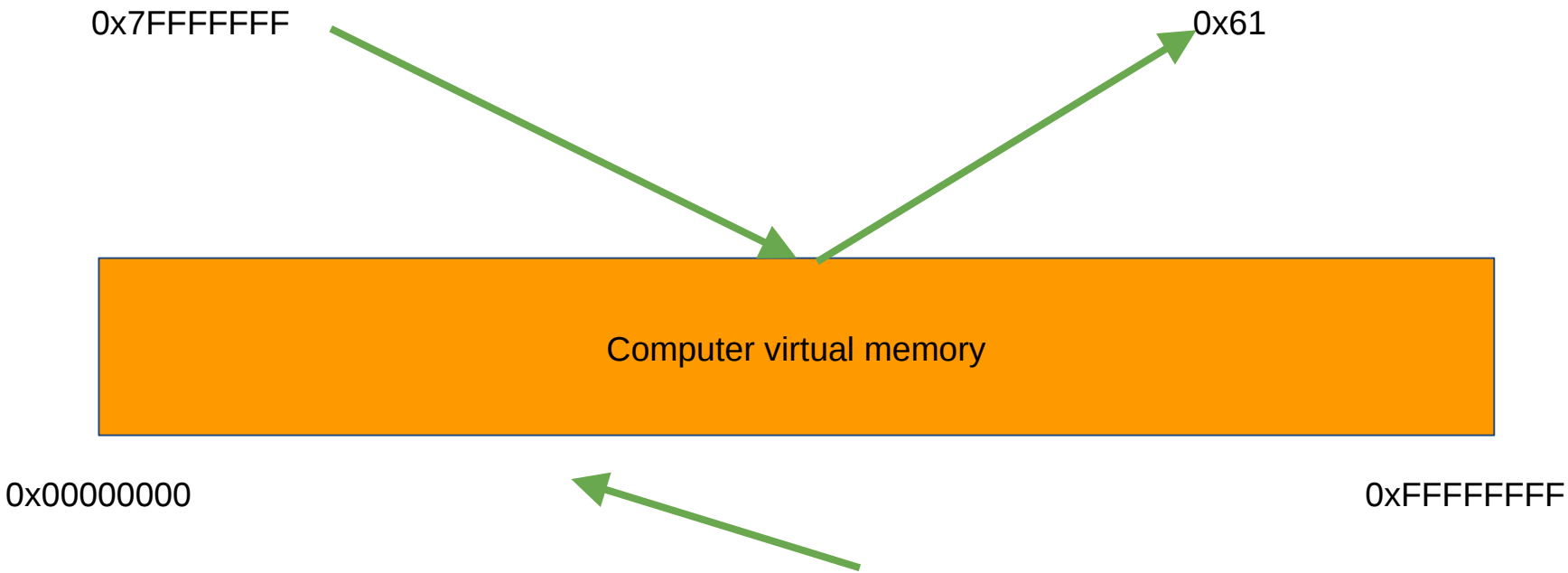
Memory review

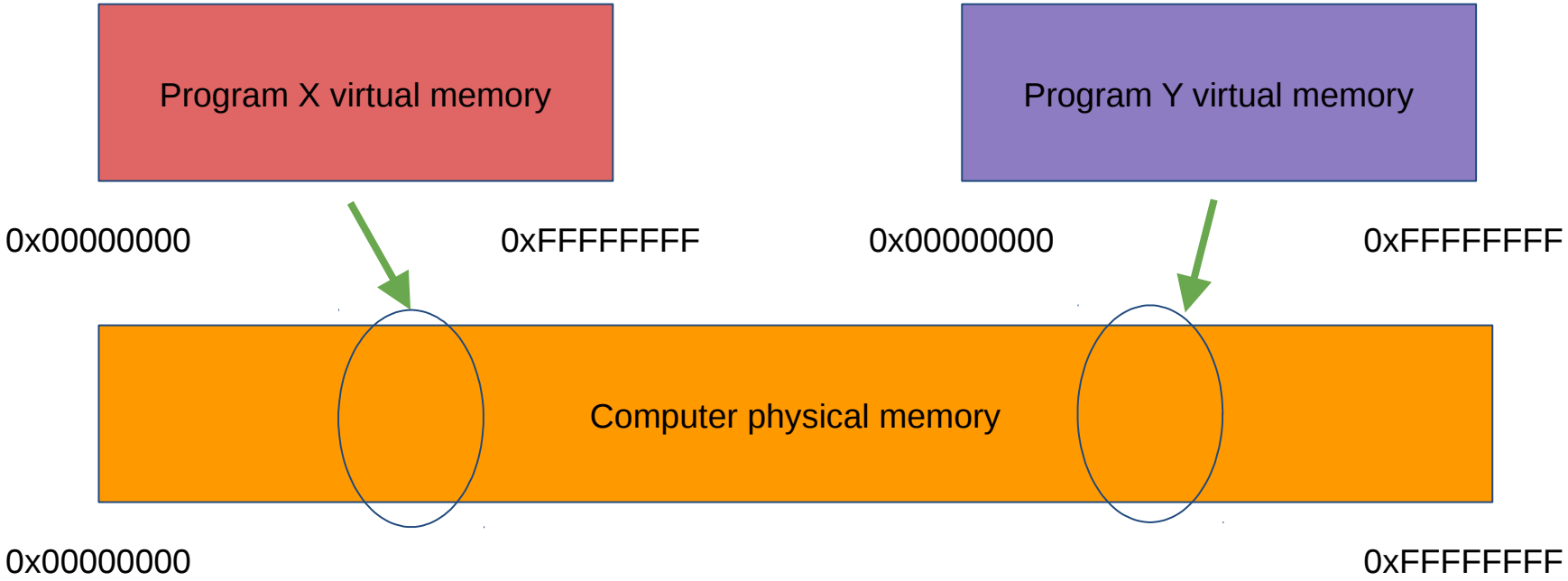
Pointers arithmetic and memory access

What's the memory composed of?



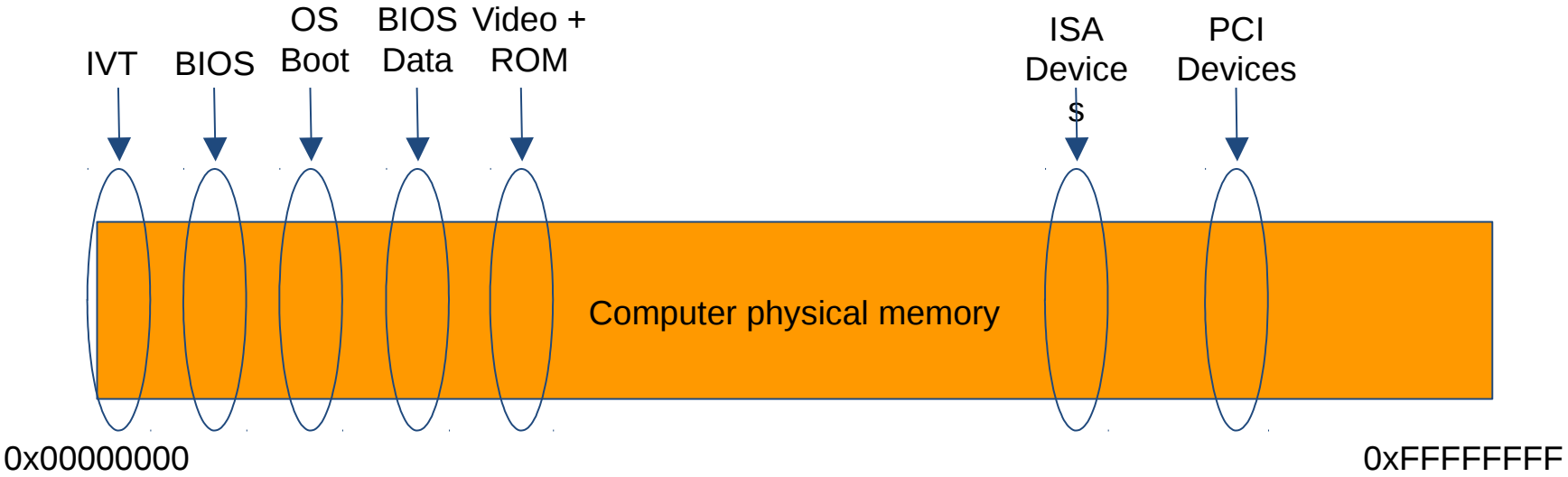
Why are we using virtual memory?

For security!



There are other reasons! Can you take a guess?

IA-32 Architecture uses memory mapped hardware!



Virtual memory is also part of the hardware support mechanism for “Swapping”

Accessing the memory

```
section .data

intArray:
        dd      0x01
        dd      0x02
        dd      0x03
        dd      0x04
        dd      0x05


global main
section .text

main:
        push ebp        ;Prolog
        mov ebp, esp

        mov eax, intArray ;Is this good?

        pop ebp         ;Epilog
        ret
```

Program contains 2 sections:

- Data
- Text

intArray is an array of 32 bits integer
dd keyword is used to “reserve” a “double word” of memory (32 bits)

An attempt to access memory is done
using:

mov eax, intArray

Is this good? Let’s try it!

Pointers

```
section .data

intArray:
        dd      0x01
        dd      0x02
        dd      0x03
        dd      0x04
        dd      0x05


global main
section .text

main:
        push ebp        ;Prolog
        mov ebp, esp

        mov eax, intArray ;Is this good?

        pop ebp         ;Epilog
        ret
```

In this example, intArray is a pointer to a region of memory. Using:

mov eax, intArray

Results in **copying** that **pointer** value **into eax**.

This does not result into copying the value stored at the memory where intArray is pointing.

intArray is therefore not “dereferenced”.

Dereferencing a pointer

```
section .data

intArray:
        dd      0x01
        dd      0x02
        dd      0x03
        dd      0x04
        dd      0x05


global main
section .text

main:
        push ebp        ;Prolog
        mov ebp, esp

        mov eax, [intArray]
        mov ebx, [intArray + 4]
        mov [intArray], ebx
        mov [intArray + 4], eax

        pop ebp         ;Epilog
        ret
```

Square brackets are used as "**dereferencing**" **operator**:

mov eax, [intArray]

Results in the first 32 bits of the array to be copied into eax.

The "**Base + displacement**" notation is also possible as in:

mov ebx, [intArray + 4]

Resulting in the second element being copied into ebx. Note that the displacement is expressed in bytes.

Assessing one byte at a time

```
section .data

intArray:
        dd      0x01
        dd      0x02
        dd      0x03
        dd      0x04
        dd      0x05


global main
section .text

main:
        push ebp        ;Prolog
        mov ebp, esp

        mov al, [intArray]
        mov bl, [intArray + 1]
        mov cl, [intArray + 2]
        mov dl, [intArray + 3]

        pop ebp         ;Epilog
        ret
```

Using an 8 bit register will force the memory access to the correct width.

The same is true with a 16 bits register.

Let's run this specific example together....

IA-32 stores values in LITTLE ENDIAN format

```
section .data

littleEndian:
        dd      0x00000000


global main
section .text

main:
        push ebp            ;Prolog
        mov ebp, esp

        mov eax, 0xAABBCCDD
        mov [littleEndian], eax

        pop ebp             ;Epilog
        ret
```

In **register** values are stored in **BIG ENDIAN**. For example, the integer 255 (0x000000FF) will be stored as is in a register.

However, when copying such a value into **memory**, the byte order will be **reversed** and stored as **LITTLE ENDIAN**. 255 will effectively be stored as 0xFF000000 once copied into memory!

You need to watch out for this! (Mostly when debugging) Values crossing 4 bytes boundaries will be seriously “distorted” by this...

After mov into littleEndian variable

```
(gdb) x /10b &littleEndian
0x804a018:      0xdd    0xcc    0xbb    0xaa    0x00    0x00    0x00    0x00
0x804a020:      0x00    0x00
```

LEA is often used to load values in registers. The square brackets, used in LEA instruction does not result in dereferencing a pointer. You can use this to load values based on arithmetics!

Every good rule has exceptions...

Beware LEA!
mov eax, [intArray] != lea eax, [intArray]

In order to validate this information, let’s disassemble a simple program and reimplement it using assembly.