Laboratorium VI

Ćwiczenie - badanie efektywności

Dominik Marek

13 listopada 2024

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

1.Zadania

Problem czytelników i pisarzy

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.

Blokowanie drobnoziarniste

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:

- zamknij zamek na pierwszym elemencie listy
- zamknij zamek na drugim elemencie
- otwórz zamek na pierwszym elemencie
- zamknij zamek na trzecim elemencie

- otwórz zamek na drugim elemencie
- 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.Rozwiązania

1.

a) Rozwiązania za pomocą semaforów

1 .Klasa Writer

Reprezentuje pisarza jako osobny wątek:

- Metoda run(): W pętli 1000 razy próbuje uzyskać dostęp do zasobu (beginWriting()), zapisuje, a następnie zwalnia zasób (endWriting()).

2. Klasa Reader

Reprezentuje czytelnika jako osobny wątek:

- Metoda run(): W pętli 1000 razy próbuje uzyskać dostęp do zasobu (beginReading()), czyta, a następnie zwalnia zasób (endReading()).

3. Klasa Library

Przechowuje zasób oraz semafory do synchronizacji dostępu między wątkami czytelników i pisarzy:

- readCount: Liczy, ilu czytelników obecnie korzysta z zasobu.
- resource: Semafor kontrolujący dostęp do zasobu, zapewniając wyłączny dostęp dla pisarzy.

- rMutex: Semafor chroniący operacje na readCount.
- serviceQueue: Semafor do kolejkowania wątków, aby zapobiec głodzeniu.

Metody:

- beginReading(): Zwiększa licznik czytelników. Jeśli pierwszy czytelnik zaczyna czytać, blokuje zasób dla pisarzy.
- endReading(): Zmniejsza licznik czytelników. Jeśli ostatni czytelnik kończy, zwalnia zasób.
- beginWriting(): Pisarz blokuje zasób i kolejkę, co zapewnia mu wyłączny dostęp.
- endWriting(): Zwalnia zasób, umożliwiając dostęp kolejnym wątkom.

4. Klasa Main

Główna klasa, która uruchamia program:

- main: Uruchamia program z różnymi liczbami czytelników i pisarzy. Dla każdej konfiguracji tworzy i uruchamia wątki Reader i Writer.
- run(): Tworzy instancję Library, inicjuje wątki czytelników i pisarzy, czeka na ich zakończenie, a następnie wyświetla oraz zapisuje do pliku csv średni czas oczekiwania dla obu grup.

```
public class Writer extends Thread {
    private int nr;
    private Library library;

    public Writer(int nr, Library library) {
        super();
        this.nr = nr;
        this.library = library;
    }

    @Override
    public void run() {
        int i = 0;
        while (i++ < 1000) {
            library.beginWriting();
            library.endWriting();
        }
    }
}
```

```java
public class Reader extends Thread {
    private int nr;
    private Library library;

    public Reader(int nr, Library library) {
        super();
        this.nr = nr;
        this.library = library;
    }

    @Override
    public void run() {
        int i = 0;
        while (i++ < 1000) {
            library.beginReading();
            library.endReading();
        }
    }
}
```

```java
import java.util.concurrent.Semaphore;

public class Library {
    private int readCount = 0;
    private Semaphore resource = new Semaphore(1);
    private Semaphore rMutex = new Semaphore(1);
    private Semaphore serviceQueue = new Semaphore(1);
    private long readerWaitingTime = 0L;
    private long writerWaitingTime = 0L;

    public void beginReading() {
        long startTime = System.nanoTime();
        try {
            serviceQueue.acquire();

            rMutex.acquire();
            readCount++;
            if (readCount == 1) {
                resource.acquire();

            }
            rMutex.release();
            serviceQueue.release();


            setReaderWaitingTime(getReaderWaitingTime() +
(System.nanoTime() - startTime));

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public void endReading() {
        try {
            rMutex.acquire();
            readCount--;
            if (readCount == 0) {
```

```
                resource.release();
            }
            rMutex.release();

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public void beginWriting() {
        long startTime = System.nanoTime();
        try {
            serviceQueue.acquire();
            resource.acquire();
            serviceQueue.release();


            setWriterWaitingTime(getWriterWaitingTime() +
(System.nanoTime() - startTime));

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public void endWriting() {
        resource.release();
    }


    public long getReaderWaitingTime() {
        return readerWaitingTime;
    }

    public void setReaderWaitingTime(long readerWaitingTime) {
        this.readerWaitingTime = readerWaitingTime;
    }

    public long getWriterWaitingTime() {
        return writerWaitingTime;
    }

    public void setWriterWaitingTime(long writerWaitingTime) {
        this.writerWaitingTime = writerWaitingTime;
    }
}
```

```
public class Main {
    private final int readersAmount;
    private final int writersAmount;

    public Main(int readersAmount, int writersAmount) {
        this.readersAmount = readersAmount;
        this.writersAmount = writersAmount;
    }
```

```
    public static void main(String[] args) {
        Main tmp;
        for (int writersAmount = 1; writersAmount <= 10; ++writersAmount) {
            for (int readersAmount = 10; readersAmount <= 100;
readersAmount += 5) {
                tmp = new Main(readersAmount, writersAmount);
                tmp.run();
            }
        }
    }

    public void run() {
        final Library library = new Library();
        Reader[] reader = new Reader[readersAmount];
        Writer[] writer = new Writer[writersAmount];

        for (int i = 0; i < readersAmount; ++i) {
            reader[i] = new Reader(i, library);
            reader[i].start();
        }

        for (int i = 0; i < writersAmount; ++i) {
            writer[i] = new Writer(i, library);
            writer[i].start();
        }

        for (int i = 0; i < readersAmount; ++i) {
            try {
                reader[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        for (int i = 0; i < writersAmount; ++i) {
            try {
                writer[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println(String.format("%d; %d; %.2f; %.2f",
                readersAmount,
                writersAmount,
                library.getReaderWaitingTime() / (double) readersAmount /
1_000_000, // Średni czas w ms
                library.getWriterWaitingTime() / (double) writersAmount /
1_000_000)); // Średni czas w ms
    }
}
```

b) rozwiązanie za pomocą zmiennych warunkowych

1. Klasy Reader i Writer

Obie klasy dziedziczą po Thread i reprezentują odpowiednio czytelników i pisarzy, którzy konkurują o dostęp do zasobu.

- **Reader**:
  - **run()**: W pętli 1000 razy próbuje uzyskać dostęp do zasobu poprzez beginReading(), wykonuje odczyt, a następnie zwalnia zasób przez endReading().
- **Writer**:
  - **run()**: W pętli 1000 razy próbuje uzyskać dostęp do zasobu przez beginWriting(), wykonuje zapis, a następnie zwalnia zasób przez endWriting().

2. Klasa Library

Jest główną klasą zarządzającą synchronizacją dostępu do zasobu między czytelnikami i pisarzami. Używa blokady (Lock) oraz zmiennych warunkowych (Condition) do kontrolowania dostępu i kolejkowania wątków.

- **Zmienne**:
  - **libraryLock**: Blokada (lock) do synchronizacji.
  - **readers**, **writers**: Zmienne warunkowe umożliwiające wątkowi czekanie na swoją kolej do zasobu.
  - **isReading**, **isWriting**: Liczniki aktualnych operacji (czytanie, pisanie).
  - **writerWaiting**, **readerWaiting**: Liczniki wątków czekających na swoją kolej.
  - **readerWaitingTime**, **writerWaitingTime**: Sumaryczne czasy oczekiwania czytelników i pisarzy.
- **Metody**:
  - **beginReading()**: Czytelnik zaczyna czytanie. Jeśli jest aktywny pisarz lub pisarz czeka, czytelnik oczekuje (zwiększa readerWaiting i czeka na readers). Po uzyskaniu dostępu zwiększa licznik isReading i zapisuje czas oczekiwania.
  - **endReading()**: Czytelnik kończy czytanie. Zmniejsza licznik isReading, a jeśli jest ostatnim czytelnikiem, sygnalizuje jednemu czekającemu pisarzowi, że zasób jest dostępny.
  - **beginWriting()**: Pisarz zaczyna pisanie. Jeśli są aktywni czytelnicy lub pisarze, pisarz czeka (zwiększa writerWaiting i czeka na writers). Po uzyskaniu dostępu ustawia isWriting na 1 i zapisuje czas oczekiwania.

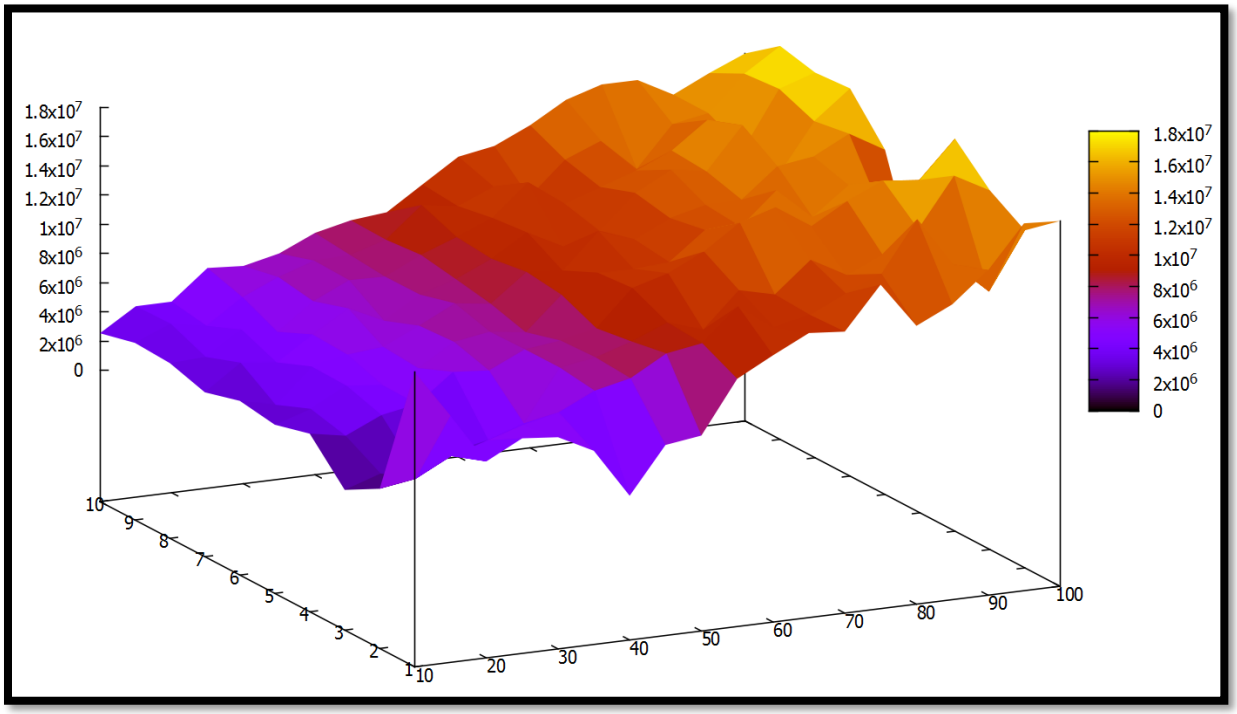
- **endWriting()**: Pisarz kończy pisanie. Zmniejsza isWriting i sprawdza, czy są czekający czytelnicy lub pisarze. Jeśli są czekający pisarze, sygnalizuje jednemu z nich. W przeciwnym razie pozwala wszystkim czekającym czytelnikom na dostęp.

3. Klasa Main

Ustawia i uruchamia wątki czytelników i pisarzy, a także zapisuje wyniki do pliku output.csv.

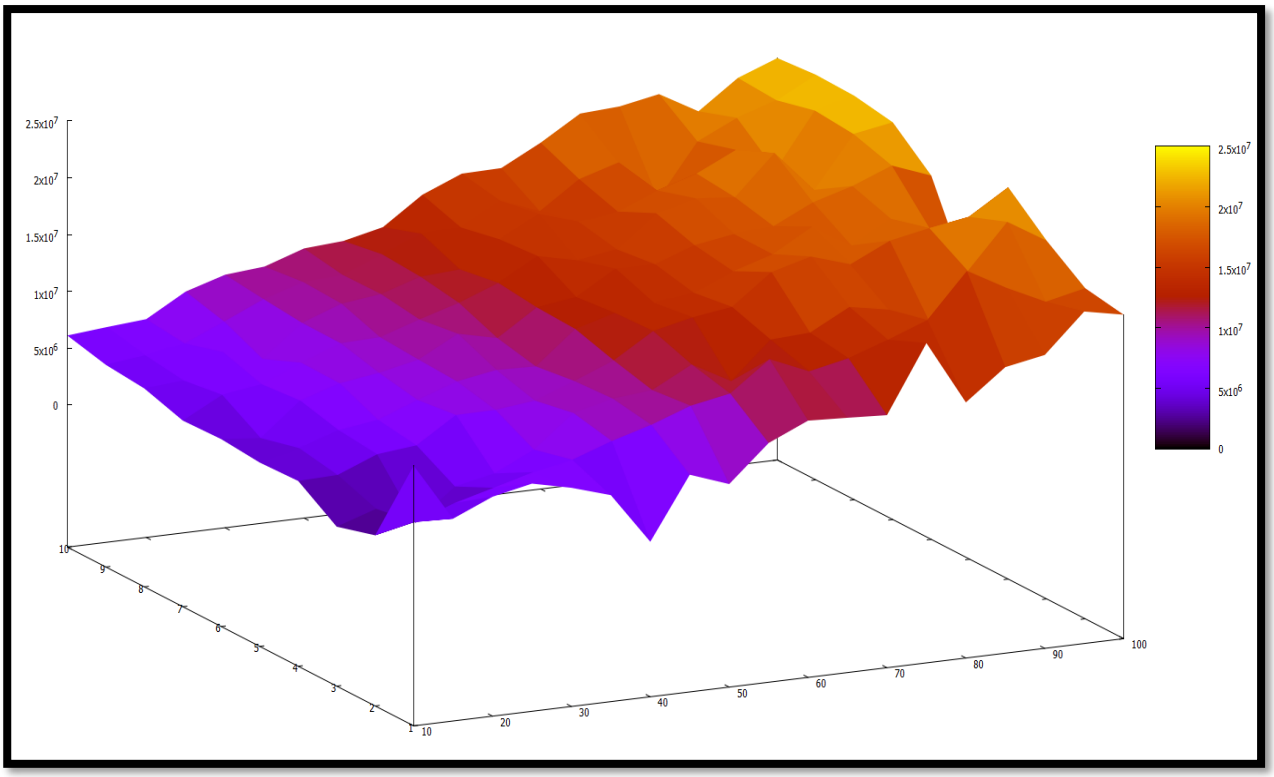
- **Zmienne**:
  - **readersAmount**, **writersAmount**: Liczba czytelników i pisarzy.
- **main()**: Tworzy plik output.csv, w którym zapisuje wyniki dla różnych konfiguracji liczby czytelników i pisarzy. Dla każdej kombinacji wywołuje run().
- **run()**: Tworzy instancję Library, inicjalizuje wątki czytelników i pisarzy, uruchamia je i czeka na ich zakończenie. Następnie oblicza średni czas oczekiwania na dostęp do zasobu i zapisuje wyniki do pliku oraz drukuje je na ekranie.

Średni czas oczekiwania jednego czytelnika (1-10 pisarze, 10-100 czytelnicy)



Średni czas oczekiwania dla jednego czytelnika jest stosunkowo niski i rośnie w sposób umiarkowany wraz ze wzrostem liczby czytelników. Wynika to z preferencji implementacji dla czytelników – nowi czytelnicy mogą uzyskać dostęp do sekcji krytycznej nawet wtedy, gdy inni czytają, o ile nie ma aktywnych ani oczekujących pisarzy. Przy niskiej liczbie pisarzy wpływ ich obecności na czas oczekiwania czytelników jest minimalny, co widać w spłaszczonym charakterze wykresu. Dopiero przy większej liczbie pisarzy i czytelników czas oczekiwania wzrasta bardziej zauważalnie.

Średni czas oczekiwania jednego pisarza (1-10 pisarze, 10-100 czytelnicy)



Czas oczekiwania pisarzy rośnie znacznie szybciej wraz ze wzrostem liczby czytelników, co jest zgodne z preferowaniem czytelników w implementacji. Czytelnicy mają tendencję do "blokowania" dostępu pisarzy do zasobu poprzez ciągłe zgłaszanie odczytów. Wzrost liczby pisarzy także zwiększa czas oczekiwania, ale główny wpływ ma liczba czytelników. Pisarze czekają na całkowite zwolnienie sekcji krytycznej, co staje się coraz trudniejsze przy dużej liczbie czytelników.

Wnioski:

Średni czas oczekiwania pisarzy jest zdecydowanie większy niż czytelników, co potwierdza przewagę, jaką implementacja daje operacjom odczytu. Wynika to z faktu, że czytelnicy są obsługiwani równolegle, a pisarze muszą czekać na całkowite zwolnienie zasobu. W obu przypadkach wzrost liczby czytelników znacząco wpływa na czas oczekiwania, ale dla pisarzy efekt jest bardziej drastyczny, co sugeruje potencjalny problem głodzenia (starvation) dla operacji zapisu w obecnym rozwiązaniu.

```
class Writer extends Thread {
    private int nr;
    private Library library;
    public Writer(int nr, Library library) {
        super();
        this.nr = nr;
        this.library = library;
    }
    @Override
    public void run() {
        int i = 0;
        while (i++ < 1000) {
            library.beginWriting();
            library.endWriting();
        }
    }
}
```

```
class Reader extends Thread {
    private final int nr;
    private Library library;
    public Reader(int nr, Library library) {
        super();
        this.nr = nr;
        this.library = library;
    }
    @Override
    public void run() {
        int i = 0;
        while (i++ < 1000) {
            library.beginReading();
            library.endReading();
        }
    }
}
```

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Library {
    private Lock libraryLock = new ReentrantLock();
    private Condition readers = libraryLock.newCondition();
    private Condition writers = libraryLock.newCondition();
```

```
    private int isReading = 0;
    private int isWriting = 0;
    private int writerWaiting = 0;
    private int readerWaiting = 0;


    private long readerWaitingTime = 0L;
    private long writerWaitingTime = 0L;

    public void beginReading() {
        long startTime = System.nanoTime();
        libraryLock.lock();
        try {
            while (writerWaiting > 0 || isWriting > 0) {
                ++readerWaiting;
                readers.await();
            }

            readerWaitingTime += System.nanoTime() - startTime;
            isReading += 1;
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        } finally {
            libraryLock.unlock();
        }
    }

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

    public void beginWriting() {
        long startTime = System.nanoTime();
        libraryLock.lock();
        try {
            while (isReading + isWriting > 0) {
                ++writerWaiting;
                writers.await();
            }

            writerWaitingTime += System.nanoTime() - startTime;
            isWriting = 1;
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        } finally {
            libraryLock.unlock();
        }
    }

    public void endWriting() {
```

```
        libraryLock.lock();
        try {
            isWriting = 0;
            if (readerWaiting == 0) {
                if (writerWaiting > 0) {
                    --writerWaiting;
                }
                writers.signal();
            } else {
                readerWaiting = 0;
                readers.signalAll();
            }
        } finally {
            libraryLock.unlock();
        }
    }

    public long getReaderWaitingTime() {
        return readerWaitingTime;
    }

    public long getWriterWaitingTime() {
        return writerWaitingTime;
    }
}
```

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class Main {
    private final int readersAmount;
    private final int writersAmount;

    public Main(int readersAmount, int writersAmount) {
        this.readersAmount = readersAmount;
        this.writersAmount = writersAmount;
    }

    public static void main(String[] args) {
        Main tmp;
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter("output.csv"))) {

writer.write("readersAmount,writersAmount,readerAvgWaitTime,writerAvgWaitTi
me");
            writer.newLine();

            for (int writersAmount = 1; writersAmount <= 10;
++writersAmount) {
                for (int readersAmount = 10; readersAmount <= 100;
readersAmount += 5) {
                    tmp = new Main(readersAmount, writersAmount);
                    tmp.run(writer);                    }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
    public void run(BufferedWriter writer) {
        final Library library = new Library();
        Reader[] reader = new Reader[readersAmount];
        Writer[] writerThread = new Writer[writersAmount];

        for (int i = 0; i < readersAmount; ++i) {
            reader[i] = new Reader(i, library);
            reader[i].start();
        }


        for (int i = 0; i < writersAmount; ++i) {
            writerThread[i] = new Writer(i, library);
            writerThread[i].start();
        }

        for (int i = 0; i < readersAmount; ++i) {
            try {
                reader[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        for (int i = 0; i < writersAmount; ++i) {
            try {
                writerThread[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        double readerAvgWaitTime = library.getReaderWaitingTime() /
(double) readersAmount / 1_000_000; // Czas w ms
        double writerAvgWaitTime = library.getWriterWaitingTime() /
(double) writersAmount / 1_000_000; // Czas w ms

        System.out.println(String.format("%d; %d; %.2f; %.2f",
                readersAmount,
                writersAmount,
                readerAvgWaitTime,
                writerAvgWaitTime));


        try {
            writer.write(String.format("%d,%d,%.2f,%.2f",
                    readersAmount,
                    writersAmount,
                    readerAvgWaitTime,
                    writerAvgWaitTime));
            writer.newLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2.

1. Klasa Node

Reprezentuje pojedynczy element listy.

Pola:

- **Object value**: Przechowuje wartość węzła. Może to być dowolny obiekt.
- **Node next**: Wskaźnik na następny węzeł w liście. Domyślnie null, co oznacza, że węzeł jest na końcu listy.
- **Lock lock:** Dla każdego węzła tworzona jest oddzielna instancja ReentrantLock. Pozwala to na synchronizację operacji na poziomie pojedynczych węzłów w przypadku implementacji z blokowaniem drobnoziarnistym.

2. Klasa CoarseGrainedLockingList

Implementuje listę z blokowaniem gruboziarnistym, gdzie jedna globalna blokada synchronizuje operacje na całej liście.

Pola:

- **Node head:** Wskaźnik na początek listy. Początkowo wskazuje na węzeł pusty (null wartość).
- **Lock listLock:** Instancja ReentrantLock, która jest jedyną blokadą synchronizującą wszystkie operacje (add, remove, contains).

Metody:

1. **add(Object o):**
    - Dodaje nowy węzeł na końcu listy.
    - Synchronizacja: Cała lista jest blokowana podczas operacji, co uniemożliwia równoczesne operacje na liście.

    Przebieg:

    - Blokuje całą listę (listLock.lock()).
    - Iteruje przez listę, aż znajdzie ostatni węzeł.
    - Dodaje nowy węzeł jako current.next.
    - Odblokowuje listę.
2. **remove(Object o):**
    - Usuwa pierwszy węzeł, którego wartość równa się o.
    - Synchronizacja: Blokuje całą listę.

Przebieg:

- Blokuje listę.
- Iteruje przez węzły, sprawdzając wartość każdego z nich.
- Jeśli znajdzie węzeł z wartością o, usuwa go (aktualizując wskaźnik next poprzedniego węzła).
- Odblokowuje listę.

3. **contains(Object o):**
    - Sprawdza, czy lista zawiera węzeł z wartością o.
    - Synchronizacja: Blokuje całą listę.

    Przebieg:

    - **Blokuje listę.**
    - **Iteruje przez listę, sprawdzając wartość każdego węzła.**
    - **Zwraca true, jeśli znajdzie węzeł z wartością o; w przeciwnym razie zwraca false.**
    - **Odblokowuje listę.**

3. Klasa FineGrainedLockingList

Implementuje listę z blokowaniem drobnoziarnistym, co pozwala na większą równoległość operacji.

Pola:

- **Node head: Wskaźnik na początek listy. Początkowo wskazuje na pusty węzeł (null wartość).**

Metody:

1. **add(Object o):**
    - Dodaje nowy węzeł na końcu listy.
    - Synchronizacja: Ta metoda w obecnej wersji nie używa blokad, ale można ją rozszerzyć o drobnoziarnistą synchronizację.

    Przebieg:

    - Iteruje przez listę, aż znajdzie ostatni węzeł.
    - Dodaje nowy węzeł jako current.next.

2. **remove(Object o):**
    - Usuwa pierwszy węzeł, którego wartość równa się o.
    - Synchronizacja: Blokuje aktualny i następny węzeł w czasie iteracji, zapewniając spójność operacji.

Przebieg:

- Przechodzi przez listę, blokując bieżący (current) i następny (nextNode) węzeł.
- Jeśli znajdzie węzeł z wartością o, usuwa go, aktualizując wskaźnik next poprzedniego węzła (prev).
- Odblokowuje węzły po zakończeniu operacji.

3. **contains(Object o):**
   - Sprawdza, czy lista zawiera węzeł z wartością o.
   - Synchronizacja: Blokuje każdy węzeł podczas sprawdzania, aby zapewnić spójność.

   Przebieg:

   - **Iteruje przez listę, blokując bieżący węzeł (current).**
   - **Jeśli znajdzie węzeł z wartością o, zwraca true.**
   - **Odblokowuje węzeł przed przejściem do następnego.**

4. Klasa PerformanceTest

Testuje wydajność obu implementacji dla różnych operacji (add, contains, remove) oraz różnych liczb wątków.

Pola:

- **FineGrainedLockingList fineGrainedList:** Instancja listy z blokowaniem drobnoziarnistym.
- **CoarseGrainedLockingList coarseGrainedList:** Instancja listy z blokowaniem gruboziarnistym.
- **int[] threadCounts:** Tablica z liczbami wątków testowych.

Metody testujące:

1. **testAddPerformance:**
   - Każdy wątek dodaje 1000 elementów do listy.
   - Mierzy czas wykonywania operacji na obu implementacjach.
2. **testContainsPerformance:**
   - Każdy wątek sprawdza obecność 1000 elementów w liście.
   - Mierzy czas wykonywania operacji.
3. **testRemovePerformance:**
   - Każdy wątek próbuje usunąć 1000 elementów z listy.

- Mierzy czas wykonywania operacji.

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Node {
    Object value;
    Node next;
    Lock lock;

    public Node(Object value) {
        this.value = value;
        this.lock = new ReentrantLock();
        this.next = null;
    }
}
```

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class CoarseGrainedLockingList {
    private Node head;
    private final Lock listLock;

    public CoarseGrainedLockingList() {
        head = new Node(null);
        listLock = new ReentrantLock();
    }

    public boolean contains(Object o) {
        listLock.lock();
        try {
            Node current = head.next;
            while (current != null) {
                if (current.value.equals(o)) {
                    return true;
                }
                current = current.next;
            }
            return false;
        } finally {
            listLock.unlock();
        }
    }

    public boolean remove(Object o) {
        listLock.lock();
        try {
            Node current = head;
            while (current.next != null) {
                Node nextNode = current.next;
                if (nextNode.value.equals(o)) {
                    current.next = nextNode.next;
                    return true;
```

```
                }
                current = nextNode;
            }
            return false;
        } finally {
            listLock.unlock();
        }
    }


    public void add(Object o) {
        listLock.lock();
        try {
            Node newNode = new Node(o);
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        } finally {
            listLock.unlock();
        }
    }
}
```

```
public class FineGrainedLockingList {
    private Node head;

    public FineGrainedLockingList() {
        head = new Node(null);    }

        public boolean contains(Object o) {
        Node current = head.next;

        while (current != null) {
            current.lock.lock();            try {
                if (current.value.equals(o)) {
                    return true;
                }
            } finally {
                current.lock.unlock();
            }
            current = current.next;
        }
        return false;
    }

        public boolean remove(Object o) {
        Node current = head;
        Node prev = null;

        while (current.next != null) {
            Node nextNode = current.next;
            nextNode.lock.lock();
 try {
                if (nextNode.value.equals(o)) {
                                        if (prev != null) {
                        prev.next = nextNode.next;
```

```
                    }
                    return true;
                }
                prev = current;
                current = nextNode;
            } finally {
                nextNode.lock.unlock();            }
        }
        return false;
    }

        public void add(Object o) {
        Node newNode = new Node(o);
        Node current = head;

        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }
}
```

```
package zad3;

public class PerformanceTest {
    public static void main(String[] args) {
        FineGrainedLockingList fineGrainedList = new
FineGrainedLockingList();
        CoarseGrainedLockingList coarseGrainedList = new
CoarseGrainedLockingList();

        int[] threadCounts = {1, 5, 10, 25, 50, 100};

                for (int threadCount : threadCounts) {
            System.out.println("Testing with " + threadCount + " threads");

                        testAddPerformance(fineGrainedList,
coarseGrainedList, threadCount);

                    testContainsPerformance(fineGrainedList,
coarseGrainedList, threadCount);

            testRemovePerformance(fineGrainedList, coarseGrainedList,
threadCount);

            System.out.println("-----------------------------------");
        }
    }


    private static void testAddPerformance(FineGrainedLockingList
fineGrainedList, CoarseGrainedLockingList coarseGrainedList, int
threadCount) {
        Runnable addTaskFine = () -> {
            for (int i = 0; i < 1000; i++) {
                fineGrainedList.add(i);
            }
        };
```

```
        Runnable addTaskCoarse = () -> {
            for (int i = 0; i < 1000; i++) {
                coarseGrainedList.add(i);
            }
        };

        long startTime = System.nanoTime();
        Thread[] threads = new Thread[threadCount];
        for (int i = 0; i < threadCount; i++) {
            threads[i] = new Thread(addTaskFine);
            threads[i].start();
        }

        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        long endTime = System.nanoTime();
        double durationInSeconds = (endTime - startTime) / 1_000_000.0;
        System.out.println("Add operation time (Fine-grained): " +
durationInSeconds + " milliseconds");


        startTime = System.nanoTime();
        for (int i = 0; i < threadCount; i++) {
            threads[i] = new Thread(addTaskCoarse);
            threads[i].start();
        }

        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        endTime = System.nanoTime();
        durationInSeconds = (endTime - startTime) / 1_000_000.0;
        System.out.println("Add operation time (Coarse-grained): " +
durationInSeconds + " milliseconds");
    }

    private static void testContainsPerformance(FineGrainedLockingList
fineGrainedList, CoarseGrainedLockingList coarseGrainedList, int
threadCount) {
        Runnable containsTaskFine = () -> {
            for (int i = 0; i < 1000; i++) {
                fineGrainedList.contains(i);
            }
        };

        Runnable containsTaskCoarse = () -> {
            for (int i = 0; i < 1000; i++) {
                coarseGrainedList.contains(i);
            }
        };
```

```
        long startTime = System.nanoTime();
        Thread[] threads = new Thread[threadCount];
        for (int i = 0; i < threadCount; i++) {
            threads[i] = new Thread(containsTaskFine);
            threads[i].start();
        }

        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        long endTime = System.nanoTime();
        double durationInSeconds = (endTime - startTime) / 1_000_000.0;
        System.out.println("Contains operation time (Fine-grained): " +
durationInSeconds + " milliseconds");

        startTime = System.nanoTime();
        for (int i = 0; i < threadCount; i++) {
            threads[i] = new Thread(containsTaskCoarse);
            threads[i].start();
        }

        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        endTime = System.nanoTime();
        durationInSeconds = (endTime - startTime) / 1_000_000.0;
        System.out.println("Contains operation time (Coarse-grained): " +
durationInSeconds + " milliseconds");
    }


    private static void testRemovePerformance(FineGrainedLockingList
fineGrainedList, CoarseGrainedLockingList coarseGrainedList, int
threadCount) {
        Runnable removeTaskFine = () -> {
            for (int i = 0; i < 1000; i++) {
                fineGrainedList.remove(i);
            }
        };

        Runnable removeTaskCoarse = () -> {
            for (int i = 0; i < 1000; i++) {
                coarseGrainedList.remove(i);
            }
        };


        long startTime = System.nanoTime();
        Thread[] threads = new Thread[threadCount];
        for (int i = 0; i < threadCount; i++) {
            threads[i] = new Thread(removeTaskFine);
            threads[i].start();
```

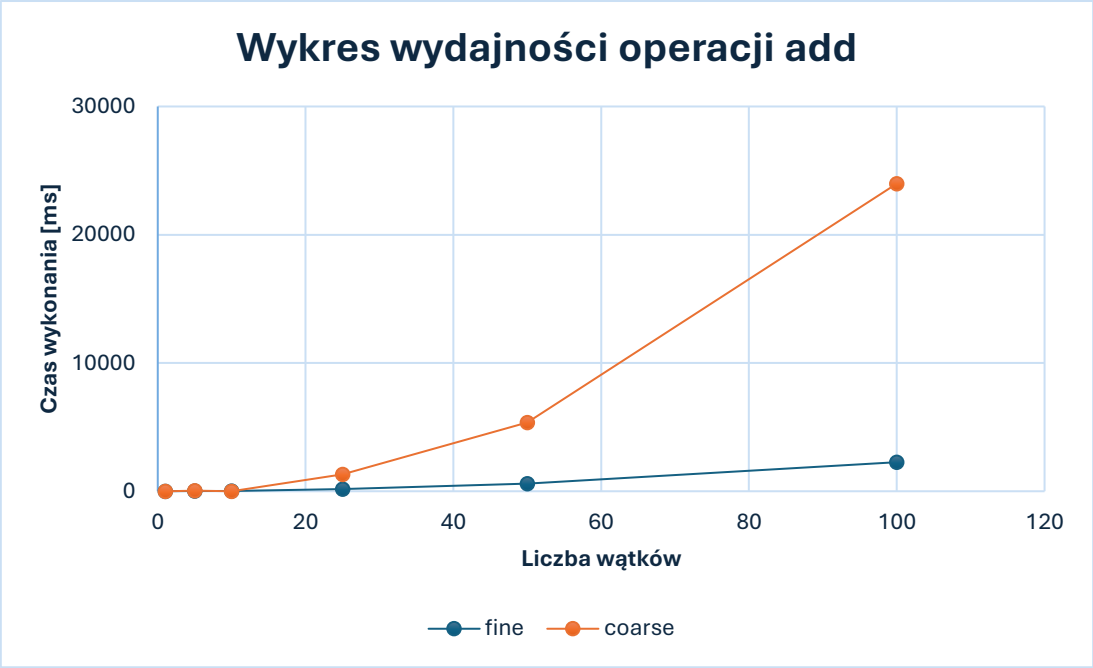
```
        }

        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        long endTime = System.nanoTime();
        double durationInSeconds = (endTime - startTime) / 1_000_000.0;
        System.out.println("Remove operation time (Fine-grained): " +
durationInSeconds + " milliseconds");


        startTime = System.nanoTime();
        for (int i = 0; i < threadCount; i++) {
            threads[i] = new Thread(removeTaskCoarse);
            threads[i].start();
        }

        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        endTime = System.nanoTime();
        durationInSeconds = (endTime - startTime) / 1_000_000.0;
        System.out.println("Remove operation time (Coarse-grained): " +
durationInSeconds + " milliseconds");
    }
}
```

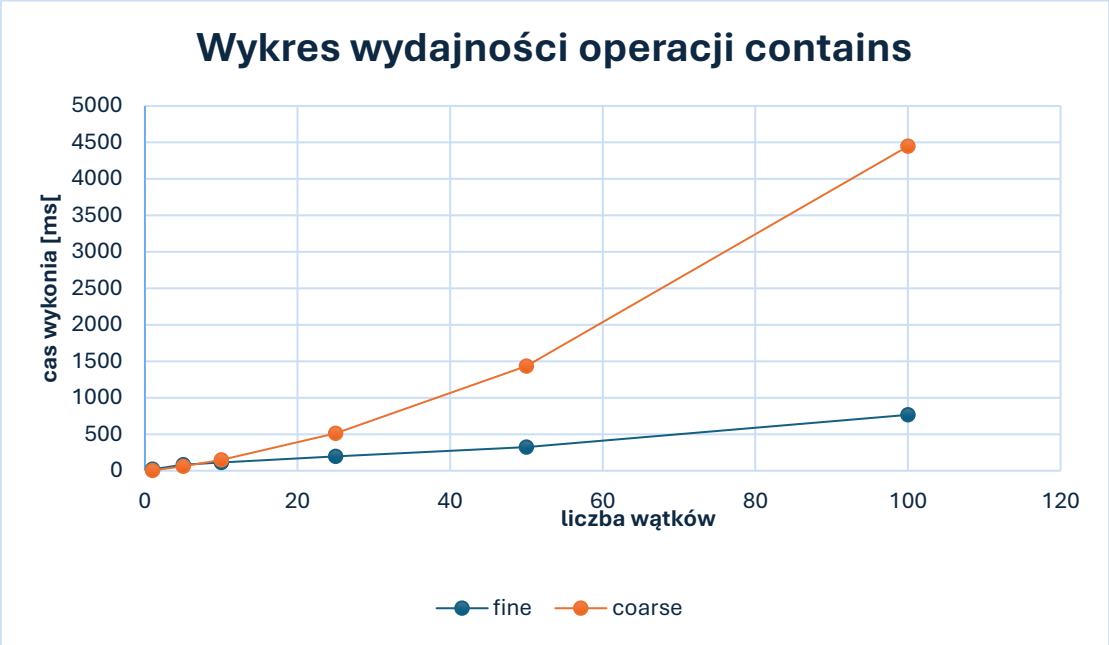
Czas tysiąckrotnego wykonania operacji add		
Liczba wątków	**Blokowanie drobnoziarniste**	**Blokowanie gruboziarniste**
1	3,6906	3,2707
5	25,8842	56,8634
10	22,8824	210, 1094
25	160,5242	1319,9039
50	590,1050	5360,5234
100	2266,2272	23982,3484



Przy 1 wątku, blokowanie drobnoziarniste jest nieco szybsze od gruboziarnistego, co wynika z braku konkurencji między wątkami. Oba podejścia działają w zasadzie sekwencyjnie. Wraz ze wzrostem liczby wątków, czas dla gruboziarnistego blokowania dramatycznie rośnie, szczególnie od 10 wątków wzwyż. Wynika to z faktu, że jeden globalny lock uniemożliwia wykonywanie równoległych operacji. Blokowanie drobnoziarniste skaluje się lepiej, ale również wykazuje wzrost czasu przy większej liczbie wątków, co jest efektem lokalnych blokad na węzłach i zarządzania dostępem do struktury.

Wniosek: Drobnoziarniste blokowanie jest znacznie bardziej efektywne przy większej liczbie wątków, co czyni je lepszym wyborem dla równoległych operacji dodawania.

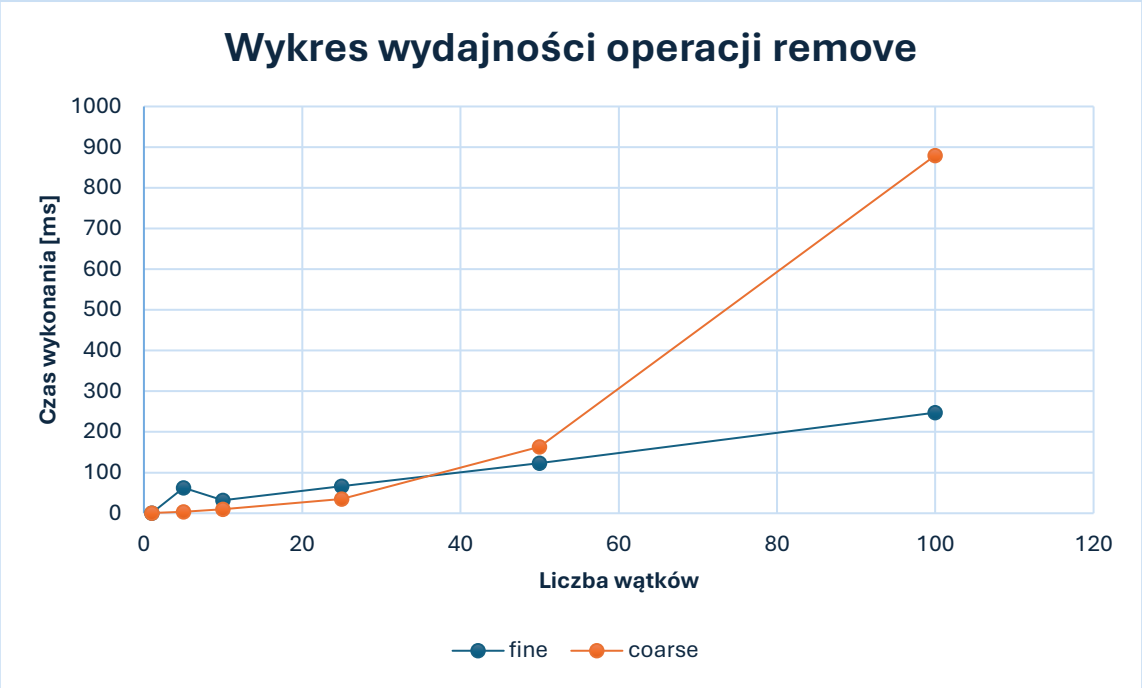
Czas tysiąckrotnego wykonania operacji contains		
Liczba wątków	**Blokowanie drobnoziarniste**	**Blokowanie gruboziarniste**
1	19,936	4,7358
5	84,594	59,2221
10	112,1681	147,2278
25	195,5780	514,7009
50	323,1023	1434,4220
100	765,9621	4446,8880



Przy małej liczbie wątków(1-10) blokowanie gruboziarniste jest szybsze, ponieważ odblokowanie i ponowne blokowanie kolejnych węzłów w drobnoziarnistym podejściu wprowadza narzut czasowy. Gdy liczba wątków rośnie, blokowanie gruboziarniste szybko traci wydajność. Wysoka konkurencja o jedną globalną blokadę drastycznie wydłuża czas operacji. Drobnoziarniste blokowanie radzi sobie znacznie lepiej, ale narzut związany z zarządzaniem lokalnymi blokadami wciąż powoduje wzrost czasu przy większej liczbie wątków.

Wniosek: Blokowanie drobnoziarniste lepiej skaluje się dla operacji odczytu, ale gruboziarniste może być bardziej efektywne w środowisku o małej liczbie wątków.

Czas tysiąckrotnego wykonania operacji remove		
Liczba wątków	Blokowanie drobnoziarniste	Blokowanie gruboziarniste
1	0,6697	0,6818
5	62,8134	3,8938
10	32,0482	9,5323
25	66,6896	35,2172
50	123,0808	163,1675
100	247,3665	879,7197



Dla liczby wątków nie przekraczającej 25, gruboziarniste blokowanie jest szybsze, ponieważ unika narzutu zarządzania wieloma blokadami. W przypadku drobnoziarnistego, konieczność blokowania i odblokowywania każdego węzła powoduje dodatkowy czas. Dla większej liczby wątków, blokowanie drobnoziarniste radzi sobie dużo lepiej.

W przypadku gruboziarnistego, dramatyczny wzrost czasu operacji przy 100 wątkach wynika z kolejkowania wątków próbujących uzyskać dostęp do globalnej blokady.

Podsumowanie: Blokowanie drobnoziarniste jest lepszym wyborem w środowiskach wielowątkowych, gdzie równoległość operacji jest kluczowa, natomiast gruboziarniste może być prostszym rozwiązaniem w mniej obciążonych aplikacjach.

3.Bibliografia

- https://www.dz5.pl/ti/java/java_skladnia.pdf ~Jacek Rumiński, Język Java – rozdział o wątkach
- http://brinch-hansen.net/papers/1999b.pdf ~ Per Brinch Hansen
- https://www.artima.com/insidejvm/ed2/threadsynch.html ~Bill Venners, Inside the Java Virtual Machine - Charter 20 Thread Synchronization
- https://docs.oracle.com/en/java/javase/19/index.html
- https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem
- https://martinfowler.com/eaaCatalog/coarseGrainedLock.html
- https://digitalscholarship.unlv.edu/cgi/viewcontent.cgi?article=4733&context=thesesdissertations