Lab 06

Problem czytelników i pisarzy

Adrian Madej 6.11.2023

1. Treść zadań

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).

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

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. Rozwiązywanie zadań

1. Zadanie 1

Koncepcja

Najpierw zaimplementujemy rozwiązanie problemu przy użyciu obu metod (semaforów i zmiennych warunkowych), a następnie przeprowadzimy testy, mianowicie dla każdej

konfiguracji zostanie przeprowadzone 200 iteracji, a wynikiem będzie ich uśredniony czas dla określonych parametrów. Każdy pisarz ma za zadanie wykonać 100 zapisów, z kolei czytelnik 200 odczytów.

Rozwiązanie oparte na semaforach będzie wykorzystywać dwa semafory – jeden dla pisarzy, a drugi dla czytelników.

Rozwiązanie za pomocą zmiennych warunkowych uzyskamy stosując zamki (lock) oraz warunki (condition), które posłużą nam do odpowiedniej synchronizacji wątków.

Na podstawie utworzonych testów zostaną stworzone wykresy.

Implementacja

Tworzymy interfejs ILibrary

```
interface ILibrary {
    void read() throws InterruptedException;

    void write() throws InterruptedException;
}
```

Tworzymy klasę Pisarza

```
class Writer extends Thread {
    private final ILibrary library;
    private final int iterations;

    public Writer(ILibrary library, int iterations) {
        super();
        this.library = library;
        this.iterations = iterations;
    }

    @Override
    public void run() {
        for (int i = 0; i < iterations; i++) {
            try {
                library.write();
            } catch (InterruptedException exception) {
                break;
            }
        }
    }
}
```

Tworzymy klasę Czytelnika

```
public class Reader extends Thread{
    private final ILibrary library;
    private final int iterations;

    public Reader(ILibrary library, int iterations){
        super();
        this.library = library;
        this.iterations = iterations;
    }

    @Override
    public void run(){
        for (int i = 0; i < iterations; i++){
            try {
                library.read();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

Implementujemy ILibrary za pomocą semaforów

```
class SemaphoreLibrary implements ILibrary {
    private final Semaphore writerSemaphore = new
Semaphore(1);
    private final Semaphore readerSemaphore = new
Semaphore(1);
    private final AtomicInteger readerCount = new
AtomicInteger(0);

    @Override
    public void read() throws InterruptedException {
        readerSemaphore.acquire();
        int readers = readerCount.incrementAndGet();
        if (readers == 1) {
            writerSemaphore.acquire();
        }
        readerSemaphore.release();
        // read
        readerSemaphore.acquire();
        readers  = readerCount.decrementAndGet();
        if (readers == 0) {
            writerSemaphore.release();
        }
        readerSemaphore.release();
    }
```

```
    @Override
    public void write() throws InterruptedException {
        writerSemaphore.acquire();
        // write
        writerSemaphore.release();
    }
}
```

Implementujemy ILibrary z pomocą zmiennych warunkowych

```
class LockLibrary implements ILibrary {
    private final Lock lock = new ReentrantLock();
    private final Condition noWriters = lock.newCondition();
    private final Condition noReadersWriters =
lock.newCondition();
    private boolean isWriter = false;
    private int readersCount = 0;

    @Override
    public void read() throws InterruptedException {
        lock.lock();
        try {
            while (isWriter) {
                noWriters.await();
            }
            readersCount++;
        } finally {
            lock.unlock();
        }
        // read
        lock.lock();
        try {
            readersCount--;
            if (readersCount == 0) {
                noReadersWriters.signal();
            }
        } finally {
            lock.unlock();
        }
    }
    @Override
    public void write() throws InterruptedException {
        lock.lock();
        try {
            while (readersCount > 0 || isWriter) {
                noReadersWriters.await();
            }

            isWriter = true;
        } finally {
            lock.unlock();
```

```
        }
        // write
        lock.lock();
        try {
            isWriter = false;
            noReadersWriters.signal();
            noWriters.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

Uzupełniamy klasę Main w celu przeprowadzenia testów:

```
public class Main {

    public static void main(String[] args) {
        List<Integer> readersNums = List.of(10, 40, 70, 100);
        List<Integer> writersNums = List.of(1, 4, 7, 10);

        LinkedList<String> results = new LinkedList<String>();

        for (Integer w : writersNums) {
            for (Integer r : readersNums) {
                ILibrary lib = new SemaphoreLibrary();
                ILibrary lib2 = new LockLibrary();
                double avg1 = testCaseWrapper(r, w, lib);
                double avg2 = testCaseWrapper(r, w, lib2);
                results.add(w + "," + r + "," + round(avg1, 2)
+ "," + round(avg2, 2));
            }
        }

        Path out = Paths.get("results.csv");
        try {
            Files.write(out, results,
Charset.defaultCharset());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static double round(double v, int places) {
        v = Math.round(v * Math.pow(10, places));
        v = v / Math.pow(10, places);
        return v;
    }

    private static double testCaseWrapper(int r, int w,
ILibrary lib) {
```

```
        LinkedList<Long> times = new LinkedList<Long>();

        IntStream.range(0, 200).forEach(i -> {
            long before = System.currentTimeMillis();

            testCase(r, w, lib);

            long diff = System.currentTimeMillis() - before;
            times.add(diff);
        });

        return times.stream().mapToLong(i ->
i).average().getAsDouble();
    }

    private static void testCase(int r, int w, ILibrary lib) {
        var executor = Executors.newFixedThreadPool(r + w);
        LinkedList<Thread> threadList = new
LinkedList<Thread>();

        IntStream.range(0, r).forEach(i ->
                threadList.add(new Reader(lib, 200))
        );
        IntStream.range(0, w).forEach(i ->
                threadList.add(new Writer(lib, 100))
        );

        threadList.forEach(executor::submit);

        executor.shutdown();
        try {
            executor.awaitTermination(Long.MAX_VALUE,
TimeUnit.NANOSECONDS);
        } catch (InterruptedException ignored) {
        }
    }
}
```

Wyniki

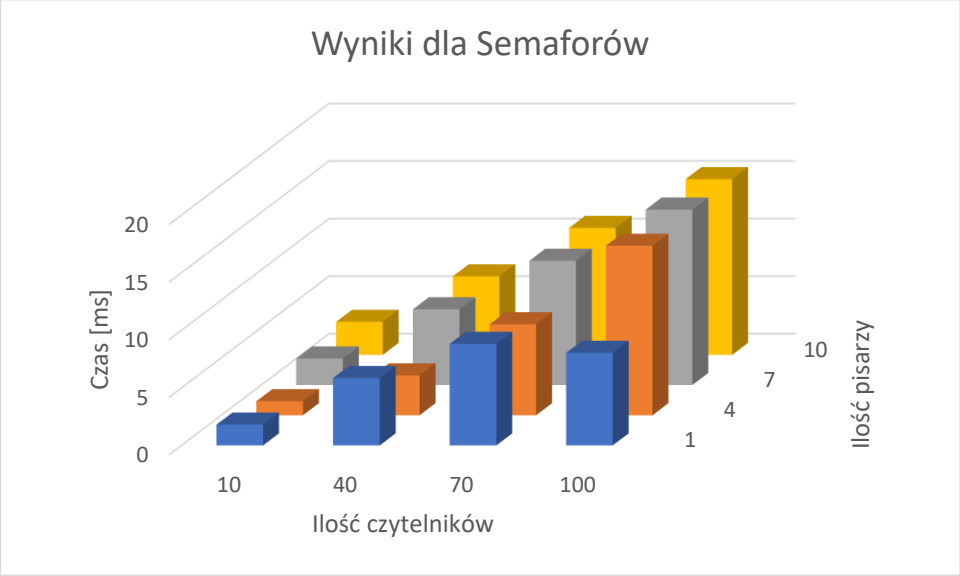
W wyniku wykonania programu utworzył się plik results.csv:

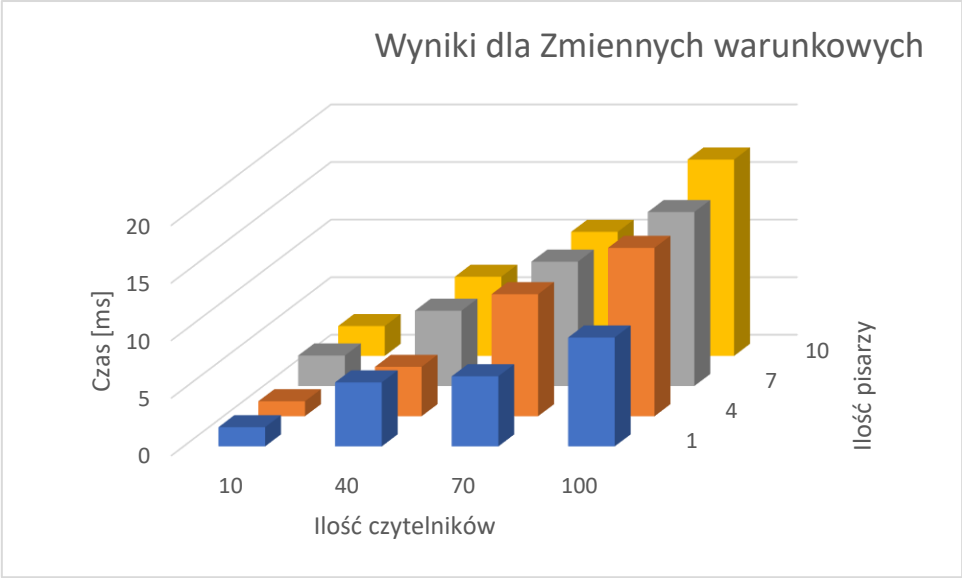
Pisarze	Czytelnicy	Czas Semafory [ms]	Czas Zmienne [ms]
1	10	1,81	1,67
1	40	5,85	5,56
1	70	8,82	6,09
1	100	8,04	9,46
4	10	1,22	1,3

4	40	3,46	4,29
4	70	7,9	10,6
4	100	14,72	14,62
7	10	2,31	2,65
7	40	6,58	6,54
7	70	10,79	10,8
7	100	15,24	15,11
10	10	2,87	2,59
10	40	6,82	6,87
10	70	11,03	10,77
10	100	15,26	17,05

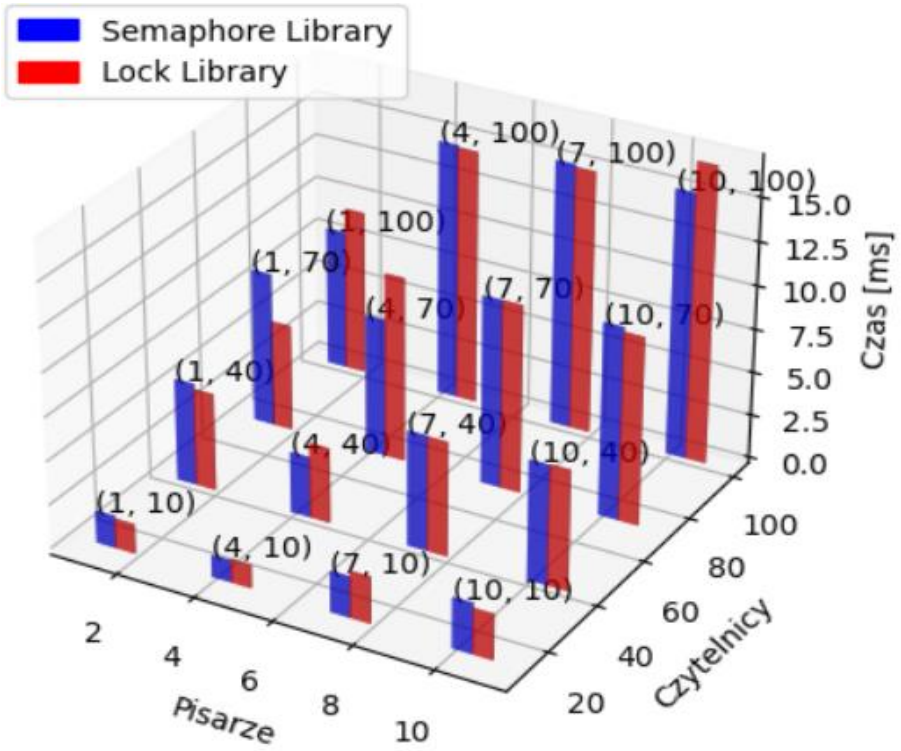
Wnioski

Na podstawie powyższych danych tworzymy wykresy 3D.





Nanosząc dane na jeden wykres otrzymujemy:



Jak widzimy różnice w czasach nie są zbyt wyraźne. Semafory okazały się w 9 przypadkach szybsze, natomiast różnica dzieje się głównie w częściach tysięcznych sekundy co sugeruje, że żadne z rozwiązań nie jest wyraźnie szybsze.

Warto zauważyć, że rozwiązanie z semaforami powoduje iż czytelnicy nie muszą czekać na wejście do czytelni nawet w wypadku, gdy na wejście czeka już pisarz, co może prowadzić do zagłodzenia pisarzy.

Rozwiązanie ze zmiennymi warunkowymi faworyzuje natomiast pisarzy, ponieważ gdy pisarz oczekuje na wejście do czytelni, żaden nowy czytelnik nie będzie już mógł do niej wejść. Wówczas, możliwe jest zagłodzenie czytelników.

2. Zadanie 2

Koncepcja

Rozwiązując zadanie zaimplementujemy oba typy list oraz zmierzymy czas wykonywania programu. Jako zmienne parametry posłużą nam czas wstawiania elementu oraz czas porównywania jednego elementu z innym. Każdy test wykonamy 10 razy. Podobnie jak poprzednio, czas wynikowy będzie średnią ze wszystkich testów. Każdy wątek ma za zadanie wykonać 30 operacji na liście. Testy przeprowadzimy dla 2 oraz 8 współdzielonych wątków.

Implementacja

Tworzymy interfejs ILockedList

```
public interface ILockedList {
    void add(Object o);

    boolean contains(Object o);

    void remove(Object o);

    void clear();

    int size();
}
```

Tworzymy węzeł

```
class Node {
    private Node next = null;
    private final Lock lock = new ReentrantLock();
    private Object data;

    public Node(Object data) {
```

```
        this.data = data;
    }

    public Node next() {
        return next;
    }

    public void setNext(Node n) {
        next = n;
    }
    public void lock() {
        lock.lock();
    }

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

    public Object getData() {
        return data;
    }
}
```

Tworzymy listę z jednym zamkiem blokującym

```
public class SingleLockedList implements ILockedList {

    private final Node sentinel = new Node(null);
    private final int insertTime;
    private final int compareTime;
    private final Lock lock = new ReentrantLock();

    public SingleLockedList(int iTime, int cTime) {
        insertTime = iTime * 1000;
        compareTime = cTime * 1000;
    }

    @Override
    public void add(Object o) {
        lock.lock();

        try {
            Node prev = sentinel;
            Node current = sentinel.next();

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

            try {
```

```
                Thread.sleep(0, insertTime);
            } catch (InterruptedException ignored) {
            }

            prev.setNext(new Node(o));
        } finally {
            lock.unlock();
        }
    }

    @Override
    public boolean contains(Object o) {
        lock.lock();
        try {
            Node current = sentinel.next();

            while (current != null) {
                try {
                    Thread.sleep(0, compareTime);
                } catch (InterruptedException ignored) {
                }

                if (current.getData().equals(o)) {
                    return true;
                }

                current = current.next();
            }
        } finally {
            lock.unlock();
        }

        return false;
    }

    @Override
    public void remove(Object o) {
        lock.lock();
        try {
            Node prev = sentinel;
            Node current = sentinel.next();

            while (current != null) {
                try {
                    Thread.sleep(0, compareTime);
                } catch (InterruptedException ignored) {
                }

                if (current.getData().equals(o)) {
                    prev.setNext(current.next());
                    return;
```

```
                }

                prev = current;
                current = current.next();
            }
        } finally {
            lock.unlock();
        }
    }

    @Override
    public void clear() {
        sentinel.setNext(null);
    }

    @Override
    public int size() {
        int size = 0;
        Node current = sentinel;
        while(current != null){
            size++;
            current = current.next();
        }

        return size - 1;
    }
}
```

Tworzymy listę „drobnoziarnistą”

```
public class LockedList implements ILockedList {
    private final Node sentinel = new Node(null);
    private final int insertTime;
    private final int compareTime;

    public LockedList(int iTime, int cTime) {
        insertTime = iTime * 1000;
        compareTime = cTime * 1000;
    }

    @Override
    public void add(Object o) {
        Node current = sentinel;
        try {
            current.lock();
            Node next = sentinel.next();

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

                Node temp = current;
```

```
                current = next;
                temp.unlock();

                next = current.next();
            }

            try {
                Thread.sleep(0, insertTime);
            } catch (InterruptedException ignored) {
            }

            current.setNext(new Node(o));
        } finally {
            current.unlock();
        }
    }

    @Override
    public boolean contains(Object o) {
        Node prev = sentinel;
        Node current = null;
        try {
            prev.lock();
            current = sentinel.next();
            while (current != null) {
                current.lock();

                try {
                    Thread.sleep(0, compareTime);
                } catch (InterruptedException ignored) {
                }

                if (current.getData().equals(o)) {
                    return true;
                }

                prev.unlock();
                prev = current;
                current = current.next();
            }
        } finally {
            if (current != null) {
                current.unlock();
            }
            prev.unlock();
        }
        return false;
    }

    @Override
    public void remove(Object o) {
```

```
        Node prev = sentinel;
        Node current = null;
        try {
            prev.lock();
            current = sentinel.next();
            while (current != null) {
                current.lock();

                try {
                    Thread.sleep(0, compareTime);
                } catch (InterruptedException ignored) {
                }

                if (current.getData().equals(o)) {
                    prev.setNext(current.next());
                    return;
                }

                prev.unlock();
                prev = current;
                current = current.next();
            }
        } catch (Exception e){
            e.printStackTrace();
        }
        finally {
            if (current != null) {
                current.unlock();
            }
            prev.unlock();
        }
    }

    @Override
    public void clear() {
        sentinel.setNext(null);
    }

    @Override
    public int size() {
        int size = 0;
        Node current = sentinel;
        while(current != null){
            size++;
            current = current.next();
        }

        return size - 1;
    }
}
```

Implemetnujemy klasę Worker wykonującą polecenia

```
class Worker extends Thread {
    private final ILockedList list;
    private final List<Integer> operations;
    private final Random random = new Random();

    public Worker(ILockedList l, List<Integer> op) {
        list = l;
        operations = op;
    }

    @Override
    public void run() {
        for (var op : operations) {
            switch (op) {
                case 0 -> // add
                        list.add(random.nextInt(20));
                case 1 -> // remove
                        list.remove(random.nextInt(20));
                case 2 -> // contains
                        list.contains(random.nextInt(20));
            }

        }
    }
}
```

Tworzymy klasę Main przeprowadzającą testy

```
public class Main {
    public static void main(String[] args) {
        var insertTime = List.of(10, 500, 999);
        var compareTime = List.of(10, 500, 999);

        var results = new LinkedList<String>();
        for (var i : insertTime) {
            for (var c : compareTime) {
                var fineGrainedList = new LockedList(i, c);
                var singleLockedList = new SingleLockedList(i,
c);
                var avg1 = testCaseWrapper(fineGrainedList);
                var avg2 = testCaseWrapper(singleLockedList);

                System.out.println("insertTime: " + i + "ns,
compareTime: " + c + "ns | avg1: " + avg1 + "ms," + " avg2: "
+ avg2 + "ms");
```

```
                results.add(i + ", " + c + ", " + round(avg1,
2) + ", " + round(avg2, 2));
            }
        }

        Path out = Paths.get("results2.csv");
        try {
            Files.write(out, results,
Charset.defaultCharset());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static double round(double v, int places){
        v = Math.round(v * Math.pow(10, places));
        v = v / Math.pow(10, places);
        return v;
    }

    public static double testCaseWrapper(ILockedList list) {
        var times = new LinkedList<Long>();

        IntStream.range(0, 10).forEach(i -> {
            long before = System.currentTimeMillis();

            testCase(list);

            long diff = System.currentTimeMillis() - before;
            times.add(diff);
        });

        return times.stream().mapToLong(i ->
i).average().getAsDouble();
    }

    private static void testCase(ILockedList list) {
        var random = new Random();
        final int OPERATIONS_NUM = 30;
        final int THREAD_NUM = 2;

        var threadList = new LinkedList<Thread>();
        IntStream.range(0, THREAD_NUM).forEach(i -> {
            var op = new LinkedList<Integer>();
            IntStream.range(0, OPERATIONS_NUM).forEach(j -> {
                op.add(random.nextInt(3));
            });
            threadList.add(new Worker(list, op));
        });

        var executor =
```

```
Executors.newFixedThreadPool(THREAD_NUM);
        threadList.forEach(executor::submit);
        executor.shutdown();

        try {
            executor.awaitTermination(Long.MAX_VALUE,
TimeUnit.SECONDS);
        } catch (InterruptedException exception) {
            exception.printStackTrace();
        }

        list.clear();
    }
}
```

Wyniki

W wyniku odpalenia programu utworzył się plik results2.csv

Dla 2 wątków otrzymujemy

Wstawianie [ns]	Porównywanie [ns]	Drobnoziarniste [ms]	Jeden zamek [ms]
10	10	418,5	603,1
10	500	391,7	536,4
10	999	368,6	491,1
500	10	363,7	518,1
500	500	405	564,7
500	999	398,9	530,3
999	10	350,5	541
999	500	400,7	519
999	999	393,5	510,2

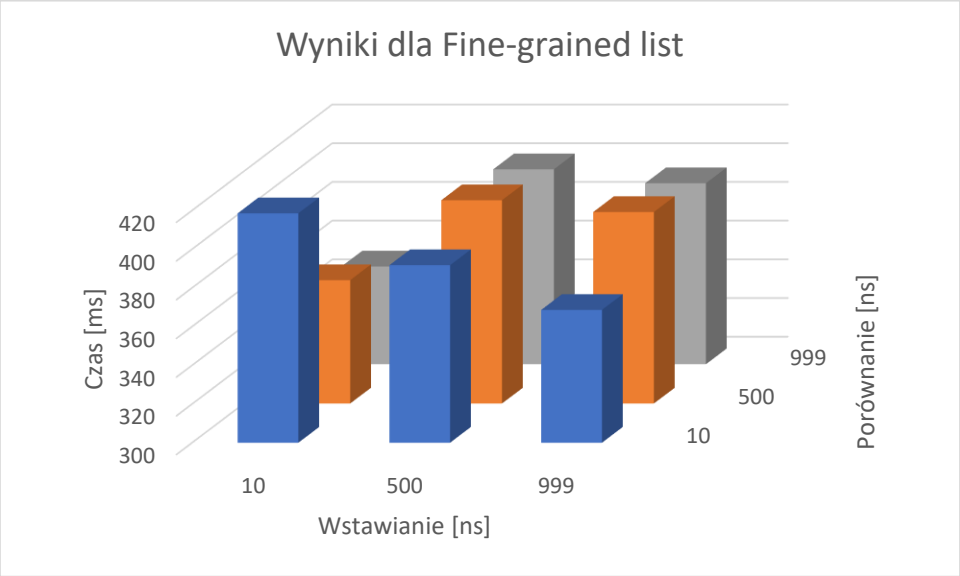
Dla 8 wątków otrzymujemy

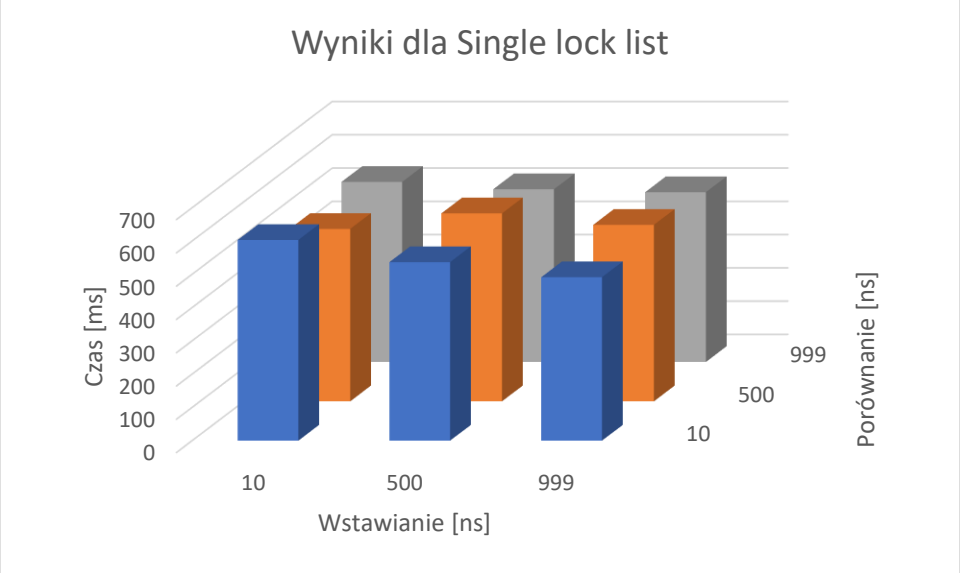
Wstawianie [ns]	Porównywanie [ns]	Drobnoziarniste [ms]	Jeden zamek [ms]
10	10	1239,1	4939
10	500	1306,6	4772,5
10	999	1321,3	4408,9
500	10	1330,9	4853,3
500	500	1292,2	4759,3
500	999	1264,3	4868,1
999	10	1292,7	4447,5
999	500	1363,3	4694
999	999	1403	4723,8

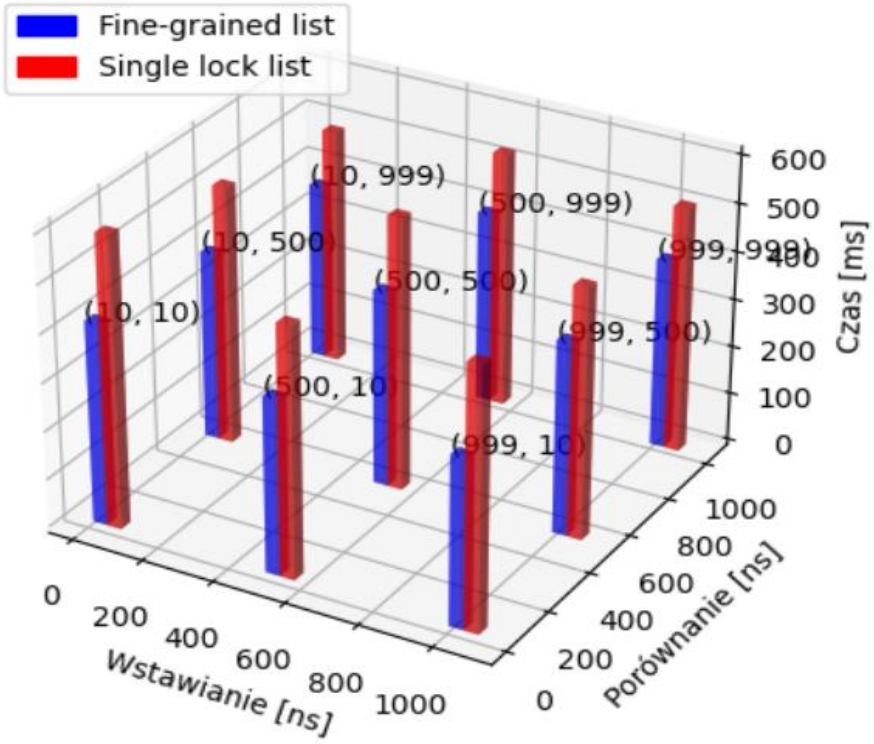
Wnioski

Z powyższych danych otrzymujemy wykresy.

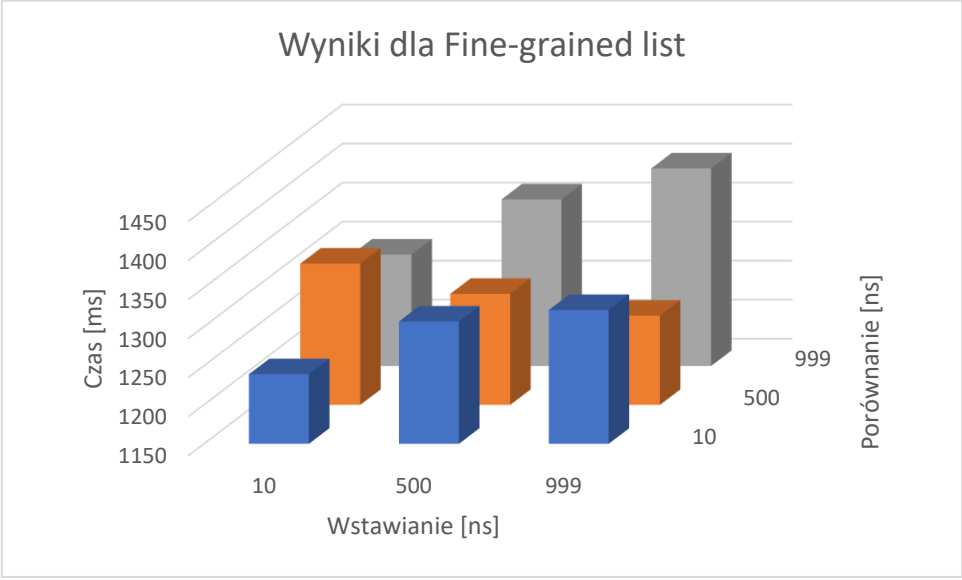
Dla 2 wątków mamy:

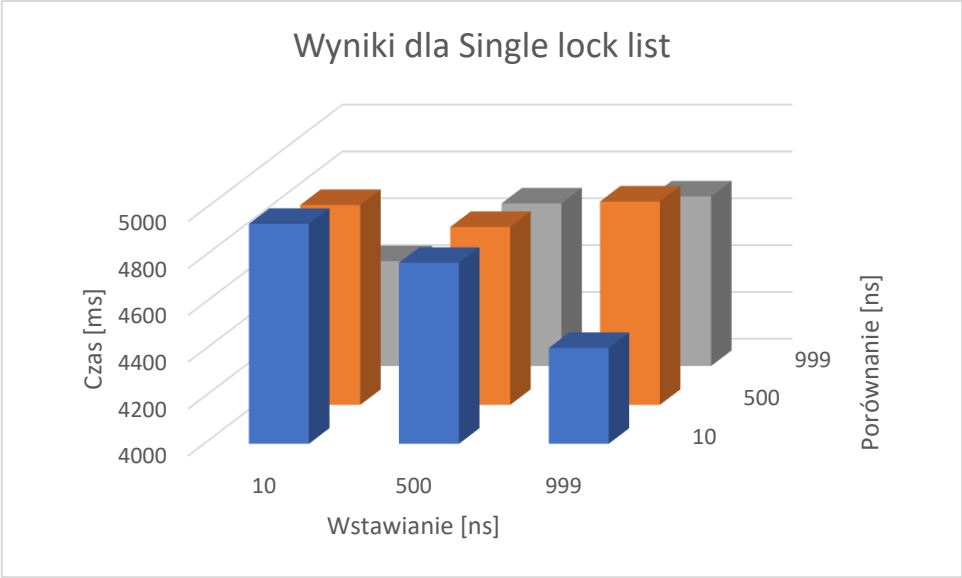


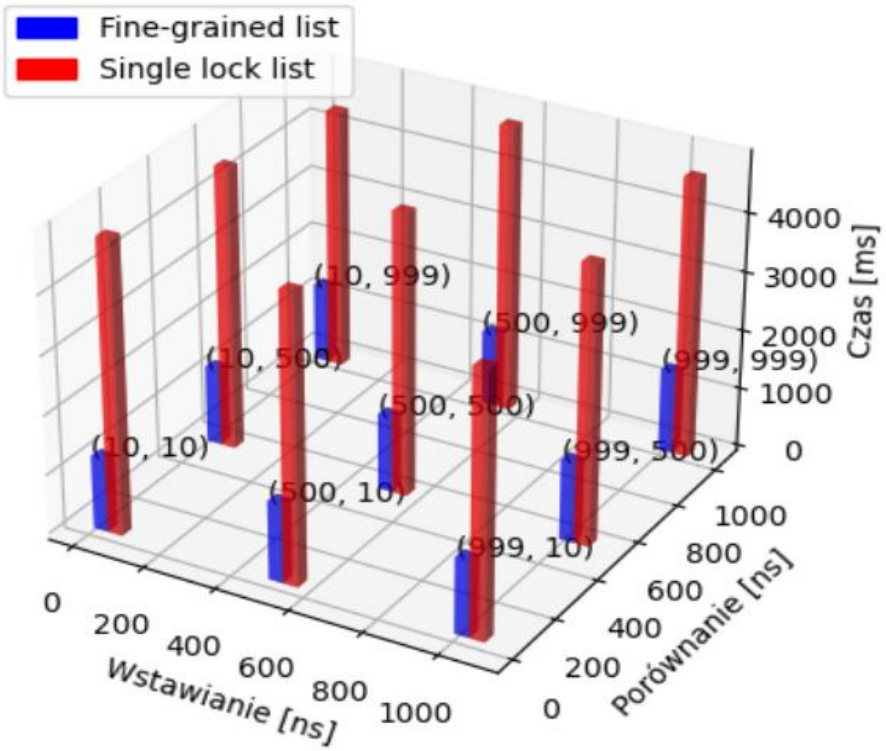




Dla 8 wątków otrzymujemy







Nietrudno zauważyć, że w obu przypadkach szybsza okazała się lista stosująca blokowanie drobnoziarniste. Przy dwóch wątkach różnica między czasami była nie większa niż 200 ms,

natomiast w przypadku wykorzystania 8 wątków wynosiła ona ponad 3 tysiące ms. Dzieje się tak, ponieważ w blokowaniu drobnoziarnistym blokujemy tylko wykorzystywane węzły, przez co inne wątki mogą pracować na innych częściach listy. Zużywa to co prawda więcej pamięci, jednakże jest to znacznie lepsze podejście, jeśli z naszej struktury chce korzystać wiele wątków.

3. Wnioski ogólne

- Semafor jest wygodny do zarządzania dostępem do zasobów. Przydatny, gdy ilość dostępnych zasobów jest kluczowa.
- Zmienna warunkowa jest przydatna w przypadku oczekiwania na konkretne warunki. Stosowana, gdy wątki muszą być powiadamiane o zmianie stanu.
- Podejście drobnoziarniste (fine-grained) do blokowania zakłada, że blokujemy tylko te fragmenty struktury danych, które są faktycznie używane przez dany wątek. To pozwala na większą równoległość, ponieważ inne wątki mogą równocześnie pracować na różnych częściach struktury danych. Jednak może prowadzić do zwiększenia zużycia pamięci.
- Globalne blokowanie blokuje dostęp do całej struktury danych, co może prowadzić do większych opóźnień i konkurencji między wątkami, zwłaszcza gdy korzysta się z większej liczby wątków.

4. Bibliografia

- Materiały laboratorium
- Baeldung – Guide to Java locks
- Baeldung – Semaphores in Java
- Dokumentacja Javy