Programare paralela si distribuita – Laborator 4

Munteanu Bianca-Stefania

Grupa 235/1

05.12.2022

**Analiza cerințelor**

* **Obiectiv:**
* Intelegerea/aprofundarea sablonului “producator-consumator”
* Intelegerea/aprofundarea sincronizarii conditionale
* Intelegerea/aprofundarea excluderii mutuale (granularitatea sectiunilor critice)
* **Tema**: Se considera **n** polinoame reprezentate prin lista de monoame. Se cere adunarea polinoamelor folosind o implementare multithreading (p threaduri).
* **Reprezentare:** - reprezentarea unui polinom in memorie: lista inlantuita (1 nod=1monom) ordonata dupa exponentii monoamelor cu urmatorul INVARIANT (predicat adevarat la orice moment al executiei) de reprezentare:

-monoamele sunt ordonate dupa exponenti

-nu se pasteaza in lista monoame cu coeficient 0;

- nu exista doua noduri (monoame) cu acelasi exponent

* **Datele de intrare**
* polinoamele se citesc din fisiere – cate un fisier pentru fiecare polinom;
* un fisier contine informatii de tip (coeficient, exponent) pentru fiecare monom al unui polinom,
* fisierele input se creeaza prin generare de numere aleatoare.

(Conditie: fisierele nu contin monoame cu coeficient egal cu 0 dar nu sunt ordonate dupa exponent!)

* **Implementare:** Java
* **Testare:** masurati timpul de executie pentru

1. 10 polinoame fiecare cu gradul maxim 1000 si cu maxim 50 monoame
   1. p = 4, 6, 8
   2. secvential
2. 5 polinoame fiecare cu gradul maxim 10000 si cu maxim 100 monoame
   1. p = 4, 6, 8
   2. secvential

Fisierele trebuie creat anterior prin adaugare de numere generate aleator. Toate rularile trebuie executate cu acelasi fisier.

* **Postconditie:** Fisierul “result.txt” va contine suma celor n polinoame.

**Proiectare**

**Reprezentare**

Cu ajutorul clasei **Monomial** vom reprezinta nodurile listei inlantuite. Fiecare obiect de acest tip va retine coeficientul, exponentul si legatura catre urmatorul monom.

class Monomial {

protected int coefficient;

protected int exponent;

protected Monomial next;

}

**PolyLinkedList** reprezinta o lista inlantuita si va contine referinta catre primul monom (nod) al listei

public class PolyLinkedList {

private Monomial head;

}

**MyQueue** este o coada sincronizata pe care o voi utiliza in varianta paralelizata a programului. Continue dimensiunea curenta, referinta catre primul si ultimul monom.

public class MyQueue {

private int size;

private Monomial head;

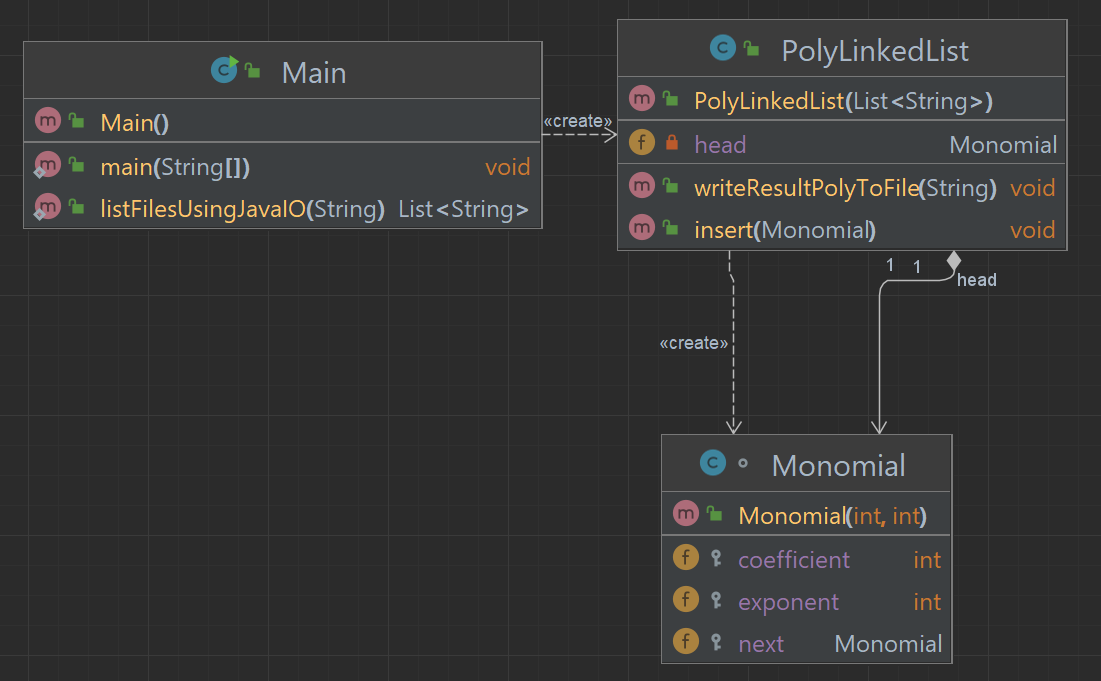
private Monomial tail;

}

**Implementare**

Am generat datele de test intr-un program separat si am copiat fisierele in cele doua proiecte corespunzatoare variantei secventiale, respectiv paralelizate.

**Metoda A) - Implementare secventiala**



Pasi:

1. Adaug intr-o lista numele fisierelor in care se afla polinoamele ce trebuie adunate
2. Pornesc timerul
3. In constructorul clasei PolyLinkedList, parcurg numele fisierelor primite ca parametru, le iterez pe rand linie cu linie si adaug fiecare monom in lista rezultat -> vezi pasul 4 (cu respectarea conditiei ca invariantul ramane adevarat dupa fiecare adaugare de monom)
4. Inserare in lista inlantuita ordonata:

* Daca lista este goala, atunci initializez primul element
* Daca exponentul primului element >= exponentul monomului curent
  + Daca au acelasi coeficient, maresc coeficientul
  + Altfel, adaug monomul current la inceputul listei
* Altfel, caut pozitia pe care trebuie sa inserez monomul current
  + Daca au acelasi coeficient, maresc coeficientul
  + Altfel, adaug monomul current pe pozitia gasita

1. Opresc timerul
2. Afisez pe ecran timpul de executie al programului

**Metoda B) - Implementare paralela – p threaduri**

O imagine care conține text, negru, tabelă de marcaj, captură de ecran

Descriere generată automat

Pasi:

1. Primesc ca argument p (nr de threaduri)
2. Adaug intr-o lista numele fisierelor in care se afla polinoamele ce trebuie adunate
3. Pornesc timerul
4. Creez threadul producator

Thread producator:

* + citeste cate un monom
  + daca coada e plina, asteapta pana cand consumatorii elibereaza un loc si notifica producatorul
  + adauga monomul intr-o structura de date de tip coada si notifica consumatorii
  + dupa ce se termina cititrea, threadul 0 isi incheie executia si mai notifica o data consumatorii pentru a aduna ultimele elemente ramase in coada

1. Creez p-1 threaduri consumator

Threadurile consumator:

* + daca coada e plina, dar inca nu am terminat de citit monoamele din fisier, asteapta
  + daca coada e plina, si am terminat de citit monoamele din fisier, consumatorii isi incheie executia
  + preiau cate un monom din coada, il aduna la polinomul reprezentat in lista si notifica threadurile

1. Pornesc threadurile
2. Astept threadurile
3. Primul thread scrie rezultatul obtinut in lista L intr-un fisier rezultat, ignorand monomii cu coeficientul 0
4. Opresc timerul
5. Afisez pe ecran timpul de executie al programului

La finalul executiei celor 2 metode, compar intr-un nou program fisierele rezultat, pentru a ma asigura ca au aceleasi elemente.

public static long filesCompareByByte(Path path1, Path path2) throws IOException

**Detaliere sincronizare**

In produce() si consume() am folosit sincronizare la nivel de bloc pentru coada.

Ex:

while (*queue*.getSize() == *maxSize*) {  
 synchronized (*queue*) {  
 //System.out.println(thName + " is waiting...Queue is full");  
 *queue*.wait();  
 }  
}

In clasa PolyLinkedList retin un obiect ReentrantLock, pe care il folosesc la inserarea unui nou element in lista, doar pe sectiunile critice. ReentrantLock este un mecanism de excludere mutuala care permite threadurilor sa blocheze o resursă (de mai multe ori) fara a intra in deadlock.

private ReentrantLock lock = new ReentrantLock();

Ex:

lock.lock();  
// if list is empty , we mark the new node as head  
if (head == null) {  
 head = monomial;  
 lock.unlock();  
} else .....

In clasa Queue metodele de enqueue() si dequeue() sunt marcate cu synchronized.

synchronized public void enqueue(Monomial Monomial)

synchronized public Monomial dequeue(){

**Rezultate**

|  |  |  |
| --- | --- | --- |
| Cazuri | **Nr procese** | **Timp executie** (ms) |
|  |  |  |
| Caz1)  nrPolinoame = 10  gradMaxim = 1000  maxMonoame = 50 | secvential | 12,90942 |
| 4 | 22.54222 |
| 6 | 22.90111 |
| 8 | 24.82689 |
|  |  |  |
| Caz2)  nrPolinoame = 5  gradMaxim = 10000  maxMonoame = 100 | secvential | 10.14872 |
| 4 | 20.79004 |
| 6 | 21,48626 |
| 8 | 23,584 |

**Analiza rezultatelor**

Observatii:

* Varianta secventiala este de cel putin 2 ori mai buna decat cea paralelizata deoarece gestionarea threadurilor e costisitoare si majoritatea operatiilor sunt aproape secventializate (pentru ca avem blocari la nivelul listei si cozii)
* Timpul creste usor odata cu marirea numarului de threaduri

In continuare, voi reprezenta grafic rezultatele obtinute pentru fiecare varianta:

**Grafice**

* **secvential vs paralel – Caz 1)**

nrPolinoame = 10

gradMaxim = 1000

maxMonoame = 50

* **secvential vs paralel – Caz 2)**

nrPolinoame = 5

gradMaxim = 10000

maxMonoame = 100