Laboratorium Iv – Teoria współbieżności

Weronika Szybińska, 14.11.2022

Treść zadania:

1. Bufor o rozmiarze 2M
2. Jest m producentów i n konsumentów
3. Producent wstawia do bufora losowa liczbę elementów (nie więcej niż M)
4. Konsument pobiera losowa liczbę elementów (nie więcej niż M)
5. Zaimplementować przy pomocy monitorów Javy oraz mechanizmów Java Concurrency Utilities
6. Przeprowadzić porównanie wydajności (np. czas wykonywania) vs. różne parametry, zrobić wykresy i je skomentować.

Opis rozwiązania:

Zadanie zostało wykonane przy użyciu klasy ReentrantLock. Producer wstawia do wspólnego bufora K, gdzie K<=M identycznych elementów (ich wartość jest określona przy konstruowaniu producenta). Konsument pobiera L, gdzie L<=M kolejnych elementów z bufora w formie ArrayList i jest wypisuje. Dostęp do bufora ma w określonym czasie tylko jeden producent lub konsument. Za dbanie o wybudzanie najdłużej czekających producentów i konsumentów odpowiadają dwa dodatkowe obiekty ReentrantLock, dzięki czemu każdy wątek ma szanse dostać się do bufora. Do wybudzania i usypiania wątków wykorzystane zostały obiekty klasy Condition. Program został zaimplementowany następująco:

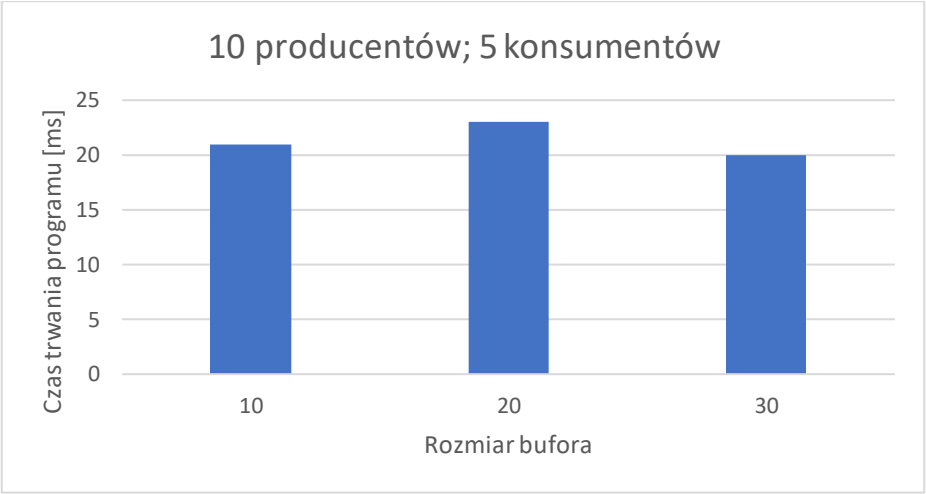
```
39  class Buffer{
40      private final int[] table;
41      private int putInx = 0;
42      private int getInx = 0;
43      private int counter = 0;
44      private int producentsQuantity;
45      private int consumersQuantity;
46
47      private final ReentrantLock lock = new ReentrantLock();
48      private final ReentrantLock producerLock = new ReentrantLock( fair: true);
49      private final ReentrantLock consumerLock = new ReentrantLock( fair: true);
50      private final Condition Empty = lock.newCondition();
51      private final Condition Full  = lock.newCondition();
52
53      public Buffer(int size, int producentsQuantity, int consumersQuantity){
54          this.table = new int[size];
55          this.producentsQuantity = producentsQuantity;
56          this.consumersQuantity = consumersQuantity;
57      }
```

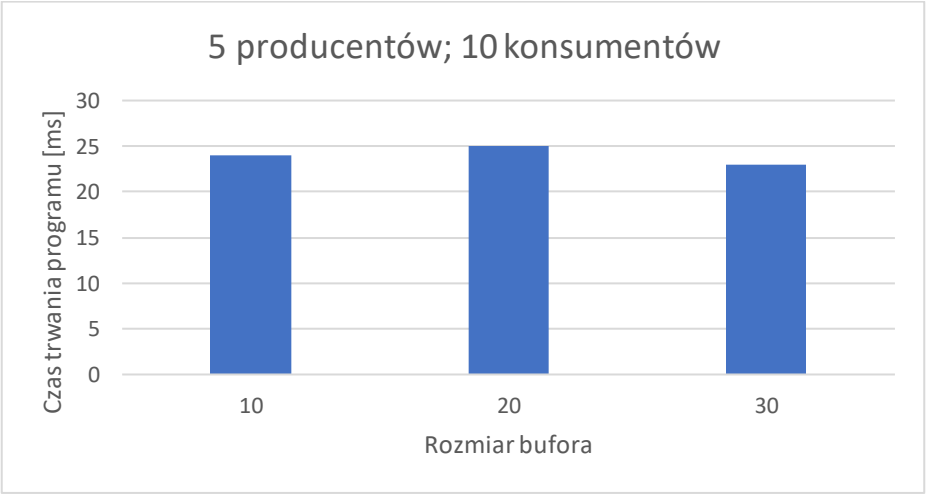
```java
public ArrayList<Integer> get(int quantity){
    consumerLock.lock();
    lock.lock();
    try {
        ArrayList<Integer> result = new ArrayList();
        while (counter < quantity) {
            if(producentsQuantity == 0){
                result.add(-1);
                return result;
            }
            try {
                Empty.await();
            } catch (InterruptedException e) {
                return result;
            }
        }
        for (int i = 0; i < quantity; i++) {
            result.add(table[getInx]);
            table[getInx] = -1;
            getInx++;
            if (getInx == table.length) {
                getInx = 0;
            }
        }
        counter -= quantity;
        Full.signal();
        return result;
    } finally {
        consumersQuantity -= 1;
        lock.unlock();
        consumerLock.unlock();
    }
}
}
```

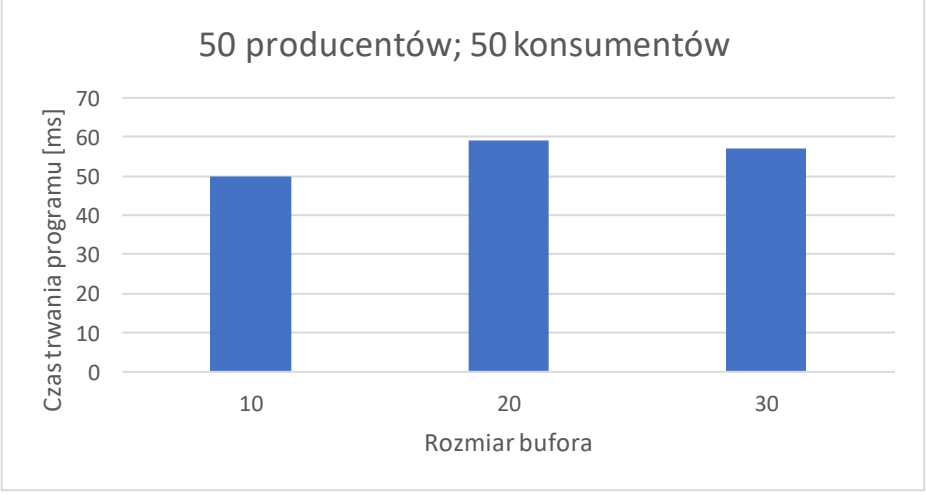
```java
public void put(int value, int quantity) {
    producerLock.lock();
    lock.lock();
    try {
        while (table.length - counter < quantity) {
            if(consumersQuantity == 0){
                return;
            }
            try {
                Full.await();
            } catch (InterruptedException e) {
                return;
            }
        }
        for (int i = 0; i < quantity; i++) {
            table[putInx] = value;
            putInx++;
            if (putInx == table.length) {
                putInx = 0;
            }
        }
        counter += quantity;
        Empty.signal();
    } finally {
        lock.unlock();
        producentsQuantity -= 1;
        producerLock.unlock();
    }
}
```

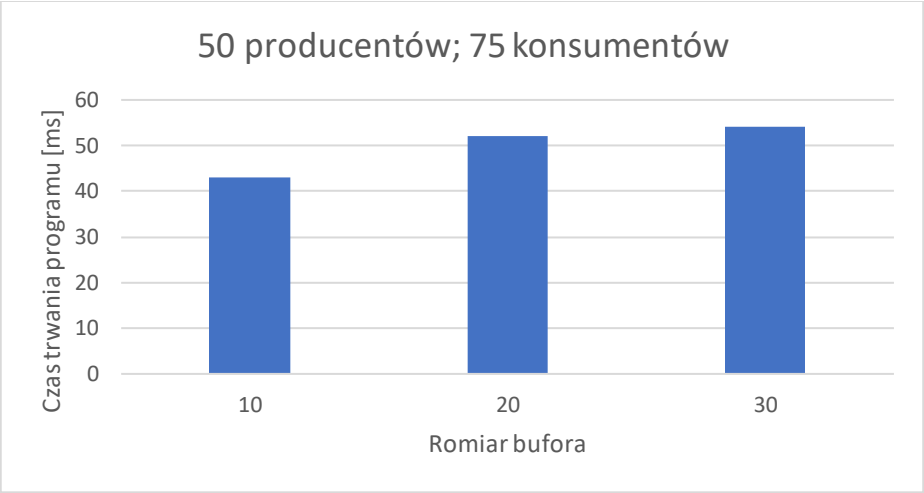
```java
class Producer extends Thread {
    private final Buffer _buf;
    private final int quantity;
    private final int value;

    public Producer(Buffer _buf, int quantity, int value){
        this._buf = _buf;
        this.quantity = quantity;
        this.value = value;
    }

    public void run() { _buf.put(value,quantity); }
}

class Consumer extends Thread {
    private final Buffer _buf;
    private final int quantity;

    public Consumer(Buffer _buf, int quantity){
        this._buf = _buf;
        this.quantity = quantity;
    }

    public void run() {

        ArrayList<Integer> x = _buf.get(quantity);
        for(int i=0;i <quantity; i++){
            System.out.println("value: " + x.get(i));
        }
    }
}
```

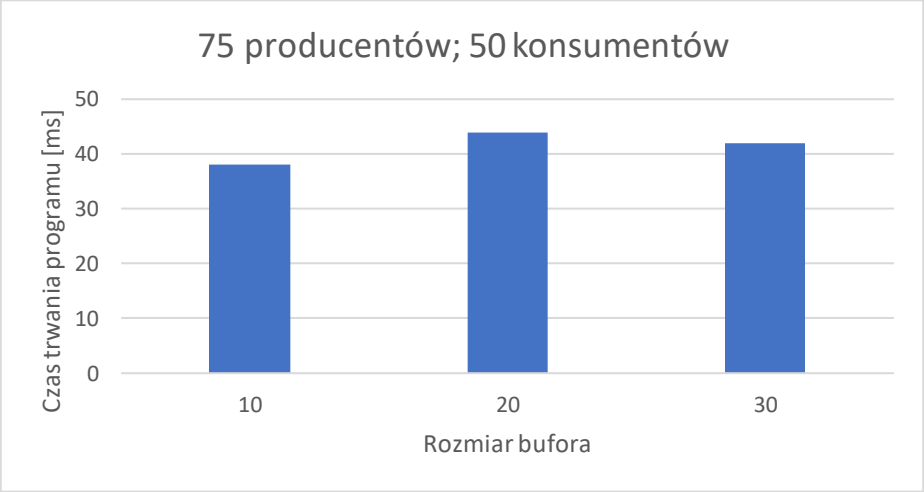
Po zaimplementowaniu programu, zmierzony został czas jego działania dla różnych ilości producentów oraz konsumentów w zależności od wielkości bufora. Na podstawie wyników utworzone zostały wykresy słupkowe.











WNIOSKI:

Na podstawie powstałych wykresów można wysunąć wniosek, że w przypadku używania klasy ReentrantLock, program jest na tyle zoptymalizowany, że rozmiar bufora nie wpływa znacząco na prędkość działania programu. Dzięki ograniczeniu na ilość wstawianych lub pobieranych przez wątki danych, program nie zawiesza się dopóki istnieją zarówno producenci jak i konsumenci i zapewnia uczciwy dostęp do bufora (wątek najdłużej czekający dostaje dostęp do metody get lub put w zależności czy jest to producent czy konsument).

BIBLIOGRAFIA:

https://www.baeldung.com/java-concurrent-locks

https://www.digitalocean.com/community/tutorials/java-lock-example-reentrantlock

https://edu.pjwstk.edu.pl/wyklady/zap/scb/W9/W9.htm