Virtual memory

Hung-Wei Tseng

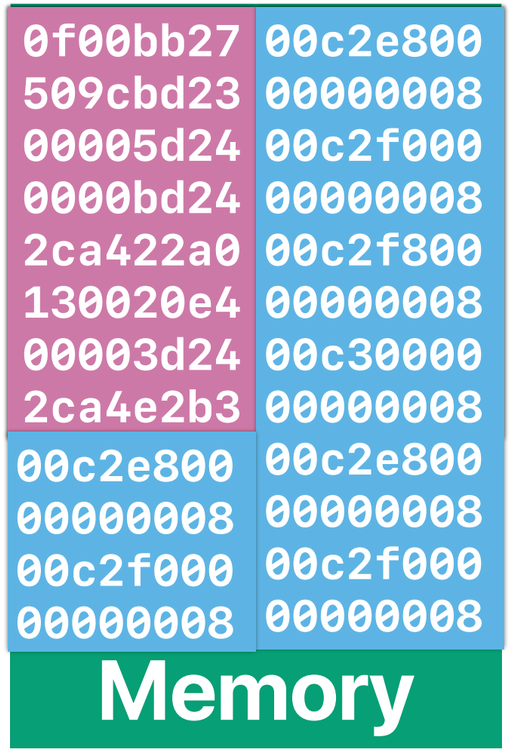
Why Virtual Memory?

If we expose memory directly to the processor (I)

00c2f800
00000008
00c30000
00000008

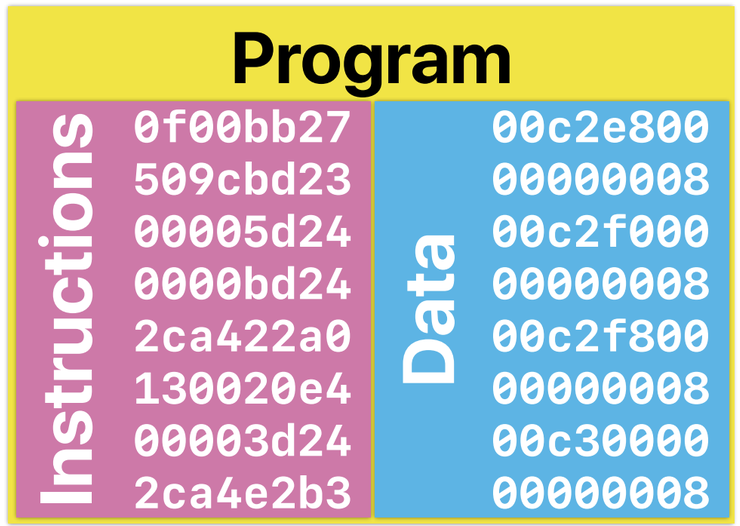
?

What if my program needs more memory?



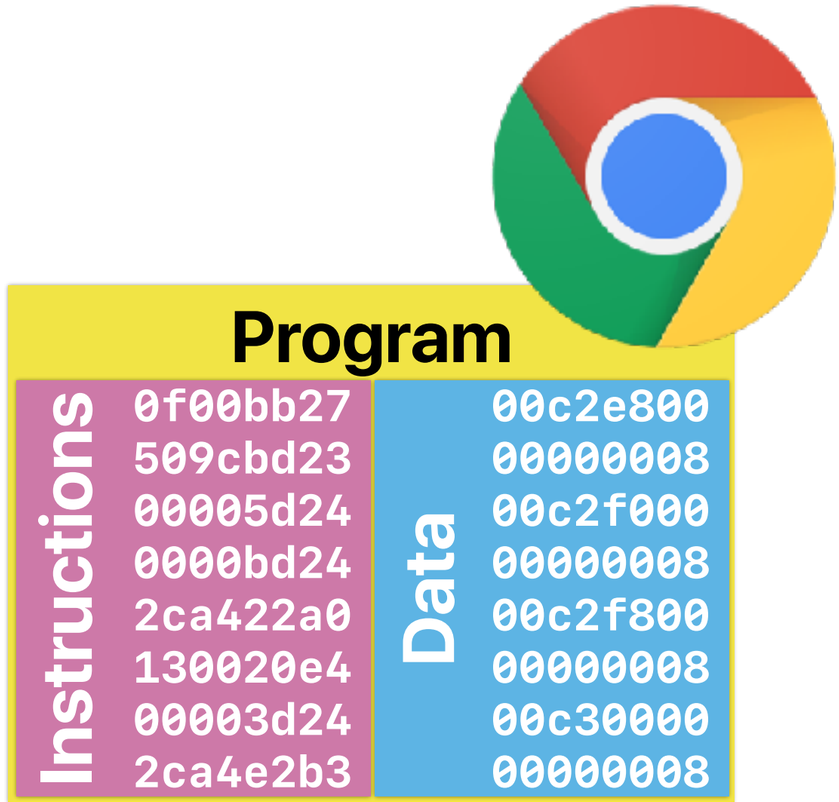
If we expose memory directly to the processor (II)

What if my program runs on a machine with a different memory size?



If we expose memory directly to the processor (III)

What if both programs need to use memory?



?

The Virtual Memory Abstraction

Virtual memory

instruction
0x0

data
0x80000000

data
0x80008000

instruction
0x0

data
0x80008000

Virtual Memory Space

Virtual Memory Space

Program

Instructions

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

Data

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

0f00bb27	00c2f800
509cbd23	00000008
00005d24	00c30000
0000bd24	00000008
0f00bb27	00c2f800
509cbd23	00000008
00005d24	00c30000
0000bd24	00000008

Memory

Program

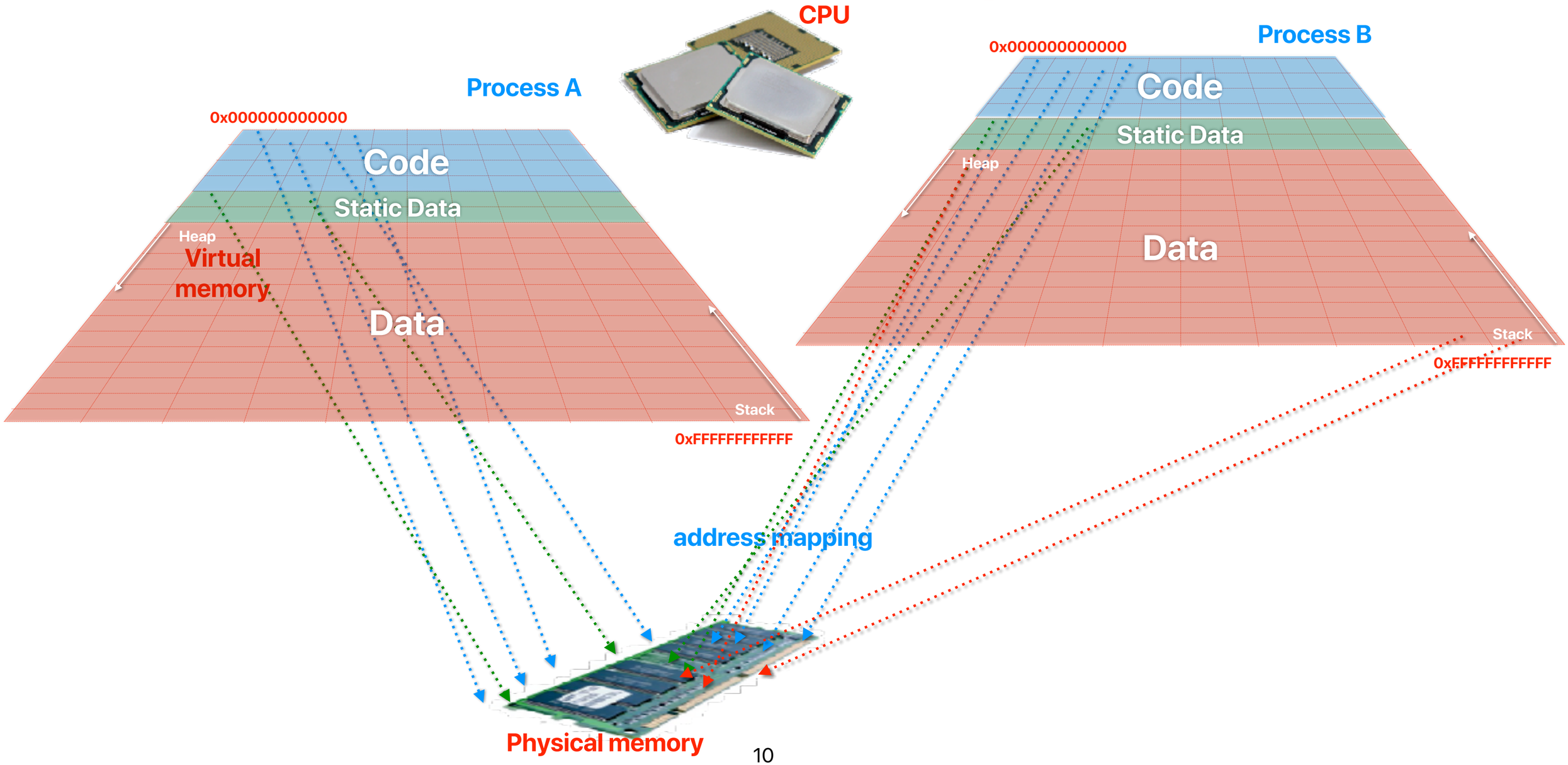
Instructions

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

Data

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

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**"

Demo revisited: Virtualization

```
double a;

int main(int argc, char *argv[])
{
    int cpu, status, i;
    int *address_from_malloc;
    cpu_set_t my_set;        // Define your cpu_set bit mask.
    CPU_ZERO(&my_set);       // Initialize it all to 0, i.e. no CPUs selected.
    CPU_SET(4, &my_set);     // set the bit that represents core 7.
    sched_setaffinity(0, sizeof(cpu_set_t), &my_set); // Set affinity of this process to the defined mask, i.e. only 7.
    status = syscall(SYS_getcpu, &cpu, NULL, NULL);
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s process_nickname\n",argv[0]);
        exit(1);
    }

    srand((int)time(NULL)+(int)getpid());
    a = rand();

    fprintf(stderr, "\nProcess %s is using CPU: %d. Value of a is %lf and address of a is %p\n",argv[1], cpu, a, &a);
    sleep(1);

    fprintf(stderr, "\nProcess %s is using CPU: %d. Value of a is %lf and address of a is %p\n",argv[1], cpu, a, &a);
    sleep(3);

    return 0;
}
```

getcpu system call to retrieve the executing CPU ID

create a random number

print the value of a and address of a

print the value of a and address of a again after sleep

Demo revisited

```
Process C is using CPU: 4. Value of a is 685161796.000000 and address of a is 0x6010b0
Process A is using CPU: 4. Value of a is 217757257.000000 and address of a is 0x6010b0
Process B is using CPU: 4. Value of a is 2057721479.000000 and address of a is 0x6010b0
Process D is using CPU: 4. Value of a is 1457934803.000000 and address of a is 0x6010b0
Process C is using CPU: 4. Value of a is 685161796.000000 and address of a is 0x6010b0
Process A is using CPU: 4. Value of a is 217757257.000000 and address of a is 0x6010b0
Process B is using CPU: 4. Value of a is 2057721479.000000 and address of a is 0x6010b0
Process D is using CPU: 4. Value of a is 1457934803.000000 and address of a is 0x6010b0
```

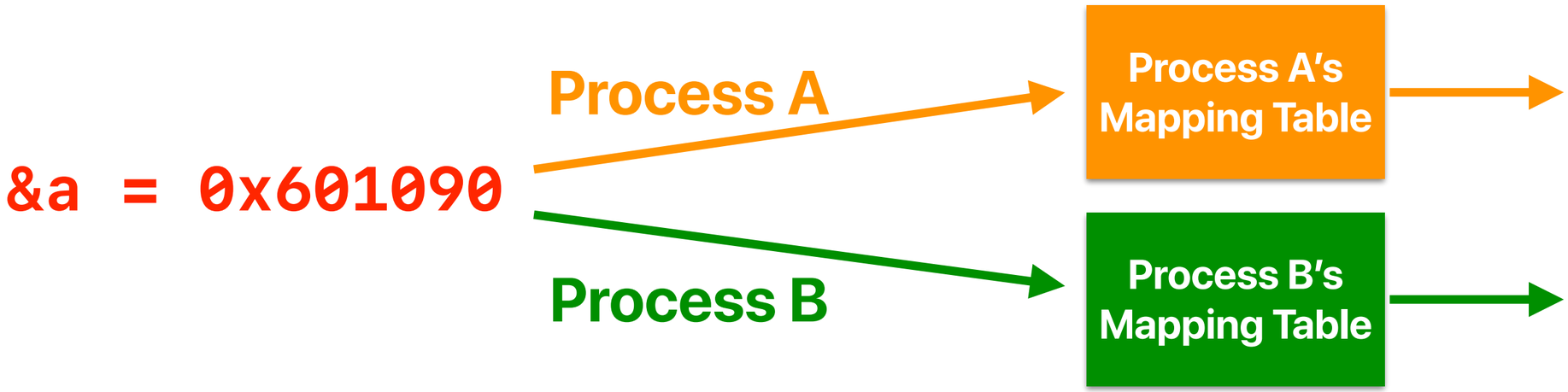
The same processor!

Different values

Different values are preserved

The same memory address!

Demo revisited

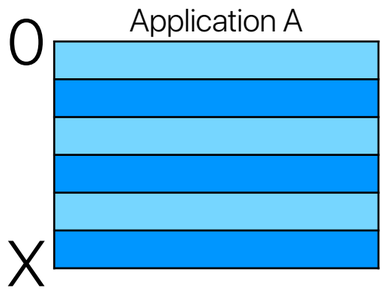


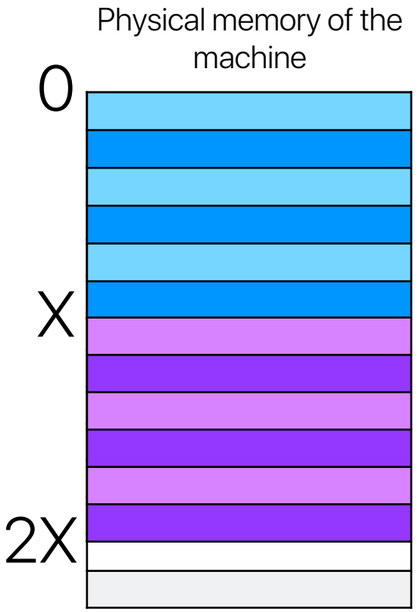

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

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 is using CPU: %d. Value of a is %lf and address of a is %p\n",getpid(), cpu, a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d is using CPU: %d. Value of a is %lf and address of a is %p\n",getpid(), cpu, a, &a);
    return 0;
}
```

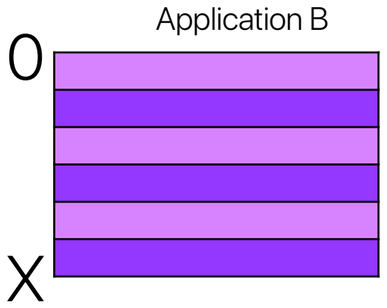
How to map from virtual to physical? Let’s start from segmentation

Segmentation

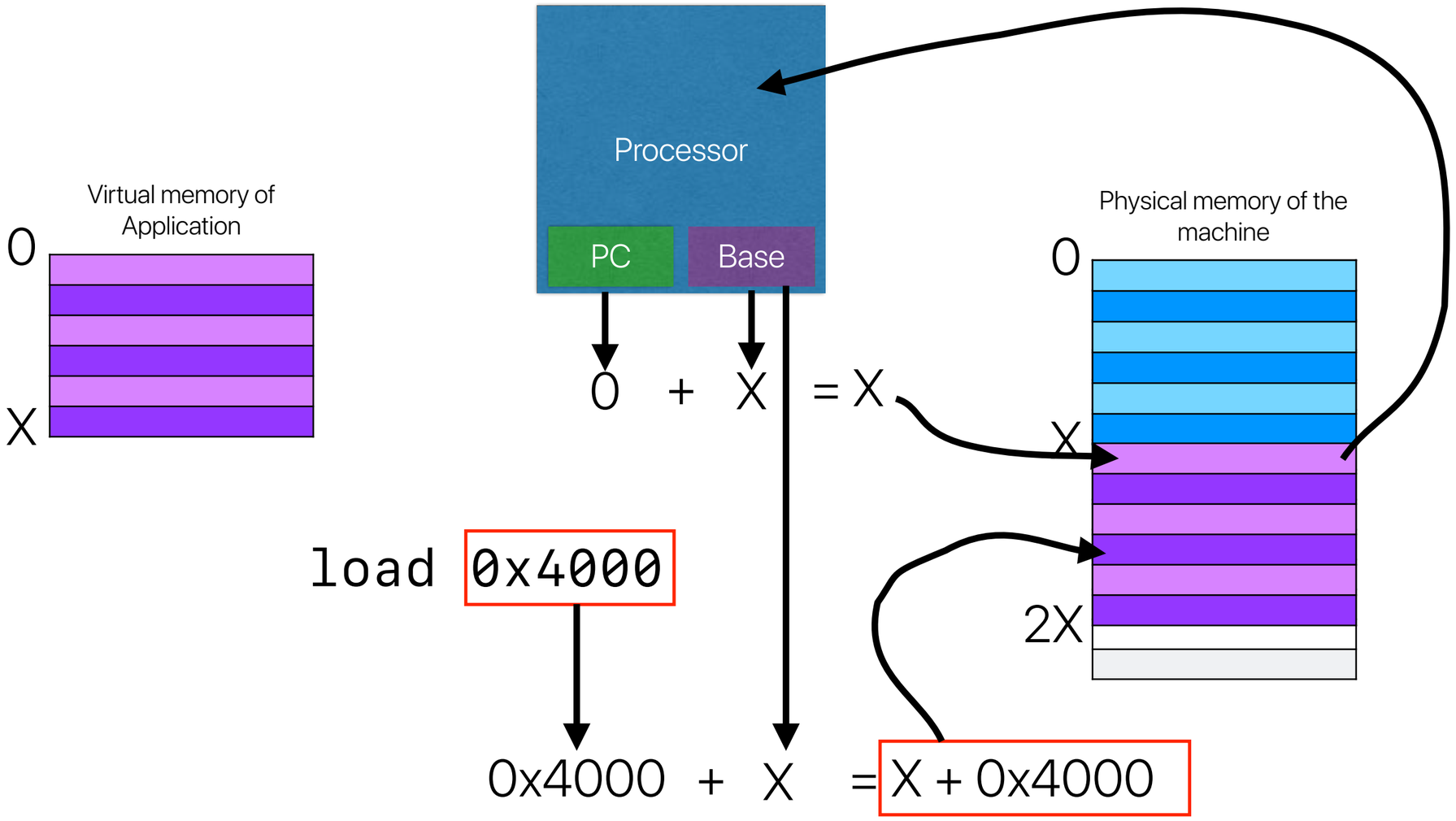
- The compiler generates code using virtual memory addresses
- The OS works together with hardware to partition physical memory space into segments for each running application
- The hardware dynamically translates virtual addresses into physical memory addresses



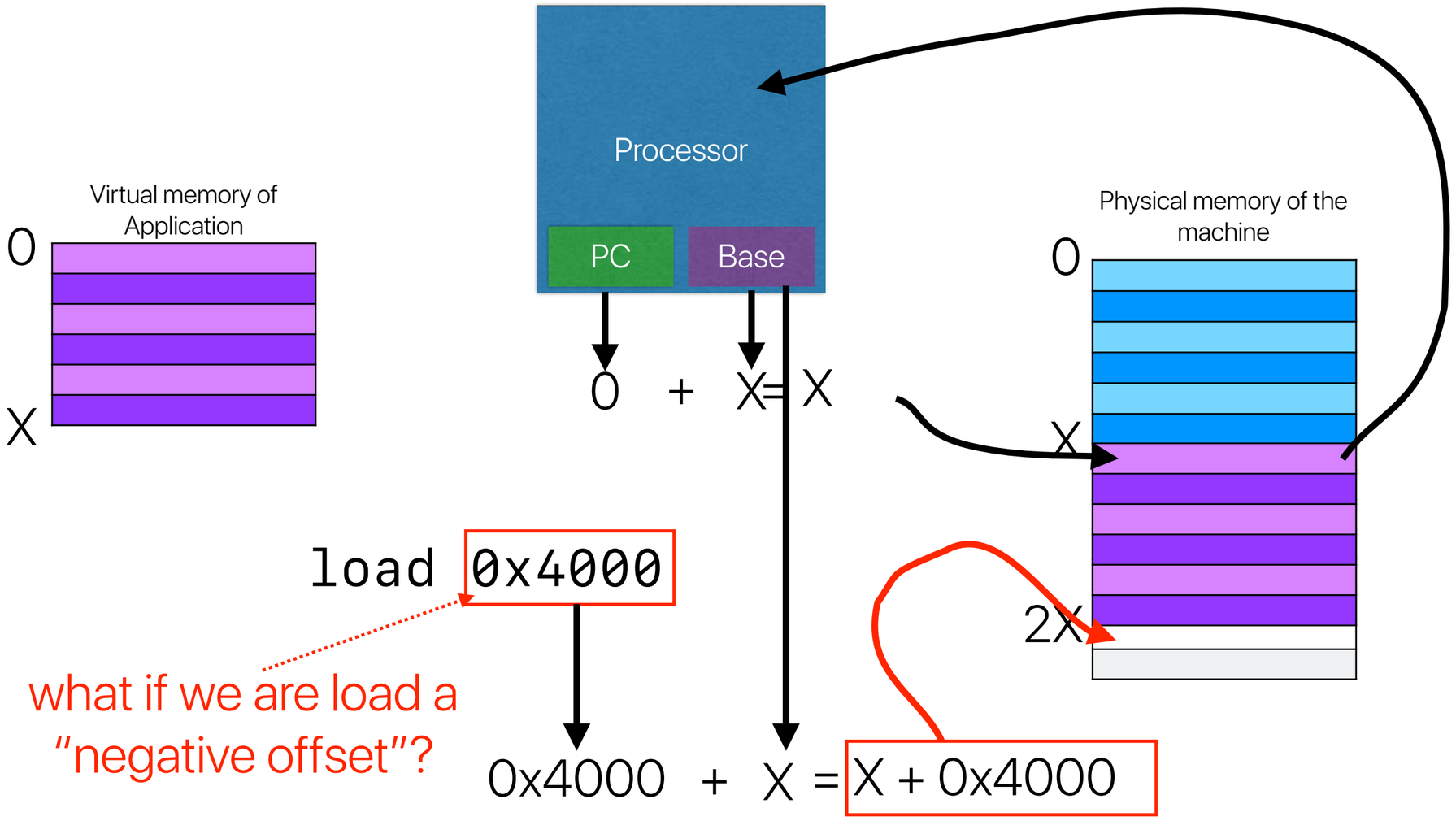




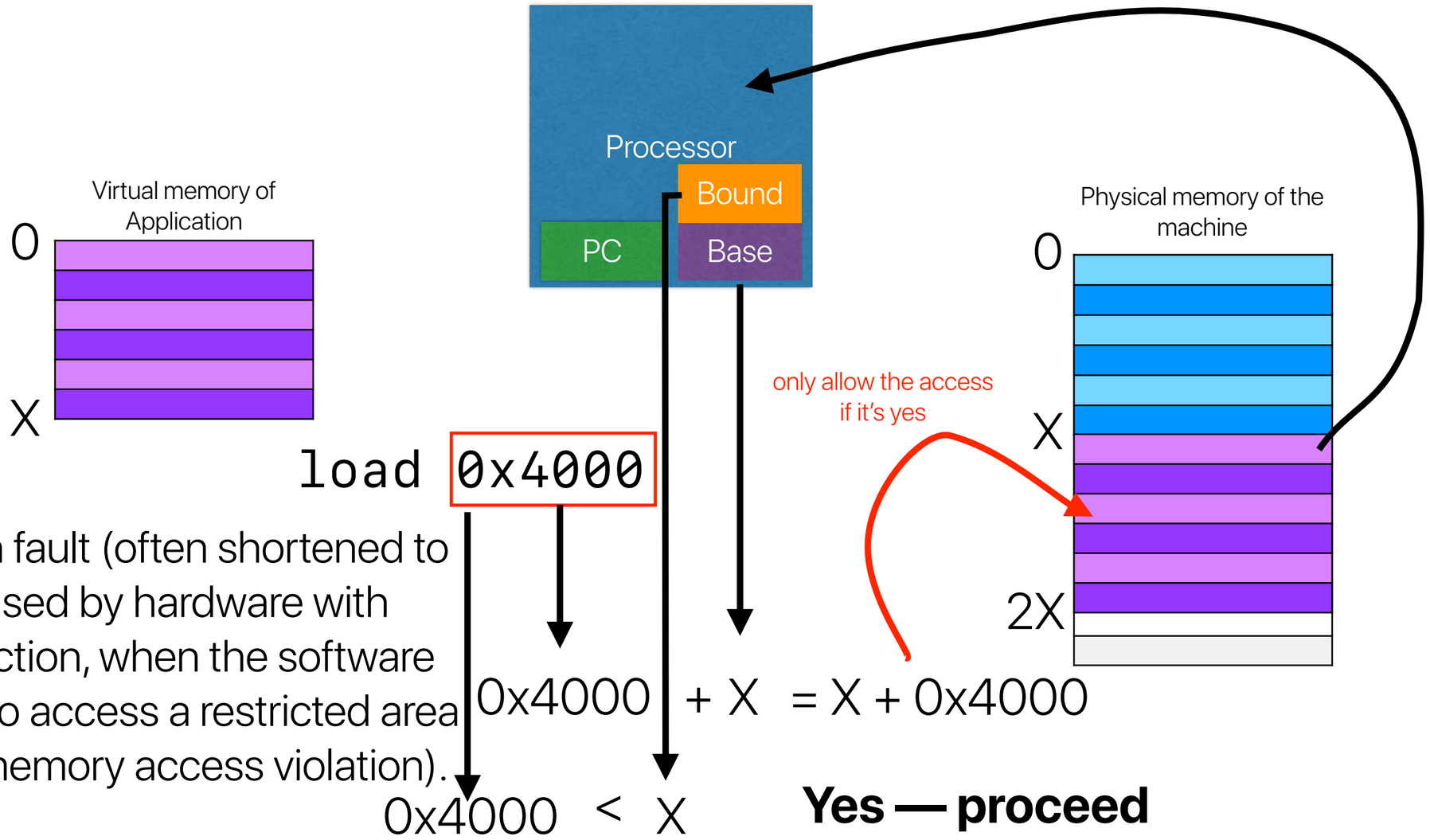
Address translation in segmentation



Protection again malicious processes

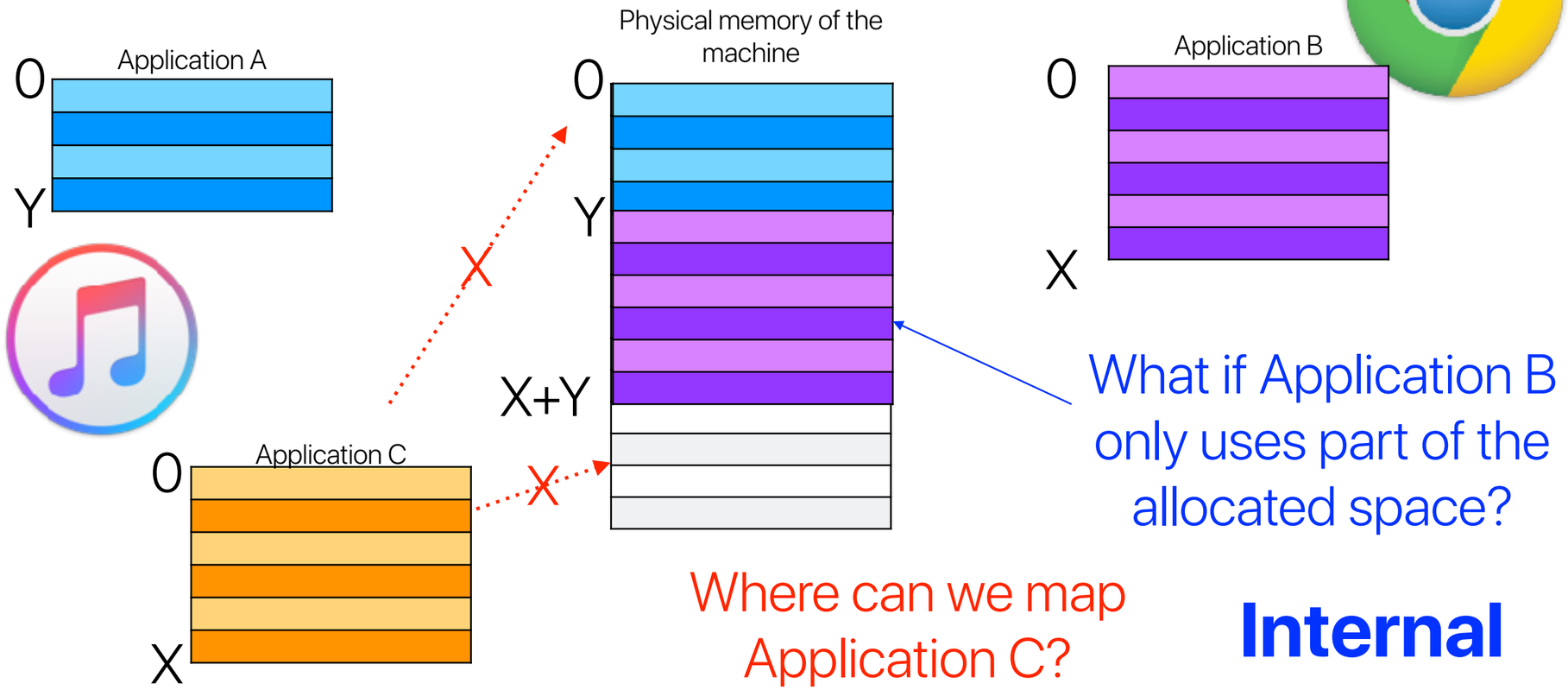


Protection again malicious processes



a segmentation fault (often shortened to segfault), raised by hardware with memory protection, when the software has attempted to access a restricted area of memory (a memory access violation).

What if?



What if Application B only uses part of the allocated space?

Where can we map Application C?

External Fragment

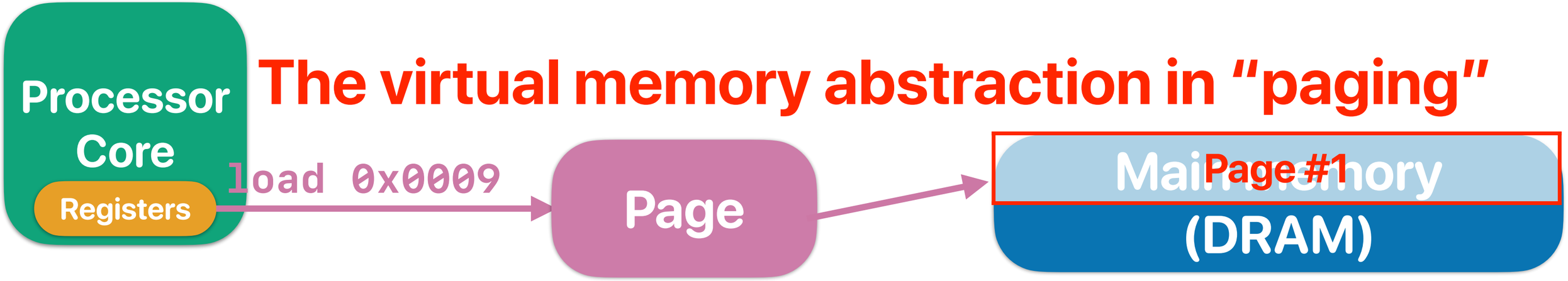
Even though we have space, we still cannot map App. C

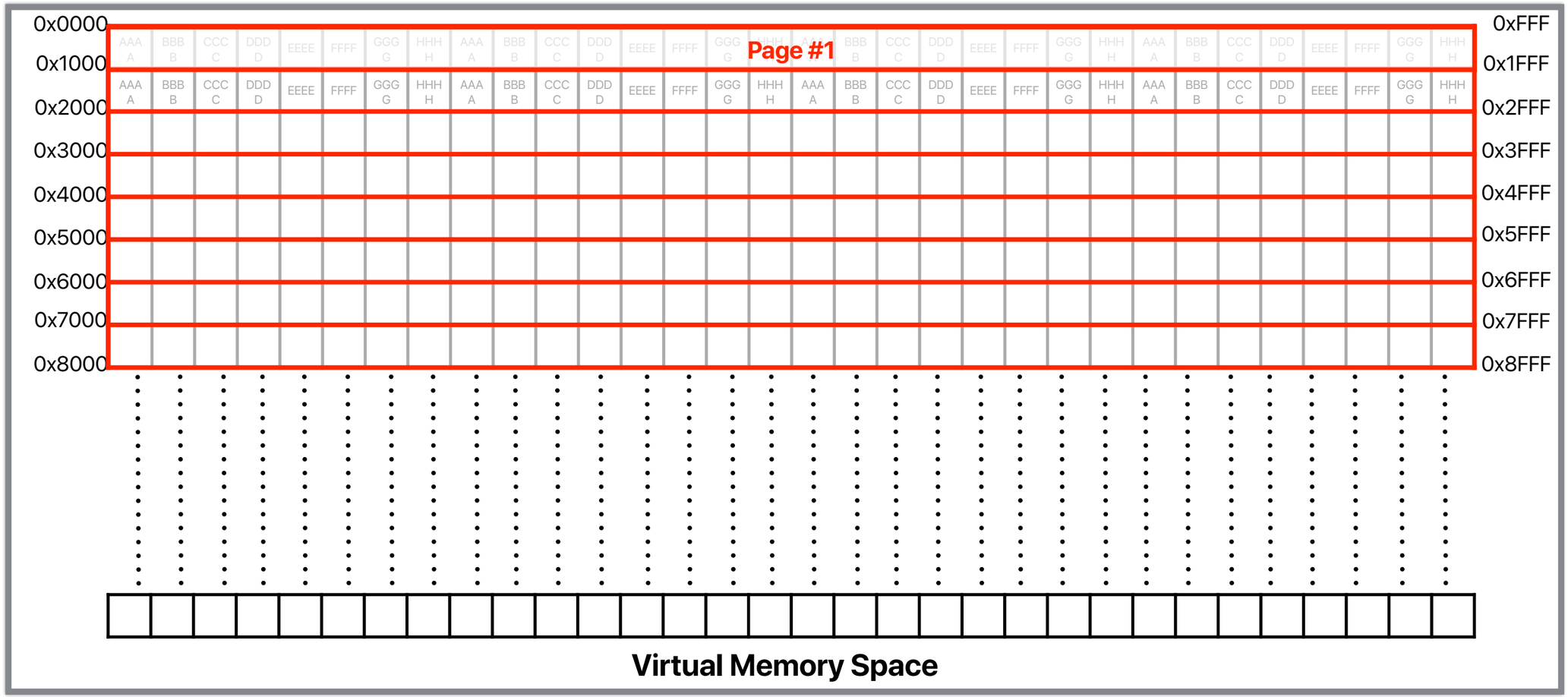
Internal Fragment

We waste some space in the allocated segment

When to create a virtual to physical address mapping? — Demand paging

The virtual memory abstraction in “paging”





Demand paging

instruction
0x0

data
0x80000000

Page fault!

Virtual Address Space for Chrome

Page fault!

instruction
0x0

Page fault!

data
0x80008000

Page fault!

Virtual Address Space for Apple Music

Program

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

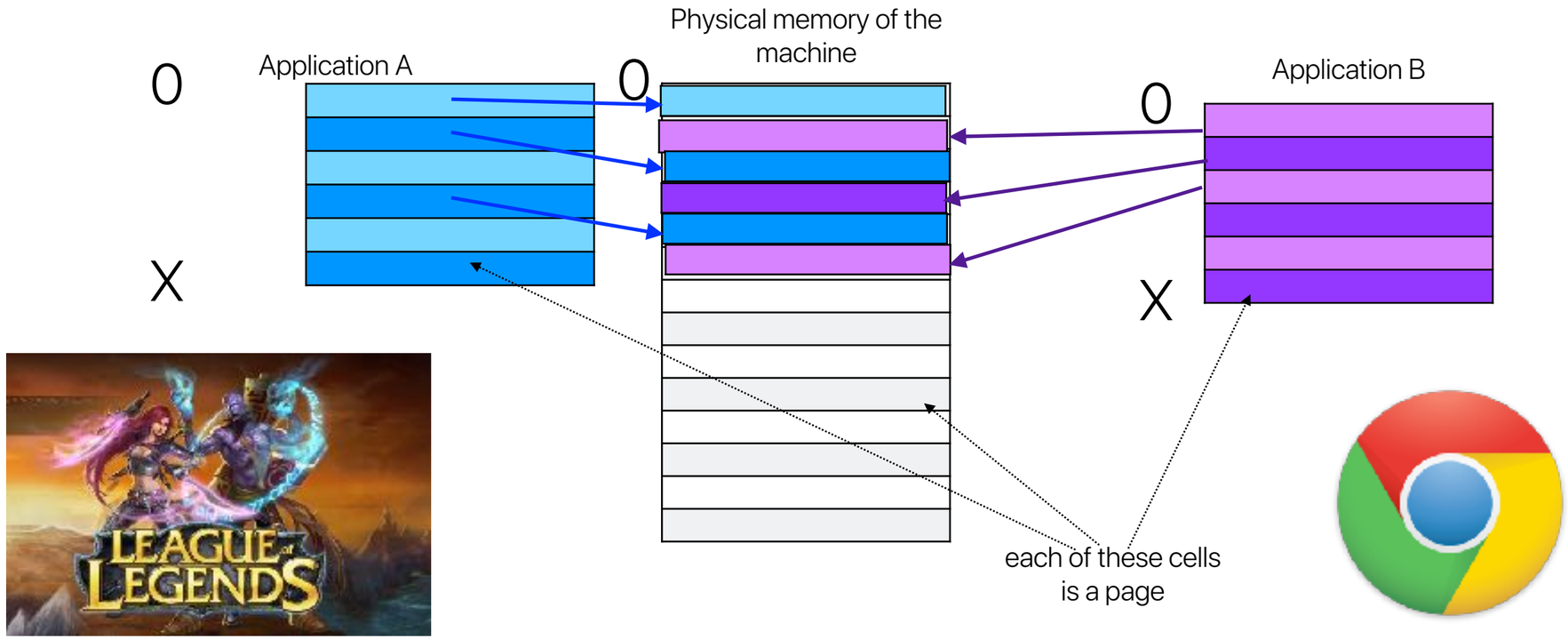
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
0f00bb27	00c2f800
509cbd23	00000008
00005d24	00c30000
0000bd24	00000008

Memory

Program

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

Demand paging



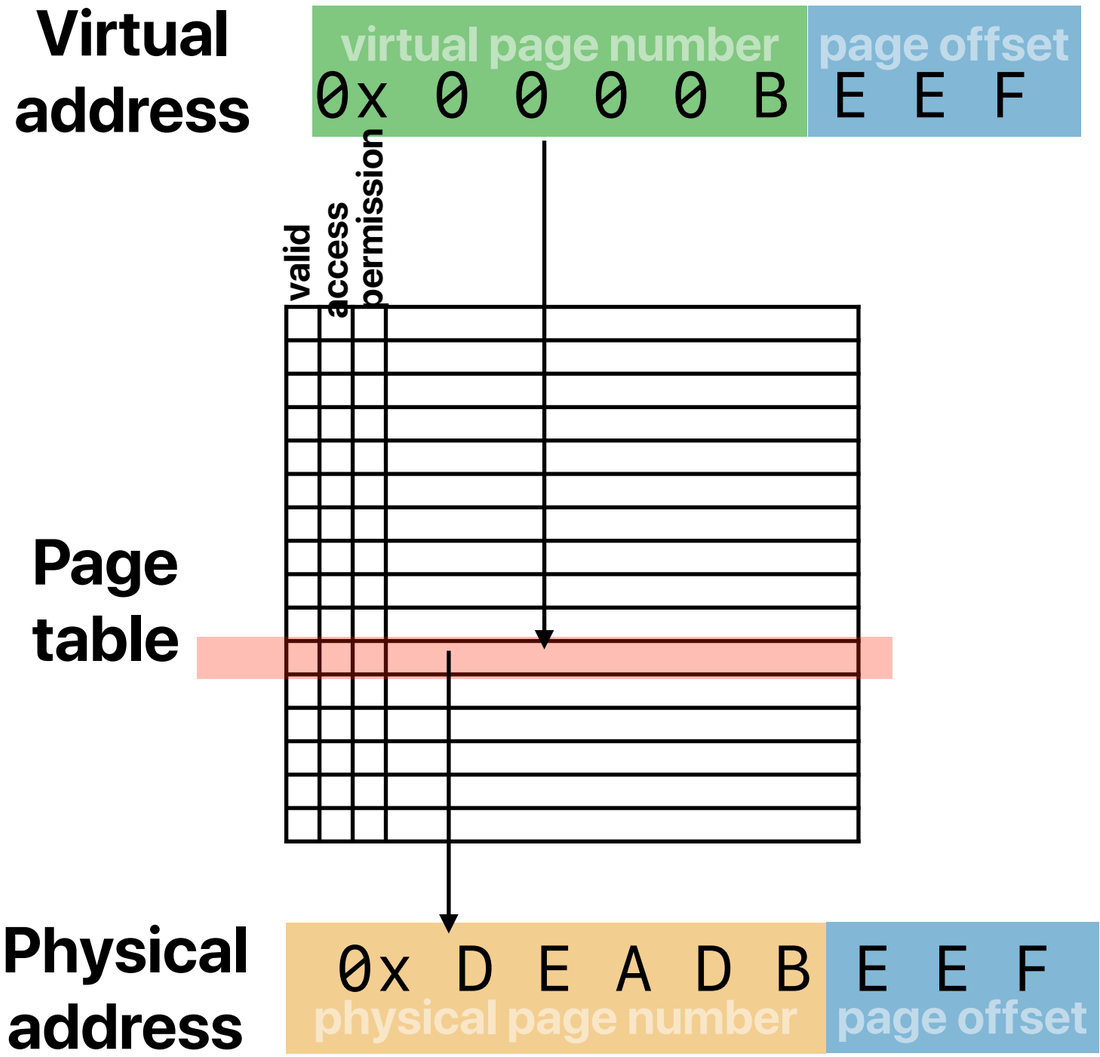
Terminology of Demand paging

- **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)
  - There is also **shadow paging** used by embedded systems, mobile phones — they load the whole program/data into the physical memory when you launch it

Address translation in 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
- In addition to valid bit and physical page #, the page table may also store
  - Reference bit
  - Modified bit
  - Permissions

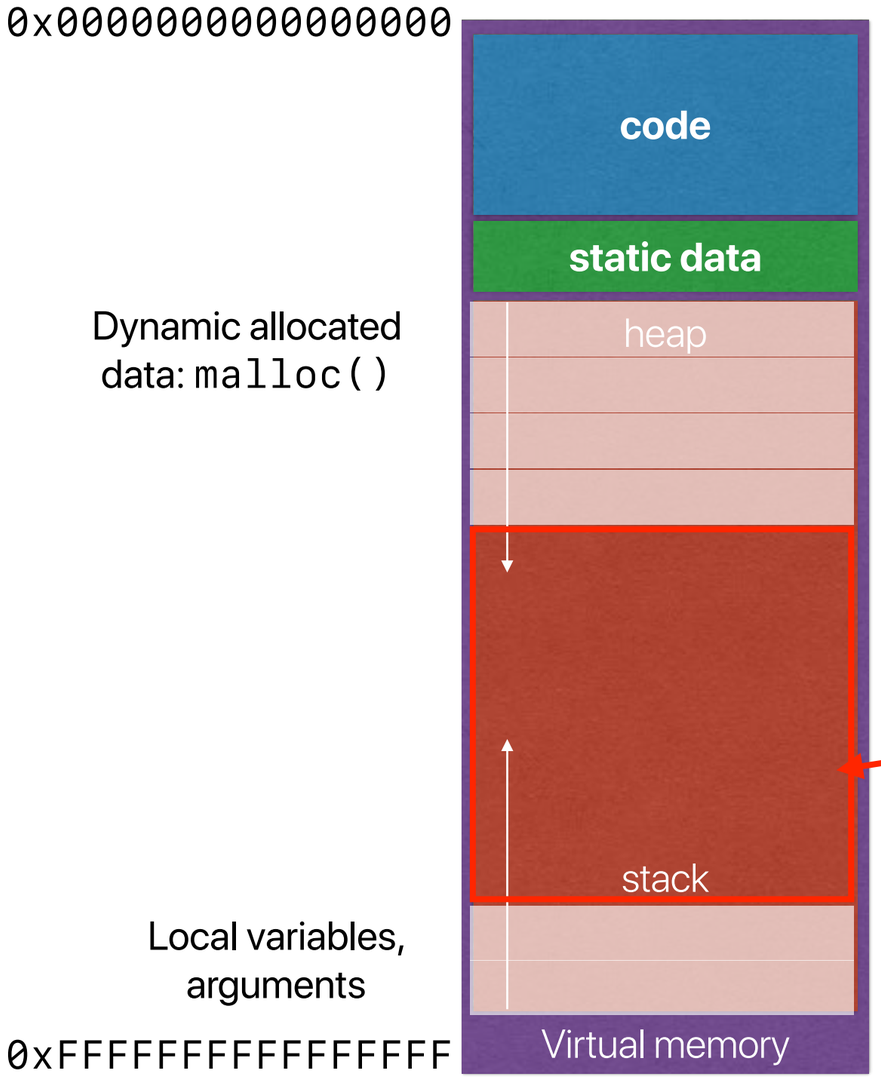


Address translation (cont.)

- Page tables are too large to be kept on the chip (millions of entries) — space overhead
- Instead, the page tables are kept in memory — memory access overhead
- Address translation in x86
  - A special register, the page table base register (PTBR), points to the beginning of the page table for the running process
  - The CPU walks through the page table to figure out the mapping
  - The contents of this register must be changed during a context switch

Smaller page tables

Do we really need a large table?



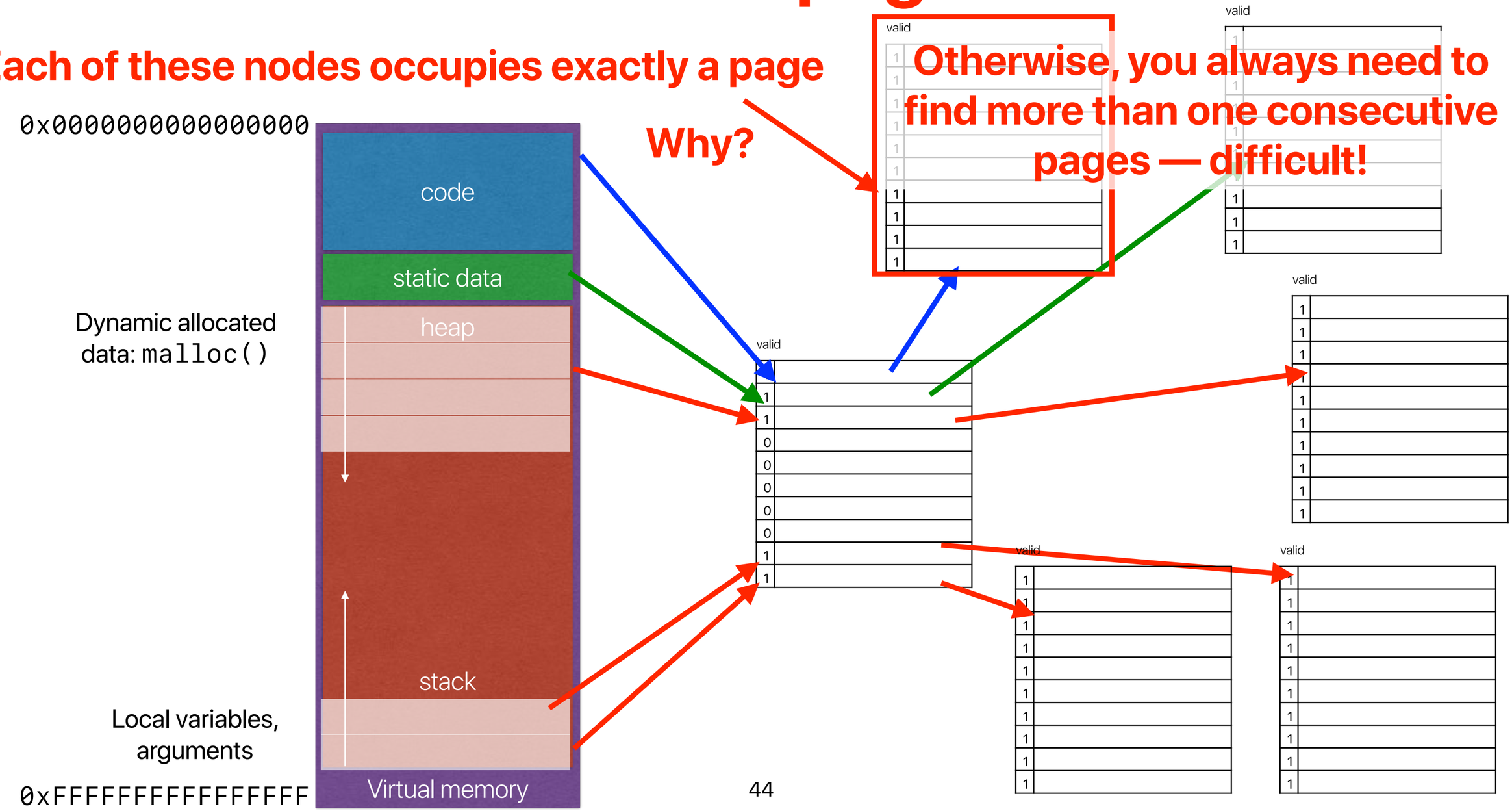
Your program probably never uses this huge area!

Hierarchical page table

Each of these nodes occupies exactly a page

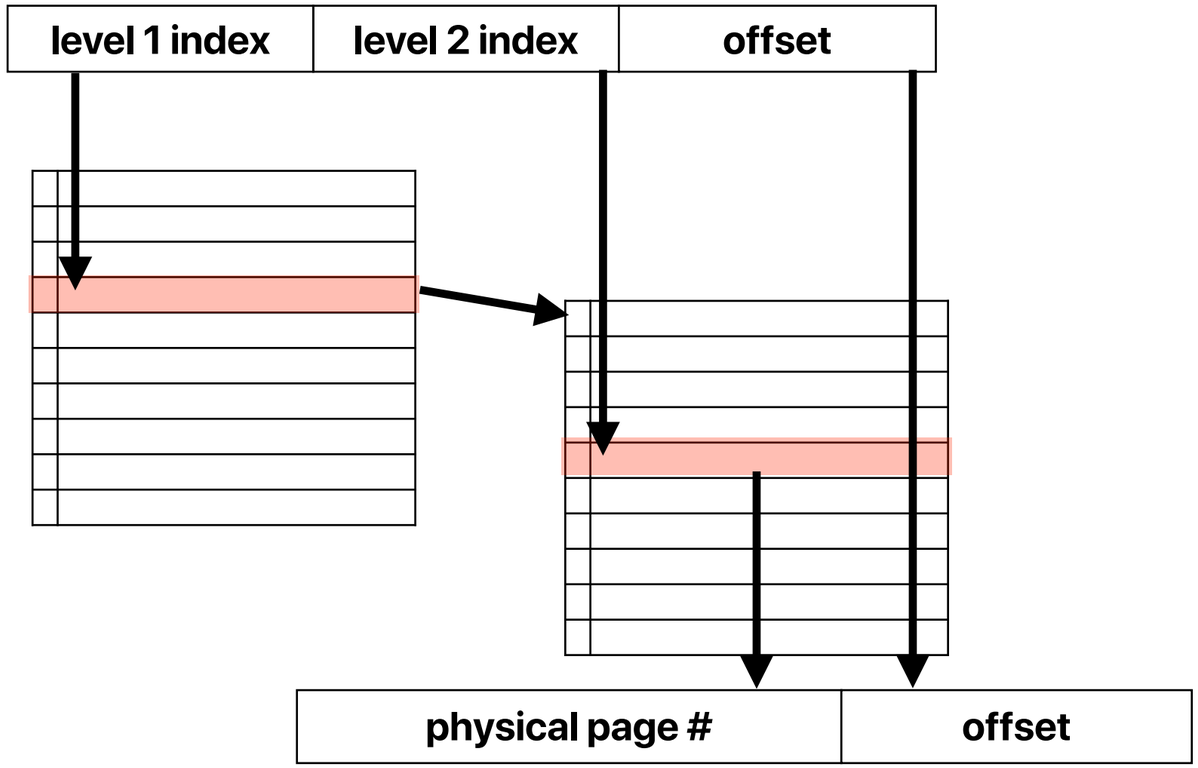
Why?

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



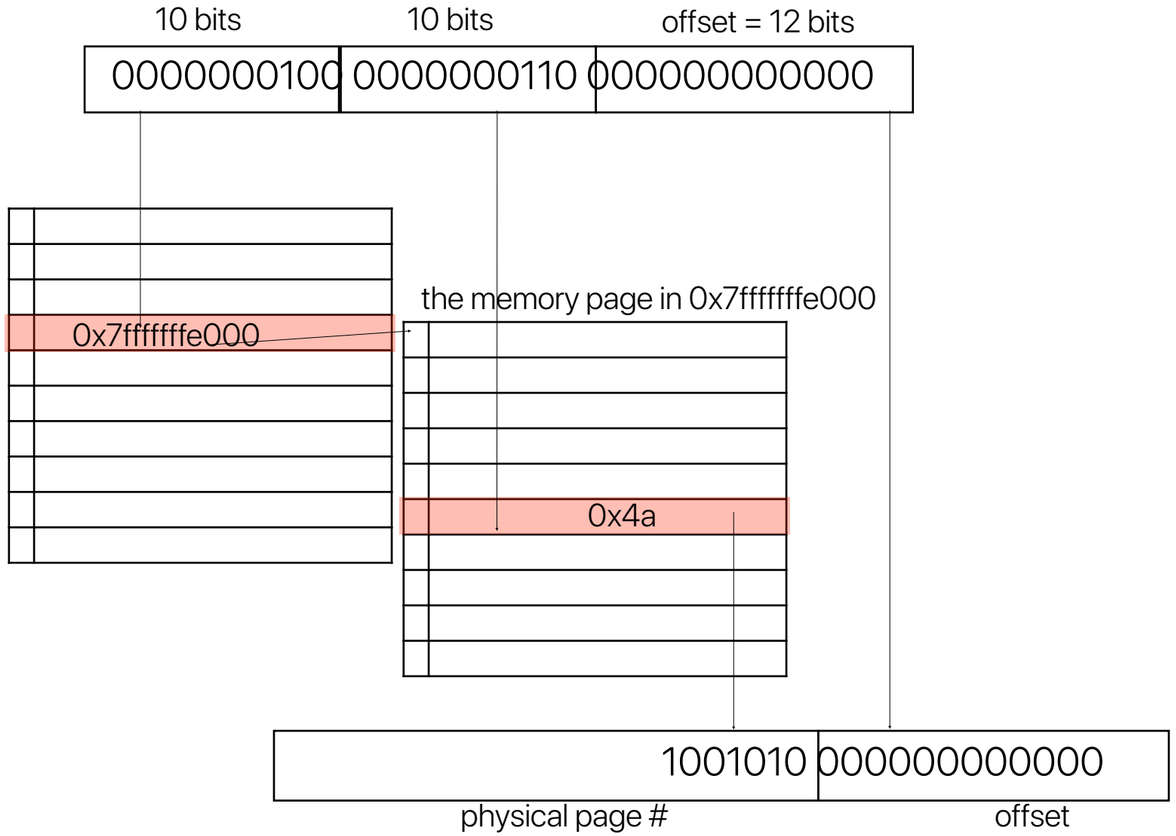
Hierarchical page table

- Break the virtual page number into several pieces
- If one piece has N bits, build an 2^N-ary tree
- Only store the part of the tree that contain valid pages
- Walk down the tree to translate the virtual address



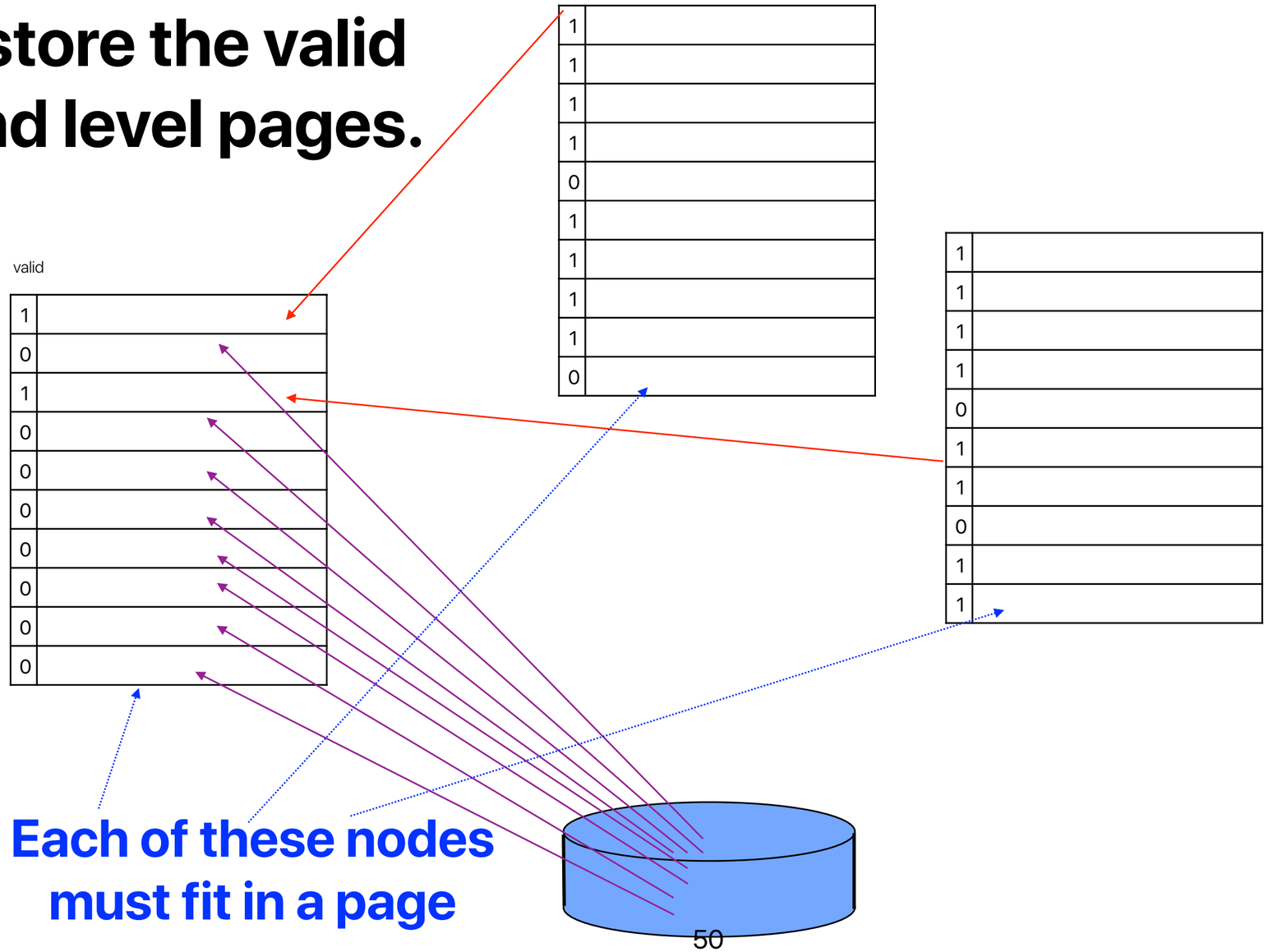
Page table walking example

- Two-level, 4KB, 10 bits index in each level
- If we are accessing 0x1006000 now...

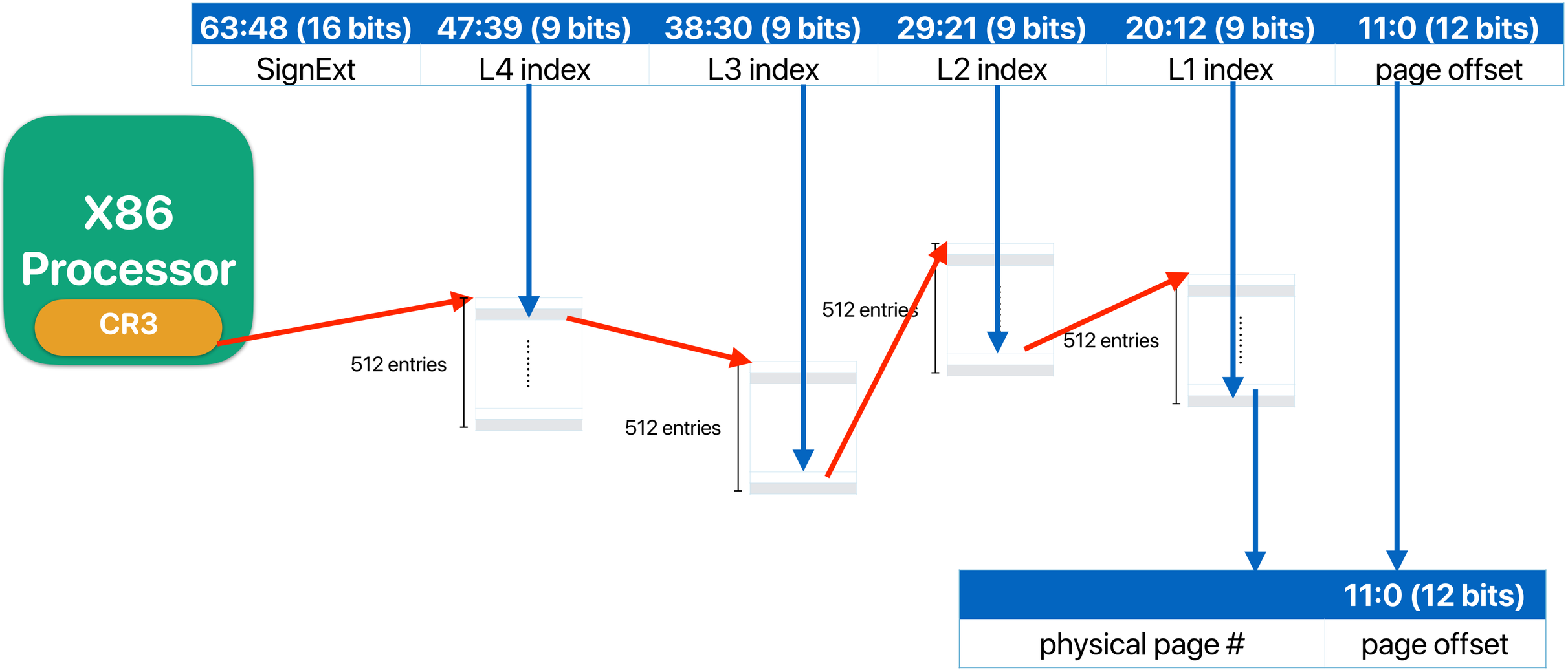


Hierarchical page table

- **Only store the valid second level pages.**

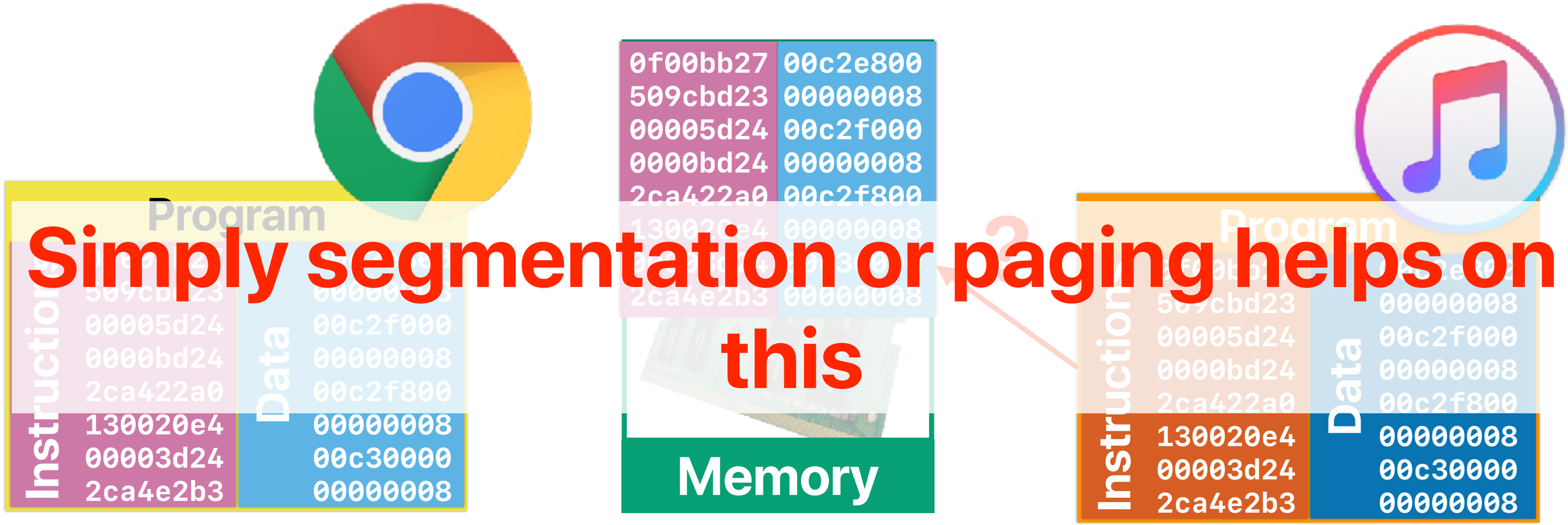


Case study: Address translation in x86-64



If we expose memory directly to the processor (III)

What if both programs need to use memory?



If we expose memory directly to the processor (I)

00c2f800
00000008
00c30000
00000008

?

What if my program needs more memory?

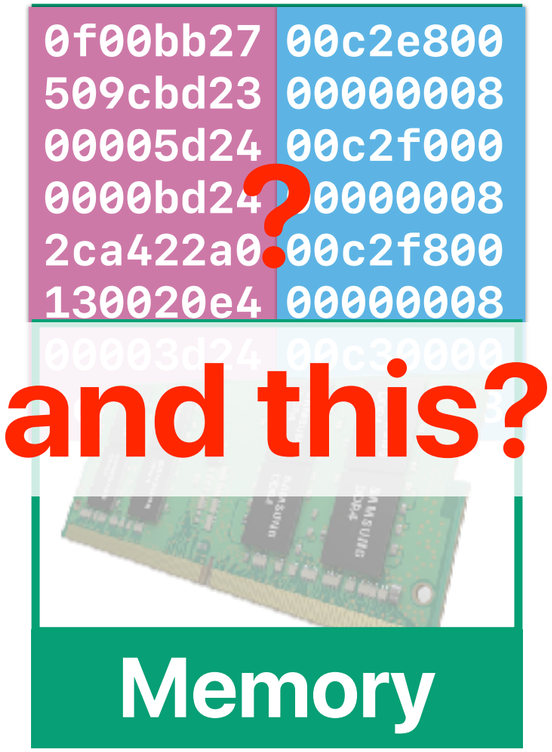
But how about this?

If we expose memory directly to the processor (II)

What if my program runs on a machine with a different memory size?

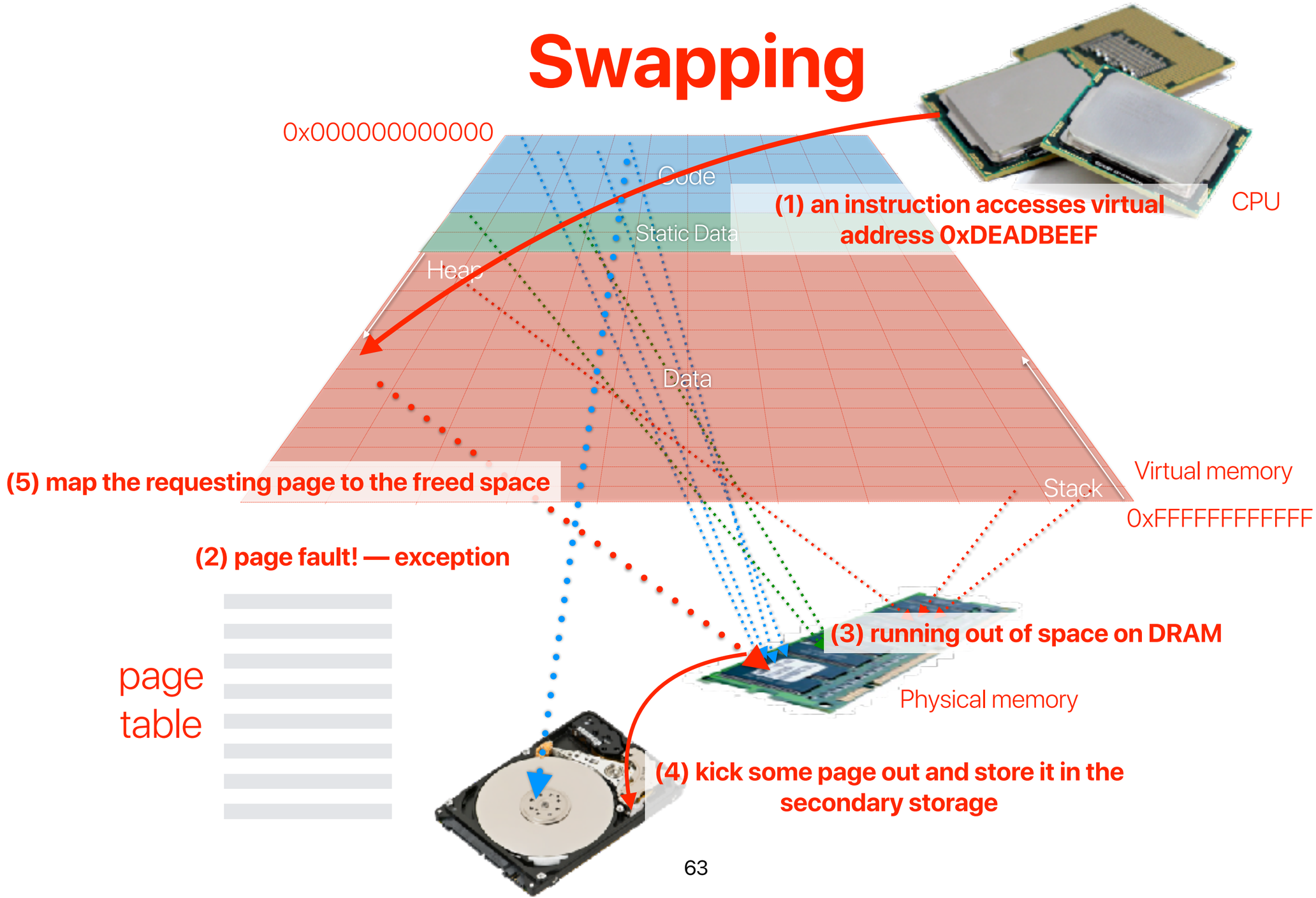
Program

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



Swapping — the mechanism when we run out of physical memory

Swapping


0x000000000000
Code
Static Data
Heap
Data
Stack
Virtual memory
0xFFFFFFFFFFFF
CPU
(1) an instruction accesses virtual address 0xDEADBEEF
(2) page fault! — exception
(3) running out of space on DRAM
Physical memory
(4) kick some page out and store it in the secondary storage
(5) map the requesting page to the freed space
page table

The mechanism: swap space in disks

- Reserve space on disks
- When you need to make rooms in the physical main memory, allocate a page in the swap space and put the content of the evicted page there
- When you need to reference a page in the swap space, make a room in the physical main memory and swap the disk space with the evicted page

Latency Numbers Every Programmer Should Know

Operations	Latency (ns)	Latency (us)	Latency (ms)	
L1 cache reference	0.5 ns			~ 1 CPU cycle
Branch mispredict	5 ns			
L2 cache reference	7 ns			14x L1 cache
Mutex lock/unlock	25 ns			
Main memory reference	100 ns			20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000 ns	3 us		
Send 1K bytes over 1 Gbps network	10,000 ns	10 us		
Read 4K randomly from SSD*	150,000 ns	150 us		~1GB/sec SSD
Read 1 MB sequentially from memory	250,000 ns	250 us		
Round trip within same datacenter	500,000 ns	500 us		
Read 1 MB sequentially from SSD*	1,000,000 ns	1,000 us	1 ms	~1GB/sec SSD, 4X memory
Disk seek	10,000,000 ns	10,000 us	10 ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000 ns	20,000 us	20 ms	80x memory, 20X SSD
Send packet CA-Netherlands-CA	150,000,000 ns	150,000 us	150 ms	

Swapping/Page replacement policies

Page replacement policy

- Goal: Identify page to remove that will avoid future page faults (i.e. utilize locality as much as possible)
- Implementation Goal: Minimize the amount of software and hardware overhead
  - Example:
    - Memory (i.e. RAM) access time: 100ns
    - Disk access time: 10ms
    - P_f: probability of a page fault
    - Effective Access Time = $10^{-7} + P_f * 10^{-3}$
  - When $P_f = 0.001$:
    Effective Access Time = 10,100ns
  - Takeaway: Disk access tolerable only when it is extremely rare

Page replacement algorithms

- Oracle: Replace the page whose next use is farthest in the future
- FIFO: Replace the oldest page
- LRU: Replace page that was the least recently used (longest since last use)

Oracle/FIFO/LRU

	Oracle	FIFO	LRU
Implementation	Impossible! — We cannot see the future	Easy — circular queue	May require hardware support or linked list or additional timestamps in page tables
Execution overhead		Low	High — you need to manipulate the list or update every counter
Performance	Optimal, but impossible	Usually not as good as LRU	Usually better than FIFO

Page replacement algorithms

- Oracle: Replace the page whose next use is farthest in the future
  - Pro: Optimal (provides reference for others)
  - Con: We can’t see into the future…
- FIFO: Replace the oldest page
  - Pro: Simple (i.e. fast implementation!)
  - Con: Often performs poorly
- LRU: Replace page that was the least recently used (longest since last use)
  - Pro: Usually good performance
  - Con: High implementation overhead

Announcement

- Reading quiz due next Tuesday
- Project due date 3/3
  - In about a month
  - Please come to our office hours if you need help for your projects