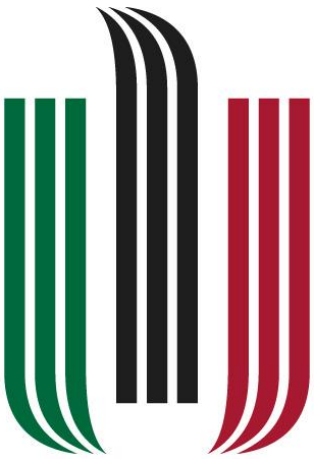
Laboratorium IV

Java Concurrency Utilities

Dominik Marek

28 października 2024



AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

1.Zadania

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.Rozwiązania

```java
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class QueueBuffer<T> {
    private final BlockingQueue<T> queue;
    private final int queueSize;

    public QueueBuffer(int size) {
        this.queueSize = size;
        this.queue = new ArrayBlockingQueue<>(size);
    }

    public void put(List<T> elementList) throws InterruptedException {
        if (2 * elementList.size() > this.queueSize)
            throw new RuntimeException("element list too big");
        for (T element : elementList)
            this.queue.put(element);
    }

    public List<T> take(int elementCount) throws InterruptedException {
        if (2 * elementCount > this.queueSize)
            throw new RuntimeException("element list too big");
        List<T> elementList = new LinkedList<>();
        for (int i = 0; i < elementCount; i++)
            elementList.add(this.queue.take());
        return elementList;
    }
}
```

```java
import java.util.LinkedList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Producer implements Runnable {
    private final QueueBuffer<Integer> buffer;
    private final int iterations_number;
    private final int producerID;
    private static final Random random = new Random();
    private static final AtomicInteger producerCount = new
AtomicInteger(0);
    private static final Lock lock = new ReentrantLock();
    private static final Condition cleanup = lock.newCondition();
    private final int  maxElementCount;

    public Producer(QueueBuffer<Integer> buffer, int repeat, int id, int M)
{
        this.buffer = buffer;
        this.iterations_number = repeat;
        this.producerID = id;
        this.maxElementCount = M;
    }
```

```
    private int getRandomElementCount() {
        return random.nextInt(this.maxElementCount) + 1;
    }

    private static List<Integer> getElementList(int count) {
        List<Integer> elementList = new LinkedList<>();
        for (int i = 0; i < count; i++)
            elementList.add(i + 1);
        return elementList;
    }

    @Override
    public void run() {
        producerCount.incrementAndGet();
        for (int i = 0; i < iterations_number; i++) {
            try {
                int elementCount = getRandomElementCount();
                System.out.println("producer[" + producerID + "]: try put "
+ elementCount);
                buffer.put(getElementList(elementCount));
                System.out.println("producer[" + producerID + "]: put " +
elementCount);
            } catch (InterruptedException e) {
                break;
            }
        }
        lock.lock();
        if (producerCount.decrementAndGet() == 0) cleanup.signal();
        lock.unlock();
        System.out.println("producer[" + producerID + "]: exit (" +
producerCount.intValue() + " remaining)");
    }
}
```

```
import java.util.Random;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Consumer implements Runnable {
    private final QueueBuffer<Integer> buffer;
    private final int iterations_number;
    private final int consumerID;
    private static final Random random = new Random();
    private static final AtomicInteger consumerCount = new
AtomicInteger(0);
    private static final Lock lock = new ReentrantLock();
    private static final Condition cleanup = lock.newCondition();
    private final int maxElementCount;

    public Consumer(QueueBuffer<Integer> buffer, int repeat, int id,int M)
{
        this.buffer = buffer;
        this.iterations_number = repeat;
        this.consumerID = id;
        this.maxElementCount = M;
    }

    private int getRandomElementCount() {
```

```
        return random.nextInt(maxElementCount) + 1;
    }

    @Override
    public void run() {
        consumerCount.incrementAndGet();
        for (int i = 0; i < iterations_number; i++) {
            try {
                int elementCount = getRandomElementCount();
                System.out.println("consumer[" + consumerID + "]: try take
" + elementCount);
                buffer.take(elementCount);
                System.out.println("consumer[" + consumerID + "]: taken " +
elementCount);
            } catch (InterruptedException e) {
                break;
            }
        }
        lock.lock();
        if (consumerCount.decrementAndGet() == 0) cleanup.signal();
        lock.unlock();
        System.out.println("consumer[" + consumerID + "]: exit (" +
consumerCount.intValue() + " remaining)");
    }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    private final static int maxElementCount = 1;
    private final static int iterationsNumber = 10000;
    private final static int producerNumber = 4;
    private final static int consumerNumber = 4;

    public static void main(String[] args) {
        QueueBuffer<Integer> queueBuffer = new
QueueBuffer<>(maxElementCount * 2);
        List<Runnable> runnableList = new ArrayList<>();
        for (int i = 0; i < producerNumber; i++) {
            runnableList.add(new Producer(queueBuffer, iterationsNumber, i,
maxElementCount));
        }
        for (int i = 0; i < consumerNumber; i++) {
            runnableList.add(new Consumer(queueBuffer, iterationsNumber, i,
maxElementCount));
        }



        runAll(runnableList);
    }

    private static void runAll(List<Runnable> runnableList) {
        List<Thread> threadList = runnableList.stream()
                .map(Thread::new)
                .toList();
```

```
        for (Thread thread : threadList) thread.start();
        try {
            Thread.sleep(1_000);
            for (Thread thread : threadList) thread.interrupt();
            for (Thread thread : threadList) thread.join();
        } catch (InterruptedException e) {
            // empty
        }
    }
}
```

Podany kod implementuje klasyczny problem producenta i konsumenta w programowaniu wielowątkowym. W tym przypadku wielu producentów generuje dane, które są następnie pobierane przez wielu konsumentów. Wspólnym elementem jest bufor, który jest zarządzany przez klasę QueueBuffer. Poniżej przedstawiam szczegółowy opis poszczególnych klas wchodzących w skład tej implementacji.

Klasa QueueBuffer:

Klasa QueueBuffer jest odpowiedzialna za zarządzanie współdzielonym buforem przy użyciu blokującej kolejki (ArrayBlockingQueue). Umożliwia ona producentom dodawanie elementów do bufora za pomocą metody put oraz konsumentom ich pobieranie przez metodę take(). Klasa ta wymusza ograniczenia na liczbę elementów, które mogą być jednocześnie przechowywane w buforze, zapewniając jednocześnie bezpieczeństwo wątkowe.

Klasa Producer:

Klasa Producer implementuje interfejs Runnable i odpowiada za generowanie danych, które mają być umieszczane w buforze. Działa w pętli, w której losowo wybiera liczbę elementów do dodania, a następnie próbuje je wstawić do bufora. Używa mechanizmu synchronizacji, aby zapewnić poprawność wątkową, a także zlicza aktywnych producentów, aby kontrolować ich zakończenie.

Klasa Consumer:

Klasa Consumer, podobnie jak Producer, implementuje interfejs Runnable i zajmuje się pobieraniem danych z bufora. W pętli losowo decyduje, ile elementów spróbuje pobrać, a następnie wykonuje operację take na buforze. Klasa ta również korzysta z mechanizmów synchronizacji, aby śledzić liczbę aktywnych konsumentów i kontrolować, kiedy wszyscy zakończą swoje działanie.

Klasa Main:

Klasa Main pełni rolę punktu wejścia aplikacji. Inicjalizuje obiekt QueueBuffer, a następnie tworzy i uruchamia wiele wątków dla producentów i konsumentów. Umożliwia to równoległe działanie wielu producentów i konsumentów, co jest kluczowe dla testowania i obserwowania działania bufora w różnych scenariuszach.

Implementacja problemu producenta i konsumenta w tym kodzie pokazuje, jak efektywnie zarządzać współdzielonymi zasobami w środowisku wielowątkowym. Użycie blokujących kolejek oraz mechanizmów synchronizacji zapewnia, że dane są bezpiecznie przekazywane między producentami a konsumentami. W każdej chwili można dostosować parametry dotyczące liczby wątków i iteracji, co czyni tę aplikację elastyczną i odpowiednią do różnych scenariuszy testowych.

2.Porównanie wydajności programu od wartości parametrów

a)

liczba producentów i konsumentów	czas wykonania [s]
100	1,161239
500	1,353878
1000	1,652596
5000	4,008435
10000	10,920346

Powyższe wyniki zostały uzyskane dla przyjętej pojemności bufora M = 10000 oraz liczby iteracji dla jedna producenta i konsumenta wynoszącej również 10000.

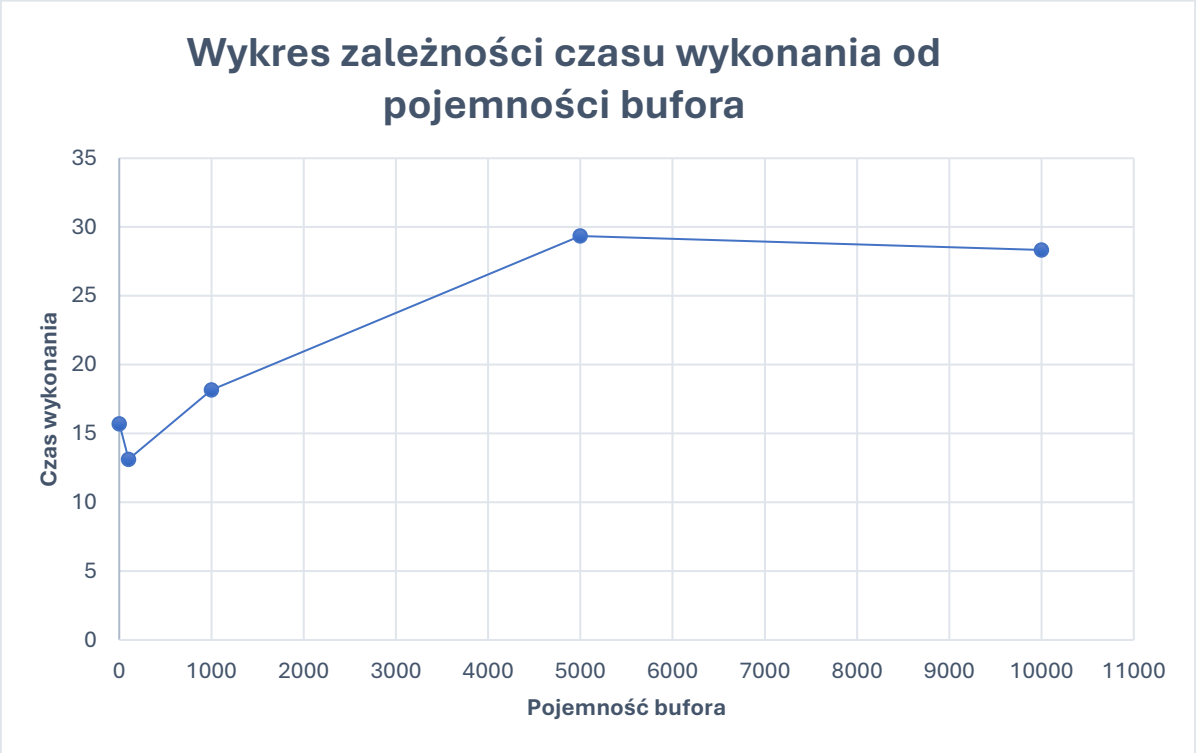
Analizując powyższe wyniki zestawione w tabeli jak również ich graficzne przedstawienie na wykresie, możemy zauważyć, iż czas wykonywania programu rośnie niemal liniowo względem zwiększania sumarycznej liczby producentów i konsumentów. Mniejszej liczby (100 i 500) czas

wykonania jest krótki (1,16 i 1,35 s), co wskazuje na wydajną współpracę przy niskiej konkurencji o dostęp do zasobów. Wzrost do 1000 producentów i konsumentów powoduje wyraźny wzrost czasu (1,65 s), co można przypisać intensywniejszej synchronizacji. Wartości 5000 i 10 000 generują znacząco dłuższy czas (4,0 i 10,9 s), wskazując na przeciążenie bufora – zarządzanie tak dużą liczbą operacji generuje opóźnienia związane z synchronizacją. Wyniki sugerują, że dla bufora o tej pojemności optymalna liczba producentów i konsumentów leży w zakresie 100-500, gdzie uzyskuje się najlepszy kompromis między współbieżnością a synchronizacją.

b)

Pojemność bufora	Czas wykonania [s]
1	15,707597
100	13,126042
1000	18,169596
5000	29,347572
10000	28,332828

Powyższe wyniki zostały uzyskane dla przyjętej liczby producentów i konsumentów równej 10000 oraz liczby iteracji dla jedna producenta i konsumenta wynoszącej również 10000.



Przy stałej liczbie producentów i konsumentów (po 10 000) oraz losowej liczbie elementów (od 0 do połowy pojemności bufora) czas wykonania zmienia się w zależności od pojemności bufora. Przy małych buforach (1 i 100) czas jest krótki (15,7 i 13,1 s), co sugeruje intensywną konkurencję o zasoby, ale także ograniczoną konieczność zarządzania pamięcią. Dla bufora o pojemności 1000 czas wzrasta do 18,2 s – większa pojemność zmniejsza blokady, ale zwiększa koszt zarządzania. Dla dużych pojemności (5000 i 10 000) czas wykonania rośnie jeszcze bardziej (29,3 i 28,3 s), co wskazuje, że zarządzanie dużą liczbą elementów , konieczność wstawianie i pobieranie większej liczby elementów z bufora i synchronizacja wpływają negatywnie na wydajność. Wyniki sugerują, że pojemność 100 zapewnia najlepszy kompromis między konkurencją o dostęp a efektywnym zarządzaniem pamięcią.

c)

Stosunek łącznej liczby producentów i konsumentów do pojemności bufora	*Czas wykonania [s]*
1	1,541194
2	1,822484
5	3,841547
10	9,419622
20	32,87473

Przyjęto M = 100 oraz liczbę iteracji dla producentów i konsumentów 10000.

Przy analizie stosunku łącznej liczby producentów i konsumentów do pojemności bufora czas wykonania wyraźnie wzrasta wraz ze wzrostem tego stosunku. Dla małych wartości (1 i 2) czas wykonania jest krótki (1,54 i 1,82 s), co sugeruje, że bufor o przyjętej pojemności efektywnie obsługuje tę liczbę operacji. Przy wyższych stosunkach, jak 5 i 10, czas wzrasta znacząco (do 3,84 i 9,42 s), ponieważ bufor jest bardziej obciążony, a synchronizacja między producentami i konsumentami staje się intensywniejsza. Najwyższy stosunek (20) powoduje drastyczny wzrost czasu (32,87 s), wskazując na przepełnienie bufora – konkurencja o zasoby znacznie opóźnia procesy. Wyniki sugerują, że przy rosnącym stosunku operacji do pojemności bufora konieczne jest zwiększenie jego rozmiaru lub zmniejszenie liczby operacji, aby uniknąć dużych opóźnień.

d)

liczba producentów	liczba konsumentów	Stosunek ilości producentów do konsumentów	Czas wykonania
200	19800	0.01	5,203666
2000	18000	0.1	19,96944
4000	16000	0.25	22,46208
6670	13330	0.5	27,01549
10000	10000	1	36,10847
13330	6670	2	23,19843
16000	4000	4	19,91022
18000	2000	10	18,36175
19800	200	100	14,98652

Przyjęto M = 5000 oraz liczbę iteracji 10000 dla producentów i konsumentów.

Przy znacznej przewagi liczby konsumentów do producentów program szybko kończy swoje działanie, gdyż po zakończeniu wymaganej liczby operacji dla producentów i opróżnienia bufora program zakończy swoje działanie. Wraz ze wzrost liczby producentów do konsumentów czas wykonywania programu rośnie, aż do osiągnięcia swojego maksimum dla równej liczby konsumentów i producentów. Przy równomiernym obciążeniu bufora liczba elementów jest losowana z szerokiego zakresu, co prowadzi do większej konkurencji o zasoby, wydłużając czas synchronizacji.

Następnie przy spadku liczby konsumentów względem producentów obserwujemy analogiczne wyniki jak wówczas przed zrównaniem liczby producentów i konsumentów. . Wyższa liczba producentów powoduje, że bufor jest regularnie wypełniany, zmniejszając ryzyko przestojów konsumentów .Natomiast dla przypadku znacznej przewagi producentów czas wykonania jest niemal trzykrotnie większy niż wówczas gdy dominowali konsumenci, jedną z przyczyn takiej sytuacji jest warunek zakończenia programu jakim jest zapełnienie bufora po tym jak już wszyscy konsumenci zakończą swoje operacje, gdy pojemność bufora jest duża może to wpłynąć na czas wykonania programu.

3.Bibliografia

- https://www.dz5.pl/ti/java/java_skladnia.pdf ~Jacek Rumiński, Język Java – rozdział o wątkach
- http://brinch-hansen.net/papers/1999b.pdf ~ Per Brinch Hansen
- https://www.artima.com/insidejvm/ed2/threadsynch.html ~Bill Venners, Inside the Java Virtual Machine - Charter 20 Thread Synchronization
- https://docs.oracle.com/en/java/javase/19/index.html
- https://docs.oracle.com/javase/1.5.0/docs/guide/concurrency/overview.html
- https://jenkov.com/tutorials/java-util-concurrent/index.html