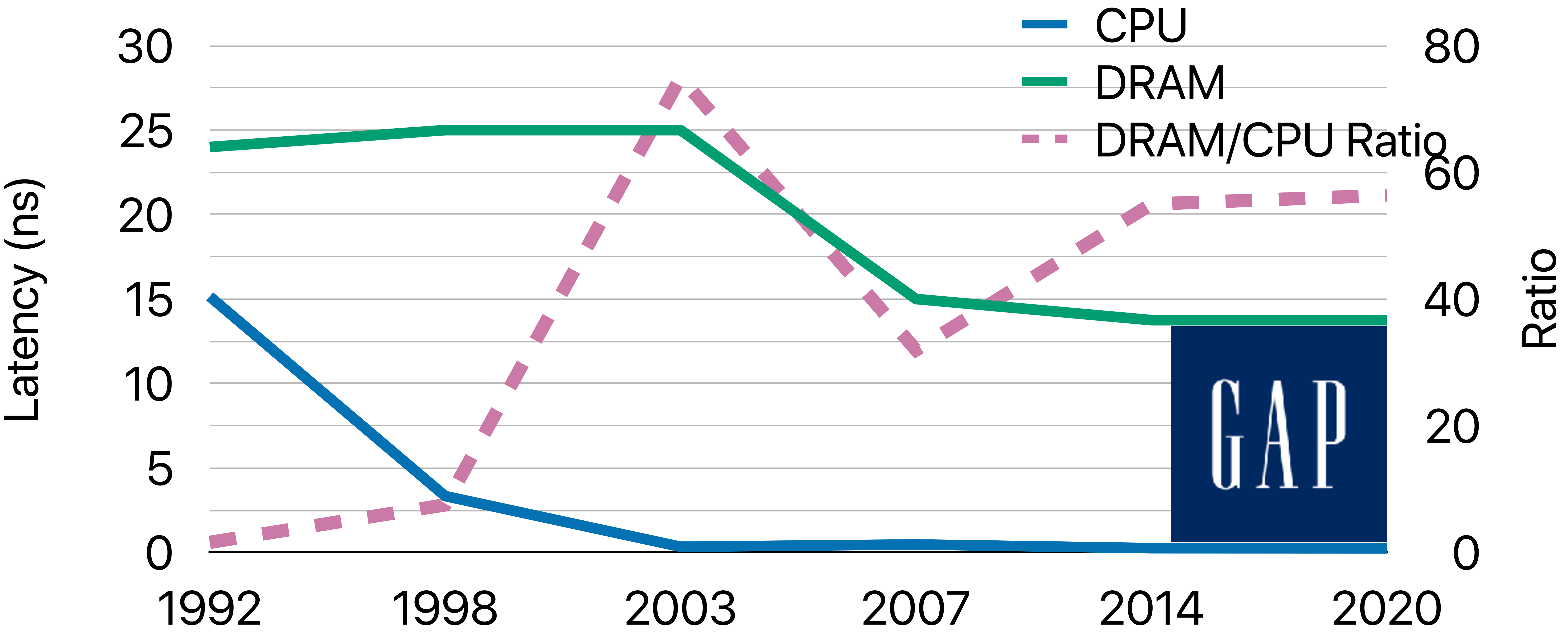
Virtual Memory: Just an Illusion

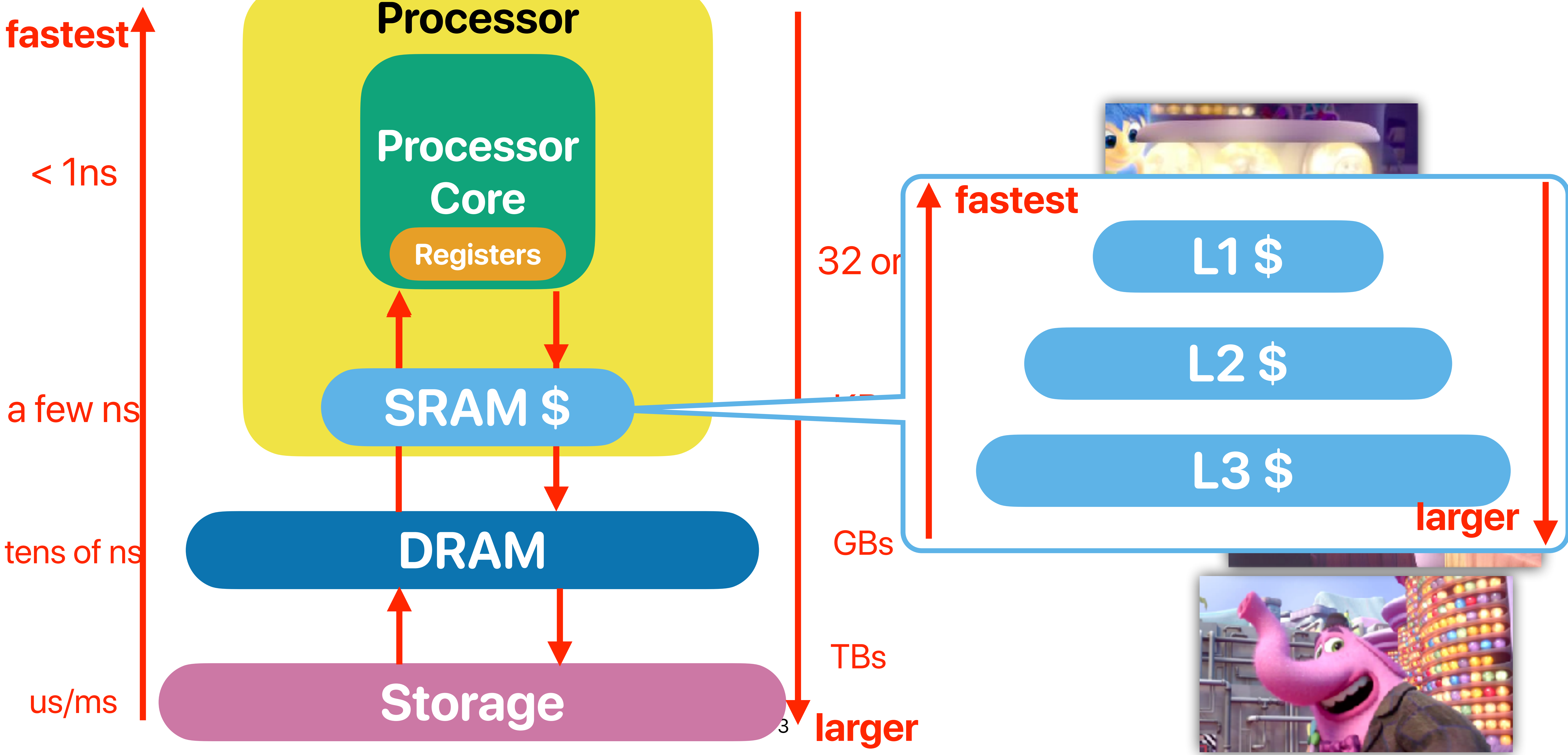
Hung-Wei Tseng

Recap: the “latency” gap between CPU and DRAM

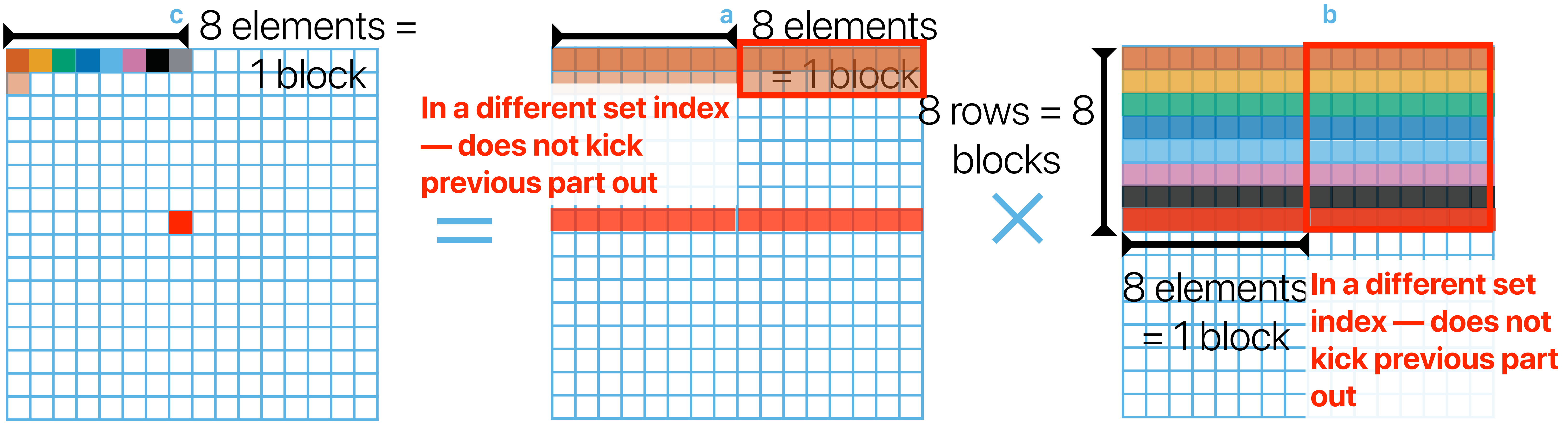


CPU	i486	Pentium II	Pentium 4	Core 2	Core i7-4790K	Core i5-10600K
DRAM	SDRAM	DDR	DDR2	DDR3	DDR4	DDR5

Recap: Memory Hierarchy



Tiling/Blocking Algorithm for Transposed Matrix Multiplications



In a different set index — does not kick previous part out

In a different set index — does not kick previous part out

We can make the "tile_size" larger without interfacing conflict misses

```
// Transpose matrix b into b_t
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
        b_t[i][j] += b[j][i];
    }
}
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                    for(kk = k; kk < k+tile_size; kk++)
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];
```

Takeaways: Optimizing cache performance through hardware

- There is no optimal cache configurations — trade-offs are everywhere
    - Increasing C — (+): capacity misses; (-): cost, access time, power
    - Increasing A — (+): conflict misses; (-): access time, power
    - Increasing B — (+): compulsory misses; (-): miss penalty
- Adding a small buffer alongside the L1 cache can —
    - Virtually add an associative set to frequently used data structures
    - Prefetched blocks won't cause conflict misses
- Software Optimization
    - Data layout — capacity miss, conflict miss, compulsory miss
    - Loop interchange — conflict/capacity miss
    - Loop fission — conflict miss — when $ has limited way associativity
    - Loop fusion — capacity miss — when $ has enough way associativity
    - Blocking/tiling — capacity miss, conflict miss
    - Matrix transpose (a technique changes layout) — conflict misses
    - Using registers whenever possible — reduce memory accesses!
- Software-control, architectural-supported approach
    - Prefetching instructions
    - Adding a tag-less, programmable small buffer alongside the L1 cache can reduce power consumption

Let’s dig into this code

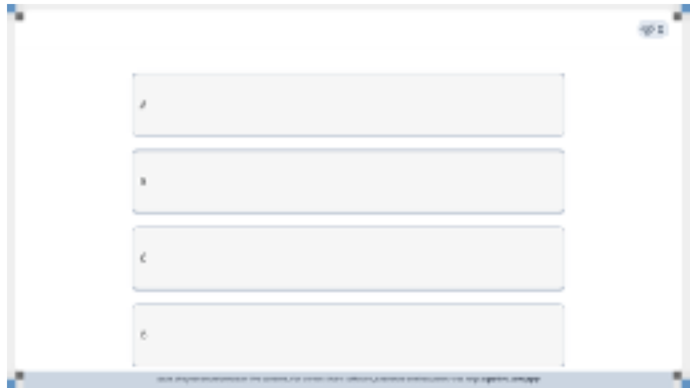
```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

https://www.pollev.com/hungweitseng close in 1:30

What will happen?

- If we execute the code on the right-hand side code on a machine with only 8 GB of physical memory installed and the dim is **33000** (requires 33000*33000*8 bytes ~ **8.12 GB** memory at least), What will happen?
  A. The program will crash in one of the malloc function call
  B. The program will crash due to a "segmentation fault" that caused by accessing NULL pointer
  C. The program will be killed automatically by the OS as it uses more than installed physical main memory
  D. The program will finish without any issue

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```



What will happen?

- If we execute the code on the right-hand side code on a machine with only 8 GB of physical memory installed and the dim is **33000** (requires 33000*33000*8 bytes ~ **8.12 GB** memory at least), What will happen?
    A. The program will crash in one of the malloc function call
    B. The program will crash due to a "segmentation fault" that caused by accessing NULL pointer
    C. The program will be killed automatically by the OS as it uses more than installed physical main memory
    D. The program will finish without any issue

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

What will happen?

- If we execute the code on the right-hand side code on a machine with only 8 GB of physical memory installed and the dim is **33000** (requires 33000*33000*8 bytes ~ **8.12 GB** memory at least), What will happen?
    A. The program will crash in one of the malloc function call
    B. The program will crash due to a “segmentation fault” that caused by accessing NULL pointer
    C. The program will be killed automatically by the OS as it uses more than installed physical main memory
    [✓ D. The program will finish without any issue]

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

Let’s dig into this code

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    // Create processes
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    // Generate rand seed
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address of a is %p\n”,getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address of a is %p\n”,getpid(), a, &a);
    return 0;
}
```

https://www.pollev.com/hungweitseng close in 1:30

Consider the following code …

- Consider the case when we run 4 instances of the given program at the same time on modern machines, how many statements correct?
  ① The printed "address of a" is the same for every running instances
  ② The printed "address of a" is different for each instance
  ③ All running instances will print the same value of a for the first printf
  ④ Each instance will print a different value of a for the first printf
  ⑤ For the fprintf of 10 Seconds later, each running instance will print the same value from it's last run
  ⑥ For the fprintf of 10 Seconds later, each running instance will print a different value from it's last run

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>
#include <stdint.h>

double a = 0;

int main(int argc, char *argv[])
{
    uint64_t i, number_of_total_processes=4, p;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s number_of_processes\n",argv[0]);
        exit(1);
    }
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d: Value of a is %lf and address of a is
%p\n",(int)getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\n10 Seconds Later -- Process %d: Value of a is %lf
and address of a is %p\n",(int)getpid(), a, &a);
    return 0;
}
```

Consider the following code …

- Consider the case when we run 4 instances of the given program at the same time on modern machines, how many statements correct?
  - ① The printed "address of a" is the same for every running instances
  - ② The printed "address of a" is different for each instance
  - ③ All running instances will print the same value of a for the first printf
  - ④ Each instance will print a different value of a for the first printf
  - ⑤ For the fprintf of 10 Seconds later, each running instance will print the same value from it's last run
  - ⑥ For the fprintf of 10 Seconds later, each running instance will print a different value from it's last run

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>
#include <stdint.h>

double a = 0;

int main(int argc, char *argv[])
{
    uint64_t i, number_of_total_processes=4, p;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s number_of_processes\n",argv[0]);
        exit(1);
    }
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d: Value of a is %lf and address of a is
%p\n",(int)getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\n10 Seconds Later -- Process %d: Value of a is %lf
and address of a is %p\n",(int)getpid(), a, &a);
    return 0;
}
```

Consider the following code ...

- Consider the case when we run 4 instances of the given program at the same time on modern machines, how many statements correct?

    ① The printed “address of a” is the same for every running instances

    ② The printed “address of a” is different for each instance

    ③ All running instances will print the same value of a for the first printf

    ④ Each instance will print a different value of a for the first printf

    ⑤ For the fprintf of 10 Seconds later, each running instance will print the same value from it’s last run

    ⑥ For the fprintf of 10 Seconds later, each running instance will print a different value from it’s last run

    A. 0

    B. 1

    C. 2

    [✓ D. 3]

    E. 4

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>
#include <stdint.h>

double a = 0;

int main(int argc, char *argv[])
{
    uint64_t i, number_of_total_processes=4, p;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s number_of_processes\n",argv[0]);
        exit(1);
    }
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d: Value of a is %lf and address of a is
%p\n",(int)getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\n10 Seconds Later -- Process %d: Value of a is %lf
and address of a is %p\n",(int)getpid(), a, &a);
    return 0;
}
```

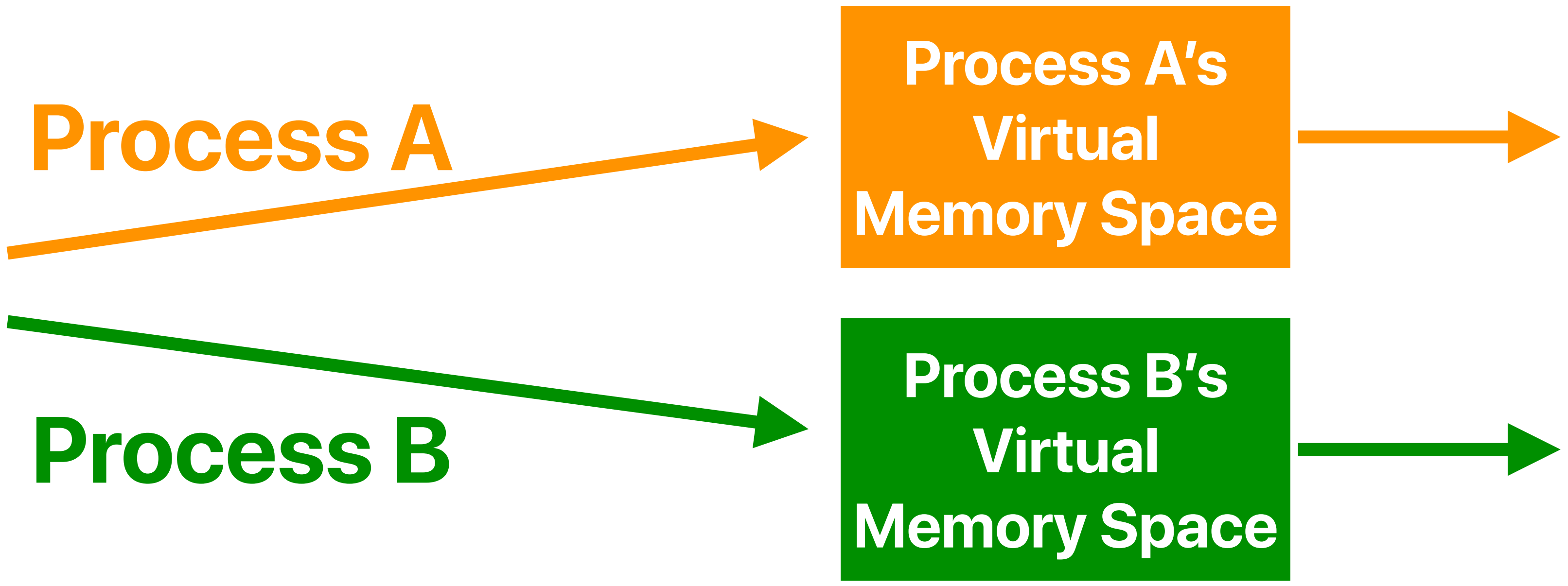
Outline

- Virtual memory
- Architectural support for virtual memory

Virtual Memory

Demo revisited

&a = 0x5da1e73ef030



Process A

Process A's Virtual Memory Space

Process B

Process B's Virtual Memory Space

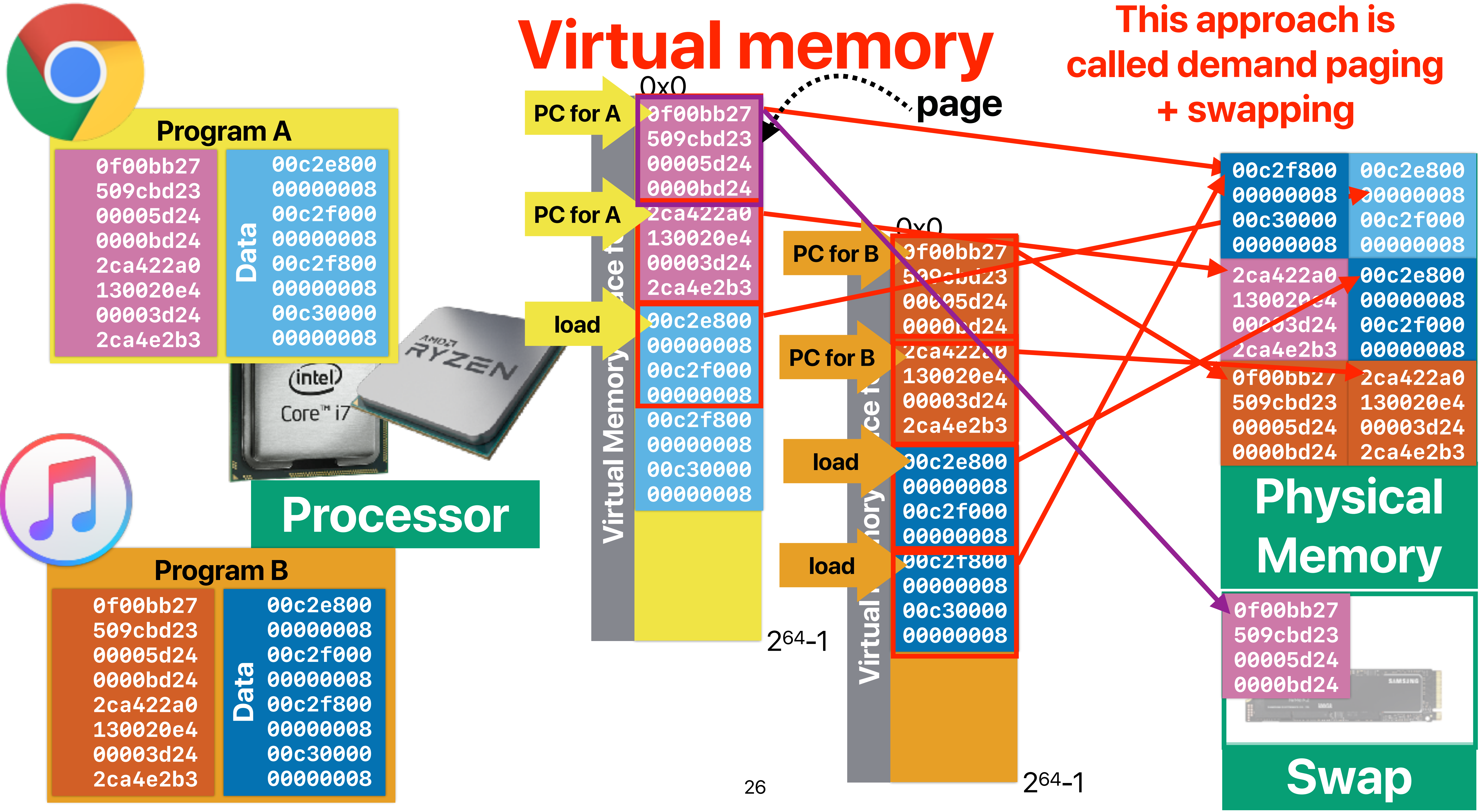
```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>
#include <stdint.h>

double a = 0;

int main(int argc, char *argv[])
{
    uint64_t i, number_of_total_processes=4, p;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s number_of_processes\n",argv[0]);
        exit(1);
    }
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d: Value of a is %lf and address of a is %p\n",(int)getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\n10 Seconds Later -- Process %d: Value of a is %lf and address of a is %p\n",(int)getpid(), a, &a);
    return 0;
}
```

Why Virtual memory?

- Allowing multiple applications to share physical main memory
  - Memory protection/isolation among programs/processes is automatically achieved
- Allowing applications to work even the installed physical memory or available physical memory is smaller than the working set of the application
  - Programmer does not need to worry about the physical memory capacity of different machines — make compiled program compatible
  - Multiple programs can work concurrently even through their total memory demand is larger than the installed physical memory

Virtual memory

This approach is called demand paging + swapping



Program A

0f00bb27
509cbd23
00005d24
0000bd24
2ca422a0
130020e4
00003d24
2ca4e2b3

Data

00c2e800
00000008
00c2f000
00000008
00c2f800
00000008
00c30000
00000008

Processor

Program B

0f00bb27
509cbd23
00005d24
0000bd24
2ca422a0
130020e4
00003d24
2ca4e2b3

Data

00c2e800
00000008
00c2f000
00000008
00c2f800
00000008
00c30000
00000008

Virtual memory space for A:

0f00bb27
509cbd23
00005d24
0000bd24
2ca422a0
130020e4
00003d24
2ca4e2b3
00c2e800
00000008
00c2f000
00000008
00c2f800
00000008
00c30000
00000008

Virtual memory space for B:

0f00bb27
509cbd23
00005d24
0000bd24
2ca422a0
130020e4
00003d24
2ca4e2b3
00c2e800
00000008
00c2f000
00000008
00c2f800
00000008
00c30000
00000008

Physical Memory:

00c2f800 00000008 00c30000 00000008	00c2e800 00000008 00c2f000 00000008
2ca422a0 130020e4 00003d24 2ca4e2b3	00c2e800 00000008 00c2f000 00000008
0f00bb27 509cbd23 00005d24 0000bd24	2ca422a0 130020e4 00003d24 2ca4e2b3

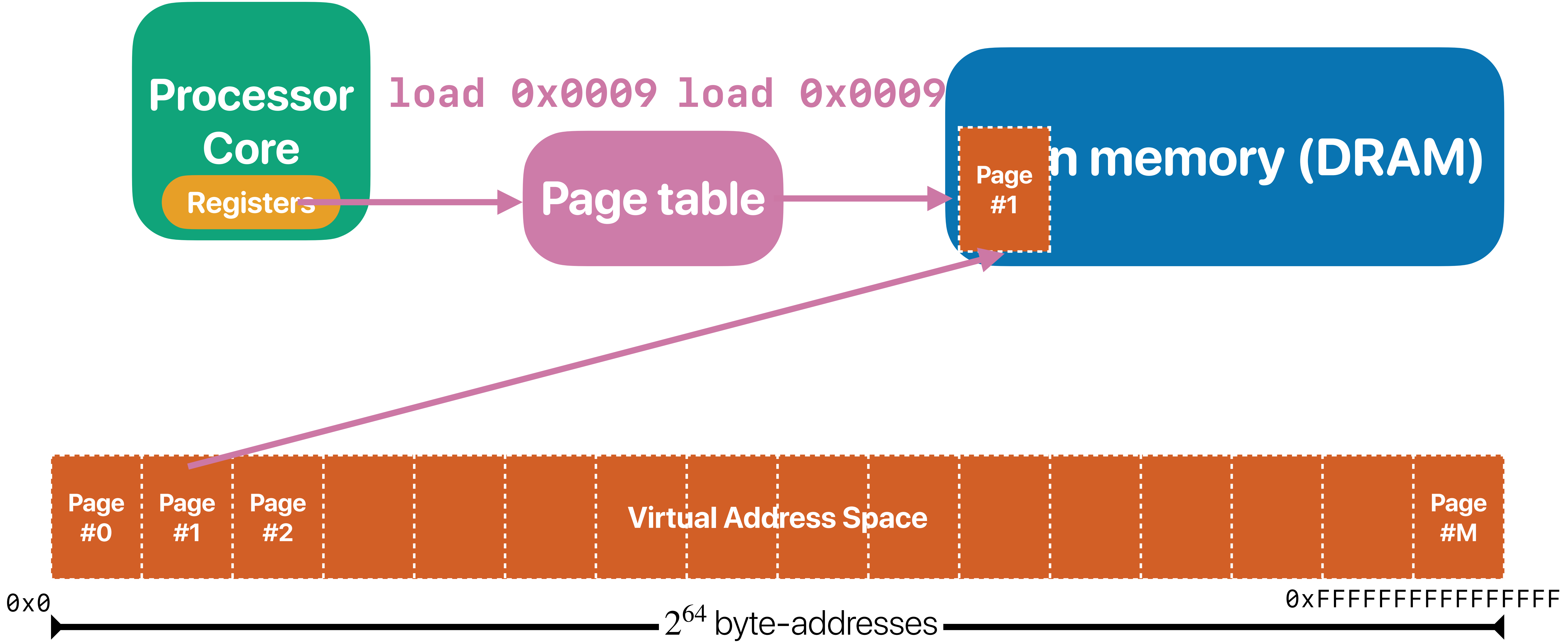
Swap:

0f00bb27
509cbd23
00005d24
0000bd24

Virtual memory

- An **abstraction** of memory space available for programs/software/programmer
- Programs execute using virtual memory address
- The operating system and hardware work together to handle the mapping between virtual memory addresses and real/physical memory addresses
- Virtual memory organizes memory locations into "**pages**"

Demand paging — another angle



Partition memory addresses into fix-sized chunks

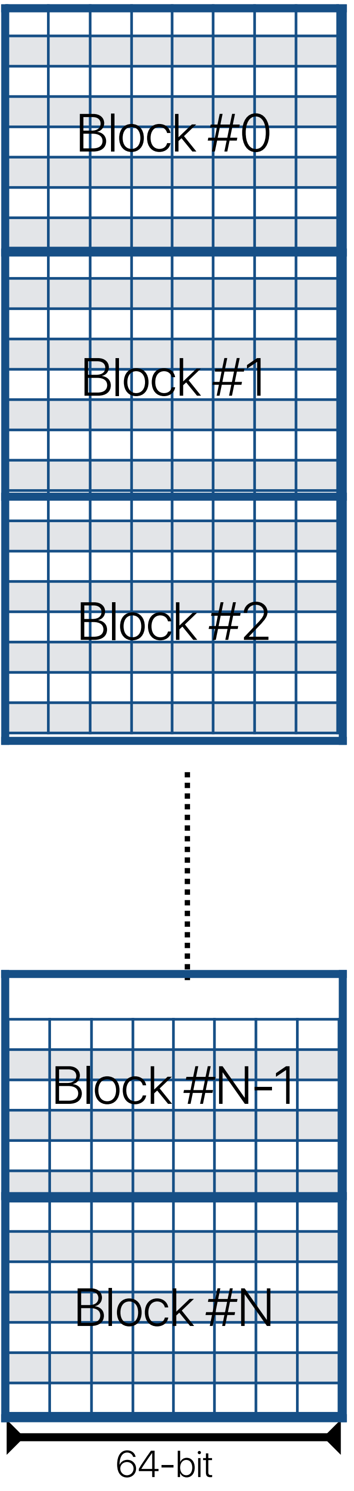
```
0b00000000000000000
0b00000000000001000
0b00000000000010000
0b00000000000011000
0b00000000000100000
0b00000000000101000
0b00000000001000000
0b00000000001001000
0b00000000010000000
0b00000000010001000
0b00000000010010000
0b00000000010011000
0b00000000000100000
0b00000000000101000
0b00000000001000000
0b00000000001001000
```

each block has a unique common prefix in there addresses!

Memory

2⁶⁴ Bytes

```
0x0000000000000000
0x0000000000000008
0x0000000000000010
0x0000000000000018
0x0000000000000020
0x0000000000000028
0x0000000000000030
0x0000000000000038
0x0000000000000040
0x0000000000000048
0x0000000000000050
0x0000000000000058
0x0000000000000060
0x0000000000000068
0x0000000000000070
0x0000000000000078
0x0000000000000080
0x0000000000000088
0x0000000000000090
0x0000000000000098
0x00000000000000A0
0x00000000000000A8
0x00000000000000B0
0x00000000000000B8

0xFFFFFFFFFFFFFF98
0xFFFFFFFFFFFFFFA0
0xFFFFFFFFFFFFFFA8
0xFFFFFFFFFFFFFFB0
0xFFFFFFFFFFFFFFB8
0xFFFFFFFFFFFFFFC0
0xFFFFFFFFFFFFFFC8
0xFFFFFFFFFFFFFFD0
0xFFFFFFFFFFFFFFD8
0xFFFFFFFFFFFFFFE0
0xFFFFFFFFFFFFFFE8
0xFFFFFFFFFFFFFFF0
0xFFFFFFFFFFFFFFF8
```



Processor Core
Registers
$

Demand paging + Swapping

- **Paging**: partition virtual/physical memory spaces into fix-sized pages
- **Page fault**: when the requested page cannot be found in the physical memory — created the demand of allocating pages!
- **Demand paging**: Allocate a physical memory page for a virtual memory page when the virtual page is needed (page fault occurs)
- Swapping: use secondary storage to store pages not in DRAM

Demand paging + Swapping v.s. caching

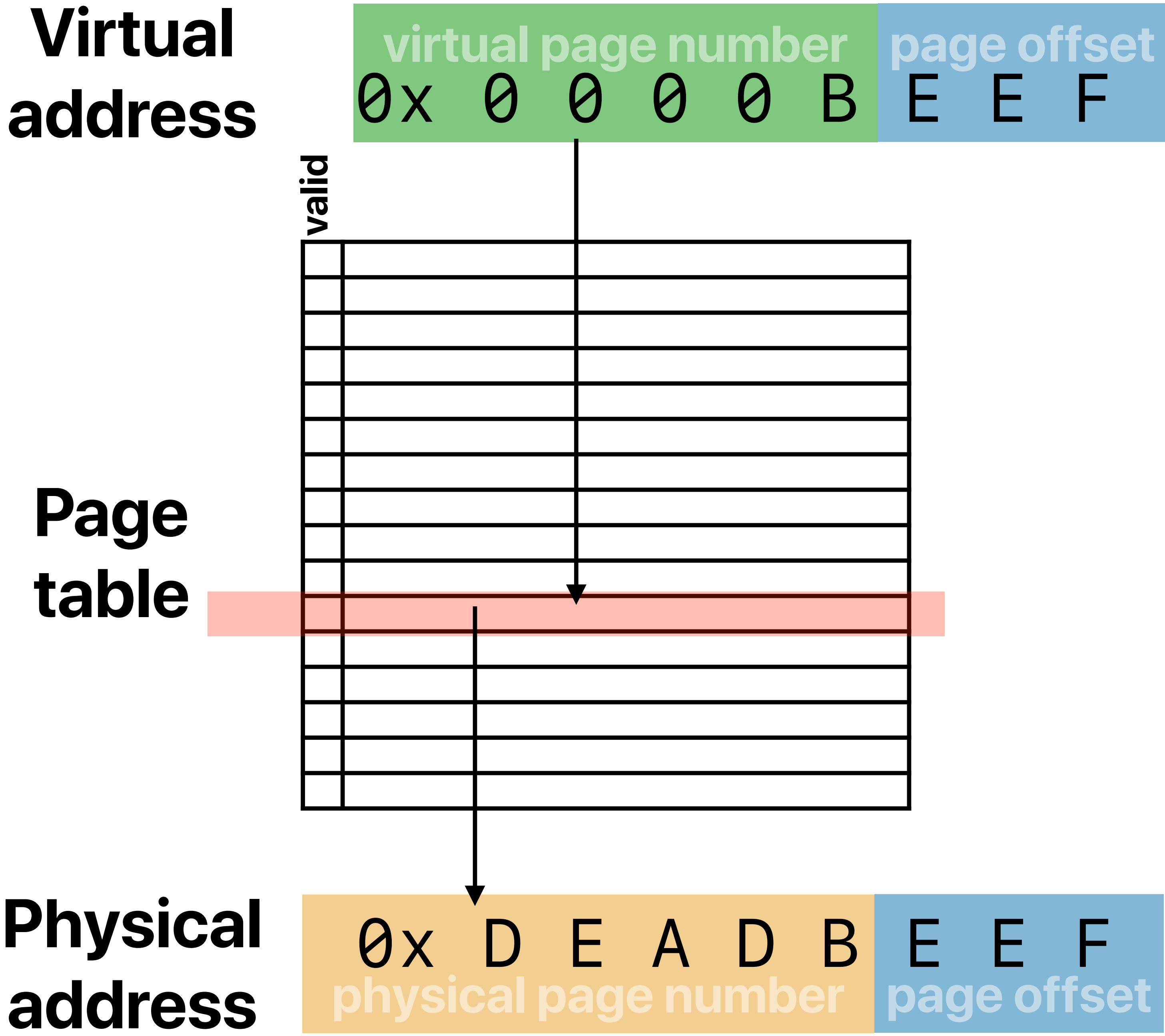
- Treating physical main memory as a “cache” of virtual memory
- The block size is the “page size”
- The page table is the “tag array”
- It’s a “fully-associate” cache — a virtual page can go anywhere in the physical main memory
- The storage serves as the lower level memory hierarchy for physical main memory

Takeaways: Virtual Memory

- Virtual memory is essential to support the success of software industry

Address translation

- Processor receives virtual addresses from the running code, main memory uses physical memory addresses
- Virtual address space is organized into "pages"
- The system references the **page table** to translate addresses
  - Each process has its own page table
  - The page table content is maintained by OS



https://www.pollev.com/hungweitseng close in 1:30

Size of page table

- Assume that we have **64-bit** virtual address space, each page is 4KB, each page table entry is 8 Bytes, what magnitude in size is the page table for a process?
  A. MB — 2^{20} Bytes
  B. GB — 2^{30} Bytes
  C. TB — 2^{40} Bytes
  D. PB — 2^{50} Bytes
  E. EB — 2^{60} Bytes

Size of page table

- Assume that we have **64-bit** virtual address space, each page is 4KB, each page table entry is 8 Bytes, what magnitude in size is the page table for a process?
  A. MB — 2^{20} Bytes
  B. GB — 2^{30} Bytes
  C. TB — 2^{40} Bytes
  [✓ D. PB — 2^{50} Bytes]
  E. EB — 2^{60} Bytes

$$\frac{2^{64}\ Bytes}{4\ KB} \times 8\ Bytes = 2^{55}\ Bytes = 32\ PB$$

If you still don't know why — you need to take CS202

Conventional page table

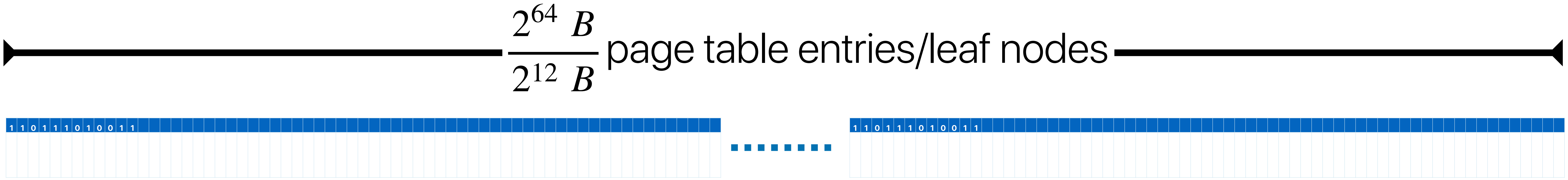
0x0 0xFFFFFFFFFFFFFFFF

Virtual Address Space

— must be consecutive in the physical memory

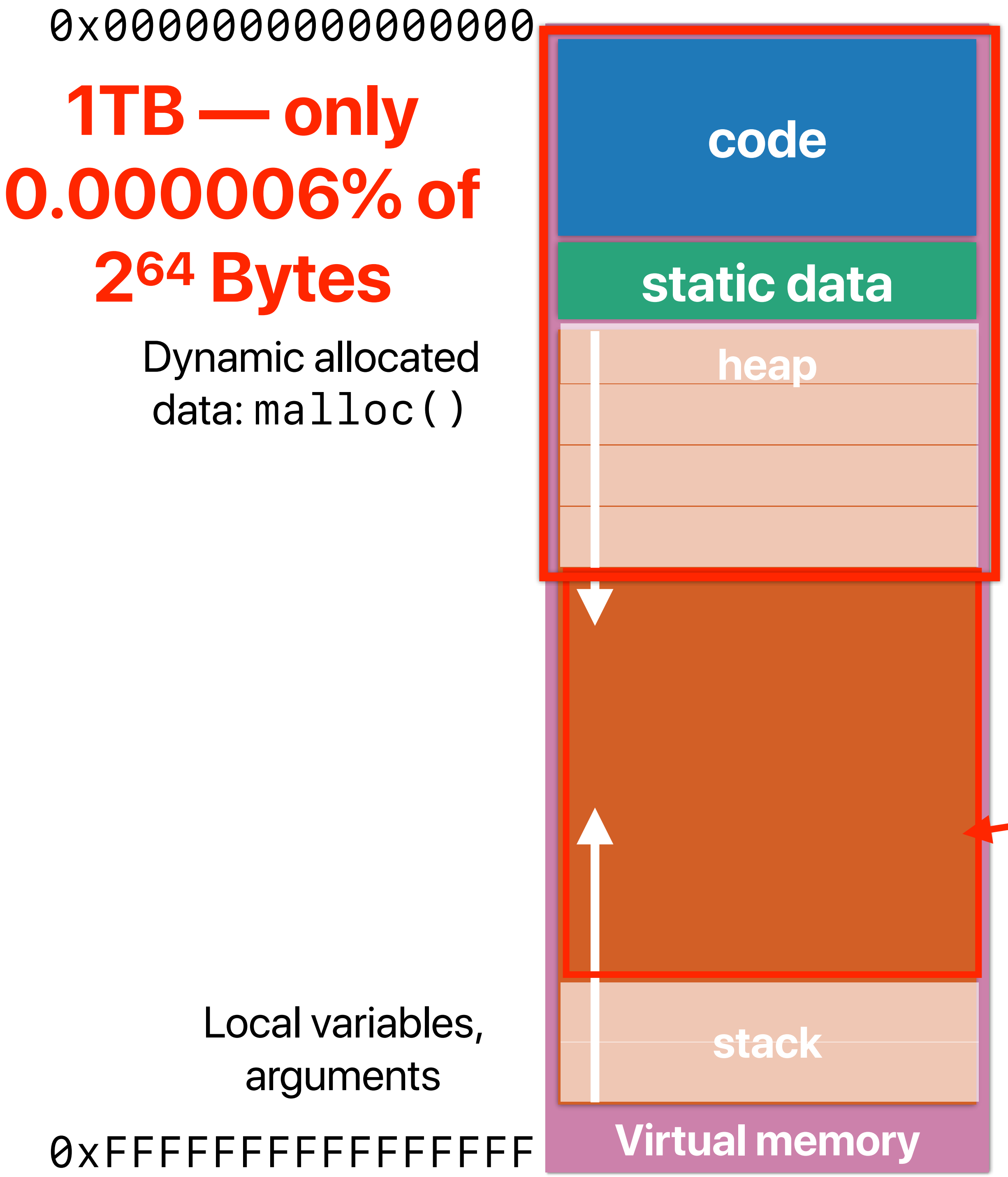
— need a big segment! — difficult to find a spot

— simply too big to fit in memory if address space is large!



$\frac{2^{64}\ B}{2^{12}\ B}$ page table entries/leaf nodes

Do we really need a large table?

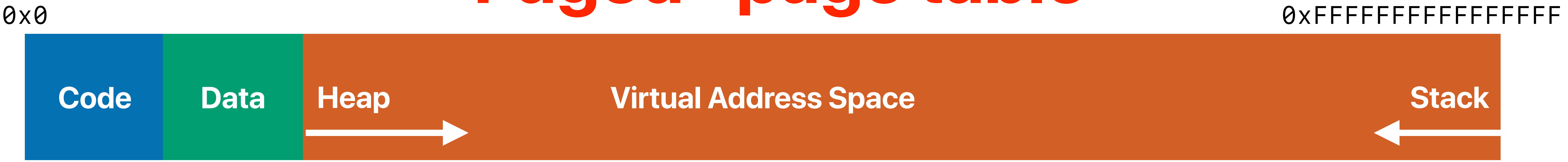


1TB — only 0.000006% of 2^{64} Bytes

Your program probably never uses this huge area!

If you still don't know why — you need to take CS202

“Paged” page table



Break up entries into pages!

Each of these occupies exactly a page

— $\frac{2^{12}\ B}{2^{3}\ B} = 2^{9}$ PTEs per node

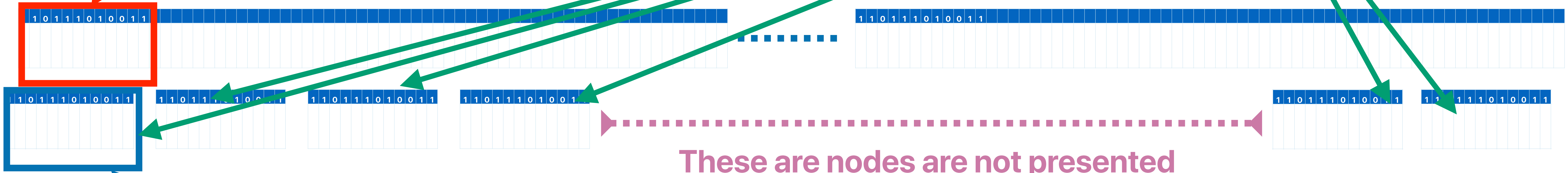
Question:

These nodes are spread out,

how to locate them in the memory?

Otherwise, you always need to find more

than one consecutive pages — difficult!

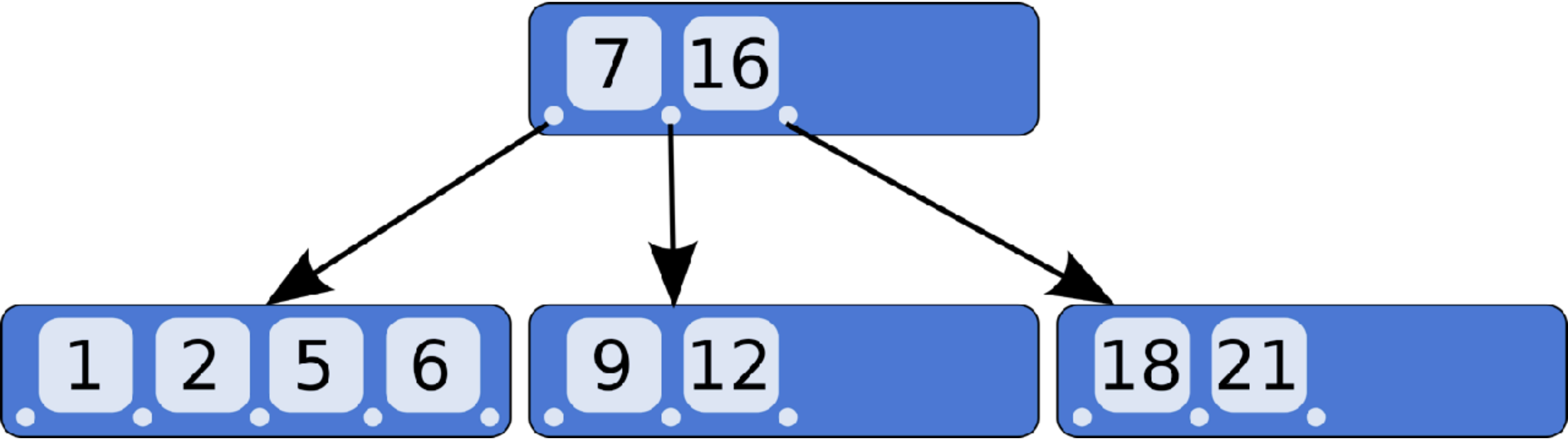


These are nodes are not presented

if they are not referenced at all — save space

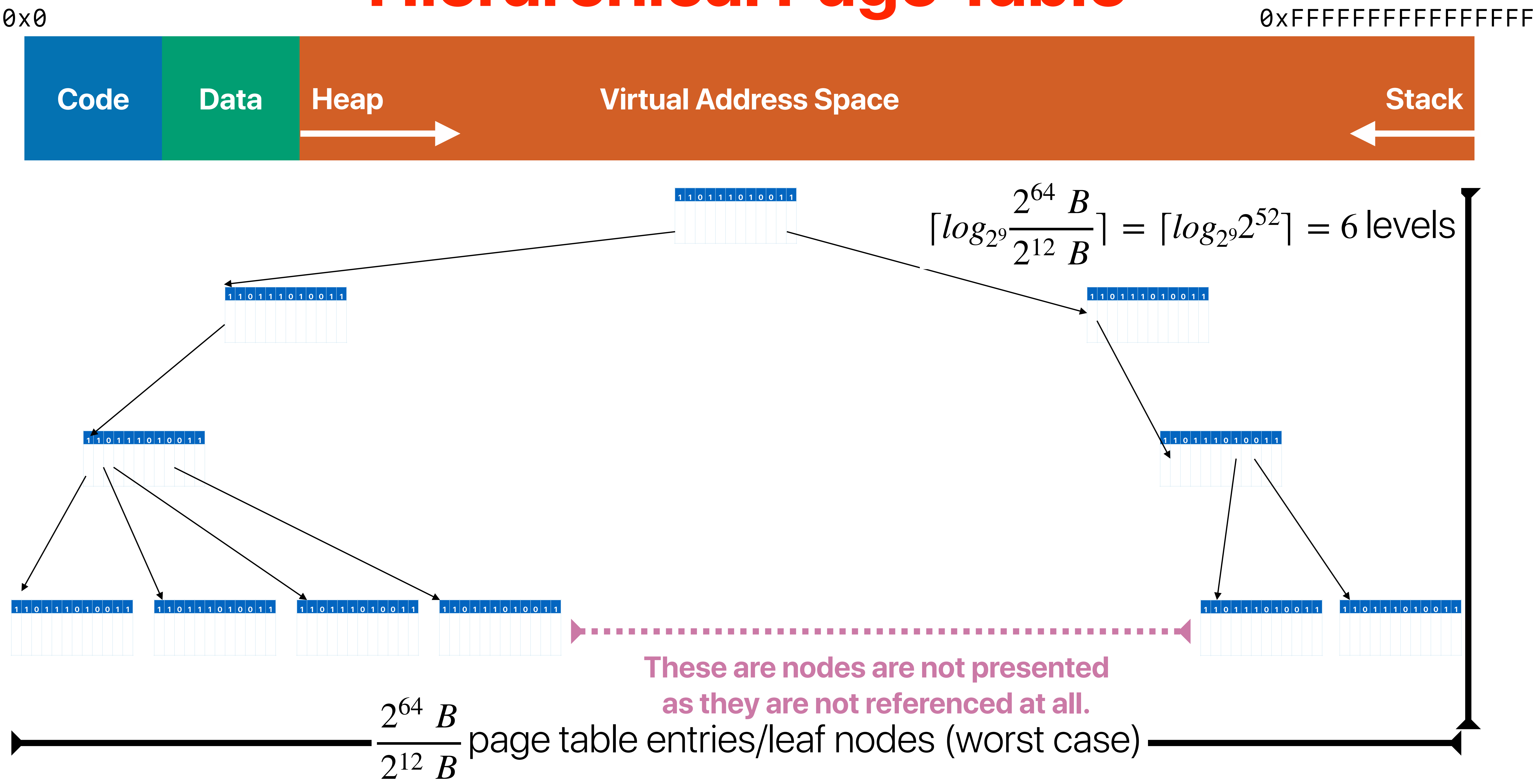
Allocate page table entry nodes “on demand”

B-tree



https://en.wikipedia.org/wiki/B-tree#/media/File:B-tree.svg

Hierarchical Page Table

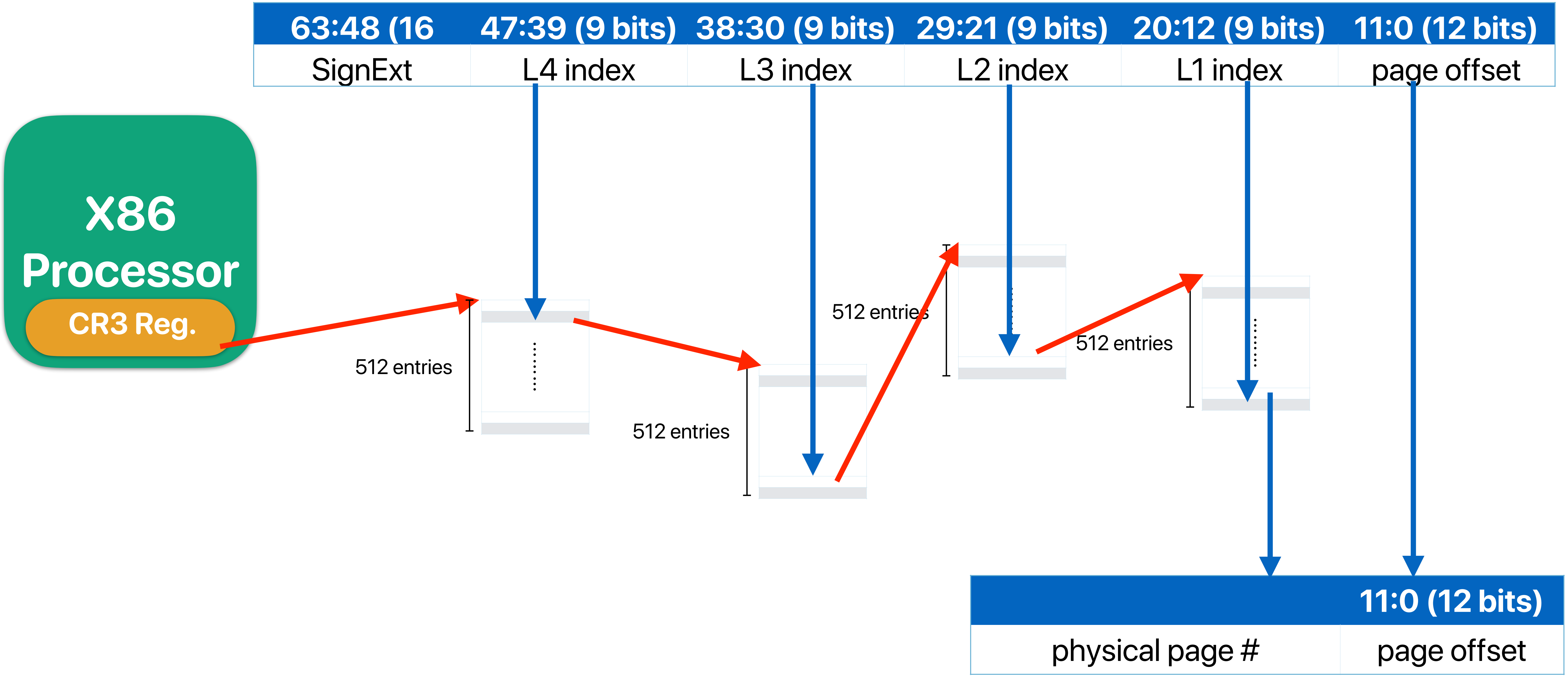


$\lceil log_{2^9}\frac{2^{64}\ B}{2^{12}\ B}\rceil = \lceil log_{2^9}2^{52}\rceil = 6$ levels

These are nodes are not presented as they are not referenced at all.

$\frac{2^{64}\ B}{2^{12}\ B}$ page table entries/leaf nodes (worst case)

Address translation in x86-64



63:48 (16	47:39 (9 bits)	38:30 (9 bits)	29:21 (9 bits)	20:12 (9 bits)	11:0 (12 bits)
SignExt	L4 index	L3 index	L2 index	L1 index	page offset

X86 Processor

CR3 Reg.

512 entries

512 entries

512 entries

512 entries

	11:0 (12 bits)
physical page #	page offset

Translation Caching: Skip, Don't Walk (the Page Table)
Thomas W. Barr, Alan L. Cox, Scott Rixner

Takeaways: Virtual Memory

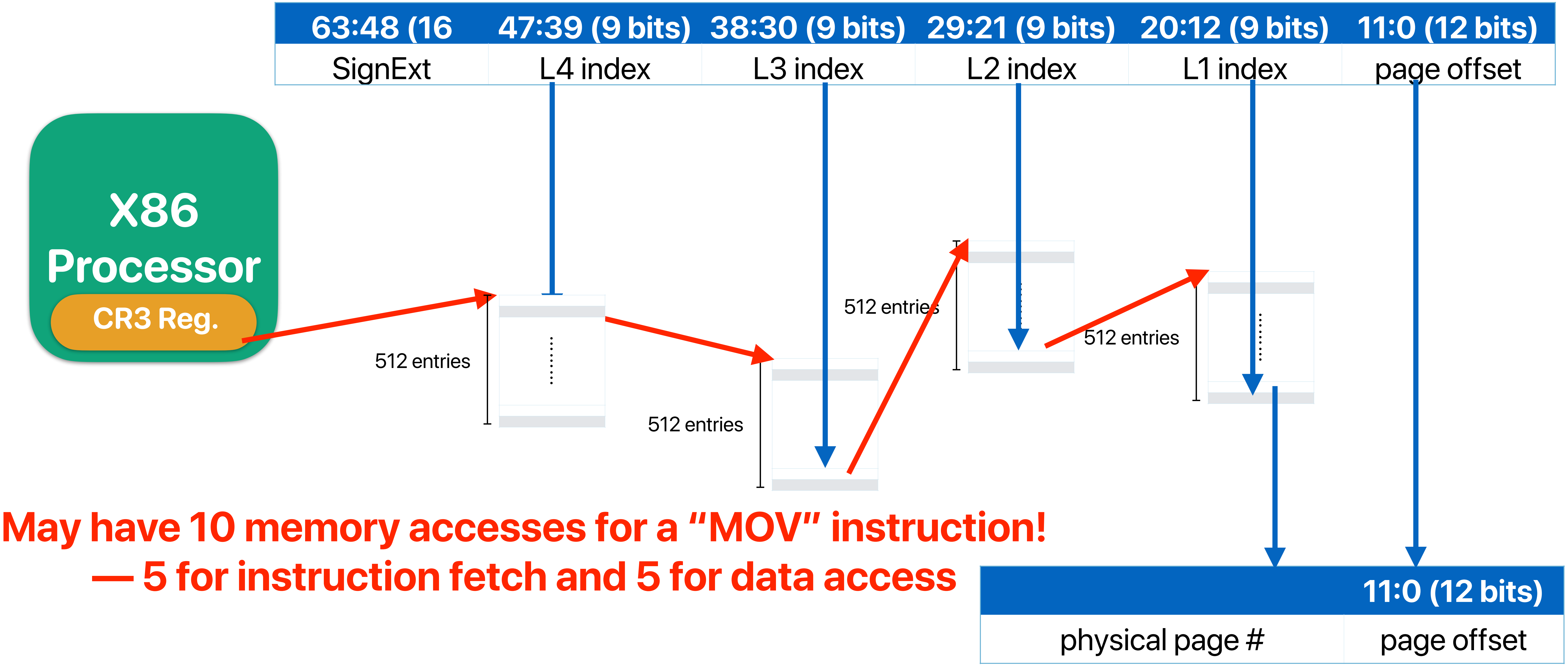
- Virtual memory is essential to support the success of software industry
- To reduce the page table size, we introduced hierarchical page table data structure

https://www.pollev.com/hungweitseng close in 1:30

When we have virtual memory…

- If an x86 processor supports virtual memory through the basic format of the page table as shown in the previous slide, how many memory accesses can a **mov** instruction that access data memory once incur?
  A. 2
  B. 4
  C. 6
  D. 8
  E. 10

Address translation in x86-64



63:48 (16	47:39 (9 bits)	38:30 (9 bits)	29:21 (9 bits)	20:12 (9 bits)	11:0 (12 bits)
SignExt	L4 index	L3 index	L2 index	L1 index	page offset

	11:0 (12 bits)
physical page #	page offset

May have 10 memory accesses for a "MOV" instruction!
— 5 for instruction fetch and 5 for data access

When we have virtual memory...

- If an x86 processor supports virtual memory through the basic format of the page table as shown in the previous slide, how many memory accesses can a **`mov`** instruction that access data memory once incur?

  A. 2

  B. 4

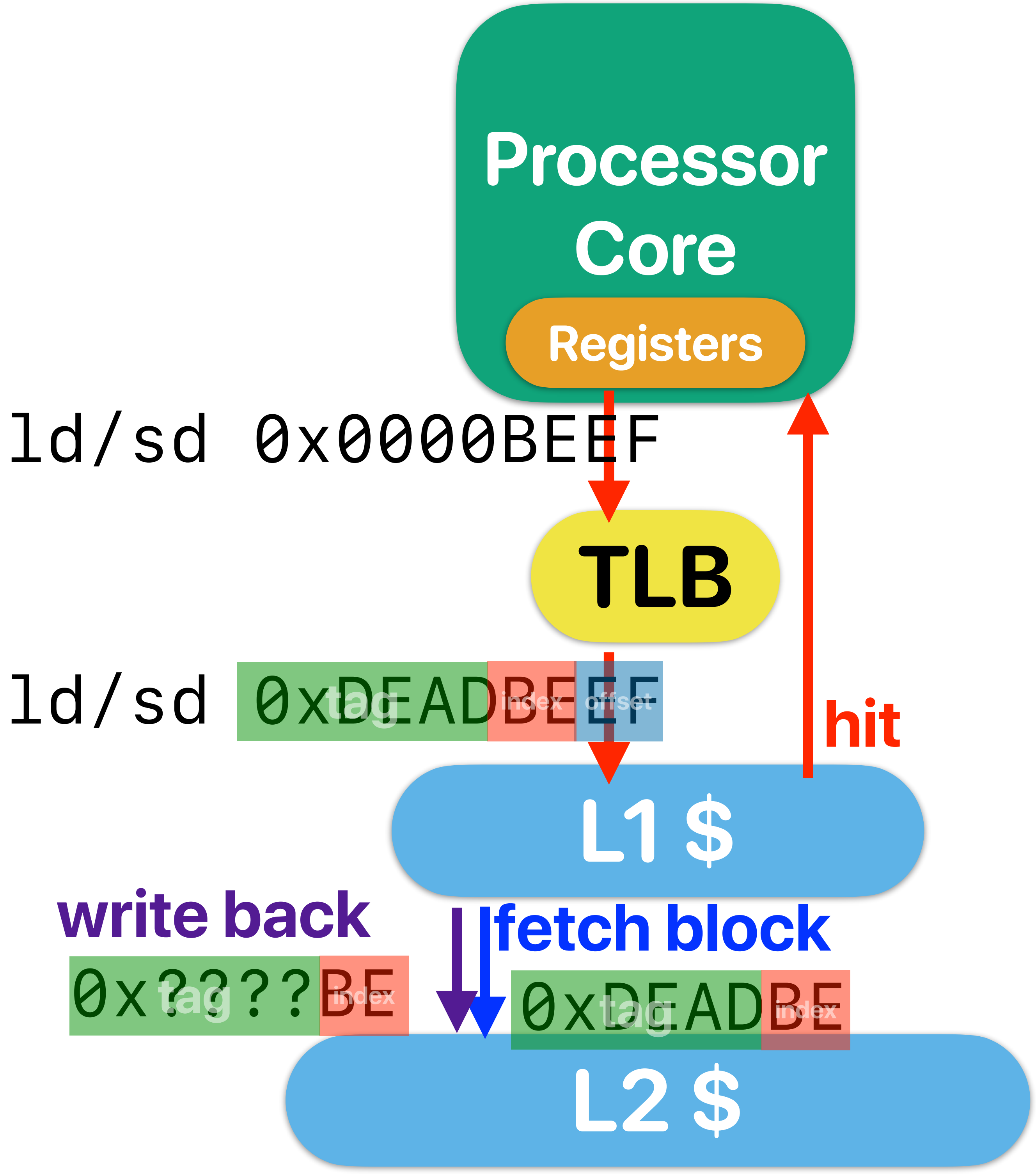
  C. 6

  D. 8

  [✓ E. 10]

Avoiding the address translation overhead

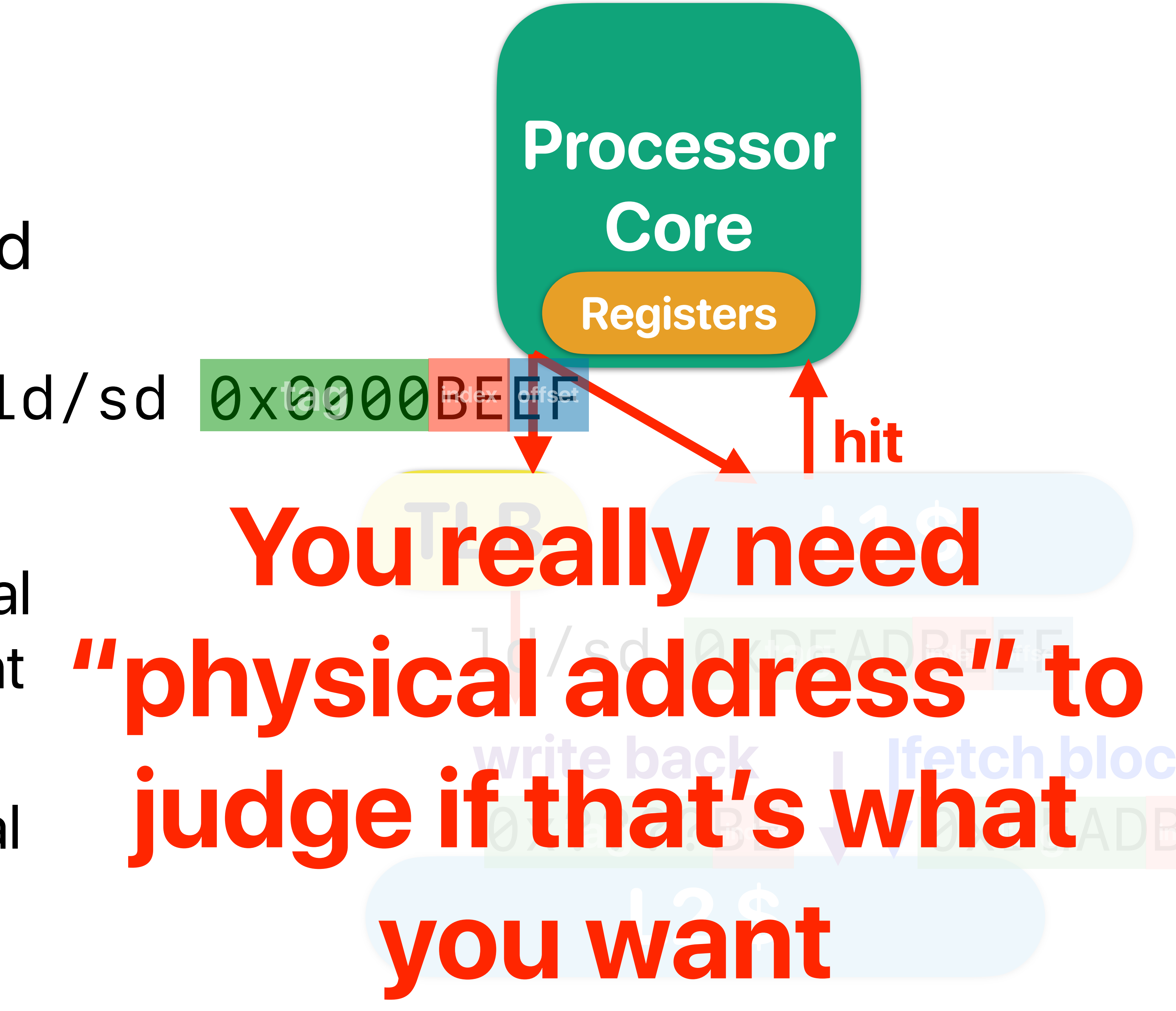
TLB: Translation Look-aside Buffer



- TLB — a small SRAM stores frequently used page table entries
- Good — A lot faster than having everything going to the DRAM
- Bad — Still on the critical path

TLB + Virtual cache

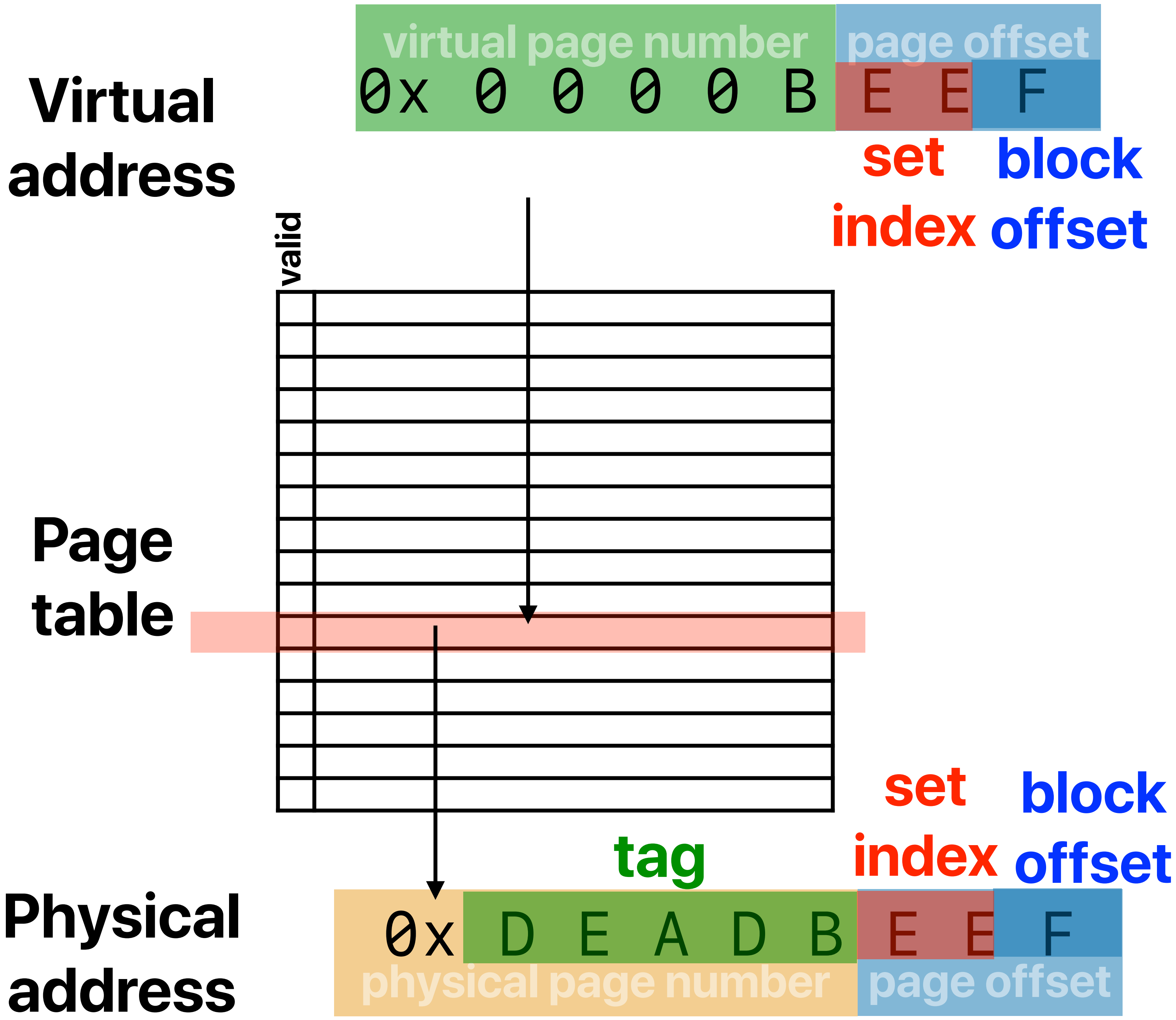
- L1 $ accepts virtual address — you don't need to translate
- Good — you can access both TLB and L1-$ at the same time and physical address is only needed if L1-$ misses
- Bad — it doesn't work in practice
  - Many applications have the same virtual address but should be pointing different **physical addresses**
  - An application can have "aliasing virtual addresses" pointing to the same **physical address**



You really need "physical address" to judge if that's what you want

Virtually indexed, physically tagged cache

- Can we find physical address directly in the virtual address
  - — Not everything — but the page offset isn't changing!
- Can we indexing the cache using the "partial physical address"?
  - — Yes — Just make set index + block set to be exactly the page offset



Virtually indexed, physically tagged cache

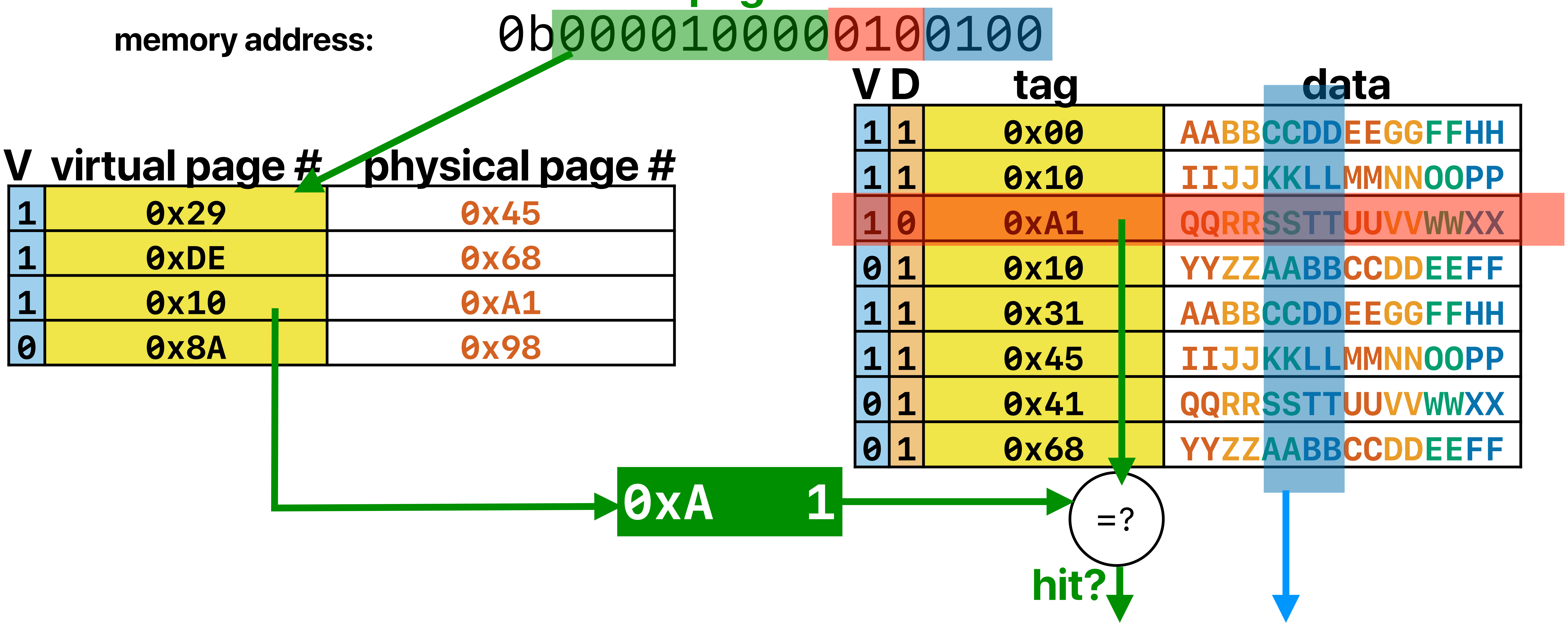
memory address: 0x0 8 2 4

set index | block offset

memory address: 0b0000100000100100

virtual page # | index | offset

0b000010000 | 010 | 0100



V	virtual page #	physical page #
1	0x29	0x45
1	0xDE	0x68
1	0x10	0xA1
0	0x8A	0x98

V	D	tag	data
1	1	0x00	AABBCCDDEEGGFFHH
1	1	0x10	IIJJKKLLMMNNOOPP
1	0	0xA1	QQRRSSTTUUVVWWXX
0	1	0x10	YYZZAABBCCDDEEFF
1	1	0x31	AABBCCDDEEGGFFHH
1	1	0x45	IIJJKKLLMMNNOOPP
0	1	0x41	QQRRSSTTUUVVWWXX
0	1	0x68	YYZZAABBCCDDEEFF

Virtually indexed, physically tagged cache

- If page size is 4KB —

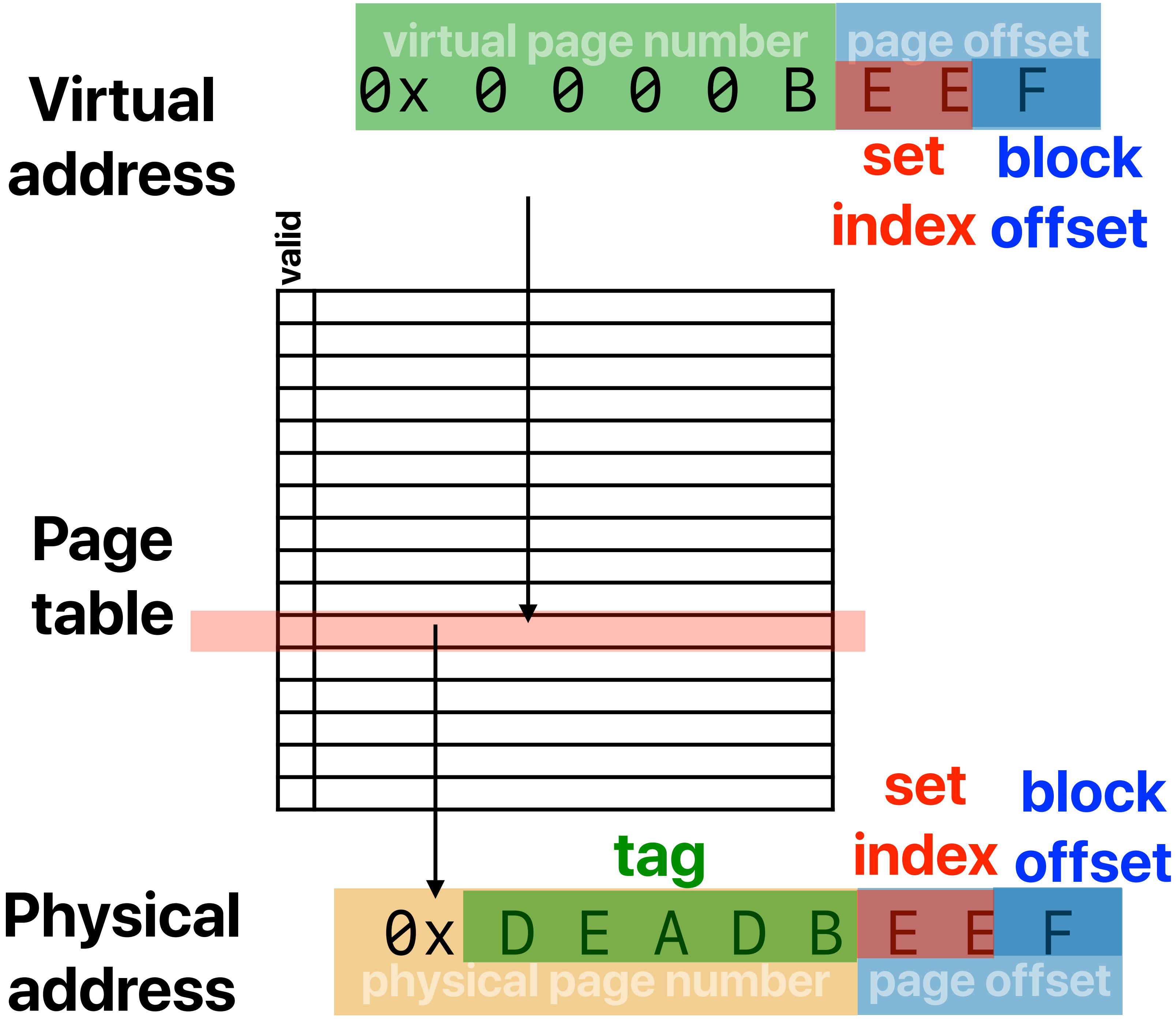
$$lg(B) + lg(S) = lg(4096) = 12$$

$$C = ABS$$

$$C = A \times 2^{12}$$

$$if\ A = 1$$

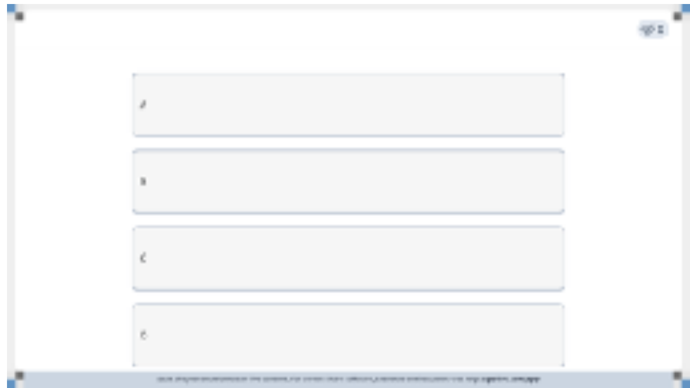
$$C = 4KB$$



https://www.pollev.com/hungweitseng close in 1:30

Virtual indexed, physical tagged cache limits the cache size

- If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the operating system use 4K pages.
  A. 32B blocks, 2-way
  B. 32B blocks, 4-way
  C. 64B blocks, 4-way
  D. 64B blocks, 8-way



Virtual indexed, physical tagged cache limits the cache size

- If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the operating system use 4K pages.

  A. 32B blocks, 2-way

  B. 32B blocks, 4-way

  C. 64B blocks, 4-way

  [✓ D. 64B blocks, 8-way]

Exactly how Core i7 9th generation configures its own cache

$$lg(B) + lg(S) = lg(4096) = 12$$

$$C = ABS$$

$$32KB = A \times 2^{12}$$

$$A = 8$$

Takeaways: Virtual Memory

- Virtual memory is essential to support the success of software industry
- To reduce the page table size, we introduced hierarchical page table data structure
- Virtually-indexed, physically tagged cache provides the efficiency for accessing cache and TLB together — but limited cache design

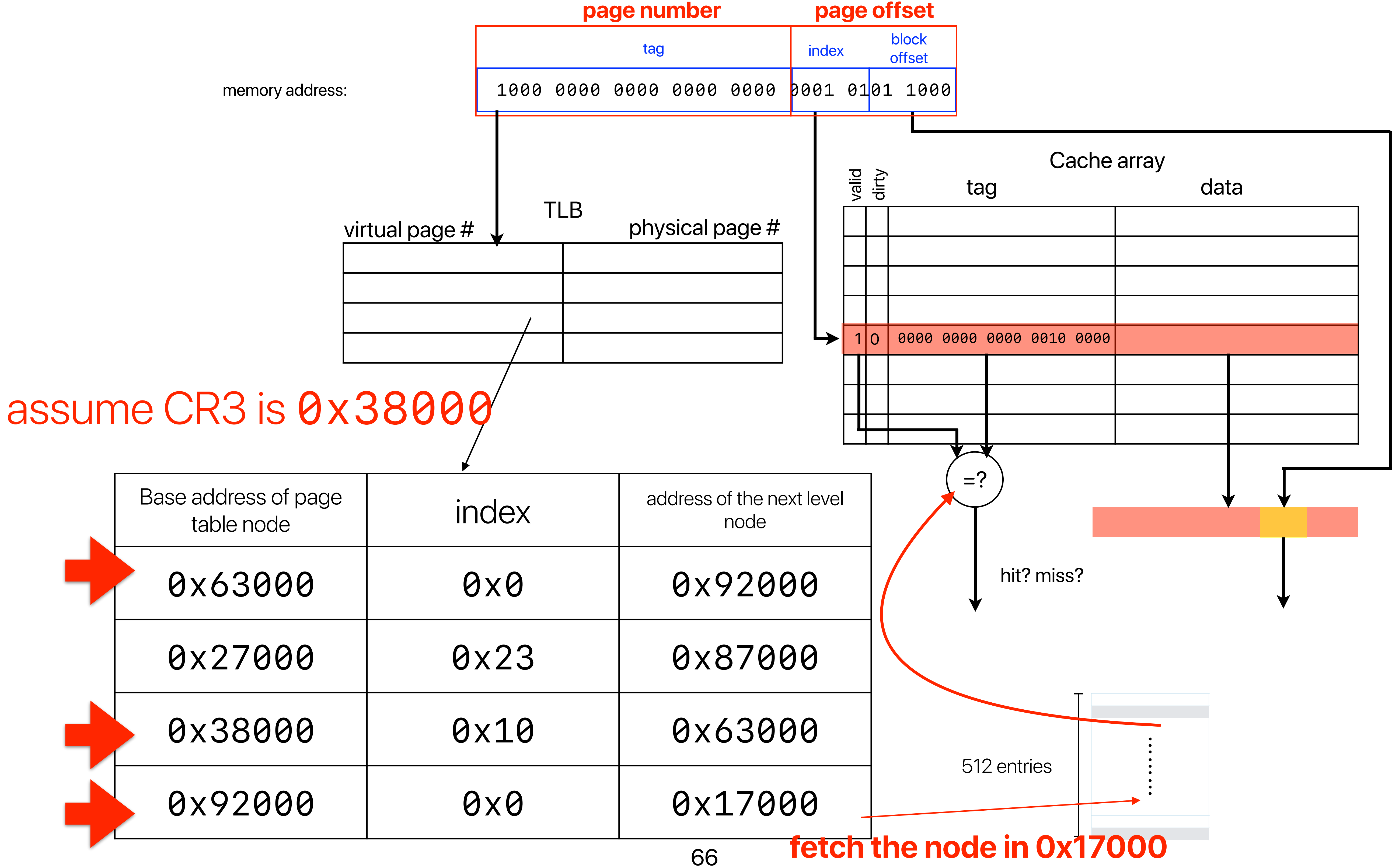
Translation Caching: Skip, Don’t Walk (the Page Table)

Thomas W. Barr, Alan L. Cox, Scott Rixner

Why should we care about this paper?

- TLB miss is expensive
  - You have to walk through multiple nodes in the hierarchical page table
  - Each node is a memory access — 100 ns
- Modern processors use memory management units (MMUs)
  - MMUs have caches, but not optimized for the timing critical TLB miss
  - Page table caches
  - Translational caches

Page table caches



assume CR3 is 0x38000

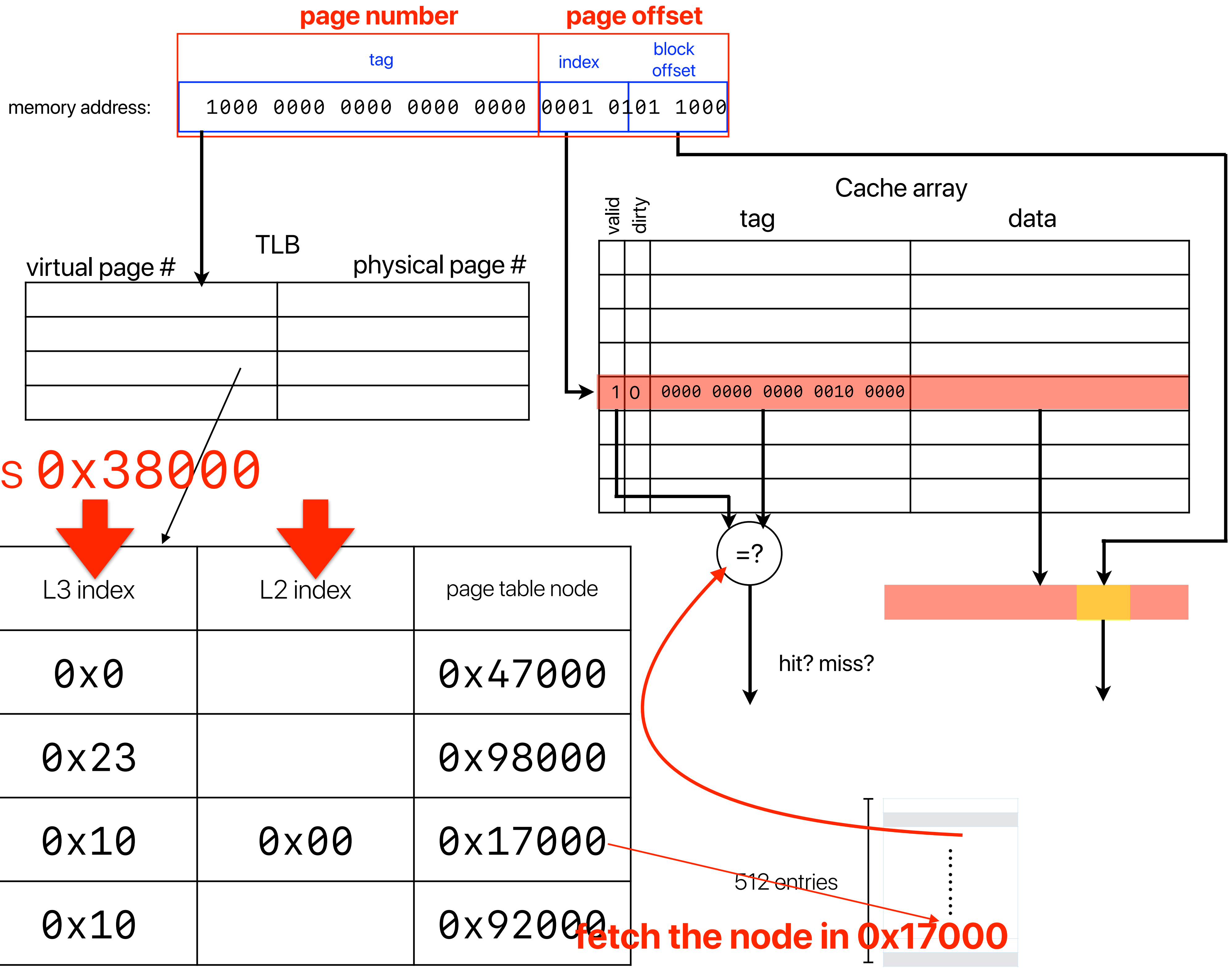
Base address of page table node	index	address of the next level node
0x63000	0x0	0x92000
0x27000	0x23	0x87000
0x38000	0x10	0x63000
0x92000	0x0	0x17000

fetch the node in 0x17000

Page table caches

- PTC caches the addresses of “page table nodes”
- PTC uses the physical address of page table nodes as the index
    - Unified page table cache (UPTC)
    - Split page table cache (SPTC)
        - Each page level get a private cache location

Translation cache



assume CR3 is 0x38000

L4 index	L3 index	L2 index	page table node
0x0	0x0		0x47000
0x30	0x23		0x98000
0x0	0x10	0x00	0x17000
0x0	0x10		0x92000

fetch the node in 0x17000

Translation caches

- Indexed by the prefix of the requesting virtual address
  - Split translational cache (STC)
  - Unified translational cache (UTC)
  - Translational-path Cache (TPC)
- Pros:
  - Allowing each level lookup to perform independently, in parallel
- Cons:
  - Less space efficient

Takeaways: Virtual Memory

- Virtual memory is essential to support the success of software industry
- To reduce the page table size, we introduced hierarchical page table data structure
- Virtually-indexed, physically tagged cache provides the efficiency for accessing cache and TLB together — but limited cache design
- Page table caches & translation caching can help reducing the TLB miss penalty

Announcement

- Assignment #2 due **this evening**
- Reading quiz #5 due **next Tuesday** before the lecture
- Assignment #3 and Programming Assignment #2 released
- Midterm and final exam will be online
  - Only be allowed at the same time as we scheduled
  - 2/13 2p-3:20p on gradescope

Computer
Science &
Engineering

203

つづく