UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

EVALUAREA PERFORMANȚEI ALGORITMILOR DE PLANIFICARE A PROCESELOR

Absolvent

Stan Cătălin-Andrei

Coordonator științific

Conf. univ. dr. ing. Paul Irofti

București, iunie 2024

Rezumat

Această lucrare vine cu o soluție la problema evaluării performanței algoritmilor de planificarea a proceselor într-un sistem de operare, prezentând un simulator destinat comparării algoritmilor, o serie dintre ei fiind deja integrați în aplicație. Este expus un API care are scopul de a oferi posibilitatea implementării unui nou algoritm, făcând abstracție de modul de operare al simulatorului. Modul de generare al datelor, vizualizarea grafică a rezultatelor cât și modul facil de comparare al algoritmilor între ei conduc la identificarea rapidă a plusurilor și minusurilor acestora. Aplicația suportă toate cazurile de interes ale unui posibil utilizator, uni-procesor, multi-procesor și timp real.

Abstract

This paper comes up with a solution to the problem of evaluating the performance of CPU scheduling algorithms in operating system by provind a simulator aimed at comparing algorithms, a number of which are already integrated into the application. An API is exposed that which provides the possibility of implementing a new algorithm, abstracting the inner workings of the simulator. The methods used for data deneration, the graphical visualization of the results and the easy way of comparing the algorithms with each other lead to quick identification of their pluses and minuses. The application supports all cases of interest of a possible user single core, multi core and real time.

Cuprins

Capitolul 1

Introducere

1.1 Motivație

Puterea de calcul este unul din factorii cheie care influențează progresul tehnologic accelerat la care asistăm astăzi. Acest lucru nu a fi posibil fără unitățile de calcul care sunt capabile să aloce resurse mai multor procese, executându-le concurent. Sistemul de operare este responsabil de această funcție fundamentală, folosindu-se de un algoritm de planificare al proceselor care decide ce proces se execută și pentru cât timp. Concluzia este evidentă, performanța unei unități centrale de procesare este strâns corelată cu algoritmul folosit. Pentru a ști cu adevărat care sunt rezultatele unui anumit algoritm de interes, acesta ar trebui implementat într-un sistem de operare și supus testelor aferente, dar este o muncă dificilă predispusă la multe erori pe parcurs. De aceea avem nevoie de o metodă rapidă care să ofere un set de rezultate aproximative și verosimile.

1.2 Contribuții

În această lucrare prezint un simulator care scopul de a evalua algoritmi de planificare al proceselor pentru toate cele trei cazuri, uni-procesor, multi-procesor și în timp real. Aplicația suportă toate funcțiile de care un utilizator are nevoie, implementarea unui algoritm nou prin intermediul unei interfețe, generarea unui set de date sau încărcarea lui dintr-un fișier și crearea automată a unui raport ce conține performanțele algoritmului. Simulatorul dispune de o colecție de algoritmi și de o metodă de comparare a acestora între ei, cât și cu cei implementați de utilizator. De asemenea, pentru algoritmii deja integrați în aplicație am dedicat un capitol prezentării lor și unul evaluării performanței acestora menit să demonstreze capabilitățile simulatorului. În cazul încărcării unui set de date dintr-un fișier și numărul de procese este mic se va genera și o diagramă *Gantt* corespunzătoare execuției proceselor. Simulatorul poate fi accesat la adresa `https://github.com/cata1212112/SchedulingSimulator`.

1.3 Metodologie

Simulatorul va fi realizat în *C++* folosind paradigma programării orientate pe obiecte, procesând evenimente discrete precum expirarea cuantei de timp a unui proces. În cazul general se vor simula n procesoare, fiecare fiind o instanță a unui simulator de evenimente discrete, rulând pe un fir de execuție separat, iar coordonarea acestora în cazul evenimentelor globale, precum intrarea unui proces în sistem și operația de balansare a muncii între procesoare, se va face folosind elemente de sincronizare standard precum barierele reutilizabile. Evaluarea algoritmilor este realizată prin generarea unor seturi de date adecvate fiecărui tip, timpii de execuție sunt obținuți din mai multe distribuții normale pentru cazurile uni-procesor și multi-procesor, iar pentru cazul în timp real se vor genera mulțimi de procese care au proprietățile necesare unei posibile planificări, folosind un algoritm specializat. Toți parametrii sunt introduși de utilizator prin intermediul unei interfețe grafice creată folosind *framework*-ul *QT* [15], iar vizualizarea metricilor este realizată folosind librăria *Matplotlib* [21] prin intermediul unui fir de execuție ce rulează Python.

1.4 Structura lucrării

În **Capitolul 2** voi prezenta noțiunile teoretice care stau la baza interpretării rezultatelor cât și pentru înțelegerea modului de funcționare și a motivării din spatele algoritmilor prezentați detaliat în **Capitolul 3** care sunt deja implementați în aplicație. **Capitolul 4** conține arhitectura simulatorului, modul de generare al datelor, cum se realizează evaluarea și maniera în care se poate integra un algoritm nou. Performanțele algoritmilor deja implementați, rezultate în urma unor cazuri de interes, se pot analiza în **Capitolul 5**.

1.5 Stadiu actual

SimSo [3] este o bibliotecă *python* care conține un simulator multi-procesor în timp real ce oferă o colecție extinsă de algoritmi, multiple metode pentru generarea unui set de date și implementarea cu ușurință a unui nou algoritm prin moștenirea unei clase de bază. În cazul uni-procesor există multiple website-uri precum [4], dar acestea dispun doar de algoritmii clasici de planificare, fără posibilitatea extinderii acestei liste, iar evaluarea se face introducând datele de mână.

Capitolul 2

Preliminarii

Pentru a putea înțelege algoritmii analizați care sunt metricile după care sunt evaluați voi prezenta suportul teoretic pe care este bazat acest subdomeniu al sistemelor de operare.

Într-un sistem de operare, procesele alternează între a aștepta un eveniment de tip I/O și un ciclu de execuție pe procesor, care, în literatură, se numește *burst time*, termen ce îl voi folosi în restul acestei lucrări. În acest simulator nu am modelat așteptarea de tip I/O, așadar, fiecare proces are un singur *burst time*, un timp la care a intrat în sistem, cunoscut sub denumirea de *arrival time*, și o prioritate. Procesele care nu sunt executate la momentul curent se află într-o coadă de așteptare numită *ready queue*.

Algoritmii se pot împărții în două categorii principale, *non-preemtive*, când procesul executat nu poate fi evacuat până nu-și termină întregul *burst time*, și *preemtive*, care permite suspendarea în timpul execuției.

2.0.1 Uni-procesor

În secțiunea 3.1.1 voi face referire la algoritmii *Round Robin* [17], *First In First Out* [16], respectiv *Shortest Job First* [18] așa că îi voi descrie sumar, împreună cu exemple de execuție pentru mulțimea de procese din tabela următoare

ID	Arrival Time	Burst Time
P1	0	7
P2	0	4
P3	1	5
P4	1	3
P5	1	12

Tabela 2.1: Procese exemplu.

- *First Come First Serve*
  - Algoritmul alocă procesorul procesului care a ajuns cel mai devreme în *ready queue* și îl execută până la finalizarea timpului acestuia de execuție în mod *non-preemtive*. Diagrama Gantt generată în urma execuției se poate observa în figura următoare

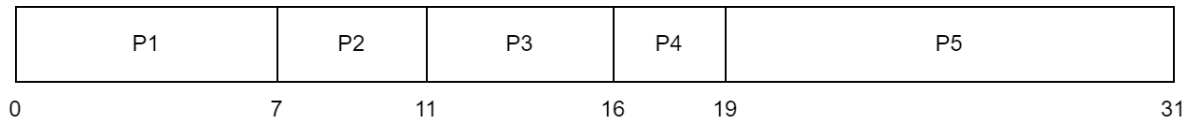

Figura 2.1: Diagrama Gantt a execuției folosind algoritmul First Come First Serve

- *Shortest Job First*
  - Algoritmul este similar cu cel precedent, doar că în acest caz este selectat procesul cu *burst time* minim și îl execută până la finalizare. Diagrama Gantt a execuției se poate observa în figura următoare

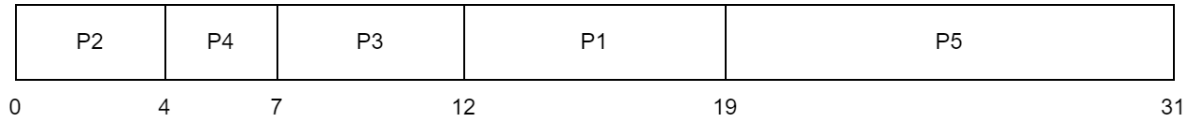

Figura 2.2: Diagrama Gantt a execuției folosind algoritmul Shortest Job First

- *Round Robin*
  - Acest algoritm este de tipul *preemtive*, permițând proceselor să execute pentru o perioadă limitată după care selectat alt proces. Dacă procesul executat nu și-a terminat *burst time*-ul el este plasat din nou *ready queue*. Limita de timp este o constantă numită cuantă. În figura următoare se poate observa execuția proceselor unde cuanta de timp a fost aleasă 3. La momentul de timp 6, procesul *P2* ar mai fi avut doar o unitate de timp de executat, dar cum cuanta a expirat, el trebuie să aștepte după toate celelalte procese până la timpul 18. Analog se întâmplă și pentru procesul *P1* la momentul de timp 15, respectiv 24.

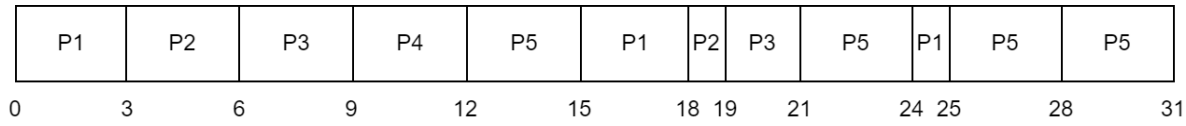

Figura 2.3: Diagrama Gantt a execuției folosind algoritmul Round Robin cu cuanta de timp 3.

În cazul algoritmilor uni-procesor vom evalua performanța acestora în funcție de următoarele criterii:

Turnaround time definit ca diferența între timpul la care procesul este finalizat și timpul la care a intrat în sistem.

Proces	FCFS	SJF	RR
P1	7	19	25
P2	11	4	19
P3	15	11	20
P4	18	6	11
P5	30	30	30
Media	14.2	14	21

Tabela 2.2: Turnaround time pentru fiecare proces în funcție de algoritm.

Waiting time definit ca timpul pe care procesul îl petrece în *ready queue*, anume diferența între *turnaround time* și *burst time*.

Proces	FCFS	SJF	RR
P1	0	12	18
P2	7	0	15
P3	10	6	15
P4	15	3	8
P5	18	18	18
Media	10	7.8	14.8

Tabela 2.3: Waiting time pentru fiecare proces în funcție de algoritm.

Response time definit ca primul moment de timp când procesul este executat.

Proces	FCFS	SJF	RR
P1	0	12	0
P2	7	0	3
P3	11	7	6
P4	16	4	9
P5	19	19	12
Media	10.6	8.4	6

Tabela 2.4: Response time pentru fiecare proces în funcție de algoritm.

Context Switches definit ca numărul trecerilor procesorului de la execuția unui proces la altul.

CPU Utilization definit ca procentul din timpul total de rulare al procesorului în care acesta a avut un proces activ. În cazul exemplului nostru, pentru fiecare algoritm procesorul execută tot timpul, deci este la 100%.

Algoritm	Schimbări de context
FCFS	4
SJF	4
RR	10

Tabela 2.5: Numărul de schimbări de context pentru fiecare algoritm.

2.0.2 Multi-procesor

În acest simulator voi trata cazul procesoarelor simetrice și nu voi lua în considerare afinitatea proceselor.

Schema generală abordată pentru acest caz este acela de a avea o coada locală de procese pentru fiecare procesor. O problemă fundamentală este cea în care un procesor are o sarcină mai mare decât celelalte, fenomen numit *load imbalance*. Soluția constă în stabilirea unui comportament de migrare al proceselor. În plus, trebuie să ne asigurăm că fiecare proces primește timp de execuție în mod echitabil, proporțional cu prioritatea acestuia.

Suportul teoretic care stă în spatele aigurării echitabilității este *Generalized Processor Sharing* [13], un algoritm ideal, care atinge egalitate perfectă între procese, cum acesta nu poate fi implementat în realitate, diferiți algoritmi au fost dezvoltați pentru a-l aproxima. În principiu, prioritatea pentru fiecare proces este un număr între -20 și 19, iar pentru fiecare prioritate $p_i \in \{-20, -19, -18, \dots, 18, 19\}$ este definit o pondere w_i, astfel pentru a asigura echitabilitatea dorim ca

$$\frac{S_i(t_1, t_2)}{S_j(t_1, t_2)} = \frac{w_i}{w_j} \tag{2.1}$$

unde $S_i(t_1, t_2)$ este timpul alocat pe procesor procesorului i în intervalul $[t_1, t_2]$. O proprietate care rezultă este aceea că

$$S_i(t_1, t_2) = \frac{w_i}{\sum_{j=1}^{n} w_j} * (t_2 - t_1) * P \tag{2.2}$$

unde P este numărul de procesoare.

În practică, pentru fiecare proces este ținut *virtual runtime*-ul acestuia definit ca $VR_i(t) = \frac{S_i(0,t)}{w_i}$, anume, timpul total alocat pe procesor scalat cu prioritatea. O metrică după care putem evalua algoritmii pentru acest caz este diferența maximă între timpii virtuali de rulare. Cu cât diferența este mai mică cu atât algoritmul este mai echitabil.

Cel mai cunoscut algoritm multi-procesor este *Completely Fair Scheduler* implementat în sistemul de operare *Linux*. Fiecare procesor are propria coadă de execuție și o constantă *P*, iar fiecărui proces îi este alocat o cuantă de timp din P, proporțională cu ponderea

acestuia în felul următor

$$timeslice_{P_i} = \frac{w_{P_i.priority}}{\sum_{p \in readyQueue} p.priority} \times P \tag{2.3}$$

Astfel se asigură echitabilitatea la nivel de procesor. La nivel de sistem de sistem există se execută periodic operația de *load balancing* care migrează procese pentru a distribui în mod egal munca. În plus această procedură este efectuată și în momentul în care un procesor devine inactiv.

2.0.3 Timp real

În cazul sistemelor de operare în timp real fiecare proces este periodic, are un timp de execuție și o limită de timp până când trebuie finalizat. În literatură, pentru a distinge procesele normale de cele periodice, cele din urmă sunt denumite *task*-uri.În această lucrare voi considera că toate procesele au ajuns la momentul de timp 0 și voi nota cu $\Gamma = \{\tau_i \mid 1 \leq i \leq n\}$ mulțimea de procese. Fiecare τ_i este definit de următoare caracteristici, C_i timpul de execuție, D_i timpul maxim de finalizare relativ la timpul de intrare în sistem, numit în literatură *deadline*, T_i perioada, r_i timpul de intrare în sistem. În continuare am considerat că D_i coincide cu T_i $\forall 1 \leq i \leq n$. Se definește utilizarea procesului τ_i raportul dintre C_i și D_i și se notează cu u_i, iar utilizarea mulțimii Γ ca $U = \sum_{i=1}^{n} u_i$.

Condițiile necesare ca mulțimea Γ să poată fi planificată sunt:

$$u_i = \frac{C_i}{D_i} \leq 1 \quad \forall\ 1 \leq i \leq n \tag{2.4}$$

$$U \leq m \tag{2.5}$$

Am notat cu m numărul de procesoare disponibile. Deoarece procesele sunt periodice, sistemul ar rula la infinit, dar este necesară observarea comportamentului algoritmului doar pe perioada de timp $[0, \mathcal{H})$, unde $\mathcal{H} = \text{lcm}(\{T_i \mid 1 \leq i \leq n\})$ este hiper-perioada mulțimii.

Pentru a înțelege algoritmii prezentați trebuie introdus conceptul de *laxity* sau *slack* care reprezintă cât de mult mai poate fi amânat un proces, astfel, pentru momentul de timp t, $laxity_{\tau_i} = D_i^{absolute} - C_i^{remaining}$, unde $C_i^{remaining}$ reprezintă cât timp de execuție mai are rămas, iar $D_i^{absolute}$ este timpul la care expiră *deadline*-ul relativ la momentul 0.

Pentru a înțelege mai bine elementele definite anterior le putem vizualiza în figura 2.4.

Algoritmii clasici pentru acest caz sunt *Earliest Deadline First* [19], care la fiecare unitate de timp execută procesul cu *deadline-ul* cel mai mic și *Rate-Monotonic* [20], care

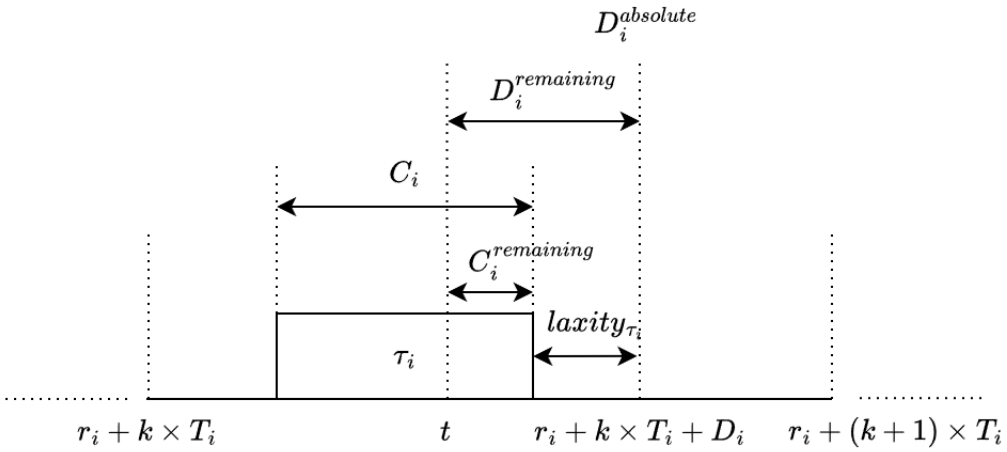

Figura 2.4: Vizualizarea elementelor folosite pentru cazul Real Time.

la fiecare unitate de timp execută procesul cu cea mai mică perioadă.

Capitolul 3

Algortimi

3.1 Uni-procesor

3.1.1 Efficient Dynamic Round Robin

În lucrarea *An Efficient Dynamic Round Robin Algorithm for CPU scheduling* [6], autorii propun o versiune modificată a algoritmului *Round Robin*, în care cuanta de timp este calculată în mod dinamic în funcție timpul de execuție al proceselor din *ready queue*. Motivația din spatele cuantei dinamice constă în faptul că performanța *Round Robin* este dată de cuanta aleasa, mai precis, o cuantă mare va induce algoritmului să se comporte precum *First Come First Serve*, iar o cuantă mică va produce un număr mare de schimbări de context. Problema algoritmului *First Come First Serve* este media ridicată a timpilor de așteptare și de finalizare atunci când procese scurte succed proceselor lungi.

Autorii propun un calcul al cuantei care să minimizeze atât numărul de schimbări de context cât și timpii medii de așteptare și finalizare. În plus, au vrut să păstreze algoritmul eficient fără să depășească complexitatea liniară în prelucrare, cum este în cazul *Shortest Job First* care necesită o sortare în prealabil de complexitate $O(n \log n)$. Pentru procesele gata de execuție se calculează *burst time*-ul maxim și se setează valoarea cuantei la 80% din acesta (Algortim 1 liniile 3-4). Următorul pas constă în execuția tuturor proceselor cu timpul de execuție mai mic decât cuanta, iar după epuizarea acestora cuanta este setată la maximul timpilor de execuție al proceselor rămase. Cuanta se recalculează de fiecare dată când intră în sistem procese noi (Algortim 1 linia 2). La linia 5 se apelează procedura de planificare a proceselor. Operația *while* este o abstractizare a modului în care simulatorul controlează execuția, explicat în secțiunea 4.1, ce o voi folosi și în explicarea algoritmilor următori.

O problemă a algoritmului *Round Robin* este aceea că în unele cazuri, după ce un proces și-a executat cuanta de timp el ar trebui să aștepte după toate celelalte procese din coadă, deși timpul lui rămas de execuție este mic (câteva unități de timp), după cum am putut vedea în Secțiunea 2.0.1. O caracteristică a algoritmului propus este faptul că

evită această problemă.

Algoritm 1: Efficient Dynamic Round Robin

Data: $readyQueue$:= coada proceselor gata de execuție
$currentTime$:= timpul curent al procesorului

```
1 while running do
2     if ∃P ∈ readyQueue s.t P.arrivalTime = currentTime then
3         BT_max ← max({P.burstTime | P ∈ readyQueue})
4         quant ← 8/10 × BT_max
5     scheduleProcesses(readyQueue, quant)
6 end
```

3.1.2 Mean Threshold Shortest Job Round Robin

În lucrarea *Mean Threshold Shortest Job Round Robin CPU Scheduling Algorithm* [14], autorii își propun să rezolve problema algoritmului *Round Robin* menționată mai sus, dar și cea a algoritmului *Shortest Job First*, în care procesele cu timp de execuție mare trebuie să aștepte după toate procesele scurte, ducând la un timp de răspuns nesatisfăcător. Pentru a realiza acest lucru, autorii prezintă o combinație a celor doi algoritmi după cum urmează, întâi se calculează media timpilor de execuție a proceselor din coadă (Algoritm 2 linia 6), procesele cu timp mai mic decât media vor vi plasate într-o nouă coada pe care se aplică algoritmul *Shortest Job First*, iar restul se vor plasa într-o coadă care se va executa folosind *Round Robin* (Algoritm 2 liniile 7-8), cu cuanta de timp aleasă de utilizator. În lucrare nu este tratat cazul în care procesele ajung la timpi diferiți așa că am decis să recalculez media și cele două cozi de fiecare dată când intră un proces nou (Algoritm 2 linia 2). Cum procedeul se repetă de mai multe ori, inițial se reunesc mulțimile *sjfQueue* și *rrQueue* cu mulțimea *readyQueue*, primele două resetându-se la mulțimea vidă (Algoritm 2 liniile 3-5), iar după ce s-a făcut împărțirea, la linia 9, se resetează *readyQueue*. În liniile 10-14 se indică faptul că mulțimea proceselor *sjfQueue* are prioritate și procesele respective se execută primele.

Metoda prezentată dispune de beneficiile celor doi algoritmi clasici, procesele scurte sunt executate rapid, ducând la o scădere a timpului mediu de așteptare, iar procesele lungi primesc timp pe CPU în mod echitabil.

Algoritm 2: Mean Threshold Shortest Job Round Robin

Data: $readyQueue$:= coada proceselor gata de execuție
$currentTime$:= timpul curent al procesorului
$sjfQueue$:= coada proceselor executate cu Shortest Job First
$rrQueue$:= coada proceselor executate cu Round Robin
$quant$:= cuanta de timp setată de utilizator

1 **while** *running* **do**
2 **if** $\exists P \in readyQueue$ s.t $P.arrivalTime = currentTime$ **then**
3 $readyQueue \leftarrow sjfQueue \bigcup rrQueue$
4 $sjfQueue \leftarrow \emptyset$
5 $rrQueue \leftarrow \emptyset$
6 $threshold \leftarrow \frac{\sum_{P \in readyQueue} P.burstTime}{readyQueue.size()}$
7 $sjfQueue \leftarrow \{P \mid P.burstTime \leq threshold\}$
8 $rrQueue \leftarrow \{P \mid P.burstTime > threshold\}$
9 $readyQueue \leftarrow \emptyset$
10 **if** $sjfQueue \neq \emptyset$ **then**
11 $scheduleProcesses(sjfQueue)$
12 **else**
13 $scheduleProcesses(readyQueue, quant)$
14 **end**
15 **end**

3.1.3 Min-Max Round Robin

În lucrarea *An Effective Round Robin Algorithm using Min-Max Dispersion Measure* [12] autorii propun altă soluție problemei ridicate de autorii articolului *Efficient Dynamic Round Robin*. Algoritmul de bază este *Round Robin*, dar de fiecare dată când un proces nou intră în sistem (Algoritm 3 linia 2) cuanta de timp este recalculată drept diferența dintre timpul maxim rămas de execuție al proceselor din coada așteptare și timpul minim, iar dacă valoarea obținută este mai mică decât un prag setat de utilizator atunci cuanta devine acest prag (Algoritm 3 liniile 3-5). La linia 6 procesele se planifică conform algoritmului *Round Robin* cu cuanta calculată anterior.

Algoritm 3: Min-Max Round Robin

Data: $readyQueue$:= coada proceselor gata de execuție
$currentTime$:= timpul curent al procesorului
$threshold$:= prag introdus de utilizator

```
1 while running do
2     if ∃P ∈ readyQueue s.t P.arrivalTime = currentTime then
3         quant ← max({P.burstTime | P ∈
            readyQueue}) − min({P.burstTime | P ∈ readyQueue})
4         if quant < threshold then
5             quant ← threshold
6     scheduleProcesses(readyQueue, quant)
7 end
```

3.1.4 Best Time Quantum Round Robin

În lucrarea *Best time quantum round robin CPU scheduling algorithm* [11] autorii propun un algoritm ce funcționează în mod similar cu cel prezentat anterior dar cuanta de timp este calculată drept media între mediana timpilor de execuție a proceselor din coada de așteptare și media acestora (Algoritm 4 liniile 3-5). Linia 2 indică faptul că procedura este repetată de fiecare dată când un proces intră în sistem, iar la linia 6 planificarea este realizată folosind algoritmul *Round Robin*.

Algoritm 4: Best Time Quantum Round Robin

Data: $readyQueue$:= coada proceselor gata de execuție
$currentTime$:= timpul curent al procesorului
$threshold$:= prag introdus de utilizator

```
1 while running do
2     if ∃P ∈ readyQueue s.t P.arrivalTime = currentTime then
3         mean ← sum({P.burstTime|P∈readyQueue}) / readyQueue.size()
4         median ← readyQueue[⌊readyQueue.size() / 2⌋]
5         quant ← (mean+median) / 2
6     scheduleProcesses(readyQueue, quant)
7 end
```

3.2 Timp Real Multi-procesor

3.2.1 Least Slack Time Rate first

În lucrarea *Least Slack Time Rate first: New Scheduling Algorithm for Multi-Processor Environment* [10] autorii prezintă o metodă de a asigna priorități în mod dinamic care nu

produce momente de timp la care procesorul să fie inactiv, slăbiciunea algoritmilor clasici precum *Earliest Deadline First*, pe cazul multiprocesor, fapt din care acesta nu poate fi optim. Autorii propun următoarea soluție, la fiecare moment de timp (Algoritmul 5 linia 2) este definită rata unui proces drept raportul dintre timpul rămas de excuție și timpul rămas până la *deadline* (Algoritmul 5 liniile 3-4). Procesele se ordonează descrescător după rată și se execută primele P pentru o unitate de timp (Algoritmul 5 liniile 6-7). În plus, înainte de începutul execuției este definit timpul maxim de execuție MOT ca diferența minimă între D_i și C_i pentru procesele din coada de execuție (Algoritmul 5 linia 1). Un proces poate rula maxim MOT unități de timp consecutiv. Modul de calcul al priorităților, cât și limitarea execuției induc o planificare fără să existe vreun procesor inactiv. Un alt argument pentru care algoritmii clasici prezentați anterior nu sunt optimi pe cazul multiprocesor este acela că ei se folosesc de caracteristici care se modifică pentru toate procesele, indiferent dacă sunt executate sau nu, pe când algoritmul *LSTR* se folosește de timpul rămas de execuție, care se modifică doar pentru pentru procesele executate, de aceea la următoarea unitate de timp, algoritmul își dă seama de *task*-urile urgente.

Algoritm 5: Least Slack Time Rate first

Data: $readyQueue$:= coada proceselor gata de execuție
$\mathcal{H}$:= hiperperioada proceselor
P := numărul de procesoare

1 $MOT \leftarrow min(\{P.relativeDeadline - P.burstTime \mid P \in readyQueue\})$
2 **for** $t = 0;\ t < \mathcal{H};\ t = t + 1$ **do**
3 **for** $\tau_i \in readyQueue$ **do**
4 $P_i = \frac{C_i^{remaining}}{D_i^{absolute} - t}$
5 **end**
6 $readyQueue.sort(\lambda\tau_i.P_i, reversed)$
7 rulează primele P procese care nu au depășit MOT pentru o unitate de timp
8 **end**

3.2.2 Linear Task Scheduling

În lucrarea *LTS: Linear Task Scheduling on Multiprocessor through Equation of the Line*[7] autorii modelează prioritatra unui *task* ca fiind 0 la intrarea în sistrm și crescând liniar la 1 cand momentul de timp coincide cu *deadline*-ul *task*-ului. Aceștia au observat că un criteriu mai rațional aste acela că un *task* să aibă prioritate inițială egală cu utilizarea lui c_i, dar au observat următoarea problemă, un proces cu o utilizare mică ajunge să fie executat prea târziu. Pentru a depăși această problemă autorii au propus să forțeze *task*-urile să execute câte o unitate de timp uniform în timpul dintre perioade, mai precis, să execute o unitate la fiecare $\frac{D_i}{C_i}$ unități de timp. În loc ca prioritatea să fie 1 la timpul D_i, să ajungă 1 la timpul $\frac{D_i}{C_i}$, după care pornește iar de la $\frac{C_i}{D_i}$. Astfel în figura 3.1 se

poate observa cum evoluează prioritatea. Pentru un moment aleator de timp t, prioritatea procesului τ_i este $P_i(t) = \frac{C_i}{D_I} + \frac{1-\frac{C_i}{D_I}}{\frac{D_i}{C_I}} \times (t \mod D_i)$, funcția este dată de ecuația liniei din Figura 3. Pentru a mări șansa celorlalte *task*-uri, care nu au fost executate la un moment de timp, la fiecare execuție a unui proces, prioritatea este penalizată cu $1 - \frac{C_i}{D_I}$, astfel ajungându-se la formula din Algoritmul 6 linia 10. *Task*-urile urgente sunt depistate print faptul ca au *laxity* egal 0 și de aceea trebuie executate obligatoriu, fiindu-le asignate prioritatea $+\infty$ (Algoritmul 6 liniile 7-8). Se sortează coada descrescător după prioritate, se execută primele P procese și se resetează penalizarea cumulată dacă *task*-urile selectate își termină execuția (Algoritmul 6 liniile 13-20), procedură repetată la fiecare moment de timp (linia 5). În liniile 1-3 se inițializează valoarea cu care fiecare proces este penalizat, respectiv penalizarea cumulată.

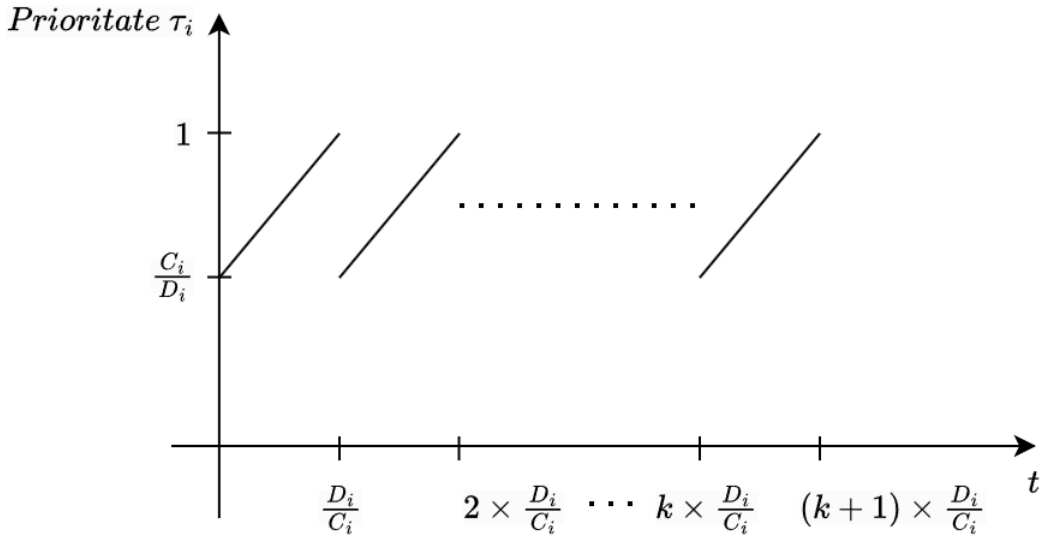

Figura 3.1: Evoluția priorității unui *task* τ_i

3.2.3 DynAmic Real-time Task Scheduling

În lucrarea *DARTS: DynAmic Real-time Task Scheduling*[8] autorii introduc noțiunea de *Dynamic Utilization* drept raportul între timpul rămas de execuție și timpul până la următorul *deadline*. Aceștia îmbunătățesc formula prin luarea în calcul a *laxity*-ului unui proces introducând noțiunea de *Dynamic Score* care este raportul dintre *Dynamic Utilization* și *laxity*. Astfel formula de acordare a priorităților devine cea din Algoritmul 7 linia 3.

$$DS_{\tau_i}(t) = \frac{C_i^{remaining}}{(D_i^{absolute} - t) * (D_i^{absolute} - t - C_i^{remaining})} \tag{3.1}$$

Analog algoritmilor anterior, la fiecare moment de timp se calculeaza *DS* pentru fiecare proces (Algoritmul 7 liniile 1-4), procesele se sortează descrescător dupa *DS* și se rulează primele P pentru o unitate de timp (liniile 5-6).

Algoritm 6: Linear Task Scheduling

Data: $readyQueue$:= coada proceselor gata de execuție

$\mathcal{H}$:= $hiperperioadaproceselor$

P := numărul de procesoare

for $\tau_i \in readyQueue$ **do**
  $Res_i = 1 - \frac{C_i}{D_i}$
  $totalRes_i = 0$
end
for $t = 0;\ t < \mathcal{H};\ t = t + 1$ **do**
  for $\tau_i \in readyQueue$ **do**
    if $D_i^{abosulte} - t - C_i^{remaining} == 0$ **then**
      $P_i = +\infty$
    else
      $P_i = \frac{C_i}{D_i} + \frac{1-\frac{C_i}{D_i}}{\frac{D_i}{C_i}} * (tmodD_i) - totalRes_I$
    end
  end
  $readyQueue.sort(\lambda\tau_i.P_i, reversed)$
  rulează primele P procese pentru o unitate de timp
  for $\tau_i \in proceseSelectate$ **do**
    if $C_i^{remaining} == 0$ **then**
      $totalRes_i \leftarrow 0$
    else
      $totalRes_i \leftarrow totalRes_i + Res_i$
    end
  end
end

Algoritm 7: DynAmic Real-time Task Scheduling

Data: $readyQueue$:= coada proceselor gata de execuție

$\mathcal{H}$:= hiperperioada proceselor

P := numărul de procesoare

for $t = 0;\ t < \mathcal{H};\ t = t + 1$ **do**
  for $\tau_i \in readyQueue$ **do**
    $P_i = \frac{C_i^{remaining}}{(D_i^{abosulte}-t)*(D_i^{abosulte}-t-C_i^{remaining})})$
  end
  $readyQueue.sort(\lambda\tau_i.P_i, reversed)$
  rulează primele P procese pentru o unitate de timp
end

3.3 Multi-procesor

3.3.1 Fair-Share Scheduling

În lucrarea *Providing fair-share scheduling on multicore computing systems via progress balancing* [9] autorii propun o metodă de realizare a operației de *load balance* din cadrul algoritmului *Completly Fair Scheduler* introdusă sub numele de *progress balancing* care presupune gruparea proceselor cu *load* mai mare față de cele cu *load* mai mic și executarea lor pe procesoare diferite. Astfel primul grup avansează mai greu decât al doilea și se obține egalitate mai bună. Autorii prezintă o demonstrație matematică a faptului că pe parcursul rulării diferența maximă între timpii virtuali de rulare este mărginită superior de o constantă, fapt ce nu este garantat în cazul *Completly Fair Scheduler* nativ. Per procesor este folosit același algoritim menționat în secțiunea 2.0.2.

Capitolul 4

Simulator

4.1 Arhitectura simulatorului

Vom considera cazul general în care vrem să simulăm un algoritm oarecare folosind n procesoare. Fiecare procesor este o instanță a unui simulator discret de evenimentele care rulează independent față de celelalte pe propriul fir de execuție, fiind coordonate de un simulator global. Înainte de a continua trebuie definite tipurile de obiecte cu care lucrăm.

Am definit clasa **Event** care conține timpul la care se petrece, tipul de eveniment și procesul la care se referă, dacă este cazul. Tipurile de evenimente sunt următoarele

ARRIVAL Folosit pentru a semnala că un proces a intrat în sistem.

CPUBURSTCOMPLETE Procesul și-a terminat tot *burst time*-ul.

TIMEREXPIRED Procesul și-a terminat cuanta de timp alocată.

LOADBALANCE Semnal pentru a realiza operația de *load balance*.

TICK Folosit pentru a forța procesoarele să ajungă la un moment de timp.

REALTIME Pentru a diferenția între procesele normale și periodice.

PREEMT Pentru a elimina evenimentele ulterioare din coadă ale unui anumit proces.

Clasa **Core** este abstractizarea unui procesor care simulează rularea și întoarce rezultatele. Clasa **OS** este responsabilă de parcurgerea evenimentelor inițiale, anume lista de procese care ajung în sistem, de atribuirea proceselor fiecărui **Core**, de efectuarea operației de *load balancing*, dacă este cazul, și de avansarea execuției până la următorul eveniment.

O variantă preliminară a simulatorului a funcționat în felul următor, fie $t_1, t_2, ...t_n$ timpii la care intră procese în sistem, cu $t_1 \leq t_2 \leq ... \leq t_n$, **OS**-ul asignează procesele ajunge la timpul t_1 **Core**-urilor disponibile (Algoritm 8 liniile 2-3), actualizează timpul

global și anunță că acesta s-a schimbat (Algoritm 8 liniile 4-6). În tot acest timp, fiecare **Core** a așteptat prin intermediul unei variabile de condiție, element de sincronizare care blochează firul de execuție curent până când altul modifică o variabilă, ca timpul global să avanseze (Algoritm 9 linia 7), după care continuă cu simularea execuției, oprindu-se la evenimentele ce depășesc timpul global (Algoritm 9 liniile 9-11). Când fiecare *Core* a ajuns la momentul în care cel mai devreme eveniment are loc după timpul global, acestea trebuie să intre în starea de așteptare până când are loc următoarea actualizare. Am folosit barierele *barieră1* și *barieră2* pentru a crea o regiune în care are loc sincronizarea firelor de execuție și **OS**-ul poate seta variabila *osTimeUpdated* la valoarea inițială pentru a forța **Core**-urile să aștepte (Algoritm 8 liniile 7-9, respectiv Algoritm 9 liniile 14-16). Adtfel operația din algoritmul 8 linia 8 este executate de către **OS** fără ca vreun **Core** să verifice din nou variabila și să o găsească *True* (Algoritm 9 linia 8).

Algoritm 8: OS

```
   Data: events =
         coada de evenimente globală
   osTime =
   timpul global al sistemului
   osTimeUpdated =
   variabilă de condiție partajată
   bariera1, bariera2 =
   bariere reutilizabile partajate
 1 while running do
 2 |   futureTime ←
   |    events.top().time()
 3 |   processEventsAtTime(futureTime)
   |
 4 |   osTime ← futureTime
 5 |   osTimeUpdated ← True
 6 |   cv.notify_all()
 7 |   barier1.arrive_and_wait()
 8 |   osTimeUpdated ← False
 9 |   barier2.arrive_and_wait()
10 end
```

Algoritm 9: Core

```
   Data: events =
         coada de evenimente locală
 1
 2
 3
 4
 5
 6
 7 while running do
 8 |   cv.wait(osTimeUpdated ==
   |    True)
 9 |   while events.top() ≤ osTime do
10 |   |   processEvent(events.top())
11 |   end
12 |
13 |
14 |   barier1.arrive_and_wait()
15 |
16 |   barier2.arrive_and_wait()
17 end
```

O diagramă a execuției se poate observa în figura următoare

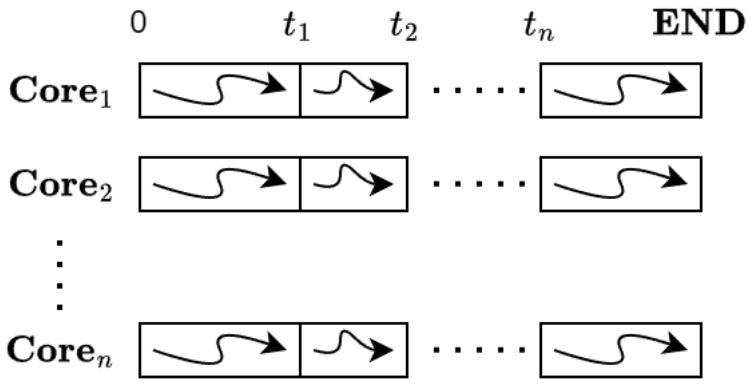

Figura 4.1: O vizualizare a modului în care se execută simularea.

Problema cu acest mod de funcționare este că dacă între două momente de timpi consecutivi t_i, t_{i+1}, un **Core** devine *idle*, stare în care nu are niciun proces de executat, nu are nicio modalitate de a anunța acest lucru. Pentru a rezolva problema am folosit următorul protocol, la momentul de timp t_i, după ce **Core**-urile au procesat evenimentele și le-au generat pe următoarele, are loc o sincronizare între acestea în care este selectat evenimentul cu timpul minim și se adaugă un eveniment la acest timp de tipul **TICK** atât în coada **OS**-ului cât și în cozile tuturor **Core**-urilor. Astfel, dacă vreunul devine *idle* la acest timp se poate efectua operația de *load balancing*.

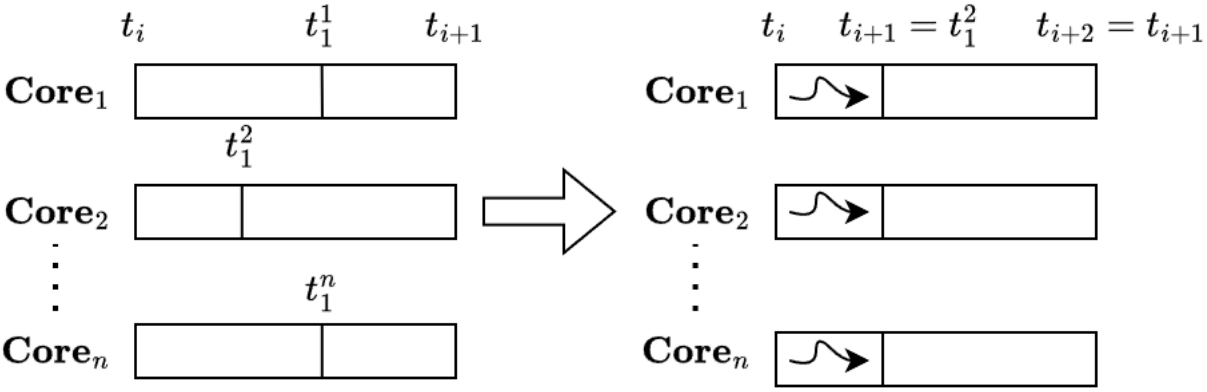

Figura 4.2: O vizualizare a noului mod în care se execută simularea.

Pentru a realiza sincronizarea, algoritmul 9 trebuie modificat în felul următor, fie t timpul minim la care a ajuns un eveniment în coadă, se procesează evenimentele care ajuns la acest timp și se generează următoarele (Algoritm 10 liniile 3-7) , fiecare **Core** anunță timpul următor la care are evenimente de procesat și se folosesc două bariere partajate doar de **Core**-uri care au scopul de a crea o zonă în care unul singur calculează timpul minim din cele anunțate și adaugă în coada **OS**-ului un eveniment de tipul **TICK**. În plus, fiecare **Core** va adăuga în coada sa un eveniment de același tip la timpul minim calculat.

Pentru a asigura faptul că un singur fir de execuție accesează coada evenimentelor globale am folosit un *mutex* împreună cu un contor ce se incrementează (modulo numărul de procesoare) de fiecare dată când un *thread* intră în zona critică și îi oferă dreptul de acces doar primului venit, când contorul este 0 (Algoritm 10 liniile 8-17) .

Algoritm 10: Core Modificat

Data: $events$ = coada de evenimente
$bariera1Core, bariera2Core$ = bariere partajate doar între core-uri
m = mutex partajat doar între core-uri
$minimumTime$ =
vector în care fiecare core stochează timpul minim al următorului eveniment
$numCores, coreID$ = numărul de core-uri, respectiv identificatorul celui curent

```
1  while running do
2      cv.wait(osTimeUpdated == True)
3      time ← events.top().getTime()
4      while events.top().getTime() == time do
5          processEvent(events.top())
6          events.pop()
7      end
8      minimumTime[coreID] = events.top().getTime()
9      bariera1Core.arrive_and_wait()
10     minEvent ← min({minimumTime[i] | 0 ≤ i ≤ numCores})
11     events.push(Event(TICK, minEvent))
12     m.lock()
13     if cnt = 0 then
14         eventsOS.push(Event(TICK, minEvent))
15     cnt = (cnt + 1)  mod numCores
16     m.unlock()
17     bariera2Core.arrive_and_wait()
18     bariera1.arrive_and_wait()
19     bariera2.arrive_and_wait()
20 end
```

Cu tot cu modificări, modul de operare al algoritmului 10 poate fi descris astfel, fiecare **Core** așteaptă ca timpul global să fie actualizat (linia 2), după care, în liniile 3-7, se procesează evenimentele. La linia 8 se anunță timpul minim la care este următorul eveniment și se așteaptă ca toate celelalte **Core**-uri să fi ajuns în acest punct (linia 9). La liniile 10-11 fiecare **Core** își adaugă în coadă evenimentul minim care o să forțeze sincronizarea acestora, iar prin intermediul mutex-ului blocat la linia 12 și deblocat la linia 16 se creează zona în care primul fir de execuție adaugă în coada globală evenimentul minim (liniile 13-15). Bariera de la linia 17 asigură că toate **Core**-urile trec în același timp

de această procedură. Barierele de la liniile 18 și 19 au același scop menționat anterior.

În procesarea evenimentelor se folosește o coadă de priorități sortată crescător după timp. Ordinea procesării este următoarea

PREEMT Primul procesat, există doar în cazul **Core**-urilor și șterge evenimentele aferente procesului referențiat

ARRIVAL **OS**-ul distribuie mai departe evenimentul **Core**-urilor, iar acestea din urmă adaugă în coada *readyQueue* procesul

CPUBURSTCOMPLETE **Core**-ul adaugă la metrici valorile timpilor de așteptare și de *turnaround* al procesului.

TIMEREXPIRED Procesul și-a terminat cuanta de timp alocată.

LOADBALANCE Folosit de **OS** care începe procedura de redistribuire a muncii.

TICK nu este tratat, având rol auxiliar.

REALTIME **OS**-ul adaugă în coada de evenimente următoarea intrare în sistem a procesului periodic.

După faza de procesare, în cazul **Core**-urilor se apelează funcția *schedule* de planificare explicată în secțiunea 4.2.

Pentru a obține imaginile ce conțin graficele, cât și pentru a apela funcția *python* descrisă în secțiunea următoare, la începutul pornirii aplicației se deschide un canal de comunicare bidirecțional (*pipe*) și se pornește un fir de execuție cu un program *python* care se conectează la acest *pipe* și așteaptă pentru o cerere. În momentul în care este nevoie apelarea funcției sau crearea unui grafic simulatorul trimite toate datele necesare prin intermediul canalului și așteaptă în mod sincron răspunsul cu rezultatul, sau cu locația imaginii generate.

Procesele sunt reprezentate de clasa **Process** care reține elemente precum identificatorul, timpul de execuție, prioritatea și identificatorul distribuției din care a fost generat. În cazul multi-procesor este reținut și *virtual runtime*-ul, iar în timp real, perioada.

4.1.1 Generarea datelor de intrare

Uni-procesor și multi-procesor

În acest caz generarea datelor este trivială, utilizatorul introduce parametrii a maximum 3 distribuții normale împreună cu numărul de procese ce trebuie generate din fiecare distribuție. Prioritatea este generată dintr-o distribuție uniformă $\mathcal{U}(0,\ 40)$, iar timpul de intrare în sistem este generat tot dintr-o distribuție uniformă $\mathcal{U}(0,\ T_{max})$, unde T_{max} este introdus tot de utilizator.

Timp real

Așa cum am văzut în secțiunea 2.0.3 generarea unei mulțimi de procese în timp real nu este un lucru trivial. Voi prezenta și folosi de metodele din lucrarea *Techniques For The Synthesis Of Multiprocessor Tasksets* [5].

Din ecuațiile 2.1 și 2.2 rezultă faptul că avem nevoie de un set de date care să satisfacă $\frac{C_i}{D_i} \leq 1 \quad \forall\ 1 \leq i \leq n$ și $\sum_{i=1}^{n} \frac{C_i}{D_i} \leq m$. Pentru a realiza acest lucru voi folosi ca date de intrare utilizarea procesorului U (obligatoriu $U \leq m$) și numărul de procese care urmează să fie generat. Se vor genera aleator perioadele proceselor și voi folosi un algoritm care are ca date de intrare numărul de procese și utilizarea țintă și obțin ca date de ieșire un vector de n elemente u format din elementele u_i (utilizarea fiecărui proces) de sumă U satisfăcând în același timp condiția 2.1. În final, C_i este egal cu $D_i \times u_i$.

Perioadele proceselor sunt generate aleator dintr-o distribuție normală în felul următor:

$$r_i \sim U(T_{\min}, T_{\max} + 1) \tag{4.1}$$

În cadrul acestui simulator $T_{\min}$ este setat la 10 și $T_{\max}$ este setat la 100. Am văzut în secțiunea 2.0.3 că este suficient să observăm comportamentul algoritmilor doar în cadrul hiper-perioadei care din cauza faptului că folosește cel mai mic multiplu comun aceasta poate fi foarte mare așa că voi folosi metoda prezentată în [22] pentru a o mărgini superior.

Am definit limita superioară egală cu $360 = 2^3 \times 3^2 \times 5^1$ după care am format o listă sortată crescător cu toate numerele cu factori primi din mulțimea $\{2, 3, 5\}$ și exponenții maximi în ordine $\{3, 2, 1\}$. Pentru a ajusta perioada T caut binar cel mai mic număr mai mare sau egal cu T din lista generată. Astfel se obține o listă de numere pentru care cel mai mic multiplu comun este mărginit superior de 360.

Pentru a genera vectorul de lungime n și suma U voi folosi algoritmul *randfixedsum* [1] care generează uniform puncte în locul geometric dat de acești vectori.

În primul rând voi prezenta cum se poate genera un vector de lungime n cu suma componentelor 1 în timp $\mathcal{O}(n)$ folosind algoritmul *UUniFast* [2]. Baza algoritmului constă în faptul că dacă considerăm variabilele aleatoare $X_i \sim \mathcal{U}(0, 1)$ pentru $1 \leq i \leq n$ atunci variabila aleatoare $\mathcal{X} = \sum_{i=1}^{n} X_i$ va avea

$$P(\mathcal{X} = u) = u^{n-1}, 0 < u \leq 1 \tag{4.2}$$

Se inițializează suma țintă cu 1 și vectorul rezultat cu vectorul nul O_n (Algoritm 11 liniile 1-2). La fiecare pas (Algoritm 11 linia 3) se generează suma a i variabile uniforme independente și se calculează diferența între suma țintă precedentă și valoarea generată, astfel, prima dată se generează suma a $n-1$ termeni și se setează primul element ca fiind diferența între 1 și valoarea generată (Algoritm 11 liniile 4-5), după care se actualizează suma țintă (Algoritm 11 linia 6) și se repetă procesul pentru $n \leftarrow n - 1$. Ultima valoare

Algoritm 11: UUniFast

Data: n = dimensiunea vectorului de ieșire
Result: $x \in \mathbf{R}^n$ astfel încât $||x||_1 = 1$

1 $sumU \leftarrow 1$;
2 $x \leftarrow O_n$;
3 **for** $i = 0;\ i < n-1;\ i = i+1$ **do**
4 $\quad nextSumU \leftarrow sumU \times rand^{\frac{1}{n-i-1}}$
5 $\quad x[i] \leftarrow sumU - nextSumU$
6 $\quad sumU \leftarrow nextSumU$
7 **end**
8 $x[n-1] \leftarrow sumU$

a vectorului este ce a rămas în $sumU$(Algoritm 11 linia 8). Există și alți algoritmi, dar acesta generează în mod uniform date.

Următorul pas în înțelegerea algoritmului *randfixedsum* constă în stabilirea domeniului în care căutam vectorii. După cum se poate vedea în figura 4.4 un vector $x \in \mathbf{R}^n$ de sumă s se situează într-un hiper-plan de dimensiune $n-1$ definit de n puncte, numit simplex. Dificultate pe care o întâlnim când lucrăm cu $s > 1$ este dată de condiția 2.4 care restrânge locul geometric al acestor vectori (zonele cu roșu închis).

Dacă nu am fi avut condiția 2.4 atunci $x \in \mathbf{R}^n$ cu $||x||_1 = s$ ar fi fost de forma $\sum_{i=1}^{n} a_i \times p_i$, unde $p_i = (0, 0, ...s, ..., 0)$, s pe poziția i și 0 în rest, iar $\sum_{i=1}^{n} a_i = 1$, cu $0 \leq a_i \leq 1$. Vectorul a poate fi generat folosind *UUniFast*, astfel obținând x.

Ideea din spatele *randfixedsum* constă în împărțirea locului geometric în mai multe simplexuri (cum se poate observa în figura 4.4) și alegerea unuia în mode aleator în funcție de volumul său (arie în cazul 3 dimensional). Orice simplex din planul R^n este definit de $n+1$ vârfuri, iar orice punct din interiorul său este o combinație liniară a acestora cu coeficienți pozitivi de sumă 1, ce pot fi generați cu algoritmul *UUniFast*.

În practică, deoarece vectorul u generat este de tipul *double*, iar în cadrul simulatorului lucrez cu perioade și timpi de execuție întregi au apărut două probleme, calculul $T_i \leftarrow u_i \times D_i$ devine defapt $T_i \leftarrow \lfloor u_i \times D_i \rfloor$, astfel dacă recalculăm utilizarea mulțimii dă mai puțin decât U, utilizarea țintă inițială. De aceea am apelat algoritmul cu utilizarea $U + \epsilon$ unde ϵ a ajuns să fie 0.4 atunci când utilizarea țintă era inițial 4. După acest pas, cea de a doua problemă a fost că la recalcularea utilizării obțineam un rezultat în intervalul $(U - \gamma, U + \gamma)$, cu $0 < \gamma < 0.1$ și doar o fracție mică din mulțime avea fix U, de aceea trebuie generat un număr mare de mulțimi.

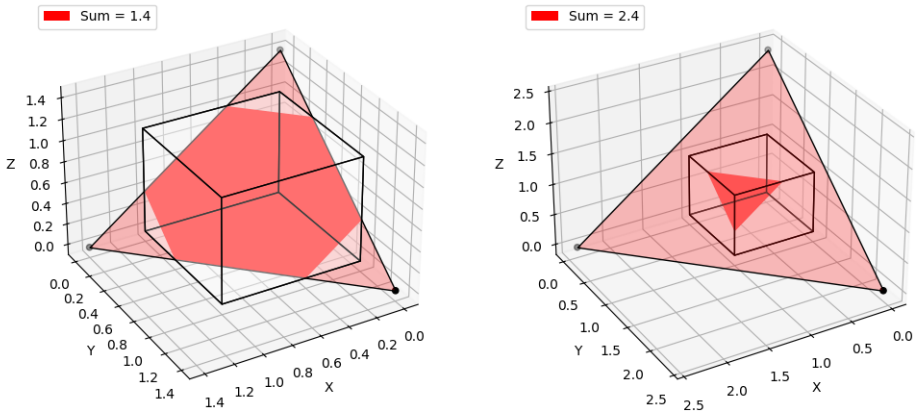

Figura 4.3: Intersecția dintre cubul unitate și locul geometric al vectorilor $x \in \mathbf{R}^3$ cu $||x||_1 = 1.4$, respectiv, vectorii cu $||x||_1 = 2.4$

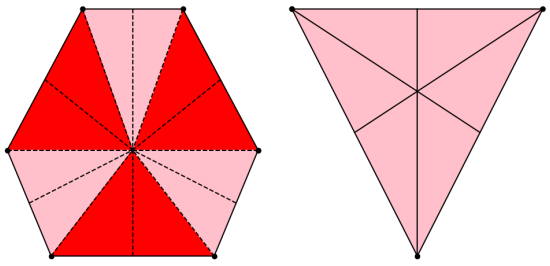

Figura 4.4: Tipurile de simplexuri pentru cazurile din Figura 4.3

4.2 Implementarea unui algoritm

Pentru a implementa un algoritm pun la dispoziție interfața *SchedulingAlgorithm* care oferă spre implementare următoarele funcții

```
virtual std::vector<Event> processArrived(std::vector<Process> p,
↪  int time, Metrics &stats)
```

Am folosit această funcție pentru a introduce procesele intrate în sistem la timpul *time* în coada de așteptare a algorimului.

```
virtual std::vector<Event> processCPUComplete(Process p, int time,
↪  Metrics &stats)
```

Această funcție are deja implementare și este folosită pentru a adăuga la metrici datele de interes procesul p care și-a terminat execuția precum timpul de așteptare, timpul de răspuns și *turnaround*.

```
void assignProcessToCPU(Process p, Metrics &stats, int time)
```

Aceasta nu este expusă, dar este o funcție utilitară pentru a asigna un proces procesorului. În interiorul ei se ține evidența schimbărilor de context și se actualizează timpul de răspuns al procesului, dacă este prima execuție a acestuia, cât și a timpului de așteptare.

```
virtual void preemtCPU(Metrics &stats, int time)
```

Aceasta este folosită pentru a controla modul în care se evacuează procesorul. Am utilizat-o pentru a introduce procesul curent în execuție în coada de așteptare și a-i actualiza timpul rămas de execuție.

Această funcție are ca parametrii un vector de procese, la ce timp au ajuns și o referință către metricile algoritmului până la timpul curent. În implementarea algoritmilor am folosit această funcție pentru a adăuga procesele în coada *readyQueue* a algoritmului și pentru a calcula diverse variabile de interes, în special pentru algoritmii în timp real.

```
virtual std::vector<Event> schedule(int time, Metrics &stats,
↪  bool timerExpired)
```

Aceasta este funcția de bază care trebuie implementată, reprezintă locul în care se află logica din spatele algoritmului. Aceasta este folosită împreună cu cele 2 utilitarele precedente. Aceasta trebuie să întoarcă evenimente de tipul *CPUBURSTCOMPLETE* pentru a semnala când procesul curent își termină execuția, *TIMEREXPIRED* pentru a semnala când se termină cuanta și *PREEMT*, pentru a șterge toate evenimentele generate de procesul care trebuie să evacueze procesorul.

```
virtual int loadBalance(int time)
```

Această funcție este folosită pentru a implementa politica de *Load Balance* a algoritmilor multi-procesor.

```
virtual int assignCPU(Process p)
```

Această funcție este folosită pentru a implementa politica de asignare a procesului p în coada de așteptare a unui procesor pentru algoritmi multi-procesor.

Capitolul 5

Experimente

5.0.1 Uni-procesor

Evaluarea acestor tip de algoritmi constă în generarea unui set de procese ce provin din mai multe distribuții normale și observarea modului cum algoritmii tratează fiecare distribuție. Pentru o intuiție asupra rezultatelor, fiecare proces intră în sistem la momentul de timp 0.

Studiu caz 1

În acest studiu vom observa comportamentul algoritmilor când se generează date aleator din două distribuții de medii $\mu_1 = 3$ și $\mu_2 = 25$, cu $\sigma_1 = \sigma_2 = 2$, iar numărul proceselor generate din prima distribuție este dublu față de cele din a doua, 200 respectiv 100.

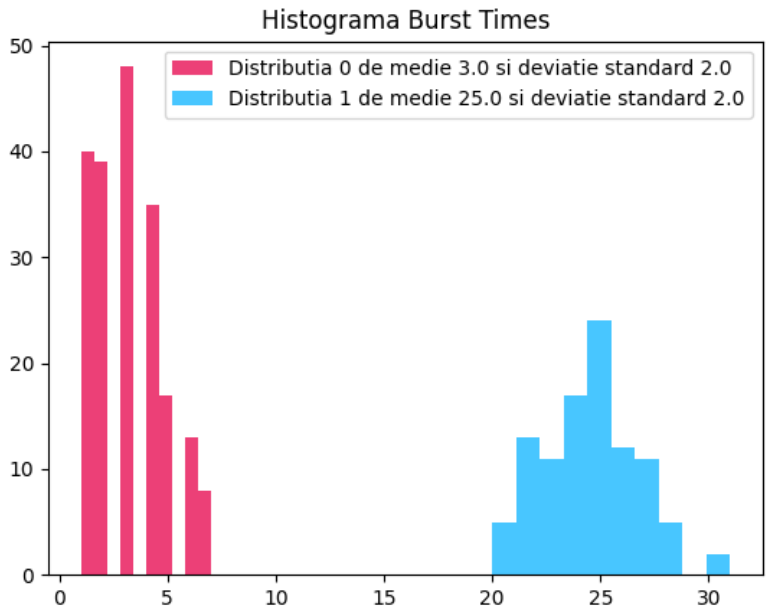

Figura 5.1: Histograma timpilor de execuție generați pentru primul studiu de caz.

Rezultatele algoritmilor se poate observa în figura următoare

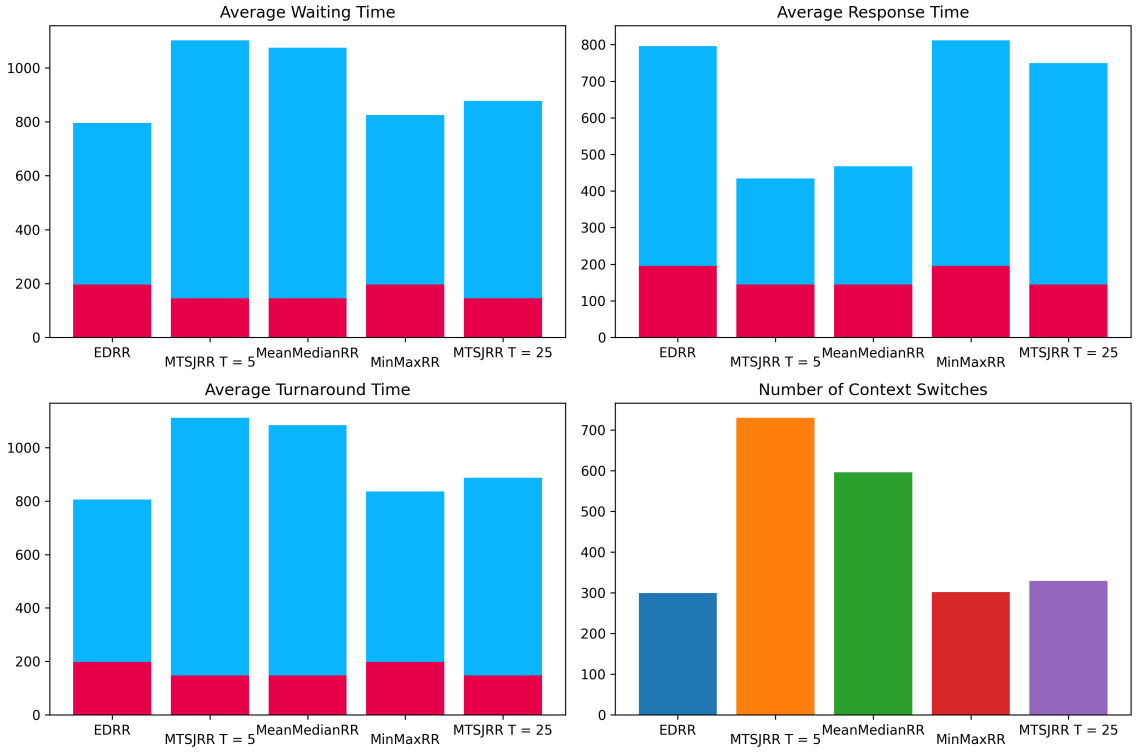

Figura 5.2: Performanțele algoritmilor pentru studiul de caz 1.

În cadrul tuturor algoritmilor, timpii de așteptare sunt mult mai ridicați pentru procesele din a doua distribuție, iar per total *Efficient Dynamic Round Robin* se dovedește a fi cel mai bun, având ca slăbiciune timpul mare de răspuns. Folosind *Mean Threshold Shortest Job Round Robin* se obține timpul de răspuns mediu cel mai bun și destul de echilibrat între cele două distribuții, dar cu un număr de schimbări de context ridicat. După cum era de așteptat *turnaround time*-ul este strâns corelat cu timpul de așteptare.

Studiu caz 2

În acest caz vom urmării performanțele în cazul în care distribuțiile se intersectează puțin și având număr egal de procese generate, anume 200. Astfel se aleg două distribuții, prima de medie 4 și deviație standard 3, iar a doua de medie 10 și deviație standard 3.

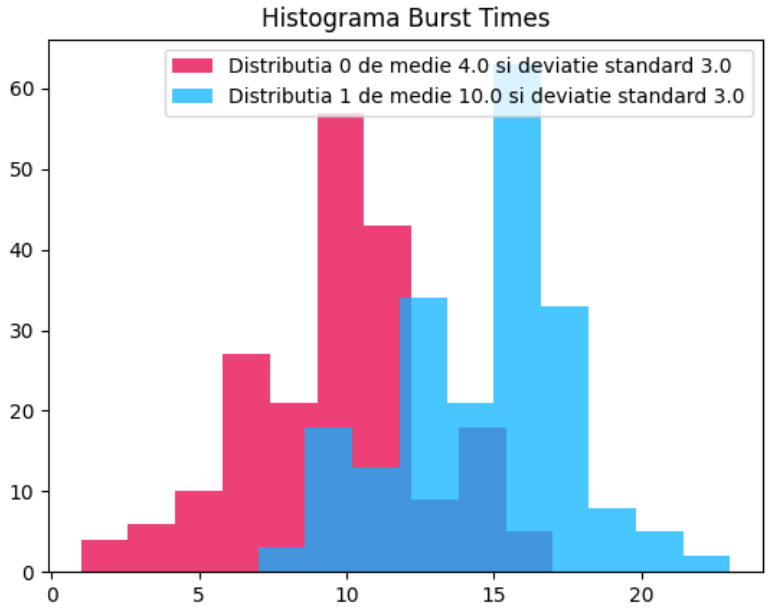

Figura 5.3: Histograma timpilor de execuție generați pentru al doilea studiu de caz.

În acest caz, în figura următoare se poate observa cum procesele generate din a doua distribuție influențează puternic metricile de evaluare, de data aceasta *Mean Threshold Shortest Job Round Robin* cu diferite praguri obținând cele mai bune rezultate.

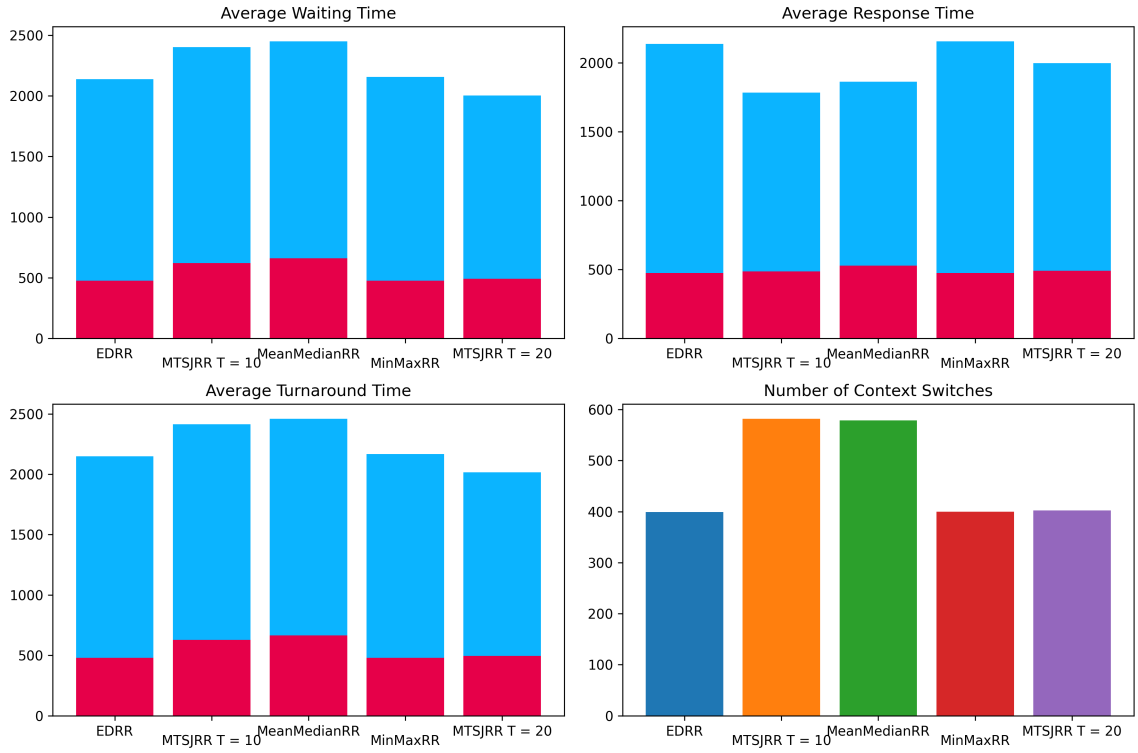

Figura 5.4: Histograma timpilor de execuție generați pentru al doilea studiu de caz.

5.0.2 Multi-procesor

În acest experiment vom genera aleator 250 de procese cu timpi de execuție proveniți dintr-o distribuție normală de medie 100 și deviație standard 20, intrate în sistem la momentul de timp 0. Diferența maximă de *virtual runtime* între două procese se află fix înainte operației de *load balancing*, de aceea am înregistrat valoarea înaintea fiecărei iterații a acesteia. Pentru comparație am implementat o variantă simplificată a algoritmului *Completly Fair Scheduler* în care procedura de migrare a proceselor constă în mutarea din coada procesorului cu *load*-ul cel mai mare în cea a procesorului cu *load*-ul cel mai mic în mod repetat. În figura următoare se poate observa cum valorile găsite prin intermediul algoritmului *Fair Share Scheduling* sunt mărginite superior pe parcursul execuției, indicând o planificare mai echitabilă a proceselor.

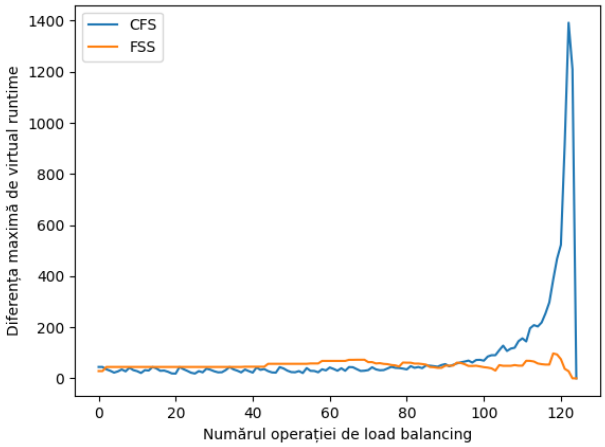

Figura 5.5: Performanțele algoritmilor *CFS* și *FSS* pentru 2 procesoare.

5.0.3 Timp real

Pentru a evalua acest tip de algoritmi voi folosi pe rând 2 și 4 procesoare. Pentru fiecare $P \in \{2, 4\}$ voi considera utilizarea țintă $U \in \{0, 0.1, ..., P - 0.1, P\}$ generând pentru fiecare un număr de mulțimi de test și voi urmări procentual câte reușește fiecare algoritm să planifice fără a rata termene limită. Având în vedere problemele de precizie enunțate în secțiunea 4.1.1, care implică faptul că de fiecare dată numărul mulțimilor de interes generate este diferit, cât și factorul stocastic al experimentelor, acestea se vor repeta de 10 ori indicând grafic și deviația standard. În plus va varia și numărul de procese din fiecare mulțime.

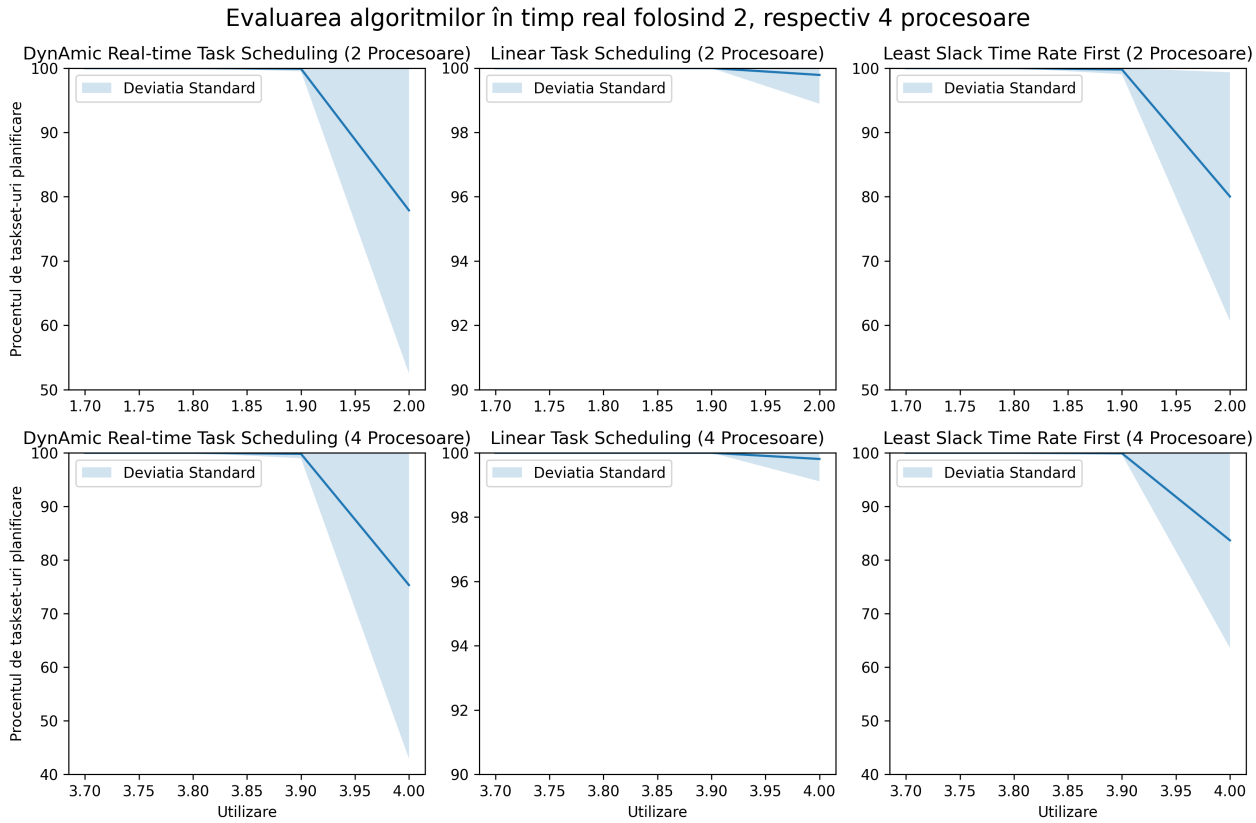

Figura 5.6: Performanțele algoritmilor pentru 2, respectiv 4 procesoare disponibile.

Performanțele algoritmilor diferă drastic în momentul în care utilizarea mulțimilor de procesoare este egală cu numărul de procese, *Linear Task Scheduling* reușind să planifice 99% din mulțimile de test, departajându-se detașat de ceilalți doi algoritmi care, în medie, planifică doar 80%. Pentru utilizare mai mică decât numărul de procese algoritmii au rezultate similare reușind să planifice toate mulțimile cu câteva excepții.

Capitolul 6

Concluzii

În această lucrare am prezentat un simulator, accesibil la adresa https://github.com/cata1212112/SchedulingSimulator, în care pot fi executați algoritmii de planificare a proceselor folosiți în cadrul unui sistem de operare. Pentru a-i demonstra eficacitatea, am ales o serie de algoritmi din literatură, acoperind toate cazurile de interes din domeniu, uni-procesor, multi-procesor și timp-real, pe care i-am prezentat și integrat în aplicație. Având în vedere scopul simulatorului, evaluarea performanțelor, am oferit un mod intuitiv de comparare a algoritmilor între ei, prin vizualizări grafice cu metricile de interes pentru fiecare caz în parte. Mai mult, utilizatorii pot folosi interfața expusă pentru a implementa orice algoritm pentru care doresc să afle comportamentul. Evaluarea se poate efectua atât prin mulțimi de test introduse de utilizator, cât și prin generarea automată a acestora. Direcțiile în care simulatorul poate fi îmbunătățit sunt multiple, pentru cazul multi-procesor se poate lua în considerare afinitatea proceselor și penalizările de timp aferente accesării memoriei cache, în cazul uni-procesor se pot lua în calcul penalizările schimbărilor de context și simularea ciclurilor de așteptare a evenimentelor de tip I/O, iar în timp real se poate extinde funcționalitatea și pentru procese asincrone, cu timpi de intrare diferiți. Deși toate cele menționate anterior reprezintă probleme reale pentru care arhitecții unui sistem de operare trebuie să găsească soluții, folosind simulatorul putem obține o serie de rezultate preliminarii aproximative ale realității.

Bibliografie

[1] Roger Stafford (2024), *Random Vectors with Fixed Sum*, https://www.mathworks.com/matlabcentral/fileexchange/9700-random-vectors-with-fixed-sum, MATLAB Central File Exchange. Accesat 20 Mai, 2024.

[2] Enrico Bini, „Measuring the Performance of Schedulability Tests", în *Real-Time Systems* 30 (Mai 2005), pp. 129–154, DOI: 10.1007/s11241-005-0507-9.

[3] Maxime Chéramy, Pierre-Emmanuel Hladik și Anne-Marie Déplanche, „SimSo: A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms", în *Proc. of the 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, WATERS, 2014.

[4] *CPU Scheduling Simulator*, https://cpu-scheduling-sim.netlify.app/, [Accessed 13-06-2024].

[5] P. Emberson, R. Stafford și R.I. Davis, „Techniques For The Synthesis Of Multiprocessor Tasksets", în *WATERS'10* (Ian. 2010).

[6] Muhammad Umar Farooq, Aamna Shakoor și Abu Bakar Siddique, „An Efficient Dynamic Round Robin algorithm for CPU scheduling", în *2017 International Conference on Communication, Computing and Digital Systems (C-CODE)*, 2017, pp. 244–248, DOI: 10.1109/C-CODE.2017.7918936.

[7] Abolfazl Ghavidel, Mohammad Hajibegloo, Abdorreza Savadi și Yasser Sedaghat, „LTS: Linear task scheduling on multiprocessor through equation of the line", în *2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS)*, 2015, pp. 1–6, DOI: 10.1109/CADS.2015.7377777.

[8] Abolfazl Ghavidel, Samaneh Sadat Mousavi Nik, Mohammad Hajibegloo și Mahmoud Naghibzadeh, „DARTS: DynAmic Real-time Task Scheduling", în *2015 7th Conference on Information and Knowledge Technology (IKT)*, 2015, pp. 1–6, DOI: 10.1109/IKT.2015.7288767.

[9] Sungju Huh și Seongsoo Hong, „Providing fair-share scheduling on multicore computing systems via progress balancing", în *Journal of Systems and Software* 125 (2017), pp. 183–196, ISSN: 0164-1212, DOI: https://doi.org/10.1016/j.jss.

2016.11.053, URL: https://www.sciencedirect.com/science/article/pii/S0164121216302412.

[10] Myunggwon Hwang, Dongjin Choi și Pankoo Kim, „Least Slack Time Rate First: New Scheduling Algorithm for Multi-Processor Environment”, în *2010 International Conference on Complex, Intelligent and Software Intensive Systems*, 2010, pp. 806–811, DOI: 10.1109/CISIS.2010.20.

[11] Dolly Khokhar și Ankur Kaushik, „Best time quantum round robin CPU scheduling algorithm”, în *International Journal of Scientific Engineering and Applied Science (IJSEAS)* 3.5 (2017), pp. 213–217.

[12] Sanjaya Kumar Panda și Sourav Kumar Bhoi, *An Effective Round Robin Algorithm using Min-Max Dispersion Measure*, 2014, arXiv: 1404.5869 [cs.OS].

[13] Abhay Kumar Parekh, „A generalized processor sharing approach to flow control in integrated services networks”, Teză de doct., Massachusetts Institute of Technology, 1992.

[14] Pragati Pathak, Prashant Kumar, Kumkum Dubey, Prince Rajpoot și Shobhit Kumar, „Mean Threshold Shortest Job Round Robin CPU Scheduling Algorithm”, în *2019 International Conference on Intelligent Sustainable Systems (ICISS)*, 2019, pp. 474–478, DOI: 10.1109/ISS1.2019.8908071.

[15] *Qt Development Framework for Cross-platform Applications*, https://www.qt.io/product/framework, [Accessed 13-06-2024].

[16] Abraham Silberschatz, Peter B. Galvin și Greg Gagne, „First-Come, First-Served Scheduling”, în *Operating System Concepts*, 10th, Wiley, 2018, pp. 206–207.

[17] Abraham Silberschatz, Peter B. Galvin și Greg Gagne, „Round-Robin Scheduling”, în *Operating System Concepts*, 10th, Wiley, 2018, pp. 209–211.

[18] Abraham Silberschatz, Peter B. Galvin și Greg Gagne, „Shortest-Job-First Scheduling”, în *Operating System Concepts*, 10th, Wiley, 2018, pp. 207–209.

[19] Abraham Silberschatz, Peter B. Galvin și Greg Gagne, „Shortest-Job-First Scheduling”, în *Operating System Concepts*, 10th, Wiley, 2018, pp. 232–233.

[20] Abraham Silberschatz, Peter B. Galvin și Greg Gagne, „Shortest-Job-First Scheduling”, în *Operating System Concepts*, 10th, Wiley, 2018, pp. 230–232.

[21] *Visualization with Python*, https://matplotlib.org/, [Accessed 13-06-2024].

[22] Jia Xu, „A method for adjusting the periods of periodic processes to reduce the least common multiple of the period lengths in real-time embedded systems”, în *Proceedings of 2010 IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications*, 2010, pp. 288–294, DOI: 10.1109/MESA.2010.5552058.