23. Virtual Memory: Design

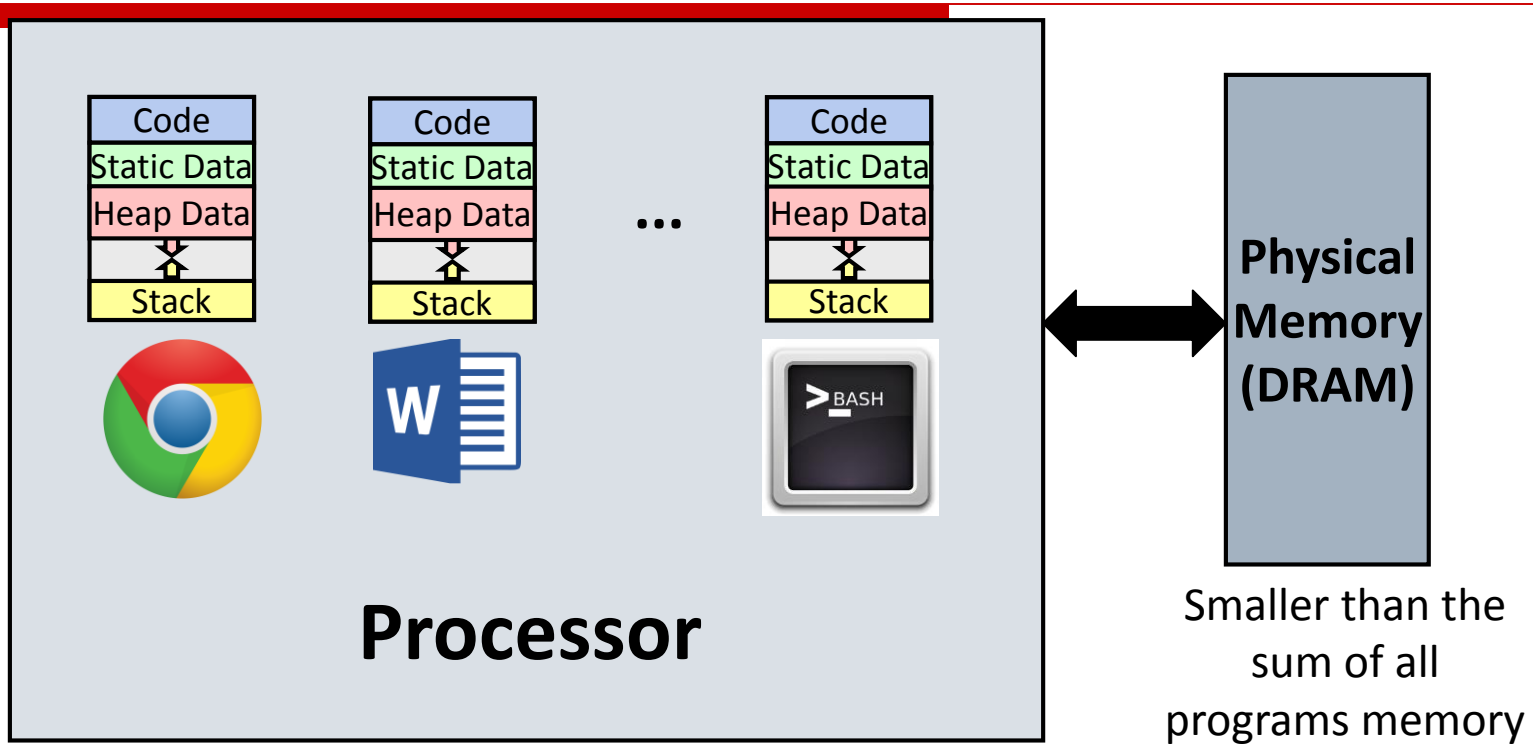
EECS 370 – Introduction to Computer Organization – Winter 2023

EECS Department
University of Michigan in Ann Arbor, USA

ANNOUNCEMENTS

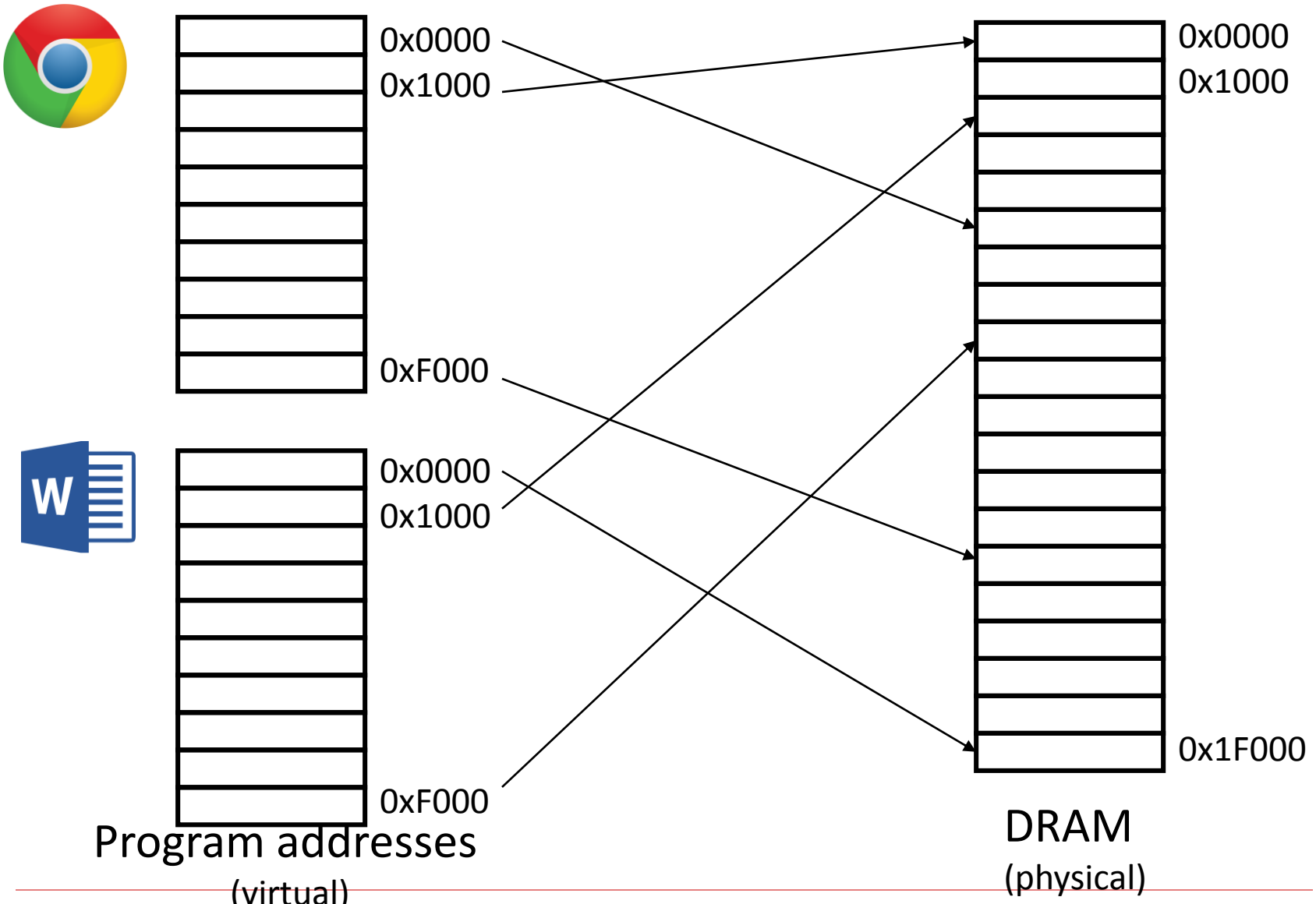
- Project 4 due Thursday, April 13th
- Homework 6 due Monday, April 17th
- Just **2** more lectures of material and one review session!
  - Notice, we have no class on April 13th

Review: Why Virtualize Memory?

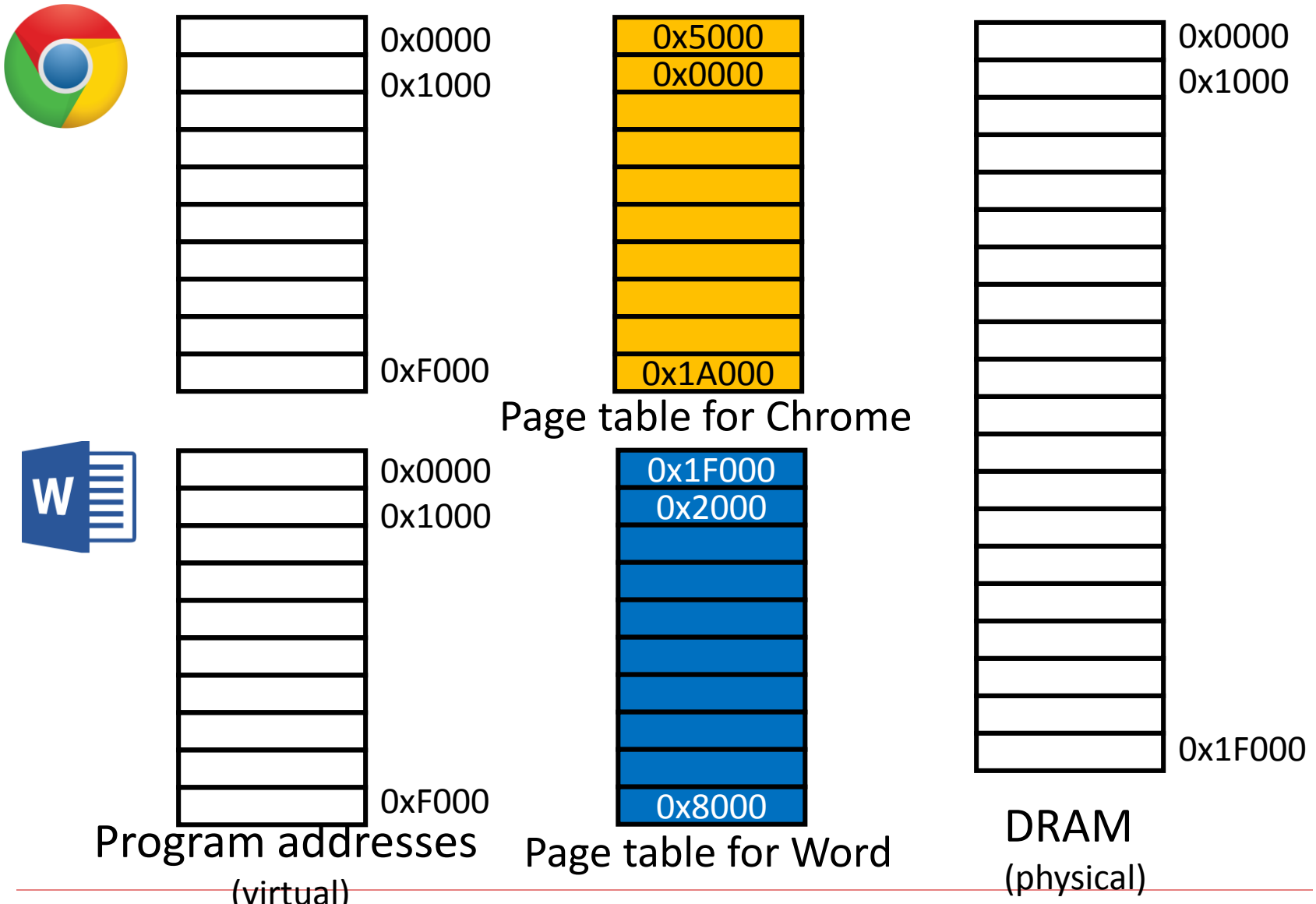


1. **Transparency**
2. **Protection**
3. **Programs not limited by DRAM Capacity**

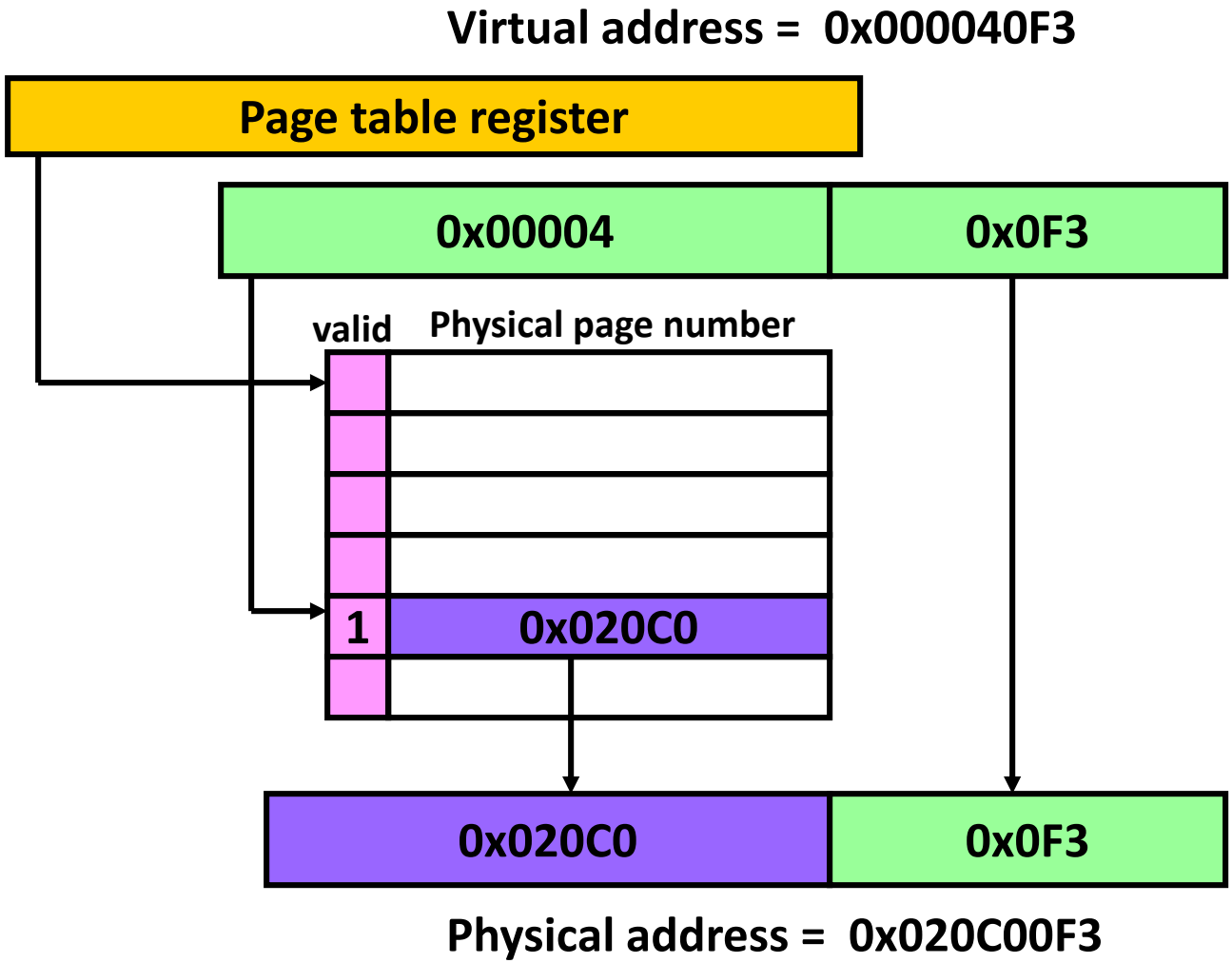
How to achieve transparency & protection?



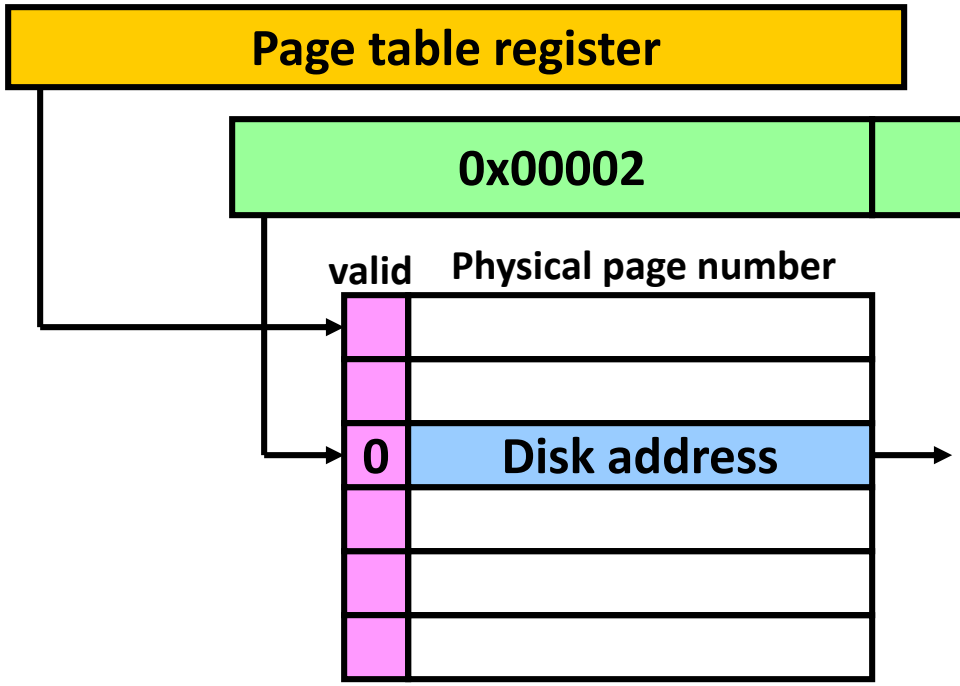
How to achieve transparency & protection?



Review: Page table lookup



Review: Page faults



Exception: page fault

1. Stop this process
2. Pick page to replace
3. Write back data
4. Get referenced page
5. Update page table
6. Reschedule process

Class Problem

- Given the following:
  - 4KB page size, physical memory of 16KB, page table stored in physical page 0 and can never be evicted, 20 bit, byte-addressable virtual address space.
  - The page table initially has virtual page 0 in physical page 1, virtual page 1 in physical page 2 and no valid data in other physical pages.
- Fill in the table on the next slide for each reference
  - Note: like caches we'll use LRU when we need to replace a page.

Class Problem (continued)

4KB page size, physical memory of 16KB, page table stored in physical page 0 and can never be evicted, 20 bit, byte-addressable virtual address space.

The page table initially has virtual page 0 in physical page 1, virtual page 1 in physical page 2 and no valid data in other physical pages.

Virt addr	Virt page	Page fault?	Phys addr
0x00F0C			
0x01F0C			
0x20F0C			
0x00100			
0x00200			
0x30000			
0x01FFF			
0x00200			

Class Problem (continued)

4KB page size, physical memory of 16KB, page table stored in physical page 0 and can never be evicted, 20 bit, byte-addressable virtual address space.

The page table initially has virtual page 0 in physical page 1, virtual page 1 in physical page 2 and no valid data in other physical pages.

Virt addr	Virt page	Page fault?	Phys addr
0x00F0C	0x0	N	0x1F0C
0x01F0C			
0x20F0C			
0x00100			
0x00200			
0x30000			
0x01FFF			
0x00200			

Class Problem (continued)

4KB page size, physical memory of 16KB, page table stored in physical page 0 and can never be evicted, 20 bit, byte-addressable virtual address space.

The page table initially has virtual page 0 in physical page 1, virtual page 1 in physical page 2 and no valid data in other physical pages.

Virt addr	Virt page	Page fault?	Phys addr
0x00F0C	0x0	N	0x1F0C
0x01F0C	0x1	N	0x2F0C
0x20F0C			
0x00100			
0x00200			
0x30000			
0x01FFF			
0x00200			

Class Problem (continued)

Virt addr	Virt page	Page fault?	Phys addr
0x00F0C	0x0	N	0x1F0C
0x01F0C	0x1	N	0x2F0C
0x20F0C	0x20	Y (into 3)	0x3F0C
0x00100			
0x00200			
0x30000			
0x01FFF			
0x00200			

Class Problem (continued)

4KB page size, physical memory of 16KB, page table stored in physical page 0 and can never be evicted, 20 bit, byte-addressable virtual address space.

The page table initially has virtual page 0 in physical page 1, virtual page 1 in physical page 2 and no valid data in other physical pages.

Virt addr	Virt page	Page fault?	Phys addr
0x00F0C	0x0	N	0x1F0C
0x01F0C	0x1	N	0x2F0C
0x20F0C	0x20	Y (into 3)	0x3F0C
0x00100	0x0	N	0x1100
0x00200	0x0	N	0x1200
0x30000	0x30	Y (into 2)	0x2000
0x01FFF	0x1	Y (into 3)	0x3FFF
0x00200	0x0	N	0x1200

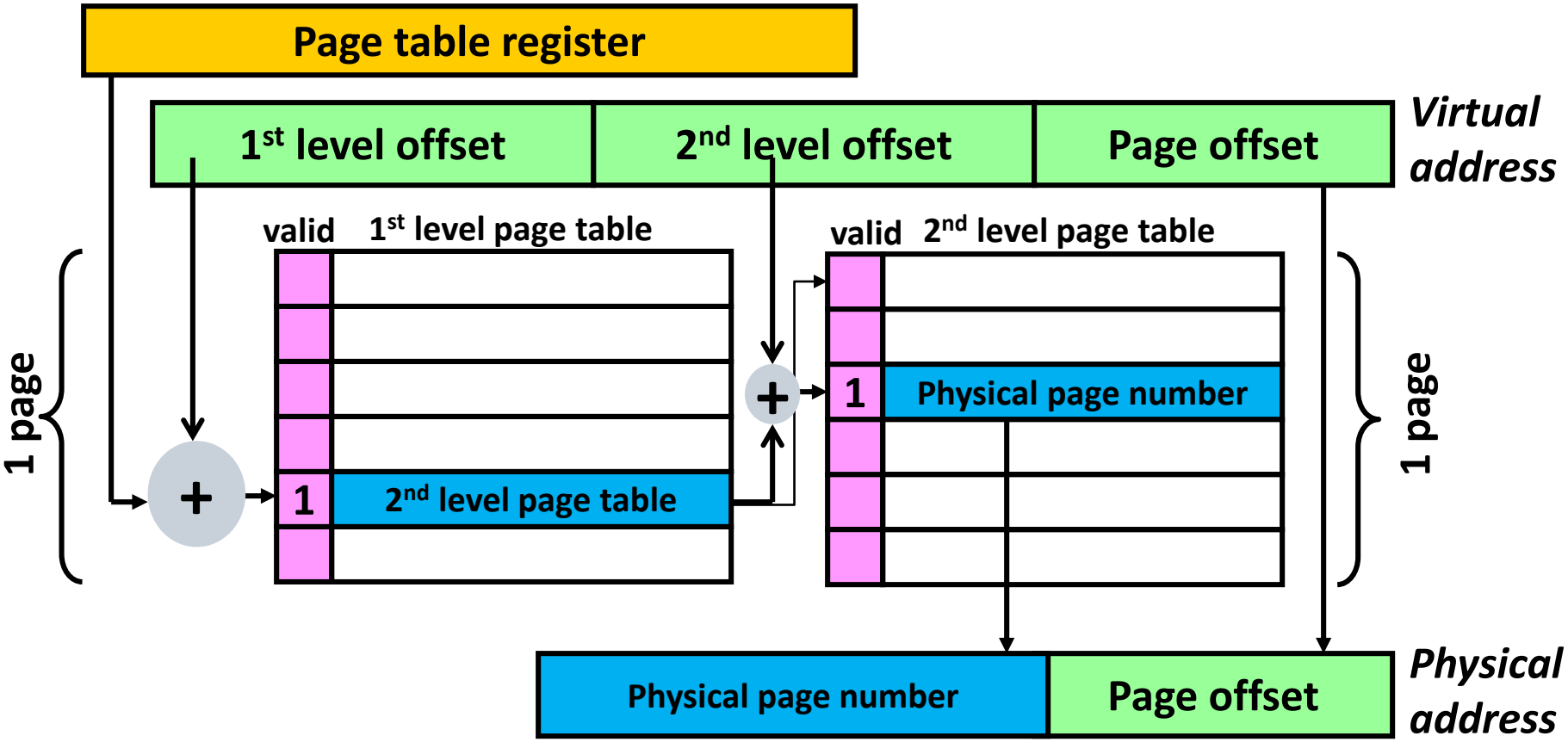
Size of the page table

- How big is a page table entry?
  - For 32-bit virtual address:
    - If the machine can support 1GB = 2^{30} bytes of physical memory and we use pages of size 4KB = 2^{12},
    - then the physical page number is 30-12 = 18 bits.
      Plus another valid bit + other useful stuff (read only, dirty, etc.)
    - Let say about 3 bytes.
- How many entries in the page table?
  - ARM virtual address is 32 bits – 12 bit page offset = 20
  - Total number of virtual pages = 2^{20}
- Total size of page table = Number of virtual pages
  * Size of each page table entry
  = 2^{20} x 3 bytes ~ 3 MB

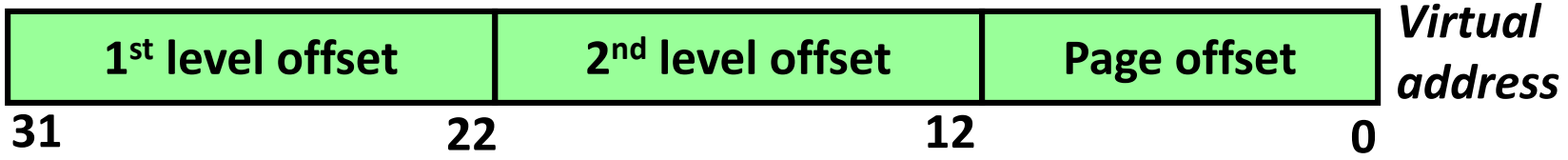
How can you organize the page table?

1. Single-level page table occupies continuous region in physical memory
   - Previous example always takes 3MB regardless of how much virtual memory is used
2. Use a multi-level page table! (Build a hierarchical page table and keep translations only if used in the program)
   - 1st level page table in physical memory
   - 2nd (or more) level page tables created only if needed
   - **Size is proportional to the amount of memory used**
   - Optimization for *common case: almost no programs use all the virtual pages at the same time*
   - ***Common case: size of multi-level page table << single-level***
   - ***Worst case: size of multi-level page table > single-level***

Hierarchical page table



Hierarchical page table – 32bit Intel x86



- How many bits in the virtual 1st level offset field? 10
- How many bits in the virtual 2nd level offset field? 10
- How many bits in the page offset? 12
- How many pages in the 1st level page table? 2^{10}=1024
- How many bytes for each entry in the 1st level page table? 4
- How many entries in the 2nd level of the page table? 2^{10}=1024
- How many bytes for each VPN in a 2nd level table? 4
- What is the total size of the page table? 4K+n*4K

  (here *n* is number of valid entries in the 1st level page table)

Class Problem (32 bit x86)

- What is the least amount of memory that could be used? When would this happen?
- What is the most memory that could be used? When would this happen?
- How much memory is used for this memory access pattern:
  0x00000ABC
  0x00000ABD
  0x10000ABC
  0x20000ABC
- How much memory if we used a single-level page table with 4KB pages? Assume entries are rounded to the nearest word (4B)

Class Problem (32 bit x86)

- What is the least amount of memory that could be used? When would this happen?
  - 4KB for 1st level page table. Occurs when no memory has been accessed (before program runs)

- What is the most memory that could be used? When would this happen?
  - 4KB for 1st level page table
    + 1024*4KB for all possible 2nd level page tables
    = 4100KB (which slightly greater than 4096KB)
  - Occurs when program uses all virtual pages (= 2^{20} pages)

Class Problem (32 bit x86)

- How much memory is used for this memory access pattern:

```
0x00000ABC  // Page fault
0x00000ABD
0x10000ABC  // Page fault
0x20000ABC  // Page fault
```

- 4KB for 1st level page table + 3*4KB for each 2L page table
  = 16 KB

Class Problem (32 bit x86)

- How much memory if we used a single-level page table with 4KB pages? Assume entries are rounded to the nearest word (4B)

  - 32 bits – 12 bit page offset = 20 bits of address
  - 2^20 (num entries) * 4B
    => 4MB

Class Problem – Multi-level VM

- Design a two-level virtual memory system of a byte addressable processor with ***24-bits long addresses***. No cache in the system. **256Kbytes of memory** installed, and no additional memory can be added.
  - **Virtual memory page: 512 Bytes**. Each page table entry must be an integer number of bytes, and must be the smallest size required to fit the physical page number + 1 bit to mark valid-entry
  - We want each second-level page table to fit exactly in one memory page, and 1st level page table entries are 3 bytes each (a memory address pointer to a 2L page table).
- Compute:
  - Number of entries in each 2nd level page table;
  - Number of virtual address bits used to index the 2nd level page table;
  - Number of virtual address bits used to index the 1st level page table;
  - Size of the 1st level page table.

Class Problem – Multi-level VM

Page Offset: 9 bits (512B page size)

Physical address = 18b (256KB Mem size)

Physical page number = 18b (256KB mem size) – 9b (offset)

Physical page number = 9b	Page offset = 9b

2nd level page table entry size: 9b (physical page number) + 1b ~ 2 bytes

2nd level page table **fits exactly in 1 page**

#entries in 2nd level page table is 512 bytes / 2 bytes = 256

#entries in 2nd level page table = 256 ➜Virtual page bits = 8b

Virtual 1st level page bits = 24 – 8 – 9 = 7b

1st level page table size = 2^7 * 3 bytes = 384B

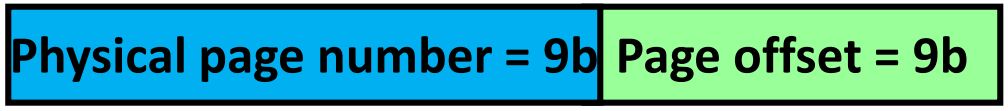
1st level = 7b	2nd level = 8b	Page offset = 9b

Virtual address = 24b

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



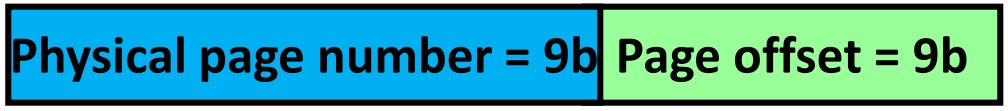
Physical address = 18b

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



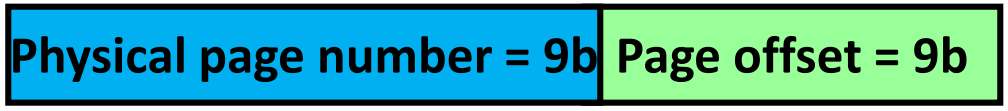
Physical address = 18b

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



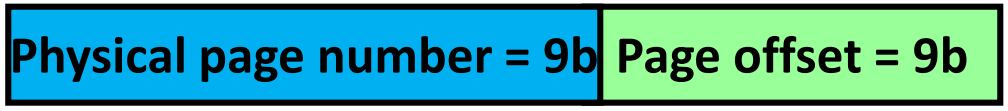
Physical address = 18b

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

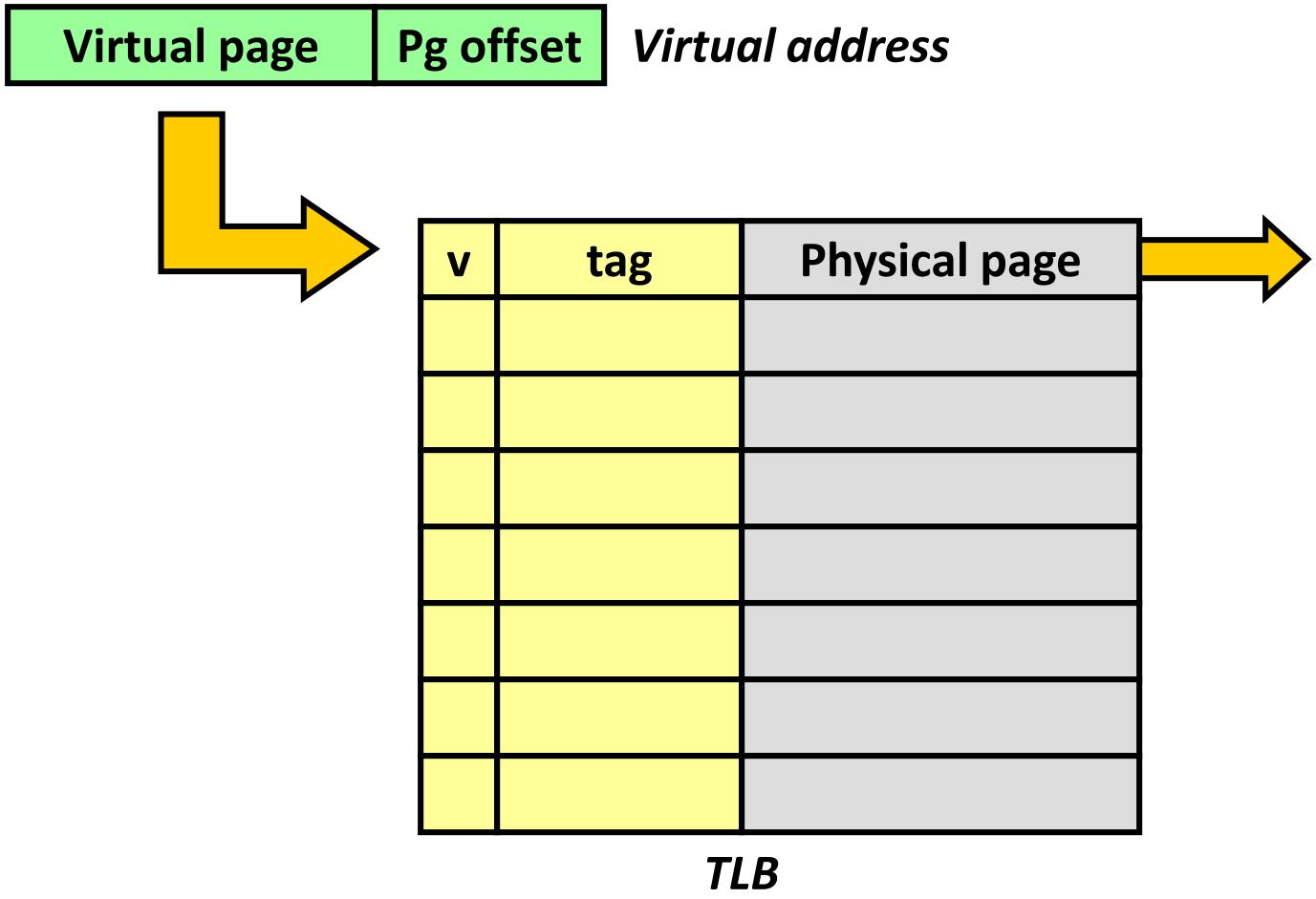
- Then we access physical memory again to get the data
  - A load instruction performs at least 2 memory reads
  - A store instruction performs at least 1 read and then a write

- 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 memory dedicated to storing a small subset of valid V-to-P translations.
- 16-512 entries common.
- Generally has low miss rate (< 1%).

Translation look-aside buffer (TLB)



Where is the TLB lookup?

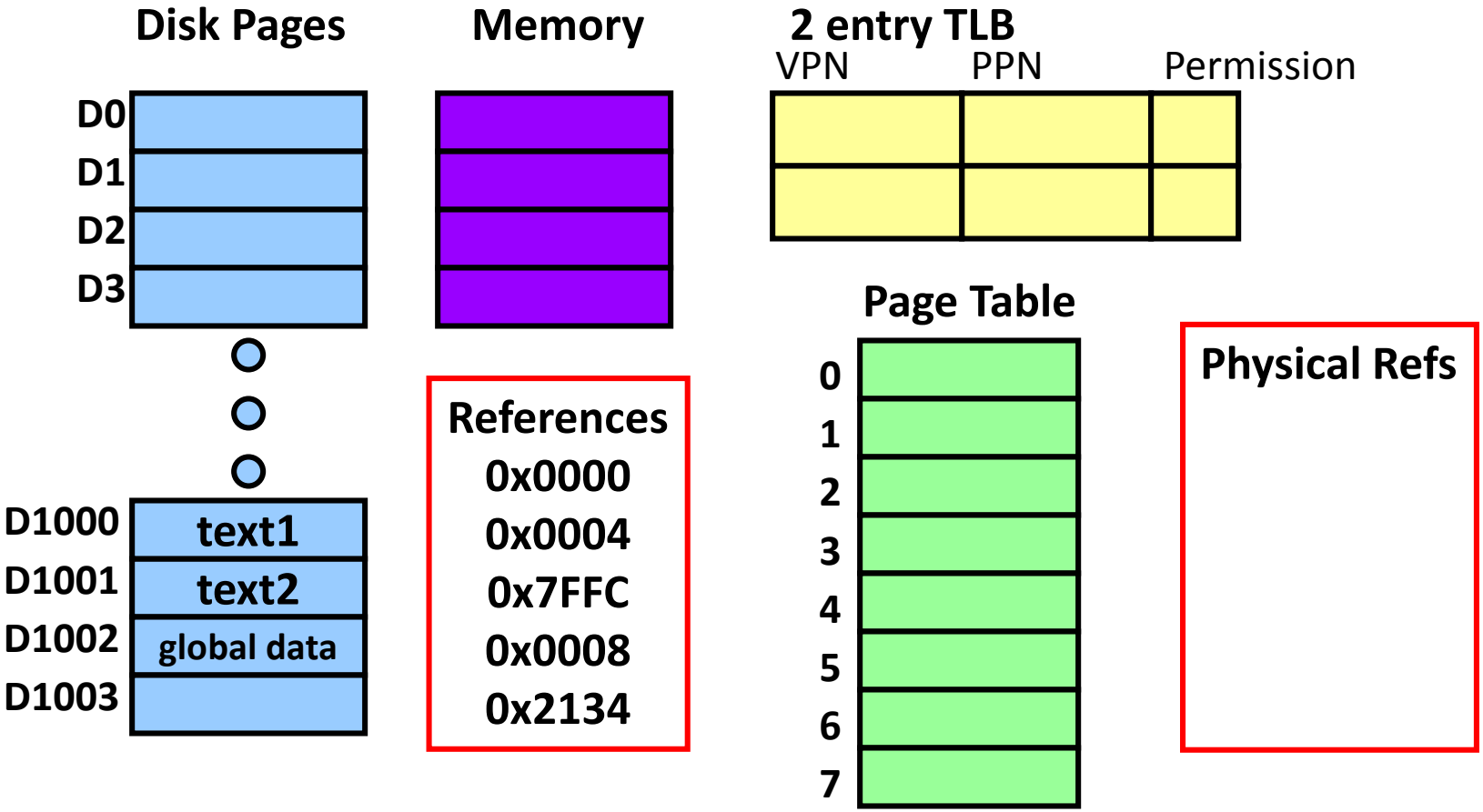
- We put the TLB lookup in the pipeline after the virtual address is calculated and before the memory reference is performed.
  - This may be before or during the data cache access.
  - In case of a TLB miss, we need to perform the virtual to physical address translation during the memory stage of the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

Putting it all together

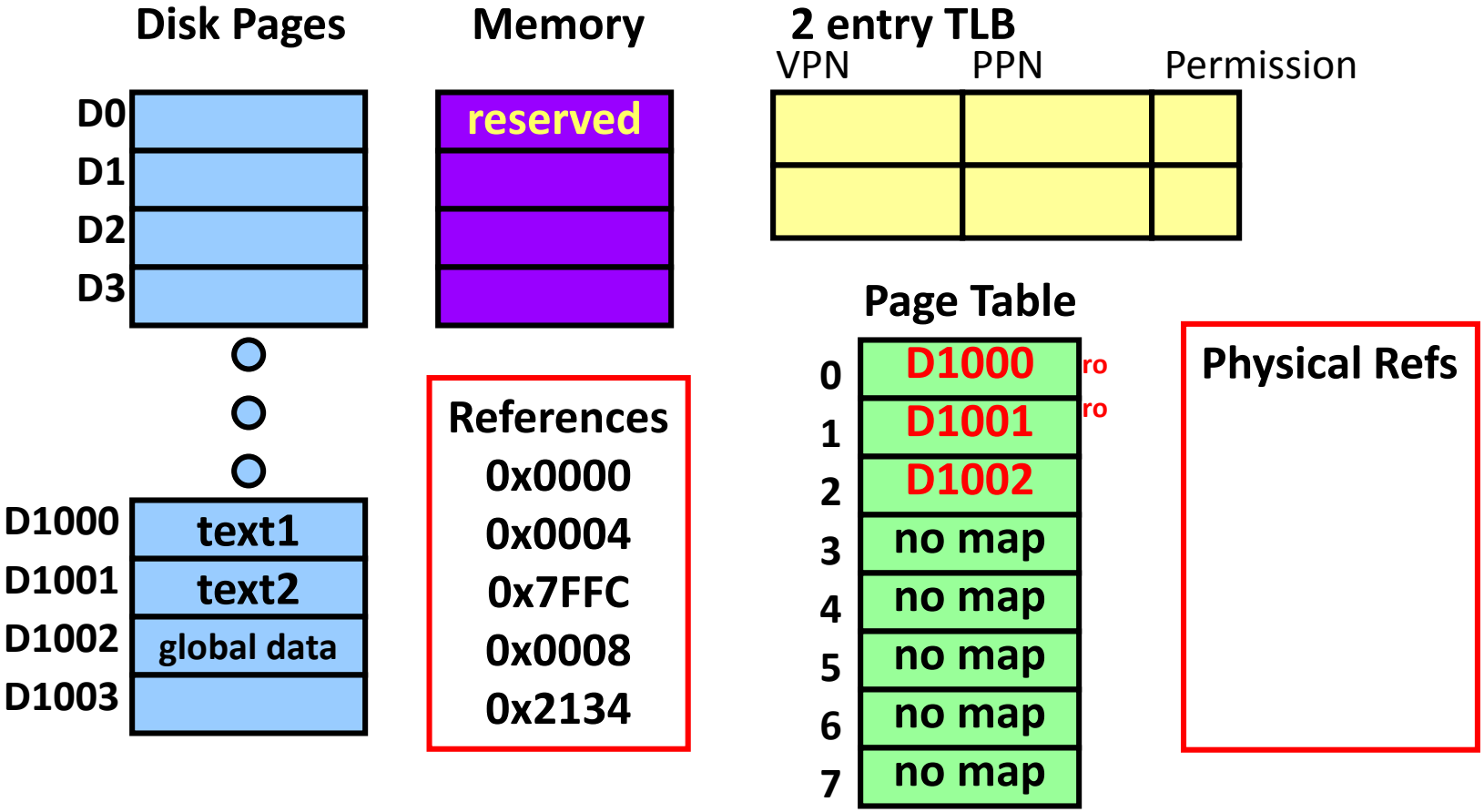
- Loading your program in memory
  - Ask operating system to create a new process
  - Construct a page table for this process
  - Mark all page table entries as invalid with a pointer to the disk image of the program
    - That is, point to the executable file containing the binary.
  - Run the program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

Loading a program into memory

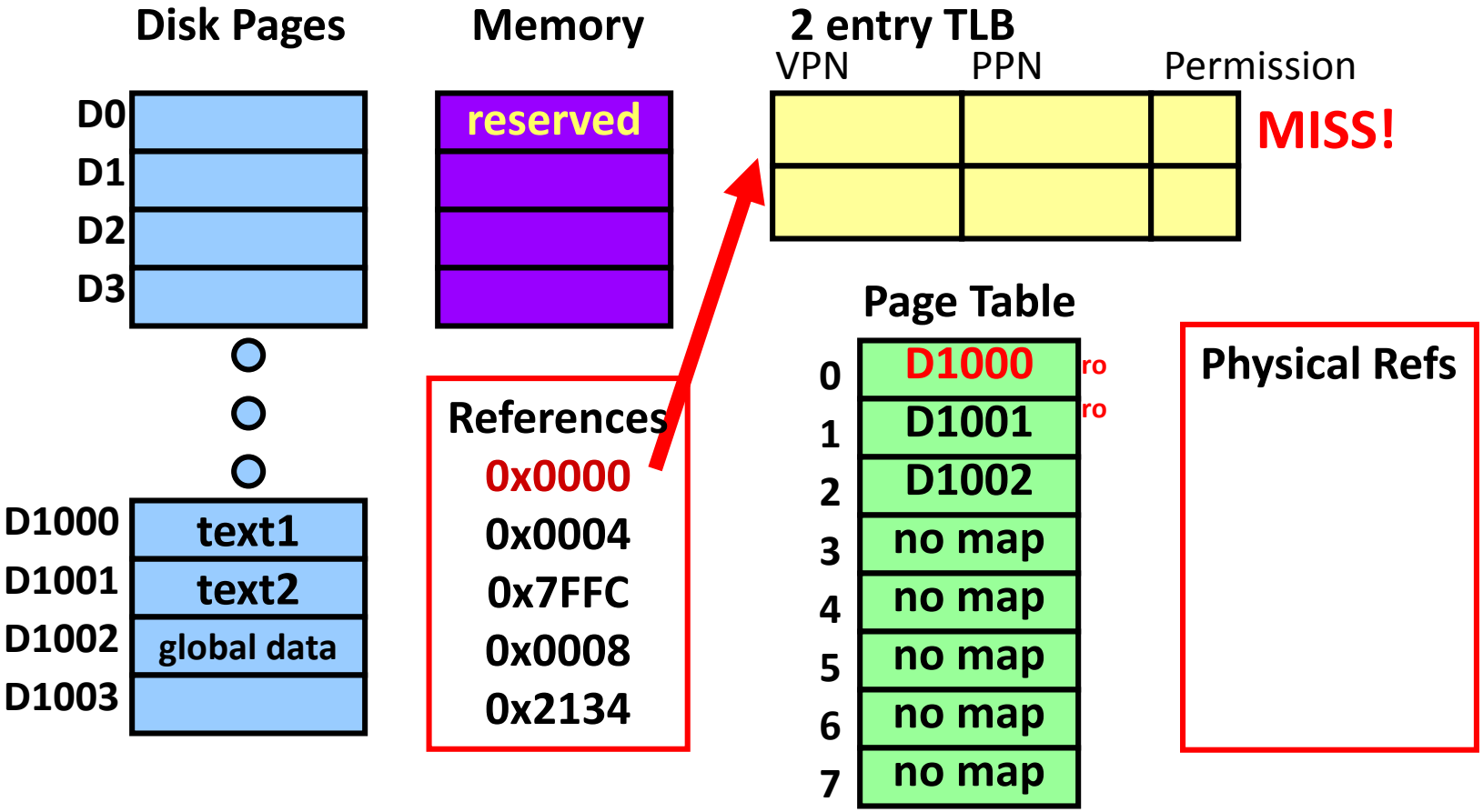
- Page size = 4 KB, Page table entry size = 4 B
- Page table register points to physical address 0x0000



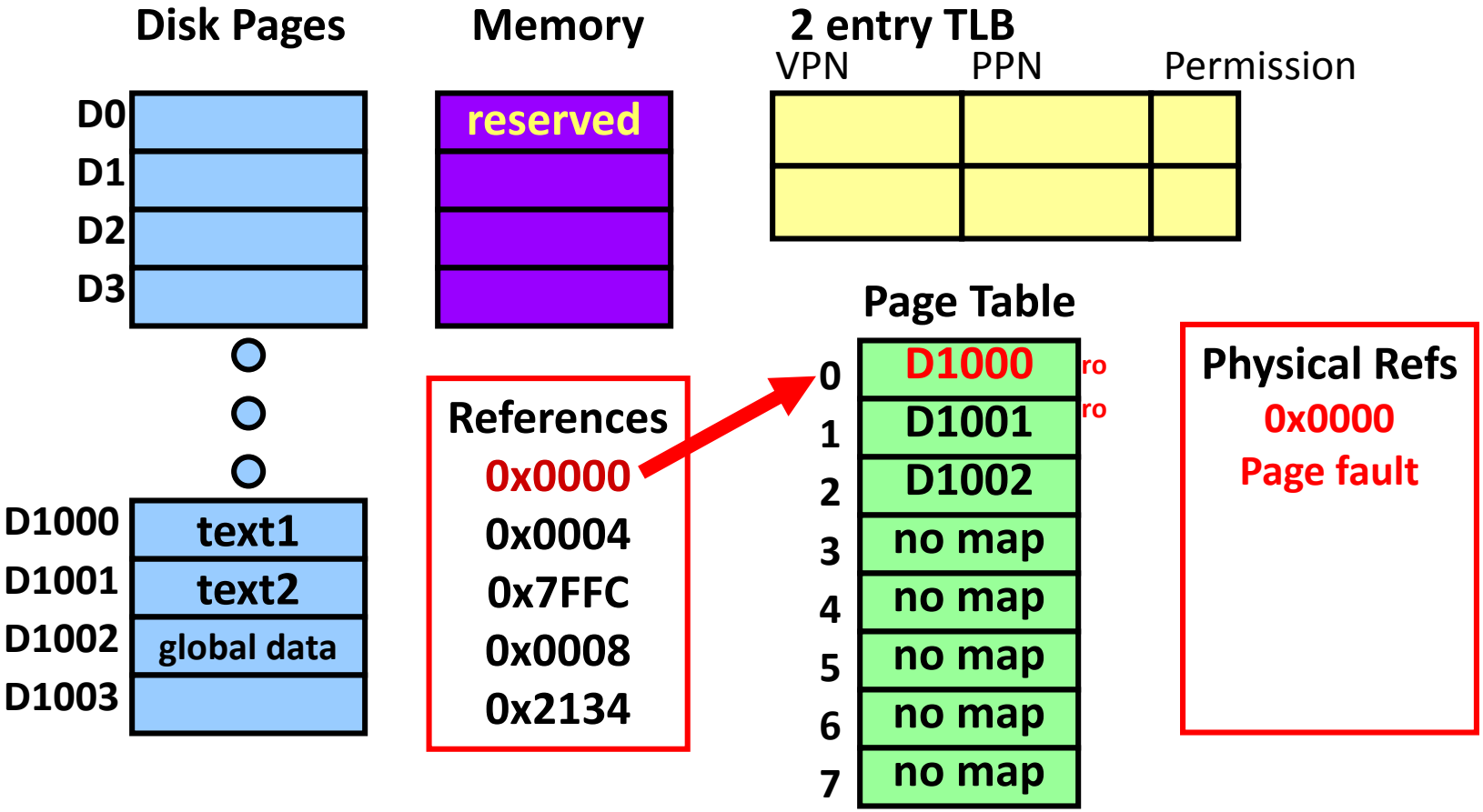
Step 1: Read executable header & initialize page table



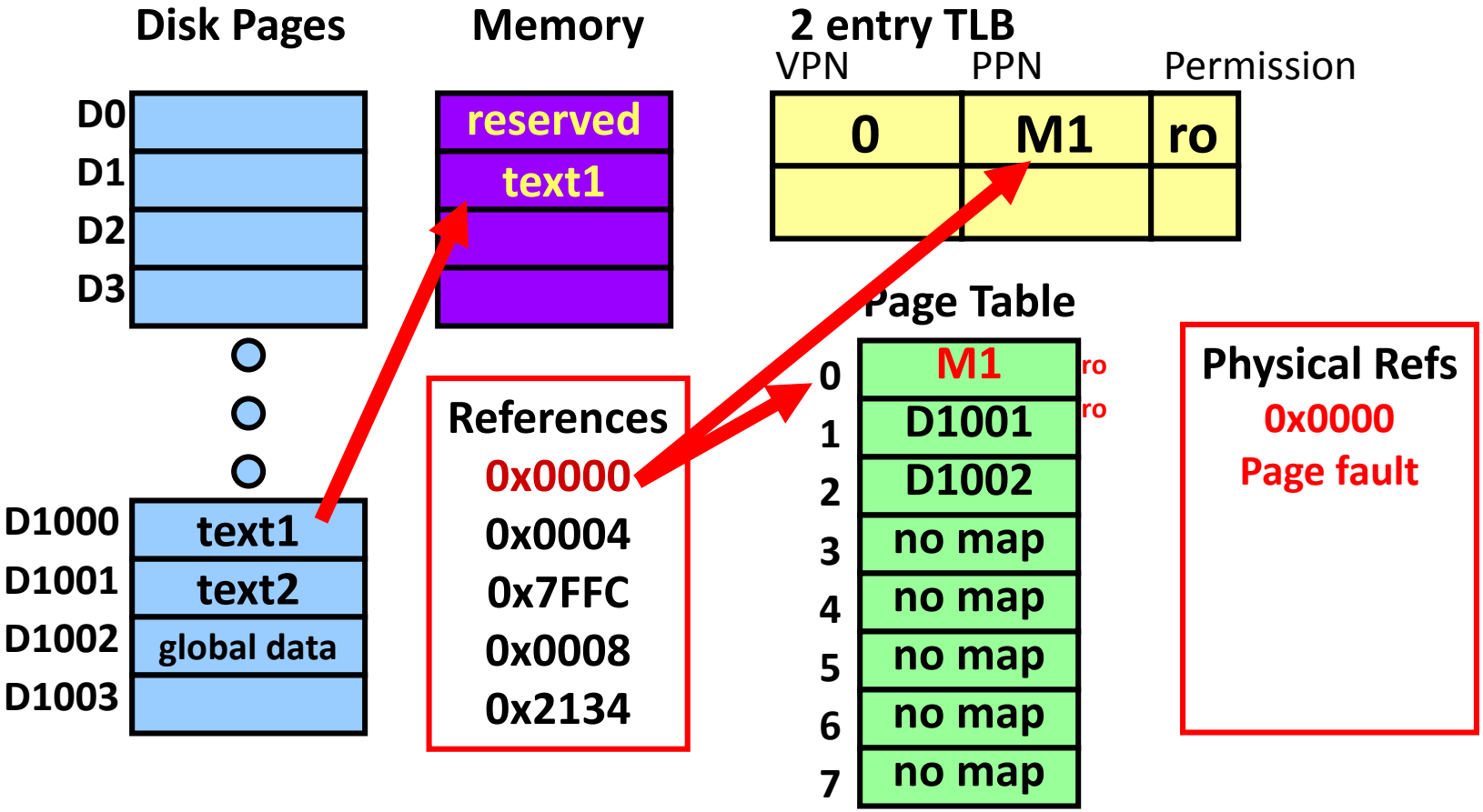
Step 2: Load PC from header & start execution



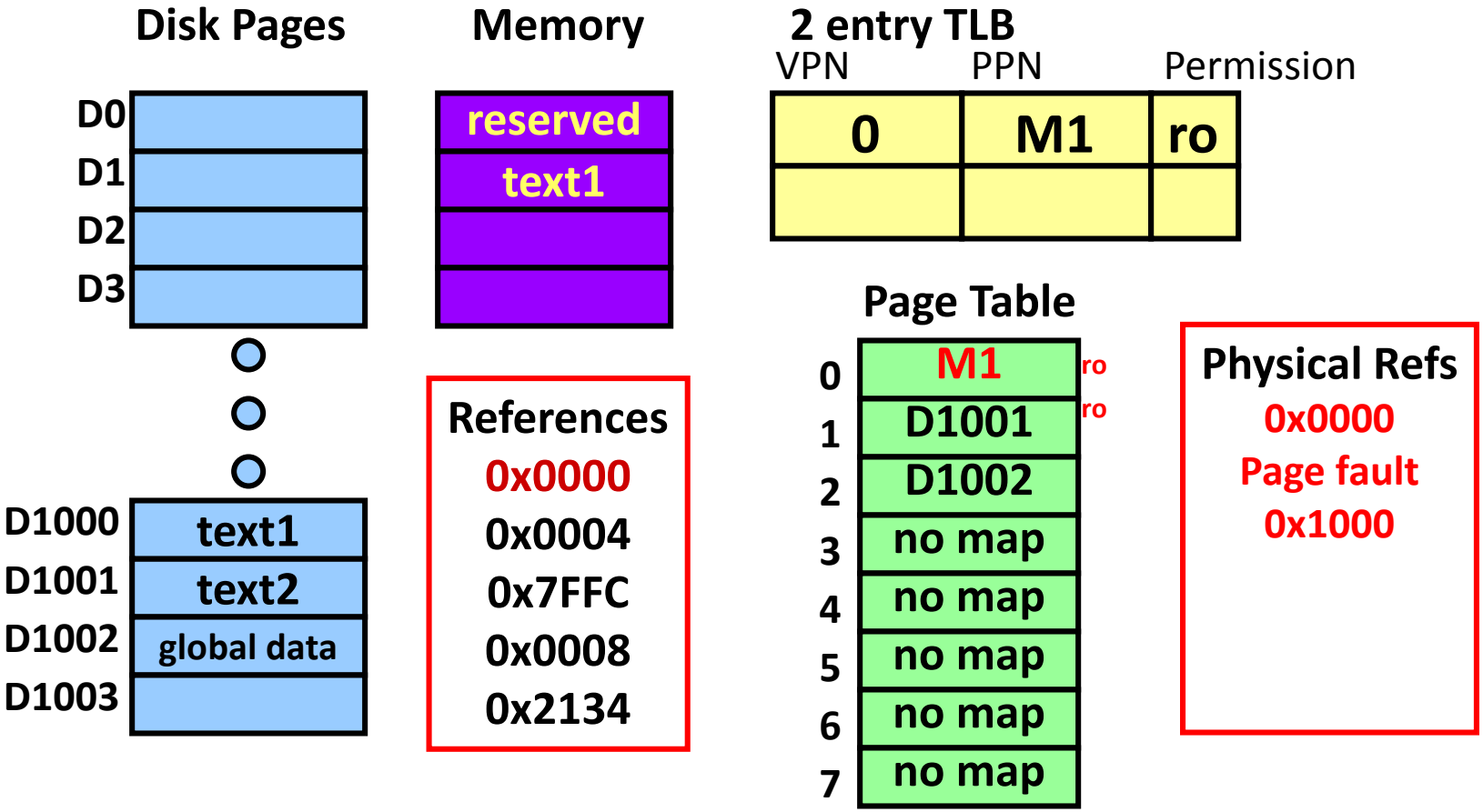
Fetching instruction 0000



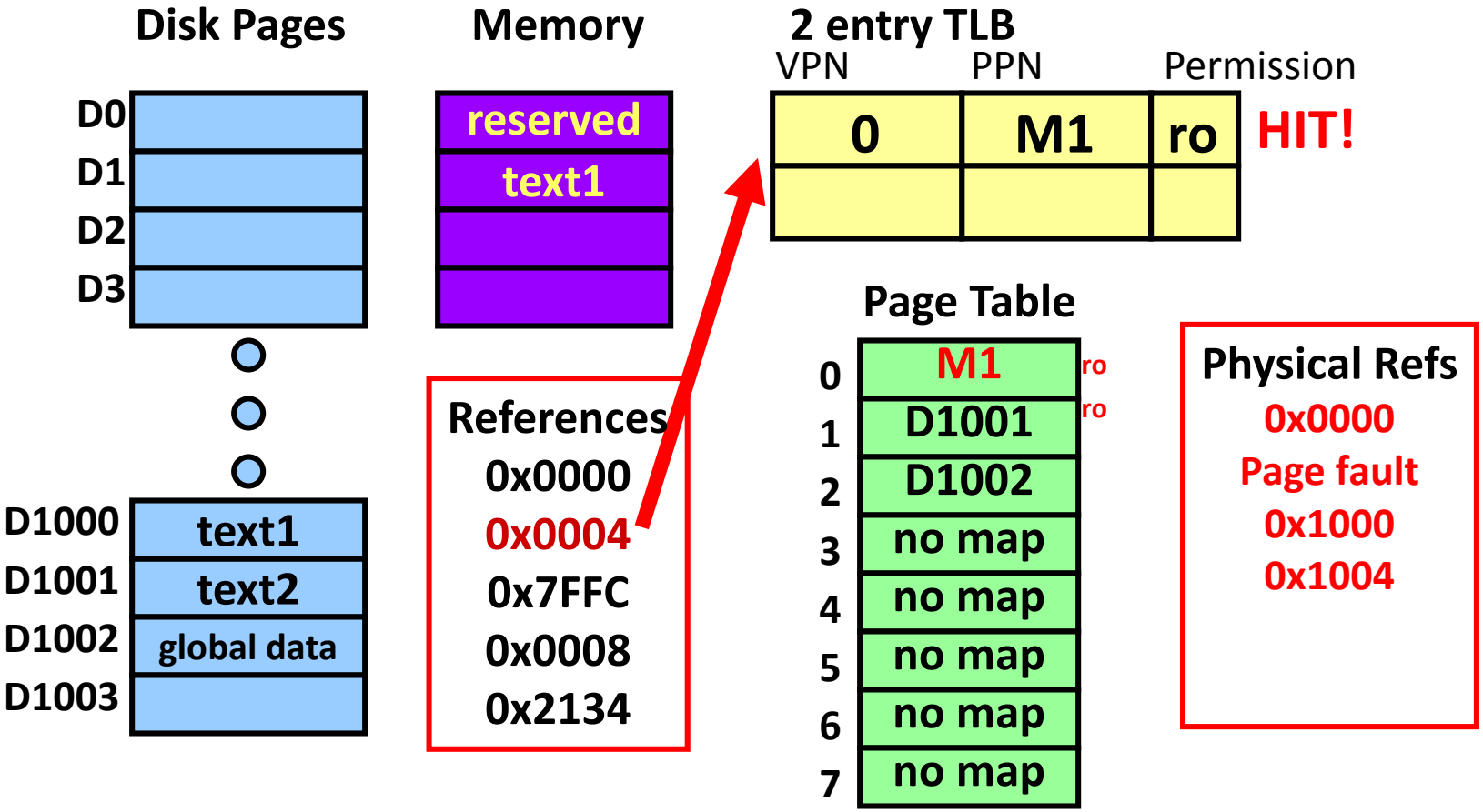
Fetching instruction 0000



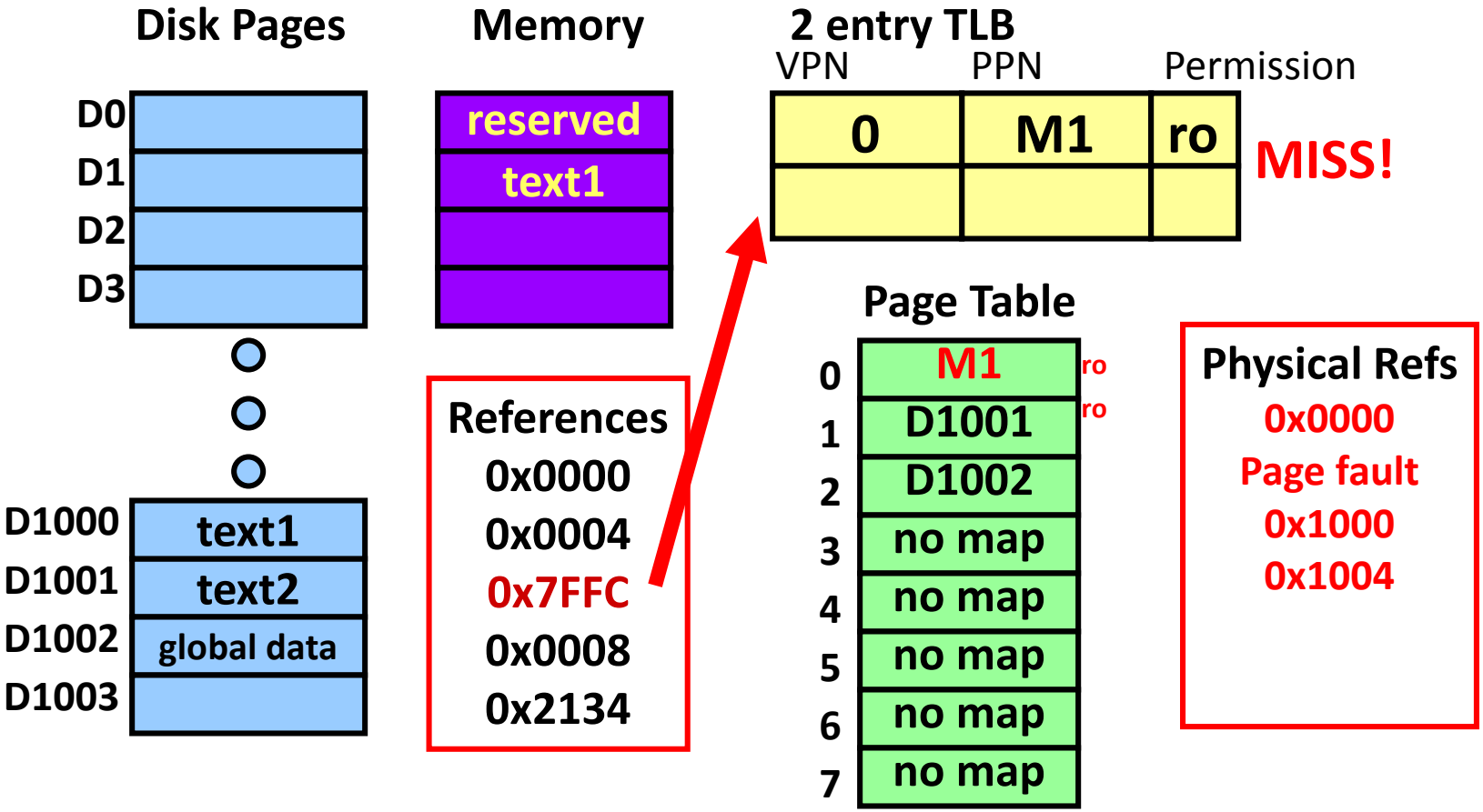
Fetching instruction 0000



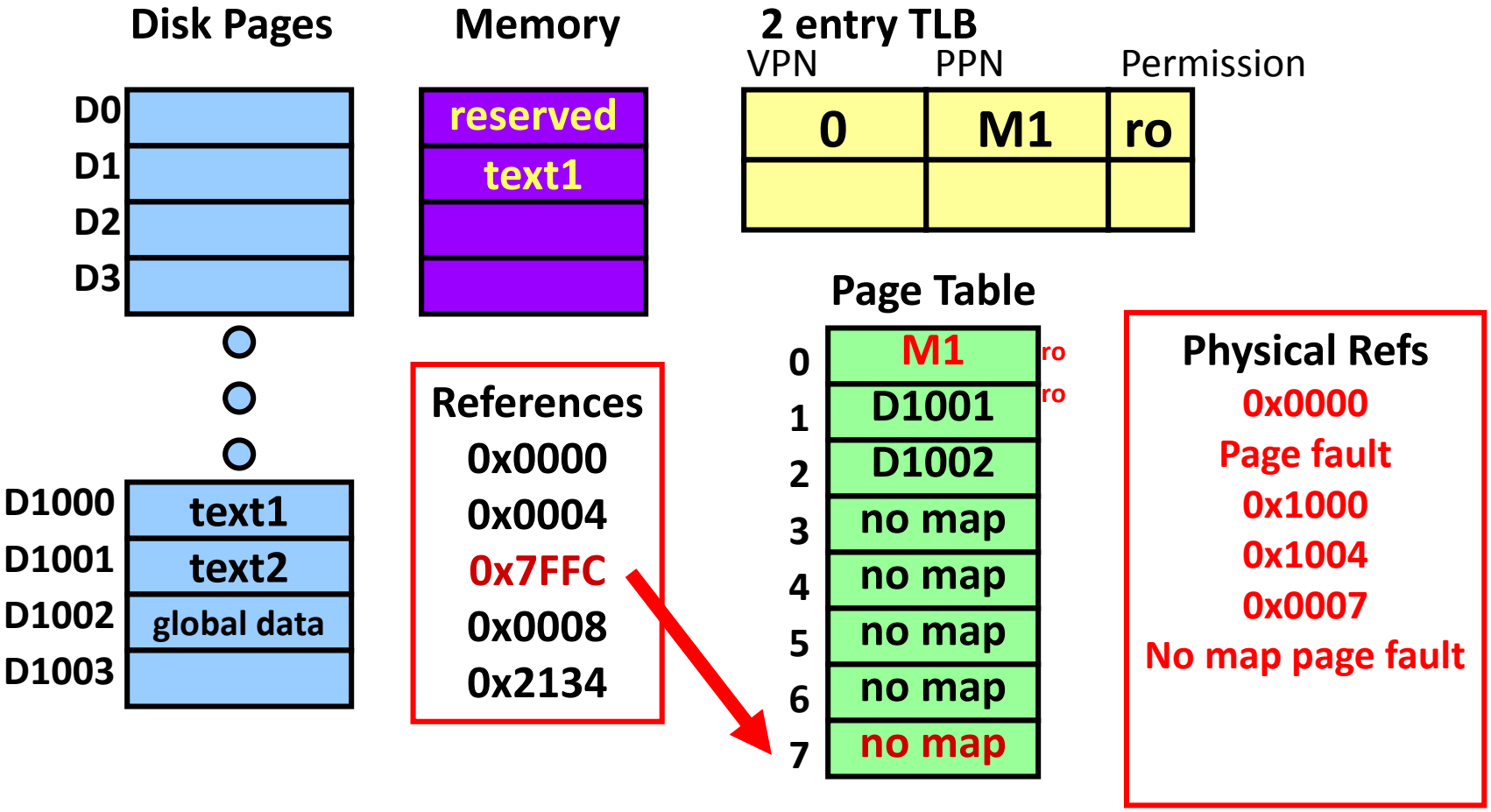
Fetching instruction 0004



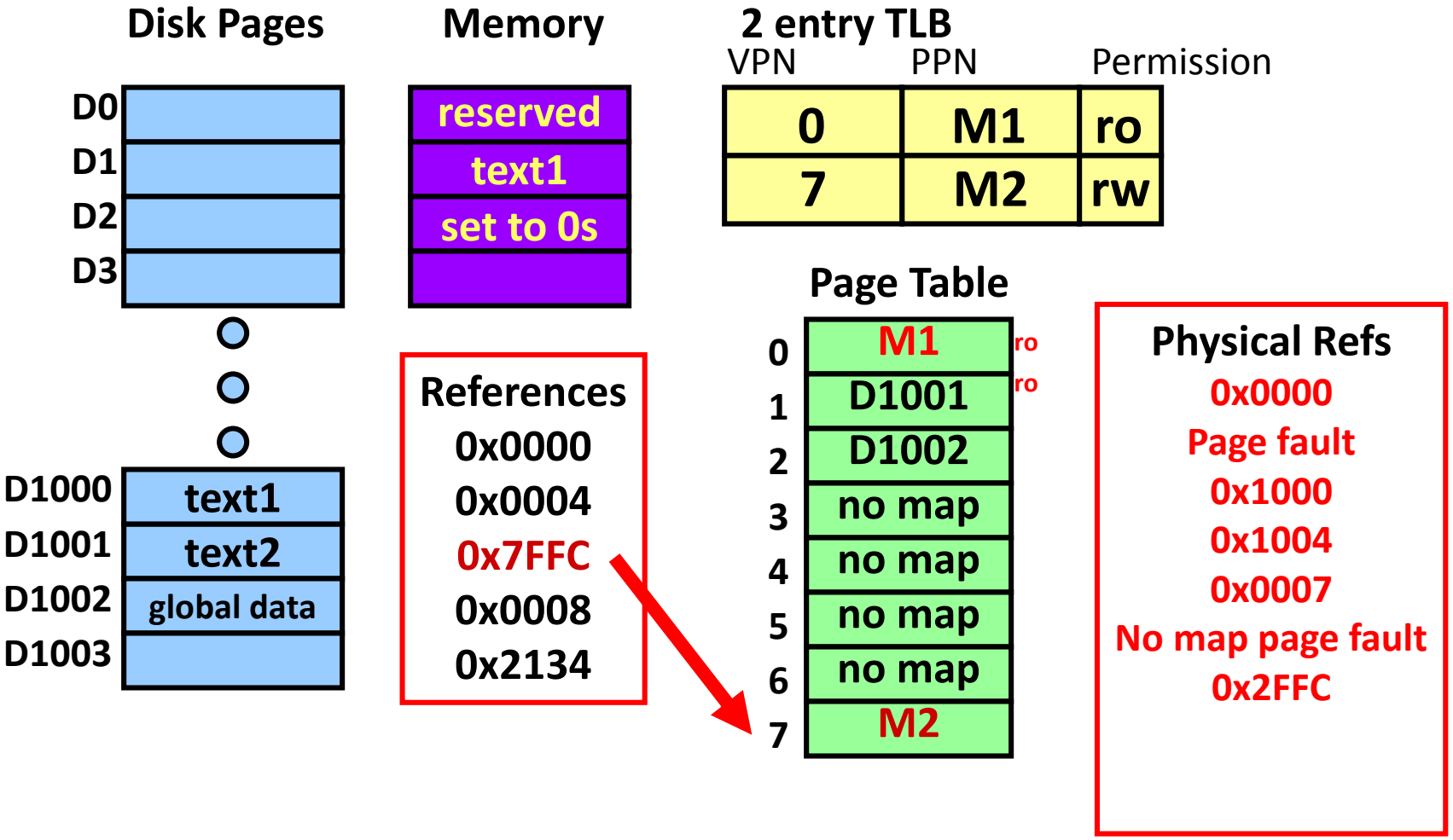
Reference 7FFC



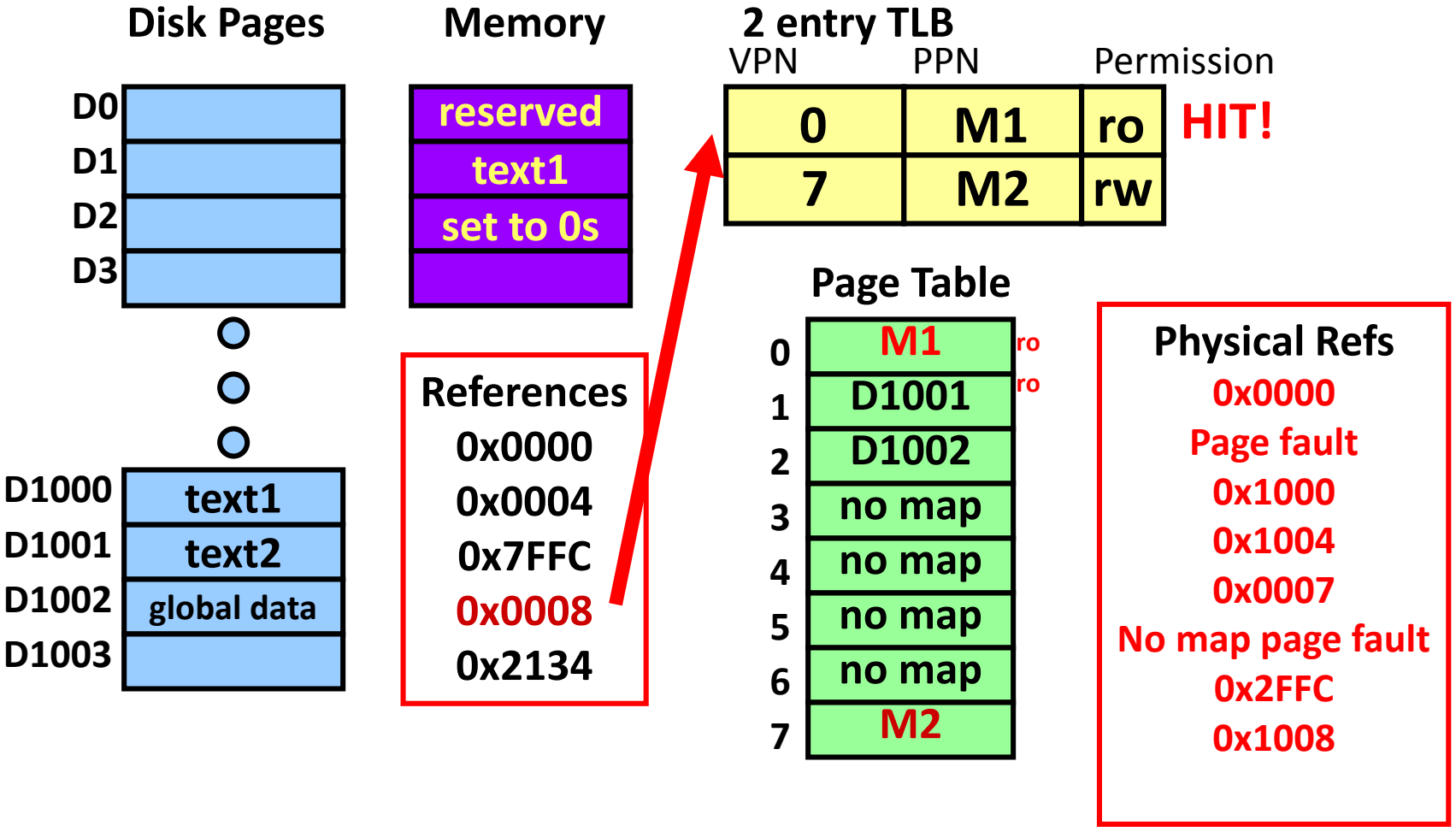
Reference 7FFC



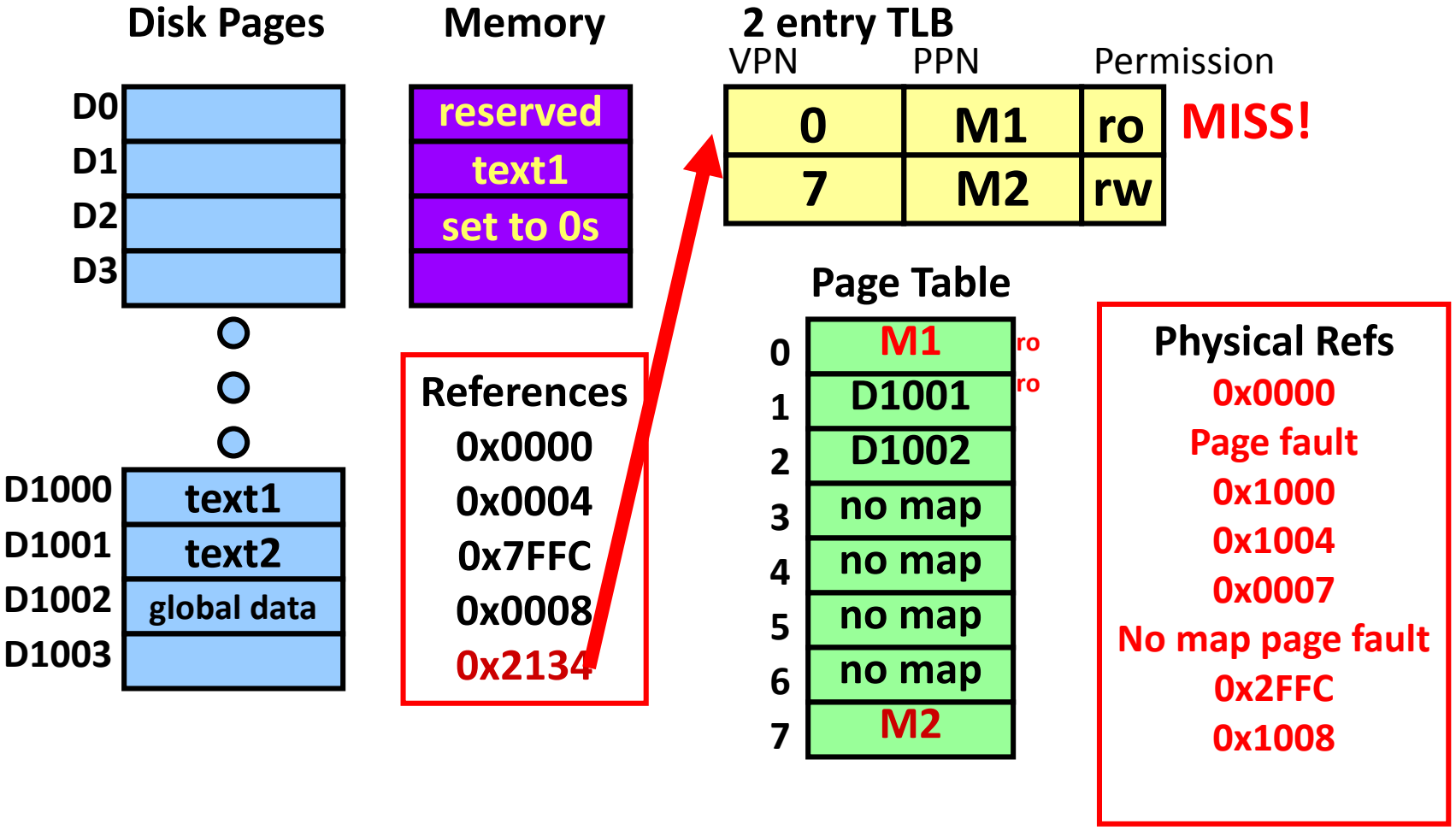
Reference 7FFC



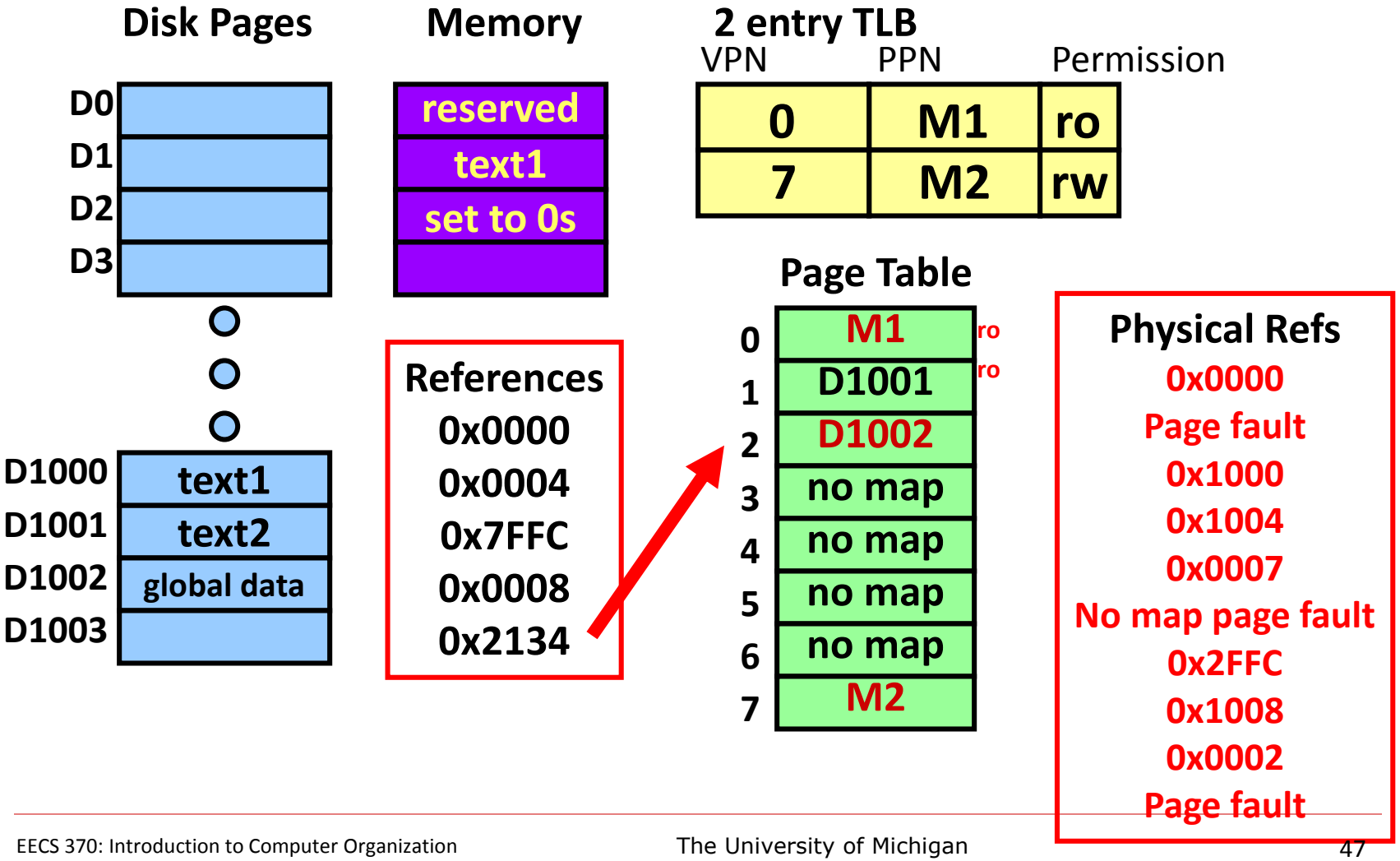
Fetching instruction 0008



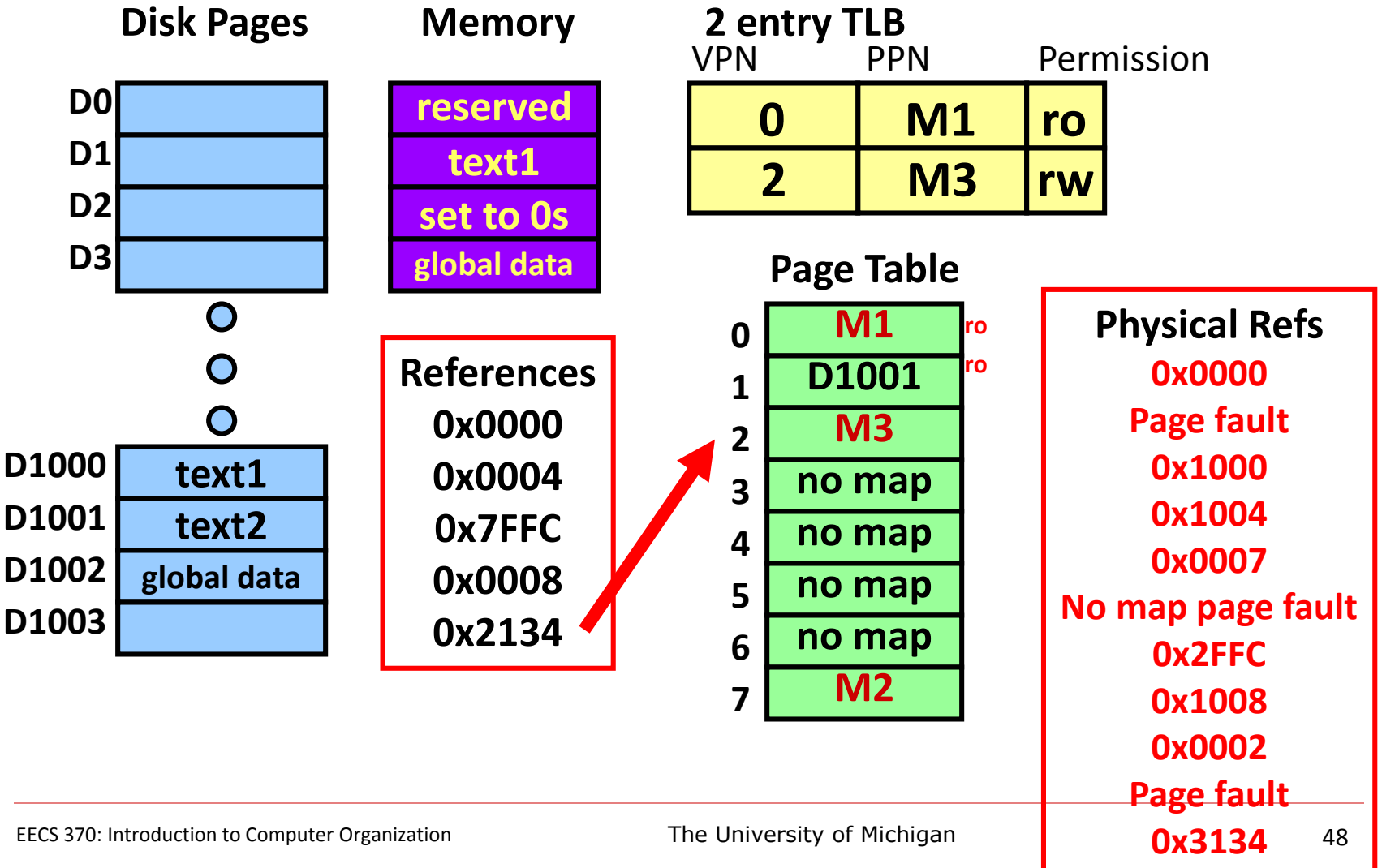
Reference 2134



Reference 2134



Reference 2134



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.

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).
    - Delayed access.
  - Virtual address (before VM translation).
    - Faster access.
    - More complex.

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