

Virtual Memory

Hung-Wei Tseng

Recap: 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;
}
```

```
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>
```

```
//#define dim 32768
```

```
//#define dim 49152
```

```
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);
```

File memory_allocation.c not changed so no update needed

BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual_memory] -bunny- ./memory_allocation

BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual_memory] -bunny-

Recap: Let's dig into this code

```
#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;
}
```

Demo revisited

```
sleep(10);  
fprintf(stderr, "\nProcess %d: Value of a is %1f and address of a is %p\n", (int) getpid(), a, &a);  
return 0;  
}
```

File virtualization.c not changed so no update needed

BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual_memory] -bunny- make

gcc -O3 virtualization.c -o virtualization

BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual_memory] -bunny- ./virtualization 4

Process 19719: Value of a is 1671139616.000000 and address of a is 0x104967050

Process 19720: Value of a is 1671156423.000000 and address of a is 0x104967050

Process 19718: Value of a is 1671122809.000000 and address of a is 0x104967050

Process 19721: Value of a is 1671173230.000000 and address of a is 0x104967050

Different values

Process 19719: Value of a is 1671139616.000000 and address of a is 0x104967050

Process 19721: Value of a is 1671173230.000000 and address of a is 0x104967050

Process 19720: Value of a is 1671156423.000000 and address of a is 0x104967050

Process 19718: Value of a is 1671122809.000000 and address of a is 0x104967050

BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual_memory] -bunny-

Different values are
preserved

The same memory
address!

Recap: Demo revisited

```
#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]);
```

```
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
```

```
    srand((int)time(NULL)+(int)getpid());
```

```
    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;
```

```
}
```

&a = 0x601090

Process A

Process B

**Process A's
Virtual
Memory Space**

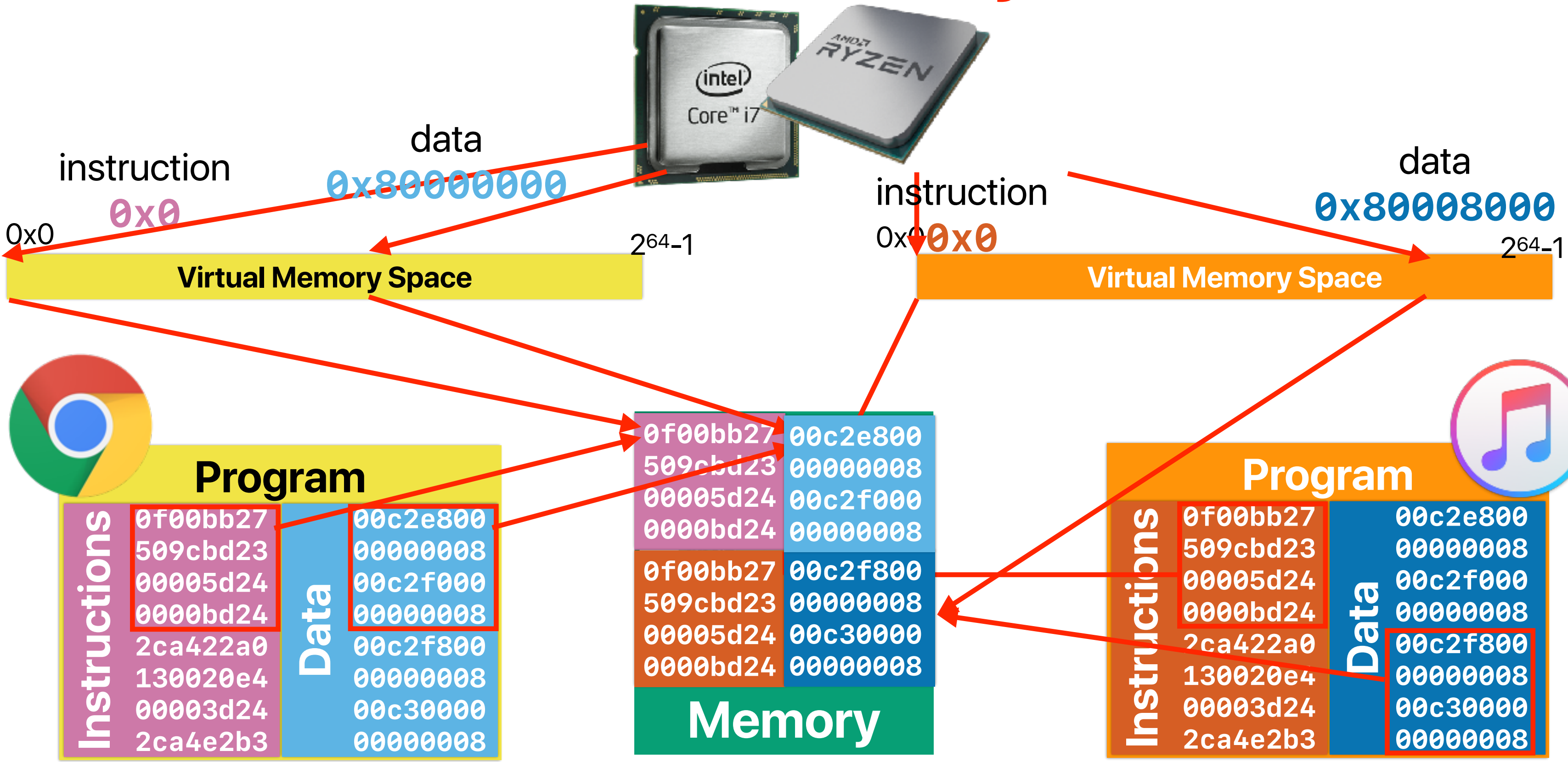
**Process B's
Virtual
Memory Space**

Outline

- Virtual memory
- Architectural support for virtual memory


Virtual Memory

Virtual memory



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**"



Processor Core

Registers

1

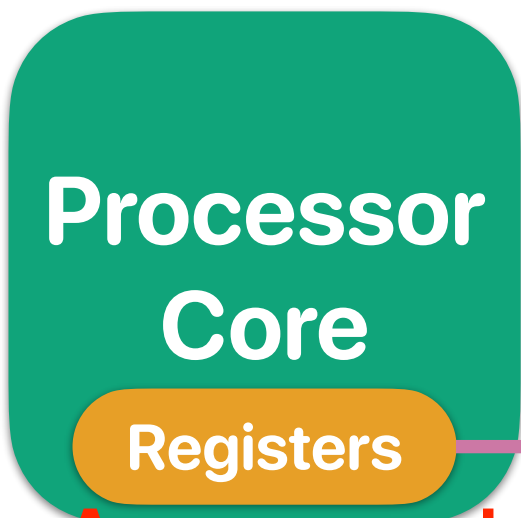
Registers

load 0x0009

Page table

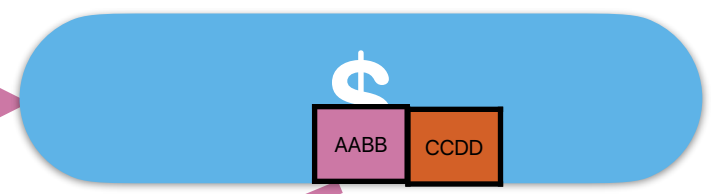
Page #1 Main memory (DRAM)





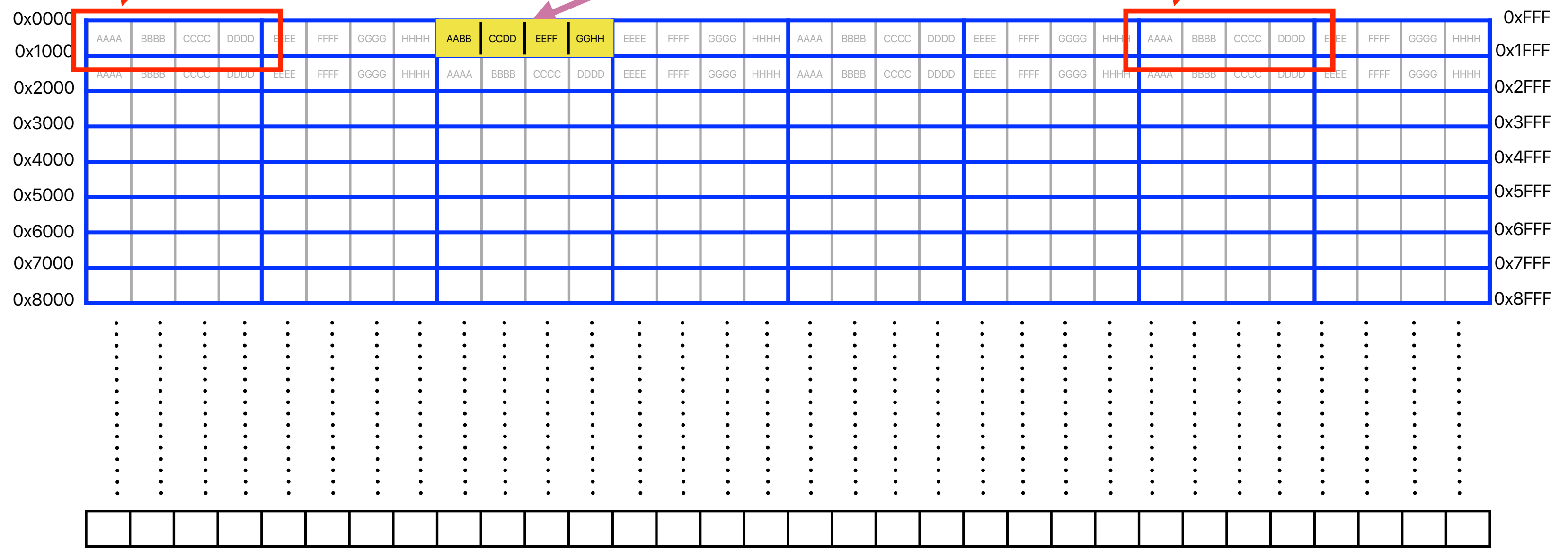
Recap: To capture "spatial" locality, \$ fetch a "block"

lw 0x0024



Assume each block is 16 bytes

"Logically" partition memory space into "blocks"



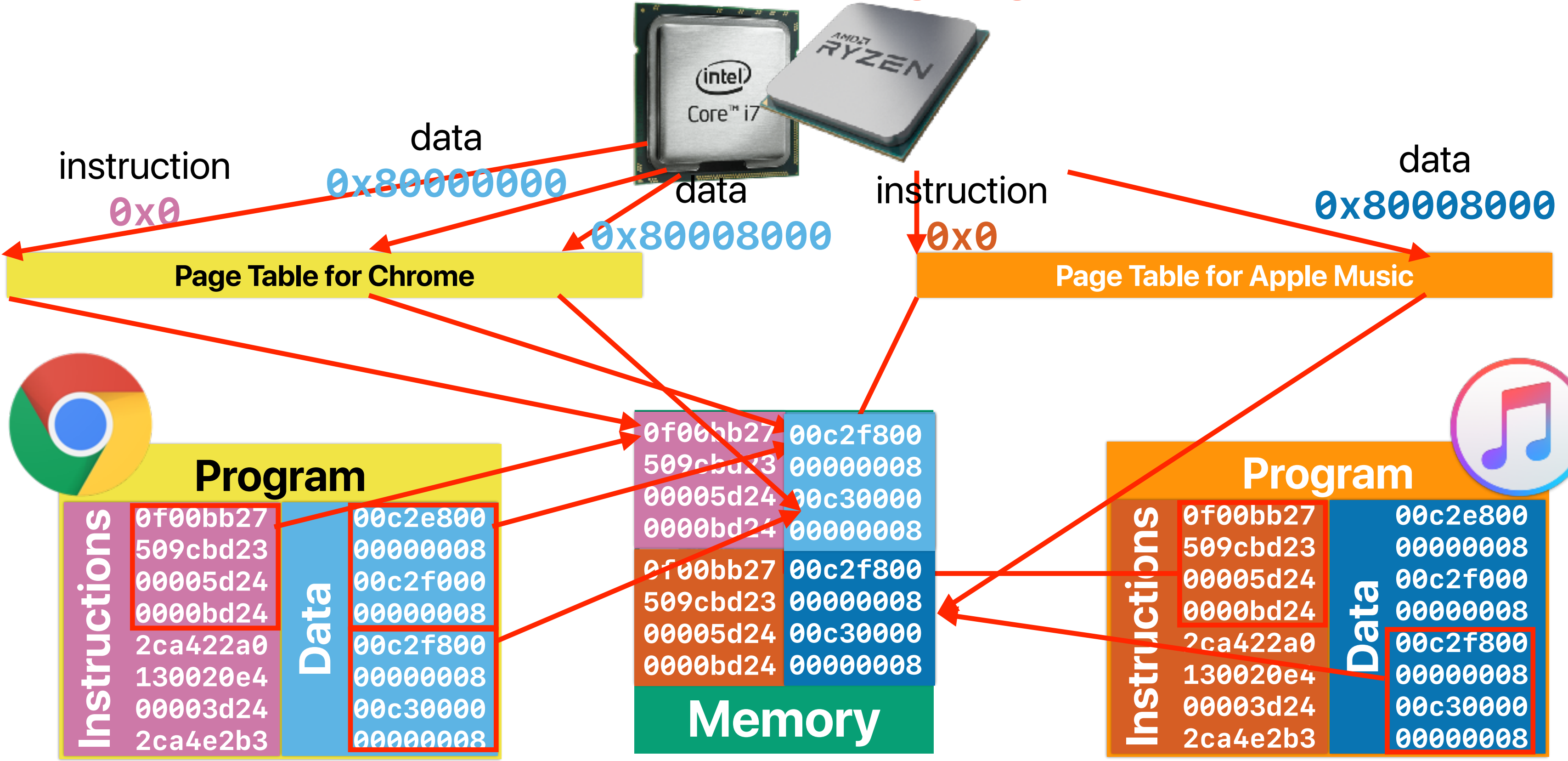
Why Virtual memory?

- 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 though their total memory demand is larger than the installed physical memory

Demand paging

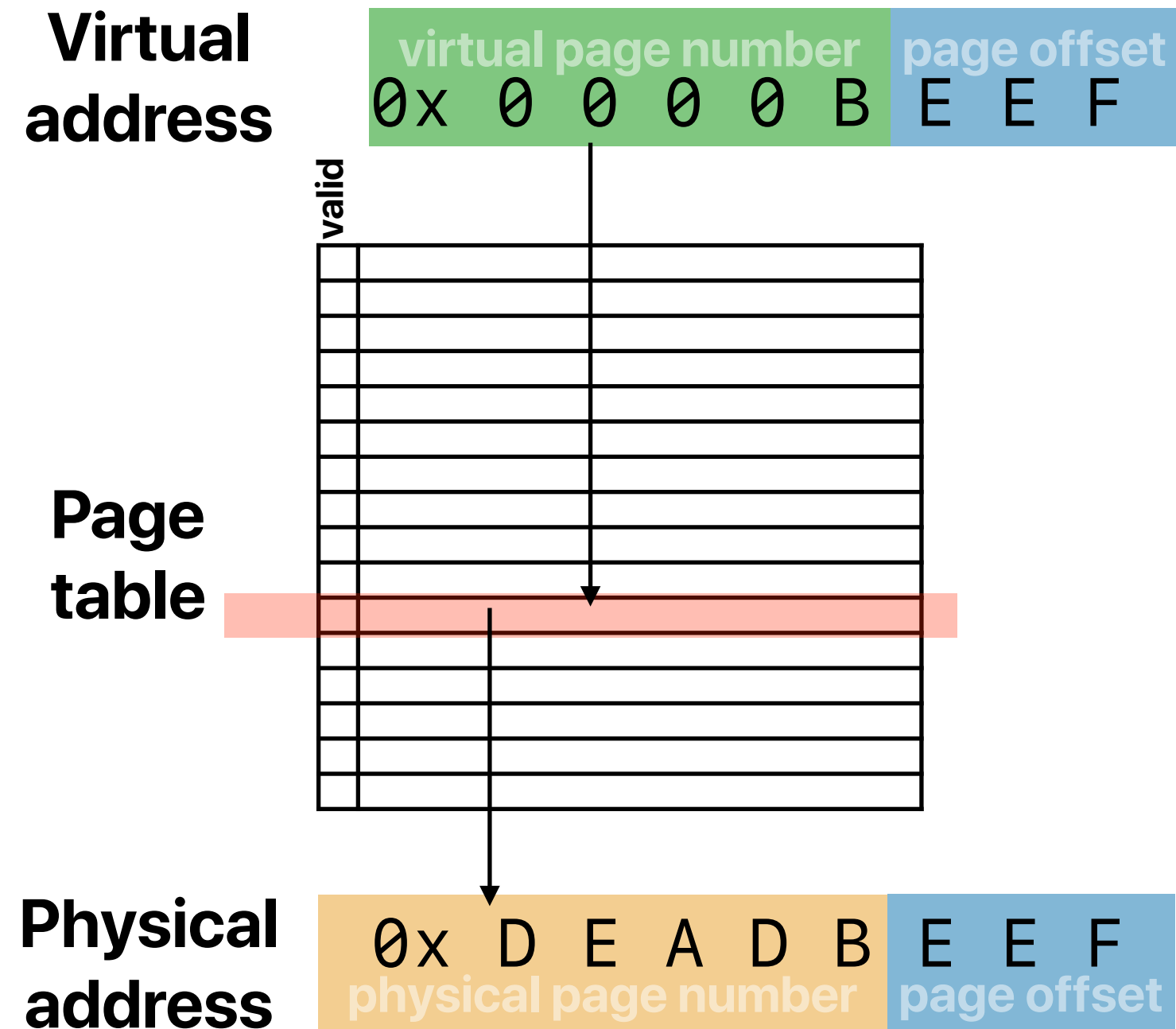
- Treating physical main memory as a “cache” of virtual memory
- The block size is the “page size”
- The page table is the “tag array”
- It’s a “fully-associate” cache — a virtual page can go anywhere in the physical main memory

Demand paging



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



Conventional page table

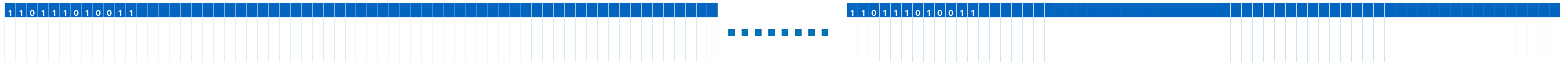
0x0

0xFFFFFFFFFFFFFFFF

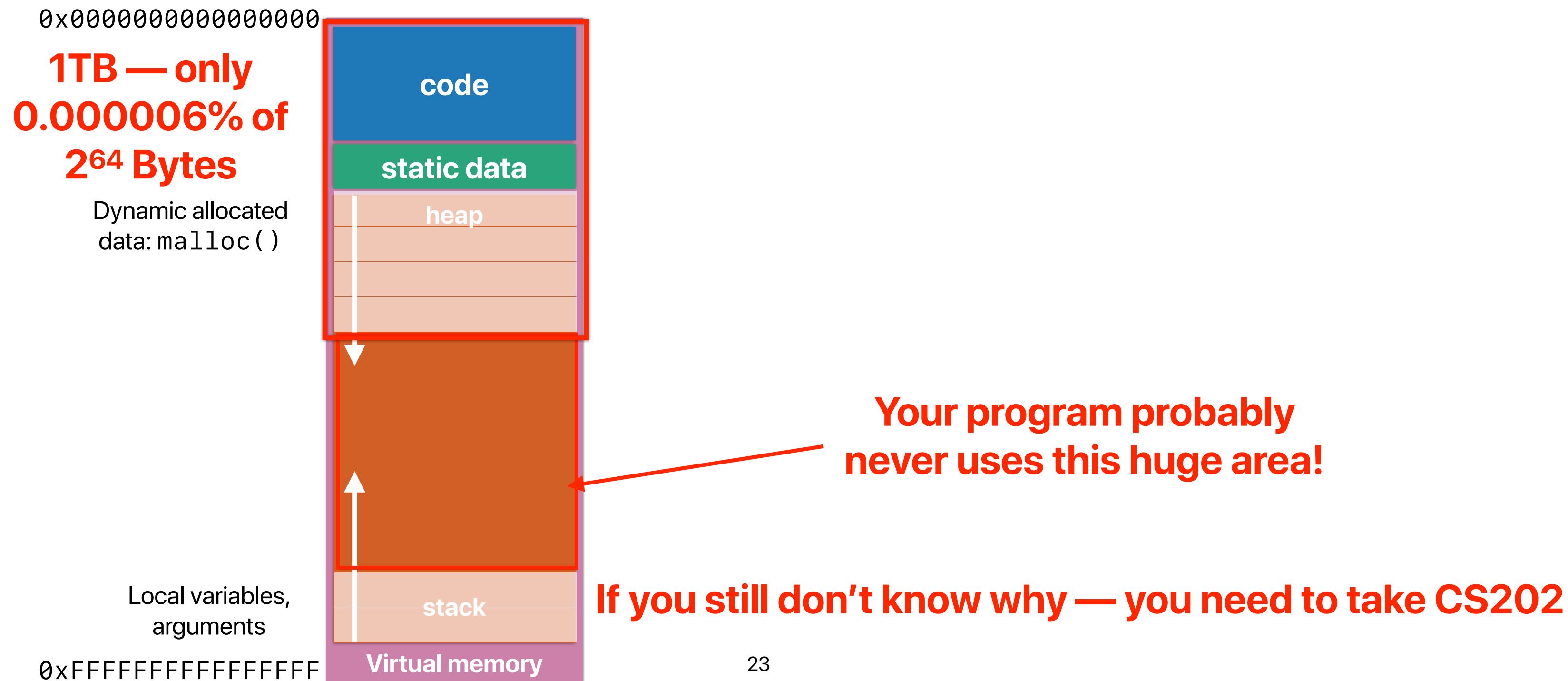
Virtual Address Space

- 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!

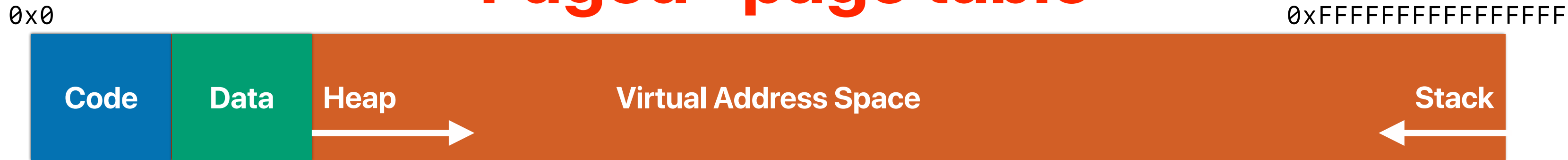
$\frac{2^{64} B}{2^{12} B}$ page table entries/leaf nodes



Do we really need a large table?



"Paged" page table

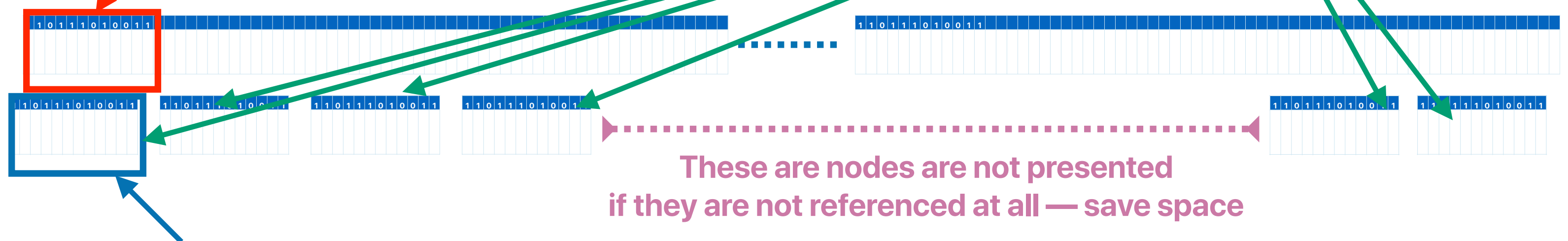


Break up entries into pages!
Each of these occupies exactly a page

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

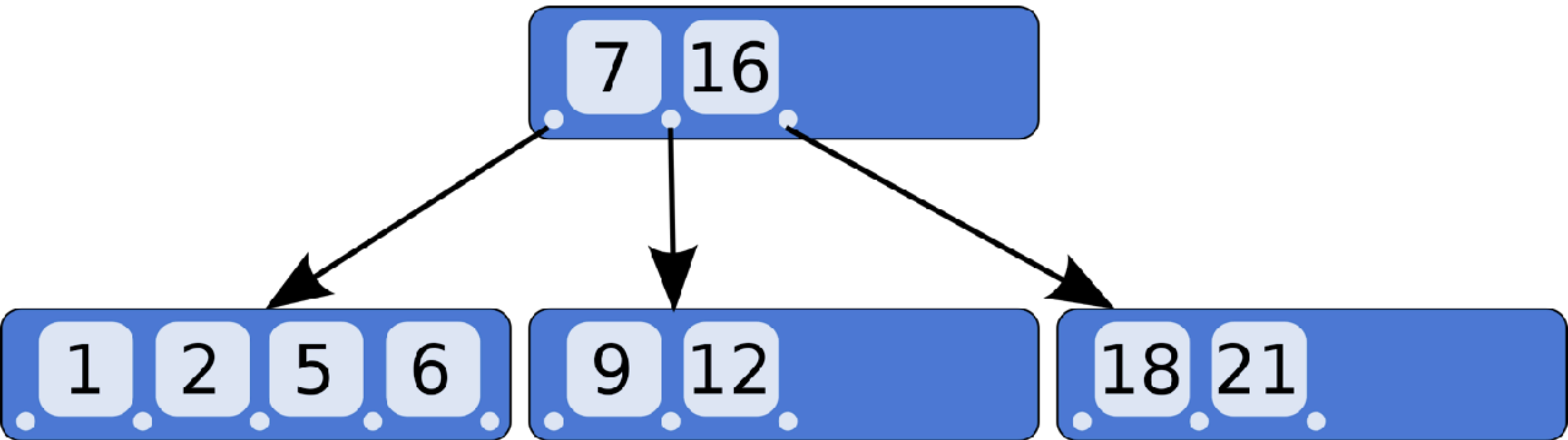
Otherwise, you always need to find more than one consecutive pages — difficult!

Question:
These nodes are spread out,
how to locate them in the memory?



Allocate page table entry nodes "on demand"

B-tree

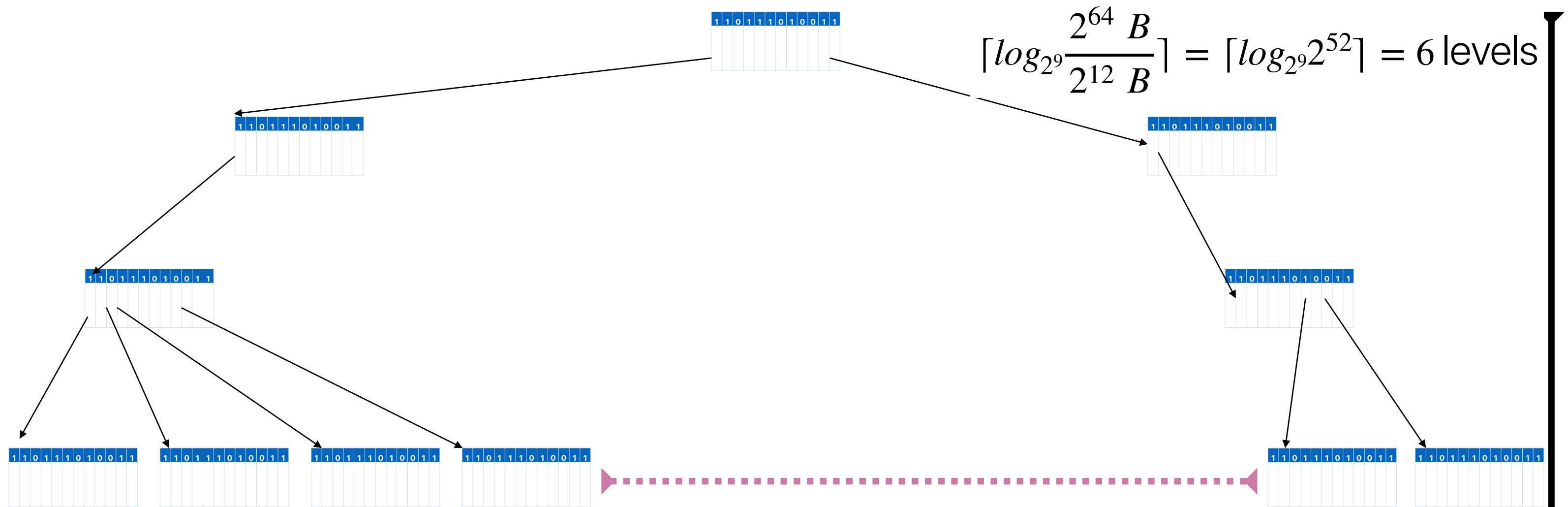
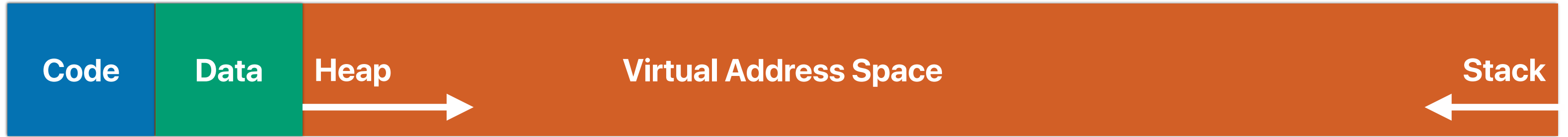


<https://en.wikipedia.org/wiki/B-tree#/media/File:B-tree.svg>

Hierarchical Page Table

0x0

0xFFFFFFFFFFFFFFFF

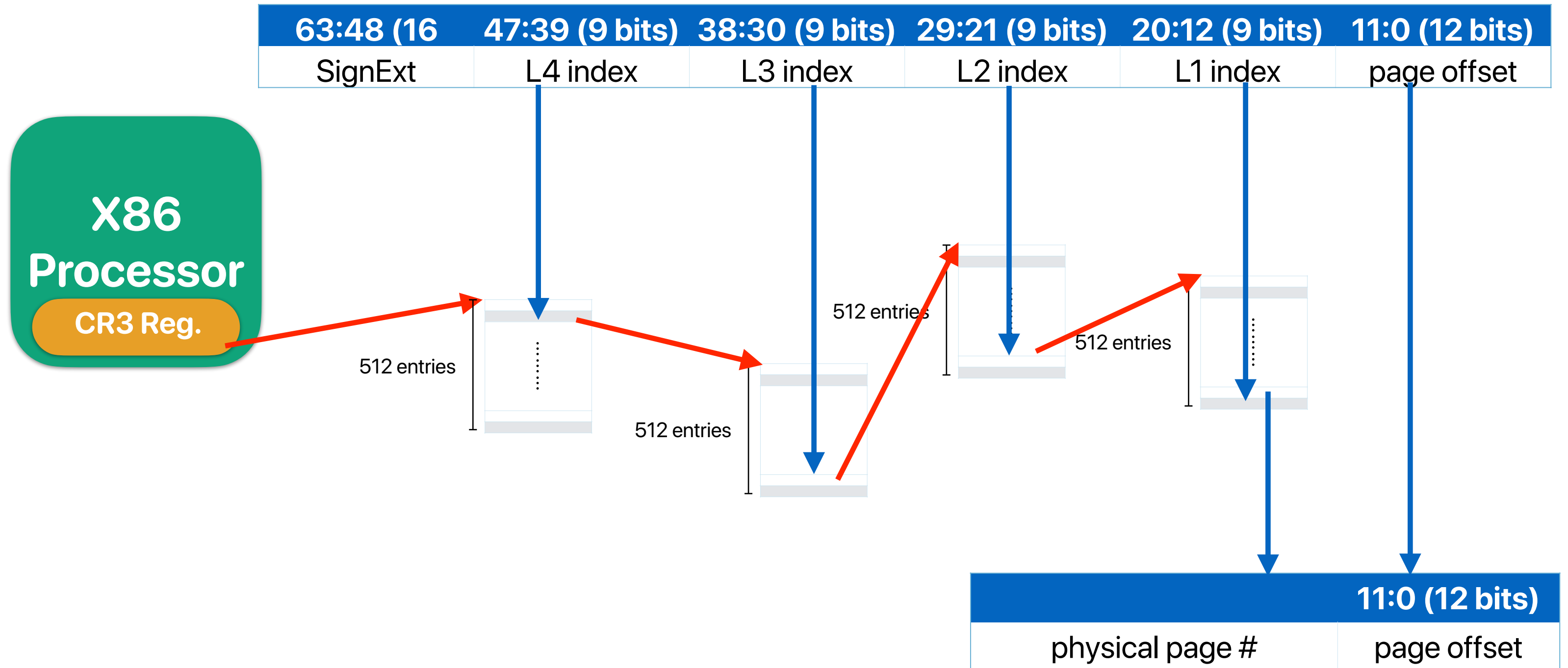


$$\lceil \log_2 \frac{2^{64} B}{2^{12} B} \rceil = \lceil \log_2 2^{52} \rceil = 6 \text{ levels}$$

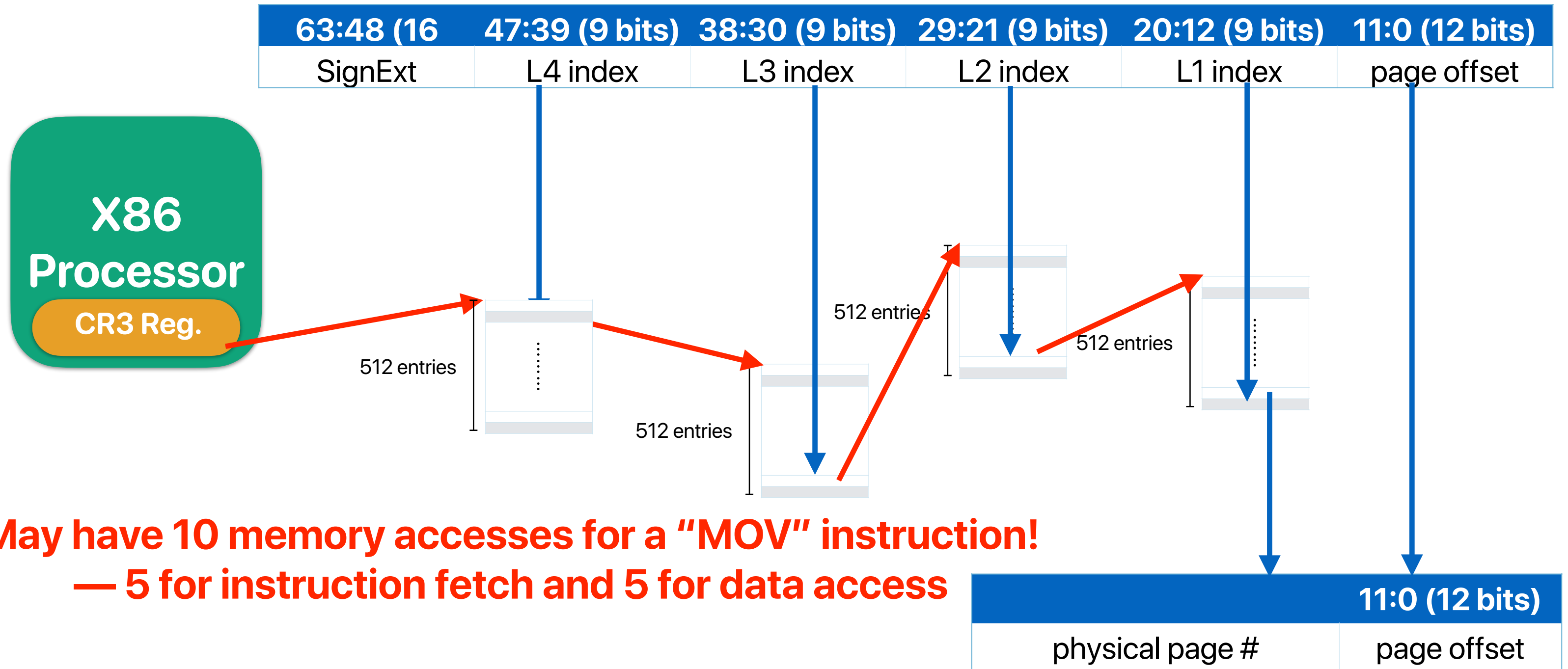
These are nodes are not presented
as they are not referenced at all.

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

Address translation in x86-64



Address translation in x86-64

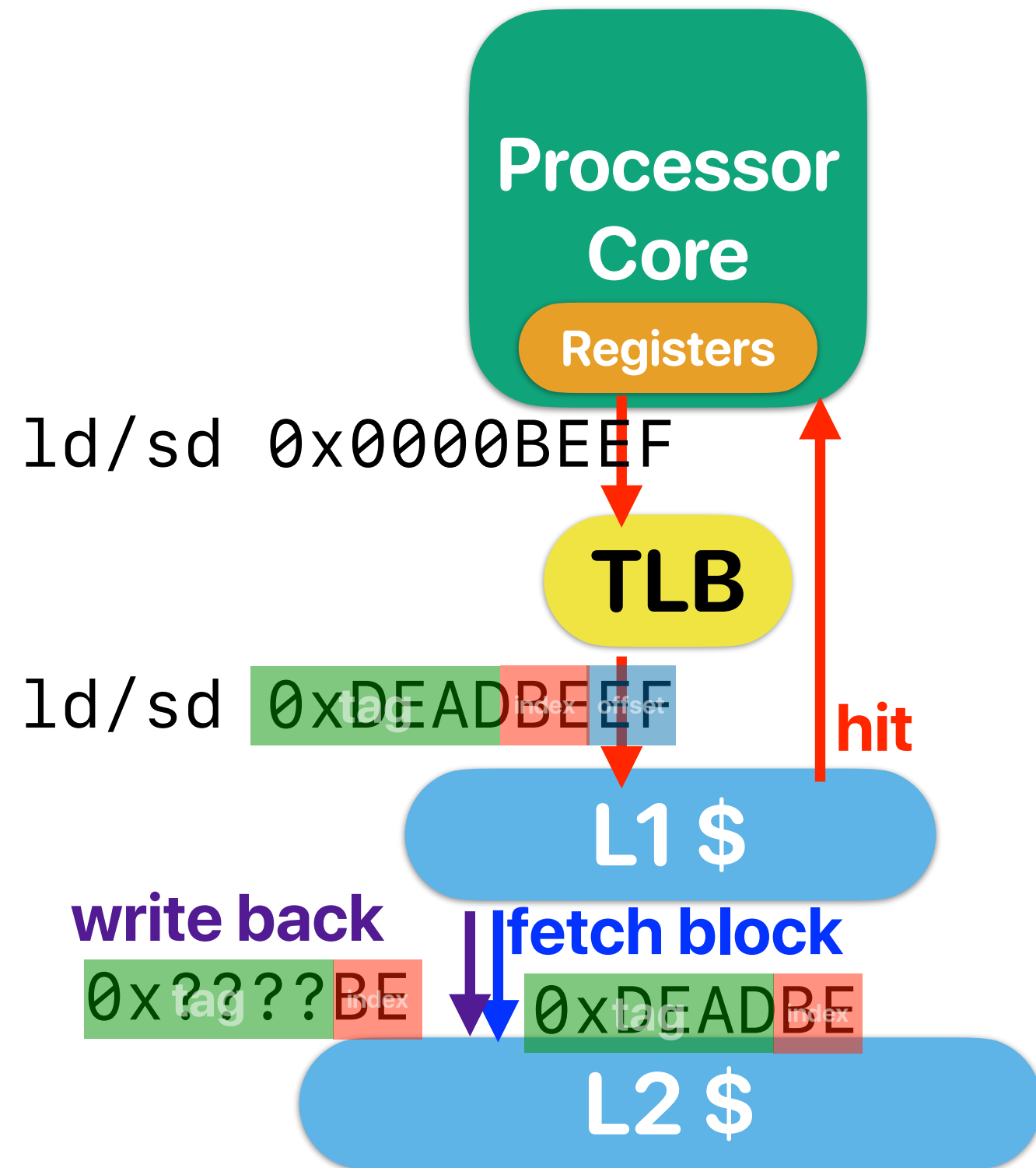


When we have virtual memory...

- 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

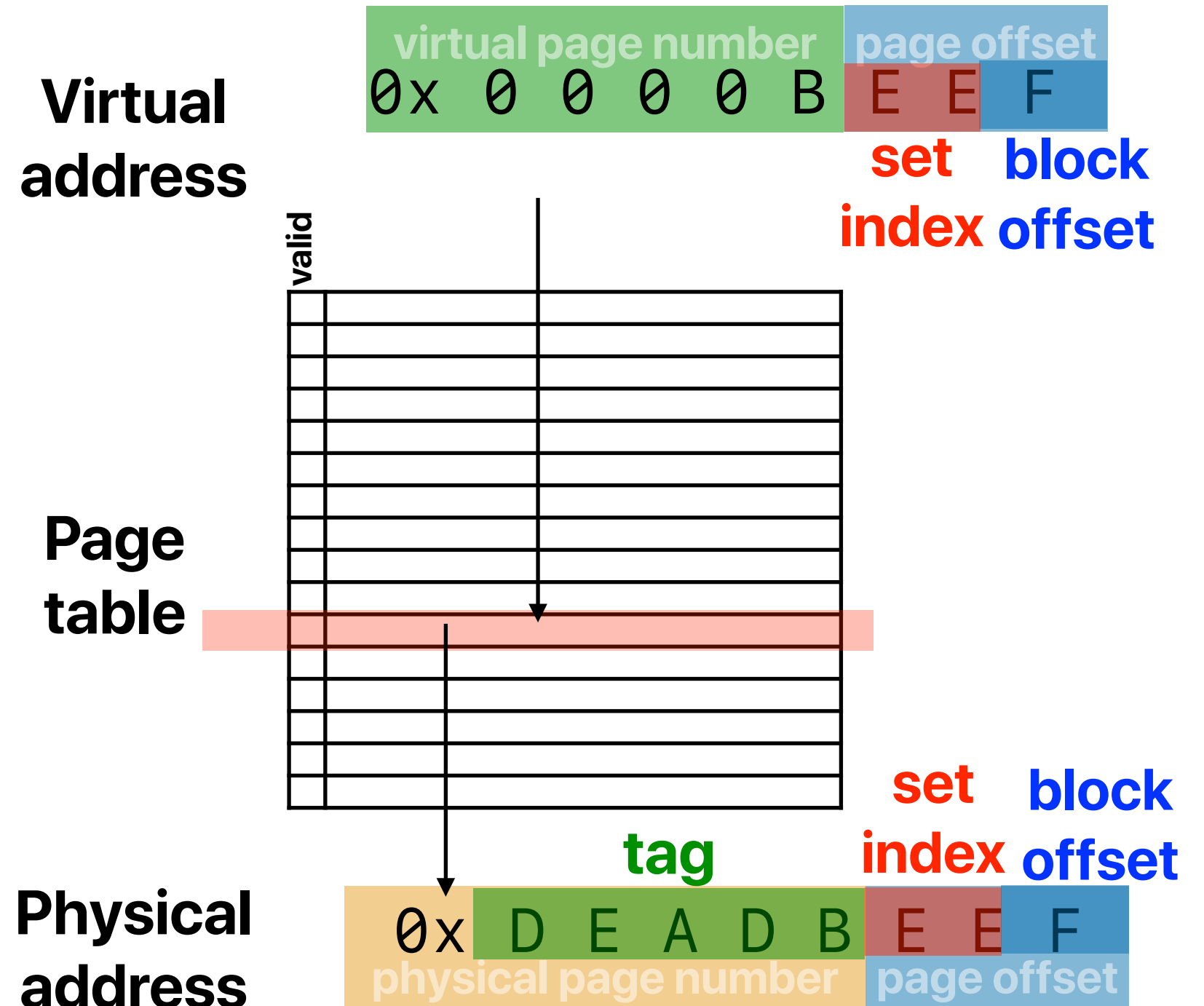
TLB + Virtual cache

- 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**



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

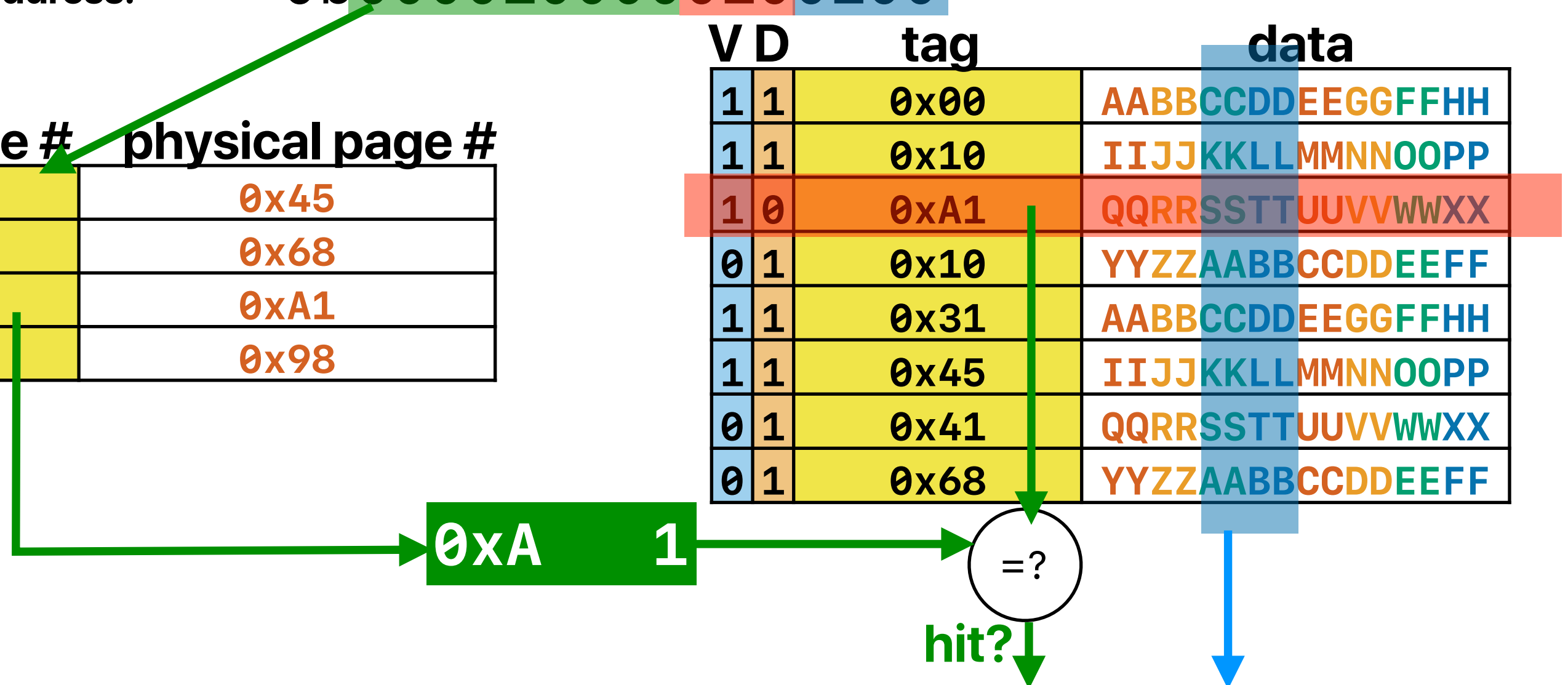
memory address:

0x0	8	2	4
		set	block

memory address: 0b0000100000100100

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

V D		tag	data
1	1	0x00	AABBCCDDEEGGFHH
1	1	0x10	IIJJKKLLMMNNOOPP
1	0	0xA1	QQRRSSTTUUVVWWXX
0	1	0x10	YYZZAABBCCDDEEFF
1	1	0x31	AABBCCDDEEGGFHH
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 $A = 1$

$$C = 4KB$$

