Lab 04

Java Concurrency Utilities

Adrian Madej 23.10.2023

1. Treść zadań

Producenci i konsumenci z losowa ilością pobieranych i wstawianych porcji

- 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
- Przeprowadzić porównanie wydajności (np. czas wykonywania) vs. różne parametry, zrobić wykresy i je skomentować

2. Rozwiązanie zadań

Koncepcja

W celu rozwiązania zadania zaimplementujemy program przy pomocy monitorów Javy oraz mechanizmów Java Concurrency. Dla każdej konfiguracji przeprowadzimy 200 iteracji a wynikiem zostanie średni czas działania programu dla danych parametrów. Postąpimy w ten sposób, ponieważ program w dużej mierze będzie zależał od losowości parametrów. Każdy wątek wykona po 100 operacji dodania i pobrania z bufora.

Przy implementacji za pomocą monitora, wątki producentów będą wstrzymywane gdy w buforze pojawi się zbyt wiele elementów, z kolei wątki konsumentów - gdy elementów będzie zbyt mało.

Przy użyciu biblioteki Java Concurrency wykorzystana zostanie biblioteka blokująca oraz pula wątków.

Poszczególne czasy zostaną zapisane do pliku result.csv.

Implementacja

Najpierw utworzymy interfejsy, które ułatwią nam pisanie programu

Interfejs IBuffer

```
public interface IBuffer {
    void unregisterProducer();
    void unregisterConsumer();

    int maxSize();

    void put(int[] products) throws InterruptedException;

    void get(List<Integer> results, int howMany) throws
InterruptedException;

    boolean isAnySideInterested();
}
```

Interfejs IExecutor

```
public interface IExecutor {
    void submit(Thread t);

    void shutdownNow();

    void awaitTermination();
}
```

Executor zakończy pracę wątków jeśli jedna ze stron skończy swoje zadania, tzn. konsumenci lub producenci.

Klasa Consumer

```
class Consumer extends Thread {
    private final IBuffer buffer;
    private final int limit;
    private final int iterations;
    private final IExecutor executor;
    private final Random random = new Random();
    public Consumer(IExecutor executor, IBuffer buffer, int
iterations) {
        this.iterations = iterations;
```

```
        this.buffer = buffer;
        limit = buffer.maxSize() / 2;
        this.executor = executor;
    }

    public void run() {
        for (int i = 0; i < iterations; i++) {
            var howMany = Math.max(1, random.nextInt(limit)-
1);
            List<Integer> results = new LinkedList<>();
            try {
                buffer.get(results, howMany);
            } catch (InterruptedException exception) {
                break;
            }
        }

        buffer.unregisterConsumer();
        if (!buffer.isAnySideInterested()) {
            executor.shutdownNow();
        }
    }
}
```

Klasa Producer

```
class Producer extends Thread {
    private final IBuffer buffer;
    private final Random random = new Random();
    private final int limit;
    private final int iterations;
    private final IExecutor executor;

    public Producer(IExecutor executor, IBuffer buffer, int
iterations) {
        this.iterations = iterations;
        this.buffer = buffer;
        this.limit = buffer.maxSize() / 2;
        this.executor = executor;
    }

    public void run() {
        for(int i = 0; i < iterations; i++) {
            var howMany = Math.max(1, random.nextInt(limit)-
1);
            int[] data = new int[howMany];
            IntStream.range(0, howMany).forEach(j -> {
                data[j] = j;
            });

            try {
```

```
                buffer.put(data);
            } catch (InterruptedException exception) {
                break;
            }
        }
        buffer.unregisterProducer();
        if(!buffer.isAnySideInterested()){
            executor.shutdownNow();
        }
    }
}
```

Klasa MBuffer (wykorzystywana przy monitorze)

```
class MBuffer implements IBuffer {
    private final LinkedList<Integer> buffer = new
LinkedList<>();
    private final int maxBufferSize;
    private final Object lock = new Object();
    private Integer producersRunning;
    private Integer consumersRunning;

    public MBuffer(int size, int m, int n) {
        maxBufferSize = size * 2;
        producersRunning = m;
        consumersRunning = n;
    }

    @Override
    public void unregisterProducer() {
        synchronized (lock) {
            producersRunning--;
        }
    }

    @Override
    public void unregisterConsumer() {
        synchronized (lock) {
            consumersRunning--;
        }
    }

    @Override
    public boolean isAnySideInterested() {
        return producersRunning > 0 && consumersRunning > 0;
    }

    @Override
    public int maxSize() {
        return maxBufferSize;
    }
```

```
    @Override
    public synchronized void put(int[] products) throws
InterruptedException {
        while (isAnySideInterested() && (buffer.size() +
products.length >= maxBufferSize)) {
            wait();
        }

        if (!isAnySideInterested()) {
            return;
        }


buffer.addAll(Arrays.stream(products).boxed().collect(Collecto
rs.toList()));
        notifyAll();
    }

    @Override
    public synchronized void get(List<Integer> results, int
howMany) throws InterruptedException {
        while (isAnySideInterested() && buffer.size() <
howMany) {
            wait();
        }

        if (!isAnySideInterested()) {
            return;
        }

        var sublist = buffer.subList(0, howMany);
        for(int i = 0; i < howMany; i++){
            var v = sublist.get(i);
            results.add(v);
        }
        sublist.clear();
        notifyAll();
    }
}
```

Klasa MExecutor (wykorzystywana przy monitorze)

```
public class MExecutor implements IExecutor {
    private final List<Thread> workers = new LinkedList<>();

    @Override
    public void submit(Thread t){
        workers.add(t);
    }
```

```
    @Override
    public void shutdownNow(){
        workers.forEach(Thread::interrupt);
    }

    @Override
    public void awaitTermination() {
        workers.forEach(Thread::start);

        workers.forEach(t -> {
            try {
                t.join();
            } catch (InterruptedException ignored) {
            }
        });
    }
}
```

Klasa CBuffer (wykorzystywana przy Java Concurrency)

```
class CBuffer implements IBuffer {
    private final int maxBufferSize;
    private final AtomicInteger producersRunning;
    private final AtomicInteger consumersRunning;
    private final BlockingQueue<Integer> buffer;

    public CBuffer(int size, int m, int n) {
        maxBufferSize = size * 2;
        producersRunning = new AtomicInteger(m);
        consumersRunning = new AtomicInteger(n);
        buffer = new ArrayBlockingQueue<>(maxBufferSize);
    }

    @Override
    public void unregisterProducer() {
        producersRunning.decrementAndGet();
    }

    @Override
    public void unregisterConsumer() {
        consumersRunning.decrementAndGet();
    }

    @Override
    public boolean isAnySideInterested() {
        return producersRunning.get() > 0 &&
consumersRunning.get() > 0;
    }

    @Override
    public int maxSize() {
```

```
        return maxBufferSize;
    }

    @Override
    public void put(int[] products) throws
InterruptedException {
        for (var v : products) {
            if(!isAnySideInterested()){
                return;
            }
            buffer.put(v);
        }
    }

    @Override
    public void get(List<Integer> results, int howMany) throws
InterruptedException {

        while (results.size() < howMany) {
            if(!isAnySideInterested()){
                return;
            }
            Integer v = buffer.take();
            results.add(v);
        }
    }
}
```

Klasa CExecutor (wykorzystywana przy Java Concurrency)

```
public class CExecutor implements IExecutor {
    private final ExecutorService executorService;
    private final List<Thread> workers = new LinkedList<>();

    public CExecutor(int threadNum) {
        executorService =
Executors.newFixedThreadPool(threadNum);
    }

    @Override
    public void submit(Thread t) {
        workers.add(t);
    }

    @Override
    public void shutdownNow() {
        executorService.shutdownNow();
    }

    @Override
    public void awaitTermination() {
```

```
        try {
            workers.forEach(executorService::submit);
        } catch (RejectedExecutionException ignored){

        }

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

Klasa PKMain

```
public class PKMain {
    enum MODE {
        MONITORS,
        JAVA_CONCURRENT
    }

    public static void main(String[] args) {
        List<Integer> producers = List.of(3, 10, 20);
        List<Integer> consumers = List.of(3, 10, 20);
        List<Integer> bufferSizes = List.of(5, 10, 50, 100);

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

        for (Integer p : producers) {
            for (Integer c : consumers) {
                for (Integer bs : bufferSizes) {
                    double avgM = test(p, c, bs,
MODE.MONITORS);
                    double avgJC = test(p, c, bs,
MODE.JAVA_CONCURRENT);

                    System.out.println(p + "p/" + c + "c [" +
bs + "]: " + avgM + "ms / " + avgJC + "ms");
                    results.add(p + ", " + c + ", " + bs + ",
" + round(avgM, 2) + ", " + round(avgJC, 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;
    }

    public static double test(int prodNum, int consNum, int
halfBufSize, MODE mode) {
        var times = new LinkedList<Long>();

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

            if (mode == MODE.MONITORS) {
                testMonitors(prodNum, consNum, halfBufSize);
            } else {
                testJavaConcurrent(prodNum, consNum,
halfBufSize);
            }

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

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

    public static void testJavaConcurrent(int m, int n, int M)
{
        var buffer = new CBuffer(M, m, n);
        var executor = new CExecutor(n+m);

        runTasks(m, n, buffer, executor);
    }

    public static void testMonitors(int m, int n, int M) {
        var buffer = new MBuffer(M, m, n);
        var executor = new MExecutor();

        runTasks(m, n, buffer, executor);
    }

    public static void runTasks(int m, int n, IBuffer buffer,
IExecutor executor){
        IntStream.range(0, n).forEach(i ->
```

```
                executor.submit(
                        new Consumer(executor, buffer, 100)
                ));
        IntStream.range(0, m).forEach(i ->
                executor.submit(
                        new Producer(executor, buffer, 100)
                ));

        executor.awaitTermination();
    }
}
```

3. Wyniki

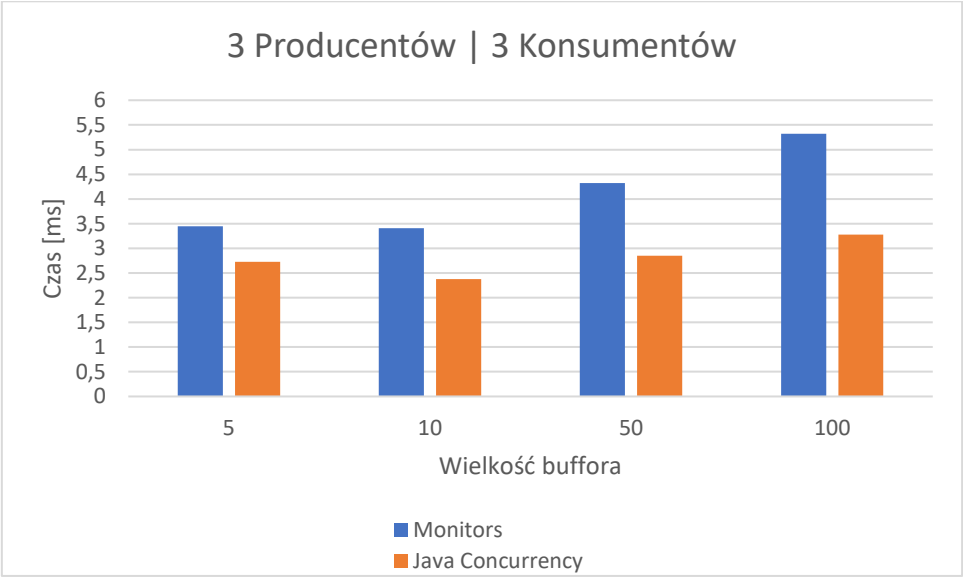
W wyniku uruchomienia programu tworzył się nam plik results.csv.

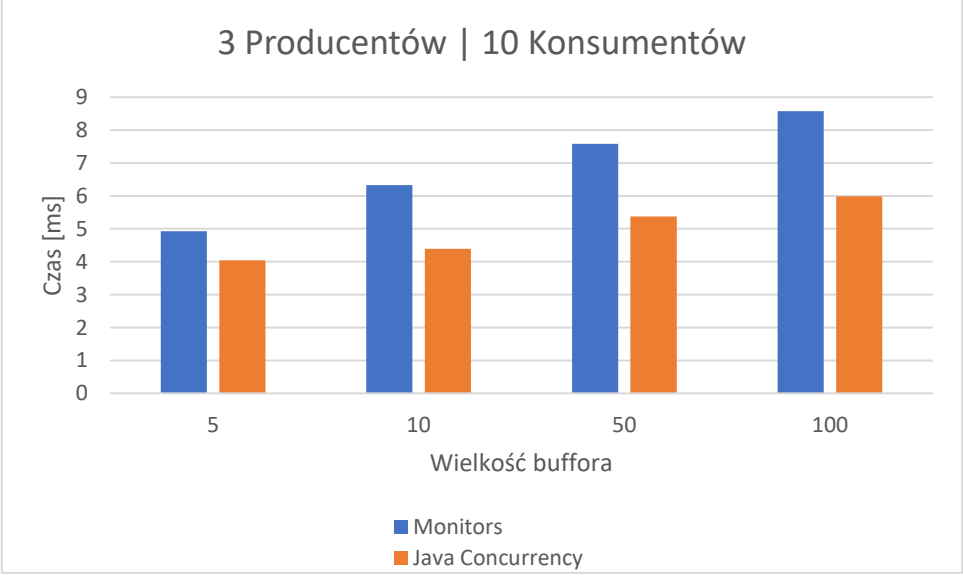
Zapisane dane przedstawię w postaci tabeli:

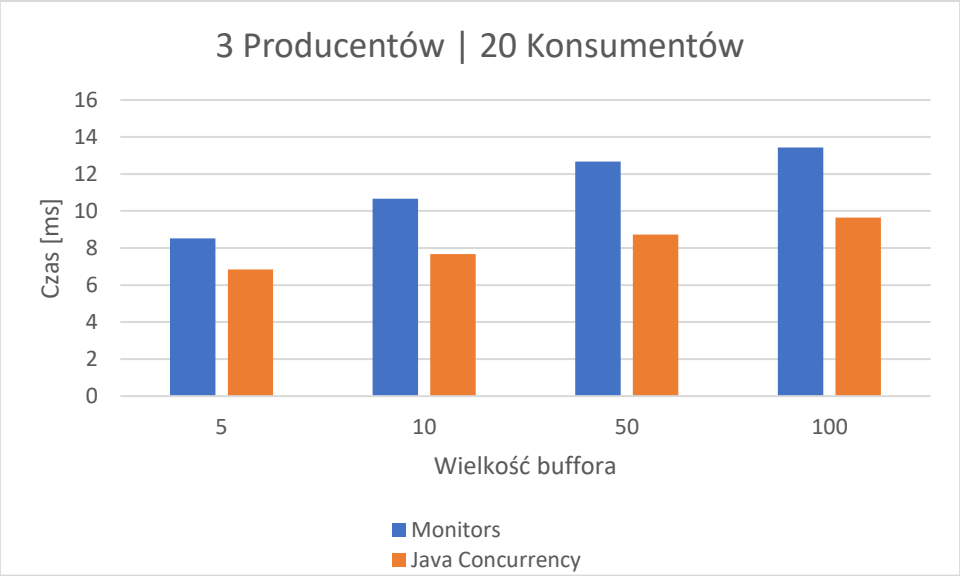
Nb of Producers	Nb of Consumers	Buffer Size	avgMonitor [ms]	avgJavaCourrency [ms]
3	3	5	3,45	2,73
3	3	10	3,41	2,38
3	3	50	4,32	2,85
3	3	100	5,32	3,28
3	10	5	4,93	4,04
3	10	10	6,33	4,39
3	10	50	7,58	5,37
3	10	100	8,57	5,99
3	20	5	8,52	6,85
3	20	10	10,66	7,68
3	20	50	12,67	8,73
3	20	100	13,43	9,65
10	3	5	4,05	3,55
10	3	10	5,36	4,06
10	3	50	6,99	5,28
10	3	100	7,47	6,04
10	10	5	12,24	7,87
10	10	10	16,7	8,92
10	10	50	20,94	10,76
10	10	100	25,28	13,07
10	20	5	17,9	10,99
10	20	10	23,01	12,85
10	20	50	28,0	15,44
10	20	100	30,85	18,11

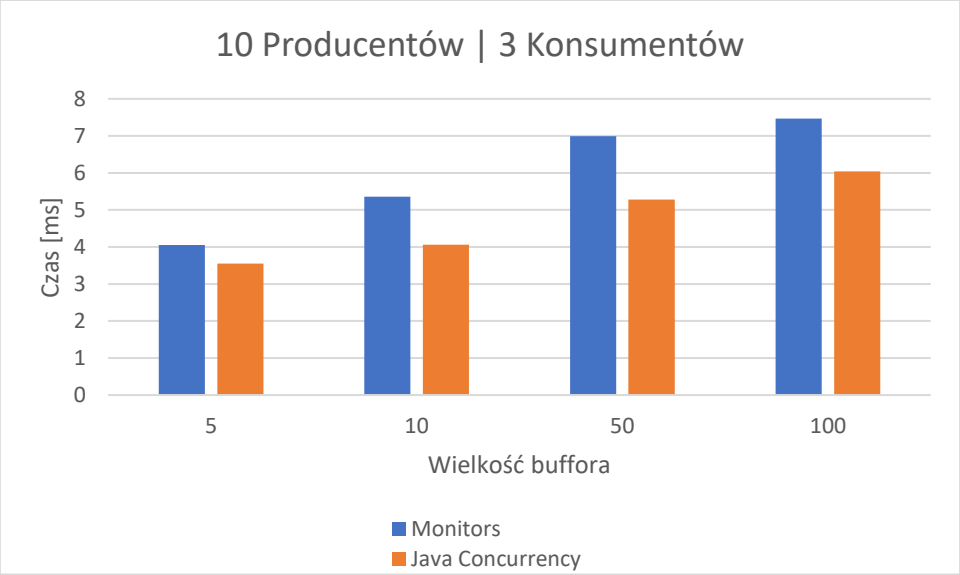
20	3	5	4,8	4,81
20	3	10	6,82	6,39
20	3	50	8,8	7,98
20	3	100	9,02	9,66
20	10	5	14,21	10,19
20	10	10	17,75	12,69
20	10	50	25,75	15.56
20	10	100	29,47	18,02
20	20	5	25,09	16,16
20	20	10	32,87	18,69
20	20	50	43,7	23,12
20	20	100	53,07	27,16

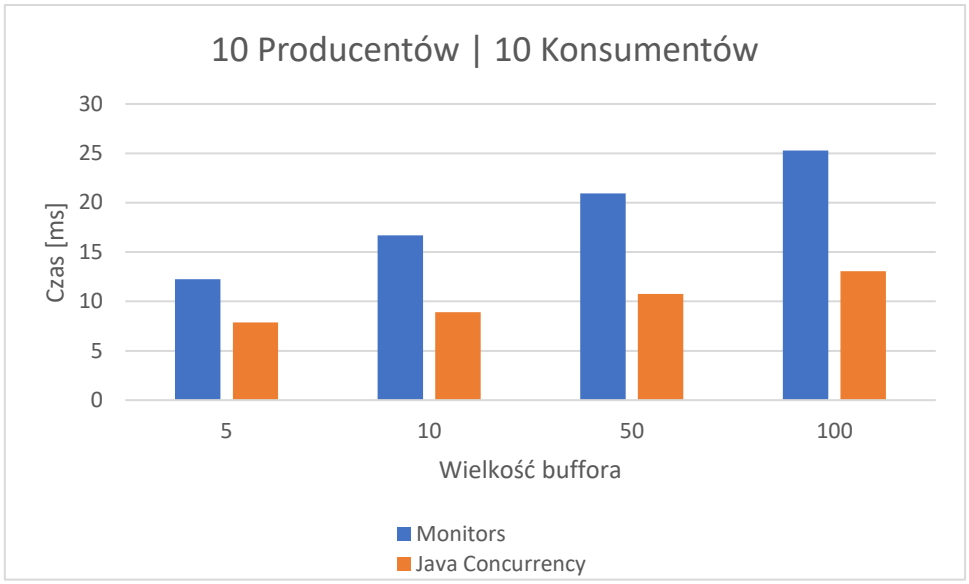
Na podstawie powyżej tabeli tworzymy wykresy:

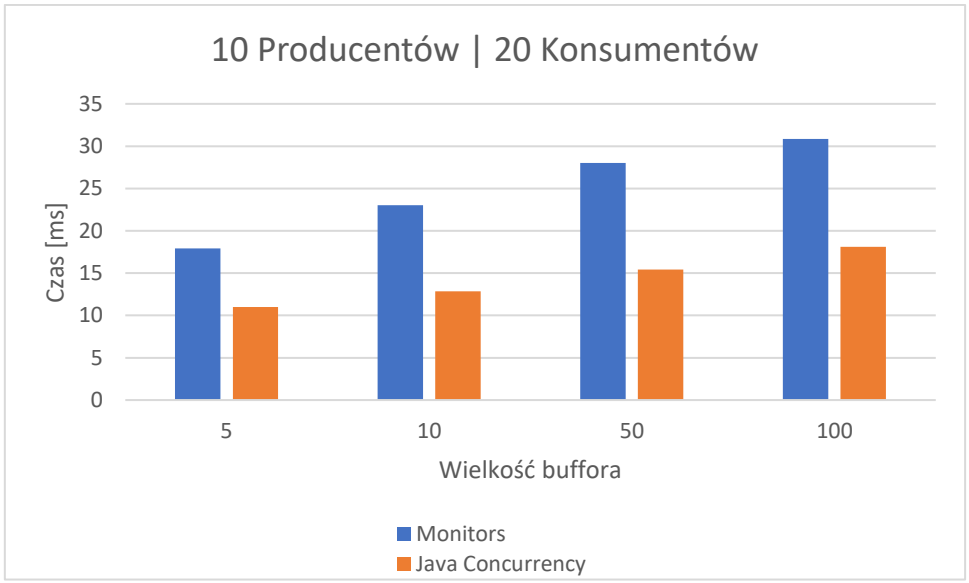


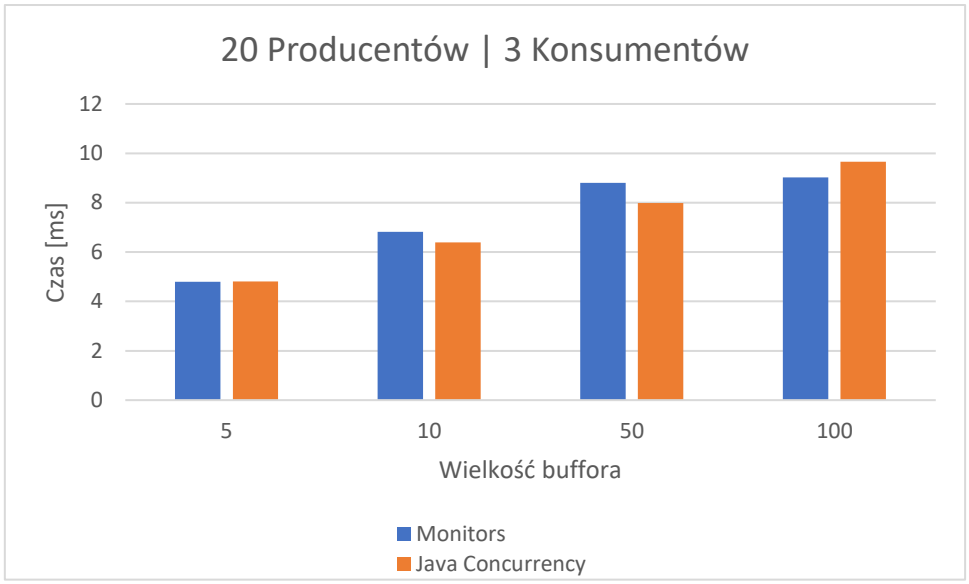


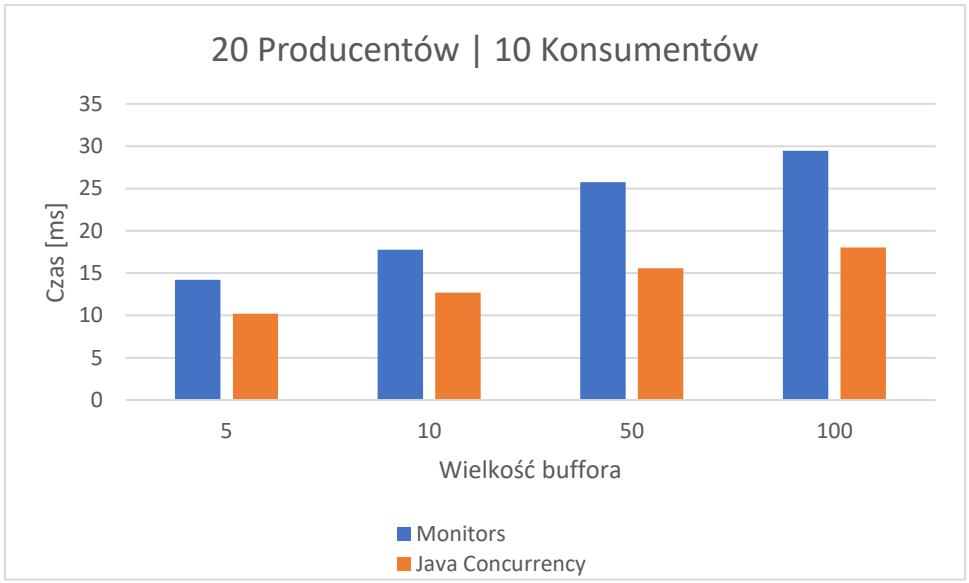

3 Producentów | 20 Konsumentów
16
14
12
10
8
6
4
2
0
Czas [ms]
5
10
50
100
Wielkość buffora
Monitors
Java Concurrency

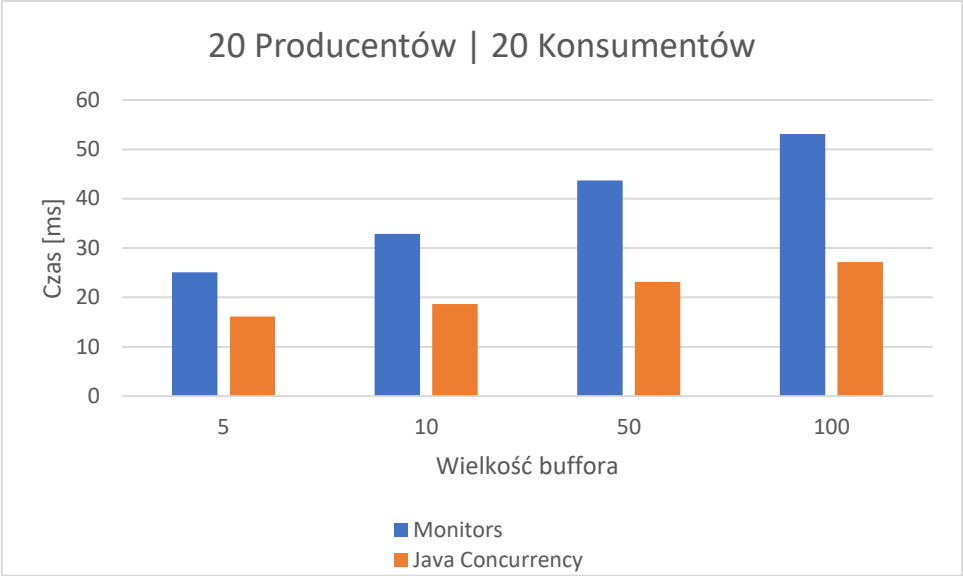

10 Producentów | 3 Konsumentów
8
7
6
5
4
3
2
1
0
Czas [ms]
5
10
50
100
Wielkość buffora
Monitors
Java Concurrency


10 Producentów | 10 Konsumentów
30
25
20
15
10
5
0
Czas [ms]
5
10
50
100
Wielkość buffora
Monitors
Java Concurrency


10 Producentów | 20 Konsumentów
35
30
25
20
15
10
5
0
Czas [ms]
5
10
50
100
Wielkość buffora
Monitors
Java Concurrency


20 Producentów | 3 Konsumentów
12
10
8
6
4
2
0
Czas [ms]
5
10
50
100
Wielkość buffora
Monitors
Java Concurrency


20 Producentów | 10 Konsumentów
35
30
25
20
15
10
5
0
Czas [ms]
5
10
50
100
Wielkość buffora
Monitors
Java Concurrency



Na podstawie powyższych wykresów można zauważyć że implementacja przy pomocy monitorów jest mniej wydajna niż implementacja przy użyciu biblioteki Java Concurrency. Objawia się to w czasie działania. Jedynie w momencie gdy liczba producentów jest znacznie większa od liczby konsumentów te czasy są zbliżone.

4. Wnioski

- Java Concurrency Utilities stanowią potężne narzędzia do rozwiązywania problemów wielowątkowych i mogą być skutecznie wykorzystane do implementacji systemu producentów i konsumentów.
- Java Concurrency Utilities oferują zoptymalizowane mechanizmy, które pozwalają na wydajne zarządzanie wątkami. To pozwala na optymalne wykorzystanie dostępnych zasobów i zminimalizowanie narzutu wydajnościowego związanego z wielowątkowym kodem.
- Monitor to koncepcja w Javie, która pozwala na synchronizację dostępu do współdzielonych zasobów przez wiele wątków. Jest to przydatny mechanizm w problemie producentów i konsumentów, ponieważ pozwala na kontrolowanie dostępu do bufora.
- Dobre rozwiązania wielowątkowe wymagają testowania i analizy, aby zoptymalizować ich wydajność i zachowanie. Dlatego ważne jest przeprowadzenie dokładnych testów i analizy wyników.

5. Bibliografia

- Materiały do laboratorium
- Java Concurrency - Baeldung
- Java Concurrency and Multithreading
- Dokumentacja Javy