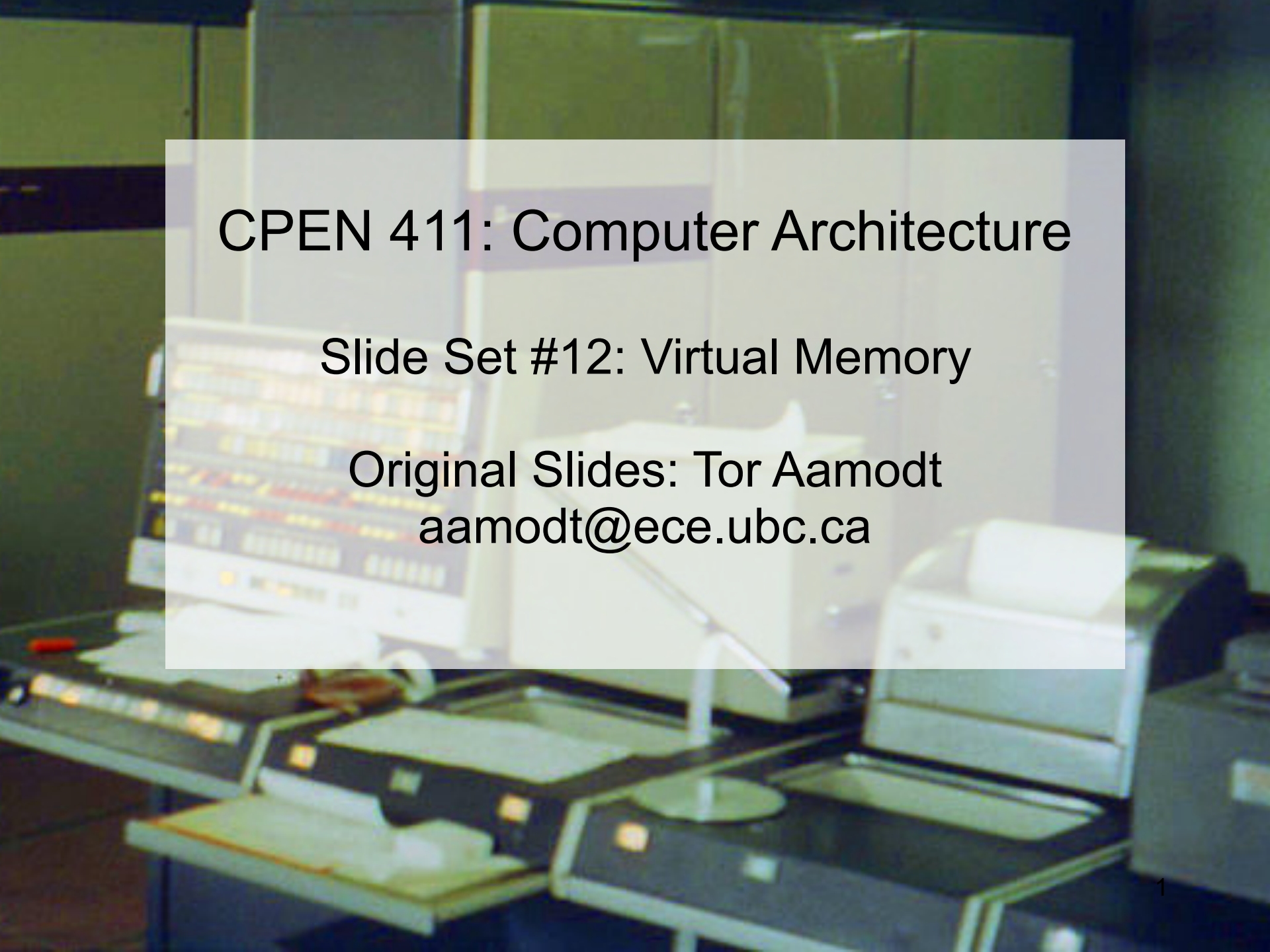


CPEN 411: Computer Architecture

Slide Set #12: Virtual Memory

Original Slides: Tor Aamodt
aamodt@ece.ubc.ca

Introduction to Slide Set 12

- In this slide set we are going to look at the virtual memory system present on all modern computer systems.

- You might recall we mentioned support for virtual memory being one of the reasons for why supporting precise exceptions is so important (which was one of the two important reasons we needed the reorder buffer). We will see how virtual memory can lead to page faults (a type of exception) in the normal operation of a program.

- The motivation for virtual memory is that it simplifies software development by freeing the programmer from having to think about low level details when trying to get their program to run correctly (they still ought to think about these details when coding for performance)

Learning Objectives

After this set of slides, you will be able to:

- Define multitasking
- Explain the motivation for using virtual memory.
- Explain what a page table is and how it is used to implement virtual memory.
- Explain what a translation look aside buffer (TLBs) is and how it operates.
- Explain interaction of TLBs and caches (virtual indexed physically tagged cache, synonym problem)

Multitasking

- Most OS's multitask
  - Run Program A and B at the same time
- Most CPUs must support multitasking
  - Not run at exactly the same time:
    - Run Program A for 20ms
    - Run Program B for 20ms
    - Run Program A for 20ms, etc
- NOTE: Multitasking is different than "hardware multithreading".

Multitasking

- Most OS's multitask
  - Run Program A and B at the same time
- Most CPUs must support multitasking
  - Not run at exactly the same time:
    - Run Program A for 20ms
    - Run Program B for 20ms
    - Run Program A for 20ms, etc
- NOTE: Multitasking is different than "hardware multithreading".

Does multitasking require a multicore processor?

A: Yes, very sure
B: Yes, but not sure
C: Not sure
D: No, but not sure
E: No, very sure

Multitasking

- Most OS's multitask
  - Run Program A and B at the same time
- Most CPUs must support multitasking
  - Not run at exactly the same time:
    - Run Program A for 20ms
    - Run Program B for 20ms
    - Run Program A for 20ms, etc
- NOTE: Multitasking is different than "hardware multithreading".

Does multitasking require a multicore processor?

A: Yes, very sure
B: Yes, but not sure
C: Not sure
D: No, but not sure
E: No, very sure ✓

Multitasking

- Most OS's multitask
  - Run Program A and B at the same time

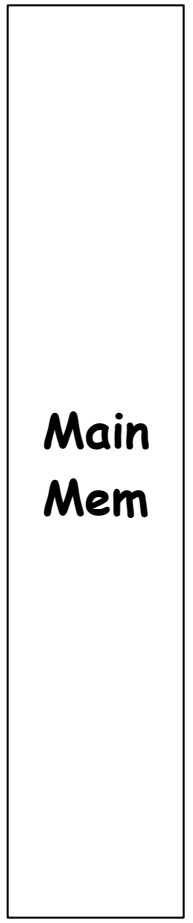
- Most CPUs must support multitasking
  - Not run at exactly the same time:
    - Run Program A for 20ms
    - Run Program B for 20ms
    - Run Program A for 20ms, etc

- NOTE: Multitasking is different than "hardware multithreading". Hardware multithreading means hardware can run instructions from two or more threads at exactly the same time. Multitasking involves providing short "time slices" to programs (so at any time only instructions from one thread are running if the hardware is single threaded).

Multiple Programs Using Memory

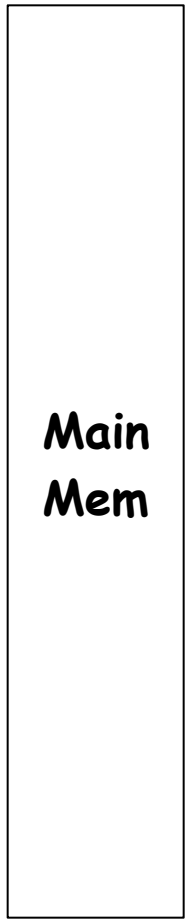
- Computer has 1024 MB DRAM
  - Internet Explorer (IE) uses 30 MB
  - MS Word uses 20 MB
  - Vista uses 1280 MB
  - Total: 1330 MB
- How to run all three programs at once?
- It gets worse….
  - Firefox assumes memory starts at address 0
  - Word assumes memory starts at address 0
    - How can they both run if they both assume memory starts at 0 ??
  - Vista ? It's the OS…. who knows!

Multiple Programs Using Memory



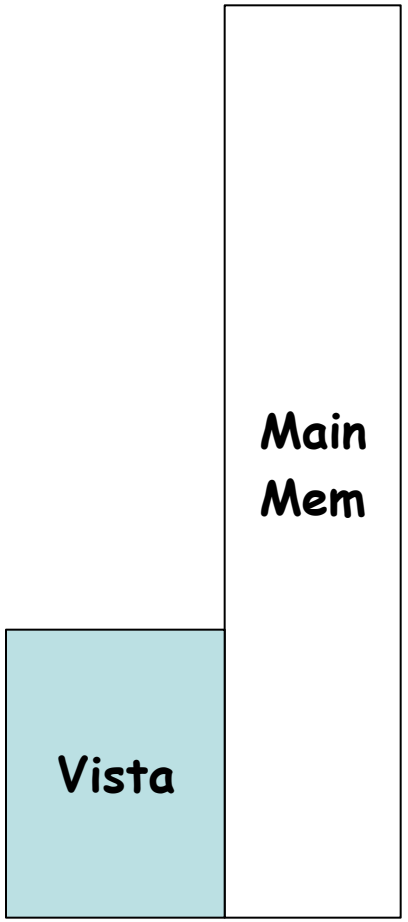
Multiple Programs Using Memory

- It gets even worse



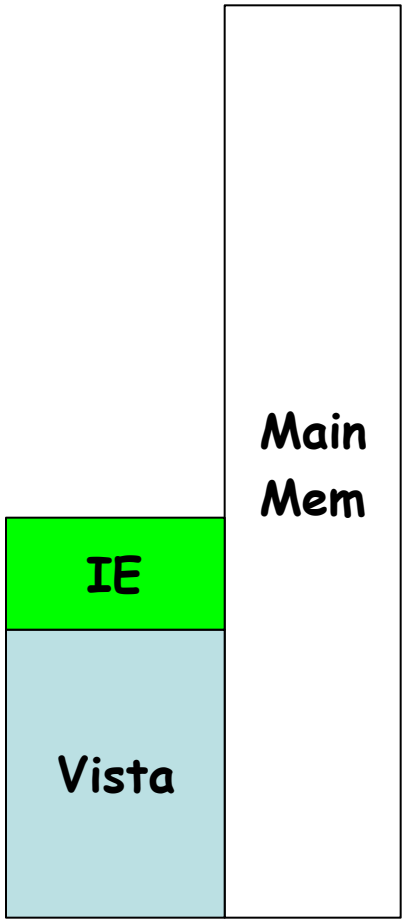
Multiple Programs Using Memory

- It gets even worse



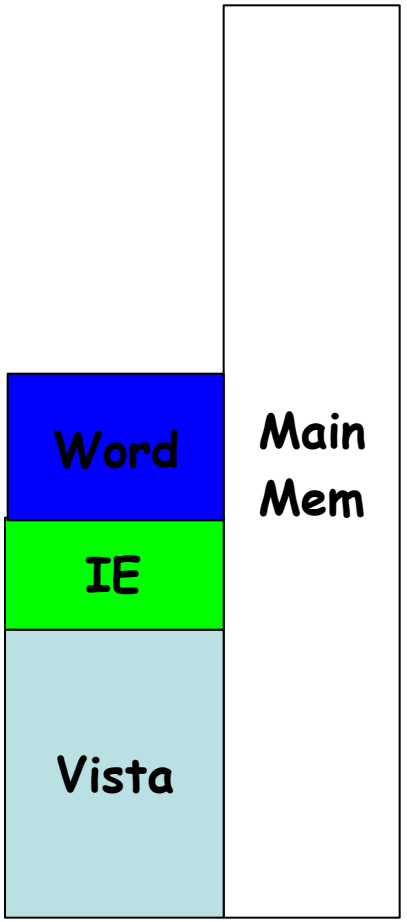
Multiple Programs Using Memory

- It gets even worse



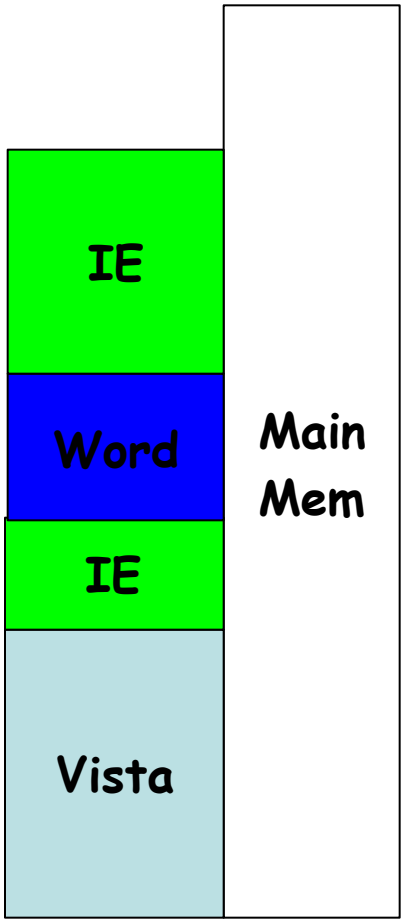
Multiple Programs Using Memory

- It gets even worse



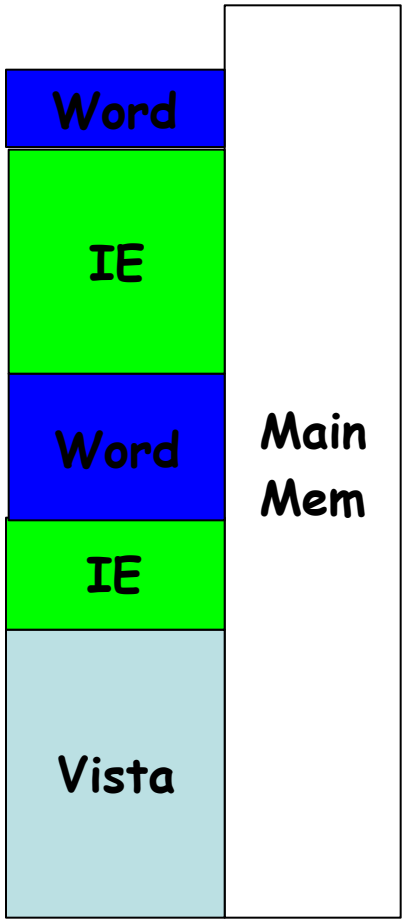
Multiple Programs Using Memory

- It gets even worse



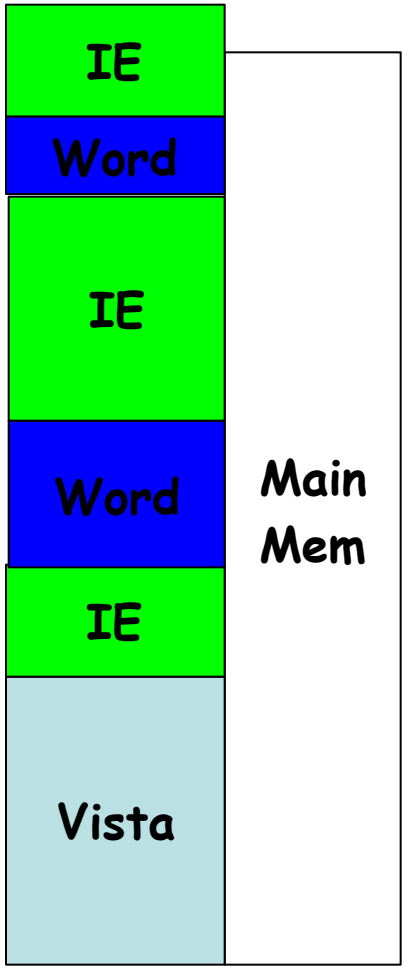
Multiple Programs Using Memory

- It gets even worse



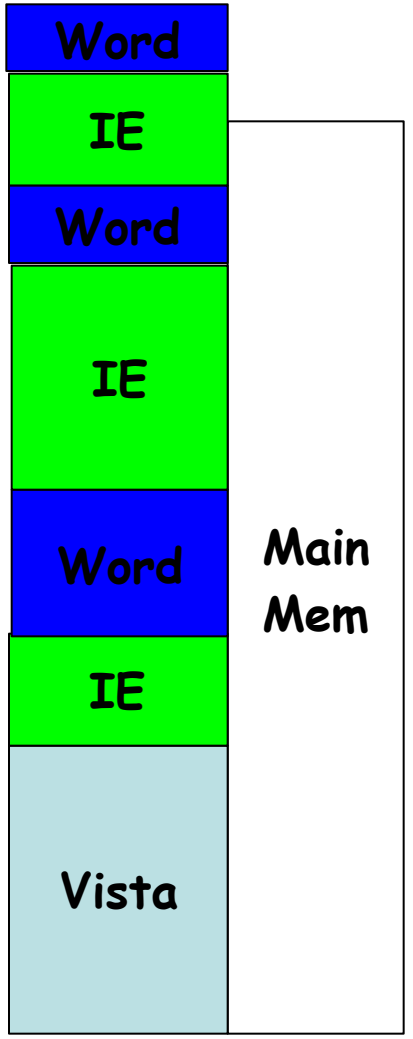
Multiple Programs Using Memory

- It gets even worse



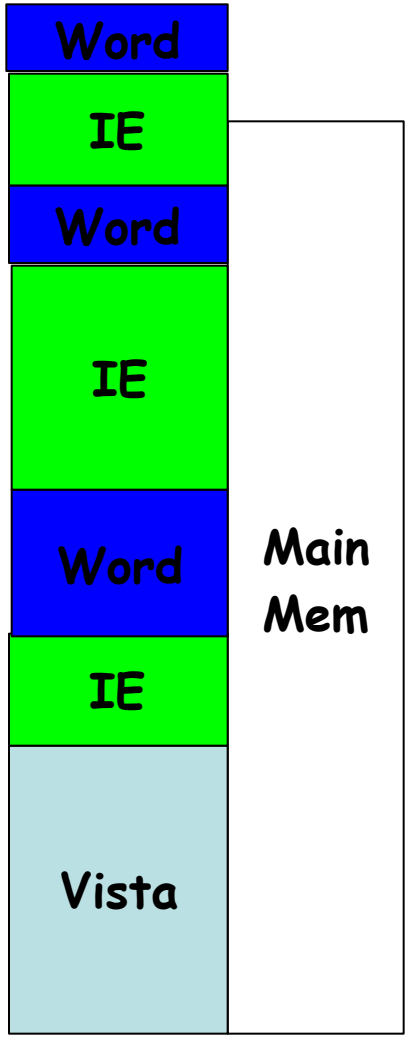
Multiple Programs Using Memory

- It gets even worse



Multiple Programs Using Memory

- It gets even worse
  - Vista starts using 1280 MB
  - IE starts using 30 MB
  - Word starts using 20 MB
  - User opens web page (cnn.com), IE wants + 72MB
  - User opens document (ps3 soln) Word wants +30MB
  - User opens 2nd web page(youtube), IE wants +40MB
  - User edits document, Word wants +10MB
  - TOTAL
    - Vista 1280MB, IE 142 MB, Word 60 MB

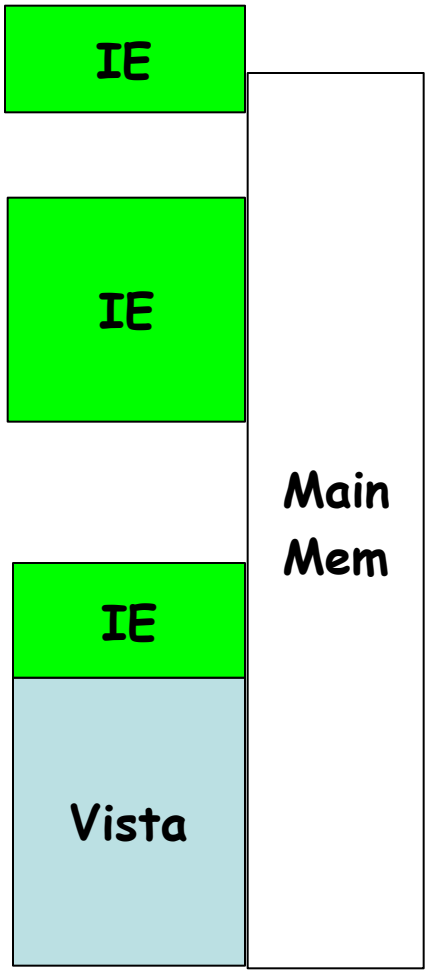


How can we divide up memory between programs?

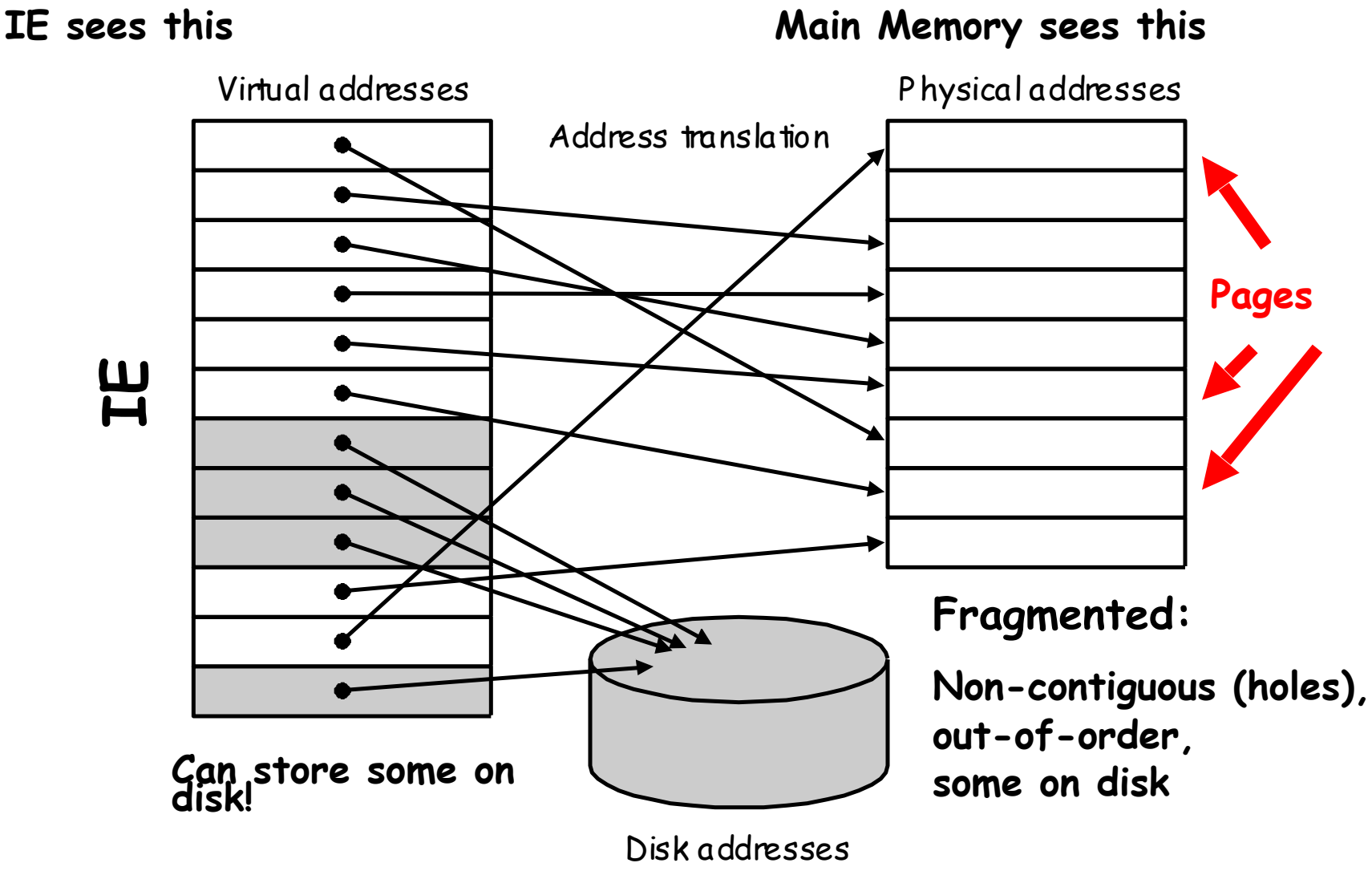
- It gets even worse
  - Exit Word
  - Start engineering program
    - Wants 300MB for a matrix
    - Must be contiguous!

    - How?
    - Move IE? Not realistic

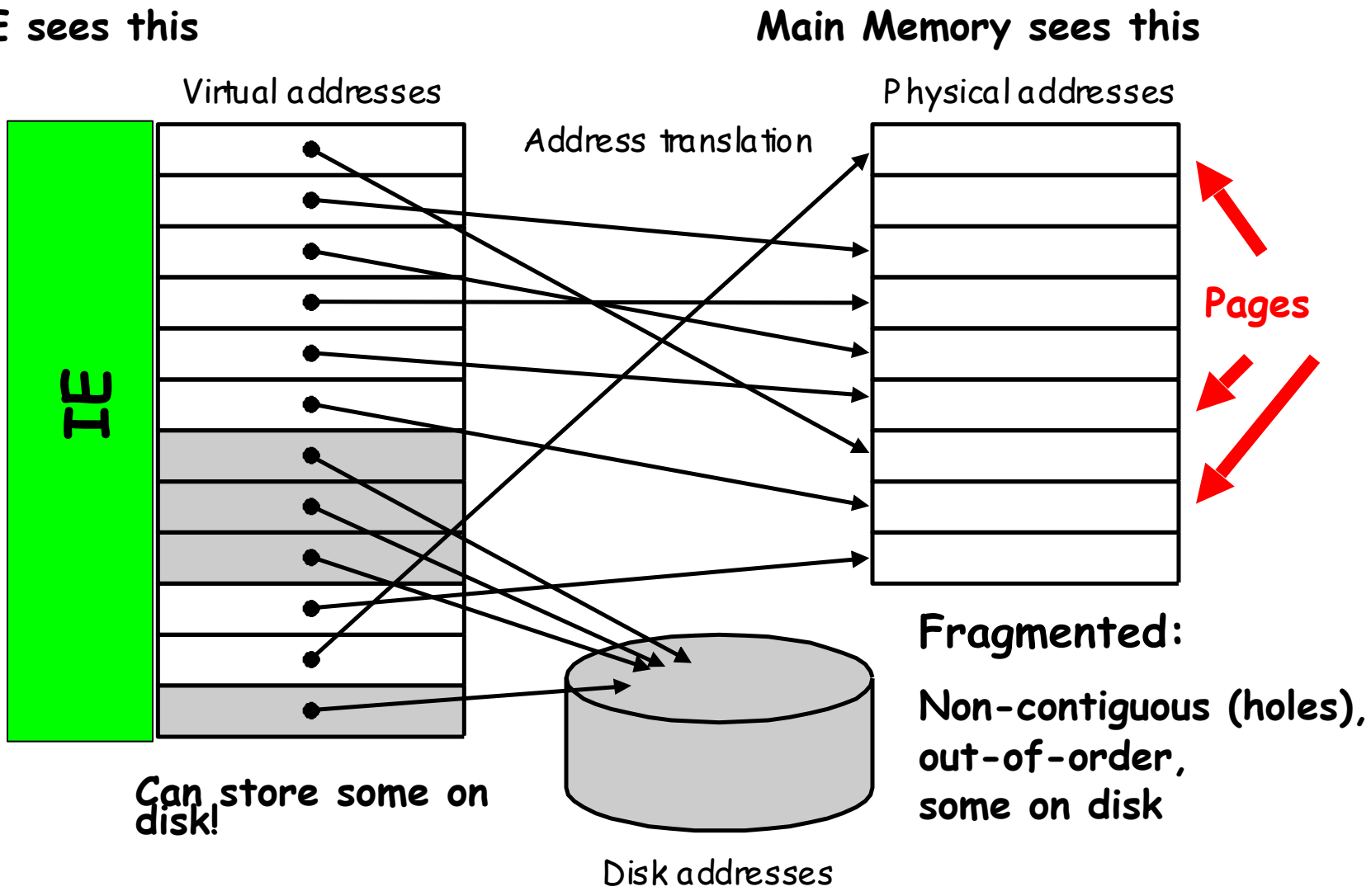
- Solution?
  - Divide Main Memory into **pages**, say 4KB each
  - Divide Programs into pages (same size)
  - Map Program Pages into Main Memory Pages
  - Store extra Program Pages on disk



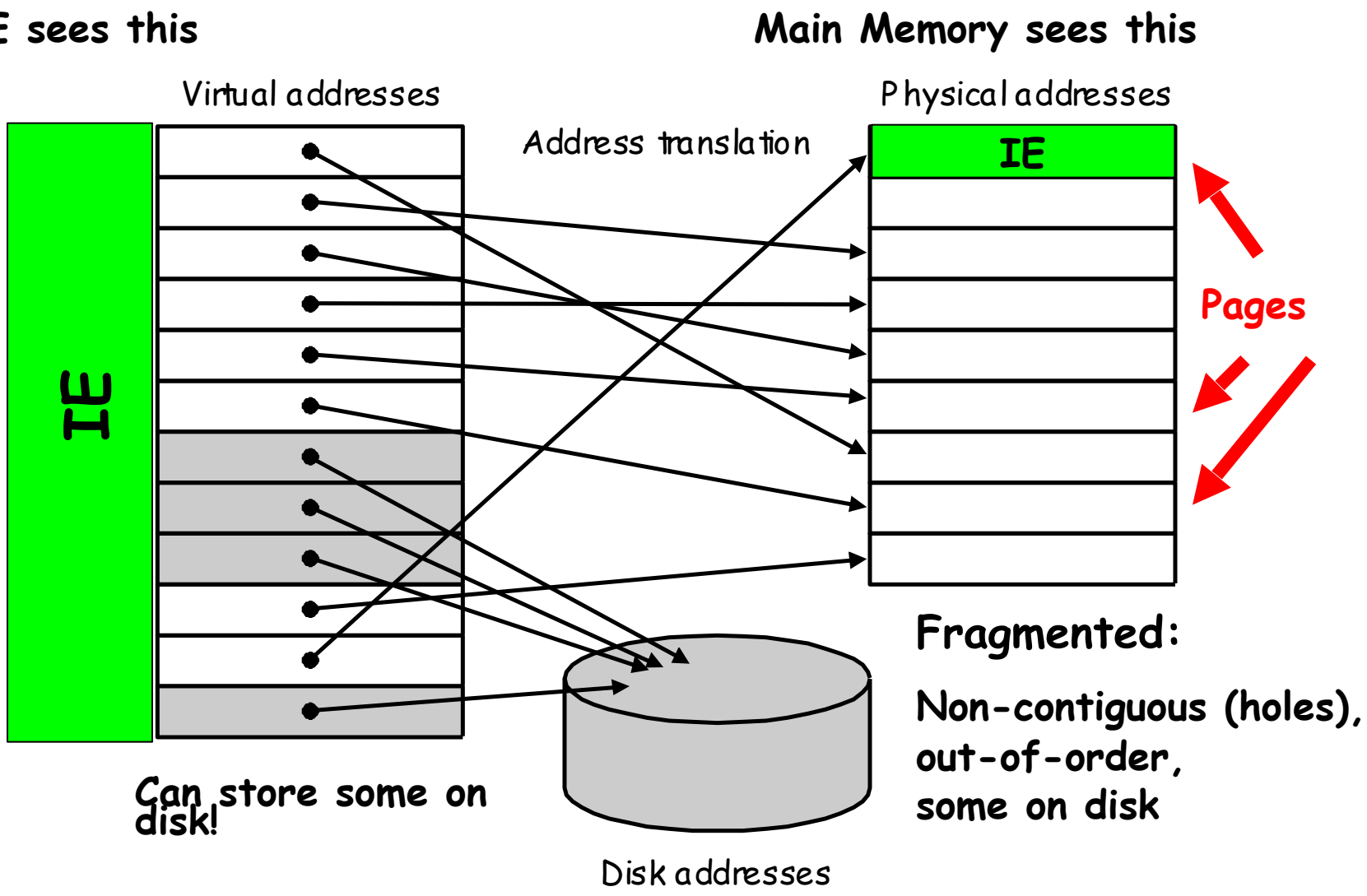
Solution: Virtual Memory



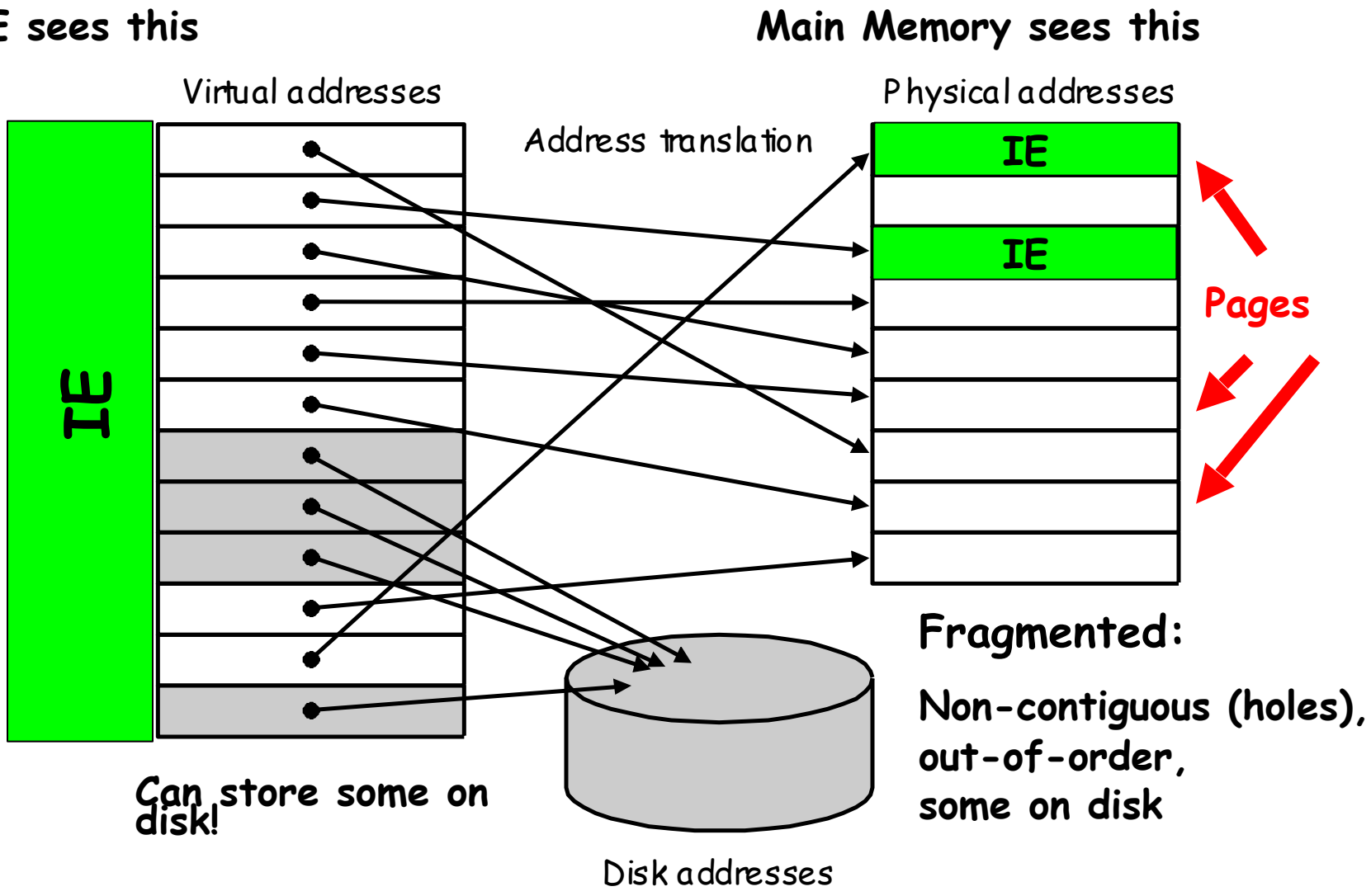
Solution: Virtual Memory



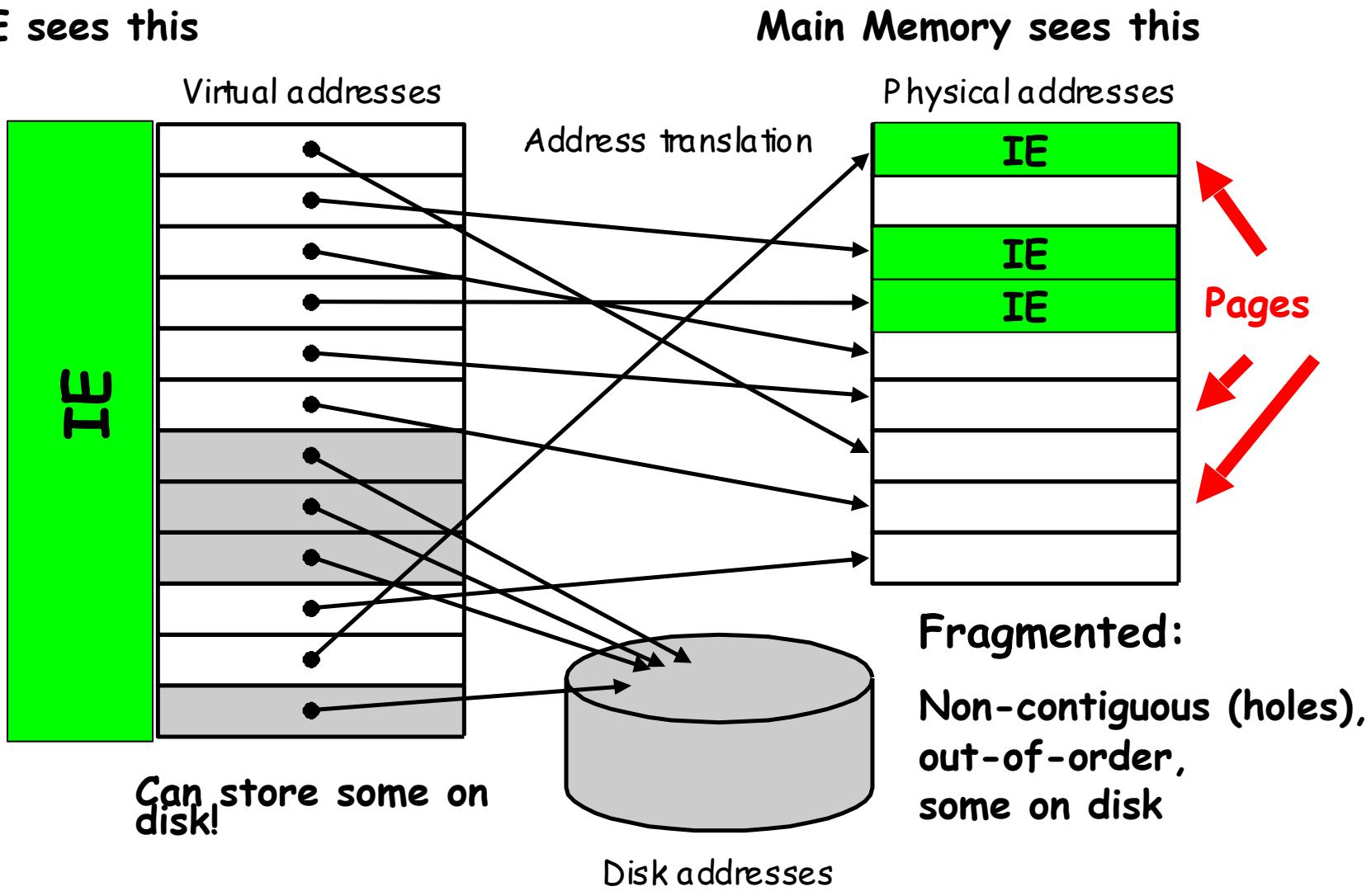
Solution: Virtual Memory



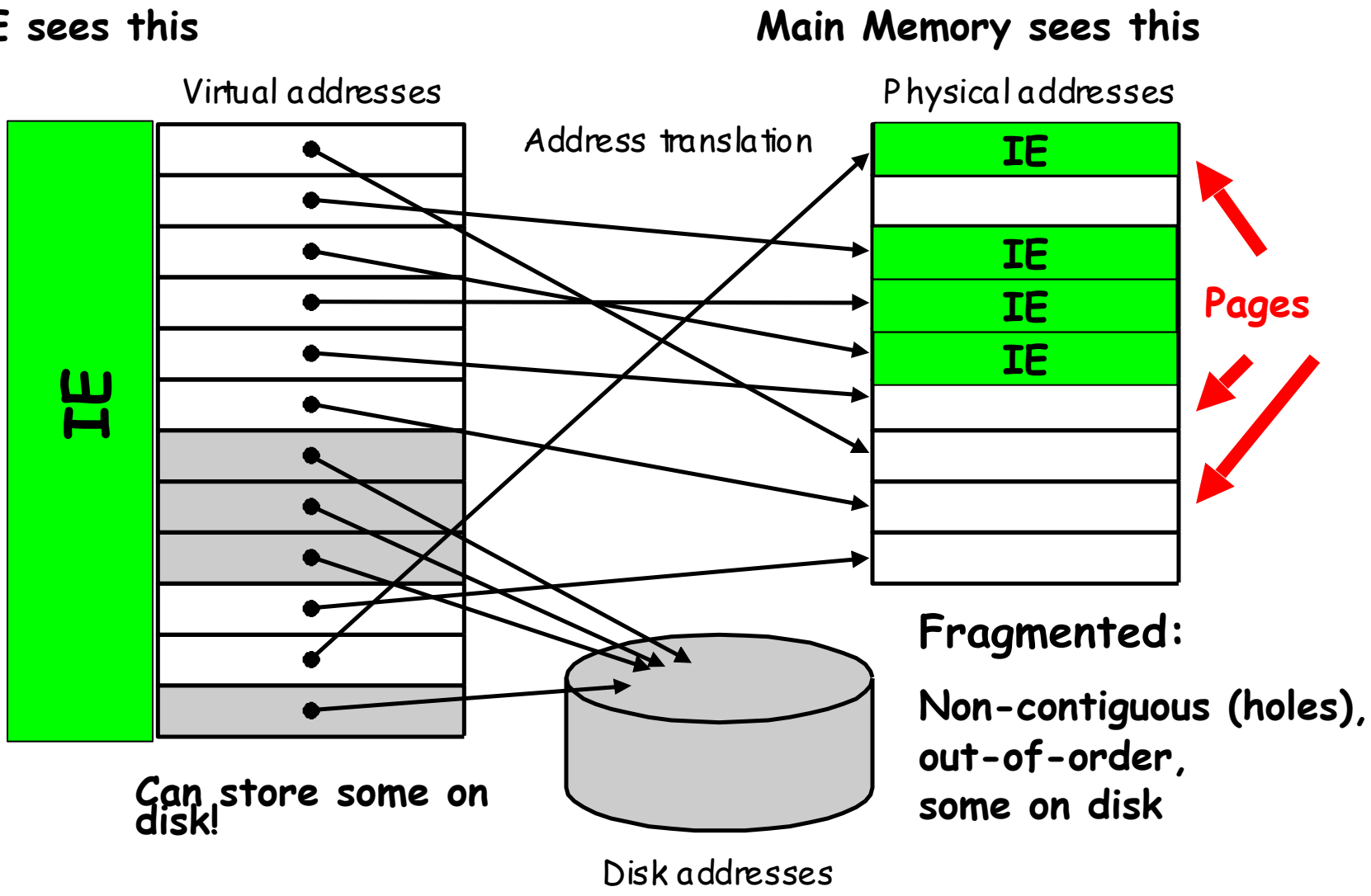
Solution: Virtual Memory



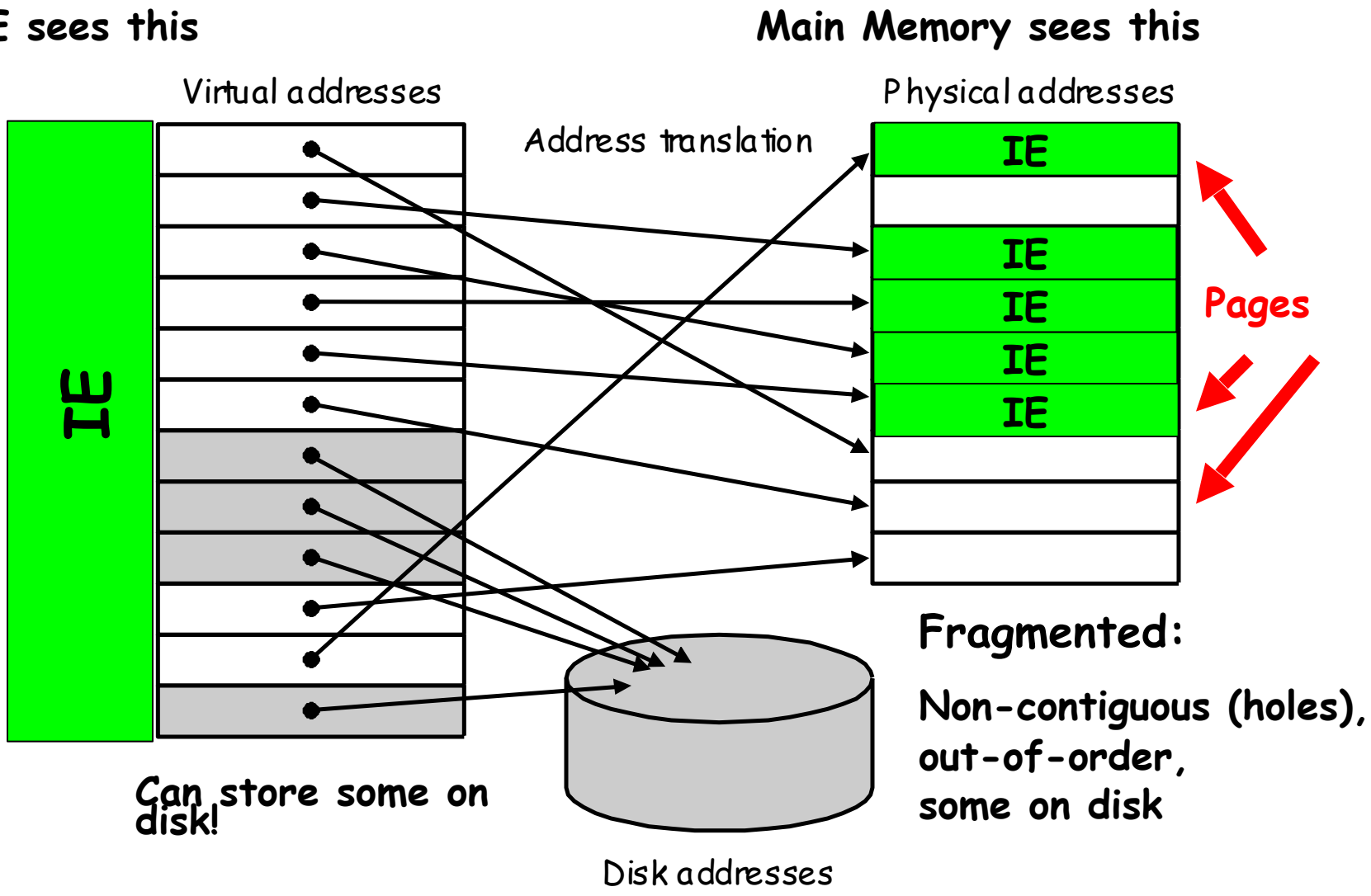
Solution: Virtual Memory



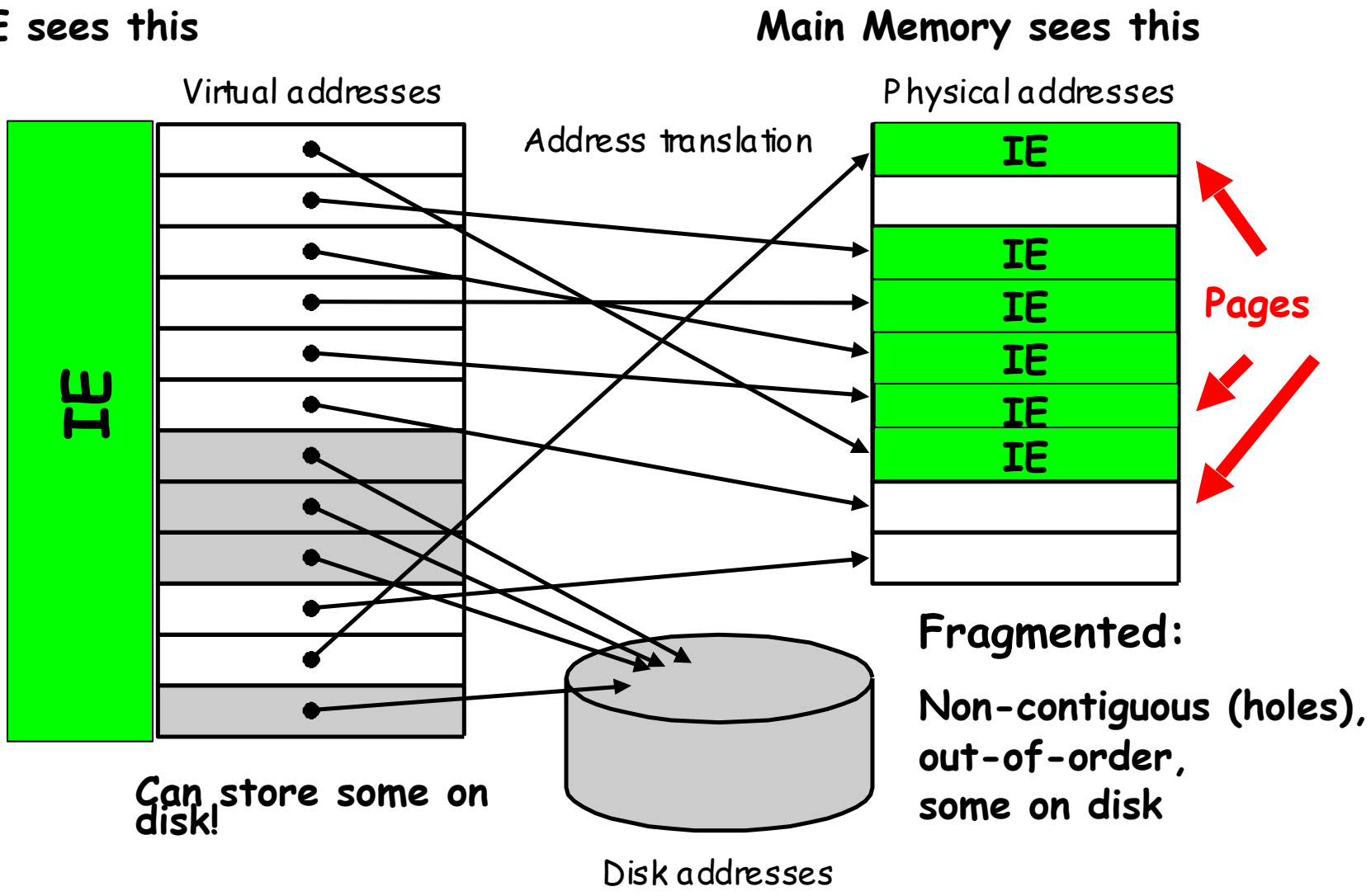
Solution: Virtual Memory



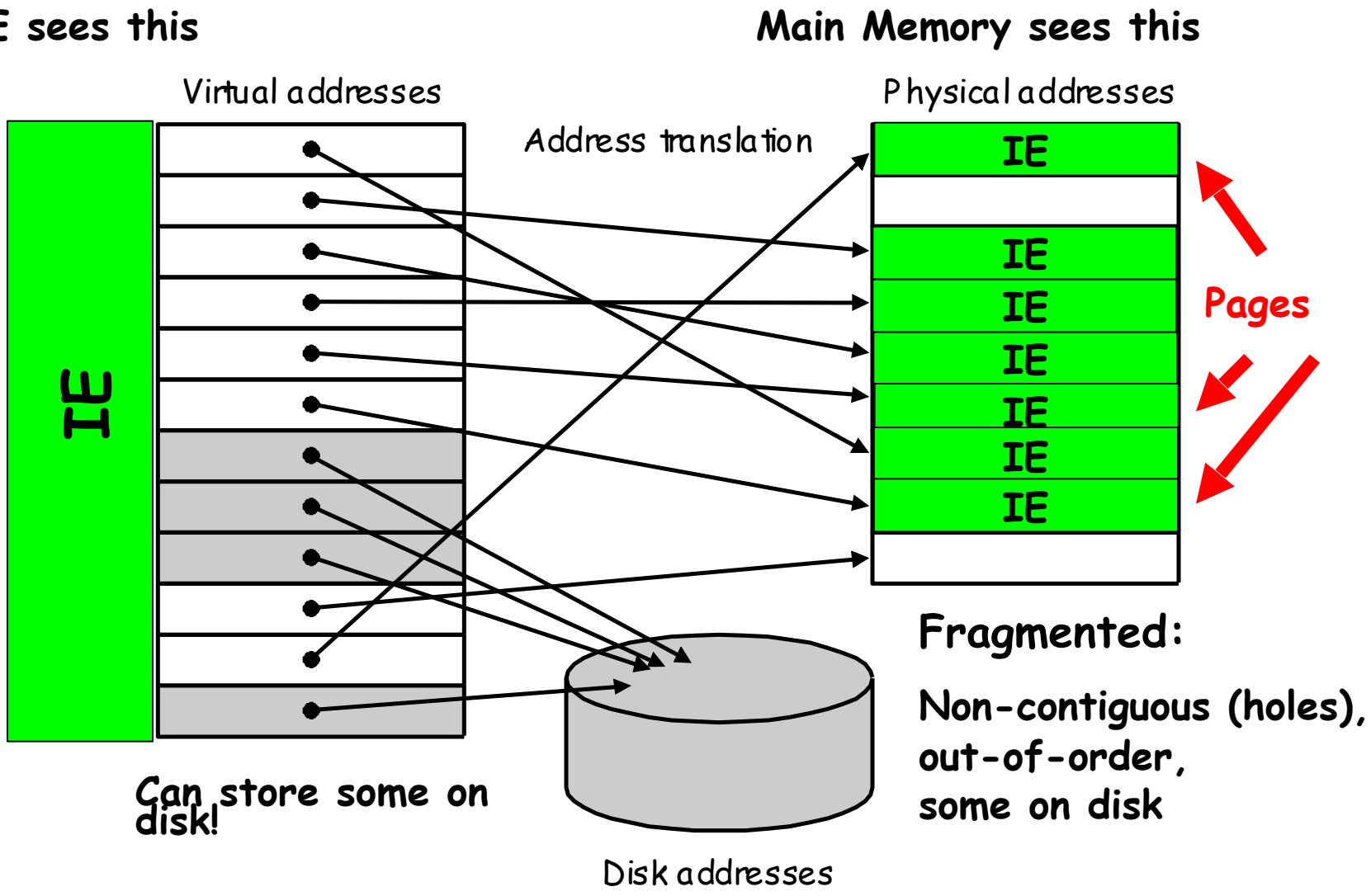
Solution: Virtual Memory



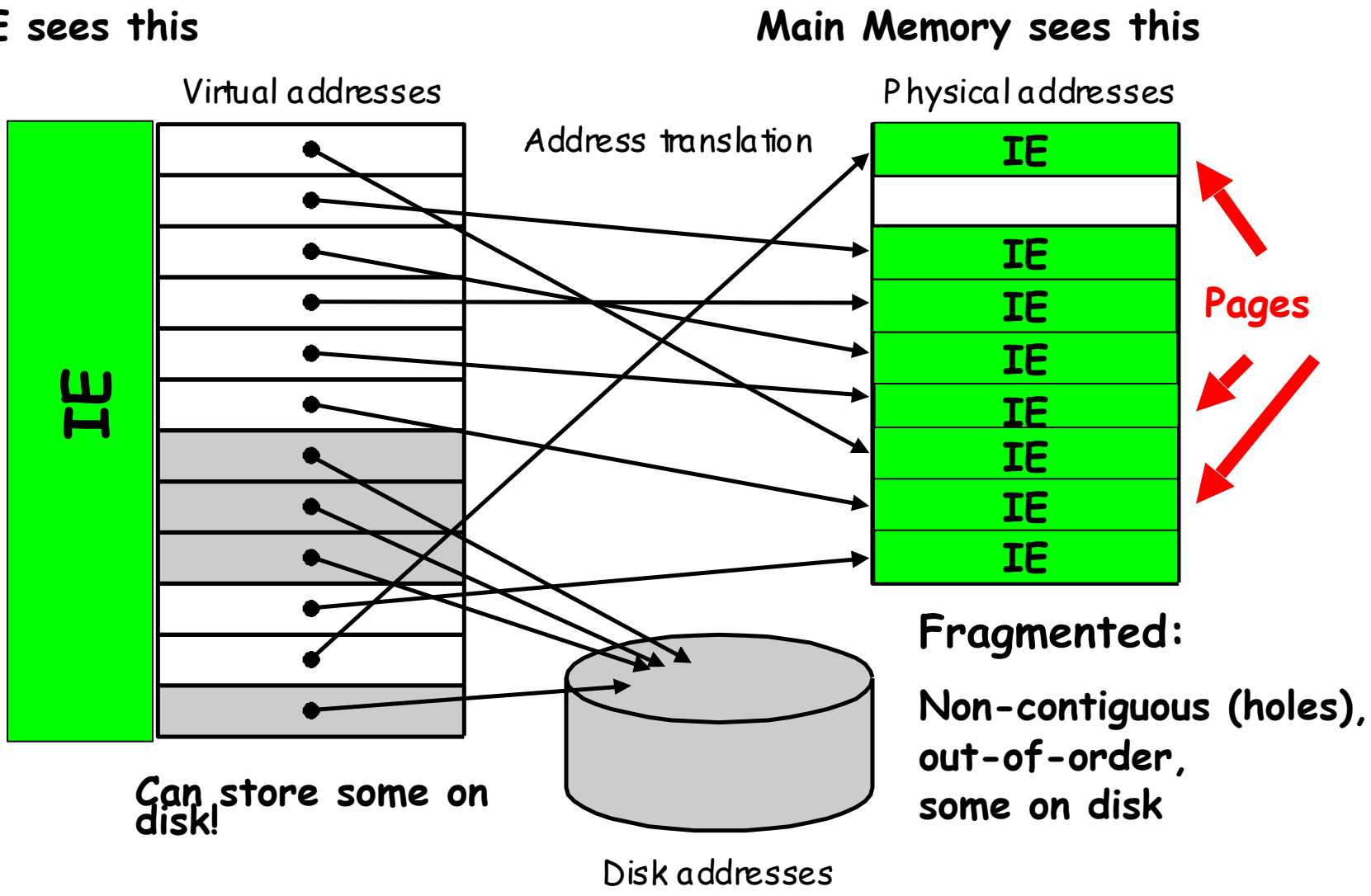
Solution: Virtual Memory



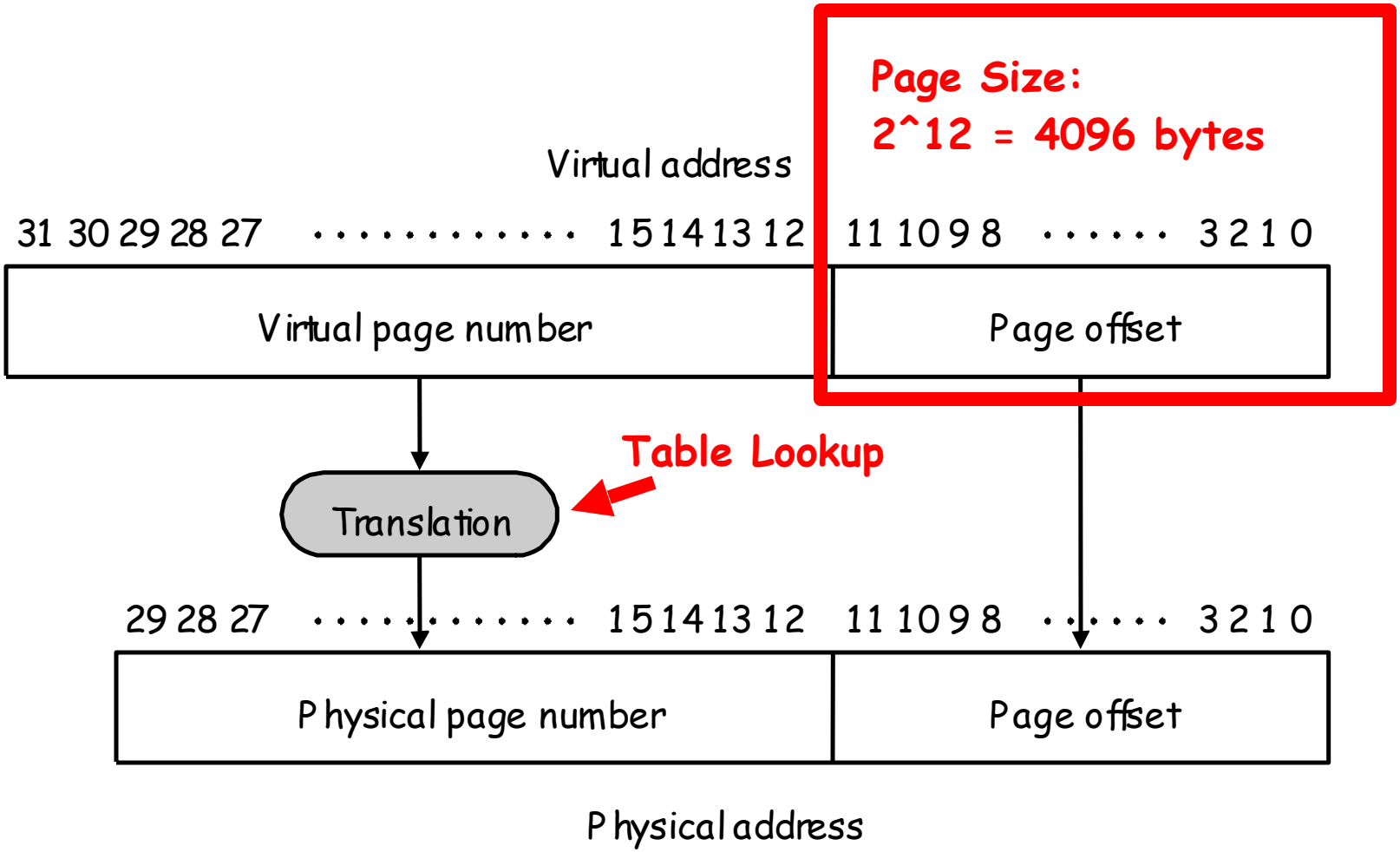
Solution: Virtual Memory



Solution: Virtual Memory



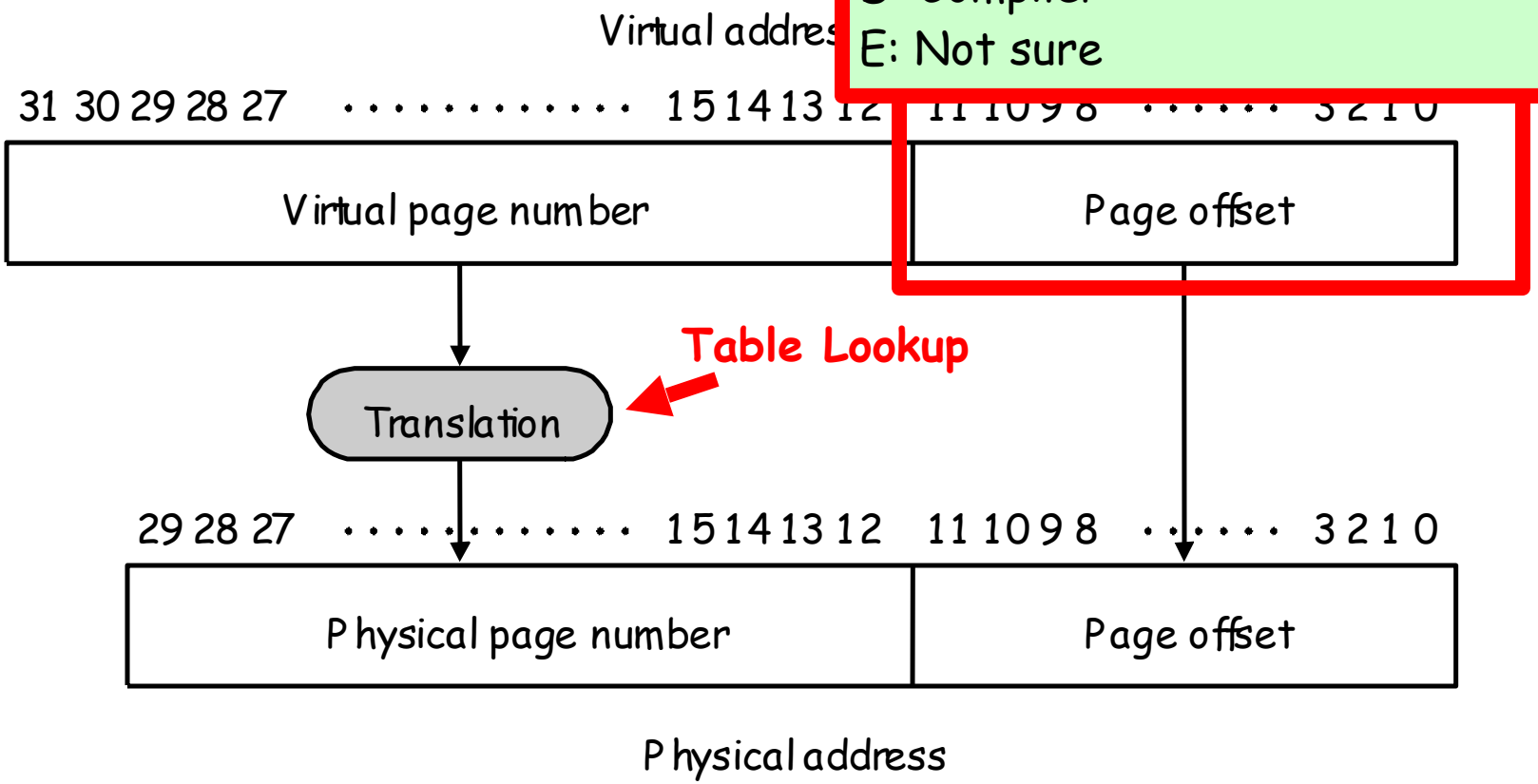
Address Translation



Address Tra

Who should determine "translation"?

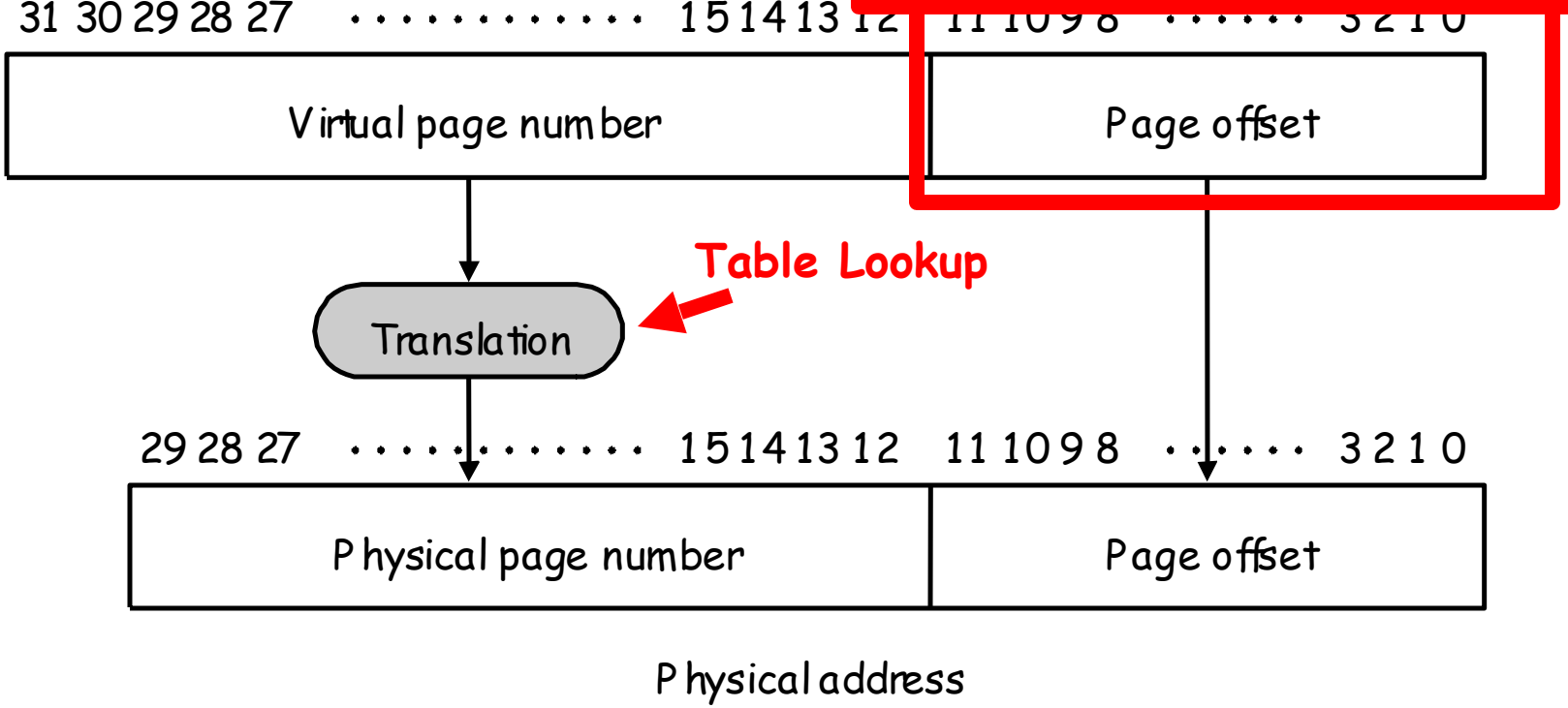
A: Programmer
B: Chip designer
C: Operating system
D: Compiler
E: Not sure



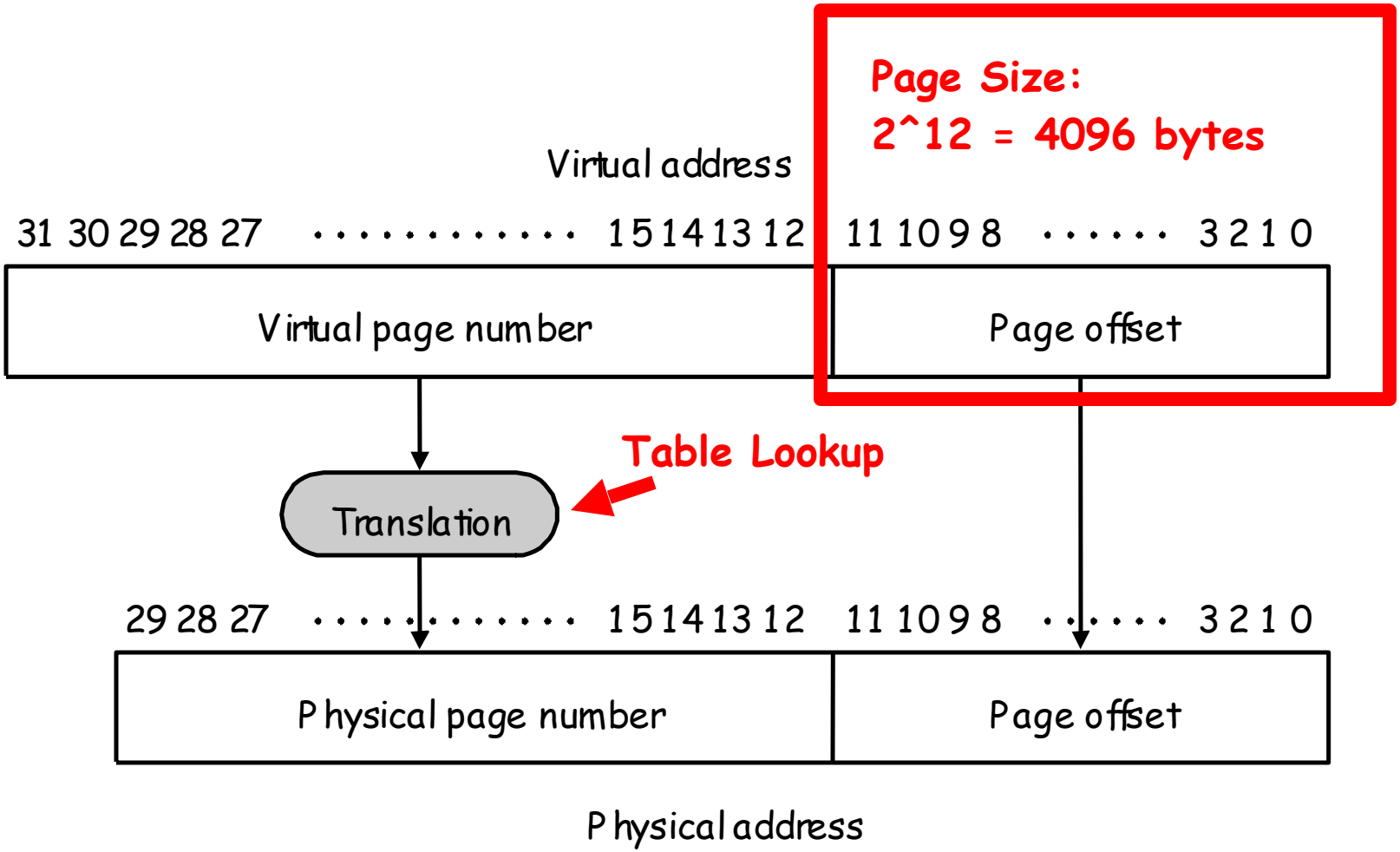
Address Tra

Who should determine "translation"?

A: Programmer
B: Chip designer
C: Operating system ✓
D: Compiler
E: Not sure



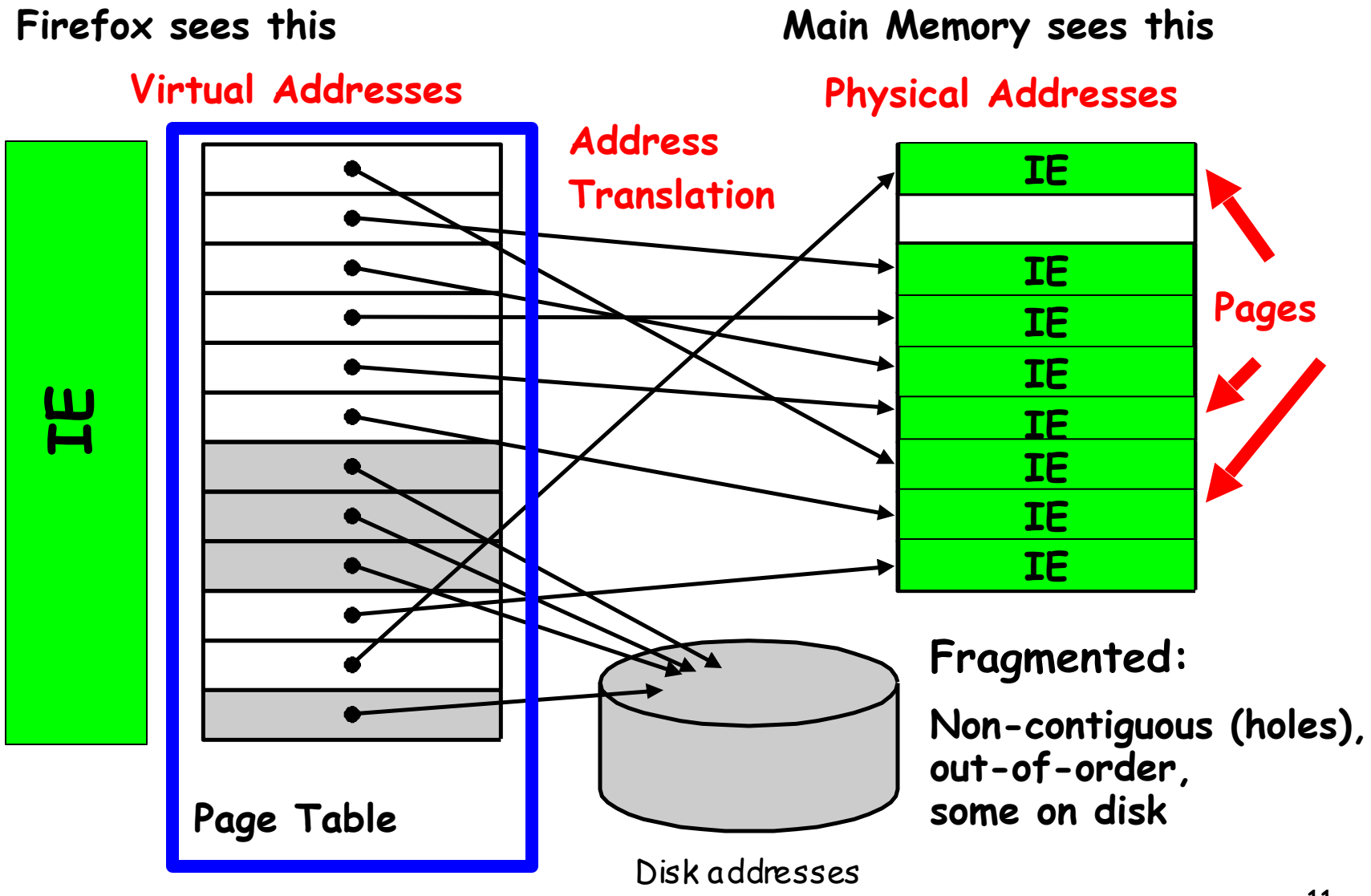
Address Translation



Virtual Memory

- Program Viewpoint
  - Addresses start at 0
  - Malloc() gives program **contiguous** memory (consecutive addresses)
  - Malloc(), Free(), Malloc(), Free()
    - Can malloc() re-use parts that are freed?
    - Causes **internal fragmentation** within one program
  - There is lots of free memory available
- CPU/OS Viewpoint
  - Keep all programs separate
    - Don't all start at 0
    - Security implications…. programs should not read each other's data!
  - Dynamic program behavior causes memory fragmentation
    - Starting & ending programs, swapping to disk
    - Causes **external fragmentation** between programs
    - Mitigated by virtual memory.
  - Limited memory available
    - Use Disk to hold extra data
    - Use Main Memory like a cache for the program data stored on disk
- New structure: Page Table

Page Table: Big Lookup Table



Page Table

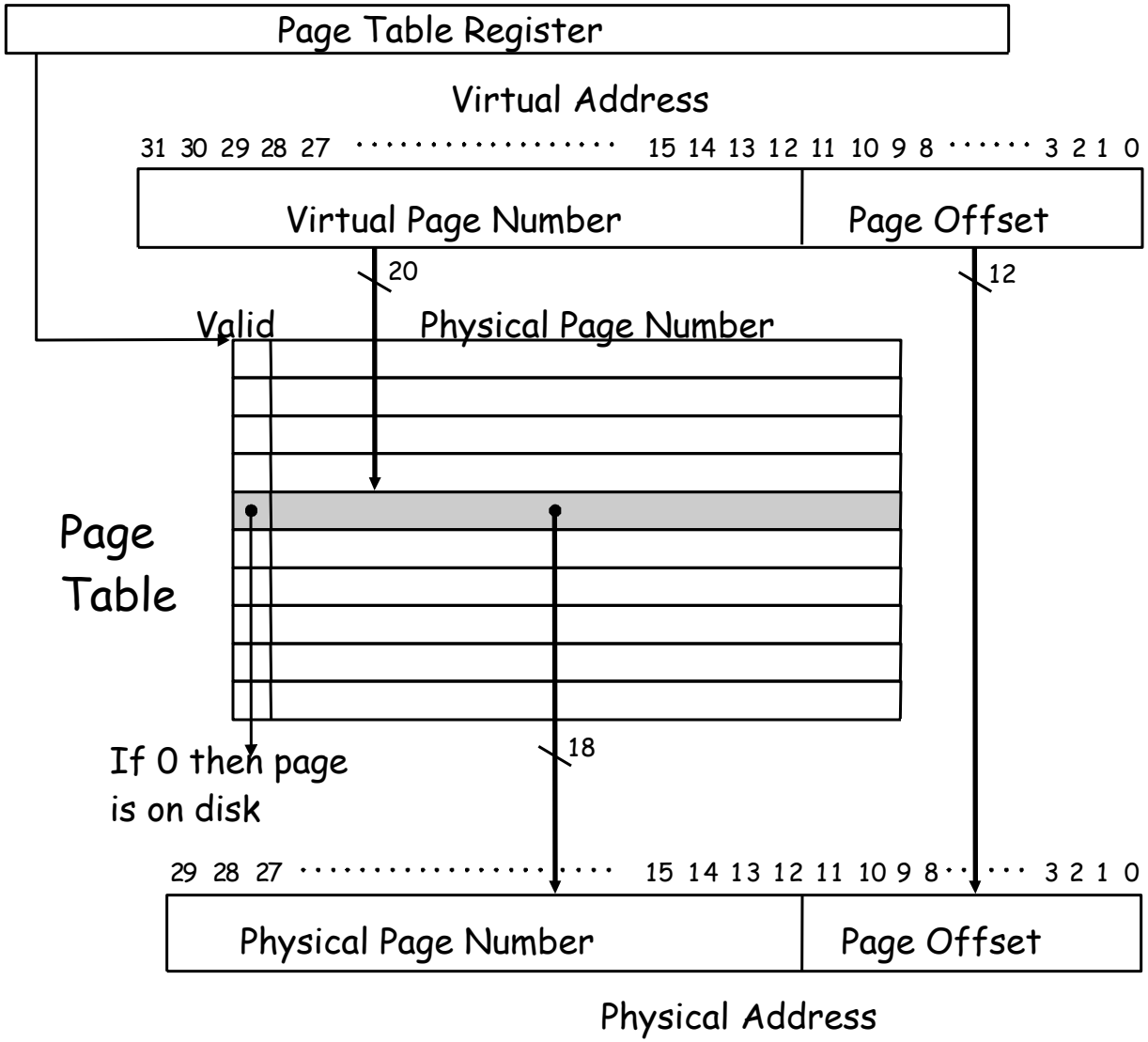
- Holds virtual-to-physical address translations
- Access like a memory
  - Input: Virtual Address
    - Only need Virtual PageNumber portion (upper bits)
    - Lower bits are PageOffset, ie which byte inside the page
  - Output: Physical Address
    - Lookup gives a Physical PageNumber
    - Combine with PageOffset to form entire Physical Address
- Where to hold the PageTable for a program?
  - Dedicated memory inside CPU? Too big! not done!
  - Instead, store it in main memory

Page Table Translation Structure

Size of Page Table?

2^20 or ~1 million entries

1 Page Table Per Program



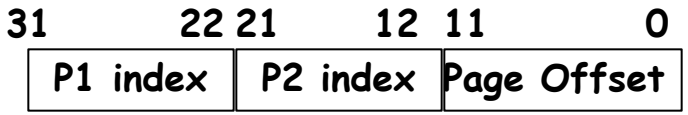
Page tables may not fit in memory!

A table for 4KB pages for a 32-bit address space has 1M entries

Each process needs its own address space!

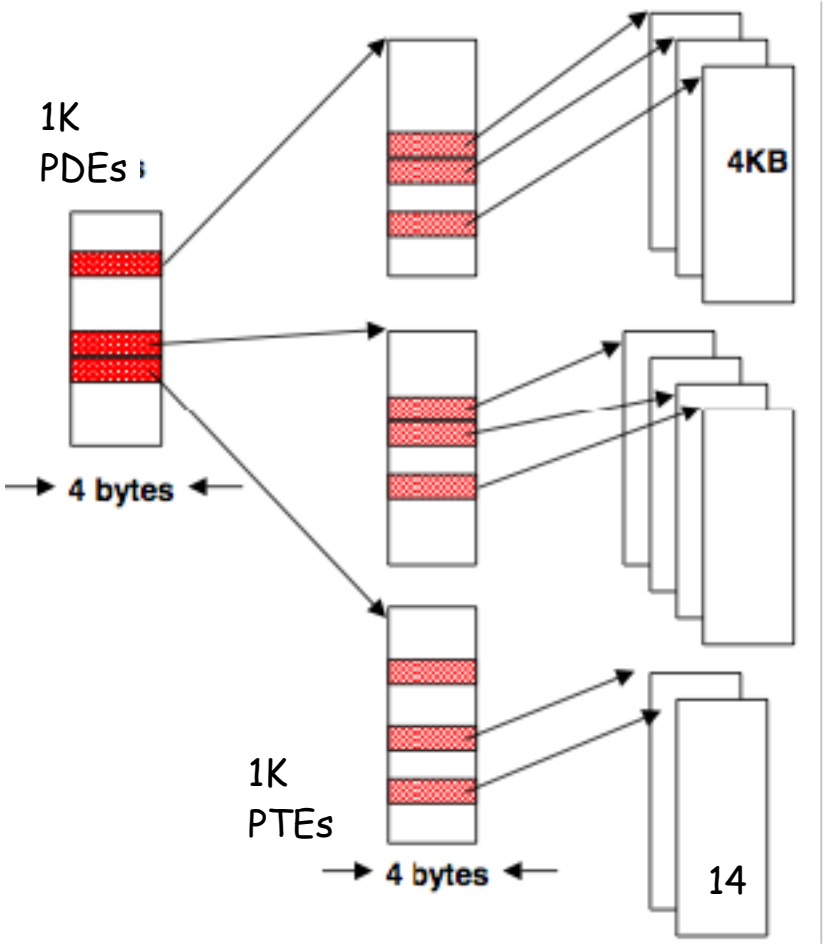
Two-level Page Tables

32 bit virtual address



Top-level table wired in main memory

Subset of 1024 second-level tables in main memory; rest are on disk or unallocated



y not fit in memory!

How many memory accesses are required for a load if virtual memory is supported with a two level page table (assuming second level entry is in memory)?

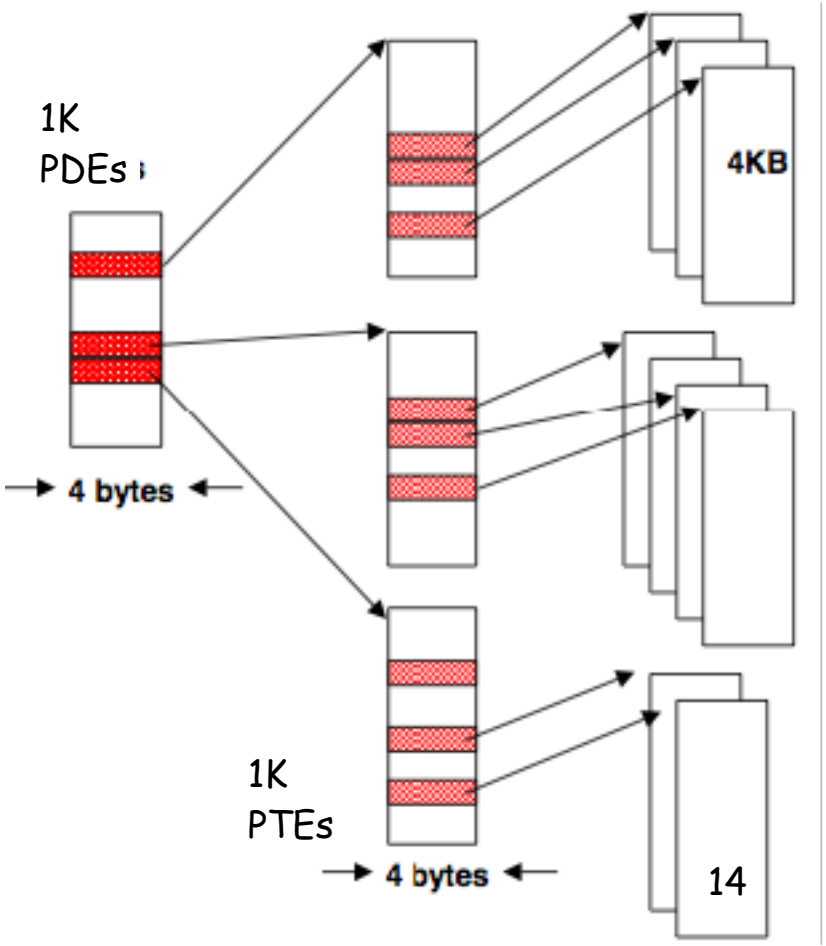
A: 1
B: 2
C: 3
D: 4
E: Not sure

pages for a 32-bit

has 1M entries

its own address space!

P1 index	P2 index	Page Offset



Top-level table wired in main memory

Subset of 1024 second-level tables in main memory; rest are on disk or unallocated

y not fit in memory!

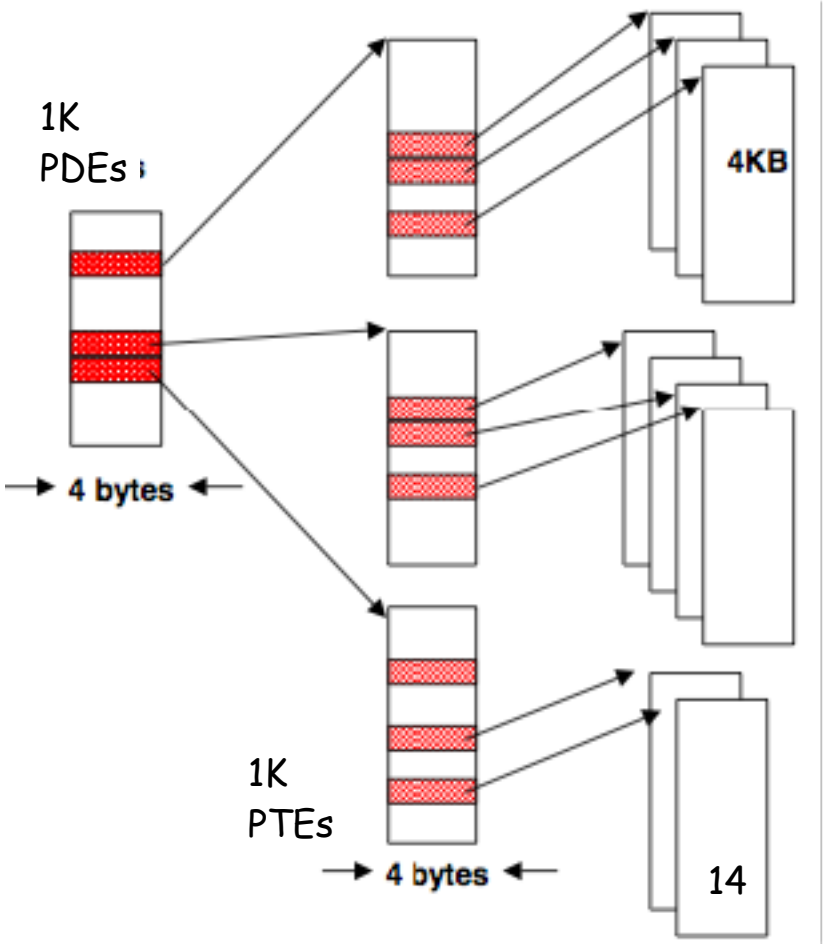
How many memory accesses are required for a load if virtual memory is supported with a two level page table (assuming second level entry is in memory)?

A: 1
B: 2
C: 3 ✓
D: 4
E: Not sure

pages for a 32-bit

has 1M entries

its own address space!



Top-level table wired in main memory

Subset of 1024 second-level tables in main memory; rest are on disk or unallocated

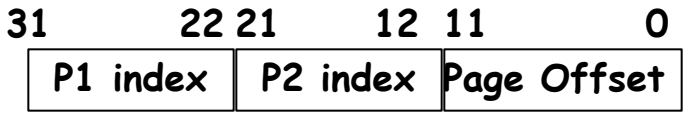
Page tables may not fit in memory!

A table for 4KB pages for a 32-bit address space has 1M entries

Each process needs its own address space!

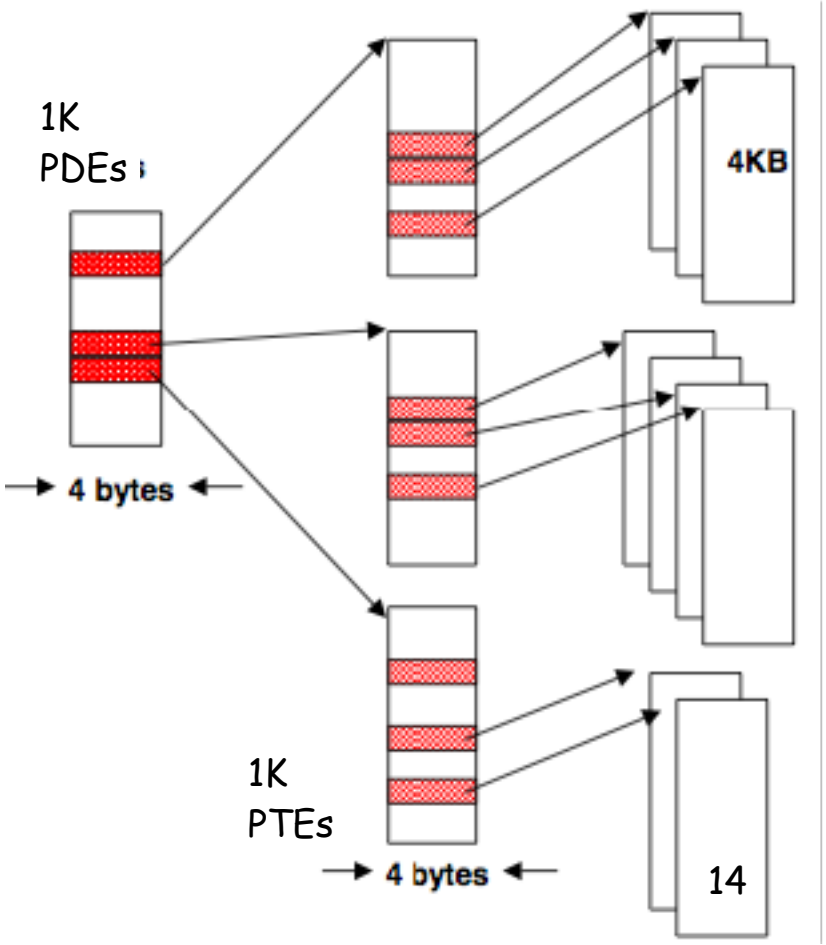
Two-level Page Tables

32 bit virtual address



Top-level table wired in main memory

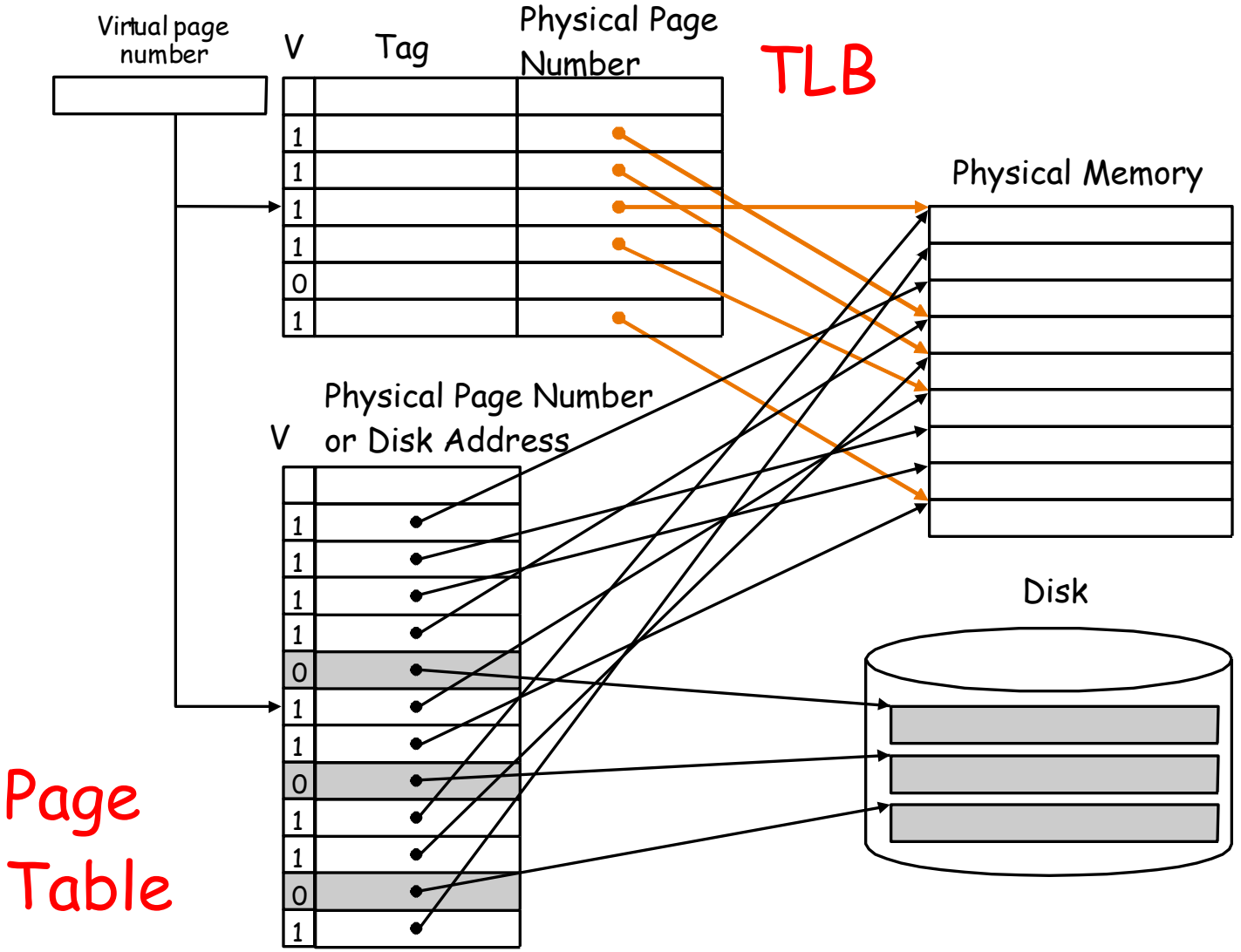
Subset of 1024 second-level tables in main memory; rest are on disk or unallocated



Page Table Implications

- CPU executes “Load” instruction
  - Recall: address is now a Virtual Address
  - First step: Lookup address translation
    - Where is page table? In main memory
    - Special CPU register called Page Table Register (PTR)
      - Holds starting address of page table
    - Read DataMem from PTR+PageNumber to get Physical Address
  - Second step: Access the data
    - Read DataMem from Physical Address
- Every load/store requires TWO memory accesses
  - Slow!
  - Can we speed up the **translation** step? Yes, use a cache!

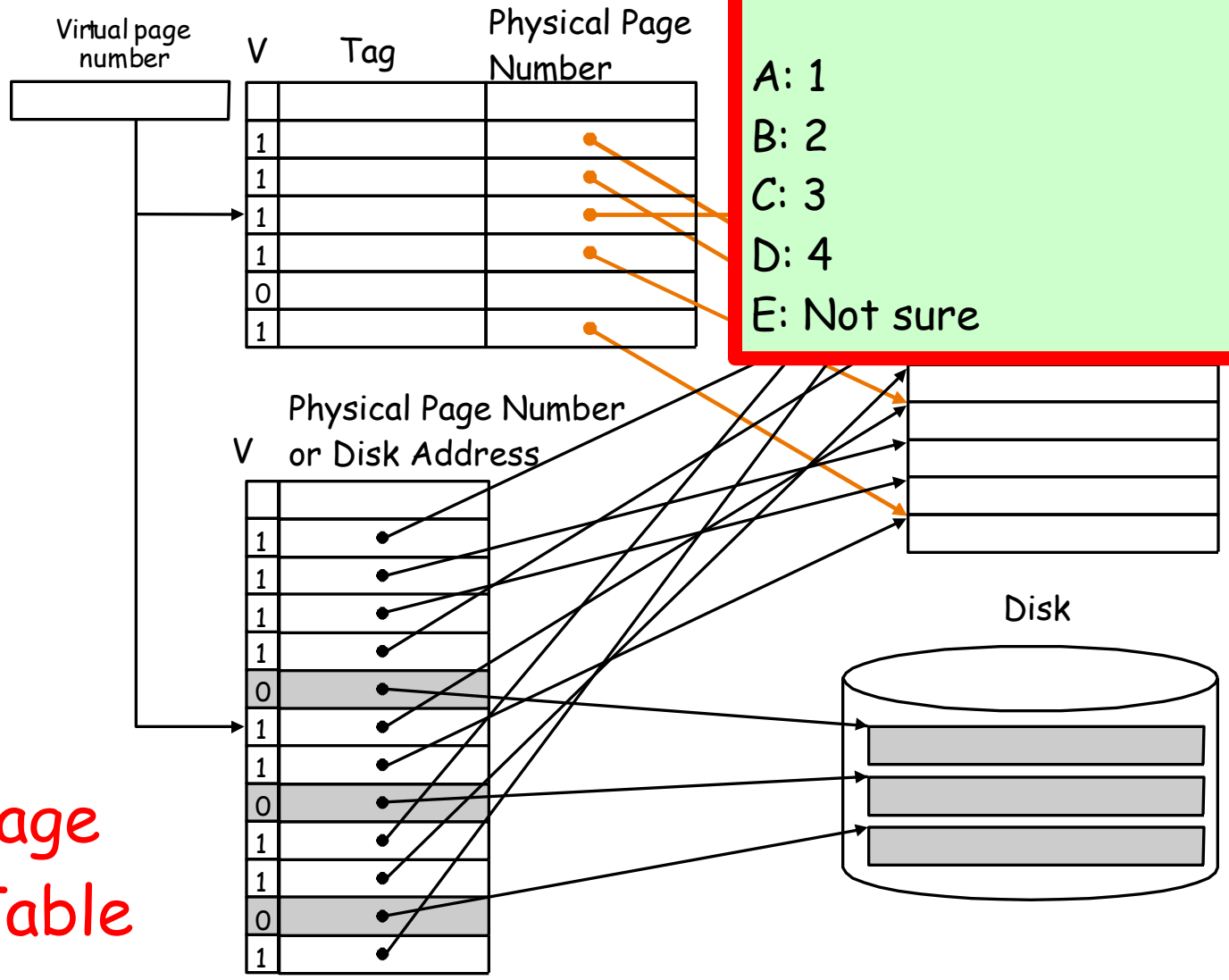
TLB: Translation Looksaside Buffer (A special cache for the Page Table)



TLB: Translation Lo
(A special cache for

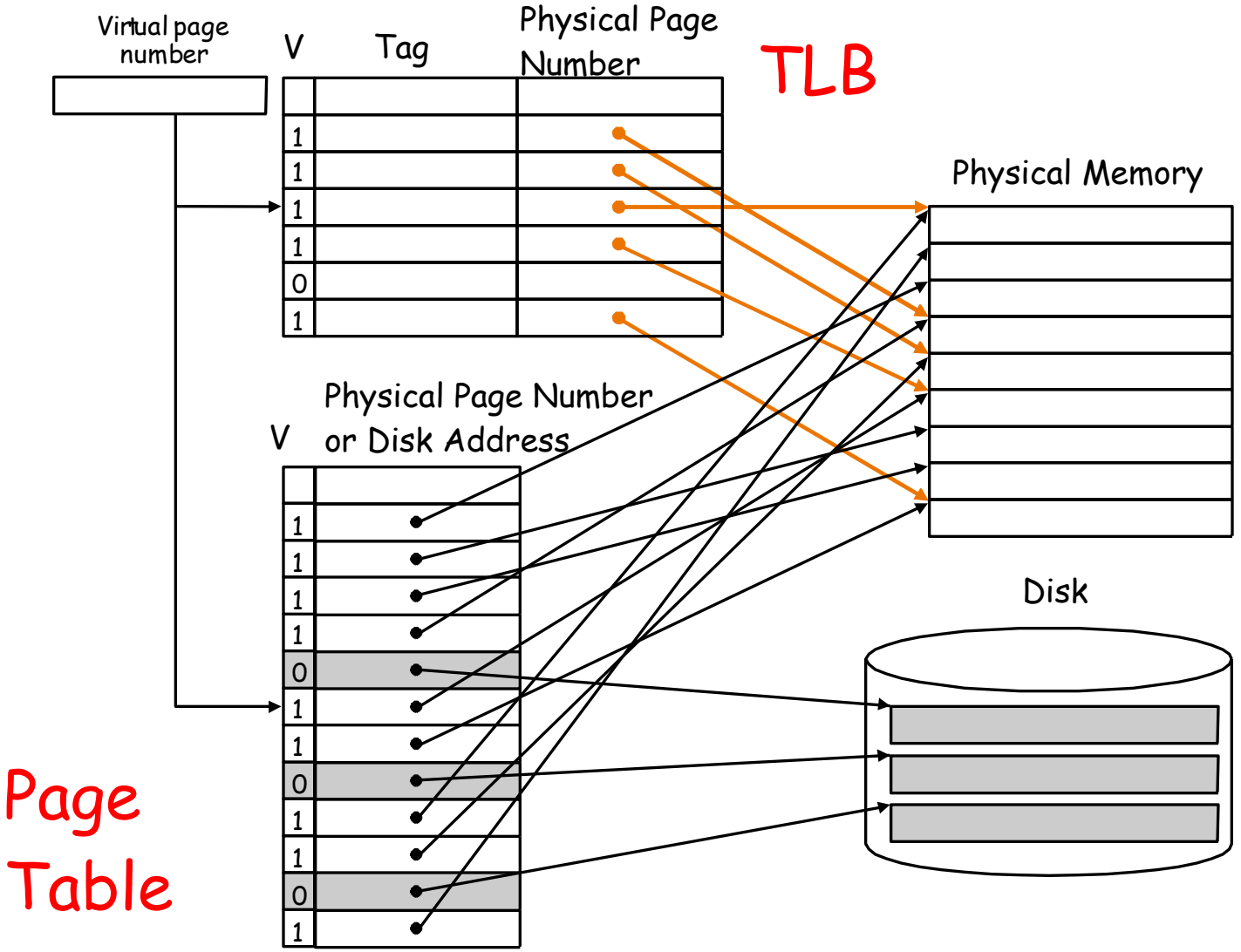
Assuming data we want is NOT in the cache, how many accesses to DRAM when a TLB hit?

A: 1
B: 2
C: 3
D: 4
E: Not sure



Page Table

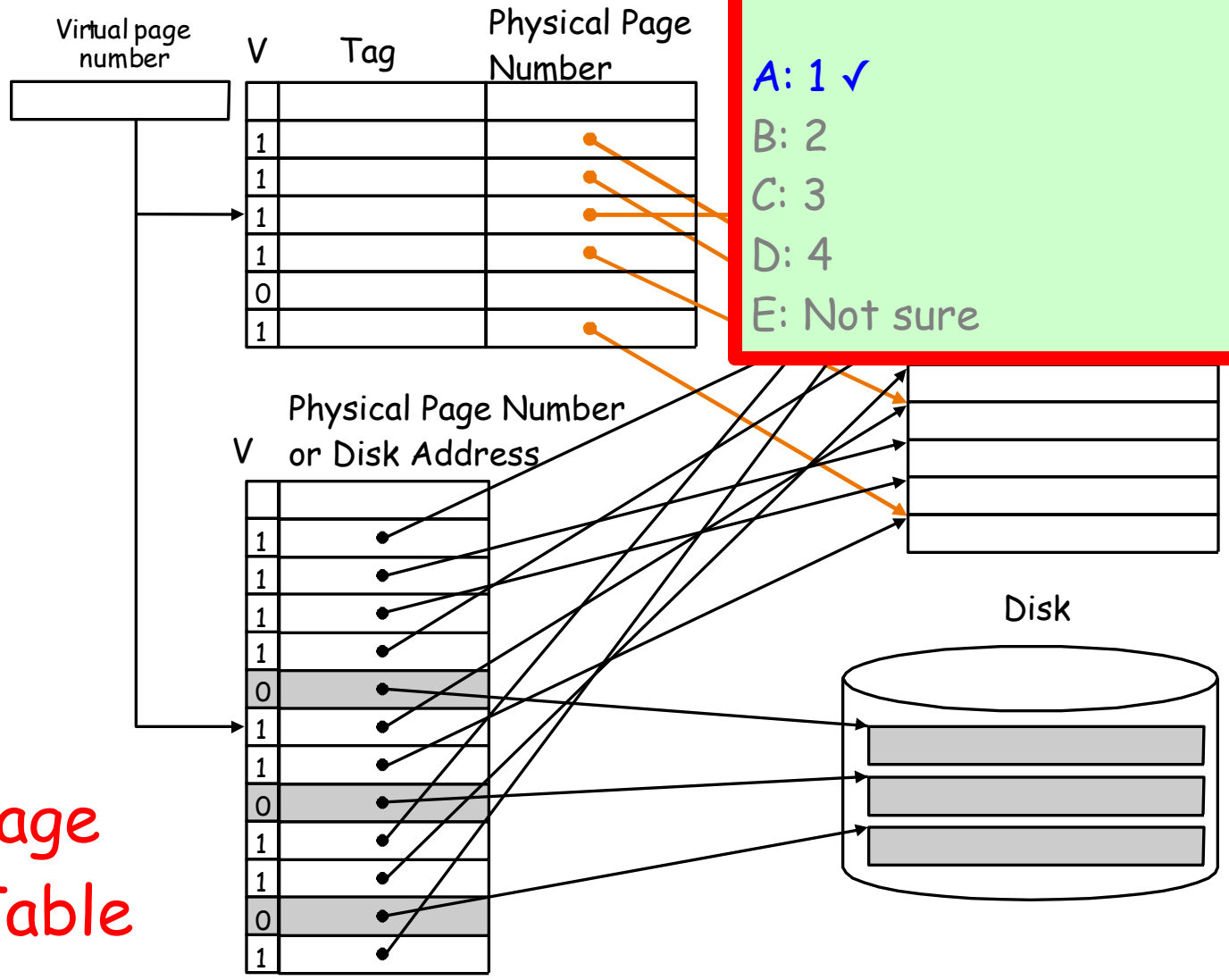
TLB: Translation Looksaside Buffer (A special cache for the Page Table)



TLB: Translation Lo
(A special cache for

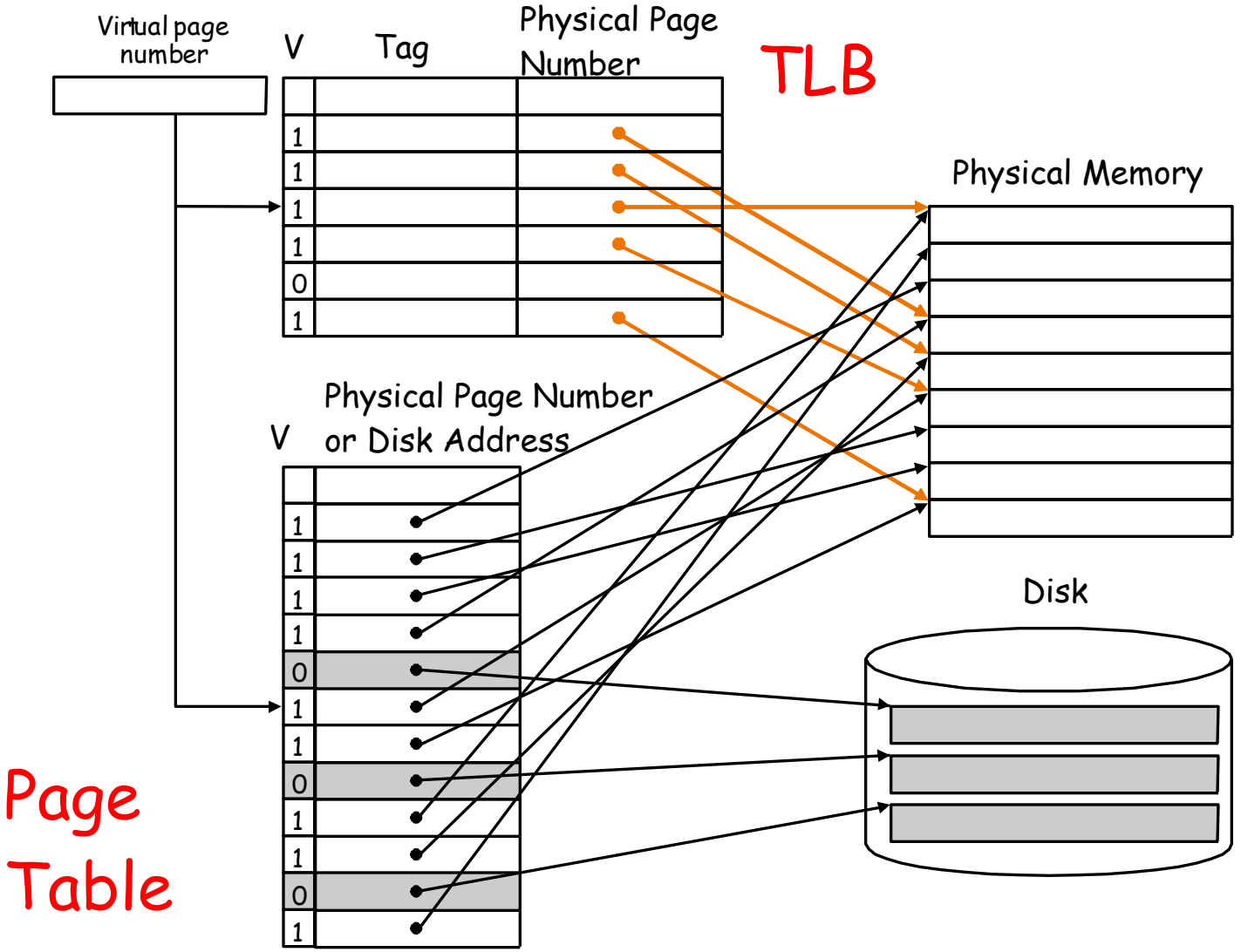
Assuming data we want is NOT in the cache, how many accesses to DRAM when a TLB hit?

A: 1 ✓
B: 2
C: 3
D: 4
E: Not sure

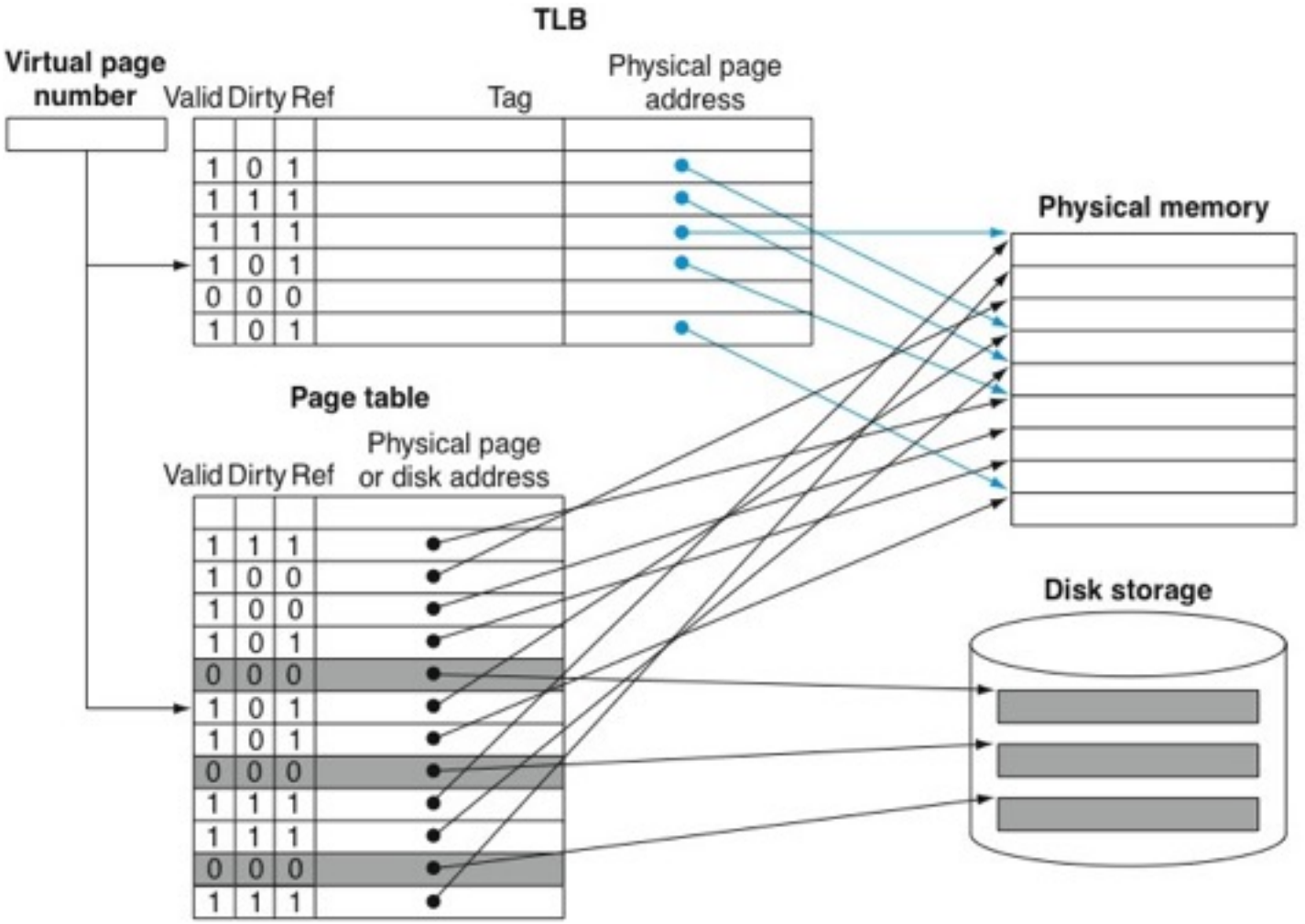


Page Table

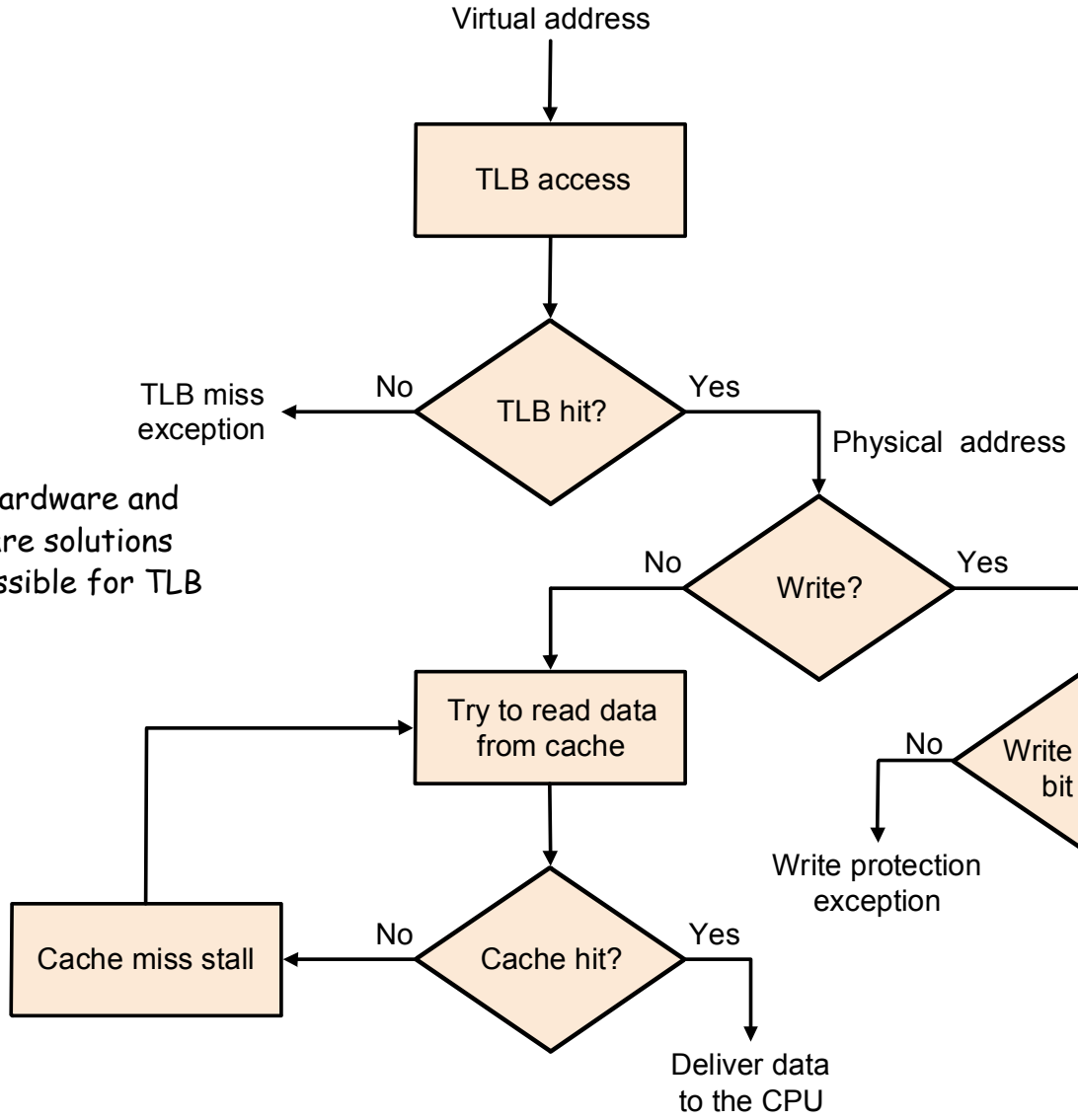
TLB: Translation Looksaside Buffer (A special cache for the Page Table)



What about writes? How to determine what to put on disk?



Memory Access:TLB Usage Algorithm



Both hardware and software solutions are possible for TLB miss

NOTICE: software designates some memory pages as READ-ONLY

Example: pages holding instructions.

It will may share some read-only pages with multiple programs, eg you run Netscape twice.

TLB Notes

- TLB Miss
  - If handled by software: raises exception on TLB miss.
  - If handled by hardware: special hardware unit will check DRAM for translation
  - If page is not in DRAM, then need OS to transfer it in ("page fault").
  - OS handles security issues (invalidating TLB entries, etc…)

- Read-only pages
  - Can share non-sensitive data, eg instructions and pre-compiled data tables in a program such as Firefox
  - Writes to read-only pages cause exception
  - OS handles, possible outcomes:
    - Illegal to write to this page
    - Change page to writable
    - If page is shared, make a new writable copy of this page for this program only. Remaining program share the original read-only copy. This is called "copy on write"

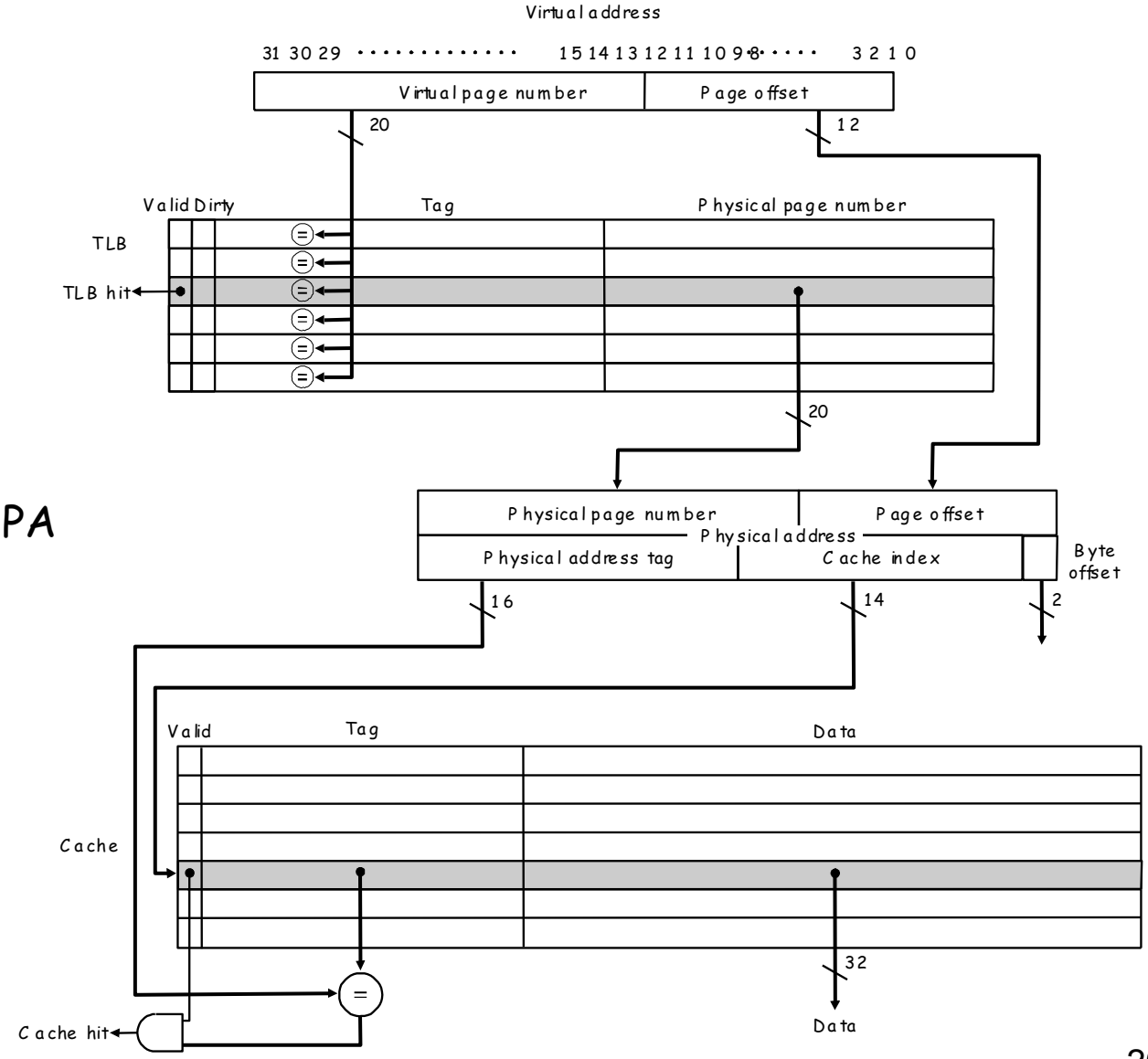
Combined TLB & Cache Structure

1. TLB lookup

2. Convert VA to PA

3. Access data cache

NOTE: TLB is now slowing our cache!!

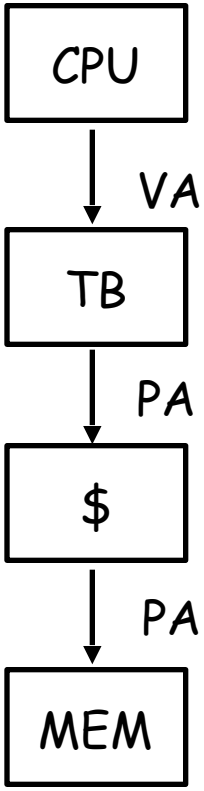


TLB first, Cache second ?

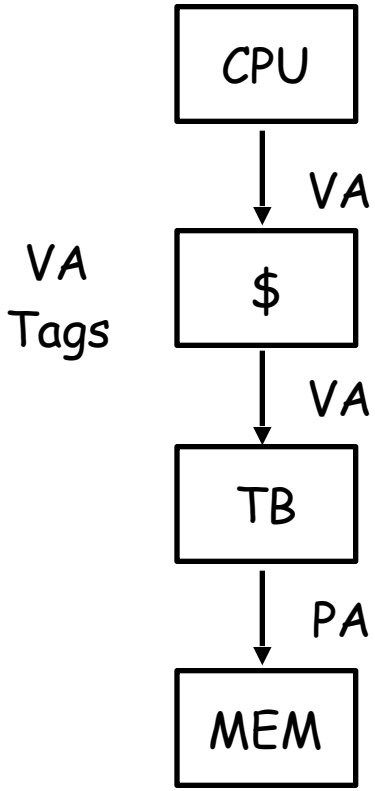
- TLB first, Cache second strategy is slow
  - Called Physically Indexed, Physically Tagged

- To improve performance:
  - Lookup TLB, Lookup Cache at same time
  - Must use Virtual Address to lookup in cache
  - Compare Physical Address (output of TLB) to TAG (output of cache) when checking cache hit
  - If OK, we can use the data
  - Called Virtually Indexed, Physically Tagged

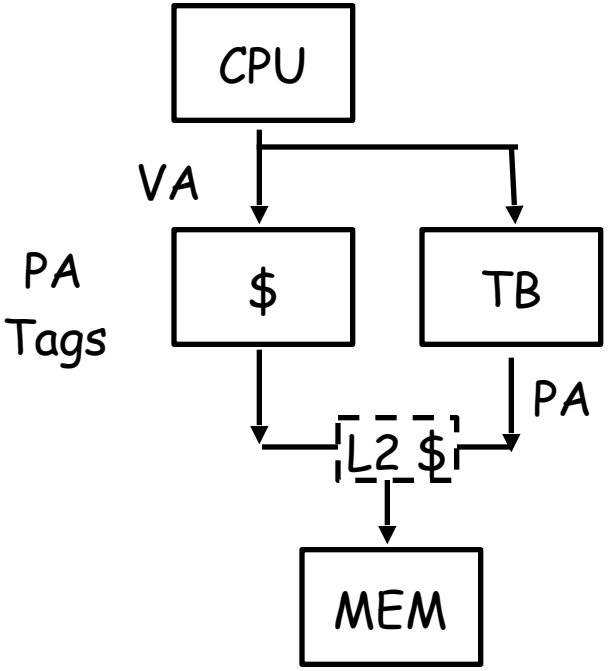
Fast hits by Avoiding Address Translation



Naive Organization

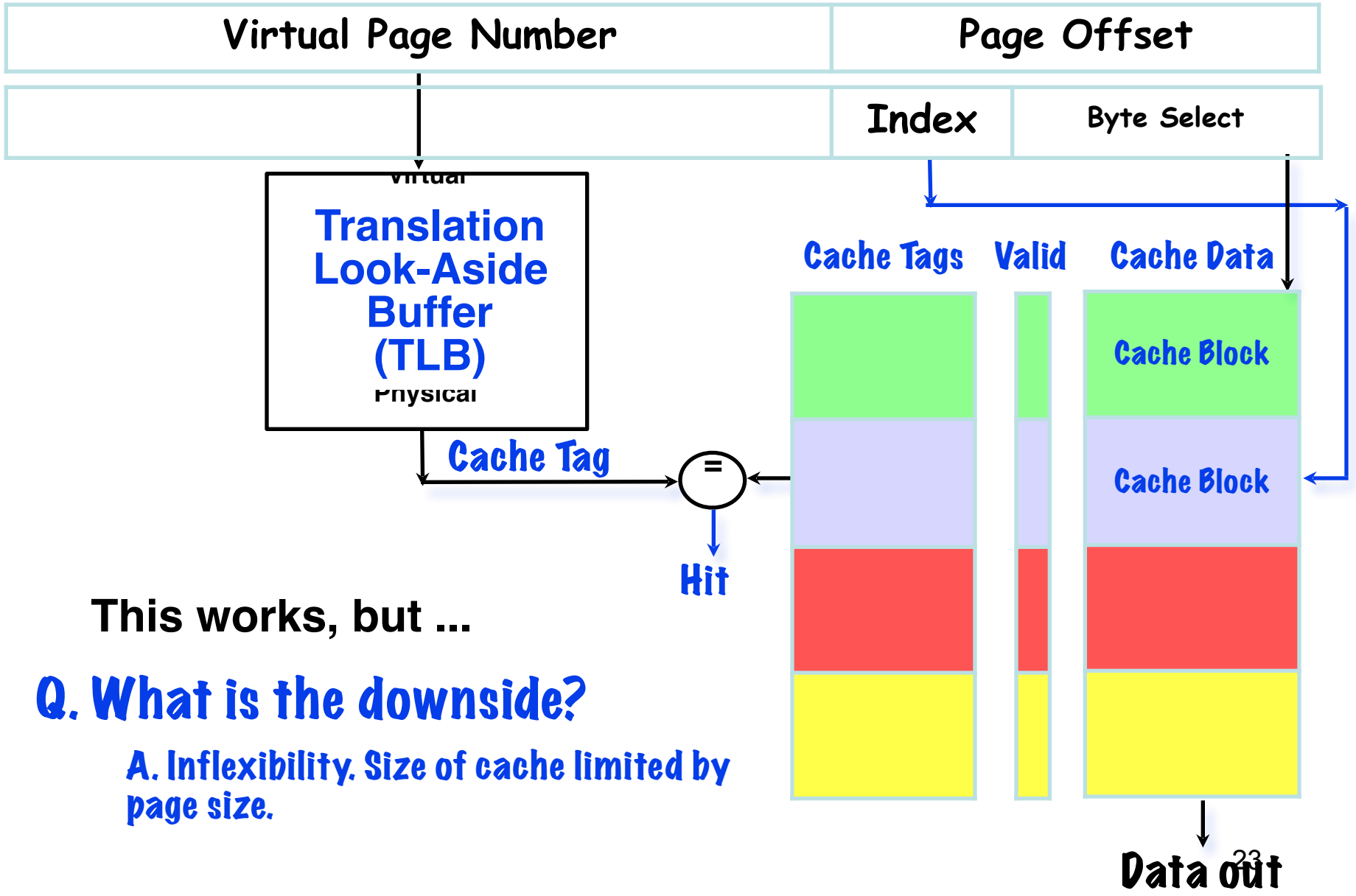


Virtually Addressed Cache?



Overlap $ access with VA translation: requires $ index to remain invariant across translation

Can TLB and caching be overlapped?



This works, but ...

Q. What is the downside?

A. Inflexibility. Size of cache limited by page size.

- The restriction that index bits should not change during address translation limits us to small caches, large page sizes, or high n-way set associative caches if you want a large cache.

- But: Really, why do we need this restriction?

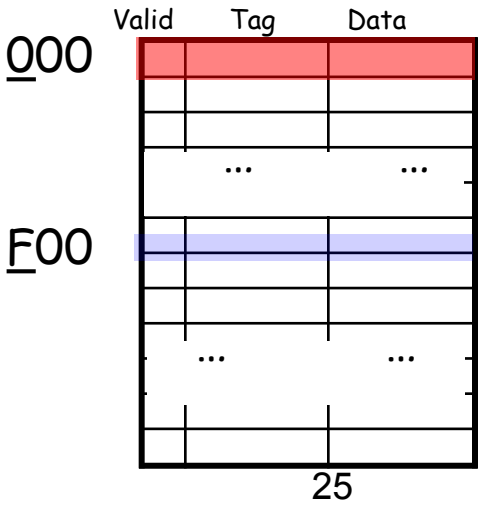
Example of "Synonym Problem" (Virtual indexed Physically Tagged Caches)

- Assume: 64 KB direct mapped cache with 16 B lines
  - block offset lower 4 bits (3...0)
  - cache index next 12 bits (15...4)
- Assume: 4KB pages
  - page offset lower 12 bits (11...0)
- 16 'synonym' locations in cache due to 4 virtual address bits (15...12)

Prog. 1 VA 0x0000 000 0 -> PA 0xFF000000
Prog. 2 VA 0x0000 F00 0 -> PA 0xFF000000

Index Prog. 1 = 0x000 (tag = 0xFF000)
Index Prog. 2 = 0xF00 (tag = 0xFF000)

What if Prog. 1 writes, yield, Prog. 2 reads?



Example of "Synon
(Virtual indexed Physical

- Assume: 64 KB direct mapped cache v
  - block offset lower 4 bits (3...0)
  - cache index next 12 bits (15...4)
- Assume: 4KB pages
  - page offset lower 12 bits (11...0)
- 16 'synonym' locations in cache due to

Prog. 1 VA 0x0000 000 0 -> PA
Prog. 2 VA 0x0000 F00 0 -> PA

Index Prog. 1 = 0x000 (tag = 0xF
Index Prog. 2 = 0xF00 (tag = 0xF

What if Prog. 1 writes, yield, Prog

Which of the following mechanisms can ensure correct operation in spite of the synonym problem ?

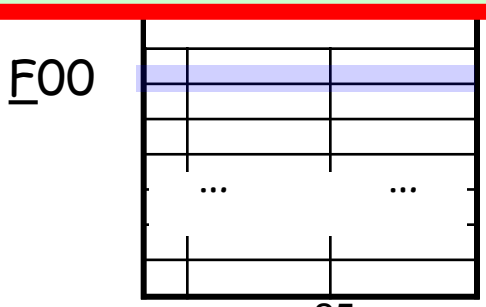
A: Search all synonym locations every access to the cache

B: Search all synonym locations only on a cache miss

C: Ensure pages allocated in physical memory are constrained so that bits 15 to 12 match virtual address.

D: All of the above

E: Not sure



Example of "Synon
(Virtual indexed Physical

- Assume: 64 KB direct mapped cache v
  - block offset lower 4 bits (3...0)
  - cache index next 12 bits (15...4)
- Assume: 4KB pages
  - page offset lower 12 bits (11...0)
- 16 'synonym' locations in cache due to

Prog. 1 VA 0x0000 000 0 -> PA
Prog. 2 VA 0x0000 F00 0 -> PA

Index Prog. 1 = 0x000 (tag = 0xF
Index Prog. 2 = 0xF00 (tag = 0xF

What if Prog. 1 writes, yield, Prog

Which of the following mechanisms can ensure correct operation in spite of the synonym problem ?

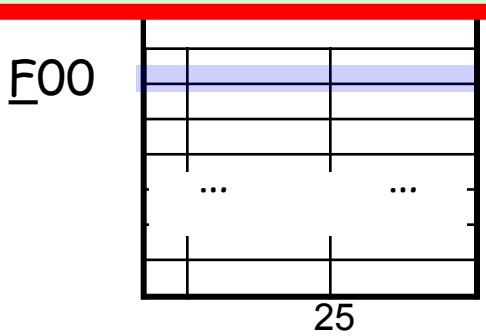
A: Search all synonym locations every access to the cache

B: Search all synonym locations only on a cache miss

C: Ensure pages allocated in physical memory are constrained so that bits 15 to 12 match virtual address.

D: All of the above ✓

E: Not sure



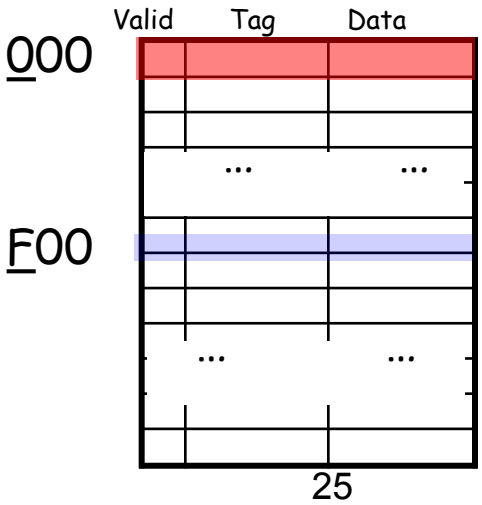
Example of "Synonym Problem" (Virtual indexed Physically Tagged Caches)

- Assume: 64 KB direct mapped cache with 16 B lines
  - block offset lower 4 bits (3...0)
  - cache index next 12 bits (15...4)
- Assume: 4KB pages
  - page offset lower 12 bits (11...0)
- 16 'synonym' locations in cache due to 4 virtual address bits (15...12)

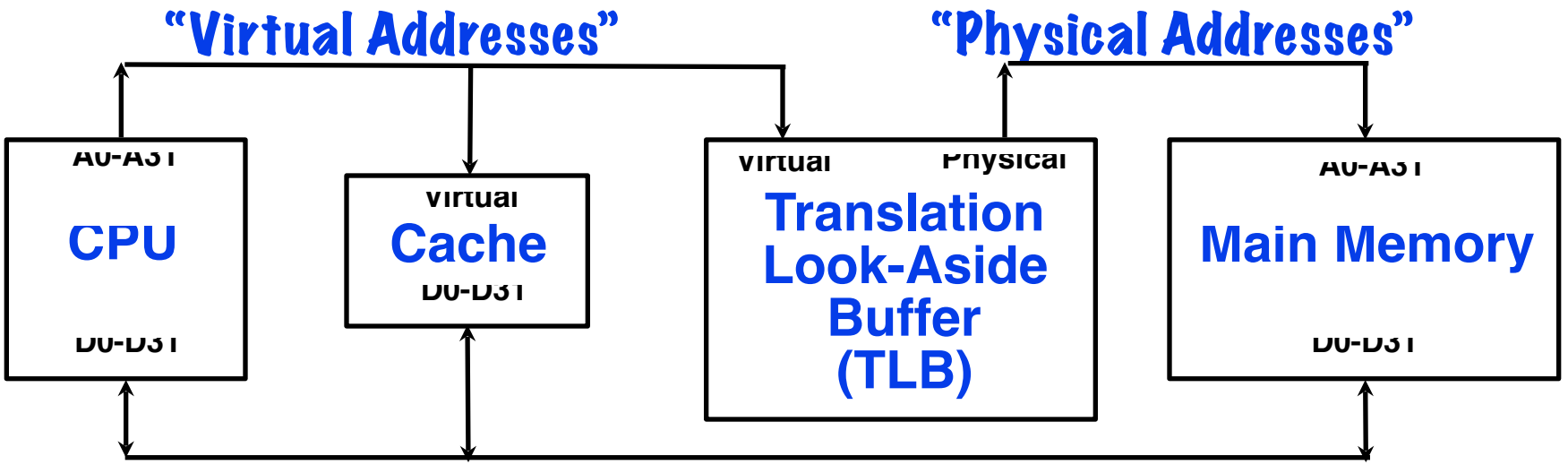
Prog. 1 VA 0x0000 000 0 -> PA 0xFF000000
Prog. 2 VA 0x0000 F00 0 -> PA 0xFF000000

Index Prog. 1 = 0x000 (tag = 0xFF000)
Index Prog. 2 = 0xF00 (tag = 0xFF000)

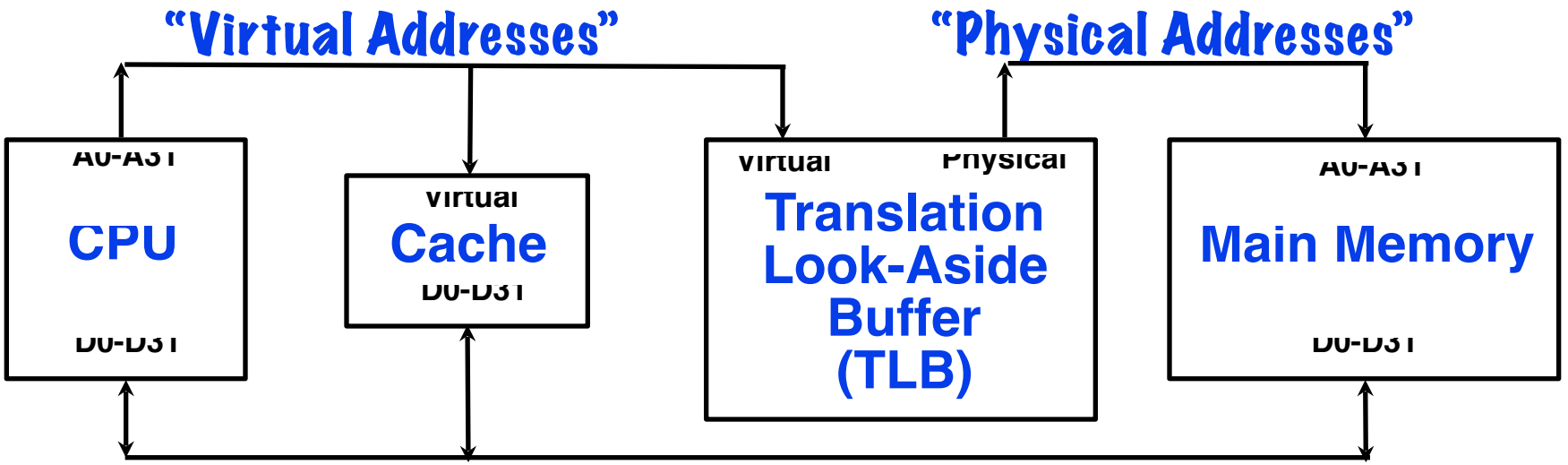
What if Prog. 1 writes, yield, Prog. 2 reads?



Use virtual addresses for cache?

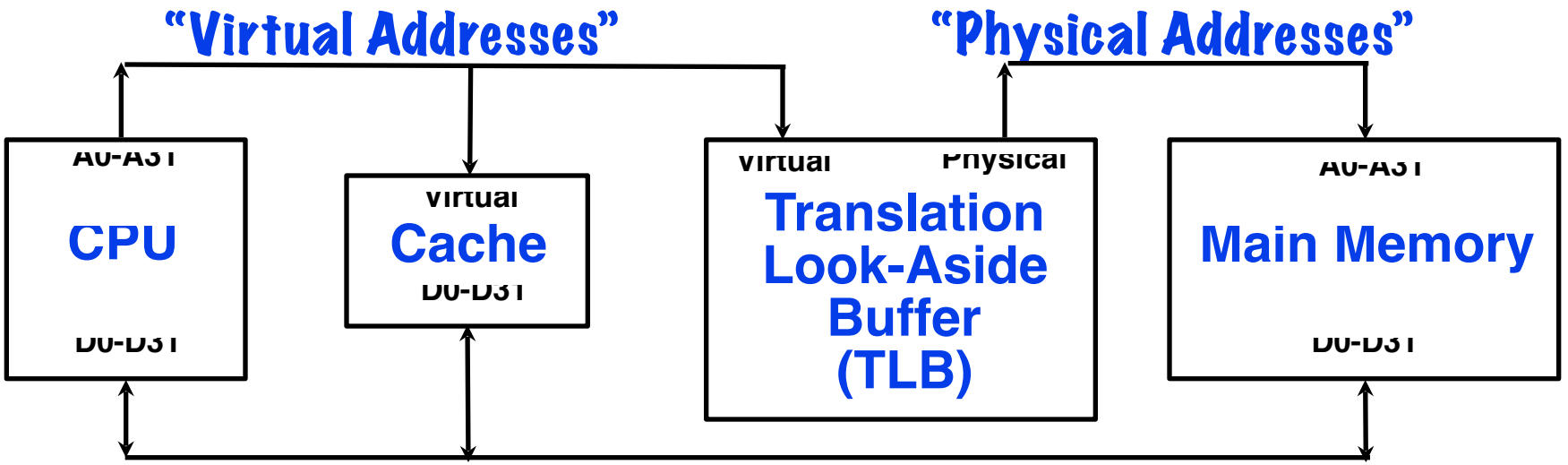


Use virtual addresses for cache?



Only use TLB on a cache miss !

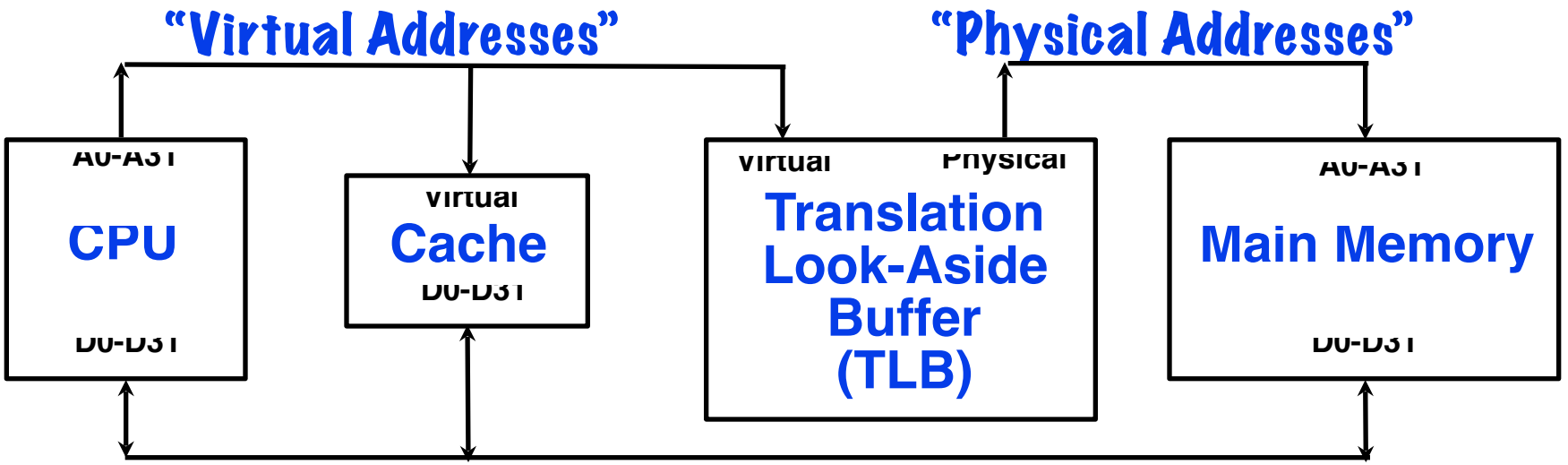
Use virtual addresses for cache?



Only use TLB on a cache miss !

Downside: a fatal problem. What is it?

Use virtual addresses for cache?

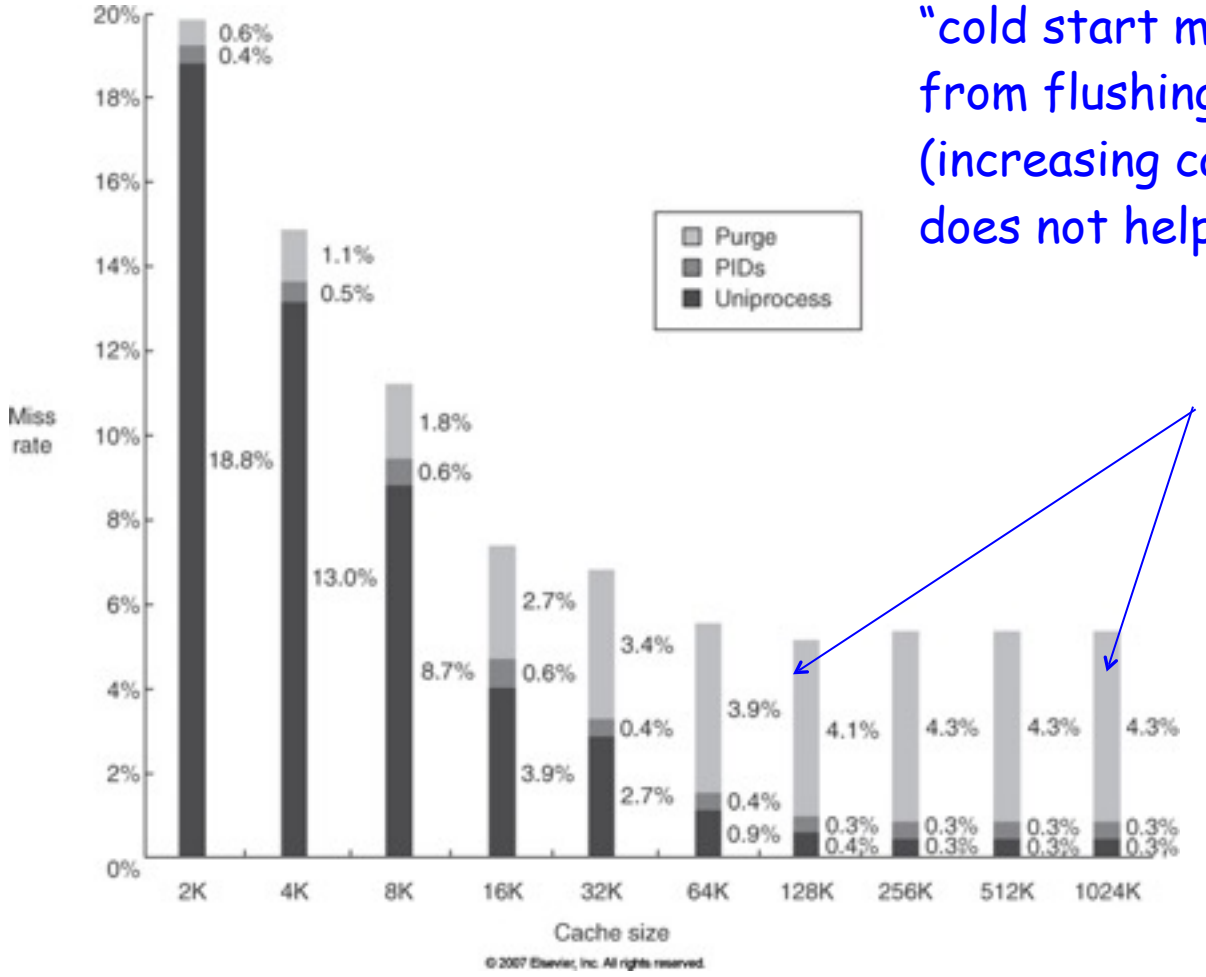


Only use TLB on a cache miss !

Downside: a fatal problem. What is it?

A. Two processes might use same virtual addresses to refer to different (physical) data; need to flush cache on context switch.

Why not just use virtual addresses for cache?



"cold start misses" resulting from flushing cache (increasing cache capacity does not help)

Substantial overhead to flush cache on context switch (dirty lines need to be written to memory if using writeback cache)