24. Virtual Memory: Design and Speed

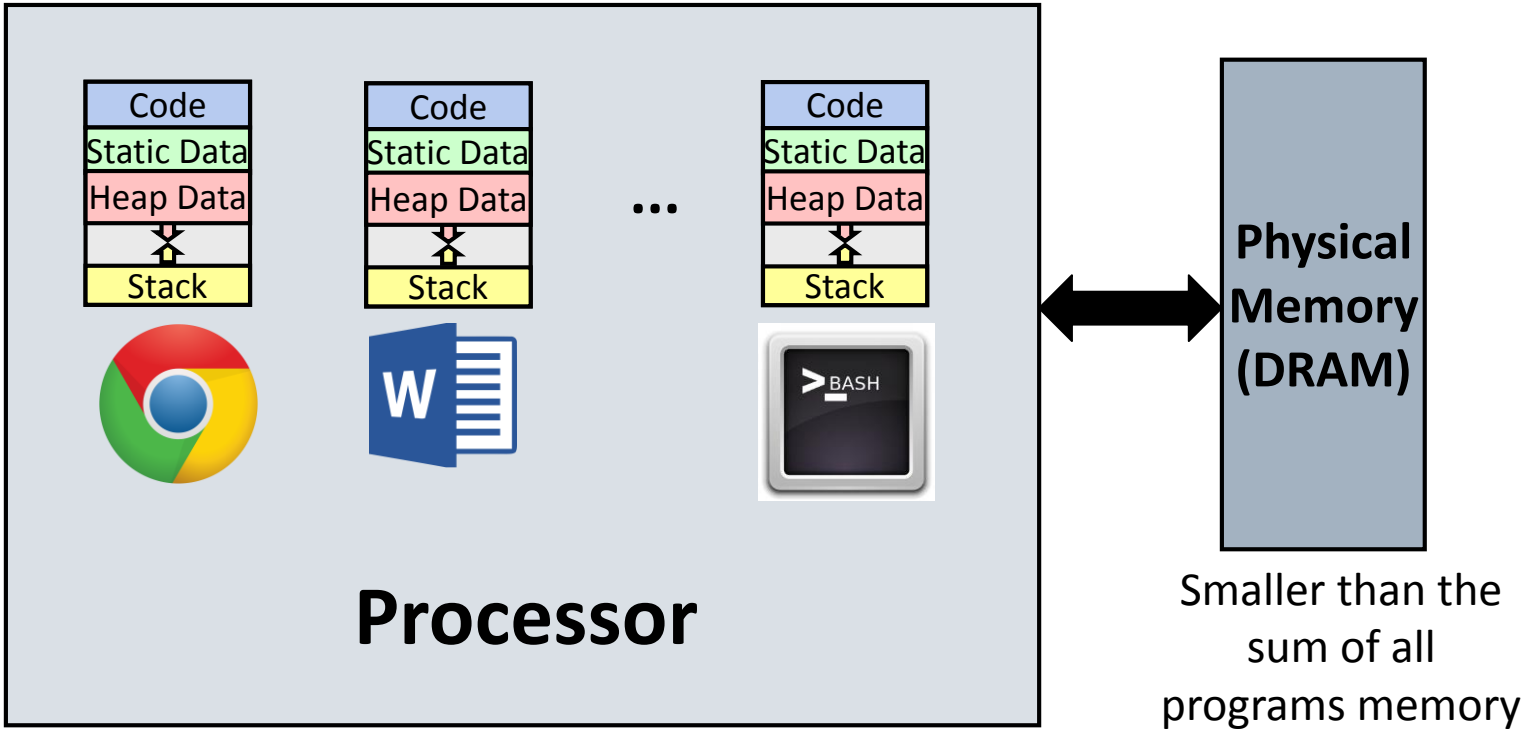
EECS 370 – Introduction to Computer Organization – Winter 2023

EECS Department
University of Michigan in Ann Arbor, USA

ANNOUNCEMENTS

- Project 4 due Thursday, April 13th
- Homework 6 due Monday, April 17th
- This is the last lecture with new material
- Review session on April 18th
  - No class this Thursday (April 13th)
- **Final exam** Thursday April 20th 10:30am-12:30pm

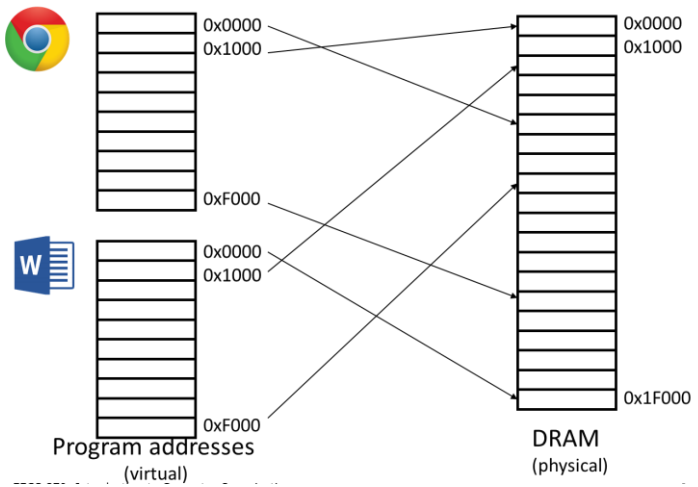
Review: Why Virtualize Memory?

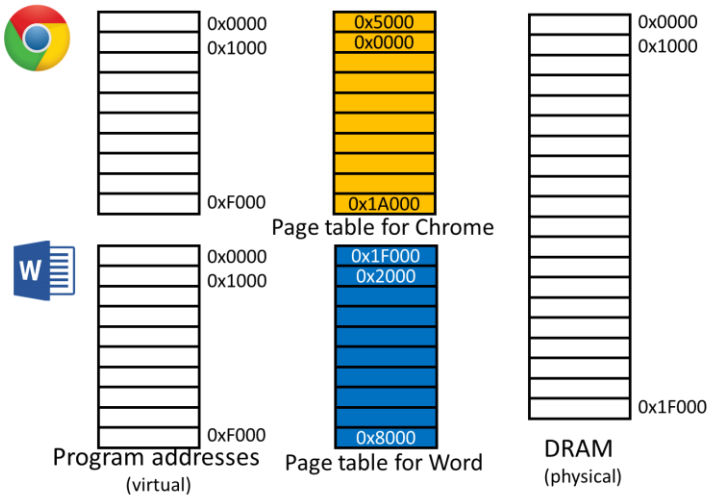


1. **Transparency**
   - Programs should not know that memory is virtualized
2. **Protection**
   - Programs should not be able to interfere with each other.
3. **Programs not limited by DRAM Capacity**

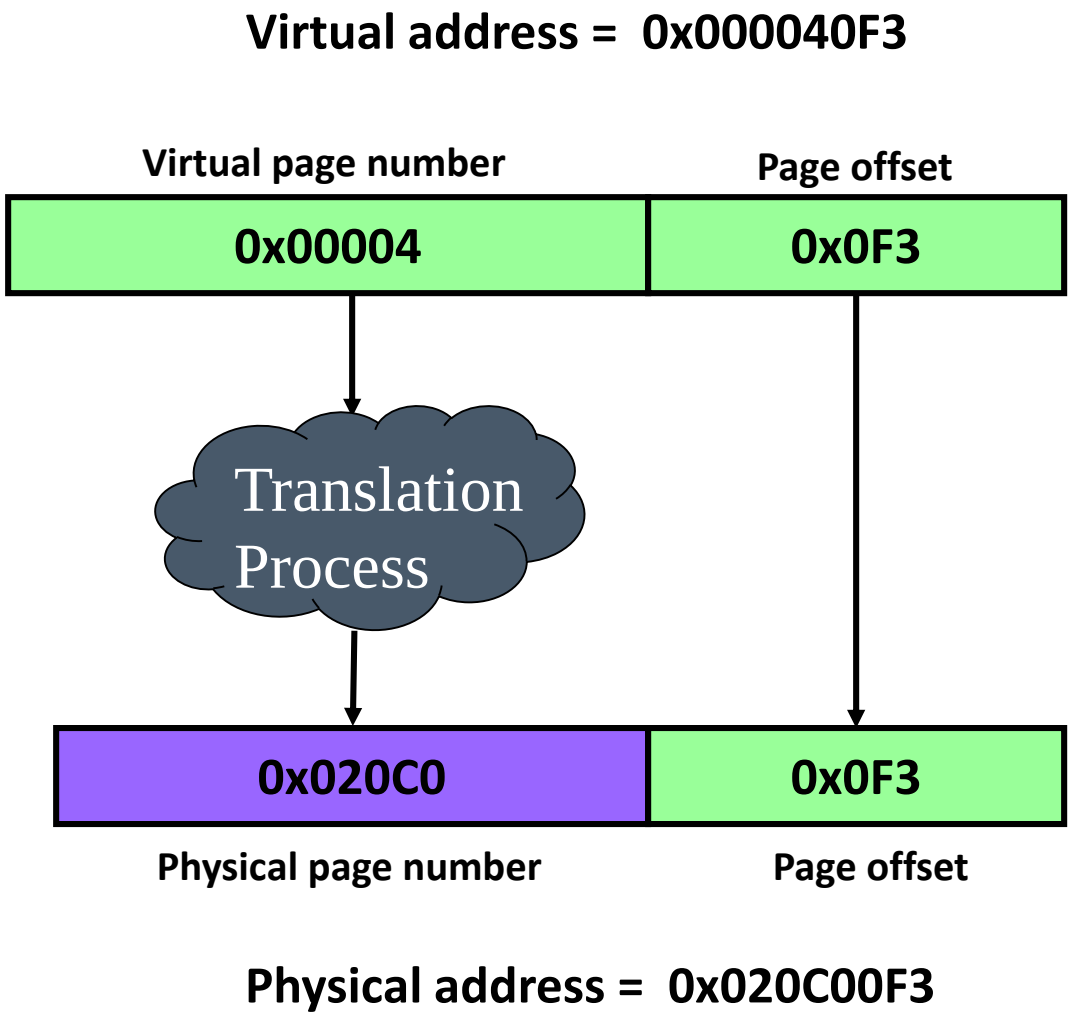
Goal: Given the Virtual Page, find the Physical page.

- Given a virtual address, we need find the data in DRAM
  - The location in DRAM is the *physical address.*
- We map "chunks" of the virtual addresses to physical addresses
  - We call these chunks "pages"
    - Pages are similar to a block in a cache.
- We have a table that lists the mapping of virtual pages to physical pages.
  - Called the page table.
- How to best structure the page table?



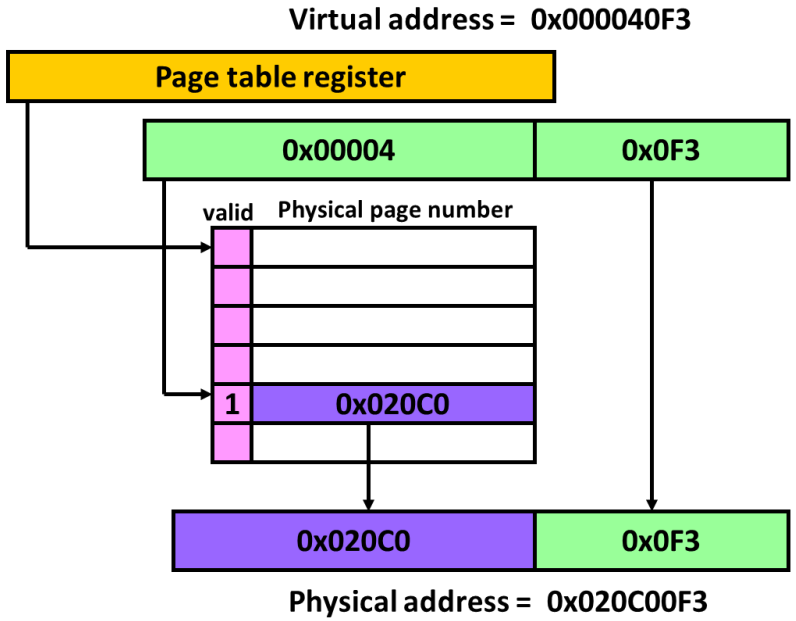


Review: Virtual to Physical Translation



How do we do translation?

- Option 1: Use an array-like structure.
  - Just have a big array indexed by the virtual address
  - Each entry of the array stores the physical page number (and some status bits like “valid”).



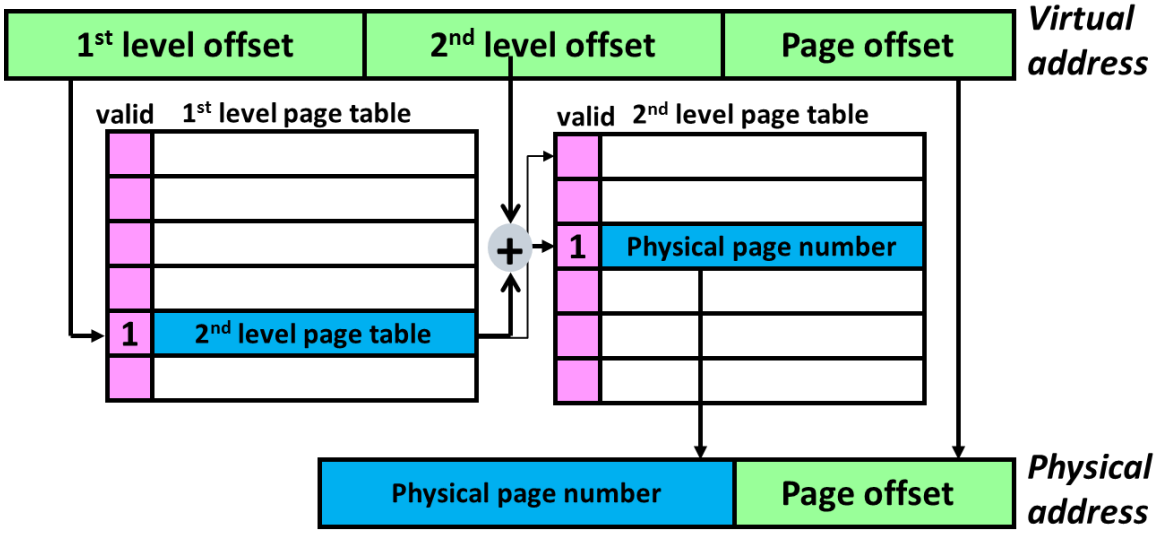
  - “Single-level page table.”

So what's wrong with a single-level page table?

- It's really big.
  - We need one entry per virtual page number that *could* be in use.
    - So with 32-bit virtual addresses and 12-bit page offsets (4KB pages), you need 2^{20} entries.
      - If each page table entry is 4 bytes, that's 4 Megabytes!
      - If we had 64-bit virtual addresses, we'd need 2^{52} entries.
        - That's just silly (that might be more DRAM than there is at UM!)
- But doesn't it need to be big?
  - I mean don't we need a mapping from every virtual address to a physical address?
  - The trick is that most processes only use a very (very) small fraction of available virtual memory.
    - So we'd like to only allocate space for the page table when it's needed.

How do we do translation?

- Option 2: Use a tree-like structure.
  - 1st level page table (*root of tree*) points to other page tables
    - Each entry contains the location of a 2nd-level page table (*leaves of tree*)
  - Only allocate 2nd level page tables *as needed*
    - 2nd level page table entries are identical to Option 1



- "Hierarchical page table"

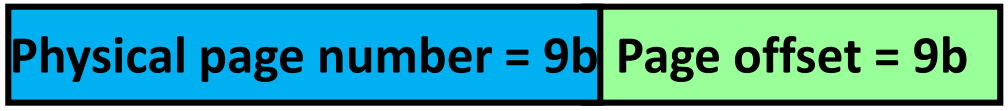
Flat vs Hierarchical

- Flat (single-level)
  - Pros:
    - One page table lookup
      - Aka one memory access *per translation*
  - Cons:
    - Always takes up a lot of memory
- Hierarchical (multi-level)
  - Pros:
    - Dynamically adjusts memory usage
    - Typically uses much less memory than single-level
  - Cons:
    - Two (or more) page table lookups
      - Aka two memory accesses *per translation*
  - *Used in most modern systems*

Exercise using previous multi-level VM

Virtual Address	1st level	2nd level	Page offset	Page fault?	Physical page num.	Physical Address
0x000F0C						
0x001F0C						
0x020F0C						

Virtual address = 24b



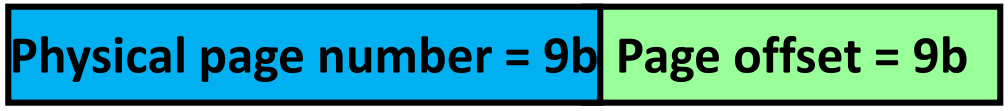
Physical address = 18b

Assume memory for page tables are “somewhere else” in memory

Exercise using previous multi-level VM

Virtual Address	1st level	2nd level	Page offset	Page fault?	Physical page num.	Physical Address
0x000F0C	0x00	0x07	0x10C	Y	0x000	0x0010C
0x001F0C						
0x020F0C						

Virtual address = 24b



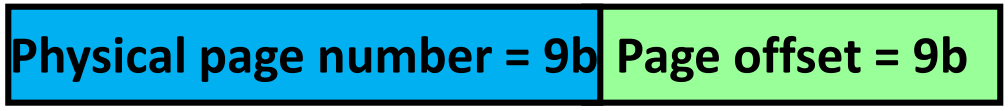
Physical address = 18b

Assume memory for page tables are "somewhere else" in memory

Exercise using previous multi-level VM

Virtual Address	1st level	2nd level	Page offset	Page fault?	Physical page num.	Physical Address
0x000F0C	0x00	0x07	0x10C	Y	0x000	0x0010C
0x001F0C	0x00	0x0F	0x10C	Y	0x001	0x0030C
0x020F0C						

Virtual address = 24b



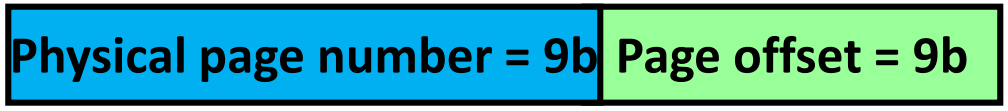
Physical address = 18b

Assume memory for page tables are “somewhere else” in memory

Exercise using previous multi-level VM

Virtual Address	1st level	2nd level	Page offset	Page fault?	Physical page num.	Physical Address
0x000F0C	0x00	0x07	0x10C	Y	0x000	0x0010C
0x001F0C	0x00	0x0F	0x10C	Y	0x001	0x0030C
0x020F0C	0x01	0x07	0x10C	Y	0x002	0x0050C

Virtual address = 24b



Physical address = 18b

Assume memory for page tables are “somewhere else” in memory

Page Replacement Strategies

- Page table indirection enables a fully associative mapping between virtual and physical pages.

- How do we implement LRU in OS?
  - True LRU is expensive, but LRU is a heuristic anyway, so approximating LRU is fine
  - Keep a "***accessed" bit per page***, cleared occasionally by the operating system. Then pick any "unaccessed" page to evict

Other VM Translation Functions

- Page data location
  - Physical memory, disk, uninitialized data
- Access permissions
  - Read only pages for instructions
- Gathering access information
  - Identifying dirty pages by tracking stores
  - Identifying accesses to help determine LRU candidate

Performance of Virtual Memory

- To translate a virtual address into a physical address, we must first access the page table in physical memory

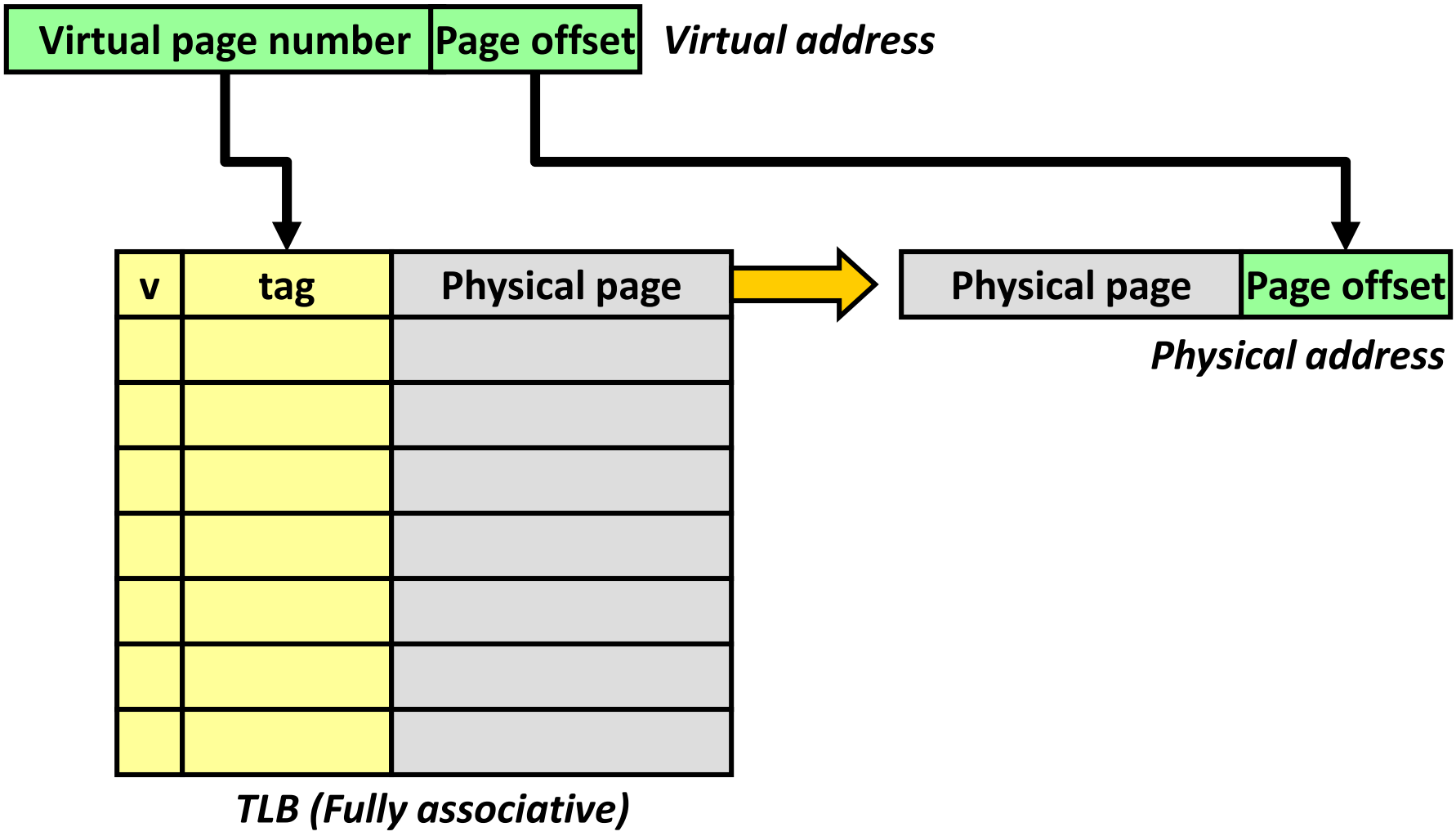
- Then we access physical memory again to get the data
  - A load instruction performs at least 2 memory reads
  - A store instruction performs at least 1 read and then a write

- Above lookup steps are ***Slow***

Translation look-aside buffer (TLB)

- We fix this performance problem by avoiding main memory in the translation from virtual to physical pages.
- Buffer common translations in a **Translation Look-aside Buffer (TLB)**, a fast cache dedicated to storing a small subset of valid page table entries.
- 16-512 entries common.
- Generally has low miss rate (< 1%).

Translation look-aside buffer (TLB)



Where is the TLB lookup?

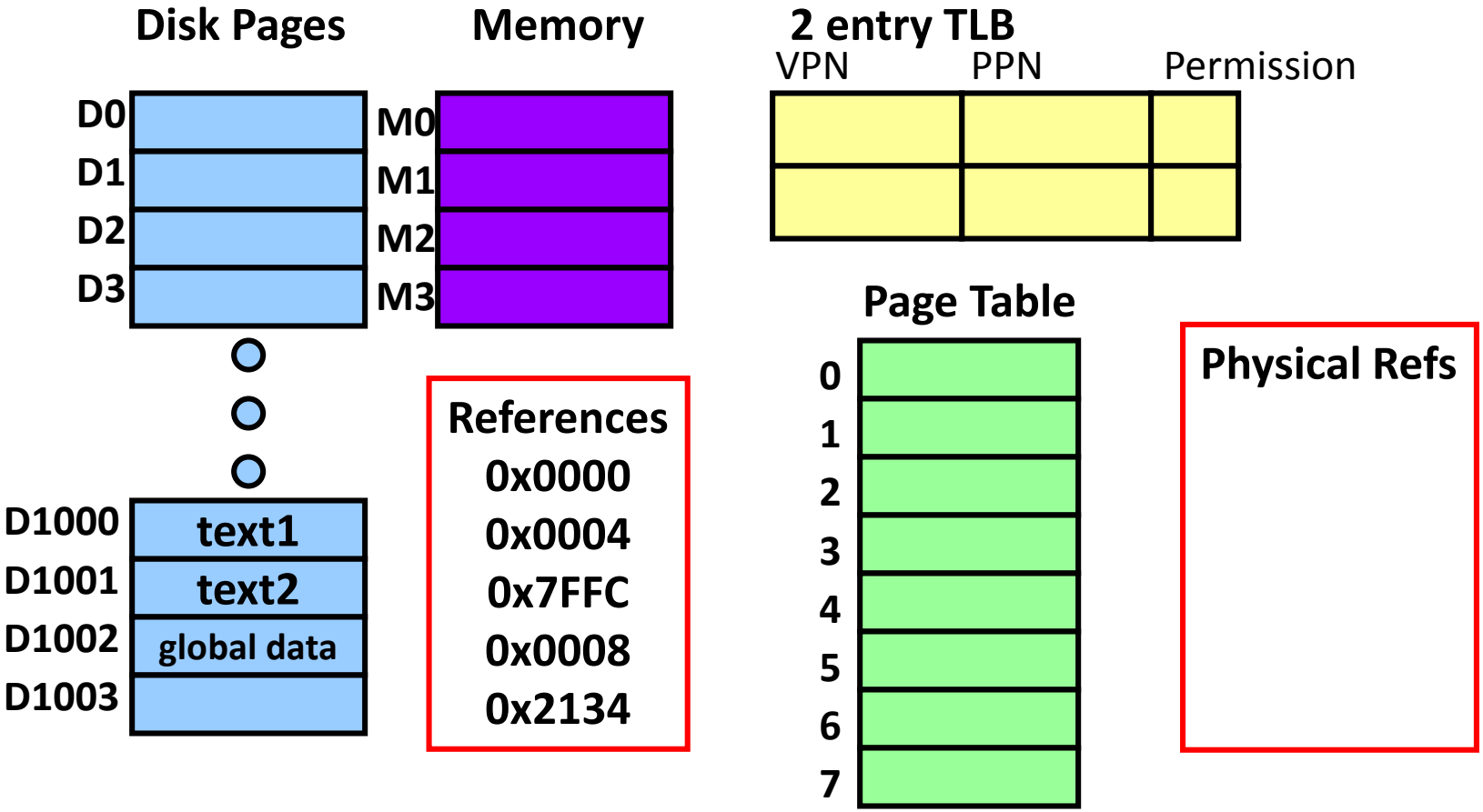
- We put the TLB lookup in the pipeline after the virtual address is calculated and before the memory reference is performed.
  - This may be before or during the data cache access.
  - In case of a TLB miss, we need to perform the virtual to physical address translation during the memory stage of the pipeline.

Putting it all together

- Loading your program in memory
  - Ask operating system to create a new process
  - Construct a page table for this process
  - Mark all page table entries as invalid with a pointer to the disk image of the program
    - That is, point to the executable file containing the binary.
  - Run the program and get an immediate page fault on the first instruction.

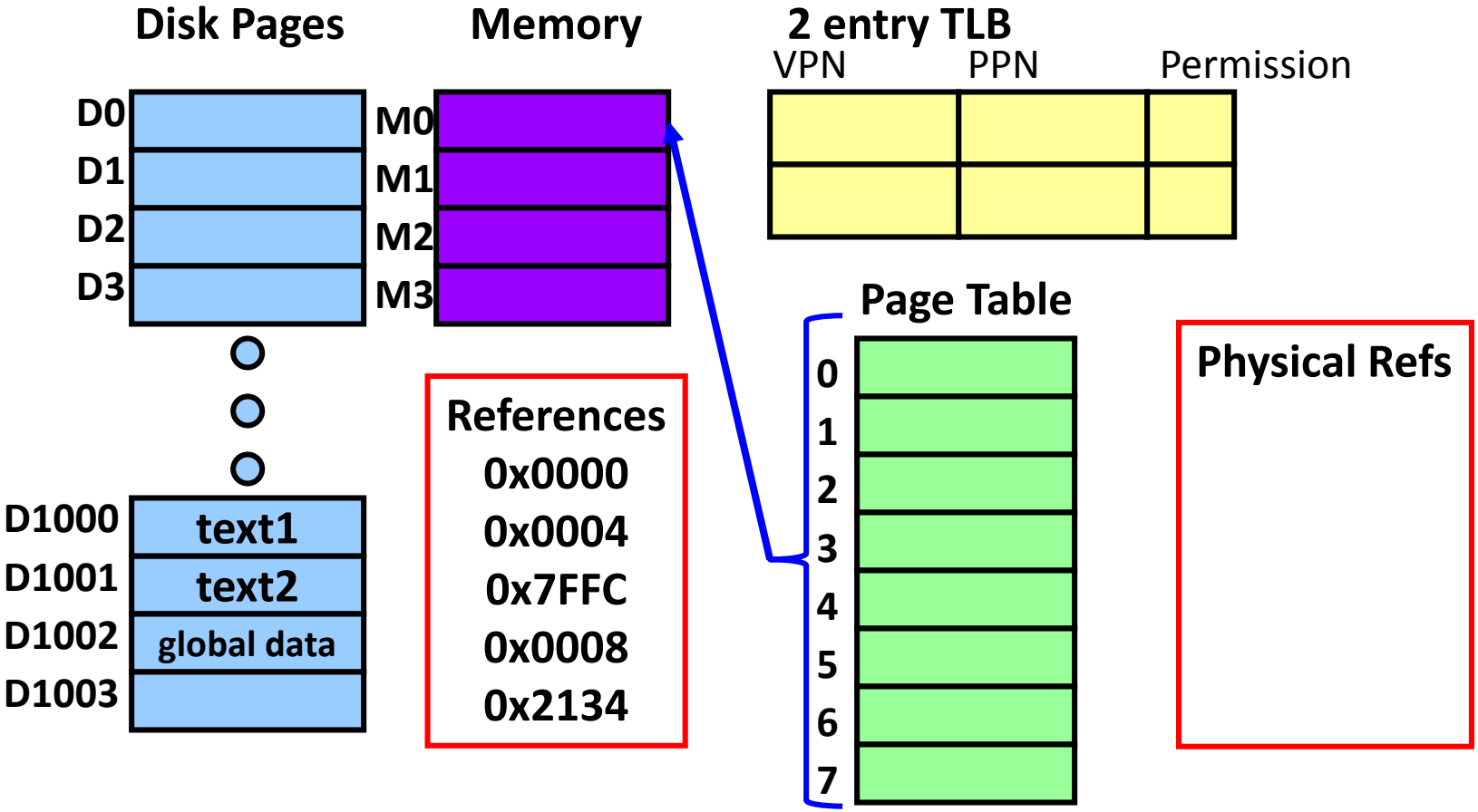
Loading a program into memory

- Page size = 4 KB, Page table entry size = 4 B
- Page table register points to physical address 0x0000

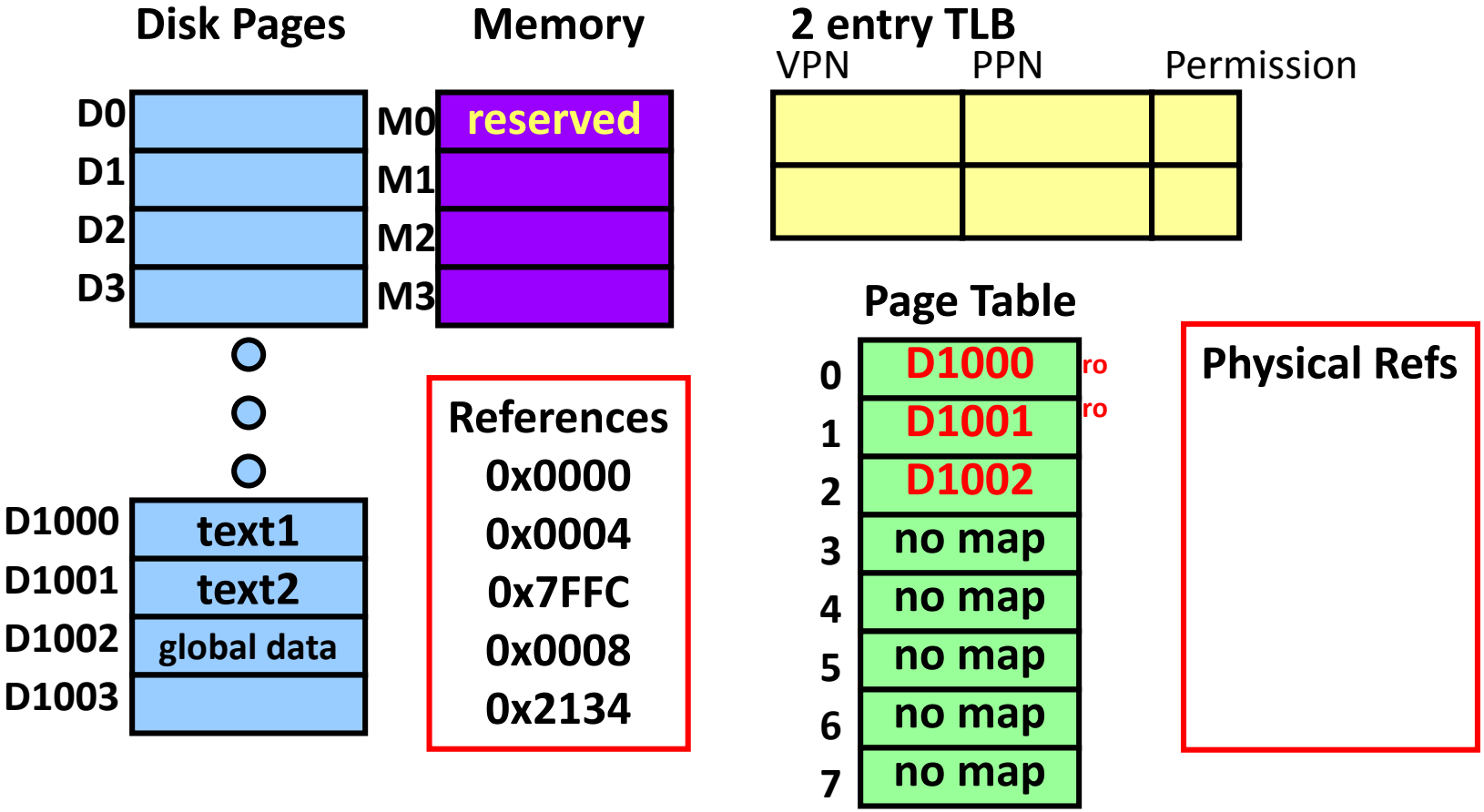


Loading a program into memory

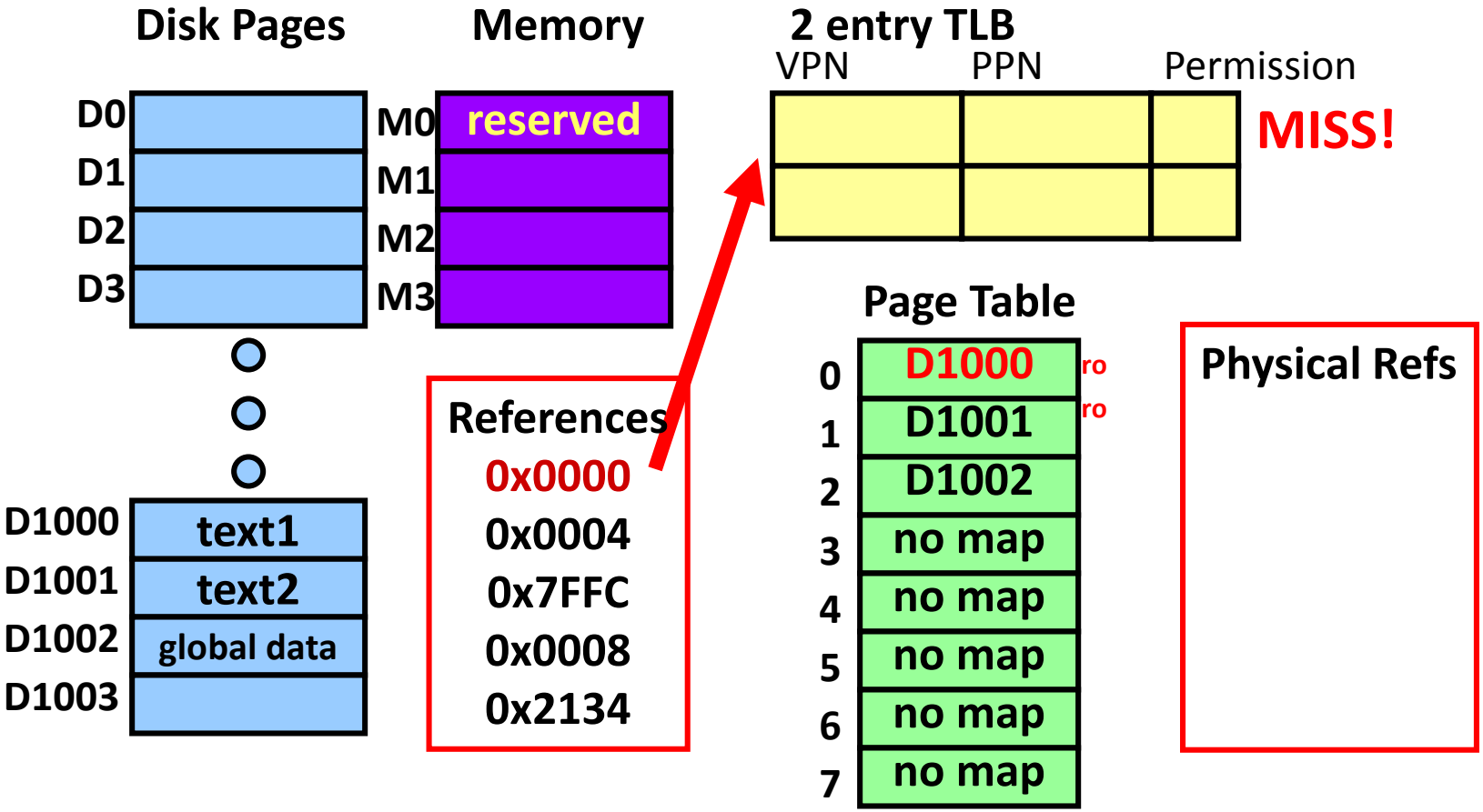
- Page size = 4 KB, Page table entry size = 4 B
- Page table register points to physical address 0x0000



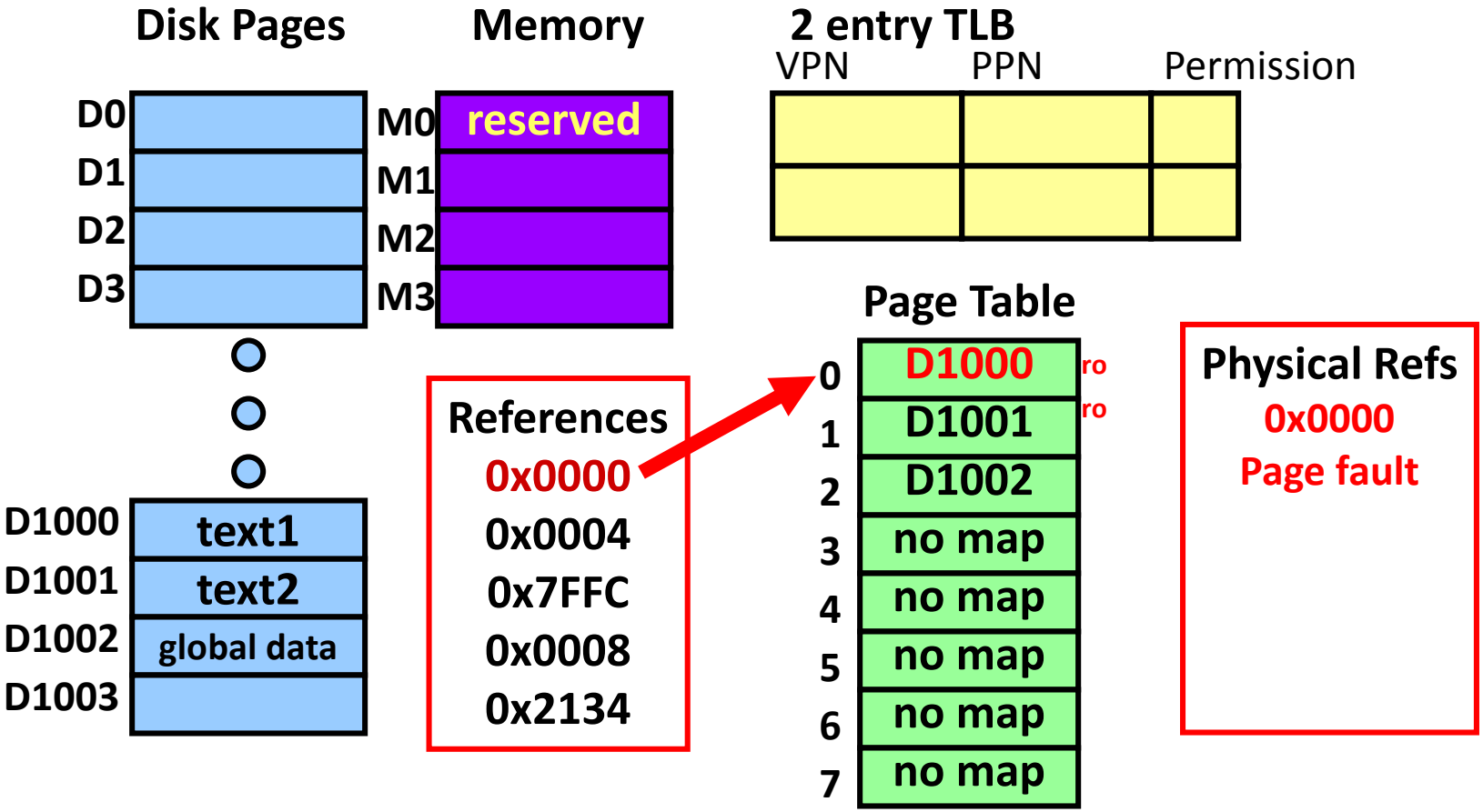
Step 1: Read executable header & initialize page table



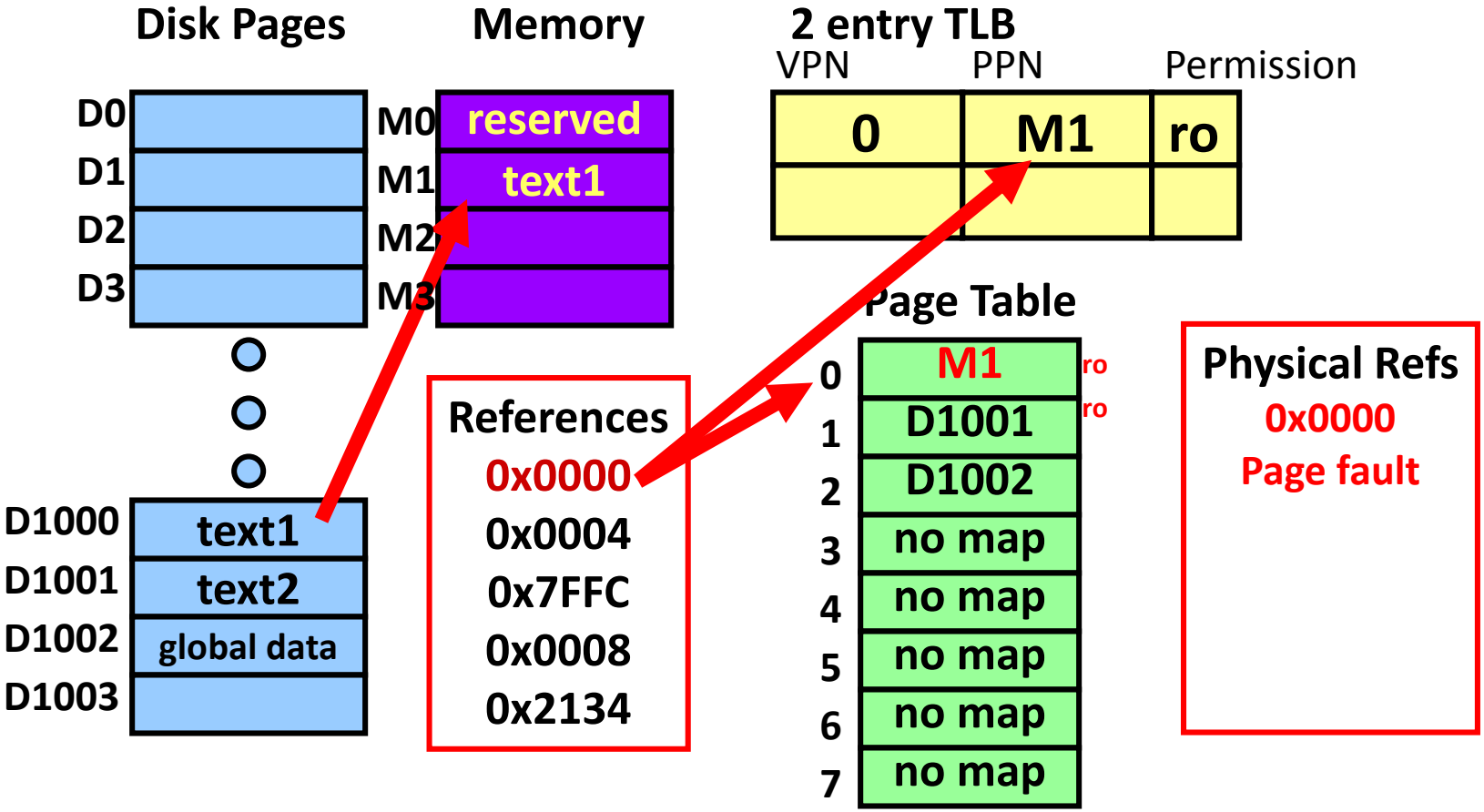
Step 2: Load PC from header & start execution



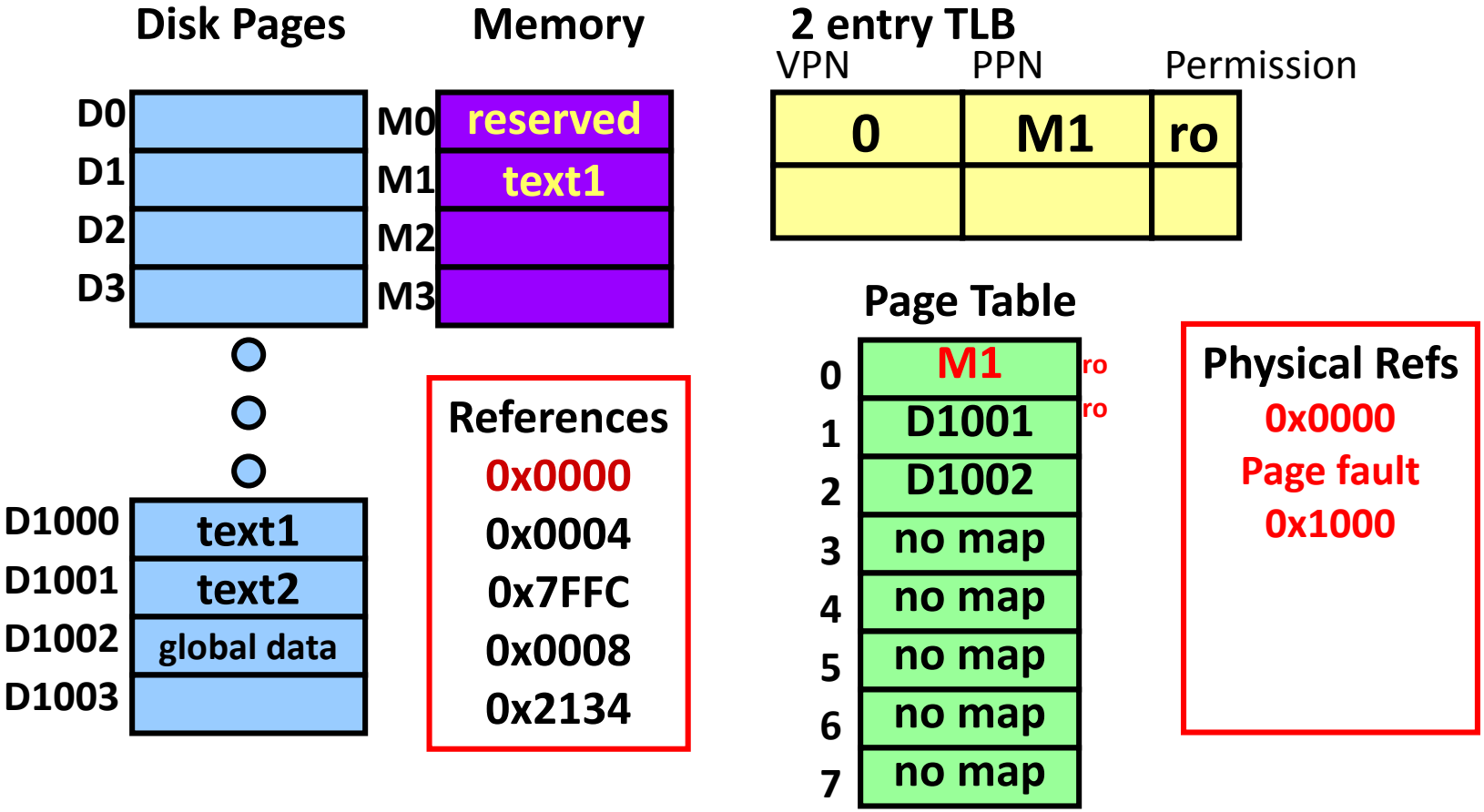
Fetching instruction 0000



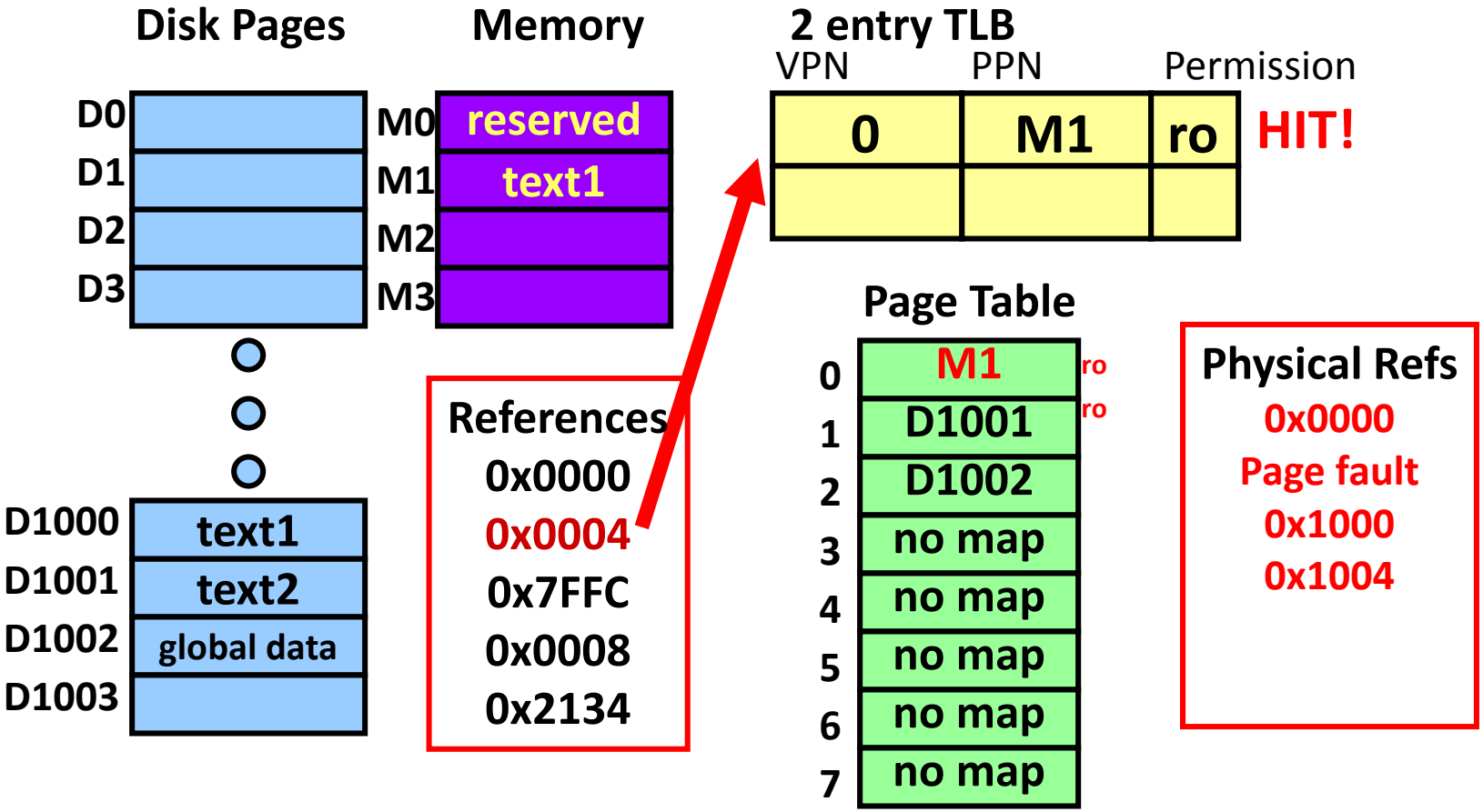
Fetching instruction 0000



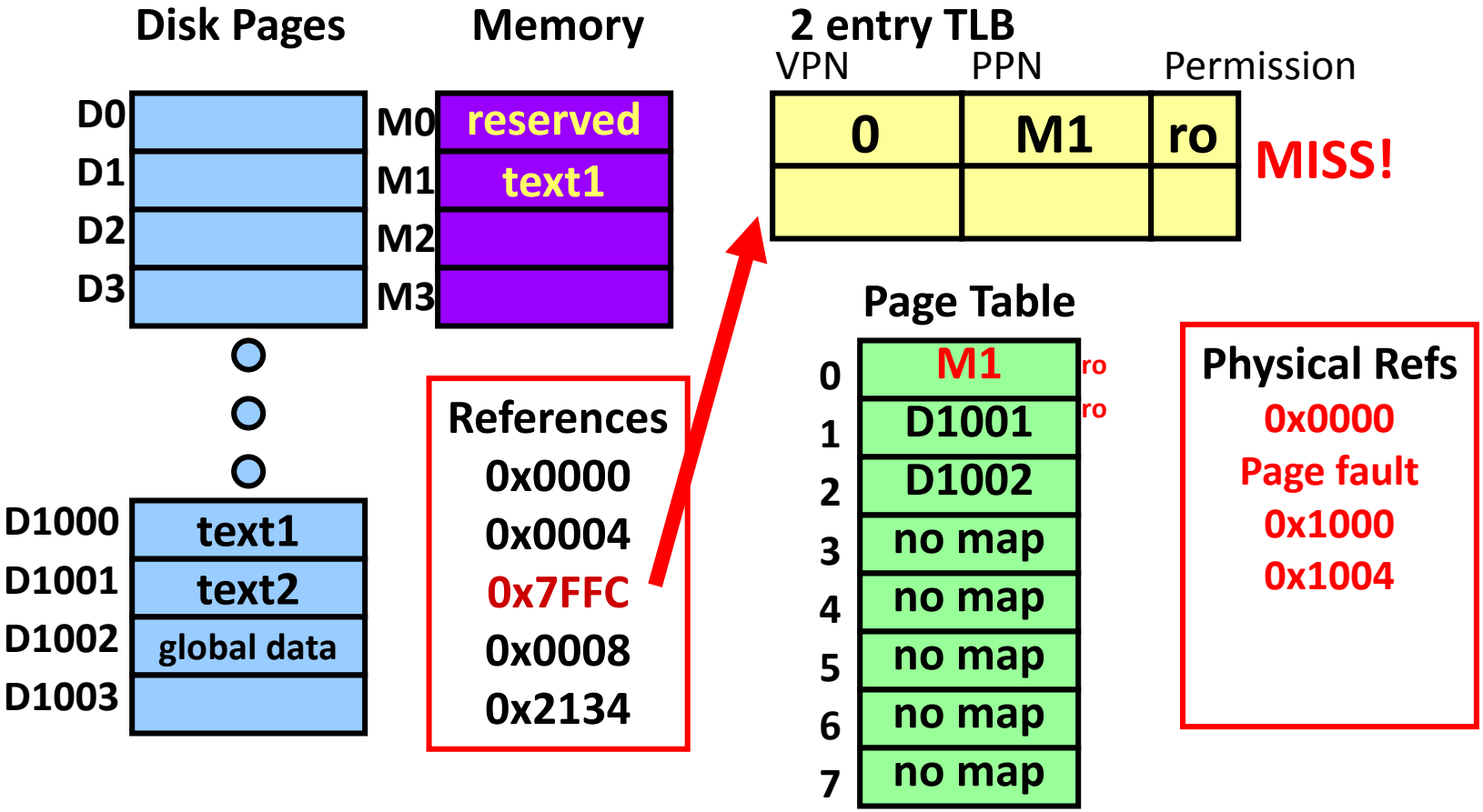
Fetching instruction 0000



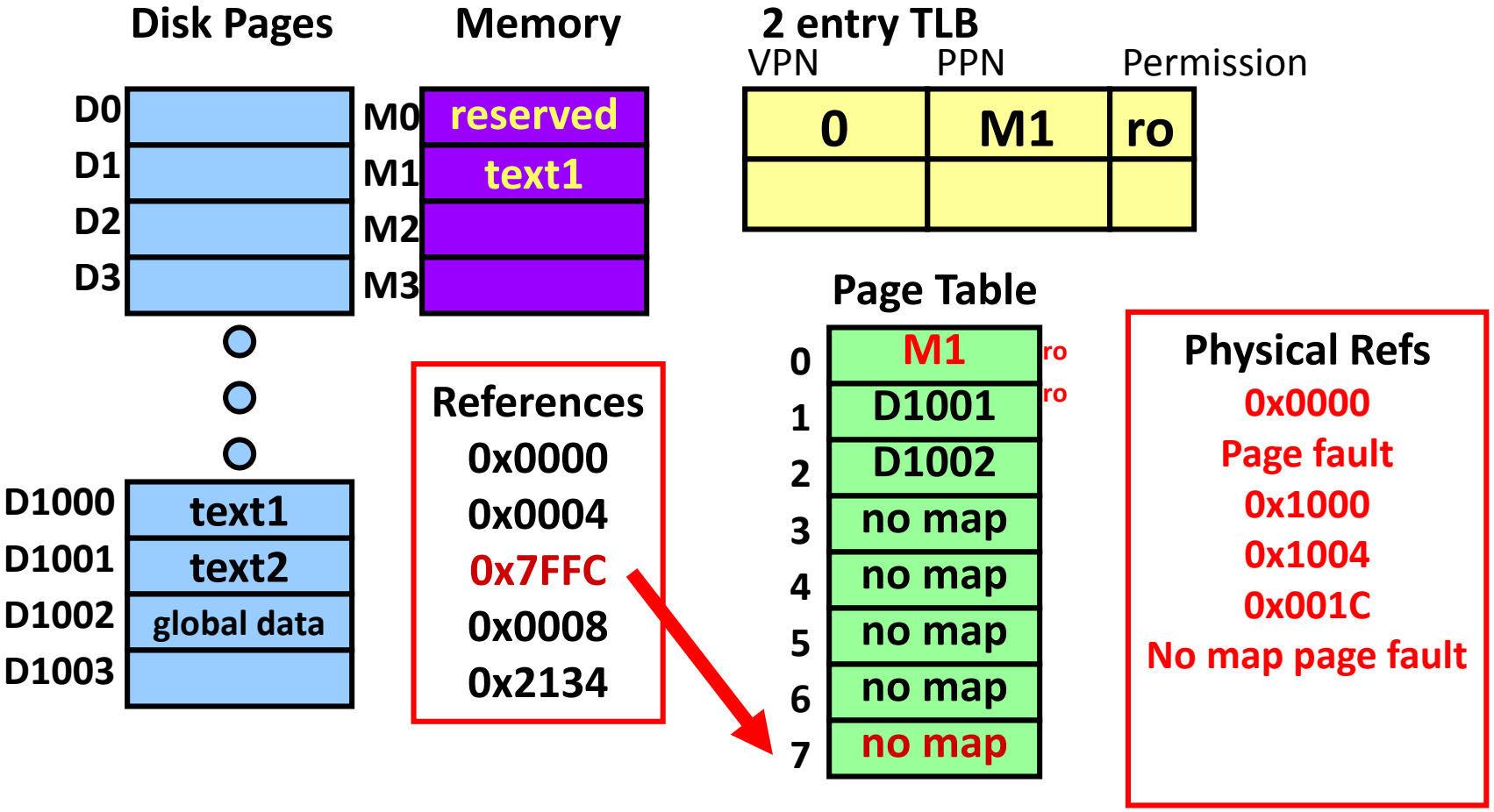
Fetching instruction 0004



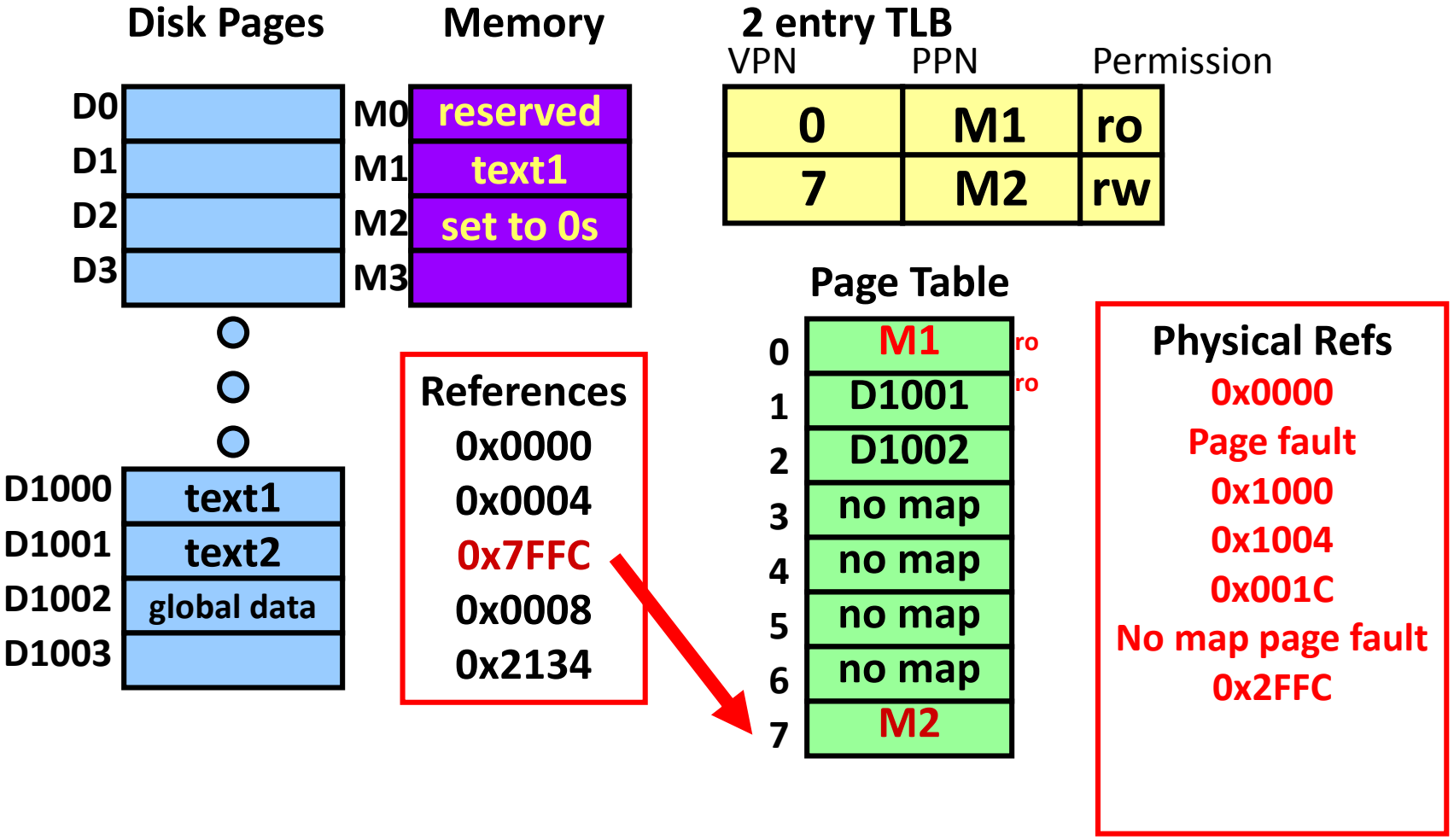
Reference 7FFC



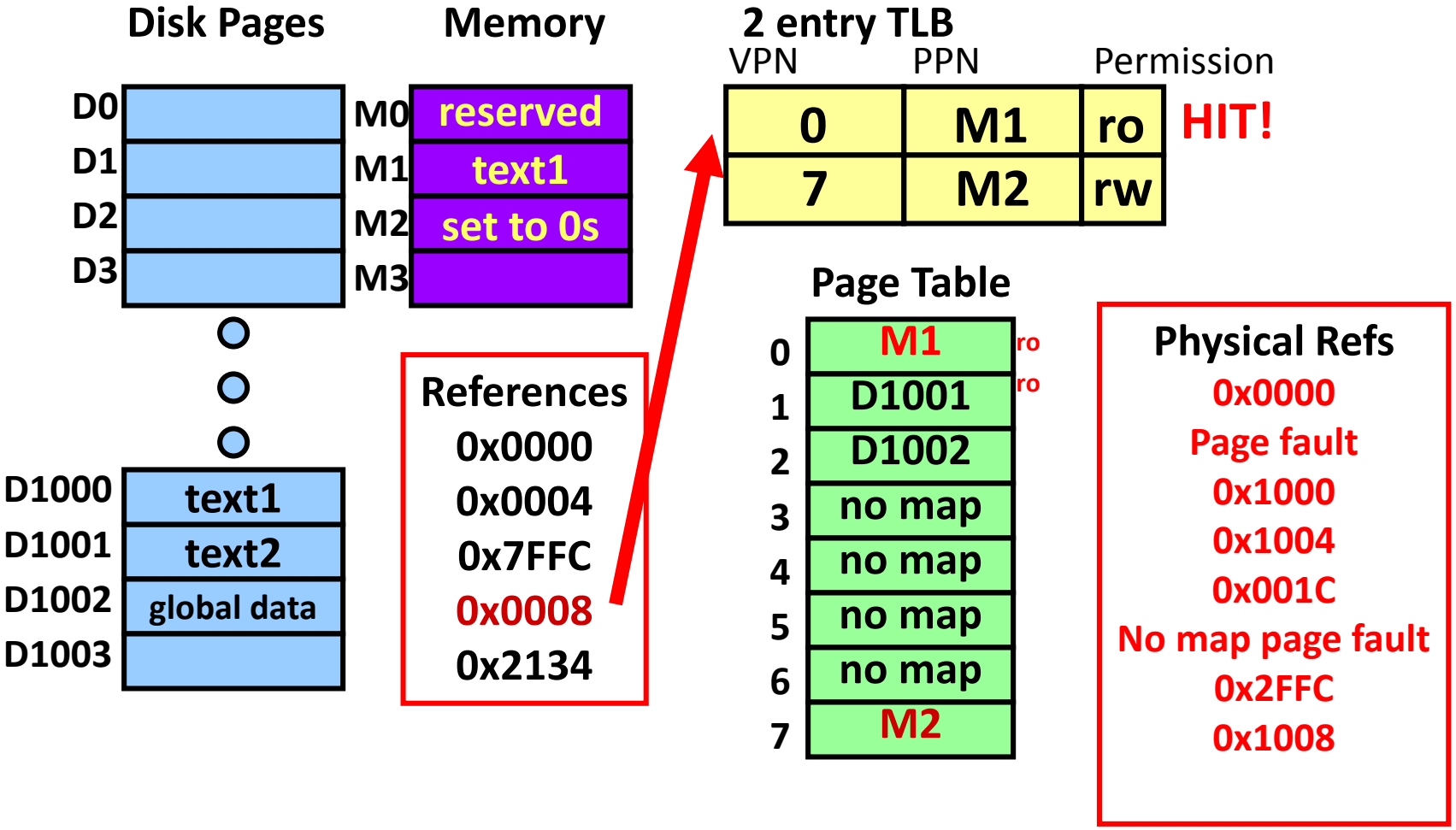
Reference 7FFC



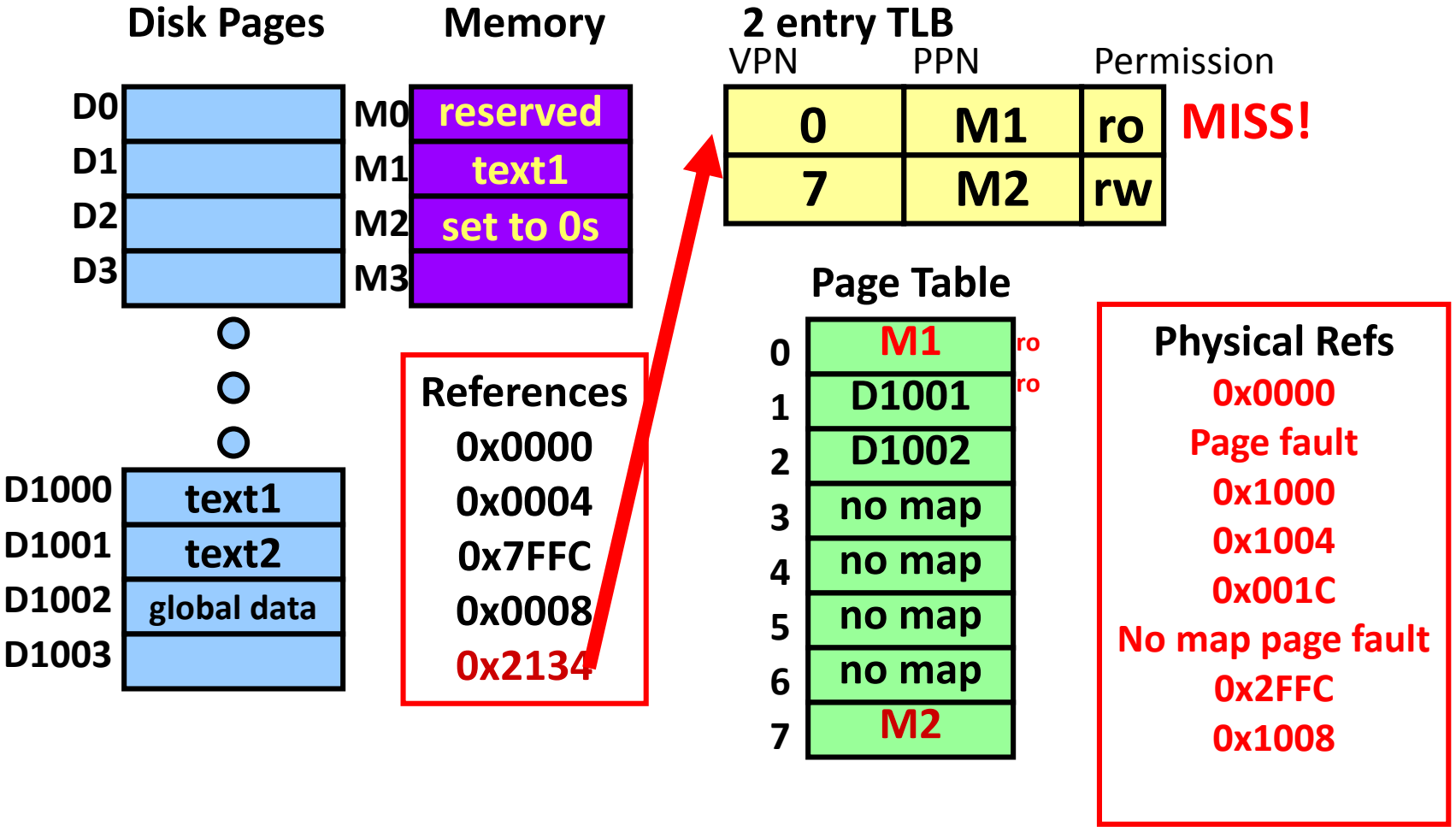
Reference 7FFC



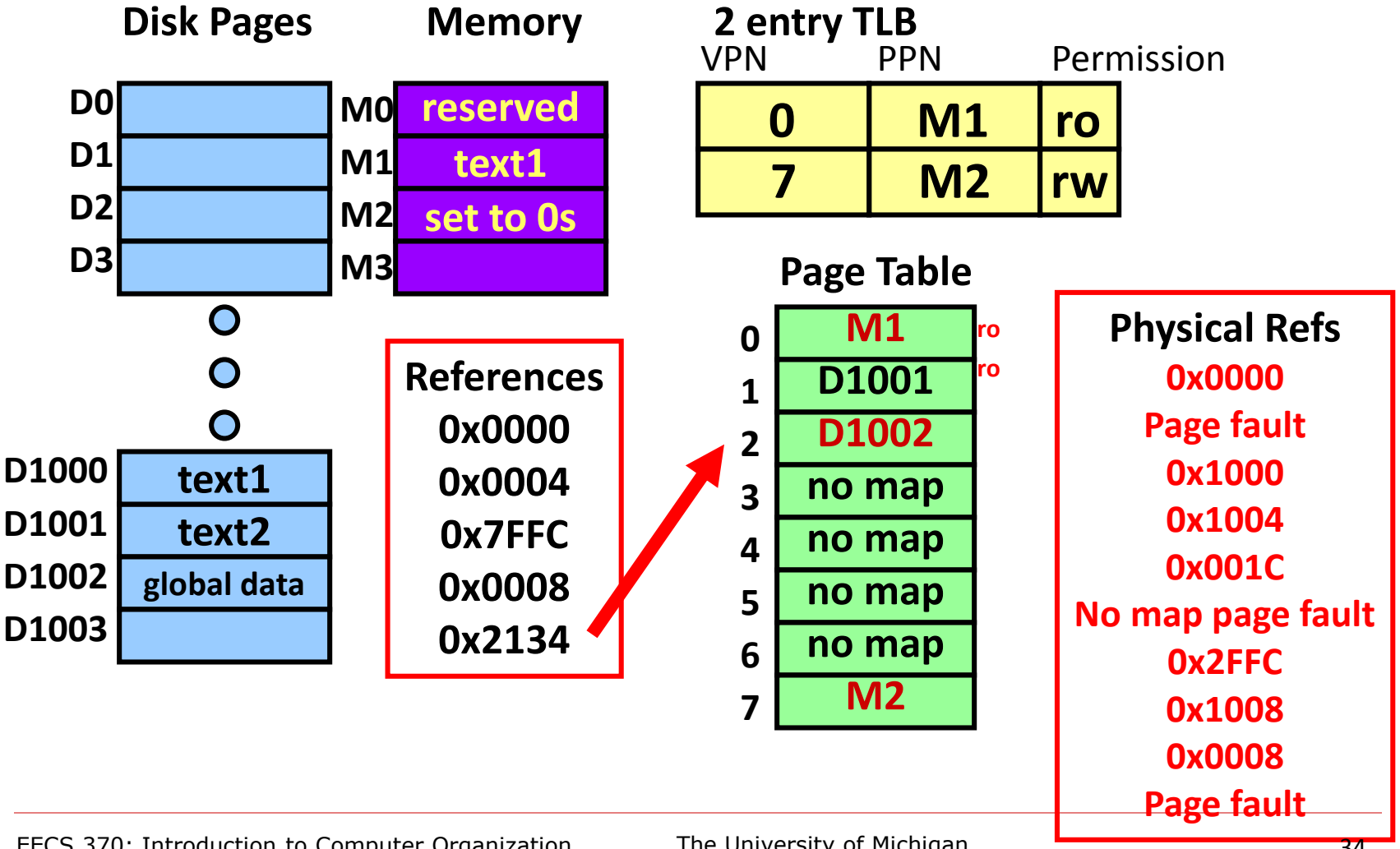
Fetching instruction 0008



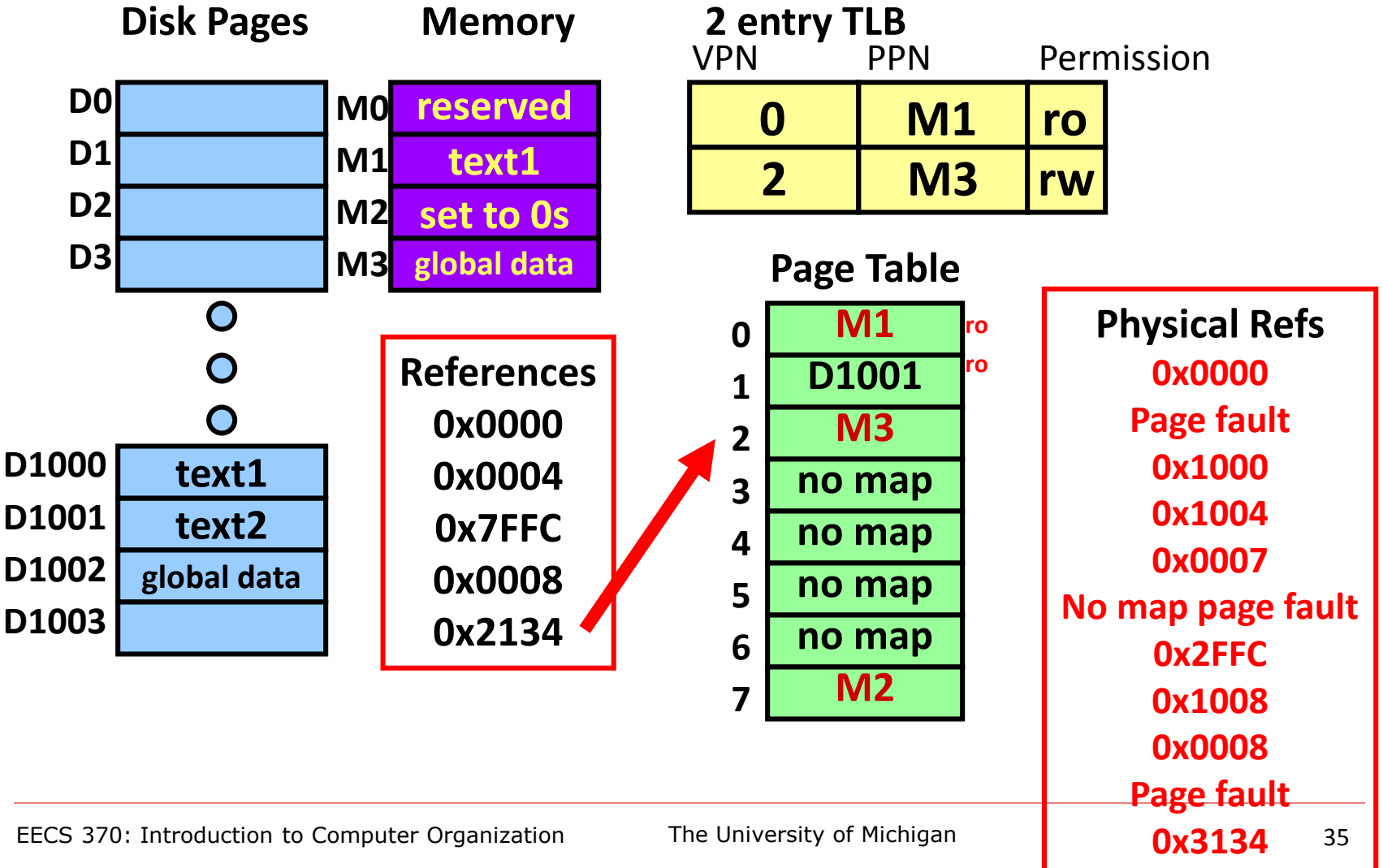
Reference 2134



Reference 2134



Reference 2134



Next topic: Placing Caches in a VM System

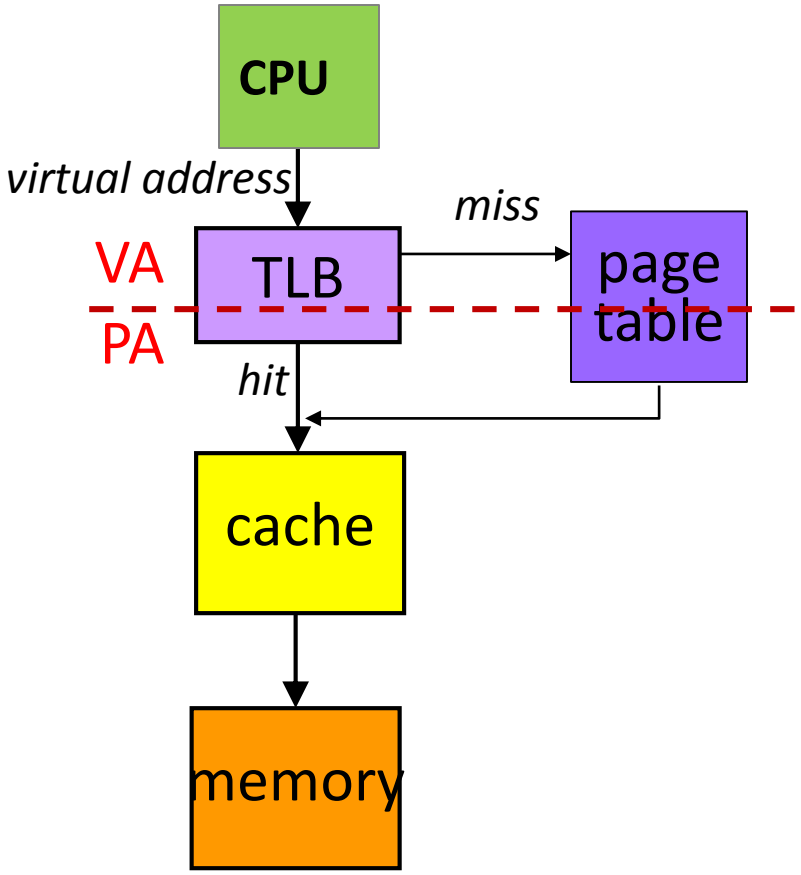
- VM systems give us two different addresses: virtual and physical

- Which address should we use to access the data cache?
  - Physical address (after VM translations).
    - We have to wait for the translation; slower.
  - Virtual address (before VM translation).
    - Faster access.
    - More complex.

Cache & VM Organization: Option 1

Physically-addressed Cache

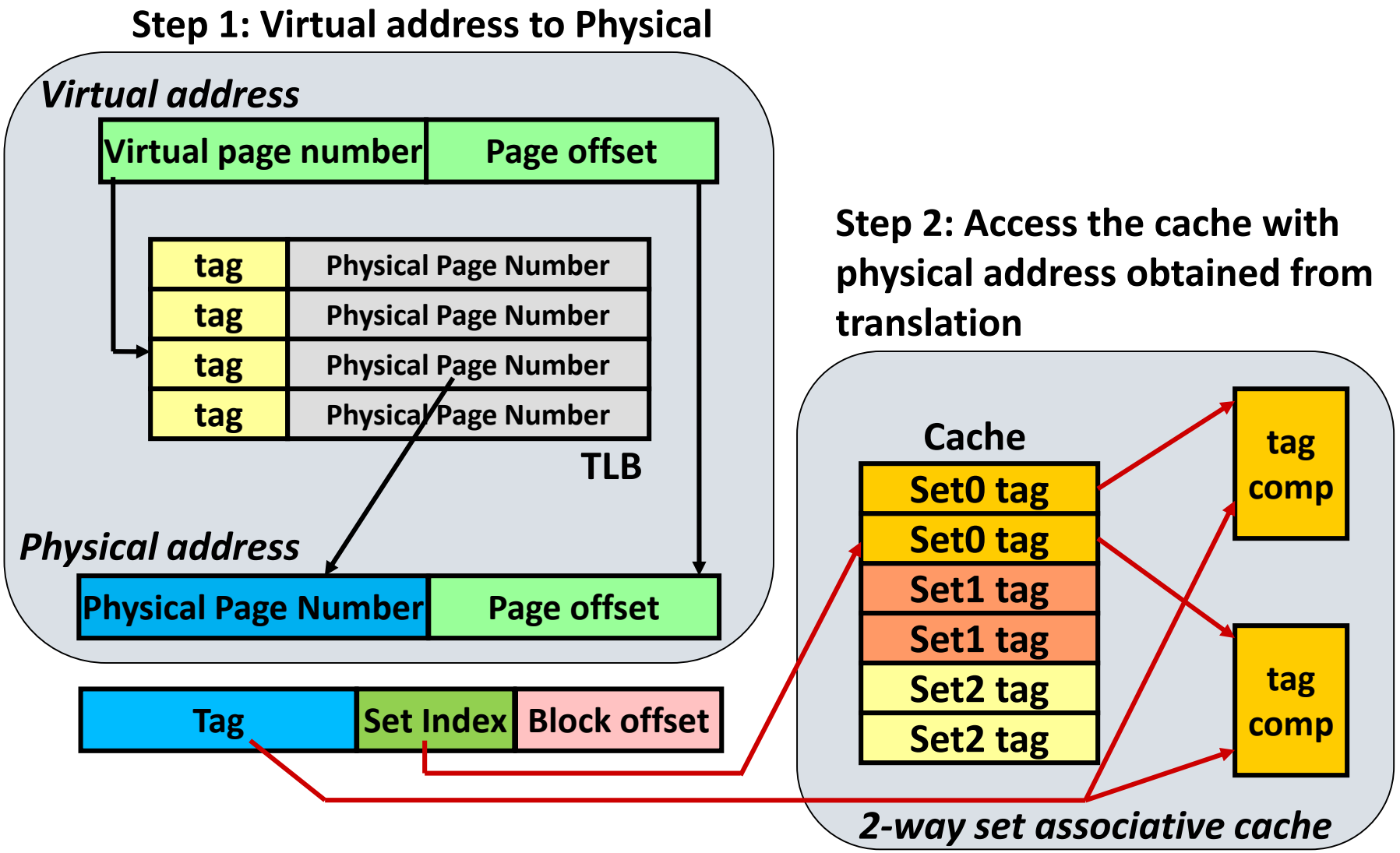
- ✗ Slower
- ✓ Low complexity



Physically addressed caches

- Perform TLB lookup ***before*** cache tag comparison
  - Use bits from ***physical*** address to access cache (tag, set index, and block offset bits)
- Slower access?
  - Tag lookup takes place ***after*** the TLB lookup
- Simplifies some VM management
  - When switching processes, TLB must be invalidated, but cache OK to stay as is
  - Implications? Might result in fewer cache misses if context switches very common (but they generally are not)

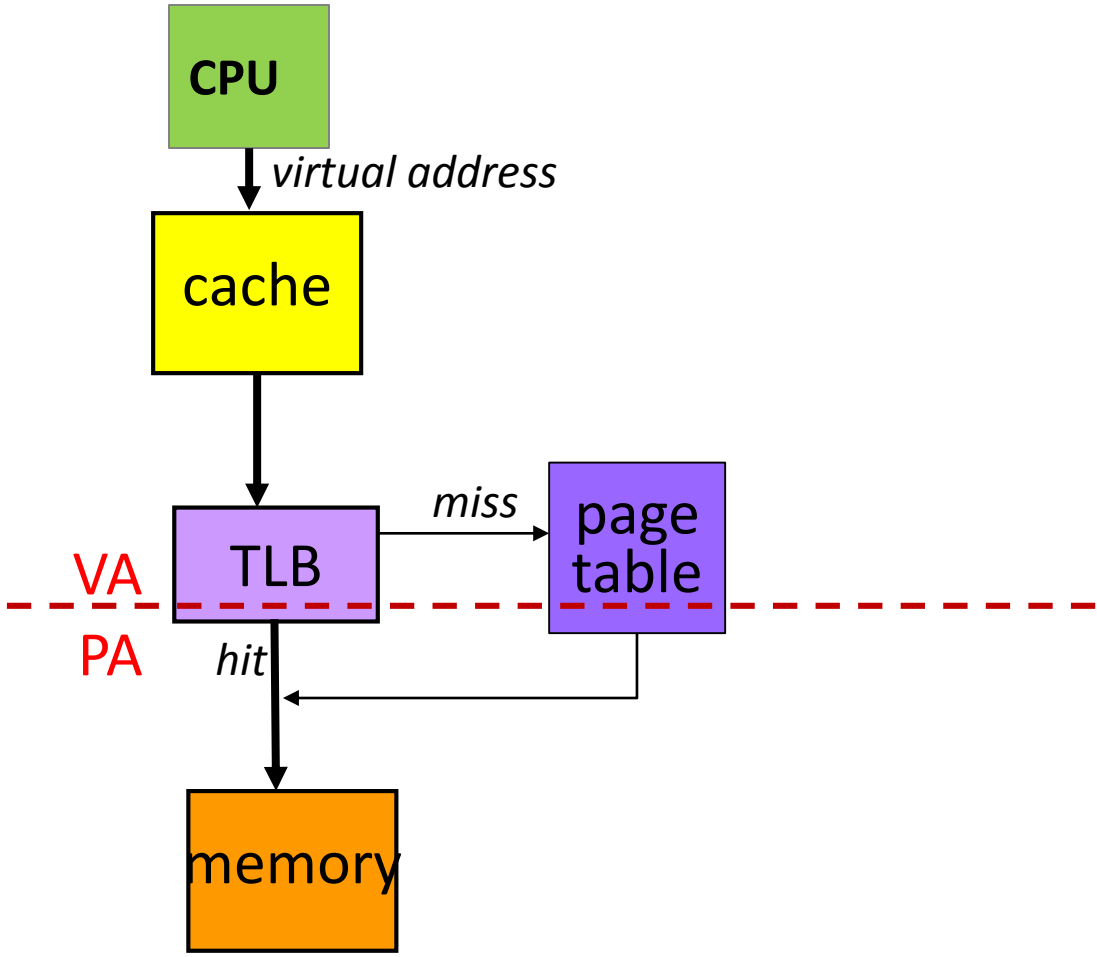
Physically addressed caches



Cache & VM Organization: option 2

Virtually-addressed Cache

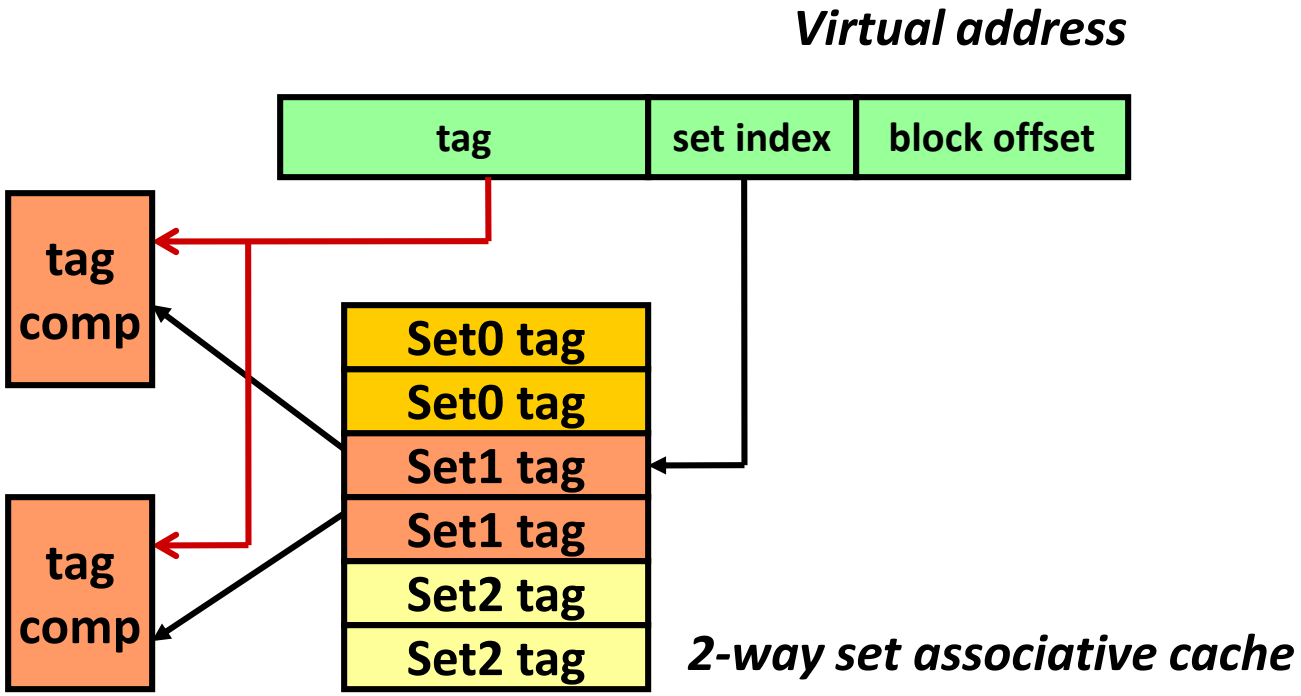
- ✗ High complexity (aliasing)
- ✓ Faster



Virtually addressed caches

- Cache uses bits from the virtual address to access cache (tag, set index, and block offset)
- Perform the TLB only if the cache gets a miss.
- Problems:
  - Aliasing: Two processes may refer to the same physical location with different virtual addresses (synonyms)
  - Two processes may have same virtual addresses with different physical addresses (homonyms)
  - When switching processes, TLB must be invalidated, dirty cache blocks must be written back to memory, and cache must be invalidated to solve homonym problem

Virtually addressed caches

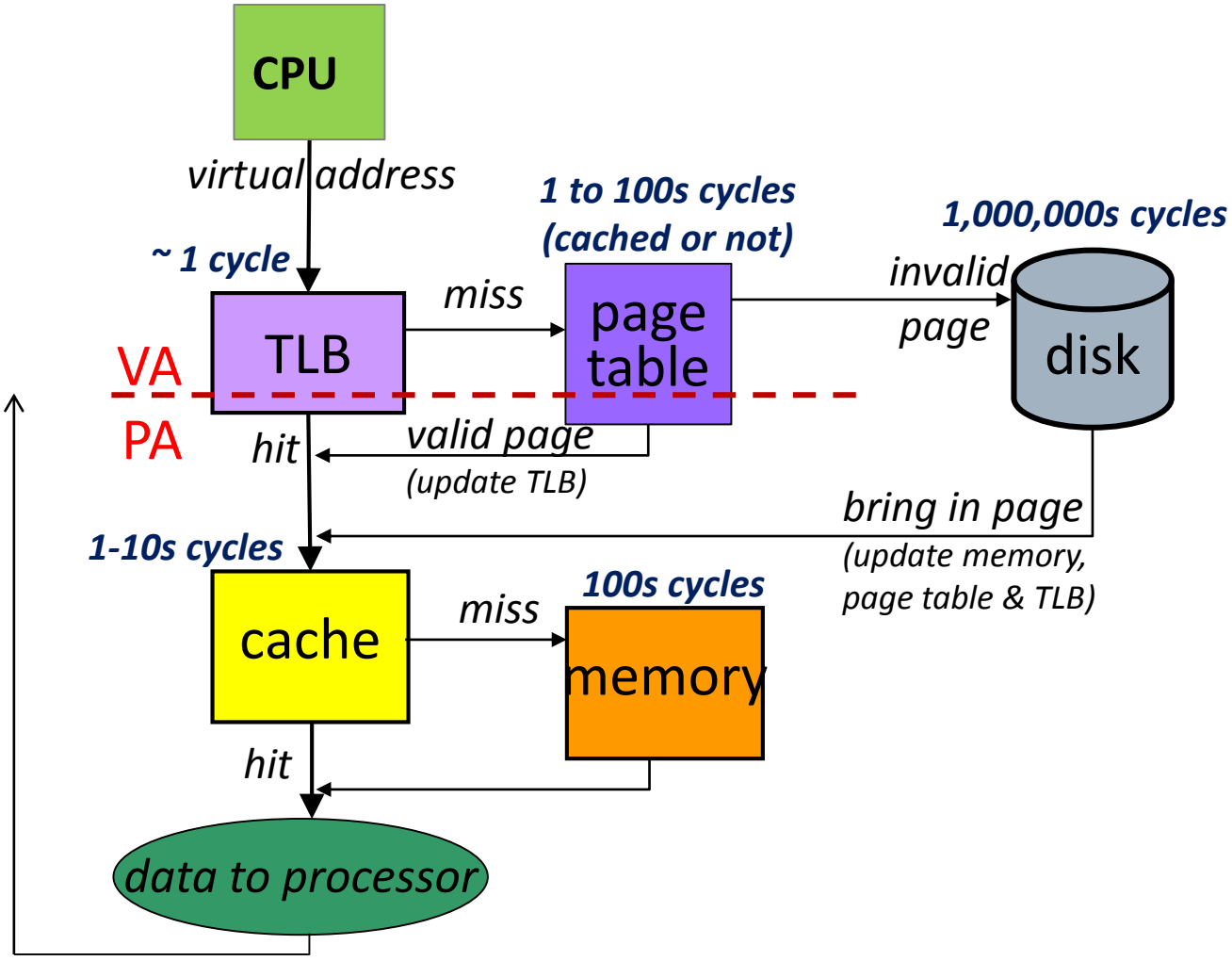


- TLB is accessed in parallel with cache lookup.
- Physical address is used to access main memory in case of a cache miss.

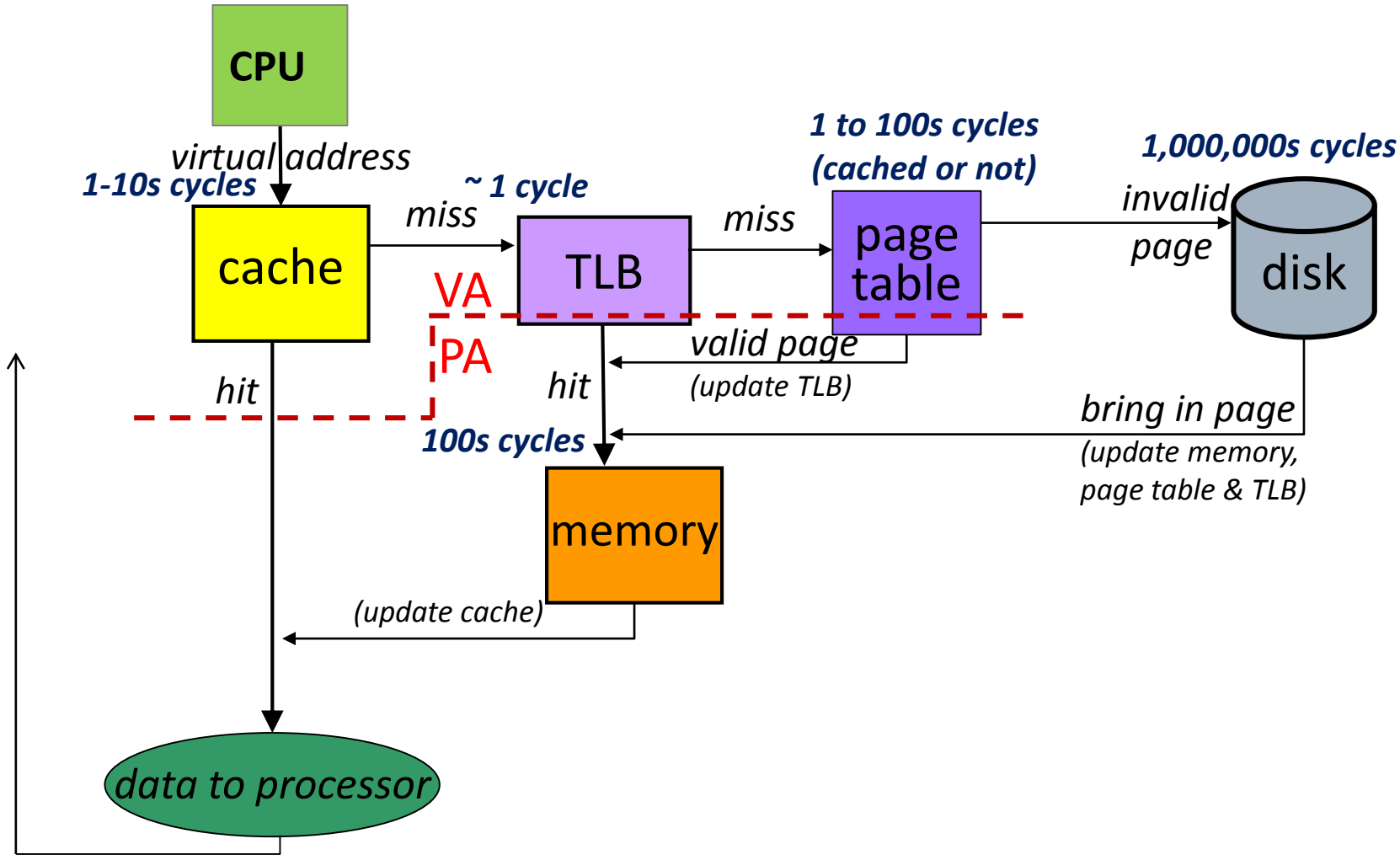
OS Support for Virtual Memory

- It must be able to modify the page table register, update page table values, etc.
- To enable the OS to do this, **BUT** not the user program, we have different execution modes for a process.
  - Executive (or supervisor or kernel level) permissions and
  - User level permissions.

Physically addressed caches: detailed flow



Virtually addressed caches: detailed flow



References (not part of Course Syllabus)

- See how Intel's memory management hardware works, Intel x86 Software Manual: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html
  - Chapter 4 is on Paging
- Linux page table management: https://www.kernel.org/doc/gorman/html/understand/understand006.html