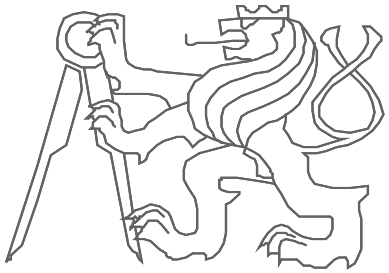
Architektury počítačů

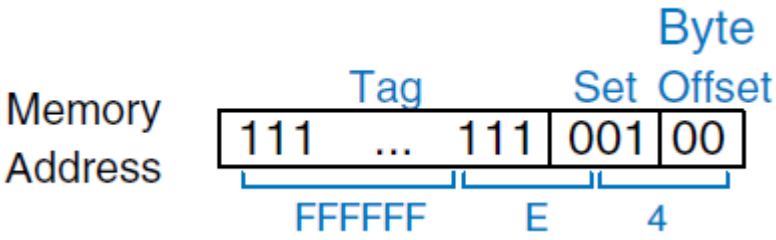
Virtuální paměť



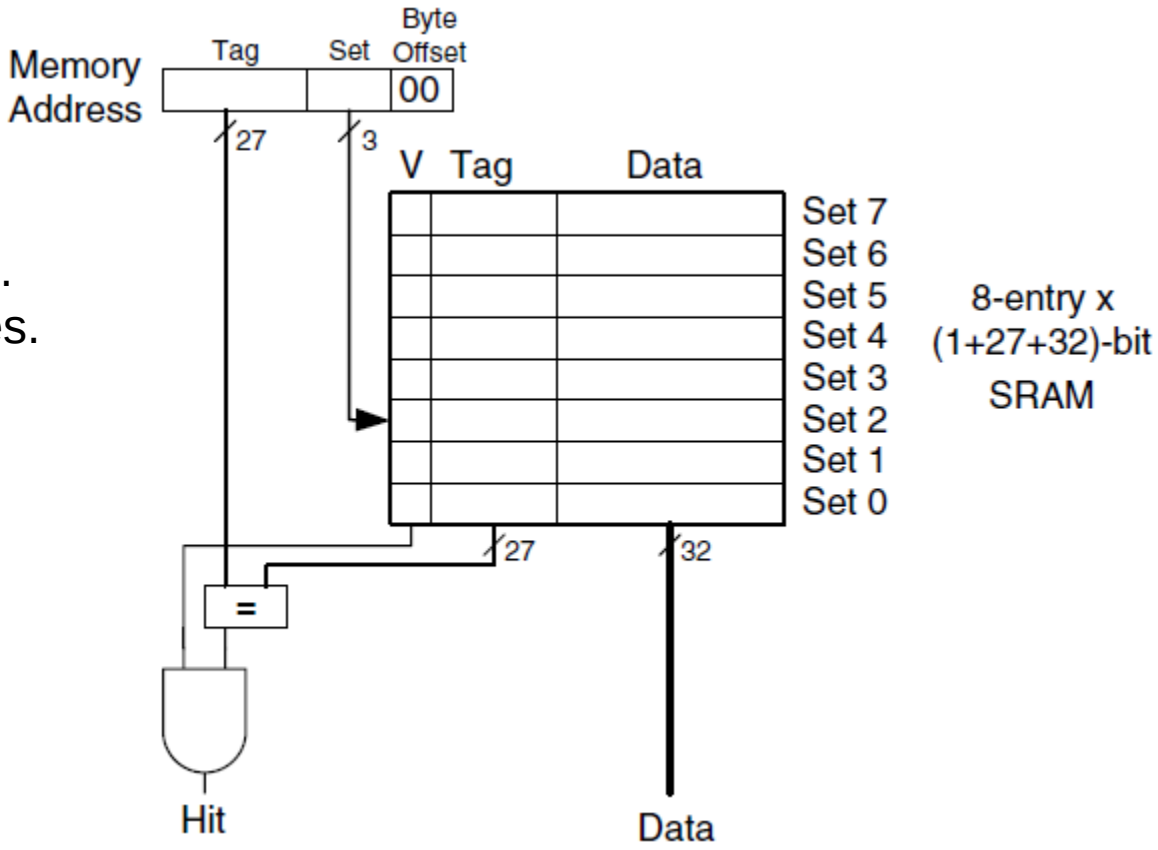
České vysoké učení technické, Fakulta elektrotechnická

*Na minulé přednášce…

Přímo mapovaná cache



přímo mapovaná cache:
one block 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.

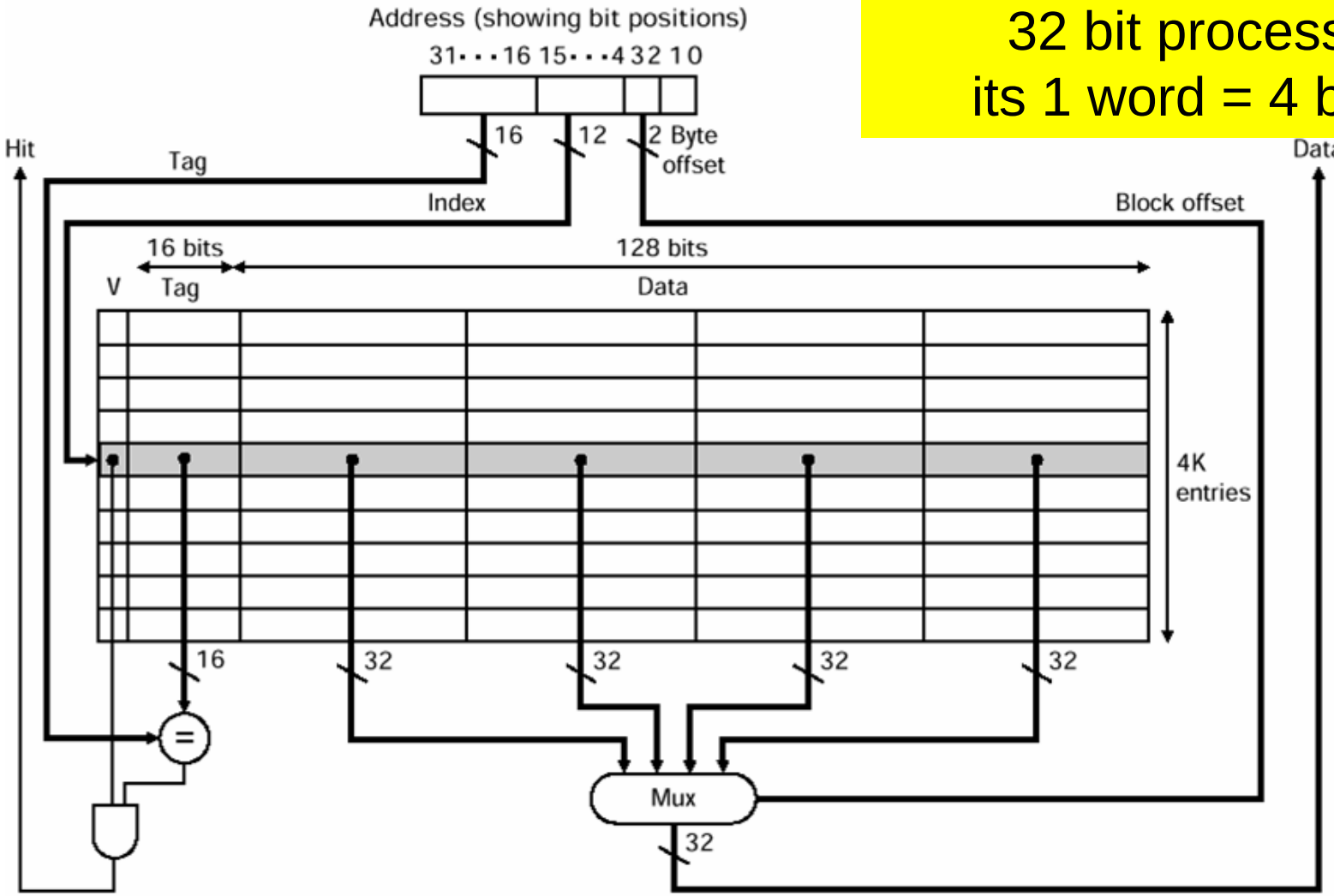
Degree of associativity – N

(rows in set)

for direct mapped cache N=1

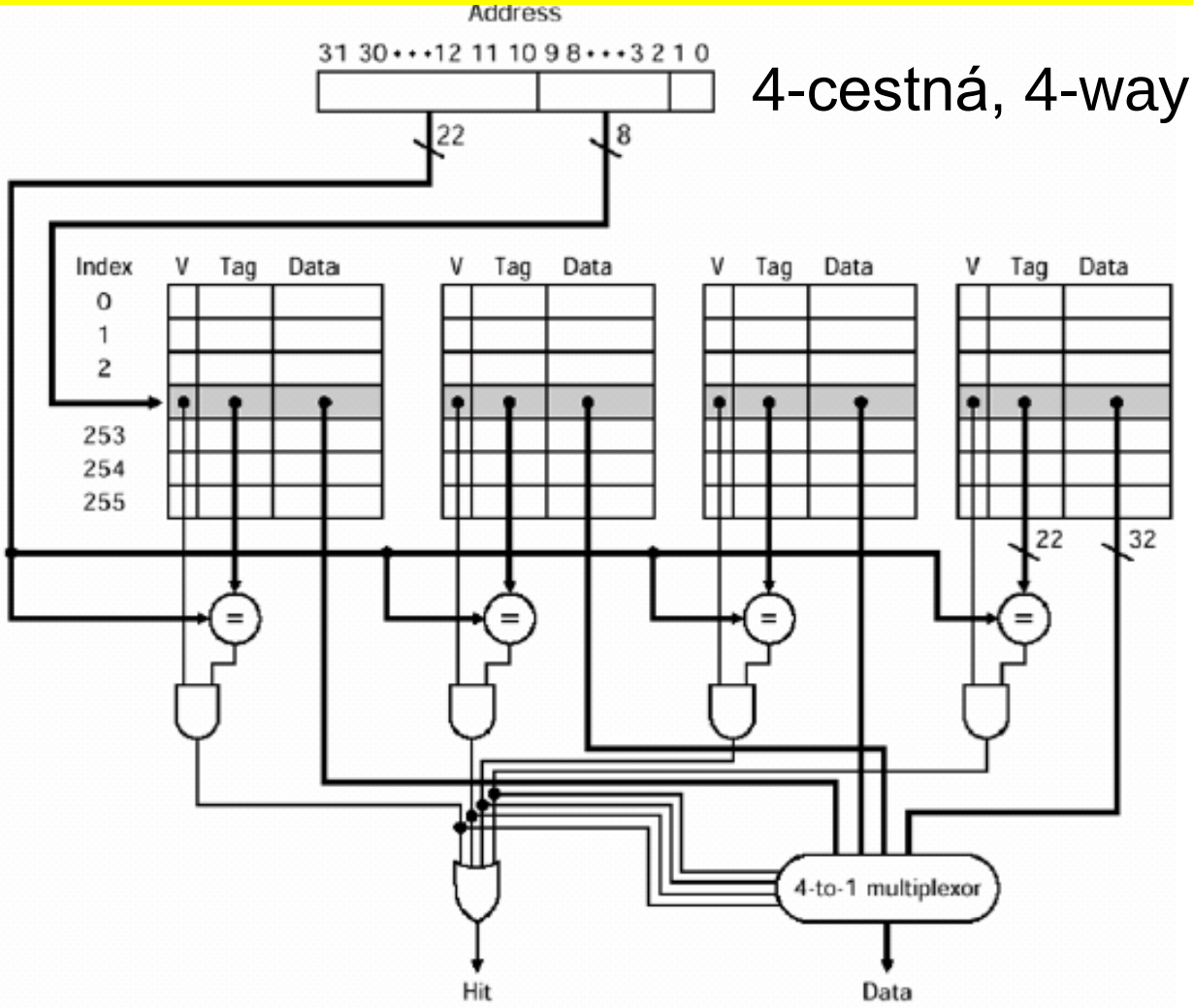
Direct mapped cache implementation

32 bit processor,
its 1 word = 4 bytes



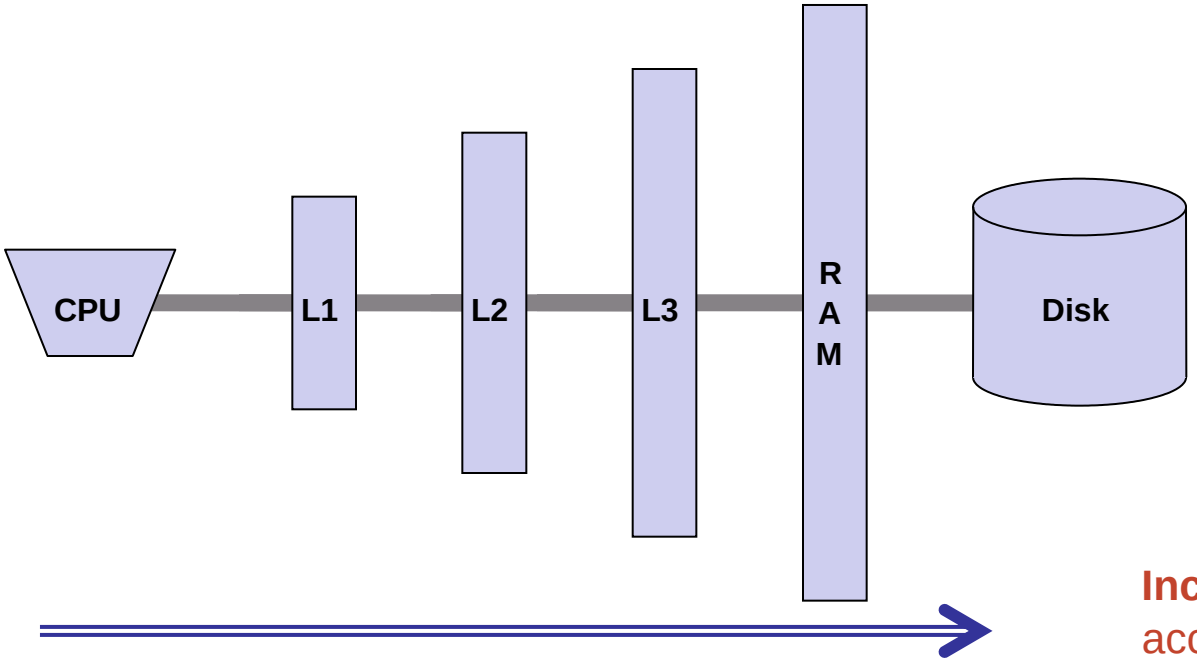
Cache s omezeným stupněm asociativity N=4

32 bit processor 1 word = 4 bytes



4-cestná, 4-way cache

Memory Hierarchy



Increasing
access time
and space

Otázka: Který třídicí program je lepší?

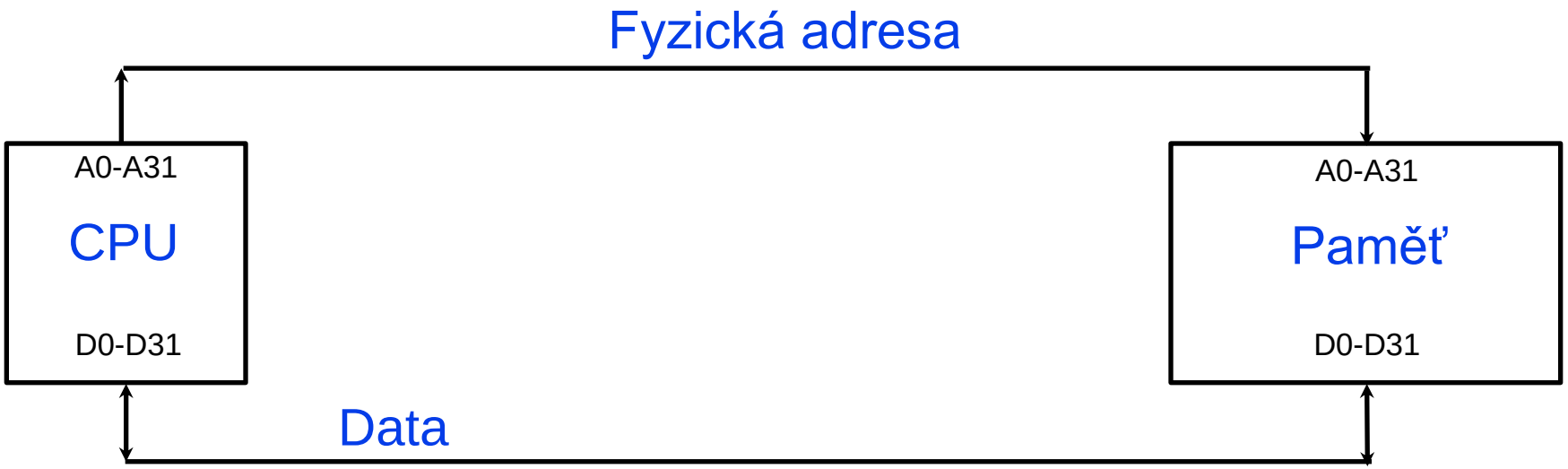
Předpokládejme, že jsme dostali následující výsledky v QtMips,
pro nějaké nepojmenované třídicí programy,
a k tomu doby přístupu *(zvolené tak, aby se nám to dobře počítalo):*
miss = 1 cyklus hodin cache + 9 cyklů hodin paměť
hit = 1 cyklus hodin cache

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

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

Dnešní téma…
Virtuální paměť

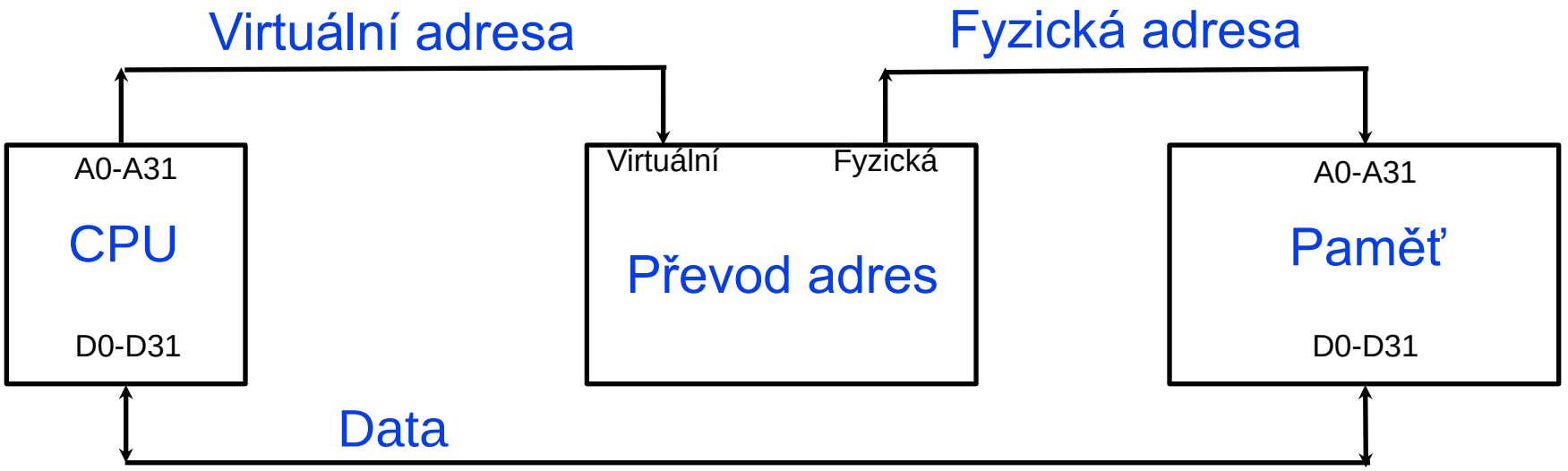
Fyzická adresa přímo do paměti?



Motivace k virtuální paměti..

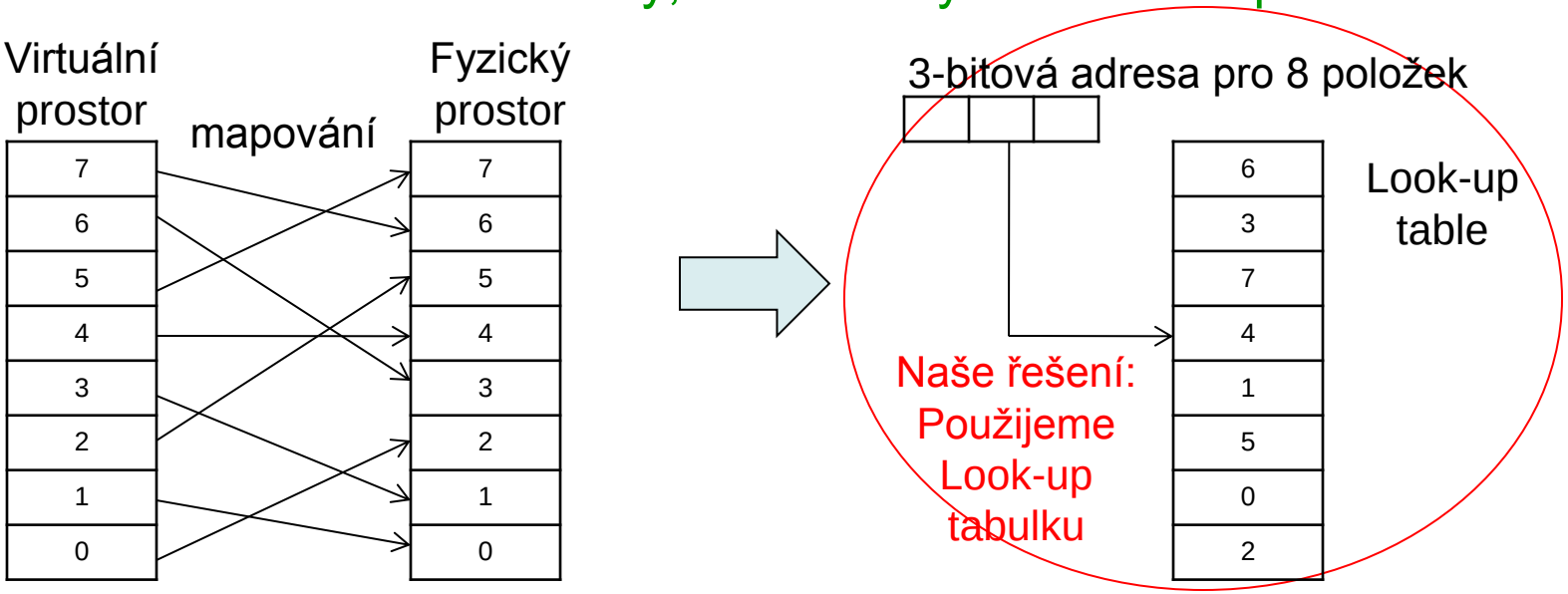
- **Běžně máme na počítači spuštěno několik desítek/stovek procesů…**
- **Umíte si představit situaci, kdy bychom rozdělili fyzikou paměť (například 1 GB) mezi tyto procesy? Jak veliký kus paměti by pak patřil jednomu procesu? Jak bychom řešili kolize – kdy nějaký program úmyslně (například virus) nebo neúmyslně (chybou programátora – práce s ukazateli) by chtěl zapisovat do kusu paměti, který jsme vyhradili jinému procesu?**
- **Řešením je právě virtuální paměť…**
- **Každému procesu vytvoříme iluzi, že celá paměť je pouze jeho a může se v ní libovolně zcela bezpečně pohybovat.**
- **Dokonce každému procesu dále vytvoříme iluzi, že má k dispozici např. 4GB paměti i když je fyzická paměť mnohem menší. Proces pak nerozlišuje mezi fyzickou pamětí a diskem (disk se mu jeví jako paměť).**
- **Základní idea: Proces adresuje ve virtuální paměti pomocí virtuálních adres. Ty pak musíme nějak přeložit na adresy fyzické.**

Virtuální a fyzické adresování



Motivace k virtuální paměti..

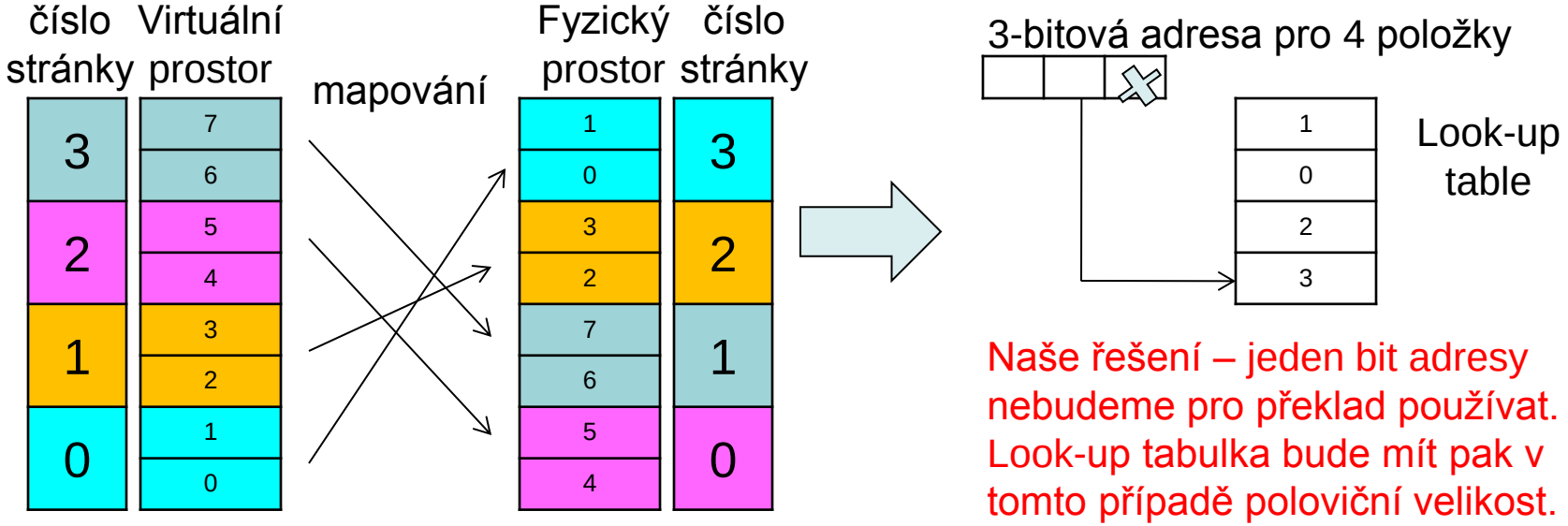
- Představme si, že máme 8B (Bajtů) virtuální prostor a 8B fyzické paměti…
- **Jak zabezpečíme překlad adres?** Předpokládejme **adresaci po bajtech**.
- **Zde je jedno řešení:** Chceme přeložit libovolnou virtuální adresu na libovolnou fyzickou adresu. Máme 3-bitovou virtuální adresu, a tu chceme přeložit na 3-bitovou fyzickou adresu. K tomu stačí tabulka o 8 záznamech, kde jeden záznam bude mít 3 bity, dohromady 8x3=24bitů/proces.



- Problém! Pokud budeme mít 4 GB virtuální prostor, naše Look-up tabulka bude zabírat 2^{32}x32 bitů = 16GB/proces!!! To je poněkud hodně…

Motivace k virtuální paměti.. - Ponaučení z předchozího slide:

- **Mapování z libovolné virtuální adresy na libovolnou fyzickou adresu je prakticky nerealizovatelný požadavek!**
- **Řešení:** Rozdělme virtuální prostor na stejně velké části – virtuální stránky, a fyzickou paměť na fyzické stránky. Ať je velikost virtuální a fyzické stránky stejná. V našem příkladu máme stránku o velikosti 2B.

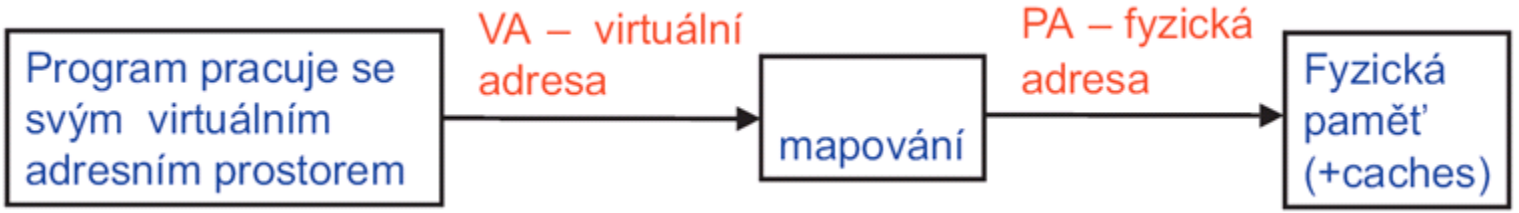


Naše řešení – jeden bit adresy nebudeme pro překlad používat. Look-up tabulka bude mít pak v tomto případě poloviční velikost.

- Naše řešení tedy překládá virtuální adresy po skupinách… Uvnitř dané stránky se pak pohybujeme za pomocí právě toho bitu, který jsme při překladu ignorovali.. Tím jsme schopni využít celý adresní prostor.

Virtualizace paměti

- **VP** je způsob správy operační paměti umožňující běžícímu procesu zpřístupnění paměťového prostoru, který je uspořádán jinak, nebo je dokonce větší, než je fyzicky připojená operační paměť.
- Převod mezi virtuální VA a fyzickou PA adresou může podporovat procesor (HW mapováním TLB, viz dále).
- V současně běžných operačních systémech je virtuální paměť implementována pomocí stránkování paměti spolu se stránkováním na disk, které rozšiřuje operační paměť o prostor na disku.



* R. Lórenc, X36APS, 2005

Příklad č.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;
}
```

Co z toho vyplývá?

- Data jsou v poli uložena za sebou.

Otázky, které se nabízí..

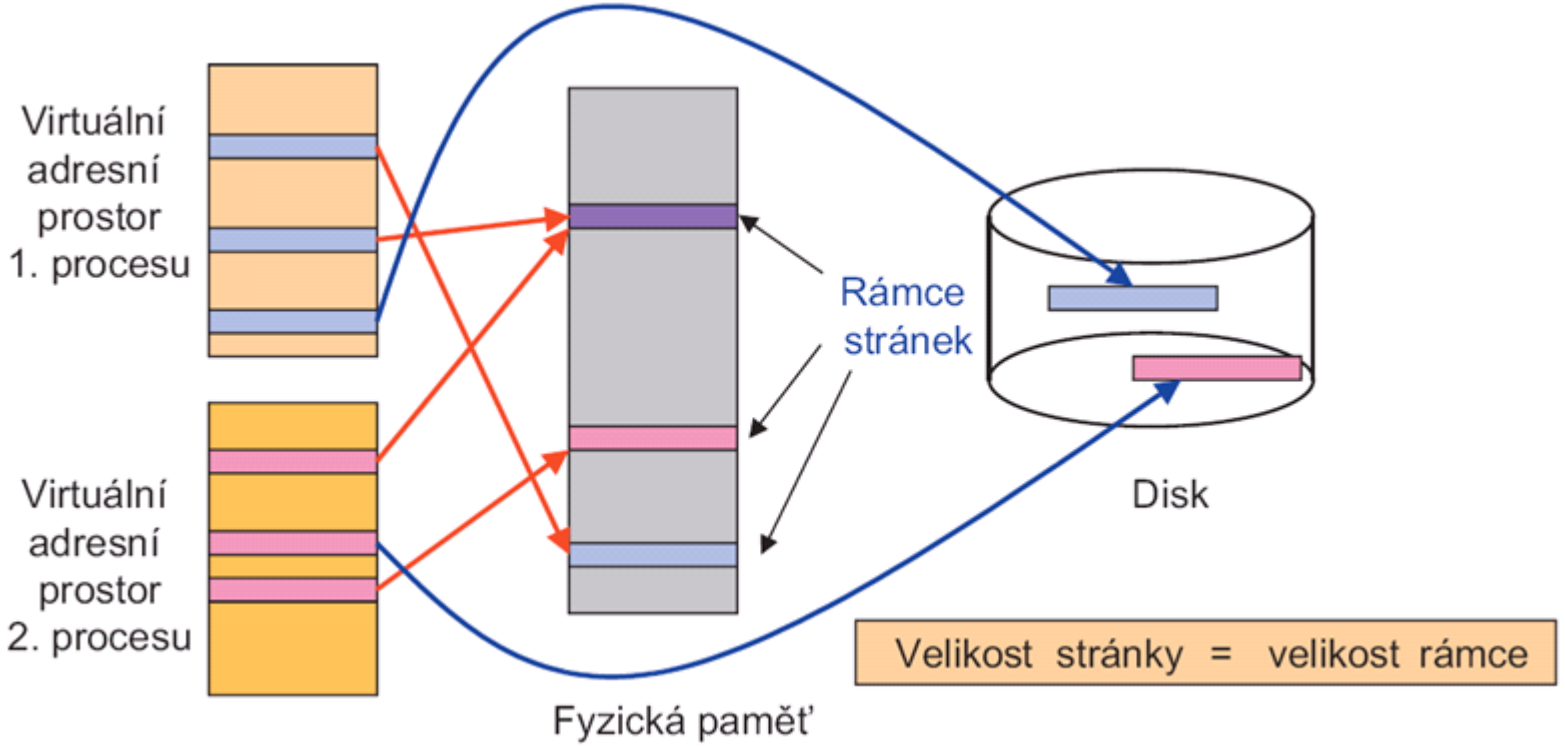
- Ale co je to za adresu?
- Kam do cache se tyto data namapují?

Výpis programu:

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

Virtuální paměť - stránkování

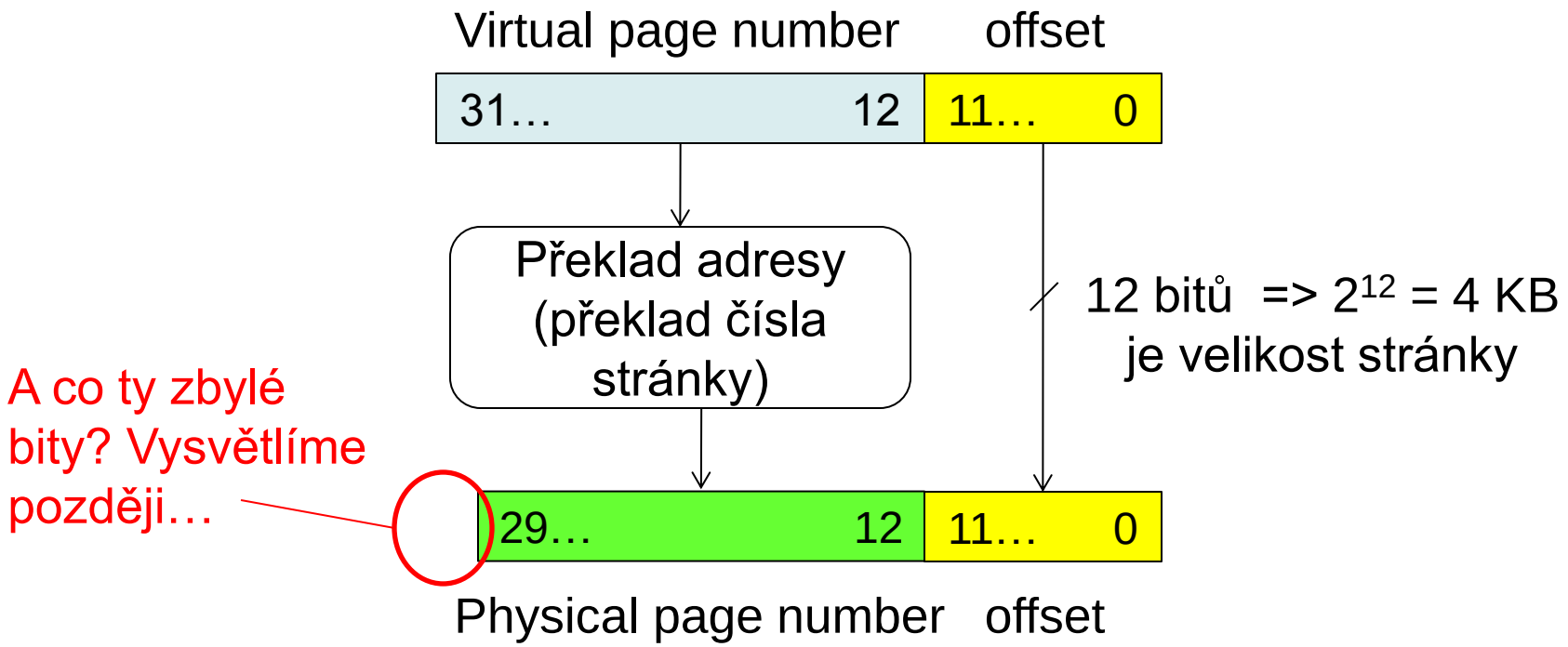
- Virtuální prostor tvoří stejně velké stránky (pages, virtual pages), které se přiřazují jednotlivým běžícím procesům.
- Fyzickou paměť tvoří stejně velké rámce (frames, physical pages).



Zde jen poznámka: některé systémy dovolovaly i různě velké stránky, např. Silicon Graphics IRIX.

Virtuální a fyzické adresování - detailněji

- Předpokládejme virtuální adresu o délce 32 bitů, **1GB fyzické paměti** a velikost stránky 4 KB



Uspořádání překladu, kdy nejnižší bity adresy zůstávají zachovány, má velmi důležitý praktický důsledek, viz dále.

Vraťme se k příkladu č.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;
}
```

Výpis programu:

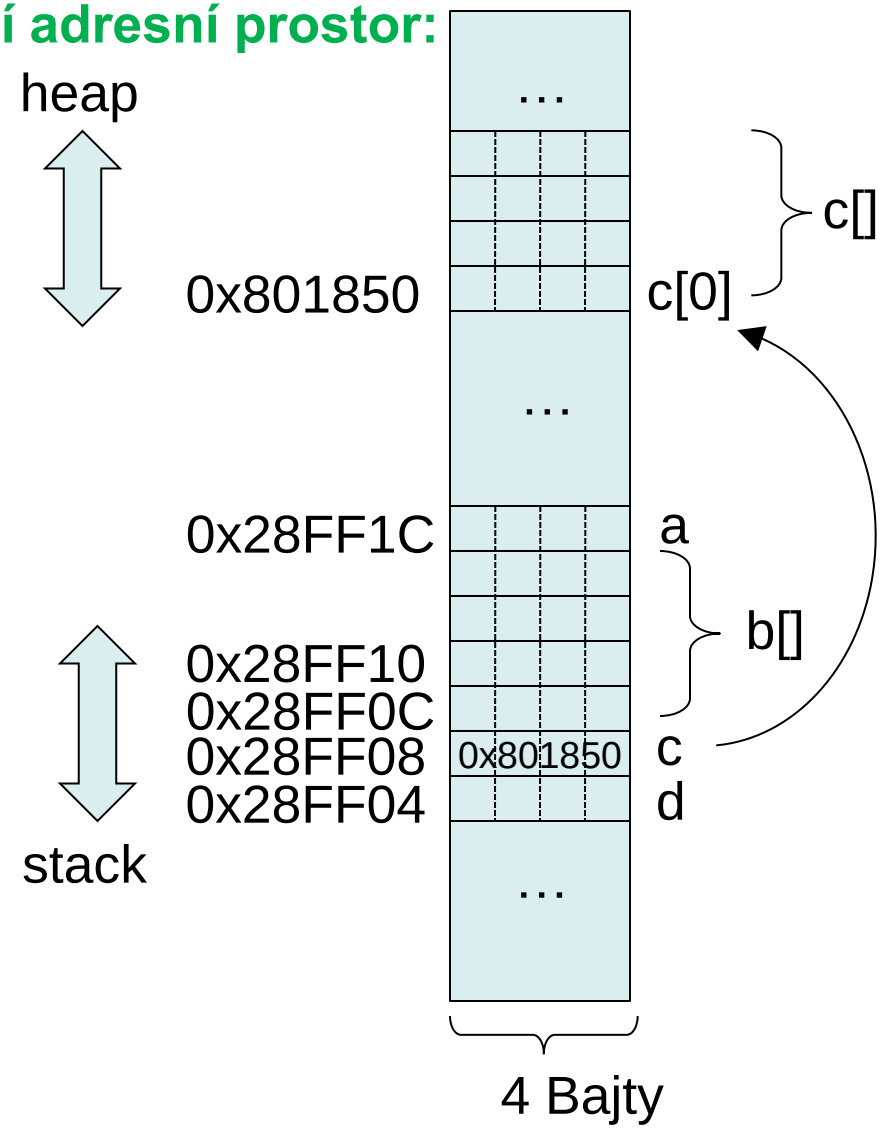
```
0028FF1C 0028FF0C 0028FF08 0028FF04
0028FF0C 0028FF10 0028FF14
00801850 00801854 00801858
```

Vraťme se k příkladu č.1

- Všimli jste si adres, na kterých se nachází proměnné ***a***, **c**, ***d*** a pole ***b***?
- Co když budeme chtít rozšířit náš program třeba o příkazy:

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

//b[2] neinicializováno..



Vraťme se k příkladu č.1

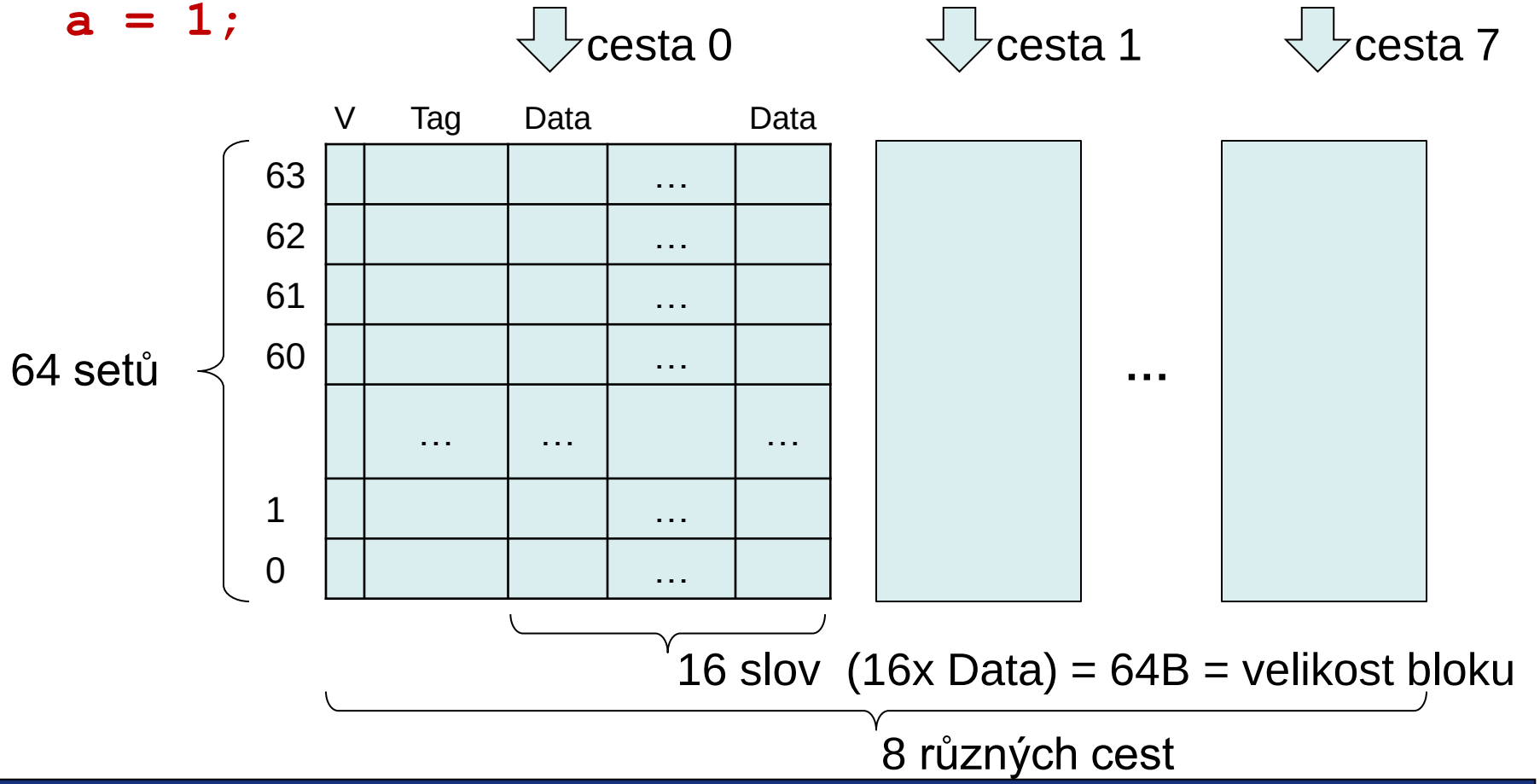
- Předpokládejme L1 datovou cache o velikosti 32kB se stupněm asociativity 8, a velikostí bloku 64B. Cache je na počátku prázdná.
- Co všechno se stane když vykonáme první řádek programu?

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

Vraťme se k příkladu č.1

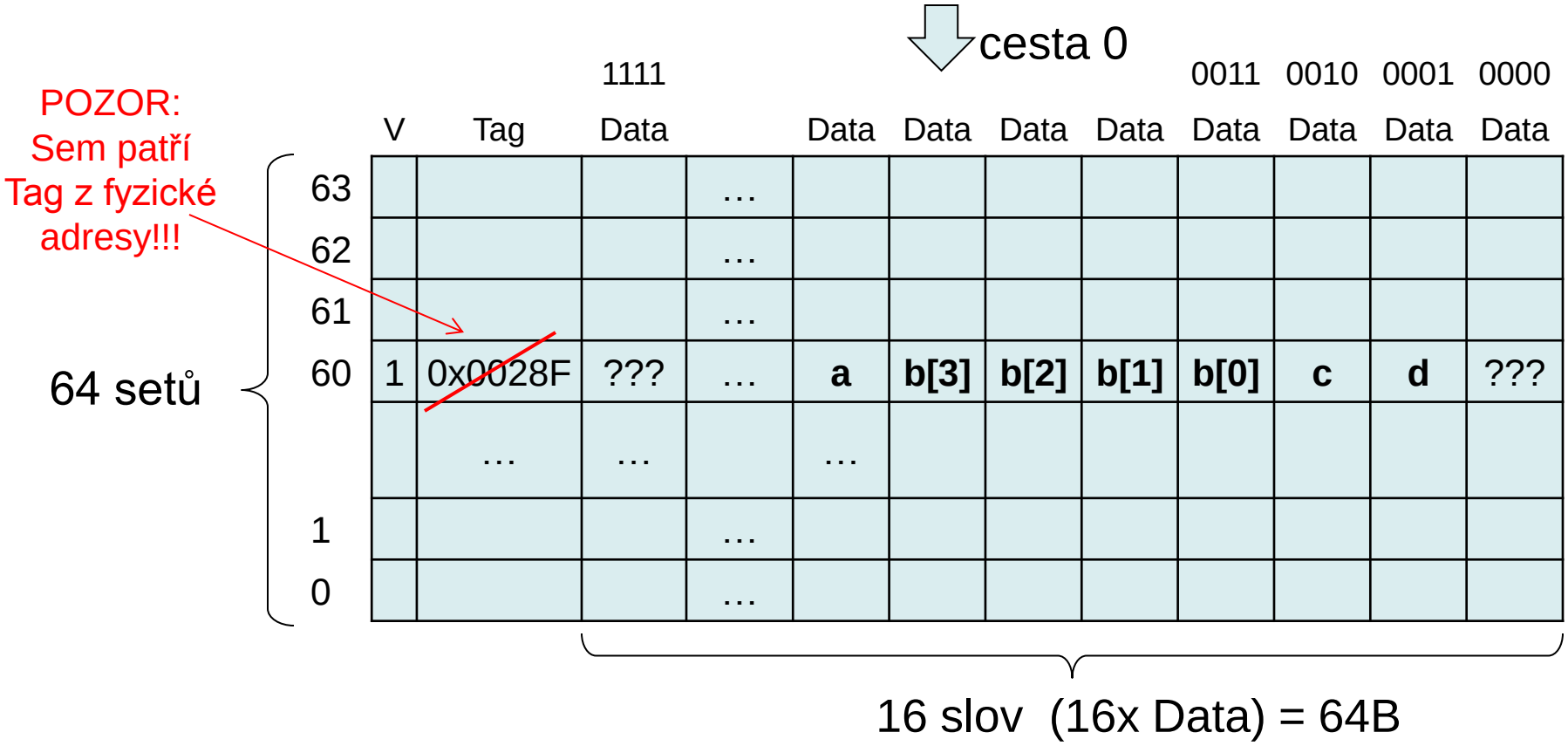
- Předpokládejme L1 datovou cache o velikosti 32kB se stupněm asociativity 8, a velikostí bloku 64B. Cache je na počátku prázdná.
- Co všechno se stane když vykonáme první řádek programu?

```
a = 1;
```



Vraťme se k příkladu č.1

- Předpokládejme L1 datovou cache o velikosti 32kB se stupněm asociativity 8, a velikostí bloku 64B. Cache je na počátku prázdná.
- Co všechno se stane když vykonáme první řádek programu?
  `a = 1; -> cache miss`

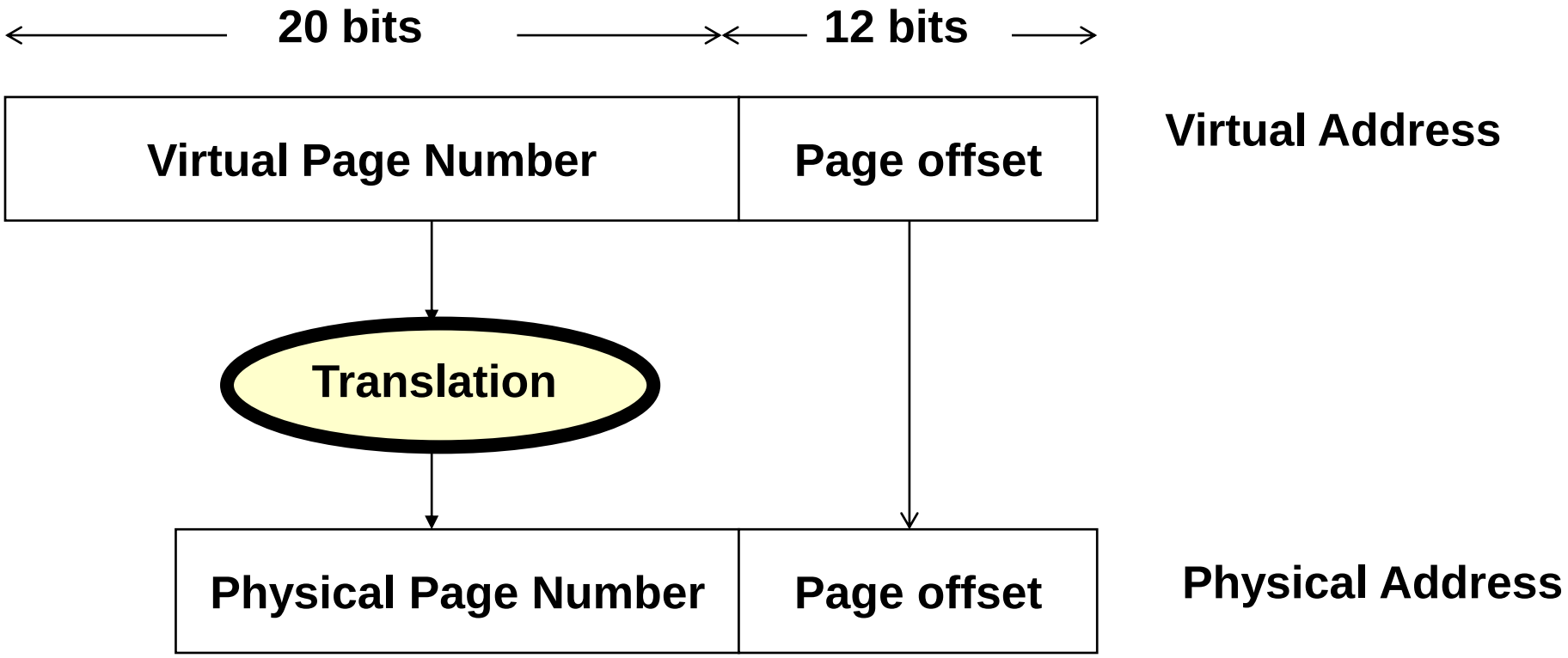


Vraťme se k příkladu č.1

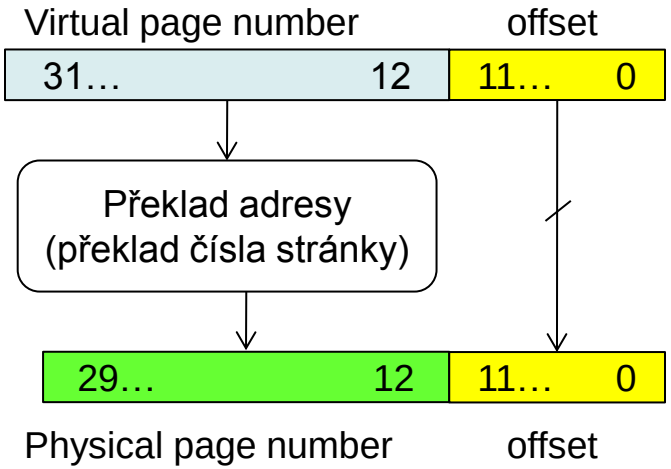
- Stránkování (realizace virtuální paměti) nenarušuje princip prostorové lokality => důležité pro cache.
- **Data na sousedících virtuálních adresách budou uloženy ve fyzické paměti vedle sebe**
  (samozřejmě pokud nepřekročí hranici stránky).
- Nastane-li page fault, tj. stránka není ve fyzické paměti, pak se:
  1. Vyvolá se výjimka, kterou obslouží OS.
  2. V paměti se zvolí stránka, která se má nahradit, a ta se uloží na disk, je-li to potřeba.
  3. Poté se do ní načte nový obsah požadované stránky (z disku), nebo se tam namapuje vymazaná stránka (při nové paměti).
  4. Adresovaný blok (cache row) se okopíruje i do cache.
  5. Další cache miss uvnitř stránky již nevyvolává page fault, a to až do doby, než je stránka nahrazena jinou stránkou.

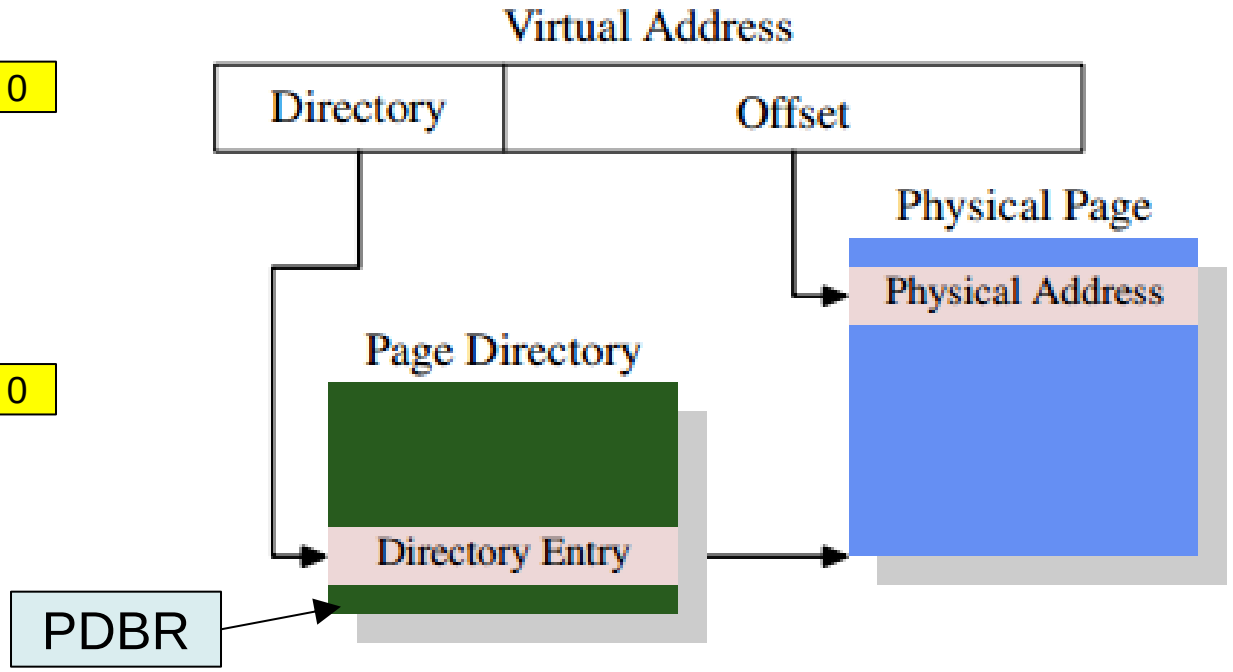
Realizace převodu adres?

- Tabulka stránek, **Page Table**.
- Jednotkou mapování jsou stránky,
- Stránka je také jednotkou přenosu mezi vedlejší a hlavní paměti.
- Mapovací funkce se nejčastěji implementuje Look-up Table (vyhledávací tabulkou).
- O překlad virtuálních adres na fyzické se stará **Memory Management Unit** (**MMU**)
- MMU je součástí CPU


20 bits
12 bits
Virtual Page Number
Page offset
Virtual Address
Translation
Physical Page Number
Page offset
Physical Address

Jednoúrovňová realizace převodu adres?



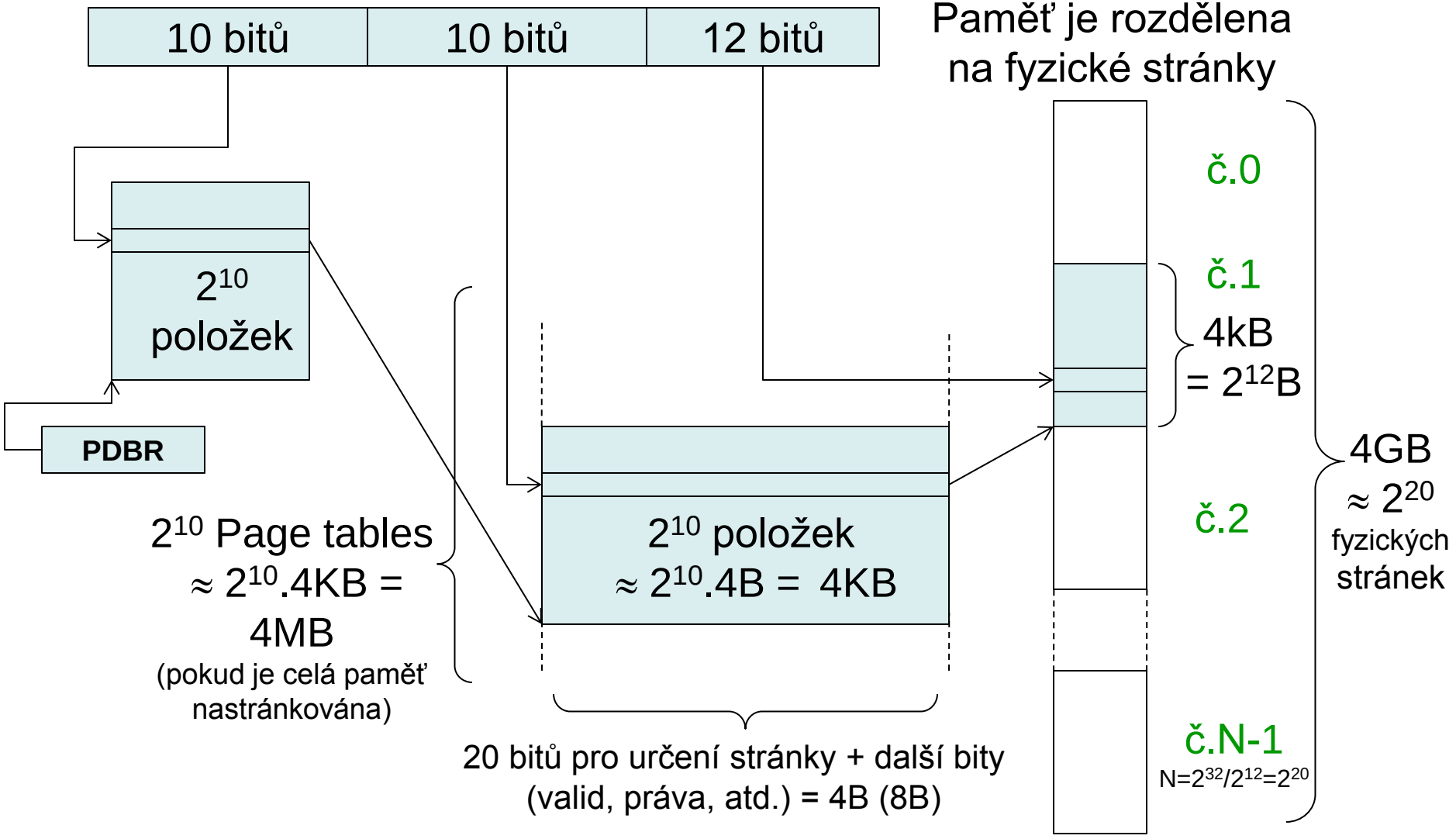


- Datová struktura pro Page Directory (Page Table) je uložena v hlavní paměti. Úkolem operačního systému je alokovat souvislou oblast paměti a počáteční adresu této oblasti uložit do speciálního registru CPU.
- **PDBR** - **Page Directory Base Register** – v x86 procesorech uložen v registru CR3, který obsahuje fyzickou adresu
- **PTBR** - Page Table Base Register– to samé…

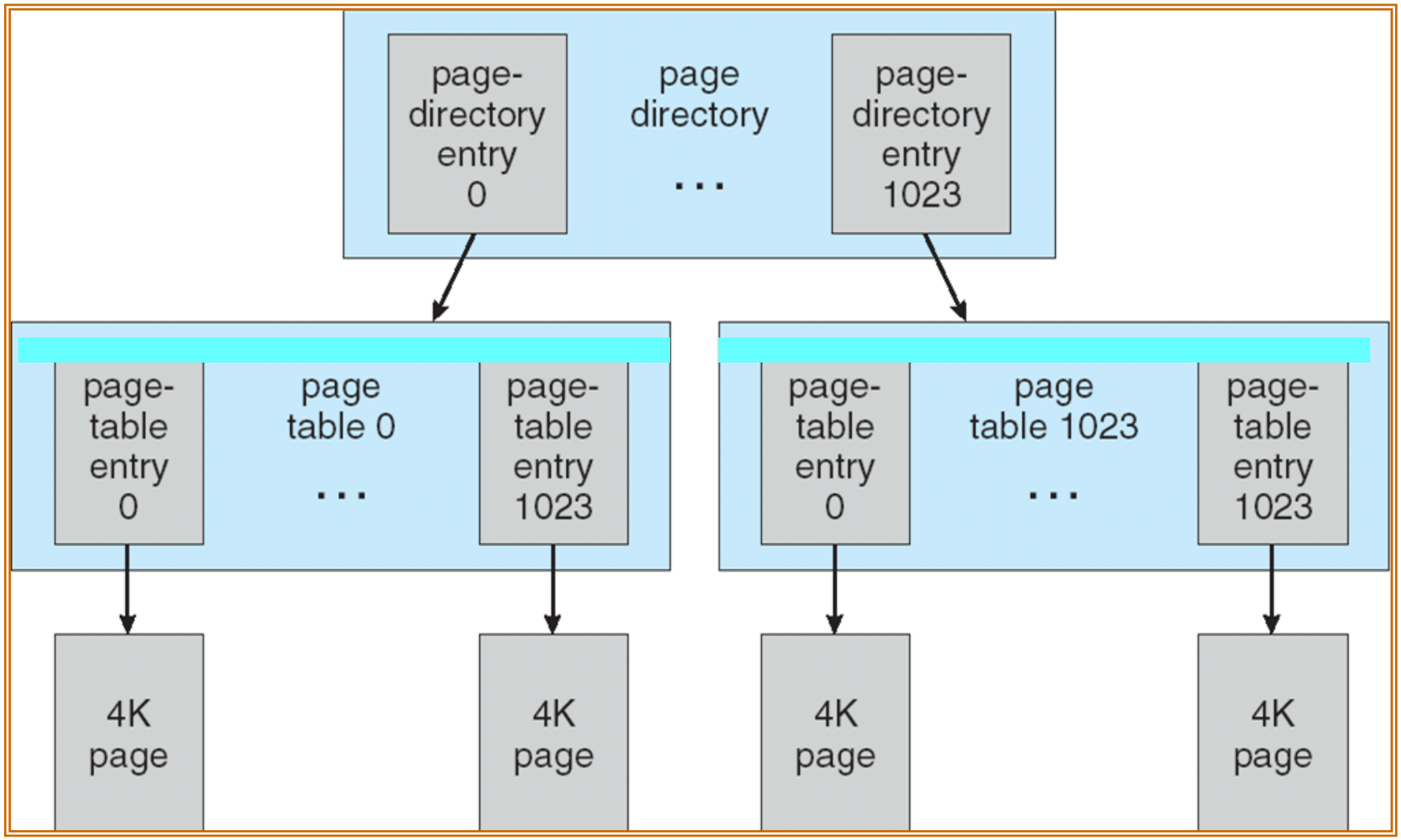
Uvažujme…

- Stránka je typicky 4 kB = 2^12
- Když budeme znát adresu stránky, postačuje nám tedy jenom 12 bitů na pohyb (adresaci) v ní. Zbývá 20 bitů (pro 32-bitovou adresu).
- Tudíž Page Directory (Page Table) by měl obsahovat 2^20 položek. To je nepraktické a přináší řadu nevýhod.
- Typický proces/vlákno se v daném „okamžiku“ pohybuje pouze v malé části svého adresního prostoru – princip časové a prostorové lokality…
- Řešením je více-úrovňové stránkování.

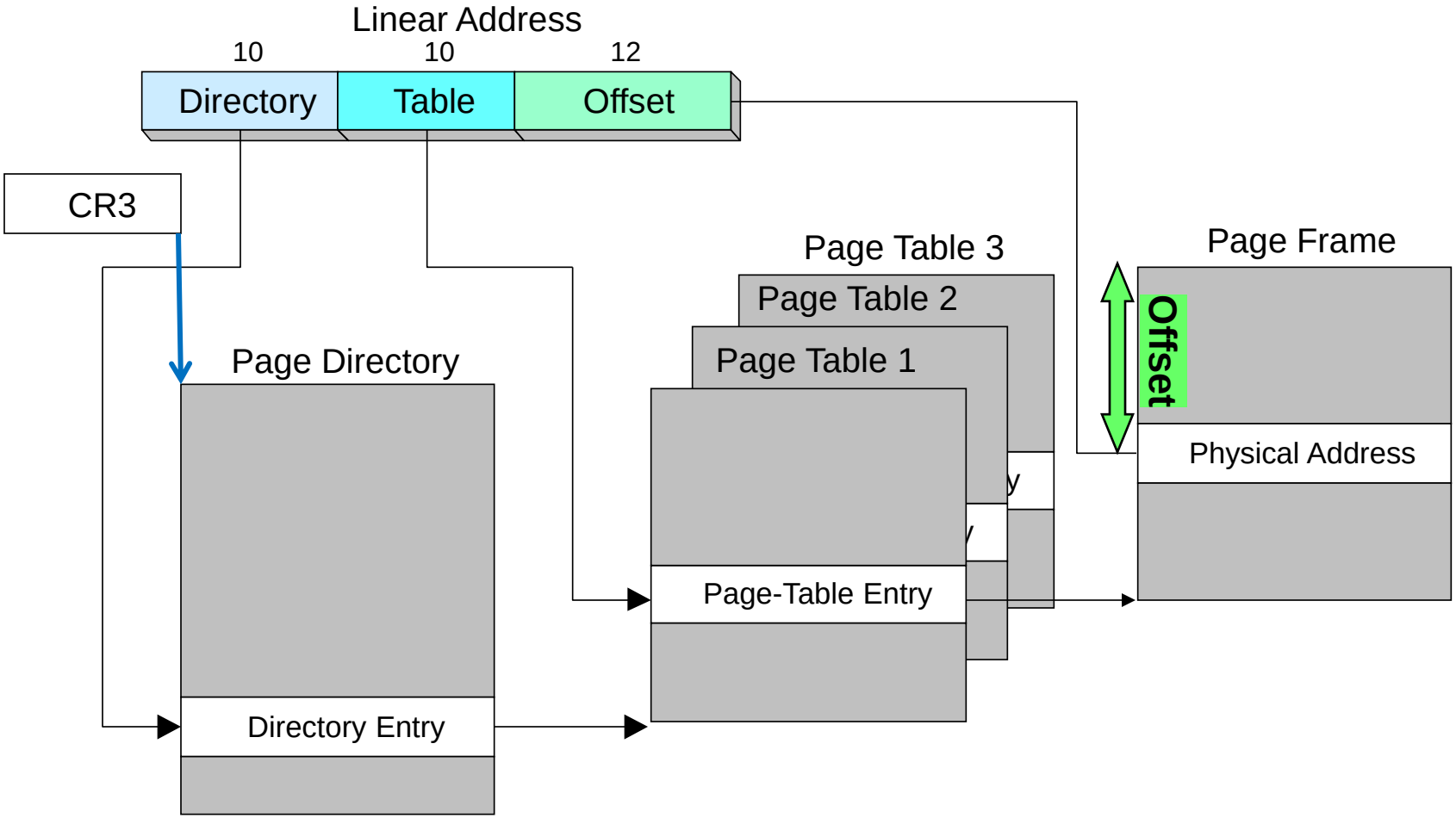
Více-úrovňové stránkování – 2 úrovně



Princip mapování 1/4
–nakreslený pomocí stromové struktury

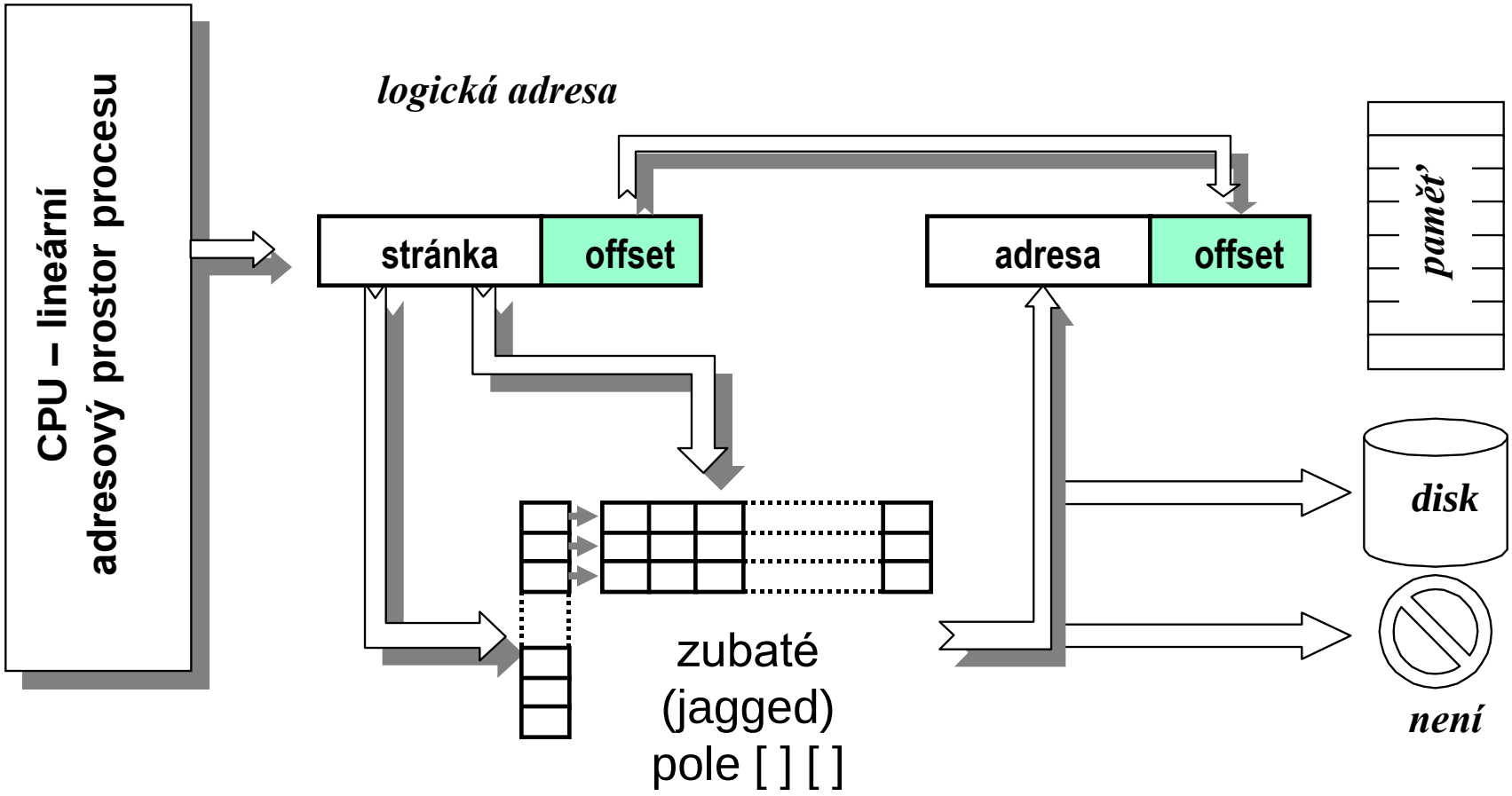


Princip mapování 2/4
– *rozklad lineární adresy na indexy*



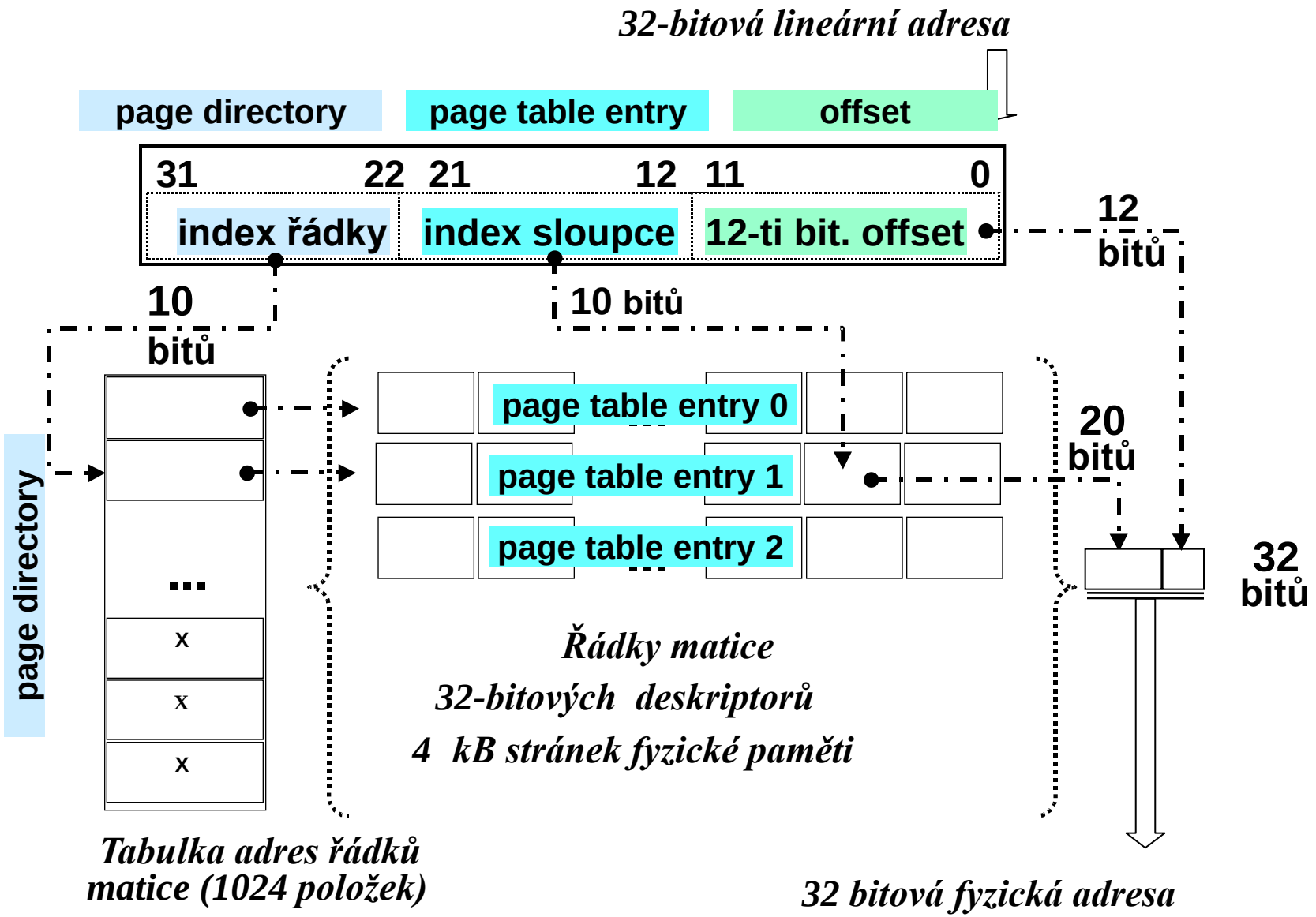
Princip mapování 3/4 – indexy do mapovací matice

- Princip převodu lineární adresy na fyzickou adresu (*page translation*)

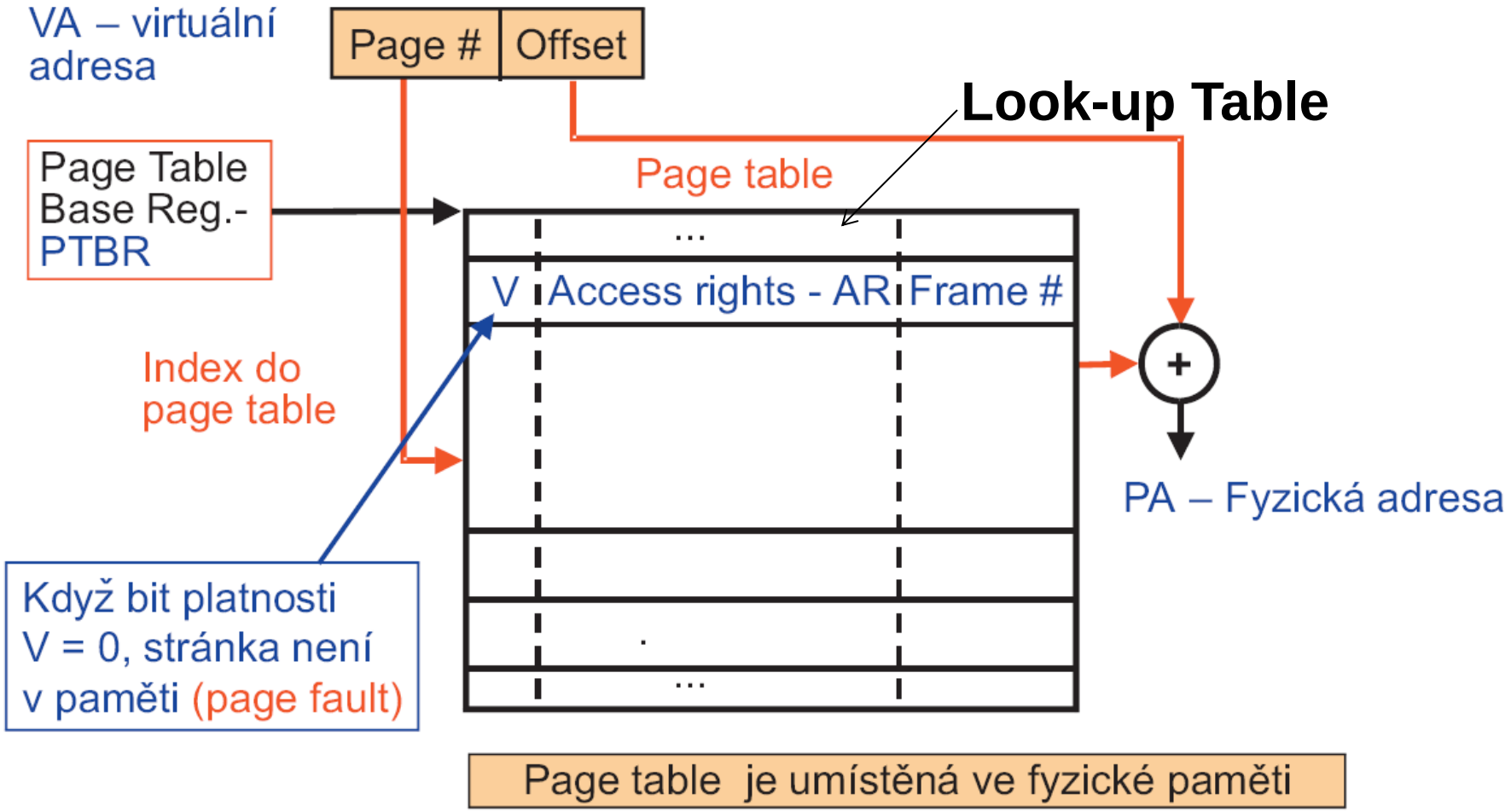


Pozn. *OS Windows i Linux přidělují paměť vždy po stránkách (zpravidla 4 KB), srovnej s "cluster"=alokační konstanta disku!*

Princip mapování 4/4 - operace procesoru

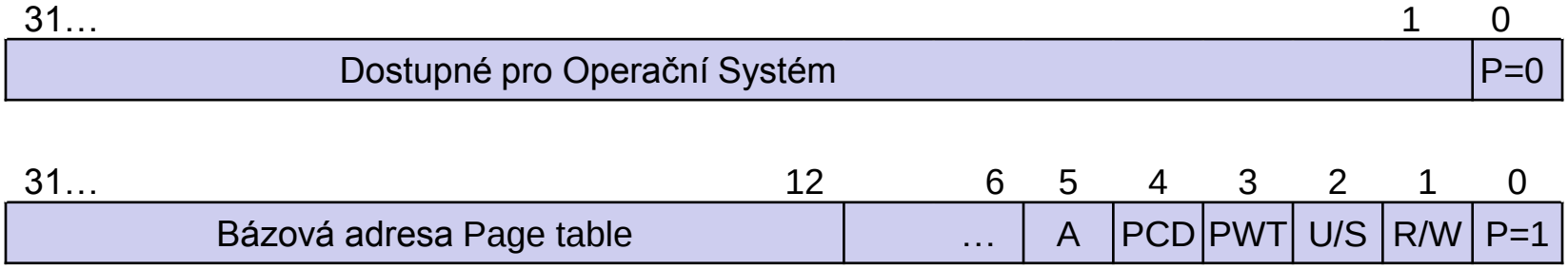


Tabulka stránek – jak vypadají položky? Význam položek…



Tabulka stránek – jak vypadají položky? Význam položek…

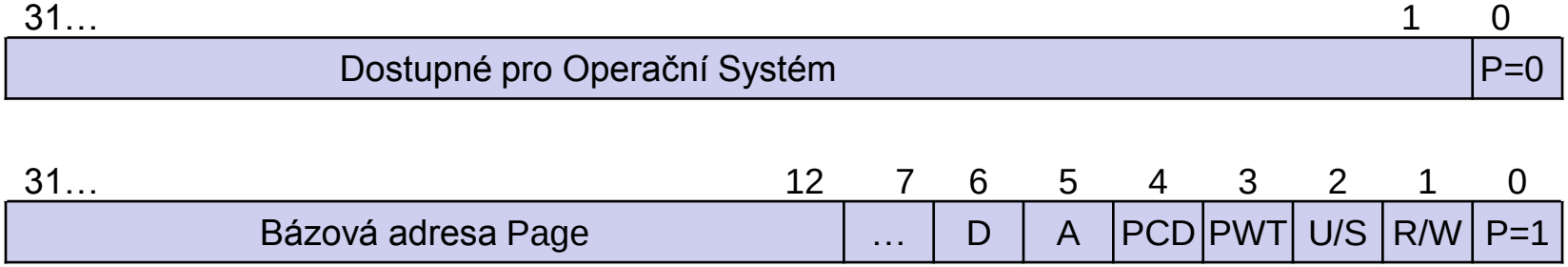
Podívejme se na položku **Page Directory** (Page Table similar)



- **P** -bit 0: Present bit – *určuje zda je stránka v paměti (1), nebo na disku (0) Někdy se tento bit označuje V – valid*.
- **R/W** -bit 1: Read/Write: *pokud 1 – R/W; pokud 0 – jenom ke čtení*
- **U/S** -bit 2: User/Supervisor: *1 – uživatelský přístup; 0 – pouze OS*
- **PWT** -bit 3: Write-through/Write-back – *zapisovací strategie pro stránku*
- **PCD** -bit 4: Cache disabled/enabled – *některé periferie jsou namapovány přímo do paměti (memory mapped I/O), čímž je umožněn zápis/čtení do/z dané periferie. Tyto adresy v paměti pak chápeme jako I/O porty. Nezapisuje se do cache..*

Tabulka stránek – jak vypadají položky? Význam položek…

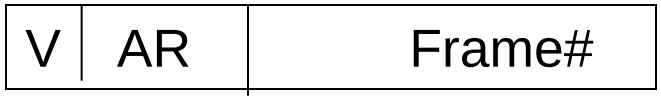
Podívejme se na položku Page Table (Page Table na 2.úrovni)



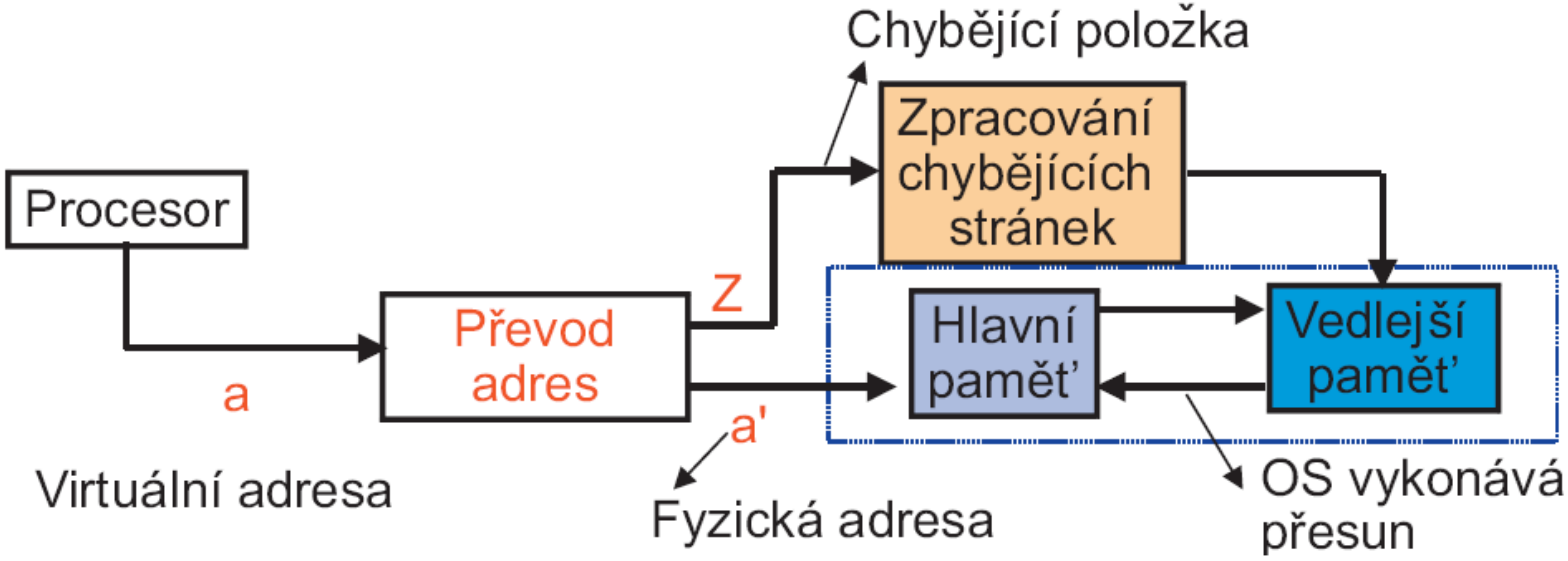
- **A** -bit 5: Accessed – *zda jsme četli/zapsali – pomáhá rozhodovat o tom, které stránky mají být odstraněny když potřebujeme uvolnit prostor v paměti*
- **D** bit 6: Dirty bit – Je nastaven pokud jsme do stránky zapsali.
- 11..7 bit - speciální použití, třeba možnost uložit typ paměti, kdy provádět update cache, apod.
- bit 31-12 - Physical Page Address

Poznámky

- Každý proces má svou Tabulku stránek,
- Tedy i svou hodnotu PTBR (bázového registru).
- To, mimochodem, zajišťuje paměťovou bezpečnost procesů.
- Co chceme abyste si zapamatovali z Formátu položky Tabulky stránek?
  - V – Validity Bit. V=0 Stránka není platná (je na disku).
  - AR – Access Rights. Přístupová práva (Read Only, Read/Write, Executable, apod.),
  - Frame# - číslo rámce (bázová adresa do nižší úrovně),
  - Popřípadě další, např. Modified/Dirty, apod. (budeme dále podle potřeby doplňovat).



Virtuální paměť: spolupráce HW a SW



Co dělat, když je výpadek stránky – Page Fault?

Paměti je nedostatek

- Pomocí LRU najdeme stránky, které můžeme uvolnit.
- Mají-li nastaven Dirty bity, zapíšeme je „nějak“ (zpravidla DMA, Direct Memory Access, přímým přístupem do paměti) na disk.
- Aktualizuje se Tabulka stránek procesu, čímž máme již paměť volnou.

Fyzická paměť je volná,

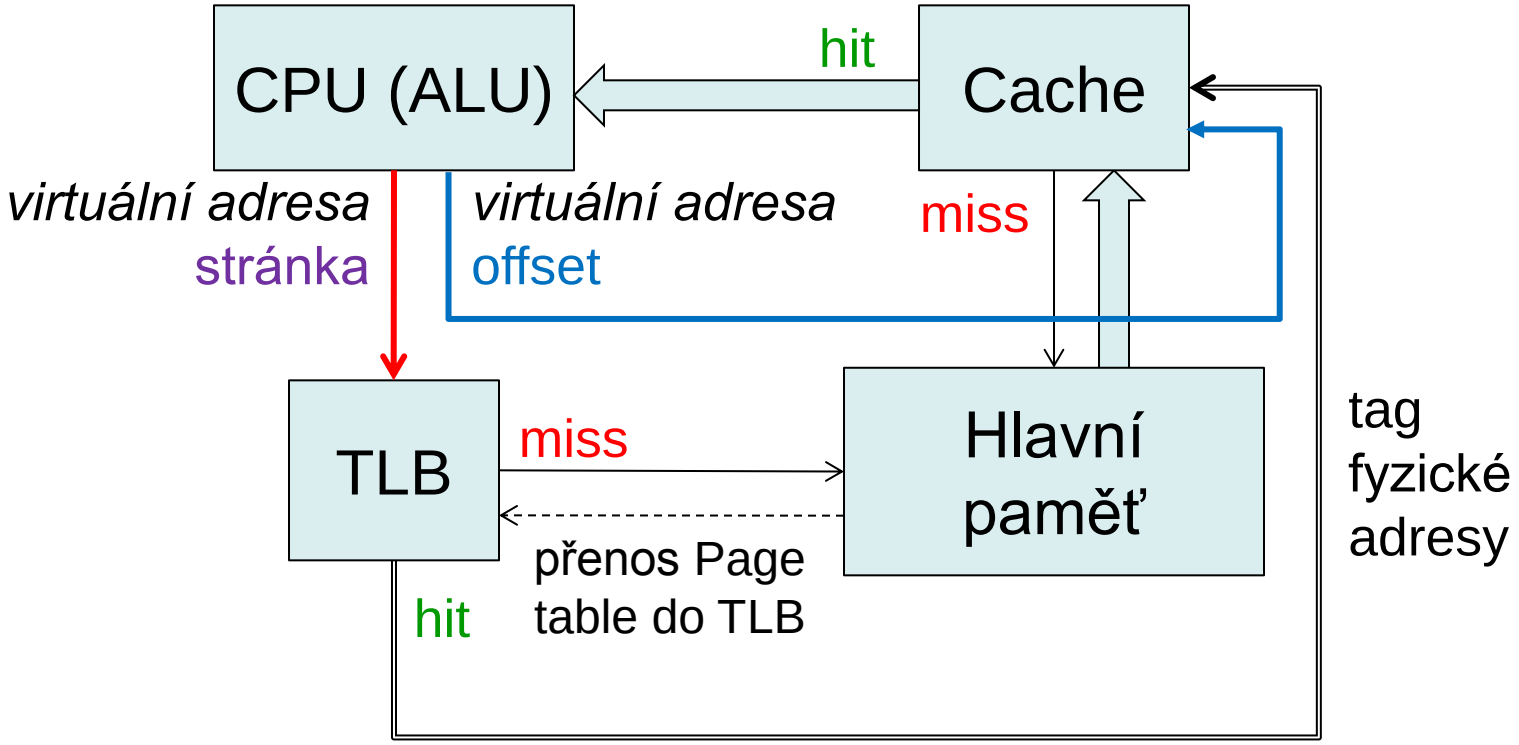
ale data jsou ve vedlejší paměti (na disku).

- Požadovaná stránka se načte (DMA) do prázdného rámce.
- Pokud stránka není na disku, tj. alokace prázdné paměti, a nežádá ji kernel, **pak se musí celá vynulovat** (z bezpečnostních důvodů).
- Po dokončení DMA přenosu se vyvolá přerušení, aktualizuje se Tabulka stránek procesu.

Během procesu stránkování lze přepnout na jiný čekající proces, který může probíhat, dokud se operace neskončí.

.

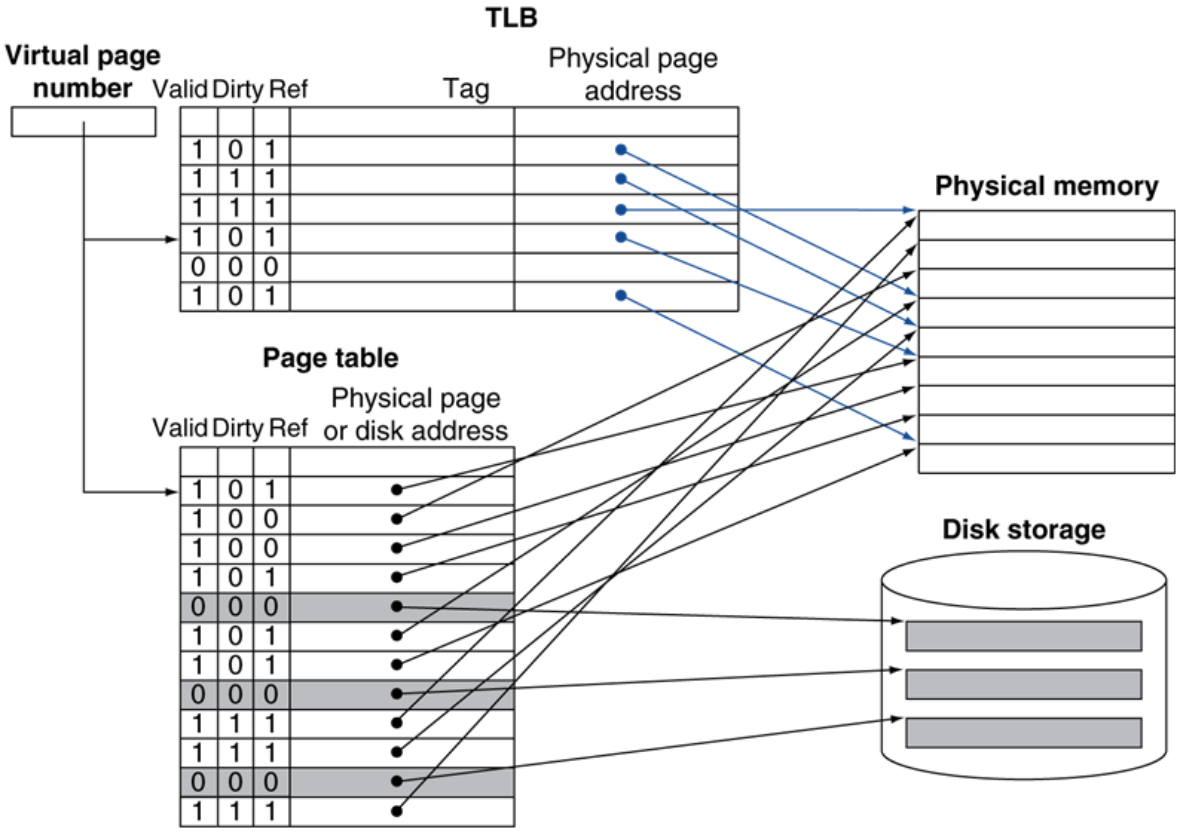
Idealizace překladu adres pomocí TLB - čtení



- Všimněte si, že může dojít k miss-u 2x
- Pokud nastane TLB miss, musíme vykonat tzv. *page walk*

Rychlá realizace Tabulky stránek - TLB

- Translation-lookaside Buffer, výstižnější by byl termín překládací keš (Translation Cache).
- Je vlastně SP (keší) adres stránek.



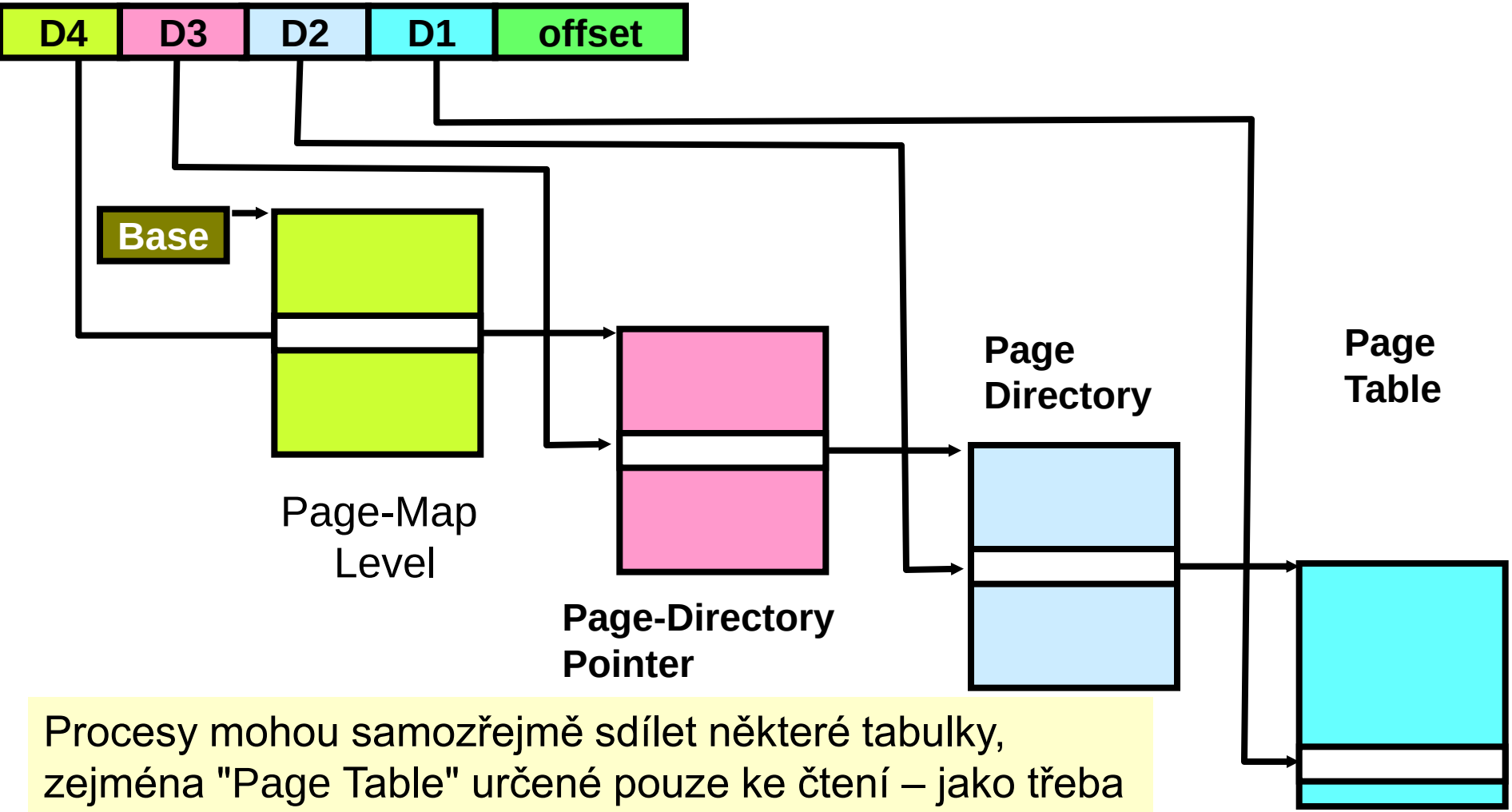
Stránkování u 64 bitových procesorů

U 64 bitových systému

1. Možno zvětšit délku stránky z 4kB až na 1GB, ale nepraktické.
2. Možno využít víceúrovnové stránkovací tabulky, přidat
   - *Page-Directory Pointer Table* (Win64: 4 až 512 adres)
   - *Page-Map Level* (Win64: až 512 adres)

   Také délka deskriptoru ve všech tabulkách zvýší ze 32 bitů na 64 bitů

Více úrovňové tabulky pro 64 bitové procesory



Procesy mohou samozřejmě sdílet některé tabulky, zejména "Page Table" určené pouze ke čtení – jako třeba knihovny

Více-úrovňové stránkování

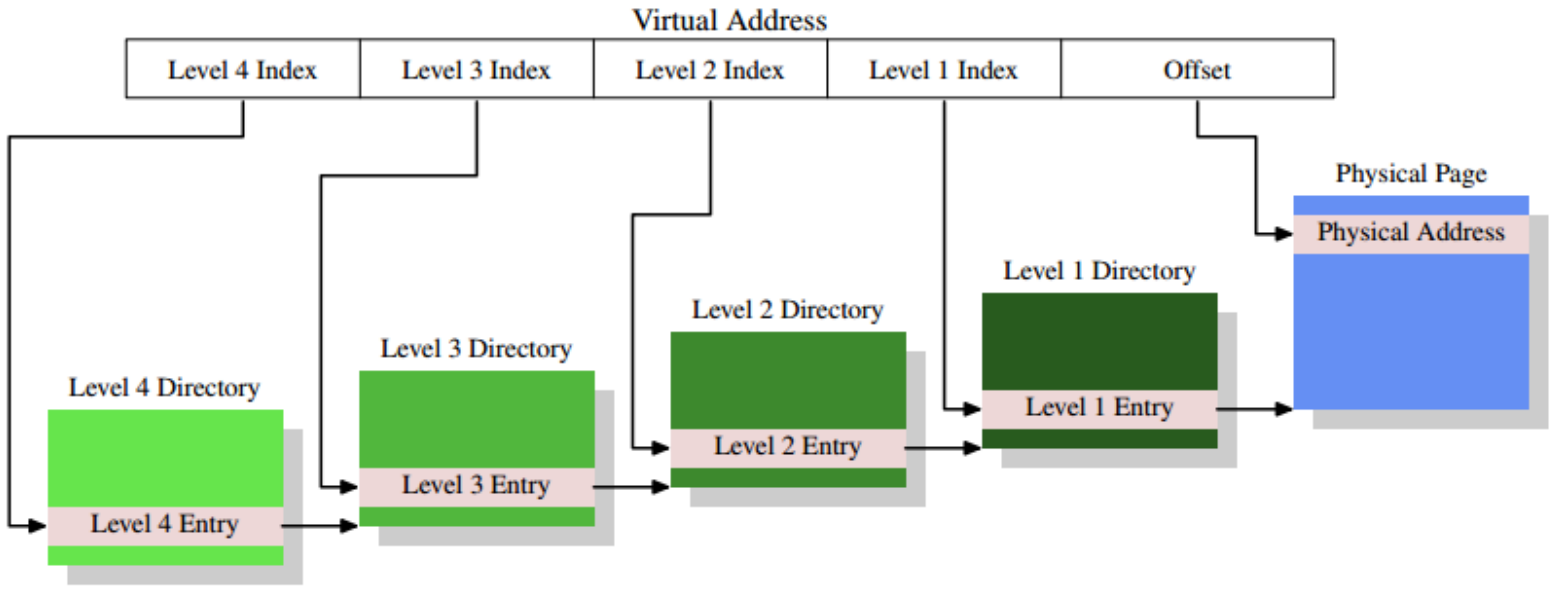
Poznámky k předchozímu slide:

- Ne každý proces využívá celý svůj adresní prostor => není nutné alokovat v druhé úrovni 2^{10} Page tables
- Tabulky stránek můžou být rovněž stránkovány

Obecné poznámky:

- Intel IA32 implementuje 2-úrovňové stránkování
  - Page Table v úrovni 1 označuje jako Page Direcory (10 bitů pro adresaci)
  - Page Table v úrovni 2 pak jako Page Table (10 bitů)
- V případě 64-bitové virtuální adresy je obvyklé používat méně bitů pro fyzickou adresu – například 48, nebo 40.
- Intel Core i7 používá 4-úrovňové stránkování a 48 bitový adresní prostor
  - Page Table v úrovni 1: Page global directory (9 bitů)
  - Page Table v úrovni 2: Page upper directory (9 bitů)
  - Page Table v úrovni 3: Page middle directory (9 bitů)
  - Page Table v úrovni 4: Page table (9 bitů)

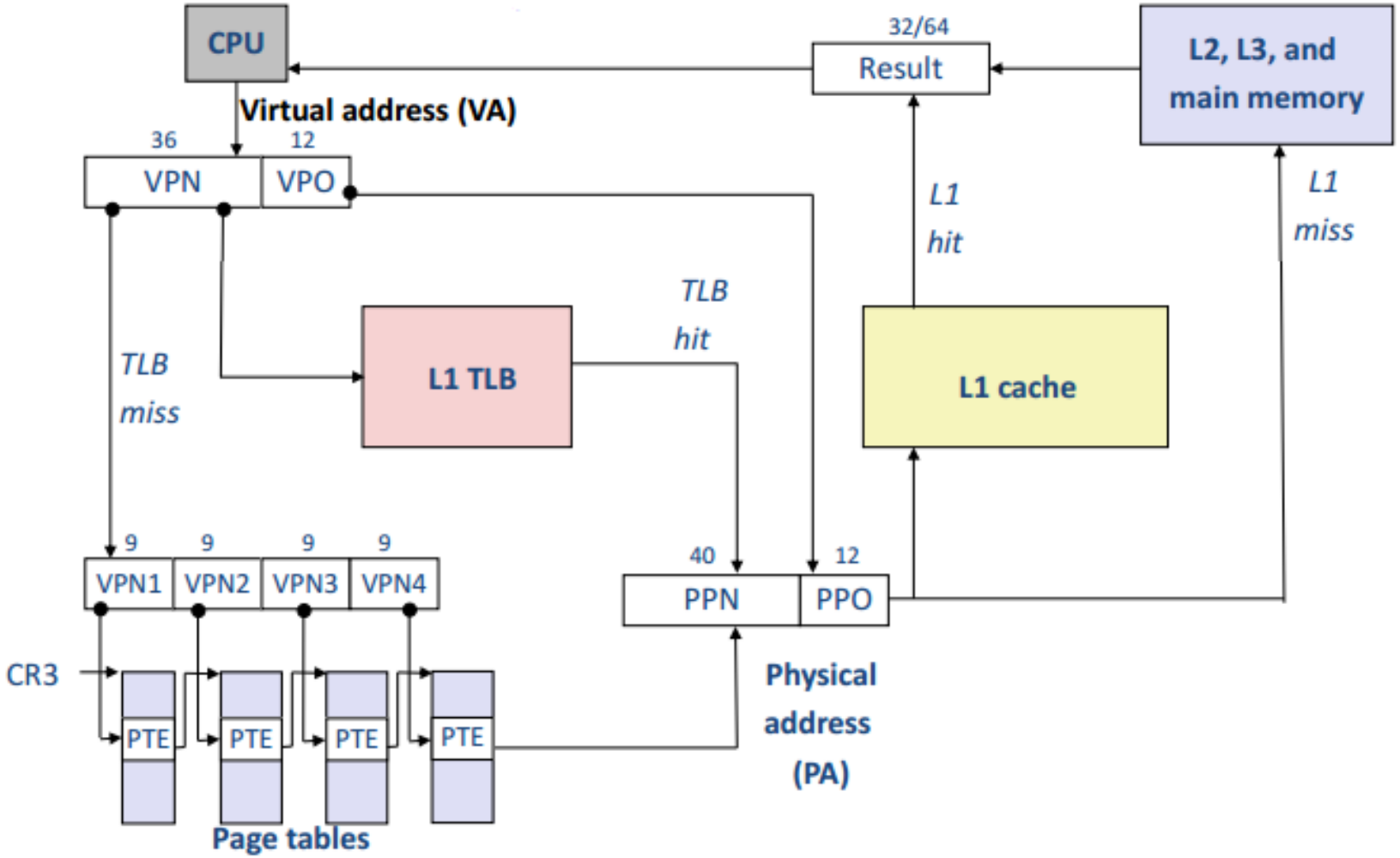
Více-úrovňové stránkování – Problém rychlosti



4-Level Address Translation

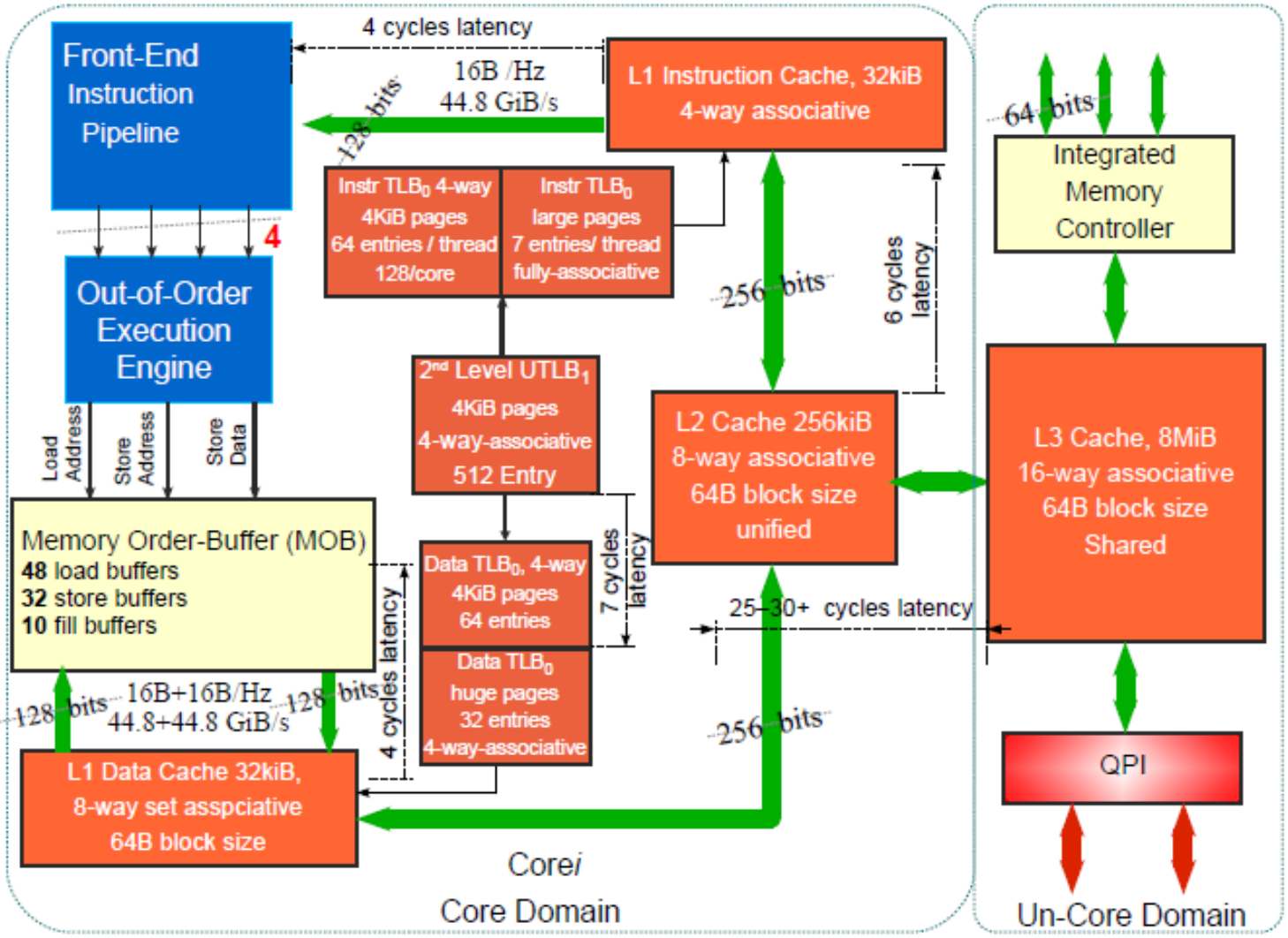
- Pokud bychom předpokládali, že všechny položky pro výpočet adresy máme již v cache, bude i tak výpočet adresy trvat velmi dlouhou (v závislosti od počtu úrovní – nelze paralelizovat).
- Výhodnější je přímo cachovat „vypočtené“ adresy.
- K tomu slouží Translation Look-Aside Buffer (TLB)
- Dnes se používají více-úrovňové TLB

Překlad adres – Intel Nehalem (Core i7)

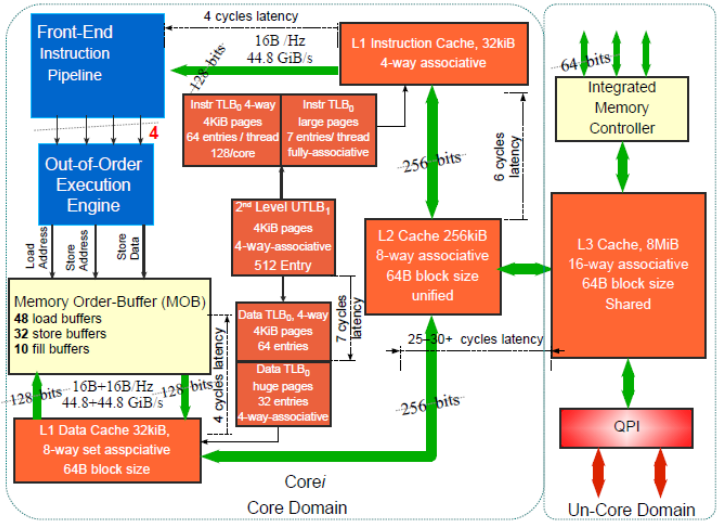


http://cs.nyu.edu/courses/spring13/CSCI-UA.0201-003/lecture18.pdf

Organizace paměti - Intel Nehalem (Core i7)



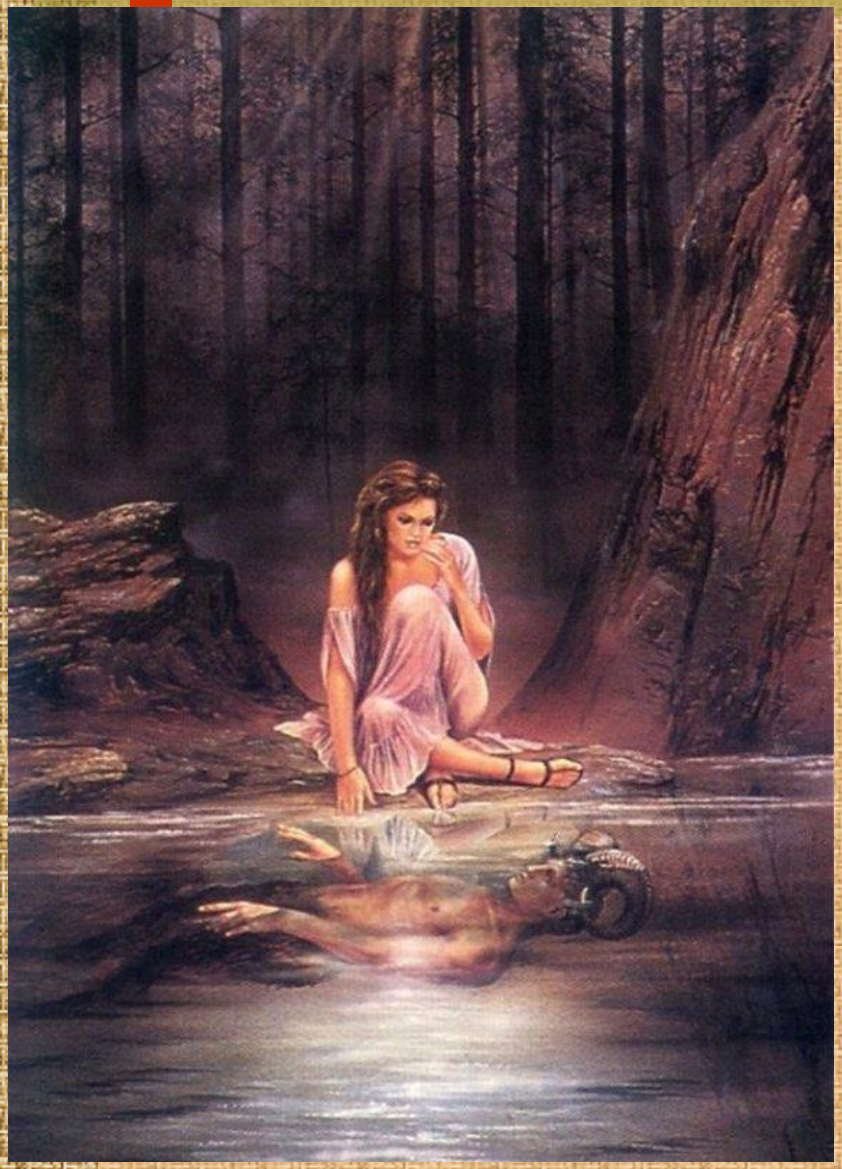
Organizace paměti - Intel Nehalem – několik poznámek



- Velkost bloku: 64B
- procesor vždy čte řádek cache ze systémové paměti zarovnán na 64B (6 LSb adresy jsou nuly) a nepodporuje částečně plněné řádky
- L1 – Harvard. V SMT sdílená oběma vlákny, Instrukční – 4-way, Datová 8-way.
- L2 – unifikovaná, 8-way, neinkluzivní, WB
- L3 – unifikovaná, 16-way, inkluzivní (řádek obsažen buď v L1 nebo L2 se nachází v L3), WB
- Store Buffers – dočasně uchovávají data pro každý zápis. Netřeba čekat na zápis do cache či paměti. Zajišťují, že zápisy jsou ve správném pořadí a také když je potřeba:
  - výjimka, přerušení, instrukce serializace, lock,..
- Můžete si také všimnout oddělených TLB (Translation Lookaside Buffer)

Pro vaší představu: typické hodnoty

	L1	Typicky pro stránkované paměti	Typicky pro TLB
Velikost v blocích	256-4k	16 000-250 000 000	40-1024
Velikost	16-64 kB	500 - 1 TB	0,25-16 KB
Velikost bloku v bytech	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]

Stránkování a fragmentace paměti

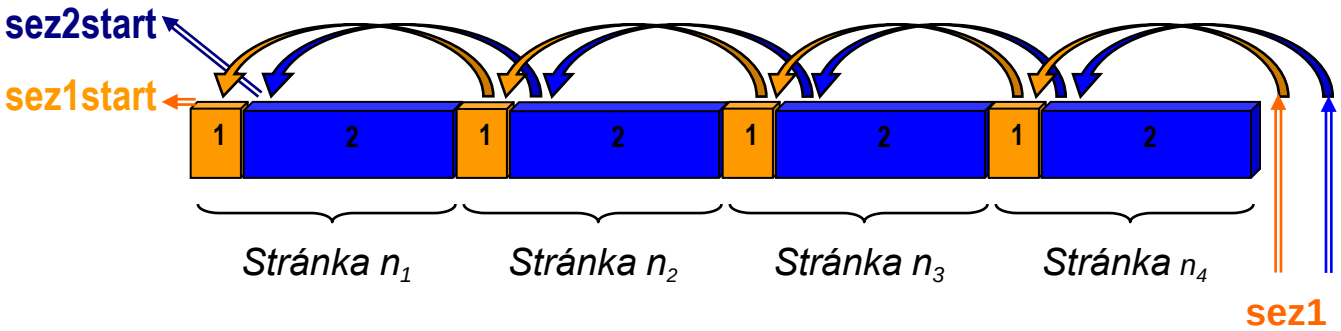
Jedno úmrtí znamená tragédii, miliony mrtvých statistiku.

[*Josef Stalin*]

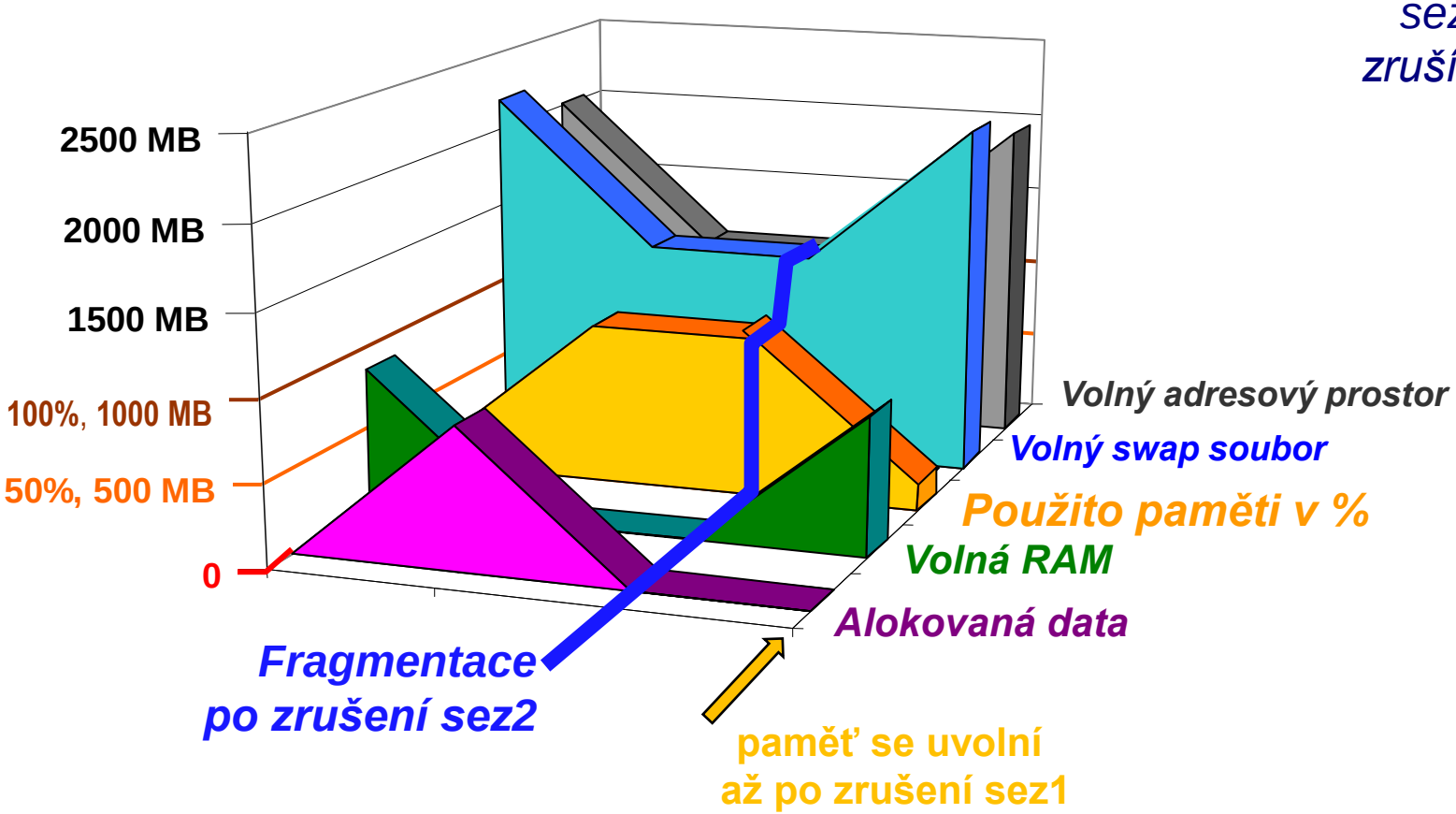
Fragmentace paměti

- **Důvody fragmentace**
  a) proměnlivé chování programu
  b) program žádá velké bloky dat, ale uvolňuje pouze malé
  c) jednotlivá úmrtí
     - ruší se tím data, která spolu nesousedí, a tak nelze scelit jejich paměťové oblasti a učinit je použitelné pro velké objekty.

Příklad fragmentace v C++



Pokud sez2 zrušíme!



Efektivnější používáni paměti – prostředek zrychlení programu

Váš program může brát v potaz velikost stránky a používat paměť efektivněji – jednak zarovnáním alokací na násobek velikosti stránky a pak redukcí interní a externí fragmentace stránek.. (pořadí alokací atd. Viz také *memory pool*)

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

Akolace paměťově zarovnaného bloku:

```
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("Velikost stranky je: %ld B.\n",
       ns.dwPageSize);
    printf("Rozsah adres pro aplikaci (a dll):
       0x%lx - 0x%lx\n",
       s.lpMinimumApplicationAddress,
       s.lpMaximumApplicationAddress);
    return 0;
}
```