缓冲区): a special cache within MMU that accelerates address translation

- The time and spatial locality. Who are they?
- Memory mapping is page-aligned.



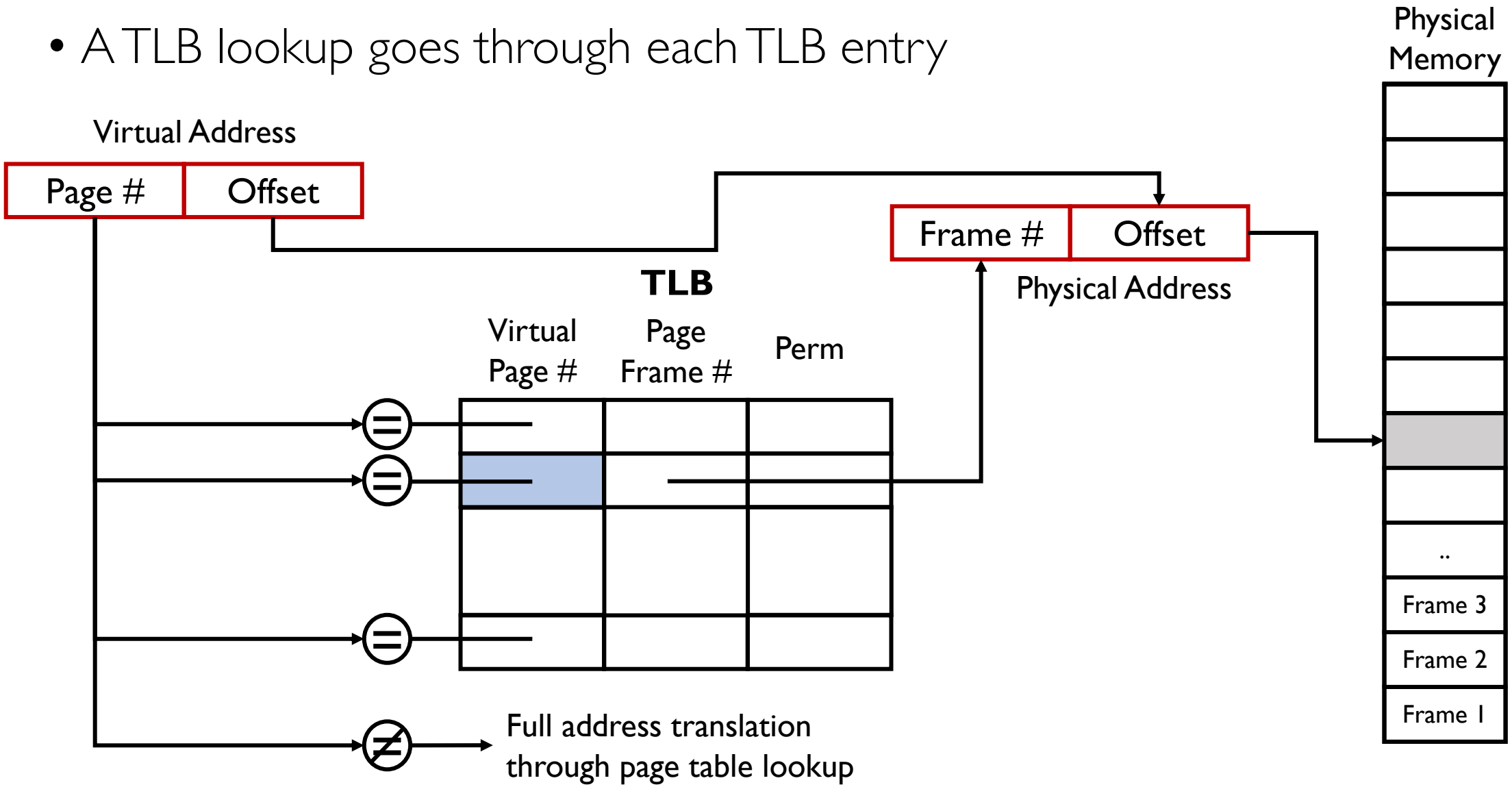
TLB Lookup

- A TLB lookup goes through each TLB entry
  - TLB hits if any entry matches so the physical page is fetched directly
  - TLB miss if none them matches. Do a full translation and use the physical address to replace an old entry in TLB.

```
TLB entry = {
        virtual page number,
        physical page frame number,
        access permissions
}
```

TLB Lookup

- A TLB lookup goes through each TLB entry



TLB Lookup

- A TLB lookup goes through each TLB entry
- TLBs are often set-associative to reduce the comparison
  - More in the cache courses

TLB Lookup

- A TLB lookup goes through each TLB entry
- TLBs are often set-associative to reduce the comparison
- Address translation cost with TLB
  - High TLB hit ratio is critical to translation performance

Cost(address translation) = Cost(TLB Lookup) +
Cost(full translation) x P(miss)

TLB Miss

- Sources
  - Page not accessed before
  - Page evicted due to limited TLB size
  - Page mapping conflict due to association
  - Other processes update the page table

TLB Miss

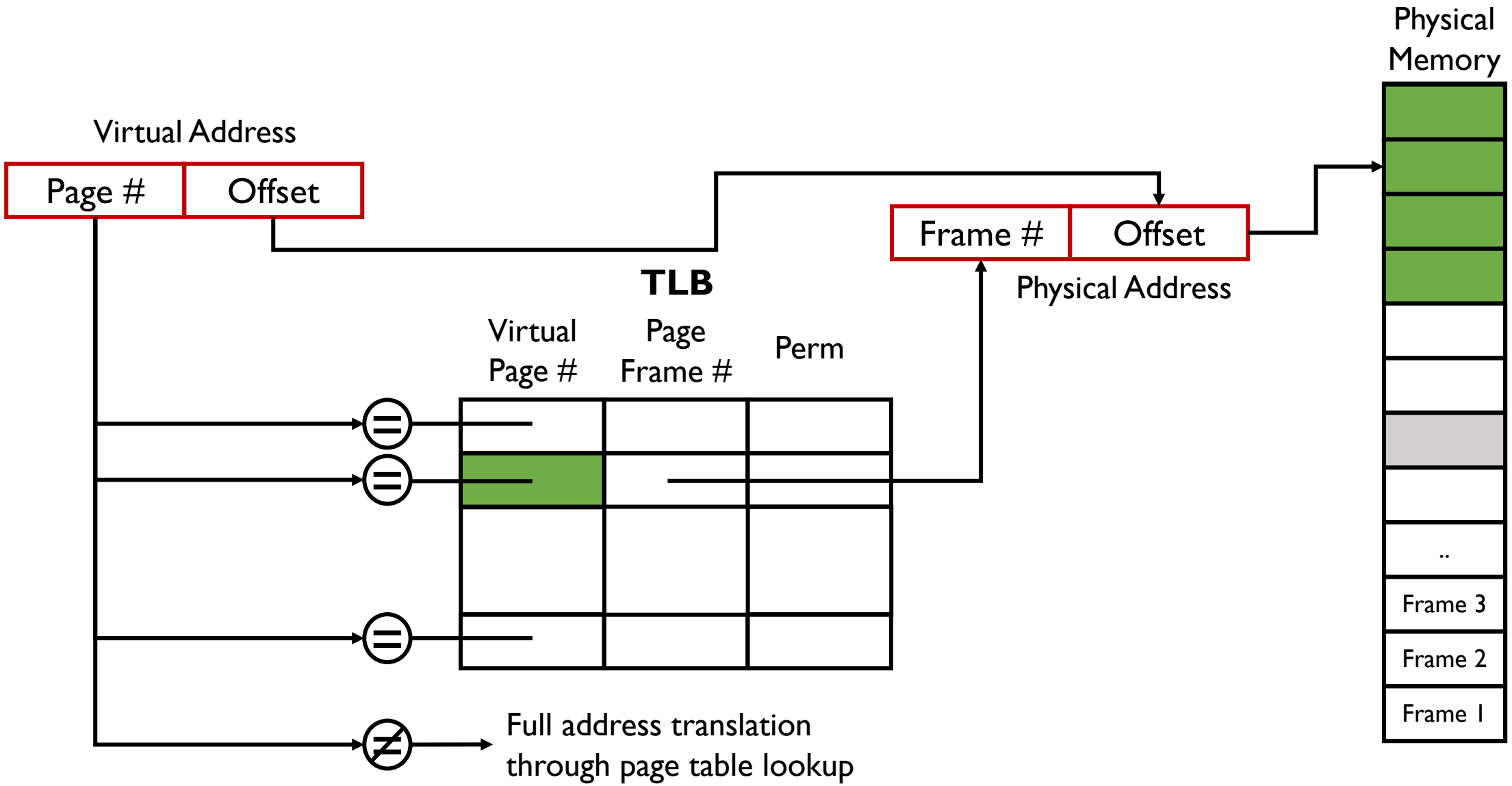
- (Mostly) Hardware traversed page tables:
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - ❑ If PTE valid, hardware fills TLB and processor never knows
    - ❑ If PTE marked as invalid, causes Page Fault, after which kernel decides what to do 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

Superpage

- Superpage: a set of contiguous pages in physical memory that map a contiguous regions of virtual memory, where the pages are aligned so that they share the same high-order (superpage) address
  - A way to increase the TLB cache hit ratio.
  - What is sacrificed?
- Matching superpages only comparing the most significant bits of the address, ignoring the offset within the superpage.
  - For a 2MB superpage, the offset is the lowest 21 bits in virtual address
  - For a 1GB superpage, the offset is the lowest 30 bits in virtual address
  - For those superpages, x86 skips one or two levels of the page table
- There is only one TLB entry for all the pages within the same superpage

Superpage


Physical Memory
Virtual Address
Page #
Offset
Frame #
Offset
Physical Address
TLB
Virtual Page #
Page Frame #
Perm
..
Frame 3
Frame 2
Frame 1
Full address translation through page table lookup

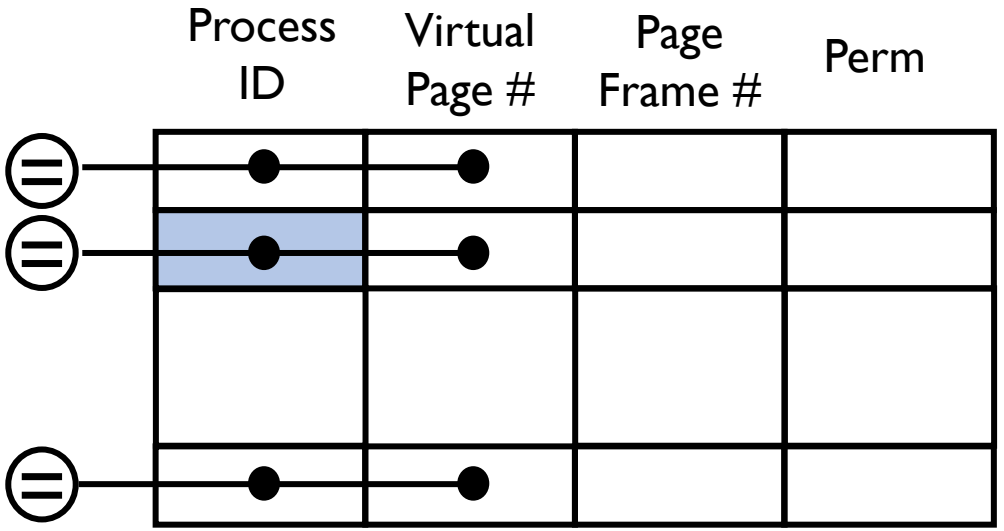
TLB Consistency

- Consistency (一致性) is a common issue for each cache: the cache must be always the same as the original data whenever the entries are modified.
  - Process context switch
  - Permission reduction
  - TLB shootdown

TLB Consistency

- Process Context Switch
  - Straightforward way: always flush the TLB when there is a context switch
  - Modern way: tagged TLB

```
TLB entry = {
        Process ID,
        virtual page number,
        physical page frame number,
        access permissions
}
```



TLB Consistency

- Permission reduction: when a mapping is discarded or the access permission reduces (from read-write to read-only).
  - Early computers discard the whole TLB; modern ones support the removal of individual TLB entries
  - If the pages are shared by processes..

```
108 static inline void
109 invlpg(void *addr)
110 {
111     asm volatile("invlpg (%0)" : : "r" (addr) : "memory");
112 }
```

TLB Consistency

- Permission reduction: when a mapping is discarded or the access permission reduces (from read-write to read-only).
  - Early computers discard the whole TLB; modern ones support the removal of individual TLB entries
  - If the pages are shared by processes..

```
108 static inline void
109 invlpg(void *addr)
110 {
111     asm volatile("invlpg (%0)" : : "r" (addr) : "memory");
112 }
```

- There is nothing to be done with permission addition (e.g., heap/stack extended, read-only to read-write). Why?

TLB Consistency

- Permission reduction: when a mapping is discarded or the access permission reduces (from read-write to read-only).
  - Early computers discard the whole TLB; modern ones support the removal of individual TLB entries
  - If the pages are shared by processes..

```
108 static inline void
109 invlpg(void *addr)
110 {
111     asm volatile("invlpg (%0)" : : "r" (addr) : "memory");
112 }
```

- There is nothing to be done with permission addition (e.g., heap/stack extended, read-only to read-write). Why?
- Can we do it in hardware instead of software?
  - The processor does not track the address where the mapping came from, so it cannot tell if a write to memory would affect a TLB entry
  - Even if it can, repeatedly checking each memory store to see if it affects any TLB entry is unnecessary

TLB Consistency

- TLB shootdown: on a multiprocessor, any processor changing their page table (and thus its TLB) needs to flush other processors' TLBs as well.
  - Multi-thread scenarios
  - Typically done through inter-processor interrupts

TLB Consistency

- TLB shootdown: on a multiprocessor, any processor changing their page table (and thus its TLB) needs to flush other processors' TLBs as well.
- The process
  - OS first modifies the page table
  - It sends a TLB flush request to all processors
  - Any processor that finishes its TLB update can resume
  - The original processor can resume only when all of the processors have acknowledged removing the old entry from their TLB. Why?

TLB Consistency

- TLB shootdown: on a multiprocessor, any processor changing their page table (and thus its TLB) needs to flush other processors' TLBs as well.
- The process
  - OS first modifies the page table
  - It sends a TLB flush request to all processors
  - Any processor that finishes its TLB update can resume
  - The original processor can resume only when all of the processors have acknowledged removing the old entry from their TLB. Why?
- High cost of TLB shootdown: linearly increases with core number
  - Optimization: batch the shootdown requests


RESEARCH-ARTICLE

Don't shoot down TLB shootdowns!

Authors: Nadav Amit, Amy Tai, Michael Wei Authors Info & Claims

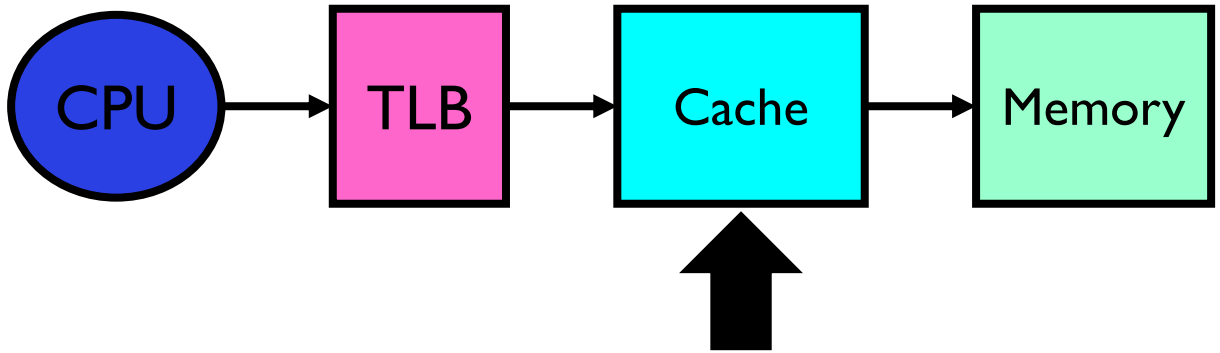
EuroSys '20: Proceedings of the Fifteenth European Conference on Computer Systems • April 2020 • Article No.: 35 • Pages 1–14 • https://doi.org/10.1145/3342195.3387518

Goals for Today

- Cache Concepts
- TLB
- Memory Cache

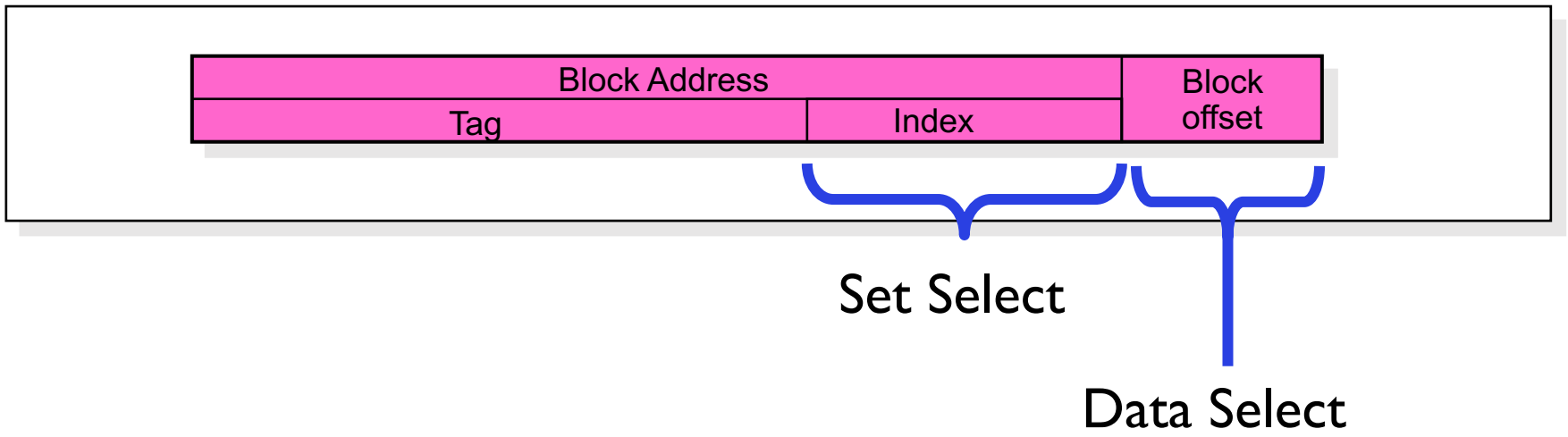
Memory Cache

- Fill the speed gap of CPU and DRAM memory



Memory Cache

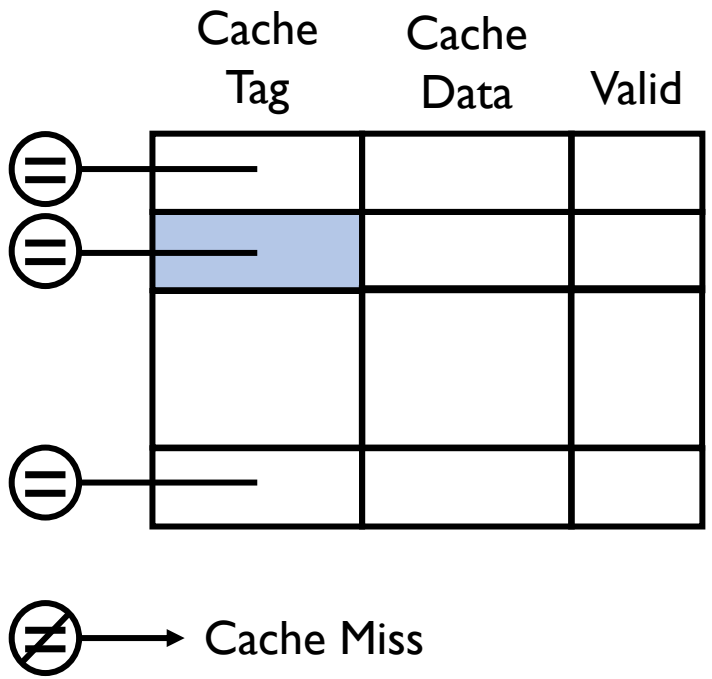
- Block (块) is the minimal unit of caching
  - Often larger than 1 word/byte to exploit the spatial locality
  - Shall not be neither too large or too small. Why?
  - Modern Intel processors use 64B
- Address fields for cache lookup



Cache Lookup

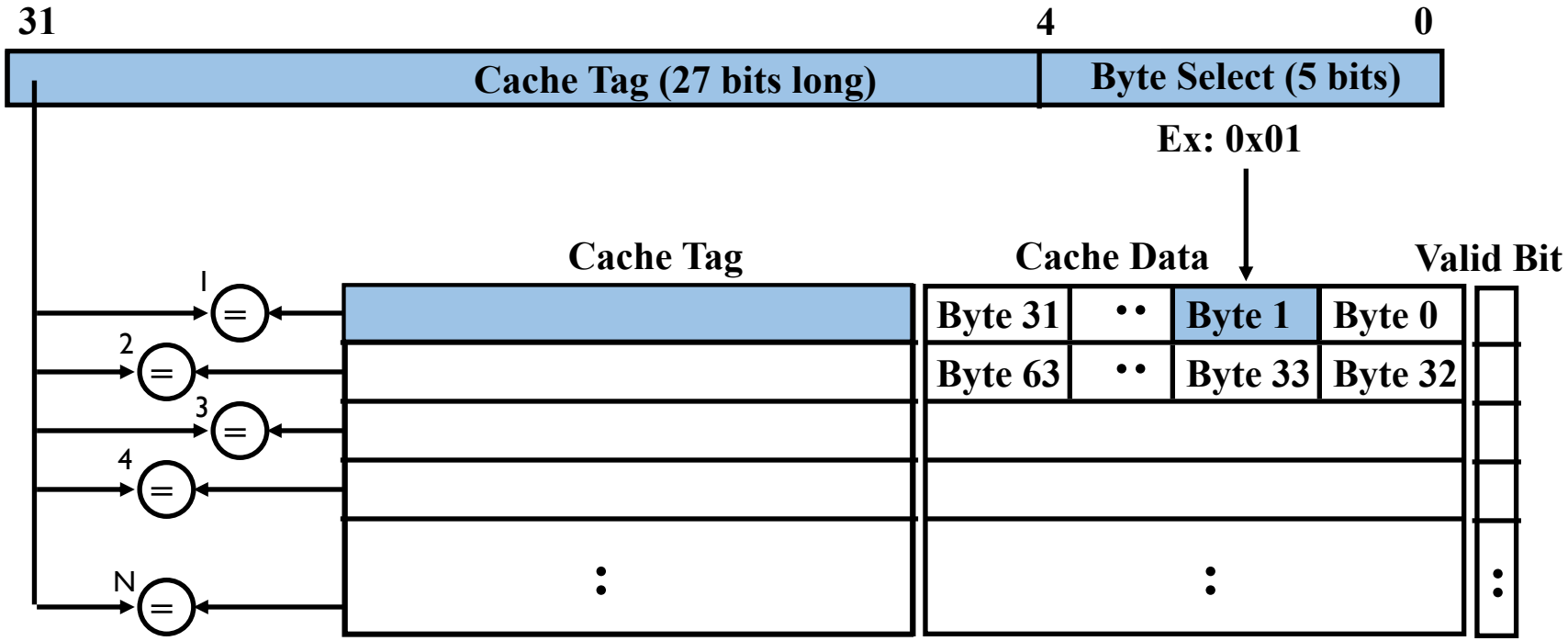
- Fully associative (全关联、完全关联): each address can be stored anywhere in the cache table
- Direct mapped (直接映射): each address can be stored in one location in the cache table
- N-way set associative (N路组关联): each address can be stored in one of N cache sets

- Tradeoffs: lookup speed and cache hit rate



Fully Associative

- Compare the cache tag on each cache line
- Example: Block Size=32B blocks
  - We need Nx 27-bit comparators



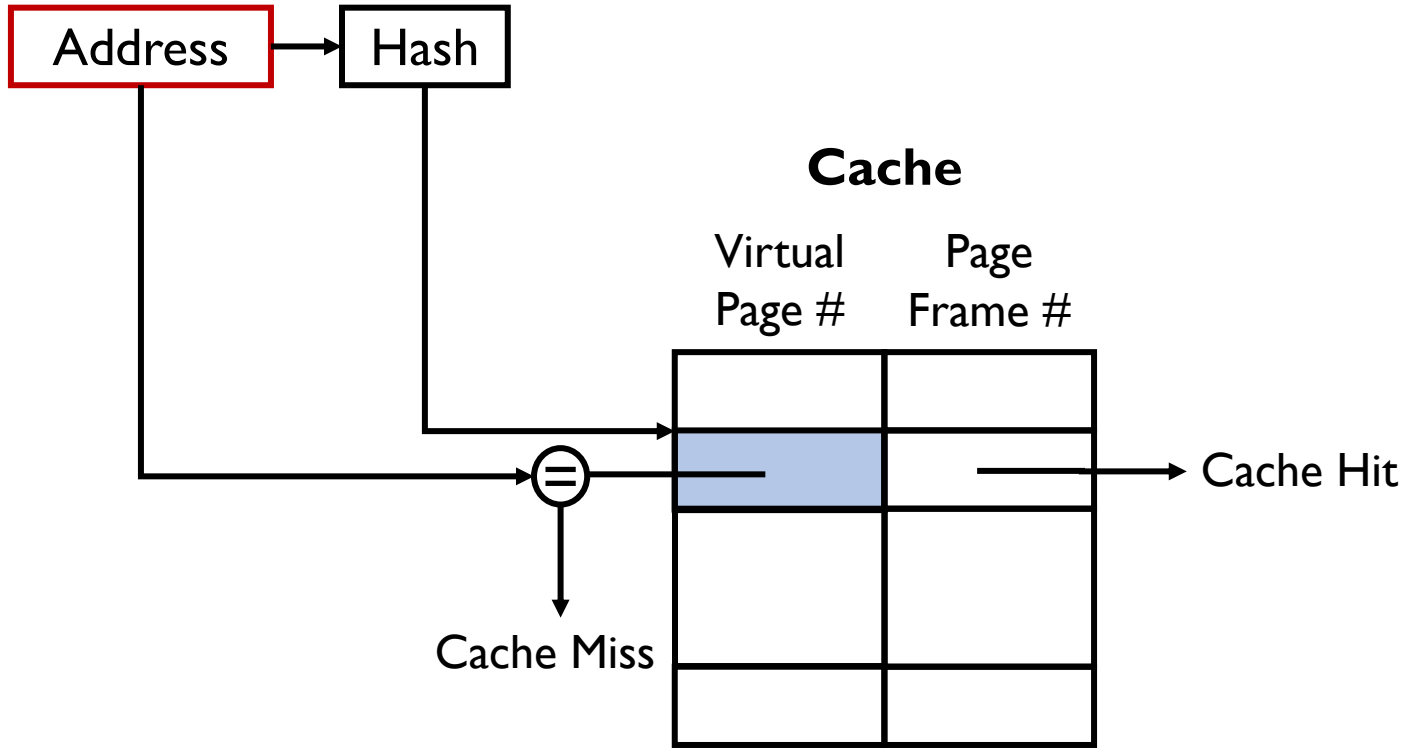
Fully Associative

- Compare the cache tag on each cache line
- Example: Block Size=32B blocks
  - We need Nx 27-bit comparators

- The drawback: performance degrades with larger cache, because there are more tags to be compared.
  - Solution #1: using larger block, but..

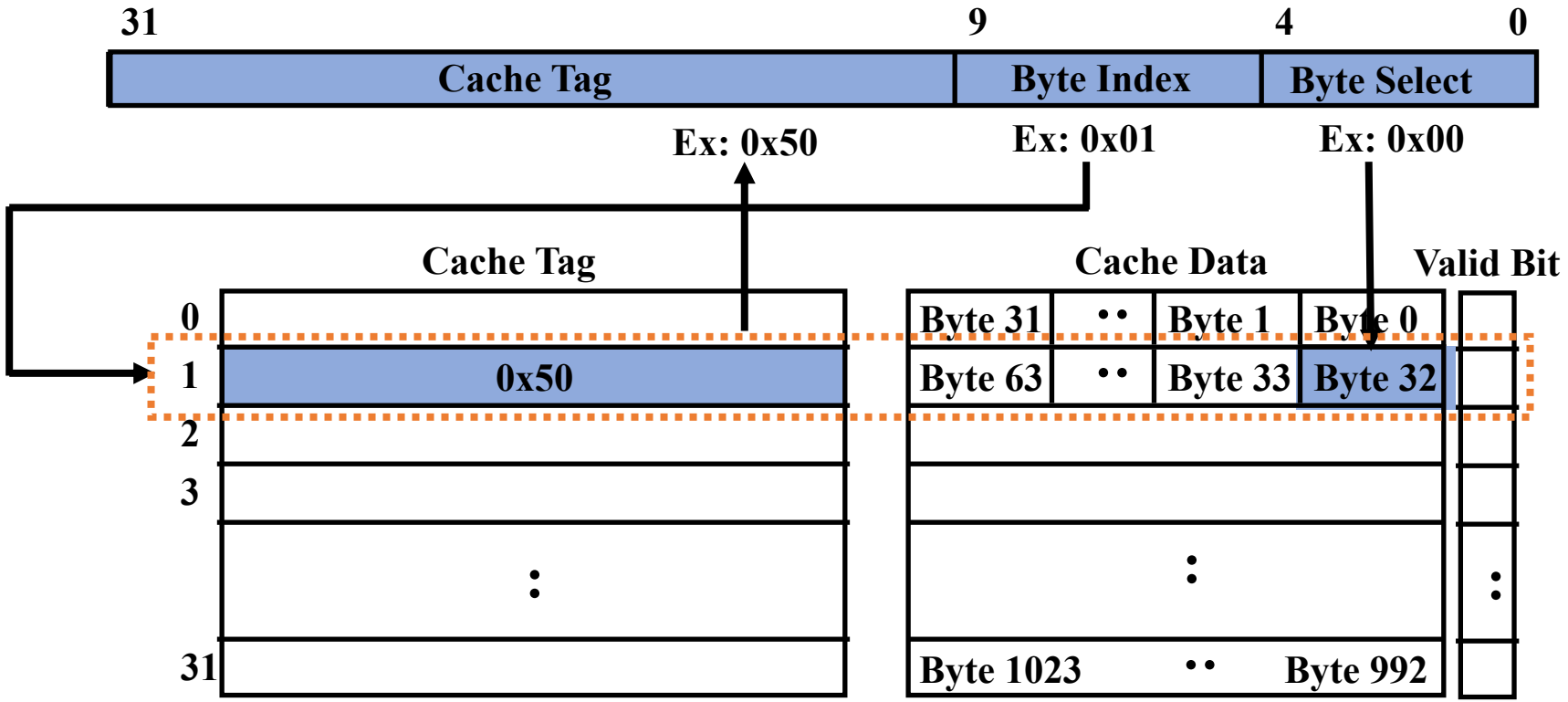
Direct Mapped

- Map to one specific cache line through a Hash function.
- Verify the address.



Direct Mapped

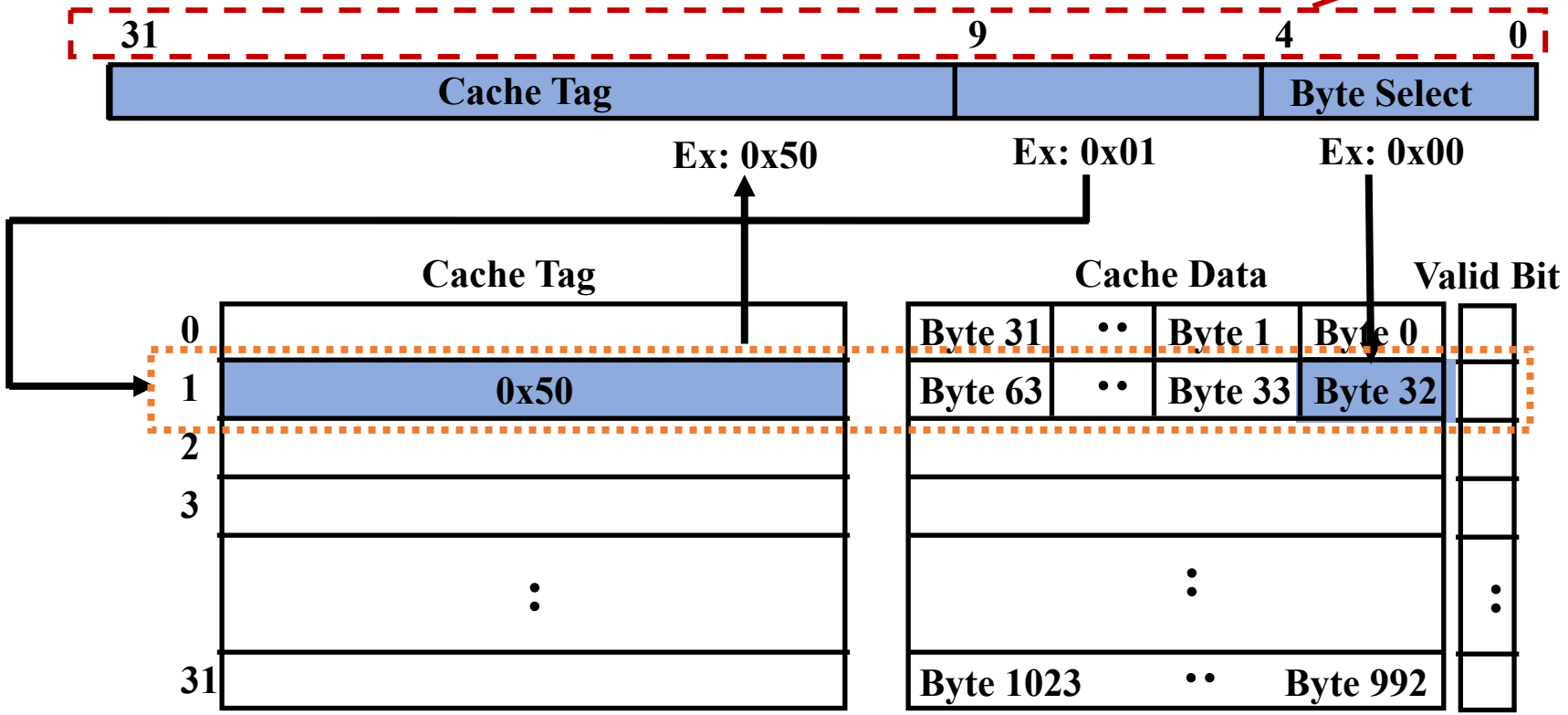
- Example: 1 KB Direct Mapped Cache with 32B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block



Direct Mapped

- Example: 1 KB Direct Mapped Cache with 32B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block

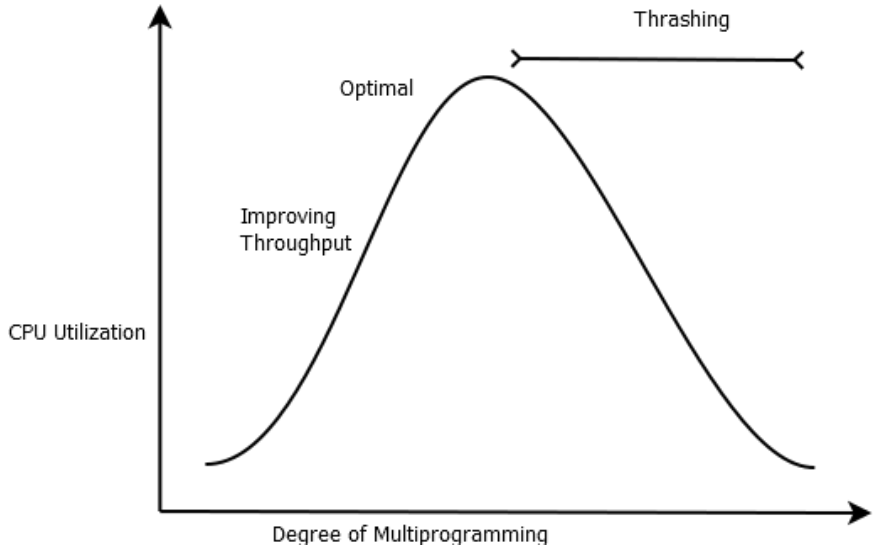
How those numbers are determined?



Direct Mapped

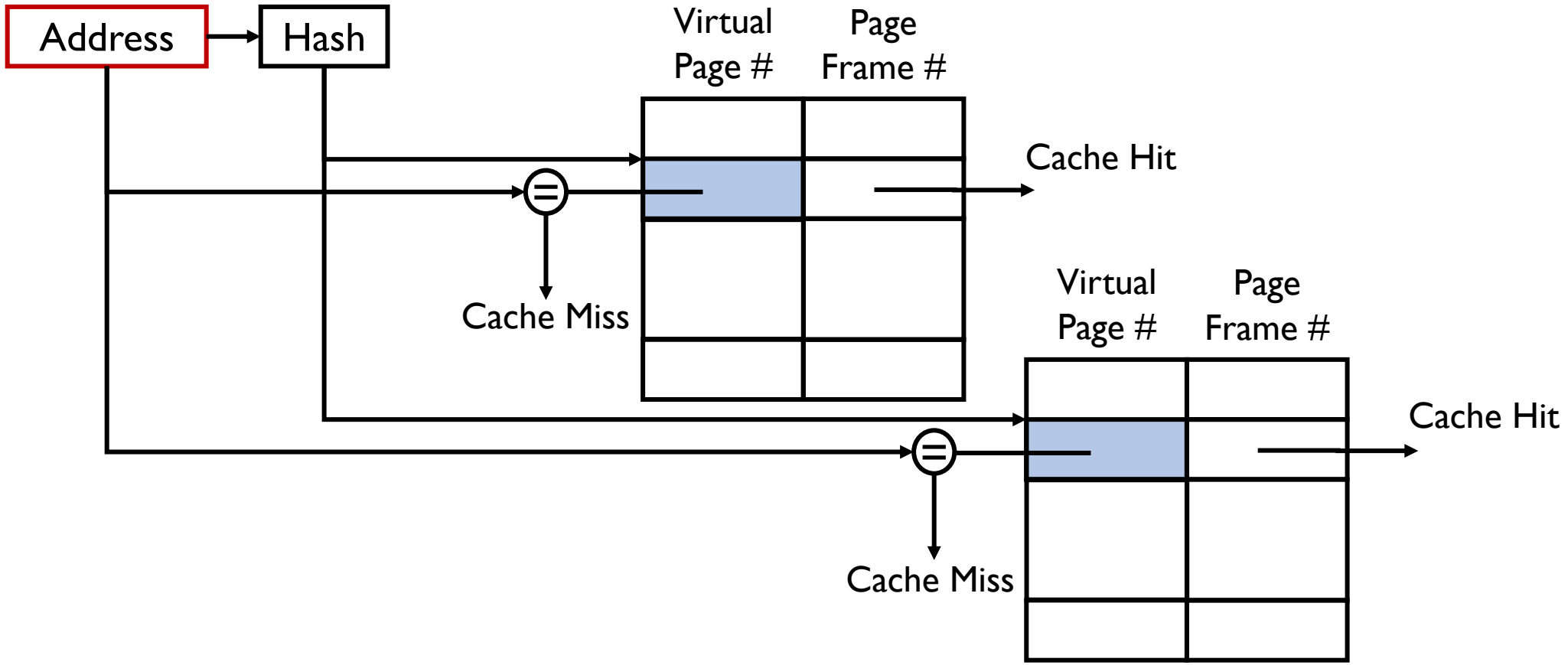
- Example: 1 KB Direct Mapped Cache with 32B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block

- The drawback: low flexibility
  - Thrash (颠簸): frequently using two addresses that map to the same cache entry.



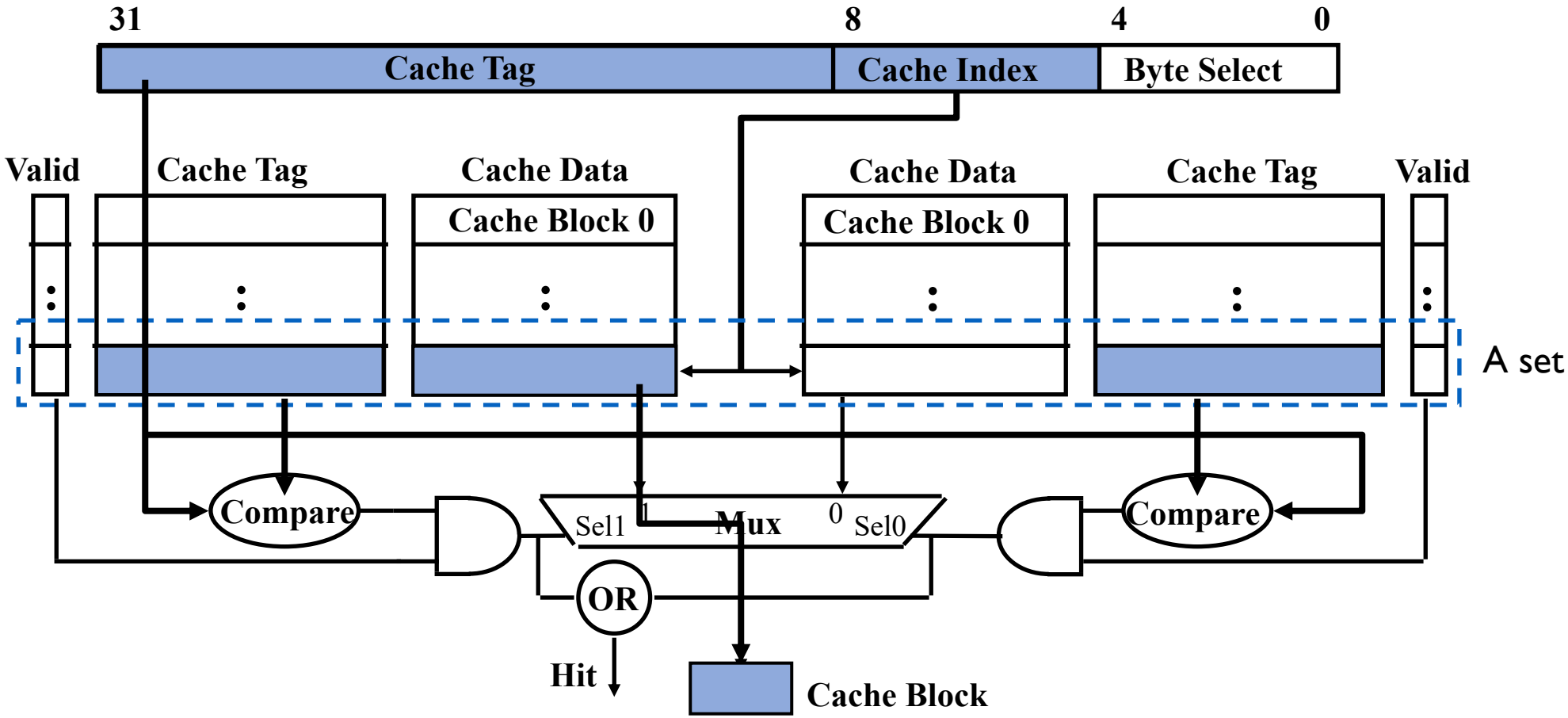
Set Associative

- N-way Set Associative: N entries per Cache Index
  - N direct mapped caches operates in parallel



Set Associative

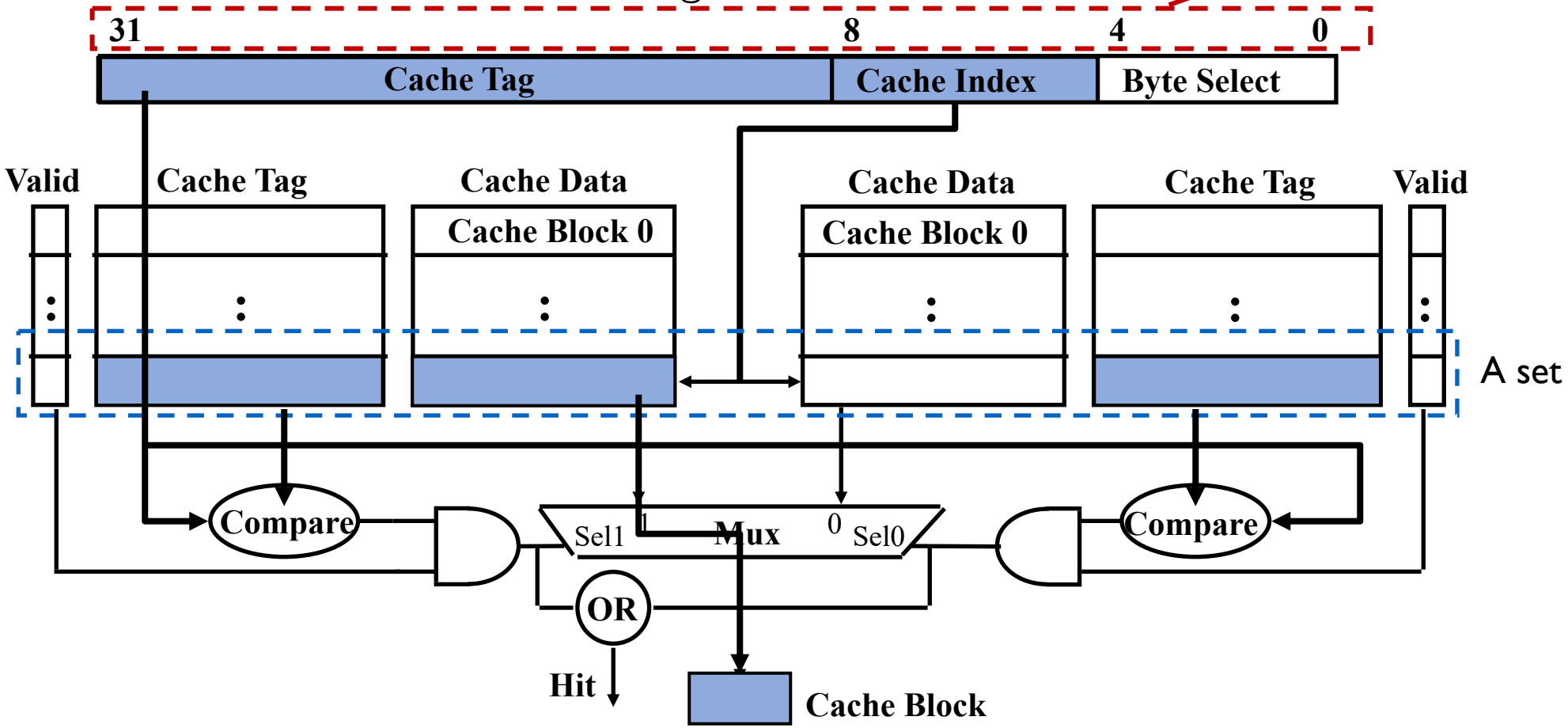
- Example: two-way set associative cache
  - Cache Index selects a "set" from the cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result



Set Associative

- 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

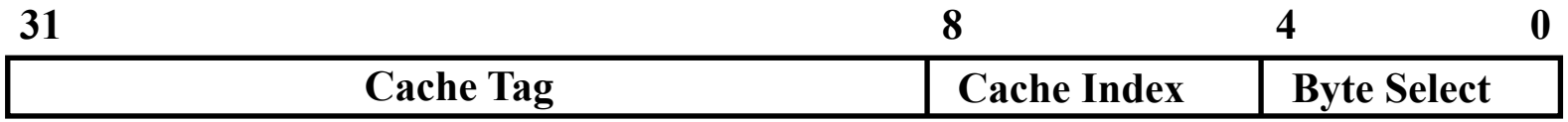
How those numbers are determined?



Set Associative

- 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

- N-way set associative is a mix of direct mapped and fully associative
  - When n = ? It becomes directed mapped
  - When n = ? It becomes fully associative

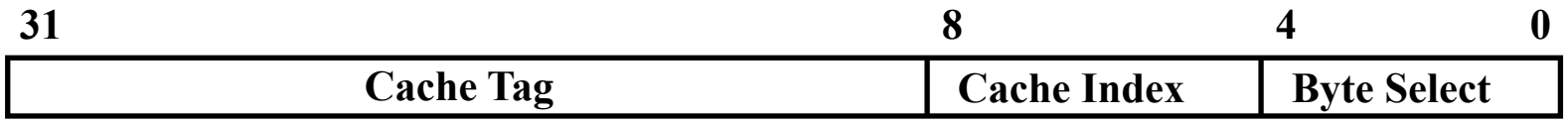


Set Associative

- 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

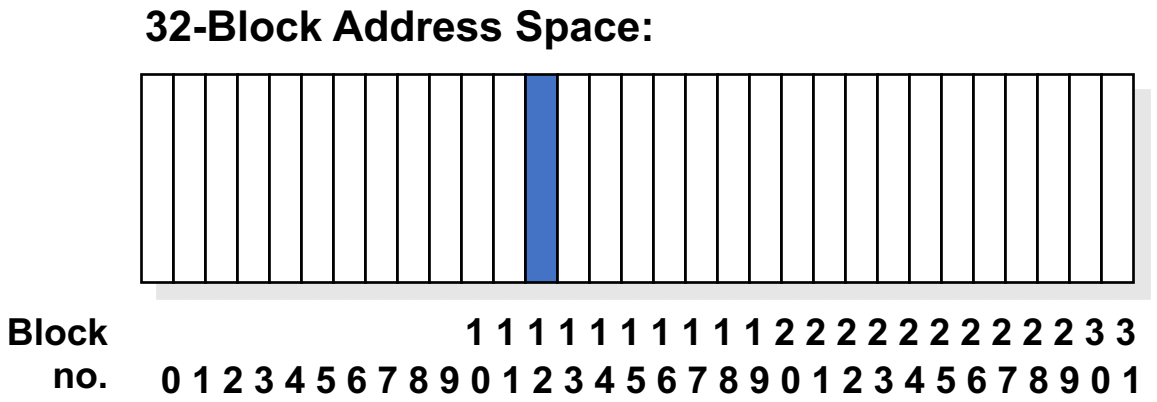
- N-way set associative is a mix of direct mapped and fully associative
  - When n = ? It becomes directed mapped
  - When n = ? It becomes fully associative

- Why use the lower bits for index, higher bits for tag?



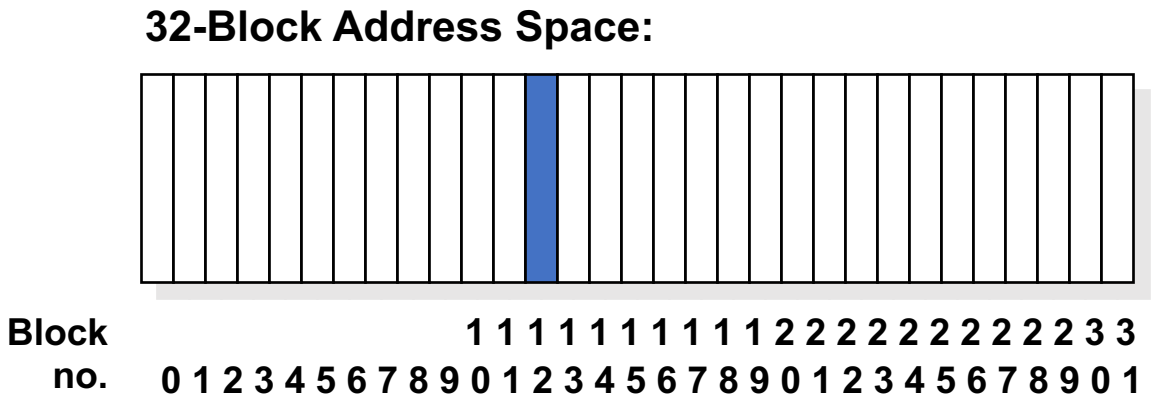
Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache??

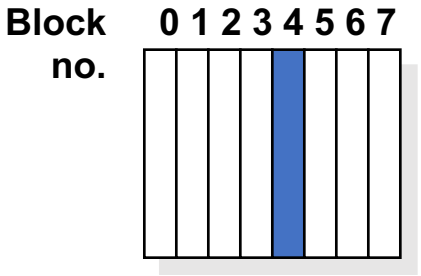


Where does a Block Get Placed in a Cache?

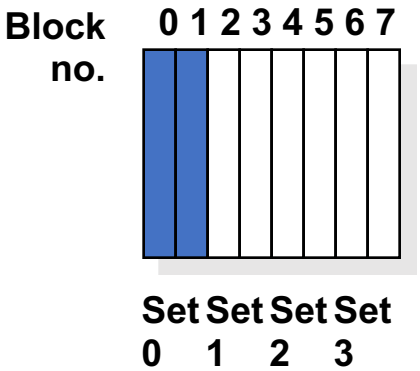
- Example: Block 12 placed in 8 block cache



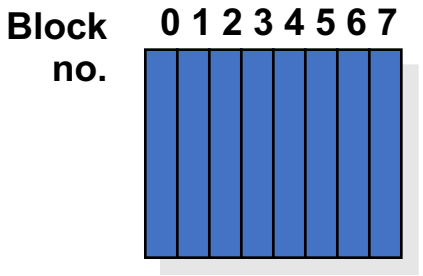
Direct mapped:
block 12 can go only into block 4 (12 mod 8)



Set associative:
block 12 can go anywhere in set 0 (12 mod 4)



Fully associative:
block 12 can go anywhere



Cache Replacement

- Direct mapped: only one possibility
- Set or fully associative
  - Random: sometimes simple is good; no extra overhead.

Cache Replacement

- Direct mapped: only one possibility
- Set or fully associative
  - Random: sometimes simple is good; no extra overhead.
  - First-In-First-Out (FIFO): could be the worst in certain workloads

FIFO															
Access	a	b	c	d	e	a	b	c	d	e	a	b	c	d	e
Block 1	a				e				d				c		
Block 2		b				a				e				d	
Block 3			c				b				a				e
Block 4				d				c				b			

Cache Replacement

- Direct mapped: only one possibility
- Set or fully associative
  - Random: sometimes simple is good; no extra overhead.
  - First-In-First-Out (FIFO): could be the worst in certain workloads
  - Least Recently Used (LRU): predicting future based on history

LRU															
Access	a	b	a	c	b	d	a	d	e	d	a	e	b	a	c
Block 1	a		✓				✓				✓			✓	
Block 2		b			✓								✓		
Block 3				c					e			✓			
Block 4						d		✓		✓					c

Cache Replacement

- Direct mapped: only one possibility
- Set or fully associative
  - Random: sometimes simple is good; no extra overhead.
  - First-In-First-Out (FIFO): could be the worst in certain workloads
  - Least Recently Used (LRU): predicting future based on history
  - Least Frequently Used (LFU)

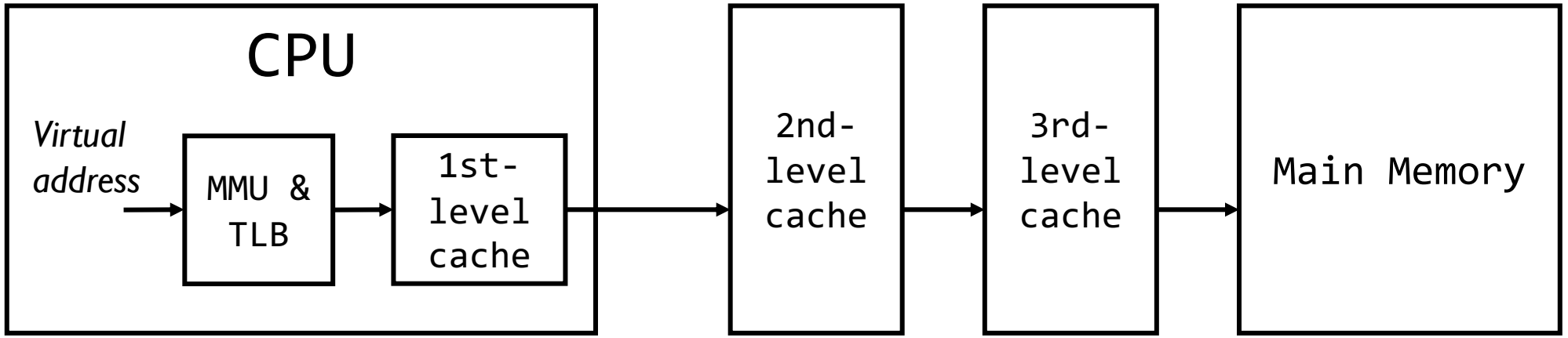
Cache Write Policies

- Write through: The information is written to both cache and memory
  - PRO: read misses cannot result in writes
  - CON: Processor is blocked on writes unless writes buffered

- Write back: The information is written only to cache
  - Modified cache block is written to main memory only when it is replaced
  - Question is block clean or dirty?
  - PRO: repeated writes not sent to DRAM; processor is not blocked on writes
  - CON: More complex; read miss may require writeback of dirty data; need a dirty bit to mark whether a block has been modified

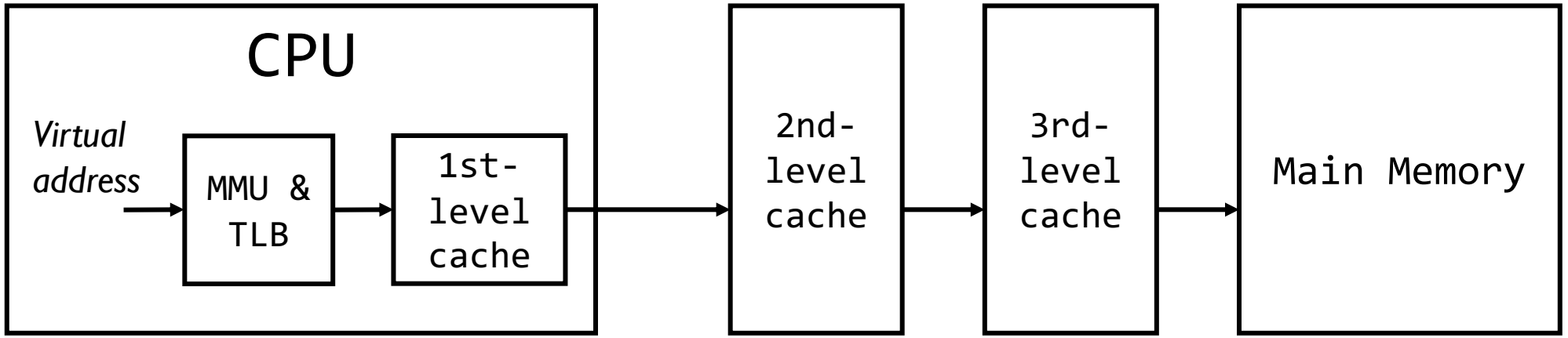
Addressed Virtually or Physically?

- The cache is addressed through virtual or physical address?
  - Note there are many levels of cache



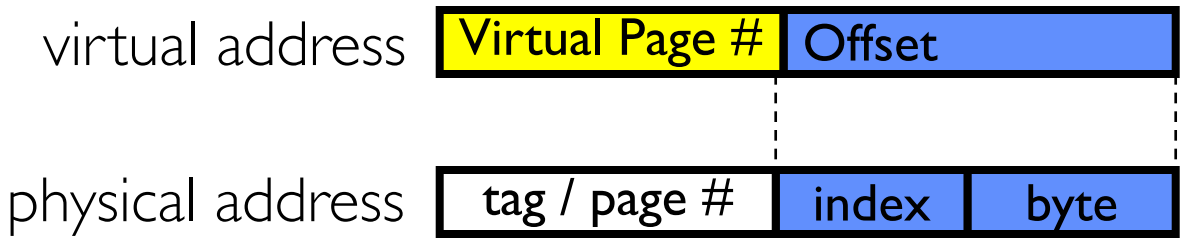
Addressed Virtually or Physically?

- The cache is addressed through virtual or physical address?
  - Note there are many levels of cache
- Every address access out of CPU is physical
  - The TLB miss cost is very high
  - Overlapping TLB and 1st-level cache as they are both in CPU



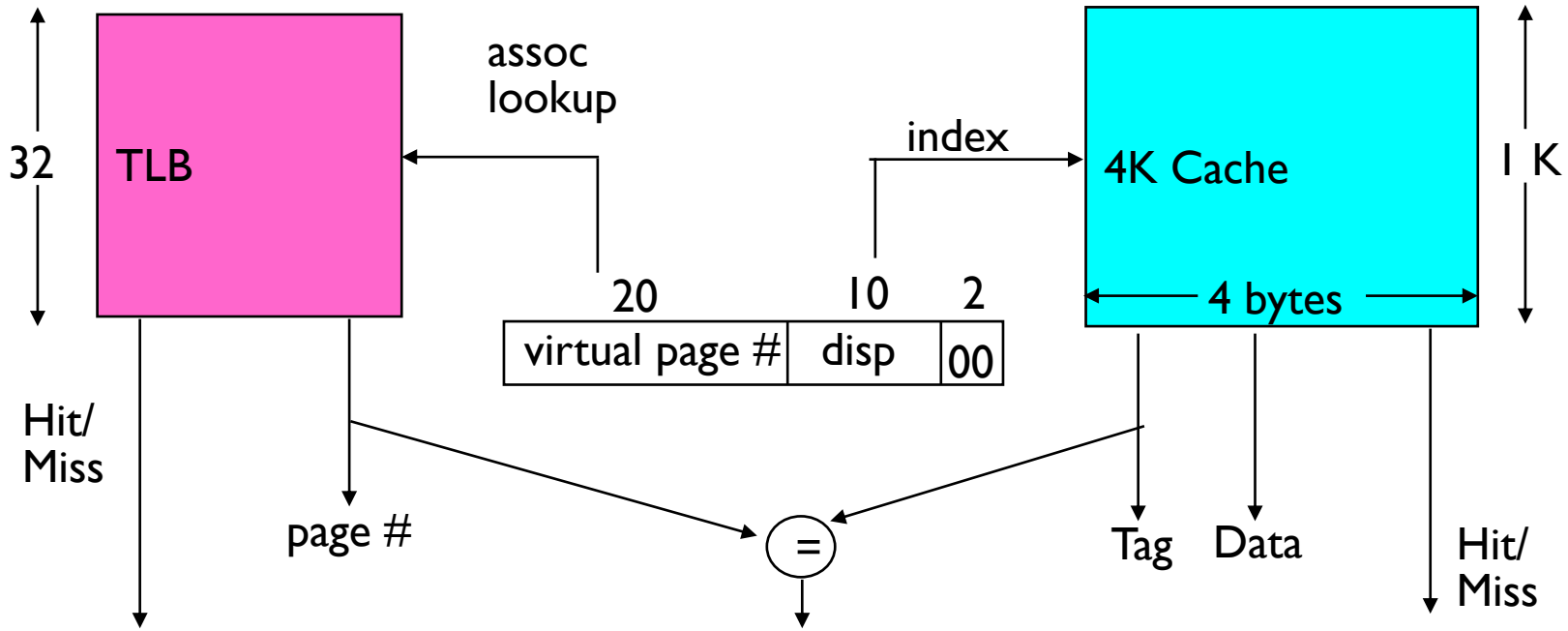
Overlapping TLB and Cache

- Key idea:
  - Offset in virtual address exactly covers the "cache index" and "byte select"
  - Thus can select the cached byte(s) in parallel to perform address translation



Overlapping TLB and Cache

- Key idea:
  - Offset in virtual address exactly covers the "cache index" and "byte select"
  - Thus can select the cached byte(s) in parallel to perform address translation



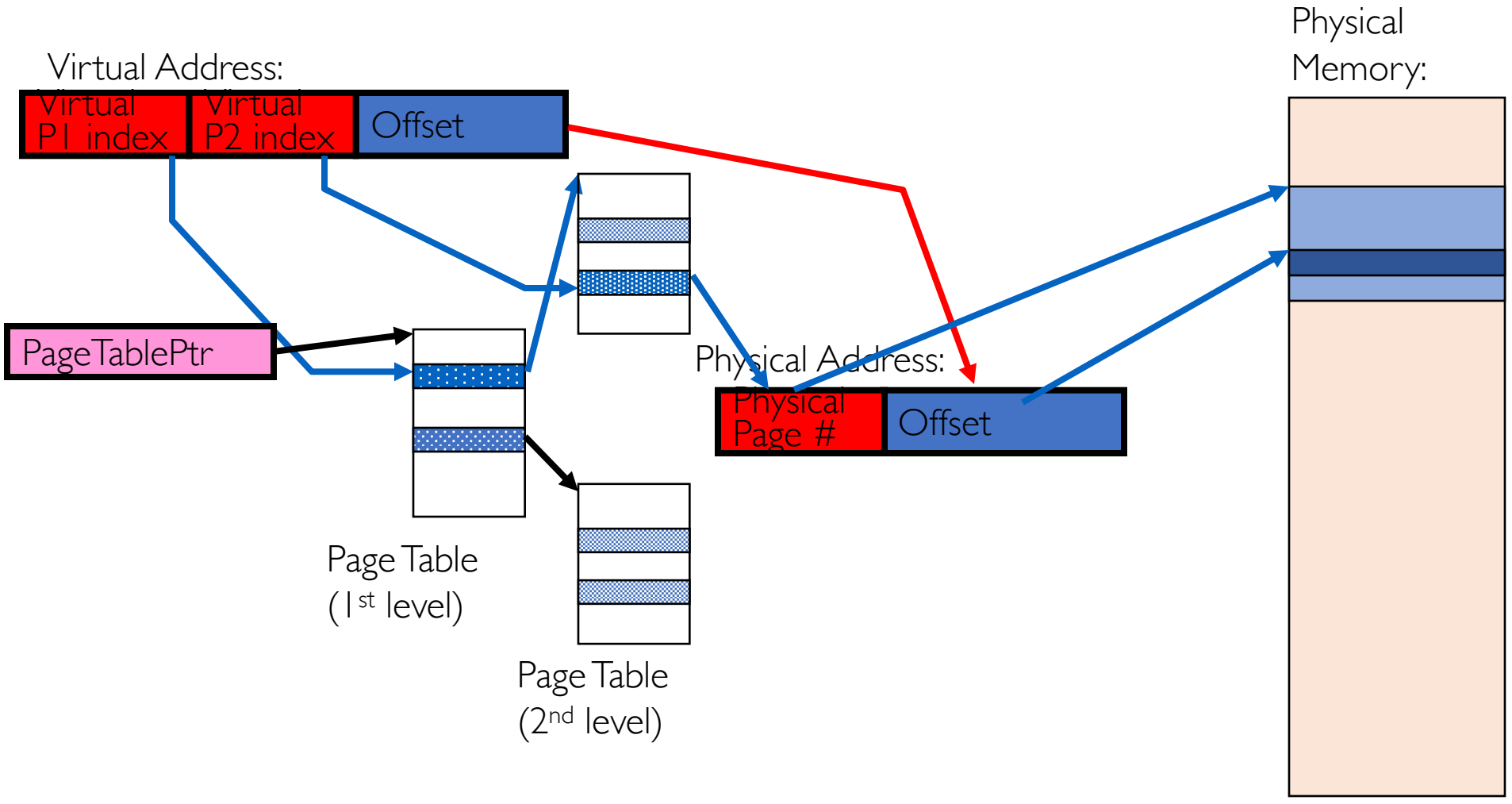
Overlapping TLB and Cache

- Key idea:
  - Offset in virtual address exactly covers the "cache index" and "byte select"
  - Thus can select the cached byte(s) in parallel to perform address translation
  - **"Virtually indexed, physically tagged" (VIPT)**

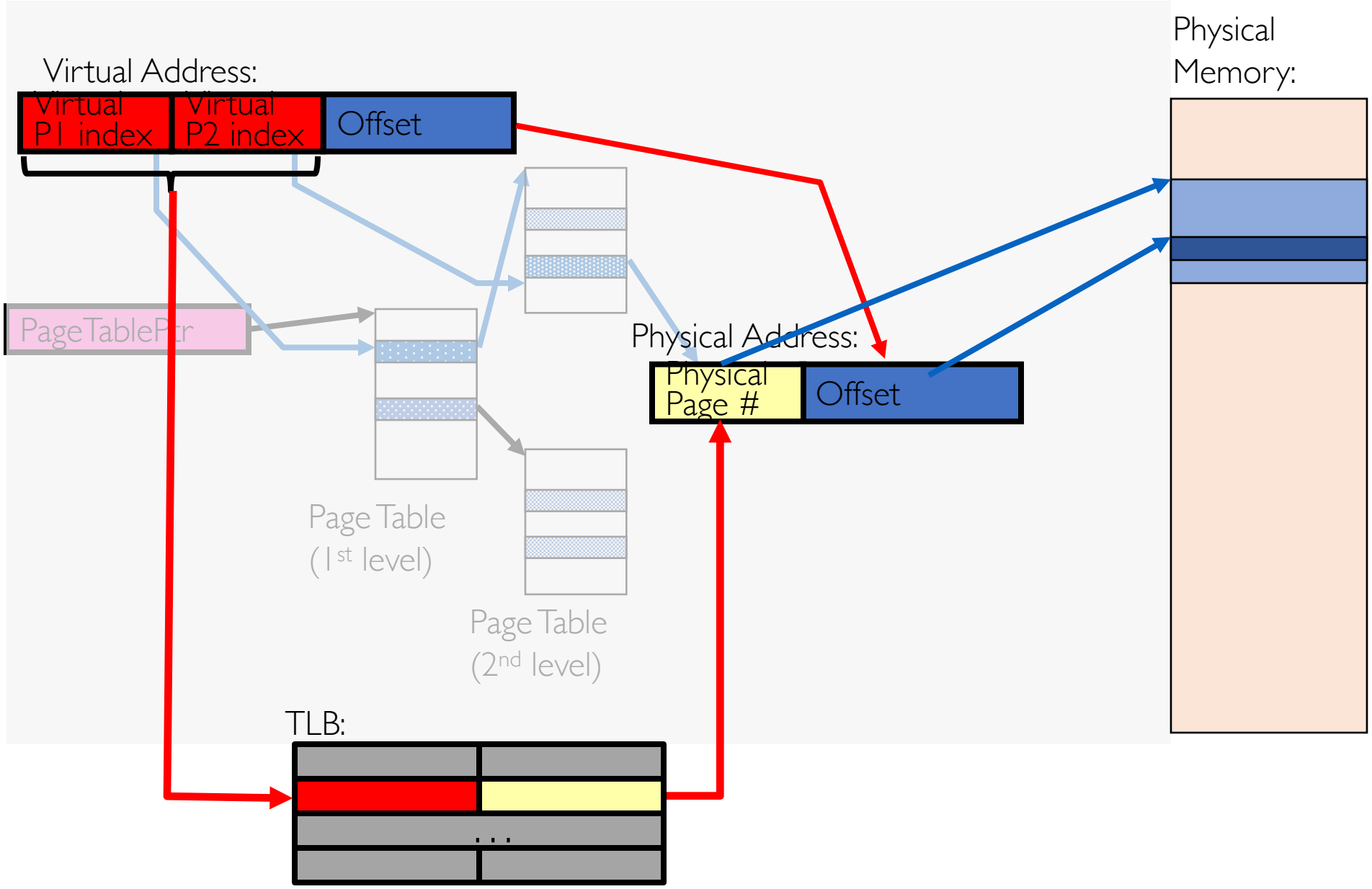
- Another option: virtually indexed, virtually tagged (VIVT)
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses
  - What's the problems?

- L1 is mostly VIPT, L2/L3 are mostly PIPT

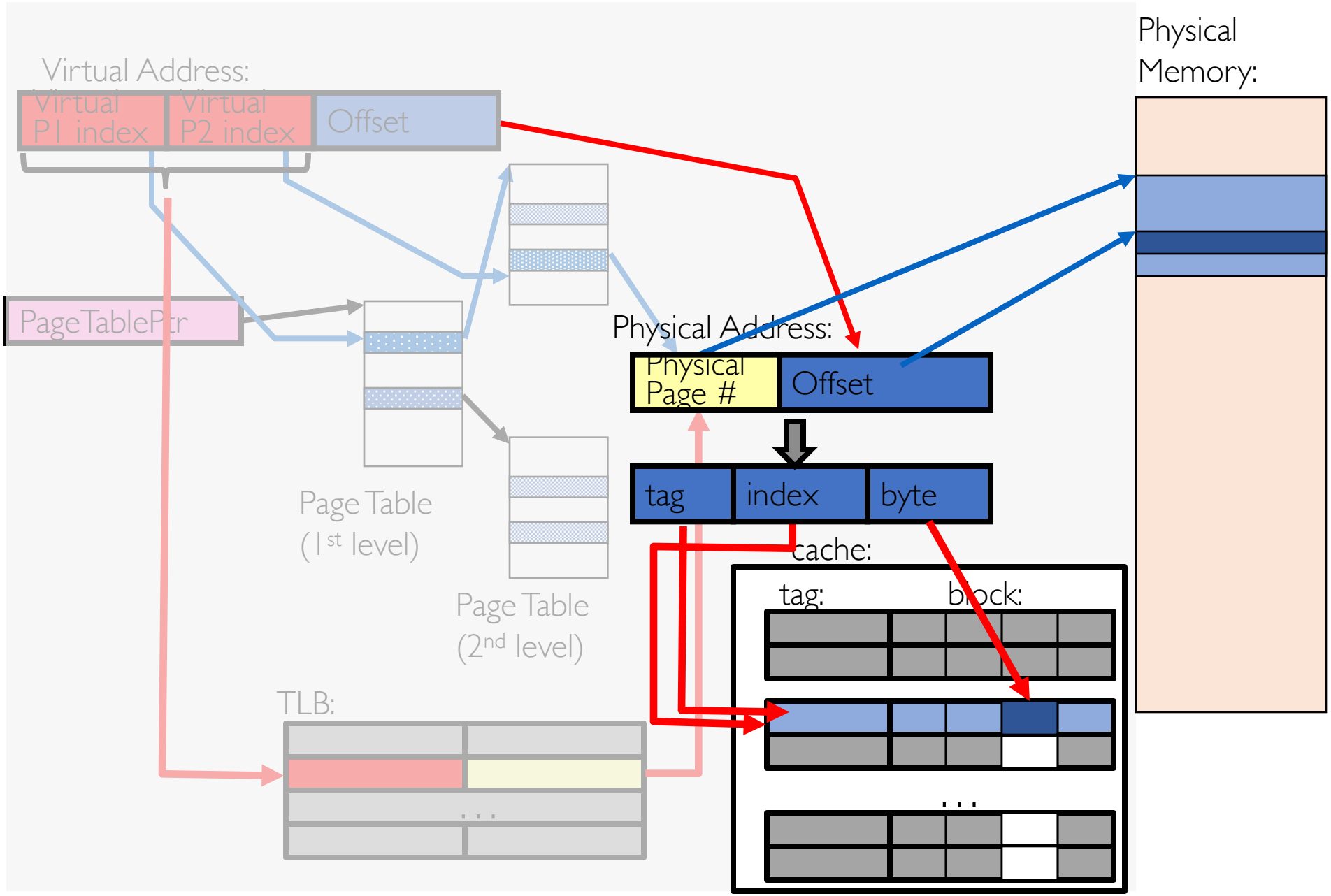
Putting Everything Together: Address Translation



Putting Everything Together: TLB

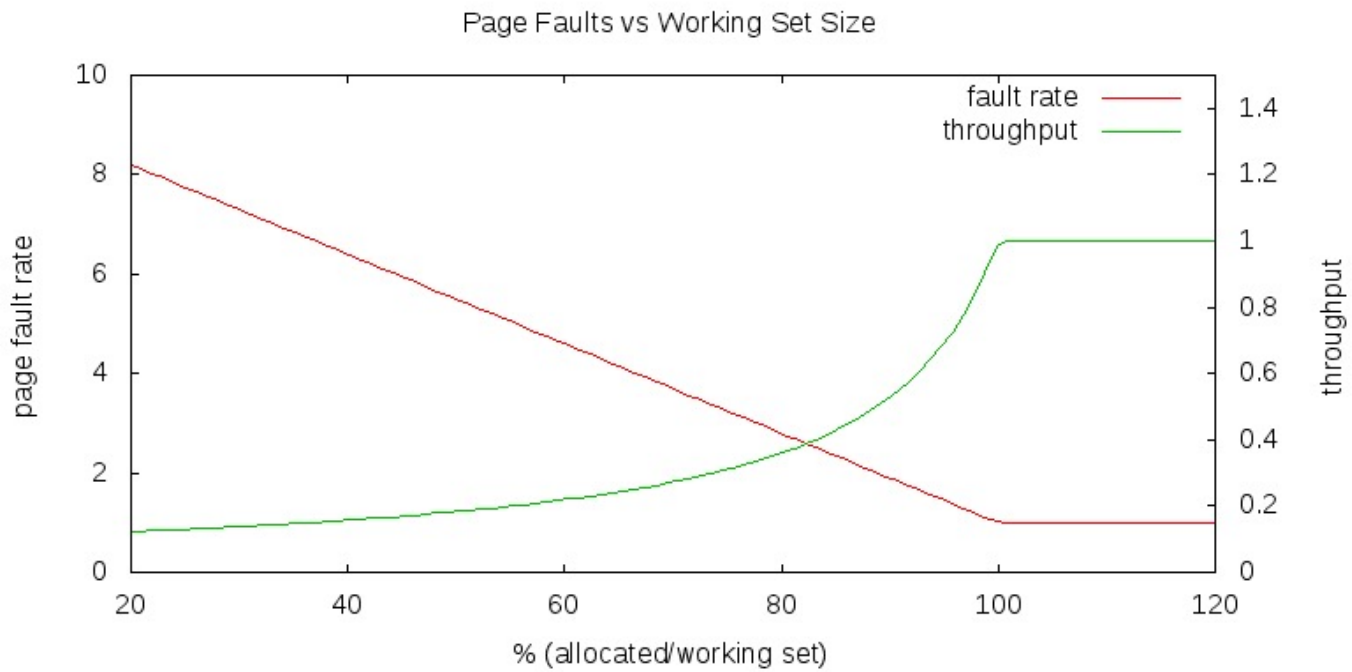


Putting Everything Together: Cache



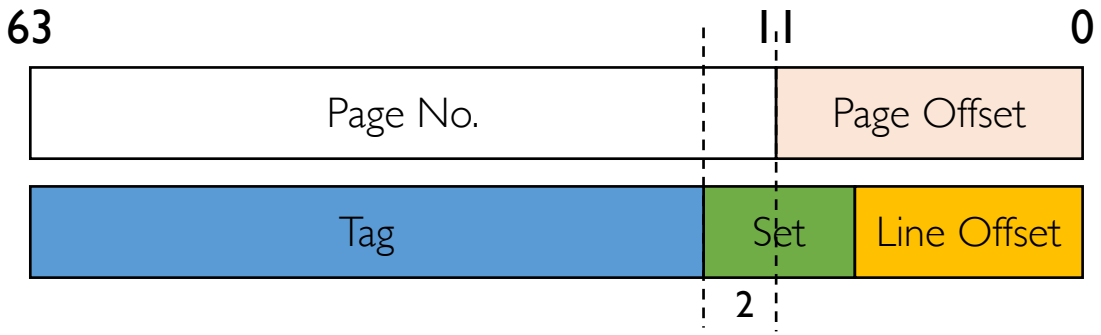
Making Cache Better Utilized

- Working Set (工作集): the memory needed by a program at a period
  - Could change at different phases
  - Better fit them into fast storage, e.g., first-level cache.



Page Coloring

- Page Coloring or Cache Coloring (着色) technique helps reduce the cache miss in an app

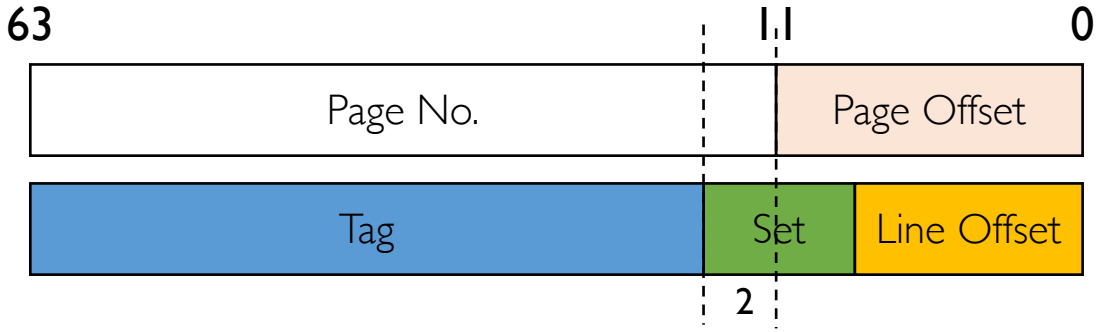


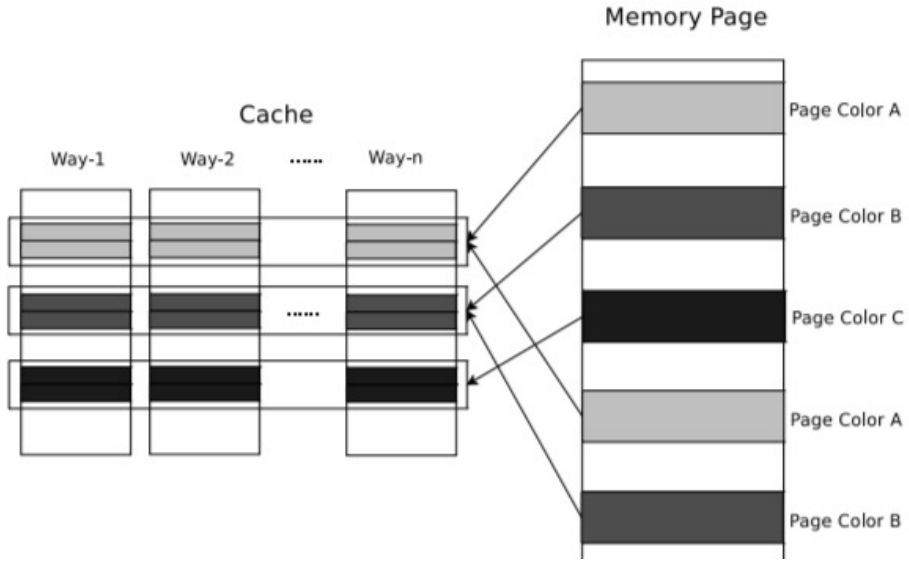
Consider two consecutive pages used by an application:

- Their virtual set number must be different
- But their physical set number could be the same after translation (when the OS maps them to the physical pages whose page numbers have the same last 2 bits). In such a case, two addresses with the same offset within these two pages will in contention for the cache set.

Page Coloring

- Page Coloring or Cache Coloring (着色) technique helps reduce the cache miss in an app





Consider two consecutive pages used by an application:

- Their virtual set number must be different
- But their physical set number could be the same after translation (when the OS maps them to the physical pages whose page numbers have the same last 2 bits). In such a case, two addresses with the same offset within these two pages will in contention for the cache set.

Solutions

- Coloring the physical pages with the cache sets
- Maps the application pages to as many colors as possible (so less contention)

Page Coloring

- Page Coloring or Cache Coloring (着色) technique helps reduce the cache miss in an app

- Page coloring only works for L2/L3 cache but not L1, why?
- Page coloring does not work for full-associative cache, why?

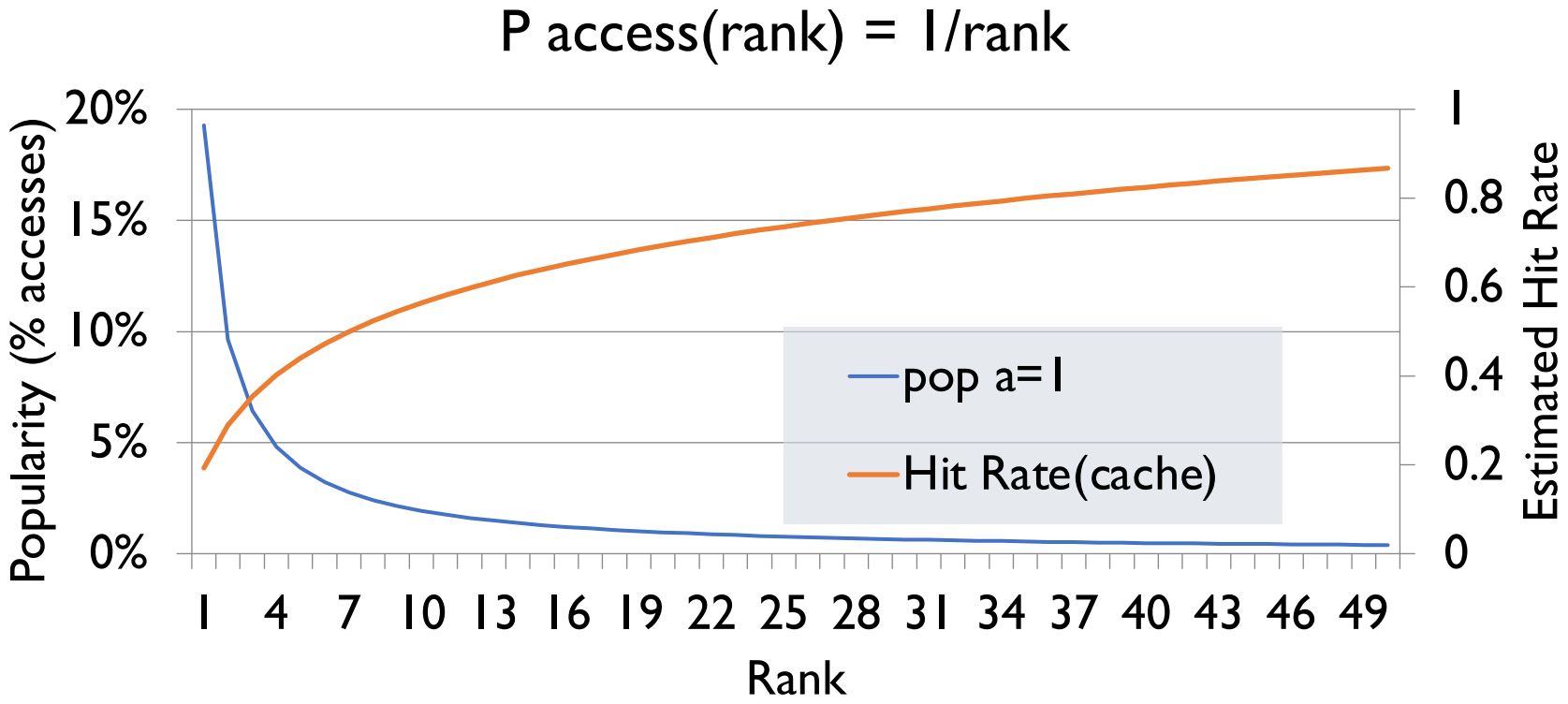
Making Cache Better Utilized

- Working Set (工作集): the memory needed by a program at a period
  - Could change at different phases
  - Better fit them into fast storage, e.g., first-level cache.

- It's important to design algorithms to adapt the working set to the memory hierarchy
  - Think of sorting a large array
  - Think of matrix multiplication (homework)

Making Cache Better Utilized

- Zipf model (齐普夫模型): the frequency of visit to the k-th most popular page $\propto \frac{1}{k^a}$, where a is a value between 1 and 2.
  - Heavy tail: a substantial portion of references will be to less popular ones



Cache Summary

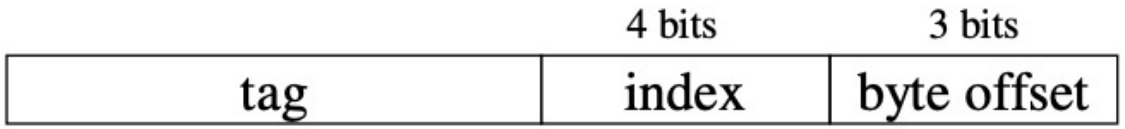
- Cache speeds up OS
  - TLB (cache of PTEs)
  - Paged virtual memory (memory as cache for disk)
  - File systems (cache disk blocks in memory)
  - DNS (cache hostname => IP address translations)
  - Web proxies (cache recently accessed pages)
- Cache complicates OS
  - E.g., TLB consistency

Homework

- Describe what is TLB shootdown, why we need it. Then search and discuss at least one optimization on TLB shootdown performance.
- Accelerate your matrix multiplication by higher cache hit ratio.
- Some exercises followed up

Exercise- I

Suppose a cache divides addresses as follows:



Fill in the values for a direct-mapped or 4-way associative cache:

	Direct-mapped	4-way associative
Block size		
Number of blocks		
Total size of cache (e.g. 32 * 128 – don't have to multiply out)		
Tag size (# bits)		

Exercise-2

1. Suppose cache has:
   - 4 byte blocks
   - 128 blocks

   Show how to break the following address into the tag, index, & byte offset.

   0000 1000 0101 1100 0001 0001 0111 1001

2. Same cache, but now 8-way associative. How does this change things?

   0000 1000 0101 1100 0001 0001 0111 1001

Exercise-3

- Given a cache that is:
  - 4-way associative
  - 32 blocks
  - 16 byte block size

  What is the cache index and byte offset for the following address:

  `0x3ab12395`

  Cache index =
  Byte offset =

  And this one:

  `0x70ff1213`

  Cache index =
  Byte offset =

Do these addresses conflict in the cache?

Exercise-4

Suppose a 32-bit address is divided up as follows for caching:

- **6 bits – byte offset**
- **5 bits – index**
- **21 bits – tag**

Fill in the following table for the two given types of caches

	Direct-mapped	2-way associative
Block size		
Number of blocks		
Total cache size		

Exercise-5

Suppose a direct-mapped cache has 16 byte blocks and a total of 128 blocks (N=128). The machine has 64 bit addresses.

1. How many address bits are used for the byte offset?

2. How many address bits are used for the index?

3. How many address bits are used for the tag?

Now suppose the cache is 4-way set associative. Answer again:

1. How many address bits are used for the byte offset?

2. How many address bits are used for the index?

3. How many address bits are used for the tag?