信息科学与技术学院
School of Information Science and Technology

CS 110
Computer Architecture
Virtual Memory

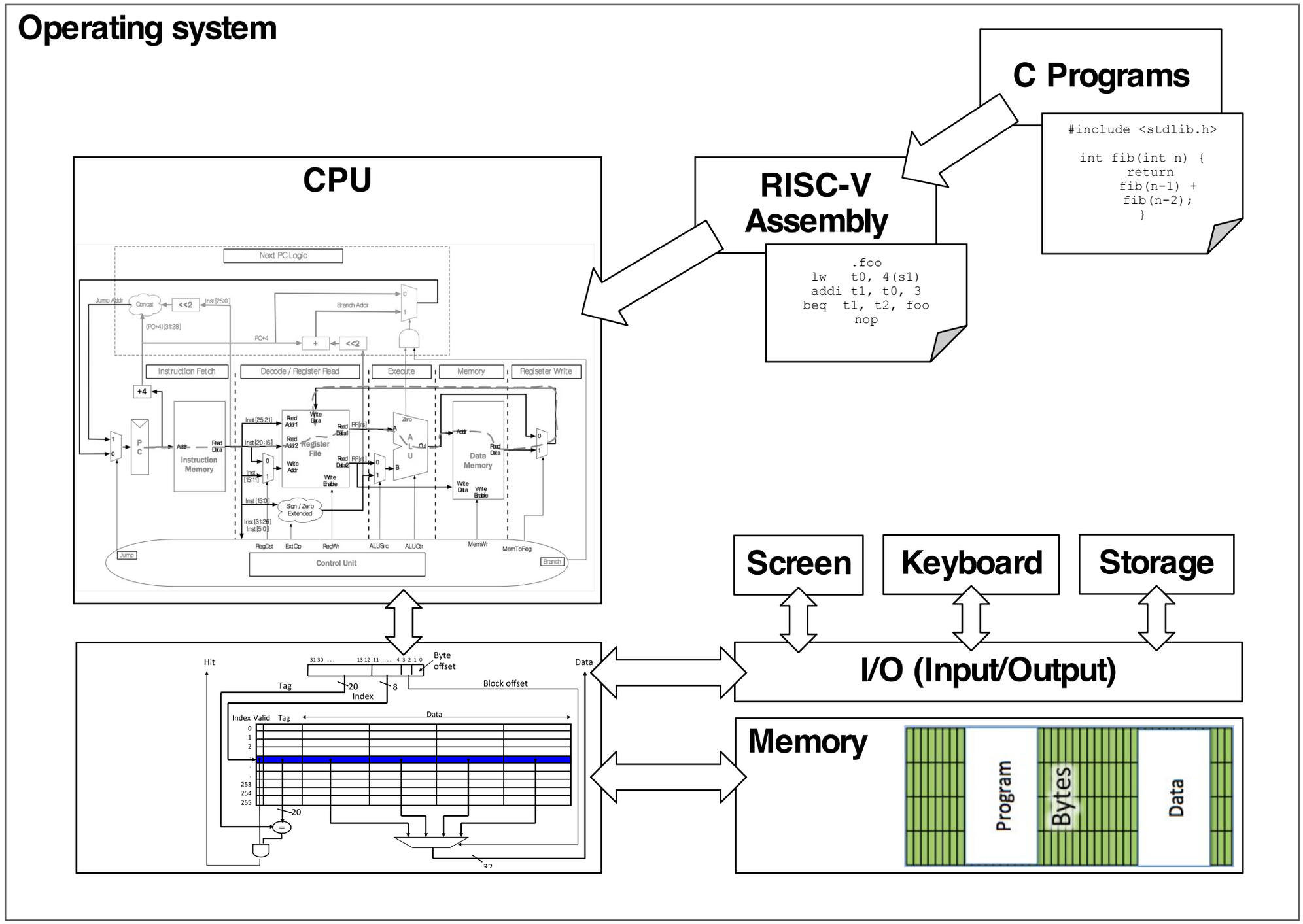
Instructors:
Chundong Wang, Siting Liu & Yuan Xiao
Course website: https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2025/index.html
School of Information Science and Technology (SIST)
ShanghaiTech University

2025/5/27

Administratives

- Final exam, June 12th 8am-10am; you can bring **3**-page A4-sized double-sided cheat sheet, **handwritten** only! (Teaching center 201/202/203); the whole course will be covered. No electronic devices (no smart watches, no calculators, etc.)
- All the assignments have been released! ✌️
  - Project 3 ddl approaching, May 29th.
  - Project 4 released, ddl June 3rd. **Will be check the 17th week during lab sessions.**
  - HW 7 ddl approaching, May 30th.
  - **HW 8 released, ddl June 5th.**
  - To check Lab 13 for Monday sessions if not done, May 26th
  - Lab 14 released, to check May 27th, 29th & June 4th (Lab Session 1 only, 1D104); Prepare in advance!
- Discussion May 30th & June 6th on *Final Review*.

CA up to now ...

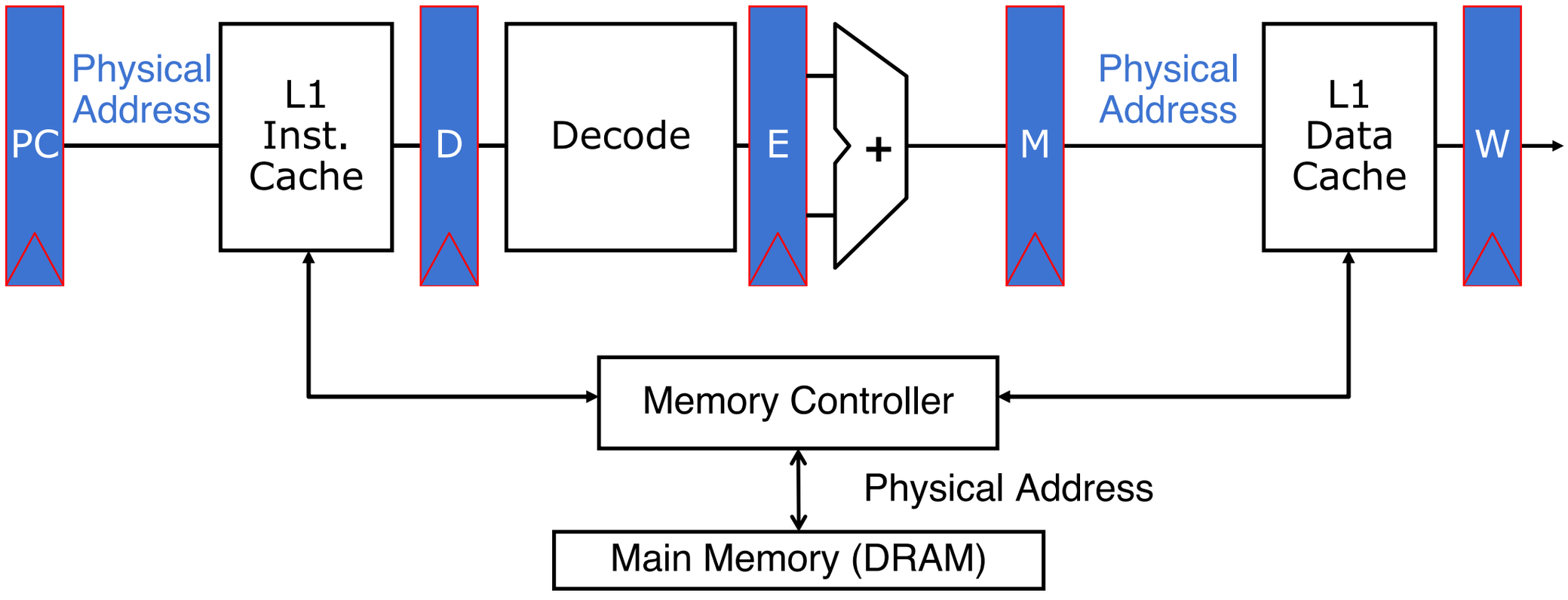

Operating system
C Programs
#include <stdlib.h>
int fib(int n) {
return
fib(n-1) +
fib(n-2);
}
RISC-V Assembly
.foo
lw t0, 4(s1)
addi t1, t0, 3
beq t1, t2, foo
nop
CPU
Next PC Logic
Instruction Fetch
Decode / Register Read
Execute
Memory
Register Write
Instruction Memory
Register File
Data Memory
Control Unit
Screen
Keyboard
Storage
I/O (Input/Output)
Memory
Program
Bytes
Data
Hit
Data
Tag
Index
Byte offset
Block offset

Protection, Translation, Paging

- Supervisor mode does not fully isolate applications from each other or from the OS.
  - Application could overwrite another application's memory.
  - Also, may want to address more memory than we actually have (e.g., for sparse data structures).
- Solution: **Virtual Memory** (this lecture). Gives each process the illusion of a full memory address space that it has completely for itself.

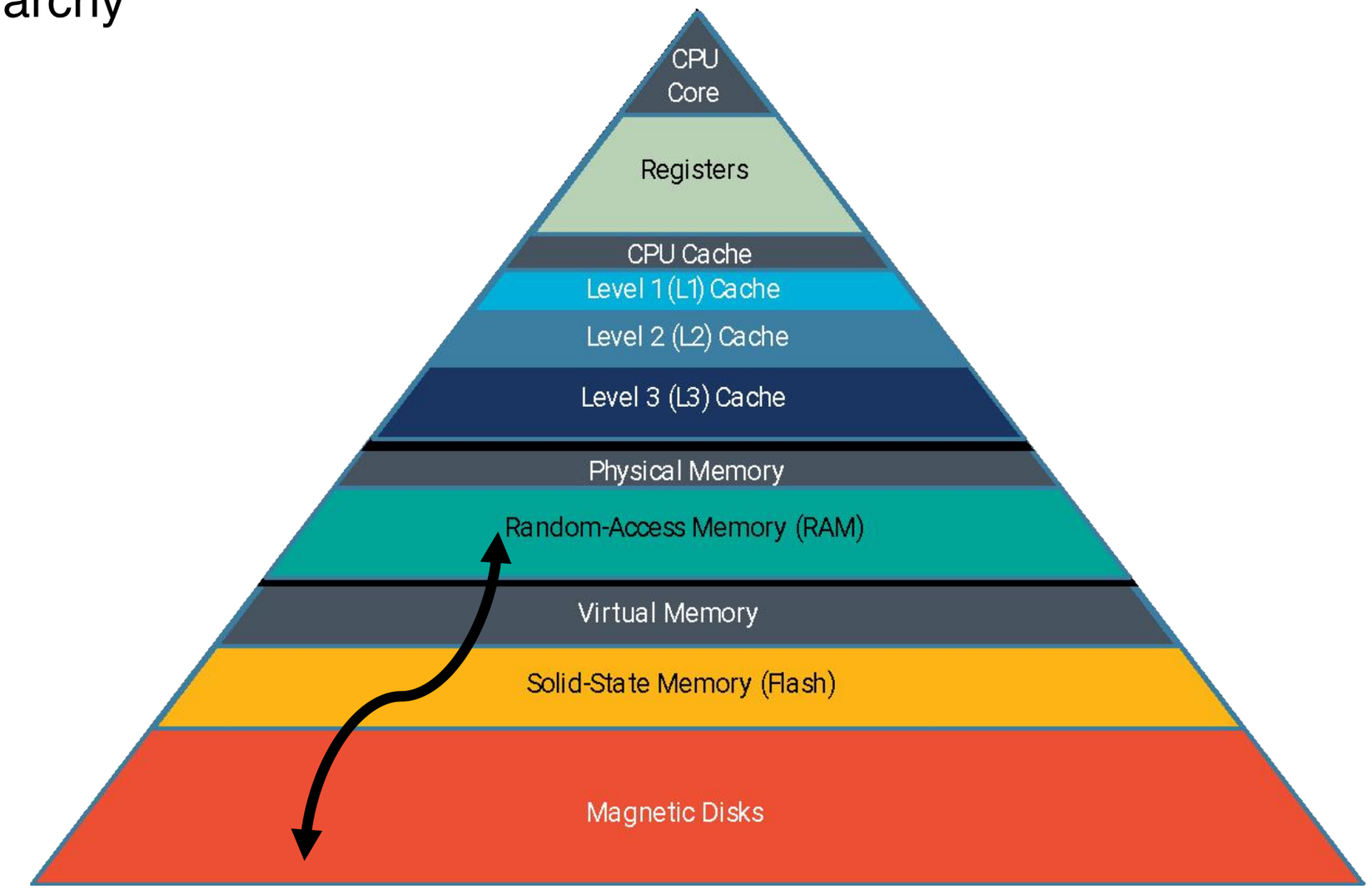
"Bare 5-Stage Pipeline"

- In a bare machine, the only kind of address is a physical address



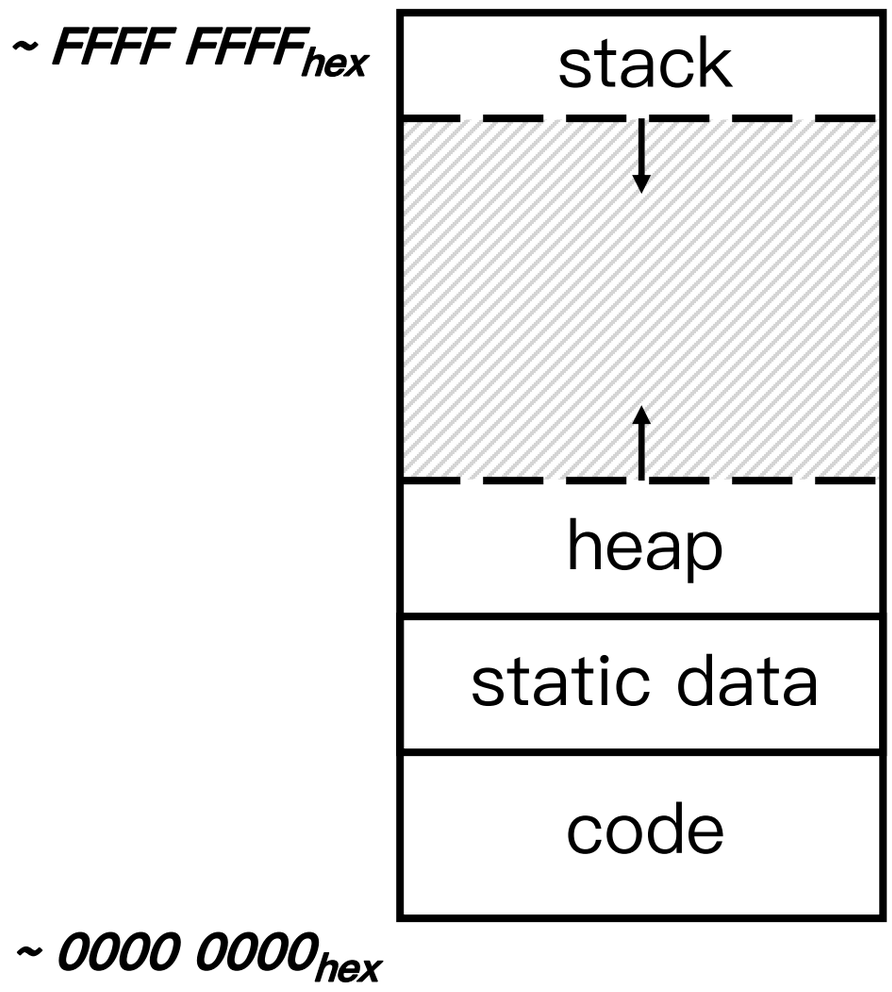
Motivation for Virtual Memory

- Adding disks to memory hierarchy
  - Need to devise a mechanism to “connect” memory and disk in the memory hierarchy



Motivation for Virtual Memory

- Adding disks to memory hierarchy
  - Need to devise a mechanism to “connect” memory and disk in the memory hierarchy
- Simplifying memory for applications
  - Applications should see the straightforward memory layout we saw earlier ->
  - User-space applications should think they own all of memory
  - So we give them a virtual view of memory



Motivation for Virtual Memory

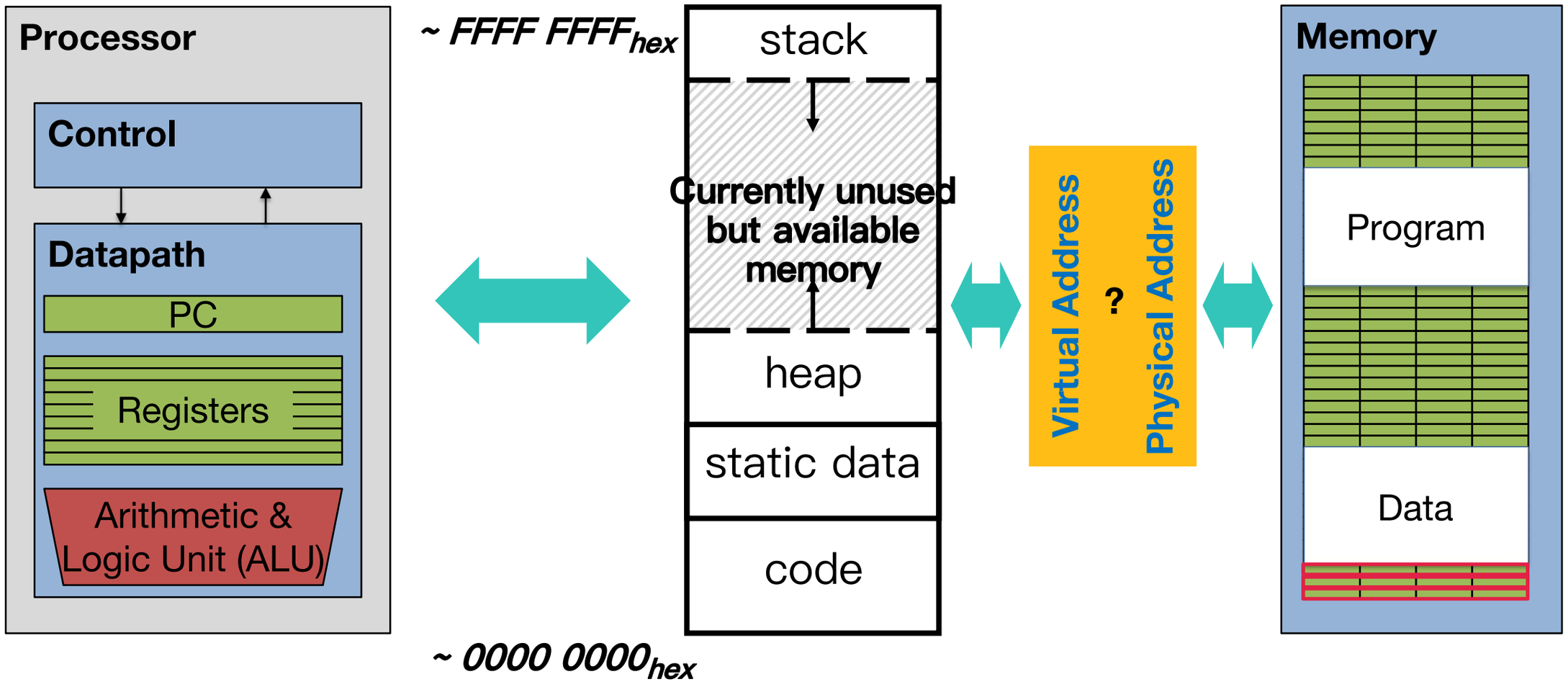
- Adding disks to memory hierarchy
  - Need to devise a mechanism to “connect” memory and disk in the memory hierarchy
- Simplifying memory for applications
  - Applications should see the straightforward memory layout we saw earlier ->
  - User-space applications should think they own all of memory
  - So we give them a virtual view of memory
- Protection between processes
  - With a bare system, addresses issued with loads/stores are real physical addresses
  - This means any program can issue any address, therefore can access any part of memory, even areas which it doesn’t own

Address Spaces

- The set of addresses labeling all of memory that we can access
- Now, 2 kinds:
  - Virtual Address Space - the set of addresses that the user program knows about
  - Physical Address Space - the set of addresses that map to actual physical cells in memory
- Hidden from user applications
- So, we need a way to map between these two address spaces
  - We should send all addresses through a mechanism that the OS controls, before they make it out to DRAM - **a translation mechanism**

Virtual vs. Physical Addresses

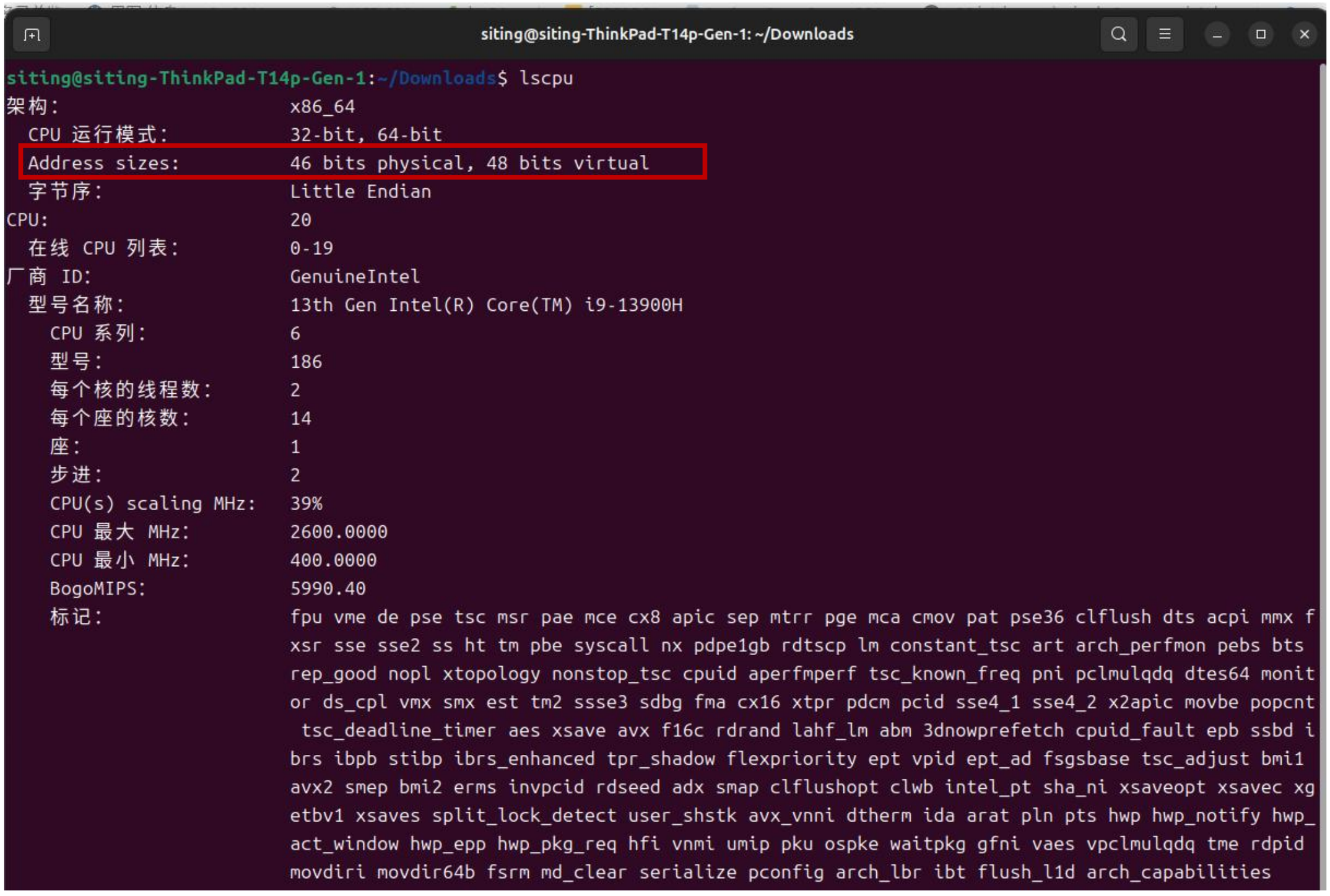
- Processes use virtual addresses, e.g., 0 ~ 0xFFFF FFFF;
  - Many processes, all using same (conflicting) addresses
- Memory uses physical addresses (also, e.g., 0 ~ 0xFFFF FFFF)
  - Memory manager maps virtual to physical addresses



Many of these (software & hardware cores)

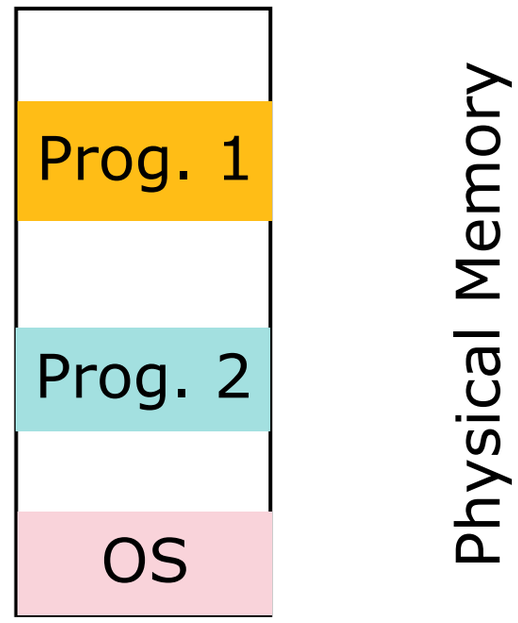
One main memory

Virtual vs. Physical Addresses-`lscpu`



Dynamic Address Translation

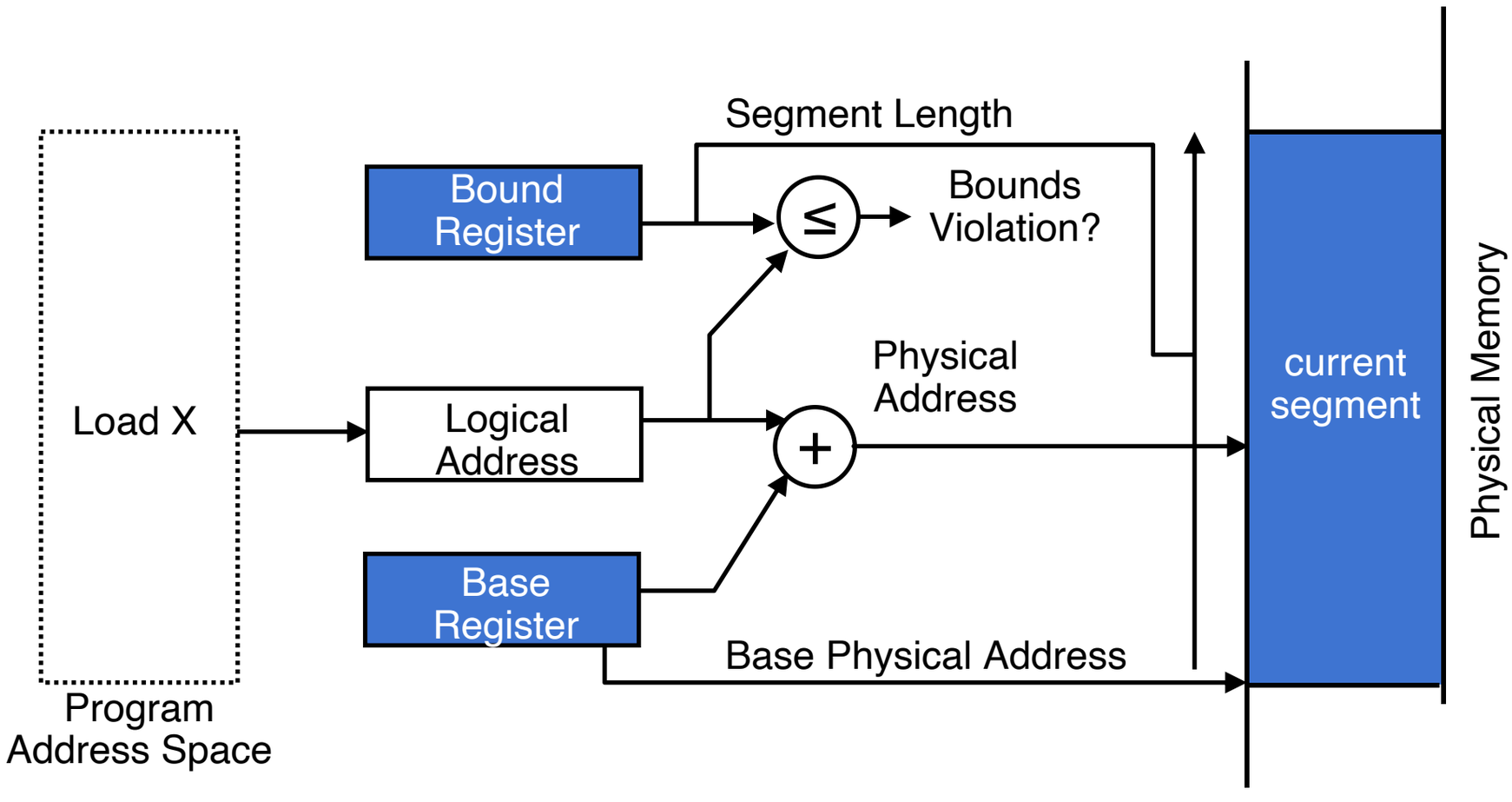
- Motivation
  - Multiprogramming, multitasking: Desire to execute more than one process at a time (more than one process can reside in main memory at the same time).
- Location-independent programs
  - Programming and storage management ease
- => **base register** ← add offset to each address
- Protection
  - Independent programs should not affect
  - each other inadvertently
- => **bound register** ← check range of access



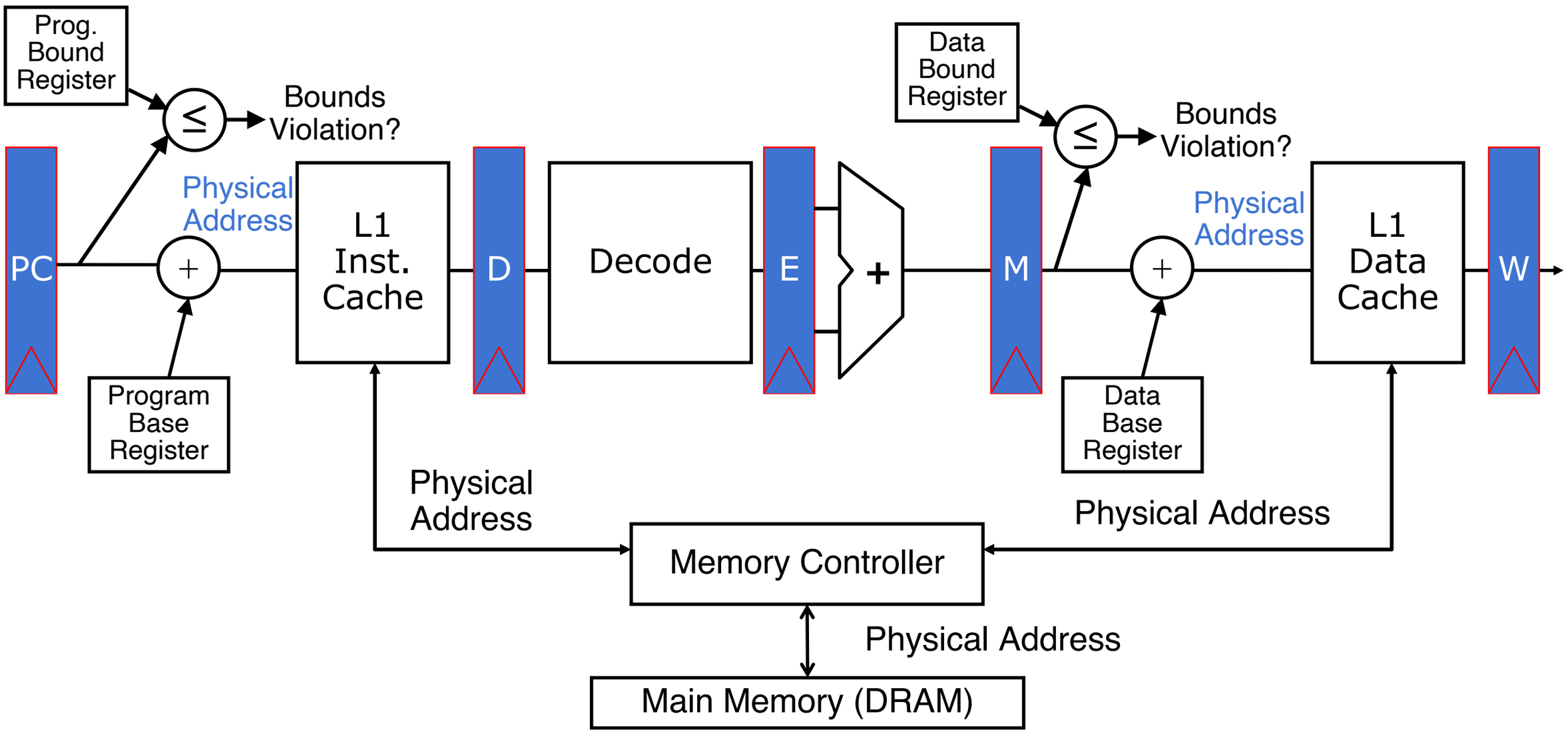
Note: Multiprogramming drives requirement for resident supervisor (OS) software to manage context switches between multiple programs

Simple Base and Bound Translation

- Base and bounds registers are visible/accessible only when processor is running in supervisor mode



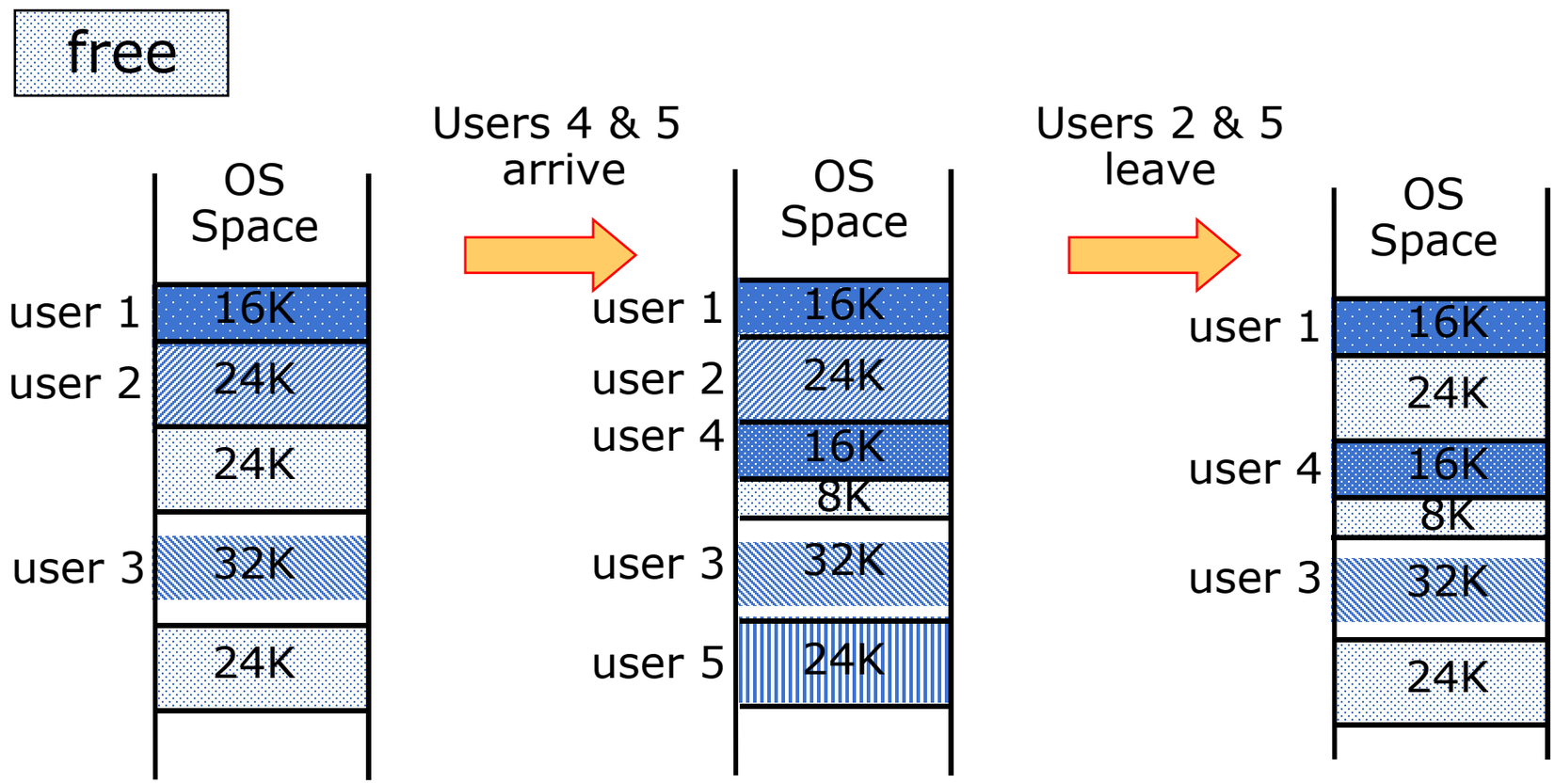
Base and Bound Machine



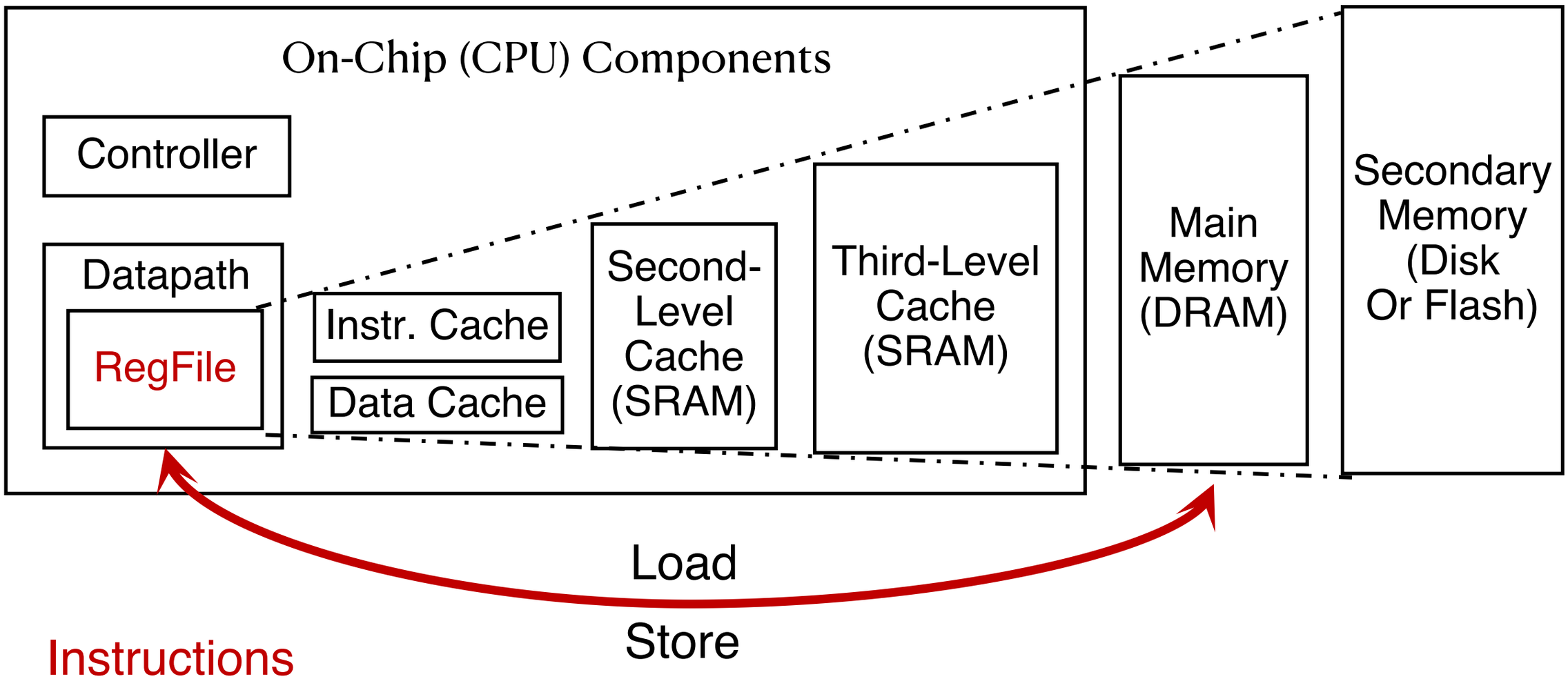
- Can fold addition of base register into (register+immediate) address calculation using an adder

Memory Fragmentation

- As programs come and go, the storage is “fragmented”.
- Therefore, at some stage programs have to be moved around to compact the storage.



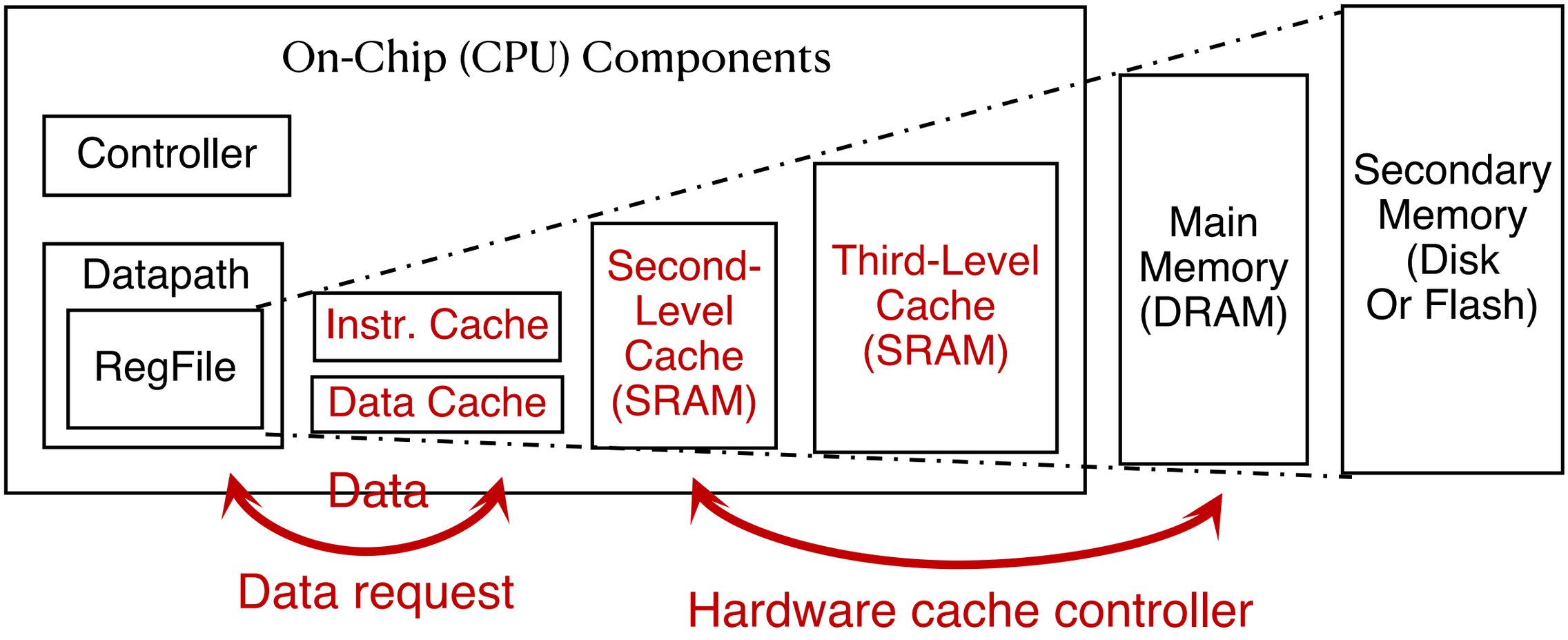
Recall: Memory Hierarchy Management



Instructions

- Either by assembly programers or generated by a compiler;
- Does not define how it is achieved.

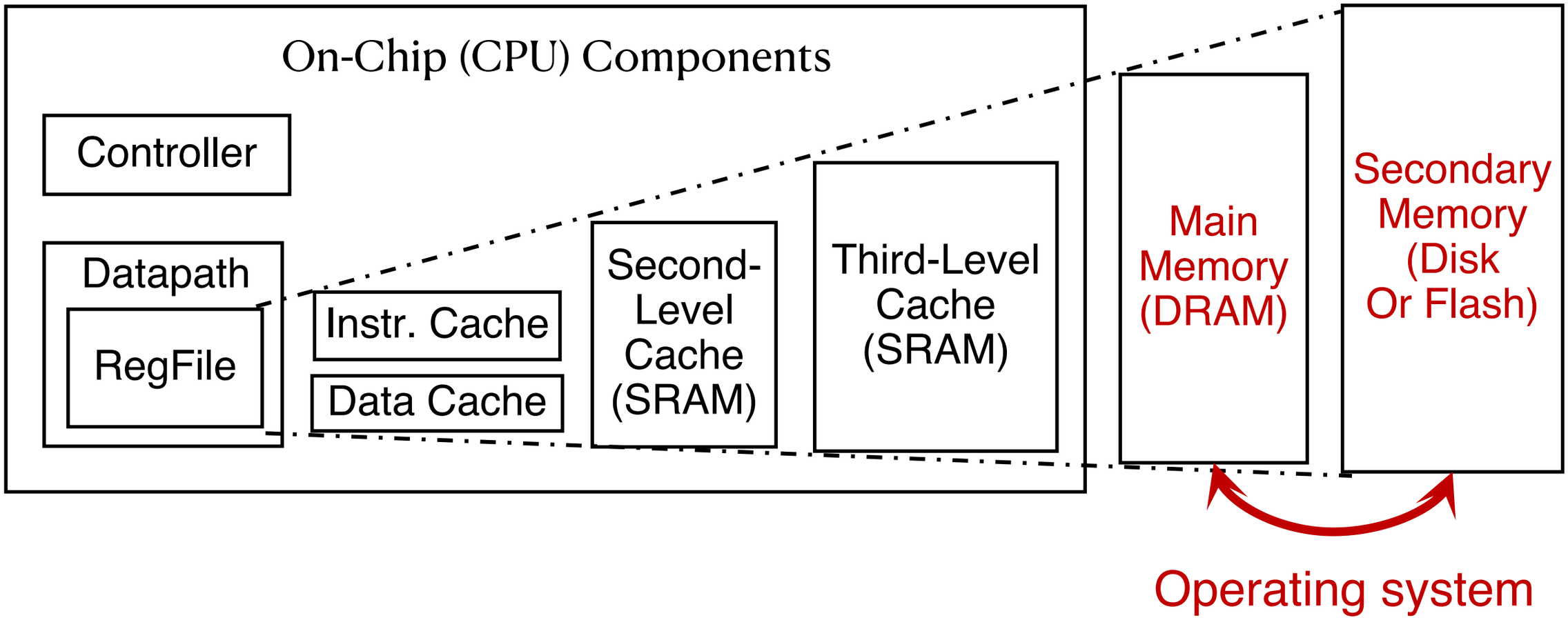
Recall: Memory Hierarchy Management



- With cache, the datapath/core does not directly access the main memory;
- Instead the core asks the caches for data with improved speed;
- A hardware cache controller is deviced to provide the desired data

(with various strategies that will be covered in future lectures).

Recall: Memory Hierarchy Management



- By the operating system (virtual memory)
- Virtual to physical address mapping assisted by the hardware

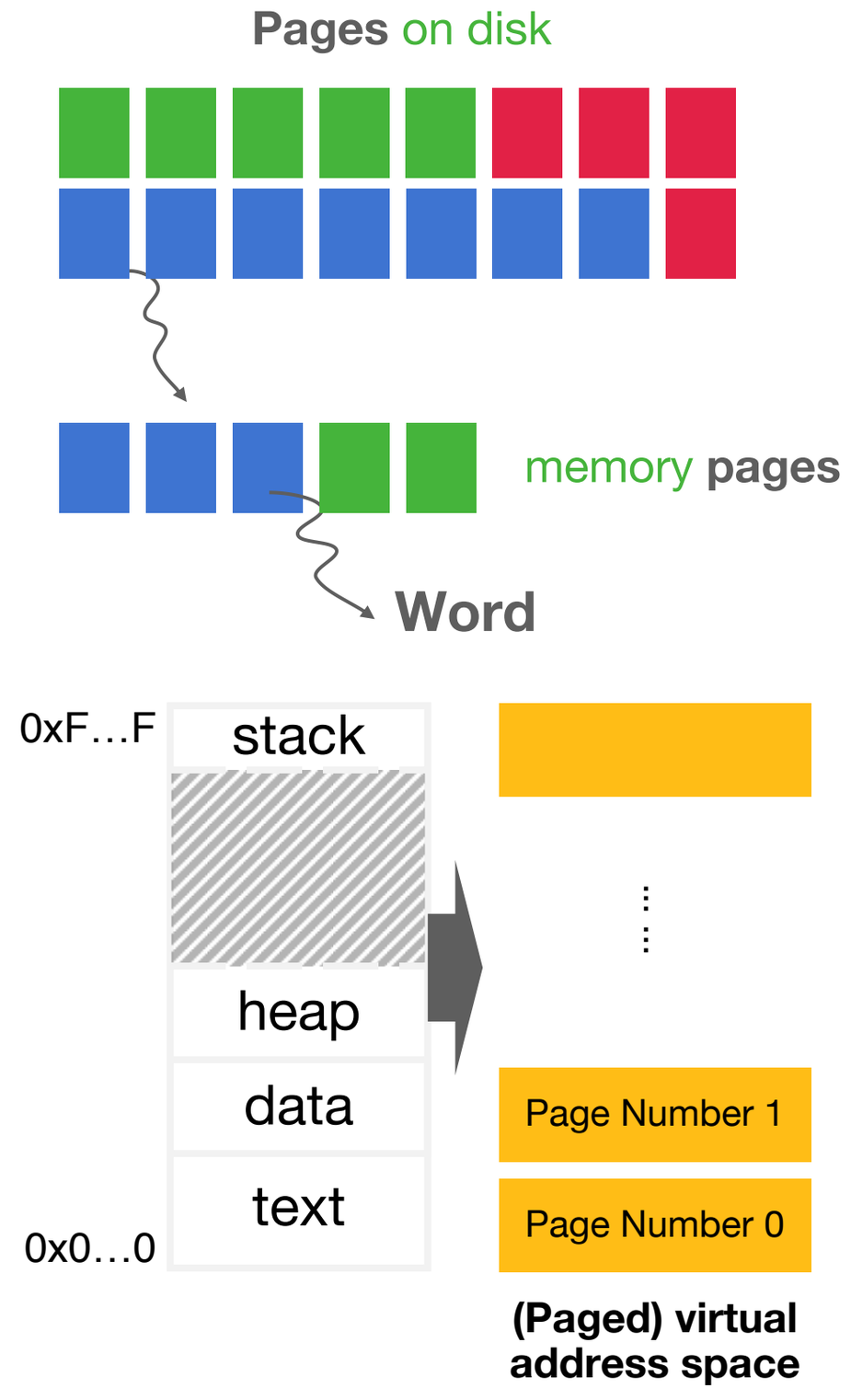
('translation lookaside buffer' or TLB, also a cache, **this lecture**)

Virtual Memory Management

- Map virtual addresses to physical addresses.
- Use both memory and disk.
  - Give illusion of larger memory by storing some content on disk.
  - Disk is usually much larger and slower than DRAM.
- Protection:
  - Isolate memory between processes.

Paged Memory

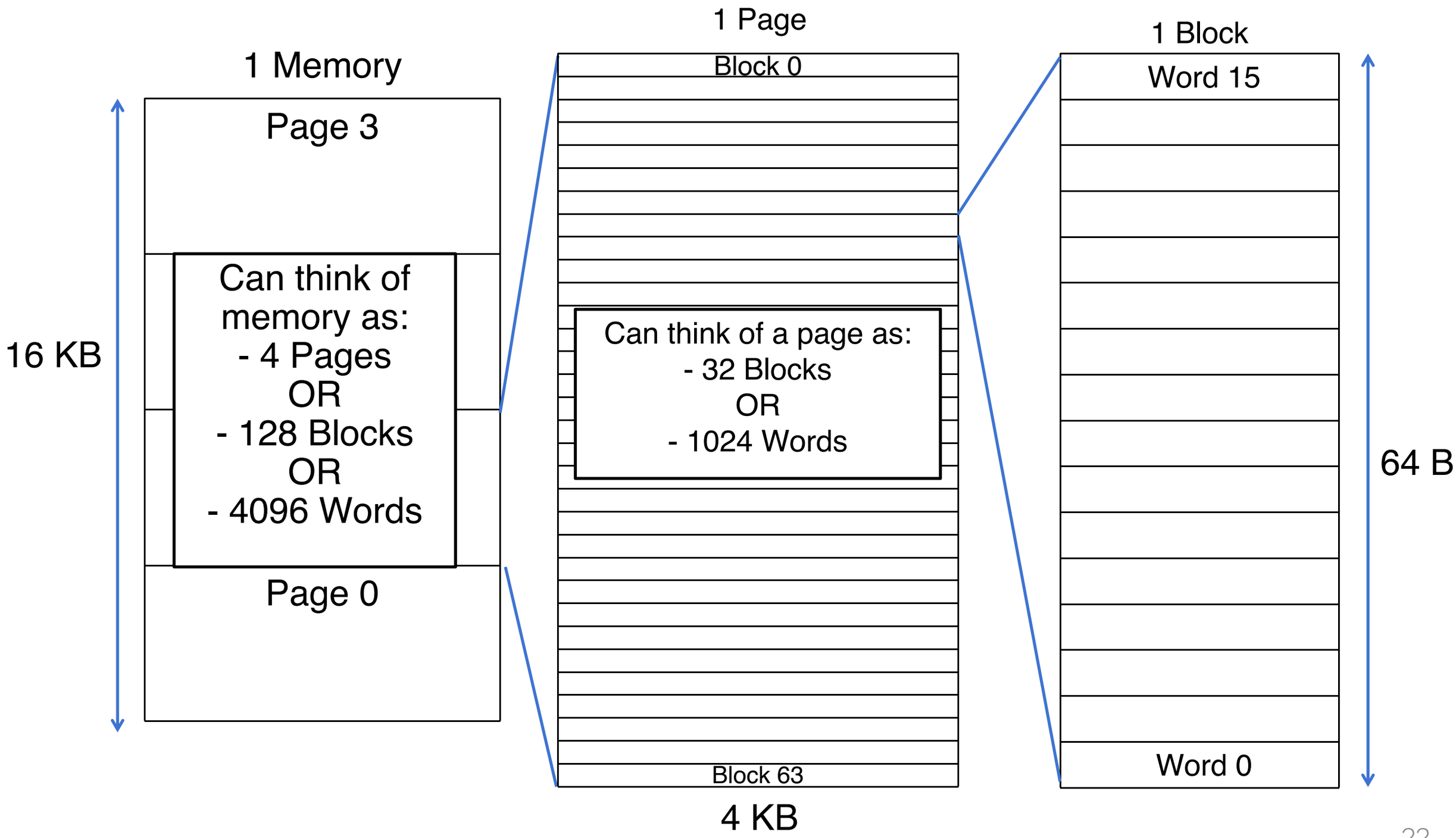
- Physical memory or DRAM is broken into pages.
  - A disk access loads an entire page into memory.
    - Should be large enough to amortize high access time.
- Typical page size: 4 KiB+ (on modern OSes)
  - Need 12 bits of page offset to address all 4 KiB bytes.



Blocks vs. Pages

- In caches, we dealt with individual blocks
  - Usually ~64B on modern systems
  - We could “divide” memory into a set of blocks
- In VM, we deal with individual pages
  - Usually ~4 KB on modern systems
    - Larger sizes also available: 2MB, very modern 1GB!
  - Now, we “divide” memory into a set of pages
- Common point of confusion: bytes, words, blocks, pages are all just different ways of looking at memory!

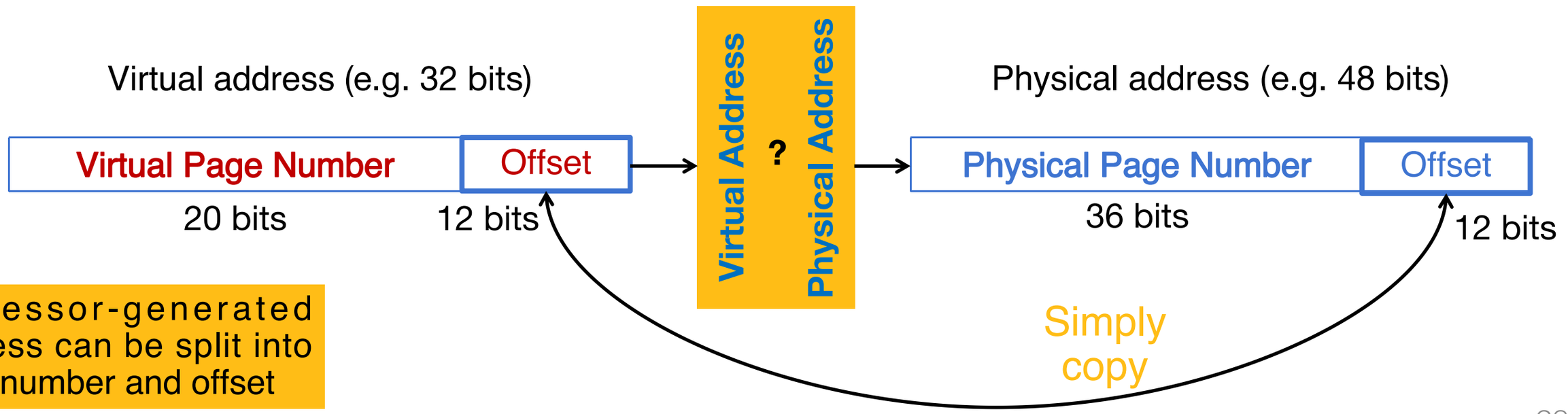
Bytes, Words, Blocks, Pages



Address Translation

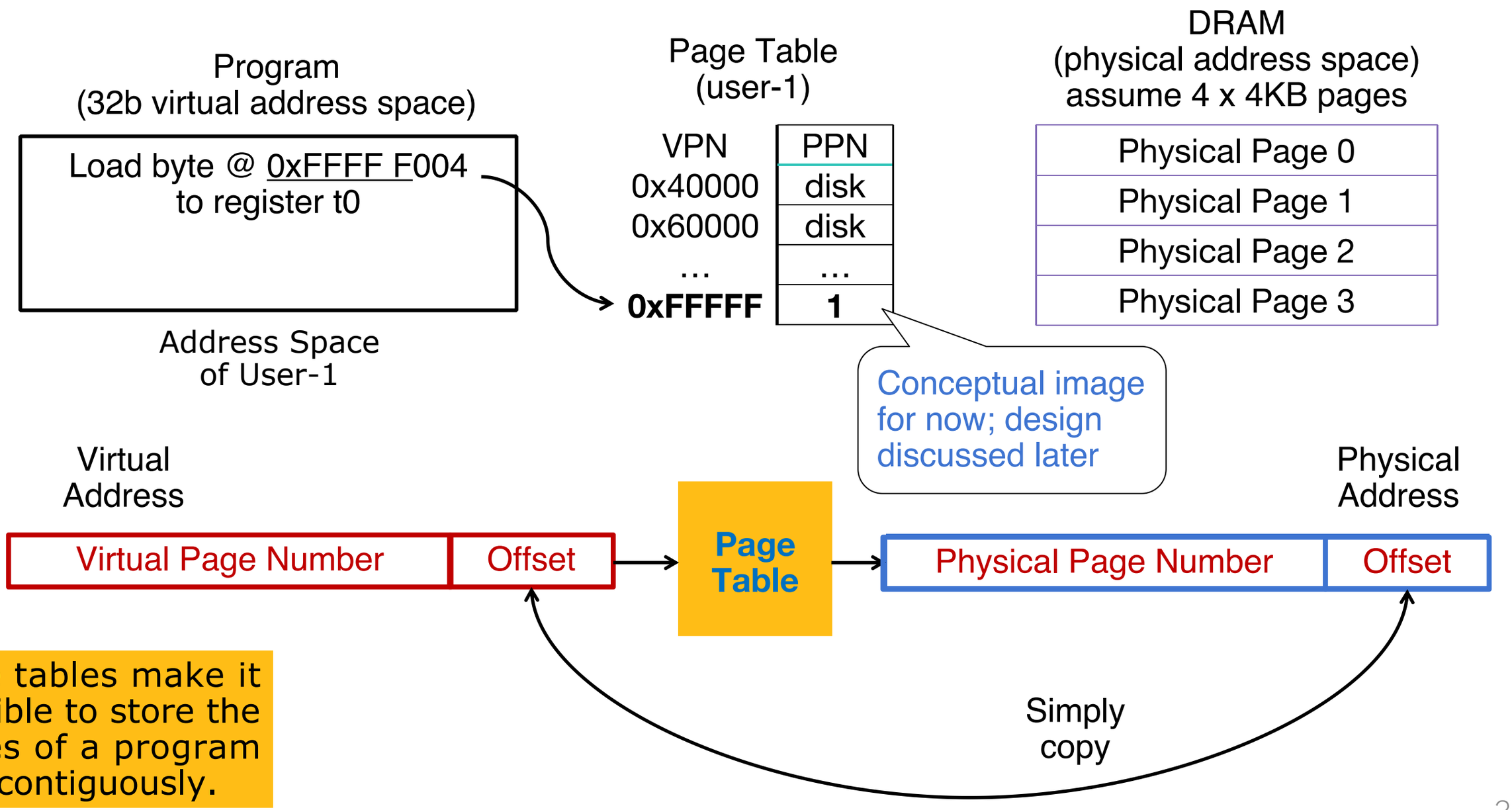
- A page table translates virtual addresses to physical addresses for a given process. Each entry in the table:
- Corresponds to a virtual page number.
- If page is in memory, entry has corresponding physical page number.
- Else, entry should tell OS to trigger page fault to load page from disk.

- Memory translation maps a **Virtual Page Number (VPN)** to a **Physical Page Number (PPN)**.



Address Translation (Cont'd)

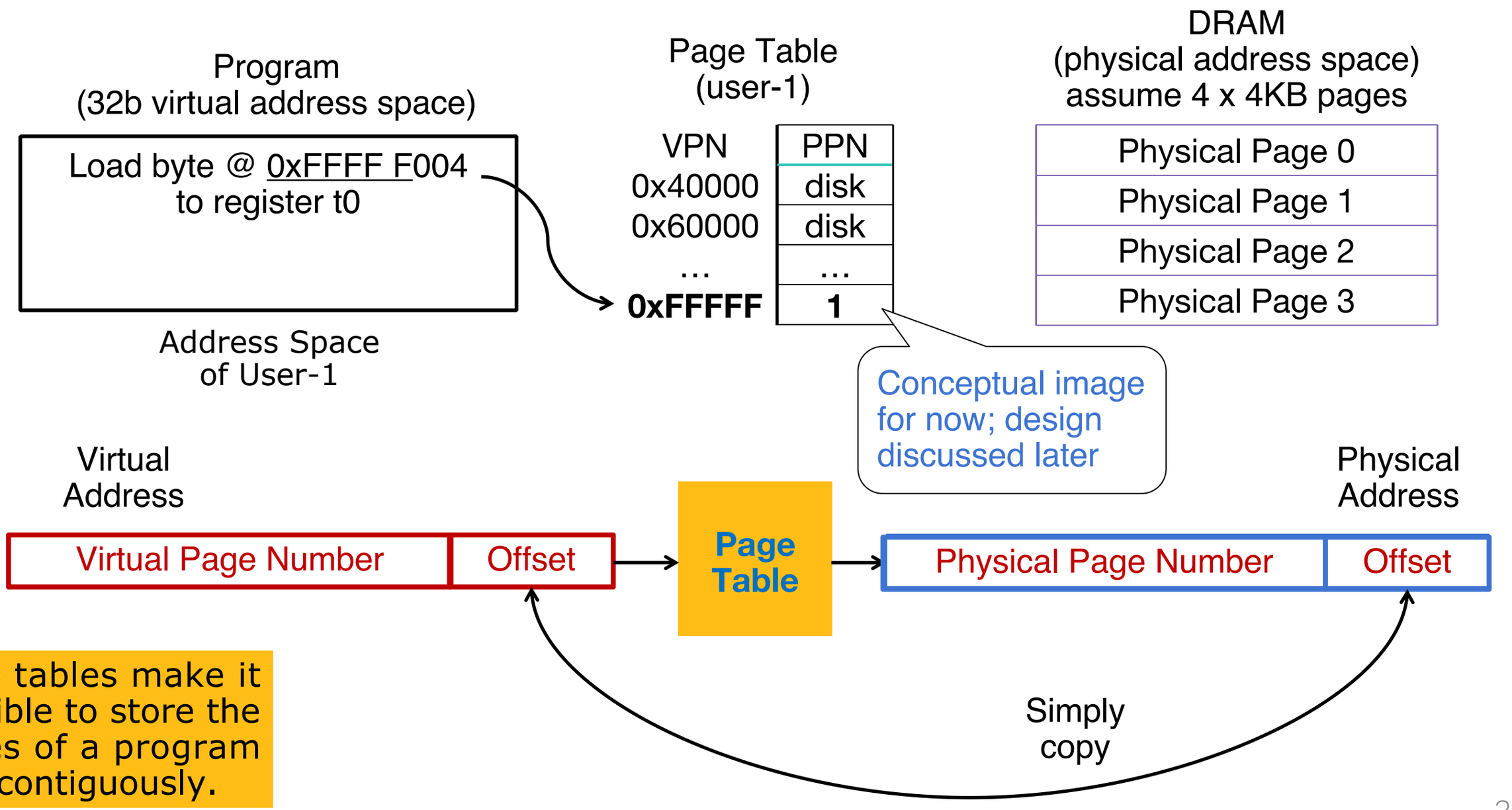
- Program executes a load specifying a virtual address (VA).
- A page table contains the physical address of the base of each page



Page tables make it possible to store the pages of a program non-contiguously.

Address Translation (Cont'd)

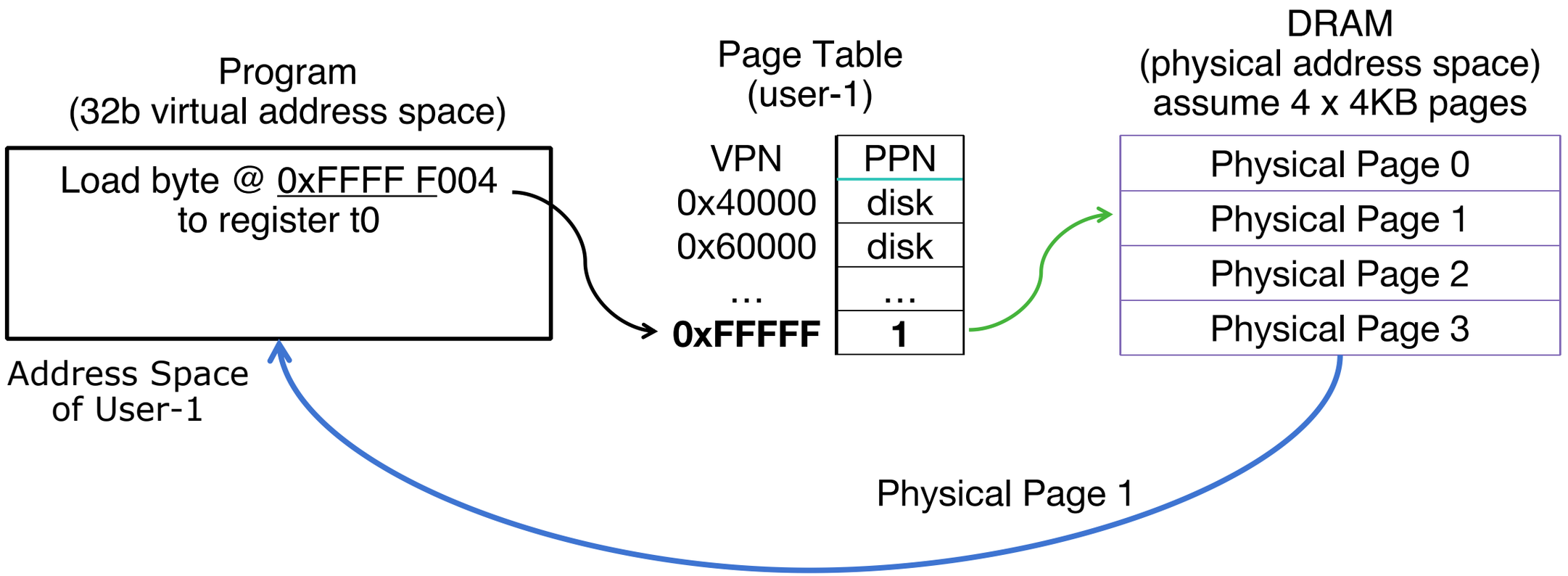
- Program executes a load specifying a virtual address (VA).
- A page table contains the physical address of the base of each page



Page tables make it possible to store the pages of a program non-contiguously.

Address Translation (Cont'd)

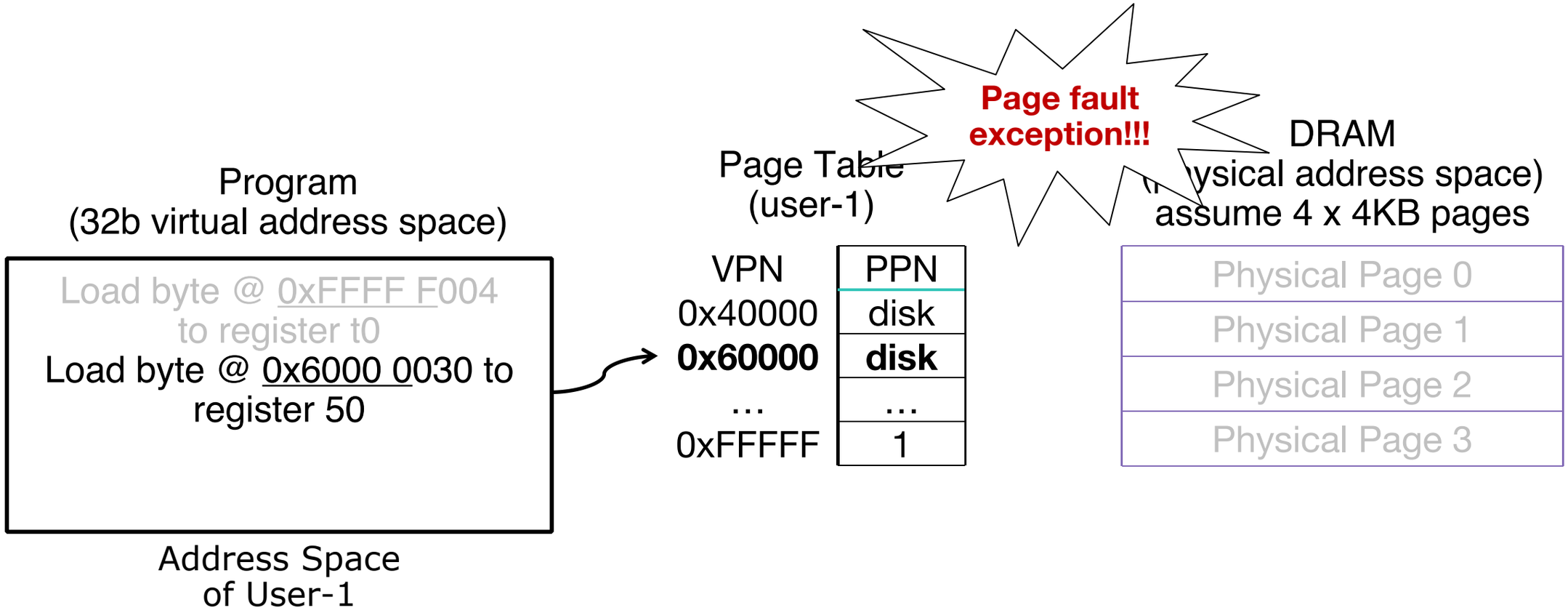
- Page table entry is valid! No page fault because page is in memory.
- Read memory at the physical address and return the data to the program.



Assume no cache

Address Translation: Page not in Memory

- Program executes a load specifying a virtual address (VA).
- OS translates VA to the physical address (PA) in memory.
  - Extract virtual page number (VPN) from VA
  - Look up physical page number (PPN) in page table
  - Construct PA: physical page number + offset (from virtual address)
- Page Fault: If physical page not in memory, then OS loads page from disk.

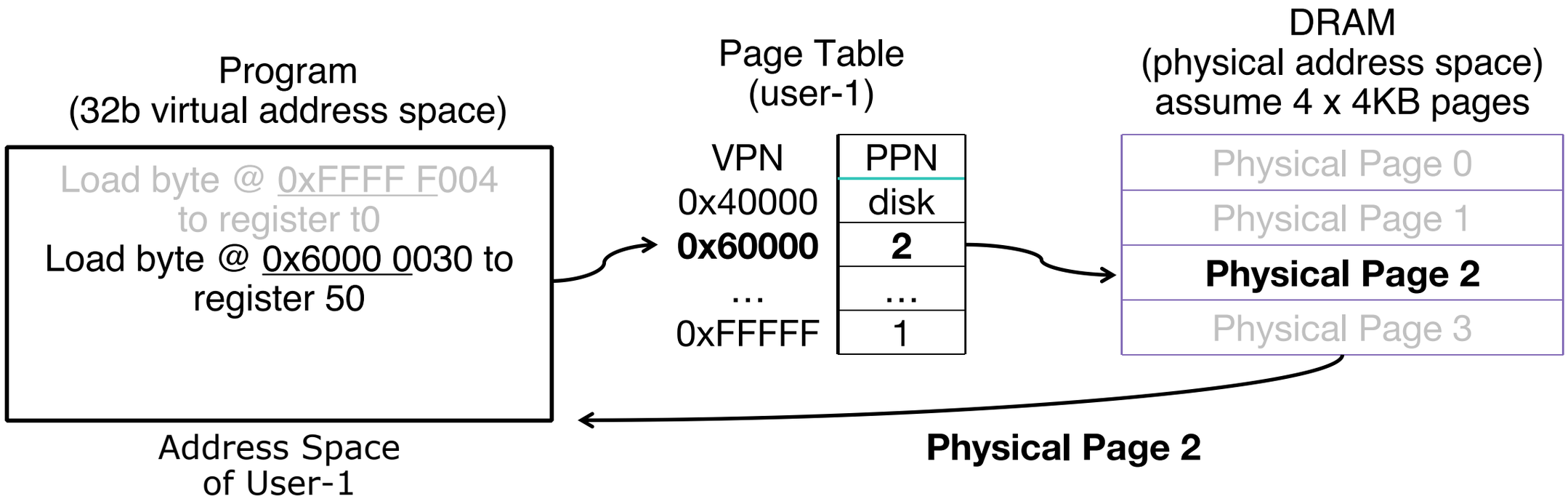


Address Translation: Page not in Memory

- Program executes a load specifying a virtual address (VA).
- OS translates VA to the physical address (PA) in memory.
  - Extract virtual page number (VPN) from VA
  - Look up physical page number (PPN) in page table
  - Construct PA: physical page number + offset (from virtual address)
- Page Fault: If physical page not in memory, then OS loads page from disk.

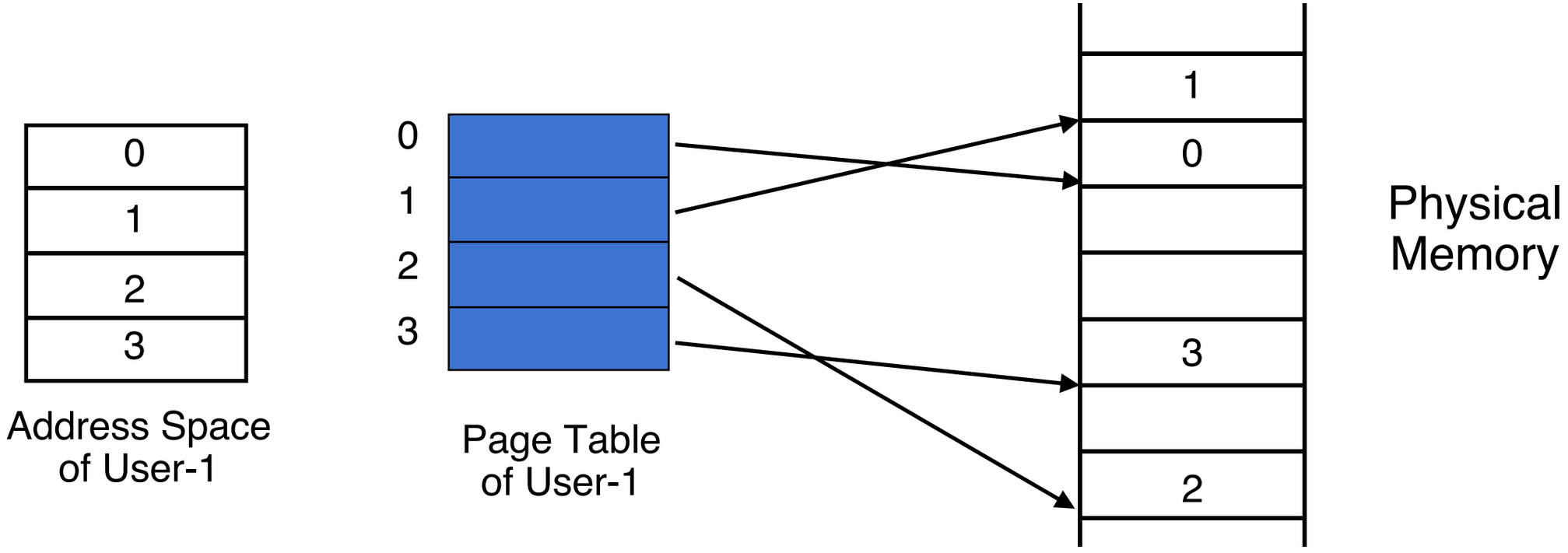
Address Translation: Page not in Memory

- Program executes a load specifying a virtual address (VA).
- OS translates VA to the physical address (PA) in memory.
  - Extract virtual page number (VPN) from VA
  - Look up physical page number (PPN) in page table
  - Construct PA: physical page number + offset (from virtual address)
- Page Fault: If physical page not in memory, then OS loads page from disk.



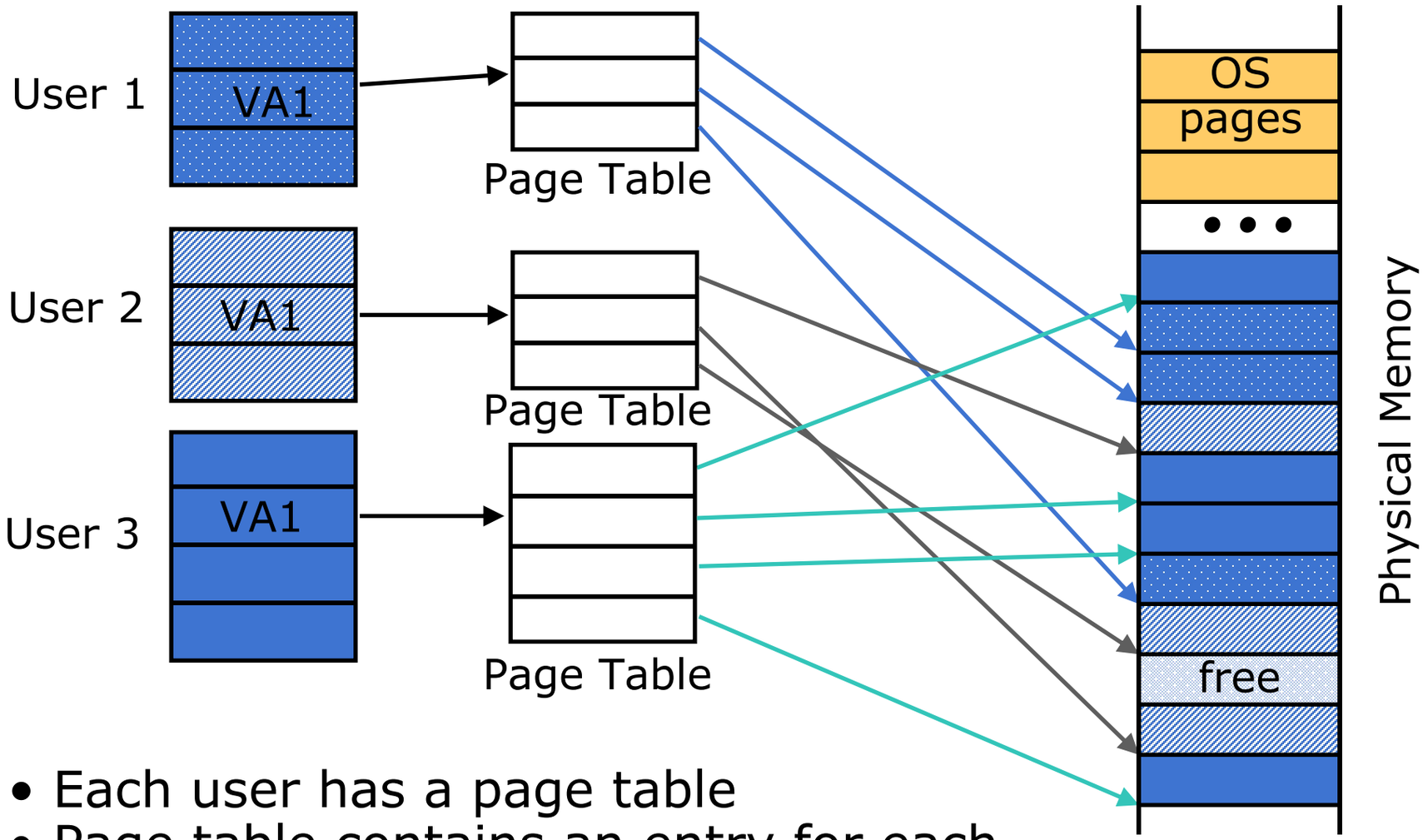
Summary: Paged Memory System

- Processor-generated address can be split into page number and offset;
- A page table contains the physical address of the base of each page;
- Page tables make it possible to store the pages of a program non-contiguously.



Summary: Paged Memory System (Cont'd)

- Processor-generated address can be split into page number and offset;
- A page table contains the physical address of the base of each page;
- Page tables make it possible to store the pages of a program non-contiguously.

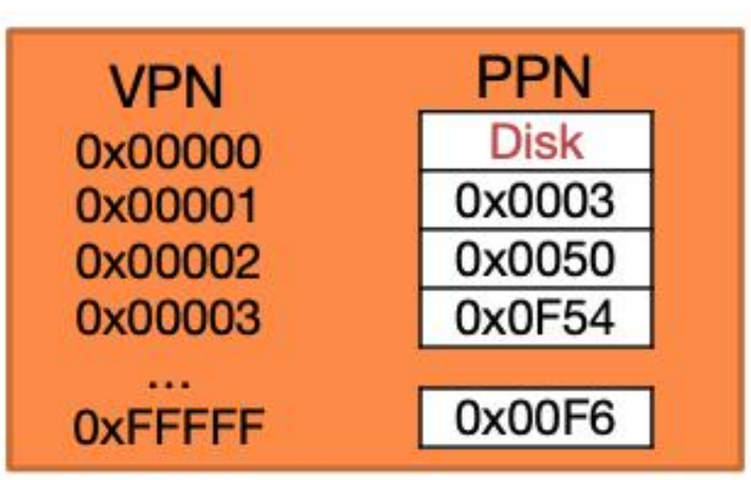


- Each user has a page table
- Page table contains an entry for each user page

Practice: Translation

- What Physical Address does this Virtual Address translate to?

`0x00003450`

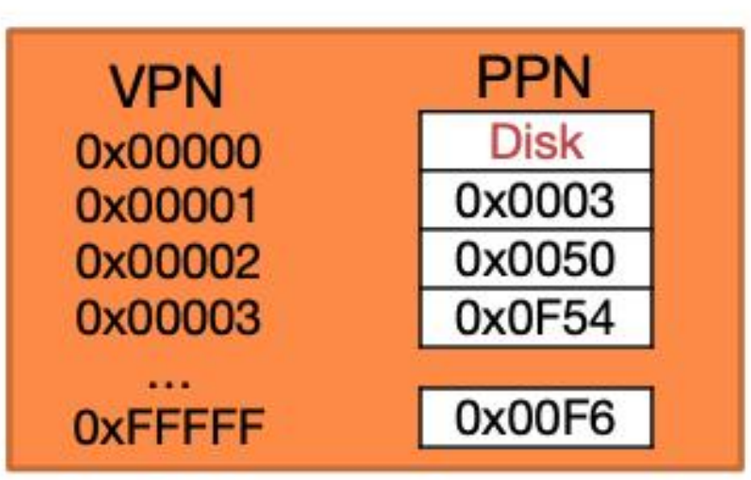


A. 0x00003450
B. 0x0000250
C. 0x00503450
D. 0x0F543450
E. 0x0F54450
F. Disk/Other

Practice: Translation

- What Physical Address does this Virtual Address translate to?

`0x00003450`



A. 0x00003450
B. 0x0000250
C. 0x00503450
D. 0x0F543450
E. 0x0F54450
F. Disk/Other

Setup of Virtual Memory

- Assume a 32-bit machine with 8GB of RAM and 16KB pages.
- How many bits would there be for each of the following?
- 1. Page offset
- 2. Virtual page number
- 3. Physical page numer

Setup of Virtual Memory

- Assume a 64-bit machine with 8GB of RAM and 16KB pages.
- How many bits would there be for each of the following?
- 1. Page offset — log2(16K) = 14 bits
- 2. Virtual page number — 64-14 = 50 bits
- 3. Physical page numer — ≧ log2(8G)-14 = 19 bits

Where Should Page Tables Reside?

- Space required by the page tables (PT) is proportional to the address space, number of users, ...
  - => Too large to keep in CPU registers
- What about cache?

Setup: Page Tables

- Assume a 32-bit machine and 16KB pages;
- Suppose each entry in the page table is 4B. How large the page table, in MB when making full use of the virtual memory? (1 entry per VPN);

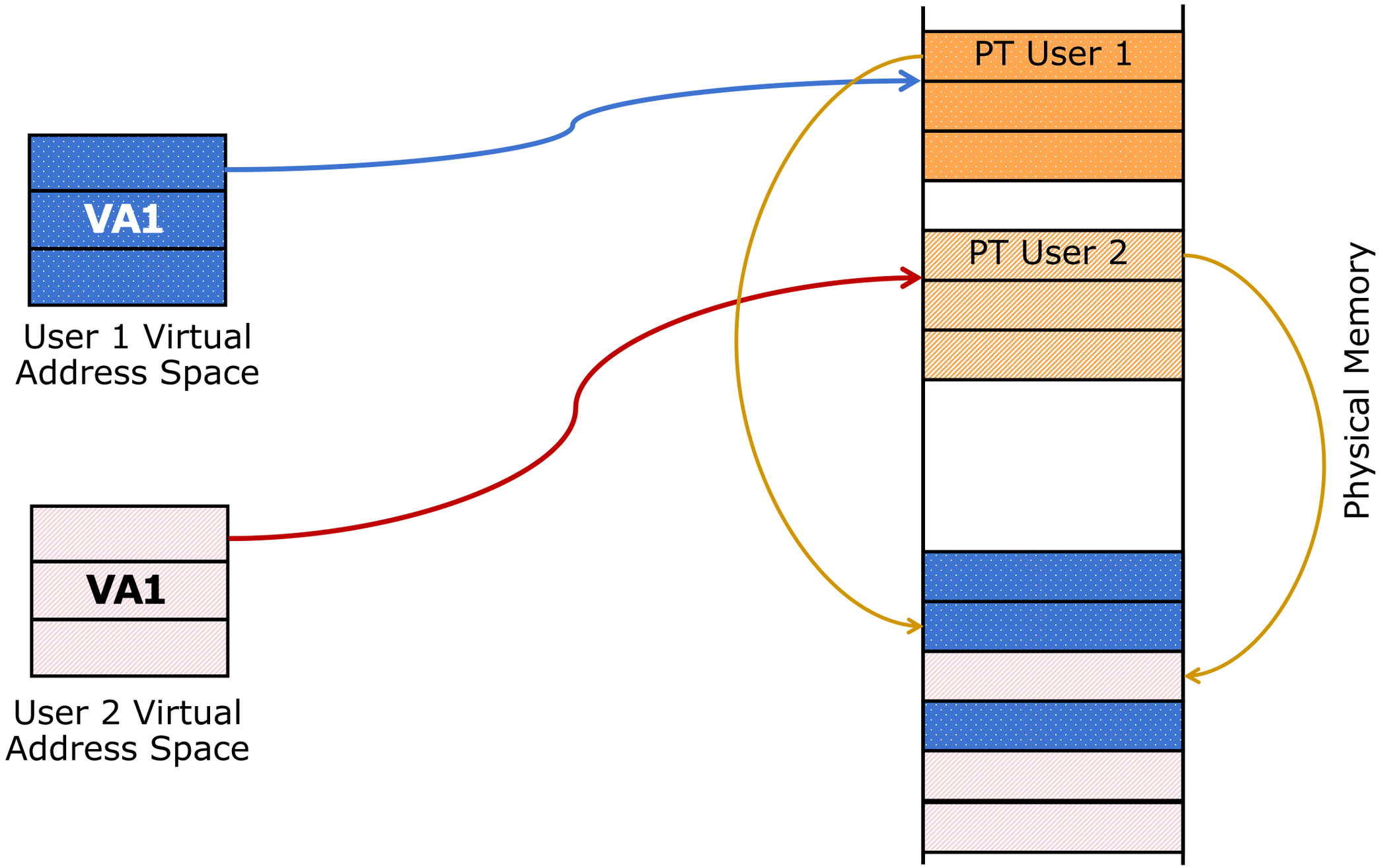
2^{18} entries x 4 B/entry = 2^{20} B = 1 MB

- There may be multiple users, so in total several tens of MB space.

Where Should Page Tables Reside?

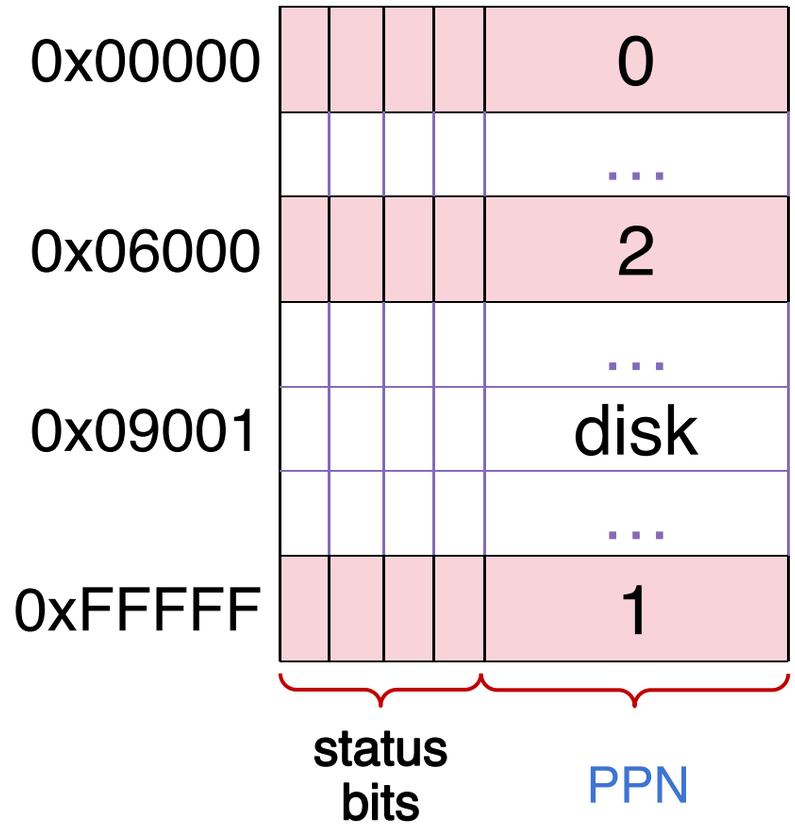
- Space required by the page tables (PT) is proportional to the address space, number of users, ...
  - => Too large to keep in CPU registers
- Idea: Keep PTs in the main memory
  - Needs one reference to retrieve the page base address and another to access the data word
    - => Cost: doubles the number of memory references!

Page Tables in Physical Memory



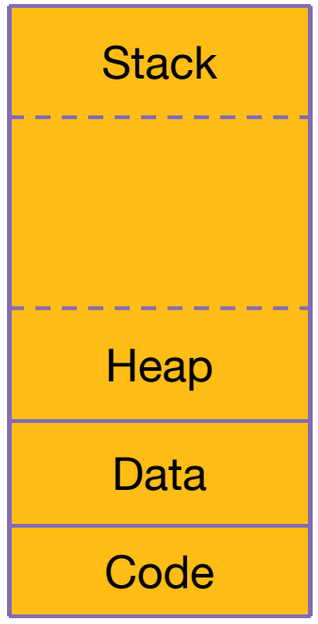
Page Tables-More Details

- One page table per process/user.
  - One entry per virtual page number.
  - Entry has physical page number (if in memory) as well as **status/protection** bits (read, write, exec, disk/physical memory, etc.).
- A page table is **NOT** a cache!!!
  - A page table is a lookup table!
  - It does NOT have data.
  - All VPNs have an entry in the page table.

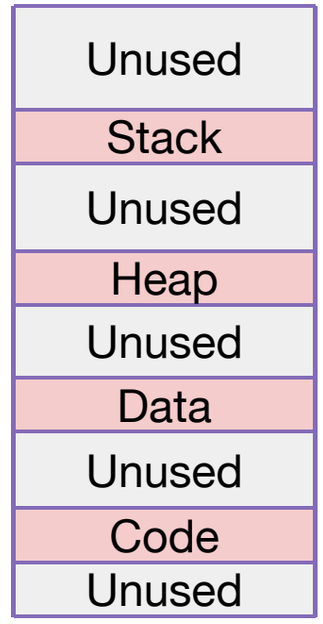


Demand Paging

- Modern Virtual Memory Systems use address translation to provide the illusion of a large, private, and uniform storage.
- Price: Address translation (via page tables) on each memory reference.
- Loading a ton of pages for every new program is wasteful.
- What if the program is just “hello world”? Most pages are never used.
- Solution: Demand paging.
  - Only load a page into memory if the user requests it.
  - When a program starts, the page table says “Disk” for every entry.



Our VM abstraction allows a program to use the full virtual address space...



...but some programs use only a tiny amount of memory.

Steps of Address Translation, Revisited

- Program wants to access memory at a given virtual address (VA);
- OS translates VA to the physical address (PA) in memory;
  - Extract virtual page number (VPN) from VA;
  - Look up physical page number (PPN) in page table;
  - Construct PA: physical page number + offset (offset same as in virtual address);
- Page Fault: If physical page not in memory, then OS loads page from disk;
- Read memory at the physical address and return the data to the program.

Memory access 1 called **page table walk (PTW)**

Demand paging means possible disk access here

Memory access 2

Status Bits

- On each memory access, first check if page table entry is “valid”.
  - Valid/on → In main memory, read/write data as directed by process.
  - Not Valid/off → On disk
    - Trigger page fault exception, OS intervenes to allocate the page into DRAM (trap handler);
    - If out of memory, select a page to replace in DRAM
    - Store outgoing page to disk, and page table entry that maps that VPN->PPN is marked as invalid/DPN
    - Read requested page from disk into DRAM and update with a valid PPN
    - Finally, read/write data as directed by process.

Dirty Bits

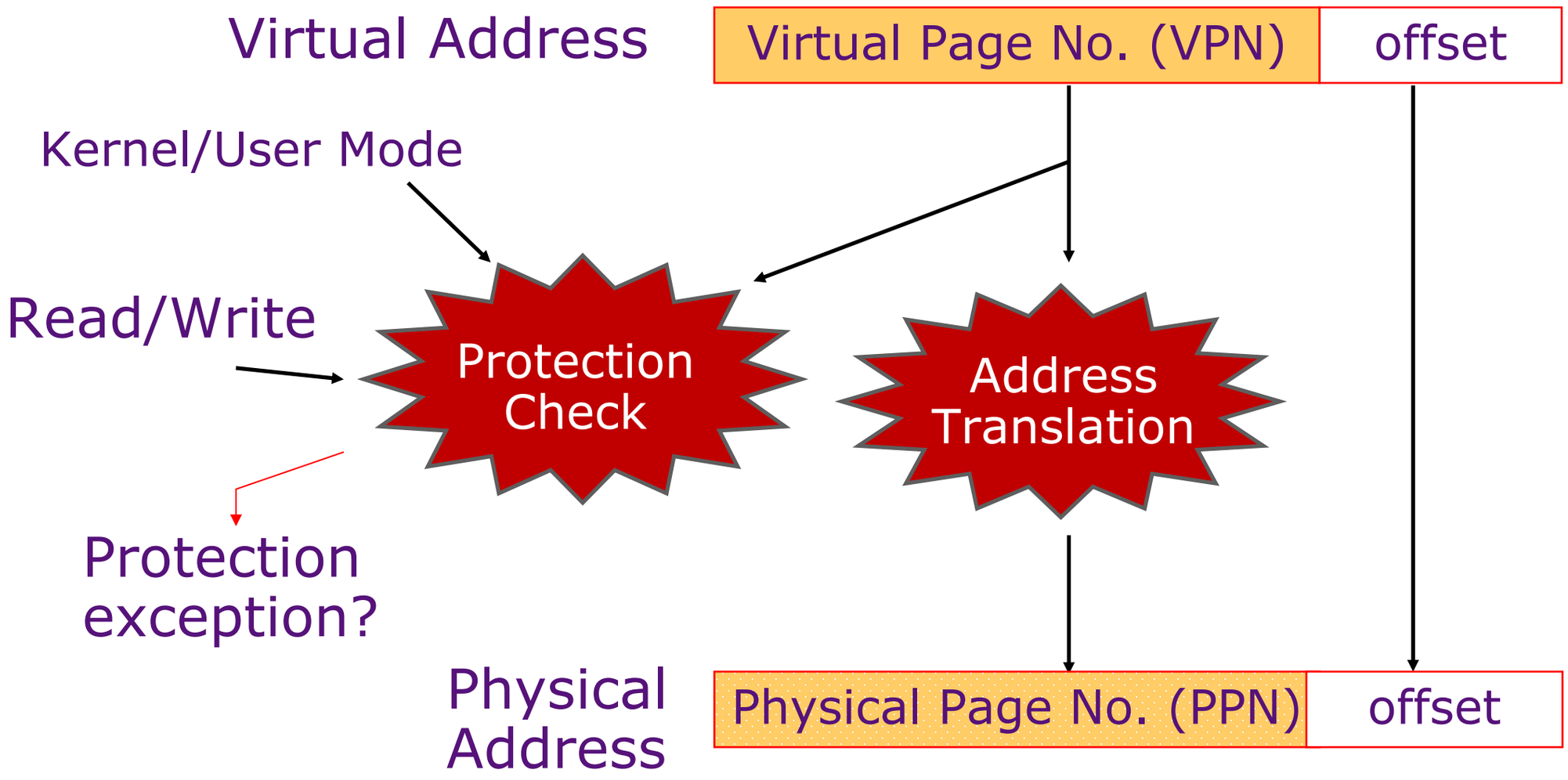
- Main memory acts like a “cache” for secondary memory/disk.
  - Should writes always go directly to disk (write-through), or
  - Should writes only go to disk when page is evicted (write-back)?

- All virtual memory systems use write-back.
  - Disk accesses take way too long!
- Recall that write-back policy requires a dirty bit to keep track of pages that have an unsynced write.
  - When a page gets replaced:
  - Dirty bit on: Write outgoing page back to disk.
  - Dirty bit off: No disk write.

Program Isolation: Goals and Solutions

- OS goals for protection:
  - Isolate memory between processes.
  - Each process gets dedicated “private” memory.
    - Errors in one program won’t corrupt memory of other programs.
    - Prevent user programs from messing with OS’s memory.
    - Allow memory sharing where safe.
- Protection with Page Tables:
  - One page table per process, provides isolation
  - Memory sharing + write/read protection bits on page table entries
  - Page table managed by OS

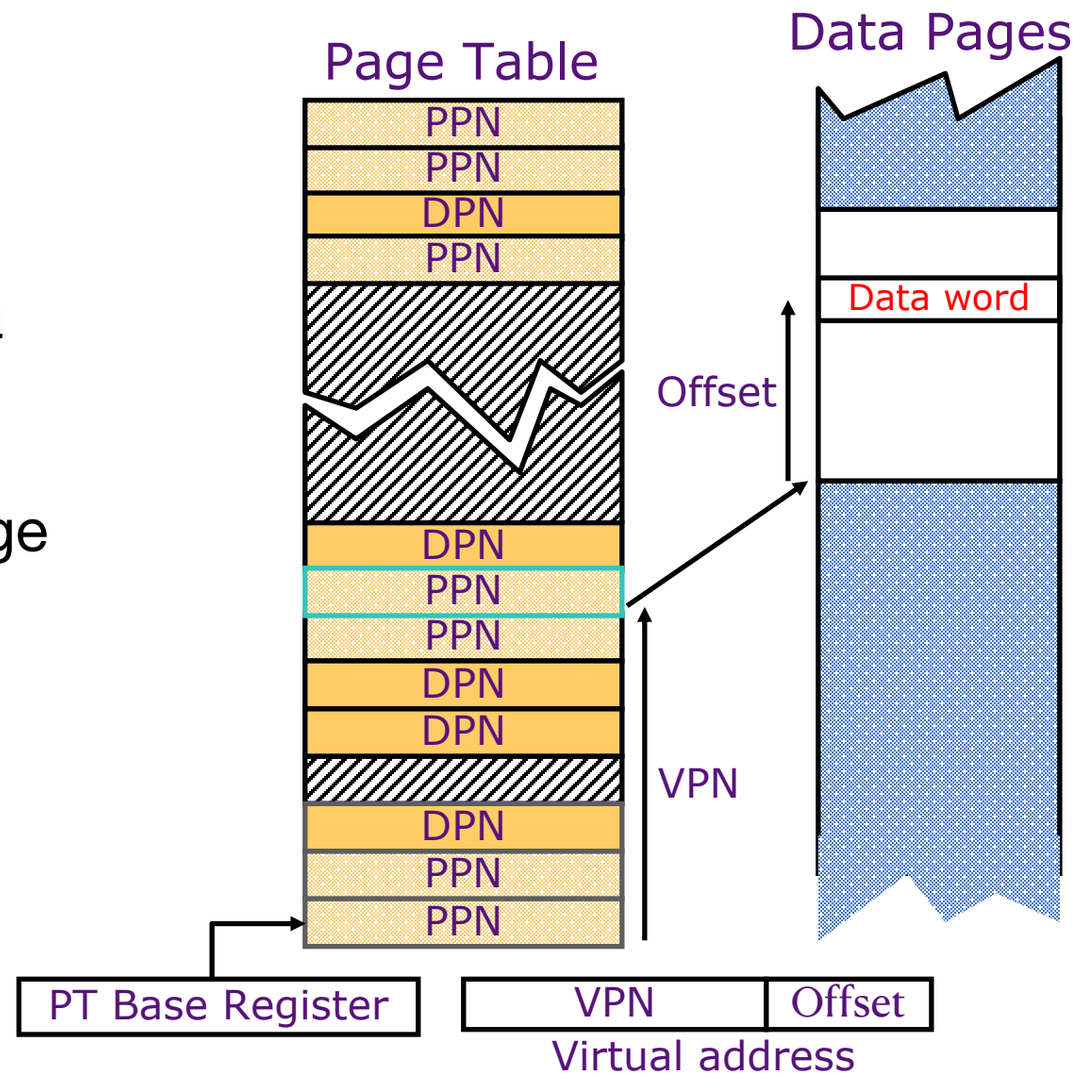
Protection



- Every instruction and data access needs address translation and protection checks

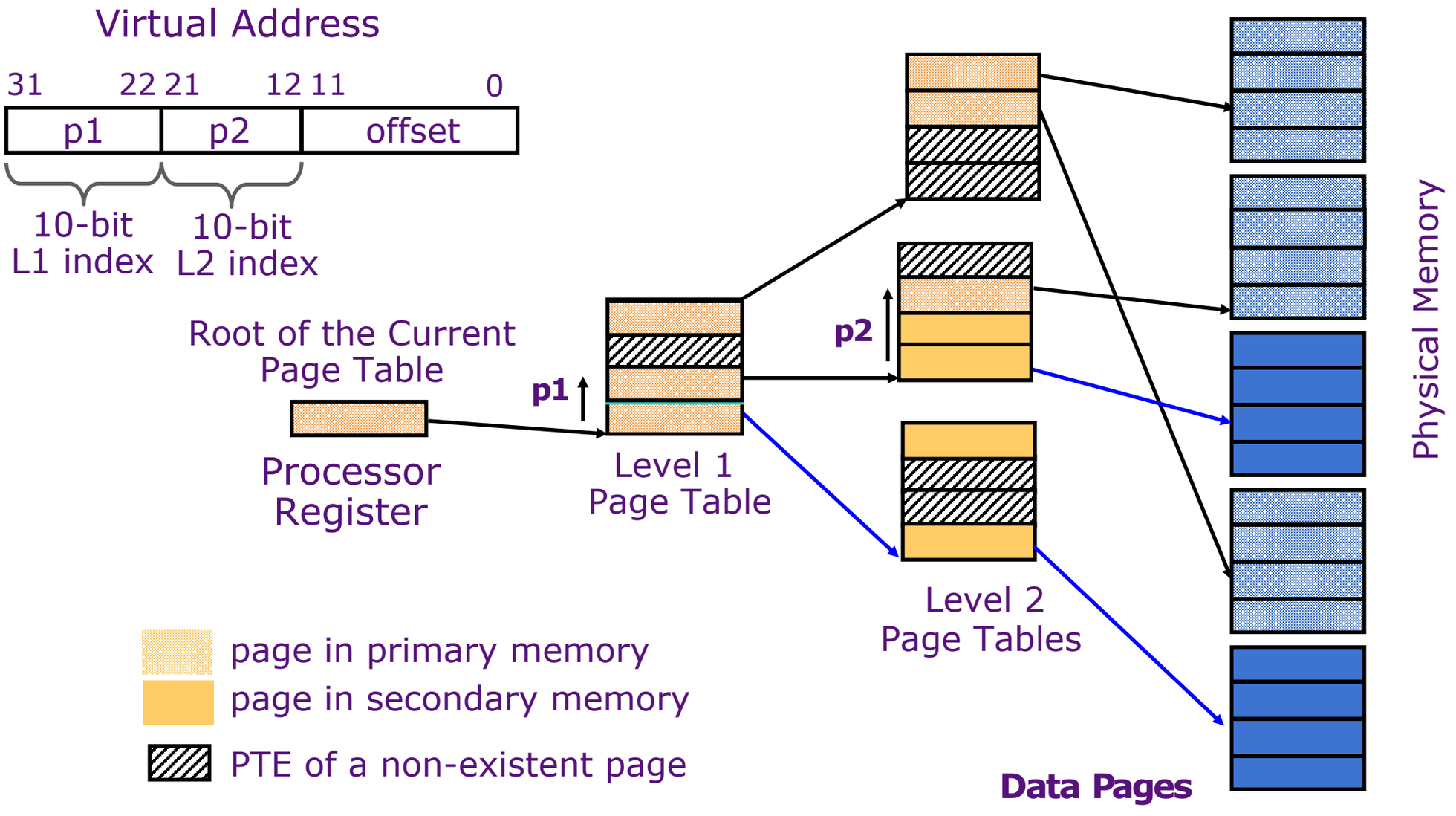
Linear (Simple) Page Table Summary

- Page Table Entry (PTE) contains:
  - 1 bit to indicate if page exists and either PPN or DPN:
    - PPN (physical page number) for a memory-resident page
    - DPN (disk page number) for a page on the disk
  - Status bits for protection and usage (read, write, exec, dirty, etc.)
- OS sets the Page Table Base Register whenever active user process changes



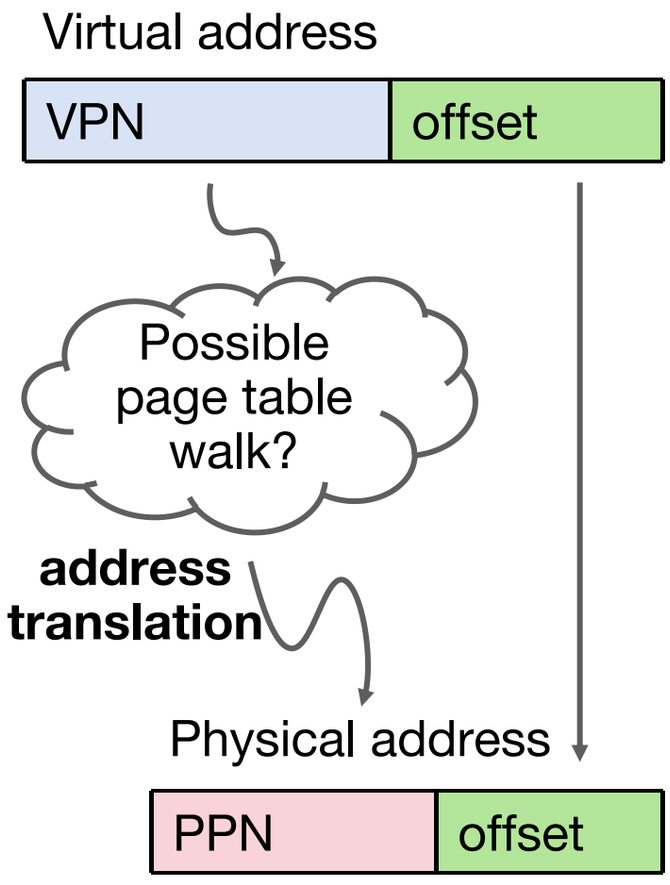
Hierarchical (Multi-level) Page Table

- Assume 32-bit machine and 4-KB pages => 2^{20} entries * 4 B/entry = 4 MB
- If 256 processes, PT = 256 * 4 MB = 1 GB
- Exploit sparsity of virtual address space use



Translation Lookaside Buffer

- Address Translation: Avoid page table walks (PTWs)
- Good Virtual Memory design should be fast (~1 clock cycle) and space efficient.
  - However, every instruction/data access needs address translation.
- A PTW is the process of accessing the page table in memory.
  - If page tables are in memory, then we must perform a PTW per instruction/data access!
- Solution: Cache some address translations in the Translation Lookaside Buffer (TLB) in a seperate “cache”.



TLB Address Lookup

- Recall that a page table stores VPN-PPN mappings in memory.
- The translation lookaside buffer (TLB) is a cache of VPN-PPN mappings.
- The TLB is much closer to CPU and caches.

VPN	PPN
...	...
0x00004	0x60C25E6
0x00005	0x71DB139
0x00006	0xEC70DB7
0x00007	0xAB12BF4
0x00008	0x2158D55
0x00009	0x45099CD
...	...

Process 1
Page Table

VPN	PPN
0x00004	0x60C25E6
0x00005	0x71DB139
0x00009	0x45099CD

The TLB

The TLB is a cache—not for program data, but for page table entries. The TLB speeds up address translation.

TLB Details

- Address translation is very expensive!
- In a two-level page table, each reference becomes several memory accesses
- Solution: Cache some translations in TLB
- TLB hit => Single-Cycle Translation
- TLB miss => Page-Table Walk to refill

VPN	PPN
...	...
0x00004	0x60C25E6
0x00005	0x71DB139
0x00006	0xEC70DB7
0x00007	0xAB12BF4
0x00008	0x2158D55
0x00009	0x45099CD
...	...

Process 1
Page Table

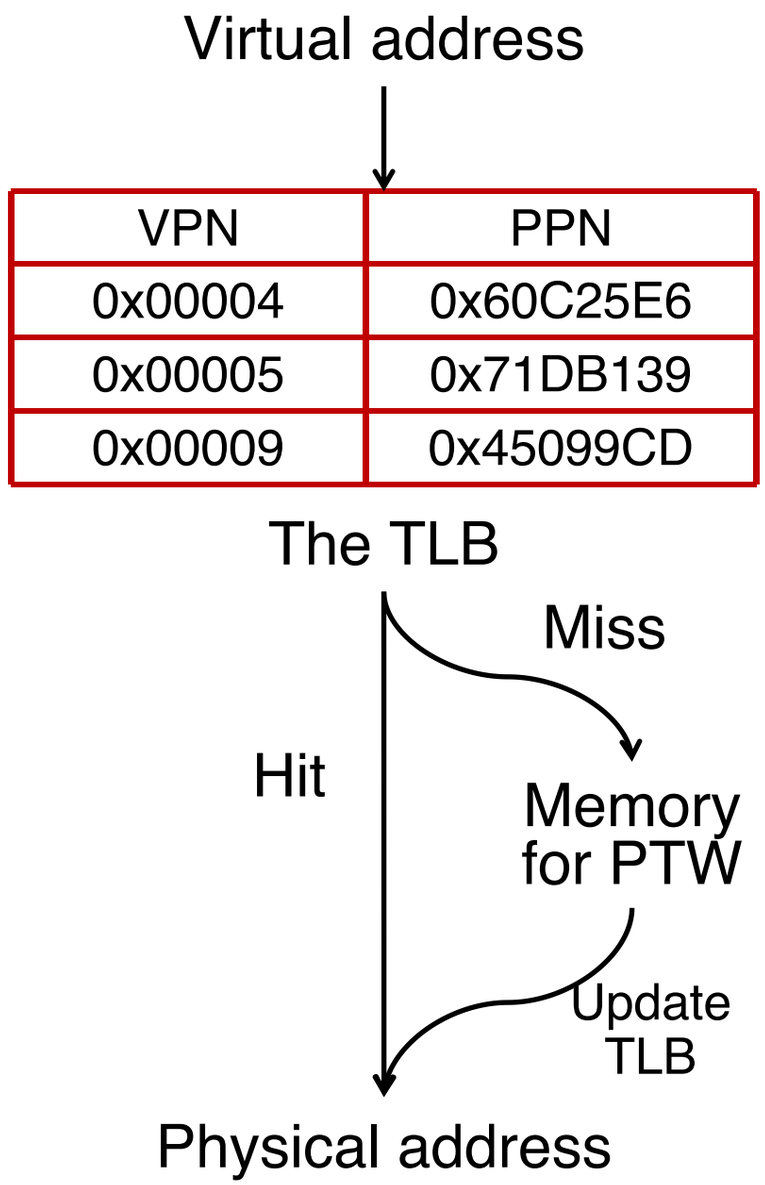
VPN	PPN
0x00004	0x60C25E6
0x00005	0x71DB139
0x00009	0x45099CD

The TLB

The TLB is a cache—not for program data, but for page table entries. The TLB speeds up address translation.

TLB Details (Cont'd)

- Technical details:
  - Typically 32-128 entries
  - Fully associative (or 2-way set associative)
  - Replacement policy: FIFO, random
    - Each entry maps a large page, hence less spatial locality across pages => more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs



TLB Details (Cont'd)

- Technical details:
  - **TLB Reach** tells us how many virtual addresses can get immediately translated by the TLB.
    - TLB Reach = # TLB entries · Page size
    - If the TLB "hits", we don't need a page table walk (avoid additional memory access for address translation).
- There is just one TLB per core, but recall page tables are per process.
- When the OS performs a context switch to run a different program:
  - Easier: OS flushes all/part of the TLB by invalidating its entries.
  - Harder: track which process corresponds to which entries

This VPN translation is for Process 1. If we switch to Process 2, entry isn't valid.

VPN	PPN
0x00004	0x60C25E6
0x00005	0x71DB139
0x00009	0x45099CD

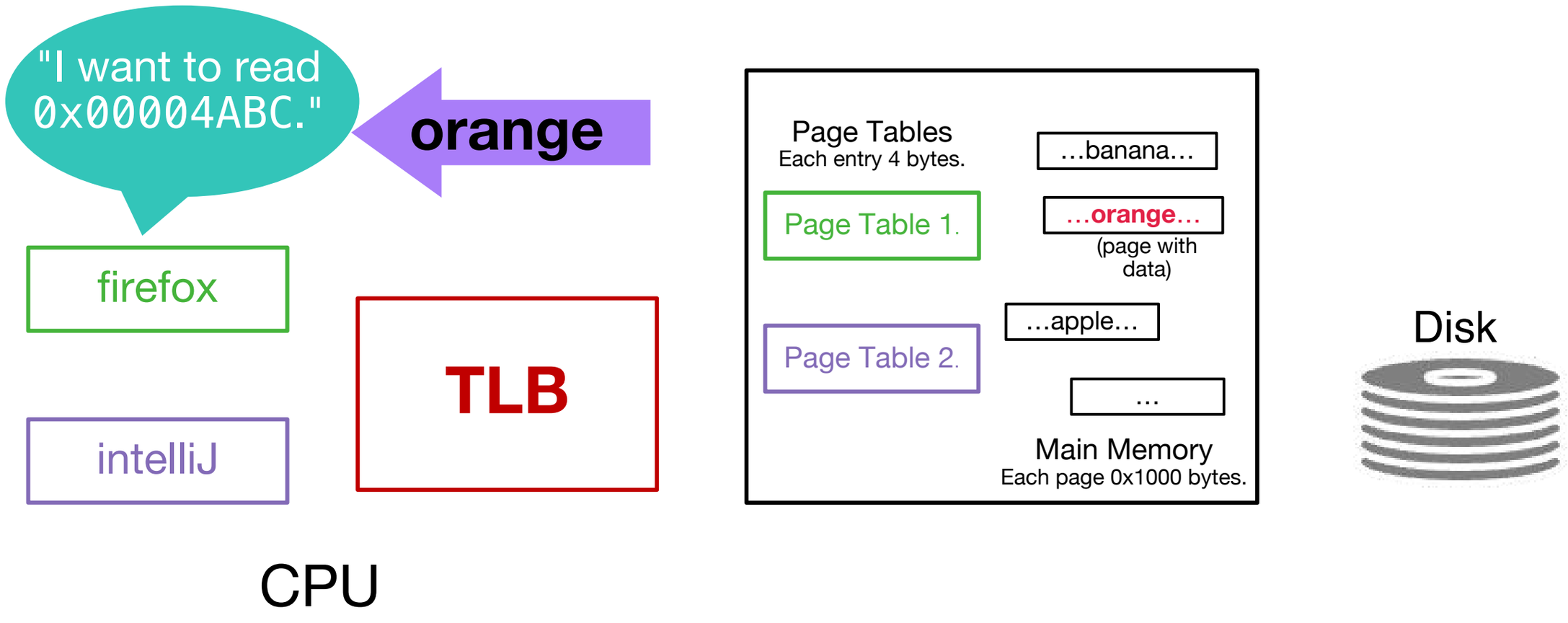
The TLB

I support instant lookup for up to $3 \times 2^{12} = 12288$ virtual addresses.

(assuming 4KiB pages, i.e., 12-bit page offset)

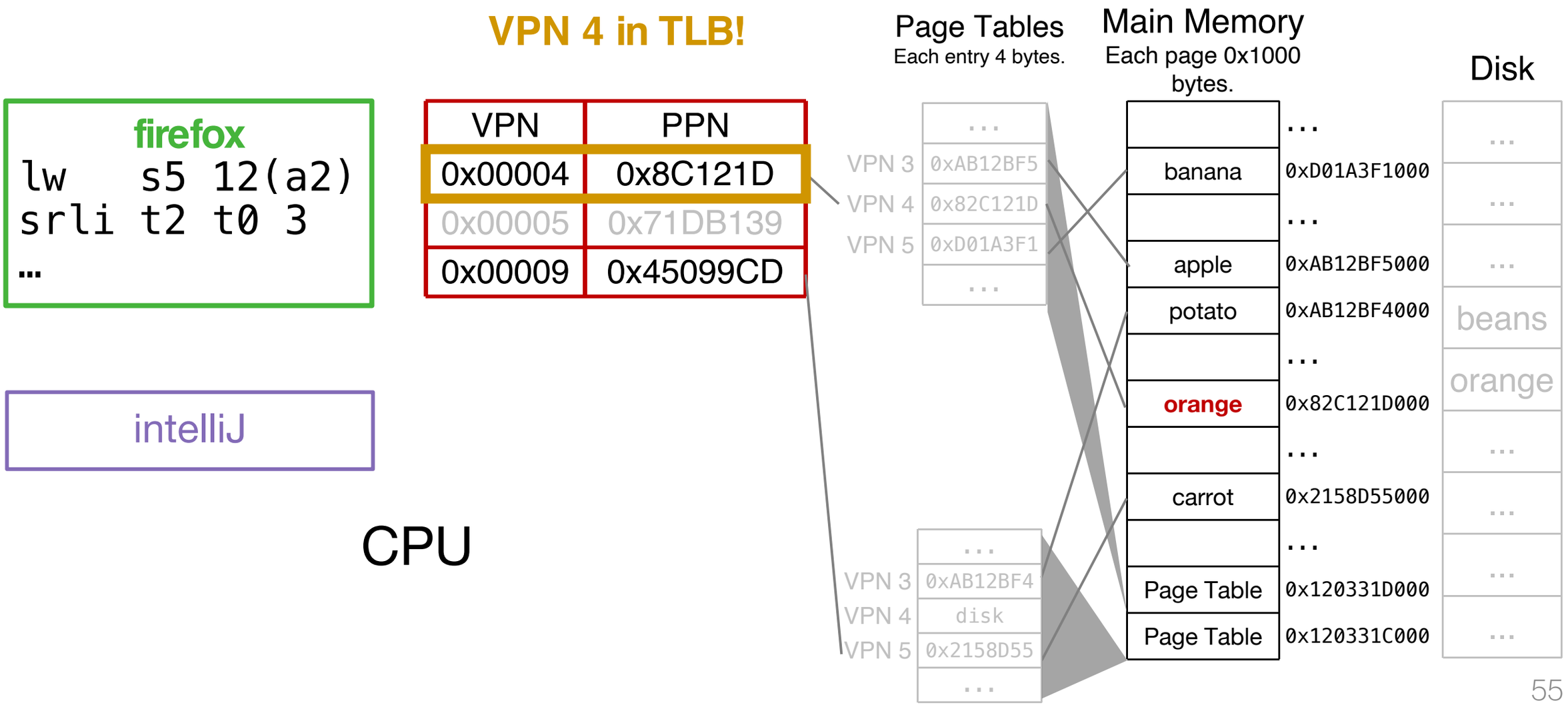
Address Translation Example

- Different cases of address translation impact the end-to-end latency of a single memory access.
  - Memory access = address translation + data access



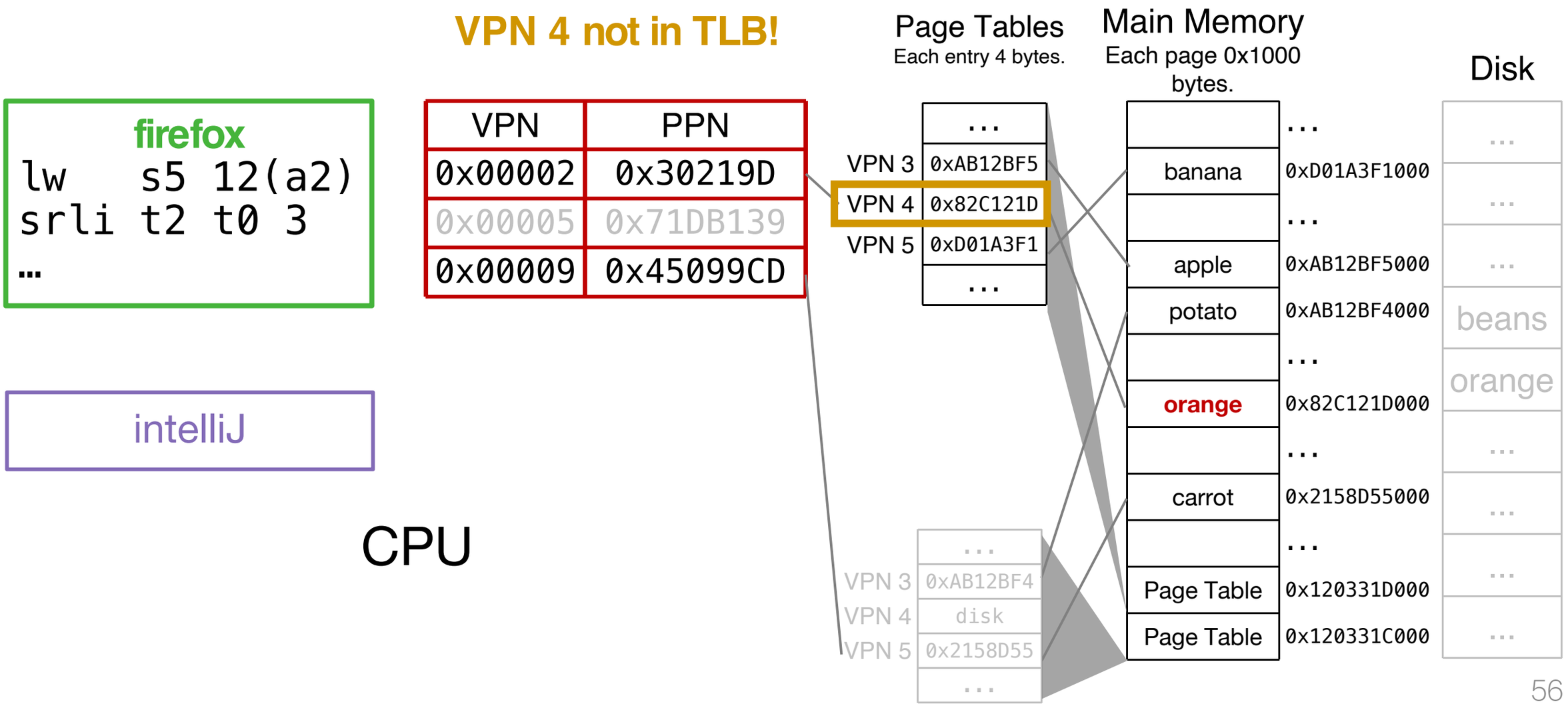
Best Case (~ 1 clock cycle)

- Different cases of address translation impact the end-to-end latency of a single memory access.
  - Memory access = address translation + data access



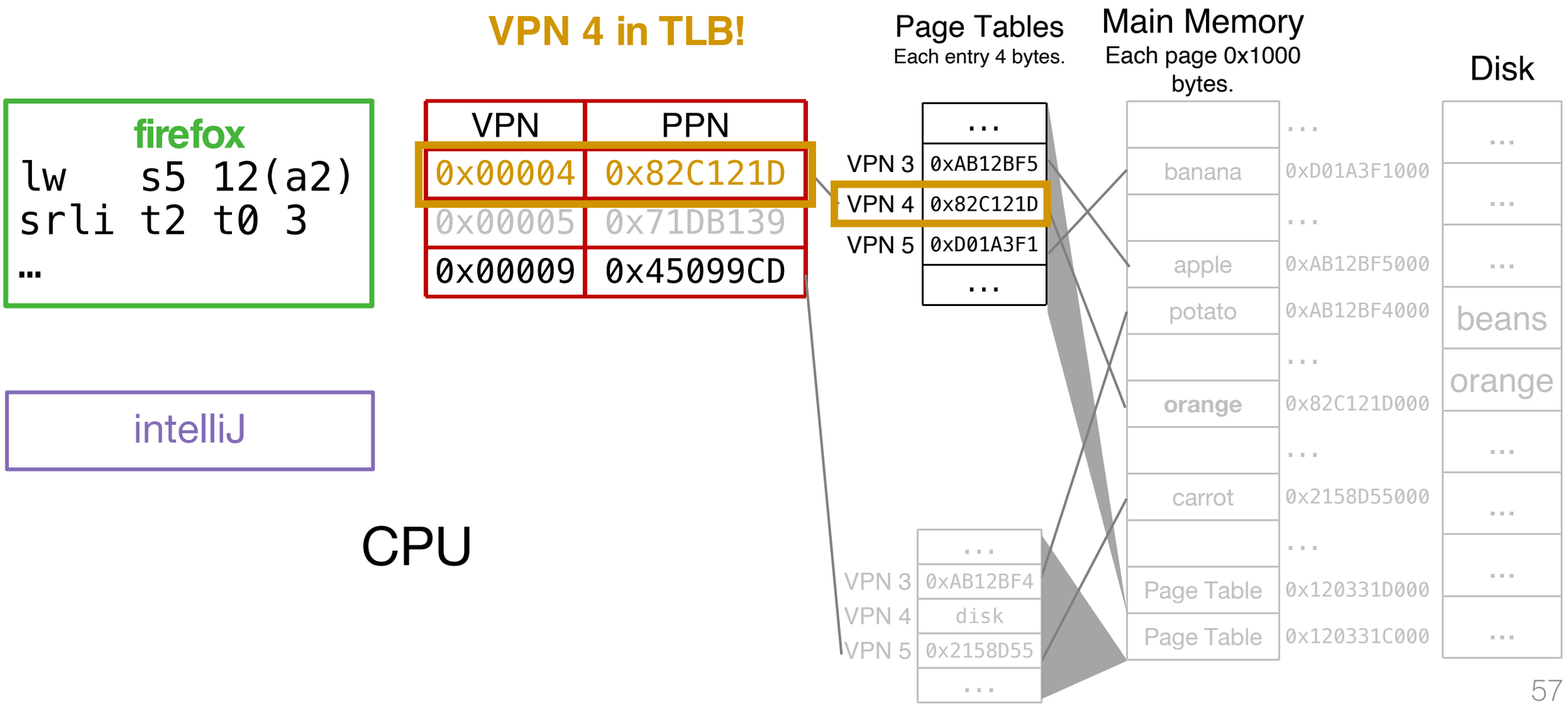
Worse Case (TLB miss + PTW)

- Different cases of address translation impact the end-to-end latency of a single memory access.
  - Page table walk (~100 cycles): Go to main memory, read page table.



Worse Case (TLB miss + PTW, cont'd)

- Different cases of address translation impact the end-to-end latency of a single memory access.
  - Page table walk (~100 cycles): Go to main memory, read page table.
  - **Update TLB** so we have this VPN/PPN mapping handy for next time.



Worst Case (Page Fault)

- **Page fault.** Go to disk to load page (~1000s cycles)

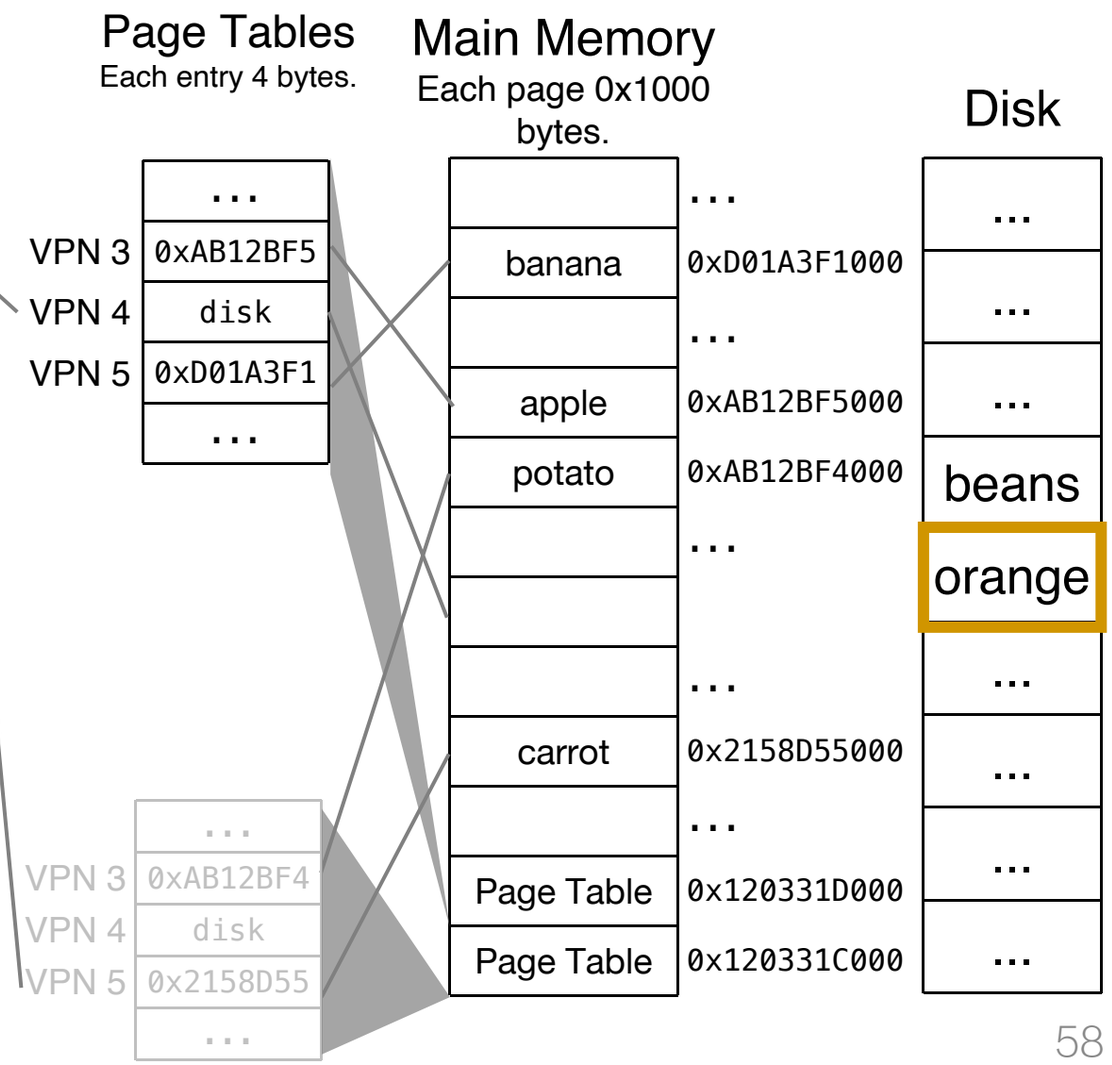
VPN 4 not in TLB!

firefox
```
lw   s5 12(a2)
srli t2 t0 3
…
```

intelliJ

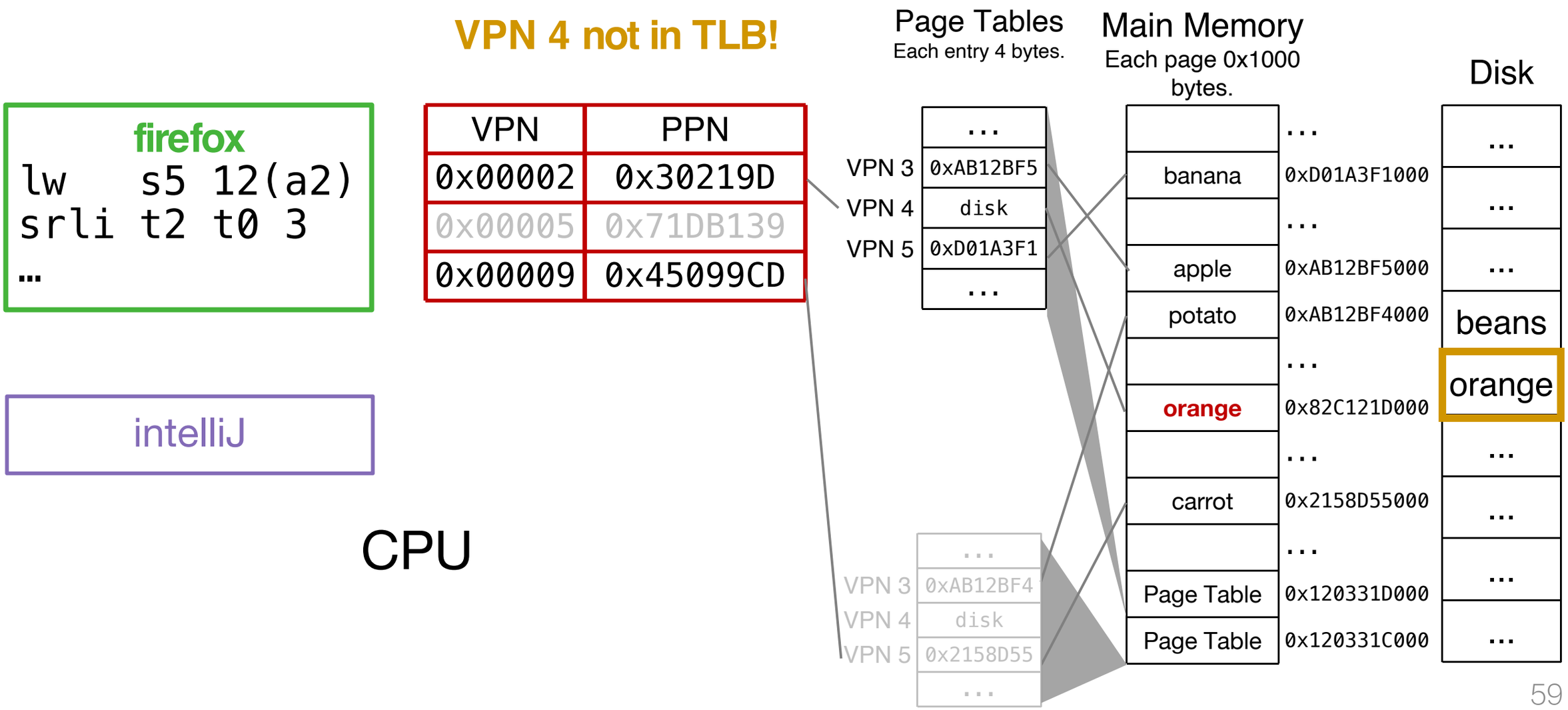
CPU

VPN	PPN
0x00002	0x30219D
0x00005	0x71DB139
0x00009	0x45099CD



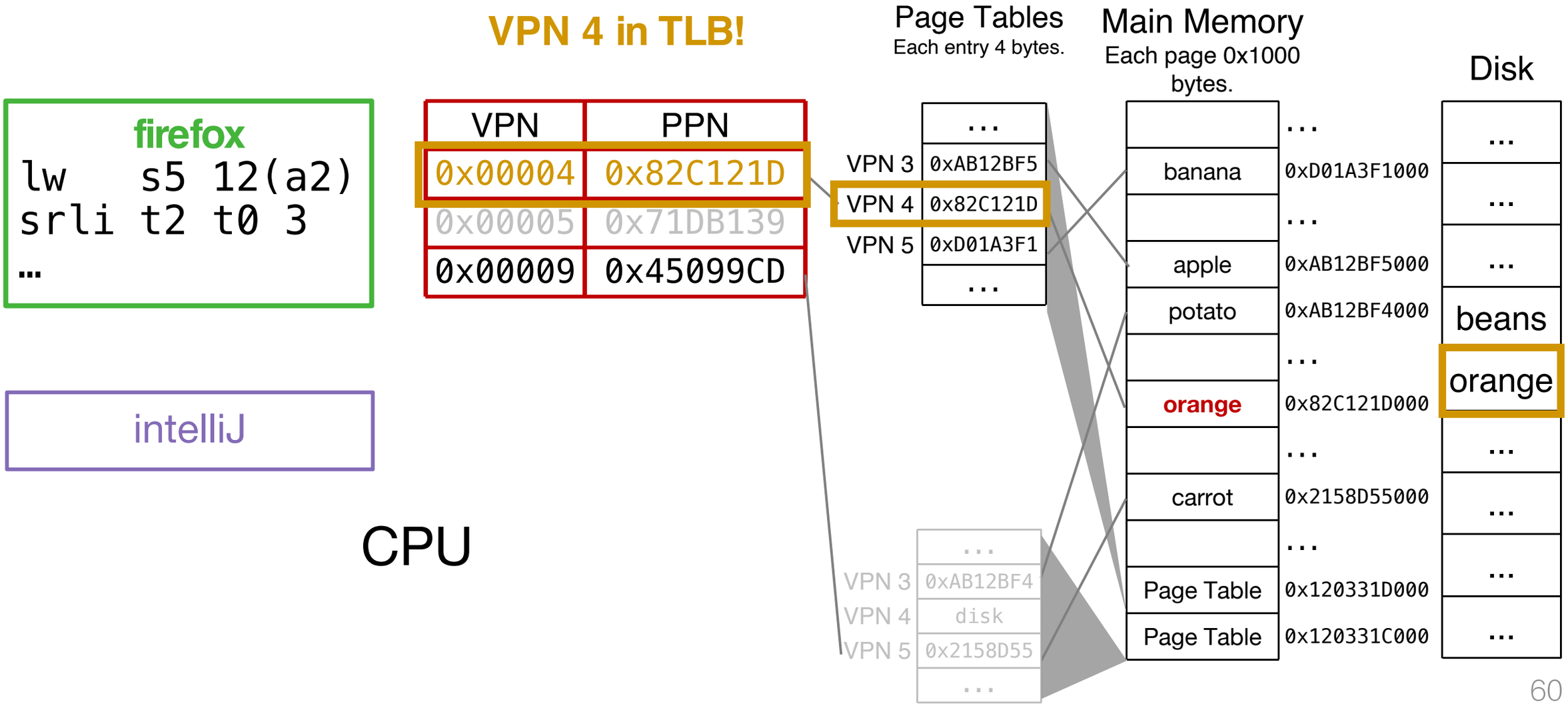
Worst Case (Page Fault, Cont'd)

- **Page fault.** Go to disk to load page (~1000s cycles)



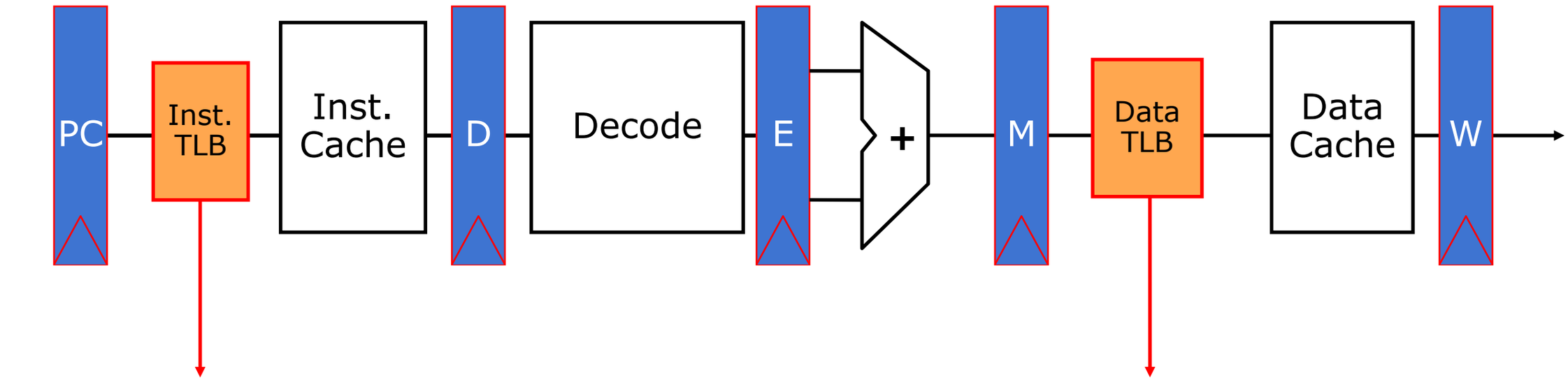
Worst Case (Page Fault, Cont'd)

- **Page fault.** Go to disk to load page (~1000s cycles)
- **Update page table** with PPN of the newly-loaded page.
- **Update TLB** so we have this VPN/PPN mapping handy for next time.

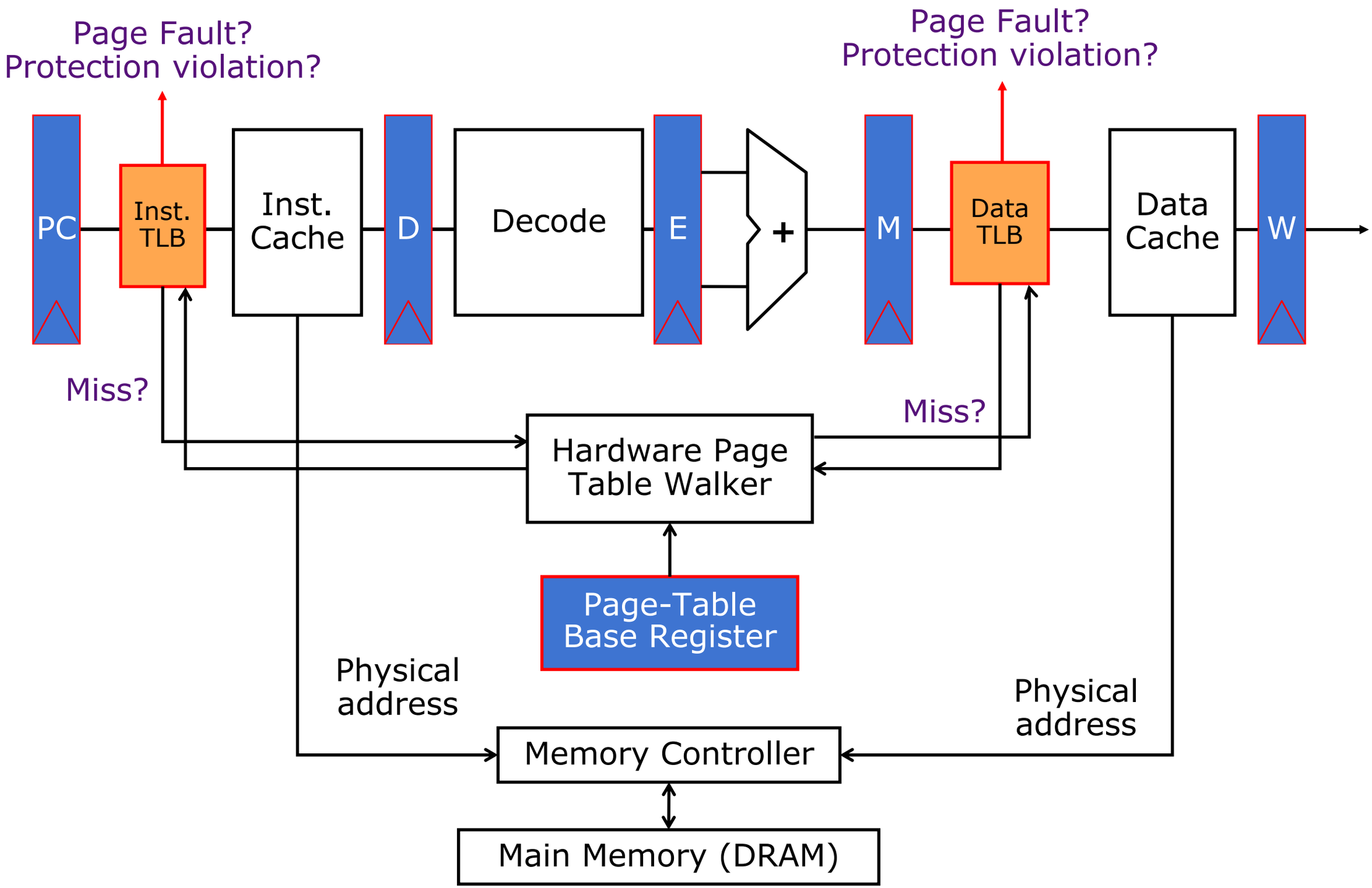


WM-related Events in Pipeline

- Handling a TLB miss needs a hardware or software mechanism to refill TLB (usually done in hardware now)
- Handling a page fault (e.g., page is on disk) needs a precise trap so software handler can easily resume after retrieving page
- Handling protection violation may abort process

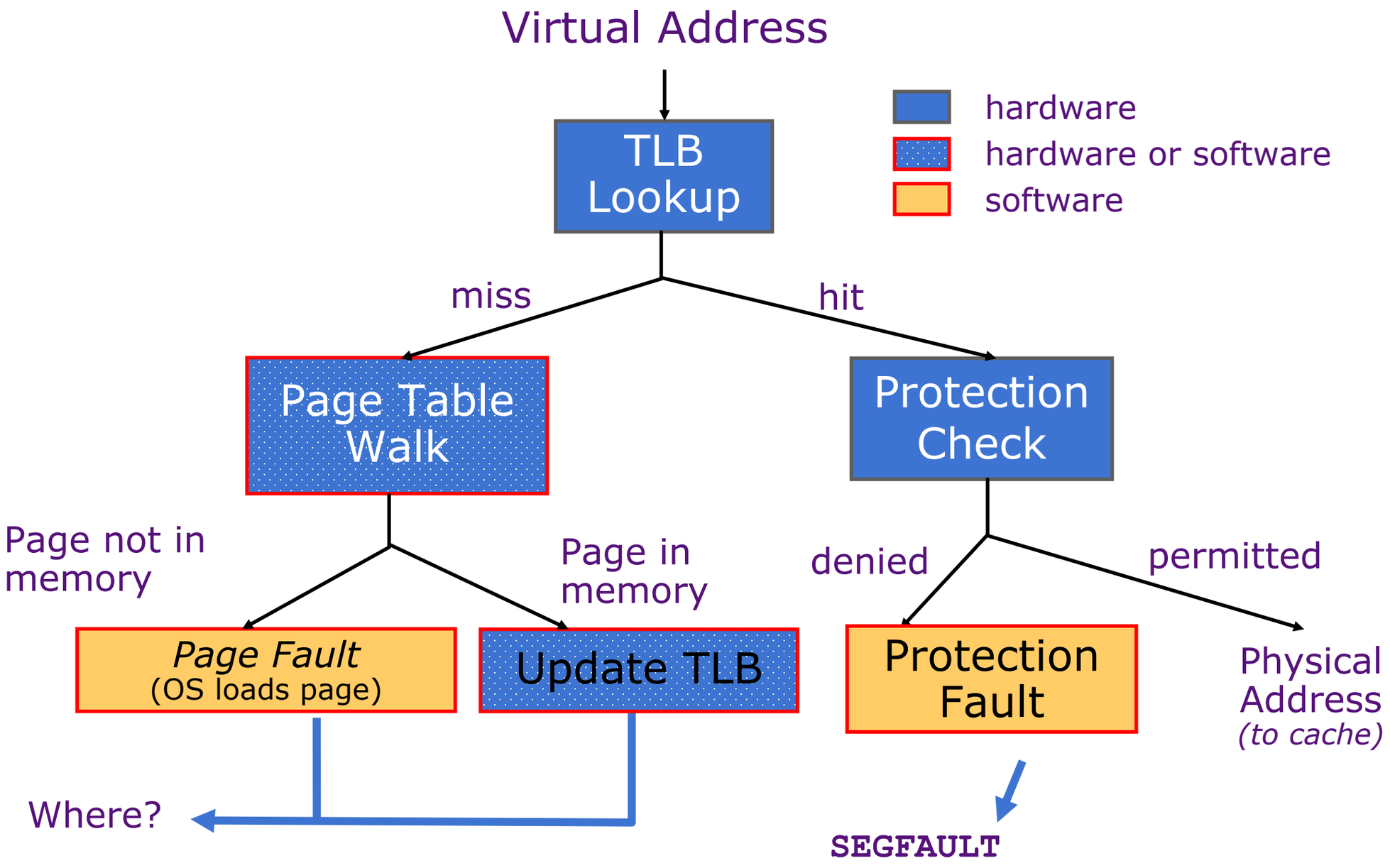


Page-based Virtual-Memory Machine with Hardware PTW



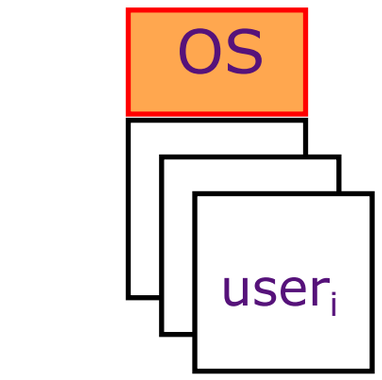
- Assume base addresses of page tables held in untranslated physical memory

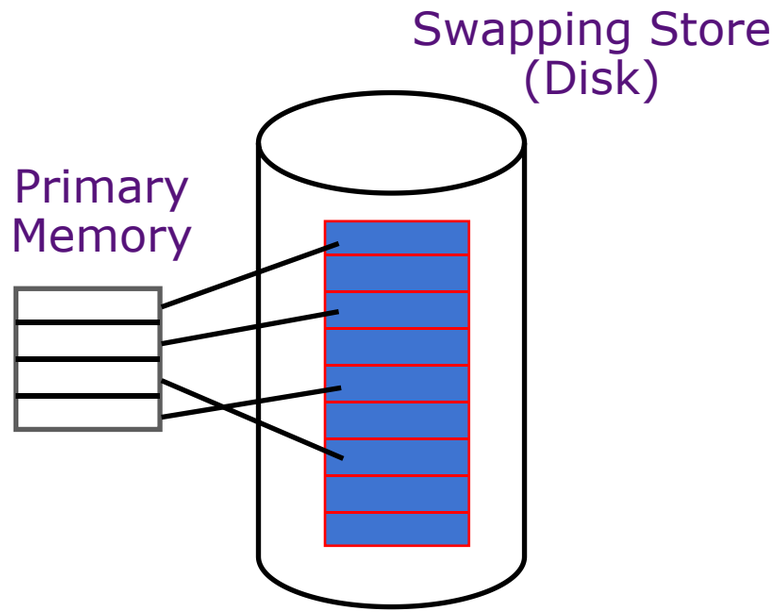
Address Translation: Wrap it Up

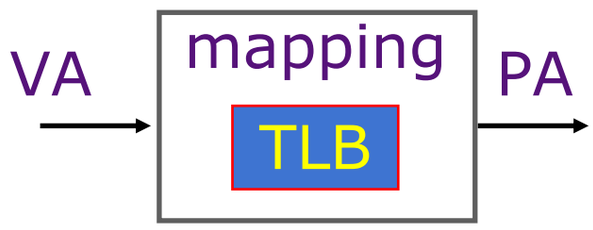


Modern Virtual Memory Systems

- Protection & Privacy
  - Several users, each with their private address space and one or more shared address spaces
  - Page table = name space
- Demand Paging
  - Provides the ability to run programs larger than the primary memory
  - Hides differences in machine configurations
- The price is address translation on each memory reference







Conclusion: VM Features Track Historical Uses

- **Bare machine, only physical addresses**
  - One program owned entire machine
- **Batch-style multiprogramming**
  - Several programs sharing CPU while waiting for I/O
  - Base & bound: translation and protection between programs (not virtual memory)
  - Problem with external fragmentation (holes in memory), needed occasional memory defragmentation as new jobs arrived
- **Time sharing**
  - More interactive programs, waiting for user. Also, more jobs/second.
  - Motivated move to fixed-size page translation and protection, no external fragmentation (but now internal fragmentation, wasted bytes in page)
  - Motivated adoption of virtual memory to allow more jobs to share limited physical memory resources while holding working set in memory