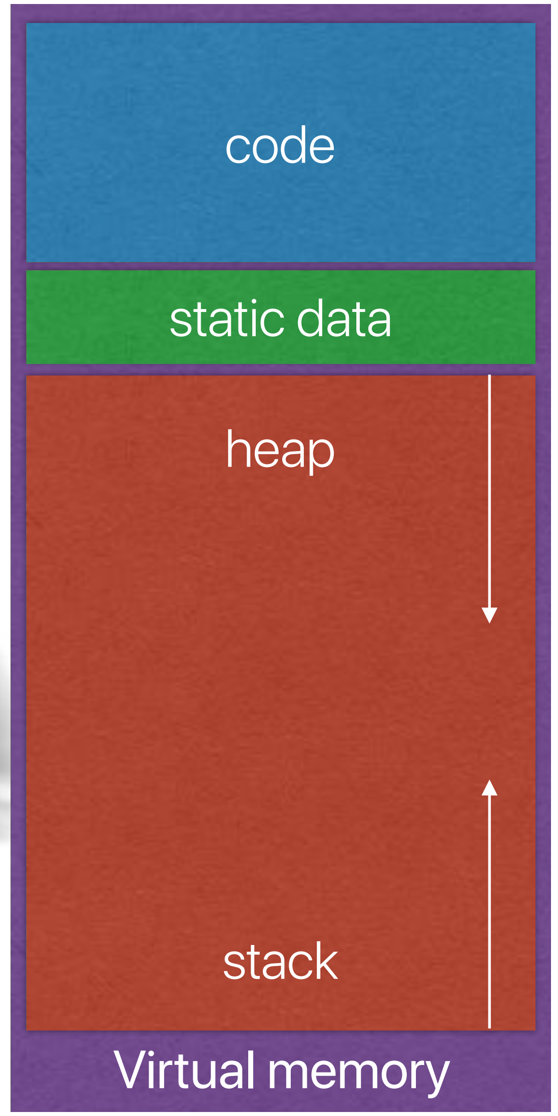
Virtual memory (I)

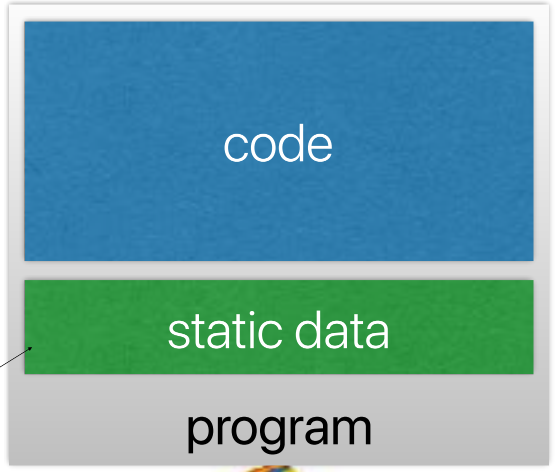
Hung-Wei Tseng

What happens when creating a process



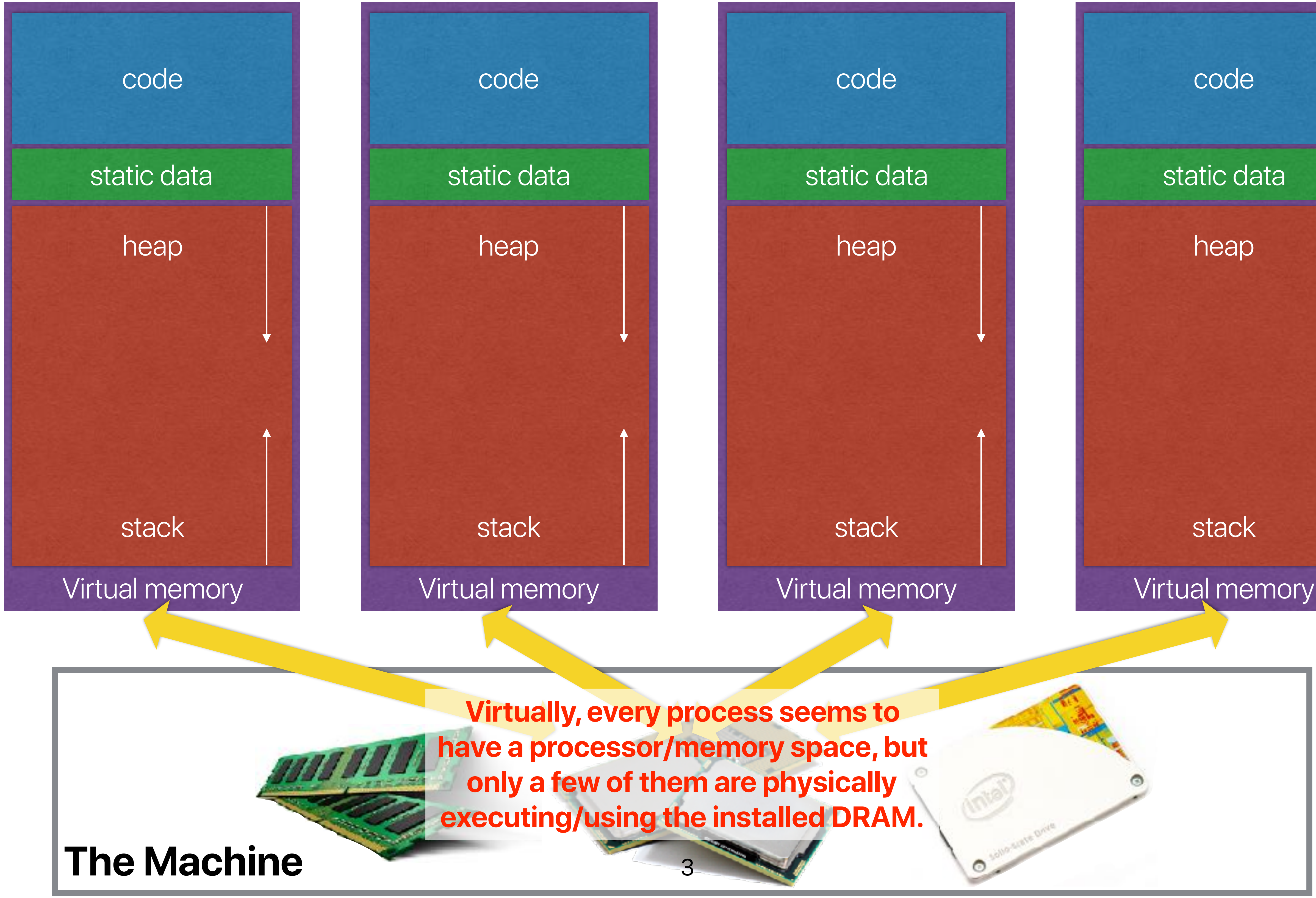
Dynamic allocated data: `malloc()`

Local variables,
arguments



Linux contains a .bss section
for uninitialized global variables

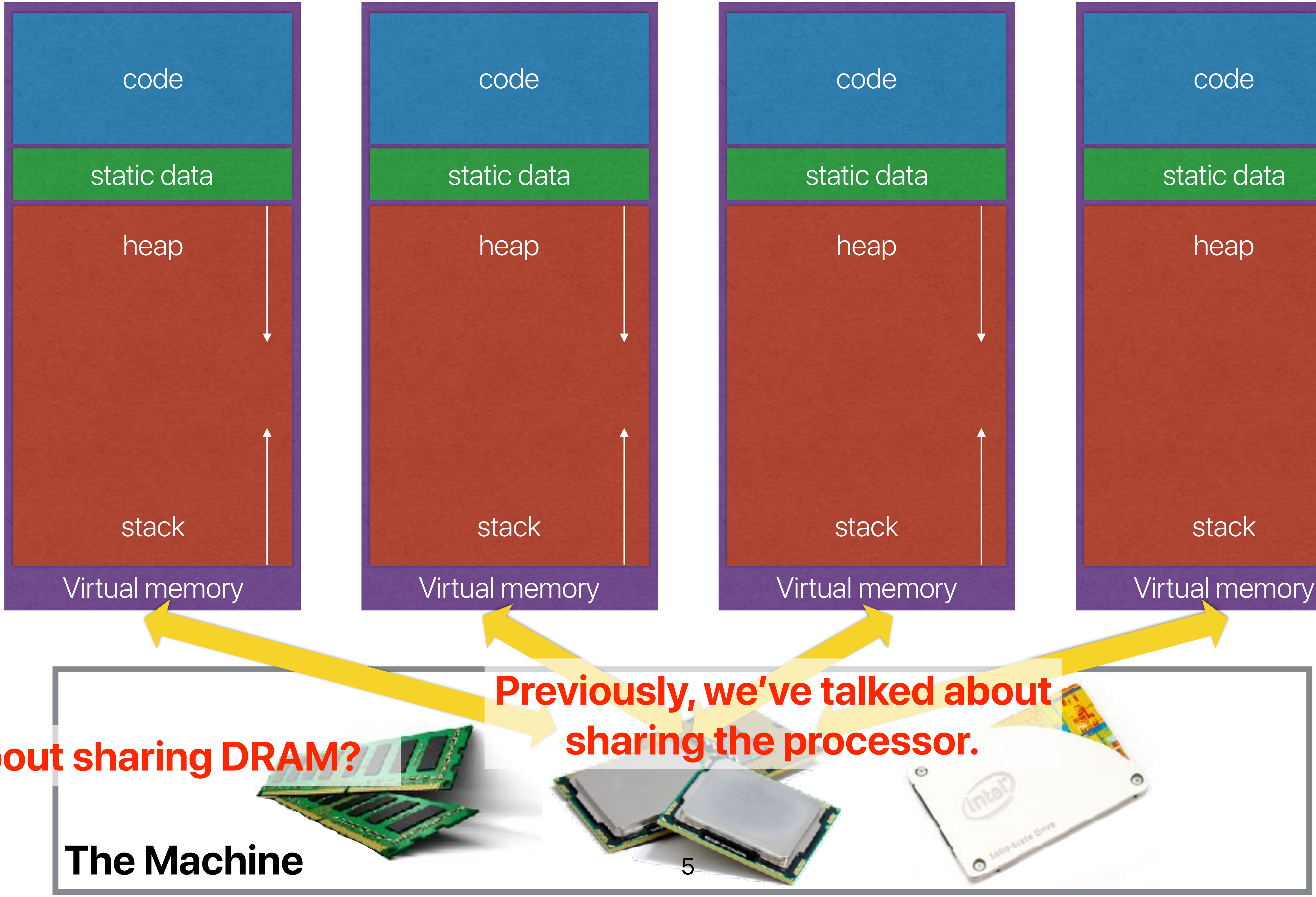
Previously, we talked about virtualization



Virtualizing the processor

- The mechanism
  - Non-preemptive/cooperative: the run process itself initiate context switches — by using system calls
  - Preemptive: the OS kernel can actively incur context switches — by using hardware (timer) interrupts
- The policy
  - Non-preemptive
    - First Come First Serve
    - ~~Shortest job first: SJF~~
  - Preemptive
    - Round robin
    - ~~Shortest Time-to-completion~~
    - Multi-level scheduling algorithm

Previously, we talked about virtualization



Previously, we've talked about sharing the processor.

How about sharing DRAM?

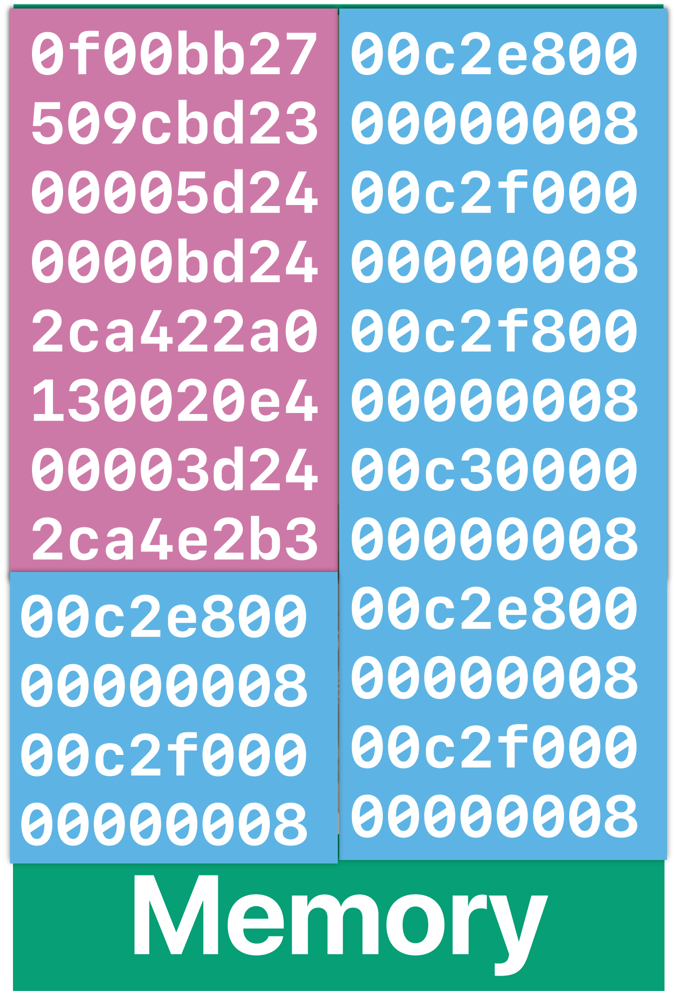
Outline

- Why virtualize your memory
- Start with the basic proposal — segmentation
- Demand paging

Why Virtual Memory?

If we expose memory directly to the processor (I)

What if my program needs more memory?

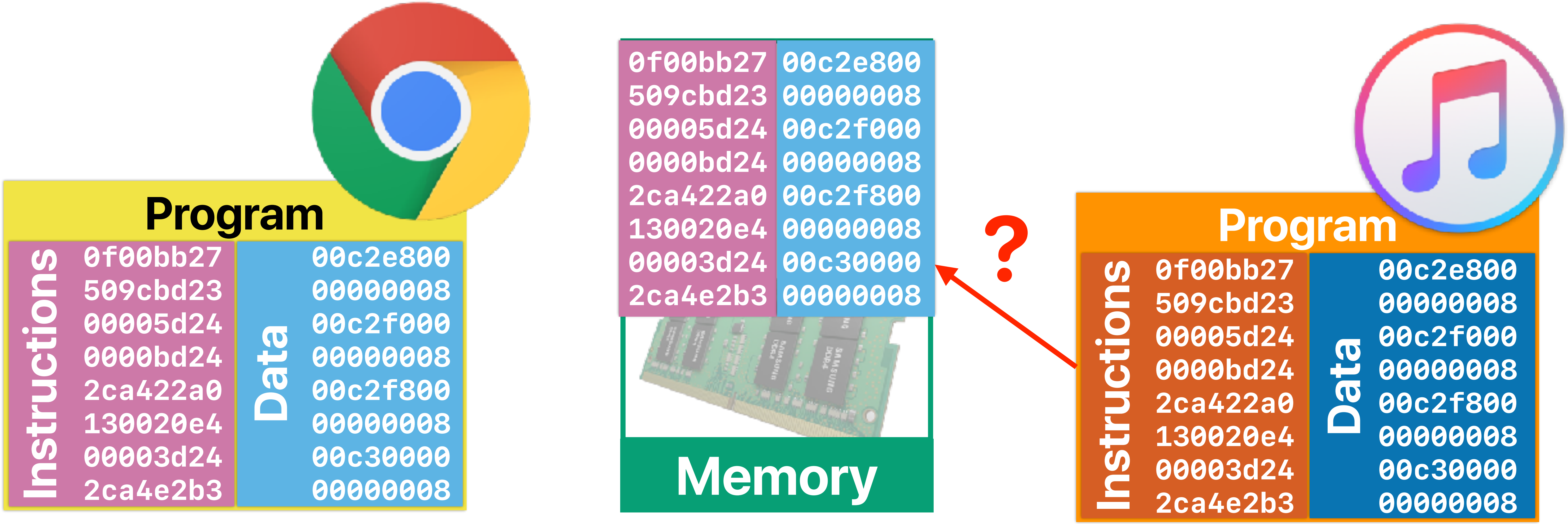


If we expose memory directly to the processor (II)

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

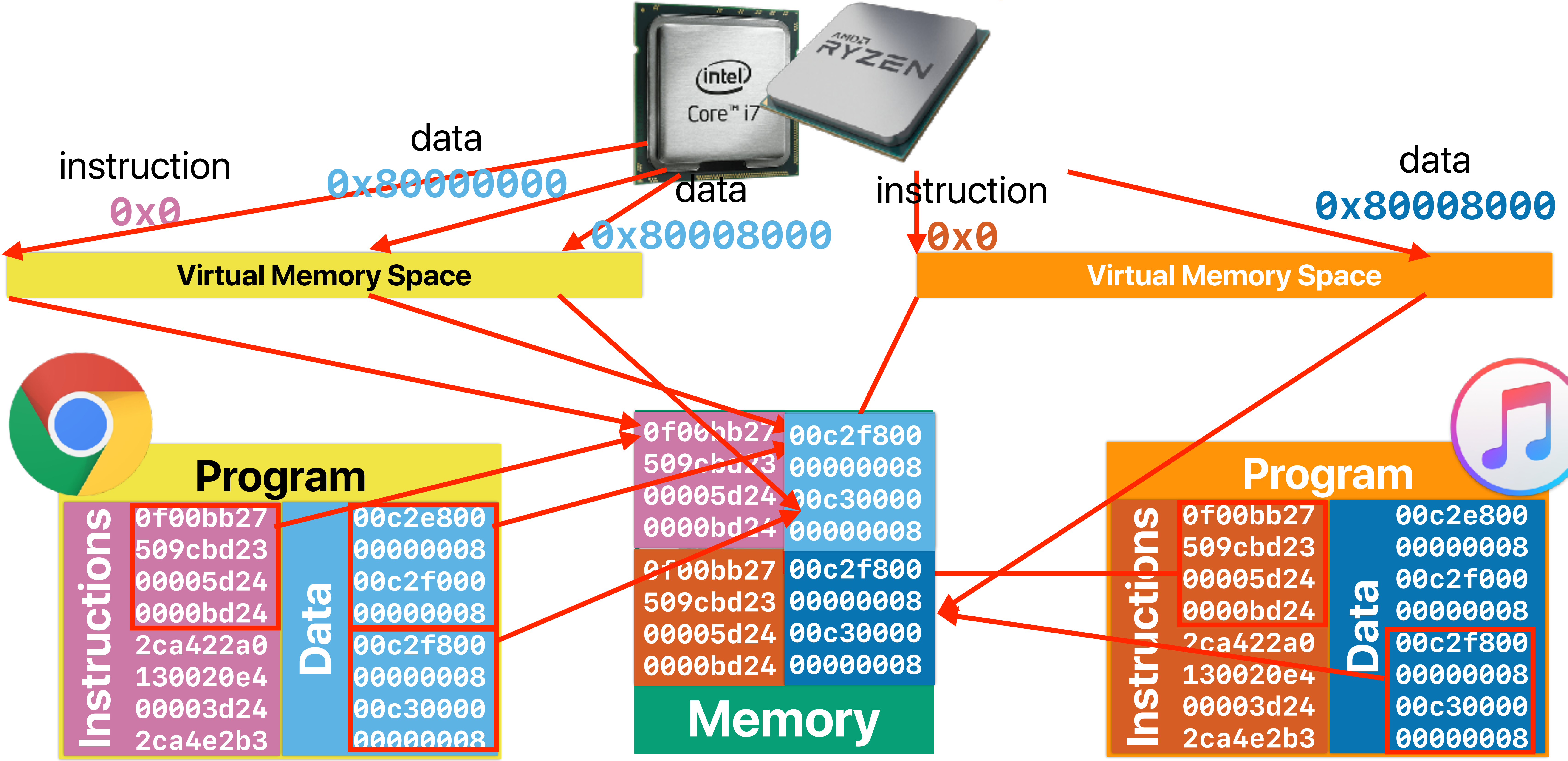
If we expose memory directly to the processor (III)

What if both programs need to use memory?

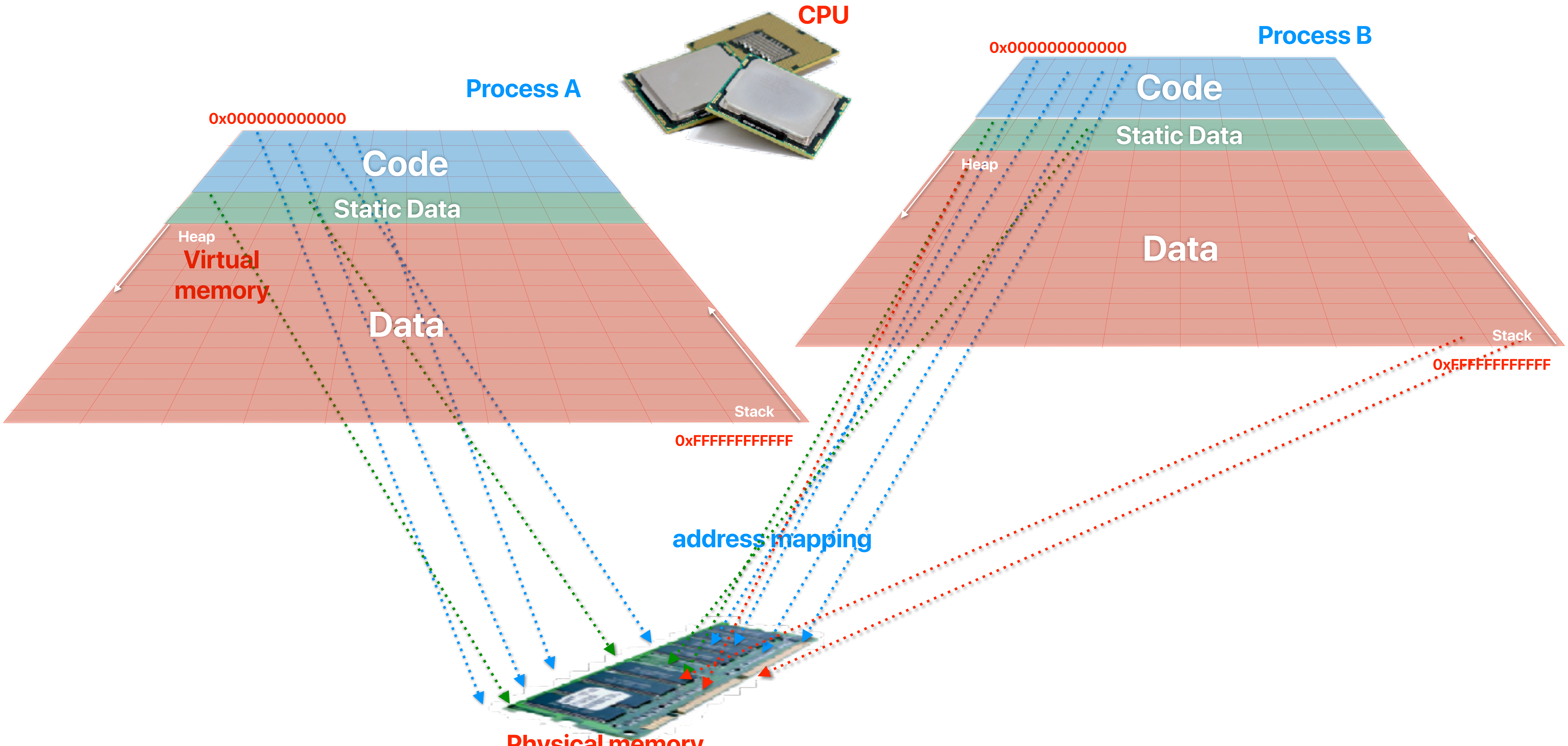


The Virtual Memory Abstraction

Virtual memory



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 revisit: Virtualization

- Some processes may use the same processor
- Each process has the same address for variable a, but different values.
- You may see the content of a compiled program using objdump

Demo: 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);
```

getcpu system call to retrieve the executing CPU ID

```
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s process_nickname\n",argv[0]);
        exit(1);
    }

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

create a random number

```
    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);
```

print the value of a and address of a

```
    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);
```

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

```
    return 0;
}
```

```
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
```

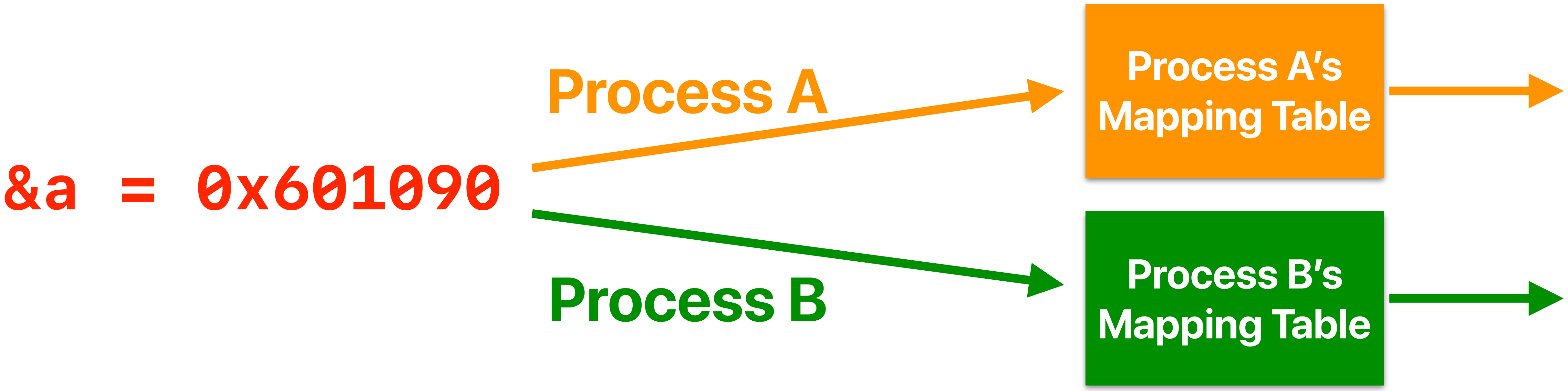
The same processor!

Different values

Different values are preserved

The same memory address!

Demo revisited

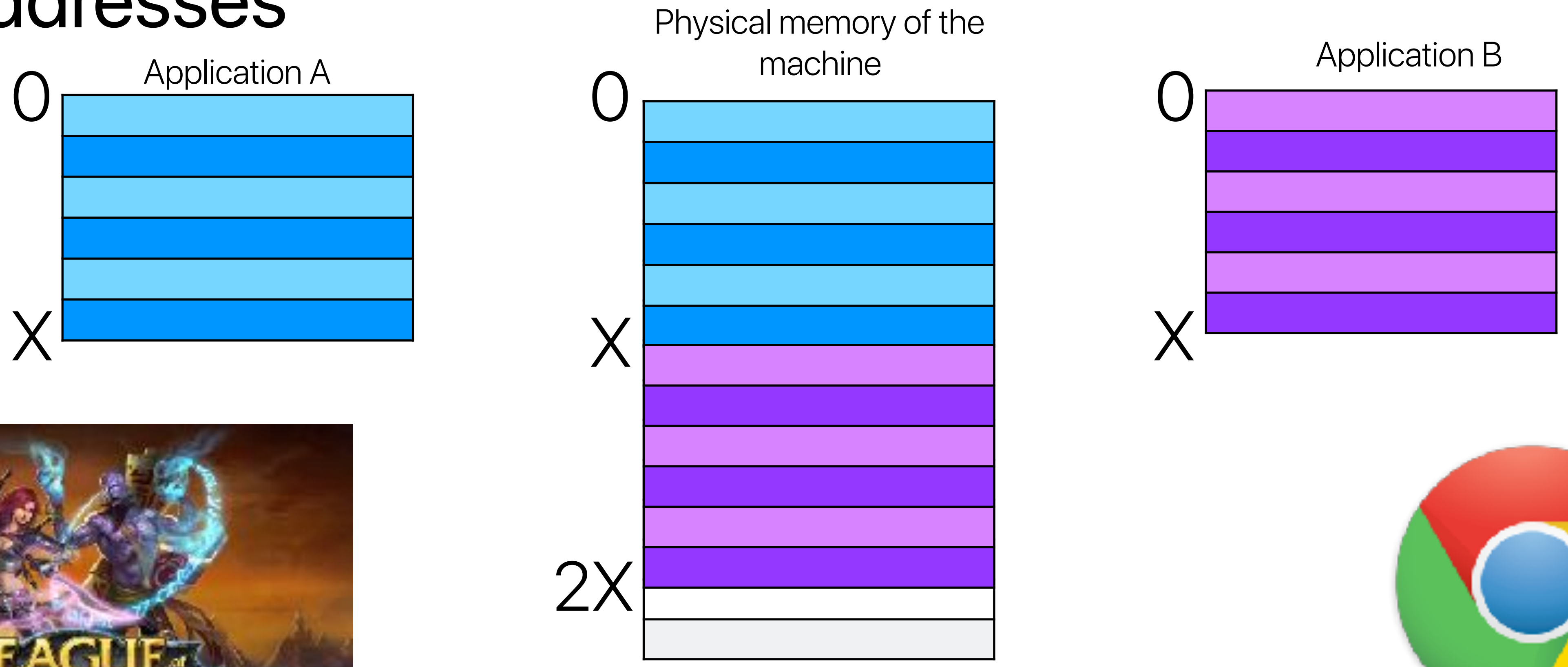



```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

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;
}
```

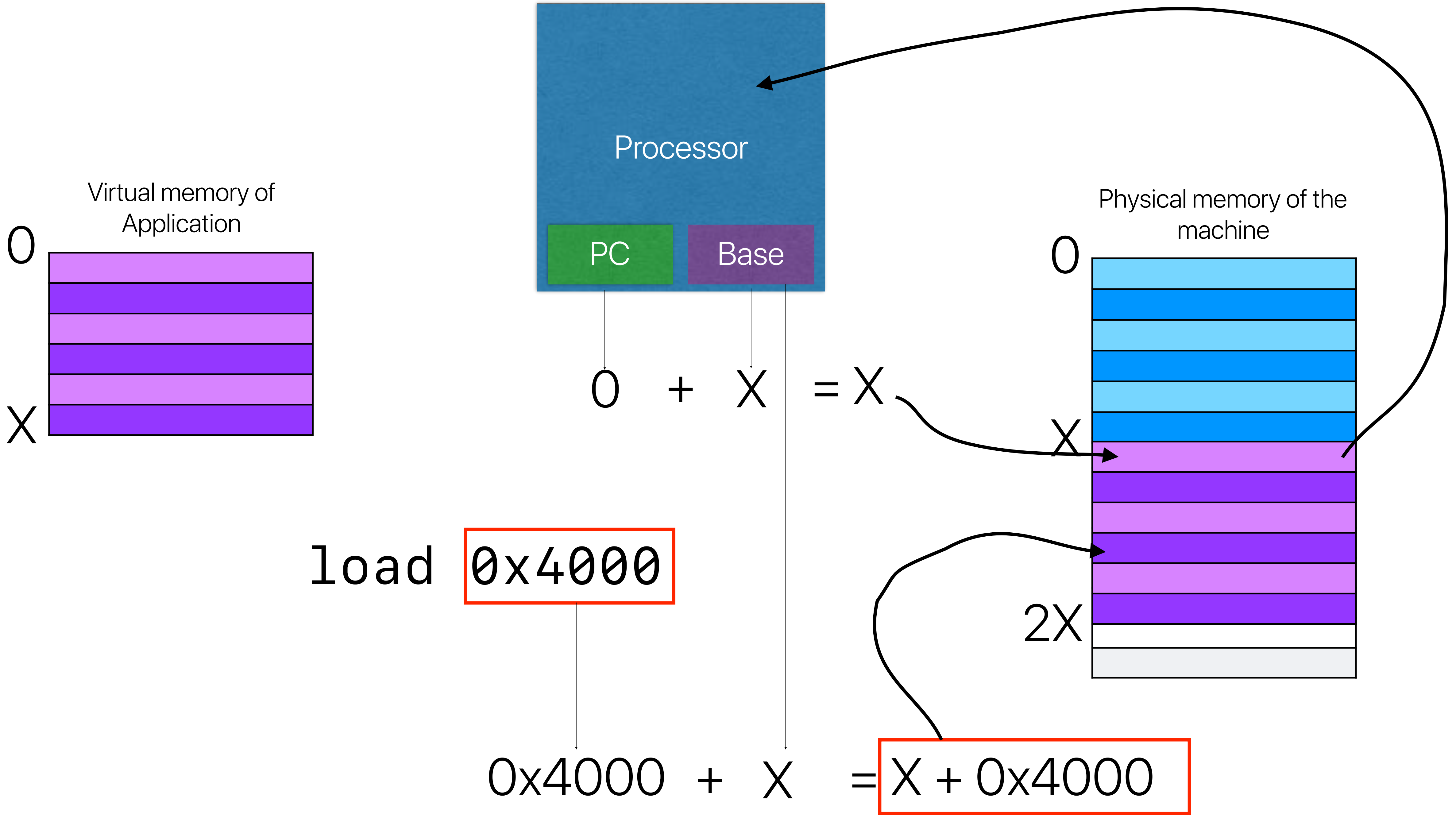
How to map from virtual to physical?
Let’s start from segmentation

Segmentation

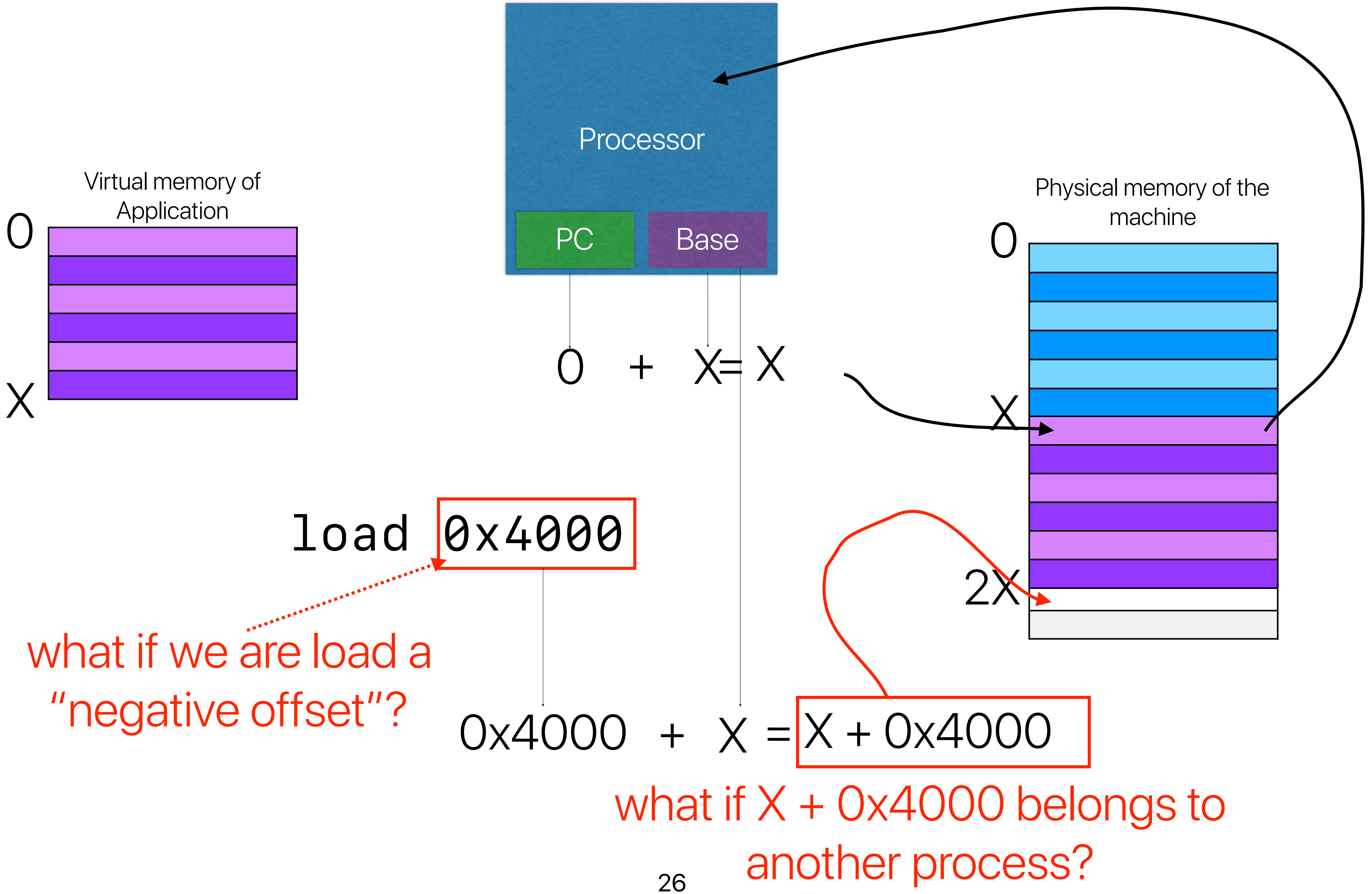
- The compiler generates code using virtual memory addresses
- The OS works together with hardware to partition physical memory space into segments for each running application
- The hardware dynamically translates virtual addresses into physical memory addresses



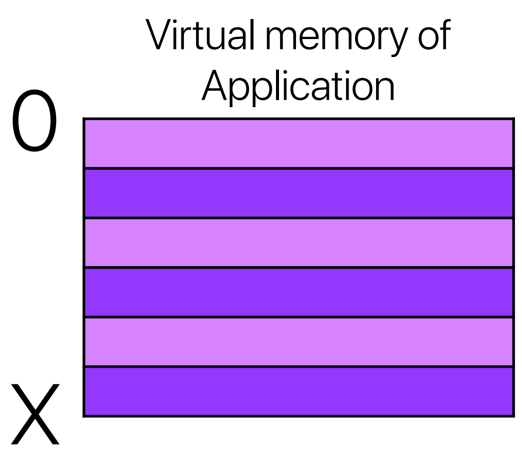
Address translation in segmentation

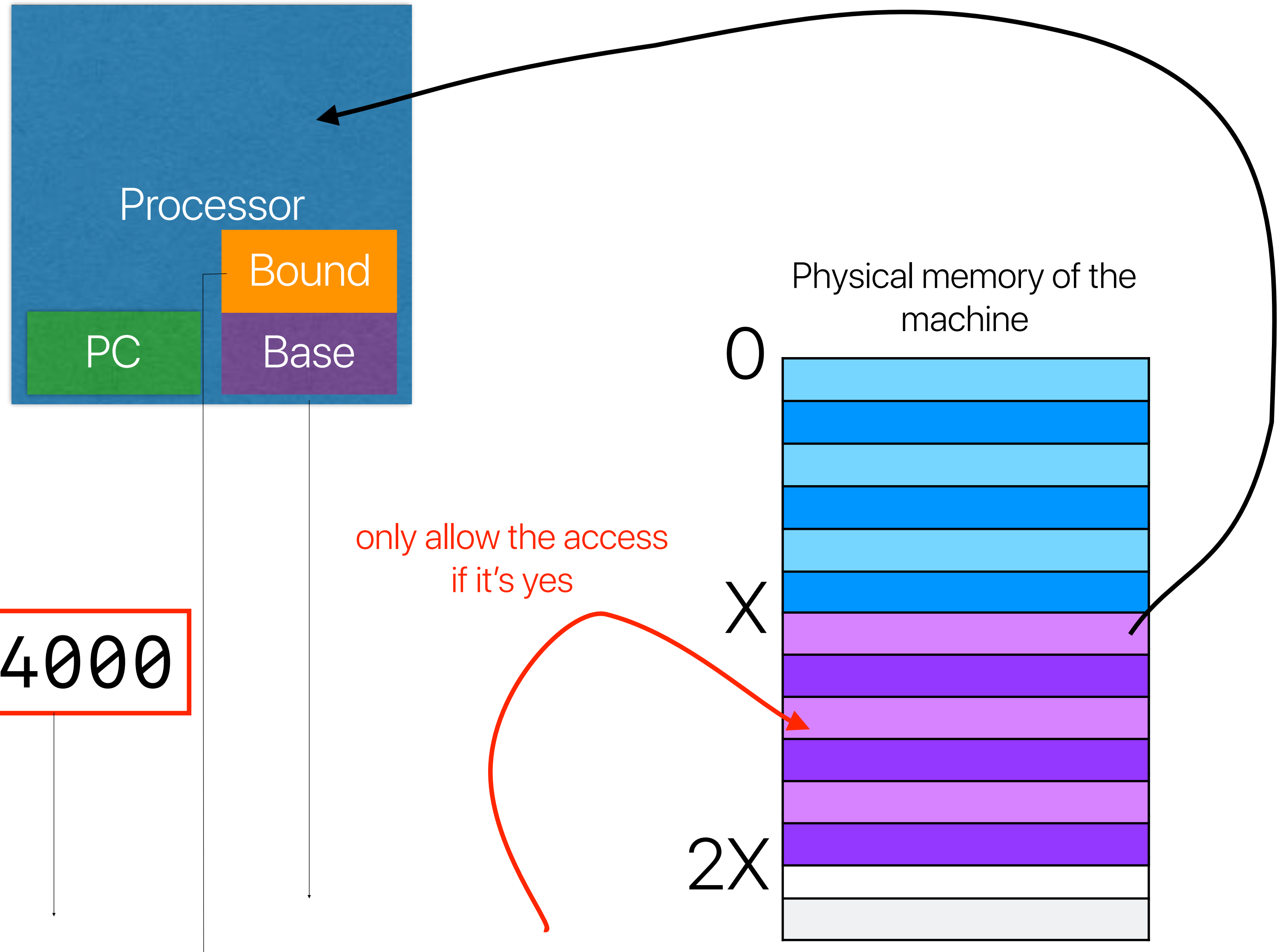


Protection again malicious processes



Protection again malicious processes





`load 0x4000`

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).

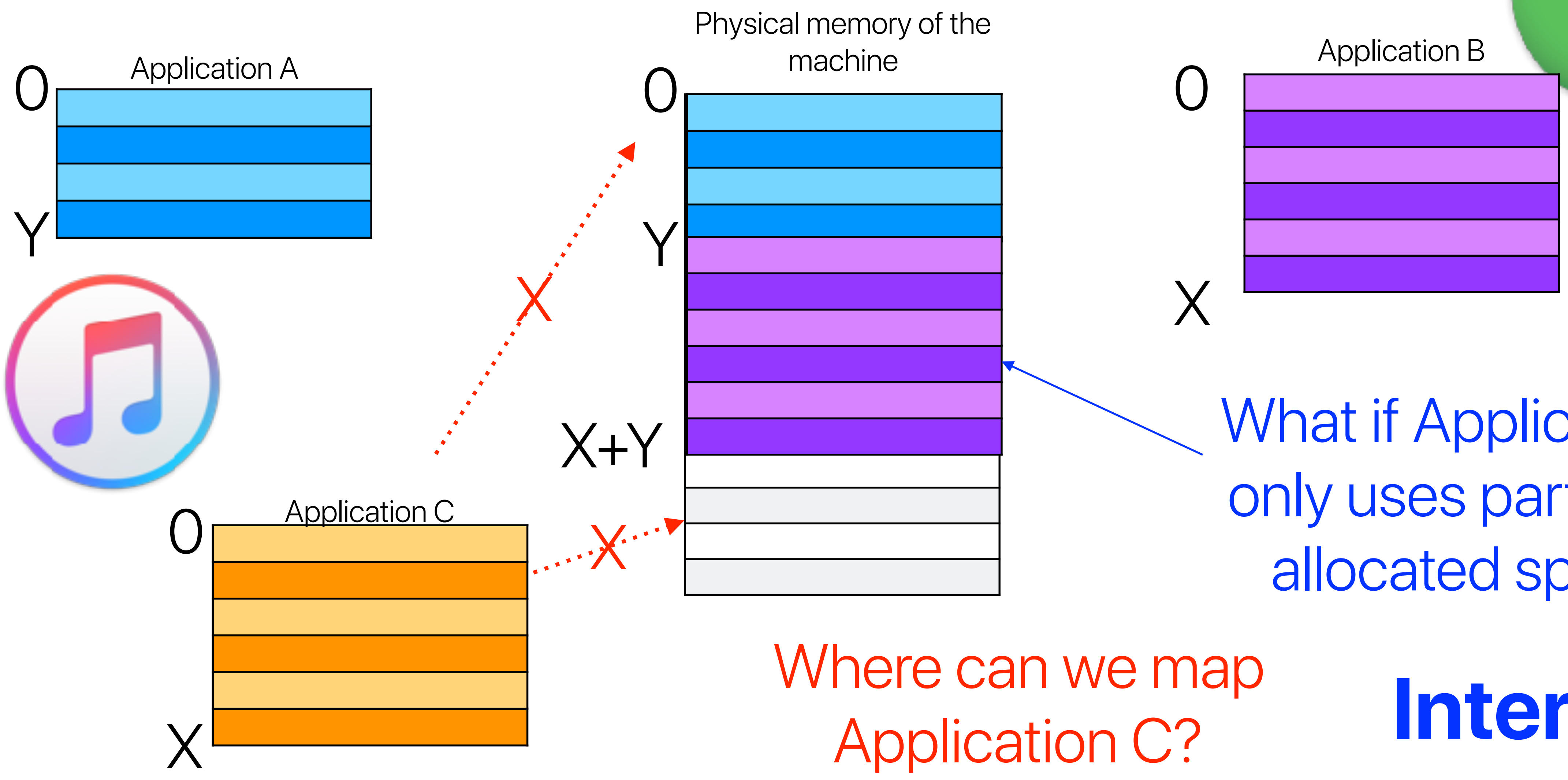
0x4000 + X = X + 0x4000

0x4000 < X

Yes — proceed

No — segmentation fault!!!

What if?



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

Where can we map Application C?

Internal Fragment

We waste some space in the allocated segment

External Fragment

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

29

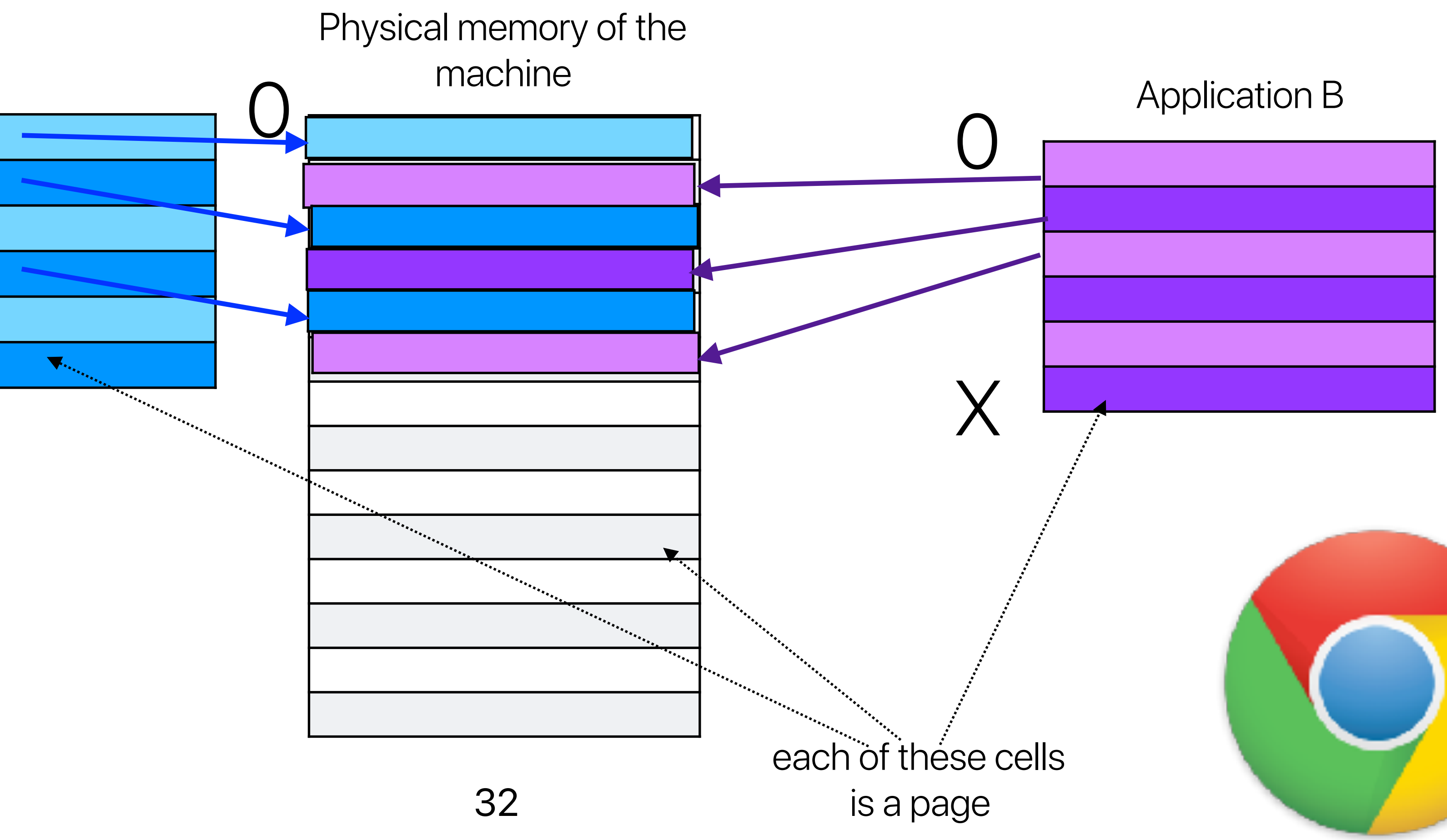
When to create a virtual to physical address mapping? — Demand paging

Demand paging

- **Paging**: partition virtual/physical memory spaces into fix-sized pages
- **Demand paging**: Allocate a physical memory page for a virtual memory page when the virtual page is needed
  - 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

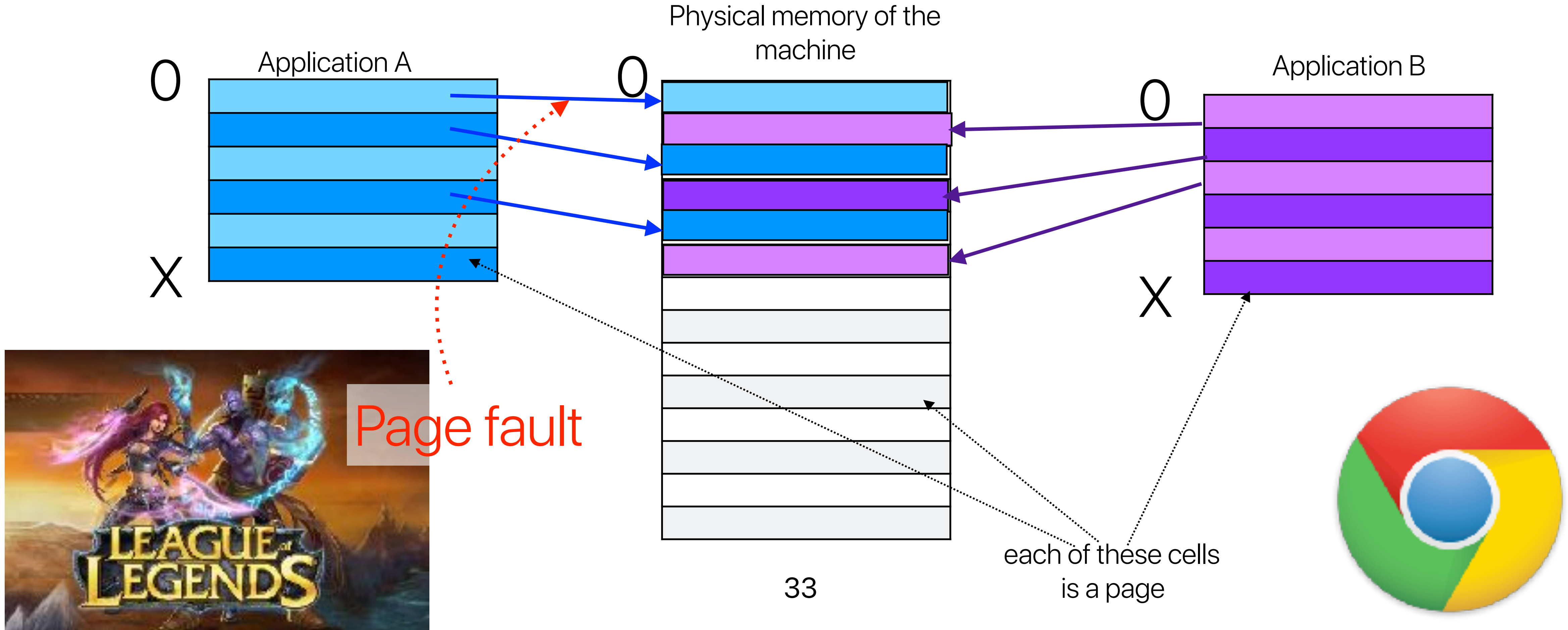
Demand paging

- **Paging**: partition virtual/physical memory spaces into fix-sized pages
- **Demand paging**: Allocate a physical memory page for a virtual memory page when the virtual page is needed



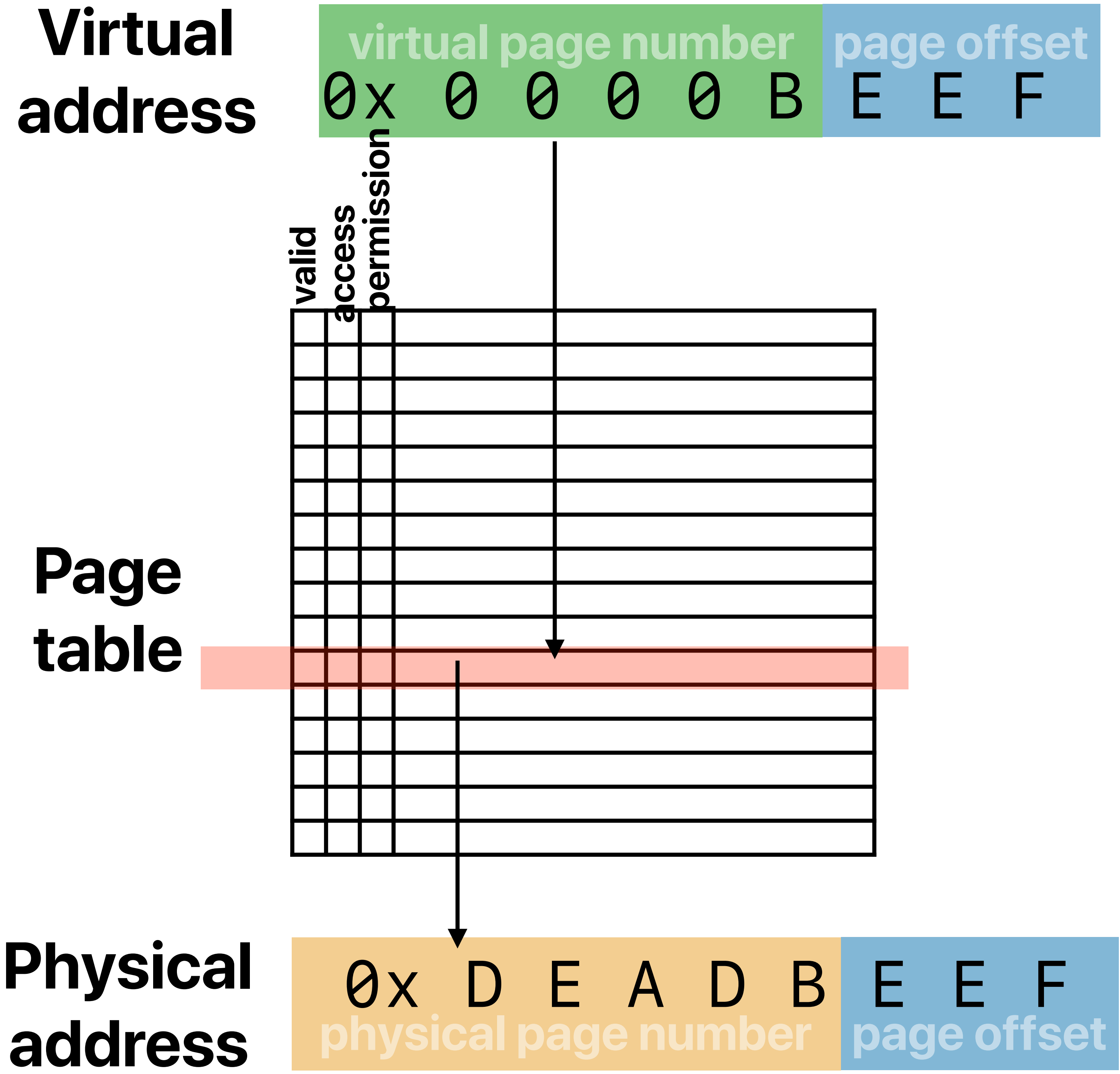
Page fault

- Page fault: if the demanding page is not in the physical memory
- How to handle page fault: the processor raises an **exception** and transfers the control (change the PC) to the page fault handler in OS code
  - Allocates a physical memory location for the page
  - Creates an entry recording this allocation — where?



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

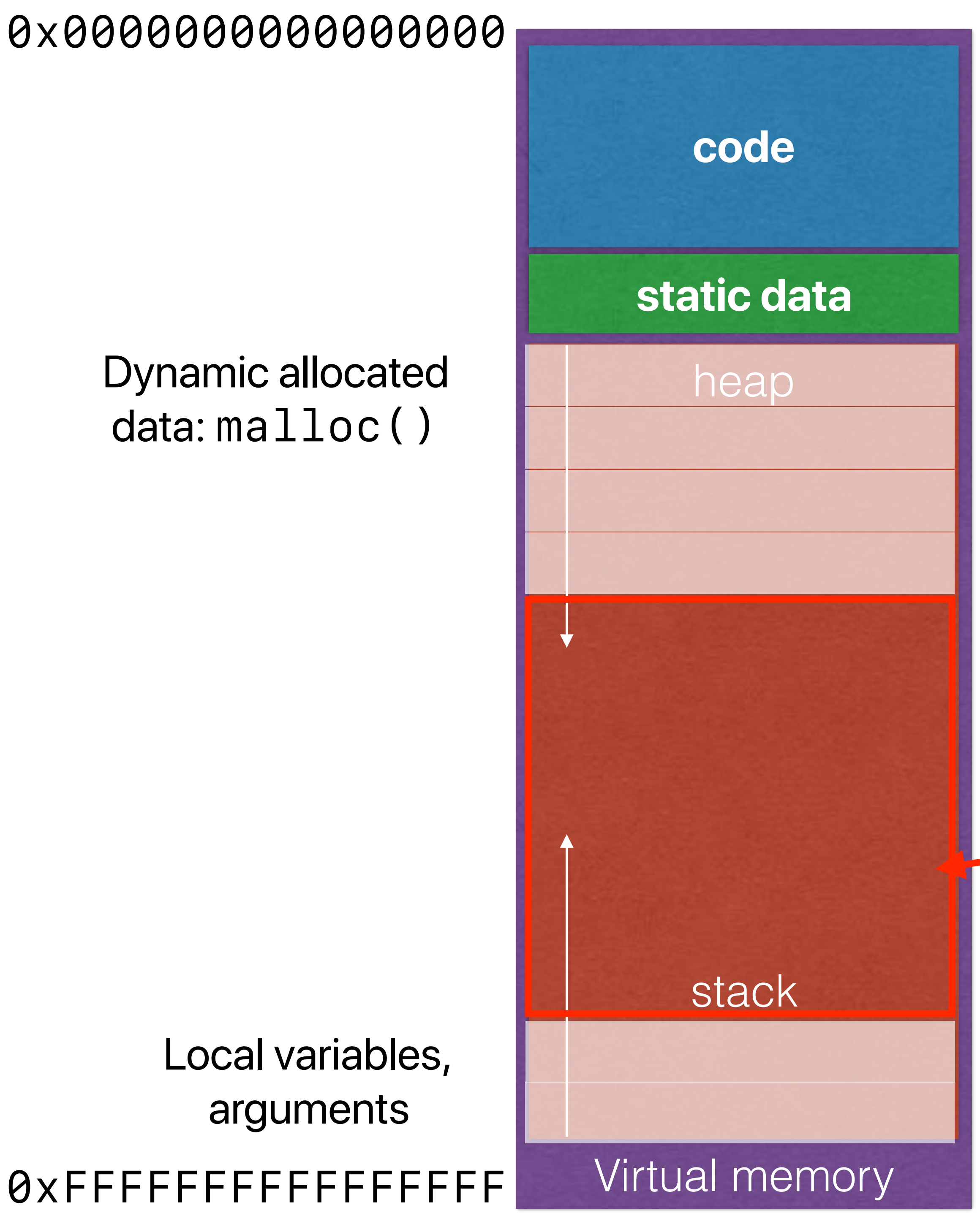


Address translation (cont.)

- Page tables are too large to be kept on the chip (millions of entries)
  — space overhead
  — memory access overhead
- Instead, the page tables are kept in memory
- 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

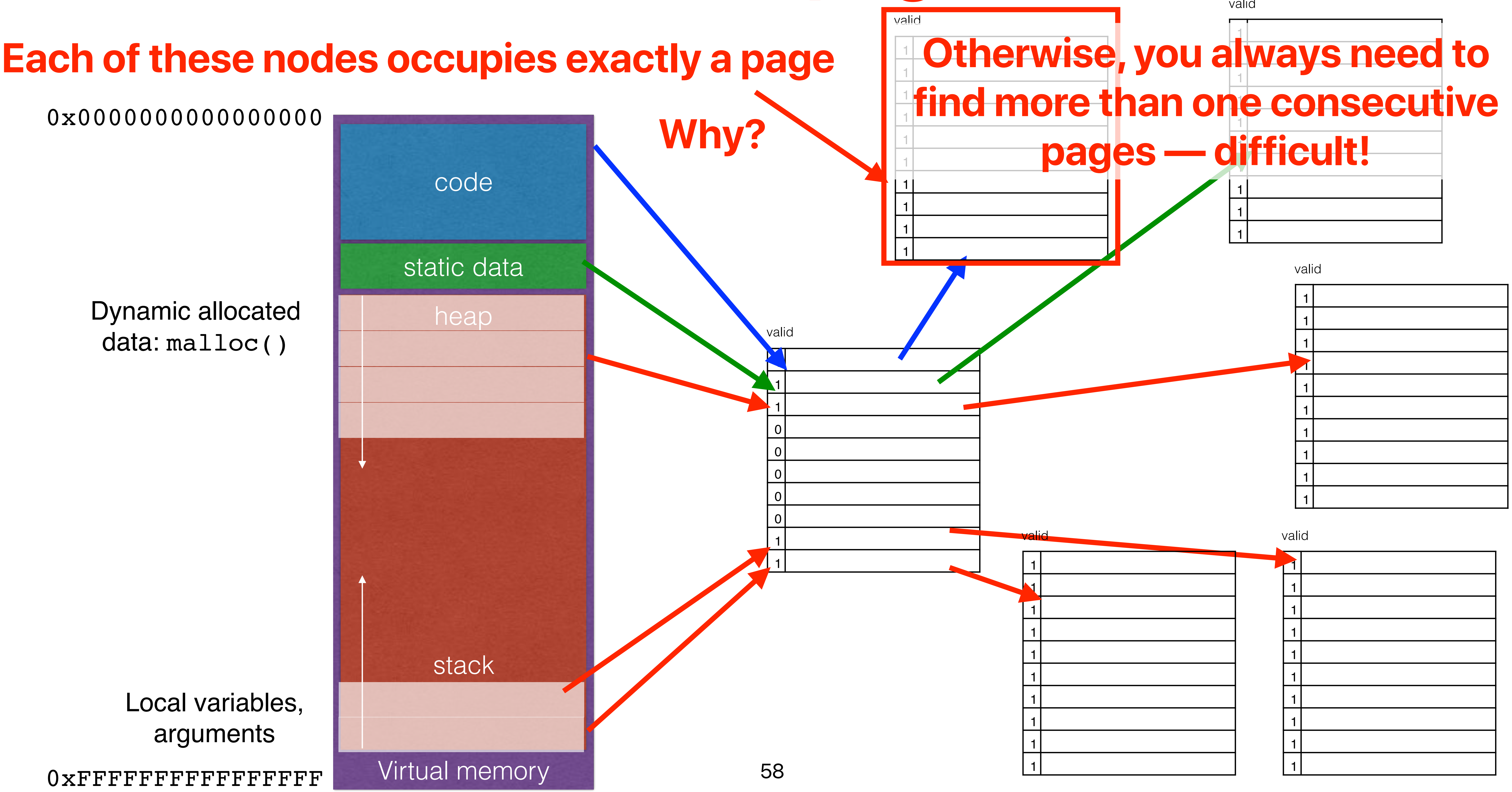
Smaller page tables

Do we really need a large table?



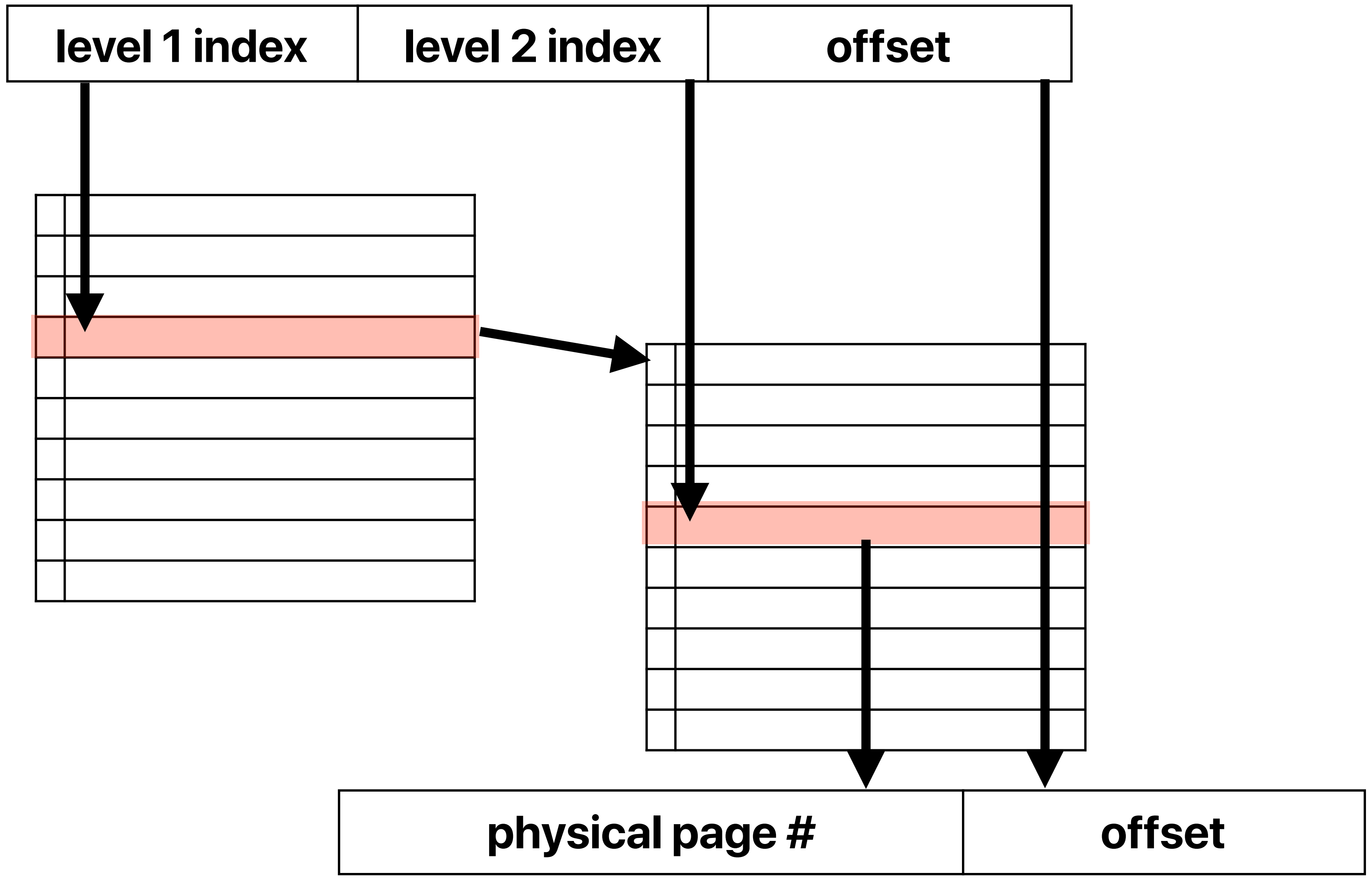
Your program probably never uses this huge area!

Hierarchical page table



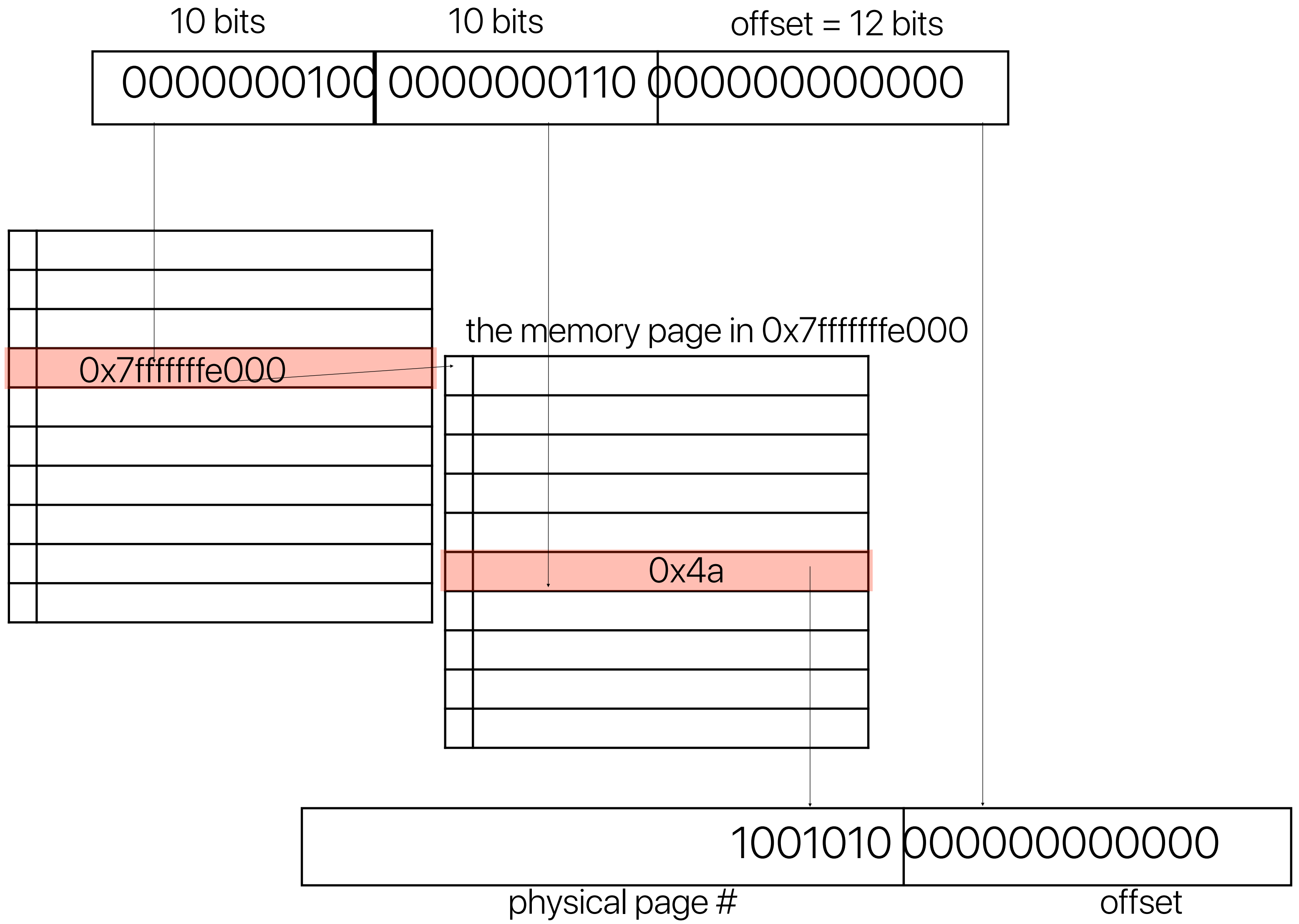
Hierarchical page table

- 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



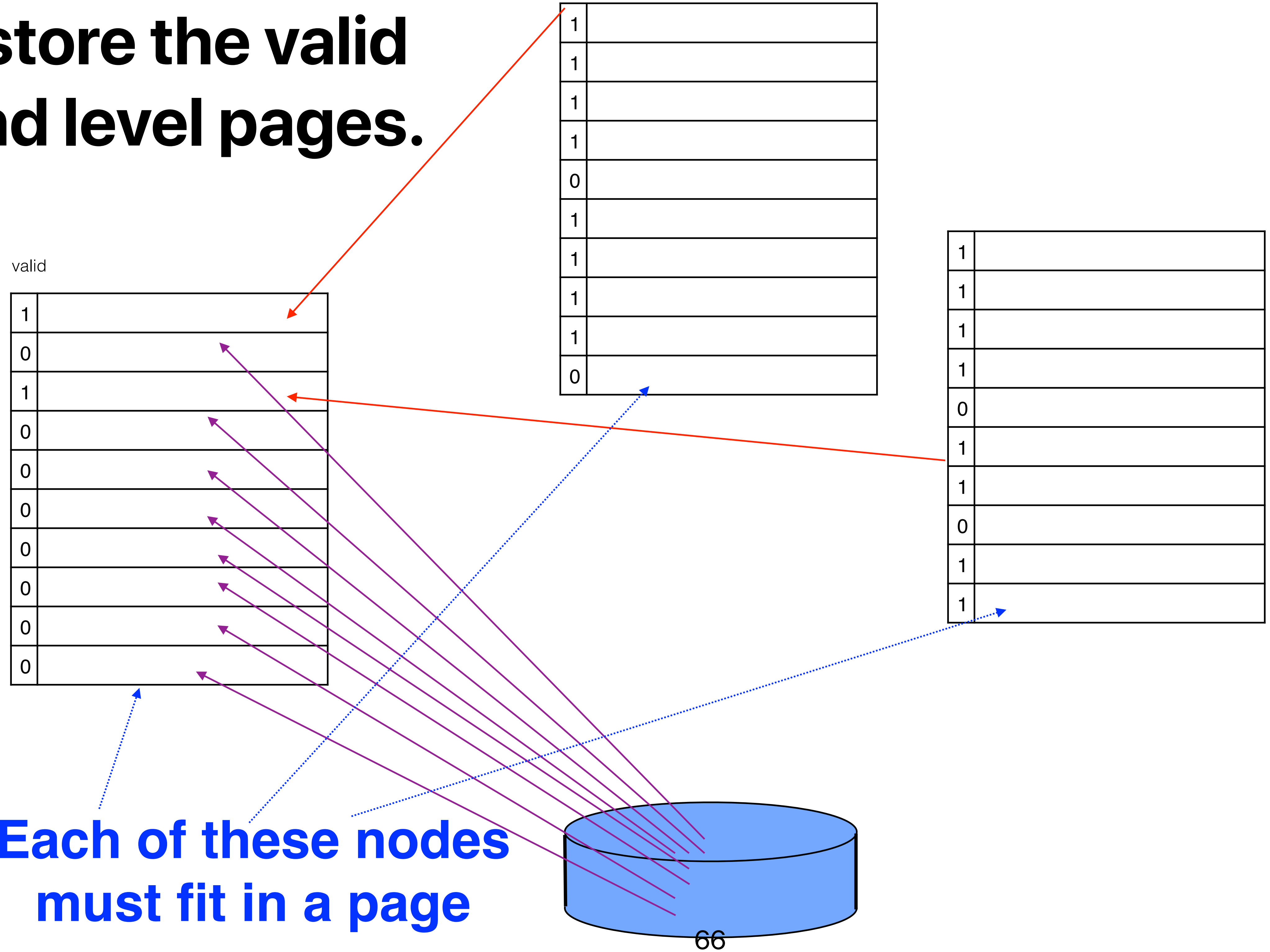
Page table walking example

- 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.



Each of these nodes must fit in a page

Case study: Address translation in x86-64