Mechanism – Paging: TLB & Hierarchical Tables

2023 Fall COMP3230A

Contents

- How to make address translation **faster** under Paging?
  - Hardware support - Translation Lookaside Buffer (TLB)
- How to **reduce memory demand** in storing the Page Tables?
  - Multi-level Page Tables
  - Inverted Page Tables

Related Learning Outcomes

- ILO 2b - describe the principles and techniques used by OS in effectively virtualizing memory resources.

Readings & References

- Required Readings
  - Chapter 19 – Paging: Faster Translation (TLBs)
    - http://pages.cs.wisc.edu/~remzi/OSTEP/vm-tlbs.pdf
  - Chapter 20 – Paging: Smaller Tables
    - http://pages.cs.wisc.edu/~remzi/OSTEP/vm-smalltables.pdf

Make Translation Faster

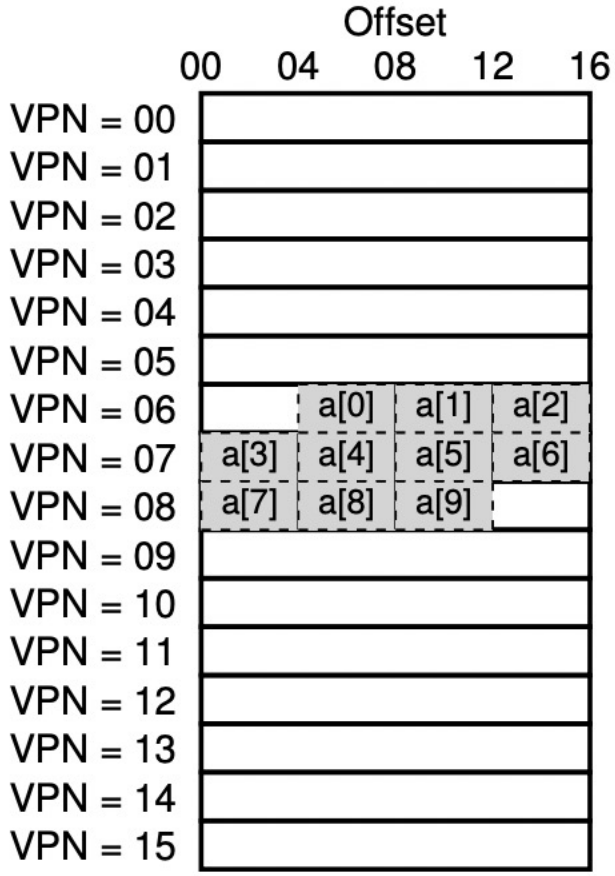
- Again need hardware support – Translation-Lookaside Buffer (**TLB**)
  - Some kind of high-speed **cache** in MMU
  - i.e., some **recently** or frequently used virtual-to-physical address **translation mappings** are cached in TLB

- Can give good performance in address translation even with relatively small TLB due to **principle of locality**
  - An instruction or data item that has been recently accessed will likely be re-accessed soon in the future – ??
  - A program accesses memory at address x, likely to access memory near x in the nearby future – ??

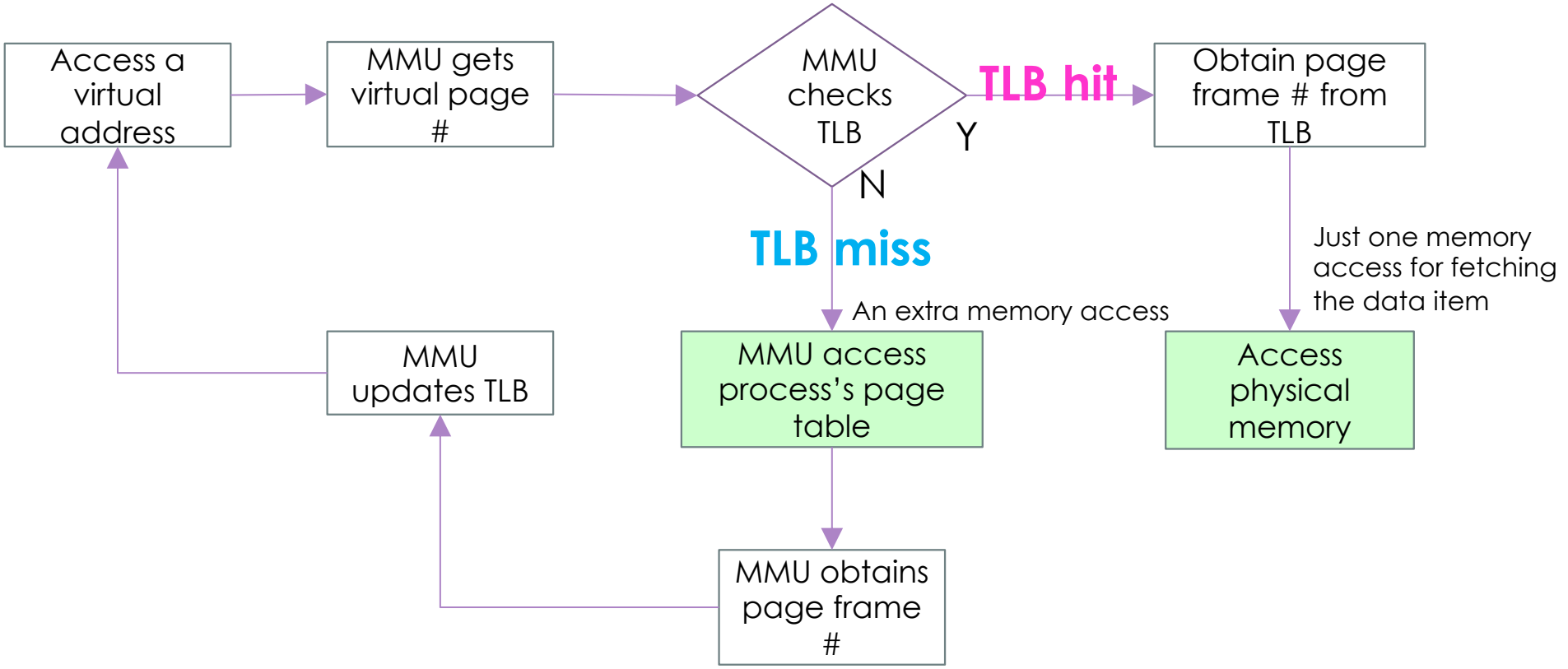
Locality: An Example

- Spatial Locality
- Temporal Locality

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```



Address Translation with TLB



Cost(*addr translation*) = Cost(*TLB lookup*) +
Cost(*full translation*) x (1 – P(*TLB hit*))

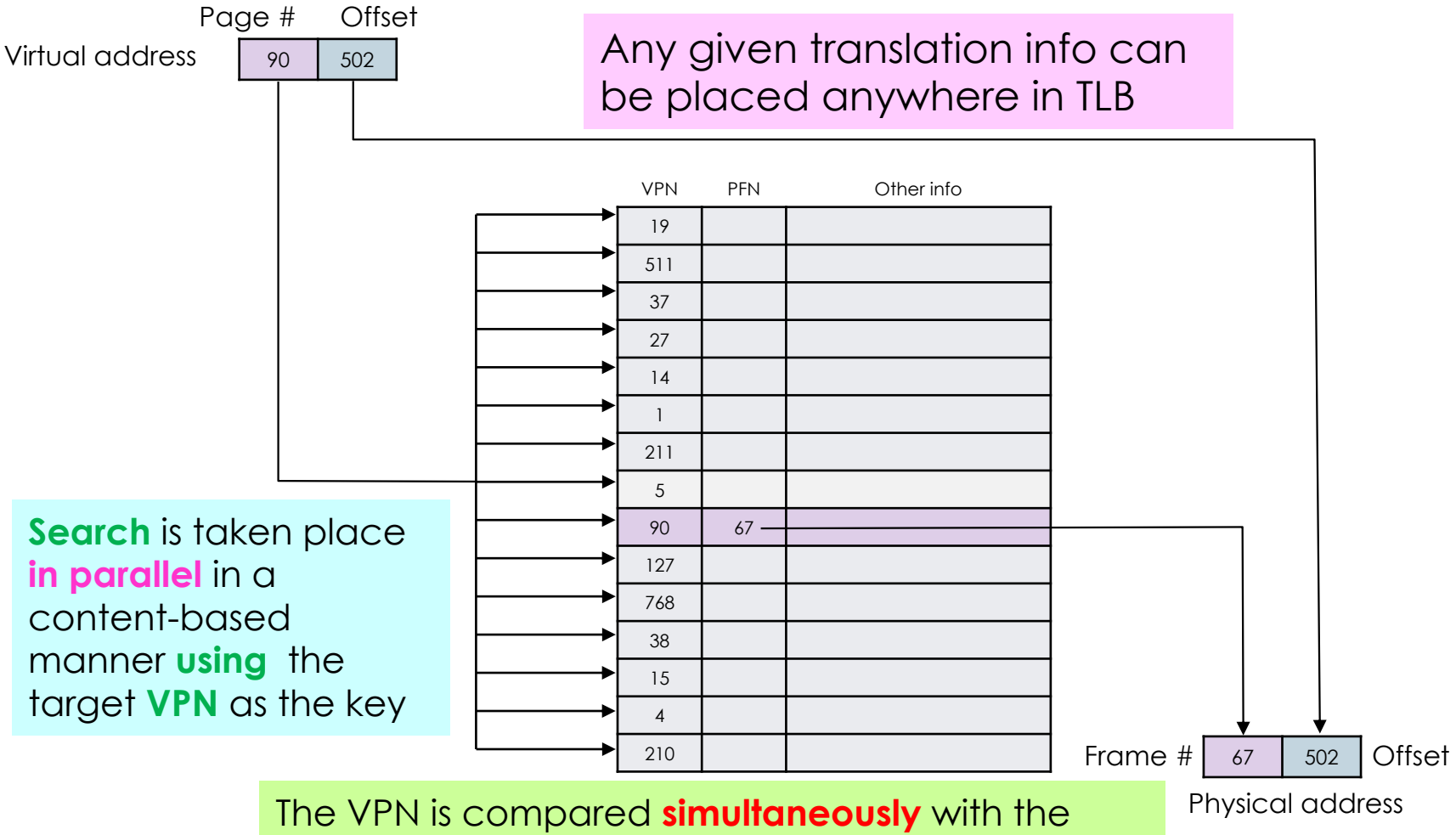
NOT FOR EXAM

More About TLB Misses

- Who handles TLB misses?

- Hardware-managed TLB (e.g., Intel x86)
  - The hardware fetches the page table (via page table base register), finds the PTE and extracts the desired translation, updates the TLB, and retries the instruction
- Software-managed TLB
  - On a TLB miss, the hardware simply raises an exception and jumps into a trap handler
  - The trap handler lookups the page table, updates the TLB with a privileged instruction, and return-from-trap
    - The hardware retries the instruction that caused the trap
  - Easier OS design but slower

TLB – Content-addressed associative memory



Issues with TLB: Context Switches

- Mapping info stored in TLB is only relevant to current running process
- Can TLB be shared between processes?
  - If not, what ought to be done during context switch?
    - CPU could simply flush (or **invalidate** the valid bit) the TLB on context switches
    - That means, every time when a process switches in, it will suffer a fair number of TLB misses
  - If shared, system does not need to flush TLB on context switch; then, how can MMU identify the set of TLB entries that is relevant to current running process?
    - Some systems add an extra field – Address Space Identifier (ASID), in the TLB, to indicate which process "owns" this TLB entries

Issues with TLB: Replacement Policy

- It is costly to have large size TLB, typical TLB might have 32, 64, or 128 entries.
- When there is a TLB miss, the system has to install a new entry in TLB. It is possible that all entries are in used, we have to "kick out" one entry to make way for the new one. Which one to replace?
- A common approach is to adopt the least-recently-used policy
  - An entry that has not recently been used is a good candidate for eviction
  - Randomly replace one entry
  - We shall have more discussion on this when we talk about memory replacement policies

NOT FOR EXAM

Issues with TLB: Consistency

- Consistency with the original data is an issue for any cache system.
- Context switches
- Permission reduction
  - OS needs to be involved when it modifies the page table.
- TLB shootdown
  - On multi-core system, each core has a TLB
  - Every of them need to be consistent when a page table entry is modified!

NOT FOR EXAM

Superpages

- A set of contiguous pages in physical memory that map a contiguous region of virtual memory
  - The pages are aligned and share the same high-order (superpage) address
- Each entry in the TLB has a flag bit, indicating whether a page or a superpage
  - For superpages, the TLB matches the superpage number
- Complicate the VM but significantly improve TLB hit rate
  - E.g., frame buffer for screen (redrawing the entire display/vertical lines)
  - Large matrices in scientific computing

NOT FOR EXAM

More Towards Efficient Address Translation

- Virtually Addressed Caches
  - Stores a copy of the contents of physical memory, indexed by VA
  - Before TLB
- Physically Addressed Caches
  - Consulted as a 2nd-level cache after TLB but before main memory
  - Speed up memory references and TLB misses
  - Many systems include a 2nd-level (fast, small, 256KB) and 3rd-level (large, slower, 2MB)

Make "Smaller" Page Table

- Page tables are big and consume too much physical memory
  - Let's have the math
    - 32-bit x86 with 4 KiB page size, a page table needs to store 2^{20} = 1048576 entries. If each entry is of 4-byte, a page table is 4 MiB in size
- Simply increases size of a page, that should reduce size of a page table
  - Increase page size to 16 KiB, a page table needs to store 2^{18} entries. If each entry is of 4-byte, the size of a page table becomes 1 MiB
- Disadvantages of having bigger pages
  - High chance of having internal fragmentation
  - Longer page loading time

Make "Smaller" Page Table

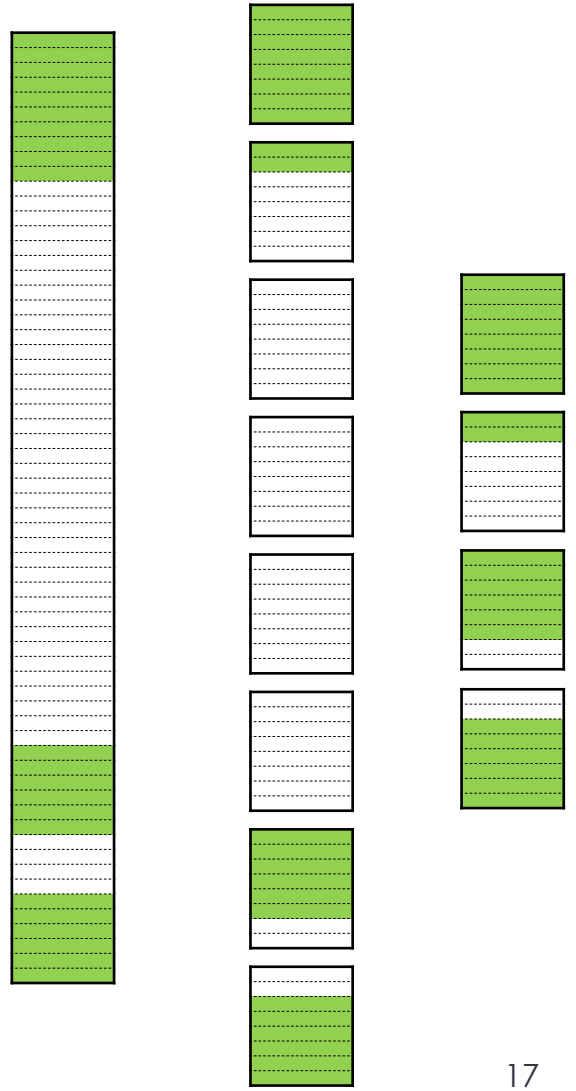
- Instead of making the actual size of a page table smaller, we want to **use less physical memory** to store the page tables. How?

- Within a page table, not all PTEs (Page Table Entries) refer to valid virtual pages. If the system just needs to allocate physical memory to used PTEs, that would consume less memory.

- The Crux
  - Is there any way just to store part of the page table in physical memory?

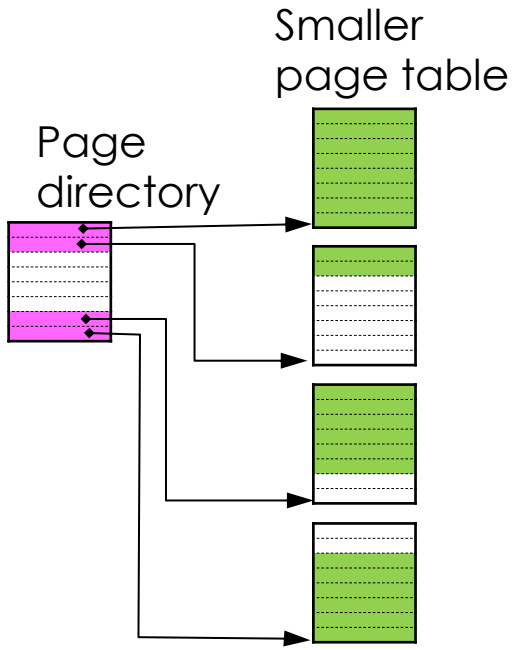
Multi-level Page Table

- **Chop up** the entire page table into **multiple page-sized** units
  - each unit is a part of the page table stored **in one frame**
- If an entire page of PTEs is invalid, do not allocate physical memory for that page
- System can place those small tables in discontiguous locations in physical memory
- The cruxes
  - How can we know whether a page unit contains valid pages?
  - How can we locate the "smaller" page tables?



Multi-level Page Table

- We need another table – **page directory** table
  - Tell us where a page frame of smaller page table is
  - Tell us whether the smaller table contains no valid pages
- If the entire page table is divided into *n* pages of smaller page tables, we need *n* entries in the page directory table
  - Each entry refers to information of a page of smaller page table
- Each page directory entry consists of a valid bit and a page frame number (which is the physical location of a valid **smaller page table**)



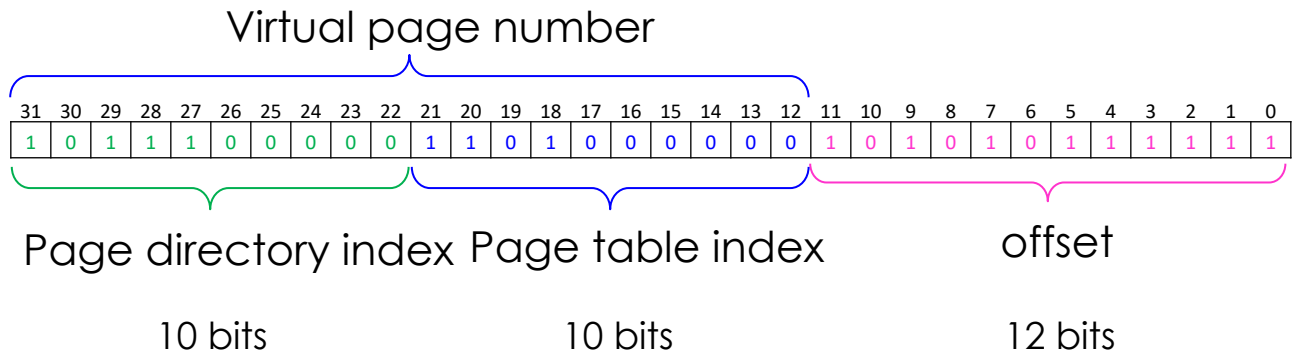
Performance Penalty

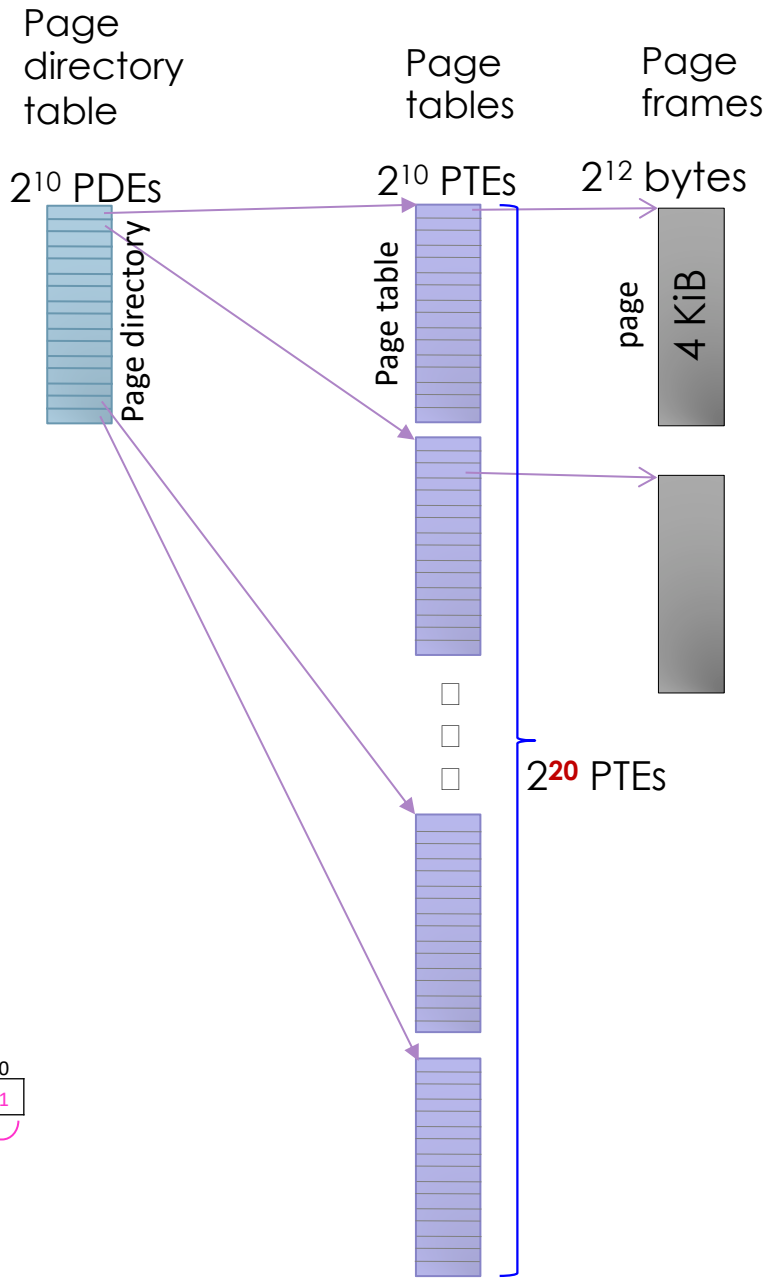
- A typical example of Time-Space Trade-off
  - We want to save valuable memory
  - However, we have one more level of indirection; that makes address translation more complicated

- If TLB hit, same as before
- If TLB miss, two extra memory accesses will be required to get the right translation information
  - **One for accessing the page directory** to find the physical address of the specific smaller page table
  - **One for access the smaller page table** to find the physical address of the requested page

Example

- A two levels scheme typically used by 32-bit address ,e.g. x86
  - Size of each virtual page = 2^{12} = 4 KiB
  - Total no. of PTEs in a linear page table = $2^{32}/2^{12} = 2^{20}$ PTEs
  - Assume each PTE is of 4-byte
  - Each frame can store $2^{12}/4 = 2^{10}$ entries (PTEs)
  - Entire page table is divided into $2^{20}/2^{10} = 2^{10}$ small page tables
  - We need 2^{10} entries (PDEs) in the directory table
  - MMU needs to know where to find the address of the page directory – ***page directory base register*** (PDBR)

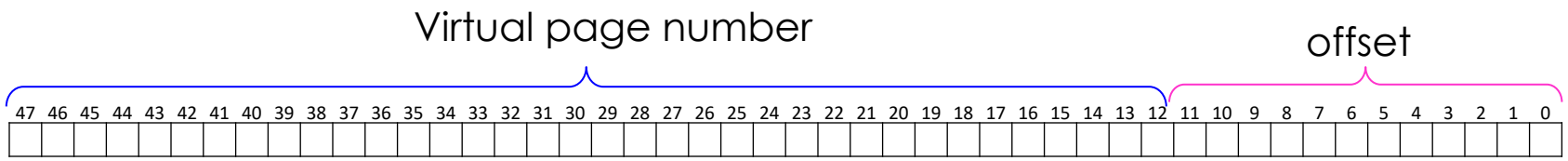




More Levels

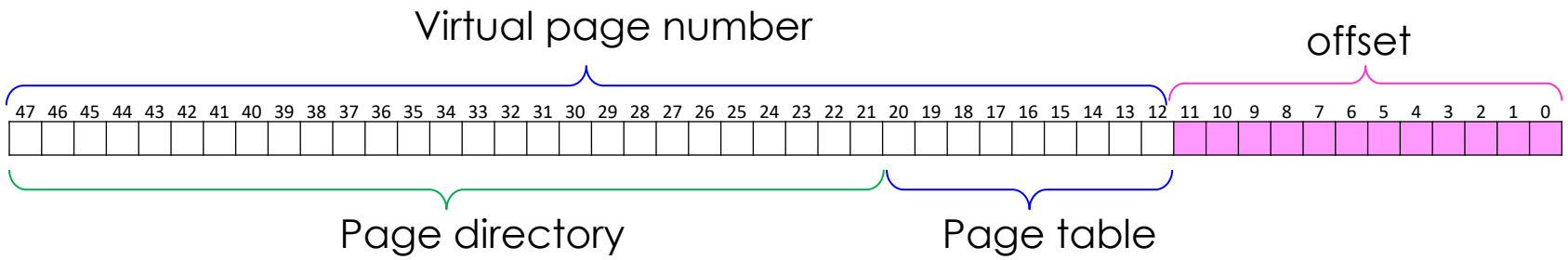
- Another example – IA32e
  - It can address **48** bits of process's virtual address space
  - Each table entry is of size **8-byte**
    - As it can access a larger physical address space, it needs more room to store physical address info
  - We keep using **4 KiB** virtual pages
  - How many levels of page tables do we need?
    - 4 levels

IA-32e with 4 KiB pages



- 4 KiB pages → 12 bits offset
- 48 – 12 = 36 bits for the virtual page number
  - Thus, there are 2^{36} PTEs

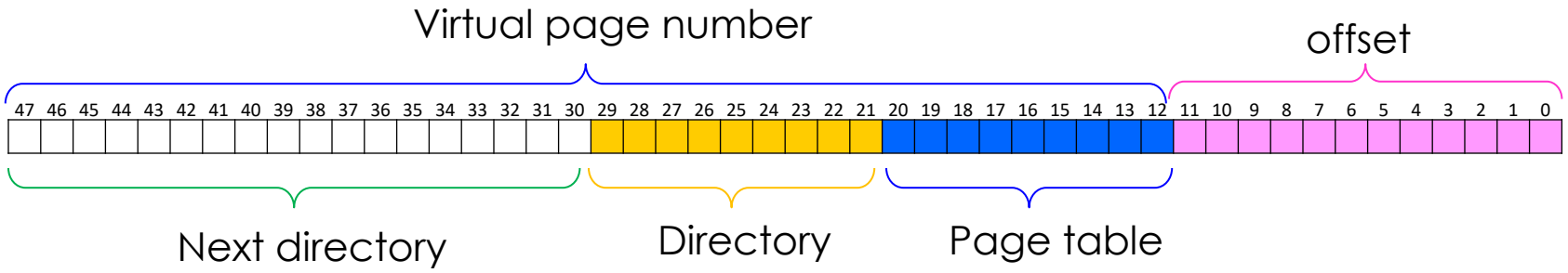
IA-32e with 4 KiB pages



- Number of PTEs in each page frame
- 4 KiB/8 bytes = $2^{12}/2^3 = 2^9$ entries

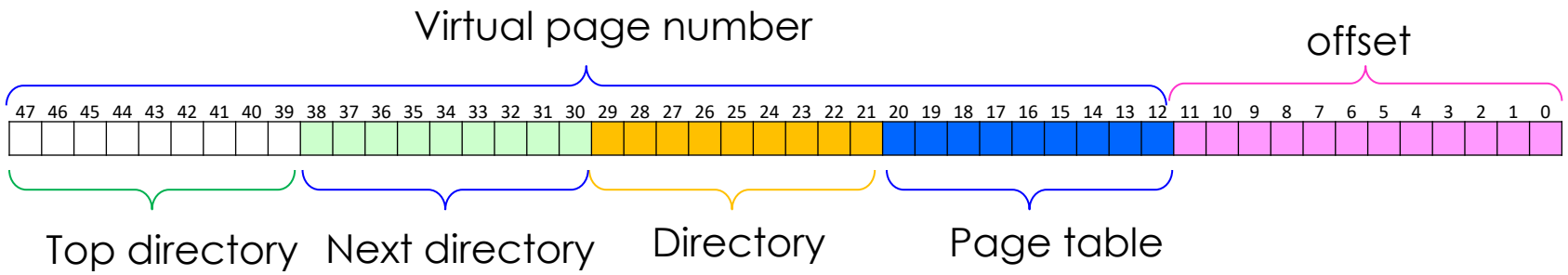
- How many smaller page tables (each in one page)?
- $2^{36}/2^9 = 2^{27}$ smaller page tables, each with 2^9 PTEs
- Thus, we need to have a page directory table with 2^{27} PDEs

IA-32e with 4 KiB pages



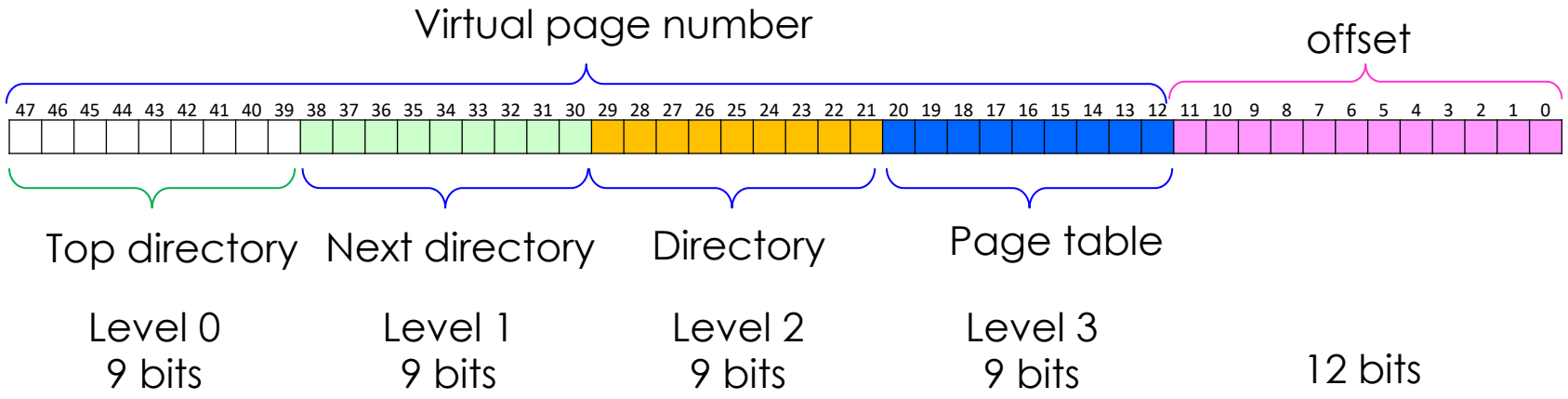
- How many pages do we need to store the Page Directory?
- We have 2^{27} PDEs; each page can store 2^9 entries
- $2^{27}/2^9 = 2^{18}$ pages
- Based on the same logic, we need to use another level of indirection to find where are the pages of page directory
- Thus, we need a **next level directory**, which consists of 2^{18} entries; each points to a smaller page directory that consists of 2^9 PDEs

IA-32e with 4 KiB pages



- How many pages do we need to store the Next Directory?
- We have 2^{18} entries; each page can store 2^9 entries
  - $2^{18}/2^9 = 2^9$ pages
- Thus, we need a top level directory, which consists of 2^9 entries; each points to a smaller next directory that consists of 2^9 PDEs

IA-32e with 4 KiB pages



- The top level directory has 2^9 PDEs, which is good enough to be stored in one single page frame.
- The physical address of the page frame storing this top level directory is kept in the page directory base register (PDBR).

Inverted Page Tables

- In 64-bit architectures, even with multilevel (can be more than 4 levels), the amount of memory consumed by the page tables can be substantial
  - One per process
- It is safe to assume that the amount of physical memory is often much smaller than the virtual address space

- "Inverted" relative to traditional page tables
  - The PTEs are indexed by the page frame number (PFN) instead of by virtual page number (VPN)
  - Each page frame has an entry in the inverted page table (rather than each virtual page in traditional page table)

Inverted Page Tables

- Given a virtual page, how can we search the inverted page table for the mapping information?
- To speedup virtual-to-physical translation
  - Uses hash functions to map virtual page to inverted PTE
- Collision
  - One or more virtual pages may hash to the same PTE
  - A linked list to chain up those pages that have the same hash value
- The PowerPC is one example of such an architecture

Summary

- By using a small but fast TLB to cache recently used address translation info, this greatly improves the performance of the paging system.
- Breaking the entire page table into page-sized units and only allocate physical memory to used smaller page tables – that significantly reduces the memory consumption
- The penalty of using multi-level scheme is the extra memory reference cost; each for each additional directory level

Operating Systems

- **Virtualization**
  - CPU Virtualization
    - Process Abstract
      - Address space
      - Process states
      - Process control block
      - Process operations API
      - Signals
    - Limited Direct Execution
      - System calls
      - Context switch
      - Interrupts
    - Scheduling
      - Scheduling metrics
      - FIFO, SJF, HRRN, STCF, RR, MLFQ
      - Multi-core scheduling, Linux CFS
  - Memory Virtualization
    - Address space
    - Address translation: dynamic relocation
    - Segmentation
    - Paging
    - TLB
    - Multi-level paging
    - Inverted page table
    - Swap space
    - Page replacement policy: FIFO, LFR, LRU, Clock
    - Thrashing
- **Concurrency**
  - Thread
    - POSIX threads (pthreads)
    - Race conditions, critical sections, mutual exclusion, atomic operations, synchronization
  - Locks
    - Atomic instructions: test-and-set, compare-and-swap
    - Mutex locks
  - Condition Variables
    - Pthread CVs
    - Producer-Consumer problem
  - Semaphores
    - Binary Semaphores
    - Counting Semaphores
    - Ordering
    - Readers-Writers problem
  - Deadlock
    - Dining philosophers' problem
    - Four necessary conditions
    - Deadlock prevention, avoidance, detection&recovery
- **Persistence**
  - I/O devices (HDD, SSD)
  - Files and Directories
    - Inode
    - File descriptor
    - Hard/Symbolic links
  - File System Implementation
    - On-disk data structure
      - Superblock, Bitmap, Inodes, Data blocks
    - Free space management
      - Bitmap, linked-list, block-list
    - Caching and buffering
    - Access control and protection
    - Journaling file system
      - Data journaling
      - Metadata journaling
- ***Advanced Topics***