

Sisteme de Operare



ș.l.dr.ing. Cristian Amarandei
camarand@cs.tuiasi.ro

Curs #01 - Conținut

- Prezentarea disciplinei
- Noțiuni introductive

Prezentarea disciplinei

Organizare

❑ Curs

- 2 ore/săptămână [Total: 28 ore curs]

❑ Laborator

- 2 ore/săptămână [Total: 28 ore laborator]

Prezentarea disciplinei

Organizare [2]

❑ Evaluare

■ Evaluarea finală

- ❑ ponderea în nota finală: **50%**
- ❑ Probă scrisă (**notă ≥ 5**)
 - Test grilă
 - 1 sau 2 probleme
 - fără acces la surse bibliografice

■ Evaluarea pe parcurs

- ❑ Activitatea la laborator (**notă ≥ 5**)
 - ponderea în nota finală: **20%**
 - Criterii de evaluare: rezolvarea temelor propuse și calitatea soluțiilor.
- ❑ Teste laborator (săptămâna 6 și 14) (**notă ≥ 5**)
 - ponderea în nota finală: **30%**
 - Probă practică
 - Acces la lucrările de laborator

Prezentarea disciplinei

□ Obiectivele cursului

- Introducerea conceptelor de bază ale sistemelor de operare: procese, thread-uri, sincronizare, concurență, fișiere, memorie;
- Analiza sistemelor de operare moderne (Linux, Solaris, Windows)
- Prezentarea unor algoritmi/tehnici de proiectare a aplicațiilor complexe
- Înțelegerea modului în care alegerea diferitelor strategii de proiectare și implementare a unui sistem de operare au implicații asupra utilizării resurselor calculatorului și asupra programelor utilizator

□ Rezultatele învățării

- Înțelegerea mecanismelor fundamentale care stau la baza funcționării unui sistem de calcul (hardware -> sistem de operare -> aplicații)
- Cunoștințe despre algoritmi/tehnici de proiectare a aplicațiilor complexe
- Dezvoltarea unor aplicații care utilizează apeluri sistem
- Utilizarea/administrarea mai eficientă a sistemelor de calcul

De ce studiem Sistemele de Operare?

▫ Pentru a învăța cum funcționează sistemele de calcul

- Arhitectura hardware/software

▫ Administrarea/utilizarea eficientă a unui sistem de calcul

- CPU, memorie, sisteme de fișiere, etc.

▫ Asigurarea performanței sistemului/aplicațiilor

- Este necesară modificarea parametrilor SO – nu se poate face fără a înțelege funcționarea acestuia.
- Înțelegerea serviciilor oferite de SO influențează modul de proiectare a aplicațiilor complexe
- Erori de funcționare ale SO -> sistem de calcul inutilizabil.
- Trebuie să aibă mai puține erori de funcționare decât aplicațiile

▫ Depanarea aplicațiilor

▫ System/low-level/kernel programming

▫ Anumite aspecte/tehnici pot fi aplicate în alte domenii

- concurență, gestiunea resurselor, tratarea erorilor, gestiunea de structuri complexe (nivelul de abstractizare furnizat de SO)

▫ Securitatea SO -> baza securității sistemului de calcul

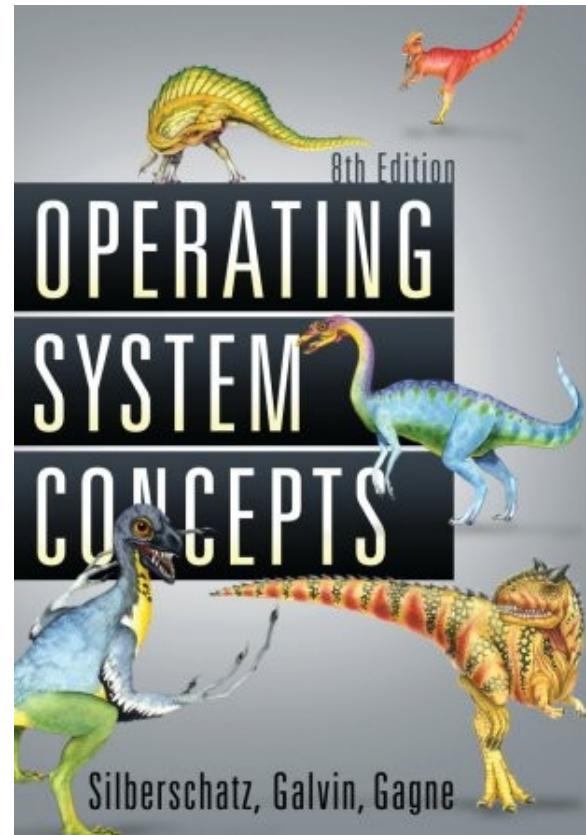
▫ Deoarece sunt peste tot ! (PC, servere, smartphones)

Cuprins

- ❑ Noțiuni introductive
- ❑ Structura unui sistem de operare
- ❑ Gestiunea proceselor
- ❑ Gestiunea memoriei
- ❑ Gestiunea sistemului de fișiere
- ❑ Sistemul de I/O

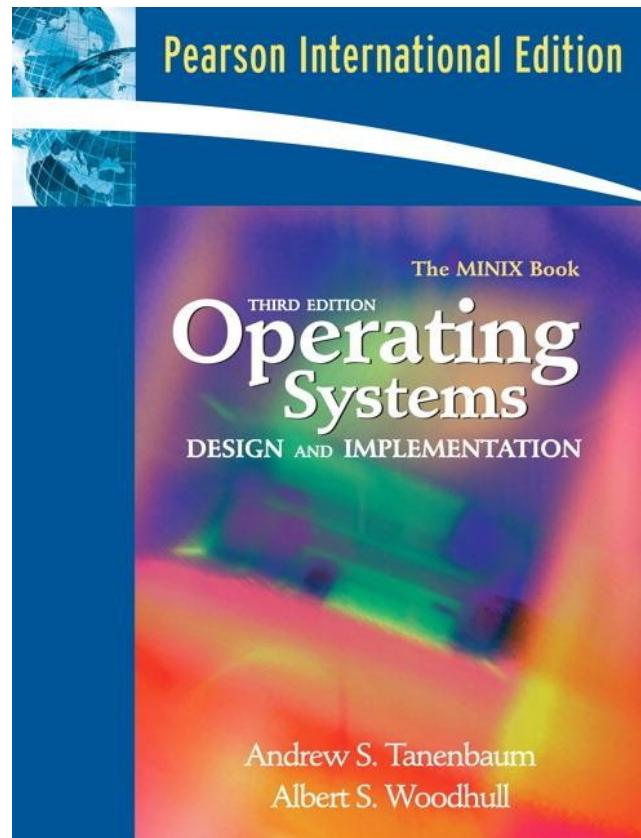
Bibliografie

- Silberschatz A.,
Galvin P. -
**Operating System
Concepts**, 9th
Edition , John Wiley
& Sons, 2012
- [http://www.os-
book.com/](http://www.os-book.com/)



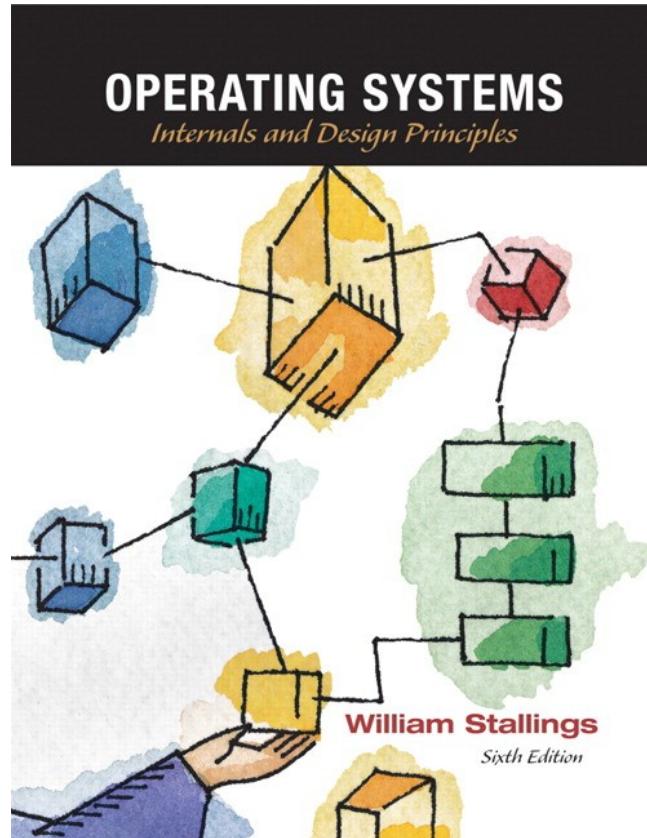
Bibliografie

- Andrew S. Tanenbaum -
Operating Systems: Design and Implementation, 3rd Edition, Prentice Hall, 2009
 - <https://github.com/citrusle/e/Studijne-materialy-FIIT/blob/master/3.%20Semeester/OS/Materialy/Operating%20Systems%20Design%20%26%20Implementation%203rd%20Edition.pdf>
 - <http://www.minix3.org/documentation/index.html>



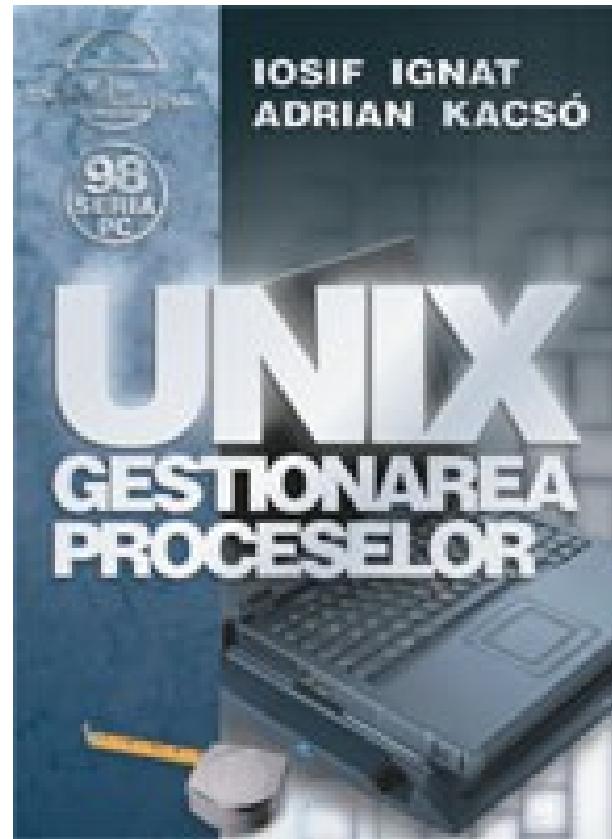
Bibliografie

- William Stallings -
**Operating Systems:
Internals and
Design Principles,**
6th Edition, Prentice
Hall, 2008
- [http://williamstallings
.com/OS/OS6e.html](http://williamstallings.com/OS/OS6e.html)



Bibliografie

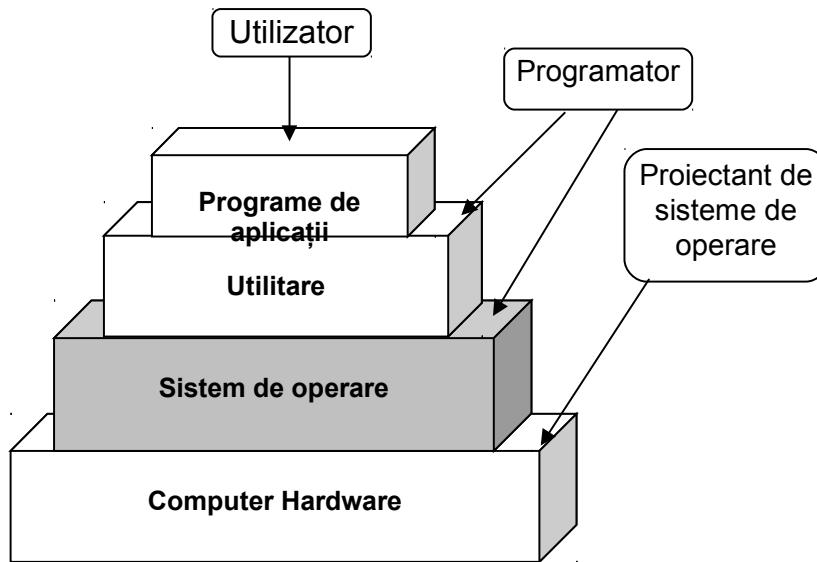
- David Solomon, Mark Russinovich - **Microsoft Windows Internals**, 4th/5th/6th Ed., 2004/2009/2012
- Boian F.M. - **Sisteme de operare interactive**, Ed. Libris, 1994
- T. Ionescu, D. Saru, J. Floroiu - **Sisteme de operare. Principii și funcționare**, Ed. Tehnică, 1997
- Ignat I. - **Unix - Gestionarea Proceselor**, Ed. Albastră, 2006



Definiția unui sistem de operare

- Un sistem de operare reprezintă un set de programe care asigură gestionarea resurselor unui sistem de calcul implementând algoritmi destinați să maximizeze performanțele și realizează o interfață între utilizator și sistemul de calcul, extinzând dar și simplificând setul de operații disponibile.

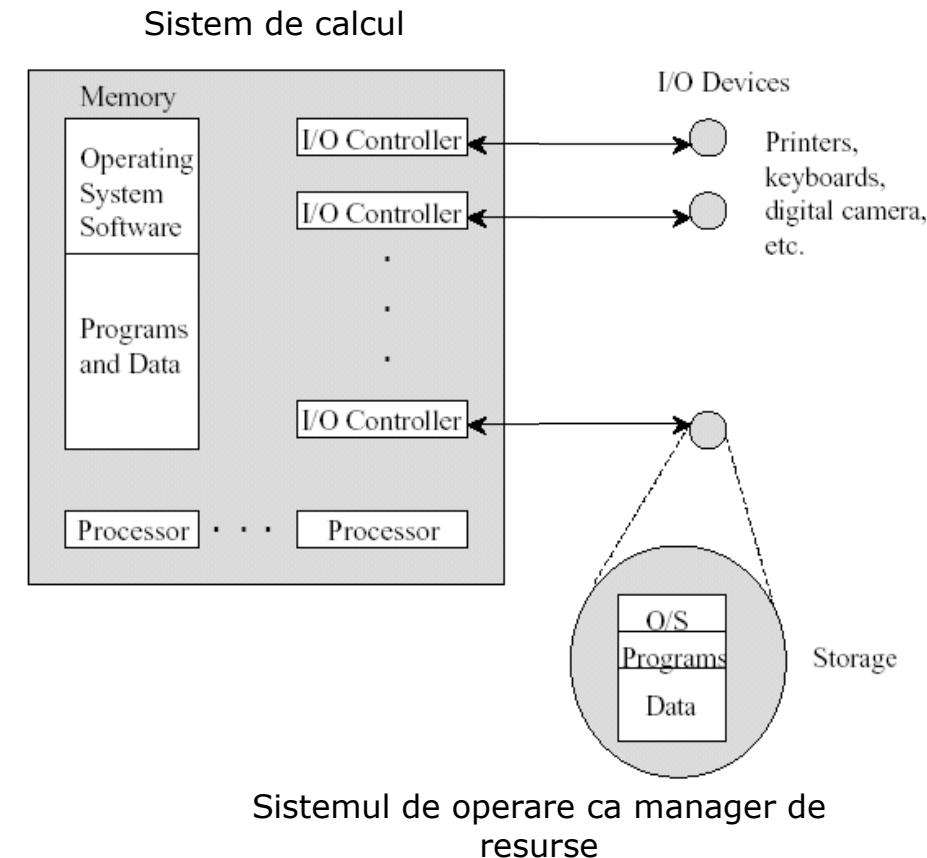
Structura unui sistem de calcul



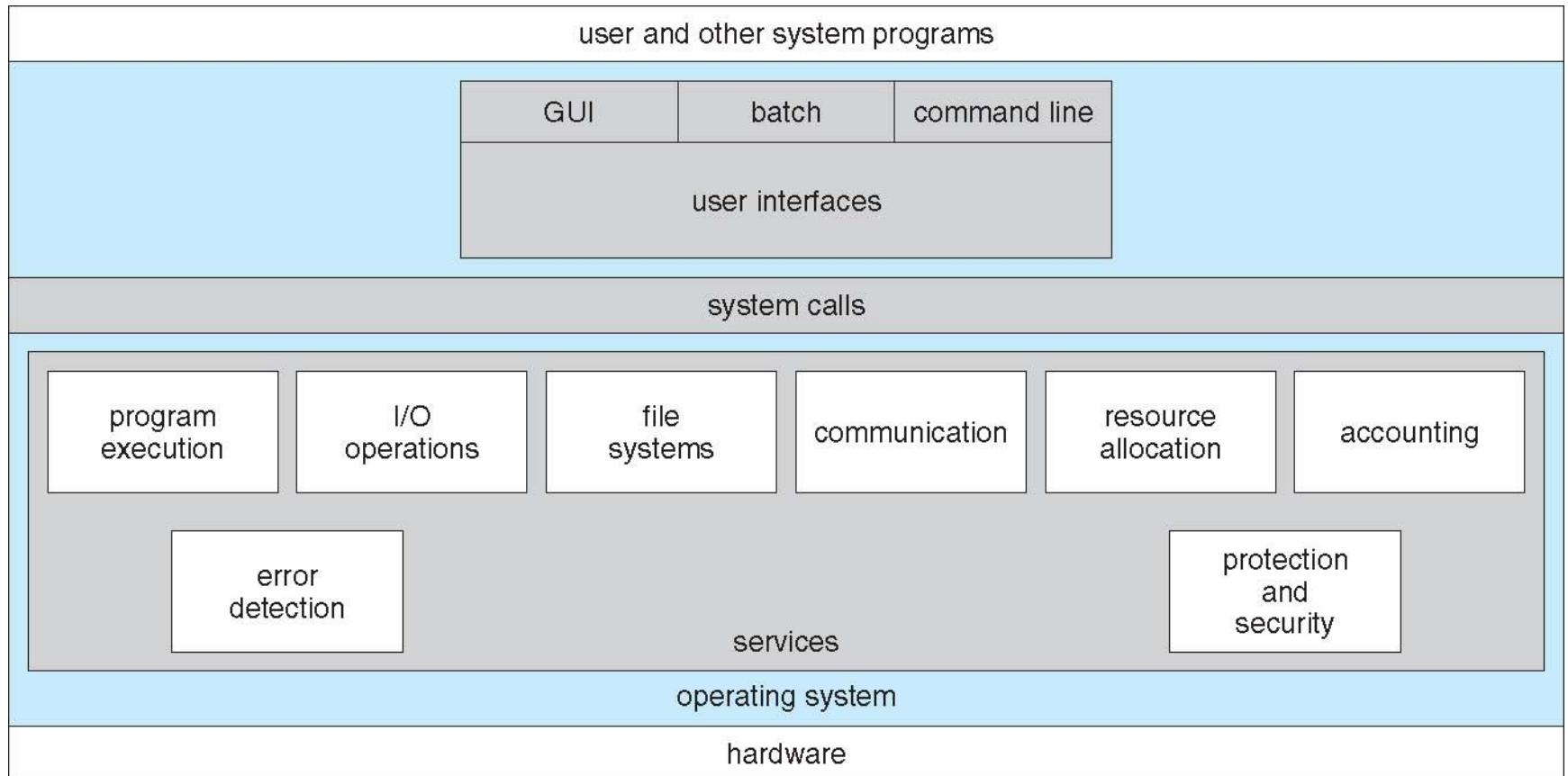
- **Hardware :**
 - CPU, memorie, dispozitive I/O
- **Sistem de operare:**
 - Controlează și coordonează utilizarea resurselor hardware
- **Programe utilitare**
 - folosite pentru administrare și operarea sistemului de calcul
- **Aplicații:**
 - procesoare de text, compilatoare, web browser, baze de date, jocuri
- **Utilizatori**

Functiile unui sistem de operare

- Interfața cu utilizatorul
- Gestiunea fișierelor
- Gestiunea perifericelor
- Gestiunea memoriei
- Gestiunea proceselor
- Tratarea erorilor
- Gestiunea sistemului



Structura unui sistem de operare



Interfața cu utilizatorul

□ Comenzile

- determină într-o mare măsură acceptarea de către utilizatori a unui sistem de operare
- introduse de utilizatori
- sunt prelucrate de interpretorul de comenzi

□ Apelurile de sistem

- definesc arhitectura unui sistem de operare
- asemănătoare apelurilor de proceduri
- sunt folosite de programatorii de sistem și programatorii de aplicații

Gestiunea fișierelor

- Fișierele reprezintă forma în care sunt păstrate informațiile
- gestiunea fișierelor - modul în cadrul sistemului de operare:
 - operațiile de creare și stergere a fișierelor
 - controlul accesului la fișiere
 - citirea și scrierea de informații din și în fișiere
 - securitatea informațiilor
 - organizarea colecției de fișiere

Gestiunea perifericelor

- pregătirea operației
- lansarea cererilor de transfer de informație
- controlul transferului propriu-zis
- tratarea erorilor

Gestiunea memoriei

- ❑ o porțiune este rezervată pentru sistemul de operare
- ❑ restul memoriei este disponibilă pentru programele utilizator
- ❑ Trebuie să rezolve probleme legate de:
 - protecția memoriei folosite de programe
 - împărțirea memoriei disponibile între programele solicitante
 - planificarea schimburilor cu memoria

Gestiunea proceselor

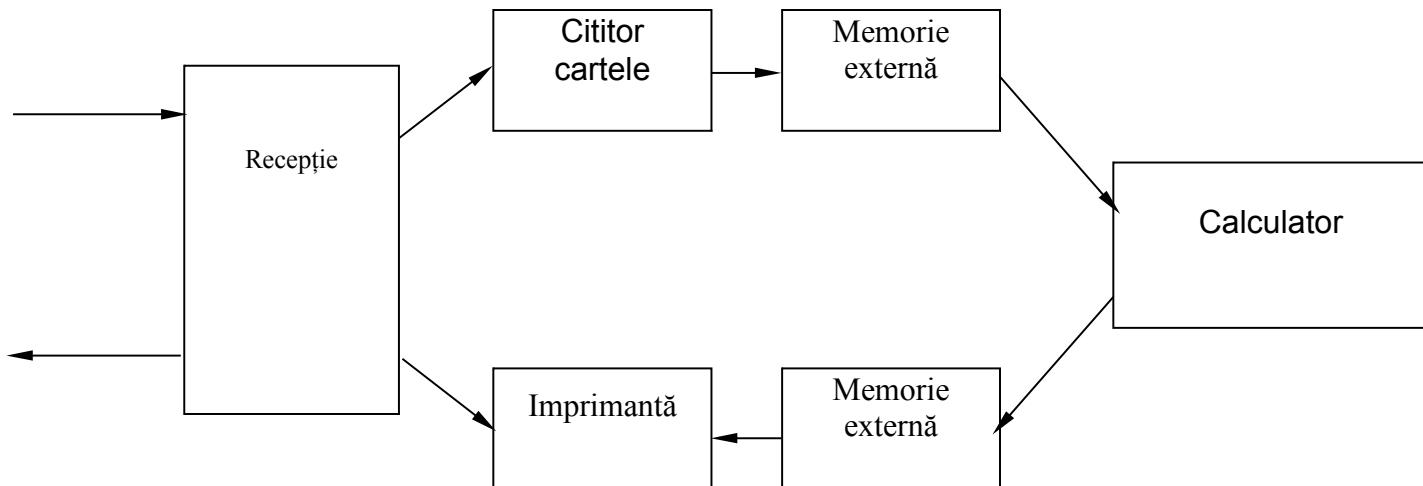
- Trebuie să rezolve probleme legate de:
 - Stările unui proces
 - Contextul unui proces
 - Sincronizarea proceselor
 - Tratarea blocajelor
 - Planificarea proceselor

Tratarea erorilor

- sistemul de operare trebuie să reacționeze la o diversitate de erori ce pot apărea atât din cauze hardware cât și software
- erorile trebuie să fie transparente utilizatorului
- SO trebuie să aibă mai puține erori de funcționare decât aplicațiile

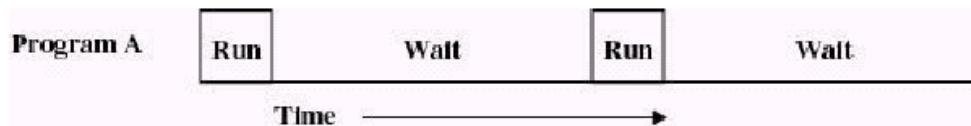
Evoluția sistemelor de operare

- a) **Sistemele de tip Batch** (SB) sau cu prelucrare pe loturi – deservește job-uri dintr-o coadă de job-uri.
 - Dezavantaje:
 - lipsa interacțiunii cu utilizatorul în timpul rulării programului
 - timpul de rulare foarte mare

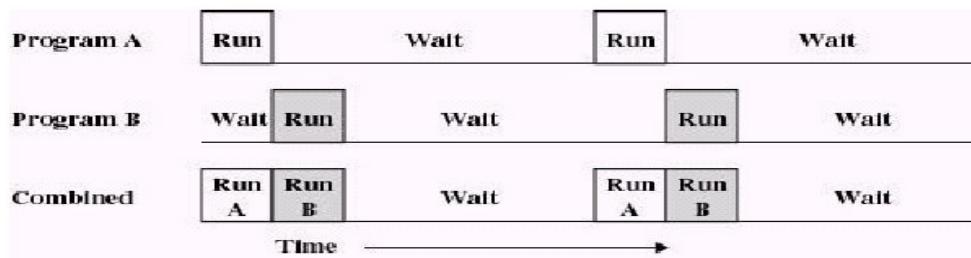


Evoluția sistemelor de operare (2)

- sistemele de tip Batch
 - cu monoprogramare

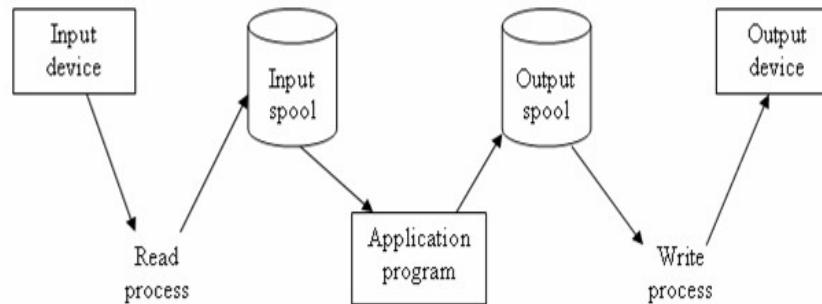


- cu multiprogramare

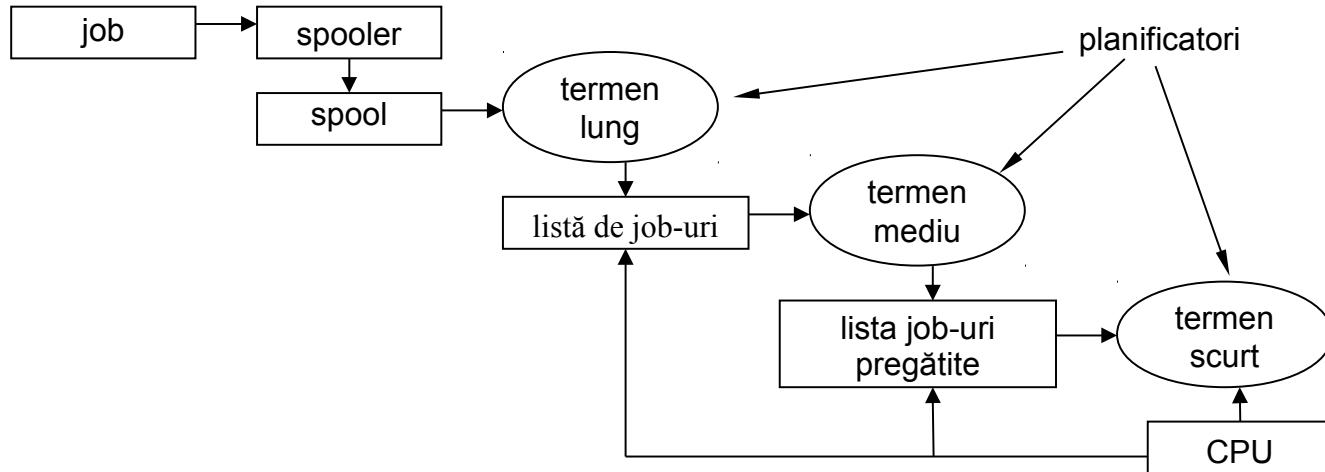


Evoluția sistemelor de operare (3)

- ❑ Sistemele cu multiprogramare au introdus tehnica job spooling
 - SPOOL = Simultaneous Peripheral Operations On-Line.
 - ❑ disponibilitatea unei memorii externe de capacitate mare și cu acces direct
 - ❑ asigurarea unei încărcări la întreaga capacitate a dispozitivelor periferice de intrare/ieșire



Evoluția sistemelor de operare (4)



Evoluția sistemelor de operare (5)

□ b) Sisteme cu divizarea (partajarea) timpului (time sharing) – SPT

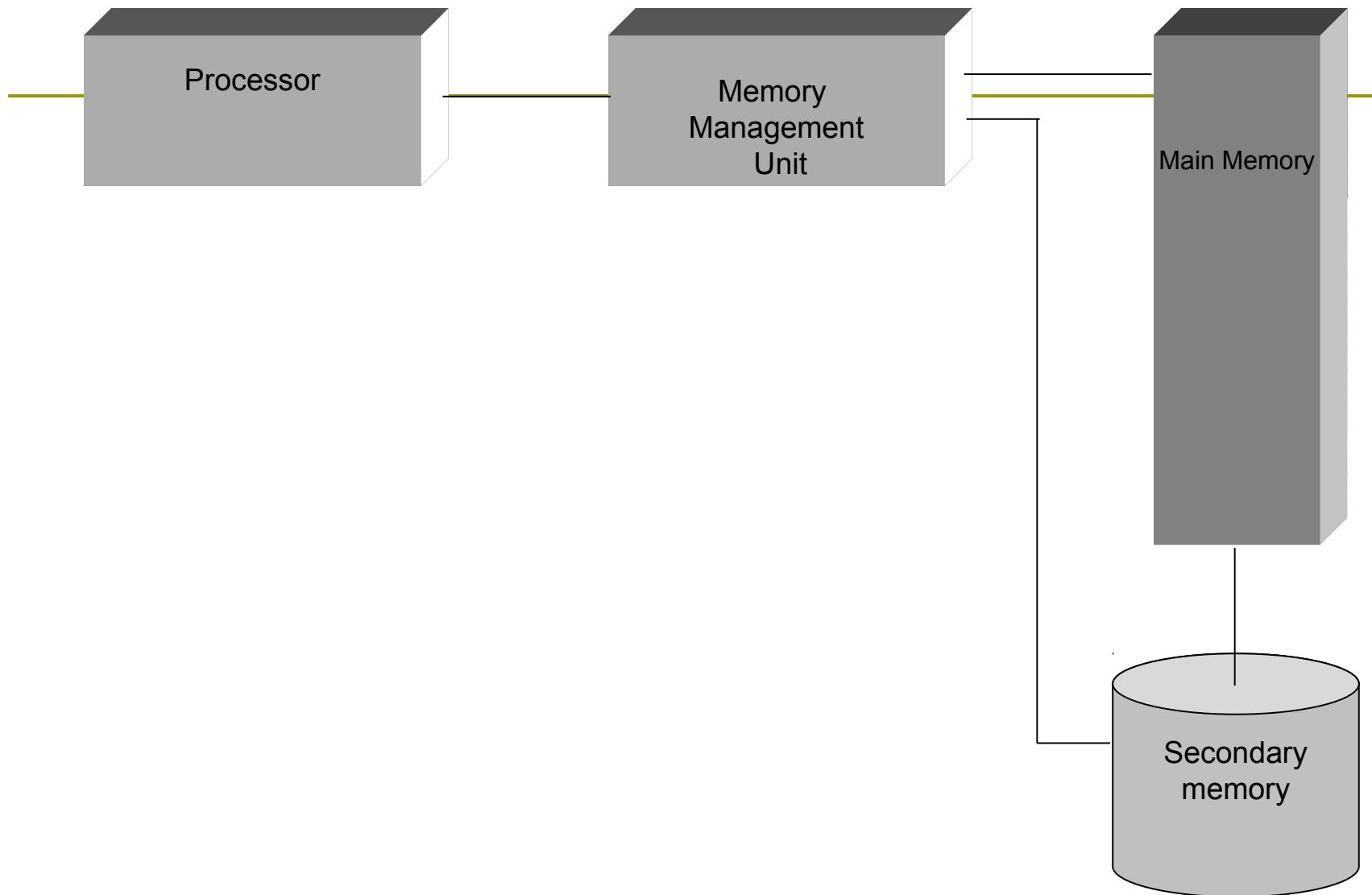
- posibilitatea ca mai mulți utilizatori să partajeze o mașină prin întrețeserea job-urilor
- nu permiteau interacțiune în timp real între utilizator și sistemul de calcul
- asigură câte un terminal pentru fiecare utilizator al sistemului și din acest motiv se mai numesc sisteme multitasking și multiuser
- se introduce un mecanism de protecție și securitatea accesului la resurse.
- Exemple:
 - CTSS (Compatible Time-Sharing System) dezvoltat la MIT pe un sistem IBM 7094
 - TSS/360 dezvoltat de IBM pentru IBM System 360/67
 - MULTICS , dezvoltat de MIT, AT&T Bell Laboratories și General Electric pe un calculator GE 635 modificat

Evoluția sistemelor de operare (6)

□ c) Sisteme cu memorie virtuală

- Prin tehnica de memorie virtuală un disc se poate folosi ca suport automat (fără intervenția programatorului) pentru extinderea memoriei interne, permitând programelor să aibă la dispoziție un spațiu de memorie virtual mai mare decât cel al memoriei interne.
- Exemple:
 - OS/MVS (Multiple Virtual Spaces) ;
 - VM/370 ("Virtual Machine for the 370") - realizează virtualizarea unui întreg sistem nu numai a memoriei.

Sisteme cu memorie virtuală



Evoluția sistemelor de operare (7)

□ d) Sisteme de operare pentru minicalculatoare

- PDP -11(anii '70-80):
 - RT-11 - gândit să deservească un singur utilizator, dar prevăzut cu o formă limitată de multiprogramare (foreground / background)
 - RSX-11 - un limbaj de comandă puternic, un sistem de fișiere dezvoltat, gestionarea memoriei interne prin partiții de dimensiune variabilă și multiprogramare.
- PDP-7 - Unix :
 - - dezvoltat la Bell Laboratories de Thompson și Ritchie, reimplementat și extins pe PDP-11
 - - Inspirat din MULTICS, permite accesul simultan al mai multor utilizatori, fiecare putând lansa procese concurente

Evoluția sistemelor de operare (8)

□ e) Sisteme de operare pentru calculatoare personale

- 1971 - microprocesor Intel 4004, apoi apar 8008, 8080, urmate de Z80 și Motorola M6800.
- CP/M:
 - primul sistem de operare pe 8 biți
 - produs de Digital Research
 - sistem monoutilizator cu interfață de comandă simplă, sistem de fișiere mononivel, interfață I/E bine dezvoltată
- micropresocarele pe 16 biți (Intel 8086, Motorola 68000)
 - MS-DOS, Windows 3.1, 3.11
 - monoutilizator, monoproces, sistem ierarhic de fișiere, interfață cu utilizatorul bine dezvoltată.

□ f) Sisteme de operare pentru sisteme de calcul distribuite

- au apărut odată cu dezvoltarea rețelelor de calculatoare
- se urmărește integrarea cât mai puternică a resurselor din rețea astfel încât localizarea resurselor să fie transparentă pentru utilizatori și aplicații (CHORUS, MACH, AMOEBA).

Clasificarea sistemelor de operare

- Sistemul de operare poate fi privit ca o mașină abstractă ce are mai multe niveluri:
 - A0 – nivelul hardware
 - A1 – nivelul microprogram (funcții BIOS)
 - A2 – nivelul funcțiilor realizate în limbaj de asamblare ce gestionează regiștrii, controller-ele, intreruperile, memoria.
 - An-1 – API interfața de apeluri sistem a S.O.

Clasificarea sistemelor de operare (2)

- După calculatoarele pe care rulează:
 - stații de lucru, în general mono-utilizator.
 - Ex: MS-DOS, WINDOWS, sistem UNIX cu facilități grafice
- pentru o rețea de calculatoare poate fi gestionată de un S.O. de tip:
 - file-sever (Netware, Unix):
 - S.O. este pe un singur calculator, stațiile având încărcate în memorie o versiune de DOS – la faza de login singura resursă comună – harddisk-ul de pe server
 - peer – to – peer :
 - Windows 3.11 (nodurile sunt echivalente) – același S.O. este încărcat de fiecare calculator ;

Clasificarea sistemelor de operare (3)

□ După accesul la memorie:

- **UMA (Uniform Memory Access)**: adică timpii de acces la memorie sunt egali pentru toate procesele din sistem (memorie globală)
- **NUMA (Non Uniform Memory Access)**: timpi de acces diferiți
- **NORMA (No Remote Memory Access)**: sistemul este cuplat într-o rețea fără memorie globală.

□ Din punctul de vedere al transmisiei de mesaje:

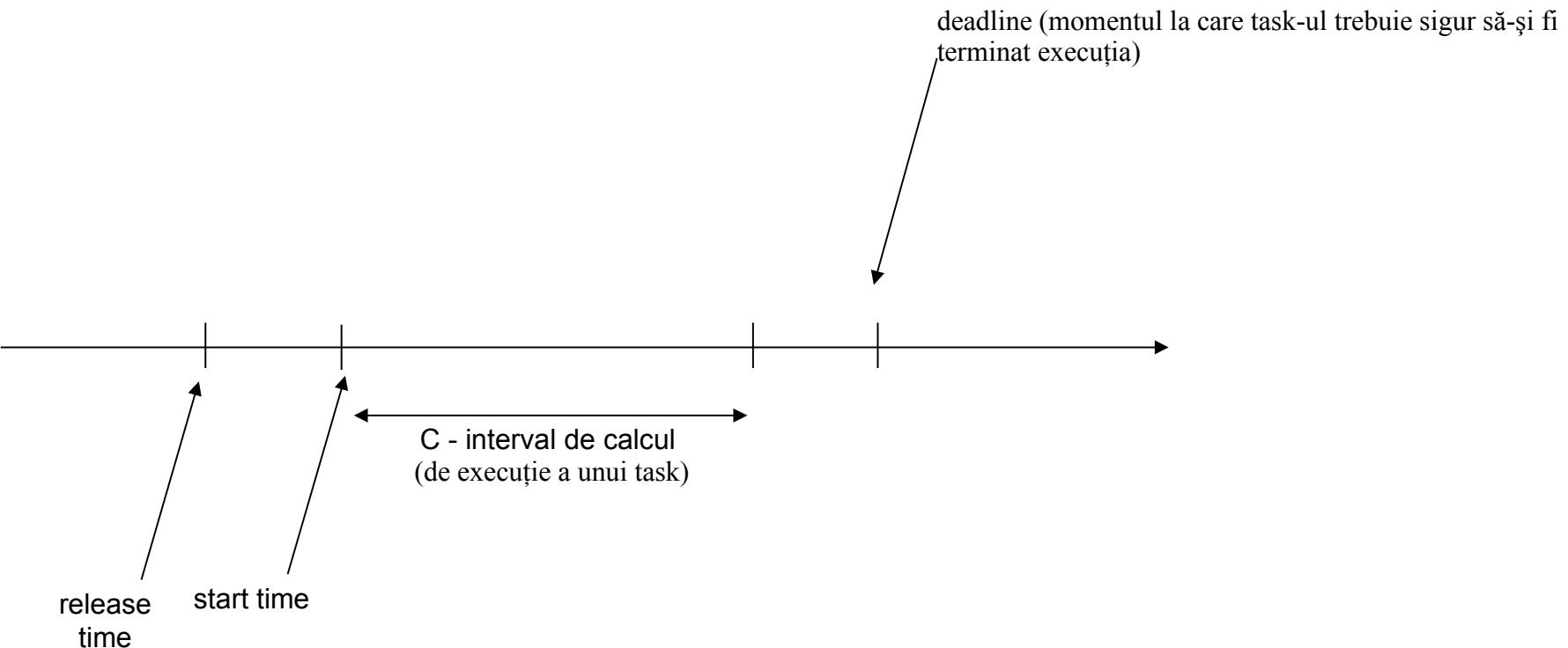
- **Sisteme de operare cu transmitere de mesaje** – permite realizarea unui S.O. distribuit (comunicare între procese atât de pe aceeași mașină cât și de pe mașini diferite).
- **Sisteme de operare fără transmitere de mesaje** – apelul procedurilor pe care le executa procesele utilizator se transmit prin apeluri de proceduri locale.

Clasificarea sistemelor de operare (4)

- ❑ Sistemele de operare de timp real
 - Trebuie să opereze în cadrul unor constrângeri de timp
- ❑ Clasificare:
 - După comportarea în timp real al task-urilor:
 - task-uri sporadice: task-ul este asincron și nepredictibil (ex: intreruperile)
 - task-uri aperiodice: are un anumit ciclu dar cu perioade de timp variabile (nu respectă o distribuție)
 - task-uri periodice: au o ciclicitate regulată (reîmprospătarea imaginii)
 - După importanța deadline-ului
 - hard - real time: acțiunea unui task ce depășește deadline-ul este 0 sau negativă
 - soft - real time: dacă acțiunea unui task depășește deadline-ul are o valoare indiferentă
- ❑ Exemple: QNX, CHORUS

Clasificarea sistemelor de operare (5)

□ Sistemele de operare de timp real

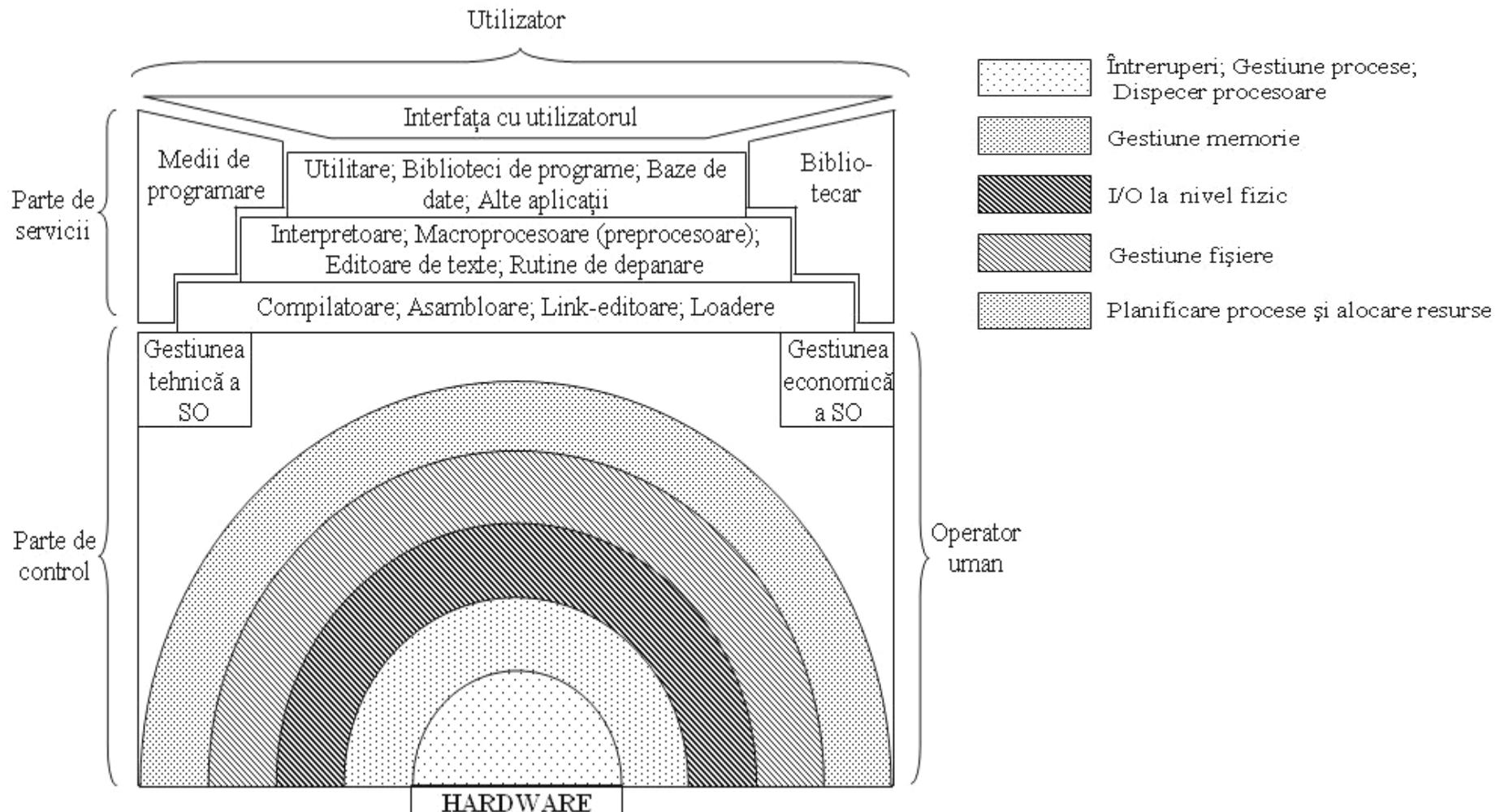


Sisteme de Operare



- Structura unui sistem de operare
- Organizarea memoriei
- Procese

Structura unui sistem de operare



Structura unui sistem de operare (2)

□ partea de control:

- se execută în mod nucleu și realizează legătura cu sistemul de calcul;
- intreruperi
- gestiune procese
- dispecer procesoare
- gestiune memorie
- I/O la nivel fizic
- gestiune fișiere
- planificare procese și alocare resurse
- gestiune tehnică a SO
- gestiune economică a SO

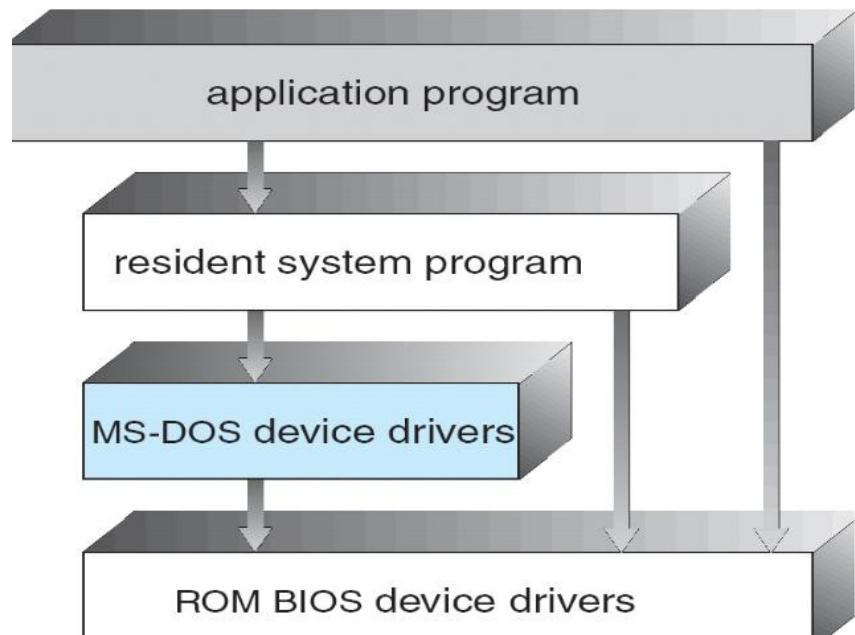
Structura unui sistem de operare (3)

□ partea de servicii:

- se execută în mod utilizator, folosind facilitățile părții de control și asigură legătura cu utilizatorul;
- compilatoarele
- asamblorul
- link-editorul sau editorul de legături
- loader: încarcarea programelor
- interpreter comenzi
- macroprocesor (preprocesor)
- editorul de texte
- rutine de depanare
- bibliotecarul
- mediile de programare
- interfața cu utilizatorul

Exemplu: MS-DOS

- Nu este modular
- Nivelurile nu sunt foarte bine separate



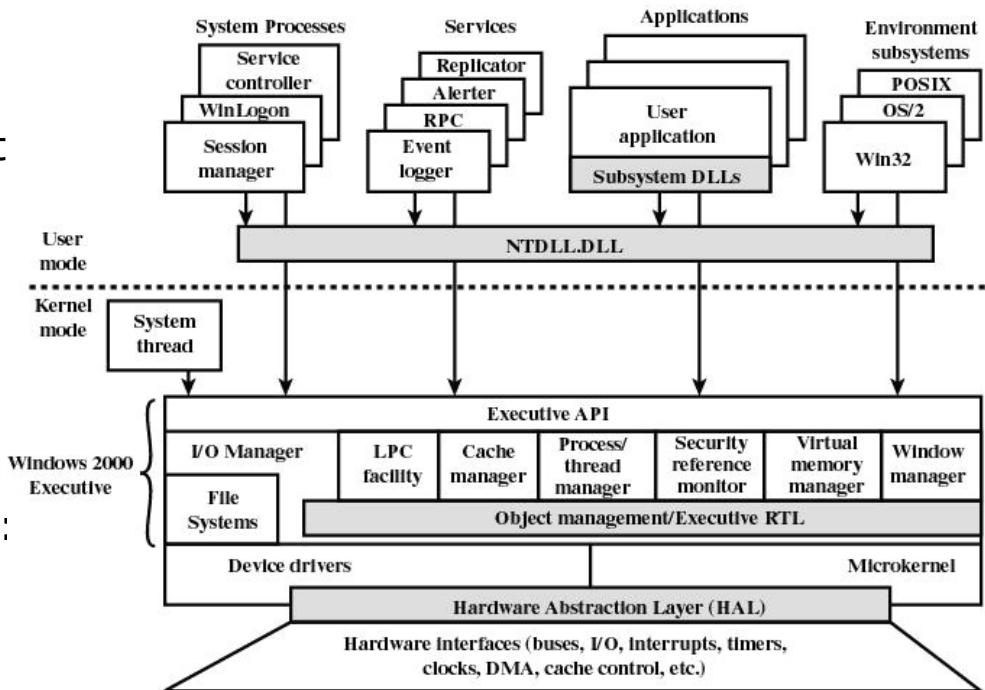
Exemplu: Windows 2000

□ microkernel modificat:

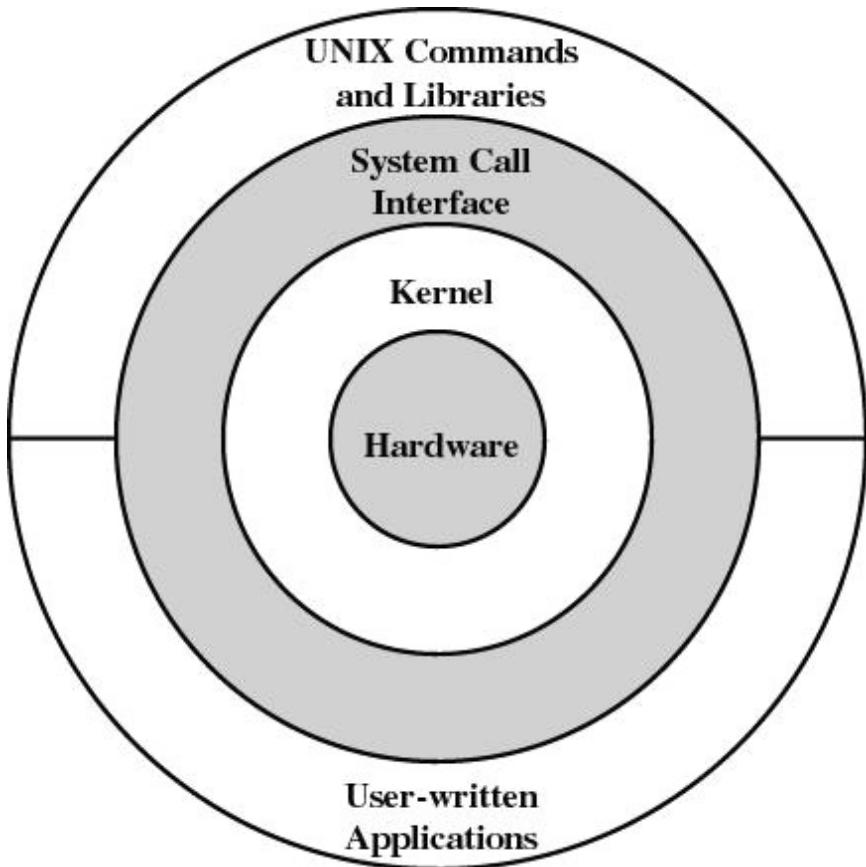
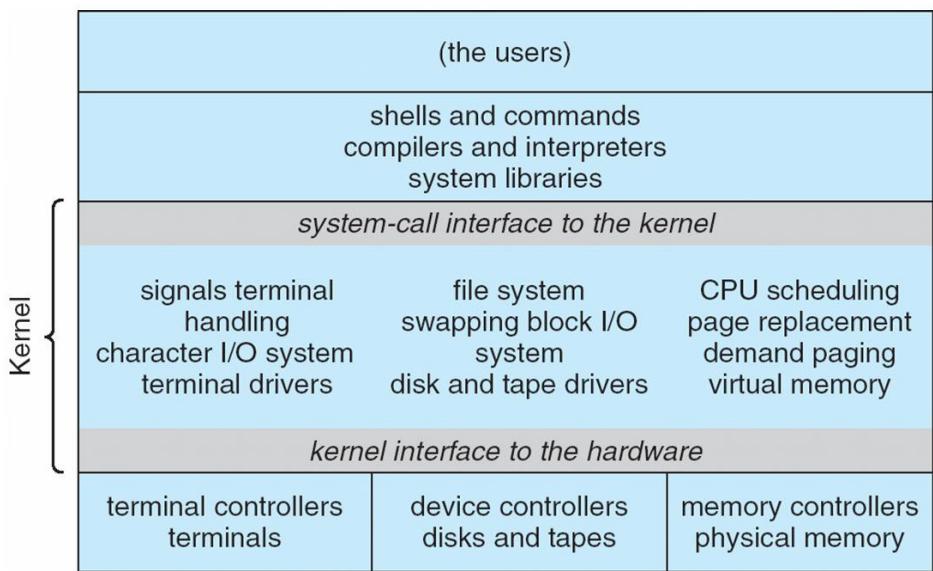
- nu are un microkernel pur;
- multe din funcțiile sistemului sunt în afara microkernel-ului și rulează în modul kernel;
- modulele pot fi șterse, înnoite, înlocuite fără a fi necesară rescrierea întregului sistem

□ Structura pe niveluri (layere):

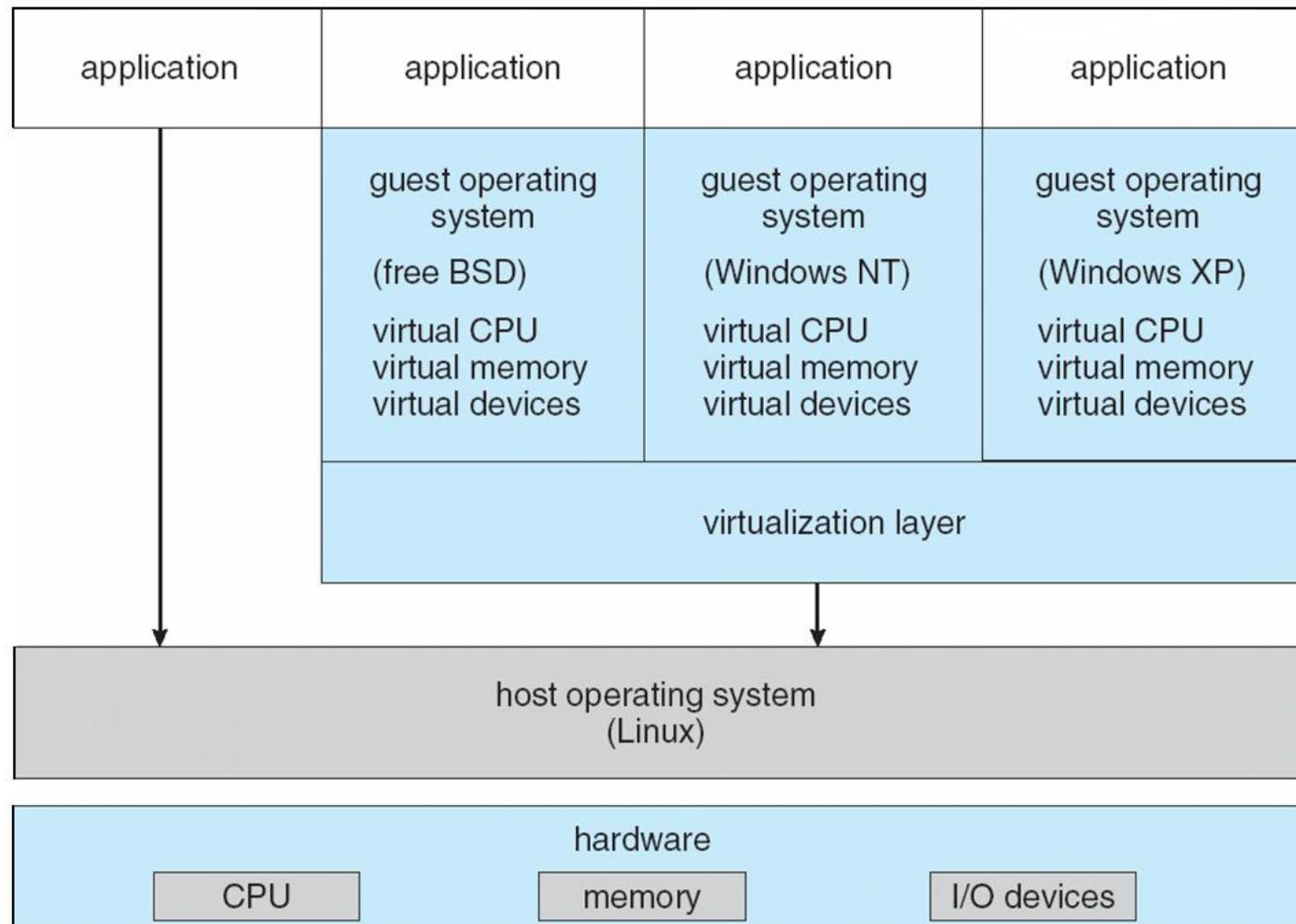
- Hardware abstraction layer (HAL): izolează sistemul de operare de specificul platformei hardware;
- Microkernel
- Drivere



Exemplu: UNIX



Exemplu: VMware



Nucleul (kernel) unui sistem de operare

- partea rezidentă a unui SO
- conține proceduri care tratează:
 - planificarea proceselor
 - tratarea erorilor
 - verificarea securității
 - tratarea inițială a apelurilor sistem
- ocupă o zonă fixă a memoriei
- include regiunea cu adresele cele mai mici, regiune în care se găsesc vectorii de intrerupere.

Nucleul (kernel) unui sistem de operare(2)

- ❑ Componentele mai puțin utilizate (componentele tranzitorii)
 - componente pot avea o zonă de memorie rezervată
 - pot fi încărcate în orice zonă de memorie disponibilă
 - memory mapped I/O

memory mapped I/O

- o porțiune a spațiului de adrese este rezervată pentru I/O și sunt mapate peste un set de registre.
- aceste adrese fac referință la registrii din interfețele hardware cu echipamentele periferice
- prin operațiile de citire/scriere de la aceste adrese se transferă date spre și de la echipamentele periferice.

Kernel – variante de implementare

▫ nucleu monolit:

- implică implementarea funcțiilor legate de gestionarea proceselor, gestionarea fișierelor, I/O și a memoriei într-un singur modul care va conține toate funcțiile aferente lor.

▫ nucleu modular:

- asigură funcționalitatea de bază la o clasă de module distincte (potrivit cu împărțirea funcțiilor), interacțiune cu ele facându-se prin apel de proceduri sau prin comunicație interproces.

▫ nucleu extensibil:

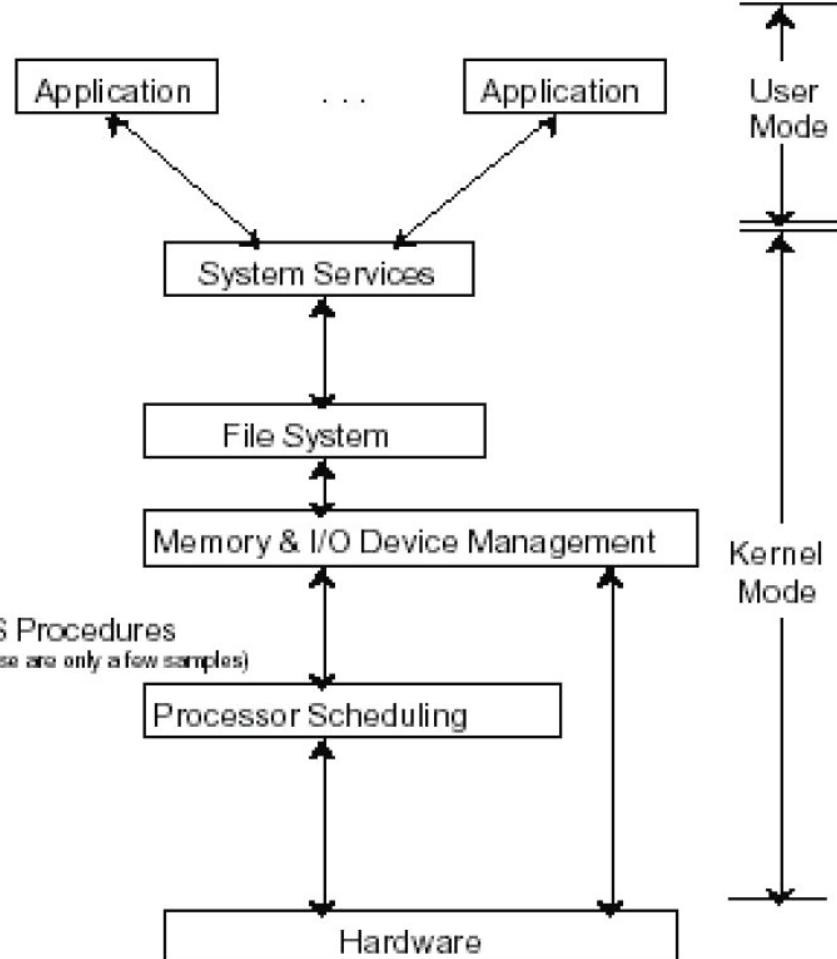
- este o combinație între nucleul monolit și cel modular pentru a realiza un schelet de nucleu (mașina abstractă) ce se poate îmbunătăți conform cerintelor funcționale (are un număr minim de funcții la care se pot adăuga altele).

▫ nucleu multnivel:

- legat de conceptul de mașină abstractă, constă în împărțirea funcțiilor în interfețe apelabile între ele.

▫ micronucleu (microkernel):

- sistemul de operare este alcătuit din mai multe procese, fiecare asigurând anumite servicii.



Organizarea memoriei

- hărțile de memorie (**memory maps**)
 - conținutul **spațiului virtual de adrese** când este în execuție un program utilizator;
 - conținutul spațiului virtual de adrese când este în execuție o componentă a sistemului de operare;
 - conținutul memoriei fizice;

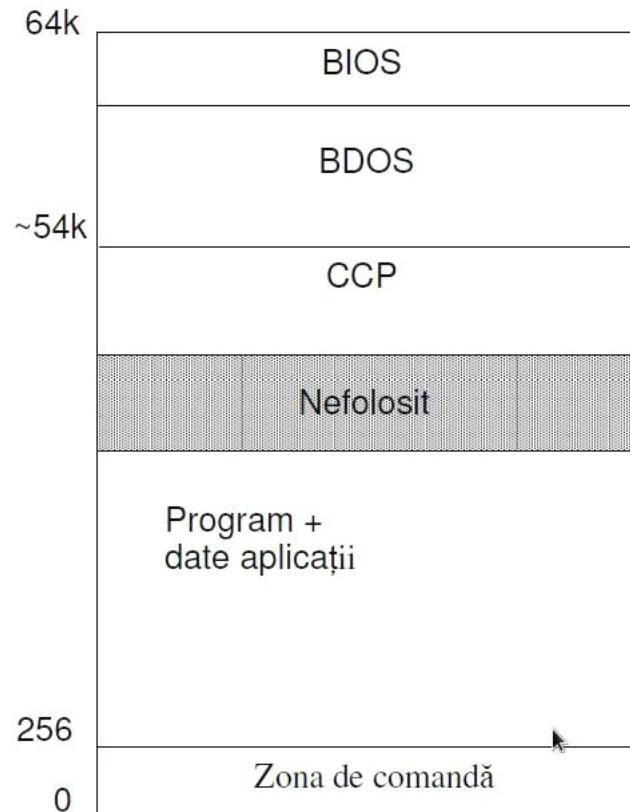
Spațiul virtual de adrese

- disproportia dintre necesarul de memorie al unui program și memoria fizică (limitată) este rezolvată de conceptul de **spațiu virtual de adrese***
 - Spațiul de adrese disponibil pentru procese este fix și declarat în SO pentru toate procesele
 - SO moderne permit utilizarea a 2^N bytes de memorie, unde N este 32 sau 64

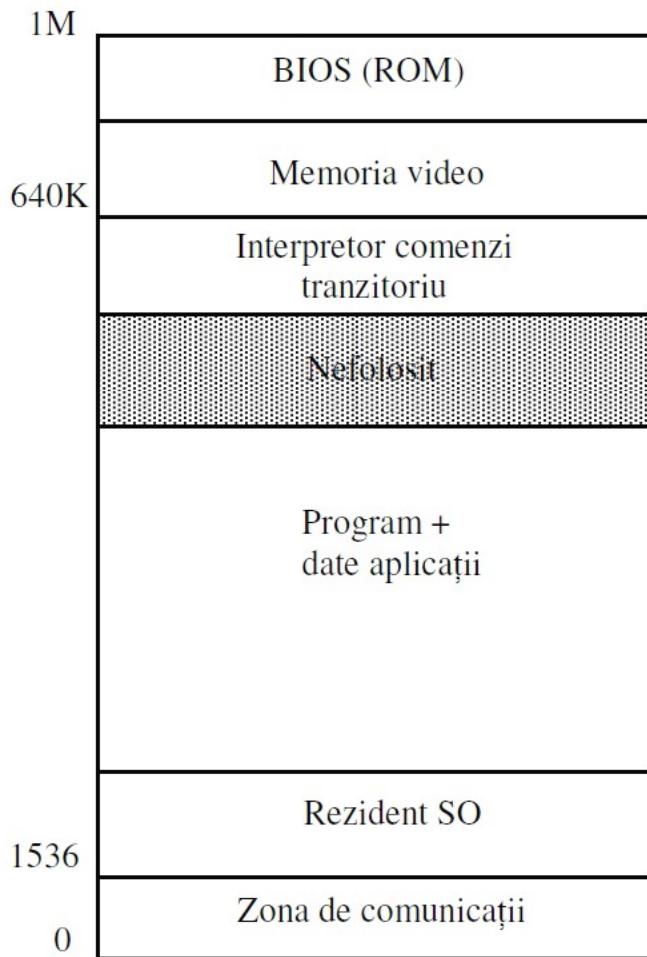
* în Cursul 1 – sisteme cu memorie virtuală

Organizarea memoriei CP/M

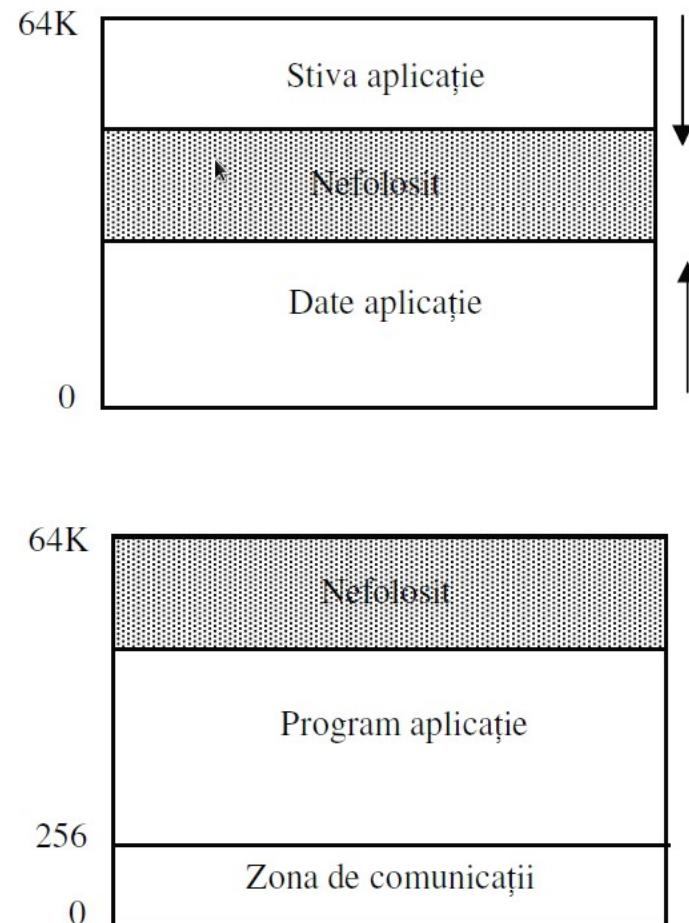
- Memoria direct adresabilă în sistem este de 64k octeți
 - limită impusă de arhitectura procesorului 8080
- primii 256 octeți
 - pentru vectorii de întrerupere și pentru informații de sistem
- 10 kocetă din zona superioară a spațiului de adrese
 - Componenetele rezidente BIOS (Basic Input / Output System) și
 - BDOS (Basic Disk Operating System)
- 2 kocetă
 - Interpretorul de comenzi CCP (Command Control Processor)



Organizarea memoriei MS-DOS



Harta memoriei la MS-DOS



Adrese virtuale într-un proces

Organizarea memoriei MS-DOS

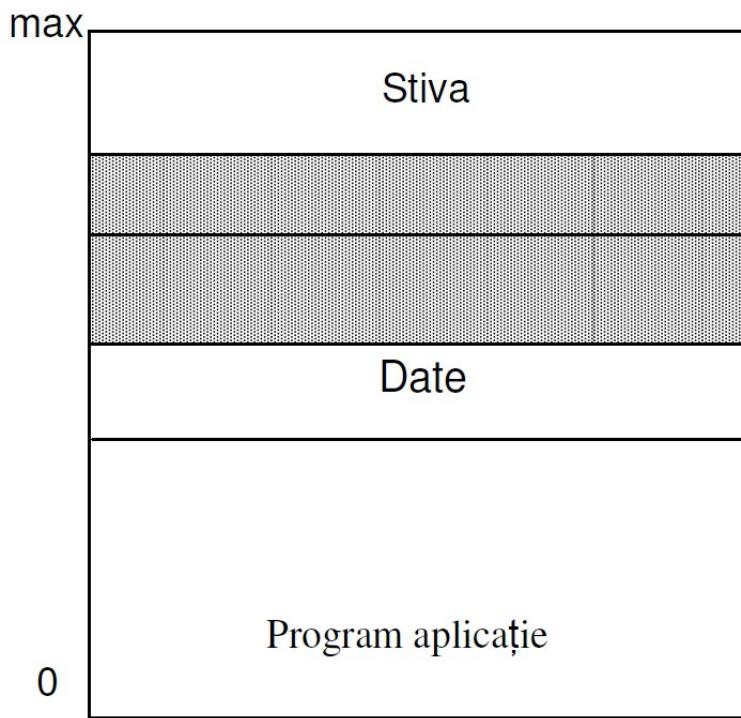
- ❑ Procesorul 8086 permite adresarea directă a 1Moctet de memorie
- ❑ rezervă zona peste 640K pentru memoria video și pentru programe memorate în ROM (o serie de programe de test activate la pornirea calculatorului)
- ❑ vectorii de întrerupere și o zonă tampon de 512 octeți ocupă primii 1536 octeți
- ❑ Partea rezidentă ocupă zona începând de la 1536 - include și o porțiune din interpretorul de comenzi
- ❑ Procesele:
 - spațiul de adrese este limitat la 4 segmente de 64Ko fiecare
 - un segment pentru cod
 - un segment pentru stivă
 - două segmente pentru date

Organizarea memoriei UNIX

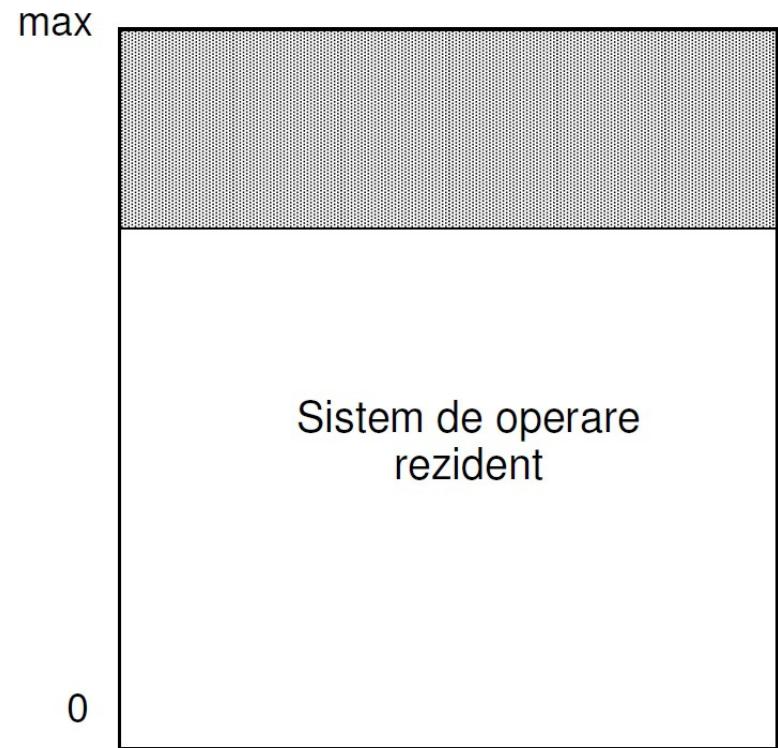
❑ Harta de memorie:

- trebuie ținut cont de caracteristicile arhitecturale ale calculatorului gazdă
- din spațiul de adrese al unui proces nu este vizibilă nici o porțiune a sistemului de operare.

Organizarea memoriei UNIX



UNIX - Adrese virtuale într-un proces

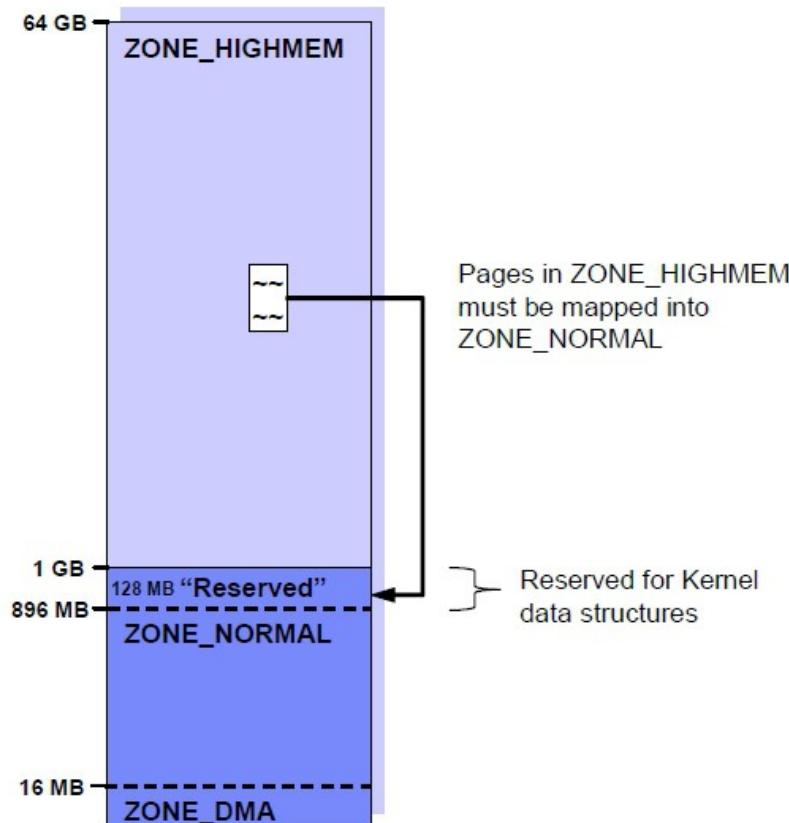


UNIX - Adrese virtuale în sistemul de operare

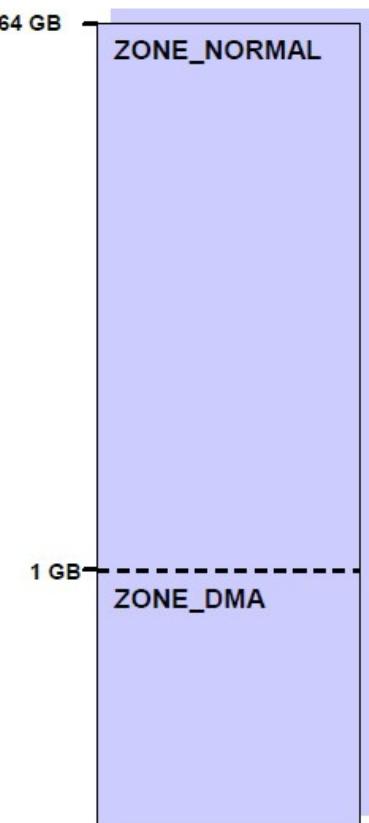
Organizarea memoriei Linux

Sursa: Linux Performance and Tuning Guidelines - <http://www.redbooks.ibm.com/abstracts/redp4285.html>

32-bit Architecture



64-bit Architecture

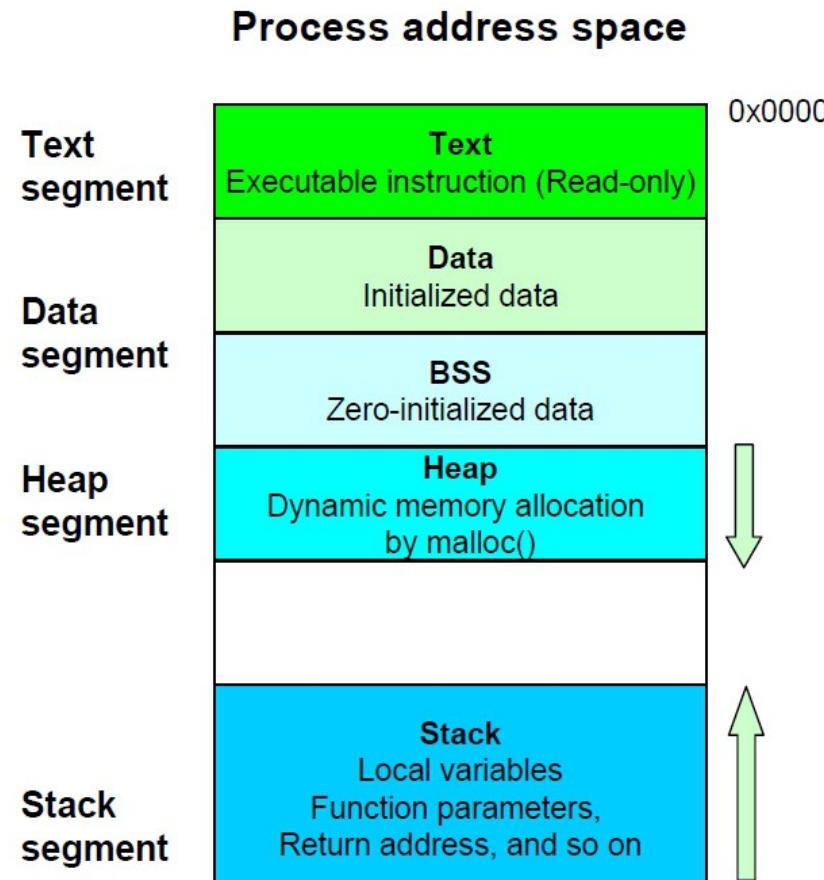


Procesoarele 386:

- folosind memoria paginată, fiecare proces poate accesa 4GB de memorie (2^{32})

Organizarea memoriei Linux

□ Memoria vazută de un proces

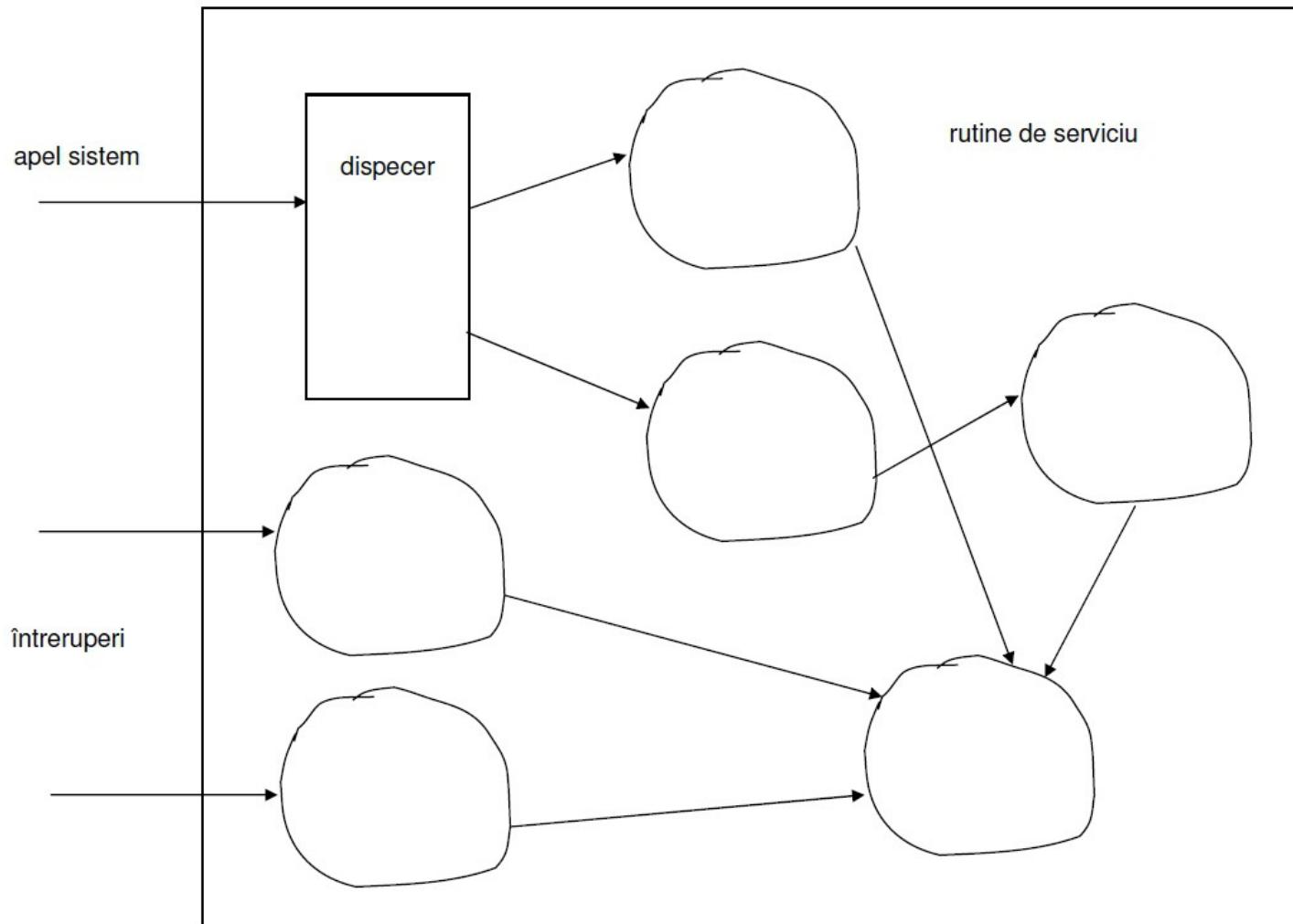


Sursa: Linux Performance and Tuning Guidelines - <http://www.redbooks.ibm.com/abstracts/redp4285.html>

Interacțiunea componentelor la execuție

- Sistemul de operare poate fi privit ca o colecție de rutine invocate prin apeluri sistem sau ca urmare a unor întreruperi
- Apelurile sistem:
 - sunt diferite de apelurile normale de rutine,
 - duc la creșterea nivelului de privilegii în timpul execuției
 - pot folosi toate instrucțiunile
 - au acces la toți registrii procesorului
 - au acces la toate locațiile de memorie.
 - au un mecanism de declanșare care este analog producerii unor întreruperi, dar sursa este în program și nu în afara acestuia
 - au același punct de intrare în sistemul de operare
 - fiecare apel este însoțit de informații suplimentare (codul apelului) transmise prin registrii procesorului
 - codul apelului este folosit de dispecer pentru a selecta rutina adecvată tratării apelului.

Declanșarea execuției rutinelor sistemului de operare



Structura pe niveluri a componentelor

- Sistemele de operare sunt structurate pe niveluri deoarece fiecare nivel poate fi dezvoltat și testat separat
- Orice procedură a unui sistem de operare ar trebui să poată apela orice altă procedură sau ar putea avea acces la orice structură de date ceea ce ar duce la o mare dificultate în determinarea efectelor schimbărilor în proceduri sau în structurile de date.
- Nu există reguli generale în repartizarea funcțiilor pe fiecare nivel
- Sarcinile critice ca planificarea proceselor, gestiunea memoriei, protecția, trebuie să apară pe un nivel mai jos decât cele ca gestiunea fișierelor și interfața cu utilizatorul.
- Numărul de niveluri nu poate fi prescris prin reguli fixe

Adaptabilitatea la configurația hardware

- ❑ este de dorit ca sistemele de operare să fie cât mai portabile
- ❑ trebuie să se poată adapta la orice configurație
 - poate duce la unele modificări în codul sursă sau doar la o specificare a unor anumiți parametri la inițializarea sistemului
- ❑ să detecteze automat caracteristicile hardware ale sistemului; să se “autoconfigureze” la încărcare

Procese

□ Definiții:

- Procesul este un program în execuție.
- Un program este o secvență de instrucțiuni.
 - Pentru un program dat (fișier executabil) pot exista unul sau mai multe procese asociate numite instanțe.
 - Procesul reprezintă invocarea dinamică a unui program împreună cu resursele necesare pentru lansarea în execuție.
 - Resursele necesare pentru rularea unui program includ: stiva utilizator, stiva sistem, memorie, identificatori de fișier, etc.

Procese (2)

- **Fiecare proces se execută într-un spațiu de adrese propriu.**
- Spațiul de adrese virtual este cuprins între 0 și adresa virtuală maximă accesibilă și este format din următoarele zone (segmente):
 - Segmentul text
 - codul programului, este protejat la scriere;
 - Segmentul de date
 - date predefinite (cunoscute la compilare) sau date alocate dinamic;
 - Segmentul de stivă
 - conține argumente, variabile locale și alte date pentru execuția funcțiilor în modul utilizator

Procese (3)

- Sistemele uniprocesor:
 - execuția unui proces este secvențială
- Sistem multitasking:
 - pe un procesor se pot executa mai multe procese, fiecare proces având alocată o cantă de timp pentru execuție după care urmează altul
 - execuția se numește secvențial concurrentă sau aparent paralelă

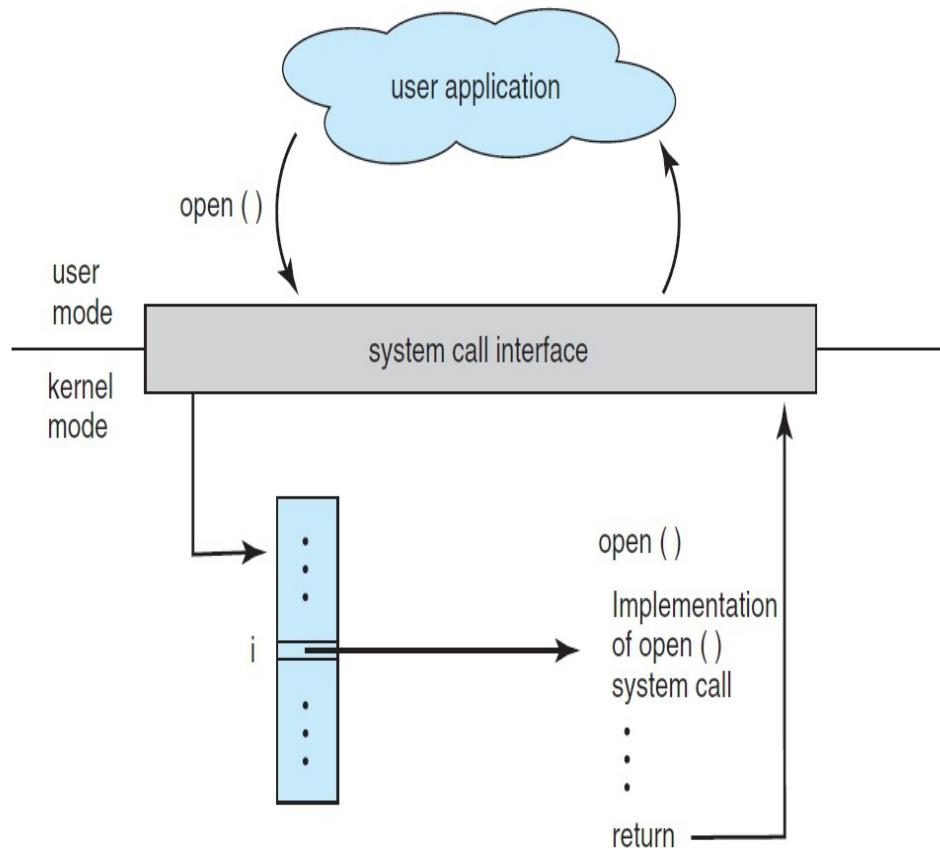
Execuția unui proces

□ Modul utilizator:

- procesele au acces numai la propria zonă de cod, date și stivă utilizator;

□ Modul nucleu:

- procesul conține instrucțiuni privilegiate și poate avea acces la structurile de date ale nucleului.



Crearea proceselor

- la inițializarea sistemului (reboot);
- la execuția unui apel de funcție pentru creare de procese (fork);
- la cererea unui utilizator de creare a unui nou proces;
- la inițializarea unui job.

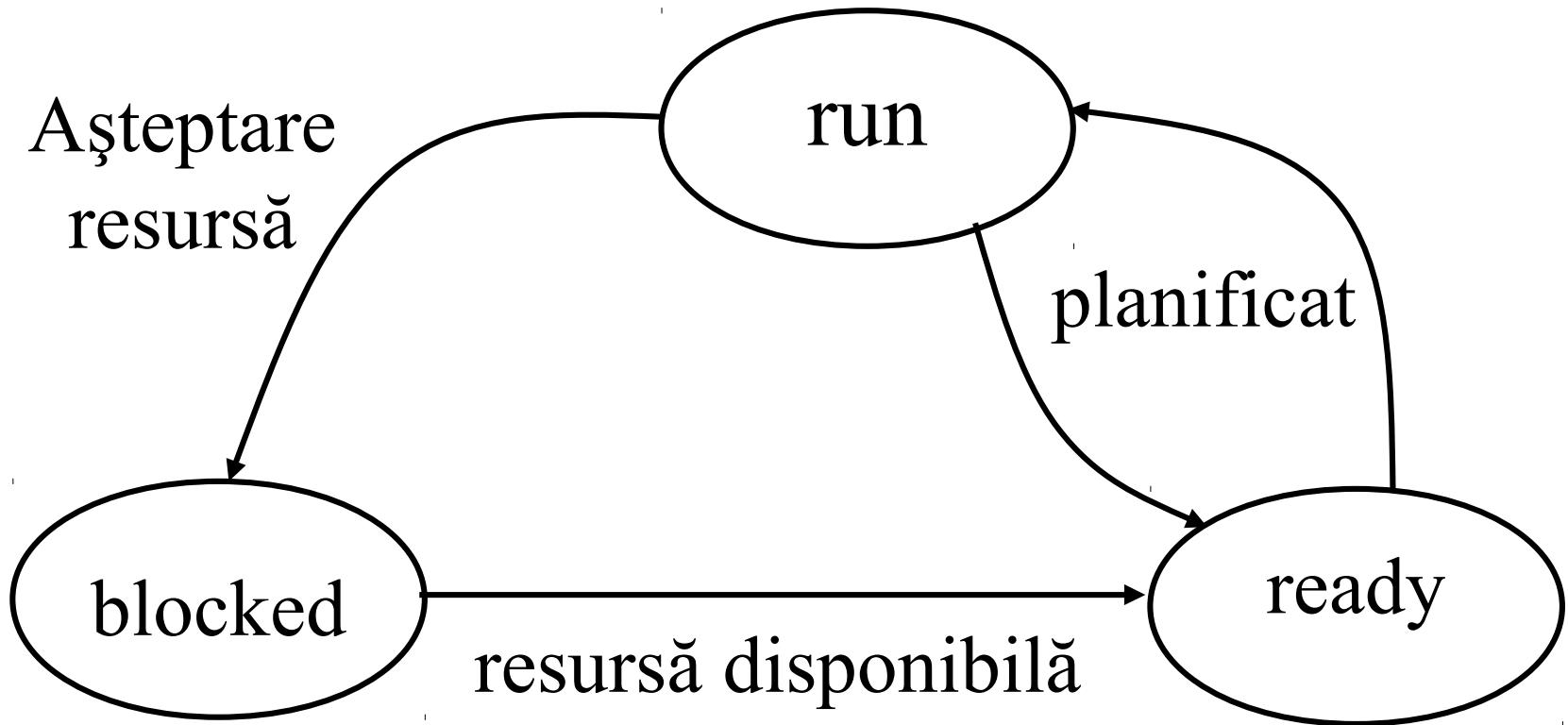
Terminarea execuției proceselor

- ❑ normal – terminarea programului (voluntar)
- ❑ eroare – terminare cu eroare (voluntar)
- ❑ fatal error – divide by 0, core dump (involuntar)
- ❑ terminat de un alt proces (involuntar)

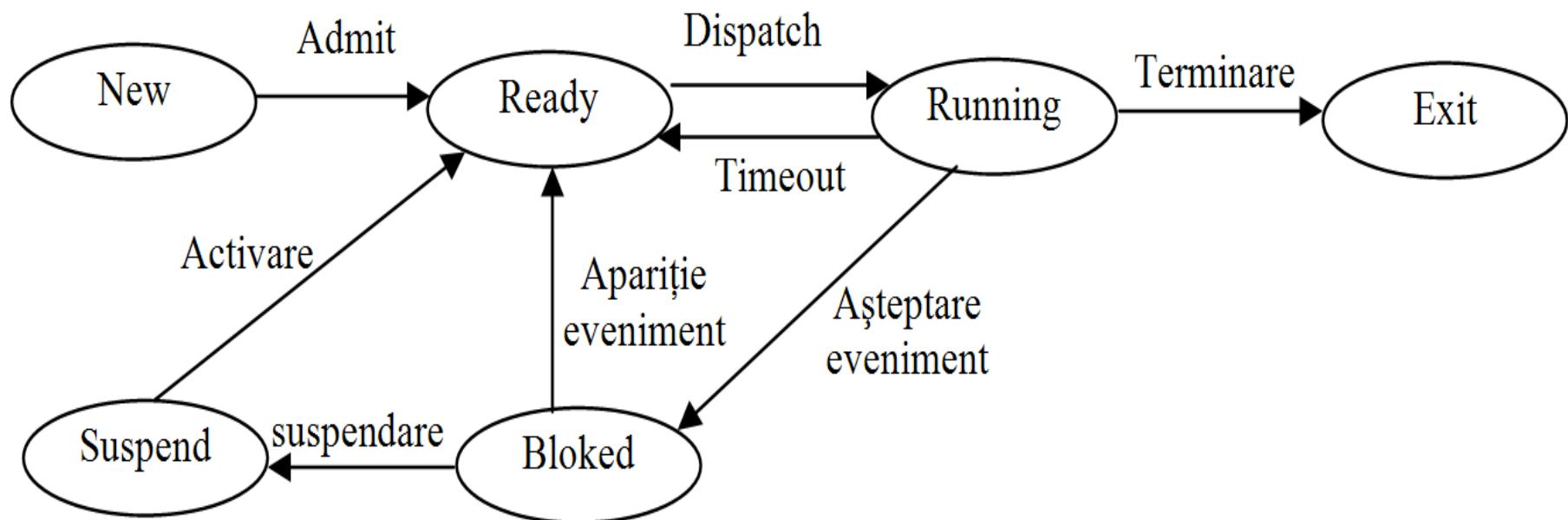
Starile unui proces

- **Starea unui proces evidențiază activitatea procesului**
 - **New** – Procesul este creat
 - **Run** – Un proces care se execută
 - **Blocat** – Dacă procesul este implicat într-o operație de I/O și dispozitivul periferic nu este liber sau pur și simplu este lent sau nu este pregătit
 - **Ready** – Dacă sistemul este multitasking, atunci un proces folosește procesorul pe durata unor cuante de timp fiind oprit de către sistemul de operare care alege alt proces pentru execuție
 - **Terminat** – Execuția procesului este terminată

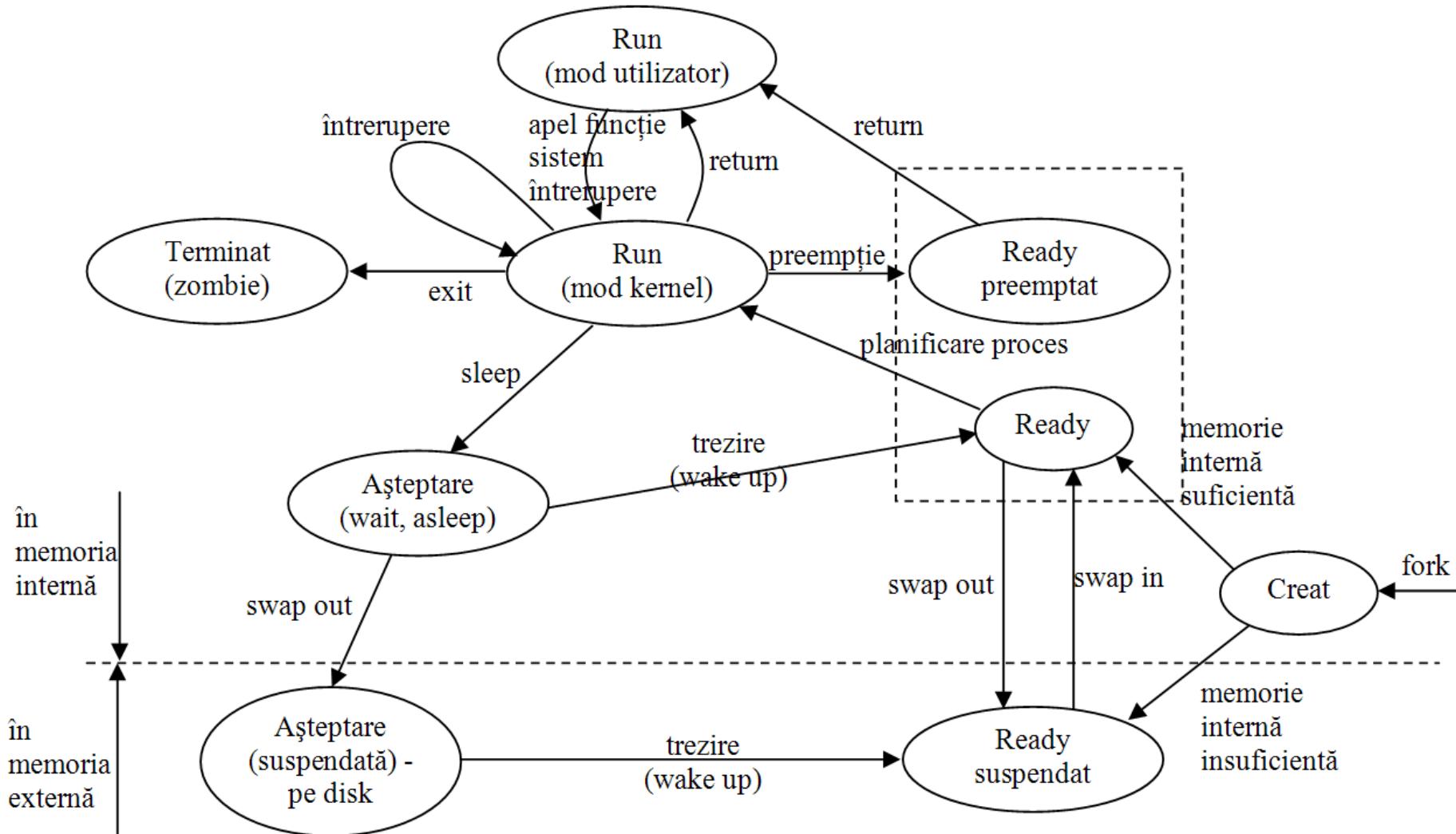
Diagrama de stare simplificată pentru un proces



Stările unui proces cu o singură stare de suspendare



Stările unui proces - UNIX



Sisteme de Operare



- Gestiunea proceselor

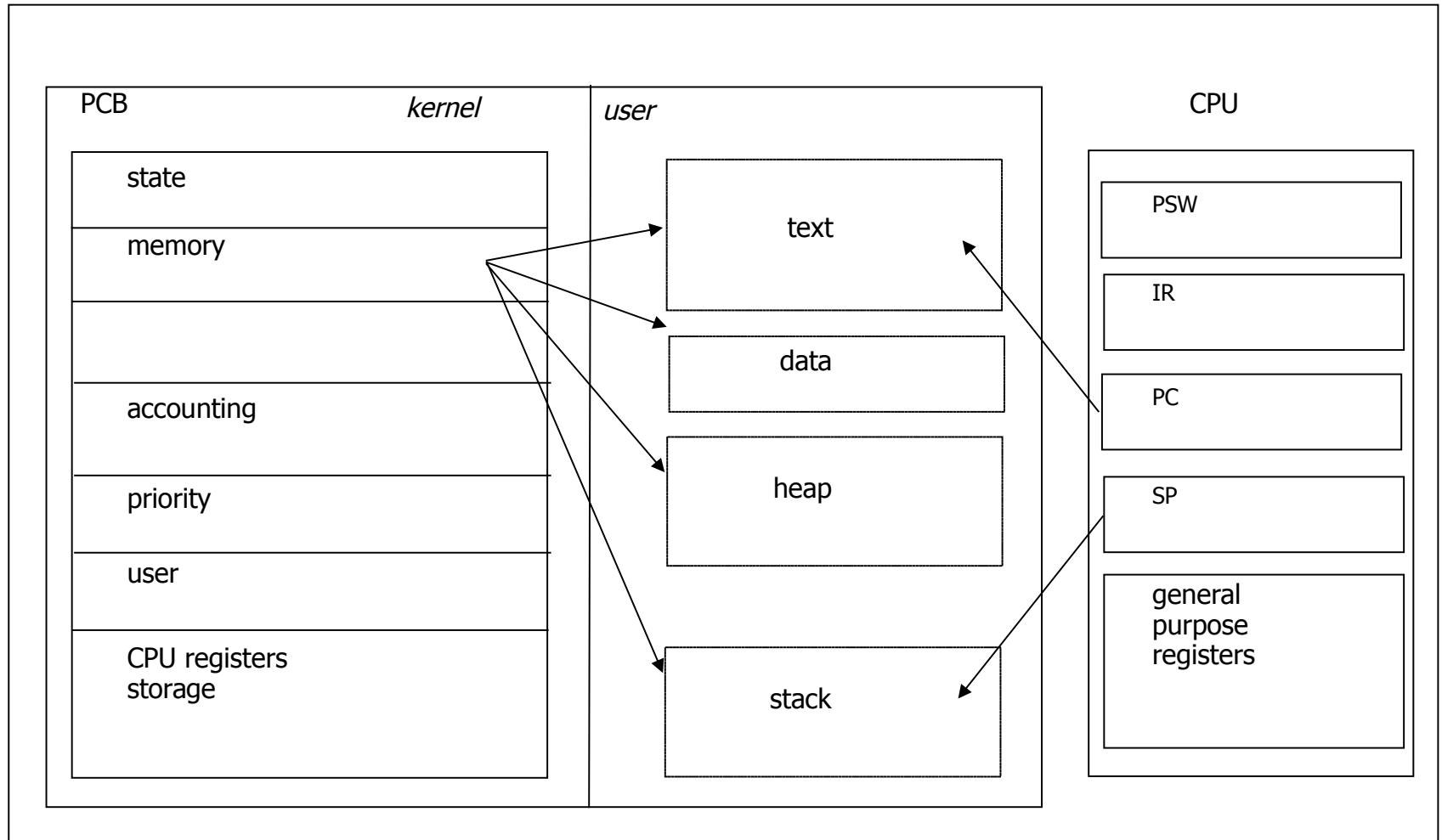
Prioritățile unui proces

- Prioritățile sunt dinamice și se calculează din secundă în secundă
- Exemplu UNIX:
 - **prioritate = baza + utilizare_recentă_CPU / constantă + valoare_nice**
 - **baza** = 60 (ceasul îintrerupe de 60 ori/sec);
 - **valoare_nice** se presupune nulă;
 - nucleul calculează **utilizare_recentă_CPU** prin împărțirea numărului de tacti la 2 în momentul calculului priorității;
 - **constantă** este 2.

Process Control Block (PCB)

- ❑ Zonă ce face parte din imaginea unui proces pe lângă zonele text, date și stivă
- ❑ Furnizează date cu privire la:
 - identificarea procesului: pid, uid, real uid, real gid, effective uid, efective gid
 - informațiile de stare: regiștrii vizibili utilizatorului, de obicei de la 8 la 32, uneori pe unele mașini RISC peste 100
 - informațiile de control al procesului

Process Control Block (PCB)



Structuri de date pentru gestiunea proceselor (Unix)

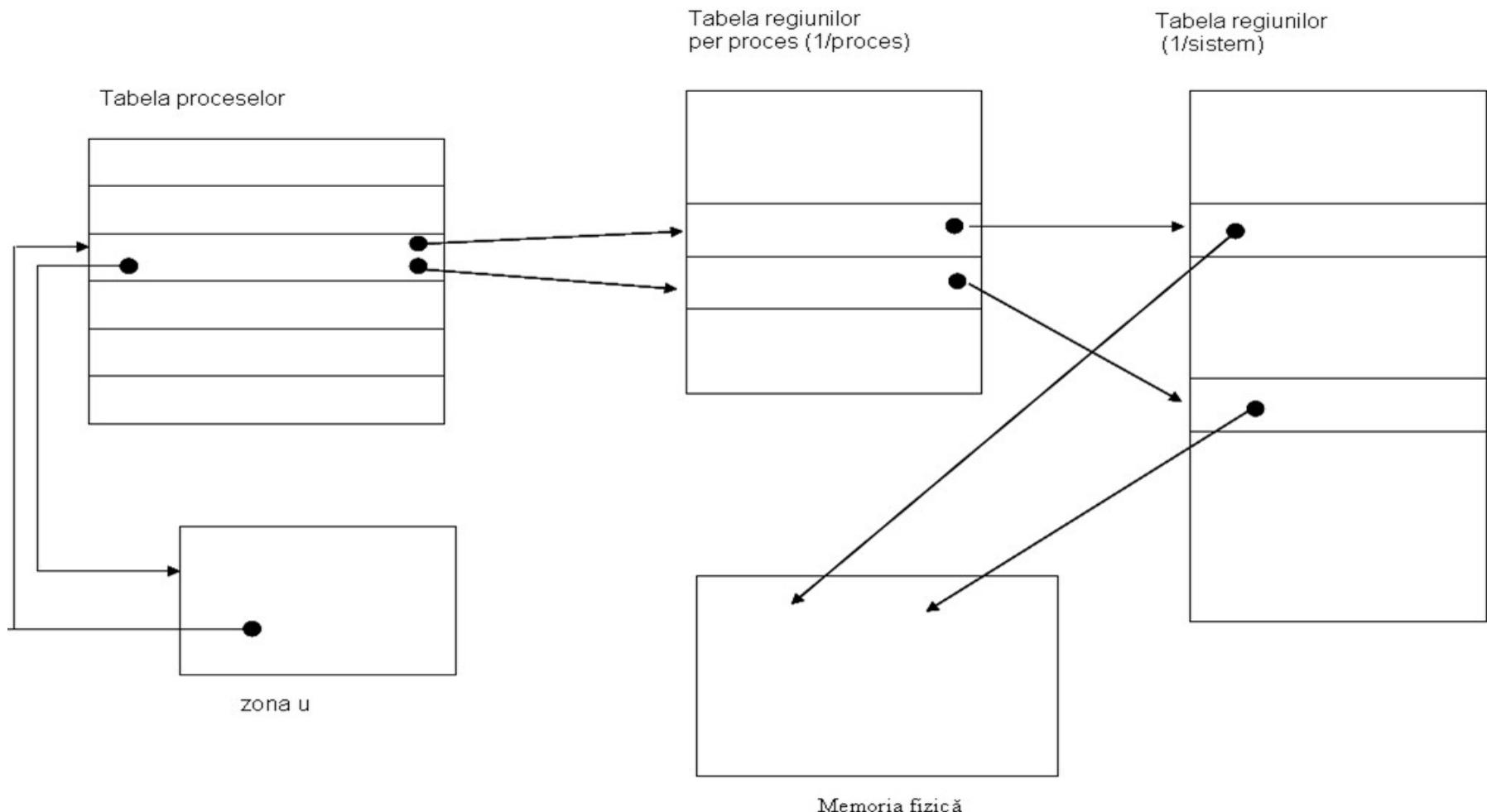


Tabela proceselor

- ❑ unică
- ❑ alocată în mod static de nucleu
- ❑ dimensiunea este fixată la generarea sistemului
- ❑ limitează numărul de procese pe care nucleul le poate gestiona pe sistemul respectiv.
- ❑ câmpuri:
 - starea procesului;
 - localizarea procesului în memoria internă sau în memoria secundară folosită pentru swapping;
 - dimensiunea procesului;
 - identificatorii ataşați utilizatorului și grupului său;
 - identificatorul procesului;
 - descriptorul evenimentului care a produs trecerea procesului în starea de aşteptare;
 - parametrii de planificare pentru obținerea procesorului;
 - semnalele trimise procesului, dar încă netratate;
 - diferiți timpi care indică timpul de execuție în mod utilizator și în mod nucleu pentru calculul priorității procesului;
 - un câmp este folosit pentru SIGALARM.

Zona u (U area)

- este generată și atașată unui proces la crearea lui
- este accesibilă nucleului numai în timpul execuției procesului la care este atașată
- Variabila **u** folosită de nucleu conține adresa virtuală a **zonei u** a procesului în curs de execuție
- Câmpurile zonei conțin:
 - un pointer la intrarea în tabela proceselor corespunzătoare procesului la care este atașată zona u;
 - identificatorul utilizatorului real și efectiv, în funcție de care se stabilesc drepturile de acces la fișiere, cozile de mesaje, memorie comună, semafoare, etc.
 - timpii de execuție ai procesului și descendenților săi în mod utilizator și nucleu;
 - modul de reacție a procesului la semnale (ignore, prelucrate, tratare de nucleu);
 - identificatorul terminalului de control („login terminal”) asociat cu procesul;
 - eroarea apărută în timpul apelului unei funcții de sistem;
 - valoarea returnată de o funcție de sistem;
 - parametrii de I/O: tipul transferului, adresa din spațiul procesului unde/de unde se transferă, deplasamentul în fișier etc;
 - directorul curent;
 - tabela descriptorilor de fișiere utilizator (TDFU);
 - dimensiunea limită a procesului și a fișierelor;
 - masca pentru drepturile de acces la fișierele create

Tabela regiunilor

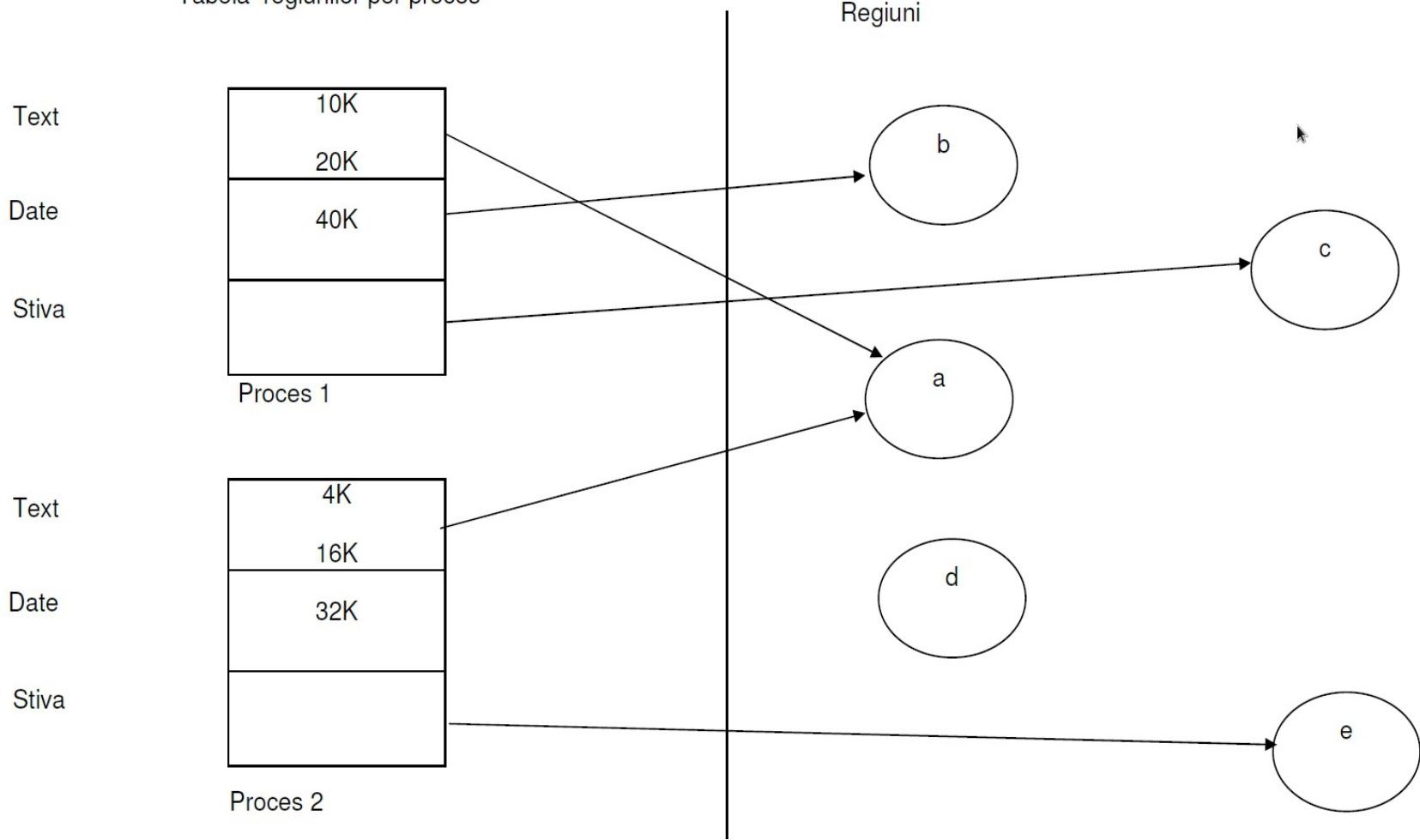
- este gestionată de nucleu
- O intrare în tabelă conține informațiile necesare identificării unei regiuni în memoria fizică internă
- O regiune este o zonă contiguă din spațiul de adrese virtual, care este tratat ca un obiect distinct ce poate fi partajat sau protejat
- Câmpurile:
 - un pointer la i-node-ul fișierului al cărui conținut se găsește în regiune;
 - tipul regiunii: text, date, memorie partajată, stivă;
 - dimensiunea regiunii;
 - adresa regiunii în memoria internă;
 - starea regiunii (poate fi o combinație de: blocat, în cerere, în curs de încărcare, validă – conținutul este încărcat în memorie și accesibil);
 - numărul de procese care referențiază regiunea.

Tabela regiunilor per proces

- asociată unui proces
- poate intra în **tabela proceselor**, în **zona u**, sau într-o **zonă de memorie alocată acestui scop**
- cele mai multe implementări plasează această zonă în tabela proceselor
- conține:
 - un pointer la intrarea corespunzătoare în tabela regiunilor;
 - adresa virtuală de început (start) a regiuni; pentru proceze diferite, o regiune partajată poate vedea adrese virtuale de început diferite;
 - drepturile de acces la regiune: read-only, read-write, read-execute.
 - conceptul de regiune este independent de tehnica de gestionare a memoriei prin paginare la cerere, prin partitiorare dinamică și swapping, prin segmentare etc.

Partajarea unei regiuni

Tabela regiunilor per proces



Contextul unui proces

- **Preemptarea** – reprezintă acțiunea prin care se întrerupe execuția unui task, fără cooperarea acestuia, cu intenția de a se putea continua execuția din același punct la o data ulterioară.
- Pentru realizarea acestui lucru este necesară realizarea unei operații denumită **schimbare de context (context switch)**.
- Această acțiune este realizată de către un task privilegiat, sau de o parte a sistemului de operare numită **planificator**, ce are posibilitatea de a preempta sau întrerupe și a relua execuția altor task-uri din sistem.
- **Contextul unui proces** reprezintă un set minimal de date utilizate de un proces, date ce trebuie salvate pentru a se putea permite întreruperea execuției unui proces într-un anumit moment și continuarea execuției la o data ulterioară.

Contextul unui proces (2)

Partea statică a contextului

Context nivel utilizator

- Text
- Date
- Stivă
- Memoria partajată

Intrarea în tabela proceselor

- Zona u
- Tabela regiunilor per proces
- Intrarea în tabela regiunilor
- Tabela paginilor

Partea statică a contextului
nivel utilizator

Partea dinamică a contextului

Stiva nucleu

Salvare context registri la intreruperi

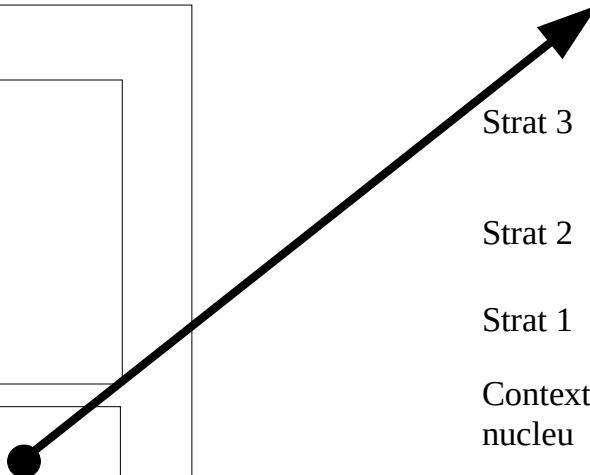
Stiva nucleu

Salvare Context Registri la apel sistem

Stiva nucleu

Salvare Context Registri pentru nivel utilizator

Nivel utilizator



Contextul unui proces (3)

□ **contextul utilizator** (spațiul de adrese utilizator):

- segmentul de text
- segmentul de date (inclusiv cele partajate din memoria comună)
- segmentul stivă-utilizator;

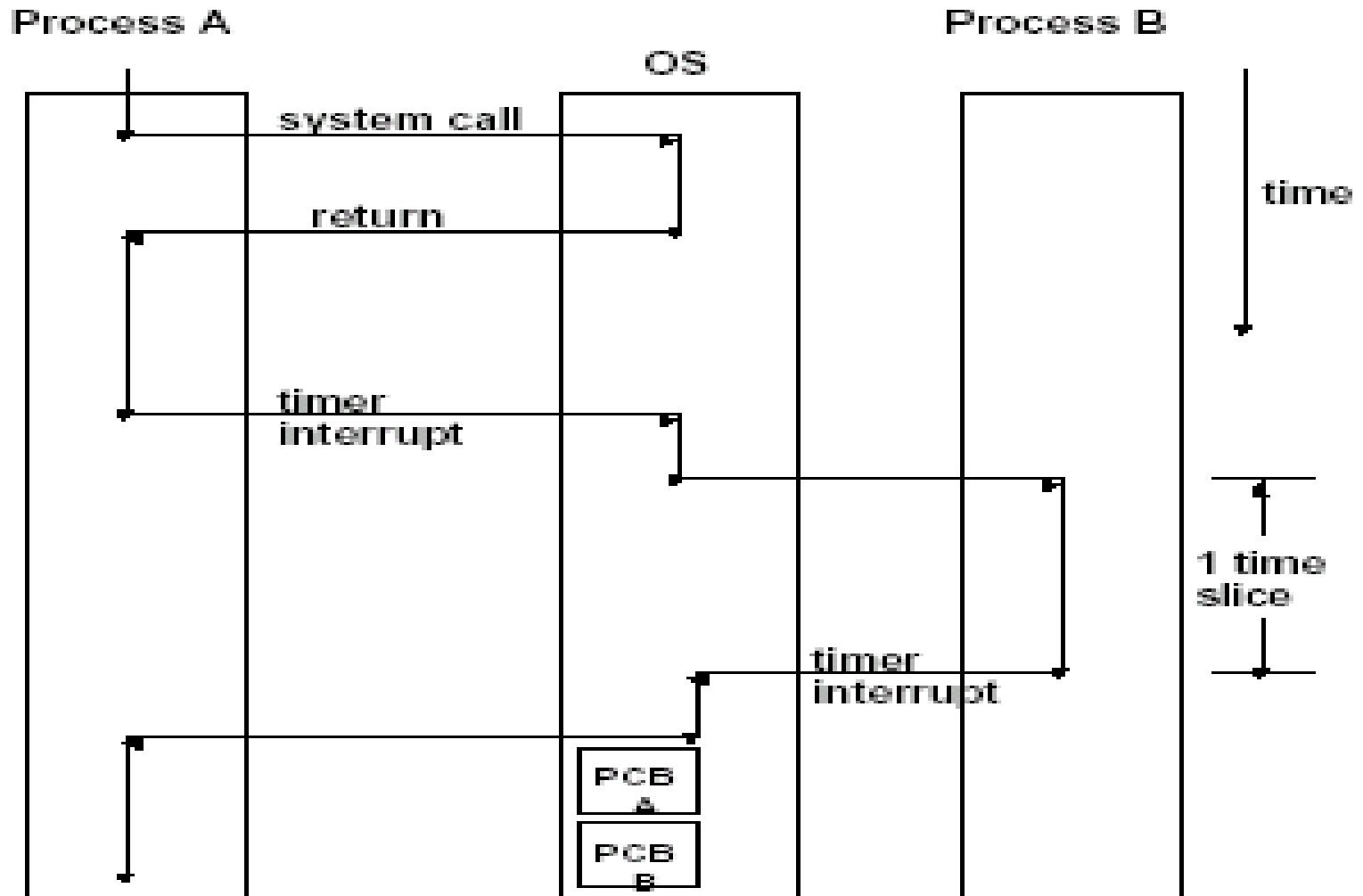
□ **contextul registrelor** (registrele hardware):

- numărătorul de instrucțiuni
- registrul de stare program
- registrul de stivă ("stack pointer" – stiva referită (utilizator sau nucleu) este funcție de modul de execuție: modul utilizator sau modul nucleu),
- registrii generali.

□ **contextul nivel nucleu** (structurile de date ale nucleului legate de proces):

- intrarea în tabela proceselor
- zona u
- tabela regiunilor per proces, intrările corespunzătoare în tabela regiunilor (în cazul gestionării memoriei prin paginare la cerere). Regiunile partajate de text sau date se consideră că fac parte din contextul fiecărui proces care le partajează.
- stiva nucleu – conține părțile stivei referitoare la apelul funcțiilor nucleului, care se execută în modul nucleu. Cind procesul se execută în mod utilizator stiva nucleu este goală.

Schimbarea de context



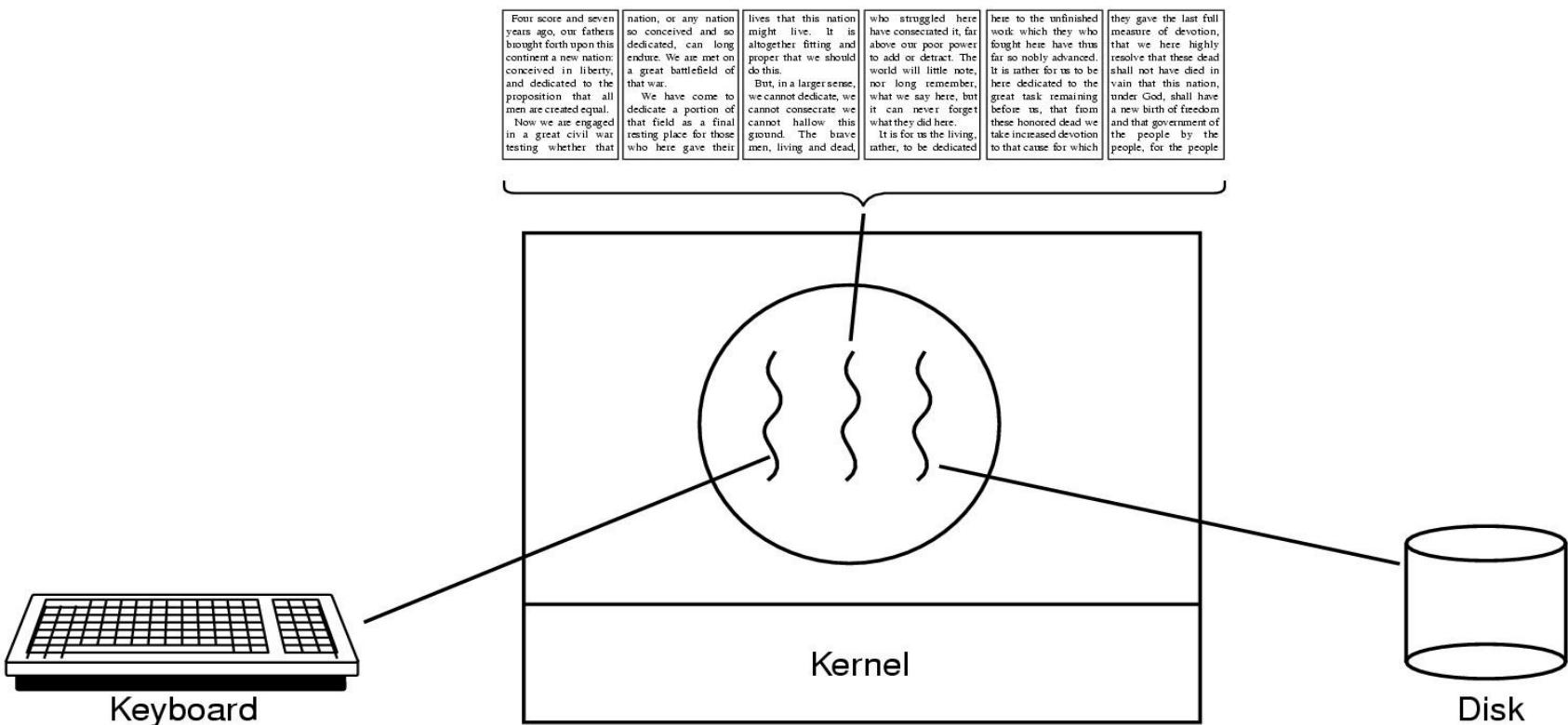
Schimbarea de context (2)

- Etapele schimbării de context:
 - decizia dacă trebuie să se execute o “schimbare de context”;
 - salvarea contextului vechiului proces;
 - alegerea noului proces de către planificatorul de procese;
 - restaurarea contextului noului proces.

Thread-uri (fire de execuție)

Definiție:

- Un thread este o entitate distinctă în cadrul unui proces (cea mai mică unitate de procesare), pe care kernelul o poate planifica pentru execuție și reprezintă unul sau mai multe subtask-uri în cadrul task-ului executat de un proces



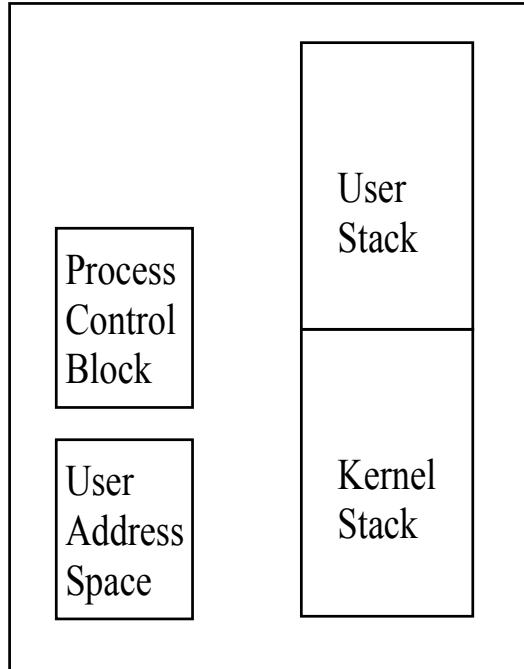
Thread-uri (2)

□ Caracteristici:

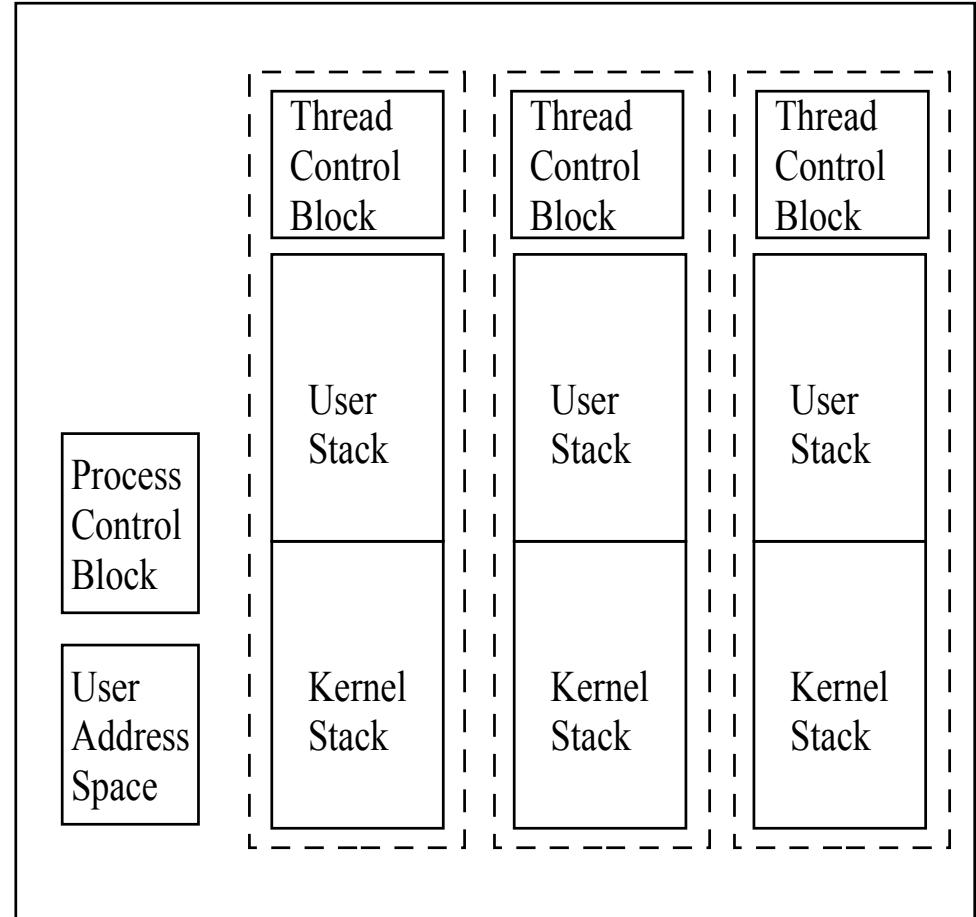
- are propria sa stare de execuție (running, ready etc),
- are stivă proprie
- are context propriu
- are acces la memoria și resursele procesului (partajate între firele de execuție).
- îi poate fi întreruptă execuția.

Modelul procesului cu unul sau mai multe thread-uri

Modelul procesului cu un singur thread



Modelul procesului cu mai multe thread-uri



Relația între thread-uri și procese

- 1 la 1:
 - fiecare proces are un singur thread (sistemele Unix – la începutul lor) ;
- n la 1:
 - fiecare proces poate avea mai multe thread-uri (cel mai folosit caz – Win2000, Linux, Solaris, OS/2, Match);
- 1 la n:
 - întâlnit în unele sisteme distribuite; un thread se poate muta pe mașini diferite și în spații diferite (Ra (Clouds), Emerald);
- n la n:
 - combinație între cele două de mai sus: poate comuta între domenii diferite pentru a realiza anumite operații cheie

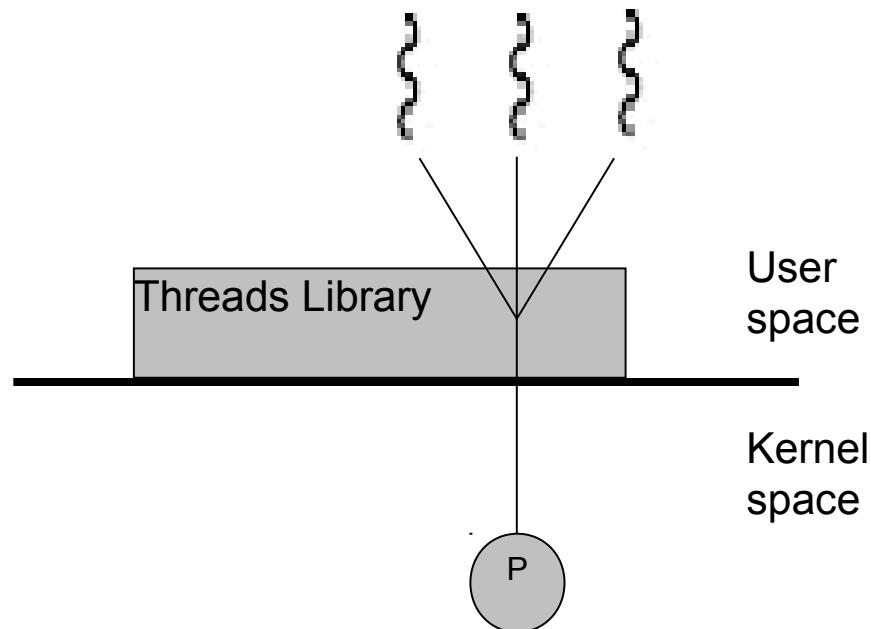
Thread-uri

- Operațiile care se pot realiza asupra thread-urilor
 - spawn: - crearea unui nou thread;
 - block: - aşteptarea unui eveniment;
 - unblock: - apariția evenimentului duce la deblocare;
 - finish: - terminarea thread-ului.

Implementarea și gestionarea thread-urilor

■ la nivel utilizator:

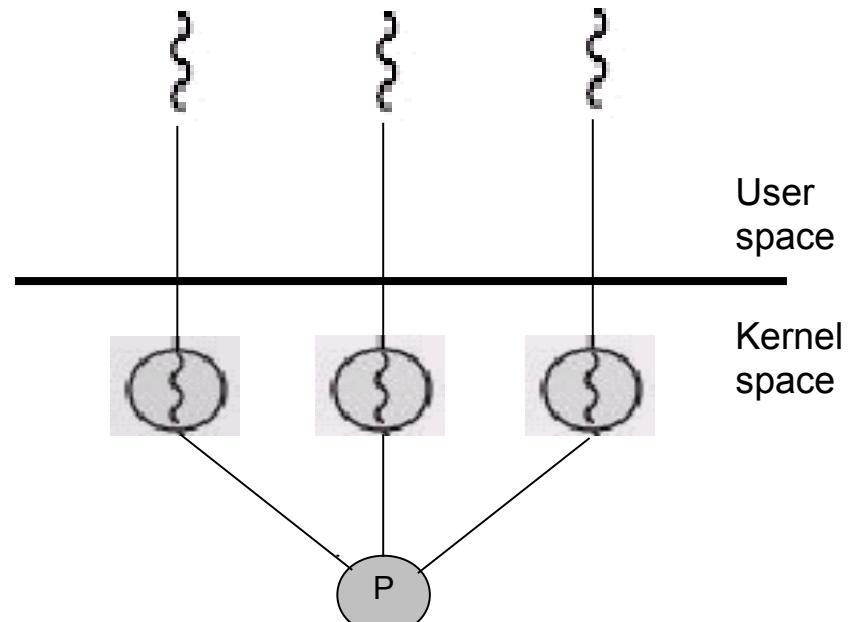
- aplicațiile creează și gestionează thread-urile prin anumite funcții de bibliotecă;
- sistemul gestionează procesul ca un întreg;
- pot apărea probleme la apelurile sistem blocante – când un thread se blochează sunt blocate toate thread-urile procesului;
- nu suportă sisteme multiprocesor, pot rula pe orice sistem de operare.



Implementarea și gestionarea thread-urilor (2)

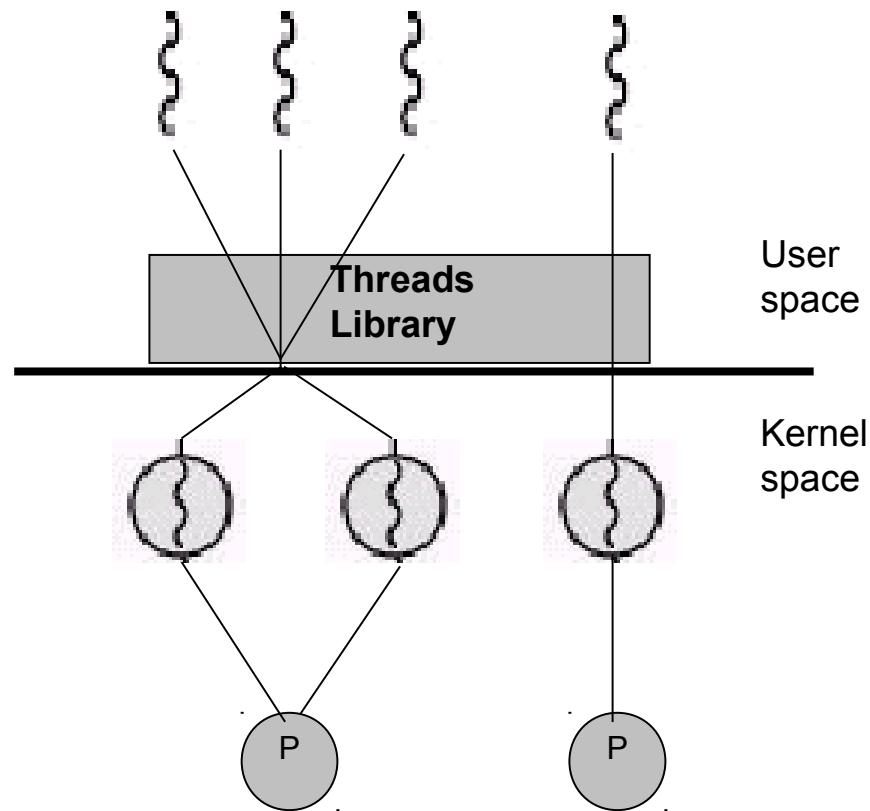
□ la nivel kernel:

- kernelul gestionează thread-ul;
- suport pentru sistemele multiprocesor;
- kernelul însuși poate avea mai multe thread-uri;
- este nevoie de comutarea între modurile user și kernel
- Exemplu: Windows 2000 și OS/2.



Implementarea și gestionarea thread-urilor (3)

- combinație între nivel utilizator și kernel:
 - fiecare thread al kernelului poate avea mai multe thread-uri utilizator;
 - crearea thread-urilor se face în spațiul utilizator (exemplu: Solaris).



Thread-uri

□ Avantaje:

- un singur proces poate avea mai multe thread-uri,
- thread-urile partajează resursele procesului
- thread-urile pot comunica foarte rapid între ele prin intermediul memoriei comune fără intervenția sistemului de operare.
- crearea unui thread durează mai puțin decât crearea unui proces.
- un thread se poate bloca fără a bloca activitatea celorlalte thread-uri sau a întregului proces.
- avantajele thread-urilor pot fi mărite dacă avem un sistem multiprocesor.

Multithreaded programming



Sursa:

http://www.reddit.com/r/aww/comments/2oagj8/multithreaded_programming_theory_and_practice

<http://highscalability.com/blog/2014/12/16/multithreaded-programming-has-really-gone-to-the-dogs.html>

Acțiuni care afectează execuția thread-urilor:

- ❑ suspendarea unui proces implică suspendarea tuturor thread-urilor aceluia proces deoarece thread-urile partajează același spațiu de adrese;
- ❑ terminarea procesului atrage terminarea tuturor thread-urilor.
- ❑ execuția unui thread poate fi terminată înaintea terminării execuției procesului.

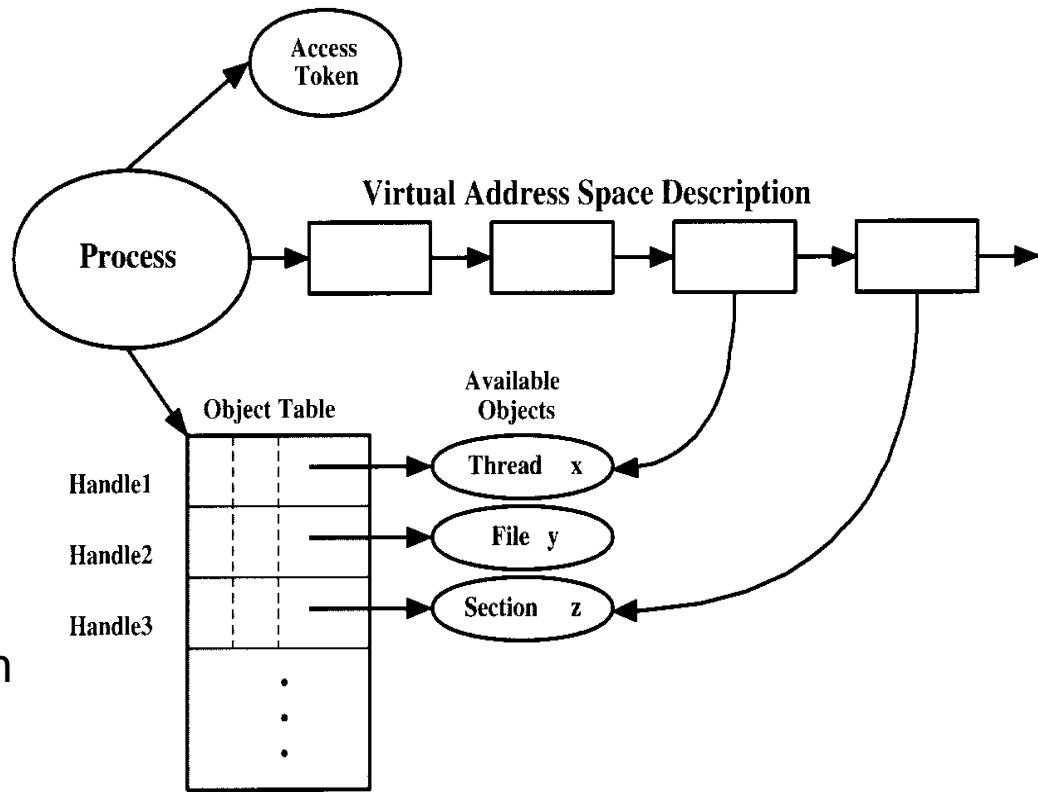
Studii de caz: Windows 2000

- ❑ Caracteristicile proceselor :
 - sunt implementate ca obiecte ;
 - pot conține mai multe thread-uri ;
 - posibilități de sincronizare atât a proceselor, cât și a thread-urilor.

Windows 2000

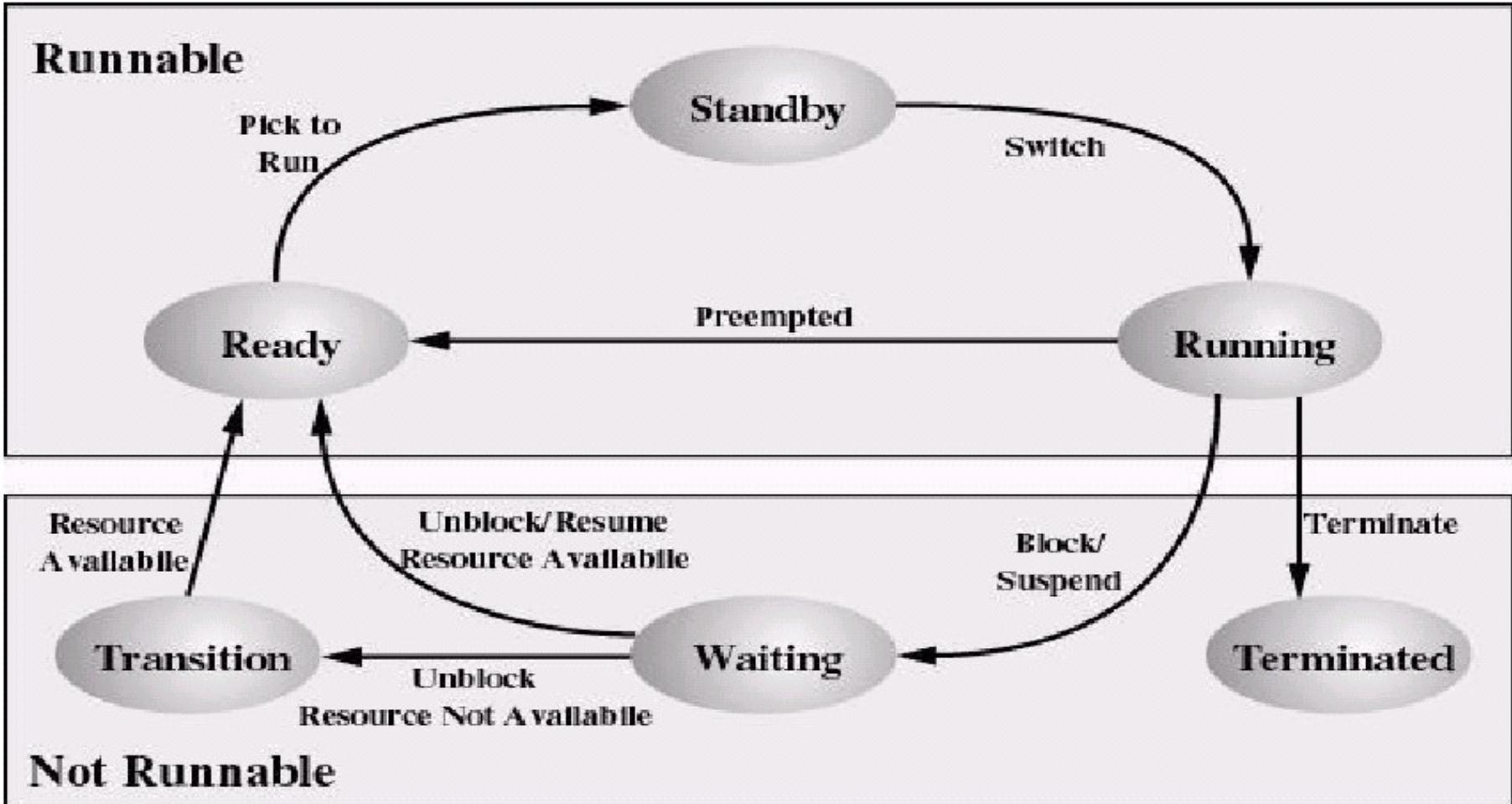
□ Componentele unui proces :

- access token : folosit la validarea accesului utilizatorului ;
- spațiul virtual de adrese
- object table : conține informații despre obiectele pe care le deține procesul
- fiecare thread are propriul său handler (descriptor – un număr care face referință la un obiect).



Windows 2000

Stările unui thread



Solaris

- Suportul pentru thread-uri este proiectat astfel încât să ofere o mare flexibilitate în exploatarea resurselor procesorului
- Concepte folosite de Solaris :
 - Process: proces Unix normal care include spațiul de adrese utilizator, stiva și PCB (process control block).
 - User-level threads:
 - Implementarea thread-urilor prin intermediul unei biblioteci în spațiul de adrese al procesului, invizibile sistemului de operare.
 - Implementarea la nivel utilizator reprezintă interfața pentru paralelismul aplicațiilor.

Solaris (2)

□ Concepte folosite de Solaris (2):

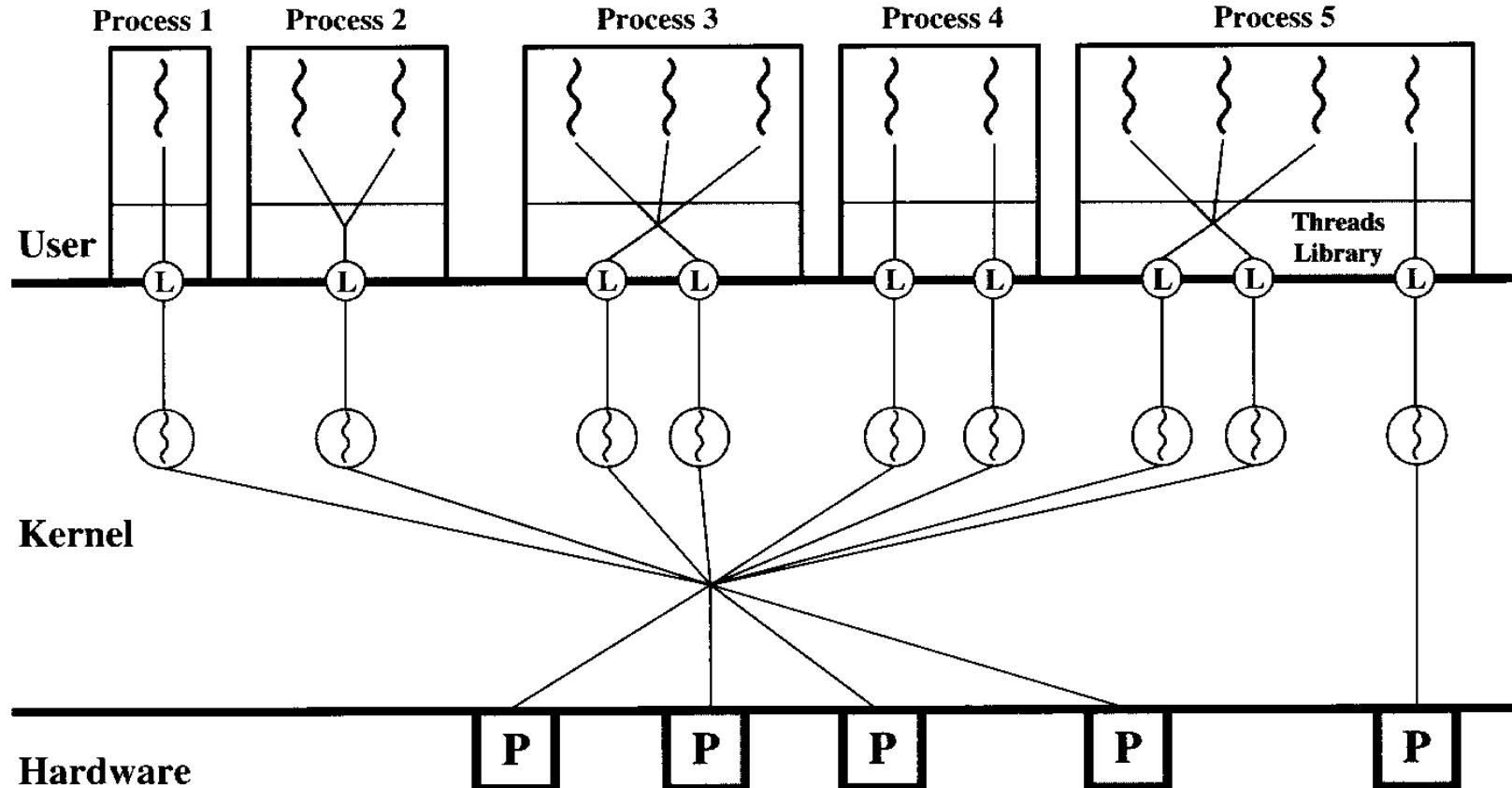
■ Lightweight processes (LWP):

- aceste procese pot fi văzute ca un intermediar între thread-urile utilizator și thread-urile kernel.
- Fiecare astfel de proces suportă unul sau mai multe thread-uri utilizator care vor corespunde unui thread kernel.
- Aceste procese sunt planificate independent de către sistemul de operare și pot rula în paralel pe mai multe procesoare.

■ Kernel threads:

- sunt entități fundamentale și sunt planificate și lansate în execuție pe unul din procesoarele sistemului.

Solaris - Arhitectura multithreading



{ User-level Thread



Kernel-level Thread



Lightweight Process



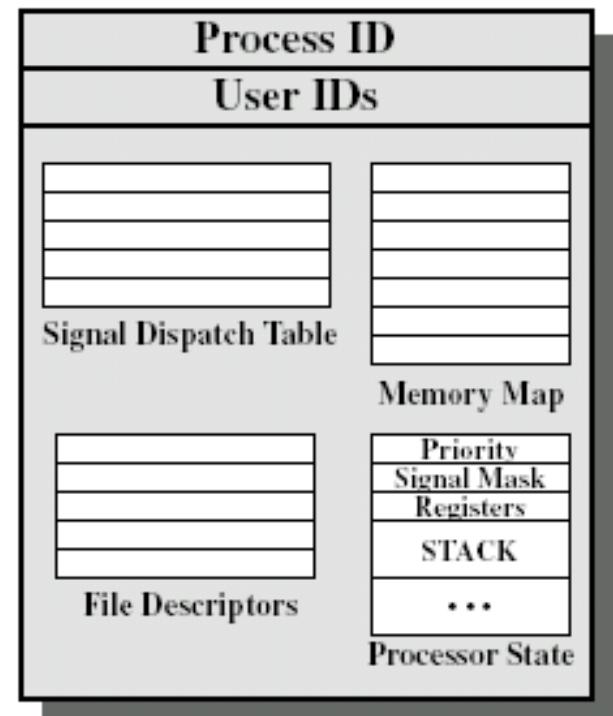
Processor

Solaris – Structura proceselor

□ sistem Unix structura proceselor include:

- processor ID
- user ID
- tabela de transmitere a semnalelor (pe care kernelul o folosește când decide ce să facă cind trimite un semnal unui proces)
- descriptorii de fișier (descriu starea fișierelor utilizate de un proces)
- harta memoriei (informații privind spațiul de adrese pentru procese)
- o structură privind starea procesorului (processor state structure - include stiva kernel pentru proces).

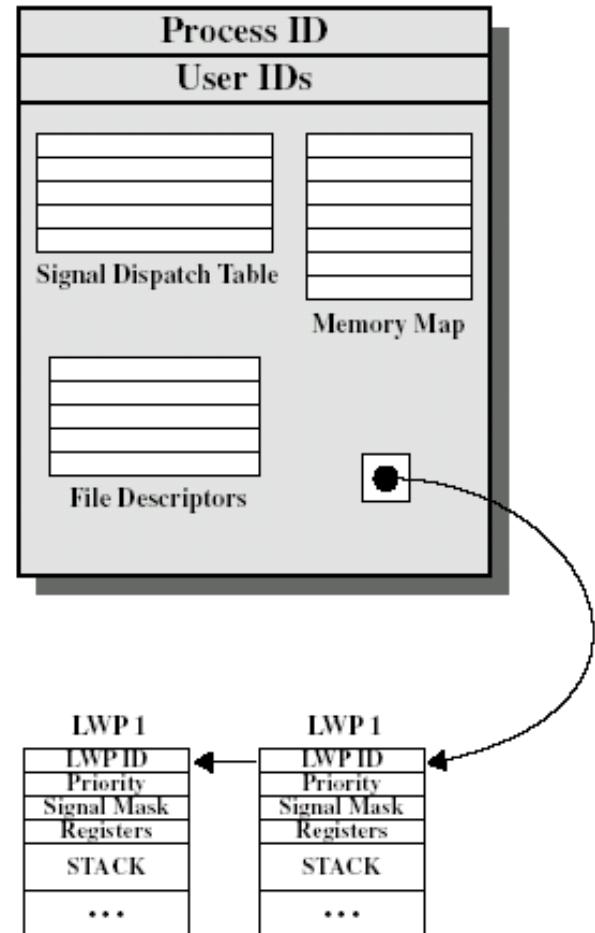
UNIX Process Structure



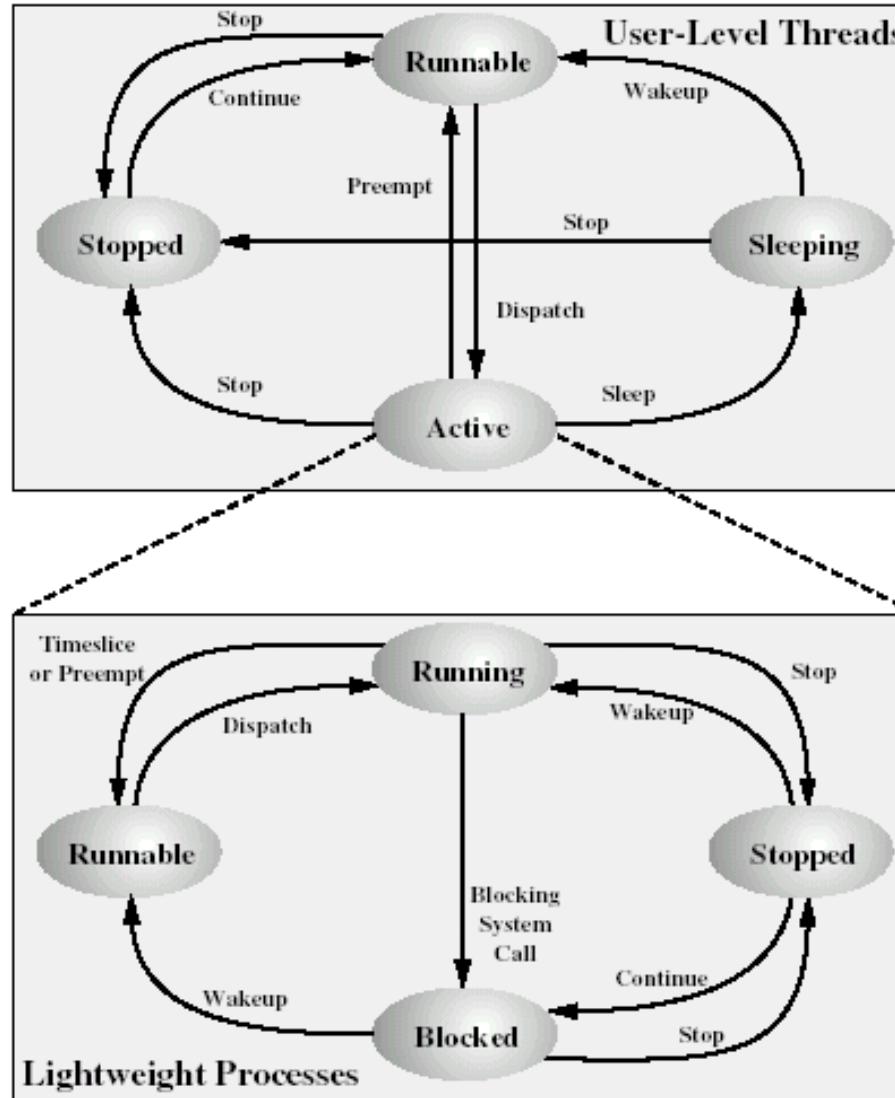
Solaris – Structura proceselor

- ❑ Se înlocuiește processor state structure cu o listă de structuri ce conțin câte un bloc pentru fiecare LWP.
- ❑ Structura unui astfel de bloc este următoarea:
 - LWP identifier
 - Prioritatea LWP și de aici cea a thread-ului kernel corespunzător
 - o mască de semnale care spune kernelului ce semnale vor fi acceptate
 - valorile salvate ale registrelor de la nivel utilizator atunci cînd LWP nu rulează
 - o stivă kernel pentru LWP care include argumentele apelurilor sistem,
 - rezultatele și codurile de eroare pentru fiecar nivel de apelare.
 - resursele utilizate
 - pointer către thread-ul kernel corespunzător
 - pointer către structura procesului

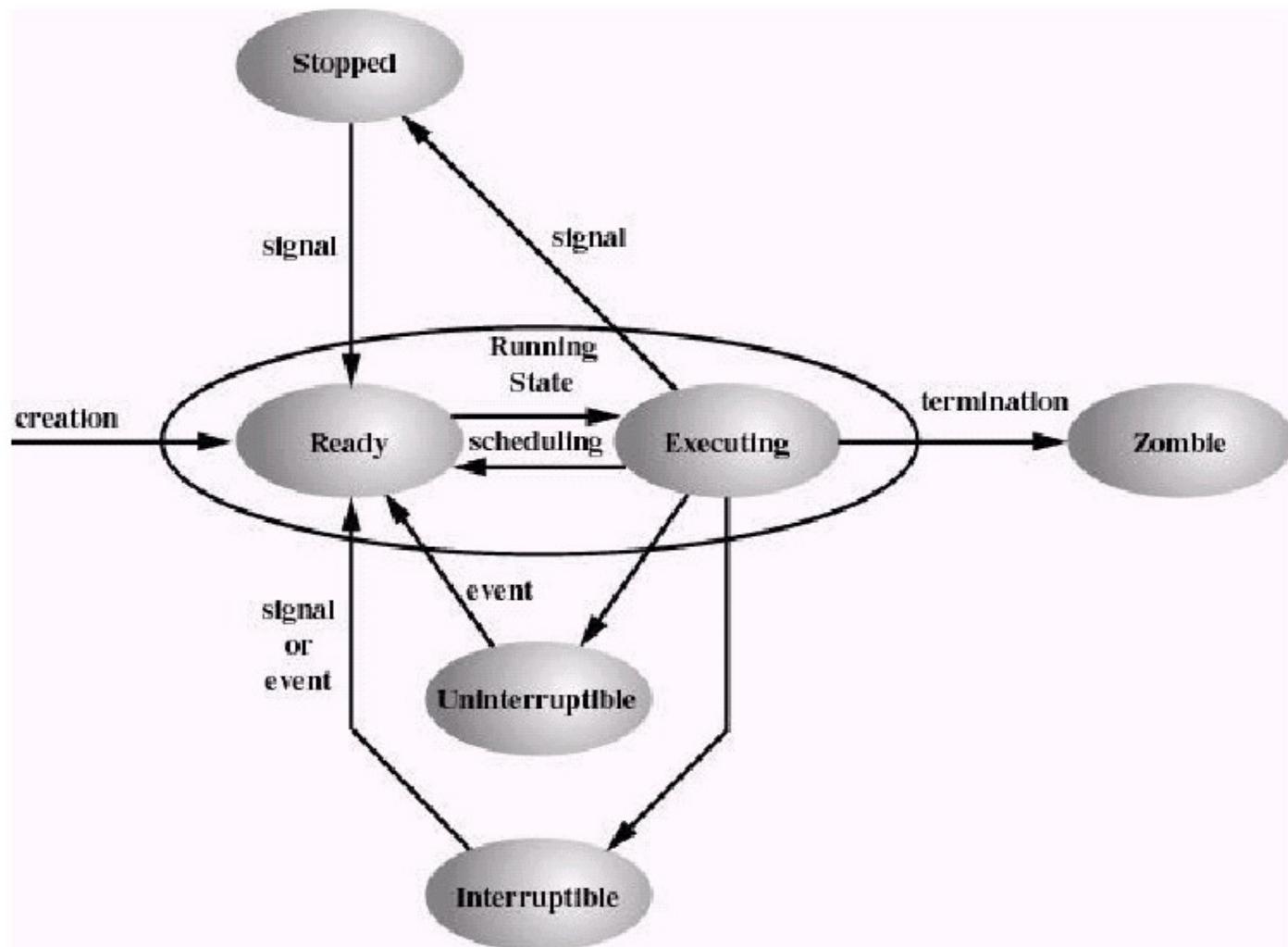
Solaris Process Structure



Solaris - Stările thread-urilor utilizator și LWP



Linux – Stările proceselor



Linux - task_struct

□ State:

- starea execuției unui proces (executing, ready, suspended, stopped, zombie).

□ Informații de planificare:

- informații folosite de Linux pentru planificarea proceselor.
- Un proces poate fi normal sau real time și are o prioritate.
- Procesele real-time sunt planificate înaintea celor normale și în cadrul fiecărei categorii există ierarhii de prioritate.
- Un contor ține minte timpul în care un proces este executat.

□ Identifieri:

- Fiecare proces are un identifier unic pentru utilizator și grup.
- Comunicația interprocese:
- Linux suportă mecanisme de comunicație interproces.

□ Legături (Links):

- Fiecare proces păstrează legături către procesul părinte, procesele care au același părinte cu el și către procesele fiu.

□ Informații despre timp:

- Informații despre timpul la care a fost creat procesul, timpul procesor consumat..

□ File system:

- pointeri către fișierele deschise de un proces.

□ Virtual memory:

- definește memoria virtuală atribuită aceluia proces.

□ Processor-specific context:

- informații despre registri și stivă

Linux – thread-uri

- ❑ Un proces nou este creat prin copierea atributelor procesului curent.
- ❑ Un proces nou poate fi “clonat” astfel încât partajează resursele (fișiere, semnale, memoria virtuală).
- ❑ Dacă două procese partajează memoria virtuală, ele vor funcționa ca thread-uri în cadrul unui singur proces.
- ❑ **Nu este definită o structură de date separată pentru un thread și nu există o distincție clară între proces și thread.**
- ❑ Pentru lucrul cu thread-uri există biblioteca <pthread.h> .

Sisteme de Operare



- Gestiunea proceselor
 - Sincronizarea proceselor
 - Secțiunea critică
 - semafoare, aşteptare activă, monitoare, bariere
 - Paradigme ale programării concurente

Sincronizarea proceselor

- Pentru execuție procesele au nevoie de acces la resursele sistemului : CPU, memorie, disc etc.
- Resursele pot fi:
 - locale sau globale,
 - private sau comune (partajabile).
- Toate resursele sunt critice (accesibile numai unui proces la un moment dat).
- Procesele concurează pentru obținerea accesului la resurse.

Sincronizarea proceselor

- **Definiție:** **Sincronizarea proceselor** reprezinta acțiunea de coordonare a activității mai multor proceze.
 - Sincronizarea presupune posibilitatea de a modifica starea unui proces la un moment dat până la apariția unor evenimente ce nu se afla sub controlul acestuia.
 - Sincronizarea impune aplicarea excluderii mutuale și o ordonare a evenimentelor.
- **Definiție:** Zona de program prin care se apelează o resursă critică se numește **secțiune critică (SC)**.
- Accesul unor proceze la resursele critice se face printr-un protocol de **excludere mutuală (EM)**, ce presupune o sincronizare ce permite modificarea stării proceselor și eventual comunicării.

Accesul la secțiunea critică

- ❑ intrarea în secțiunea critică
- ❑ tratarea resursei critice din secțiunea critică
- ❑ ieșirea din secțiunea critică.

Condițiile pentru realizarea excluderii mutuale într-o secțiune critică

- ❑ Un singur proces la un moment dat trebuie să execute instrucțiunile din secțiunea critică ;
- ❑ Dacă mai mult de un proces sunt blocați la intrarea în secțiunea critică și nici un alt proces nu execută instrucțiunile din secțiunea critică, într-un timp finit unul din procesele blocate se deblochează și va intra în secțiunea critică (pe rând vor intra și celelalte).
- ❑ Blocarea unui proces în afara secțiunii critice să nu împiedice intrarea altui proces în secțiunea critică.
- ❑ Să nu existe procese privilegiate (mecanismul să fie echitabil pentru toate procesele).

Excluderea mutuală - implementare

- ❑ Semafoare
- ❑ Așteptare pe condiție (așteptare activă)
- ❑ Monitoare
- ❑ Bariere

Semafoare

- **Definitie:** Semaforul este un **mecanism de sincronizare** a execuției proceselor care acționează în mod concurrent asupra unor resurse partajate.
 - Poate fi o variabilă sau o structură de date abstractă
 - Este utilizat pentru controlul accesului la o resursă comună într-un mediu multiproces, multiutilizator
- Semafoarele se caracterizează prin:
 - Valoare: val(s)
 - coadă de aşteptare: Q(s)
- Semafoarele sunt de două tipuri :
 - binare (lucrăază cu valori de 0 și 1)
 - întregi (lucrăază cu valori între -n și n).
- Cozile de aşteptare sunt de tip FIFO.
- **Operațiile cu semafoare:**
 - creare
 - distrugere
 - operația de intrare în secțiunea critică : p(s), p = cerere de intrare în S.C.
 - operația de ieșire din secțiunea critică : v(s), v = cerere de ieșire în S.C.

Semafoare

P(s) :

```
val(s) = val(s)-1 ;  
if ( val(s) < 0 )  
{  
    //trece procesul în coada de așteptare a semaforului respectiv  
    // schimbă starea procesului în blocat  
}
```

V(s):

```
val(s) = val(s) +1 ;  
if ( val(s) ≤ 0 )  
{  
    //scoate primul proces din coada de așteptare a semaforului  
    // schimbă starea procesului în gata de execuție  
}
```

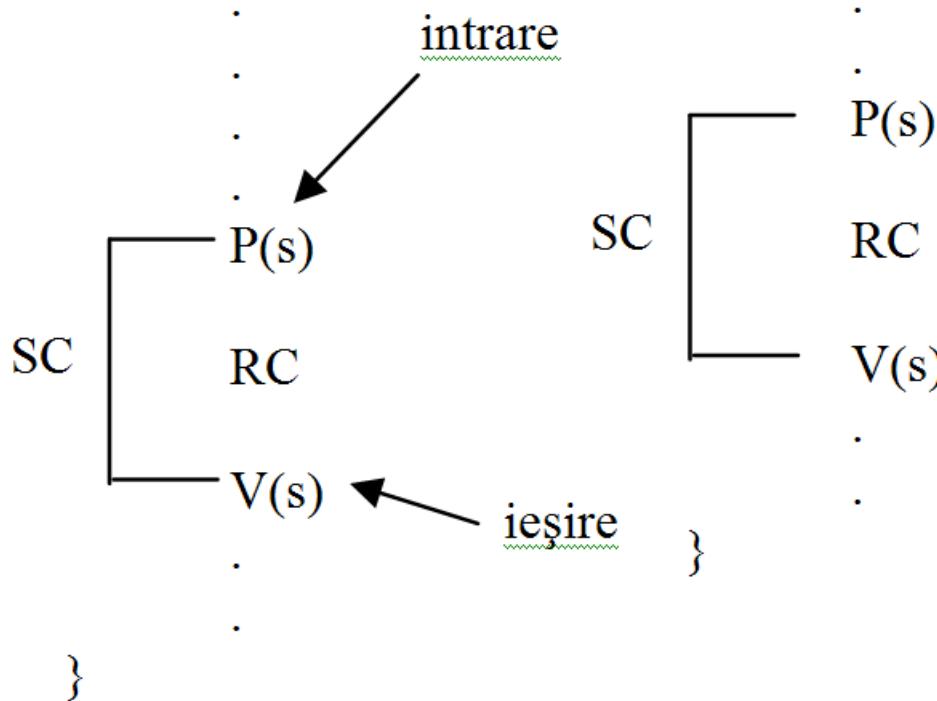
Excluderea mutuală pentru 2 procese

int s=1

P1:

while()

{



Valoarea unui semafor la un moment dat

- **$V(s) = V_0(s) + nv(s) - np(s)$**
 - $V(s)$ este valoarea unui semafor la un moment dat;
 - **np** este numarul de treceri prin $P(s)$
 - **nv** este numărul de treceri prin $V(s)$
- Dacă $V(s) > 0$, acest număr ne dă numărul de procese ce pot executa instrucțiunile din secțiunea critică.
- Daca $V(s) < 0$, atunci acest număr ne dă numărul de procese blocate la semafor.
- Secțiunile critice pot fi atât imbricate cât și întrețesute.
- Numărul de procese trecute de semaforul **s** este:
 - **$nt(s)=\min\{V_0(s)+nv(s), np(s)\}$**

Semafoare private

```
int s=0
P1:
while()
{
    .
    .
    .
    .
    .
    .
    P(s)
    SC
        RC
    V(s)
    .
    .
}
P2:
while()
{
    .
    .
    .
    if(cond)
        V(s)
    .
    .
}
```

- ❑ numai un singur proces poate aplica P și V asupra lui
- ❑ celelalte procese putând executa numai V.
- ❑ Valoarea inițială a unui semafor privat este 0.
- ❑ Semaforul privat permite sincronizarea execuției unui proces cu o condiție externă lui (eventual cu un alt proces).

Proiectarea semafoarelor

- Secțiunile critice trebuie încadrate de P și V ceea ce uneori este mai greu de urmărit ;
- Un proces nu poate fi distrus în SC ;
- Verificarea corectitudinii programelor nu este ușoară ;
- Gestiunea cozii poate duce la apariția unor probleme de proiectare.

Sincronizarea proceselor cu semafoare

- ❑ Semafoarele întregi se folosesc pentru a gestiona un nr. de resurse identice (componentele unei zone tampon și resurse fizice).
- ❑ Resursele se alocă la cerere și se eliberează după folosirea lor.
- ❑ Când nu mai există copii disponibile procesele se blochează.

Problemă: 3 resurse identice, un semafor și 3 procese

- execuția primitivelor P este critică

int s = 3;

p₁()

{

P(s);

(*) – se ocupa prima SC

P(s);

(*) – se ocupa a II-a SC

V(s);

V(s);

}

p₂()

{

P(s);

(*) – se ocupa prima SC

P(s);

(*) – se ocupa a II-a SC

V(s);

V(s);

}

p₃()

{

P(s);

(*) – se ocupa prima SC

P(s);

(*) – se ocupa a II-a SC

V(s);

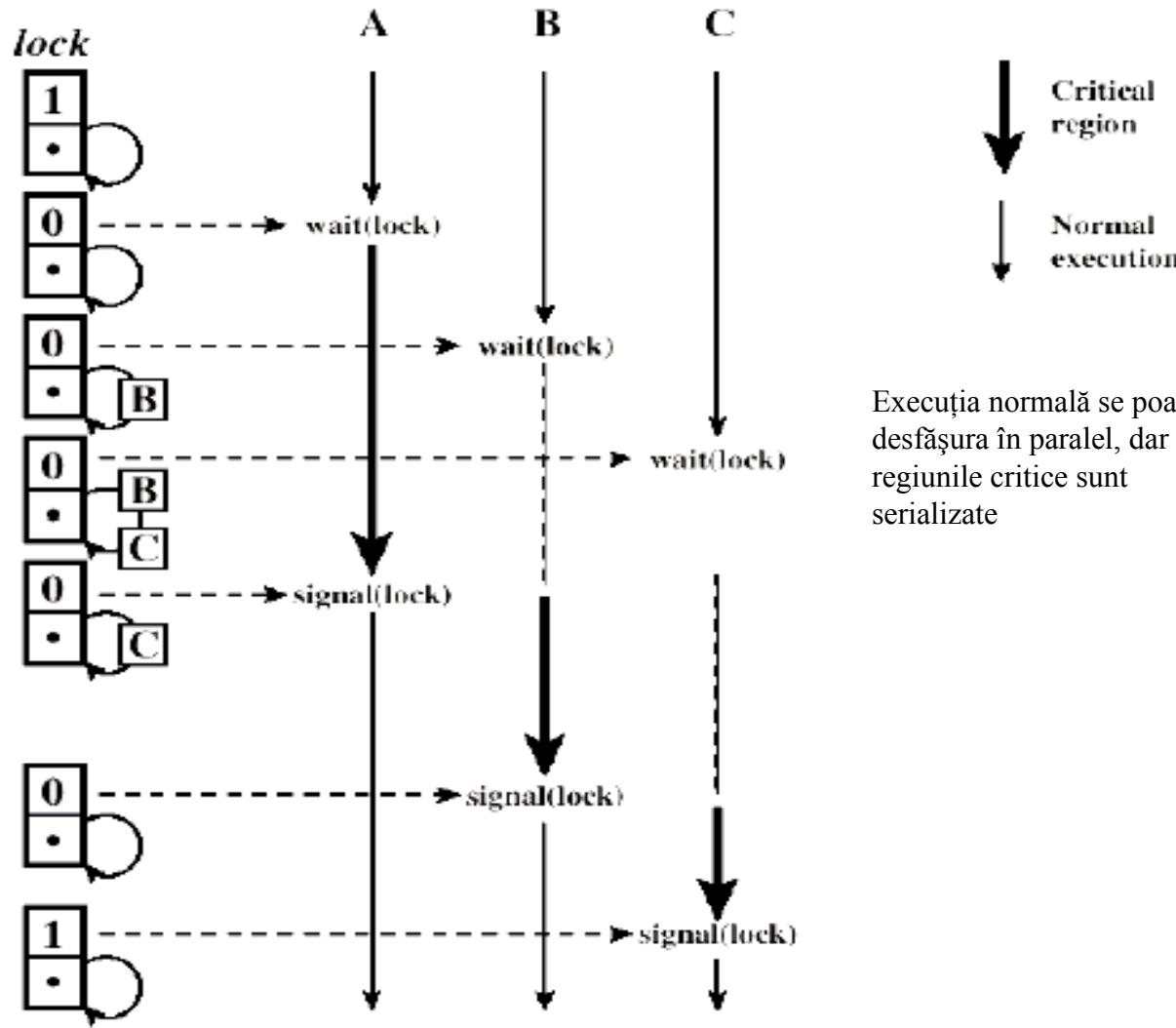
V(s);

}

Rezolvare

```
int s =3, s1 =2;  
    p1()  
    {  
        .....  
        P(s1);  
        P(s);  
        // – se ocupa prima RC  
        P(s);  
        // – se ocupa a II-a RC  
        V(s);  
        V(s);  
        V(s1);  
    }
```

Exemplu de execuție pentru 3 procese



Execuția normală se poate desfășura în paralel, dar regiunile critice sunt serialized

Sincronizarea prin aşteptare activă

□ condiții:

- există variabile comune care sunt testate;
- testul și modificarea variabilelor să fie operații indivizibile.

```

boolean flag [2] = {false, false};
int turn;
void P0()
{
    while (true)
    {
        flag [0] = true;
        turn = 1;
        while (flag [1] && turn == 1);

        /* RC */;
        flag [0] = false;
        ...
    }
}

```

```

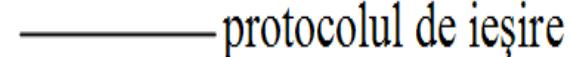
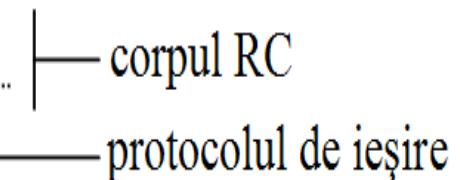
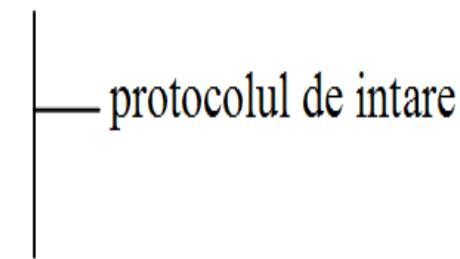
void P1()
{
    while (true)
    {
        flag [1] = true;
        turn = 0;
        while (flag [0] && turn == 0);

        /* RC */;
        flag [1] = false;
        ...
    }
}

```

SC are 3 componente :

- protocolul de intrare
- corpul
- protocolul de iesire



Dezavantajele aşteptării active

- consumarea inutilă de timp CPU pentru un proces care aşteaptă.
- Există 2 operații care trebuie realizate: **testarea** și **modificarea** flag (de aceea au fost introduse variabilele flag și turn).
- Gradul ridicat de dificultate în elaborarea protocoalelor de intrare și iesire lucru ce poate duce pe lângă neclarități și la apariția de erori.

Sincronizarea folosind monitoare

- Un monitor reprezintă o structură de date formată din:
 - **variabile de sincronizare** numite și **variabile condiții**,
 - **resurse partajate**
 - **proceduri de acces** la resurse. Procedurile pot fi: externe sau interne.

Sincronizarea folosind monitoare

```
monitor monitor-name
{
    declarații variabile locale;

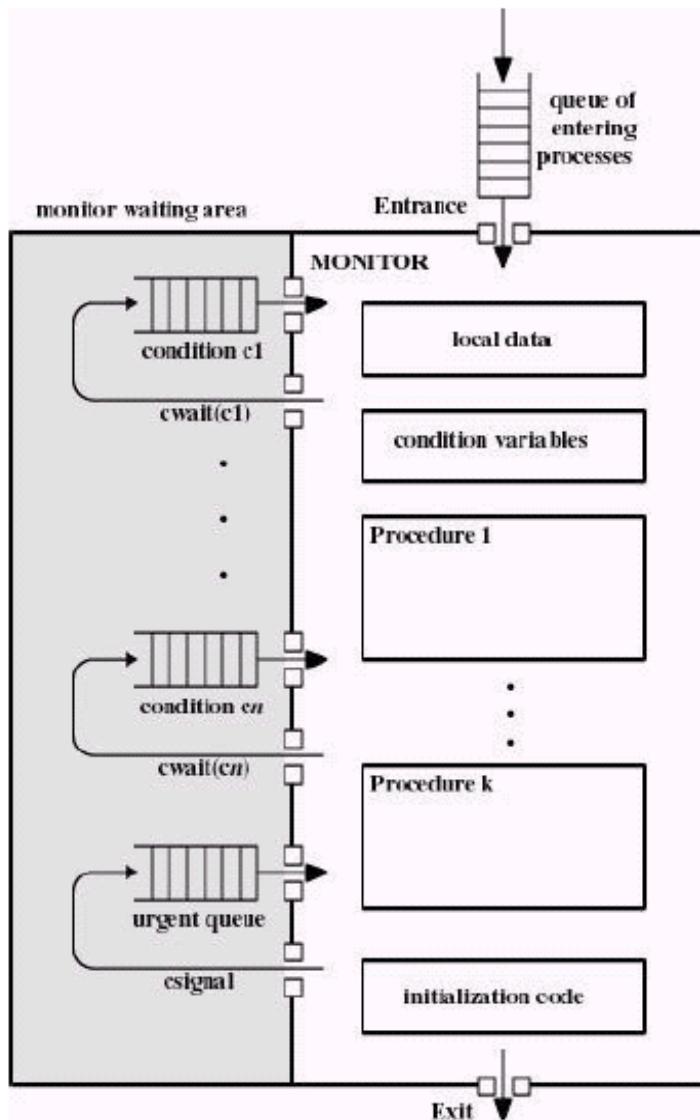
    procedure P1(...) {
        ...
    }

    ...

    procedure Pn() {
        ...
    }
}
```

- Variabilele locale ale unui monitor nu sunt vizibile decât pentru procedurile lui.
- Procedurile unui monitor sunt puncte de intrare (pot fi accesate din exterior) și ele se executa prin excludere mutuală.
- **Primitivelor de sincronizare**
 - **wait** – suspendă procesul care execută wait pe o variabilă de condiție și face disponibil monitorul pentru un alt apel;
 - **signal** – reactivează un proces în aşteptare pe o variabilă de condiție.

Structura unui monitor



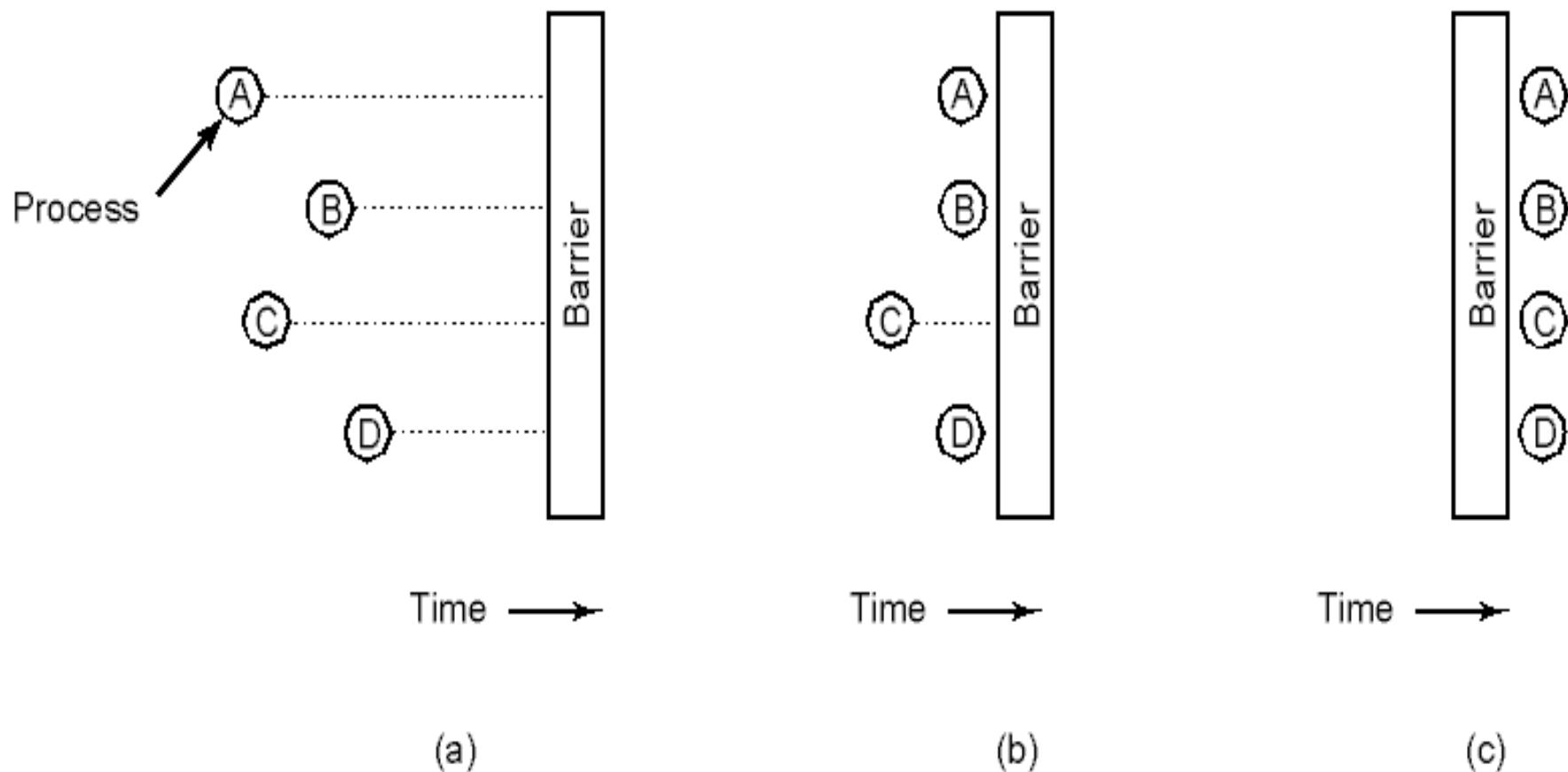
Cozile de aşteptare ale unui monitor

- Fiecare variabilă c_i are atașată o coadă și odată accesată o procedură a monitorului, un proces în aşteptare pe o condiție va ceda accesul unui alt proces în aşteptare la intrarea în monitor.
- Procesele suspendate după execuția lui *signal (c_i.signal)* nu eliberează excluderea mutuală în monitor deoarece:
 - fie există un proces în execuție a unei proceduri în monitor, eventual în aşteptare pe o condiție;
 - fie este același proces care-si continuă execuția când nu mai sunt altele de reactivat.
- Coada proceselor suspendate este mai prioritara față de aceea de la intrarea în monitor și a cozilor de aşteptare pe condiție

Stările unui proces la accesarea unei proceduri din monitor

- aşteptare în coada de intrare a monitorului ;
- aşteptare într-o coadă pe o variabilă de condiție (`wait`) ;
- suspendarea prin execuția signal, care reactivează un proces în aşteptare pe o variabilă de condiție ;
- execuția normală a instrucțiunilor unei proceduri din monitor.

Sincronizarea folosind bariere



Sincronizarea prin transmiterea de mesaje

- Mesajele pot avea dimensiune fixă sau nu.
- Dacă două procese P și Q vor să transmită mesaje trebuie să stabilească o legătură între ele.
- Detalii de implementare și gestionare a comunicațiilor sunt lăsate în grija sistemului de operare.
- Sistemul de transmitere a mesajelor trebuie să ofere funcții de tip:
 - send(destinație, mesaj)
 - receive(sursă, mesaj)

Sincronizarea prin transmiterea de mesaje

- Atât procesul care trimite mesajele, cât și cel care le primește pot fi blocate sau nu
- **Send/receive blocante**: ambele proceze sunt blocate până la terminarea comunicării ;
- **Send neblocant, receive blocant** : procesul ce execută send își continuă execuția imediat ce a trimis mesajul, iar procesul ce execută receive este blocat până la sosirea mesajului ;
- **Send/receive neblocante**: nici un proces nu așteptă terminarea operațiilor de comunicație;

Sincronizarea prin transmiterea de mesaje

Adresarea

- **directă**: în acest caz, funcțiile send/receive includ identificatorul procesului destinație ;
- **indirectă**: în acest caz mesajele sunt trimise unei structuri de date partajate ce constau în cozi numite mailbox (casuțe poștale)

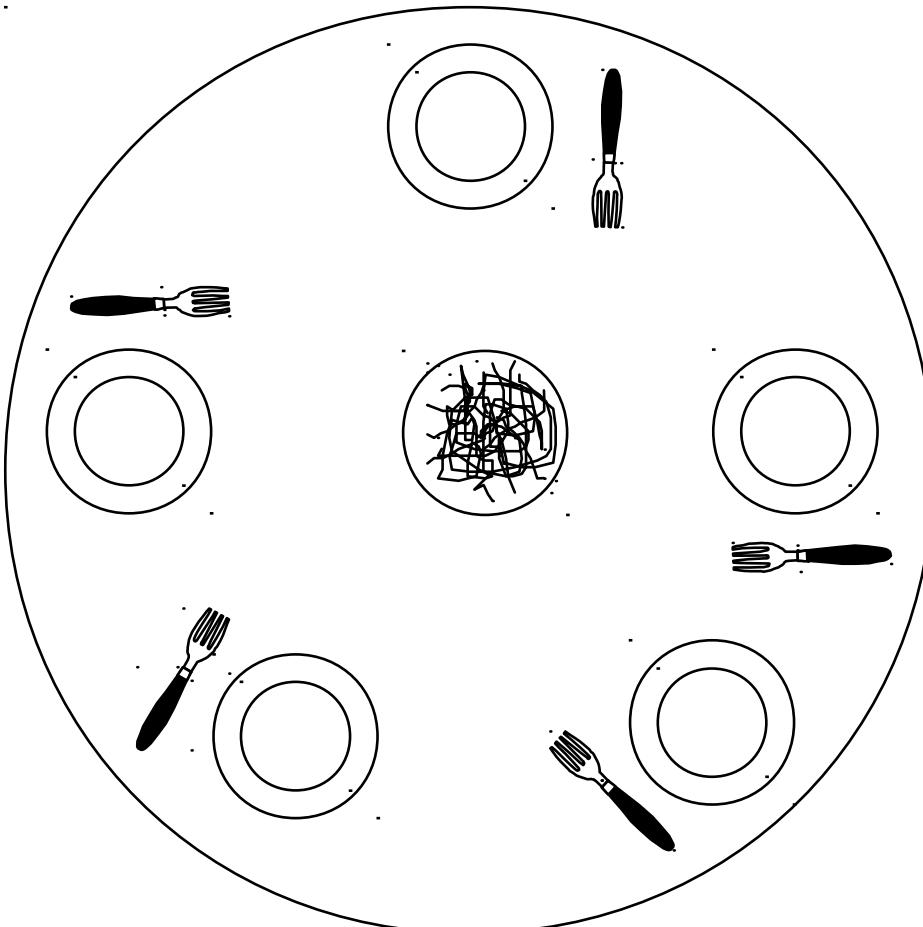
Excluderea mutuală folosind mesaje

```
const int n /* number of processes */;  
void P(int i)  
{  
    message msg;  
    while (true)  
    {  
        receive (mutex, msg);  
        /* critical section */;  
        send (mutex, msg);  
        /* remainder */;  
    }  
}  
void main()  
{  
    create_mailbox (mutex);  
    send (mutex, null);  
    parbegin (P(1), P(2), . . . , P(n));  
}
```

Paradigme ale programării concurente

- Problema celor 5 filosofi
- Problema producător – consumator
- Problema cititori/scriitori
- Problema bărbierului

Problema celor 5 filosofi



- Resurse: furculite
- Procese: filosofi.

Problema celor 5 filosofi

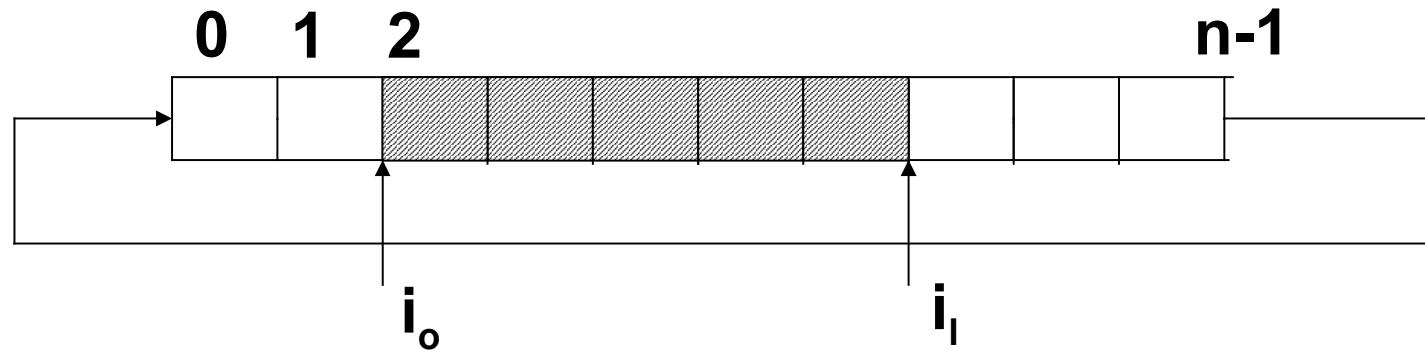
```
semaphore fork[5] = {1,1,1,1,1};  
int i;  
void philosopher(int i)  
{  
    while(true)  
    {  
        think( );  
        P(fork[i]);  
        P(fork[(i+1) mod 5]);  
        eat( );  
        V(fork[(i+1) mod 5]);  
        V(fork[i]);  
    }  
}
```

```
int fork[5] = {1,1,1,1,1};  
int valet = 4; int i ;  
void philosopher(int i)  
{  
    while (true)  
    {  
        P(valet); // - intră în cameră  
        P(fork[i]); // - ridică furculița stânga  
        P( fork[( i + 1 ) % 5]); // - ridică furculița  
        dreapta  
        // - mănâncă  
        V(fork[i]); // - eliberează furculița stânga  
        V( fork[( i + 1 ) % 5]); // - eliberează  
        furculița dreapta  
        V(valet); // - ieșe din cameră  
    }  
}
```

Problema producător – consumator

- implementează un protocol de comunicație între 2 sau mai multe procese care utilizează 2 sau mai multe resurse duale (locațiile ocupate sau libere dintr-un buffer):
- unul sau mai mulți producători introdu datele în buffer;
- un singur consumator extrage datele din buffer.

Problema producător – consumator buffer-ul circular cu n locații



□ Condiții statice:

- elementele din buffer sunt citite de consumator în aceeași ordine în care sunt scrise de producător
- nici un element nu trebuie pierdut sau introdus în plus.

□ Constante:

- i_o – locația unde este un mesaj care se poate citi;
- i_l – locația de la care putem scrie.

Problema producător – consumator

Condiții de sincronizare:

- dacă buffer-ul este gol consumatorul se blochează până când există cel puțin un mesaj.
- dacă buffer-ul este plin producatorul se blochează până când există cel puțin o locație liberă.

Problema producător – consumator

1 producător - 1 consumator

```
int i_o =i_l =0;  
int liber =n, ocupat =0;  
int buf[n];  
void producator (int mes) {  
    .....  
    P(liber);  
    buf [i_l] =mes; //adaugă mesaj  
    i_l = (i_l +1) %n;  
    V(ocupat);  
}  
  
int consumator()  
{  
    .....  
    P(ocupat);  
    mes=buf [i_o]; //citește mesaj  
    i_o = (i_o +1) %n;  
    V(liber);  
    .....  
    consuma(mes);  
}
```

„k” producători și „l” consumatori

- i_1 este resursă critică pentru producători
- se introduce un semafor **liber** pe care îl inițializăm cu **n**.
- i_0 resursă critică pentru consumatori
- se introduce semaforul **ocupat** inițializat cu 0.
- Semaforul **s** asigură accesul la secțiunea critică și este inițializat cu 1.

Problema producător – consumator „k” producători și „l” consumatori

```
int i_o = i_l=0;
int liber =n, ocupat =0, s =1;
void producator(int mes)
{
    .....
    P(liber);
    P(s);
    buf[i_l] =mes; // adaugă mesaj
    i_l =(i_l +1) %n;
    V(s);
    V(ocupat);
    .....
}

int consumator()
{
    .....
    P(ocupat);
    P(s);
    mes=buf[i_o]; //citește mesaj
    i_o =(i_o +1) %n;
    V(s);
    V(liber);
    consuma(mes);
}
```

Problema producător – consumator buffer infinit

- un semafor **s** pentru a realiza excluderea mutuală asupra buffer-ului;
- un semafor **n** pentru a sincroniza producătorul și consumatorul asupra dimensiunii buffer-ului

Problema producător – consumator buffer infinit

```
int n = 0, s = 1; //semafoare
int i_o = i_l =0;
void producător ()
{
    while (true)
    {
        produce();
        P(s);
        buf[i_l] =mes; // adaugă mesaj
        i_l =(i_l +1) %n;
        V(s);
        V(n);
    }
}

void consumator()
{
    while (true)
    {
        P(n);
        P(s);
        mes=buf[i_o]; //citește mesaj
        i_o =(i_o +1) %n;
        V(s);
        consume();
    }
}
```

Sisteme de Operare



- Paradigme ale programării concurente
- Elemente de blocaj:
 - Alocarea resurselor
 - Tratarea blocajelor

Problema cititori/scriitori soluția cu prioritatea cititorilor asupra scriitorilor

```
nt nrcit =0;  
semafor s, scriere;  
s =scriere =1 ;  
cititor()  
{
```

```
.....  
P(s);  
nrcit =nrcit +1;  
if(nrcit ==1) P(scriere);  
V(s);  
.....//lucrul cu resurse
```

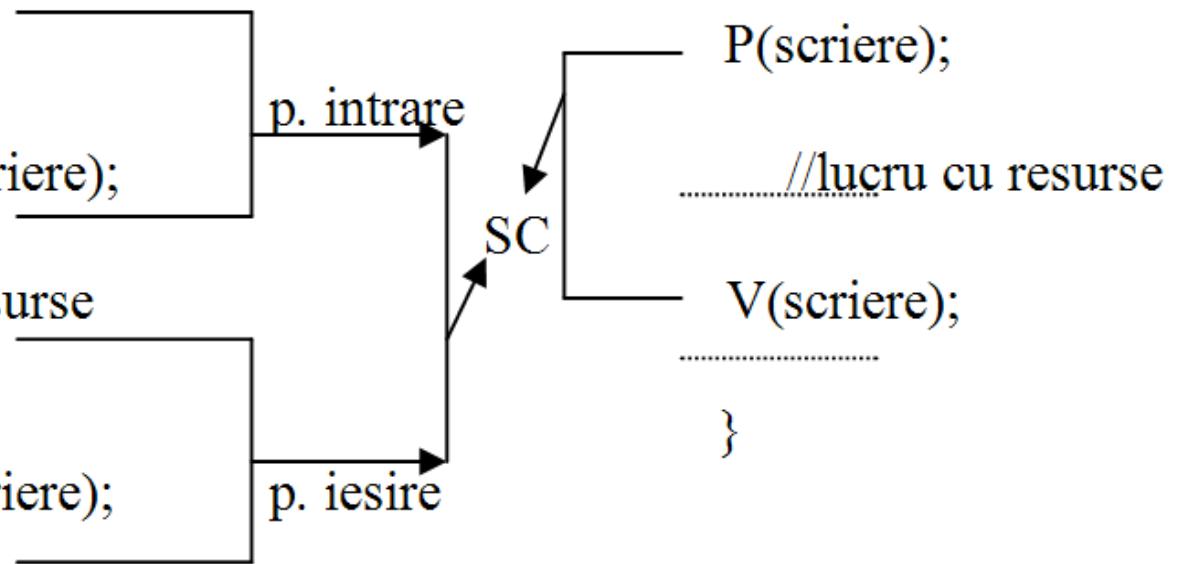
```
P(s);  
nrcit =nrcit - 1;  
if(nrcit ==0) V(scriere);  
V(s);  
.....
```

```
scriitor()  
{
```

```
.....  
P(scriere);  
.....//lucru cu resurse
```

```
V(scriere);  
.....  
}
```

```
}
```



Problema cititori/scriitori soluția cu prioritatea scriitorilor asupra cititorilor

```
int nrcit =nrscr =0;  
semafor s, s1, s2, citire, scriere;  
s =s1 =s2 =citire =scriere =1 ;  
  
    cititor_i()  
    {  
  
        .....  
        P(s2);  
        .....  
        P(citire);  
        P(s);  
        nrcit =nrcit +1;  
        if(nrcit ==1) P(scriere);  
        V(s);  
        V(citire);  
        V(s2);  
        ..... //lucrul cu resursa  
        P(s);  
        nrcit =nrcit -1;  
        if(nrcit==0) V(scriere);  
        V(s);  
    }  
  
    scriitor_j()  
    {  
  
        .....  
        P(s1);  
        nrscr =nrscr +1;  
        if(nrscr ==1) P(citire);  
        V(s1);  
        ..... //lucrul cu resursa  
        P(scriere);  
        .....  
        V(scriere);  
        P(s1);  
        nrscr =nrscr -1;  
        if(nrscr ==0) V(citire);  
        V(s1);  
        .....  
    }
```

Problema cititori/scriitori soluția cu prioritatea scriitorilor asupra cititorilor

- Este garantat faptul că în momentul execuției de către primul cititor a unui **V(citire)**, în sirul de așteptare a variabilei **citire** se află cel mult procese cititori (dintre care primul va fi blocat).
- Accesul la semaforul **citire** trebuie făcut într-o secțiune critică între **P(s2)** și **V(s2)**. Astfel următorii cititori se vor bloca pe **s2** și nu pe **citire**.
- Semaforul **s2** asigură execuția unui singur cititor în secțiunea critică dată de semaforul **citire**.
- Citire asigură blocarea cititorului dacă există un scriitor blocat pe **P(citire)**
- semaforul **s**, asigură excluderea la prelucrarea variabilei comune **nrcit** și este folosit numai când sunt procese care citesc.

Problema cititori/scriitori soluția cu prioritatea scriitorilor asupra cititorilor

- implementare prin transmitere de mesaje
 - Dacă Count > 0 nici un scriitor nu așteaptă; putem avea mai mulți cititori;
 - Dacă Count = 0 avem un scriitor care așteaptă;
 - Dacă Count < 0 avem cereri de scriere care așteaptă să se termine citirile.

Problema cititori/scriitori soluția cu prioritatea scriitorilor asupra cititorilor

```
void reader(int i)
{
    message rmsg;
    while (true) {
        rmsg = i;
        send (readrequest, rmsg);
        receive (mbox[i], rmsg);
        READUNIT ();
        rmsg = i;
        send (finished, rmsg);
    }
}
void writer(int j)
{
    message rmsg;
    while(true) {
        rmsg = j;
        send (writerequest, rmsg);
        receive (mbox[j], rmsg);
        WRITEUNIT ();
        rmsg = j;
        send (finished, rmsg);
    }
}
```

```
void controller()
{
    while (true)
    {
        if (count > 0) {
            if (!empty (finished)) {
                receive (finished, msg);
                count++;
            }
        } else if (!empty (writerequest)) {
            receive (writerequest, msg);
            writer_id = msg.id;
            count = count - 100;
        } else if (!empty (readrequest)) {
            receive (readrequest, msg);
            count--;
            send (msg.id, "OK");
        }
        if (count == 0) {
            send (writer_id, "OK");
            receive (finished, msg);
            count = 100;
        }
        while (count < 0) {
            receive (finished, msg);
            count++;
        }
    }
}
```

□ Sursa: William Stallings - Operating Systems: Internals and Design Principles, 6th Edition, Prentice Hall, 2008

Problema bărbierului

- Un bărbier are o sală de aşteptare cu n scaune. Dacă nu are clienți bărbierul doarme. Dacă un client care intră nu găsește loc să se așeze pleacă, iar dacă este liber un scaun se așează pe el. Atât timp cât există cel puțin un client așezat pe scaunul bărbierului, bărbierul lucrează.

Problema bărbierului

```
#define CHAIR 5 //scaune pentru clienți
semaphore client=0;
// numărul de clienți care așteaptă să fie servizi
semaphore bărbier=1;
//numărul de bărbieri care așteaptă clienți
semaphore mutex=1;
// semafor pt excluderea mutuală
int waiting=0; // clienții așteaptă să fie servizi
void barber( )
{
    while(true){
        P(client); // daca numărul de clienți este 0
        bărbierul doarme
        P(mutex); // acces la variabila waiting
        waiting --; //scade numărul de clienți care
        așteaptă
        V(mutex); //eliberare scaun = iau clientul
        Bărbierește( );
        V(bărbier); //bărbier gata de lucru
    }
}
```

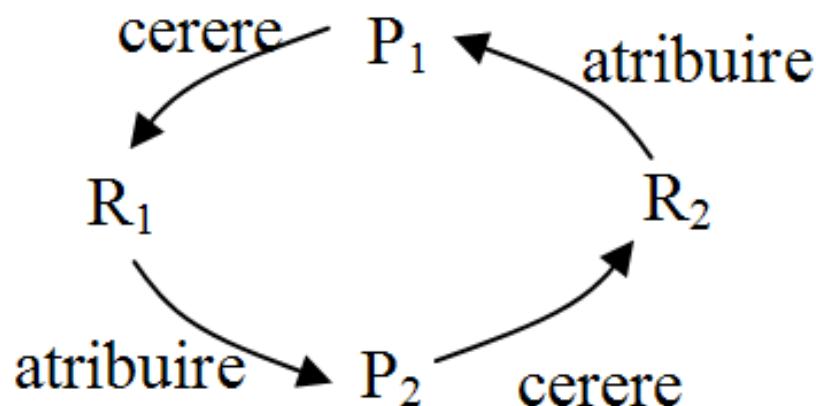
```
void client( )
{
    P(mutex); //intrare în regiunea critică
    if (waiting< CHAIR){
        // dacă nu sunt scaune libere pleacă
        waiting++; // crește numărul clienților
        V(client);
        // dacă este cazul trezește bărbierul
        V(mutex);
        //eliberează accesul pentru un nou client ce
        poate intra
        P(bărbier);
        //așteaptă să se elibereze un bărbier
        este_bărbierit( ); //este servit
    }else{
        V(mutex); //nu sunt scaune libere
    }
}
```

Elemente de blocaj

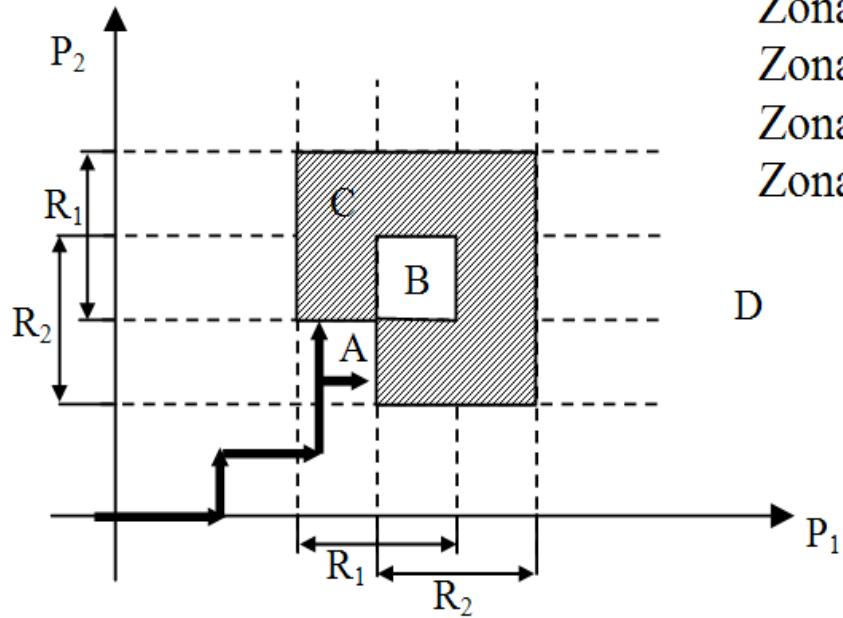
- Se numește **blocaj** situația în care o resursă cerută de un proces este menținută în starea ocupat de către alt proces aflat la rândul lui în aşteptarea eliberării unei resurse. Etapele parcuse de un proces pentru obținerea unei resurse sunt:
 - **cerere de acces** – dacă cererea nu este satisfăcută imediat, procesul este nevoit să aștepte;
 - **utilizare** – procesul poate folosi resursa;
 - **eliberare** – procesul eliberează resursa.
- Resursele pot fi:
 - **reutilizabile** (utilizate de un proces și apoi eliberate pentru a putea fi utilizate de alte procese – timp CPU, canale I/O, memoria principală și virtuală , fișiere, baze de date, semafoare)
 - **consumabile** (întreruperi, semnale, mesaje, informații din buffer-ele de I/O).

Elemente de blocaj

- Se spune că un set de procese se află în starea de **interblocare** atunci când oricare proces din set se află în aşteptarea unui eveniment ce poate fi produs numai de către un alt proces din setul respectiv.



Elemente de blocaj

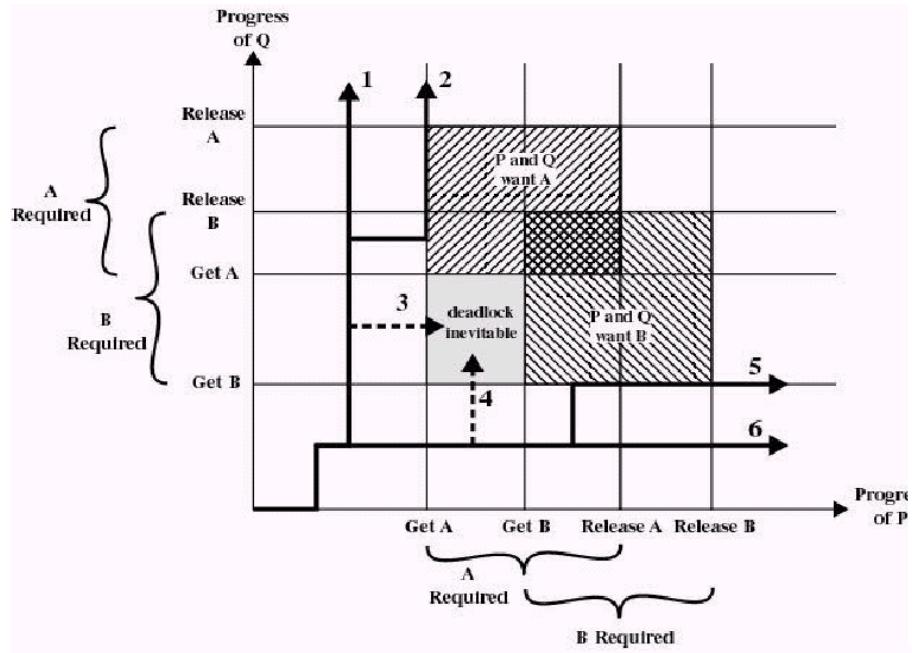


Zona A este zonă de *potențial blocaj* (*zonă nesigură*) ;
Zona B este o zonă ce nu poate fi atinsă ;
Zona C este *zonă de blocaj* dacă se vine din zona A ;
Zona D este *zonă liberă* de blocaj.

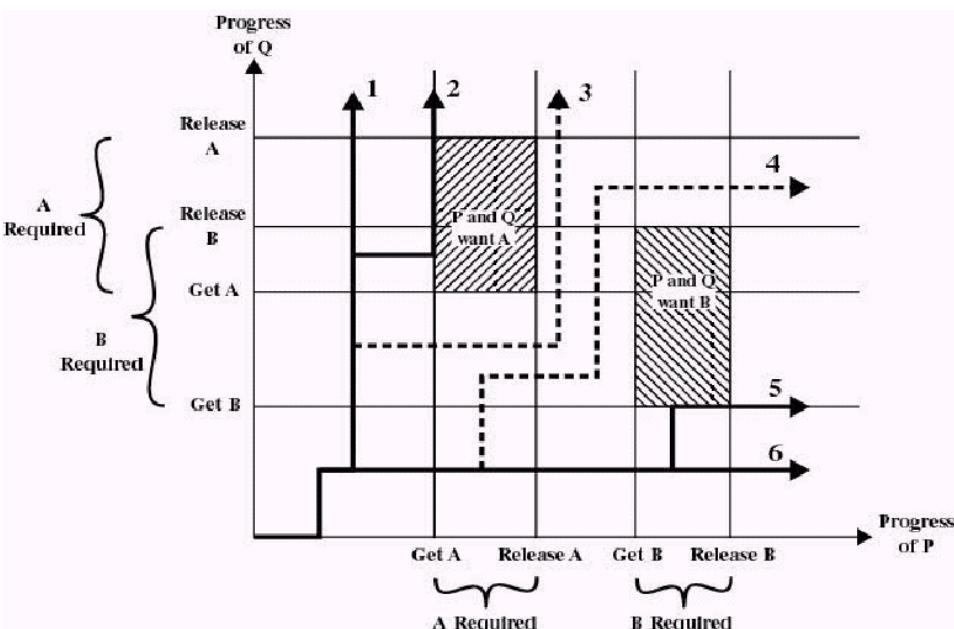
Elemente de blocaj

- Interblockarea apare în sistem dacă și numai dacă sunt îndeplinite simultan următoarele condiții:
 - **excluderea mutuală**: există cel puțin o resursă ocupată în mod exclusiv de către un proces ;
 - **ocupare și aşteptare**: există cel puțin un proces care menține ocupată cel puțin o resursă critică și aşteaptă să obțină resurse suplimentare ocupate în acel moment de către alte procese ;
 - **imposibilitatea achiziționării forțate**: resursele nu pot fi achiziționate forțat de către un proces de la procesul care le ocupă în momentul respectiv și sunt eliberate numai de către procesele care le ocupă după terminarea sarcinilor ;
 - **așteptare circulară**.

Elemente de blocaj



Rularea proceselor P și Q cu blocaj



Rularea proceselor P și Q fără blocaj

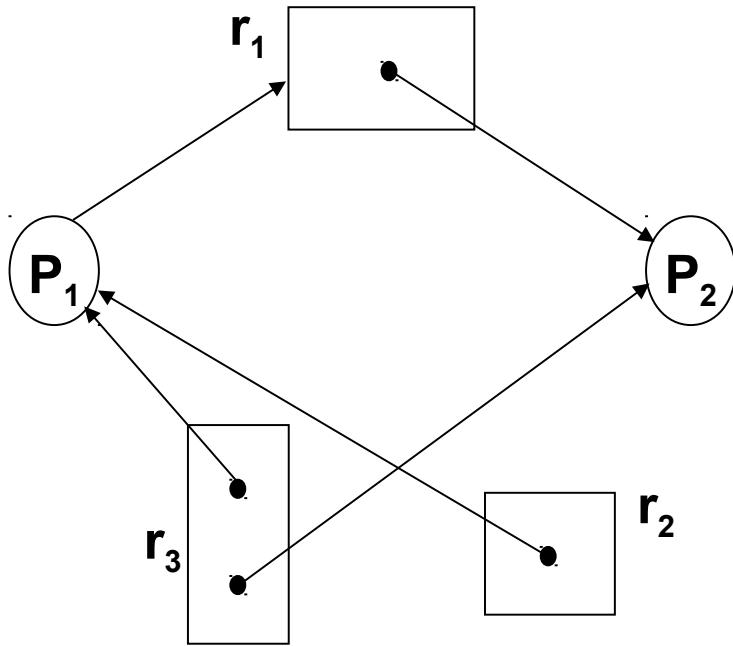
Graful de alocare a resurselor

- **Definiție:** Starea de interblocare poate fi descrisă prin folosirea unui graf orientat numit graf de alocare a resurselor sistemului. Aceasta este format dintr-o pereche $\mathbf{G}=(\mathbf{N}, \mathbf{A})$ unde \mathbf{N} reprezintă un set de noduri și \mathbf{A} un set de arce.
- Setul de noduri conține două multimi:
 - $\mathbf{P} = (\mathbf{p}_1, \mathbf{p}_2, \dots \mathbf{p}_n)$ – setul care conține toate procesele din sistem și
 - $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \dots \mathbf{r}_n)$ – setul care conține toate tipurile de resurse din sistem.

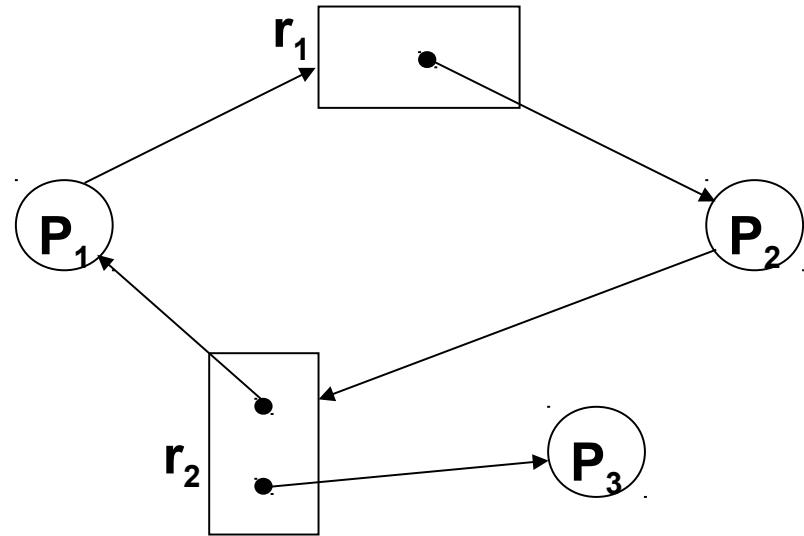
Graful de alocare a resurselor

- Fiecare element din setul **A** de arce reprezintă o pereche ordonată (p_i, r_j) sau (r_j, p_i) în care p_i este un proces din **P** iar r_j este o resursă din **R**.
- Dacă $(p_i, r_j) \in A$ atunci există un arc orientat de la procesul p_i la tipul de resursă r_j , ceea ce înseamnă că procesul p_i a formulat o cerere pentru un element al tipului de resursă r_j și așteaptă obținerea ei.
- Dacă $(r_j, p_i) \in A$ atunci există un arc orientat de la tipul de resursă r_j la procesul p_i , ceea ce înseamnă că procesului p_i i-a fost alocat un element al tipului de resursă r_j .
- Un arc de tipul (p_i, r_j) se numește arc cerere și (r_j, p_i) se numește arc de alocare.

Graful de alocare a resurselor



Graf de alocare a resurselor fără interblocare



Graf de alocare a resurselor cu interblocare

Graful de alocare a resurselor

- Conform definiției anterioare, **dacă graful nu conține bucle atunci în sistem nu există interblocare**. Dacă apare o singură buclă, interblocarea poate să apară.
- **Dacă fiecare tip de resursă este format dintr-un singur element atunci existența unei bucle în cadrul grafului arată că în sistem a apărut o interblocare**, fiecare proces implicat fiind în această stare.
- **Dacă fiecare tip de resursă conține mai multe elemente atunci existența unei bucle în cadrul grafului nu implică în mod necesar apariția interblocării** (este o condiție necesară nu și suficientă).

Graful de alocare a resurselor

- O stare se numește **sigură** dacă resursele pot fi alocate proceselor (fie căruia în parte) până la nivelul cererii într-o ordine oarecare și cu evitarea blocării.
- O mulțime de procese este într-o stare sigură dacă există o **secvență sigură**.
- O secvență de procese $\langle p_1, p_2, \dots, p_n \rangle$ se numește **sigură** pentru starea de alocare dacă pentru fiecare p_i pot fi atribuite resursele cerute din cele disponibile plus cele disponibilizate de p_j cu $j < i$.
- Dacă resursele de care are nevoie p_i nu sunt imediat disponibile toate, atunci p_i așteaptă până se termină p_j anterior sau o parte din ei, după care poate folosi resursele lor. Dacă nu există o astfel de secvență, starea respectivă de alocare este **nesigură**.

Metode de tratare a blocajelor

Prevenirea blocajelor

- ❑ sistemul trebuie proiectat în aşa fel încât să excludă posibilitatea blocării (este o soluție foarte restrictivă ce limitează accesul la resurse și impune restricții proceselor).
- ❑ Prevenirea blocajelor se poate realiza prin:
 - înlăturarea excluderii mutuale (soluție nerealistă);
 - înlăturarea stării de tip Hold-and-Wait :
 - la crearea proceselor să se realizeze cererile pentru toate resursele necesare;
 - blocarea proceselor până când toate cererile de acces pot fi satisfăcute simultan;
 - procesele să poată aștepta un timp îndelungat eliberarea resurselor;
 - resursele alocate unui proces ce pot rămâne un timp îndelungat nefolosite să poată fi utilizate de alte procese
 - înlăturarea planificării nepreemptive:
 - dacă unui proces îi sunt blocate unele cereri, atunci el să elibereze resursele alocate;
 - dacă un proces cere o resursă care este alocată altui proces, SO poate preempta al doilea proces și să îi ceară să elibereze resursa;
 - înlăturarea așteptării circulare:
 - definirea unei ordonări liniare a resurselor (o prioritate)
 - odată ce o resursă a fost obținută, numai resursele care urmează din listă pot fi obținute

Metode de tratare a blocajelor

Evitarea blocajelor

- ❑ Permiterea apariției a trei dintre condițiile de blocare, dar trebuie asigurat faptul că nu se ajunge niciodată în starea de blocaj;
- ❑ O decizie este luată dinamic dacă alocarea curentă a resurselor poate duce la o situație de blocaj dacă este permisă – este necesară cunoașterea posibilelor cereri ce pot apare.
- ❑ Se poate realiza dacă:
 - Nu se permite startarea unui proces dacă cererile de resurse pot duce la blocaj
 - Nu se permit cererile successive de resurse ale unui proces dacă alocarea lor poate duce la blocaj

Metode de tratare a blocajelor

Detectarea blocajelor

- cererile de resurse sunt permise ori de câte ori este posibil; periodic, SO rulează algoritmi de detectie a blocajelor.
- Strategii de rezolvarea situațiilor în care se detectează blocaje:
 - sunt operte toate procesele blocate;
 - se salvează starea proceselor blocate la un moment anterior apariției blocajului (checkpoint) și se restartează procesele (este posibil să se ajungă din nou la blocaj)
 - procesele sunt operte succesiv până când se iese din starea de blocaj
 - alocarea preemptivă a resurselor până la dispariția blocajului
- Criterii de selecție a proceselor blocate:
 - cel mai puțin timp procesor consumat până în prezent;
 - cele mai puține rezultate produse până în prezent;
 - cel mai mult timp de rulare rămas estimat;
 - cele mai puține resurse alocate din totalul celor cerute până în prezent;
 - cea mai mică prioritate.

Metode de tratare a blocajelor

Revenirea din blocaje

- oprirea tuturor proceselor blocate
- se salvează starea proceselor blocate la un moment anterior apariției blocajului (checkpoint) și se restartează procesele:
 - se presupune că avem puncte de control a rulării și mecanisme pentru restartarea proceselor din acele puncte de control;
 - este posibil să se ajungă din nou la blocaj (se presupune că dacă avem suficient timp, blocajul nu va reapărea)
- procesele sunt opriate succesiv până când se iese din starea de blocaj
- alocarea preemptivă a resurselor până la disparația blocajului
- restartarea proceselor de la un punct anterior obținerii resurselor
- achiziționarea forțată a resurselor de la anumite procese și alocarea lor altor procese până la eliminarea blocajului:
 - se alege un proces "victimă" (deadlock victim) de la care vor fi achiziționate resursele, urmărindu-se asigurarea unui "cost" minim – ce poate include numărul resurselor ocupate și mărimea duratei de execuție consumate deja de către acesta;
 - se reia execuția procesului de la care a fost achiziționată resursa (procesul este "întors în timp" până ajunge într-o stare sigură și pornind de la această stare se reia execuția);
 - "infometarea": dacă în alegerea "victimei" sistemul se bazează în principal pe factorul cost, este posibil ca de fiecare dată să fie desemnat același proces ca "victimă", astfel că el nu va putea niciodată să-și încheie normal execuția. Pentru evitarea unor astfel de situații este necesar să se impună o limitare a numărului de alegeri ca "victimă", de exemplu prin includerea în cost a numărului de întoarceri în timp.

Metode de tratare a blocajelor

Alte soluții

- Gruparea resurselor în clase, cu o strategie de evitare a blocajului diferită pentru fiecare clasă de resurse:
 - **Memoria virtuală**: se previn situațiile de blocaj prin impunerea alocării spațiului de memorie virtuală necesar o singură dată;
 - **Sistemul de fișiere**: strategiile de evitare a blocajului pot fi implementate cu succes, prevenirea blocajului prin ordonarea resurselor este de asemenea posibil;
 - **Memoria centrală**: strategiile preemptive sunt foarte utile în acest caz;
 - **Resursele interne**: - resursele utilizate de către sistem (blocul de control al procesului)

Metode de tratare a blocajelor

Alte solutii (2)

- Ordonarea claselor de mai sus este folosind în cadrul fiecărei clase următoarele abordări:
 - prevenirea interblocării prin ordonarea resurselor interne (în timpul execuției nu este necesară alegerea uneia dintre cererile nerezolvate);
 - prevenirea interblocării prin achiziție forțată a memoriei centrale (se poate evacua oricând un proces în memoria virtuală);
 - evitarea interblocării în cazul resurselor procesului (informațiile necesare despre formularea cererilor de resurse pot fi obținute din liniile de comandă)
 - alocarea prealabilă a spațiului din memoria virtuală asociat fiecărui proces utilizator (în general se cunoaște necesarul maxim de memorie al fiecărui proces).

Metode de tratare a blocajelor

Algoritmul bancherilor

- ❑ Fiecare nou proces apărut în sistem trebuie să declare numărul maxim de elemente din fiecare tip de resursă care i-ar putea fi necesar, număr ce nu poate depăși numărul total de resurse din sistem.
- ❑ În momentul în care un proces formulează o cerere pentru un set de resurse, trebuie să se verifice dacă va lăsa sistemul într-o stare sigură, caz în care operația este permisă.
- ❑ Altfel, procesul trebuie să aștepte până când vor fi eliberate de către alte procese suficient de multe resurse care să satisfacă și cererea sa.

Algoritmul bancherilor

- ◻ structuri de date pentru codificarea stării de alocare a resurselor sistemului:
- ◻ **m** – numărul maxim de instanțe pentru fiecare resursă (o resursă poate avea mai multe copii);
- ◻ **n** – numărul proceselor;
- ◻ **Disponibile[k]**; – un vector de dimensiune **m** care indică numărul de resurse disponibile aparținând fiecărui tip de resursă. Dacă **Disponibile[j] = k** înseamnă că din resursa **j** avem disponibile **k** copii;
- ◻ **Max[i][j]; i=1÷n, j=1÷m**; matrice de dimensiune **n x m** care indică numărul maxim de cereri ce pot fi formulate de către fiecare proces; dacă **Max[i][j] = k** înseamnă că procesul **p_i** poate cere cel mult **k** elemente de tip resursă **r_j**;
- ◻ **Alocate[i][j]; i=1÷n, j=1÷m**; matrice de dimensiune **n x m** care indică numărul de resurse din fiecare tip care sunt alocate în mod curent fiecărui proces; dacă **Alocate[i][j] = k** înseamnă că procesul **p_i** are alocate **k** elemente de tip resursă **r_j**;
- ◻ **Necesare[i][j]; i=1÷n, j=1÷m**; matrice de dimensiune **n x m** care indică numărul de resurse ce ar mai putea fi necesare fiecărui proces; dacă **Necesare[i][j] = k** înseamnă că procesul **p_i** ar mai avea nevoie în plus de **k** elemente din resursa **r_j** pentru a se termina; **Necesare = Max - Alocate**;

Algoritmul bancherilor

Algoritm de verificare a siguranței sistemului (de verificare a stării):

Temp[m];

Terminate[n];

Pas 1: *Temp =Disponibile; Terminate[i] = 0, $\forall i$;*

Pas 2: Găsește un i dacă există, astfel încât:

a) *Terminate[i] =0;*

b) *Temp \geq Necesare[i];*

//linia din Necesare[i] corespunzătoare componentei i

Dacă nu există un astfel de i **goto Pas 4.**

Pas 3: *Temp =Temp + Alocate[i];*

//linia din Alocate[i][j] corespunzătoare componentei i

*Terminate[i] =1; **goto Pas 2.***

Pas 4: Dacă *Terminate[i] =1, $\forall i$,* atunci sistemul este într-o stare sigură.

Algoritmul bancherilor

Algoritm de verificare a cererii

- Dacă **Cerere_i[j] = k** , procesul **p_i** are nevoie de **k** elemente din tipul de resursă **r_j**.
- Cînd **p_i** realizează o cerere de resurse, vor fi parcuse următoarele etape:
 - Pas 1:** Dacă $Cerere_i \leq Necesare_i$, **goto Pas 2**; altfel eroare (procesul a depășit limita maxim admisă);
 - Pas 2:** Dacă $Cerere_i \leq Disponibile_i$, **goto Pas 3**; altfel procesul **p_i** este nevoit să aștepte (resursele nu sunt disponibile);
 - Pas 3:** Se simulează alocarea resurselor cerute de procesul **p_i** modificând starea de alocare a resurselor astfel:
 $Disponibile_i = Disponibile_i - Cerere_i;$
 $Alocate_i = Alocate_i + Cerere_i;$
 $Necesare_i = Necesare_i - Cerere_i;$
- Dacă starea de alocare a resurselor rezultată este sigură, se alocă procesului **p_i** resursele cerute.
- Dacă noua stare este nesigură, procesul **p_i** trebuie să aștepte, iar sistemul reface starea de alocare a resurselor existentă înainte de execuția **Pas 3**.

Algoritmul bancherilor

Exemplu:

- Avem 5 procese: p_0, p_1, p_2, p_3, p_4 și 3 resurse $A=10, B=5, C=7$;

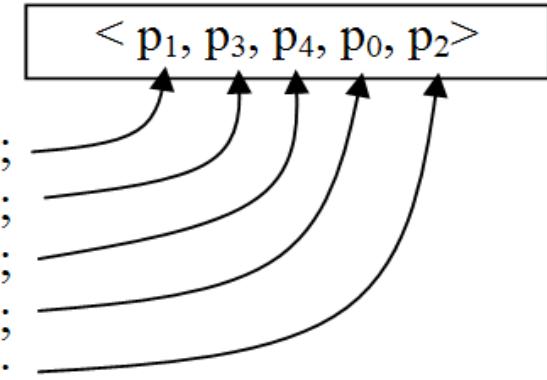
Nr. crt	Alocate	Max	Disponibile ABC	Necesare ABC
p_0	0 1 0	7 5 3		7 4 3
p_1	2 0 0	3 2 2		1 2 2
p_2	3 0 2	9 0 2	3 3 2	6 0 0
p_3	2 1 1	2 2 2		0 1 1
p_4	0 0 2	4 3 3		4 3 1

Algoritmul bancherilor

Exemplu:

Trebuie să găsim o secvență de forma $\langle p_1, p_2, \dots, p_n \rangle$

- 1) $Temp = (3, 3, 2); Terminate[1] = 0; i=1, Terminate[1] = 1;$
- 2) $Temp = (5, 3, 2); Terminate[3] = 0; i=3, Terminate[3] = 1;$
- 3) $Temp = (7, 4, 3); Terminate[4] = 0; i=4, Terminate[4] = 1;$
- 4) $Temp = (7, 4, 5); Terminate[0] = 0; i=0, Terminate[0] = 1;$
- 5) $Temp = (7, 5, 5); Terminate[2] = 0; i=2, Terminate[2] = 1;$



- Starea curentă este o stare sigură deoarece există o secvență de alocare (de satisfacere a necesarului) conform algoritmului verificare a siguranței sistemului:
 - $\langle p_1, p_3, p_4, p_0, p_2 \rangle$
 - Condiție: starea respectivă este sigură.
- Presupunem că Cerere₁ = (1,0,2);
- Se pune problema dacă cererea poate fi satisfăcută sau nu. Verificăm condițiile de la pașii 1 și 2 din algoritmul de verificare a cererii.

Algoritmul bancherilor

Exemplu:

	Alocate ABC	Disponibile ABC	Necesare ABC
p_0	0 1 0	2 3 0	7 4 3
p_1	3 0 2		0 2 0
p_2	3 0 2		6 0 0
p_3	2 1 1		0 1 1
p_4	2 0 2		4 3 1

- Dacă se execută din nou algoritmul de verificare a siguranței sistemului se găsește secvența: $\langle p_1, p_3, p_4, p_0, p_2 \rangle$.
- Dacă în noua stare vom avea următoarele cereri:
 - Cerere₄ = (3,3,0) sau Cerere₀ = (0,2,0),
 - amândouă vor duce în stări nesigure și nu vor putea fi satisfăcute (condițiile din algoritmul de verificare a cererii vor fi satisfăcute numai pentru Cerere₀).

Algoritmul bancherilor

Algoritm pentru detecția blocajului

Pas 1: $\text{Temp} = \text{Disponibile};$

$\text{for}(i = 1; i \leq n, i++)$

$\text{if}(Alocate}_i \neq 0)$

$\text{Terminate}[i] = 0;$

else

$\text{Terminate}[i] = 1;$

Pas 2: Găsește i astfel încât:

a) $\text{Temp} \geq \text{Cerere}_i$

b) $\text{Terminate}[i] = 0$

Dacă nu există i **goto Pas 4.**

Pas 3: $\text{Temp} = \text{Temp} + \text{Alocate}_i$

$\text{Terminate}[i] = 1;$

goto Pas 2;

Pas 4: Dacă există i astfel încât $\text{Terminate}[i] = 0$, atunci starea respectiva este o stare de blocaj, deci procesul i este blocat.

Algoritm bancherilor

Algoritm pentru detecția blocajului

Exemplu:

- procese și resursele A =7; B=2; C =6

	Alocate ABC	Cerere ABC	Disponibile
p ₀	0 1 0	0 0 0	0 0 0
p ₁	2 0 0	2 0 2	
p ₂	3 0 3	0 0 0	
p ₃	2 1 1	1 0 0	
p ₄	0 0 2	0 0 2	

Algoritmul bancherilor

Algoritm pentru detecția blocajului

- Executându-se algoritmul se gaseste secvența: $\langle p_0, p_2, p_3, p_1, p_4 \rangle$ astfel încât $Terminate[i] = 1, \forall i$, deci starea respectivă nu este în blocaj.
- Presupunem că lucrăm cu o altă cerere $Cerere_2 = (0,0,1)$
- Rulându-se algoritmul observăm că starea respectivă este o stare de blocaj.
- Singurul proces care poate rula este p_0 , în rest pentru toate celelalte procese vom avea $Terminate[i] = 0$.

	Alocate ABC	Cerere ABC	Disponibile
p_0	0 1 0	0 0 0	0 0 0
p_1	2 0 0	2 0 2	
p_2	3 0 3	0 0 1	
p_3	2 1 1	1 0 0	
p_4	0 0 2	0 0 2	

Algoritmul bancherilor

- ❑ Când trebuie apelat algoritmul de detectie?
 - Răspunsul depinde de doi factori:
 - ❑ cât de frecvent apare starea de blocaj
 - ❑ câte procese sunt afectate de blocaj.
 - Dacă blocajul apare frecvent, atunci algoritmul trebuie utilizat foarte des.
 - Resursele alocate proceselor blocate rămân inițializate până se iese din starea de blocaj.

Sisteme de Operare

□ Planificarea proceselor

- Funcționarea unui planificator
- Implementarea unui planificator
- Algoritmi de planificare

Planificarea proceselor

- ❑ Planificarea este funcția principală a sistemelor de operare. Aproape toate resursele unui sistem de calcul sunt planificate înaintea utilizării
- ❑ Datorită naturii unor constrângeri relative la consumul de resurse, problema planificării se reduce la găsirea unui algoritm eficient pentru gestionarea accesului și utilizarea resurselor pe baza unei metriki de performanță.
- ❑ Tipurile de resurse luate în calcul sunt timpul CPU și capacitatea memoriei.
- ❑ Clasificare:
 - planificare uniprocesor (task-uri cu resurse independente sau partajate)
 - planificare multiprocesor:
 - ❑ planificare statică
 - ❑ planificare dinamică

Planificarea proceselor

- problema planificării uniprocesor poate fi privită ca o problemă de căutare:
 - avem n procese $\langle p_1, p_2, \dots, p_n \rangle$ și vrem să găsim o secvență de execuție astfel încât toate procesele să se poată executa.
- Alocarea proceselor este diferită de planificare:
 - alocarea se referă la task-uri cu resurse independente, deci nu se realizează predevență între ele
 - planificarea se referă la procese care au resurse partajate deci și relații de predevență.

Planificarea proceselor

□ Planificarea statică:

- există un set de procese $\langle p_1, p_2, \dots p_n \rangle$.
- Se obține o planificare a proceselor pe sistemul multiprocesor dat în aşa numita fază de testare a fezabilității planificării, procesele se vor executa exact în ordinea stabilită de această planificare fără ca în timpul rulării proceselor să apară elemente necunoscute despre procese.

□ Planificarea dinamică:

- se pot varia caracteristicile (constrângerile) proceselor odată cu apariția unor noi procese.

Deciziile de planificare

1. Când un proces trece din starea running în starea waiting (cerere I/O, crearea unui proces fiu sau aşteptarea terminării acestuia);
 2. Când un proces trece din starea running în starea ready (când apare o intrerupere);
 3. Când un proces trece din starea waiting în starea ready (terminarea unei operații I/O);
 4. Când un proces este terminat.
- Doar în cazurile 1 și 4, atunci planificarea este **nepreemptivă**;
 - altfel, planificarea este **preemptivă**.

Planificarea nepreemptivă

- ❑ odată ce procesorul a fost alocat unui proces, procesul păstrează CPU până când se termină sau până când trece într-o stare de wait
- ❑ este folosită în cazul MSDOS, Windows 3.11
- ❑ nu necesită existența unui timer

Planificarea preemptivă

- ❑ necesită costuri suplimentare:
 - fie cazul a două procese A și B care partajează o dată;
 - ❑ procesul A poate fi în mijlocul operației de modificare a datei când este preemptat și procesul B rulează;
 - ❑ procesul B poate încerca să citească data care în momentul respectiv nu este consistentă.
 - ❑ În acest caz este necesară introducerea de mecanisme suplimentare pentru coordonarea accesului la resursa comună.

Planificarea preemptivă și kernelul

- are influență și asupra proiectării kernelului sistemului de operare.
 - În timpul procesării unui apel sistem, kernelul poate fi ocupat cu o activitate asupra comportamentului unui proces, ceea ce poate duce la modificarea unor date ale kernelului (cozile I/O).
 - Ce se poate întâmpla dacă procesul este preemptat în mijlocul acestor modificări și kernelul are nevoie să citească sau să modifice datele?

Planificarea preemptivă și kernelul

- Întreruperile pot interveni oricând
- nu pot fi tot timpul ignorate de kernel
- secțiunile de cod afectate de întreruperi trebuie protejate împotriva utilizării simultane (concurrente) de mai multe procese.
 - se inhibă (disable) tratarea întreruperilor la intrarea în secțiunea critică (utilizarea datelor) și se reactivează (enable) la ieșire .

Niveluri de planificare

- ❑ Planificarea pe termen lung
- ❑ Planificarea pe termen mediu
- ❑ Planificarea pe termen scurt

Planificarea pe termen lung

- ❑ are drept sarcină alegerea job-ului ce va fi executat, alocarea resurselor necesare și crearea proceselor.
- ❑ rulează cu frecvența cea mai mică și trebuie să separe tipurile de job-uri în funcție de solicitări.
- ❑ La unele sisteme poate avea un rol minim sau poate lipsi (la sistemele time-sharing)

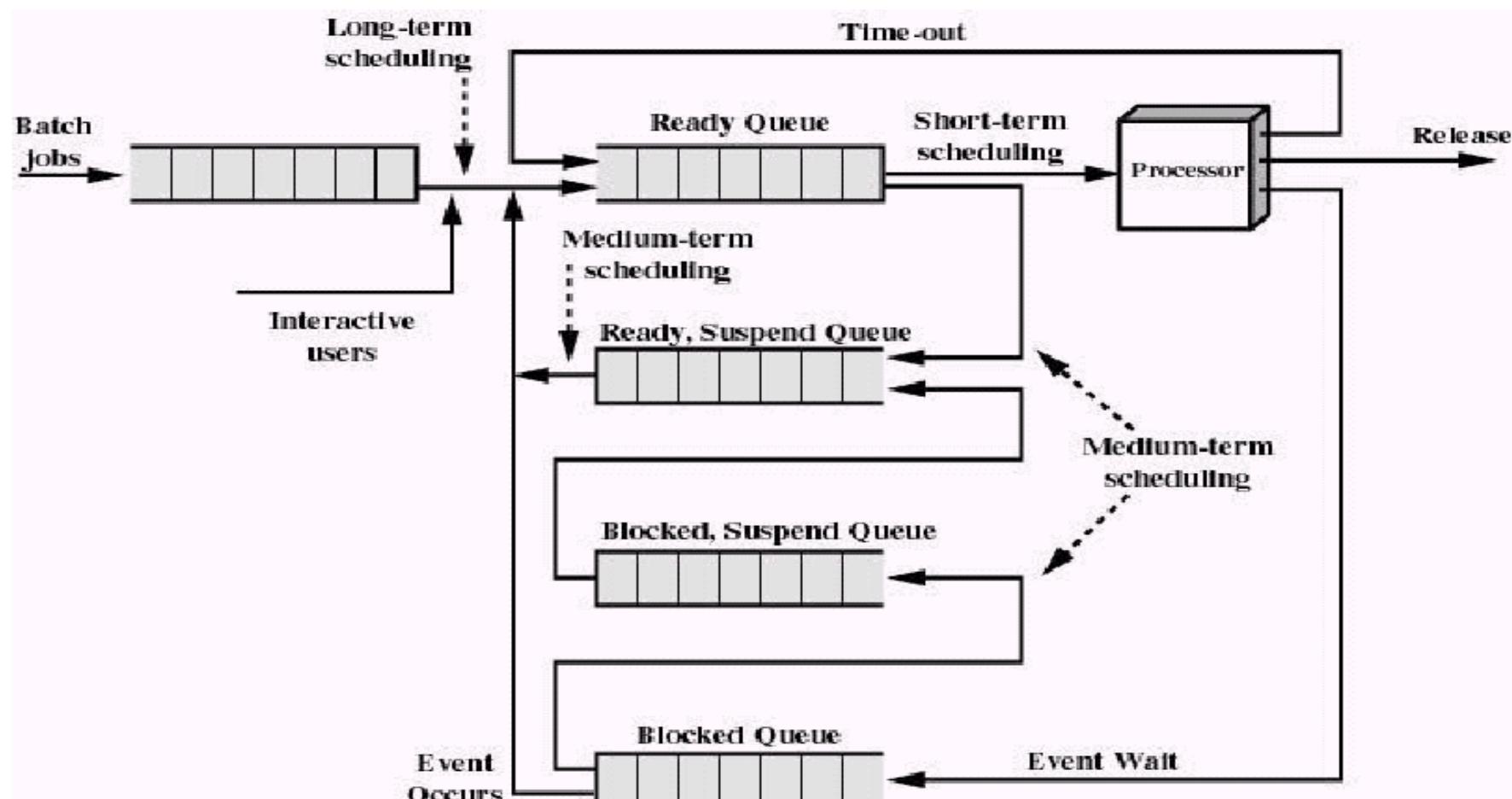
Planificarea pe termen mediu

- ❑ Planificatorul pe tremen mediu este cel care decide momentele în care se fac evaluări, procesele care se evacuează și care se readuc în memorie pentru continuarea execuției.
- ❑ Stabilitatea sistemelor de tip time-sharing depinde de resursele sistemului de calcul și de aceea la aceste sisteme se aplică tehnica de swapping prin care procesele sunt trecute din starea run în swap și din swap în ready.
- ❑ Prin aceste evacuări temporare, gradul de multiprogramare scade și se prelungesc durata execuției, dar se permite accesul simultan al mai multor utilizatori în sistem.

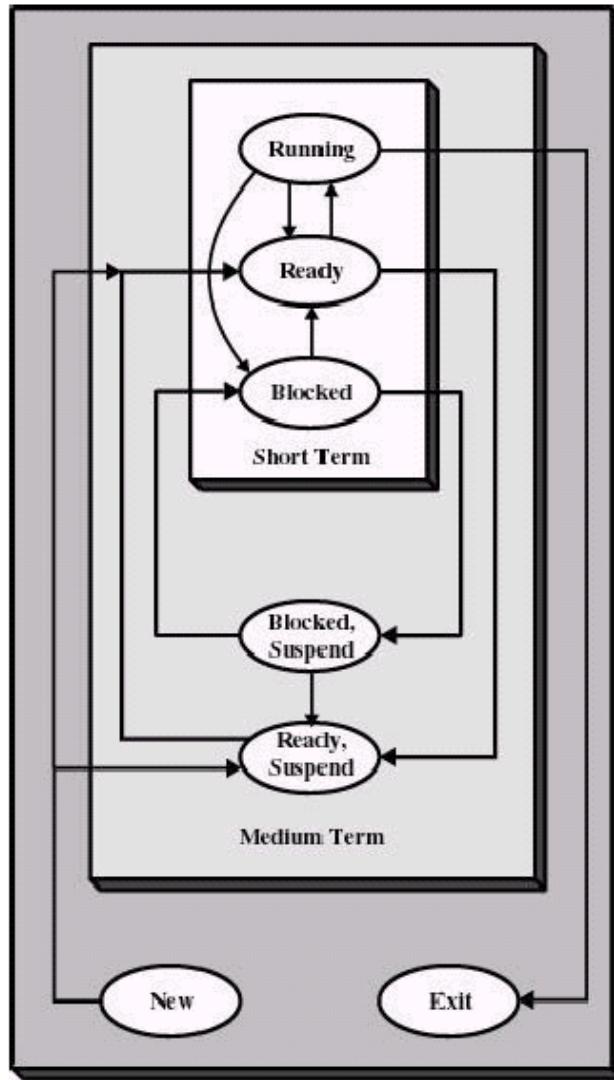
Planificarea pe termen scurt

- ❑ are în vedere trecerea alternativă a proceselor din starea **ready** în starea **run** și invers.
- ❑ Tot la acest nivel se desfășoară și activitatea dispecerului de procesoare.

Cozile de aşteptare ale planificatorului



Stările proceselor și tipul planificării



Planificare proceselor - Metrici

- **timpul de aşteptare** al unui proces: este timpul cât un proces aşteaptă în coada de execuție

$$T_{\text{wait}} = \frac{\sum_{i=1}^n t_{\text{wait}}(p_i)}{n}$$

Planificare proceselor - Metrici

- **timpul de ciclare** reprezintă timpul din momentul creării procesului (intrare în coada gata de execuție) până în momentul terminării execuției

$$t_{ciclare}(p_i) = t_{wait}(p_i) + \tau(p_i)$$

$\tau(p_i)$ - timpul de execuție

$T_{ciclare} = \frac{\sum_{i=1}^n t_{ciclare}(p_i)}{n}$ - timpul de ciclare mediu

Încărcarea CPU

$$\rho = \alpha \cdot \frac{1}{\beta}$$

- α - rata medie de sosire a noilor procese în coada de execuție
- β - rata medie de deservire a proceselor ($1/\beta$ este timpul mediu de execuție)
- Dacă $\rho < 1$ ($\alpha < \beta$) avem cazul în care se poate aplica un algoritm de planificare (încărcare normale) – stare stabilă.
- Dacă $\rho > 1$ ($\alpha > \beta$), atunci unitatea centrală va fi saturată indiferent de algoritmul de planificare.
- Dacă $\rho = 1$ ($\alpha = \beta$), lista de execuție nu este suficient de lungă.
- Algoritmii ce vor fi prezentati se vor referi la cazul $\rho < 1$ ($\alpha < \beta$).

Funcționarea unui planificator

- descrierea funcționării unui planificator trebuie avute în vedere două aspecte:
 - regulile de acțiune
 - implementarea în contextul sistemului de operare.
- Fixarea sarcinilor unui planificator, indiferent de nivelul la care acționează, se face precizând:
 - modalitatea de intervenție
 - funcția de prioritate
 - regula de arbitraj

Funcționarea unui planificator

Modalitatea de intervenție

- ❑ stabilește momentele în care planificatorul intră în acțiune
 - momentele impuse de proces
 - ❑ atunci când un proces își termină activitatea sau când așteaptă terminarea unor operații de I/O - planificatorul este partajat;
 - ❑ planificatorul este apelat de procese ca un subprogram
 - momentele impuse de SO
 - ❑ SO intervine indiferent de starea proceselor pe care le planifică – planificatorul este master.

Funcționarea unui planificator

Funcția de prioritate

- are ca argumente procesele și parametrii sistemului.
- Determinarea priorității se face având în vedere criterii cum ar fi:
 - cererea de memorie,
 - atingerea unui timp de servire de către CPU,
 - timpul real din sistem,
 - timpul total de servire,
 - valorile priorităților externe,
 - necesarul de timp rămas până la terminarea procesului etc.

Funcționarea unui planificator

Funcția de prioritate

- Algoritmii de planificare trebuie să îndeplinească următoarele criterii:
 - să realizeze scopurile de performanță pentru care au fost elaborați;
 - să aibă o durată foarte mică de execuție, pentru a nu crește în mod nejustificat timpul de execuție alocat SO.
- Există multe metode matematice de planificare care dau soluția optimă în niște restricții date
- Proiectanții de sisteme de operare preferă algoritmi euristică, mai simpli și cu rezultate mai mult sau mai puțin apropiate de cea optimă, deoarece un model sofisticat consumă mai mult timp făcând să crească timpul alocat SO, deci randamentul global să scadă.

Funcționarea unui planificator

Regula de arbitraj

- stabilește o ordine în caz de priorități egale:
 - servirea în ordine cronologică
 - servirea circulară sau aleatoare.
- Observație: Informațiile legate de starea proceselor care trebuie planificate sunt obținute din PCB-ul fiecărui proces.

Implementarea unui planificator

- ❑ Toate tipurile de planificatoare sunt implementate prin intermediul semafoarelor și folosesc facilitățile oferite de gestiunea memoriei.
- ❑ În funcție de tipul sistemului de operare, planificatoarele dețin module specializate de alocare și eliberare a resurselor, prevenire, detectare și ieșire din blocaje.
- ❑ Conceptele de multiprogramare și programare în timp real sunt implementate cu ajutorul planificării pe termen scurt.
- ❑ Tot la acest nivel are loc selectarea proceselor candidate la resursele disponibile.
- ❑ La nivel mediu este potrivit să se modifice prioritățile proceselor, dacă SO permite această facilitate.

Algoritmi de planificare

Criterii de evaluare

- ❑ Utilizarea CPU
- ❑ Răspunsul (throughput)
- ❑ Numărul de procese terminate în unitatea de timp
- ❑ Timpul de ciclare (turnaround) – timpul mediu de la sosirea unui proces până la terminarea lui
- ❑ Timp de așteptare (waiting time) în lista gata de execuție
- ❑ Timpul de răspuns (response time) – timpul mediu de la sosirea proceselor până la prima execuție
- ❑ Eficiența planificatorului = overhead-ul introdus de planificator.
- ❑ Reprezentarea proceselor se va face cu ajutorul diagramelor Gantt.

Algoritmi de planificare

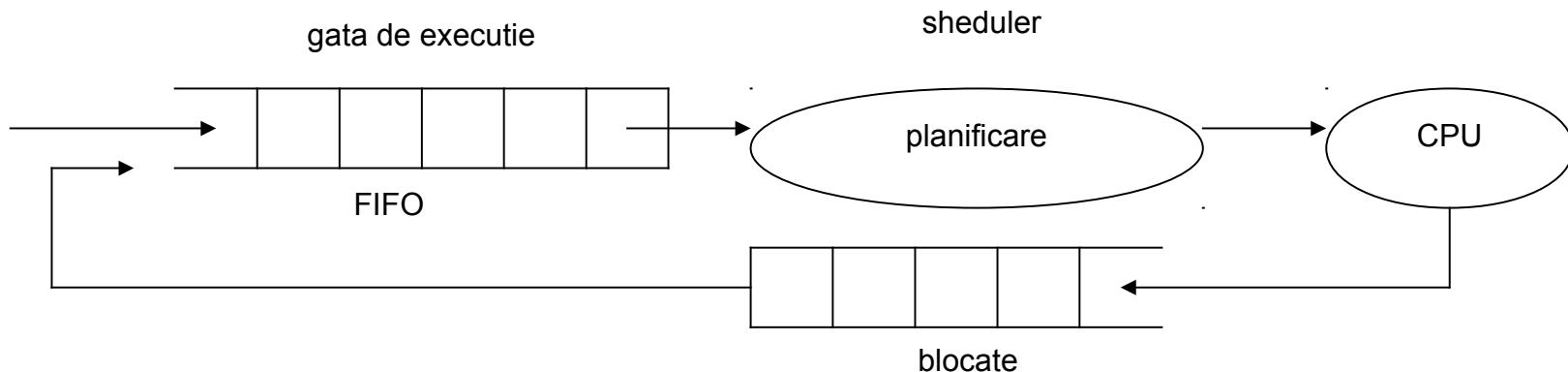
FCFS (First-Come, First-served)

- ❑ Procesele sunt planificate pe măsura sosirii lor în coada gata de execuție.
- ❑ Procesorul este alocat procesului care îl cere primul.
- ❑ Algoritmul mai este denumit și **FIFO**.
- ❑ Unitatea de măsură a performanței este timpul mediu de așteptare.
- ❑ În acest caz $\text{pri}(i) = t_i$, unde t_i este momentul când sosește în coada gata de execuție. Dacă pentru două procese i și j , avem $\text{pri}(i) < \text{pri}(j)$ atunci $t_i < t_j$.
- ❑ În general vom spune ca un proces este de prioritate cu atât mai mare cu cât valoarea $\text{pri}(i)$ este mai mică.

Algoritmi de planificare

FCFS (First-Come, First-served)

- Pentru FCFS planificarea se va face în funcție de prioritatea maximă. Lista gata de execuție va fi o listă FIFO.



Algoritmi de planificare

FCFS (First-Come, First-served)

Exemplu:

procese	temp de execuție τ	prioritate
0	40	4
1	20	2
2	50	1
3	30	3

Planificare:

0	40	60	110	140
P_0	P_1	P_2	P_3	

$$t_{\text{wait}}(0) = 0, \quad t_{\text{ciclare}}(0) = 40$$

$$t_{\text{wait}}(1) = 40, \quad t_{\text{ciclare}}(1) = 60$$

$$t_{\text{wait}}(2) = 60, \quad t_{\text{ciclare}}(2) = 110$$

$$t_{\text{wait}}(3) = 110, \quad t_{\text{ciclare}}(3) = 140$$

$$T_{\text{wait}} = (0 + 40 + 60 + 110) / 4 = 52,5$$

$$T_{\text{ciclare}} = (40 + 60 + 110 + 140) / 4 = 87,5$$

$t_{\text{out}} = 140 / 4 = 35 \Rightarrow$ în medie 35 de unități de timp pentru fiecare proces care trebuie să se execute.

Algoritmi de planificare

Shortest-Job-First (SJF) nepreemptiv

- Procesul planificat pentru execuție este procesul cu timpul de execuție cel mai mic. Acest algoritm se mai numește și **Shortest-Job-Next (SJN)**.

Pentru cazul anterior avem:

$$t_{\text{wait}}(1) = 0, \quad t_{\text{ciclare}}(1) = 20$$

$$t_{\text{wait}}(3) = 20, \quad t_{\text{ciclare}}(3) = 50$$

$$t_{\text{wait}}(0) = 50, \quad t_{\text{ciclare}}(0) = 90$$

$$t_{\text{wait}}(2) = 90, \quad t_{\text{ciclare}}(2) = 140$$

$$T_{\text{wait}} = (0 + 20 + 50 + 90) / 4 = 40$$

$$T_{\text{ciclare}} = (20 + 50 + 90 + 140) / 4 = 75$$

Planificare:

0	20	50	90	140
P ₁	P ₃	P ₀	P ₂	

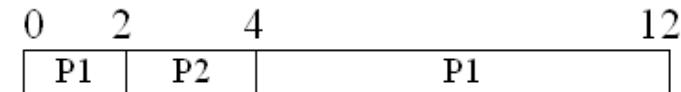
Observație: Timpul total este același dar timpul mediu de așteptare este mai mic. Acest algoritm este optim dacă toate job-urile ajung în același timp.

Algoritmi de planificare

Shortest-Job-First (SJF) preemptiv

- Procesul planificat pentru execuție este procesul cu timpul de execuție rămas cel mai mic. Acest algoritm mai este numit și **Shortest Remaining Time First**.

procese	Timp de execuție τ	prioritate	Timp sosire
P1	10	1	0
P2	2	2	2

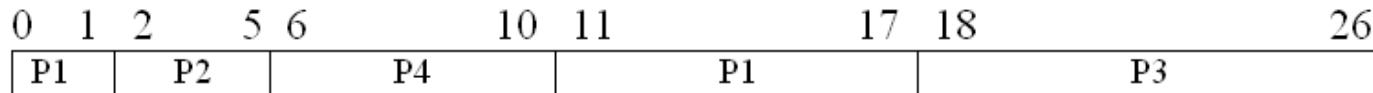


$$t_{\text{wait}}(1) = 4 - 2 = 2$$

$$t_{\text{wait}}(2) = 0$$

$$T_{\text{wait}} = (0 + 2) / 2 = 1$$

procese	Timp de execuție τ	Timp sosire
P1	8	0
P2	4	1
P3	9	2
P4	5	3

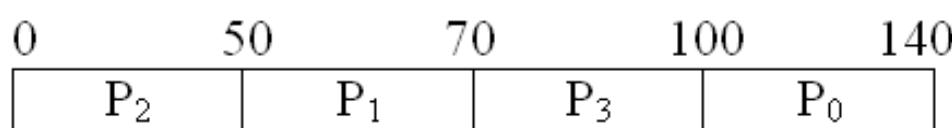


Algoritmi de planificare

PS (Priority Scheduling)

- ❑ Fiecare proces are asociată prioritate, fiind lansate în execuție de la prioritatea cea mai mică la prioritatea cea mai mare.
- ❑ Este cel mai folosit algoritm.

Planificare:



$$T_{\text{wait}} = (0 + 50 + 70 + 100) / 4 = 55$$

$$T_{\text{ciclare}} = (50 + 70 + 100 + 140) / 4 = 90$$

În acest caz, algoritmul funcționează cu *priorități statice*.

Algoritmi de planificare

PS (Priority Scheduling)

- Se poate introduce un mecanism de priorități dinamice, în care pe măsură ce timpul de așteptare crește, va crește și prioritatea.
- Această situație este întâlnită la Unix - se oprește periodic (la fiecare secundă) activitatea sistemului și se recalculează prioritatea fiecărui proces.
- Astfel se garantează un timp mediu de răspuns rezonabil pentru fiecare proces din sistem, dar nu se asigură răspuns prompt la o execuție secvențială.
- Prioritățile se stabilesc astfel:
 - job-ul primește prioritatea la intrarea în sistem și o păstrează până la sfârșit (este posibil să apară fenomenul de "înfometare", dacă apar multe procese cu prioritate mare);
 - SO calculează prioritățile după reguli proprii și le atașează dinamic proceselor în execuție. Această variantă este folosită la planificarea pe termen mediu.

Algoritmi de planificare

Round-Robin (RR)

- este un algoritm preemptiv destinat sistemelor de tip **time-sharing** și se bazează pe distribuirea în mod egal a timpului de procesare între procese.
- Este folosită o **cuantă de timp q** (cu valori între 10 și 100 milisecunde) pe durata căreia sunt executate pe rând părți din fiecare proces.
- Dacă se introduce și **timpul consumat c** prin schimbarea contextului, fiecare proces va primi de fapt **c+q** unități de timp.
- Unele procese se pot termina înainte de expirarea cuantei de timp, moment în care se invoca planificatorul care reface prioritățile, resetează cuanta de timp și replanifică procesele.
- Replanificarea are loc și la apariția unui proces nou.
- Algoritmul RR se implementează folosind întreruperea de ceas a sistemului respectiv.

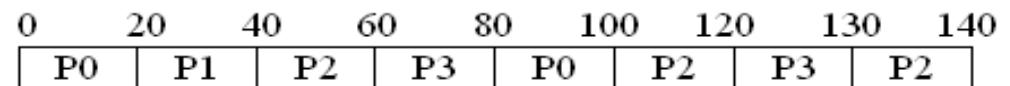
Algoritmi de planificare

Round-Robin (RR)

- Dacă cuanta de timp q este mai mare algoritmul tinde către FCFS.
- Algoritmul RR funcționează cel mai bine când 80% din procese au timpii de execuție mai mari decât cuanta q .

Fie $c = 0$ și $q = 20$

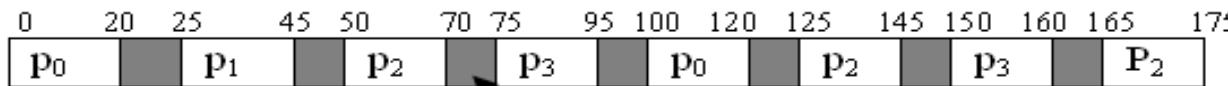
Procese	temp de execuție τ
0	40
1	20
2	50
3	30



$$t_{ciclare} = (100 + 40 + 140 + 130) / 4 = 102.5;$$

$$t_{wait} = (60 + 20 + 40 + 40 + 10 + 60 + 40) / 4 = 67.5;$$

Dacă consideram $c=5$, $q=20$ avem:



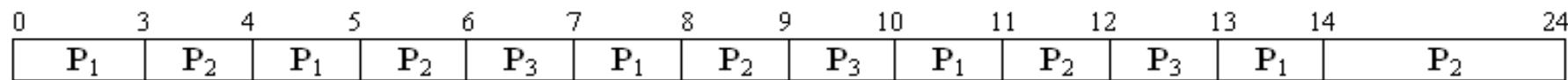
timpii pierduți cu schimbarea contextului

Algoritmi de planificare

Round-Robin (RR)

procese	Timp de execuție τ	Timp sosire
P1	7	0
P2	14	3
P3	3	6

Fie $c = 0$ și $q = 1$



Algoritmi de planificare

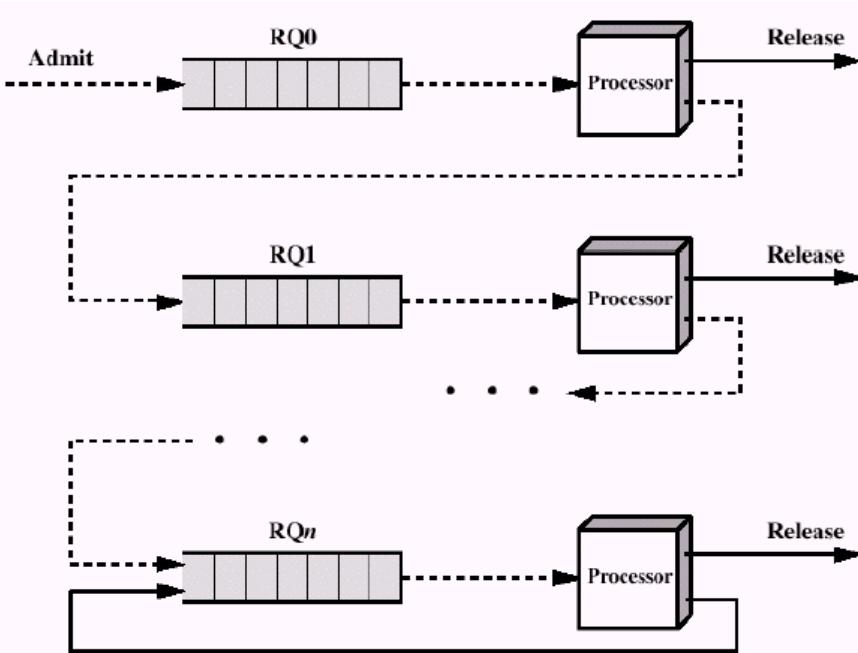
▫ **Highest Response Ratio Next (HRRN)**

- Este ales procesul cu cea mai mare rată de deservire (response ratio – rr):
- În general sunt favorizate procesele cu durata de execuție cea mai mică. Dacă așteptarea procesului crește, atunci crește și valoarea ratei de deservire.

$$rr = \frac{\text{time spent waiting} + \text{expected service time}}{\text{expected service time}}$$

Algoritmi de planificare

□ Algoritmul Feedback



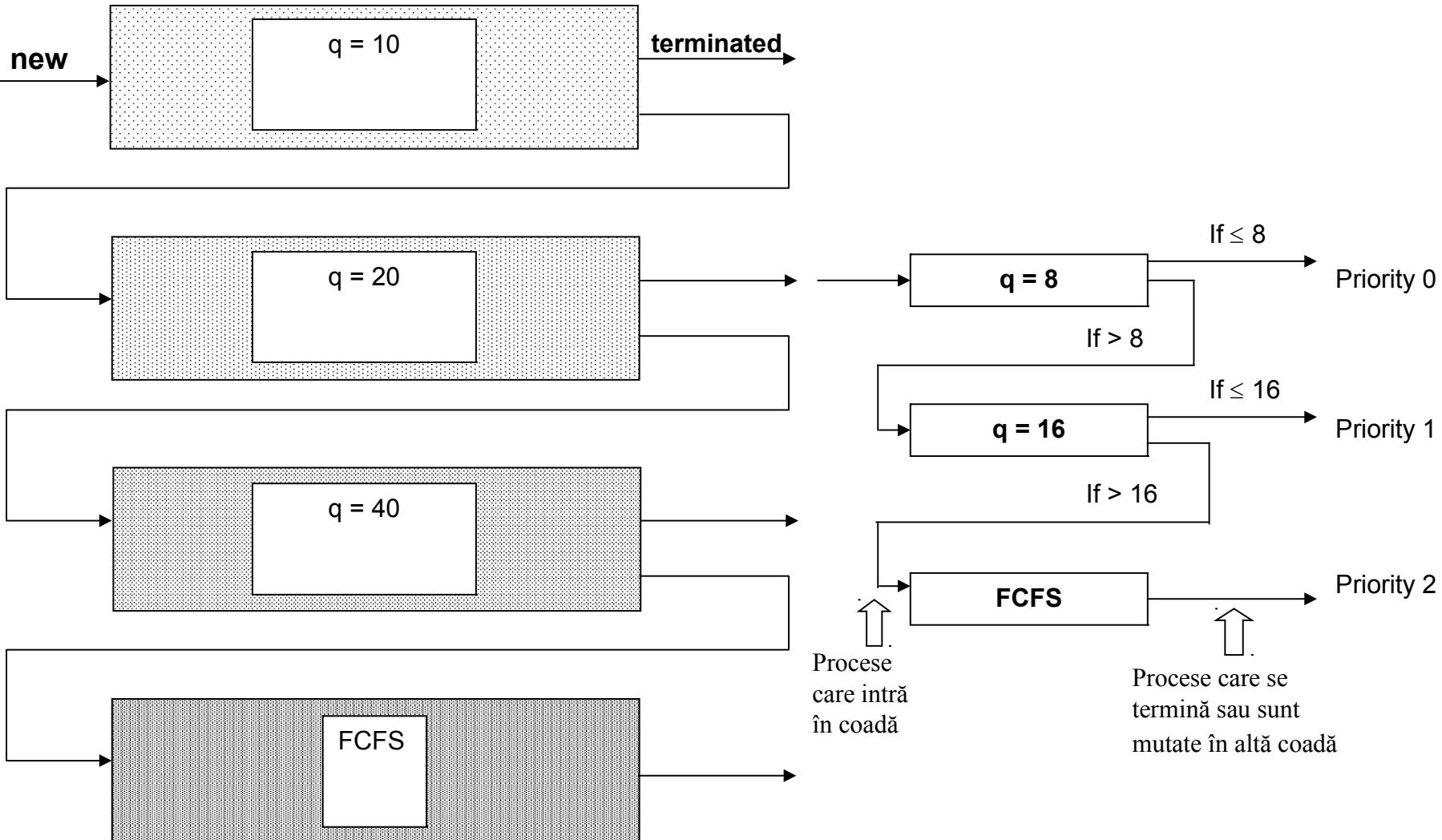
- este folosit atunci când nu se cunoaște timpul de care mai are nevoie un proces ca să-și termine execuția.
- Sunt penalizate procesele care rulează prea mult și poate duce la apariția fenomenului de "înfometare" (process/resource starvation) dacă nu variem algoritmii de planificare și prioritățile în funcție de cozile de așteptare.

Algoritmi de planificare

□ Planificarea cu liste multivel

- Este o combinație între:
 - algoritmii bazați pe priorități,
 - Round-Robin și
 - algoritmi folosiți pentru tratarea proceselor de aceeași prioritate (HRRN).
- Presupunem că lista gata de execuție este formată din **n** subliste în care procesele au prioritățile între **1** și **m**.
- În acest caz procesul **P_i**, din sublista **k** va avea prioritatea **k**.
- Pentru a înlătura dezavantajul unui timp de așteptare mare pentru subliste apropiate de **n** se poate folosi o schemă de planificare care să favorizeze subliste de mare prioritate (algoritmi nepreemptivi în liste și între liste algoritmi preemptivi).

Planificarea cu listele multinivel



Inversarea priorității

- apare atunci când un proces de prioritate scăzută acceseează o secțiune critică apoi un proces de prioritate mare acceseează și el SC respectivă și se blochează.
- procese de prioritate intermediară vor împiedica de asemenea primul proces (de prioritatea cea mai scăzută) să deblocheze secțiunea critică.

Prevenirea inversării priorității

- Se folosește “moștenirea priorității”:
 - de fiecare dată când un proces deține o SC pentru care așteaptă și alte procese i se acordă respectivului proces maximul priorității proceselor aflate în așteptare.
 - Problema este că moștenirea priorității micșorează eficiența algoritmului de planificare și crește overhead-ul (i se da prioritate maxima ca să nu fie preemptat și să termine lucrul cu SC).

Prevenirea inversării priorității

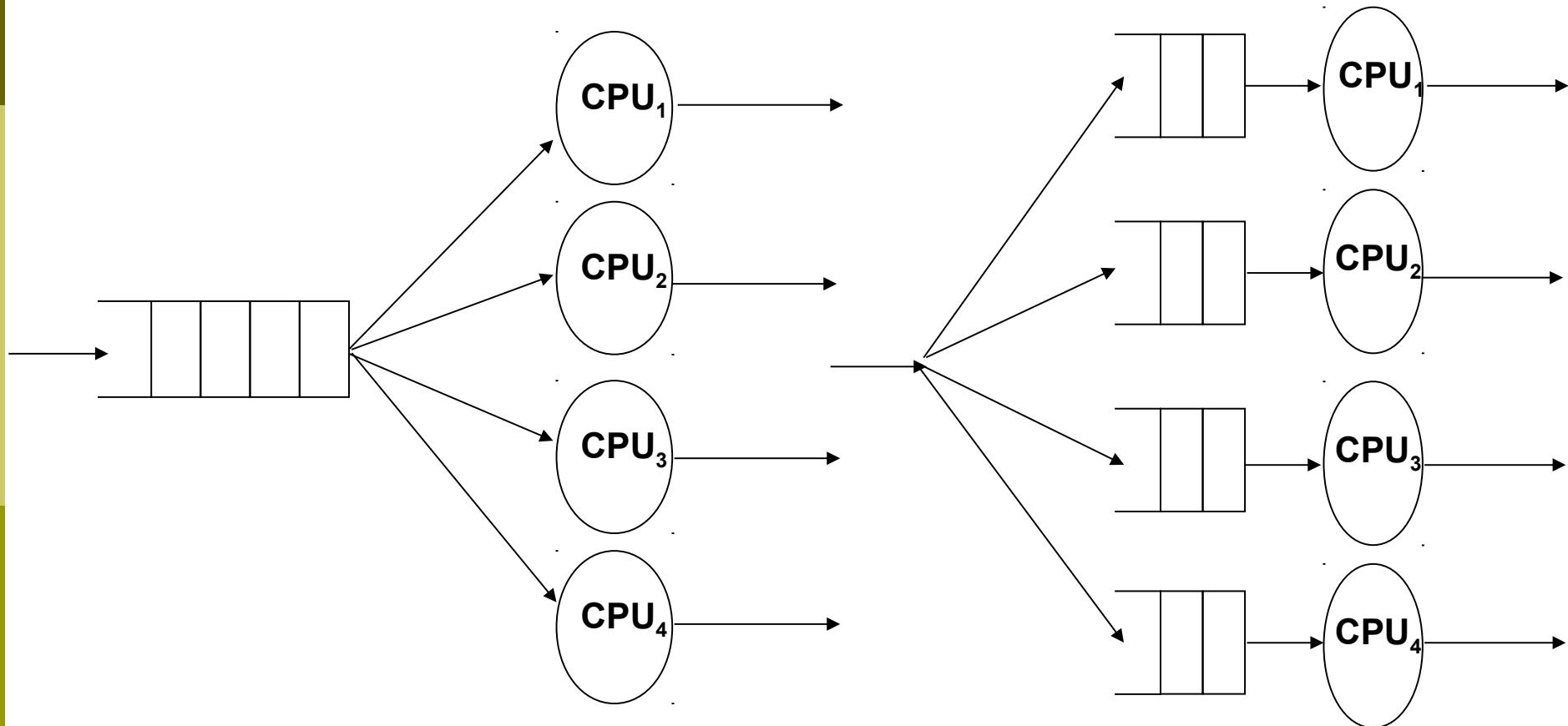
- Preemptia poate interacționa cu sincronizarea într-un context multiprocesor generând un alt efect nedorit numit **efectul de convoi**:
 - un proces accesează o secțiune critică după care se suspendă.
 - Alte procese care au nevoie de secțiunea critică vor trebui suspendate până când primul se va trezi (va fi reactivat) și va termina secțiunea critică.
 - În acel moment toate procesele suspendate din cauza sincronizării vor fi trezite încercând pe baza priorității să acceseze secțiunea critică.
 - Astfel, se creează efectul de convoi.
- În general, **efectul de convoi** apare când o mulțime de procese au nevoie de o resursă pentru un timp scurt, iar un altul deține resursa pentru un timp mult mai lung blocându-le pe primele.

Sisteme de Operare



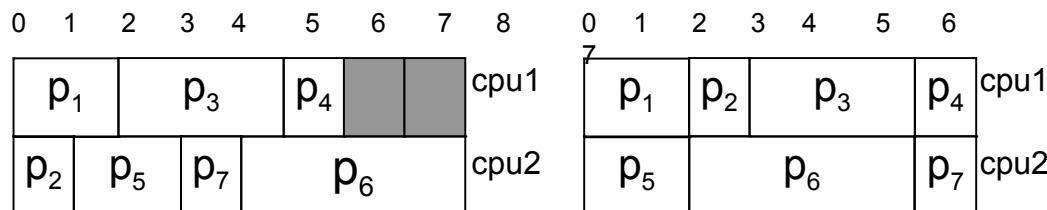
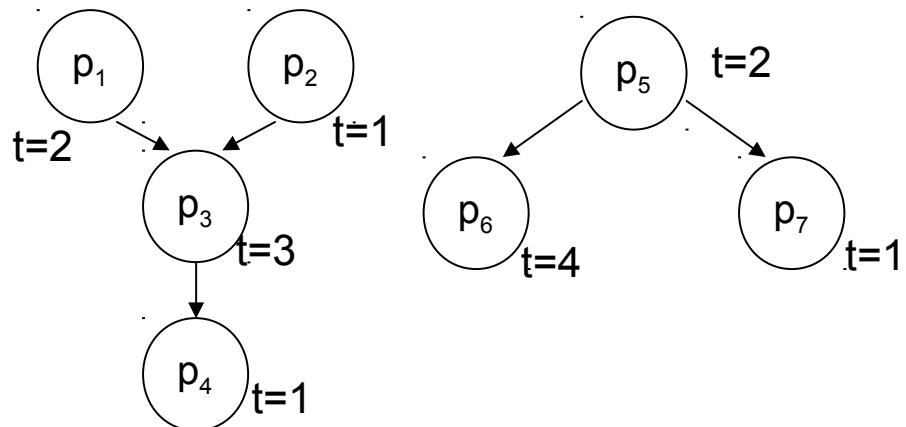
- Planificarea în sistemele multiprocesor
- Planificarea în sistemele de timp real
- Studii de caz
 - UNIX, Linux, Windows

Planificarea în sistemele multiprocesor



Echilibrarea Încărcării

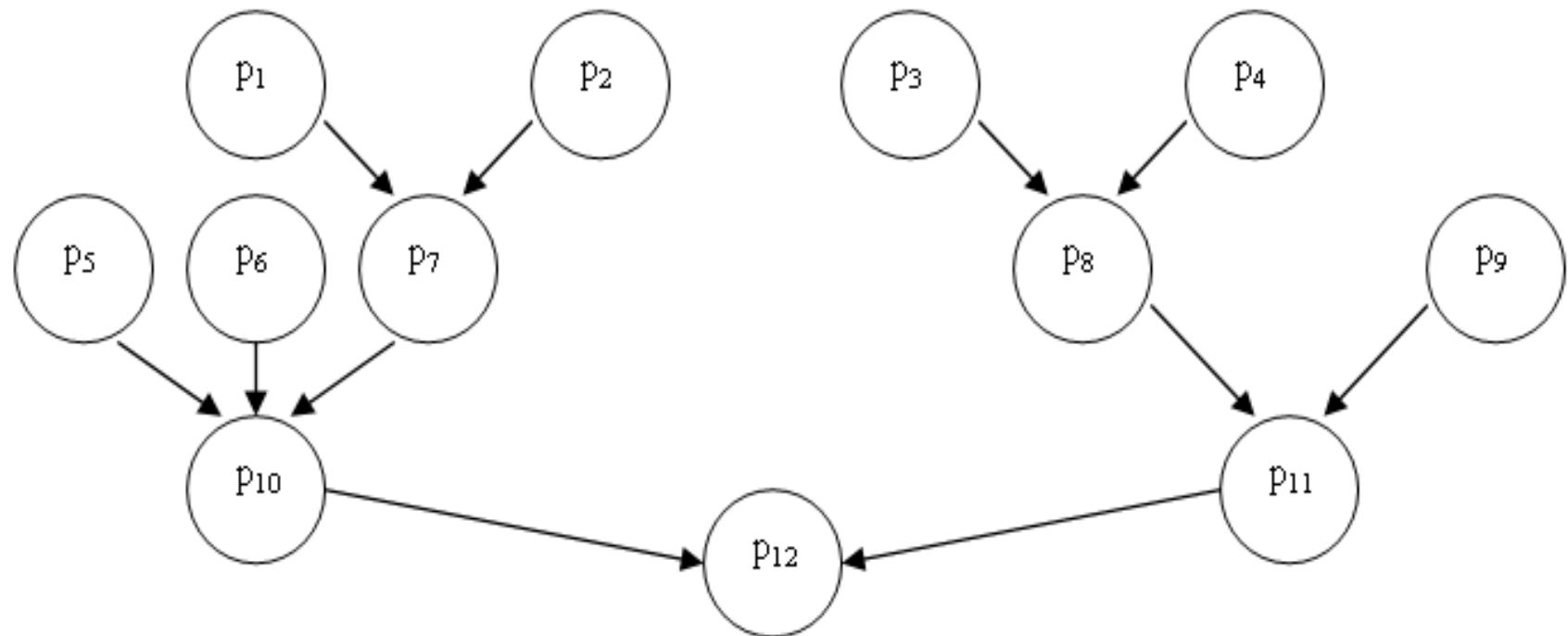
- În cazul planificării multiprocesor procesele din lista gata de execuție formează un graf de precedență.
- Obiectivul planificării multiprocesor este optimizarea planificării pentru $n \geq 2$ CPU pentru **procese cu grafuri de precedență cu timpi de execuție cunoscuți**.



Graful de precedență al proceselor

Algoritm de planificare

- **Pas 1:** se aşteaptă ca un proces să devină idle;
- **Pas 2:** se defineşte mulţimea \mathbf{R} a proceselor pentru care fiecare predecesor şi-a terminat execuţia;
- **Pas 3:** se alege un $\mathbf{R}' \subset \mathbf{R}$ şi se găsesc $\mathbf{p} \in \mathbf{R}'$ astfel încât **length(p)** este maxim;
- **Pas 4:** se alege un $\mathbf{p} \in \mathbf{R}'$ din cei găsiţi anterior şi se atribuie procesorului idle;
- **Pas 5:** dacă nu mai sunt procese de planificat algoritmul s-a terminat; altfel goto **Pas 1.**



Planificarea pentru 3 procesoare este:

	0	1	2	3	4	5
cpu1	p ₃	p ₂	p ₈	p ₁₀	p ₁₂	
cpu2	p ₄	p ₅	p ₆	p ₁₁		
cpu3	p ₁	p ₉	p ₇			

■ La momentul 0 :

procesele p₁, p₂, p₃, p₄ sunt fără predecesori și au length=3;
procesele p₅, p₆, p₉ au length=2.

Planificarea proceselor în sistemele de timp real

- Din punct de vedere al importanței deadline-ului sistemele de timp real se împart în:
 - **hard - real time**: task-urile trebuie terminate într-un anumit interval de timp.
 - **soft - real time**: sunt mai puțin restrictive și impun prioritatea proceselor critice asupra celorlalte procese din sistem

Planificarea în sistemele de timp real - hard-real time

- În general când un proces este lansat în execuție este specificat și necesarul de timp pentru realizarea operațiilor de I/O.
- Planificatorul poate accepta procesul, garantând că se va termina la timp sau îl poate respinge.
- Acest lucru se mai numește și rezervarea resurselor:
 - planificatorul știe exact cât va dura execuția fiecărei funcții sistem.
 - sunt construite pentru sisteme dedicate și nu au funcționalitatea unui sistem de uz general.

Planificarea în sistemele de timp real - soft-real time

- Adăugarea unei astfel de funcționalități unui sistem de tip time-sharing poate duce la:
 - o alocare defectuoasă a resurselor,
 - întârzieri mari în rularea proceselor
 - apariția fenomenului de "înfometare" pentru unele procese.
- Implementarea acestei funcționalități presupune o mare atenție în proiectarea planificatorului.
 - sistemul trebuie să aibă o planificare bazată pe priorități
 - procesele de timp real trebuie să aibă prioritatea cea mai mare, prioritate care nu trebuie să scadă în timp.
 - lansarea proceselor trebuie să fie rapidă.

Preemptarea apelurilor sistem

- introducerea punctelor de preemptare în apelurile sistem de lungă durată
 - trebuie verificat dacă sunt procese cu prioritate mai mare care trebuie rulate.
 - pot fi plasate numai în locații sigure ale kernelului (unde nu se fac modificări ale structurilor de date ale kernelului).
- Întreg kernelul să poată fi preemptat:
 - trebuie asigurat faptul că structurile de date ale kernelului sunt protejate prin utilizarea mecanismelor de sincronizare.
 - toate datele kernelului sunt protejate împotriva modificării lor de către procesele cu prioritate ridicată.

Clase de algoritmi de planificare

- ❑ Static table-driven
 - se încearcă realizarea întregii planificări (se determină când trebuie să fie executat un task).
- ❑ Static priority-driven preemptive
 - poate fi folosită planificarea bazată pe priorități.
- ❑ Dynamic planning-based
 - o soluție ar fi crearea unei planificări care să conțină taskurile planificate anterior precum și cele nou sosite.
- ❑ Dynamic best effort
 - când apare un proces nou, sistemul îi atribuie o prioritate bazată pe caracteristicile procesului.

Planificarea în funcție de deadline

- trebuie cunoscute diverse informații despre task:
 - ready time
 - starting deadline – timpul în care un proces trebuie lansat
 - timpul în care se termină procesul
 - timpul de rulare
 - cererile de resurse
 - prioritatea
 - structura procesului.

Studii de caz

UNIX

- Algoritmii de planificare sunt proiectați pentru a da un timp de răspuns bun proceselor utilizatorilor într-un sistem de tip time-sharing.
- Pentru SVR4, schemele de planificare includ și componente ale sistemelor de timp real.
- Sistemul de operare UNIX implementează o listă multinivel cu feedback
 - În cozile de prioritate este folosit algoritmul round-robin (RR) cu o cantă de 1 secundă.
 - Dacă un proces nu se termină sau nu se blochează într-o secundă este preemptat.
 - Prioritatea este în funcție de tipul procesului și informațiile din perioadele de rulare anterioare.

Studii de caz UNIX

$$CPU_j = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i-1)}{2} + nice_j$$

- ▣ **CPU_j(i)** = utilizarea procesorului de către procesul **j** în intervalul de timp **i**.
- ▣ **P_j(i)** = prioritatea procesului **j** la începutul intervalului **i**. Valorile mici implică o prioritate mai mare
- ▣ **Base_j** = prioritate de bază a procesului **j**.
- ▣ **nice_j** = valoare la dispoziția utilizatorului.

Studii de caz

UNIX

- Prioritatea fiecărui proces este recalculată la fiecare secundă sau de fiecare dată când apare un proces nou.
- Scopul lui **Base_j** este împărțirea proceselor la cozile de prioritate.
- **CPU_{j(i)}** și **nice_j** sunt folosite pentru a preveni migrarea proceselor dintr-o coadă în alte cozi de prioritate.
- Aceste **cozi de prioritate** sunt folosite pentru a optimiza accesul la dispozitivele de tip bloc (disk) și permit sistemului de operare să răspundă rapid la apelurile sistem.

Studii de caz

UNIX

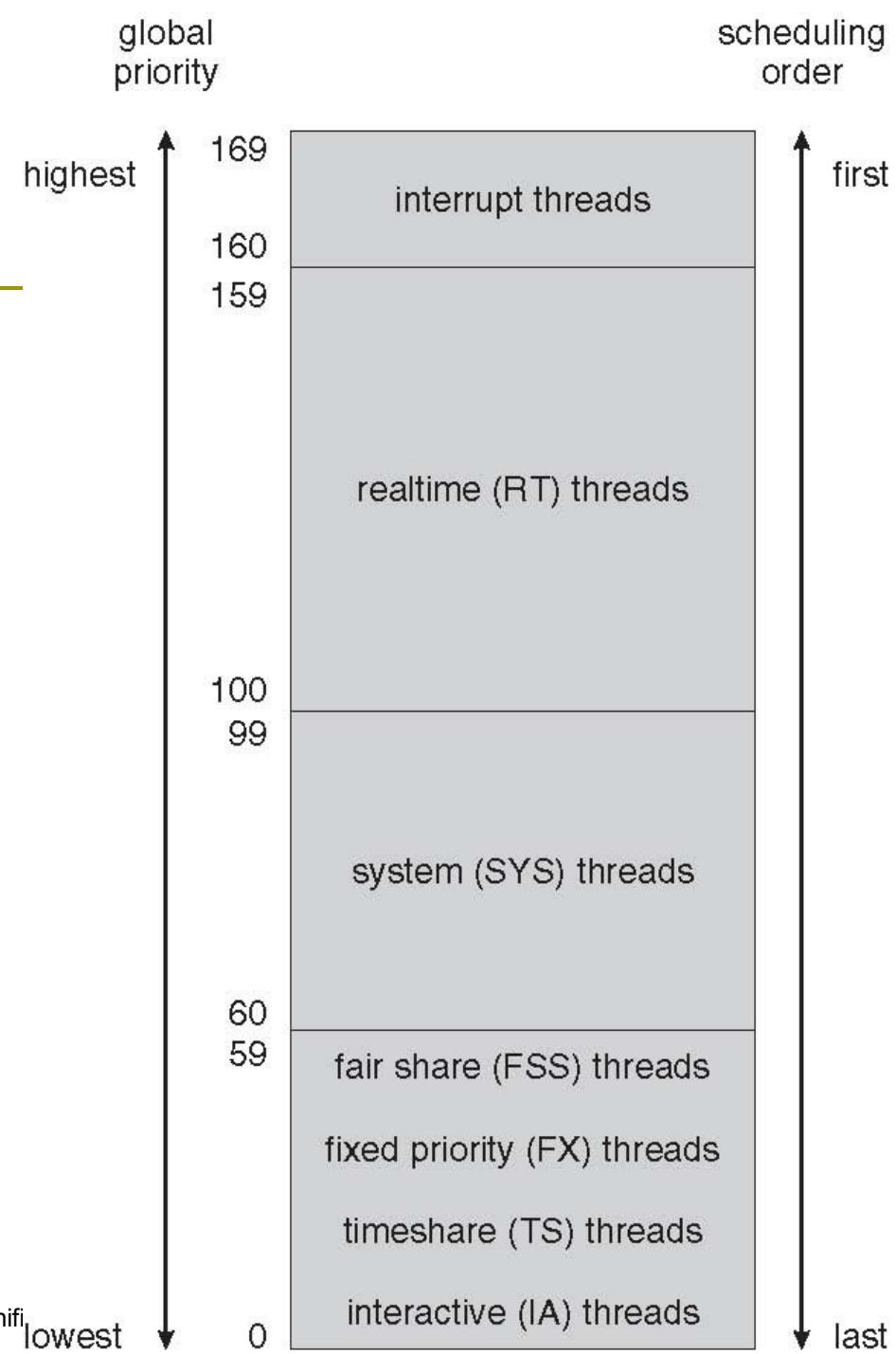
- În ordinea descrescătoare a priorității aceste **cozi de prioritate** sunt:
 - swap-ul;
 - block I/O device control;
 - lucrul cu fișiere;
 - character I/O device control;
 - procesele utilizator.
- Această ierarhie permite o utilizare eficientă a sistemului de I/O.
- În cadrul ultimei liste, a proceselor utilizator, utilizarea informațiilor despre execuțiile anterioare penalizează procesele care consumă foarte mult timp de lucru cu sistemele de I/O.
- Împreună cu algoritmul RR, această strategie de planificare poate satisface cerințele unui sistem de tip time-sharing.

Studii de caz

UNIX

- Există un set de **160 niveluri de prioritate** împărțite în trei clase:
 - **real time (159 ÷ 100):**
 - Procesele cu aceste niveluri de prioritate sunt selectate pentru execuție înaintea kernelului sau a oricărui proces.
 - pot folosi punctele de preemptare pentru a planifica procesele kernelului și procesele utilizator.
 - **kernel (99 ÷ 60):**
 - Aceste procese sunt selectate pentru rulare înaintea proceselor de tip time-sharing, dar trebuie să acorde prioritate proceselor de timp real.
 - **time-shared (59 ÷ 0):**
 - Sunt procesele cu prioritatea cea mai mică destinate proceselor utilizator, altele decât cele de timp real.
- Sunt introduse punctele de preemptare în kernel fapt ce permite întreruperea kernelului dacă datele sunt în siguranță, iar sincronizarea accesului la resurse se face cu ajutorul semafoarelor.

Studii de caz UNIX - Solaris



Studii de caz

Linux

- Kernel 1.2 – planificatorul utilizează o coadă circulară și o politică de planificare round-robin
- Kernel 2.2 – se introduc clasele de planificare
 - Politici diferite pentru următoarele tipuri de task-uri: real-time, non-preemptible și non-real-time
 - Suport pentru symmetric multiprocessing (SMP).
- Kernel 2.4 - **O(n) scheduler**
- Kernel 2.6 -2.6.23 - **O(1) scheduler**
 - Timp de planificare constant indiferent de numărul de task-uri
- Kernel post-2.6.23 - **CFS (Completely Fair Scheduler)**
 - complexitatea algoritmului **O(log N)**
 - Alegerea unui task se face în timp constant
 - Reinserarea durează $O(\log N)$

Studii de caz

Linux

□ O(1) scheduler

- Două clase de priorități:

- real-time (valoarea nice: 0-99)
 - cuanta de 200 ms
- time-sharing (valoarea nice: 100-140)
 - cuanta de 10 ms

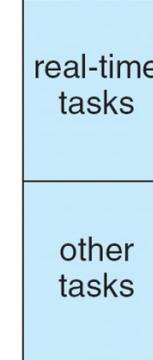
□ Kernel pre-2.6

- foloseste o lista multinivel cu feedback pentru implementarea cozii de procese în starea ready

□ Kernel post-2.6.23

- Foloseste red-black tree pentru implementarea cozii de procese în starea ready

numeric priority	relative priority	time quantum
0	highest	200 ms
•		
•		
•		
99		
100		
•		
•		
•		
140	lowest	10 ms



Studii de caz

Linux

□ clase de planificare:

■ SCHED_FIFO (FIFO real-time)

- nu este întrerupt decât dacă a apărut un proces cu prioritate mai mare, procesul se blochează sau eliberează singur procesorul
- dacă este întrerupt, procesul este pus în coada de aşteptare asociată priorității lui.
- Când un proces devine ready și are o prioritate mai mare decât procesul care se execută, procesul curent este preemptat și este executat cel cu prioritatea mai mare. Dacă sunt mai multe procese cu prioritate mare, este ales procesul care a aşteptat cel mai mult.

□ SCHED_RR (round-robin real-time)

- la sfârșitul fiecărei cuante un alt proces cu o prioritate mai mare sau egală este planificat

□ SCHED_OTHER (non-real-time)

- folosit atunci când există procese gata de execuție și nu intră în categoria proceselor real-time

Studii de caz

Linux

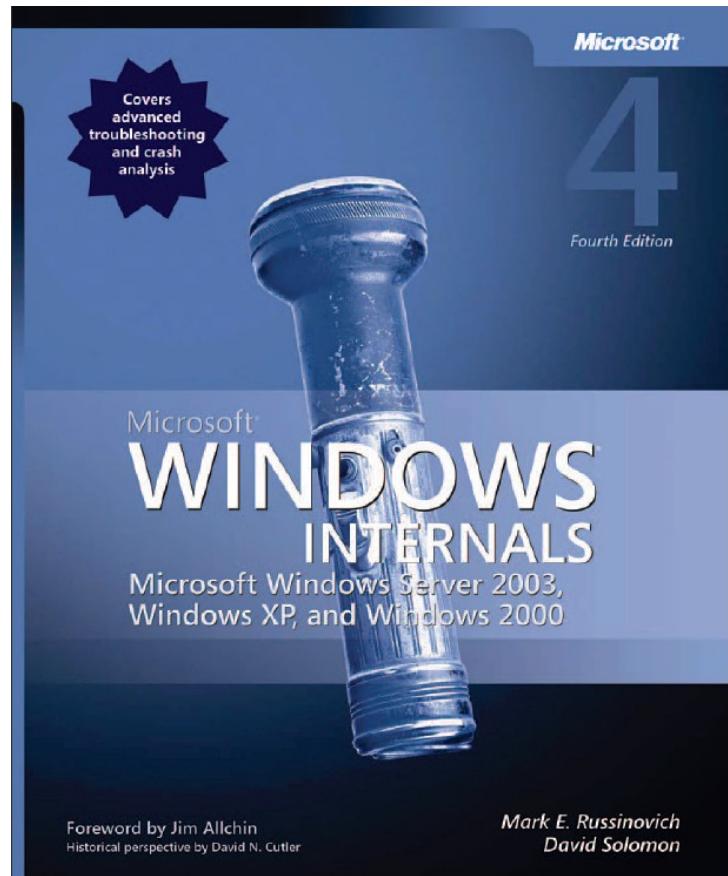
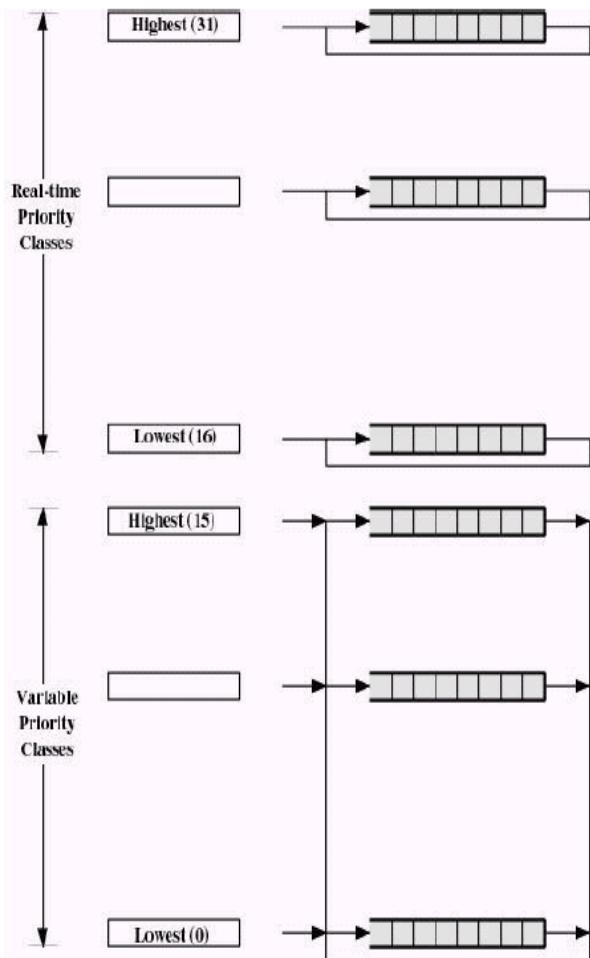
- ❑ Understanding the Linux Kernel
 - <http://oreilly.com/catalog/linuxkernel/chapter/ch10.html>
- ❑ Inside the Linux scheduler
 - <http://www.ibm.com/developerworks/linux/library/l-scheduler/>
- ❑ Inside the Linux 2.6 Completely Fair Scheduler
 - <http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/>
- ❑ http://en.wikipedia.org/wiki/Completely_Fair_Scheduler
- ❑ Linux 2.6.8.1 CPU Scheduler Paper
 - http://joshuaas.net/linux/linux_cpu_scheduler.pdf
 - http://en.wikipedia.org/wiki/Scheduling_%28computing%29

Studii de caz

Windows2000 si XP

- două clase de priorități fiecare cu 16 niveluri de prioritate, deci în total 32 de niveluri de prioritate:
 - Real time (31-16) : toate procesele au prioritate fixă, care nu se modifică în timpul rulării
 - Variable (15 – 1): prioritatea procesului se poate modifica în timpul rulării și crește pe măsură ce procesul este blocat și scade când este executat în cuanta sa de timp.
- Un thread ce ruleaza cu prioritatea 0 este folosit pentru managementul memoriei
- În cadrul fiecărui nivel de prioritate este folosit algoritmul Round-robin.

Studii de caz Windows2000 si XP



- Cozile de priorități pentru lansarea proceselor la Windows 2000 și XP

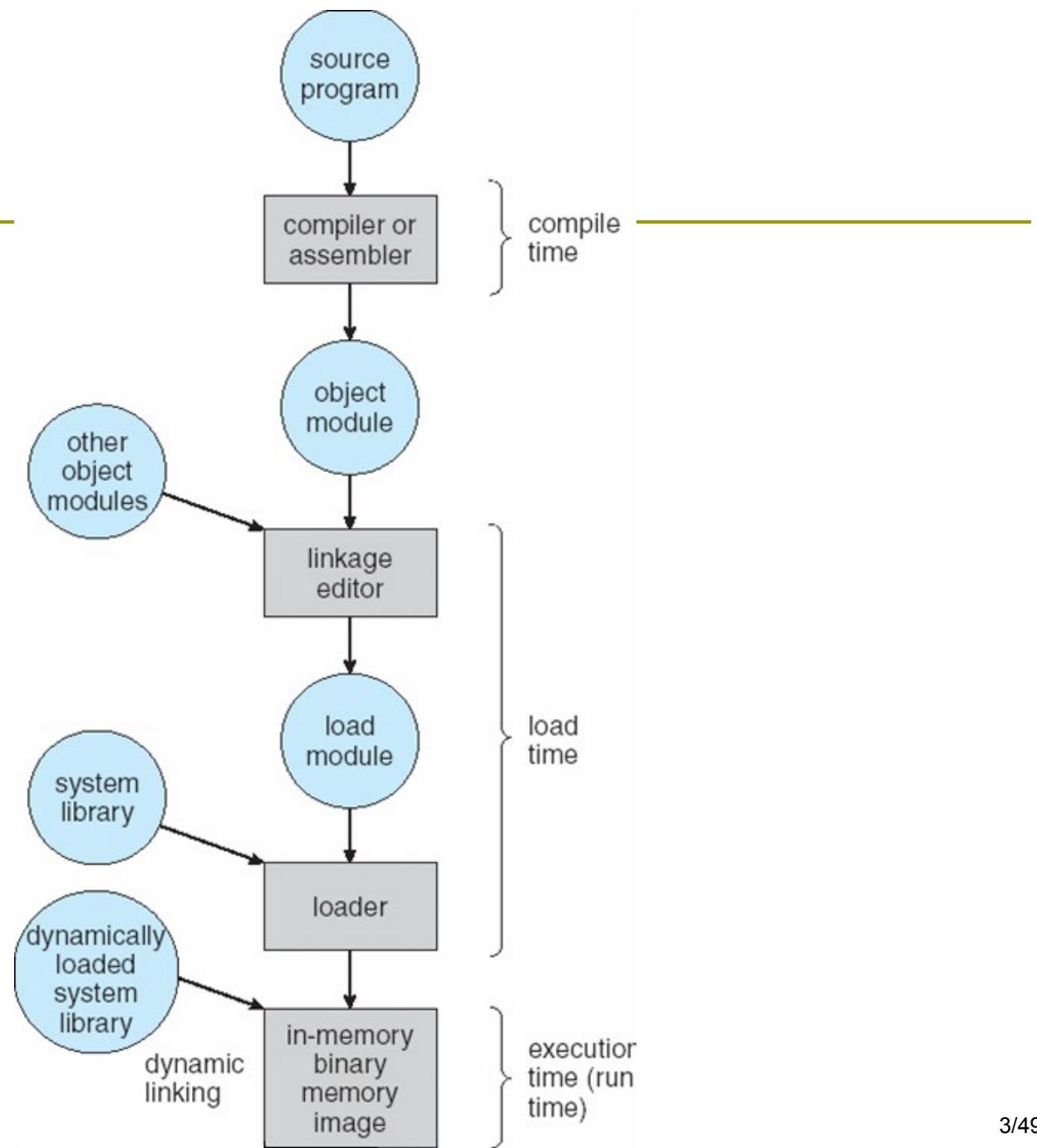
Sisteme de Operare



- Gestiunea memoriei
 - Scheme de alocare a memoriei
 - Memoria virtuală

Gestiunea memoriei

- Pentru a putea fi executat, un proces are nevoie de o anumită cantitate de memorie.
- Dacă sistemul suportă multiprogramare, este necesar ca în memorie să fie prezente mai multe programe, fiecare folosind zonele de memorie alocate independent de eventualele programe active. Pe durata execuției unui proces, necesarul de memorie poate varia.
- Spațiul de memorie principală a unui sistem de calcul (aceea care poate fi accesată direct de către CPU) este fix și trebuie gestionat cât mai eficient.



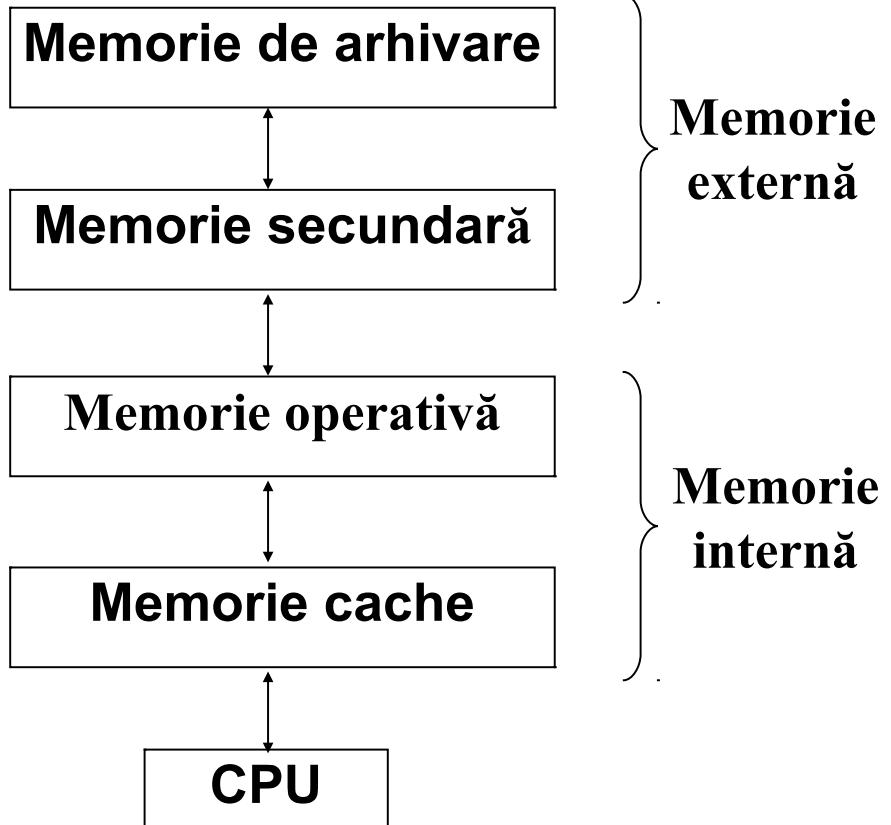
Gestiunea memoriei

□ Obiective:

- calculul de translatare a adresei (relocare)
- protecția memoriei
- organizarea și alocarea memoriei operative
- gestiunea memoriei secundare
- politici de schimb între proces, memoria operativă și memoria secundară

Gestiunea memoriei

Structura ierarhică a memoriei



■ Memoria cache

- conține informațiile cele mai recent utilizate de CPU, are capacitate mică dar timp de acces foarte rapid.
- La fiecare acces CPU verifică dacă data se află în memoria cache și apoi solicită memoria operativă. Dacă este, are loc transferul între CPU și memoria operativă, iar dacă nu se caută data în nivelurile superioare.

Gestiunea memoriei

- **Principiul vecinătății:**
 - dacă la un moment dat se solicită o informație dintr-un anumit loc, atunci solicitarea din momentul următor se va face cu o mare probabilitate la o informație din apropierea precedentei.
 - Informația cerută de CPU este adusă din nivelul în care se află, dar împreună cu ea sunt aduse și un număr de locații vecine astfel încât să umple memoria cache.

Gestiunea memoriei

Structura ierarhică a memoriei

□ Memoria operativă

- conține programele (codul) și datele pentru toate procesele existente în sistem.
- În momentul în care un proces este terminat și distrus, spațiul din memoria operativă pe care l-a ocupat este eliberat și va fi alocat altui proces.
- Viteza de acces este foarte mare (memoria SDRAM, DDRAM).

Gestiunea memoriei

Structura ierarhică a memoriei

□ Memoria secundară

- apare la SO care dețin mecanisme de memorie virtuală.
- Această memorie este privită ca o extensie a memoriei operative.
- Suportul ei principal este discul magnetic și din acest motiv este mult mai lentă decât memoria operativă.

Memoria expandată

- ❑ Este mecanism ce permite ca mai multe chip-uri de memorie operativă să aibă, alternativ, aceeași adresă de memorie.
- ❑ Apare la primele sisteme IBM PC din seria 8086 și 8080, la care memoria disponibilă era de 640Ko.
- ❑ Este o memorie secundară care are ca suport memoria internă.

Gestiunea memoriei

Structura ierarhică a memoriei

□ Memoria de arhivare

- Este gestionată de utilizator și constă din fișiere, baze de date etc, rezidente pe diferite suporturi de stocare a informației.

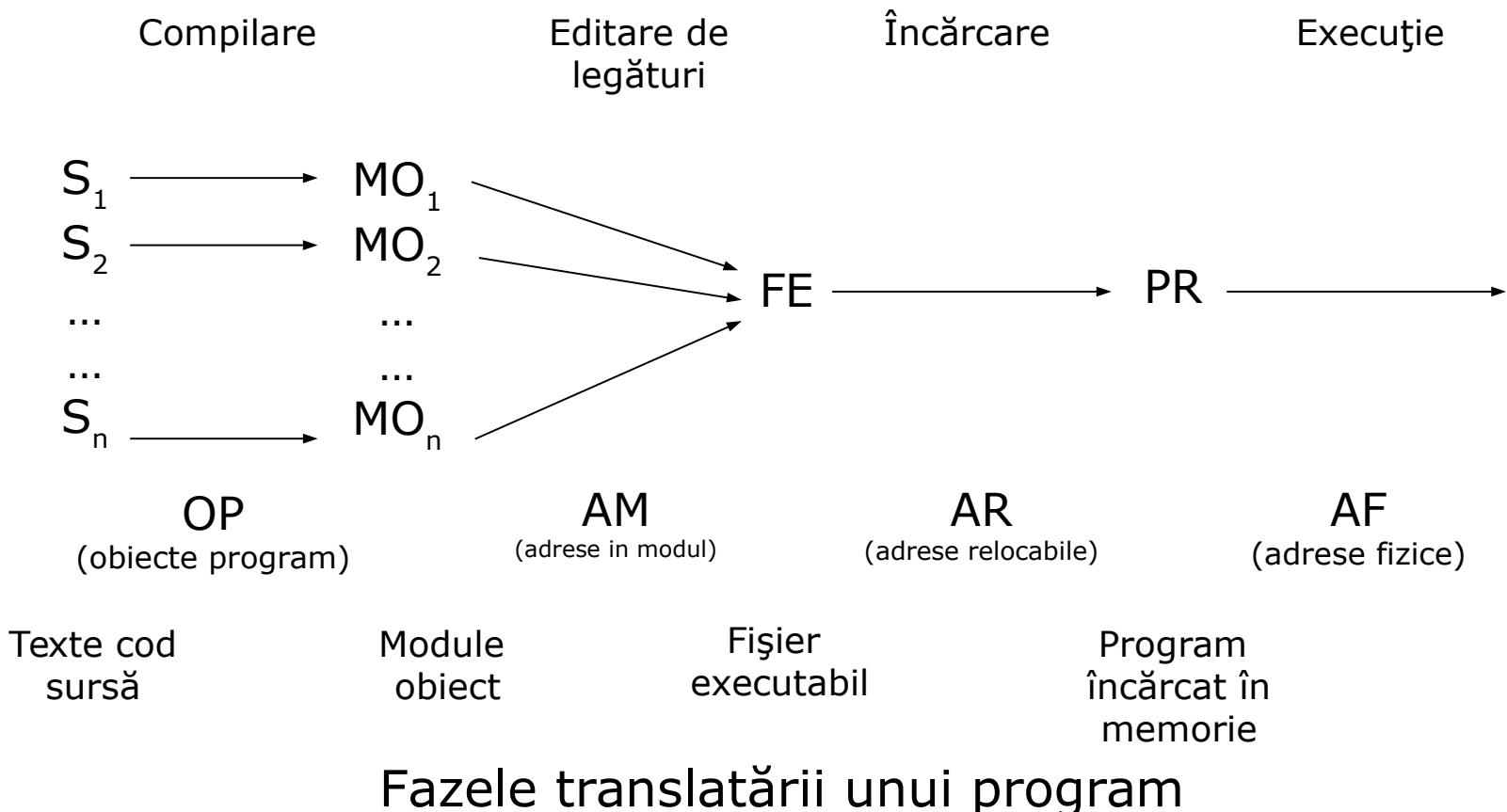
□ Memoria cache și memoria operativă formează memoria internă.

- Accesul CPU se face în mod direct.
- Pentru a avea acces la datele din memoria secundară și de arhivare, acestea trebuie mutate în memoria internă.

Gestiunea memoriei

Mecanisme de translatare a adresei

- Adresarea memoriei constă în realizarea legăturii între un obiect al programului și adresa corespunzătoare din memoria fizică



Gestiunea memoriei

Mecanisme de translatare a adresei

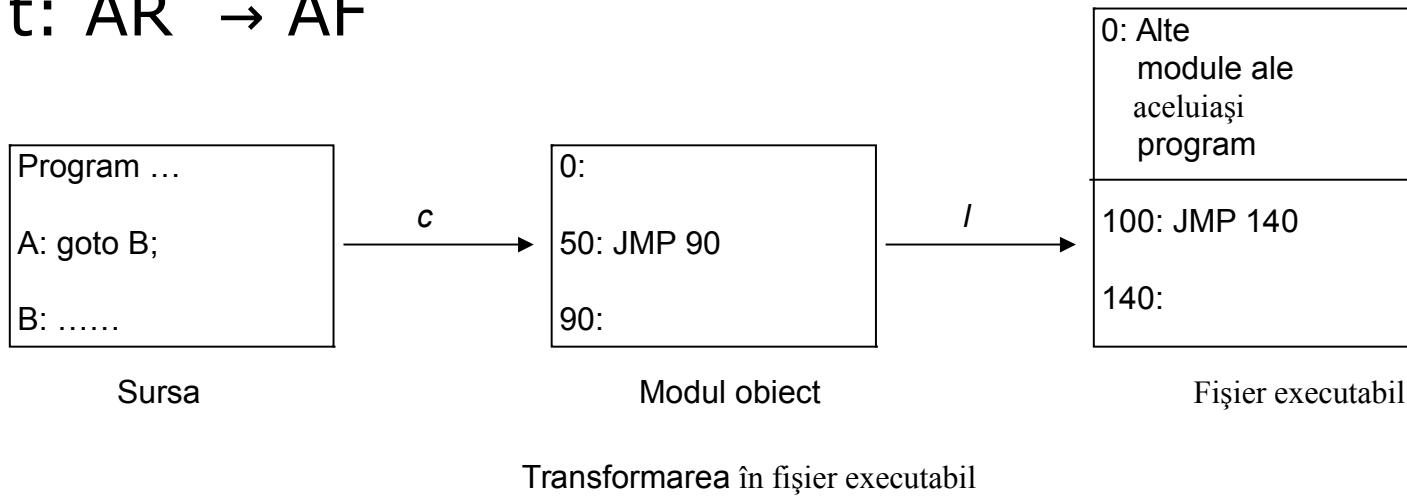
- Calculul de adresă este modalitatea prin care se ajunge de la OP la AF și necesită trei faze corespunzătoare fazelor programului:
 - **compilare**: textul sursă este transformat în module obiect. Numele de obiectele din program sunt transformate în numere reprezentând **adrese în cadrul modulului (AM)**.
 - Prima funcție din cadrul calculului de adresă este:
 - $c: OP \rightarrow AM$
 - și este executată de compilator. Modul ei de evaluare depinde de limbaj, compilator și SO.
 - **editare de legături**: sunt grupate mai multe module pentru a forma fișierul executabil. Adresele din cadrul modulului sunt transformate în **adrese relocabile (AR)** de către editorul de legături.
 - Funcția este:
 - $I: AM \rightarrow AR$

Gestiunea memoriei

Funcția de translatare (relocare) a adresei

- transformă adresele relocabile în **adrese fizice (AF)**, este executată de CPU și depinde de sistemul de calcul și de existența dispozitivelor de management a memoriei:

- t: AR → AF



Gestiunea memoriei

□ Moduri de adresare uzuale

■ adresarea absolută:

- are loc atunci când $t(x)=x$, $\forall x \in AR$ sau altfel spus când $AR=AF$.

■ adresarea bazată:

- presupune utilizarea unui registru general ca registru de bază:

- $t(x) = (R_b) + x$, $\forall x \in AR$

- toate adresele relocabile sunt mărite cu ajutorul registrului de bază.

■ adresarea indexată:

- presupune specificarea, în cadrul instrucțiunii mașină a unui **registru de index R_i** . După ce se obține o adresă provizorie AF1 (în funcție de arhitectura procesorului), adresa definitivă se obține astfel:

$$AF2 = AF1 + (R_i)$$

- Acest mod de adresare se folosește pentru localizarea elementelor în cadrul unui tablou.

Gestiunea memoriei

□ Moduri de adresare uzuale

■ adresarea relativă:

- se folosește pentru realizarea de salturi într-un program, precizându-se sensul și numărul de locații peste care trebuie sărit pentru a ajunge la noua adresă:

$$AF2 = AF1 + I$$

■ adresarea indirectă:

- După ce procesorul obține o adresă AF1, el interpretează conținutul de la AF1 nu ca o valoare a unui operand, ci ca o nouă adresă:

$$AF2 = (AF1).$$

- Acest mod de adresare se aplică în special la invocarea parametrilor actuali din cadrul unui subprogram. Programul apelant transmite subprogramului lista cu adresele parametrilor actuali, după care subprogramul are acces la ei folosind adresarea indirectă.

Gestiunea memoriei

□ Protecția memoriei

- Fiecare sistem de calcul și sistem de operare trebuie să aibă mecanisme care să asigure utilizarea corectă a spațiului de memorie de către toate procesele.
- Fiecare entitate de memorie alocată conține o **cheie de protecție**, iar fiecare entitate de program încărcabilă în memorie la un moment dat are o **cheie de acces**.

Gestiunea memoriei

Protecția memoriei

- Cheia de protecție - un sir de biți prin care se specifică posibilitățile zonei respective:
 - R: din zonă se poate citi.
 - Se permite execuția instrucțiunilor mașină care aduc datele din zonă în registri. Dacă este numai read-only, memorarea datelor din registri în zona respectivă este interzisă.
 - În aceste zone se trec de obicei elementele constante ale proceselor.
 - W: în zonă se poate scrie.
 - Este permisă memorarea datelor din registri în zona respectivă (dacă este interzisă avem situația de la read-only).
 - Există două cazuri speciale de scriere, care la unele SO sunt specificate separat: extindere (scrierea la sfârșitul zonei) și ștergere (pregătirea zonei astfel încât extinderea la momentul următor se va face de la începutul zonei) și sunt folosite mai mult la fișiere decât la memorie.
 - E: conținutul zonei poate fi executat
 - există instrucțiuni mașină ce pot fi executate.
 - În aceste zone conținutul rămâne neschimbăt, fiind interzis proceselor să modifice zona respectivă.
 - La unele SO este asimilată cu posibilitatea de read-only.

Gestiunea memoriei

Protecția memoriei

□ Cheia de acces

- un sir de biți reprezentând drepturi de acces.
- Este primită de un proces la încărcare

□ Protecția memoriei se asigură executând doi pași:

- la fiecare solicitare a unei locații de memorie se compară cheia de protecție cu cheia de acces.
- În caz de neconcordanță accesul este interzis și procesul se termină cu mesaj de eroare.
- Dacă cheile coincid, se compară posibilitățile zonei solicitate cu drepturile de acces ale procesului și cu acțiunea cerută de proces.
- Accesul este permis numai dacă răspunsul la aceste comparații este afirmativ.

Scheme de alocare a memoriei

□ alocare reală:

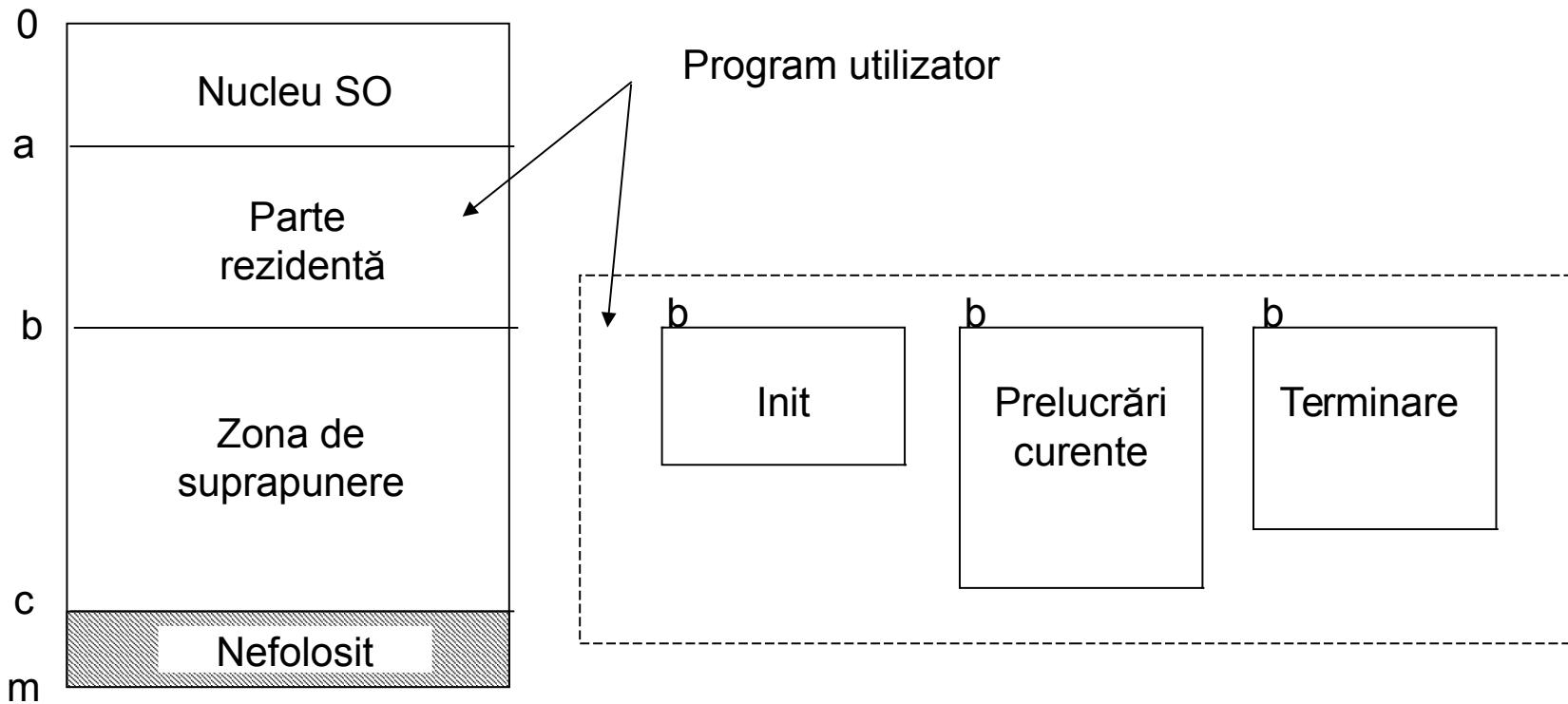
- la sistemele monoutilizator
- la sistemele multiutilizator:
 - cu partiții fixe (statică):
 - absolută;
 - relocabilă;
 - cu partiții variabile (dinamică)

□ alocare virtuală:

- paginată;
- segmentată;
- segmentată și paginată

Alocarea la sistemele monoutilizator

- La sistemele monoutilizator este disponibil aproape întreg spațiul de memorie.
- Gestiunea spațiului cade în sarcina utilizatorului, existând tehnici de suprapunere (overlay) pentru a-și putea rula programele mari:



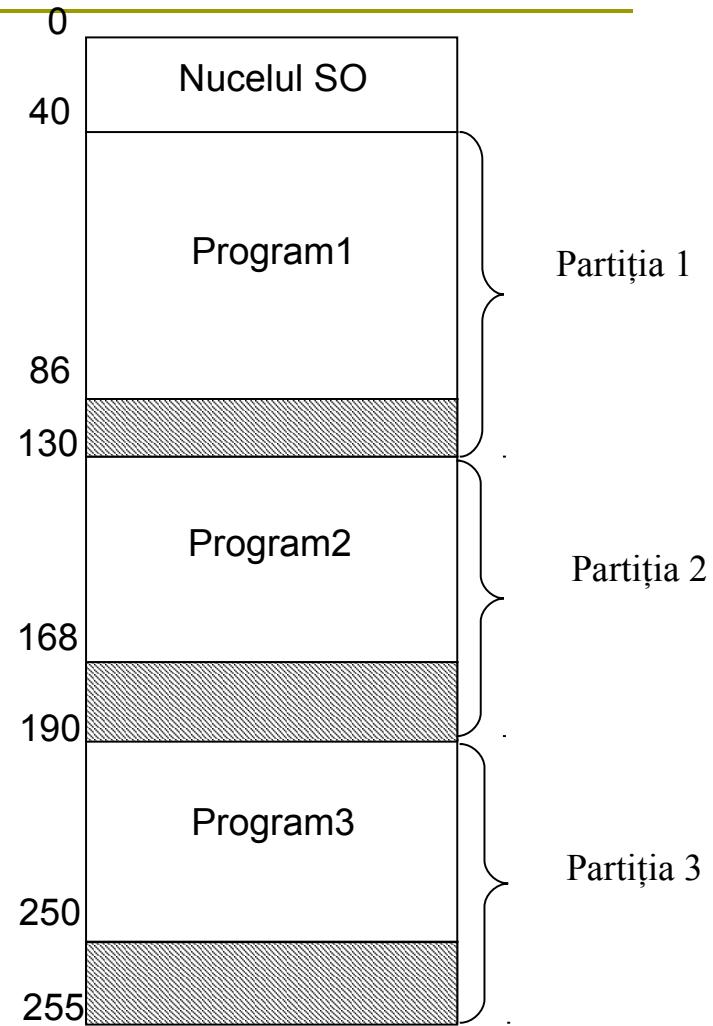
Alocarea memoriei la un sistem monoutilizator folosind suprapunerea

tehnici de suprapunere

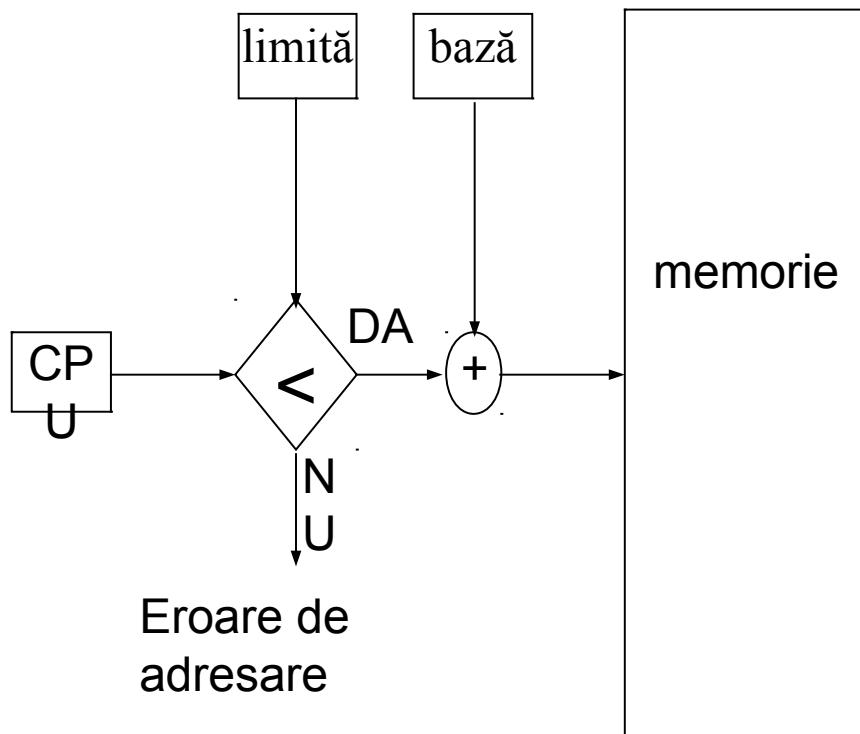
- presupune păstrarea în memorie a instrucțiunilor și datelor necesare la un moment dat.
- Când este nevoie de alte instrucțiuni acestea sunt încărcate în memorie în locul celor de care nu mai este nevoie:
- zona dintre adresele **0** și **a-1** este rezervată nucleului sistemului de operare ce rămâne acolo de la încărcare și până la oprirea sistemului.
- Între **c** și **m-1** este spațiul nefolosit de către programul utilizator activ (dacă memoria are m locații).
- Adresa **c** variază de la un program utilizator la altul.

Alocarea cu partii fixe

- Această alocare se mai numește și alocare statică sau alocare **MFT – Memory Fix Task.**
- presupune decuparea memoriei în zone de lungime fixă numite partii.
- O parte este alocată unui proces pe toată durata execuției lui, indiferent dacă o ocupă complet sau nu.
- partiiile au lungimi diferite



Alocarea cu partii fixe



Adresarea cu registru bază

- **Alocarea absolută:** se face pentru programe pregătite de editorul de legături pentru a fi rulate într-o zonă de memorie și numai acolo.
- **Alocarea relocabilă:** adresarea în partiție se face cu bază și deplasament: la încărcarea în memorie a programului, în registrul lui de bază se pune adresa de început a partiției.

Alocarea cu partiții fixe

□ Fixarea dimensiunii partiților

- Nu se pot prevedea cantitățile de memorie pe care le vor solicita procesele încărcate în aceste partiții
- Alegerea unor partiții de dimensiuni mari scade probabilitatea ca unele procese să nu poată fi executate, dar și numărul de procese active din sistem.
- La fiecare partiție există un sir de procese care așteaptă să fie executate
 - Modul în care este organizat sistemul de așteptare poate influența performanțele de ansamblu ale sistemului de calcul și poate atenua efectul unei dimensionări defectuoase a partiților.

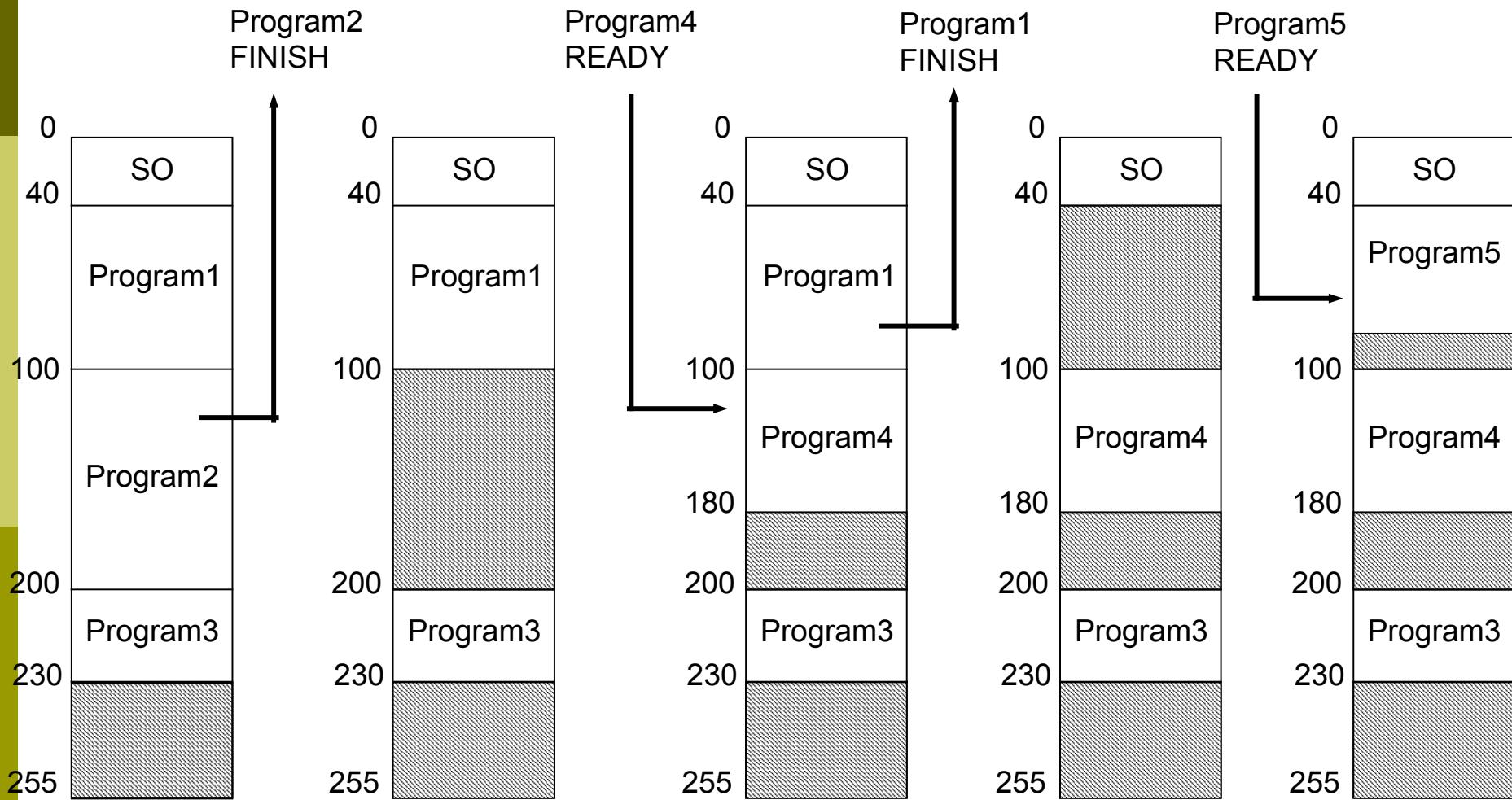
Alocarea cu partii fixe

- Legarea proceselor de partii
 - fiecare partie are o coadă proprie:
 - este o metodă mai simplă din punctul de vedere al sistemului de operare (se întâlnește la primele sisteme monoutilizator IBM OS/MFT).
 - o singură coadă pentru toate partiiile:
 - se poate alege partia cea mai potrivită pentru plasarea unui proces.

Alocarea cu partii variabile

- Această alocare se mai numește și alocare dinamică sau alocare **MVT – Memory Variable Task** (era folosită la PDP11).
- Este o extensie a alocării cu partii fixe și permite o exploatare mai eficientă a memoriei sistemului de calcul:
 - numărul și dimensiunea partiiilor se modifică automat în funcție de:
 - solicitări
 - de capacitatea memoriei rămasă disponibilă la un moment dat

Alocarea memoriei cu partii variabile



Fragmentarea internă a memoriei

- La intrarea în sistem, procesele sunt plasate în memorie într-un spațiu în care începe cea mai lungă ramură a sa.
- Spațiul liber în care a intrat procesul se împarte în două partiții: una în care este procesul și una liberă.
- Dacă sistemul funcționează un timp îndelungat se ajunge la situația în care numărul spațiilor libere va crește, iar dimensiunea lor va scădea.

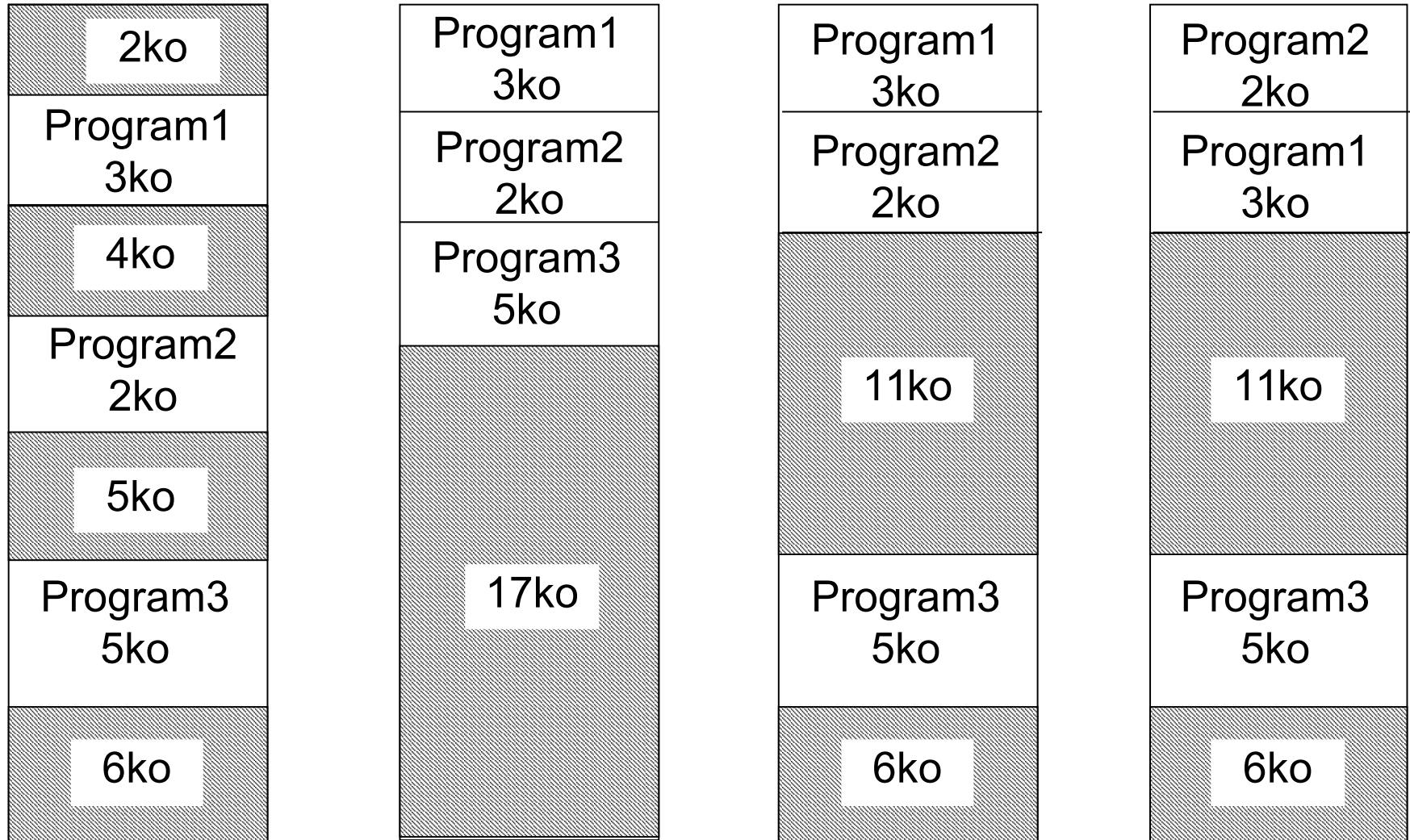
Fragmentarea internă a memoriei

- Dacă un proces nu are spațiu să se încarce în memorie, sistemul de operare poate lua următoarele decizii:
 - procesul așteaptă până se eliberează o cantitate suficientă de memorie;
 - sistemul de operare încearcă alipirea unor spații de memorie libere vecine, pentru a obține un spațiu de memorie liber suficient de mare pentru încărcarea programului;
 - sistemul de operare decide efectuarea unei operații de compactare a memoriei (relocare) – deplasarea partițiilor active către partitia unde se află partea rezidentă a SO pentru a se absorbi toate "fragmentele" de memorie neutilizate.

Compactarea memoriei

- Compactarea memoriei este, de regulă, o operație costisitoare și în practică se aleg soluții de compromis:
 - se lansează periodic compactarea, indiferent de starea sistemului. Procesele care nu au loc în memorie așteaptă compactarea sau terminarea altui proces;
 - se realizează o compactare parțială pentru a asigura loc numai procesului care așteaptă;
 - se încearcă numai mutarea unor dintre procese și se colecționează spațiile libere rămase.

Compactarea memoriei



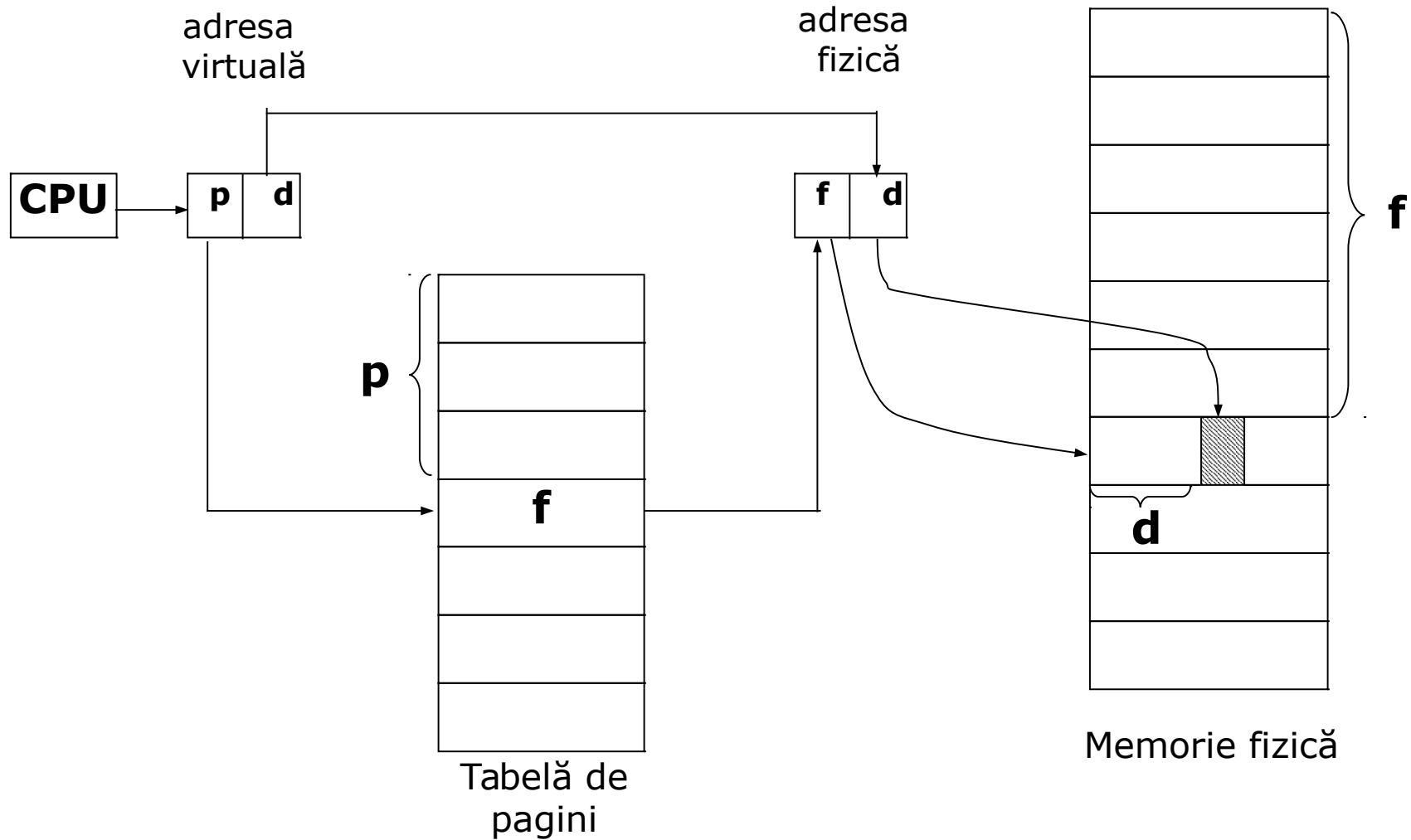
Alocarea paginată a memoriei

- Paginarea este o metodă care rezolvă problema fragmentării:
 - memoria alocată unui program nu este contiguă, adică programul poate fi încărcat în memorie acolo unde există memorie disponibilă.
- Această alocare presupune că:
 - instrucțiunile și datele fiecărui program sunt împărțite în zone de lungime fixă, numite pagini virtuale. Fiecare adresă relocabilă (AR) aparține unei pagini virtuale. Paginile virtuale se păstrează în memoria secundară.
 - memoria operativă este împărțită în **zone de lungime fixă**, numite **pagini fizice** sau **cadre**.
 - Lungimea unei pagini fizice este fixată prin hardware.
 - Paginile virtuale și cele reale au aceeași lungime (o putere a lui 2) și reprezintă o constantă a sistemului (de exemplu: 1Ko, 2Ko etc).

Translatarea unei pagini virtuale într-o pagină fizică

- fiecare **AR (adresă relocabilă)** este o pereche de forma **(p, d)** unde **p** este numărul paginii și **d** este adresa în cadrul paginii (deplasarea în pagină).
- fiecare **AF (adresă fizică)** este o pereche de forma **(f, d)** unde **f** este numărul paginii fizice și **d** este adresa în cadrul paginii.
- calculul funcției de translare se face prin hardware.

Translatarea unei pagini virtuale într-o pagină fizică

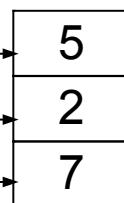
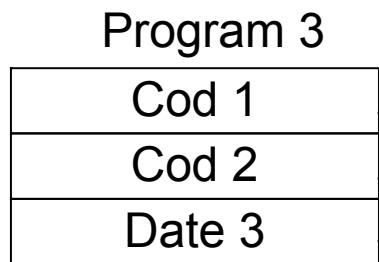
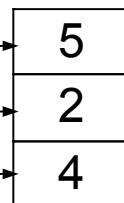
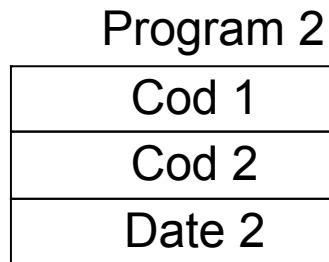
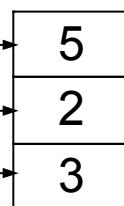
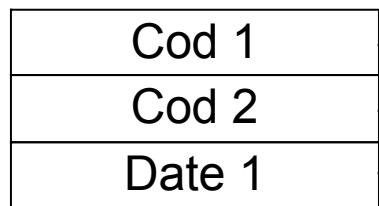


Alocarea paginată a memoriei

- Fiecare proces are propria lui tabelă de pagini, în care este trecută adresa fizică a paginii virtuale, dacă ea este prezentă în memoria operativă.
- La încărcarea unei noi pagini virtuale, aceasta se depune într-o pagină fizică liberă și în acest fel are loc o proiecțare a spațiului virtual peste cel real
- se folosește mai eficient memoria operativă.
- folosirea în comun a instrucțiunilor unor proceduri de către mai multe programe. O procedură care permite acest mod de lucru se numește **procedură reentrantă (cod reentrant - instrucțiuni pure fără date)**.

Procedură reentrantă alocată cu paginare

Program 1



Pagini virtuale

Tabele de pagini

0	
1	
2	Cod 2
3	Date 1
4	Date 2
5	Cod 1
6	
7	Date 3
8	
:	

Implementarea tabelei de pagină

- Dacă dimensiunea tabelei de pagină este redusă se poate utiliza un set de registre specializate, foarte rapide, care să asigure o eficiență ridicată a translării adreselor.
- Instrucțiunile destinate încărcării sau modificării acestor registre trebuie accesate numai de către sistemul de operare.
- Dacă dimensiunea tabelei este mare, este preferabil ca ea să fie păstrată în memoria principală, într-o zonă indicată de valoarea unui registru specializat numit **registrul de bază al tabelei de pagină (RBTP)**

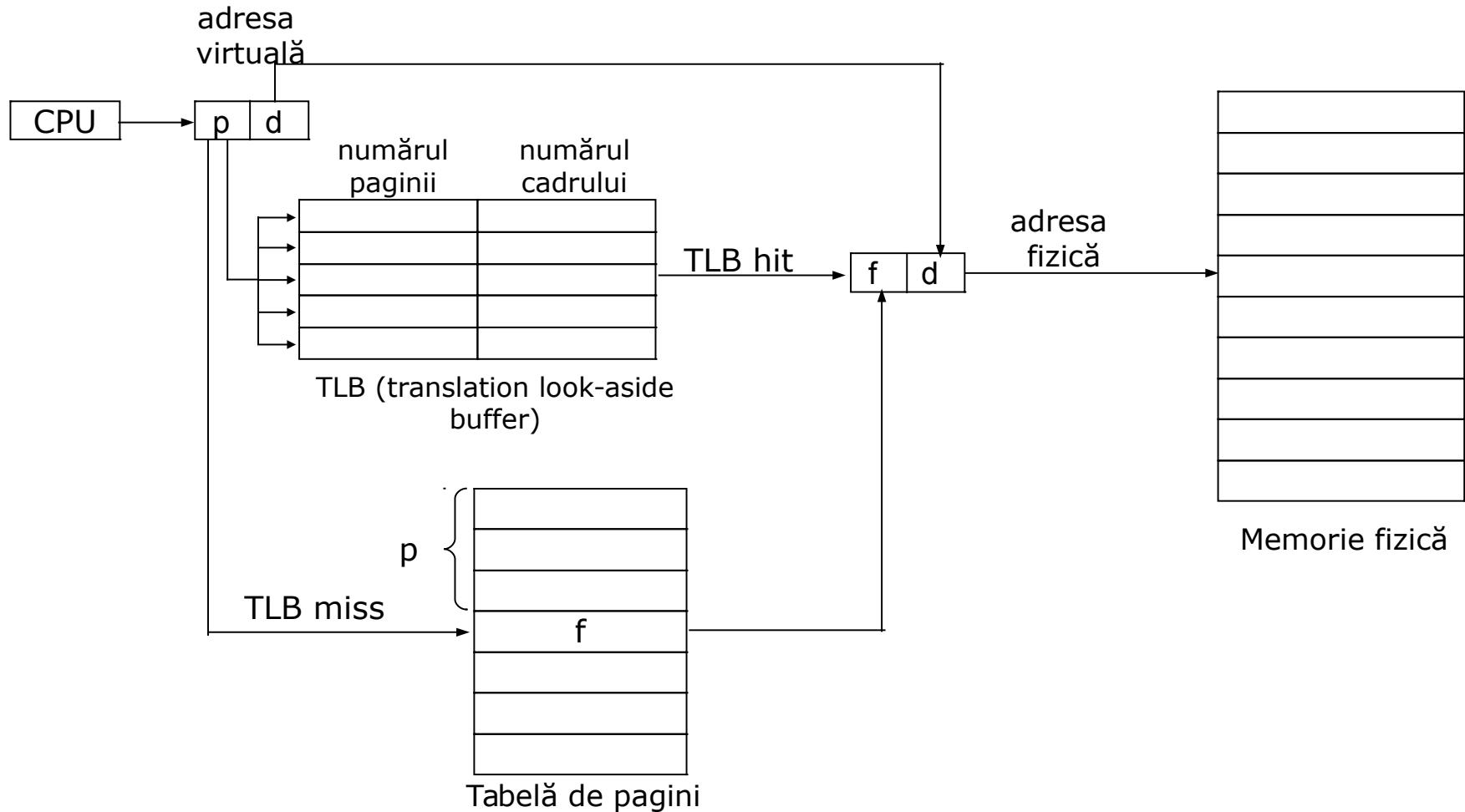
Implementarea talelei de pagină

- Dacă trebuie să se lucreze cu o altă tabelă de pagină decât cea curentă trebuie reîncărcat registrul cu o altă valoare scăzând astfel timpul de schimbare al contextului.
- O particularitate a acestei soluții este faptul că pentru a accesa o zonă de memorie utilizator sunt necesare două operații de acces la memorie – una pentru tabela de pagină și alta pentru cuvântul propriu-zis.
- Folosind valoarea din RBTP deplasată cu numărul de pagină - p (aflat în adresa logică), se determină mai întâi numărul de cadru – c asociat paginii, care împreună cu deplasamentul în pagină – d dă adresa reală.

Implementarea tăbelei de pagină

- O altă soluție ar fi folosirea unei memorii hardware speciale (un set de registre asociative sau **translation look-aside buffer - TLB**), de mică dimensiune, cu următoarele caracteristici:
 - fiecare registru are două părți, cheie (conține numărul paginii) și valoare (numărul cadrului).

Translarea paginii folosind TLB



Paginarea multinivel

- Sistemele de calcul suportă un spațiu logic de adresare foarte mare (2^{32} sau 2^{64}) și din acest motiv tabela de pagini trebuie să fie foarte mare.
- Pentru un sistem cu spațiul logic de adresare pe 32 de biți , dacă avem mărimea paginii de 4K bytes (2^{12}), atunci numărul de intrări în **tabela de pagini** ar trebui să fie de peste 1 milion ($2^{32} / 2^{12} = 2^{20} = 1048576$).
- Deoarece fiecare intrare constă în 4 bytes, fiecare proces poate avea nevoie de 4 Mbytes de spațiu de adresare pentru tabela de pagini.
- Este aproape imposibil să alocăm tabela de pagini într-o zonă contiguă de memorie.

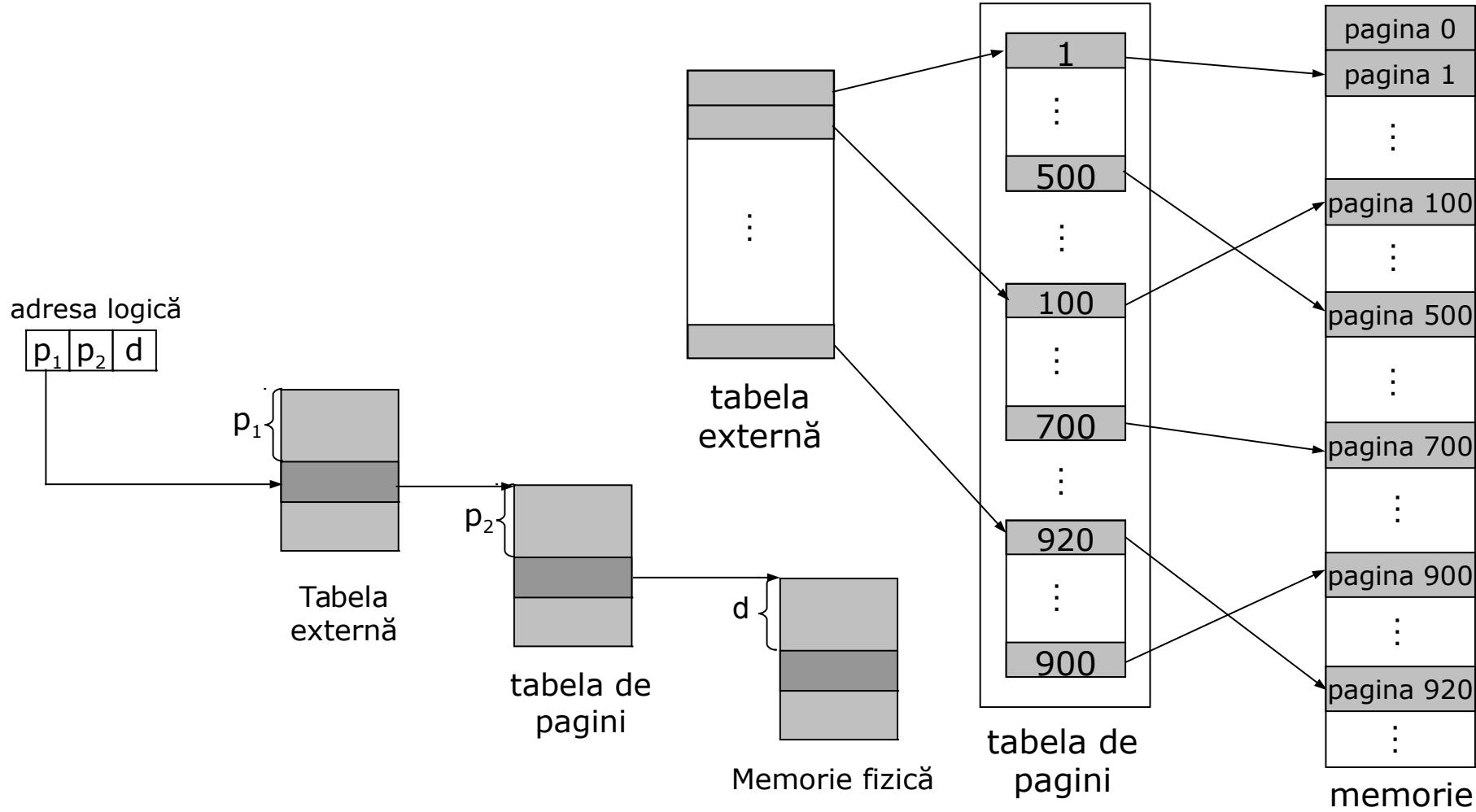
Paginarea multinivel

- Soluția pentru rezolvarea acestei probleme este paginarea multinivel:
 - adresa logică o putem împărți în numărul de pagină de 20 biți și deplasamentul în pagină de 12 biți.
- Deoarece vrem să paginăm tabela de pagini, putem împărți numărul paginii în două părți:

Numărul paginii	deplasament
p_1	p_2
10	12

- unde p_1 este index în tabela de pagini externă, p_2 este deplasamentul în această tabelă și d este deplasamentul în pagina de memorie.

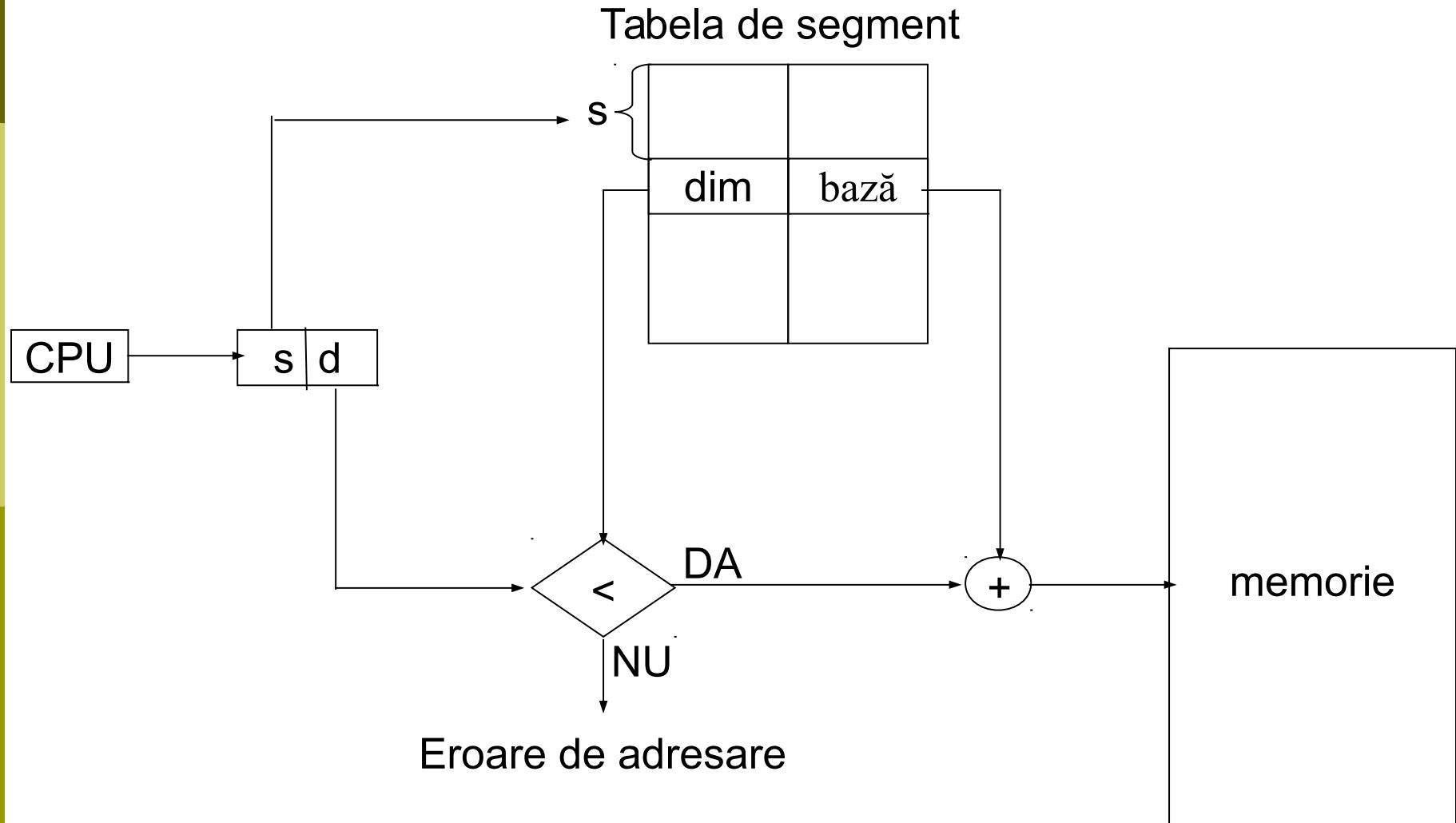
Translarea adresei în cazul paginării cu 2 niveluri



Alocarea segmentată a memoriei

- Mecanismul de alocare segmentată introduce faptul că textul unui program poate fi plasat în zone de memorie distincte, fiecare zonă conținând o bucată de program numită segment.
- Fiecare segment este caracterizat prin nume și lungime.
- O adresă virtuală este o pereche **(s, d)**, unde **s** este numărul segmentului și **d** este deplasamentul în cadrul segmentului.
- Acestei perechi îi corespunde o adresă fizică, iar corespondența este realizată prin tabela de segment ce conține un număr de intrări egal cu numărul de segmente din program.

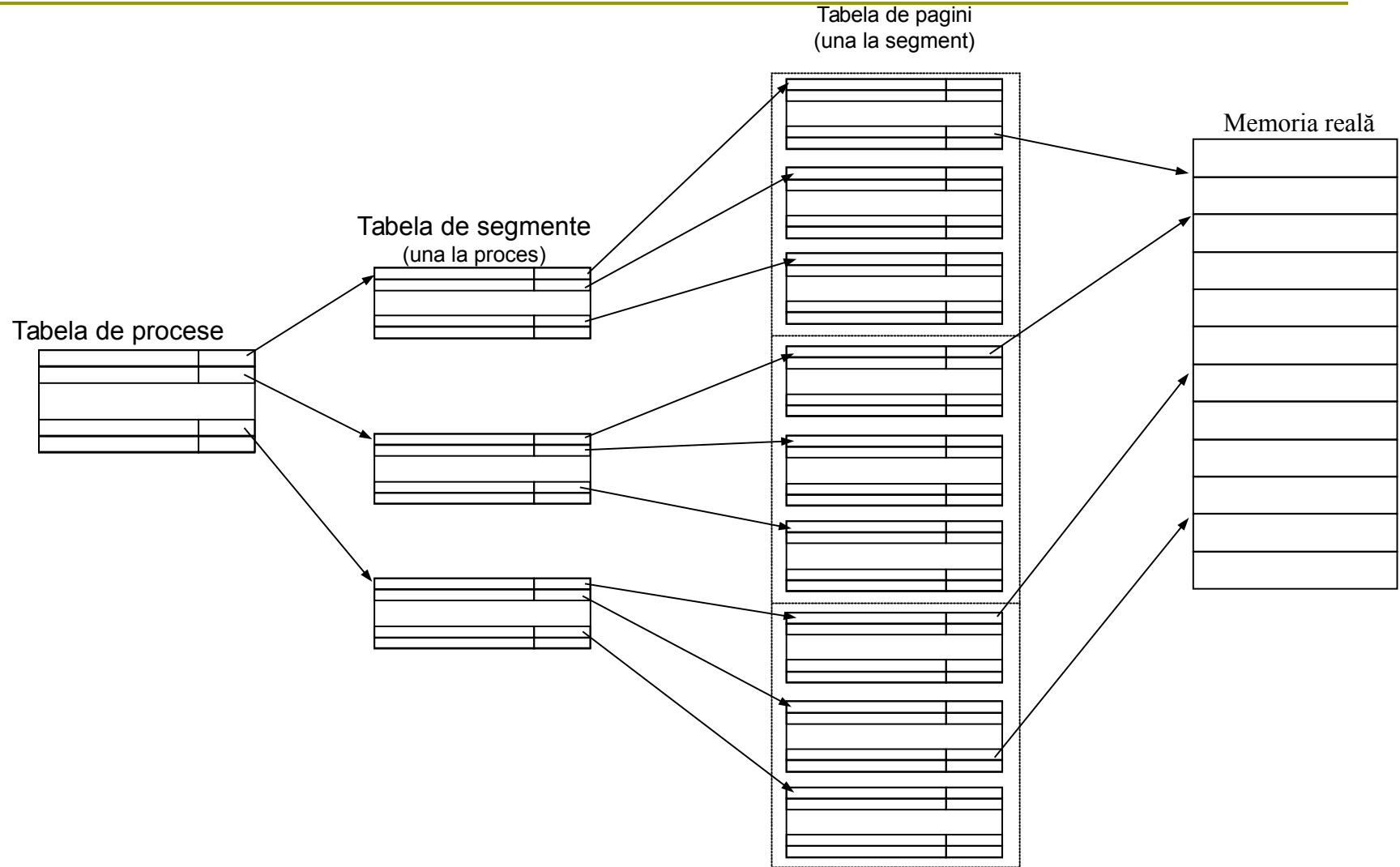
Translatarea adresei virtuale cu ajutorul tableei de segment



Avantaje față de alocarea pe partiții

- se pot crea **segmente reentrantă** ce pot fi folosite de mai multe procese.
- se poate realiza protecția memoriei prin adăugarea unor drepturi de acces.

Alocarea segmentată și paginată

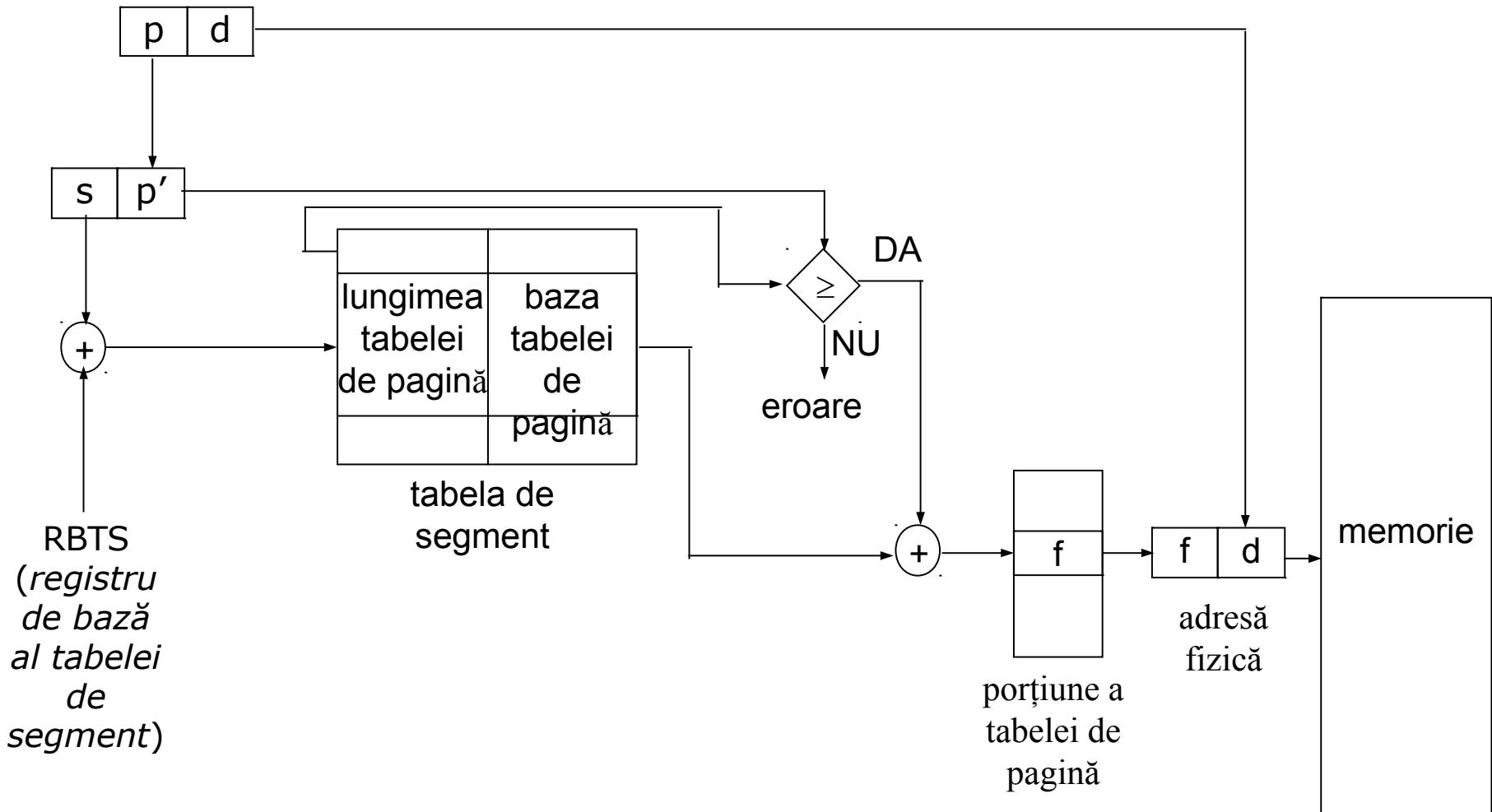


Alocarea segmentată și paginată

- Fiecare proces are propria lui tabelă de segmente.
- Fiecare segment are propria lui tabelă de pagini.
- Fiecare intrare în tabela de segmente are un câmp rezervat adresei de început a tabelei de pagini proprii segmentului.
- Adresa virtuală este de forma **(s, p , d)**, unde **s** este numărul segmentului, **p** este numărul paginii virtuale în cadrul segmentului, iar **d** este deplasamentul în cadrul paginii.
- Adresa fizică este de forma **(f, d)**, unde **f** este numărul paginii fizice, iar **d** este deplasamentul în cadrul paginii.

Paginarea segmentelor

Adresă logică



Sisteme de Operare

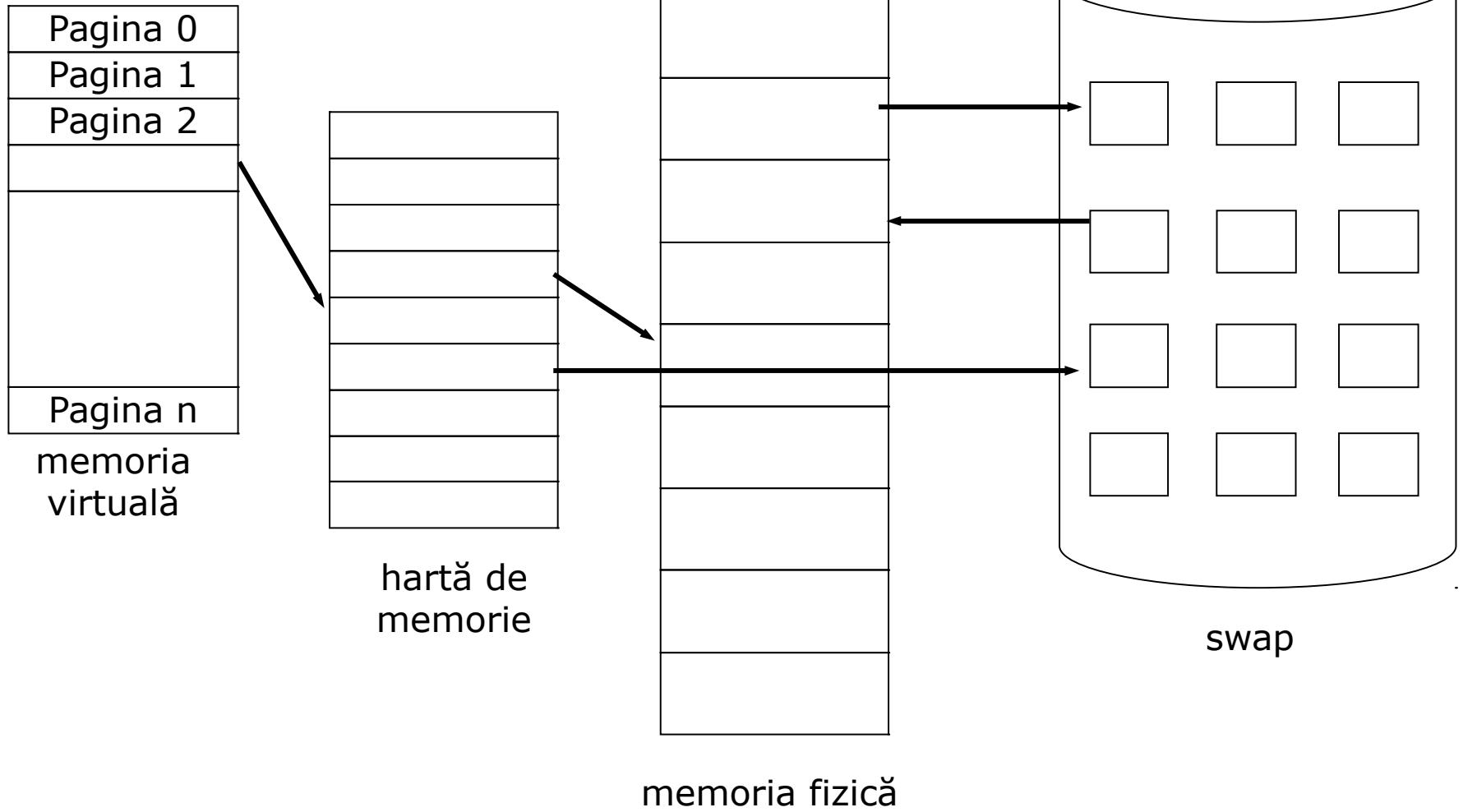
□ Gestiunea memoriei

- Memoria virtuală
- Planificarea schimburilor cu memoria (înlocuirea paginii)
- Algoritmi de înlocuire a paginii
- Alocarea cadrelor
- Studii de caz: Linux, Unix, Windows

Memoria virtuală

- Memoria virtuală este o tehnică ce permite execuția proceselor, chiar dacă acestea nu se află integral în memorie.
- Un avantaj direct este acela al rulării unor programe cu dimensiuni mai mari decât cele ale memoriei fizice

Memoria virtuală



Implementarea memoriei virtuale

- În sistemele cu paginare prin metoda numită paginare la cerere, metodă ce poate fi folosită și la sistemele cu segmentare sau cu paginarea segmentelor.
- O altă metodă este segmentarea la cerere, mult mai rar folosită, datorită faptului că algoritmii sunt mai complicați (din cauza mărimii variabile a segmentelor).

Paginarea la cerere

- se introduce în memorie întregul program, ci numai câteva pagini, atunci când sunt necesare
- Tabela de pagină corespunzătoare unui proces va conține același tip de informații ca și în cazul paginării obișnuite, având în plus un bit care va avea rolul de a semnala prezența sau absența din memorie a paginii la care se referă.

Tabela de pagină în cazul în care anumite pagini nu se află în memoria principală

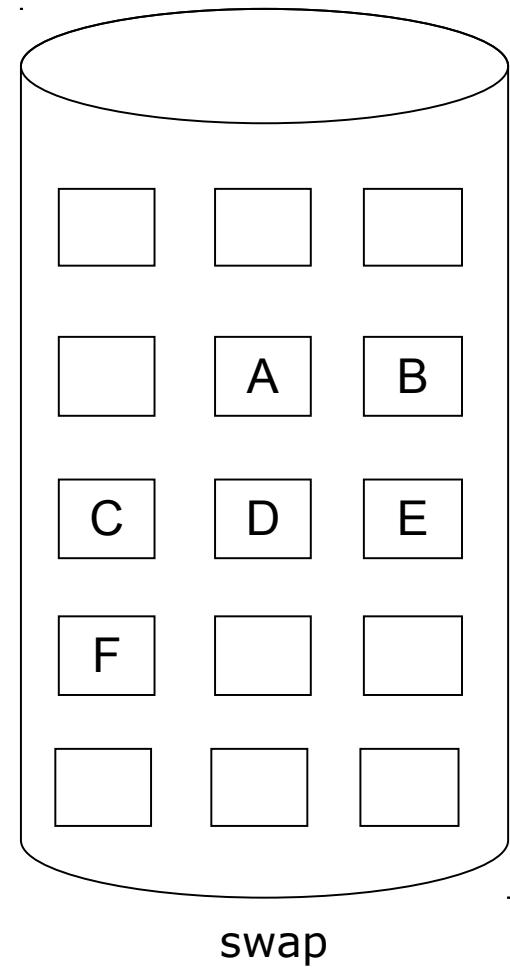
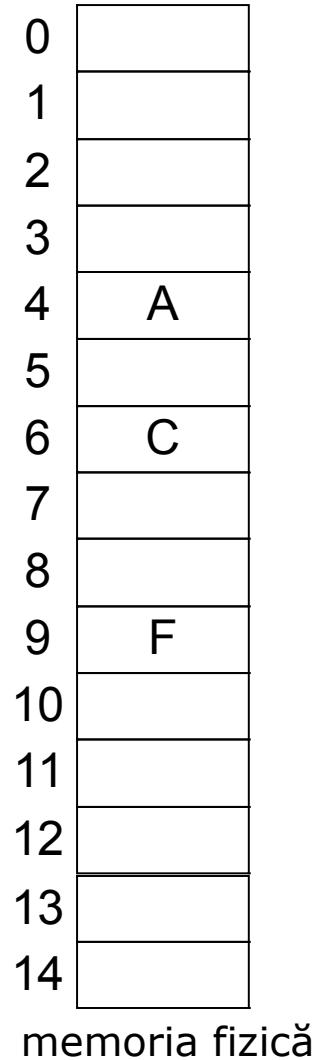
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

memoria logică

cadru Bit valid/invalid

0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

tabela de pagină



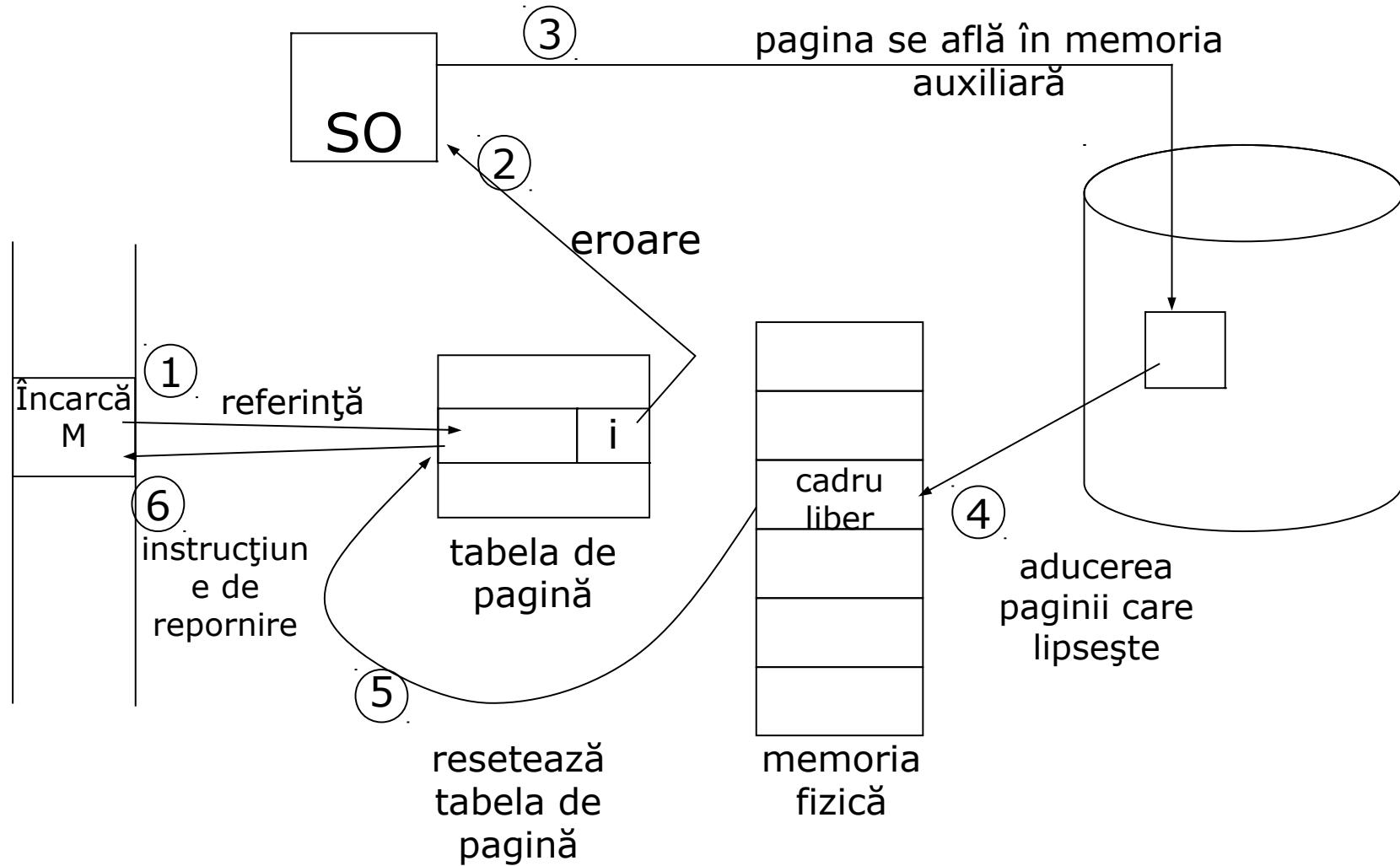
Eroarea de tip pagină lipsă

- Cât timp se folosesc paginile aflate în memorie, execuția se desfășoară normal.
- Dacă se încearcă folosirea unei pagini, care nu este încă în memorie, se va genera o **eroare de tip pagină lipsă**.
- Dacă se încearcă o folosire nepermisă a unei adrese de memorie (ex: indice incorrect de vector) se generează o eroare ce duce la încheierea forțată a programului.

Etapele de rezolvare ale unei greșeli de pagină

- se verifică o tabelă internă a procesului (păstrată în blocul de control) pentru a stabili dacă încercarea de accesare este permisă sau nu (apartine spațiului de adrese logice asociate procesului); dacă este nepermisă programul se termină.
- dacă adresa este corectă și pagina respectivă nu a fost adusă în memorie, trebuie încărcată din memoria auxiliară;
- se caută un cadru liber în memoria fizică
- se planifică discul pentru citirea paginii dorite în cadrul astfel alocat;
- odată cu încheierea operației de citire se actualizează atât tabela internă asociată procesului, cât și tabela de pagină, astfel încât să semnaleze apariția în memorie a noii pagini.
- se reia execuția instrucțiunii întrerupte de apariția erorii de adresare.

Etapele de rezolvare ale unei greșeli de pagină



Tratarea paginii lipsă

- generarea erorii către sistemul de operare;
- se salvează regiștrii utilizator și starea programului;
- se verifică dacă întreruperea este de tip pagină lipsă;
- se verifică dacă este corectă referirea paginii și se determină poziția ei pe disc;
- se inițiază citirea de pe disc într-un cadru disponibil al memoriei interne;
- se așteaptă în “coada” asociată discului până în momentul în care este satisfăcută cererea de citire;
- se așteaptă datorită timpului de căutare și/sau latență a discului;
- se începe transferul paginii în cadrul disponibil;
- în timpul așteptării, UC poate fi alocată unui alt utilizator (planificarea UC);
- se salvează registrele și starea programului celuilalt utilizator;
- se verifică dacă întreruperea a fost generată de către disc;
- se actualizează tabela de pagină și celelalte tabele astfel încât să reflecte existența în memorie a noii pagini;
- se așteaptă ca UC să fie alocată din nou procesului;
- se refac conținutul regisrelor utilizator, starea programului și noua tabelă de pagină, după care se reia execuția instrucțiunii întrerupte.

Planificarea schimburilor cu memoria (înlocuirea paginii)

- Datorită gradului ridicat de multiprogramare apare “supra-alocarea memoriei”:
 - în timpul rulării unui program apare o eroare de tip “pagină lipsă”
 - sistemul de operare verifică tabelele interne pentru stabilirea cauzei (pagină lipsă sau încercare de accesare nepermisă a memoriei),
 - încearcă localizarea paginii pe disc, dar în urma consultării listei de cadre disponibile constată că toată memoria este ocupată (nu există nici un cadru liber).

Planificarea schimburilor cu memoria (înlocuirea paginii)

- Se pot lua următoarele decizii:
 - încheierea execuției programului
 - evacuarea unui program pe disc (cu eliberarea cadrelor alocate – se reduce gradul de multiprogramare)
 - folosirea metodei de înlocuire a paginii.

Metoda înlocuirii paginii

- dacă nu este nici un cadru liber, se caută și se eliberează un cadru care nu este utilizat în acel moment prin memorarea pe disc a paginii conținute și se face modificarea tabelelor corespunzătoare pentru a indica faptul că pagina nu se mai află în memorie.
- cadrul eliberat este folosit pentru aducerea paginii care a fost solicitată în momentul apariției erorii de tip pagină lipsă.

Rutina de deservire a înlocuirii paginii

- ❑ localizarea paginii dorite pe discul magnetic;
- ❑ găsirea unui cadru liber;
- ❑ dacă există un cadru liber, el va fi folosit;
- ❑ altfel, cu ajutorul unui algoritm de înlocuire a paginii, se alege un cadru, se transferă pe disc pagina pe care o conține și se modifică tabelele de pagină și cadru;
- ❑ se aduce în cadrul astfel eliberat pagina dorită și se actualizează tabelele de pagină și cadru;
- ❑ se reia execuția programului utilizator.
- ❑ dacă nu există cadre libere sunt necesare două transferuri de pagină, ceea ce duce la dublarea timpului de deservire a erorii de tip pagină lipsă și la mărirea timpului efectiv de acces

Rutina de deservire a înlocuirii paginii

- dacă nu există cadre libere?
 - sunt necesare două transferuri de pagină, ceea ce duce la dublarea timpului de deservire a erorii de tip pagină lipsă și la mărirea timpului efectiv de acces
 - Pentru evitarea acestor efecte se asociază prin hardware fiecărei pagini un bit suplimentar, numit bit de modificare, care este modificat ori de câte ori este modificat conținutul paginii (prin scrierea unui cuvânt sau a unui octet).
 - Când o pagină este aleasă pentru a fi înlocuită, se verifică mai întâi bitul de modificare asociat, și dacă este setat înseamnă că în pagină a apărut cel puțin o modificare față de momentul în care a fost citită de pe disc. În acest caz, pagina trebuie salvată de pe disc cu noul ei conținut.
 - Dacă bitul de modificare nu este setat (pagina nu a fost modificată) și copia de pe disc nu a fost suprascrisă, se poate renunța la salvarea ei. Astfel timpul de I/O se poate reduce la jumătate și se diminuează durata de tratare a erorii de tip pagină lipsă.

Algoritmi de înlocuire a paginii

- Evaluarea unui algoritm se face prin executarea sa asupra unei anumite secvențe de referiri la memorie și prin calcularea numărului de erori de tip pagină lipsă apărute.
- Sirul de referiri la memorie se numește *sir de referință* și poate fi generat (cu un generator de numere aleatoare) sau poate fi înregistrat prin urmărirea unui anumit sistem.

Algoritmi de înlocuire a paginii (2)

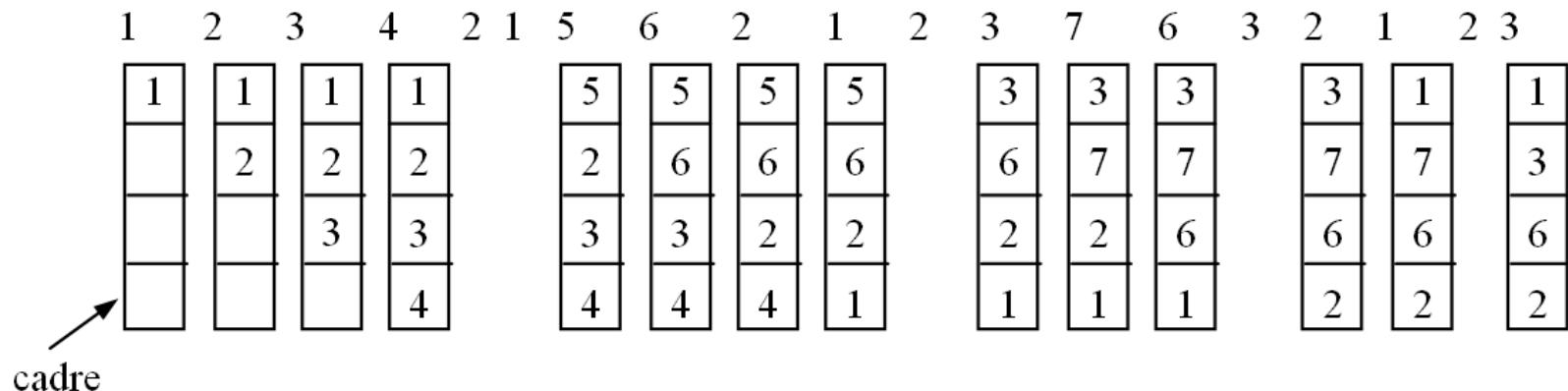
- Numărul de erori apărute în timpul execuției unui anumit algoritm de înlocuire a paginii asupra unui sir de referință depinde de numărul de cadre disponibile în memoria internă (dacă numărul cadrelor crește scade numărul erorilor).
- Pentru algoritmii care vor fi studiați în continuare, se consideră că avem o memorie cu 4 cadre și ca sirul de referință al adreselor este:
- 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

Algoritmul FIFO

- fiecărei pagini i se asociază momentul de timp la care a fost adusă în memorie și atunci când este necesară o înlocuire se alege pentru înlocuire cea mai “veche” pagină.
- O variantă des utilizată este crearea unei cozi FIFO în care se păstrează toate paginile aduse în memorie, iar pagina înlocuită va fi tot timpul la începutul cozii (noile pagini vor fi plasate la sfârșitul cozii).

Algoritmul FIFO

șir de referință



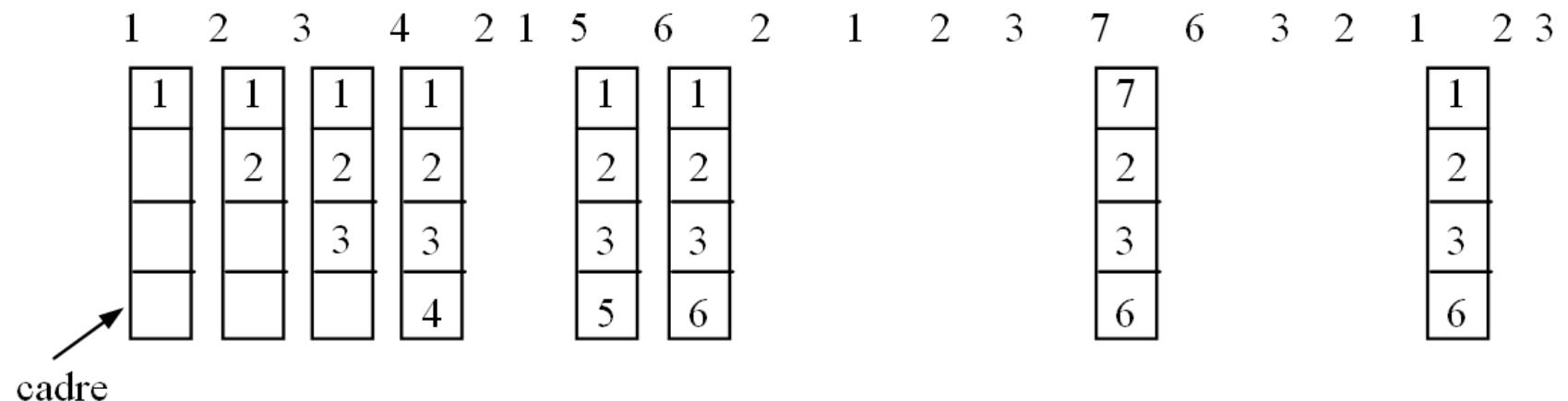
- FIFO este afectat de “**anomalia lui Belady**”:
 - este posibil ca **rata de apariție a erorilor să crească odată cu creșterea numărului de cadre alocate și nu să scadă cum ar fi fost normal.**
- În funcție de sirul de referință este posibil ca numărul de erori de tip pagină lipsă pentru 4 cadre disponibile sa fie mai mare decât numărul corespunzător pentru 3 cadre.

Algoritm de înlocuire optimală

- se alege pentru înlocuire pagina care a stat cel mai mult timp neutilizată (folosește momentul de timp la care este utilizată pagina, în timp ce FIFO folosește momentul la care a fost adusă pagina)
- Implementarea acestui tip de algoritm este dificilă deoarece trebuie cunoscut sirul de referință și, din acest motiv, este folosit numai pentru realizarea unor studii comparative.

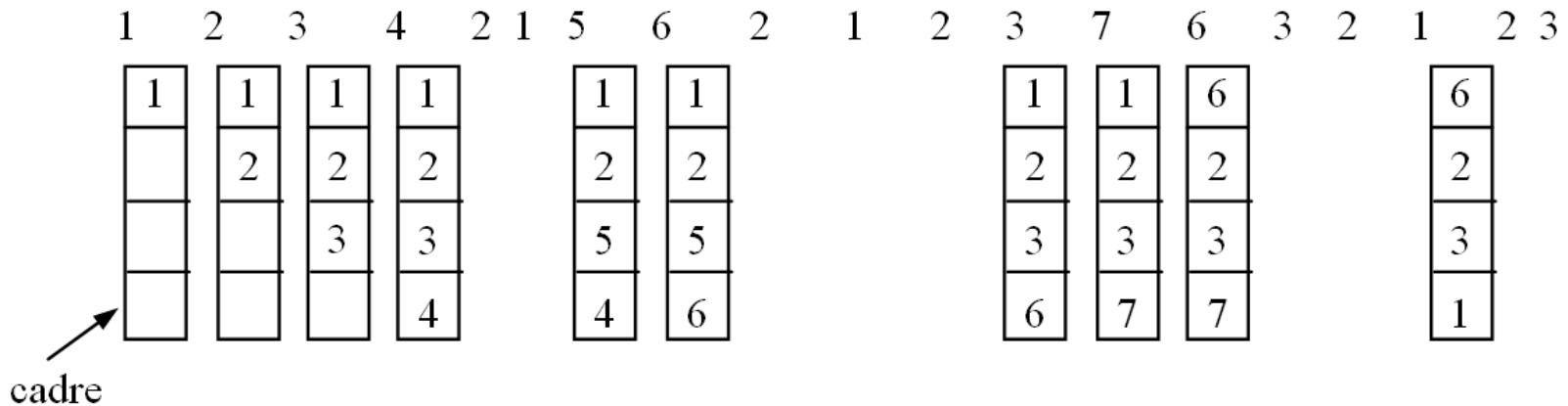
Algoritm de înlocuire optimă

șir de referință



Algoritm LRU (Last Recently Used)

șir de referință



- asociază fiecărei pagini momentul de timp al ultimei utilizări.
- algoritmul alege pentru înlocuire pagina cu cea mai lungă durată de neutilizare.

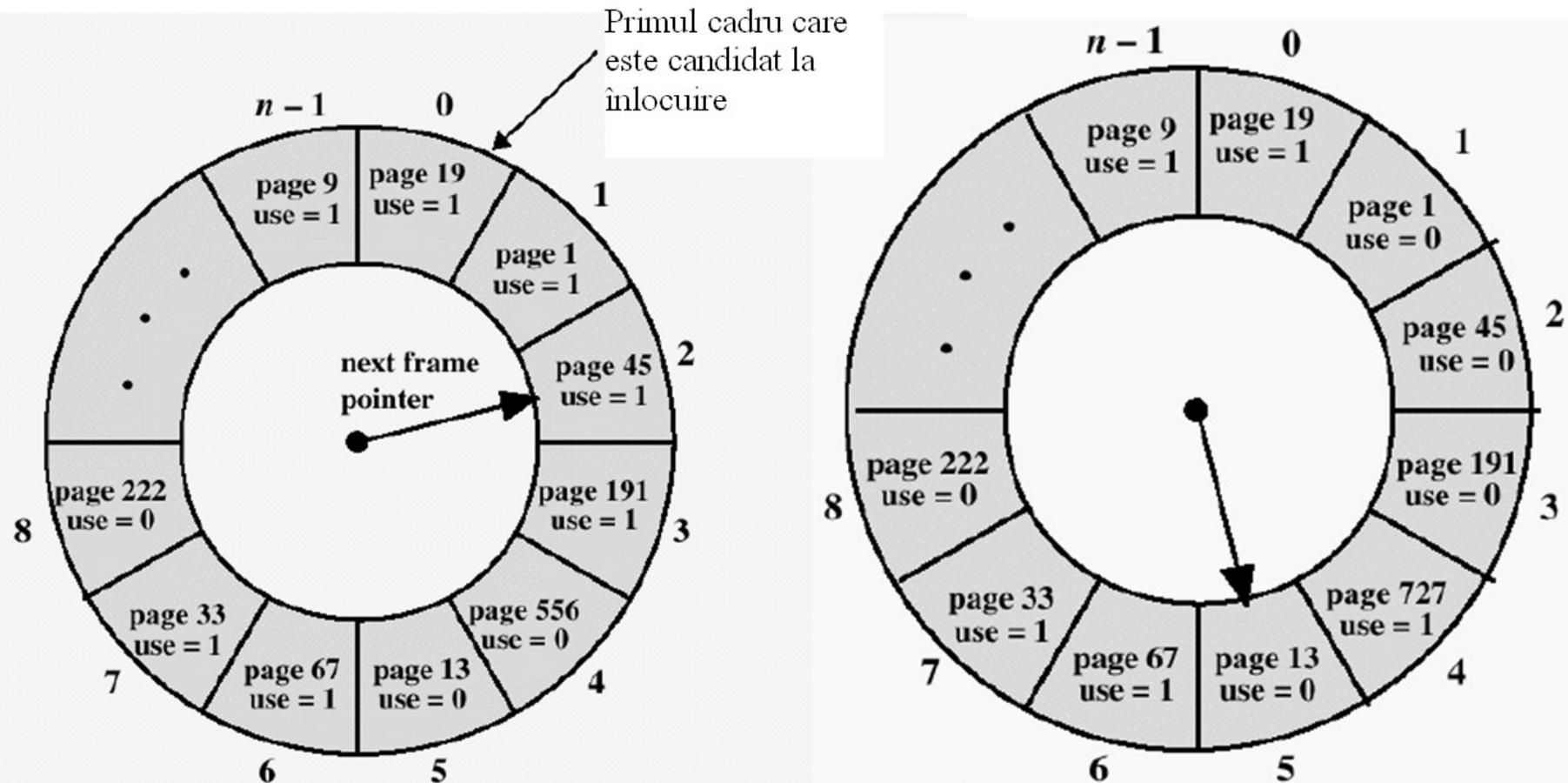
Algoritm LRU (Last Recently Used)

- Deoarece este un algoritm de înlocuire optimală a paginii care examinează trecutul și nu viitorul, este posibil ca o pagină să fie înlocuită chiar dacă ea urmează a fi referită din nou la momentul următor.
- Cu toate acestea, datorită numărului redus de erori de tip pagină lipsă, algoritm LRU este mult mai performant decât algoritmul FIFO.
- Algoritmul nu este afectat de anomalia lui Belady, deoarece paginile deja încărcate în memorie reprezintă cele mai recent utilizate n pagini și își păstrează caracteristica chiar și în cazul creșterii numărului de cadre utilizate.

Algoritmul “a doua sansă” (Clock Algorithm)

- Algoritmul de bază este FIFO.
- În momentul verificării stării unei pagini, se inspectează bitul de referire asociat:
 - Dacă este 0, pagina va fi înlocuită;
 - dacă are valoarea 1, i se dă paginii o a doua sansă (bitul de referire asociat primește valoarea 0, iar timpul de sosire în memorie primește valoarea momentului curent), trecându-se (în ordinea FIFO) la verificarea stării următoarei pagini.
 - O pagină căreia i s-a acordat a doua sansă rămâne în memorie până în momentul în care toate celelalte pagini au fost înlocuite (sau au primit o a doua sansă).
- Dacă o pagină este folosită destul de des pentru ca valoarea bitului să rămână 1, ea nu va fi înlocuită niciodată.
- Dacă toți biții au valoarea 1, algoritmul se transformă în unul de tip FIFO.

Algoritmul “a două sansă” (Clock Algorithm)



Algoritmul LFU

(Least Frequently Use – cea mai puțin frecvent utilizată):

- Acest algoritm înregistrează numărul de referiri corespunzătoare fiecărei pagini.
- Algoritmul va selecta pentru înlocuire pagina care are asociat cel mai mic număr de referiri.
- Principalul dezavantaj al acestei metode se observă în cazul în care la începutul programului, o pagină este intens folosită și apoi nu mai este utilizată deloc, ea rămâne în memorie, deoarece numărul de referiri este foarte mare.

Algoritmul MFU (Most Frequently Use – cea mai frecvent utilizată):

- Acest algoritm consideră că pagina care are asociat cel mai mic număr de referiri este probabil cea mai recent introdusă în memorie și este posibil ca programul să o folosească în continuare.
- Implementarea algoritmilor LFU și MFU este foarte costisitoare și din acest motiv sunt mai puțin folosiți.

Clasificarea paginilor

- În funcție de valorile pe care biții de referire și de modificare le pot lua, paginile se pot clasifica în:
 - (0,0) nefolosite și nemodificate;
 - (0,1) nefolosite (recent), dar modificate;
 - (1,0) folosite, dar nemodificate;
 - (1,1) folosite și modificate.

Algoritmi de înlocuire a paginii

- alte tehnici

- Algoritmii de înlocuire a paginii pot fi utilizati împreună cu alte proceduri:
- sistemul păstrează o rezervă de cadre libere:
 - Când apare o eroare de tip pagină lipsă, se alege un cadru , dar, înainte ca acesta să fie evacuat din memorie, se citește pagina dorită într-un cadru liber.
 - Procedura permite programului să-și reînceapă execuția cât mai repede, fără a mai aștepta ca pagina aleasă să fie scrisă pe disc (după realizarea acestei operații, cadrul asociat paginii alese pentru înlocuire este atașat rezervei de cadre)

Algoritmi de înlocuire a paginii

- alte tehnici

- folosirea unei liste a paginilor modificate:
 - ori de câte ori dispozitivul de paginare este inactiv, se alege una dintre paginile modificate pentru a fi scrisă pe disc, după care i se setează bitul de modificare asociat.
 - Această “evacuare preventivă” face inutilă evacuarea paginii în momentul în care ea va fi aleasă pentru a fi înlocuită, crescând performanțele sistemului.

Algoritmi de înlocuire a paginii

- alte tehnici

- folosirea unei rezerve de cadre libere și a unor informații care să precizeze ce pagină corespunde fiecărui cadru înainte ca acesta să fie inclus în rezervă.
 - Deoarece conținutul cadrului nu se modifică în urma scrierii pe disc, vechea pagină poate fi reutilizată direct din rezerva de cadre libere (dacă este necesar), fără folosirea altor operații de I/O (acest lucru este posibil doar dacă pagina respectivă nu a fost utilizată pentru memorarea altei pagini).

Alocarea cadrelor

- Indiferent de varianta de alocare a memoriei, pentru un sistem monoutilizator, unui program i se alocă oricare dintre cadrele disponibile.
- În cazul multiprogramării combine cu paginarea la cerere, problemele de alocare trebuie să ia în considerare următoarele constrângeri:
 - nu pot fi alocate mai multe cadre decât există (cu excepția cazului paginilor folosite în comun);
 - dacă numărul cadrelor alocate fiecărui proces este prea mic, rata de apariție a erorii de tip pagină lipsă crește;
 - arhitectura sistemului de calcul poate impune asigurarea unui număr minim de cadre alocate.

Metode de alocare

□ Metoda primei potriviri (First-fit)

- memoria solicitată este alocată în prima zonă în care începe; principalul avantaj este simplitatea căutării de spațiu liber.

□ Metoda celei mai bune potriviri (Best-fit)

- se caută acea zonă liberă care după alocare lasă cel mai puțin spațiu liber.

- Avantaj: economisește zonele de memorie.

- Dezavantaj:

- timp suplimentar de căutare și proliferarea blocurilor libere de lungime mică (fragmentare internă excesivă)
- poate fi eliminat parțial dacă lista de spații libere este păstrată nu în ordinea crescătoare a adreselor, ci în ordinea crescătoare a lungimii spațiilor libere.

Metode de alocare

□ Metoda celei mai rele potriviri (Worst-fit)

- se caută zonele libere care după alocare lasă cel mai mult spațiu liber.
- Deși fragmentarea internă nu evoluează foarte rapid, timpul de căutare este mai mare decât cel de la metoda primei potriviri.
- Și aici este posibil ca lista spațiilor libere să nu fie păstrate în ordinea crescătoare a adreselor, ci în ordinea crescătoare a lungimii spațiilor libere.

Metode de alocare

□ Algoritmul Buddy-system

- această metodă exploatează reprezentarea binară a adreselor și faptul că din rațiuni tehnologice, dimensiunea memoriei interne este un multiplu al unei puteri a lui doi.
- Notăm cu **n** cea mai mare putere a lui 2 prin care se poate exprima dimensiunea memoriei interne și cu **m** puterea lui 2 care definește unitatea de alocare a memoriei.
- Dimensiunea spațiilor ocupate și a celor libere sunt de forma **2^k** , unde **$m \leq k \leq n$** .
- Ideea principală este de a păstra liste separate de spații libere pentru fiecare dimensiune **2^k** .
- Astfel, vom avea **$n-m+1$** liste de spații disponibile. Astfel, fiecare spațiu liber sau ocupat de dimensiune **2^k** are adresa de început un multiplu de **2^k** .

Metode de alocare

Algoritmul Buddy-system

- Definiție:
 - două spații libere de ordinul k se numesc camarazi (Buddy) de ordin k , dacă adresele lor A_1 și A_2 verifică relațiile:
 - $A_1 < A_2$, $A_2 = A_1 + 2^k$ și $A_1 \bmod 2^{k+1} = 0$,
 - sau
 - $A_2 < A_1$, $A_1 = A_2 + 2^k$ și $A_2 \bmod 2^{k+1} = 0$.
 - dacă într-o listă de ordin k apar doi camarazi, sistemul îi concatenează într-un spațiu de dimensiune 2^{k+1} .

Metode de alocare

Algoritmul Buddy-system

- Algoritmul de alocare este următorul:
 - se determină cel mai mic număr **p**, $m \leq p \leq n$ pentru care numărul de octeți solicitați verifică relația $o \leq 2^p$.
 - se caută, în această ordine, în liste de ordin **p**, **p+1**, **p+2**, ... **n** o zonă liberă, de dimensiune cel puțin **o**.
 - dacă se găsește o zonă de ordin **p**, atunci aceasta este alocată și se șterge din lista de ordinul **p**.
 - dacă se găsește o zonă de ordin **k>p**, atunci se alocă primii **2^p** octeți, se șterge zona din lista de ordin **k** și se creează, în schimb, alte **k-p** zone libere cu dimensiunile: **2^p**, **2^{p+1}**, ..., **2^{k-1}**.

Metode de alocare

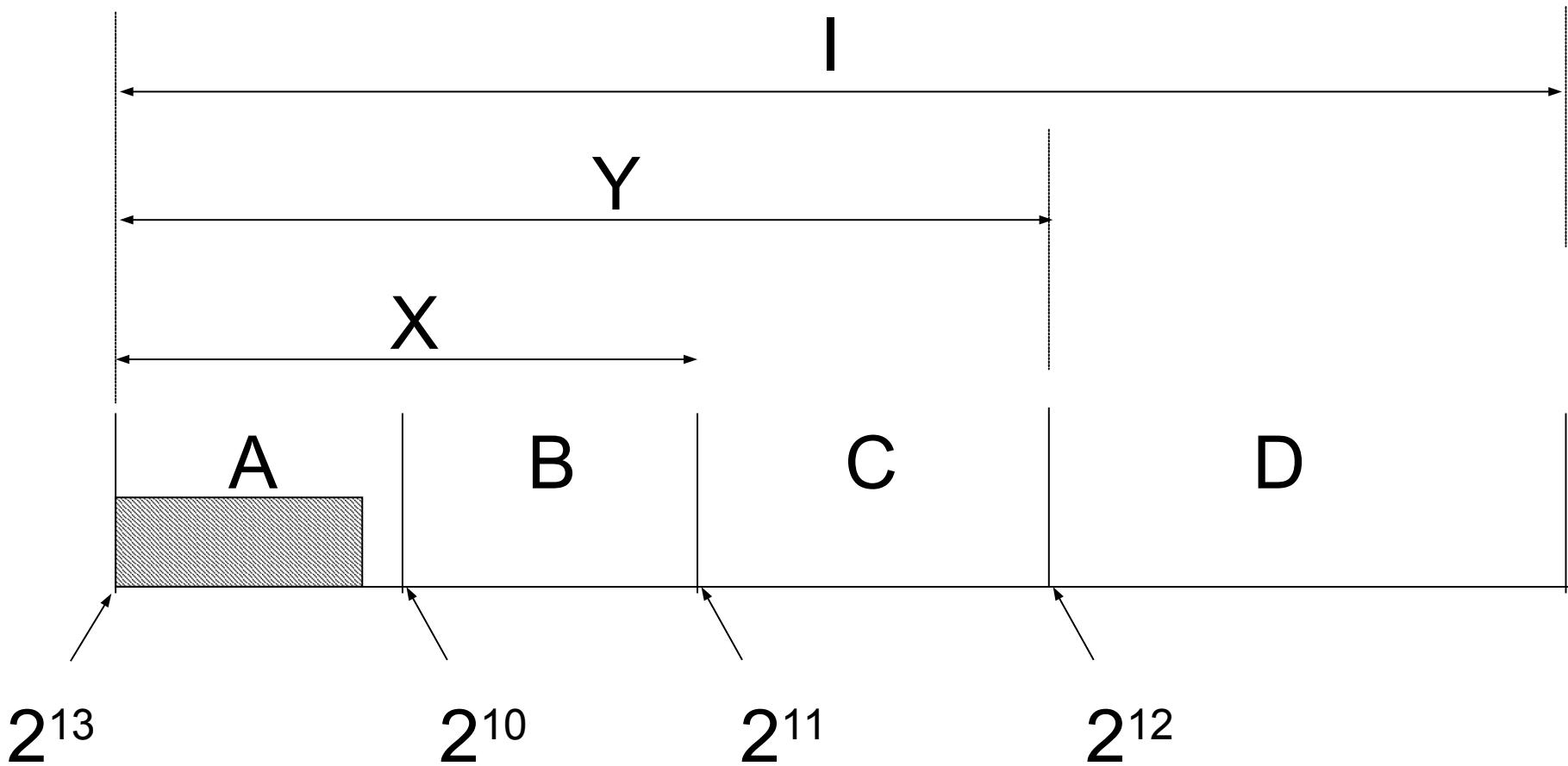
Algoritmul Buddy-system

- Se dorește alocarea a 1000 octeți ($p = 10$)
- Nu s-au găsit zone libere nici de dimensiune 2^{10} , nici 2^{11} și nici 2^{12} .
- Prima zonă liberă este de dimensiune 2^{13} și o notăm cu **I**.
- Ca rezultat al alocării a fost ocupată zona **A** de dimensiune 2^{10} și au fost create încă trei zone libere:
 - **B** de dimensiune 2^{10}
 - **C** de dimensiune 2^{11}
 - **D** de dimensiune 2^{12}
- Zonele **B**, **C** și **D** vor trece în liste de ordine **10**, **11** și **12**, iar zona **I** va fi ștearsă din lista de ordin **13**.

Metode de alocare

Algoritmul Buddy-system

□ Alocarea și eliberarea memoriei



Metode de alocare

Algoritmul Buddy-system

- Algoritmul de eliberare a memoriei:
 1. se introduce zona respectivă în lista de ordin **p**.
 2. se verifică dacă zona eliberată are un camarad de ordin **p**.
 - Dacă da, atunci zona este comasată cu acest camarad și formează împreună o zonă liberă de dimensiune **2^{p+1}** .
 - Atât zona eliberată, cât și camaradul ei se sterg din lista de ordin **p**, iar zona nou apărută va trece în lista de ordin **p+1**.
 3. Se execută pasul 2 în mod repetat, mărind de fiecare dată **p** cu o unitate, până când nu se mai pot face comasări.

Metode de alocare

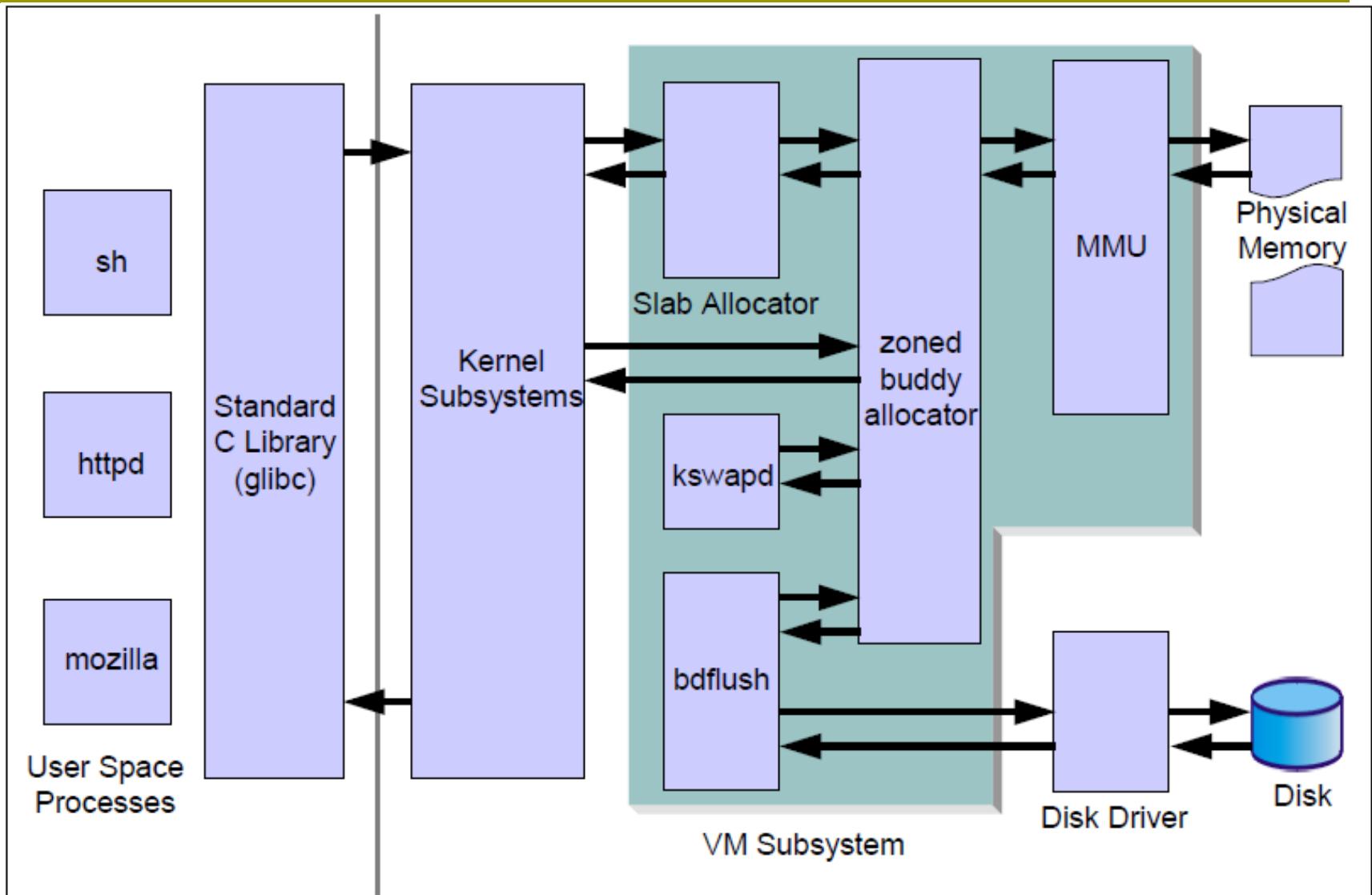
Algoritmul Buddy-system

□ Exemplu:

- Presupunem că sunt libere zonele A, C și D, iar zona B este ocupată.
- Se eliberează zona B și se execută pașii:
 - se trece zona B în lista de ordin 10;
 - se depistează că zonele A și B sunt camarazi. Ca urmare, cele două zone vor fi comasate și se va forma o zonă X care va fi trecută în lista de ordin 11, iar zonele A și B se sterg din lista de ordin 10.
 - se depistează că zonele X și C sunt camarazi. Ca urmare ele vor fi comasate, formând zona Y, care va fi trecută în lista de ordin 12. X și C vor fi sterse din lista de ordin 11.
 - se găsește că Y și C sunt camarazi și vor fi comasate, formând zona I, ce va fi trecută în lista de ordin 13, iar Y și C vor fi sterse din lista de ordin 12.

Studiu de caz

Linux - Memoria virtuală



Studiu de caz

Linux - Memoria virtuală

- Adresarea memoriei virtuale:
- Sistemul de operare Linux folosește o structură de tabelă cu trei niveluri ce constau în (fiecare tabela are dimensiunea unei pagini):
 - **Page directory**: un proces activ are o singură pagină director care este de dimensiunea unei pagini. Fiecare intrare în pagina director pointează către o pagină din page middle directory. Pagina director trebuie să fie în memoria principală pentru un proces activ.
 - **Page middle directory**: fiecare intrare pointează către o pagină din tabela de pagini.
 - **Page table**: fiecare intrare pointează către o pagină virtuală a procesului.

Studiu de caz

Linux - Memoria virtuală

- o adresă virtuală în Linux este văzută ca având patru câmpuri:
 - cel mai semnificativ (din stânga) este utilizat ca index în page directory;
 - următorul este index în page middle directory;
 - al treilea câmp este index în page table
 - ultimul este offset-ul în pagina de memorie selectată.

Studiu de caz

Linux - Memoria virtuală

- ❑ Această structură este independentă de platformă și a fost proiectată pentru procesoarele Alpha de 64 biți care oferă suport pentru 3 niveluri de paginare.
- ❑ Pentru procesoarele Pentium pe 32 biți și x86 care au implementat un mecanism hardware de paginare cu două niveluri, Linux definește dimensiunea **page middle directory** ca fiind 1.

Studiu de caz

Linux - Alocarea paginilor

- este folosit algoritmul Buddy.
- Kernelul menține o listă care conține un grup de pagini contigüe de dimensiune fixă iar fiecare grup poate consta în 1, 2, 4, 8, 16 sau 32 pagini.
- Pe măsură ce paginile sunt alocate și eliberate din memorie, grupurile disponibile sunt împărțite și regrupate folosind algoritmul buddy.

Studiu de caz

Linux - Înlocuirea paginii

- ❑ Algoritmul folosit de Linux pentru înlocuirea paginii este bazat pe algoritmul "a două sănsă" (clock algorithm).
- ❑ În cazul Linux, bitul suplimentar de utilizare este înlocuit cu o variabilă de 8 biți.
- ❑ De fiecare dată când o pagină este accesată, această variabilă este incrementată.
- ❑ În paralel, Linux parcurge lista paginilor și decrementează variabilele pentru fiecare pagină.
- ❑ Dacă o pagină are variabila egală cu 0, atunci este considerat ca fiind cel mai bun candidat pentru înlocuire.
- ❑ O valoare mare a variabilei indică faptul că pagina este folosită frecvent și este mai puțin luată în considerare la o posibilă înlocuire. De aceea, algoritmul folosit de Linux este considerat ca fiind o formă de LFU (Least Frequently Used).

Studiu de caz

Linux - Alocarea memoriei pentru kernel

- Alocarea memoriei pentru kernel este bazată pe mecanismele de alocare a memoriei virtuale utilizator.
- Este folosit algoritmul Buddy pentru a aloca sau elibera unități de una sau mai multe pagini.
- Deoarece cantitatea de memorie minimă alocată în acest mod este de o pagină, algoritmul va fi ineficient deoarece kernelul are nevoie de zone de memorie de dimensiune impară pentru un timp foarte scurt.
- Pentru rezolvarea acestei probleme, Linux foloseste *slab allocation* (allocare pe bucăți (plăci) sau pe porțiuni) în cadrul unei pagini alocate.
- Pentru arhitectura Pentium/x86, pagina este de 4 kbytes iar bucățile alocate pot fi de 32, 64, 128, 252, 508, 2040 sau 4080 bytes. În esență, Linux păstrează un set de liste, câte una pentru fiecare bucătă memorie. Zonele alocate pot fi gestionate după modelul algoritmului Buddy.

Studiu de caz

Unix / Solaris

- Deoarece este un sistem de operare independent de platformă, managementul memoriei variază de la un sistem la altul.
- Primele versiuni de UNIX nu foloseau memorie virtuală.
- Implementările curente, inclusiv SVR4 și Solaris 2.x, folosesc memorie virtuală paginată.
- De fapt, în SVR4 și Solaris 2.x, sunt folosite două scheme de management a memoriei.

Studiu de caz

Unix / Solaris

- Sistemul cu paginare oferă posibilitatea memoriei virtuale de a aloca pagini atât în memoria principală pentru procese, cât și alocarea paginilor pentru bufferele blocurilor de pe disc.
- Deși este o schemă eficientă de alocare atât pentru procesele utilizator, cât și pentru operațiile de I/O, este mai puțin potrivită pentru alocarea memoriei kernelului.

Studiu de caz

Unix / Solaris - Alocarea paginilor

- **Disk block descriptor:** asociată fiecărei pagini a procesului o intrare această tabelă descrie copia de pe disc a memoriei virtuale.

Swap device number	Device block number	Type of storage
--------------------	---------------------	-----------------

- **Page frame data table:** Descrie fiecare zonă din memoria principală și este indexată după numărul zonei.

Page state	Reference count	Logical device	Block number	Pfdata pointer
------------	-----------------	----------------	--------------	----------------

Studiu de caz

Unix / Solaris - Alocarea paginilor

- **Swap-use table:** Este folosită câte o tabelă de acest tip pentru fiecare dispozitiv de păstrare a swap-ului, cu câte o intrare pentru fiecare pagină.

Reference count	Page/storage unit number
-----------------	--------------------------

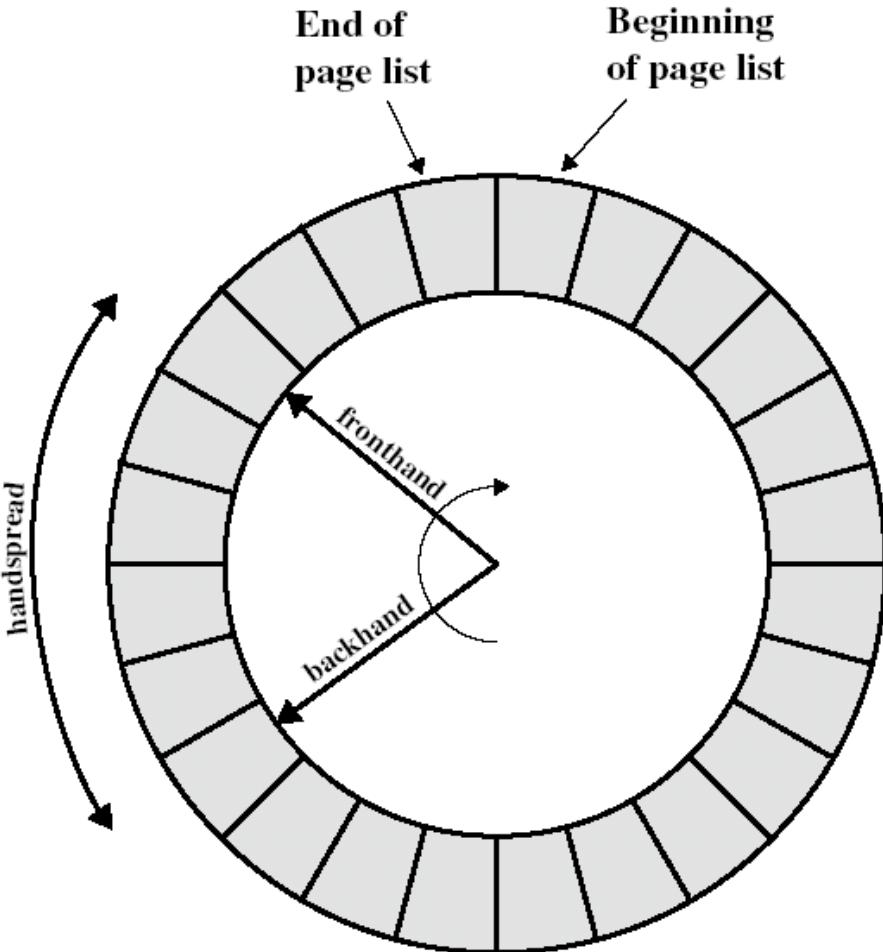
Studiu de caz

Unix / Solaris - Înlocuirea paginilor

- ❑ Structura page frame data table este folosită pentru înlocuirea paginilor.
- ❑ Algoritmul folosit în cazul SVR4 este o variantă a algoritmului “a doua şansă”.
- ❑ Algoritmul folosește bitul de referință pentru fiecare pagină care poate fi trecută în swap.
 - Bitul este setat pe 0 când este adusă în memorie prima dată și este setat pe 1 când pagina este folosită pentru citire sau scriere.
- ❑ Pe de altă parte, are loc o parcursere în sens invers pentru verificarea bitului de referință.
 - Dacă este 1, atunci pagina respectivă este ignorată, iar dacă este 0 atunci înseamnă că pagina nu a fost referită între cele două parcurgeri și este posibil să fie înlocuită.

Studiu de caz

Unix / Solaris - Înlocuirea paginilor



- Doi parametri caracterizează funcționarea acestui algoritm:
 - Scanrate: numărul de treceri peste lista de pagini în cele două sensuri, măsurat în pagini pe secundă.
 - Handspread: intervalul dintre cele două parcurgeri.
- Acești doi parametri au valori diferite la bootare în funcție de memoria disponibilă.
- Împreună determină durata de utilizare a unei pagini înainte de a fi trecută în swap datorită faptului că nu este utilizată.

Unix / Solaris - Alocarea memoriei pentru kernel

- Kernel-ul alocă și eliberează tabele mici și buffer-ele în timpul execuției în următoarele cazuri:
 - **pathname translation**: poate aloca un buffer pentru a copia path-ul din spațiul utilizator;
 - funcția **allocb()** alocă buffer-ele de dimensiune variabilă;
 - multe implementări de UNIX alocă aşa numitele **structuri zombie** pentru a păstra informațiile de ieșire ale unui proces terminat;
 - în SVR4 și Solaris, kernelul alocă diverse obiecte (structuri **proc**, **vnodes** și **file block descriptor**) dinamic atunci când are nevoie;
- este folosit un **algoritm Buddy modificat**: costul alocării unui bloc liber de memorie trebuie să fie mai mic decât costul alocării folosind algoritmii **first-fit** sau **best-fit**

Studiu de caz

Windows 2000

- La Windows 2000, pentru folosirea memoriei virtuale, se poate adresa un spațiu de adrese pe 32 de biți.
- fiecare proces poate accesa 2GB pentru el și 2GB din spațiul partajat cu celelalte procese.
- Paginarea la Windows 2000:
 - o pagină poate fi în 3 stări:
 - available: nu este folosită curent de proces;
 - reserved: rezervată, dar nu este luată în calcul pentru aflarea memoriei ocupate de proces;
 - committed: paginile pentru care managerul memoriei virtuale a rezervat spațiu în tabela de pagini.

Studiu de caz

Windows 2000

- Dacă apare o eroare de tip pagină lipsă, este selectată o pagină din setul de pagini rezervate pentru alocare.
- Dacă memoria este suficientă, este permisă creșterea zonelor rezervate pe măsură ce paginile sunt încărcate în memorie.
- Dacă este puțină memorie disponibilă, sunt evacuate din setul paginilor rezervate paginile cel mai puțin folosite.

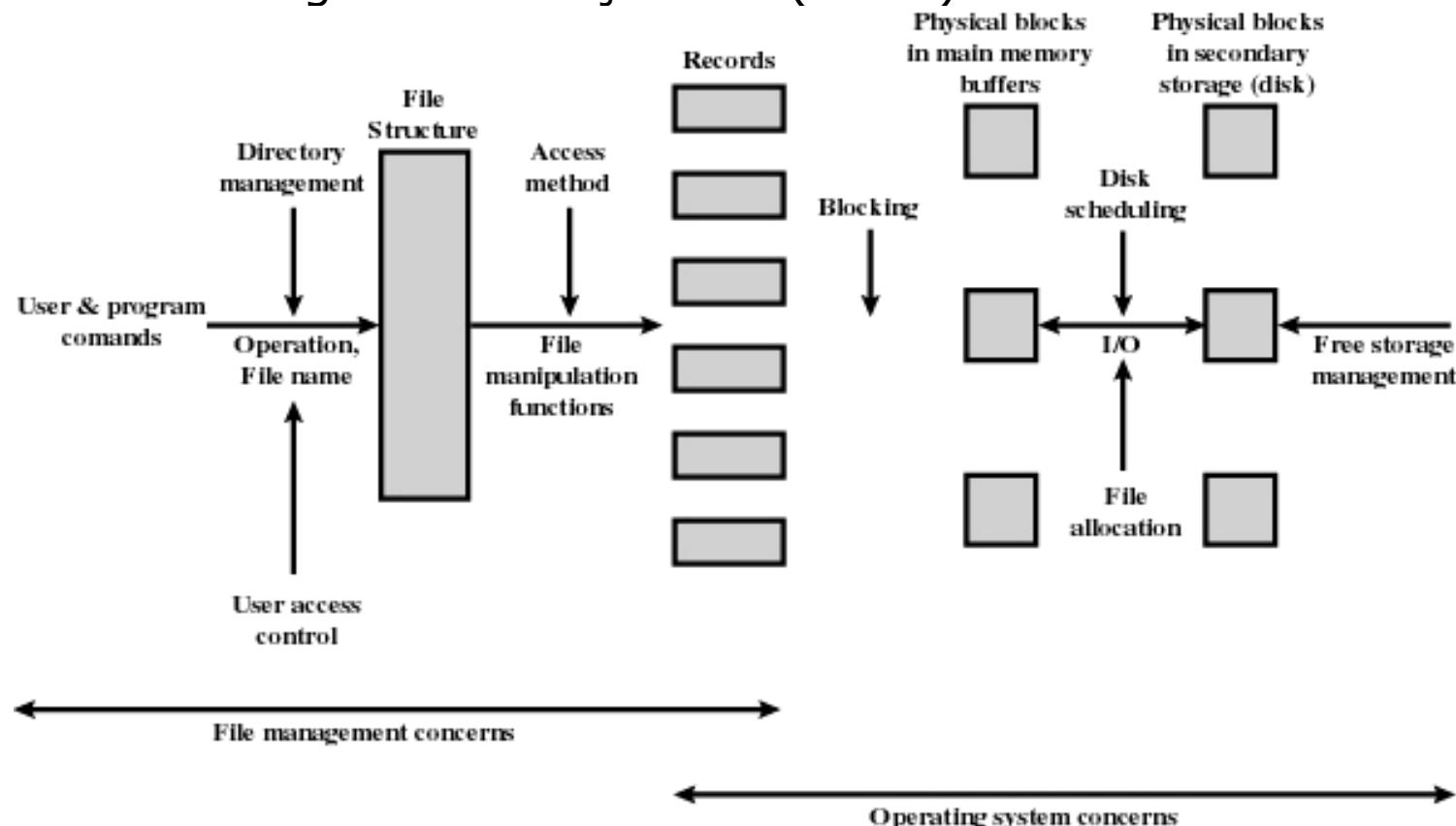
Sisteme de Operare

□ Gestiunea sistemului de fișiere

- Acțiunile SGF la nivel de fișier
 - Conceptul de fișier
 - Categorii de fișiere
 - Descriptorul de fișier
 - Moduri de organizare ale fișierelor
- Acțiunile SGF la nivel de suport disc
- Sisteme de directoare (cataloage)
- Alocarea spațiului pentru fișiere pe disc
- Evidența spațiului liber de pe disc

Gestiunea sistemului de fișiere

- Se face în mod transparent pentru utilizator prin intermediul sistemului de gestiune a fișierelor (SGF)



Acțiunile SGF la nivel de fișier

- Una din principalele funcții ale sistemului de operare este implementarea unei “mașini virtuale”, concretizate în cazul fișierelor prin asigurarea unor primitive de manipulare a fișierelor la un nivel cât mai abstract, care să nu implice cunoștințe legate de modul de stocare
- Utilizatorul manipulează fișierele ca pe o secvență contiguă de octeți în cadrul căreia se poate poziționa oriunde, poate citi sau scrie, chiar dacă alocarea spațiului pe disc se face la nivel de blocuri (de regulă, de dimensiune fixă cuprinsă între 512 octeți și 4k octeți).

Acțiunile SGF la nivel de fișier

- Operațiile realizate de un sistem de fișiere se referă la:
 - denumirea și manipularea fișierelor, accesibile utilizatorilor ca entități cu nume și asupra cărora se pot efectua operații (citire, scriere, execuție);
 - asigurarea persistenței datelor care presupune o independentă față de crearea sau distrugerea proceselor din sistem (regăsirea după intervale mari de timp), posibilitatea refacerii structurii și conținutului în caz de accidente;
 - asigurarea mecanismelor de acces concurrent al proceselor din sistem la informațiile stocate.

Conceptul de fișier

- Informațiile sunt grupate de către SO în ansambluri distincte numite fișiere.
- Prin intermediul programelor, un utilizator are acces la un moment dat la o mică entitate din cadrul unui fișier, numită și articol sau înregistrare
 - este dificilă definirea exactă a unui articol în cadrul unui fișier

Categorii de fișiere din punct de vedere al structurii

- secvența de octeți:
 - fișierul se prezintă ca o secvență de octeți a cărei interpretare este lăsată la latitudinea programelor utilizator; oferă un grad maxim de flexibilitate în reprezentarea datelor (UNIX, Windows etc.);
- secvența de înregistrări:
 - fișierul este organizat ca o succesiune de înregistrări de lungime fixă;
 - operațiile de citire și scriere se fac la nivel de înregistrare (a apărut datorită cartelelor perforate și nu mai sunt folosite în prezent);
- fișiere cu structură arborescentă:
 - fișierul este format din înregistrări, care conțin atât informația propriu-zisă, cât și un câmp cheie (situat pe o poziție prestabilită în cadrul fiecărei înregistrări).
 - Valoarea câmpului cheie este folosită de către sistemul de operare pentru gestionarea unei structuri logice interne de tip arbore a fișierelor.

Categorii de fișiere din punct de vedere al tipului

- ❑ fișiere normale (regular files):
 - conțin informații utilizator;
- ❑ directoare (directories):
 - fișiere sistem destinate gestionării structurii sistemului de fișiere;
- ❑ fișiere speciale de tip caracter/bloc (character/block special files):
 - destinate utilizării în conjuncție cu dispozitivele periferice;
- ❑ legături simbolice (symbolic links):
 - furnizează posibilitatea accesării unui același fișier fizic pe căi logice multiple.

Fișierele normale

❑ ASCII:

- fișierul este format din linii de text terminate cu caracterele de control CR (cariage return) și/sau LF (line feed);

❑ binare:

- fișierul este organizat ca secvență de octeți, dar căruia sistemul de operare îi asociază o anumită structură internă (fișierele executabile și arhivele).

Descriptorul de fișier

- Informațiile de descriere a unui fișier sunt conținute într-un articol special numit **descriptor de fișier**.
- De regulă, acesta nu este memorat la un loc cu articolele de informație ale fișierului, ci în director.

Descriptorul de fișier

- conține patru grupe de informații:
 - identificatorul fișierului
 - informațiile privind adresele fizice disc pe care le ocupă
 - informații de control al accesului la fișier
 - informații de organizare și calendaristice

Descriptorul de fișier

□ identificatorul fișierului:

- este compus dintr-o pereche (**N, i**)
 - **N** este numele simbolic al fișierului
 - **i** este un număr prin care descriptorul este reperat pe disc în mod direct
 - este numit în diverse moduri: **i-node** (UNIX), file handle (MSDOS).
- Prin **N**, SGF realizează legătura cu utilizatorul și prin **i** cu celealte componente ale SO.
- Mai este folosit și de utilitarele de refacere a unei structuri de fișiere compromise.

Descriptorul de fișier

- informațiile privind adresele fizice disc pe care le ocupă:
 - aici vor fi memorate informațiile necesare regăsirii articolelor fișierului pe disc.
- informații de control al accesului la fișier:
 - aici sunt precizate informațiile legate de cine și ce drepturi are pentru a accesa fișierul.
 - tot aici sunt trecute informații legate de reglementarea accesului simultan la fișier.

Descriptorul de fișier

- informații de organizare și calendaristice:
 - Aici se află informații legate de:
 - modul de organizare al fișierului (secvențial, secvențial-indexat, înlăncuit, etc);
 - formatul articolelor (fix, variabil, fișier text și standardul de codificare);
 - data și ora creării (ultimei actualizări, ultimei citiri),
 - numărul total de consultări a fișierului;
 - dispoziții de păstrare a fișierului (permanent, temporar, păstrabil un număr de zile etc...)

Operațiile asigurate de SGF

□ prin functii de bibliotecă:

- **create**: crearea unui nou fișier;
- **delete**: ștergerea fișierului;
- **open**: indică SO intenția unui proces de a accesa date conținute într-un fișier existent;
- **read**: citirea din fișier a unui număr specificat de octeți, începând de la poziția curentă, în spațiul de adresare al procesului care a inițiat operația;
- **write**: înscrierea în fișier a unui număr specificat de octeți începând de la poziția curentă, cu date conținute în spațiul de adresare al procesului care a inițiat operația;
- **append**: adăugarea de date la sfârșitul unui fișier existent;
- **close**: eliberarea structurilor de date sistem, ca urmare a terminării operațiilor pe care un proces le-a executat asupra unui fișier;
- **seek**: setează poziția curentă în fișier;
- **rename**: redenumește un fișier existent;
- **setattrives, getatributes**: operații de setare și citire a atributelor asociate unui fișier.

□ asociate unor comenzi:

- copierea, concatenarea, compararea, listarea, sortarea (ordonarea).

Operațiile asigurate de SGF

□ **open**

- informează SO că un anume fișier va deveni activ.
- Prima sarcină a funcției de deschidere a unui fișier este de a face legătura între informațiile de identificare a fișierului și variabila din program prin care utilizatorul se referă la fișier.

Operațiile asigurate de SGF

- **open** - în funcție de condițiile de deschidere

- A Pentru un fișier care există deja pe disc:**

- Realizează legătura dintre variabila din program ce indică fișierul, suportul pe care se află fișierul și descriptorul de fișier aflat pe disc.
 - Verifică, pe baza informațiilor de acces, dacă utilizatorul are sau nu drept de acces la fișier.

- B Pentru un fișier nou, ce urmează a fi creat:**

- Alocă spațiul pe disc pentru memorarea viitoarelor articole ale fișierului. În funcție de tehnica de alocare folosită, se poate aloca începând de la o singură unitate de alocare (sector, bloc, cluster) și terminând cu cantitatea maximală de spațiu estimată de utilizator pentru fișierul în cauză.

- C Atât la fișier nou, cât și la fișier deja existent:**

- Alocă memorie internă pentru zonele tampon necesare accesului la fișier.
 - Încarcă rutinele de acces la articolele fișierului și execută instrucțiunile lor de inițializare.
 - Generează o formă cadru a instrucțiunilor de comandă a canalului I/O.

- D Memorează unele dintre datele obținute la A, B, C și cele din descriptorul fișierului într-o structură de date în memoria internă cunoscută sub numele de bloc de control al fișierului (File Control Bloc sau FCB). De regulă fiecare utilizator are pentru fiecare fișier, propriul sau FCB (la unele SO este în zonele gestionate de sistem sau în monitorul sistemului)**

Operațiile asigurate de SGF

□ **close**

- informează SO că un fișier încetează a mai fi activ.
- Unele SO cer anunțarea explicită a închiderii, altele efectuează automat închiderea la sfârșitul programului.
 - La închidere, se reactualizează informațiile generale despre fișier (lungime, adrese de început și sfârșit, data modificării etc).

Operațiile asigurate de SGF

- **close** - în funcție de condițiile de deschidere
 - **Pentru fișiere temporare, create în programul curent și de care nu mai este nevoie în continuare:**
 - Șterge fișierul și eliberează spațiul disc ocupat.
 - **Pentru fișiere nou create și care trebuie reținute:**
 - Creează o nouă intrare în directorul discului
 - **Pentru toate fișierele care trebuie reținute:**
 - Inserează, dacă este cazul, marcajul EOF după ultimul articol al fișierului.
 - Golește (pune pe suport), dacă este cazul, ultimele informații existente în zonele tampon.
 - Pune la zi grupele de informații din descriptorul de fișier cu valorile curente
 - Dacă fișierul a fost modificat, marchează acest lucru și eventual pune numele fișierului într-o listă a sistemului. Acest lucru trebuie făcut în eventualitatea unei salvări automate de către SO a tuturor fișierelor modificate.
 - **Pentru toate fișierele, permanente sau temporare:**
 - Aduce capul de citire al discului în poziția zero
 - Eliberează spațiul de memorie ocupat de FCB.
 - Eliberează spațiile ocupate de zonele tampon în memoria operativă.
 - Eliberează (eventual) perifericul sau perifericele pe care a fost montat suportul fișierului.

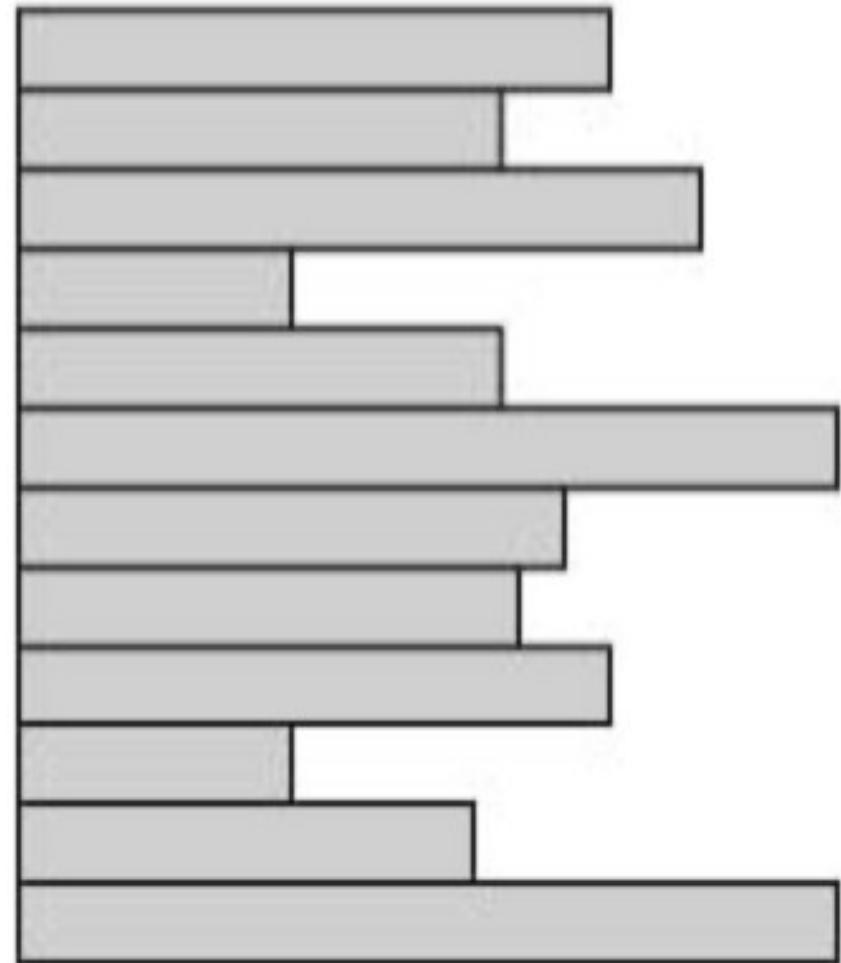
Moduri de organizare ale fișierelor

- ❑ Organizarea fișierelor este modalitatea în care sunt aranjate articolele sau unele părți din ele astfel încât să permită modurile de acces dorite.

Moduri de organizare ale fișierelor

❑ Fișiere de tip "pile"(grămadă)

- datele sunt colectate în ordinea în care apar;
- scopul este acumularea și salvarea unei cantități de date;
- înregistrările pot avea câmpuri diferite;
- nu au o structură bine definită;
- accesul la o înregistrare poate fi foarte costisitor din punct de vedere al timpului.



Moduri de organizare ale fișierelor

□ Fișiere cu organizarea secvențială

- înregistrările au un format fix (dimensiunea și ordinea articolelor este fixă, înregistrările sunt stocate în ordine pe baza unei chei – este un câmp particular în cazul unei înregistrări);
 - căutarea într-un astfel de fișier implică procesarea tuturor înregistrărilor;
 - este dificilă inserarea unor articole noi, de obicei fiind păstrate într-un fișier separat (log file sau transaction file);
 - o soluție ar fi folosirea de liste înlăntuite pentru organizarea structurii fișierului.

Moduri de organizare ale fișierelor

❑ Fișiere cu acces direct prin poziție

- un astfel de fișier are, de regulă, articole cu format fix, care sunt plasate în sectoare vecine. Există posibilitatea de a avea acces direct la orice sector de disc alocat fișierului în cauză.
- soluția se poate extinde cu mici modificări și la fișierele cu format variabil (de regulă, se reține fie lista adreselor disc la care începe fiecare articol, fie lista lungimilor acestor articole).
- majoritatea implementărilor de limbaje de programare de nivel înalt au mecanisme pentru acces direct prin poziție pentru articole cu format fix (limbajul C oferă funcția fseek).

Moduri de organizare ale fișierelor

❑ Fișiere inverse

- **Un fișier de bază**
- se creează automat un **fișier invers**,
 - conține pentru fiecare cheie specificată de utilizator, adresele disc la care se află articolele care conțin cheia respectivă.
- Atât fișierul de bază, cât și cel invers se creează și se actualizează simultan.
- Un astfel de fișier permite un mod foarte natural de organizare pentru a se realiza **acces direct prin conținut** și este folosit la unele SGBD-uri.

Moduri de organizare ale fișierelor

□ **Fișiere multilistă**

- Dacă fișierul de bază este foarte mare, atunci folosirea fișierelor inverse este ineficientă, deoarece articolele fișierului invers devin foarte lungi și greu de manipulat
- Acest lucru este înlăturat prin **fișierele multilistă**, dar accesul este secvențial.
- Pentru a putea realiza astfel de fișiere, utilizatorul definește o serie de atrbute și o serie de valori ale acestora care vor reprezenta **cheile fișierului**

Moduri de organizare ale fișierelor

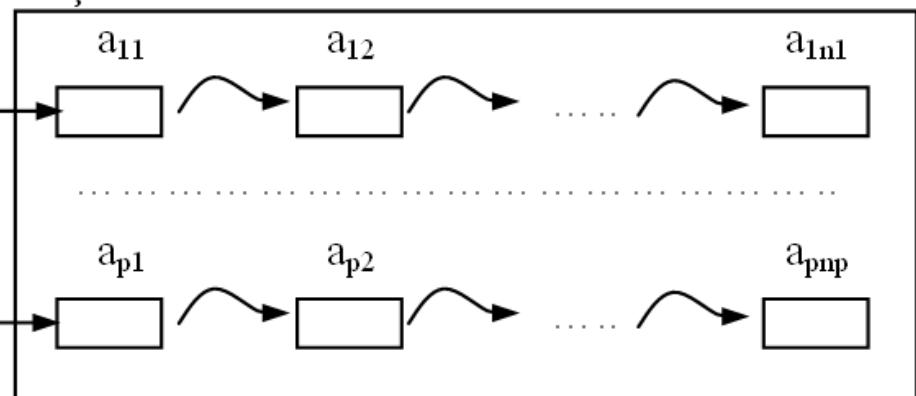
□ Fișiere multilistă

- Fiecarei chei dintr-un articol îi este atașată un pointer către articolul următor care conține aceeași cheie.
- Articolele fișierului de bază sunt legate prin atâtea liste câte chei sunt în fișier.
- Acest mod de organizare se pretează pentru **acces direct prin conținut**.

Fișier de chei

(articol1 articol2) a₁₁
(articol1 articol2) a₂₁
.....
(articol1 articol2) a_{p1}

Fișier de bază



Moduri de organizare ale fișierelor

❑ Fișiere secvențial – indexate

- Acest tip de organizare a fișierelor este cel mai des folosit pentru **accesul direct prin conținut**.
- Articolele vor fi scrise pe suport în acces secvențial și plasate în ordinea crescătoare a indexului (unic pentru fiecare articol).
- Articolele sunt grupate în blocuri de informații numite pagini
 - rezultă că toate valorile indexului dintr-o pagină sunt mai mari sau mai mici decât valorile indexului din altă pagină.
- Împreună cu fișierul se creează și o tabela de indecsi, în care se va trece pentru fiecare pagină, adresa disc a paginii și valoarea celui mai mare index din pagină.
- În funcție de dimensiunea fișierului, dacă tabela de indecsi este prea mare se poate reorganiza sub forma unui arbore.
- Acest tip de fișiere se regăsesc la sistemele de operare RSX și CP/M.

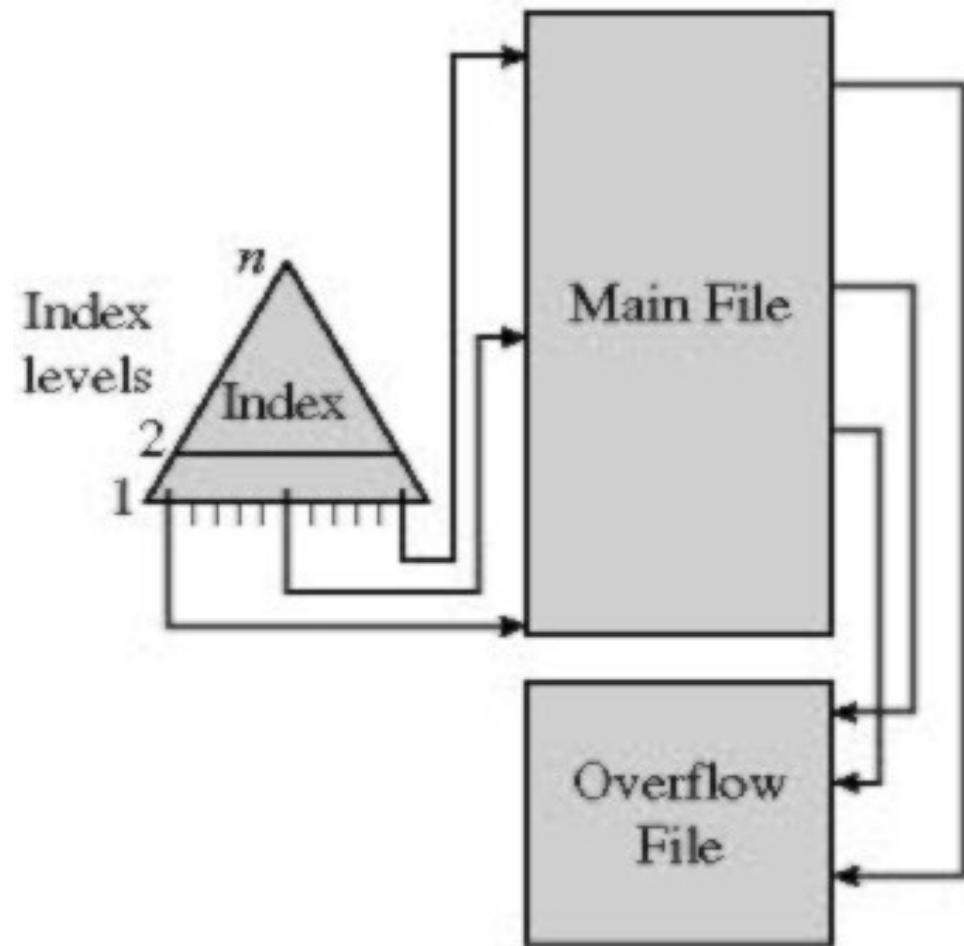
Moduri de organizare ale fișierelor

□ **Fișiere secvențial – indexate**

- Actualizarea unui astfel de fișier se face astfel:
 - ▣ articolele fișierului se leagă între ele printr-o listă simplu înlănită.
 - ▣ Articolul inserat este memorat pe disc într-o altă zonă numită parte de depășire (overflow file) și va face parte împreună cu precedentul și cu succesorul din aceeași listă simplu înlănită.
 - ▣ Dacă se fac mai multe inserări între două chei vecine din partea principală, atunci lista va lega mai multe articole în partea de depășire scăzând randamentul accesului la disc.
 - ▣ Stergerile contribuie și ele la scăderea randamentului, deoarece rămâne loc nefolosit în partea principală.
 - ▣ Din aceste motive este necesară reorganizarea frecventă a unui astfel de fișier.

Moduri de organizare ale fișierelor

- Fișiere secvențial – indexate



Moduri de organizare ale fișierelor

□ **Fișiere selective**

- a apărut aproximativ în același timp cu cele secvențial-indexate ca o replică a acestora în următorul sens:
 - dacă fișierele secvențial-indexate au funcția de regăsire materializată în tabelele de index, fișierele selective materializează funcția de regăsire printr-un simplu calcul efectuat de CPU.
 - Se elimină astfel accesele la disc de căutare în tabele, dar funcția de regăsire este dependentă de datele ce vor fi înmagazinate în fișierul respectiv.
- Este folosit tot pentru acces direct prin conținut.

Moduri de organizare ale fișierelor

□ Fișiere selective – organizare

- se presupune că datele care se vor înregistra permit existența unui atribut de valori unice pentru fiecare articol (un index);
- se poate defini o funcție:
 - ▣ $f: \{\text{valori_posibile_index}\} \rightarrow \{0, 1, \dots, n-1\}$ numită funcție de randomizare sau funcție **hash**;
 - ▣ definirea ei se face de utilizator; n - numărul de clase în care se împart articolele; toate articolele pentru care se obține aceeași valoare a funcției se spune că sunt sinonime;
- se recomandă ca numărul de articole sinonime să fie apropiat de numărul de articole care încap într-un bloc de informație (**pagina**);

Moduri de organizare ale fișierelor

❑ Fișiere selective

■ Scrierea:

- ❑ se aplică funcția de randomizare valorii indexului și se obține numărul clasei în care se află articolul.
- ❑ Partea principală a fișierului este formată din câte o pagină pentru fiecare clasă.
- ❑ Dacă pagina din partea principală nu este încă plină, articolul se pune în pagina respectivă; dacă este plină se pune în partea de depășire.
- ❑ Articolul este legat printr-o listă simplu înlănțuită de ultimul articol sinonim cu el din partea principală.

■ Citirea:

- ❑ se furnizează indexul articolului dorit căruia i se aplică funcția de randomizare, determinând clasa de sinonime din care face parte. În cadrul clasei căutarea articolului se face secvențial.

Moduri de organizare ale fișierelor

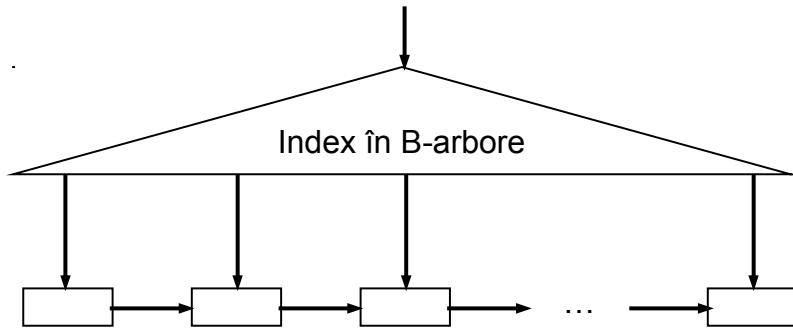
□ **Fișiere organize folosind B-arbori**

- Această tehnică a fost adoptată datorită faptului că tabelele de indecsi pot fi foarte mari, precum și datorită nevoii de a reduce numărul de accese la disc.
- Un fișier bazat pe B-arbore este net superior unui secvențial indexat din următoarele motive:
 - cheile pot fi furnizate în orice ordine, nu neapărat în ordine crescătoare;
 - dispare zona de depășire, performanțele de regăsire rămânând aceleași indiferent de numărul adăugărilor și ștergerilor;
 - căutarea se face strict în acces direct, eliminându-se căutarea secvențială în cadrul unei pagini.
- B-arborii nu permit și accesul secvențial în ordinea crescătoare a cheilor. Din acest motiv au fost introduse două noi variante ce permit și acest lucru: B+-arborele și B-arborele extins.

Moduri de organizare ale fișierelor

□ Fișiere organize folosind B+-arbori

- acest tip de arbore conține chei (și pointeri la fișierul de bază) numai în nodurile terminale (frunză) ale arborelui.
- Nodurile interioare, organizate ca un B-arbore, au doar rolul de a indica nodul terminal ce conține cheia căutată.
- Nodurile terminale se leagă între ele printr-o listă înlănțuită, simplă sau dublă, permitând astfel și accesul secvențial într-un mod foarte eficient.



Moduri de organizare ale fișierelor

- ❑ Fișiere organizate folosind B-arborele extins
 - Din rațiuni practice a apărut problema realizării accesului relativ într-un B-arbore
 - ❑ dacă la un moment dat ne aflăm pe cheia **c** (cheie curentă) în B-arbore, cum s-ar putea ajunge în mod direct la o cheie **k** (cheie nouă) aflată cu **n** chei (în ordine crescătoare) **înainte sau după** cheia **c**?

Moduri de organizare ale fișierelor

- Fișiere organizate folosind B-arborele extins
 - Soluția constă în adăugarea unei informații suplimentare pe lângă fiecare pointer din B-arbore, informație ce constă dintr-un număr întreg care reține câte chei conține subarborele indicat de pointer
 - Pentru realizarea accesului relativ, trebuie efectuate încă plus câteva operații de adunare și scădere.
 - Pentru trecerea de la cheia curentă la o cheie nouă, se parcurg nodurile B-arborelui de la strămoșul comun al celor două chei și până la cel ce conține cheia nouă.

Acțiunile SGF la nivel de suport disc

- SGF are trei sarcini principale:
 - implementarea unui sistem de regăsire a fișierelor aflate pe volumul de disc respectiv;
 - evidența spațiului neutilizat pe disc
 - punerea la dispoziția componentelor care creează sau extind fișiere, a acestui spațiu atunci când este nevoie (alocarea spațiului pentru fișiere pe disc).

Sisteme de directoare (cataloage)

- Rezolvă o problemă fundamentală a sistemului de fișiere:
 - realizarea **legăturii** între **numele** atribuit de utilizator unui fișier și **localizarea fizică** pe disc a informației asociate fișierului respectiv, operație numită **mapare**.
- Structura unui director poate fi asimilată cu aceea a unei tabele în care fiecare intrare este asociată unui fișier și conține informațiile necesare accesului:
 - numele fișierului și adresa unei structuri de date sistem care conține attributele fișierului
 - informații despre localizarea acestuia pe disc

Sisteme de directoare (cataloage)

- Operațiile asigurate de sistemul de operare pentru lucrul cu directoare sunt:
 - **create**: crearea unui director;
 - **delete**: ștergerea unui director;
 - **open**: “deschiderea” unui director, de exemplu în scopul citirii și afișării intrărilor;
 - **close**: eliberează structurile de date sistem care au fost alocate unui director ca urmare a unei operații open;
 - **readdir**: returnează următoarea intrare într-un director. Conținutul directoarelor poate fi citit și utilizând operația read (definită pentru fișiere) dar aceasta trebuie să aibă ca rezultat returnarea întregii tabele asociate directorului respectiv, obligând utilizatorul să cunoască structura acesteia în scopul regăsirii informațiilor căutate;
 - **rename**: dă posibilitatea redenumirii unui director;
 - **link**: dă posibilitatea stabilirii unei legături simbolice care are ca efect apariția numelui unui același fișier în mai multe locuri în cadrul sistemului de fișiere;
 - **unlink**: are ca efect ștergerea unei intrări într-un director.

Sisteme de directoare (cataloage)

□ Tipuri de structuri logice

■ directoare cu un singur nivel:

- este cea mai simplă structură.
- toate fișierele de pe disc sunt în același director, fapt ce impune anumite restricții asupra numelui și numărului fișierelor.
 - numărul de intrări în tabelă este limitat de dimensiunea tabelei director și nu pot exista două fișiere cu același nume chiar dacă aparțin la utilizatori diferiți.
- acest sistem era folosit la sistemele de operare CP/M și SIRIS.

Sisteme de directoare (cataloage)

□ Tipuri de structuri logice (2)

■ directoare cu două niveluri:

- O astfel de organizare elimină neajunsul unicitatii numelui de fisier.
- O astfel de structura contine la nivel superior directorul Master (MFD – Master File Directory).
- Aceasta contine cate o intrare pentru fiecare utilizator al sistemului, intrare care pointeaza catre un director Utilizator (UFD – User File Directory).
- Din fiecare UFD se pointeaza catre fisierele utilizatorului respectiv. Acest sistem a fost folosit la sistemele de operare SIRIS si RSX.

Sisteme de directoare (cataloage)

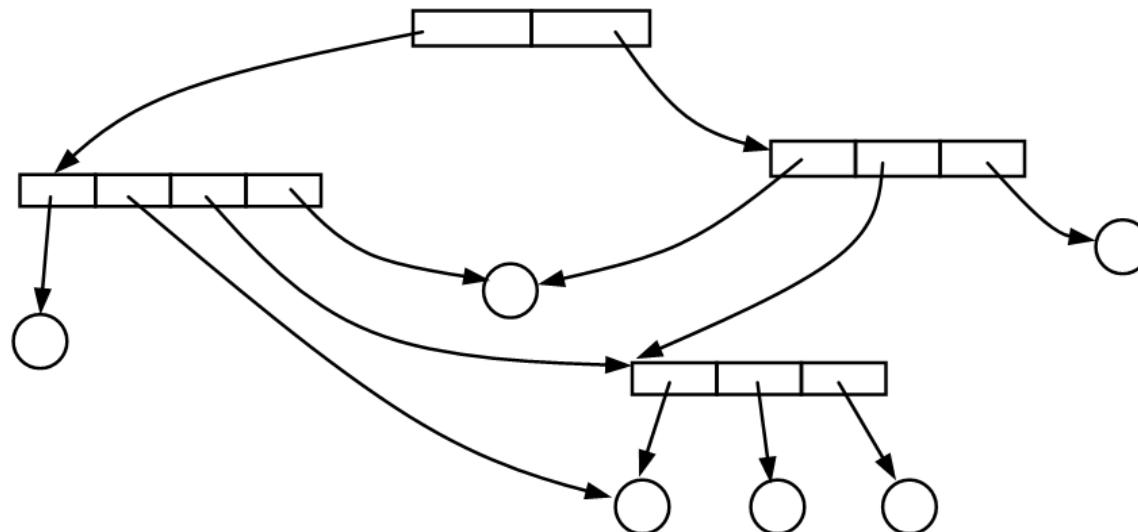
- Tipuri de structuri logice (3)
 - directoare cu structură de arbore
 - reprezintă extinderea cea mai firească a structurii cu două niveluri.
 - A început să fie folosită la SO UNIX și la MS-DOS de la versiunea 2, singura deosebire între cele două fiind caracterul de separare între numele de director care este "/" pentru UNIX și "\\" pentru MS-DOS.

Sisteme de directoare (cataloage)

□ Tipuri de structuri logice (4)

■ directoare cu structură de graf aciclic

- acest tip de structură este util deoarece directoarele cu structură de arbore nu permit ca anumite fișiere sau directoare să fie accesibile din mai multe directoare părinte
- util când se dorește partajarea unei porțiuni a structurii de fișiere între mai mulți utilizatori.



Alocarea spațiului pentru fișiere pe disc

- **alocare statică (preallocare)**: în care trebuie specificată dimensiunea spațiului ce va fi ocupat de fișier, lucru destul de greu de estimat, dar care va conduce la o alocare contiguă pe disc a fișierului;
- **alocare dinamică**: în care fișierul va ocupa atât spațiu cât este necesar, dar pe disc nu vom mai avea locații contigue.
 - Este necesară folosirea unei tabele de alocare a fișierelor (File Allocation Table) care va păstra informațiile legate de spațiul alocat fiecărui fișier.
- Cele mai cunoscute metode de alocare a spațiului pentru fișiere sunt:
 - alocarea contiguă, alocarea înlănțuită și alocarea indexată.

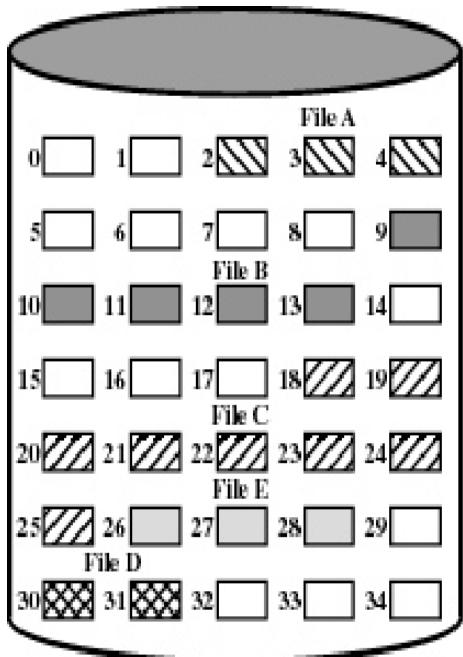
Alocarea spațiului pentru fișiere pe disc

□ Alocarea contiguă

- **un fișier pe disc trebuie să ocupe un set de adrese consecutive pe disc. În descriptorul de fișier se pun adresa de început și lungimea zonei alocate.**
- Problemele ridicate de acest mod de alocare coincid cu cele care apar la alocarea memoriei cu partii variabile: apare fenomenul de fragmentare și se efectuează compactarea.
- Acest tip de alocare a fost folosită la SIRIS (alocă un număr întreg de cilindri pentru fiecare fișier) și parțial la RSX.

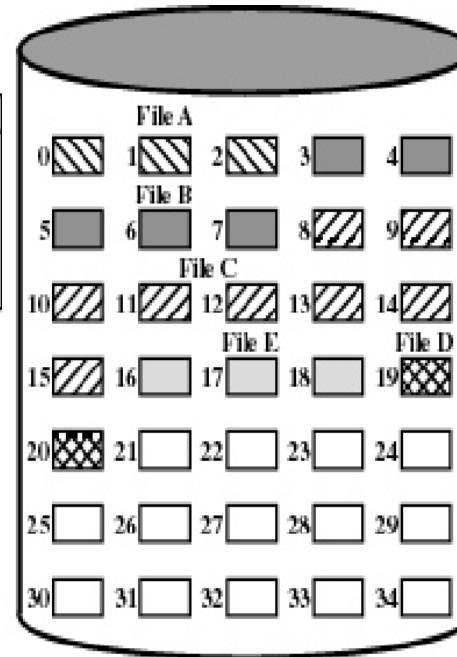
Alocarea spațiului pentru fișiere pe disc

□ Alocarea contiguă



Alocare contiguă

File Allocation Table		
File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3



Alocare contiguă (după compactare)

File Allocation Table		
File Name	Start Block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

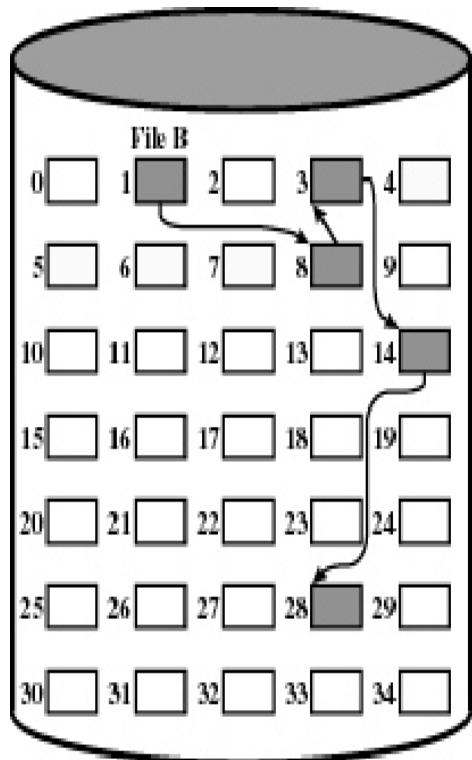
Alocarea spațiului pentru fișiere pe disc

□ Alocarea înlățuită

- **Fiecare fișier este înregistrat într-un sir de blocuri legate între ele printr-o listă înlățuită.**
- se folosește pentru a se evita fenomenul de fragmentare.
- Această metodă este foarte bună pentru accesul secvențial la fișiere.
- Acum mod de alocare este utilizat la fișierele secvențial-înlățuite
- Sistemul de fișiere de la MSDOS FAT12/16/32 folosește o variantă a acestui mod de alocare

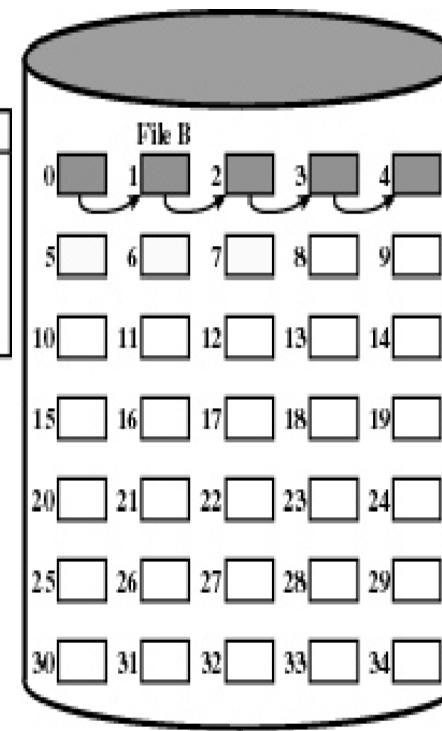
Alocarea spațiului pentru fișiere pe disc

□ Alocarea înlănită



Alocare înlănită

File Allocation Table		
File Name	Start Block	Length
...
File B	1	5
...



Alocare înlănită (după compactare)

File Allocation Table		
File Name	Start Block	Length
...
File B	0	5
...

Alocarea spațiului pentru fișiere pe disc

□ Alocarea indexată

- deoarece alocarea înlățuită nu permite accesul direct, această problemă este rezolvată prin alocarea indexată.
- **La acest tip de alocare, pe lângă blocurile atașate fișierului, la crearea fișierului respectiv, creează un bloc special, numit bloc de index.**
 - În acest bloc, se trec în ordine adresele tuturor sectoarelor ocupate de fișierul respectiv.

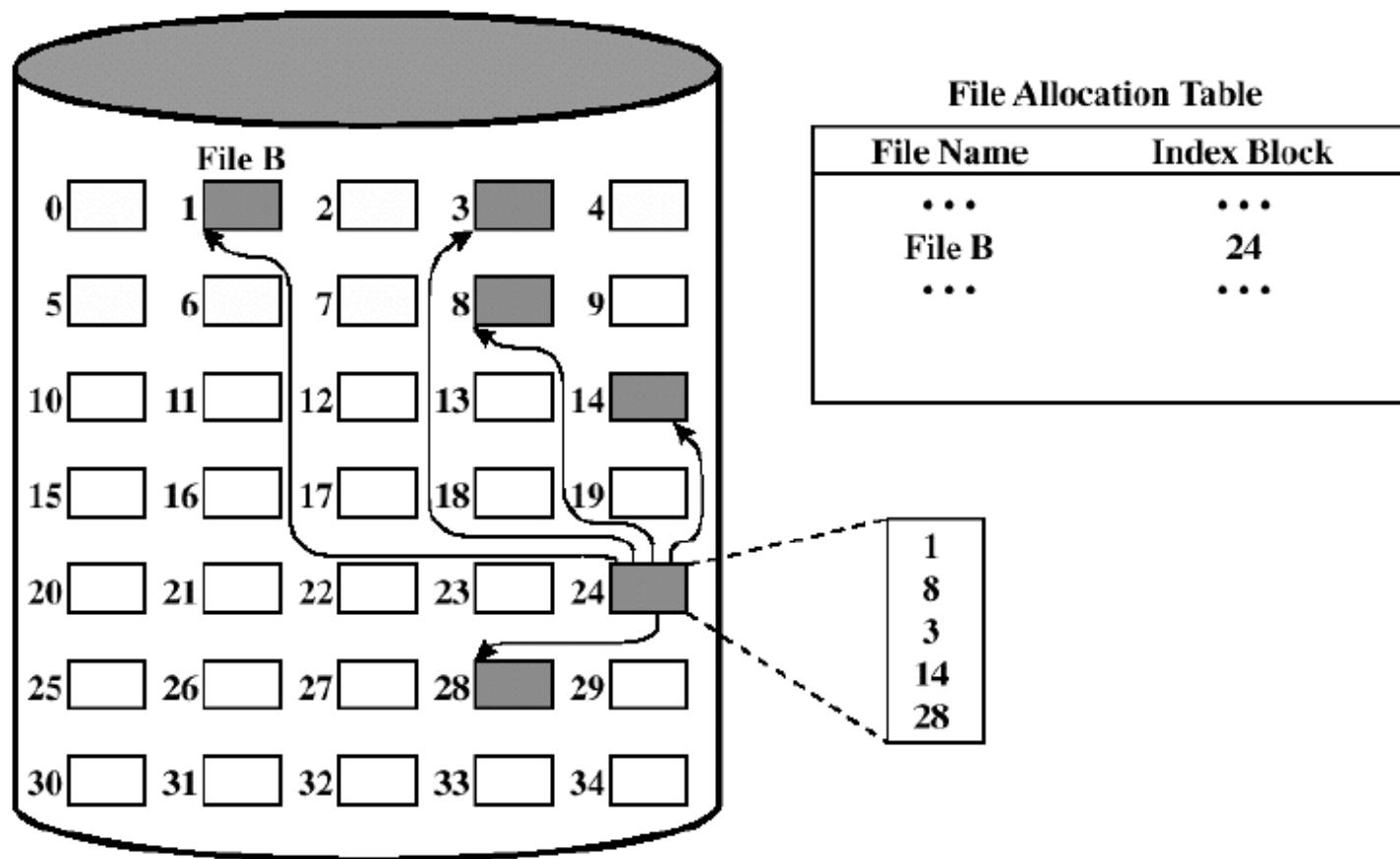
Alocarea spațiului pentru fișiere pe disc

□ Alocarea indexată

- În descriptorul fișierului (din director) există un pointer către blocul de index.
 - Dacă fișierul este mare și ocupă mai multe adrese decât încap într-un bloc, atunci se adoptă una din soluțiile următoare:
 - se creează mai multe blocuri de index, legate între ele sub forma unei liste simplu înlănuite;
 - utilizarea unor blocuri index multinivel, numărul de niveluri va depinde de cât de mare se va dori a fi dimensiunea unui fișier;
 - utilizarea unei combinații între cele două soluții.
- Această schemă de alocare este foarte flexibilă și este folosită cel mai mult în diverse sisteme de operare (UNIX adoptă o schemă de alocare indexată cu trei niveluri).

Alocarea spațiului pentru fișiere pe disc

□ Alocarea indexată



Evidența spațiului liber de pe disc

- Principala problemă care se pune la alocarea spațiului pe disc este utilizarea eficientă a spațiului și informațiile să fie introduse/obținute rapid.
- Rezolvarea acestora presupune, pe lângă o alocare eficientă a spațiului și o parte de evidență a spațiului liber de pe disc.
- Metodele de evidență a spațiului liber de pe disc sunt bazate pe **fișiere inverse** și pe **liste înlănuite**.

Evidența spațiului liber de pe disc

▫ fișiere inverse

- În directorul volumului, există o zonă rezervată pentru ocuparea volumului, numită și **tabela de ocupare a volumului** (TOV), în care este rezervat câte un bit pentru fiecare bloc de pe disc.
- Dacă blocul este liber atunci bitul este zero, în caz contrar este unu.

Exemplu: 0011100001111100001111111111011000

- La alocarea unui fișier se caută și se ocupă atâtea blocuri câte sunt necesare fișierului, după care toți biții corespunzători primesc valoarea unu.
- La ștergerea fișierului biții corespunzători se pun pe zero.
- Acest mod de evidență a fost folosit: la CP/M, unde TOV se află pe pista 2, la RSX, unde este într-un fișier BITMAP.SYS.

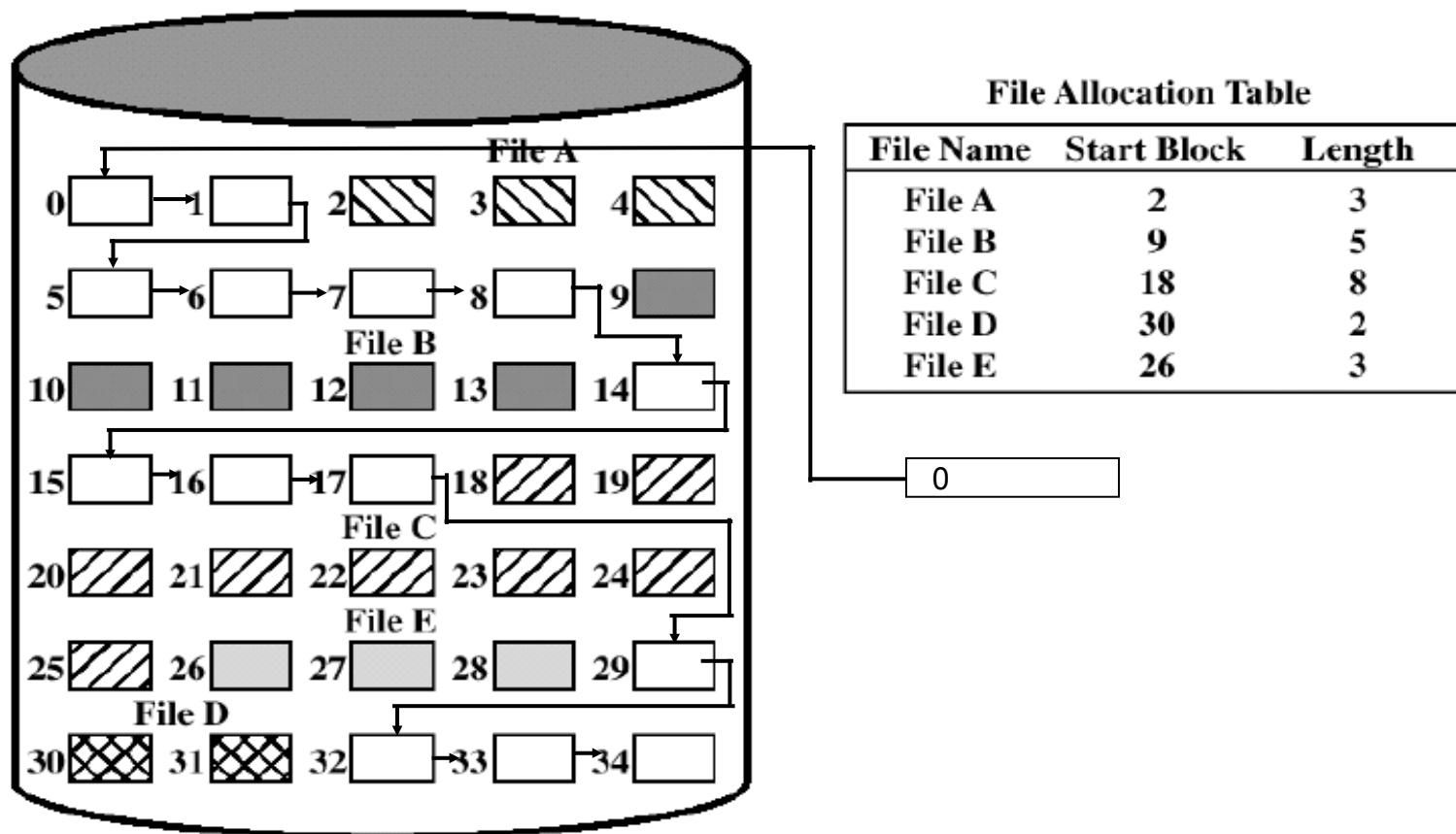
Evidența spațiului liber de pe disc

□ Listă Înlăntuită

- În directorul volumului, există un pointer către primul sector liber, care la rândul lui conține un pointer la următorul sector liber și.a.m.d. Ultimul sector are pointer de valoare zero.
- Ordinea apariției blocurilor libere în listă este dată de cererea și eliberarea de blocuri până în momentul respectiv.
- Astfel, dacă se cere ocuparea a **p** blocuri, se vor ocupa primele **p** poziții din listă, după care al **p+1** – lea bloc liber din listă va deveni primul liber. Cererea de eliberare se face nominal pentru fiecare bloc, lista actualizându-se conform ștergerii dintr-o listă simplu înlăntuită.
- Această schemă nu este foarte eficientă, deoarece pentru ocuparea unor sectoare este necesar un consum mare de timp pentru operațiile de I/O care caută spațiu liber
- Începutul listei pointează spre primul bloc

Evidență spațiului liber de pe disc

❑ listă înlățuită



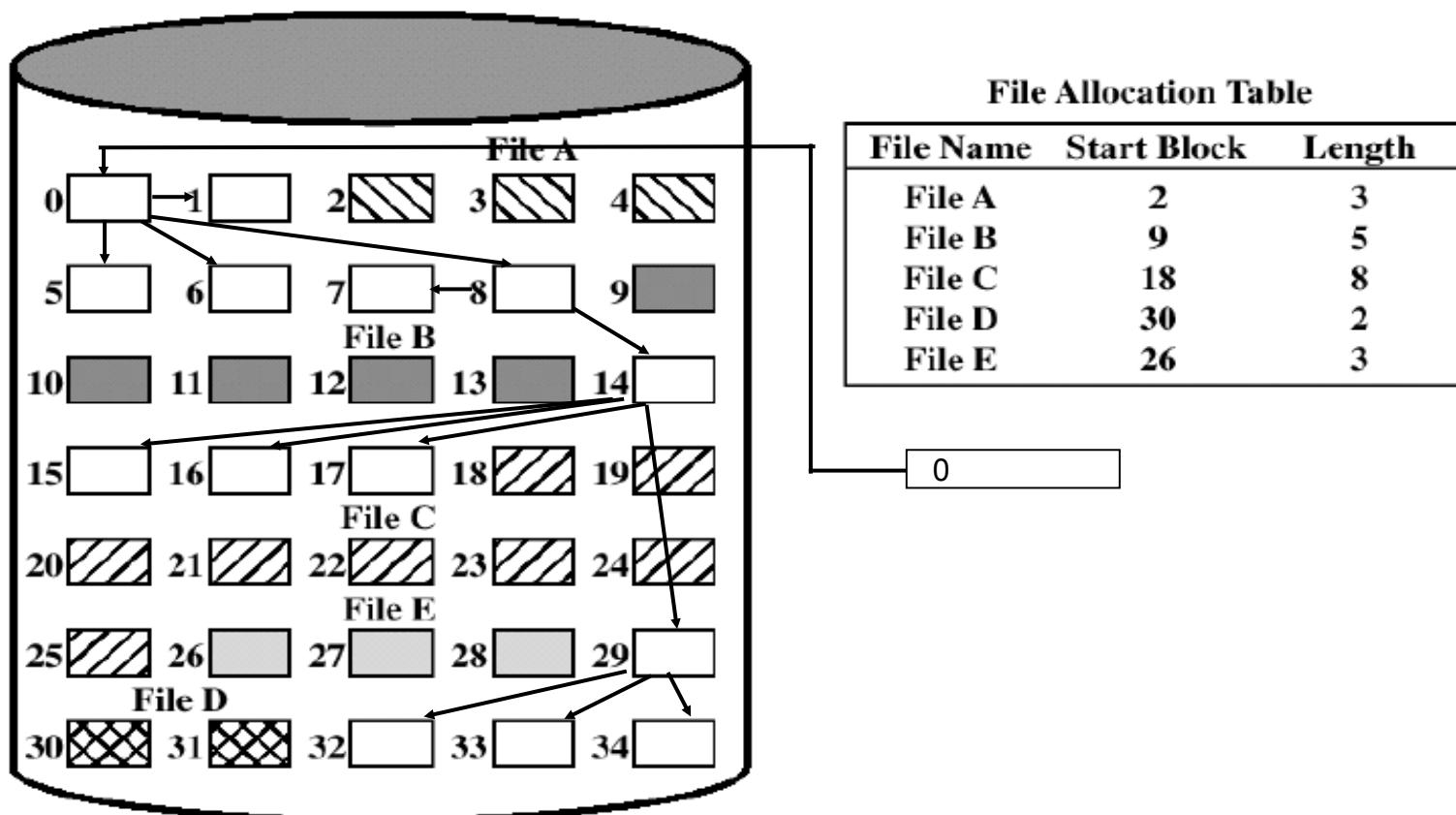
Evidența spațiului liber de pe disc

❑ Listă înlănțuită și indexată

- Această metodă este o îmbunătățire a metodei anterioare și constă în următoarele:
 - ❑ Presupunem că într-un bloc pot fi înregistrate n adrese de blocuri disc. În primul bloc liber, indicat din director, se memorează adresele a n blocuri libere.
 - ❑ Primele $n-1$ sunt adrese de sectoare efectiv libere, iar ultima adresă pointează la un bloc liber care conține, iarăși, adresele a n blocuri libere și.a.m.d.
 - ❑ Astfel numărul de operații de I/O necesare pentru căutarea de spațiu liber se micșorează în medie de $n-1$ ori.
 - ❑ Ca urmare procedurile de eliberare și alocare de blocuri vor fi complicate, dar acest lucru este compensat de reducerea masivă a numărului de accese la disc.
 - ❑ Mai mult, de aici se poate face ușor și evidența spațiului liber într-o structură arborescentă

Evidență spațiului liber de pe disc

- ❑ listă înlățuită și indexată



Sisteme de Operare



- Planificarea accesului la disc
- Algoritmi de planificare a accesului la disc
- Managementul swap-ului
- Studii de caz

Planificarea accesului la disc

- Pentru a putea avea acces la un anumit sector de disc, unitatea de disc trebuie să acționeze în trei etape:
 - poziționarea furcii pe care se află capetele de citire pe cilindrul care conține sectorul dorit;
 - așteptarea rotației discului până când sectorul trece prin fața capului de citire/scriere;
 - schimbul propriu-zis de informație.

Planificarea accesului la disc

- Operația de poziționare este cea mai mare consumatoare de timp
- există o coadă de așteptare pentru accese la disc, partajată între mai mulți utilizatori:
 - Ordinea de deservire este decisă de către SO, astfel încât serviciile aceluiași utilizator să se facă în ordinea sosirii lor
 - În același timp, SO trebuie să planifice serviciile oferite utilizatorilor, astfel încât timpul total de poziționare să fie cât mai mic.
- timpului necesar transferului de informație nu se ia în considerare, deoarece aceasta este o constantă de construcție a discului.

Planificarea accesului la disc

- Tehnicile folosite pentru îmbunătățirea timpului de acces la disc variază în funcție de producător.
- Sistemul de operare poate beneficia de acestea, folosind drivere care îi permit gestiunea simultană a mai multor discuri, care realizează o suprapunere a operațiilor de poziționare la mai multe discuri simultan.
- Deși sectoarele de pe pistele exterioare sunt mai lungi decât cele de pe pistele interioare, volumul de informație este același, lucru folosit de producătorii de discuri pentru a îmbunătăți performanțele.
- Din punctul de vedere al SO sunt importante metodele de reducere a timpului de acces la disc, de așteptare a rotației și a timpului de poziționare.

Reducerea timpului de aşteptare a rotației

- se poate face prin ordonarea adecvată a cererilor de acces la disc existente la un moment dat pentru același cilindru.
- Cel mai folosit algoritm este SLTF (Short Latency Time First) – procesul care aşteaptă cel mai puțin va fi servit primul

Reducerea timpului de poziționare

- Există o serie de elemente care nu țin de planificare și care pot influența randamentul discului:
 - structura informației pe disc influențează timpul de poziționare.
 - Astfel, pentru un fișier secvențial, alocat într-o zonă contiguă pe disc, timpul de acces este mult mai mic decât cel de la un fișier secvențial - indexat.

Algoritmi de planificare a accesului la disc

- Acești algoritmi se referă la planificarea poziționării capetelor de citire/scriere a discului.
- Pentru exemplificare, vom considera următoarea secvență de cereri de poziționare pe următorii cilindri: 98, 183, 37, 122, 14, 124, 65, 67 și vom presupune că avem capetele poziționate pe cilindrul 53 (discul are 199 cilindri).

Algoritmi de planificare a accesului la disc

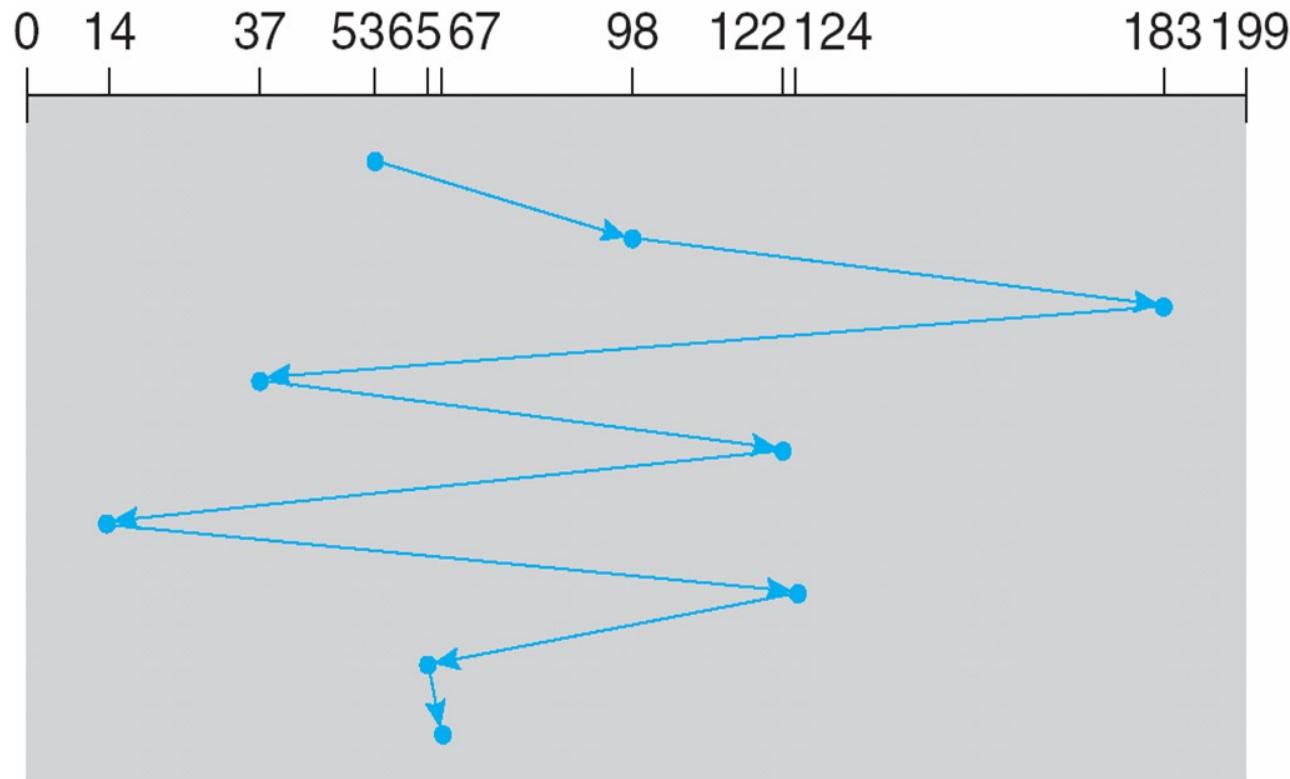
- FCFS (First Come First Served)
 - **execută cererile în ordinea în care au venit.**
 - Deși este simplu, algoritmul nu este foarte eficient, deoarece sunt necesare mișcări de poziționare însumând 640 cilindri pentru secvența de cereri de mai sus.

Algoritmi de planificare a accesului la disc

□ FCFS (First Come First Served)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



Algoritmi de planificare a accesului la disc

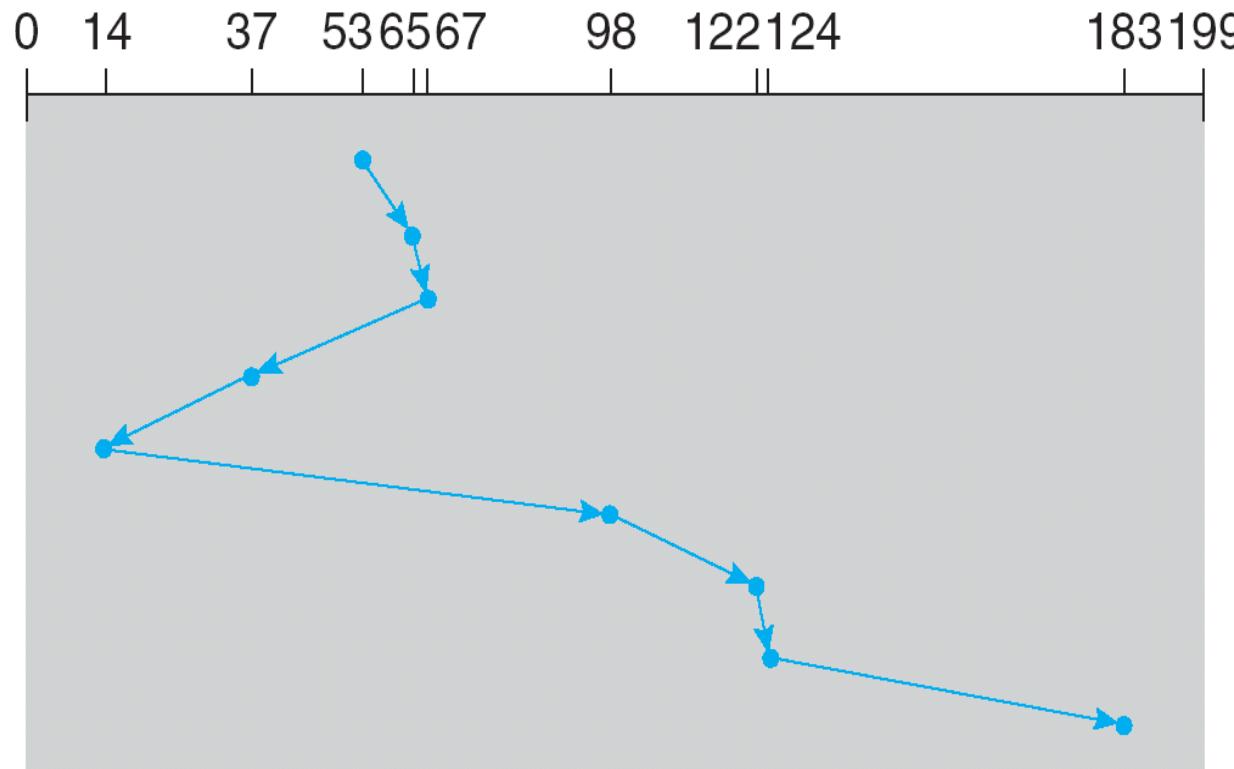
- SSTF (Shortest Seek Time First)
 - **execută de fiecare dată operația care solicită cea mai scurtă poziționare, față de locul curent.**
 - Pentru secvența de cereri luată ca exemplu, avem următoarea ordine de deservire: 65, 67, 37, 14, 98, 122, 124, 183.
 - Această ordine necesită poziționări peste numai 236 cilindri.
 - Principalul dezavantaj este acela că poate amâna infinit anumite cereri deoarece coada de așteptare se modifică permanent.

Algoritmi de planificare a accesului la disc

□ SSTF (Shortest Seek Time First)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



Algoritmi de planificare a accesului la disc

□ SCAN

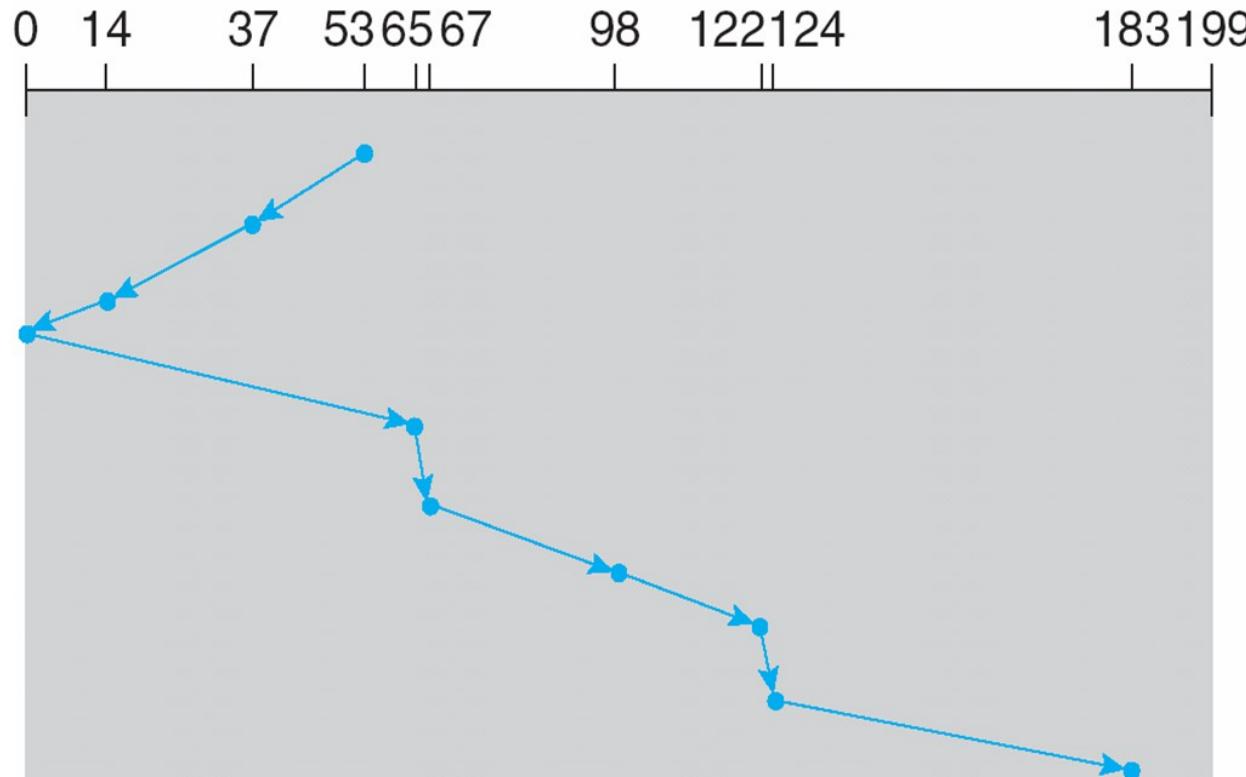
- pentru acest algoritm, **mișcarea capetelor începe de la ultimul cilindru, mergând spre primul și servind toate cererile pe care le întâlnește.**
- După ce a servit cererea de la cea mai mică adresă de cilindru solicitat, capetele își schimbă sensul de mers, servind cererile apărute ulterior până la servirea cererii de pe cilindrul solicitat cu cea mai mare adresă, apoi, iar se schimbă sensul ș.a.m.d.
- Pentru exemplul nostru, dacă servirea cilindrului 53 s-a făcut la mișcarea spre adrese mici, atunci servirea se face în ordinea: 37, 14, 65, 67, 98, 122, 124, 183. Numărul de deplasări totale va fi de 192 cilindri.
- Este posibil ca unele cereri să aștepte două parcurgeri ale discului până sunt servite.

Algoritmi de planificare a accesului la disc

□ SCAN

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



Algoritmi de planificare a accesului la disc

□ C-SCAN (circular SCAN)

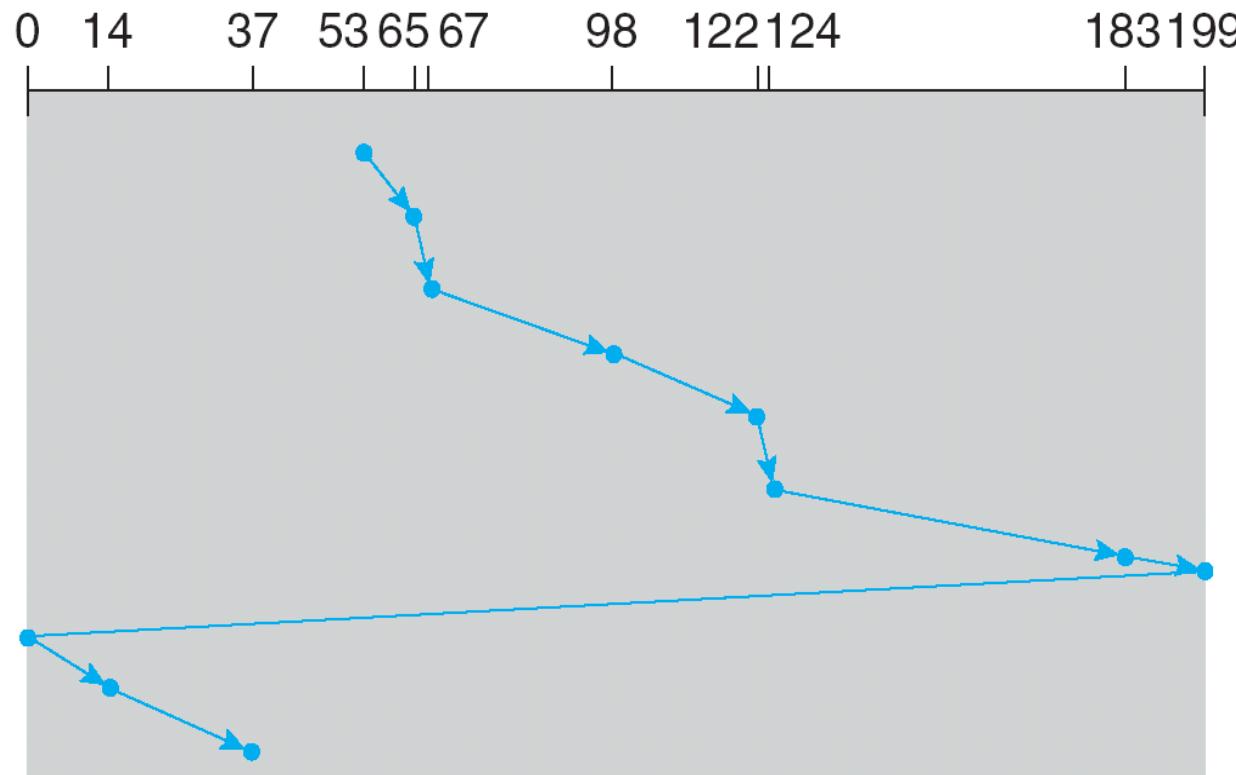
- spre deosebire de algoritmul SCAN, **servirea cererilor se face numai când capetele se deplasează de la adrese mici spre adrese mari.**
- Când se ajunge la ultimul cilindru, capetele sunt retrase automat pe cilindrul 0 (de regulă cu mare viteză), după care se reîncep servirile apărute în coadă.
- Pentru exemplul nostru, ordinea de servire a cererilor este: 65, 67, 98, 122, 124, 183; 199; retragere la 0; 14, 37.
- La unele tipuri de unități de disc retragerea capetelor pe cilindrul 0 durează mai puțin decât o poziționare obișnuită și în acest caz putem considera că fiecare cerere este servită cel mult după parcurgerea în întregime a cilindrilor discului.

Algoritmi de planificare a accesului la disc

□ C-SCAN (circular SCAN)

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



Algoritmi de planificare a accesului la disc

□ LOOK

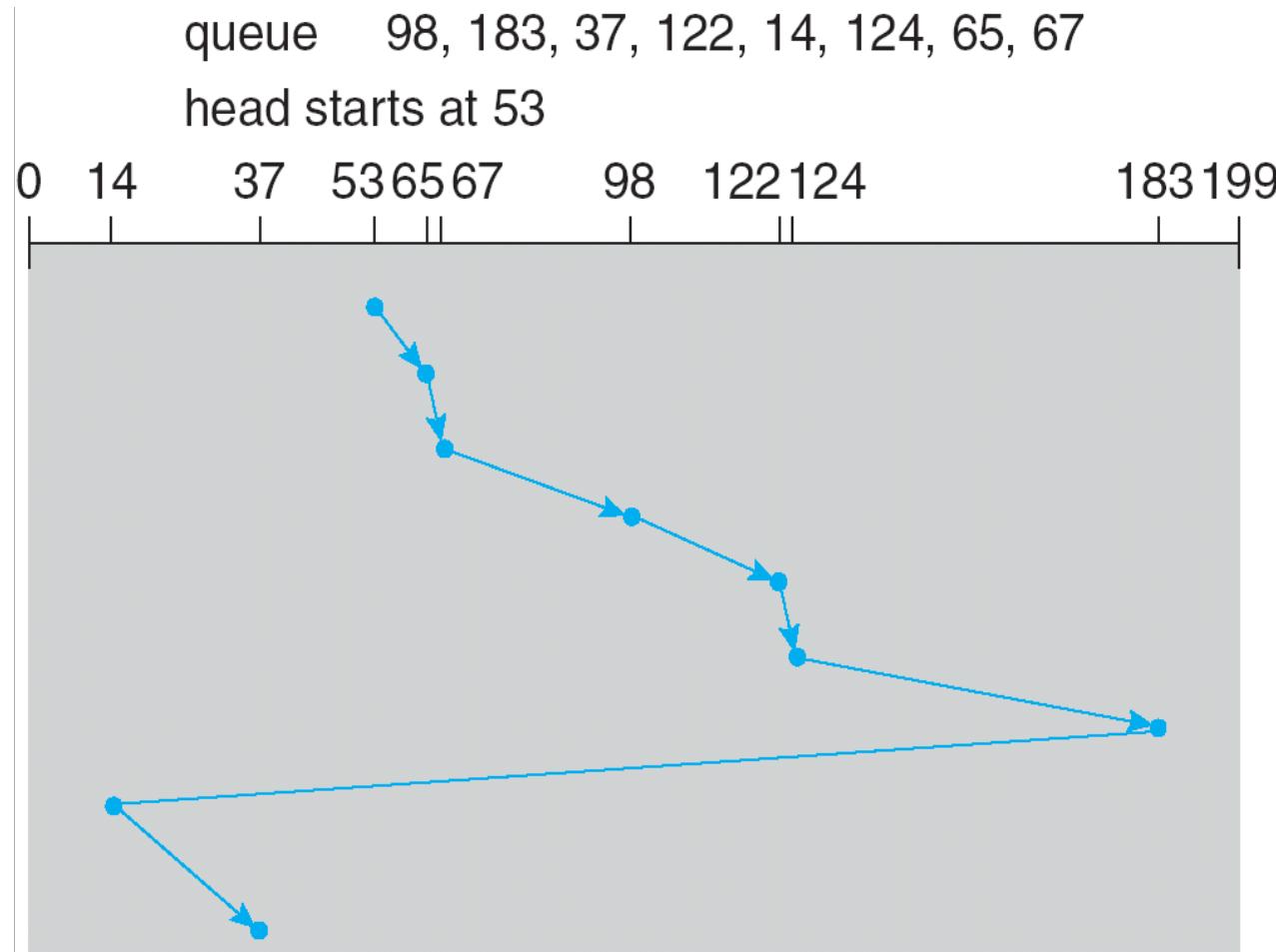
- **capul de citire/scrivere se deplasează spre cilindrul cel mai apropiat de acela la care se găsește**
- Exemplu: 37, 14, 65, 67, 98, 122, 124, 183

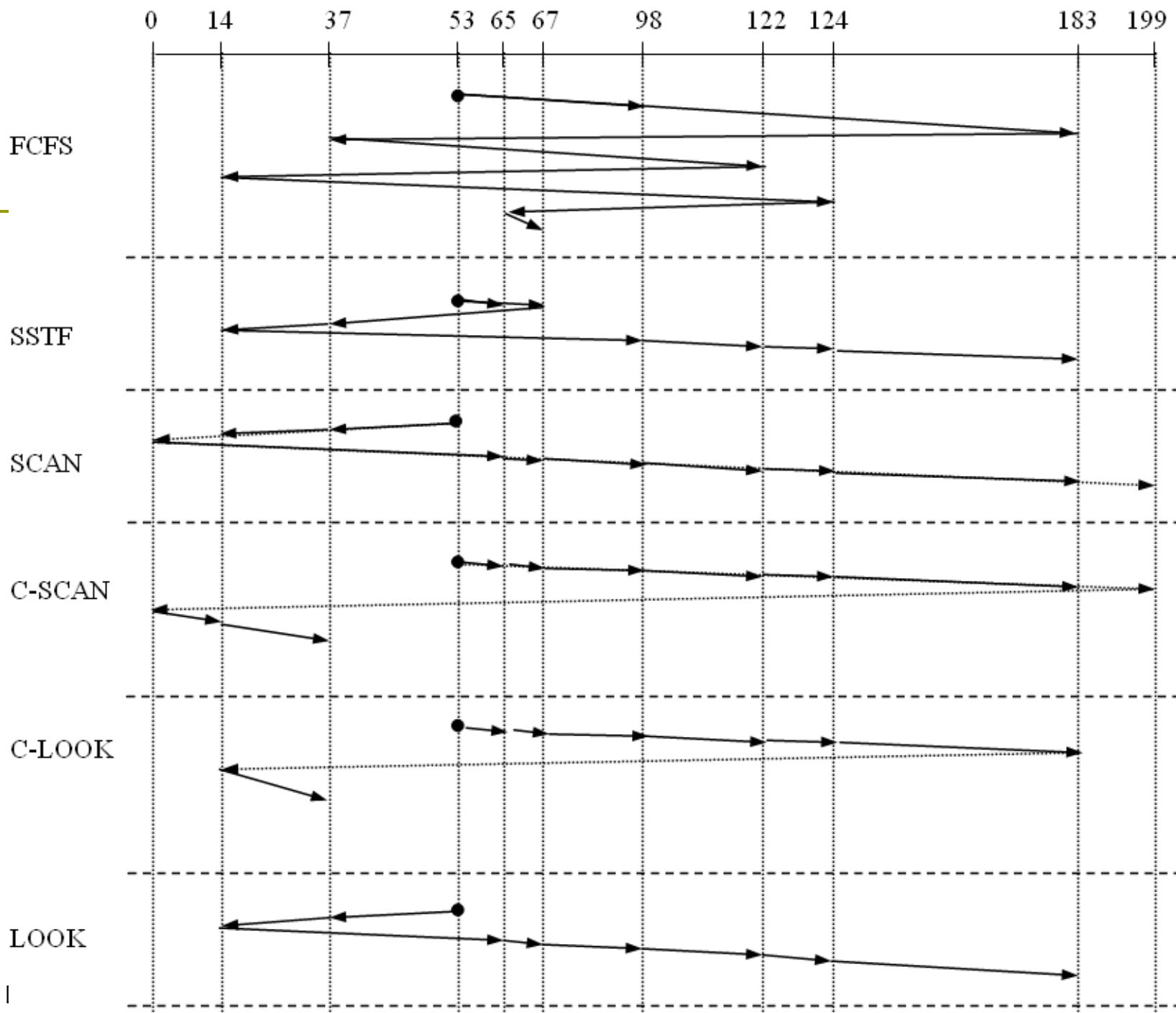
□ C-LOOK

- **capetele se deplasează cât de mult posibil în direcția în care există cereri după care, dacă nu mai sunt cereri în direcția curentă este schimbată direcția.**
- La schimbarea direcției nu este servită nici o cerere.
- Exemplu: 65, 67, 98, 122, 124, 183, 14, 37

Algoritmi de planificare a accesului la disc

□ C-LOOK





Algoritmi de planificare a accesului la disc

- Cererile de acces la disc sunt influențate și de metoda de alocare a fișierelor.
- localizarea directoarelor și a blocurilor de index este foarte importantă, deoarece deschiderea unui fișier înseamnă parcurserea structurii de directoare și din acest motiv, plasarea directoarelor la jumătatea distanței dintre marginile discului poate reduce semnificativ numărul de deplasări ale capetelor.
- **producătorii de discuri implementează algoritmii de servire a cererilor la disc în controller-ul de disc.**
- **Sistemul de operare va trimite cererile în ordinea FCFS, iar controller-ul le va prelua și le va servi într-o ordine optimă.**

Managementul swap-ului

- implementarea spațiului de swap trebuie realizată astfel încât să realizeze cele mai bune performanțe posibile.
- spațiul de swap este utilizat în funcție de felul în care sistemul de operare implementează algoritmii de acces la memorie.
 - De exemplu, poate fi folosit pentru păstrarea întregii imagini a procesului inclusiv zonele de cod și date.
- Sistemele de operare pot oferi posibilitatea utilizării mai multor spații de swap.

Managementul swap-ului

- Spațiul de swap poate fi localizat în sistemul normal de fișiere ca un simplu fișier de dimensiune mare (cazul Windows) și, în acest caz, rutine de acces la fișier pot fi folosite pentru utilizarea swap-ului.
- adăugarea unui nou spațiu de swap se poate face prin crearea unui nou fișier.
- această implementare nu este foarte eficientă datorită faptului că navigarea prin structura de directoare a sistemului consumă mult timp și necesită accese suplimentare la disc.
- fragmentarea externă poate duce la creșterea timpului de citire și scriere din swap.

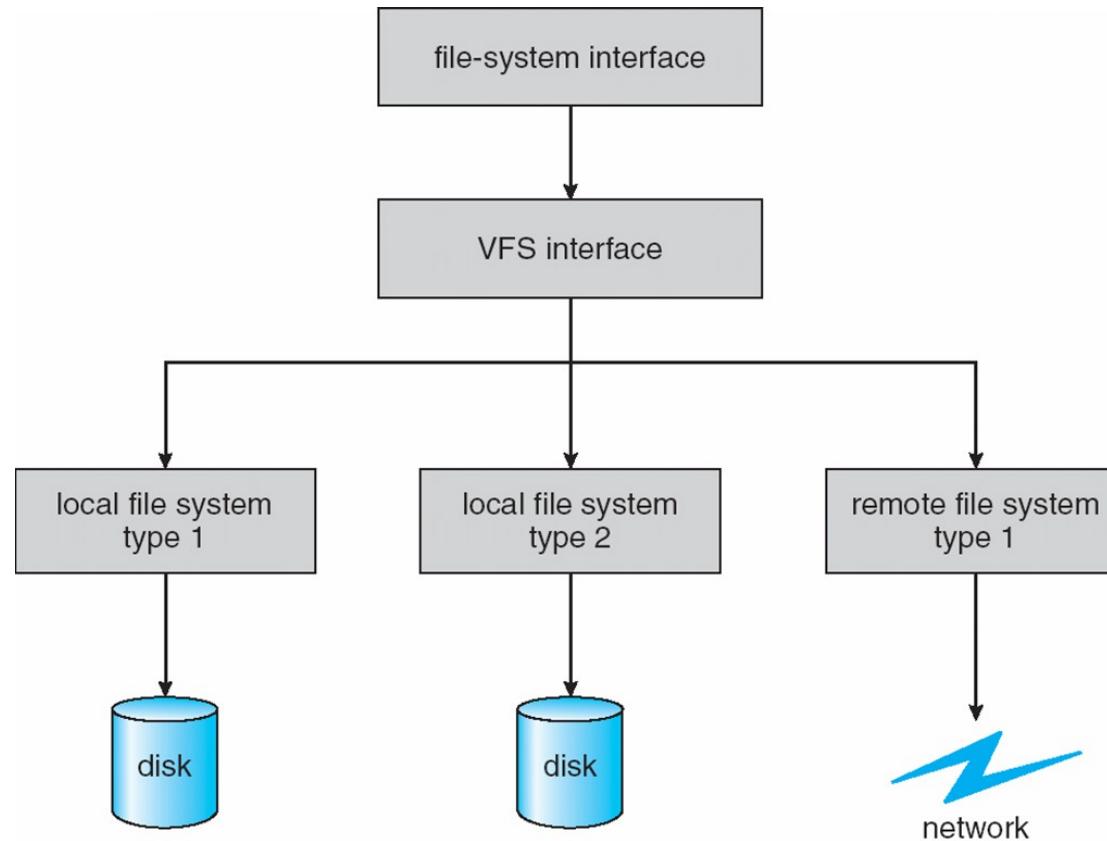
Managementul swap-ului

- O altă metodă este crearea unei partiții speciale pentru swap.
- Avantajul aceste metode este viteza.
- Algoritmii sunt optimizați pentru o extragere și o stocare foarte rapidă a datelor.
- Fragmentarea internă este destul de mare, dar acest lucru este acceptabil, deoarece timpul de viață al datelor în swap este destul de redus decât într-un sistem de fișiere.
- Este permisă și existența unor fișiere suplimentare de swap.

Virtual File Systems

- ❑ Reprezita o modalitate de implementarea a sistemelor de fisiere in diverse sisteme de operare, asemanatoare principiilor POO
- ❑ Permit utilizarea acelorasiapeluri sistem pentru pentru a accesa diverse sisteme de fisiere
- ❑ apele sistem se fac catre VFS si nu catre un anume sistem de fisiere

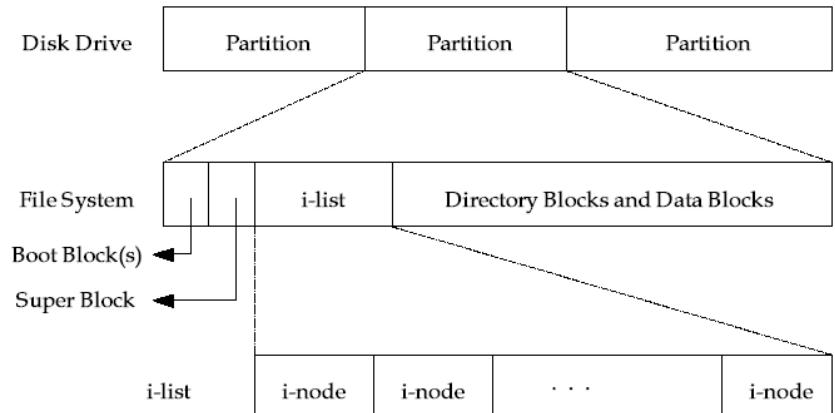
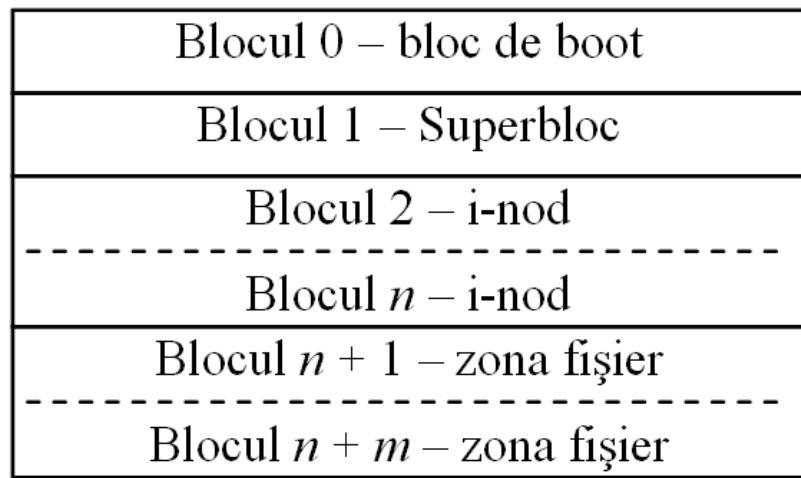
Virtual File Systems



Studii de caz - Unix

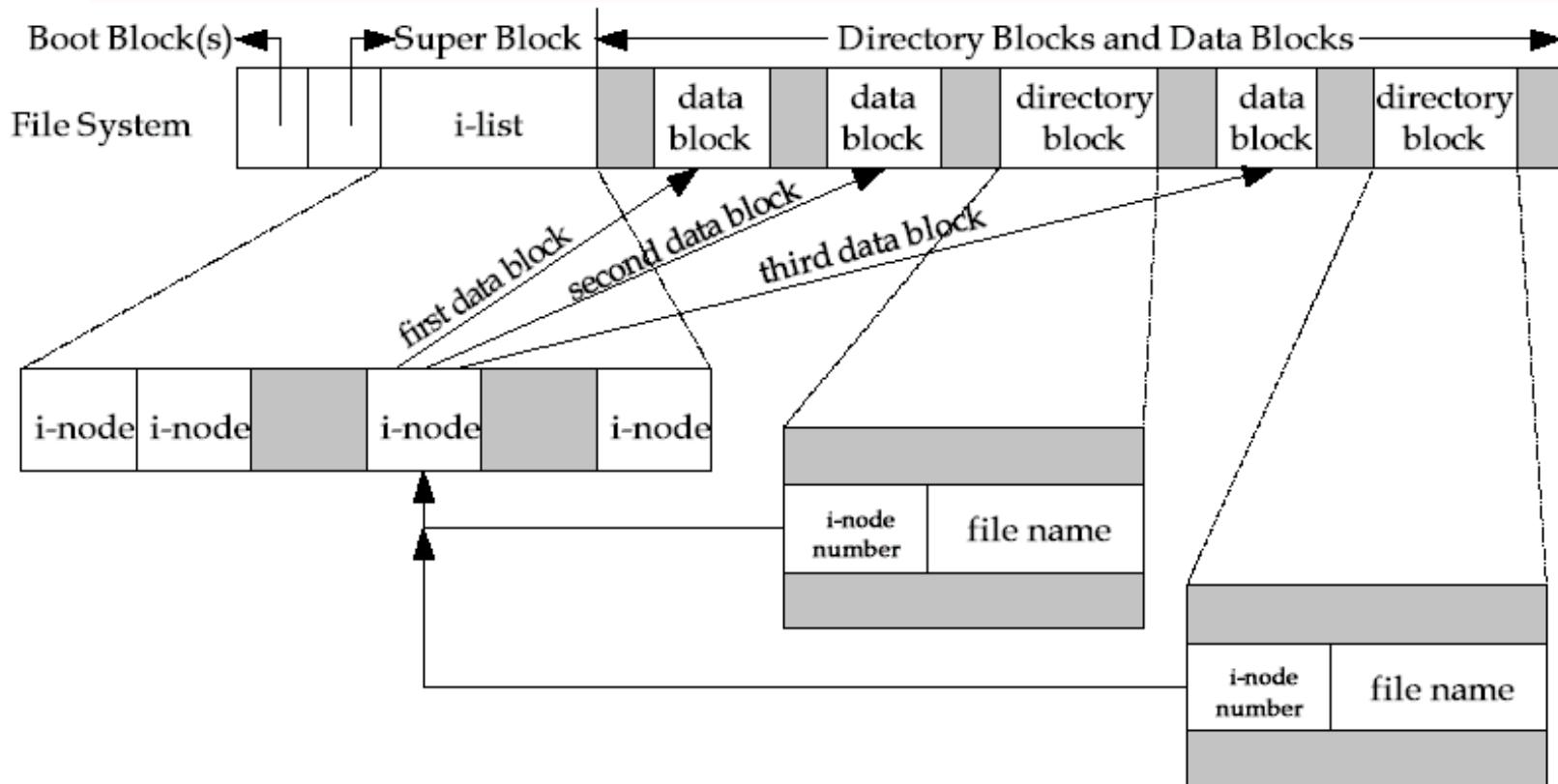
- Sistemul de fișiere Unix este o structură de date rezidentă pe disc. și este formată din 4 categorii de blocuri:
 - Blocul 0 – conține programul de încărcare al SO;
 - Blocul 1 sau superbocul conține o serie de informații prin care se definește sistemul de fișiere de pe disc: numărul de i-noduri, numărul de zone definite pe disc, pointeri spre harta de biți a alocării i-nodurilor, pointeri spre harta de biți a spațiului liber disc, dimensiunile zonelor disc, etc. ;
 - Blocurile de la 2 la n (n - este o constantă a formatării discului) conțin lista de i-noduri (index-node), listă numită și i-listă.
 - i-nodul este o structură de date ce reprezintă, de fapt, descriptorul de fișier. Numărul de ordine al unui i-nod în cadrul i-listei se numește i-număr.

Studii de caz - Unix



Studii de caz - Unix

□ Structura detaliată a unui disc



Studii de caz - Unix

❑ Structura detaliată a unui disc

- Partea cea mai mare a discului este rezervată zonei fișierelor.
- În superbloc este realizată evidența spațiului liber și acest lucru se face prin metoda fișierului invers.
- Informațiile necesare alocării sunt fixate în i-noduri.

Studii de caz - Unix

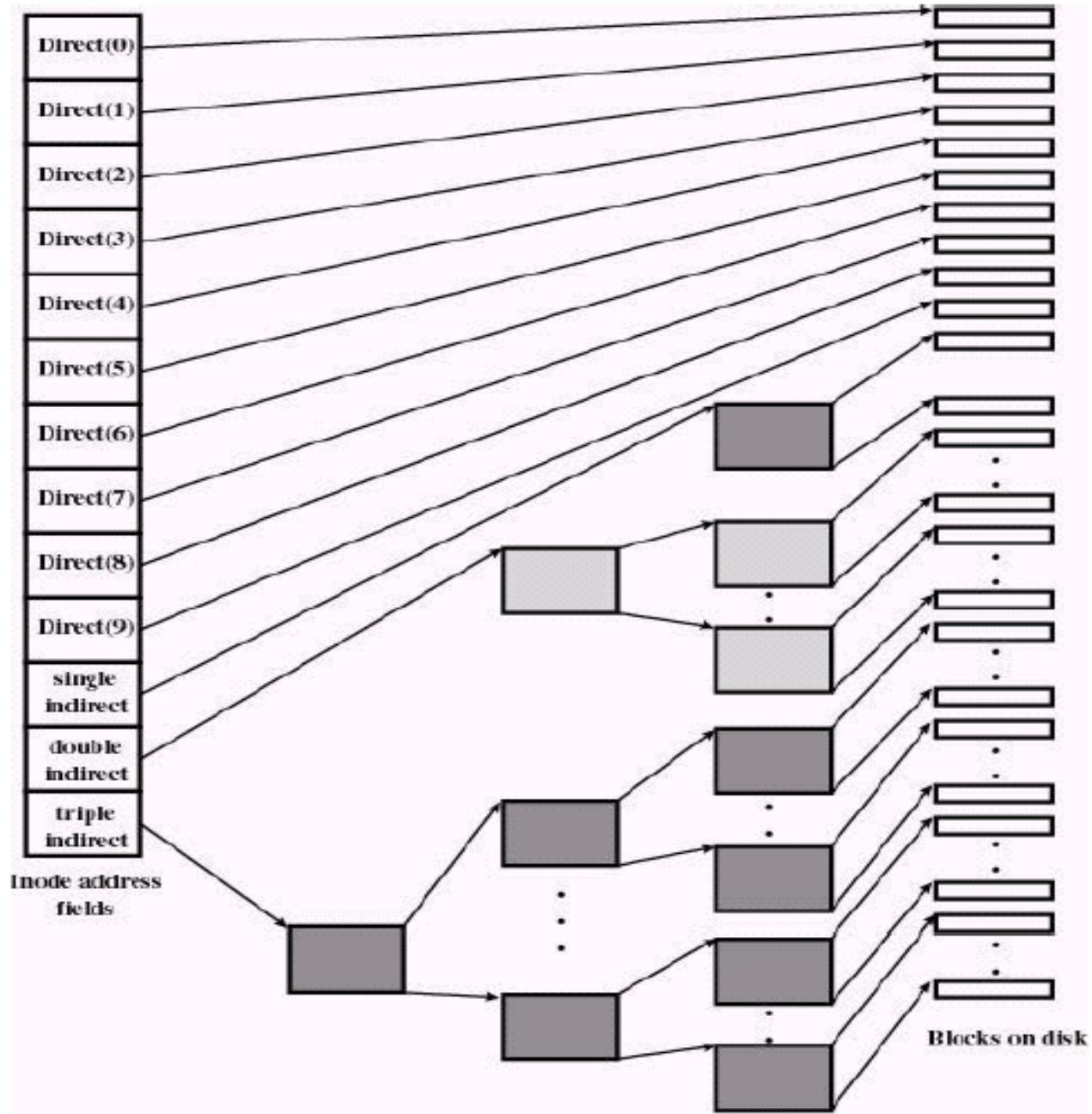
▫ Structura detaliată a unui disc

- Structura unui director este de forma:

Numele fișierului	i-număr
-------------------	---------

- Informațiile care apar într-un i-nod sunt:

- identifierul proprietarului fișierului,
- identifierul de grup,
- biții de protecție,
- lungimea fișierului,
- data creării și a ultimei actualizări,
- numărul de legături la fișiere,
- indicarea faptului că fișierul este un director,
- pointerii spre blocurile alocate fișierului (13 intrări)



Studii de caz – Unix

❑ Structura unui i-nod

- Primele 10 intrări
 - adresele primelor 10 blocuri de câte 512 octeți, care aparțin fișierului
- Intrarea 11:
 - un bloc de indirectare simplă, ce conține adresele următoarelor 128 blocuri de câte 512 octeți care aparțin fișierului
- Intrarea 12:
 - adresa unui bloc de indirectare dublă, care conține adresele a 128 blocurilor de indirectare simplă,
 - fiecare bloc conține adresele a 128 blocuri de indirectare de câte 512 octeți, fiecare cu informații ce aparțin fișierului
- Intrarea 13
 - conține adresa unui bloc de indirectare triplă, care conține adresele a 128 blocuri de indirectare dublă.
 - Numărul de accese necesare pentru a obține direct un octet oarecare este cel mult 4, iar pentru fișiere mici acest număr este mai mic.

Studii de caz - Unix

- Alocarea fișierelor: bazată pe blocuri, este o alocare dinamică.
 - Este utilizat un index pentru a ști în orice moment localizarea pe disc a părților din fișier.
 - Lungimea unui bloc pentru UNIX System V este de 1Kbyte
 - fiecare bloc poate stoca 256 adrese de blocuri
 - dimensiune maximă posibilă a unui fișier de 16Gbytes.
- Avantaje:
 - i-nodul are o dimensiune fixă, relativ mică și, din acest motiv, poate fi păstrat în memorie un timp mai îndelungat;
 - fișierele mici pot fi accesate cu indirectare simplă sau fără indirectare, lucru ce duce la reducerea timpului de acces la fișier;
 - Dimensiunea maximă a unui fișier este suficient de mare pentru a satisface orice aplicație.

Studii de caz – JFS (Journal File System)

- Dezvoltarea sistemelor de calcul și a utilizării lor fără întrerupere a dus la necesitatea apariției unei metode de protecție a sistemului de fișiere în caz de accidente.
- au apărut tehnici de păstrare a informațiilor legate de toate operațiunile realizate cu discul (**jurnale** sau **log-uri**)
- Astfel, **scrierile pe disc sunt realizate asincron** datorită faptului că, după terminarea unei cereri de scriere la disc, **informațiile sunt înregistrate în log-uri și apoi făcute modificările în tabelele descriptorilor de fișier.**

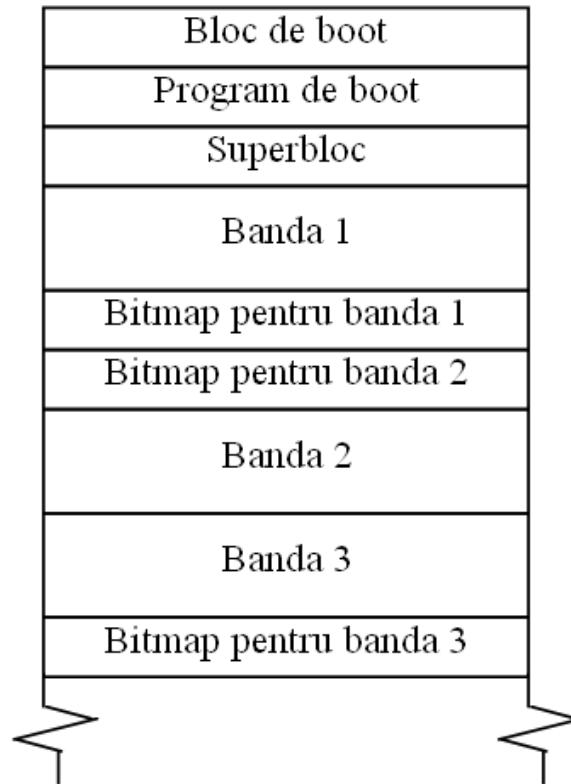
Studii de caz – JFS (Journal File System)

- Informațiile din log-uri sunt actualizate astfel:
 - este folosit un cursor (pointer)
 - Atunci când cursorul înaintează odată cu scrierea pe disc a blocurilor asociate tranzacției și informațiile legate de tranzacția respectivă sunt șterse din log-uri
 - Pentru refacerea informației se reiau toți pașii de la ultima poziție a cursorului
- Avantaje:
 - refacerea foarte rapidă a datelor – depinde de dimensiunea log-urilor și nu de dimensiunea sistemului de fișiere.
- Dezavantaje:
 - scrieri suplimentare pe disc;
 - pierderea de spațiu liber pentru păstrarea log-urilor (nesemnificativ în raport cu posibilitățile de refacere a informației)

Studii de caz – HPFS

High Performance File System

- ❑ Structura de date folosită pentru descrierea localizării pe disc a fișierelor este B-arborele
- ❑ organizează discul în volume și rezervă primele 18 sectoare pentru blocul de boot, superbloc și blocul “de rezervă”.
- ❑ Aceste blocuri conțin informațiile de control care sunt utilizate pentru inițializarea sistemului, gestiunea sistemului de fișiere și refacerea sistemului de fișiere după producerea de erori.



Studii de caz – HPFS

High Performance File System

- Restul sistemului de fișiere este organizat în benzi de 8MB, fiecare având asociată o hartă de biți de dimensiune 2KB în care fiecare bit corespunde unui bloc de 4 KB din cadrul benzii și indică prin 0 sau 1 faptul că blocul respectiv este liber sau alocat.
- Hărțile de biți sunt situate alternativ la sfârșitul și la începutul benzilor aşa încât să poată fi făcute alocări contigue de până la 16 MB (două benzi consecutive).

Studii de caz – HPFS

High Performance File System

- Pentru descrierea fișierelor, HPFS utilizează F-noduri.
- Acestea sunt structuri de date care conțin attributele și informațiile legate de localizarea pe disc a fișierelor.
- Fiecare F-nod conține referiri la cel mult 8 extensii.
- Extensiile sunt entități distincte stocate în fiecare volum, fiecare din ele putând adresa blocuri ce însumează până la 16 MB.
- pot fi suportate dimensiuni ale fișierelor de până la 128MB.
- Pentru fișierele mai mari, F-nodurile conțin un număr de 12 adrese către noduri de alocare, ce pot fi utilizate pentru a adresa mai multe extensii.

Studii de caz – Windows

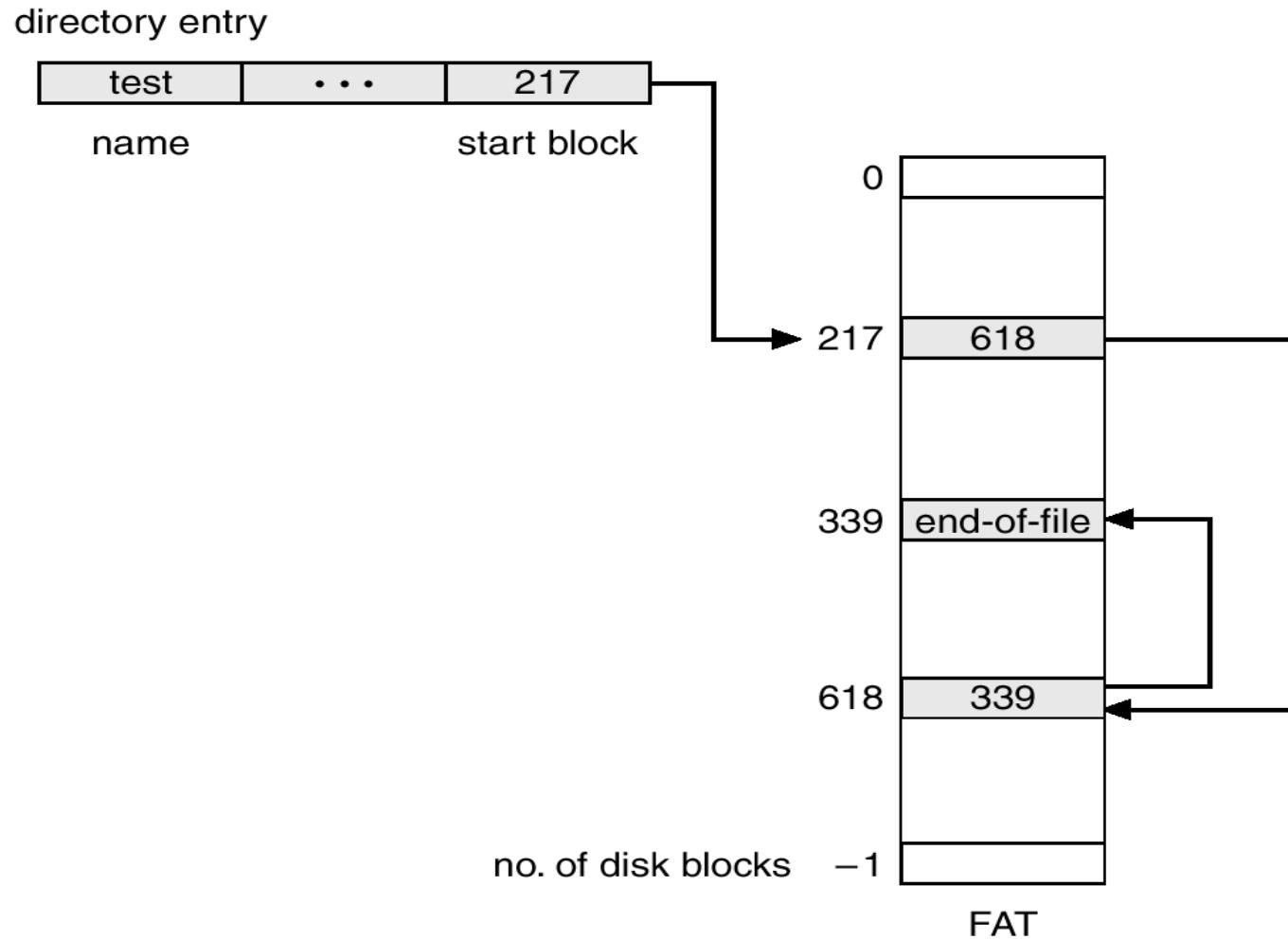
- Windows NT oferă posibilitatea optării pentru 2 sisteme de fișiere diferite:
 - NTFS
 - FAT16.
- Windows 2000/XP și ulterior
 - NTFS – imbunatatit fata de versiunea de la Windows NT
 - FAT32

Studii de caz – Windows

❑ File Allocation Table (FAT)

- Sistemul de fișiere FAT folosește ca mecanism de alocare lista înlănuțuită de indecși.
- Sistemul de operare întreține o tabelă de alocare ce conține câte o intrare pentru fiecare bloc memorat pe disc.
- Tabela FAT are rolul de a reda modul de înlănuire a blocurilor ce alcătuiesc un anumit fișier, memorând în fiecare intrare din FAT indexul blocului următor.

Studii de caz – Windows File Allocation Table (FAT)



Studii de caz – Windows

❑ File Allocation Table (FAT)

- O alternativă a acestei metode de alocare memorează pointerii către blocul următor chiar în blocurile de date, renunțând la utilizarea unei liste separate.
- Avantajul acestei reprezentări mai compacte conduce la alte două dezavantaje majore:
 - ❑ Primul este legat de accesul aleator la datele conținute în fișier:
 - Înlătuirea ar trebui urmărită citind practic bloc-cu-bloc, adică efectuând un număr mare de operații de I/O, în timp ce păstrarea tabelei FAT în memorie conduce la obținerea unei viteze acceptabile.
 - ❑ Al doilea dezavantaj se referă la dimensiunea datelor dintr-un bloc, care nu mai este multiplu de doi ci diferența dintre dimensiunea unui bloc – multiplu de doi - și un număr de octeți alocați pointerului către blocul următor.
 - Deși pare o problemă minoră, totuși în multe situații acest fapt poate conduce la penalizări asupra eficienței operațiilor de I/O (care vizează în general transferul unor structuri de date de dimensiune putere a lui doi).

Studii de caz – Windows

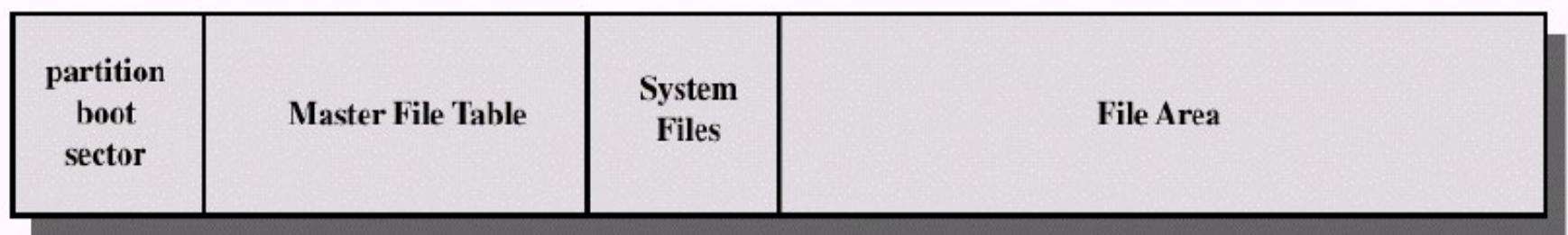
❑ New Tehnology File System (NTFS)

- Sistemul de fișiere NTFS folosește pentru descrierea localizării pe disc a fișierelor o structură de tip B-arbore.
- Fiecare volum NTFS conține o **Master File Table (MFT)** care este, de fapt, un fișier ce cuprinde informații despre fișierele și directoarele volumului respectiv.
- **MFT** este organizată ca o succesiune de înregistrări dintre care primele 16 sunt utilizate pentru descrierea **MFT** însăși și pentru furnizarea de informații necesare refacerii după situații de avarie.
- Următoarele înregistrări din **MFT** descriu fișierele și directoarele.
- Dacă au o dimensiune suficient de mică (cel mult 1500 octeți), informațiile despre fișiere și directoare sunt înglobate într-o singură înregistrare **MFT**, iar în caz contrar înregistrările **MFT** vor conține adrese către una sau mai multe extensii.

Studii de caz – Windows

❑ New Tehnology File System (NTFS)

Structura unui volum NTFS



Studii de caz – Windows

❑ New Tehnology File System (NTFS)

- NTFS asigură suport pentru spațiul de nume și pentru dimensiuni mari ale fișierelor și volumelor,
- asigură un mecanism de securitate comparabil cu al Unix-ului prin intermediul listei de control a accesului (access control list – ACL) asupra fișierelor și a directoarelor la nivel de utilizator.
- Se asociază drepturi specifice utilizatorilor și grupurilor asupra acțiunilor pe care utilizatorii le pot efectua asupra fișierelor și directoarelor, permitând distribuția drepturilor fără a da acces la tot sistemul.

Studii de caz

□ Sisteme de fișiere distribuite

- permit distribuirea sistemului de fișiere pe mașini diferite fizic, păstrând totuși datele disponibile de pe aceste mașini.
- Un avantaj imediat este backup-ul ușor al datelor și managementul acestora.

Studii de caz

□ Sisteme de fișiere distribuite

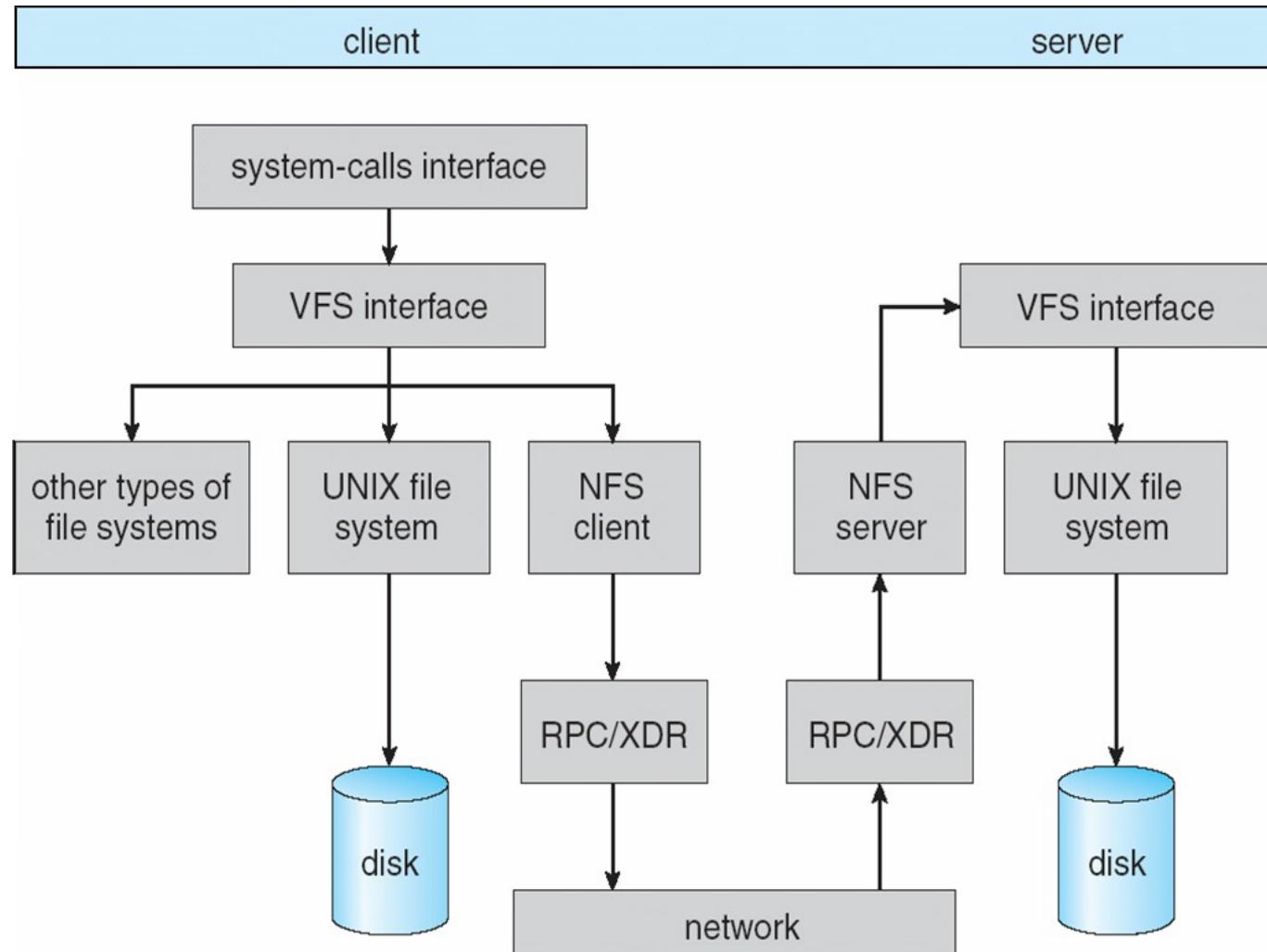
- NFS (Network File System dezvoltat de SUN Microsystems)
- RFS (dezvoltat de AT&T) pe platforme UNIX
- Microsoft DFS: NTFS si Active Directory.
- Novel NetWare File System
- AFS – AndrewFileSystem (Carnegie Mellon University si IBM)
- Open AFS – implementare open source a AFS
- CODA – derivat din AFS2, dezvoltat la Carnegie Mellon University
- Hadoop si CloudStore (implementari open source in Java/C++ a Google File System)
- Amazon S3
- Lustre –open source; dezvoltat de Cluster File Systems si preluat de SUN Microsystem
- PVFS – open source; devoltat de The Parallel Architecture Research Laboratory at Clemson University, Mathematics and Computer Science Division at Argonne National Laboratory, si Ohio Supercomputer Center.
- Global File System (GFS si GFS2) - RedHat
- General Parallel File System – IBM; suport pentru AIX,Linux, Windows

Studii de caz - NFS

❑ Sisteme de fișiere distribuite – NFS

- Foloseste RPC (remote procedure call)
- Pentru functionare trebuie să fie active serviciile:
 - portmap,
 - rpcgssd, rpcidmapd, rpcsvcgssd
 - nfs, nfslock
- NFS v3:
 - nu menține starea fișierelor deschise pe un server;
 - este mai puțin eficient, deoarece serverul de fișiere trebuie să redeschidă fișierele pentru fiecare tranzacție realizată prin rețea, lucru ce poate fi îmbunătățit prin folosirea cache-ului;
 - Refacerea după o cădere a unui server este transparentă clientului, deoarece serverul nu are nici o stare de refăcut.

Studii de caz - NFS

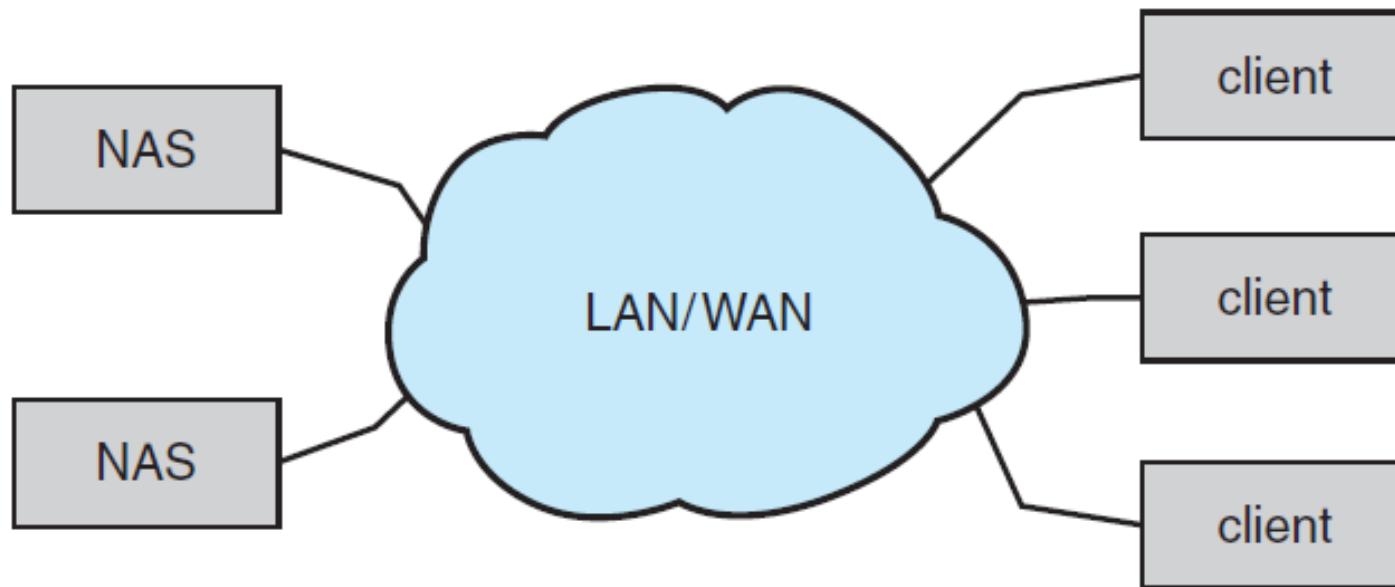


Dispozitive de stocare

- Sistemele de calcul acceseaza dispozitivele de stocare in doua moduri
 - Prin porturile de I/O (host-attached storage);
 - Prin conexiuni în rețea, adeseori referite ca network-attached storage

Network-Attached Storage (NAS)

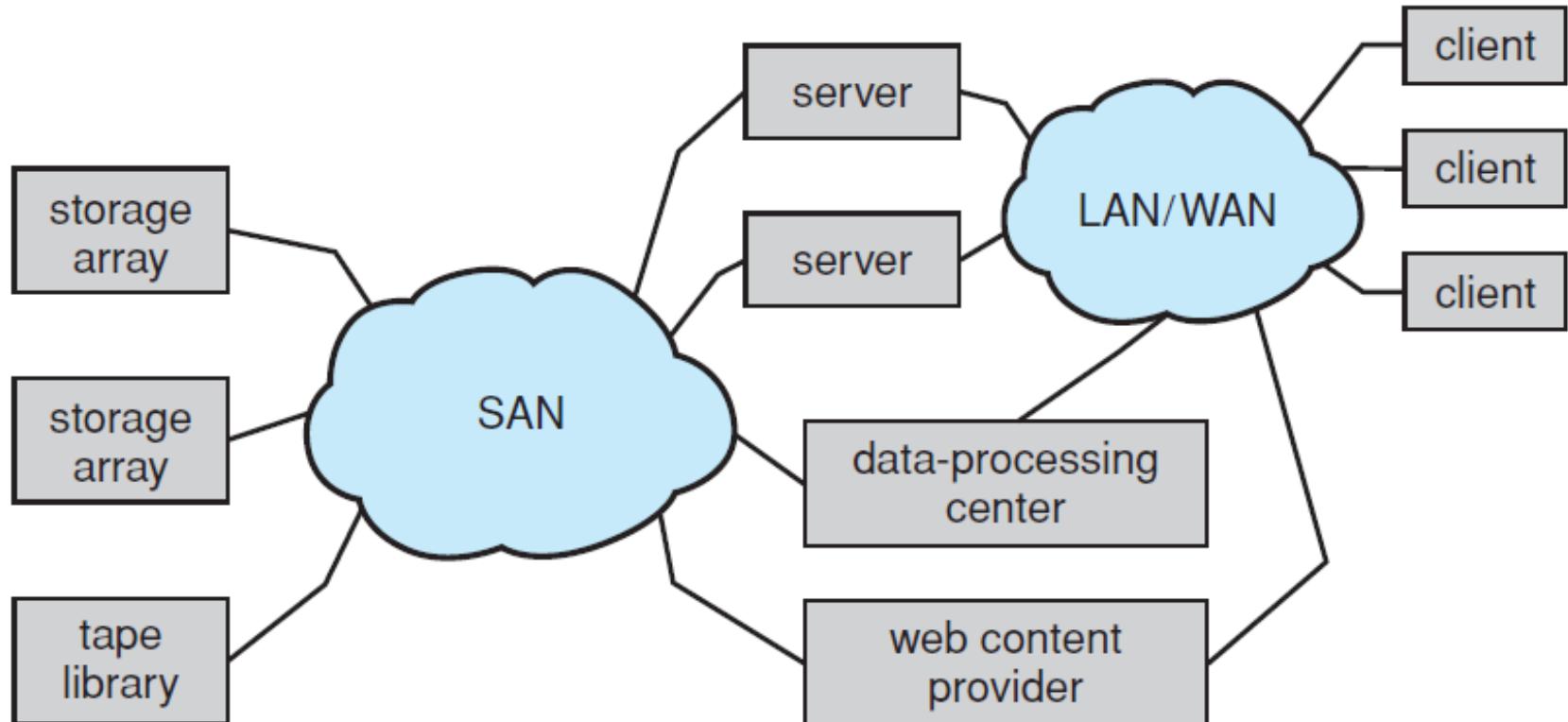
- Sistem de stocare specializat accesat peste interfețe de comunicații de date



Storage-Area Network

- storage-area network (SAN) – **rețea de date privată ce folosește protocoale de comunicații pentru dispozitive de stocare** în locul protocoalelor de comunicație în rețea, ce conectează servere și unități de stocare

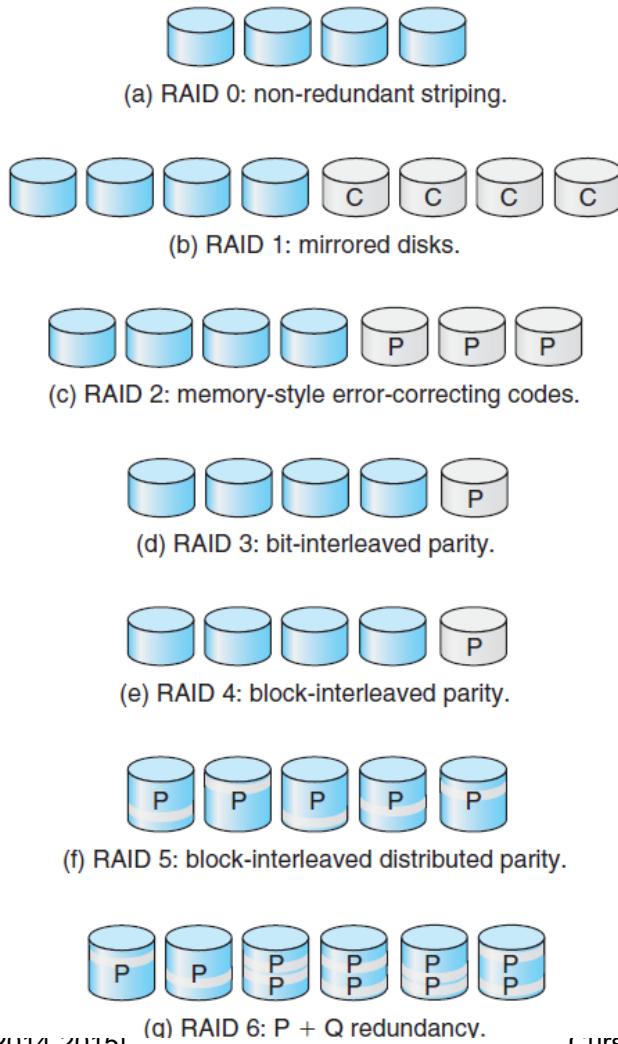
Storage-Area Network



Disponibilitatea datelor

- Pentru asigurarea disponibilității datelor sunt folosite diferite tehnici de organizare a discurilor numite **redundant arrays of independent disks (RAID)**

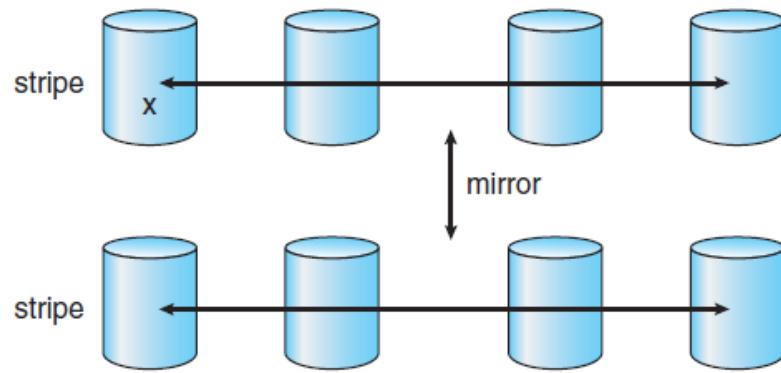
Configuratii RAID



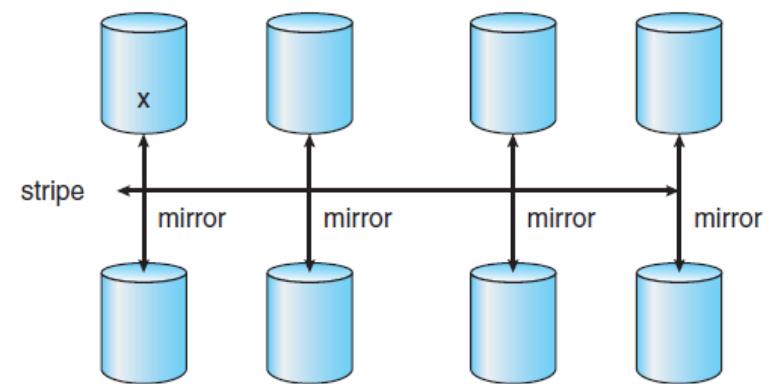
- P - error-correcting bits
- C - a second copy of the data

Sursa: Silberschatz A., Galvin P. -
Operating System Concepts, 9th Edition
, John Wiley & Sons, 2012

Configuratii RAID



□ RAID 0+1



□ RAID 1+0

Referințe

- ❑ http://en.wikipedia.org/wiki/Comparison_of_file_systems
- ❑ http://en.wikipedia.org/wiki/List_of_file_systems
- ❑ http://en.wikipedia.org/wiki/List_of_file_systems#Distributed_file_systems

Sisteme de Operare



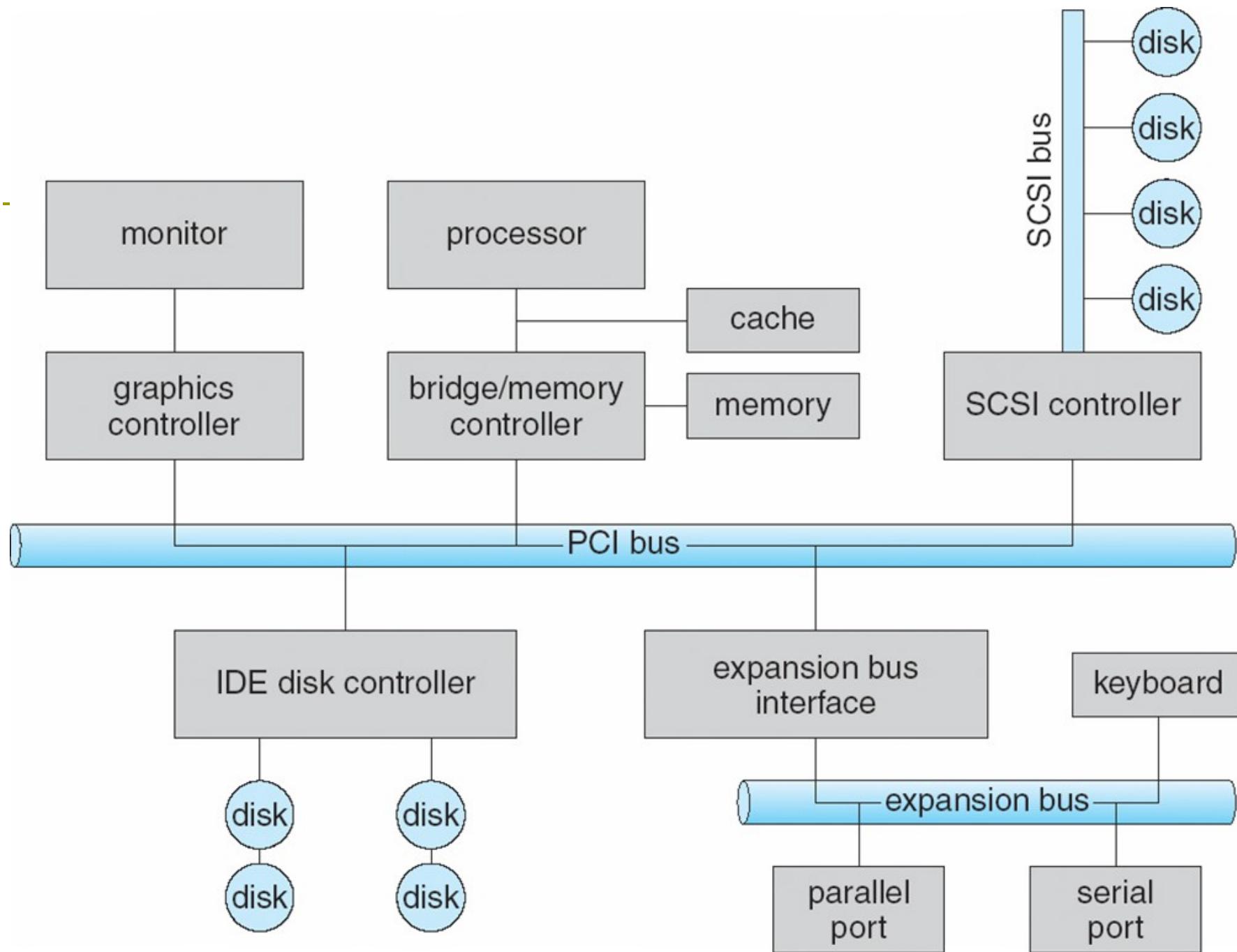
- ❑ Sistemul de I/O
- ❑ Accesul direct la memorie
- ❑ Buffer-ele sistemului de I/O

Sistemul de I/O

- ❑ generarea de comenzi către dispozitive
- ❑ tratarea întreruperilor
- ❑ tratarea erorilor posibile
- ❑ furnizarea unei interfețe utilizator cât mai ușor de utilizat și cu un grad cât mai ridicat de standardizare.
- ❑ este destul de dificil de realizat o generalizare din cauza multitudinii de dispozitive periferice
- ❑ dispozitivele se clasifică în funcție de sensul în care se vehiculează informația :
 - dispozitive de intrare (tastatura, mouse)
 - dispozitive de ieșire (display, imprimanta).
- ❑ Sunt dispozitive care pot vehicula informație în ambele sensuri (discurile și benzile magnetice, adaptoarele de rețea).

Sistemul de I/O caracteristicile periferelor

- viteza de acces
 - variază cu câteva ordine de mărime de la un dispozitiv la altul;
- unitatea de transfer
 - poate fi caracter, octet, cuvânt, bloc sau înregistrare, în funcție de natura dispozitivului periferic;
- reprezentarea datelor
 - datele pot fi codificate în diverse moduri, depinzând de diferite medii de intrare/iesire;
- operațiile posibile cu un anumit dispozitiv I/O
 - sunt determinate în principiu de sarcina pe care dispozitivul o îndeplinește în cadrul sistemului.
 - Operațiile de citire/scriere aplicabile în conjuncție cu anumite dispozitive nu au sens pentru altele.
- condițiile de eroare
 - au diferite cauze (de la lipsa hârtiei la eroarea codului de control pentru un transfer de date) și implicit modalitățile de tratare sunt diferite.



Obiectivele proiectarii unui sistem de I/O

- independența față de codul de caractere
 - sistemul de I/O trebuie să recunoască diversele coduri de caractere utilizate de dispozitivele periferice și să prezinte datele într-un format standard.
- independența față de dispozitivele periferice
 - posibilitatea scrierii programelor, astfel încât să nu necesite modificări ale codului atunci când se modifică tipul dispozitivului periferic față de cel prevăzut inițial, lucru ce presupune furnizarea unor operații a căror sintaxă și semantică să fie cât mai asemănătoare pentru o clasă cât mai mare de dispozitive periferice.
 - Aici apare și denumirea uniformă a dispozitivelor periferice din cazul UNIX și Windows, unde fiecare dispozitiv are asociat un fișier, dispozitivele fiind denumite prin intermediul numelui fișierului asociat.
- eficiența operațiilor
 - dispozitivele periferice pot introduce penalizări sub aspectul timpului de acces, datorate atât diferenței mari dintre viteza de calcul a unității centrale și cea de transfer a datelor precum și dintre viteza de transfer a datelor și viteza ansamblurilor mecanice mobile ce intră în componența multora dintre dispozitivele periferice.
 - au apărut mecanisme care să conducă la creșterea eficienței (DMA, spooling etc.).

Evoluția sistemului de I/O

- ❑ Procesorul controla direct dispozitivul periferic
- ❑ controller-ele de I/O:
 - procesorul utilizează mecanismul programmed I/O fără întreruperi
 - procesorul trebuie să gestioneze detaliile lucrului cu dispozitivul de I/O
- ❑ controller-ele sau modulele de I/O cu întreruperi:
 - procesorul nu mai pierde timp așteptând terminarea operației de I/O
- ❑ apariția DMA (Direct Memory Access)
 - blocurile de date sunt mutate direct în memorie fără implicarea procesorului
 - procesorul este implicat numai la începutul și sfârșitul operației
- ❑ modulul de I/O este un procesor separat
 - apar canalele de I/O
 - este posibil accesul la memoria principală pentru instrucțiuni
- ❑ procesoarele de I/O cu propria memorie
 - este un computer în adevăratul sens al cuvântului
 - pot fi controlate un set mare de dispozitive de I/O
 - o utilizare frecventă este controlul comunicației cu terminalele interactive

Structura sistemului de I/O

□ Structura ierarhică

- nivelurile sunt caracterizate de nivelul de complexitate, timp de acces, nivelul de abstractizare
- nivelurile inferioare pot lucra la intervale de timp de ordinul nanosecundelor

□ Structura logică

- dispozitive I/O logice – toate dispozitivele sunt privite ca resurse logice (permis operațiile open, read, write)
- Dispozitivele I/O – operațiile și datele sunt convertite în secvențe de instrucțiuni I/O
 - Se pot folosi buffer-e pentru a crește viteza de lucru
- Planificare și control – creează și gestionează cozile de așteptare pentru operațiile I/O și realizează planificarea acestor operații
 - La acest nivel sunt gestionate întreruperile

Structura hardware a sistemului de I/O

- Dispozitivele de I/O se pot clasifica în două categorii:
 - dispozitive bloc(discul):
 - stochează informația sub forma unor blocuri de dimensiune fixă, fiecare având asociată o adresă cu ajutorul căreia poate fi accesat individual.
 - dispozitive caracter(imprimanta, mouse, terminale, adaptoare de rețea):
 - lucrează cu siruri de caractere cărora nu le conferă o structură pe blocuri;
 - nu pot fi adresate individual și nu pot constitui obiectul unor operații de căutare.

Structura hardware a sistemului de I/O

- unitățile de bandă
 - nu se pot implementa în mod eficient operații de acces aleator, deși sunt structurate bloc și permit operații de căutare.
- ceasul de timp real
 - nu poate fi încadrat în nici una din categoriile de mai sus
 - are sarcina de a genera întreruperi la intervale de timp bine stabilite.

Controller

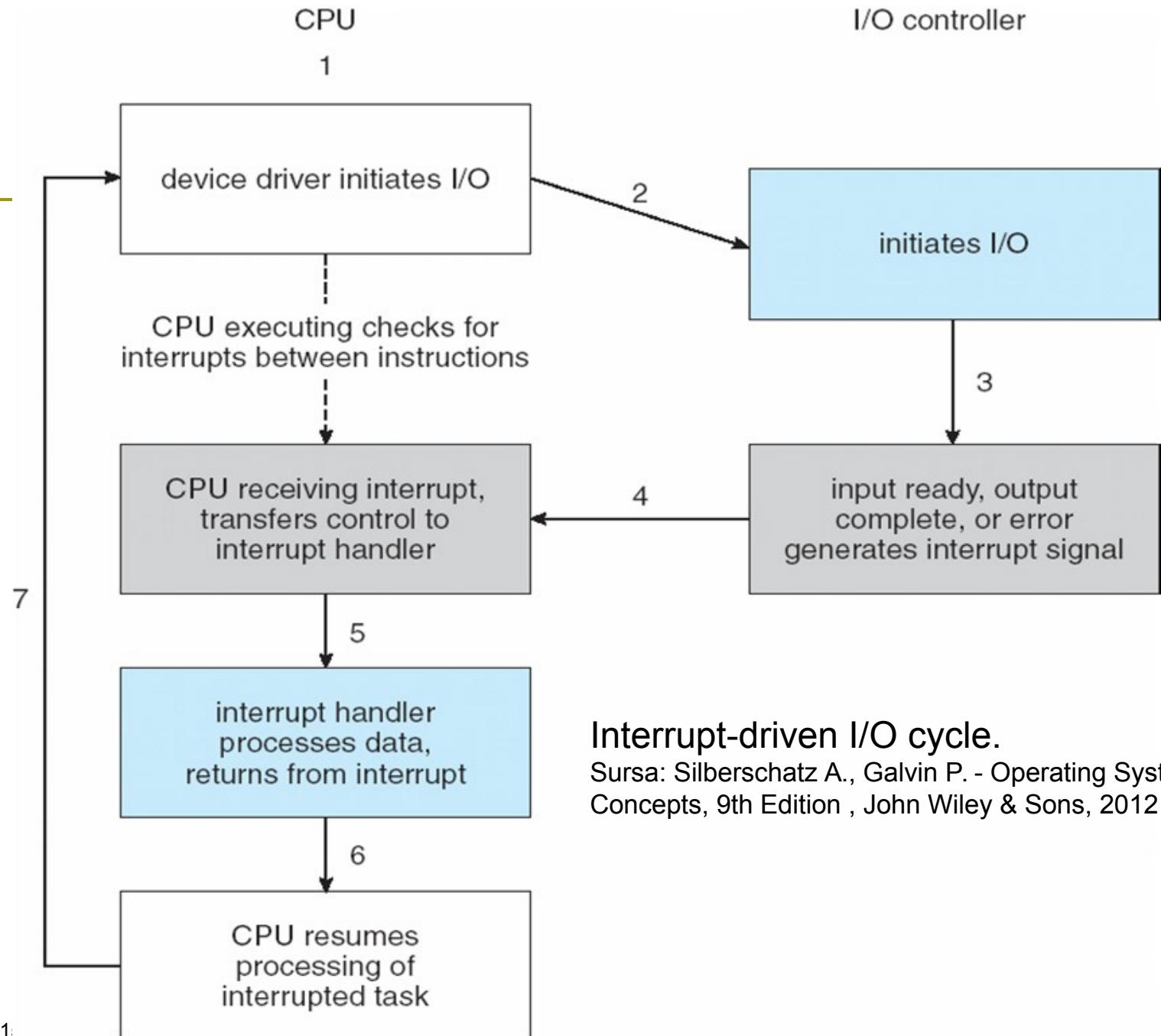
- Dispozitivele de I/O sunt formate dintr-o componentă mecanică și una electronică numită controller.
- Un controller poate gestiona mai multe dispozitive identice.
- Distincția între controller și dispozitivul propriu-zis este necesară, deoarece sistemul de operare interacționează cu controller-ul.

Controller

- operația de citire de pe un disc magnetic:
 - controller-ul este cel care poziționează capetele și citește de pe disc un sir de biți care cuprinde:
 - un antet ce conține informațiile înscrise la momentul formatării discului: numărul cilindrului, al sectorului, dimensiunea unui sector;
 - biți de informație stocați în sectorul respectiv;
 - un cod de corecție a erorilor.
 - Controller-ul asamblează bit cu bit un bloc de date căruia îi calculează suma de control, ce trebuie să fie identic cu codul citit de pe disc și abia după aceea blocul de date respectiv este trecut în memoria principală.

Comunicația dintre controller și unitatea centrală

- se realizează prin intermediul unor registre, care de cele mai multe ori fac parte din spațiul de adrese de memorie (sunt mapate în memorie – memory mapped I/O).
 - registrele mapate în memorie
 - se accesează la fel ca orice locație de memorie, singura diferență fiind timpul de acces mai redus.
 - sunt utilizate de sistemul de operare pentru a înscrie parametri și comenzi și pentru a citi starea dispozitivului respectiv și codurile de eroare.
- O comandă odată acceptată de controller, SO lasă dispozitivul să o execute în timp ce el planifică alte taskuri.
- După terminarea operației, controller-ul generează o întrerupere, care permite SO să preia controlul și să analizeze rezultatul.



Interrupt-driven I/O cycle.

Sursa: Silberschatz A., Galvin P. - Operating System Concepts, 9th Edition , John Wiley & Sons, 2012

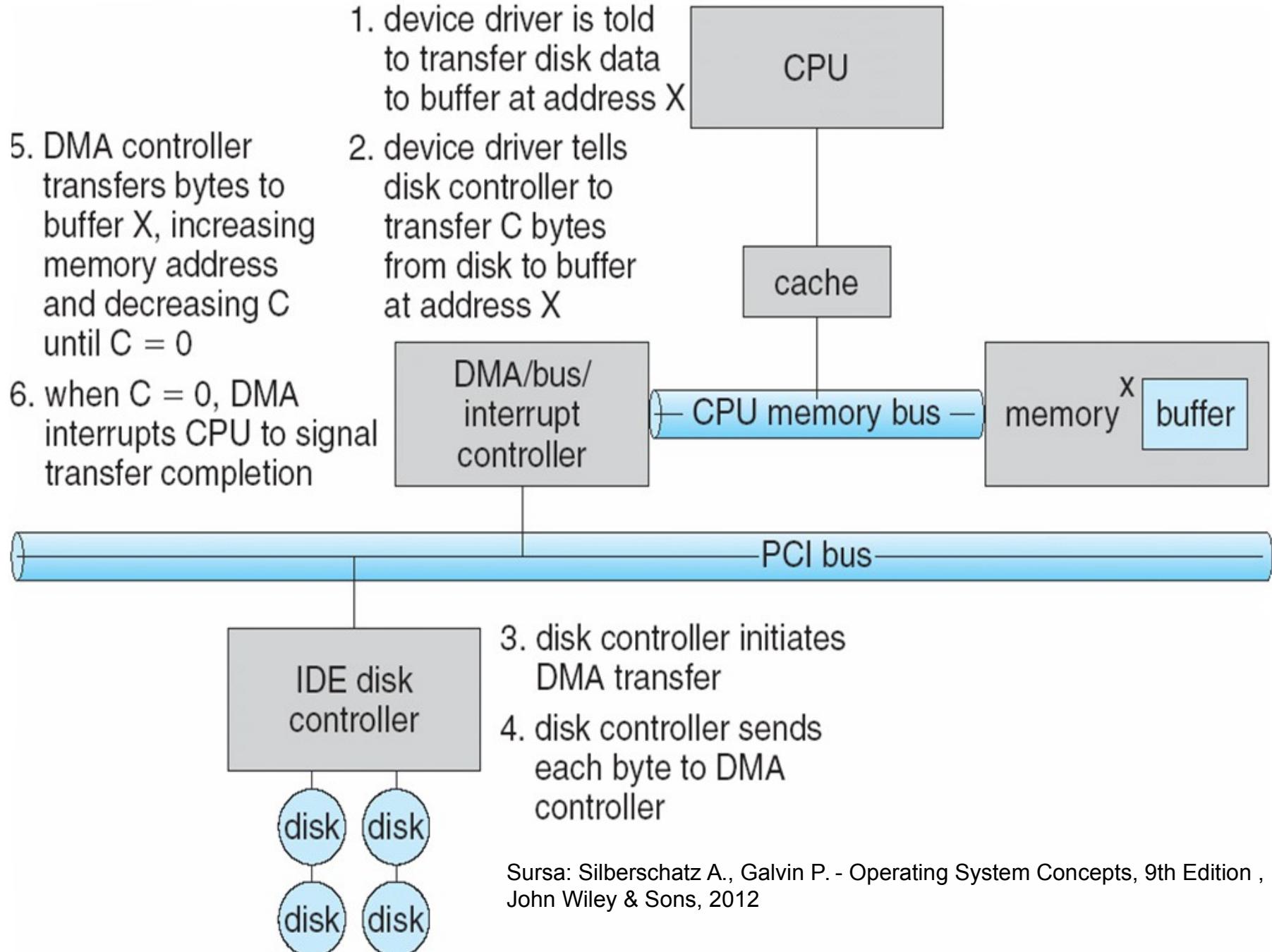
PC – adresele porturilor I/O (lista parțială)

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

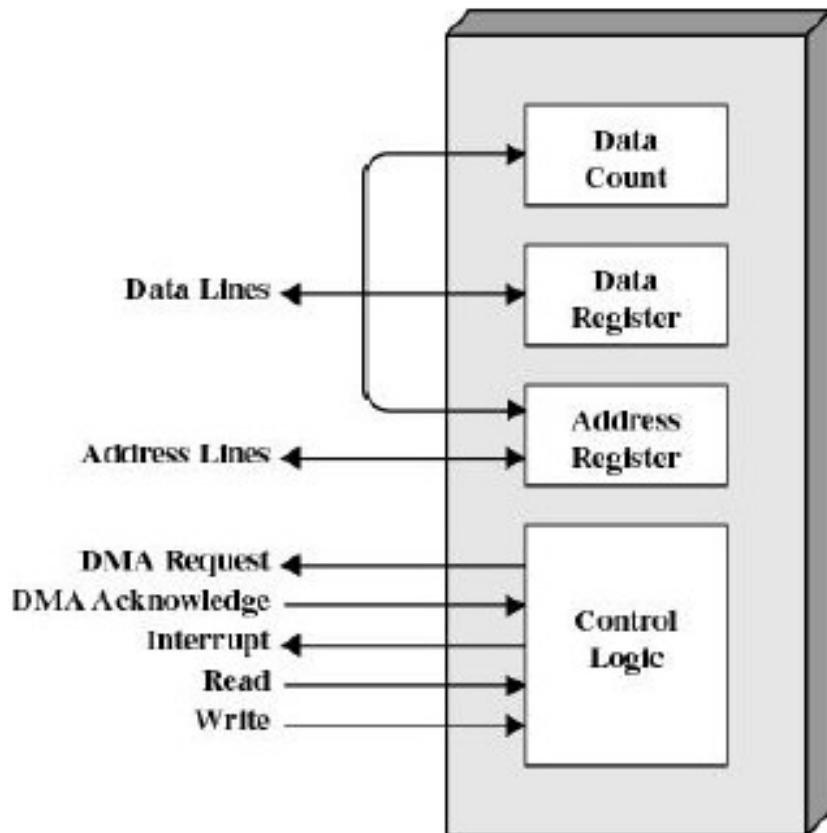
Accesul direct la memorie (Direct Memory Access – DMA)

- operația de copiere în memorie este efectuată de către controller și nu de către unitatea centrală.
 - În acest mod se obține o utilizare mai eficientă a acesteia.
 - Utilitatea acestui mecanism este justificată de necesitatea transferului unui volum mare de date.
- Fără DMA:
 - Dacă un controller de disc primește comanda de citire de pe disc a unui bloc (corespunzător unuia sau mai multor sectoare) precizat prin adresă, convertește adresa într-un număr de cilindru-sector-cap.
 - Conținutul blocului este citit bit-cu-bit în buffer-ul intern al controller-ului și verificat dacă nu are erori, după care controller-ul semnalizează printr-o intrerupere terminarea operației, urmând ca sistemul de operare să transfere cuvânt-cu-cuvânt conținutul buffer-ului intern în memoria principală lucru ce duce la o utilizare ineficientă a tipului de lucru a unității centrale.
 - Acest mecanism se mai numește și programmed I/O.

5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion

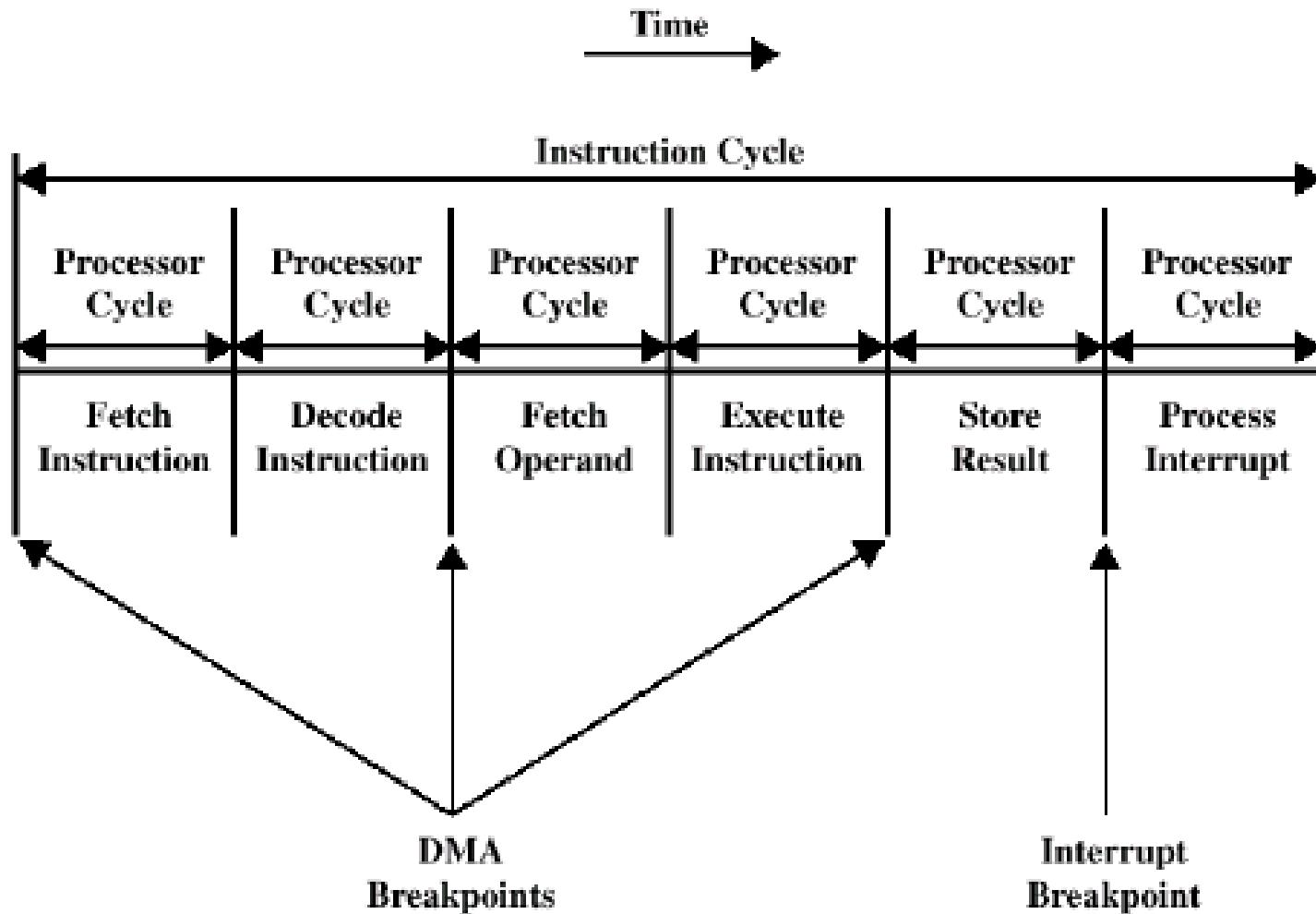


DMA – schema bloc



□ Conținutul blocului de date este stocat în buffer-ul intern și verificat, controller-ul aşteaptă eliberarea magistralei sistemului și transferă întregul bloc în memorie.

Punctele de oprire a execuției ciclului de instrucțiuni la apariția DMA și a întreruperilor



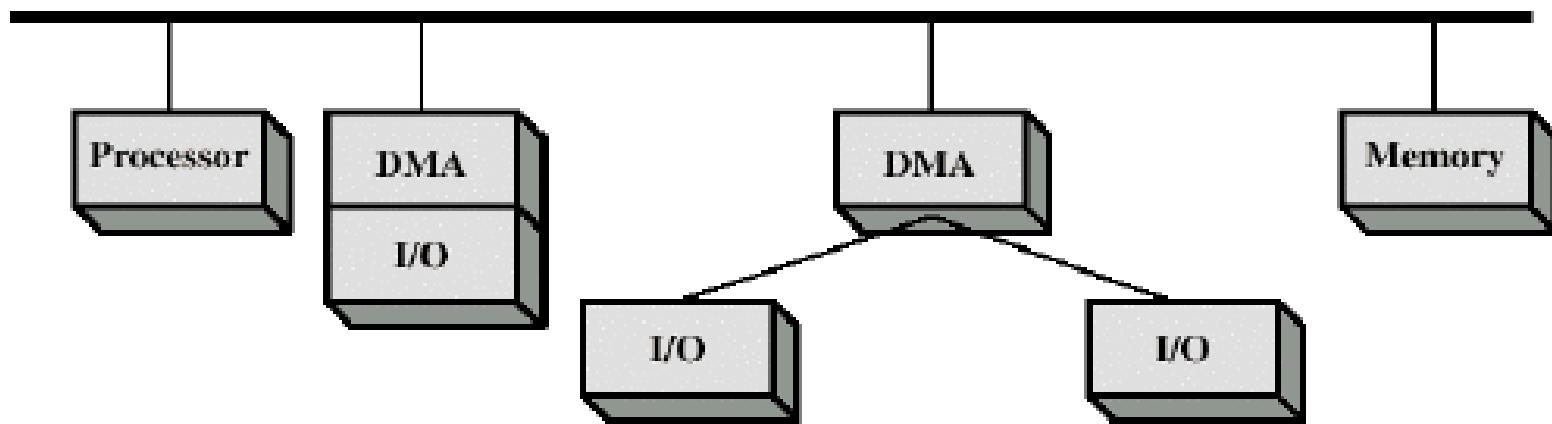
Variante de implementare ale DMA

- ❑ Single bus cu modulul DMA separat
 - toate modulele partajează aceeași magistrală
 - ieftină, dar ineficientă



Variante de implementare ale DMA (2)

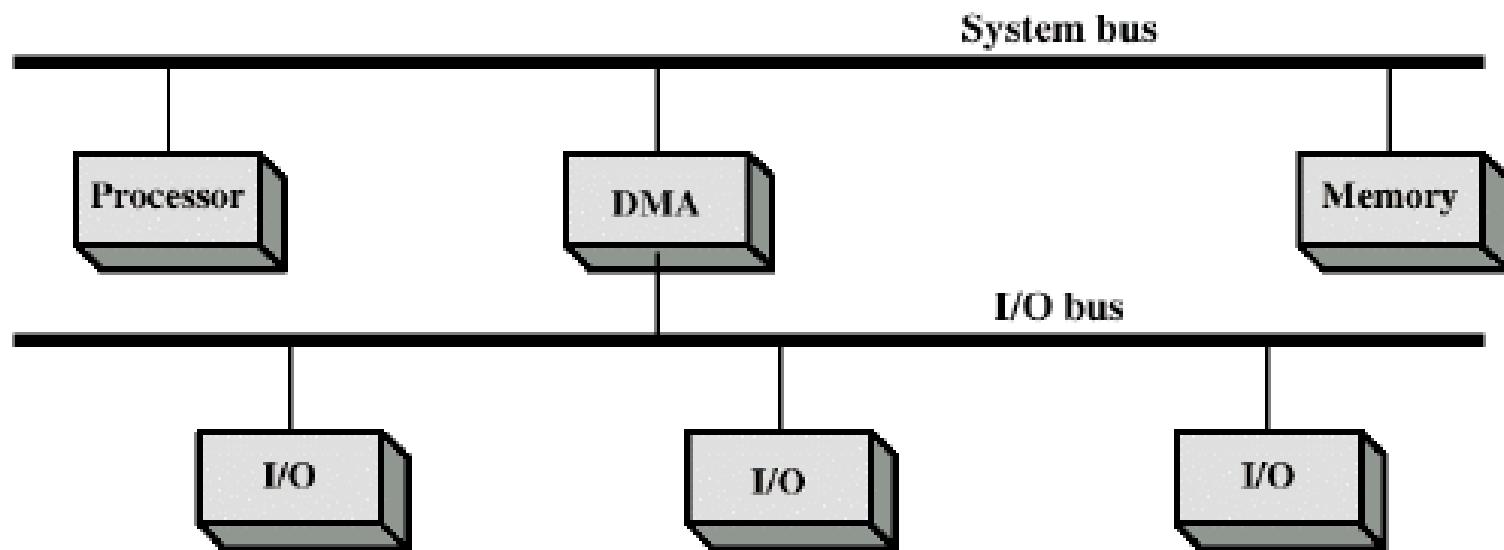
- ❑ Single bus cu modulele DMA-I/O integrate
 - există o cale de comunicație separată între modulul DMA și modulele de I/O



Variante de implementare ale DMA (3)

□ bus I/O separat

- o singură interfață între modulele DMA și I/O
- ușurează expandarea configurației



Utilizarea DMA

- are implicații și asupra organizării informației pe discul magnetic:
 - Datorită mișcării continue de rotație a discului, citirea unui sector se realizează numai în timpul cât el trece pe sub capul de citire/scriere.
 - După citire, controller-ul verifică datele și le transferă în memoria principală, timp în care, în general, nu poate citi și următorul sector care tocmai trece pe sub capul de citire/scriere (se presupune că se transferă o cantitate de date mai mare decât cea conținută într-un sector).
 - La încheierea transferului, controller-ul trebuie să aștepte o perioadă de timp până când următorul sector ce trebuie citit va ajunge sub capul de citire/scriere.

Utilizarea DMA (2)

- Corelarea vitezei de transfer a informației cu viteza de rotație a discului, în scopul minimizării acestei așteptări, conduce la ideea de întrețesere (interleaving):
 - constă în plasarea a două sectoare adiacente din punct de vedere logic la o distanță de câteva sectoare fizice, astfel încât următorul sector logic să se găsească sub capul de citire/scriere exact în momentul încheierii transferului sectorului logic anterior.
- Mecanismul DMA este aplicabil și în cazul operațiilor de scriere.
 - Adresa fizică și lungimea zonei memorie furnizate ca parametri controller-ului, vor indica localizarea și dimensiunea datelor ce trebuie preluate de controller și scrise pe disc.

Buffer-ele sistemului de I/O

- Operațiile de I/O din spațiul de memorie al utilizatorului duc la apariția următoarelor probleme:
 - paginile care păstrează data ce trebuie transferată trebuie să rămână în memorie
 - apar limitări la adresa acțiunilor sistemului de operare
 - procesele nu pot fi transferate complet în swap sau pot apărea blocaje (deadlock):
 - procesele așteaptă terminarea unei operații de I/O
 - sistemul de I/O așteaptă ca procesul să fie adus din swap
- Există posibilitatea de a citi în avans unele date (read in advance), precum și de a întârzia scrierea (se combină mai multe cereri de scriere pe disc atunci când se realizează scrierea).
- Sistemul de operare atribuie un singur buffer în memoria principală pentru operațiile de I/O.

Buffer-ele sistemului de I/O

- Procesele utilizator pot procesa un bloc de date, în timp ce următorul bloc este citit.
- Datorită tehnicii de swapping, blocurile de date care trebuie să fie trecute în spațiul de memorie al utilizatorului sunt trecute în memoria sistemului.
- Sistemul de operare păstrează informațiile legate de atribuirea buffer-elor sistem proceselor utilizatorilor.
- Transferul de date către dispozitivele de I/O se face prin scrierea de către procese în buffer-e și apoi are loc transferul efectiv al datelor.

Buffer-ele sistemului de I/O

□ Buffer-e de tip bloc

- transferul datelor se face în buffer,
- blocurile sunt mutate în memorie atunci când este necesar;
- un alt bloc este transferat în buffer.

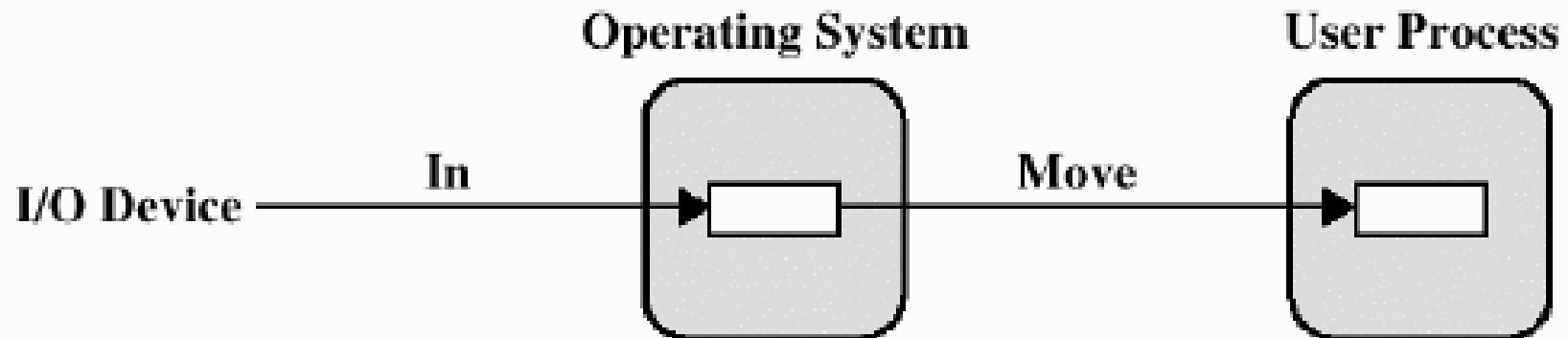
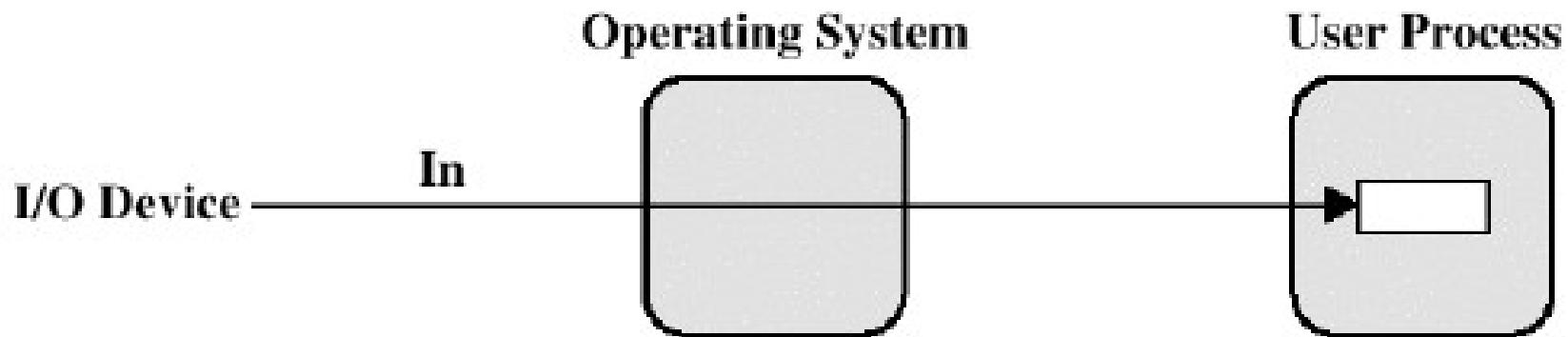
□ buffer-e de tip caracter

- Exemplu: introducerea datelor de la terminal.
Scrierea la terminal se face linie cu linie.

Implementarea buffer-elor sistemului de I/O

□ Un singur buffer

- Sistemul de operare atribuie un singur buffer în memoria principală pentru operațiile de I/O.



Implementarea buffer-elor sistemului de I/O (2)

- Dispozitivul de I/O transferă datele în buffer-ele sistemului și SO copiează date în spațiul de memorie al utilizatorului.
- Immediat ce un transfer este terminat se încearcă citirea în avans a următorului bloc.
- Procesele utilizator pot lucra cu un singur bloc de date, în timp ce următorul bloc este citit.
- Timpul de transfer al unui bloc:

$$\mathbf{max[C,T] + M}$$

- C = timp de calcul,
 - T = timp de realizare a operației de I/O,
 - M = timpul de transfer cu buffer-ul
- Observație: fără buffer-e timpul este C+T

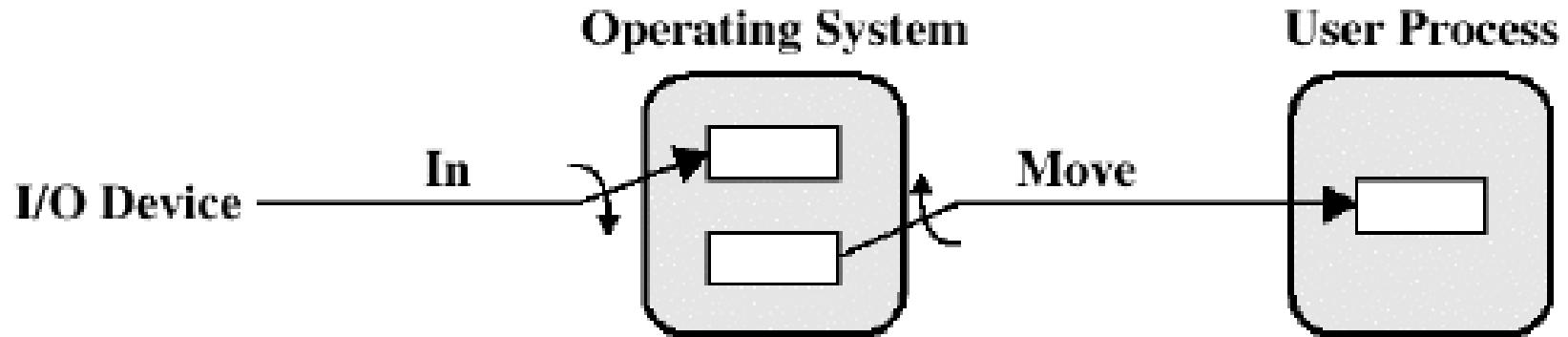
Implementarea buffer-elor sistemului de I/O (3)

- ❑ Sistemul de operare va urmări buffer-ele atribuite utilizatorilor;
- ❑ nu este de dorit trecerea în swap a unui proces care aşteaptă terminarea unei operații de I/O.
- ❑ Totuși, sistemul de operare are posibilitatea de a transfera procesele în swap fără ca acest lucru să afecteze operațiile de I/O.
- ❑ Pentru dispozitivele de tip caracter buffer-ele pot transfera biți (bytes) sau linii.

Implementarea buffer-elor sistemului de I/O (4)

□ Buffer dublu:

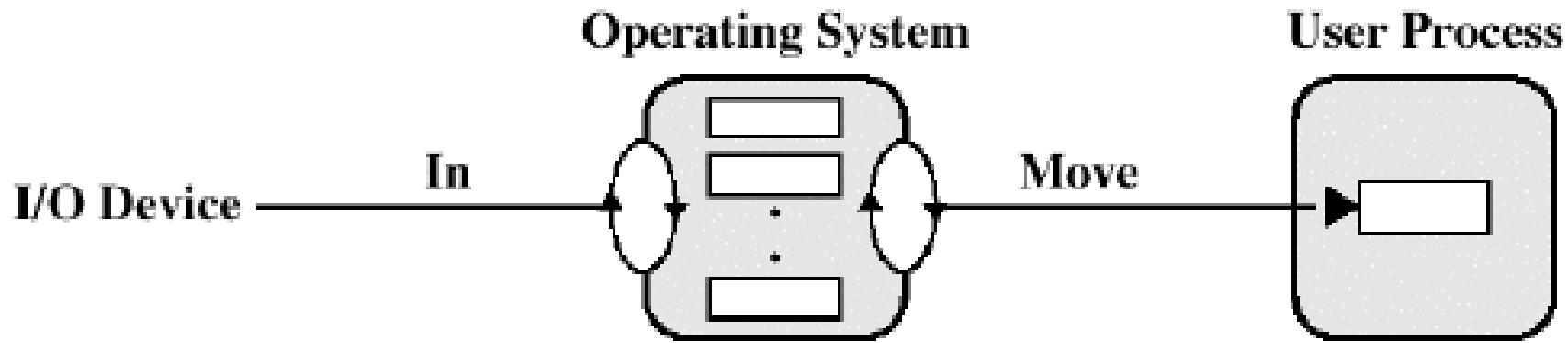
- Un proces poate transfera date dintr-un buffer, în timp ce sistemul de operare umple sau golește celălalt buffer.



Implementarea buffer-elor sistemului de I/O (5)

□ Buffer circular:

- Sunt folosite mai mult de două buffer-e, fiecare buffer constituind o unitate de buffer-e circulare.



Structura software a sistemului de I/O

- Software-ul destinat gestiunii dispozitivelor periferice este structurat pe patru niveluri:
 - rutinele de tratare a intreruperilor;
 - drivere-ele asociate dispozitivelor periferice;
 - programe sistem independente de dispozitive;
 - primitive de nivel utilizator.

Rutinele de tratare a întreruperilor

- Rolul rutinelor de tratare a întreruperilor
 - identificarea sursei întreruperii (adică dispozitivul care a generat-o)
 - de a reinițializa linia de întrerupere respectivă
 - memorarea stării dispozitivului (în cazul în care aceasta va fi necesară ulterior)
 - “trezirea” (printr-o operație signal) procesului care a inițiat operația de I/O.

Driver-e

- Un driver acceptă cereri la nivelul software superior și le transpune în comenzi pe care le transmite controller-ului, înscriind valori corespunzătoare în registrele acestuia din urmă.
- Driver-ele înglobează în totalitate acea parte a codului care este dependentă de dispozitivele periferice asociate, fiind totuși capabile să gestioneze mai multe tipuri de dispozitive periferice înrudite.

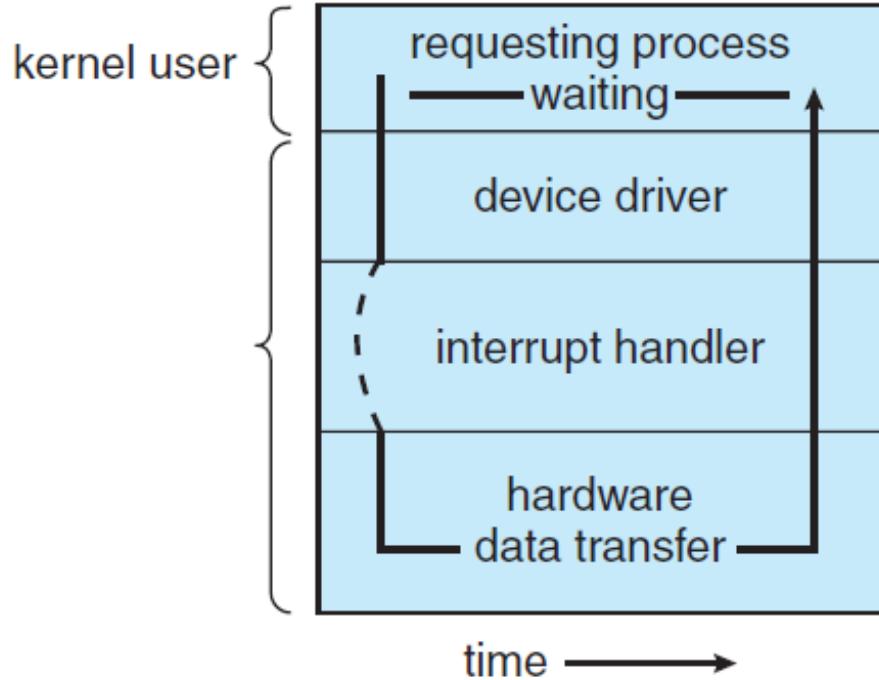
Primitive de nivel utilizator

- ❑ au rolul de a transfera parametrii lor apelurilor sistem pe care le inițiază, iar uneori oferă posibilitatea formatării datelor de intrare/ieșire
- ❑ pot lucra în mod sincron sau asincron
 - Sincron:
 - ❑ returnează parametrii numai după ce operația de I/O a fost realizată efectiv, fapt care o recomandă pentru a fi utilizată în cazul operațiilor cu durată redusă sau care poate fi estimată.

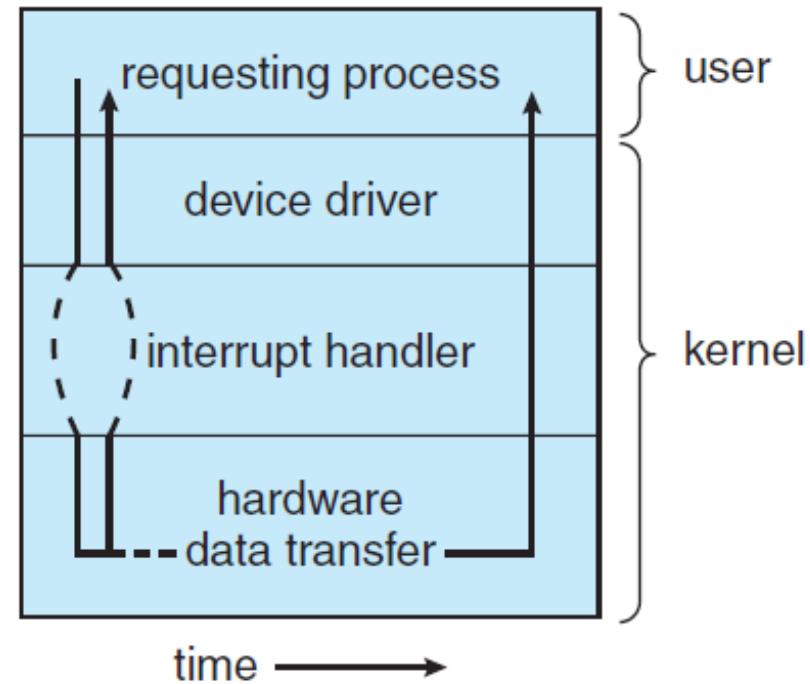
Primitive de nivel utilizator

■ Asincron :

- are doar rolul de inițiere a operației, ea returnând imediat un cod de eroare în cazul în care operația nu poate fi efectuată sau 0 în caz de succes.
- un proces își poate continua execuția în paralel cu efectuarea operației, testând periodic starea de evoluție a acesteia.
- Momentul în care operația se încheie este marcat printr-o notificare (procedură definită de utilizator și asociată unui eveniment) pe care sistemul de operare o lansează automat.
- O problemă care apare la folosirea acestui tip de primitive este disponibilitatea buffer-ului ce definește zona de memorie în care se află datele ce urmează a fi transferate: procesul inițiator trebuie să evite citirea/scrierea conținutului acestei zone atât timp cât operația nu a fost terminată, sarcină ce revine, în general, programatorului.
- Acest tip de primitive sunt folosite în cazul operațiilor cu o durată mare sau greu de estimat.



Sincron



Asincron

Spooling (Simoultaneous Peripheral Operation On-Line)

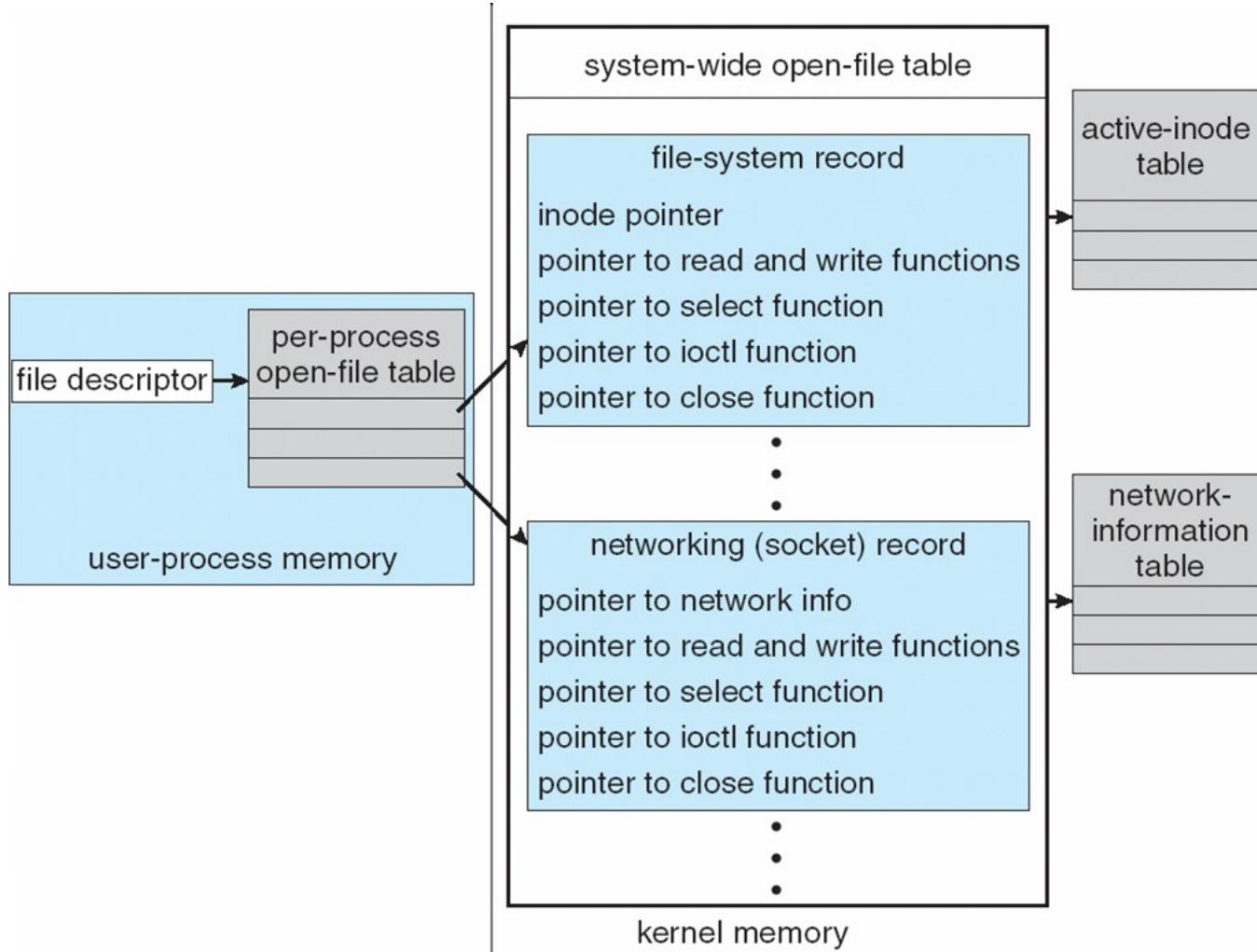
- ❑ constă în stocarea informației destinate transferului pe un mediu intermediar (de regulă, pe disc) și deblocarea procesului inițiator.
- ❑ Dispozitivele periferice dedicate au dus la apariția unor procese specializate în gestionarea cererilor pe care alte procese din sistem le formulează către dispozitivul respectiv și a unei zone (pe disc) reprezentând mediul intermediar de stocare a datelor care fac obiectul transferului.

Spooling (Simoultaneous Peripheral Operation On-Line)

❑ Exemplu - Imprimanta

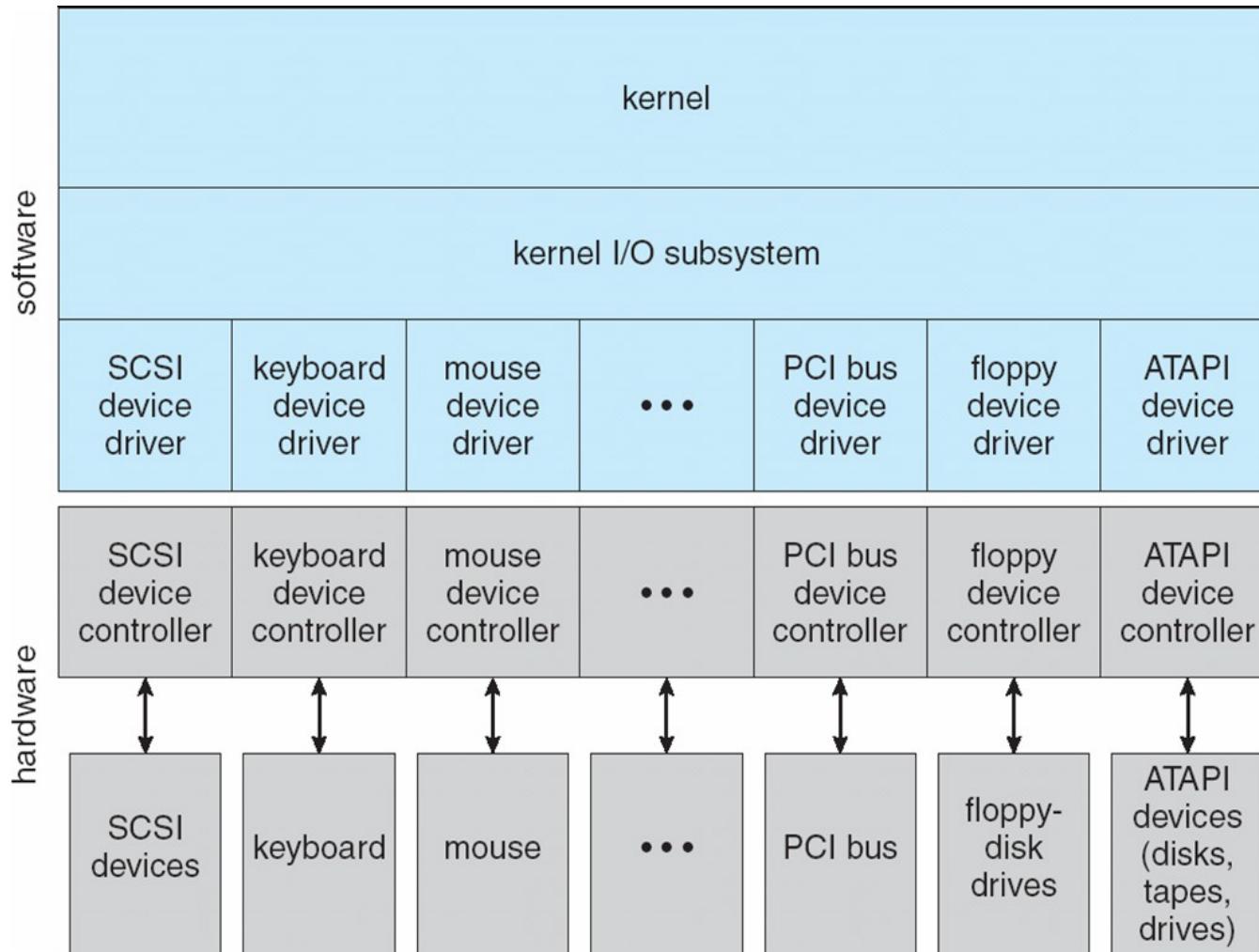
- se realizează prin stocarea într-un fișier a textului pe care procesul dorește să-l tipărească, urmată de deblocare procesului.
- Un alt proces, destinat strict gestionării cererilor către imprimantă (printer daemon), va iniția tipărire efectivă a textului în momentul în care imprimanta va deveni liberă.

Studiul de caz UNIX I/O Kernel subsystem

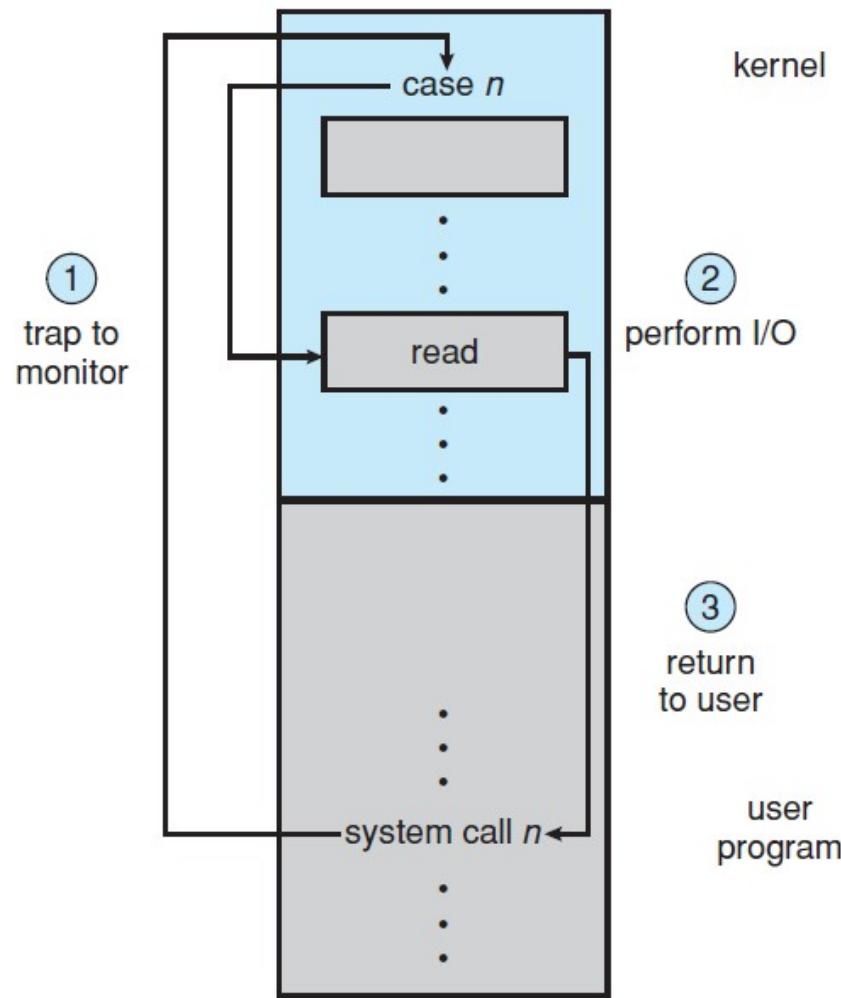


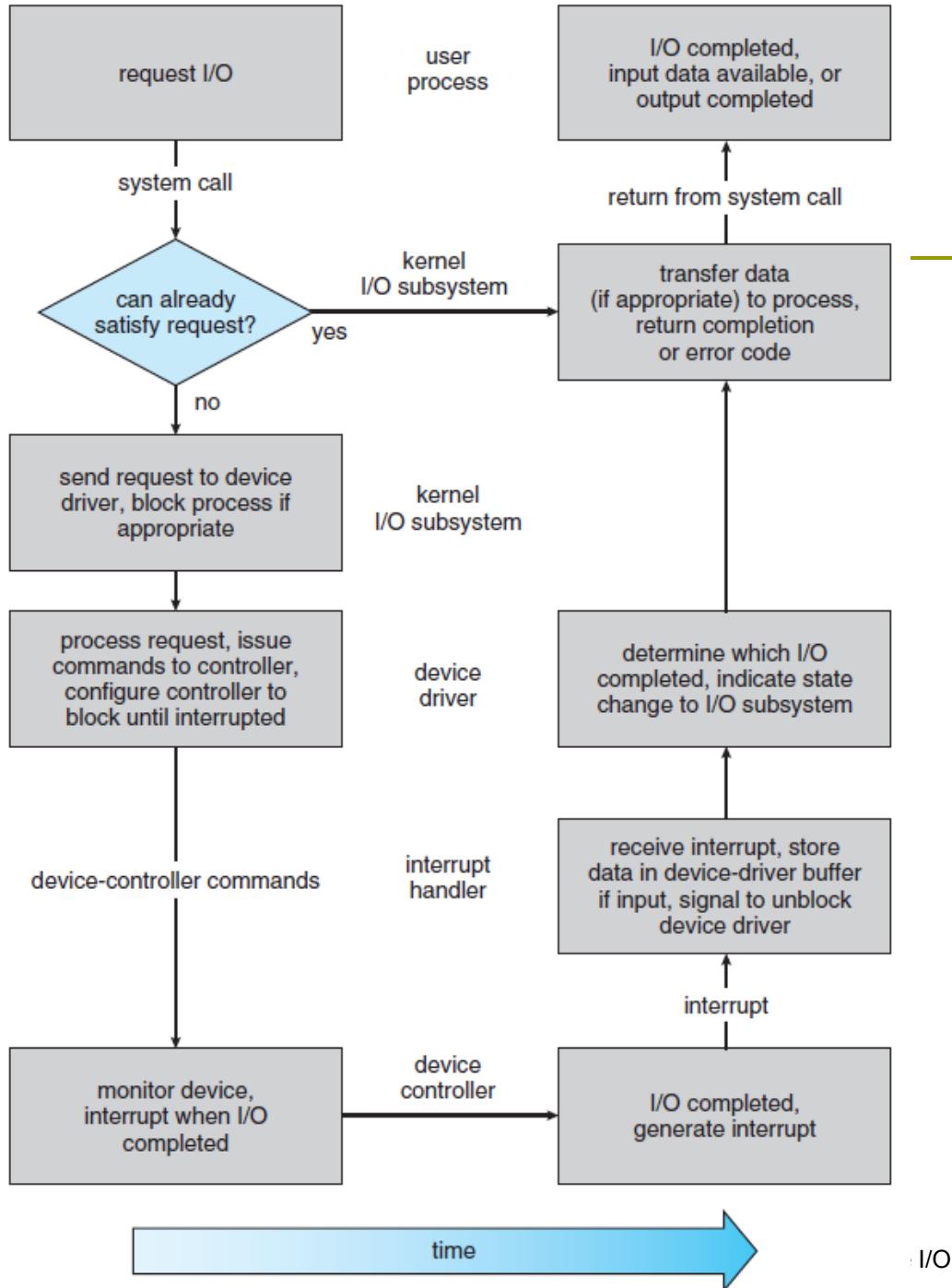
Studiu de caz

UNIX I/O Kernel subsystem



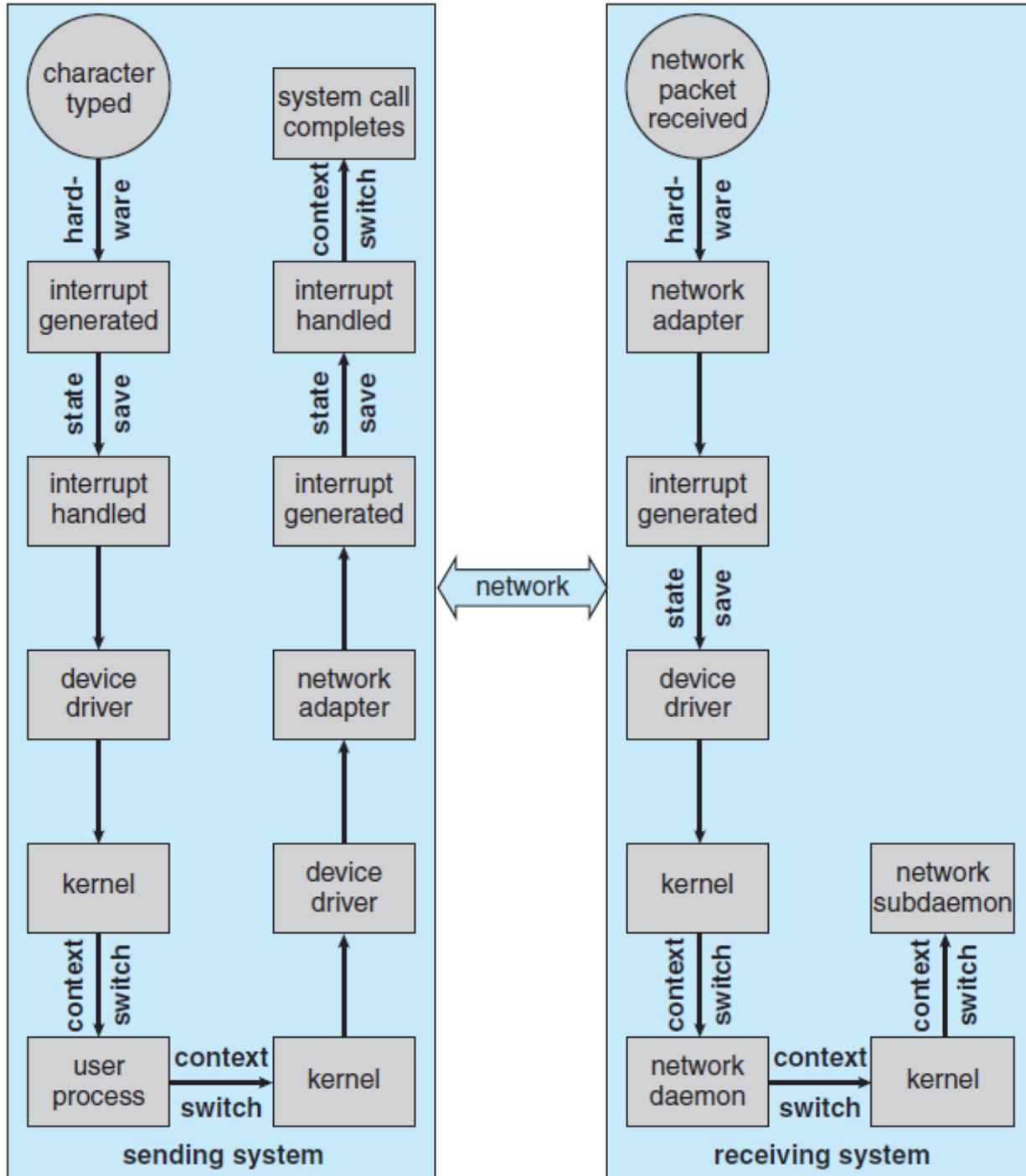
Utilizarea apelurilor sistem pentru I/O





The life cycle of an I/O request.

Sursa: Silberschatz A., Galvin P. -
Operating System Concepts, 9th Edition ,
John Wiley & Sons, 2012



Intercomputer communications.

Sursa: Silberschatz A., Galvin P. - Operating System Concepts, 9th Edition , John Wiley & Sons, 2012

Performanța sistemului de I/O

- Sistemul de I/O are un rol important în performanța sistemului
 - generează cereri către CPU să execute codul driverelor și să planifice procesele eficient pe măsură ce acestea se blochează și se deblochează.
 - Schimbările de context foarte dese introduc încărcare suplimentară asupra CPU și a cache-ului.
 - Sistemul de I/O dezvăluie problemele din mecanismele de tratare a întreruperilor din kernel.
 - Sistemul de I/O încarcă magistrala de date la care este conectată memoria atât în timpul transferului de date dintre controller și memoria fizică cât și în timpul copierii informațiilor din bufferele nucleului și spațiul de adrese al aplicațiilor.
 - Deși sistemele moderne pot gestiona un număr foarte mare de întreruperi pe secundă, gestiunea întreruperilor este un task costisitor. Fiecare întrerupere generează o schimbare a stării sistemului - tratarea întreruperii și revenirea la starea anterioară.