

Laboratorium 6 - Problem czytelników i pisarzy

Piotr Karamon

18.11.2024r.

Treści zadań

Zadanie 1

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.

Zadanie 2

- 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:

```
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
```

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.

Zadanie 1

Problem czytelników i pisarzy został rozwiązyany na dwa sposoby - przy pomocy semaforów oraz zmiennych warunkowych.

- Wersja z semaforami, preferuje czytelników, czyli jeżeli jakiś czytelnik jest już w bibliotece, to kolejni czytelnicy mogą wchodzić "od razu", natomiast pisarze muszą czekać.
- Wersja ze zmiennymi warunkowymi, preferuje pisarzy. Jeżeli czytelnik chce wejść do biblioteki, to musi sprawdzić oczywiście czy nie ma pisarza w środku, ale również czy jacyś inni pisarze nie oczekują na wejście.

W obu przypadkach zaimplementowano interfejs Library, który zawiera dwie metody `read` oraz `write`.

```
interface Library {  
    void read() throws InterruptedException;  
    void write() throws InterruptedException;  
}
```

Rozwiązanie z semaforami

W rozwiązaniu wykorzystujemy:

- semafor `resource`, który kontroluje dostęp do zasobu
- semafor `readCountMutex`, który kontroluje dostęp do zmiennej `readCount`, która przechowuje liczbę czytelników w bibliotece.
- symulujemy czytanie i pisanie za pomocą `Thread.sleep(1)`

```
class SemaphoreLibrary implements Library {  
    private final Semaphore readCountMutex = new Semaphore(1);  
    private final Semaphore resource = new Semaphore(1);  
    private int readCount = 0;  
  
    @Override  
    public void read() throws InterruptedException {  
        readCountMutex.acquire();  
        readCount++;  
        if (readCount == 1) {  
            resource.acquire();  
        }  
        readCountMutex.release();  
  
        // reading  
        Thread.sleep(1);  
  
        readCountMutex.acquire();  
        readCount--;  
        if (readCount == 0) {  
            resource.release();  
        }  
        readCountMutex.release();  
    }  
  
    @Override  
    public void write() throws InterruptedException {  
        resource.acquire();  
  
        // writing  
        Thread.sleep(1);  
  
        resource.release();  
    }  
}
```

Rozwiązanie ze zmiennymi warunkowymi

W rozwiązaniu wykorzystujemy:

- `lock` - zamek dla całej biblioteki
- `canRead` - zmienna warunkowa dla czytelników, która sygnalizuje, że można czytać
- `canWrite` - zmienna warunkowa dla pisarzy, która sygnalizuje, że można pisać

- `readCount` - liczba czytelników w bibliotece
- `writersWaiting` - liczba pisarzy oczekujących na wejście
- `isWriting` - flaga informująca czy pisarz pisze

```

class ConditionVariablesLibrary implements Library {
    private final Lock lock = new ReentrantLock();
    private final Condition canRead = lock.newCondition();
    private final Condition canWrite = lock.newCondition();
    private int readCount = 0;
    private int writersWaiting = 0;
    private boolean isWriting = false;

    @Override
    public void read() throws InterruptedException {
        lock.lock();
        try {
            while (isWriting || writersWaiting > 0) {
                canRead.await();
            }
            readCount++;
        } finally {
            lock.unlock();
        }

        // reading
        Thread.sleep(1);

        lock.lock();
        try {
            readCount--;
            if (readCount == 0) {
                canWrite.signal();
            }
        } finally {
            lock.unlock();
        }
    }

    @Override
    public void write() throws InterruptedException {
        lock.lock();
        try {
            writersWaiting++;
            while (isWriting || readCount > 0) {
                canWrite.await();
            }
            writersWaiting--;
            isWriting = true;
        } finally {
            lock.unlock();
        }

        // writing
        Thread.sleep(1);
    }
}

```

```

        try {
            lock.lock();
            isWriting = false;
            if (writersWaiting > 0) {
                canWrite.signal();
            } else {
                canRead.signalAll();
            }
        } finally {
            lock.unlock();
        }
    }
}

```

Klasa LibraryWithStats

W celu wykonania pomiarów średniego czasu `read()` oraz `write()` tworzymy klasę `LibraryWithStats`, która jest dekoratorem dla `Library`. W obu wariantach w tych metodach znajduje się wykonanie `sleep(1)`, więc różnice pomiędzy średnimi czasami wykonania `read()` oraz `write()` będą wynikały z różnych czasów oczekiwania na zasób.

```

class LibraryWithStats implements Library {
    private final Library library;
    private long totalReadTime = 0;
    private long totalWriteTime = 0;
    private long totalReads = 0;
    private long totalWrites = 0;

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

    @Override
    public void read() throws InterruptedException {
        long start = System.nanoTime();
        library.read();
        totalReadTime += System.nanoTime() - start;
        totalReads++;
    }

    @Override
    public void write() throws InterruptedException {
        long start = System.nanoTime();
        library.write();
        totalWriteTime += System.nanoTime() - start;
        totalWrites++;
    }

    public long getAvgReadTimeMicro() {
        return totalReadTime / totalReads / 1_000;
    }

    public long getAvgWriteTimeMicro() {
        return totalWriteTime / totalWrites / 1_000;
    }
}

```

Wątki czytelników i pisarzy

Wątki czytelników i pisarzy wywołują odpowiednią metodę określoną liczbę razy.

```
class Reader implements Runnable {
    private final Library library;
    private final int id;
    private final int amountOfReads;

    public Reader(Library library, int id, int amountOfReads) {
        this.library = library;
        this.id = id;
        this.amountOfReads = amountOfReads;
    }

    @Override
    public void run() {
        for (int i = 0; i < amountOfReads; i++) {
            try {
                library.read();
//                System.out.println("Reader id " + id + " read " + i + " times");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class Writer implements Runnable {
    private final Library library;
    private final int id;
    private final int amountOfWrites;

    public Writer(Library library, int id, int amountOfWrites) {
        this.library = library;
        this.id = id;
        this.amountOfWrites = amountOfWrites;
    }

    @Override
    public void run() {
        for (int i = 0; i < amountOfWrites; i++) {
            try {
                library.write();
//                System.out.println("Writer id " + id + " wrote " + i + " times");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Kod testujący

W celu wykonania pomiarów, tworzymy określoną liczbę wątków, uruchamiamy je, czekamy aż zakończą działania oraz oczywiście zbieramy czasy wykonań, wynikiem działania programu jest tabela w formacie .CSV.

```

public class ReadersWriters {
    public static void main(String[] args) {
        System.out.println("numberOfReaders,numberOfWriters," +
"semaphoreTime,conditionVariablesTime,semaphoreReadTime," +
"semaphoreWriteTime,conditionVariablesReadTime,conditionVariablesWriteTime");

        for (int i = 1; i <= 100; i++) {
            for (int j = 1; j <= 10; j++) {
                long semaphoreTimes = 0;
                long conditionVariablesTime = 0;
                long semaphoreReadTime = 0;
                long semaphoreWriteTime = 0;
                long conditionVariablesReadTime = 0;
                long conditionVariablesWriteTime = 0;

                final long N = 3;

                for (int k = 0; k < N; k++) {
                    var semLib = new LibraryWithStats(new SemaphoreLibrary());
                    var condLib = new LibraryWithStats(new ConditionVariablesLibrary());
                    semaphoreTimes += testCase(i, j, semLib);
                    conditionVariablesTime += testCase(i, j, condLib);
                    semaphoreReadTime += semLib.getAvgReadTimeMicro();
                    semaphoreWriteTime += semLib.getAvgWriteTimeMicro();
                    conditionVariablesReadTime += condLib.getAvgReadTimeMicro();
                    conditionVariablesWriteTime += condLib.getAvgWriteTimeMicro();
                }

                System.out.printf("%d,%d,%d,%d,%d,%d,%d,%d%n",
                    i,
                    j,
                    semaphoreTimes / N,
                    conditionVariablesTime / N,
                    semaphoreReadTime / N,
                    semaphoreWriteTime / N,
                    conditionVariablesReadTime / N,
                    conditionVariablesWriteTime / N
                );
            }
        }
    }

    private static long avg(List<Long> times) {
        return (long) times.stream().mapToDouble(Long::doubleValue).average().orElseThrow();
    }

    private static long testCase(int amountOfReaders, int amountOfWriters, Library library) {
        var readers = IntStream.range(0, amountOfReaders)
            .mapToObj(i -> new Reader(library, i, 10))
            .toList();
        var writers = IntStream.range(0, amountOfWriters)
            .mapToObj(i -> new Writer(library, i, 10))
            .toList();

        try (var executor = Executors.newFixedThreadPool(amountOfReaders + amountOfWriters)) {
            readers.forEach(executor::submit);

```

```
writers.forEach(executor::submit);

long startTime = System.currentTimeMillis();
executor.shutdown();
if (executor.awaitTermination(5, java.util.concurrent.TimeUnit.MINUTES)) {
    long endTime = System.currentTimeMillis();
    return endTime - startTime;
} else {
    executor.shutdownNow();
    throw new RuntimeException("Executor did not finish in time");
}
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
}
```

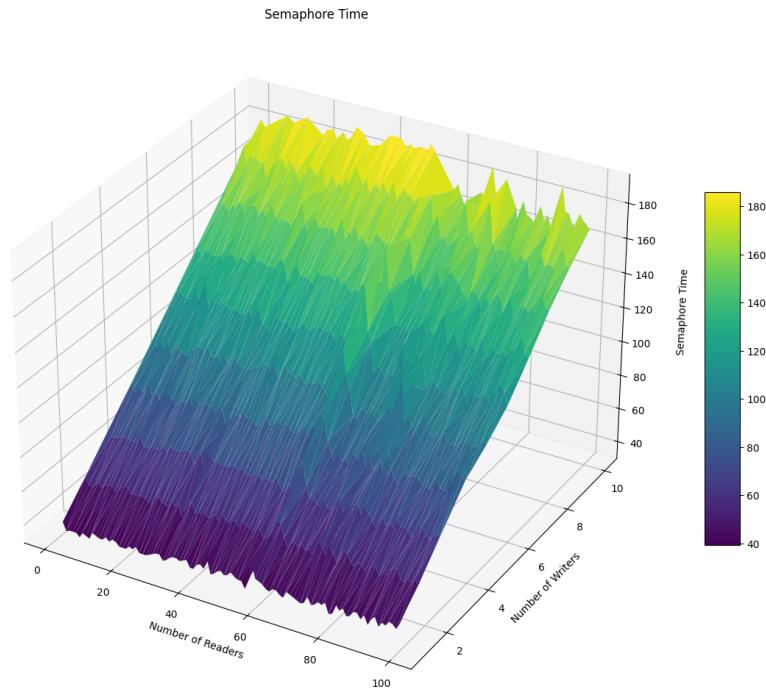
Wykresy

Wykorzystując wyjście programu testującego tworzymy wykresy 3D, które przedstawiają:

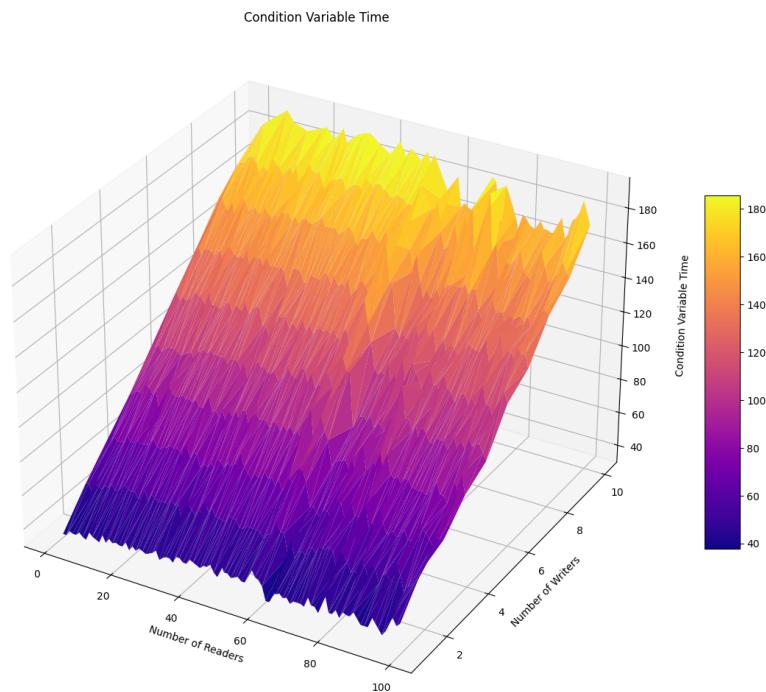
- czas wykonania
 - czas wykonania `read()`
 - czas wykonania `write()`

w zależności od liczby czytelników i pisarzy. Czas wykonania całego programu jest podany w milisekundach, natomiast średni czas wykonania `read()` oraz `write()` w mikrosekundach.

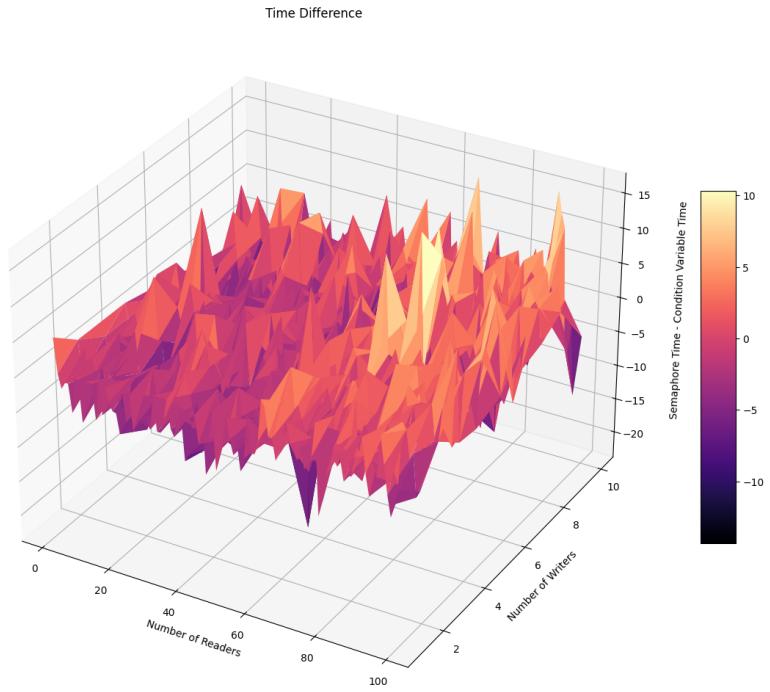
Całkowity czas wykonania



Rysunek 1: Całkowity czas wykonania dla wersji z semaforami.



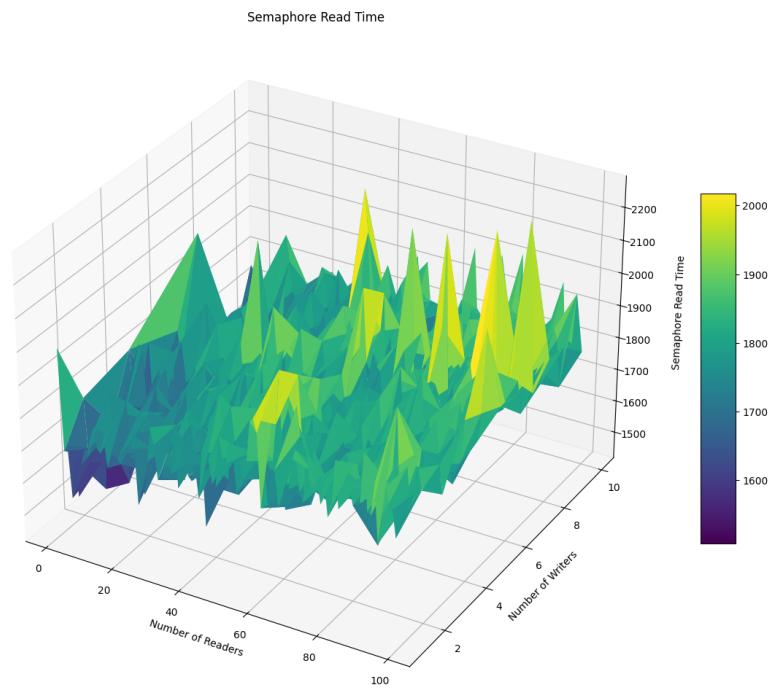
Rysunek 2: Całkowity czas wykonania dla wersji ze zmiennymi warunkowymi.



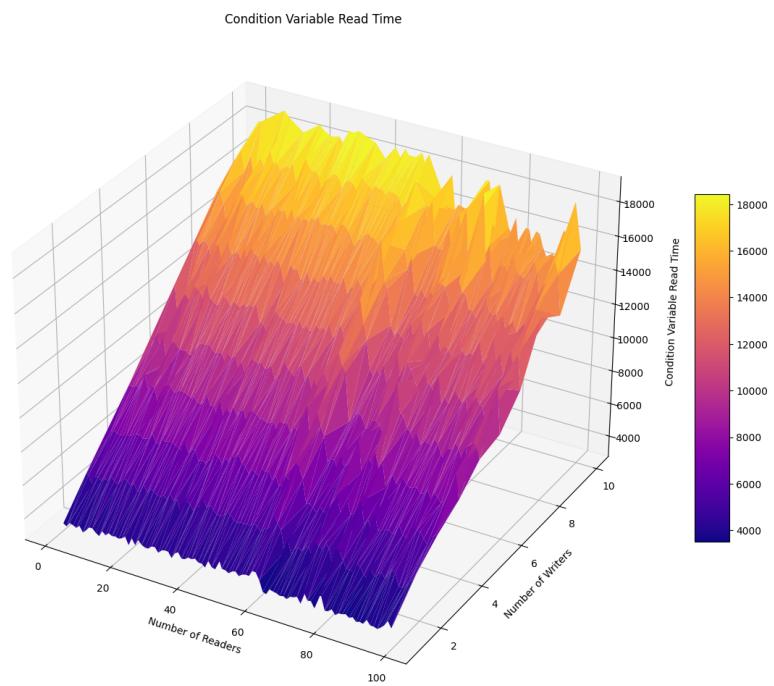
Rysunek 3: Różnica pomiędzy czasem wykowania dla wersji z semaforami i zmiennymi warunkowymi.

Wnioski: Całkowity czas wykonania dla obu wersji jest zbliżony, najlepiej obrazuje to wykres różnicę. Otrzymany kształt wygląda jak zaburzona szumem płaszczyzna równoległa do płaszczyzny xOy , skupiona na osi z w otoczeniu zera.

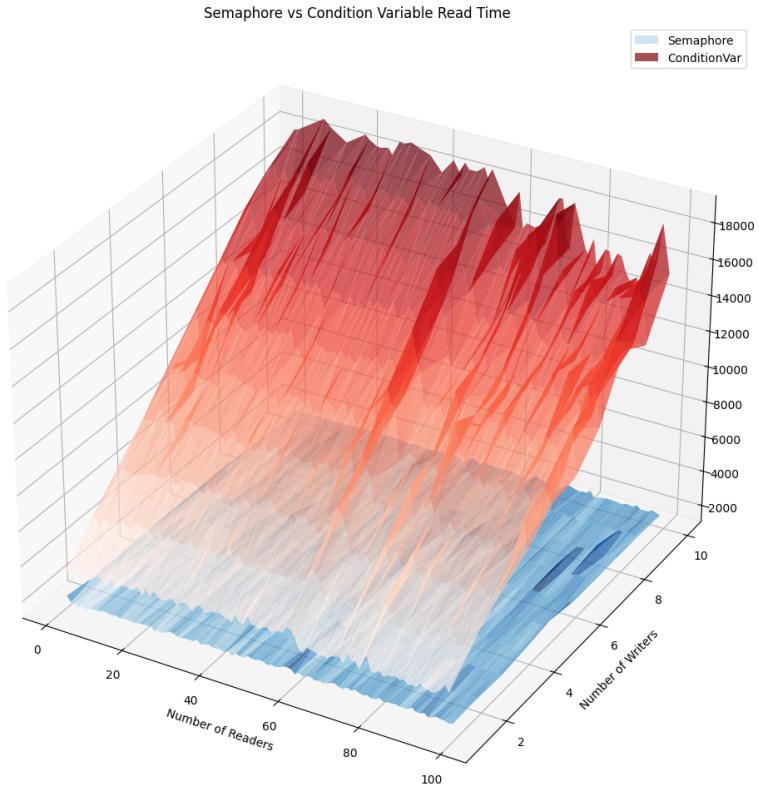
Czas wykonania read()



Rysunek 4: Średni czas wykonania read() dla wersji z semaforami.



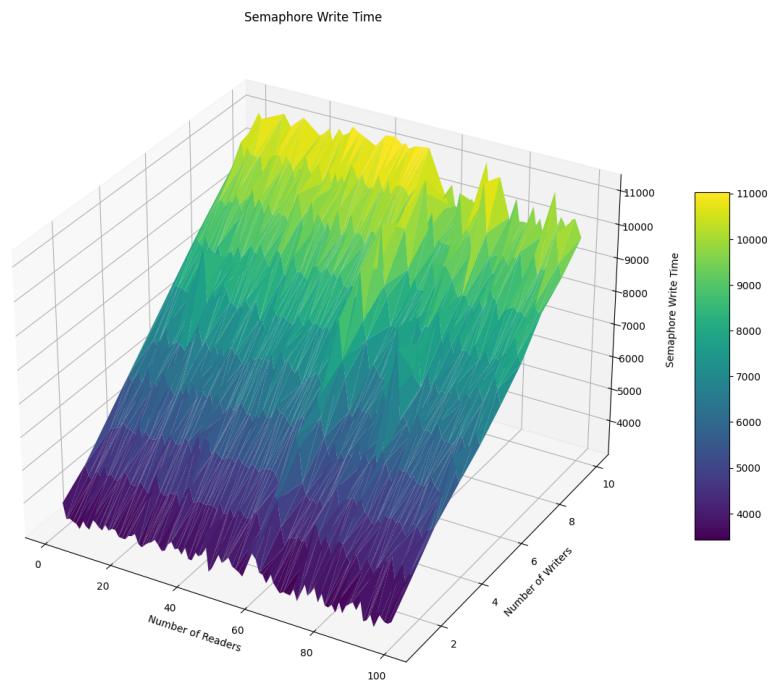
Rysunek 5: Średni czas wykonania read() dla wersji ze zmiennymi warunkowymi.



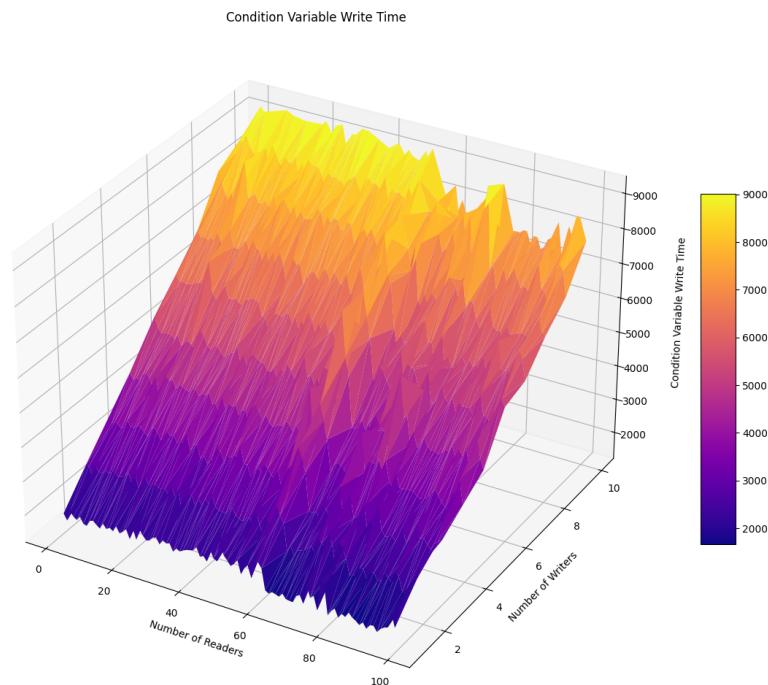
Rysunek 6: Średni czas wykowania `read()` dla obu wersji.

Wnioski: Wersja z semaforami, preferującą czytelników wedle oczekiwania szybciej wykonuje `read()`. Widzimy, że dla wersji z semaforami, czas wykonania `read()` jest w miarę stały, niezależnie od liczby czytelników i pisarzy. Jest to zrozumiałe, ponieważ czytelnicy mogą wchodzić do biblioteki równolegle, a obecność oczekujących pisarzy nie wpływa na czas oczekiwania czytelników. Dla wersji ze zmiennymi warunkowymi, która to preferuje pisarzy, czas wykonania `read()` rośnie znacząco z liczbą pisarzy, co jest zrozumiałe, ponieważ czytelnicy muszą czekać nie tylko na obecnego pisarza, ale również na innych oczekujących.

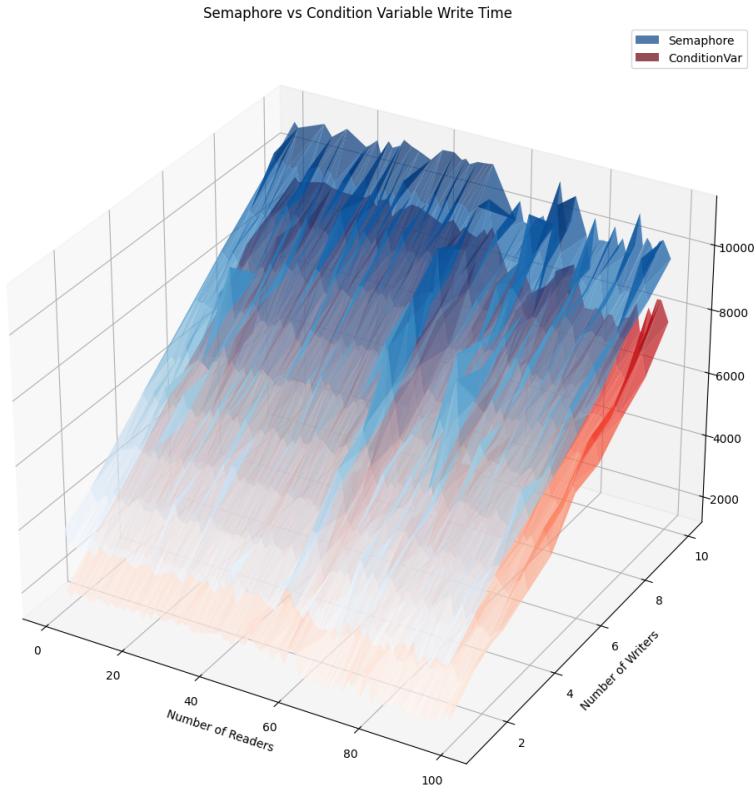
Czas wykonania write()



Rysunek 7: Średni czas wykonania write() dla wersji z semaforami.



Rysunek 8: Średni czas wykonania write() dla wersji ze zmiennymi warunkowymi.



Rysunek 9: Średni czas wykowania write() dla obu wersji.

Wnioski: Co ciekawe nie mamy tutaj sytuacji całkowicie symetrycznej z funkcją `read()`. Wersja preferująca pisarzy jest wedle oczekiwania szybsza w metodzie `write()`. Ale nadal czas ten rośnie wraz z liczbą czytelników, co jest zrozumiałe, ponieważ pisarze wymagają dostępu wyłącznego do biblioteki.

Podsumowanie

Problem czytelników i pisarzy został rozwiązyany na dwa sposoby z preferencją dla czytelników oraz pisarzy. W każdej z zaimplementowanych wersji, istnieje ryzyko zagłodzenia jednej z grup. Wersję sprawiedliwą można uzyskać np. poprzez zastosowanie kolejek FIFO. Takie rozwiązanie nosi jednak za sobą duży narzut czasowy. Zaproponowane dwa rozwiązania różnią się czasem wykonania operacji `read()` oraz `write()`. Jeżeli nasz program o wiele częściej czyta niż pisze, to warto zastosować przedstawioną wersję z semaforami, bo jest prosta i zapewnia szybkie wykonanie `read()`. Jeżeli natomiast zależy nam na jak najszybszym wprowadzaniu zmian w zasobie to możemy zastosować przedstawioną wersję ze zmiennymi warunkowymi.

Zadanie 2

Tworzymy klasę opisującą jeden węzeł listy `Node`, który zawiera wartość, referencję do następnego węzła oraz zamek.

```
class Node {  
    Object value;  
    Node next;  
    Lock lock = new ReentrantLock();  
  
    public Node(Object value) {  
        this(value, null);  
    }  
  
    public Node(Object value, Node next) {  
        this.value = value;  
        this.next = next;  
    }  
  
    public void lock() {  
        lock.lock();  
    }  
  
    public void unlock() {  
        lock.unlock();  
    }  
  
    public Node getNext() {  
        return next;  
    }  
  
    public void setNext(Node next) {  
        this.next = next;  
    }  
  
    public Object getValue() {  
        return value;  
    }  
  
    public void setValue(Object value) {  
        this.value = value;  
    }  
}
```

Tworzymy interfejs `MyList`, który zawiera trzy metody podane w zadaniu.

```
interface MyList {  
    boolean remove(Object value);  
    boolean contains(Object value);  
    void add(Object value);  
}
```

Wersja z jednym zamkiem

Wykorzystujemy tutaj zamek związany z monitorem obiektu. Implementacja to standardowe operacje na liście jednokierunkowej. Przekazujemy czasy operacji porównywania i wstawiania w celu dokonania pomiarów.

```

class SingleLockList implements MyList {
    private final Node head = new Node(null, null);
    private final int compareTime;
    private final int insertTime;

    public SingleLockList(int compareTime, int insertTime) {
        this.compareTime = compareTime;
        this.insertTime= insertTime;
    }

    @Override
    public synchronized boolean remove(Object value) {
        Node pred = head;
        Node curr = head.getNext();
        while (curr != null) {
            try {
                Thread.sleep(compareTime);
            } catch (Exception ignored) {
            }

            if (curr.getValue().equals(value)) {
                pred.setNext(curr.getNext());
                return true;
            }

            pred = curr;
            curr = curr.getNext();
        }
        return false;
    }

    @Override
    public synchronized boolean contains(Object value) {
        Node curr = head.getNext();
        while (curr != null) {
            try {
                Thread.sleep(compareTime);
            } catch (Exception ignored) {
            }

            if (curr.getValue().equals(value)) {
                return true;
            }

            curr = curr.getNext();
        }
        return false;
    }

    @Override
    public synchronized void add(Object value) {
        Node last = head;
        while (last.getNext() != null) {
            last = last.getNext();
        }

        try {
            Thread.sleep(insertTime);

```

```

        } catch (Exception ignored) {
    }

    last.setNext(new Node(value));
}
}

```

Wersja z zamkiem na jeden element

Implementacja wszystkich metod w tej wersji, opiera się na prostym schemacie działania:

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 nie musimy blokować całej listy, co oznacza że jeżeli np. dany wątek wykonuje operację `add()` i przeszedł już kilka pierwszych elementów to drugi wątek może zacząć przeglądać ten początek w celu np. sprawdzenia czy istnieje dany element w liście.

```

class LockPerElementList implements MyList {
    private final Node head = new Node(null, null);
    private final int compareTime;
    private final int insertTime;

    public LockPerElementList(int compareTime, int insertTime) {
        this.compareTime = compareTime;
        this.insertTime = insertTime;
    }

    @Override
    public boolean remove(Object value) {
        Node prev = head;
        Node curr = null;
        try {
            prev.lock();
            curr = prev.getNext();
            while (curr != null) {
                curr.lock();
                try {
                    Thread.sleep(compareTime);
                } catch (Exception ignored) {
                }

                if (curr.getValue().equals(value)) {
                    prev.setNext(curr.getNext());
                    return true;
                }
            }
        } finally {
            prev.unlock();
        }
    }
}

```

```

        prev = curr;
        curr = prev.getNext();
    }
} finally {
    if (curr != null) {
        curr.unlock();
    }
    prev.unlock();
}

return false;
}

@Override
public boolean contains(Object value) {
    Node prev = head;
    Node curr = null;

    try {
        prev.lock();
        curr = prev.getNext();
        while (curr != null) {
            curr.lock();
            try {
                Thread.sleep(compareTime);
            } catch (Exception ignored) {
            }

            if (curr.getValue().equals(value)) {
                return true;
            }

            prev.unlock();
            prev = curr;
            curr = prev.getNext();
        }
    } finally {
        if (curr != null) {
            curr.unlock();
        }
        prev.unlock();
    }
    return false;
}

@Override
public void add(Object value) {
    Node curr = head;

    try {
        curr.lock();
        Node next = curr.getNext();

        while (next != null) {
            next.lock();
            curr.unlock();

```

```

        curr = next;
        next = curr.getNext();
    }

    try {
        Thread.sleep(insertTime);
    } catch (Exception ignored) {
    }

    curr.setNext(new Node(value));
} finally {
    curr.unlock();
}
}

}

```

Kod testujący

Tworzymy 4 wątki Worker, które wykonują operacje add, remove, contains na liście. Operacje dla nich są losowe. Mierzymy jedynie czas wykonania. Wynikiem programu jest tabela w formacie .csv.

```

class Worker implements Runnable {
    private final MyList list;
    private final List<Character> operations;

    public Worker(MyList list, List<Character> operations) {
        this.list = list;
        this.operations = operations;
    }

    @Override
    public void run() {
        for (Character operation : operations) {
            switch (operation) {
                case 'a' -> list.add(ThreadLocalRandom.current().nextInt(30));
                case 'r' -> list.remove(ThreadLocalRandom.current().nextInt(30));
                case 'c' -> list.contains(ThreadLocalRandom.current().nextInt(30));
            }
        }
    }
}

public class ConcurrentLinkedList {
    public static void main(String[] args) {
        System.out.println("insertTime,compareTime,SingleLockList,LockPerElementList");

        for (int insertTime = 1; insertTime <= 10; insertTime++) {
            for (int compareTime = 1; compareTime <= 10; compareTime++) {
                var singleLockList = new SingleLockList(compareTime, insertTime);
                var lockPerElementList = new LockPerElementList(compareTime, insertTime);
                long singleLockListTime = avgTestCase(singleLockList);

                long lockPerElementListTime = avgTestCase(lockPerElementList);

                System.out.println(insertTime + "," +
                    compareTime + "," +

```

```

        singleLockListTime + "," +
        lockPerElementListTime);
    }

}

private static long avgTestCase(MyList list) {
    final int amountOfTests = 3;
    long sum = 0;
    for (int i = 0; i < amountOfTests; i++) {
        sum += testCase(list);
    }
    return sum / amountOfTests;
}

private static long testCase(MyList list) {
    var threads = new ArrayList<Thread>();
    final int amountOfThreads = 4;
    final int amountOfOperations = 25;
    Random random = new Random();

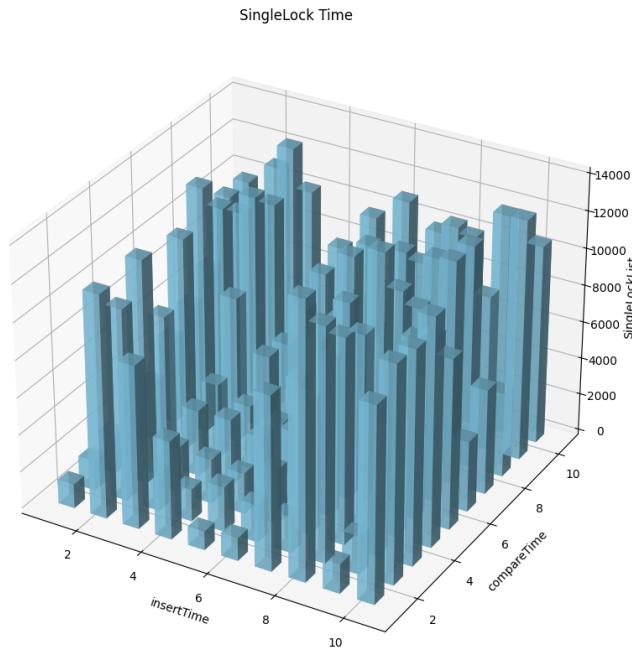
    for (int i = 0; i < amountOfThreads; i++) {
        var operations = new ArrayList<Character>();
        for (int j = 0; j < amountOfOperations; j++) {
            final int operation = random.nextInt(3);
            switch (operation) {
                case 0 -> operations.add('a');
                case 1 -> operations.add('r');
                case 2 -> operations.add('c');
            }
        }
        threads.add(new Thread(new Worker(list, operations)));
    }

    var startTime = System.currentTimeMillis();
    threads.forEach(Thread::start);
    threads.forEach(t -> {
        try {
            t.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
    var endTime = System.currentTimeMillis();
    return endTime - startTime;
}
}

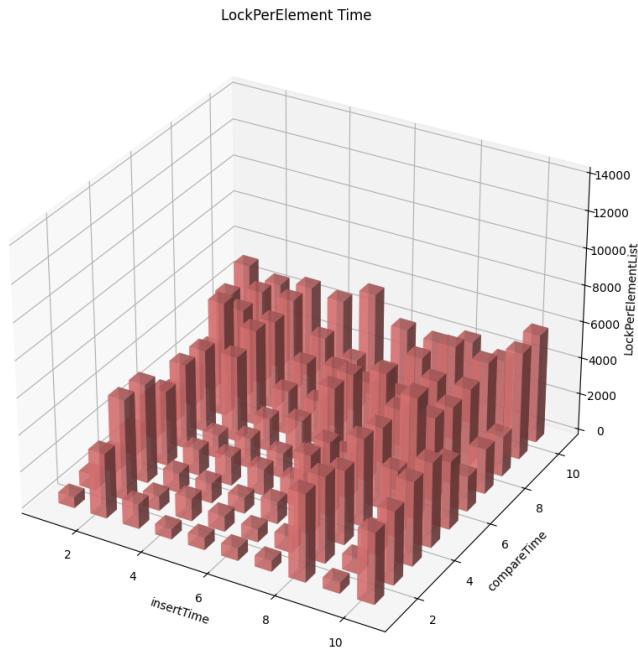
```

Wykresy

Wykorzystując wyjście programu testującego tworzymy wykresy 3D, które przedstawiają czas wykonania w zależności od czasu operacji porównywania i wstawiania. Czasy na osiach x oraz y są w milisekundach.



Rysunek 10: Całkowity czas wykonania dla wersji z jednym zamkiem.



Rysunek 11: Całkowity czas wykonania dla wersji z zamkiem na jeden element.

Wnioski: Wedle oczekiwania wersja z drobnoziarnistym blokowaniem jest szybsza, bo pozwala na wykonywanie wielu operacji równolegle. Choć jej implementacja jest bardziej skomplikowana, to jednak przyspieszenie jest znaczące. Ten przykład pokazuje, że czasami tworzenie struktur współbieżnych to nie zawsze jedynie założenie blokady na całą strukturę.

Bibliografia

- Problem czytelników i pisarzy