**Laborator 5**

1. **Analiza Cerintelor**

Se considera n polinoame reprezentate prin lista de monoame.

Se cere adunarea polinoamelor folosind o implementare multithreading (p threaduri).

Consideratii generale:

- 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

- 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!)

1. **Proiectare si implementare**

public class Node {  
 public int coefficient; **// coeficientul unui monom**   
  
 public int exponent; **// exponentul unui monom**  
  
 public Node next;  
  
 public Lock lock = new ReentrantLock(); **// lock**   
 public Node() {  
 }  
  
 public Node(int coefficient, int exponent) {  
 this.coefficient = coefficient;  
 this.exponent = exponent;  
 }  
  
 public void lock(){ **// blocheaza lockul**   
 lock.lock();  
 }  
  
 public void unlock(){ **//elibereaza lockul**  
  
 lock.unlock();  
 }

}

public class MyQueue {  
 final Integer MAX\_SIZE = 5; **// dimensiunea maxima a cozii**   
 final Node[] items = new Node[MAX\_SIZE]; **// Coada in care se depoziteaza monoamele**   
 final Lock lock = new ReentrantLock();  
 final Condition notFull = lock.newCondition();  
 final Condition notEmpty = lock.newCondition();  
 int putptr, takeptr, count;  
  
  
 public MyQueue(){  
  
 }  
 public void add(Node node) throws InterruptedException {  
 lock.lock();  
 try {  
 while (count == items.length) **// cat timp coada este plina threadurile asteapta**   
 notFull.await();  
 items[putptr] = node;  
 if (++putptr == items.length) putptr = 0;  
 ++count;  
 notEmpty.signal(); **// se notifica threadule ca s-a adaugat un nod**  
 } finally {  
 lock.unlock();  
 }  
 }  
  
 public Node poll() throws InterruptedException {  
 lock.lock();  
 try {  
 while (count == 0) **// cat timp coada este vida threadurile asteapta**  
 notEmpty.await();  
 Node x = items[takeptr];  
 if (++takeptr == items.length) takeptr = 0;  
 --count;  
 notFull.signal();  
 return x;  
 } finally {  
 lock.unlock();  
 }  
  
 }  
  
 public void stop(Integer noOfWorkers) throws InterruptedException {  
 for (int i = 0; i < noOfWorkers; ++i) {  
 this.add(null); **// se adauga monoame null pentru a anunta threadurile workers ca s-au epuizat toate monoamele POISON PILL**  
  
  
 }  
  
 }  
  
}

public class MyList {  
 public Node sentinelHead;  
 public Node sentinelTail;  
  
 MyList(){  
 sentinelHead = new Node(0,0);  
 sentinelTail = new Node(0,0);  
 sentinelHead.next = sentinelTail;  
 }  
  
 public void addNode(Node node){  
 Node prev = sentinelHead;  
 prev.lock();   
 Node current = sentinelHead.next;  
 current.lock();

**// tot timpul blochez cate o pereche de 2 noduri**   
 while( current != sentinelTail) {  
 if (node.exponent == current.exponent) {  
 current.coefficient += node.coefficient;  
 if (current.coefficient == 0) {  
 // delete the node  
 current.next.lock();  
 prev.next = current.next;  
 current.next.unlock();  
 }  
 break;  
 }  
 else if ( current.exponent > node.exponent){  
 node.next = current;  
 prev.next = node;  
 break;  
 }  
 prev.unlock();  
 prev = current;  
 current = current.next;  
 current.lock();  
 }

if (current == sentinelTail) {  
 node.next = sentinelTail;  
 prev.next = node;  
 }  
 prev.lock.unlock();  
 current.lock.unlock();  
 }  
  
 public List<Node> getResultSum() {  
 List<Node> result = new ArrayList<>();  
 for (Node current = sentinelHead.next; current != sentinelTail; current = current.next) {  
 result.add(current);  
 }  
 return result;  
 }  
}

1. **Executie**
2. static class Producer implements Runnable{  
     
    int nrOfWorkers;  
    MyQueue myQueue;  
    int polynomialNumber;  
    CyclicBarrier producerBarrier;  
    int start;  
    int end;  
    public Producer(MyQueue myQueue, int nrOfWorkers, int polynomialNumber, CyclicBarrier producerBarrier, int start, int end) {  
    this.myQueue = myQueue;  
    this.nrOfWorkers = nrOfWorkers;  
    this.polynomialNumber = polynomialNumber;  
    this.producerBarrier = producerBarrier;  
    this.start = start;  
    this.end = end;  
    }  
     
    @Override  
    public void run() {  
    for(int nr = start; nr < end; nr++) {  
     
    // each producer should read a number of files  
    String filename = "Lab5/resources/polynom[" + nr + "].in";  
    try (BufferedReader bufferedReader = new BufferedReader(new FileReader(filename))) {  
    String line;  
    while ((line = bufferedReader.readLine()) != null) {  
    if (line.equals(""))  
    continue;  
    String[] numbers = line.split(" ");  
    Integer coef = Integer.*valueOf*(numbers[0]);  
    Integer exp = Integer.*valueOf*(numbers[1]);  
    Node node = new Node(coef, exp);  
    myQueue.add(node);  
    }  
    } catch (FileNotFoundException e) {  
    throw new RuntimeException(e);  
    } catch (IOException e) {  
    throw new RuntimeException(e);  
    } catch (InterruptedException e) {  
    throw new RuntimeException(e);  
    }  
    }  
    try {  
    // barrier  
    producerBarrier.await();

**// dupa ce toate threaduile producator au trecut de bariara se adagua noduri null in coada pentru a opri threadurile workers**  
 myQueue.stop(nrOfWorkers);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 } catch (BrokenBarrierException e) {  
 throw new RuntimeException(e);  
 }  
  
 }  
 }  
}

static class Worker implements Runnable{  
  
 MyQueue myQueue;  
 MyList result;  
  
 public Worker(MyQueue myQueue, MyList result) {  
 this.myQueue = myQueue;  
 this.result = result;  
 }  
  
 @Override  
 public void run() {  
 Node node = null;  
 try {  
 node = myQueue.poll();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 while (node != null) {  
 result.addNode(node);  
 try {  
 node = myQueue.poll();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 }  
}

public static void main(String[] args) {  
 *nrOfThreads* = args.length > 1 ? Integer.*valueOf*(args[0]) : 5;  
// polynomialNumber = args.length > 2 ? Integer.valueOf(args[1]) : 2;  
 *nrOfProducers* = 2;  
 *nrOfWorkers* = *nrOfThreads* - *nrOfProducers*;  
 MyQueue myQueue = new MyQueue();  
 Thread[] readers = new Thread[*nrOfProducers*];  
 *producersBarrier* = new CyclicBarrier(*nrOfProducers*);  
 int rest = *polynomialNumber* % *nrOfProducers*;  
 int start = 0;  
 int end = *polynomialNumber* / *nrOfProducers*;  
 for(int i = 0; i < *nrOfProducers*; i++) {  
 if( rest > 0){  
 end++;  
 rest--;  
 }  
 readers[i] = new Thread(new Producer(myQueue, *nrOfWorkers*, *polynomialNumber*, *producersBarrier*, start, end));  
 start = end;  
 end += *polynomialNumber* / *nrOfProducers*;  
 }  
  
 Thread[] workers = new Thread[*nrOfWorkers*];  
 MyList result = new MyList();  
 for(int i = 0 ; i < *nrOfWorkers*; i++) {  
 workers[i] = new Thread(new Worker(myQueue, result));  
 }  
 long startTime = System.*nanoTime*();  
 for(int i = 0; i < *nrOfProducers*; i++) {  
 readers[i].start();  
 }  
 for(int i = 0 ; i < *nrOfWorkers*; i++) {  
 workers[i].start();  
 }  
  
 try {  
 for(int i = 0; i < *nrOfProducers*; i++) {  
 readers[i].join();  
 }  
 for(int i = 0 ; i <*nrOfWorkers*; i++) {  
 workers[i].join();  
 }  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 List<Node> resultNodes = result.getResultSum();  
 long endTime = System.*nanoTime*();  
 System.*out*.println((double)(endTime - startTime)/1E6);//ms  
  
 try(BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter("Lab5/resources/polynomPar.out"))){  
 for(Node node: resultNodes){  
 bufferedWriter.write(node.coefficient + " " + node.exponent + "\n");  
 }  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }

}

1. **Cazuri de testare**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Numar Polinoame** | **Nr Threaduri** | **Nr Prod** | **Timp de executie** | **Timp de excutie lab 4** |
| 10 polinoame fiecare cu gradul maxim 1000 si cu maxim 50 monoame MAX = 20 | 1 |  | 16.5866 | 14.34758 |
|  | 4 | P1 = 2 | **11.5194** | **17.35755** |
|  | 4 | P1 =3 | **10.7906** |  |
|  | 6 | P1 = 2 | **12.6349** | **19.00652** |
|  | 6 | P1 = 3 | **12.0737** |  |
|  | 8 | P1 = 2 | **11.878** | **19.17593** |
|  | 8 | P1 = 3 | **12.6268** |  |
| 5 polinoame fiecare cu gradul maxim 10000 si cu maxim 100 monoame MAX = 30 | 1 |  | **11.1122** | **13.12514** |
|  | 4 | P1 = 2 | **12.9298** | **15.07311** |
|  | 4 | P1 =3 | **12.358** |  |
|  | 6 | P1 = 2 | **14.7497** | **15.65924** |
|  | 6 | P1 = 3 | **14.7677** |  |
|  | 8 | P1 = 2 | **15.4056** | **23.04057** |
|  | 8 | P1 = 3 | **16.0211** |  |

1. **Concluzii**
   1. Varianta paralela nu ofera un timp mai bun, probabil din cauza ca dimensiunea datelor este in continuare redusa.
   2. Se observa o imbunatatire a timpului paralel fata de timpul de la labul 4 deoarece lock-ul se realizeaza la nivel de 2 noduri, nu la nivelul intregii liste.