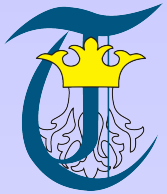


Universitatea *Transilvania* Brașov
Facultatea de Inginerie Electrică și Știința Calculatoarelor
Departamentul de Electronică și Calculatoare

Sisteme de operare

– curs 11 –

2012.05.22

ș.l. dr. ing. Kertész Csaba-Zoltán

3.6.4. Considerații privind implementarea sistemelor de paginare a memoriei

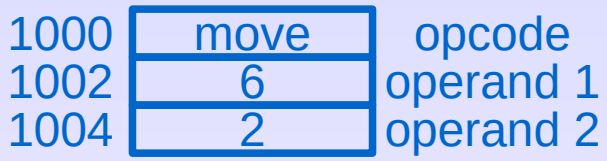
- proiectantul unui sistem de paginare va trebui să aleagă:
  - algoritmul de înlocuire a paginilor
  - strategia de alocare a paginilor (locală sau globală)
  - utilizarea soluției de tip demand paging sau prepaginare
- mai există câteva probleme practice legate de implementare

Instruction Backup

- când un program accesează o pagină care nu este în memorie, instrucțiunea care a cauzat page fault este oprită din execuție și se va executa un trap
- după ce SO a încărcat pagina necesară va trebui să reexecute instrucțiunea care a cauzat page fault
- acest lucru este foarte greu de implementat

Instruction Backup

- de exemplu o instrucțiune de transfer din memorie din M68 000
  - move.l #6(A1), 2(A0)
  - instructiunea are 6 byte



- depinzând de care din referințele la memorie (cea pentru opcode, primul operand sau al doilea) a cauzat page fault ultima valoare pentru PC poate fi 1000, 1002, 1004
- SO nu poate determina unde începe instrucțiunea

Blocarea paginilor în memorie

- vom lua în considerare interacțiunea între operațiile I/O și sistemul de memorie virtuală
- de ex.:
  - un proces care execută un apel sistem (`read`) pentru a citi dintr-un fișier sau device într-un buffer din spațiul său de adrese
  - în timp ce procesul așteaptă terminarea operației I/O, procesul este suspendat, și controlul este dat altui proces
  - noul proces va cauza un page fault
  - algoritmul de paginare este global

Blocarea paginilor în memorie

- există șansa că pagina conținând bufferul I/O să fie eliminată din memorie
- în același timp se execută un transfer DMA către acest buffer, dar pagina fiind înlocuită, transferul nu va avea loc către buffer
- soluții:
  - blocarea paginilor angajate în operații I/O în memorie
  - realizarea tuturor operațiilor I/O în bufferele din kernel, apoi transferarea datelor în paginile proceselor

Partajarea paginilor

- în sistemele timesharing este avantajoasă utilizarea unor pagini partajate
- de exemplu dacă mai mulți utilizatori execută un program, e avantajos să avem o singură copie a codului executat
- paginile de date nu pot fi partajate!
- chiar și în cazul paginilor care conțin cod apare o problemă:
  - la schimbarea proceselor toate paginile vor fi eliminate, și celălalt proces va genera page faulturi
  - paginile partajate nu trebuie eliminate

Paging Daemons

- majoritatea implementărilor sistemelor de paginare utilizează un daemon numit paging daemon
- aceasta doarme în majoritatea timpului, dar devine activ periodic pentru a inspecta starea memoriei
- dacă există puține page frame-uri libere, daemonul va selecta câteva pentru a le elimina din memorie (pe baza unui algoritm de înlocuire a paginilor)
- menținând câteva page frame libere => performanțe mai bune

Secvența de evenimente la apariția unui page fault

1. se execută un trap, se salvează PC în stivă, la unele CPU se salvează informațiile despre starea instrucțiunii curente în registre speciale
2. se execută o rutină în asamblare pentru a salva regiștrii de uz general și alte informații, rutina execută un apel către kernel
3. SO determină faptul că a apărut un page fault și încearcă să descopere ce pagină virtuală este necesară (din registre speciale sau prin încărcarea instrucțiunii și analizarea operanzilor)

Secvența de evenimente la apariția unui page fault

4. SO cunoaște adresa virtuală necesară și o verifică dacă e validă
   - se verifică dacă protecția zonei de memorie respectivă este consistentă cu accesul
     - dacă nu: se trimite procesului un semnal sau procesul este omorât
     - dacă da: SO încearcă obținerea unui page frame liber
   - dacă nu există nici un page frame liber, se execută o rutină ce implementează un algoritm de înlocuire a paginilor pentru a selecta pagina care va fi înlocuit

Secvența de evenimente la apariția unui page fault

5. dacă pagina aleasă a fost modificată, ea va fi planificat pentru a fi transferată pe disc și se execută comutarea proceselor
   - procesul care a cauzat page fault e suspendat până când pagina e transferată în memorie
   - cât timp are loc transferul, frame-ul ales este marcat ca fiind ocupat, pentru a nu fi utilizat de alt proces
6. după eliberarea page-frame-ului, SO determină adresa de pe disc unde se află pagina cerută și planifică o operație cu discul pentru a aduce în memorie

Secvența de evenimente la apariția unui page fault

7. întreruperea de disc va semnala că pagina a fost transferată
   - tabela de pagini este actualizată iar page-frame-ul în care a fost plasat e marcat ca fiind în starea normală
8. PC este setat pentru a arăta pe instrucțiunea care a cauzat page fault-ul
9. procesul care a cauzat page fault-ul este planificat și se revine în rutina care a apelat kernelul
10. rutina reface regiștrii și celelalte informații pe care le-a salvat anterior și continuă execuția

3.7. Segmentarea

- memoria virtuală realizată prin paginare este unidimensională, deoarece adresele cresc de la 0 până la o adresă maximă
  - există un singur spațiu de adrese
- pentru a evita unele probleme este bine să avem 2 sau mai multe spatii de adrese
  - de ex. dacă un program lucrează cu multe tabele dinamice, una din tabele poate să crească peste spațiul alocat
- o soluție generală, elegantă este să creăm spatii de adrese independente => **segmente**

Segmente

- fiecare segment constă dintr-o secvență liniară de adrese, de la 0 până la dimensiunea maximă a segmentului
- lungimea segmentului poate fi între 0 și dimensiunea maximă permisă
- segmentele vor avea dimensiuni diferite
- dimensiunea segmentelor se poate modifica în timpul execuției
  - de ex.: stiva: segmentul crește când se adaugă în stivă, și scade când se extrage din stivă

Segmente

- deoarece fiecare segment are un spațiu de adrese independente, segmentele pot să crească independent fără să se afecteze între ele
- pentru a specifica o adresă pentru o memorie virtuală utilizând segmentarea vom utiliza 2 componente
  - un număr de segment
  - o adresă în cadrul segmentului
- de obicei un segment are un singur scop: segment de date, segment de cod, segment de stivă

Avantajele segmentării

- segmentarea facilitează partajarea procedurilor sau datelor între mai multe procese
- ex.: shared object (.so)
  - bibliotecile comune pot fi plasate într-un segment care va fi partajat de mai multe procese
- segmentele pot avea diferite tipuri de protecții:
  - segmentul de cod poate fi numai executat
  - segmentul de date poate fi citit/scris, nu și executat

Paginare vs. Segmentare

	paginare	segm.
utilizarea metodei este transparentă utilizatorului?	nu	da
câte spații liniare de adrese există	1	multe
spațiul de adrese poate depăși dimensiunea memoriei fizice?	da	da
este adaptabil la modificarea dimensiunii datelor conținute?	nu	da
se poate realiza distincția și protecția între cod și date?	nu	da
se poate partaja cod?	nu	da

Implementarea segmentării

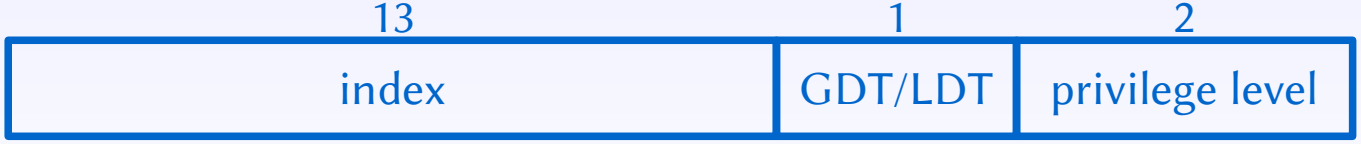
- implementarea segmentării diferă de implementarea paginării prin faptul că paginile au dimensiuni fixe, iar segmentele nu
- datorită dimensiunilor diferite pot apare fragmentarea
  - necesită compactarea memoriei
- există implementări care utilizează și segmentare și paginare: se combină avantajele celor două
  - mărimea uniformă a paginilor, utilizare mai eficientă a memoriei, ușurință în programare, protecție

Segmentare la i386

- 16k segmente de 4GB fiecare => 64TB dimensiunea maximă a memoriei virtuale
- nu este important numărul de segmente ci dimensiunea lor: puține programe folosesc un număr mare de segmente, dar multe programe folosesc segmente de ordinul MB
- pentru a implementa memoria virtuală se utilizează 2 tabele
  - LDT (Local Descriptor Table)
  - GDT (Global Descriptor Table)

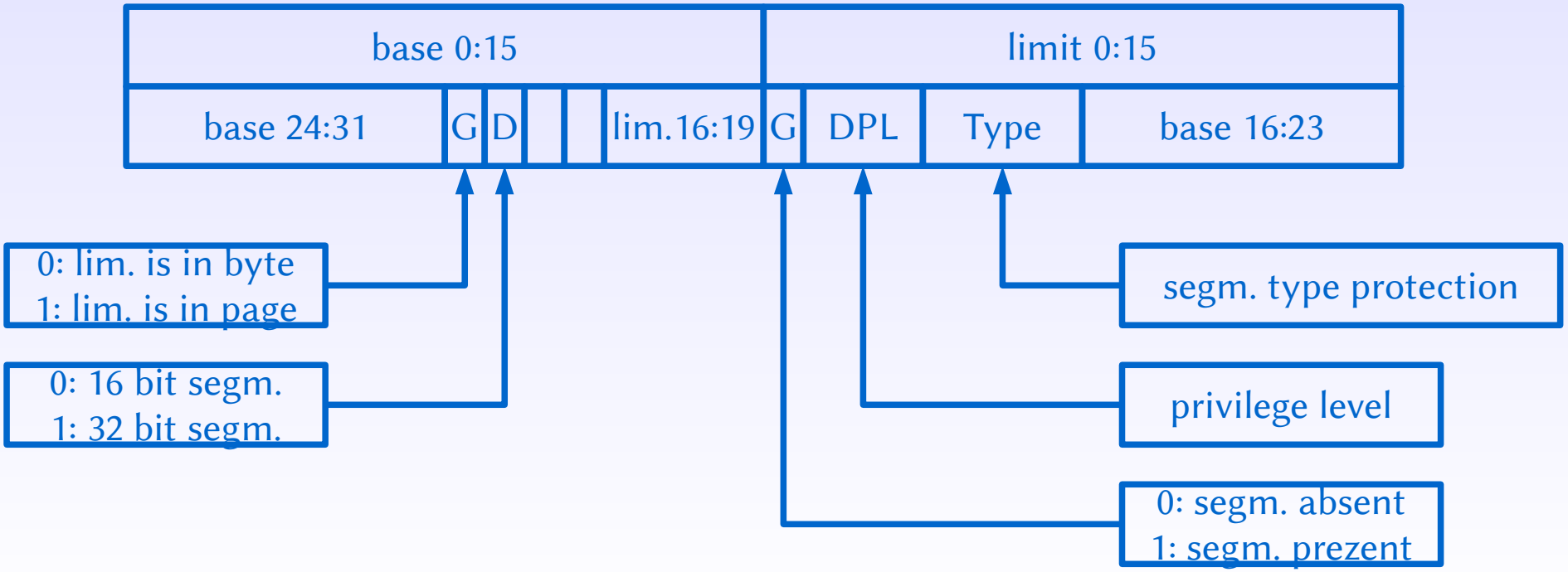
Segmentarea la i386

- fiecare program are propriul LDT, dar există un singur GDT partajat de toate programele
- LDT descrie segmentele locale fiecărui program (codul, date și stiva), iar GDT descrie segmentele sistem (codul SO)
- pentru a accesa un segment procesorul încarcă un selector în unul din cei 6 regiștri de segment (CS, DS, SS, ES, FS, GS)
- selectorul:



Segmentarea la i386

- indexul reprezintă un deplasament în GDT sau LDT (fiecare tabelă poate descrie 8k segmente)
- descriptorul 0 nu va fi utilizat
- descriptorul de segment:



Segmentarea la i386

- se folosește indexul pentru a găsi descriptorul de segment
- se verifică dacă offsetul nu depășește limita segmentului
- se adună la bază offsetul pentru a forma o adresă liniară
- adresa liniară este tratată ca o adresă virtuală și transmisă la MMU
- i386 permite 4 nivele de protecție, 0 cel mai privilegiat, accesul la date de pe un nivel mai prioritar nu este permis