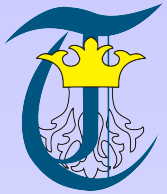


Universitatea *Transilvania* Brașov
Facultatea de Inginerie Electrică și Știința Calculatoarelor
Departamentul de Electronică și Calculatoare

Sisteme de operare

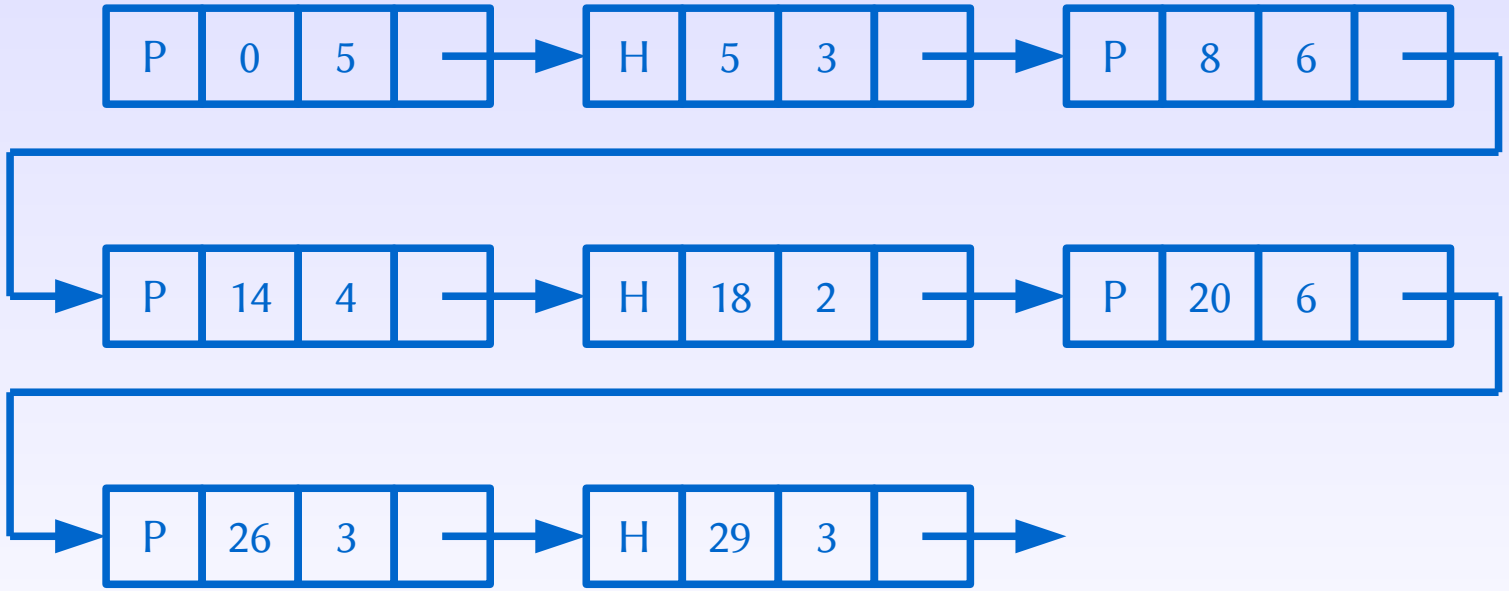
– curs 8 –

2010.04.17

ș.l. dr. ing. Kertész Csaba-Zoltán

3.2.3. Managementul memoriei utilizând liste înlănțuite

- memoria va fi reprezentată ca o listă de segmente H (hole) și P (process)
- de ex.:



Liste înlănțuite

- în exemplul prezentat sortarea listei s-a făcut după adrese
- sortarea după adrese aduce avantajul că actualizarea listei în cazul terminării procesului sau eliminării din memorie poate fi implementat foarte ușor:
  - dacă procesul are alte procese în vecinătate, eticheta P se va schimba în H
  - dacă procesul are o gaură vecină atunci se va fuziona cu gaura respectivă prin eliminarea elementului din listă și modificarea începutului și lungimii din elementul vecin

Algoritmi de alocare a memoriei

- folosit pentru un proces nou sau unul transferat de pe disc
- algoritmul **first fit**
  - managerul va căuta în lista de segmente până când găsește un segment H ≥ decât dimensiunea memoriei ce trebuie alocată
  - acest segment va fi împărțit în două: o bucată pentru proces iar celălalt memoria neutilizată rămasă
  - este un algoritm rapid

Algoritmi de alocare a memoriei

- algoritmul **next fit**
  - algoritm similar cu first fit, numai că va memora segmentul pe care a alocat și următoarea dată va continua de acolo
- algoritmul **best fit**
  - caută în toată lista și va alege cel mai mic segment în care încape procesul
  - este mai încet decât cele anterioare
  - în mod surprinzător conduce la utilizarea mai proastă a memoriei decât primele două (tinde să umple memoria cu segmente foarte mici)

Algoritmi de alocare a memoriei

- algoritmul **worst fit**
  - caută cel mai mare segment liber
  - procesul va utiliza o parte din acest segment, iar restul va fi suficient de mare pentru a putea fi folosit de un alt proces
  - nu este foarte eficient
- algoritmul **quick fit**
  - se menține liste separate pentru câteva din dimensiunile de segmente utilizate des
  - găsirea unui segment liber se face foarte rapid, dar concatenarea spațiilor eliberate este lentă

3.2.4. Managementul memoriei utilizând buddy system

- buddy system este un algoritm de management al memoriei care exploatează faptul că se utilizează numere reprezentate în cod binar pentru adrese
- astfel se va putea accelera concatenarea segmentelor neutilizate adiacente atunci când se termină un proces sau este transferat pe disc
- managerul de memorie menține o listă a blocurilor libere de mărime 1,2,4,8,16 până la dimensiunea maximă a memoriei

Buddy systems

- inițial memoria este liberă și lista blocurilor de 1MB conține un singur element, celelalte liste sunt goale
- când un proces trebuie alocat se va căuta în lista blocurilor (de putere a lui 2 imediat următoare a dimensiunii procesului) o zonă liberă
- dacă nu se găsește în lista respectivă, se va împarte un bloc de dimensiune dublă în două jumătăți adiacente (buddies)

Buddy systems

- când un bloc este eliberat, managerul de memorie caută în lista buddy-urilor blocuri libere pentru a fi concatenate
- algoritmul este destul de ineficient în utilizarea memoriei
  - un proces va ocupa o zonă de memorie rotunjită la cea mai apropiată putere a lui 2, astfel rămân zone goale: **fragmentare internă**

Buddy systems

Inițial	1024k					
Req 70k	A	128k		256k		512k
Req 35k	A	B	64k	256k		512k
Req 80k	A	B	64k	C	128k	512k
Return A	128k	B	64k	C	128k	512k
Req 60	128k	B	D	C	128k	512k
Return B	128k	64k	D	C	128k	512k
Return D	256k			C	128k	512k
Return C	1024k					

3.2.5. Alocarea spațiului pentru swap

- algoritmii prezentați până acum au fost folosiți pentru a stoca procesele în memoria principală
- atunci când un proces este transferat din memorie pe disc va trebui să-i alocăm spațiu pe disc
- pentru a realiza managementul spațiului de pe disc se vor utiliza aceiași algoritmi ca pentru memoria principală, cu restricția că spațiul alocat pe disc trebuie să fie multiplu de sectoare de pe disc

3.2.6. Analiza sistemelor cu swapping

- sistemele cu liste și bitmap conduc la o formă de fragmentare externă ușor de realizat
- printr-o umplere aleatoare a memoriei cu procese și zone libere se pot estima probabilitățile ca unele zone să rămână nealocate
- de obicei numărul de zone libere este jumătate din numărul de procese (zonele libere adiacente se concatenează)
- se calculează mărimea medie a zonelor libere (influențat de numărul și tipul proceselor)

3.3. Memoria virtuală

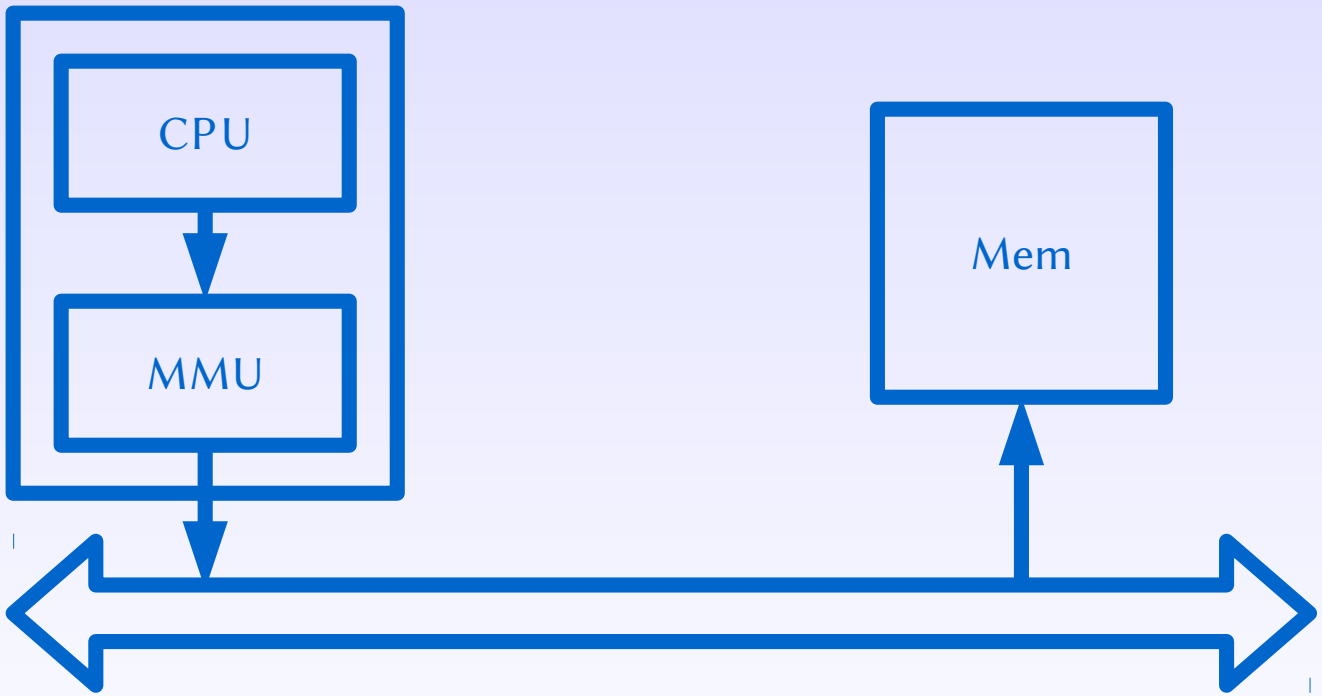
- metodă dezvoltată în 1961 de Fotheringham
- memoria combinată a programului, datelor și stivei poate depăși mărimea memoriei principale
- SO va trebui să mențină părțile care se utilizează la un moment dat în memorie, iar restul să le păstreze pe disc
- de ex. un proces de 1MB poate rula într-o memorie de 256kB, alegând care bucată se execută și transferând bucata respectivă în memorie

3.3.1. Paginarea

- adresele utilizate într-un program sunt numite **adrese virtuale** și vor forma spațiul adreselor virtuale
- în cazul calculatoarelor fără memorie virtuală, adresa virtuală este plasată direct pe magistrala de adrese: adresa virtuală = adresa fizică
- dacă se utilizează memoria virtuală, adresa virtuală nu va fi plasată pe magistrală ci va fi preluat de MMU (Memory Management Unit)

MMU

- MMU va translata adresa virtuală într-o adresă fizică, ca va fi apoi plasată pe magistrala de adrese



Paginarea

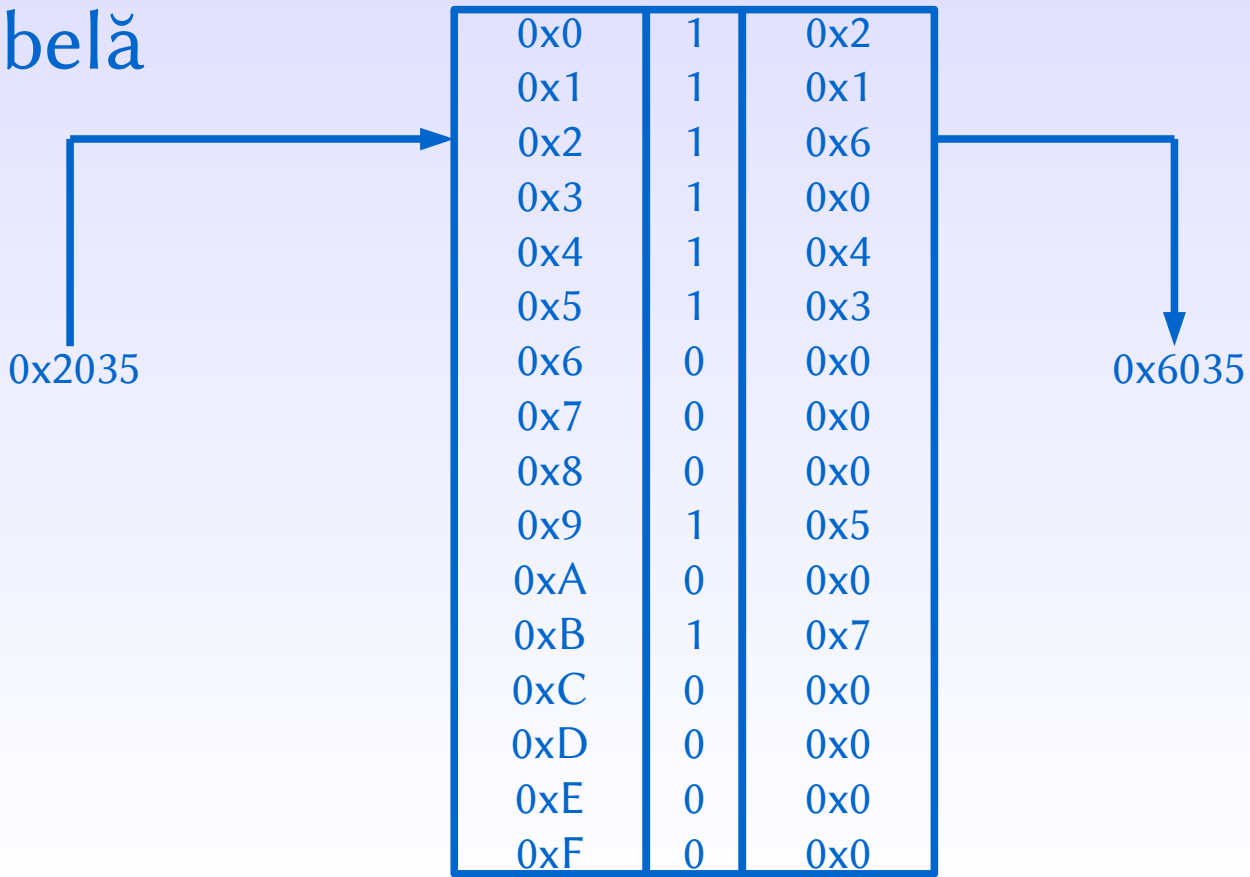
- spațiul virtual de adrese este împărțit în pagini
- paginile vor fi mapate în memorie pe anumite frame page-uri (de același dimensiune ca pagina)
- instrucțiunile de adresare vor folosi adrese virtuale din interiorul acestor pagini, iar MMU va calcula exact ce frame page îi corespunde adresei respective
- dat fiind faptul că spațiul de adrese virtuale este mai mare decât spațiul de adrese fizice, nu toate paginile pot fi mapate în memorie deodată

Page fault

- fiecare pagină precizează printr-un bit dacă este prezent în memorie sau nu
- dacă programul încearcă să utilizeze o pagină nemapată în memorie MMU va genera un trap (întrerupere software) către sistemul de operare numit page fault
- când prinde un page fault, SO va muta pagina cea mai puțin utilizată din memorie pe disc, și va aduce pagina referită de pe disc în memorie și va mapa corespunzător pagina respectivă

Implementare MMU

- MMU este implementat prin folosirea unei tabele de look-up
- numărul paginii va fi utilizat ca index în această tabelă



3.3.2. Tabele de pagini

- page table este utilizat pentru maparea paginilor virtuale în page frame-uri
- poate fi văzut ca o funcție având un număr de pagini virtuale ca argument iar rezultatul evaluării funcției va fi numărul de page frame
- probleme:
  - page table poate deveni destul de mare
    - de ex.: 32biţi, pagini 4k => 1M pagini (pt. fiecare proces)
  - maparea trebuie să fie rapidă
    - maparea este făcut la fiecare referire la memorie

Implementări de tabele de pagini

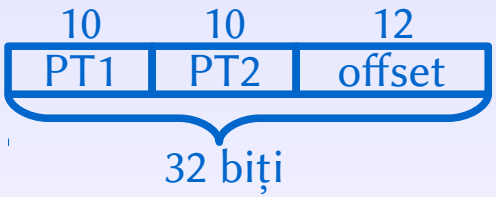
- cel mai simplu este folosirea unui singur page table alcătuit dintr-un masiv de registrii hardware foarte rapizi
  - la activarea procesului sistemul de operare încarcă registrele cu page table corespunzător din memorie
  - nu se face acces la memorie în plus pentru mapare
  - este foarte scumpă
  - crește timpul de comutare a proceselor

Implementări de tabele de pagini

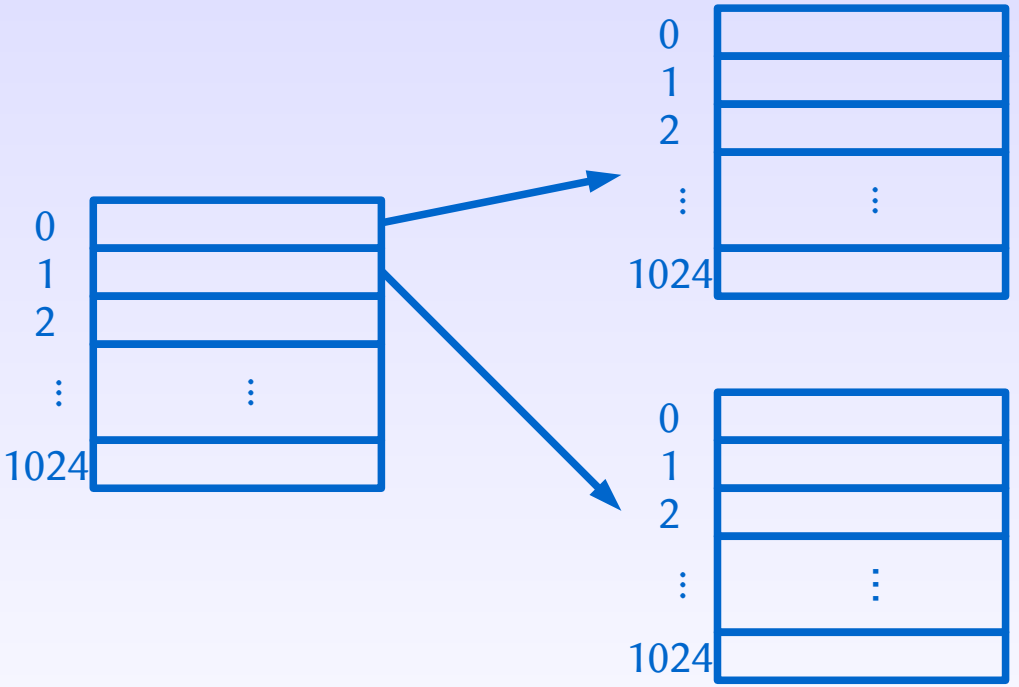
- o altă implementare menține page table în memorie și folosește un singur registru care conține un pointer la începutul tabelei
  - la comutarea proceselor va trebui încărcat doar un registru => comutare rapidă
  - translatarea se face utilizând referințe la memorie: fiecare acces la memorie va include încă un acces în plus => timpul de acces este mai mare

Tabele de pagini multi-nivel

- se folosesc pentru a rezolva problema construirii unor tabele de pagini mari în memorie



2^{20} pagini de 4k



Tabele de pagini multi-nivel

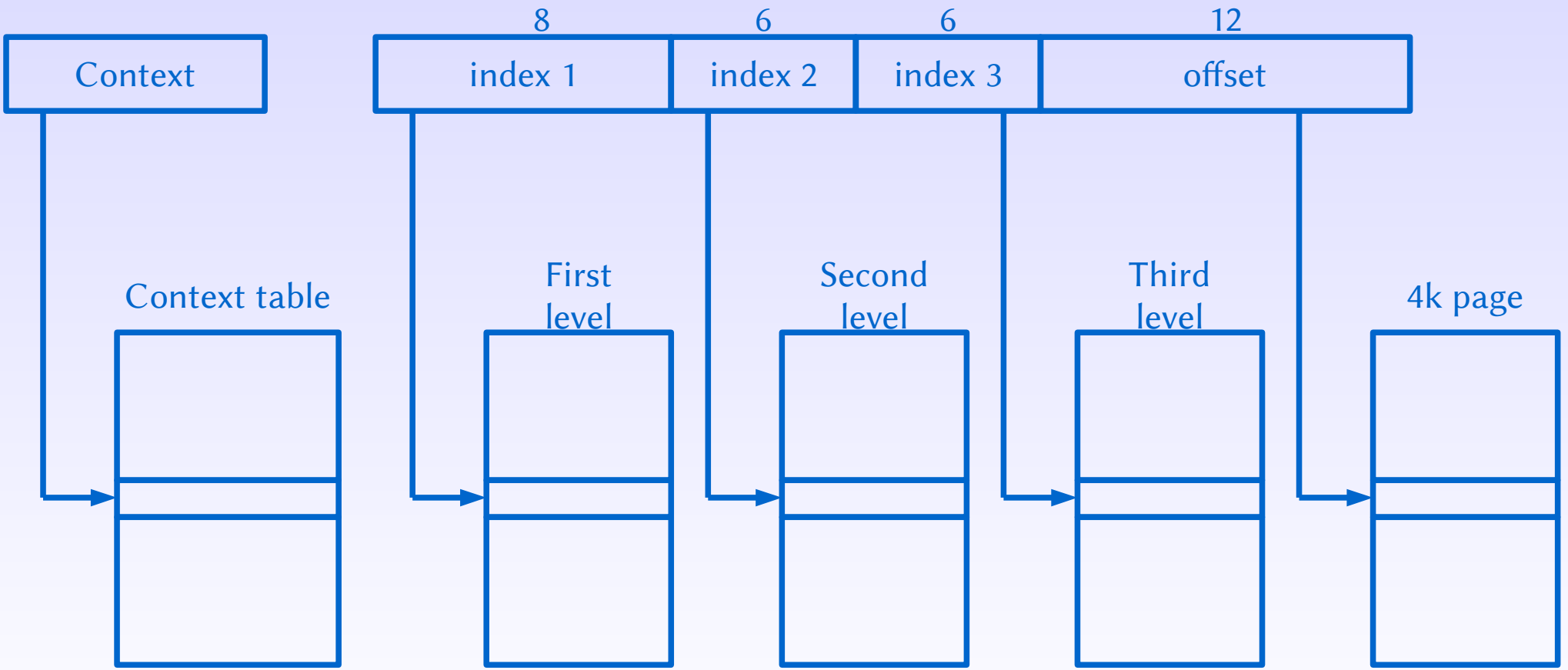
- nu va trebui memorat toate tabelele în memorie la un moment dat
  - de exemplu dacă un proces necesită 12MB (4MB text, 4MB date, 4MB stack), va rămâne foarte mult spațiu neutilizat între date și stack
- când o adresă virtuală este trimisă la MMU, aceasta va extrage PT1 și va utiliza ca index în page table de top
  - fiecare linie în tabela top va reprezenta 4MB
- după aceasta MMU va utiliza PT2 pentru index în tabela selectată de PT1

Tabele de pagini multi-nivel

- dacă pagina determinată nu este în memorie bitul P (present) este 0 cauzând page fault
- aceasta este adevăr și pentru tabela top-level
- adresele neutilizate nu vor fi încărcate niciodată în memorie
- de ex.: programul de 12MB va avea nevoie de 4 tabele de pagini (1 top, 1 text, 1 date și 1 stack) de câte 1024 linii în loc de 1 milion de linii (pentru spațiul de adrese pe 32 de biți)
- extinderea tabelelor pe mai mult de 3 nivele devine costisitoare

3.3.3. Exemple de hardware pentru paginare

- paginare cu 3 nivele: SPARC



Paginarea în SPARC

- când se încarcă un proces în memorie, SO îi atribuie un număr de context pe care îl va păstra până la terminare
- MMU poate memora 4096 contexte
- când se dorește accesarea unui cuvânt din memorie se trimite la MMU contextul și adresa virtuală
- pentru a accelera procesul de translatare se folosește o memorie asociativă

3.3.4. Memoria asociativă

- majoritatea implementărilor memorează page table în memorie, ceea ce afectează performanțele sistemului
  - pentru accesul în memorie e nevoie de cel puțin încă un acces în plus pentru a accesa tabela de pagini
- soluția pornește de la observația: un număr mare de accese se referă la un număr mic de adrese
  - numai o mică parte din liniile tabelului de pagini vor fi utilizate

Memoria asociativă

- soluția este de a utiliza un dispozitiv hardware minimal pentru a mapa adresele virtuale în adrese fizice fără a folosi page table din memorie
  - se va folosi o memorie asociativă (numit și Translation Lookaside Buffer)

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	rw	31
1	20	0	r x	38
1	130	1	rw	29
1	129	1	rw	62
1	19	0	r x	50
1	21	0	r x	45
1	860	1	rw	14
1	861	1	rw	75

Memoria asociativă

- când se furnizează o adresă virtuală către MMU, se va verifica dacă numărul paginii virtuale se află în memoria asociativă, comparând în paralel toate intrările
- dacă se găsește o potrivire și tipul accesului este valid, page frame-ul va fi luat din memoria asociativă fără a mai utiliza page table din memorie
- dacă nu corespunde accesul se va genera **protection fault**

Memoria asociativă

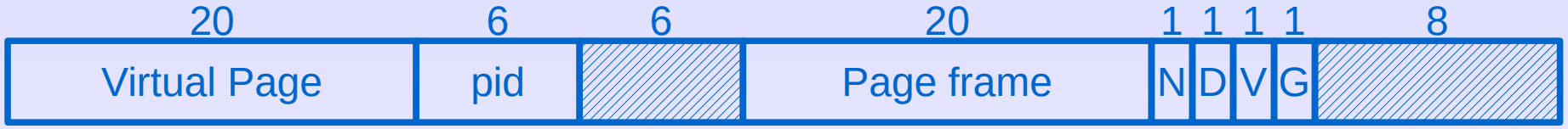
- dacă numărul de pagini nu se află în memoria asociativă, MMU va prelua linia corespunzătoare din page table și va plasa în memoria asociativă în locul alteia
- fracțiunea din referințele la memorie care sunt satisfăcute utilizând memoria asociativă se numește **hit ratio**
  - cu cât mai mare hit ratio cu atât mai performant sistemul

Memoria asociativă în sisteme multitasking

- fiecare proces are propria tabelă de pagini și propria mapare de adrese virtuale
- când se rulează un proces nou, aceasta nu poate utiliza conținutul memoriei asociative
- soluții:
  - se asigură o instrucțiune pentru a invalida memoria asociativă (se resetează biții valid)
  - extindem memoria asociativă cu un câmp PID și adăugăm un registru care conține identificatorul procesului curent

Exemplu hardware: MIPS R2000

- au eliminat tablele de pagini
- CPU conține o memorie asociativă cu 64 linii



N: 0 use cache, 1 no cache
D: 0 entry is clean, 1 dirty
V: 0 invalid, 1 valid
G: 0 check pid, 1 do not check

- CPU generează adresa virtuală, hard-ul compară numărul paginii virtuale cu câmpurile corespunzătoare din memoria asociativă
- dacă se găsește, are loc translatarea hardware

MIPS R2000

- dacă adresa virtuală nu se găsește în memoria asociativă, nu se va mai căuta în tabele de pagini, ci se va cauza un **trap**
- SO determină care pagină virtuală este necesară și va înlocui o linie din memoria asociativă cu informațiile determinate
- dacă pagina nu se află în memorie se va executa page fault normal
- se generează page fault și pentru nepotrivirile din memoria asociativă