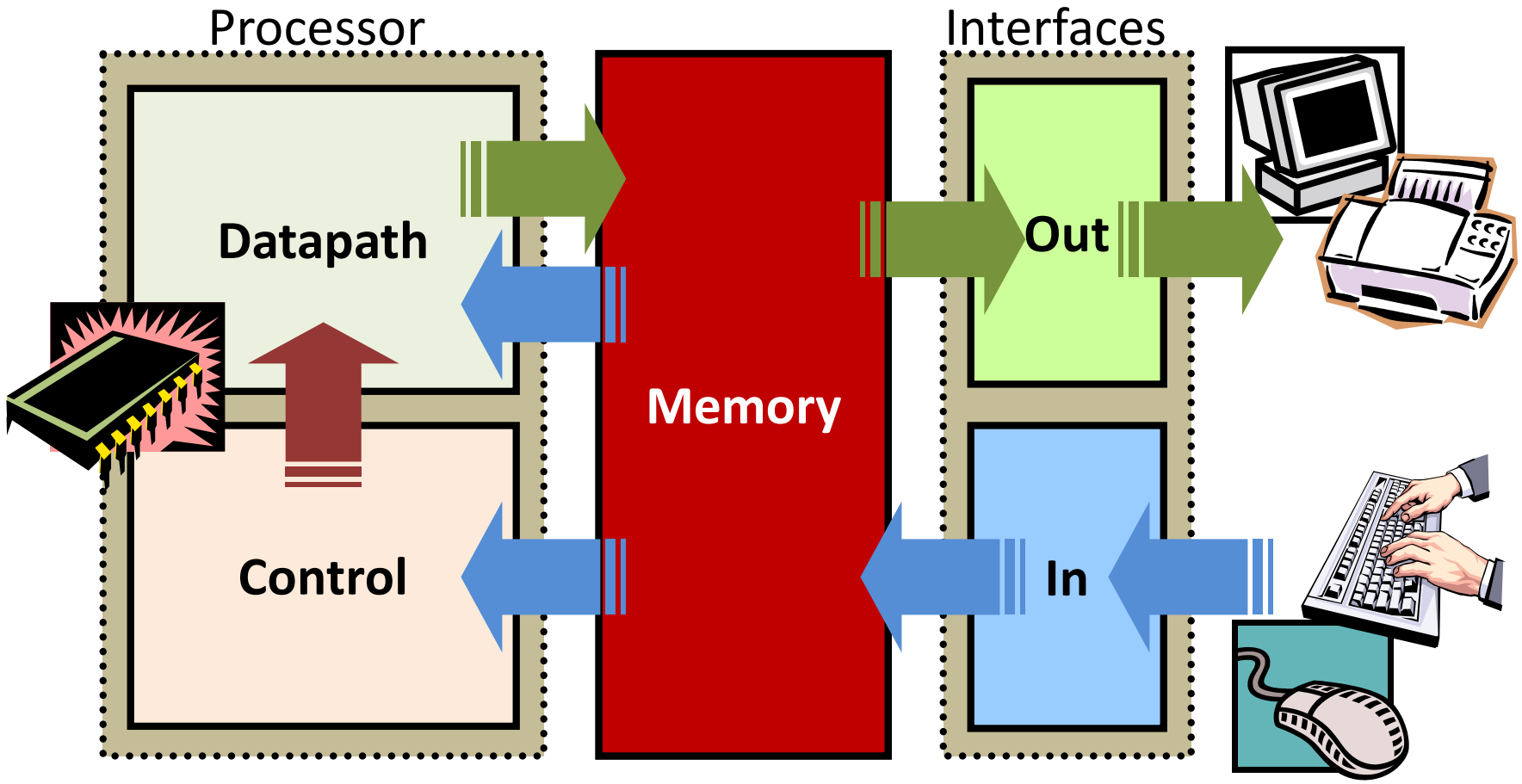
CS-200
Computer Architecture
—
Part 3c. Memory Hierarchy
Virtual Memory

Paolo Ienne
<paolo.ienne@epfl.ch>

The Five Classic Components of a Computer



Segmentation Fault? Bus Error?

```
[18]icvm0100:~/tmp> cat code.c
#include <stdlib.h>
#include <stdio.h>
int main() {
  int *p = (int *) 1234;    /* li t0, 1234    */
  printf("%i",*p);}         /* la a0, format  */
                            /* lw a1, 0(t0)   */
                            /* jal printf     */

[19]icvm0100:~/tmp> gcc code.c -o code

[20]icvm0100:~/tmp> code
Segmentation fault (core dumped)

[21]icvm0100:~/tmp>
```

The OS has detected our access to address 1234 and, because it is **not "our" memory**, it has **blocked** it!

But **HOW**?!

Overview

Three problems to solve:

- How to **“protect” memory** so that each program *(processes)* running “simultaneously” in the system can only access itw own data? How can we isolate processes?
- What happens if the main **memory** (DRAM) is **not sufficient** for the execution of a program? Can we use our disk? How?
- How do we run several programs *(processes)* “simultaneously”? How do we load **multiple programs in memory**? Where?

Needs of Multiprogrammed System

- **Relocation**
  - All programs must be written without knowledge of where they will be in memory
- **Protection**
  - Programs can access only their own data
- **Space**
  - If several programs run at the same time, memory shortage will be even more a problem

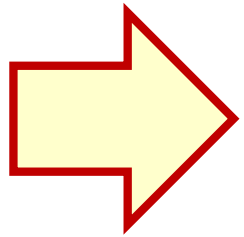
Simple Solution: Relocation at Load Time

```
                0x0000: add     v0, zero, zero          # v0 = 0
0x1234                  add     t0, zero, zero          # t0 = 0
                0x0008: sltu    t2, t0, a1              # t2 = (t0 < a1)
0x123c                  beq     t2, zero, 0x003c        # if (!t2) goto fin   0x1270
                        lw      t3, 0(a0)               # t3 = mem[a0]
                        addi    t4, zero, 32            # t4 = 32
                0x0014: beq     t4, zero, 0x0030        # if (!t4) goto next  0x1264
0x1248                  andi    t1, t3, 1               # t1 = t3 & 1
                        add     v0, v0, t1              # v0 = v0 + t1
                        srl     t3, t3, 1               # t3 = t3 >> 1
                        subi    t4, t4, 1               # t4 = t4 - 1
                        j       0x0014                  # goto inner  0x1248
                0x0030: addi    t0, t0, 1               # t0 = t0 + 1
0x1264                  addi    a0, a0, 4               # a0 = a0 + 4
                        j       0x0008                  # goto outer  0x123c
                0x003c: ret                             # return to caller
0x1270
```

Simple Solution: Relocation at Load Time

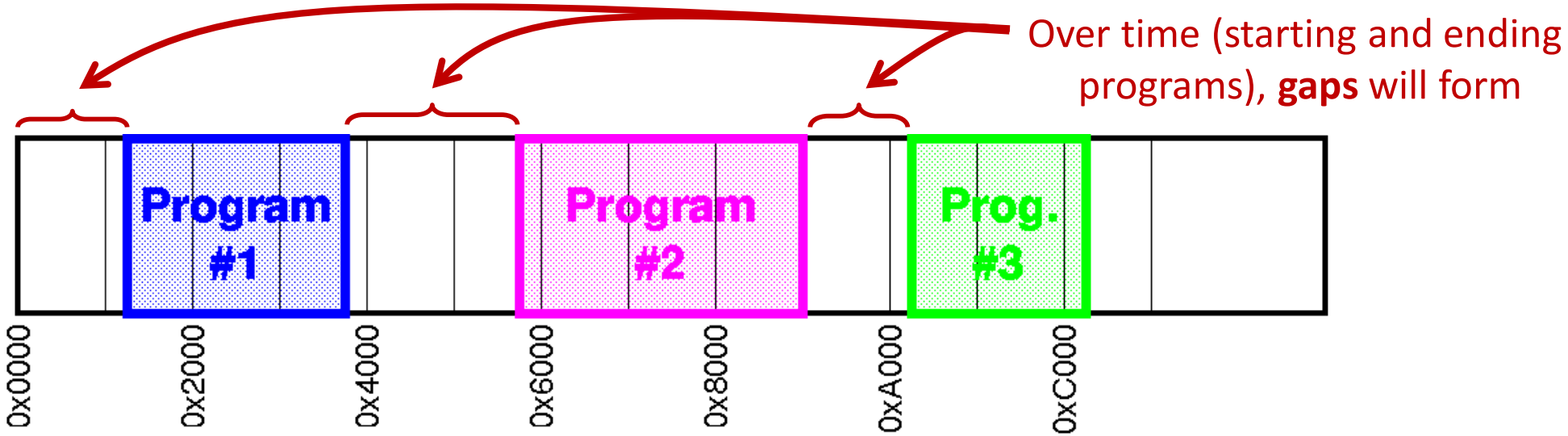
- Relocation must actually take place at **binary** level, not **assembly code**
- We need **relocation tables** to know **where** are the addresses to change

```
0x0000: 00 00 10 20
        00 00 40 20
        01 05 50 2B
        10 0A 00 0B
        8C 8B 00 00
        20 0C 00 20
        10 0C 00 05
        31 69 00 01
        00 49 10 20
        00 0B 58 42
        21 8C FF FF
        08 00 00 06
        21 08 00 01
        20 84 00 04
        08 00 00 02
        03 E0 00 08
```



```
0x1234: 00 00 10 20
        00 00 40 20
        01 05 50 2B
        10 0A 00 0B
        8C 8B 00 00
        20 0C 00 20
        10 0C 00 05
        31 69 00 01
        00 49 10 20
        00 0B 58 42
        21 8C FF FF
        08 00 04 8F
        21 08 00 01
        20 84 00 04
        08 00 04 92
        03 E0 00 08
```

Simple Solution: Relocation at Load Time



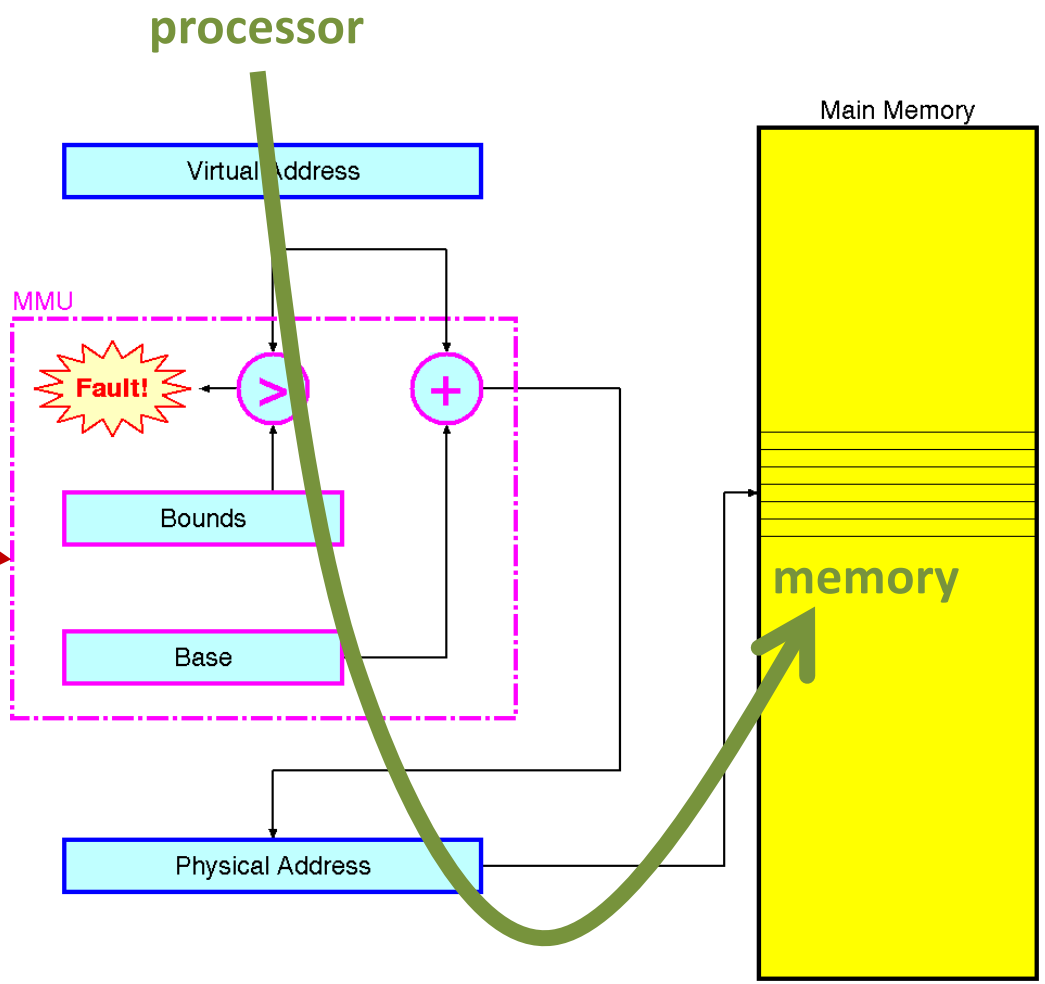
Program #4

- **Limitations:**
  - Large amount of work to do at load time
  - **Inflexible** → cannot be changed later!

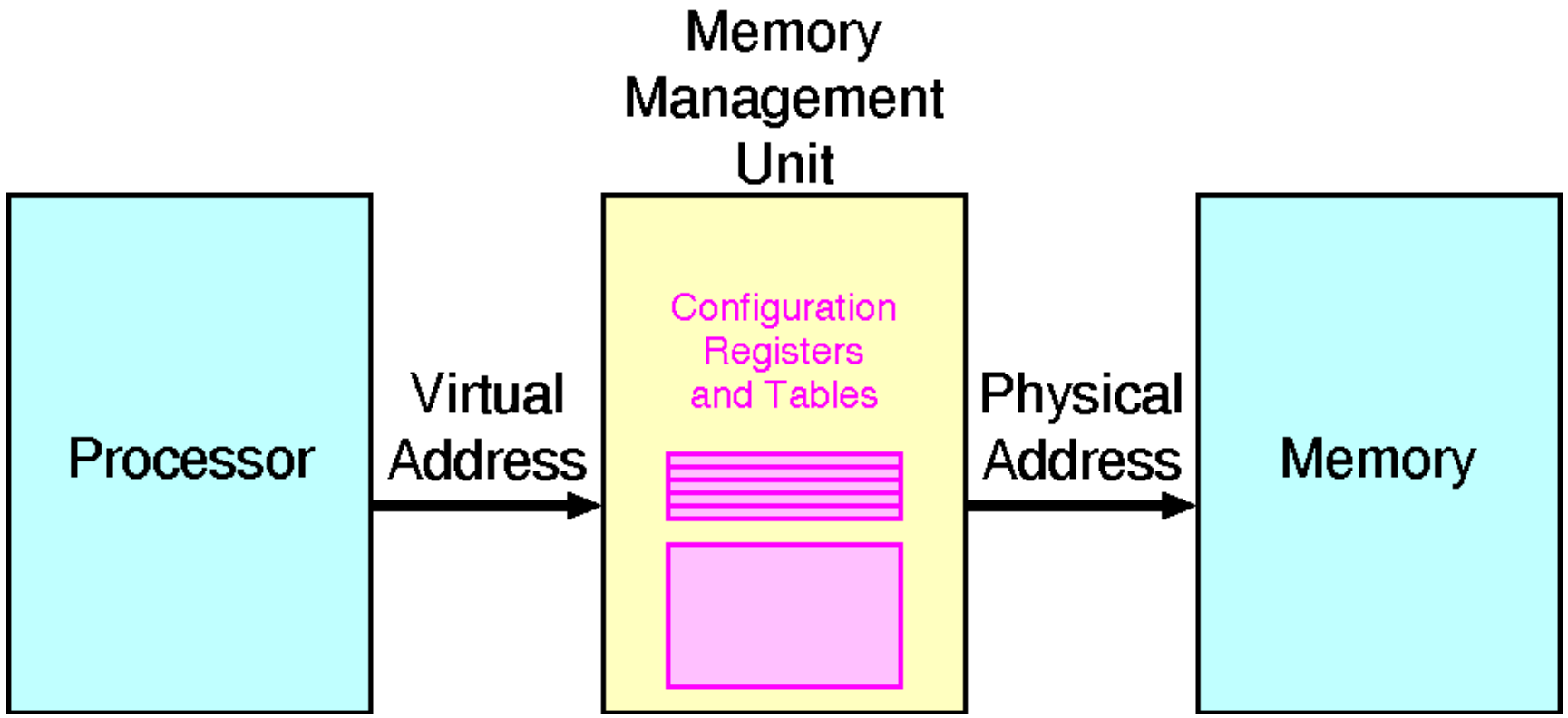
How can I load a program **bigger than the individual gaps**?!
→ garbage collection…

Relocation in Hardware: *Base and Bounds* MMU

A **Memory Management Unit** can perform this relocation **dynamically**



Memory Management Unit (MMU)



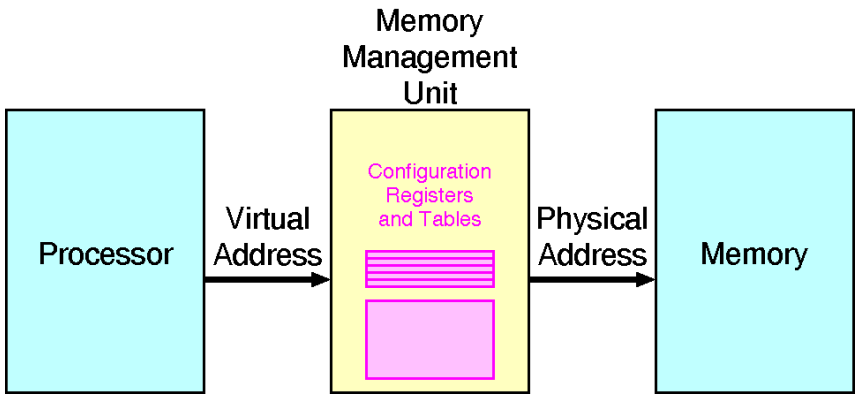
Virtual and Physical Memory

Virtual Memory: memory that the OS allows a program to believe it has

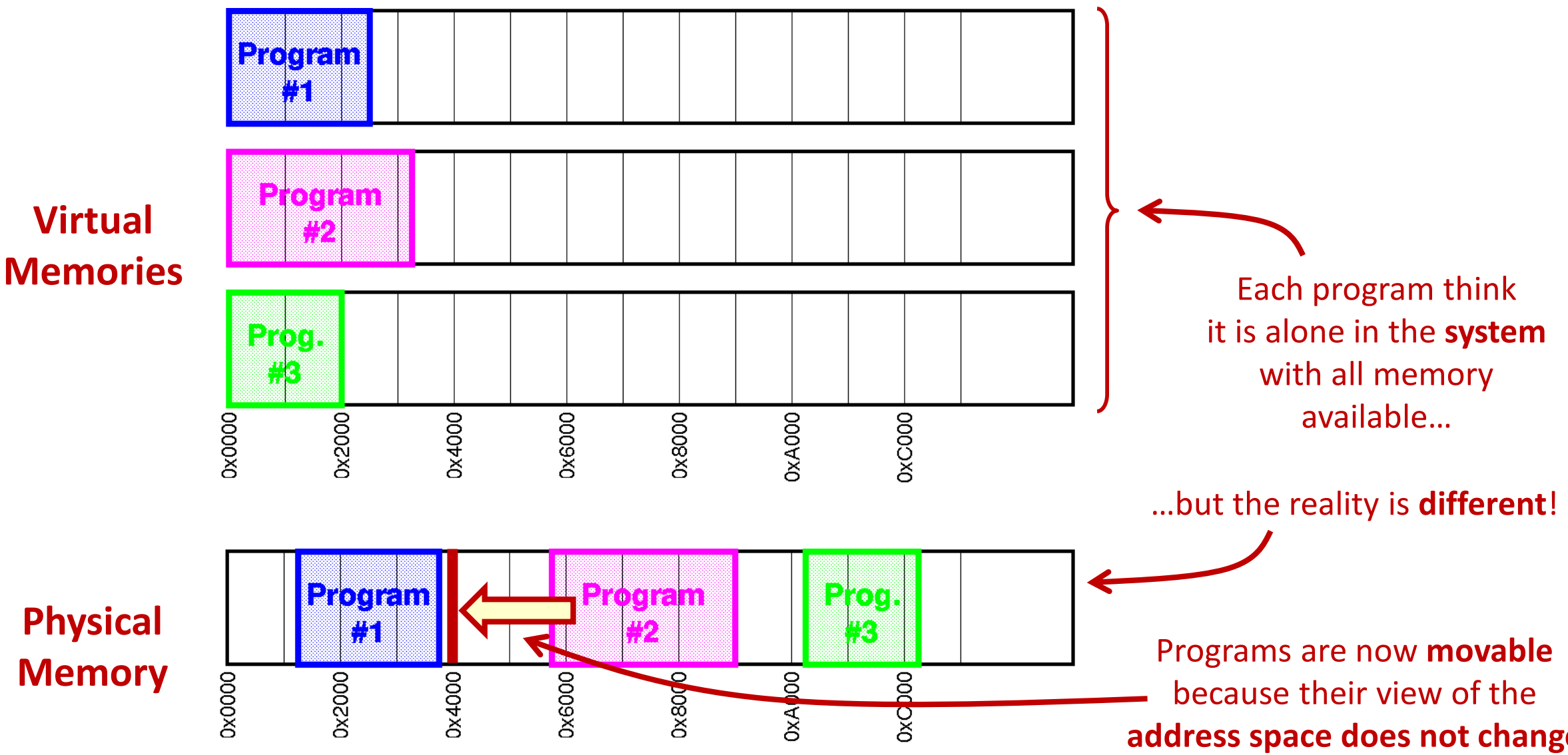
Physical Memory: memory actually available in the computer

Virtual Address: conventional address used by a program → the MMU must translate it into a physical address at the time of an access

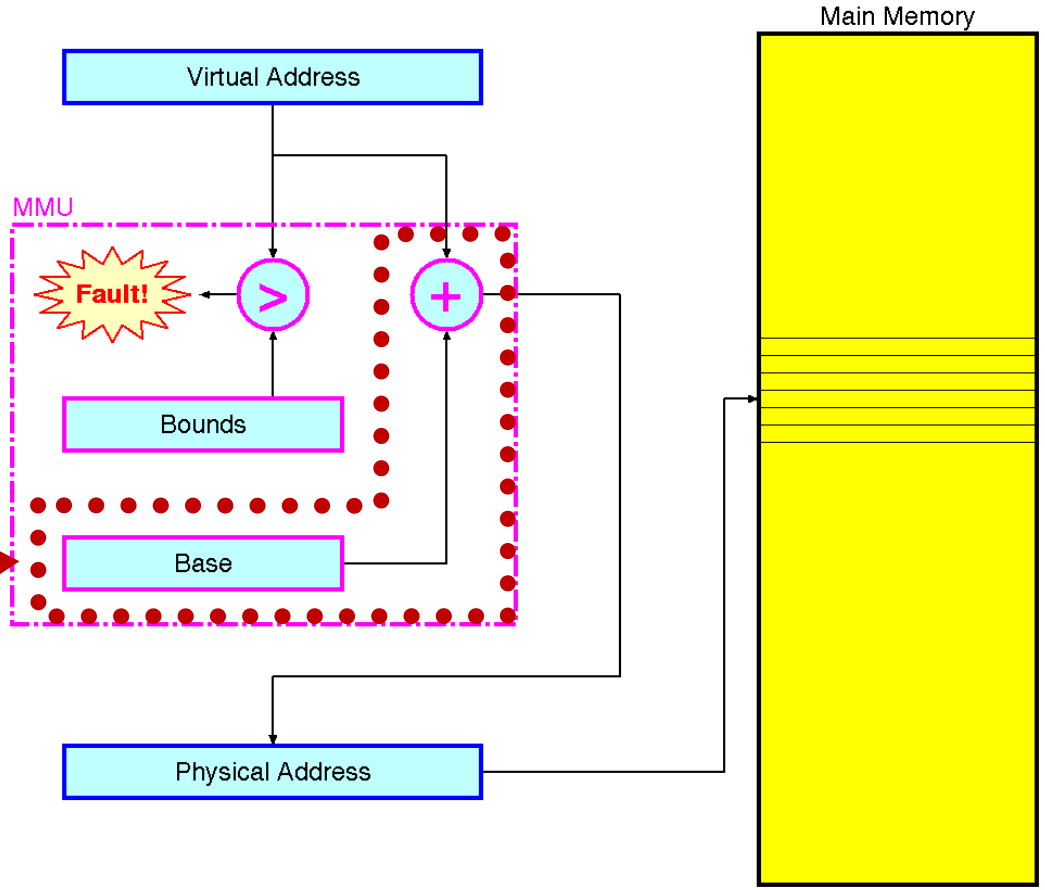
Physical Address: real location in physical memory; identifies actual storage



Virtual and Physical Memory



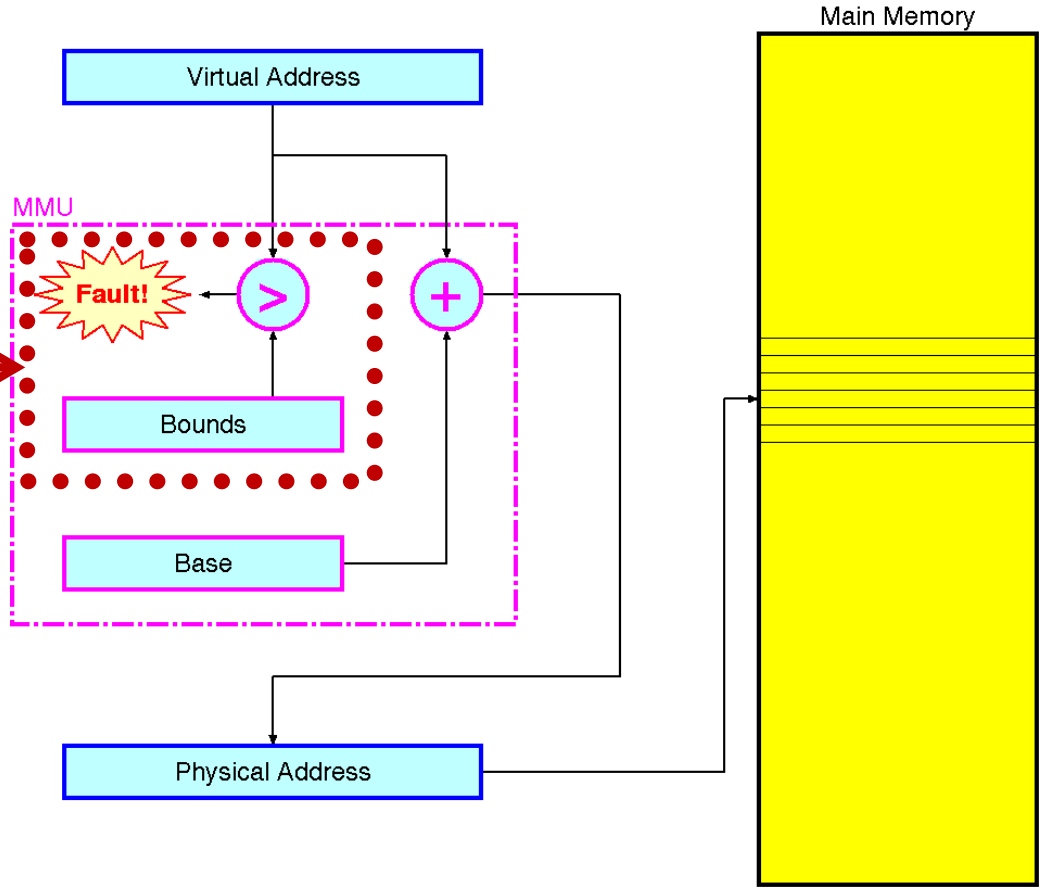
Relocation in Hardware: *Base and Bounds* MMU



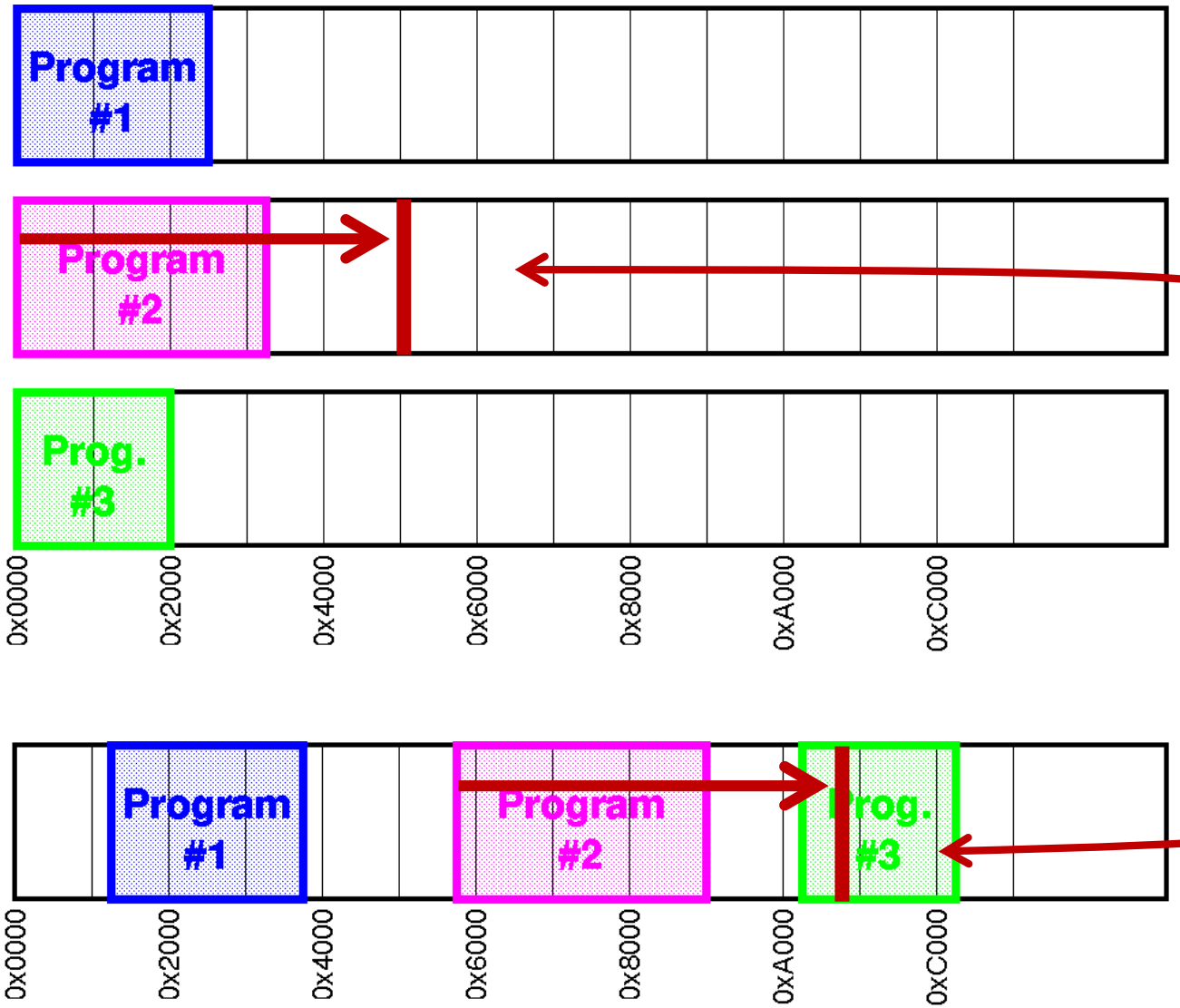
This part of the circuit is doing nothing else than what we were doing when relocating at load time: **add an offset**

Relocation in Hardware: *Base and Bounds* MMU

This part of the circuit is checking if the program is asking for an address that is **too big…**



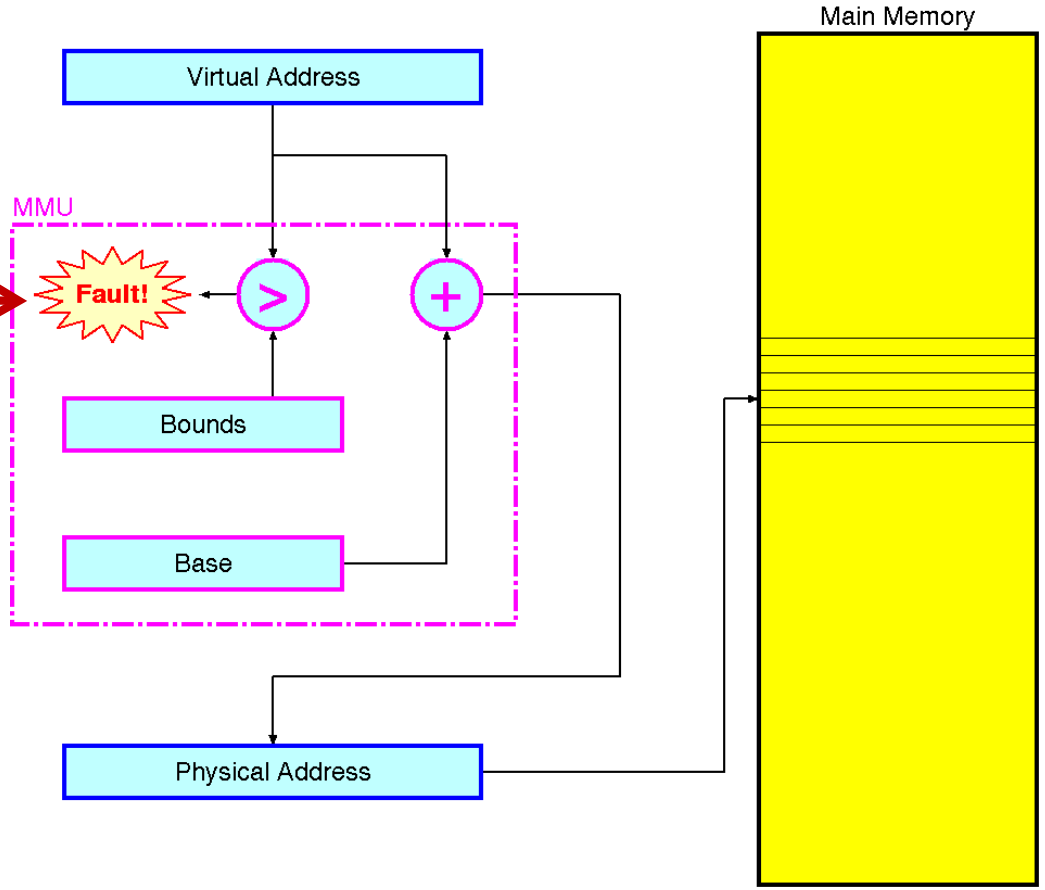
Virtual and Physical Memory



If Program #2 were allowed to make an **access beyond its occupied space**…

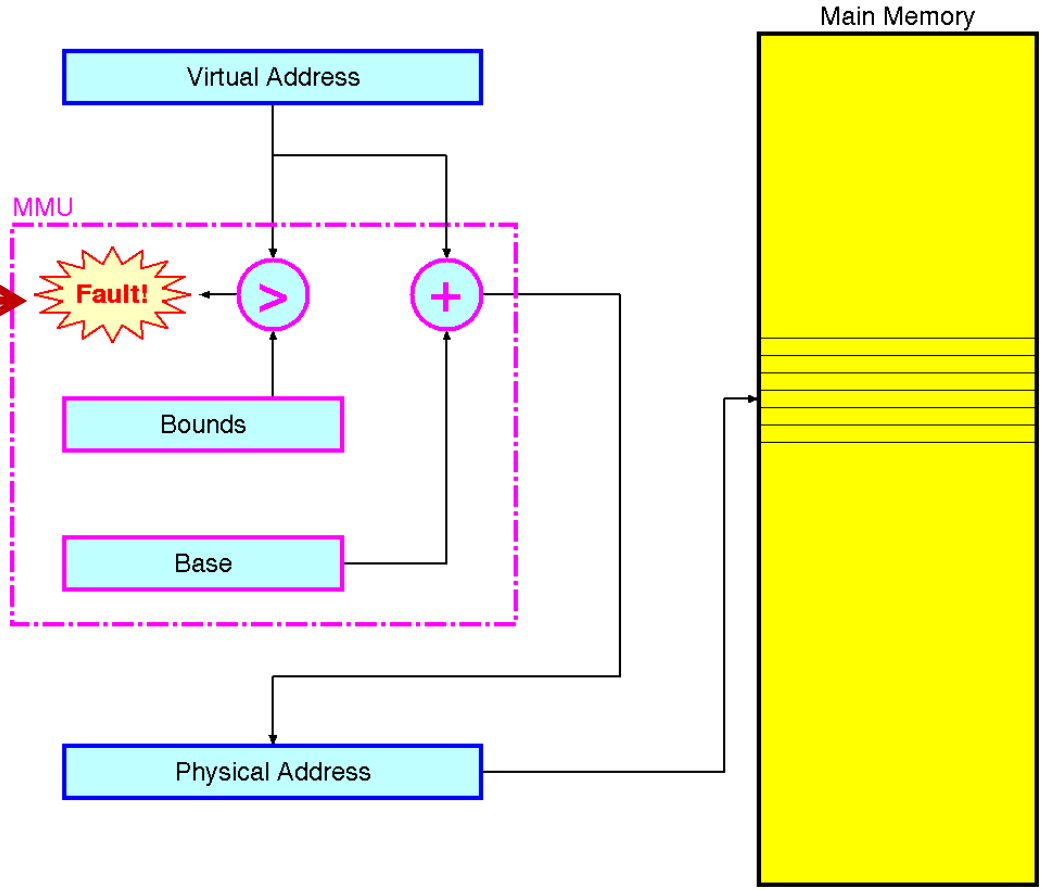
…it would potentially access **data or instructions of other programs**!

Relocation in Hardware: *Base and Bounds* MMU



If the program does that, we raise an **exception** and most likely **the OS will kill the program**

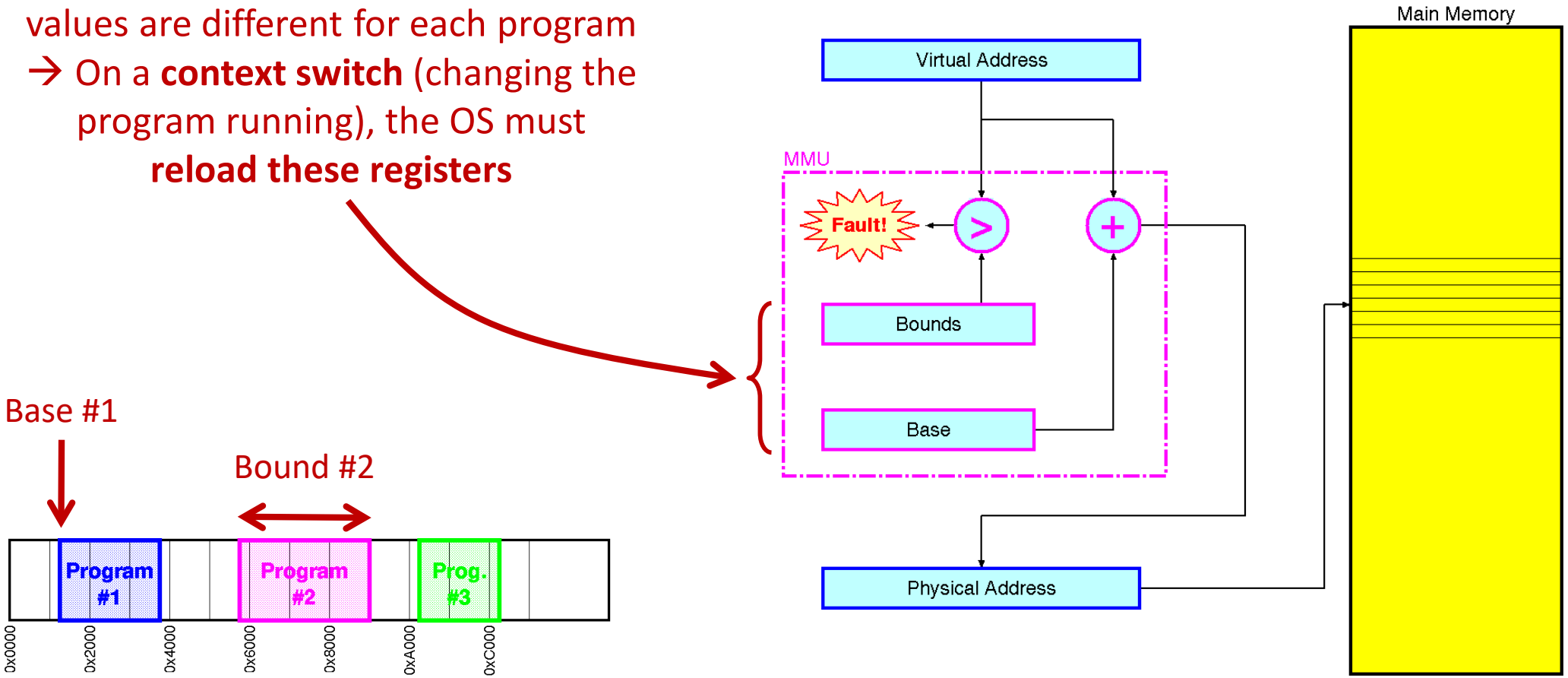
Relocation in Hardware: *Base and Bounds* MMU



If the program does that, we raise an **exception** and most likely **the OS will kill the program**

Relocation in Hardware: *Base and Bounds* MMU

Of course, the **Base** and **Bounds** values are different for each program → On a **context switch** (changing the program running), the OS must **reload these registers**



Needs of Multiprogrammed System

- **Relocation**
  - All programs must be written without knowledge of where they will be in memory
- **Protection**
  - Programs can access only their own data
- **Space**
  - If several programs run at the same time, memory shortage will be even more a problem

Protection is a but crude: **one chunk of memory**

Space allocation may need **garbage collection**, **moving programs and data**, etc.

Segmentation and Paging

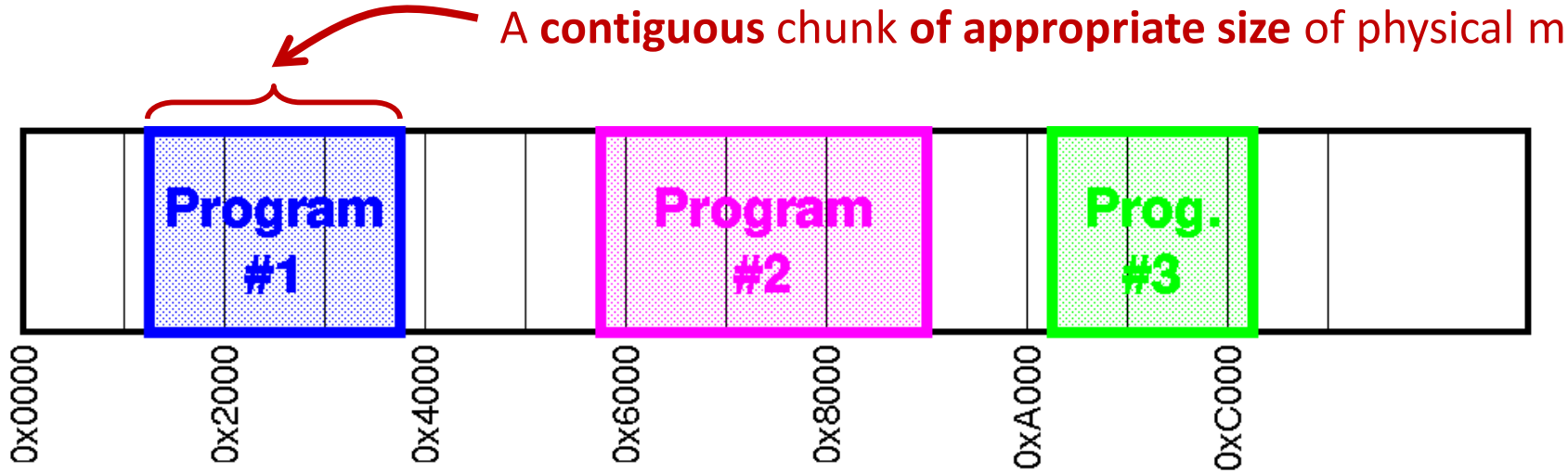
- **Segmentation** (an extension of **Base&Bounds**) splits the physical memory exactly as needed by each program
  - Arbitrary start of a block
  - Arbitrary length
  - Multiple blocks per application

- **Paging** splits the memory in equal small blocks (e.g., 4-64KiB) and assigns as many as needed to each program

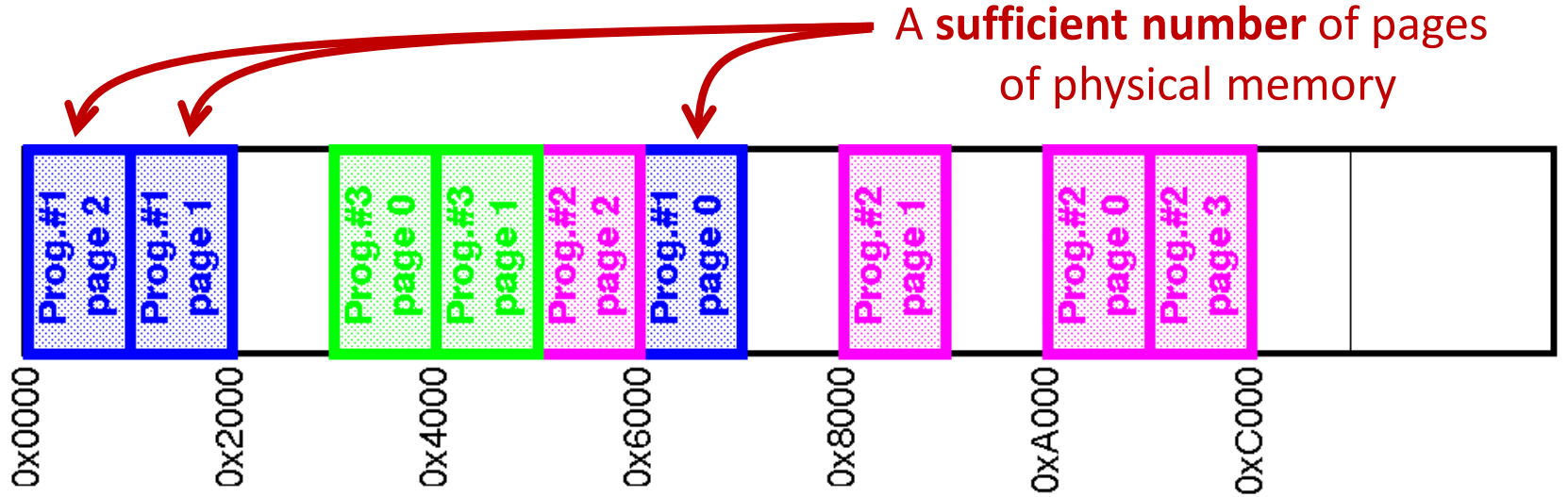
As often, names are not always consistent…

Segmentation and Paging

Segmentation

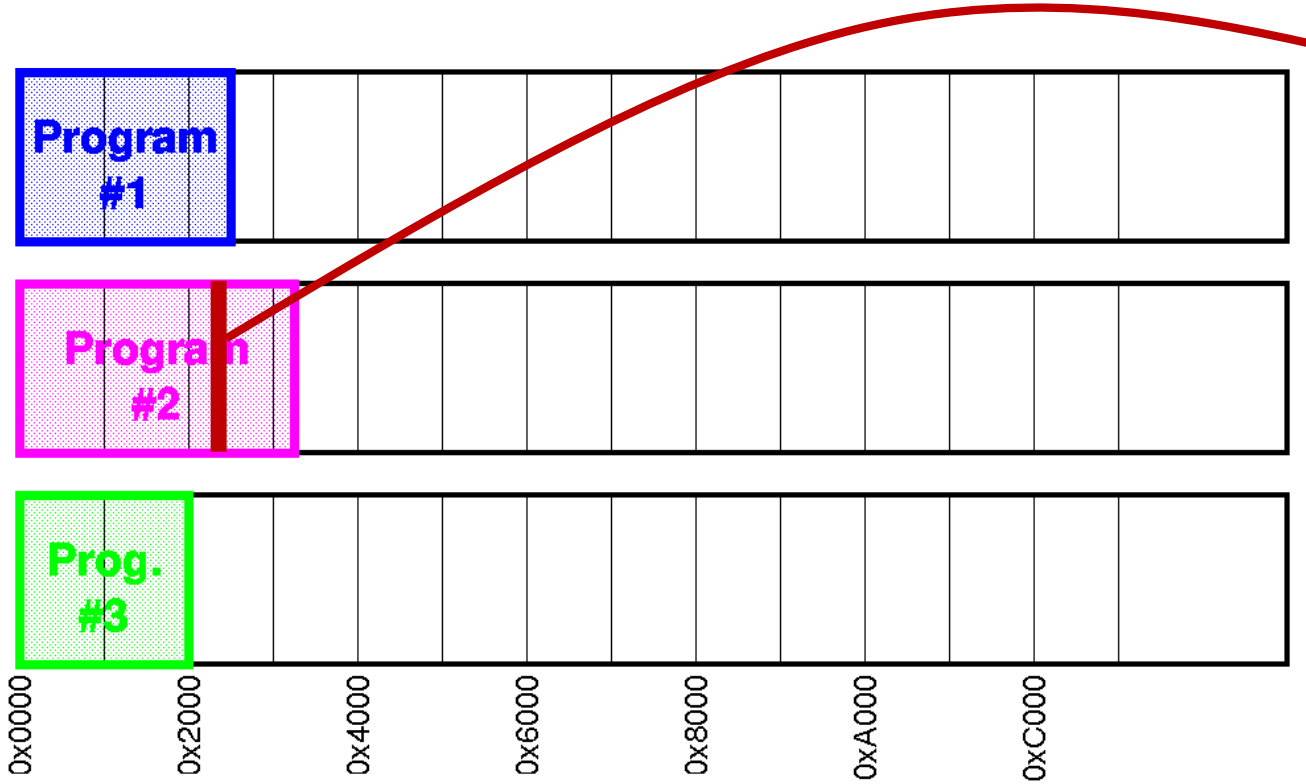


Paging

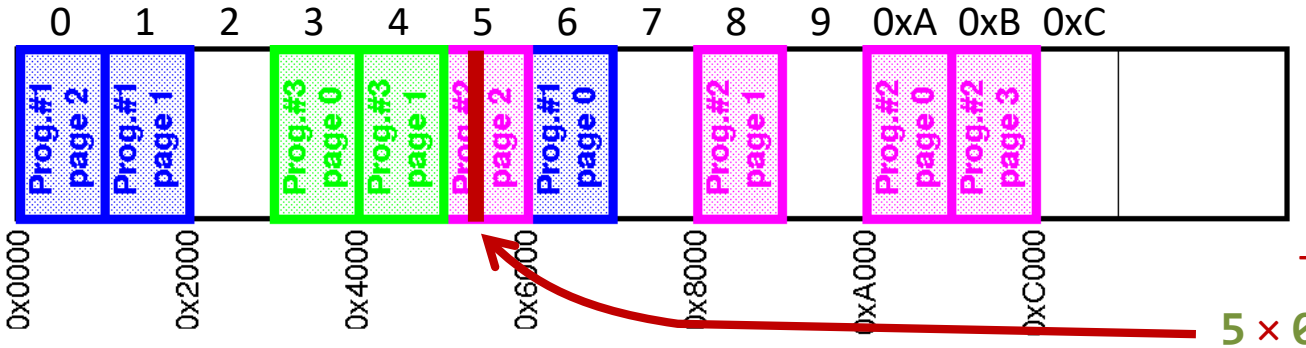


How Do We Translate Now?

Virtual Memories



Physical Memory



Where is in **physical memory**, the **virtual address** `0x2345` of **Program #2**?

If pages are `0x1000` in size. it is part of the page

`0x2345` / `0x1000` = **page 2**

and it is at this position in the page

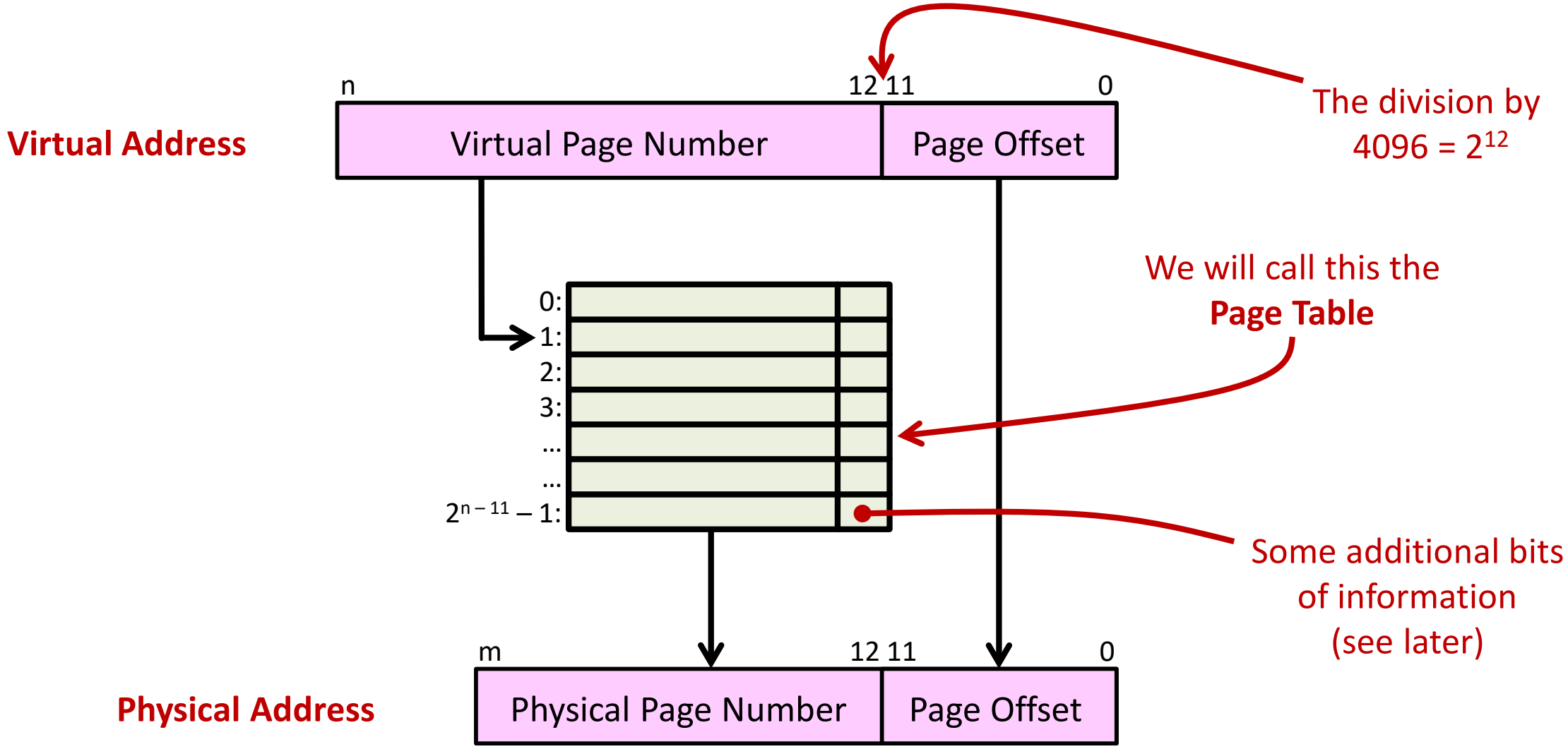
`0x2345` mod `0x1000` = `0x345`

Program #2	
0	0xA
1	8
2	5
3	0xB

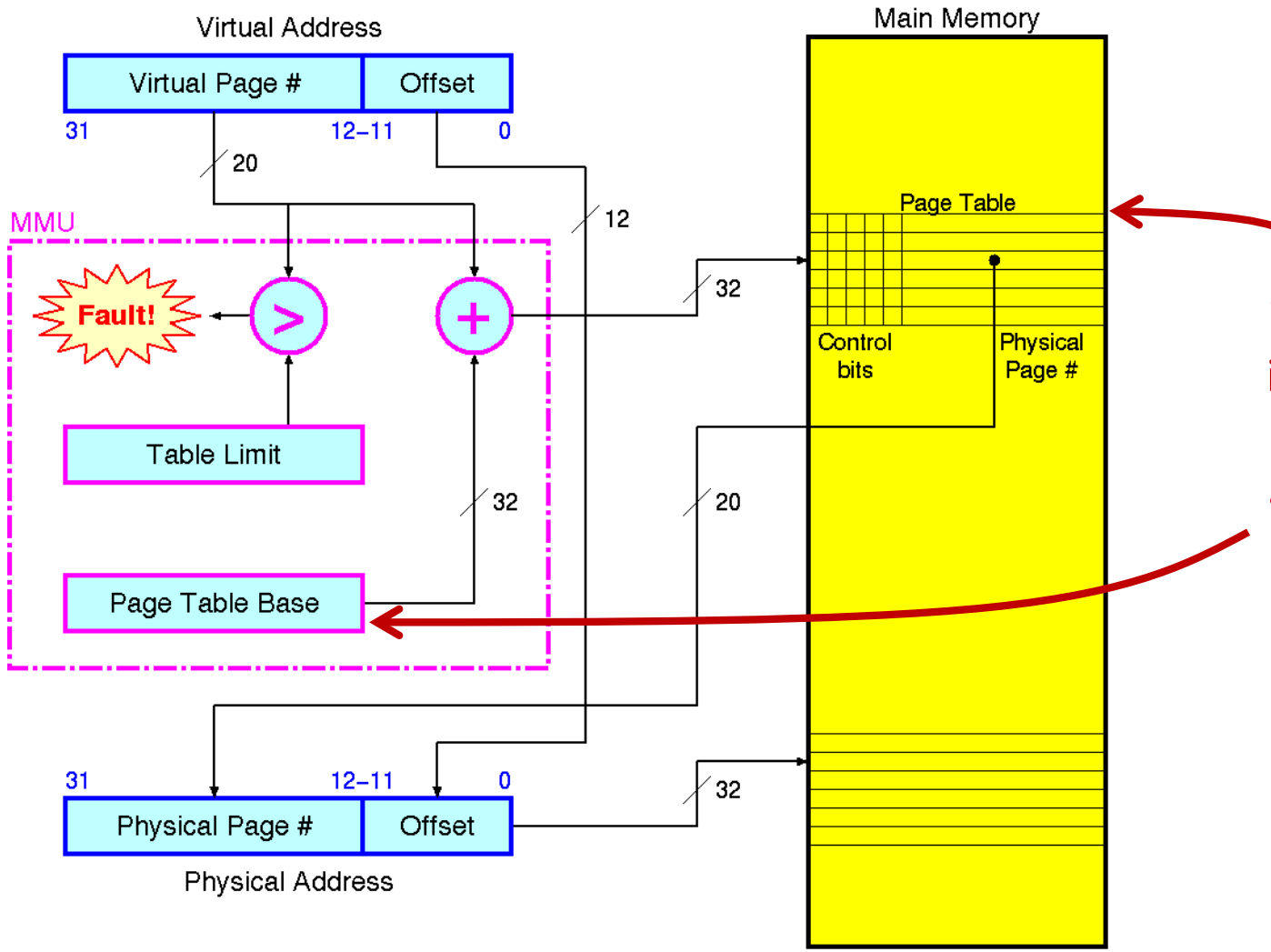
The physical address is

5 × `0x1000` + `0x345` = `0x5345`

How Do We Translate Now?



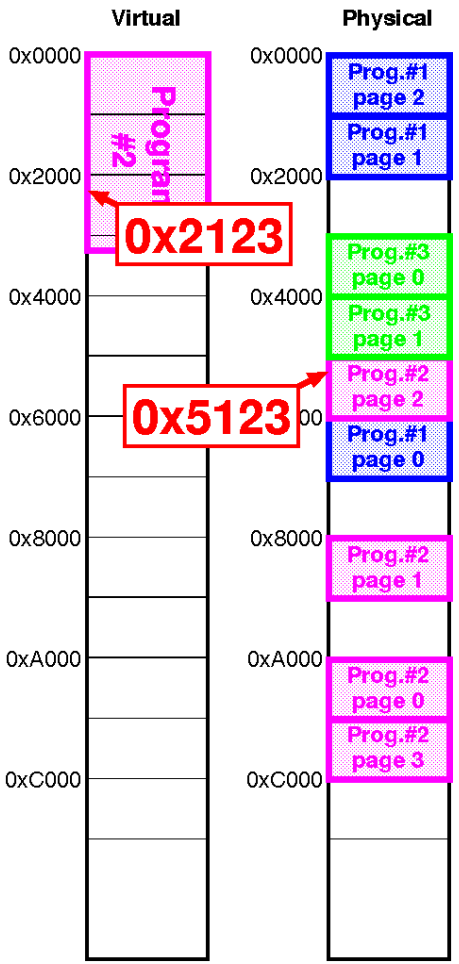
Virtual Address Translation in a Paged MMU

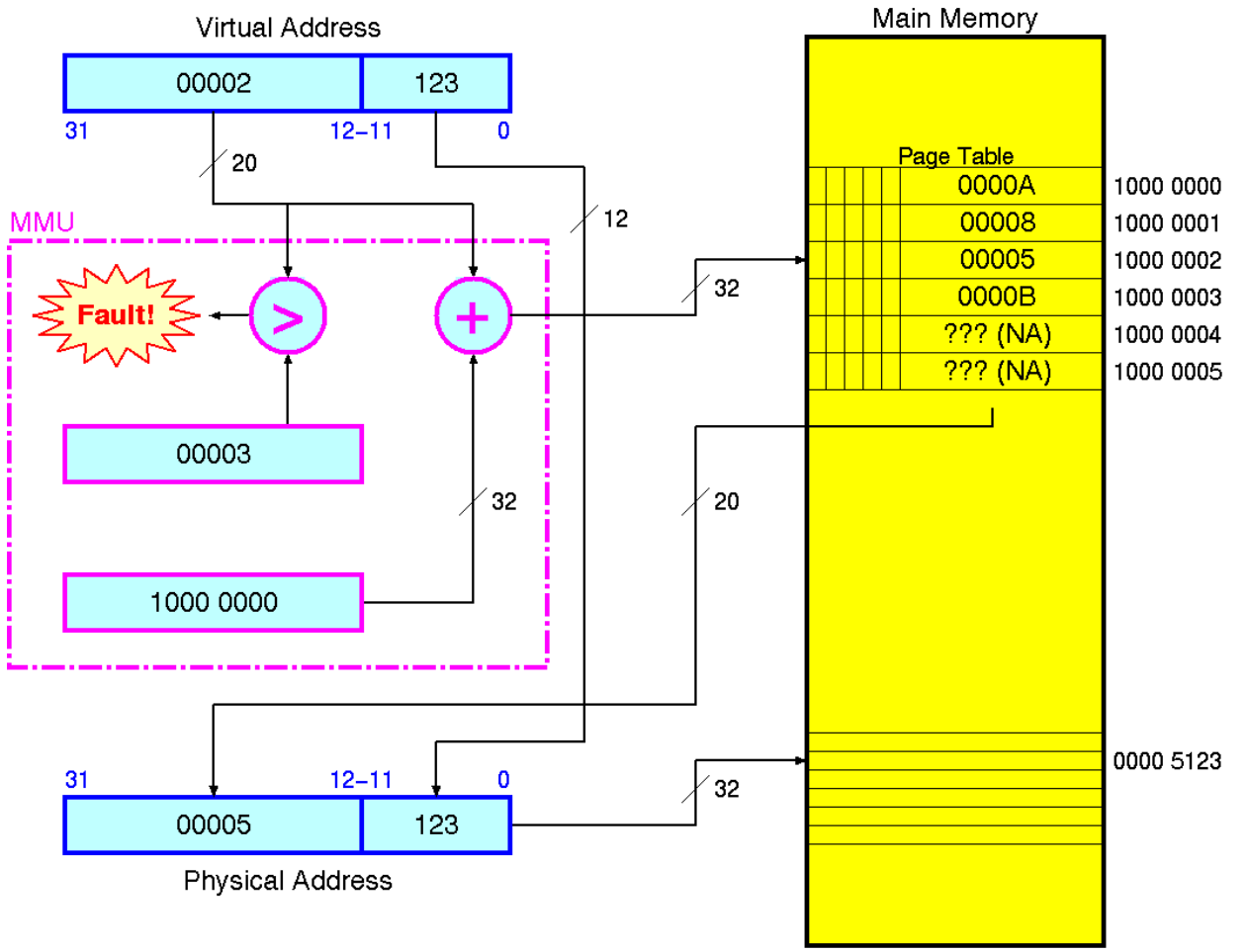


We place the **Page Table** in memory somewhere…

…and we give its address to the **MMU**

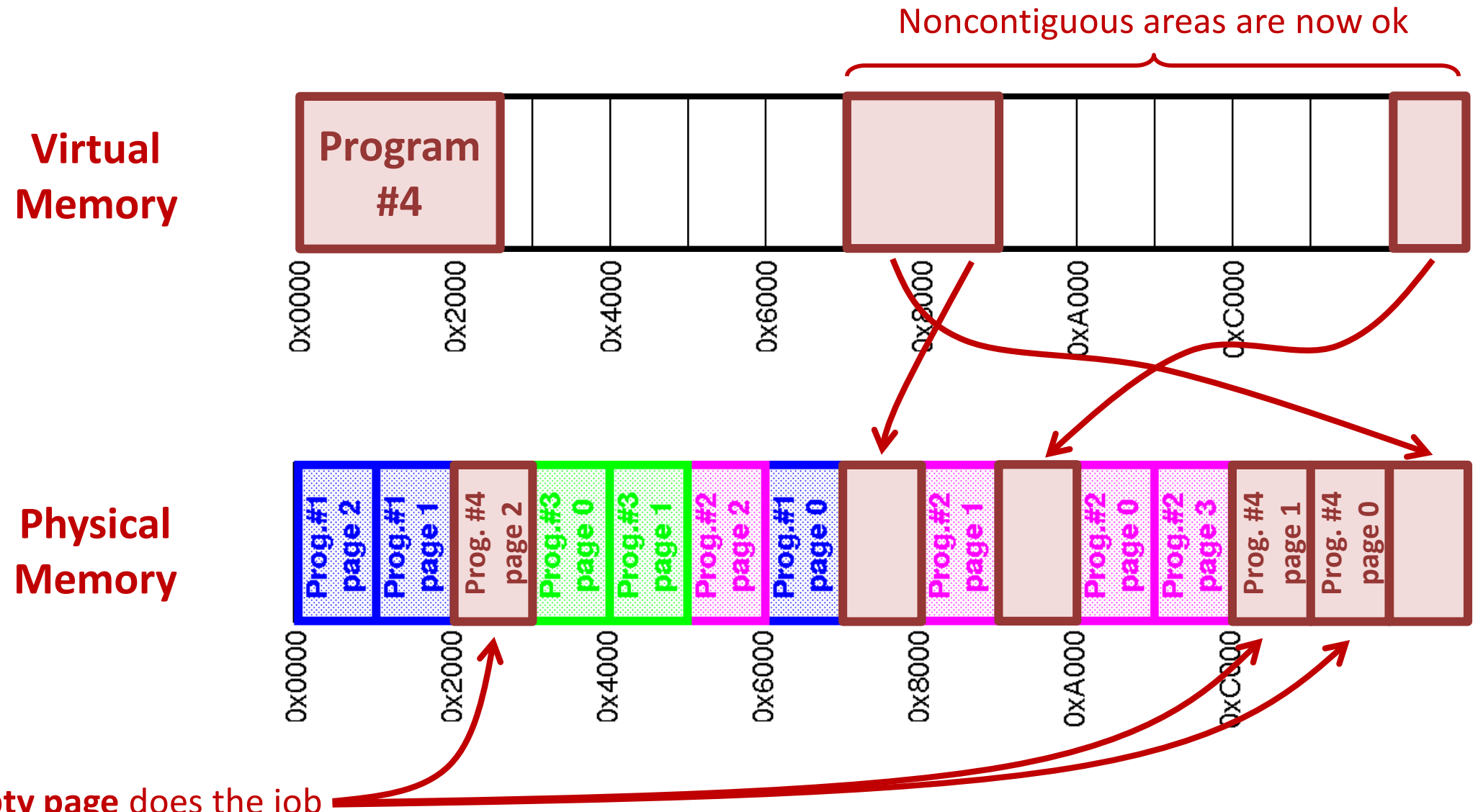
Virtual Address Translation for Program #2





A different page table per program

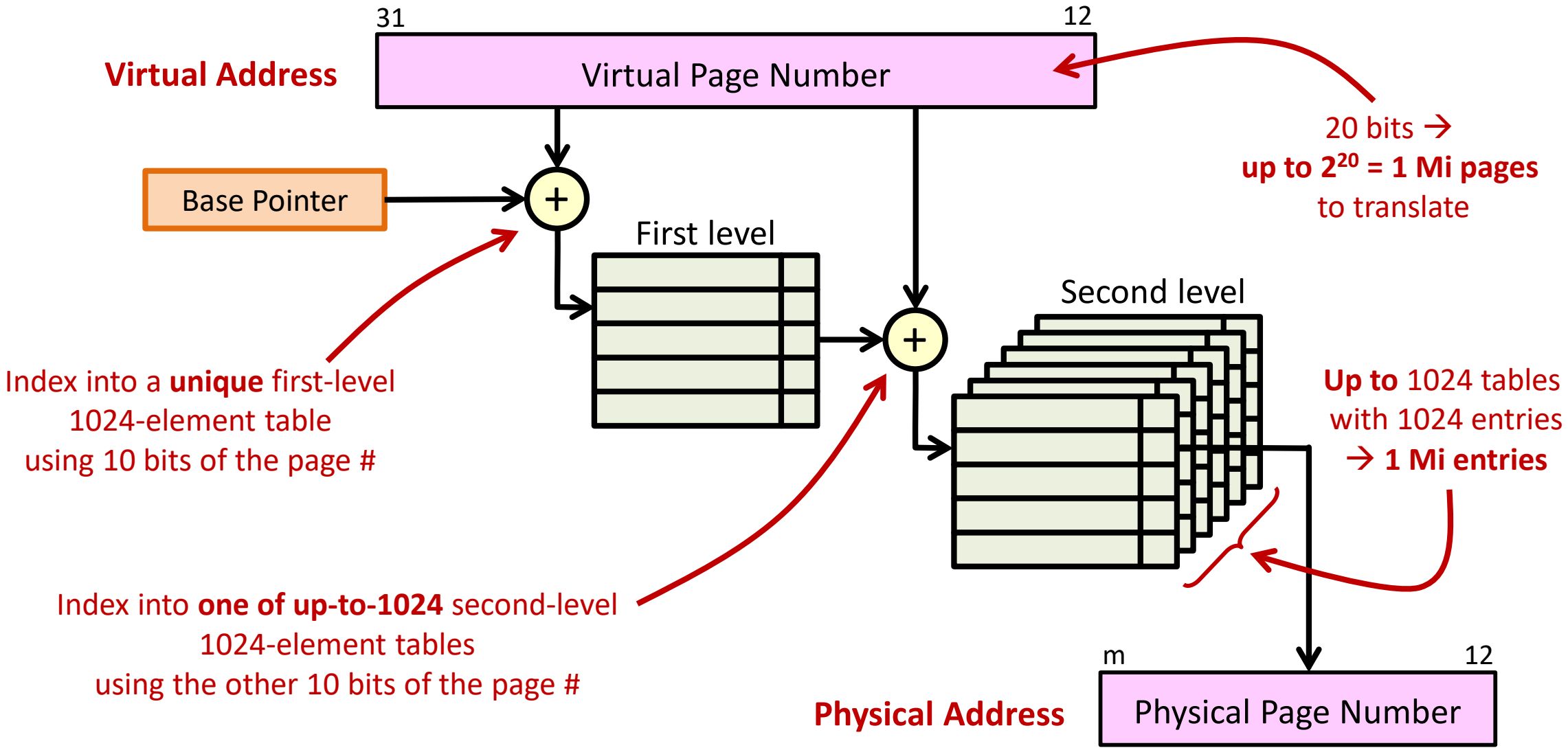
Memory Allocation Is Easy Now



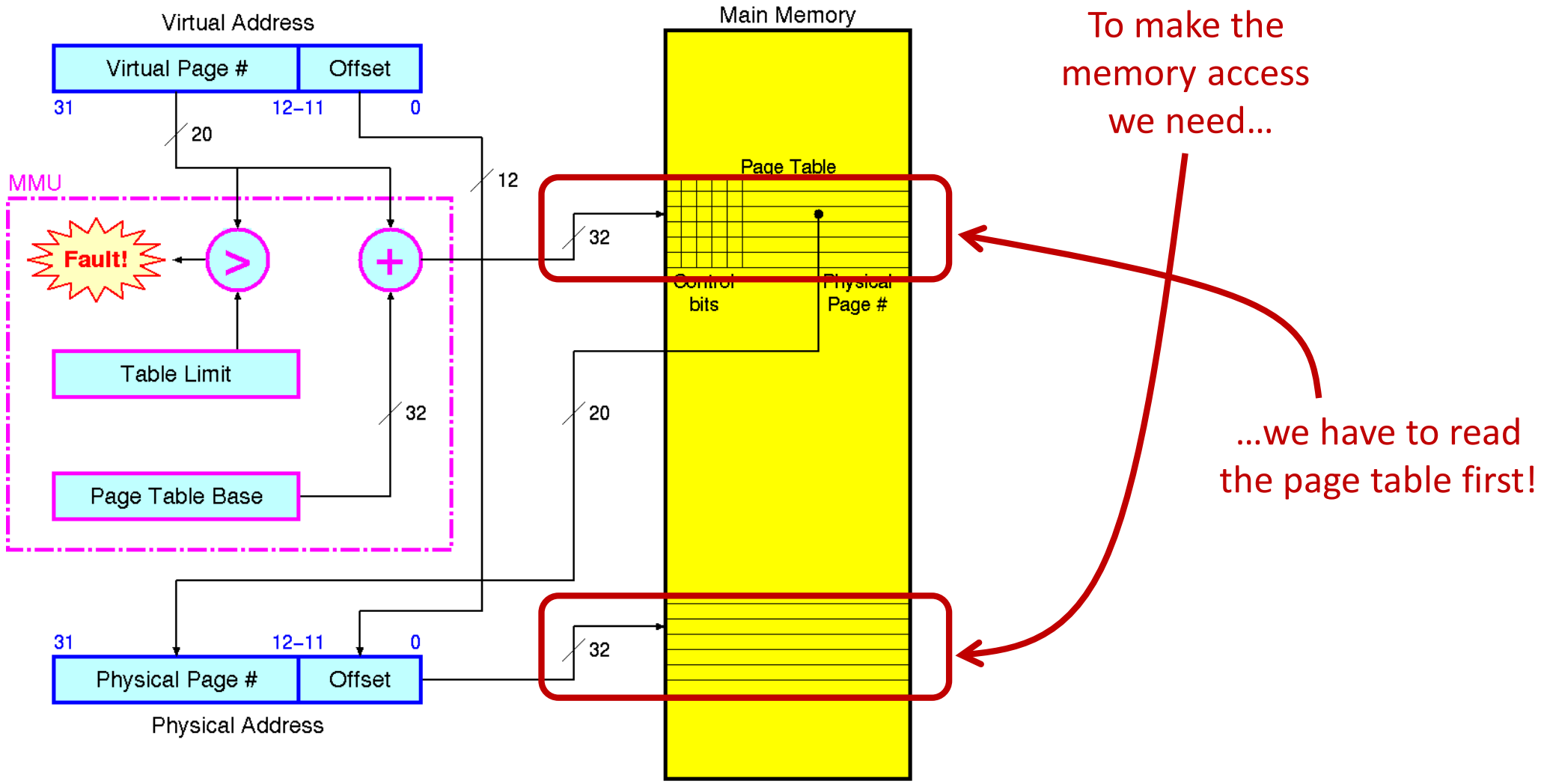
Page Tables Can Be Big

- Page tables could be **very large**
  - E.g., 64 GiB of memory in 4 KiB pages require 2^{24} entries or **~64 MiB**
- For a program that uses only a few MB, **most entries are empty**
- Several possible solutions (see COD and exercise book):
  - Hashed Tables
  - Paged Segmentation
  - **Multilevel Page Tables**
  - …

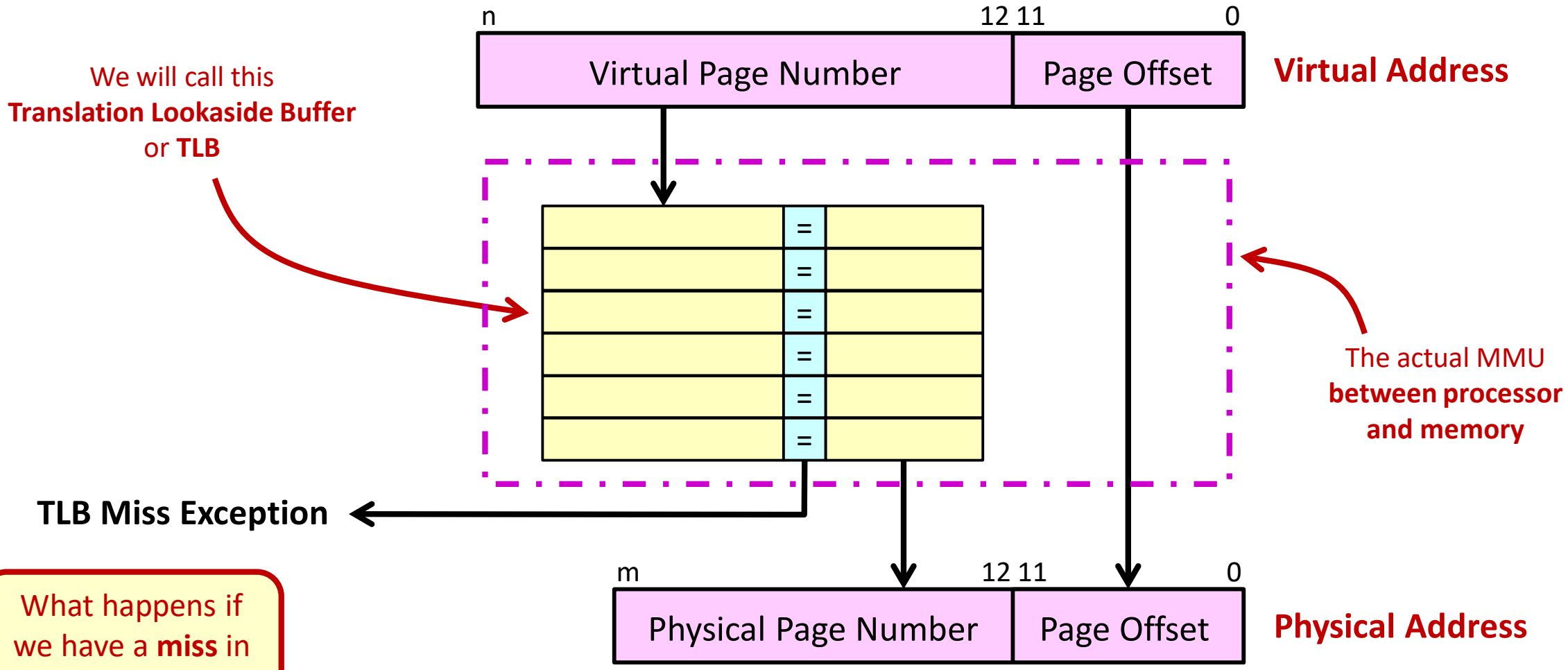
Multilevel (or Hierarchical) Page Tables



Two Memory Accesses Every Time?!



A Specialized "Cache" for the Translations



What happens if we have a **miss** in the **TLB**?!

TLB Miss

- The processor gets an **exception**:
  - The user program **stops execution**
  - The OS is invoked and **searches** the translation in the **Page Table**
  - If it **does not find the translation**, the user is trying to access memory that has not been allocated to it → **kill the program** and we are done
  - Otherwise, it **places the translation in the TLB**
  - **Restarts execution** from the user program's memory instruction that generated the TLB Miss
  - By construction, this time the **TLB will hit and the user program will continue**

Memory Protection

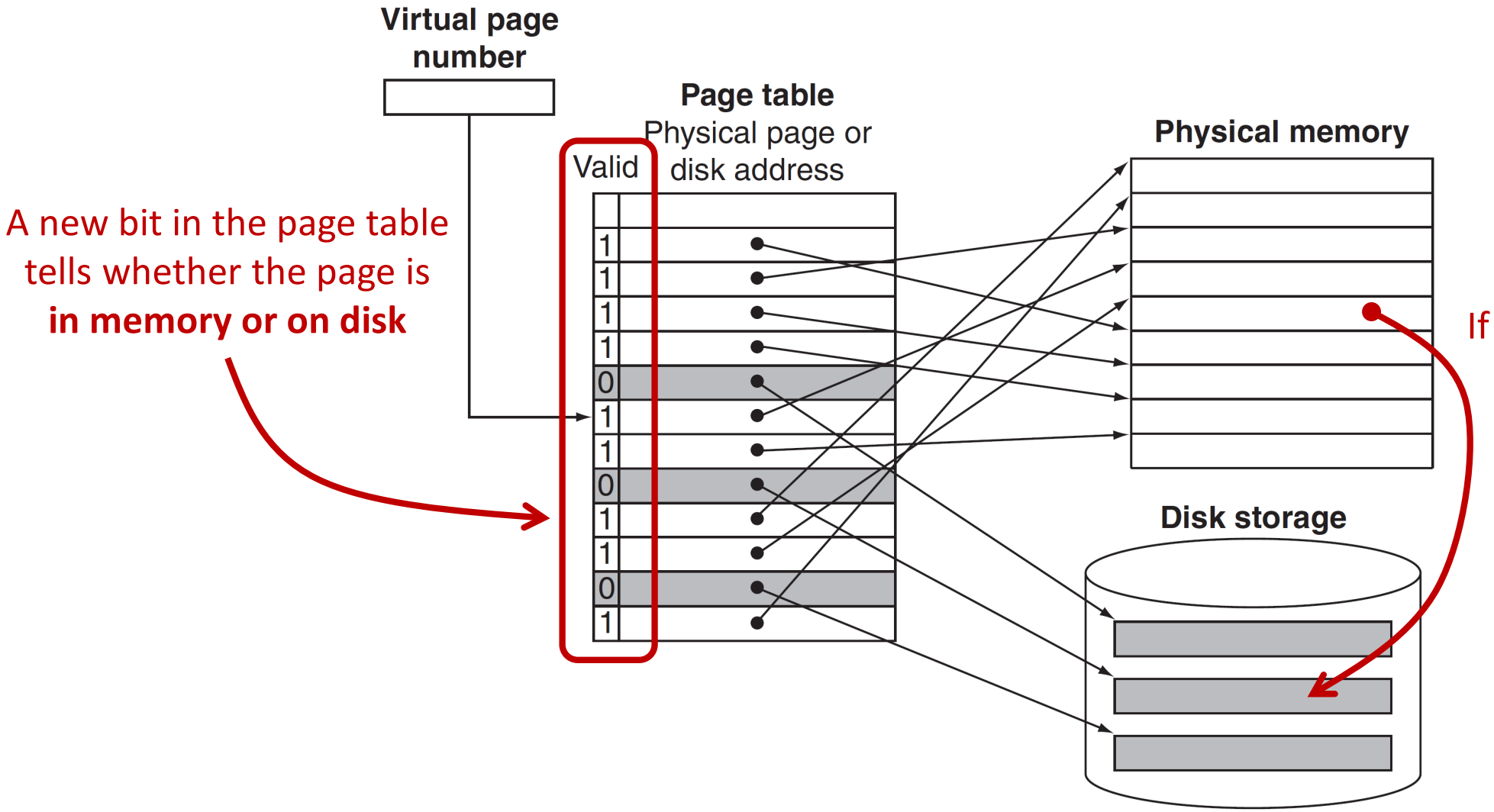
- Typically **Page Table** entries have several attributes (OS specific):
  - Valid (to indicate presence in main memory)
  - Allocated (to indicate existence}
  - Dirty (to indicate a copy-back is needed)
  - Used (to help determine which page to replace)
  - **Readable**
  - **Writable**
  - **Executable**
  - …

- If the **TLB can be written only by the OS** (e.g., kernel mode), the OS can **protect the Page Tables** (prevent users from writing them), **protect its code**, and thus control completely **memory access rights**

Needs of Multiprogrammed System

- **Relocation**
  - All programs must be written without knowledge of where they will be in memory
- **Protection**
  - Programs can access only their own data
- **Space**
  - If several programs run at the same time, memory shortage will be even more a problem

Not All Pages Need to Be in Main Memory!



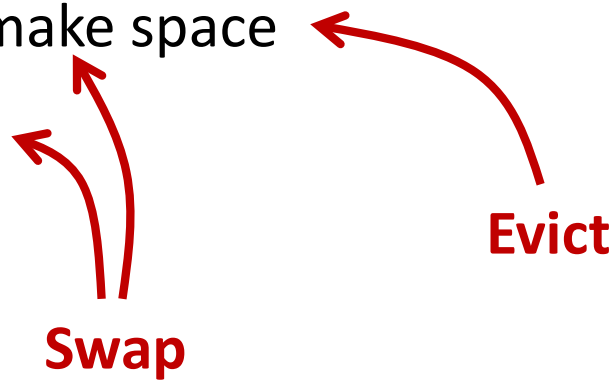
A new bit in the page table tells whether the page is **in memory or on disk**

If **memory is full**, a page can be moved to **disk**

Source: COD, © Morgan Kauffman 1998

TLB Miss – Revised

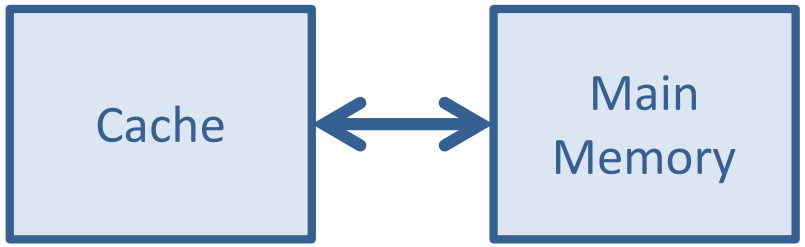
- When the OS **searches the page table** after a TLB miss, now there is a new possibility: the **addressed page is on disk**
  - Copy another **page from memory to disk** to make space
  - **Bring back into memory** the addressed page
  - **Update** the page table
  - **Update** the TLB
  - Continue as usual



- Were are these pages on disk? Depends on the OS…
  - Linux puts them in a special **raw** partition called `swap`
  - Windows puts them in the file `pagefile.sys`

Caches vs. Virtual Memory

Cache



Cache holds the most frequently used blocks

If cache is full → evict (LRU, etc.)

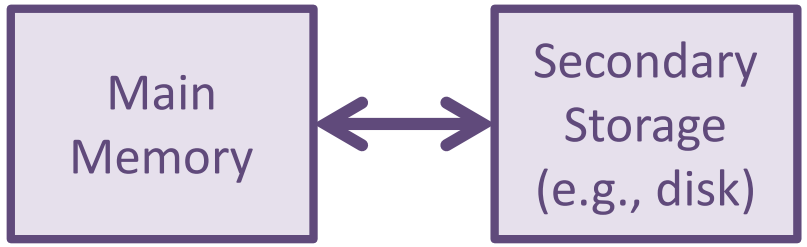
Cache miss → **penalty 10×-100×**

A cache block is typically **64 bytes**

Some caches can be **write-through**

A cache miss is resolved in **HARDWARE**

Virtual Memory



Main memory holds the most frequently used pages

If main memory is full → evict/swap (LRU, etc.)

Page fault → **penalty 100,000×-10,000,000×**

A page is typically **4,096 bytes**

Virtual Memory can only be **copy-back** → dirty bit

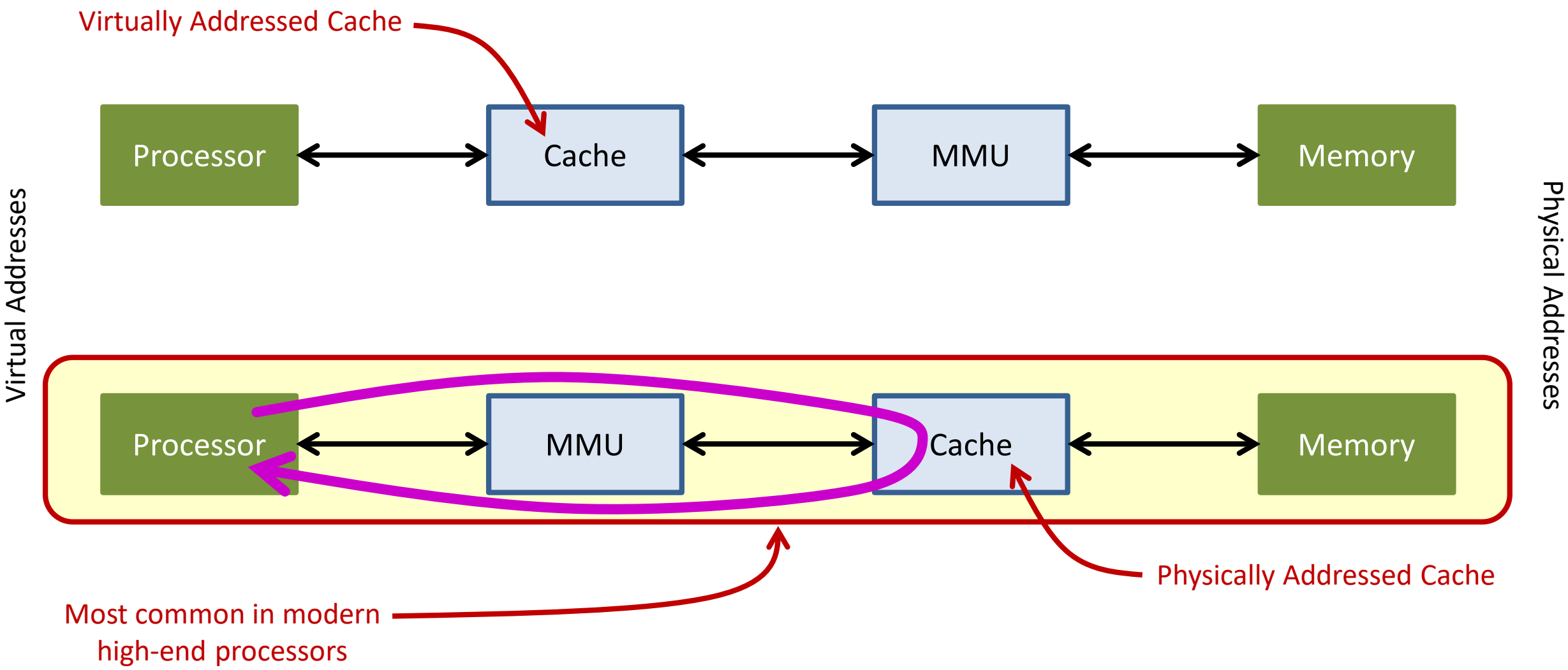
Clever **replacement policies**

A page fault is resolved in **SOFTWARE**

Page Table Attributes – Revisited

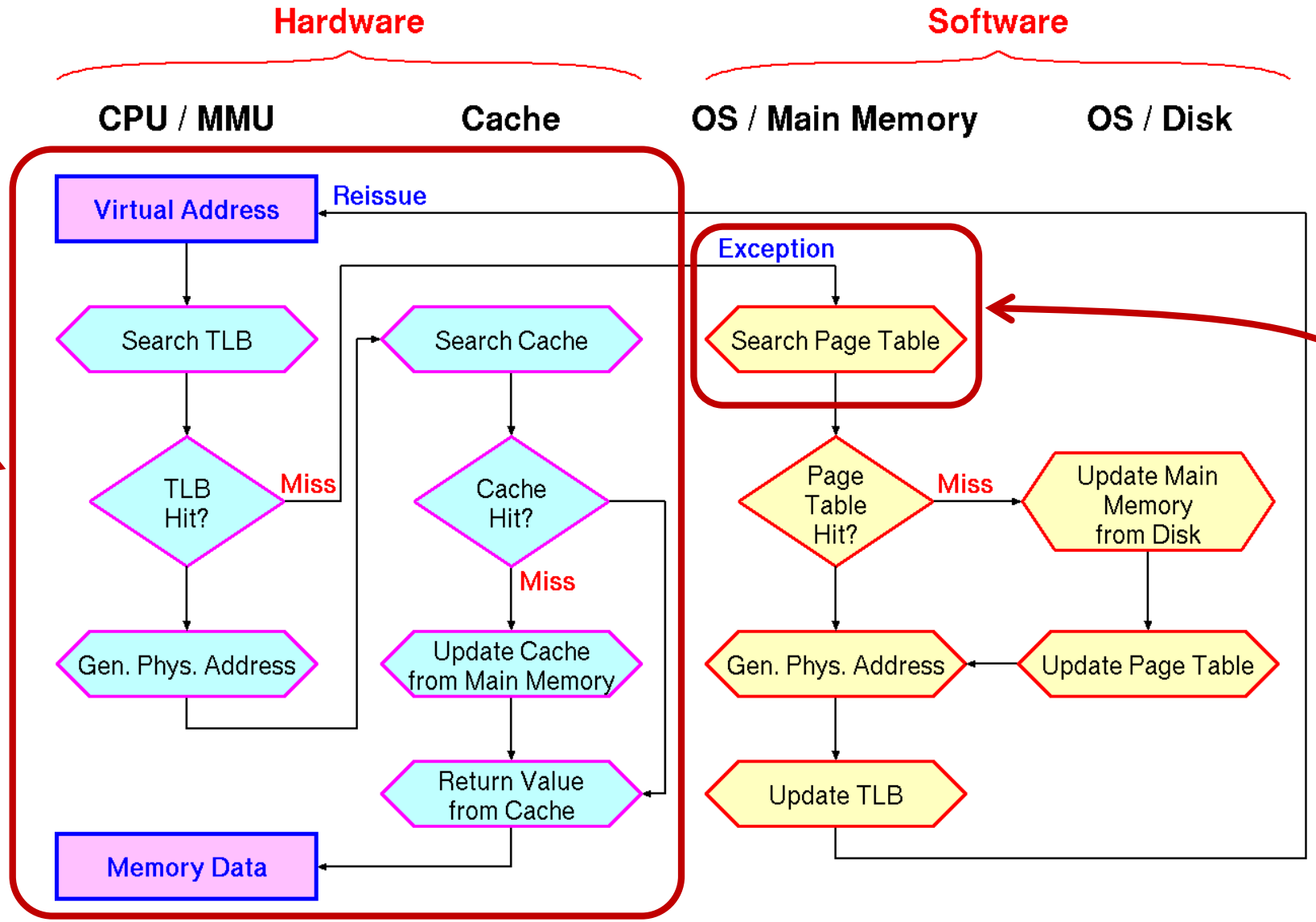
- Typically **Page Table** entries have several attributes (OS specific):
  - **Valid** (to indicate presence in main memory)
  - **Allocated** (to indicate existence}
  - **Dirty** (to indicate a copy-back is needed)
  - **Used** (to help determine which page to replace)
  - Readable
  - Writable
  - Executable
  - …

Virtual Memory ↔ Cache



TLB Misses, Cache Misses and Page Faults

If all goes well, everything in **hardware** → **FAST**



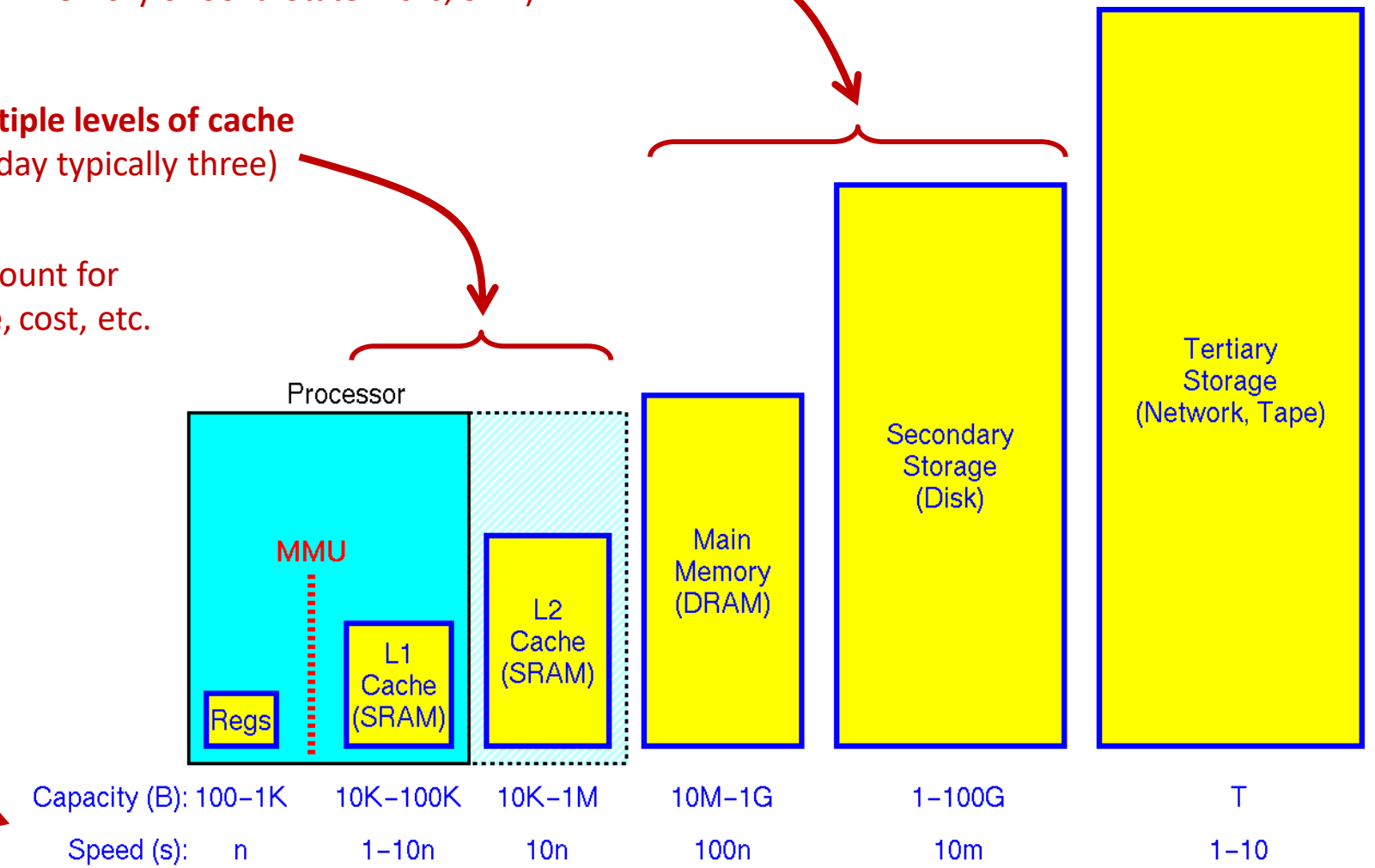
Modern processors also **search the page table in hardware**

Overall Picture: The System Side

New types of memory (e.g., FLASH is added either as main memory or Solid-State Disks, SDD)

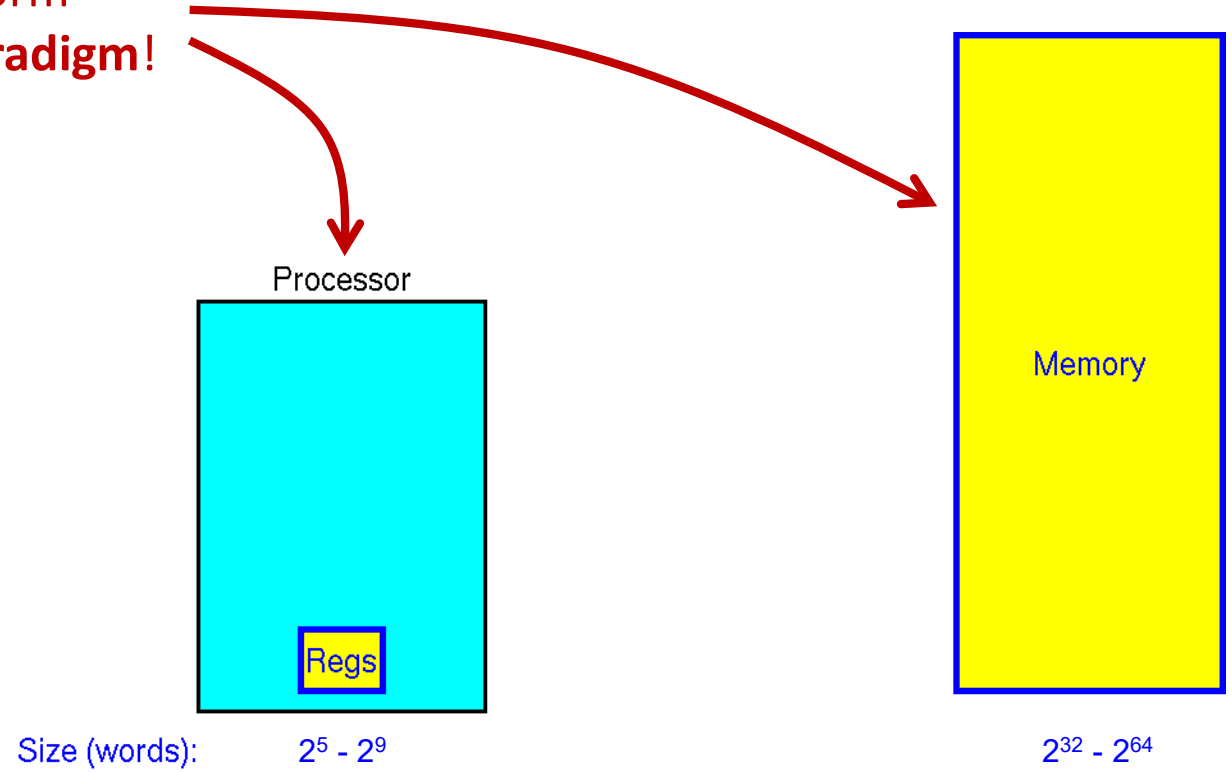
Multiple levels of cache (today typically three)

A complex design to account for **trade-offs in performance**, cost, etc.



Overall Picture: The Programmer Side

A **simple** uniform **programming paradigm**!



Summary

- **Virtual memory** offers the illusion of a perfectly uniform and identical memory system to each individual program
- Additionally, **virtual memory** is a form of caching between main memory and secondary storage
- A **Memory Management Unit** implements mechanisms to translate virtual addresses into physical ones
- **Translation Lookaside Buffers** are special the “caches” (software managed!) used to perform the translation efficiently in the MMUs
- As with caches, all this is **transparent to users**: programs read and write memory oblivious of all this—and exceptions are used to correct problems
- It is a complex interaction of hardware (MMU, TLB, caches) and software; **exceptions are an essential ingredient**

References

- Patterson & Hennessy, COD – RISC-V Edition
  - **Section 5.7**