Universitatea “Politehnica” din Bucureşti

Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei

OPTIMIZAREA GESTIUNII DE PROCESE CU CONSTRANGERI RT

Proiect de diplomă

prezentat ca cerinţă parţială pentru obţinerea titlului de

Inginer în domeniul *Calculatoare şi Tehnologia Informaţiei*

programul de studii de licenţă *Ingineria Informaţiei*

Conducător ştiinţific — *Conf. Dr. Ing. Ştefan STANCESCU*

Absolvent — *Constantin JERCA*

2016

Anexa 1

Universitatea "Politehnica" din Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei
Departamentul de Electronica Aplicata si Ingineria Informatiei

4.03.2016

Aprobat Director de Departament:
Prof. Dr. Ing. Sever Pasca

TEMA PROIECTULUI DE DIPLOMĂ
a studentului Jerca C. Constantin 441A

1. Titlul temei: OPTIMIZAREA GESTIUNII DE PROCESE CU CONSTRANGERI RT

2. Contribuţia practică, originală a studentului va consta în (*în afara* părţii de documentare):

Se va realiza un program de analiza a metodelor de gestiune de procese in timp real, concurente la mai multe resurse distincte. Metodele de gestiune, prin acordare de prioritati proceselor conduse, vor fi comparate pentru mai multe configuratii de procese concurente, ruland in diferite scenarii de functionare. Se vor trasa grafice de performante si se vor decide procedeele optime de stabilire a ordinii de executie individuala a proceselor, in functie de particularitatile rularii. Se va indica o procedura de alegere si adaptare a gestiunii de procese in timp real dupa caracteristicile configuratiei de procese. Sistemul de operare – Linux.

3. Proiectul se bazează pe cunoştinţe dobîndite în principal la următoarele discipline:

Sisteme de Operare, Inginerie Software, Programare Obiect-Orientata, Microcontrolere, Structuri de Date si Algoritmi.

4. Proprietatea intelectuală asupra proiectului aparţine: *UPB/Student*

5. Locul de desfăşurare a activităţii: *UPB*

6. Realizarea practică rămâne în proprietatea: *UPB*

7. Data eliberării temei: 15.12.2015

CONDUCĂTOR LUCRARE: *Conf. Dr.Ing. Stefan STANCESCU*

STUDENT: *Constantin JERCA*

Declaraţie de onestitate academică

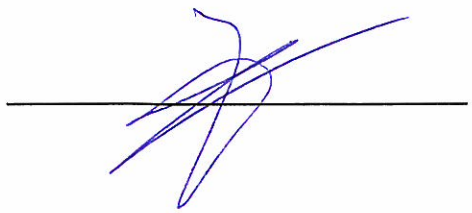
Prin prezenta declar că lucrarea cu titlul "*Optimizarea gestiunii de procese cu constrangeri RT*", prezentată în cadrul Facultăţii de Electronică, Telecomunicaţii şi Tehnologia Informaţiei a Universităţii "Politehnica" din Bucureşti ca cerinţă parţială pentru obţinerea titlului de *Inginer* în domeniul *Calculatoare şi Tehnologia Informaţiei*, programul de studii *Ingineria Informaţiei* este scrisă de mine şi nu a mai fost prezentată niciodată la o facultate sau instituţie de învăţământ superior din ţară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referinţe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi fac referinţă la sursă. Reformularea în cuvinte proprii a textelor scrise de către alţi autori face referinţă la sursă. Înţeleg că plagiatul constituie infracţiune şi se sancţionează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor şi măsurătorilor pe care le prezint ca fiind făcute de mine, precum şi metodele prin care au fost obţinute, sunt reale şi provin din respectivele simulări, experimente şi măsurători. Înţeleg că falsificarea datelor şi rezultatelor constituie fraudă şi se sancţionează conform regulamentelor în vigoare.

Bucureşti, *24.06.2016*

Absolvent *Constantin JERCA*



Cuprins

Lista figurilor

Lista tabelelor

Lista acronimelor

RT	-	În timp real (eng. real time)
RTS	-	Sistem în timp real (eng. real time system)
RTOS	-	Sistem de operare în timp real (eng. real time operating system)
TCB	-	Blocul de control al unei sarcini (eng. task control block)
CPU	-	Unitatea centrala de procesare (eng. central processing unit)
EDF	-	Algoritm de planificare Earliest Deadline First
LSTF	-	Algoritm de planificare Least Slack Time First
RM	-	Algoritm de planificare Rate Monolithic
DM	-	Algoritm de planificare Deadline Monolithic
RR	-	Algoritm de planificare Round Robin
FIFO	-	Algoritm de planificare First In First Out
FPS	-	Algoritm de planificare Fixed Priority
MUF	-	Algoritm de planificare Maximum Urgency
SMP	-	Multiprocesare simetrică
RCU	-	Mecanism sincron read-copy update din nucleul Linux
IRQ	-	Întrerupere
API	-	Interfaţa aplicaţiei cu utilizatorul (eng. aplication interface)

Introducere

Sistemele în timp real au evoluat de-a lungul ultimelor decenii, într-o manieră relativ calmă. Performanţa a crescut, se poate spune, în mod dramatic, dar principalele paradigme au fost destul de stabile de la mijlocul anilor `80. Acest lucru se schimbă în ultimii ani. Marea schimbare aparută în domeniul embedded este folosirea mai multor procesoare, ce duce în destul de multe cazuri la o reevaluare a metodelor vechi si a modului de a gândi în timp real.

Linux are o poziție puternică în toate tipurile de sisteme embedded, variind de la produse electronice de consum la dispozitive de rețea și o gamă largă de aplicații industriale, inclusiv sistemele legate de securitate. Resursele tehnologice potrivite pentru disponibilitate ridicată, în timp real, şi siguranţa sistemelor critice au fost în continuă expansiune și îmbunătățire. La baza acestei evoluții este disponibilitatea sistemelor de operare stabile cu proprietăți de încredere în timp real. Extinderea și îmbunătățirea proprietăților în timp real Open Source RTOS este un efort de cercetare și dezvoltare continuă.

Motivul alegerii acestei teme derivă din dorinţa de a aprofunda atât teoretic, cât şi practic domeniul sistemelor în timp real, domeniu de o importanţă majoră în ultimii ani, odată cu introducerea conceptului de Internet of Things. Posibilitatea ca sistemele electronice să răspundă în timp real la evenimentele din exterior fac ca sistemele în timp real să fie de o importanţă majoră în dezvoltarea tehnologiilor autonome.

Contribuţia studentului constă în realizarea unui program de analiză a metodelor de gestiune a preceselor în timp real, concurente la mai multe resurse distincte. Acest program va compara mai multe configuraţii de procese concurente, rulând în diferite scenarii de funcţionare. Se vor trasa grafice de performanţă si se vor decide procedeele optime de stabilire a ordinii de execuţie individuală a proceselor, în functie de particularităţile rularii. Se va indica o procedură de alegere şi adaptare a gestiunii de procese în timp real după caracteristicile configuraţiei de procese.

Rezultatele proiectului vor consta în analize de performanţă a principalelor politici de gestiune a proceselor în timp real ale kernelului RT Linux pe baza unor configuraţii de procese anterior stabilite. Deoarece constrângerile fundamentale ale sistemelor în timp real sunt date de timp, necesitatea unei gestiuni optime a proceselor este foarte mare sau chiar vitală în cazul sistemelor hard.

Capitolul 1.
Concepte de bază a sistemelor în timp real

Un sistem de calcul în timp real este capabil să execute foarte fiabil programe cu cerințe de sincronizare foarte specifice, lucru important pentru multe proiecte științifice și de inginerie. Componenta cheie necesară pentru a construi un sistem în timp real este un sistem de operareîn timp real; alte componente hardware și software de piese care alcătuiesc un întreg sistem în timp real sunt discutate în secțiunea următoare.

Cu toate că componenta principală necesară pentru crearea unui sistem în timp real este RTOS, sunt necesare mai multe piese de software și hardware pentru a construi un sistem în timp real de la început până la sfârșit.

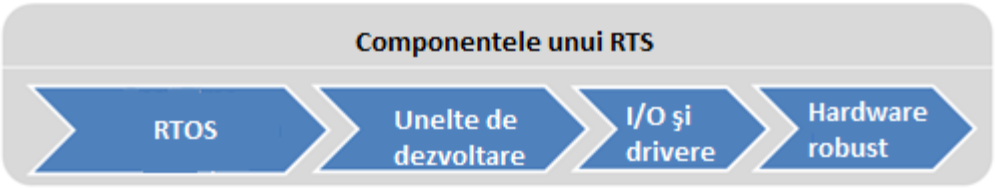


Figura 1.1 Componentele unui sistem în timp real [1]

Un sistem în timp real este un sistem care trebuie să răspundă la stimulii de intrare generaţi în exterior într-o perioadă finită și specificată. El controlează un mediu prin primirea de date, procesarea acestora, precum și returnarea rezultatelor suficient de rapid pentru a afecta mediul în acel moment.Buna funcţionare a sistemului se bazează atât pe generarea de răspunsuri corecte, cât şi pe producerea acestora în anumite intervale de timp stabilite (critice în timp).

O altă definitie a sistemelor în timp real este că sunt sistemele care trebuie să proceseze informații și să producă un răspuns într-un anumit timp, altfel risca consecințe grave.[3]Într-un sistem cu constrângeri de timp real contează dacăacțiunea corectă sau răspunsul corect au fost date până la un anumit termen: acesta este fie până la termenul limită stabilit sau răspunsul este inutil.

1.1 Caracteristici ale sistemelor în timp real

Un sistem în timp real are anumite caracteristici speciale fie inerente fie impuse. Desigur nu toate RTS prezintă toate aceste caracteristici, totuși orice limbaj de programare sau sistem de operare folosit pentru programarea efectivă a unui RTS trebuie să aibă facilitați care să suporte aceste caracteristici.

a. Dimensiunea si complexitatea

Dimensiunea si complexitatea RTS variază. În cazul sistemelor de mică dimensiune si complexitate scazută, ele pot fi întreţinute de către un singur utilizator. Sistemele complexe

presupun o dezvoltare ierarhizată pe module de catre mai multe echipe. RTS complexe pot ajunge pana la 20 de milioane de linii de cod - asamblare sau C (Statia Spatială Freedom).[2]

b. Controlul concomitent al componentelor separate ale sistemului

Dispozitivele operează în paralel în lumea reală. Pentru a modela acest paralelism este mai bine sa creăm entitaţi concurente în program.

c. Facilitaţi pentru interacţiunea cu componente hardware specifice

Un RTS trebuie să dispună de modalitati de comunicare sau interacţiune cu dispozitivele hardware adiacente acestora.

d. Amestec hardware / software

Sistemele în timp real sunt în general formate atât din module implementate software (sisteme embedded), cât şi din module implementate exclusiv hardware (System on Chip).

e. Fiabilitate şi siguranţă extremă

RTS controlează în mod obişnuit mediul în care operează; lipsa de control poate duce la pierderea de vieți omenești, deteriorarea mediului sau a pierderilor economice grave.

f. Timp garantat de răspuns

Trebuie să fim capabili de a prezice cu încredere timpii de răspuns în cel mai rau caz pentru sisteme. Eficienţa este importantă, dar predictibilitatea este esenţială. Un răspuns bun dar dat prea târziu este echivalentul unui răspuns rău.

1.2 Clasificarea sistemelor în timp real

Principalul criteriu de clasificare a sistemelor în timp real este satisfacerea limitei timpului de răspuns. Depăşirea deadline-ului impus unui proces poate avea diverse consecinţe. În funcţie de aceste consecinţe sistemele în timp real au fost clasificate în diferite categorii.

a. Sisteme de tip hard

Sistemele în timp real de tip hard sunt sisteme unde este imperativ ca răspunsurile să apară în limita de timp predefinită. Neîndeplinirea acestor constrângeri de timp poate duce la efecte catastrofice. Aceste sisteme sunt considerate sisteme de siguranţă critică. Ele sunt folosite atunci când nerespectarea deadline-ului duce la căderi de sistem sau pagube importante.

Ex: sisteme de control al traficului, sisteme aeronautice, sisteme medicale, etc.

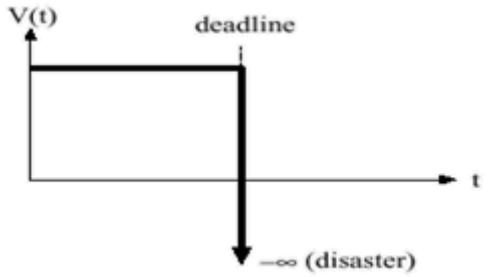


Figura 1.2 Reprezentarea limitei de timp pentru sisteme hard[2]

b. Sisteme de tip soft

Sistemele în timp real de tip soft sunt sistemele unde limitele de timp sunt importante, insă dacăsunt ratate câteva, sistemul continuă să functioneze. Ratarea deadline-urilor duce la scăderea performanţei sistemelor si nu la căderea lor.

Ex: sisteme de achiziţii de date, sisteme multimedia.

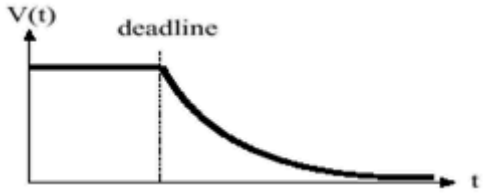


Figura 1.3 Reprezentarea limitei de timp pentru sisteme soft [2]

c. Sisteme de tip firm

Sistemele în timp real de tip firm sunt sisteme care au in componenţă atât subsisteme de tip hard, cât şi subsisteme de tip soft. Ratarea limitei de timp la sistemele firm nu este de dorit deoarece există o funcţie cost care creşte atunci când răspunsurile nu sunt date conform deadline-urilor. Dacătimpul de răspuns depăşeşte o anumită limită atunci răspunsul nu mai este valabil.

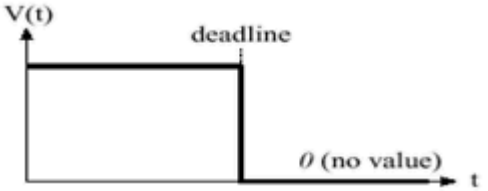


Figura 1.4 Reprezentarea limitei de timp pentru sisteme firm [2]

Sincronizarea între procesele externe şi acţiunile interne (task-uri) ale unui sistem poate fi definită în funcţie de trecerea timpului, sau timpul actual, caz în care sistemul este bazat pe clock sau definită în funcţie de eveniment, caz în care sistemul este bazat pe evenimente. Dacă relaţia între acţiunile din sistem şi procesele externe este definită mai puţin strict atunci sistemul este interactiv.

În cazul **sistemelor bazate pe clock** finalizarea operațiunilor în termenul specificat depinde denumărul de operațiuni care urmează să fie efectuate şi viteza computerului.Sincronizarea este de

obicei obținuta prin adăugarea unui ceas la calculatorul sistem și folosind un semnal de la acest ceas pentru a întrerupe operarea computerului la un interval de timp fix predeterminat. Procesele la aceste sisteme sunt ciclice si periodice

În cazul **sistemelor bazate pe evenimente** acțiunile urmează să fie efectuate nu în anumite perioade sau intervale de timp, dar, ca răspuns la un eveniment. Sistemul trebuie să răspundă într-un termen maxim acordat unui anumit eveniment.Evenimentele au loc la intervale non-deterministe și sarcinile bazate pe evenimente sunt denumite sarcini periodice.

Sistemele interactivereprezintă cea mai mare clasă de RTS-uri (servere bancare automate, sisteme de rezervare pentru hoteluri, companii aeriene și închiriere de mașini, etc.). [3]

Capitolul 2.
Sisteme de operare în timp real

Un sistem de operare în timp real este un sistem de operare cu caracteristici speciale care îl fac potrivit pentru construirea de aplicații de calcul în timp real.Un RTOS bun este unul care permite un comportament mărginit (previzibil) în toate cazurile de încărcare a sistemului. Rețineți totuși că RTOS în sine nu poate garanta corectitudinea sistemului, ci doar este o tehnologie care sa permită dezvoltarea, adică furnizează programatorului facilități care pot fi folosite pentru a construi o aplicatie corectă. Viteza, deși importantă pentru îndeplinirea cerințelor generale, prin ea însăși nu poate să îndeplinească cerințele pentru un RTOS.

2.1 Principiile sistemelor de operare în timp real

Următoarele sunt cerințe importante pe care un sistem de operare trebuie să le îndeplinească pentru a fi considerat un RTOS în sens contemporan:

- Sistemul de operare trebuie să fie multithread şi preemptiv, de exemplu să poată să proceseze mai multe fire de execuţie şi să fie capabil să anticipeze sarcini dacă este necesar.
- Sistemul de operare trebuie să suportediferite prioritaţi ale sarcinilor şi firelor de execuţie.
- Trebuie să existe un sistem de moștenire de prioritate. Moștenirea prioritară este un mecanism prin care se asigură că sarcinile prioritare inferioare nu pot împiedica executarea sarcinilor cu prioritate mai mare.
- Sistemul de operare trebuie să sprijine diferite tipuri de mecanisme de sincronizare thread / task.
- Pentru un răspuns previzibil:
    - Timpul pentru fiecare sistem de apel în funcție de executare ar trebui să fie previzibil și independent de numărul de obiecte din sistem.
    - Porțiuni non preemptibile ale funcțiilor de kernel necesare pentru sincronizare interproces și comunicare sunt extrem de optimizate, pe termen scurt și deterministe.
    - Porțiunile non-preemptibile ale rutinei de gestionare a întreruperilor sunt păstrate mici și deterministe.
    - Manipulatoarele (eng. handlers) de întreruperi sunt programate și executate în funcţie deprioritățile corespunzătoare.
    - Timpul maxim în care întreruperile sunt mascate de sistemul de operare și de drivere de dispozitiv trebuie să fie cunoscute.
    - Timpul maxim folosit de driverele de dispozitiv pentru a procesa o întrerupere, precum și informațiile specifice IRQ referitoare la aceste drivere de dispozitiv, trebuie să fie cunoscute.
    - Latența de întrerupere (timpul de întrerupere pentru a rula sarcini) trebuie să fie previzibil și compatibil cu cerințele de aplicare.

- Pentru un răspuns rapid:
    - Overheadul de run-time este scăzut prin reducerea schimbarilor de context inutile.
    - Temporizările importante, cum ar fi timpul de schimbare de context, latența de întrerupere, latența semafoarelor de obținere / de eliberare trebuie să fie minime.

Pe un sistem cu un singur procesor, doar o singură sarcină (task) poate rula în orice moment dat, prin urmare, celelalte sarcini trebuie să fie într-o altă stare decât cea de rulare. Numărul celorlalte stări, numele dat acestor stări și căile de tranziție între diferitele stări variază în funcție de sistemul de operare. O diagramă tipică de stari este prezentată în figura 2.1 și diferitele stări sunt descrise mai jos.

Stările in care se pot gasii sarcinile (task-urile):

- Starea „**în execuţie**”
  Aceasta este task-ul care are controlul asupra CPU. Acesta va fi în mod normal, task-ul care are cea mai mare prioritate actuală a task-urilor care sunt gata pentru a rula.
- Starea „**gatade execuţie**”
  Pot exista mai multe task-uri în această stare. Atributele task-ului și resursele necesare pentru a rula trebuie să fie disponibile pentru ca acesta să fie pus în starea "gata".
- Starea „**blocat**”
  Executarea task-urilor plasate în aceasta stare a fost suspendată, deoarece task-ul necesită unele resurse care nu sunt disponibile sau pentru că task-ul este în așteptarea unor semnale de intrare sau a terminarii timpului.
- Starea „**nou**”
  Sistemul de operare este conștient de existența acestui task, dar lui nu i-a fost alocată o prioritate și un context și nu a fost inclusă în lista task-urilor planificabile.
- Starea „**terminat**”
  Sistemul de operare nu a fost încă adus la cunoștință de existența acestui task, deși poate fi rezident în memoria calculatorului.

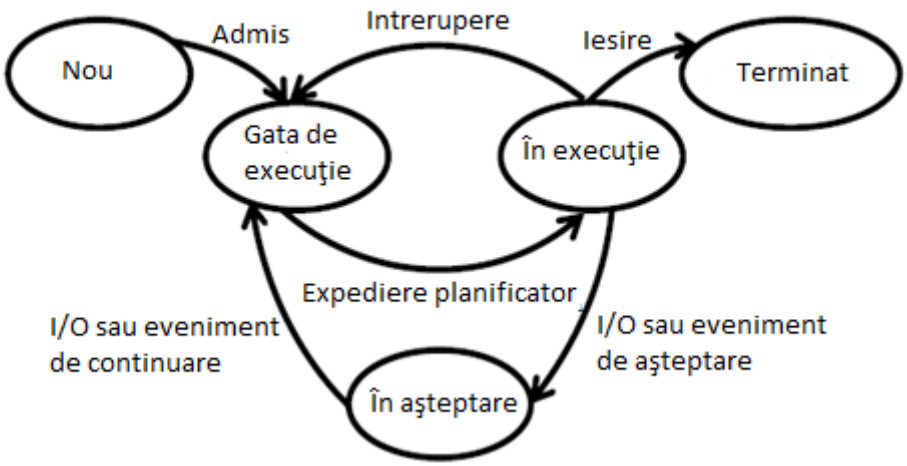


Figura 2.1 Diferitele stări in care un task poate fi în timpul ciclului său de viață de execuție în cadrul unui RTOS Task State Transitions [3]

RTOS-urile furnizează funcții pentru a crea, inițializa și activa task-uri noi. Ele oferă funcții pentru a aduna informații cu privire la activitățile existente în sistem, pentru numirea task-ului, verificarea stării unuitask dat, opțiuni pentru executarea task-ului, cum ar fi utilizarea de co-procesor, modele specifice de memorie, precum și stergerea task-ului. Ștergerea necesită adesea măsuri speciale de precauție, în special în ceea ce privește semafoarele, pentru task-urile de memorie partajată.

De fiecare dată când un task este comutat, contextul său de execuție, reprezentat de conținutul contorului de program, stivă și registre, este salvat de către sistemul de operare într-o structură de date speciala, numita **bloc de control al task-ului**, astfel încât task-ul poate fi reluat data viitoare când este programat. În mod similar, contextul trebuie să fie restaurat din blocul de control al task-ului atunci când starea task-ului este de executare.

Caracteristicile majore care diferențiază RTOS de celelalte sisteme de operare sunt următoarele:

- Implementarea explicită a unei politici de planificare sub forma unui modul **planificator** (eng. scheduler). Planificatorul este el însăși o sarcină care se execută de fiecare dată când o întrerupere internă sau externă se produce și calculează decizia privind efectuarea tranzițiilor de stare pentru fiecare taskdin sistem care a fost generat și încă nu a fost încheiat. Se calculează această decizie bazată pe nivelul curent de prioritate a task-urilor si disponibilitatea diferitelor resurse ale sistemului. Planificatorul calculează, de asemenea, nivelurile de prioritate actuale ale sarcinilor pe baza mai multor factori, cum ar fi termene, dependențe de calcul, timpii de așteptare.
- În baza deciziilor planificatorului, **dispecerul**efectuează de fapttranziția de stare a task-urilor prin:
  a. salvarea stării computationale sau contextul task-ului în execuție din mediul hardware.
  b. permițând următoruluitask să se execute prin încărcarea contextului procesului în mediul hardware.

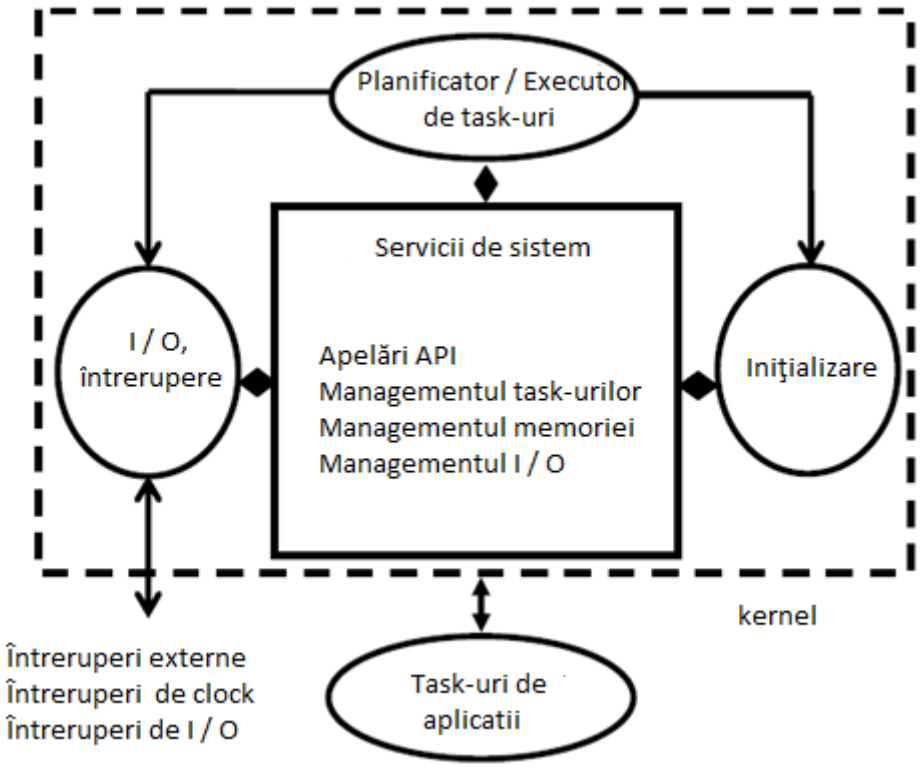


Figura 2.2 Structura unui kernel RTOS [3]

2.2 Exemple de sisteme de operare în timp real

Conform unui studiu realizat in 2014 [4] următoarele RTOS sunt printre primele 10 sisteme de operare în timp real utilizate pe piața sistemelor integrate:

- Green Hills Software INTEGRITY
- Wind River VxWorks
- QNX Neutrino
- FreeRTOS
- Micrium µC/OS-II, III
- Windows CE
- TI-RTOS Kernel (previously called DSP/BIOS)
- RTEMS open source RTOS designed for embedded systems, mainly used for missile and space probes control
- RT-Preempted Linux

Capitolul 3.
Planificatorul de procese

Pentru un set dat de lucru, problema generală de planificare solicită o ordine în funcţie de care procesele urmează să fie executate astfel încât diferitele constrângeri sunt îndeplinite. De obicei, un proces în timp real se caracterizează prin timpul de execuţie, timpul de gata, limita de timp şi cerinţele privind resursele. Executarea unui proces poate sau nu poate fi întreruptă (planificarea preemptivă sau non-preemptivă). Peste setul de procese, există o relaţie de prioritate care limitează ordinea de execuţie. Executarea unui proces nu poate începe până la executarea tuturor predecesorilor săi (în funcţie de relaţia de prioritate) este finalizată. Sistemul pe care procesele urmează să fie executate este caracterizat prin cantitatea de resurse disponibile.

Practic, problema de planificare este de a stabili un program pentru executarea proceselor, astfel încât toate acestea sunt finalizate înainte de termenul limită de ansamblu.

Având în vedere un sistem în timp real, abordarea de planificare corespunzătoare ar trebui să fie proiectată în funcţie de proprietăţile sistemului şi sarcinile care apar în ea.

3.1 Obiective şi cerinţe de bază

Următoarele obiective trebuie să fie luate în considerare în planificarea unui sistem în timp real:

- Satisfacerea constrângerilor de temporizare ale sistemului.
- Prevenirea accesului simultan la resurse şi dispozitive partajate.
- Atingerea unui grad ridicat de utilizare, respectându-se constrângerile de temporizare ale sistemului; însă acest lucru nu este un factor principal.
- Reducerea costurilor de context, switch-uri cauzate de preemţiune.
- Reducerea costurilor de comunicare în sisteme distribuite în timp real; ar trebui să găsim calea optimă pentru a descompune aplicaţia în timp real în porţii mai mici, în scopul de a avea costuri minime de comunicare între porţiunile de ajutor reciproc (fiecare porţiune este alocată unui calculator).

În plus, următoarele elemente sunt dorite în sisteme avansate în timp real:

- Având în vedere o combinaţie de activităţi în timp real hard, firm şi soft, ceea ce implică posibilitatea de a aplica politici de planificare dinamice care respectă criteriile de optimalitate.
- Sarcina de planificare pentru un sistem în timp real al cărui comportament este în mod dinamic adaptiv, reconfigurabil, reflexiv şi inteligent.
- Acoperirind fiabilitatea, securitatea şi siguranţa.

Cerinţele de bază ale unui planificator într-un RTOS sunt:

- **Multitasking și preemptibil**: Pentru a suporta multitasking în aplicațiile de timp real un RTOS ar trebui să fie multitasking și preemptibil. Planificatorul ar trebui să fie în măsură să preîntâmpine orice task în sistem și să dea resursele necesare pentru task-ul care are nevoie de ele.
- **Identificarea dinamică a deadline-ului**: Un RTOS ar trebui să fie în măsură să identifice în mod dinamic task-ul cu cel mai vechi deadline. Pentru a se ocupa de termenele limită, planificatorul poate converti termenele limită la niveluri de prioritate, care sunt utilizate pentru alocarea resurselor. Această tehnică este folosită dtorită lipsei unei soluții mai bune și dă eroari mai puține.
- **Sincronizare predictibilă**: sunt necesare mecanisme predictibile de comunicare inter-task și de sincronizare. Sincronizarea predictibilă necesită un compromis. Abilitatea de a bloca sau debloca resursele e modalitatea de a menţine integritatea datelor.
- **Niveluri suficiente de prioritate**: Atunci când se utilizează planificarea task-urilor prioritizate, RTOS trebuie să aibă un număr suficient de nivele de prioritate pentru implementarea efectivă. Inversiunea priorităţilor are loc atunci când un task cu prioritate mai mare trebuie să aştepte un task cu prioritate mai mică pentru a elibera o resursă și la rândul său, task-ul prioritar inferior este în aşteptarea unui task cu prioritate medie. Două rezolvări care se ocupă cu inversiunea priorităţilor, și anume moştenirea prioritară și protocolul de moştenire a priorităţii, au nevoie de un nivel de prioritati suficient.

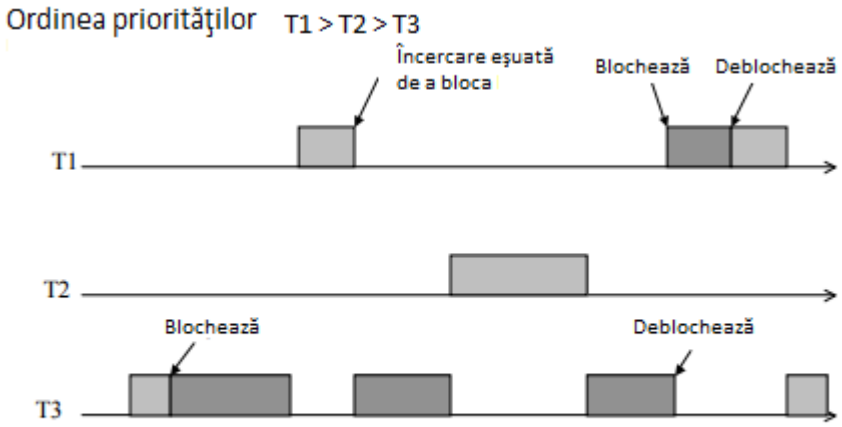


Figura 3.1.1 Problema inversării de prioritate:
Procesul T2 cauzează întârzierea procesului T1 cu o prioritate mai mare [9]

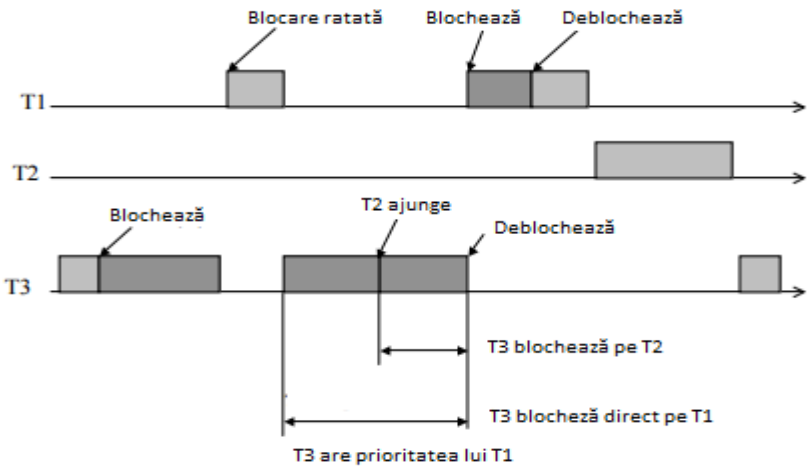


Figura 3.1.2 Problema inversării de prioritate:
Protocolul de moştenire a priorităţii [9]

- **Latențele predefinite**: Timing-ul apelurilor de sistem trebuie să fie definit utilizând următoarele specificații:
  - Latența de schimbare a task-urilor sau timpul pentru a salva contextul unui task ce se execută în prezent și de a comuta la altul.
  - Latență de întrerupere sau timpul scurs între executarea ultimei instrucțiuni a task-ului întrerupt și prima instrucțiune de tratare a întreruperii.
  - Latență de întrerupere de expediere sau timpul pentru a comuta de la ultima instrucțiune din rutina de tratare a întreruperii la următoarul task programat să ruleze.

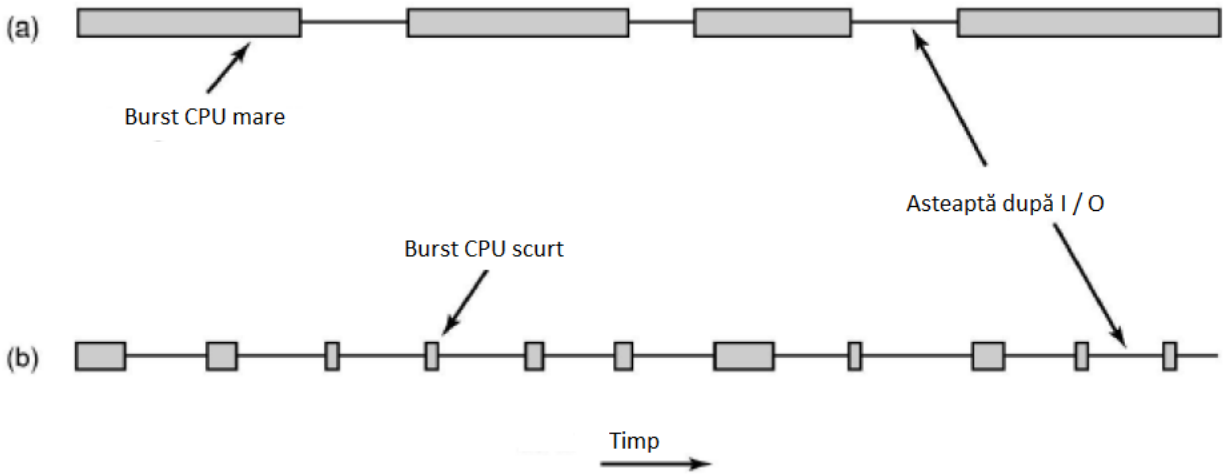


Figura 3.2 Alternarea perioadelor de utilizare a CPU cu cele de aşteptare a datelor de la un dispozitiv de I/O pentru:
(a) un proces computaţional,
(b) un proces de I/O. [8]

3.2 Procese în timp real

Fiecare proces ce are loc într-un sistem în timp real are unele proprietăți de timp. Aceste proprietăți ar trebui să fie luate în considerare atunci când dorim gestionarea proceselor pe un sistem în timp real. Proprietățile de timp ale unui anumit proces se referă la următoarele elemente:

- **Timpul de lansare**: momentul în care procesul este gata pentru prelucrare.
- **Termenul limită (deadline)**: timpul până când executarea procesului ar trebui să fie finalizată, după ce procesul este lansat.
- **Intârzierea minimă**: valoarea minimă de timp care trebuie să treacă înainte de executarea procesului, după ce procesul a fost lansat.
- **Intârziere maximă**: valoarea maximă de timp permisă, care trece înainte de executarea procesului, după ce procesul a fost lansat.
- **Timpul de execuţie în cel mai rău caz**: timpul maxim necesar pentru a finaliza procesul, după ce procesul a fost lansat. Timpul de execuție în cel mai rău caz este cunoscut, de asemenea, ca si timpul de răspuns in cel mai rău caz.
- **Timpul de execuţie**: timpul necesar fără întreruperi, pentru a finaliza procesul, după ce procesul a fost lansat.
- **Prioritatea**: urgența relativă a procesului.

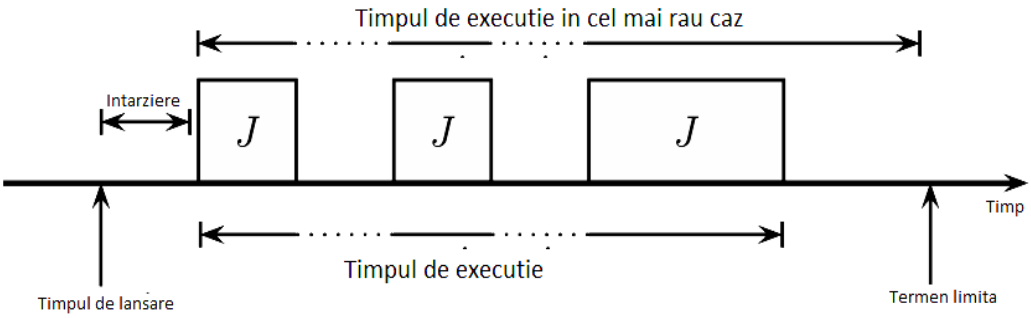


Figura 3.3 Proprietățile de timp ale unui proces în timp real

Procesele în timp real pot fi clasificate în funcţie de periodicitate, anticipare, priorităţi sau dependenţa de alte procese. În continuare vom defini diferite tipuri de procese în funcţie de criterile de clasificare prezentate anterior.

Procesele **periodice** sunt procese în timp real, care sunt activate (lansate) în mod regulat la rate fixe (perioade). În mod normal, procesele periodice au constrângeri de timp care indică faptul că instanțele din ele trebuie să se execute o dată pe perioada P.

Procesele **aperiodice** sunt procese în timp real, care sunt activate neregulat la un anumit grad de limitare necunoscut. Constrângerea de timp este, de obicei, un termen limită (deadline).

Procesele **sporadice** sunt procese în timp real, care sunt activate neregulat, cu un grad de limitare cunoscut. Gradul de limitare se caracterizează printr-o perioadă minimă de intre-sosire, adică un interval minim de timp între două lansări succesive. Constrângerea de timp este, de obicei, un timp limită (deadline).

În unii algoritmi de planificare în timp real, un proces poate fi **preemptiv** dacă un alt proces de prioritate mai mare devine gata de execuţie. În schimb, executarea unui proces de bază **non-preemptiv** ar trebui să fie finalizată fără întrerupere, odată ce acesta este pornit.

În planificarea de prioritate determinată, o prioritate este alocată fiecărui proces. Atribuirea de priorități se poate face static sau dinamic în timp ce sistemul este pornit.

Având în vedere un sistem în timp real, un proces care urmează să înceapă executarea poate solicita să primească informațiile furnizate de către un alt proces al sistemului. Prin urmare, executarea unui proces trebuie începută după terminarea execuției celuilalt proces. Acesta este conceptul de dependență. Procesele **dependente** folosesc memoria partajată sau datele de comunicare pentru a transfera informațiile generate de un alt proces. În timp ce decidem despre planificarea unui sistem în timp real care conține unele procese dependente, ar trebui să luam în considerare ordinea timpului de pornire și de terminare a proceselor.

3.3 Algoritmi de planificare a proceselor în timp real

Tehnicile de planificare în timp real pot fi împărțite în mare măsură în două categorii: static și dinamic. Algoritmi statici atribuie priorități la timpul de proiectare (înainte ca procesele să fie lansate în execuţie). Toate prioritățile atribuite rămân constante pe toată durata de viață a unui proces. Algoritmi dinamici atribuie priorități în timpul rulării, pe baza unor parametri de execuție a proceselor.

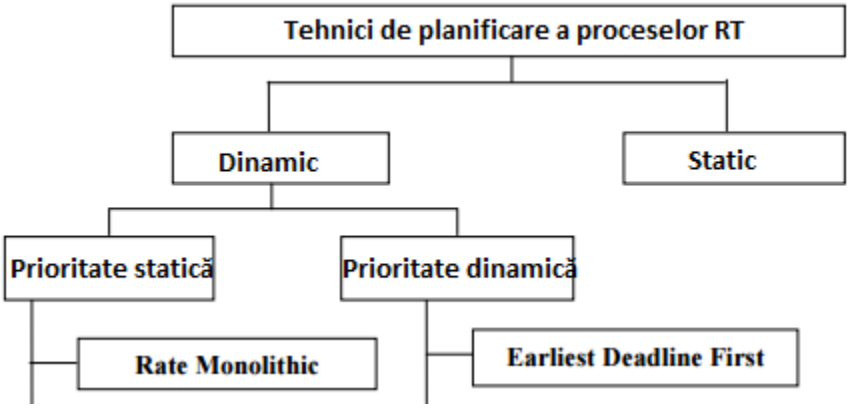


Figura 3.4 Clasificarea algoritmilor de planificare a proceselor RT

Planificarea dinamică poate fi, fie cu prioritate statică, fie cu prioritate dinamică. Rate Monolithic, Deadline Monolithic şi Fixed Priority sunt exemple de planificare dinamică, cu prioritate statică. Earliest Deadline First, Feedback EDF, Maximum Urgency și Least Slack Time First sunt exemple de planificare dinamică, cu prioritate dinamică. Algoritmi EDF și LST sunt optimi în condițiile în care task-urile sunt preemptive, există doar un singur procesor, iar procesorul nu este supraîncărcat, dar performanța lor scade exponențial în cazul în care sistemul devine puțin prea încărcat.

În continuare vom da exemple de cativa algoritmi de planificare a proceselor RT:

Rate Monotonic

Planificatorul RM acordă o mai mare prioritate proceselor cu perioade mai mici. Prin urmare, ori de câte ori un set de procese sunt programate cu programatorul RM, procesele sunt întotdeauna prioritizate în aceeași ordine. Din acest motiv, RM este considerat un algoritm de planificare dinamic cu prioritate statică. RM e un algoritm stabil, simplu de înţeles şi usor de implementat. Dezavantajele acestui algoritm sunt utilizarea scazută a CPU-ului si faptul că se ocupă doar de procese independente.

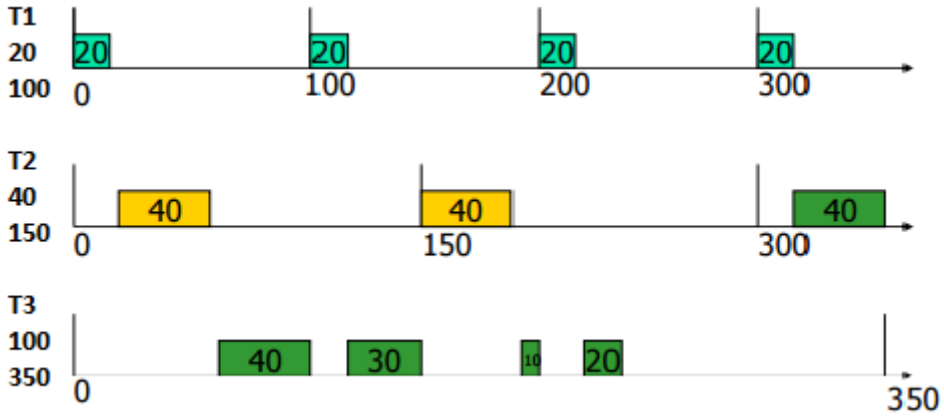


Figura 3.5 Exemplu de planificare cu algoritmul RM [6]

Earliest Deadline First

Algoritmul EDF acordă prioritate proceselor cu cel mai apropiat deadline. În timp ce sistemul execută, deadline-ul fiecarui proces se va schimba în raport cu timpul curent al sistemului. Deşi este adevărat că fiecare dintre deadline-uri se va schimba cu o cantitate identică de timp în raport unul cu altul, EDF este un algoritm de planificare dinamică cu prioritate dinamică. El este simplu si lucrează bine în teorie. Este un algoritm optimal ce are cea mai bună utilizare a CPU-ului, insă este greu de implementat si instabil (daca un proces nu işi atinge deadline-ul sistemul devine instabil si orice proces ulterior poate sa esueze).

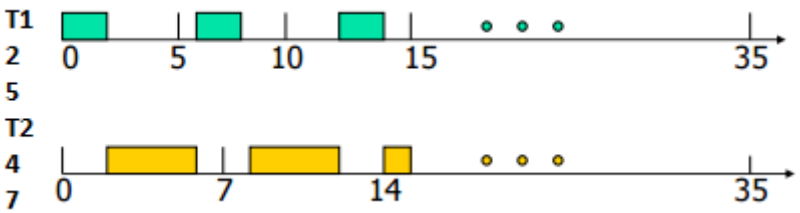


Figura 3.6 Exemplu de planificare cu algoritmul EDF [7]

Least Slack Time First

Algoritmul LSTF acordă prioritate procesului cu cea mai mică cantitate de „slack”. „Slack” este definit ca deadline-ul minus munca rămasă. Un proces cu un număr mare de „slack” va avea o prioritate mai mică în raport cu un proces cu un număr mic de „slack”. Procesul cu un număr mic pentru un deadline și un număr mare pentru cantitatea rămasă de muncă va avea cea mai mare prioritate. Acest algoritm poate fi implementat cu un minim al timpului de angajament al unui proces pentru a preveni o oscilație rapidă între procese atunci când preemțiunea este inclusă. Când toate procesele din volumul de lucru au „slack-ul” diferit doar prin minimul timpului de angajament al unui proces atunci planificatorul LSTF este esenţial identic cu cel RR. [3]

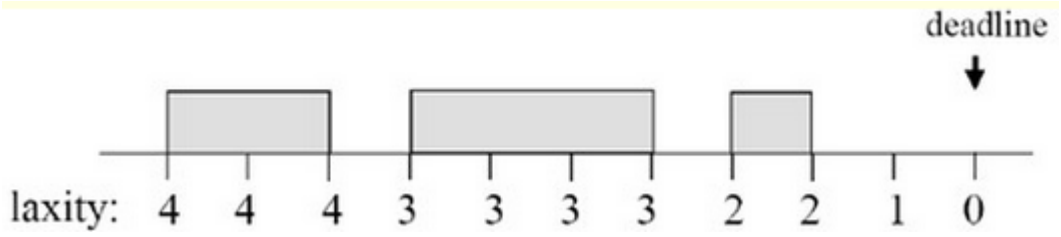


Figura 3.7 Exemplu de planificare cu algoritmul LSTF [3]

Round Robin

Algoritmul RR alocă un interval de timp specificat pentru fiecare proces al cărui lungime este cunoscută sub numele de cuantă de timp. Ordinea proceselor este aceeaşi ca ordinea în care au ajuns ele în sistem. Din acest punct de vedere, algoritmul RR se aseamană cu FIFO. În cazul în care un task este gata de a fi executat dar nu mai are nimic de facut atunci următoarul task, în aceeași ordine, este verificat.

Acest ciclu continuă până când se găseşte un proces care are ceva de făcut în ciclul curent sau până când sunt verificate toate procesele. În cazul în care nu există niciun proces cu ceva de executat, atunci sistemul nu face nimic util în acest ciclu. Dacă există un proces cu ceva de făcut în ciclul curent atunci procesul este lansat și executat pentru un timp egal cu cuanta de timp. [7]

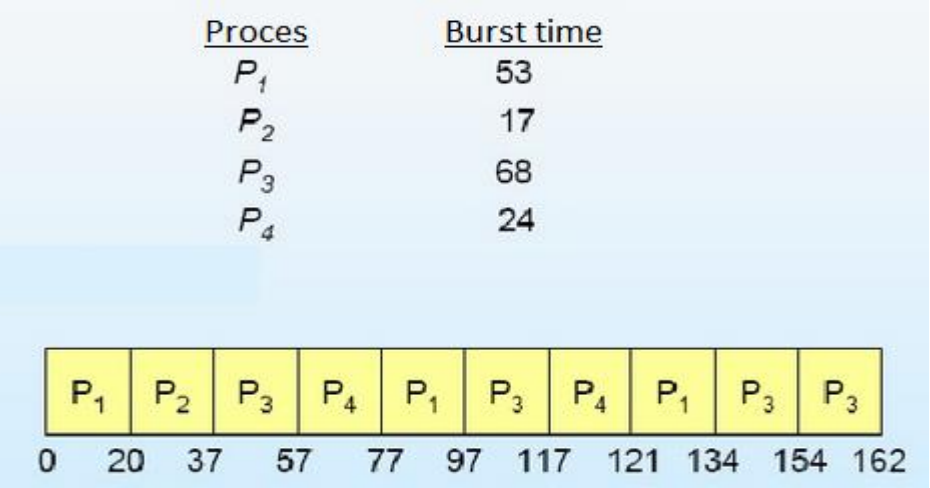


Figura 3.8 Exemplu de planificare cu algoritmul RR [7]

3.4 Afinitatea procesorului

Abilitatea în Linux de a lega unul sau mai multe procese la unul sau mai multe procesoare, numită afinitatea procesorului, este o caracteristică lung solicitată. Ideea este de a spune că rulează întotdeauna procesul x pe procesorul y sau că rulează aceste procese pe toate procesoarele, mai puţin procesorul y. Planificatorul se supune apoi la ordinea dictată, iar procesul se execută numai pe procesoarele permise. [5]

Masca de afinitate CPU a unui fir de execuţie determină setul de procesoare pe care este eligibil pentru a rula. Pe un sistem multiprocesor, stabilirea măştii de afinitate a CPU-ului poate fi utilizată pentru a obține beneficii de performanță. De exemplu, dedicând un singur procesor la un anumit fir de executie (setarea măștii de afinitate a acelui fir pentru a specifica un singur procesor, și setarea măstii de afinitate a tuturor celorlalte fire pentru a exclude acel procesor), este posibil să se asigure viteza maximă de execuţie pentru acel fir. Un fir de execuţie restricţionat să ruleze pe un singur procesor evită, de asemenea, costul de performanță cauzat de invalidarea memoriei cache care are loc atunci când un fir de execuţie încetează execuţia pe un procesor și apoi reîncepe execuția pe un alt procesor.

Există două tipuri de afinitate CPU. Prima afinitate, soft, numită, de asemenea, afinitate naturală, este tendința unui planificator de a încerca să păstreze procesele pe același CPU atâta timp cât este posibil. Este pur și simplu o încercare, dacă este vreodată nefezabil, procesele vor migra cu siguranță la un alt procesor. Noul planificator O(1) prezintă afinitate naturală excelentă. La capătul opus, este planificatorul care are afinitate CPU săracă. Acest comportament are ca rezultat efectul de ping-pong. Planificatorul ricoșează procesele între mai multe procesoare de fiecare dată când sunt programate și reprogramate. Tabelul 1 este un exemplu de afinitate naturală slabă, Tabelul 2 arată afinitatea naturală bună. [5]

Timp	t1	t2	t3	t4
Process A	CPU 0	CPU 1	CPU 0	CPU 1

Tabel 3.1 Efectul de ping-pong(afinitate slabă)

Timp	t1	t2	t3	t4
Process A	CPU 0	CPU 0	CPU 0	CPU 0

Tabel 3.2 Afinitate bună

Un beneficiu important al afinităţii procesoarelor se găseşte la aplicaţiile în timp real sau aplicații de altfel sensibile la timp. În această abordare, toate procesele de sistem sunt legate la un subset de procesoare din sistem. Aplicația specializată este apoi legată la procesoarele rămase. De obicei, într-un sistem multiprocesor, cererea de specialitate este legată la un singur procesor, iar toate celelalte procese sunt legate la celelalte. Acest lucru asigură faptul că cererea de specialitate primește atenția deplină a procesorului.

Interfața de afinitate CPU oferă un mecanism simplu, dar puternic pentru controlul planificarii proceselor pe anumite procesoare. Utilizatorii cu mai mult de un procesor pot găsi apelurile de sistemul utile în stoarcerea unei ultime picături de performanța din sistemele lor sau pentru a se asigura că timpul procesorului este disponibil chiar şi pentru sarcina cea mai exigentă în timp real. Desigur, utilizatorii cu un singur procesor nu trebuie să se simtă lăsati pe dinafară, deoarece ei pot utiliza apelurile de sistem, dar ele nu vor fi prea utile.

Capitolul 4.
Planificatorul Linux Real Time

Planificatorul este componenta de nucleu care decide care proces va fi următorul executat de către CPU. Fiecare proces are o politică de planificare asociată și o prioritate de planificare statică, *sched_priority*. Planificatorul ia decizii bazate pe cunoașterea politicii de planificare și de prioritatea statică a tuturor proceselor de pe sistem.

API-ul planificatorului Linux

Funcţiile API	Definire funcţie
sched_setscheduler (2)	Setează algoritmul de planificare și parametrii firului de execuţie specificat.
sched_getscheduler(2)	Returnează algoritmul de planificare a firului de execuţie specificat.
sched_setparam(2)	Setați parametrii de planificare a firului de execuţie specificat.
sched_getparam(2)	Returnează parametrii de planificare a firului de execuţie specificat.
sched_get_priority_max(2)	Returnează prioritatea maximă disponibilă în algoritmul de planificare specificat.
sched_get_priority_min(2)	Returnează prioritatea minimă disponibilă în algoritmul de planificare specificat.
sched_rr_get_interval(2)	Returnează valoarea cuantei de timp utilizată pentru firele de execuţie care sunt planificate cu algoritmul RR.

sched_yield(2)	Cauzează apelantul să renunțe la CPU, astfel încât un alt fir să fie executat.
sched_setaffinity(2)	(Linux specific) Setează afinitatea de procesor a unui fir de execuţie specific.
sched_getaffinity(2)	(Linux specific) Returnează afinitatea de procesor a unui fir de execuţie specific.
sched_setattr(2)	Setați politica de planificare și parametrii unui fir de execuţie specificat. Acest (Linux specific) apel de sistem oferă un superset al funcționalității sched_setscheduler (2) și sched_setparam (2).
sched_getattr(2)	Returnează politica de planificare şi parametrii unui fir de execuţie specificat. Acest (Linux specific) apel de sistem oferă un superset al funcționalității sched_setscheduler (2) și sched_setparam (2).

Tabel 4.1 API-ul planificatorului Linux [10]

4.1 Politicile de planificare disponibile in RT Linux

Pentru procesele programate sub una dintre politicile de planificare normale **(OTHER, IDLE, BATCH)**, prioritatea nu este utilizată în luarea deciziilor de planificare (aceasta trebuie să fie specificată ca 0). Aceste politici nu pot fi considerate în evaluarea real time. [10]

Procese programate sub una dintre politicile în timp real **(FIFO, RR)** au o valoare de prioritate în intervalul de la 1 (scăzut) la 99 (ridicat). După cum arată şi numărul, fire de execuţie în timp real au întotdeauna prioritate mai mare decât firele normale. POSIX.1 necesită o implementare pentru a sprijini doar un minim de 32 nivele de prioritate distincte pentru politicile în timp real. Programele portabile ar trebui să utilizeze *sched_get_priority_min(2)* şi *sched_get_priority_max (2),* pentru a găsi gama de priorități acceptate pentru o anumită politică. [10]

Conceptual, planificatorul păstrează o listă de procese executabile pentru fiecare posibilă valoare a priorităţii. Pentru a determina care fir de execuţie rulează următorul, planificatorul caută în lista valoarea nevidă cu cea mai mare prioritate statică şi selectează firul de execuţie de la capul acestei liste.

Politica de planificare a unui fir de execuţie determină unde va fi introdus acesta în lista de fire de execuţie cu prioritate statică egală şi modul în care se va deplasa în interiorul acestei liste.

Toată planificarea este preemptivă: în cazul în care un fir de execuţie cu o prioritate mai mare statică devine gata pentru a rula, firul care rulează în prezent va fi preempted şi va reveni la lista de aşteptare pentru nivelul de prioritate statică. Politica de planificare determină ordonarea numai în lista de fire de execuţie executabile cu prioritate statică egale.

În paragraful următor vom prezenta politicile folosite în proiectul curent (singurele politici în timp real din Linux RT-Preempted):

1. **First In First Out**

FIFO poate fi utilizat numai cu priorităţi statice mai mari decât 0, ceea ce înseamnă că, atunci când un thread FIFO devine executabil, acesta va fi preferat întotdeauna oricărui alt thread planificat cu algoritmii normali (OTHER, BATCH sau IDLE). FIFO este un simplu algoritm de planificare, fără timp de feliere. Pentru threadurile programate în cadrul politicii FIFO, se aplică următoarele reguli:

- Un thread FIFO, care a fost preempted de un alt thread de prioritate mai mare va rămâne în fruntea listei de priorităţi şi îşi va relua execuţia de îndată ce toate threadurile de prioritate mai mare sunt blocate din nou.
- Când un thread FIFO devine executabil, acesta va fi inserat la sfârşitul listei sale de prioritate.
- Un apel la *sched_setscheduler* (2), *sched_setparam* (2), sau *sched_setattr* (2), va pune threadul FIFO (sau RR) identificat prin pID la începutul listei dacă acesta e executabil. Ca o consecinţă, acesta poate preempta threadul care rulează în prezent în cazul în care acesta are aceeaşi prioritate.
- Un thread ce apelează *sched_yield* (2) va fi pus la sfârşitul listei de priorităţi.
- Un thread FIFO rulează până când, fie este blocată de o solicitare I/O, fie este preempted de un thread de prioritate mai mare, sau apelează *sched_yield* (2).

2. **Round Robin**

RR este o simplă îmbunătăţire a FIFO. Tot ceea ce s-a descris mai sus pentru FIFO se aplică, de asemenea, la RR, cu excepţia faptului că fiecărui thread îi este permis să ruleze numai pentru o cuantă maximă de timp. În cazul în care un thread RR a fost executat pentru o perioadă de timp egală sau mai mare decât cuanta de timp, acesta va fi pus la sfârşitul listei de priorităţi. Un thread RR, care a fost preempted de un thread de prioritate mai mare şi, ulterior, îşi reia execuţia ca un fir de execuţie, va completa porţiunea rămasă din cuanta sa de timp RR. Lungimea cuantei de timp poate fi aflată folosind *sched_rr_get_interval* (2).

4.2 Caracteristici real-time în nucleul principal Linux

De la versiunea de kernel 2.6.18 mai departe, Linux-ul devine treptat echipat cu capabilități în timp real, dintre care majoritatea sunt derivate din fostele patch-uri *realtime-preempt* dezvoltate de Ingo Molnar, Thomas Gleixner, Steven Rostedt, și alţii. Până când ele vor fi complet fuzionate în nucleul principal, acestea trebuie să fie instalate pentru a obține cele mai bune performanțe în timp real. [11]

Fără patch-uri și înainte de includerea lor deplină în nucleu principal, configurația nucleului oferă doar trei clase de preempțiune, care prevăd reducerea considerabilă a celui mai rău caz de latență de planificare:

CONFIG_PREEMPT_NONE ..... deloc
CONFIG_PREEMPT_VOLUNTARY ..... putin
CONFIG_PREEMPT_DESKTOP ..... considerabil

Cu patch-urile aplicate sau după includerea lor completă în nucleu principal, elementul de configurare suplimentar CONFIG_PREEMPT_RT devine disponibil. Dacă această opțiune este selectată, Linux este transformat într-un sistem de operare în timp real. Politicile de planificare FIFO și RR sunt apoi folosite pentru a rula un thread cu prioritate adevărată în timp real și o latență de planificare în cel mai rău caz minimă. [10]

4.3 Realtime-Preempt

Scopul patch-uri PREEMPT_RT este de a minimiza cantitatea de cod a nucleului care este nonpreempted, minimizând în acelaşi timp, cantitatea de cod care trebuie modificată în scopul de a oferi opţiunea de preempted. Secţiunile care sunt în mod normal preempted sunt secţiunile de tratare a întreruperilor sau cele de anulare a întreruperilor. Patch-ul PREEMPT_RT valorifică capacitățile SMP ale nucleului Linux pentru a adăuga acest atribut de preempţiune în plus, fără a necesita o rescriere completă a nucleului. Într-un sens, se poate gândi vag de preempţiune ca adăugarea unui nou procesor la sistem, și apoi utilizarea primitivelor de blocare normală pentru a se sincroniza cu orice acțiune întreprinsă de procesul preempted.

Patch-ul PREEMPT-RT converteşte Linux într-un nucleu complet preempted. El realizează acest lucru făcând:

- Efectuarea primitivele de blocare din nucleu preemptibile (folosind spinlocks), prin reimplementarea cu rt_mutexes. [14]
- Secţiunile critice protejate prin spinlock_t și rwlock_t sunt acum preempted. Crearea de secțiuni de bază nonpreempted este încă posibilă cu raw_spinlock_t (aceleași API-uri, cum ar fi spinlock_t). [14]
- Punerea în aplicare a moștenirilor prioritare pentru spinlock-uri și semafoare în nucleu. [14]
- Conversia tratării întreruperilor din contextul nucleului în contextul firelor de execuţie: patch-ul PREEMPT-RT tratează întreruperile soft în contextul firelor de execuţie, care este reprezentat de un *task_struct* ca un proces comun în spațiu utilizatorului. Cu toate acestea este de asemenea posibil să se înregistreze o întrerupere în contextul nucleului. [14]

În tabelul 4.2 vor fi prezentate opţiunile de configurare ale PREEMPT_RT. Acestea se pot schimba, dar ele dau o idee generală asupra tipurilor de caracteristici de depanare disponibile în PREEMPT_RT.

Opţiuni de configurare	Descriere
PREEMPT_NONE	Selectează cazul tradiţional de nonpreempţiune pentru încărcări de lucru ale serverelor (opţiune de nivel înalt de selectare a timpul de preempt).
PREEMPT_VOLUNTARY	Permite puncte de preempţiune voluntară, dar preemţiunea nucleu nu e en-gros. Acestă configuraţie este destinată pentru desktop (opţiune de nivel înalt de selectare a timpul de preempt).
PREEMPT_DESKTOP	Permite puncte de preempţiune voluntare împreună cu secţiuni non-critice de preemţiune. Acestă configuraţie este destinată pentru utilizarea de latenţă redusă pentru desktop (opţiune de nivel înalt de selectare a timpul de preempt).
PREEMPT_RT	Permite preemţiune completă, inclusiv secţiunile critice (opţiune de nivel înalt de selectare a timpul de preempt).
PREEMPT	Permite preemţiunea în secţiunile critice ale nucleului (opţiune de configurare de selecţie a caracteristicilor).
PREEMPT_BKL	Provoacă preemţiunea secţiunilor critice big-kernel-lock (opţiune de configurare de selecţie a caracteristicilor).
PREEMPT_HARDIRQS	Provoacă întreruperile hard să ruleze în contextul proceselor, făcândule astfel preemptibile. Cu toate acestea, întreruperile

	marcate ca SA_NODELAY vor continua să ruleze în contextul întreruperilor hardware (opţiune de configurare de selecţie a caracteristicilor).
PREEMPT_RCU	Provoacă secţiunile critice RCU read-side pentru a fi preemptibile (opţiune de configurare de selecţie a caracteristicilor).
PREEMPT_SOFTIRQS	Provoacă întreruperile soft să ruleze în contextul proceselor, făcându-le astfel preemptibile (opţiune de configurare de selecţie a caracteristicilor).
CRITICAL_PREEMPT_TIMING	Măsoară timpul maxim pe care nucleul îl petrece cu preemțiunea dezactivată (opţiune de configurare ale debbuging-ului).
CRITICAL_IRQSOFF_TIMING	Măsoară timpul maxim pe care nucleul îl petrece cu întreruperile hardware dezactivate (opţiune de configurare ale debbuging-ului).
DEBUG_IRQ_FLAGS	Face ca nucleul să valideze argumentele flag-urilor spin_unlock_irqrestore() și primitivelor similare (opţiune de configurare ale debbuging-ului).
DEBUG_RT_LOCKING_MODE	Permite comutarea spinlock-urilor de la nonpreemptive la preemptive. Acest lucru este util pentru dezvoltatorii care doresc să evalueze overhead-ul mecanismelor PREEMPT_RT (opţiune de configurare ale debbuging-ului).
DETECT_SOFTLOCKUP	Provoacă nucleul să arunce urma stivei curentă a oricărui proces care îşi petrece mai mult de 10 secunde în nucleu, fără a fi replanificat (opţiune

	de configurare ale debbuging-ului).
LATENCY_TRACE	Înregistrează urmele apelurilor funcţilor care reprezintă evenimente cu latență mare (opţiune de configurare ale debbuging-ului).
LPPTEST	Activează un driver de dispozitiv care efectuează măsurători de latență pe bază portului paralel (opţiune de configurare ale debbuging-ului).
PRINTK_IGNORE_LOGLEVEL	Provoacă mesajele all_printk() să fie aruncate către consolă. În mod normal e o idee foarte proastă, dar e de ajutor atunci când alte instrumente de depanare eșuează (opţiune de configurare ale debbuging-ului).
RT_DEADLOCK_DETECT	Găseşte ciclurile de timp mort (opţiune de configurare ale debbuging-ului).
RTC_HISTOGRAM	Generează datele pentru histogramele de latență ale aplicaților (opţiune de configurare ale debbuging-ului).
WAKEUP_TIMING	Măsoară timpul maxim de când un fir de execuţie de mare prioritate este trezit până la momentul în care începe de fapt execuţia (opţiune de configurare ale debbuging-ului).

Tabel 4.2 Opţiunile de configurare ale patch-ului PREEMPT_RT [12]

Aproape toate rutinele de tratare a întreruperilor rulează în contextul unui proces în mediul PREEMPT_RT. Cu toate că orice întrerupere poate fi marcată SA_NODELAY pentru a face ca acesta să ruleze în contextul întreruperii, numai întreruperile fpu_irq, irq0, irq2 și lpptest au SA_NODELAY specificat. Dintre acestea, numai irq0 (întreruperea timer-ului per-CPU) este utilizată în mod normal. fpu_irq este pentru întreruperile de coprocesor în virgulă mobilă și lpptest este utilizată pentru benchmarking-ul latențelor întreruperilor. Rețineți că temporizatoarele software

(add_timer () și friends) nu se execută în contextul întreruperilor hardware, în schimb, se execută în contextul proceselor și sunt pe deplin prioritare. [12]

Conceptul de secvenţe de cod de anulare a întreruperilor preempted poate părea o contradicție, dar este important să se țină cont de filozofia PREEMPT_RT. Această filozofie se bazează pe capacitățile SMP ale nucleului Linux să se ocupe de cursele de tratări de întreruperi, ținând cont de faptul că cele mai multe tratări de întreruperi se execută în contextul proceselor. Orice cod care interactionează cu o tratare de întrerupere trebuie să fie pregătit pentru a face față cu tratarea de întrerupere care rulează simultan pe un alt procesor.

Pentru a se reduce perioada de latenţă, există câteva modificări în PREEMPT_RT al căror scop principal este de a reduce sau de a intrerupe programarea latenta.

Prima astfel de schimbare implică hardware-ul x86 MMX / SSE. Acest hardware este manipulat în nucleu cu preempted dezactivat, iar acest lucru înseamnă că, uneori, trebuie să se aștepte până când precedente instrucțiuni MMX / SSE se execută complet. Unele instrucțiuni MMX / SSE nu sunt nicio problemă, dar altele durează foarte mult, de aceea PREEMPT_RT refuză să le folosească pe cele lente. [12]

A doua modificare se aplică la tabloul de repartiţie al variabilelelor per-CPU, ca o alternativă la dezactivarea anterioară a întreruperilor.

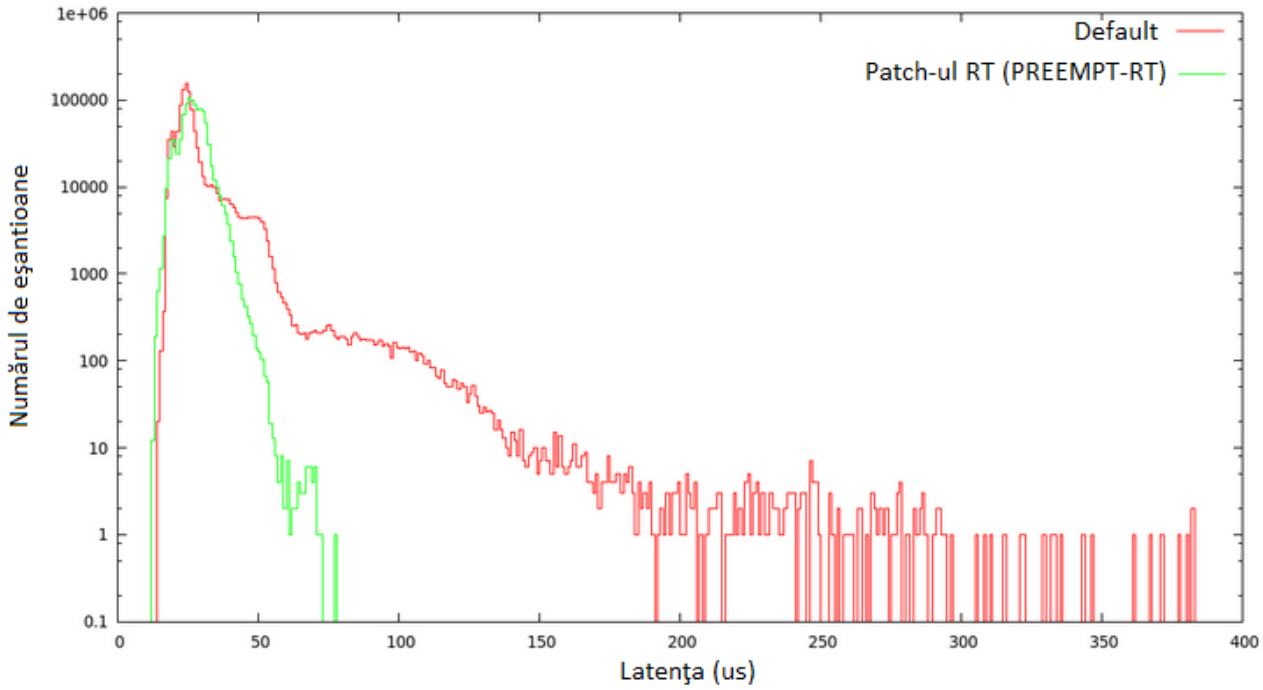


Figura 4.1 Influenţa numărului de eşantioane asupra latenței
în cazul Linux preemted default şi cazul patch-ului PREEMPT-RT [13]

Capitolul 5.
Analiza metodelor de gestiune a proceselor în timp real

Latența este un interval de timp între stimulare și răspuns, sau, dintr-un punct de vedere mai general, o întârziere de timp între cauza și efectul unor schimbări fizice în sistemul care se observă. Latența este o consecință fizică a vitezei limitate cu care o interacțiune fizică se poate propaga. Această viteză este întotdeauna mai mică sau egală cu viteza luminii. Prin urmare, orice sistem fizic, care are dimensiuni spațiale diferite de zero, va experimenta un fel de latență, indiferent de natura de stimulare la care a fost expus.

Calculatoarele rulează seturi de instrucțiuni numite procese. În sistemele de operare, executarea procesului poate fi amânată în cazul în care alte procese sunt, de asemenea, executate. Latența este întârzierea dintre instrucțiunea de proces care comandă o tranziție și hardware-ul de fapt, trecerea tensiunii de la mare la mic sau mic la mare.

Datorită importanței sale, în lucrarea curentă, latența va fi principalul factor de analiză.

Date inițiale generale

Sistemul de operare în timp real pe care s-a lucrat este Linux 14.04 cu kernelul 2.12.33, la care s-a aplicat patch-ul PREEMPT-RT linux-2.12.33-rt47. Simulările s-au realizat pe un sistem cu un procesor cu 2 coruri şi frecvenţa de bază de 1.9 GHz.

Simulările au fost făcute în diverse condiţii: monoprocesor sau multiprocesor, în condiţii de încărcare a sistemului slabe sau ridicate.

Algoritmul de testare al celor două politici de gestiune a proceselor în timp real disponibile în RT-Linux (FIFO şi Round Robin) este următorul:

- Generarea de procese independente în timp real;
- Acordarea de priorităţi statice (aleatoare, egale sau stabilite);
- Stabilirea afinităţii procesorului (monoprocesor sau multiprocesor);
- Stabilirea algoritmului de gestiune a proceselor;
- Alegerea parametrilor proprii ai proceselor în timp real (durata totală a unui fir de execuţie, distanţa dintre două fire de execuţie, tipul de selectarea a timpului, CLOCK_MONOTONIC – default, timpul se ia de la ceas sau CLOCK_REALTIME – timpul se ia de la sistem gettimeofday());
- Alegerea duratei totale a programului de simulare sau numărul de repetări (loops);
- Folosirea clock_nanosleep(). Dă o mai mare asemănare cu sistemele în timp real hard;

Setul de lucru (workload)

Am ales ca set de lucru **5 procese** în timp real independente cu priorităţi distincte sau asemanătoare în intervalul de la 1 la 99 (minimul şi maximul algoritmilor FIFO şi RR). Prioritatea 0 înseamnă că sistemul nu mai e în timp real. **Durata totală** a programului a fost aleasă **30 de secunde**, **durata** de wait() **într-un thread** de **1000 de microsecunde**, iar **durata dintre**

executarea a două thread-uri succesive de **500 de microsecunde**. Aceste valori nu sunt chiar exacte deoarece în timpul executării programului de simulare erau deschise şi alte fire de execuţie (scheduler, întreruperi, alte procese).

Toate rezultatele latenţei vor fi în **microsecunde**.

Rezultate obţinute

FIFO	**Monoprocesor**		**Testare într-un mediu de calcul puţin solicitat**		
Proces	**Prioritatea**	**Numărul de apelări a thread-urilor**	**Latenţă minimă**	**Latenţă maximă**	**Latenţă medie**
P0	50	29995	2	52	3
P1	48	19997	3	19	4
P2	38	14997	3	17	4
P3	22	11998	3	17	4
P4	19	9998	2	38	2

Tabel 5.1 Rezultate opţinute FIFO 1

FIFO	**Multiprocesor**		**Testare într-un mediu de calcul puţin solicitat**		
Proces	**Prioritatea**	**Numărul de apelări a thread-urilor**	**Latenţă minimă**	**Latenţă maximă**	**Latenţă medie**
P0	50	29997	2	27	3
P1	54	19998	3	18	4
P2	77	14998	2	21	3
P3	22	11999	2	16	4
P4	36	9999	2	11	3

Tabel 5.2 Rezultate opţinute FIFO 2

Observăm că odată cu creşterea numărului de procesoare scade atât durata latenţei maxime, cât şi latenţa medie. Acest lucru este normal deoarece creşterea numărului de procesoare duce la paralelism, deci, teoretic, la scăderea timpului de execuţie a proceselor. Numărul de apelări a thread-urilor rămâne constant.

FIFO	**Monoprocesor**		**Testare într-un mediu de calcul foarte solicitat**		
Proces	**Prioritatea**	**Numărul de apelări a thread-urilor**	**Latenţă minimă**	**Latenţă maximă**	**Latenţă medie**
P0	50	29999	2	724	10
P1	54	19998	3	70	9
P2	77	14999	3	79	11
P3	22	11998	3	85	10
P4	36	9999	3	427	10

Tabel 5.3 Rezultate opţinute FIFO 3

FIFO	Multiprocesor		Testare într-un mediu de calcul foarte solicitat		
Proces	**Prioritatea**	**Numărul de apelări a thread-urilor**	**Latenţă minimă**	**Latenţă maximă**	**Latenţă medie**
P0	50	29998	3	92	10
P1	54	19998	3	76	9
P2	77	14999	3	58	8
P3	22	11999	3	71	8
P4	36	9999	3	52	6

Tabel 5.4 Rezultate opţinute FIFO 4

Observăm că şi în cazul unei testări într-un mediu solicitat, odată cu creşterea numărului de procesoare scade atât durata latenţei maxime, cât şi latenţa medie. Se mai constantă că prioritatea proceselor nu influentează latenţa, însă în toate cazurile anterioare primul proces lansat are numărul de apelări cel mai mare.

RR	Monoprocesor		Testare într-un mediu de calcul puţin solicitat		
Proces	**Prioritatea**	**Numărul de apelări a thread-urilor**	**Latenţă minimă**	**Latenţă maximă**	**Latenţă medie**
P0	50	29997	2	67	3
P1	48	19998	3	14	4
P2	38	14998	2	27	4
P3	22	11999	3	14	4
P4	19	9999	2	57	3

Tabel 5.5 Rezultate opţinute RR 1

RR	Multiprocesor		Testare într-un mediu de calcul puţin solicitat		
Proces	**Prioritatea**	**Numărul de apelări a thread-urilor**	**Latenţă minimă**	**Latenţă maximă**	**Latenţă medie**
P0	50	29990	2	12	3
P1	48	19993	3	16	4
P2	38	14995	2	8	2
P3	22	11996	2	10	3
P4	19	9996	2	9	3

Tabel 5.6 Rezultate opţinute RR 2

Algoritmul RR este puţin mai eficient din punct de vedere al latenţei medii decât FIFO în cazul unui sistem multiprocesor şi al unei testări intr-un mediu puţin solicitat deoarece acesta are un număr mai mic de apelări al thread-urilor.

RR	Monoprocesor		Testare într-un mediu de calcul foarte solicitat		
Proces	**Prioritatea**	**Numărul de apelări a thread-urilor**	**Latenţă minimă**	**Latenţă maximă**	**Latenţă medie**
P0	50	29989	2	259	9
P1	48	19993	3	65	8
P2	38	14994	3	73	9
P3	22	11996	3	63	9
P4	19	9996	2	268	9

Tabel 5.7 Rezultate opţinute RR 3

RR	Multiprocesor		Testare într-un mediu de calcul foarte solicitat		
Proces	**Prioritatea**	**Numărul de apelări a thread-urilor**	**Latenţă minimă**	**Latenţă maximă**	**Latenţă medie**
P0	50	29994	3	80	9
P1	48	19996	2	62	8
P2	38	14997	3	66	8
P3	22	11997	3	65	9
P4	19	9998	3	65	8

Tabel 5.8 Rezultate opţinute RR 4

Şi în cazul unei testări într-un mediu foarte solicitat, algoritmul RR este mai bun din punct de vedere al latenţei maxime şi medii. Se observă că indiferent de condiţiile de simulare latenţa minimă rămâne constantă.

Media latenţelor					
Minime		Medii		Maxime	
FIFO	RR	FIFO	RR	FIFO	RR
2.65	**2.5**	**6.4**	**5.95**	**108.95**	**65**

Tabel 5.9 Media latenţelor

Limitele tolerabile pentru latență pentru procesarea în timp real este un subiect de investigații și dezbateri, dar este estimată a fi între 6 și 20 de milisecunde. [15]

Maximul latenţei obţinute după o perioadă de **o oră**, într-un **mediu solicitat**, e de **1.2 ms**, deci ambele metode de gestiune a proceselor se încadrează în acest interval al limitei tolerabile, deci îşi îndeplinesc cu succes atribuţiile şi demonstreză că sunt politici viabile de planificare în timp real, cu un plus pentru RR care, mai ales la latenţa maximă, prezintă un plus.

Break Trace

Comanda de break trace se trimite atunci când latenţa depăşeşte un anumit prag stabilit.

	Comanda break trace			
	FIFO		RR	
	Monoprocesor	Multiprocesor	Monoprocesor	Multiprocesor
Pragul de break trace [us]	25	20	25	20
Valoarea latenţei de break [us]	69	25	32	21
Timpul de break [us]	$13.1205 * 10^6$	$0.1305 * 10^6$	$2.205 * 10^6$	$2.6745 * 10^6$

Tabel 5.10 Comanda de break trace

- Pragul de break trace este valoarea care dacă e întrecută de latenţa maximă a oricărui dintre procesele în execuţie, programul se opreşte.
- Valoarea latenţei de break este valoarea de maxim a latenţei care a întrecut pragul de break.
- Timpul de break e timpul care a trecut de la execuţia primului thread până la oprirea programului.

Valorile latenţei de break şi a timpului de break sunt valori medii calculate în urma a 20 de experimente rulând în aceleaşi scenarii de funcţionare.

Concluziile proiectului

Odată cu creşterea segmentului de tehnologii Internet of Things format din dispozitive embedded şi în timp real, cunoaşterea sistemelor în timp real şi a sistemelor de operare în timp real a devenit un lucru crucial pentru dezvoltatori.

Principalul avantaj, dar şi principala problemă a acestor sisteme este timpul. De aceea gestiunea optimă a timpului şi a răspunsurilor proceselor în timp este foarte importantă. Lucrarea de faţă vine în ajutorul rezolvării acestei probleme. Ea a avut ca şi scop determinarea unui scenariu optim de gestiune a proceselor cu constrângeri real time.

S-a realizat un program de analiză a politicilor de planificare a preceselor în timp real, concurente la mai multe resurse distincte. Acest program compară mai multe configuraţii de procese concurente, rulând în diferite configuraţii. S-au completat tabele şi s-au trasat grafice de performanţă, în funcţie de particularităţile rulării. Se va indica o procedură de alegere si adaptare a gestiunii de procese în timp real după caracteristicile configuraţiei de procese.

Rezultatele proiectului constau în analize de performantă a principalilor algoritmi de gestiune a proceselor în timp real ale kernelului RT Linux pe baza unor configuraţii de procese anterior stabilite. Deoarece constrângerile fundamentale ale sistemelor în timp real sunt date de timp, necesitatea unei gestiuni optime a proceselor este foarte mare sau chiar vitală în cazul sistemelor real time hard.

Kernelul Linux cu patch-ul de timp real a fost foarte util în analiza algoritmilor prin API-urile disponibile, dar şi prin instrumentele de configurare a sistemului.

Un dezavantaj al acestui sistem real time îl reprezintă faptul că dispune de doar două metode de gestiune a proceselor în timp real (First In First Out şi Round Robin), metode destul de simple şi cu rezultate bune doar în anumite cazuri, cazuri care nu se regăsesc întotdeauna în realitate. Un alt dezavantaj îl reprezintă asemănarea dintre ele, RR fiind o variantă puţin înbunătăţită a FIFO (adăugarea cuantei de timp, coantă de timp care în cazul proceselor real time, procese simple, nu este întrecută).

Din comparaţia dintre cele două metode a rezultat un mic avantaj pentru RR în cazul proceselor cu fire de execuţie de scurtă durată (procese predominante în domeniul real time). Atunci cînd durata firelor de execuţie e aproximativ egală cu cuanta de timp RR, atunci cei doi algoritmi dau aceiaşi satisfacţie. Ordinea de execuţie a proceselor e aceiaşi, dar RR porneşte mai puţine fire d execuţie decât FIFO, motiv pentru care are o latenţă mai mică.

Algoritmul RR se recomandă în sistemele cu partajarea timpului deoarece el asigură o distribuţie echitabilă între procese. Ambele metode sunt mai rentabile pe sistemele monoprocesor deoarece, în cazul sistemelor în timp real cu mai multe procesoare, acestea provoacă o scădere a utilizării procesorului.

Ca posibile direcţii de dezvoltare s-au remarcat numărul mic de algoritmi prezenti în sistemul de operare real time Linux. Posibilităţi de dezvoltare a acestor algoritmi există, în sensul îmbunataţirii perfomantelor legate de o folosire totală a procesorului pe parcursul planficării proceselor, de reducerea timpului in care procesorul nu este activ şi de scăderea latenţei, a numărului de preempţiuni.

Bibliografie

[1] Jim Huang, „Making Linux do Hard Real-Time”, Free Electrons, 26.07.2014

[2] Alan Burns and Andy Wellings, Real-Time Systems and Programming Languages (4th ed.), Addison-Wesley, 2009

[3] Liu, Jane W.S., Real-time systems, Upper Saddle River, NJ: Prentice Hall, 2000

[4] Embedded Market Study, UBM Tech Electronics, 2014
http://bd.eduweb.hhs.nl/es/2014-embedded-market-study-then-now-whats-next.pdf
accesat la data 22.05.2016

[5] Robert Love, „CPU Affinity”, Linux Jurnal, nr. 111 / 2003

[6] Lehoczky, J.P., Sha, L., and Ding, Y. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior". Proceedings of IEEE Real-Time System Symposium, 1989

[7] Kevin Churnetski, Real-time scheduling algorithms, task visualization, Rochester Institute of Technology, 2006

[8] Ing. Cojocaru Şiegfried, Mecanisme de comunicaţie în timp real în sistemele informatice industriale, teză de doctorat

[9] Insup Lee, OS Overview – Real-Time Scheduling, CSE 480/CIS 700, University of Pennsylvania, 2006

[10] Michael Kerrisk, man7.org, http://man7.org/linux/man-pages/man7/sched.7.html, accesat la data de 18.06.2016

[11] Luotao Fu, Robert Schwebel, Kernel Development Group, https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO accesat la data 29.07.2016

[12] Paul McKenney, “A realtime preemption overview”, LWN.net, August 10, 2005

[13] EMLID, Raspberry PI Real-Time Kernel, APM, NEWS, TUTORIALS, https://emlid.com/raspberry-pi-real-time-kernel/ accesat la data de 30.06.2016

[14] Theodore Ts'o, Darren Hart, Jekacur, https://rt.wiki.kernel.org/ , accesat la data de 30.06.2016

[15] Sara Kudrle, „Fingerprinting for Solving A/V Synchronization Issues within Broadcast Environments", Motion Imaging Jernal, Iulie 2011

Anexa 1 Codul sursă pentru măsurarea duratei şi a ordinii proceselor

```
/* Se compileaza cu:
 *             gcc Main.c -o Main -lpthread -
lm
 * si se executa cu:
 *             ./Main
 */
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>
#include <sys/time.h>
#include <math.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <ctype.h>
#include <syscall.h>

#define handle_error_en(en, msg) do { errno =
en; perror(msg); exit(EXIT_FAILURE); } while
(0)
#define handle_error(msg) do { perror(msg);
exit(EXIT_FAILURE); } while (0)

#define DEBUG 1

//#define ALGORITHM    (SCHED_FIFO)   // Set
scheduling policy to FIRST IN FIRST OUT
#define ALGORITHM             (SCHED_RR)
       // Set sscheduling policy to ROUND
ROBIN

#define NUM_THREAD                    100
              // Number of threads
#define PRIORITY_MEAN         50
       // Mean for priority normal
distribution
#define PRIORITY_STDDEV               5
              // Standard deviation for
priority normal distribution
#define CNTR_MEAN                     200000
       // Mean for wait counter normal
distribution
#define CNTR_STDDEV                   20000
       // Standard deviation for wait counter
normal distribution

// Functions declarations
void *pthread_function_wait(void *ptr);
double normal_distribution(double mean, double
stddev);
void set_CPU_affinity(int num_CPUs);

// Global variables
int times = 200000;
int count[NUM_THREAD];

// Main
int main(void)
{
       FILE* fp;
    char* fn = "Priority.txt";
    char* fm = "Durata.txt";

       int i, counter[NUM_THREAD];
       int* params;
       pthread_t thread[NUM_THREAD];
       pthread_attr_t attr[NUM_THREAD];
       struct sched_param parm[NUM_THREAD];
       int iret[NUM_THREAD];

       params =
(int*)malloc(sizeof(int)*NUM_THREAD);

       /*     Set CPU affinity to only run on
1 core. */
       set_CPU_affinity( 0 );

       /* Init threads attributes.   */
       for(i = 0; i < NUM_THREAD; i++)
       {
              pthread_attr_init(&attr[i]);
       }

       srand(time(NULL));

       /*     Set threads policy and priority
       */
       for(i = 0; i < NUM_THREAD; i++)
       {

       pthread_attr_getschedparam(&attr[i],
&parm[i]);
              parm[i].sched_priority =
(int)normal_distribution(PRIORITY_MEAN,
PRIORITY_STDDEV);

              #ifdef DEBUG
                     fp = fopen(fn, "a");
                     fprintf(fp, " Prioritate
proces %d = %d \n", i,
parm[i].sched_priority);
                     fclose(fp);
              #endif


       pthread_attr_setschedpolicy(&attr[i],
ALGORITHM);

       pthread_attr_setschedparam(&attr[i],
&parm[i]);
       }

       for(i = 0; i < NUM_THREAD; i++)
       {
              counter[i] =
(int)normal_distribution(CNTR_MEAN,
CNTR_STDDEV);

              #ifdef DEBUG
                     fp = fopen(fm, "a");
                     fprintf(fp, " Wait
couter proces %d = %d \n", i, counter[i]);
                     fclose(fp);
              #endif
       }
       /*     Create independent threads each
of which will execute wait function with
different counter      */
       for(i = 0; i < NUM_THREAD; i++)
       {
              params[i] = counter[i];

              iret[i] =
pthread_create(&thread[i], &attr[i], (void*)
pthread_function_wait, (void*) &params[i]);
              if(iret[i] != 0)
       handle_error_en(iret[i],
"pthread_create");


       pthread_setschedparam(thread[i],
ALGORITHM, &parm[i]);
       }

       sleep(10);
```

```
        /* Destroy the thread attributes
object, since it is no longer needed */
        for(i = 0; i < NUM_THREAD; i++)
        {
                iret[i] =
pthread_attr_destroy(&attr[i]);
                if (iret[i] != 0)
handle_error_en(iret[i],
"pthread_attr_destroy");
        }

        /*      Wait till threads are complete
before main continues. Unless we
                wait, we run the risk of
executing an exit which will terminate
                the process and all threads
before the threads have completed.    */
        for(i = 0; i < NUM_THREAD; i++)
        {
                iret[i] =
pthread_join(thread[i], NULL);
                if (iret[i] != 0)
        handle_error_en(iret[i],
"pthread_join");
        }

        exit(0);                //return
EXIT_SUCCESS;
}

double normal_distribution(double mean, double
stddev)
{
        static double V1, V2, S;
        static int phase = 0;
        double X;

        if(phase == 0)
        {
                do
                {
                        V1 = (2*((double)rand()
/ RAND_MAX) - 1);
                        V2 = (2*((double)rand()
/ RAND_MAX) - 1);
                        S = V1 * V1 + V2 * V2;
                }
                while(S >= 1 || S == 0);

                X = V1 * sqrt(-2 * log(S) / S);
        }
        else
        {
                X = V2 * sqrt(-2 * log(S) / S);
        }

        phase = 1 - phase;

        return (stddev * X) + mean;
}

void *pthread_function_wait(void *ptr)
{
        int counter;
        counter = *(int*) ptr;

        printf("Thread %ld \n",
syscall(SYS_gettid));

        struct timespec t1, t2, rr_t;
        float t_start, t_stop;
        FILE* fp;
    char* fn = "Res.txt";

//      while(times > 0)
//      {
                if( clock_gettime(
CLOCK_REALTIME, &t1) == -1 )
                {
                        perror( "clock gettime"
);
                        exit( EXIT_FAILURE );
                }

                while((1000 * counter) > 0)
                {
                        counter--;
                }

                if( clock_gettime(
CLOCK_REALTIME, &t2) == -1 )
                {
                        perror( "clock gettime"
);
                        exit( EXIT_FAILURE );
                }

                times--;

//      }

        t_start = t1.tv_nsec;
        t_stop = t2.tv_nsec;
        fp = fopen(fn, "a");
        fprintf(fp, "Start time = %f, Stop time
= %f \n", t_start, t_stop);
        fclose(fp);

        return (void*) NULL;
}

void set_CPU_affinity(int num_CPUs)
{
        cpu_set_t mask;

        /*      CPU_ZERO initializes all the
bits in the mask to zero      */
        CPU_ZERO(&mask);
        /*      CPU_SET sets only the bit
corresponding to cpu   */
        CPU_SET(num_CPUs, &mask);

        /*      Set CPU affinity for a task
        */
        if (sched_setaffinity(0, sizeof(mask),
&mask) == -1)
        {
                printf("Could not set CPU
Affinity \n");
        }
}
```

Anexa 2 Codul sursă pentru numărarea apelurilor planificatorului către procese

```
// FIFO scheduling policy of Linux, Thread1
priority is higher than Thread2 priority
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>

void *print_message_function(void *ptr);

int count1 = 0;
int count2 = 0;
int times = 200000; // need set enough big to
see effecting

int main(void) {
        pthread_t thread1, thread2;
        char *message1 = "Thread 1";
        char *message2 = "Thread 2";
        int iret1, iret2;

        //CPU affinity
        int num_CPUs = 0; // only run on 1 core
        cpu_set_t mask;

        CPU_ZERO(&mask);
        CPU_SET(num_CPUs, &mask);
        if (sched_setaffinity(0, sizeof(mask),
&mask) == -1)
        {
                printf("Could not set CPU
Affinity \n");
        }

        //set attributes
        pthread_attr_t attr1, attr2;
        struct sched_param parm1, parm2;
        pthread_attr_init(&attr1);
        pthread_attr_init(&attr2);

        // create threads
        pthread_attr_getschedparam(&attr1,
&parm1);
        parm1.sched_priority =
sched_get_priority_max(SCHED_FIFO);
        pthread_attr_setschedpolicy(&attr1,
SCHED_FIFO);
        pthread_attr_setschedparam(&attr1,
&parm1);

        iret1 = pthread_create(&thread1,
&attr1, (void*) print_message_function,
(void*) message1);
        pthread_setschedparam(thread1,
SCHED_FIFO, &parm1);

        pthread_attr_getschedparam(&attr2,
&parm2);
        parm2.sched_priority =
sched_get_priority_min(SCHED_FIFO);
        pthread_attr_setschedpolicy(&attr2,
SCHED_FIFO);
        pthread_attr_setschedparam(&attr2,
&parm2);

        iret2 = pthread_create(&thread2,
&attr2, (void*) print_message_function,
(void*) message2);
        pthread_setschedparam(thread2,
SCHED_FIFO, &parm2);

        pthread_join(thread1, NULL);

        pthread_join(thread2, NULL);

        printf("count1 = %d, count2 = %d\n",
count1, count2);
        printf("Thread 1 returns: %d\n",
iret1);
        printf("Thread 2 returns: %d\n",
iret2);
        exit(0);
        //return EXIT_SUCCESS;
}

void *print_message_function(void *ptr) {
        char *message;
        message = (char *) ptr;

        while (times > 0) {
                int i = 0;
                for (i = 0; i < 20000; i++)
                        i++; // only for delay

                if (strcmp(message, "Thread 1")
== 0) {
                        count1 += 1; // count
times Thread1 run
                        printf("\n thread 1");
                } else {
                        count2 += 1; // count
times Thread2 run
                        printf("\n thread 2");
                }

                times--;
        }

        return (void*) NULL;
}
```

```
// Round Robin scheduling policy of Linux,
Thread1 priority is higher Thread2 priority.
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>

void *print_message_function(void *ptr);

int count1 = 0;
int count2 = 0;
int times = 100000; // need set enough big to
see effecting

int main(void) {
	pthread_t thread1, thread2;
	char *message1 = "Thread 1";
	char *message2 = "Thread 2";
	int iret1, iret2;

	//Setting CPU affinity
	int num_CPUs = 0; // only run on 1 core
	cpu_set_t mask;

	/* CPU_ZERO initializes all the bits in
the mask to zero. */
	CPU_ZERO(&mask);
	/* CPU_SET sets only the bit
corresponding to cpu. */
	CPU_SET(num_CPUs, &mask);
	/* sched_setaffinity returns 0 in
success */
	if (sched_setaffinity(0, sizeof(mask),
&mask) == -1)
	{
		printf("Could not set CPU
Affinity \n");
	}
	//====================================
==========

	//set attributes
	pthread_attr_t attr1, attr2;
	struct sched_param parm1, parm2;
	pthread_attr_init(&attr1);
	pthread_attr_init(&attr2);

	//====================================
==========
	/* Create independent threads each of
which will execute function */

	pthread_attr_getschedparam(&attr1,
&parm1);
	parm1.sched_priority =
sched_get_priority_max(SCHED_RR);
	pthread_attr_setschedpolicy(&attr1,
SCHED_RR);
	pthread_attr_setschedparam(&attr1,
&parm1);

	iret1 = pthread_create(&thread1,
&attr1, (void*) print_message_function,
(void*) message1);
	pthread_setschedparam(thread1,
SCHED_RR, &parm1);

	//====================================
==========
	pthread_attr_getschedparam(&attr2,
&parm2);
	parm2.sched_priority =
sched_get_priority_min(SCHED_RR);
	pthread_attr_setschedpolicy(&attr2,
SCHED_RR);
	pthread_attr_setschedparam(&attr2,
&parm2);

	iret2 = pthread_create(&thread2,
&attr2, (void*) print_message_function,
(void*) message2);
	pthread_setschedparam(thread2,
SCHED_RR, &parm2);

	//====================================
==========
	/* Wait till threads are complete
before main continues. Unless we  */
	/* wait we run the risk of executing an
exit which will terminate   */
	/* the process and all threads before
the threads have completed.   */

	pthread_join(thread1, NULL);
	pthread_join(thread2, NULL);

	printf("count1 = %d, count2 = %d\n",
count1, count2);
	printf("Thread 1 returns: %d\n",
iret1);
	printf("Thread 2 returns: %d\n",
iret2);
	exit(0);
	//return EXIT_SUCCESS;
}

//========================================
===
void *print_message_function(void *ptr) {
	char *message;
	message = (char *) ptr;

	while (times > 0) {
		int i = 0;
		for (i = 0; i < 20000; i++)
			i++; // only for delay

		if (strcmp(message, "Thread 1")
== 0) {
			count1 += 1; // count
times Thread1 run
		} else {
			count2 += 1; // count
times Thread2 run
		}

		times--;
	}

	return (void*) NULL;
}
```

```
// NORMAL scheduling policy of Linux, Thread1
and Thread2 are not equal priority.
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>

void *print_message_function(void *ptr);

int count1 = 0;
int count2 = 0;
int times = 100000; // need set enough big to
see effecting

int main(void) {
	pthread_t thread1, thread2;
	char *message1 = "Thread 1";
	char *message2 = "Thread 2";
	int iret1, iret2;

	//Setting CPU affinity
	int num_CPUs = 0; // only run on 1 core
	cpu_set_t mask;

	/* CPU_ZERO initializes all the bits in
the mask to zero. */
	CPU_ZERO(&mask);
	/* CPU_SET sets only the bit
corresponding to cpu. */
	CPU_SET(num_CPUs, &mask);
	/* sched_setaffinity returns 0 in
success */
	if (sched_setaffinity(0, sizeof(mask),
&mask) == -1)
	{
		printf("Could not set CPU
Affinity \n");
	}
	//====================================
==========

	//set attributes
	pthread_attr_t attr1, attr2;
	struct sched_param parm1, parm2;
	pthread_attr_init(&attr1);
	pthread_attr_init(&attr2);

	//====================================
==========
	/* Create independent threads each of
which will execute function */

	pthread_attr_getschedparam(&attr1,
&parm1);
	parm1.sched_priority =
sched_get_priority_max(SCHED_OTHER);
	pthread_attr_setschedpolicy(&attr1,
SCHED_OTHER);
	pthread_attr_setschedparam(&attr1,
&parm1);

	iret1 = pthread_create(&thread1,
&attr1, (void*) print_message_function,
(void*) message1);
	pthread_setschedparam(thread1,
SCHED_OTHER, &parm1);

	//====================================
==========
	pthread_attr_getschedparam(&attr2,
&parm2);
	parm2.sched_priority =
sched_get_priority_min(SCHED_OTHER);
	pthread_attr_setschedpolicy(&attr2,
SCHED_OTHER);
	pthread_attr_setschedparam(&attr2,
&parm2);

	iret2 = pthread_create(&thread2,
&attr2, (void*) print_message_function,
(void*) message2);
	pthread_setschedparam(thread2,
SCHED_OTHER, &parm2);

	//====================================
==========

	/* Wait till threads are complete
before main continues. Unless we  */
	/* wait, we run the risk of executing
an exit which will terminate  */
	/* the process and all threads before
the threads have completed.   */

	pthread_join(thread1, NULL);
	pthread_join(thread2, NULL);

	printf("count1 = %d, count2 = %d\n",
count1, count2);
	printf("Thread 1 returns: %d\n",
iret1);
	printf("Thread 2 returns: %d\n",
iret2);
	exit(0);
	//return EXIT_SUCCESS;
}

//========================================
===
void *print_message_function(void *ptr) {
	char *message;
	message = (char *) ptr;

	while (times > 0) {
		int i = 0;
		for (i = 0; i < 20000; i++)
			i++; // only for delay

		if (strcmp(message, "Thread 1")
== 0) {
			count1 += 1; // count
times Thread1 run
		} else {
			count2 += 1; // count
times Thread2 run
		}

		times--;
	}

	return (void*) NULL;
}
```

Anexa 3 Funcţii C folosite la simularea şi determinarea latenţei

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdarg.h>
#include <unistd.h>
#include <fcntl.h>
#include <getopt.h>
#include <pthread.h>
#include <signal.h>
#include <sched.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <limits.h>
#include <linux/unistd.h>

#define DEFAULT_INTERVAL 1000
#define DEFAULT_DISTANCE 500

#ifndef SCHED_IDLE
#define SCHED_IDLE 5
#endif
#ifndef SCHED_NORMAL
#define SCHED_NORMAL SCHED_OTHER
#endif


static int clock_nanosleep(clockid_t clock_id,
int flags, const struct timespec *req,
               struct timespec *rem)
{
       if (clock_id ==
CLOCK_THREAD_CPUTIME_ID)
               return -EINVAL;
       if (clock_id ==
CLOCK_PROCESS_CPUTIME_ID)
               clock_id =
MAKE_PROCESS_CPUCLOCK (0, CPUCLOCK_SCHED);

       return syscall(__NR_clock_nanosleep,
clock_id, flags, req, rem);
}

static void set_latency_target(void)
{
       struct stat s;
       int ret;

       if (stat("/dev/cpu_dma_latency", &s) ==
0) {
               latency_target_fd =
open("/dev/cpu_dma_latency", O_RDWR);
               if (latency_target_fd == -1)
                       return;
               ret = write(latency_target_fd,
&latency_target_value, 4);
               if (ret == 0) {
                       printf("# error setting
cpu_dma_latency to %d!: %s\n",
latency_target_value, strerror(errno));

       close(latency_target_fd);
                       return;
               }
               printf("# /dev/cpu_dma_latency
set to %dus\n", latency_target_value);
       }
}

static inline void tsnorm(struct timespec *ts)
{
       while (ts->tv_nsec >= NSEC_PER_SEC) {
               ts->tv_nsec -= NSEC_PER_SEC;
               ts->tv_sec++;
       }
}

/*
 * Check the error status of
sched_setscheduler
 * If an error can be corrected by raising the
soft limit priority to
 * a priority less than or equal to the hard
limit, then do so.
 */
static int setscheduler(pid_t pid, int policy,
const struct sched_param *param)
{
       int err = 0;

try_again:
       err = sched_setscheduler(pid, policy,
param);
       if (err) {
               err = errno;
               if (err == EPERM) {
                       int err1;
                       err1 =
raise_soft_prio(policy, param);
                       if (!err1) goto
try_again;
               }
       }

       return err;
}

static void handlepolicy(char *polname)
{
       if (strncasecmp(polname, "other", 5) ==
0)
               policy = SCHED_OTHER;
       else if (strncasecmp(polname, "batch",
5) == 0)
               policy = SCHED_BATCH;
       else if (strncasecmp(polname, "idle",
4) == 0)
               policy = SCHED_IDLE;
       else if (strncasecmp(polname, "fifo",
4) == 0)
               policy = SCHED_FIFO;
       else if (strncasecmp(polname, "rr", 2)
== 0)
               policy = SCHED_RR;
       else    /* default policy if we don't
recognize the request */
               policy = SCHED_OTHER;
}

int main(int argc, char **argv)
{
       sigset_t sigset;
       int signum = SIGALRM;
       int mode;
       struct thread_param **parameters;
       struct thread_stat **statistics;
       int max_cpus =
sysconf(_SC_NPROCESSORS_CONF);
       int i, ret = -1;
       int status;

       process_options(argc, argv);

       if (check_privs())
               exit(EXIT_FAILURE);

       /* Checks if numa is on, program exits
if numa on but not available */
       numa_on_and_available();
```

```
	/* lock all memory (prevent swapping)
*/
	if (lockall)
		if
(mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
			perror("mlockall");
			goto out;
		}

	/* use the /dev/cpu_dma_latency trick
if it's there */
	set_latency_target();

	kernelversion = check_kernel();

	if (kernelversion == KV_NOT_SUPPORTED)
		warn("Running on unknown kernel
version...YMMV\n");

	setup_tracer();

	if (check_timer())
		warn("High resolution timers
not available\n");

	mode = use_nanosleep + use_system;

	sigemptyset(&sigset);
	sigaddset(&sigset, signum);
	sigprocmask (SIG_BLOCK, &sigset, NULL);

	signal(SIGINT, sighand);
	signal(SIGTERM, sighand);

	parameters = calloc(num_threads,
sizeof(struct thread_param *));
	if (!parameters)
		goto out;
	statistics = calloc(num_threads,
sizeof(struct thread_stat *));
	if (!statistics)
		goto outpar;

	for (i = 0; i < num_threads; i++) {
		pthread_attr_t attr;
		int node;
		struct thread_param *par;
		struct thread_stat *stat;

		status =
pthread_attr_init(&attr);
		if (status != 0)
			fatal("error from
pthread_attr_init for thread %d: %s\n", i,
strerror(status));

		node = -1;
		if (numa) {
			void *stack;
			void *currstk;
			size_t stksize;

			/* find the memory node
associated with the cpu i */
			node =
rt_numa_numa_node_of_cpu(i);

			/* get the stack size
set for for this thread */
			if
(pthread_attr_getstack(&attr, &currstk,
&stksize))
				fatal("failed to
get stack size for thread %d\n", i);

			/* if the stack size is
zero, set a default */
			if (stksize == 0)
				stksize =
PTHREAD_STACK_MIN * 2;

			/*  allocate memory for
a stack on appropriate node */
			stack =
rt_numa_numa_alloc_onnode(stksize, node, i);

			/* set the thread's
stack */
			if
(pthread_attr_setstack(&attr, stack, stksize))
				fatal("failed to
set stack addr for thread %d to 0x%x\n",
					i,
stack+stksize);
		}

		/* allocate the thread's
parameter block  */
		parameters[i] = par =
threadalloc(sizeof(struct thread_param),
node);
		if (par == NULL)
			fatal("error allocating
thread_param struct for thread %d\n", i);
		memset(par, 0, sizeof(struct
thread_param));

		/* allocate the thread's
statistics block */
		statistics[i] = stat =
threadalloc(sizeof(struct thread_stat), node);
		if (stat == NULL)
			fatal("error allocating
thread status struct for thread %d\n", i);
		memset(stat, 0, sizeof(struct
thread_stat));

		/* allocate the histogram if
requested */
		if (histogram) {
			int bufsize = histogram
* sizeof(long);

			stat->hist_array =
threadalloc(bufsize, node);
			if (stat->hist_array ==
NULL)
				fatal("failed to
allocate histogram of size %d on node %d\n",
					histogram,
i);
			memset(stat->hist_array,
0, bufsize);
		}

		if (verbose) {
			int bufsize =
VALBUF_SIZE * sizeof(long);
			stat->values =
threadalloc(bufsize, node);
			if (!stat->values)
				goto outall;
			memset(stat->values, 0,
bufsize);
			par->bufmsk =
VALBUF_SIZE - 1;
		}

		par->prio = priority;
		 if (priority && (policy ==
SCHED_FIFO || policy == SCHED_RR))
			par->policy = policy;
```

```
                else {
                        par->policy =
SCHED_OTHER;
                        force_sched_other = 1;
                }
                if (priospread)
                        priority--;
                par->clock =
clocksources[clocksel];
                par->mode = mode;
                par->timermode = timermode;
                par->signal = signum;
                par->interval = interval;
                if (!histogram) /* same
interval on CPUs */
                        interval += distance;
                if (verbose)
                        printf("Thread %d
Interval: %d\n", i, interval);
                par->max_cycles = max_cycles;
                par->stats = stat;
                par->node = node;
                switch (setaffinity) {
                case AFFINITY_UNSPECIFIED: par-
>cpu = -1; break;
                case AFFINITY_SPECIFIED: par-
>cpu = affinity; break;
                case AFFINITY_USEALL: par->cpu
= i % max_cpus; break;
                }
                stat->min = 1000000;
                stat->max = 0;
                stat->avg = 0.0;
                stat->threadstarted = 1;
                status = pthread_create(&stat-
>thread, &attr, timerthread, par);
                if (status)
                        fatal("failed to create
thread %d: %s\n", i, strerror(status));

        }

        while (!shutdown) {
                char lavg[256];
                int fd, len, allstopped = 0;
                static char *policystr = NULL;
                static char *slash = NULL;
                static char *policystr2;

                if (!policystr)
                        policystr =
policyname(policy);

                if (!slash) {
                        if (force_sched_other) {
                                slash = "/";
                                policystr2 =
policyname(SCHED_OTHER);
                        } else
                                slash =
policystr2 = "";
                }
                if (!verbose && !quiet) {
                        fd =
open("/proc/loadavg", O_RDONLY, 0666);
                        len = read(fd, &lavg,
255);
                        close(fd);
                        lavg[len-1] = 0x0;
                        printf("policy: %s%s%s:
loadavg: %s          \n\n",
                                policystr, slash,
policystr2, lavg);
                }

                for (i = 0; i < num_threads;
i++) {

        print_stat(parameters[i], i, verbose);
                        if(max_cycles &&
statistics[i]->cycles >= max_cycles)
                                allstopped++;
                }

                usleep(10000);
                if (shutdown || allstopped)
                        break;
                if (!verbose && !quiet)
                        printf("\033[%dA",
num_threads + 2);

                if (refresh_on_max) {

        pthread_mutex_lock(&refresh_on_max_lock
);

        pthread_cond_wait(&refresh_on_max_cond,

&refresh_on_max_lock);

        pthread_mutex_unlock(&refresh_on_max_lo
ck);
                }
        }
        ret = EXIT_SUCCESS;

 outall:
        shutdown = 1;
        usleep(50000);

        if (quiet)
                quiet = 2;
        for (i = 0; i < num_threads; i++) {
                if (statistics[i]-
>threadstarted > 0)

        pthread_kill(statistics[i]->thread,
SIGTERM);
                if (statistics[i]-
>threadstarted) {

        pthread_join(statistics[i]->thread,
NULL);
                        if (quiet && !histogram)

        print_stat(parameters[i], i, 0);
                }
                if (statistics[i]->values)

        threadfree(statistics[i]->values,
VALBUF_SIZE*sizeof(long), parameters[i]-
>node);
        }

        if (histogram) {
                print_hist(parameters,
num_threads);
                for (i = 0; i < num_threads;
i++)

        threadfree(statistics[i]->hist_array,
histogram*sizeof(long), parameters[i]->node);
        }

        if (tracelimit) {
                print_tids(parameters,
num_threads);
                if (break_thread_id) {
                        printf("# Break thread:
%d\n", break_thread_id);
                        printf("# Break value:
%llu\n", (unsigned long
long)break_thread_value);
```

```
                }
        }


        for (i=0; i < num_threads; i++) {
                if (!statistics[i])
                        continue;
                threadfree(statistics[i],
sizeof(struct thread_stat), parameters[i]-
>node);
        }

 outpar:
        for (i = 0; i < num_threads; i++) {
                if (!parameters[i])
                        continue;
                threadfree(parameters[i],
sizeof(struct thread_param), parameters[i]-
>node);
        }
 out:
        /* ensure that the tracer is stopped */
        if (tracelimit)
                tracing(0);


        /* close any tracer file descriptors */
        if (trace_fd >= 0)
                close(trace_fd);

        if (enable_events)
                /* turn off all events */
                event_disable_all();

        /* turn off the function tracer */
        fileprefix = procfileprefix;
        if (tracetype)
                setkernvar("ftrace_enabled",
"0");
        fileprefix = get_debugfileprefix();

        /* unlock everything */
        if (lockall)
                munlockall();

        /* Be a nice program, cleanup */
        if (kernelversion < KV_26_33)
                restorekernvars();

        /* close the latency_target_fd if it's
open */
        if (latency_target_fd >= 0)
                close(latency_target_fd);

        exit(ret);
}
```