Sprawozdanie

Teoria Współbieżności

Laboratorium 4

Joanna Bryk, środa 17:50

1. Wstęp

Zadanie polegało na implementacji programu producenci i konsumenci z buforem o rozmiarze 2M. Producenci i konsumenci mieli korzystać z buforu taki sposób, aby pobierali i wstawiali losową liczbę porcji. Program został zaimplementowany przy pomocy monitorów Javy oraz mechanizmów Java Concurrency Utilities (m.in. *ReentantLock, Condition*).

2. Implementacja

W implementacji korzystałam z rozwiązania zaproponowanego w podręczniku *Programowanie współbieżne i rozproszone*, Zbigniew Weiss, Tadeusz Gruźlewski (zadanie 4.4.3). Rozwiązanie to wykorzystuje 4 zmienne warunkowe – *firstProd, restProd, firstCons, restCons*. Zmienne *firstProd* i *firstCons* wstrzymują pierwszy proces w kolejce. W funkcji *put* sprawdzamy, czy *portion_list* nie jest za długa (bo jest mniej wolnych miejsc w buforze). Kiedy ten warunek jest spełniony, lokujemy nowe dane w buforze i informujemy o tym czekające wątki. Podobnie przedstawia się funkcja *get*. Obie te operacje znajdą się w bloku *lock.lock()* zakończonym *lock.unlock()*. Wykorzystałam *ReentrantLock*, funkcje *lock(), await()* i *signal()*.

```
public class Buffer {

    private LinkedList<Integer> bufferList = new LinkedList<>();
    private final int size;

    final ReentrantLock lock = new ReentrantLock();
    final Condition firstProd  = lock.newCondition();
    final Condition firstCons = lock.newCondition();
    final Condition restProd = lock.newCondition();
    final Condition restCons = lock.newCondition();

    public Buffer(int M) {
        this.size = 2 * M;
    }

    public void put(ArrayList<Integer> portion_list) throws
InterruptedException{
        lock.lock();
        try {
            if(lock.hasWaiters(firstProd)) restProd.await();
            while(size - bufferList.size() < portion_list.size())
                firstProd.await();
            bufferList.addAll(portion_list);
            restProd.signal();
            firstCons.signal();
        } finally {
            lock.unlock();
```

```
        }
    }

    public void get(int size) throws InterruptedException {
        lock.lock();
        try {
            if(lock.hasWaiters(firstCons)) restCons.await();
            while(bufferList.size() < size)
                firstCons.await();
            for(int i=0; i<size; i++) {
                bufferList.removeLast();
            }
            restCons.signal();
            firstProd.signal();
        } finally {
            lock.unlock();
        }
    }
```

Niewielkiej zmianie uległy także funkcje r*un()* zarówno w klasie *Consumer* jak i *Producer*.

```
public void run() {
    for (int i = 0; i < 100; i++) {
        int portion = (int) (Math.random() * (_buf.getSize() / 2 - 1));
        try {
            _buf.get(portion);
        } catch (InterruptedException e) {
            return;
        }
    }
}
```

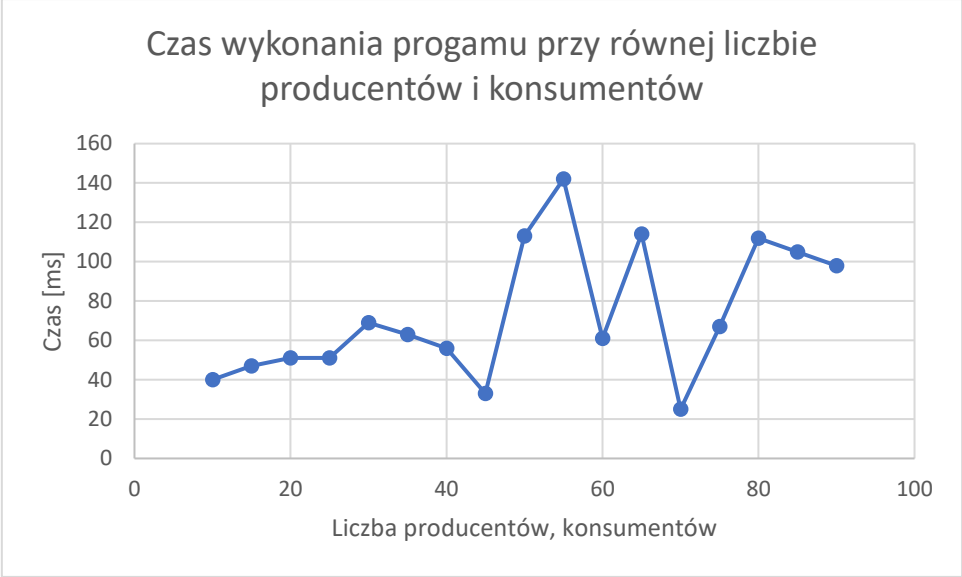
Producer:

```
public void run() {
    for (int i = 0; i < 100; i++) {
        ArrayList<Integer> portion_list = new ArrayList<>();
        int portion = (int) (Math.random() * (_buf.getSize() / 2 - 1));
        for(int j = 0; j<length; j++ ){
            portion_list.add((int) (Math.random()));
        }
        try {
            _buf.put(portion_list);
        } catch (InterruptedException e) {
            return;
        }
    }
}
```

Oczywiście może dojść do sytuacji, w której jest zbyt dużo producentów lub konsumentów w stosunku do liczby drugiej grupy. Wtedy, jeśli któryś z procesów zakończył działanie, wymuszam koniec działanie drugiego.

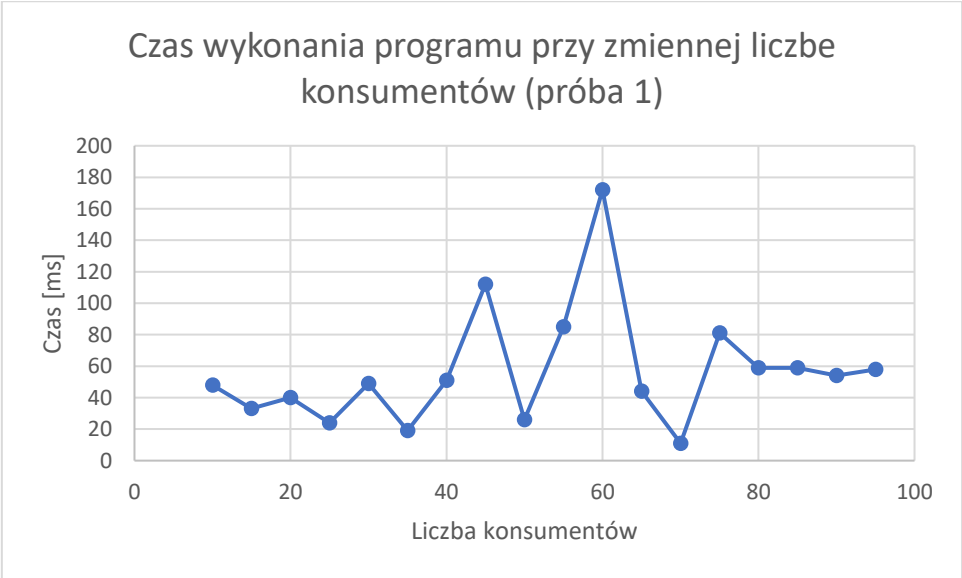
3. Wykresy

Na początku uruchamiałam program z równą liczbą producentów i konsumentów przy rozmiarze buforu 100.



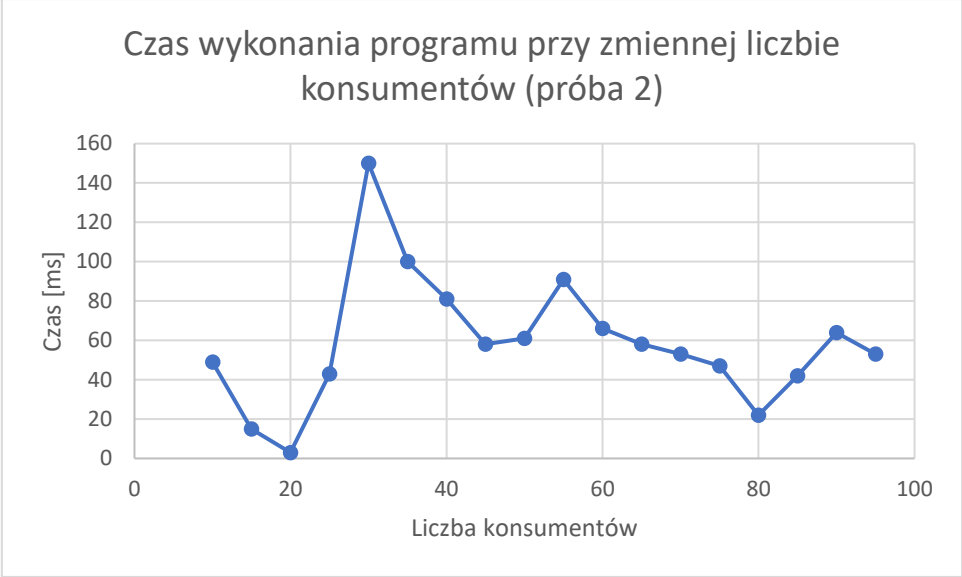
Rysunek 1Wykres przy równej liczbie konsumentów i producentów

Następnie ustaliłam stałą liczbę producentów równą 50 i zmieniałam liczbę konsumentów w zakresie od 10 do 95.



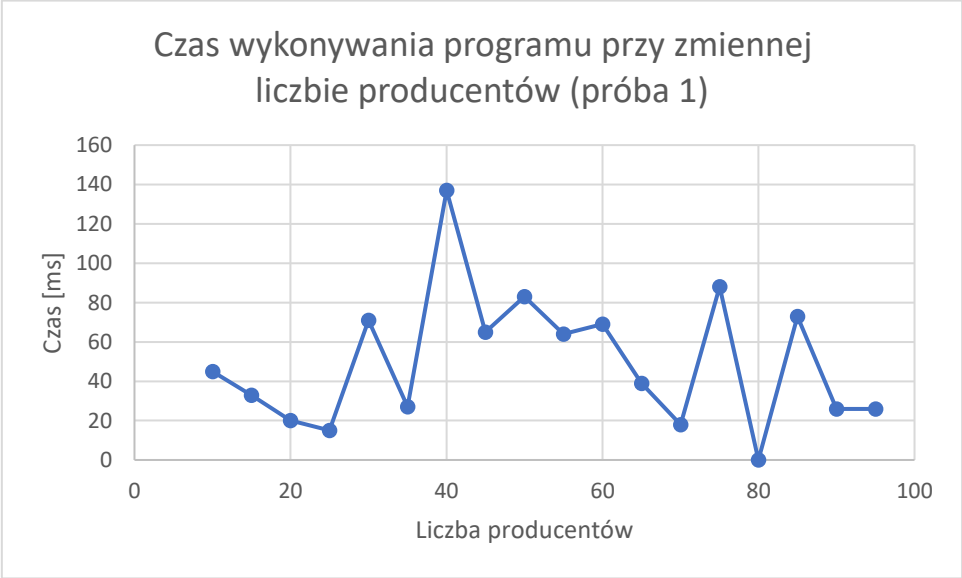
Rysunek 2

Postanowiłam przeprowadzić jeszcze jedną próbę:

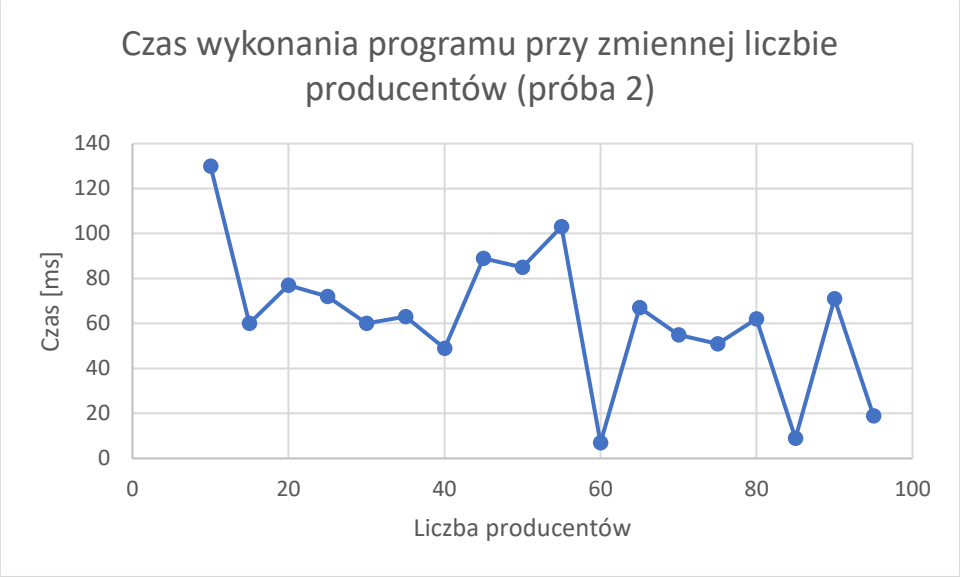


Rysunek 3

Ten sam eksperyment przeprowadziłam przy zmiennej liczbie producentów.



Rysunek 4



Rysunek 5

4. Podsumowanie

Wydawałoby się oczywistym, że powinna zachodzić zależność – im więcej producentów, konsumentów czas wykonania programu także się zwiększa. Jednak uważam, że wysunięcie takiego wniosku bazując na wykresie (rysunek 1.) nie byłoby poprawne. Niestety w kolejnych próbach przy zmiennej liczbie konsumentów i producentów nie jestem w stanie określić głównej zależności. Czasy programów mają podobne wartości z niewieloma wartościami odstającymi. Myślę, że jest to spowodowane faktem, że zarówno producenci jak i konsumenci operują na losowych rozmiarach porcji przez co czasy wykonywania programów mogą różnić się w nieoczekiwany sposób. Co więcej, w sytuacjach, kiedy jeden z producenci lub konsumenci skończyli swoje działanie, wymuszałam zakończenie programu. Możliwe, że do wyciągnięcia bardziej sprecyzowanych wniosków potrzebne jest wykonanie większej ilości prób, operując większymi liczbami.

5. Bibliografia

- Z. Weiss, T. Gruźlewski, *Programowanie współbieżne i rozproszone w przykładach i zadaniach*, WNT, 1993
- https://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/ReentrantLock.html
- https://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/Condition.html