

Teoria Współbieżności - Sprawozdanie 6

Maciej Brzeżawski

Listopad 2024

1 Treść zadania

1.1 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.

1.2 Blokowanie drobnoziarniste

- Proszę zaimplementować listę, w której każdy węzeł składa się z wartości typu Object, referencji do następnego węzła oraz zamka (lock).
- Proszę zastosować metodę drobnoziarnistego blokowania do następujących metod listy:
 1. boolean contains(Object o); //czy lista zawiera element o
 2. boolean remove(Object o); //usuwa pierwsze wystąpienie elementu o
 3. boolean add(Object o); //dodaje element o na końcu listy
- 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 Problem czytelników i pisarzy

2.1 Implementacja programu

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

class CzytelnicyPisarze {
    private int liczbaCzytelnikow = 0;
    private boolean pisarzPisze = false;
    private final Lock lock = new ReentrantLock();
    private final Condition czytelnicyCondition = lock.newCondition();
    private final Condition pisarzeCondition = lock.newCondition();
    private int oczekujacyPisarze = 0;

    private final Semaphore accessSemaphore = new Semaphore(1);

    public void zaczynaCzytac(int id) throws InterruptedException {
        lock.lock();
        try {
            while (pisarzPisze || oczekujacyPisarze > 0) {
                czytelnicyCondition.await();
            }
            liczbaCzytelnikow++;
        } finally {
            lock.unlock();
        }
    }

    public void konczyCzytac(int id) {
        lock.lock();
        try {
            liczbaCzytelnikow--;
            if (liczbaCzytelnikow == 0) {
                pisarzeCondition.signal();
            }
        } finally {
            lock.unlock();
        }
    }

    public void zaczynaPisac(int id) throws InterruptedException {
        accessSemaphore.acquire(); // Semafor blokuje dostęp dla innych pisarzy
```

```

        lock.lock();
        try {
            oczekujacyPisarze++;
            while (pisarzPisze || liczbaCzytelnikow > 0) {
                pisarzeCondition.await();
            }
            oczekujacyPisarze--;
            pisarzPisze = true;
        } finally {
            lock.unlock();
        }
    }

    public void konczyPisac(int id) {
        lock.lock();
        try {
            pisarzPisze = false;
            if (oczekujacyPisarze > 0) {
                pisarzeCondition.signal();
            } else {
                czytelnicyCondition.signalAll();
            }
        } finally {
            lock.unlock();
        }
        accessSemaphore.release(); // Zwolnienie semafora po zakończeniu pisania
    }
}

// Klasa reprezentująca czytelników
class Czytelnik extends Thread {
    private final CzytelnicyPisarze zasoby;
    private final int id;

    public Czytelnik(CzytelnicyPisarze zasoby, int id) {
        this.zasoby = zasoby;
        this.id = id;
    }

    @Override
    public void run() {
        try {
            zasoby.zaczynaCzytac(id);
            Thread.sleep((int) (Math.random() * 50)); // Symulacja odczytu
            zasoby.konczyCzytac(id);
        }
    }
}

```

```

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

// Klasa reprezentująca pisarzy
class Pisarz extends Thread {
    private final CzytelnicyPisarze zasoby;
    private final int id;

    public Pisarz(CzytelnicyPisarze zasoby, int id) {
        this.zasoby = zasoby;
        this.id = id;
    }

    @Override
    public void run() {
        try {
            zasoby.zaczynaPisac(id);
            Thread.sleep((int) (Math.random() * 50)); // Symulacja zapisu
            zasoby.konczyPisac(id);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        int[] liczbyCzytelnikow = {10, 20, 50, 100};
        int[] liczbyPisarzy = {1, 2, 5, 10};

        System.out.println("Pomiar czasu wykonania dla różnych konfiguracji:");
        System.out.println("Liczba Czytelników | Liczba Pisarzy | Czas (ms)");

        for (int liczbaCzytelnikow : liczbyCzytelnikow) {
            for (int liczbaPisarzy : liczbyPisarzy) {
                long czasWykonania = zmierzCzasWykonania(liczbaCzytelnikow,
                    liczbaPisarzy);
                System.out.printf("%17d | %13d | %10d ms%n", liczbaCzytelnikow,
                    liczbaPisarzy, czasWykonania);
            }
        }
    }
}

```

```

private static long zmierzCzasWykonania(int liczbaCzytelnikow, int liczbaPisarzy) {
    CzytelnicyPisarze zasoby = new CzytelnicyPisarze();
    Thread[] czytelnicy = new Thread[liczbaCzytelnikow];
    Thread[] pisarze = new Thread[liczbaPisarzy];

    // Tworzenie i uruchamianie wątków czytelników
    for (int i = 0; i < liczbaCzytelnikow; i++) {
        czytelnicy[i] = new Czytelnik(zasoby, i);
    }

    // Tworzenie i uruchamianie wątków pisarzy
    for (int i = 0; i < liczbaPisarzy; i++) {
        pisarze[i] = new Pisarz(zasoby, i);
    }

    long startTime = System.currentTimeMillis();

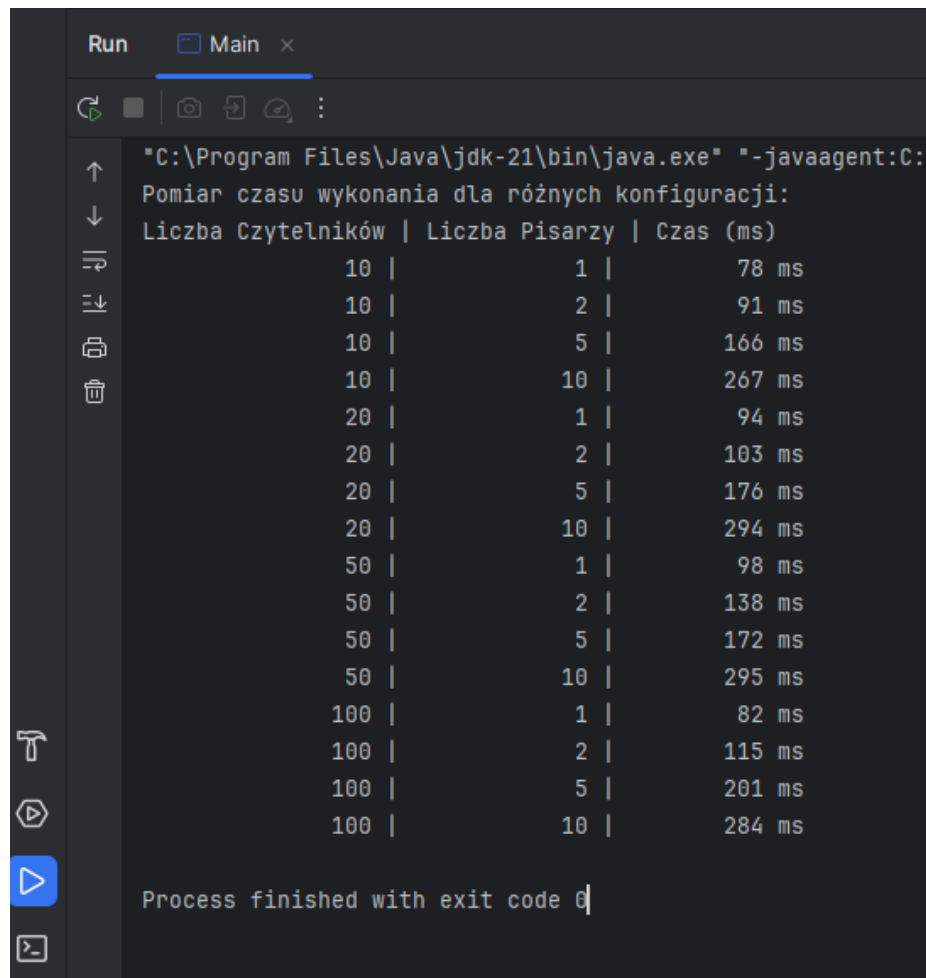
    for (Thread czytelnik : czytelnicy) czytelnik.start();
    for (Thread pisarz : pisarze) pisarz.start();

    try {
        for (Thread czytelnik : czytelnicy) czytelnik.join();
        for (Thread pisarz : pisarze) pisarz.join();
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }

    long endTime = System.currentTimeMillis();
    return endTime - startTime;
}
}

```

2.2 Wyniki pomiarów

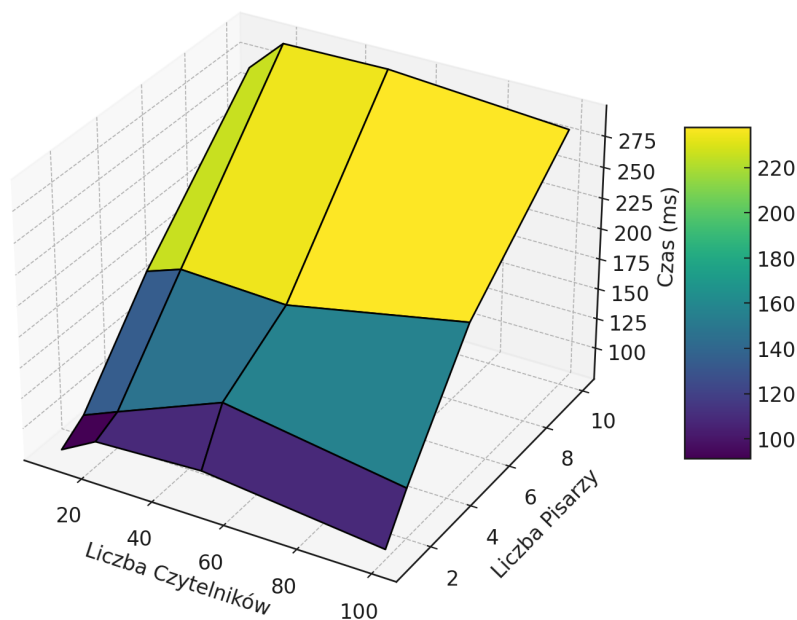


The screenshot shows a Java IDE's Run console window. The title bar says "Run" and "Main". The console output shows the command used to run the program: `"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\Java\jdk-21\bin\javaagent.jar" -jar C:\Program Files\Java\jdk-21\bin\java.exe`. Below the command, the text "Pomiar czasu wykonania dla różnych konfiguracji:" is displayed. This is followed by a table with three columns: "Liczba Czytelników" (Number of Readers), "Liczba Pisarzy" (Number of Writers), and "Czas (ms)" (Time in ms). The table contains 16 rows of data. At the bottom of the console, it says "Process finished with exit code 0".

Liczba Czytelników	Liczba Pisarzy	Czas (ms)
10	1	78 ms
10	2	91 ms
10	5	166 ms
10	10	267 ms
20	1	94 ms
20	2	103 ms
20	5	176 ms
20	10	294 ms
50	1	98 ms
50	2	138 ms
50	5	172 ms
50	10	295 ms
100	1	82 ms
100	2	115 ms
100	5	201 ms
100	10	284 ms

Rysunek 1: Pomiar czasu wykonania dla różnych konfiguracji

Wykres 3D: Czas wykonania w zależności od liczby wątków



Rysunek 2: Czas wykonania w zależności od liczby wątków

2.3 Wnioski

- Zwiększenie liczby pisarzy powoduje znaczny wzrost czasu wykonania. Im więcej pisarzy, tym częstsze są blokady dostępu do zasobów, ponieważ tylko jeden pisarz może mieć dostęp w danym momencie.
- Wpływ liczby czytelników na czas wykonania jest mniej znaczący niż liczby pisarzy. Czytelnicy mogą współdzielić zasoby, co minimalizuje opóźnienia, chyba że oczekują na zakończenie operacji pisania.
- Największy czas wykonania odnotowano przy 10 pisarzach i 50-100 czytelnikach. W tej konfiguracji występuje duża konkurencja o zasoby, co skutkuje większą liczbą blokad i opóźnień.

3 Blokowanie drobnoziarniste

3.1 Implementacja listy z drobnoziarnistym blokowaniem

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

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

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

class FineGrainedList {
    private final Node head;

    public FineGrainedList() {
        head = new Node(null); // wartownik (sentinel)
    }

    // Metoda dodająca element na końcu listy
    public void add(Object o) {
        Node current = head;
        current.lock.lock();
        try {
            while (current.next != null) {
                Node next = current.next;
                next.lock.lock();
                current.lock.unlock();
                current = next;
            }
            current.next = new Node(o);
        } finally {
            current.lock.unlock();
        }
    }

    // Metoda sprawdzająca, czy lista zawiera element
    public boolean contains(Object o) {
        Node current = head;
```



```

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

// Metoda usuwająca pierwsze wystąpienie elementu
public boolean remove(Object o) {
    Node current = head;
    current.lock.lock();
    try {
        while (current.next != null) {
            Node next = current.next;
            next.lock.lock();
            try {
                if (next.value.equals(o)) {
                    current.next = next.next;
                    return true;
                }
            }
            current = next;
        } finally {
            next.lock.unlock();
        }
    }
    return false;
} finally {
    current.lock.unlock();
}
}
}

```

3.2 Implementacja listy z globalnym blokowaniem

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

class CoarseGrainedList {
    private final Node head;
    private final Lock lock;

    public CoarseGrainedList() {
        head = new Node(null); // wartownik (sentinel)
        lock = new ReentrantLock();
    }

    // Metoda dodająca element na końcu listy
    public void add(Object o) {
        lock.lock();
        try {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = new Node(o);
        } finally {
            lock.unlock();
        }
    }

    // Metoda sprawdzająca, czy lista zawiera element
    public boolean contains(Object o) {
        lock.lock();
        try {
            Node current = head;
            while (current != null) {
                if (current.value != null && current.value.equals(o)) {
                    return true;
                }
                current = current.next;
            }
            return false;
        } finally {
            lock.unlock();
        }
    }

    // Metoda usuwająca pierwsze wystąpienie elementu
}
```

```

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

```

3.3 Pomiary wydajności

```

public class Main {
    public static void main(String[] args) {
        int[] kosztyOperacji = {10, 50, 100, 200, 500, 1000};

        System.out.println("Porównanie wydajności:");
        System.out.println("Koszt Operacji |  
Drobndziarniste Blokowanie (ms) | Blokowanie Globalne (ms)");

        for (int kosztOperacji : kosztyOperacji) {
            FineGrainedList fineList = new FineGrainedList();
            CoarseGrainedList coarseList = new CoarseGrainedList();

            // Testowanie listy z drobndziarnistym blokowaniem
            long startFine = System.currentTimeMillis();
            testList(fineList, kosztOperacji);
            long endFine = System.currentTimeMillis();
            long fineTime = endFine - startFine;

            // Testowanie listy z globalnym blokowaniem
            long startCoarse = System.currentTimeMillis();
            testList(coarseList, kosztOperacji);
            long endCoarse = System.currentTimeMillis();
            long coarseTime = endCoarse - startCoarse;

            // Wypisanie wyników

```

```

        System.out.printf("%15d | %30d | %25d%n", kosztOperacji, fineTime, coarseTime);
    }
}

// Metoda do testowania listy
private static void testList(Object list, int kosztOperacji) {
    Runnable addTask = () -> {
        for (int i = 0; i < 100; i++) {
            if (list instanceof FineGrainedList) {
                ((FineGrainedList) list).add(i);
            } else {
                ((CoarseGrainedList) list).add(i);
            }
            simulateHeavyOperation(kosztOperacji);
        }
    };

    Thread[] threads = new Thread[10];
    for (int i = 0; i < threads.length; i++) {
        threads[i] = new Thread(addTask);
    }

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

    // Czekanie na zakończenie wszystkich wątków
    for (Thread thread : threads) {
        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

// Symulacja kosztownej operacji
private static void simulateHeavyOperation(int koszt) {
    try {
        Thread.sleep(koszt);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
}

```

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\I
Porównanie wydajności:
Koszt Operacji | Drobndziarniste Blokowanie (ms) | Blokowanie Globalne (ms)
10 | 1598 | 1584
50 | 6306 | 6294
100 | 10970 | 10997
200 | 20429 | 20372
500 | 50812 | 50675
1000 | 100851 | 100803

Process finished with exit code 0
```

Rysunek 3: Porównanie wydajności

3.4 Wnioski

3.4.1 Interpretacja wyników

- **Drobndziarniste blokowanie** nie przynosi znaczącej przewagi nad globalnym blokowaniem, gdy koszt operacji jest wysoki (powyżej 50 ms). W takich przypadkach większość czasu spędzana jest na samej operacji (symulacja kosztownego zadania `Thread.sleep()`), a czas blokowania dostępu do struktury staje się mniej istotny.
- Przy niskim koszcie operacji (10 ms), różnice w wydajności również są znikome. Oznacza to, że narzut związany z używaniem wielu zamków (dla każdego węzła) w drobndziarnistym blokowaniu praktycznie równoważy się z korzyściami wynikającymi z możliwości równoległego dostępu do różnych części listy.

3.4.2 Wnioski praktyczne

- Drobndziarniste blokowanie jest korzystne w sytuacjach, gdy:
 1. Koszt operacji na elementach jest niski lub średni.
 2. Istnieje duża konkurencja pomiędzy wątkami o dostęp do różnych części struktury.
- Globalne blokowanie okazuje się bardziej wydajne i prostsze w implementacji, gdy koszt operacji jest bardzo wysoki. W takich przypadkach blokady stanowią niewielką część całkowitego czasu wykonywania operacji.

4 Bibliografia

Abraham Silberschatz, Peter B. Galvin, Greg Gagne. *Operating System Concepts*. Wiley, 2020. ISBN: 978-1119800361.

Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006. ISBN: 978-0321349606.

Andrew S. Tanenbaum, Herbert Bos. *Modern Operating Systems*. Pearson, 2015. ISBN: 978-0133591620.

Oracle. *The Java Tutorials - Concurrency*. Oracle, 2023.

Maurice Herlihy, Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2020. ISBN: 978-0124159501.

A. S. Tanenbaum, M. Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson, 2007. ISBN: 978-0132392273.

Douglas Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 2000. ISBN: 978-0201310092.