CS162
Operating Systems and Systems Programming
Lecture 15

Virtual Memory (2)

Professor Natacha Crooks

https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Recall: Memory Management Wishlist

Memory Protection

Memory Sharing

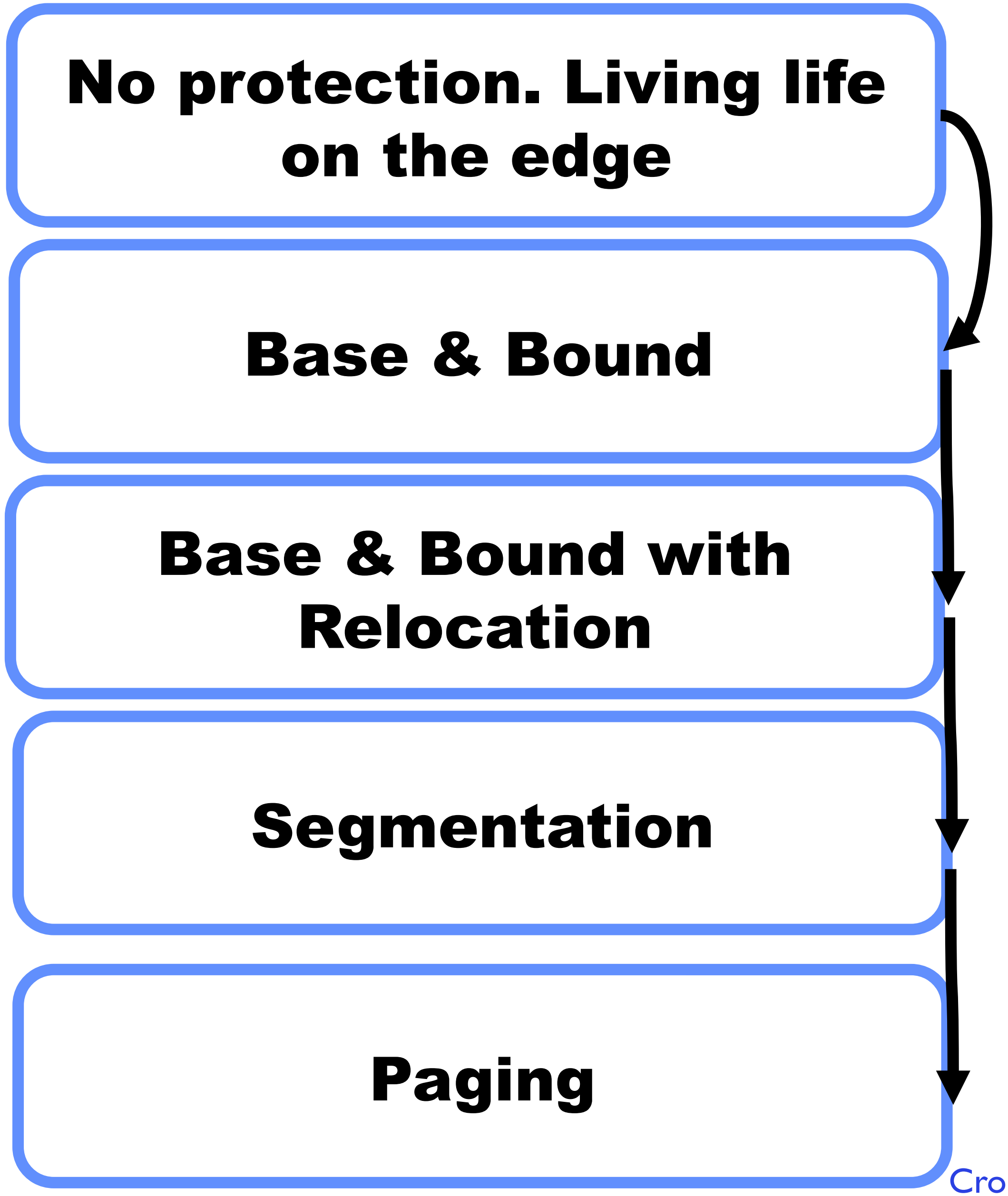
Flexible Memory Placement

Support for Sparse Addresses

Runtime Lookup Efficiency

Compact Translation Table

Recall: Increasingly powerful mechanisms



No protection. Living life on the edge

Base & Bound

Base & Bound with Relocation

Segmentation

Paging

Can access all memory

Absolute memory addressing. Hard to relocate

Internal fragmentation when address space is sparse

External fragmentation as assigning variably sized chunks

Paging

Divide logical address space of process into fixed sized chunks called pages

View physical memory as an array of fixed-sized slots called **page frames**

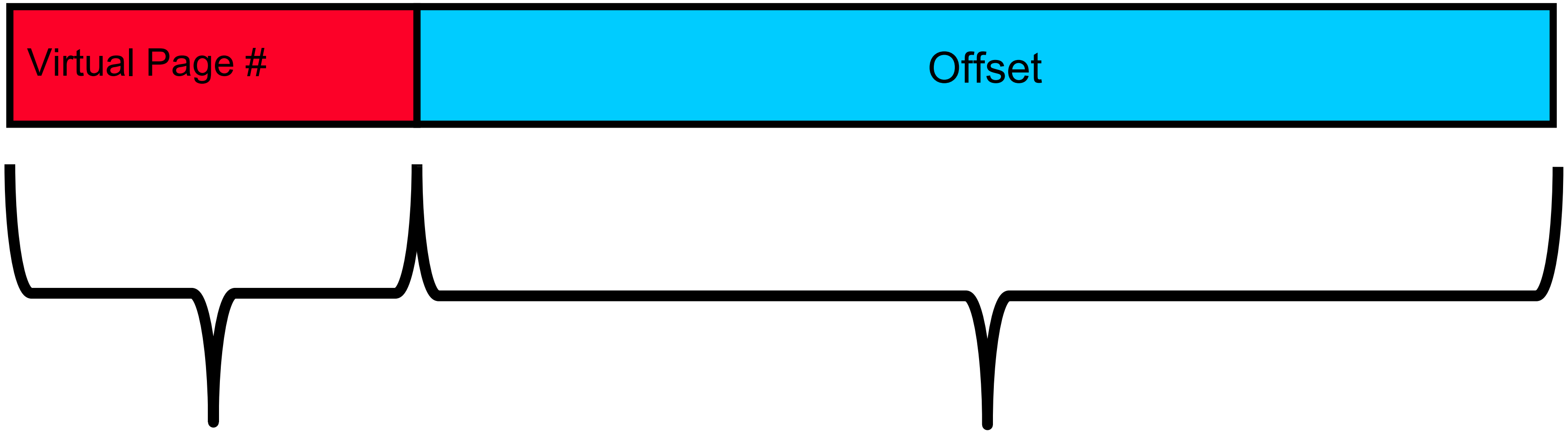
Each page frame can contain a

single virtual-memory page

Pages should be small to minimise internal fragmentation (1K-16k)

How to Implement Simple Paging?

Interpret virtual address as two components

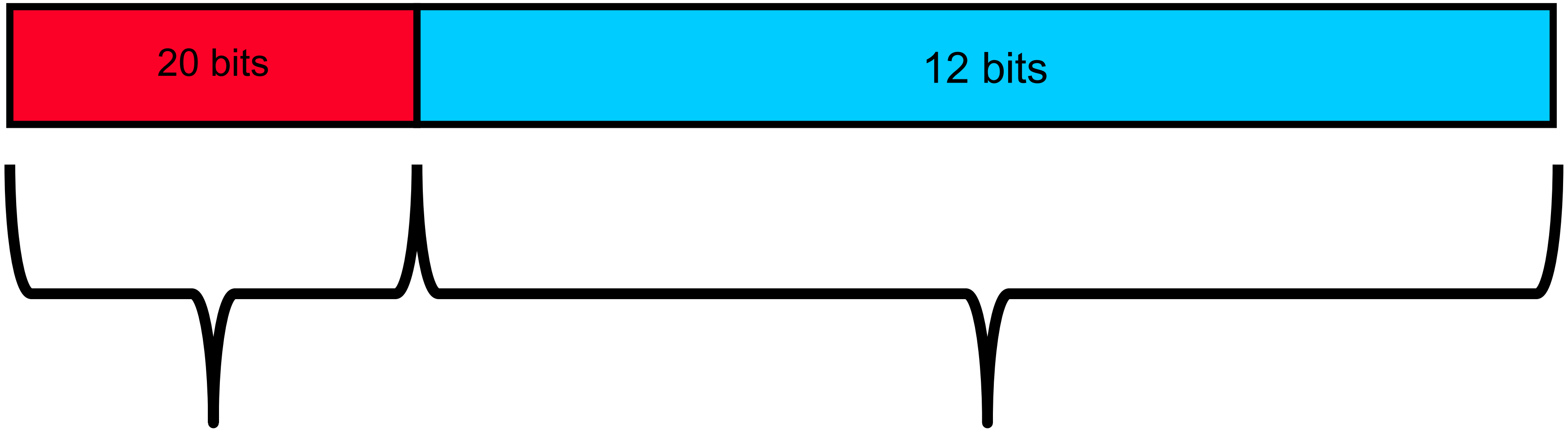


no. of bits specifies no. of pages in VA space

no. of bits specifies size of page

How to Implement Simple Paging?

Interpret virtual address as two components



20 bits

12 bits

2^20 pages

Page Size
2^12 = 4096 B = 4KB

A (Simplified) Page Table

A page table stores
virtual-to-physical address translations

One page table per process. Lives in memory.

Address stored in the in the Page Table Base Register

PTBR value saved/restored in PCB on context switch

How to access a byte?

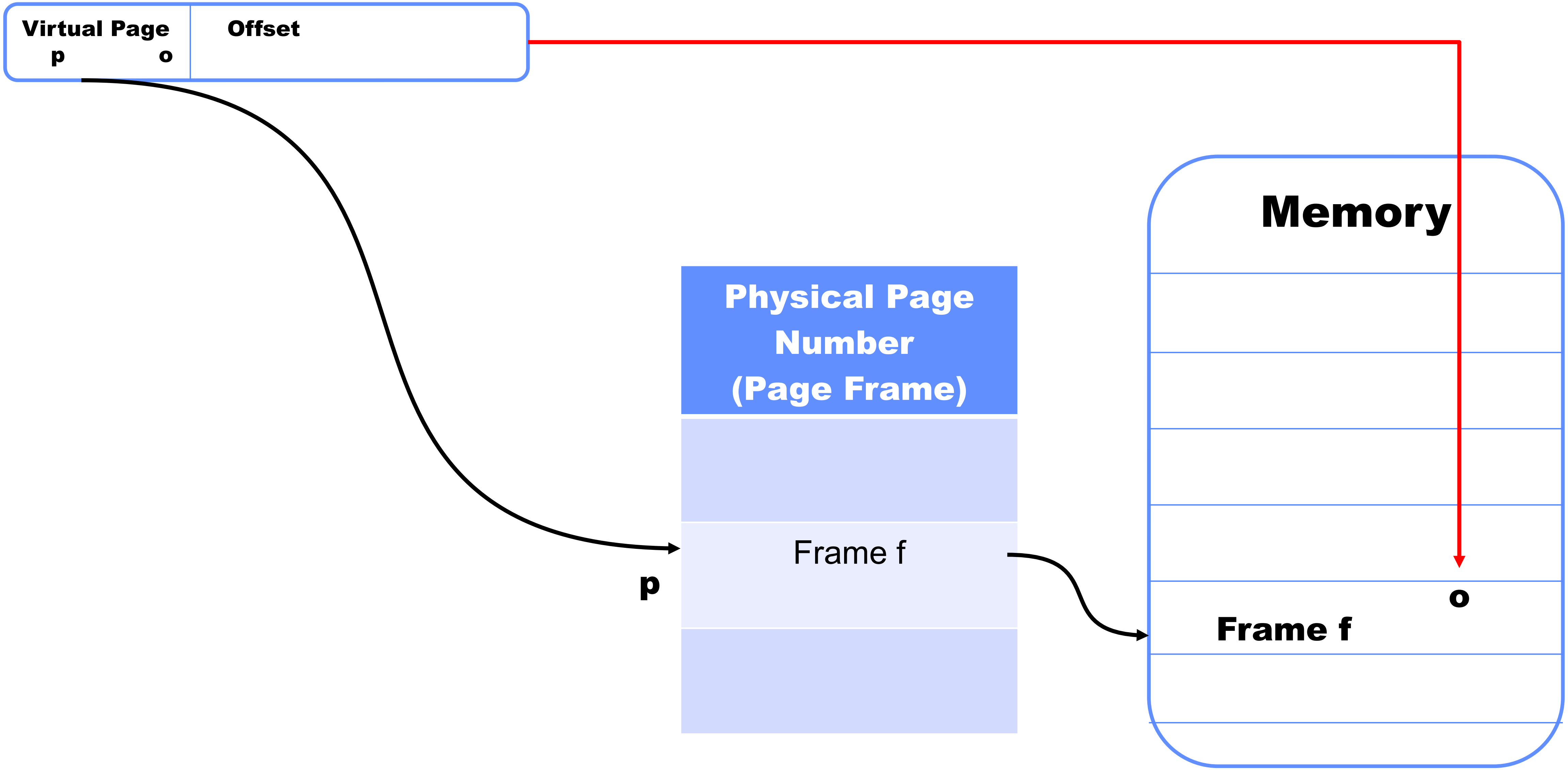
Extract page number (first p bits)

Map virtual page number into a frame number
(also called physical page number) using a page table

Extract offset (last o bits)

Convert to physical memory location: access byte at offset in frame

A (Simplified) Page Table



Example: A Mini Page Table

Assume we have a 64 bytes (2^6) of physical memory

Assume we want pages of 4 bytes (2^2)

How long should our addresses be?

6 bits

How many offset bits should we assign?

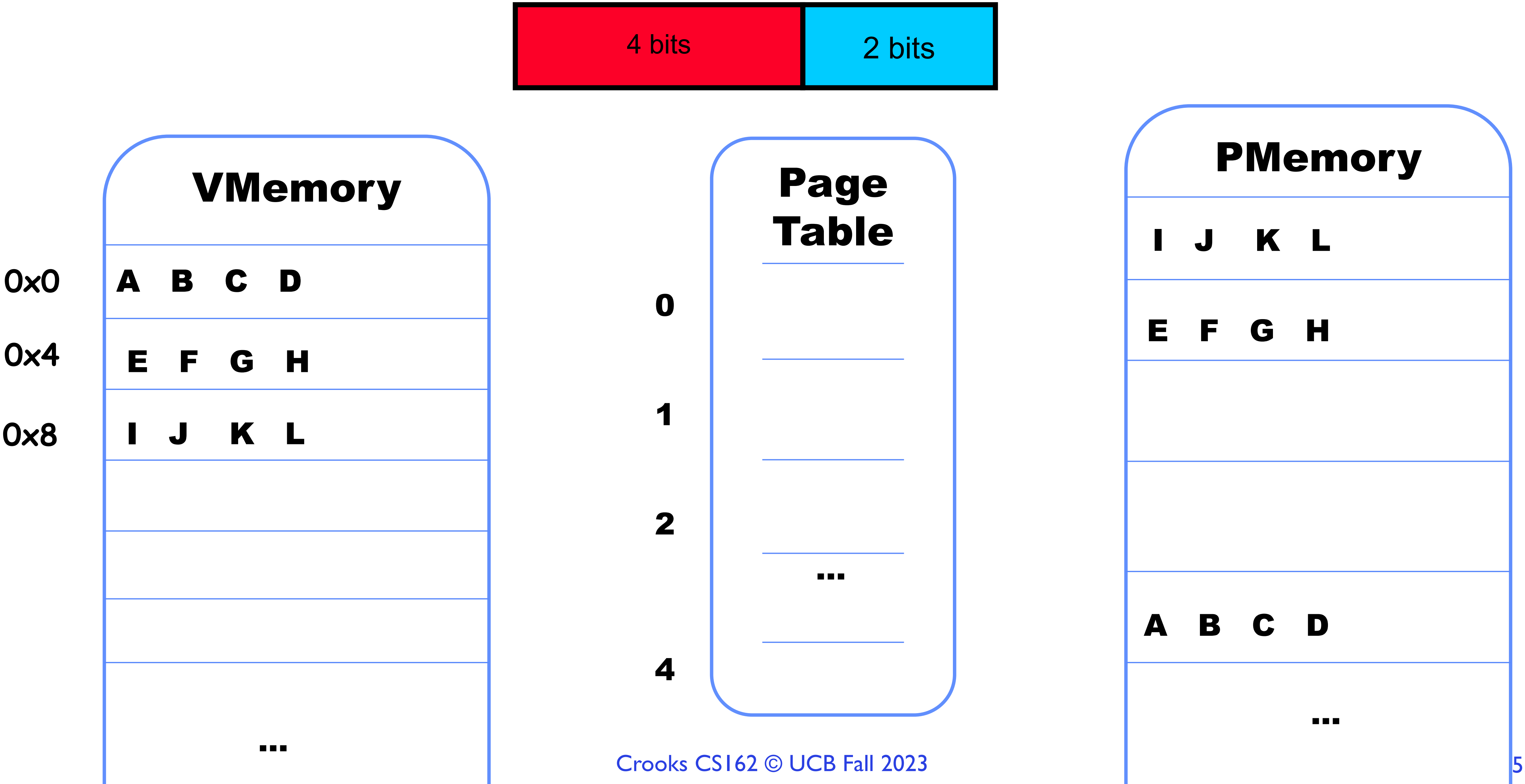
2 bits

How many virtual pages can we have?

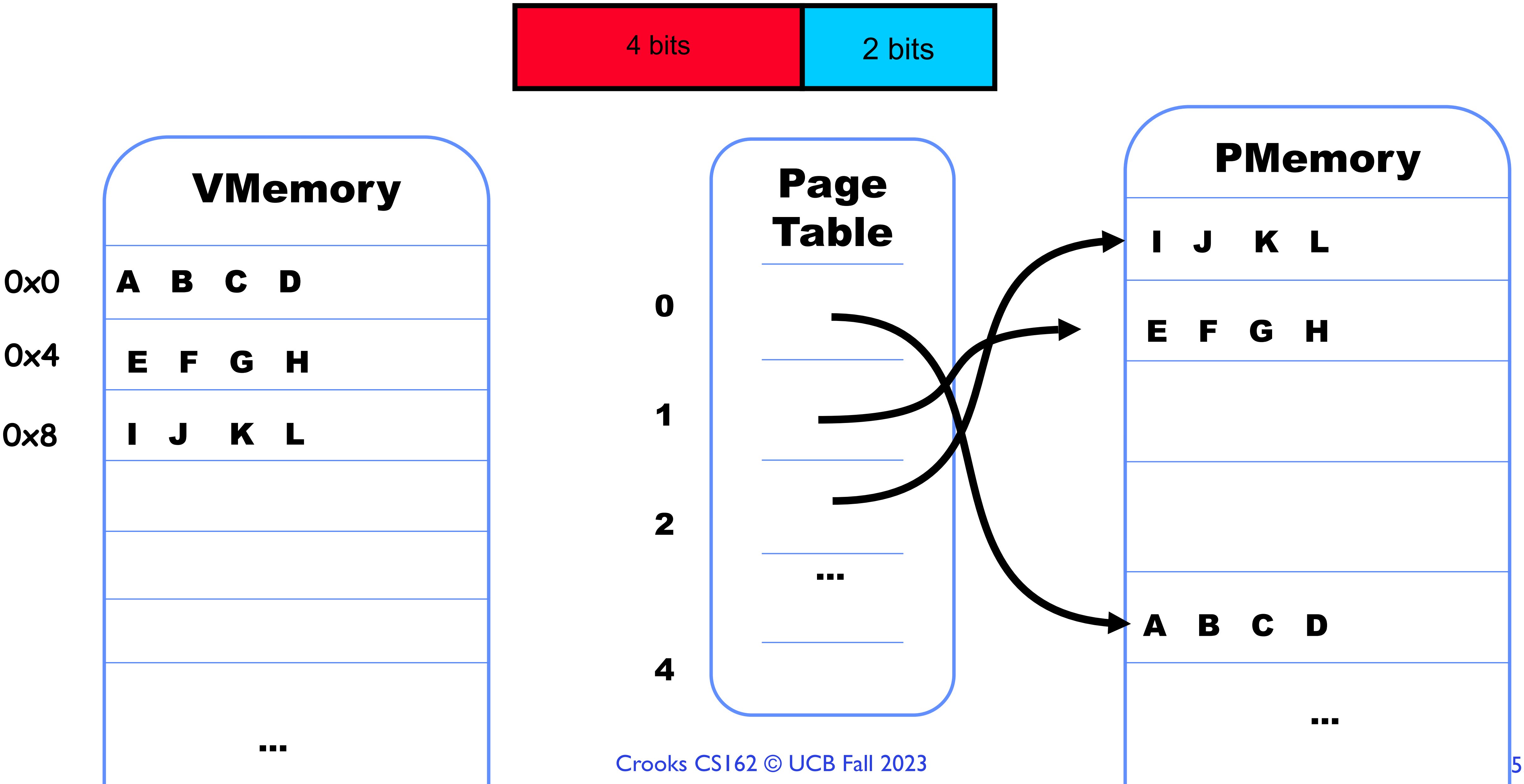
6 bit addresses, 2 bit for offsets, 4 bits for VPN.

2^4 = 16 pages

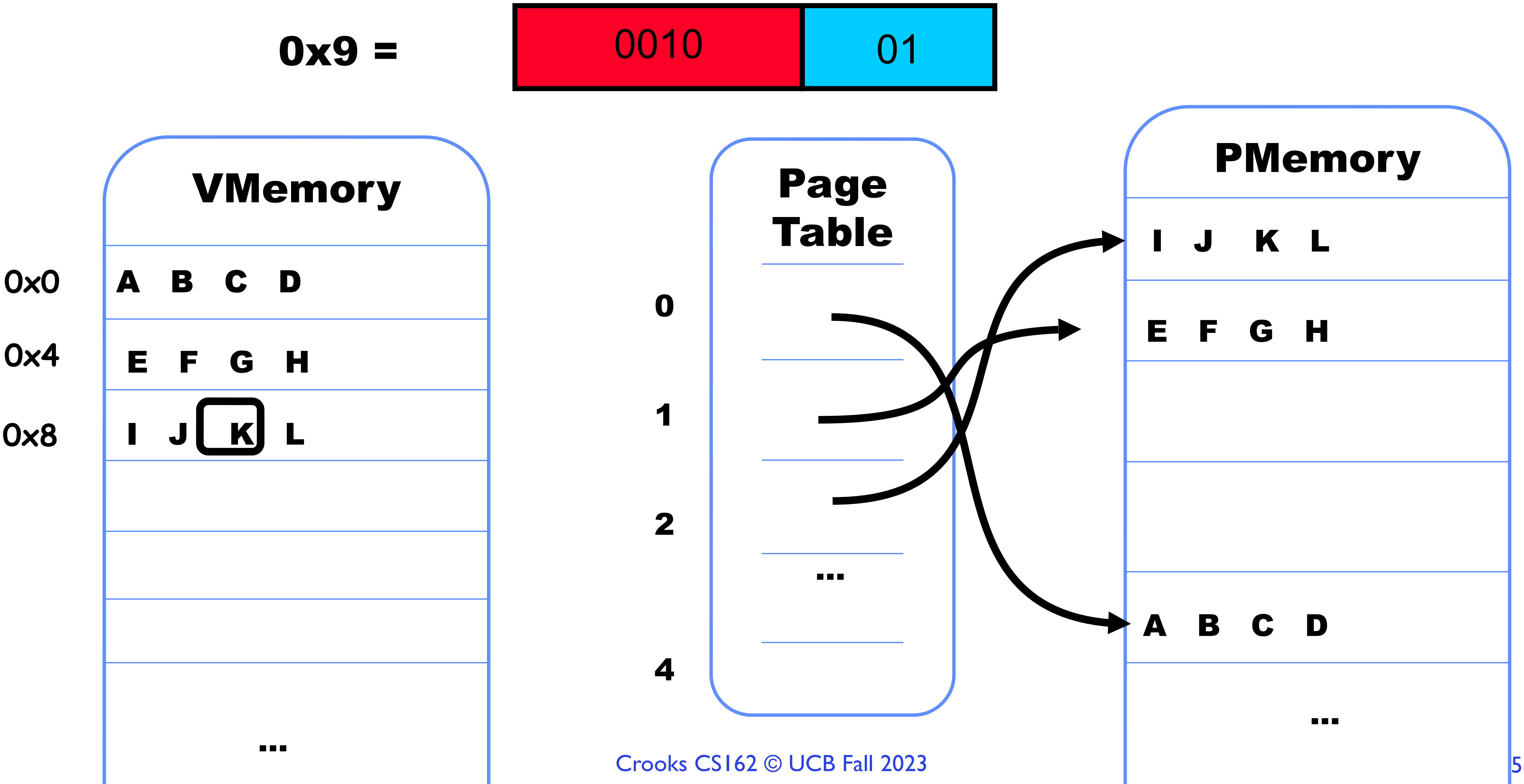
Example: A Mini Page Table



Example: A Mini Page Table

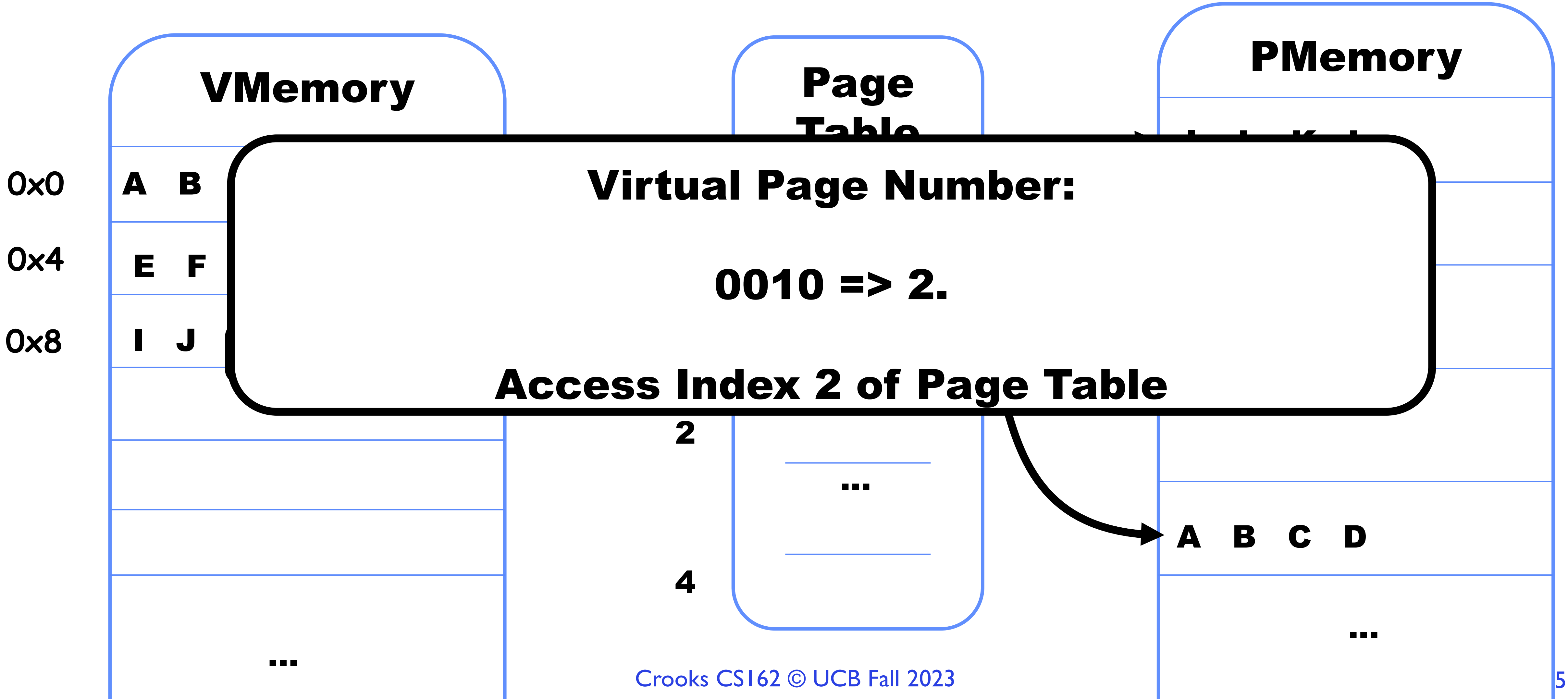


Example: A Mini Page Table



Step 1: Extract Virtual Page Number

0x9 = | 0010 | 01 |

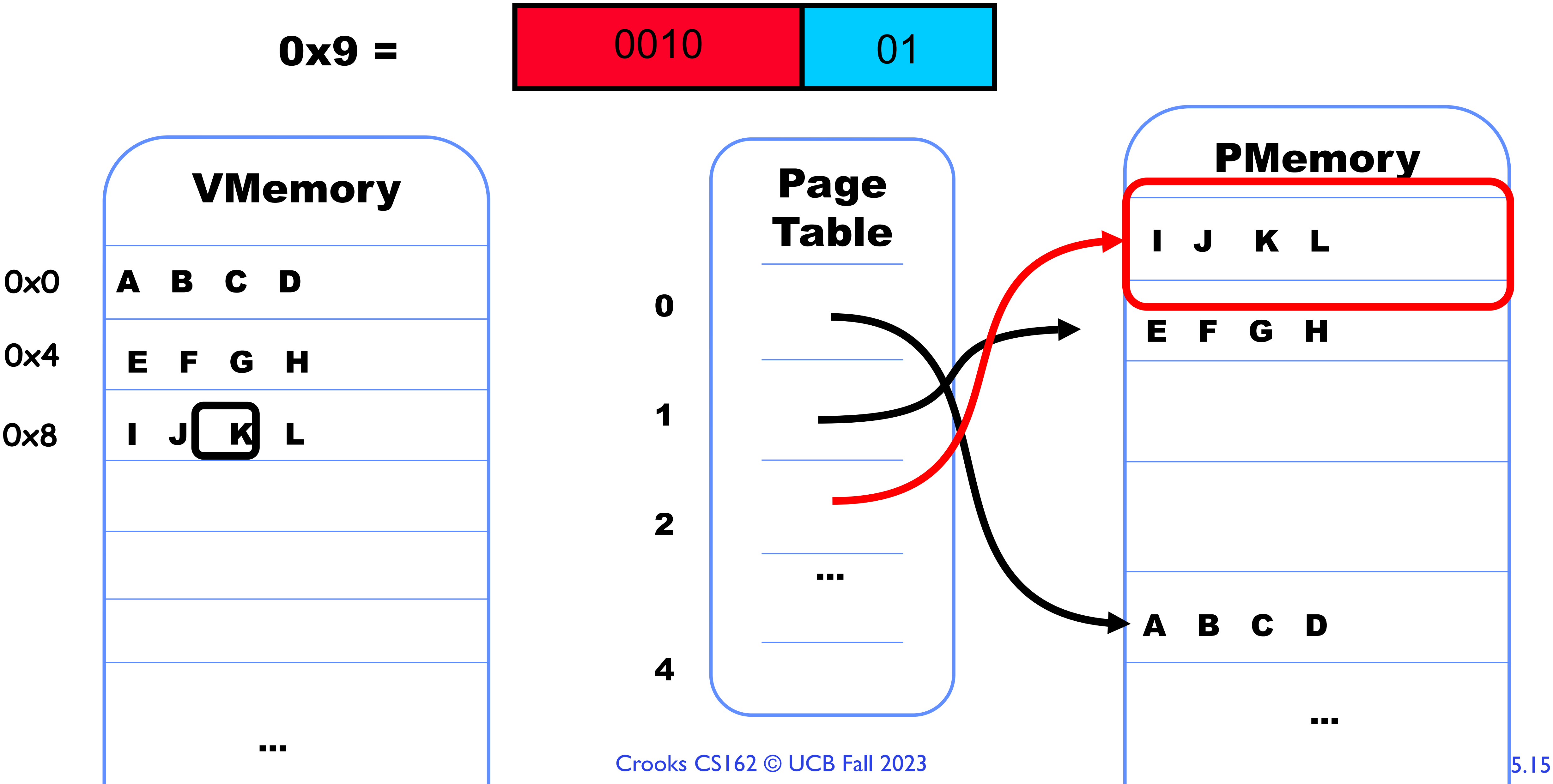


Virtual Page Number:

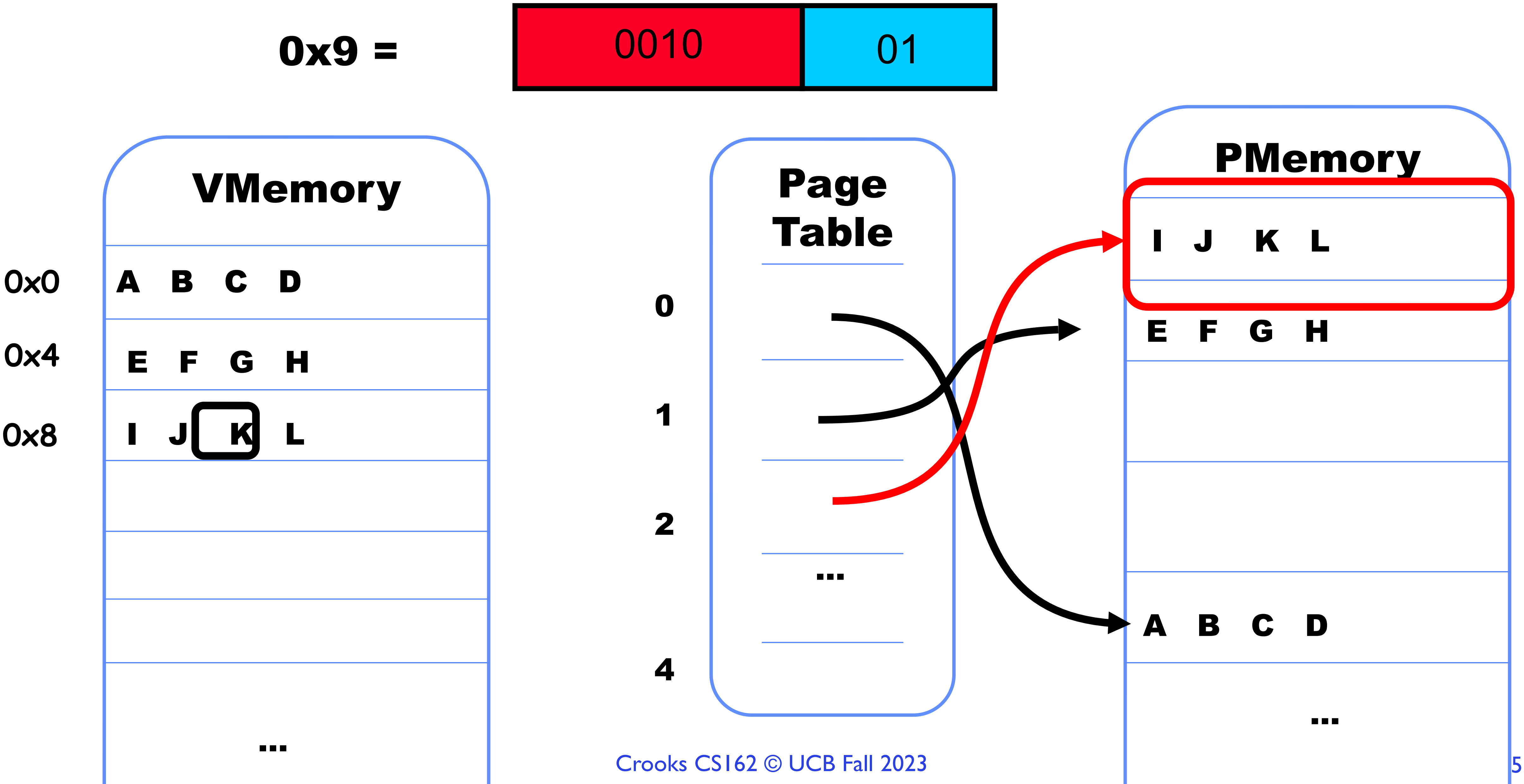
0010 => 2.

Access Index 2 of Page Table

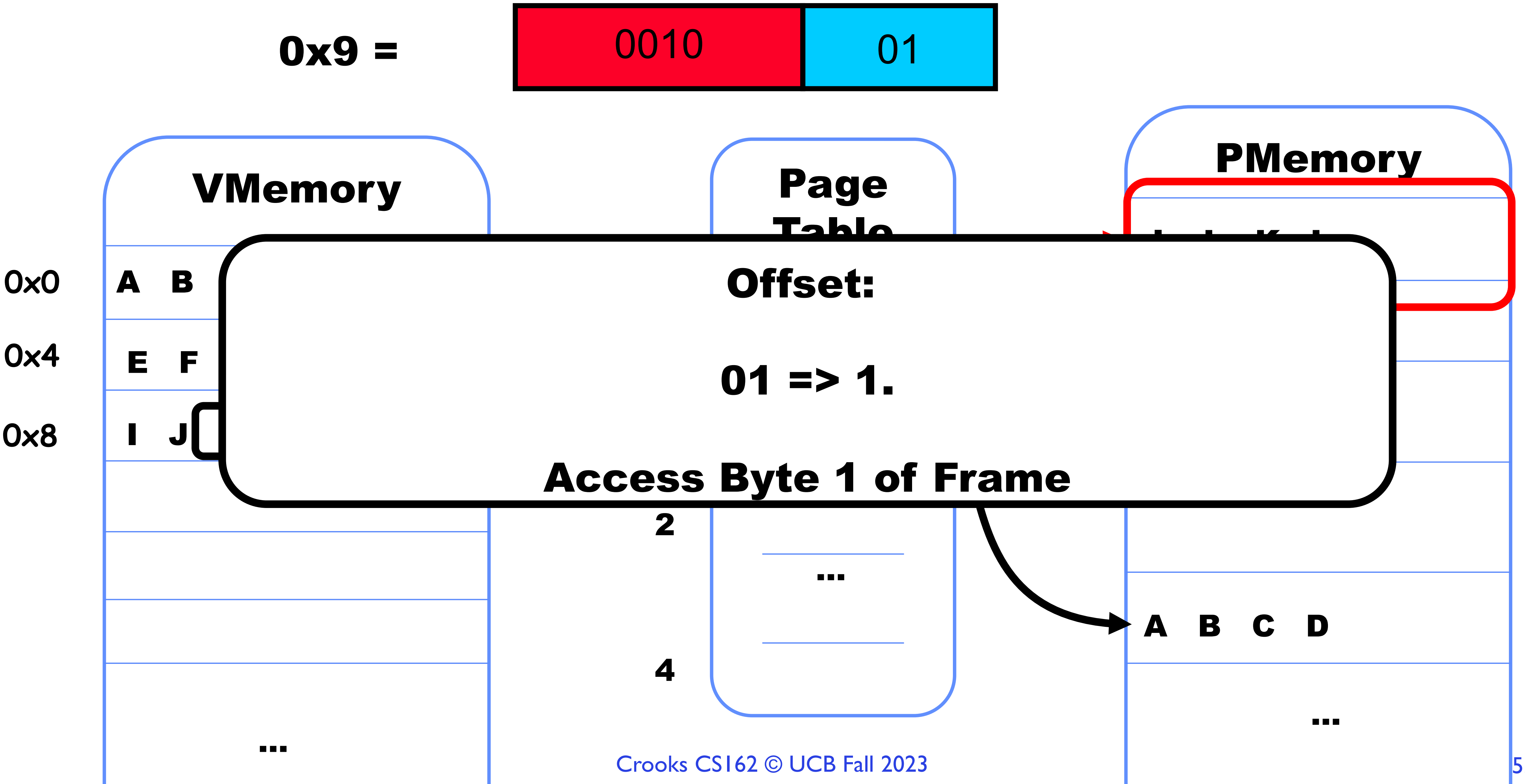
Step 2: Identify Physical Page Number



Step 3: Extract Frame Offset



Step 3: Extract Frame Offset

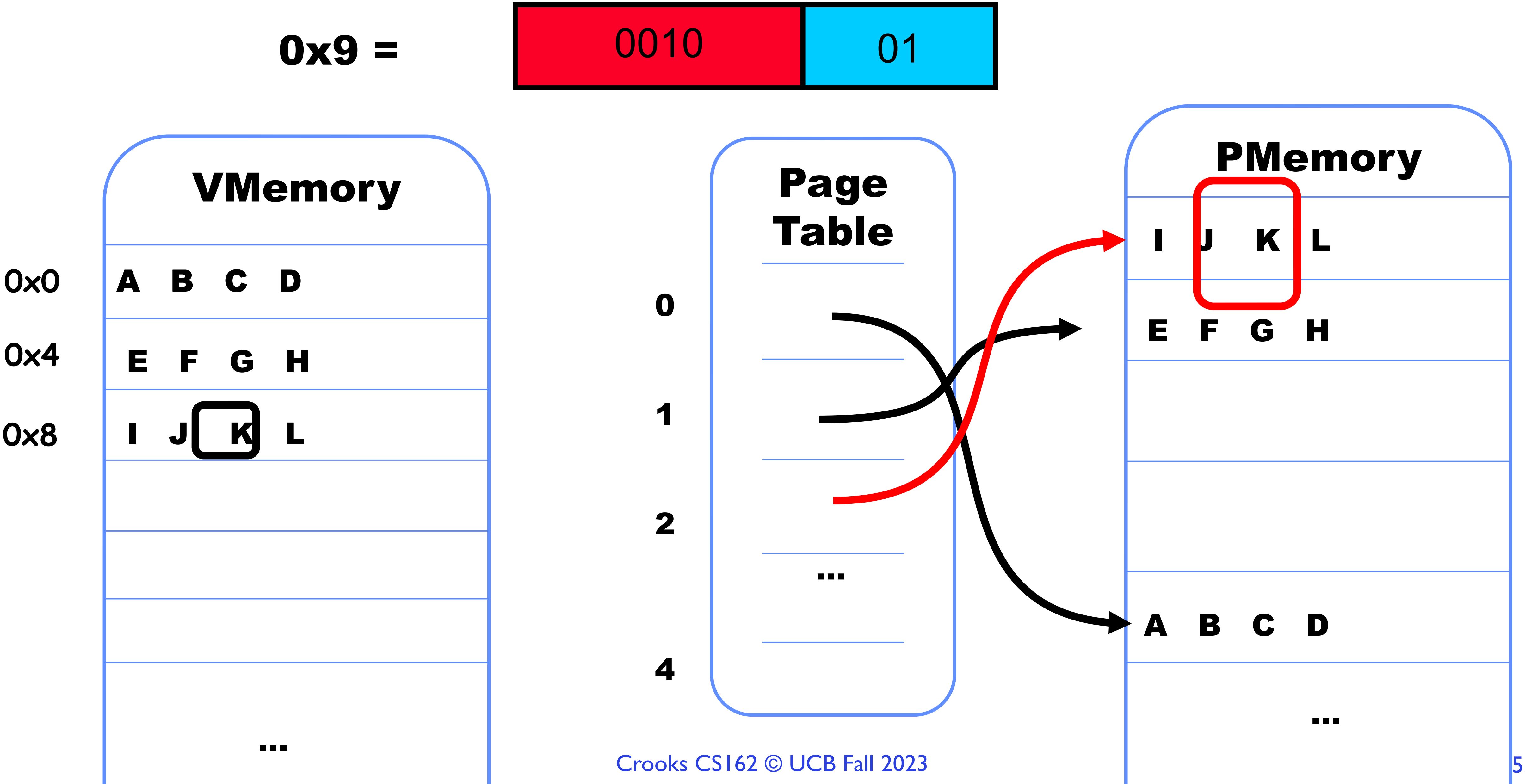


Offset:

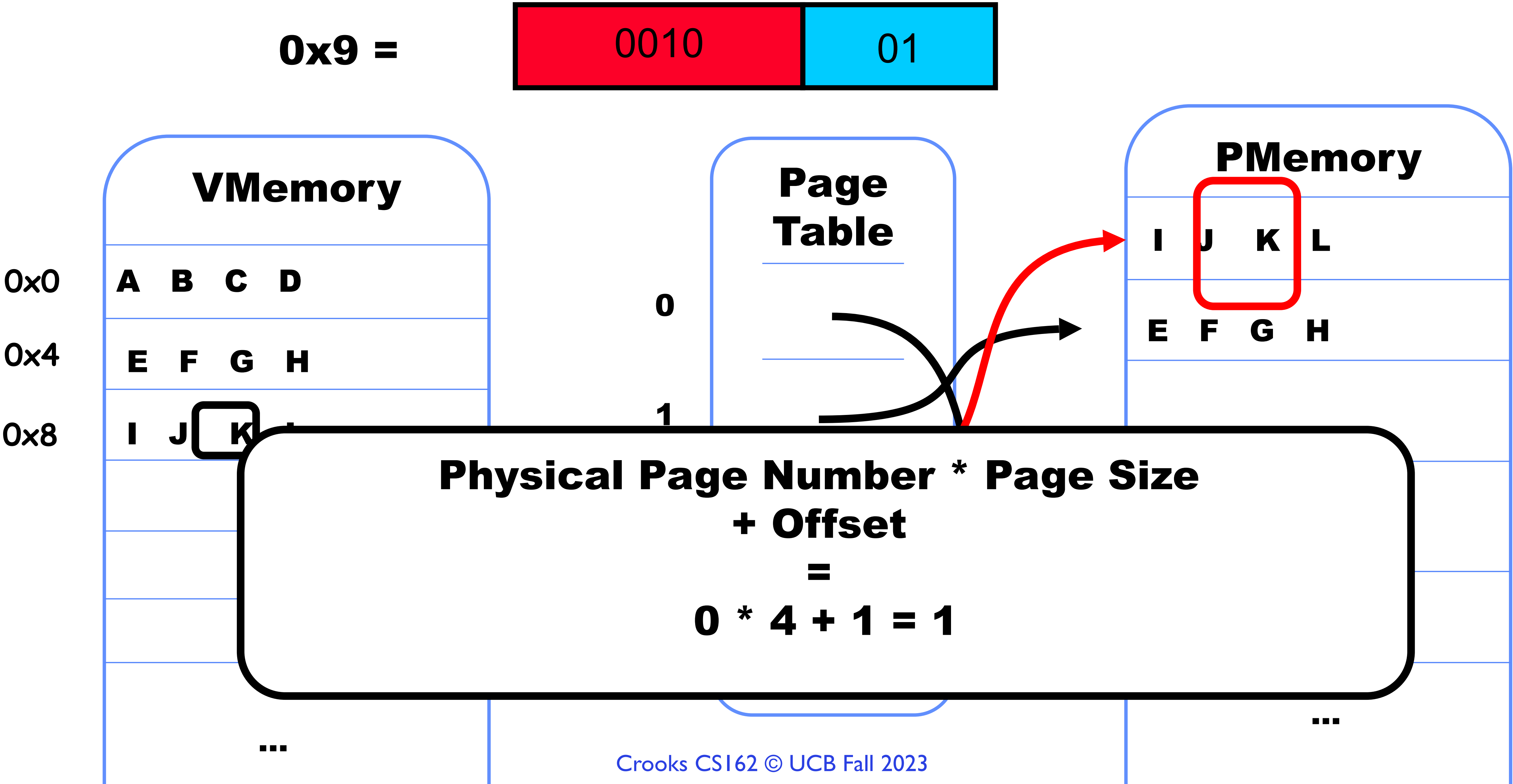
01 => 1.

Access Byte 1 of Frame

Step 3: Extract Frame Offset



Step 4: Convert to Physical Address



Physical Page Number * Page Size + Offset

=

0 * 4 + 1 = 1

What is a page table entry? (32 bits)

Page Frame Number (Physical Page Number)	Free (OS)	0	PS	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

P: Present (same as "valid" bit in other architectures)

W: Writeable

U: User accessible

PWT: Page write transparent: external cache write-through

PCD: Page cache disabled (page cannot be cached)

A: Accessed: page has been accessed recently

D: Dirty: page has been modified recently

PS: Page Size

Size of page table entry: PFN (20 bits) + 12 bits for access control/caching

4 bytes

The Great Power of the PTE

Demand Paging

Keep only active pages in memory
Place others on disk and mark their PTEs invalid

Copy-on-Write

UNIX fork gives *copy* of parent address space to child. Use combination of page sharing + marking pages as read-only

Zero Fill On Demand

New data pages must carry no information

Mark PTEs as invalid; page fault on use gets zeroed page

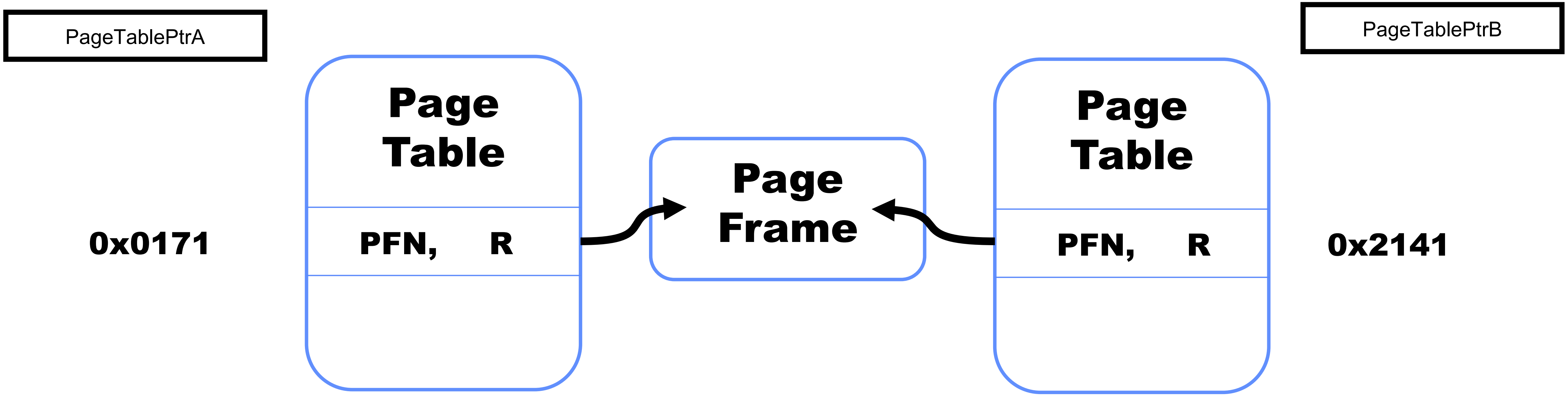
Data Breakpoints

For debugger, mark instruction page as read-only. Will trigger page-fault when try to execute

Paging & Sharing

Processes share a page by each mapping a page of their own virtual address space to the same frame

Use protection bits for fine-sharing



Where is page sharing used ?

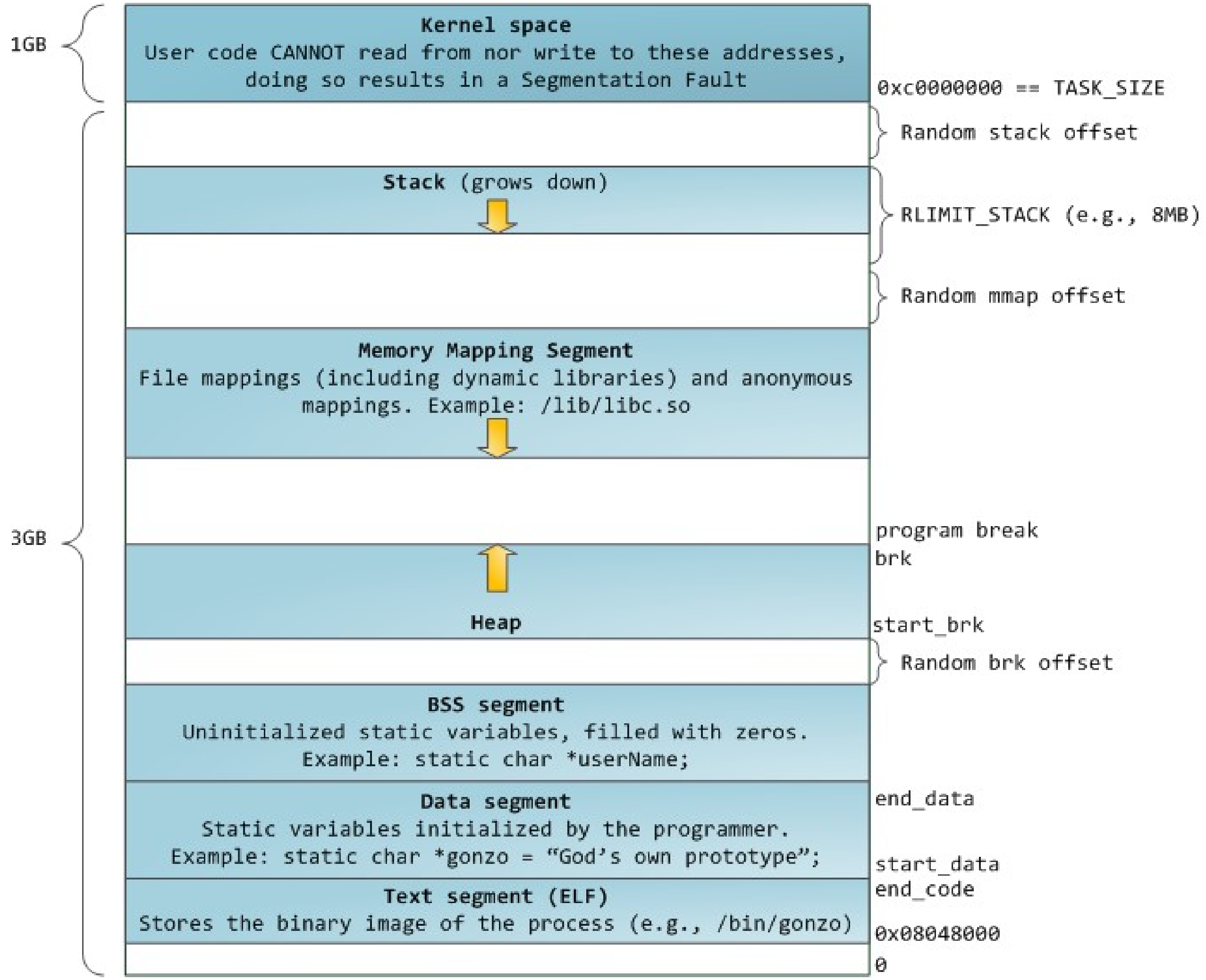
Kernel region of every process has the same page table entries

Different processes running same binary!

Do not need to duplicate code segments

Shared-memory segments between different processes

Memory Layout for Linux 32-bit


1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
3GB
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer. Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

An aside: Meltdown

From the paper:

Meltdown is a novel attack that allows overcoming memory isolation completely by providing a simple way for any user process to read the entire kernel memory of the machine it executes on, including all physical memory mapped in the kernel region. Meltdown does not exploit any software vulnerability, i.e., it works on all major operating systems.

MELTDOWN

```
raise_exception();
// the line below is never reached
access(probe_array[data * 4096]);]
```

Are we done?

How big can a page table get on x86 (32 bits)?

4KB page => 2^12
2^32/2^12 => 2^20 pages
2^20 * 4 bytes = 4 MB (approx.)
That’s a lot per process!!

How big can a page table get on x86 (64 bits)?

4KB page => 2^12
2^64/2^12 => 2^52 pages
2^20 * 8 bytes = 36 petabytes (approx.)
That’s a lot per process!!

Limitations of paging

Space overhead

With a 64-bit address space, size of page table can be huge

Time overhead

Accessing data now requires two memory accesses must also access page table, to find mapped frame

Internal Fragmentation

4KB pages

The Secret to the Whole of CS

Batching

Caching

Indirection

Specialised Hardware

Sparsity

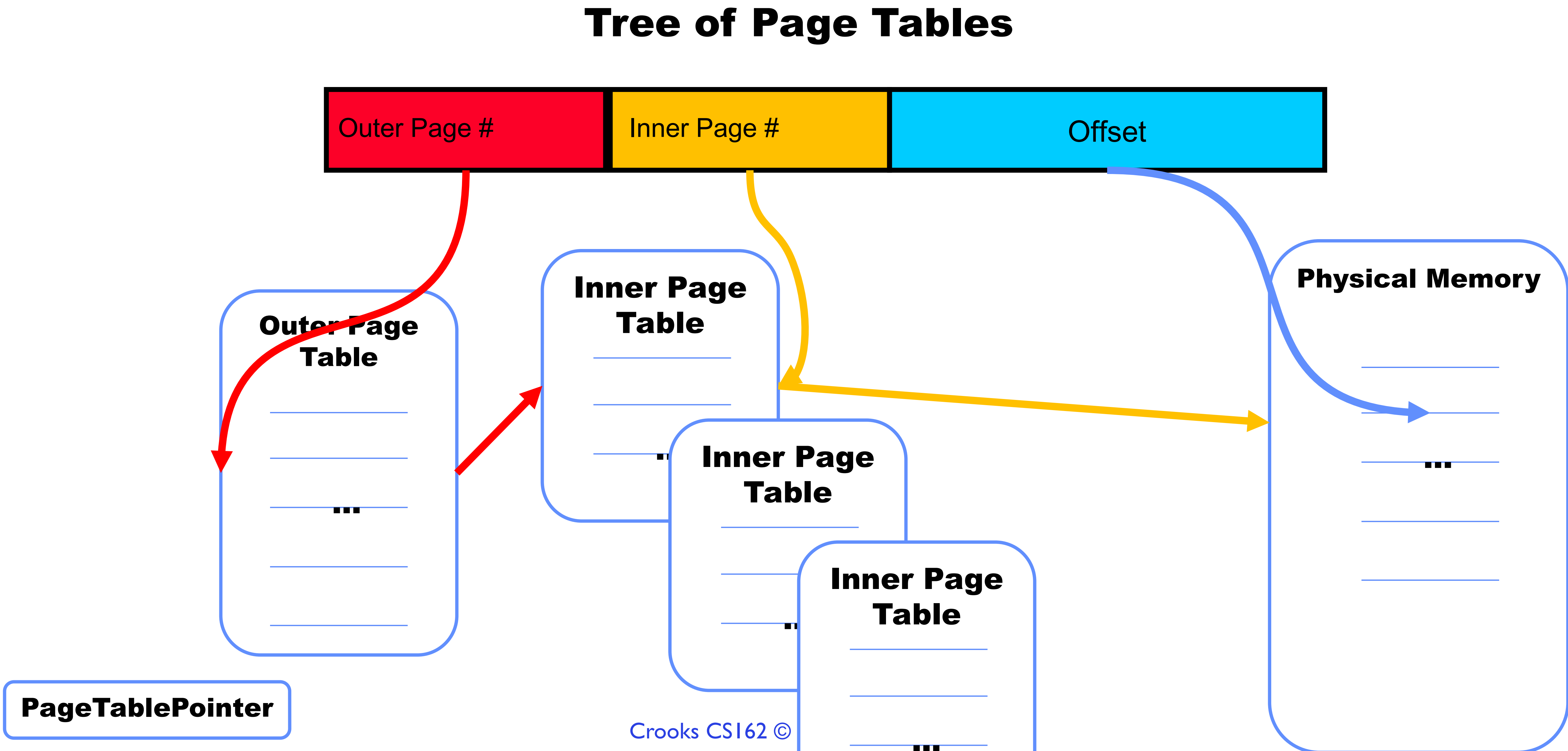
Address space is sparse, i.e. has holes that are not mapped to physical memory

Most this space is taken up by page tables mapped to nothing

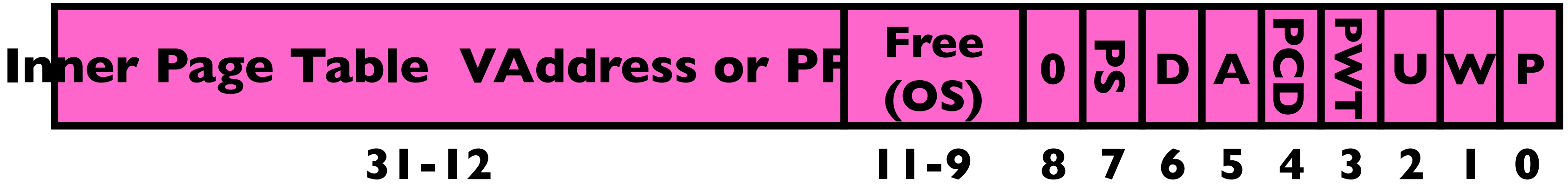
Process has access to full 2^64 bytes (virtually)

Physically, that would be 17,179,869,184 gigabytes

Paging the page table: 2-level paging



V2: What is a page table entry? (32 bits)



P: Present (same as "valid" bit in other architectures)

W: Writeable

U: User accessible

PWT: Page write transparent: external cache write-through

PCD: Page cache disabled (page cannot be cached)

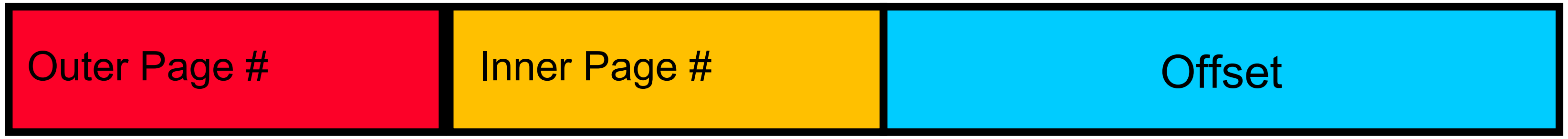
A: Accessed: page has been accessed recently

D: Dirty: page has been modified recently

PS: Page Size

Paging the page table: 2-level paging

Tree of Page Tables



Outer Page #

Inner Page #

Offset

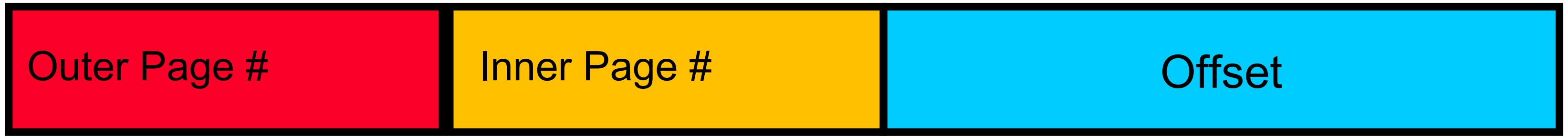
Number of top-level pages

Ensure that fits on a single page

Defines size of a page

Paging the page table: 2-level paging

Tree of Page Tables


Outer Page #

Inner Page #

Offset

10 bits

10 bits

4 KB
12 bits

Want to make sure that inner page table fits in a page!
2^12/2^2 = 2^10

Example: x86 classic 32-bit address translation

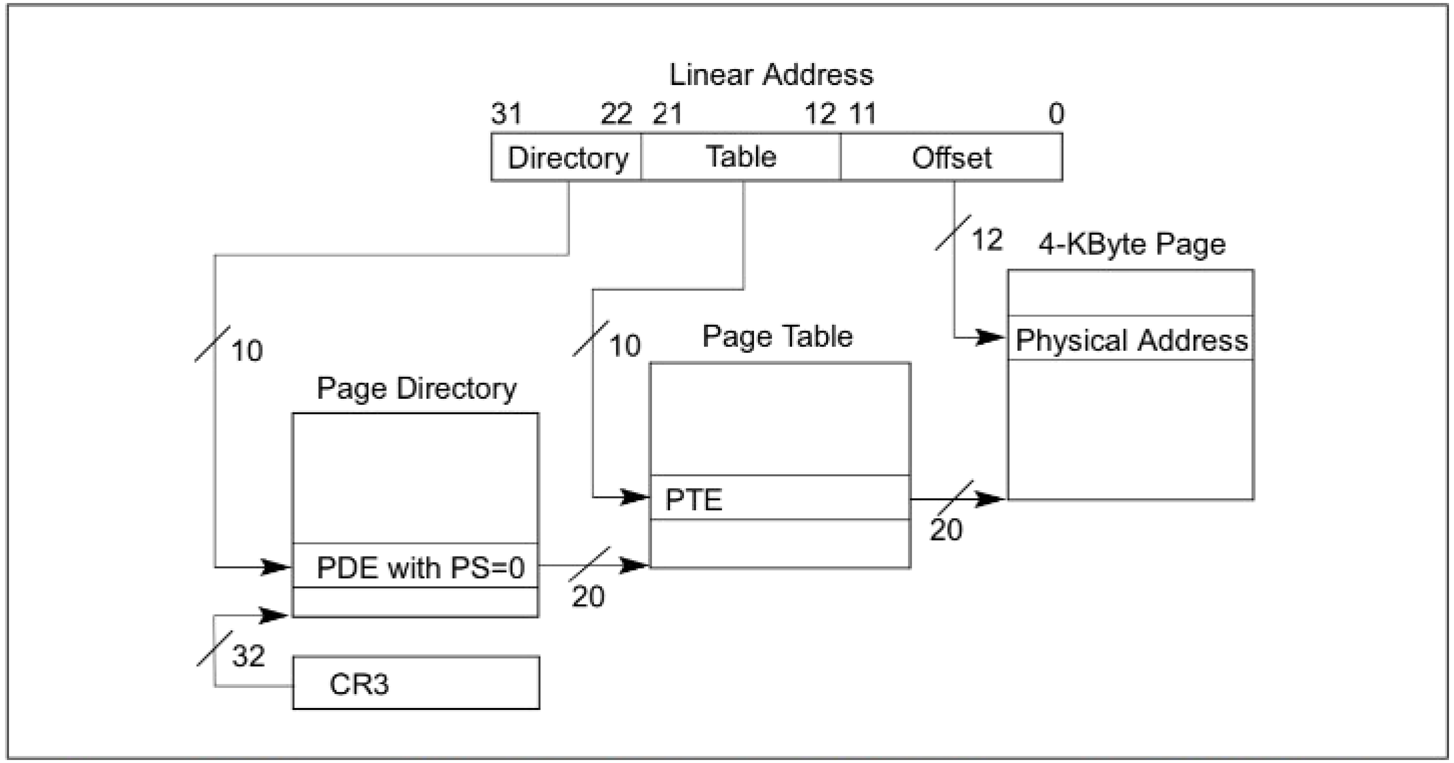
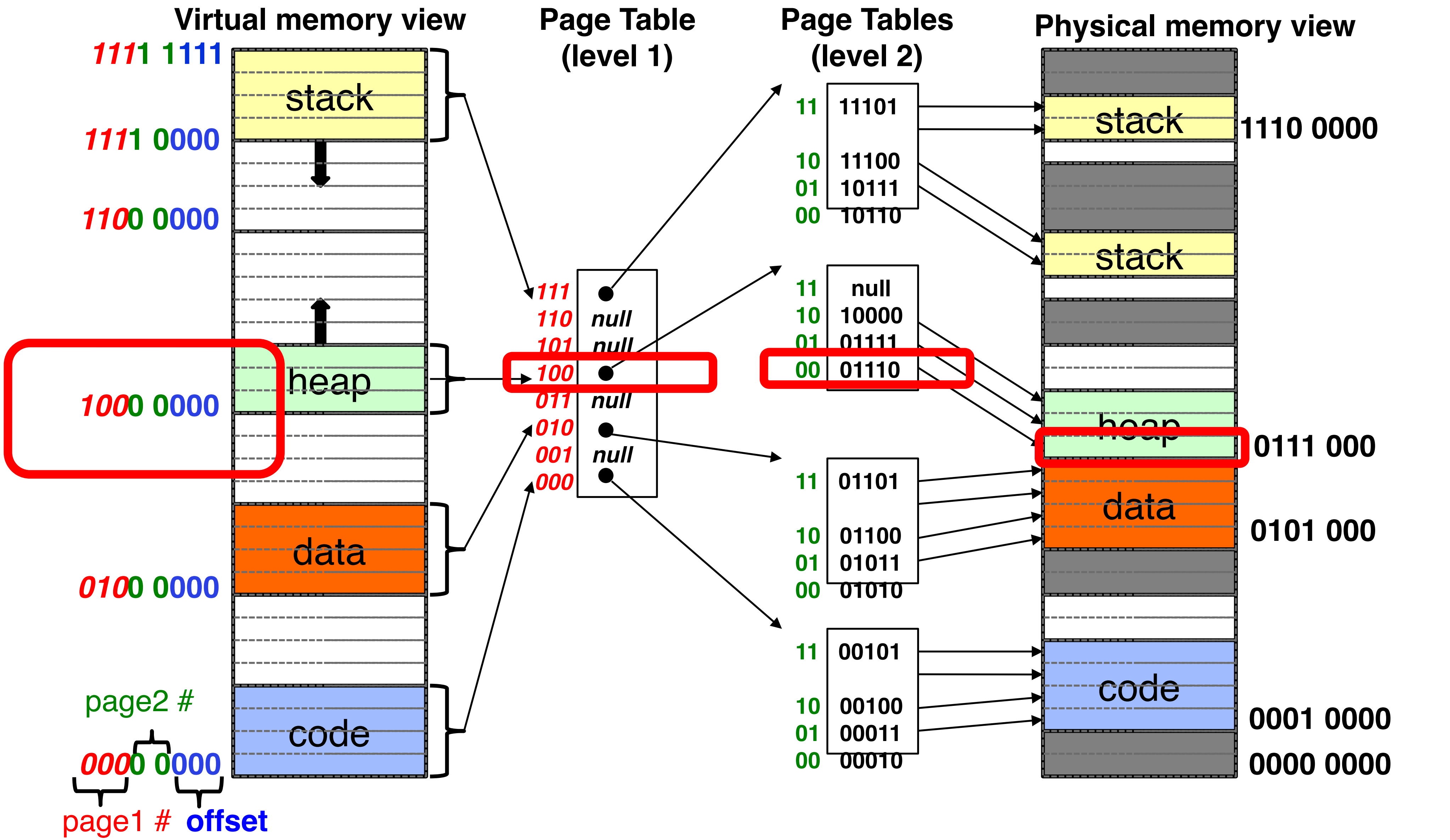


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

Top-level page-table: Page Directory

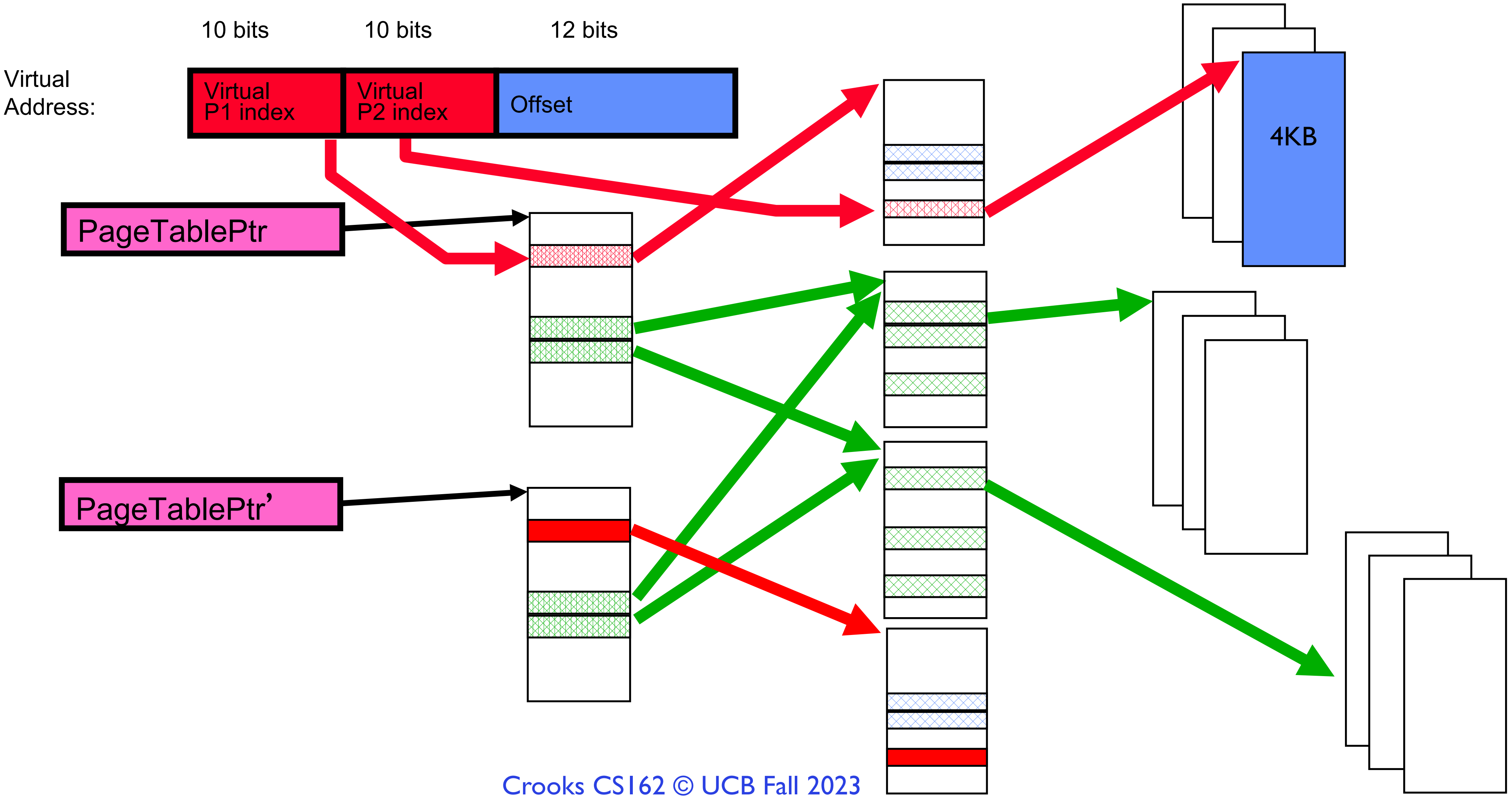
Inner page-table: Page Directory Entries

Example Address Space View



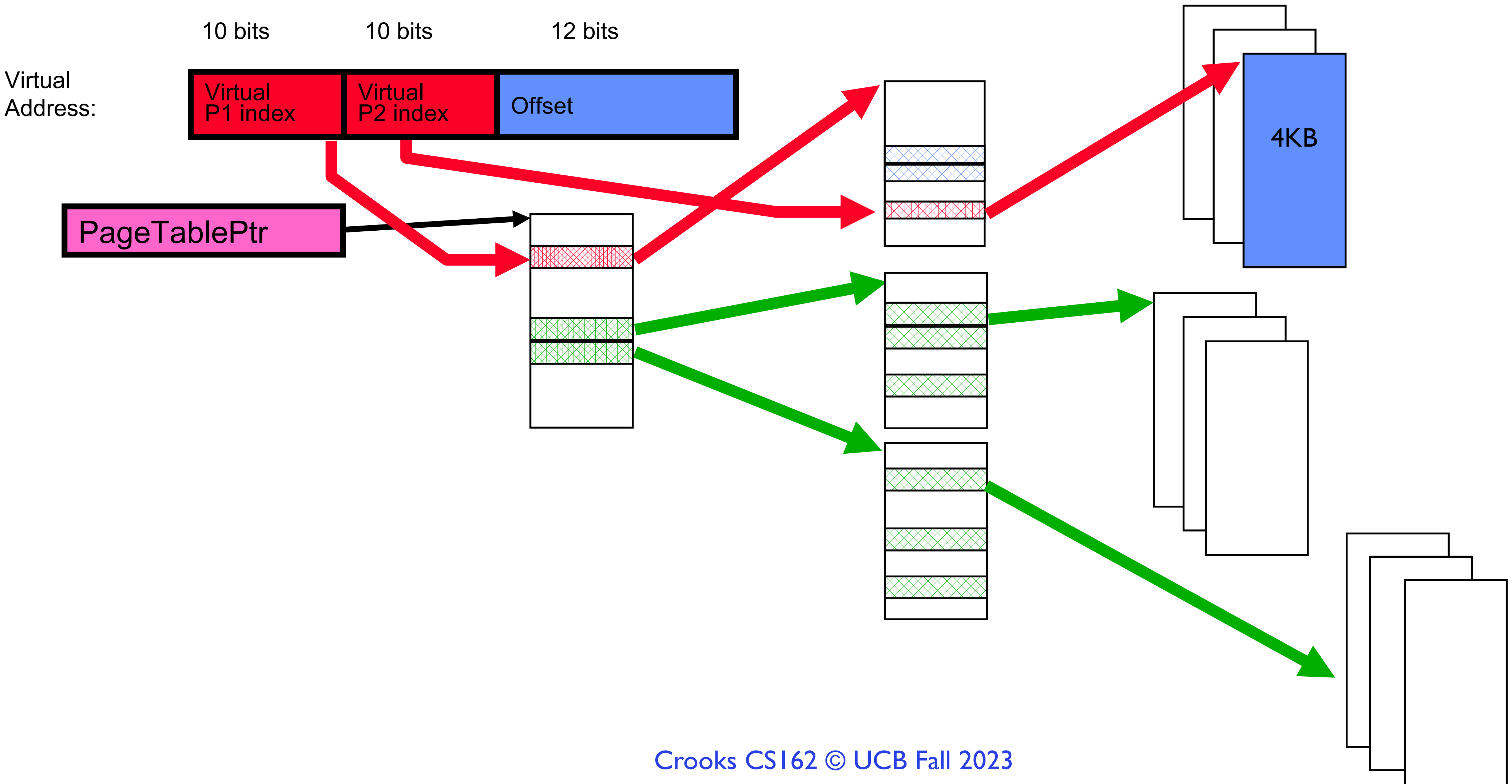
Sharing with multilevel page tables

Entire regions of the address space can be efficiently shared



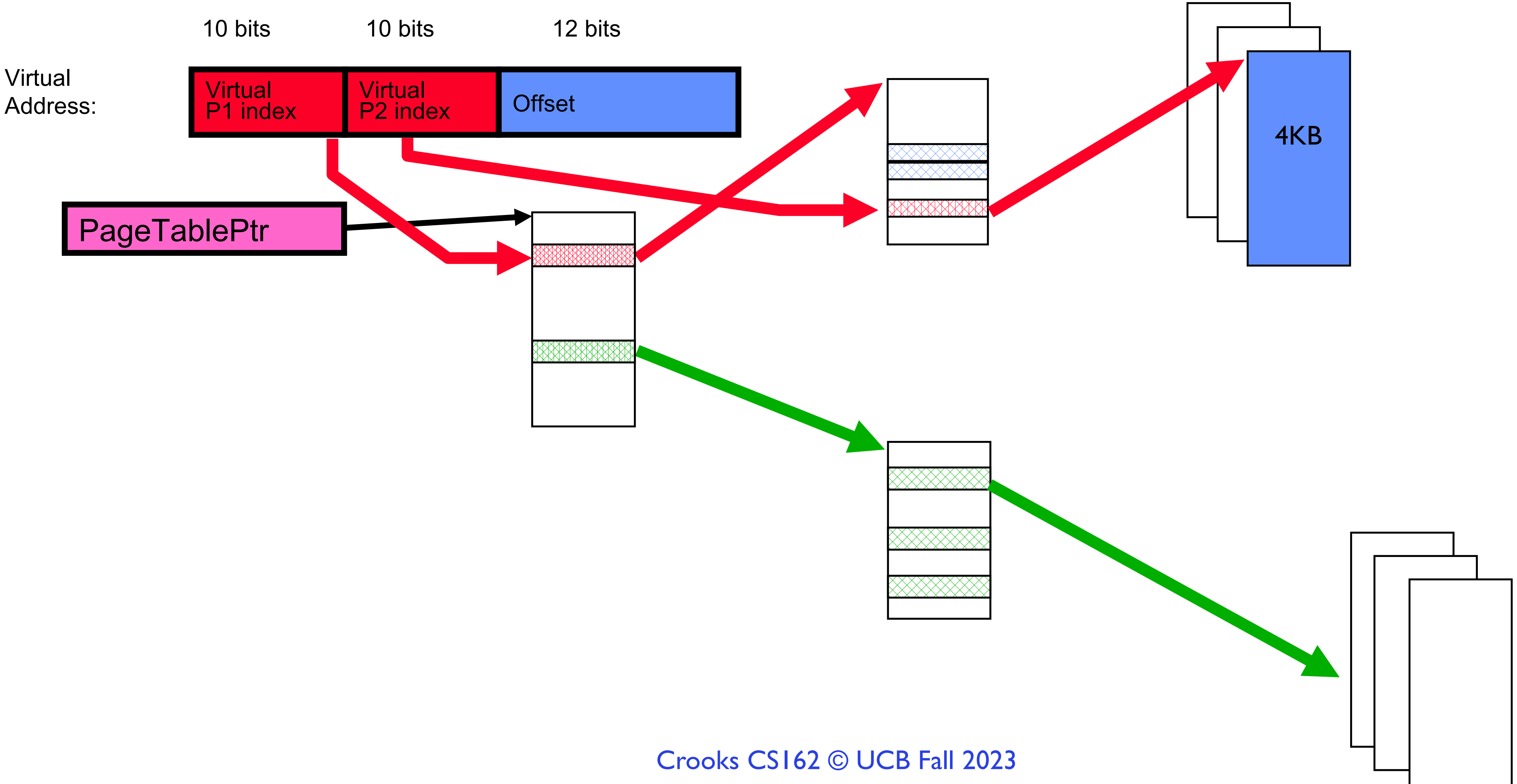
Marking entire regions as invalid!

If region of address space unused, can mark entire inner region as invalid



Marking entire regions as invalid!

If region of address space unused, can mark entire inner region as invalid



Has this helped?

Assuming 10/10/12 split:

Size of Page Table

Outer: (2^10 * 4 bytes) +
Inner: 2^10 * (2^10 * 4 bytes)

Overhead of indirection! BUT Marking inner pages as invalid helps when address spaces are sparse

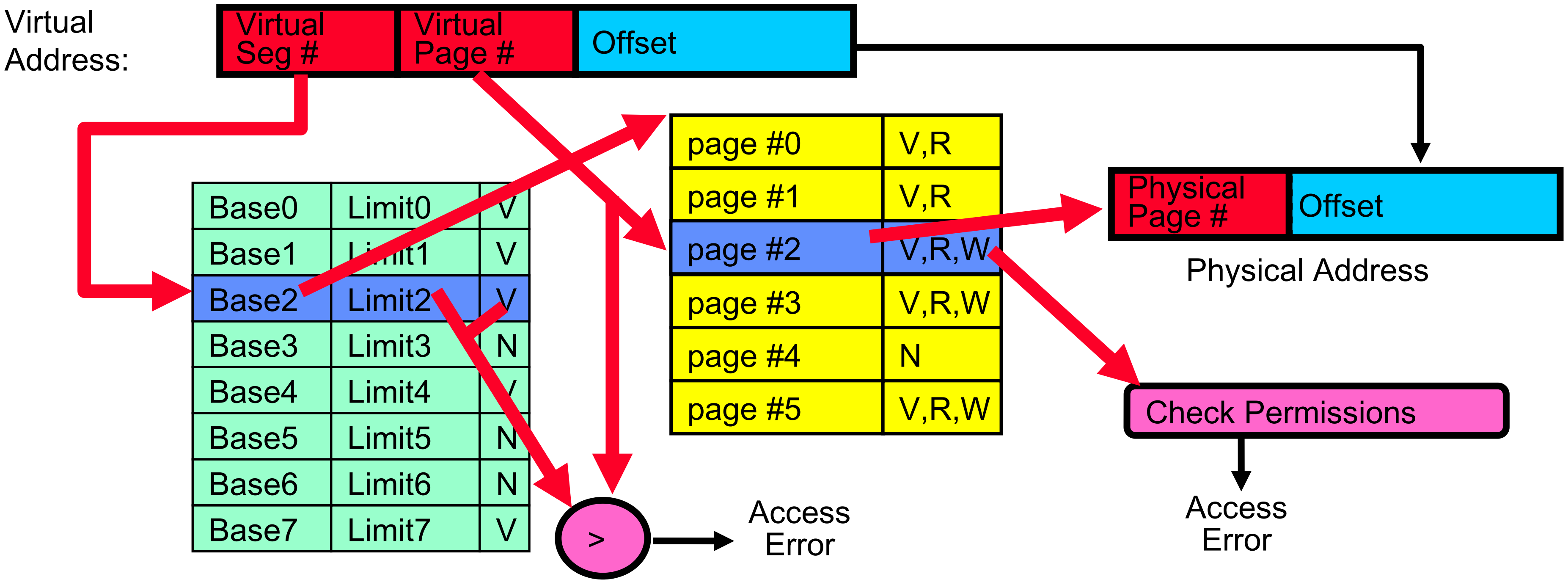
Downside: now have to do

two memory accesses for translation

Paged Segmentation

Use segments for top level. Paging within each segment.

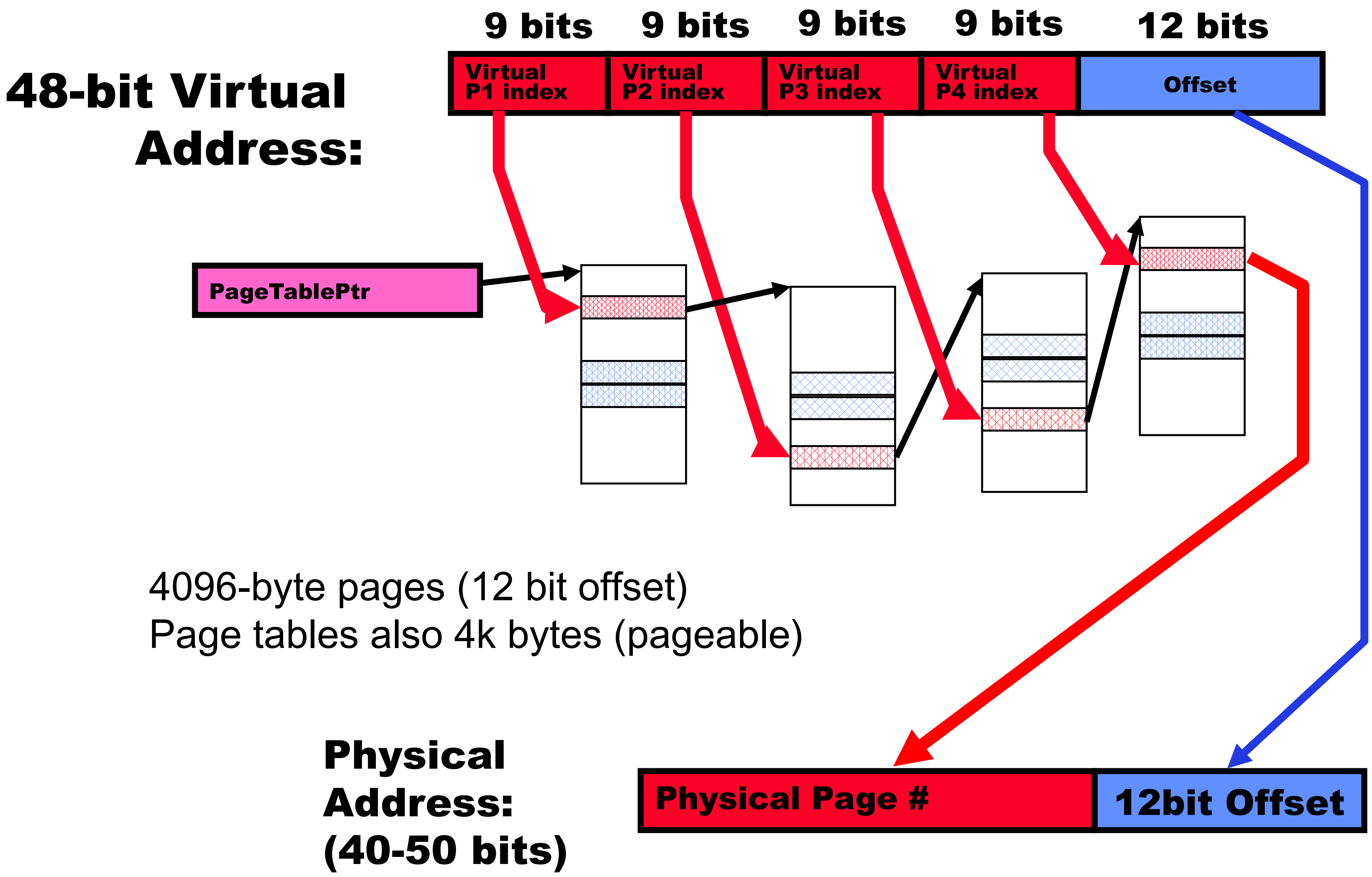
Used in x86 (32 bit).
Code Segment, Data Segment, etc.



Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

X86 64 bits has a four-level page table!



4096-byte pages (12 bit offset)
Page tables also 4k bytes (pageable)

Inverted Page Table

A single page table that has an entry for each physical page of the system

Each entry contains process ID + which virtual page maps to physical page

Physical memory much smaller than virtual memory

Size proportional to size of physical memory

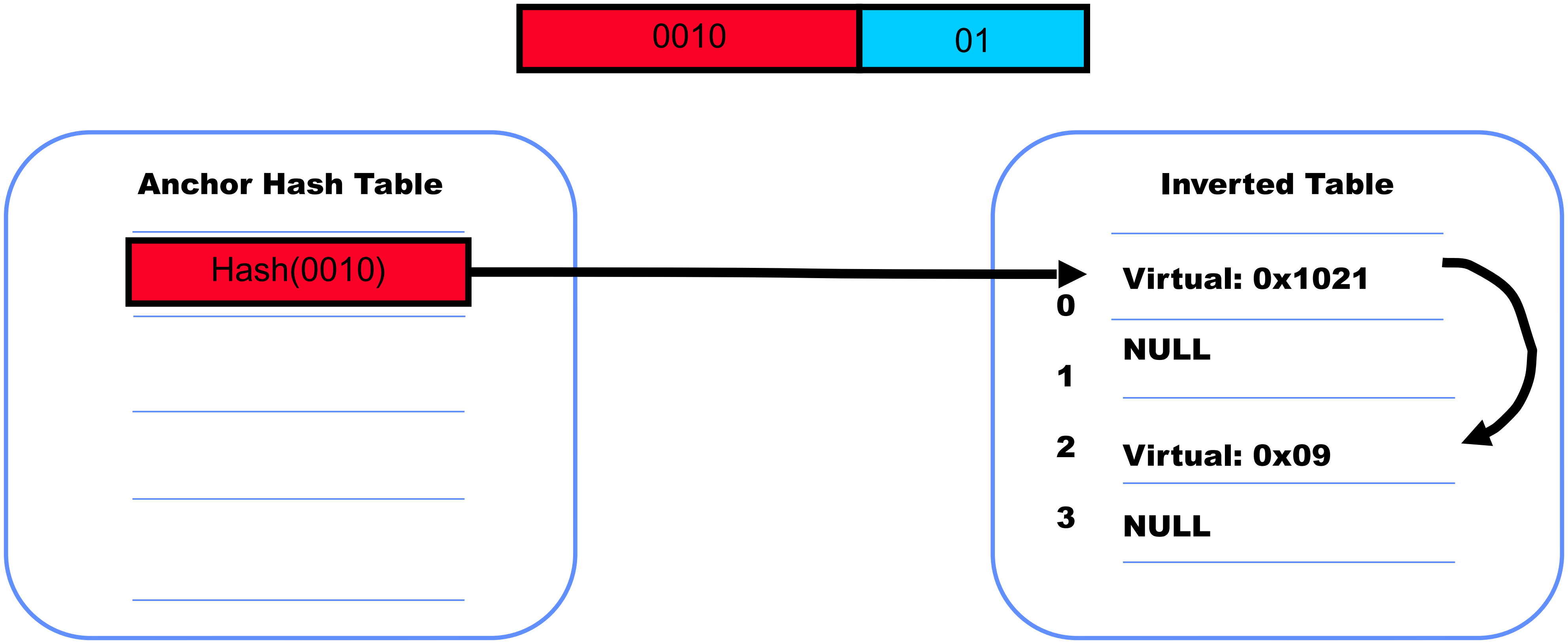
Inverted Table

0	**Virtual: 0x1021**
1	**NULL**
2	**Virtual: 0x0123**
3	**NULL**

Inverted Page Table

Don't we have it backwards?

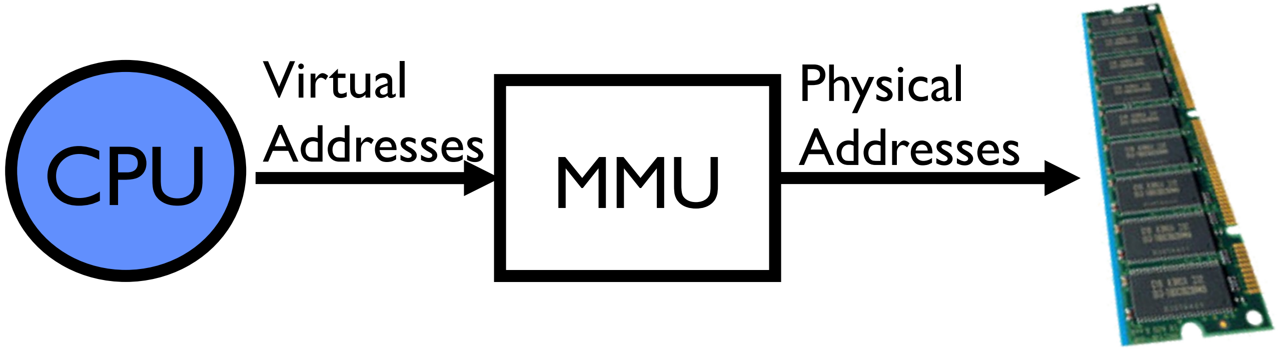
Add a hash table. Virtual memory can only map to specific physical frames



Address Translation Comparison

	Advantages	Disadvantages
Simple Segmentation	Fast context switching (segment map maintained by CPU)	External fragmentation
Paging (Single-Level)	No external fragmentation Fast and easy allocation	Large table size (~ virtual memory) Internal fragmentation
Paged Segmentation	Table size ~ # of pages in virtual memory Fast and easy allocation	Multiple memory references per page access
Multi-Level Paging		
Inverted Page Table	Table size ~ # of pages in physical memory	Hash function more complex No cache locality of page table

How is the Translation Accomplished?



MMU must translate virtual address to physical address on every instruction fetch, load or store

What does the MMU need to do to translate an address?
Read, check, and update PTE
(set accessed bit/dirty bit on write)

How can we speedup translation?

MMU must make at least 2 memory reads to walk page table. Slow!

Use specialized hardware to
cache virtual-physical memory translations!

Introducing the Translation Lookaside Buffer (TLB)

Recall: CS61c Caching Concept

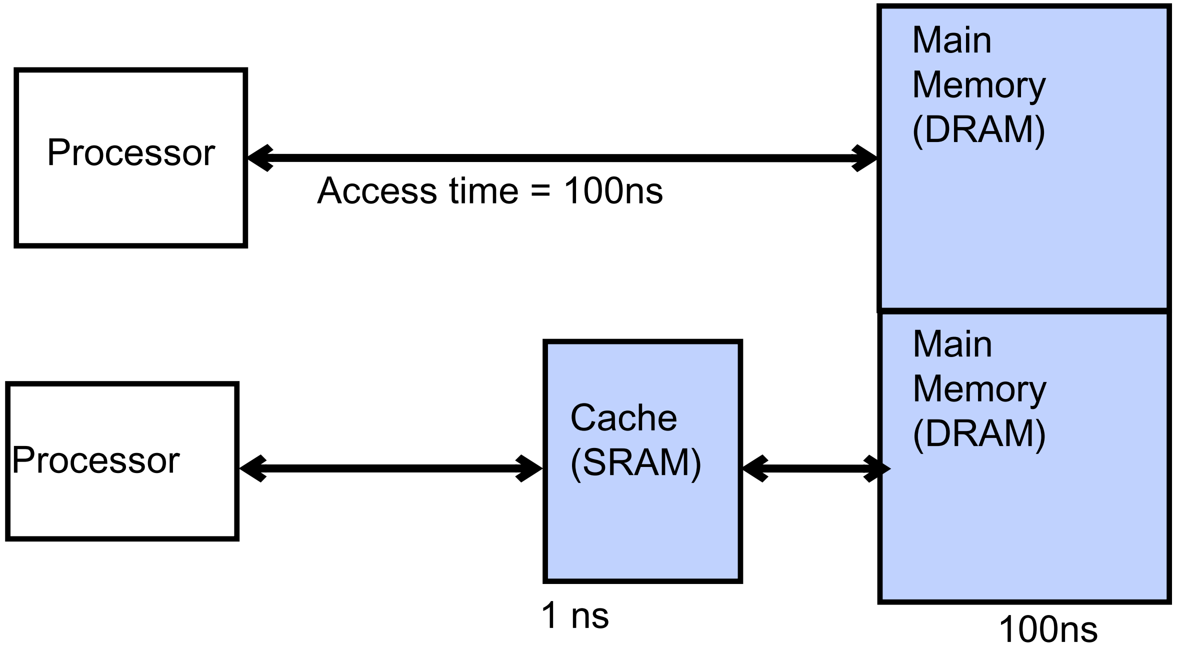
Cache: a repository for copies that can be accessed more quickly than the original

Only good if:
Frequent case frequent enough and
Infrequent case not too expensive

Important measure
Average Access time =
(Hit Rate x Hit Time) + (Miss Rate x Miss Time)

Recall: In Machine Structures (eg. 61C) …

Caching is the key to memory system performance



Recall: In Machine Structures (eg. 61C) …

Average Memory Access Time (AMAT)

= (Hit Rate x HitTime) + (Miss Rate x MissTime)

Where HitRate + MissRate = 1

HitRate = 90% => AMAT = (0.9 x 1) + (0.1 x 101)=11 ns

HitRate = 99% => AMAT = (0.99 x 1) + (0.01 x 101)=2.01 ns

MissTime_{L1} includes

$\text{HitTime}_{L1}+\text{MissPenalty}_{L1} \equiv \text{HitTime}_{L1} + \text{AMAT}_{L2}$

Why Does Caching Help? Locality!

Temporal Locality (Locality in Time):

Keep recently accessed data items closer to processor

Spatial Locality (Locality in Space):

Move contiguous blocks to the upper levels

Recall: Memory Hierarchy

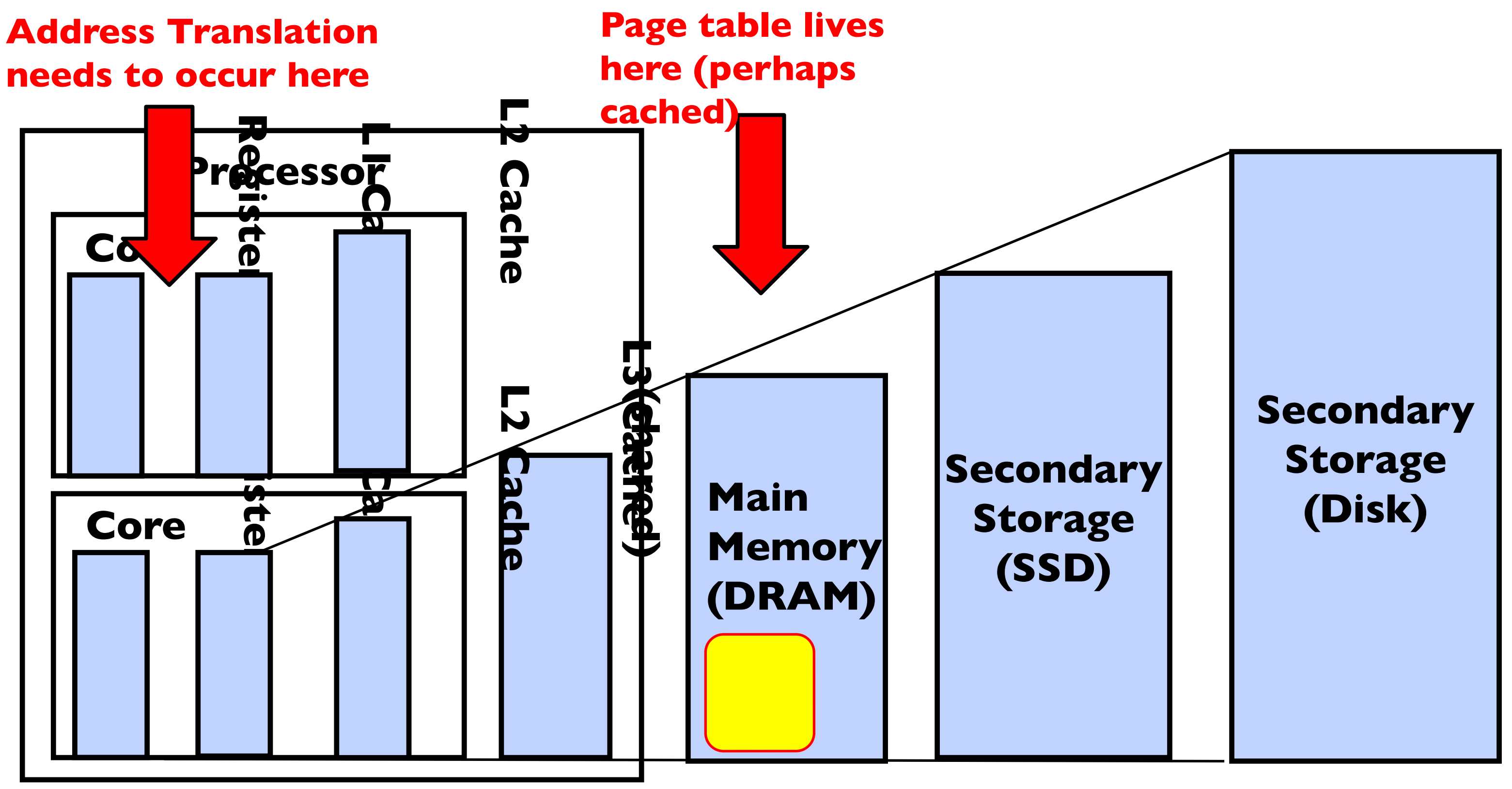
Take advantage of the principle of locality to:

1) Present the illusion of having as much memory as in the cheapest technology

2) Provide average speed similar to that offered by the fastest technology

Recall: fast but small/expensive. Slow but large!

Recall: Memory Hierarchy

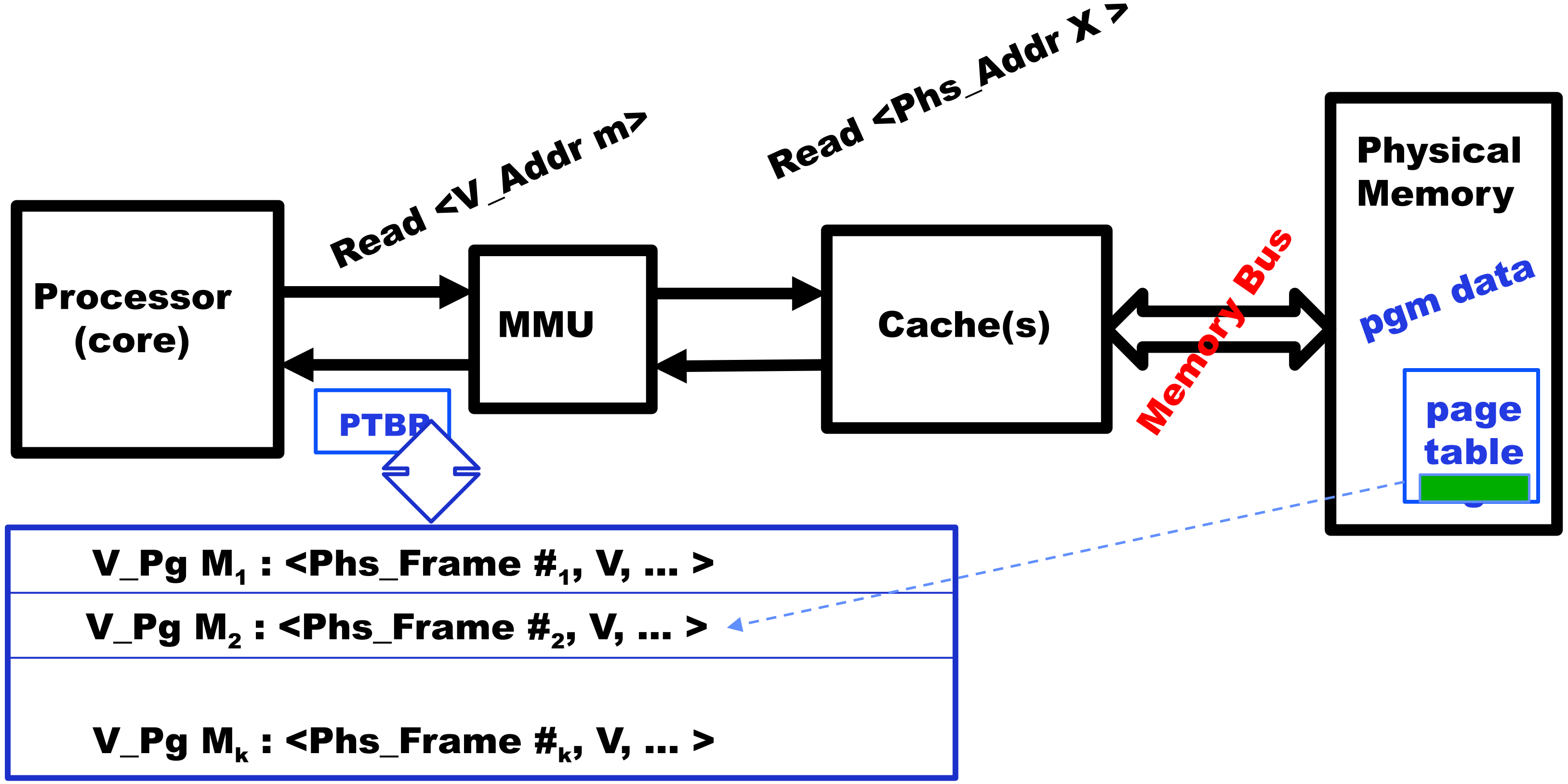


Speed (ns):	0.3	1	3	10-30	100	100,000 (0.1 ms)	10,000,000 (10 ms)
Size (bytes):	100Bs	10kBs	100kBs	MBs	GBs	100GBs	TBs

How do we make Address Translation Fast?

Cache results of recent translations !

Cache Page Table Entries using Virtual Page # as the key



V_Pg M_1 : <Phs_Frame $\#_1$, V, ... >

V_Pg M_2 : <Phs_Frame $\#_2$, V, ... >

V_Pg M_k : <Phs_Frame $\#_k$, V, ... >

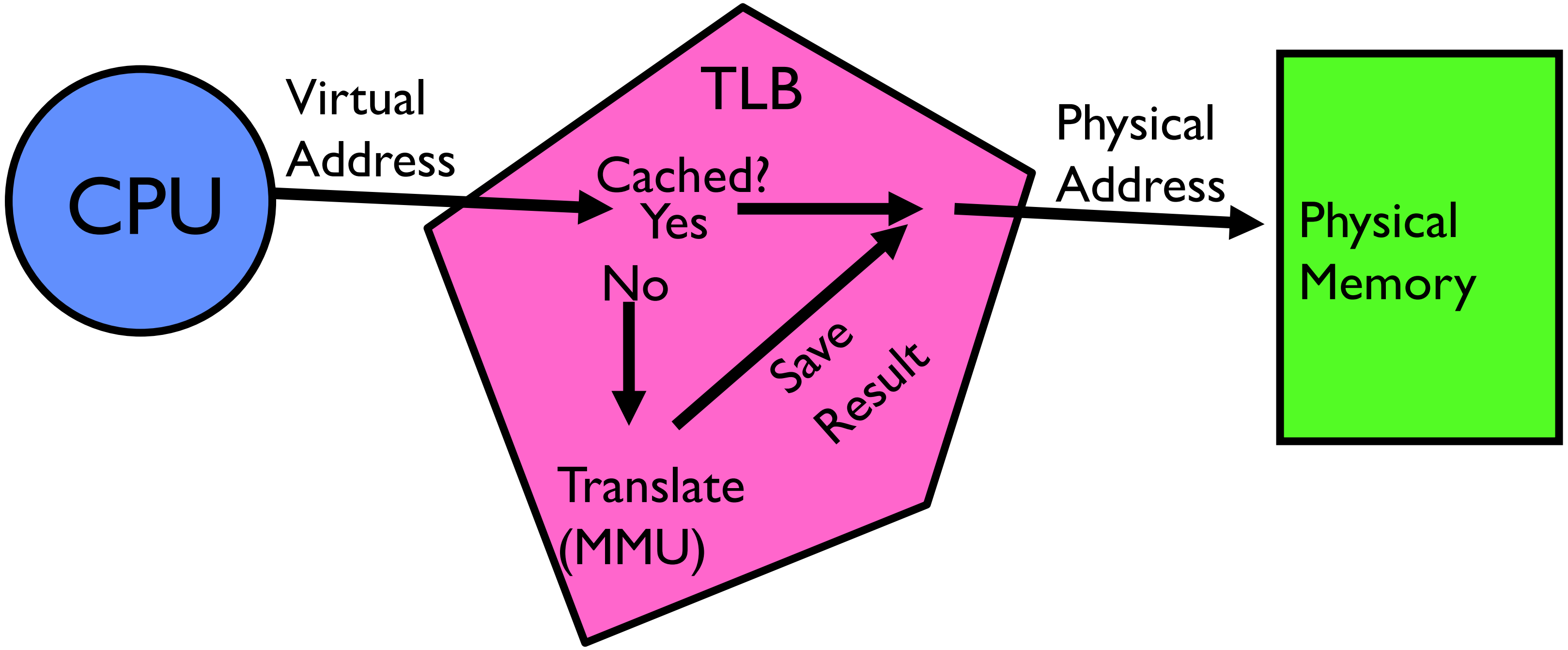
Translation Look-Aside Buffer

Record recent Virtual Page # to Physical Frame # translation

If present, have the physical address without reading any of the page tables !!!

Caches the end-to-end result

Caching Applied to Address Translation



Does page locality exist?

Instruction accesses spend a lot of time on the same page (since accesses sequential)

Stack accesses have definite locality of reference