Controlul si descrierea proceselor

Introducere

Crearea şi terminarea proceselor

Stările unui proces

Sincronizarea proceselor

Semafoare

Utilizarea semafoarelor în gestionarea resurselor critice

Probleme clasice de coordonare a proceselor

Fire de execuţie

Introducere

- Procesul este abstractizarea unui program în execuţie şi reprezintă elementul computaţional de bază în sistemele de calcul moderne. Toate sistemele de operare moderne au fost construite în jurul conceptului de proces. Multe probleme pe care sist. de op. trebuie sa le rezolve, sunt legate de procese.
- Termenul de proces, poate fi defini prin una din următoarele modalităţi:

  - Un program în executie.

  - O instanţă a unui program ce se execută pe un calculator.

  - O entitate ce poate fi asignată şi executată de un processor.

  - O unitate a unei activităţi caracterizată de execuţia unei secvenţe de instructiuni, o stare curentă şi o mulţime de resurse ale sistemului asociată.
- Elementele esentiale ale unui proces sunt:

  - codul programului (care poate fi partajat cu un alt procese care execută acelaşi program);

  - un **set de date** asociat acelui cod.
- Administrarea proceselor se referă la toate serviciile oferite de sistemul de operare pentru gestionarea activităţii tuturor proceselor care sunt executate la un moment dat. Pentru a realiza această activitate, sistemul de operare administrează mai mute structuri de date care vor fi detaliate în cursurile urmatoare.

Crearea şi terminarea proceselor

- Următoarele evenimente conduc la crearea unui nou proces:

  - primirea spre execuţie a unui nou job (în cazul sistemelor cu prelucrare în loturi);

  - loginarea unui nou utilizator (în cazul sistemelor interactive);

  - crearea procesului de către sistemul de operare pentru a oferi un serviciu unui proces existent în sistem.

  - Declararea în cadrul unui proces (proces tată) a unui nou proces (proces fiu), care se vor executa în paralel.
- Când un nou proces este adăugat celor în curs de execuţie sistemul de operare construieşte o structură de date necesară administrării lui. Această structură de date se numeşte **blocul de control al procesului**(PCB – **P**rocess **C**ontrol **Bl**ock); este este o zonă de memorie, ce conţine următoarele informţii:

  - **identificatorul procesului**, prin care se face distincţia faţă de alte procese din sistem.

  - **nivelul de prioritate** al procesului, în raport cu alte procese din sistem.

  - starea procesului, ale cărei valori le vom prezenta imediat.

- valoarea **contorului de program**, care indică adresa următoarei instrucţiuni care urmează să fie executată pentru acest proces.
- valorile **regiştrilor unităţii centrale**; când apare o întrerupere şi CPU trece la execuţia altui proces, valorile acestor regiştri şi ale contorului de program trebuie salvate, pentru a permite reluarea corectă a execuţiei procesului; când procesul primeşte din nou serviciile CPU, valorile salvate în PCB-ul procesului respectiv sunt atribuite regiştrilor acesteia.
- informaţii de **contabilizare**, referitoare la valoarea timpului CPU utilizat de proces, valori limită ale timpului CPU (între care este executat procesul respectiv), conturi utilizator, numere ale altor procese etc;
- informaţii legate de operaţiile de I/O (cereri de operaţii de I/O în curs de execuţie, perifericele asignate procesului, lista fişierelor deschise de proces etc.)
- informaţii de **planificare** a CPU(prioritatea procesului, pointer către coada de planificare a execuţiei procesului etc.).
- informaţii cu privire la modalitatea de **alocare** a memoriei interne;
- un pointer către un alt PCB.

- **Terminarea execuţiei proceselor** se poate realiza din următoarele motive:

  - **Terminare normală**. Procesul execută o rutină a sist. de operare prin care semnalează că a atins şi executat ultima instr. cod maşină din fişierul executabil.

  - **Depăşirea** intervalului de timp alocat de sistem.

  - **Memorie nedisponibilă**. Procesul cere mai multă memorie decât cea pe care o poate oferi sistemul.

  - **Violarea memoriei**. Procesul încearcă să acceseze o locaţie de memorie la care nu are permis accesul.

  - **Eroare de protectie**. Procesul încearcă să acceseze o resursă într-un mod în care nu-i este permis (de exemplu, scrierea într-un fişier asupra căruia nu are acest drept).

  - **Eroare aritmetică**. Procesul încearcă să execute o operaţie interzisă (de exemplu împărţire la zero) sau un calcul al cărui rezultat depăşeşte valoarea maximă stabilită prin hardware.

-**Depăşirea timpului de aşteptare** pentru producerea unui eveniment.

- **Eşec în efectuarea unei operaţii de I/O** (de exemplu imposibilitatea găsirii unui fişier, nerealizarea operaţiilor de citire sau scriere după un anumit număr de încercări, lansarea unei operaţii invalide-citirea de la imprimantă).

- Instructiune invalidă. Procesul încearcă să execute o instructiune inexistentă.

- Instructiune privilegiată. Procesul încearcă să execute o instructiune rezervată sistemului de operare.

- **Date eronate**. Tipul datelor referenţiate este eronat sau este invocată o variabilă neiniţializată.

- **Intervenţia** unui operator uman sau a sistemului de operare în situaţii în care execuţia procesului duce la blocarea sistemului de calcul.

Stările unui proces

- **Starea** unui proces este definită ca mulţimea activităţilor executate de acesta. În cele ce urmează vom presupune că avem un sistem de calcul cu un singur procesor(CPU). Stările unui proces sunt:

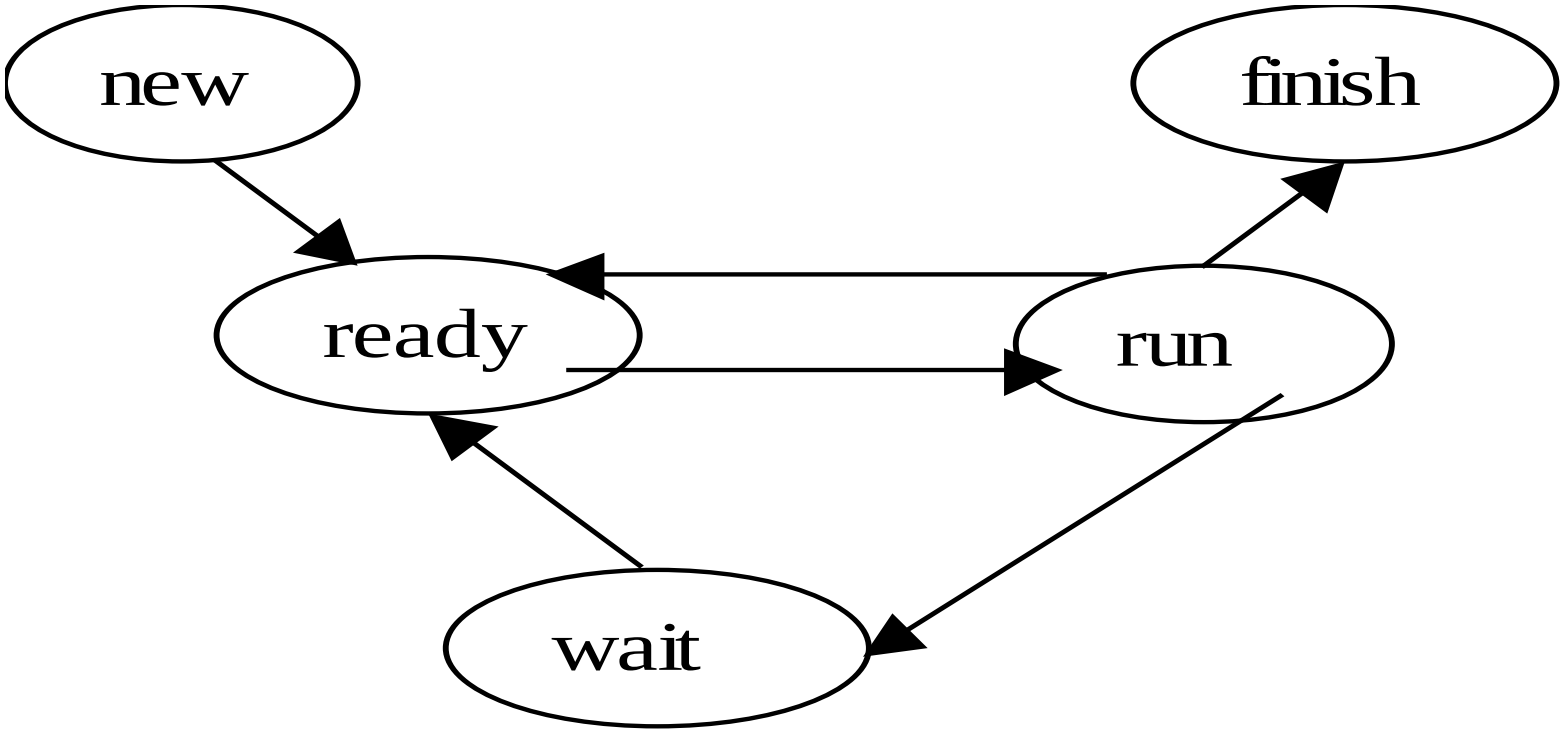
  - new corespunde definirii procesului. Procesul primeşte un identificator, se crează blocul de control al procesului, urmând ca execuţia să se facă în momentul în care sistemul poate aloca resursele necesare ;

  - run procesul este în execuţie, adică instrucţiunile sale sunt executate de CPU;

  - wait procesul aşteaptă apariţia unui anumit eveniment, cum ar fi terminarea unei operaţii de I/O sau primirea unui semnal;

  - ready procesul este gata de execuţie aşteptînd să fie servit de către procesor;

  - finish terminarea execuţiei procesului.
- În sistemele cu multiprogramare, procesele trec dintr-o stare în alta, în funcţie de specificul fiecăruia sau de strategia de planificare adoptată. În figura urm. este prezentată diagrama tranziţiilor unui proces în timpul execuţiei.


new
finish
ready
run
wait

- Tranziţiile posibile sunt:

  null→new: Un nou proces este creat ptr. a executa un program.

  new → ready: înseamnă că procesul este luat în considerare pentru a fi executat;

  ready → run: se produce atunci când procesului îi este alocat un procesor;

  run → ready se produce atunci când, conform unei politici de planificare, procesorul trebuie alocat unui alt proces;

  run → wait se produce atunci când procesorul întâlneşte o cerere de executatre a unei operaţii de intrare/ieşire;

  wait → ready se produce atunci când operaţia de I/O s-a terminat;

  run → finish, înseamnă terminarea execuţiei procesului.

Sincronizarea proceselor

- Despre un proces se spune că este **independent** dacă execuţia lui nu afectează sau nu poate fi afectată de execuţia altor procese din sistem.
- Un astfel de proces:

  - poate fi **oprit sau repornit** fără a genera efecte nedorite;

  - este **determinist**(ieşirile depind numai de starea de intrare);

  - este **reproductibil**(rezultatele sunt totdeauna aceleaşi, pentru aceleaşi condiţii de intrare);

  - **nu partajează date** cu alte procese din sistem.
- Dacă execuţia unui proces poate fi afectată de execuţia altor procese din sistem sau poate influenţa stările unor procese, atunci spunem că procesul este **cooperant**. În acest caz, procesul partajează date împreună cu alte procese din sistem, evoluţia lui nu este deterministă, fiind influenţată de stările acestora, nu este reproductibil etc.
- **Sincronizarea** proceselor reprezintă un mecanism prin care un proces activ este capabil să:

  - blocheze execuţia altui proces(trecerea din starea run în ready şi invers)

  - se autoblocheze(să treacă în starea wait);

  - activeze un alt proces.

- **Problema secţiunii critice.** Considerăm un sistem în care există n procese cooperante p_0, p_1,.., p_{n-1}.
- De **exemplu**, să presupunem că două dintre procese, p_0 şi p_1 au acces la o variabilă v; primul proces scade o valoare c_0, iar al doilea adaugă o valoare c_1 la v. Dacă secvenţa de operaţii executate de cele două procese asupra variabilei v se derulează necontrolat, atunci rezultatele sunt total imprevizibile. Putem extinde problema enunţată asupra fişierelor, bazelor de date, tabele din memorie etc.
- O astfel de resursă logică partajată de către două sau mai multe procese se numeşte **resursă critică**.
- Partea de program(segmentul de cod) în care un proces accesează o resursă critică se numeşte **secţiune critică**.
- Problema care se pune este stabilirea unor reguli după care un proces poate să intre în propria sa secţiune critică şi să comunice celorlate procese cooperante când a părăsit-o.
- Astfel, structura unui proces este:

```
do{
      <secţiune de intrare>
      <secţiune critică>
      <secţiune de ieşire>
    <secţiune rămasă>
}while(1);
```

- O soluţie corectă a problemei secţiunii critice trebuie să satisfacă următoarele condiţii:

  - excludere mutuală: la un anumit moment, un singur proces îşi execută propria lui secţiune critică;

  - evoluţie(progres): un proces care nu este în secţiunea sa critică, nu poate să blocheze intrarea altor procese în propriile lor secţiuni critice, atunci când acestea doresc acest lucru;

  - aşteptare limitată: între momentul formulării unei cereri de acces în propria secţiune critică de către un proces şi momentul obţinerii accesului, trebuie acordat un număr limitat de accese celorlalte procese în propriile lor secţiuni critice.

Semafoare

- Conceptul de semafor a fost introdus de Dijkstra. Un semafor s este o pereche (v,q);

  - v este un întreg ce reprezintă valoarea semaforului, fiind iniţializat la crearea semaforului cu o valoare v_0

  - q este o coadă, iniţial vidă în care sunt introduse procesele care nu reuşesc să treacă de semaforul s.
- Asupra semaforului pot acţiona două operaţii indivizibile, W(wait) şi S(signal), executate asupra unui anumit proces.
- Operaţia W poate fi considerată ca o încercare de trecere a semaforului iar operaţia S ca o permisiune de trecere.
- Efectul celor două primitive este descris în continuare ; presupunem ca p este un proces care încearcă să treacă de semaforul s.
- Efectul primitivei W

```
do {
    v--:
    If (v < 0)//procesul p  se introduce în coadă
    { Stare(p)  = wait;
          q← p}
    }
    while(1):
```

- Efectul primitivei S

```
do {
    v++:
    If  (v <= 0)//procesul p este scos din coadă şi
                //activat
        {
                q→ p;
                Stare(p) = ready}
                } while(1):
```

- **Observaţii.**
- 1. Dacă nw şi ns reprezintă numărul primitivelor W, respectiv S executate pe semaforul s, atunci $v = v_0$-nw+ns
- 2. Valoarea iniţială v_0 este un întreg pozitiv.
- 3. La un moment dat, dacă valoarea semaforului v este negativă, respectiv pozitivă, valoarea ei absolută reprezintă numărul proceselor blocate în coadă, respectiv numărul proceselor care pot trece de semaforul s.
- 4. Dacă presupunem că la un moment dat numărul proceselor care au trecut cu succes de semaforul s este nt, atunci evident că $nt \geq nw$. De asemenea: $nt = \min\{v_0+ns, nw\}$
- 5. Modificarea valorilor întregi ale semafoarelor trebuie să fie executate indivizibil, adică atunci când un process modifică valoarea semaforului nici un alt proces nu poate face acest lucru.

Utilizarea semafoarelor în gestionarea resurselor critice.

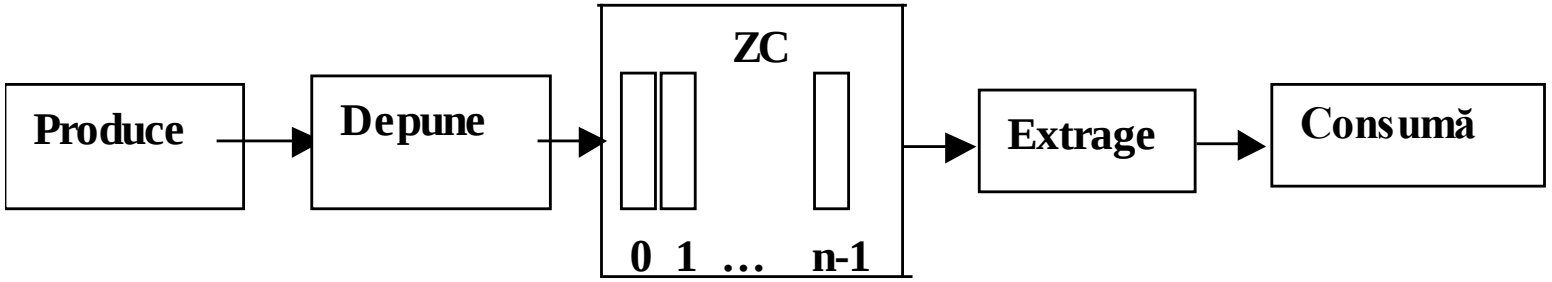
- Putem folosi semafoarele pentru a rezolva problema unei secţiuni critice, partajată de către n procese. Procesoarele folosesc în comun un semafor `mutex` (**mut**ual **ex**clusion), a cărui valoare iniţială este `1`.
- Fiecare proces p_i este organizat astfel:

```
do {
W(mutex);
<secţiune critică>
S(mutex);
<secţiune rămasă>
  } while(1);
```

- Observăm că secţiunea critică este precedată de o primitivă `W`, care aplicată semaforului `mutex` va permite intrarea în secţiunea critică numai în cazul în care resursa critică cerută este liberă.
- După execuţia resursei critice, urmează execuţia primitivei `S` asupra aceluiaşi semafor, al cărei efect este eliberarea resursei şi eventual deblocarea unui proces din coada semaforului, dacă aceasta este nevidă, care să treacă în secţiunea sa critică.
- De asemenea, operaţiile `W` şi `S` trebuie să fie indivizibile.
- Se poate demonstra corectitutinea soluţiei,

Probleme clasice de coordonare a proceselor

- **Problema producător/consumator.** Procesele de tip producător/ consumator apar destul de frecvent în cadrul sistemelor de operare, de exemplu atunci când două procese comunică între ele. Procesul de tip producător **generează informaţiile**(mesajele), care vor fi **folosite** de către procesul de tip consumator. Această problemă poate fi rezolvată prin utilizarea unei zone de memorie accesibilă tuturor proceselor care cooperează. Ea este structurată în funcţie de natura informaţiilor schimbate şi de regulile care guvernează comunicaţia propriu-zisă. Această zonă comună este o resursă critică, ce trebuie protejată.
- Fie procesele p_i şi p_j care comunică între ele. Cele două procese partajează o zonă comună ZC, care este divizată în n celule-mesaj, în care se depun(producătorul), respectiv se extrag(consumatorul) mesaje de dimensiune fixă. Procesele p_i şi p_j au o evoluţie ciclică ce cuprinde operaţiile de producere şi de depunere, respectiv de extragere şi de consumare a unui mesaj din ZC, aşa cum este redat în figura urm.:



- Depunerea, respectiv extragerea unui mesaj din zona comună, trebuie să satisfacă următoarele restricţii:

  a) Consumatorul nu poate să extragă un mesaj, pe care producătorul este în curs să-l depună, adică operaţiile de depunere şi extragere trebuie să se execute în excludere mutuală la nivelul mesajului.

  b) Producătorul nu poate să depună un mesaj atunci când ZC este plină, iar dacă ZC este vidă, consumatorul nu poate extrage nici un mesaj.
- Aceste situaţii limită trebuie detectate şi semnalate proceselor care depind de ele.
- Condiţia a) poate fi satisfăcută prin introducerea unui semafor de excludere mutuală, asociat zonei comune ZC, `mutex(ZC)`, iniţializat cu valoarea `1`.
- Restricţia b) se poate respecta, utilizând semafoarele s_0 şi s_1 pentru a înregistra, în orice moment, starea de ocupare a zonei ZC; s_0, iniţializat cu `0` va indica numărul de celule-mesaj depuse în ZC, iar s_1, iniţializat cu n, va indica numărul celulelor-mesaj libere în ZC.
- Structura proceselor p_i, respectiv p_j este prezentată în continuare.

```
{pi: proces producător de mesaje}
do {
                    construieşte mesaj:
                    W(s1);
                    W(mutex(ZC));
                    ZC←mesaj;
                    S(mutex(ZC));
                    S(s1);
} while(1);
{pj: proces consumator de mesaje}
do {
                    W(s0);
                    W(mutex(ZC));
                    ZC→mesaj;
                    S(mutex(ZC));
                    S(s0);
                    Consumă mesaj
} while(1);
```

- **Problema cititori/scriitori.**
- Atunci când o structură de date, de exemplu un fişier sau o bază de date este accesată în comun de către mai multe procese concurente, dintre care unele doresc doar **să citească** informaţii, iar altele doresc **să actualizeze**(citească, scrie sau modifice) structura respectivă, cele două tipuri de procese se împart generic în **cititori** şi respectiv **scriitori.**
- Se impun următoarele restricţii:

  - Doi cititori pot accesa simultan obiectul partajat;

  - Un cititor şi un scriitor nu pot accesa în acelaşi timp obiectul disputat.
- Pentru a respecta cele două condiţii, accesul la resursă va fi protejat; problema de sincronizare astfel apărută se numeşte **cititori/scriitori.**
- Problema cititori/scriitori poate fi formulată în mai multe variante, toate implicând lucrul cu priorităţi.
- Cea mai simplă dintre acestea, impune ca nici un cititor să nu fie obligat să aştepte decât în cazul în care un scriitor a obţinut deja permisiunea de utilizare a obiectului disputat.
- O altă variantă cere ca, de îndată ce un scriitor este gata de execuţie, el să îşi realizeze scrierea cât mai repede posibil, adică dacă un scriitor aşteaptă pentru a obţine accesul la obiectul disputat, nici un cititor nu va mai primi permisiunea de a accesa la resursă.

- O soluţie a primei variante a problemei cititori/scriitori permite proceselor să partajeze semafoarele A(“Acces”), Sc(“Scrie”) şi variabila cu valori întregi C(“contor”).
- Sc, iniţializat cu 1 este folosit de ambele tipuri de procese. El asigură excluderea mutuală a scriitorilor; este folosit de către primul/ultimul cititor care intră/iese din propria secţiune critică, dar nu şi de către cititorii care intră/ies în/din aceasta în timp ce alţi cititori se află în propriile secţiuni critice.
- Semaforul A, iniţializat cu 1 este folosit pentru protejarea variabilei C, care este iniţializată cu 0 şi are rolul de a memora numărul proceselor care citesc din obiectul partajat la momentul considerat.
- Primul cititor care accesează resursa critică(C=1) blochează accesul scriitorilor, iar ultimul cititor (C=0), deblochează accesul la obiectul partajat.
- Structura generală celor doua procese este prezentată în continuare.

```
do
    {W(A);
    C++;
    if (C ==1 )  W(Sc);
    S(A);
    <citirea>
    W(A);
    C--;
    if (C == 0) S(Sc);}
while(1);
```

- a) Procesul cititor

```
do {
    W(Sc);
<scrirea>
    S(Sc);
    }
while(1);
```

- b) Procesul scriitor

Fire de execuţie

- În mod tradiţional, fiecărei aplicaţii îi corespunde un proces.
- Pe calculatoarele moderne se execută multe aplicaţii în care, din punct de vedere logic, sunt structurate pe activităţi multiple, care s-ar putea desfăşura simultan.
- **Exemplu:**

  - în cadrul unui „browser” de WWW, o activitate poate fi afişarea unor texte sau imagini, pe când alta poate fi regăsirea unor informaţii de pe anumite servere dintr-o reţea de calculatoare.

  - un procesor de texte poate executa simultan afişarea de imagini sau texte, citirea unor comenzi de la tastatură şi verificarea greşelilor gramaticale sau de sintaxă ale textului introdus.
- Modelul proceselor, se bazează pe două considerente principale:

  - procesul dispune de o colecţie de resurse, între care un **spaţiu de memorie**, necesare execuţiei sale;

  - procesul are un **singur fir de control**, adică la un moment dat cel mult o singură instrucţiune a procesului poate fi în execuţie pe un procesor.

- Într-un sistem de operare multiproces, una dintre operaţiile de bază ale SO, o constituie **comutarea proceselor,** conform unui algoritm de planificare.
- Comutarea proceselor implică salvarea de către nucleu a unor informaţii suficiente, care ulterior să fie restaurate, atunci când procesorul respectiv va primi din nou serviciile CPU, pentru ca execuţia să fie reluată cu instrucţiunea următoare celei care a fost executată ultima.
- Aceste informaţii se referă la diverse categorii de resurse deţinute de proces, ceea ce face ca operaţia de comutare a proceselor să fie relativ costisitoare din punctul de vedere al timpului procesor consumat pentru efectuarea ei.
- De asemenea, crearea unui nou proces în sistem este o operaţie de o complexitate deosebită.

- Atât cerinţele unor noi domenii de aplicaţii, cât şi dezvoltările tehnologice şi arhitecturale ale sistemelor de calcul(în special sistemele multiprocesor), au impus renunţarea la condiţia de unicitate a firului de control în cadrul unui proces. A apărut astfel conceptul de **fir de execuţie** (**thread**); mai multe asemenea fire pot coexista într-un proces, utilizând în comun resursele procesului. Avantajele principale fiind:

  - eliminarea costului creării de noi procese;

  - **simplificarea comutării**, pentru că numărul de informaţii salvate/restaurate va fi mai redus;

  - posibilitatea ca o **aplicaţie interactivă să-şi continue execuţia,** chiar dacă o parte a ei este blocată sau execută o operaţie mai lungă în timp; de **exemplu**, un browser de WWW menţine interacţiunea cu utilizatorul, chiar dacă a început încărcarea unei imagini;

  - utilizarea arhitecturilor multiprocesor, care permit ca fiecărui fir de control să-i fie alocat un procesor.

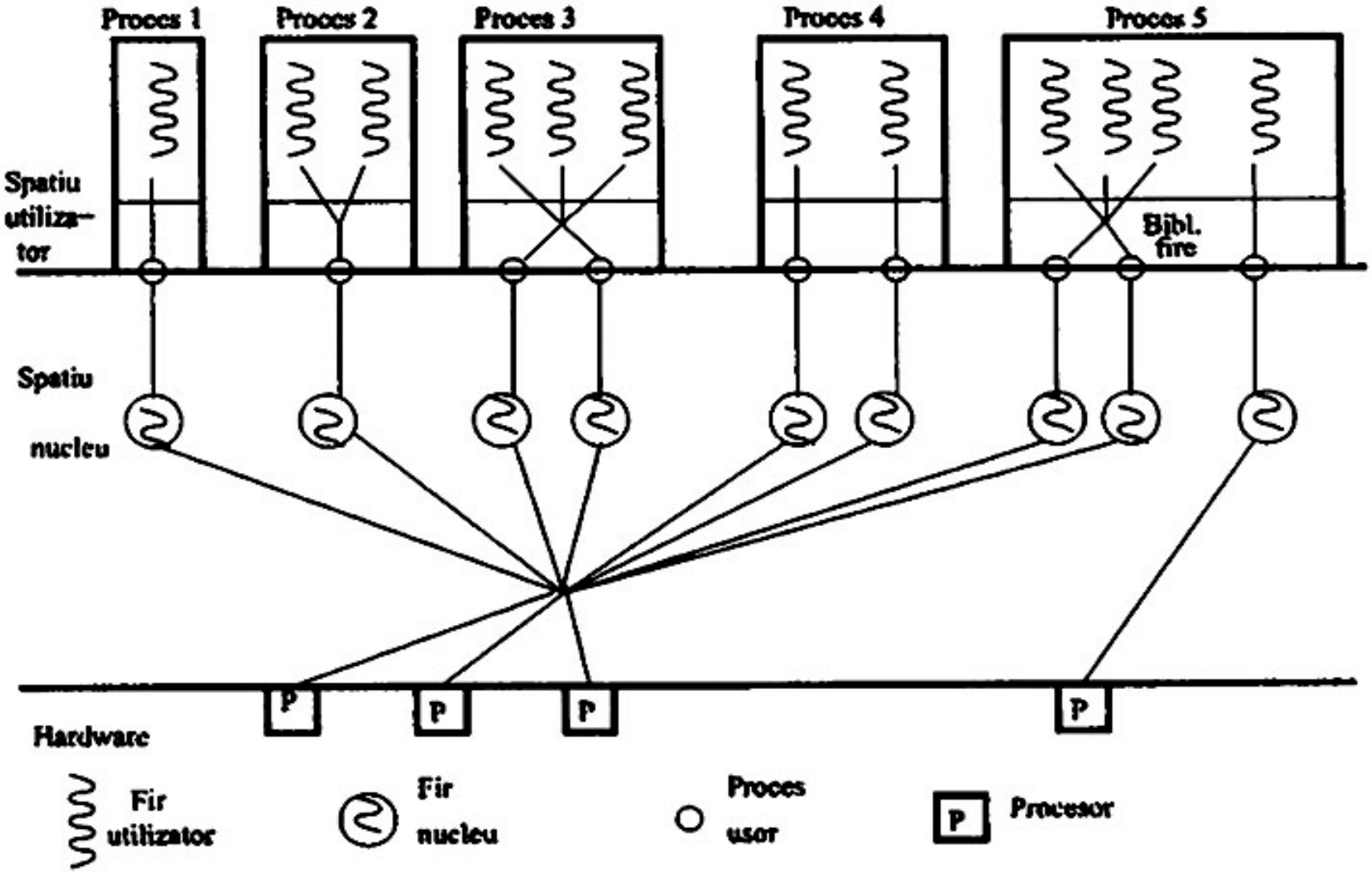
- **Firele de execuţie** pot coexista într-un proces(în acelaşi spaţiu de memorie), evident cu necesitatea de a fi sincronizate atunci când solicită acces la resurse partajate, ca şi în cazul proceselor.
- Resursele proprii fiecărui fir de execuţie sunt, în general limitate la: stivă, variabile locale şi numărător(contor) de program.
- Variabilele globale ale unui program vor fi implicit accesibile tuturor firelor din acel program.
- Firele de execuţie se împart în;

  • **fire ale utilizatorului,** implementate de către o bibliotecă ce poate fi accesată de către utilizator şi care oferă posibilităţi de creare, planificare şi administrare a **thred**-urilor, fără a fi necesar suportul nucleului; acestora li se asigură spaţiu de memorie în vederea execuţiei lor în spaţiul utilizator;

  • **fire ale nucleului SO,** implementate de către sistemul de operare, care realizează crearea, planificarea şi administrarea lor în spaţiul nucleului, ele realizând anumite operaţii protejate cerute de către procese.

- Utilizarea firelor de execuţie se poate realiza folosind mai multe modele:
  - **mai multe fire utilizator care corespund la un fir al nucleului,** este caracterizat de eficienţă, deoarece administrarea lor este realizată în spaţiul utilizatorului, dar în cazul în care unul dintre firele utilizator generează un apele de sistem, prin care intră în starea de blocare, întregul proces va fi blocat; de asemenea, deoarece numai un fir de execuţie poate, la un moment dat să acceseze nucleul, nu este posibil ca mai multe fire de execuţie să se execute în paralel, pe un sistem multiprocesor.
  - **un fir utilizator care corespunde la un fir al nucleului** permite concurenţa execuţiei, deoarece atunci când un fir de execuţie a iniţiat un apel de sistem care generează blocarea, celelalte fire de execuţie se pot executa în continuare; dezavantajul constă în numărul mare de fire nucleu care trebuie create.
  - **mai multe fire utilizator care corespund la mai multe fire ale nucleului(**în număr mai mic sau egal decât cele ale utilizatorului) este o situaţie specifică, sistemelor multiprocesor, deoarece este posibil, ca la un moment dat, mai multe fire ale unei aplicaţii, care sunt servite de procesoare diferite, să lanseze un acelaşi apel de sistem, care să fie rezolvat de fire nucleu diferite.

- Există mai multe variante conceptuale şi de implementare, după cum firele de execuţie se manifestă numai în spaţiul utilizator sau au reprezentare şi în nucleu. Modelul cel mai complet este cel oferit în sistemul de operare Solaris, prezentat în figura urm.
- **Procesele uşoare** (**lightweight process**, prescurtat LWP) pot fi considerate ca o corespondenţă între firele la nivelul utilizatorului şi firele din nucleu. Fiecare proces uşor este legat, pe de o parte de una sau mai multe fire utilizator, iar pe de altă parte, de un fir nucleu. Planificarea pentru execuţie a LWP se face independent, astfel că LWP separate pot fi executate simultan în sisteme multiprocesor.
- Figura urm. ilustrează diversele relaţii care se pot stabili între cele 4 tipuri de entităţi. Se observă, în primul rând că un LWP este vizibil şi în spaţiul utilizator, ceea ce înseamnă că vor exista structuri de date corespunzătoare LWP atât în cadrul unui proces, cât şi în cadrul nucleului.
- **Procesul 1** din figura urm. este un proces "clasic": el are un singur fir de execuţie la nivelul utilizatorului, legat cu un LWP. **Procesul 2** este o ilustrare a situaţiei în care firele de execuţie multiple sunt vizibile numai în spaţiul utilizator: toate sunt puse în corespondenţă cu un singur LWP, deci la un moment dat numai unul dintre firele procesului 2 poate fi în execuţie, pentru că planificarea se face la nivelul LWP. **Procesul 3** are mai multe fire de execuţie, multiplexate pe un număr mai mic de LWP. Aplicaţiile de acest gen conţin paralelism, al cărui grad poate fi controlat prin numărul de LWP puse la dispoziţie.


Proces 1
Proces 2
Proces 3
Proces 4
Proces 5
Spatiu utilizator
Bibl. fire
Spatiu nucleu
Hardware
P
P
P
P
Fir utilizator
Fir nucleu
Proces usor
P
Procesor

- **Procesul 4** are fiecare fir de execuţie pus în corespondenţă cu un LWP, ceea ce face ca nivelul de paralelism din aplicaţie să fie egal cu cel din nucleu.
- Acest aranjament este util dacă firele de execuţie ale aplicaţiei sunt frecvent suspendate cu blocare(fiecare fir suspendat cu blocare ţine blocat şi LWP-ul asociat).
- P**rocesul 5** conţine atât fire care multiplexează un număr de LWP, cât şi un fir asociat permanent cu un LWP, care la rândul său este asociat permanent cu un procesor.
- Pe lângă situaţiile ilustrate în figură, unde există numai fire de execuţie nucleu asociate unor fire utilizator, nucleul poate conţine şi fire care nu sunt asociate unor fire utilizator. Asemenea fire sunt create, rulate şi apoi distruse de nucleu, pentru a executa anumite funcţii ale sistemului.
- Utilizarea de fire nucleu în locul proceselor, contribuie la reducerea costurilor de comutare în interiorul nucleului.