Laboratorium Nr 5
"Problem czytelników i pisarzy, blokowanie drobnoziarniste"
Radosław Kopeć
04.11.2020

I. Zadanie 1

1. Treść zadania

Ćwiczenie - badanie efektywności

Problem czytelników i pisarzy proszę rozwiązać przy pomocy: semaforów i zmiennych warunkowych

Proszę wykonać pomiary dla różnej ilości czytelników (10-100) i pisarzy (od 1 do 10).

W sprawozdaniu proszę narysować 3D wykres czasu w zależności od liczby wątków i go zinterpretować.

2. Koncepcja rozwiązania

Bufor będzie tablicą liczb całkowitych o konkretnej wielkości. Moja implementacja będzie faworyzowała czytelników gdyż dopiero gdy żaden czytelnik nie będzie czytał zasobu to dopiero wtedy pisarz będzie mógł coś zapisać do bufora. Implementacja tego mechanizmu polega na stworzeniu dwóch semaforów dla każdego pola bufora, oraz dodatkowej tablicy liczb całkowitych określającą liczbę czytelników na danym polu.

Pisarz chcąc uzyskać dostęp do zasobu próbuje przejąć semafor danego pola. Gdy kończy swoje działania zwalnia semafor.

Czytelnik ma trochę trudniejszą budowę. Chcąc uzyskać dostęp do zasobu najpierw zajmuje semafor pilnujący dostępu do tablicy przechowującej liczbę czytelników na danym polu. Gdy mu się to uda sprawdza ilu czytelników czyta dane pole. Jeśli żaden czytelnik nie czyta pola to blokuje on dostęp do zasobu próbując zająć semafor buffora dla danego pola. Natomiast gdy jacyś czytelnicy już czytają dane pole to po nie próbuje zająć semafora bufora gdyż wtedy by się zablokował.

Czytelnik wykonuje swoje prace po czym próbuje zakończyć. I tutaj podobnie najpierw uzyskuje dostęp do tablicy przechowującej liczbę czytelników na danym polu. Sprawdza

liczbę czytelników i jeśli jest ona większa od zera to nie zwalnia semafora bufora dla swojego pola. Natomiast gdy jest on ostatnim czytelnikiem oznacza to że musi zwolnić semafor bufora żeby ewentualny pisarz mógł wejść.

Operacja czytania dla czytelnika będzie czytała losowała index z bufora z którego czytelnik chce przeczytać. Analogicznie będzie działać operacja zapisu dla pisarza.

3. **Implementacja i wyniki:**

Reader

```
public class Reader extends Thread{

    private final IBuffer _buf;
    private final int operations;
    public Reader(IBuffer buffer, int operations){
        super();
        this._buf = buffer;
        this.operations = operations;
    }

    public void run() {

        Random random = new Random();

        for (int i = 0; i < operations; i++) {
            _buf.get(random.nextInt(_buf.getSize()));
        }
    }

}
```

IBuffer

```
public interface IBuffer {
    void put(int value,int index);
    int get(int index);
    int getSize();
}
```

Writer

```
public class Writer extends Thread {

    private final IBuffer _buf;
    private final int operations;

    public Writer(IBuffer buffer, int operations){
        super();
        this._buf = buffer;
        this.operations = operations;
    }

    public void run() {

        Random random = new Random();

        for (int i = 0; i < operations; i++) {
            int toWrite = random.nextInt( bound: 10000);
            int randomIndex = random.nextInt(_buf.getSize());
            _buf.put(toWrite,randomIndex);
        }
    }

}
```

Buffer

```
public class Buffer implements IBuffer{

    private final int[] buffer;
    private final int size;
    private final int[] howManyReadersInField;
    private final Semaphore[] readSemaphore;
    private final Semaphore[] indexSemaphore;
    private final long readTime;
    private final long writeTime;

    public Buffer(int size,long readTime, long writeTime){
        this.size = size;
        this.readTime = readTime;
        this.writeTime = writeTime;

        buffer = new int[size];
        howManyReadersInField = new int[size];
        readSemaphore = new Semaphore[size];
        indexSemaphore = new Semaphore[size];

        for (int i =0;i < size; i++){
            buffer[i]= 0;
            indexSemaphore[i] = new Semaphore( permits: 1);
            readSemaphore[i] = new Semaphore( permits: 1);
            howManyReadersInField[i] = 0;
        }

    }
```

Kluczowe metody bufora:

```
public void put(int value,int index) {

    try {
        readSemaphore[index].acquire();

    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    buffer[index] = value;
    try {
        Thread.sleep( millis: 0,(int) writeTime);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    readSemaphore[index].release();

}
```

```
public int get(int index){

    try {
        indexSemaphore[index].acquire();
        howManyReadersInField[index]++;

        //when we are first, we should to block writer
        if(howManyReadersInField[index] == 1){
            readSemaphore[index].acquire();
        }
        indexSemaphore[index].release();

    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    int result = buffer[index];
    try {
        Thread.sleep( millis: 0,(int) readTime);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    try {
        indexSemaphore[index].acquire();

        howManyReadersInField[index]--;
        //when we are last reader, we should unlock writer
        if(howManyReadersInField[index] == 0){
            readSemaphore[index].release();
        }
        indexSemaphore[index].release();

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return result;
}
```

Klasa testujaca:

```
public static void main(String[] args) {

    final int minReaders =10;
    final int maxReaders = 100;
    final int minWriters = 1;
    final int maxWriters = 10;
    final int writers = maxWriters - minWriters + 1;
    final int readers = maxReaders - minReaders + 1;
    final int bufferSize = 10;
    final int countOfReaderOperations = 10;
    final int countOfWriterOperations = 10;
    long[][] measurement = new long[writers][readers];
    final long readTime = 100;
    final long writeTime = 1;


    for(int r = minReaders; r<= maxReaders; r++){
        for(int w = minWriters; w <= maxWriters; w++){
            ArrayList<Thread> threads = new ArrayList<~>();

            Buffer buffer = new Buffer(bufferSize,readTime,writeTime);

            for(int i=0;i<r;i++){
                threads.add(new Reader(buffer,countOfReaderOperations));
            }
            for(int i=0;i<w;i++){
                threads.add(new Writer(buffer,countOfWriterOperations));
            }

            measurement[w - minWriters][r - minReaders] = measureTime(threads);
        }
        System.out.println(((double)r/maxReaders)*100 + "%");
    }

    create3dPlot(measurement,writers,readers, surfaceName: "The time in function of readers and writers");
}
```

```
private static long measureTime(ArrayList<Thread> threads){
    long startTime = System.nanoTime();

    for( Thread thread: threads){
        thread.start();
    }

    for( Thread thread: threads){
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    return System.nanoTime() - startTime;
}
```

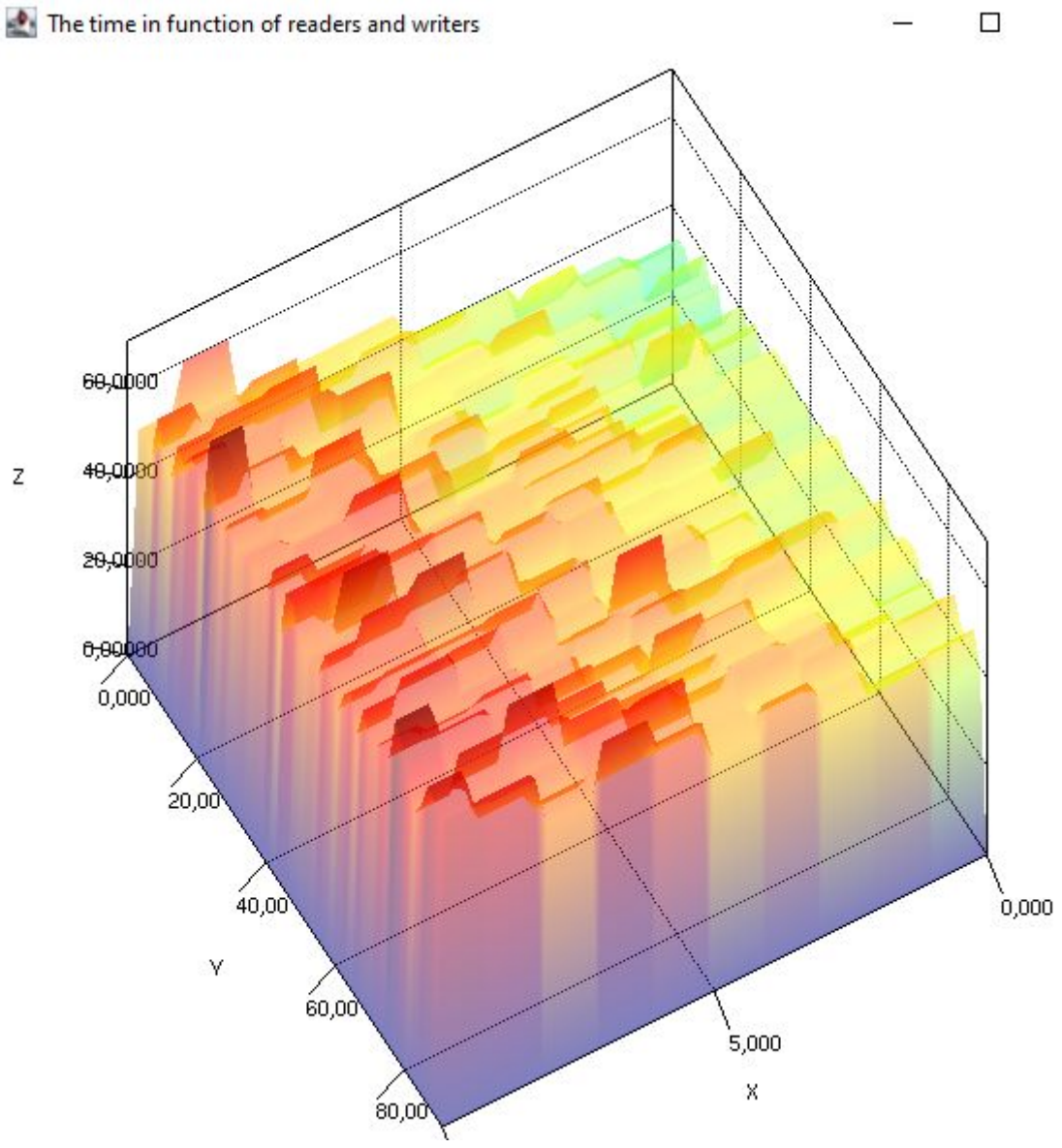
```
    private static void create3dPlot(long[][] table,int X,int Y,String surfaceName){

// Define a function to plot
        Mapper mapper = (x, y) -> {
                if(x >= X || y  >= Y) return 0;
                return (double) table[(int) x ][ (int) y]/1000000;
        };

// Define range and precision for the function to plot
        Range xRange = new Range(0, X);
        Range yRange = new Range(0, Y);
        int steps = 50;

// Create a surface drawing that function
        Shape surface = Builder.buildOrthonormal(new OrthonormalGrid(xRange, steps,yRange,steps), mapper);
        surface.setFaceDisplayed(true);
        surface.setColorMapper(new ColorMapper(new ColorMapRainbow(), surface.getBounds().getZmin(), surface.getBounds().getZmax(), new Color(
        surface.setWireframeDisplayed(false);
        surface.setWireframeColor(Color.BLACK);

// Create a chart and add the surface
        Chart chart = new AWTChart(Quality.Advanced);
        chart.add(surface);
        chart.open(surfaceName,  width: 600,  height: 600);
    }
```

Wyniki pomiarów:

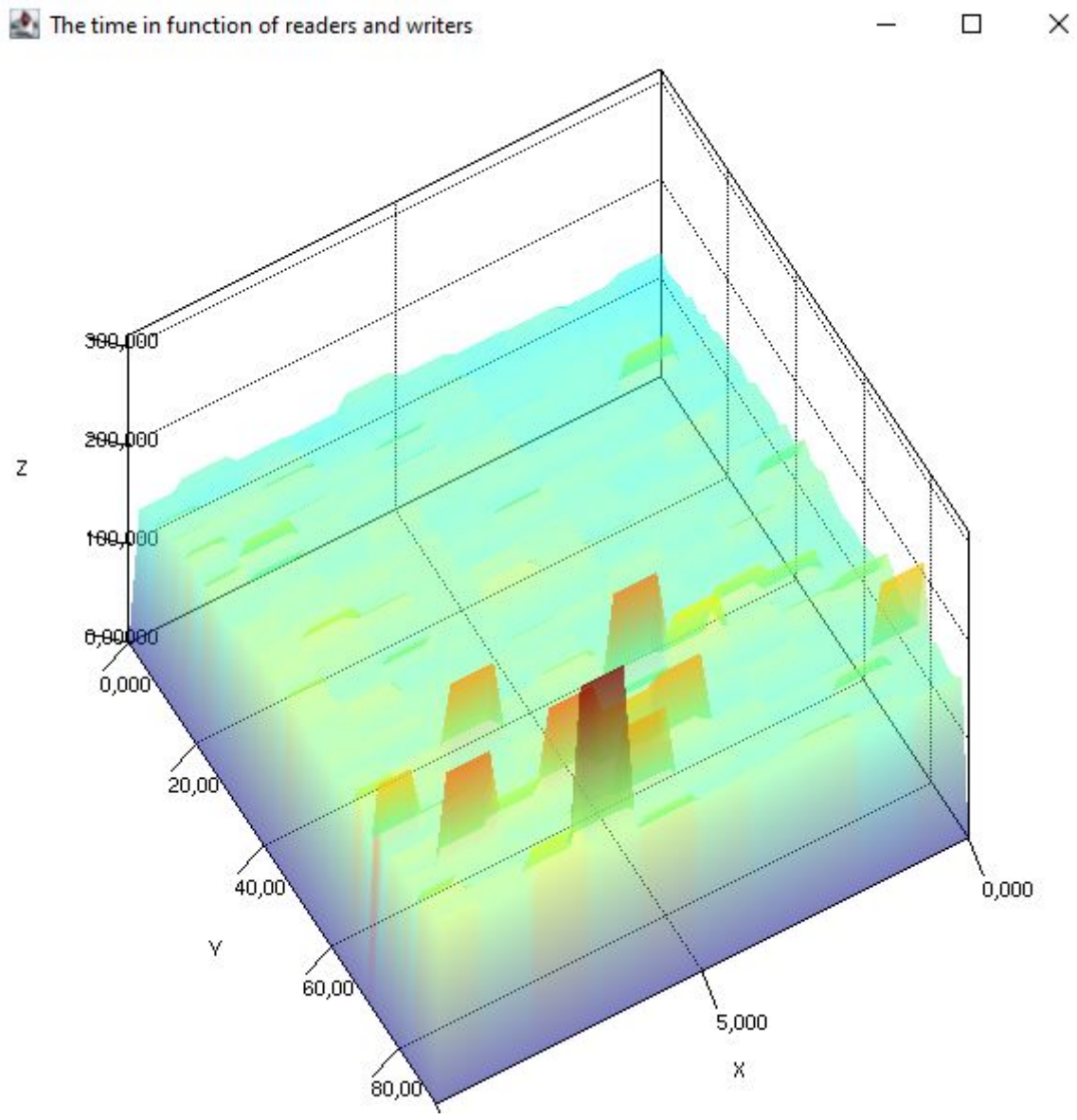
Dla różnych stosunków czasów operacji dla Czytelnika i Pisarza.

Legenda:
X - liczba pisarzy
Y -liczba czytelników
Z - czas wykonania

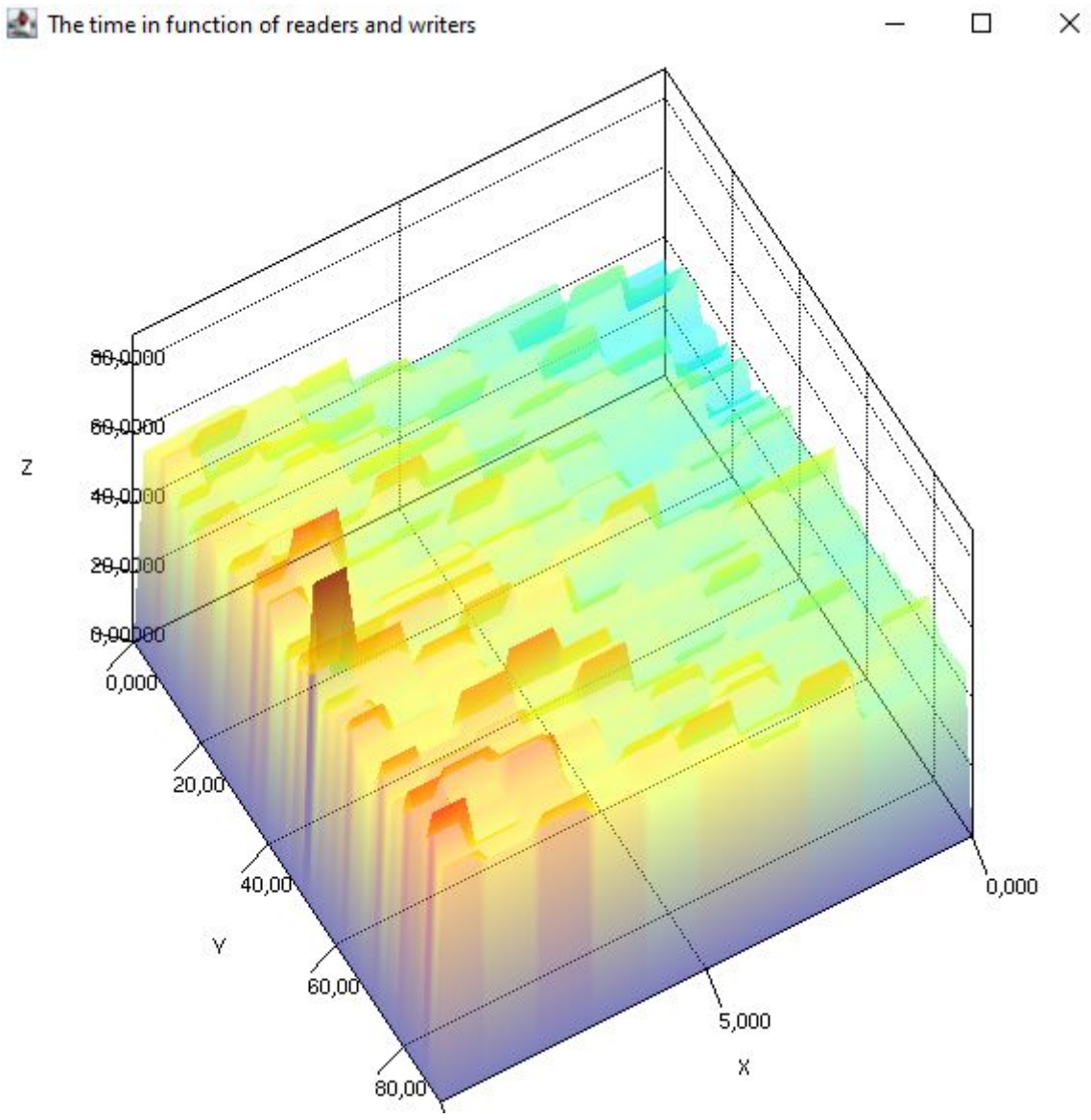
Wykres 1. czas_czytania/czas_zapisu = 1



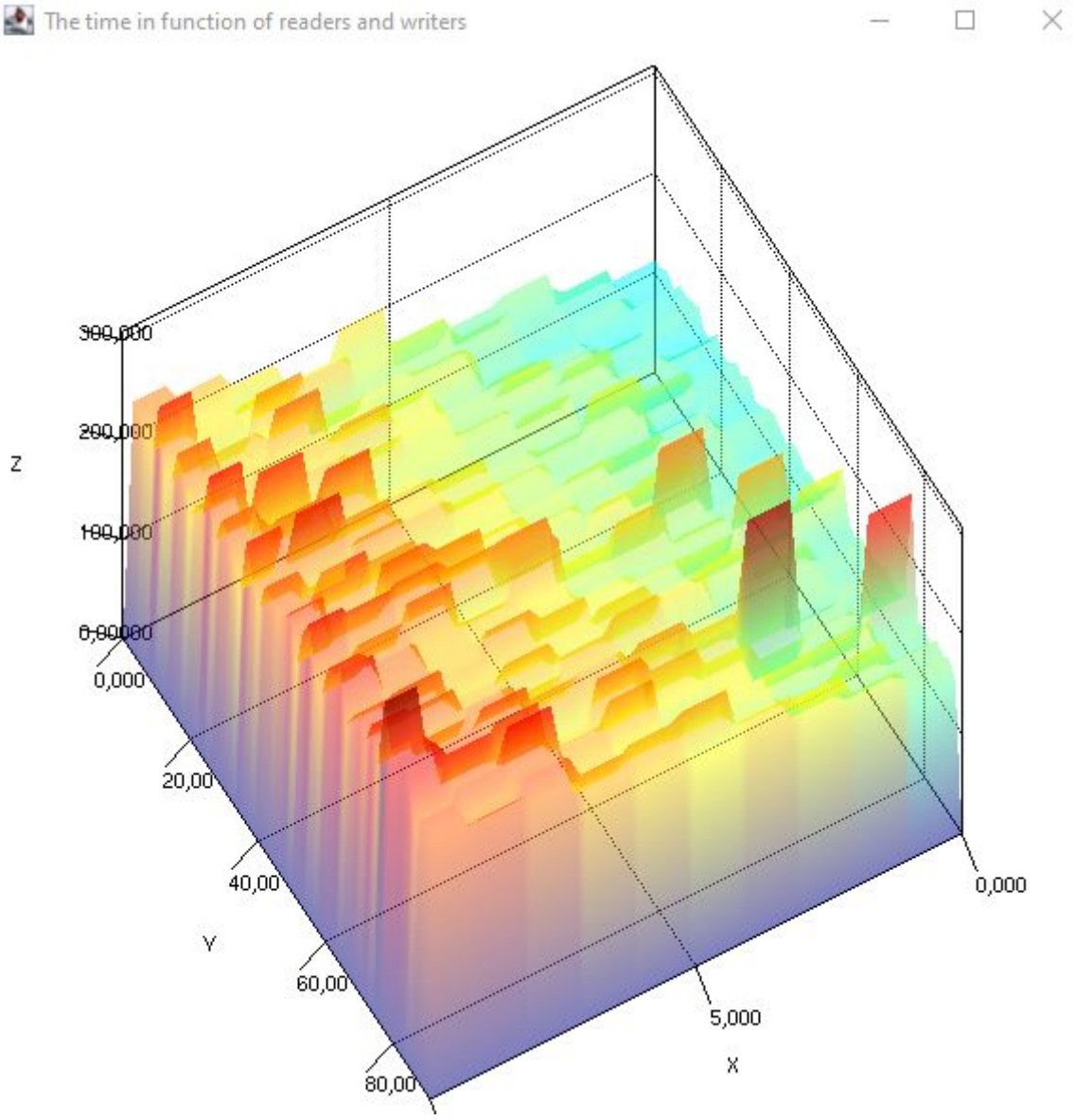
Wykres 2. czas_czytania/czas_zapisu = 10



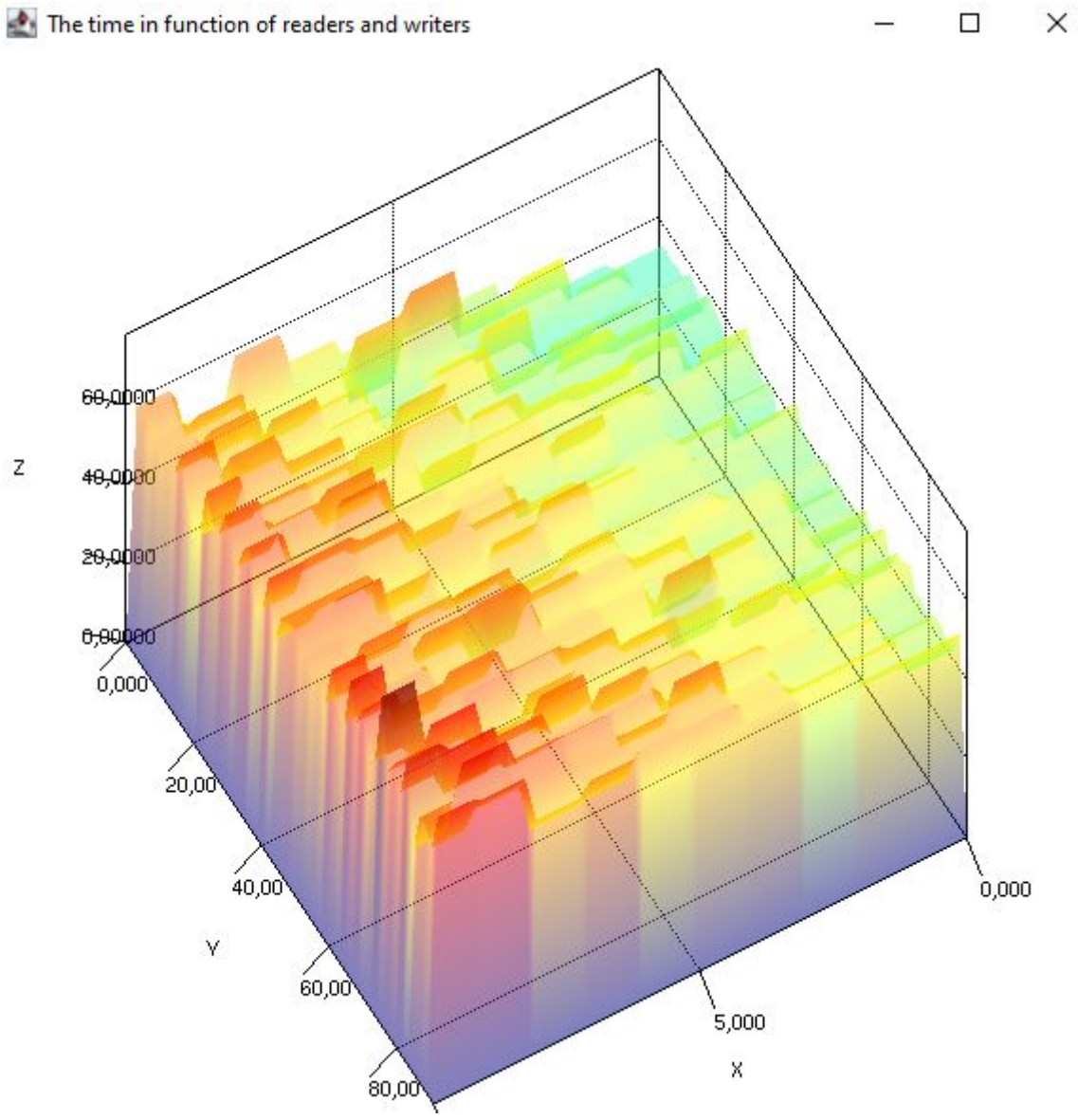
Wykres 3. czas_czytania/czas_zapisu = 100



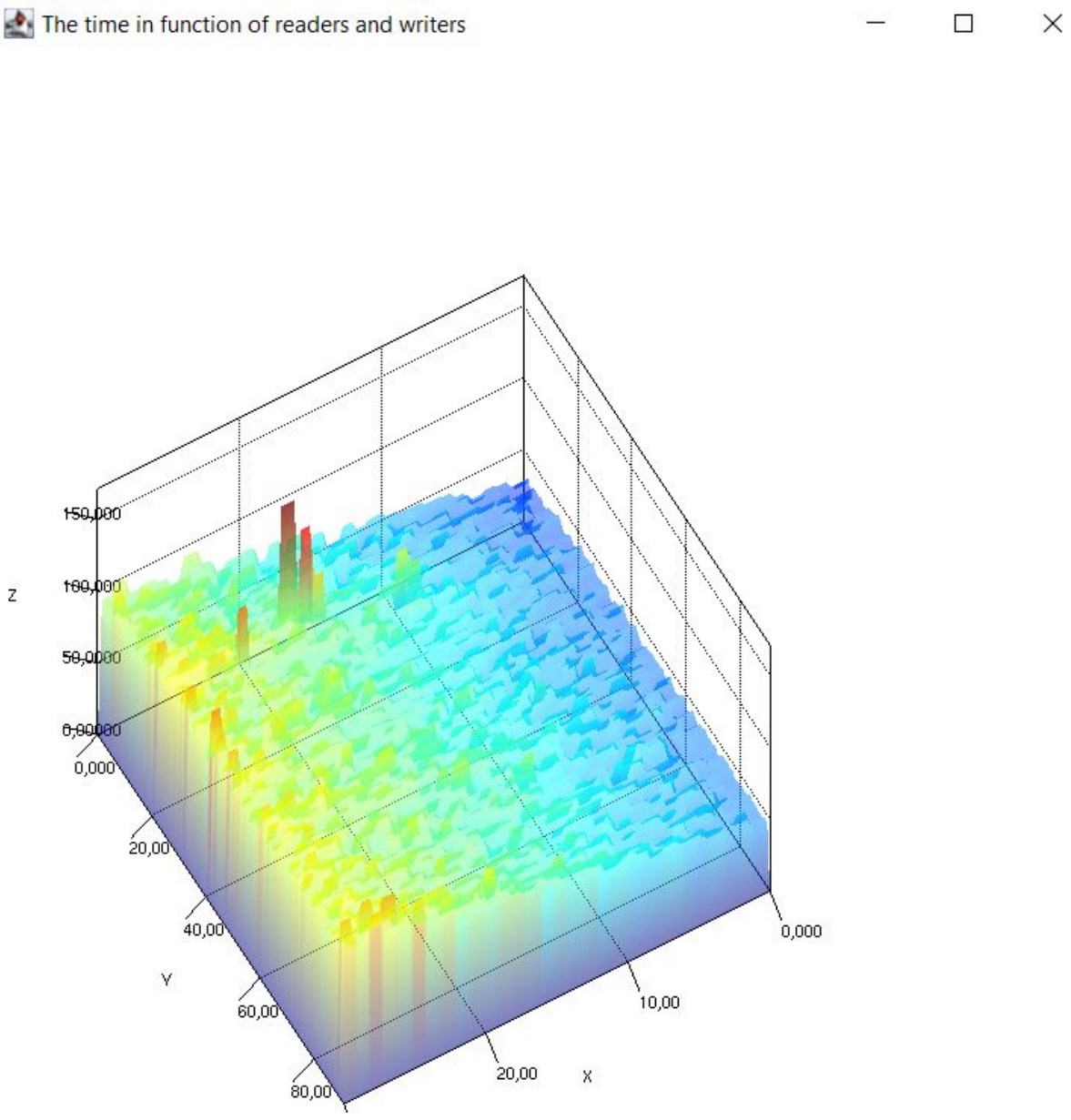
Wykres 4. czas_czytania/czas_zapisu = 0.1



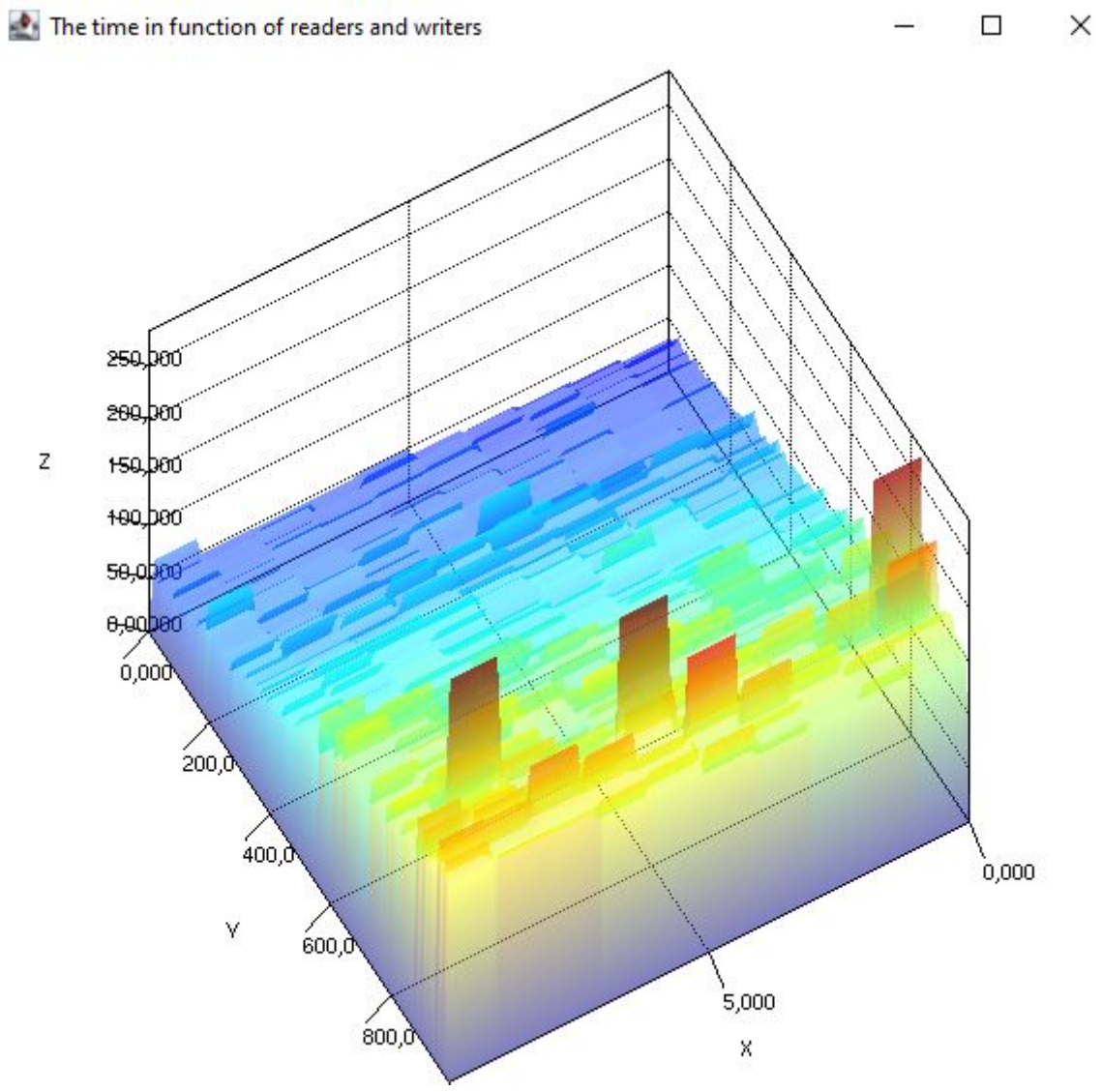
Wykres 5. czas_czytania/czas_zapisu = 0.01



Wykres 6. Dla producentów od 1 do 30 i czytelników od 1 do 100 czas_czytania/czas_zapisu = 1



Wykres 7. Dla producentów od 1 do 10 i czytelników od 1 do 1000 czas_czytania/czas_zapisu = 1



4. Wnioski

- Wykres 7. Mamy tutaj ogromną przewagę czytelników (aż do 1000). Można zauważyć, że czas rośnie liniowo w stosunku do liczby czytelników. Jeżeli liczba czytelników znacznie przekracza liczbę pisarzy to czas wykonania jest wprost proporcjonalny do liczby czytelników. Pisarze są wtedy głodzeni przez dużą liczbę czytelników.

- Wykres 6. Mamy dużą liczbę pisarzy przewagę. Można zauważyć, że czas rośnie liniowo w stosunku do liczby pisarzy. Jeżeli liczba psiarzy jest duża to zajmują oni

cały zasób, wiele czytelników musi czekać na zwolnienie zasobu.

- Wykresy 2,3. wykres tutaj nabiera płaskiego stałego kształtu. Czytelnicy szybko wykonują swoje operacje co zwiększa szanse psiarza na dostęp do zasobu, co za tym idzie zmniejsza zjawisko głodzenia pisarzy.

- Wykresy 1,4,5. Gdy czas zapisu jest większy od czasu czytania pisarze blokują wielu czytelników, dużo wątków długo czeka na dostęp do zasobu, powoduje to zwiększenie średniego czasu oczekiwania. Na tych wykresach można zauważyć, że głównie wraz ze zwiększaniem się liczby pisarzy rośnie czas wykonywania.

5. Bibliografia

http://home.agh.edu.pl/~funika/tw/lab5/

II. Zadanie 2

1. Treść zadania

Blokowanie drobnoziarniste

1. Zamek (lock) jest przydatny wtedy, gdy operacje zamykania/otwierania nie mogą być umieszczone w jednej metodzie lub bloku synchronized. Przykładem jest zakładanie blokady (lock) na elementy struktury danych, np. listy. Podczas przeglądania listy stosujemy następujący algorytm:

1. zamknij zamek na pierwszym elemencie listy
2. zamknij zamek na drugim elemencie
3. otwórz zamek na pierwszym elemencie
4. zamknij zamek na trzecim elemencie
5. otwórz zamek na drugim elemencie

6. powtarzaj dla kolejnych elementów

Dzięki temu unikamy konieczności blokowania całej listy i wiele wątków może równocześnie przeglądać i modyfikować różne jej fragmenty.

Ćwiczenie

1. Proszę zaimplementować listę, w której każdy węzeł składa się z wartości typu Object, referencji do następnego węzła oraz zamka (lock).
2. Proszę zastosować metodę drobnoziarnistego blokowania do następujących metod listy:

```
boolean contains(Object o); //czy lista zawiera element o

boolean remove(Object o); //usuwa pierwsze wystąpienie elementu o

boolean add(Object o); //dodaje element o na końcu listy
```

3. Proszę porównać wydajność tego rozwiązania w stosunku do listy z jednym zamkiem blokującym dostęp do całości. Należy założyć, że koszt czasowy operacji na elemencie listy (porównanie, wstawianie obiektu) może być duży - proszę wykonać pomiary dla różnych wartości tego kosztu.

2. **Koncepcja rozwiązania**

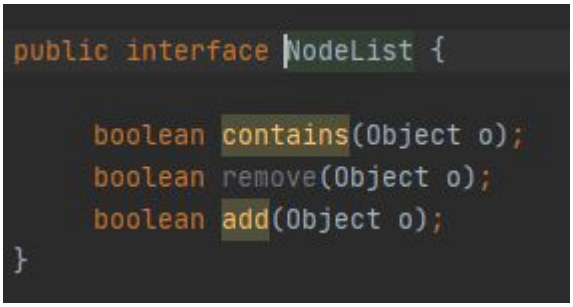
Koncepcja jest bardzo prosta w jednej z list dostęp do całej listy będzie strzegł zamek “Lock”, za każdym razem jak chcemy wykonać jakąkolwiek operację musimy uzyskać dostęp do całej listy. Powoduje to, że każda operacja wymaga dostępu do całej listy.
Druga lista będzie miała zamek na każdym polu “Node”, sposób przemieszczania się po liście będzie analogiczny do przedstawionego przez Pana przykładu. Stworzę też jedną klasę NodeListExecutor która będzie miała za zadanie wykonać kilka operacji na liście.

Pozwolę sobie zbadać wydajności list w zależności nie tylko od kosztu, ale również od ilości wątków obsługujących tę listę, ponieważ przewaga kolejki z wieloma blokadami polega

właśnie na wielodostępie. Sprawdzę też przypadek dla jednego wątku, listy powinny się wtedy zachowywać podobnie.

3. **Implementacja i wyniki:**

NodeList



```
public interface NodeList {

    boolean contains(Object o);
    boolean remove(Object o);
    boolean add(Object o);
}
```

NodeListExecutor

```
public class NodeListExecutor extends Thread {

    private final NodeList list;
    private final int operations;

    public NodeListExecutor(NodeList list,int operations){
        super();
        this.list = list;
        this.operations = operations;
    }

    @Override
    public void run() {

        for(int i=0;i< operations;i++){
            list.add(i);
            list.contains(i);

        }
    }
}
```

Node

```java
public class Node {

    Object node;
    Node next;
    final Lock lock = new ReentrantLock();
    final int cost;

    public Node(Object node, Node next, int cost) {
        this.node = node;
        this.next = next;
        this.cost = cost;
        executeCost();
    }

    public void executeCost(){
        try {
            Thread.sleep( millis: 0,cost);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public boolean hasNext(){
        executeCost();
        return next != null;
    }
    public void lock() { lock.lock(); }

    public void unlock() { lock.unlock(); }

    public Object getNode() {
        executeCost();
        return node;

    }
```

```java
public void setNode(Object node) {
    this.node = node;
    executeCost();
}


public Lock getLock() { return lock; }

public Node getNext() {
    executeCost();
    return next;
}

public void setNext(Node next) {
    executeCost();
    this.next = next;
}

@Override
public String toString() {
    if(hasNext()){
        return node.toString() + " " + next.toString() + " ";
    }
    return node.toString();
}
```

NodeListMultipleLocks

```java
private final Node nodeHead;
private final int cost;

public NodeListMultipleLocks(int cost) {
    this.nodeHead = new Node( node: null, next: null,cost);
    this.cost = cost;
}

@Override
public boolean contains(Object o){
    Node pointer = nodeHead;

    pointer.lock();
    Node nextPointer = pointer.getNext();

    while (pointer.hasNext()){
        nextPointer.lock();
        if(nextPointer.getNode().equals(o)){
            pointer.unlock();
            nextPointer.unlock();
            return true;
        }
        Node tmp = pointer;
        pointer = pointer.getNext();
        tmp.unlock();
        nextPointer = nextPointer.getNext();
    }
    pointer.unlock();
    return false;
}
```

```java
@Override
public boolean remove(Object o){
    Node pointer = nodeHead;

    pointer.lock();
    Node nextPointer = pointer.getNext();

    while (pointer.hasNext()){
        nextPointer.lock();
        if(nextPointer.getNode().equals(o)){
            pointer.setNext(nextPointer.getNext());
            pointer.unlock();
            nextPointer.unlock();
            return true;
        }
        Node tmp = pointer;
        pointer = pointer.getNext();
        tmp.unlock();
        nextPointer = nextPointer.getNext();
    }
    pointer.unlock();
    return false;
}
```

```java
@Override
public boolean add(Object o){

    Node pointer = nodeHead;

    pointer.lock();
    Node nextPointer = pointer.getNext();

    while (pointer.hasNext()){
        nextPointer.lock();
        Node tmp = pointer;
        pointer = pointer.getNext();
        tmp.unlock();
        nextPointer = nextPointer.getNext();
    }

    pointer.setNext(new Node(o, next: null,cost));
    pointer.unlock();
    return true;
}

@Override
public String toString() {

    if(nodeHead.hasNext()){
        return "[" + nodeHead.getNext().toString() + "]";
    }
    return "[]";
}
```

NodeListWithOneLock

```java
public class NodeListWithOneLock implements NodeList {

    private final Node nodeHead;
    private final Lock lock = new ReentrantLock();
    private final int cost;

    public NodeListWithOneLock(int cost) {
        this.nodeHead = new Node( node: null, next: null,cost);
        this.cost = cost;
    }

    @Override
    public boolean contains(Object o){

        lock.lock();

        try {
            Node pointer = nodeHead;
            Node nextPointer = pointer.getNext();

            while (pointer.hasNext()) {
                if (nextPointer.getNode().equals(o)) {
                    return true;
                }
                pointer = pointer.getNext();
                nextPointer = nextPointer.getNext();
            }
        }
        finally {
            lock.unlock();

        }
        return false;
    }
```

```java
@Override
public boolean remove(Object o){

    lock.lock();
    try {
        Node pointer = nodeHead;
        Node nextPointer = pointer.getNext();

        while (pointer.hasNext()) {
            if (nextPointer.getNode().equals(o)) {
                pointer.setNext(nextPointer.getNext());
                return true;
            }
            pointer = pointer.getNext();
            nextPointer = nextPointer.getNext();
        }
    }
    finally {
        lock.unlock();
    }
    return false;
}
```

```java
@Override
public boolean add(Object o){

    lock.lock();
    try {
        Node pointer = nodeHead;
        Node nextPointer = pointer.getNext();

        while (pointer.hasNext()) {
            pointer = pointer.getNext();
            nextPointer = nextPointer.getNext();
        }

        pointer.setNext(new Node(o,  next: null, cost));
    }
    finally {
        lock.unlock();
    }
    return true;
}
```

```java
@Override
public String toString() {

    if(nodeHead.hasNext()){
        return "[" + nodeHead.getNext().toString() + "]";
    }
    return "[]";
}
```

Klasa testująca:

```java
    public static void main(String[] args) {

//        za zasadne uznam zbadanie wydajności list w zależności nie tylko od
//        zajmuje on po porstu cała kolejke, przewaga kolejki z wieloma bloka

//        Global Parameters
        final int operations = 1;

//       Cost measurement parameters
        final int minCost = 1;
        final int maxCost = 100;
        final int costStep = 1;
        final int threadsForCostMeasurement = 5;
        final int countOfCost = maxCost - minCost + 1;

//       Threads measurement parameters

        final int minExecutors = 1;
        final int maxExecutors = 30;
        final int costForThreadMeasurement = 10;
        final int countOfExecutors = maxExecutors - minExecutors + 1;

//        Visualization

        double[] oneLockCost = new double[countOfCost];
        double[] multipleLocksCost = new double[countOfCost];
        double[] xCostAxis = new double[countOfCost];

        double[] oneLockThreads = new double[countOfExecutors];
        double[] multipleLocksThreads = new double[countOfExecutors];
        double[] xThreadsAxis = new double[countOfExecutors];

        double[] oneLockCostOneThread = new double[countOfCost];
        double[] multipleLocksCostOneThread = new double[countOfCost];
        double[] xConstOneThreadAxis = new double[countOfCost];
```

```
//        Measurement the time for multiple Threads and const cost

        for(int cost=minCost;cost<=maxCost;cost+=costStep) {

            NodeList oneLockList = new NodeListWithOneLock(cost);
            NodeList multipleLocksList = new NodeListMultipleLocks(cost);

            ArrayList<Thread> executorsOneLock = new ArrayList<>();
            ArrayList<Thread> executorsMultipleLocks = new ArrayList<>();

            for (int i = 0; i < threadsForCostMeasurement; i++) {
                executorsOneLock.add(new NodeListExecutor(oneLockList, operations));
                executorsMultipleLocks.add(new NodeListExecutor(multipleLocksList, operations));
            }

            oneLockCost[cost - minCost] = (double) measureTime(executorsOneLock);
            multipleLocksCost[cost - minCost] = (double) measureTime(executorsMultipleLocks);
            xCostAxis[cost - minCost] = cost;
            System.out.println("Cost Measurement: " +  ((double) cost/maxCost) * 100.0 + "%");
        }
```

```
//        Measurement the time for multiple costs and const count of threads
        System.out.println("Threads measurement:");
        for(int ex =minExecutors; ex <= maxExecutors; ex++) {

            NodeList oneLockList = new NodeListWithOneLock(costForThreadMeasurement);
            NodeList multipleLocksList = new NodeListMultipleLocks(costForThreadMeasurement);

            ArrayList<Thread> executorsOneLock = new ArrayList<>();
            ArrayList<Thread> executorsMultipleLocks = new ArrayList<>();

            for (int i = 0; i < ex; i++) {
                executorsOneLock.add(new NodeListExecutor(oneLockList, operations));
                executorsMultipleLocks.add(new NodeListExecutor(multipleLocksList, operations));
            }

            oneLockThreads[ex - minExecutors] = (double) measureTime(executorsOneLock);
            multipleLocksThreads[ex - minExecutors] = (double) measureTime(executorsMultipleLocks);
            xThreadsAxis[ex - minExecutors] = ex;

            System.out.println("Threads Measurement: " + ((double) ex/maxExecutors) * 100 + "%");

        }
```

```java
//        Measurement the time for multiple costs and one thread

        for(int cost=minCost;cost<=maxCost;cost+=costStep) {

            NodeList oneLockList = new NodeListWithOneLock(cost);
            NodeList multipleLocksList = new NodeListMultipleLocks(cost);

            ArrayList<Thread> executorsOneLock = new ArrayList<>();
            ArrayList<Thread> executorsMultipleLocks = new ArrayList<>();

            executorsOneLock.add(new NodeListExecutor(oneLockList, operations));
            executorsMultipleLocks.add(new NodeListExecutor(multipleLocksList, operations));

            oneLockCostOneThread[cost - minCost] = (double) measureTime(executorsOneLock);
            multipleLocksCostOneThread[cost - minCost] = (double) measureTime(executorsMultipleLocks);
            xConstOneThreadAxis[cost - minCost] = cost;
            System.out.println("Cost Measurement one thread: " + ((double) cost/maxCost) * 100 + "%");
        }
```

```java
//        visualization

        draw2DPlots(xCostAxis,oneLockCost,xCostAxis,multipleLocksCost, name: "Lock na całą listę", name2: "Lock na każdy element listy", plotName: "Porównanie zależności czasu od kosztu");
        draw2DPlots(xThreadsAxis,oneLockThreads,xThreadsAxis,multipleLocksThreads, name: "Lock na całą listę", name2: "Lock na każdy element listy", plotName: "Porównanie zależności czas
        draw2DPlots(xConstOneThreadAxis,oneLockCostOneThread,xConstOneThreadAxis,multipleLocksCostOneThread, name: "Lock na całą listę", name2: "Lock na każdy element listy", plotName:

    }
```

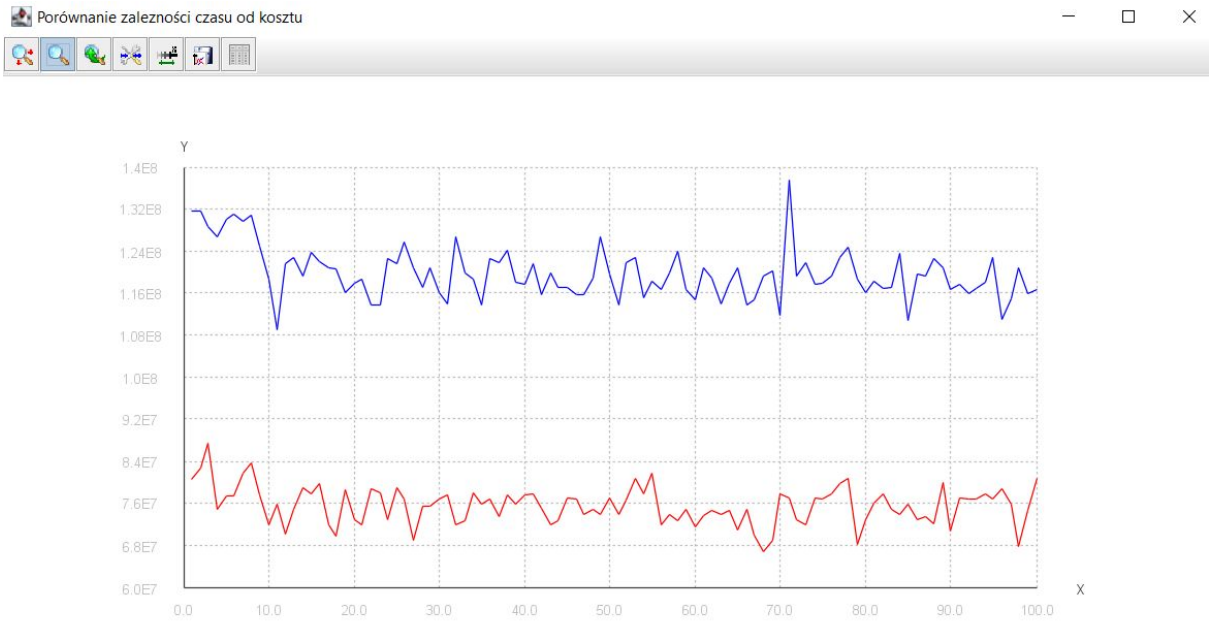
```java
    private static long measureTime(ArrayList<Thread> threads){
        long startTime = System.nanoTime();

        for( Thread thread: threads){
            thread.start();
        }

        for( Thread thread: threads){
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        return System.nanoTime() - startTime;
    }
```

```java
    private static void draw2DPlots(double[] arguments, double[] values, double[] arguments2, double[] values2, String name, String name2, String plotName){

        Plot2DPanel plot = new Plot2DPanel();
        // add a line plot to the PlotPanel
        plot.addLinePlot(name, arguments, values);
        plot.addLinePlot(name2, arguments2, values2);

        // put the PlotPanel in a JFrame, as a JPanel
        JFrame frame = new JFrame(plotName);
        frame.setContentPane(plot);
        frame.setVisible(true);
    }
}
```

4. Wyniki pomiarów i wnioski

Czerwony wykres - lista blokadą na każde pole
Niebieski wykres - lista z jedną blokadą

Zależność czasu wykonania od kosztu **dla wielu wątków.**

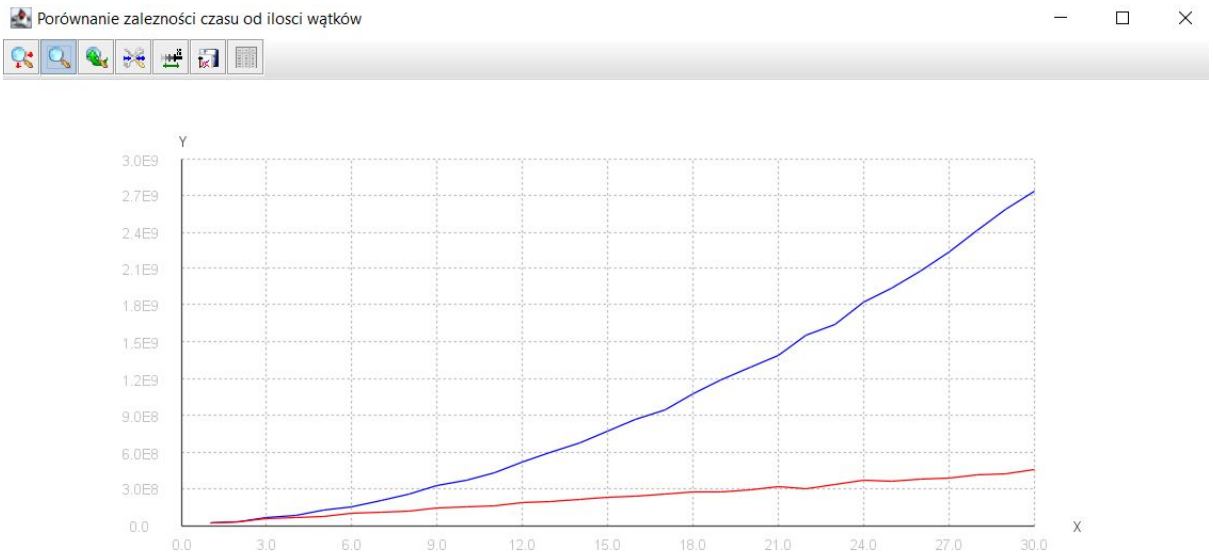


Wniosek:

Czas wykonania operacji dla obu list oscyluje w stałym przedziale co oznacza, że koszt nie miał tu dużego wpływu. Natomiast lista z blokowaniem drobnoziarnistym wypadła lepiej, ma generalnie niższy czas operacji. Jest to spowodowane tym, że przykład ten był wykonywany dla stałej liczby wątków równej 5, więc wątki korzystały z dobrodziejstw listy z wieloma zamkami.

Zależność czasu wykonania od ilości wątków **dla stałego kosztu.**

Czerwony wykres - lista blokadą na każde pole
Niebieski wykres - lista z jedną blokadą

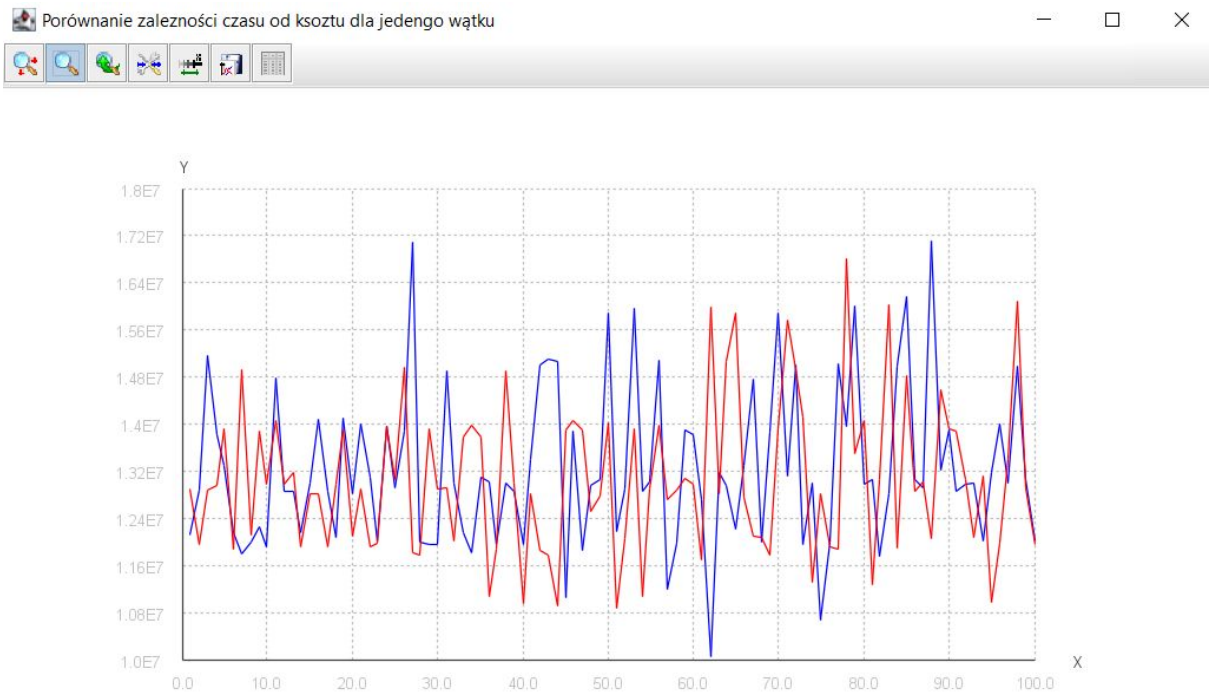


Wniosek:

Jak widać lista z wieloma zamkami znacznie lepiej radzi sobie z wieloma wątkami. Wykres czerwony rośnie naprawdę wolno w porównaniu z wykresem niebieskim. Czas dla listy z jedna blokadą jest wprost proporcjonalny do liczby wątków.

Zależność czasu wykonania od kosztu **dla jednego wątku.**

Czerwony wykres - lista blokadą na każde pole
Niebieski wykres - lista z jedną blokadą



Wniosek:

Jest to przypadek dla jednego wątku. Obie listy zachowują się bardzo podobnie. Czas operacji oscyluje dla nich wokół tego samego przedziału. Jest to spowodowane tym, że dla jednego wątku uzyskuje on dostęp do całej listy na wyłączność. Obie listy zachowują się wtedy jak zwykła lista. Gdybyśmy zwiększali liczbę wątków to wykres niebieski przesówałby się o dodatnią stałą.

Powodem dla którego koszt nie miał dużego wpływu na wyniki jest to, że wahał się on w dość małym przedziale 1 do 100.

5. Bibliografia

http://home.agh.edu.pl/~funika/tw/lab5/
https://github.com/yannrichet/jmathplot

Radosław Kopeć