Sprawozdanie

Teoria Współbieżności

Laboratorium 5

Joanna Bryk, środa 17:50

1. Wstęp
   1.1. Pierwszym zadaniem było zaimplementowanie rozwiązania problemu czytelników i pisarzy przy pomocy semaforów i zmiennych warunkowych. Kolejnym krokiem było wykonanie wykresu 3D czasu w zależności od liczby wątków.
   1.2. Zadanie drugie polegało na implementacji listy, w której wykorzystywany jest mechanizm blokowania drobnoziarnistego. Następnie należało zaimplementować listę, w której stosuje się jeden zamek blokujący dostęp do całości i porównać wydajności tych rozwiązań.
2. Implementacja
   2.1. W zadaniu tym wzorowałam się na rozwiązaniu z podręcznika *Programowanie współbieżne i rozproszone w przykładach i zadaniach* autorstwa Z. Weiss, T. Gruźlewski (zadanie 4.2.3, rozwiązanie poprawne). Stworzyłam nowe klasy *Writer*, *Reader* i *Library*.

```
class Reader extends Thread {
    private Library library;
    public Reader(Library library) {
        this.library = library;
    }

    public void run() {
        for(int i = 0; i < 100; i++){
            library.beginReading();
            library.endReading();
        }
    }

}
class Writer extends Thread {
    private Library library;

    public Writer(Library library) {
        this.library = library;
    }

    public void run() {
        for(int i = 0; i < 100; i++){
            library.beginWriting();
            library.endWriting();
        }
    }
}
```

```
public class Library {
    private final ReentrantLock lock = new ReentrantLock();
    private final Condition readers = lock.newCondition();
    private final Condition writers = lock.newCondition();
    private int reading = 0;
    private int writing = 0;

    public Library() {
    }

    public void beginReading() {
        lock.lock();
        try {
            while (writing > 0 && lock.hasWaiters(writers)) {
                readers.await();
            }
            reading += 1;
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        } finally {
            lock.unlock();
        }
    }

    public void endReading() {
        lock.lock();
        try {
            reading -= 1;
            if (reading == 0)
                writers.signal();
        } finally {
            lock.unlock();
        }
    }

    public void beginWriting() {
        lock.lock();
        try {
            while (reading + writing > 0) {
                writers.await();
            }
            writing = 1;
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        } finally {
            lock.unlock();
        }
    }

    public void endWriting() {
        lock.lock();
        try {
            writing = 0;
            if (!lock.hasWaiters(readers))
                writers.signal();
            else
                readers.signalAll();
        } finally {
            lock.unlock();
        }
```

```
    }
}
```

2.2. Zgodnie z poleceniem każdy element listy składał się z wartości typu Object, referencji do następnego węzła oraz zamka (użyty *java.util.concurrent.locks.ReentrantLock).* Dodatkowo zaimplementowałam funkcje pomocnicze.

```
public class Node {

    private final Object o;
    private Node next_element;
    private final ReentrantLock lock = new ReentrantLock();

    public Node(Object o, Node next_element) {
        this.o = o;
        this.next_element = next_element;
    }

    public void addNext(Node next_element) {
        this.next_element = next_element;
    }

    public boolean isLast() {
        return this.next_element == null;
    }

    public Object getObject() {
        return this.o;
    }

    public Node getNext_element() {
        return this.next_element;
    }

    public ReentrantLock getLock() {
        return this.lock;
    }
}
```

Następnie zaimplementowane zostały dwie klasy – *ListManyLocks* i *ListOneLock*. W każdej z nich znajdują się trzy metody: *boolean contains (Object o), boolean remove (Object o), boolean add (Object o).* Działanie tych metod jest podobne jedynie z różnicą dotyczącą blokowania.

Lista z blokowaniem drobnoziarnistym:

```
public class ListManyLocks {

    private Node head;
    private int sleep_time;

    public ListManyLocks(Object o, int sleep_time) {
        this.sleep_time = sleep_time;
        head = new Node(o, null);

    }

    boolean contains(Object o) {
```

```
        Node curr = head;
        curr.getLock().lock();

        while (curr != null) {
            if (curr.getObject() == o) {
                try {
                    Thread.sleep(sleep_time);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                curr.getLock().unlock();
                return true;
            }
            if (!curr.isLast())
                curr.getNext_element().getLock().lock();
            curr.getLock().unlock();
            curr = curr.getNext_element();
        }

        return false;

    }

    //czy lista zawiera element o
    boolean remove(Object o) {

        head.getLock().lock();

        if (head.getObject() == o) {
            Node tmp = head;
            head = head.getNext_element();
            tmp.getLock().unlock();
            try {
                Thread.sleep(sleep_time);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return true;
        }
        Node curr = head;
        while (!curr.isLast()) {
            Node next = curr.getNext_element();
            next.getLock().lock();
            if (next.getObject() == o) {
                Node next_next = next.getNext_element();
                curr.addNext(next_next);
                curr.getLock().unlock();
                next.getLock().unlock();
                try {
                    Thread.sleep(sleep_time);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                return true;
            }
            curr.getLock().unlock();
            curr = curr.getNext_element();
        }
        return false;
    }
```

```
    boolean add(Object o) {
        Node curr = head;
        curr.getLock().lock();

        while (!curr.isLast()) {
//            System.out.println("Dodaje" + o);
            curr.getNext_element().getLock().lock();
            curr.getLock().unlock();
            curr = curr.getNext_element();
        }
        try {
            Thread.sleep(sleep_time);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        curr.addNext(new Node(o, null));
        curr.getLock().unlock();
        return true;
    }
}
```

```
public class ListOneLock {

    private Node head;
    private final ReentrantLock lock = new ReentrantLock();
    private int sleep_time;


    public ListOneLock(Object o, int sleep_time) {
        this.head = new Node(o, null);
        this.sleep_time = sleep_time;
    }

    boolean contains(Object o) {
        lock.lock();
        try {

            Node curr = head;

            while (curr != null) {
                if (curr.getObject() == o) {
                    try {
                        Thread.sleep(sleep_time);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    return true;
                }
                curr = curr.getNext_element();
            }
        } finally {
            lock.unlock();
        }
        return false;
    }

    boolean remove(Object o) {
        lock.lock();
```

```
        try {
            if (head.getObject() == o) {
                head = head.getNext_element();
                try {
                    Thread.sleep(sleep_time);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                return true;
            }
            Node curr = head;
            while (!curr.isLast()) {
                Node next = curr.getNext_element();
                if (next.getObject() == o) {
                    Node next_next = next.getNext_element();
                    curr.addNext(next_next);
                    try {
                        Thread.sleep(sleep_time);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    return true;
                }
                curr = curr.getNext_element();
            }
        } finally {
            lock.unlock();
        }
        return false;
    }

    public boolean add(Object o) {
        lock.lock();
        try {
            Node curr = head;
            while (!curr.isLast()) {
                curr = curr.getNext_element();
            }
            try {
                Thread.sleep(sleep_time);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            curr.addNext(new Node(o, null));
        } finally {
            lock.unlock();
        }
        return true;
    }
}
```

Dodatkowo stworzyłam klasy *ListManyLocksThread* i *ListOneLockThread.*

```
public class ListManyLocksThread extends Thread {
    private ListManyLocks listManyLocks;

    public ListManyLocksThread(ListManyLocks list) {
        this.listManyLocks = list;
    }
```

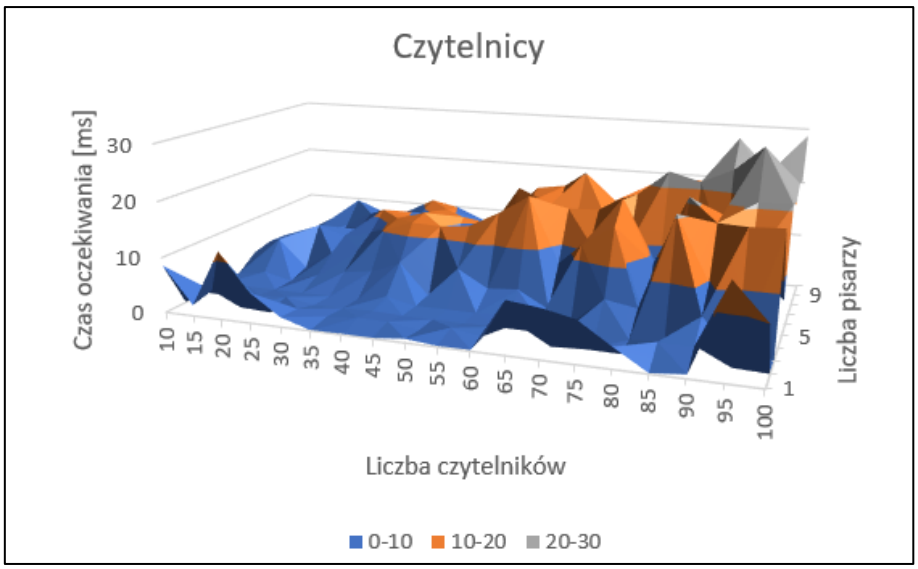
```
    public void run() {
        for (int i = 0; i < 10; i++) {
            if (i % 2 == 0) {
                this.listManyLocks.add(i);
                boolean check = this.listManyLocks.contains(i + 1);
                System.out.println("Lista nie zawiera : " + (i + 1) + "
" + check);
            } else {
                this.listManyLocks.add("NIEPARZYSTA");
                boolean check1 = this.listManyLocks.contains(i - 1);
                System.out.println("Lista zawiera : " + (i - 1) + " " +
check1);

                this.listManyLocks.remove(i-1);
                check1 = this.listManyLocks.contains(i - 1);
                System.out.println("Lista zawiera (druga proba) : " + (i
- 1) + " " + check1);
            }
        }
    }
}
```

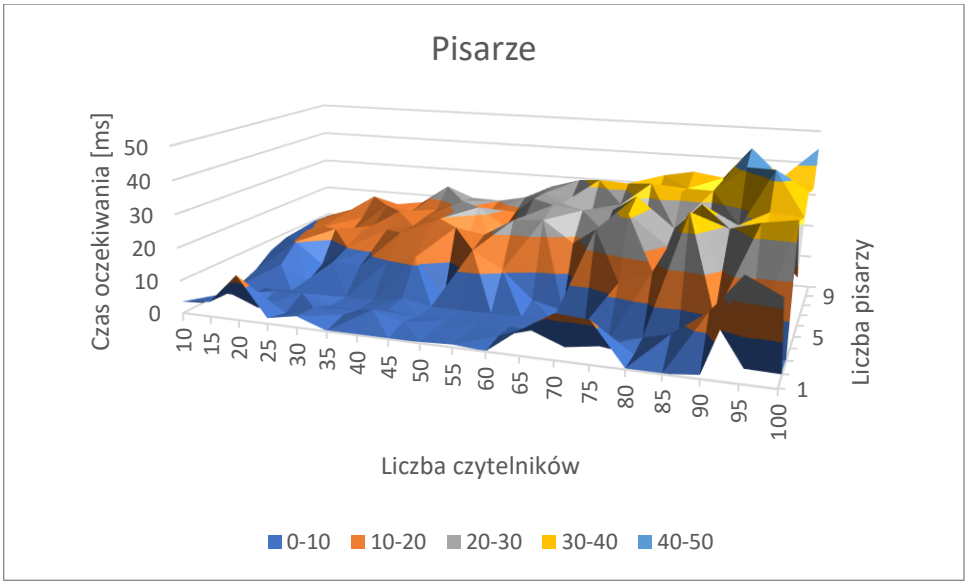
Klasa *ListOneLockThread* wyglądała identycznie – jedyna różnica to oczywiście zmiana typu listy na *ListOneLock*.

3. Wykresy

3.1. Stworzyłam wykresy zależności czasu oczekiwania pisarza (a następnie czytelnika) od liczby czytelników (10-100) i pisarzy (1-10).

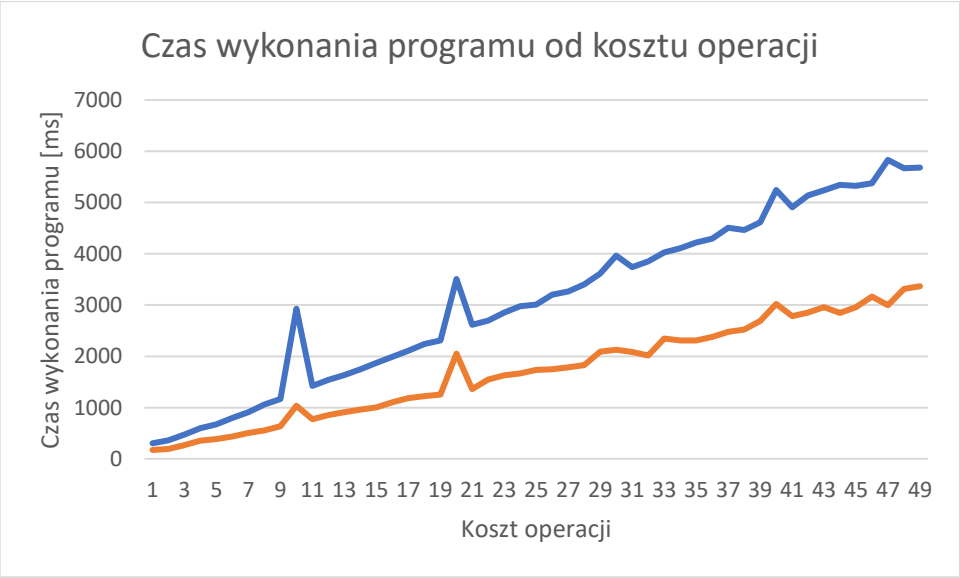


Rysunek 1 Średni czas czekania czytelników



Rysunek 2 Średni czas czekania pisarzy

3.2. Kolejny wykres wykonałam po uruchomieniu drugiego programu. Oś X (koszt operacji) dotyczy zmiennej *sleep_time*.



Rysunek 3 Czas wykonania programu

Pomarańczowa linia odpowiada programowi z blokowaniem drobnoziarnistym, niebieska z listą z jednym zamkiem.

4. Podsumowanie

4.1. Na obu wykresach widoczna jest zależność – im więcej wątków, tym czas oczekiwania zarówno pisarzy jak i czytelników jest większy. Można również zauważyć, że średni czas czekania czytelnika jest mniejszy niż czas czekania pisarza.

4.2. Na wykresie widać, że różnica pomiędzy blokowaniem drobnoziarnistym a blokowaniem z jednym zamkiem jest szczególnie widoczna, kiedy koszt jednej operacji jest duży. W przypadku, kiedy koszt jest mniejszy, wydajność obu mechanizmów zdaje się być podobna z lekką przewagą blokowania drobnoziarnistego. Na wykresie widoczne są również wartości odstające, prawdopodobnie wynika to z działań systemowych spowodowanych uruchamiania wątków w pętli.

5. Bibliografia:

5.1. https://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/Condition.html
5.2. https://home.agh.edu.pl/~funika/tw/lab5/
5.3. Z. Weiss, T. Gruźlewski, *Programowanie współbieżne i rozproszone w przykładach i zadaniach*, WNT, 1993