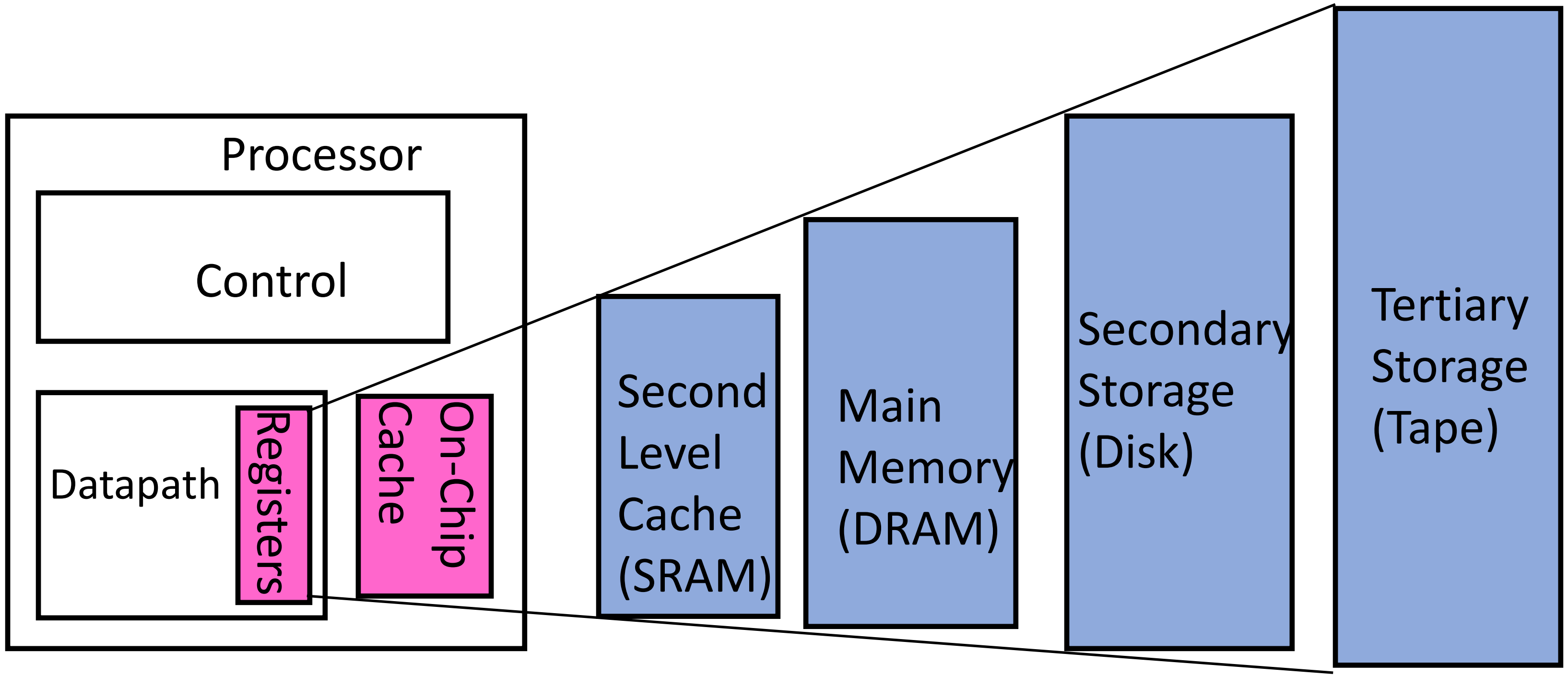
Operating Systems
Lecture 8

Demand Paging

Prof. Mengwei Xu

Recap: Memory Hierarchy

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



Speed (ns): 1s 10s-100s 100s 10,000,000s (10s ms) 10,000,000,000s (10s sec)

Size (bytes): 100s Ks-Ms Ms-Gs Gs-Ts Ts-Ps

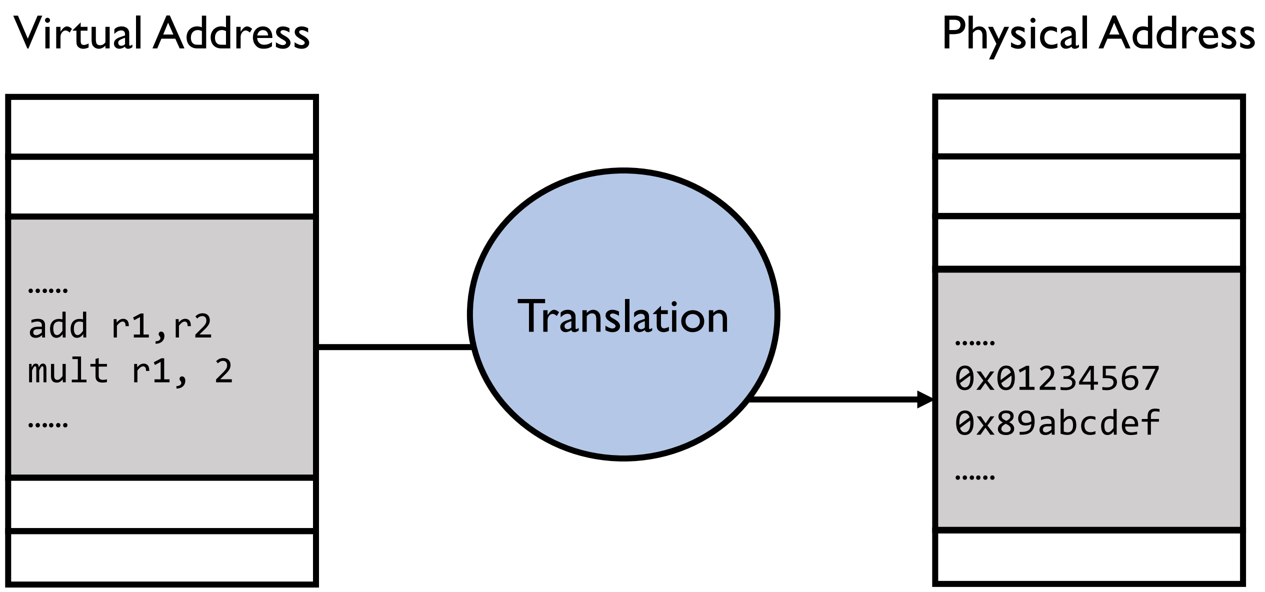
Recap: Locality

- 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

Recap: TLB as a Cache

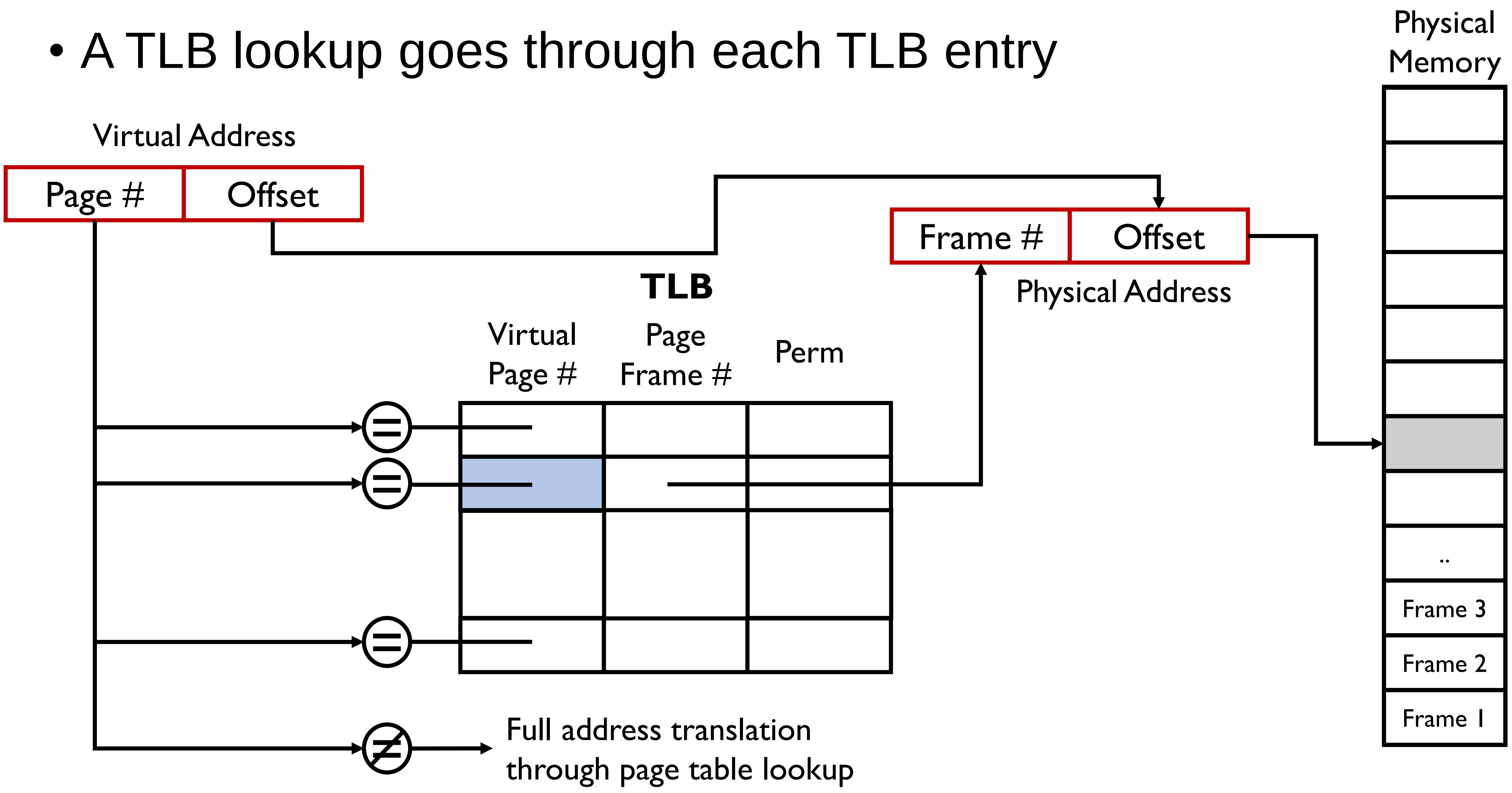
- Translation Lookaside Buffers (TLB, 转换检测缓冲区): a special cache within MMU that accelerates address translation

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



Recap: TLB Lookup

- A TLB lookup goes through each TLB entry

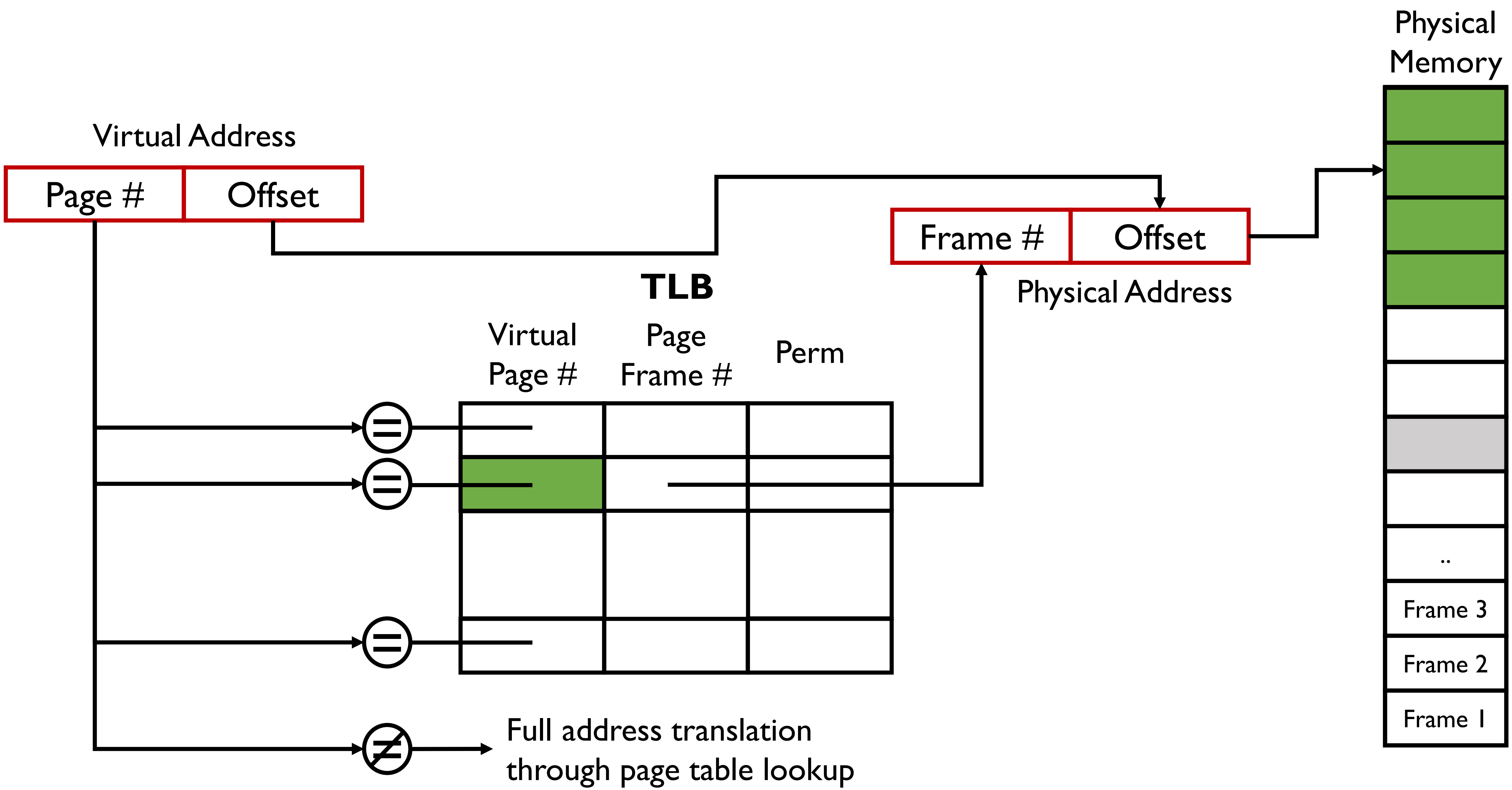


Recap: TLB Miss

- (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

Recap: Superpage

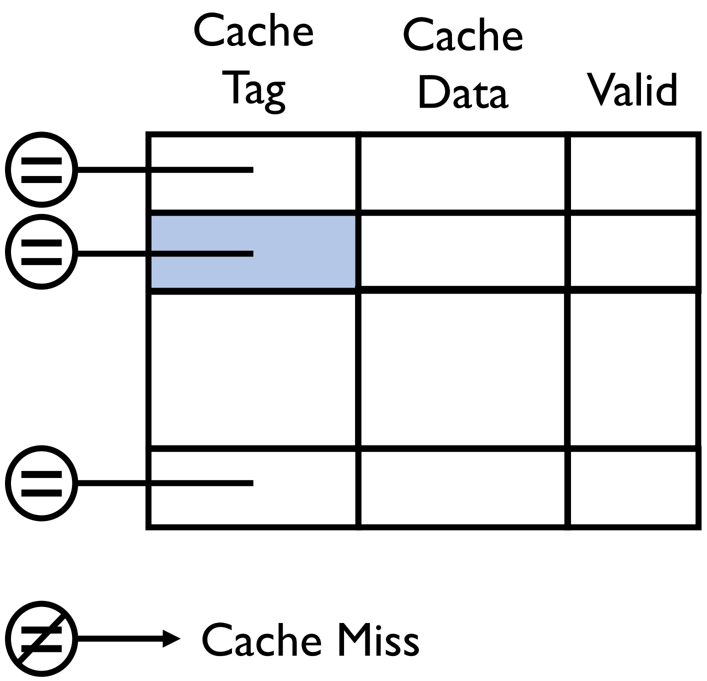


Recap: TLB Consistency

- 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

Recap: Cache Lookup

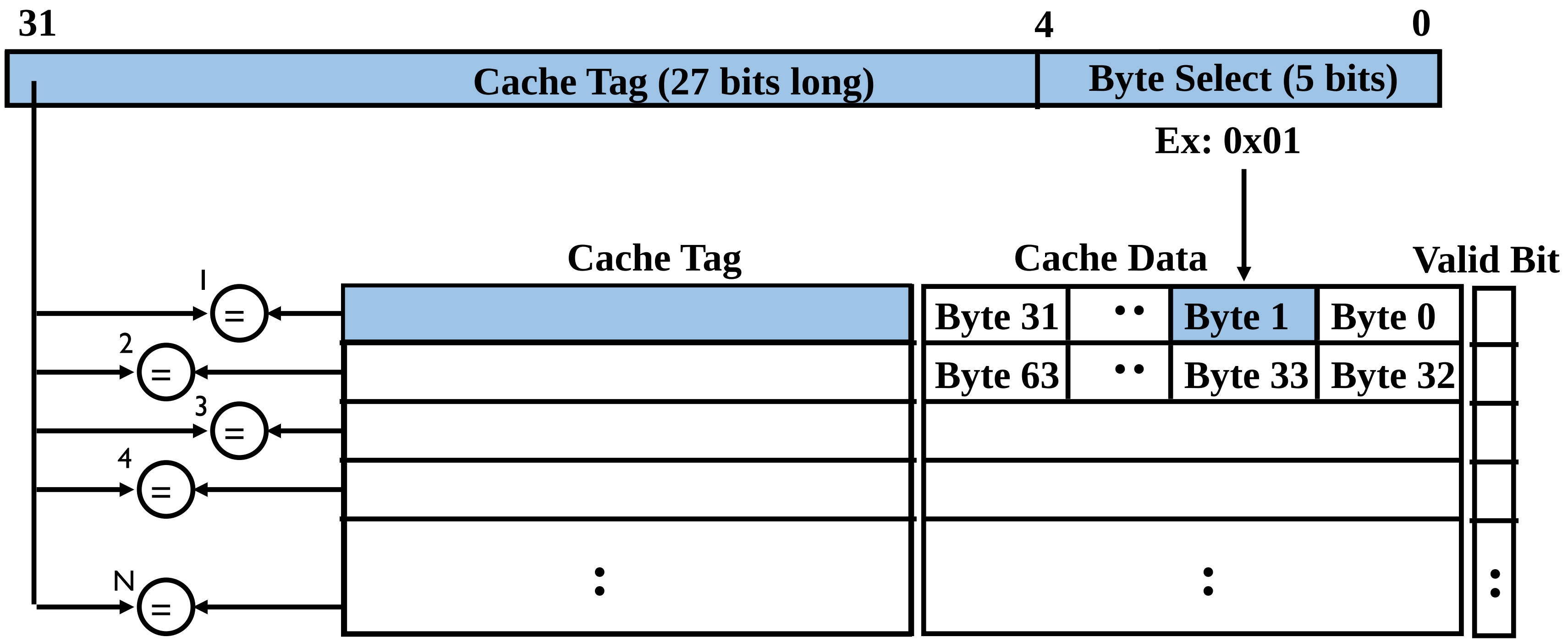
- Fully associative (全关联、完全关联): each address can be stored anywhere in the cache table
- Direct mapped (直接映射): each address can be stored in one location in the cache table
- N-way set associative (N路组关联): each address can be stored in one of N cache sets



- Tradeoffs: lookup speed and cache hit rate

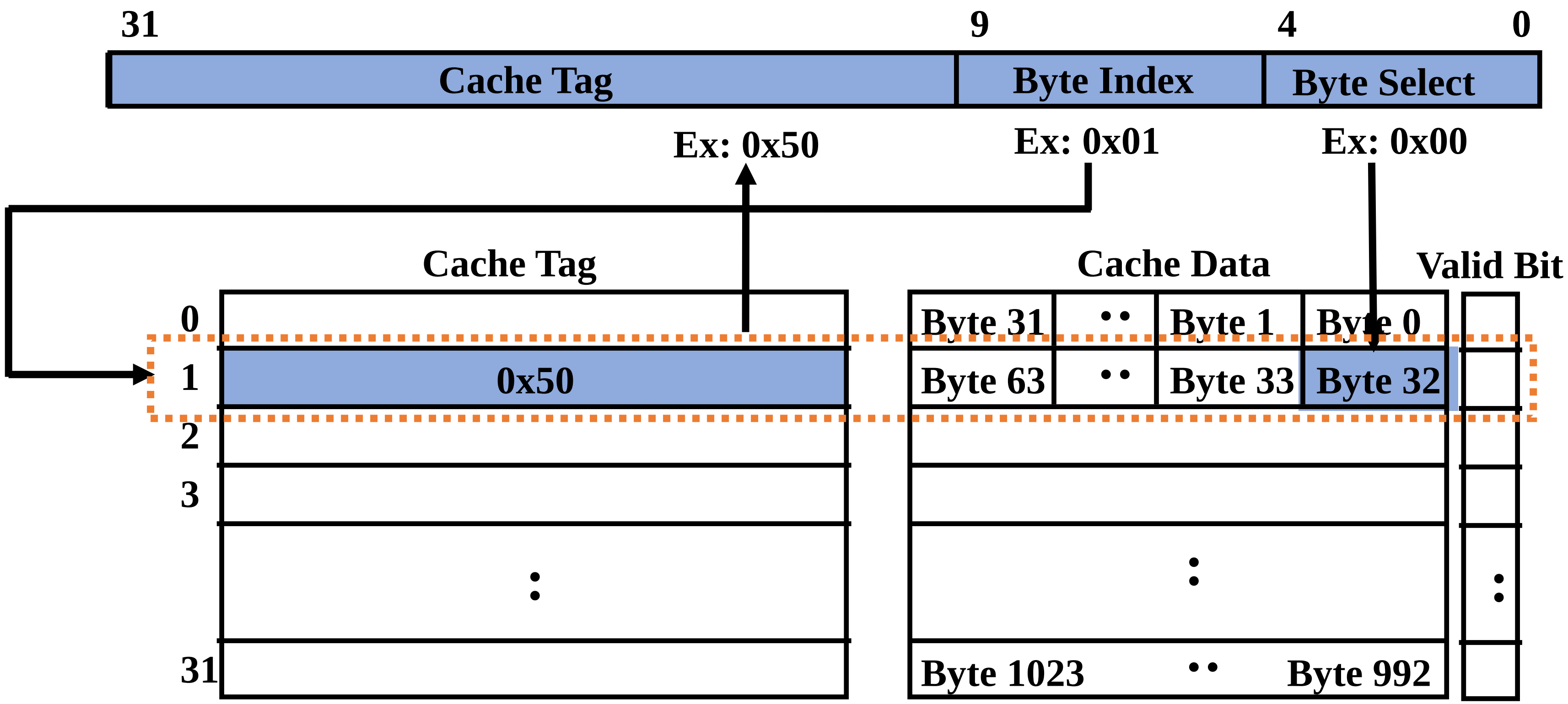
Recap: Fully Associative

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



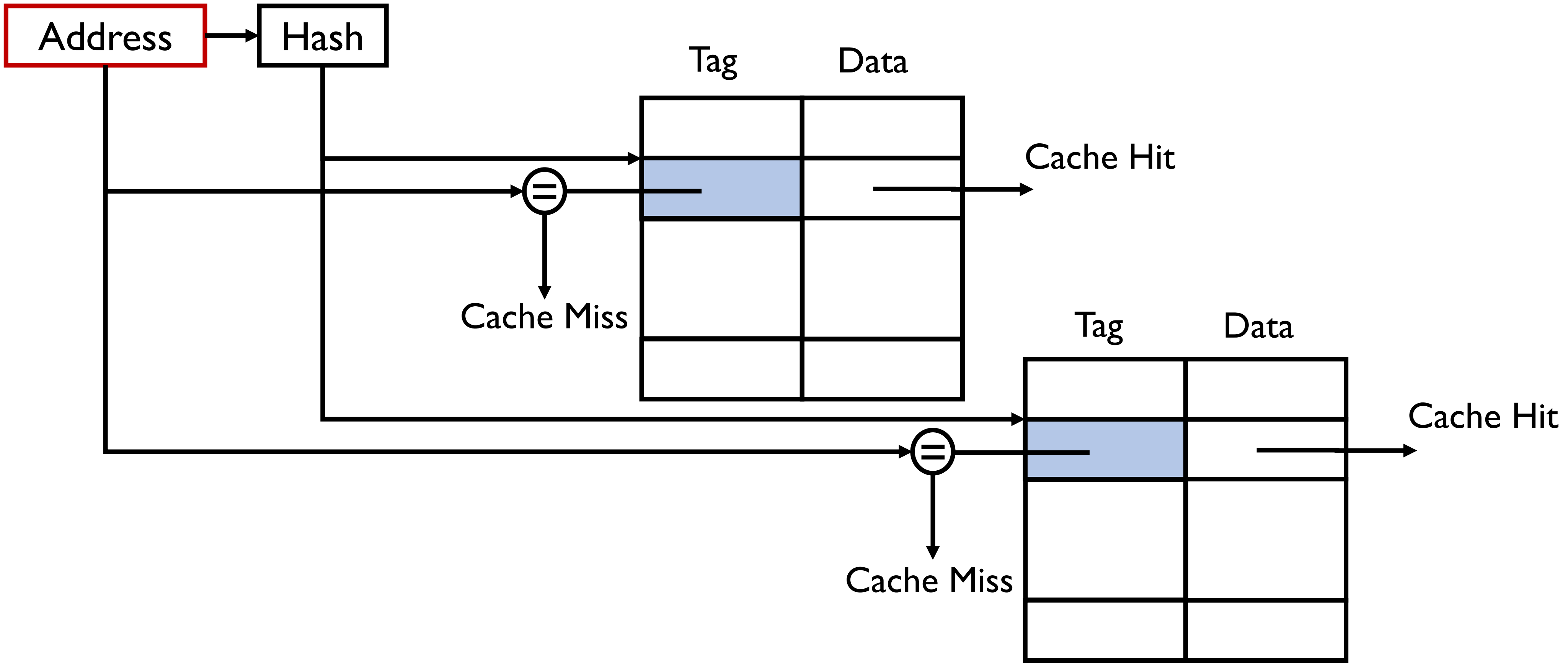
Recap: Direct Mapped

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



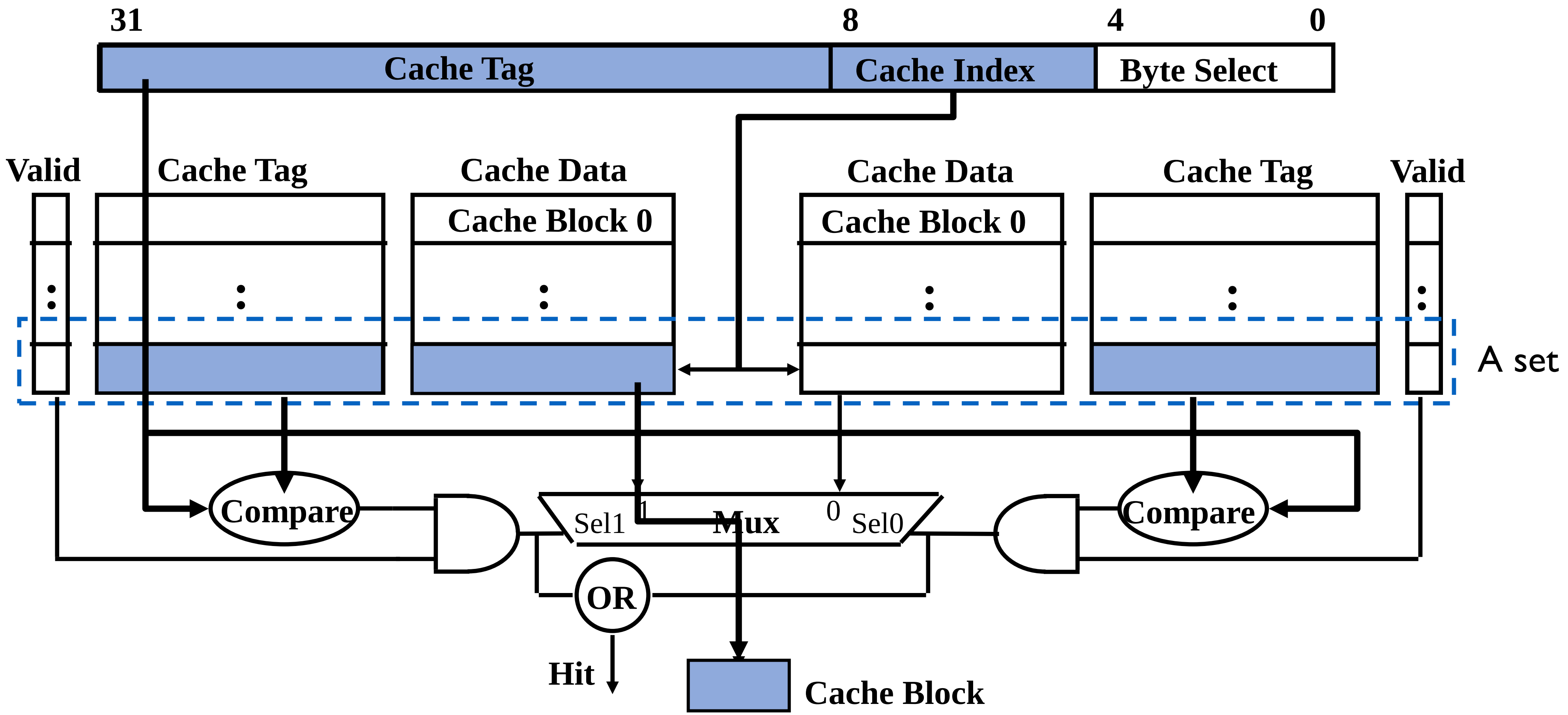
Recap: Set Associative

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



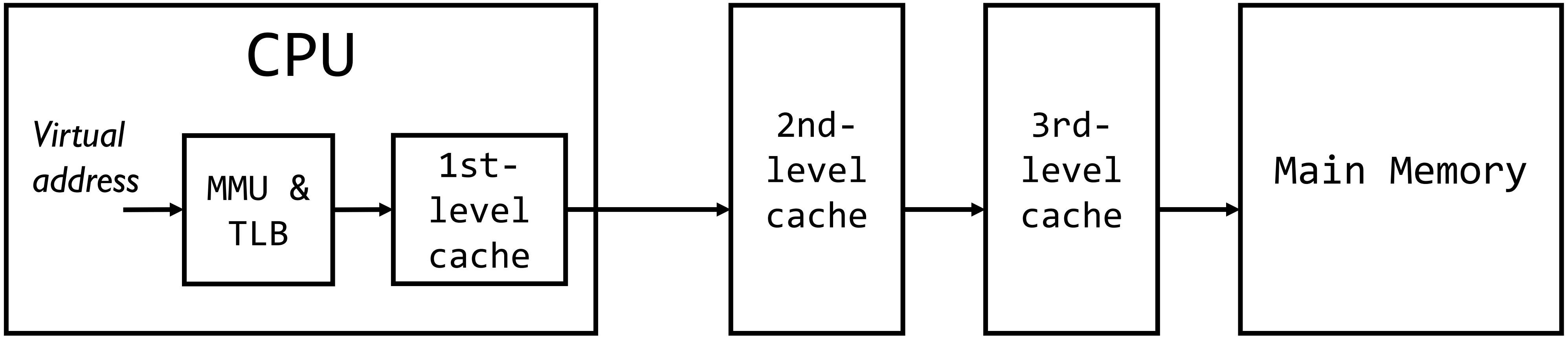
Recap: Set Associative

- 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



Recap: 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



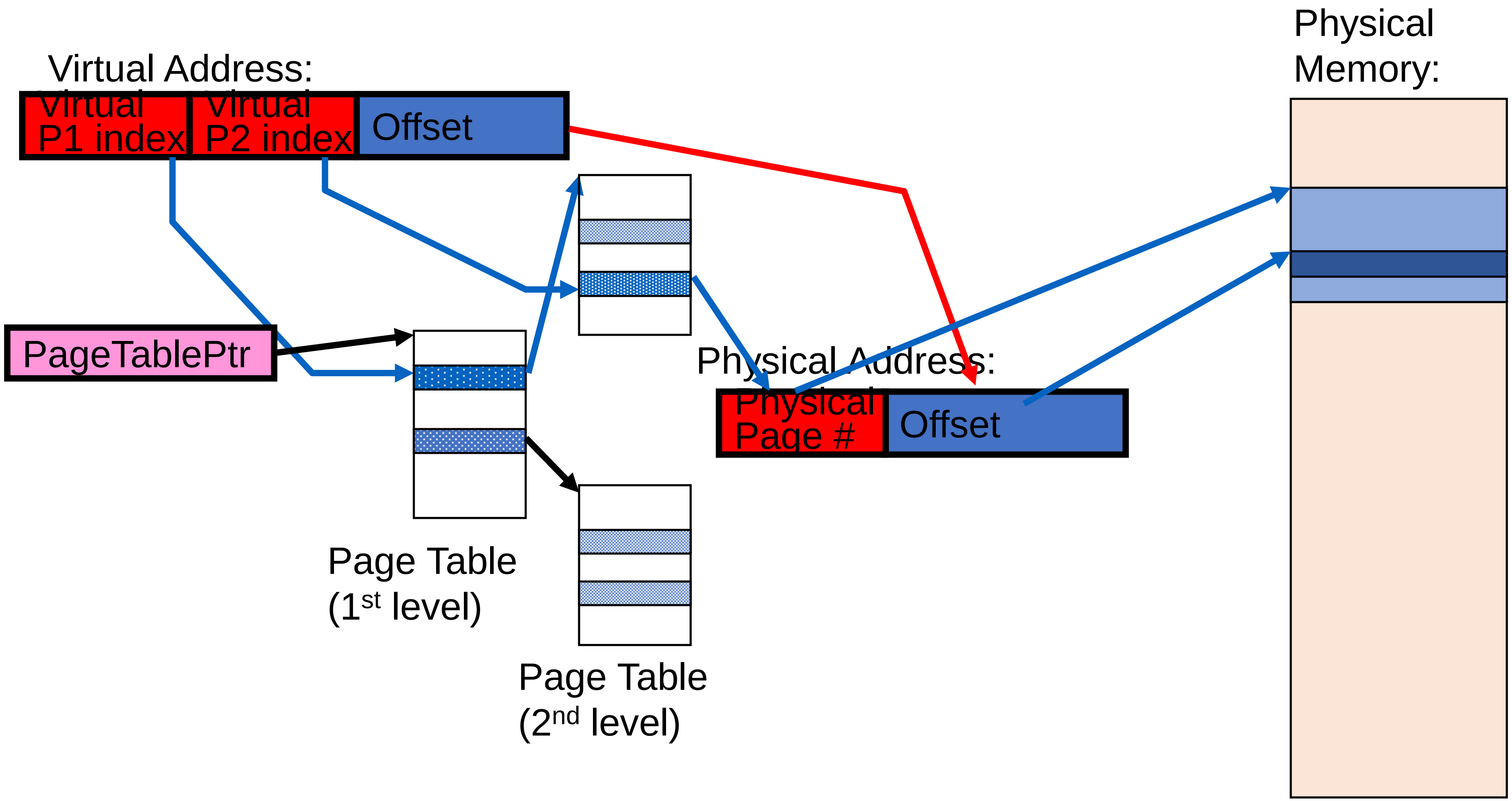
Recap: Overlapping TLB and Cache

- 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)

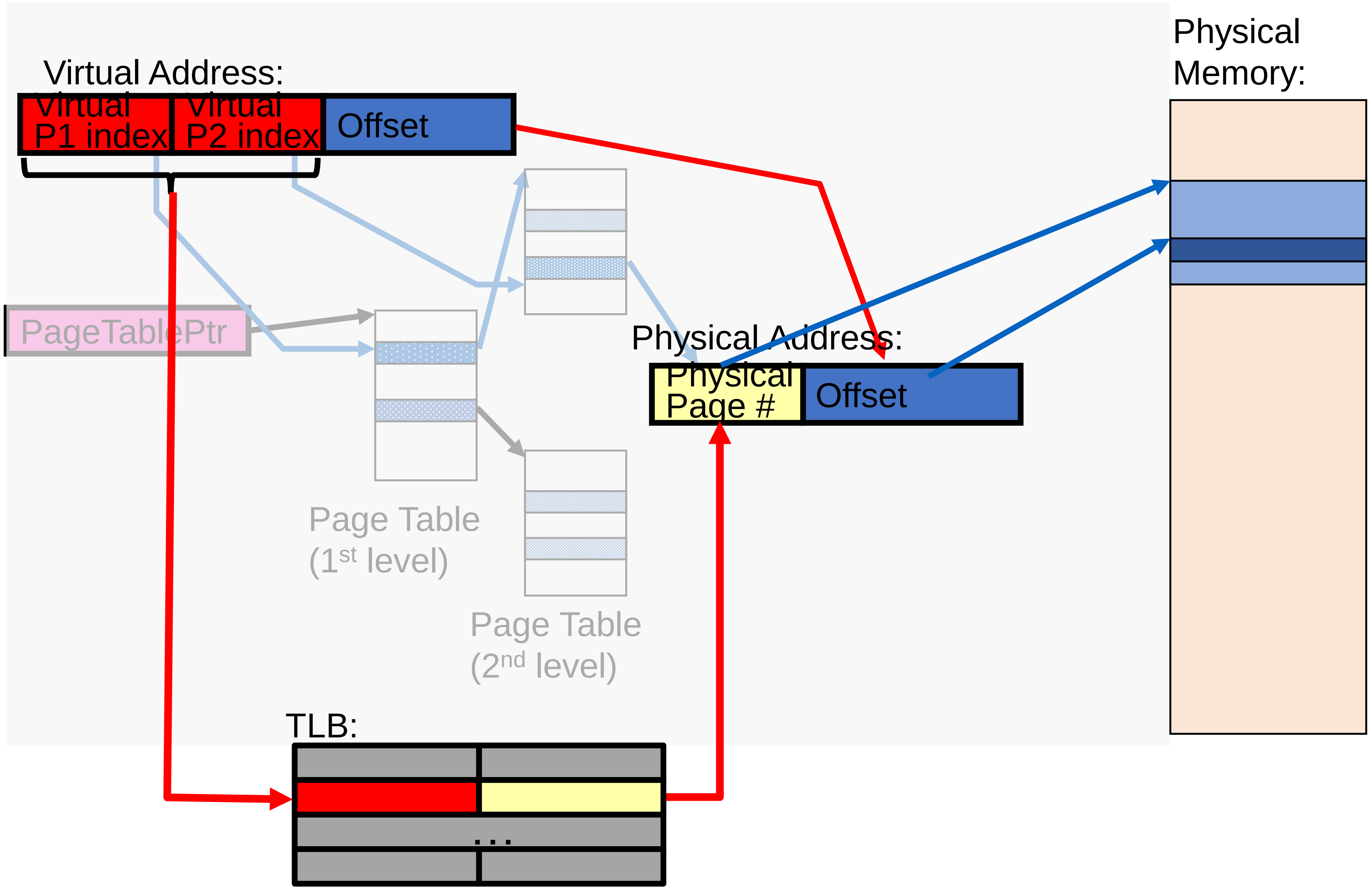
- Another option: virtually indexed, virtually tagged (VIVT)
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses
  - What's the problems?

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

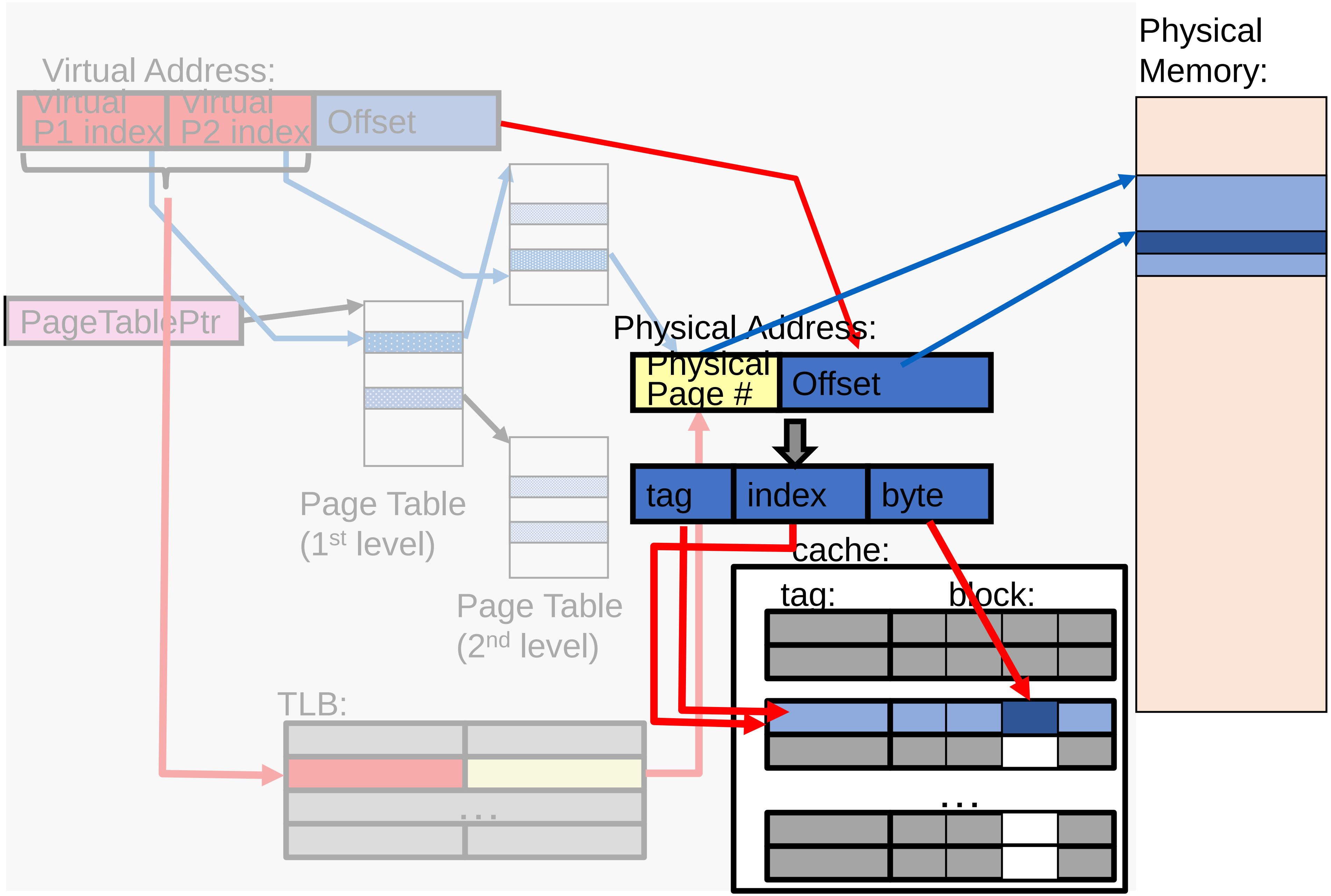
Recap: Putting Everything Together: Address Translation



Recap: Putting Everything Together: TLB

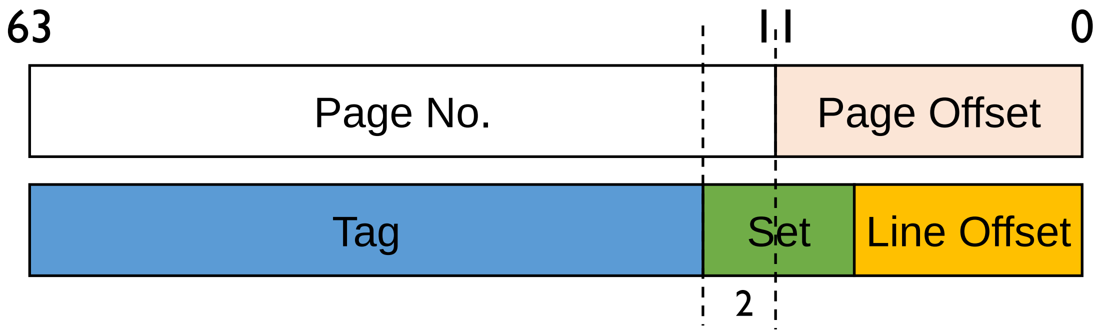


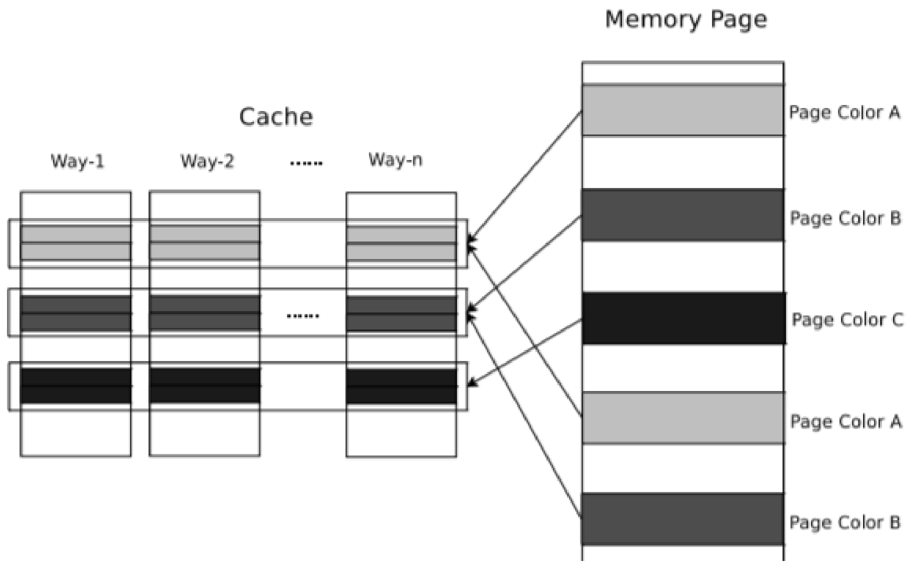
Recap: Putting Everything Together: Cache



Recap: Page Coloring

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





Consider two <u>consecutive</u> pages used by an application:

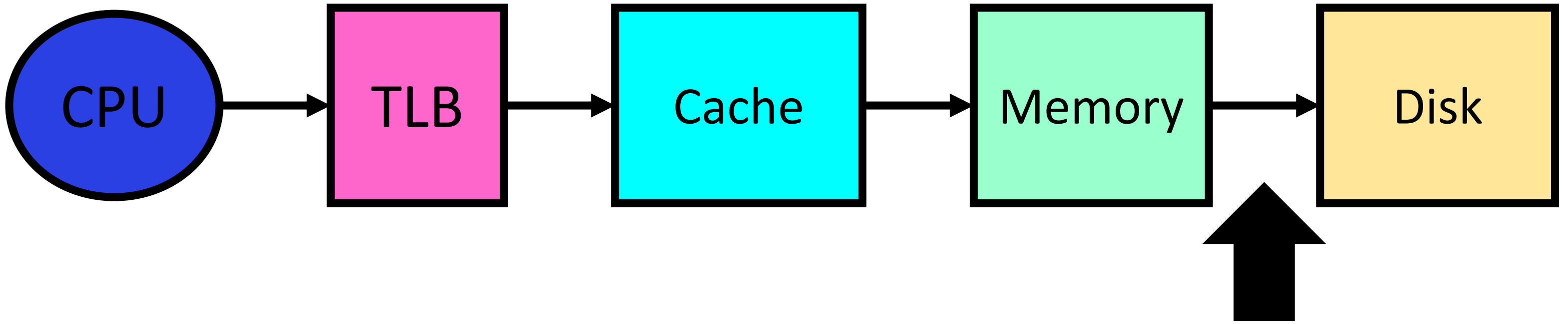
- Their virtual set number must be different
- But their physical set number could be the same after translation (when the OS maps them to the physical pages whose page numbers have the same last 2 bits). In such a case, two addresses with the same offset within these two pages will in contention for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

Cache Hierarchy

- Memory as cache for secondary disk



Demand Paging (需求分页)

- Modern programs require a lot of physical memory, but they don't use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory

Demand Paging (需求分页)

- Modern programs require a lot of physical memory, but they don’t use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user’s code to be in memory
- Solution: use main memory as cache for disk
  - “lazy” memory allocation

Demand Paging (需求分页)

- Modern programs require a lot of physical memory, but they don’t use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user’s code to be in memory
- Solution: use main memory as cache for disk
  - “lazy” memory allocation
- An illusion of infinite memory
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - More programs fit into memory, allowing more concurrency
  - **Principle: page table for transparent management**

Demand Paging as Cache

- What is block size?
  - 1 page
- What is organization of this cache (i.e. direct-mapped, set-associative, fully-associative)?
  - Fully associative: arbitrary virtual $\rightarrow$ physical mapping
- How do we find a page in the cache when look for it?
  - First check TLB, then page-table traversal
- What is page replacement policy? (i.e. LRU, Random…)
  - This requires more explanation… (kinda LRU)
- What happens on a miss?
  - Go to lower level to fill miss (i.e. disk)
- What happens on a write? (write-through, write back)
  - Write-back – need dirty bit!

Memory-mapped Files

- Memory-mapped Files (内存映射文件) is a segment of virtual memory that has been assigned a direct byte-for-byte correlation with some portion of a file or file-like resource
  - A special case of demand paging
  - A replacement for syscall read()/write()

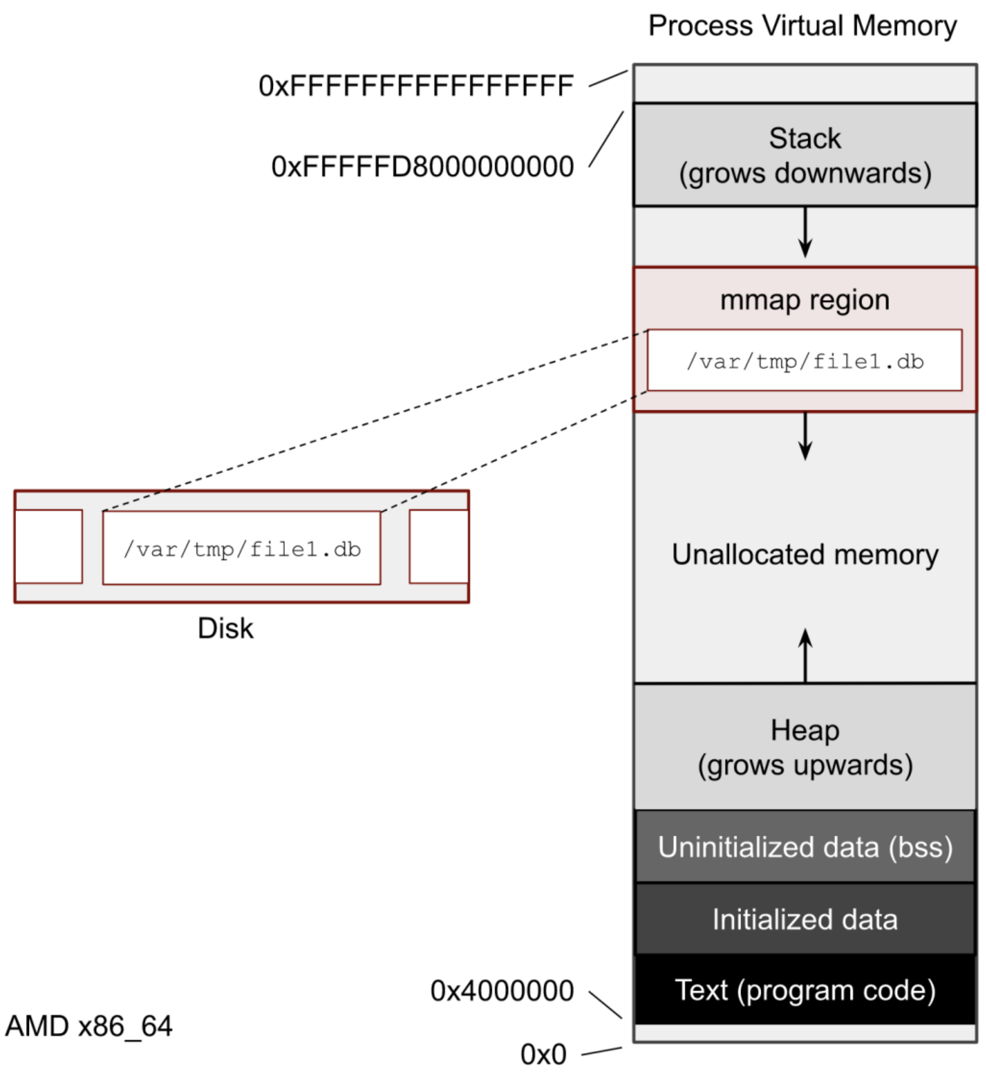
```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

mmap(): creates a new mapping in the virtual address space of the calling process. The virtual address starts at *addr* with length *length*. The contents of a file mapping are initialized using length bytes starting at *offset* offset in the file (or other object) referred to by the file descriptor *fd*.

- If addr is NULL, the OS picks a location
- Return value: the address of new mapping

Memory-mapped Files



https://biriukov.dev/docs/page-cache/5-more-about-mmap-file-access/

```c
int main() {
    int fd;
    char *mapped_data;
    struct stat file_stat;

    // Open the file for reading and writing
    fd = open("example.txt", O_RDWR);

    // Get file size
    if (fstat(fd, &file_stat) < 0) {
        return -1;
    }

    // Map the file into memory
    mapped_data = mmap(NULL, file_stat.st_size, PROT_READ |
PROT_WRITE, MAP_SHARED, fd, 0);

    // Modify the file in memory
    strncpy(mapped_data, "Modified", 8);

    // Sync changes to disk
    if (msync(mapped_data, file_stat.st_size, MS_SYNC) == -1) {
        return -1;
    }

    // Unmap the file and close fd
    if (munmap(mapped_data, file_stat.st_size) == -1) {
        return -1;
    }
    close(fd);
    return 0;
}
```

Memory-mapped Files

- PROS
  - Transparency – the program can use pointers to access those data
  - Zero copy I/O – the OS just changes the page table entries without copying the data into memory; read()/write() needs to copy the data twice (disk-kernel-user)
  - Pipelining – the program can start executing as soon as the page table has been set
  - Interprocess communication – sharing becomes easy
  - Large files – which pages shall be in memory? OS handles it for you
- CONS
  - Frequent page faults
  - A few more..

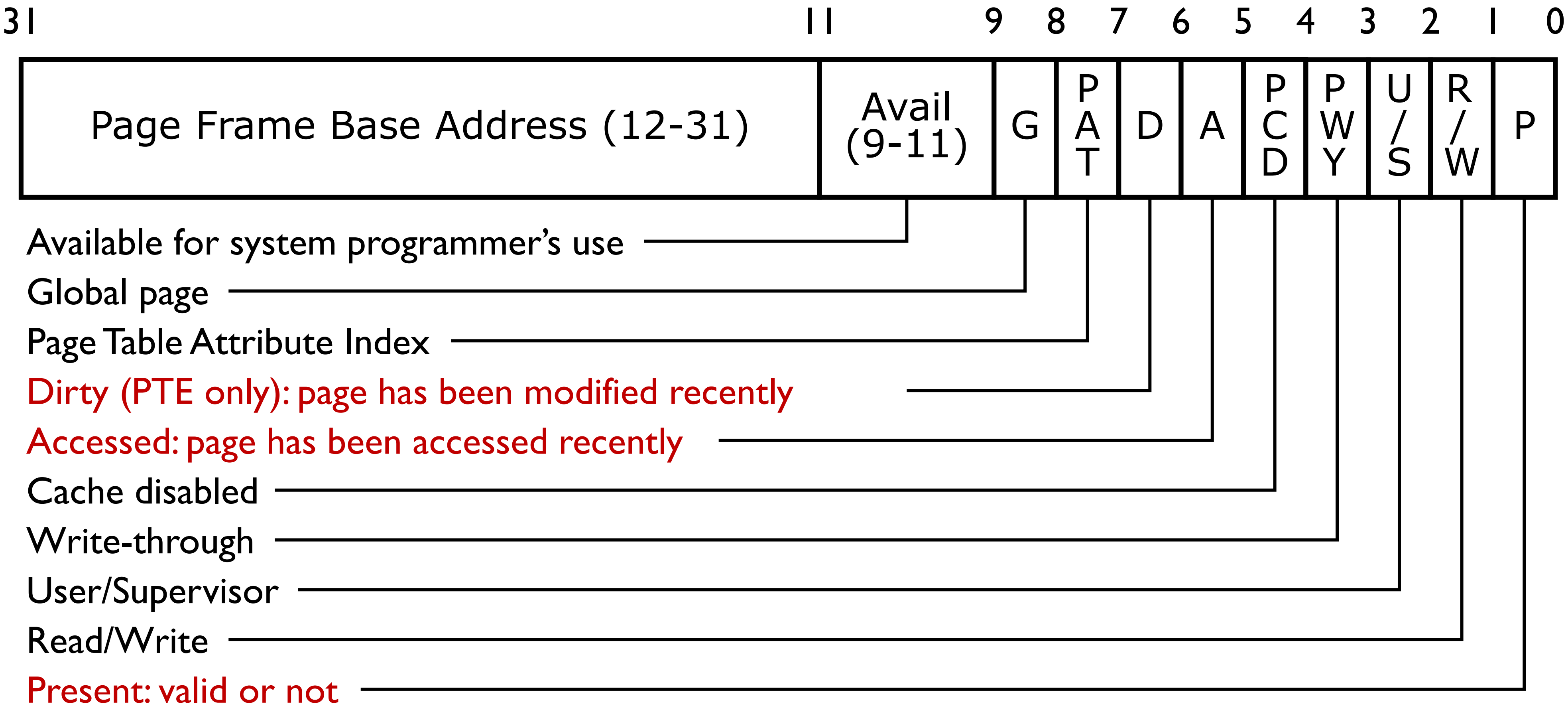
When to use mmap

- Use mmap when:
  - Random Access: access data in a non-sequential manner
  - Large Files: for very large files that may not fit into memory
  - Multiple Processes: data sharing across processes
  - Memory-Mapped I/O and automatic caching

- Instead, use read/write when:
  - Small files
  - Streaming data access
  - Portability: not every OS has mmap!

Implementation of Memory-mapped Files

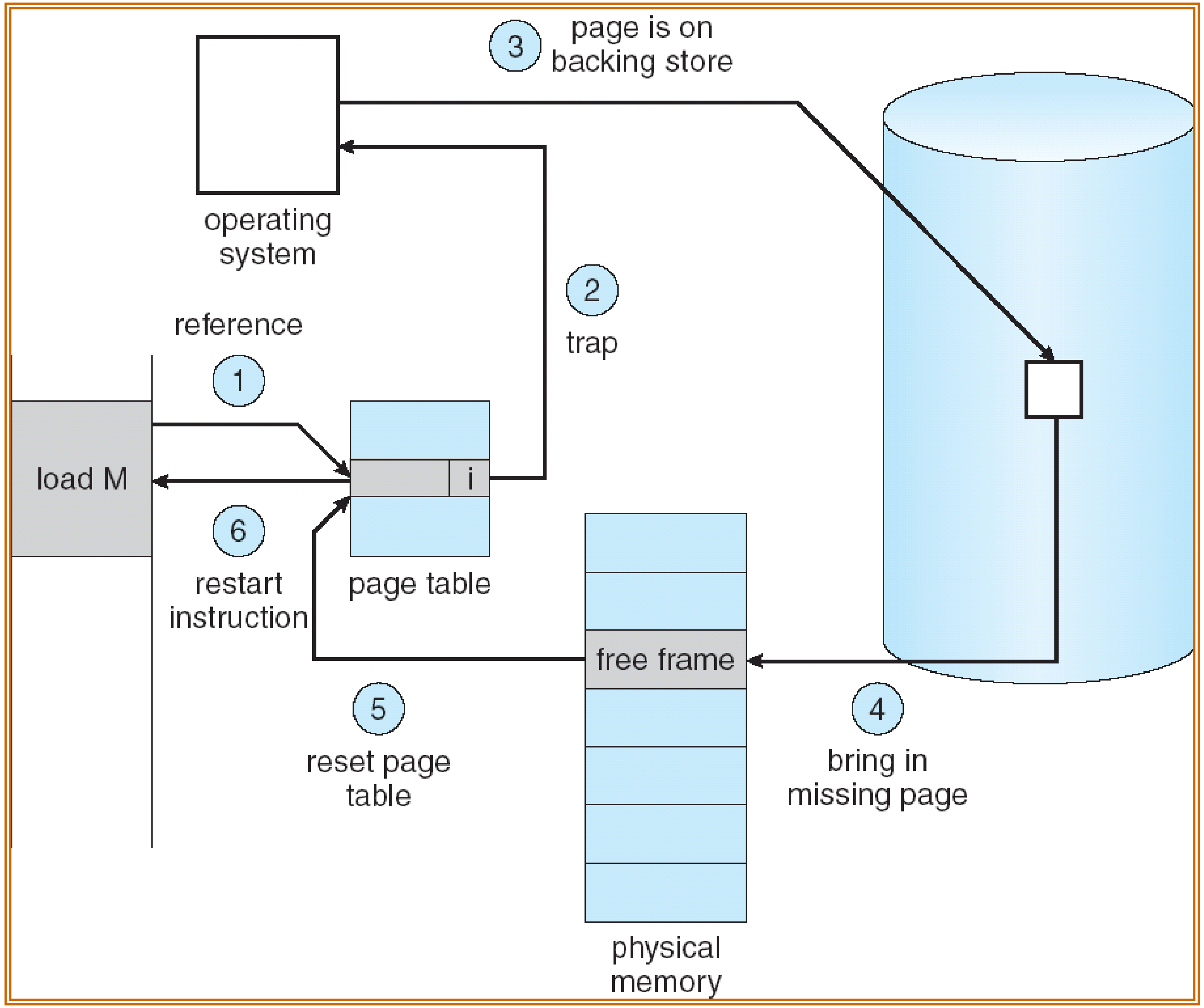
- Set up mapping
  - Initialize the page table entries and setting them to invalid



Implementation of Memory-mapped Files

- When program accesses an invalid address
  1. [MMU] TLB miss; full page table lookup
  2. [MMU + OS] Trapping into page fault handler
  3. [OS] Convert virtual address to file offset
  4. [OS] Allocate a new page frame in memory
  5. [OS] Read data from disk to the memory (blocked)
  6. [CPU] Disk interrupt when read completes
  7. [OS] Updating page table by marking the entry as valid
  8. [OS] Resume process
  9. [MMU] TLB miss; full page table lookup
  10. [MMU] TLB update

Implementation of Memory-mapped Files



Detailed Page Fault Process

Before Page Fault (done by hardware)

Process executes a memory load or store instruction, or fetches an instruction

↓

The address is first feed to cache or instruction prefetch buffer, if it is there, done;

↓

If it is not there, the address is feed to TLB, to try to find a page translation entry for it (note that the address is virtual address);

→

If hit in TLB, check for R/W/E permissions.

→ If not approved, page fault!

↓

If approved, take the physical page number, concatenate it with the page offset, and send the address to memory bus;

↓

If miss in TLB, MMU tries to find page table entry;

↓

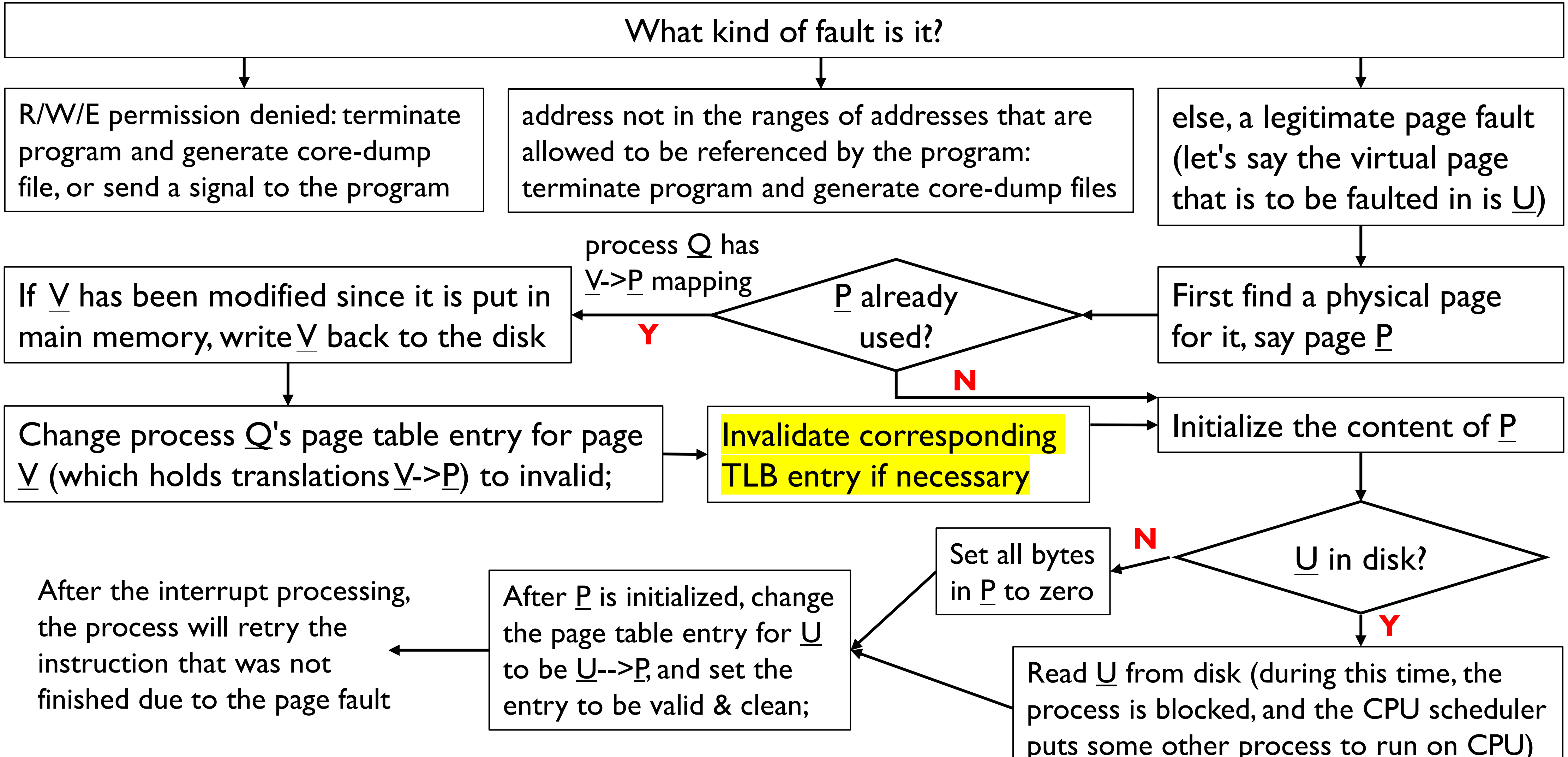
If page table entry found, the entry is put to TLB --- some entry has to be replaced out of TLB, hardware-decided replacement, random replacement;

→ If MMU can't find page table entry or permission not satisfied, page fault!

ref: https://pages.cs.wisc.edu/~cao/cs537/lecture17.txt

Detailed Page Fault Process

Handling Page Fault (done by hardware)



ref: https://pages.cs.wisc.edu/~cao/cs537/lecture17.txt

The Dirty Bit

- How does OS know which pages have been modified?
  - Assuming every page has been modified is correct but inefficient
- The hardware tracks it with a dirty bit in page table entry
  - Initialized to 0
  - Set to 1 whenever there is a store instruction for the page
- The TLB also has a dirty bit
- Unix has a background thread to clean pages when it’s too full

Allocating New Page Frame

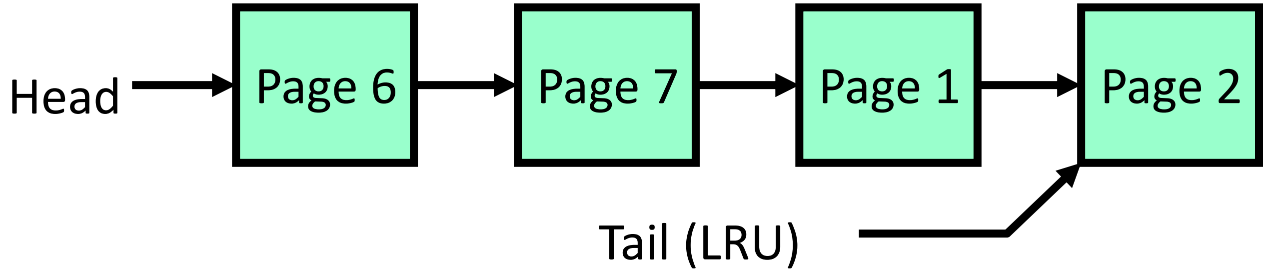
- If there is an empty page, use it
- If there is no empty page
  - Select a page to evict
    - ❑ Need a lightweight policy
  - Find page table entries that point to the evicted page
    - ❑ Core map – an array that maps physical page frames back to the table entries
  - Set page table entry to invalid
    - ❑ TLB shootdown is needed. Why?
  - Copy back any changes to the evicted page
    - ❑ Write back
    - ❑ The same for application exit
    - ❑ **Dirty bit**

Page Eviction Policy

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

Page Eviction Policy

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



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

Page Eviction Policy

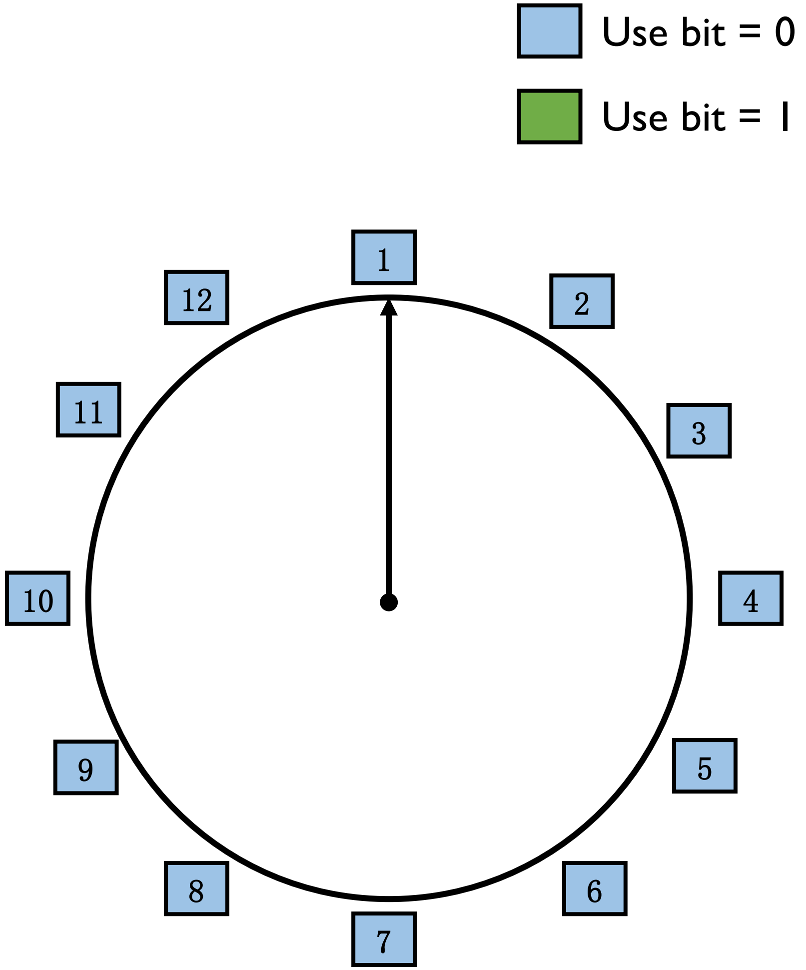
- Why we can implement LRU for TLB entry replacement, but not demand paging replacement?

Page Eviction Policy

- Why we can implement LRU for TLB entry replacement, but not demand paging replacement?
  - TLB is purely handled in hardware (MMU)
  - TLB has fewer entries (typically 16-512)

Page Eviction Policy

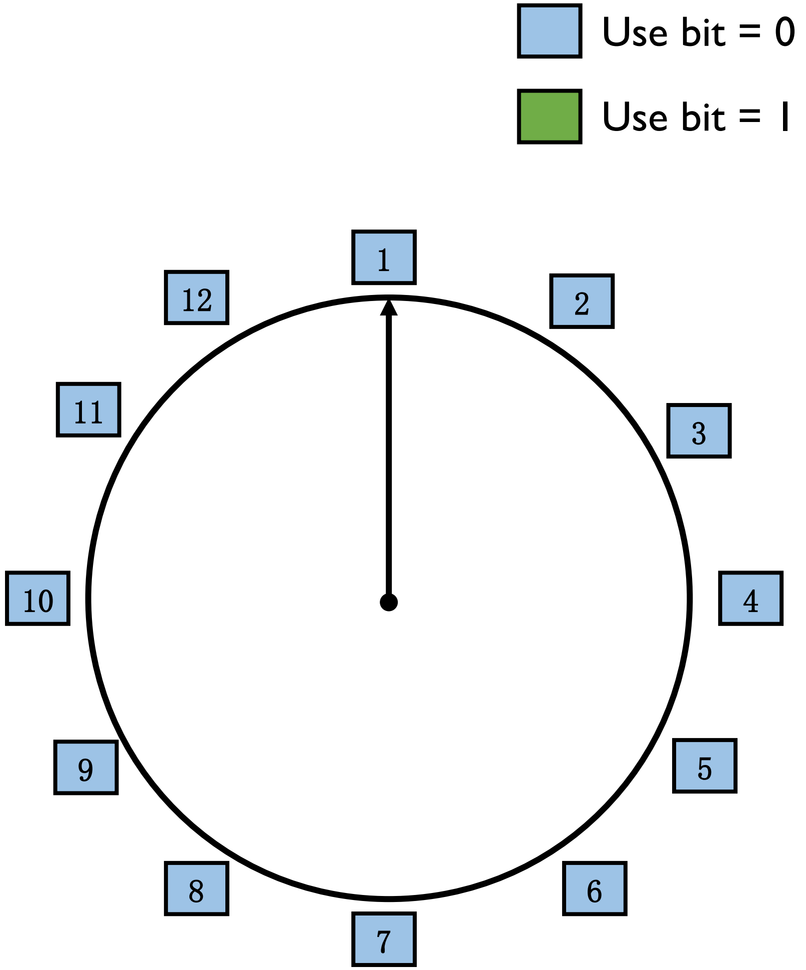
- Clocking algorithm: approximating LRU
- Implementation with the *<u>use</u>* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream:

Page Eviction Policy

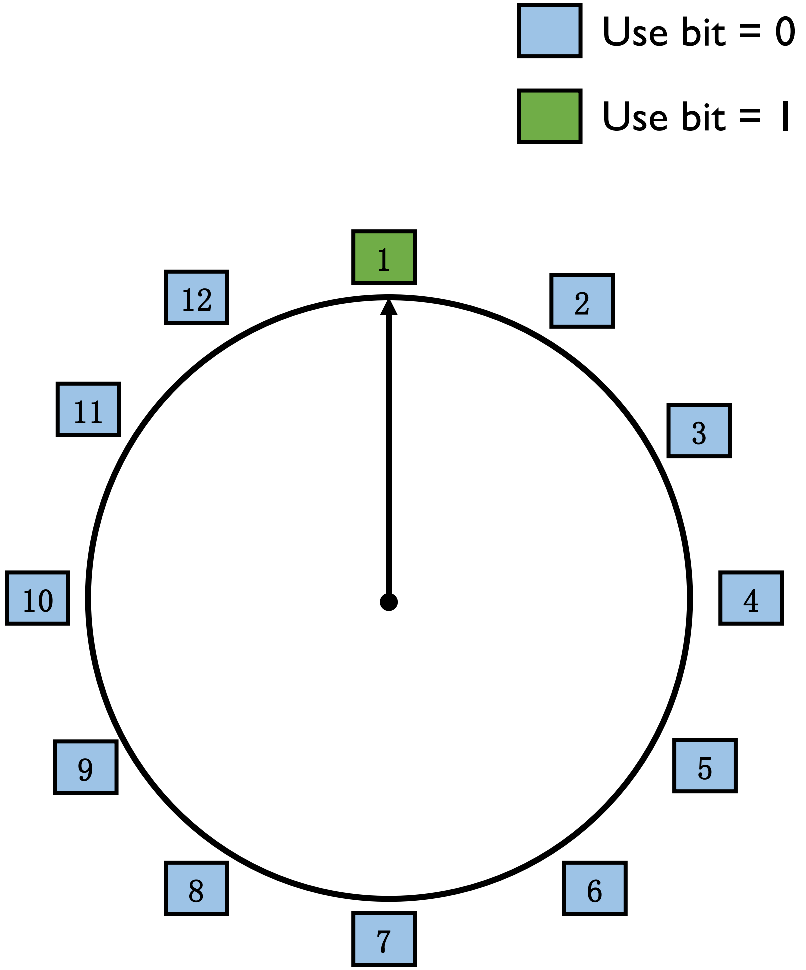
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1

Page Eviction Policy

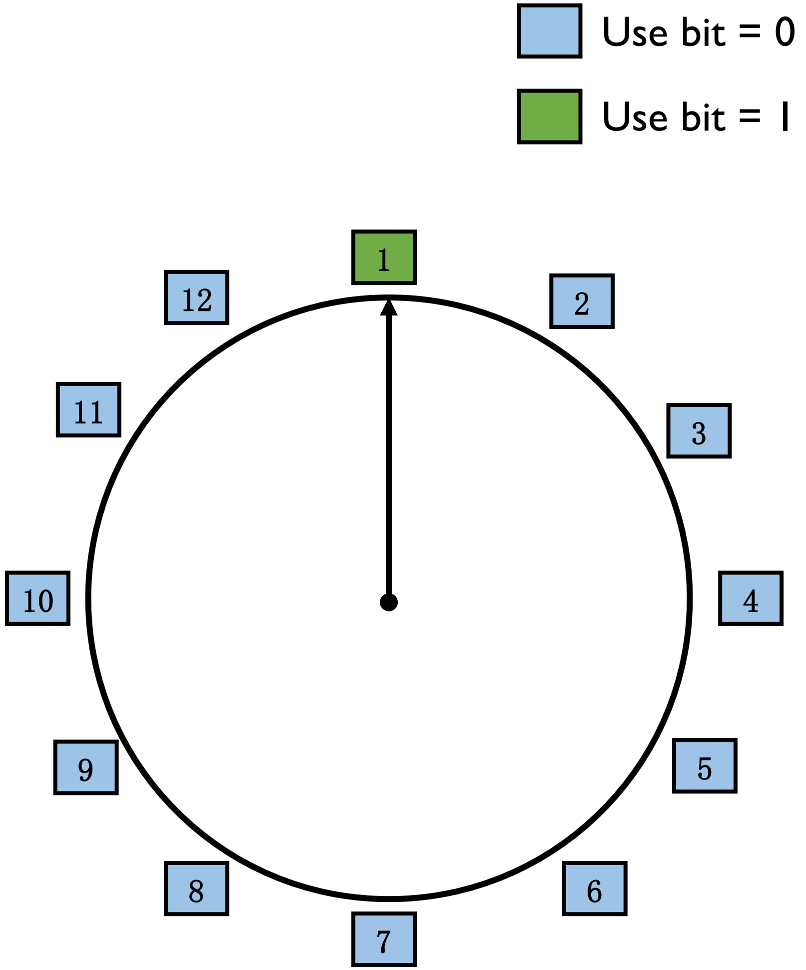
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1

Page Eviction Policy

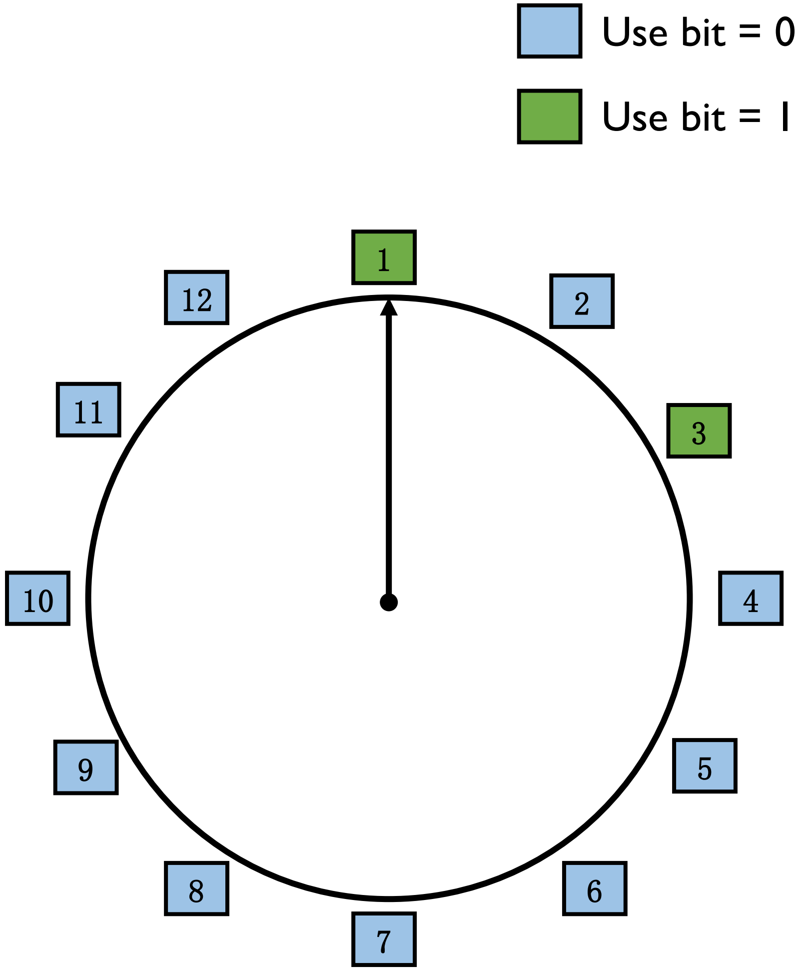
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1 3

Page Eviction Policy

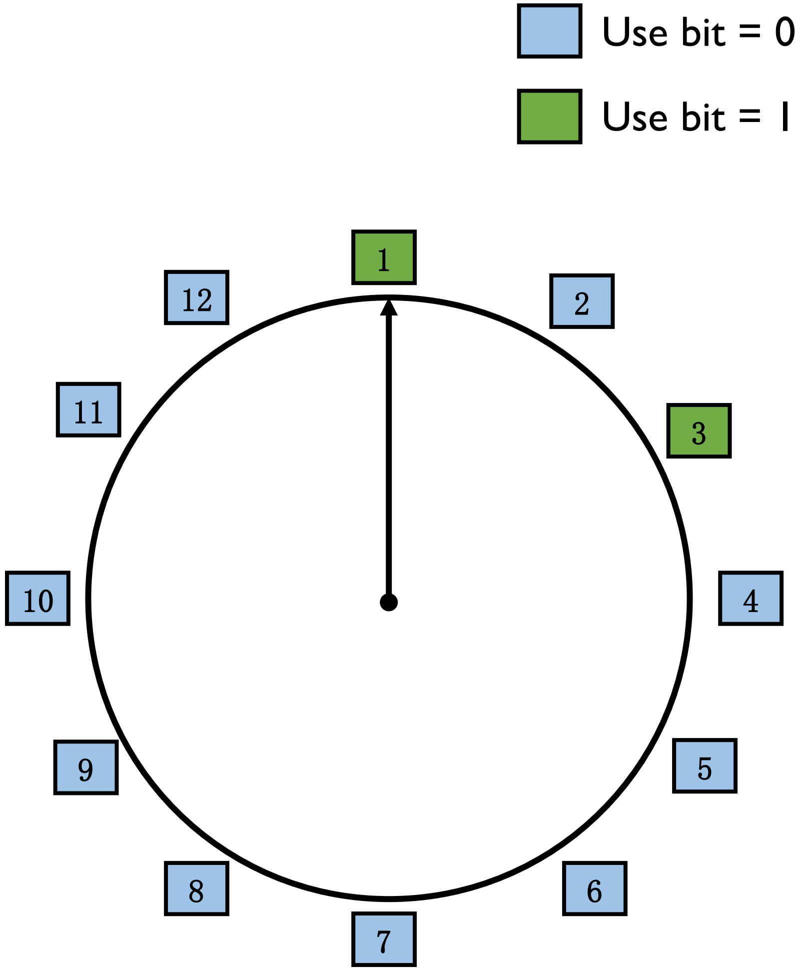
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1 3

Page Eviction Policy

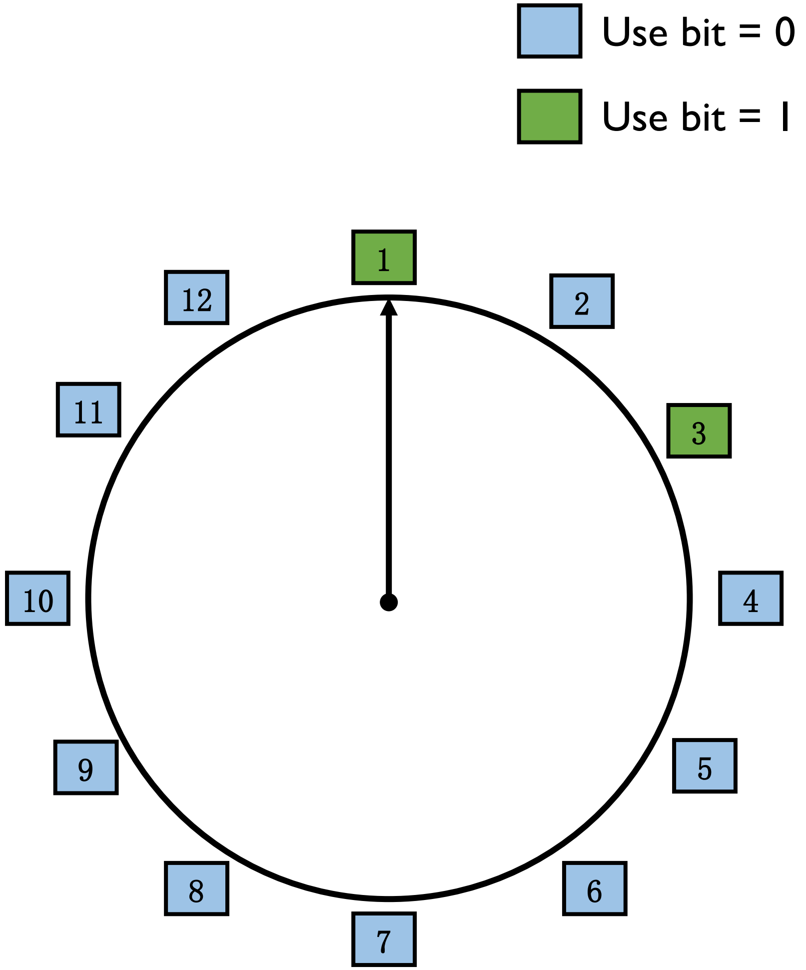
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1 3 1

Page Eviction Policy

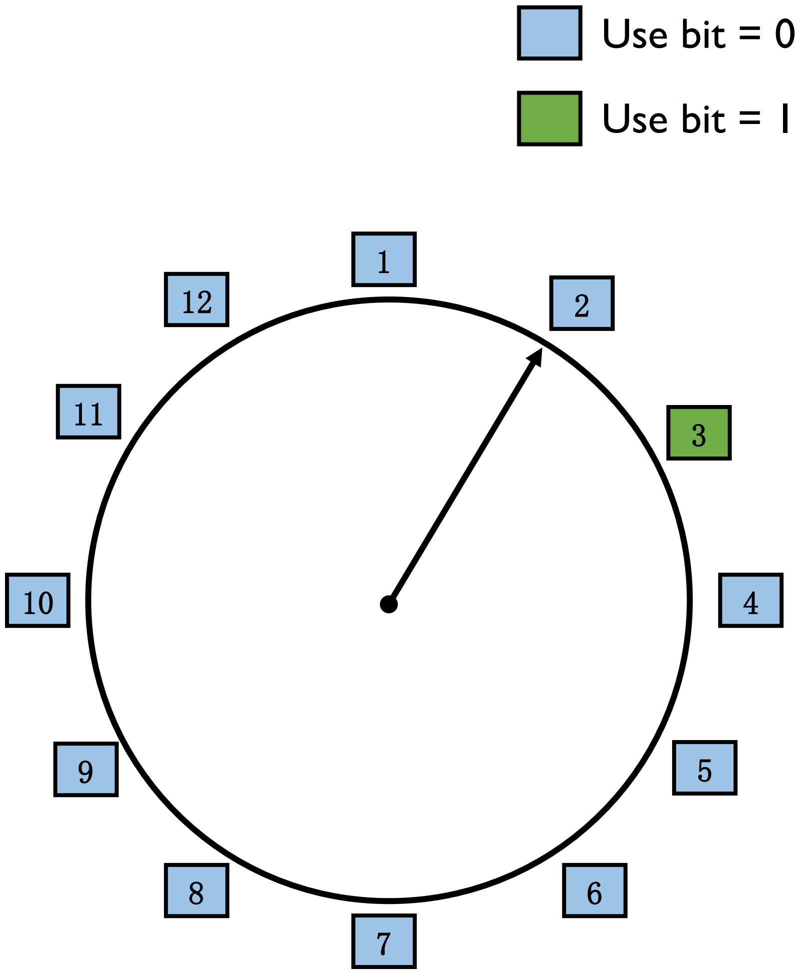
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1 3 1 20

Page Eviction Policy

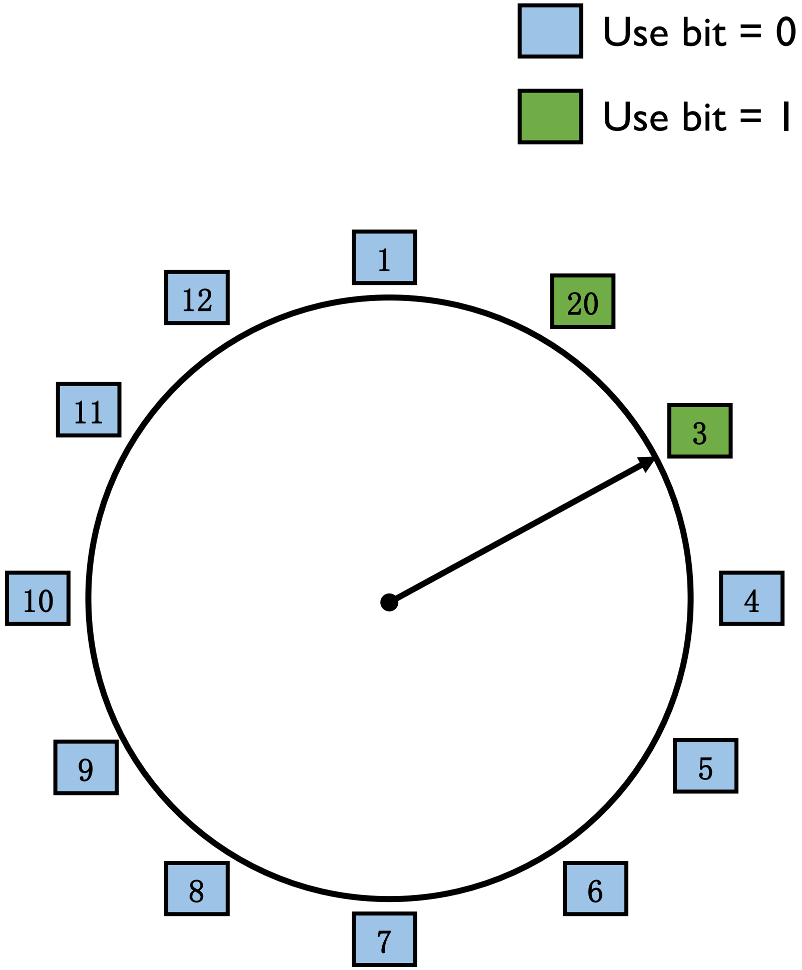
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1 3 1 20

Page Eviction Policy

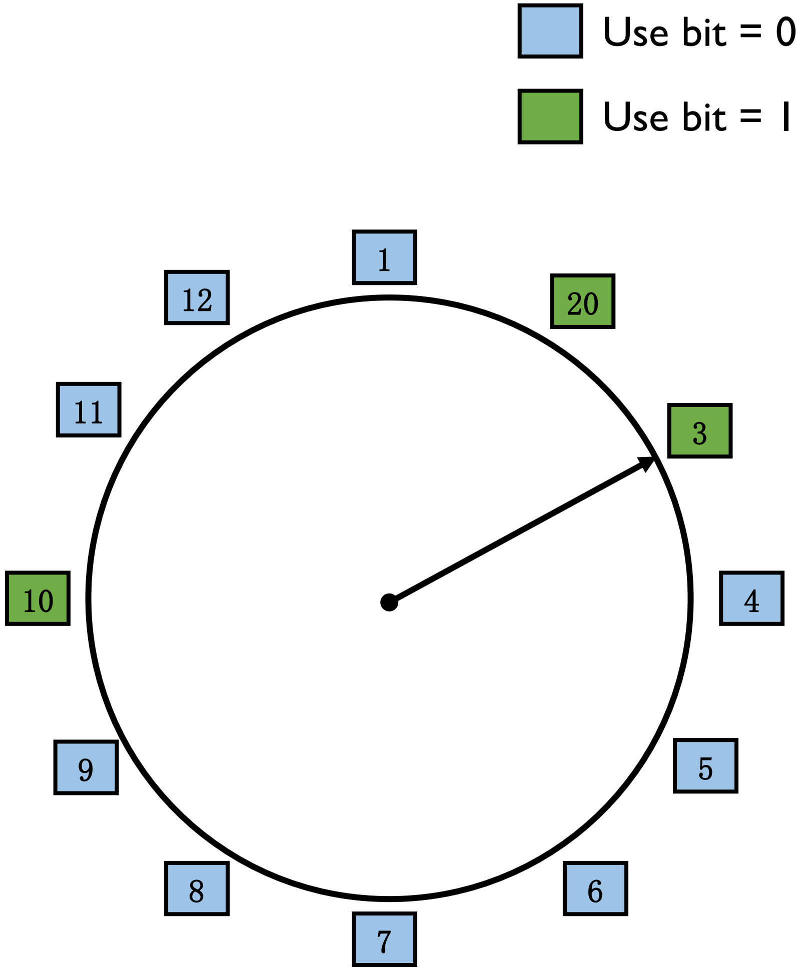
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1 3 1 20

Page Eviction Policy

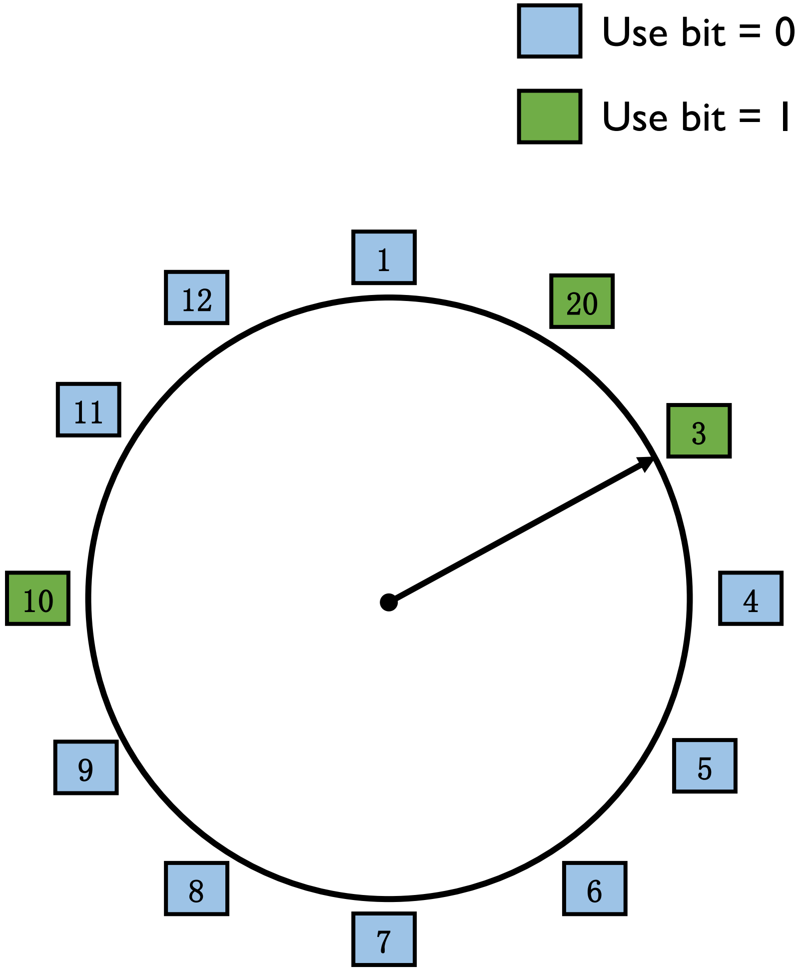
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1 3 1 20 10

Page Eviction Policy

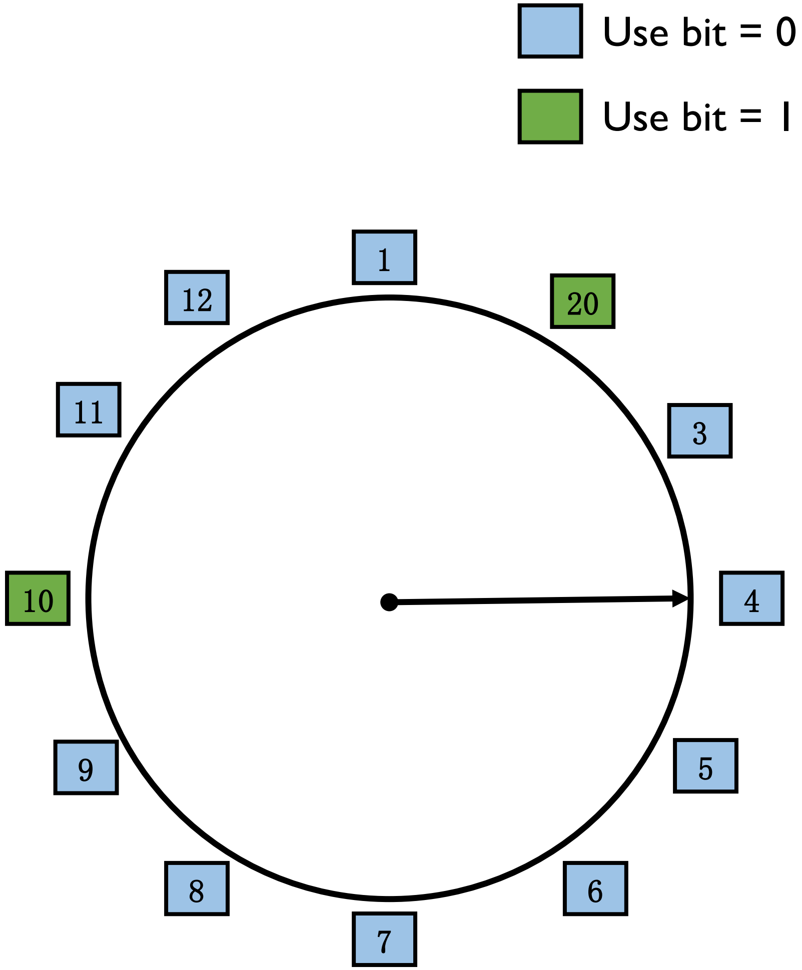
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1 3 1 20 10 25

Page Eviction Policy

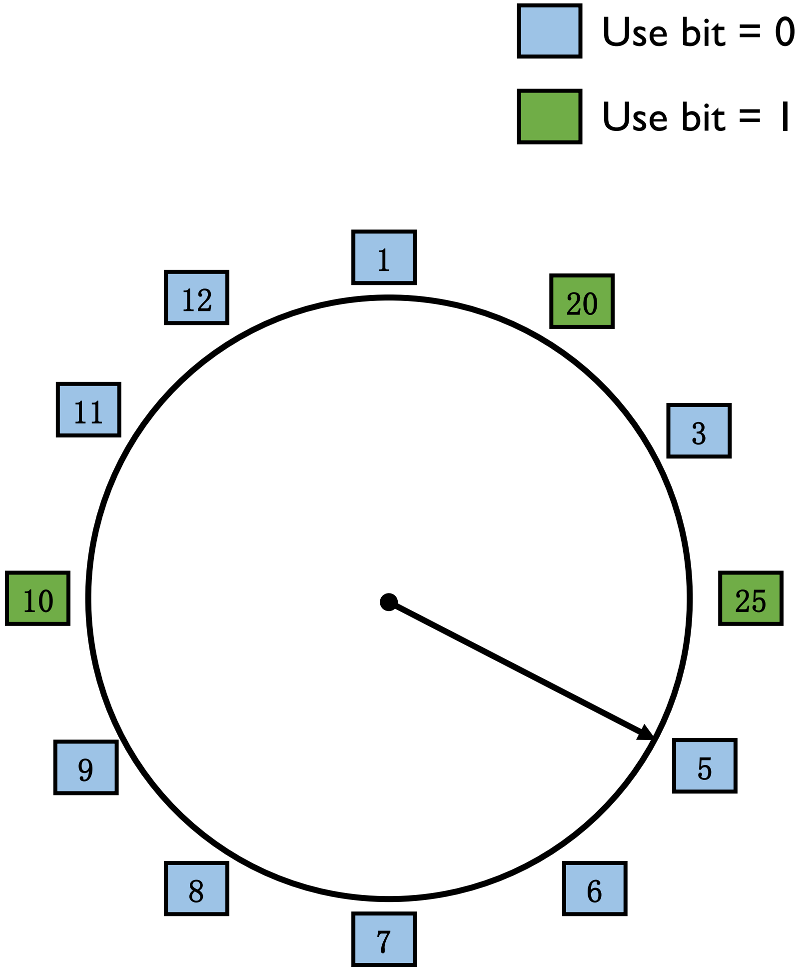
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1 3 1 20 10 25

Page Eviction Policy

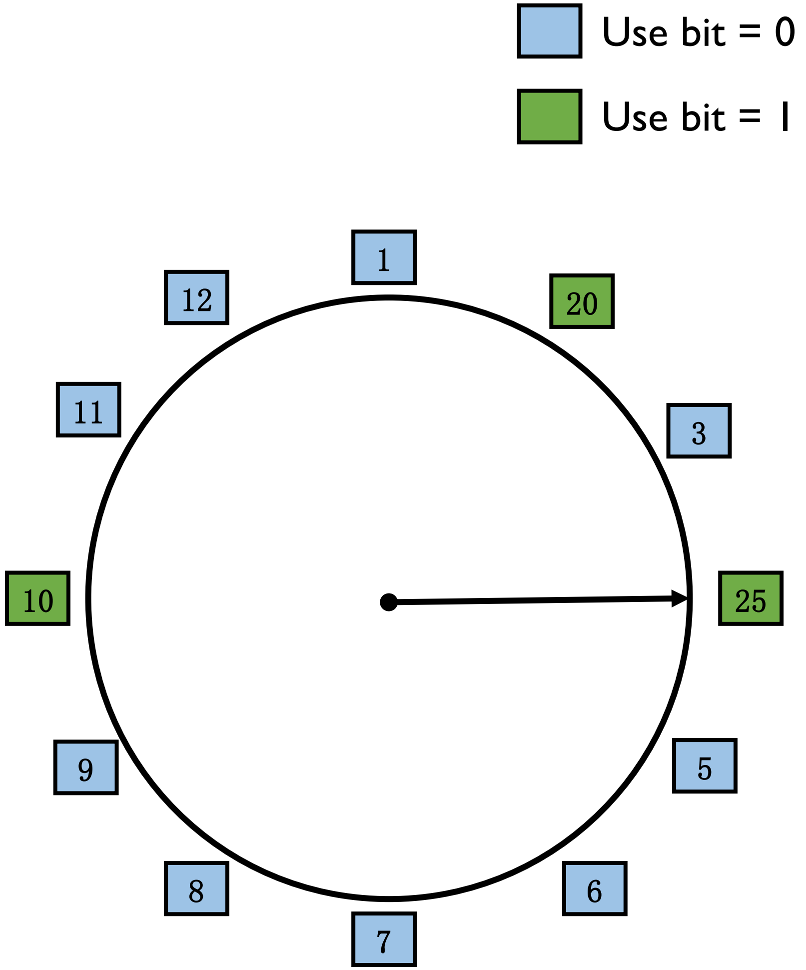
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1 3 1 20 10 25

Page Eviction Policy

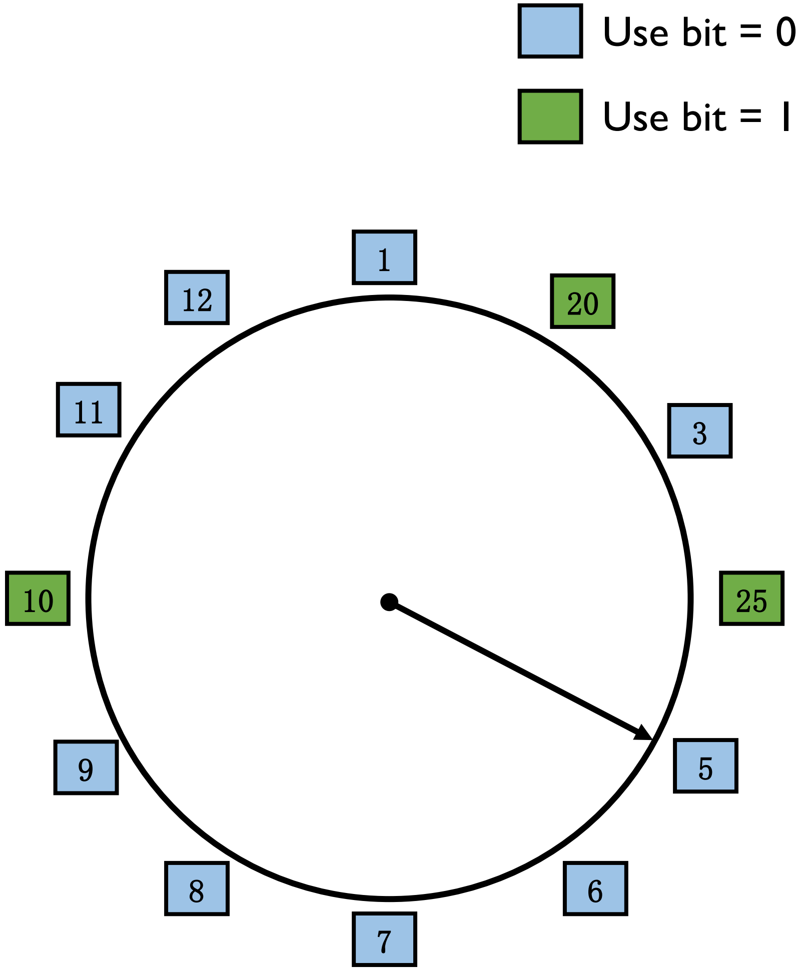
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1 3 1 20 10 25 2

Page Eviction Policy

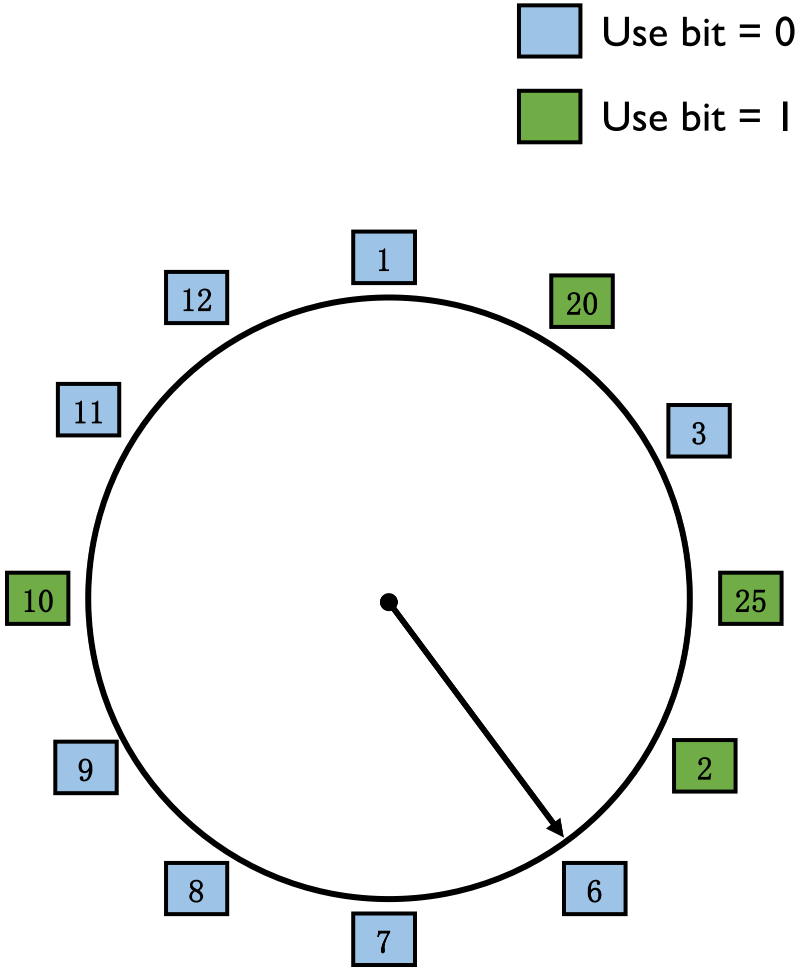
- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it



Page reference stream: 1 3 1 20 10 25 2

Page Eviction Policy

- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it

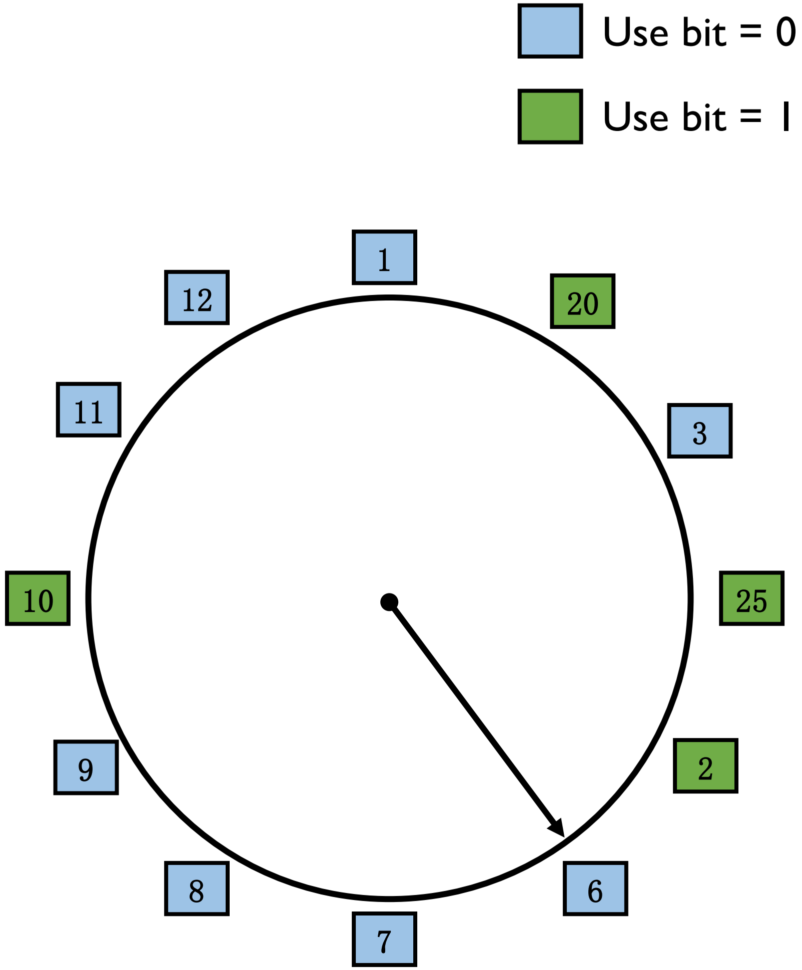


Page reference stream: 1 3 1 20 10 25 2

Page Eviction Policy

- Clocking algorithm: approximating LRU
- Implementation with the *use* bit
  - Initialized to 0 in page table
  - Set to 1 whenever there is a page access
- When we need to evict a page, we look at the page under the hand:
  - If its use bit = 1, clear it and move the hand, repeat;
  - If its use bit = 0, evict it

What if hand moving slowly? Good sign or bad sign?
What if hand moving quickly? Good sign or bad sign?



Page reference stream: 1 3 1 20 10 25 2

N^{th} Chance Version of Clock Algorithm

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

Details of Clock Algorithms

- Which bits of a PTE entry are useful to us?
  - Use: set when page is referenced; cleared by clock algorithm
  - Modified: set when page is modified, cleared when page written to disk
  - Valid: ok for program to reference this page
  - Read-only: ok for program to read page, but not modify
    - ❑For example for catching modifications to code pages!
- Do we really need hardware-supported “modified” bit?
  - No. Can emulate it (BSD Unix) using read-only bit
    - ❑ Initially, mark all pages as read-only, even data pages
    - ❑ On write, trap to OS. OS sets software “modified” bit, and marks page as read-write.
    - ❑ Whenever page comes back in from disk, mark read-only

Allocation of Page Frames

- How do we allocate memory among different processes?
  - Does every process get the same fraction of memory? Different fractions?
  - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
  - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    - instruction is 6 bytes, might span 2 pages
    - 2 pages to handle *from*
    - 2 pages to handle *to*
- Possible Replacement Scopes:
  - Global replacement – process selects replacement frame from set of all frames; one process can take a frame from another
  - Local replacement – each process selects from only its own set of allocated frames

Allocation of Page Frames

- Self-paging (自分页): ach process is responsible for managing its own page faults and memory allocation, rather than relying on a global operating system-wide policy.

- Global page management
  - each process/user is assigned its fair share of page frames using max-min scheduling algorithm
  - when memory is full, the page eviction happens to the process with the most allocated memory.
    - ❑ Avoid malicious attackers that wants as much as resources

Summary

- To support demand paging, what do CPU/OS contribute?
  - CPU: memory management (MMU), a few bits in page table entry, etc
  - OS: page table manipulation, eviction strategy, page fault handler, etc

Advanced: Android Memory Management

- How does Android (or other mobile OSes) handle memory inefficiency, e.g., too many apps opened?
  - Strategy #1: swapping (demand paging)
  - (primary) Strategy #2: low memory killer (LMK)
    - vs. out-of-memory (OOM) killer in Linux
  - Android prefers the 2nd one, because:
    - Flash memory has limited write endurance.
    - Disk I/O is generally slower and consumes more power compared to RAM access.
    - Responsive time is more important on mobile apps