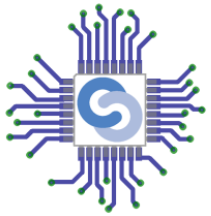


Calculatoare Numerice (2)

-Cursul 4 –

Memoria virtuală

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

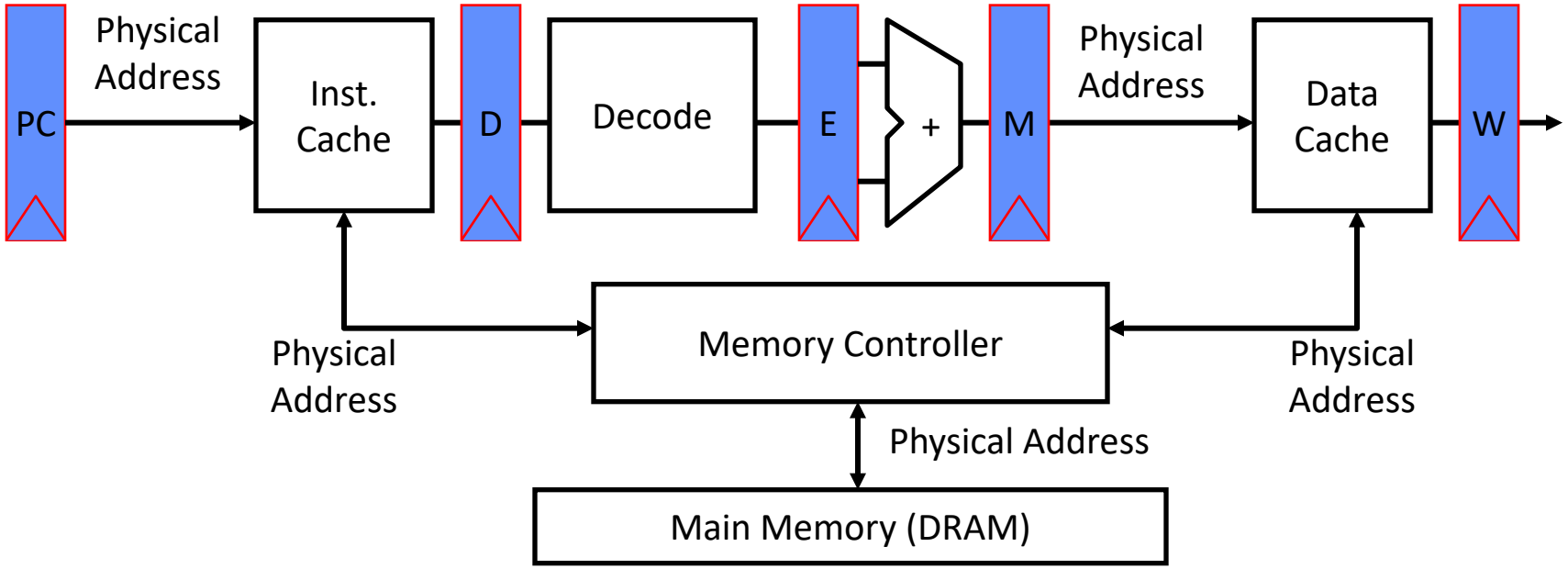
Comic of the Day

http://dilbert.com/strips/comic/2008-02-12/

Din episodul anterior

- 3 C's of cache misses
  - Compulsory, Capacity, Conflict
- Write policies
  - Write back, write-through, write-allocate, no write allocate
- Ierarhiile cache multi-nivel reduc penalizările la miss
  - 3 niveluri în sistemele moderne (unele au chiar 4!)
  - Prezența L2 modifică organizarea L1
- Prefetching: aduce date în cache înainte de o cerere a CPU-ului
  - Prefetching poate să irosească lățime de bandă și să cauzeze poluarea cache-ului
  - Software vs hardware prefetching
- Optimizări software pentru cache
  - Loop interchange, loop fusion, cache tiling

Mașina de calcul simplă



- Într-o mașină de calcul foarte simplă, orice adresă este o adresă fizică

Adrese absolute

EDSAC, anii 50'

- Nu aveam concurență, un singur fir de execuție cu acces nerestricționat la întreaga mașină (RAM + I/O devices)
- Adresele dintr-un program depindeau de locația de memorie unde programul era încărcat
- *Dar* era mult mai convenabil pentru programatori să scrie subrutine independente de locație

Cum poate fi obținută independența de locații?

Linker și/sau loader care modifică adresele subrutinelor atunci când creează imaginea în memorie a programului respectiv

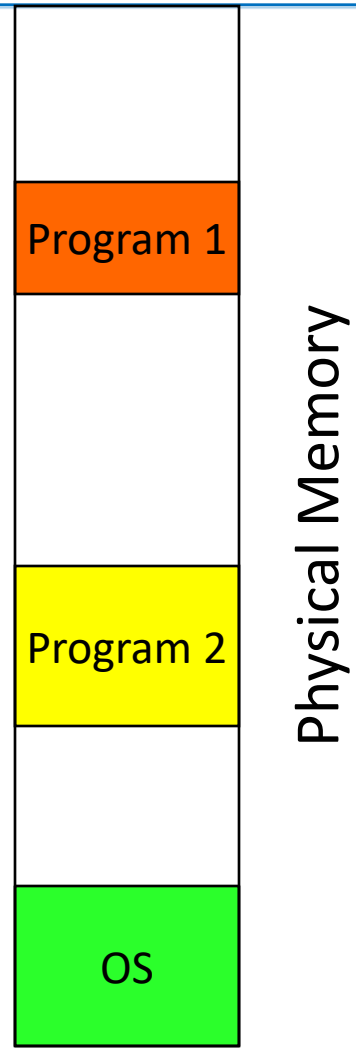
Dynamic Address Translation

- Motivație
  - La primele mașini de calcul I/O era lent și fiecare transfer pe I/O implica și procesorul (programmed I/O)
  - Productivitate mărită dacă CPU și I/O pentru 2 sau mai multe programe erau suprapuse. Cum?

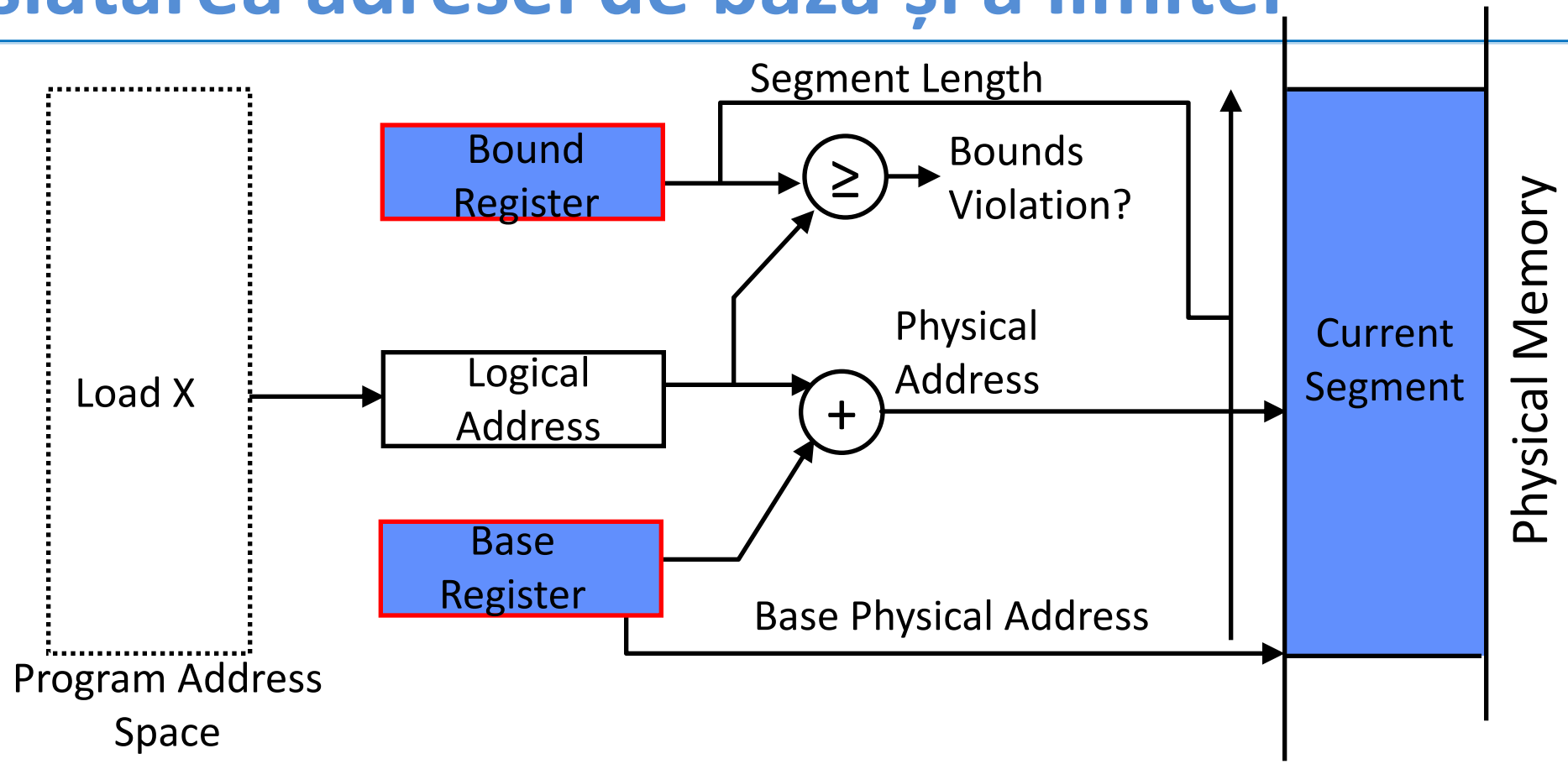
  -> multiprogramare cu DMA , I/O devices, întreruperi
- Programe independente de locație
  - Ușurință la programare și la manangementul memoriei

    -> avem nevoie de un registru de ***bază***
- Protecție
  - Programele independente nu ar trebui să se afecteze reciproc

    -> avem nevoie de un registru ***limită***
- Programe multiple care rulează pe aceeași mașină necesită software supervizor pentru a mijloci schimbarea de context dintre programele respective

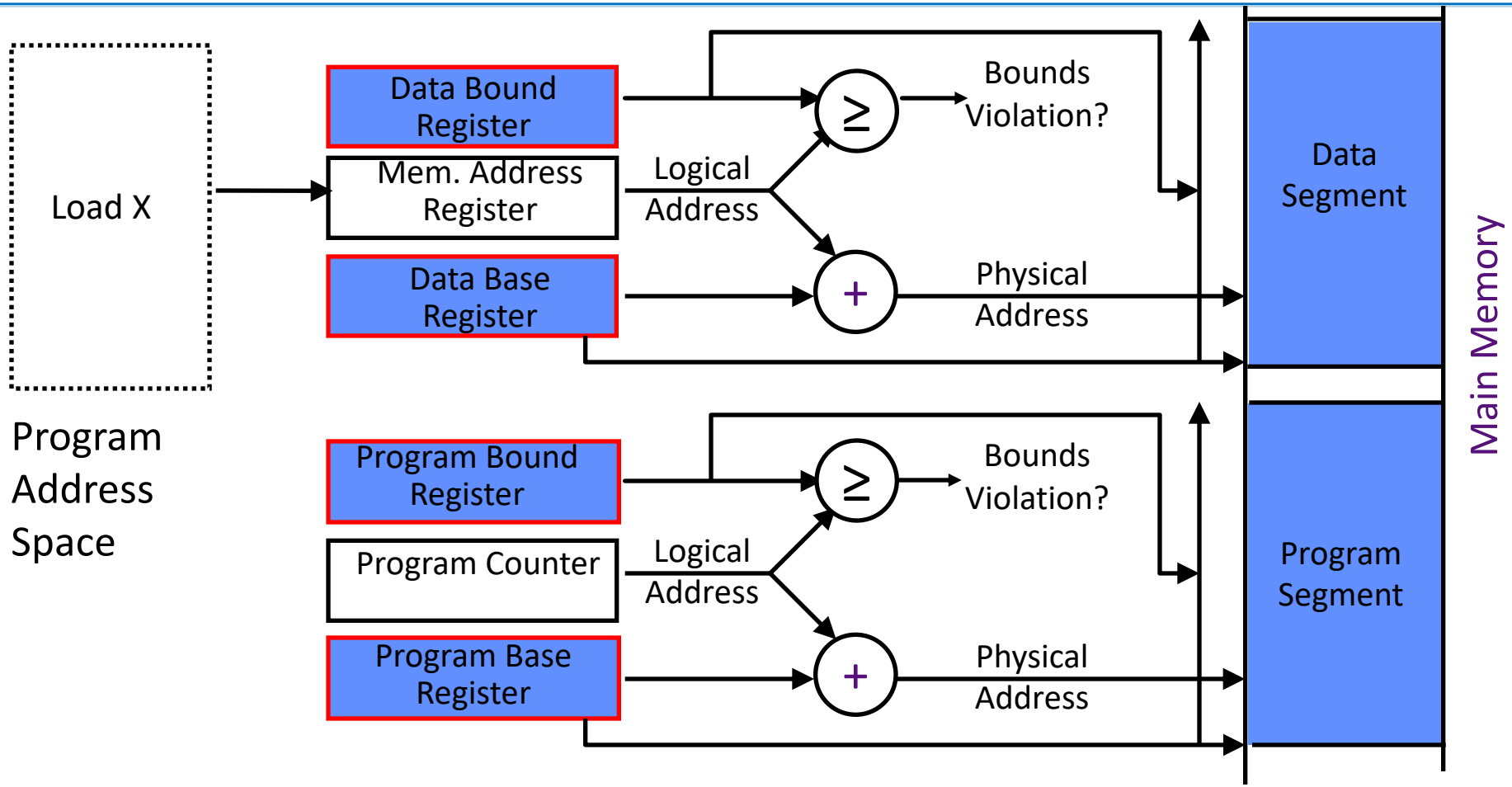


Translatarea adresei de bază și a limitei



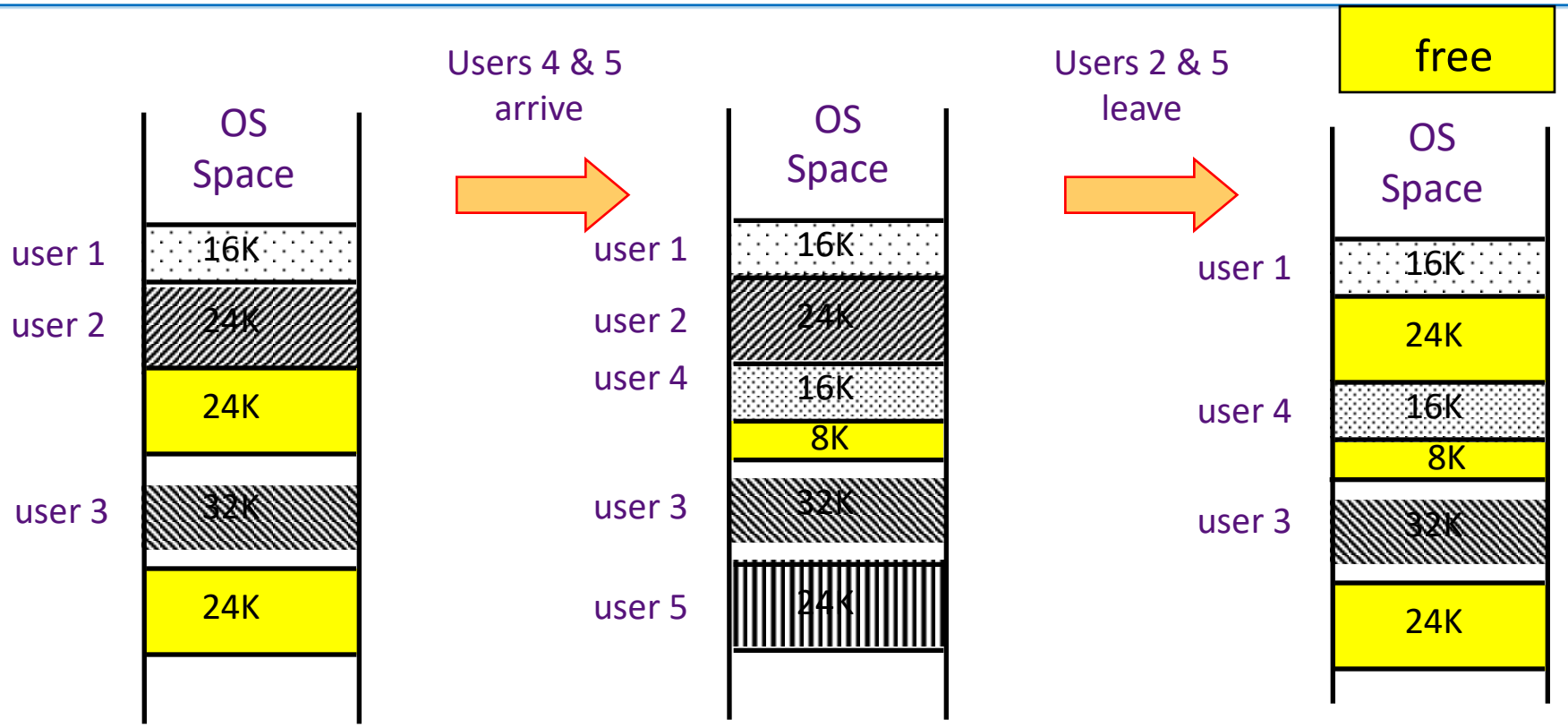
Registrele pentru bază și limită sunt vizibile/accesibile doar atunci când procesorul funcționează în modul *supervizor*

Zone separate pentru program și date



Care este avantajul acestei delimitări?

Fragmentarea memoriei



Pe măsură ce utilizatorii ”vin și pleacă”, memoria este ”fragmentată”.
Prin urmare, la un moment dat programele trebuie să fie mutate pentru a compacta memoria.

http://dilbert.com/strips/comic/2008-02-13/

Writer

```c
#include <stdio.h>

int
main (void)
{

  int *address = (int *) 0xCAFEBABE;

  printf ("Memory address is: 0x%x\n", address);

  *address = 0xDEADBEEF;

  return 0;
}
```

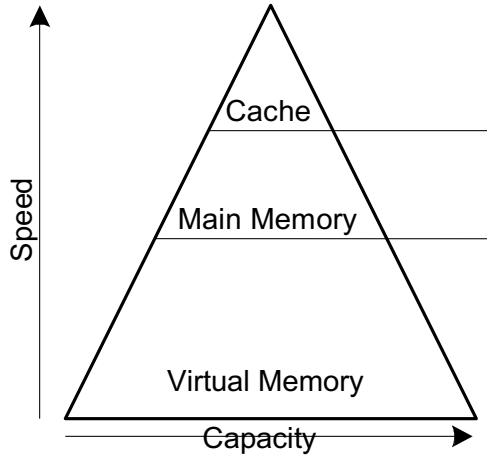
Reader

```c
#include <stdio.h>

int
main (void)
{

  int *address = (int *) 0xCAFEBABE;

  printf ("Memory address is: 0x%x\n", address);

  printf ("Content of that address is: 0x%x\n", *address);

  return 0;
}
```


Memoria Virtuală

- Creează iluzia unei memorii mai mari
- Memoria principală (DRAM) funcționează drept cache pentru hard-disk

Ierarhia memoriilor



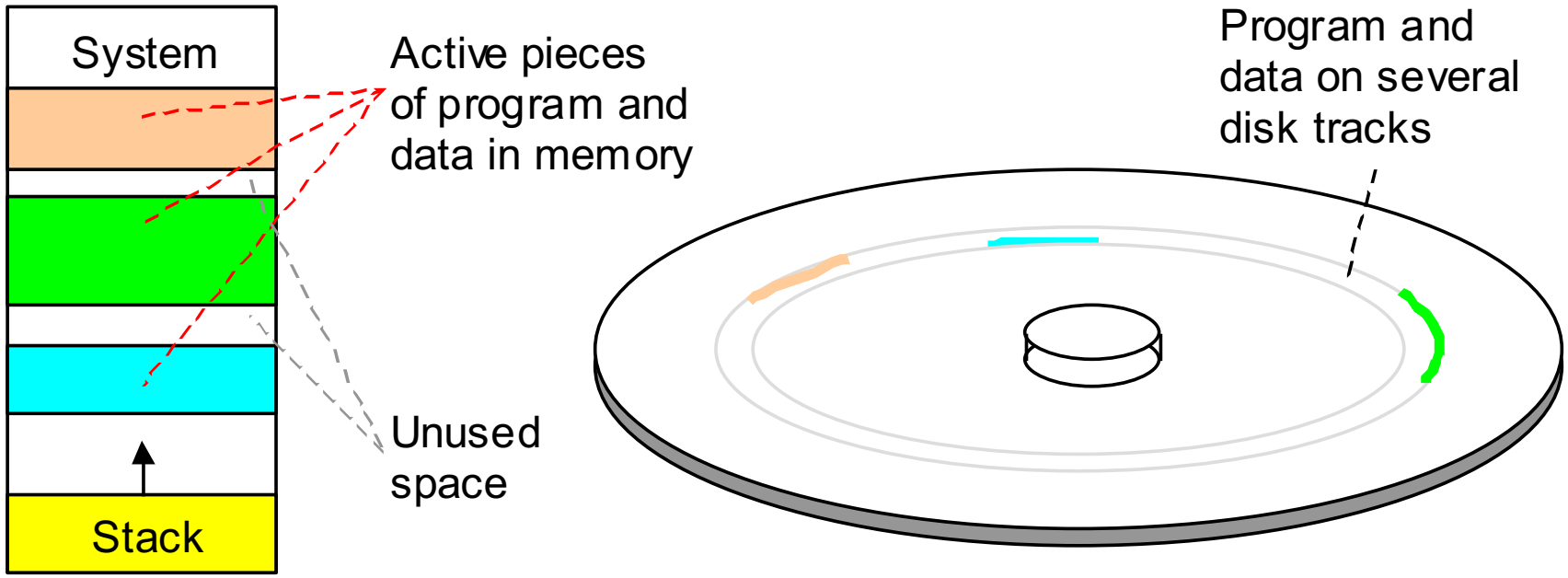
Technologie	Preț / GB	Timp acces (ns)	Lățime de bandă (GB/s)
SRAM	$10,000	1	25+
DRAM	$10	10 - 50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

- **Memorie fizică:** DRAM (Memoria principală)
- **Memoria virtuală:** Mapată peste RAM și hard drive
  - Lentă, mare, ieftină

Hard Disk

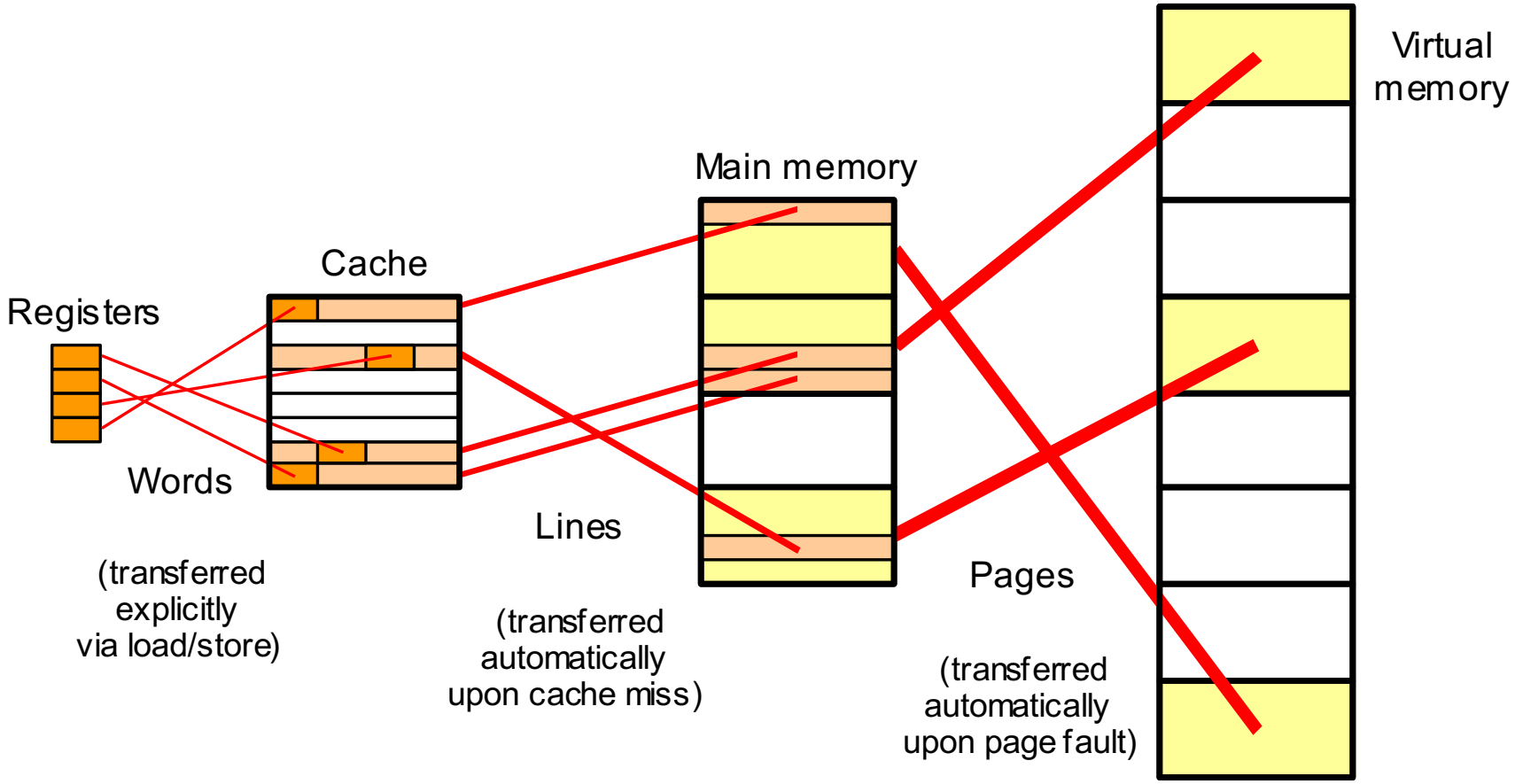
Durează milisecunde pentru a căuta locația corectă de pe disc (*seek time*).

Necesitatea unei memorii virtuale



Segmente dintr-un program în memoria principală și pe disc

Ierarhia memoriilor: tabloul complet



Fluxul de date în ierarhia de memorii

Memoria virtuală

- **Adrese virtuale**

– Programele folosesc adresele virtuale
– Întregul spațiu de adrese virtuale stocat pe hard disk
– Subset al datelor din DRAM
– CPU translatează adresele virtuale în ***adrese fizice*** (adrese DRAM)
– Datele care nu sunt în DRAM sunt luate de pe hard-drive

- **Protecția memoriei**

– Fiecare program are propria mapare de la adrese virtuale la adrese fizice
– Două programe pot folosi aceleași adrese virtuale pentru a accesa date complet diferite
– Programele nu trebuie să fie ”conștiente” de faptul că există alte programe care rulează concurent pe aceeași mașină
– Un program (sau virus) nu poate corupe memoria folosită de alt program

http://dilbert.com/strips/comic/2008-02-14/

Analogia cache/memorie virtuală

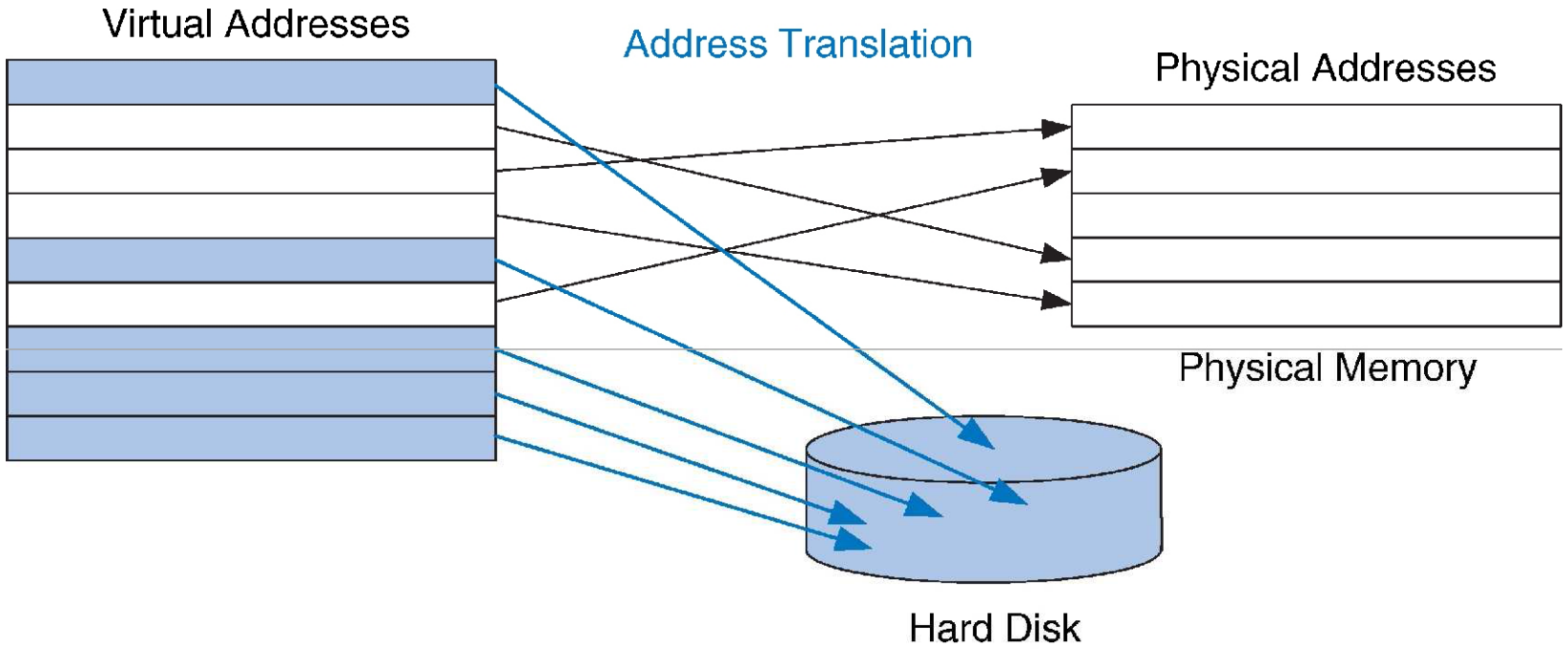
Cache	Virtual Memory
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Tag	Virtual Page Number

Memoria fizică joacă rolul de cache pentru memoria virtuală

Definiții

- **Dimensiunea paginii:** cantitatea de memorie transferată de la hard-disk către DRAM într-o singură tranzacție
- **Translatarea adreselor:** determinarea adresei fizice din cea virtuală
- **Tabela de pagini:** lookup table folosit pentru translatarea adreselor virtuale în adrese fizice

Adrese fizice și virtuale



Majoritatea acceselor fac hit în memoria fizică

Dar, programele au capacitatea memoriei virtuale

Sisteme cu memorie paginată

- Adresa generată de procesor poate fi împărțită în:

Page Number	Offset

- Tabela de pagini conține adresa fizică pentru începutul fiecărei pagini

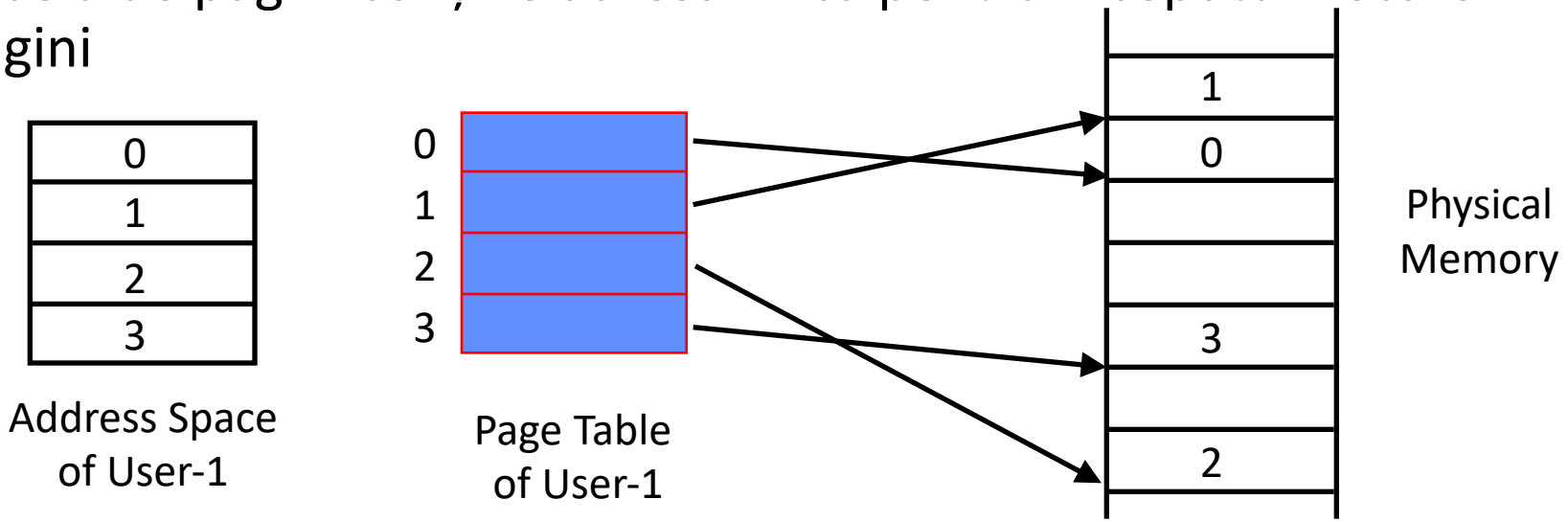
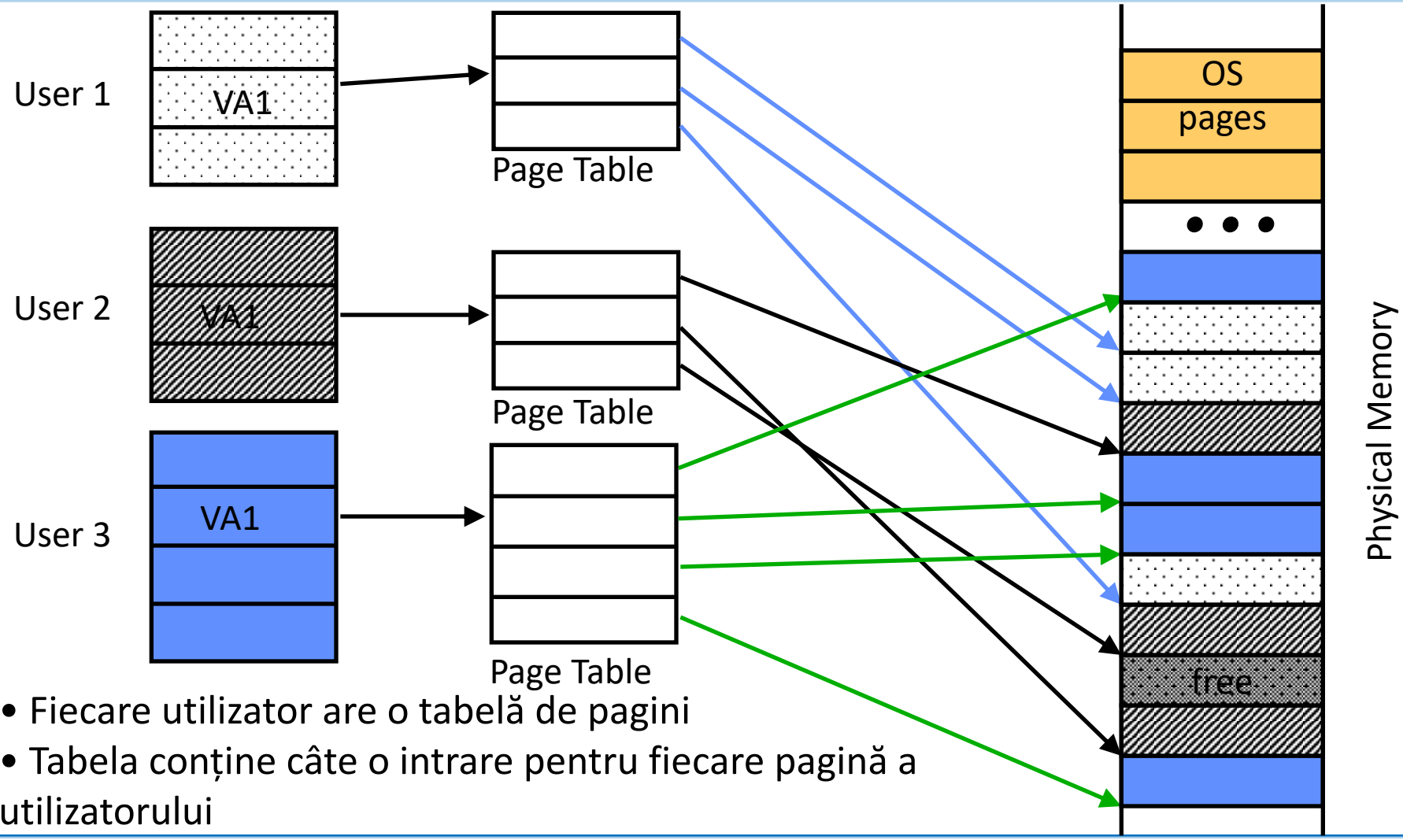


Tabela de pagini permite stocarea paginilor unui program în zone de memorie care nu sunt contigue.

Spațiu de adrese privat pentru fiecare utilizator



- Fiecare utilizator are o tabelă de pagini
- Tabela conține câte o intrare pentru fiecare pagină a utilizatorului

Unde ar trebui să rezide tabelele de pagini?

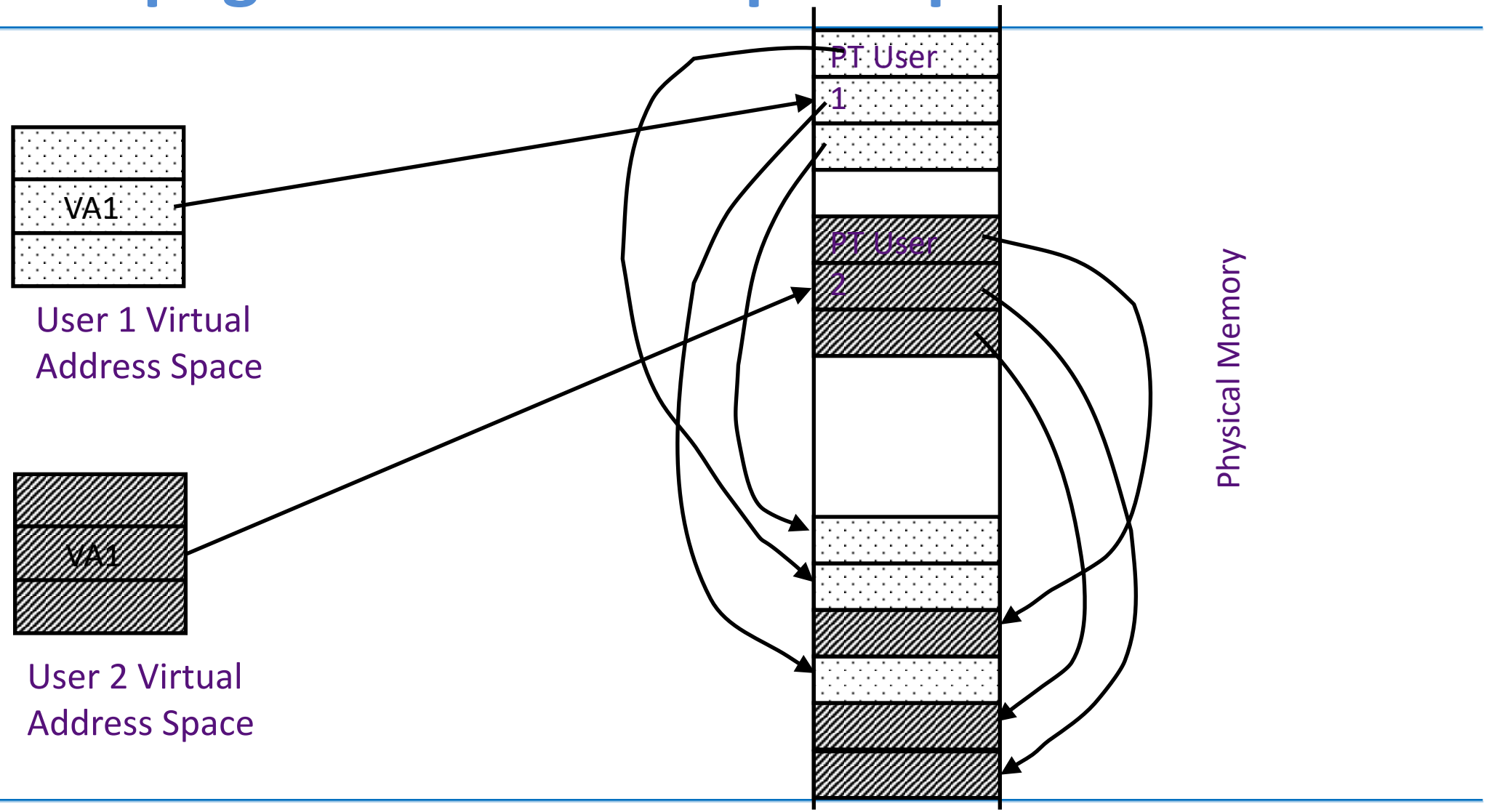
- Spațiul necesar pentru tabele de pagini (PT) este proporțional cu spațiul de adresă, numărul de utilizatori, etc.

  $\Rightarrow$ *Prea mare pentru a fi ținut în registrele generale*

- Idee: Ține PT în memoria principală
  - Are nevoie de o referință la memorie pentru a citi adresa de bază și alta pentru a accesa un cuvânt de date (de fiecare dată)

    $\Rightarrow$ *dublează numărul de accese la memorie!*

Tabelele de pagini în memoria principală



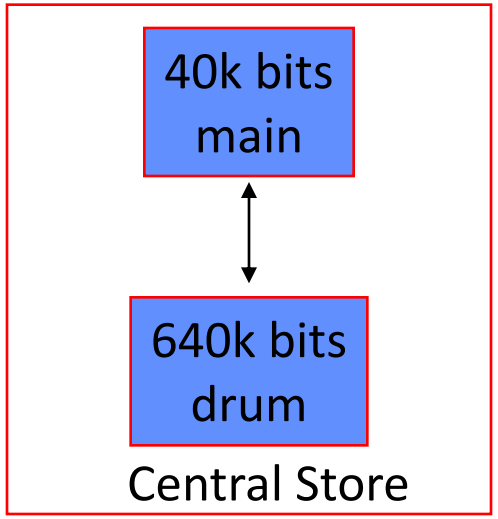
O problemă la începutul anilor șaizeci

- Existau deja multe aplicații pentru care toate datele nu puteau să intre în memoria principală (de ex. Programe de contabilitate cu mii de înregistrări)
  - *Sistemul cu memorie paginată reducea fragmentarea dar tot necesita ca întreg programul să rezide în memoria principală*

Manual Overlays

- Presupunem că o instrucțiune poate accesa orice adresă din memoria principală

- *Metoda 1:* programatorul ține evidența tuturor adreselor din memoria princ. Și inițiază un transfer I/O doar când este necesar
  - *Dificil, predispus la erori!*
- *Metoda 2:* inițializare automată a unui tranfer I/O prin translatarea software a adreselor
  - *Brooker's interpretive coding, 1960*
  - *Ineficient!*



Ferranti Mercury
1956

Nu e doar o tehnică de "magie neagră" antică, este folosită și la procesoarele IBM Cell din Playstation 3, de exemplu.

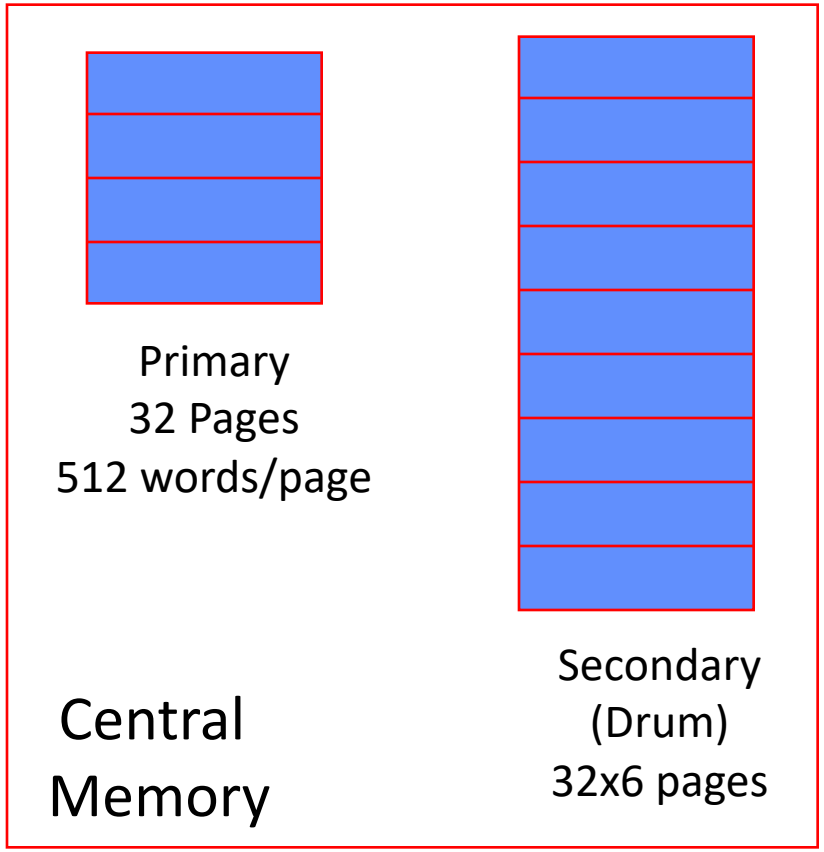
Paginare la cerere pentru Atlas (1962)

"A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor."

Tom Kilburn

Memoria primară servește drept "cache" Pentru memoria secundară

Utilizatorul "vede" 32 x 6 x 512 cuvinte pentru stocare



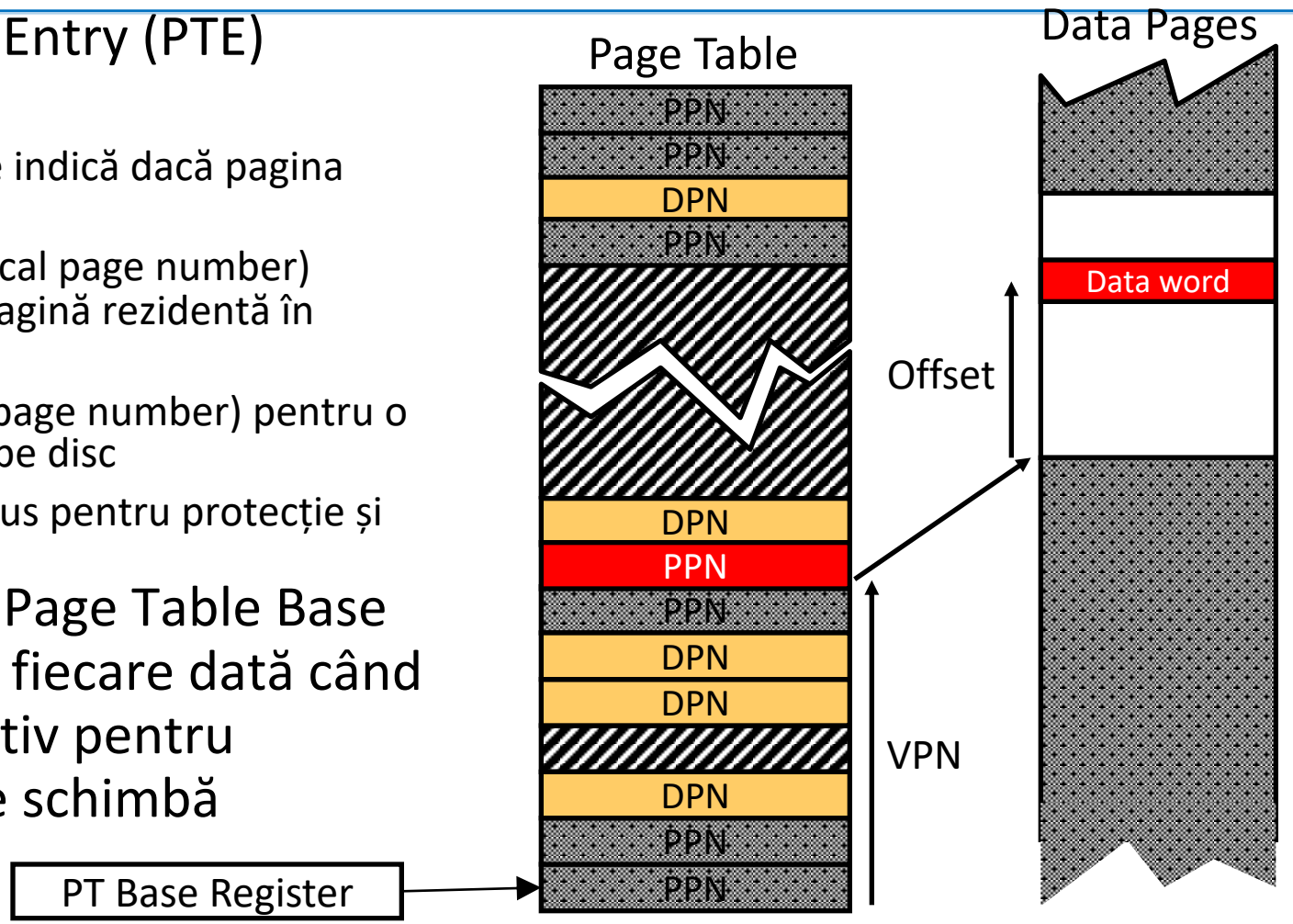
Algoritmul de paginare pentru Atlas

La un page fault:

- Se inițializează transferul unei pagini libere noi
- Este actualizat Page Address Register (PAR)
- Dacă nu a mai rămas nici o pagină liberă, selectează o pagină pentru a fi înlocuită (bazat pe utilizare)
- Pagina înlocuită este scrisă în memorie
  - Pentru a minimiza latența de acces la memorie, se va selecta prima pagină goală întâlnită în memorie
- Tabela de pagini este actualizată pentru a indica la noua locație a paginii din memorie

Tabelă liniară de pagini

- Page Table Entry (PTE) conține:
  - Un bit care indică dacă pagina există
  - PPN (physical page number) pentru o pagină rezidentă în memorie
  - DPN (disk page number) pentru o pagină de pe disc
  - Biți de status pentru protecție și utilizare
- SO setează Page Table Base Register de fiecare dată când procesul activ pentru utilizator se schimbă

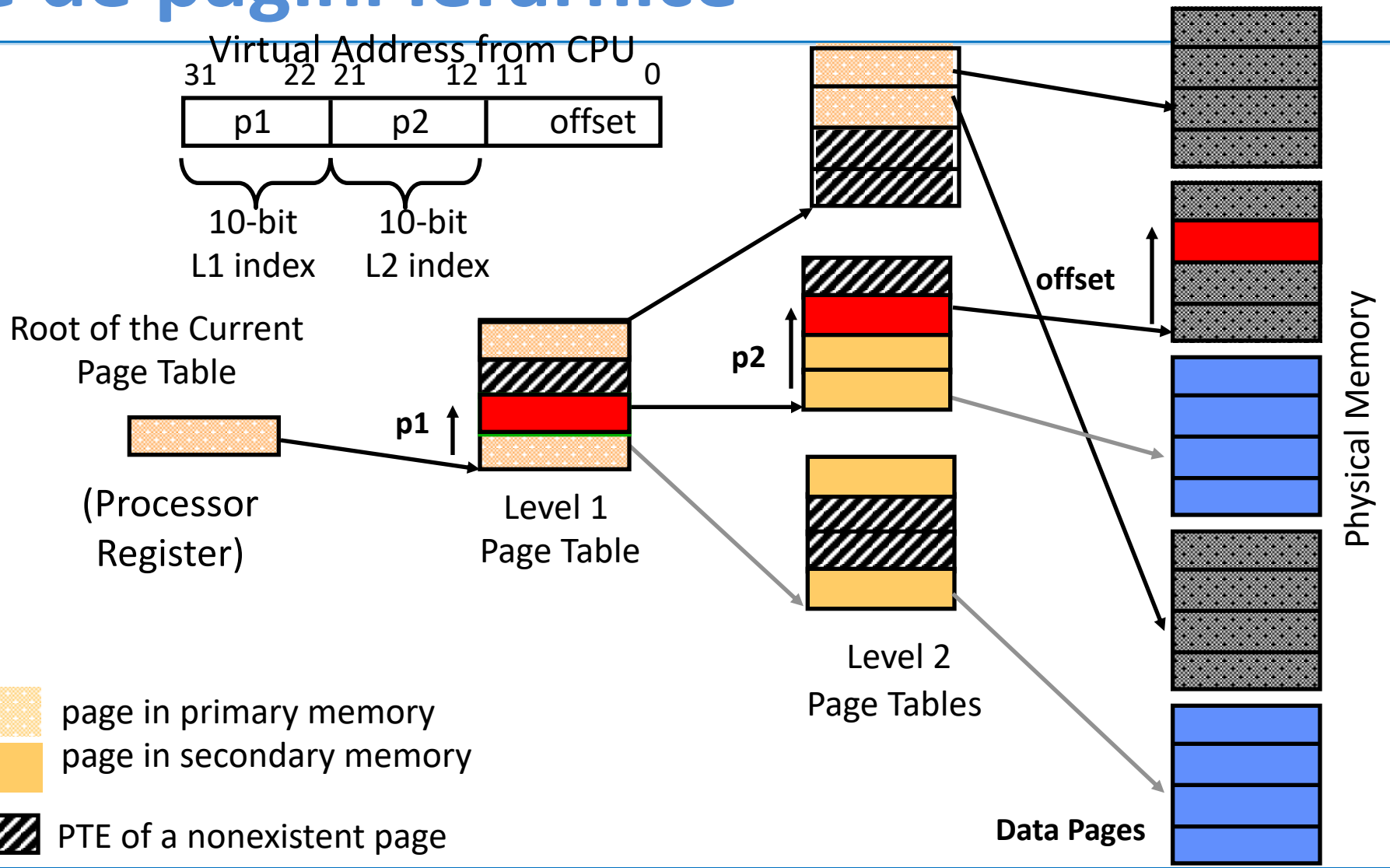


Dimensiunea unei tabele de pagini liniare

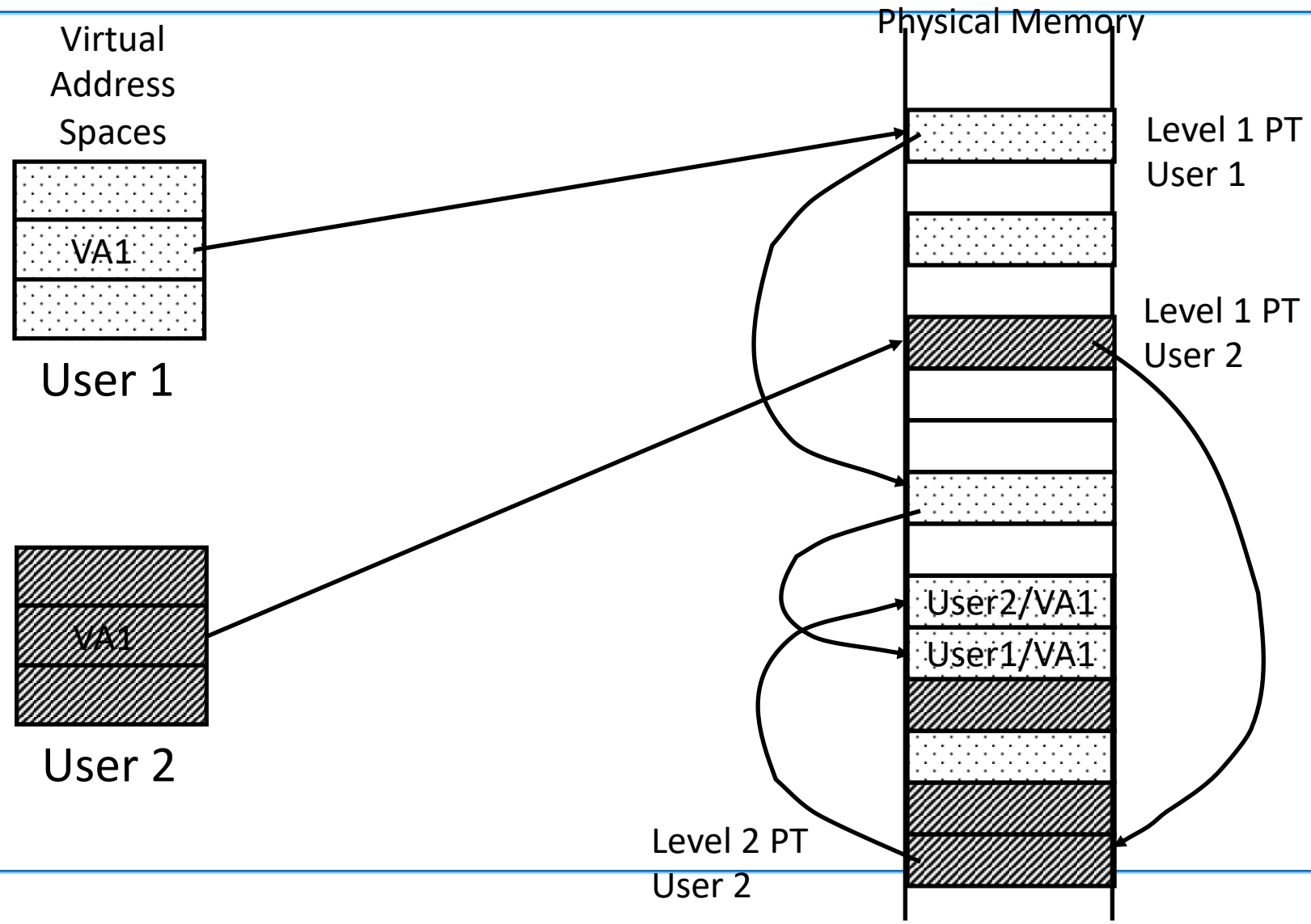
- Cu adrese pe 32 de biți, pagini de 4KB & PTE de 4 octeți:
  - 220 PTE, 4 MB tabelă pagini per utilizator
  - 4GB de swap necesari pentru a face back-up la tot spațiul de adrese virtuale
- Pagini mai mari?
  - Fragmentare internă (Nu e folosită toată memoria dintr-o pagină)
  - Penalizare pentru dimensiune (trebuie mai mult timp pentru a citi de pe disc)
- Cum scalează asta la un spațiu de adrese pe 64 de biți???
  - Chiar și o pagină de 1MB ar avea nevoie de 244 PTE-uri de 8 biți (=35TB!)

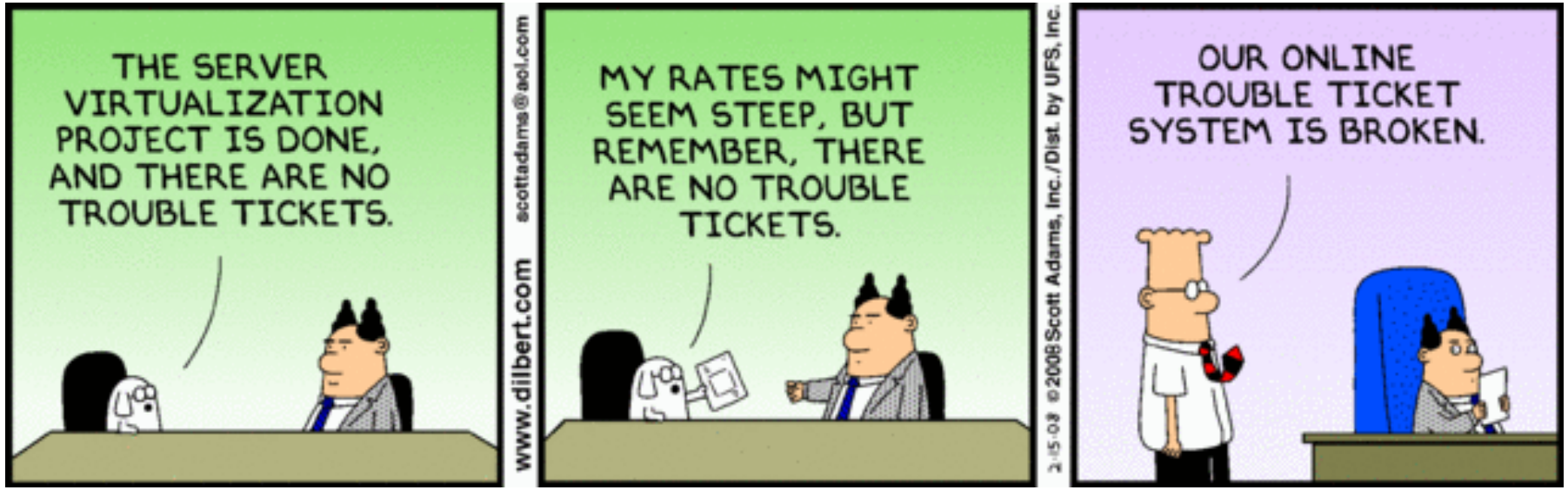
Care este ideea salvatoare?

Tabele de pagini ierarhice



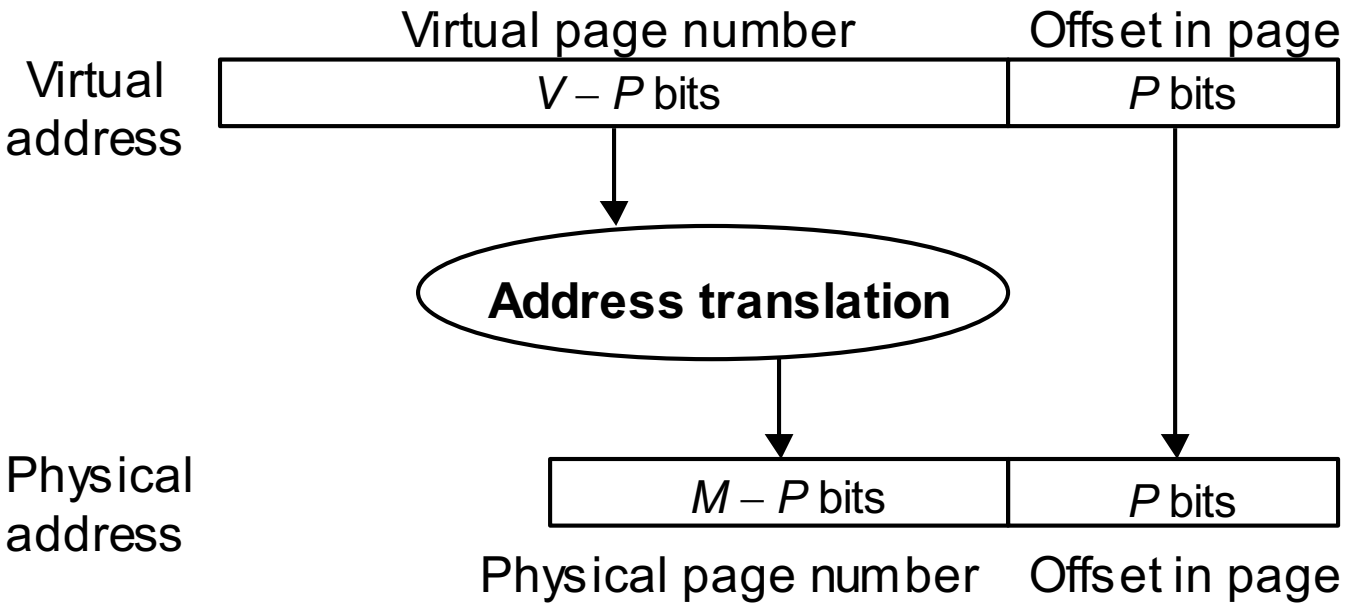
Tabele de pagini pe două niveluri în memoria fizică





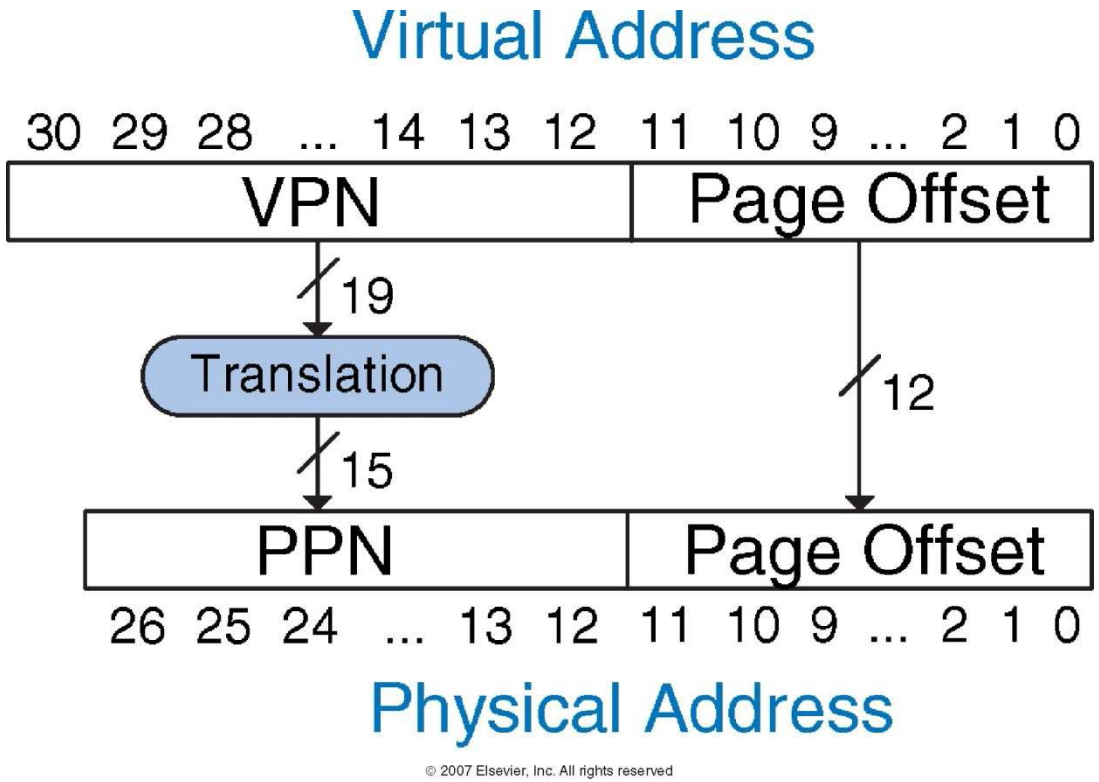
http://dilbert.com/strips/comic/2008-02-15/

Translatarea adreselor în memoria virtuală

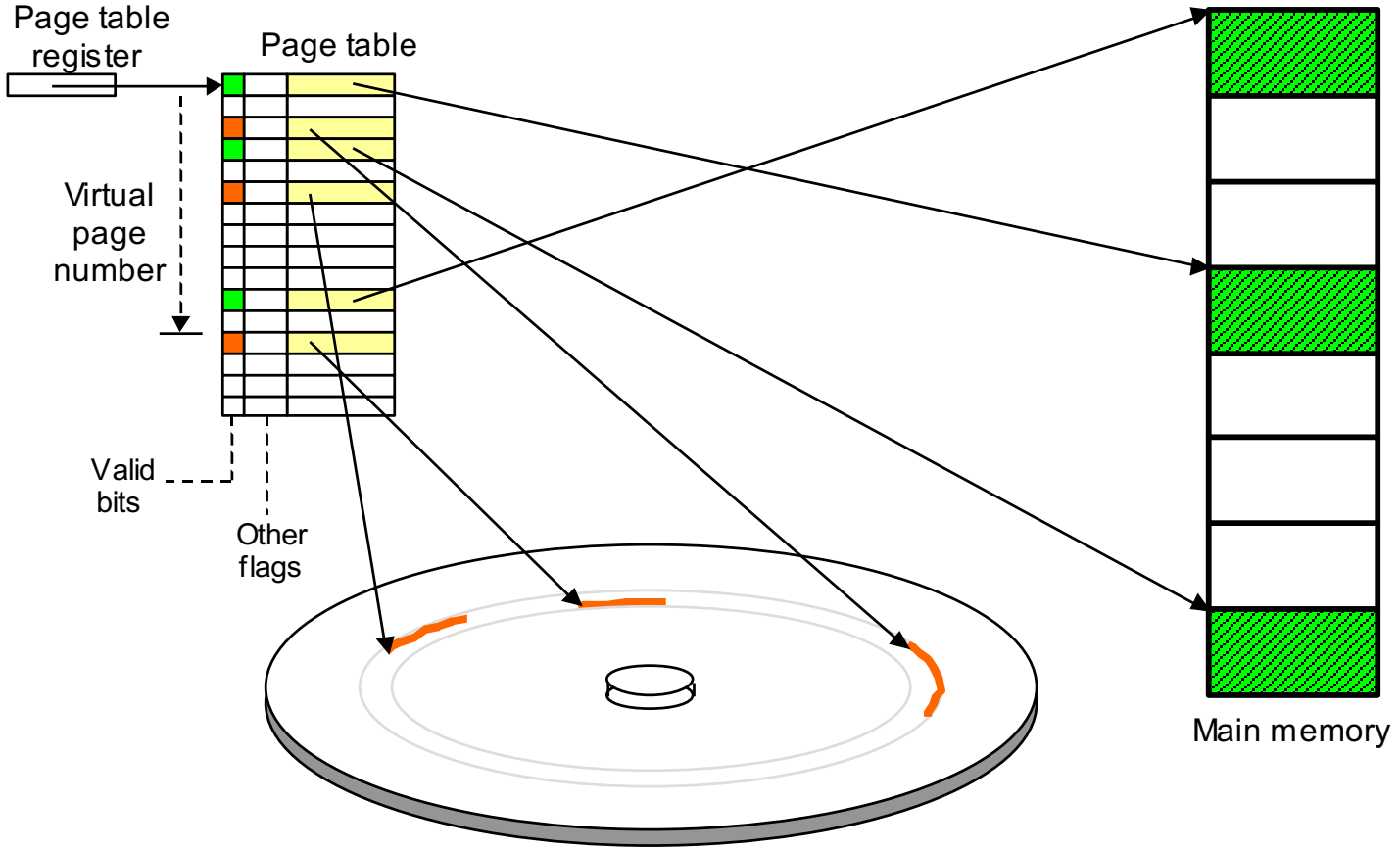


Parametrii translatării virtual-fizic

Translatarea adreselor



Tabele de pagini si translatarea adreselor



Rolul tabelei de pagini în procesul translației adreselor virtuale în adrese fizice

Exemplu memorie virtuală

- **Sistem:**
  - Dimensiunea mem. virtuale: 2 GB = 2^{31} octeți
  - Dimensiunea memoriei fizice: 128 MB = 2^{27} octeți
  - Dimensiunea paginii: 4 KB = 2^{12} octeți

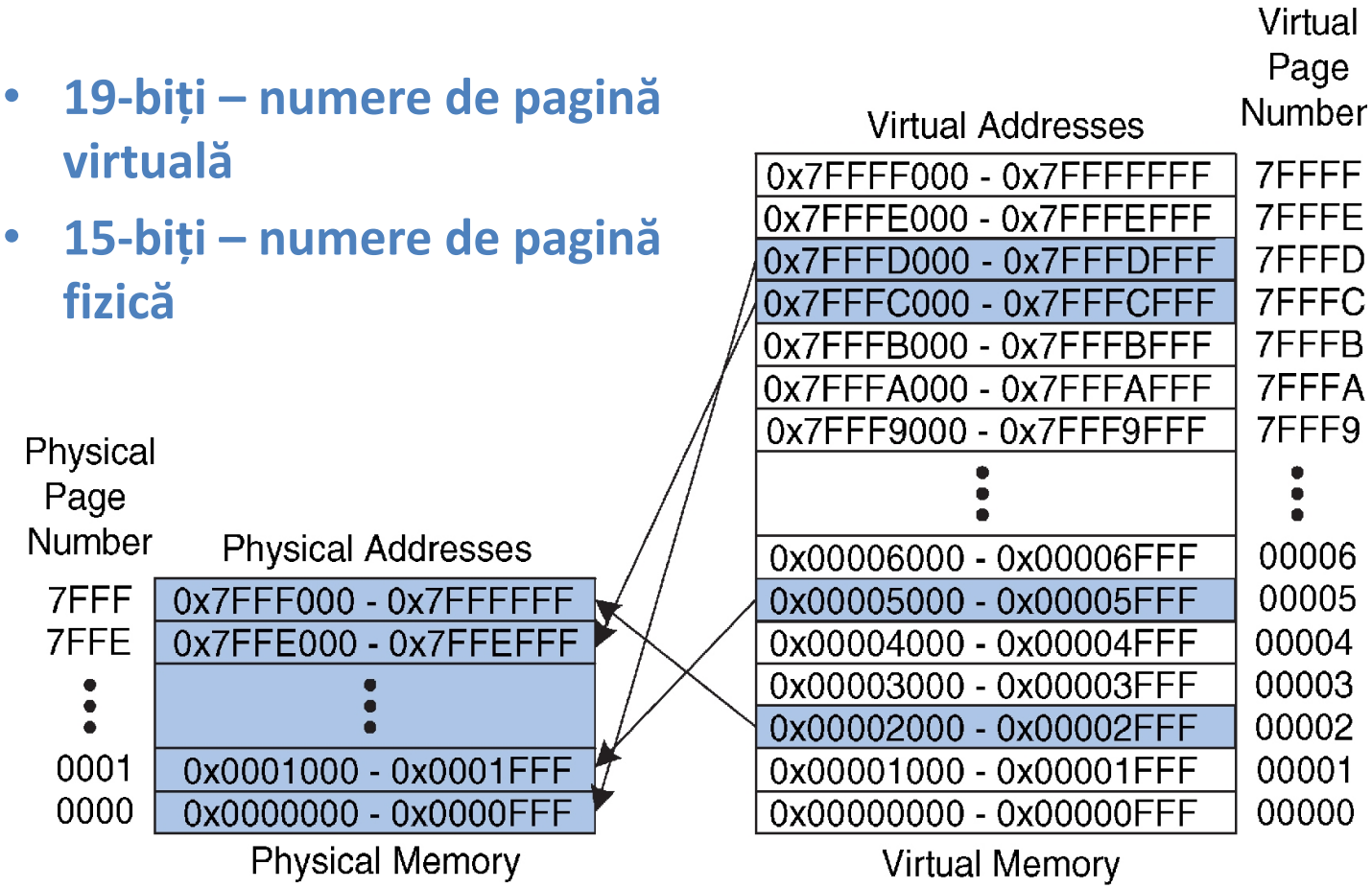
Exemplu memorie virtuală

- **Sistem:**
  - Memoria virtuală: 2 GB = 2^{31} octeți
  - Memoria fizică: 128 MB = 2^{27} octeți
  - Dimensiune pagină: 4 KB = 2^{12} octeți
- **Organizare:**
  - Adrese virtuale: **31** biți
  - Adrese fizice: **27** biți
  - Offset pagină: **12** biți

  - # pagini virtuale = $2^{31}/2^{12} = \mathbf{2^{19}}$ (VPN = 19 biți)
  - # pagini fizice = $2^{27}/2^{12} = \mathbf{2^{15}}$ (PPN = 15 biți)

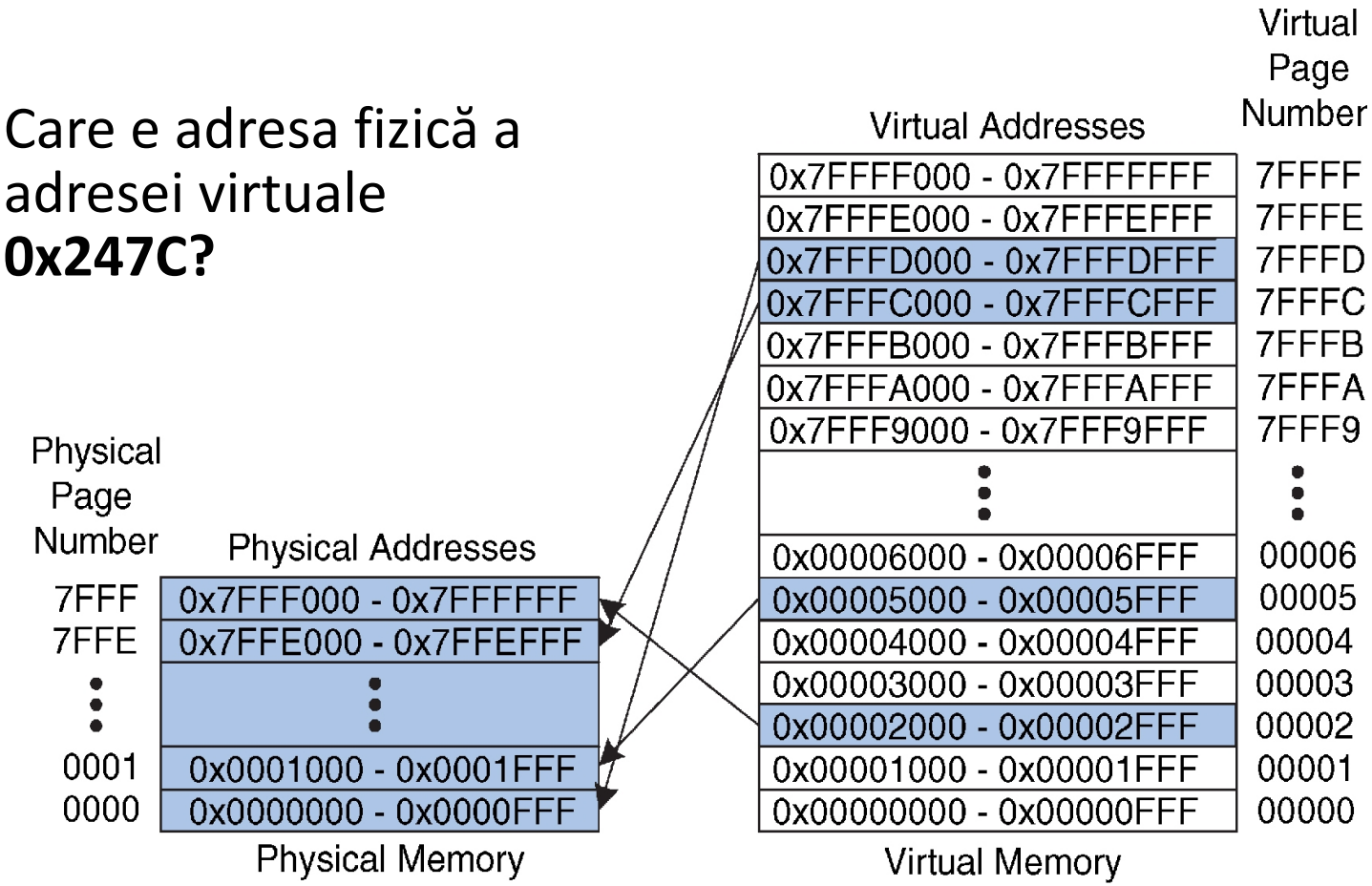
Exemplu memorie virtuală

- 19-biți – numere de pagină virtuală
- 15-biți – numere de pagină fizică



Exemplu memorie virtuală

Care e adresa fizică a adresei virtuale **0x247C?**



Exemplu memorie virtuală

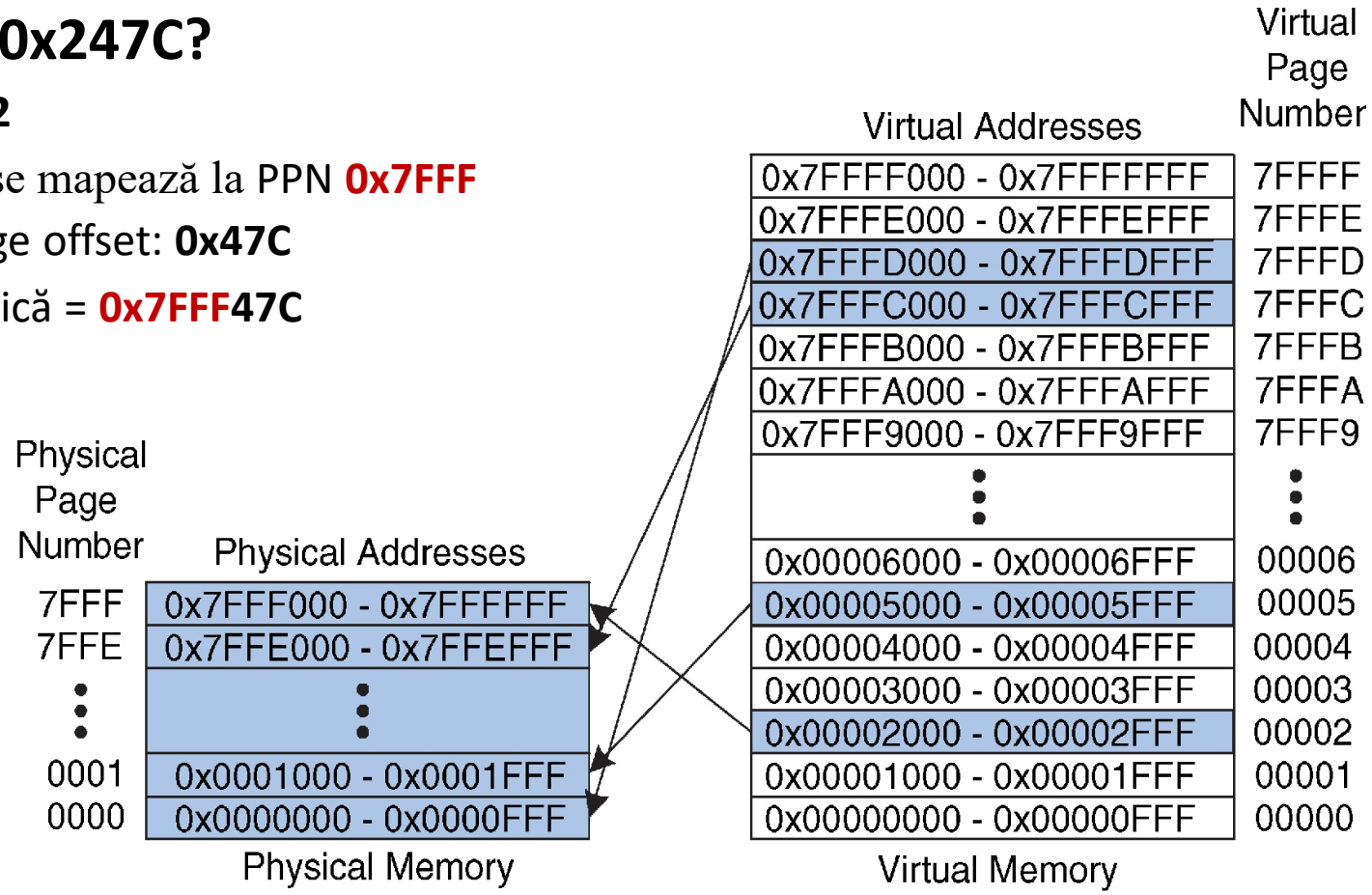
Care e adresa fizică a adresei virtuale **0x247C?**

–VPN = **0x2**

–VPN 0x2 se mapează la PPN **0x7FFF**

–12-bit page offset: **0x47C**

–Adresa fizică = **0x7FFF47C**

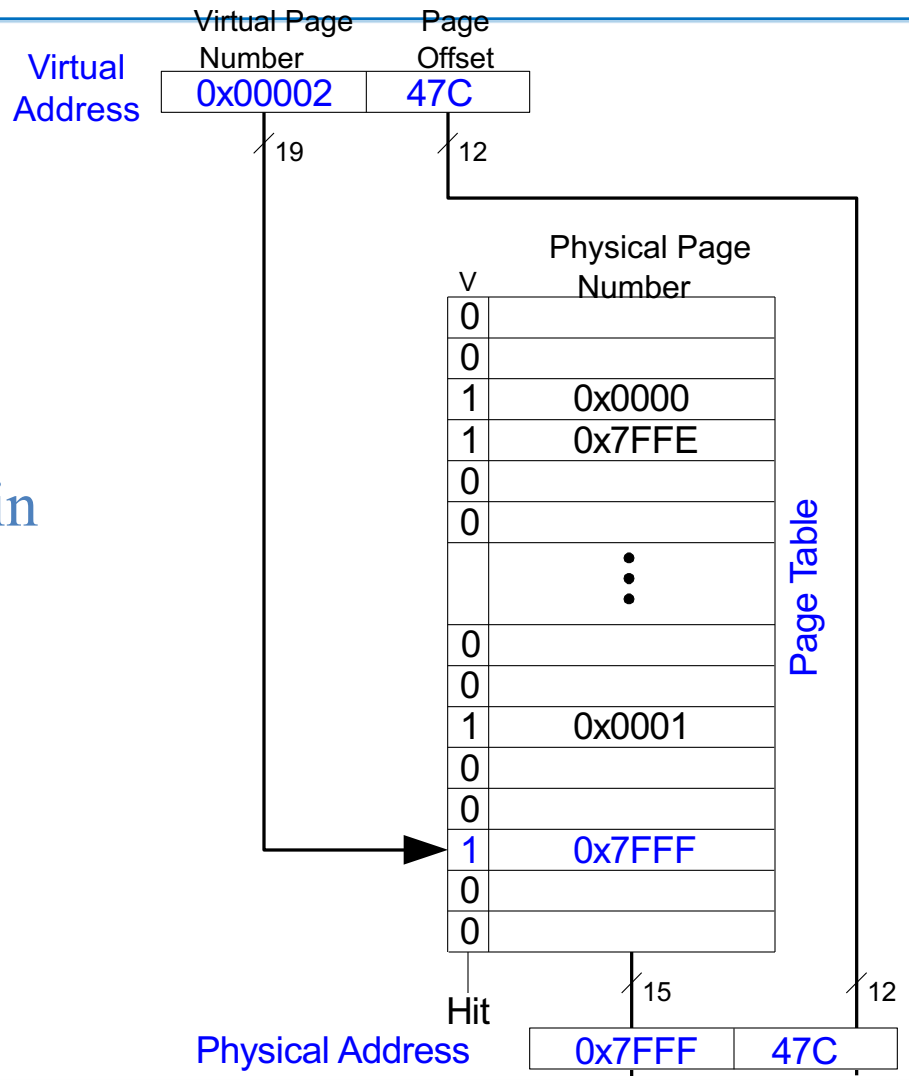


Cum se face translatarea?

- **Tabela de pagini**
  - Intrare pentru fiecare pagină virtuală
  - Câmpuri pentru fiecare intrare:
    - **Bit validitate:** 1 dacă pagina este în memoria fizică
    - **Numărul paginii fizice:** unde este localizată pagina

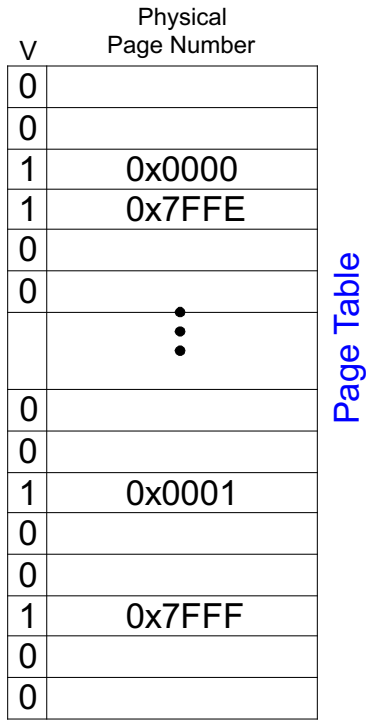
Exemplu paginare

VPN este indexul din tabela de pagini



Exemplul 1

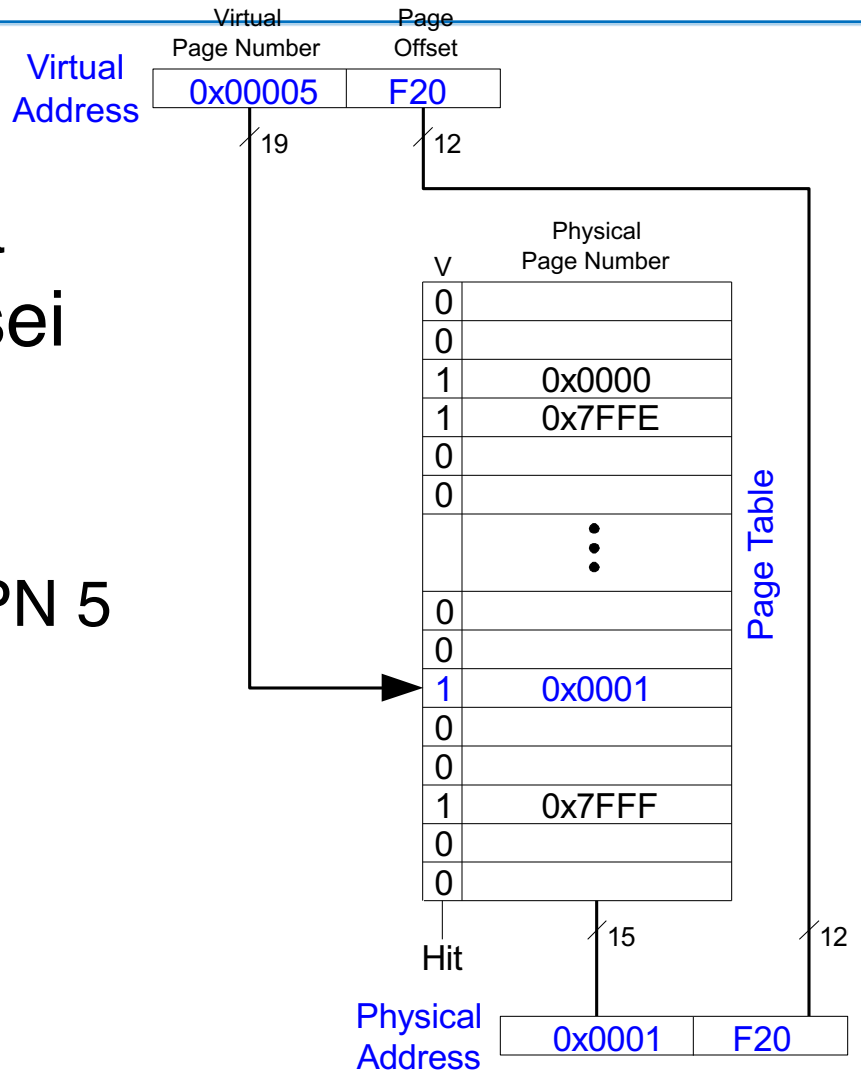
Care este adresa fizică corespunzătoare adresei virtuale **0x5F20**?



Exemplul 1

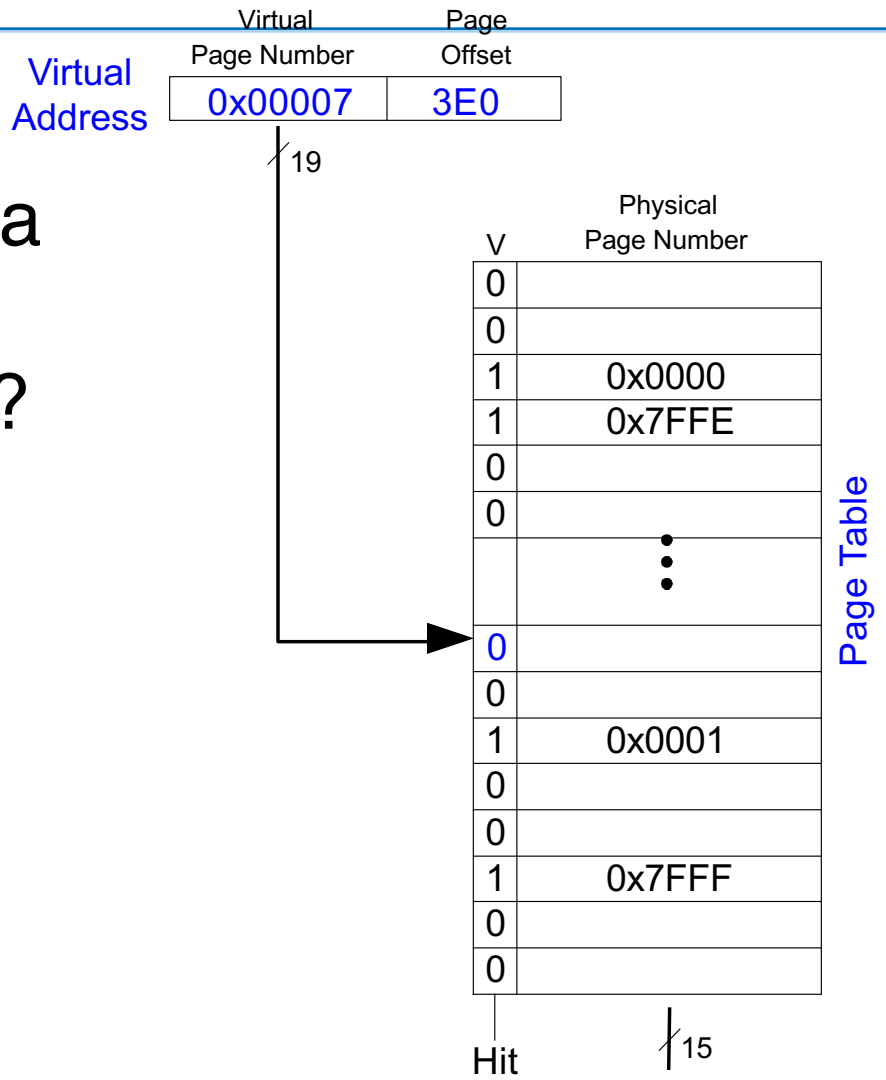
Care este adresa fizică corespunzătoare adresei virtuale **0x5F20**?

- VPN = **5**
- Intrarea 5 în page table VPN 5 => pagina fizică **1**
- Adresa fizică: **0x1F20**



Exemplul 2

Care este adresa fizică a adresei virtuale **0x73E0**?



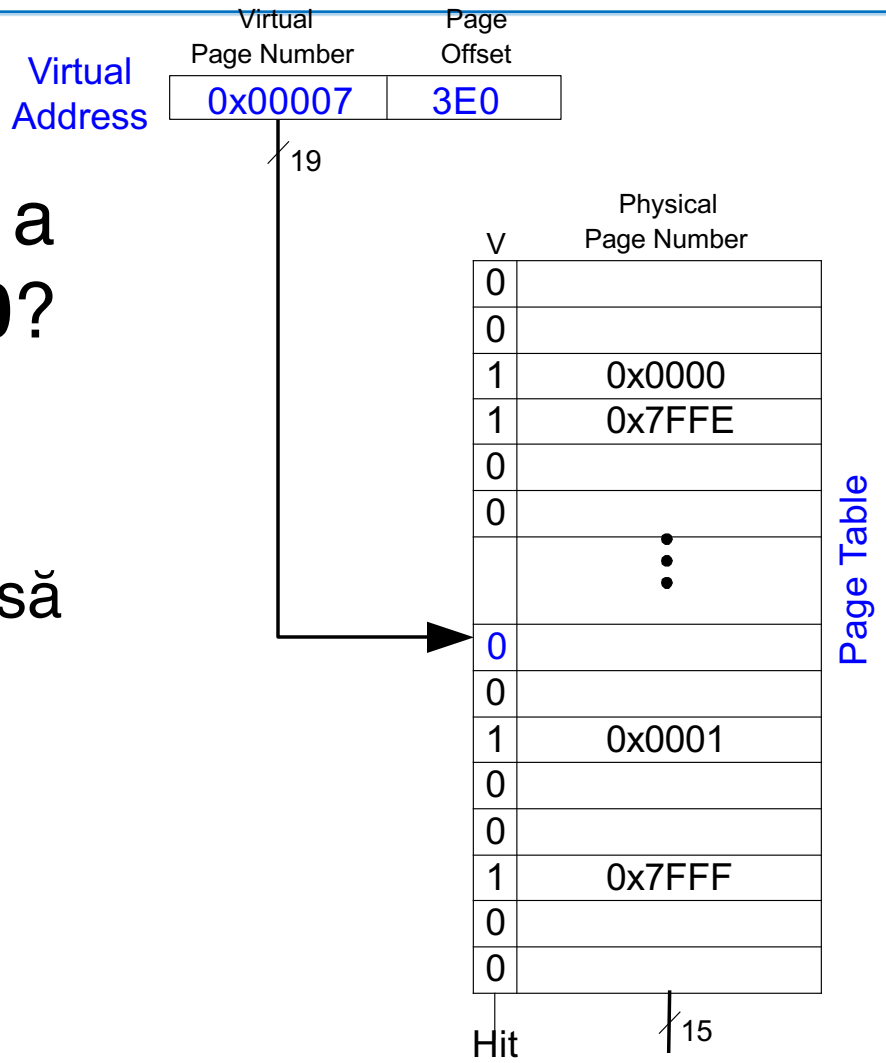
Exemplul 2

Care este adresa fizică a adresei virtuale **0x73E0**?

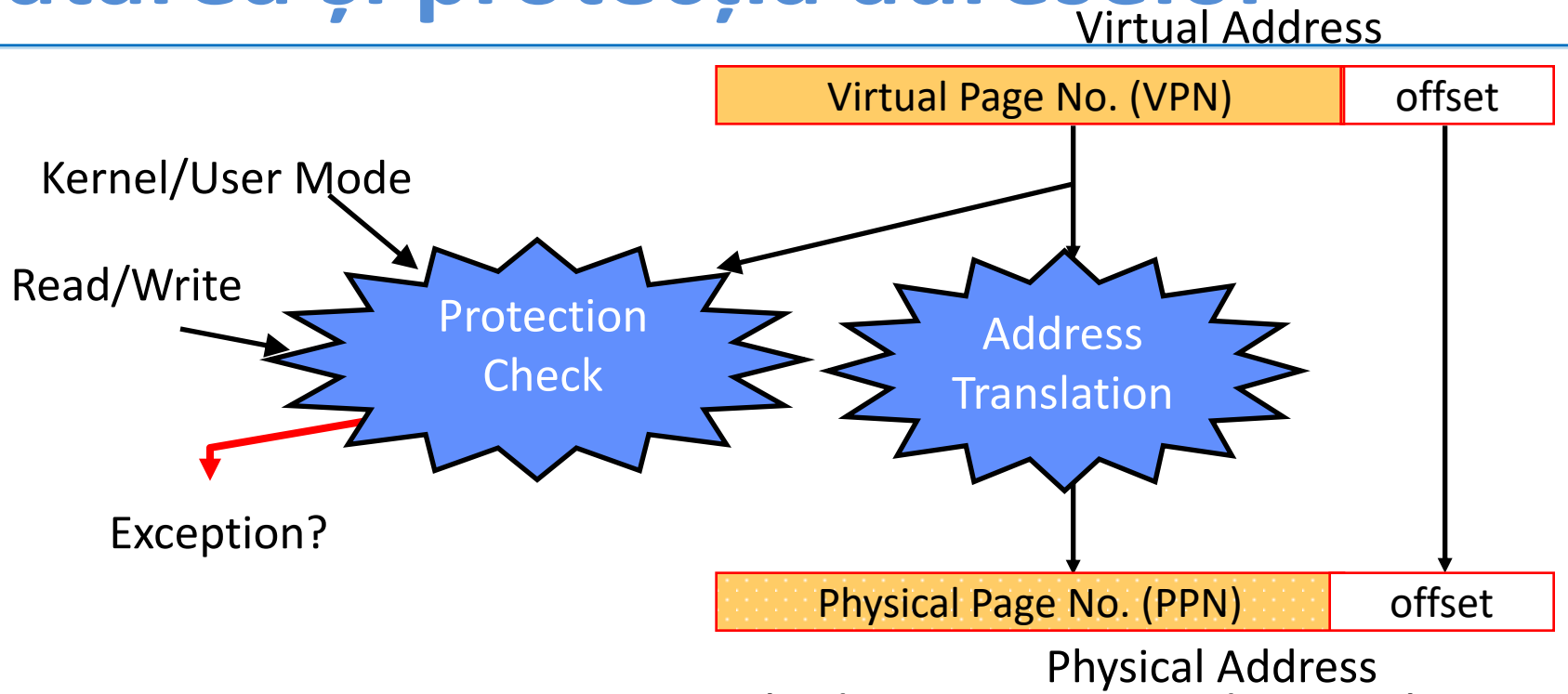
–VPN = **7**

–Intrarea 7 este invalidă

–Pagina virtuală trebuie adusă în memoria fizică de pe disc



Translatarea și protecția adreselor



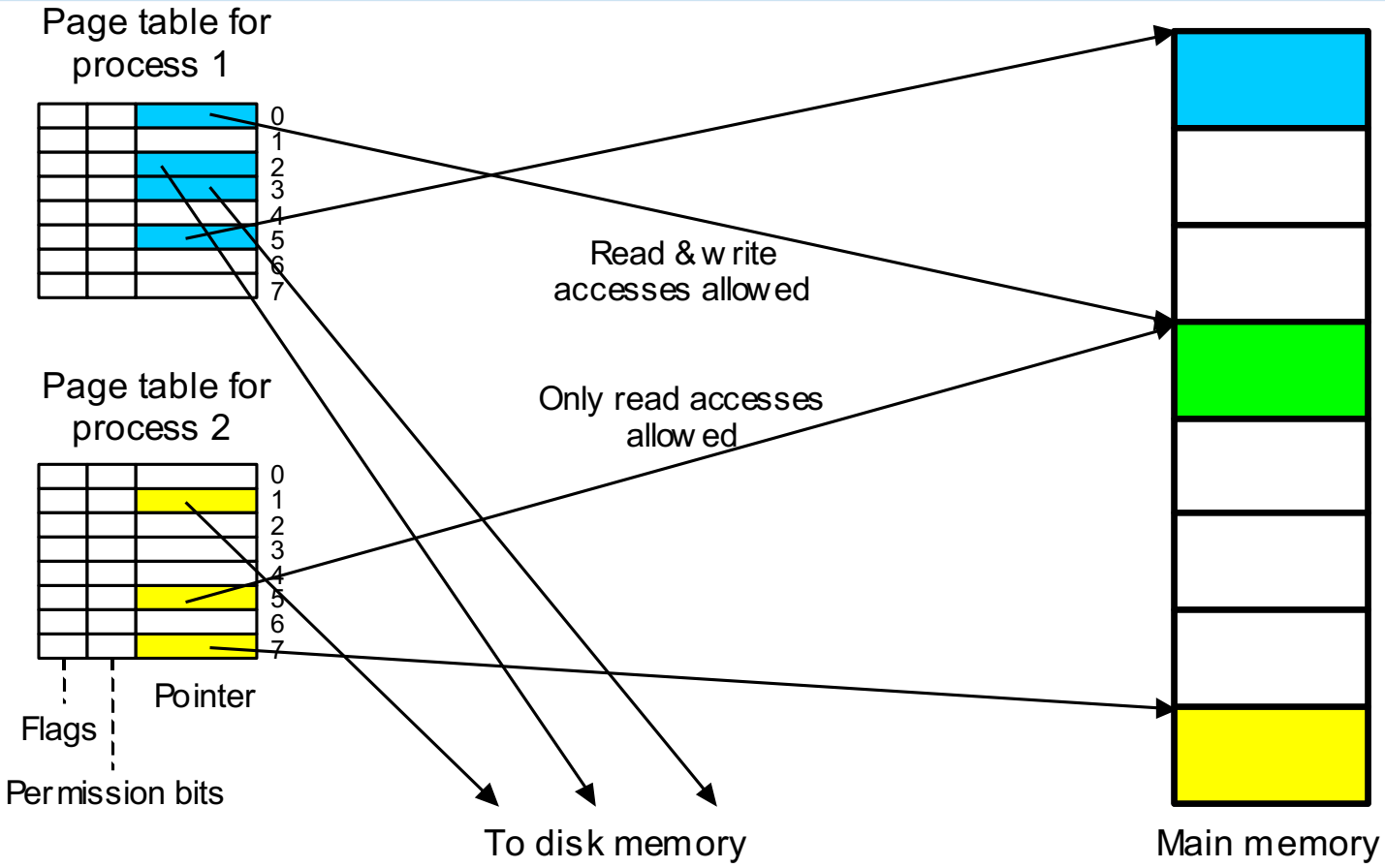
- Fiecare instrucțiune și acces la date are nevoie de translatarea adreselor și verificări la protecție

O mașină virtuală bună trebuie să fie rapidă (~ un ciclu) și eficientă la ocuparea spațiului

Protecția memoriei principale

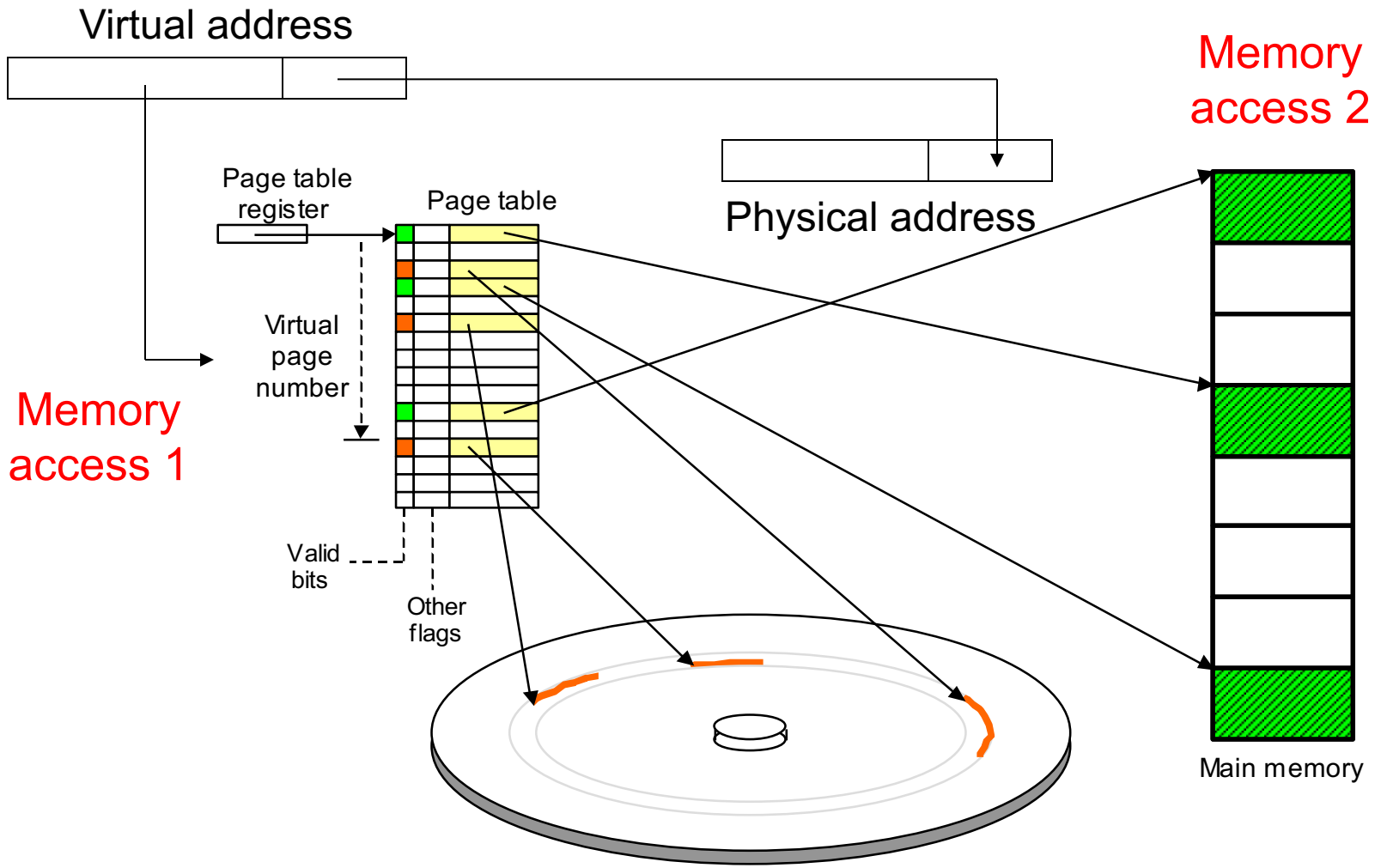
- Procese multiple (programe) rulează concomitent pe o mașină de calcul
- Fiecare proces are tabela de pagini proprie
- Fiecare proces poate folosi întregul spațiu de adrese virtuale
- Un proces poate accesa doar paginile fizice mapate în propria tabelă de pagini

Protecția și partajarea memoriei virtuale



Memoria virtuală facilitează partajarea datelor și protecția memoriei

Penalizarea de latență pentru memoria virtuală



Probleme cu tabela de pagini

- **Tabela de pagini este prea mare**
  - de obicei este localizată în memoria fizică
- Load/store necesită 2 accese la memoria principală:
  - Unul pentru translație (page table read)
  - Unul pentru acces date (după translație)
- Taie performanța memoriei la jumătate
  - ***Nu și dacă facem lucrurile deștept…***

Translation Lookaside Buffer (TLB)

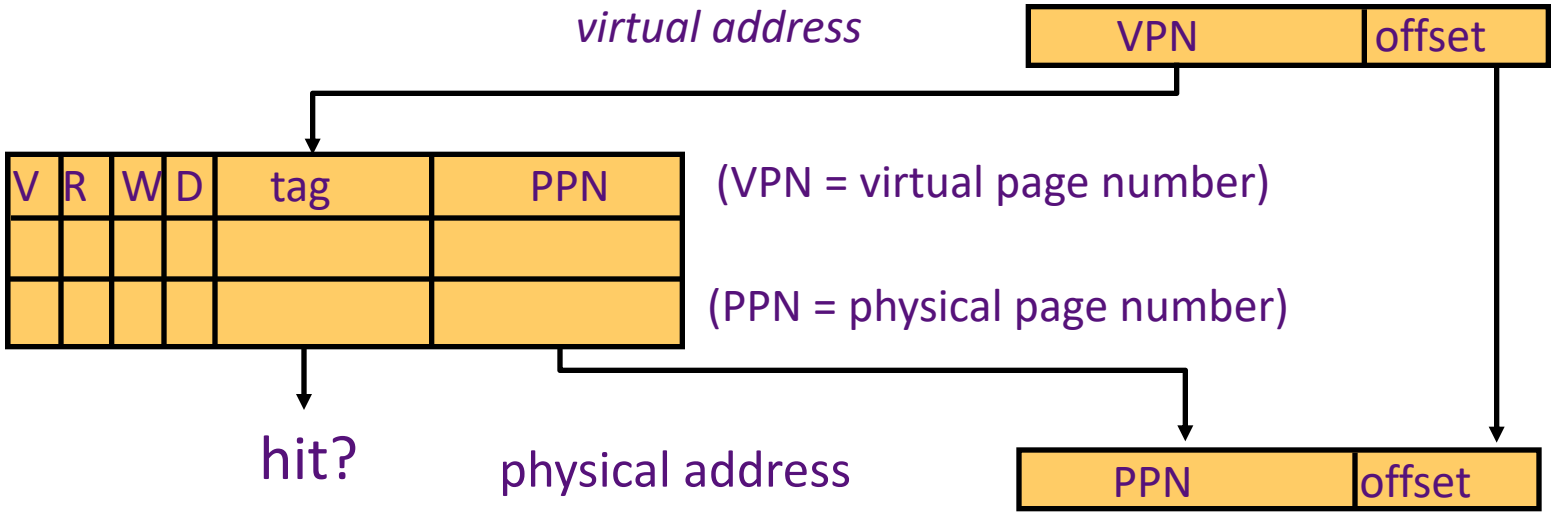
Translatarea adreselor este foarte costisitoare!

La o tabelă cu două niveluri, fiecare referință se traduce prin mai multe referințe la memorie

Soluție: *Translatarea TLB în cache*

TLB hit ⇒ *Single-Cycle Translation*

TLB miss ⇒ *Page-Table Walk to refill*



Translation Lookaside Buffer (TLB)

- Este un cache mic care conține cele mai recente translații
- Reduce numărul de accese la memorie pentru *majoritatea* operațiilor load/store (de la 2 la 1)

TLB

- Accese la tabela de pagini: localitate temporală mărită
  - Pagină de dimensiuni mari, deci load/store consecutive probabil vor accesa aceeași pagină
  - TLB
  - Mic: accesat în < 1 ciclu
  - De obicei 16 - 512 intrări
  - Complet asociativ
  - > 99 % hit rate
  - Reduce nr. accese la memorie de la 2 la 1 pentru operații load/store

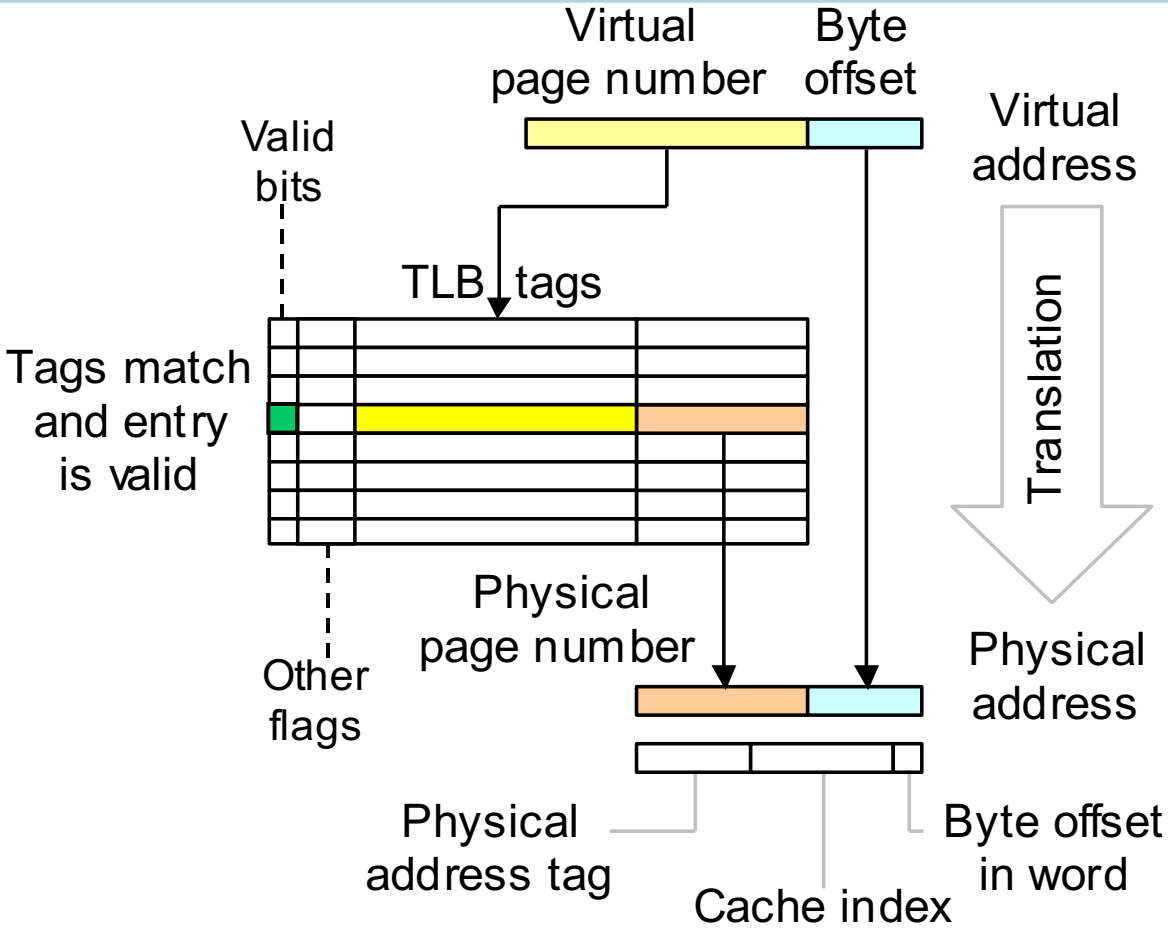
Design TLB

- De obicei 32-128 intrări, de obicei complet asociativ
  - Fiecare intrare descrie o pagină mare, deci mai puțină localitate spațială între diferitele pagini ➔ mai probabil ca două intrări să fie în conflict
  - Câteodată TLB mai mare (256-512 intrări) care sunt 4-8 way set-asociative
  - Sistemele mai mari au TLB multi-nivel (L1 și L2)
- Random or FIFO replacement policy
- TLB Reach: dimensiunea celui mai mare spațiu virtual de adresă care poate fi mapat simultan de TLB

Exemplu: 64 intrări TLB, pagini de 4KB, o pagină per intrare

TLB Reach = *64 intrări * 4 KB = 256 KB (dacă e contiguu)* ?

Translation Lookaside Buffer



Program page in virtual memory

```
     lw    $t0,0($s1)
     addi  $t1,$zero,0
L:   add   $t1,$t1,1
     beq   $t1,$s2,D
     add   $t2,$t1,$t1
     add   $t2,$t2,$t2
     add   $t2,$t2,$s1
     lw    $t3,0($t2)
     slt   $t4,$t0,$t3
     beq   $t4,$zero,L
     addi  $t0,$t3,0
     j     L
D:   ...
```

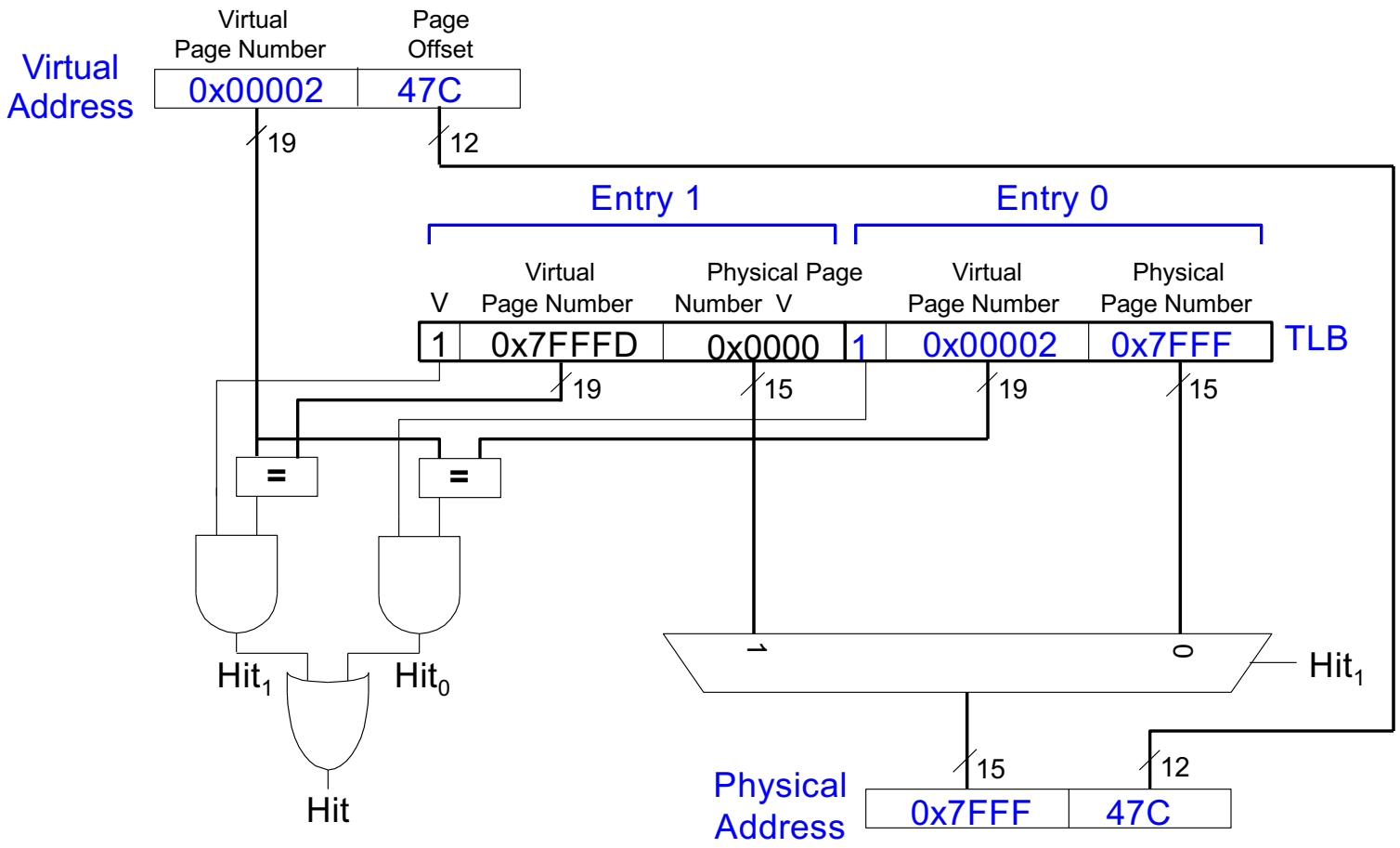
Toate instrucțiunile de pe această pagină au aceeași adresă virtuală de pagină – sunt supuse aceleiași translații

Translatare virtual-fizic a adreselor folosind un TLB și cum este folosită adresa fizică obținută pentru a accesa memoria cache

Tratarea unui TLB Miss

- Software (MIPS, Alpha)
  - Un TLB miss produce o excepție și sistemul de operare reîncarcă TLB. Trebuie să existe un mod de adresare privilegiat ”netranslatat” pentru refacerea TLB.
- Hardware (SPARC v8, x86, PowerPC, RISC-V)
  - Avem o unitate memory management unit (MMU) care reface TLB.
  - Dacă o pagină lipsă (date sau PT) este găsită în timpul reîncărcării TLB, MMU renunță și generează o excepție de Page Fault pentru instrucțiunea originală.

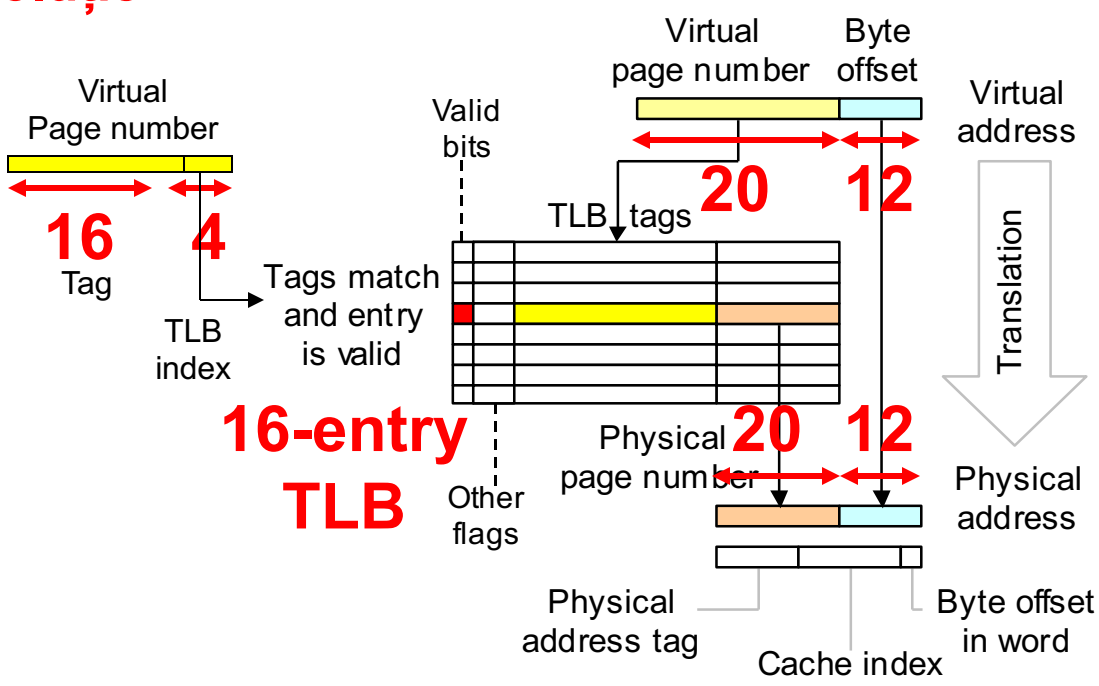
Exemplu TLB cu două intrări



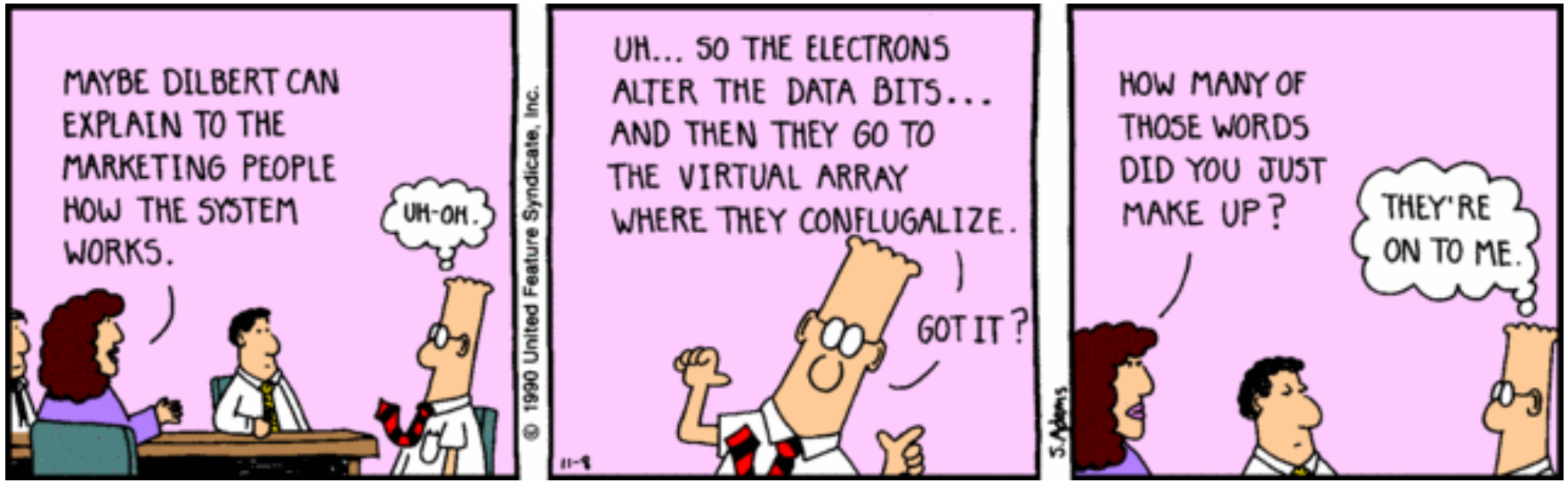
Translatarea adreselor via TLB

Un algoritm de translatare convertește o adresă virtuală de 32 de biți într-o adresă fizică de 32 de biți. Memoria este adresabilă la nivel de octet și are pagini de 4KB. Este folosit un TLB cu 16 intrări mapat direct. Care sunt componentele adreselor virtuale și fizice și lățimea diferitelor câmpuri ale TLB

Soluție



TLB word width =
16-bit tag +
20-bit phys page # +
1 valid bit +
Other flags
≥ 37 bits



http://dilbert.com/strips/comic/1990-11-08/

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252