

ALGORITMI PARALELI ȘI DISTRIBUIȚI

Tema #2 Planificarea de task-uri într-un datacenter

Responsabili: Radu-Ioan Ciobanu, Daria Corpaci, Cosmin Lovin, Cătălin Picior, Dorian Verna, Dragoș Sofia, Alexandru Tudor, Stela Josan-Gulică, Theodor Oprea, Vlad Stanciu

Termen de predare: 17-12-2023 23:59 (soft), 20-12-2023 23:59 (hard)
Ultima modificare: 05-12-2023 20:05

Cuprins

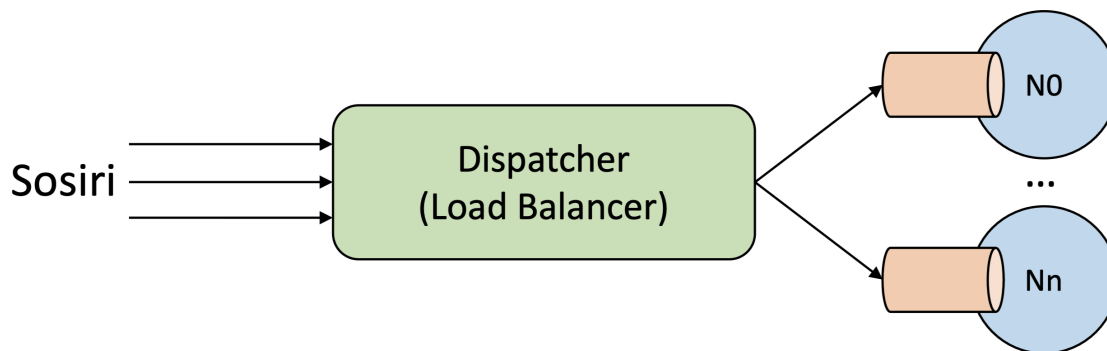
Cerință	2
Sistemul de planificare a task-urilor	2
Politici de planificare	2
Round Robin (RR)	2
Shortest Queue (SQ)	2
Size Interval Task Assignment (SITA)	2
Least Work Left (LWL)	3
Prioritatea task-urilor	3
Preemptarea task-urilor	3
Proprietățile task-urilor	3
Exemple	3
Detalii de implementare	5
Dispatcher	6
Host	6
Rulare	6
Notare	6
Testare	7

Cerință

Scopul acestei teme este implementarea unui sistem de planificare a task-urilor într-un datacenter, folosind Java Threads.

Sistemul de planificare a task-urilor

În cadrul acestei teme, se va simula arhitectura din imaginea de mai jos. Așa cum se poate observa, sistemul este compus din două elemente principale. Dispatcher-ul (sau load balancer-ul, cu verde în imagine) are rolul de a prelua task-uri care sosesc în sistem (de exemplu, de la clienți ai datacenter-ului) și de a le trimite către nodurile din datacenter pe baza unor politici prestabilite. Nodurile (marcate cu albastru în imagine, și al căror număr poate varia) au rolul de a executa task-urile pe care le primesc în funcție de priorități, de a le preempta pe cele care rulează pentru task-uri mai importante, etc. Fiecare nod din sistem are și câte o coadă în care stochează task-urile care vor fi rulate mai târziu. Scopul principal al acestei teme este implementarea logicii de la dispatcher și de la nodurile de calcul.



Politici de planificare

Dispatcher-ul pe care trebuie să îl implementați pentru această temă poate să funcționeze cu una din patru politici de planificare: Round Robin, Shortest Queue, Size Interval Task Assignment, Least Work Left.

Round Robin (RR)

În această politică de planificare, task-urile sunt alocate, pe măsură ce vin, nodului $(i + 1) \% n$, unde i este ID-ul ultimului nod la care s-a alocat un task, iar n este numărul total de noduri. Se începe întotdeauna cu ID-ul 0.

Shortest Queue (SQ)

La Shortest Queue, dispatcher-ul alocă un task acelui nod care are coada de task-uri în așteptare și execuție de dimensiune minimă. Pentru a calcula coada minimă, se ține cont inclusiv de task-urile deja aflate în rulare la un nod. Astfel, dacă avem două noduri, unul cu un task în execuție și niciunul în coada de așteptare, și altul cu niciun task în execuție sau în așteptare, dispatcher-ul va trimite un task nou la al doilea nod. Dacă sunt mai multe noduri cu aceeași dimensiune a cozii, se va trimite un task către nodul cu ID-ul mai mic.

Size Interval Task Assignment (SITA)

În cadrul acestei politici de planificare, există un număr fix de trei noduri, iar task-urile sunt grupate în trei categorii: scurte, medii, lungi. Fiecare din cele trei noduri este specific unui tip de task (în ordine, de la mic la mare), așa că task-urile sunt asigurate pe nodul corespunzător pe măsură ce sosesc în sistem. De exemplu, dacă sosește în sistem un task scurt, el va fi alocat nodului 0.

Least Work Left (LWL)

Politica Least Work Left este similară cu SQ, cu diferența că nu se mai ține cont de dimensiunea cozilor de la fiecare nod, ci de durată totală de calcule rămase de executat. Cantitatea de calcule se referă la timpul rămas de procesat la fiecare nod, care se poate aproxima la granularitate de secunde (cu alte cuvinte, dacă o coadă mai are de lucrat 2.9 secunde și alta 2.95, se consideră că cele două cozi sunt egale și se alege nodul cu ID-ul mai mic).

Prioritatea task-urilor

Prioritatea unui task reprezintă importanța acestuia în procesul de planificare. Astfel, dacă un nod are un task în coada de așteptare și primește de la dispatcher un alt task cu o prioritate mai mare, cel ce-al doilea task va fi planificat înainte de cel cu prioritate mai mică. Dacă două task-uri au priorități egale, vor fi planificate în ordinea sosirii la nod.

Preemptarea task-urilor

În anumite situații, un task aflat în execuție poate fi întrerupt pentru a se rula un alt task cu o prioritate mai mare. Acest lucru se numește preemptare, și se poate realiza doar dacă task-ul aflat în rulare este preemptibil. Astfel, dacă un nod rulează un task preemptibil și dispatcher-ul îi asignează un alt task cu o prioritate mai mare, primul task va fi preemptat și al doilea va începe să fie executat.

Proprietățile task-urilor

În sistemul propus în această temă, un task este caracterizat de următoarele proprietăți:

- **ID-ul** → întreg unic, de la 0 la $n - 1$, unde n este numărul total de task-uri
- **timpul de start** → momentul de timp la care task-ul intră în sistem
- **durata** → cantitatea de timp necesară execuției acestui task pe oricare nod de calcul
- **tipul** → un task poate fi definit ca scurt, mediu sau lung (această proprietate este relevantă doar pentru politica de planificare SITA)
- **prioritatea** → întreg care definește importanța acestui task (un task cu prioritate mai mare va avea întâietate la planificare în fața altor task-uri cu prioritatea mai mică)
- **preemptibilitatea** → o valoare de tip boolean care specifică dacă acest task poate fi preemptat de către unul cu o prioritate mai mare.

Exemple

Task-urile din exemple sunt definite de tuplul $\langle \text{timp_start}, \text{durată}, \text{tip}, \text{prioritate}, \text{preemptibilitate} \rangle$, unde timpul de start și durata sunt în secunde, tipul task-ului poate fi *scurt*, *mediu*, *lung* sau N/A^1 , iar preemptibilitatea poate fi 0 sau 1.

Primul exemplu are următoarele caracteristici:

- două noduri de calcul
- politica Round Robin
- șase task-uri cu următoarele caracteristici:

– T_0 : $\langle 0, 5, N/A, 100, 1 \rangle$

¹Tipul task-ului are relevanță doar pentru politica SITA.

- $T_1: \langle 2, 6, N/A, 200, 0 \rangle$
- $T_2: \langle 5, 1, N/A, 100, 0 \rangle$
- $T_3: \langle 1, 4, N/A, 100, 1 \rangle$
- $T_4: \langle 3, 6, N/A, 200, 0 \rangle$
- $T_5: \langle 6, 2, N/A, 100, 0 \rangle$

Se poate observa că acest scenariu are atât task-uri cu priorități diferite, cât și task-uri preemptibile. O planificare corectă a celor șase task-uri folosind politica Round Robin este prezentată în imaginea de mai jos (pe prima linie se află momentul de timp de la începerea rulării în secunde). Se poate observa că task-urile sunt alocate pe rând celor două noduri, în funcție de momentul de start. De asemenea, se mai poate observa faptul că task-urile 0 și 3 ajung să fie preemptate de task-urile 1 și 4, deoarece acestea au prioritate mai mare.

Nodul	0	1	2	3	4	5	6	7	8	9	10	11	12
N0	0	0	1	1	1	1	1	1	0	0	0	2	
N1		3	3	4	4	4	4	4	4	3	3	5	5

Al doilea exemplu are următoarele caracteristici:

- două noduri de calcul
- politica Shortest Queue
- șase task-uri cu următoarele caracteristici:

- $T_0: \langle 0, 7, N/A, 100, 0 \rangle$
- $T_1: \langle 2, 3, N/A, 100, 0 \rangle$
- $T_2: \langle 4, 4, N/A, 200, 0 \rangle$
- $T_3: \langle 1, 8, N/A, 100, 0 \rangle$
- $T_4: \langle 3, 1, N/A, 100, 0 \rangle$
- $T_5: \langle 5, 2, N/A, 200, 0 \rangle$

În acest test, niciun task nu este preemptibil, dar există task-uri cu priorități diferite. Se poate observa în imaginea de mai jos că task-urile 2 și 5, deși sosesc la cele două noduri mai târziu decât task-urile 1 și 4, ajung să fie planificate înainte de acestea datorită priorităților mai mari.

Nodul	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1							0				2			1
2								3		5	4			

Al treilea exemplu are următoarele caracteristici:

- trei noduri de calcul (obligatoriu la politica SITA)
 - politica Size Interval Task Assignment
 - nouă task-uri cu următoarele caracteristici:
- $T_0: \langle 0, 2, \textit{scurt}, 100, 0 \rangle$
 - $T_1: \langle 1, 2, \textit{scurt}, 100, 0 \rangle$
 - $T_2: \langle 2, 1, \textit{scurt}, 100, 0 \rangle$
 - $T_3: \langle 0, 5, \textit{mediu}, 100, 0 \rangle$
 - $T_4: \langle 3, 2, \textit{scurt}, 100, 0 \rangle$

- T_5 : $\langle 4, 4, \text{mediu}, 100, 0 \rangle$
- T_6 : $\langle 0, 8, \text{lung}, 100, 0 \rangle$
- T_7 : $\langle 1, 5, \text{lung}, 100, 0 \rangle$
- T_8 : $\langle 10, 1, \text{scurt}, 100, 0 \rangle$

Imaginea de mai jos prezintă rularea planificatorului în acest context, și se poate observa că toate task-urile scurte sunt rulate în ordine pe primul nod, cele medii pe al doilea nod, cele mari pe ultimul. Deoarece toate task-urile au aceleași priorități și niciunul nu este preemptibil, ele se rulează în ordinea sosirii la nodurile de calcul.

Nodul	0	1	2	3	4	5	6	7	8	9	10	11	12
1		0		1	2		4				8		
2					3				5				
3								6					7

Al patrulea exemplu are următoarele caracteristici:

- două noduri de calcul
- politica Least Work Left
- șase task-uri cu următoarele caracteristici:
 - T_0 : $\langle 0, 7, N/A, 100, 0 \rangle$
 - T_1 : $\langle 1, 3, N/A, 100, 0 \rangle$
 - T_2 : $\langle 2, 4, N/A, 100, 0 \rangle$
 - T_3 : $\langle 5, 7, N/A, 100, 0 \rangle$
 - T_4 : $\langle 9, 1, N/A, 100, 0 \rangle$
 - T_5 : $\langle 10, 2, N/A, 100, 0 \rangle$

Imaginea de mai jos prezintă rularea acestei politici. Se poate observa că, atunci când un task intră în sistem, este alocat întotdeauna nodului cu mai puține procesări rămase de făcut. De exemplu, task-ul 3 intră în sistem la momentul 5, atunci când nodul 1 mai are de procesat 2 secunde din task-ul 0, iar nodul 2 mai are de procesat 3 secunde din task-ul 2. Astfel, task-ul 3 va fi asignat la nodul 1.

Nodul	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1							0							3
2				1				2		4		5		

Detalii de implementare

Scopul vostru este de a implementa, folosind Java Threads, componentele de dispatcher și de noduri de calcul. În [repository-ul temei](#), găsiți în directorul **src** scheletul temei, care funcționează după următoarea logică:

- există mai multe thread-uri (din clasa **TaskGenerator**) care, pentru fiecare rulare, citesc task-uri (clasa **Task**) dintr-un fișier și le introduc în sistem (elementul de “Sosiri” din diagrama cu arhitectura)
- când intră în sistem, un task e trimis către dispatcher (clasa **Dispatcher**), care, pe baza algoritmului de planificare selectat, îl preia și îl trimite la un nod de calcul
- un nod de calcul (clasa **Host**) primește task-ul, îl stochează, și în cele din urmă îl execută.

Cerința presupune implementarea a două clase (**MyDispatcher** și **MyHost**) care extind clasele abstracte **Dispatcher** și **Host**, descrise în continuare.

Dispatcher

Clasa abstractă **Dispatcher** conține o referință la algoritmul de planificare folosit (din enum-ul **SchedulingAlgorithm**) și o lista de noduri de calcul, pe care le primește în constructor. De asemenea, mai conține o metodă abstractă **addTask**, care este apelată atunci când un task nou intră în sistem. La apelul acestei metode, dispatcher-ul trebuie să decidă cărui nod de calcul îi va trimite task-ul, pe baza politicii de planificare setate. Pentru a trimite un task către un nod de calcul, se apelează metoda **addTask** a acestuia.

Host

Clasa abstractă **Host** reprezintă un nod de calcul și extinde clasa **Thread**, deci logica principală va trebui implementată în metoda **run**. Pe lângă aceasta, clasa **Host** mai conține metoda abstractă **addTask** menționată mai sus (apelată de dispatcher pentru asignarea unui task la un nod de calcul), două metode abstracte **getQueueSize** și **getWorkLeft** pentru aflarea elementelor din coada nodului și cantitatea de muncă rămasă, și o metodă abstractă **shutdown**, care are rolul de a opri execuția nodului.

Atenție! Fiecare nod de calcul are rolul de a executa task-urile pe care le primește de la dispatcher. Execuția unui task cu durata de N secunde presupune că nodul de calcul unde acesta e asignat va petrece în total N secunde pentru acest task în metoda **run**. Modul în care faceți acest lucru este la latitudinea voastră (rularea unui task poate presupune consum de cicli pe procesor, așteptare, etc., important este să dureze cele N secunde). Toată planificarea se va face pe măsură ce vin task-urile, ca într-un sistem real.

Atenție! Așa cum s-a menționat anterior, unele task-uri sunt preemptibile, ceea ce înseamnă că, dacă un task preemptibil A rulează și un alt task B cu prioritate mai mare sosește la același nod de calcul, execuția lui A va fi întreruptă și se va începe execuția lui B . Execuția unui task preemptat va fi reluată doar în momentul în care nu mai există alte task-uri cu prioritate mai mare la nodul curent.

Atenție! În momentul în care un task este finalizat, trebuie apelată metoda **finish** pe obiectul **Task**.

Rulare

Tema se va compila și rula în felul următor:

```
$ javac *.java
$ java Main
```

La execuția temei, se va rula întreaga suită de teste. Dacă doriți să rulați doar o parte din teste, puteți modifica declarația array-ului **tests** din funcția **main** a clasei **Main**. Fișierele de intrare ale unui test conțin liste cu task-urile care fac parte din testul respectiv.

Notare

Tema se poate testa local sau pe VMChecker, pe modelul temei 1 și după cum se explică mai jos. Tema se va încărca pe [Moodle](#) sub forma unei arhive Zip care va trebui să conțină cel puțin următoarele fișiere **în rădăcina arhivei**:

- *MyHost.java* - implementarea proprie a logicii nodurilor de calcul
- *MyDispatcher.java* - implementarea proprie a logicii de planificare
- *README* - fișier text în care să se descrie implementarea temei.

Toate celelalte fișiere din schelet vor fi suprascrise de checker, deci nu trebuie să le modificați. Pe lângă fișierele cu implementările proprii de noduri de calcul și dispatcher, puteți să creați orice alte fișiere adiționale doriți, cu mențiunea că tema trebuie să se poată compila și rula cu comenzile prezentate mai sus.

Punctajul este divizat după cum urmează:

- **40p** - teste simple (fără priorități diferite, fără task-uri preemptibile), câte 10p pentru fiecare politică de planificare
- **40p** - teste cu task-uri cu priorități diferite, câte 10p pentru fiecare politică de planificare
- **40p** - teste cu task-uri cu priorități diferite și preemptibile, câte 10p pentru fiecare politică de planificare
- **30p** - claritatea codului și a explicațiilor din README.

Nerespectarea următoarelor cerințe va duce la depuneri:

- **-150p** - pseudo-sincronizarea firelor de execuție
- **-150p** - neexecutarea propriu-zisă a task-urilor la nodurile de calcul (de exemplu, dacă se calculează în avans la ce moment de timp ar trebui să se termine fiecare task).

Atenție! Checker-ul vă dă cele 120 de puncte pentru planificare, dar acela nu este punctajul final (dacă, de exemplu, ați încălcat vreuna din restricțiile de mai sus, veți avea nota finală 0).

Testare

Pentru a vă putea testa tema local, găsiți în [repository-ul temei](#) un set de fișiere de intrare de test, precum și un script Bash (numit *local.sh*) pe care îl puteți rula pentru a vă verifica implementarea. Rularea locală în acest mod necesită existența Docker pe sistemul vostru. Script-ul din repository va fi folosit și pentru testarea automată pe VMChecker Next, în fix aceleași condiții². Scriptul de testare locală se rulează astfel:

```
$ ./local.sh checker
```

De asemenea, puteți rula direct folosind Java dintr-un IDE sau din terminal.

Atenție! Dacă aveți un calculator Apple cu procesor ARM (ex. M1) și vreți să rulați local folosind Docker, va trebui să activați opțiunea *Use Rosetta for x86/amd64 emulation on Apple Silicon* care se găsește în meniul *Settings* din Docker Desktop. Mai multe detalii puteți găsi [aici](#).

Pentru testarea folosind VMChecker Next, găsiți detalii într-un document intitulat “[Tutorial VMChecker Next](#)” aflat pe [pagina de Moodle](#) a cursului de APD.

Atenție! Checker-ul are instalat Oracle JDK 17.

²Nota obținută în urma rulării automate poate fi scăzută pe baza elementelor de depunere descrise mai sus.