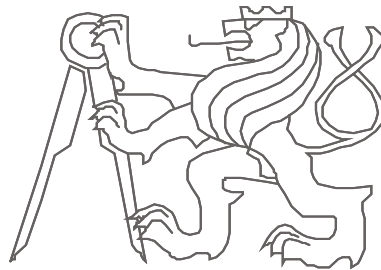


Computer Architectures

Virtual Memory

Richard Šusta, Pavel Píša

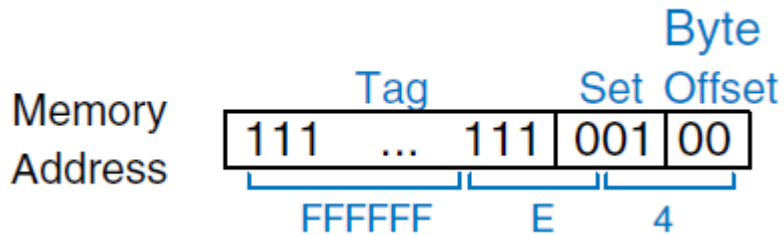


Czech Technical University in Prague, Faculty of Electrical Engineering

* *The last lecture...*

Direct Mapped Cache

One block only in each set



Number of sets – S

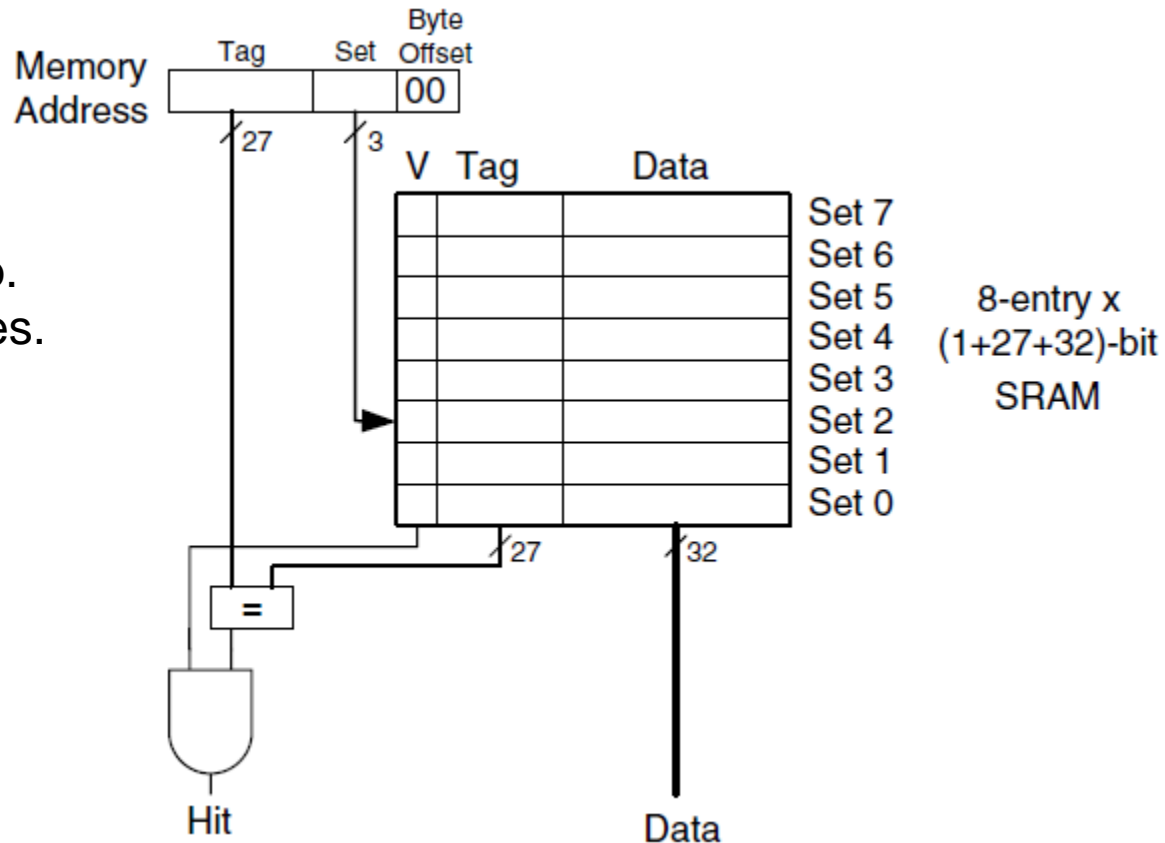
Number of blocks – B

in row, each block has size b.
In QtMips, b is always=4 bytes.

Degree of associativity – N

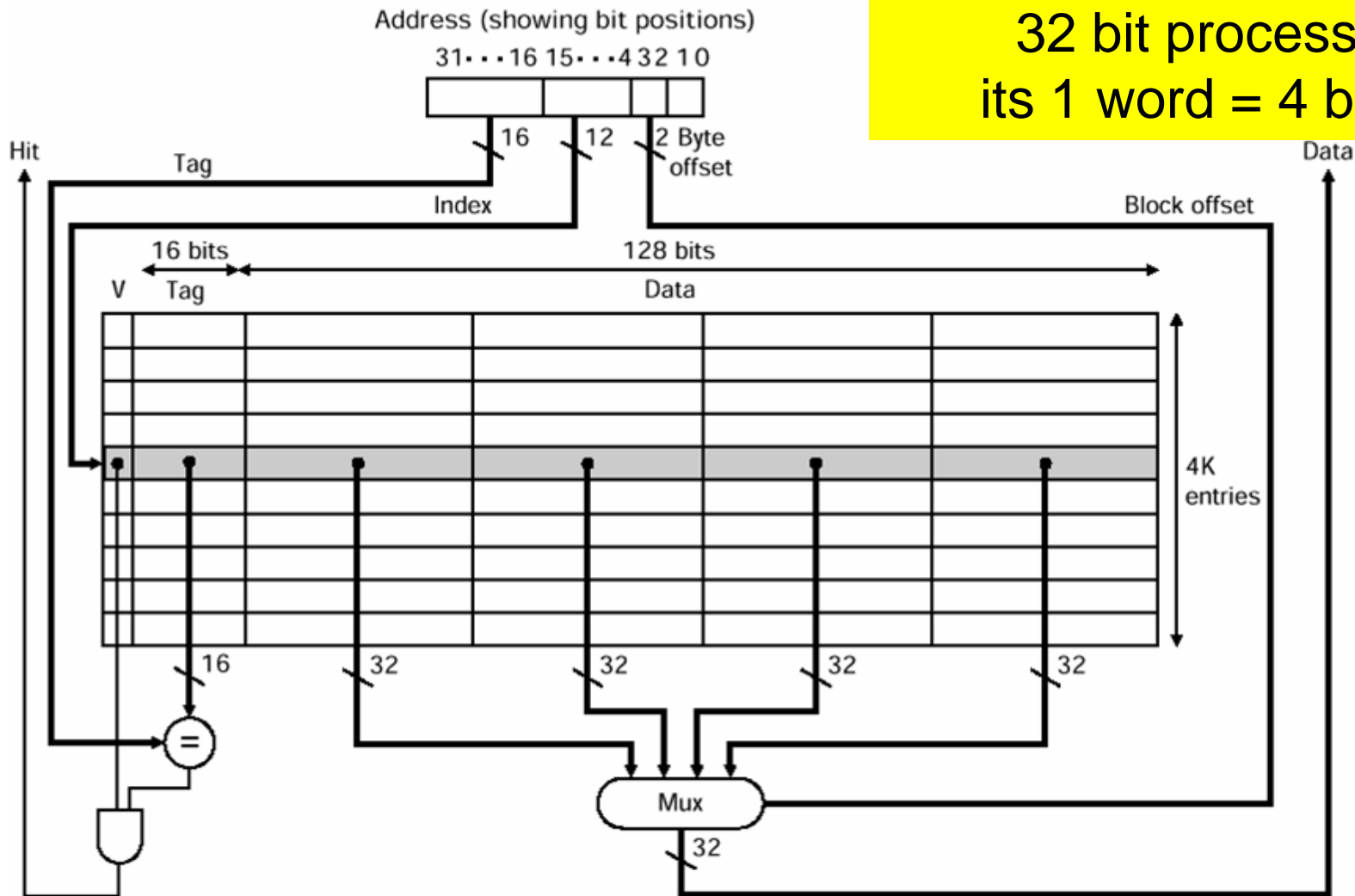
(rows in set)

for direct mapped cache N=1



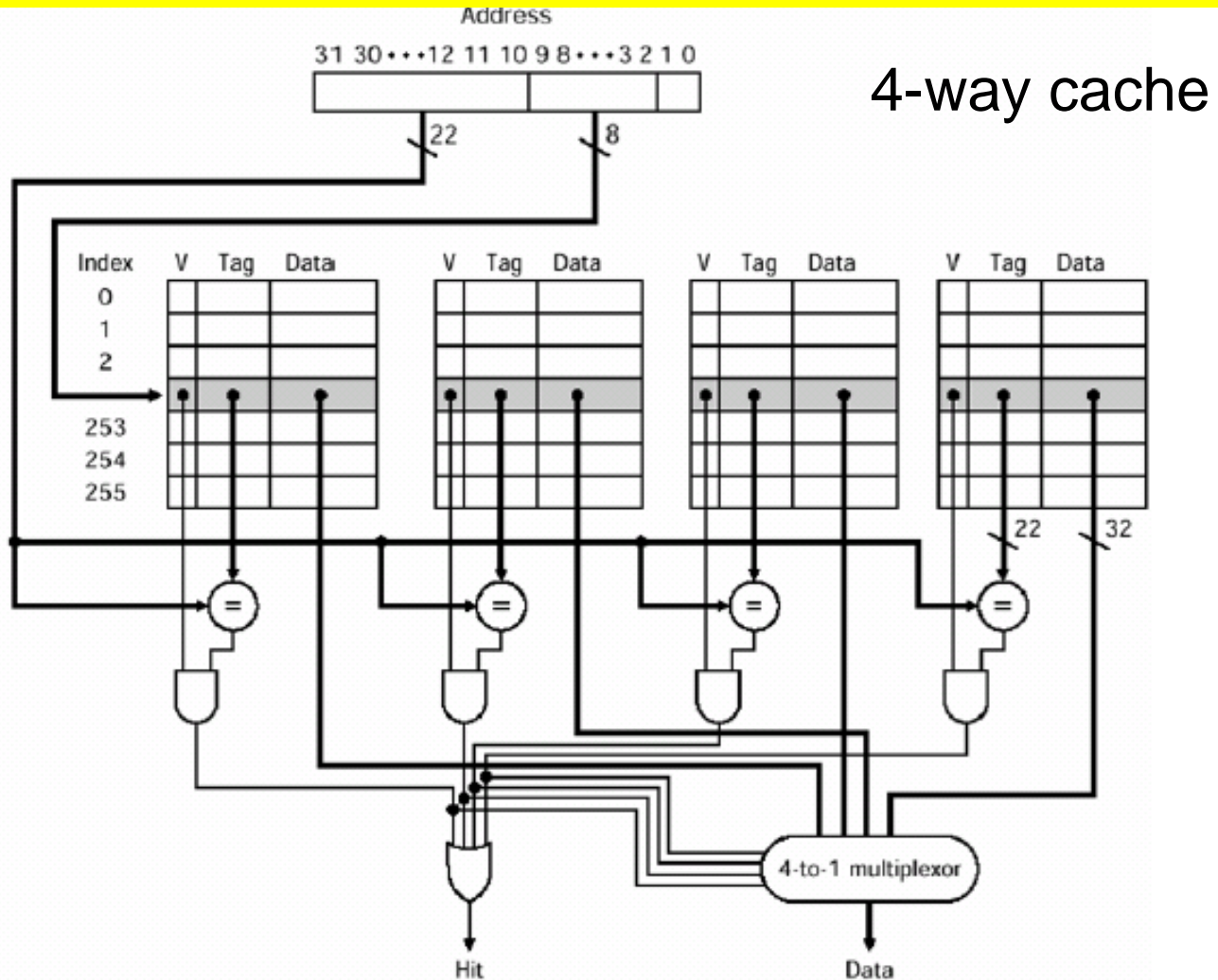
Direct mapped cache implementation

32 bit processor,
its 1 word = 4 bytes

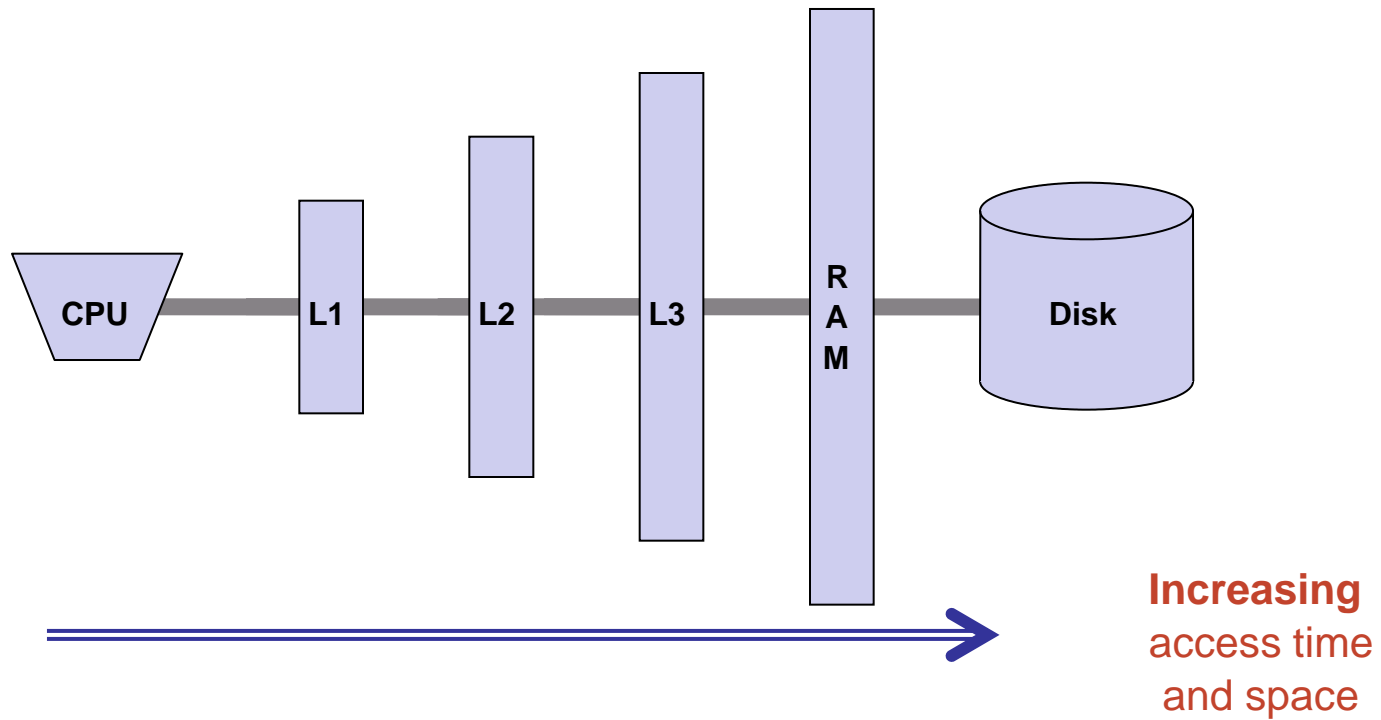


4-way cache

32 bit processor 1 word = 4 bytes



Memory Hierarchy



Q: Which sorting program is better?

Suppose we got the following results in QtMips, for some unnamed sorting programs, and access times (chosen to count on us):

miss = 1 clock cycle - cache + 9 clock cycles - memory

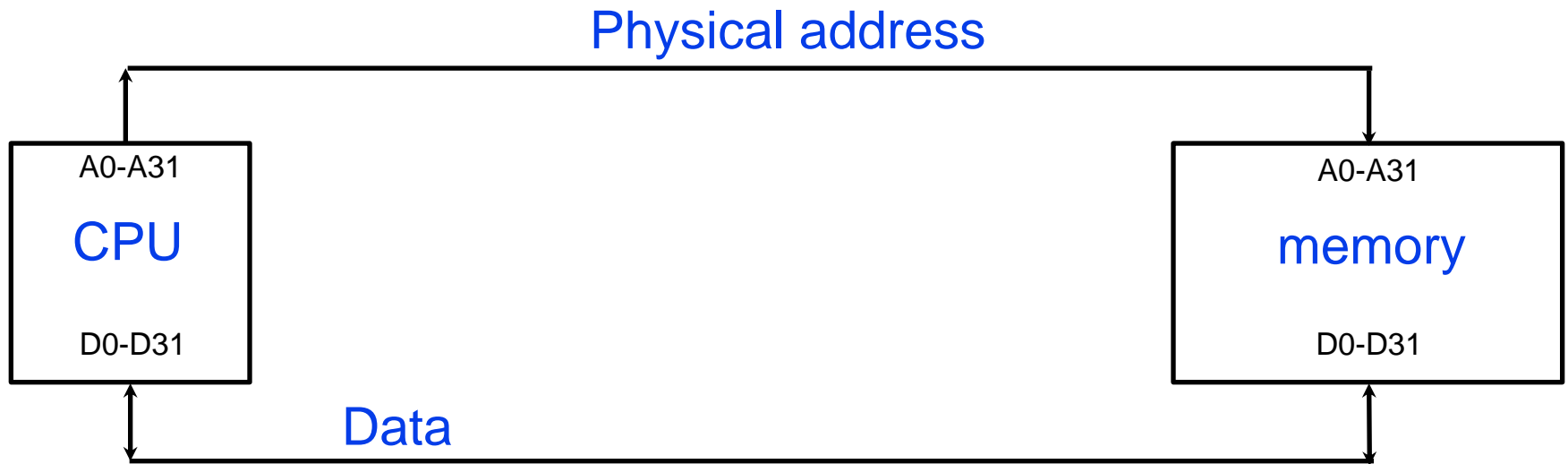
hit = 1 clock cycle cache

Cache			Sort 1			Sort 2		
S	B	N	Hit	Miss	Improved	Hit	Miss	Improved
4	1	1						
1	1	4	250	200	180 %	50	150	~115 %
2	1	2						
16	1	1	435	15	690 %	185	15	~540 %

S - number of sets, B - number of blocks, N - Degree of associativity

**Today...
Virtual memory*

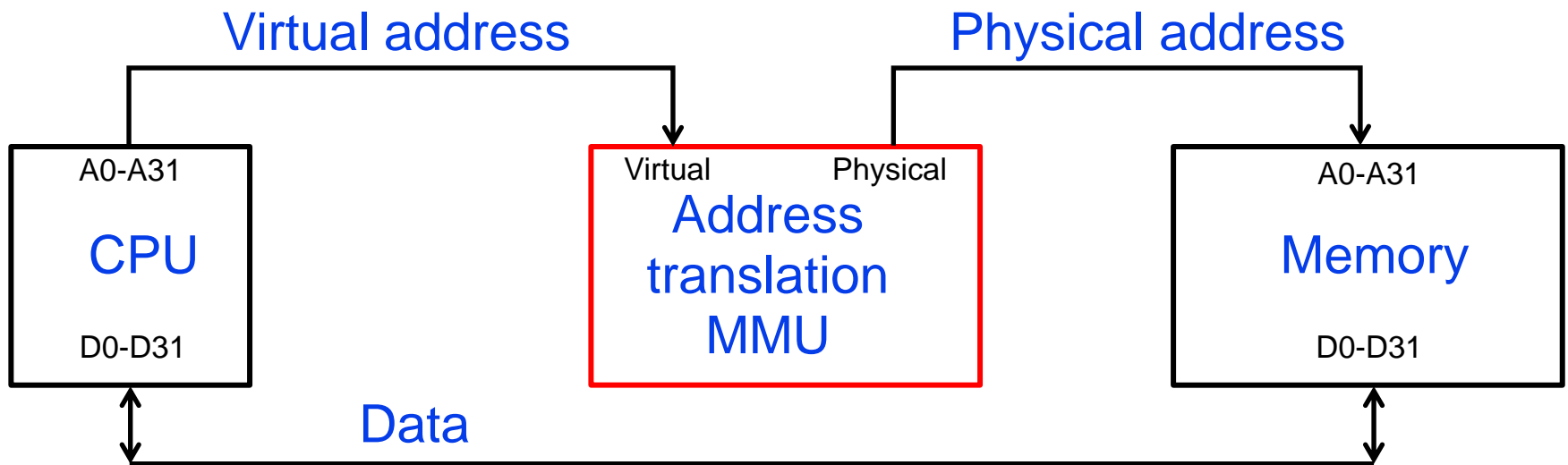
Physical address to memory?



Virtual Memory Motivation ..

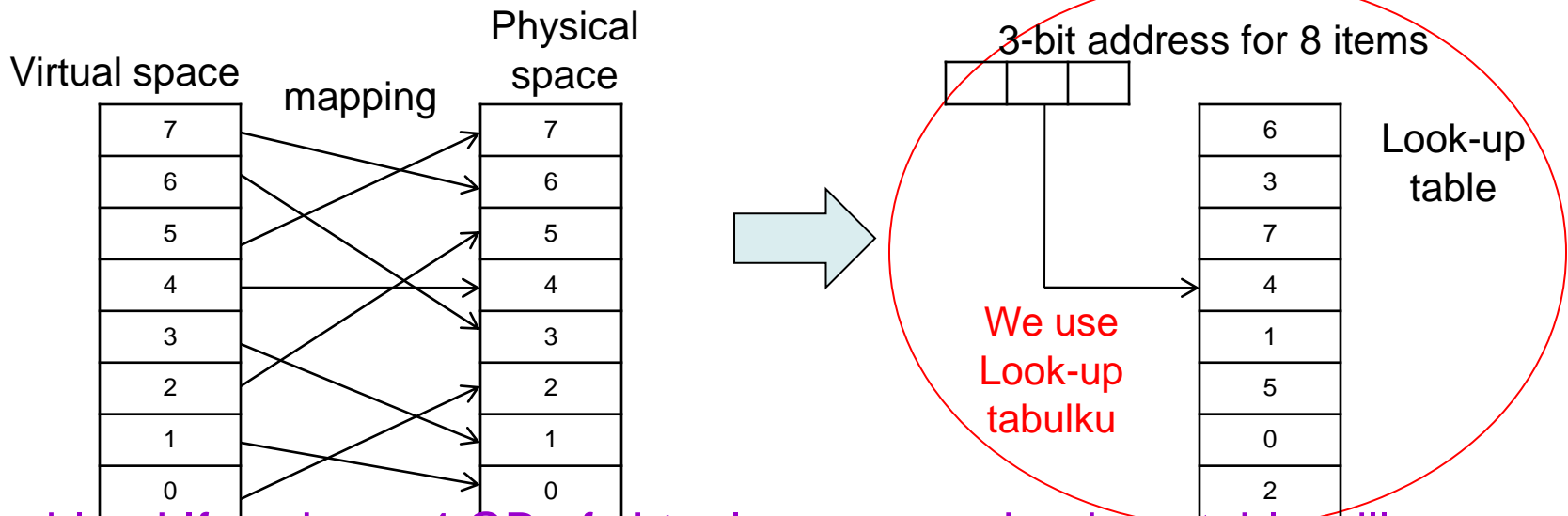
- Normally we have several tens / hundreds of processes running on your computer...
- Can you imagine a situation where we would divide physics memory (for example, 1 GB) between these processes? How big a piece of memory would belong to one process? How would we deal with collisions - when would a program intentionally (for example, a virus) or inadvertently (by a programmer's error - working with pointers) want to write to a piece of memory that we reserved for another process?
- The solution is just virtual memory...
- We create an illusion to every process that the entire memory is just its and can move freely within it.
- We will even create the illusion of having, for example, 4GB of memory even though the physical memory is much smaller. The process does not distinguish between physical memory and disk (the disk appears to be memory).
- **The basic idea: The process addresses the virtual memory using virtual addresses.** We then have to translate them somehow into physical addresses.

Virtual/physical address and data



Virtual Memory Motivation

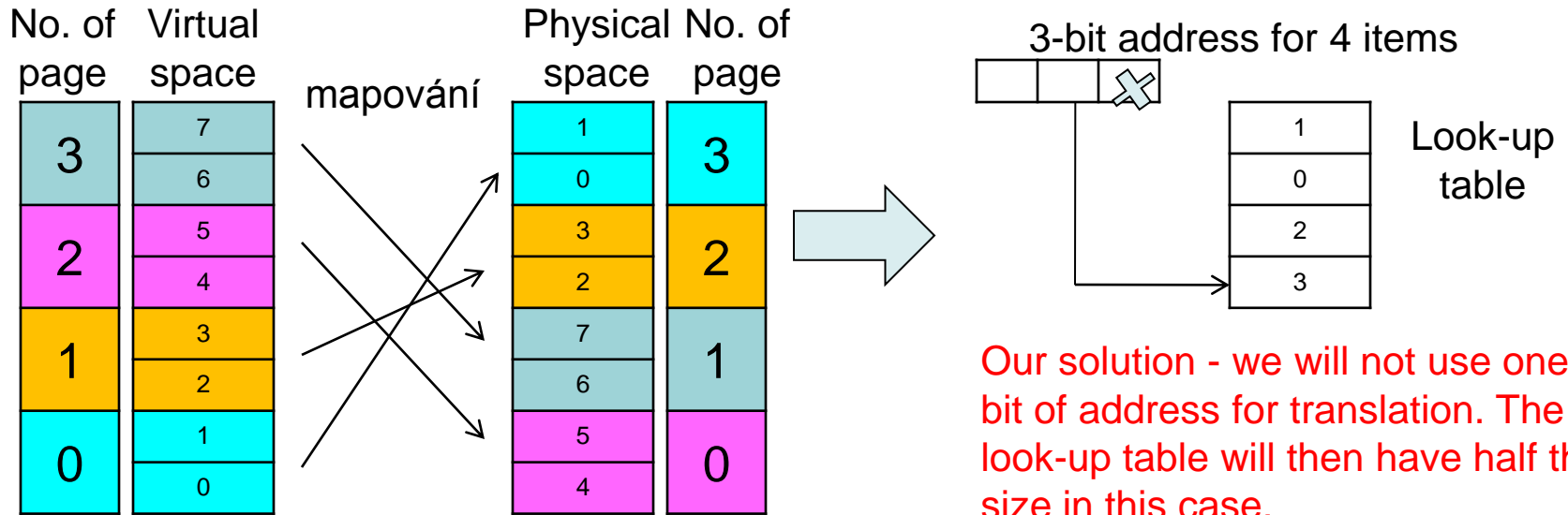
- Imagine that we have 8B (Bytes) virtual space and 8B physical memory...
- How do we provide address translation? Assume addressing by bytes.
- Here is one solution: We want to translate any virtual address to any physical address. We have a 3-bit virtual address, and we want to translate it to a 3-bit physical address. To do this, you need a table of 8 records where one record will have 3 bits, together $8 \times 3 = 24$ bit / process.



- Problem! If we have 4 GB of virtual space, our Look-up table will occupy $2^{32} \times 32$ bits = 16GB / process !!! That's a little bit...

Motivation to virtual memory - Lesson from previous slide :

- **Mapping from any virtual address to any physical address is a virtually unrealistic requirement!**
- **Solution:** Divide the virtual space into equal parts - virtual pages, and physical memory on physical pages. Make the virtual and physical size the same. In our example, we have a 2B page.



Our solution - we will not use one bit of address for translation. The look-up table will then have half the size in this case.

- So our solution translates virtual addresses in groups... We move inside the page using the bit we ignored during the translation. We are able to use the entire address space.

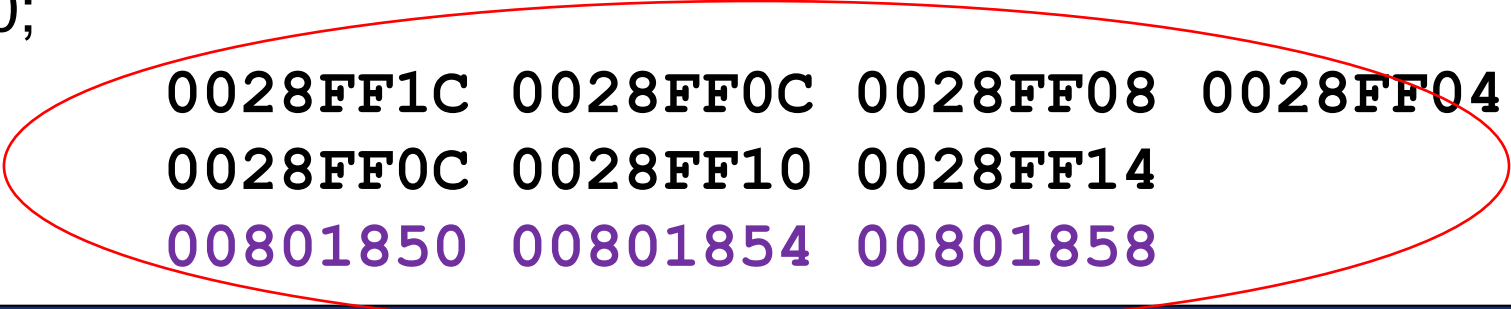
Memory addressing virtualization

- Memory addressing virtualization is a method of managing memory that allows a running memory space access process to be organized differently, or even greater than the physically attached memory.
- The conversion between the virtual address and the physical address can be supported by the processor (HW TLB mapping, see below).
- In current operating systems, virtual memory is implemented by memory paging along with disk paging, which extends the memory to disk space.

Example 1

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a, b[4], *c, d;
    c = (int*)malloc(4*sizeof(int));
    printf("%p %p %p %p\n",&a,&b,&c,&d);
    printf("%p %p %p\n",&b[0],&b[1],&b[2]);
    printf("%p %p %p\n",&c[0],&c[1],&c[2]);
    free(c);
    return 0;
}
```

Output:



```
0028FF1C 0028FF0C 0028FF08 0028FF04
0028FF0C 0028FF10 0028FF14
00801850 00801854 00801858
```

What does that mean?

- The data is stored consecutively in the field.

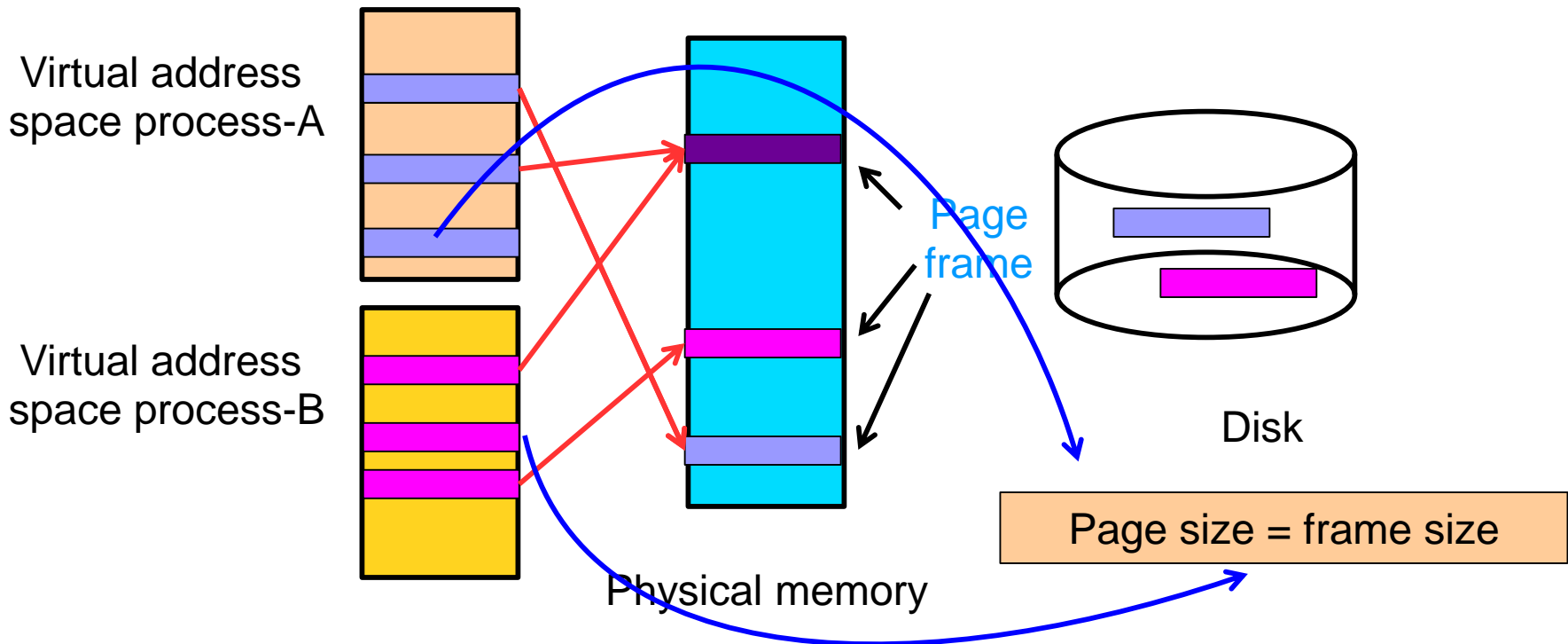
Questions that are offered ..

- But what is that address?

Where to cache this data?

Virtual memory - Paging

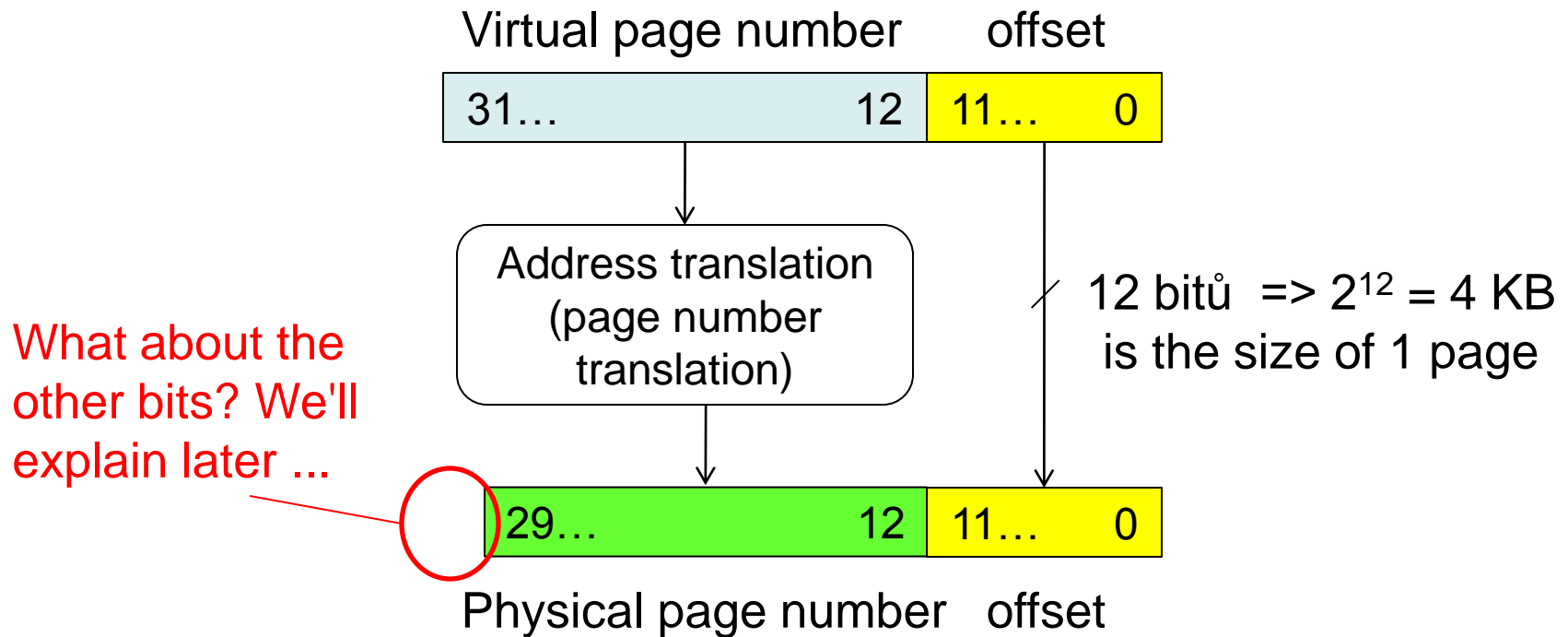
- Process virtual memory content is divided into aligned pages of same size (power of 2, usually 4 or 8 kB)
- Physical memory consists of page frames of the same size



Note: Some systems allowed page with different sizes, e.g. Silicon Graphics IRIX.

Virtual and physical addressing - in more detail

- Assume a 32-bit virtual address, **1GB of physical memory**, and a 4-KB page size



The arrangement of the translation, where the lowest bits of the address remain, has a very important practical consequence, see below.

Let's return to example 1

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a, b[4], *c, d;
    c = (int*)malloc(4*sizeof(int));
    printf("%p %p %p %p\n",&a,&b,&c,&d);
    printf("%p %p %p\n",&b[0],&b[1],&b[2]);
    printf("%p %p %p\n",&c[0],&c[1],&c[2]);
    free(c);
    return 0;
}
```

Output:

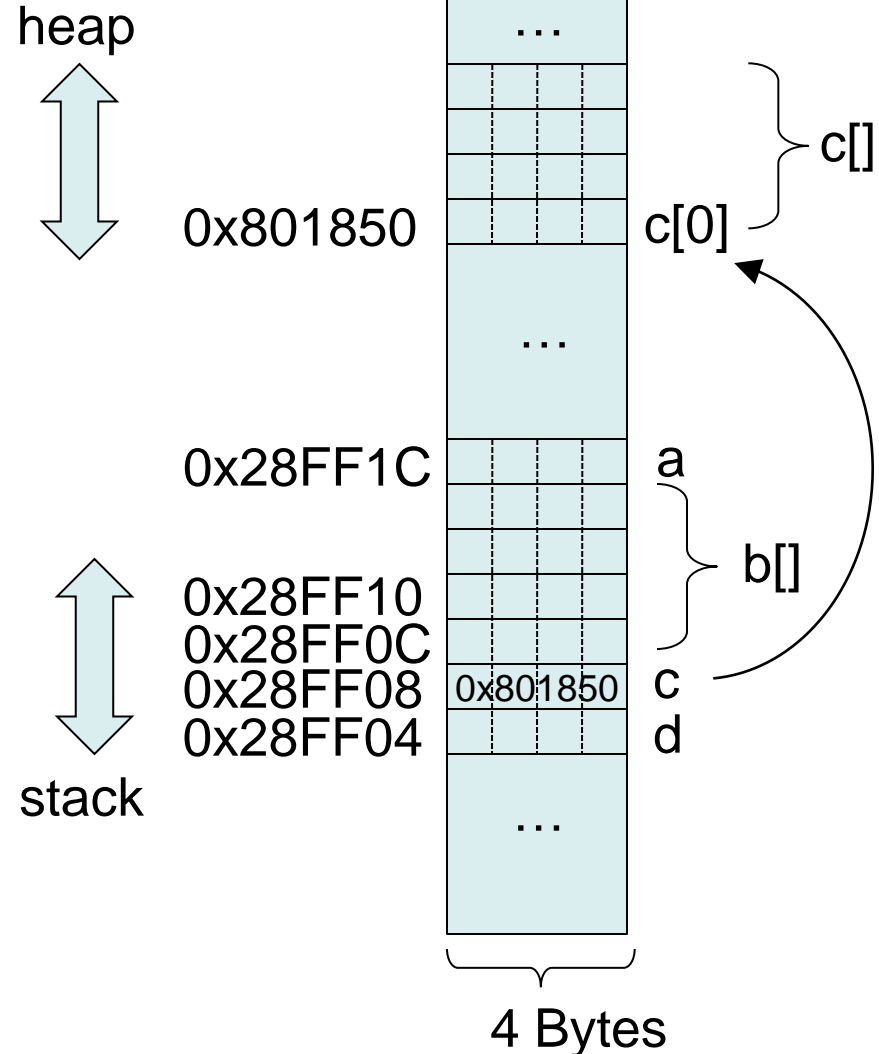
```
0028FF1C 0028FF0C 0028FF08 0028FF04
0028FF0C 0028FF10 0028FF14
00801850 00801854 00801858
```

Let's return to example 1

Virtual address space:

- Have you noticed the addresses on which the variables `a`, `c`, `d`, and `b` are located?
- What if we want to extend our program with commands like:

```
a = 1;  
b[0] = a+1;  
b[1] = b[0]+1;  
d = b[2];  
//b[2] uninitialized...
```



Let's return to example 1

- Assume an L1 data cache of 32kB with associativity of 8, and block size of 64B. The cache is initially empty.
- What happens when we execute the first line of the program?

```
a = 1;
```

```
b[0] = a+1;
```

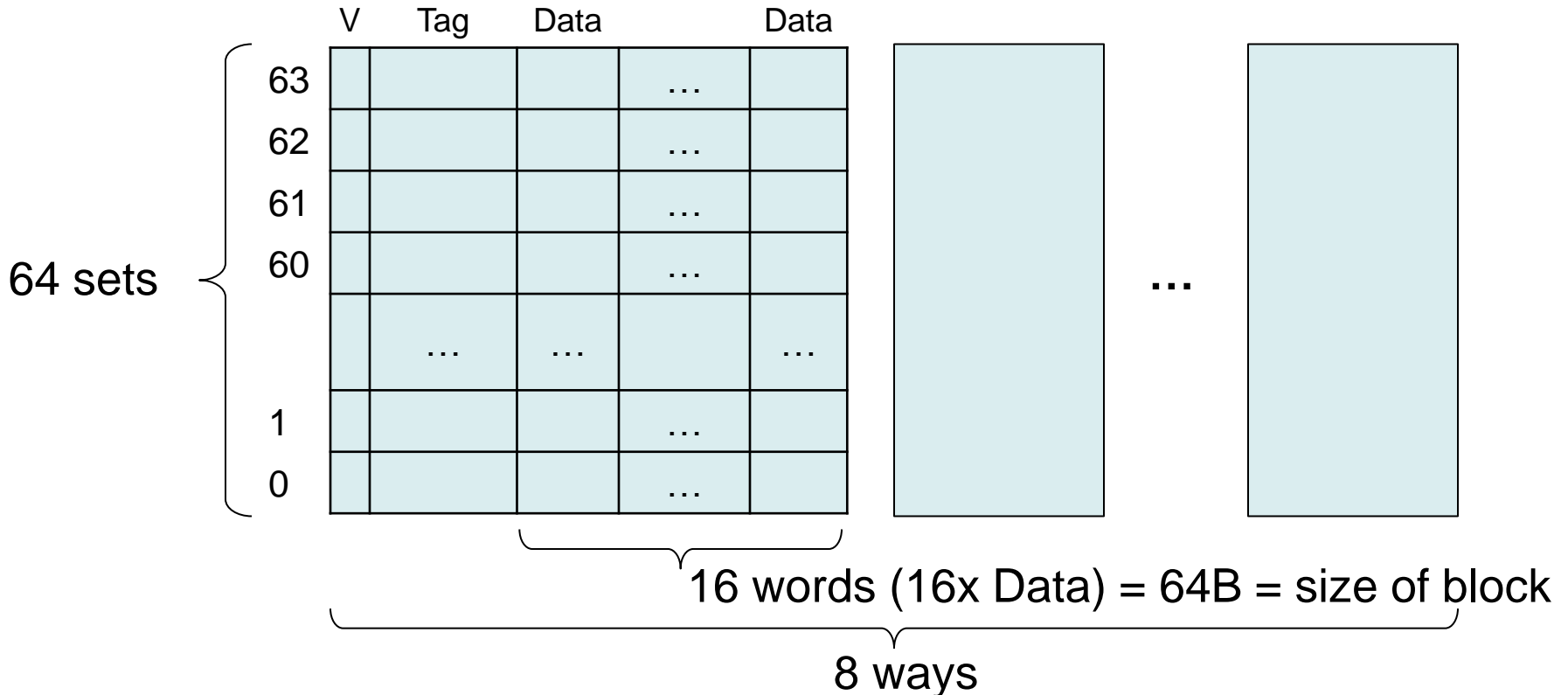
```
b[1] = b[0]+1;
```

```
d = b[2];
```

Let's return to example 1

- Assume an L1 data cache of 32kB with associativity of 8, and block size of 64B. The cache is initially empty.
- What happens when we execute the first line of the program?

a = 1;



Let's return to example 1

- Assume an L1 data cache of 32kB with degree of associativity = 8, and block size of 64B. The cache is initially empty.
- What happens when we execute the first line of the program?

a = 1; -> cache miss

way0

ATTENTION:
This is a Tag
from a
physical
address !!!

64 setu

	V	Tag	1111 Data	Data	Data	Data	Data	0011 Data	0010 Data	0001 Data	0000 Data	
63			...									
62			...									
61			...									
60	1	0x0028F	???	...	a	b[3]	b[2]	b[1]	b[0]	c	d	???
									
1			...									
0			...									

16 words (16x Data) = 64B

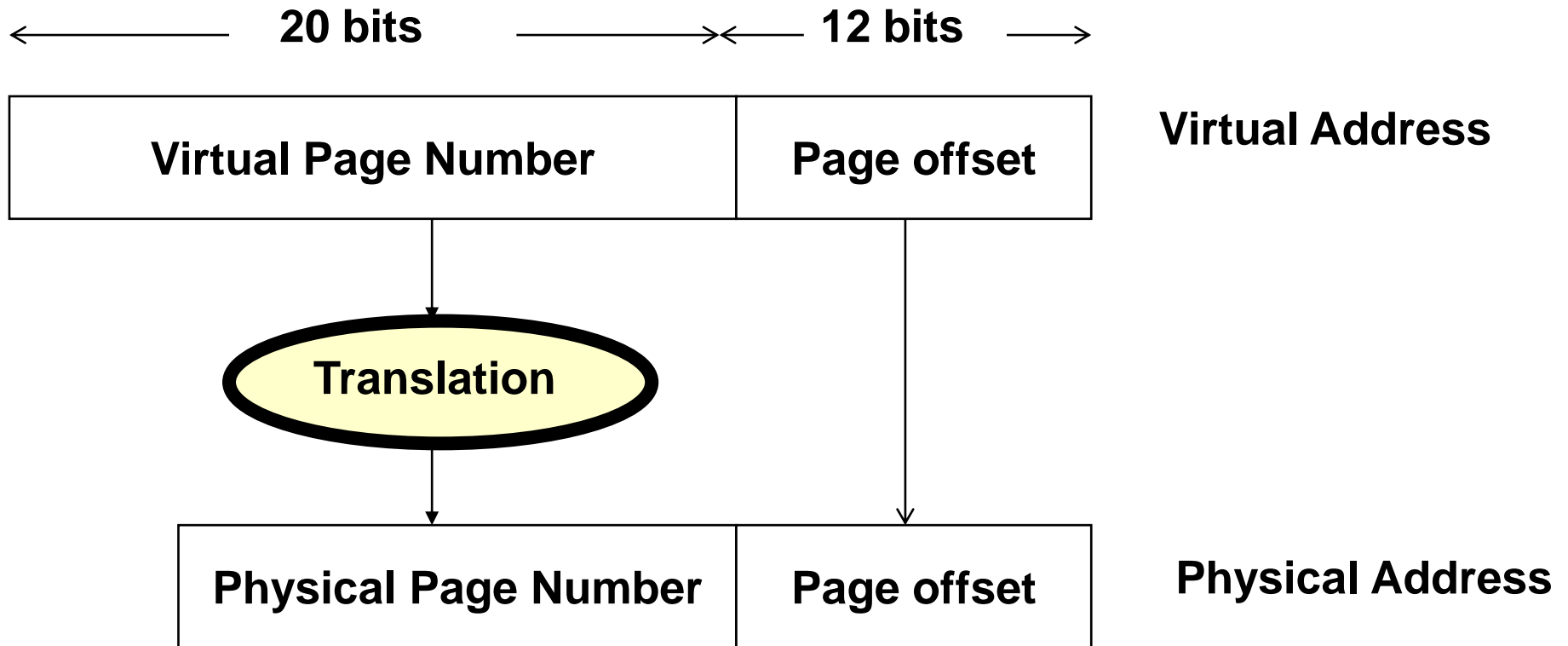
Let's return to example 1

- Paging (realization of virtual memory) does not interfere with the principle of spatial location => important for cache.
- Data on adjacent virtual addresses will be stored in physical memory side by side (of course if they do not cross the page boundary).
- If page fault occurs, i.e. the page is not in physical memory, then:
 1. An exception is handled by the OS.
 2. The page to be replaced is selected in the memory and stored on disk if needed.
 3. Then the new content of the requested page (from the disk) is read, or the zeroed page is mapped (in case of new memory).
 4. The cache row is also cached.
- Another cache miss inside the page no longer invokes the page fault until the page is replaced by another page.

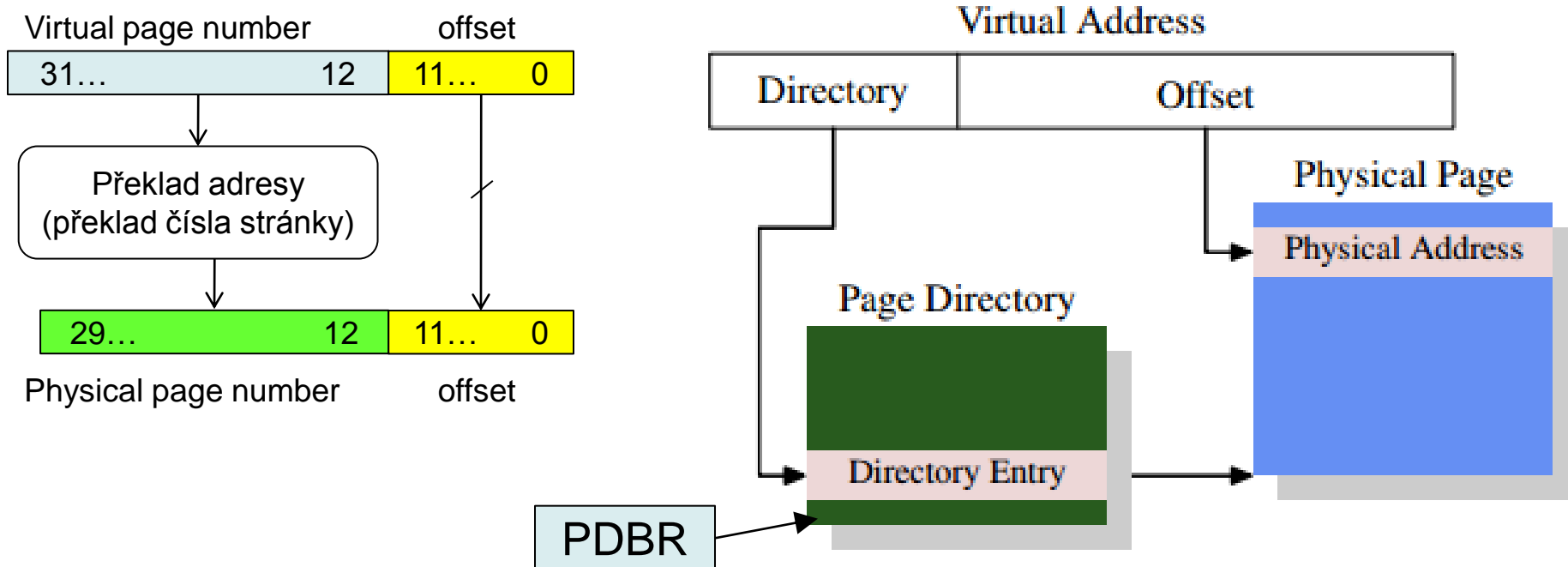
Address translation

- **Page Table**
 - Root pointer/page directory base register (x86 CR3=PDBR)
 - Page table directory PTD
 - Page table entries PTE
- Basic mapping unit is a page (page frame)
- Page is basic unit of data transfers between main memory and secondary storage
- Mapping is implemented as look-up table in most cases
- Address translation is realized by **Memory Management Unit (MMU)**
- Example follows on the next slide:

Address Translation



Single-level page table (MMU)?

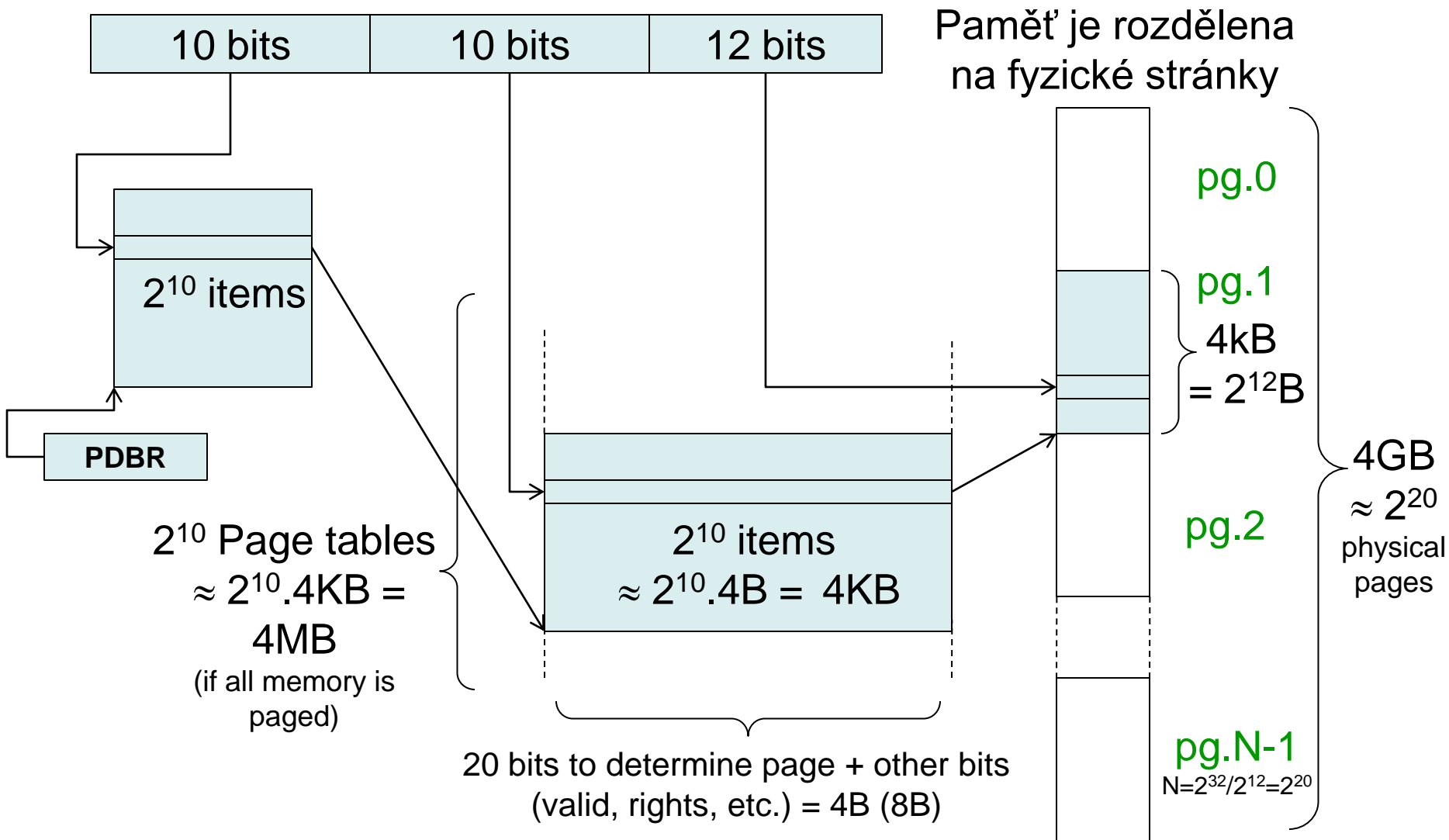


- Page directory is represented as data structure stored in main memory. OS task is to allocate physically continuous block of memory (for each process/memory context) and assign its start address to special CPU/MMU register.
- PDBR - page directory base register – for x86 register CR3 – holds physical address of page directory start, alternate names PTBR - page table base register – the same thing, page table root pointer URP, SRP on m68k

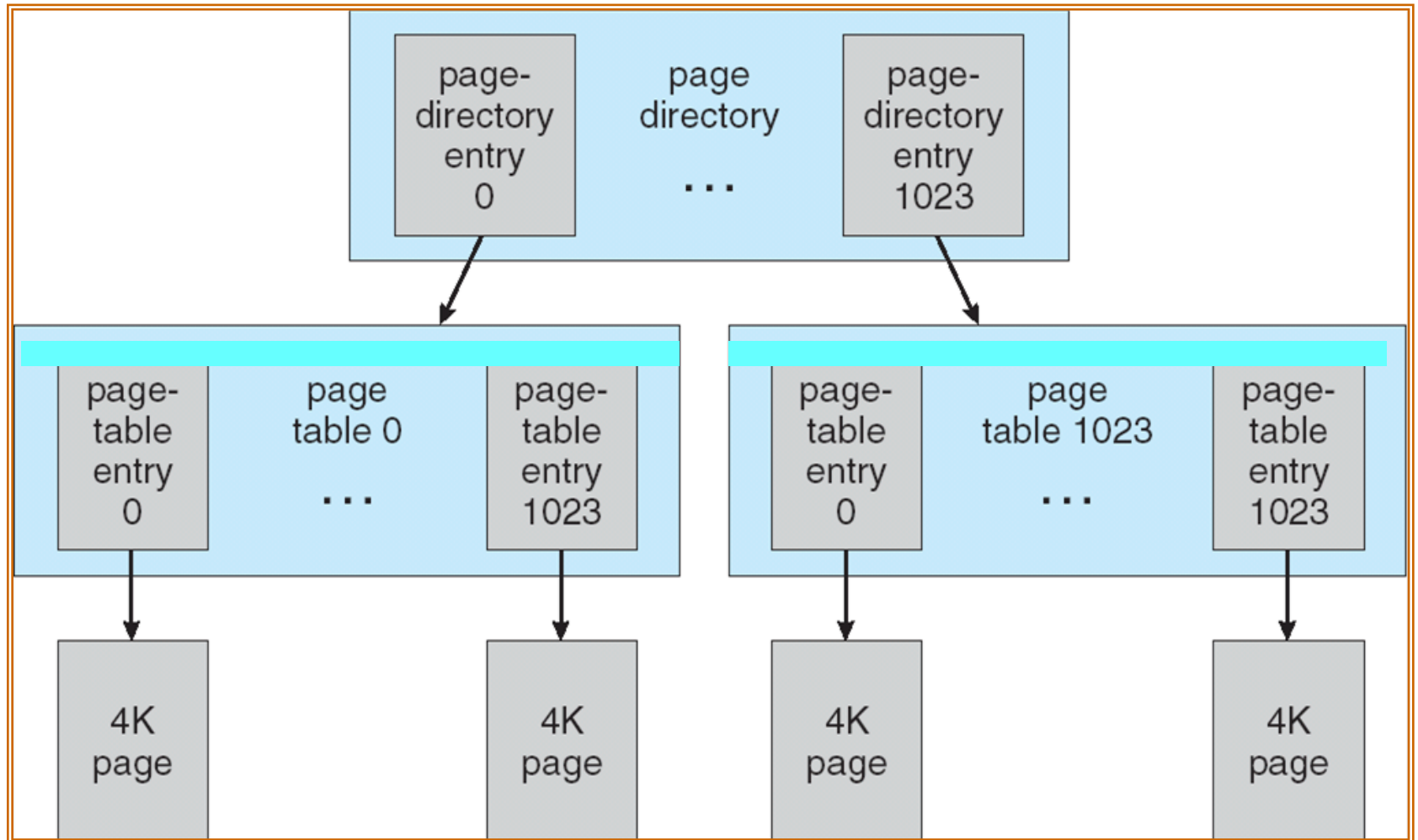
But consider memory consumed by page table

- Typical page size is 4 kB = 2^{12}
- 12 bits (offset) are enough to address data in page (frame). There are 20 bits left for address translation on 32-bit address/architecture.
- The fastest map/table look-up is indexing \Rightarrow use array structure
- The page directory is an array of 2^{20} entries (PTE). That is big overhead for processes that do not use whole virtual address range. There are another problems as well (physical space allocation fragmentation when large compact table is used for each process, etc.)
- Solution: multi-level page table – lower levels populated only for used address ranges.

Multi-level paging - 2 levels

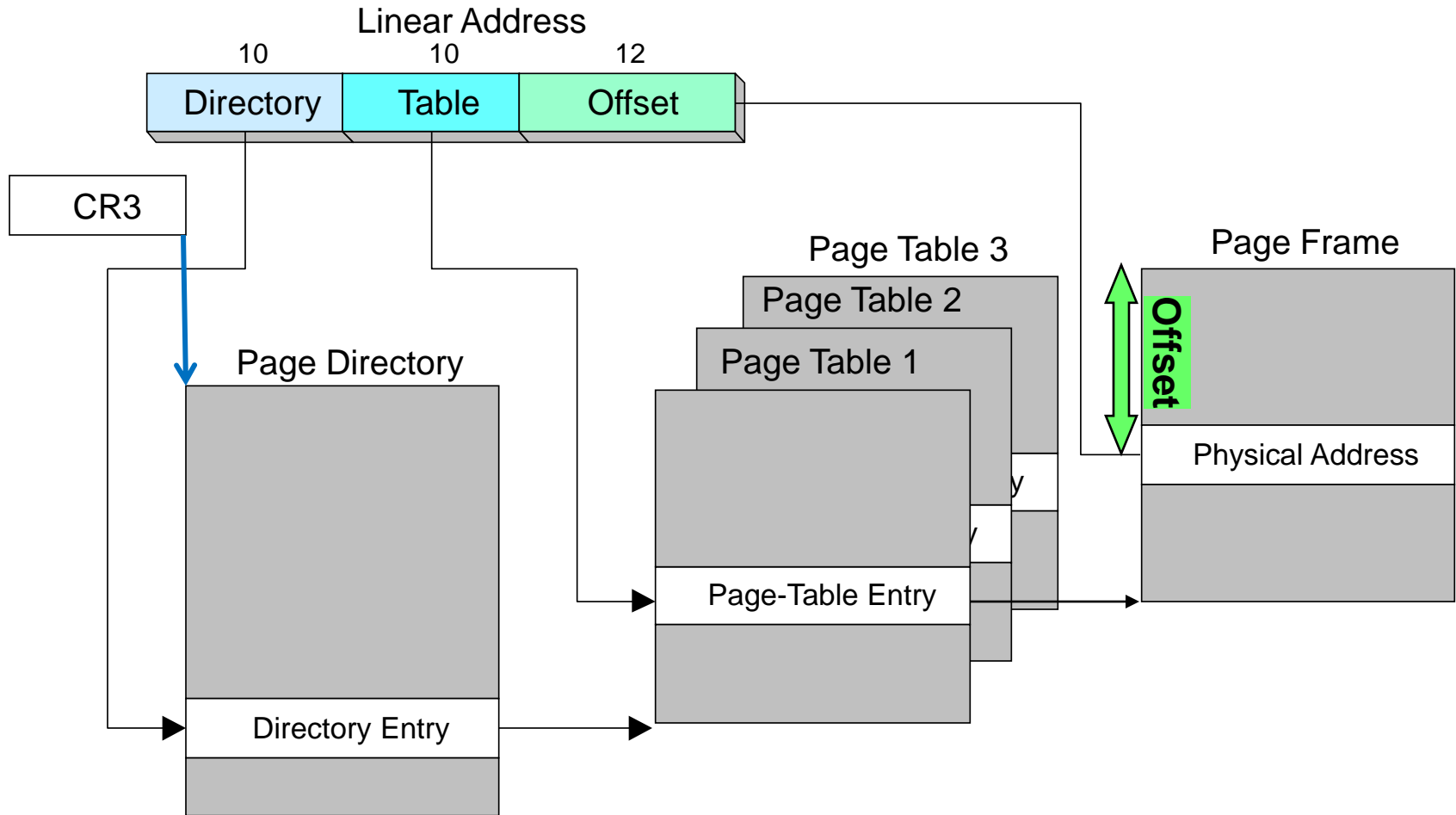


Principle of mapping 1/4 - drawn using tree structure



Principle of mapping 2/4

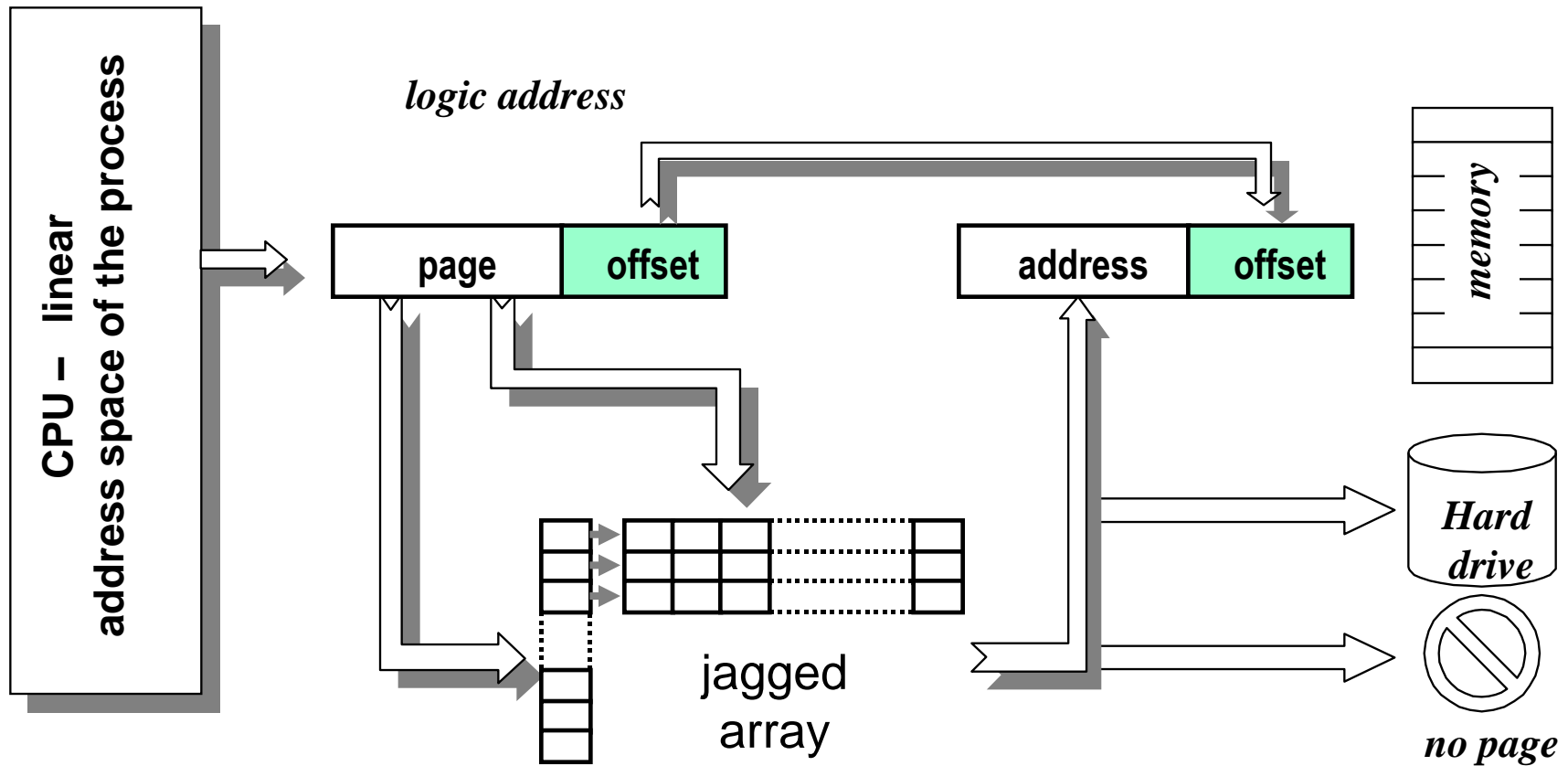
- decomposition of linear address into indexes



Mapping Principle 3/4

- Indexes to Mapping Matrix

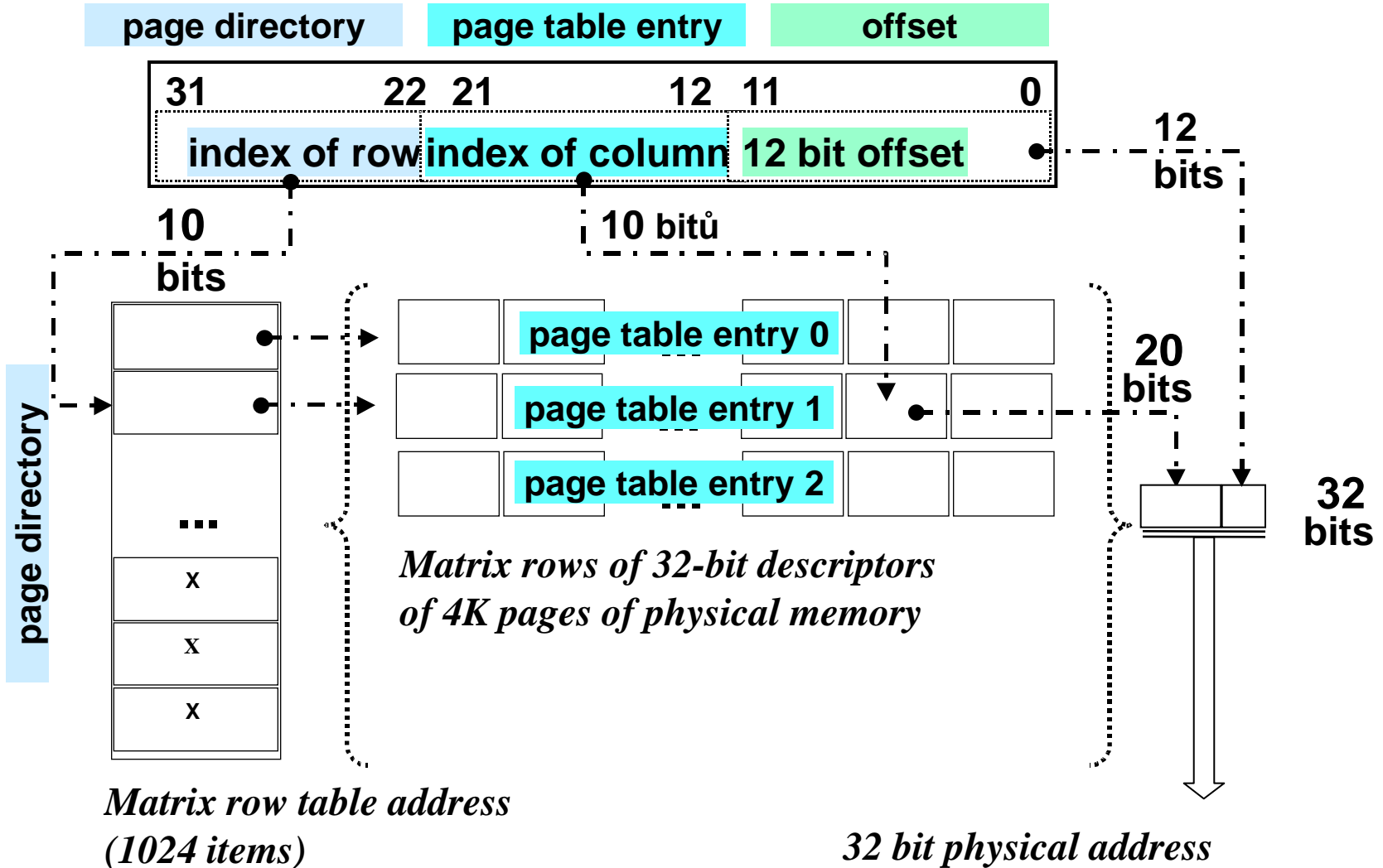
- The principle of converting a linear address to a physical address



Note: Windows and Linux OS allocate memory always by page (usually 4 KB), compare with "cluster" = disk allocation constant!

Mapping Principle 4/4 - Processor Operation

32-bit linear address



What is in page table entries?

VA – virtual address

Page Table Base Register
PTBR

Index into pagetable

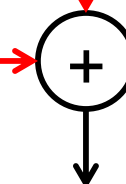
Page valid bit – if = 0,
page not in the memory
results in **page fault**

Page #	Offset
--------	--------

Look-up Table

Page table

V	Access rights	Frame#

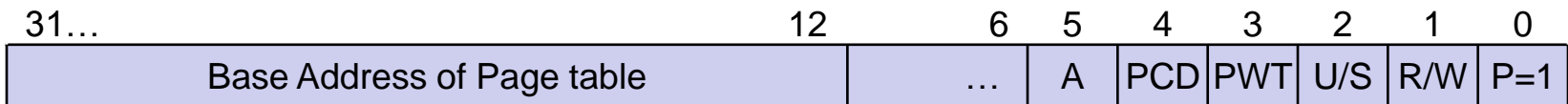
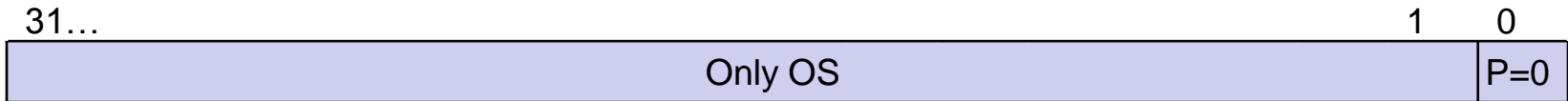


PA – physical address

Page table placed in physical memory

Page Table - How Do Items Look? Meaning of items...

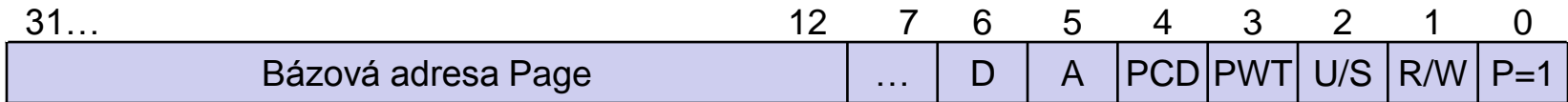
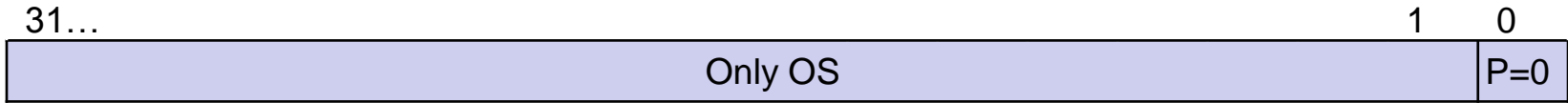
Let's look at the item **Page Directory** (Page Table similar)



- **P** -bit 0: Present bit – *determines whether the page is in memory (1) or on disk (0)* Sometimes this bit is called *V* - valid.
- **R/W** -bit 1: Read/Write: *if 1 then R/W; otherwise only read*
- **U/S** -bit 2: User/Supervisor: *1 – user access; 0 – only OS*
- **PWT** -bit 3: Write-through/Write-back – *writing strategy for the page*
- **PCD** -bit 4: Cache disabled/enabled – *some peripherals are mapped directly into memory (memory mapped I/O), allowing write / read to / from the periphery. These memory addresses are then I/O ports and they are not written into cache.*

Page Table - How Do Items Look? Meaning of items...

Let's look at the item Page Table (Page Table on 2nd level)



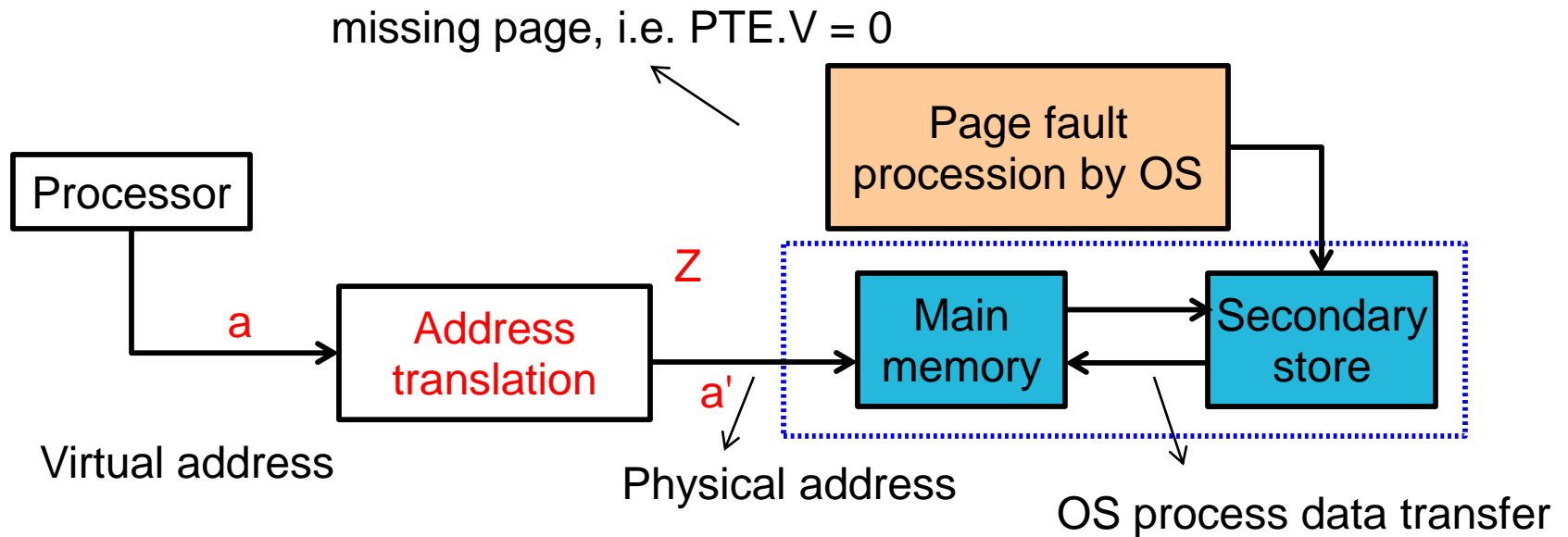
- **A** -bit 5: Accessed – *Whether we have read / written - helps decide which pages to remove when we need to free up memory space*
- **D** bit 6: Dirty bit – *it is set if we wrote into page.*
- 11..7 bit - special use, as memory type, or when to update cache, etc.
- 31-12 bit - Physical Page Address

Notes

- Each process has a Page Table, that is, its PTBR (base register) value.
- This, by the way, ensures the memory security of processes.
- What do we want you to remember from the Page Table Item Format?
- V – Validity Bit. V=0 Page is not valid (stored of HD).
 - AR – Access Rights. (Read Only, Read/Write, Executable, etc.),
 - Frame# - frame number (base address into lower level),
 - Other as modified/Dirty, and so on.



Virtual memory – Hardware and software interaction



What to do on Page Fault?

If memory is low

- Using LRU, we find the pages that can be released.
- If they have set dirty bits, we write them "somehow" (usually by DMA, Direct Memory Access, direct memory access) to disk.
- The Process Page Table is updated to allow free memory.

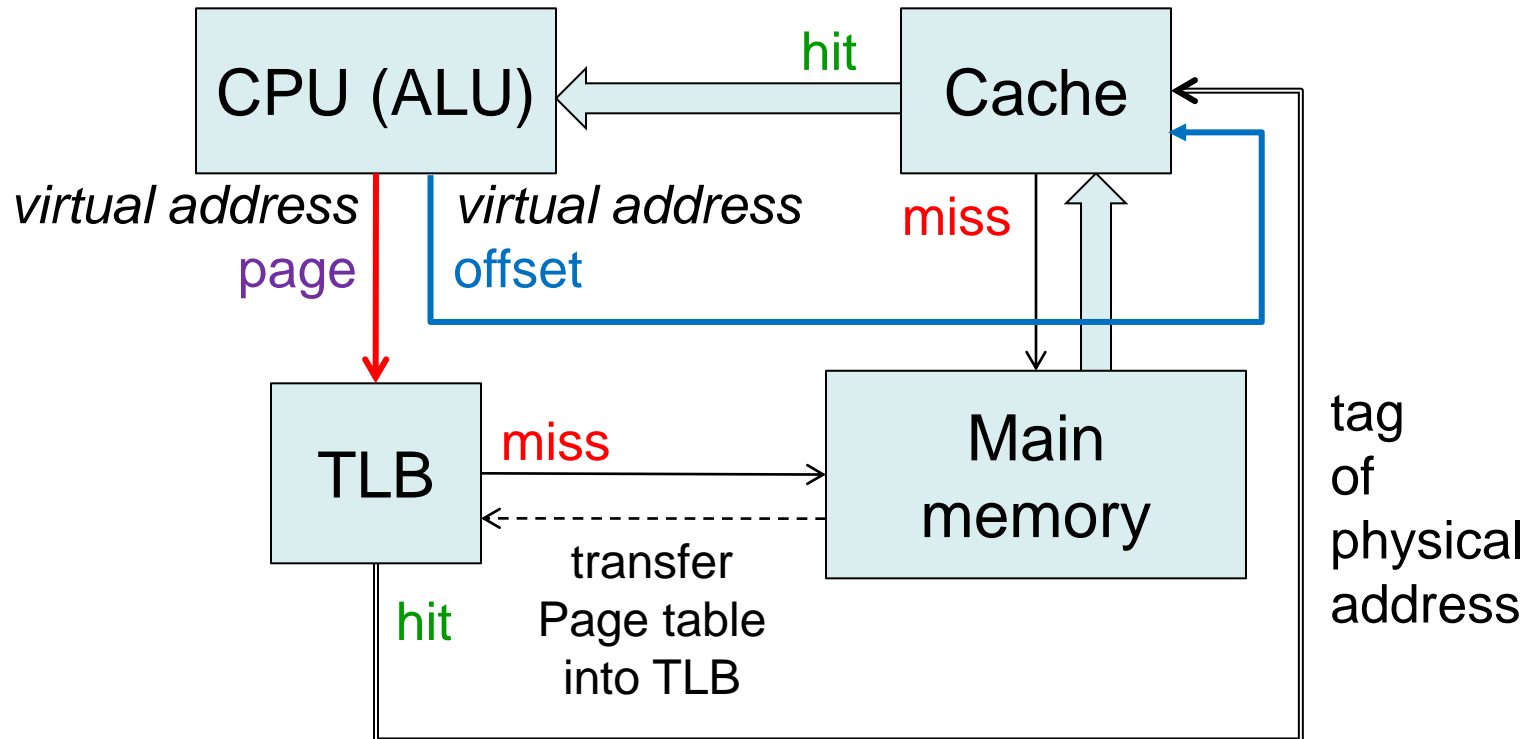
Physical memory is free,

but our data are in a secondary memory (on the disk)..

- The requested page is loaded (by DMA) into an empty frame.
- If the page is not on the disk, i.e. empty memory allocation, and it is not requested by the kernel, then it must be completely cleared (for security reasons).
- When DMA transmission is complete, an interrupt is invoked, the Process Page Table is updated.

During the paging process, you can switch to another pending process that can continue until the operation is finished.

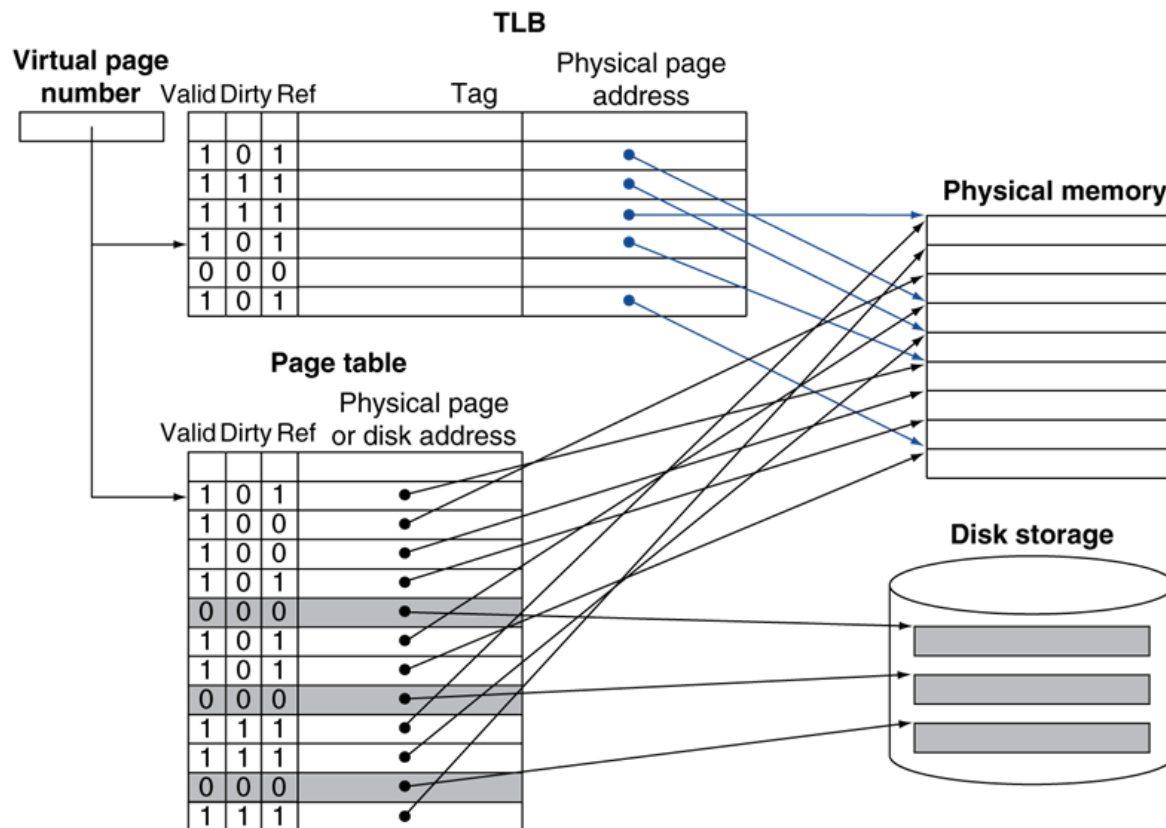
TLB-idealized address translation - reading



- Note that there may be 2x miss
- If a TLB miss occurs, we must execute a *page walk*.

Fast MMU/address translation using TLB

- Translation-Lookaside Buffer, or may it be, more descriptive name – Translation-Cache
- Cache of frame numbers where key is page virtual addresses



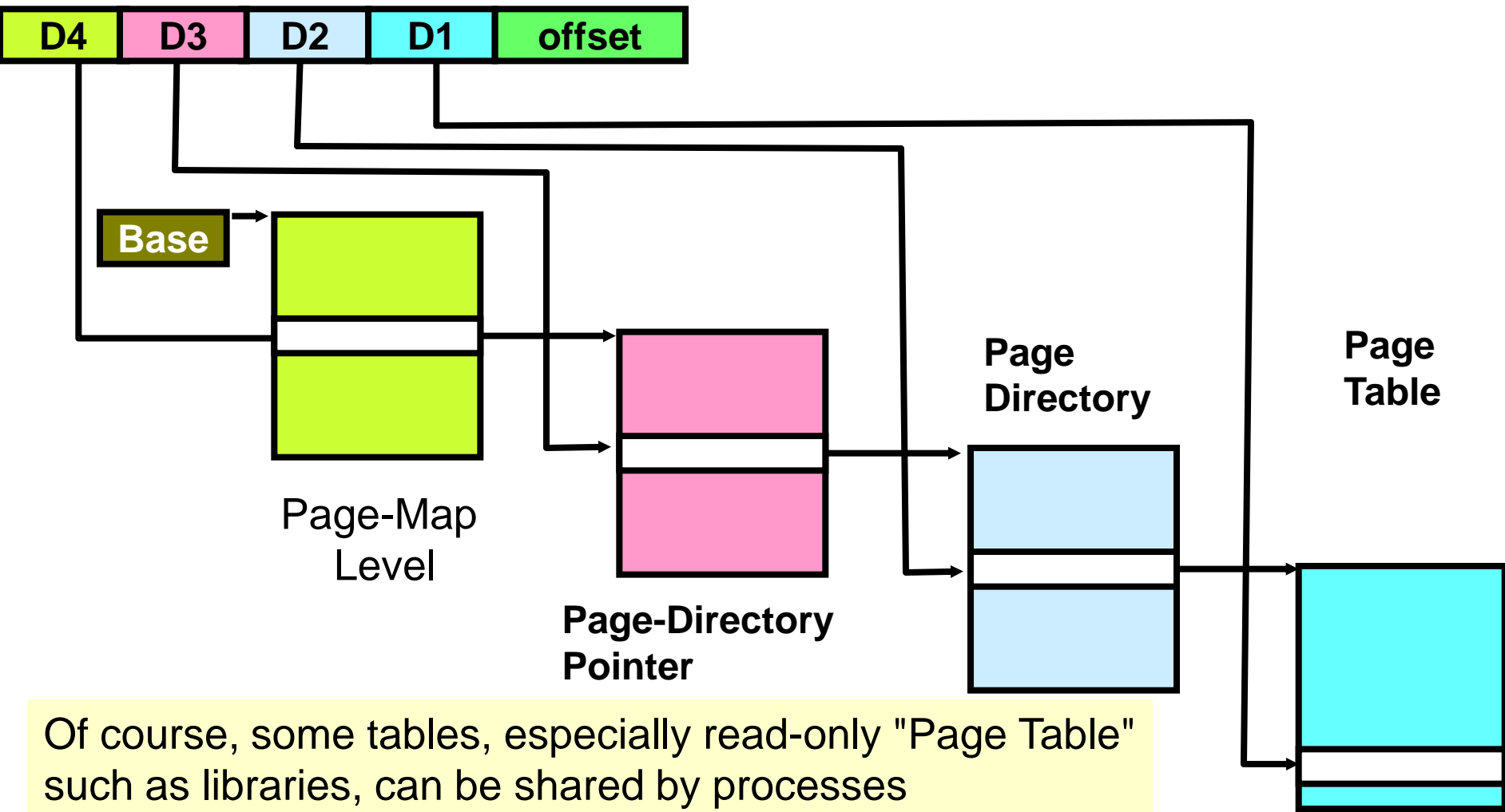
*Paging on 64-bit processors

On 64 bit systems

1. It is possible to increase the page length from 4kB to 1GB, but impractical.
2. You can use multi-level paging tables, add
 - *Page-Directory Pointer Table* (Win64: from 4 to 512 address)
 - *Page-Map Level* (Win64: up to 512 adrees)

Also, the descriptor length in all tables is increased from 32 bits to 64 bits

Multi-level tables for 64-bit processors



Multi-level paging

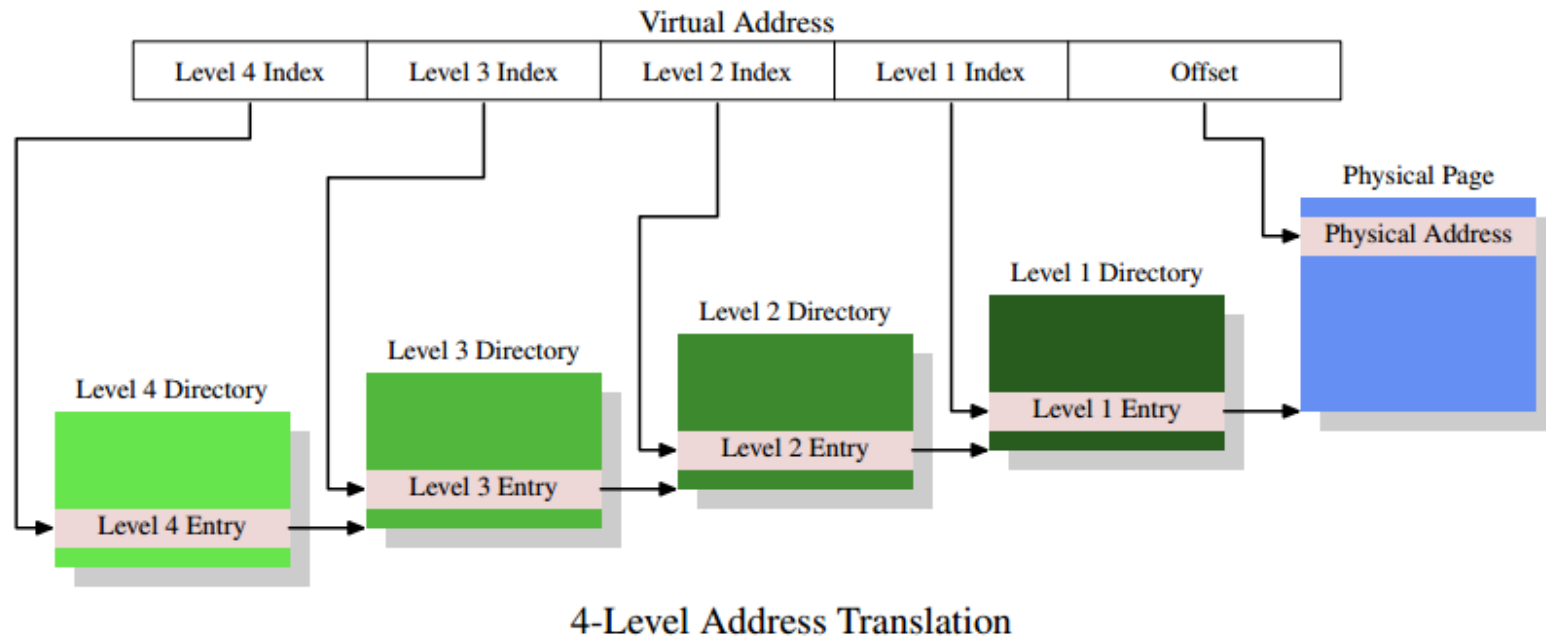
Notes on the previous slide :

- Not every process uses its entire address space => is not necessary to allocate in the second level 2^{10} Page tables
- Page tables can also be paged

General notes :

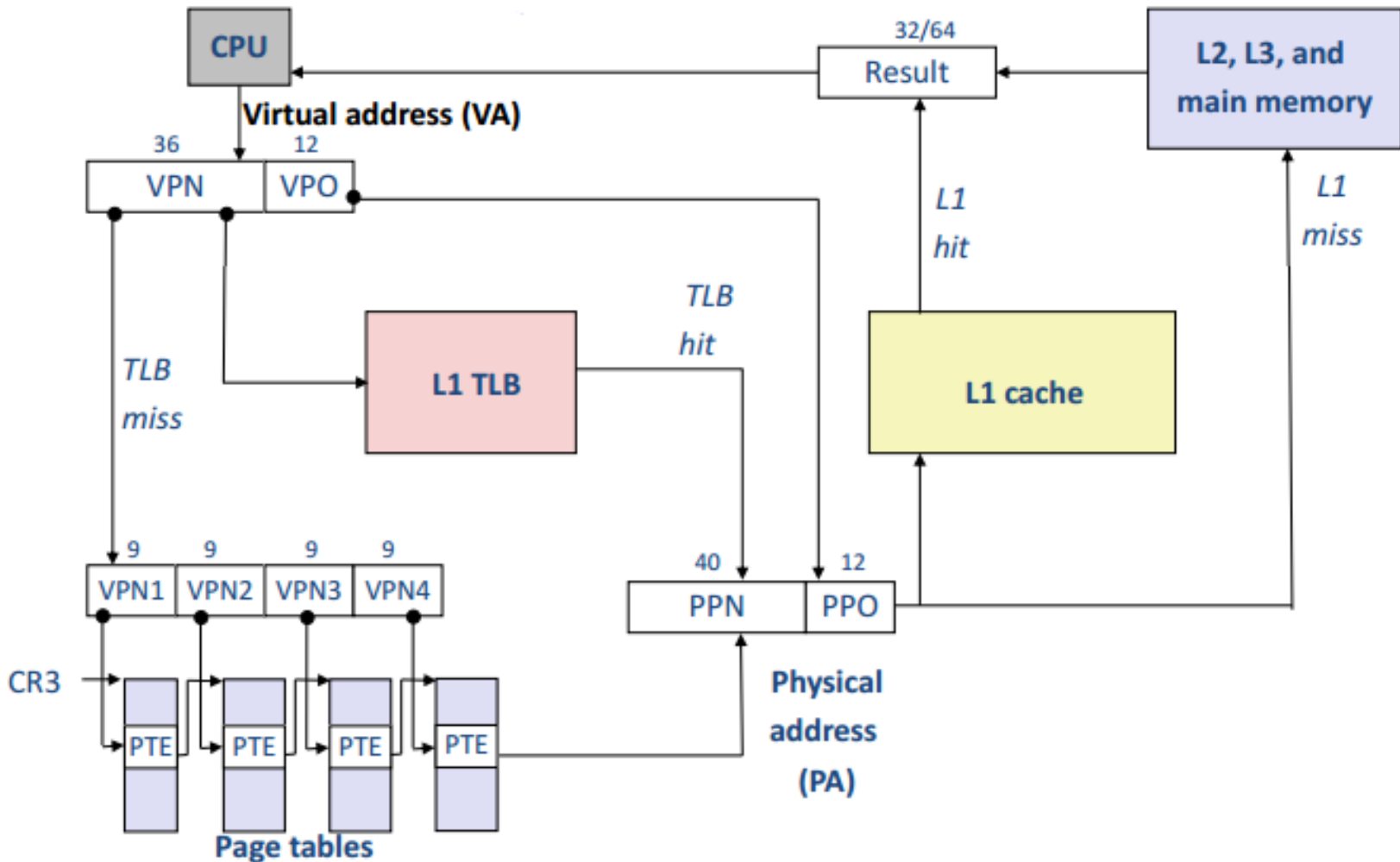
- Intel IA32 implements 2-level paging
 - Page Table on level 1 is Page Directory (10 bits of address)
 - Page Table on level 2 is Page Table (next 10 bits of address)
- In the case of a 64-bit virtual address, it is customary to use fewer bits for a physical address - for example, 48, or 40.
- Intel Core i7 uses 4-level paging and 48 bit address space
 - Page Table level 1: Page global directory (9 bits)
 - Page Table level 2: Page upper directory (9 bits)
 - Page Table level 3: Page middle directory (9 bits)
 - Page Table level 4: Page table (9 bits)

Multi-level page table – translation overhead



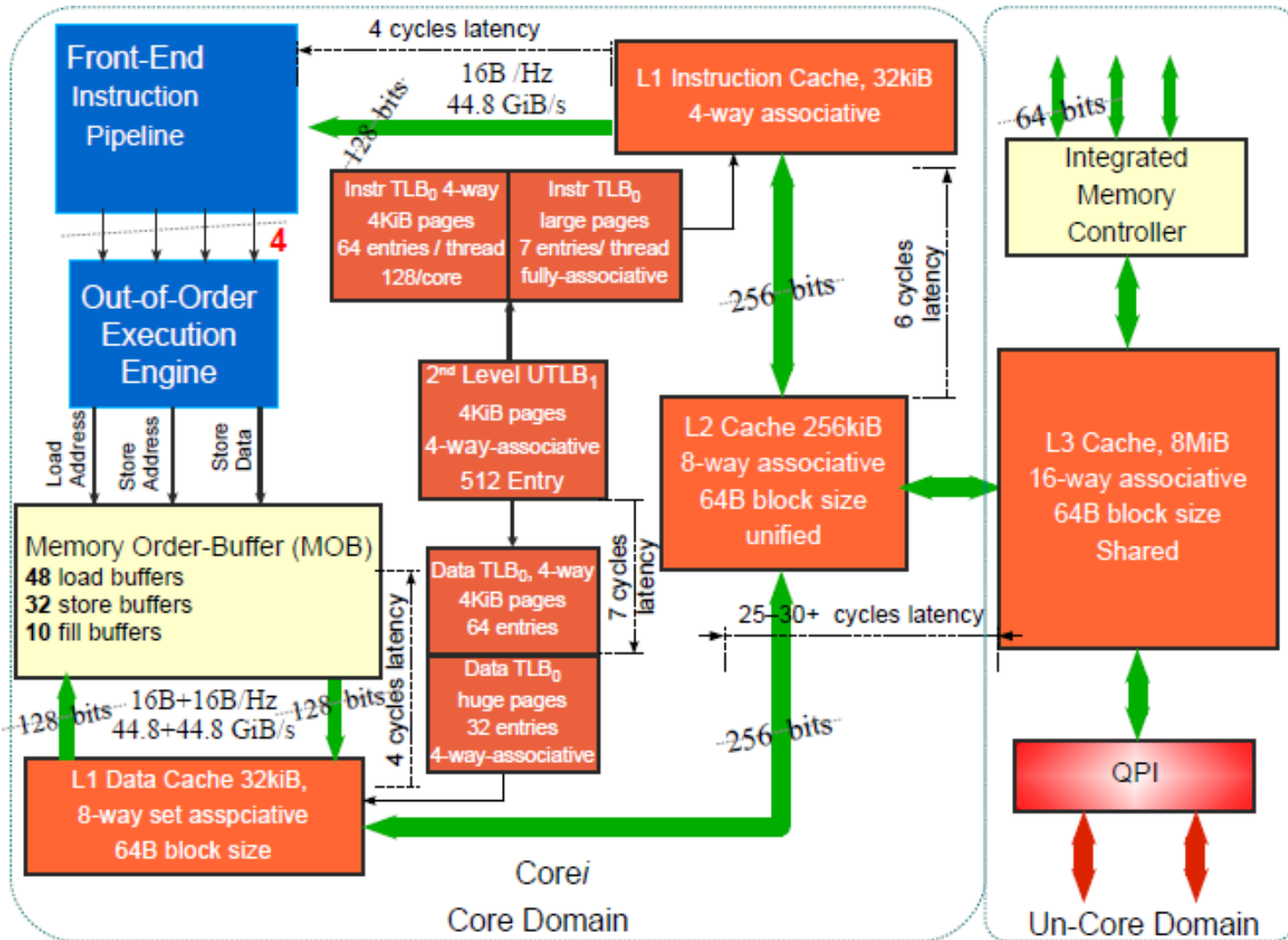
- Translation would take long time, even if entries for all levels were present in cache. (One access per level, they cannot be done in parallel.)
- The solution is to cache found/computed physical addresses
- Such cache is labeled as Translation Look-Aside Buffer
- Even multi-level translation caching are in use today

Paging – Intel Nehalem (Core i7)



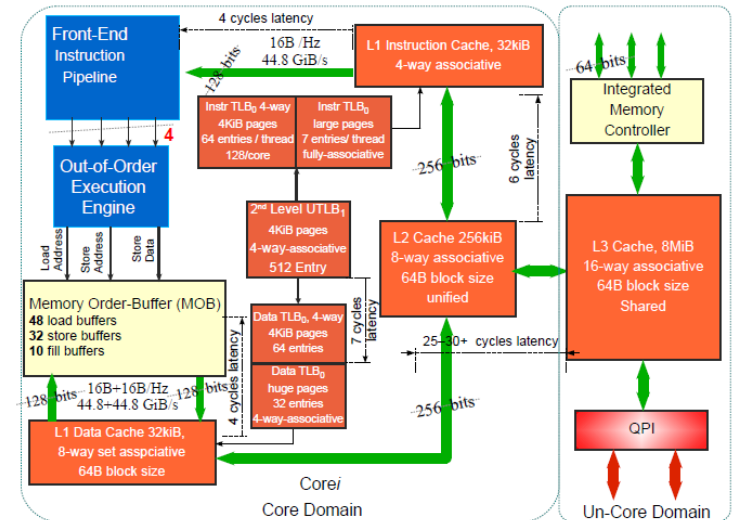
<http://cs.nyu.edu/courses/spring13/CSCI-UA.0201-003/lecture18.pdf>

Memory management - Intel Nehalem (Core i7)



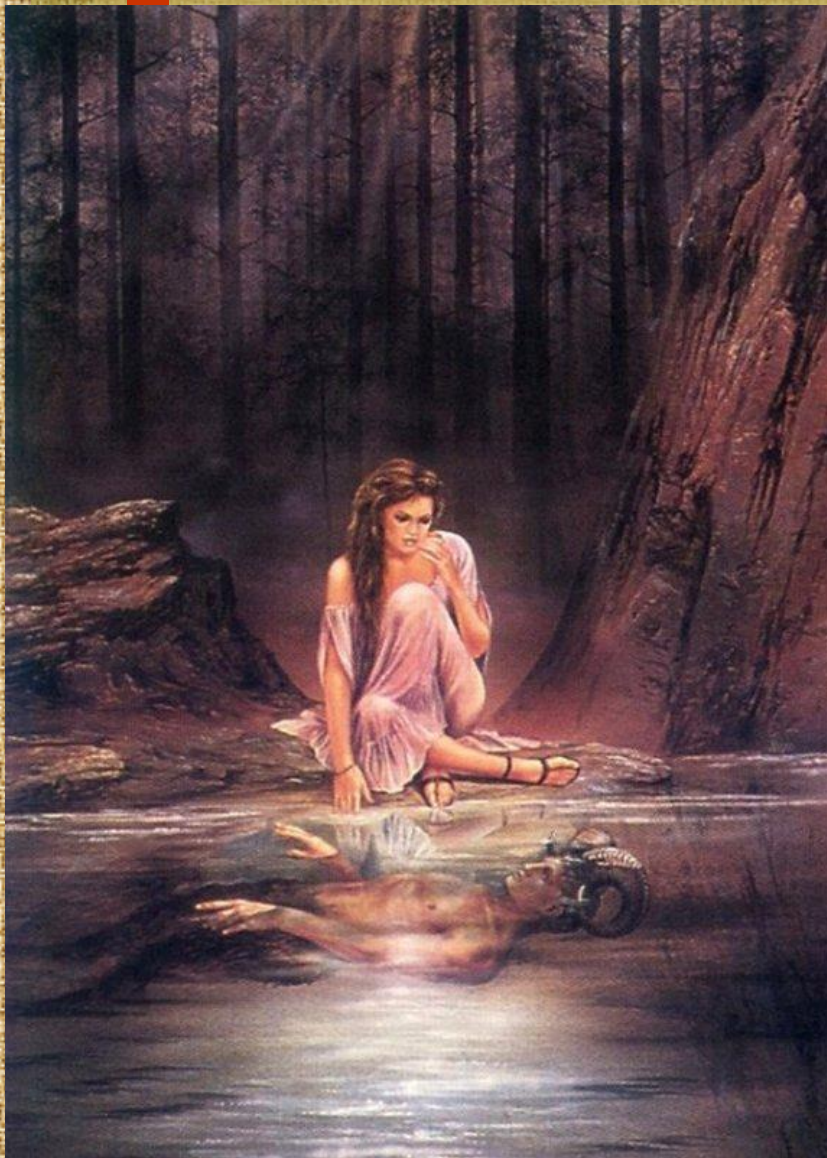
Memory management - Intel Nehalem – notes

- Block Size: 64B
- Processor always reads line cache from system memory aligned to 64B (6 LSB of addresses are zeros) and does not support partially filled lines
- L1 - Harvard. In SMT shared by both threads, Instruction - 4-way, Data 8-way.
- L2 - unified, 8-way, non-inclusive, WB
- L3 - unified, 16-way, inclusive (L1 or L2 included in L3), WriteBack
- Store Buffers - temporarily store data for each listing. Needless to wait for writing to cache or memory. They ensure that writes are in the correct order and also when needed: - exception, interrupt, serialization instruction, lock, ..
- You may also notice separate TLBs (Translation Lookaside Buffer)



Typical values

	L1	Paged memories	TLB
Size in blocks	256-4k	16 000-250 000 000	40-1024
Size in bytes	16-64 kB	500 - 1 TB	0,25-16 KB
Size of block in bytes	16-64	4k-64k	4-32
Miss penalty (clock cycles)	10-25	10M-100M	100-1000
Miss rates	2 % - 5 %	0,00001-0,0001%	0,01-2 %



[Royo]

Paging and memory fragmentation

One death is a tragedy. A million deaths is just a statistic.

[Joseph Stalin]

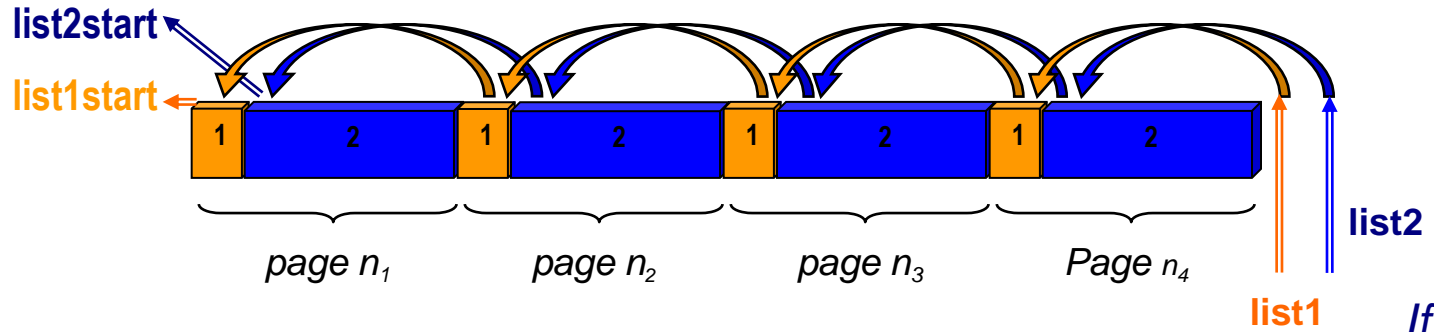


■ Reasons of fragmentations

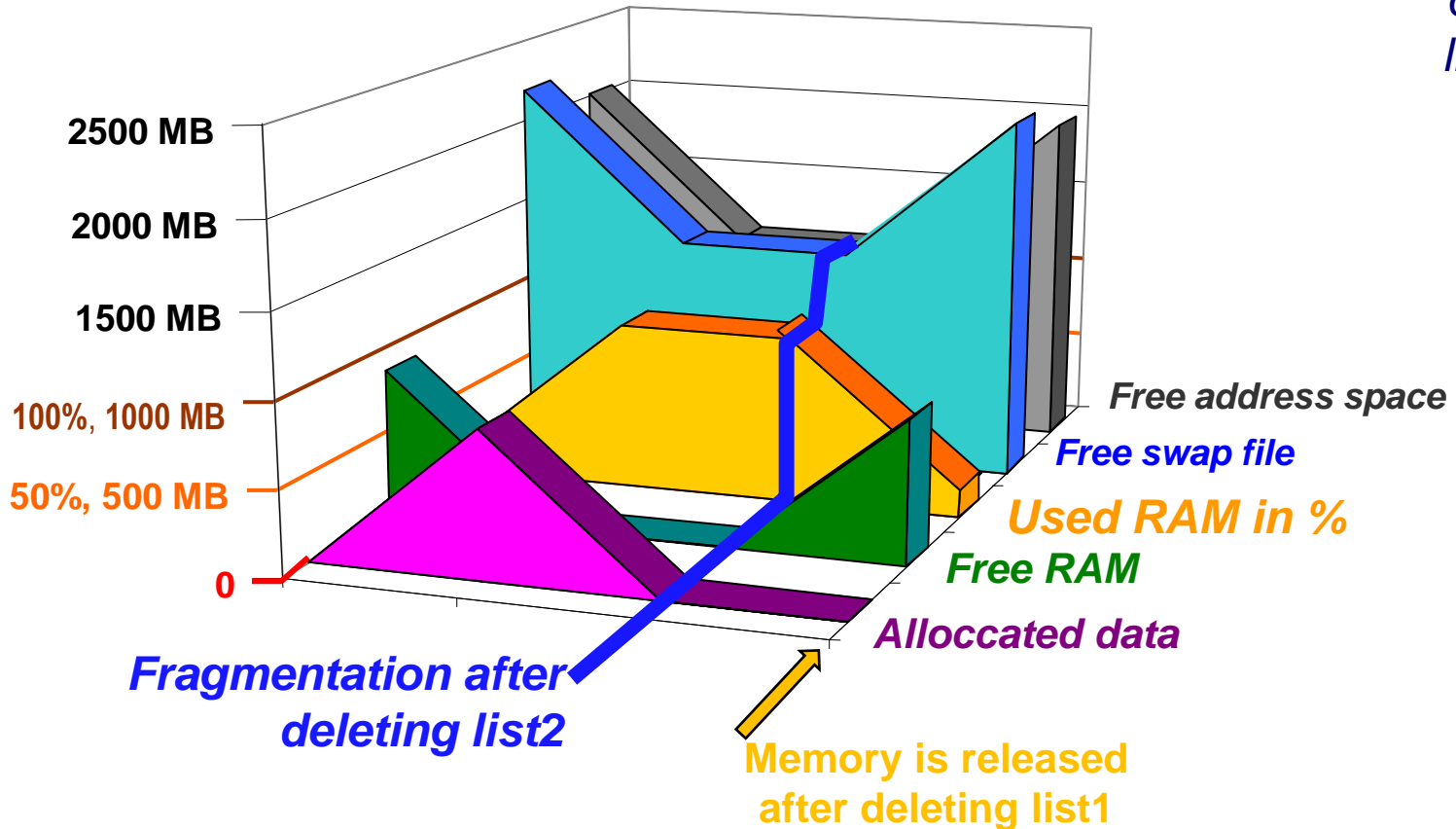
- a) variable program behavior
- b) the program requests large blocks of data but releases only small ones
- c) single death
 - Data that are not adjacent to each other are dropped, so their memory areas cannot be recombined and used for large objects.



Example of Fragmentation in C++



If we delete list2...!



More efficient use of memory - a way to accelerate programs

Your program may consider page size and use memory more efficiently - by aligning allocations to multiple page sizes and then reducing internal and external page fragmentation .. (allocation order, etc. See also memory pool)

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
    printf(„Velikost stranky je: %ld B.\n“,
           sysconf(_SC_PAGESIZE));
    return 0;
}
```

Allocation of block aligned in memory:

```
void * memalign(size_t size, int boundary)
void * valloc(size_t size)
```

windows

```
#include <stdio.h>
#include <windows.h>

int main(void) {
    SYSTEM_INFO s;
    GetSystemInfo(&s);
    printf("Size of page is: %ld B.\n",
        ns.dwPageSize);
    printf("Address space for application:
    0x%lx - 0x%lx\n",
        s.lpMinimumApplicationAddress,
        s.lpMaximumApplicationAddress);
    return 0;
}
```