Programarea aplicațiilor pentru conducere în timp real

Descrierea CIP a Bibliotecii Naționale a României
LUPU, CIPRIAN
Programarea aplicațiilor pentru conducere în timp real / Ciprian Lupu, Monica Dragoicea. - Bucureşti : Editura Academiei Oamenilor de Ştiință din România, 2011
Bibliogr.
Index
ISBN 978-606-8371-27-6

I. Dragoicea, Monica

681.5:004.031.43

Editura Academiei Oamenilor de Ştiință din România

Adresa: Splaiul Independenței, nr. 54, sectorul 5, cod 050094 Bucureşti, România

Redactor: ing. Mihail CĂRUȚAŞU

Documentarist: ing. Ioan BALINT

Coperta: ing. sist. Adrian Nicolae STAN

Ciprian Lupu

Monica Dragoicea

Programarea aplicațiilor pentru conducere în timp real

Editura Academiei Oamenilor de Ştiință din România

Bucureşti

2011

Cuprins

1. Problematica sistemelor în timp real 7
Introducere 7
Sisteme în timp real 7
Sisteme de operare în timp real 10
Procese şi fire de execuție 13
Concurența în sistemele în timp real 15
Planificarea taskurilor în sisteme în timp real 18
Tranziții de stare ale proceselor / firelor de execuție 24
2. Aspecte concurențiale ale proiectării aplicațiilor în timp real 27
Aspecte ale programării concurente 27
Sincronizarea taskurilor 29
Excluderea mutuală 30
Comunicația între taskuri 31
Inversiunea de prioritate 31
3. Aspecte temporale ale aplicațiilor în timp real 35
Măsurarea timpului 35
Planificarea taskurilor pe condiție de timp 36
Depăşiri de timp (timeouts) 38
Contoare de timp (timer-e) 38
Accesul la o bază de timp - standardul ANSI C 39
4. Organizarea aplicațiilor software în timp real 43
Aspecte ale proiectării aplicațiilor în timp real 43
Implementarea algoritmilor de conducere în timp real 45
5. Arhitecturi hardware de conducere în timp real 55
Introducere 55
Funcții 57
Concepte ale sistemelor în TR şi aplicabilitate 59
Concepte 59
Aplicabilitate 60
Tipuri de structuri de TR 61
Alte exemple clasice 65
6. Implementarea practică a aplicațiilor de conducere în timp real . 67
Elemente de bază 67
Elemente componente ale unei aplicații de timp real 68
Întreruperile 69

Implementarea programului principal (bucla infinita) 71
Mecanismul de protecție WatchDog. 75
Implementarea unui driver de achiziție 75
Algoritmi de reglare 81
Modele, filtre, implementare 82
Algoritmi de reglare bi şi tri - poziționali 83
Algoritmi de reglare PID 88
Vizualizarea datelor – consola operator 90
Setarea şi înregistrarea valorilor alarmelor 94
Semnalizarea situațiilor de funcționare anormala 94
Generarea unor rapoarte statistice 95
7. Bibliografie 97

Capitolul 1

Problematica sistemelor în timp real

Introducere

Sistemele în timp-real şi sistemele încorporate (*embedded*) reprezintă o tehnologie informațională digitală care este integrată mediului în care trăim astăzi şi care ne influențează din ce în ce mai mult fiecare aspect al vieții. Această observație nu se referă doar la aplicațiile cu aspecte critice de siguranță, cum sunt cele dedicate industriei automobilelor, aviației, aplicațiilor din domeniile aerospațial sau medical, ci şi celor dedicate comunicațiilor, telefoniei mobile şi tehnologiilor de tip "e-", caselor inteligente, a obiectelor de uz general, sau divertismentului. Toate aceste aplicații au impact deosebit în toate sferele activității umane, incluzând aspecte de securitate, viață privată, precum şi modele de activitate productivă sau stil de viață.

Astăzi, mai mult de 98% din producția de microprocesoare este utilizată în sistemele incorporate, nemaifiind vizibilă utilizatorului sub forma dispozitivelor electronice definite general de termenul "calculator". Domeniul sistemelor incorporate utilizează o gamă largă de elemente structurale software şi hardware dedicate, noi procesoare şi metode de procesare a datelor, senzori, elemente de execuție, medii şi infrastructuri de comunicație, combinate într-o sinergie aproape invizibilă pentru utilizator, dar omniprezentă în jurul nostru.

Acest capitol prezintă principalele aspecte referitoare la proiectarea şi implementarea sistemelor în timp-real, punând accent pe caracteristici de implementare a *sistemelor în timp-real utilizate pentru conducerea proceselor*. Prin definiție, un *sistem în timp-real* conduce un proces fizic luând în considerare aspecte referitoare la execuția în timp. Aceste sisteme vor funcționa sub constrângeri mult mai severe de timp decât un program software obişnuit, ceea ce le impune o fiabilitate sporită pe perioade lungi de timp. Anumite constrângeri sunt inerente domeniului de funcționare, şi anume planificarea, predictibilitatea şi robustețea. Alte constrângeri sunt impuse de necesitatea de a reduce costurile de implementare prin limitarea memoriei disponibile sau a capacității de procesare a procesorului. Cele mai multe sisteme în timp-real dispun de memorie limitată şi de un suport hardware minimal. Privite în ansamblu, aceste constrângeri complică semnificativ efortul de dezvoltare a acestui tip de sisteme de calcul.

Sisteme în timp real

Vorbim despre un **sistem în timp real** atunci când există o interacțiune continuă a acestuia cu mediul înconjurător (sistemul este de tip reactiv). Deoarece mediul extern se schimbă în **timp real**, el impune constrângeri asupra timpului de procesare a informației în sistemul respectiv, **cu procesare în timp-real**. De

asemenea, sistemul în timp-real controlează şi / sau reacționează simultan diverse / la diverse aspecte ale mediului (sistemul în timp-real procesează date în mod concurent).

Un **sistem (cu procesare) în timp-real** este un sistem de calcul (calculator) pentru care momentul în care este generat un rezultat al calculelor este important (deoarece intrarea prezentată pentru procesare sistemului corespunde unei activități din lumea reală, iar ieşirea trebuie corelată cu această intrare). Timpul de procesare trebuie să fie suficient de mic pentru obținerea unor rezultate coerente în raport cu timpul de execuție. Astfel, funcționarea corectă a unui sistem în timp-real este validată de două componente, şi anume *corectitudinea rezultatului* produs şi *momentul de timp* la care este generat acest rezultat. Generarea unui set de date pentru procesare este definită de un **eveniment**, iar momentul de timp la care trebuie livrat răspunsul este specificat de un ***deadline***, *Figura 1.1*.

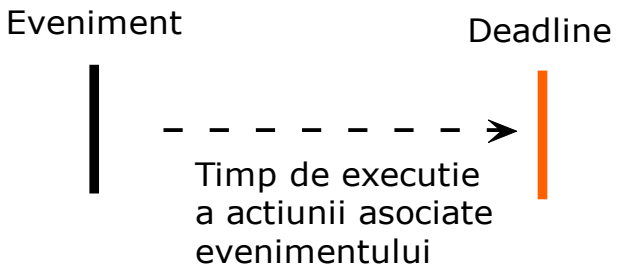

Figura 1.1.: Asocierea evenimentelor şi *deadline*-urilor în sisteme în timp real [1]

În funcție de tipul *deadline*-ului, sistemele în timp-real pot fi clasificate astfel:

- **sisteme în timp-real de tip soft**. Utilitatea[1] răspunsului descreşte în timp după expirarea *deadline*-ului, *Figura 1.2* c) (de exemplu, aplicații multimedia). Deoarece defectarea unui sistem în timp-real de tip soft nu are consecințe fatale, proiectarea lor necesită proceduri mai puțin riguroase decât sistemele în timp-real de tip hard;
- **sisteme în timp-real de tip ferm**. Răspunsul sistemului îşi pierde utilitatea imediat după expirarea *deadline*-ului, însă consecințele nu sunt grave, *Figura 1.2* b) (de exemplu, tranzacțiile efectuate într-o bază de date);
- **sisteme în timp-real de tip hard (critice)**. Consecințele neîndeplinirii *deadline*-ului pot fi catrastrofale, utilitatea răspunsului după expirarea *deadline*-ului este zero, *Figura1.2* a) (de exemplu, sisteme de conducere pentru avioane, centrale nucleare, instalații chimice, sisteme de conducere pentru automobile etc).

[1] Funcțiile de utilitate modelează importanța unei acțiuni ("utilitatea" sa) ca o funcție de timp, de la apariția evenimentului care provoacă acțiunea sau de la începutul propriu-zis al respectivei acțiuni.

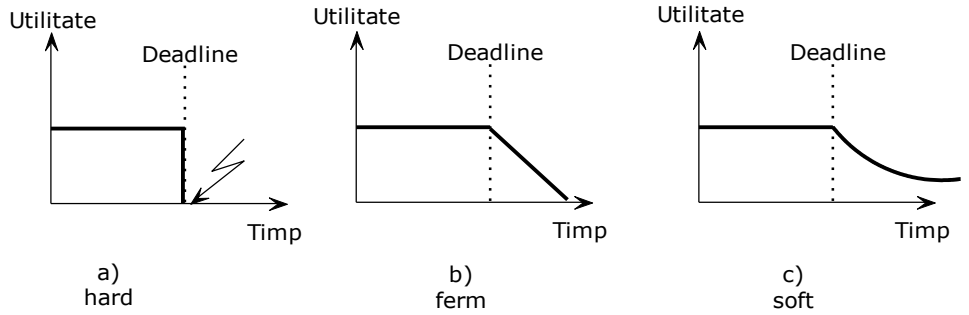

Figura 1.2: Clasificarea sistemelor în timp real în funcție de *deadline*-uri [1]

Deoarece funcționarea sistemelor în timp-real critice vizează aspecte critice, procedurile de proiectare asociate trebuie să asigure siguranța în funcționare în toate situațiile specificate, chiar dacă acestea apar foarte rar, şi trebuie evaluată prin certificări ale unor agenții specializate.

Procesarea datelor în sisteme în timp-real are trei componente majore:

- **interacțiunea cu mediul extern**. Acțiunile din mediul cu care interacționează sistemele care controlează activități specifice se desfăşoară în paralel, astfel încât aplicațiile software care rulează pe astfel de sisteme, controlând interfețe cu entități din lumea reală (roboți, benzi transportoare etc), trebuie să reflecte natura paralelă a sistemului în structura programului. Astfel, o problemă majoră a dezvoltării de software pentru sisteme în timp-real este exprimarea concurenței (paralelismului) în structura programului;
- **timpul**. După cum s-a menționat anterior, într-un sistem în timp real contează atât corectitudinea răspunsului, cât şi momentul de timp la care este furnizat acest rezultat, moment specificat prin *deadline*-uri;
- **fiabilitatea**. Un sistem în timp real trebuie să fie fiabil, să poată fi reconfigurat în caz de funcționare defectuoasă, deoarece funcționarea sa guvernează acțiuni critice.

Sistemele în timp real variază de la sisteme cu un singur procesor (care au câteva sute de linii de cod) până la sisteme distribuite multiplatformă şi multilimbaj (care au milioane de linii de cod). Un sistem în timp-real poate fi descompus din punct de vedere conceptual într-o mulțime de subsisteme, denumite clustere (*Figura 1.3*), care corespund obiectului condus prin software (engl. *controlled cluster*), sistemului de calcul în timp-real (engl. *computational cluster*), şi operatorului uman (engl. *operator cluster*).

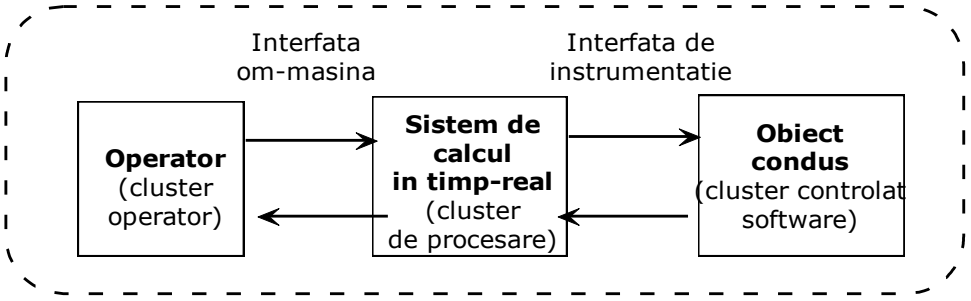

Figura 1.3: Structura unui sistem în timp real [1]

Dacă sistemul de calcul în timp real este distribuit, sistemul în timp real va fi format dintr-o mulțime de noduri (calculatoare) interconectate prin intermediul unei rețele de comunicație în timp real (*Figura 1.4*).

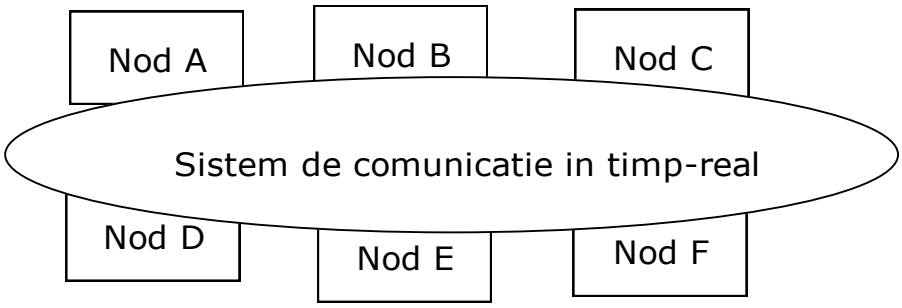

Figura 1.4: Structura unui sistem distribuit [1]

Interfața om-mașină (între operatorul uman şi sistemul de calcul în timp real) este formată din dispozitive de intrare (de exemplu, tastatura) şi de ieşire (de exemplu, monitorul) care asigură interacțiunea cu operatorul uman. Interfața de instrumentație este formată din senzori şi elemente de execuție care transformă semnale fizice ale obiectului condus (tensiuni, curenți etc) în semnale numerice şi invers. Un nod care conține o interfață de instrumentație poartă denumirea de *nod de interfață*.

Sisteme de operare în timp real

Un concept important legat de noțiunea de sistem în timp real este acela de **task**. Definind un task ca fiind o funcție individuală care se execută (eventual în mod repetat) în contextul unei anumite aplicații software, aplicația poate fi descrisă ca reprezentând o trecere progresivă printr-o secvență de taskuri. La fiecare moment de timp, în cadrul aplicației se execută un anumit task, şi acest task trebuie planificat (engl. *task scheduling*). Astfel, datorită principalei sale funcții, aceea de

management al proceselor (şi implicit, de planificare a taskurilor implementate ca şi procese) sistemul de operare devine cea mai importantă componentă a unui sistem în timp-real.

Un sistem de operare reprezintă o mulțime de biblioteci software care au un rol dublu într-un sistem în timp real (*Figura 1.5*):

- de a furniza un nivel de abstractizare astfel încât software-ul de deasupra sistemului de operare să fie mai puțin dependent de hardware-ul pe care rulează aplicația;
- de a gestiona diversele resurse hardware şi software ale sistemului, pentru a asigura o funcționare eficientă şi fiabilă a întregului sistem.

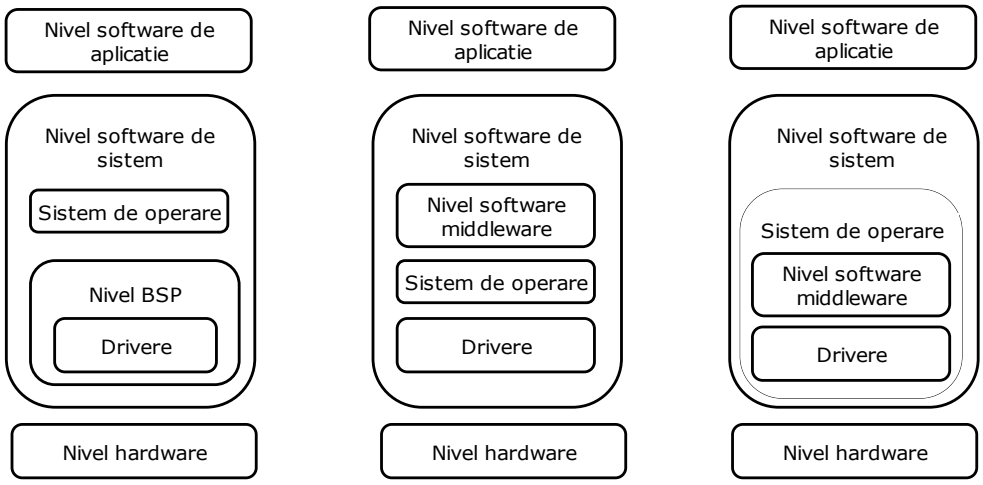

Figura 1.5: Model de organizare a unui sistem (încorporat) în timp real [1]

Cea mai importantă componentă a unui sistem de operare este **nucleul** (engl. *kernel*), peste care pot fi "adăugate" diverse alte componente. Nucleul este componenta care asigură funcționalitatea de bază a sistemului de operare, şi anume:

- **gestiunea proceselor**. De cele mai multe ori, în sistemele în timp real noțiunile de proces şi task se referă la acelaşi concept. Un proces este de fapt un cod program care se află într-o anumită stare de execuție. Procesul are propriul său spațiu de adresă şi un singur fir de execuție (engl. *flow of control*). Contorul program al procesului conține adresa următoarei instrucțiuni care trebuie executată. Într-un sistem cu multiprogramare la un moment dat în memoria fizică pot fi încărcate mai multe procese, astfel încât acestea să poată fi executate concurent, partajând atât unitatea centrală, cât şi dispozitivele periferice;
- **gestiunea memoriei**. În cazul sistemelor în timp real spațiul de memorie este partajat de mai multe procese, astfel încât accesul la zone de memorie şi alocarea acestora trebuie gestionate prin mecanisme

specifice. Sistemul de operare are nevoie de un mecanism de securitate care să protejeze atât codul taskului aflat în execuție de alte taskuri independente, cât şi propriul cod de sistem față de codul taskului pe care îl execută. În general, responsabilitățile de gestiune de memorie ale nucleului asigură:

- gestiunea corespondențelor între memoria fizică şi referințele de memorie ale taskului;
- determinarea procesului care va fi încărcat în spațiul disponibil de memorie;
- alocarea şi dealocarea memoriei alocate proceselor;
- serveşte cererile de alocare / dealocare de memorie ale proceselor (de tip alloc / malloc, sau alte rutine de alocare / dealocare specifice);
- urmăreşte utilizarea memoriei în cadrul componentelor sistemului;
- coerența memoriei cache (pentru sistemele care au un astfel de nivel);
- protecția memoriei alocate proceselor;

- **gestiunea operațiilor de I/O**. Dispozitivele de I/O sunt resurse partajate care pot fi utilizate de mai multe procese, şi astfel, la fel ca şi în cazul memoriei, sunt necesare mecanisme de acces şi alocare a acestora. De asemenea, prin intermediul sistemului de gestiune a operațiilor de I/O poate fi implementat şi managementul sistemului de fişiere, ca o metodă de stocare şi gestiune a datelor sub forma fişierelor. Astfel:
  - Un sistem de operare furnizează o interfață uniformă către dispozitivele de I/O care execută o varietate de funcții prin intermediul funcțiilor de sistem (engl. *system calls*) ale nucleului. În acest fel se asigură protecția dispozitivelor de I/O, deoarece procesele de tip utilizator nu pot accesa aceste dispozitive decât prin prin intermediul funcțiilor de sistem dedicate. În acelaşi timp se asigură o schemă de partajare eficientă a dispozitivelor de I/O între mai multe procese. Schema de management a operațiilor de I/O a sistemului de operare este construită în mod generic ca o interfață către procesele de tip utilizator şi către *driver*-ele pentru dispozitive (engl. *device drivers*), care cuprinde totodată şi un mecanism de buffer-caching. Astfel, sistemul de operare poate gestiona comunicația sincronă şi asincronă cu procesele sale de I/O;
  - Sistemele de operare, atât cele de uz general, cât şi cele dedicate în special implementării sistemelor în timp real, asigură suport de gestiune de memorie pentru scheme de memorare de tip sistem de fişiere temporar sau permanent pentru diverse dispozitive de memorare (memorii Flash, memorie RAM, hard disk). Sistemul de fişiere reprezintă o colecție de fişiere, împreună cu protocolul

de gestiune dedicat. De exemplu, FAT32 - File Allocation Table (memoria alocată în sectoare organizate în clustere, accesate pe 32 biți), NFS - Network File System (bazat pe RPC - Remote Procedure Call şi XDR - Extended Data Representation), FFS - Flash File System (pentru memorii de tip Flash).

Algoritmii utilizați pentru gestiunea sistemului de fişiere reprezintă software de tip *middleware* şi / sau software de aplicație, instalat (engl. *mounted*) într-o anumită locație (engl. *mount point*) pe dispozitivul de memorare. În relație cu sistemul de fişiere, nucleul furnizează cel puțin următoarele mecanisme pentru gestiunea sistemului de fişiere:

- mapare fişiere pe un al doilea sistem de stocare, de exemplu, memorie Flash sau memorie internă RAM;
- suport pentru primitivele de manipulare a fişierelor şi directoarelor:
  - *atribute şi definiții fişiere*: tip (executabil, obiect, sursă, multimedia etc), dimensiune, permisiune de acces (citire, scriere, execuție, adăugare, ştergere etc), apartenență (*ownership*) etc;
  - *operații asupra fişierelor*: creare, ştergere, citire, scriere, deschidere, închidere etc;
  - *metode de acces la fişiere*: secvențial, direct etc.;
  - *crearea, ştergerea şi accesul la directoare*.

Procese şi fire de execuție

Diferitele sisteme de operare dau adeseori înțelesuri diferite unor termeni ca **proces**, **fir de execuție**, **task**, **program** etc. De exemplu, sistemul de operare în timp-real QNX foloseşte în mod uzual doar termenii proces şi fir de execuție. O **aplicație** dezvoltată sub QNX este formată dintr-o **colecție de procese**, iar termenul **program** este în mod uzual echivalent cu **proces**.

Un **fir de execuție** (engl. *thread*) reprezintă un singur flux de execuție (control). Prin intermediul firelor de execuție este posibilă gestionarea diferită a spațiului de adresă alocat unui proces (*Figura 1.6*). Spațiile de cod, date şi memorie disponibilă (*heap*) sunt aceleaşi pentru toate firele de execuție care aparțin aceluiaşi proces. Comunicația între firele de execuție se face prin zonele comune de memorie. Fiecare fir de execuție are starea lui. Comutarea între firele de execuție este mult mai simplă decât în cazul proceselor (se salvează câțiva regiştri alocați firului respectiv şi stiva) şi rapidă.

Un **proces** este un cod program care se află într-o anumită stare de execuție. Procesul are propriul său spațiu de adresă (*Figura 1.7*) şi un singur flux de execuție (engl. *flow of control*). Contorul program al procesului conține adresa următoarei instrucțiuni care trebuie executată.

Un **proces** este un program încărcat în memorie, recunoscut în mod unic printr-un identificator (pid). Un proces deține propriile resurse (*Figura 1.8*), cum sunt memoria (cod şi date), fişiere deschise, o identitate prin apartenența la un grup de utilizatori (user id şi group id), contoare de timp (*timer*-e), canale de

comunicație etc. Resursele deținute de un proces sunt protejate, astfel încât alte procese nu pot avea acces la ele.

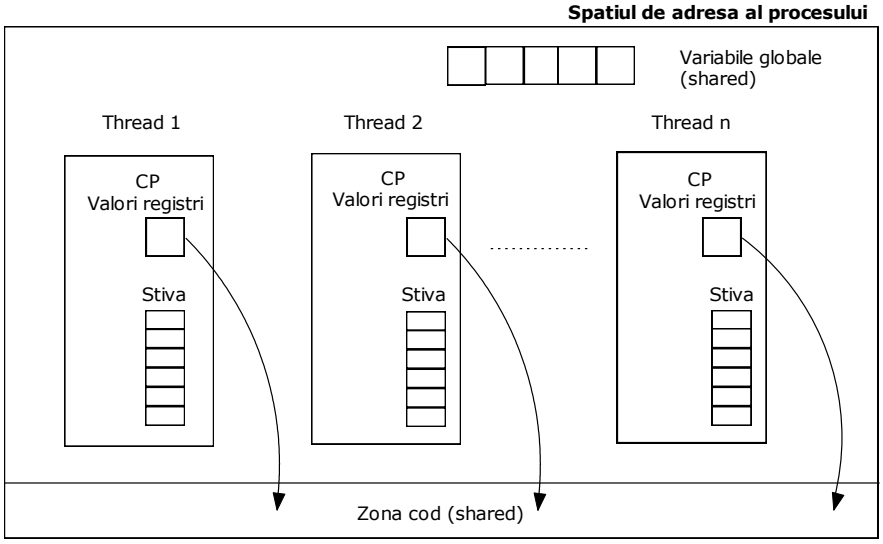

Figura 1.6: Gestiunea memoriei în cazul firelor de execuție [1]

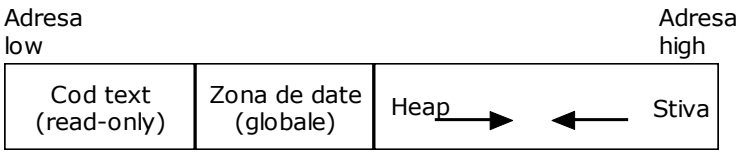

Figura 1.7: Gestiunea memoriei în cazul proceselor [1]

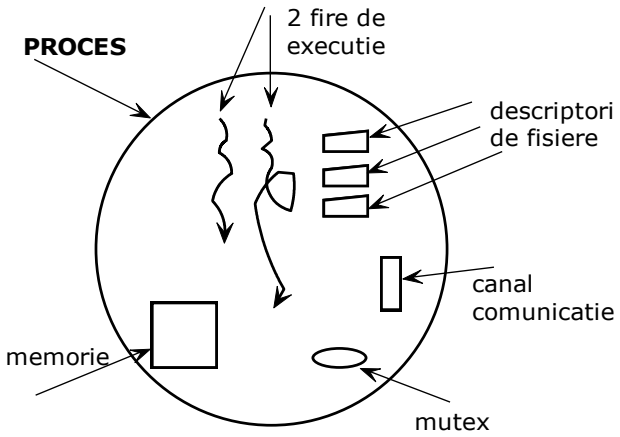

Figura 1.8: Procese şi fire de execuție [1]

Procesele pot fi create şi lansate în execuţie prin mai multe mecanisme, atât din linia de comandă, cât şi din interiorul altui program (proces). Un proces poate fi creat din interiorul altui program în două feluri:

- prin transformarea programului (procesului) original în alt proces, complet nou;
- prin pornirea din programul (procesul) original a unui proces cu totul nou.

Mecanismele tradiţionale de tip UNIX furnizează mai multe funcţii şi familii de funcţii pentru crearea proceselor concurente:

- funcţia system(). Primeşte ca argument o linie de comandă, la fel cum această comandă ar putea fi executată din shell (de exemplu, system("pwd"));
- familia de funcţii exec(). Transformă procesul curent în alt proces (procesul curent nu mai execută programul curent, ci alt program). Identificatorul procesului (pid - ul) nu se schimbă;
- funcţia fork(). Se creează un proces nou, identic cu cel care rulează în mod curent. Cele două procese vor rula în paralel, au acelaşi cod şi date, dar pid-urile diferă;
- familia de funcţii spawn(). Se creează un alt proces fiu (cu alt identificator – pid), care va executa programul specificat de argumentele funcţiei spawn().

Concurenţa în sistemele în timp real

O aplicaţie software în timp real, care rulează într-un sistem în timp real, este compusă în mod obişnuit din mai multe acţiuni care se execută concurent. Problema apare atunci când între aceste acţiuni apar interacţiuni. Interacţiunile posibile între acţiuni pot fi de mai multe tipuri şi pot fi exprimate prin operaţii specifice procesării concurente. Procesarea concurentă a informaţiilor reprezintă un aspect fundamental al sistemelor de calcul moderne, cărora le sunt caracteristice:

- **multiprogramarea** (gestiunea mai multor procese sau fire de execuţie într-un sistem uniprocesor);
- **multiprocesarea** (gestiunea mai multor procese sau fire de execuţie într-un sistem multiprocesor);
- **procesarea distribuită** (gestiunea proceselor multiple care se execută pe mai multe calculatoare care sunt conectate într-o reţea şi care pot avea sisteme de fişiere diferite sau arhitecturi hardware diferite).

Programarea paralelă (sau **concurentă**) reprezintă activitatea de scriere a unui program / aplicaţie software care conţine o serie de părţi necesar a se afla în execuţie în acelaşi moment. Părţile componente ale programului, care vor fi executate în paralel, poartă denumirea, în mod generic, de **taskuri.**

Taskul este cea mai mică unitate de prelucrare a informaţiei căreia i se atribuie o identitate, reprezentând un program în formă executabilă compus dintr-o succesiune de instrucţiuni executate secvenţial, astfel:

- două **taskuri** sunt **paralele** (sau **concurente**) dacă prima instrucțiune a unui task începe să se execute înainte ca ultima instrucțiune a celuilalt task să fie finalizată;
- două sau mai multe **taskuri** se numesc **disjuncte** (independente) dacă nu schimbă informații între ele sau dacă nu utilizează resurse (zone de memorie, dispozitive periferice) în comun. În caz contrar, taskurile interacționează.

Taskul reprezintă de fapt execuția unui program secvențial. Execuția sa începe cu citirea datelor de intrare şi a stării interne a taskurilor şi se termină cu generarea rezultatelor şi actualizarea stării interne. Semnalul de control care inițiază execuția taskului este furnizat de către sistemul de operare.

Într-un sistem cu multiprogramare (uniprocesor), taskurile pot fi intercalate (*Figura 1.9*) în timp, astfel încât să se creeze aparența execuției simultane.

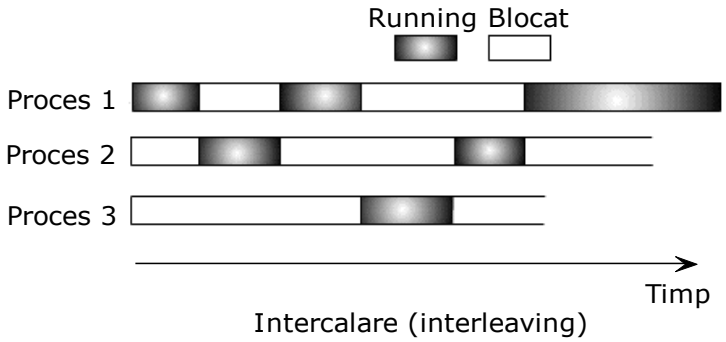

Figura 1.9: Planificarea taskurilor într-un sistem cu multiprogramare [1]

Astfel, în cazul echipamentelor monoprocesor, dotate cu un sistem de operare în timp-real cu facilități multitasking (de exemplu QNX, Timesys Linux, VxWorks) execuția paralelă a taskurilor se face intercalat, pe principiul diviziunii timpului unității centrale între task-uri (engl. *time slicing*). La un moment dat un singur task deține controlul procesorului, astfel încât are loc o execuție pseudo-paralelă (sau execuție paralelă logică). Un sistem de operare multitasking permite execuția concurentă a mai multor programe. Deoarece pe un calculator dotat cu un singur procesor poate rula un singur program la un moment dat, caracterul multitasking este obținut prin rularea unui program (task) un interval specificat de timp (de exemplu, un *time slice* egal cu 10 ms), după care este oprit (se salvează contextul de execuție pentru a putea fi repornit mai târziu), după care se activează alt task. Intervalul de timp (*time slice*) fiind suficient de mic, se creează iluzia rulării concurente a taskurilor.

Într-un sistem multiprocesor este posibilă atât intercalarea (engl. *interleaving*) cât şi suprapunerea (engl. *overlapping*) execuției taskurilor (*Figura 1.10*). Astfel, în cazul sistemelor multiprocesor poate fi realizat un paralelism real al execuției taskurilor, în sensul că fiecare task poate fi executat pe un alt procesor.

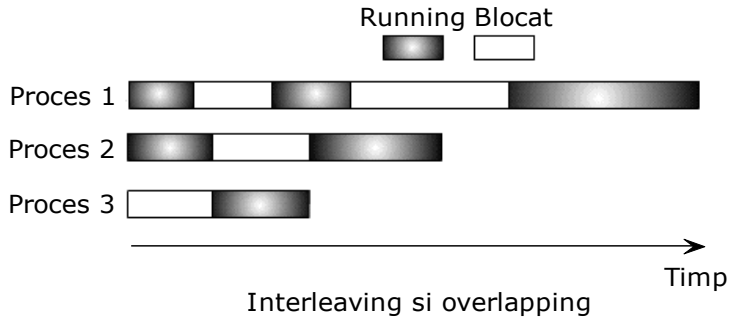

Figura 1.10: Planificarea taskurilor într-un sistem cu multiprocesare [1]

Ambele tehnici, intercalarea şi suprapunerea, pot fi văzute ca şi exemple de programare concurentă, şi ambele prezintă aceleaşi probleme. În cazul unui singur procesor, problemele apar din cauza faptului că viteza relativă de execuţie a taskurilor nu poate fi predictată. Ea depinde de activităţile tuturor celorlalte procese care se execută în sistem, de modul în care sistemul de operare tratează întreruperile şi de strategiile de planificare ale sistemului de operare.

Termenul **multitasking** este folosit în două contexte:

- într-un **sistem multi-user.** Un sistem multi-user este un calculator care permite ca mai mulţi utilizatori să lucreze la simultan, de la terminale diferite. Fiecare utilizator îşi rulează propriul program independent de ceilalţi şi programele de aplicaţie nu interacţionează între ele;
- într-o **aplicaţie multitasking**. Mai multe programe (taskuri) lucrează împreună pentru a implementa o anumită aplicaţie (de exemplu, cea mai mare parte a produselor software pentru uz industrial, la care un task asigură interfaţa cu operatorul, al doilea colectează datele de la senzori, al treilea comunică cu un calculator central aflat la distanţă, alte taskuri implementează algoritmi numerici de conducere a proceselor etc).

Într-o aplicaţie software organizată sub forma mai multor taskuri (aplicaţie multitasking) programele (taskurile) trebuie să lucreze împreună pentru a executa global o anumită funcţie. Această cooperare necesită transfer de date între taskuri şi sincronizare între ele, prin mecanisme specifice de sincronizare şi comunicare (engl. **IPC** - *Interprocess Communication*). Astfel, concurenţa implică multe probleme de proiectare, incluzând comunicaţia între procese, partajarea şi competiţia pentru resurse, sincronizarea activităţilor proceselor multiple, alocarea timpului procesorului către procese prin mecanisme specifice de planificare (engl. *task scheduling*).

Planificarea taskurilor în sisteme în timp real

Planificarea (engl. *scheduling*) reprezintă un concept asociat sistemelor de operare, prin care se asigură mecanismele necesare pentru a face posibilă concurența.

Un **context de planificare**:

- include un *motor de execuție* denumit **planificator** (engl. *scheduler*) care execută o anumită politică (strategie) de planificare (engl. *scheduling algorithm*);
- dispune de un număr de *resurse* care sunt *planificate* în funcție de politica de planificare;
    - resursele includ (au) acțiuni care se execută cu o anumită prioritate în raport cu alte acțiuni.

Sistemul de operare furnizează motorul de execuție (planificatorul) şi politicile de planificare. Utilizatorul furnizează resursele planificabile, adică taskuri sau resurse care trebuie să fie (sau nu) protejate de accesul simultan prin mecanisme de tipul semafoarelor (de exemplu).

Pentru sistemele în timp real politicile de planificare trebuie să fie:

- **stabile**. O politică de planificare stabilă este aceea care, în cazul supraîncărcarii sistemului (*overload*), poate detecta (predicta) *a priori* care task (taskuri) nu îşi va putea îndeplini *deadline*-ul;
- **optimale**. Este acea politică de planificare care poate planifica orice set de taskuri ce poate fi planificat de oricare alt algoritm de planificare;
- **responsive**. Este acea strategie de planificare care permite ca evenimentele apărute să fie tratate fără întârziere;
- **robuste**. În acest caz execuția în timp a taskului nu este afectată de faptul că un alt task nu îşi îndeplineşte propriul cadru de timp (de exemplu, în cazul planificării de tip round-robin fără preempțiune, un singur task care nu funcționează cum trebuie va împiedica toate celelalte taskuri să se execute).

Politicile de planificare pot fi de tip:

- **fair**. Planifică în aşa fel încât toate taskurile progresează mai mult sau mai puțin uniform (în aceeaşi manieră);
    - exemple: executivul de tip ciclic, executiv ciclic de tip *time-triggered*, round-robin, round-robin preemptiv (cu diviziunea timpului);
- **bazate pe priorități** (engl. *priority-driven*). Sunt politici de tip unfair, deoarece anumite taskuri (cele cu prioritate mai mare) sunt planificate preferențial față de cele cu prioritate mai mică. Atunci când mai multe taskuri sunt gata de execuție, selecția se va face în funcție de priorități. În cazul unui planificator preeemptiv cu priorități, taskul care se executa va fi scos din execuție de un task cu prioritate mai mare care este gata de execuție. Planificatoarele bazate pe priorități sunt responsive atâta timp cât prioritatea taskului activat de evenimentul exterior este mai mare decât cea a taskului care se executa în momentul apariției evenimentului.

Din această cauză, în astfel de sisteme întreruperile au cea mai mare prioritate;

- exemple: *Rate Monotonic Scheduling* - **RMS**, *Deadline Monotonic Scheduling* - **DMS**, *Maximum Urgency First* - **MUF**, *Earliest Deadline First* - EDF, *Least Laxaty* – **LL.**

Există două concepte asociate planificării în sistemele în timp real sunt, şi anume:

- **importanța**. Se referă la valoarea pe care o are pentru performanța corectă a sistemului finalizarea unei acțiuni specifice;
- **urgența**. Urgența unei acțiuni se referă la cât de aproape este *deadline*-ul asociat unei anumite acțiuni, fără a lua în considerare importanța sa.

Cele mai multe planificatoare furnizează o singură posibilitate de ierarhizare a executării acțiunilor, şi anume prin intermediul **priorității** (cu ea se controlează atât importanța, cât şi urgența).

Principalele noțiuni asociate în cazul planificarii taskurilor sunt:

- **mecanismul de tip round-robin.** Planificarea de tip round-robin (*Figura 1.11*) se referă la o metodă de planificare a taskurilor care presupune organizarea unei mulțimi de taskuri într-o coadă circulară, taskul care se execută în mod curent fiind primul task din coadă. După ce îşi termină execuția (sau şi-o întrerupe în mod temporar) el va fi pus la sfârşitul cozii, deci în mod relativ va deveni cel mai puțin prioritar. Taskurile se execută secvențial, până la finalizare, într-o ordine de tip FIFO. De multe ori această manieră de planificare se utilizează în conjuncție cu un executiv de tip ciclic;
- **preempțiunea taskurilor**. Este o procedură prin care este posibilă întreruperea execuției unui task mai puțin prioritar aflat în execuție de către un task cu prioritate mai mare, aflat în stare gata de execuție.

În funcție de combinațiile acestor noțiuni în cadrul unei metode de planificare putem clasifica algoritmii de planificare ai taskurilor.

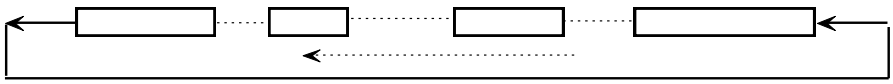

Figura 1.11: Planificare de tip round-robin [1]

În general, strategiile (politicile) de planificare a taskurilor se încadrează în următoarele patru categorii principale:

- **planificator preemptiv cu planificare simplă de tip round-robin**. În cazul planificatoarelor de tip round-robin cu diviziunea timpului (*Figura 1.12*), fiecărui task executabil îi este ataşată o cuantă fixă de timp, denumita *time slice*, în care taskul se poate executa. Punctele de preempțiune sunt marcate de întreruperile de ceas. Dacă taskul nu reuşeşte să-şi termine execuția între două puncte de preempțiune, contextul său va fi salvat, după care va fi plasat la coada listei de taskuri

executabile. Va fi refăcut apoi contextul următorului task din lista gata de execuție şi execuția acestuia va fi reluată (va fi lansat în execuție). Taskul care şi-a întrerupt temporar execuția devine în mod relativ cel mai puțin prioritar. După cum se observă în *Figura 1.12*, un task care o importanță mai mare (denumit aici *task de timp real - task TR*) va fi adăugat la coada de procese gata de execuție şi va aştepta să fie lansat în execuție după toate celelalte taskuri aflate în coada de aşteptare. În general, aceasta nu este o soluție acceptabilă pentru aplicații în timp-real;

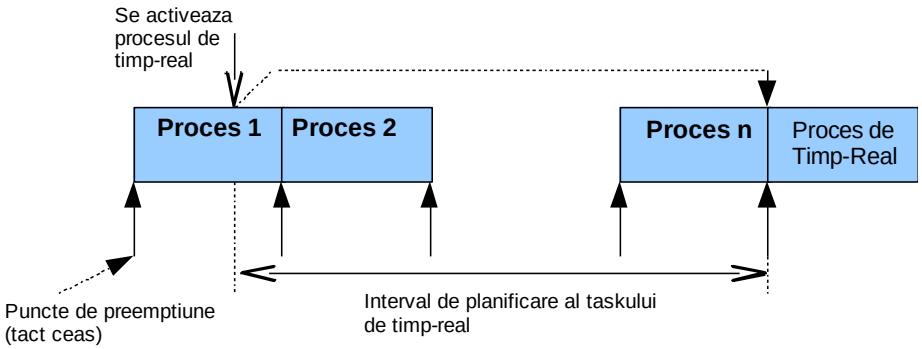

Figura 1.12: Planificator preemptiv cu planificare simplă de tip round-robin [1]

- **planificator non-preemptiv cu priorități**. În cazul unui planificator non-preemptiv, taskul (procesul) care se execută în mod curent nu poate fi întrerupt până când nu îşi întrerupe el singur execuția. Se poate utiliza totuşi un mecanism de planificare bazat pe priorități, dându-i-se taskului de timp real prioritatea cea mai mare, astfel încât un task TR aflat în stare gata de execuție (READY) să poată fi planificat imediat ce taskul curent se blochează sau îşi termină normal execuția (*Figura 1.13*). Taskul TR este adăugat la începutul cozii de taskuri (procese) gata de executie;

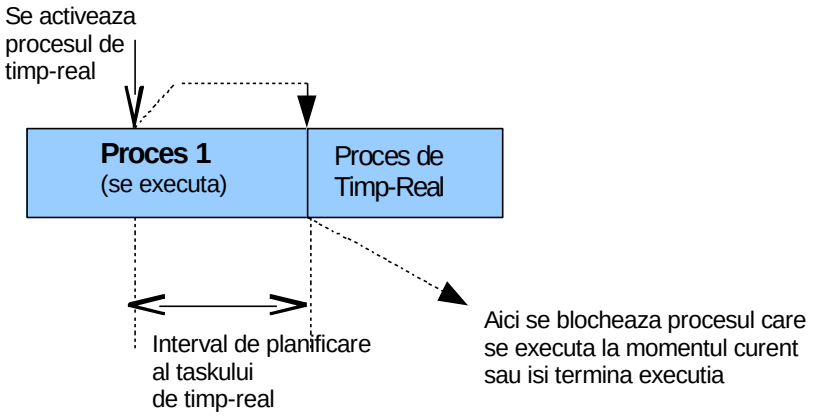

Figura 1.13: Planificator non-preemptiv cu priorități [1]

- **combinarea priorităților cu întreruperile de ceas**. În cazul planificatorului preemptiv cu priorități şi puncte de preempțiune, punctele de preempțiune apar la intervale de timp egale. Când apare un punct de preempțiune, taskul care se executa în mod curent va fi scos din execuție (engl. *preempted*) dacă în coada de procese (taskuri) READY aşteaptă un task cu prioritate mai mare (*Figura 1.14*). Deşi în acest caz întârzierea lansării în execuție a taskului TR poate fi de ordinul a câteva ms, soluția nu este optimă pentru aplicațiile în timp real critice.

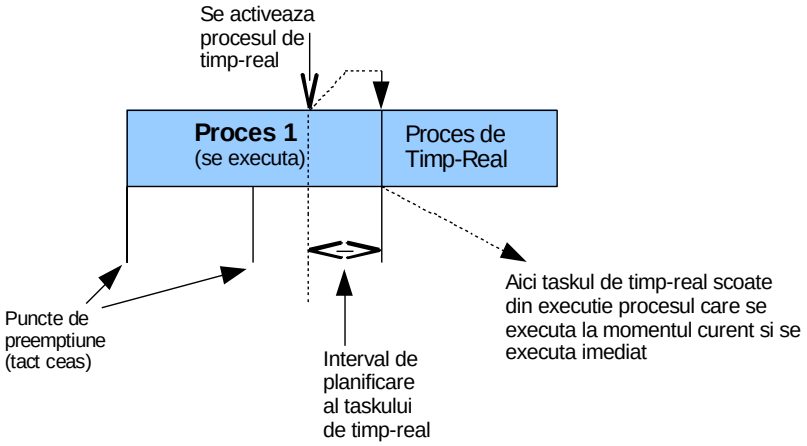

Figura 1.14: Planificator preemptiv cu priorități şi puncte de preempțiune [1]

Principale **politici de planificare de tip fair** sunt (a) executivul de tip ciclic, (b) executivul ciclic de tip time-triggered, (c) mecanismul de tip round-robin, (d) planificarea de tip round-robin cu diviziunea timpului.

Executivul de tip ciclic se se foloseşte atunci când există un număr fix de procese periodice şi poate fi dezvoltată o schemă completă de planificare, a cărei execuție repetată va forța procesele să se execute la momentul dorit. Un executiv de tip ciclic este de fapt o tabelă de apeluri de proceduri, fiecare procedură reprezentând o parte a codului corespunzător unui task. Executivul rulează setul de proceduri (taskuri), complet fiecare dintre ele, într-un ciclu infinit, lista de taskuri fiind stabilită la început, înainte de rularea aplicației.

Avantajul executivului de tip ciclic este simplitatea, având o predictibilitate mare a execuției. Este însă nonresponsive, instabil, non-optimal, non-robust, necesită ajustare, este bun pentru taskuri scurte (fiecare task trebuie să se execute într-un interval de timp scurt, pentru ca politica să fie de tip fair). Dacă un task e mai lung, programatorul trebuie să-l spargă în taskuri mai mici (scurte). Tabela de taskuri, care se repetă periodic, este denumită **frame major (***Figura 1.15***)**. Fiecare frame major (principal) este format dintr-un anumit număr de felii mai mici de timp, denumite **frame**-uri **minore**, care au durate fixe, în care sunt planificate taskurile pentru execuîie, cu ajutorul unui timer.

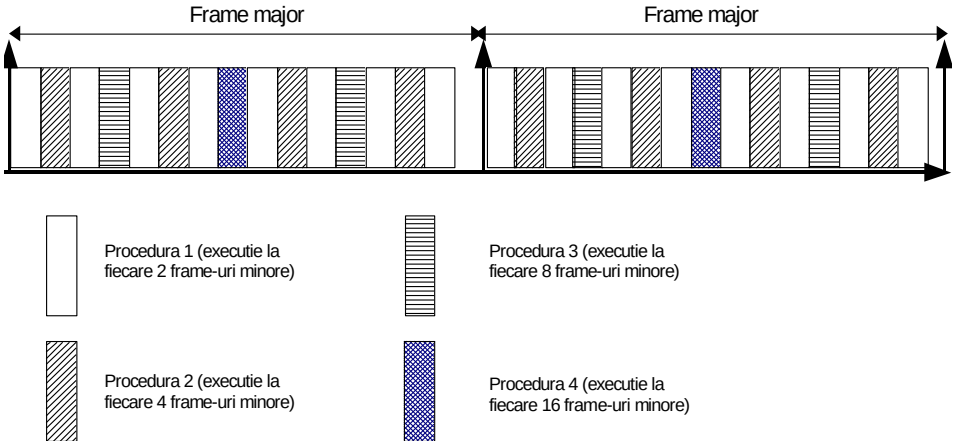

Figura 1.15: Planificarea taskurilor în cazul unui executiv de tip ciclic [1]

Operațiile sunt implementate sub forma unor proceduri, organizate într-o listă predefinită pentru fiecare frame major. La începutul fiecarui ciclu (frame) minor, taskul timer va apela fiecare procedură din listă. De exemplu, pentru 5 procese / taskuri se execută o listă de proceduri:

```
while(1){
    asteapta_intrerupere();
    procedura_pentru_a();
    procedura_pentru_b();
    procedura_pentru_c();
    asteapta_intrerupere();
    procedura_pentru_a();
    procedura_pentru_b();
    procedura_pentru_d();
    procedura_pentru_e();
    // etc ...
}
```

În figura 1.15 este exemplificat modul de lucru pentru 4 proceduri care se repetă într-un frame major, presupunând că durata unui frame minor este 10 ms. Se considera că cele 4 proceduri trebuie să se execute cu o frecventa de 50 Hz, 25 Hz, 12.5 Hz şi 6.25 Hz, respectiv cu perioada de 20 ms, 40 ms, 80 ms şi 160 ms.

Principalele **politici de planificare bazate pe priorități** (unfair) sunt următoarele:

- algoritmul **RMS** - *Rate Monotonic Scheduling*. Toate taskurile se presupun a fi periodice, cu *deadline*-ul la sfârşitul perioadei (*Figura 1.16*). Prioritățile se ataşează în momentul proiectării, astfel încât taskurile cu perioadele cele mai scurte au prioritatea cea mai mare (*Figura 1.17*). Avantajele sunt stabilitatea, optimalitatea, robustețea, dar este posibil să nu fie aplicabil în cazul unor sisteme complexe;

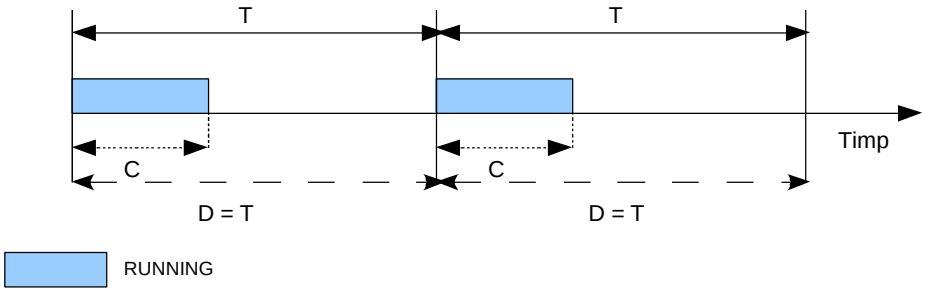

Figura 1.16: Planificarea de tip RMS [1]

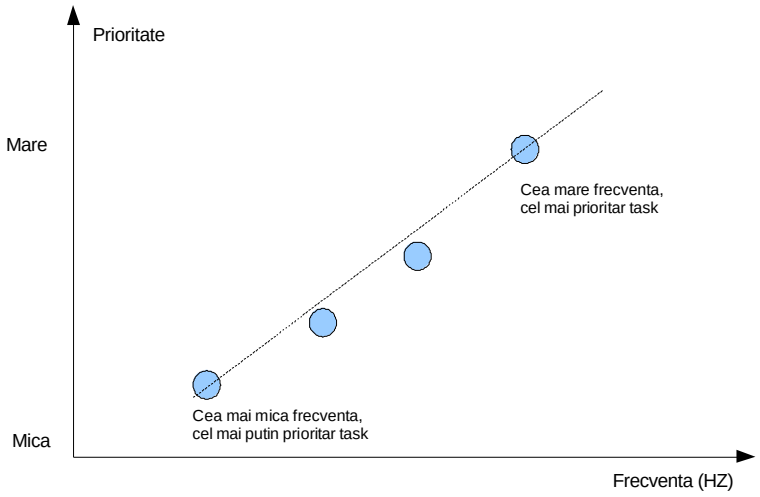

Figura 1.17: Asocierea priorităților în cazul planificarii de tip RMS [1]

- algoritmul **DMS** - *Deadline Monotonic Scheduling*. Funcționează la fel ca şi planificatorul RMS, cu excepția faptului că nu se presupune că *deadline*-ul taskului este la sfârşitul perioadei de execuție (*Figura 1.18*). Priorităţile sunt asociate în momentul proiectării aplicației, pe baza lungimii *deadline*-ului taskului. Avantaj - stabil, optimal, robust;

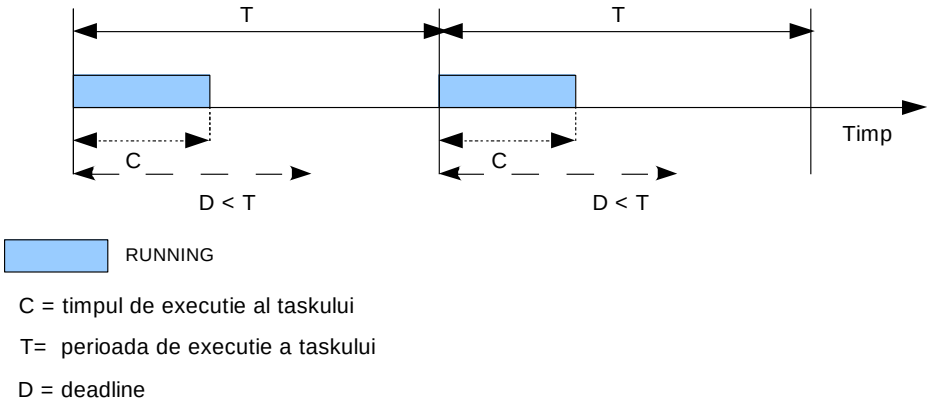

Figura 1.18: Planificarea de tip DMS [1]

- algoritmul **EDF** - *Earliest Deadline First*. Prioritățile sunt asociate în timpul execuției aplicației, atunci când taskurile sunt gata de execuție, pe baza aproprierii *deadline*-ului taskului (cu cât *deadline*-ul este mai aproape, cu atât prioritatea este mai mare).

Tranziții de stare ale proceselor / firelor de execuție

Sunt posibile diverse tranziții de stare ale proceselor sau firelor de execuție. Un set general de stări şi tranziții posibile sunt prezentate în *Figura 1.19,* în care:

- un task este în starea **instalat** (creat – **NEW**) dacă are create structurile de date corespunzătoare (bloc de control task, descriptor de task etc.);
- un task este în starea **gata de execuție** (**READY**) dacă sunt îndeplinite toate condiîiile şi poate prelua unitatea centrală. Planificatorul de taskuri ia în considerare doar taskurile aflate în această stare. Un task intră în starea gata de execuție dacă se află în execuție (la expirarea *time slice*-ului) sau dacă a fost scos din execuție de catre alt task mai prioritar. Dacă era în starea **BLOCKED** / **SUSPENDED**, taskul poate intra în starea **READY** dacă apare evenimentul care a determinat trecerea taskului în starea **SUSPENDED**;
- un task este în starea **în execuție** (**RUNNING**) atunci când este executat efectiv de către procesor. Un task poate intra în execuție atunci când este creat (dacă nici un alt task nu este gata de execuție) sau prin tranziție din starea gata de execuție (dacă este eligibil pentru a fi executat pe baza priorității sale sau în funcție de poziția sa în coada round-robin);
- un task este în starea **blocat** (**BLOCKED** / **SUSPENDED**) dacă aşteaptă finalizarea unei operații de I/O, dacă aşteaptă trecerea unui interval de timp (sleep Δt) sau dacă este blocat nedefinit (sleep(0), suspend etc.). Taskurile care aşteaptă pe o anumită resursă, deci nu sunt gata de execuție, se afla în starea suspendat sau blocat (pe condiție de timp sau pe eveniment).

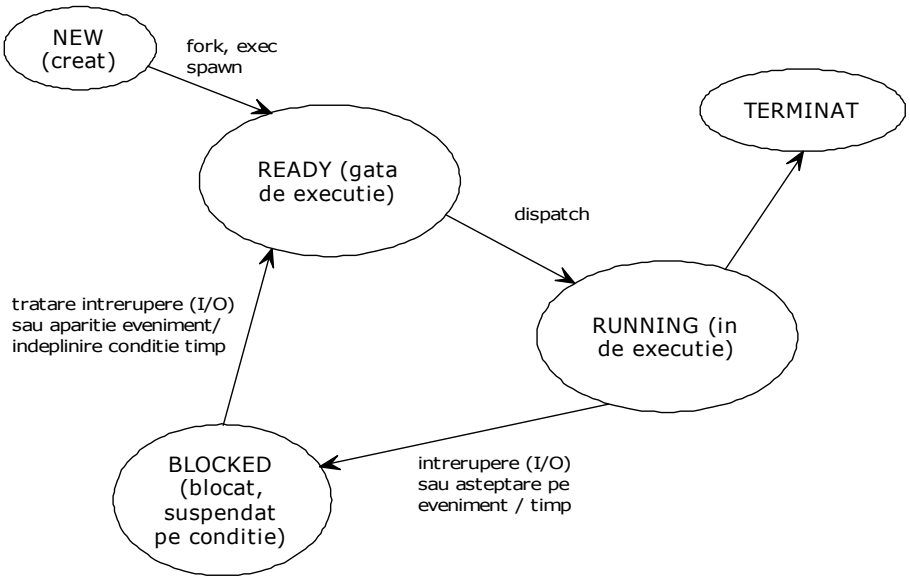

Figura 1.19: Diagrama generală a tranzițiilor de stare posibile ale unui proces sau fir de execuție [1]

Pentru stările **READY** şi **BLOCKED**, nucleul organizează cozi de aşteptare (coada de procese **READY** şi coada de procese **BLOCKED**, *Figura 1.20*).

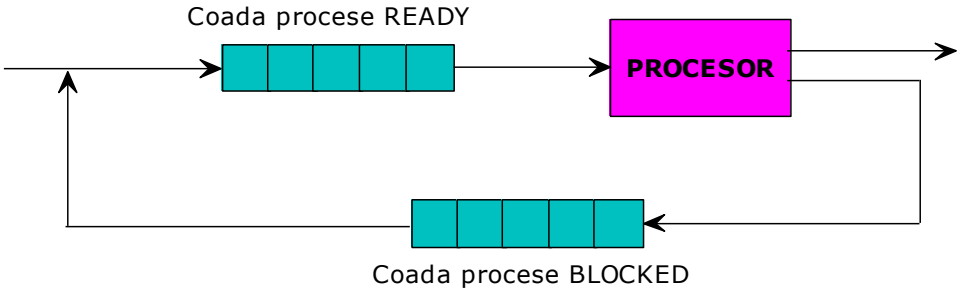

Figura 1.20: Organizarea cozilor de aşteptare pentru stările READY şi BLOCKED [1]

Un task blocat este scos din starea BLOCKED atunci când este îndeplinită condiția pentru care aşteaptă şi este trecut în coada READY.