Universitatea “Politehnica” din Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei

Algoritmi de gestiune de procese concurente în sisteme distribuite cu implementare în nucleele sistemelor de operare

Lucrare de disertaţie

prezentată ca cerinţă parţială pentru obţinerea titlului de
Master în domeniul *Inginerie Electronică şi Telecomunicaţii*
programul de studii de masterat *Ingineria Informaţiei şi a Sistemelor de Calcul*

Conducător ştiinţific — *Conf. dr. ing. Ștefan STĂNCESCU*

Absolvent — *ing. Marian Adrian CEPOI*

2016

Anexa 2

Universitatea "Politehnica" din Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei
Departamentul : Electronică Aplicată și Ingineria Informației

Aprobat Coordonator program master IISC:

Prof.dr.ing. Radu DOGARU

TEMA LUCRĂRII DE DIZERTAŢIE
a masterandului CEPOI M.D. Marian Adrian, IISC

1. Titlul temei: **Algoritmi de gestiune de procese concurente în sisteme distribuite cu implementare in nucleele sistemelor de operare**

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

Se vor compara performanțele sistemelor de gestiune de procese în funcție de algoritmul de gestiune a concurentei și a localizării în zona protejată sau în zona de nucleu al sistemelor de operare.

Se vor folosi sisteme de operare linux curente (Raspbian, etc.) si mediu de dezvoltare Raspberry PI. Se cor simula scenarii cu seturi de procese concurente in mai multe configuratii si caracteristici (nr de procese, dimensiuni procese, durate de viata, frecvente de aparitie/disparitie, etc.).

Se vor obține concluzii cu privire la metode de acces concurent la resurse comune.

3. Proprietatea intelectuală asupra proiectului aparţine: *UPB*

4. Locul de desfăşurare a activităţii: *UPB*

5. Realizarea practică rămâne în proprietatea: *UPB*

6. Data eliberării temei: *10.12.2015*

CONDUCĂTOR LUCRARE:

Conf. dr. ing. Ştefan STĂNCESCU

STUDENT:

Marian Adrian CEPOI

Anexa 5

Declarație de onestitate academică

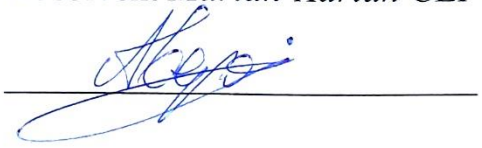
Prin prezenta declar că lucrarea cu titlul "*Algoritmi de gestiune de procese concurente în sisteme distribuite cu implementare în nucleele sistemelor de operare*", prezentată în cadrul Facultății de Electronică, Telecomunicații și Tehnologia Informației a Universității "Politehnica" din București ca cerință parțială pentru obținerea titlului de *Master* în domeniul *Inginerie electronică și telecomunicații*, programul de studii *Ingineria informației și a sistemelor de calcul* este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare.

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

București, *23.06.2016*

Absolvent *Marian-Adrian CEPOI*



CUPRINS

Lista figurilor

Lista acronimelor

POSIX	Portable Operating System Interface / Interfață Portabilă a Sistemului de Operare
CFS	Complet Scheduler Fair / Planificator Complet Echitabil
FIFO	First In First Out / Primul Intrat Primul Ieșit
RR	Round Robin
GPIO	General Purpose Input/Output / Scop General I/O

INTRODUCERE

Lucrarea de față își propune studierea algoritmilor de gestiune de procese concurente implementați în nucleul sistemului de operare Linux.

Linux a luat cu asalt lumea calculatoarelor prin oferirea unui mediu de operare gratuit și acces la codul sursă, ajutat fiind de restricțiile impuse de sistemele de operare și aplicațiile brevetate ce nu permit dezvoltatorilor să aducă o îmbunătățire a acestora.

Atât prezentul cât și viitorul procesării îl reprezintă sistemele multi-nucleu, în care procesarea se face prin fire de execuție, ceea ce duce la scăderea semnificativă a timpului de execuție al aplicațiilor. Totuși, această îmbunătățire a timpului de execuție implică necesitatea planificării cât mai bune a firelor de execuție.

Prezenta lucrare dorește a sublinia performanțele diferitelor metode de planificare a execuției proceselor dintr-un sistem de operare Linux și a evidenția posibilitățile de îmbunătățire în vederea obținerii unei performanțe mai mari.

Proiectul a fost realizat folosind tendințele revoluției digitale, reprezentate de sistemele dedicate, platforme ce oferă performanțe și mobilitate mare la costuri mici.

În prima parte s-au trecut în revistă aspecte importante ce țin de arhitectura sistemului de operare linux, oferind o privire de ansamblu asupra subiectului luat în discuție.

În cea de-a doua parte s-au explicat principiile ce stau la baza planificării execuției, precum și diferitele metode folosite în scopul acestei planificări.

Partea principală a lucrării o reprezintă *Capitolul 3,* în care sunt descrise elementele principale ale prezentului studiu. S-a dezvoltat un instrument prin intermediul căruia se generează fire de execuție cu anumiți parametri, oferind rezultate asupra performanțelor algoritmilor studiați. De asemenea, s-au realizat o serie de teste pe baza cărora s-au tras anumite concluzii.

Lucrarea se încheie prin prezentarea principalelor idei și concluzii referitoare la acest studiu.

I. MEDIUL DE OPERARE LINUX

Linux este un fenomen al zilelor noastre, reușind să acapareze segmentul sistemelor de operare pentru servere și super-calculatoare cu un procent covârșitor de 98,8% (vezi *Figura 1.1*). Acest lucru se datorează caracterului open-source, fapt ce permite dezvoltatorilor din întreaga lume să contribuie la îmbunătățirea și dezvoltarea acestuia.

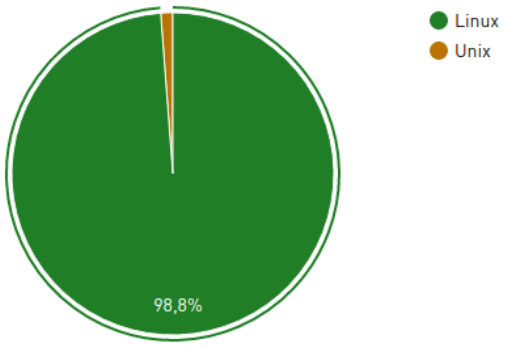


Figura 1.1. Statistica sistemelor de operare pentru supercalculatoare (Noiembrie 2015).
Sursa: http://www.top500.org/statistics/list/ , accesat la data 04.06.2016.

Astăzi, mai mult de 98 % din super-calculatoarele din întreaga lume (inclusiv întregul top 10), peste 84 % din toate smartphone-urile, multe milioane de calculatoare personale, în jur de 70 % din toate serverele de web, un procent important de tablete, și multe alte aparate (DVD-playere, mașini de spălat, modemuri DSL, routere, etc.) rulează Linux, fiind de departe cel mai frecvent utilizat sistem de operare din lume. [5]

Inițial, Linux a fost dezvoltat în anul 1991 de către Linus Torvalds ca un sistem de operare pentru calculatoarele personale compatibile IBM, bazate pe microprocesorul Intel 80386. Activitatea lui Linus Torvalds nu se oprește aici, el rămânând implicat în îmbunătățirea sistemului Linux, menținându-l la curent cu diverse dezvoltări hardware și coordonând activitățile a sute de dezvoltatori din întreaga lume. Contribuția semnificativă a dezvoltatorilor constă în principal în realizarea disponibilității Linux și pe alte arhitecturi precum: Alpha de la Hewlett-Packard, Itanium a companiei Intel, AMD64 a companiei AMD, PowerPC și zSeries a companiei IBM. [1]

Una dintre caracteristicile notabile ale sistemului UNIX este aceea că dezvoltarea sa nu a fost controlată de un singur furnizor sau organizație. Mai degrabă, multe grupuri atât comerciale cât și non-comerciale, au contribuit la evoluția sa. Această istorie a dus la mai multe caracteristici inovatoare, însă a avut, de asemenea, o consecință negativă, aceea că implementările UNIX s-au separat de-a lungul timpului, astfel încât dezvoltarea aplicațiilor care să funcționeze pe toate implementările UNIX a devenit din ce în ce mai grea. [2]

Avantajele Linux: [3]

- Este gratuit;
- Este portabil, putând rula pe orice platformă hardware;
- Implică un grad ridicat de securitate;
- Este scalabil, putând rula atât pe sisteme dedicate (exemplu: Raspberry PI) care implică performanțe hardware reduse, cât și în supercalculatoare alcătuite dintr-un număr variabil de noduri de procesare;
- Timpul de rezolvare și fixare a bug-urilor din sistemul de operare sau din aplicațiile Linux este foarte scăzut, uneori de ordinul orelor, fapt datorat miilor de oameni din întreaga lume care au contribuit la dezvoltarea și testarea acestora și care pot găsi și rezolva destul de rapid problemele apărute.

Dezavantajele Linux: [3]

- Poate fi greu de utilizat și confuz pentru începători;
- Fiind gratuit, nu ridică un foarte mare grad de încredere utilizatorilor.

Linux vine sub o varietate de distribuții din ce în ce mai centrate pe utilizator. O distribuție Linux este o colecție de aplicații (de obicei, open source) având la bază un nucleu Linux, cele mai populare distribuții fiind: Ubuntu (un produs al companiei Canonical), Fedora, Mint, Debian, Red Hat, CentOS, etc (vezi *Figura 1.4*).

O distribuție (sau pe scurt, *distro*) poate grupa aplicații de tip server, instrumente de gestionare a sistemului, documentație și multe alte aplicații într-un depozit central de aplicații securizat. O distribuție își propune să ofere un aspect comun, o administrare sigură și ușoară a aplicațiilor și adesea, un scop operațional specific. [5]

Întrucât există o mare varietate de distribuții Linux, iar dezvoltatorii acestora nu controlează conținutul lor, nu există un "standard" Linux. Fiecare distribuitor Linux se bazează de obicei pe o versiune stabilă de nucleu disponibilă la un anumit moment, cu o serie de patch-uri aplicate.
Aceste patch-uri oferă de obicei caracteristici care, într-o măsură mai mare sau mai mică, sunt capabile să ofere o diferențiere competitivă pe piață. În unele cazuri, aceste patch-uri sunt ulterior acceptate în nucleul principal. De fapt, unele caracteristici noi ale nucleului au fost inițial dezvoltate de o companie de distribuție, și au apărut în distribuția lor, înainte de a fi în cele din urmă integrate în nucleul principal. [2]

Figura 1.2. Distribuții Linux.
Sursa: http://www.picateshackz.com/2015/04/linux-powerful-distros-for-hacking-or.html, accesat la data 04.06.2016.

1.1. Arhitectura

Sistemul de operare Linux are următoarele componente principale:

- **Nucleul** (sau *kernel*-ul) este componenta de bază a sistemului de operare Linux care gestionează comunicația între dispozitivele hardware și componentele și aplicațiile software. De asemenea, acesta gestionează resursele sistemului cum ar fi: procesorul, memoria și interfața de rețea și este responsabil pentru majoritatea activităților sistemului de operare. Este alcătuit din diferite module și interacționează direct cu nivelul hardware, oferind un grad de abstractizare prin ascunderea detaliilor de nivel fizic; [4]
- **Bibliotecile de sistem** care conțin metode și funcții de creare și manipulare a proceselor, de tratare și gestionare a accesului la diferite fișiere, precum și instrumente prin intermediul cărora se pot dezvolta și implementa diverse aplicații și funcționalități noi sistemului de operare; [4]

- **Utilitarele de sistem** ce sunt construite pe baza bibliotecilor de sistem și permit utilizatorilor să administreze sistemul: să gestioneze procesele de sistem, executarea aplicațiilor, navigarea în sistemul de fișiere, configurările de rețea și internet, etc. [4]

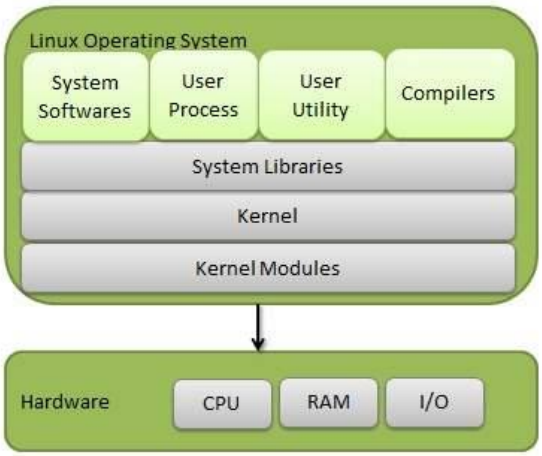


Figura 1.3. Componentele Linux.
Sursa: http://www.tutorialspoint.com/operating_system/os_linux.htm, accesat la data 04.06.2016.

Linux oferă tot ceea ce este necesar în procesul de dezvoltare: compilatoare, biblioteci, instrumente de dezvoltare și depanare. Aceste pachete vin cu fiecare distribuție Linux standard, iar cele mai noi dezvoltări includ suport pentru accelerare 3D, eficientizarea procesului de actualizare a sistemului. [3]

Arhitectura Linux este alcătuită următoarele niveluri:

- **Nivelul fizic**: cuprinzând toate dispozitivele periferice (memorie internă - RAM, memorie externă - HDD, procesor, etc.);
- **Nucleul**: abstractizează nivelul fizic și realizează comunicarea acestuia cu componentele de pe nivelurile superioare;
- **Interpretorul Shell**: reprezintă interfața cu nucleul și mapează comenzile utilizatorului în funcții nucleu;
- **Utilitare**: instrumente software prin intermediul cărora unul sau mai mulți utilizatori au acces la toate funcționalitățile sistemului de operare.

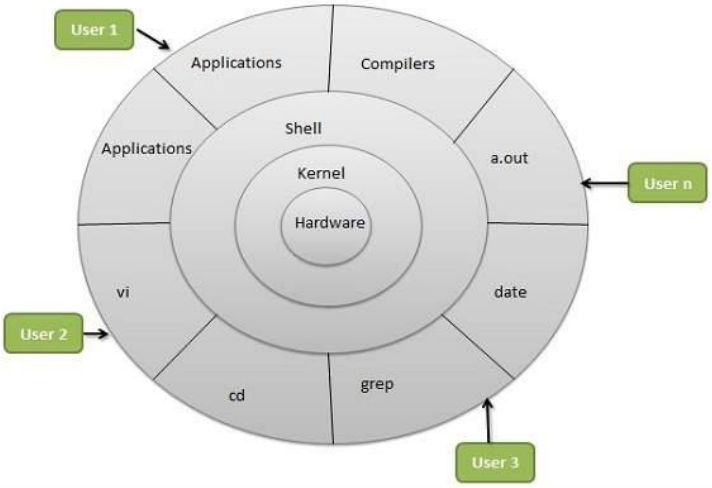


Figura 1.4. Arhitectura Linux.
Sursa: http://www.tutorialspoint.com/operating_system/os_linux.htm, accesat la data 04.06.2016.

1.2. Nucleul Linux

Nucleul sau kernel-ul reprezintă un nivel intermediar între nivelul fizic (hardware-ul) și nivelul aplicație (software). Rolul său este de a transfera cererile aplicațiilor către componentele hardware și de a acționa ca un driver de nivel scăzut pentru a adresa dispozitivele și componentele sistemului. [6]

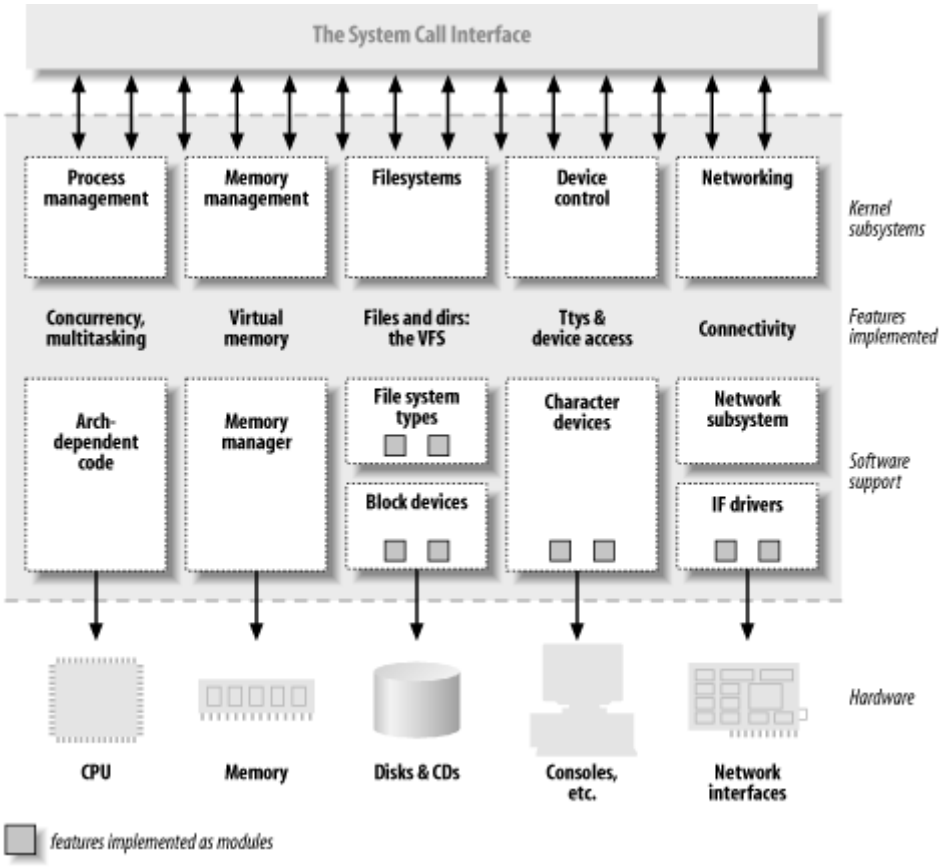


Figura 1.5. Structura nucleului Linux. [7]

Nucleul are în general patru responsabilități de bază: [4]

- Gestiunea dispozitivelor;
- Gestiunea memoriei;
- Gestiunea proceselor;
- Tratarea apelurilor de sistem.

Gestiunea dispozitivelor

Dispozitivele (mouse, tastatură, monitor, CD, etc.) atașate la calculator, permit comunicarea și partajarea informațiilor între calculator și lumea exterioară. Nucleul furnizează programe cu o interfață care standardizează și simplifică accesul la dispozitive și în același timp, coordonează accesul prin mai multe procese la fiecare dispozitiv în parte. [2]

Deoarece fiecare dispozitiv funcționează în mod diferit, este necesar ca nucleul să știe ce poate face dispozitivul și cum să adreseze și să manipuleze fiecare dispozitiv, astfel încât întregul sistem să funcționeze corespunzător. Aceste informații sunt stocate în driver-ul dispozitivelor. Fără un astfel de driver, nucleul nu cunoaște dispozitivul și, prin urmare, nu va fi în măsură să-l controleze. [4]

De asemenea, pe lângă driver-ele dispozitivelor, nucleul gestionează și comunicarea între acestea. Astfel, se reglementează accesul la componentele partajate, astfel încât toate driver-ele să

poată funcționa corespunzător. În același timp, comunicarea trebuie să respecte reguli stricte, iar nucleul să se asigure că aceste reguli sunt respectate. [4]

Gestiunea memoriei

În timp ce memoriile calculatoarelor sunt foarte mari în conformitate cu standardele de acum un deceniu sau două, dimensiunea aplicațiilor a crescut, de asemenea în mod corespunzător, astfel că, memoria fizică (RAM) rămâne o resursă limitată pe care nucleul trebuie să o împartă între procesele sistemului într-o manieră echitabilă și eficientă. La fel ca majoritatea sistemelor de operare moderne, Linux utilizează gestionarea memoriei virtuale, o tehnică care conferă două avantaje principale: [2]

- Procesele sunt izolate unul de altul și de nucleu, astfel încât un proces nu poate citi sau modifica memoria altui proces sau a nucleului;
- Doar o parte a unui proces trebuie să fie păstrată în memorie, reducând astfel cerințele de memorie ale fiecărui proces și permițând mai multor procese să existe în memoria RAM simultan. Acest lucru conduce la o mai bună utilizare a procesorului, deoarece crește probabilitatea ca, în orice moment în timp, să existe cel puțin un proces pe care procesorul să îl poată executa. [2]

Gestiunea proceselor

Pentru a se asigura că fiecare proces primește suficient timp de procesor, nucleul acordă priorități proceselor și oferă fiecăruia dintre acestea o anumită perioadă de timp de procesor înainte de a se opri procesul și de a continua execuția următorului proces.

Gestiunea proceselor se ocupă nu numai cu acordarea de timp de procesor (numită *planificare* sau *scheduling*), ci și cu privilegiile de securitate, informațiile privind proprietatea proceselor, comunicarea între procese și multe altele. [4]

Tratarea apelurilor de sistem

În cele din urmă, pentru ca un nucleu să lucreze efectiv într-un sistem, acesta trebuie să asigure mijloacele prin care sistemul și programatorul să se controleze și să dea sau să primească informații pe baza cărora pot fi luate noi decizii. Prin intermediul apelurilor de sistem, un programator poate scrie aplicații care interoghează nucleul pentru informații sau cere nucleului să efectueze o anumită sarcină (de exemplu, să manipuleze un dispozitiv și să returneze rezultatul). Desigur, astfel de apeluri trebuie să se utilizeze în condiții de siguranță, astfel încât programatorii rău intenționați să nu poată periclita funcționarea sistemului, cu un apel de sistem bine-elaborat. [4]

Codul nucleului se execută într-un mod privilegiat special denumit *modul nucleu* (sau *kernel mode*) cu acces deplin la toate resursele calculatorului. Acest cod reprezintă un singur proces, ce se execută în spațiul unic de adrese și nu necesită nici o schimbare de context. Prin urmare, este foarte eficient și rapid. Nucleul rulează fiecare proces, furnizează servicii de sistem pentru procese și oferă acces protejat la dispozitive.

Bibliotecile de sistem, aplicațiile utilizatorilor și alte aplicații de sistem funcționează în *modul utilizator* (sau *user mode*), care nu are acces la hardware-ul sistemului și codul nucleului.

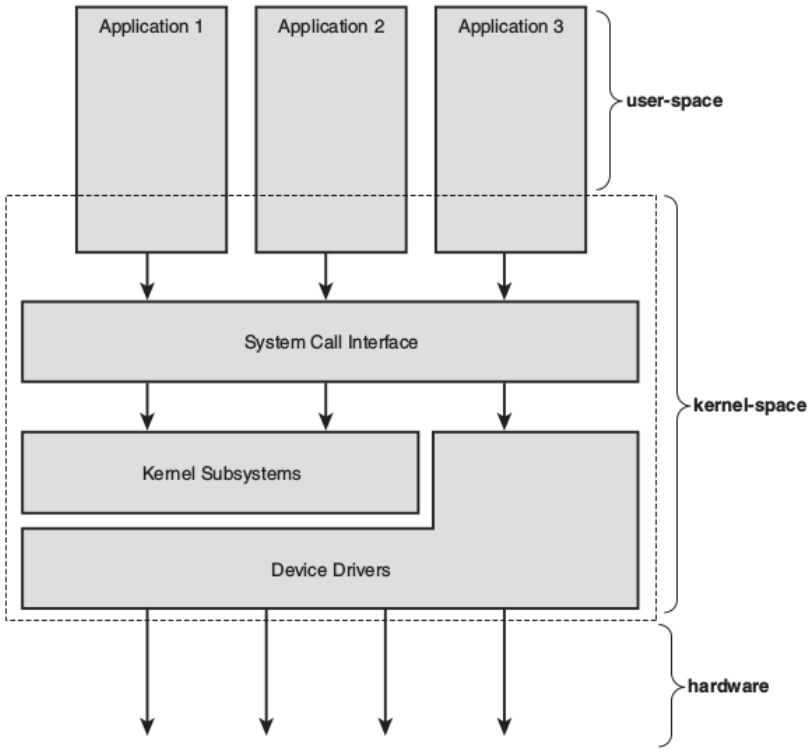


Figura 1.6. Relația dintre aplicații, nucleu și nivelul fizic. [8]

1.3. Procese și fire de execuție

Un proces este un program (cod obiect stocat pe un anumit mediu), în execuție. Procesele sunt, cu toate acestea, mai mult decât doar executarea codului programului (adesea numit *secțiunea de text* în Unix). Acestea includ, de asemenea, un set de resurse, cum ar fi fișierele deschise și semnalele de așteptare, date interne ale nucleului, starea procesorului, un spațiu de adrese de memorie cu una sau mai multe mapări de memorie, unul sau mai multe fire de execuție, precum și o secțiune de date care conține variabile globale. [8]

Procesele sunt asemenea ființelor umane: sunt generate, au o viață mai mult sau mai puțin semnificativă, opțional pot genera unul sau mai multe procese copil și într-un final mor, cu diferența că procesele au un singur părinte. [1]

Procesele sunt rezultatul viu al programelor ce rulează în sistem. Nucleul trebuie să gestioneze toate aceste detalii în mod eficient și transparent. [8]

Din punctul de vedere al nucleului, procesele sunt entitățile între care nucleul trebuie să împartă diferitele resurse ale calculatorului. Din resursele care sunt limitate (cum ar fi memoria), nucleul alocă inițial o anumită cantitate de resurse unui proces și ajustează această alocare pe durata executării procesului ca răspuns la cerințele procesului și la cererea globală a sistemului pentru acea resursă. Atunci când se termină procesul, toate aceste resurse sunt eliberate pentru a fi reutilizate de către alte procese. Alte resurse, cum ar fi procesorul și lățimea de bandă a rețelei, pot fi reînnoite, dar trebuie să fie împărțite echitabil între toate procesele. [2]

Un proces este împărțit logic în următoarele părți, cunoscute sub numele de *segmente*:

- *Text*: segment ce conține instrucțiunile programului;
- *Data*: segment ce conține variabilele statice folosite de program;

- *Heap*: reprezintă o zonă din care programele pot aloca dinamic memorie suplimentară;
- *Stack*: sau *segmentul de stivă*, este o porțiune de memorie care crește și se micșorează pe măsură ce funcțiile sunt apelate și returnate și care este utilizat pentru alocarea de memorie pentru variabilele locale și informațiile despre apelul funcțiilor. [2]

Atunci când se creează un proces, este aproape identic cu părintele său. Acesta primește o copie (logică) a spațiului de adrese al părintelui și execută același cod ca părinte, care începe la următoarea instrucțiune după apelul de sistem de creare a procesului. Cu toate că părintele și copilul pot partaja paginile care conțin codul programului (segmentul *text*), ele au copii separate ale datelor (segmentele de *stivă* și *heap*), astfel încât modificările aduse de copil într-o locație de memorie sunt invizibile pentru părinte (și vice-versa). [1]

Un program în sine nu este un proces. Un proces este un program activ și resurse conexe. Într-adevăr, două sau mai multe procese pot executa același program. De fapt, două sau mai multe procese pot exista partajând diverse resurse, cum ar fi fișiere deschise sau un spațiu de adrese. [8]

Un proces își începe execuția când, în mod evident, este creat. În Linux, acest lucru are loc prin intermediul apelului de sistem *fork()*, care creează un nou proces prin duplicarea celui existent.

Procesul care apelează *fork()* este părintele, în timp ce noul proces este copilul. Procesul părinte își reia execuția și copilul începe execuția în același loc: unde returnează apelul *fork()*. [8][2]

De multe ori, imediat după *fork*, este de dorit să se execute un program nou, diferit. Familia de apeluri de funcții *exec()* creează un nou spațiu de adrese și încarcă noul program în acest spațiu. [8]

În cele din urmă, un program își termină execuția prin apelul de sistem *exit()*. Această funcție încheie procesul și eliberează toate resursele sale. Un proces părinte poate solicita informații despre starea unui copil terminat prin apelul de sistem *wait4()*, care permite unui proces să aștepte terminarea unui proces specific. [8][2]

Un sistem de operare trebuie să aloce resursele hardware printre cerințele concurente potențiale ale mai multor procese. În cazul procesorului, resursa care urmează să fie alocată este timpul de execuție a procesorului, iar modul de alocare este planificarea (*scheduling*). In acest fel, planificatorul este componenta sistemului de operare responsabilă cu acordarea dreptului de acces procesorului la o listă de mai multe procese gata de execuție. Această idee este ilustrată în diagrama cu cinci stări din *Figura 1.7*. [9]

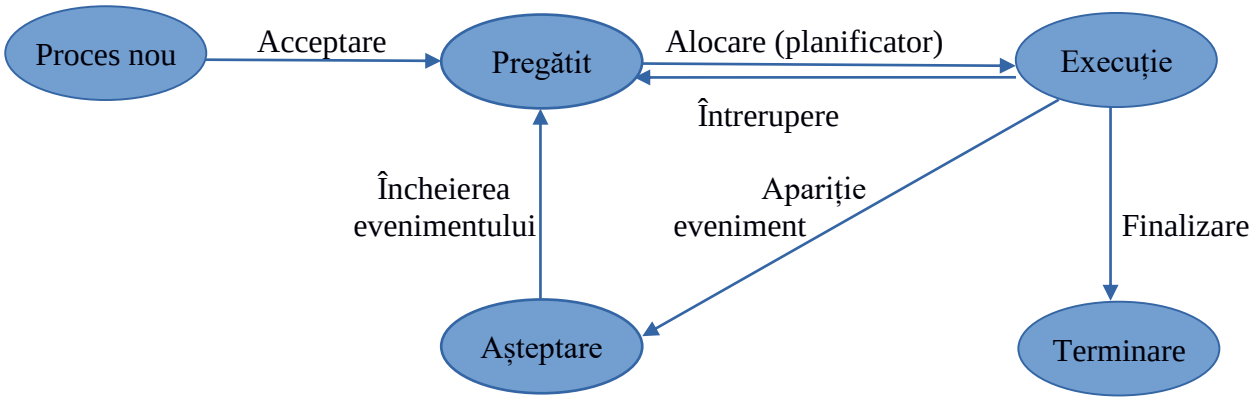


Figura 1.7. Ciclul de viață al unui proces. [9]

Un nou proces este creat prin copierea atributelor procesului curent, astfel încât partajează resurse cum ar fi: fișiere și memoria virtuală. Atunci când cele două procese partajează aceeași memorie virtuală, ele funcționează ca fire de execuție ale aceluiași proces. Cu toate acestea, nu este definită separat o structură de date pentru un fir de execuție. [11]

Termenul *fir de execuție* (sau *thread*) provine din conceptul unui singur fir de execuție, adică o cale liniară de parcurgere a codului. POSIX[1] definește firul de execuție ca fiind resursa necesară reprezentării unei singure căi de execuție a codului unui program. Firele de execuție au devenit o modalitate utilizată pe scară largă de implementare a concurenței în cadrul aplicațiilor. Un proces conține unul sau mai multe fire de execuție. [10]

În cazul implementărilor UNIX moderne, fiecare proces poate avea unul sau mai multe fire de execuție care au în comun aceeași memorie virtuală, precum și o serie de alte atribute. Fiecare fir execută același cod de program și partajează aceleași segmente de date și heap. Cu toate acestea, fiecare fir de execuție are propriul segment de stivă care conține variabile locale și informații legate de apelurile de funcții. [2]

Avantajul principal al utilizării firelor de execuție este acelea că firele de execuție fac ușor schimb de date între ele (prin variabile globale). Mai mult decât atât, o aplicație pe mai multe fire de execuție (*multithread*) poate profita în mod transparent de posibilitățile de procesare paralelă ale sistemelor multiprocesor. [2]

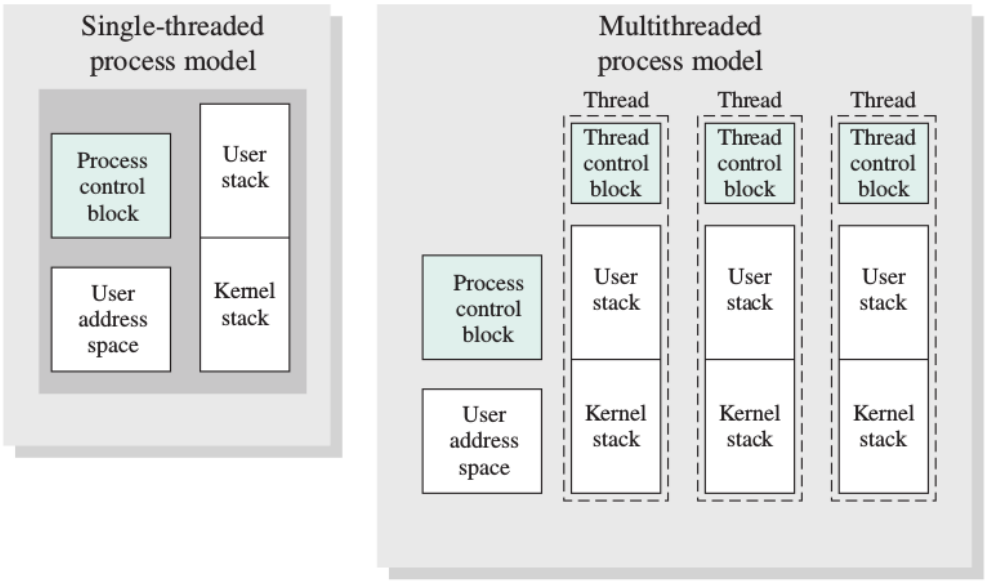


Figura 1.8. Modele ale proceselor cu unul și mai multe fire de execuție. [11]

Figura 1.8 ilustrează distincția între firele de execuție și procese din punct de vedere al gestionării proceselor. În modelul proceselor cu un singur fir de execuție, reprezentarea procesului include blocul său de control și spațiul de adrese, precum și stivele de nivel utilizator și nucleu pentru a gestiona comportamentul de apelare/revenire al executării procesului. În timp ce procesul se execută, acesta controlează registrele procesorului. Conținutul acestor registre este salvat în momentul în care procesul nu se execută. Într-un mediu multithread, există tot un singur bloc de control al procesului și un singur spațiu de adrese asociat procesului, însă acum există stive separate pentru fiecare fir de execuție în parte, precum și un bloc de control separat pentru fiecare fir de execuție conținând valori ale registrului, prioritatea și alte informații referitoare la starea firului de execuție. [11]

1 POSIX (Portable Operating System Interface) - este o familie de standarde specificate de către IEEE Computer Society pentru menținerea compatibilității între sistemele de operare. POSIX definește interfața de programare a aplicațiilor (API), împreună cu interpretoarele în linie de comandă și interfețele utilitarelor, pentru compatibilitate software cu diferite variante de Unix și alte sisteme de operare. Sursa: https://en.wikipedia.org/wiki/POSIX, accesat la data 05.06.2016.

II. PLANIFICAREA PROCESELOR

Planificatorul proceselor reprezintă subsistemul nucleului care decide ce proces rulează, la ce moment de timp și pentru cât timp. Planificatorul proceselor (sau pur și simplu planificatorul) divide resursa finită reprezentând timpul de procesor între toate procesele care rulează într-un sistem la un moment dat. Planificatorul reprezintă baza unui sistem de operare multitasking, cum ar fi Linux. Hotărând care proces rulează la momentul următor, planificatorul este responsabil pentru utilizarea eficientă a resurselor sistemului și pentru a oferi utilizatorilor impresia că se execută mai multe procese simultan. [8]

Planificatorul Linux se bazează pe tehnica partajării timpului: mai multe procese rulează în modul "multiplexare în timp", deoarece timpul procesorului este împărțit în bucăți, câte una pentru fiecare proces în execuție. Desigur, un singur procesor poate rula doar un singur proces la un moment dat. În cazul în care un proces care se execută nu este terminat atunci când expiră cuanta sa de timp, se poate realiza o comutare între procese. Partajarea timpului se bazează pe întreruperi ale contorului de timp și este astfel, transparent proceselor. [1]

Un sistem de operare multitasking este acela care poate intercala simultan executarea mai multor procese. Pe mașinile multiprocesor, o astfel de funcționalitate permite proceselor să ruleze efectiv simultan (în paralel), pe procesoare diferite. Pe fiecare tip de sistem (uniprocesor sau multiprocesor), este permisă trecerea proceselor în starea de blocare sau așteptare pană când este posibilă execuția acestora. În consecință, un sistem modern Linux poate avea mai multe procese în memorie, dar doar unul sau unele dintre acestea aflate în execuție. [8]

Linux distinge trei clase de procese:

- *Procesele interactive* - interacționează în mod constant cu utilizatorii lor și, prin urmare, consumă mult timp așteptând apariția unui eveniment. Atunci când primește o intrare, procesul trebuie să fie „trezit” rapid, sau utilizatorul va observa că sistemul nu răspunde. Varianța întârzierii trebuie să fie de asemenea limitată, sau utilizatorul va considera că este ceva în neregulă cu sistemul; [1] [12]
- *Procese discontinue* - nu au nevoie de interacțiunea cu utilizatorul și prin urmare, de multe ori se execută în fundal. Din moment ce astfel de procese nu au nevoie să fie foarte receptive, ele sunt adesea penalizate de planificator; [1] [12]
- *Procese în timp real* – prezintă cerințe foarte puternice de planificare. Astfel de procese nu ar trebui să fie blocate de procese cu prioritate mai mică. Acestea ar trebui să aibă un timp de răspuns scurt cu o variație minimă. [1] [12]

Procesele în Linux se găsesc în următoarele stări: [12]

- *RUNNING* - un proces care rulează la un moment dat;
- *INTERRUPTIBLE* - proces în așteptare (întrerupt), dar poate fi „trezit”;
- *UNINTERRUPTIBLE* - proces în așteptare (întrerupt), dar care nu poate fi „trezit”;
- *STOPPED* - proces oprit de un utilizator sau de un bloc de control;
- *ZOMBIE* – proces care și-a terminat execuția, dar care există încă în tabela de procese.

2.1. Politica de planificare

Nucleul Linux s-a dezvoltat rapid în ultimii ani, introducând o mulțime de noi funcționalități și aducând o îmbunătățire semnificativă în ceea ce privește fiabilitatea, flexibilitatea și performanța. De asemenea, Linux realizează inovații continue și dezvoltări în ceea ce privește planificarea proceselor, care este componenta cheie a oricărui sistem de operare. [13]

Planificatorul introdus în versiunea 1.2 a nucleului Linux utiliza o coadă circulară pentru gestionarea proceselor active, care funcționa cu o politică de planificare Round-Robin. Versiunea 2.2 a nucleului Linux a introdus ideea claselor de planificare. Începând cu versiunea 2.6, planificatorul denumit O(1), a devenit mult mai eficient și mult mai scalabil, dar a devenit greu de manipulat în nucleu și a fost în mod fundamental dificil de gestionat. Ingo Molnar, creatorul planificatorului O(1), a dezvoltat apoi *CFS (Complet Scheduler Fair)*, și l-a introdus în versiunea de nucleu 2.6.23. [13]

Dezvoltatorul planificatorului CFS, Ingo Molnar, îl rezumată într-o singură propoziție: *"CFS, practic modelează un procesor multitasking precis ideal pe un hardware real"*. Un procesor ideal pentru multitasking este un procesor care poate rula mai multe procese în același timp (în paralel), dând fiecăruia o parte egală a puterii de calcul a procesorului (nu timp, ci putere de calcul). În cazul în care se execută un singur proces la un anumit moment de timp, acesta va primi 100% din puterea de calcul a procesorului. Cu două procese în execuție, fiecare ar avea exact 50% din puterea de calcul (în paralel). În mod similar, cu patru procese în execuție, fiecare ar obține 25% din puterea de calcul a procesorului și așa mai departe. Prin urmare, acest procesor ar fi "corect" pentru toate sarcinile care rulează în sistem (vezi *Figura 1.9*). [14]

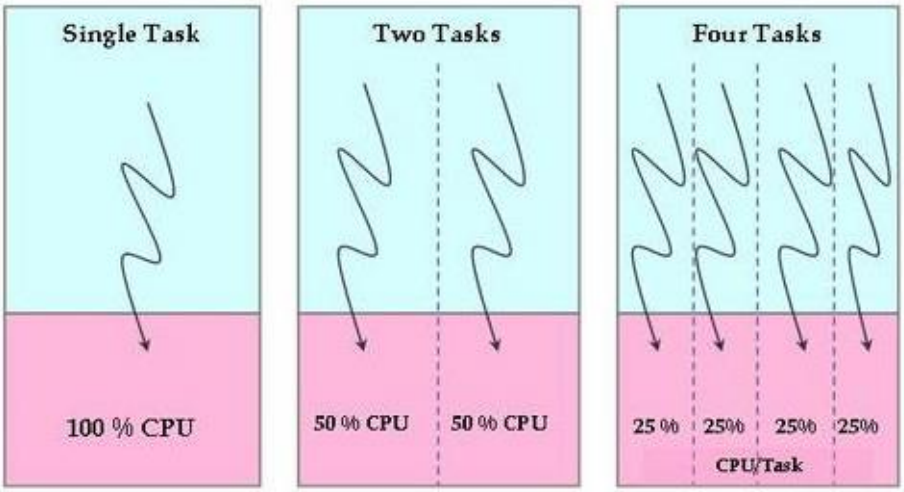


Figura 2.1. Procesare ideală. [14]

În mod evident, acest procesor ideal nu există, dar planificatorul CFS încearcă să imite un astfel de procesor în software. Pe un procesor real din lumea reală, doar un singur proces poate fi alocat unui procesor la un anumit moment de timp. Prin urmare, toate celelalte procese, așteaptă în această perioadă. Astfel că, procesul care rulează primește 100% din puterea de calcul, toate celelalte sarcini având 0% (vezi *Figura 1.10*). [14]

Politica de planificare a CFS încearcă să elimine această incorectitudine din sistem prin folosirea unui ceas la o fracțiune din viteza ceasului real al procesorului. Rata de creștere a acestui ceas se calculează prin împărțirea întregului timp (în nanosecunde), la numărul total de procese în așteptare, valoarea rezultată reprezentând cantitatea de timp de procesor la care fiecare proces are dreptul. [14]

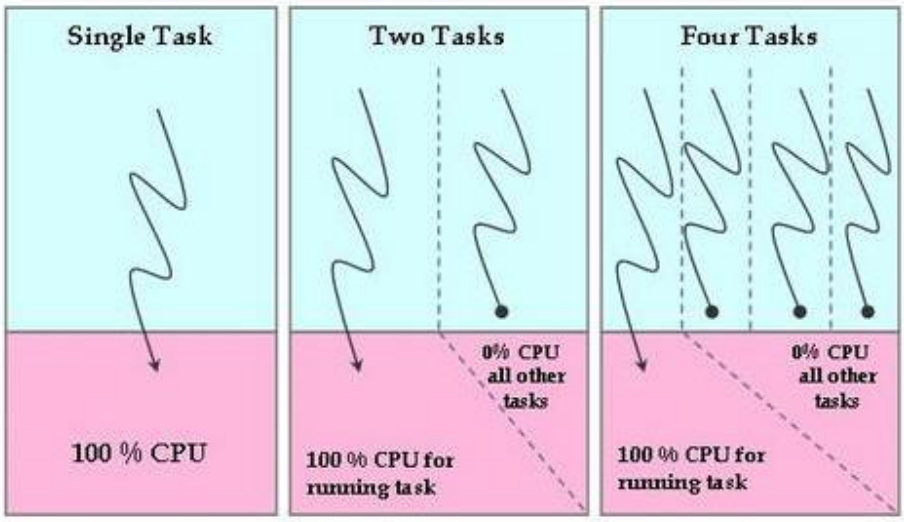


Figura 2.2. Procesare reală. [14]

Ideea principală a *CFS* este de a menține echilibrul (corectitudinea) timpului de procesor care este partajat de procesele active. Aceasta înseamnă că două procese care au aceeași proprietate, prioritatea și sumă calculată, își vor opri execuția aproape în același timp, în cazul în care se execută simultan. [13]

2.1.1. Priorități

Nucleul Linux implementează două intervale de prioritate separate. Primul interval este reprezentat de valoarea „*nice”*, ce semnifică un număr între -20 și +19, având ca valoare implicită 0. Valori mari ale parametrului *nice* corespund unei priorități mici. Procesele cu o valoare *nice* mai mică (prioritate mai mare) primesc o proporție mai mare a timpului procesorului, comparativ cu procesele cu o valoare *nice* mai mare (prioritate mai mică). Valorile *nice* reprezintă gama standard de priorități utilizate în toate sistemele Unix, deși diferite sisteme Unix le aplică în moduri diferite, reflectându-se asupra algoritmilor de planificare. [8]

Al doilea interval de prioritate reprezintă prioritatea în timp real. Valorile sunt configurabile, dar ținând cont de intervalul implicit 0 - 99, inclusiv. Vis-a-vis de valorile *nice*, valorile mari ale priorității de timp real corespund unei priorități efective mici. Toate procesele de timp real au o prioritate mai mare decât procesele normale. Linux implementează prioritățile în timp real, în conformitate cu standardele Unix relevante, în mod specific POSIX.1b. [8]

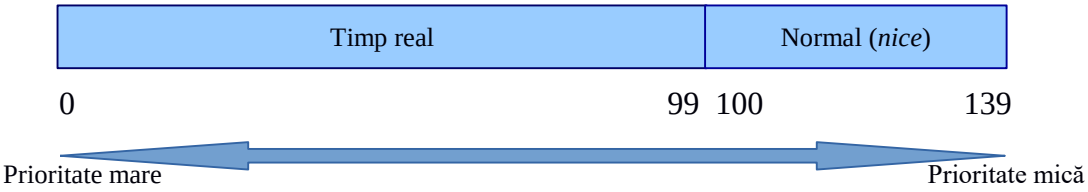


Figura 2.3. Intervalele de prioritate în Linux.

2.1.2. Descriptori de procese

Nucleul Linux stochează lista proceselor din sistem într-o listă circulară dublu-înlănțuită denumită „*task list*” (vezi *Figura 2.4*). [8]. Fiecare element din respectiva listă reprezintă un descriptor de proces ce deține toate informațiile despre respectivul proces și care este o structură în limbajul de programare C de tipul *struct task_struct*. Această structură este definită în fișierul sursă *<linux/sched.h>:*

```
struct task_struct {
      volatile long state; /*starea: -1 neexecutabilă, 0 executabilă, >0
                                          Oprit */
      void *stack;

      ...

      unsigned int flags; /* fanioanele procesului */

      ...

      int prio; /* prioritatea procesului */
      unsigned int policy; /* algoritmul de planificare */

      ...

      struct mm_struct *mm;

      ...

      pid_t pid;  /* ID-ul procesului */
      pid_t tgid; /* ID-ul de grup */
      ...
      struct task_struct __rcu *real_parent; /* procesul părinte */
      struct list_head children; /* lista copiilor */
      struct list_head sibling;  /* legătură în lista copiilor */

      ...
}
```

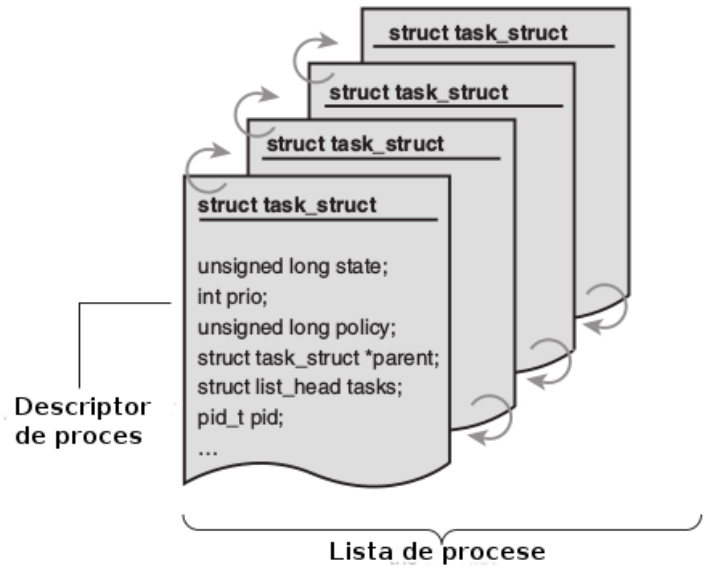


Figura 2.4. Lista descriptorilor de procese. [8]

2.2. Algoritmi de planificare

Algoritmii de planificare a proceselor trebuie să îndeplinească mai multe obiective: timp de răspuns scurt, rată bună de procesare a proceselor de fundal, evitarea înfometării proceselor, recalcularea cerințelor proceselor de mică și mare prioritate, și așa mai departe. [1]

Există diferiți algoritmi de planificare care au proprietăți diferite, iar alegerea unui anumit algoritm poate favoriza o anumită clasă de procese în defavoarea alteia. În scopul alegerii algoritmului într-o anumită situație, trebuie să luăm în considerare proprietățile diferiților algoritmi. Se pot găsi criterii pentru compararea algoritmilor de planificare care pot face o diferență substanțială în considerarea unui algoritm ca fiind cel mai bun. [9][15]

Dintre aceste criterii putem aminti: [9][15]

- *Utilizarea procesorului*. În mod normal, utilizarea procesorului poate varia de la 0% la 100%. Este de dorit ca procesorul să fie utilizat la capacitate maximă (100%) pe cât mai mult posibil;
- *Throughput*. O măsură a procesării este numărul de procese care sunt executate pe unitate de timp, numit *throughput* (sau *debit*). Pentru procesele lungi, această rată poate fi de un proces pe oră; pentru sarcini scurte, acesta poate fi de zece procese pe secundă;
- *Durata de viață*. Din punctul de vedere al unui anumit proces, cel mai important criteriu este timpul necesar executării acelui proces. Intervalul de timp de la momentul apariției unui proces, la momentul finalizării acestuia reprezintă d*urata de viață*. *Durata de viață* este suma timpului de așteptare pentru a intra în memorie, a timpului de așteptare în coada de așteptare și a timpului de execuție propriu-zisă. Este de dorit minimizarea duratei de viață în vederea reducerii timpului de așteptare al utilizatorului;
- *Timpul de așteptare*. Algoritmii de planificare nu afectează timpul de execuție al unui proces, ci numai timpul petrecut de un proces în coada de așteptare;
- *Timpul de răspuns*. Reprezintă timpul de la crearea procesului, la returnarea primului răspuns. *Timpul de răspuns* este în general limitat de viteza dispozitivului de ieșire.

Din punctul de vedere al gestionării execuției proceselor, se pot distinge două tipuri de algoritmi:

- *Preemptivi*: atunci când un proces se află în starea de execuție și apare o întrerupere, se poate opri execuția procesului în vederea tratării respectivei întreruperi, urmând a fi reluată după finalizarea evenimentului care a generat întreruperea. Această abordare este costisitoare din punct de vedere al planificării execuției, însă este avantajoasă în situațiile de blocaj; [16]
- *Non-preemptivi*: specifică faptul că nu este posibilă întreruperea execuției unui proces, sugerându-se în mod implicit așteptarea finalizării procesului. Acest lucru presupune costuri de planificare reduse, însă înlesnește apariția blocajelor. [16]

În nucleul Linux, algoritmii de planificare sunt definiți în următoarele fișiere sursă care se regăsesc în directorul *linux/kernel/sched* al nucleului:

- ***fair.c*** – în acest fișier sursă sunt definiți algoritmii *OTHER* (*SCHED_OTHER*), *BATCH* (*SCHED_BATCH*) și *IDLE* (*SCHED_IDLE*);
- ***rt.c*** – în acest fișier sursă sunt implementați algoritmii *FIFO* și *RR*;
- ***core.c*** – reprezintă nucleul planificatorului de procese cu apelurile de sistem asociate.

2.2.1. Algoritmul OTHER (Normal)

Algoritmul *OTHER* sau *Normal*, este implementat în nucleul Linux sub denumirea „*SCHED_OTHER”* și reprezintă planificatorul standard al sistemului de operare Linux bazat pe tehnica partajării timpului (sau *time-sharing*) fiind destinat tuturor proceselor care nu necesită mecanisme speciale de timp real. Acest algoritm se folosește cu o prioritate statică 0, alegerea, în vederea execuției, a unui anumit proces din lista de procese active făcându-se pe baza unei priorități dinamice reprezentată de valoarea *nice*. [17]

SCHED_OTHER folosește divizarea timpului, ceea ce înseamnă că fiecare fir de execuție se execută pentru o perioadă limitată de timp, după care firului de execuție următor îi este permis să ruleze.

2.2.2. Algoritmul FIFO (First In First Out)

Algoritmul *FIFO* (*First În First Out – Primul Intrat Primul Ieșit*) este implementat sub denumirea „*SCHED_FIFO*” și este cel mai simplu algoritm de planificare implementat în nucleul Linux. Ideea de bază a acestui algoritm este aceea că primul proces care solicită acces la procesor va fi primul care își va începe execuția. Procesele active sub acest algoritm de planificare sunt gestionate simplu, printr-o coadă FIFO. Primul proces (din capul cozii) își va începe execuția și va rula atât timp cât îi va fi necesar urmând ca, după finalizarea acestuia, să-și înceapă execuția următorul proces din coada de așteptare (vezi *Figura 2.5*). [15]

Un proces care rulează în planificarea bazată pe algoritmul FIFO, își poate înceta execuția în următoarele situații: [2]

a) Procesul renunță în mod voluntar la procesor trecând pe ultima poziție a cozii;
b) Își finalizează execuția;
c) Este întrerupt de un alt proces care are o prioritate mai mare, în acest caz, procesul trecând pe prima poziție a cozii și reluându-și execuția după finalizarea procesului cu prioritate mai mare.

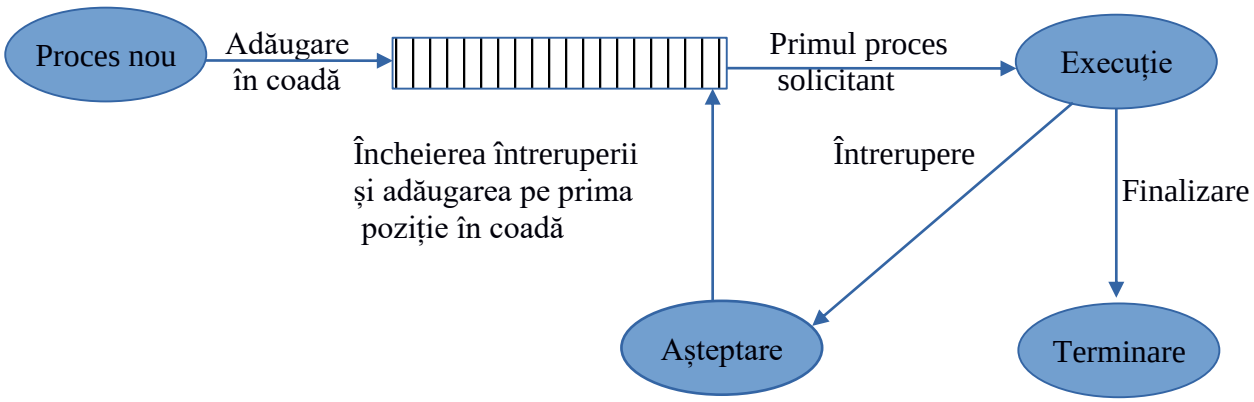


Figura 2.5. Algoritmul FIFO (First In First Out). [9]

2.2.3. Algoritmul RR (Round Robin)

Algoritmul *RR (Round Robin)* este implementat în nucleul Linux sub denumirea „*SCHED_RR”* și funcționează pe baza tehnicii de divizare în timp. Acest algoritm presupune definirea unui interval de timp denumit *cuantă* (de obicei de ordinul câtorva milisecunde, exemplu: 10 milisecunde) și toate procesele active din sistem vor primi un timp de execuție echivalent cuantei definite apriori.

Planificarea *SCHED_RR* este asemănătoare planificării *SCHED_FIFO*, însă se diferențiază de cea din urmă prin faptul că implementează o coadă circulară în care fiecare proces se execută pentru o durată echivalentă cuantei de timp. Dacă un anumit proces nu și-a terminat execuția în momentul expirării cuantei de timp, acesta este întrerupt și va trece pe ultima poziție din coadă, realizându-se comutarea către următorul proces din coadă (vezi *Figura 2.6*). Se va continua pe acest principiu până la finalizarea tuturor proceselor din coadă. [15]

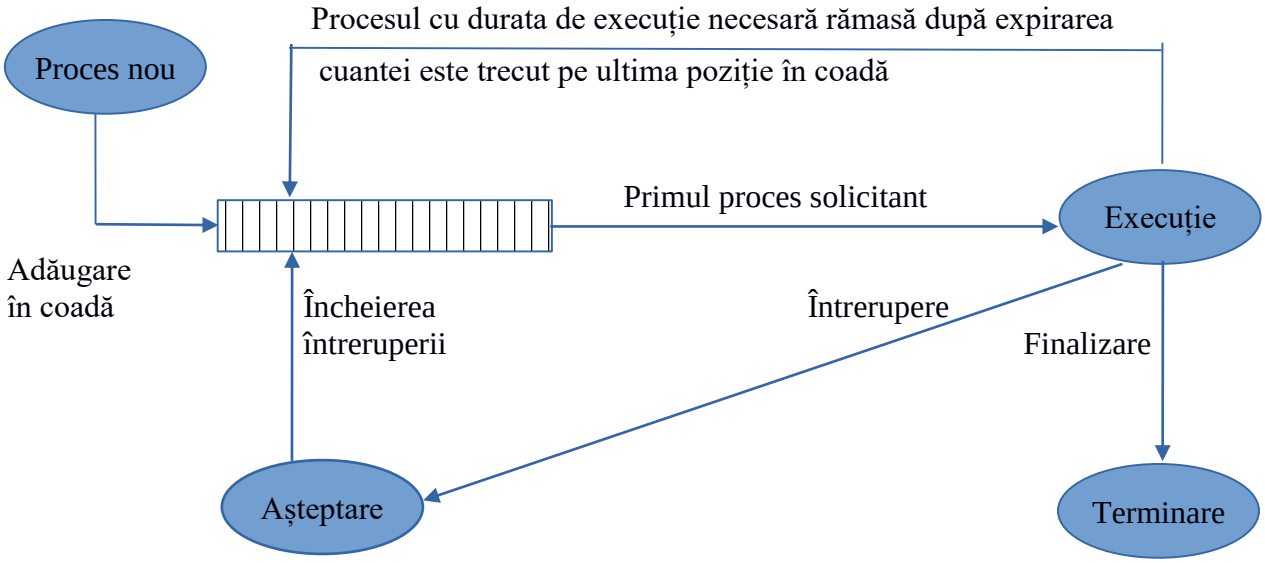


Figura 2.6. Algoritmul RR (Round Robin). [9]

2.2.4. Algoritmul BATCH

Algoritmul *BATCH* este implementat în nucleul Linux începând cu versiunea 2.6.16 sub denumirea „*SCHED_BATCH”*. Acest algoritm este similar cu algoritmul *OTHER* în sensul că se folosește cu o prioritate statică 0, execuția proceselor active făcându-se pe baza unei priorități dinamice reprezentată de valoarea *nice*. Diferența între cei doi algoritmi este aceea că algoritmul *BATCH* presupune mereu că procesele utilizează în mod intensiv procesorul, fiind util în cazul sarcinilor de procesare neinteractive. [18]

2.2.5. Algoritmul IDLE

Algoritmul *IDLE* este implementat în nucleul Linux începând cu versiunea 2.6.23 sub denumirea „*SCHED_IDLE”*. Acest algoritm este de asemenea, similar cu *SCHED_OTHER*, însă oferă o funcționalitate echivalentă cu o valoare *nice* foarte mică (adică mai mică decât +19). Valoarea *nice* a unui proces nu are nici o semnificație pentru acest algoritm, fiind destinat pentru executarea proceselor cu prioritate redusă, care vor primi o proporție semnificativă din timpul procesorului numai în cazul în care nici un alt proces nu solicită acces la procesor. [18]

III. STUDIU EXPERIMENTAL

În scopul realizării prezentului studiu s-au folosit tendințele revoluției digitale, reprezentate de sistemele dedicate, platforme ce oferă performanțe și mobilitate mare la costuri mici.

3.1. Mediul de lucru

Mediul de procesare folosit în prezenta lucrare este Raspberry Pi, un mini-calculator apărut pentru prima dată în anul 2011, fiind produs de către *Raspberry Pi Foundation*.

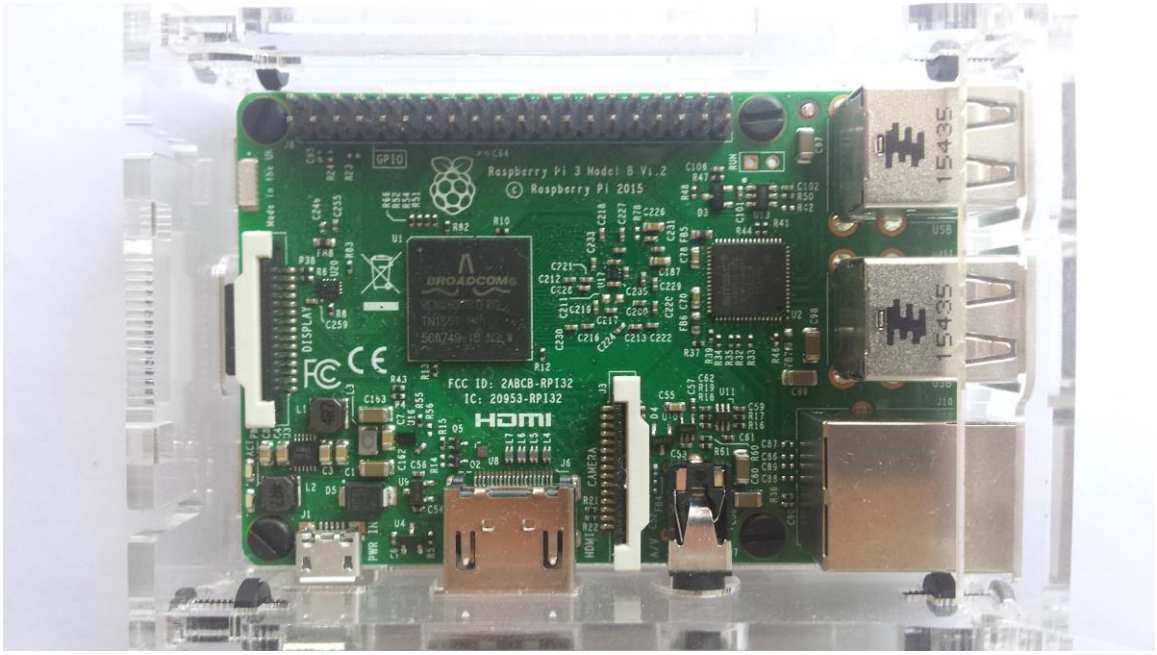


Figura 3.1. Raspberry PI 3 Modelul B

Versiunea Raspberry PI 3 Modelul B utilizată, dispune de următoarele specificații tehnice:

- Procesor Quad-core Broadcom BCM2837 ARMv7 pe 64 de biți cu o frecvență la 1,2 GHz;
- Memorie RAM 1 GB;
- Modul WIFI BCM43143;
- Modul Bluetooth;
- 40 de pini GPIO (*General Purpose Input/Output*);
- 4 porturi USB;
- 1 port HDMI;
- 1 port Audio 3,5 mm;
- 1 port Ethernet;
- 1 port Micro SD pentru încărcarea sistemului de operare.

În ceea ce privește sistemul de operare, s-a folosit distribuția oficială pentru Raspberry PI denumită *Raspbian*, mai exact, versiunea 8 (*jessie*) ce are la bază versiunea de nucleu *4.1.19-v7+*.

3.2. Instrumentul software

În scopul comparării performanțelor algoritmilor de planificare, a fost implementat un program prin intermediul căruia se poate genera un număr variabil de fire de execuție în interiorul unui singur proces creat prin rularea programului.

Programul a fost dezvoltat în limbajul de programare C, utilizând următoarele biblioteci de funcții:

```
#include <stdio.h> /* Pentru Intrare / Ieșire */
#include <pthread.h> /* Pentru fire de execuție definite în standardul POSIX */
#include <sched.h> /* Pentru algoritmii de planificare */
#include <unistd.h> /* Constante simbolice. Exemplu: getpid */
#include <string.h> /* Rutine de tratare a șirurilor */
#include <sys/time.h> /* Tipuri timp. Exemplu: time_t */
#include <time.h> /* Tipuri timp. Exemplu: tm */
```

Firele de execuție se pot genera cu o politică de planificare bazată pe unul din algoritmii implementați în nucleul Linux, care au fost descriși în capitolul anterior, și anume: *OTHER, FIFO (First În First Out), RR (Round Robin), BATCH, IDLE*.

Algoritmii de planificare implementați într-un sistem Linux pot fi identificați prin executarea în linia de comandă a comenzii (vezi *Figura 3.2*):

```
$ chrt -m
```

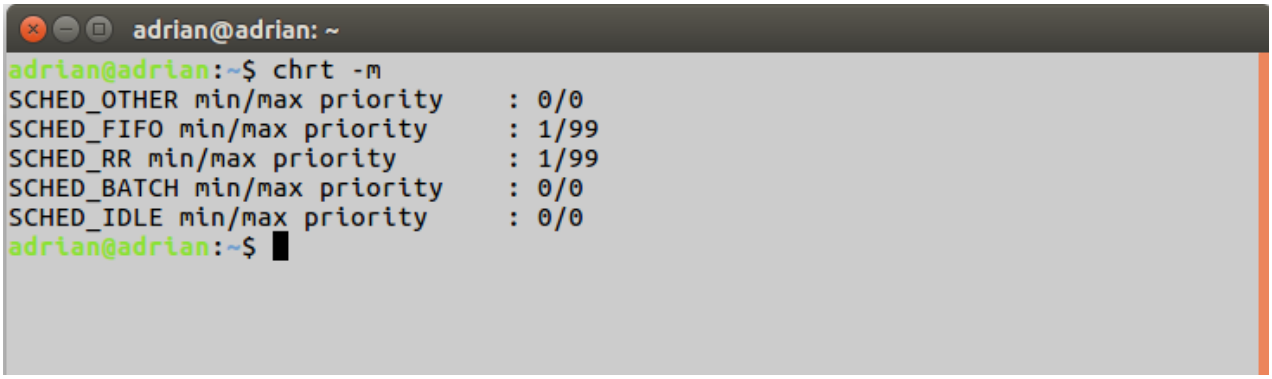


Figura 3.2. Algoritmii de planificare

De asemenea, firele de execuție, pe lângă politica de planificare, pot primi ca parametru și priorități folosind următoarea funcție definită în program:

```
int Prioritate (int pr) {
      unsigned int seed;
       prio[pr] = 100 - (rand_r(&seed) % (100 - pr) + pr);
       return prio[pr];
}
```

Această funcție generează priorități pseudo-aleatoare în intervalul [0, 100].

Pentru sincronizarea firelor de execuție s-a folosit bariera de sincronizare definită astfel:

```
static pthread_barrier_t bariera;
```

Pentru a asigura exclusivitatea accesului la resursele partajate s-a folosit mecanismul de excludere mutuală (*mutex*) definit astfel:

```
pthread_mutex_t lock;
```

Firele de execuție apelează aceeași funcție:

```
static void *operatii(void *arg){
.....
}
```

care primește ca parametru id-ul firului de execuție și calculează înmulțirea a două matrici (de dimensiuni 300 x 300 în acest studiu) și introduce întârzieri aleatoare firelor de execuție:

```
int nsecs = random() % 20 + 1; /* Așteptare între 1 și 20 de secunde */
sleep(nsecs);
```

Generarea firelor de execuție se face prin intermediul funcției:

```
pthread_create(&tid[idThread], NULL, operatii, (void *) idThread);
```

Finalizarea firelor de execuție este realizată cu funcția:

```
pthread_join(tid[idThread], NULL);
```

Pentru ca firele de execuție să ruleze sub un anumit algoritm de planificare, este necesară setarea acestuia prin intermediul apelării funcției:

```
sched_setscheduler(0, policy, &param);
```

unde:

- `policy`: reprezintă algoritmul de planificare;
- `param`: reprezintă prioritatea ce va fi atribuită firului de execuție.

Compilarea programului este realizată cu următoarea comandă:

```
$ gcc program.c -o program -lpthread
```

Execuția propriu-zisă a programului se realizează prin comanda:

```
$ sudo ./program nr_bariere algoritm nr_thread-uri
```

unde:

- `sudo`: pentru a schimba algoritmul de planificare al sistemului este necesară acordarea unor drepturi de administrator;
- `nr_bariere`: reprezintă numărul de bariere (niveluri) de sincronizare;
- `algoritm`: reprezintă algoritmul de planificare utilizat și poate fi selectat introducând de la tastatură un număr astfel:
  - 0, pentru *OTHER*;
  - 1, pentru *FIFO*;
  - 2, pentru *RR*;

- 3, pentru *BATCH*;
- 5, pentru *IDLE;*

- `nr_thread-uri`: numărul de fire de execuție ce se doresc a fi generate.

Exemplu de execuție a programului cu o barieră de sincronizare, algoritmul FIFO și 10 fire de execuție (vezi *Figura 3.3*):

```
$ sudo ./program 1 1 10
```

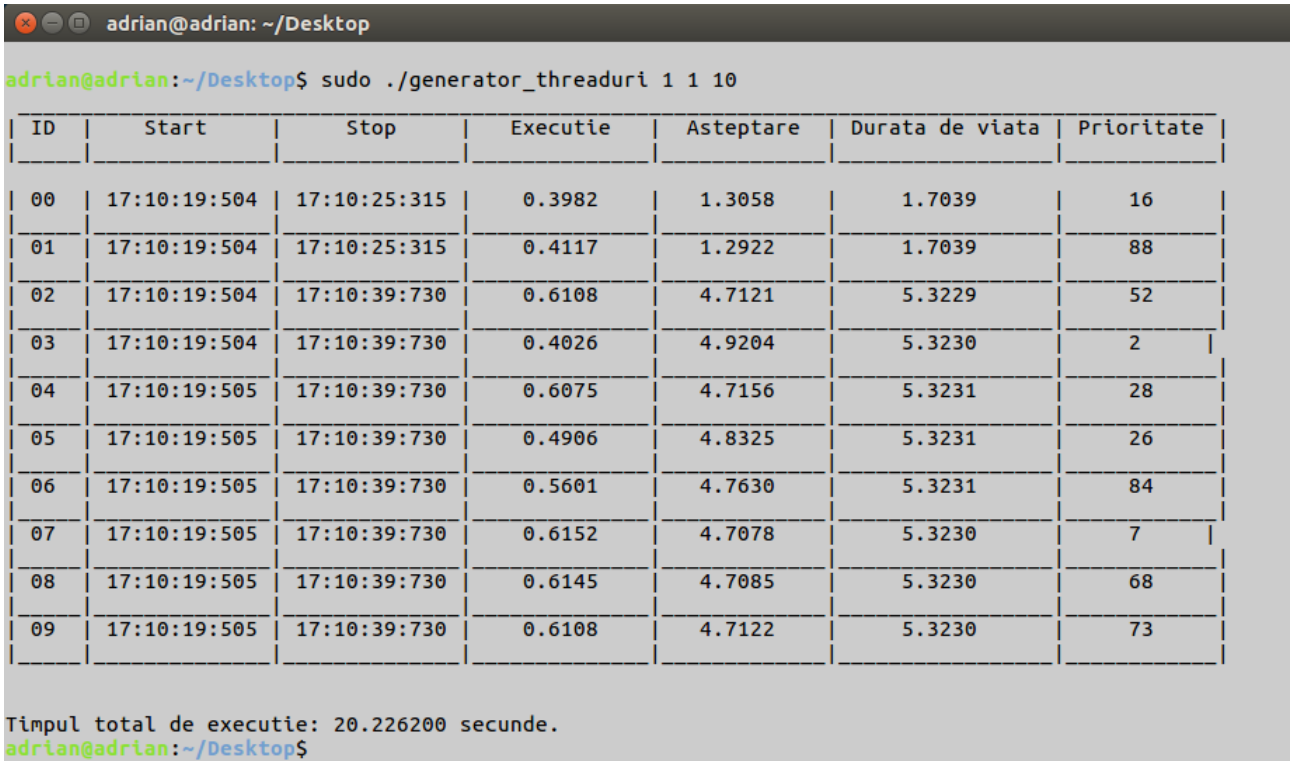


Figura 3.3. Execuția programului

Rezultatele obținute în urma rulării programului sunt salvate automat într-un fișier text sub forma celor ilustrate în *Figura 3.3*.

3.2. Rezultate

În vederea realizării comparației algoritmilor de planificare implementați în nucleul Linux, au fost realizate mai multe experimente grupate în următoarele cazuri:

A) În primul caz s-a considerat generarea a 2 fire de execuție pentru fiecare algoritm în parte cu priorități aleatoare. Rezultatele obținute sunt ilustrate în figurile următoare:

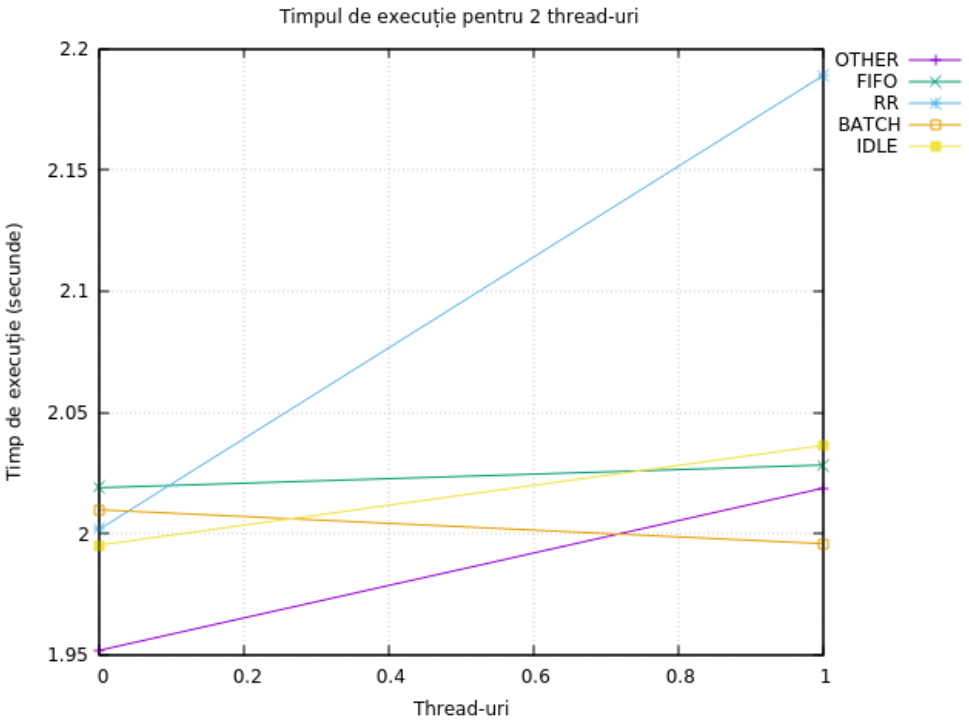


Figura 3.4. Timpul de execuție pentru 2 thread-uri cu priorități aleatoare

În graficul ilustrat de *Figura 3.4* este reprezentat timpul de execuție al celor 2 fire de execuție generate. Se poate observa, o latență mai mare în cazul algoritmului RR, cele mai mici valori ale timpului de execuție obținându-se în cazul algoritmului OTHER. În cazul algoritmului FIFO, timpul de execuție este relativ constant.

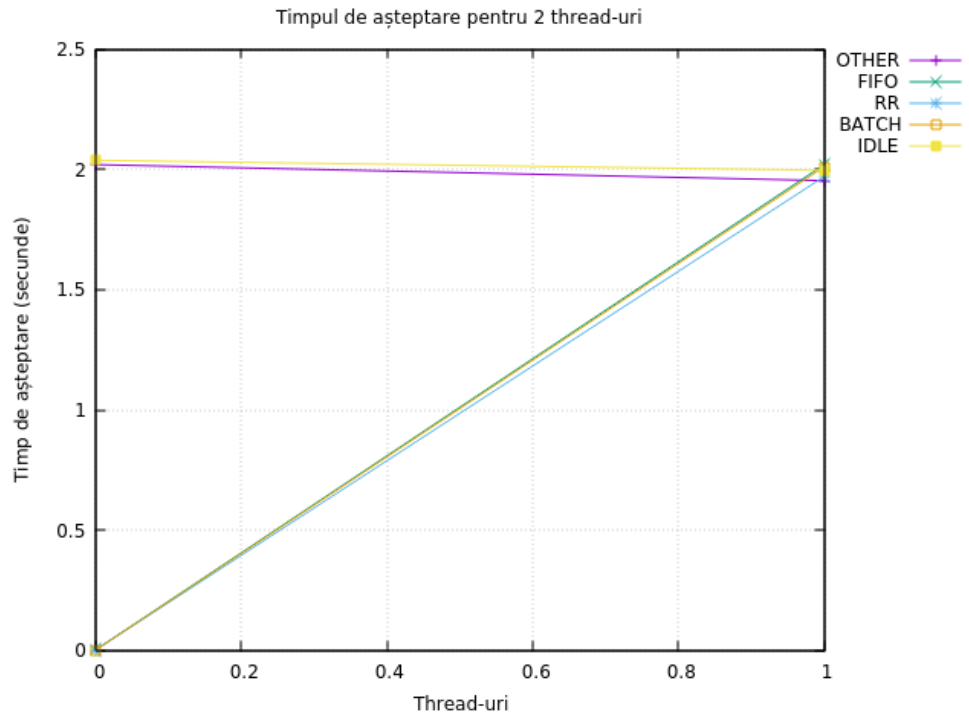


Figura 3.5. Timpul de așteptare pentru 2 thread-uri cu priorități aleatoare

În *Figura 3.5* este reprezentat timpul de așteptare al celor 2 fire de execuție. Se poate observa o echivalență a timpului de așteptare în cazul algoritmilor IDLE și OTHER, aceștia având și cele mai

mari valori spre deosebire de algoritmii FIFO, RR și BATCH care au de asemenea valori asemănătoare ale timpului.

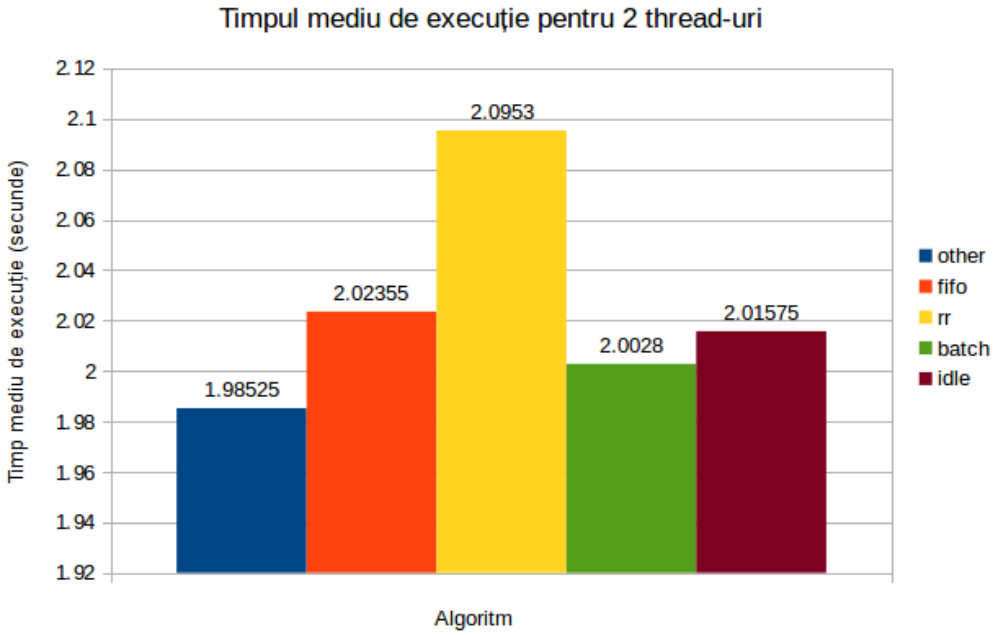


Figura 3.6. Timpul mediu de execuție pentru 2 thread-uri cu priorități aleatoare

Se observă în *Figura 3.6* o reprezentare a timpului mediu de procesare al celor 2 fire de execuție. În acest caz, s-a obținut o eficiență mai mare (un timp mediu de procesare mai mic) pentru algoritmul OTHER, cea mai mică eficiență rezultând pentru algoritmul RR.

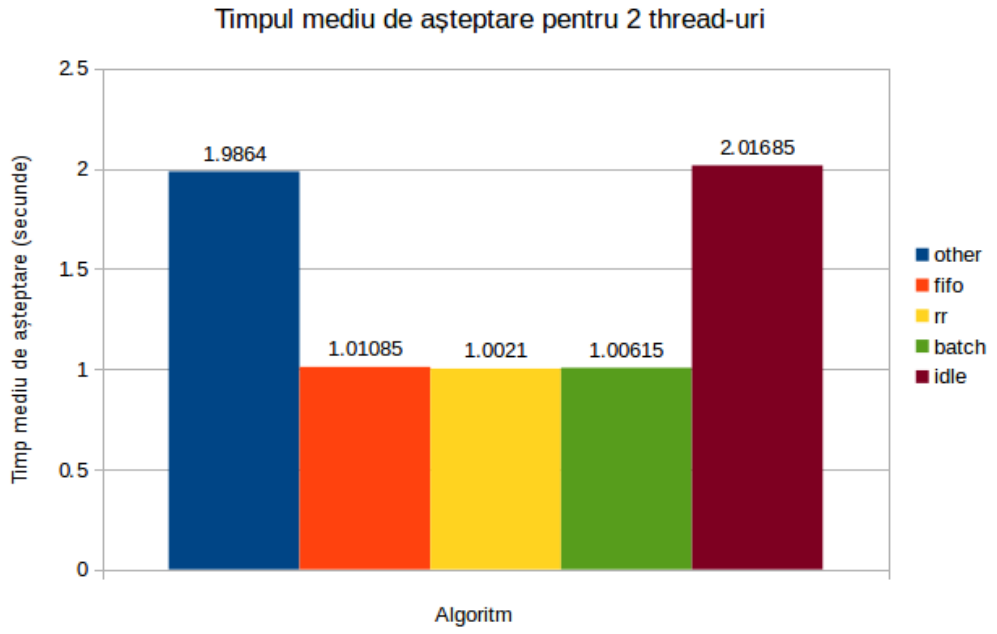


Figura 3.7. Timpul mediu de așteptare pentru 2 thread-uri cu priorități aleatoare

În *Figura 3.7* se poate observa un timp mediu de așteptare pentru cele 2 fire de execuție aproximativ dublu (2 secunde) în cazul algoritmilor OTHER și IDLE față de algoritmii FIFO, RR și BATCH care au de asemenea, valori asemănătoare ale timpului mediu de așteptare (o secundă).

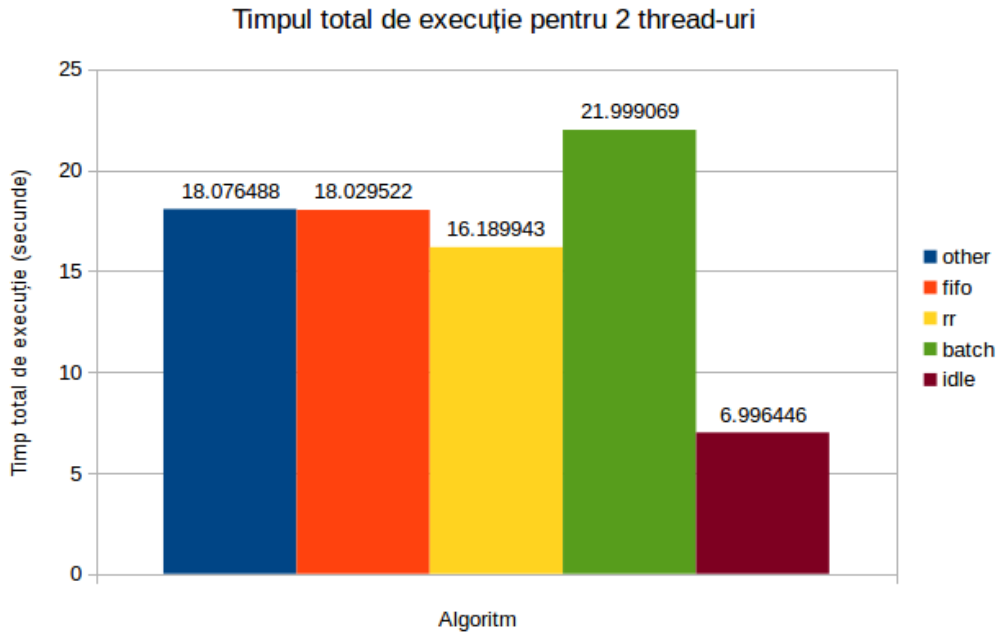


Figura 3.8. Timpul total de execuție pentru 2 thread-uri cu priorități aleatoare

În *Figura 3.8* este reprezentat timpul total de execuție calculat din momentul creării celor două fire de execuție și până în momentul finalizării acestora. Din acest punct de vedere, se poate considera că cel mai eficient algoritm este algoritmul IDLE, pe ultimul loc situându-se algoritmul BATCH.

B) *În al doilea caz, s-a considerat generarea a 4 fire de execuție pentru fiecare algoritm în parte cu priorități aleatoare. Rezultatele obținute sunt reprezentate în figurile următoare:*

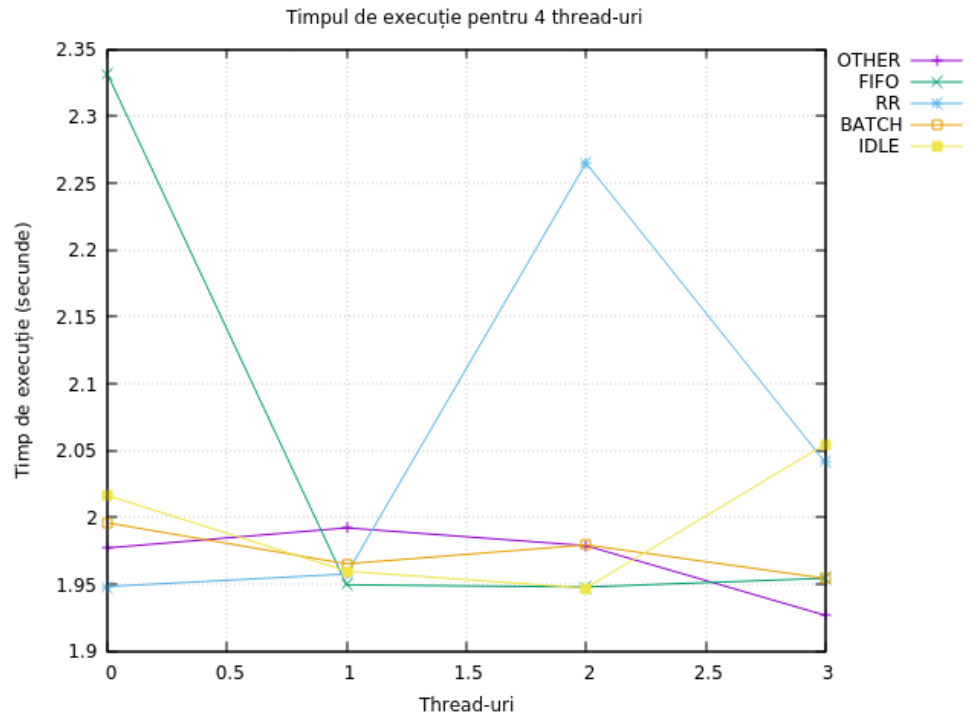


Figura 3.9. Timpul de execuție pentru 4 thread-uri cu priorități aleatoare

Graficul reprezentat în *Figura 3.9* sugerează un timp de execuție variabil și vizibil mai mare pentru cele 4 fire de execuție generate în cazul algoritmului RR. Valori mai mici și relativ apropiate ale timpului de execuție s-au obținut pentru algoritmii OTHER și BATCH.

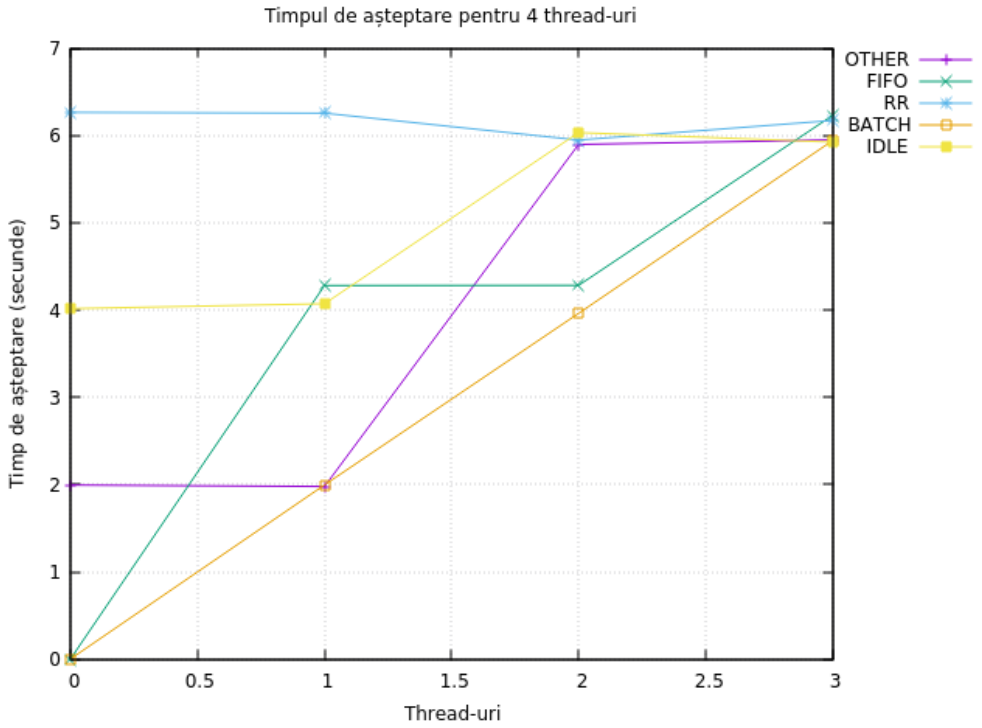


Figura 3.10. Timpul de așteptare pentru 4 thread-uri cu priorități aleatoare

Se poate observa în figura de mai sus, un timp de așteptare relativ constant și mai mare în cazul algoritmului RR. De asemenea, timpul de așteptare are o tendință crescătoare pentru algoritmii OTHER, FIFO, IDLE și BATCH.

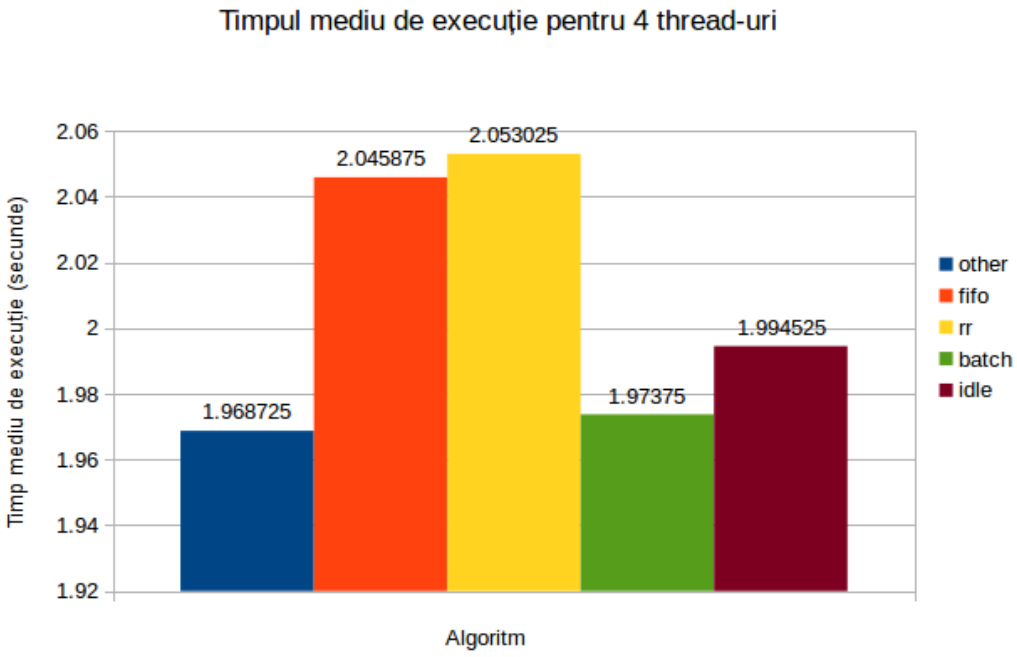


Figura 3.11. Timpul mediu de execuție pentru 4 thread-uri cu priorități aleatoare

Din *Figura 3.11* rezultă un timp mediu de execuție mai mare în cazul algoritmilor FIFO și RR și mai mic pentru algoritmii OTHER și BATCH.

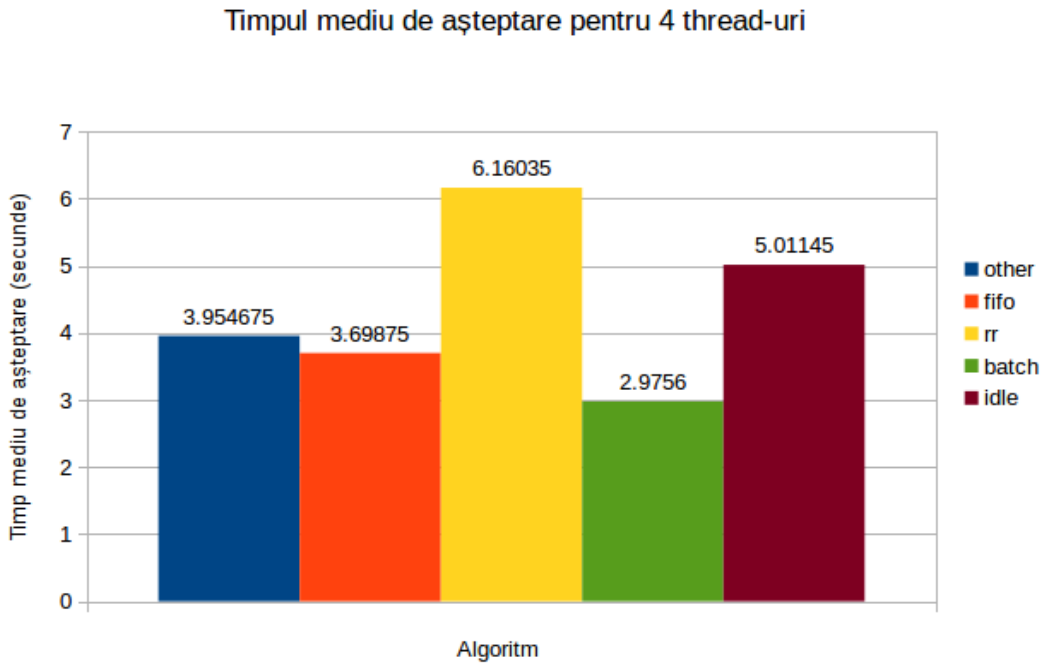


Figura 3.12. Timpul mediu de așteptare pentru 4 thread-uri cu priorități aleatoare

În acest caz, timpul mediu de așteptare este defavorabil în cazul algoritmului RR, o valoare mică obținându-se pentru algoritmul BATCH.

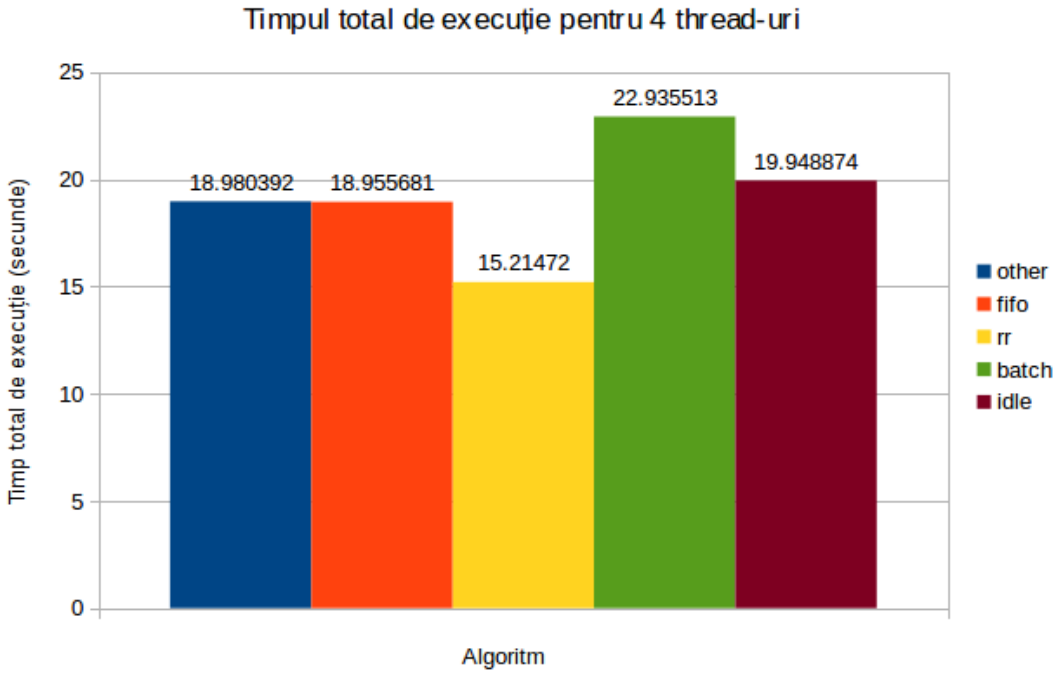


Figura 3.13. Timpul total de execuție pentru 4 thread-uri cu priorități aleatoare

Conform graficului din *Figura 3.13*, timpul total de execuție este, asemănător cazului anterior A), mai mare în cazul algoritmului BATCH.

C) *În al treilea caz, s-a considerat generarea a 8 fire de execuție pentru fiecare algoritm în parte cu priorități aleatoare. Rezultatele obținute sunt reprezentate în figurile următoare:*

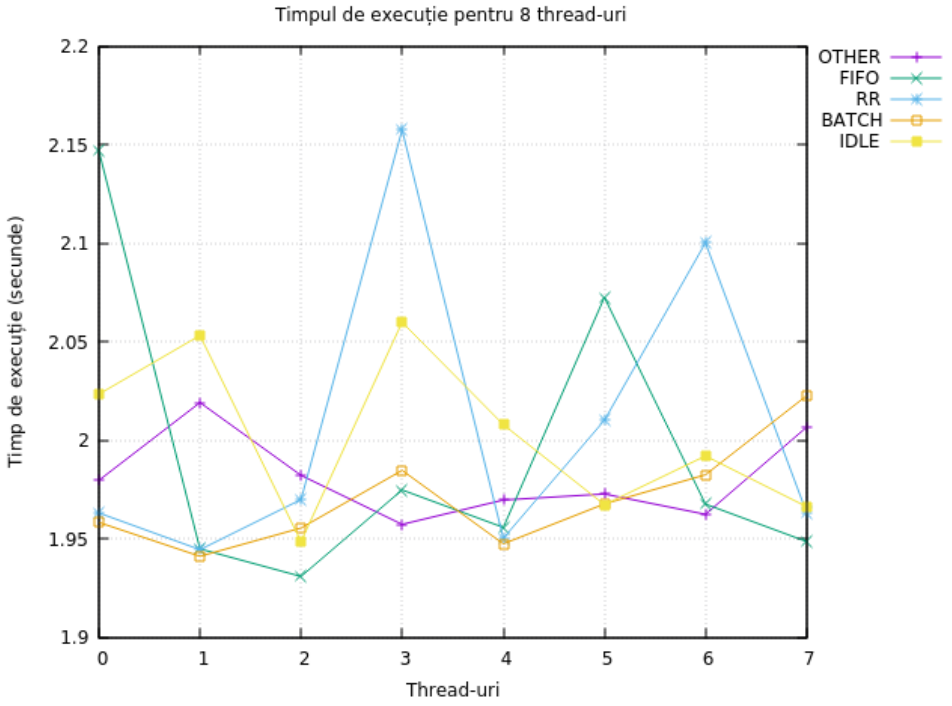


Figura 3.14. Timpul de execuție pentru 8 thread-uri cu priorități aleatoare

În figura de mai sus, se poate observa un grad de variație mare a timpului de execuție pentru algoritmii FIFO, RR și IDLE, o eficiență mai mare obținându-se pentru algoritmii OTHER și BATCH care prezintă un grad redus de variație.

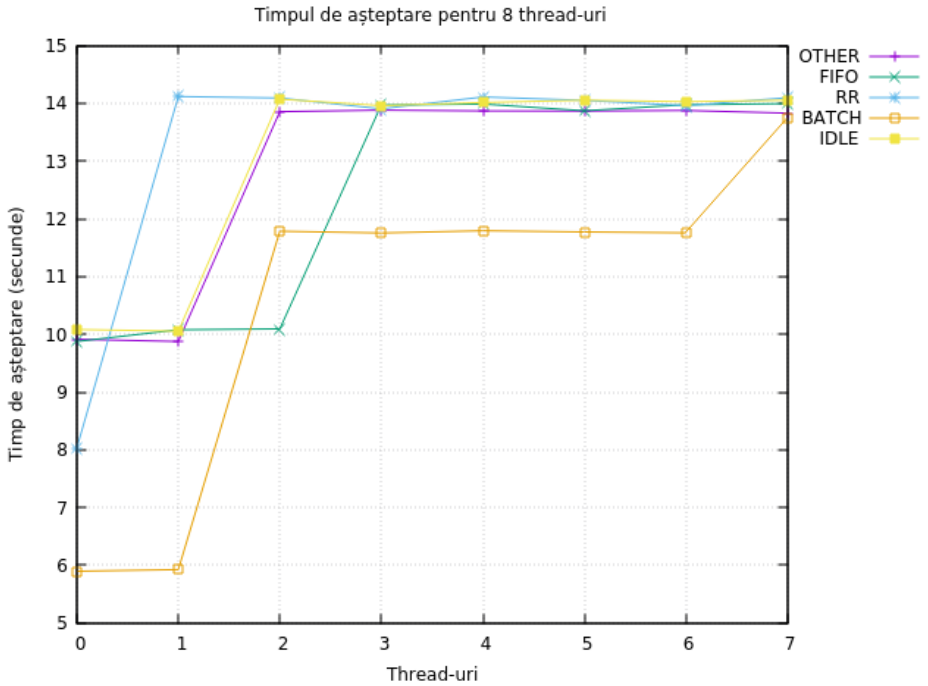


Figura 3.15. Timpul de așteptare pentru 8 thread-uri cu priorități aleatoare

Timpul de așteptare reprezentat în *Figura 3.15*, are o tendință de creștere pentru toți algoritmii, valori optime (mici), obținându-se pentru algoritmul BATCH.

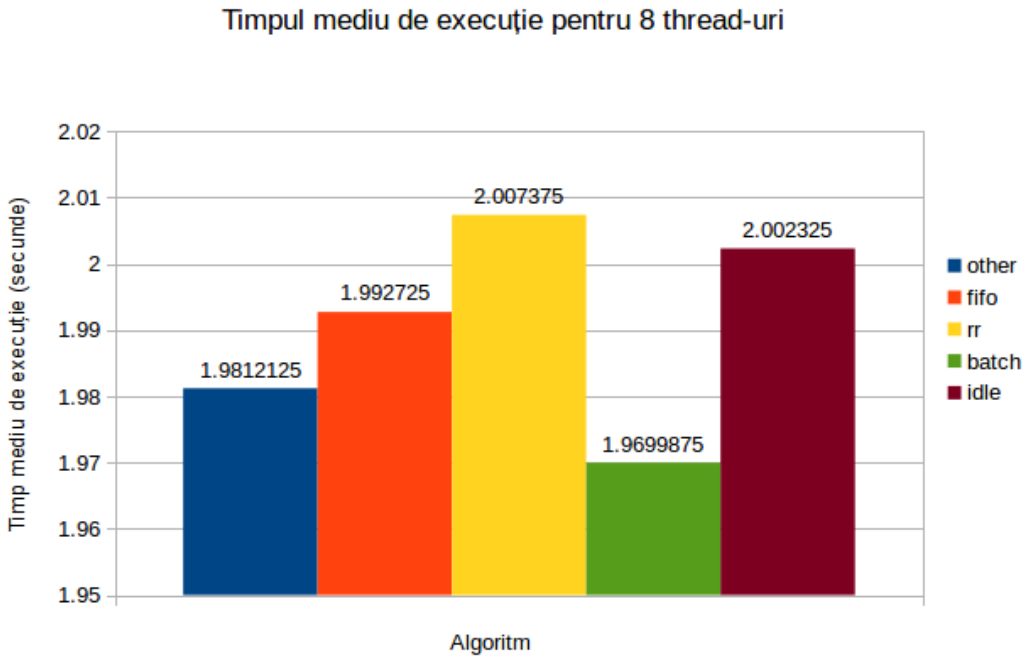


Figura 3.16. Timpul mediu de execuție pentru 8 thread-uri cu priorități aleatoare

Timpul mediu de procesare pentru cele 8 fire de execuție generate, atinge valoarea optimă în cazul algoritmului BATCH (1,969 secunde), cea mai mare valoare obținându-se pentru algoritmul RR.

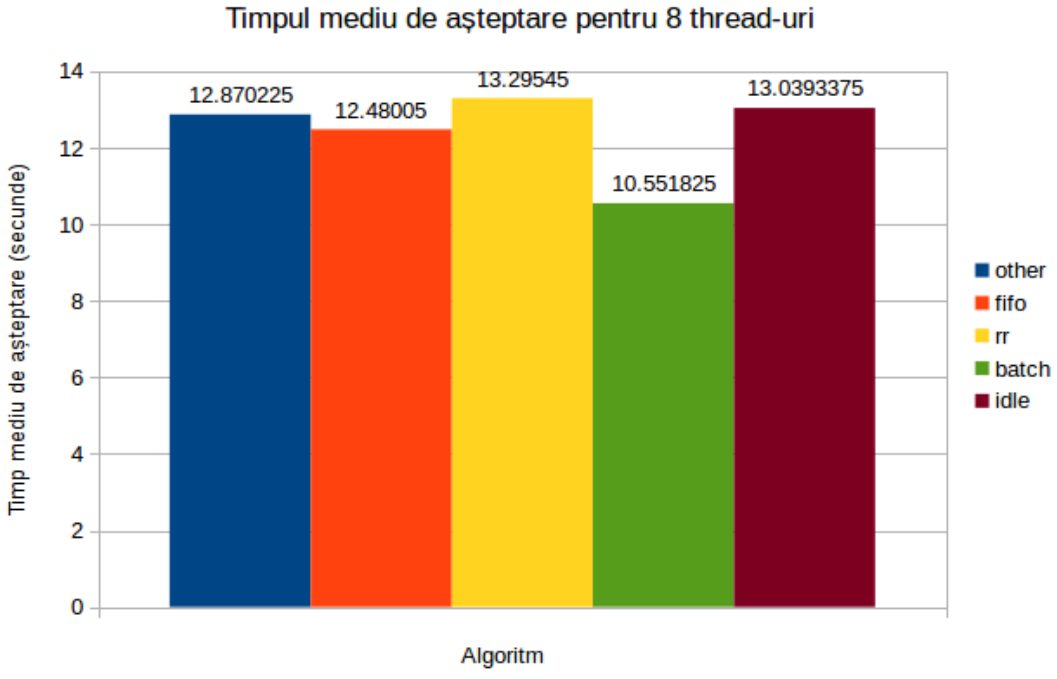


Figura 3.17. Timpul mediu de așteptare pentru 8 thread-uri cu priorități aleatoare

Și în ceea ce privește timpul mediu de așteptare, algoritmul BATCH returnează valoarea cea mai optimă spre deosebire de algoritmul RR care are o durată de așteptare mai mare.

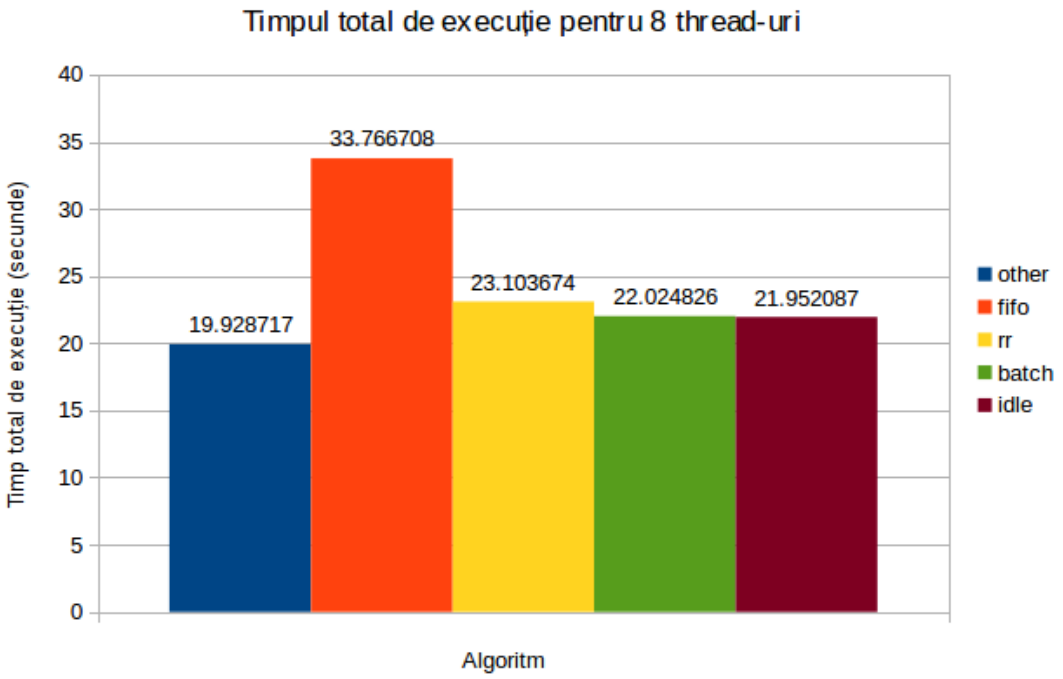


Figura 3.18. Timpul total de execuție pentru 8 thread-uri cu priorități aleatoare

În ceea ce privește timpul total de execuție în acest caz, se constată că cel mai ineficient algoritm este FIFO, de cealaltă parte primul loc ocupându-l algoritmul OTHER.

D) În al patrulea caz, s-a considerat generarea a 16 fire de execuție pentru fiecare algoritm în parte cu priorități aleatoare. Rezultatele obținute sunt ilustrate în figurile următoare:

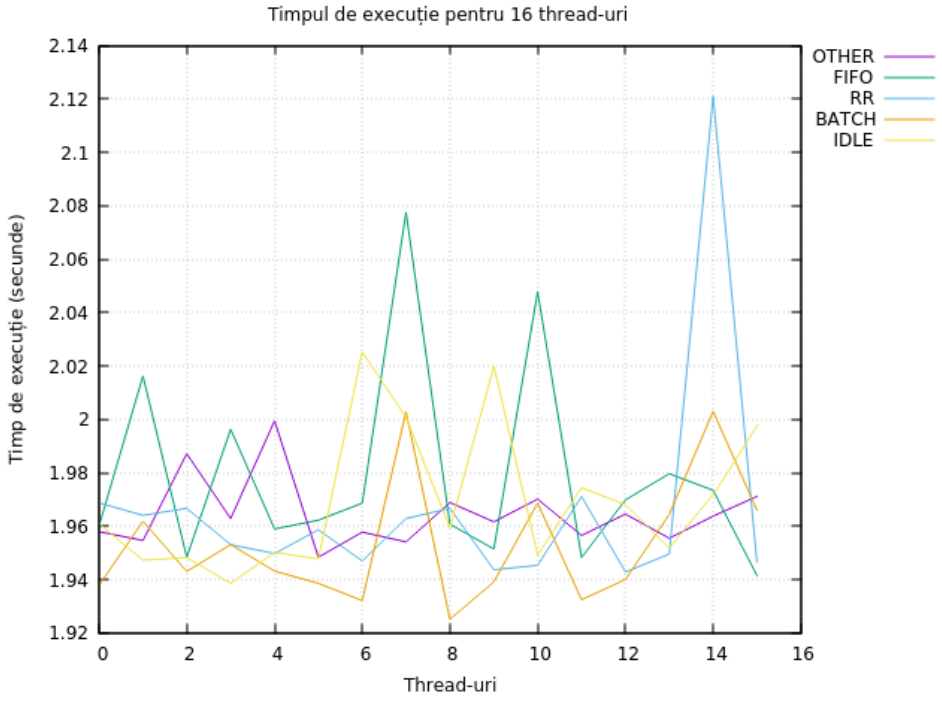


Figura 3.19. Timpul de execuție pentru 16 thread-uri cu priorități aleatoare

În *Figura 3.19*, se poate observa că algoritmul OTHER prezintă timpi de execuție relativ constanți și cu valori mai mici comparativ cu restul algoritmilor.

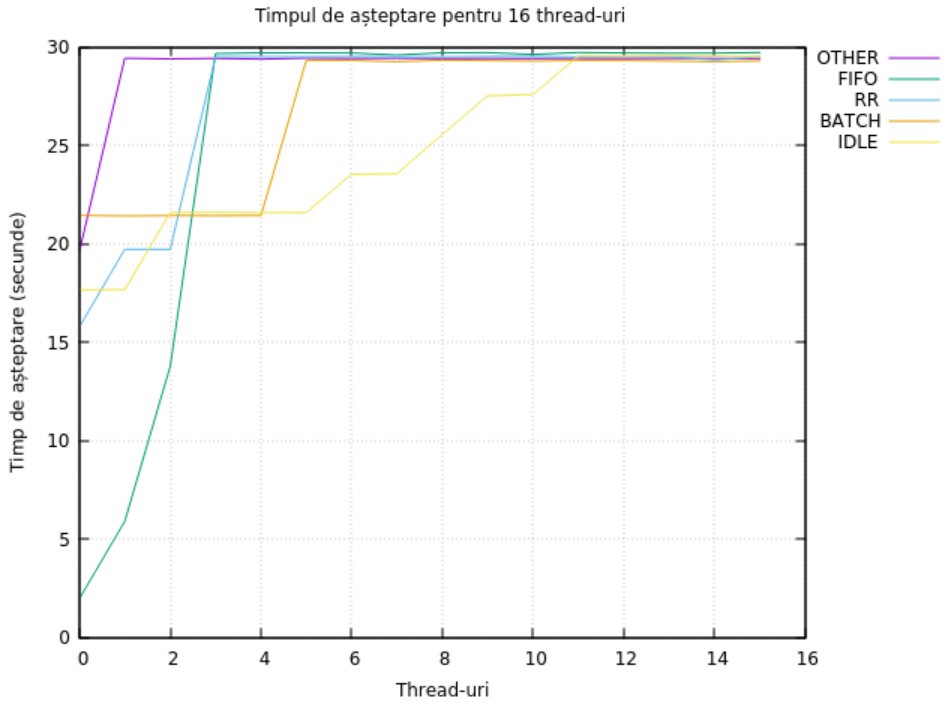


Figura 3.20. Timpul de așteptare pentru 16 thread-uri cu priorități aleatoare

În graficul de mai sus, se observă o limitare a timpului de așteptare în jurul valorii de 29 de secunde pentru toți algoritmii cu excepția algoritmului IDLE care prezintă cele mai mici valori ale timpului de așteptare.

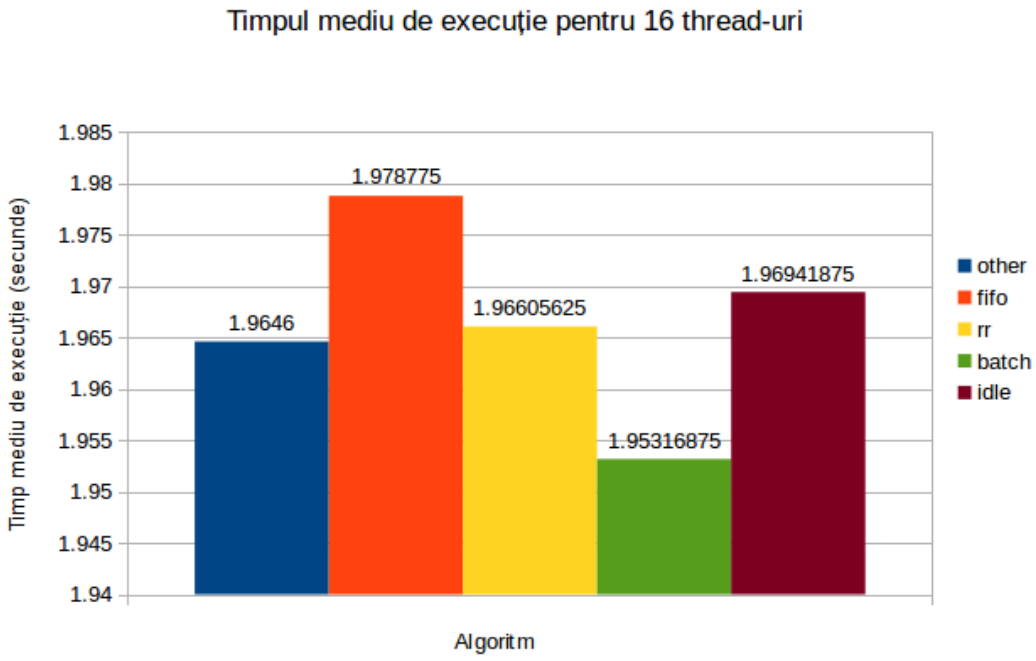


Figura 3.21. Timpul mediu de execuție pentru 16 thread-uri cu priorități aleatoare

În acest caz, asemănător celui anterior (pentru 8 fire de execuție), timpul mediu de procesare al firelor de execuție este optim în cazul algoritmului BATCH.

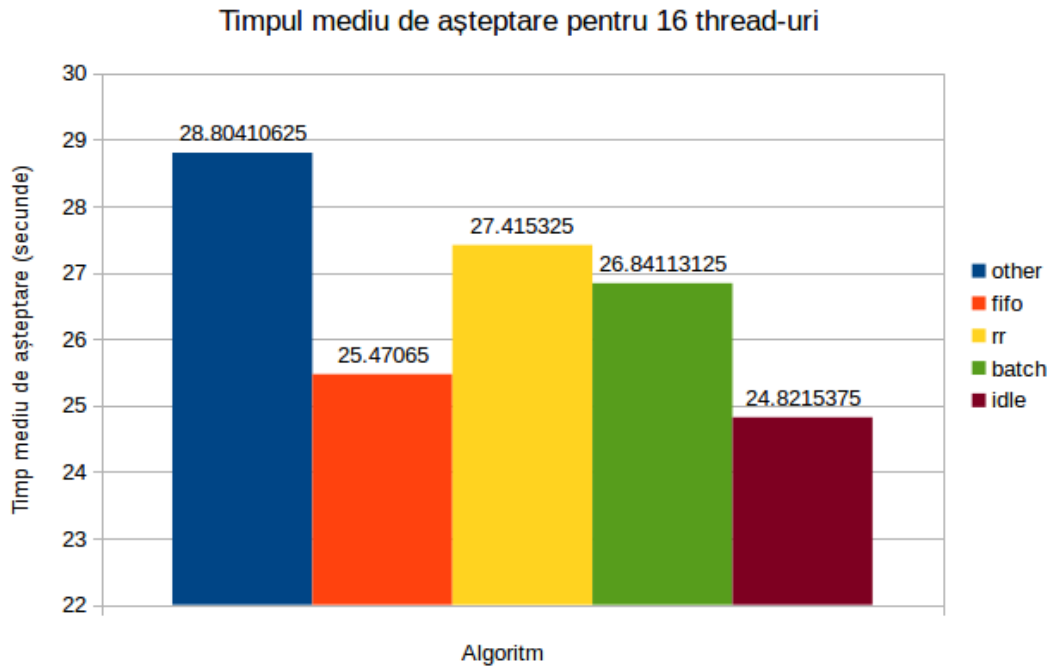


Figura 3.22. Timpul mediu de așteptare pentru 16 thread-uri cu priorități aleatoare

Timpul mediu de așteptare reprezentat în figura de mai sus, are o valoare optimă pentru algoritmul IDLE. De cealaltă parte, cea mai mare valoare se obține în cazul algoritmului OTHER.

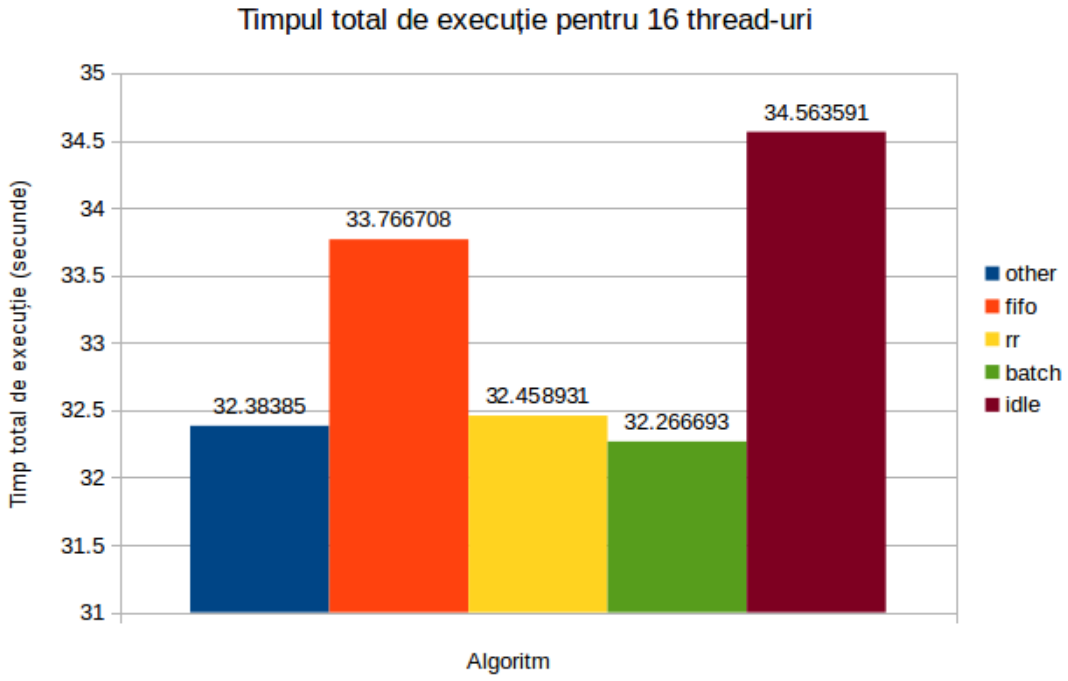


Figura 3.23. Timpul total de execuție pentru 16 thread-uri cu priorități aleatoare

Conform *Figurii 3.23*, se poate spune că cel mai ineficient algoritm este algoritmul IDLE, urmat de algoritmul FIFO.

E) În al cincilea caz, s-a considerat generarea a 32 de fire de execuție pentru fiecare algoritm în parte cu priorități aleatoare. Rezultatele obținute sunt ilustrate în figurile următoare:

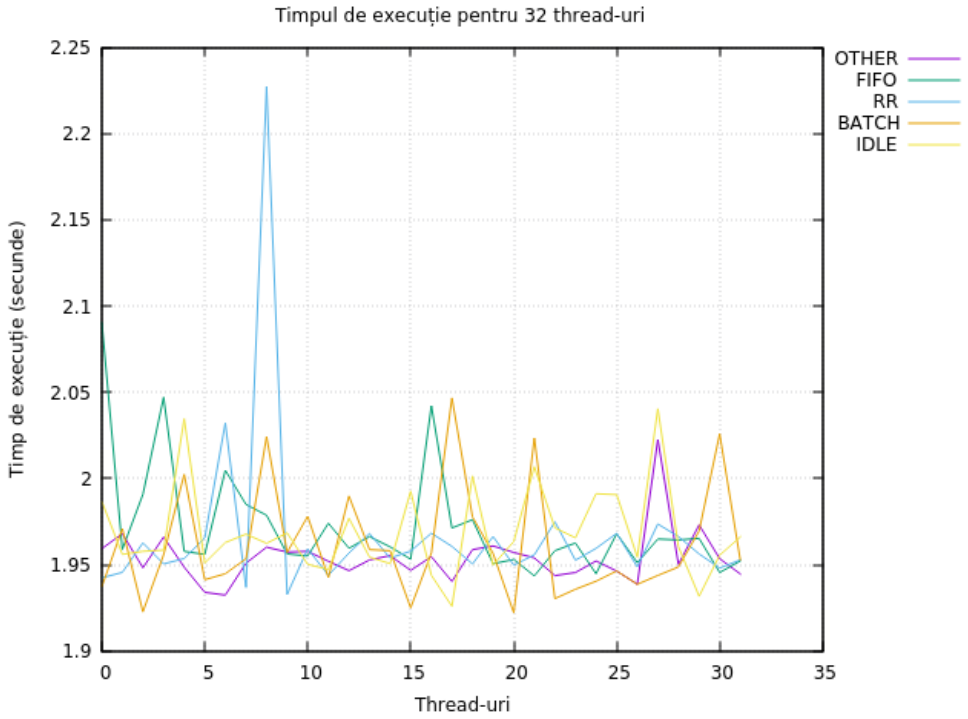


Figura 3.24. Timpul de execuție pentru 32 de thread-uri cu priorități aleatoare

În graficul timpului de execuție al algoritmului RR din figura de mai sus, se poate observa un spike care poate fi datorat activității sistemului de operare. Și în acest caz, un comportament optim se obține pentru algoritmul OTHER.

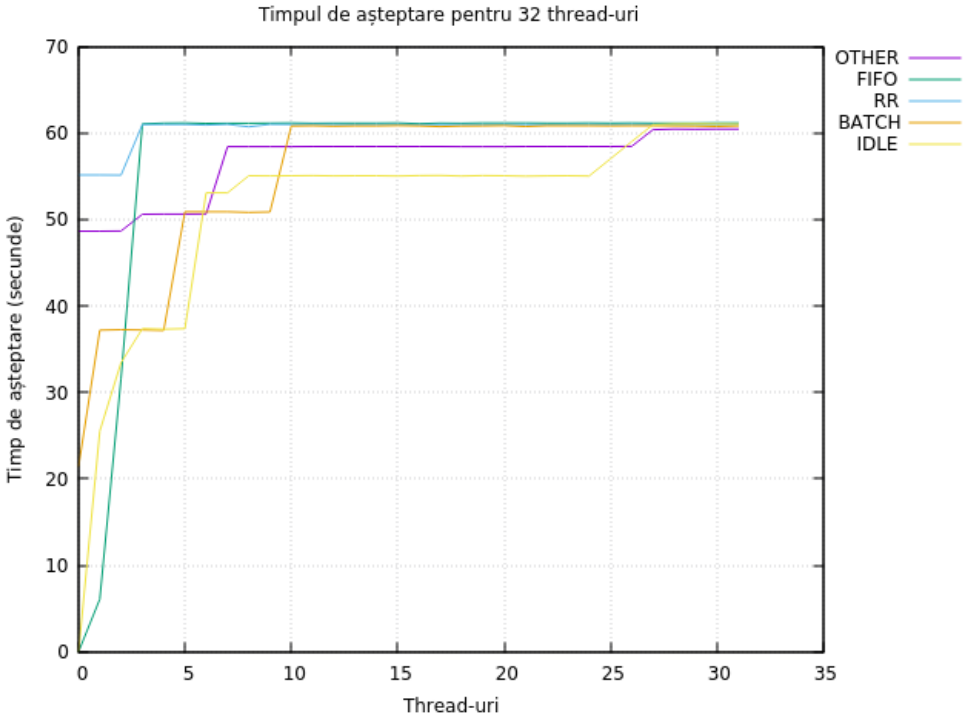


Figura 3.25. Timpul de așteptare pentru 32 de thread-uri cu priorități aleatoare

Și în acest caz, reprezentat de Figura 3.25, timpul de așteptare are valori mai mici pentru algoritmul IDLE, ceilalți algoritmi având valori asemănătoare, în jurul pragului de 60 de secunde.

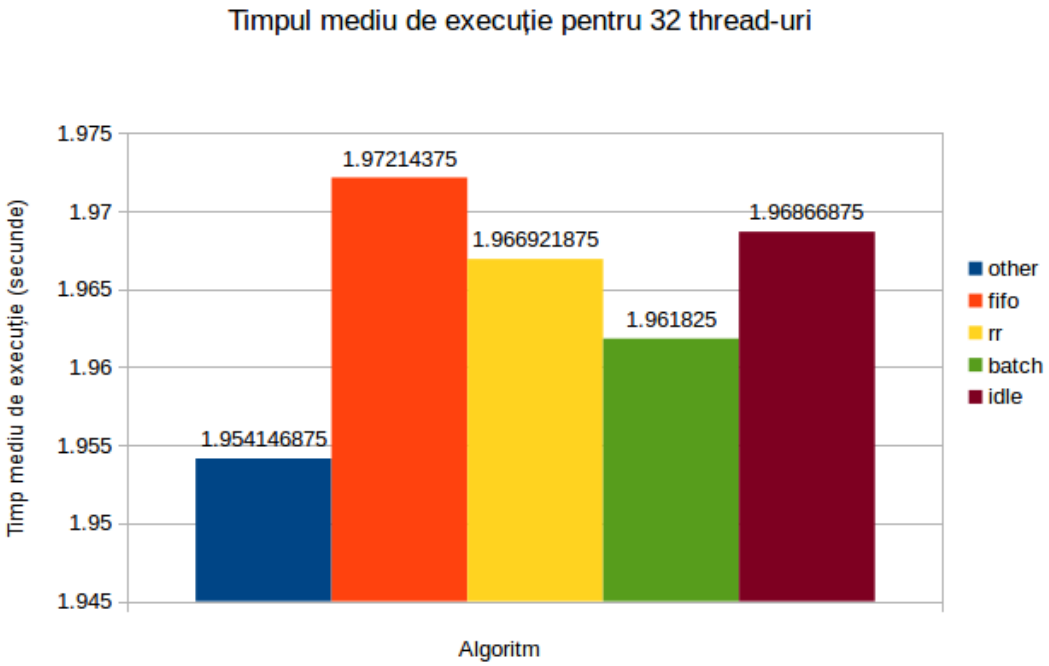


Figura 3.26. Timpul mediu de execuție pentru 32 de thread-uri cu priorități aleatoare

În acest caz, conform figurii de mai sus, timpul mediu de execuție are o valoare optimă (1,95 secunde) pentru algoritmul OTHER.

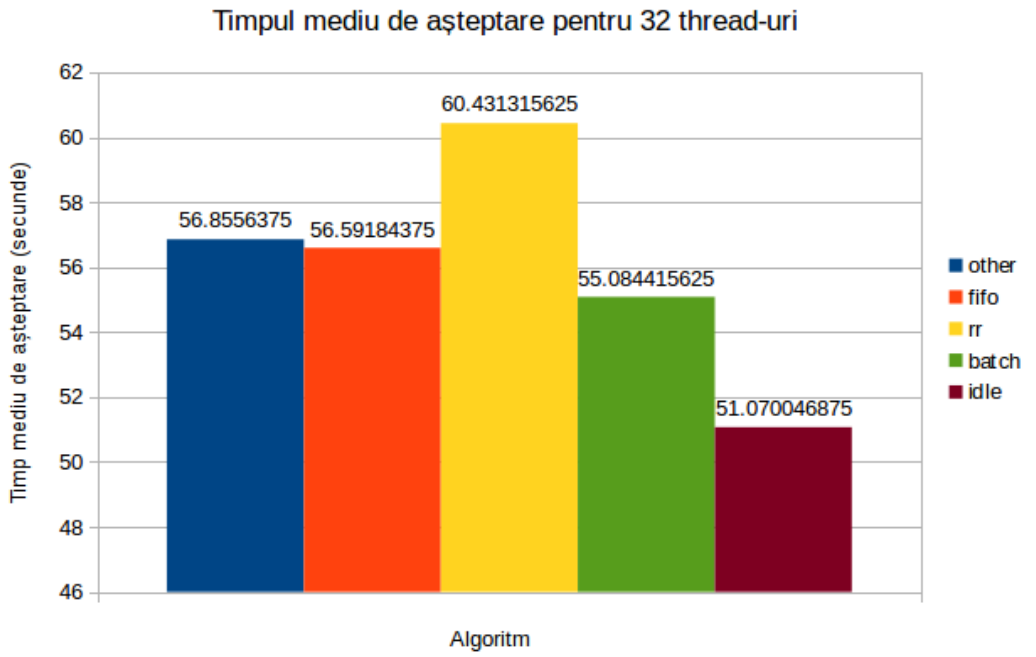


Figura 3.27. Timpul mediu de așteptare pentru 32 de thread-uri cu priorități aleatoare

În ceea ce privește timpul de așteptare, algoritmul IDLE se dovedește a fi cel mai eficient la fel ca în cazul anterior.

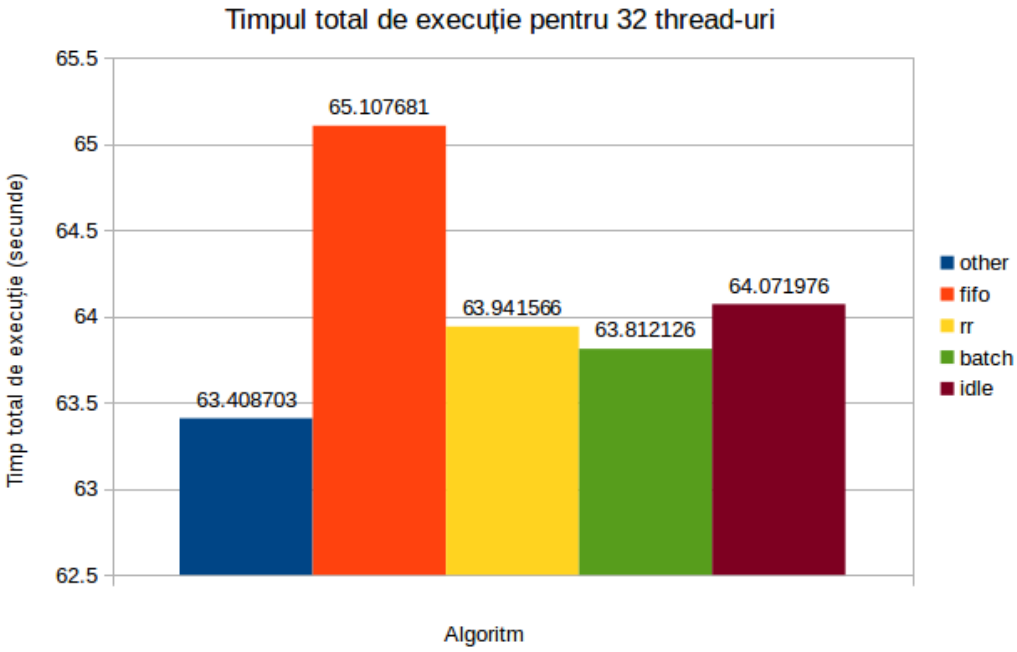


Figura 3.28. Timpul total de execuție pentru 32 de thread-uri cu priorități aleatoare

Timpul total de execuție ilustrat în graficul de mai sus are cea mai mică valoare pentru algoritmul OTHER și cea mai mare pentru algoritmul FIFO.

F) *În cel de-al șaselea caz, s-au generat 64 de fire de execuție pentru fiecare algoritm în parte cu priorități aleatoare. Rezultatele obținute sunt ilustrate în figurile următoare:*

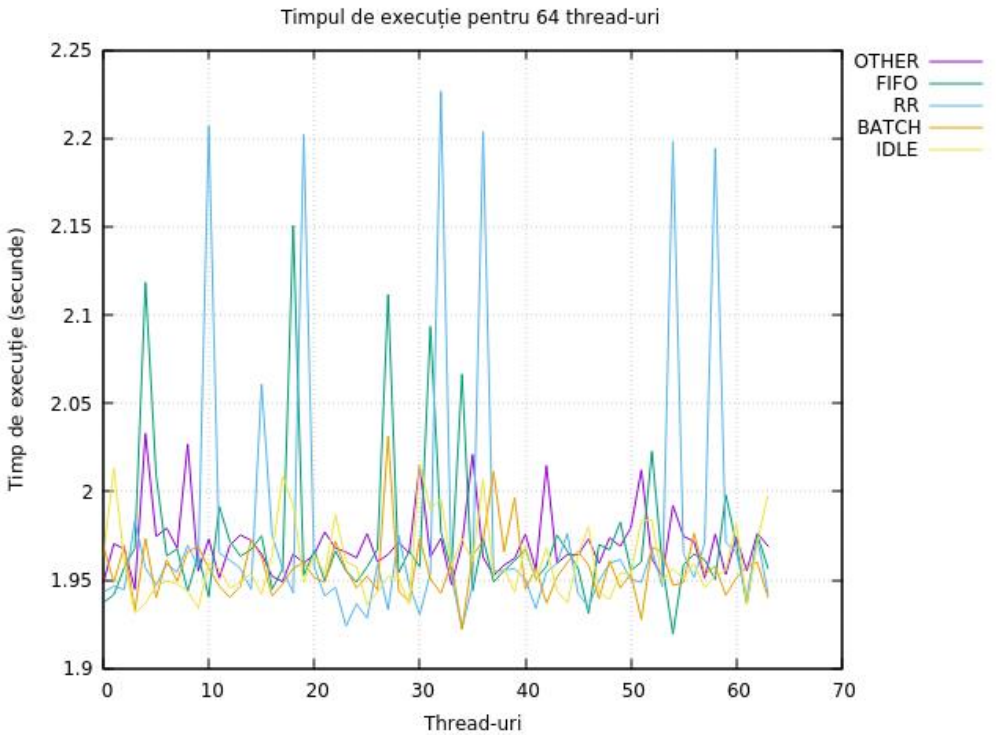


Figura 3.29. Timpul de execuție pentru 64 de thread-uri cu priorități aleatoare

În *Figura 3.29* se poate constata o variație mare a timpilor de execuție în cazul algoritmului RR, fapt datorat întreruperilor frecvente ale firelor de execuție pe motivul expirării cuantei de timp alocate acestora.

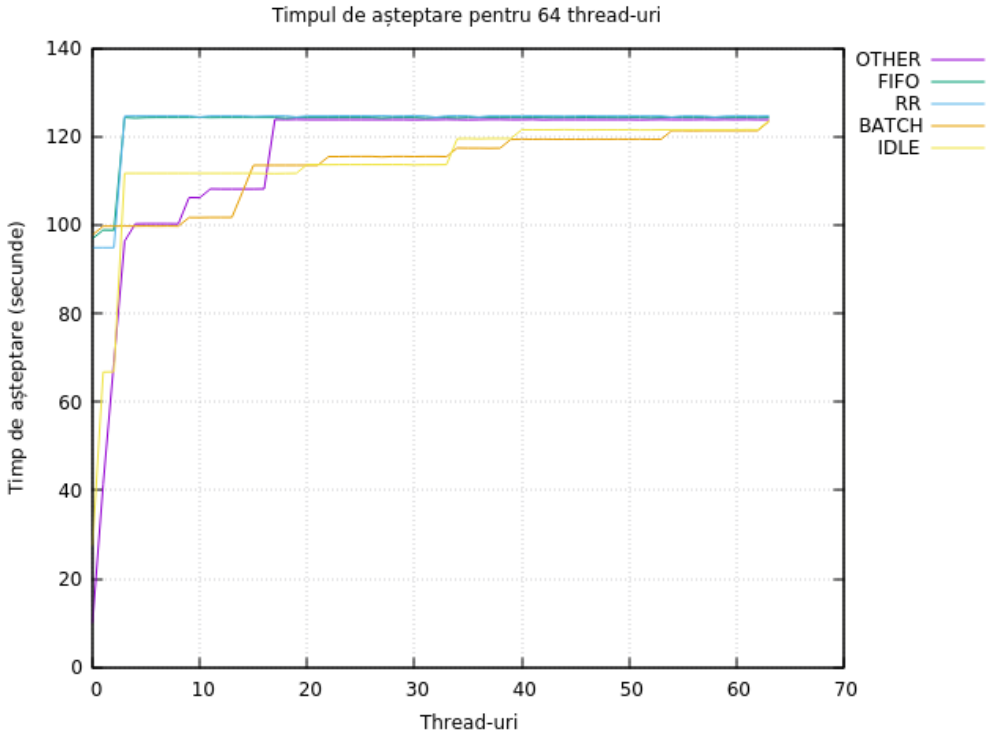


Figura 3.30. Timpul de așteptare pentru 64 de thread-uri cu priorități aleatoare

În acest caz, se obțin valori optime ale timpului de așteptare pentru algoritmul BATCH, urmat de algoritmul IDLE.

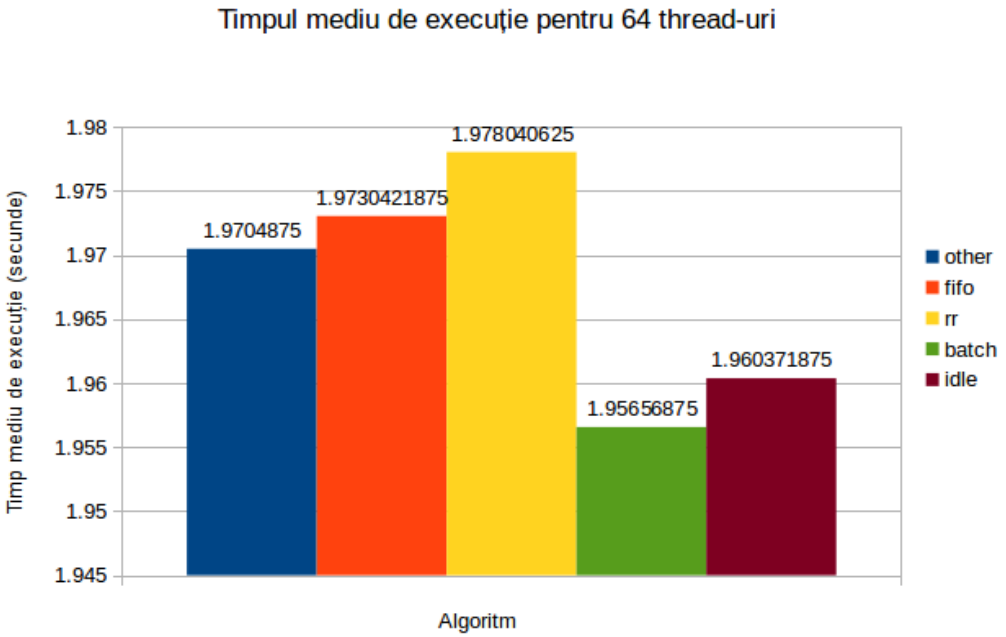


Figura 3.31. Timpul mediu de execuție pentru 64 de thread-uri cu priorități aleatoare

În acest caz, timpul mediu de execuție este minim în cazul algoritmilor BATCH și IDLE și are valori mari pentru algoritmii RR și FIFO.

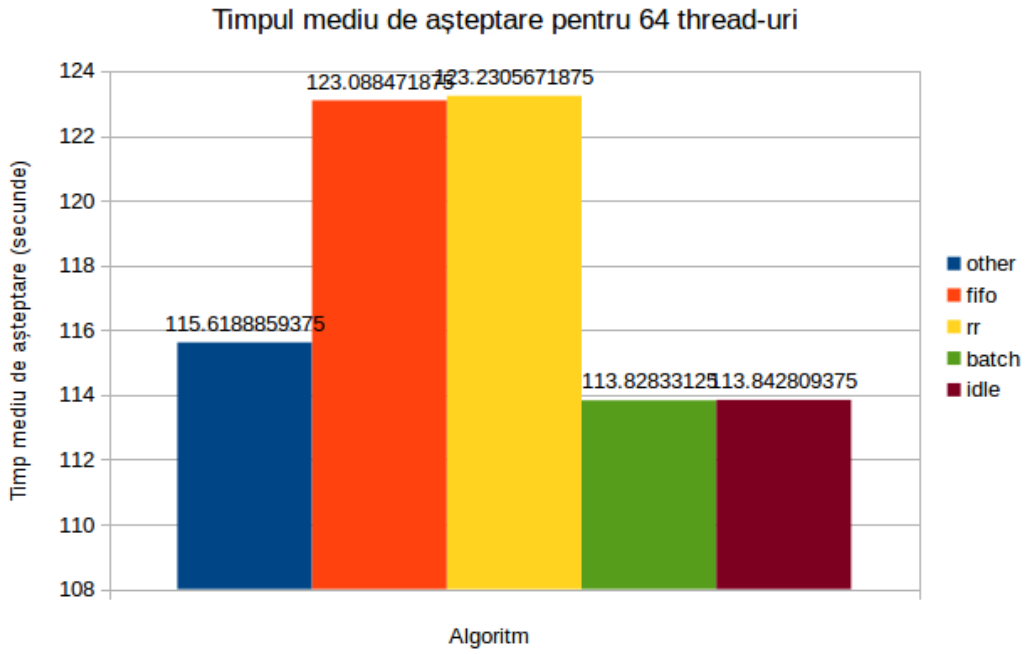


Figura 3.32. Timpul mediu de așteptare pentru 64 de thread-uri cu priorități aleatoare

Se poate observa în figura de mai sus, valori foarte mari ale timpului de așteptare în cazul algoritmilor FIFO și RR și cele mai mici valori pentru algoritmii BATCH și IDLE, asemănător cazului anterior (cu 32 de fire de execuție).

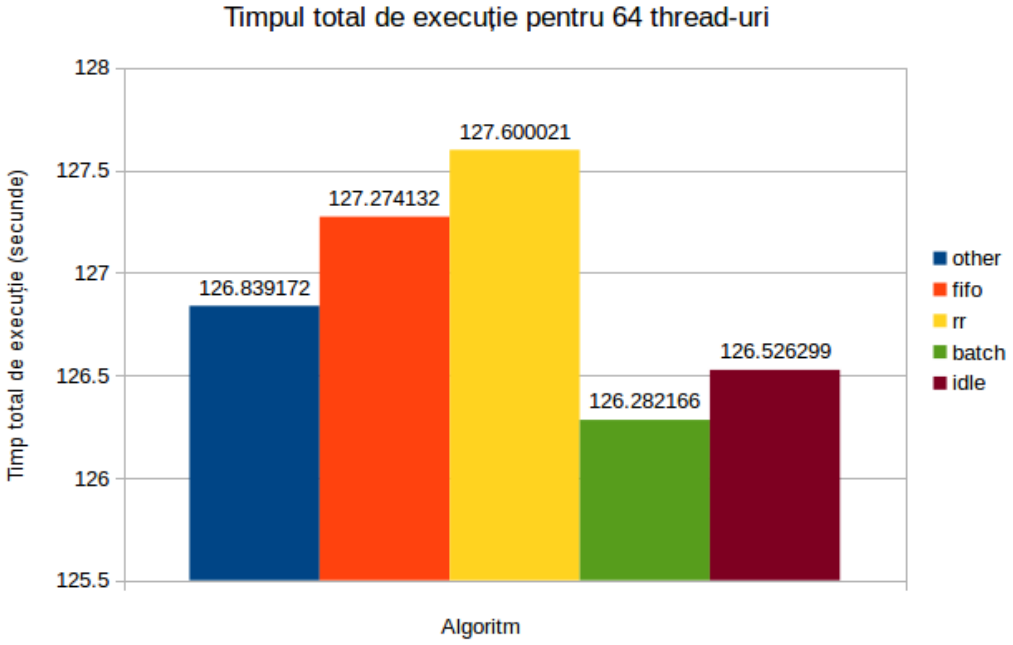


Figura 3.33. Timpul total de execuție pentru 64 de thread-uri cu priorități aleatoare

Timpul total de execuție este optim pentru algoritmul BATCH, algoritmul RR obținând cea mai mare valoare.

G) *Pentru următorul caz, s-au generat 2 fire de execuție pentru algoritmii FIFO și RR cu aceeași prioritate (10). Rezultatele obținute sunt ilustrate în figurile următoare:*

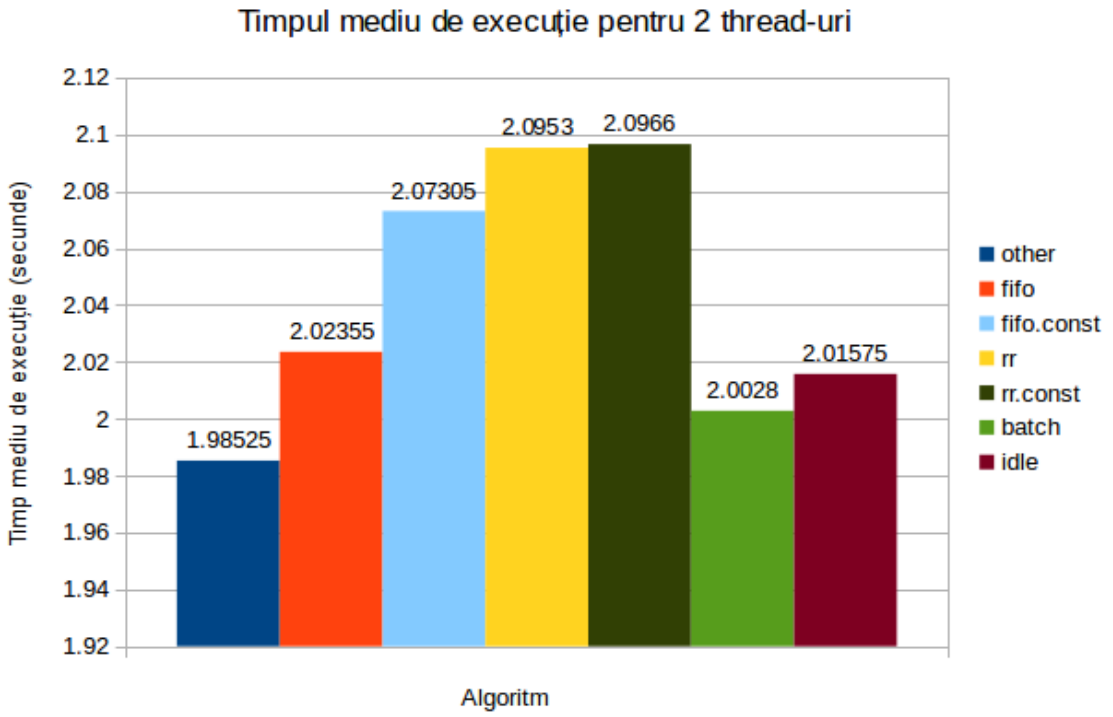


Figura 3.34. Timpul mediu de execuție pentru 2 thread-uri cu prioritate constantă

Din figura de mai sus, se constată o creștere a timpului mediu de execuție în cazul algoritmilor FIFO și RR în situația în care s-a folosit o prioritate constantă cu valoarea 10.

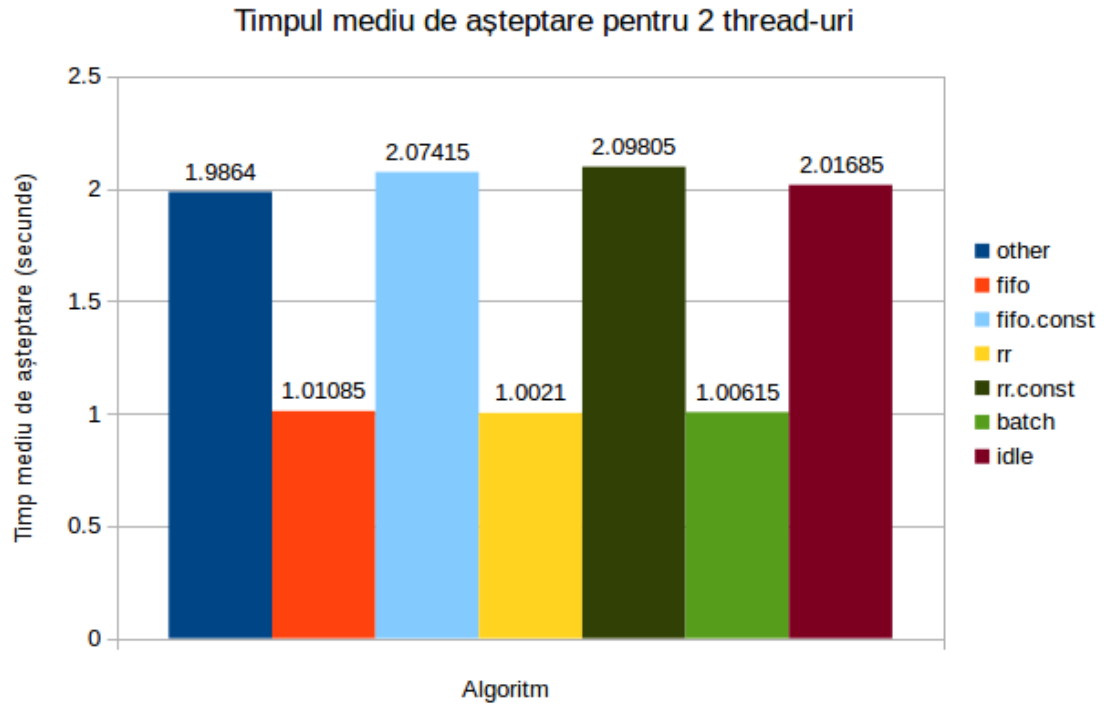


Figura 3.35. Timpul mediu de așteptare pentru 2 thread-uri cu prioritate constantă

Se poate observa în *Figura 3.35* că, în cazul în care păstrăm prioritatea constantă pentru algoritmii FIFO și RR, valorile timpului mediu de așteptare sunt aproximativ duble față de cazul unei valori aleatoare a priorității.

H) *Pentru următorul caz, s-au generat 4 fire de execuție pentru algoritmii FIFO și RR cu aceeași prioritate (10). Rezultatele obținute sunt ilustrate în figurile următoare:*

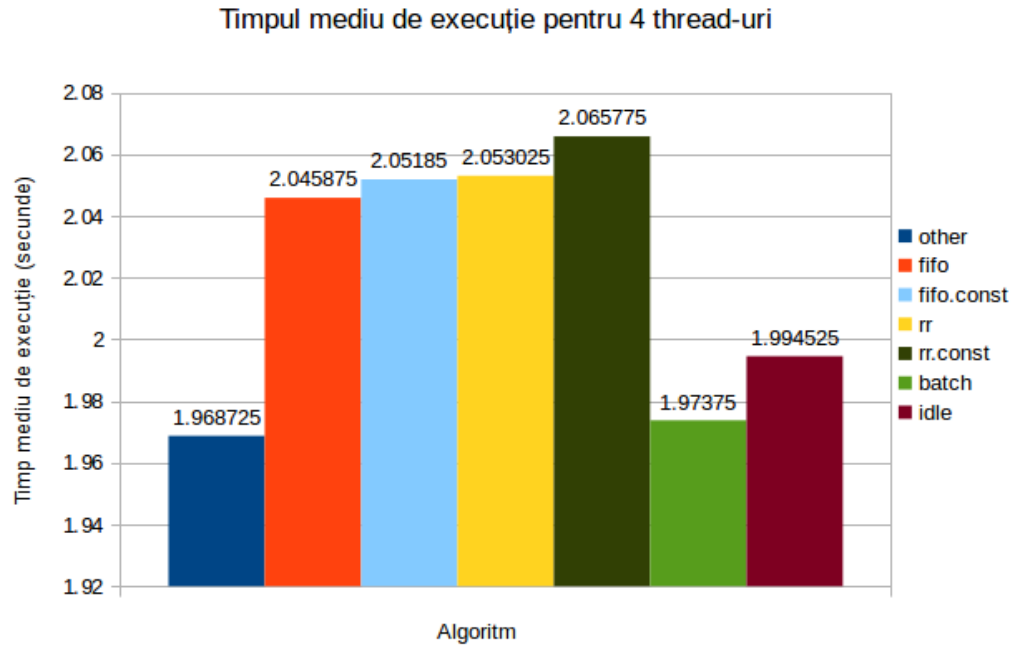


Figura 3.36. Timpul mediu de execuție pentru 4 thread-uri cu prioritate constantă

Și în acest caz, asemănător celui precedent, o valoare constantă a priorității conduce la o creștere a timpului mediu de execuție pentru algoritmii FIFO și RR.

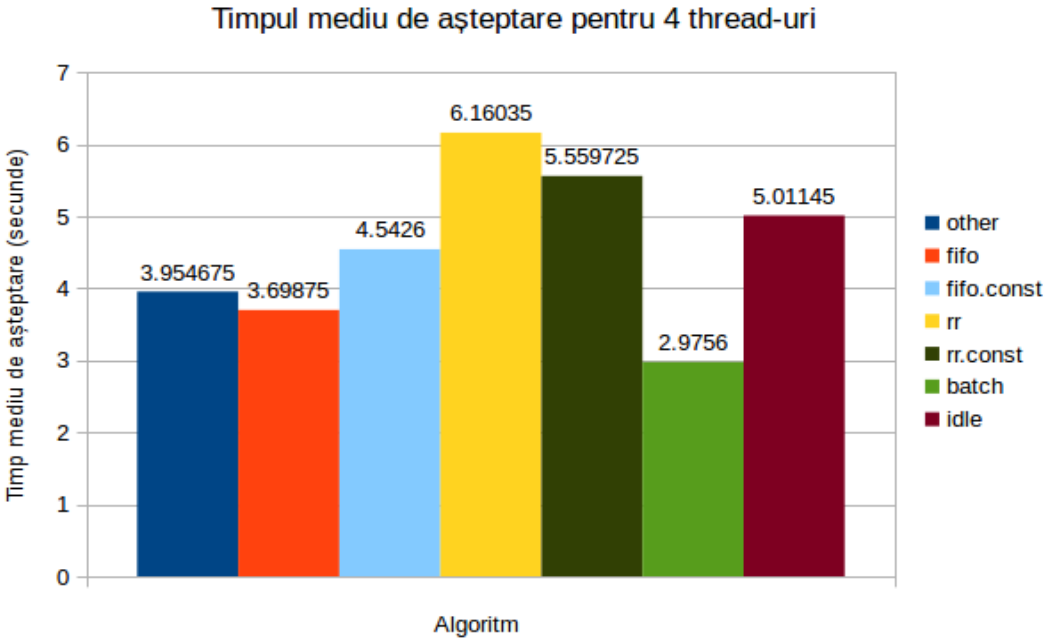


Figura 3.37. Timpul mediu de așteptare pentru 4 thread-uri cu prioritate constantă

Spre deosebire de cazul anterior, timpul mediu de așteptare pentru algoritmul RR este mai mic pentru o valoare constantă a priorității. În ceea ce privește algoritmul FIFO, timpul mediu de așteptare se păstrează la un nivel mai mare decât în cazul precedent.

I) *Pentru următorul caz, s-au generat 8 fire de execuție pentru algoritmii FIFO și RR cu aceeași prioritate (10). Rezultatele obținute sunt ilustrate în figurile următoare:*

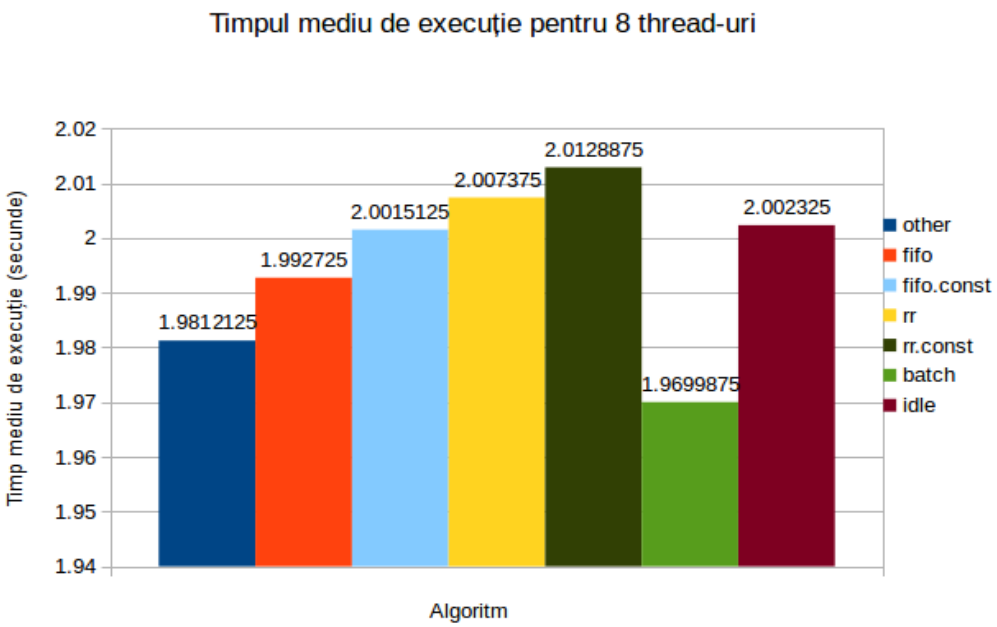


Figura 3.38. Timpul mediu de execuție pentru 8 thread-uri cu prioritate constantă

Ca și în celelalte cazuri anterioare, se constată creșterea timpului mediu de execuție pentru algoritmii FIFO și RR.

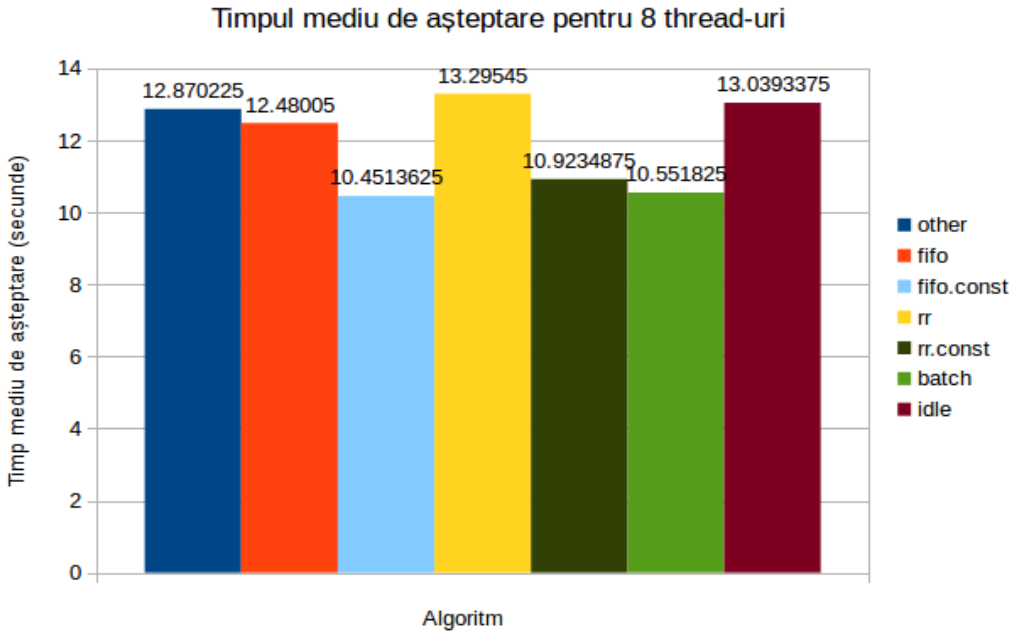


Figura 3.39. Timpul mediu de așteptare pentru 8 thread-uri cu prioritate constantă

În ceea ce privește timpul mediu de așteptare pentru algoritmii FIFO și RR, acesta a scăzut comparativ cu situațiile precedente.

J) *Pentru următorul caz, s-au generat 16 fire de execuție pentru algoritmii FIFO și RR cu aceeași prioritate (10). Rezultatele obținute sunt ilustrate în figurile următoare:*

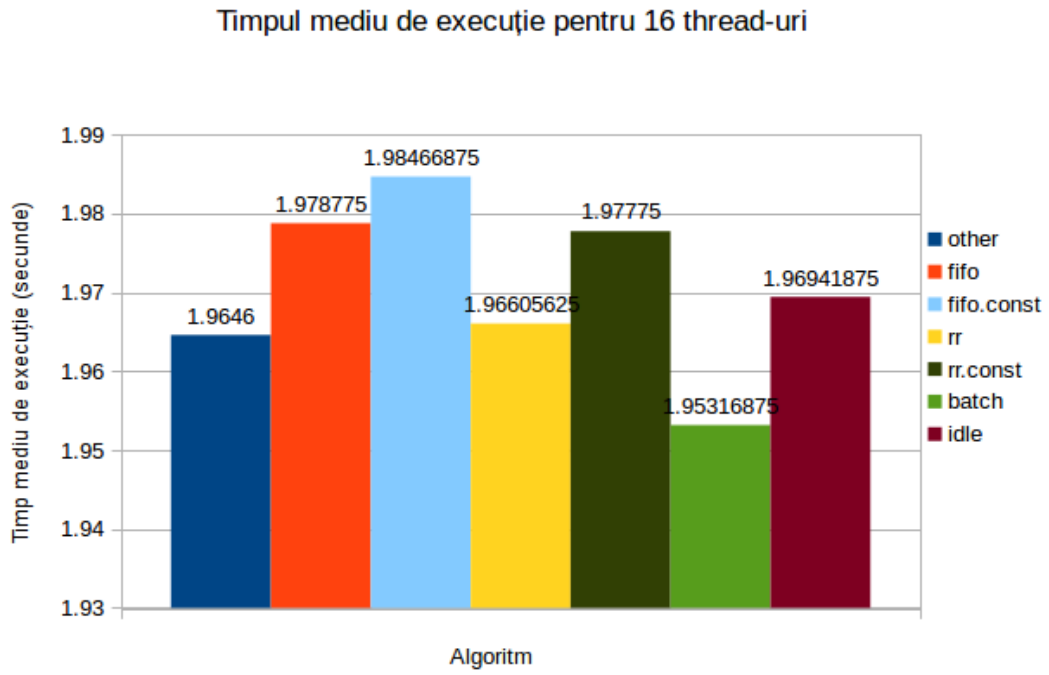


Figura 3.40. Timpul mediu de execuție pentru 16 thread-uri cu prioritate constantă

Se poate observa că timpul mediu de execuție se păstrează la o valoare mai mare pentru algoritmii FIFO și RR în cazul unei priorități constante.

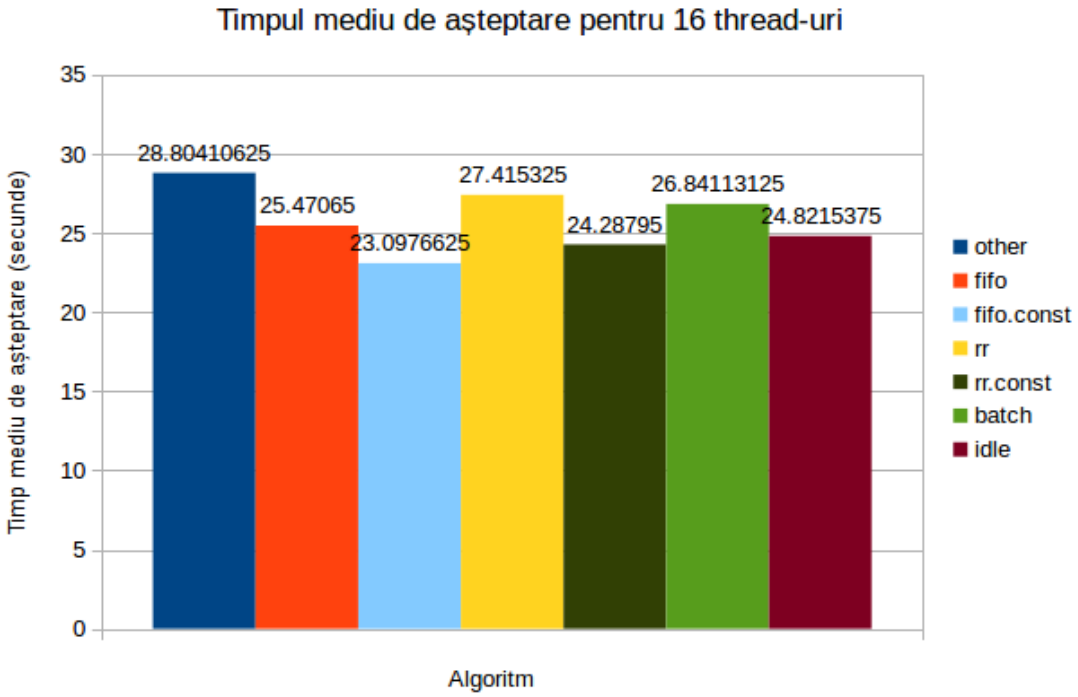


Figura 3.41. Timpul mediu de așteptare pentru 16 thread-uri cu prioritate constantă

De asemenea, timpul mediu de așteptare se menține la o valoare mai mică și în acest caz.

K) *Pentru următorul caz, s-au generat 32 de fire de execuție pentru algoritmii FIFO și RR cu aceeași prioritate (10). Rezultatele obținute sunt ilustrate în figurile următoare:*

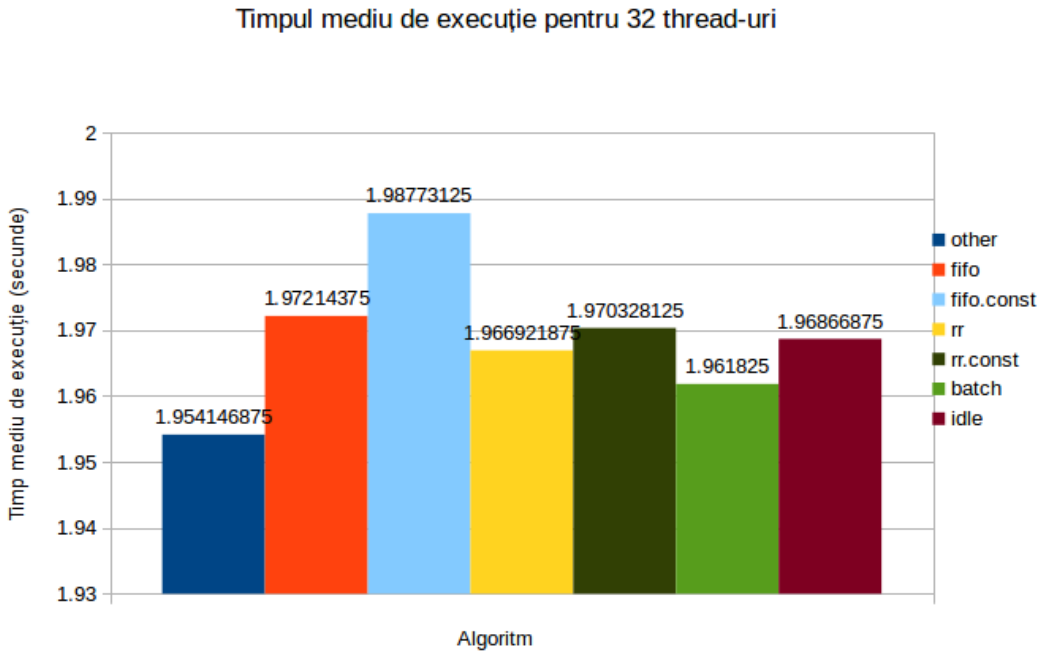


Figura 3.42. Timpul mediu de execuție pentru 32 de thread-uri cu prioritate constantă

Și acest caz, dovedește o eficiență mai bună a algoritmilor FIFO și RR pentru valori aleatoare ale priorității, spre deosebire de menținerea unei valori constante a acesteia.

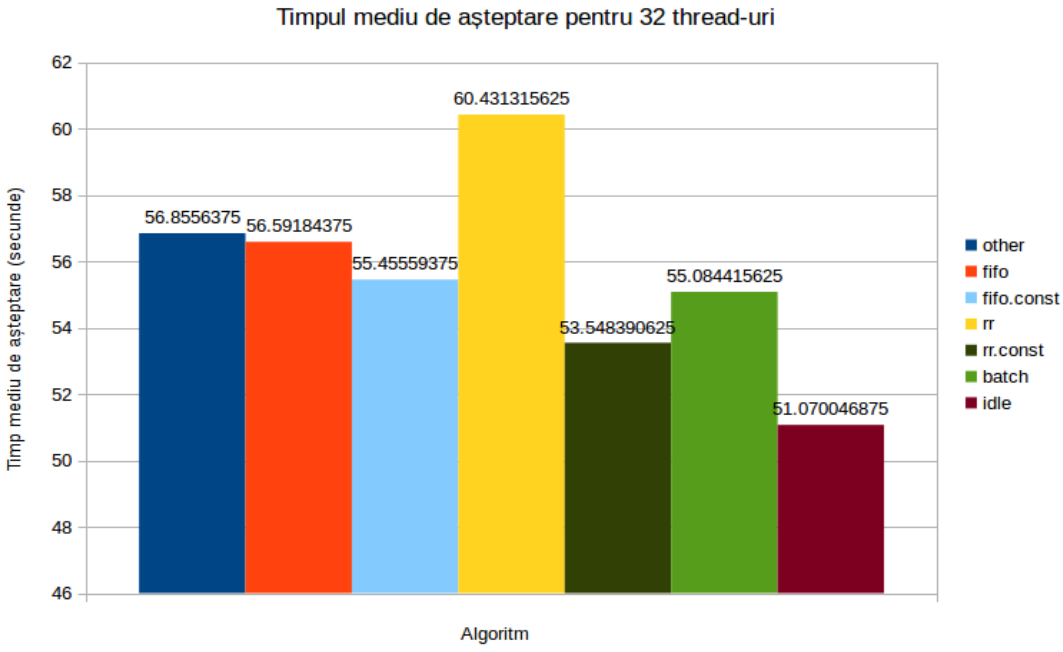


Figura 3.43. Timpul mediu de așteptare pentru 32 de thread-uri cu prioritate constantă

După cum se poate observa în *Figura 3.43*, în cazul algoritmului RR, timpul mediu de așteptare este mult mai mic pentru o valoare constantă a priorității.

L) *Pentru următorul caz, s-au generat 64 de fire de execuție pentru algoritmii FIFO și RR cu aceeași prioritate (10). Rezultatele obținute sunt ilustrate în figurile următoare:*

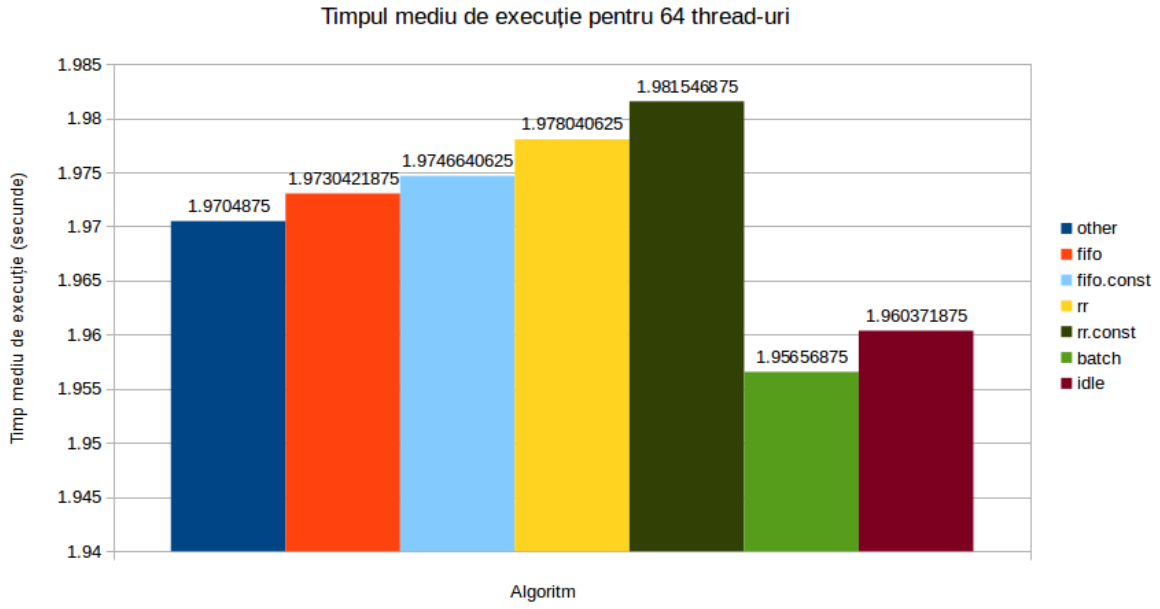


Figura 3.44. Timpul mediu de execuție pentru 64 de thread-uri cu prioritate constantă

Se poate constata că, indiferent de numărul de fire de execuție generate, timpul mediu de execuție este mai mare dacă se folosește o prioritate constantă.

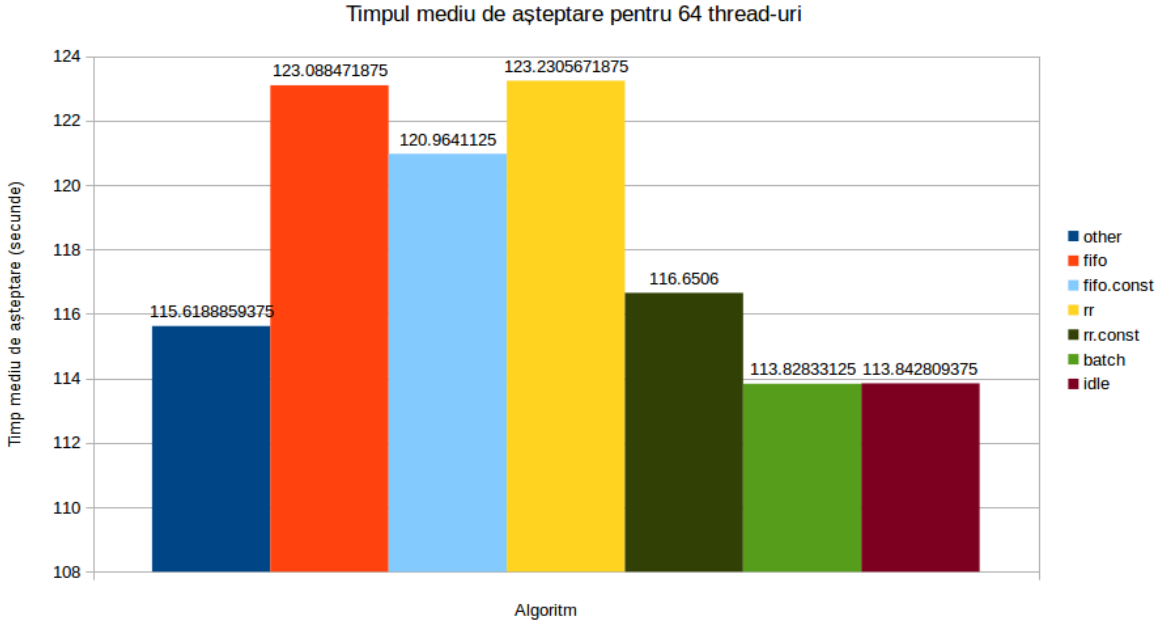


Figura 3.45. Timpul mediu de așteptare pentru 64 de thread-uri cu prioritate constantă

Și în acest caz, timpul mediu de așteptare s-a menținut la o valoare mai mică prin folosirea unei priorități constante.

CONCLUZIILE PROIECTULUI

Obiectivul prezentului proiect, care reprezintă și contribuția personală, a fost acela de a analiza, dezvolta și testa un instrument software în vederea generării unor fire de execuție cu diferiți parametri, pe baza cărora să se poată analiza și compara algoritmii de planificare a rulării firelor de execuție implementați în nucleul Linux.

S-a încercat rezumarea în câteva pagini a ceea ce semnifică și întreprinde vastul domeniu al sistemelor de operare Linux, după care s-au prezentat părțile constituente ale arhitecturii acestora și rezultatele experimentale obținute în urma simulării unor anumite scenarii.

Au fost analizate aspectele teoretice legate de semnificația proceselor și firelor de execuție ca unități de procesare în sistemele de operare Linux curente și totodată, s-au studiat algoritmii de planificare implementați în nucleul Linux și anume: algoritmul *OTHER*, algoritmul *BATCH*, algoritmul *IDLE*, algoritmul *FIFO (First In First Out)* și algoritmul *RR (Round Robin).*

În prezenta lucrare au fost folosite următoarele:

- Mediul de dezvoltare Raspberry Pi versiunea 3 Modelul B;
- Sistemul de operare Raspbian ce are la bază versiunea de nucleu 4.1.19-v7+;
- Instrumentul software dezvoltat, descris în *Anexa A1*;
- Scriptul de reprezentare grafică prezentat în *Anexa A2*.

Pe baza instrumentelor software și hardware enumerate anterior, s-au efectuat o serie de experimente care să testeze funcționalitatea algoritmilor de planificare descriși în *Capitolul 2* (*Planificarea proceselor*). Aceste experimente au constat în generarea unor fire de execuție cu diferiți parametri specifici: algoritmul de planificare și prioritatea respectivelor fire de execuție.

Rezultatele experimentelor au fost grupate într-o serie de cazuri în funcție de numărul firelor de execuție generate și de prioritatea selectată care a fost aleasă în prima fază cu valori pseudo-aleatoare cuprinse între 1 și 99, iar în ce-a de-a doua situație ca având valoarea constantă 10.

Rezultatele obținute în urma experimentelor au fost reprezentate grafic prin intermediul codului sursă din *Anexa A2*, fiind ilustrate de figurile din *Capitolul 3 (Studiu Experimental)*. Respectivele reprezentări grafice sugerează evoluția diferiților timpi caracteristici firelor de execuție în funcție de algoritmii de planificare și prioritățile utilizate.

Parametrii studiați și rezultați din experimentele realizate sunt următorii:

- *Timpul de execuție*: pentru fiecare fir de execuție în parte, reprezentând durata intervalului de timp de la achiziționarea variabilei de blocare (*lock*) care permite firului de execuție să acceseze resursele partajate și până la eliberarea respectivelor resurse;
- *Timpul de așteptare*: pentru fiecare fir de execuție în parte, reprezentând diferența de timp dintre durata de viață a firelor de execuție (de la momentul creării acestora și până la finalizarea lor) și durata de procesare propriu-zisă a acestora;
- *Timpul mediu de execuție*: ca fiind media aritmetică a timpilor de execuție pentru fiecare fir în parte;
- *Timpul mediu de așteptare*: ca fiind media aritmetică a timpilor de așteptare pentru fiecare fir de execuție în parte;
- *Timpul total de execuție*: reprezentat ca durata intervalului de timp de la crearea firelor de execuție și până la finalizarea tuturor acestor fire.

Primele 6 cazuri descrise în *Capitolul 3* cuprind reprezentările grafice ale parametrilor enumerați anterior pentru situația în care s-au folosit, pentru fiecare algoritm în parte, valori pseudo-aleatoare ale priorității și s-a variat numărul firelor de execuție: 2, 4, 8, 16, 32, 64.

S-a constatat că, în ceea ce privește timpul mediu de execuție, cele mai bune rezultate s-au obținut pentru algoritmul *OTHER*, iar cele mai nefavorabile s-au obținut pentru algoritmul *RR*.

În ceea ce privește timpul mediu de așteptare, algoritmi *BATCH* și *IDLE* au obținut cele mai bune rezultate, iar de cealaltă parte, cele mai mari valori ale timpului mediu de așteptare s-au obținut pentru algoritmul *RR*.

Timpul total de execuție s-a dovedit a fi mai mic în cazul algoritmului OTHER și mai mare în cazul algoritmului *FIFO*.

Ultimele 6 cazuri prezentate în *Capitolul 3* cuprind reprezentările grafice ale parametrilor descriși anterior pentru situația în care s-a folosit, pentru fiecare algoritm în parte, o valoare constantă a priorității (10) și s-a variat numărul firelor de execuție: 2, 4, 8, 16, 32, 64.

S-a observat că, în ceea ce privește timpul mediu de execuție pentru algoritmii *FIFO* și *RR*, acesta are valori optime pentru priorități aleatoare. Valorile constante ale priorității în cazul acestor algoritmi conduc la o creștere a timpului mediu de execuție.

Pe de altă parte, utilizarea unei priorități constante a firelor de execuție implică obținerea unor valori mai mici ale timpului mediu de așteptare. Acesta este un avantaj pentru utilizator care nu va sesiza întârzieri în executarea acțiunilor sale.

În dezvoltările ulterioare se va lua în calcul realizarea unei interfețe grafice care să ofere o mai bună interacțiune cu utilizatorul. Totodată, se va avea în vedere dezvoltarea aplicației luându-se în considerare implementarea unor noi algoritmi de planificare, precum și optimizarea părții de procesare în scopul utilizării cât mai eficiente a memoriei utilizate.

BIBLIOGRAFIE

[1] Daniel P. Bovet, Marco Cesati, ”*Understanding the Linux Kernel*”, 3rd Edition, O'Reilly, ISBN: 0-596-00565-2, November 2005;

[2] Michael Kerrisk, ”*The Linux Programming Interface: A Linux and UNIX System Programming Handbook*”, No Starch Press, San Francisco, CA, 2010;

[3] Machtelt Garrels, “*Introduction to Linux - A Hands on Guide*”, version 1.27, ISBN: 1596821124, pages: 9-15, Jun 2008;

[4] Sven Vermeulen, “*Linux Sea*”, ebook, version 1.21, pages: 1-11, 2015;

[5] Paul Cobbaut, “*Linux Fundamentals*”, ebook, pages: 4-12, 2015;

[6] Wolfgang Mauerer, “*Professional Linux kernel architecture*”, Wiley Publishing, Inc., pages: 2-132, 2008;

[7] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, “*Linux Device Drivers*”, 3rd Edition, O’Reilly Media Inc., page: 6, February 2005;

[8] Robert Love, “*Linux Kernel Development*”, 3rd Edition, Addison-Wesley Professional, pages: 4-67, 2010;

[9] Neetu Goel, R.B. Garg, “*A Comparative Study of CPU Scheduling Algorithms*”, International Journal of Graphics & Image Processing, vol. 2, issue 4, pages: 245-251, November 2012;

[10] Dave McCracken, “*POSIX Threads and the Linux Kernel*”, Proceedings of the Ottawa Linux Symposium, Ottawa, Ontario Canada, pages: 330–337, June 26th–29th, 2002;

[11] William Stallings, “*Operating Systems: Internals and Design Principles*”, 8th Edition, Pearson Education, pages: 182-186, 2014;

[12] Zdenek Slanina, Vilem Srovnal, “*Embedded linux scheduler monitoring*”, In 12th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2007), Univ. Patras, Patras, Greece, vol. 1-3, pages: 760 – 763, 25-28 Sept. 2007;

[13] Wenbo Wu, Xinyu Yao, Wei Feng, Yong Chen, “*Research on Improving Fairness of Linux Scheduler*”, In Proceeding of the IEEE International Conference on Information and Automation (ICIA 2013), Yinchuan, China, pages: 409-414, August 2013;

[14] Chandandeep Singh Pabla, “*Completely Fair Scheduler*”, Linux Journal, Belltown Media, Houston Texas, USA, Volume 2009 Issue 184, Article No. 4, August 2009;

[15] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, “*Operating System Concepts*”, 9th Edition, John Wiley & Sons, pages: 265-300, December 2012;

[16] Ekta, Satinder, “*Process scheduling algorithms: a review*”, International Journal of Science, Technology & Management (IJSTM 2015), Volume No. 04, Special Issue No. 01, pages: 24-32, May 2015;

[17] Simone Demblon, Sebastian Spitzner, “*Linux Internals: (to the power of -1)*”, The Shuttleworth Foundation, pages: 80-87, 2005;

[18] Michael Kerrisk, “*The Linux man-pages project*”, release version 4.06, 2015, https://www.kernel.org/doc/man-pages/ accesat la data 11.06.2016;

ANEXE

A1. Codul sursă

```
#include <stdio.h>
#include <pthread.h>
#include <sched.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>

#ifndef SCHED_BATCH
#define SCHED_BATCH     3
#endif
#ifndef SCHED_IDLE
#define SCHED_IDLE      5
#endif

pthread_mutex_t lock;
static pthread_barrier_t bariera; // Bariera de sincronizare pentru thread-uri

static int nrBar;    // Numărul de bariere
int s, nrThreduri;
struct timeval t1, t2;
float t_total, *t_viata, *t_exec, *t_ast, *start, *stop;
clock_t *t_start, *t_stop;
char p, *local_start[100], *local_stop[100];
int prioInit = 10, prio[100];
struct sched_param param; // Structura pentru setarea priorității
int policy;

/* Funcţia pentru calcularea priorităţilor pseudo-aleatoare */
int Prioritate(int pr){
    unsigned int seed;
    prio[pr] = 100 - (rand_r(&seed) % (100 - pr) + pr);
    return prio[pr];
}

/* Funcţia pentru calcularea timpului de start şi de stop al thread-urilor */
float Timp(int thread, int m){
    float sec;
    struct timeval tv;
    gettimeofday(&tv, NULL);
    time_t ctime = tv.tv_sec;
    struct tm *t = localtime(&ctime);
    if (m == 0)
        asprintf(&local_start[thread],     "%02d:%02d:%02d:%03d",     t->tm_hour,
t->tm_min,   t->tm_sec,   tv.tv_usec/1000);   //   Timpul   de   start   în   format:
HH:MM:SS.mmm.
    else
        asprintf(&local_stop[thread],     "%02d:%02d:%02d:%03d",     t->tm_hour,
t->tm_min, t->tm_sec, tv.tv_usec/1000);

    sec = tv.tv_sec + tv.tv_usec*0.000001;
    return sec;
}
```

```
/* Funcţia thread-urilor. Specifică activitatea acestora */
static void *Operatii(void *arg){
    int s, br, nsecs;
    clock_t x, y;
    int idThread = (int) arg;
    pid_t self_id = getpid(); // Aflarea id-ului thread-ului
    if (policy == SCHED_FIFO || policy == SCHED_RR)
        param.sched_priority = Prioritate(idThread); // Setarea priorităţii
    else
        param.sched_priority = 0;

    if(sched_setscheduler(0, policy, &param) != 0){ // Setarea algoritmului de
planificare
        printf("Eroare la setarea algoritmului!\n");
        perror("");
        exit(1);
    }

    srandom(time(NULL) + idThread);

    for (br = 0; br < nrBar; br++) {


        /* Fiecare thread aşteaptă până ce toate thread-urile ajung la barieră
după care îşi continuă execuţia. */
        s = pthread_barrier_wait(&bariera);

    }

    nsecs = random() % 20 + 1;       // Aşteaptă între 1 şi 20 de secunde
    sleep(nsecs);
    pthread_mutex_lock(&lock);
    x = clock();  // Se interoghează timpul de start al execuţiei
    int n = 300;    // Dimensiunea matricilor: 300 x 300.
    int i, j,  k;
    float **a, **b, **c;
    a = (float**)malloc(sizeof(float*)*n);
    b = (float**)malloc(sizeof(float*)*n);
    c = (float**)malloc(sizeof(float*)*n);
    for (i = 0; i < n; i++){
        a[i] = (float*)malloc(sizeof(float)*n);
        b[i] = (float*)malloc(sizeof(float)*n);
        c[i] = (float*)malloc(sizeof(float)*n);
    }
    for (i = 0; i < n; i++)
        for(j = 0; j < n; j++){
            a[i][j] = rand()/(float)RAND_MAX;
            b[i][j] = rand()/(float)RAND_MAX;
        }
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++){
            c[i][j] = 0;
            for (k = 0; k < n; k++)
                c[i][j] += a[i][k]*b[k][j];
        }
    pthread_mutex_unlock(&lock);
    y = clock();        // Se interoghează timpul de stop al execuţiei
    t_exec[idThread] = ((float)(y - x))/CLOCKS_PER_SEC;

    return NULL;
}
```

```
int main(int argc, char *argv[]){
    nrBar = atoi(argv[1]);  // Numărul de bariere introdus de la tastatură.
    p = atoi(argv[2]);     // Algoritmul de planificare introdus de la tastatură.
    nrThreduri = atoi(argv[3]);     // Numărul de thread-uri introdus de la
tastatură.
    int idThread;
    pthread_t *tid;
    struct timespec ts;
    int rr;
    char const *fn = "REZULTATE.txt"; // Fișierul de stocare al rezultatelor.
    FILE *fp;
    int concurrency;
    concurrency = nrThreduri;
    pthread_setconcurrency(concurrency);
    if (p == 0) policy = SCHED_OTHER;
    else if (p == 1) policy = SCHED_FIFO;
    else if (p == 2) policy = SCHED_RR;
    else if (p == 3) policy = SCHED_BATCH;
    else if (p == 5) policy = SCHED_IDLE;
    tid = calloc(sizeof(pthread_t), nrThreduri);
    if (tid == NULL)
        printf("EROARE | calloc");
    t_viata = calloc(sizeof(float), nrThreduri);
    t_start = calloc(sizeof(clock_t), nrThreduri);
    t_stop = calloc(sizeof(clock_t), nrThreduri);
    t_exec = calloc(sizeof(float), nrThreduri);
    t_ast = calloc(sizeof(float), nrThreduri);
    start = calloc(sizeof(float), nrThreduri);
    stop = calloc(sizeof(float), nrThreduri);
    s = pthread_barrier_init(&bariera, NULL, nrThreduri); // Inițializarea
barierei de sincronizare.
    if (s != 0)
        printf("EROARE | %d pthread_barrier_init", s);

    /* Create 'nrThreduri' threads */
    fp = fopen(fn, "a"); // Deschiderea fișierului text.
    printf(" _____________________________________________________________________\n");
    printf("| ID  |    Start     |     Stop     |   Executie   | Asteptare  |
Durata de viata | Prioritate |\n");

printf("|___|_____________|_____________|_____________|_____________|_____________|_____________|\n");
    fprintf(fp, "\nID\t\t Start\t\t Stop\t\t\t Executie\t\tAsteptare\t\t Durata
de viata\t\t Prioritate\n");  // Scrierea rezultatelor în fișier.

    pthread_mutex_init(&lock, NULL);  // Inițializarea mutexului.
    gettimeofday(&t1, NULL);

     /* Generează thread-urile. */
    for (idThread = 0; idThread < nrThreduri; idThread++) {
        t_start[idThread] = clock();
        start[idThread] = Timp(idThread, 0);
        s = pthread_create(&tid[idThread], NULL, Operatii, (void *) idThread);
        if (s != 0)
            printf("EROARE | %d pthread_create", s);
    }

    /* Așteaptă ca toate thread-urile să se finalizeze. */
    for (idThread = 0; idThread < nrThreduri; idThread++) {
        s = pthread_join(tid[idThread], NULL);
        if (s != 0)
            printf("EROARE | %d pthread_join", s);
        t_stop[idThread] = clock();
        stop[idThread] = Timp(idThread, 1);
```

```
        t_viata[idThread]          =          ((float)(t_stop[idThread]          -
t_start[idThread]))/CLOCKS_PER_SEC;
        t_ast[idThread] = t_viata[idThread] - t_exec[idThread];
        fprintf(fp, "%02d\t\t%s\t\t%s\t\t%.4f\t\t %.4f\t\t\t %.4f\t\t\t\t\t\t%d\n",
idThread,    local_start[idThread],    local_stop[idThread],    t_exec[idThread],
t_ast[idThread], t_viata[idThread], prio[idThread]);
        printf("| %02d  | %s | %s |    %.4f    |   %.4f    |     %.4f      |     %d
|\n",  idThread,  local_start[idThread],  local_stop[idThread],  t_exec[idThread],
t_ast[idThread], t_viata[idThread], prio[idThread]);

printf("|__|_________|__________|_________|_________|__________|__________|\n");
    }
    gettimeofday(&t2, NULL);

    if (policy == SCHED_RR){
        rr = sched_rr_get_interval(0, &ts);
        printf("Cuanta   de   timp:   %lu   milisecunde.\n",   ts.tv_sec*1000   +
ts.tv_nsec/1000000);
    }
    t_total = (t2.tv_sec - t1.tv_sec + 0.000001*(t2.tv_usec - t1.tv_usec));
    printf("\n\nTimpul total de executie: %f secunde.\n", t_total);
    fclose(fp);
    pthread_mutex_destroy(&lock);
    free(tid);
    exit(EXIT_SUCCESS);
}
```

A2. Codul de reprezentare grafică

S-a folosit utilitarul gnuplot împreună instrucțiunile sale[2] pentru realizarea scriptului:

```
  set   autoscale   # setare automată a axelor
  set xtic auto
  set ytic auto
  set title "Timpul de execuție pentru 64 thread-uri"
  set xlabel "Thread-uri"
  set ylabel "Timp de execuție (secunde)"
  set key outside;
  set key right top;
  set label "Yield Point" at 0.003,260
  set arrow from 0.0028,250 to 0.003,280
  set grid
  plot "exec_64v" using 1:2 title 'OTHER' with lines, \
      "exec_64v" using 1:3 title 'FIFO' with lines, \
      "exec_64v" using 1:4 title 'RR' with lines, \
      "exec_64v" using 1:5 title 'BATCH' with lines, \
      "exec_64v" using 1:6 title 'IDLE' with lines
```

[2] http://www.gnuplot.info/