Lab 13

CSP

Adrian Madej 14.01.2024

1. Treść zadania

Zaimplementuj w Javie z użyciem JCSP rozwiązanie problemu producenta i konsumenta z buforem N-elementowym tak, aby każdy element bufora był reprezentowany przez odrębny proces (taki wariant ma praktyczne uzasadnienie w sytuacji, gdy pamięć lokalna procesora wykonującego proces bufora jest na tyle mała, że mieści tylko jedną porcję). Uwzględnij dwie możliwości:

a) kolejność umieszczania wyprodukowanych elementów w buforze oraz kolejność pobierania nie mają znaczenia

b) pobieranie elementów powinno odbywać się w takiej kolejności, w jakiej były umieszczane w buforze

c) Proszę wykonać pomiary wydajności kodu dla obu przypadków, porównać wydajność z własną implementacją rozwiązania problemu

2. Rozwiązywanie zadań

Zadanie a

Zaimplementujemy problem producenta i konsumenta z buforem N-elementowym nie uwzględniając kolejności umieszczania i pobierania elementów z bufora.

W naszej implementacji komórka bufora będzie posiadała referencję do kanałów producenta, konsumenta i kanału, który będzie sygnalizował czy dana komórka bufora jest dostępna czy zajęta. Każda z komórek bufora ma za zadanie sygnalizować swoją dostępność, oczekując aż jeden z producentów zapisze do niej dane oraz jeden z konsumentów je z niej odczyta.

Producenci korzystają z klasy Alternative, która wybiera jedną z dostępnych komórek bufora (jeśli żadna z komórek nie jest dostępna, to czeka). Po otrzymaniu dostępu, producent oznacza komórkę jako zajętą i wpisuje do niej dane.

Konsumenci również korzystają z klasy Alternative lokalizując komórki z danymi. Po ich odczytaniu, komórka zostaje oznaczona jako dostępna i jest gotowa na kolejne przyjęcie danych.

Implementacja

Implementujemy klasę Buffer

```
class Buffer implements CSProcess {
    private final One2OneChannelInt in;
    private final One2OneChannelInt out;
    private final One2OneChannelInt free;

    public Buffer(One2OneChannelInt in,
                  One2OneChannelInt out,
                  One2OneChannelInt free) {
        this.out = out;
        this.in = in;
        this.free = free;
    }

    public void run() {
        while (true) {
            free.out().write(0);
            out.out().write(in.in().read());
        }
    }
}
```

Implementujemy klasę Producenta dodającą produkty

```
class Producer implements CSProcess {
    private final One2OneChannelInt[] out;
    private final One2OneChannelInt[] free;
    private final int n;

    public Producer(One2OneChannelInt[] out,
One2OneChannelInt[] free, int n) {
        this.out = out;
        this.n = n;
        this.free = free;
    }
```

```
    public void run() {
        var guards = new Guard[free.length];
        for (int i = 0; i < out.length; i++) {
            guards[i] = free[i].in();
        }

        var alt = new Alternative(guards);
        for (int i = 0; i < n; i++) {
            var index = alt.select();
            free[index].in().read();

            var item = (int) (Math.random() * 100) + 1;
            out[index].out().write(item);
        }
    }
}
```

Implementujemy klasę Konsumenta pobierającą produkty

```
class Consumer implements CSProcess {
    private final One2OneChannelInt[] in;
    private final int n;

    public Consumer(final One2OneChannelInt[] in, int n) {
        this.in = in;
        this.n = n;
    }

    public void run() {
        var start = System.currentTimeMillis();
        var guards = new Guard[in.length];
        for (int i = 0; i < in.length; i++)
            guards[i] = in[i].in();

        var alt = new Alternative(guards);
        for (int i = 0; i < n; i++) {
            int index = alt.select();
            int item = in[index].in().read();
        }

        var end = System.currentTimeMillis();
        System.out.println((end - start) + "ms");
        System.exit(0);
    }
}
```

Sprawdzamy działanie programu oraz mierzymy jego czas działania

```
public class Main {
    static final int buffersNum = 10;
    static final int itemsNum = 10000;

    public static void main(String[] args) {
        var channelIntFactory = new
StandardChannelIntFactory();
        var prodChannel =
channelIntFactory.createOne2One(buffersNum);
        var consChannel =
channelIntFactory.createOne2One(buffersNum);
        var bufferChannel =
channelIntFactory.createOne2One(buffersNum);

        var procList = new CSProcess[buffersNum + 2];
        procList[0] = new Producer(prodChannel,
bufferChannel, itemsNum);
        procList[1] = new Consumer(consChannel, itemsNum);

        IntStream.range(0, buffersNum).forEach(i -> {
            procList[i + 2] =
                    new Buffer(
                            prodChannel[i],
                            consChannel[i],
                            bufferChannel[i]
                    );
        });

        new Parallel(procList).run();
    }
}
```

Otrzymaliśmy informację, że program wykonywał się 149ms.

Zadanie b

Zaimplementujemy problem producenta i konsumenta z buforem N-elementowym, gdzie pobieranie elementów odbywa się w takiej kolejności, w jakiej były umieszczane w buforze.

Dane są przetwarzane po kolei (potokowo) - każda komórka bufora przechowuje referencję na swojego następnika. Ostatnia komórka bufora posiada wskazanie na obiekt konsumenta, przez co uzyskamy niejako przetwarzanie potokowe.

Program wygląda następująco: producent wstawia dane do pierwszej komórki bufora, w tym czasie komórki przesuwają dane do następnych komórek, tzn. komórka pierwsza przekaże dane do następnej jeśli jest pusta itd. aż ostatni bufor poda dane do konsumenta, który je przetworzy. Taki sposób przetwarzania zapewni nam kolejność produkowanych i odbieranych danych niezależnie od ilości producentów i konsumentów.

Implementacja

Klasa Buffer

```
class Buffer implements CSProcess {
    private final One2OneChannelInt in;
    private final One2OneChannelInt out;

    public Buffer(One2OneChannelInt in,
                  One2OneChannelInt out) {
        this.out = out;
        this.in = in;
    }

    public void run() {
        while (true) {
            out.out().write(in.in().read());
        }
    }
}
```

Klasa Producenta

```
class Producer implements CSProcess {
    private final One2OneChannelInt out;
    private final int n;

    public Producer(One2OneChannelInt out, int n) {
        this.out = out;
        this.n = n;
    }

    public void run() {
        for (int i = 0; i < n; i++) {
            var item = (int) (Math.random() * 100) + 1;
            out.out().write(item);
        }
    }
}
```

Klasa Konsumenta

```
class Consumer implements CSProcess {
    private final One2OneChannelInt in;
    private final int n;

    public Consumer(final One2OneChannelInt in, int n) {
        this.in = in;
        this.n = n;
    }

    public void run() {
        var start = System.currentTimeMillis();
        for (int i = 0; i < n; i++) {
            int item = in.in().read();
        }

        var end = System.currentTimeMillis();
        System.out.println((end - start) + "ms");
        System.exit(0);
    }
}
```

Klasa Main

```
     public class Main {
    static final int buffersNum = 10;
    static final int itemsNum = 10000;

    public static void main(String[] args) {
        var channelIntFactory = new
StandardChannelIntFactory();
        var channels =
channelIntFactory.createOne2One(buffersNum + 1);

        var procList = new CSProcess[buffersNum + 2];
        procList[0] = new Producer(channels[0], itemsNum);
        procList[1] = new Consumer(channels[buffersNum],
itemsNum);

        IntStream.range(0, buffersNum).forEach(i -> {
            procList[i + 2] =
                    new Buffer(
                            channels[i],
                            channels[i + 1]
                    );
        });

        new Parallel(procList).run();
```

```
    }
}
```

W wyniku wywołania program otrzymaliśmy informację, że wykonywał on się 197ms

Zadanie c

Do własnej implementacji problemu wykorzystam metody wait()/notify()

Rozwiązanie problemu opiera się na wspólnym dostępnie producentów i konsumentów do ograniczonego swoją wielkością bufora. Każdy z wątków (producenta i konsumenta) ma określoną liczbę operacji jaką musi wykonać (producenci dodają zasoby, a konsumenci je pobierają).

Implementacja

Klasa Buffer

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

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) {
            var item = (int) (Math.random() * 100) + 1;
            _buf.put(item);
        }
    }
}
```

Klasa Test służąca do pomiaru czasu

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


        int nbOfProducts = 10000;
        int nbOfProducerOperation = nbOfProducts /
                nbOfProducers;
        int nbOfConsumerOperation = nbOfProducts /
                nbOfConsumers;

        Buffer buffer = new Buffer();
        var start = System.currentTimeMillis();

        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);
            }
        });
        var end = System.currentTimeMillis();
        System.out.println((end - start) + "ms");
    }
}
```

Klasa wykonująca program

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

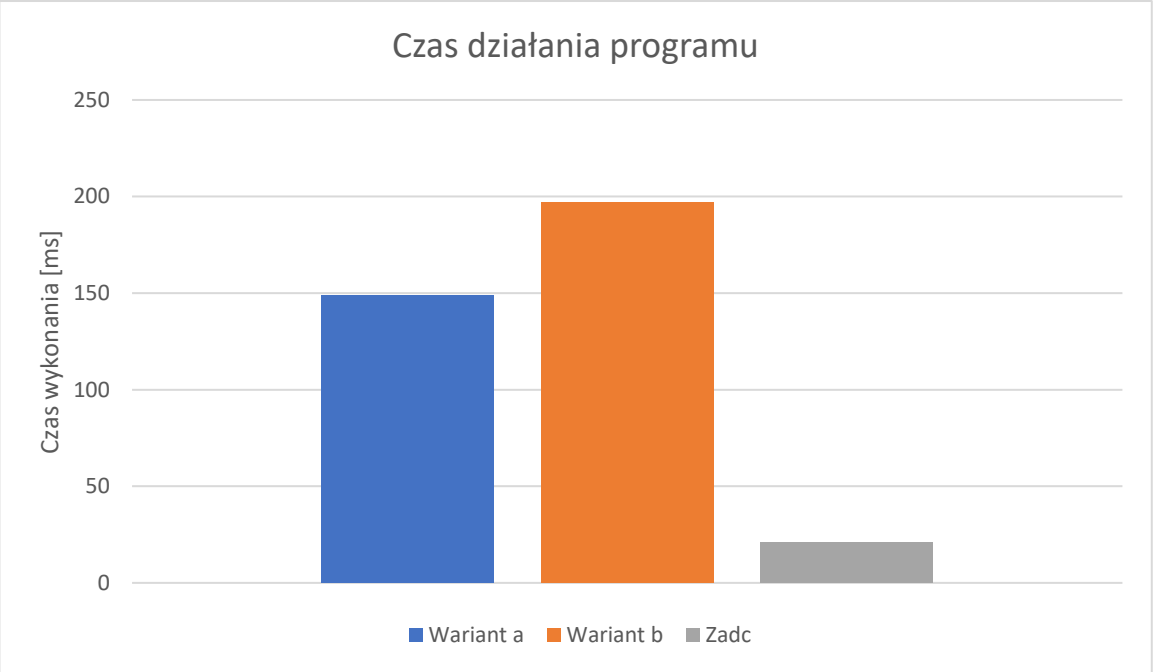
W wyniku uruchomienia programu dostaliśmy czas 21ms.

3. Wyniki

Powyższe programy miały różne czasy działania:

- Wariant a = 149ms
- Wariant b = 197ms
- Wariant c = 21ms,

gdzie poszczególne warianty odpowiadają programom realizowanym w odpowiednich podpunktach.



Porównując czasy wykonania powyższych programów można wyciągnąć wniosek, że wersja nie uwzględniająca kolejności pobierania elementów jest szybsza od tej, która tą kolejność zachowuje. Dzieje się tak dlatego, że dbanie o kolejność zajmuje dodatkowy czas procesora.

Najszybsze jednak okazało się użycie metod wait()/notify(). Nie ma co się dziwić, w końcu w CSP procesy komunikują tylko za pomocą czytania i zapisywania danych poprzez kanały, co wymaga dużo czasu.

4. Wnioski

- Kanały CSP są synchroniczne, a więc nie wspierają wewnętrznego buforowania. Można jednak zbudować procesy, które realizują buforowanie;
- Kanały obsługują tylko odczyt i zapis jako operacje przenoszące dane;
- Podstawowym typem kanałów jest one-to-one. Mogą łączyć tylko jedną parę procesów, pisarza i czytelnika. Można również zdefiniować kanały do odczytu i do zapisu z/do wielu procesów.
- Model CSP może pomóc w projektowaniu systemów, które wykorzystują współbieżność w sposób kontrolowany, co jest istotne w aplikacjach wielowątkowych i rozproszonych.
- Modele CSP są przydatne do projektowania rozproszonych systemów, ponieważ umożliwiają opisywanie komunikacji między procesami na różnych węzłach sieci.

5. Bibliografia

(Każdy podpunkt jest hiperłączem)

- Strona laboratorium
- Communicating Sequential Processes for JavaTM (JCSP)
- Communicating Sequential Processes (CSP)
- Programming Techniques