

Teoria Współbieżności - Sprawozdanie 4

Maciej Brzeżawski

Listopad 2024

1 Treść zadania

Producenci i konsumenci z losową ilością pobieranych i wstawianych porcji

Dany jest bufor, do którego producent może wkładać dane, a konsument pobierać. Napisać program, który zorganizuje takie działanie producenta i konsumenta, w którym zapewniona będzie własność bezpieczeństwa i żywotności.

- Bufor o rozmiarze $2M$
- Jest m producentów i n konsumentów
- Producent wstawia do bufora losową liczbę elementów (nie więcej niż M)
- Konsument pobiera losową 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 Implementacja kodu

2.1 Klasa Main

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        int bufferSize = 100; // Set 2 * M (M can be 50)
        int maxItems = 50; // Maximum number of items produced/consumed at once
        int producerCount = 5; // Number of producers
        int consumerCount = 5; // Number of consumers

        Buffer buffer = new Buffer(bufferSize);

        ArrayList<Thread> threads = new ArrayList<>();

        long startTime = System.currentTimeMillis();

        // Start producer threads
        for (int i = 0; i < producerCount; i++) {
            Thread producerThread = new Thread(new Producer(buffer, maxItems));
            threads.add(producerThread);
            producerThread.start();
        }

        // Start consumer threads
        for (int i = 0; i < consumerCount; i++) {
            Thread consumerThread = new Thread(new Consumer(buffer, maxItems));
            threads.add(consumerThread);
            consumerThread.start();
        }

        // Let the simulation run for a set period (e.g., 10 seconds)
        Thread.sleep(10000);

        long endTime = System.currentTimeMillis();
        System.out.println("Execution Time: " + (endTime - startTime) + " ms");

        // Stop all threads
        for (Thread thread : threads) {
            thread.interrupt();
        }
    }
}
```

2.2 Klasa Buffer

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

public class Buffer {
    private final int maxSize;
    private final LinkedList<Integer> list = new LinkedList<>();
    private final Lock lock = new ReentrantLock();
    private final Condition notFull = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();

    public Buffer(int maxSize) {
        this.maxSize = maxSize;
    }

    public void produce(int items) throws InterruptedException {
        lock.lock();
        try {
            while (list.size() + items > maxSize) {
                notFull.await();
            }
            for (int i = 0; i < items; i++) {
                list.add(1); // Adding dummy data
            }
            System.out.println("Produced: " + items);
            notEmpty.signalAll();
        } finally {
            lock.unlock();
        }
    }

    public void consume(int items) throws InterruptedException {
        lock.lock();
        try {
            while (list.size() < items) {
                notEmpty.await();
            }
            for (int i = 0; i < items; i++) {
                list.poll();
            }
            System.out.println("Consumed: " + items);
            notFull.signalAll();
        }
    }
}
```

```

        } finally {
            lock.unlock();
        }
    }
}

```

2.3 Klasa Consumer

```

import java.util.concurrent.ThreadLocalRandom;

public class Consumer implements Runnable {
    private final Buffer buffer;
    private final int maxItems;

    public Consumer(Buffer buffer, int maxItems) {
        this.buffer = buffer;
        this.maxItems = maxItems;
    }

    @Override
    public void run() {
        try {
            while (true) {
                int items = ThreadLocalRandom.current().nextInt(1, maxItems + 1);
                buffer.consume(items);
                Thread.sleep(ThreadLocalRandom.current().nextInt(100));
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

```

2.4 Klasa Producer

```
import java.util.concurrent.ThreadLocalRandom;

public class Producer implements Runnable {
    private final Buffer buffer;
    private final int maxItems;

    public Producer(Buffer buffer, int maxItems) {
        this.buffer = buffer;
        this.maxItems = maxItems;
    }

    @Override
    public void run() {
        try {
            while (true) {
                int items = ThreadLocalRandom.current().nextInt(1, maxItems + 1);
                buffer.produce(items);
                Thread.sleep(ThreadLocalRandom.current().nextInt(100));
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

```
... Consumed: 31
Produced: 29
Produced: 2
Consumed: 31
Consumed: 1
Consumed: 4
Consumed: 3
Consumed: 14
Consumed: 7
Consumed: 9
Produced: 47
Produced: 7
Consumed: 2
Produced: 15
Consumed: 14
Consumed: 40
Produced: 47
Consumed: 20
Produced: 23
Consumed: 35
Produced: 40
Consumed: 22
Produced: 23
Consumed: 33
Produced: 31
Consumed: 50
Produced: 45
Consumed: 22
Produced: 21
Consumed: 49
Produced: 48
Produced: 5
Consumed: 11
Produced: 14
Consumed: 8
Consumed: 29
Produced: 24
Consumed: 18
```

tw3 > src > main > java > org > example > Producer > run

Rysunek 1: Fragment wyniku programu

3 Porównanie wydajności: czas wykonania i różne parametry

3.1 Implementacja nowej klasy Main

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        int[] producerCounts = {2, 5, 10};
        int[] consumerCounts = {2, 5, 10};
        int[] bufferSize = {50, 100, 200};
        int[] maxItemsList = {10, 25, 50};

        for (int m : producerCounts) {
            for (int n : consumerCounts) {
                for (int bufferSize : bufferSize) {
                    for (int maxItems : maxItemsList) {
                        Buffer buffer = new Buffer(bufferSize);
                        ArrayList<Thread> threads = new ArrayList<>();

                        long startTime = System.currentTimeMillis();

                        for (int i = 0; i < m; i++) {
                            threads.add(new Thread(new Producer(buffer, maxItems)));
                        }

                        for (int i = 0; i < n; i++) {
                            threads.add(new Thread(new Consumer(buffer, maxItems)));
                        }

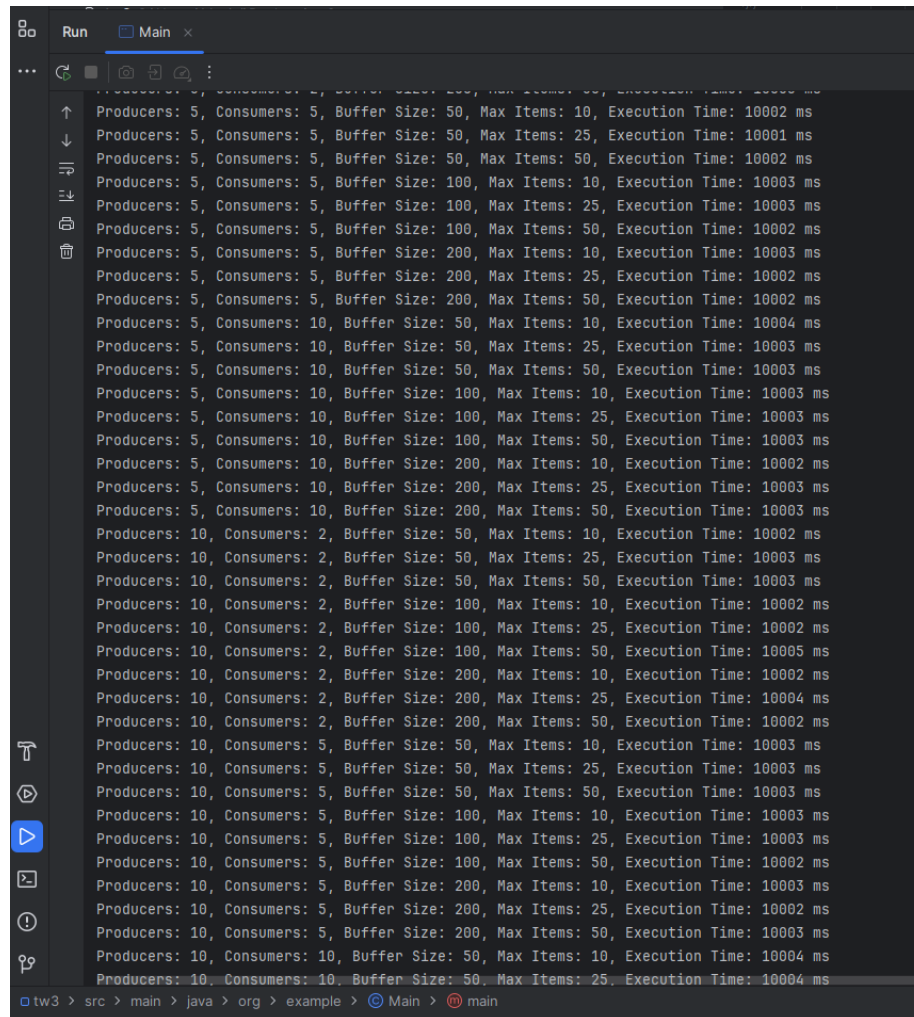
                        threads.forEach(Thread::start);
                        Thread.sleep(10000);

                        long endTime = System.currentTimeMillis();
                        long executionTime = endTime - startTime;

                        System.out.printf("Producers: %d, " +
                                         "Consumers: %d, " +
                                         "Buffer Size: %d, " +
                                         "Max Items: %d, " +
                                         "Execution Time: %d ms%n",
                                         m, n, bufferSize, maxItems, executionTime);

                        threads.forEach(Thread::interrupt); ...
                    }
                }
            }
        }
    }
}
```

3.2 Wyniki porównania wydajności

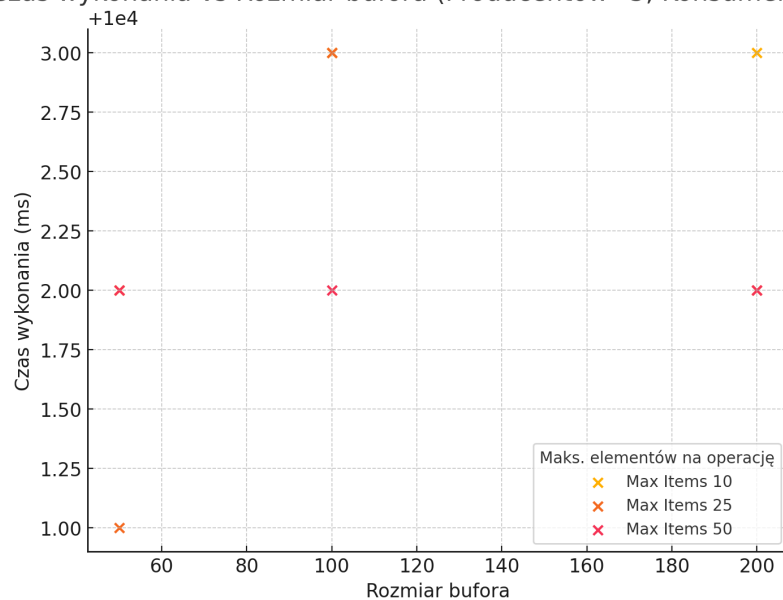


```
Run Main x
Producers: 5, Consumers: 5, Buffer Size: 50, Max Items: 10, Execution Time: 10002 ms
Producers: 5, Consumers: 5, Buffer Size: 50, Max Items: 25, Execution Time: 10001 ms
Producers: 5, Consumers: 5, Buffer Size: 50, Max Items: 50, Execution Time: 10002 ms
Producers: 5, Consumers: 5, Buffer Size: 100, Max Items: 10, Execution Time: 10003 ms
Producers: 5, Consumers: 5, Buffer Size: 100, Max Items: 25, Execution Time: 10003 ms
Producers: 5, Consumers: 5, Buffer Size: 100, Max Items: 50, Execution Time: 10002 ms
Producers: 5, Consumers: 5, Buffer Size: 200, Max Items: 10, Execution Time: 10003 ms
Producers: 5, Consumers: 5, Buffer Size: 200, Max Items: 25, Execution Time: 10002 ms
Producers: 5, Consumers: 5, Buffer Size: 200, Max Items: 50, Execution Time: 10002 ms
Producers: 5, Consumers: 10, Buffer Size: 50, Max Items: 10, Execution Time: 10004 ms
Producers: 5, Consumers: 10, Buffer Size: 50, Max Items: 25, Execution Time: 10003 ms
Producers: 5, Consumers: 10, Buffer Size: 50, Max Items: 50, Execution Time: 10003 ms
Producers: 5, Consumers: 10, Buffer Size: 100, Max Items: 10, Execution Time: 10003 ms
Producers: 5, Consumers: 10, Buffer Size: 100, Max Items: 25, Execution Time: 10003 ms
Producers: 5, Consumers: 10, Buffer Size: 100, Max Items: 50, Execution Time: 10003 ms
Producers: 5, Consumers: 10, Buffer Size: 200, Max Items: 10, Execution Time: 10002 ms
Producers: 5, Consumers: 10, Buffer Size: 200, Max Items: 25, Execution Time: 10003 ms
Producers: 5, Consumers: 10, Buffer Size: 200, Max Items: 50, Execution Time: 10003 ms
Producers: 10, Consumers: 2, Buffer Size: 50, Max Items: 10, Execution Time: 10002 ms
Producers: 10, Consumers: 2, Buffer Size: 50, Max Items: 25, Execution Time: 10003 ms
Producers: 10, Consumers: 2, Buffer Size: 50, Max Items: 50, Execution Time: 10003 ms
Producers: 10, Consumers: 2, Buffer Size: 100, Max Items: 10, Execution Time: 10002 ms
Producers: 10, Consumers: 2, Buffer Size: 100, Max Items: 25, Execution Time: 10002 ms
Producers: 10, Consumers: 2, Buffer Size: 100, Max Items: 50, Execution Time: 10005 ms
Producers: 10, Consumers: 2, Buffer Size: 200, Max Items: 10, Execution Time: 10002 ms
Producers: 10, Consumers: 2, Buffer Size: 200, Max Items: 25, Execution Time: 10004 ms
Producers: 10, Consumers: 2, Buffer Size: 200, Max Items: 50, Execution Time: 10002 ms
Producers: 10, Consumers: 5, Buffer Size: 50, Max Items: 10, Execution Time: 10003 ms
Producers: 10, Consumers: 5, Buffer Size: 50, Max Items: 25, Execution Time: 10003 ms
Producers: 10, Consumers: 5, Buffer Size: 50, Max Items: 50, Execution Time: 10003 ms
Producers: 10, Consumers: 5, Buffer Size: 100, Max Items: 10, Execution Time: 10003 ms
Producers: 10, Consumers: 5, Buffer Size: 100, Max Items: 25, Execution Time: 10003 ms
Producers: 10, Consumers: 5, Buffer Size: 100, Max Items: 50, Execution Time: 10002 ms
Producers: 10, Consumers: 5, Buffer Size: 200, Max Items: 10, Execution Time: 10003 ms
Producers: 10, Consumers: 5, Buffer Size: 200, Max Items: 25, Execution Time: 10002 ms
Producers: 10, Consumers: 5, Buffer Size: 200, Max Items: 50, Execution Time: 10003 ms
Producers: 10, Consumers: 10, Buffer Size: 50, Max Items: 10, Execution Time: 10004 ms
Producers: 10, Consumers: 10, Buffer Size: 50, Max Items: 25, Execution Time: 10004 ms
```

Rysunek 2: Fragment wyników programu

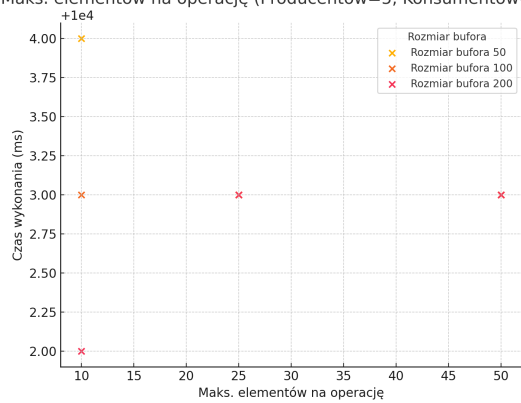
3.2.1 Wykresy w zależności od parametrów programu

Czas wykonania vs Rozmiar bufora (Producentów=5, Konsumentów=5)



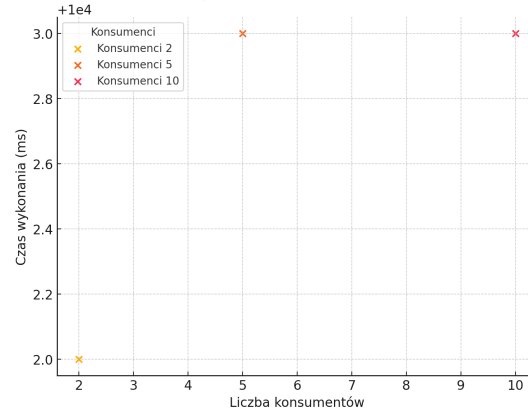
Rysunek 3: Czas wykonania vs Rozmiar bufora

Czas wykonania vs Maks. elementów na operację (Producentów=5, Konsumentów=10, Rozmiar bufora=100)



Rysunek 4: Czas wykonania vs Maks. elementów

Czas wykonania vs Liczba konsumentów (Producentów=10, Rozmiar bufora=100, Maks. elementów=25)



Rysunek 5: Czas wykonania vs Liczba konsumentów

4 Wnioski

Na podstawie przeprowadzonych wykresów możemy wyciągnąć kilka wniosków dotyczących wpływu poszczególnych parametrów na czas wykonania w modelu producent-konsument. Analiza przedstawia się następująco:

Wpływ rozmiaru bufora na czas wykonania (Producentów=5, Konsumentów=5)

- **Stabilność czasu wykonania:** Przy różnych rozmiarach bufora (50, 100, 200) czas wykonania jest względnie stabilny, niezależnie od maksymalnej liczby elementów przetwarzanych w jednej operacji.
- **Minimalne różnice:** Wzrost rozmiaru bufora przynosi niewielkie, ale zauważalne różnice, co może wynikać z mniejszej liczby operacji blokujących, gdy większy bufor zapobiega jego częstemu przepełnieniu się.
- **Wniosek:** Dla umiarkowanej liczby producentów i konsumentów (5 na 5) rozmiar bufora nie ma znaczącego wpływu na czas wykonania. System działa efektywnie, nawet przy mniejszym buforze.

Wpływ maksymalnej liczby elementów na operację na czas wykonania (Producentów=5, Konsumentów=10, Rozmiar bufora=100)

- **Niska zmienność:** Czas wykonania przy różnych maksymalnych ilościach elementów na operację (10, 25, 50) jest również bardzo stabilny.
- **Zarządzanie buforem:** Wyższa maksymalna liczba elementów na operację nie prowadzi do większych różnic w czasie wykonania, co sugeruje, że system dobrze zarządza buforem przy rosnącej liczbie jednoczesnych operacji.
- **Wniosek:** W przypadku większej liczby konsumentów (10) i umiarkowanej liczby producentów (5), zmiana liczby elementów na operację nie wpływa na czas wykonania, ponieważ bufor skutecznie zapobiega częstym operacjom blokowania.

Wpływ liczby konsumentów na czas wykonania (Producentów=10, Rozmiar bufora=100, Maks. elementów=25)

- **Stabilność dla rosnącej liczby konsumentów:** Wzrost liczby konsumentów przy stałym rozmiarze bufora i maksymalnej liczbie elementów na operację ma bardzo niewielki wpływ na czas wykonania.
- **Optymalne wykorzystanie zasobów:** System utrzymuje stabilny czas wykonania, co sugeruje, że większa liczba konsumentów nie prowadzi do większych opóźnień ani nie powoduje „wąskich gardeł” w procesie pobierania elementów z bufora.
- **Wniosek:** Dla większej liczby producentów (10) i zmiennej liczby konsumentów czas wykonania pozostaje stabilny, co świadczy o efektywności zarządzania procesami i buforem.

Podsumowanie ogólne

Z analizy wynika, że dla wszystkich testowanych konfiguracji system producent-konsument wykazuje stabilność czasu wykonania, co oznacza, że:

- **Rozmiar bufora, maksymalna liczba elementów na operację i liczba konsumentów** mają ograniczony wpływ na czas wykonania, o ile są utrzymywane w umiarkowanych wartościach.
- **System efektywnie zarządza konkurencją** między producentami i konsumentami, nawet przy wzroście liczby jednoczesnych operacji.

5 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.

William Stallings. *Operating Systems: Internals and Design Principles*. Pearson, 2018. ISBN: 978-0134670959.