Lab 03

Teoria współbieżności

Adrian Madej 16.10.2023

1. Treści zadań:

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.

Zrealizować program:

1. przy pomocy metod wait()/notify(). Kod szkieletu
    a. dla przypadku 1 producent/1 konsument
    b. dla przypadku n1 producentów/n2 konsumentów (n1>n2, n1=n2, n1<n2)
    c. wprowadzić wywołanie metody sleep() i wykonać pomiary, obserwując zachowanie producentów/konsumentów

2. przy pomocy operacji P()/V() dla semafora:
    a. n1=n2=1
    b. n1>1, n2>1

3. Przetwarzanie potokowe z buforem

- Bufor o rozmiarze N - wspólny dla wszystkich procesów!
- Proces A będący producentem.
- Proces Z będący konsumentem.
- Procesy B, C, ..., Y będące procesami przetwarzającymi. Każdy proces otrzymuje dana wejściowa od procesu poprzedniego, jego wyjście zaś jest konsumowane przez proces następny.

- Procesy przetwarzają dane w miejscu, po czym przechodzą do kolejnej komórki bufora i znowu przetwarzają ja w miejscu.
- Procesy działają z rożnymi prędkościami.

2. Rozwiązanie zadań:

Problem producenta i konsumenta polega na efektywnym współdzieleniu ograniczonych zasobów między dwiema rodzajami zadań lub procesów: producentami i konsumentami. Producent wytwarza pewne zasoby, a konsumenci je pobierają. Problem polega na tym, aby zapewnić, że producenci nie będą produkować zbyt wielu zasobów, gdy nie ma na to miejsca w buforze, oraz że konsumenci nie będą próbować pobierać zasobów, które nie istnieją**.**

1. Implementacja przy użyciu wait() oraz notify()

Rozwiązanie problemu opiera się na wspólnym dostępnie producentów i konsumentów do ograniczonego swoją wielkością bufora (w naszym przypadku maxBufferSize = 10). Każdy z wątków (producenta i konsumenta) ma określoną liczbę operacji jaką musi wykonać (producenci dodają zasoby, a konsumenci je pobierają). Ilość produktów wytworzonych musi być równoważna ilości produktów pobranych co zapewnia nam NWW (najmniejsza wspólna wielokrotność). Zatem, nasze klasy wyglądają następująco

Klasa Buffer

```
import java.util.LinkedList;

class Buffer {
    private final int maxBufferSize = 10;
    private final LinkedList<Integer> buffList = new
LinkedList<>();

    public boolean isEmpty(){
        return this.buffList.isEmpty();
    }

    public synchronized void put(int i) {
        while(buffList.size() >= maxBufferSize){
            try{
                wait();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
```

```
            }
        }
        buffList.add(i);

        // System.out.println("Producer put " + i);
        notifyAll();
    }

    public synchronized int get() {
        while(buffList.isEmpty()){
            try{
                wait();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        int result = buffList.removeFirst();

        // System.out.println("Consumer took " + result);

        notifyAll();
        return result;
    }
}
```

Klasa Calculator to klasa pomocnicza, pomoże nam wyznaczyć NWW

```
public class Calculator {
    private static Calculator calculator = null;

    private Calculator(){};
    public static Calculator getInstance(){
        if (calculator == null){
            calculator = new Calculator();
        }
        return calculator;
    }
    public int lcm(int a, int b) {
        return a * (b / gcd(a, b));
    }
    public int gcd(int a, int b) {
        while (b > 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
}
```

Klasa Consumer

```
class Consumer extends Thread {
    private final Buffer _buf;
    private final int nbOfOperations;

    Consumer(Buffer buffer, int nbOfOperations){
        this._buf = buffer;
        this.nbOfOperations = nbOfOperations;
    }

    public void run() {
        for (int i = 0; i < nbOfOperations; ++i) {
            _buf.get();
        }
    }
}
```

Klasa Producer

```
class Producer extends Thread {
    private final Buffer _buf;
    private int nbOfOperations;

    Producer(Buffer buffer, int nbOfOperations){
        this._buf = buffer;
        this.nbOfOperations = nbOfOperations;
    }

    public void run() {
        for (int i = 0; i < nbOfOperations; ++i) {
            _buf.put(i);
        }
    }
}
```

Klasa Test.

Metoda runTest umożliwia przeprowadzenie testu, przyjmuje ona liczbę producentów i konsumentów, oblicza NWW, a następnie ustala ilość operacji, którą muszą wykonać poszczególne wątki

```
import java.util.ArrayList;
import java.util.stream.IntStream;

public class Test {
    public static void runTests(int nbOfProducers, int
nbOfConsumers){
```

```
        System.out.println("Nb of Producers: " +
nbOfProducers);
        System.out.println("Nb of Consumers: " +
nbOfConsumers);
        System.out.println("----------------------------------
--");
        int lcm = Calculator.getInstance().lcm(nbOfProducers,
nbOfConsumers);
        int nbOfProducts = lcm * 10;
        int nbOfProducerOperation = nbOfProducts /
nbOfProducers;
        int nbOfConsumerOperation = nbOfProducts /
nbOfConsumers;
        System.out.println("Each producer will produce: " +
nbOfProducerOperation + " products");
        System.out.println("Each consumer will consume: " +
nbOfConsumerOperation + " products");

        Buffer buffer = new Buffer();
        ArrayList<Thread> threadArrayList = new ArrayList<>();

        IntStream.range(0, nbOfProducers).forEach(i ->
threadArrayList.add(new Producer(buffer,
nbOfProducerOperation)));
        IntStream.range(0, nbOfConsumers).forEach(i ->
threadArrayList.add(new Consumer(buffer,
nbOfConsumerOperation)));

        threadArrayList.forEach(Thread::start);
        threadArrayList.forEach(thread -> {
            try {
                thread.join();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        });
        System.out.println("All producers and consumers have
finished");
        if(buffer.isEmpty()) System.out.println("All products
have been consumed");
        else System.out.println("Not all products have been
consumed");
        System.out.println("----------------------------------
--");
        System.out.println("----------------------------------
--");
    }
```

Klasa PKmon wykorzystuje powyższą metoda do przeprowadzenia testu

```
import static org.example.Test.runTests;

public class PKmon {
    public static void main(String[] args) {
        runTests(1, 1);
    }
}
```

Wyniki podpunktów 1a, 1b

Dodajmy wywołania testujące w metodzie main

```
public class PKmon {
    public static void main(String[] args) {
        runTests(1, 1);      // a.
        runTests(5,4);       // b. n1 > n2
        runTests(4, 4);      // b. n1 = n2
        runTests(3, 5);      // b. n1 < n2
    }
```

Odpalamy program i dostajemy na wyjściu:

Nb of Producers: 1

Nb of Consumers: 1

Each producer will produce: 10 products

Each consumer will consume: 10 products

All producers and consumers have finished

All products have been consumed

Nb of Producers: 5

Nb of Consumers: 4

Each producer will produce: 40 products

Each consumer will consume: 50 products

All producers and consumers have finished

All products have been consumed

Nb of Producers: 4

Nb of Consumers: 4

Each producer will produce: 10 products

Each consumer will consume: 10 products

All producers and consumers have finished

All products have been consumed

Nb of Producers: 3

Nb of Consumers: 5

Each producer will produce: 50 products

Each consumer will consume: 30 products

All producers and consumers have finished

All products have been consumed

Jak widzimy program działa prawidłowo, wszystkie przedmioty zostały skonsumowane, nie doszło do zakleszczeń i wyścigów. Możemy przedstawić jakie operacje wykonują poszczególne wątki, zrobimy to dla problemu 1 producenta i 1 konsumenta.

Nb of Producers: 1

Nb of Consumers: 1

Each producer will produce: 10 products

Each consumer will consume: 10 products

Producer put 0

Producer put 1

Producer put 2

Producer put 3

Producer put 4

Producer put 5

Producer put 6

Producer put 7

Producer put 8

Producer put 9

Consumer took 0

Consumer took 1

Consumer took 2

Consumer took 3

Consumer took 4

Consumer took 5

Consumer took 6

Consumer took 7

Consumer took 8

Consumer took 9

All producers and consumers have finished

All products have been consumed

Serie produkcji wynikają z faktu, iż po wysłaniu powiadomienia, wątek zwalniający monitor, ponownie go zajmuje.

W celu wykonania podpunktu 1c. musimy zmodyfikować klasy producenta i konsumenta o losowy czas czekania. Zmiany te zajdą w pętlach w metodzie run().

Klasa Consumer

```
import java.util.Random;

class Consumer extends Thread {
    private final Buffer _buf;
    private final int nbOfOperations;
    private final Random random = new Random();

    Consumer(Buffer buffer, int nbOfOperations){
        this._buf = buffer;
        this.nbOfOperations = nbOfOperations;
    }

    public void run() {
        for (int i = 0; i < nbOfOperations; ++i) {
            _buf.get();
            try{
                Thread.sleep(random.nextInt(900));
            }catch (InterruptedException e){
                throw new RuntimeException(e);
            }
        }
    }
}
```

Klasa Producer

```
import java.util.Random;

class Producer extends Thread {
    private final Buffer _buf;
    private final int nbOfOperations;
    private final Random random = new Random();

    Producer(Buffer buffer, int nbOfOperations){
        this._buf = buffer;
        this.nbOfOperations = nbOfOperations;
    }

    public void run() {
        for (int i = 0; i < nbOfOperations; ++i) {
            _buf.put(i);
            try{
                Thread.sleep(random.nextInt(700));
            }catch (InterruptedException e){
```

```
                throw new RuntimeException(e);
            }
        }
    }
}
```

Wyniki podpunktu 1c

Program dalej działa prawidłowo, nie ma zakleszczeń, zwraca jednakowe wyniki, jednakże czas jego wykonywania jest znacznie dłuższy. Dodatkowo przy analizowaniu poszczególnych operacji możemy znaleźć różnice:

Nb of Producers: 1

Nb of Consumers: 1

Each producer will produce: 10 products

Each consumer will consume: 10 products

Producer put 0

Consumer took 0

Producer put 1

Consumer took 1

Producer put 2

Producer put 3

Consumer took 2

Producer put 4

Producer put 5

Producer put 6

Consumer took 3

Producer put 7

Producer put 8

Consumer took 4

Consumer took 5

Producer put 9

Consumer took 6

Consumer took 7

Consumer took 8

Consumer took 9

All producers and consumers have finished

All products have been consumed

Jak widzimy nie ma teraz jednej „serii” producentów i konsumentów.

2. Implementacja przy użyciu semaforów

Do realizacji zadania posłużą nam semafory zaimplementowane w poprzednim laboratorium. Będą to:

Semafor binarny

```
class BinarySemaphore {
    private boolean _stan = true;
    private int _czeka = 0;
    public BinarySemaphore() {
    }
    public synchronized void P() {
        while (!_stan) {
            try {
                _czeka++;
                wait();
                _czeka--;
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        _stan = false;
    }
    public synchronized void V() {
        _stan = true;
        if(_czeka > 0){
            notify();}
    }
}
```

Semafor licznikowy

```
public class GeneralSemaphore {
    private final BinarySemaphore mutex; // dostęp do metod P
i V
    private final BinarySemaphore resource; // dostęp do
pozwoleń
    private int permits;
    public GeneralSemaphore(int permits){
        if(permits < 0){
            throw new IllegalArgumentException("Permits must >
0.");
        }else{
            mutex = new BinarySemaphore();
            resource = new BinarySemaphore();
            this.permits = permits;
        }
    }
    public void P(){
        resource.P();
        mutex.P();
        permits--;
        if (this.permits > 0) {
            resource.V();
        }
        mutex.V();
    }
    public void V(){
        mutex.P();
        permits++;
        if(permits == 1){
            resource.V();
        }
        mutex.V();
    }
}
```

Program w zasadzie pozostaje bez zmian w stosunku do programu z zadania 1. Jedyne co się zmienia to klasa Buffer. Będzie ona wykorzystywała 3 semafory:

- slotsAvailable – zlicza ile pozostało miejsca w buforze
- notEmpty – zlicza ile produktów jest w buforze
- mutex – dba o dostęp do struktury _buf

Ostatecznie klasa Buffer2 prezentuje się następująco:

```
import java.util.LinkedList;

class Buffer2 {
    private final LinkedList<Integer> _buf = new
LinkedList<>();
    private final GeneralSemaphore slotsAvailable;
    private final GeneralSemaphore notEmpty;
    private final BinarySemaphore mutex;
    private final int MAX_ITEMS_IN_BUFFER = 10;

    public Buffer2() {
        this.slotsAvailable = new
GeneralSemaphore(MAX_ITEMS_IN_BUFFER);
        this.notEmpty = new GeneralSemaphore(0);
        this.mutex = new BinarySemaphore();
    }

    public boolean isEmpty(){
        return this._buf.isEmpty();
    }

    public void put(int i) {
        this.slotsAvailable.P();
        this.mutex.P();
        this._buf.add(i);
        this.mutex.V();
        // System.out.println("Producer put " + i);
        this.notEmpty.V();
    }

    public int get() {
        this.notEmpty.P();
        this.mutex.P();
        int v = this._buf.removeFirst();
        this.mutex.V();
        // System.out.println("Consumer took " + result);
        this.slotsAvailable.V();
        return v;
    }
}
```

W pozostałych klasach nic nie ulega zmianie (jedynie zamiast klasy Buffer przyjmujemy klasę Buffer2)

Wyniki 2 a,b

Podobnie jak w poprzednim zadaniu wykonujemy kod testujący

```java
import static org.example.Test.runTests;

public class PKmon {
    public static void main(String[] args) {
        runTests(1, 1);      // a. n1 = n2 = 1
        runTests(5, 4);      // b. n1 > n2
        runTests(3, 5);      // b. n1 < n2
    }
}
```

Otrzymujemy następujące wyniki:

Nb of Producers: 1

Nb of Consumers: 1

Each producer will produce: 10 products

Each consumer will consume: 10 products

All producers and consumers have finished

All products have been consumed

Nb of Producers: 5

Nb of Consumers: 4

Each producer will produce: 40 products

Each consumer will consume: 50 products

All producers and consumers have finished

All products have been consumed

Nb of Producers: 3

Nb of Consumers: 5

Each producer will produce: 50 products

Each consumer will consume: 30 products

All producers and consumers have finished

All products have been consumed

Jak widać wyniki są identyczne jak w poprzednim zadaniu zatem program działa poprawnie.

3. Przetwarzanie potokowe z buforem

Do rozwiązania problemu posłuży nam n + 1 semaforów ogólnych (n – liczba wątków przetwarzających, jeden dodatkowy semafor dla konsumenta). Na początku semafory są opuszczone. Producent dodając produkt, podnosi semafor dla pierwszego wątku przetwarzającego i kontynuuje swoje działanie. Po zakończonym przetwarzaniu, wątek podnosi semafor dla następnego wątku, umożliwiając mu rozpoczęcie pracy. Klasy producenta, konsumenta oraz przetwarzacza symulują pracę poprzez wywołanie metody sleep().

Potok wykorzystany w zadaniu:

- Producent dodaje do bufora kolejne liczby
- Do liczby zostaje dodane 9
- Liczba zostaje przemnożona przez 3
- Liczba zostaje zamieniona na napis, napis podwajamy
- Dodajemy słowo „prefix” do napisu (na początku słowa)
- Odwracamy słowo (np. z pies robimy seip)
- Konsument pobiera ostateczną wartość

Program wygląda następująco:

Klasa Buffer3

```
import java.util.concurrent.Semaphore;
import java.util.function.Function;
import java.util.stream.IntStream;

class Buffer3 {
    private final Object[] buffer;
    private final int bufferSize;
```

```
    private final Semaphore[] semaphores;
    private final int transformOperationsNum;

    public Buffer3(int bufferSize, int transformThreads) {
        buffer = new Object[bufferSize];
        this.bufferSize = bufferSize;
        transformOperationsNum = transformThreads + 1;
        semaphores = new Semaphore[transformOperationsNum];
        IntStream.range(0, transformOperationsNum).forEach(i -
> {
            semaphores[i] = new Semaphore(0);
        });
    }

    public int getBufferSize() {
        return bufferSize;
    }

    public void put(int i) {
        buffer[i] = i;
        semaphores[0].release();
    }

    public void transform(int i, int operationIndex,
Function<Object, Object> transformFunction) {
        try {
            semaphores[operationIndex].acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        buffer[i] = transformFunction.apply(buffer[i]);
        semaphores[operationIndex + 1].release();
    }

    public Object get(int i) {
        try {
            semaphores[transformOperationsNum - 1].acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return buffer[i];
    }
}
```

Klasa Consumer2

```
import java.util.Random;
import java.util.List;
import java.util.LinkedList;
import java.util.stream.IntStream;
```

```
class Consumer2 extends Thread {
    private final List<String> results = new LinkedList<>();

    private final Buffer3 _buf;
    private final Random random = new Random();

    public Consumer2(Buffer3 buffer) {
        _buf = buffer;
    }

    public void run() {
        IntStream.range(0, _buf.getBufferSize()).forEach(i ->
{
            try {
                Thread.sleep(random.nextInt(400));
            } catch (InterruptedException ignored) {
            }
            Object obj = _buf.get(i);
            results.add(obj.toString());
        });
    }

    public List<String> getResults() {
        return results;
    }
}
```

Klasa Producer2

```
import java.util.Random;
import java.util.stream.IntStream;

class Producer2 extends Thread {
    private final Buffer3 _buf;
    private final Random random = new Random();

    public Producer2(Buffer3 buffer) {
        _buf = buffer;
    }

    @Override
    public void run() {
        IntStream.range(0, _buf.getBufferSize()).forEach(i ->
{
            try {
                Thread.sleep(random.nextInt(400));
            } catch (InterruptedException ignored) {
            }
            _buf.put(i);
        });
```

```
    }
}
```

Klasa PipeThread

```
import java.util.Random;
import java.util.function.Function;
import java.util.stream.IntStream;

class PipeThread extends Thread {
    private final Function<Object, Object> transformFunction;
    private final Buffer3 _buf;
    private final Random random = new Random();
    private final int operationID;

    public PipeThread(Buffer3 buffer, int operationIndex,
Function<Object, Object> func) {
        transformFunction = func;
        _buf = buffer;
        operationID = operationIndex;
    }

    @Override
    public void run() {
        IntStream.range(0, _buf.getBufferSize()).forEach(i ->
{
            try {
                Thread.sleep(random.nextInt(300));
            } catch (InterruptedException ignored) {
            }
            _buf.transform(i, operationID, transformFunction);
        });
    }
}
```

Klasa Pipe

Odpowiada ona za testowanie programu. Porównuje ona wartości wyliczone przez wątek konsumenta do wartości spodziewanych.

```
import java.util.*;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class Pipe {
    public static void main(String[] args) {
        IntStream.range(0, 10).forEach(i -> runCase());
```

```
    }

    public static void runCase() {
        int BUFFER_SIZE = 100;

        Function<Integer, Integer> addNine = i -> (i + 9);
        Function<Integer, Integer> multByThree = i -> (3 * i);
        Function<Integer, String> duplicateString = i ->
(i.toString() + i.toString());
        Function<String, String> addPrefix = s -> ("prefix" +
s);
        Function<String, String> addSufix = s -> (new
StringBuilder(s).reverse().toString());

        List<Function> transforms = Arrays.asList(
                multByThree,
                addNine,
                duplicateString,
                addPrefix,
                addSufix
        );

        Buffer3 buffer = new Buffer3(BUFFER_SIZE,
transforms.size());
        List<Thread> threadList = new LinkedList<>();
        threadList.add(new Producer2(buffer));
        IntStream.range(0, transforms.size()).forEach(i -> {
            threadList.add(new PipeThread(buffer, i,
transforms.get(i)));
        });

        var consumer = new Consumer2(buffer);
        threadList.add(consumer);
        threadList.forEach(Thread::start);
        threadList.forEach(t -> {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        List<String> calculatedResults =
consumer.getResults();
        List<String> expectedResults = IntStream
                .range(0, BUFFER_SIZE)
                .mapToObj(i -> {
                    List<Object> tempList = new
ArrayList<>(transforms.size());
                    tempList.add(i);
                    transforms.forEach(t ->{
```

```
                        Object o =
tempList.get(tempList.size() - 1);
                        tempList.add(t.apply(o));
                    });
                    return tempList.get(tempList.size() -
1).toString();
                })
                .collect(Collectors.toList());

        Collections.sort(calculatedResults);
        Collections.sort(expectedResults);
        boolean res =
calculatedResults.equals(expectedResults);
        if (res) System.out.println("Result Equals");
        else System.out.println("Result not Equals");
    }
}
```

Wyniki 3

Po wywołaniu programu otrzymujemy

Result Equals

Result Equals

Result Equals

Result Equals

Result Equals

Result Equals

Result Equals

Result Equals

Result Equals

Result Equals

Podane wyniki dowodzą poprawności implementacji.

Odpowiedzmy na pytanie: „**Od czego zależy prędkość obróbki w systemie potokowym?”.**

Prędkość przetwarzania w systemie potokowym zależy w głównej mierze od wydajności poszczególnych wątków biorących udział w przetwarzaniu. To oznacza, że prędkość jest ściśle uzależniona od efektywności każdego wątku w systemie. Nawet jeśli większość wątków działa sprawnie, to jedynie jeden wątek działający suboptymalnie może wpłynąć na wydajność całego systemu. Takie niesprawne działanie jednego wątku jest często określane jako "wąskie gardło" systemu. W praktyce oznacza to, że nawet jeśli większość wątków działa błyskawicznie, wystarczy, że jeden wątek wykonuje skomplikowane obliczenia lub działa w sposób nieoptymalny, aby opóźnić cały proces przetwarzania.

3. Wnioski

Wnioski do zadań 1,2

- W języku Java można wykorzystać wbudowane monitory do zapewnienia bezpiecznego dostępu do współdzielonych zasobów. Wykorzystanie monitorów jest stosunkowo proste i pozwala na tworzenie mechanizmów obsługujących dostęp do tych zasobów.
- Monitory w Javie zapewniają mechanizmy synchronizacji, które pozwalają na kontrolowanie dostępu do współdzielonych danych. Mogą być wykorzystywane do rozwiązania problemów synchronizacji w wielowątkowych aplikacjach.
- Używanie semaforów jest prostsze niż zmiennych warunkowych.
- Problem Producent-Konsument można rozwiązać zarówno za pomocą zmiennych warunkowych, jak i semaforów.
- Warto byłoby rozważyć inną strukturę bufora, np. użycie kolejki FIFO.

Wnioski do zadania 3

- Wydajność systemu potokowego jest ściśle uzależniona od wydajności każdego indywidualnego wątku biorącego udział w przetwarzaniu. Nawet najszybsze i najlepiej zoptymalizowane wątki mogą być opóźnione przez pojedynczy, działający suboptymalnie wątek.
- Suboptymalnie działający wątek staje się "wąskim gardłem" systemu, co oznacza, że stanowi on główne ograniczenie prędkości przetwarzania. W praktyce oznacza to, że cały proces przetwarzania nie może działać szybciej niż ten pojedynczy wątek.
- Optymalizacja wydajności każdego wątku jest kluczowym czynnikiem w osiągnięciu optymalnej prędkości przetwarzania w systemie potokowym.

4. Bibliografia

- Materiały do laboratorium
- Per Brinch Hansen Java Insecure Parallelism
- Pipeline Design Pattern in Java
- Oracle Java Concurrency Tutorial