Laboratorium 4 - Producenci i konsumenci z losowa ilością pobieranych i wstawianych porcji

Piotr Karamon

04.11.2024r.

Treści zadań

Zadanie

- Bufor o rozmiarze 2M
- Jest m producentów i n konsumentów
- Producent wstawia do bufora losowa liczbę elementów (nie więcej niż M)
- Konsument pobiera losowa liczbę elementów (nie więcej niż M)
- Zaimplementować przy pomocy monitorów Javy oraz mechanizmów Java Concurrency Utilities
- Przeprowadzic porownanie wydajnosci (np. czas wykonywania) vs. rozne parametry, zrobic wykresy i je skomentowac

Zadanie

Dwie różne implementacje bufora będą dzedziczyły po interfejsie `Buffer`. W obu przypadkach zaimplementujemy metody `put` oraz `get`, które będą odpowiedzialne za dodawanie i pobieranie elementów z bufora. Również oba bufory będą miały metody `removeProducer` oraz `removeConsumer`, które będą służyły producentom i konsumentom do informowania bufora o zakończeniu pracy.

Kod interfejsu `Buffer`

```
interface Buffer {
    void put(Collection<Integer> values) throws InterruptedException;
    Collection<Integer> get(int howMany) throws InterruptedException;
    void removeProducer();
    void removeConsumer();
}
```

`BufferUsingMonitors`

Używamy metod `wait` oraz `notifyAll` w celu synchronizacji wątków. W przypadku metody `put`, jeżeli w buforze jest za mało miejsca na wstawienie nowych elementów, to wątek producenta czeka na powiadomienie od konsumenta. W przypadku metody `get`, jeżeli w buforze jest za mało elementów, to wątek konsumenta czeka na powiadomienie od producenta.

```
class BufferUsingMonitors implements Buffer {
    private final int capacity;
    private final List<Integer> buffer;
    private int producers;
    private int consumers;

    public BufferUsingMonitors(int M, int m, int n) {
        this.buffer = new LinkedList<>();
        this.capacity = 2 * M;
        producers = m;
        consumers = n;
    }

    @Override
    public synchronized void put(Collection<Integer> values) throws InterruptedException
    ↪ {
        while (buffer.size() + values.size() > capacity) {
            if (consumers == 0) {
                throw new InterruptedException();
            }
            wait();
        }

        buffer.addAll(values);
//          System.out.println("Producer produced " + values);
        notifyAll();
    }

    @Override
    public synchronized Collection<Integer> get(int howMany) throws InterruptedException
    ↪ {
        while (buffer.size() < howMany) {
            if (producers == 0) {
                throw new InterruptedException();
            }
            wait();
        }


        List<Integer> result = new LinkedList<>();
        for (int i = 0; i < howMany; i++) {
            result.add(buffer.removeFirst());
        }
```

```
//          System.out.println("Consumer consumed" + result);
        notifyAll();
        return result;
    }


    @Override
    public synchronized void removeProducer() {
        producers--;
        if (producers == 0) {
            notifyAll();
        }
    }

    @Override
    public synchronized void removeConsumer() {
        consumers--;
        if (consumers == 0) {
            notifyAll();
        }
    }
}
```

BufferUsingJavaUtils

Używamy klas `ReentrantLock` oraz `Condition` z pakietu `java.util.concurrent.locks`. `ReentrantLock` jest zamkiem podoobnym do zamka związanego z `synchronized`, ale daje większą kontrolę nad blokowaniem i odblokowywaniem wątków. `Condition` jest mechanizmem, który pozwala na czekanie na powiadomienie od innego wątku. Przewagą `Condition` nad `wait` oraz `notifyAll` jest to, że możemy mieć wiele warunków, na które czekamy. To pozwala na redukcję ilości fałszywych przebudzeń.

```
class BufferUsingUtils implements Buffer {
    private final int capacity;
    private final List<Integer> buffer;
    private final Lock lock;
    private final Condition notFull;
    private final Condition notEmpty;
    private int producers;
    private int consumers;

    public BufferUsingUtils(int M, int m, int n) {
        this.capacity = 2 * M;
        this.buffer = new LinkedList<>();
        this.lock = new ReentrantLock();
        this.notFull = lock.newCondition();
        this.notEmpty = lock.newCondition();
        this.producers = m;
        this.consumers = n;
    }
```

```
    @Override
    public void put(Collection<Integer> values) throws InterruptedException {
        lock.lock();
        try {
            while (buffer.size() + values.size() > capacity) {
                if (consumers == 0) {
                    throw new InterruptedException();
                }
                notFull.await();
            }
            buffer.addAll(values);
//              System.out.println("Producer produced " + values);
            notEmpty.signalAll();
        } finally {
            lock.unlock();
        }
    }

    @Override
    public Collection<Integer> get(int howMany) throws InterruptedException {
        lock.lock();
        try {
            while (buffer.size() < howMany) {
                if (producers == 0) {
                    throw new InterruptedException();
                }
                notEmpty.await();
            }
            List<Integer> result = new LinkedList<>();
            for (int i = 0; i < howMany; i++) {
                result.add(buffer.removeFirst());
            }
//              System.out.println("Consumer consumed" + result);
            notFull.signalAll();
            return result;
        } finally {
            lock.unlock();
        }
    }


    @Override
    public void removeProducer() {
        lock.lock();
        try {
            producers--;
            if (producers == 0) {
                notEmpty.signalAll();
            }
        } finally {
```

```
            lock.unlock();
        }
    }

    @Override
    public void removeConsumer() {
        lock.lock();
        try {
            consumers--;
            if (consumers == 0) {
                notFull.signalAll();
            }
        } finally {
            lock.unlock();
        }
    }
}
```

Kod testujący

Tworzymy dwie metody:

- `testRun` przeprowadza test dla podanych parametrów i zwraca czas wykonania
- `avgTestRun` przeprowadza test dla podanych parametrów i zwraca średni czas wykonania

```
private static long avgTestRun(int amountOfRuns, int M, int m, int n, Supplier<Buffer>
↪   bufferSupplier) {
    long sum = 0;
    for (int i = 0; i < amountOfRuns; i++) {
        long res = testRun(M, m, n, bufferSupplier.get());
        sum += res;
    }
    return Math.round((double) sum / (double) amountOfRuns);
}


private static long testRun(int M, int m, int n, Buffer buffer) {
    var producers = IntStream.range(0, m)
            .mapToObj(i -> new Producer(buffer, M, 500))
            .map(Thread::new)
            .toList();

    var consumers = IntStream.range(0, n)
            .mapToObj(i -> new Consumer(buffer, M, 500))
            .map(Thread::new)
            .toList();

    long startTime = System.currentTimeMillis();
```

```
    producers.forEach(Thread::start);
    consumers.forEach(Thread::start);
    Stream.concat(producers.stream(), consumers.stream())
            .forEach(t -> {
                try {
                    t.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });

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

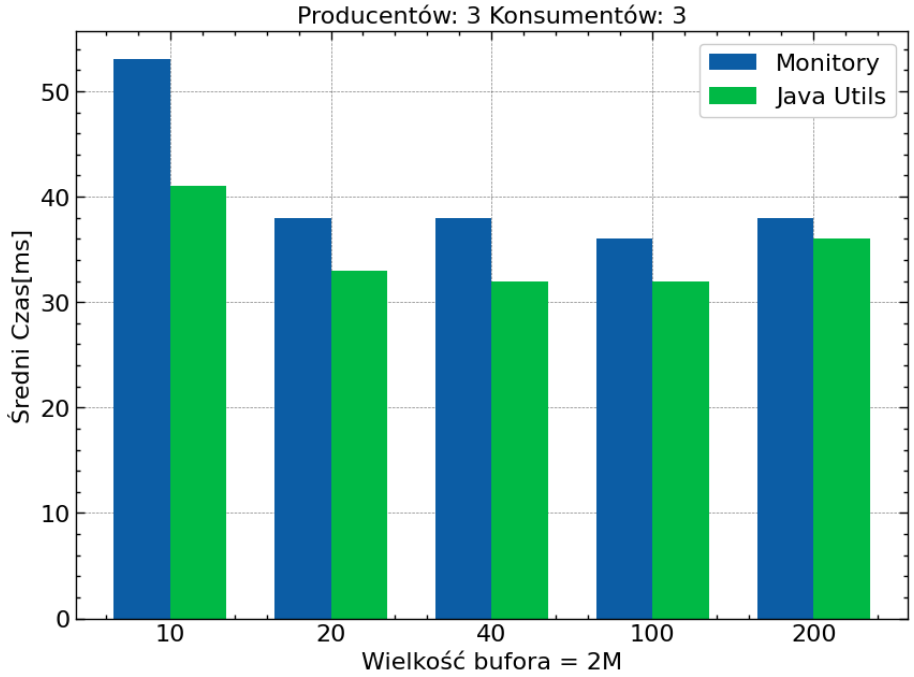
Kod testujemy dla różnych wartości dla parametrów: `M`, `m` oraz `n`. Ich wybór jest przedstawiony w metodzie `main`.

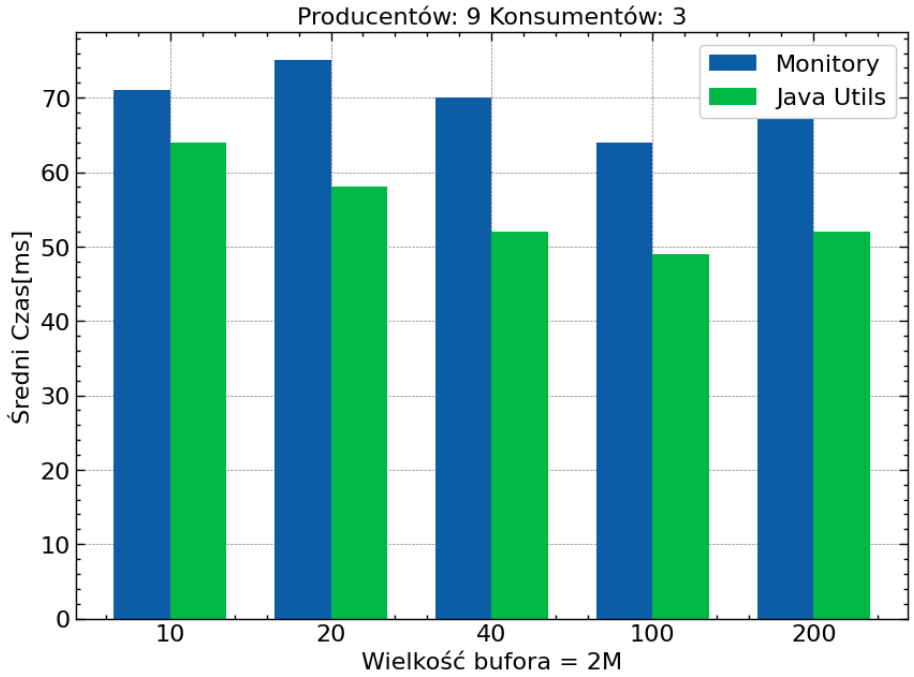
```
public static void main(String[] args) {
    int amountOfRuns = 15;

    List<Integer> ms = List.of(3, 9, 3, 12, 20, 1, 20);
    List<Integer> ns = List.of(3, 3, 9, 15, 20, 20, 1);
    List<Integer> Ms = List.of(5, 10, 20, 50, 100);

    System.out.println("M,m,n,avgTimeUsingMonitors,avgTimeUsingUtils");
    for (int i = 0; i < ms.size(); i++) {
        for (var M : Ms) {
            var m = ms.get(i);
            var n = ns.get(i);

            Supplier<Buffer> bufferSupplierUsingMonitors = () -> new
            ↪ BufferUsingMonitors(M, m, n);
            Supplier<Buffer> bufferSupplierUsingUtils = () -> new BufferUsingUtils(M, m,
            ↪ n);
            long avgTimeUsingMonitors = avgTestRun(amountOfRuns, M, m, n,
            ↪ bufferSupplierUsingMonitors);
            long avgTimeUsingUtils = avgTestRun(amountOfRuns, M, m, n,
            ↪ bufferSupplierUsingUtils);
            System.out.println(M + ", " + m + ", " + n + ", " +
                               avgTimeUsingMonitors + ", " + avgTimeUsingUtils);
        }
    }
}
```

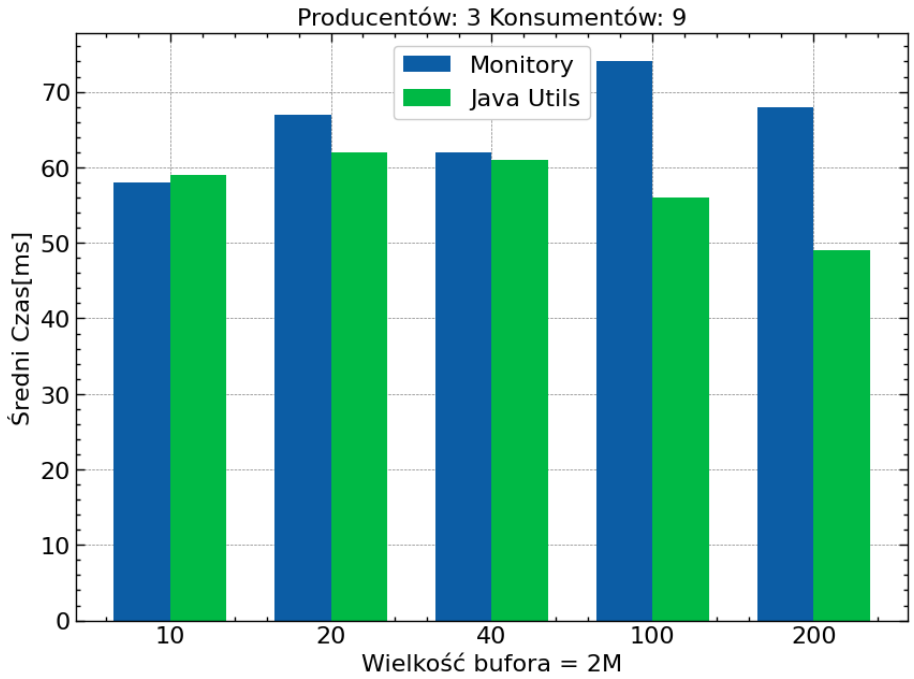
Program zwraca wyniki w postaci tabeli na standardowe wyjście.

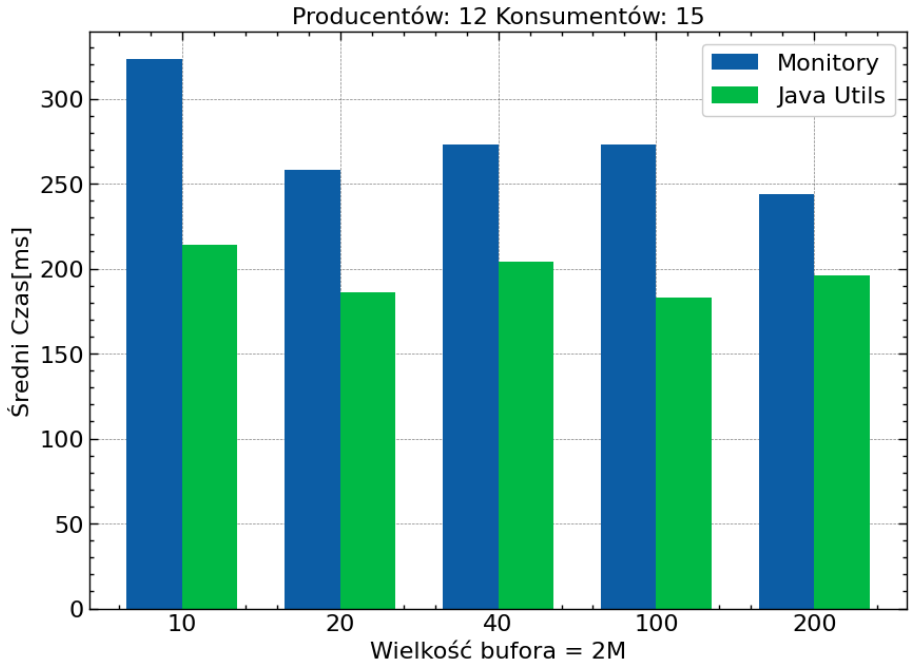
Wyniki

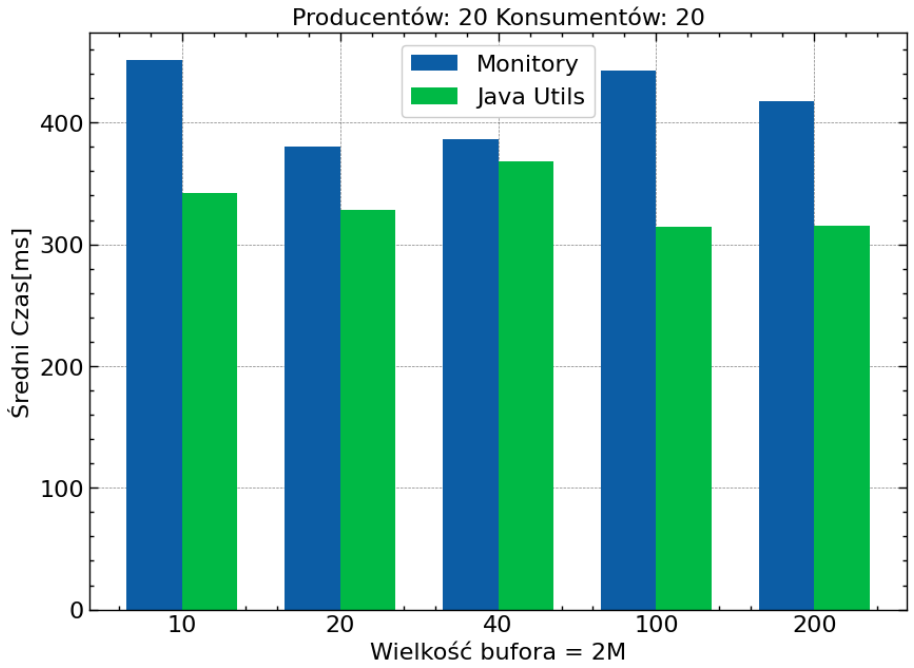
Wyniki przeprowadzonych testów zostały przedstawione na wykresach. Każdy wykres przedstawia zależność czasu wykonania od rozmiaru bufora, czyli `2M`.

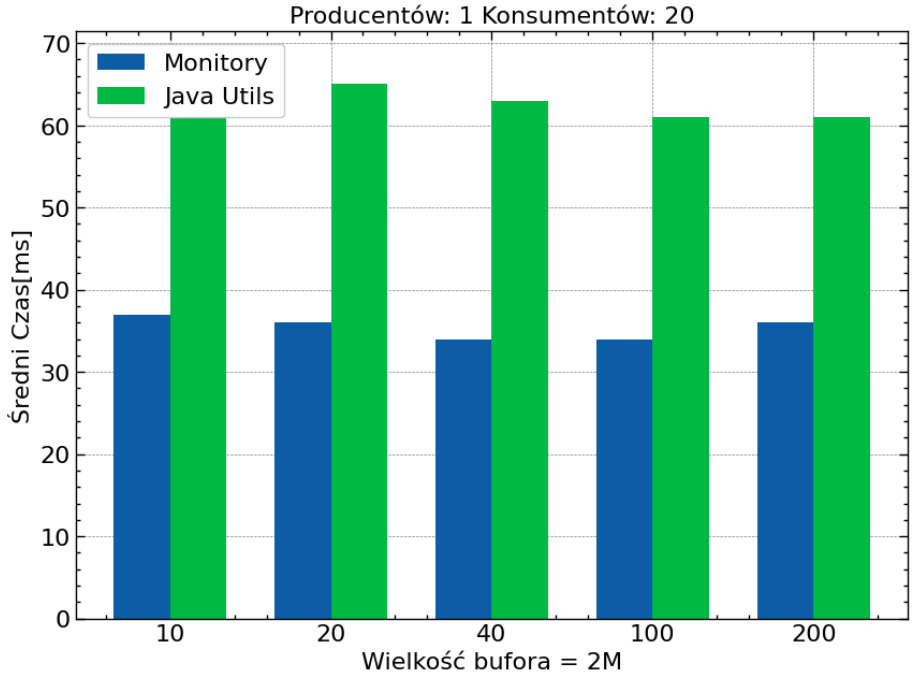

Producentów: 3 Konsumentów: 3
Monitory
Java Utils
Średni Czas[ms]
0
10
20
30
40
50
10
20
40
100
200
Wielkość bufora = 2M

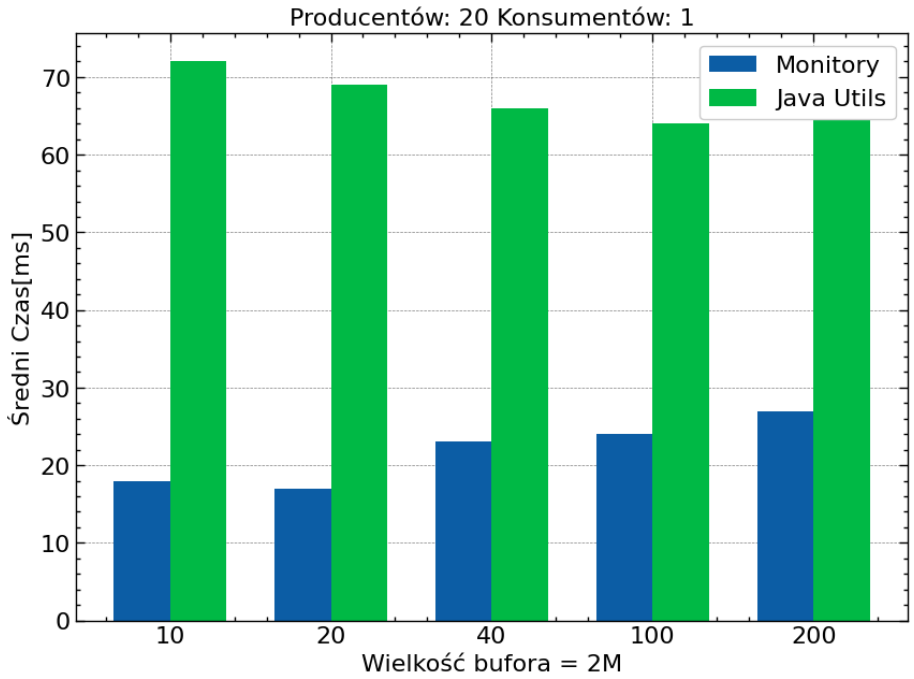

Producentów: 9 Konsumentów: 3
Monitory
Java Utils
Średni Czas[ms]
0
10
20
30
40
50
60
70
10
20
40
100
200
Wielkość bufora = 2M


Producentów: 3 Konsumentów: 9
Monitory
Java Utils
Średni Czas[ms]
0
10
20
30
40
50
60
70
10
20
40
100
200
Wielkość bufora = 2M


Producentów: 12 Konsumentów: 15
Monitory
Java Utils
Średni Czas[ms]
0
50
100
150
200
250
300
10
20
40
100
200
Wielkość bufora = 2M


Producentów: 20 Konsumentów: 20
Monitory
Java Utils
Średni Czas[ms]
0
100
200
300
400
10
20
40
100
200
Wielkość bufora = 2M


Producentów: 1 Konsumentów: 20
Monitory
Java Utils
Średni Czas[ms]
0
10
20
30
40
50
60
70
10
20
40
100
200
Wielkość bufora = 2M



Widzimy, że wersja bufora korzystająca z `Java Concurrency Utilities` jest szybsza niż wersja korzystająca z `monitorów` w większości przypadków. Przyśpieszenie zapewne wynika z faktu, że `Condition` pozwala na bardziej precyzyjne kontrolowanie przebudzeń wątków. To co jest ciekawe, to fakt, że w skrajnych przydatkach, gdzie mamy zaledwie jednego producenta lub konsumenta, to wersja korzystająca z monitorów jest szybsza. Oznacza to, że badając wydajność współbieżnych programów, warto zwrócić uwagę na skrajne przypadki, jeżeli uważamy, że mogą wystąpić w naszym programie.

Warto zwórócić uwagę, na to że biblioteka `Java Concurrency Utilities` zawiera wiele innych mechanizmów, które nie zostały użyte w tym zadaniu. Możliwe, że zastosowanie innych mechanizmów pozwoliłoby na jeszcze większe przyspieszenie.

Bibliografia

- Bill Venners: *Inside the Java Virtual Machine Chapter 20*
- Java Concurrency Utilities
- Java Concurrency Utilities - Condition
- Java Concurrency Utilities - ReentrantLock