**Funcțiile nucleului unui SO**

***Nucleul SO - colecție de funcții care sunt necesare funcționării primare a sistemului de calcul***

Cele mai evidente: - funcțiile pentru intrare-ieşire (I/O)- utilizatorul apelează o astfel de funcție printr-un ***apel de sistem*** – expl procedura cu mnemonic "***write_file(…)***"

***Nucleul MS-DOS –***

**-** suită de funcții

***- BIOS, (Basic Input-Output System)*** funcții înscrise în memoriile nevolatile

- funcții pentru accesul discului - *structură de directoare şi fişiere*

**Funcțiile nucleului unui SO**

***Nucleul SO multitasking***

***Apelurile sau comenzile din programele utilizator*** care se adresează SO servesc pentru ***comunicarea între aceste programe*** şi ***SO pentru realizarea următoarelor acțiuni:***

– activarea task-urilor, planificarea în raport cu timpul a task-urilor, sincronizarea, suspendarea sau terminarea execuției task-urilor;

– alocarea sau modificarea priorității task-urilor;

– inițierea operațiilor de I/O, inclusiv accesarea sistemelor de fişiere;

– obținerea de informații despre stările task-urilor;

– obținerea de informații despre timpul sistemului etc.

Aceste ***apelări de servicii SO*** sunt tratate ***de rutine*** componente ale ***SO***

**Funcțiile nucleului unui SO**

**Apelurile de sistem UNIX oferă portabilitate şi o interfață eficientă aplicațiilor utilizator**

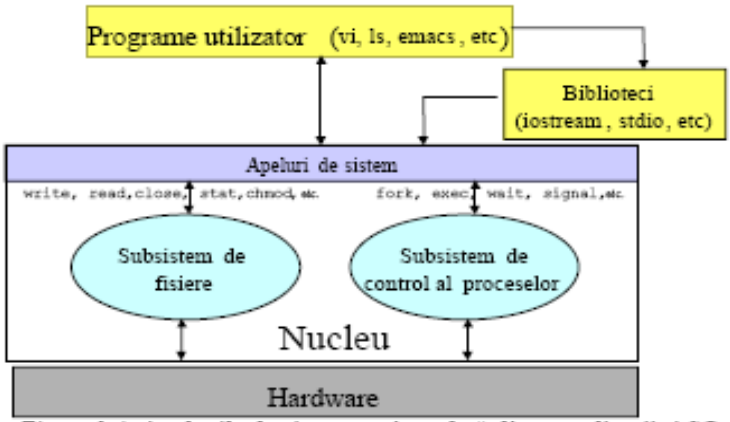


Figura 1.4. **Apelurile de sistem - o interfață dintre aplicații și SO**

**Funcțiile nucleului unui SO**

- Prin apelurile de sistem, nucleul oferă un suport eficient pentru ***rularea programelor de aplicație pe maşina respectivă***

- ***Permite fişierelor să devină programe care se execută*** (şi care atunci capătă denumirea de **task-uri sau procese**)

-***Funcții auxiliare ale nucleului***: gestiunea resurselor calculatorului(mem. etc.) în aşa fel încât simultan să poată exista mai multe procese distincte

## Funcțiile nucleului unui SO

**Implementarea multitasking-ul** prin "*time sharing*"

**"*ceas de timp real*"** (un periferic) generează periodic întreruperi..... Toate procedurile, care ***servesc la tratarea întreruperilor***, sunt tot parte a nucleului (numele lor englezesc este ***"interrupt handlers"***).

Soseşte întreruperea de ceas; → *programul nr.1* → se sare la procedura de tratare a întreruperii din nucleu → memorează starea acestuia (valoarea Program Counter-ului şi a celorlalți regiştri) → ia din alt loc valoarea pe care o salvase pentru PC-ul *programului nr.2*, şi face un salt acolo.

Rezultatul: - **executarea întrețesută a instrucțiunilor;**

**OBS:-** dacă programele nu accesează nici o zonă de memorie sau fişier sau periferic în comun ***coexistența lor nu se face simțită****;*

- programele nu trebuie scrise într-un fel special pt. a coexista

## Funcțiile nucleului unui SO

- Scrierea programelor nu trebuie sa tina cont de executia intretesuta a lor;
- Pt executie ele dispun de acelasi spatiu adresabil 000000 – FFFFFF
- Cea mai simpla solutie – adresarea aceluiasi spatiu de memorie iar conflictele sa fie rezolvate de sistemul de operare

## Cum rezolvă un SO multitasking **concurența în utilizarea memoriei?**

**Orice sistem de operare multitasking transformă şi accesul la memorie în aşa fel încât procesele să nu se poată influența în moduri nedorite.**

SOLUTIA: Utilizarea unui tip de ***adresare relativa***

ESENTA:

-Toate adresele la care fac referinta programele sunt ***adrese relative***, care sunt adrese virtuale din punct de vedere al memoriei efective.

- Fiecarui program i se aloca un shift in momentul executiei programului pentru a se obtine o ***adresa fizica***.

## Funcțiile nucleului unui SO

- O astfel de gestiune e realizata de un submodul hardware al CPU: ***UMM (Unitate de Management a Memoriei*** )

- Mecanismul este numit **"*memorie virtuală*"**
  - Da impresia fiecarui proces care se executa ca toata memoria este a lui.
  -Solutia este de a **inghesui mai multe procese simultan** in memoria disponibila fizic
  - ***Traduce fiecare acces la adresa X a unuia din procese la o altă adresă***

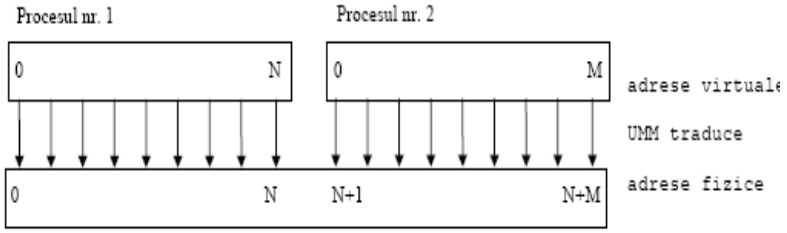



Figura 1.5. Mecanismul memoriei virtuale

## Funcțiile nucleului unui SO

OBS

- pe sistemele **monoprocesor** cele două procese nu se execută niciodată simultan ci pe rând;
- pe un sistem **multiprocesor** - un calculator cu mai multe procesoare - două procese se POT executa simultan şi în esență funcționarea memoriei virtuale este aceeaşi ;

- De fiecare dată când comută de la un proces la altul (deci când tratează întreruperea de la ceas), nucleul trebuie să schimbe şi felul în care UMM **traduce adresele**. Acest lucru îl face ***CPU*** discutând cu ***UMM*** cam în acelaşi fel în care discută cu un **controler**: îi transmite tot felul de **parametri**.

- alta aplicatie a memoriei virtuale: Se pot executa **în time-sharing** procese al căror total de ***memorie necesară*** depăşeşte ***cantitatea existentă***. Această tehnică se numeşte ***"swapping"***.

## Funcțiile nucleului unui SO

*Când un proces este înlocuit cu altul, **părți din memoria ocupată de el sunt mutate pe disc** (de exemplu într-un fişier). Când vine din nou la rând, acele **părți sunt aduse** la loc în memoria fizică (discul, de obicei, este mult mai mare decât memoria). Această tehnică se numeşte **"swapping"**.*

Dezavantaj- citirea/ scrierea pe disc e **mult mai lentă** decât accesarea memoriei.

OBS

- Izolarea proceselor în zone de memorie disjuncte face sarcina programatorului mai uşoară şi calculatorul mai eficient: se pot rula simultan programe ale unor utilizatori diferiți, care nu se vor incomoda unul pe altul nicicum.
- Un program nu poate scrie/citi în/din memoria altuia, pentru ca nici una din adresele pe care el le referă nu este tradusă de UMM în vreo adresă a celuilat.

## Funcțiile nucleului unui SO

- Funcții pentru gestionarea task-urilor;
- Funcții pentru gestionarea timpului sistem;
- Funcții pentru gestionarea memoriei;
- Funcții pentru gestionarea evenimentelor;
- Funcții pentru gestionarea perifericelor I/O;
- Funcții care să asigure suportul pentru comunicații dintre procese etc.

## Funcțiile nucleului unui SO

### Gestionarea task-urilor într-un context multitasking

- Fiecărui task i se ataşează, în contextul timpului real, un indicator de importanță denumit ***prioritate***, ce poate fi :
- **un atribut fix**, **constant** al task-ului ;
- **un atribut** a cărui valoare este **variabilă** şi se stabileşte în momentul activării sale (sau al reactivării).

**OBS**

Există un număr limitat de nivele de priorități (de ex. 4 - 256) şi fiecare task este asociat, static sau dinamic, unuia dintre aceste nivele;

În cadrul fiecărui nivel de prioritate, dacă există mai multe task-uri cu aceeaşi prioritate, ele se vor diferenția între ele prin "**vechimea**" avută (din momentul de timp în care au fost activate), de regulă, după criteriul "***primul venit – primul servit***" (FIFO - First In First Out).

**Prioritățile permit:**

• asigurarea unor anumiți timpi de răspuns;

• utilizarea în comun, de către mai multe task-uri a unor resurse comune (echipamente periferice, zone de memorie, etc.).

- Task-urile ce sunt la un moment dat **activate şi încărcate** în memoria internă **îşi dispută dreptul** de a utiliza **unitatea centrală** (presupunând un sistem monoprocesor).

- Componenta ***planificator*** a oricărui sistem de operare dirijează procesele în execuție aşa încât în intervalul în care un proces este în aşteptare după o resursă / terminarea unei operații de I/O, un alt proces să beneficieze de procesor.

  - astfel, instrucțiunile fiecărui proces se execută "**pe secvențe**" alternând cu secvențe de instrucțiuni de la alte procese;

  - fiecare proces primeşte controlul, procesorul îi execută câteva instrucțiuni maşină, starea lui este salvată şi controlul este transmis unui alt proces ş.a.m.d;

  - mai multe procese înaintează simultan spre finalizarea activității lor.

- Task-urile sunt plasate de către planificatorul SO într-un şir de aşteptare şi sunt executate în ordinea de prioritate a acestora.

- Ori de câte ori SO preia controlul, el întrerupe sau suspendă execuția task-ului în execuție. După ce SO îşi termină acțiunea, decizia acestuia, în ceea ce priveşte acordarea dreptului de a utiliza unitatea centrală poate fi:

  - **de reluare a execuției task-ului întrerupt** ;
  - de **trecere la execuția task-ului cu prioritatea momentan cea mai mare din şirul de aşteptare;**
  - **de trecere la execuția unui task oarecare;**
  - în cazul în care mai **multe task-uri au aceeaşi prioritate**, ele sunt executate conform politicii **primul venit - primul servit**
  - în ordinea **specifică sistemelor cu acces multiplu (*multi-user*)** unde fiecărui task i se acordă pe rând o cuantă limitată de timp (**politica "*time-sharing*").**
  - unele SO pot oferi facilitatea de "**activare în condiții de criză de timp**", prin **creşterea automată a priorității unui task**, dacă acesta nu a fost executat de un anumit timp.

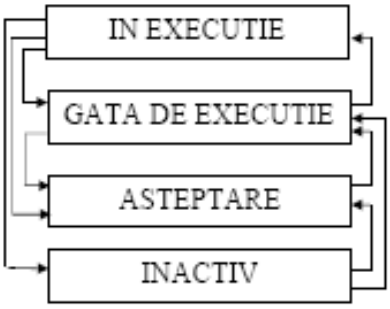
SO examinează şirul de aşteptare al task-urilor "***gata de execuție***" şi decide alocarea CPU imediat după producerea şi rezolvarea următoarelor evenimente :

- un ***eveniment extern*** (o întrerupere dinspre un proces sau un operator uman);
- un ***eveniment intern*** (o întrerupere generată de o operație I/O);
- o ***întrerupere de la dispozitivul*** "***ceas de timp real***";
- ***apelarea din programe*** a unei funcții realizată de SO (***apel de sistem***);
- ***terminarea*** **sau** ***suspendarea*** **execuției task-ului activ**;
- ***trecerea unui anumit interval de timp***;

Pentru a putea relua execuția unui task, la întreruperea lui se memorează (se salvează) întregul context, adică toate informațiile necesare reluării task-ului (conținutul registrelor, starea unui indicator etc.).

**Stările task-urilor**

Un task se poate găsi la un moment dat într-una din următoarele patru stări

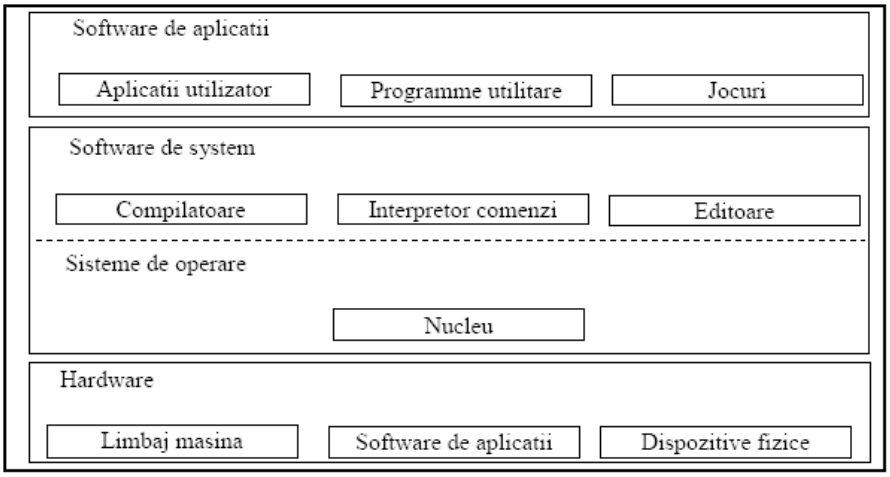



- a) ***în execuție*** *(**RUN**)*: utilizează CPU în acel moment; într-un sistem monoprocesor un singur task se poate găsi la un moment dat în această stare;
- b) ***gata de execuție*** *(**READY**)*: are toate resursele necesare rulării, mai puțin procesorul; aşteaptă, într-un şir de aşteptare pentru a i se aloca CPU spre a fi executat;
- c) ***blocat (WAIT)***: are nevoie şi de alte resurse; aşteaptă să i se aloce memorie, sau terminarea unei operații I/O, producerea unui eveniment extern, trecerea unui anumit interval de timp, etc.

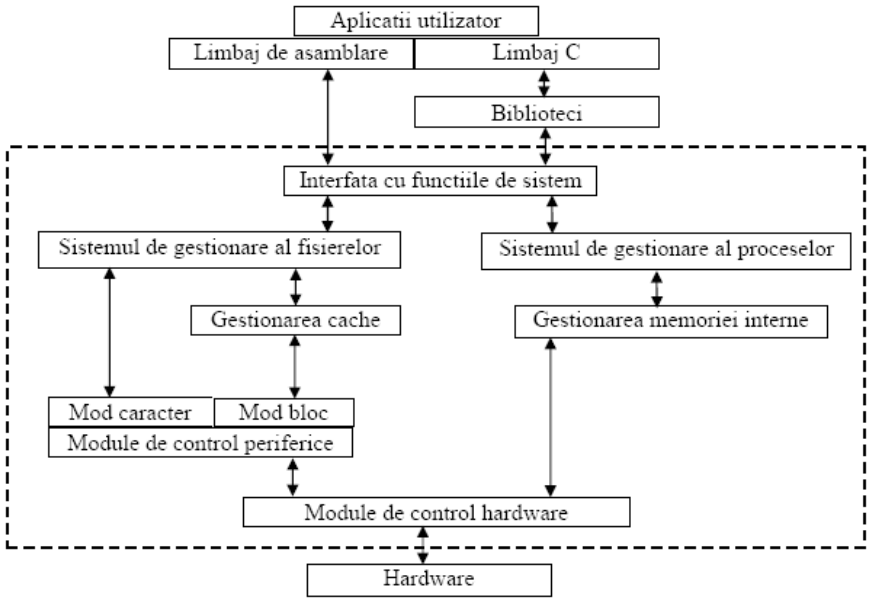
- **Inactiv**: procesului i s-au atribuit de către sistemul de operare doar un identificator (nume), o zonă de cod si o zonă pentru memoria stivă.

## *Functionalitati de baza ale UNIX pentru utilizarea programelor scrise in C*

Structura ierarhizata a unui sistem de calcul si a utilizarii sale poate fi reprezentata ca in figura urmatoare :




In cazul sistemului UNIX aceasta structura poate fi detaliata pentru a obtine :



## Interfața cu funcțiile de sistem

Aceste funcții se mai numesc şi apeluri sistem (system calls)

Sistemul de gestionare a fişierelor

Cuprinde funcții care realizează:

- operații cu fişiere: creare, citire, adăugare, scriere, ştergere
- alocarea/eliberarea spațiului de memorie
- administrarea memoriei neutilizate
- schimbarea structurii sistemului de fişiere prin montarea sau demontarea unui dispozitiv
- asigurarea identificării şi protecției fişierelor

- **Sistemul de gestionare a proceselor**

  Funcțiile acestui modul sunt următoarele:

  - Trecerea proceselor prin diferite stări
  - Planificarea proceselor pentru executarea lor pe procesor
  - Operații asupra proceselor: creare, terminare, trecerea în starea de aşteptare
  - Comunicarea între procese
  - Sincronizarea proceselor

- **Modulul de gestionare a memorie cache**

  Funcțiile acestui modul:

  - alocarea şi eliberarea unui bufffer cache
  - citirea sincronă sau asincronă de pe o memorie externă în cache
  - scrierea pe o memorie externă a conținutului memorie cache

- **Modulul de gestionare a dispozitivelor periferice**

  Funcțiile acestui modul:

  - accesul proceselor la dispozitive eterne
  - asigurarea selecției driver-ului corespunzător perifericului
  - planificarea operațiilor de intrare-ieşire

  Periferice caracter: tastatura, ecranul

  Periferice bloc:memoria pe disc, memoria internă.

- **Modulul de gestionare a memorie interne**

  – Rolul:
    - Ținerea evidenței regiunilor de memorie alocate fiecărui proces şi a celor libere
    - Alocarea şi eliberarea memoriei necesare unui proces în conformitatea cu tehnicile de paginare

## Funcțiile nucleului unui SO

**Ce înțelegem prin tratarea evenimentelor?**

**- evenimente externe**
**- întreruperi** **tratare ≈ similară**
**- evenimente interne**

***Controlul este preluat de SO care va identifica mai întâi evenimentul, după care, ca răspuns la acest eveniment, are loc una sau alta sau ambele din următoarele acțiuni:***

- **se execută o rutină de tratare a întreruperii, fără a se afecta planificarea execuției taskurilor;**
- **se activează un task prin plasarea lui într-un şir de aşteptare, conform priorității sale (pentru a i se aloca CPU pentru execuție) sau, în mod excepțional, se poate trece direct la executarea task-ului.**

## Funcțiile nucleului unui SO

- rutinele specifice de tratare a evenimentelor pot fi asociate diferitelor tipuri de întreruperi cu ajutorul unor ***"tabele de asociere"*** **(tabela vectorilor de întrerupere);**

- eventual pentru a **asigura timpii reduşi de tratare a întreruperilor**, pot fi şi microprogramate;

- în majoritatea SO reale, **rutinele de tratare a întreruperilor au prioritate maximă;**

- evenimentele externe sunt semnalate sistemului de calcul prin intermediul unor dispozitive fizice speciale cum sunt ***"liniile de întrerupere", "indicatorii de evenimente"*** sau ***"cuvintele de stare de întrerupere".***

## *Cum facilitează un SO comunicarea dintre programe ?*

***Cerințe impuse:***

***Programele şi, respectiv, task-urile ce alcătuiesc ansamblul de programe al unui sistem de calcul de tip multitasking comunică destul de intens între ele, atât pentru a utiliza date comune cât şi pentru a-şi sincroniza în timp desfăşurarea execuției lor***;

***Soluții:***

1. ***Datele pot fi utilizate în comun cu mai multe task-uri, prin memorarea lor în zone de memorie internă declarate ca fiind comune mai multor task-uri şi la care toate task-urile vor avea acces (memorie partajată);***
2. ***Sistemul de lucru cu fişiere care permite generarea şi utilizarea unor colecții comune de date cu volume sensibil mai mari, la care diferitele task-uri pot avea acces într-un mod controlabil (fişiere partajate)*** ;
3. ***Facilitatea oferită de unele SO de a se transmite între task-uri anumite mesaje care constau dintr-un volum limitat de date.***
4. ***Utilizarea în comun a unor subrutine, în special pentru operațiile cu I/O şi de comunicare cu module ale SO (planificator etc.).***

OBS

**Metoda 3 (mesaje)**

Pentru un număr limitat de task-uri ce comunică între ele printr-un asemenea mecanism, acesta poate fi foarte avantajos, asigurând timpi reduşi de răspuns şi folosind pentru executarea intercomunicării doar locații din memoria internă.

Prin acest mecanism de transmitere de mesaje între task-uri se poate realiza şi **sincronizarea** între task-uri paralele ceea ce este o facilitate foarte utilă pentru comunicarea între task-uri.

**Metoda 4 (subrutine)**

În cazul SO multitasking, în timpul execuției unei rutine comune mai multor task-uri, pot apare întreruperi, re-entranța acestor rutine trebuind să fie asigurată de către SO.

Există mai multe metode de a trata această comunicare dintre task-uri, cum ar fi :

- blocarea altor task-uri ce ar putea apela subrutina în curs de execuție;
- subrutina să fie re-entrantă;
- duplicarea subrutinei (câte o copie a subrutinei pentru fiecare task care o foloseşte);
- inhibarea întreruperilor pe durata execuției unei subrutine utilizabile în comun de mai multe task-uri.

## 2. Concepte de programare concurentă

***Premize:***

***programele concurente se caract. prin*** execuție ***"întrețesută "***

***programele concurente nu sunt programe absolut independente între ele***

> ***Task-urile sunt porțiuni ale unui program ansamblu care urmăreşte realizarea unei anumite prelucrări şi care a fost divizat în părți (denumite task-uri) care pot fi executate în paralel, concurând la a obține în acest scop resursa CPU dar şi colaborând între ele.***

***Mecanisme de descriere şi manilupare a proceselor concurente***

*Concurență = Paralelism + Interacțiune*

> *Îmbinarea celor două aspecte este esențială, deoarece* ***paralelism fără interacțiune reprezintă un caz particular al multi-tasking-ului iar interacțiune fără paralelism reprezintă cazul particular simplist al lanțului de mai multe programe secvențiale.***

### 2.1. Concepte abstracte utilizate în descrierea concurenței

#### *2.1.1. Paradigme de programare nesecvențială*

> *In general,* ***execuția unui program secvențial*** *este* ***deterministă****. Astfel, pentru acelaşi set de date de intrare, programul execută* ***aceeaşi secvență de instrucțiuni şi produce aceleaşi rezultate.***

***Paradigma programării secvențiale are două caracteristici de bază:***

- ***ordinea textuală a instrucțiunilor furnizează ordinea de execuție a acestora.***
- ***instrucțiuni succesive se vor executa fără a se suprapune, în timp, una cu cealaltă.***

***Nici una dintre aceste două proprietăți de bază nu sunt valabile în cazul programării nesecvențiale.***

***Programarea nesecvențială presupune*** :

> ***Implementarea mai multor entități de program executate în acelaşi timp cărora li se asociază task-uri distincte ce furnizează părți ale rezultatului final.***
>
> Grad de ***nedeterminism*** în timpul execuției.

## *Paradigme de programare nesecvențială*

Paradigmele de ***programare nesecvențială diferă între ele prin***

- tipul sistemului de calcul pe care se execută entitățile concurente: sistem monoprocesor, multiprocesor, sistem distribuit
- modul cum sunt partajate de către entități informațiile de context şi resursele sistemului
- gradul de interacțiune între entități, funcție de existența unei "autorități tutelare" care să asigure coordonarea entităților

Paradigmele de ***programare nesecvențială*** pot fi grupate în trei mari categorii:

- paradigme de programare paralelă;
- paradigme de programare concurentă;
- paradigme de programare distribuită.

Paradigmele de *programare nesecvențială*

***a) Programare paralelă- caracteristici ale programelor paralele:***

- ***Un program paralel constă din unul sau mai multe sarcini de lucru (task-uri). Aceste task-uri se execută simultan şi numărul lor poate varia în timpul execuției programului.***
- ***Fiecare task încapsulează un program secvențial şi o memorie locală. De fapt, un task reprezintă o maşină virtuală von Neumann. (O maşină von Neumann conține o unitate centrală de procesare – CPU - conectată la o unitate de memorare şi execută operații de citire şi scriere asupra unor date din memoria ataşată.)***
- ***Task-urile pot fi ataşate procesoarelor fizice în diverse moduri, dar această ataşare nu afectează semantica programului. In particular, unui singur procesor îi pot fi ataşate mai multe taskuri.***
- ***Se pot defini interfețe de comunicare între aceste task-uri şi, de asemenea, modalități de acces la resursele comune, în conformitate cu modelul de paralelism folosit.***

- **Modelul de paralelism bazat pe *task-uri* şi *canale***
- ***Modelul bazat pe schimb de mesaje* (message-passing)**
- ***Modelul de paralelism al datelor***
- **Modelul *memoriei partajate***

Modele de paralelism, diferențiate după caracteristicile taskurilor

- **Modelul de paralelism bazat pe *task-uri* şi *canale***

Caracteristici

- Fiecare task are asociat un set de porturi de intrare (***inports***) şi un set de porturi de ieşire (***outports***), conectate între ele prin **cozi de mesaje numite *canale***. Aceste canale pot fi ***create*** sau ***şterse dinamic***.

- Un task poate executa operațiile:
  - creare / ştergere canale;
  - citiri de la porturile de intrare / scrieri la porturile de ieşire;
  - creare / ştergere de alte taskuri.

Imaginea globală a unui proces de calcul ce constă dintr-un set taskuri

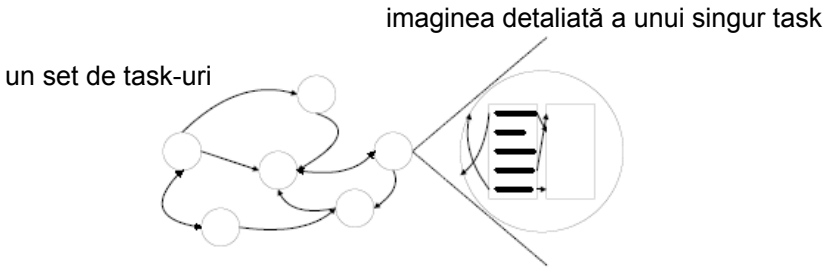



Figura 2.1. Modelul taskuri şi canale

In cadrul unui task s-au evidențiat:

- setul lui de instrucțiuni (primul dreptunghi)
- unitatea de memorie locală (celălalt dreptunghi)
- interfața externă este asigurată printr-un set de porturi (ilustrate prin săgețile ce pleacă de la instrucțiuni spre exterior)
- task-urile sunt conectate prin canale, reprezentate prin săgeți
- un canal reprezintă o coadă de mesaje în care sunt plasate, respectiv din care sunt extrase mesaje

## Modele de paralelism, diferențiate după caracteristicile taskurilor

### *Modelul bazat pe schimb de mesaje* (*message-passing*)

Caracteristici

- Creează task-uri multiple;
- Fiecare task este identificat printr-un ***nume*** şi încapsulează ***date locale***;
- Task-urile interacționează trimițând şi primind ***mesaje*** la, respectiv de la task-uri specificate prin nume.

***Cele două modele diferă numai prin mecanismul folosit pentru transferul datelor. La primul model se vorbeşte de "trimis mesaje pe canalul x", iar la al doilea de "trimis mesaje task-ului n".***

### *Modelul de paralelism al datelor*

Exploatează ***concurența*** care derivă din aplicarea aceleiaşi operații mai multor elemente dintr-o structură de date. De exemplu, “adaugă 2 la elementele unui tablou” sau “măreşte salariul tuturor angajaților cu x lei”. Programatorul are sarcina de a preciza cum sunt partiționate datele în task-uri.

**Modele de paralelism diferențiate după caracteristicile taskurilor**

**Modelul *memoriei partajate***

- In acest model, task-urile ***partajează un spațiu comun de adrese***, în care scriu şi citesc date.
- Pentru controlul accesului la memoria partajată se folosesc mecanisme adecvate, aşa cum vom vedea în secțiunile următoare.
- Un avantaj al acestui model constă în faptul că nu se specifică explicit cine este "*producătorul*" şi cine sunt "*consumatorii*" unei date.

***Proprietăți de bază ale programelor paralele:***

***concurență*, *scalabilitate* şi m*odularitate***

***Concurența*** se referă la posibilitatea de a executa mai multe acțiuni simultan. Ea este o proprietate esențială, mai ales dacă programul se execută pe un sistem multiprocesor.

***Scalabilitatea*** presupune funcționarea programului, cel puțin la aceiaşi parametri de performanță, dacă numărul de procesoare creşte.

***Modularitatea*** implică descompunerea unor entități de execuție complexe în componente mai simple. Evident, această proprietate nu este specifică numai programării paralele, ci şi altor paradigme de programare şi proiectare.

## b) *Programare concurentă*

- ***task-urile cooperează între ele în timpul execuției****;*
- problema este descompusă în subprobleme relativ independente;
- execuția acestor task-uri poate să fie intercalată şi urmează un ***scenariu de paralelism la nivel logic***;
- Două task-uri active pot face schimb de informații, pot să aştepte unul după altul etc;

OBS - ***mecansime specifice*** pentru comunicarea şi sincronizarea elementelor de execuție implicate în programul concurent;

- problema tipica de programare concurentă este ***problema producă-torului şi a consumatorului***.

In fiecare moment, asupra recipientului acționează ***numai un singur proces***, de orice tip ar fi el (acces exclusiv la recipient).

Dacă ***recipientul este plin***, ***producătorii*** care vor să depună în el ***rămân în aşteptare*** până când un consumator extrage un obiect.

Dacă ***recipientul este gol***, ***consumatorii*** care vor să extragă ***rămân în aşteptare*** până când un producător depune un obiect.

## Procesor logic

Atât concurența cât şi paralelismul necesită ***acces controlat la resursele partajate*** *(dispozitive de I/O, fişiere, înregistrări din baze de date, structuri de date globale, etc).*

*Conceptul de* ***paralelism*** *este implicit mai apropiat de* ***hardware.***

*Conceptul de* ***concurență*** *este mai apropiat de* ***software,*** *fiind* ***o proprietate stabilită în principal de proiectantul programului.***

Pentru a concepe modele de programare concurentă independente de hardware, se asociază fiecărei entități executabile *din program un* ***procesor logic.***

**DEFINIȚIE ŞI CARACTERISTICI**

- Fiecare procesor logic este o ***maşină secvențială*** care execută pe rând instrucțiunile din procesul alocat.
- Mai multe ***procesoare logice*** corespund unui ***procesor fizic***.
- Acesta are implementat un ***mecanism de planificare*** pentru a da controlul procesoarelor logice aflate în gestiunea sa.

## Procesor logic

- Nu se fac ipoteze cu privire la ***vitezele relative*** ale operațiilor corespunzătoare procesoarelor logice.
- Din acest punct de vedere, ***paralelismul*** poate fi considerat un caz particular al concurenței, pentru sisteme multiprocesor, în care fiecărui procesor logic îi corespunde un procesor fizic.
- De obicei, ***sistemele de operare implementează concurența***.

**CONCLUZII**

- Din cele expuse rezultă că ***gestiunea entităților concurente*** presupune, pe lângă ***creare*** şi ***terminare***, operații de ***sincronizare****,* adică de **coordonare a task-urilor** care nu sunt complet independente, de ***comunicare****,* schimb de informații între taskuri şi de ***planificare****,* adică stabilirea priorităților taskurilor de executat.
- Aceste operații, precum şi ***mecanismele specifice***, ***formale sau implementate pe unele platforme***, sunt subiectele principale ale prezentei abordări.

## Avantajele paradigmei de programare concurentă

- Controlul unor activități multiple, relativ independente şi gestiunea unor evenimente **asincrone, externe**. Programele sunt adesea scrise pentru a simula sau răspunde la evenimente din lumea reală, iar în lumea reală, concurența este un lucru obişnuit. Modelarea unui astfel de comportament este posibilă dacă mediul de programare suportă noțiunea de concurență.
- Creşterea eficienței programelor consumatoare de timp. Operațiile de I/O, sau de aşteptare (de exemplu, *sleep*) vor bloca numai procesul/thread-ul apelant până la terminarea operației respective.
- O reacție mai rapidă a calculatorului la acțiunile utilizatorului. Pentru aceste cereri se vor crea procese sau thread-uri cu prioritate mai mare.
- Îmbunătățirea performanței. Dacă sunt disponibile procesoare multiple, care lucrează în paralel, execuția programului este mult mai rapidă. Această situație poartă numele de **concurență reală.**
- Pe sistemele uniprocesor este posibilă îmbunătățirea performanței prin **modelare concurentă**, dacă programul efectuează operații consumatoare de timp. Cât timp o activitate aşteaptă terminarea unei operații de I/O, procesorul poate executa alte sarcini de calcul. În acest caz, este vorba de o **concurență logică**.

### *c) Programarea distribuită*

DEFINIȚIE CARACTERISTICI

- ***Programarea distribuită implică procesarea logică, simultană, la distanță, a unor task-uri pe platforme eterogene, plasate în diferite puncte din rețea.***

- ***Contextele proceselor distribuite sunt strict separate***. Procesele nu partajează între ele părți ale mediului de execuție.

- Două procese active pot schimba informații numai prin ***transfer de mesaje***.

-Spre deosebire de programarea concurentă, ***în contexte distribuite nu există o autoritate centrală de coordonare a proceselor***, şi nici nu se gestionează stări globale ale proceselor.

-Pot să apară ***probleme*** de:

- gestiunea întârzierilor de comunicare;
- administrare a rețelelor (securizate mai mult sau mai puțin);
- administrare a calculatoarelor din rețea în momentul când acestea nu mai funcționează la parametrii normali.

Probleme care implică o abordare distribuită

**Metodele şi tehnicile specifice programării distribuite au astăzi un spectru extrem de larg**. În prezent, sub termenul de ***middleware*** sunt cunoscute tehnologiile de **nivel 6** din **modelul OSI**.

În prezenta lucrare vom aborda doar *tangențial*, în ultimul capitol, elemente de programare distribuită.

-Serviciile Internet

- Aplicațiile client/server în care programul server şi aplicațiile client se găsesc pe maşini diferite dar interconectate

- Soluții client/server ale bazelor de date aflate la distanță, agenți mobili, etc.

**modelul de aplicații distribuite client/server:**

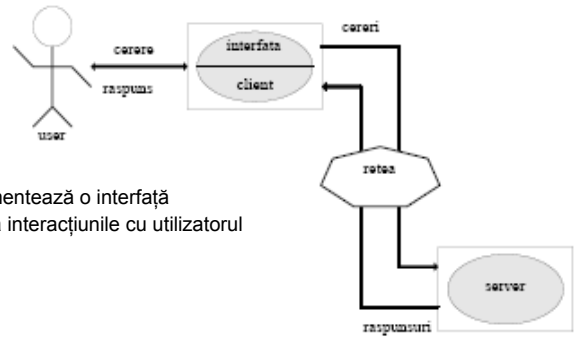
***program client-*** se execută pe ***maşina locală***, comunică printr-o ***conexiune rețea*** cu:

***program server-*** se execută pe ***maşina la distanță***



Utilizatorul vede din întreaga aplicație doar programul client, care acceptă intrări şi produce rezultate

aplicația client trimite cererile utilizatorului la server şi prezintă utilizatorului răspunsurile primite de la server




Programul client implementează o interfață utilizator şi gestionează interacțiunile cu utilizatorul

Figura 2.2. **Client / server în context distribuit**

- O clasă importantă de aplicații distribuite o reprezintă programele care au la bază arhitecturi precum CORBA (Common Object Request Broker Architecture), standardizată de OMG (Object Management Group) sau DCOM (Distributed Component Object Model) dezvoltată de Microsoft.
- Aceste arhitecturi de tip middleware oferă suportul tehnic necesar pentru ca aplicațiile distribuite să coopereze indiferent de protocolul de comunicație sau de arhitectura sistemului gazdă.
- CORBA reprezintă o arhitectură multiplatformă care permite crearea de aplicații orientate-obiect, distribuite, – obiecte diferite se pot găsi pe maşini diferite din rețea- şi eterogene –diferite tehnologii de rețea, platforme hardware sau sisteme de operare; implementarea obiectelor poate folosi limbaje de programare diferite.
- Tehnologia Microsoft OLE/COM oferă un model de gestiune a unor componente integrabile în interfețe utilizator. Arhitectura DCOM (Distributed COM) extinde tehnologia COM, pentru a oferi suport pentru comunicarea obiectelor aflate pe calculatoare diferite, dintr-o rețea LAN, WAN sau chiar din Internet.

## Avantajele programării distribuite

- Serviciile şi datele unei aplicații distribuite sunt adesea duplicate, astfel încât aplicația să-şi poată continua execuția în siguranță, chiar dacă o parte din sistem nu funcționează. Dacă datele şi serviciile sunt duplicate în totalitate, utilizatorii nu vor detecta eventualele disfuncționalități din sistem. În cazul unei duplicări parțiale, sistemul va continua să funcționeze, oferind un număr restrâns de servicii.

- Un sistem care include numeroase organizații diferite este mai uşor de administrat ca o entitate distribuită. Fiecare organizatie îşi poate structura, gestiona şi controla partea sa din sistem, conform propriilor legi, reguli şi preferințe. Avantajul administrativ al unui sistem distribuit este evident în sistemul World-Wide Web, unde fiecare site este gestionat separat.

- Performanța mărită a unui sistem distribuit se obține datorită posibilității de a executa mai multe programe, în acelaşi timp, pe maşini diferite. O problemă computațională complexă poate fi divizată în sub-probleme (mai mult sau mai puțin independente), care se pot executa pe maşini diferite, iar rezultatele vor fi combinate, pe una dintre maşini, pentru a obține un răspuns final.

## Avantajele programării distribuite

- O arhitectură distribuită permite colaborarea la distanță a unor persoane implicate în rezolvarea unei anumite probleme. Operând pe maşini diferite, aceste persoane pot observa şi manipula informații partajate (programe, date, documente, etc).

### *2.1.2. Interacțiunea task-urilor*

Un ***proces*** sau ***task*****,** este un ***calcul*** care poate fi executat concurent (în paralel) cu alte calcule. El este o abstractizare a activității procesorului, fiind considerat ca un ***program în execuție***. Existența unui proces este condiționată de existența a trei factori:

• **o *procedură*** - o *succesiune de instrucțiuni* dintr-un *set predefinit de instrucțiuni*, cu rolul de descriere a unui calcul - descrierea unui algoritm.

• **un *procesor*** - dispozitiv hardware/software ce recunoaşte şi poate executa setul predefinit de instrucțiuni, şi care este folosit, în acest caz, pentru a executa succesiunea de instrucțiuni specificată în procedură;

• **un *mediu*** - constituit din resursele sistemului: o parte din memoria internă, un spațiu disc destinat unor fişiere, periferice magnetice, echipamente audio-video etc. - asupra căruia acționează procesorul în conformitate cu secvența de instrucțiuni din procedură.

**Deosebirea dintre proces şi program**

***Procesul*** *are un* ***caracter dinamic***, el precizează o ***secvență de activități*** în curs de execuție, iar ***programul*** *are un* ***caracter static***, el numai descrie textual această secvență de activități.

### *Interacțiunea task-urilor*

Evoluția în *paralel a două procese* trebuie înțeleasă astfel:

Pi şi Pj sunt executate concurent dacă:

*max (Ii, Ij) <= min (Hi, Hj)*

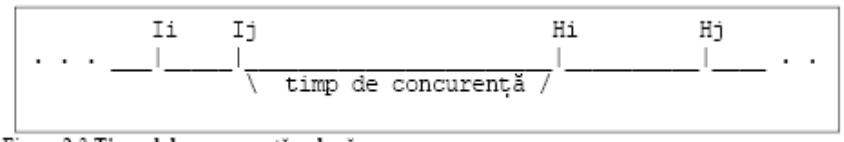



Figura 2.3 **Timpul de concurență a două procese**

Dacă sistemul dispune de două procesoare, atunci este posibil un paralelism efectiv, în sensul că atât Pi cât şi Pj sunt executate simultan de câte un procesor. Dacă există un singur procesor, atunci acesta execută alternativ grupuri de instrucțiuni din cele două procese.
Decizia de comutare aparține sistemului de operare.

## *Interacțiunea task-urilor*

Task-ul, elementul de bază al programării concurente se bucură de următoarele proprietăți de bază :

• ***indivizibilitate*** - adică este o entitate atomică (din punct de vedere al concurenței) ce nu poate fi divizată în alte unități, cărora SO să le aloce resurse în mod autonom;

• ***secvențialitate*** - în sensul că în interiorul său nu există paralelism, execuția fiind secvențială;

• ***asincronism*** - în sensul că (task-urile) acțiunile se desfăşoară în paralel între ele, relativ independent (exceptând momentele de interacțiune care presupun o sincronizare între ele);

• ***temporalitate*** - în sensul că task-ul se manifestă şi este recunoscut ca atare numai în intervalul de timp dintre lansarea sa în execuție şi terminarea sa.

Interacțiunea între task-uri concrete se manifestă sub trei forme principale, care vor fi examinate în capitolele ce urmează, şi anume:

- excludere reciprocă
- sincronizare
- intercomunicare

### 2.2. Situații de excepție generate de concurență

**Executia simultana a mai multor procese care acceseaza pe durata vietii lor o serie de resurse comune poate genera situatii ciudate de comportament in executia programelor.**

### 2.2.1 Rezultate inconsistente (Race condition)

**Generata de lipsa unui mecanism de sincronizare a task-urilor**

***Să presupunem că există două procese P1 şi P2 care au dreptul să modifice o aceeaşi variabilă v sub forma:***

***P1: v = v+1 şi P2 :v = v+1***

***pentru a ilustra actiunile repetate ale celor 2 procese le descriem in specificarea PARBEGIN- PAREND***

```
PARBEGIN
   P1: . . . v = v + 1; . . .
   |
   P2: . . . v = v + 1; . . .
PAREND
```

***Fiecare dintre cele două procese execută această incrementare concurent şi de un număr neprecizat de ori.***

***Dacă cele două procese interferează în execuția acestei instrucțiuni, rezultatele nu vor fi cele aşteptate.***

Pentru execuția atribuirii *v*=*v*+1 sunt necesare trei instrucțiuni maşină.

```
r=v ;
r=r+1;
v=r;
```

```
P1: r1=v;     P2: . . .
    r1=r1+1;      . . .
    v=r1;         . . .
    . . .          r2=v;
    . . .          r2=r2+1;
    . . .          v=r2;
```

Prima secvență de dublă incrementare

```
P1: r1=v;     P2:. . .
                  r2=v
    r1=r1+1;
                  r2=r2+1;
    v=r1;
     . . .        v=r2;
```

A doua secvență de dublă incrementare

## Situații de excepție generate de concurență

Pentru evitarea acestor **rezultate inconsistente**, este necesară includerea unor mecanisme de sincronizare, astfel încât secvențele de cod corespunzător atribuirilor *v*=*v*+1 să nu fie executate întrețesut.

Există multe prelucrări în care apar acest gen de situații. De exemplu, un task citeşte valoarea unei date dintr-o locație, în timp ce alt task atribuie o altă valoare pentru această dată.

Un alt exemplu: două procese accesează simultan acelaşi articol al aceluiaşi fişier de pe disc. Fiecare dintre ele citeşte conținutul articolului, îl prelucrează şi depune conținutul înapoi. Rezultatul final este dat de ultima scriere a articolului. Normal, acest rezultat depinde de succesiunile celor două citiri şi scrieri.

## Situații de excepție generate de concurență

### *2.2.2 Live-lock*

*Sub denumirea de î**nfometare** (starvation, live-lock) este precizată situația în care mai **multe procese aşteaptă să obțină o resursă critică**, dar accesul la ea **NU** este oferit într-o manieră echitabilă.*

*Spunem că se află în starea starvation acele procese care aşteaptă relativ mult, în raport cu altele care chiar se pot termina de executat, sau procesele acelea care aşteaptă practic un timp indefinit după resursa respectivă.*

Exemplu: va fi selectat să acceseze discul procesul care solicită acces la sectorul cel mai apropiat de precedentul sector citit. La un moment dat este accesat sectorul 1000.

Un proces cere acces la sectorul 2000, în timp ce alte două procese solicită, în mod repetat, acces la sectoarele 999 şi 1001.

Concluzie

Infometarea este o situație de nedorit în programarea concurentă. Evitarea ei se poate face, spre exemplu, dacă din când în când se schimbă, într-o manieră echitabilă, regulile de acces la resursă.

## Situații de excepție generate de concurență

### *2.2.3. Impas*

***Să presupunem că într-un sistem concurent se execută două procese P1, P2 care au nevoie, în diverse stadii ale execuției de aceleaşi două resurse nepartajabile R1, R2. Procesele ocupă resursele în diverse stadii ale execuției lor şi le eliberează la terminare.***

*Să ne imaginăm următorul* ***scenariu de evoluție în timp*** *a celor două procese:*

- ***0. Procesele P1 şi P2 sunt lansate în execuție, iar resursele R1 şi R2 sunt ambele libere.***
- ***1. Procesul P1 solicită resursa R1 şi o ocupă, ea fiind liberă.***
- ***2. Procesul P2 solicită resursa R2 şi o ocupă, ea fiind liberă.***
- ***3. Procesul P1 solicită resursa R2 şi intră în starea de aşteptare, deoarece R2 este deja ocupată de către procesul P2.***
- ***4. Procesul P2 solicită resursa R1 şi intră în starea de aşteptare, deoarece R1 este deja ocupată de către procesul P1.***

***Din acest moment, ambele procese se află în starea de aşteptare, din care teoretic nu vor mai putea ieşi niciodată!***

***Acest fenomen este cunoscut în literatură sub mai multe denumiri: impas, interblocare, deadlock, deadly embrace***, etc.

Impasul este o stare gravă care conduce la eşecul execuției întregii aplicații.

## Situații de excepție generate de concurență

In 1971, Coffman, Elphic şi Shoshani au indicat ***patru condiții necesare pentru apariția impasului:***

-procesele solicită **controlul exclusiv** asupra resurselor pe care le cer (***condiția de excludere mutuală***);

-procesele ***păstrează resursele deja ocupate atunci când aşteaptă alocarea altor resurse*** (condiția de ***wait for*****);**

-***resursele nu pot fi şterse*** din procesele care le țin ocupate, până când ele nu sunt utilizate complet (condiția de ***nepreempție***);

-există un lanț de procese în care fiecare dintre ele aşteaptă după o resursă ocupată de altul din lanț (condiția de ***aşteptare circulară***);

***Practica reliefează câteva abordări posibile***:

**1.** prima abordare, a cărei utilizare nu este prea recomandată, constă în ***ignorarea impasului***, în speranta că acesta nu se va produce. Si dacă totuşi apare sistemul este oprit în mod forțat.

**2.** a doua abordare permite producerea impasului, însă ***detectează*** **apariția lui**. Procesele sunt **terminate selectiv** sau sunt readuse la o **stare anterioară** şi **suspendate temporar** până când "pericolul" a trecut. Această soluție este parțial acceptabilă; în schimb NU este potrivită, spre exemplu, pentru sistemele în timp real.

**3.** A treia abordare constă în ***prevenirea*** impasului prin modificarea unor condiții care pot conduce la impas.

Prevenirea impasului

***1)*** să se impună ca fiecare proces ***să ocupe din start toate resursele care-i sunt necesare***, indiferent de momentele la care le utilizează efectiv.
Dezavantaj: o utilizare nejudicioasă a resurselor.

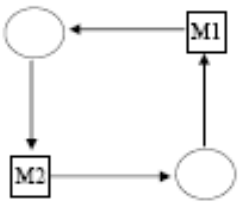
**2)** stabilirea de către sistemul de operare a unei ***ordini de solicitare a resurselor*, (R1, R2, ... Rk** ), la care trebuie să se alinieze toate procesele.
Dezavantaj: Soluția este funcțională, dar are în ea o mare doză de artificial.

**3)** ***sistemul să controleze, înainte de fiecare alocare, dacă nu cumva este posibilă apariția impasului*.** Se construieşte un ***graf de alocare*** *a* **resurselor**. Acesta este un **graf orientat** (**X**, **U**), având ca **noduri procesele şi resursele, iar arcele reprezentind atribuiri(realizate sau posibil a fi realizate) sunt numai între procese şi resurse, astfel:**

- **X** = {P1, P2, ..., Pn, R1, R2, ..., Rm}
- Există un arc (Rj, Pi) în **U** dacă procesul Pi a ocupat resursa Rj.
- Există un arc (Pi, Rj) în **U** dacă procesul Pi aşteaptă să ocupe resursa Rj.

  În acest graf, dacă există un ciclu, atunci este posibil să apară impasul.

  Şi ca atare daca in urma alocarii unei resurse se ajunge la un graf ciclic, alocarea e anulată şi procesul solicitant e pus în aşteptare pina la eliberare resursei cind se incearca din nou alocarea.



Datorita dificultatii realizarii prevenirii situatiei de impas cele mai multe sisteme multitasking se gasesc in prima din cele 3 situatii adica ignora pur si simplu posibilitatea aparitiei impasului.

Exista sisteme evoluate care au implementat functii de prevenire sau de evitare a impasului. Este cazul special al unor sisteme de gestiune a bazelor de date( rezervarea biletelor).

In **UNIX**, functia de blocare a fisierelor **lockf** realizeaza deasemenea detectia eventualelor situatii de impas:

- de fiecare data cind un proces trebuie sa blocheze accesul la un fisier, sistemul de operare verifica daca nu este posibila aparitia unui impas.

- daca un impas este posibil, sistemul de operare refuza blocarea accesului la fisierul respectiv