CS250P: Computer Systems Architecture
Virtual Memory

Sang-Woo Jun

Fall 2023

UCI

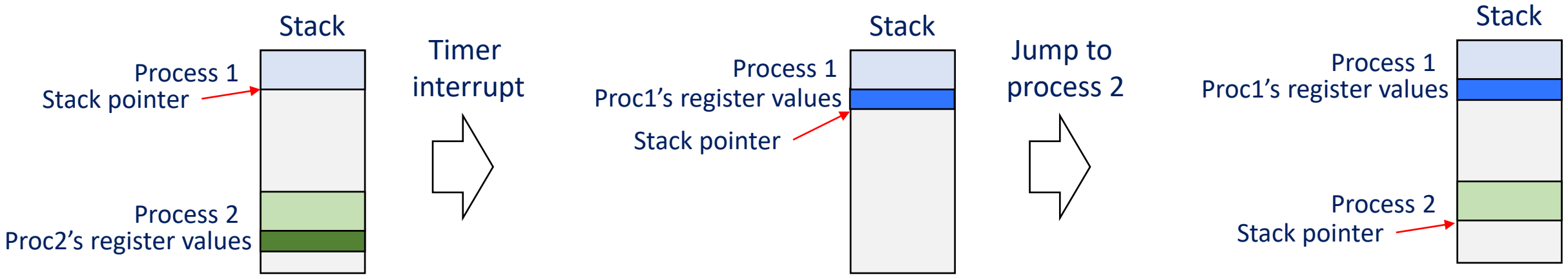
Large amount of material adapted from MIT 6.004, “Computation Structures”, Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”, and CS 152 Slides by Isaac Scherson

So far…

- Operating System goals:
  - Protection and privacy: Processes cannot access each other's data
  - Abstraction: OS hides details of underlying hardware
    - e.g., processes open and access files instead of issuing raw commands to disk
  - Resource management: OS controls how processes share hardware resources (CPU, memory, disk, etc.)
- Key enabling technologies:
  - User mode + supervisor mode w/ privileged instructions
  - Exceptions to safely transition into supervisor mode
  - Virtual memory to provide private address spaces and abstract the machine's storage resources **(today!)**

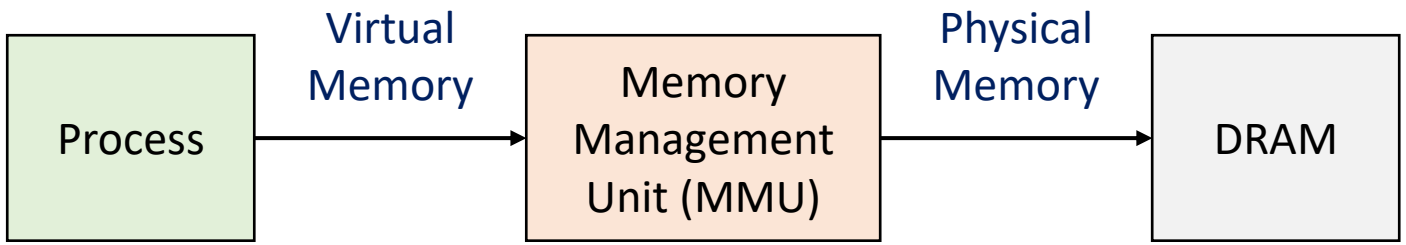
Remember context switching

- All running processes still share the same memory space
  - **Does not support isolation between processes**
  - e.g., If process 1 uses too much stack space, it will overrun process 2's stack space
  - e.g., If process 1 maliciously writes to random address regions, it can overwrite process 2's stack space



Isolation solution: Virtual Memory

- Illusion of a large, private, uniform store
- Virtual address
  - Address generated by a process (given to LW, SW, etc…)
  - Specific to a process's private address space
- Physical address
  - Actual address on the physical DRAM module
  - Only managed/visible to the operating system
- A hardware memory management unit (in the CPU, configured by OS) performs the translation



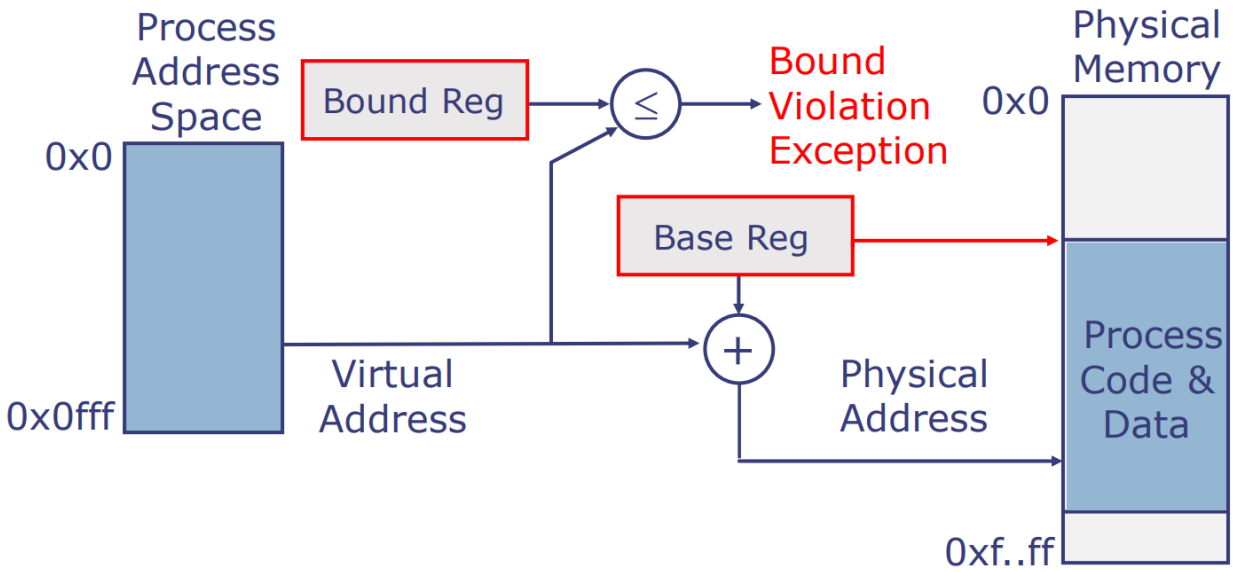
Isolation solution: Virtual Memory

- Each process can only interact with memory space via virtual memory
  - The OS controls which physical memory region is accessible by each process
  - Processes cannot access physical memory of other processes, or of OS
  - Each process generates read/write requests without worrying about safety

- Virtual memory also supports demand paging
  - When physical memory runs out, space is freed by moving some data to storage
  - This can be done transparently by the MMU
  - Provides the illusion of a very large memory (but with performance overhead)

An old solution: Segmentation

- A “base-and-bound” address translation
  - Each program’s data is allocated in a contiguous segment of physical memory
  - Physical address = Virtual Address + Segment Base
  - Bound register provides safety and isolation
  - Base and Bound registers should not be accessed by user programs (only accessible in supervisor mode)
- Often many register sets, separately for code, data, and stack segments
  - Protects buggy software from overwriting code, etc



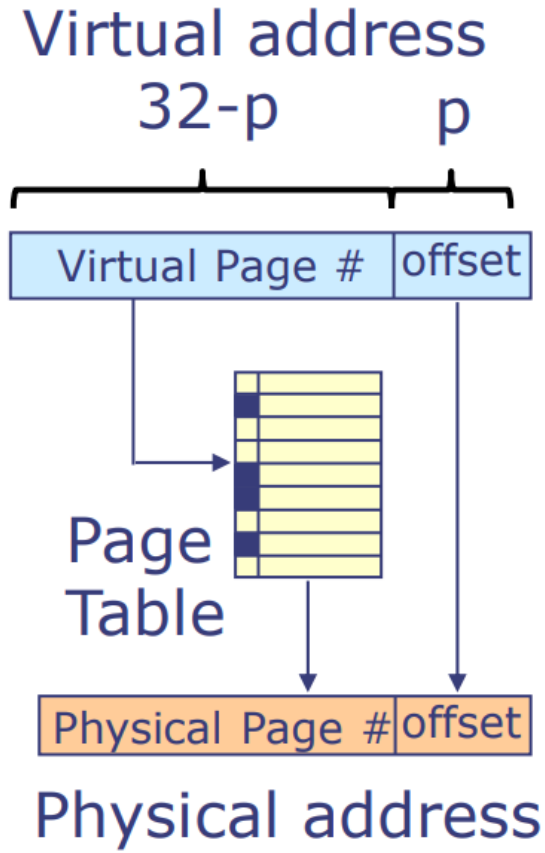
An old solution: Segmentation

- Requirement of contiguous memory space caused memory fragmentation
  - Small gaps between allocated process segments cannot be used efficiently
  - New processes cannot be allocated that space, because we don't know how much space it will need
  - Multiple segment registers sets (code, stack, etc) helped a bit, but not a fundamental solution
  - Sometimes required costly defragmentation/compaction to collect free space
- No longer used in most modern processors

One reason hardware context switching is not used in x86
Segment registers are still part of the hardware-defined context

Modern solution: Paged virtual memory

- Physical memory is divided into fixed-size units called pages
  - Most modern architectures use 4 KiB pages
- Virtual address is interpreted as a pair of <virtual page offset, and page internal offset>
- Use a per-process page table to translate virtual page offset to physical page offset
  - Page table contains the physical page number (i.e., starting physical address) for each virtual page



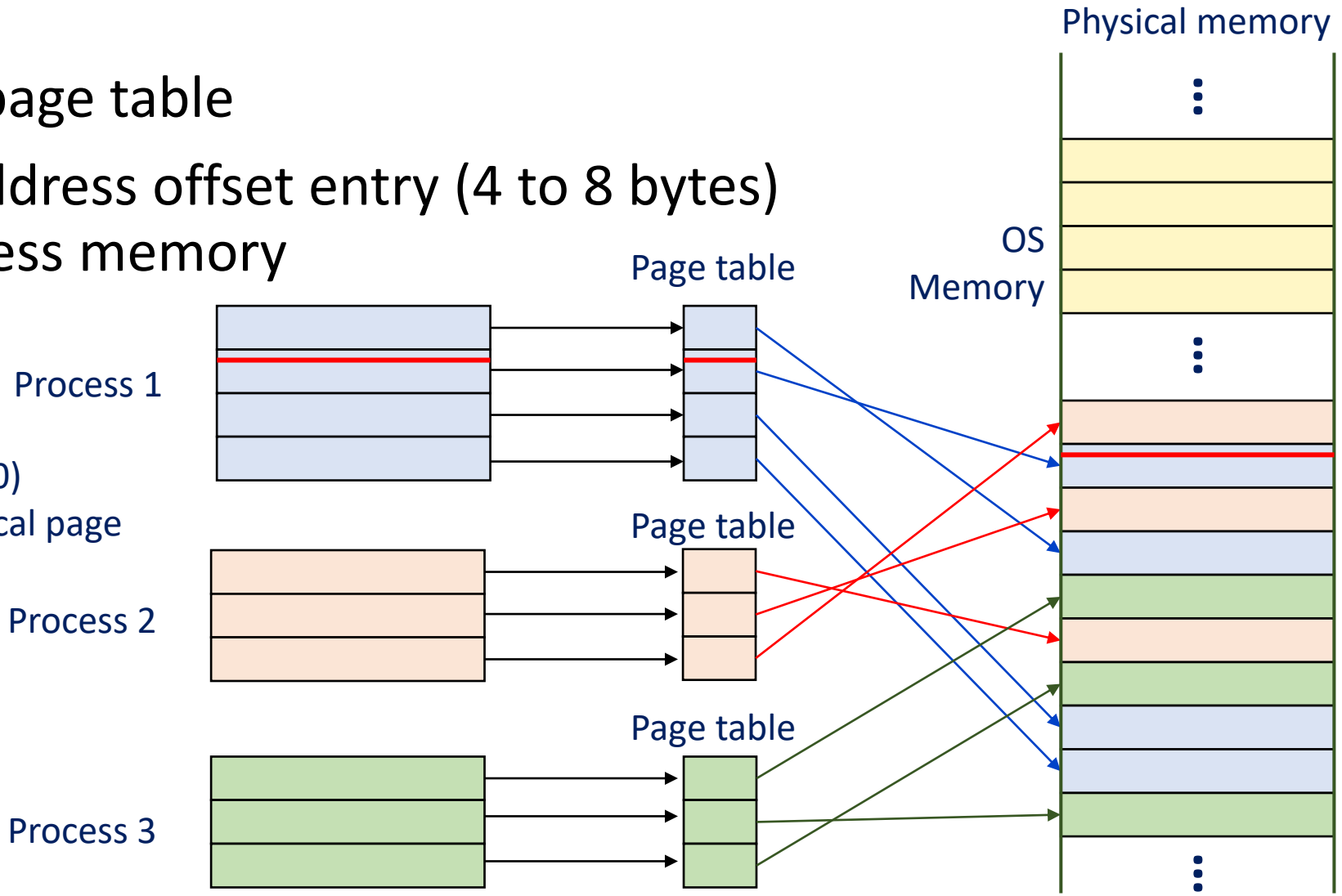
Who does this translation? Processor hardware, or OS software?

Private virtual address space per process

- Each process has a page table
- Page table has an address offset entry (4 to 8 bytes) for each page of process memory

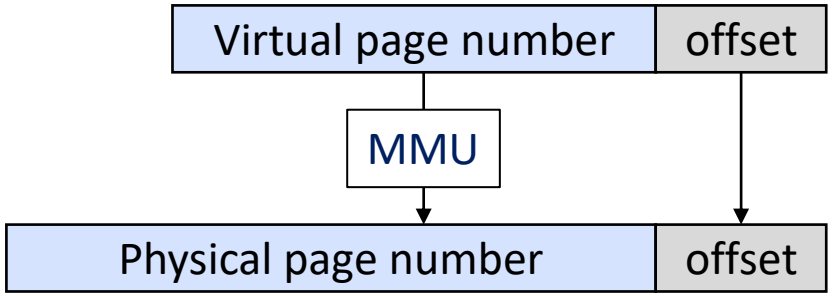
Example:
Process 1 issues write to **0x1100**
0x1100 is offset **0x100 in page 2** (0x1000)
Page table entry 2 is consulted for physical page

"Page table walk" handled in hardware by MMU



Virtual and physical address sizes

- The size of virtual and physical address spaces can be different!
  - Implementations may have larger physical address space than virtual
  - Since multiple processes share a physical memory
- Typically design decision regarding resource efficiency



Write protection

- A Page Table Entry (PTE) is typically augmented with metadata flags about each page
  - More flags -> Less space for actual address offset!
- One such flag is “Read-only” flag
  - Pages can be write protected by the OS
  - Attempting to write to a write protected page causes an exception
    - “General Protection Fault” in x86, “Access fault” in RISC-V
  - Program code pages are often write protected
    - Cannot be updated maliciously by code injection, etc

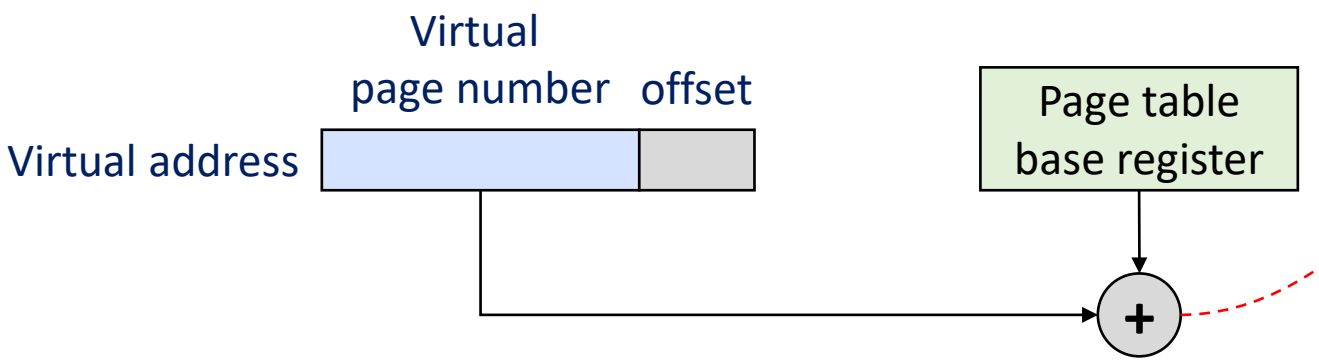
Paging vs. segmentation

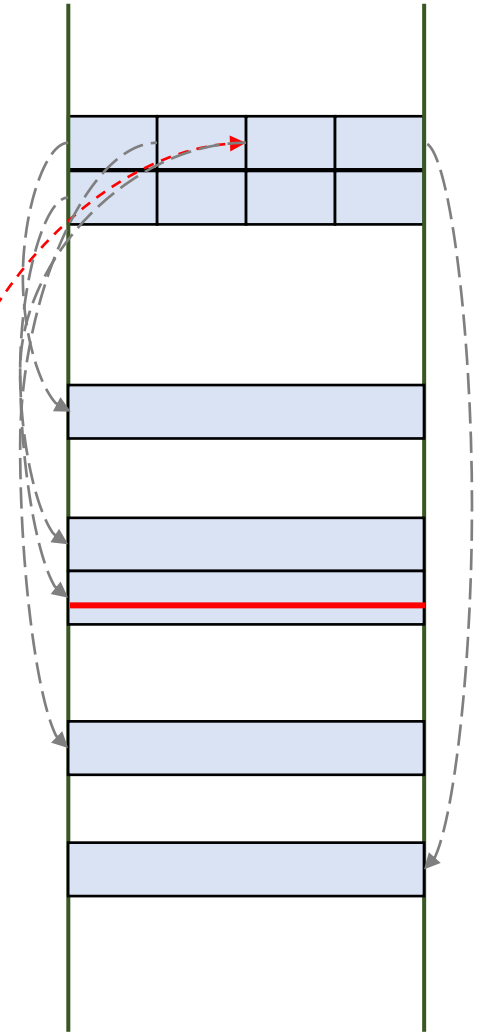
- Page table makes it possible to store pages non-contiguously
  - Drastically reduces fragmentation issue
  - Processes can be mapped to share some physical pages, often mapped to different virtual addresses

- But! Page table is MUCH larger than base+bound registers
  - For a 4 GiB address space, each process needs 4 GiB/4 KiB = 1 Million entries
  - Assuming 4 byte entries, 4 MiB per process
  - Base+bound registers were 2 * 4 bytes = 8 bytes per process!
  - No way 4 MiB per process can fit on-chip, meaning page tables must reside in off-chip DRAM

Page table implementation

- Physical page number
  = Mem[PT base + virtual page number]
- Physical address
  = Physical page number + offset
- Each cache miss (and DRAM access) involves **two physical memory access!**





Page table is entirely in physical memory space!

Eight great ideas

- ❑ Design for Moore's Law
- ❑ Use abstraction to simplify design
- ❑ Make the common case fast
- ❑ Performance via parallelism
- ❑ Performance via pipelining
- ❑ Performance via prediction
- ❑ Hierarchy of memories
- ❑ Dependability via redundancy

Translation Lookaside Buffer (TLB)

- Solution: Use some on-chip memory to cache virtual-to-physical address translations
  - TLB hit -> Immediate translation
  - TLB miss -> Must walk page table and populate TLB for later
- Similarly to data caches, TLB is:
  - indexed by a subset of the page number
  - tagged by the remainder of the page number
  - Involves trade-offs involving associativity
  - (Mostly) transparent to software
  - Often multi-level structure

e.g., x86 Haswell has L1 TLB of 64 items, 4-way, and L2 TLB of 1024 items, 8-way

Translation Lookaside Buffer (TLB)

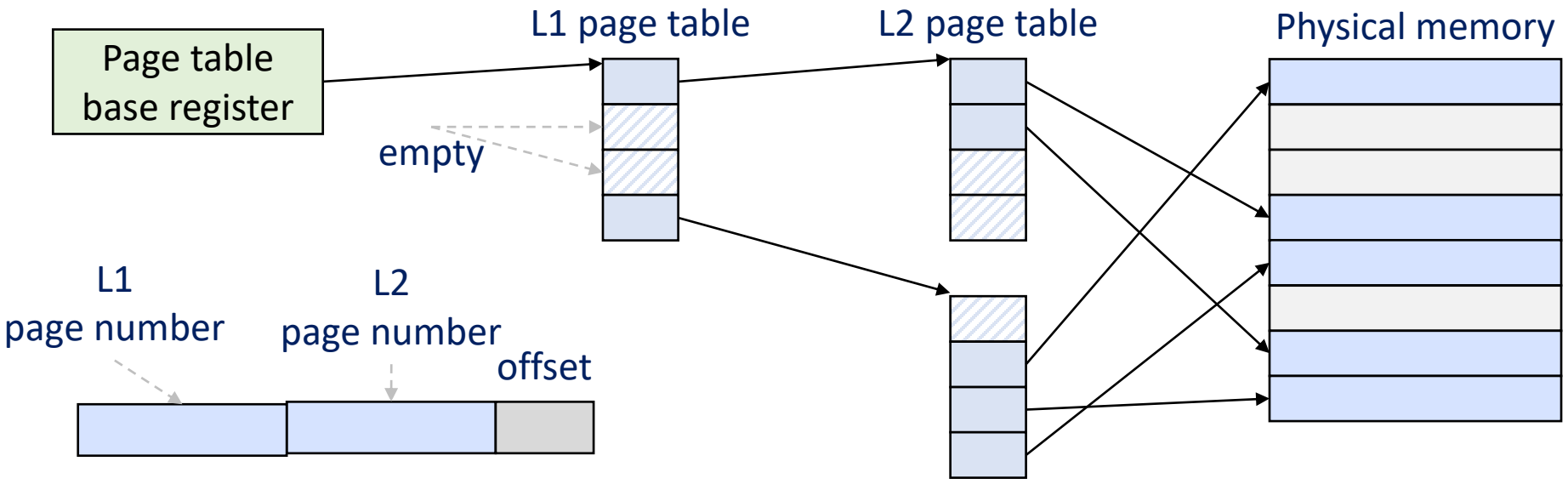
- Typical miss rates are very low
  - ~1% for typical applications and SPEC benchmarks
  - Page size (~4KiB) is much larger than cache lines (64 bytes), making good use of locality
  - But often over 20% for random access-intensive workloads like graph analytics
    - What does memory access (LW, SW, etc) look like for these applications?

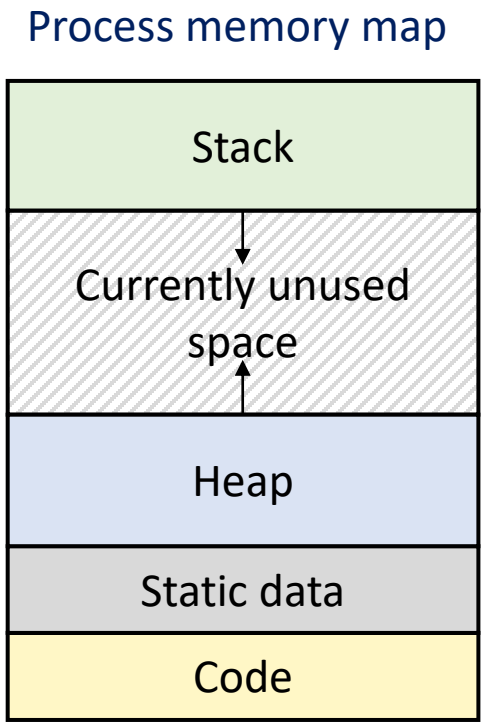
Another issue: Size of page tables

- Remember: For a 4 GiB address space, each process needs 4 GiB/4 KiB = 1 Million entries
  - Assuming 4 byte entries, 4 MiB per process
  - Per process! 256 concurrent processes consume 1 GiB just for the page table
  - “ps aux | wc -l” on my desktop tells me 261 processes are running…
- How do we reduce the size of page tables?
  - Larger page sizes? May waste memory capacity! (Fragmentation!)
  - Typical solution: Hierarchical page tables

Hierarchical page tables

- Most programs typically do not use all available virtual memory space
  - Meaning most of the page table is never used
- Hierarchical page tables exploit this to save space
  - Entries are only created when accessed
  - If L1 table entry is never accessed, no L2 page table created!





Hierarchical page tables

- Operating system must be aware of the hierarchy
  - Construct a level-1 page table per process
  - Unallocated virtual memory access causes an exception, and the exception handler creates level-2 page tables as required
  - Virtual memory access is a concerted effort btw OS/architecture!
- Multi-level page tables mean TLB misses result in more than two memory accesses
  - Importance of, and pressure on, TLB increases!

Aside: Real-world ISAs – RISC-V

- RISC-V defines three different virtual memory specifications
  - Sv32: Two level hierarchy, 32-bit virtual, 34-bit physical address
  - Sv39: Three level, 39-bit virtual, 56-bit physical address
  - Sv48: Four level, 48-bit virtual, 56-bit physical address
  - Up to chip designers to choose what to support
  - OS can learn which is implemented by reading the CSR “satp” (Supervisor Address Translation and Protection)
  - Page table base register is a CSR, “sptbr” (Supervisor Page Table Base Register)
- Some specs have different PTE sizes
  - Sv32 has 32 bit PTEs (34-bit physical address space – 11 bit offset < 32 bits)
  - Sv39 and Sv48 has 64-bit PTEs

Aside: Real-world ISAs – x86-64

- Complicated story, but currently…
  - Transitioning from 4-level page table to 5 levels (Intel Ice lake series)
  - 4 levels: 48-bit virtual address space, 46-bit physical address space
  - 5 levels: 57-bit virtual address space, 52-bit physical address space
  - 64-bit PTEs (Page index + many flags)
  - Page table base register is “CR3” (Control register 3)

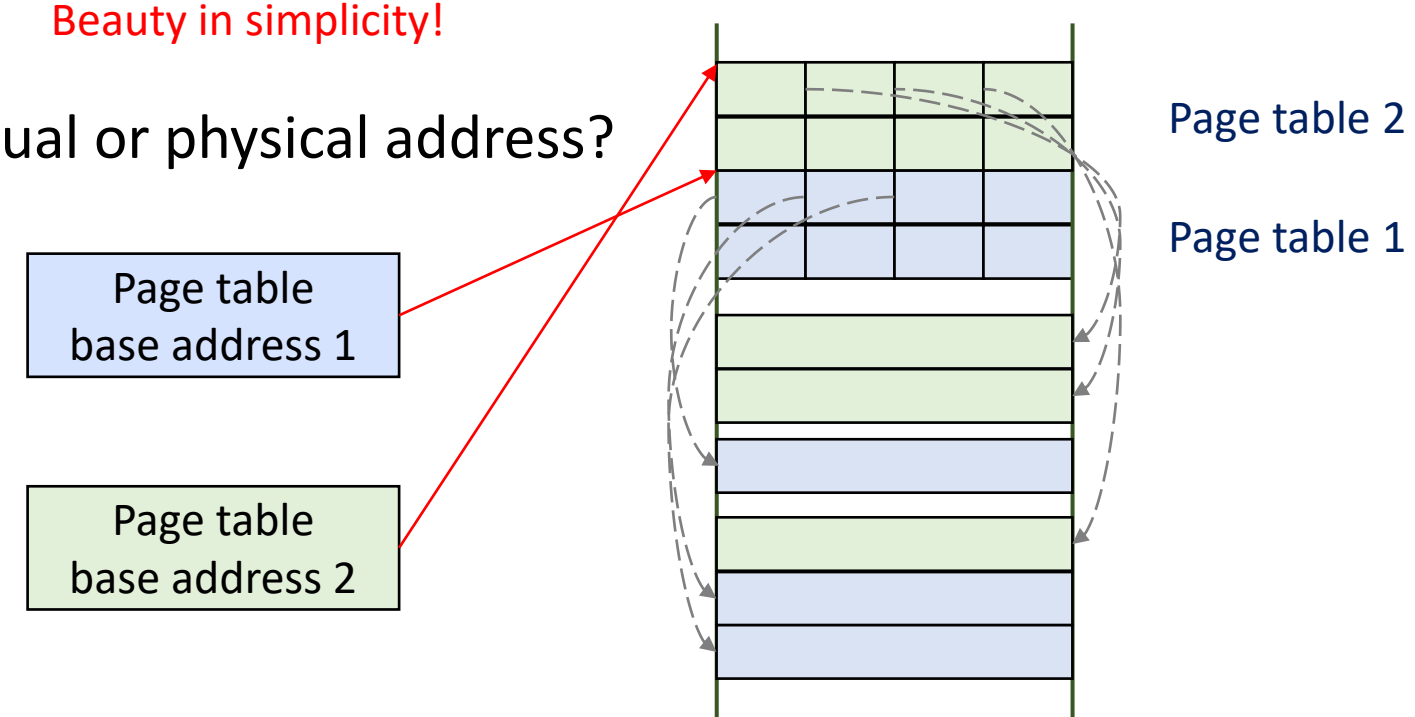
Virtual memory and multiprocessing

- Things seem simple enough so far
  - OS provides a multi-level page table to hardware MMU
  - MMU maintains a TLB to transparently speed up virtual memory translation
- Things become tricky with multiple concurrent processes

Context switching with virtual memory

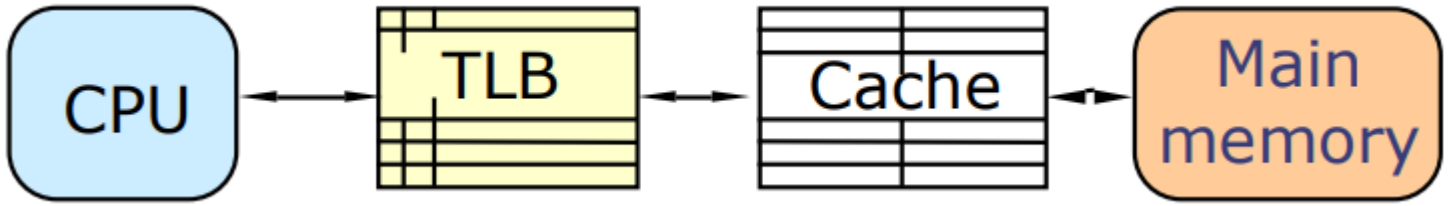
- Each process has a different private virtual memory
  - Achieved simply by considering the page table base register as part of the context
  - By restoring the contents of the base register during context switching, the processor will now walk the correct, corresponding virtual memory table
- But what about caches?
  - Do we index cache using virtual or physical address?

Beauty in simplicity!



Caching with multiprocessing

- Cache entries can either be based on virtual or physical address
  - Remember, caches simply return a value based on “address”. (virtual? physical?)
- If cache is based on physical address
  - Pros: simple to implement
  - Cons: Slow why?



“Physically Indexed, Physically Tagged (PIPT) Cache”

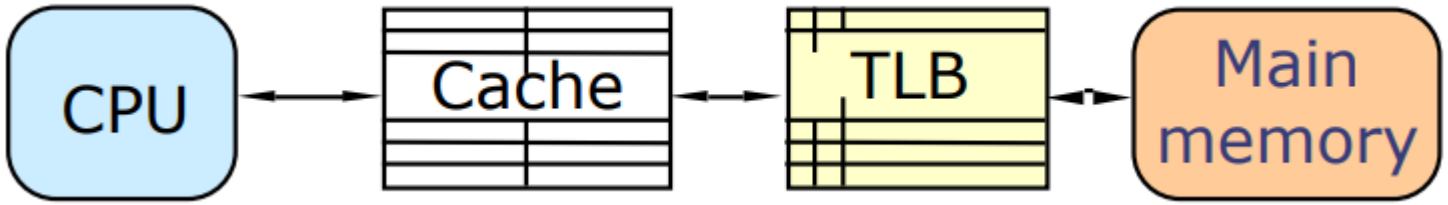
Physically Indexed Physically Tagged Cache

- PIPT caches are typically slow
  - We can't know the (physical) address to cache before going through virtual memory translation
    - Every cache access requires a virtual -> physical translation!
  - No cache benefits if TLB miss
  - TLB hits still result in performance loss because TLB lookup also has latency
- Benefits of PIPT
  - Simple to implement, as cache is unaware of virtual memory
  - Shared memory between processes are handled automatically
    - Same physical memory mapped to virtual memories of different processes…

Q: Is this a problem for caches larger than L1?

Virtually Indexed Virtually Tagged Cache

- If cache is based on virtual address
  - Cache hits are fast! No translation required before access
  - But, different processes can and will be using the same virtual memory address referring to different physical locations ("Homonyms problem")
    - Low performance is bad, but INCORRECT BEHAVIOR IS NOT ACCEPTABLE!
  - Naïve solution: Cache flushed after every context switch for correctness
    - Compulsory misses after every context switch… Performance hit!
  - Also, shared physical memory can be cached twice or more



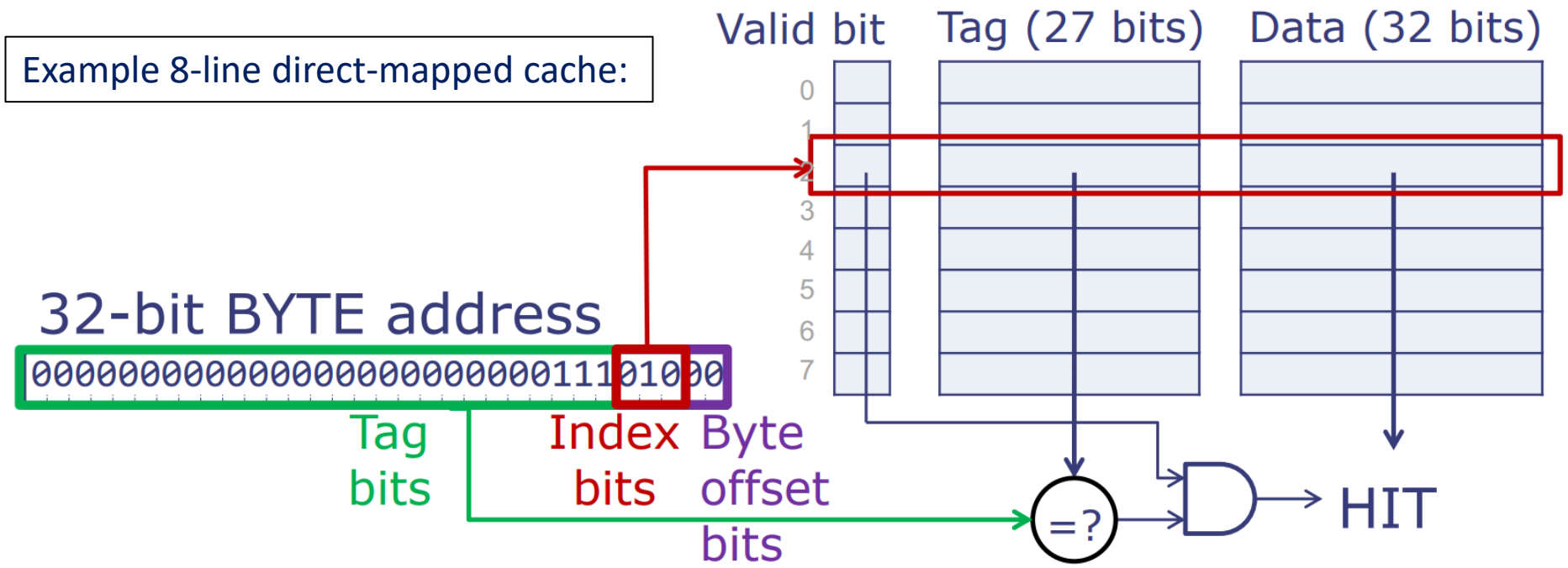
"Virtually Indexed, Virtually Tagged (VIVT)"

Virtually Indexed Virtually Tagged Cache

- One solution: “Address Space ID”
  - Virtual address can be augmented with a small, process-specific ASID (Address-Space ID), so that TLB entries from different processes can be distinguished
  - Pros: High performance!
  - Cons: Requires additional hardware and OS support
  - Cons: May limit number of concurrent processes
    - E.g., ARM sometimes uses 8-bit ASIDs -> 256 concurrent processes?!
    - With more, caches are invalidated, ASID allocation starts over (By Linux!)

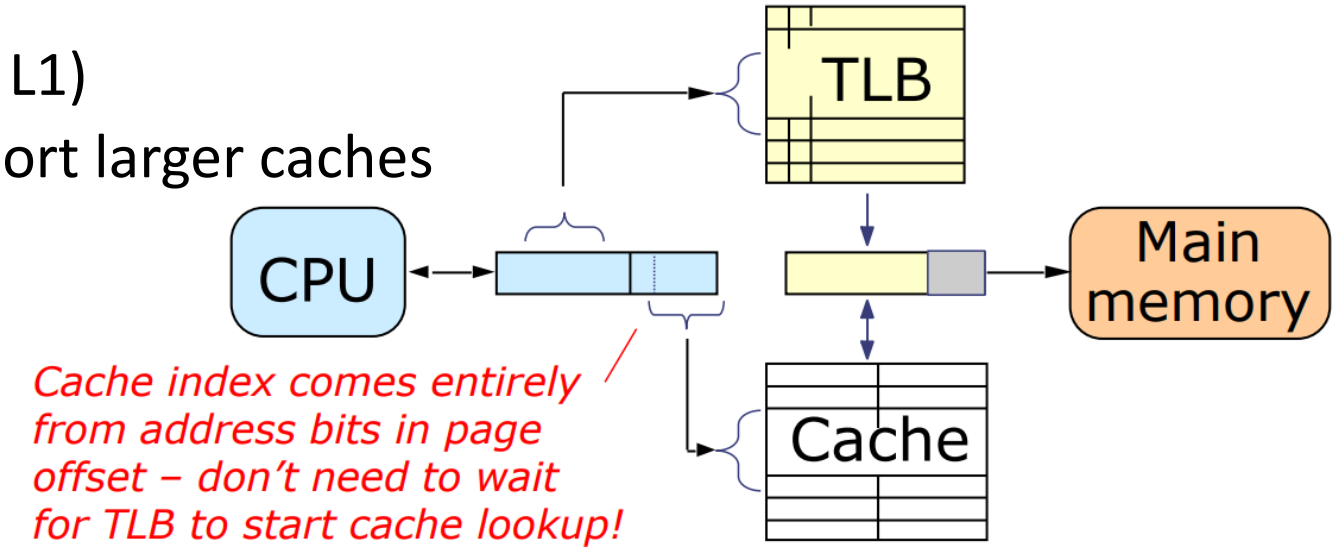
Can we do better?
Cache index and tags revisit

- Remember, cache entries are identified via index and tags
  - From different subsets of the address value



Mixed physical and virtual caching

- Another solution: mixing virtual and physical addressing
- Virtually-Indexed, Physically Tagged (VIPT) cache
  - **IF** (cache index bit width ≤ page offset bit width)
    TLB and cache lookups can be done in parallel
    - Cache lookup done only using page offset, which is not translated by virtual memory
      -> TLB lookup overhead is hidden!
  - Only works for small caches (e.g., L1)
  - Can increase associativity to support larger caches



Another problems: Synonyms

- What if two processes share a physical page?
- No issue for PIPT caches
- For VIVT or VIPT, the same physical location may be cached in two or more cache entries
  - Can’t keep track of data changes when processes write different things!
- General solution: Cache needs to ensure same physical address is not cached twice
  - Requires checking cache before populating it

Real world: x86 implementation

- Example: Intel Haswell (And many other Intel designs)
  - 32 KiB L1 instruction and data caches
  - 8-way set associative
  - 32 KiB ÷ 8 = 4 KiB sets… Does this number sound familiar?
  - (cache index == page offset)!
- Of course, different designs tried different approaches
  - Example: VIPT with larger caches, with low chance of homonyms
  - Speculatively execute assuming no homonyms, and rollback if homonyms discovered

Real world (2): Apple M1

- Apple M1 (2020) has 128 KiB of L1 data cache… How?
  - 4x the Intel size!
- M1's virtual memory page size is… 16 KiB
  - Intel's is 4 KiB (4x of intel!)
  - We see a pattern…!
- Still 8-way set-associative L1 caches!
  - Intel: 32 KiB ÷ (64 Bytes per line * 8-way) = 64 cache lines (sets)
  - Apple: 128 KiB ÷ (128 Bytes per line * 8-way) = 128 cache lines (sets)
  - We really don't want to have wider addressing for L1 cache! (propagation delay!)

Real world: L2 caches and onwards

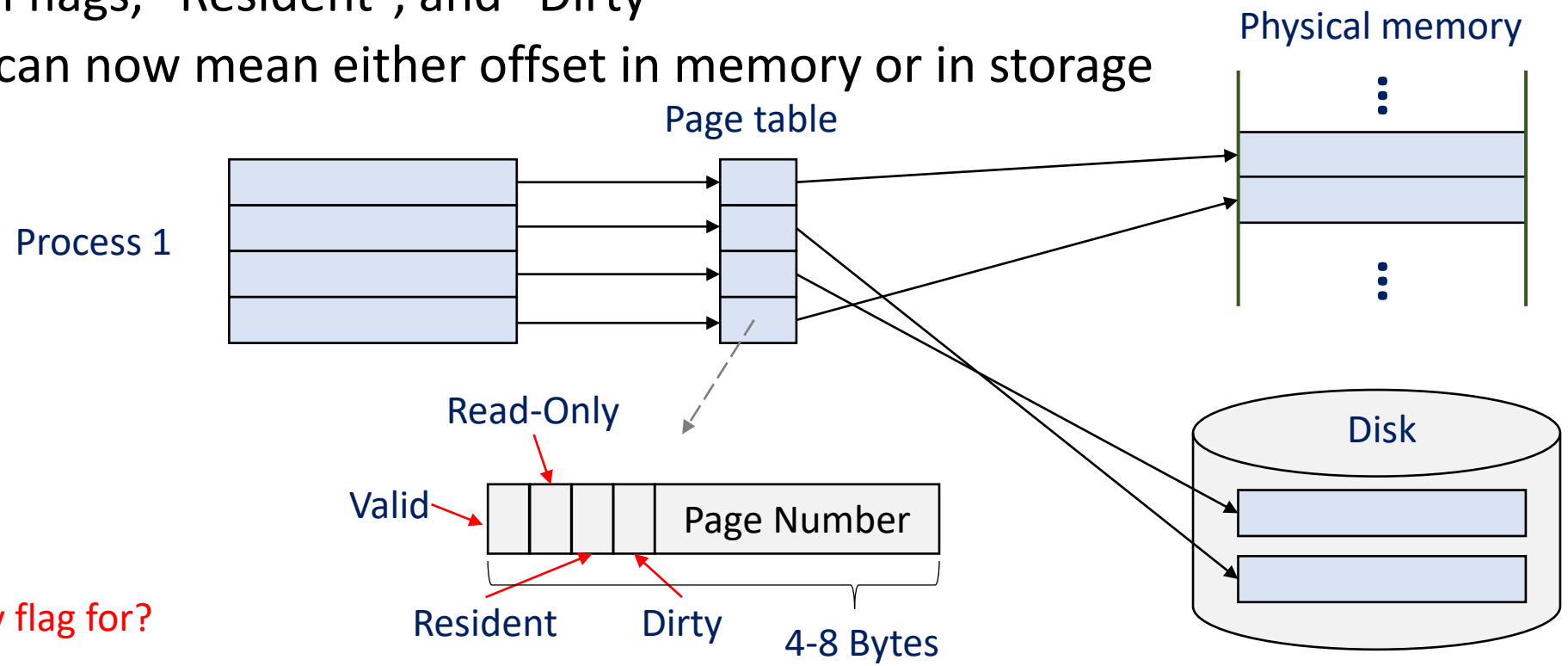
- L2 and later caches are typically all PIPT
  - Why? Are the issues with PIPT solved now?
- Yes! Because during VIPT, we already referenced the TLB
  - And potentially suffered the overhead of TLB misses

Virtual memory so far

- Efficient way to provide memory isolation between processes
  - Page-level mapping from virtual memory to physical memory per process
  - TLBs to transparently reduce page table lookups
- Virtualizing has another major benefit, "demand paging"
  - Some pages can be backed up to secondary storage, freeing DRAM for more pages
  - "Swap space" on storage used to "swap out" less-used pages
  - Provides illusion of very large memory capacity

Demand paging

- Now a page can be resident either in memory or in storage
- Page table entry needs additional information to handle this
  - Two additional flags, “Resident”, and “Dirty”
  - Page number can now mean either offset in memory or in storage



What is the dirty flag for?

Demand paging and caching

- Effectively, DRAM has become a cache for disk
  - But no associativity concerns, because the page table keeps track of the mapping instead of using lower bits of the address
    - Kind of like fully associative caches, no conflict misses!
- Should the cache be write-back, or write-through?
  - Remember: write through writes to the backing storage (disk in this case) whenever there is a write -> Performance limited by disk bandwidth
  - Write back: Only write to disk if the in-memory resident page is evicted, and if its contents have changed since the last time it was written
    - We need a “dirty” flag, just like write back caches!

Aside: Page eviction policy

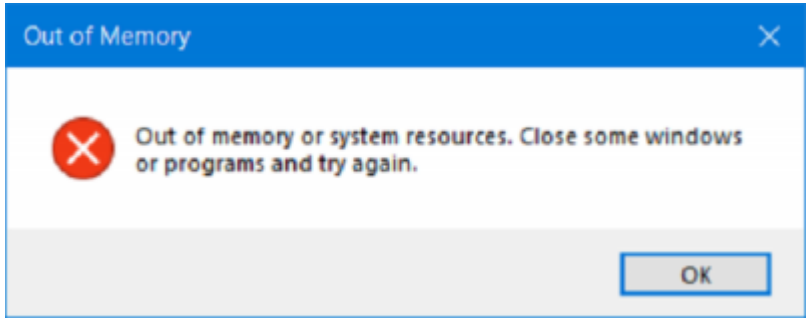
- When do pages get moved to storage?
  - Simple answer: when memory runs out (Kind of like caches!)
  - For example, malloc(), or if a swapped out page needs to be brought back in
  - This is an OS topic…

- How do we pick which pages will get moved to storage?
  - On-chip caches had pseudo-LRU
  - LRU would be ideal, but again pure LRU is not feasible
    - Don’t want to scan all page table entries every time
    - So, some form of pseudo-LRU is typically popular

Handling swapped pages

- What happens when a page is swapped out to disk, and a process wants to access it?
  - It will have to be transparently swapped back in, before servicing the memory request
  - Do we want the hardware to handle this, like page table walks?
- No, handling is left to the operating system software
  - Storage access requires a lot of logic for handling protocols, etc
  - Lots of other entities want to use storage, intelligent scheduling is complicated
- What happens when swap space is depleted?
  - Error message, and exit user program

Swap space typically configured statically by the OS

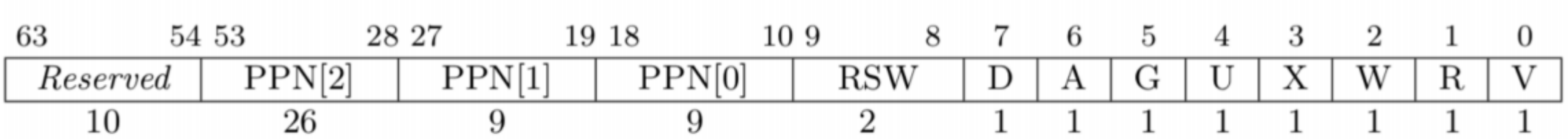


Hardware support for swapped pages

- Virtual memory access attempt into a non-resident page raises an exception (“Page Fault”)
  - MMU checks the page table whether the “Resident” flag is set
  - The OS page fault handler is invoked, which copies the page from storage to memory, sets the resident flag, and returns
  - The software can continue oblivious to swapping
- Effectively, the only hardware support is a new exception raised by MMU

Some more typical page table flags

- Some more flags include
  - “Executable (X)”, “readable (R)”, “user mode accessible (U)”
    - Extension of “read-only” flag, assigns restrictions on page access and raises an exception when violated
    - Memory is often divided into kernelspace and userspace
  - “Was accessed (A)”, …
    - For better LRU implementation

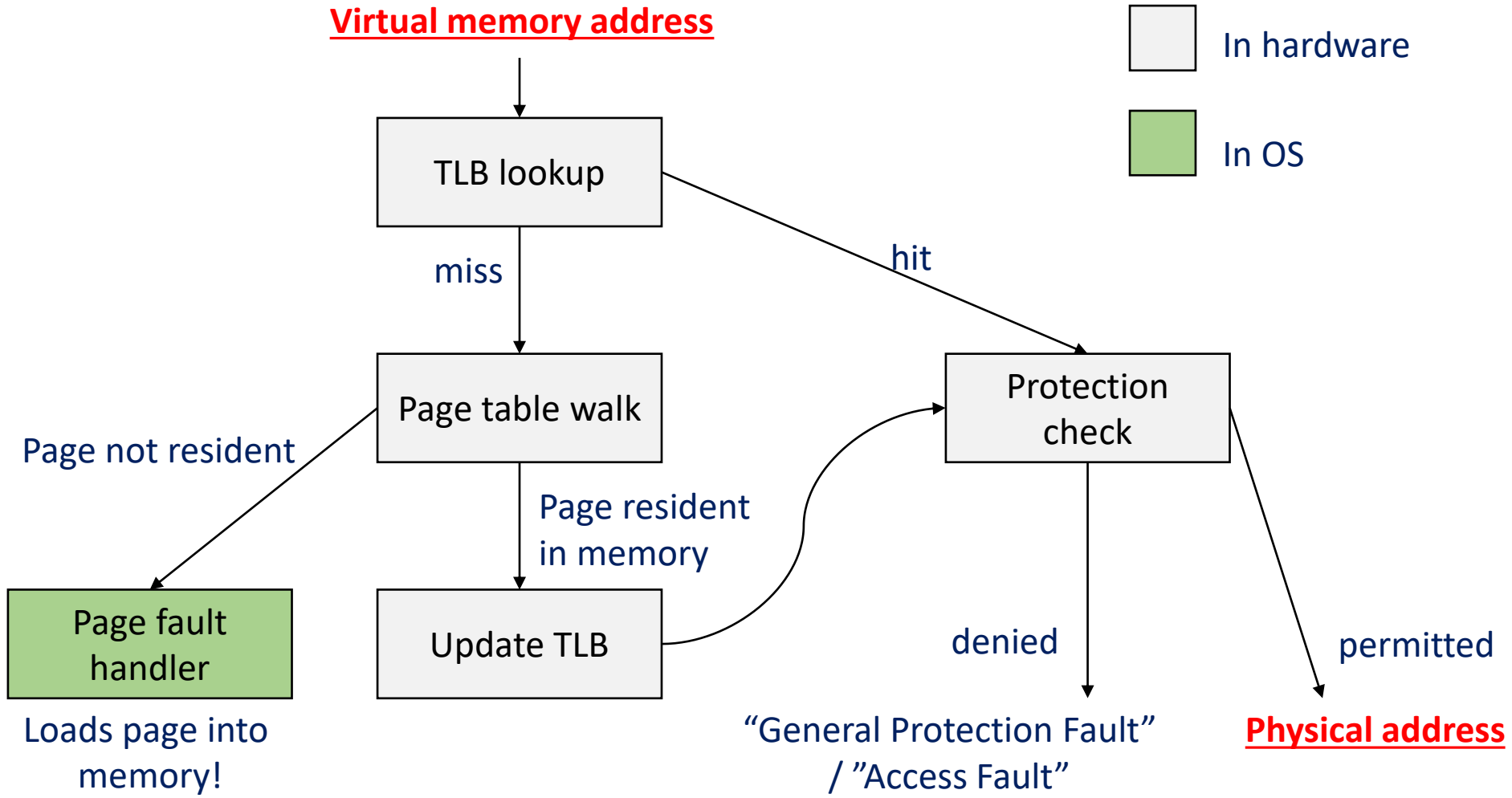


63–54	53–28	27–19	18–10	9–8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

RISC-V Example

Figure 4.18: Sv39 page table entry.

Address translation overview



Aside: Copy-on-write optimization

- When fork() is called on a process, it conceptually makes a copy of the entire process memory
  - Not all of the memory is used immediately
  - If the process is bash, for example, fork is called to execute another program, and copied memory is never used
- Optimization: copy-on-write
  - New process memory copies only page table, pointing to the same hardware memory
  - All pages are marked read-only for both processes
  - When write attempts raise access faults, only make copies then

Aside: On-demand storage access

- Example 1: When executing a program, not all parts of it may be used immediately
  - Don’t load all of the executable file contents immediately
  - Instead, create a page table and mark most of it non-resident
  - When access attempts raise page faults, only then read from storage
- Example 2: mmap()
  - mmap() returns a pointer, to which an area in storage is mapped
  - No data is actually read or copied initially, only page table entries marked non-resident
  - Read/write to the mmap’d pointer raises exceptions, which OS handles transparently

Page size and fragmentation

- Remember 1: We talked about making the page size larger to reduce the page table size
- Remember 2: Paging-based virtual memory removed the fragmentation issue suffered by the segmentation approach

- There is actually another type of fragmentation
  - External fragmentation: Gaps between blocks are not useful
  - Internal fragmentation: Space within blocks are not useful

If the page size is too large and access patterns are not friendly,
most of the loaded page may never be accessed (Internal fragmentation)
Not an efficient use of memory capacity