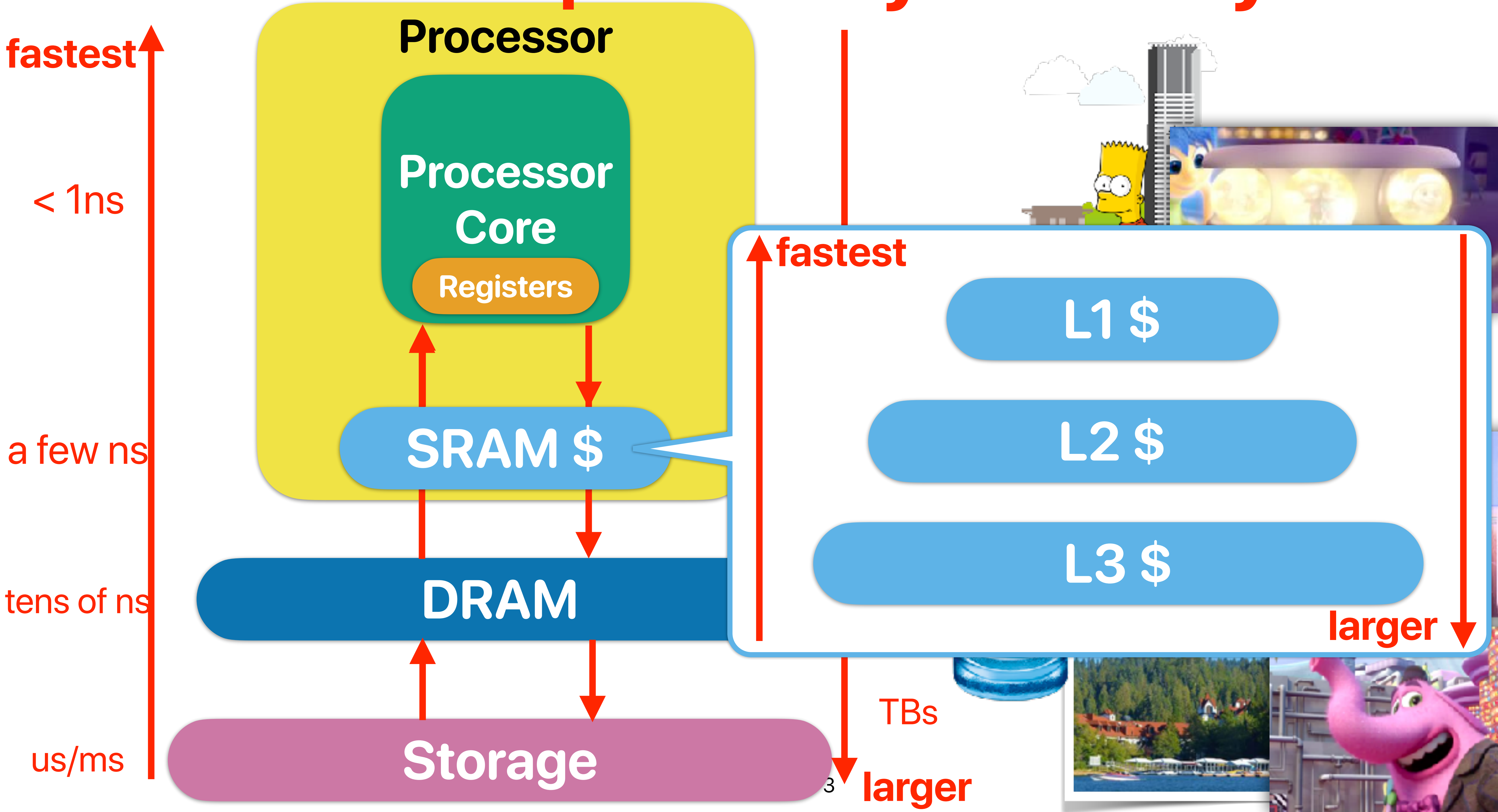
Virtual Memory: Just an Illusion

Hung-Wei Tseng

Recap: Memory Hierarchy



Recap: NVIDIA Tegra X1

100% miss rate!

- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

C = ABS
32KB = 4 * 64 * S
S = 128
offset = lg(64) = 6 bits
index = lg(128) = 7 bits
tag = the rest bits

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b0001000000000000000000	0x8	0x0	Compulsory Miss	
b[0]	0x20000	0b0010000000000000000000	0x10	0x0	Compulsory Miss	
c[0]	0x30000	0b0011000000000000000000	0x18	0x0	Compulsory Miss	
d[0]	0x40000	0b0100000000000000000000	0x20	0x0	Compulsory Miss	
e[0]	0x50000	0b0101000000000000000000	0x28	0x0	Compulsory Miss	a[0-7]
a[1]	0x10008	0b0001000000000000001000	0x8	0x0	**Conflict Miss**	b[0-7]
b[1]	0x20008	0b0010000000000000001000	0x10	0x0	**Conflict Miss**	c[0-7]
c[1]	0x30008	0b0011000000000000001000	0x18	0x0	**Conflict Miss**	d[0-7]
d[1]	0x40008	0b0100000000000000001000	0x20	0x0	**Conflict Miss**	e[0-7]
e[1]	0x50008	0b0101000000000000001000	0x28	0x0	**Conflict Miss**	a[0-7]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Loop optimizations

Loop interchange

A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
  for(i = 0; i < ARRAY_SIZE; i++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

Loop fission

B

```
  double a[8192], b[8192], c[8192], \
         d[8192], e[8192];
  for(i = 0; i < 512; i++) {
      e[i] = (a[i] * b[i] + c[i])/d[i];
  }
```

A

```
  double a[8192], b[8192], c[8192], \
         d[8192], e[8192];
  for(i = 0; i < 512; i++)
      e[i] = a[i] * b[i] + c[i];
  for(i = 0; i < 512; i++)
      e[i] /= d[i];
```

Loop fusion

A

```
  double a[8192], b[8192], c[8192], \
         d[8192], e[8192];
  for(i = 0; i < 512; i++)
      e[i] = a[i] * b[i] + c[i];
  for(i = 0; i < 512; i++)
      e[i] /= d[i];
```

B

```
  double a[8192], b[8192], c[8192], \
         d[8192], e[8192];
  for(i = 0; i < 512; i++) {
      e[i] = (a[i] * b[i] + c[i])/d[i];
  }
```

Recap: Matrix Transpose

```
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
    for(k = 0; k < ARRAY_SIZE; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
        for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
          for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
            for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
              c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```

Reduce capacity/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 < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
        for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
          for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
            for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
              // Compute on b_t
              c[ii][jj] += a[ii][kk]*b_t[jj][kk];
    }
  }
}
```

Reduce conflict misses

Recap: Summary of Optimizations

- Software
  - Data layout — capacity miss, conflict miss, compulsory miss
  - Blocking — capacity miss, conflict miss
  - Loop fission — conflict miss — when $ has limited way associativity
  - Loop fusion — capacity miss — when $ has enough way associativity
  - Loop interchange — conflict/capacity miss
- Hardware
  - Prefetch — compulsory miss
  - Write buffer — miss penalty
  - Bank/pipeline — miss penalty
  - Critical word first and early restart — miss panelty

What will happen?

- If we execute the code on the right-hand side code on a machine with only 32 GB of physical memory installed and the dim is "70000" (requires 70000*70000*8 bytes ~ 37 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;
}
```

Outline

- Virtual memory
- Architectural support for virtual memory
- Advanced hardware support for virtual memory

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;
}
```

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

Consider the following code …

- Consider the case when we run multiple instances of the given program at the same time on modern machines, which pair of statements is 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
  ④ Some instances will print the same value of a
  ⑤ Each instance will print a different value of a

  A. (1) & (3)
  B. (1) & (4)
  C. (1) & (5)
  D. (2) & (3)
  E. (2) & (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>

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
B
C
D
E

Consider the following code …

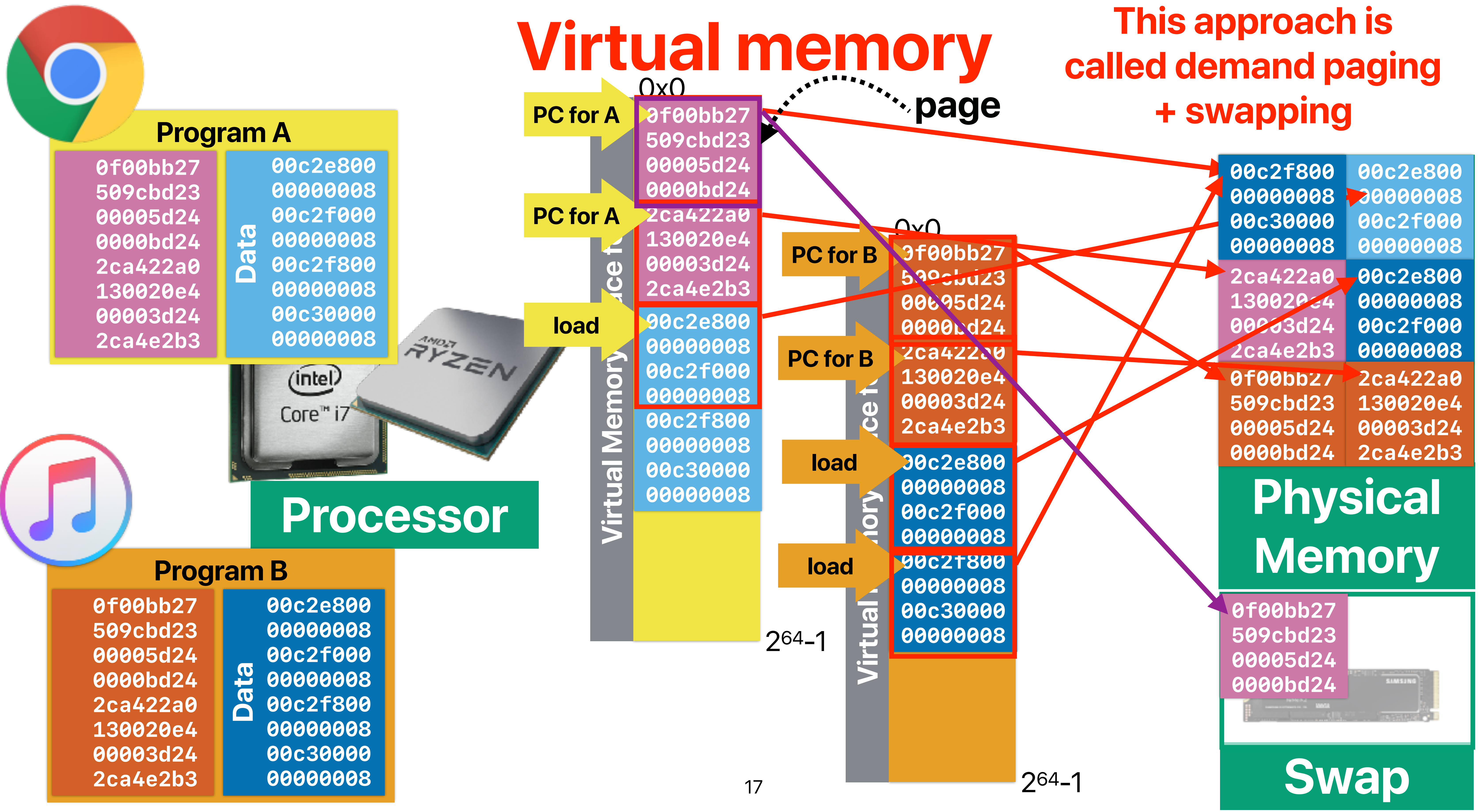
- Consider the case when we run multiple instances of the given program at the same time on modern machines, which pair of statements is correct?
  - ① The printed "address of a" is the same for every running instances
  - ② The printed "address of a" is different for each instance
  - ③ All running instances will print the same value of a
  - ④ Some instances will print the same value of a
  - ⑤ Each instance will print a different value of a
  - A. (1) & (3)
  - B. (1) & (4)
  - [✓ C. (1) & (5)]
  - D. (2) & (3)
  - E. (2) & (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>

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;
}
```

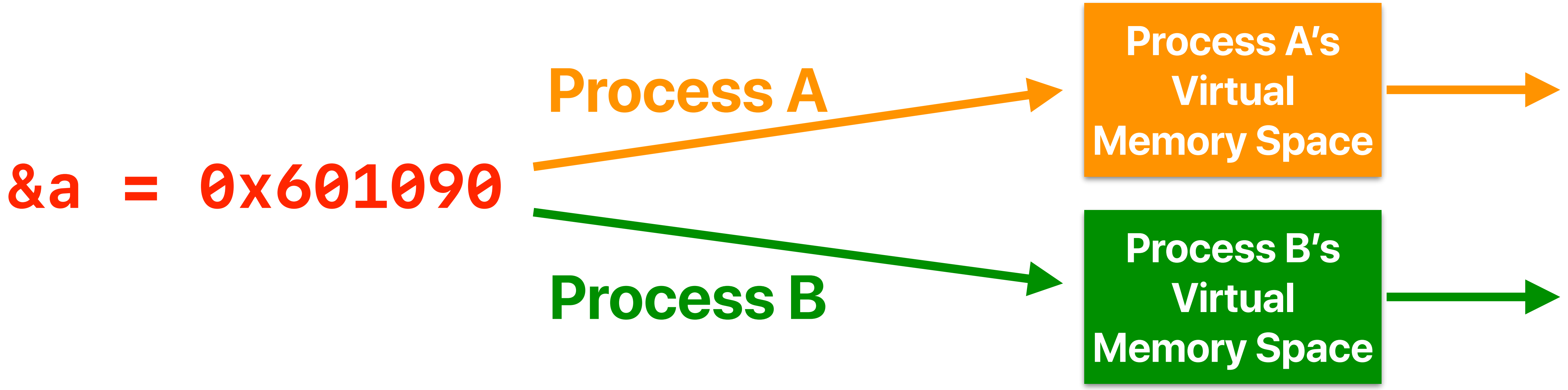
Virtual Memory

Virtual memory

This approach is called demand paging + swapping



Demo revisited


&a = 0x601090
Process A
Process B
Process A's Virtual Memory Space
Process B's Virtual Memory Space

```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]);
    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;
}
```

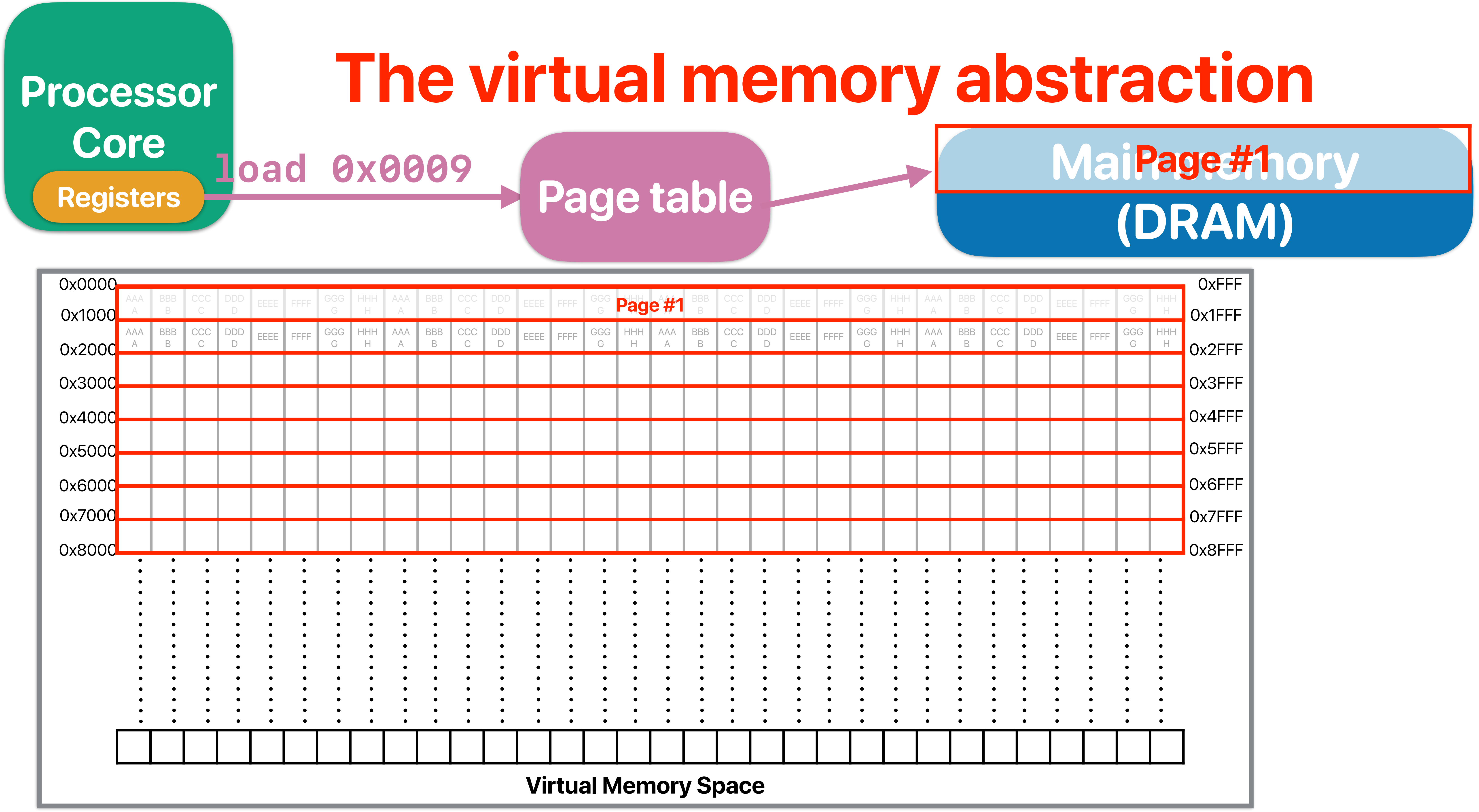
Virtual memory

- 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**"

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

The virtual memory abstraction

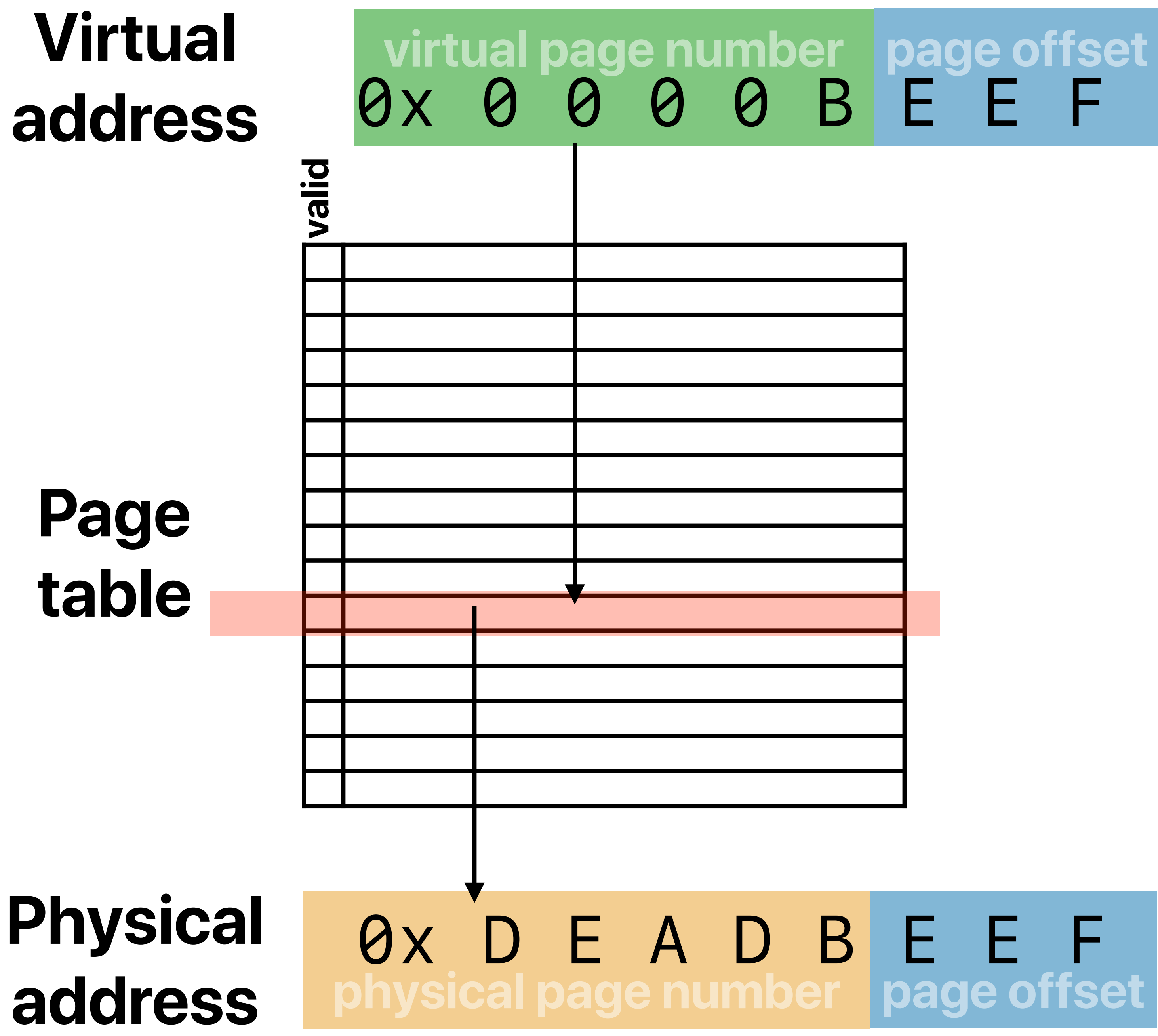


Demand paging

- 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

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

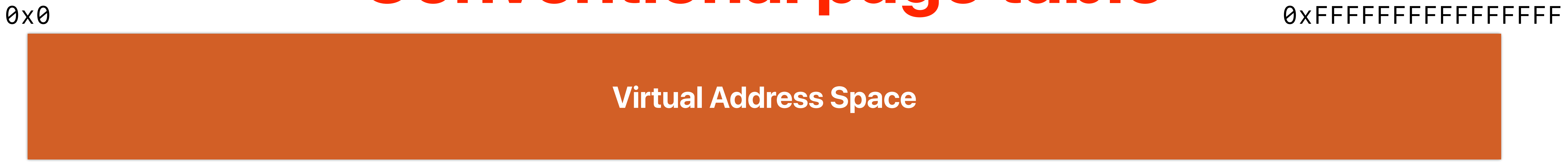
Size of page table

- 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

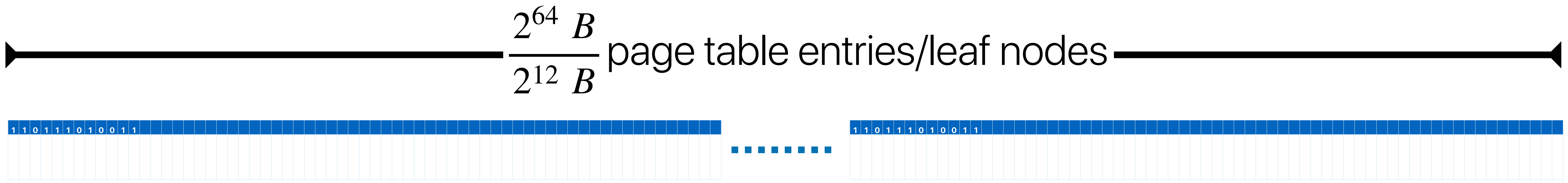
Conventional page table



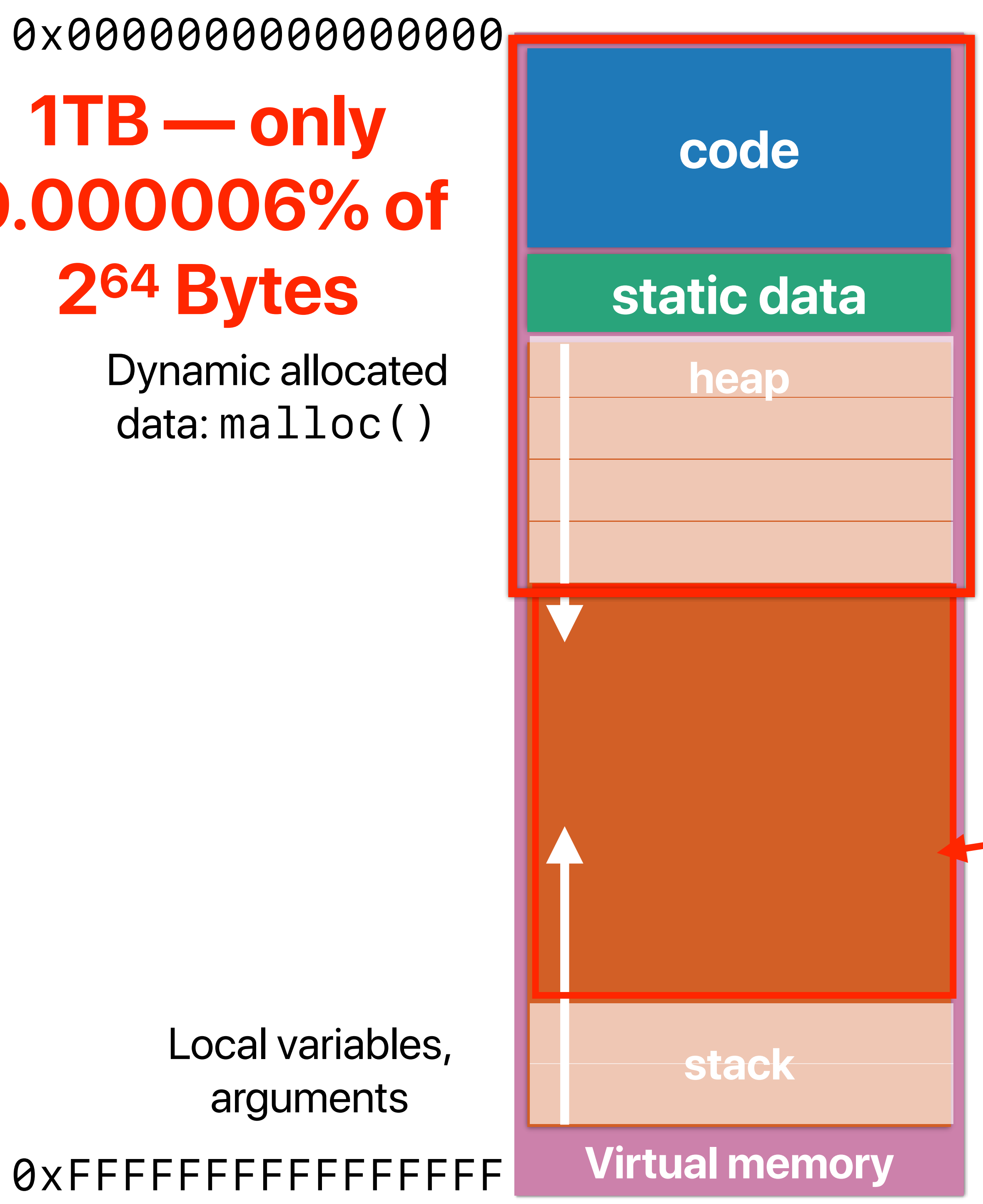
— must be consecutive in the physical memory

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

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



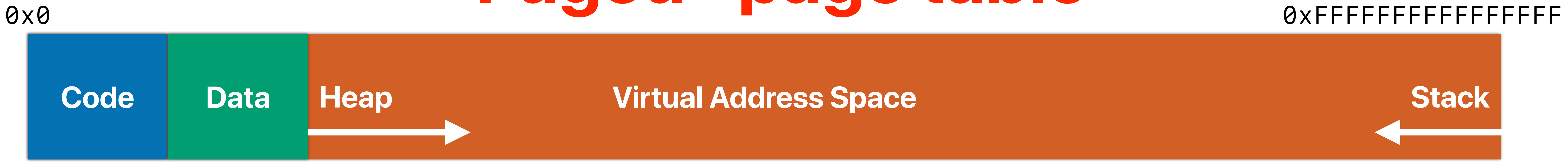
Do we really need a large table?


0x0000000000000000
1TB — only 0.000006% of 2^{64} Bytes
Dynamic allocated data: malloc()
code
static data
heap
stack
Local variables, arguments
0xFFFFFFFFFFFFFFFF
Virtual memory

Your program probably never uses this huge area!

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

“Paged” page table

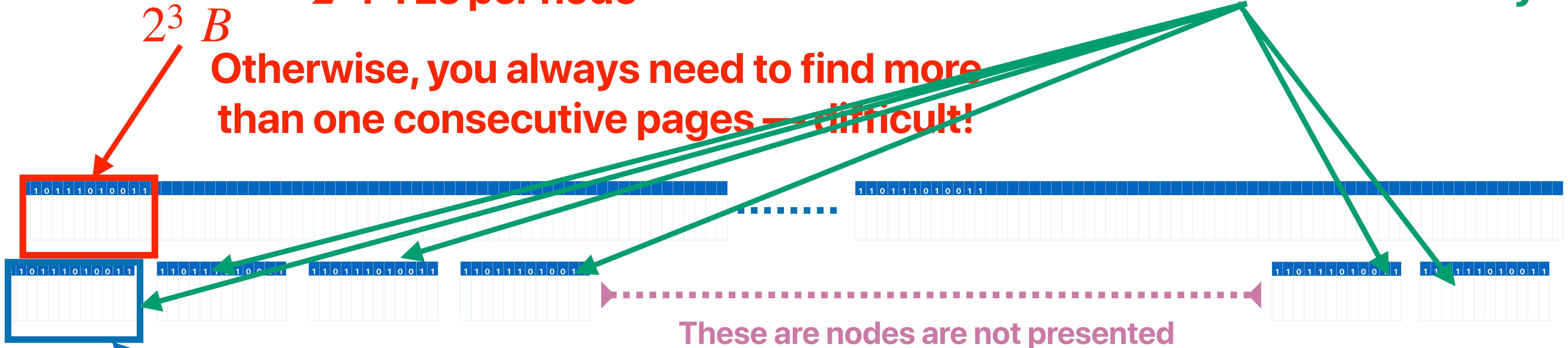


Break up entries into pages!
Each of these occupies exactly a page

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

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

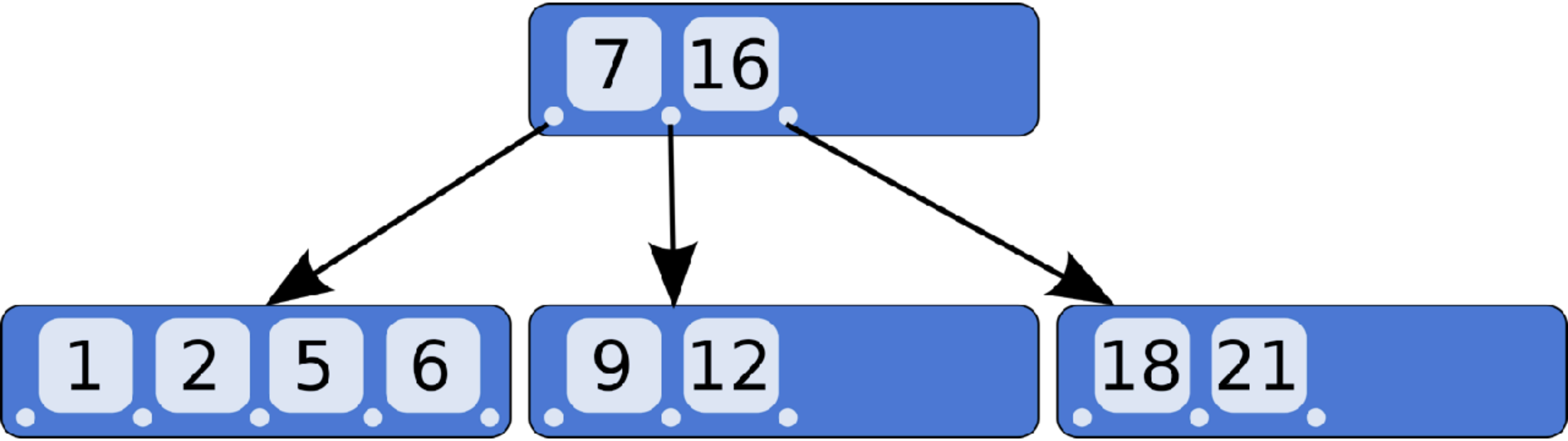
Question:
These nodes are spread out, how to locate them in the memory?



These are nodes are not presented if they are not referenced at all — save space

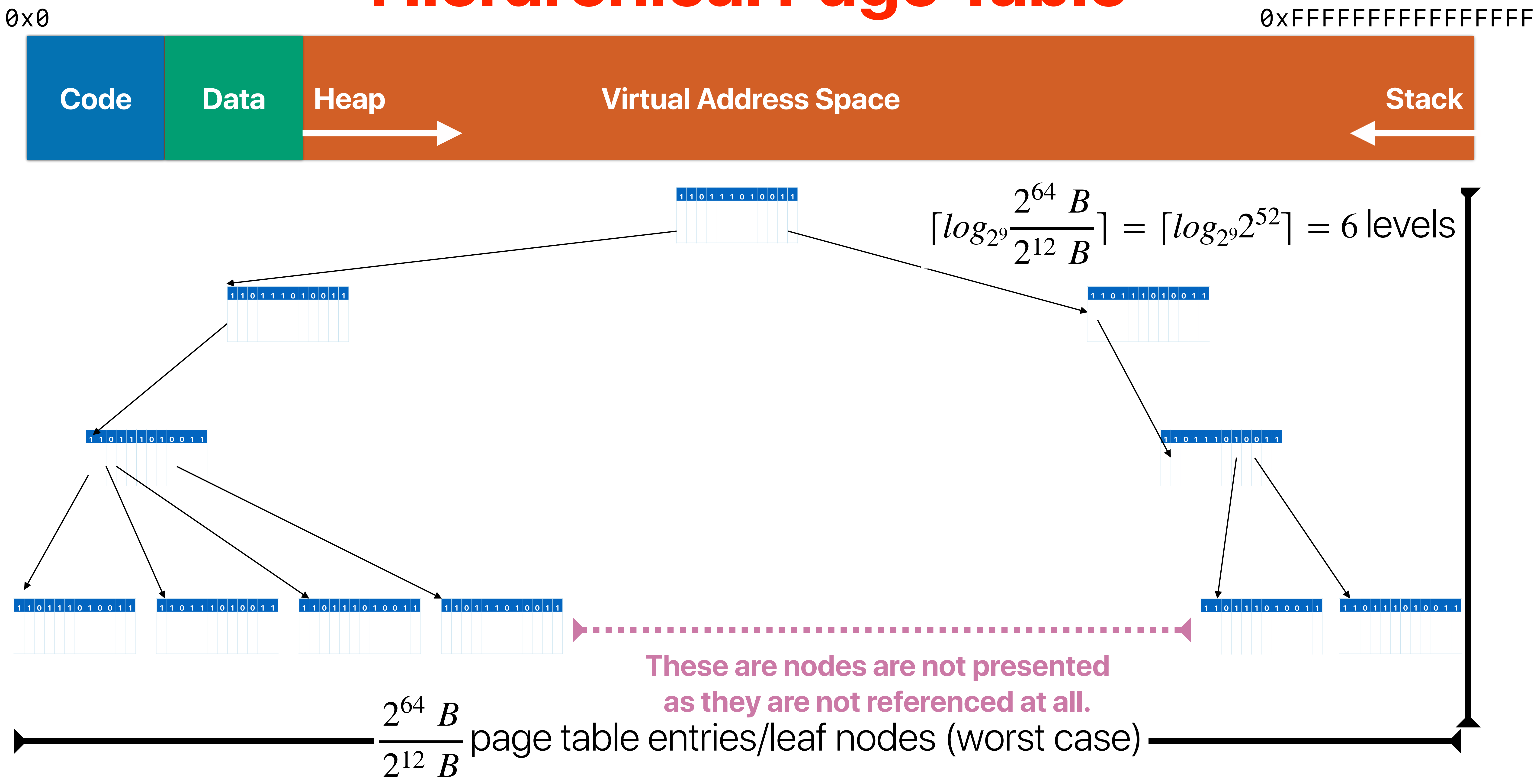
Allocate page table entry nodes “on demand”

B-tree



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

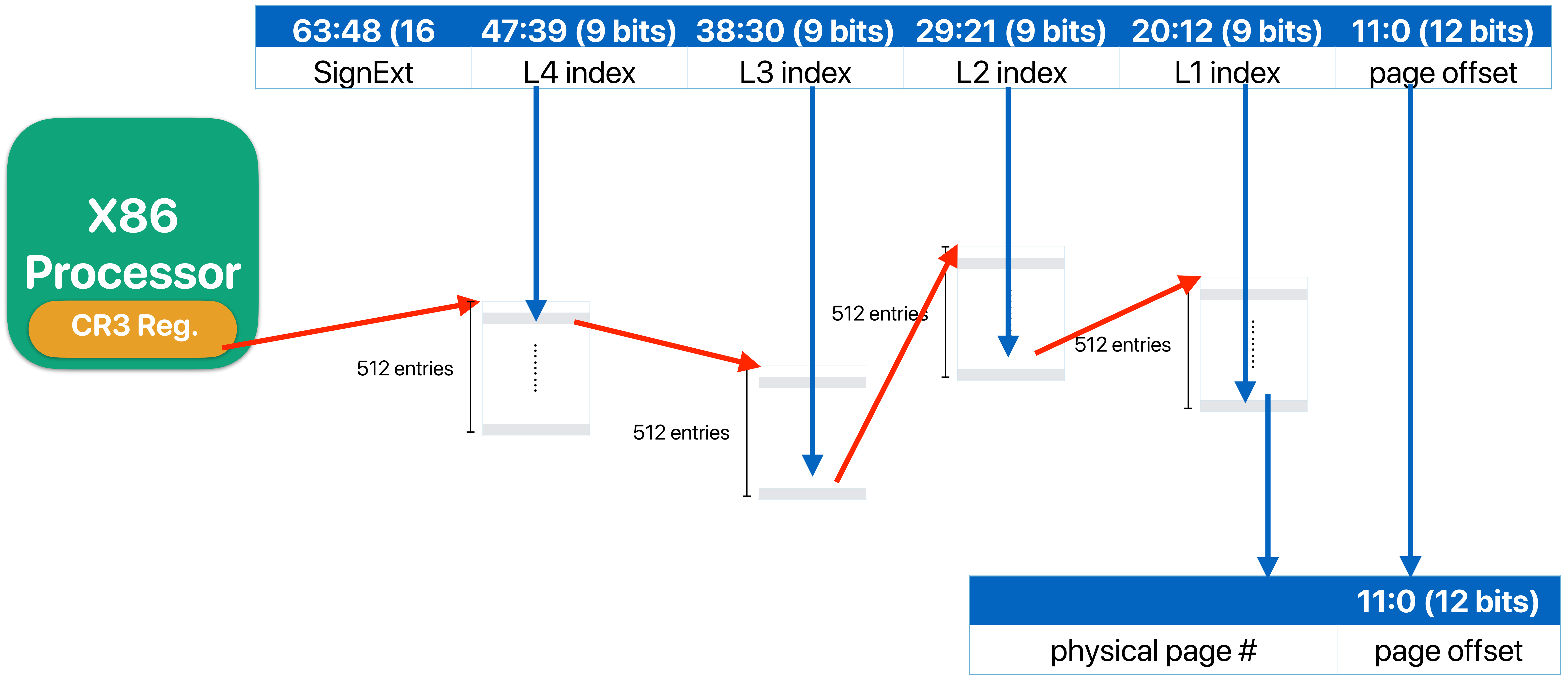
Hierarchical Page Table


0x0
0xFFFFFFFFFFFFFFFF
Code
Data
Heap
Virtual Address Space
Stack
These are nodes are not presented as they are not referenced at all.

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

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

Address translation in x86-64

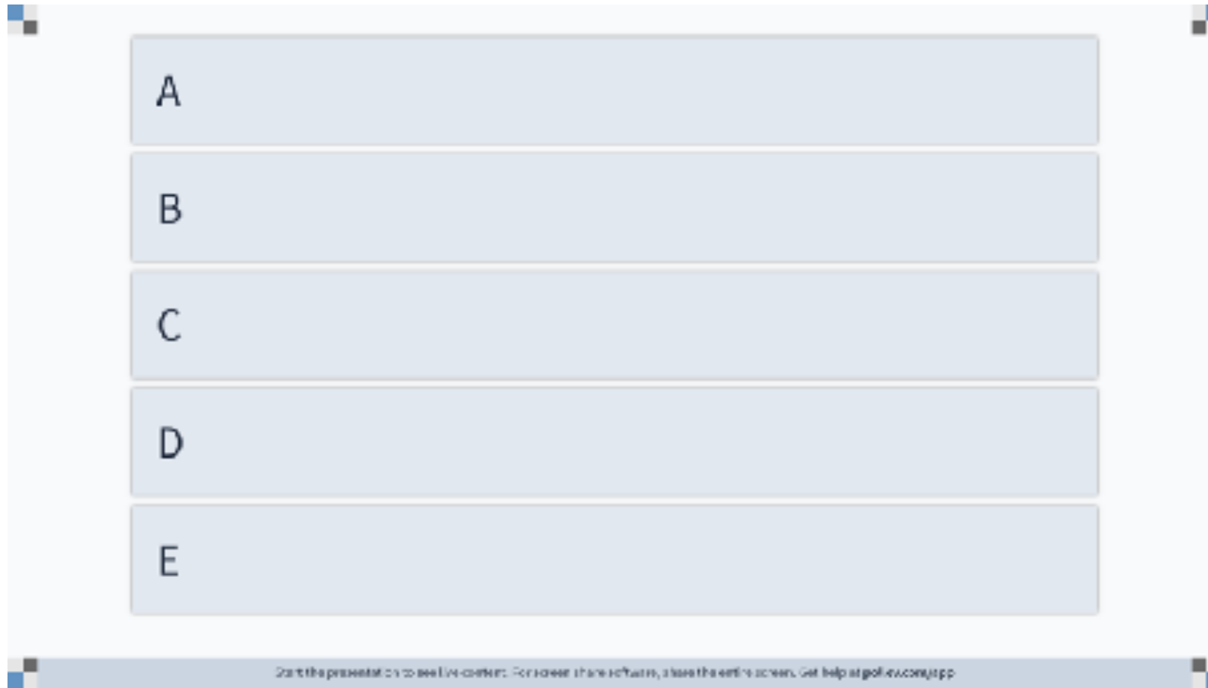


Translation Caching: Skip, Don't Walk (the Page Table)
Thomas W. Barr, Alan L. Cox, Scott Rixner

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


A

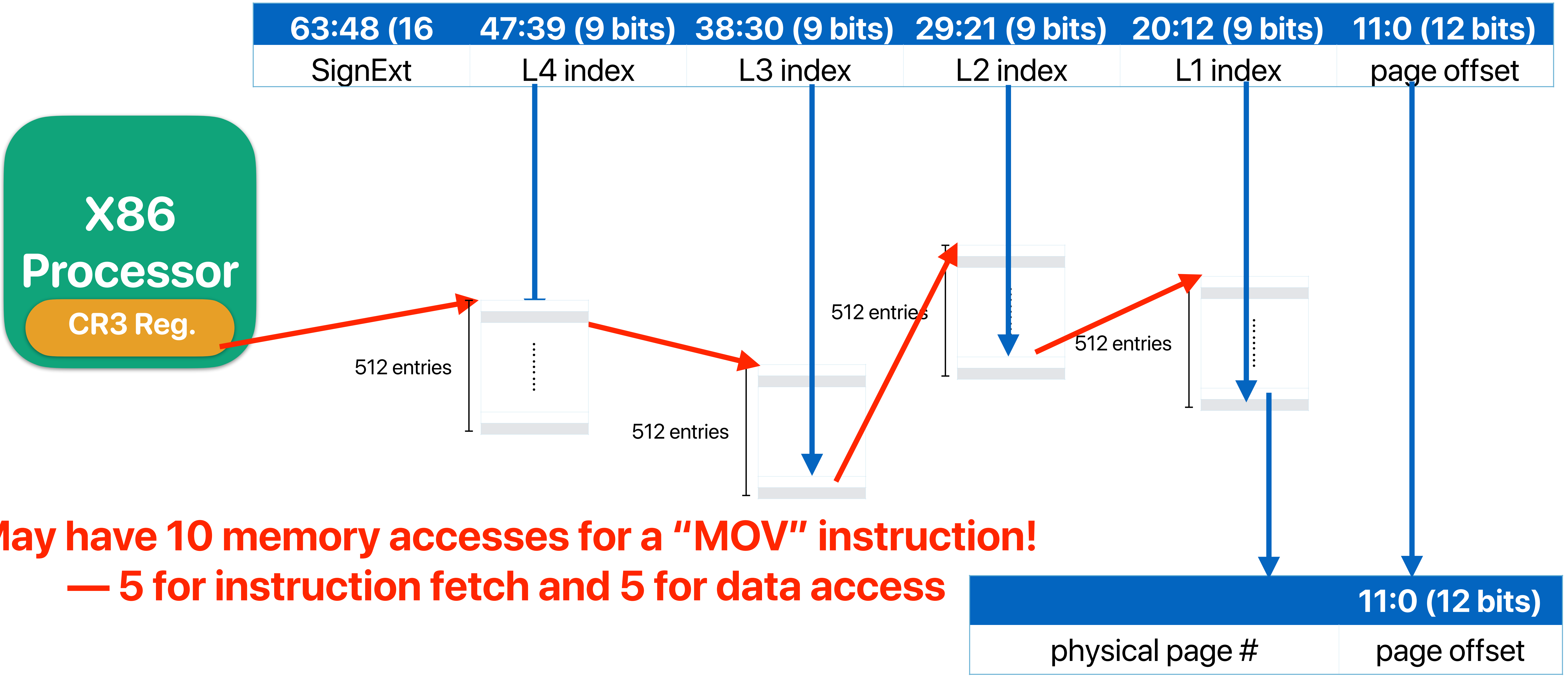
B

C

D

E

Address translation in x86-64



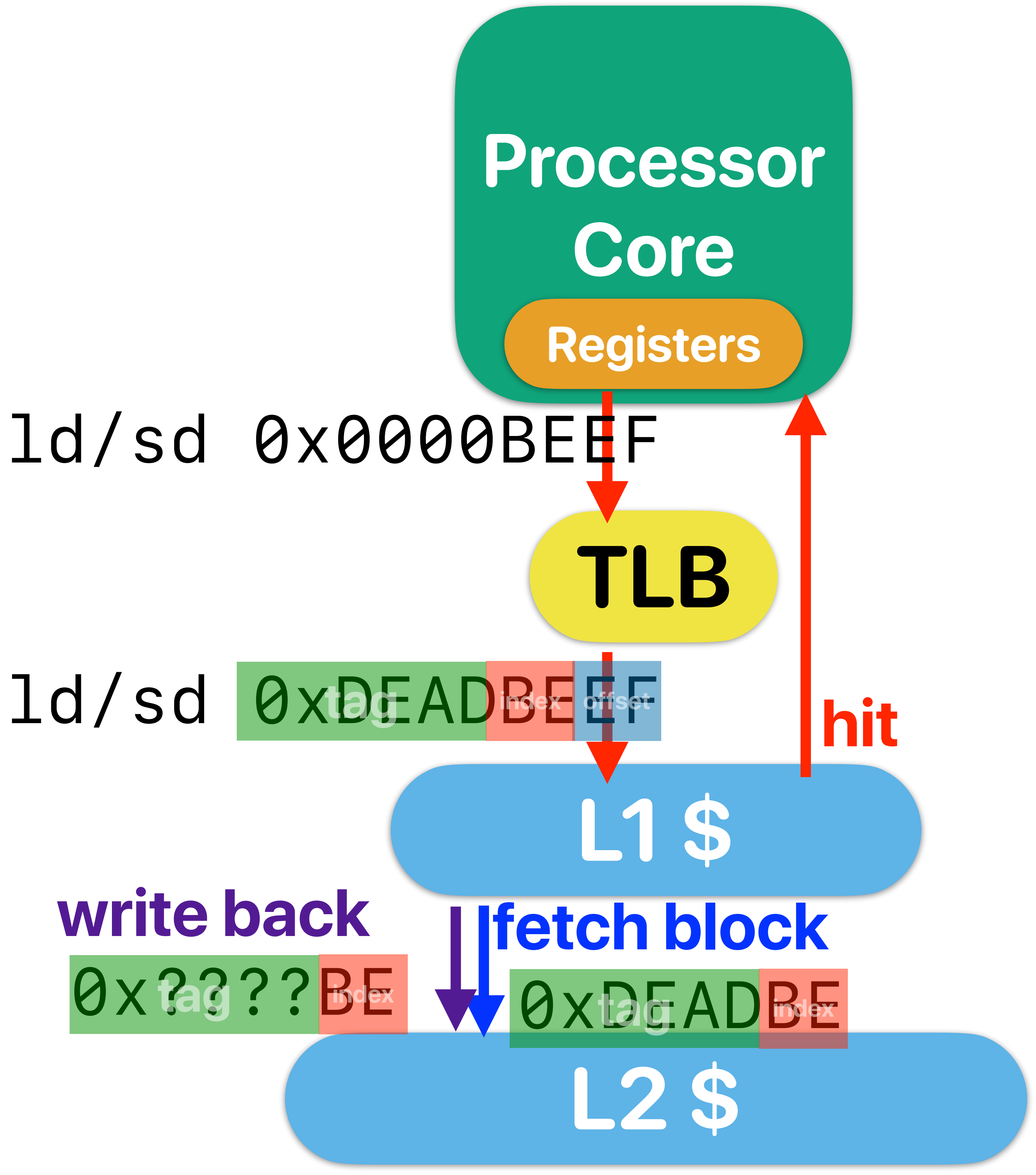
May have 10 memory accesses for a "MOV" instruction!
— 5 for instruction fetch and 5 for data access

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

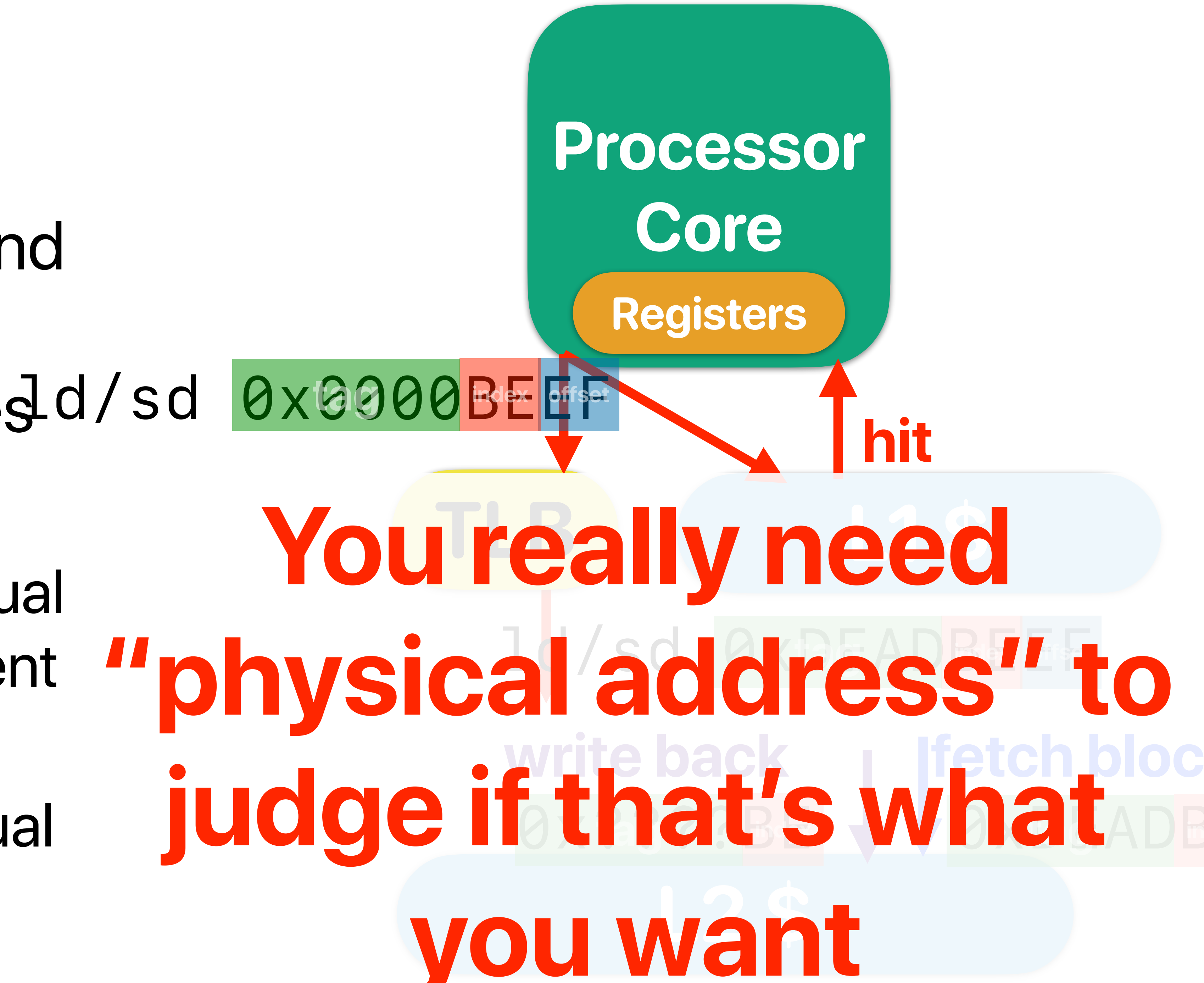
TLB: Translation Look-aside Buffer



- 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

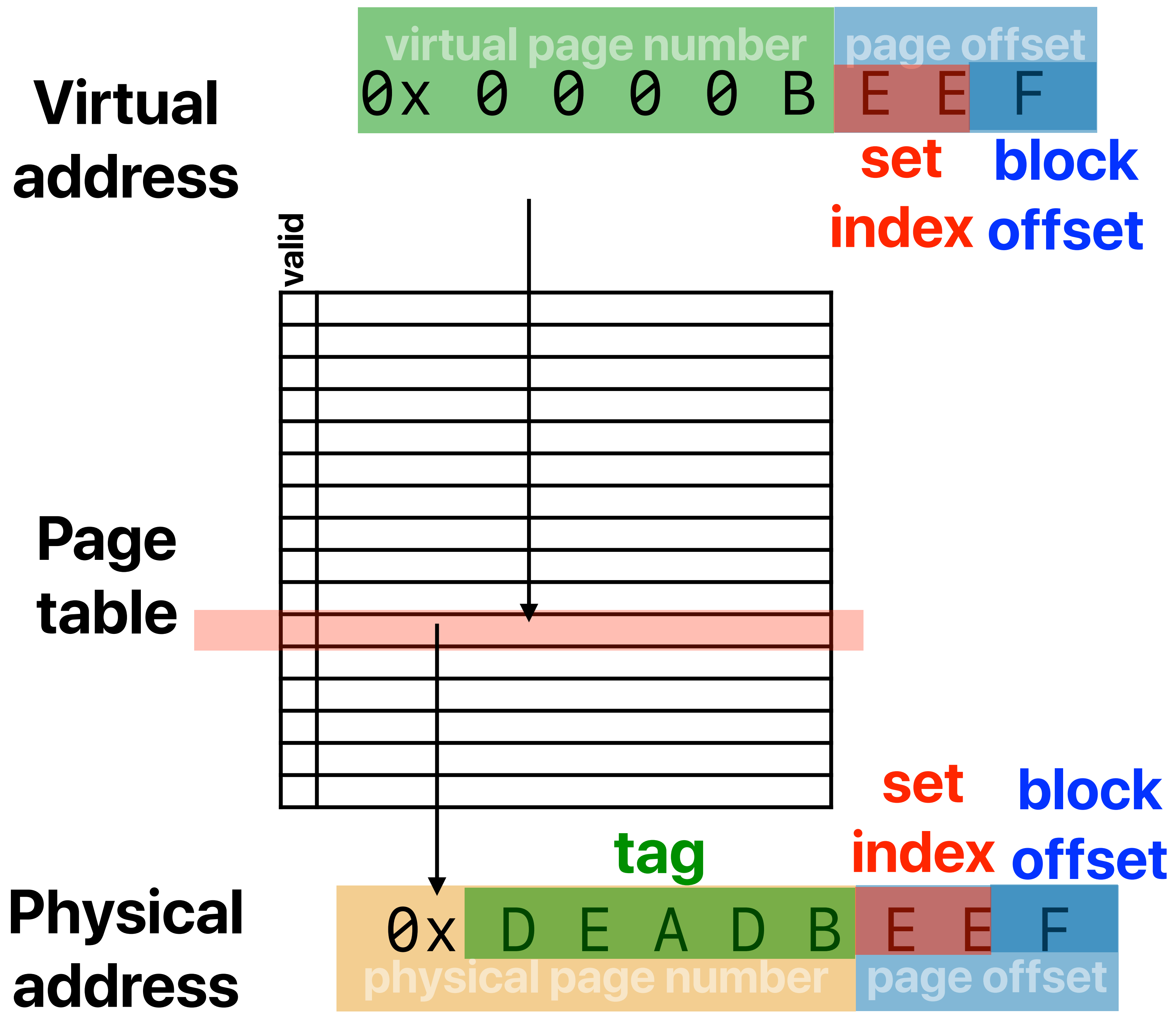
- L1 $ accepts virtual address — you don't need to translate
- Good — you can access both TLB and L1-$ at the same time and physical address is only needed if L1-$ misses
- Bad — it doesn't work in practice
  - Many applications have the same virtual address but should be pointing different **physical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



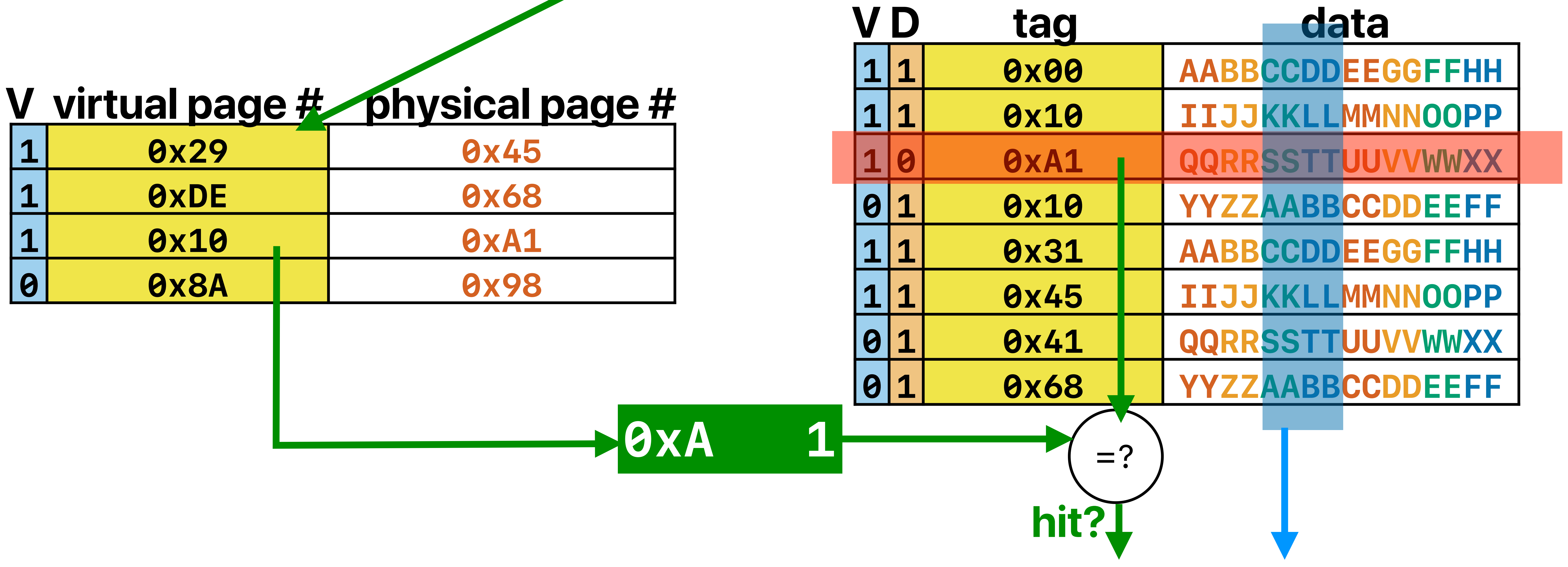
Virtually indexed, physically tagged cache

memory address: 0x0 8 2 4

set index | block offset

virtual page # | index | offset

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	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 —

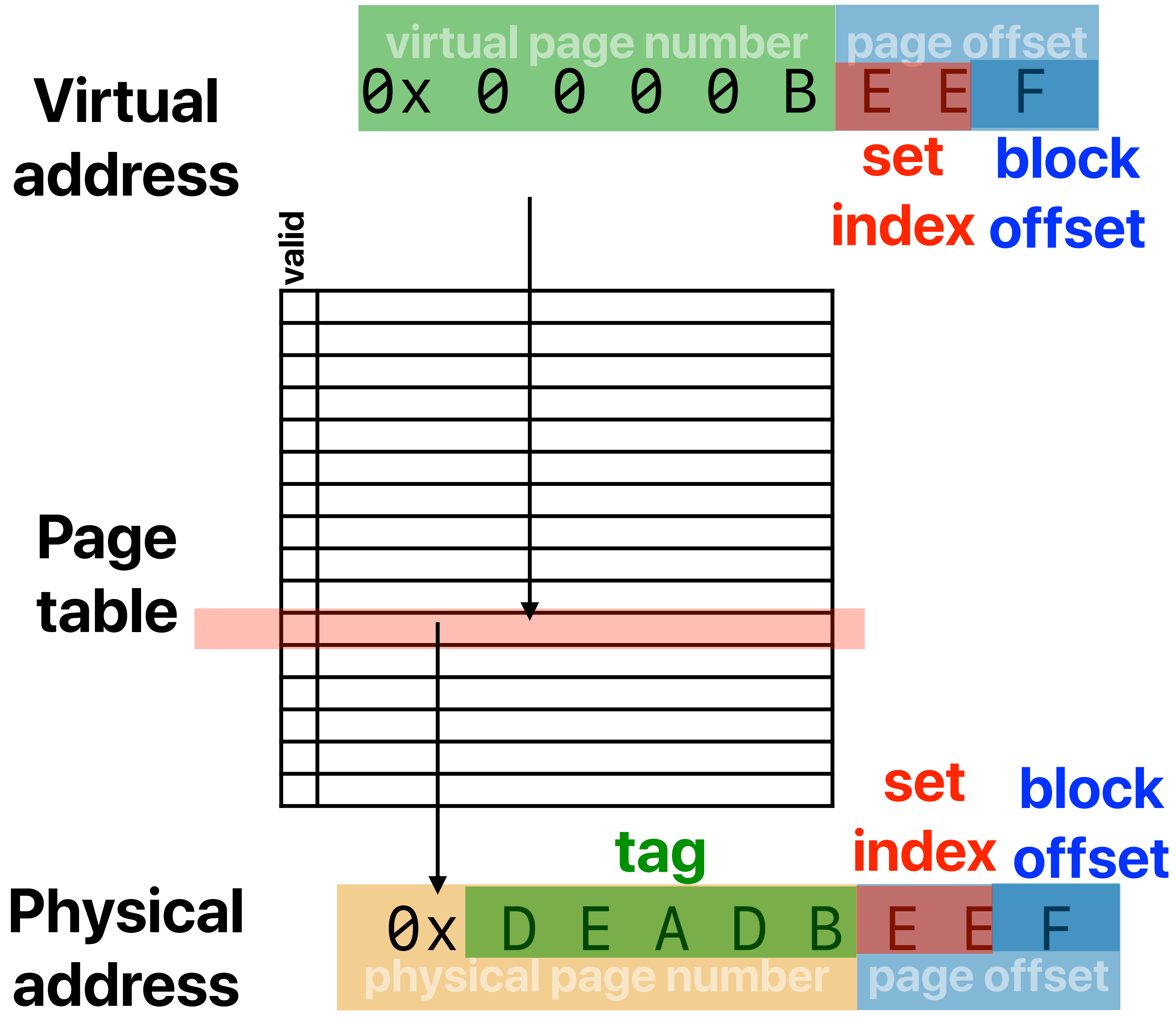
$$lg(B) + lg(S) = lg(4096) = 12$$

$$C = ABS$$

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

$$if\ A = 1$$

$$C = 4KB$$



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

A

B

C

D

E

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 configures its own cache

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

$$C = ABS$$

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

$$A = 8$$

About midterm

Format of the midterm

- Multiple choices * 16 — like your poll/reading quizzes multiple choices questions
- Short answer questions * 3
- Homework style correctness questions * 3
  - You need to clearly write down the original form of the applied equation/formula
  - You need to replace each term accordingly with numbers
  - You will have some credits for right equations even though the final number isn’t correct
  - You will receive 0 credits if we only see the numbers

Sample Midterm

Programmer's impact

- By adding the "sort" in the following code snippet, what the programmer changes in the performance equation to achieve **better** performance?

```
std::sort(data, data + arraySize);

for (unsigned c = 0; c < arraySize*1000; ++c) {
        if (data[c%arraySize] >= INT_MAX/2)
            sum ++;
    }
}
```

A. CPI
B. IC
C. CT
D. IC & CPI
E. CPI & CT

Amdahl's Law on Multicore Architectures

- Regarding Amdahl's Law on multicore architectures, how many of the following statements is/are correct?
  ① If we have unlimited parallelism, the performance of each parallel piece does not matter as long as the performance slowdown in each piece is bounded
  ② With unlimited amount of parallel hardware units, single-core performance does not matter anymore
  ③ With unlimited amount of parallel hardware units, the maximum speedup will be bounded by the fraction of parallel parts
  ④ With unlimited amount of parallel hardware units, the effect of scheduling and data exchange overhead is minor

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

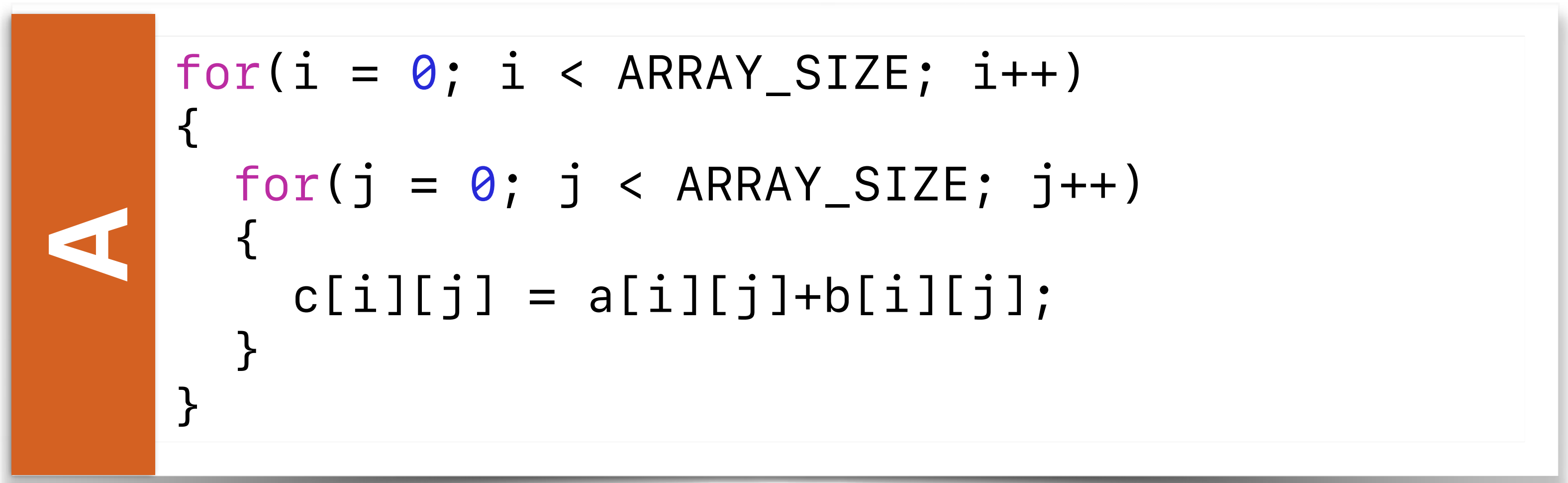
How programmer affects performance?

- Performance equation consists of the following three factors
  ① IC
  ② CPI
  ③ CT

How many can a **programmer** affect?

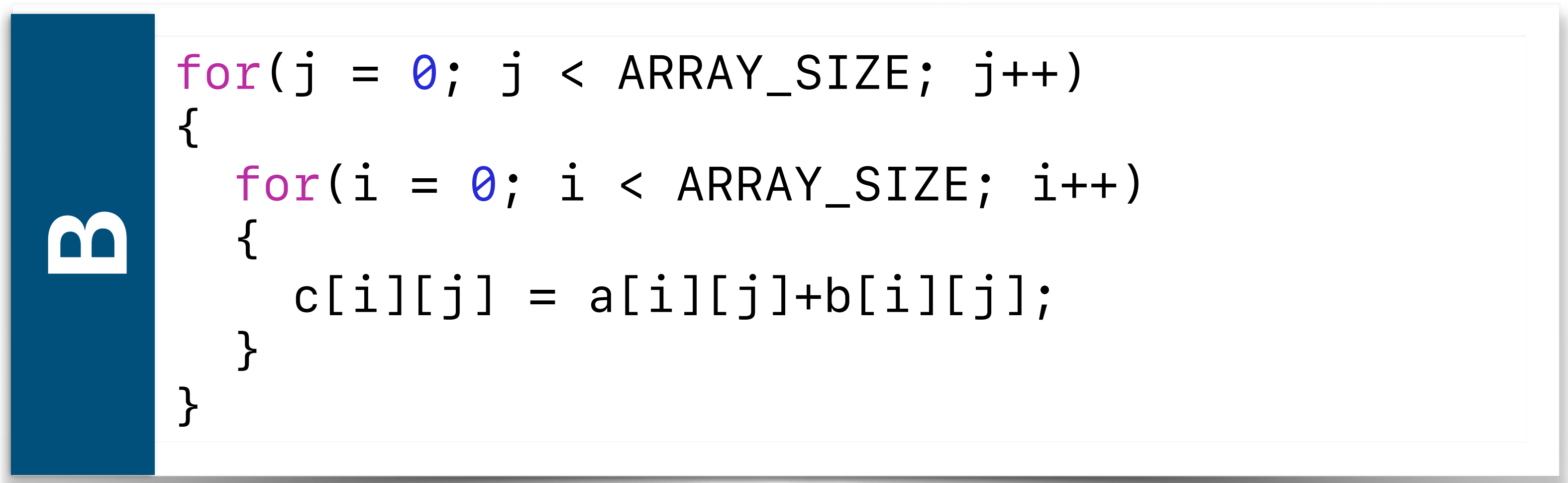
A. 0
B. 1
C. 2
D. 3

Demo — programmer & performance



A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```



B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
  for(i = 0; i < ARRAY_SIZE; i++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

- How many of the following make(s) the performance of A better than B?
  ① IC
  ② CPI
  ③ CT
  A. 0
  B. 1
  C. 2
  D. 3

Data locality

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint32_t i = 0; i < m; i++) {
     result = 0;
    for(uint32_t j = 0; j < n; j++) {
          result += matrix[i][j]*vector[j];
    }
    output[i] = result;
 }
```

A. Access of `matrix` has temporal locality, `vector` has spatial locality
B. Both `matrix` and `vector` have temporal locality, and `vector` also has spatial locality
C. Access of `matrix` has spatial locality, `vector` has temporal locality
D. Both `matrix` and `vector` have spatial locality and temporal locality
E. Both `matrix` and `vector` have spatial locality, and `vector` also has temporal locality

3Cs and A, B, C

- Regarding 3Cs: compulsory, conflict and capacity misses and A, B, C: associativity, block size, capacity
  How many of the following are correct?
  ① Increasing associativity can reduce conflict misses
  ② Increasing associativity can reduce hit time
  ③ Increasing block size can increase the miss penalty
  ④ Increasing block size can reduce compulsory misses
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

intel Core i7

- L1 data (D-L1) cache configuration of Core i7
  - Size 32KB, 8-way set associativity, 64B block
  - Assume 64-bit memory address
  - Which of the following is NOT correct?

A. Tag is 52 bits

B. Index is 6 bits

C. Offset is 6 bits

D. The cache has 128 sets

Virtual indexed, physical tagged cache limits the cache size

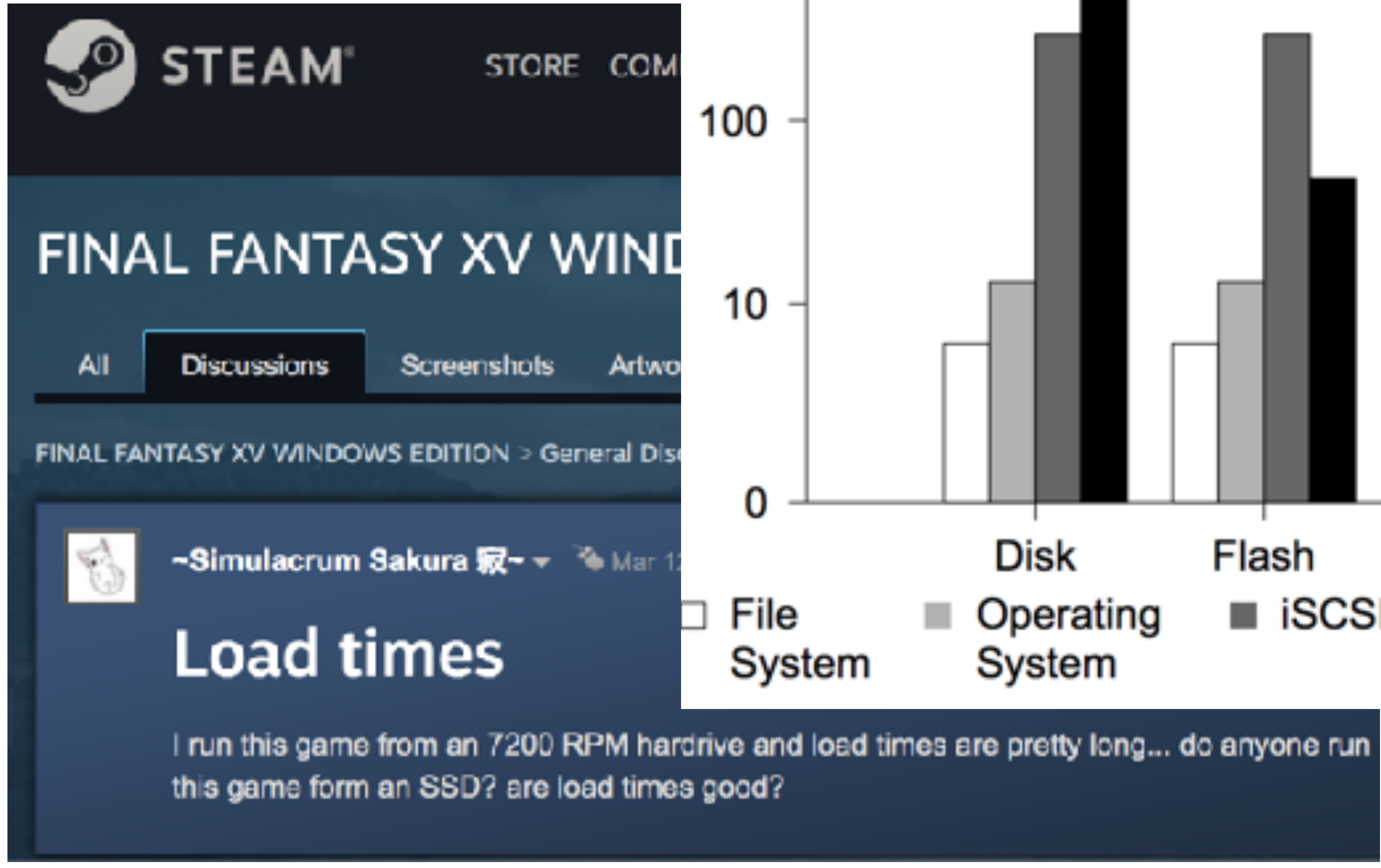
- If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the system use 4K pages.
  A. 32B blocks, 2-way
  B. 32B blocks, 4-way
  C. 64B blocks, 4-way
  D. 64B blocks, 8-way

When we have virtual memory...

- In a modern x86-64 processor supports virtual memory through, how many memory accesses can an instruction incur?
    A. 2
    B. 4
    C. 6
    D. 8
    E. More than 10

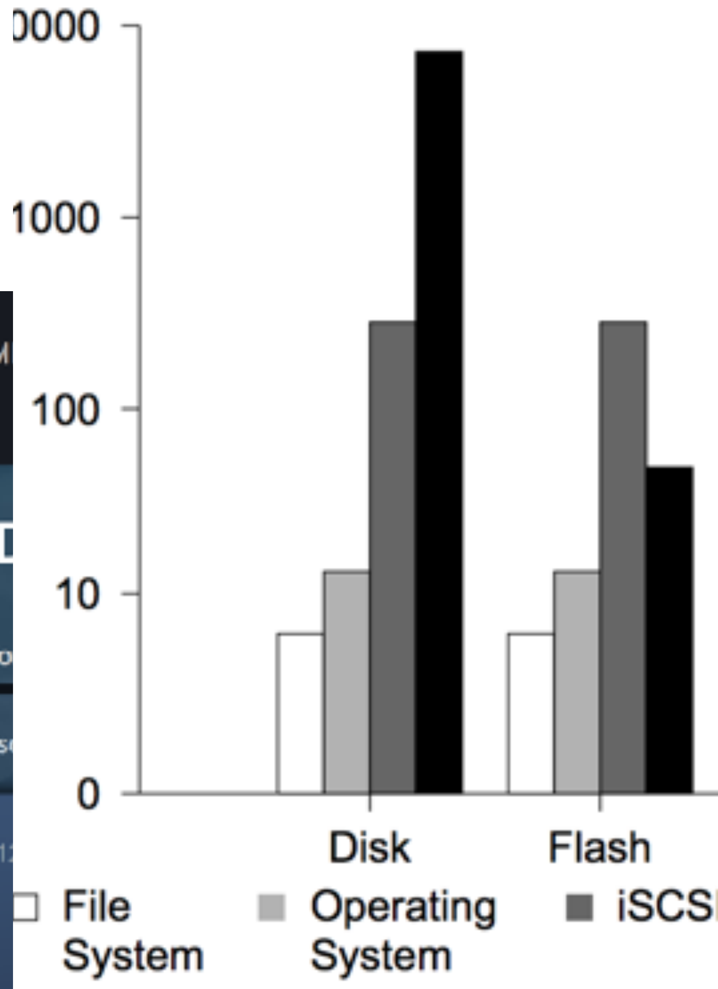
Practicing Amdahl's Law (2)

- Final Fantasy XV spends lots of time loading a map — within which period that 95% of the time on the accessing the H.D.D., the rest in the operating system, file system and the I/O protocol. If we replace the H.D.D. with a flash drive, which provides 100x faster access time and a better processor to accelerate the software overhead by 2x. By how much can we speed up the map loading process?
  A. ~7x
  B. ~10x
  C. ~17x
  D. ~29x
  E. ~100x



Load times

I run this game from an 7200 RPM hardrive and load times are pretty long... do anyone run this game form an SSD? are load times good?



What data structure is performing better

	Array of objects	object of arrays

	`struct grades`
	`{`
	`  int id;`
	`  double *homework;`
	`  double average;`
	`};`
		`struct grades`
		`{`
		`  int *id;`
		`  double **homework;`
		`  double *average;`
		`};`

average of each homework	`for(i=0;i<homework_items; i++)`
	`{`
	`gradesheet[total_number_students].homework[i] = 0.0;`
	`   for(j=0;j<total_number_students;j++)`
	`gradesheet[total_number_students].homework[i]`
	`+=gradesheet[j].homework[i];`
	`   gradesheet[total_number_students].homework[i] /=`
	`(double)total_number_students;`
	`}`
		`for(i = 0;i < homework_items; i++)`
		`{`
		`  gradesheet.homework[i][total_number_students] = 0.0;`
		`  for(j = 0; j <total_number_students;j++)`
		`  {`
		`      gradesheet.homework[i][total_number_students] +=`
		`gradesheet.homework[i][j];`
		`  }`
		`      gradesheet.homework[i][total_number_students] /=`
		`total_number_students;`
		`}`

- Considering your workload would like to calculate the average score of **one of the homework** for **all students**, which data structure would deliver better performance?
    A. Array of objects
    B. Object of arrays

Which of the following schemes can help Tegra?

- How many of the following schemes mentioned in "improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers" would help NVIDIA's Tegra for the code in the previous slide?
  ① Missing cache
  ② Victim cache
  ③ Prefetch
  ④ Stream buffer
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

The result of `sizeof(struct struct_8)`

- Consider the following data structure:

```
struct struct_8 {
    uint8_t a;
    uint64_t b;
    uint8_t c;
} ;
```

What’s the output of
`printf(“%lu\n”,sizeof(struct struct_8))?`

A. 10
B. 17
C. 24
D. 36
E. 80

What if the code look like this?

- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i]; //load a, b, c and then store to e
for(i = 0; i < 512; i++)
    e[i] /= d[i]; //load e, load d, and then store to e
```

What's the data cache miss rate for this code?

A. ~10%

B. ~20%

C. ~40%

D. ~80%

E. 100%

Pipelined access and multi-banked caches

- Assume each bank in the $ takes 10 ns to serve a request, and the $ can take the next request 1 ns after assigning a request to a bank — if we have 4 banks and we want to serve 4 requests, what's the speedup over non-banked, non-pipelined $? — pick the closest one
  A. 1x — no speedup
  B. 2x
  C. 3x
  D. 4x
  E. 5x

Summary of Optimizations

- Regarding the following cache optimizations, how many of them would help improve miss rate?
  ① Non-blocking/pipelined/multibanked cache
  ② Critical word first and early restart
  ③ Prefetching
  ④ Write buffer
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Sample short answer questions (< 40 words)

- What are the limitations of compiler optimizations? Can you list two?
- Please define Amdahl's Law and explain each term in it
- Please define the CPU performance equation and explain each term.
- Can you list two things affecting each term in the performance equation?
- What's the difference between latency and throughput? When should you use latency or throughput to judge performance? When will throughput mislead performance?
- What's "benchmark" suite? Why is it important?
- Why TFLOPS or inferences per second is not a good metrics?

Amdahl's Law for multiple optimizations

- Assume that a program is mainly composed of 2 functions — baseline_int() and matrix(). If the program takes 60% in baseline_int() and remaining 30% in matrix().
  - If there exists an optimization that speedup baseline_int by 10x. What's the total speedup?
  - If there exists an optimization that speedup matrix by 30x. What's the total speedup?
  - What if we can apply both optimizations?
  - What if there is an overhead applying these optimizations?

Speedup of Y over X

- Consider the same program on the following two machines, X and Y. By how much Y is faster than X?

	Clock Rate	Instructions	Percentage of Type-A	CPI of Type-A	Percentage of Type-B	CPI of Type-B	Percentage of Type-C	CPI of Type-C
Machine X	3 GHz	5000000000	20%	8	20%	4	60%	1
Machine Y	5 GHz	5000000000	20%	13	20%	4	60%	1

How can deeper memory hierarchy help in performance?

- Assume that we have a processor running @ 2 GHz and a program with 30% of load/store instructions. If the computer has “perfect” memory, the CPI is just 1. Now, in addition to DDR4, whose latency 26 ns, we also got a 2-level SRAM caches with
  - it’s 1st-level one at latency of 0.5ns and can capture 90% of the desired data/instructions.
  - the 2nd-level at latency of 5ns and can capture 60% of the desired data/instructions

  What’s the average CPI?

Cache simulation

- The processor has a 32KB, 64B blocked, 4-way L1 cache. Consider the following code:

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

- What's the total miss rate? How many of the misses are compulsory misses? How many of the misses are conflict misses?
- How can you improve the cache performance of the above code through changing hardware?
- How can you improve the performance **without** changing hardware?

Announcement

- Assignment #2 due this Thursday
  - Please start early — some programs take very long time to finish
  - As mentioned in the beginning of each assignment — server busy cannot be a reason for late submission or extension
  - If you have questions that cannot be address online, please come to office hours
- Midterm next Tuesday
  - You can only take the exam in-person, closed-book, closed-note
  - Please bring your student ID and we will check
  - You may review/focus on the materials/topics covered in lectures
  - You SHOULD review your assignments

Computer
Science &
Engineering

203

つづく