

Jumătatea a 2-a a semestrului

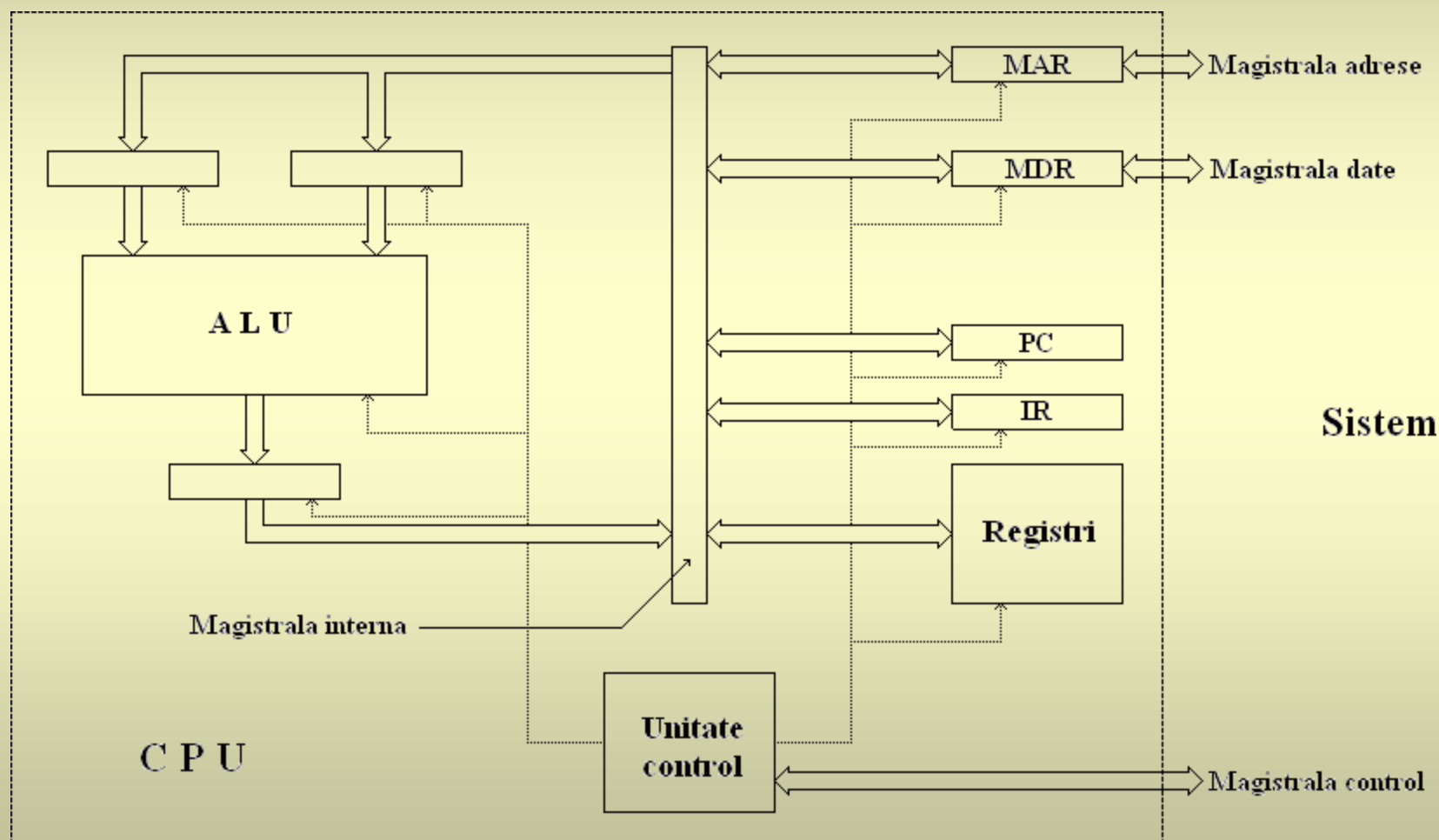
- curs
 - structura hardware a unui sistem de calcul
 - finalizare - test scris
 - condiție susținere test - cel mult 2 absențe la laborator
- laborator
 - limbaj de asamblare
 - finalizare - test practic

Cuprins

- I. Unitatea centrală de procesare (CPU)
- II. Îmbunătățirea performanței CPU
- III. Dispozitivele periferice
- IV. Sistemul de întreruperi
- V. Sistemul de operare

I. Unitatea centrală de procesare (CPU)

Structura CPU (1)



Structura CPU (2)

- unitatea aritmetică și logică (ALU)
 - efectuează calculele propriu-zise
- regiștrii de uz general
- unitatea de control
 - comandă celelalte componente
 - stabilește ordonarea temporală a operațiilor
- magistrala internă

Structura CPU (3)

- contorul program (PC)
 - reține adresa următoarei instrucțiuni de executat
 - actualizat de procesor
 - uzual nu este accesibil prin program
- registrul de instrucțiuni (IR)
 - reține codul ultimei instrucțiuni aduse din memorie

Structura CPU (4)

- regiștrii de interfață
 - asigură comunicarea cu magistralele sistemului
 - de adrese: MAR (*Memory Address Register*)
 - de date: MDR (*Memory Data Register*)
- regiștrii temporari
 - intermediari între diverse componente
 - exemple: regiștrii operanzi ALU, registrul rezultat ALU

II. Îmbunătățirea performanței CPU

Cum putem crește performanța?

- eliminarea factorilor care frânează CPU
 - exemplu - folosirea memoriei cache
- structuri cât mai simple
 - nu mai este posibil la procesoarele actuale
- creșterea frecvenței ceasului
 - limitată de tehnologie
- execuția instrucțiunilor în paralel

Creșterea performanței - tehnici

- Structura de tip *pipeline*
- Multiplicarea unităților de execuție
- Predicția salturilor
- Execuția speculativă
- Predicația
- Execuția *out-of-order*
- Redenumirea regiștrilor
- *Hyperthreading*
- Arhitectura RISC

II.1. Pipeline

Ideea de pornire

- execuția unei instrucțiuni - număr mare de pași
- în pași diferiți se folosesc resurse diferite ale CPU
- execuția unei instrucțiuni poate începe înainte de terminarea celei anterioare
- instrucțiunile se execută (parțial) în paralel

O primă implementare

Procesorul Intel 8086

- format din două unități
 - unitatea de interfață cu magistrala (BIU)
 - comunicarea cu exteriorul
 - unitatea de execuție (EU)
 - execuția propriu-zisă a operațiilor
- BIU și EU pot lucra în paralel

Principiul benzii de asamblare

- execuția unei instrucțiuni - n pași
- la un moment dat - n instrucțiuni în execuție
- fiecare instrucțiune - în alt pas

	pas 1	pas 2	...	pas $n-1$	pas n
instrucțiune 1			...		
instrucțiune 2			...		
⋮	⋮				
instrucțiune $n-1$...		
instrucțiune n			...		

Pipeline

- secvența pașilor (*stagii*) prin care trece execuția unei instrucțiuni
- trecerea între două stagii - la fiecare ciclu de ceas
- cât durează până la terminarea unei instrucțiuni?
 - prima instrucțiune - n cicluri de ceas
 - următoarele instrucțiuni - câte 1 ciclu de ceas !

Performanța unui pipeline

- rezultatul obținut la fiecare stadiu trebuie reținut
- regiștri de separație - plasați între stagii
- frecvența ceasului - dată de stagiul cel mai lung
- pași mai simpli
 - număr de stagii mai mare
 - frecvență mai mare a ceasului

Execuția unei instrucțiuni

1. depunerea valorii PC (adresa instrucțiunii)
în MAR
2. citirea din memorie
3. preluarea codului instrucțiunii în MDR
4. depunerea codului instrucțiunii în IR
5. actualizarea valorii PC

Execuția unei instrucțiuni

6. decodificarea instrucțiunii de către unitatea de control
7. citire operand din memorie
 - depunere adresă operand în MAR
 - comandă citire
 - preluare operand în MDR
- 7'. selecție registru care conține operandul

Execuția unei instrucțiuni

8. depunere operand în registru operand ALU
9. repetare pași 7-8 pentru al doilea operand
10. transmiterea către ALU a codului operației dorite
11. preluare rezultat în registrul rezultat ALU
12. testare condiție salt

Execuția unei instrucțiuni

13. salt (dacă este cazul)

14. scriere rezultat în memorie

- depunere rezultat în MDR
- depunere adresă în MAR
- comandă scriere

14'. scriere rezultat în registrul destinație

Evoluție

- Intel Pentium III - 10 stagii
- Intel Pentium IV (Willamette, Northwood) - 20 stagii
- Intel Pentium IV (Prescott) - 32 stagii
- AMD Athlon - 17 stagii

Probleme

- nu toate instrucțiunile se pot executa în paralel
- dependență - o instrucțiune trebuie să aștepte terminarea alteia
- conflict în accesul la aceeași resursă

Parametri de performanță

- latența (*latency*) - numărul de cicluri de ceas necesar pentru execuția unei instrucțiuni
 - dat de numărul de stagii
- rata de execuție (*throughput*) - numărul de instrucțiuni terminate pe ciclu de ceas
 - teoretic - egală cu 1
 - practic - mai mică (din cauza dependențelor)

Tipuri de dependențe

- structurale
- de date
- de control

Dependențe structurale

- instrucțiuni aflate în stagii diferite au nevoie de aceeași componentă
- o singură instrucțiune poate folosi componenta la un moment dat
- celelalte instrucțiuni care au nevoie de ea sunt blocate

Dependențe structurale - exemple

- ALU
 - instrucțiuni aritmetice
 - calculul adreselor operanzilor
 - actualizarea valorii PC
- accesele la memorie
 - citire cod instrucțiune
 - citire operand
 - scriere rezultat

Dependențe de date

- o instrucțiune calculează un rezultat, alta îl folosește
- a doua instrucțiune are nevoie de rezultat înainte ca prima să-l obțină
- a doua instrucțiune este blocată

Dependențe de date - exemplu

```
mov  eax, 7
```

```
sub  eax, 3
```

- prima instrucțiune: scrierea în `eax` - în ultimul stadiu
- a doua instrucțiune: utilizarea `eax` - în primele stagii (decodificare)
 - așteaptă până când prima instrucțiune depune rezultatul în `eax`

Dependențe de control (1)

Actualizarea valorii PC (uzual)

- adunarea la vechea valoare a dimensiunii codului instrucțiunii anterioare
- încărcarea unei valori noi - instrucțiuni de salt

Dependențe de control (2)

Tipuri de instrucțiuni de salt

- necondiționat
 - se face saltul întotdeauna
- condiționat
 - se face saltul numai dacă este îndeplinită o anumită condiție
 - altfel se continuă cu instrucțiunea următoare

Dependențe de control (3)

Adresa de salt - moduri de exprimare

- valoare constantă
 - absolută
 - deplasament față de adresa instrucțiunii curente
- valoarea dintr-un registru
- valoarea dintr-o locație de memorie

Dependențe de control (4)

Adresa de salt - exemple:

`jmp 1594`

`jmp short -23`

`jmp eax`

`jmp dword ptr [esi]`

Dependențe de control (5)

Probleme

- calculul adresei de salt - în ultimele stagii de execuție
- instrucțiunile următoare (multe!) au început deja execuția
- dacă se face salt - efectele lor trebuie anulate

Dependențe de control (6)

Probleme

- "golirea" pipeline-ului → pierdere de performanță
 - operații complicate
 - durează mult până la terminarea primei instrucțiuni → scade rata de execuție
- o instrucțiune din 7 (în medie) este de salt !

Tratarea dependențelor

Soluții

- staționarea (*stall*)
- avansarea (*forwarding*)

Staționarea (1)

- atunci când o instrucțiune folosește un rezultat care încă nu a fost calculat
- instrucțiunea "stă" (nu trece la etapa următoare)
- echivalent cu inserarea unei instrucțiuni care nu face nimic (*nop*)
- spunem că în pipeline a fost inserată o bulă (*bubble*)

Staționarea (2)

- instrucțiunea trece mai departe când devine disponibil rezultatul de care are nevoie
- sunt necesare circuite de detecție
- nu e o soluție propriu-zisă
 - nu elimină efectiv dependența
 - asigură doar execuția corectă a instrucțiunilor
 - dacă o instrucțiune staționează, vor staționa și cele de după ea

Avansarea (1)

```
add dword ptr [eax], 5  
sub ecx, [eax]
```

- rezultatul adunării - calculat de ALU
- durează până când este scris la destinație
- instrucțiunea de scădere poate prelua rezultatul adunării direct de la ALU

Avansarea (2)

Avantaj

- reduce timpii de așteptare

Dezavantaje

- necesită circuite suplimentare complexe
- trebuie considerate relațiile între toate instrucțiunile aflate în execuție (în pipeline)

II.2. Multiplicarea unităților de execuție

Unități superscalare

- ideea de bază - mai multe ALU
- se pot efectua mai multe calcule în paralel
- folosită împreună cu tehnica pipeline
- MAR și MDR nu pot fi multiplicare
- cât de mult se pot multiplica ALU?
 - depinde de structura și eficiența pipeline

Unități superpipeline

- mai multe pipeline în același procesor
 - de obicei 2
- 2 (sau mai multe) instrucțiuni executate complet în paralel
- restricții
 - accesele la memorie și periferice - secvențial
 - unele instrucțiuni pot fi executate de un singur pipeline

II.3. Predicția salturilor

Predicție (1)

- rezolvarea dependențelor de control
- ideea de bază - a "prezice" dacă un salt se execută sau nu
 - nu se așteaptă terminarea instrucțiunii de salt
- predicție corectă - fără blocaje în pipeline
- predicție eronată - se execută instrucțiuni care nu trebuiau executate
 - efectul acestora trebuie anulat

Predicție (2)

- spor de performanță - cât mai multe predicții corecte (nu neapărat 100%)
- o instrucțiune executată eronat produce efecte doar când rezultatul este scris la destinație
- rezultatele instrucțiunilor - memorate intern de procesor până când se verifică dacă predicția a fost corectă

Scheme de predicție

Tipuri de scheme

- statice
 - întotdeauna aceeași decizie
- dinamice
 - se adaptează în funcție de comportarea programului

Scheme statice de predicție (1)

1. Saltul nu se execută niciodată

- rata predicțiilor corecte $\approx 40\%$
- ciclurile de instrucțiuni
 - apar des în programe
 - salturi frecvente

Scheme statice de predicție (2)

2. Saltul se execută întotdeauna

- rata predicțiilor corecte $\approx 60\%$
- ratări dese - structuri de tip *if*

Scheme statice de predicție (3)

3. Salturile înapoi se execută întotdeauna, cele înainte niciodată
- combină variantele anterioare
 - rată superioară a predicțiilor

Scheme dinamice de predicție (1)

- procesorul reține într-un tabel comportarea la salturile anterioare
 - salt executat/neexecutat
- un singur element pentru mai multe instrucțiuni de salt
 - tabel mai mic → economie de spațiu

Scheme dinamice de predicție (2)

Tipuri de predictor

- locali
 - rețin informații despre salturile individuale
- globali
 - iau în considerare corelațiile dintre instrucțiunile de salt din același program
- micști

Intel Pentium

- *Branch Target Buffer* (BTB)
 - cache asociativ pe 4 căi
 - 256 intrări
- Stările unei intrări
 - puternic lovit - se face salt
 - slab lovit - se face salt
 - slab nelovit - nu se face salt
 - puternic nelovit - nu se face salt

Implementarea BTB (1)

Memorarea și evoluția unei stări

- contor cu saturație pe 2 biți
 - poate număra crescător și descrescător
 - gama de valori - între 0 (00) și 3 (11)
 - din stările extreme nu se poate trece mai departe (doar înapoi)
- la fiecare acces, starea se poate schimba
 - condiția de salt este adevărată - incrementare
 - condiția de salt este falsă - decrementare

Implementarea BTB (2)

- codificarea stărilor
 - puternic lovit - 11 (se face salt)
 - slab lovit - 10 (se face salt)
 - slab nelovit - 01 (nu se face salt)
 - puternic nelovit - 00 (nu se face salt)
- de ce 4 stări?
 - a doua șansă - comportament pe termen lung

Implementarea BTB (3)

stare curentă	stare următoare	
	condiție salt adevărată	condiție salt falsă
00	01	00
01	10	00
10	11	01
11	11	10

Utilizarea cache-ului în predicție

Cache-ul de instrucțiuni

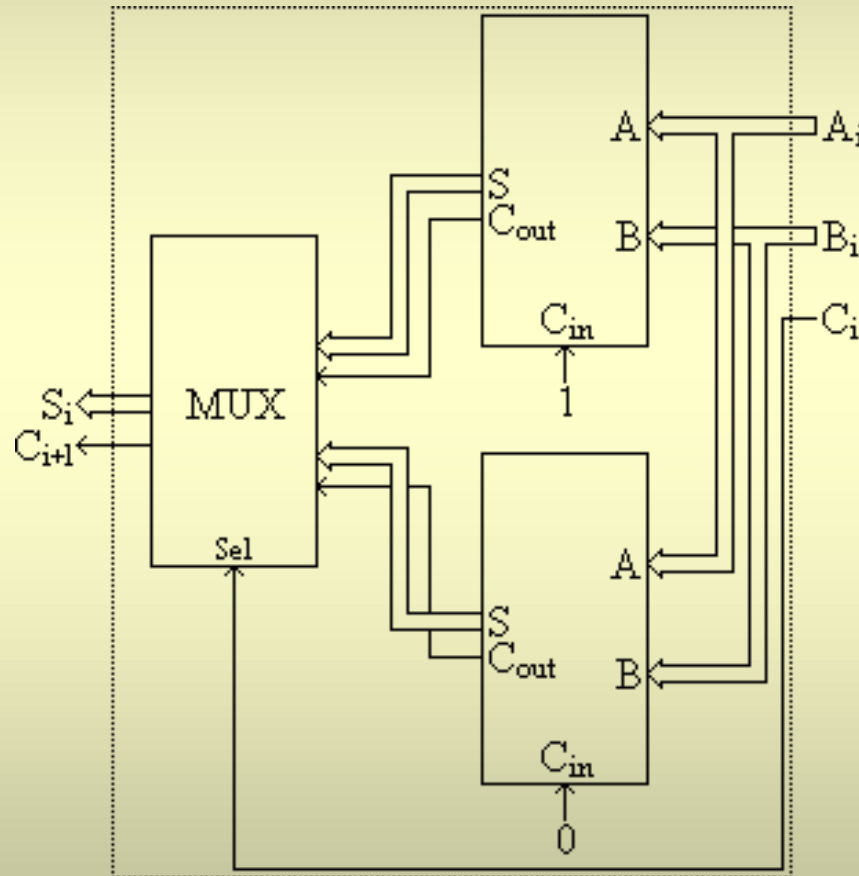
- reține vechea comportare a unui salt
 - condiție
 - adresă destinație
- *trace cache*
 - memorează instrucțiunile în ordinea în care sunt executate
 - nu în ordinea fizică

II.4. Execuția speculativă

Execuție speculativă

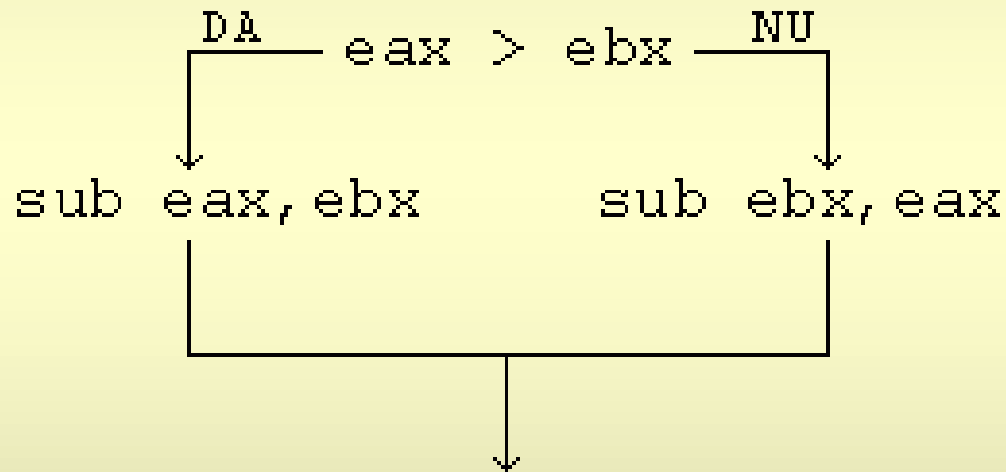
- înrudită cu predicția salturilor
- se execută toate variantele posibile
 - înainte de a ști care este cea corectă
- când se cunoaște varianta corectă, rezultatele sale sunt validate
- se poate utiliza și în circuite simple

Exemplu - sumatorul cu selecție



Cum funcționează? (1)

- instrucțiuni de salt condiționat



- ambele variante se execută în paralel
- cum se modifică regiștrii `eax` și `ebx`?

Cum funcționează? (2)

- nici una din variante nu îi modifică
- rezultatele scăderilor sunt depuse în regiștri temporari
- când se cunoaște relația între `eax` și `ebx`
 - se determină varianta corectă de execuție
 - se actualizează valorile `eax` și `ebx` conform rezultatelor obținute în varianta corectă

Execuție speculativă vs. predicție

- nu apar predicții eronate
 - rata de succes - 100%
- necesită mulți regiștri pentru rezultatele temporare
- gestiunea acestora - complicată
- fiecare variantă de execuție poate conține alte salturi etc.
 - variantele se multiplică exponențial

II.5. Predicația

Predicație (1)

- folosită în arhitectura Intel IA-64
 - și în alte unități de procesare
- asemănătoare cu execuția speculativă
- procesorul conține regiștri de predicate
 - predicat - condiție booleană (bit)
- fiecare instrucțiune obișnuită are asociat un asemenea predicat

Predicație (2)

- o instrucțiune produce efecte dacă și numai dacă predicatul asociat este *true*
 - altfel rezultatul său nu este scris la destinație
- instrucțiunile de test pot modifica valorile predicatelor
- se pot implementa astfel ramificații în program

Exemplu

- pseudocod

```
if (R1==0) {
```

```
    R2=5;
```

```
    R3=8;
```

```
}
```

```
else
```

```
    R2=21;
```

Exemplu (continuare)

- limbaj de asamblare "clasic"

```
cmp R1, 0
```

```
jne E1
```

```
mov R2, 5
```

```
mov R3, 8
```

```
jmp E2
```

```
E1: mov R2, 21
```

```
E2:
```

Exemplu (continuare)

- limbaj de asamblare cu predicate

cmp R1, 0, P1

<P1>mov R2, 5

<P1>mov R3, 8

<P2>mov R2, 21

- predicatele P1 și P2 lucrează în pereche
 - P2 este întotdeauna inversul lui P1
 - prima instrucțiune îi modifică pe amândoi

II.6. Execuția out-of-order

Execuție out-of-order

- instrucțiunile nu se mai termină obligatoriu în ordinea în care și-au început execuția
- scop - eliminarea unor blocaje în pipeline
- posibilă atunci când între instrucțiuni nu există dependențe

Exemplu

```
in al, 278
```

```
add bl, al
```

```
mov edx, [ebp+8]
```

- prima instrucțiune - foarte lentă
- a doua instrucțiune trebuie să aștepte terminarea primeia
- a treia instrucțiune nu depinde de cele dinaintea sa - se poate termina înaintea lor

II.7. Redenumirea regiștrilor

Dependențe de date

- apar când două instrucțiuni folosesc aceeași resursă (variabilă/registru)
- numai când cel puțin una din instrucțiuni modifică resursa respectivă
- dacă resursele sunt regiștri, unele dependențe se pot rezolva prin redenumire


Tipuri de dependențe de date

- RAW (*read after write*)
 - prima instrucțiune modifică resursa, a doua o citește
- WAR (*write after read*)
 - invers
- WAW (*write after write*)
 - ambele instrucțiuni modifică resursa

Dependențe RAW

- dependențe "adevărate"
- nu pot fi eliminate

```
mov  eax, 5  
sub  ebx, eax
```

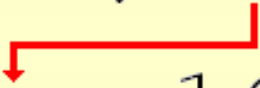


- valoarea scrisă în `eax` de prima instrucțiune este necesară instrucțiunii următoare

Dependențe WAR

- numite și antidependențe

```
add esi, eax
mov eax, 16
sub ebx, eax
```



- prima instrucțiune trebuie executată înaintea celei de-a doua (nu se poate în paralel)

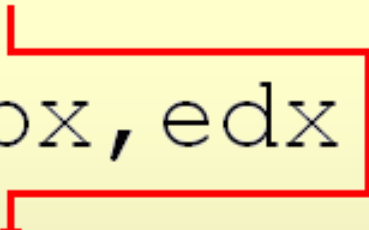
Dependențe WAR - rezolvare

add esi,eax		add esi,eax
mov eax,16	→	mov reg_tmp,16
sub ebx,eax		sub ebx,reg_tmp


Dependențe WAW

- dependențe de ieșire

```
div ecx
sub ebx, edx
mov eax, 5
add ebp, eax
```



Dependențe WAW - rezolvare

<code>div ecx</code>		<code>div ecx, 3</code>
<code>sub ebx, edx</code>		<code>sub ebx, edx</code>
<code>mov eax, 5</code>		<code>mov reg_tmp, 5</code>
<code>add ebp, eax</code>		<code>add ebp, reg_tmp</code>

Utilitate

- ajută la creșterea performanței?
- crește potențialul de paralelizare
- mai eficientă în combinație cu alte tehnici
 - structura superpipeline
 - execuția out-of-order

Eficiența abordării

- mai mulți regiștri
 - folosiți intern de procesor
 - nu sunt accesibili programatorului
- de ce?
 - redenumirea se face automat
 - programatorul poate greși (exploatare ineficientă a resurselor)
 - creșterea performanței programelor vechi

II.8. Hyperthreading

Hyperthreading (1)

- avem un singur procesor real, dar acesta apare ca două procesoare virtuale
- sunt duplicate componentele care rețin starea procesorului
 - regiștrii generali
 - regiștrii de control
 - regiștrii controllerului de întreruperi
 - regiștrii de stare ai procesorului

Hyperthreading (2)

- nu sunt duplicate resursele de execuție
 - unitățile de execuție
 - unitățile de predicție a salturilor
 - magistralele
 - unitatea de control a procesorului
- procesoarele virtuale execută instrucțiunile în mod întrepătruns

De ce hyperthreading?

- exploatează mai bine structura pipeline
 - când o instrucțiune a unui procesor virtual se blochează, celălalt procesor preia controlul
- nu oferă același câștig de performanță ca un al doilea procesor real
- dar complexitatea și consumul sunt aproape aceleași cu ale unui singur procesor
 - componentele de stare sunt foarte puține

II.9. Arhitectura RISC

Structura clasică a CPU

CISC (*Complex Instruction Set Computer*)

- număr mare de instrucțiuni
- complexitate mare a instrucțiunilor → timp mare de execuție
- număr mic de regiștri → acces intensiv la memorie

Observații practice

- multe instrucțiuni sunt rar folosite
- 20% din instrucțiuni sunt executate 80% din timp
 - sau 10% sunt executate 90% din timp
 - depinde de sursa de documentare...
- instrucțiunile complexe pot fi simulate prin instrucțiuni simple

Structura alternativă

RISC (*Reduced Instruction Set Computer*)

- set de instrucțiuni simplificat
 - instrucțiuni mai puține (relativ)
 - și mai simple funcțional (elementare)
- număr mare de regiștri (zeci)
- mai puține moduri de adresare a memoriei

Accesele la memorie

- format fix al instrucțiunilor
 - același număr de octeți, chiar dacă nu toți sunt necesari în toate cazurile
 - mai simplu de decodificat
- arhitectură de tip *load/store*
 - accesul la memorie - doar prin instrucțiuni de transfer între memorie și regiștri
 - restul instrucțiunilor lucrează doar cu regiștri

Avantajele structurii RISC

- instrucțiuni mai rapide
- reduce numărul de accese la memorie
 - depinde de capacitatea compilatoarelor de a folosi regiștrii
- accesele la memorie - mai simple
 - mai puține blocaje în pipeline
- necesar de siliciu mai mic - se pot integra circuite suplimentare (ex. cache)

II.10. Arhitecturi paralele de calcul

Calcul paralel - utilizare

- comunicare între aplicații
 - poate fi folosită și procesarea concurentă
- performanțe superioare
 - calcule științifice
 - volume foarte mari de date
 - modelare/simulare
 - meteorologie, astronomie etc.

Cum se obține paralelismul?

- structuri pipeline
 - secvențial/paralel
- sisteme multiprocesor
 - unitățile de calcul - procesoare
- sisteme distribuite
 - unitățile de calcul - calculatoare

Performanța

- ideal - viteza variază liniar cu numărul de procesoare
- real - un program nu poate fi paralelizat în întregime
- exemple
 - operații de I/O
 - sortare-interclasare

Scalabilitatea (1)

- creșterea performanței o dată cu numărul procesoarelor
- probleme - sisteme cu foarte multe procesoare
- factori de limitare
 - complexitatea conexiunilor
 - timpul pierdut pentru comunicare
 - natura secvențială a aplicațiilor

Scalabilitatea (2)

- funcțională pentru un număr relativ mic de procesoare
- număr relativ mare
 - creșterea de performanță nu urmează creșterea numărului de procesoare
- număr foarte mare
 - performanța se plafonează sau poate scădea

Sisteme de memorie

- după organizarea fizică a memoriei
 - centralizată
 - distribuită
- după tipul de acces la memorie
 - comună (partajată)
 - locală

Tipuri de sisteme multiprocesor

- sisteme cu memorie partajată centralizată
- sisteme cu memorie partajată distribuită
- sisteme cu schimb de mesaje

Memorie partajată centralizată

- denumiri
 - UMA (*Uniform Memory Access*)
 - SMP (*Symmetrical Multiprocessing*)
- memorie comună
- accesibilă tuturor procesoarelor
- timpul de acces la memorie
 - același pentru orice procesor și orice locație

Memorie partajată distribuită

- denumiri
 - DSM (*Distributed Shared Memory*)
 - NUMA (*Non-Uniform Memory Access*)
- memoria este distribuită fizic
- spațiu de adrese unic
 - văzut de toate procesoarele
- accesul la memorie - neuniform

Sisteme cu schimb de mesaje

- multicalculatoare
- fiecare procesor are propria memorie locală
 - nu este accesibilă celorlalte procesoare
- comunicarea între procesoare
 - transmiterea de mesaje explicite
 - similar rețelelor de calculatoare

Comunicare

- conectarea
 - între procesoare
 - între procesoare și memorie
- variante
 - sisteme cu memorie partajată centralizată - magistrală
 - sisteme cu memorie partajată distribuită - rețele de interconectare

Comunicare pe magistrală

- economic
- proiectare simplă
- performanță mai redusă
- nu scalează bine cu numărul de procesoare
- rol important - memoriile cache
 - probleme de coerență

Rețele de interconectare (1)

- între
 - procesoare
 - procesoare și memorie
- scop
 - flexibilitate
 - performanță
 - mai multe accese în paralel

Rețele de interconectare (2)

- tipuri de conectare
 - totală - fiecare cu fiecare
 - parțială - unele perechi de componente nu sunt conectate direct
- conectare procesoare-memorii
 - mai multe circuite de memorie
 - pot fi accesate în paralel de procesoare diferite

Coerența memoriei

- trebuie ca toate procesoarele să folosească ultima valoare scrisă pentru o variabilă partajată
- problema - memoriile cache
- scopul - orice variabilă partajată să aibă aceeași valoare
 - în toate cache-urile
 - în memoria principală

Coerența - cache *write-back*

x - variabilă partajată

Procesor	Acțiune	Valoare cache A	Valoare cache B	Valoare memorie
				9
A	i=x;	9		9
B	j=x;	9	9	9
A	x=5;	5	9	9
B	k=x;	5	9	9

Coerența - *cache write-through*

x - variabilă partajată

Procesor	Acțiune	Valoare cache A	Valoare cache B	Valoare memorie
				9
A	i = x ;	9		9
B	j = x ;	9	9	9
A	x = 5 ;	5	9	5
B	k = x ;	5	9	5

Ce înseamnă coerența? (1)

1. ordinea execuției

- a) procesorul P scrie în variabila X
- b) apoi procesorul P citește X
- între a) și b) nu există alte scrieri în X

→ citirea b) returnează valoarea scrisă de a)

Ce înseamnă coerența? (2)

2. viziune coerentă a memoriei

- a) procesorul P scrie în variabila X
 - b) apoi procesorul Q ($Q \neq P$) citește X
 - între a) și b) nu există alte scrieri în X
 - între a) și b) a trecut suficient timp
- citirea b) returnează valoarea scrisă de a)

Ce înseamnă coerența? (3)

3. serializarea scrierilor

- a) procesorul P scrie în variabila X
- b) procesorul Q ($Q=P$ sau $Q \neq P$) scrie în variabila X

→ toate procesoarele văd cele două scrieri în aceeași ordine

– nu neapărat a) înaintea b)

Menținerea coerenței cache-urilor

- protocoale de menținere a coerenței
- se bazează pe informațiile despre liniile de cache
 - *invalid* - datele nu sunt valide
 - *dirty* - doar cache-ul curent deține valoarea actualizată
 - *shared* - cache-ul curent deține valoarea actualizată, la fel memoria principală și eventual alte cache-uri

Tipuri de protocoale

- *directory based*
 - informațiile despre fiecare linie de cache -
ținute într-un singur loc
- *snooping*
 - fiecare cache are o copie a liniei partajate
 - fără centralizarea informației
 - cache-urile monitorizează magistrala
 - detectează schimbările produse în liniile de cache

Actualizarea cache-urilor

- fiecare cache anunță modificările făcute
- celelalte cache-uri reacționează
- contează doar operațiile de scriere
- variante
 - scriere cu invalidare (*write invalidate*)
 - scriere cu actualizare (*write update*)

Scriere cu invalidare (1)

- un procesor modifică o dată
- modificarea se face în cache-ul propriu
 - toate celelalte cache-uri sunt notificate
- celelalte cache-uri
 - nu au o copie a datei modificate - nici o acțiune
 - au o copie a datei modificate - își invalidează linia corespunzătoare
 - valoarea corectă va fi preluată când va fi nevoie

Scriere cu invalidare (2)

x - variabilă partajată

Proc.	Acțiune	Reacție cache	Cache A	Cache B	Memorie
					9
A	i=x;	read miss	9		9
B	j=x;	read miss	9	9	9
A	x=5;	invalidation	5	inv.	9
B	k=x;	read miss	5	5	5

Scriere cu actualizare (1)

- un procesor modifică o dată
- modificarea se face în cache-ul propriu
 - toate celelalte cache-uri sunt notificate
 - se transmite noua valoare
- celelalte cache-uri
 - nu au o copie a datei modificate - nici o acțiune
 - au o copie a datei modificate - preiau noua valoare

Scriere cu actualizare (2)

x - variabilă partajată

Proc.	Acțiune	Reacție cache	Cache A	Cache B	Memorie
					9
A	i=x;	read miss	9		9
B	j=x;	read miss	9	9	9
A	x=5;	invalidation	5	5	5
B	k=x;	read hit	5	5	5

Invalidare vs. actualizare (1)

- mai multe scrieri succesive în aceeași locație
 - *write invalidation* - o singură invalidare (prima dată)
 - *write update* - câte o actualizare pentru fiecare scriere
 - mai avantajos - invalidare

Invalidare vs. actualizare (2)

- mai multe scrieri în aceeași linie de cache
 - modificarea unei locații necesită invalidarea/actualizarea întregii linii
 - *write invalidation* - o singură invalidare (prima dată)
 - *write update* - câte o actualizare pentru fiecare scriere
 - mai avantajos - invalidare

Invalidare vs. actualizare (3)

- "timpul de răspuns"
 - timpul între modificarea unei valori la un procesor și citirea noii valori la alt procesor
 - *write invalidation* - întâi invalidare, apoi citire (când este necesar)
 - *write update* - actualizare imediată
 - mai avantajos - actualizare

Invalidare vs. actualizare (4)

- ambele variante au avantaje și dezavantaje
- scrierea cu invalidare - ocupare (mult) mai redusă a memoriei și magistralelor
- scrierea cu actualizare - rata de succes a cache-urilor mai mare
- mai des folosită - invalidarea

III. Dispozitivele periferice

Dispozitiv periferic

- realizează o formă oarecare de comunicare între procesor și "lumea exterioară"
- pentru gestiunea comunicării cu procesorul, include un controller I/O
 - de obicei, acesta conține o serie de regiștri care rețin informații necesare pentru comunicare
 - date
 - informații de stare
 - comenzi (de la procesor)

Intrări-ieșiri (I/O)

- cum privește sistemul comunicarea
 - I/O proiectat pe memorie
 - citirile/scrierile sunt văzute ca și cum ar fi realizate în locații de memorie
 - adresele I/O - în spațiul de adrese de memorie
 - aceleași semnale de control ca pentru memorie
 - I/O izolate
 - adresele I/O - separate de adresele de memorie
 - semnale de control diferite de ale memoriei

Moduri de comunicare (1)

- intrări-ieșiri programate
 - *programmed I/O*
 - programul așteaptă într-o buclă până când perifericul inițiază un transfer
 - eficient dacă se știe dinainte momentul când perifericul va solicita comunicarea
 - consum inutil de timp de execuție al procesorului

Moduri de comunicare (2)

- acces direct la memorie
 - *Direct Memory Access* (DMA)
 - un controller specializat (DMA controller) efectuează transferurile
 - foarte rapid
 - preia controlul magistralelor și transferă datele direct între periferic și memorie
 - fără intervenția procesorului
 - util pentru transferul volumelor mari de date

Moduri de comunicare (3)

- intrări-ieșiri gestionate prin întreruperi
 - interrupt-driven I/O
 - când un periferic dorește să inițieze comunicarea, notifică procesorul
 - printr-o cerere de întrerupere
 - în restul timpului, procesorul poate îndeplini alte sarcini
 - metoda cea mai flexibilă

Magistrale (1)

- căi de comunicație a informației
- o magistrală leagă între ele mai mult de 2 componente printr-o cale unică
- descrierea unei magistrale
 - semnalele electrice folosite
 - reguli de comunicare - trebuie respectate de toate părțile implicate
 - mod de conectare

Magistrale (2)

Accesul la magistrală

- mai multe entități pot solicita simultan accesul
- este necesară o procedură de arbitraj
 - decide cine va primi accesul
 - celelalte trebuie să aștepte eliberarea magistralei

Arbitrarea magistralelor

Tipuri de arbitrare a magistralelor

- centralizat
 - decizia o ia un circuit specializat (arbitru)
- descentralizat
 - componentele se înțeleg între ele
 - pe baza regulilor care definesc funcționarea magistralei

Conectarea la magistrală

- probleme de natură electrică
- mai multe circuite conectate împreună
 - în intrare și ieșire
- nu se pot conecta mai multe ieșiri
 - nivelele diferite de tensiune ar distruge circuitele
- o soluție - multiplexarea
 - toate ieșirile sunt conectate la un multiplexor

Circuite *tri-state*

- ieșirea are 3 stări posibile
 - 0
 - 1
 - impedanță infinită (*High-Z*)
- primele două corespund valorilor obișnuite
- a treia implică decuplarea de pe magistrală
 - ca și cum ieșirea circuitului nici nu ar fi conectată la magistrală

Circuite *open-collector*

- în unele cazuri se numesc *open-drain*
 - în funcție de tehnologia utilizată
- este posibilă conectarea mai multor ieșiri simultan
- valoarea rezultată - funcția AND între ieșirile conectate

Magistrale - privire generală

Avantaje

- activitatea pe magistrală - ușor de controlat
- economic - structură relativ simplă

Dezavantaj

- performanțe mai scăzute
 - doar 2 componente pot comunica la un moment dat

IV. Sistemul de întreruperi

Ce este o întrerupere?

- procesorul poate suspenda execuția programului curent
- scop - tratarea unor situații neprevăzute
- după tratare se reia programul întrerupt
- scopul inițial - comunicarea cu perifericele
- procesorul nu "așteaptă" perifericele
 - acestea îl solicită când este necesar

Întreruperi hardware

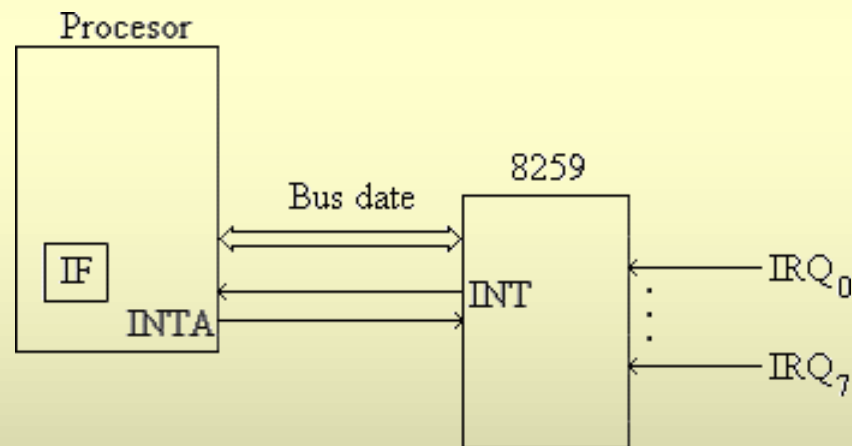
- dezactivabile (*maskable*)
 - procesorul poate refuza tratarea lor
 - depinde de valoarea bistabilului IF (*Interrupt Flag*): 1 - acceptare, 0 - refuz
 - IF poate fi modificat prin program
- nedezactivabile (*non-maskable*)
 - procesorul le tratează întotdeauna

Controllerul de întreruperi (1)

- circuit specializat
- preia cererile de întrerupere de la periferice
- le trimite procesorului
- rezolvă conflictele (mai multe cereri simultane) - arbitrare
 - fiecare periferic are o anumită prioritate

Controllerul de întreruperi (2)

- inițial - Intel 8259
 - se pot folosi mai multe (cascadare)



- astăzi - integrat în chipset

Tratarea întreruperilor - etape (1)

- dispozitivul periferic generează o cerere de întrerupere pe linia sa IRQ_i
- controllerul trimite un semnal pe linia INT
- procesorul verifică valoarea bistabilului IF
 - numai pentru întreruperi dezactivabile
 - dacă este 0 - refuză cererea; stop
 - dacă este 1 - răspunde cu un semnal INTA

Tratarea întreruperilor - etape (2)

- se întrerupe execuția programului curent
- se salvează în stivă regiștrii (inclusiv PC)
- se șterge bistabilul IF
 - se blochează execuția altei întreruperi în timpul execuției programului pentru întreruperea în curs
 - poate fi repus pe 1 prin program

Tratarea întreruperilor - etape (3)

- identificare periferic (sursa cererii)
 - controllerul depune pe magistrala de date un octet *type*
 - indică perifericul care a făcut cererea
 - maximum $2^8=256$ surse de întrerupere
 - fiecare sursă are rutina proprie de tratare
 - periferice diferite - tratări diferite

Tratarea întreruperilor - etape (4)

- determinarea adresei rutinei de tratare
 - adresa fizică 0 - tabelul vectorilor de întreruperi
 - conține adresele tuturor rutinelor de tratare
 - dimensiune: $256 \text{ adrese} \times 4 \text{ octeți} = 1 \text{ KB}$
 - octet type = $n \rightarrow$ adresa rutinei de tratare: la adresa $n \times 4$

Tratarea întreruperilor - etape (5)

- salt la rutina de tratare a întreruperii
- execuția rutinei de tratare
- revenire în programul întrerupt
 - restabilire valoare bistabil IF
 - restabilire valoare regiștri (din stivă)
 - reluarea execuției programului de unde a fost întrerupt

Extindere

- sistem puternic și flexibil
- poate fi extins - utilizare mai largă
- programele trebuie întrerupte și în alte situații (nu doar pentru comunicarea cu perifericele)
- util mai ales pentru sistemul de operare

Tipuri de întreruperi

- *hardware* - generate de periferice
- *excepții (traps)* - generate de procesor
 - semnalează o situație anormală
 - exemplu: împărțire la 0
- *software* - generate de programe
 - folosite pentru a solicita anumite servicii sistemului de operare

V. Sistemul de operare

Rolul sistemului de operare (1)

- program cu rol de gestiune a sistemului de calcul
- face legătura între hardware și aplicații
- pune la dispoziția aplicațiilor diferite servicii
- supraveghează buna funcționare a aplicațiilor
 - poate interveni în cazurile de eroare

Rolul sistemului de operare (2)

- pentru a-și îndeplini sarcinile, are nevoie de suport hardware
 - cel mai important - sistemul de întreruperi
- componente principale
 - nucleu (*kernel*)
 - drivere

V.1. Nucleul sistemului de operare

Nucleul sistemului de operare

- în mare măsură independent de structura hardware pe care rulează
- "creierul" sistemului de operare
- gestiunea resurselor calculatorului - hardware și software
 - funcționare corectă
 - alocare echitabilă între aplicații

Moduri de lucru ale procesorului

- utilizator (*user mode*)
 - restricționat
 - accesul la memorie - numai anumite zone
 - accesul la periferice - interzis
- nucleu (*kernel mode*)
 - fără restricții

Modul de rulare al programelor

- sistemul de operare - în mod nucleu
 - poate efectua orice operație
- aplicațiile - în mod utilizator
 - nu pot realiza anumite acțiuni
 - apelează la nucleu

Trecerea între cele două moduri

- prin sistemul de întreruperi
- utilizator → nucleu
 - apel întrerupere software
 - generare excepție (eroare)
- nucleu → utilizator
 - revenire din rutina de tratare a întreruperii

Consecințe

- codul unei aplicații nu poate rula în modul nucleu
- avantaj: erorile unei aplicații nu afectează alte programe
 - aplicații
 - sistemul de operare
- dezavantaj: pierdere de performanță

Structura nucleului

- nu este un program unitar
- colecție de rutine care cooperează
- funcții principale
 - gestiunea proceselor
 - gestiunea memoriei
 - gestiunea sistemelor de fișiere
 - comunicarea între procese

V.2. Apeluri sistem

Apeluri sistem (*system calls*)

- cereri adresate nucleului de către aplicații
- acțiuni pe care aplicațiile nu le pot executa singure
 - numai în modul nucleu al procesorului
 - motiv - siguranța sistemului
- realizate prin întreruperi software
- similar apelurilor de funcții

Etapele unui apel sistem (1)

1. programul depune parametrii apelului sistem într-o anumită zonă de memorie
2. se generează o întrerupere software
 - procesorul trece în modul nucleu
3. se identifică serviciul cerut și se apelează rutina corespunzătoare

Etapele unui apel sistem (2)

4. rutina preia parametrii apelului și îi verifică
– dacă sunt erori - apelul eșuează
5. dacă nu sunt erori - realizează acțiunea cerută
6. terminarea rutinei - rezultatele obținute sunt depuse într-o zonă de memorie accesibilă aplicației care a făcut apelul

Etapele unui apel sistem (3)

7. procesorul revine în mod utilizator
8. se reia execuția programului din punctul în care a fost întrerupt
 - se utilizează informațiile memorate în acest scop la apariția întreruperii
9. programul poate prelua rezultatele apelului din zona în care au fost depuse

Apeluri sistem - concluzii

- comunicare între aplicație și nucleu
 - depunere parametri
 - preluare rezultate
- acțiunile critice sunt realizate într-un mod sigur
- mari consumatoare de timp
- apeluri cât mai rare - lucru cu buffere

Cum folosim bufferele

Exemplu - funcția *printf*

- formatează textul, apoi îl trimite spre ecran
- nu are acces direct la hardware
 - se folosește de un apel sistem
 - *write* (în Linux)
- de fapt, *printf* depune textul formatat într-o zonă proprie de memorie (buffer)
 - doar când bufferul e plin se face un apel sistem

V.3. Drivere

Ce sunt driverele?

- module de program care gestionează comunicarea cu perifericele
 - specializate - câte un driver pentru fiecare periferic
- parte a sistemului de operare
 - acces direct la hardware
 - se execută în modul nucleu al procesorului

Utilizare

- nu fac parte din nucleu
 - dar se află sub comanda nucleului
- folosite de rutinele de tratare ale întreruperilor hardware
- înlocuire periferic → înlocuire driver
 - organizare modulară
 - nu trebuie reinstalat tot sistemul de operare

V.4. Gestiunea proceselor

Procese (1)

- se pot lansa în execuție mai multe programe în același timp (*multitasking*)
- paralelismul nu este real
 - doar dacă sistemul are mai multe procesoare
 - altfel - concurență
- un program se poate împărți în mai multe secvențe de instrucțiuni - *procese*
 - se pot executa paralel sau concurent

Procese (2)

- sistemul de operare lucrează cu procese
 - nu cu programe
- la lansare, un program constă dintr-un singur proces
 - poate crea alte procese
 - care pot crea alte procese ș.a.m.d.
- un procesor poate executa la un moment dat instrucțiunile unui singur proces

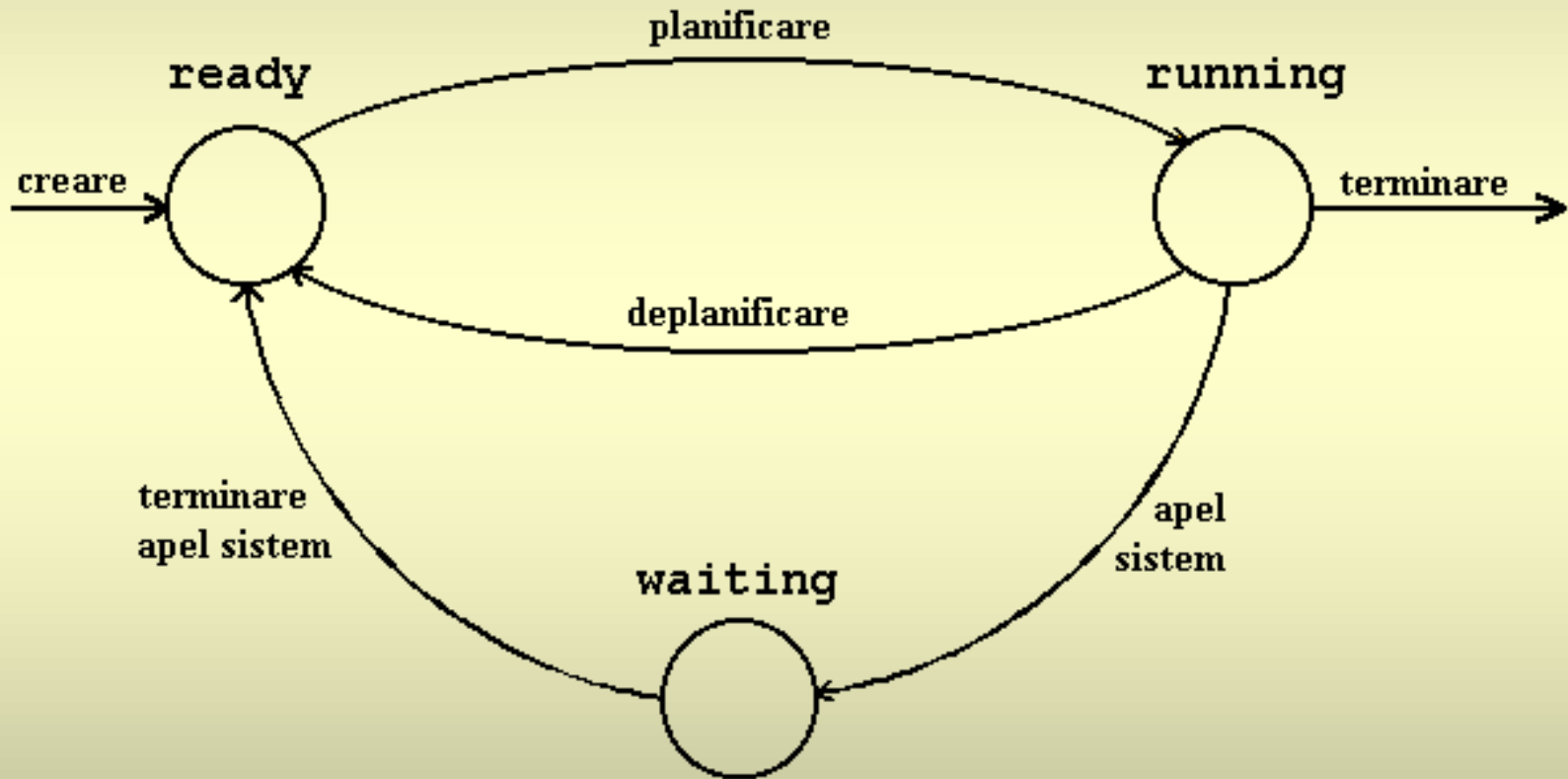
Procese (3)

- fiecare proces are propriile zone de memorie (cod, date, stivă, ...)
 - separate de ale celorlalte procese
- la crearea unui nou proces, i se alocă spațiu de memorie
- la terminarea unui proces, memoria ocupată este eliberată
 - chiar dacă programul în ansamblul său continuă

Stările unui proces (1)

- în execuție (*running*)
 - instrucțiunile sale sunt executate de procesor
- gata de execuție (*ready*)
 - așteaptă să fie executat de procesor
- în așteptare (*waiting*)
 - așteaptă terminarea unui apel sistem
 - nu concurează momentan pentru planificarea la procesor

Stările unui proces (2)



Stările unui proces (3)

- procesul aflat în execuție părăsește această stare
 - la terminarea sa
 - normală sau în urma unei erori
 - la efectuarea unui apel sistem (\rightarrow *waiting*)
 - când instrucțiunile sale au fost executate un timp suficient de lung și este rândul altui proces să fie executat (deplanificare)

Forme de multitasking

- non-preemptiv
 - nu permite deplanificarea unui proces
 - un proces poate fi scos din execuție doar în celelalte situații
 - dezavantaj - erorile de programare pot bloca procesele (ex. buclă infinită)
- preemptiv

Deplanificarea

- cum știe sistemul de operare cât timp s-a executat un proces?
- este necesară o formă de măsurare a timpului
- ceasul de timp real
 - dispozitiv periferic
 - generează cereri de întrerupere la intervale regulate de timp

Fire de execuție (1)

- un proces se poate împărți la rândul său în mai multe fire de execuție (*threads*)
 - uzual, un fir constă în execuția unei funcții din codul procesului
- firele de execuție partajează resursele procesului (memorie, fișiere deschise etc.)
 - comunicare mai simplă - prin variabile globale
 - risc mai mare de interferențe nedorite

Fire de execuție (2)

- când un proces este planificat la procesor, se va executa unul dintre firele sale
 - deci este necesară și aici o formă de planificare
- cine realizează planificarea?
- variante
 - sistemul de operare (mai rar)
 - aplicația - prin funcții de bibliotecă specializate

V.5. Gestiunea memoriei

Gestiunea memoriei

Funcții

- alocarea zonelor de memorie către aplicații
- prevenirea interferențelor între aplicații
- detectarea și oprirea acceselor incorecte

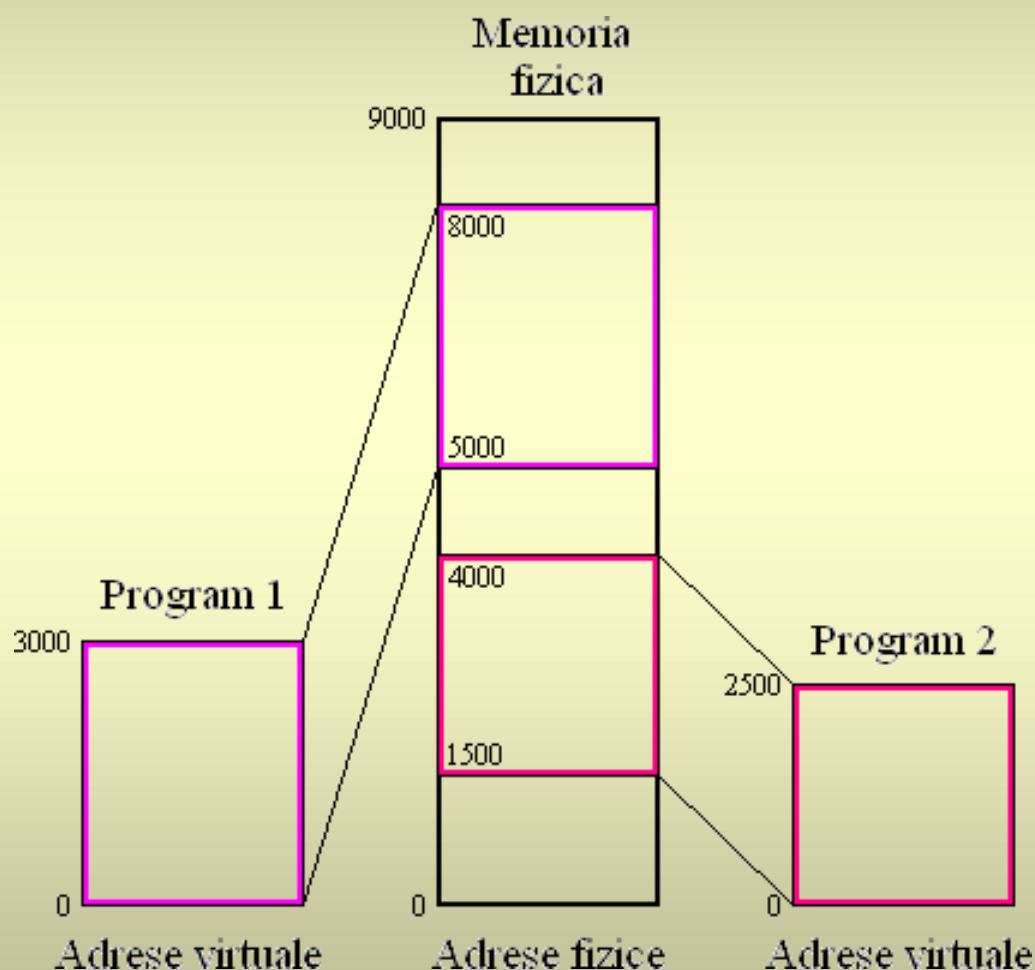
Problema fundamentală

- mai multe aplicații → zone de memorie disjuncte
- fiecare aplicație → anumite zone de memorie; care sunt aceste zone?
 - depind de ocuparea memoriei la acel moment
 - nu pot fi cunoscute la compilare

Soluția

- două tipuri de adrese
 - virtuale - aplicația crede că le accesează
 - fizice - sunt accesate în realitate
- corespondența între adresele virtuale și cele fizice - gestionată de sistemul de operare

Adrese virtuale și fizice (1)



Adrese virtuale și fizice (2)

Gestionarea adreselor fizice și virtuale

- 2 metode diferite
 - segmentare
 - paginare
- pot fi folosite și împreună
- componentă dedicată a procesorului - MMU (*Memory Management Unit*)

V.5.1. Segmentarea memoriei

Principiul de bază (1)

- segment - zonă continuă de memorie
- conține informații de același tip (cod, date etc.)
- vizibil programatorului
- adresa unei locații - formată din 2 părți
 - adresa de început a segmentului
 - deplasamentul în cadrul segmentului (*offset*)

Principiul de bază (2)

- la rulări diferite ale programului, segmentele încep la adrese diferite
- efectul asupra adreselor locațiilor
 - adresa de început a segmentului trebuie actualizată
 - deplasamentul - nemodificat
- problema este rezolvată numai parțial
 - dorim ca adresa să nu fie modificată deloc

Descriptori (1)

- descriptor de segment - structură de date pentru gestionarea unui segment
- informații reținute
 - adresa de început
 - dimensiunea
 - drepturi de acces
 - etc.

Descriptori (2)

- descriptorii - plasați într-un tabel
- accesul la un segment - pe baza indicelui în tabelul de descriptori (selector)
- adresa virtuală - 2 componente
 - indicele în tabelul de descriptori
 - deplasamentul în cadrul segmentului
- adresa fizică = adresa de început a segmentului + deplasamentul

Descriptori (3)

- la rulări diferite ale programului, segmentele încep la adrese diferite
- efectul asupra adreselor locațiilor
 - nici unul
 - trebuie modificată doar adresa de început a segmentului în descriptor
 - o singură dată (la încărcarea segmentului în memorie)
 - sarcina sistemului de operare

Accesul la memorie (1)

- programul precizează adresa virtuală
- identificare descriptor segment
- verificare drepturi acces
 - drepturi insuficiente - generare excepție
- verificare deplasament
 - dacă deplasamentul depășește dimensiunea segmentului - generare excepție

Accesul la memorie (2)

- dacă s-a produs o eroare la pașii anteriori
 - rutina de tratare a excepției termină programul
- dacă nu s-a produs nici o eroare
 - calcul adresă fizică (adresă început segment + deplasament)
 - acces la adresa calculată

Exemplificare (1)

Tabel descriptori (simplificat)

Indice	Adresa început	Dimensiune
0	65000	43000
1	211000	15500
2	20000	30000
3	155000	49000
4	250000	35000

Exemplificare (2)

Exemplu 1:

```
mov byte ptr ds:[eax], 25
```

- $ds = 3 \rightarrow$ adresa început segment = 155000
- $eax = 27348 < 49000$
 - deplasament valid (nu depășește dimensiunea segmentului)
- adresa fizică: $155000 + 27348 = 182348$

Exemplificare (3)

Exemplu 2:

```
add dword ptr ss:[ebp], 4
```

- $ss = 1 \rightarrow$ adresa început segment = 211000
- $ebp = 19370 > 15500$
 - deplasament invalid (depășește dimensiunea segmentului)
 - eroare \rightarrow generare excepție

Cazul Intel (1)

3 tabele de descriptori

- global (GDT - *Global Descriptor Table*)
 - accesibil tuturor proceselor
- local (LDT - *Local Descriptor Table*)
 - specific fiecărui proces
- de întreruperi (IDT - *Interrupt Descriptor Table*)
 - nu este direct accesibil aplicațiilor

Cazul Intel (2)

Segmentele - accesate cu ajutorul selectorilor

Structura unui selector (16 biți)

- primii 13 biți - indicele în tabelul de descriptori
 - maximum 8192 descriptori/tabel
- 1 bit - tabelul folosit (global/local)
- ultimii 2 biți - nivelul de privilegii
 - 0 - cel mai înalt, 3 - cel mai scăzut

Cazul Intel (3)

31	24				19	16	15	14	13	12		8	7			0
BASE 31-24					G	D / B	0	A V L	LIMIT 19-16		P	DPL	TYPE		BASE 23-16	
BASE 15-0										LIMIT 15-0						

Cazul Intel (4)

- intervin nivelele de privilegii a 3 entități
 1. CPL (*Current Privilege Level*)
 - al procesului - reținut de procesor
 2. RPL (*Requested Privilege Level*)
 - cel solicitat - preluat din selector
 3. DPL (*Descriptor Privilege Level*)
 - cel al segmentului accesat - din descriptor

Cazul Intel (5)

- relațiile dintre aceste nivele de privilegii decid dacă se poate realiza accesul
- condiția pentru realizarea accesului:
 $CPL \leq DPL$ și $RPL \leq DPL$ (simultan)
- orice altă situație indică o încercare de acces la un nivel prea înalt
 - generare excepție

Zone libere și ocupate (1)

Problemă

- crearea unui segment nou → plasare în memorie
- este necesară o zonă liberă continuă suficient de mare
- pot exista mai multe asemenea zone - care este aleasă?

Zone libere și ocupate (2)

Algoritmi de plasare în memorie

- *First Fit* - prima zonă liberă găsită suficient de mare
- *Best Fit* - cea mai mică zonă liberă suficient de mare
- *Worst Fit* - cea mai mare zonă liberă (dacă este suficient de mare)

Zone libere și ocupate (3)



Fragmentare (1)

Fragmentarea externă a memoriei

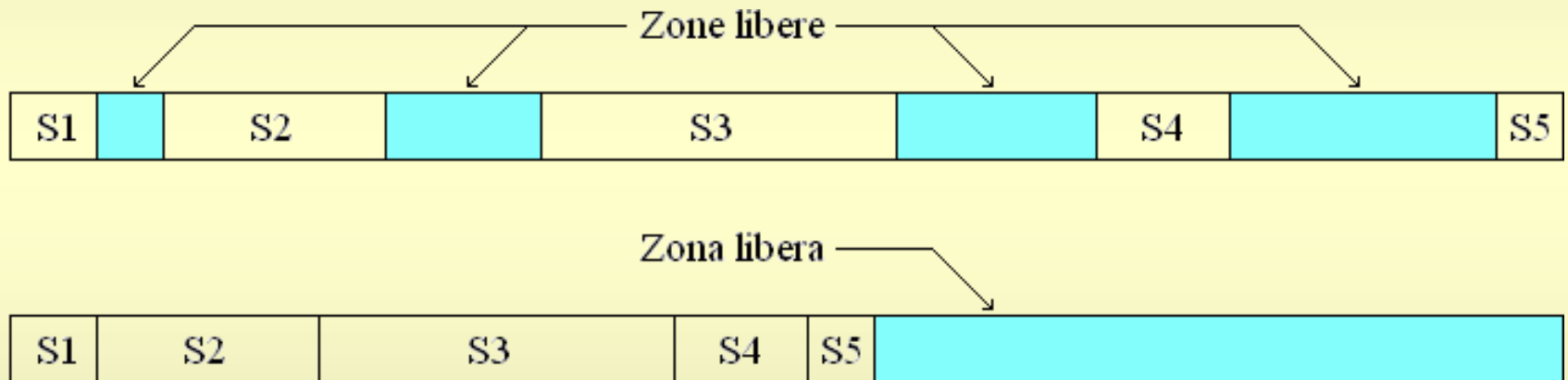
- multe zone libere prea mici pentru a fi utilizate
- apare după un număr mare de alocări și eliberări de segmente
- indiferent de algoritmul folosit
- plasarea unui segment poate eșua, chiar dacă spațiul liber total ar fi suficient

Fragmentare (2)

Eliminarea fragmentării externe

- compactarea memoriei
 - deplasarea segmentelor astfel încât să nu mai existe zone libere între ele
 - se crează o singură zonă liberă, de dimensiune maximă
 - realizată de un program specializat, parte a sistemului de operare

Compactare (1)



Compactare (2)

Rularea programului de compactare a memoriei

- consumă mult timp
 - mutarea segmentelor în memorie
 - actualizarea descriptorilor de segment
- nu poate fi rulat foarte des
- numai când este necesar

Compactare (3)

Situații în care se poate decide rularea programului de compactare

- când plasarea în memorie a unui segment eșuează din lipsă de spațiu
- la intervale regulate de timp
- când gradul de fragmentare a memoriei depășește un anumit nivel

Segmentarea - concluzii

Probleme ale mecanismului de segmentare

- gestiune complicată
 - suprapunerea segmentelor - greu de detectat
- fragmentarea externă - de obicei puternică
 - mult spațiu liber nefolosit
- compactarea consumă timp

V.5.2. Paginarea memoriei

Principiul de bază

- spațiul adreselor virtuale - împărțit în pagini (*pages*)
 - zone de dimensiune fixă
- spațiul adreselor fizice - împărțit în cadre de pagină (*page frames*)
 - aceeași dimensiune ca și paginile
- dimensiune - uzual 4 KB

Tabele de paginare (1)

- corespondența între pagini și cadre de pagină - sarcina sistemului de operare
- structura de bază - tabelul de paginare
- câte unul pentru fiecare proces care rulează
- permite detectarea acceselor incorecte la memorie

Tabele de paginare (2)

- la rulări diferite ale programului, paginile sunt plasate în cadre diferite
- efectul asupra adreselor locațiilor
 - nici unul
 - trebuie modificat tabelul de paginare
 - o singură dată (la încărcarea paginii în memorie)
 - sarcina sistemului de operare

Accesul la memorie

- programul precizează adresa virtuală
- se determină pagina din care face parte
- se caută pagina în tabelul de paginare
 - dacă nu este găsită - generare excepție
- se determină cadrul de pagină corespunzător
- calcul adresă fizică
- acces la adresa calculată

Exemplificare (1)

Tabel de paginare (simplificat)

Pagini	0	1	2	8	9	11	14	15
Cadre de pagină	5	7	4	3	9	2	14	21

Exemplificare (2)

Exemplu 1:

- dimensiunea paginii: 1000
- adresa virtuală: 8039
 - pagina: $[8039/1000]=8 \rightarrow$ cadrul de pagină 3
 - deplasament: $8039 \% 1000=39$ (în cadrul paginii)
- adresa fizică: $3 \cdot 1000 + 39 = 3039$

Exemplificare (3)

Exemplu 2:

- dimensiunea paginii: 1000
- adresa virtuală: 5276
 - pagina: $[5276/1000]=5$
 - nu apare în tabelul de paginare
 - eroare → generare excepție

Restricții

Construirea tabelelor de paginare

- sarcina sistemului de operare
- trebuie să evite suprapunerile între aplicații
- restricții
 - o pagină virtuală poate să apară pe cel mult o poziție într-un tabel de paginare
 - un cadru de pagină fizică poate să apară cel mult o dată în toate tabelele de paginare existente la un moment dat

Fragmentare (1)

Fragmentarea internă a memoriei

- între pagini nu există spațiu → nu apare fragmentare externă
- fragmentarea internă
 - spațiu liber nefolosit în interiorul unei pagini
 - nu poate fi preluat de alt proces
 - nu se poate face compactare
- mai puțin severă decât fragmentarea externă

Fragmentare (2)

Alegerea dimensiunii paginilor

- putere a lui 2 (nu rămân resturi de pagină)
- odată aleasă, nu se mai schimbă
- se stabilește ca un compromis
 - prea mare - fragmentare internă puternică
 - prea mică - mult spațiu ocupat de tabelele de paginare
 - uzual - 4 KB

Chiar atât de simplu?

Procesor pe 32 biți

- spațiu de adrese: 4 GB ($=2^{32}$)
 - dimensiunea paginii: 4 KB ($=2^{12}$)
- tabel cu 2^{20} elemente
- ar ocupa prea mult loc
 - consumă din memoria disponibilă aplicațiilor
- la procesoarele pe 64 biți - și mai rău

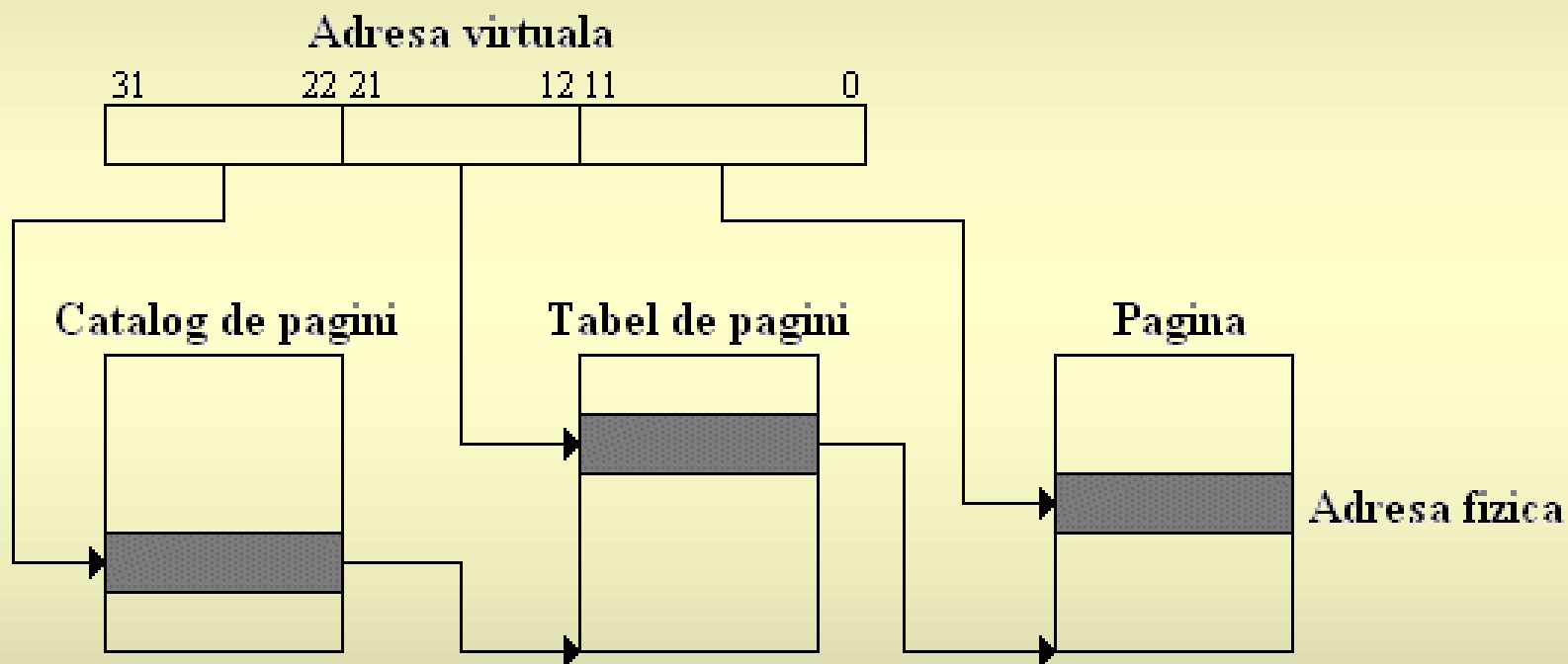
Soluția 1

- tabele de pagini inversate
- nu se rețin toate cele 2^{20} elemente
 - doar paginile folosite
- plasare/căutare în tabel - funcție *hash*
- greu de implementat în hardware
 - viteză
 - evitare coliziuni

Soluția 2

- tabele pe mai multe nivele
- cazul Intel - 2 nivele
 - catalog de pagini (*Page Directory*)
 - tabel de pagini (*Page Table*)
- elementele din catalog - adrese de tabele de pagini
- se alocă doar tabelele de pagini folosite

Structura Intel



Performanța (1)

- cataloagele și tabelele de pagini se află în memorie
 - prea mari pentru a fi reținute în procesor
 - prea multe - specifice fiecărui proces
- efect - performanță scăzută
 - pentru fiecare acces la memorie solicitat de proces - 2 accese suplimentare
- soluția - cache dedicat

Performanța (2)

- TLB (*Translation Lookaside Buffer*)
 - în interiorul procesorului
 - reține corespondențe între pagini virtuale și cadre de pagină fizice
 - ultimele accesate
 - trebuie invalidat atunci când se trece la execuția altui proces

V.5.3. Memoria virtuală

Ideea de pornire

Problema

- aplicațiile - consum mare de memorie
- memoria disponibilă - insuficientă

Cum se poate rezolva?

- capacitatea discului hard - foarte mare
- nu toate zonele de memorie ocupate sunt accesate la un moment dat

Memoria virtuală

Soluția - memoria virtuală (*swap*)

- unele zone de memorie - evacuate pe disc
- când este nevoie de ele, sunt aduse înapoi în memorie

Cine gestionează memoria virtuală?

- sunt necesare informații globale
- sistemul de operare

Fișierul de paginare

- conține zonele de memorie evacuate pe disc
- informații pentru regăsirea unei zone stocate
 - adresele din memorie
 - programul căruia îi aparține
 - dimensiunea
 - etc.

Politica de înlocuire (1)

- problema - aceeași ca la memoria cache
- aducerea unei zone de memorie din fișierul de paginare implică evacuarea alteia
 - care?
- scop - minimizarea acceselor la disc
- politică inefficientă → număr mare de accese la disc → scăderea vitezei

Politica de înlocuire (2)

- set de lucru (*working set*) - zonele de memorie necesare programului la un moment dat
- uzual mult mai mic decât totalitatea zonelor folosite de program
- dacă încapă în memorie - puține accese la disc

Politica de înlocuire (3)

- se va selecta pentru evacuare zona care nu va fi necesară în viitorul apropiat
- nu se poate ști cu certitudine - estimare
 - pe baza comportării în trecutul apropiat
- paginare la cerere (*demand paging*) - evacuare pe disc numai dacă este strict necesar

Implementare

- prin intermediul mecanismelor de gestiune a memoriei, deja discutate
 - dacă un program încearcă să acceseze o locație aflată temporar pe disc, este necesar același tip de detecție
 - memoria virtuală poate fi folosită împreună atât cu segmentarea, cât și cu paginarea
- rolul sistemului de întreruperi - sporit

Accesul la memorie (1)

Cazul paginării

1. programul precizează adresa virtuală
2. se determină pagina din care face parte
3. se caută pagina în tabelul de paginare
4. dacă pagina este găsită - salt la pasul 9
5. generare excepție
6. rutina de tratare caută pagina în fișierul de paginare

Accesul la memorie (2)

Cazul paginării (cont.)

7. dacă pagina nu este în fișierul de paginare - programul este terminat
8. se aduce pagina în memoria fizică
9. se determină cadrul de pagină corespunzător
10. calcul adresă fizică
11. acces la adresa calculată

Reducerea acceselor la disc (1)

- duce la creșterea performanței
- o pagină este salvată pe disc și readusă în memorie de mai multe ori
- readucerea în memorie - copia de pe disc nu este ștearsă
- pagina și copia sa de pe disc sunt identice până la modificarea paginii din memorie

Reducerea acceselor la disc (2)

- evacuarea unei pagini din memorie
 - dacă nu a fost modificată de când se află în memorie - nu mai trebuie salvată
 - util mai ales pentru paginile de cod
- este necesar sprijin hardware pentru detectarea acestei situații
 - este suficient să fie detectate operațiile de scriere

Reducerea acceselor la disc (3)

- tabelul de paginare - structură extinsă
 - fiecare pagină are un bit suplimentar (*dirty bit*)
 - indică dacă pagina a fost modificată de când a fost adusă în memorie
 - resetat la aducerea paginii în memorie
- instrucțiune de scriere în memorie
 - procesorul setează bitul paginii care conține locația modificată

V.5.4. Comunicarea între procese

Comunicare (1)

- pentru a putea coopera, procesele trebuie să-și poată transmite date
 - uneori volume mari
- implementare fizică - zone de memorie comune
 - variabile partajate
 - structuri de date mai complexe, prevăzute cu metode specifice de acces

Comunicare (2)

- este necesar ca două sau mai multe procese să acceseze aceeași zonă de memorie
 - același segment să apară simultan în tabelele de descriptori ale mai multor procese
 - același cadru de pagină să apară simultan în tabelele de paginare ale mai multor procese
- în oricare caz, sistemul de operare controlează zonele comune
 - iar procesele sunt conștiente de caracterul partajat al acestora

Excludere mutuală (1)

- accesul la o resursă comună poate dura
 - și poate consta în mai multe operații
- apare pericolul interferențelor
- exemplu
 - un proces începe accesul la o variabilă comună
 - înainte de a termina, alt proces începe să o acceseze
 - variabila poate fi modificată în mod incorect

Excludere mutuală (2)

- accesul la o resursă comună - doar în anumite condiții
- excludere mutuală
 - la un moment dat, un singur proces poate accesa o anumită resursă
- mecanisme de control
 - semafor - cel mai simplu
 - structuri partajate ale căror metode de acces asigură excluderea mutuală (ex. monitor)

Implementare

- accesul la o resursă poate fi controlat (și blocat) doar de către sistemul de operare
- deci orice formă de accesare a unei resurse partajate implică un apel sistem
- dacă se lucrează la nivel jos (variabile partajate + semafoare), este sarcina programatorului să se asigure că apelul este realizat corect

Fire de execuție - comunicare

- în cazul firelor de execuție ale aceleiași proces, variabilele globale sunt automat partajate
 - viteză mai mare
 - crește riscul erorilor de programare
- necesitatea excluderii mutuale este prezentă și aici

V.5.5. Utilizarea MMU

Hardware

Cazul Intel

- segmentarea
 - nu poate fi dezactivată
 - dar poate fi "evitată" prin software
- paginarea
 - poate fi activată/dezactivată

Sistemul de operare

Cazurile Windows, Linux

- segmentarea
 - nu este utilizată în practică
 - toate segmentele sunt dimensionate astfel încât să acopere singure întreaga memorie
- paginarea
 - pagini de 4 KB
 - Windows poate folosi și pagini de 4 MB

Utilitatea MMU (1)

Avantaje

- protecție la erori
- o aplicație nu poate perturba funcționarea alteia
- verificările se fac în hardware
 - mecanism sigur
 - viteză mai mare

Utilitatea MMU (2)

Dezavantaje

- gestiune complicată
- memorie ocupată cu structurile de date proprii
 - tabelul de descriptori
 - tabelul de paginare
- viteză redusă - dublează numărul acceselor la memorie (sau mai mult)

Utilitatea MMU (3)

Concluzii

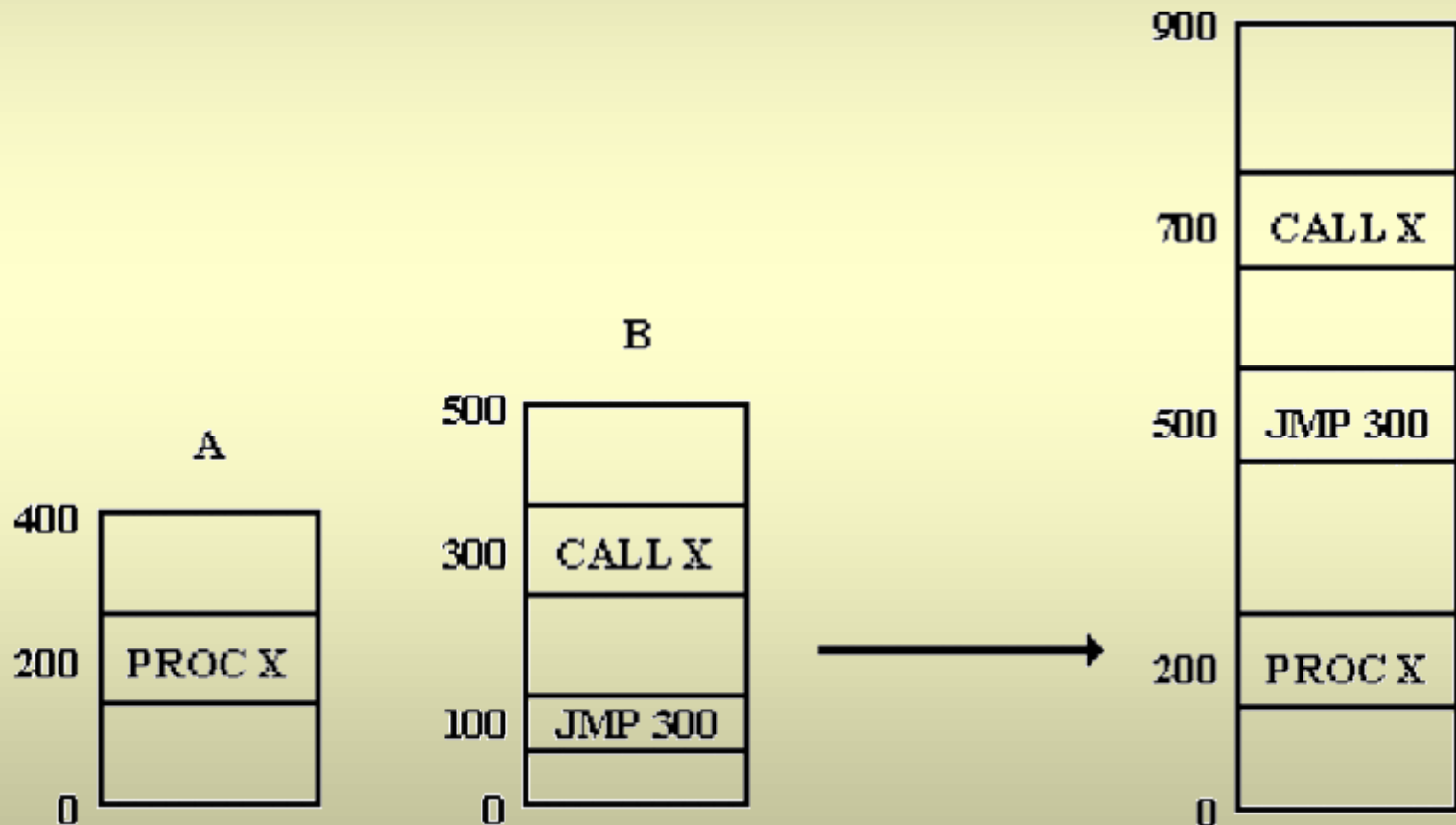
- scăderea de performanță poate fi compensată folosind cache-uri
- procesoarele de azi oferă suficientă viteză
- sisteme multitasking - risc mare de interferențe
- mecanismele MMU trebuie folosite

V.6. Crearea și execuția programelor

Crearea unui program - faze

- compilarea
 - traducerea comenzilor scrise într-un limbaj sursă în instrucțiuni pentru procesor
- editarea legăturilor (*linking*)
 - tratează aspecte privitoare la gestiunea memoriei într-un program

Crearea unui fișier executabil din mai multe module sursă



Problema relocării

- instrucțiunea de salt - adresa de salt nu mai este corectă
- module compilate independent - fiecare presupune că începe la adresa 0
- afectează și instrucțiunile care accesează date (adrese de memorie)
- adresele sunt relocate (deplasate) față de momentul compilării

Problema referințelor externe

- funcția X - apelată din alt modul decât cel în care este definită
- la momentul compilării
 - se știe că este definită în alt modul
 - este imposibil de determinat la ce adresă se va găsi funcția în programul final

Crearea programelor

- se poate scrie un program dintr-un singur modul?
- nu întotdeauna
- programe foarte complexe - modularitate
- biblioteci de funcții - module separate
 - precompilate
 - codul sursă nu este disponibil

Fazele creării unui program

- compilarea modulelor
 - fișier sursă → fișier obiect
 - fișierele obiect conțin informații necesare în faza editării de legături
- editarea legăturilor
 - fișiere obiect → fișier executabil
 - se folosesc informațiile din fișierele obiect

Structura unui fișier obiect (1)

1. antetul

- informații de identificare
- informații despre celelalte părți ale fișierului

2. tabela punctelor de intrare

- conține numele simbolurilor (variabile și funcții) din modulul curent care pot fi apelate din alte module

Structura unui fișier obiect (2)

3. tabela referințelor externe

- conține numele simbolurilor definite în alte module, dar utilizate în modulul curent

4. codul propriu-zis

- rezultat din compilare
- singura parte care va apărea în fișierul executabil

Structura unui fișier obiect (3)

5. dicționarul de relocare

- conține informații despre localizarea instrucțiunilor din partea de cod care necesită modificarea adreselor cu care lucrează
- forme de memorare
 - hartă de biți
 - listă înlănțuită

Editorul de legături (1)

1. construiește o tabelă cu toate modulele obiect și dimensiunile acestora
2. pe baza acestei tabele atribuie adrese de start modulelor obiect
 - adresa de start a unui modul = suma dimensiunilor modulelor anterioare

Editorul de legături (2)

3. determină instrucțiunile care realizează accese la memorie și adună la fiecare adresă o constantă de relocare
 - egală cu adresa de start a modulului din care face parte
4. determină instrucțiunile care apelează funcții sau date din alte module și inserează adresele corespunzătoare

Execuția programelor

- la ce adresă începe programul când este încărcat în memorie?
- nu se știe la momentul când este creat
- toate adresele din program depind de adresa de început
- concluzie: problema relocării apare din nou la lansarea programului în execuție

Soluția 1

- Fișierul executabil conține informații de relocare
 - aceste informații sunt utilizate de sistemul de operare la încărcarea programului în memorie
 - pentru a actualiza referințele la memorie
 - exemplu: sistemul de operare DOS

Soluția 2

- Utilizarea unui registru de relocare
 - încărcat întotdeauna cu valoarea adresei de început a programului curent
 - acces la memorie - la adresa precizată prin instrucțiune se adună valoarea din registrul de relocare
 - dependentă de hardware
 - nu toate procesoarele au registru de relocare

Soluția 3

- Programele conțin numai referiri la memorie relative la contorul program
 - program independent de poziție
 - poate fi încărcat în memorie la orice adresă
 - foarte greu de scris
 - instrucțiuni de salt relative - cu restricții
 - instrucțiuni care lucrează cu adrese de date relative la contorul program - nu există

Soluția 4

- Paginarea memoriei
 - programul poate fi mutat oriunde în memoria fizică
 - programul crede că începe de la adresa 0, chiar dacă nu este așa
 - dependentă de suportul hardware (mecanismul de paginare)

Biblioteci partajate (1)

Legare dinamică

- proceduri și variabile care nu sunt incluse permanent în program
 - numai atunci când este nevoie de ele
- proceduri și variabile partajate de mai multe programe

Biblioteci partajate (2)

Utilitatea legării dinamice

- proceduri care tratează situații excepționale
 - rar apelate
 - ar ocupa inutil memoria
- proceduri folosite de multe programe
 - o singură copie pe disc
 - o singură instanță încărcată în memorie

Biblioteci partajate (3)

Tipuri de legare dinamică

- implicită
- explicită

Legare implicită

- folosește biblioteci de import
 - legate static în fișierul executabil
 - indică bibliotecile partajate necesare programului
- la lansarea programului
 - sistemul de operare verifică bibliotecile de import
 - încarcă în memorie bibliotecile partajate care lipsesc

Legare explicită (1)

- programul face un apel sistem specific
- cere legarea unei anumite biblioteci partajate
- dacă bibliotecă nu există deja în memorie, este încărcată
- legătura cu o bibliotecă partajată poate fi realizată sau distrusă în orice moment

Legare explicită (2)

- exemplu - Windows

```
//legare explicită a unui modul  
hLib=LoadLibrary("module");  
//se obține un pointer la o funcție  
fAddr=GetProcAddress(hLib,"func");  
(fAddr)(2,3,8); //apel funcție  
FreeLibrary(hLib); //eliberare modul  
(fAddr)(2,3,8); //eroare, funcția nu  
mai este disponibilă
```

Legare explicită (3)

- **exemplu - Linux**

```
//legare explicită a unui modul  
hLib=dlopen("module",RTLD_LAZY);  
//se obține un pointer la o funcție  
fAddr=dlsym(hLib,"func");  
(fAddr)(2,3,8); //apel funcție  
dlclose(hLib); //eliberare modul  
(fAddr)(2,3,8); //eroare, funcția nu  
mai este disponibilă
```