Badanie porównawcze - metody synchronizacji wątków

Szymon Twardosz

29 listopad 2023

1 Wprowadzenie

1.1 Cel

Eksperyment polegał na zmierzeniu stopnia zagłodzenia, oraz szybkości zaimplementowanych metod synchronizacji wątków w Javie. Problemem branym pod uwagę podczas synchronizacji wątków był problem producenta-konsumenta. W ramach badania zaimplementowane zostały klasy:

1. **FourConditionBuffer** - rozwiązanie problemu na czterech zmiennych Condition oraz jednej zmiennej Lock
2. **ThreeLockBuffer** - rozwiązanie problemu na trzech zmiennych Lock i jednej zmiennej Condition

Załączone wykresy to pojedyncze przykłady przeprowadzonych testów. W rzeczywistości dla jednego zestawu parametrów przeprowadzone zostało kilka testów.

1.2 Dane techniczne

Doświadczenie realizowane było w języku Java wersji 17 oraz środkowiska IntelliJ. Sprzęt doświadczalny to laptop z systemem operacyjnym Windows 10 x64, o 4-rdzeniowym procesorze AMD Ryzen 7 3750 2.30GHz, z zainstalowaną pamięcią RAM 8,00GB

2 Implementacje metod

2.1 FourConditionBuffer - 1 Lock 4 Conditions

Metoda posiadająca dwa rodzaje kolejek:

- First - kolejka dla wątków (danego typu), które pierwsze znalazły się w buforze (wtedy kolejka First, była pusta).
- Other - kolejka dla wątków (danego typu), do której trafiają wątki gdy w kolejce First znajduje się już inny wątek.

Nowy wątek, który rozpoczyna funkcję consume() sprawdza czy na kolejce typu First nie czeka już żaden wątek. Wątek czekający na tej kolejce ma gwarancję pierwszeństwa w stosunku do pozostałych wątków danego typu. Jeżeli kolejka jest pusta, to on staję się tym wątkiem. W przeciwnym przypadku czeka w kolejce Other. Konsument z kolejki First, po konsumcji elementów z bufora (jeżeli jest w stanie to zrobić) budzi jeden wątek z kolejki Other oraz jeden wątek z kolejki First przeciwnego typu (tutaj Producer). W ten sposób mechanizm ten ma gwarantować brak zagłodzenia.

```java
@Override
public void consume(Consumer person, int request) {
    try{
        this.lock.lock();

        while(this.waitingConsumer){
            this.otherConsumerCondition.await();
        }

        this.waitingConsumer = true;

        while (buffer < request){
            this.firstConsumerCondition.await();
        }

        this.waitingConsumer = false;

        buffer -= request;

        this.otherConsumerCondition.signal();
        this.firstProducerCondition.signal();


    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } finally {
        this.lock.unlock();
    }
}
```

```java
public void produce(Producer person, int request) {
    try {
        this.lock.lock();

        while (this.waitingProducer){
            this.otherProducerCondition.await();
        }

        this.waitingProducer = true;

        while(buffer + request > maxBuffer){

            this.firstProducerCondition.await();
        }

        this.waitingProducer = false; // Stop waiting

        buffer += request;

        this.otherProducerCondition.signal();
        this.firstConsumerCondition.signal();


    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } finally {
        this.lock.unlock();
    }
}
```

Rysunek 1: FourConditionBuffer - metoda consume() i produce()

2.2 ThreeLockBuffer - 3 Locki 1 Condition

Metoda, która posiada jedną wspólną zmienną Lock oraz dwie zmienne Lock dla poszczególnych klas wątków. Na początku wątek zdobywa swojego klasowego Locka. Następnie czeka na możliwość wejścia do wspólnego Lock'a. Później oczekuje na możliwość konsumpcji/produkcji (jeżeli nie jest w stanie tego zrobić). Po operacji na buforze „budzi"wątek przeciwnego typu czekający na Condition.
System ten ma zagwarantować brak zagłodzenia. W związku z tym, że po zdobyciu Lock'a danego typu, wątek posiada „pierwszeństwo"przed pozostałymi wątkami swojej klasy. Jest to spowodowane faktem, iż daną zmienną Lock może w jednym momencie posiadać tylko jeden wątek. Następnie o wspólny Lock konkurują maksymalnie dwa wątki o przeciwnych typach.

```java
@Override
public void consume(Consumer person, int request) {
    try{
        this.consumerLock.lock();

        this.commonLock.lock();


        while (buffer < request){
            this.waiting.await();
        }

        buffer -= request;

        this.waiting.signal();


    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } finally {
        this.commonLock.unlock();
        this.consumerLock.unlock();
    }
}
```

```java
@Override
public void produce(Producer person, int request) {
    try {
        this.producerLock.lock();

        this.commonLock.lock();

        while(buffer + request > maxBuffer){

            this.waiting.await();
        }

        buffer += request;


        this.waiting.signal();


    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } finally {

        this.commonLock.unlock();
        this.producerLock.unlock();
    }

}
```

Rysunek 2: ThreeLockBuffer - metoda consume() i produce()

3 Czas spędzony w Buforze

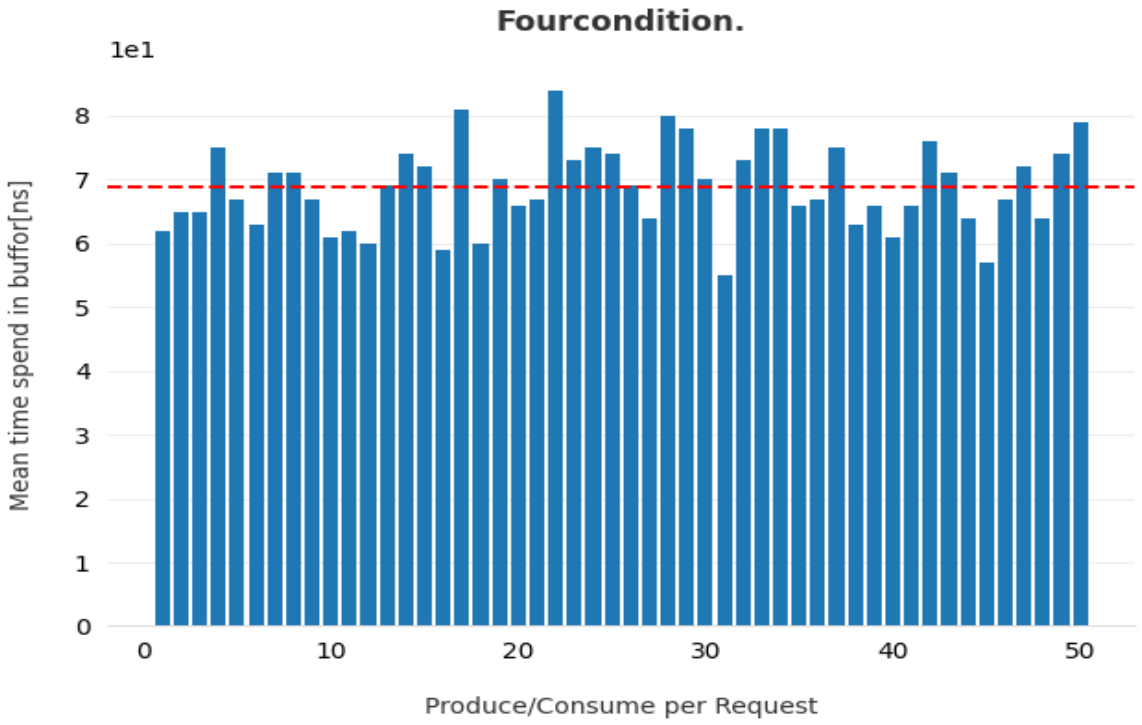
3.1 Opis eksperymentu

Eksperyment polegał na zmierzeniu średniego czasu, jaki zajmuję w buforze wątek, o określonej produkowanej/konsumowanej części. Jego celem jest sprawdzenie czy dany bufor (jego implementacja w Javie) faktycznie nie zagładza wątków. Każdy wątek działa w nieskończonej pętli. Za każdym razem, przed wejściem do bufora losuję określoną liczbę elementów. Następnie włączany jest zegar, który mierzy ile czasu dany wątek spędzi w buforze. Wątek wywołuje metodą consume()/produce(). Następnie zegar jest zatrzymywany i czas zapisywany jest w odpowiednie miejsce w tablicy wyników. Na końcu dla każdej liczby elementów, liczony jest średni czas jaki wątek musiał spędzić w buforze.
Dla każdego pomiaru wyodrębnić należy użyte parametry. Są to:

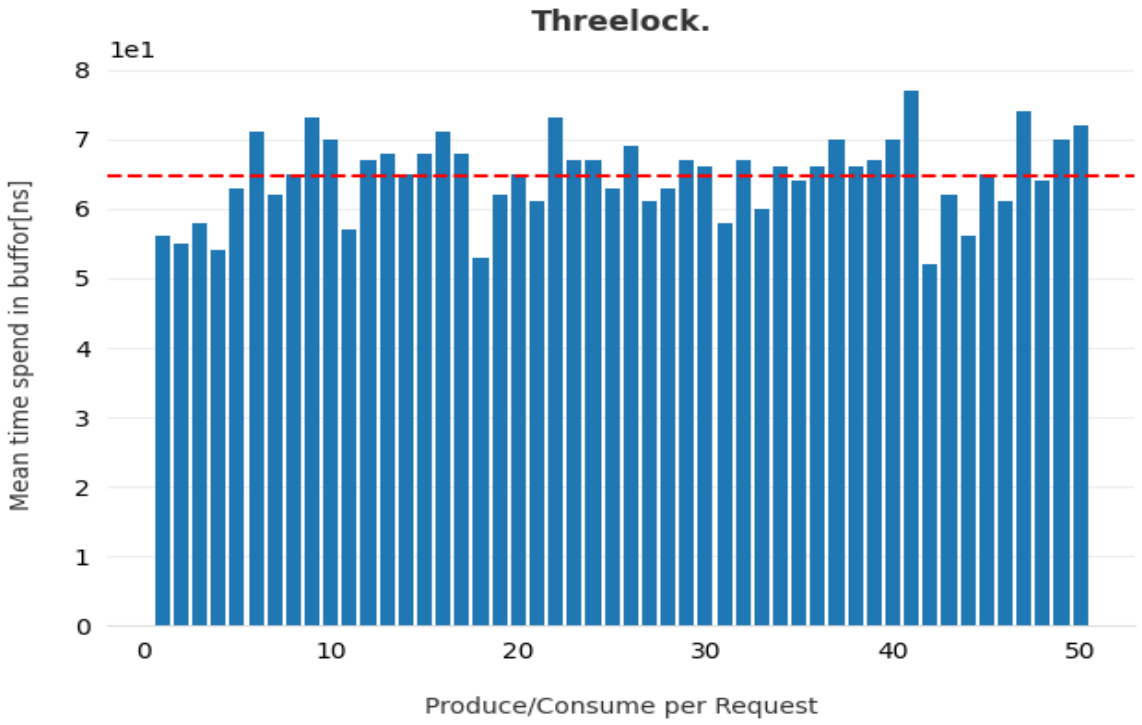
- Czas trwania testu dla pojedynczego bufora
- Pojemność bufora
- Liczba wątków producenta i konsumenta
- Maksymalna liczba wysłana do bufora (maksymalna prośba o produkcję/konsumpcję)

3.2 Wyniki

- **Parametry:** czas trwania: 5 minut
  pojemność bufora: 100
  liczba wątków klasy Producer (Consumer): 10
  Maksymalna liczba wysyłana do Bufera: 50



Rysunek 3: Wyniki dla FourConditionBuffer



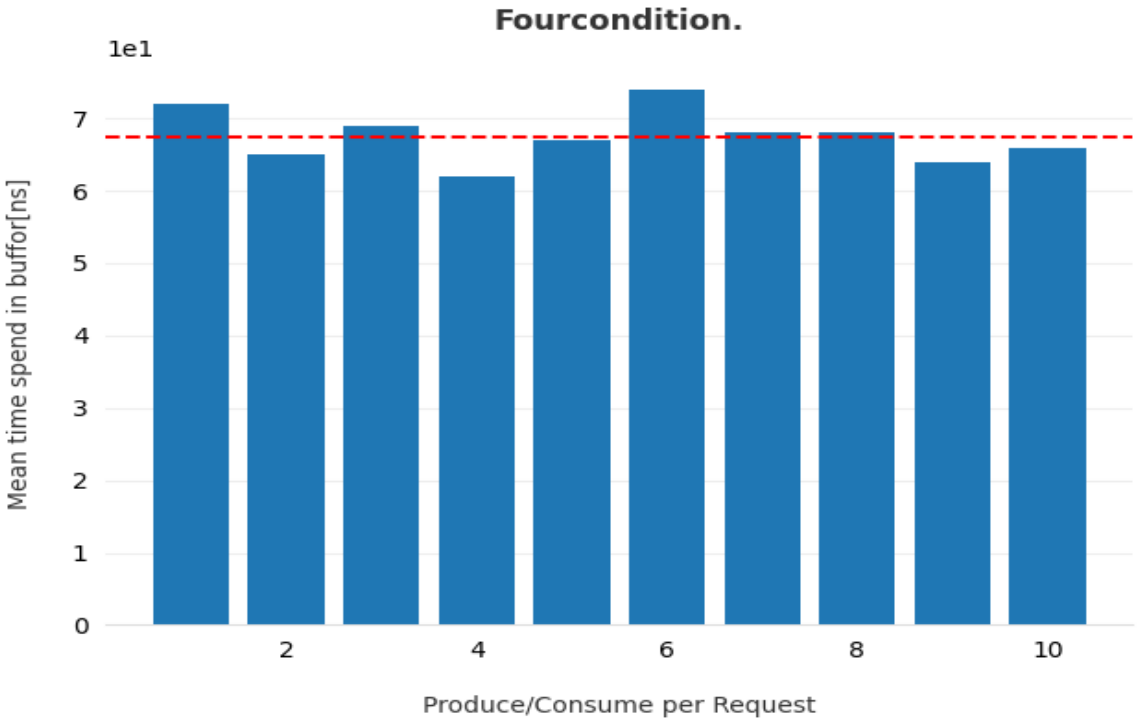
Rysunek 4: Wyniki dla ThreeLockBuffer

Wnioski

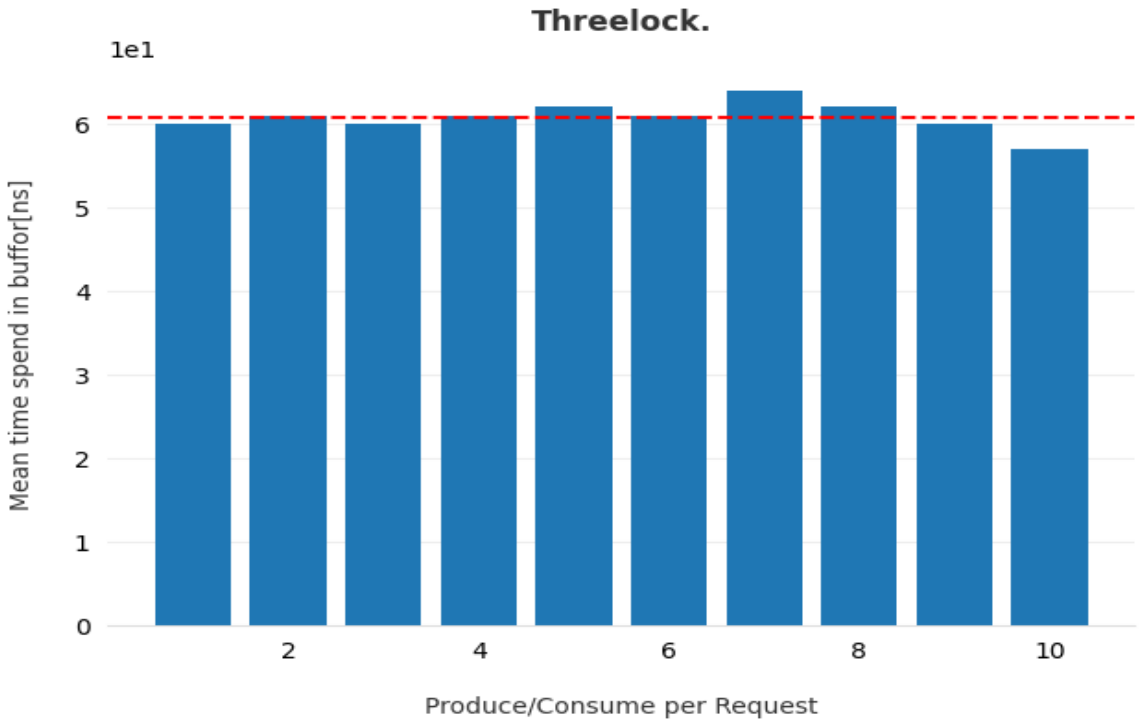
Analizując powyższe wykresy, wnioskuję iż zaimplementowane bufory nie posiadają tendencji do głodzenia wątków. Poszczególne wątki spędzają podobną ilość czasu w buferze. Czy sytuacja

może się jeszcze poprawić ? Co się stanie jak zmniejsze maksymalną liczbę wysyłana do bufera ?

- **Parametry:** czas trwania: 5 minut
  pojemność bufora: 100
  liczba wątków klasy Producer (Consumer): 10
  Maksymalna liczba wysyłana do Bufera: 10



Rysunek 5: Wyniki dla FourConditionBuffer



Rysunek 6: Wyniki dla ThreeLockBuffer

Wnioski Zmniejszenie maksymalnej liczby wysyłanej do bufera polepszyła jego jakość (mniejsze zagładzanie). Dla każdej „prośby"wysyłanej do bufora, czas, jaki wątek spędził w monitorze przybliżył się do średniej. Zatem, jeżeli programiście zależy na nie zagładzaniu wątków, to lepiej aby odpowiednio zwiększył bufor, badź zmniejszył maksymalną liczbę wysyłaną do bufora.

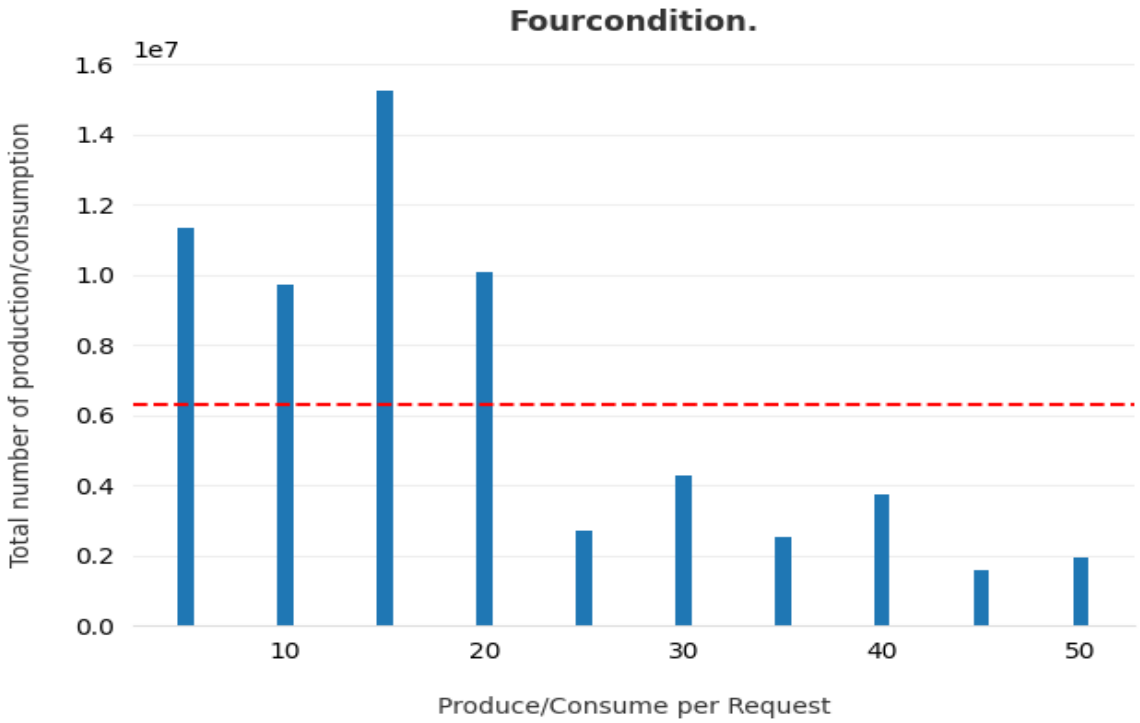
4 Liczba produkcji/konsumpcji w określonym czasie

4.1 Opis

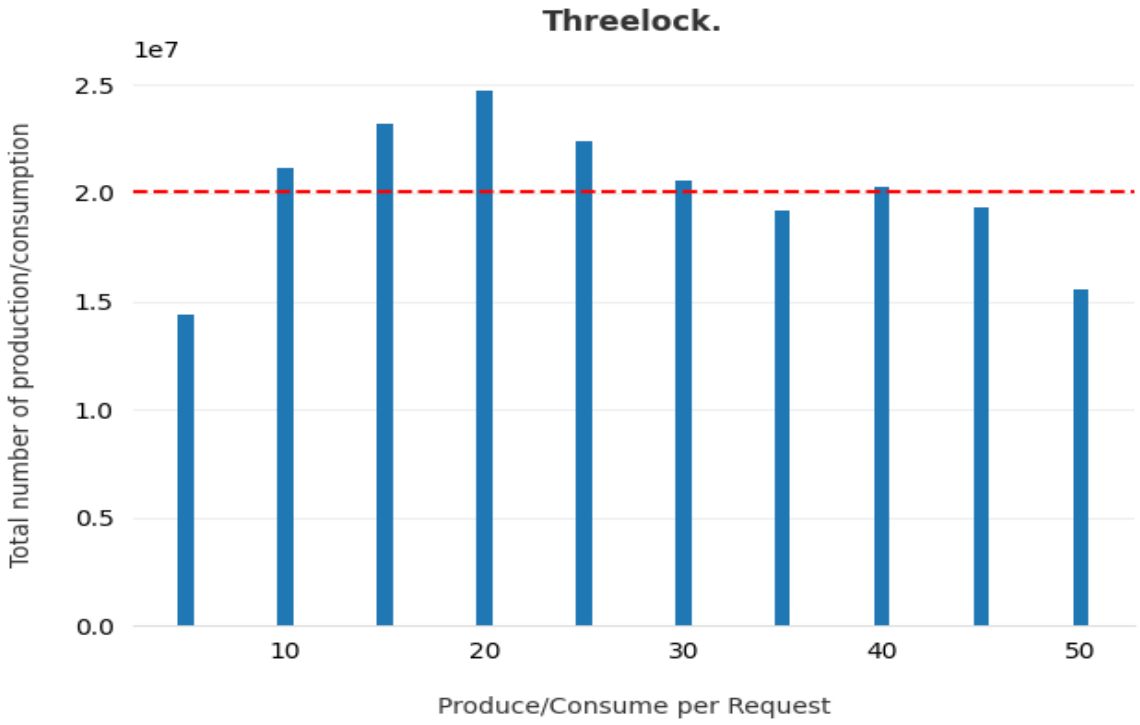
Eksperyment pierwszy może kumulować pewne błędy, związane z mierzeniem czasu. Jest on tam mierzony wielokrotnie, za każdym razem gdy wątek wykona metodę produce() czy consume(). Pomiar czasu może nie być dostatecznie dokładne, co wpływa na błąd pomiaru. Zatem w celu potwierdzenia braku zagładzania wątków wykonany został drugi eksperyment. Polega on na pomiarze liczby konsumpcji/produkcji jakie wykonuję każdy wątek. Ponadto, każdy z zainicjalizowanych wątków posiada określoną (z góry przypisaną) liczbę jaką wysyła do bufora.W związku z tym w systemie działają wątki zarówno „chciwe"jak i te mało wymagające.

4.2 Wyniki

- **Parametry:** czas trwania: 5 minut
  pojemność bufora: 100
  liczba wątków klasy Producer (Consumer): 7
  Maksymalna liczba wysyłana do Bufera: 50



Rysunek 7: Wyniki dla FourConditionBuffer

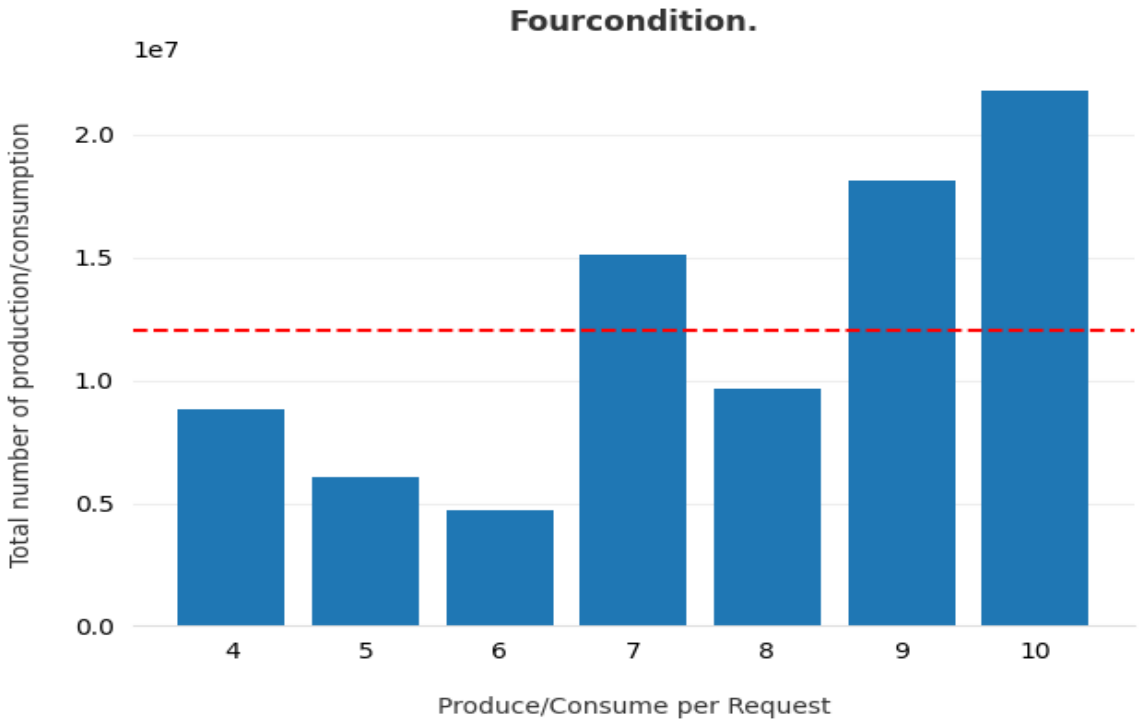


Rysunek 8: Wyniki dla ThreeLockBuffer

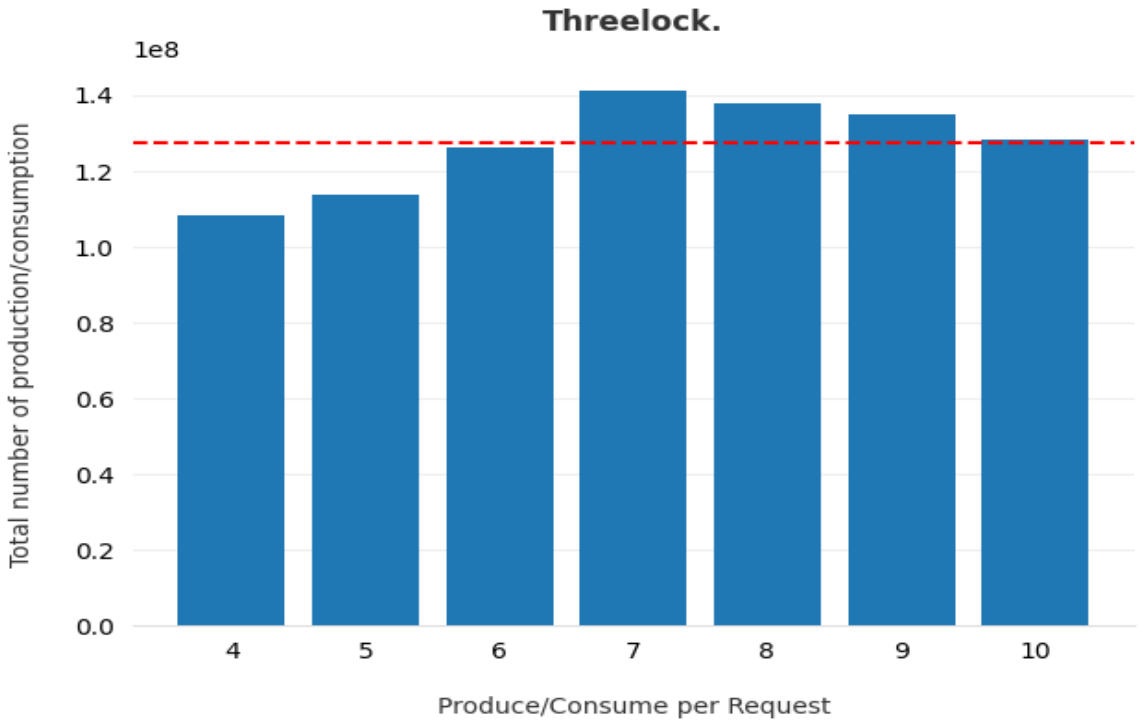
Wnioski Dla tego typu eksperymentu zaobserować można pewną zmianę. Teoretycznie zaimplementowane metody nie powinny zagładzać wątków. Mimo to niektóre z nich są zagładzane. Szczególnie dobrze widać to na metodzie z czterema zmiennymi Condition. „Chciwe"wątki o wiele

rzadziej wykonują operację na buferze. Czy zmiana parametru maxRequest (maksymalna liczba wysyłana do bufora) polepszy sytuację?

- **Parametry:** czas trwania: 5 minut
  pojemność bufora: 100
  liczba wątków klasy Producer (Consumer): 10
  Maksymalna liczba wysyłana do Bufera: 10



Rysunek 9: Wyniki dla FourConditionBuffer



Rysunek 10: Wyniki dla ThreeLockBuffer

Wnioski Metoda oparta na trzech zmiennych Lock nadal radzi sobie z zagładzaniem. Natomiast, implementacja FourCondition, mimo „łagodniejszych"parametrów nadal zagładza wątki. Tym razem zagładzanymi wątkami są te o mniejszych wymaganiach.

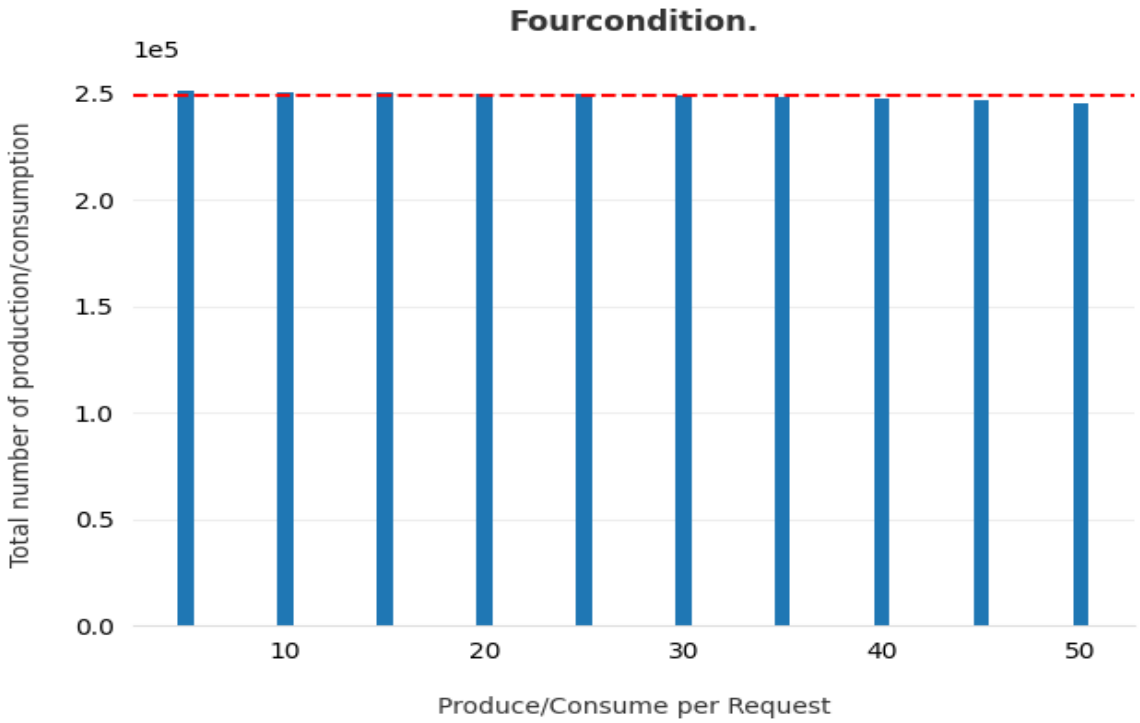
5 Liczba produkcji/konsumpcji w określonym czasie + metoda sleep()

5.1 Opis

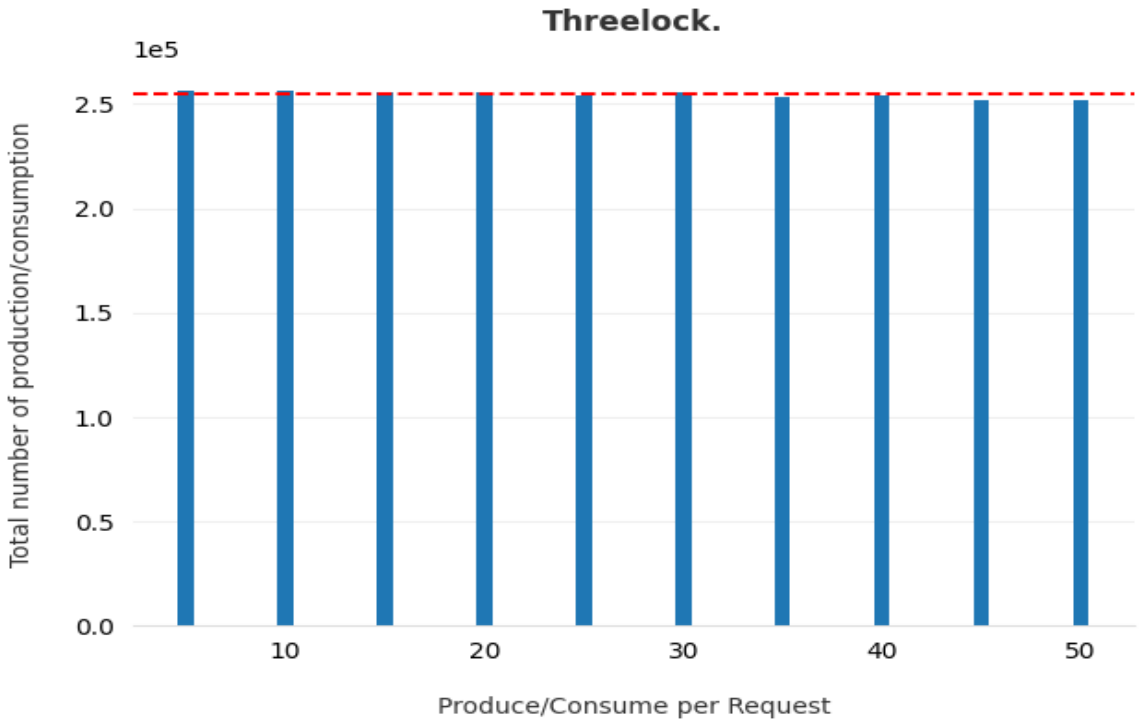
Eksperyment ten jest kopią porzedniego ekperymentu. Jednak, dodatkowo, każdy wątek po wykonaniu operacji na buforze, wywołuje funkcję sleep() na jedną nanosekundę.
Parametry dla tego testu są takie same jak poprzednio.

5.2 Wyniki

- **Parametry:** czas trwania: 5 minut
  pojemność bufora: 100
  liczba wątków klasy Producer (Consumer): 10
  Maksymalna liczba wysyłana do Bufera: 50



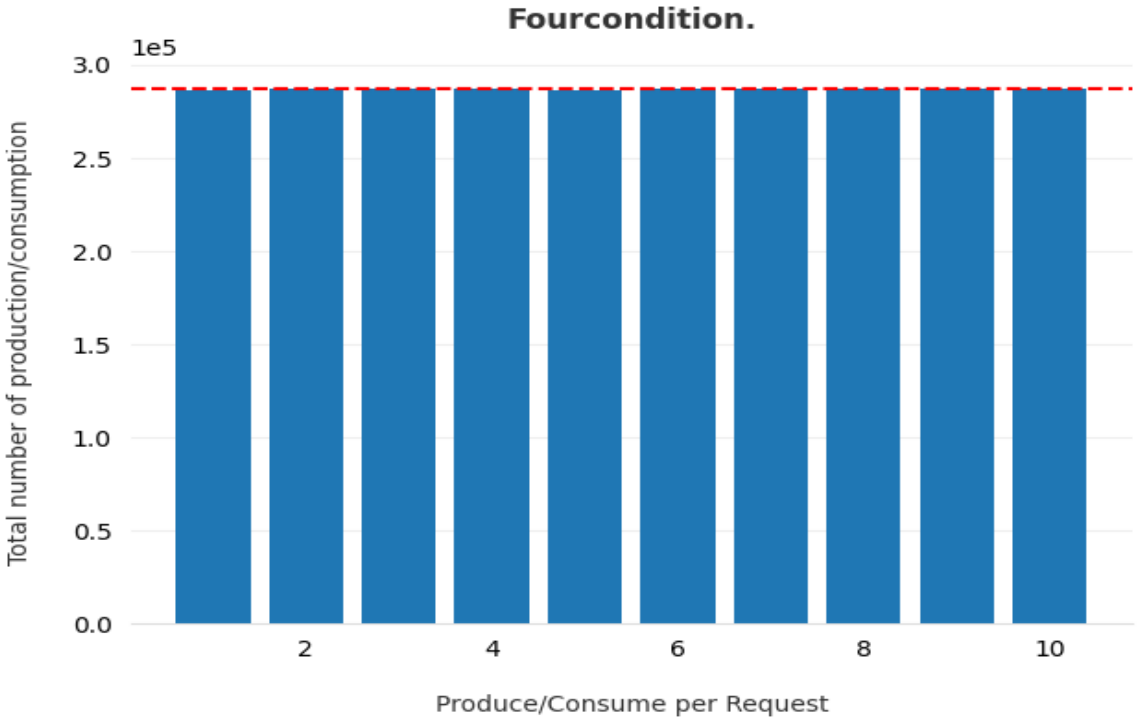
Rysunek 11: Wyniki dla FourConditionBuffer



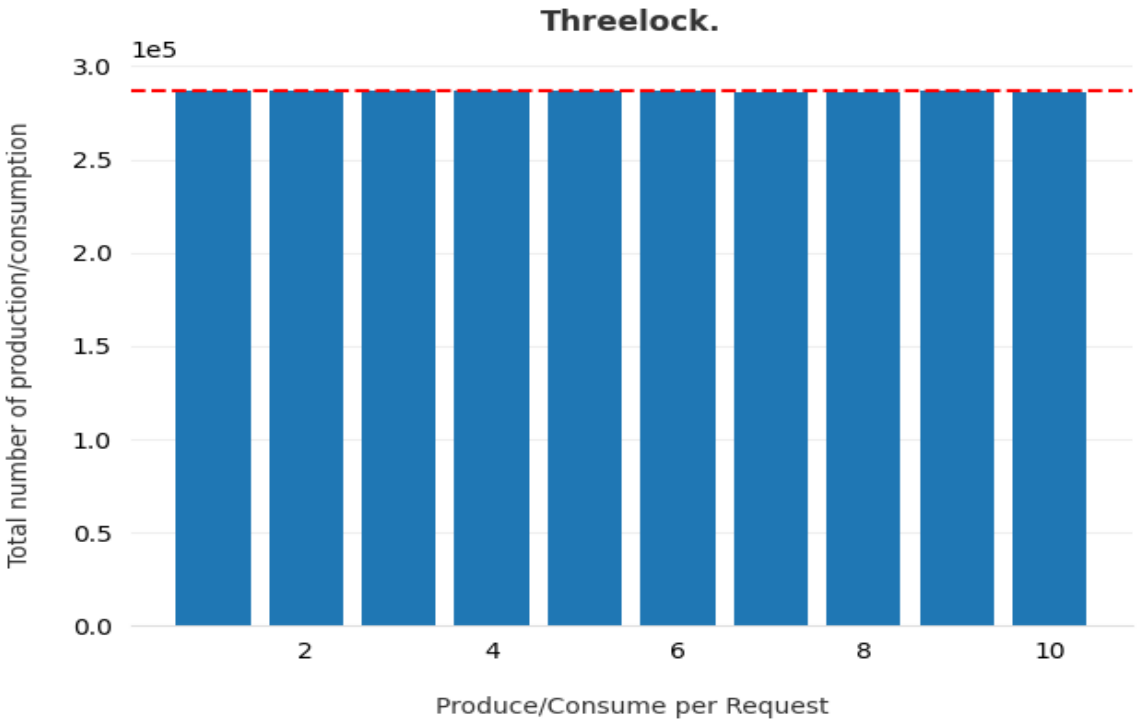
Rysunek 12: Wyniki dla ThreeLockBuffer

- **Parametry:** czas trwania: 5 minut
  pojemność bufora: 100
  liczba wątków klasy Producer (Consumer): 10

Maksymalna liczba wysyłana do Bufera: 10



Rysunek 13: Wyniki dla FourConditionBuffer



Rysunek 14: Wyniki dla ThreeLockBuffer

Wnioski W tym eksperymencie udowodniono, iż zaimplementowane metody w wyżej wymienio-

nych warunkach nie zagładzają wątków. Eksperyment ten nie był aż tak wrażliwy na parametr maksymalnej liczby wysyłanej do bufora. Może być to spowodowane faktem stosowania metody sleep(), która „desynchronizuje"wątki. Dodanie tej metody w znaczny sposób polepszyło wyniki metod, jeżeli chodzi o zagładzanie.

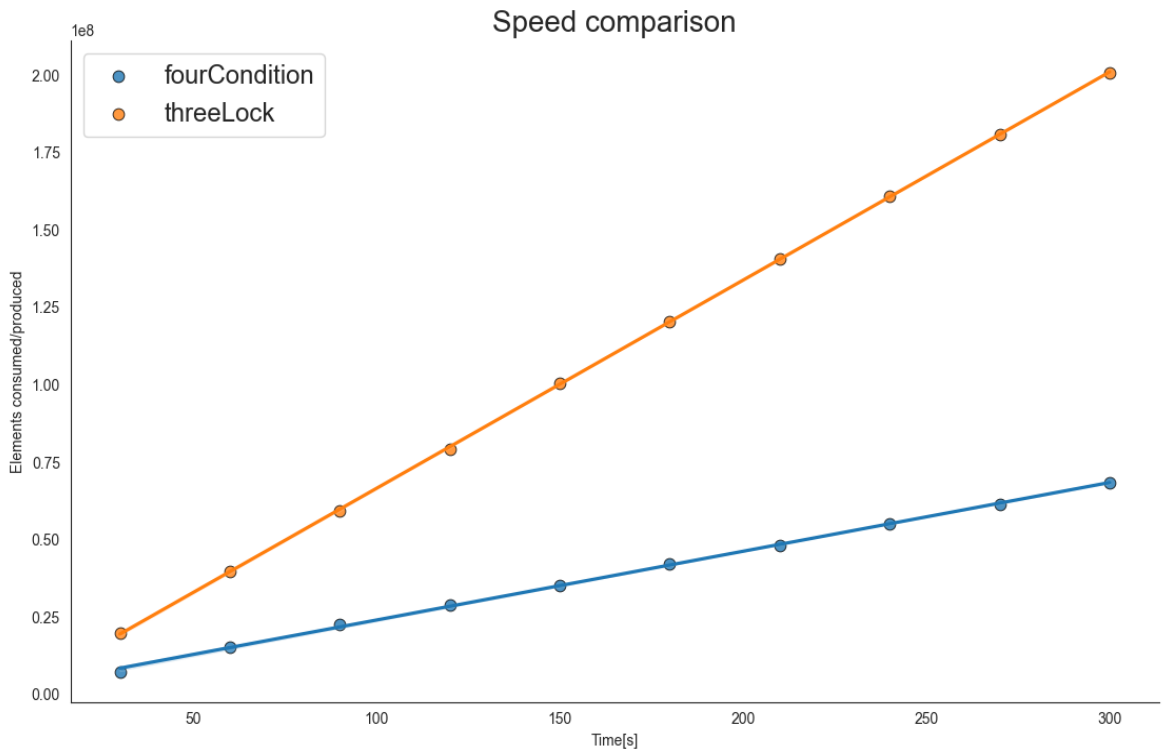
6 Pomiar efektywności

6.1 Opis

Ostatni test to pomiar efektywności zaimplementowanych metod. Każda z nich została włączona na określony czas. Cyklicznie sprawdzana była ilość operacji wykonanych na buforze (przez wszystkie wątki w systemie). Na końcu na wykresie wyświetlony został wykres zależności operacji na buforze od czasu z prostą najlepszego dopasowania. Parametry są takie same jak dla poprzedniego eksperymentu.

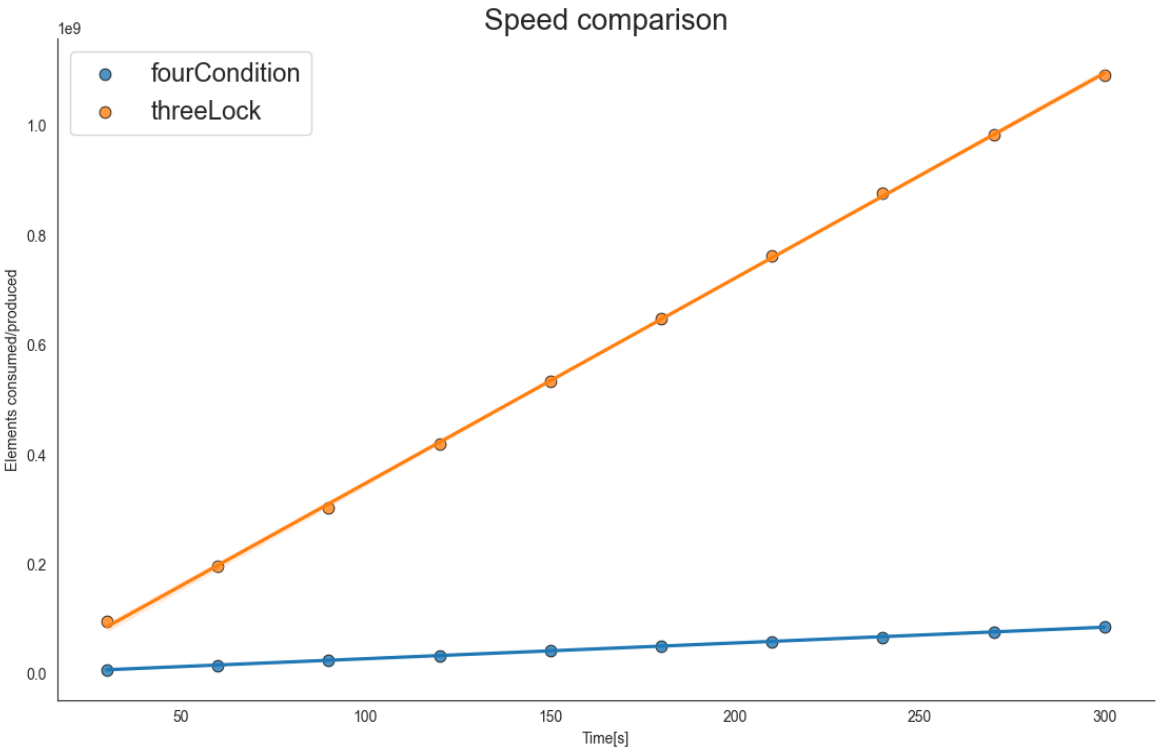
6.2 Wyniki

- **Parametry:** czas trwania: 5 minut
  pojemność bufora: 100
  liczba wątków klasy Producer (Consumer): 10
  Maksymalna liczba wysyłana do Bufera: 50
  Liczba rund: 10



Rysunek 15: Porównanie efektywności

- **Parametry:** czas trwania: 5 minut
  pojemność bufora: 100
  liczba wątków klasy Producer (Consumer): 10
  Maksymalna liczba wysyłana do Bufera: 10
  Liczba rund: 10



Rysunek 16: Porównanie efektywności

7 Wnioski

Analizując powyższe wykresy efektywności, można wywnioskować, że metoda oparta na trzech Lockach jest znacznie szybsza od tej opartej na czterech Conditions. Oprócz tego lepiej radzi sobie z problemem zagładzania. Metoda FourConditions dla wątków, które produkują ustaloną liczbę elementów (każdy inną) ma tendencję do zagładzania. Pokazuję to wykres znajdujący się na rysunku 8 i 9. Co ciekawe przy większej maksymalnej liczbie wysyłanej do bufora (równej 50) zagładzane są wątki „chciwe". Jeżeli jednak ten parametr się zmniejsza to zagładzane są wątki mniej wymagające. W odróżnieniu, metoda ThreeLock dobrze poradziła sobie z wszystkimi testami prezentowanymi w sprawozdaniu.
Podsumowując, metoda ThreeLock lepiej radzi sobie z synchronizacją wątków, zarówno pod względem efektywności jak i zagładzania.