Laboratorium XIII – Teoria współbieżności

Weronika Szybińska, 22.01.2023

Treść zadania:

1. 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.

Rozwiązanie:

Do wykonania zadań wykorzystana została biblioteka JCSP.

Na początku napisany został program w którym kolejność umieszczania wyprodukowanych elementów w buforze oraz kolejność pobierania nie mają znaczenia. Utworzone zostały 4 klasy: Main, Consumer, Producer, Buffer. Implementacja przedstawiona została poniżej. Ilość buforów oraz ilość przekazywanych elementów jest ustawiana na początku klasy Main, jako odpowiednio bufferSize oraz items.

```java
import org.jcsp.lang.Parallel;
import org.jcsp.lang.CSProcess;
import org.jcsp.lang.One2OneChannelInt;
import org.jcsp.lang.StandardChannelIntFactory;
public final class Main
{
    public static void main (String[] args)
    {
        final int bufferSize = 10;
        final int items = 10000;
        StandardChannelIntFactory factory = new StandardChannelIntFactory();
        final One2OneChannelInt[] channels_producer = factory.createOne2One(bufferSize);
        final One2OneChannelInt[] channels_toGo = factory.createOne2One(bufferSize);
        final One2OneChannelInt[] channels_consumer = factory.createOne2One(bufferSize);

        CSProcess[] processes = new CSProcess[bufferSize + 2];
        processes[0] = new Producer(channels_producer, channels_toGo, items);
        processes[1] = new Consumer(channels_consumer, items);
        for (int i = 0; i < bufferSize; i++) {
            processes[i + 2] = new Buffer(channels_producer[i], channels_consumer[i], channels_toGo[i]);
        }

        Parallel par = new Parallel(processes);
        par.run();
    }
}
```

```java
import org.jcsp.lang.CSProcess;
import org.jcsp.lang.One2OneChannelInt;
import org.jcsp.lang.Guard;
import org.jcsp.lang.Alternative;

public class Producer implements CSProcess
{
    private final One2OneChannelInt[] production;
    private final One2OneChannelInt[] toGo;
    private final int items;
    public Producer (final One2OneChannelInt[] production, final One2OneChannelInt[] toGo, final int items)
    {
        this.production = production;
        this.toGo = toGo;
        this.items = items;
    }
    public void run ()
    {
        final Guard[] guards = new Guard[toGo.length];
        for (int i = 0; i < toGo.length; i++)
            guards[i] = toGo[i].in();
        final Alternative alternative = new Alternative(guards);
        for (int i = 0; i < items; i++)
        {
            int index = alternative.select();
            toGo[index].in().read();
            int item = (int)(Math.random()*100)+1;
            production[index].out().write(item);
        }
    }
}
```

```java
import org.jcsp.lang.CSProcess;
import org.jcsp.lang.One2OneChannelInt;
import org.jcsp.lang.Guard;
import org.jcsp.lang.Alternative;

public class Consumer implements CSProcess
{
    private One2OneChannelInt consumption[];
    private int items;
    public Consumer (final One2OneChannelInt consumption[], final int items)
    {
        this.consumption = consumption;
        this.items = items;
    }
    public void run ()
    {
        long start = System.nanoTime();
        final Guard[] guards = new Guard[consumption.length];
        for (int i = 0; i < consumption.length; i++)
            guards[i] = consumption[i].in();
        final Alternative alternative = new Alternative(guards);
        for (int i = 0; i < items; i++)
        {
            int index = alternative.select();
            int item = consumption[index].in().read();
// System.out.println(index + ": " + item);
        }
        long end = System.nanoTime();
        System.out.println((end - start) + "ns");
        System.exit( status: 0);
    }
}
```

```java
public class Buffer implements CSProcess
{
    private final One2OneChannelInt production;
    private final One2OneChannelInt consumption;
    private final One2OneChannelInt toGO;
    public Buffer (final One2OneChannelInt production,
                   final One2OneChannelInt consumption,
                   final One2OneChannelInt toGO)
    {
        this.consumption = consumption;
        this.production = production;
        this.toGO = toGO;
    }
    public void run ()
    {
        while (true)
        {
            toGO.out().write( 0);
            consumption.out().write(production.in().read());
        }
    }
}
```

W zadaniu drugim zaimplementowane zostało rozwiązanie problemu producenta i konsumenta z buforem N-elementowym, gdzie pobieranie elementów powinno odbywać się w takiej kolejności, w jakiej były umieszczane w buforze. Poniżej przedstawiona jest impelmentacja. Ilość buforów oraz ilość przekazywanych elementów jest ustawiana na początku klasy Main, jako odpowiednio bufferSize oraz items.

```java
import org.jcsp.lang.Parallel;
import org.jcsp.lang.CSProcess;
import org.jcsp.lang.One2OneChannelInt;
import org.jcsp.lang.StandardChannelIntFactory;
public final class Main
{
    public static void main (String[] args)
    {
        final int bufferSize = 10;
        final int items = 10000;
        StandardChannelIntFactory factory = new StandardChannelIntFactory();
        final One2OneChannelInt[] channels = factory.createOne2One( bufferSize + 1);
        CSProcess[] processes = new CSProcess[bufferSize + 2];
        processes[0] = new Producer(channels[0], items);
        processes[1] = new Consumer(channels[bufferSize], items);
        for (int i = 0; i < bufferSize; i++) {
            processes[i + 2] = new Buffer(channels[i], channels[i + 1]);
        }
        Parallel par = new Parallel(processes);
        par.run();
    }
}
```

```java
import org.jcsp.lang.CSProcess;
import org.jcsp.lang.One2OneChannelInt;
public class Producer implements CSProcess
{
    private final One2OneChannelInt consumption;
    private final int items;
    public Producer (final One2OneChannelInt consumption, final int items)
    {
        this.consumption = consumption;
        this.items = items;
    }
    public void run ()
    {
        for (int i = 0; i < items; i++)
        {
            int item = (int)(Math.random()*100)+1;
            consumption.out().write(item);
        }
    }
}
```

```java
import org.jcsp.lang.CSProcess;
import org.jcsp.lang.One2OneChannelInt;
public class Consumer implements CSProcess
{
    private One2OneChannelInt production;
    private int items;
    public Consumer (final One2OneChannelInt production, final int items)
    {
        this.production = production;
        this.items = items;
    }
    public void run ()
    {
        long start = System.nanoTime();
        for (int i = 0; i < items; i++)
        {
            int item = production.in().read();
// System.out.println(item);
        }
        long end = System.nanoTime();
        System.out.println((end - start) + "ns");
        System.exit( status: 0);
    }
}
```

```java
import org.jcsp.lang.CSProcess;
import org.jcsp.lang.One2OneChannelInt;

public class Buffer implements CSProcess
{
    private final One2OneChannelInt production;
    private final One2OneChannelInt consumption;
    public Buffer (final One2OneChannelInt production,
                   final One2OneChannelInt consumption)
    {
        this.consumption = consumption;
        this.production = production;
    }
    public void run ()
    {
        while (true)
            consumption.out().write(production.in().read());
    }
}
```

2. Pomiary wydajności kodu dla obu przypadków, porównać wydajność z własną implementacją rozwiązania problemu. Wyniki podane są w jednostce ns.

Zad.1

1	2	3	4	5	**ŚREDNIA**
166550000	152979300	215691900	159266500	207403300	**180 378 200**

Zad.2

1	2	3	4	5	**ŚREDNIA**
173070100	191878100	153111100	247764500	167401600	**212 022 860**

Własna implementacja z laboratorium III (użycie notify() oraz wait())

1	2	3	4	5	**ŚREDNIA**
26307010	23187810	25311110	24776450	2685460	**26 133 056**

Podsumowanie:

Z wyników można wywnioskować że wariant drugi jest wolniejszy od pierwszego co wydaje się być dosyć oczywiste. W przypadku utrzymywania kolejności w buforze każdy element przekazywany jest przez kanał n razy, gdzie n to liczba buforów. W pierwszym zadaniu za to element przekazywany jest tylko 2 razy. Można przypuszczać, że przy większym rozmiarze bufora różnica wydajności dwóch pierwszych rozwiązań byłaby jeszcze większa. Trzecie rozwiązanie, używające metod wait() i notify() jest ewidentnie szybsze od dwóch poprzednich implementacji. Jest to zapewne spowodowane użyciem jednego dużego bufora zamiast wielu mniejszych.

Bibliografia:

https://www.cs.kent.ac.uk/projects/ofa/jcsp/

https://home.agh.edu.pl/~funika/tw/lab-csp/tw-csp.pdf