SPRAWOZDANIE XIII

TEORIA WSPÓŁBIEŻNOŚCI

CSP

DAWID BIAŁKA

DATA LABORATORIUM 12.01.2021

DATA ODDANIA 26.01.2021

Zad. 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.

Koncepcja

Zadania zostaną rozwiązane przy wykorzystaniu pakietu JCSP, który wspiera konstrukcje CSP reprezentowane przez interfejsy, klasy i metody. Zadanie 1 a) i 1 b) zostaną rozwiązane przy wykorzystaniu podanego pseudokodu. W zadaniu c) bufor nie będzie podzielny a synchronizacja zostanie dokonana przy użyciu semaforów.

Implementacja i wyniki

Zad. 1 a)

Klasa Main

```
public final class MainA
{
  public static void main (String[] args)
  {
    final int numOfBuffers = 10;
    final int numOfItems = 10000;
    StandardChannelIntFactory fact = new StandardChannelIntFactory();
    final One2OneChannelInt channels_prod[] = fact.createOne2One(numOfBuffers);
    final One2OneChannelInt channels_jeszcze[] = fact.createOne2One(numOfBuffers);
    final One2OneChannelInt channels_cons[] = fact.createOne2One(numOfBuffers);
    CSProcess[] procList = new CSProcess[numOfBuffers + 2];
    procList[0] = new ProducerA(channels_prod, channels_jeszcze, numOfItems);
    procList[1] = new ConsumerA(channels_cons, numOfItems);

    for (int i = 0; i < numOfBuffers; i++)
      procList[i + 2] = new BufferA(channels_prod[i], channels_cons[i], channels_jeszcze[i]);

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

Klasa Consumer

```
public class ConsumerA implements CSProcess
{
   private One2OneChannelInt in[];
   private int howMany;

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

   public void run ()
   {
      long start = System.nanoTime();
      final Guard[] guards = new Guard[in.length];
      for (int i = 0; i < in.length; i++)
         guards[i] = in[i].in();
      final Alternative alt = new Alternative(guards);

      for (int i = 0; i < howMany; i++)
      {
         int index = alt.select();
         int item = in[index].in().read();
         // System.out.println(index + ": " + item);
      }

      long end = System.nanoTime();
      System.out.println((end - start) / 1000000);
      System.exit(0);
   }
}
```

Klasa Producer

```
public class ProducerA implements CSProcess
{
   private One2OneChannelInt out[];
   private One2OneChannelInt jeszcze[];
   private int howMany;

   public ProducerA(final One2OneChannelInt out[], final One2OneChannelInt jeszcze[], final int
howMany)
   {
      this.out = out;
      this.jeszcze = jeszcze;
      this.howMany = howMany;
   }

   public void run ()
   {
      final Guard[] guards = new Guard[jeszcze.length];
      for (int i = 0; i < out.length; i++)
         guards[i] = jeszcze[i].in();

      final Alternative alt = new Alternative(guards);

      for (int i = 0; i < howMany; i++)
      {
         int index = alt.select();
         jeszcze[index].in().read();
         int item = (int)(Math.random()*100)+1;
         out[index].out().write(item);
      }
   }
}
```

Klasa Buffer

```
public class BufferA implements CSProcess
{
   private One2OneChannelInt in;
   private One2OneChannelInt out;
   private One2OneChannelInt jeszcze;
   public BufferA (final One2OneChannelInt in, final One2OneChannelInt out, final One2OneChannelInt
jeszcze)
   {
      this.out = out;
      this.in = in;
      this.jeszcze = jeszcze;
   }

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

Przykład działania:

```
0: 60
1: 42
2: 32
9: 71
0: 28
1: 88
2: 11
9: 64
0: 97
1: 40
```

Zad 1 b)

Klasa Main

```
public final class MainB
{
   public static void main (String[] args)
   {
      final int numOfBuffers = 10;
      final int numOfItems = 10000;
      StandardChannelIntFactory fact = new StandardChannelIntFactory();
      final One2OneChannelInt channels[] = fact.createOne2One(numOfBuffers + 1);
      CSProcess[] procList = new CSProcess[nBuffers + 2];
      procList[0] = new ProducerB(channels[0], numOfItems);
      procList[1] = new ConsumerB(channels[numOfBuffers], numOfItems);

      for (int i = 0; i < numOfBuffers; i++)
         procList[i + 2] = new BufferB(channels[i], channels[i + 1]);

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

Klasa Consumer

```
public class ConsumerB implements CSProcess
{
   private One2OneChannelInt in;
   private int howMany;

   public ConsumerB (final One2OneChannelInt in, final int howMany)
   {
      this.in = in;
      this.howMany = howMany;
   }

   public void run ()
   {
      long start = System.nanoTime();

      for (int i = 0; i < howMany; i++)
      {
         int item = in.in().read();
         //System.out.println(item);
      }

      long end = System.nanoTime();
      System.out.println((end - start) + "ns");
      System.exit(0);
   }
}
```

Klasa Producer

```
public class ProducerB implements CSProcess
{
   private One2OneChannelInt out;
   private int howMany;

   public ProducerB(final One2OneChannelInt out, final int howMany)
   {
      this.out = out;
      this.howMany = howMany;
   }

   public void run ()
   {

      for (int i = 0; i < howMany; i++)
      {
         int item = (int)(Math.random()*100)+1;
         out.out().write(item);
         //System.out.println("Producer " + item);
      }
   }
}
```

Klasa Buffer

```
public class BufferB implements CSProcess
{
   private One2OneChannelInt in, out;

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

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

Przykład działania:

```
Producer 65
Producer 45
Producer 33
Producer 86
Producer 62
Producer 82
Producer 9
Producer 39
Producer 61
Producer 89
Producer 84
Consumer 65
Consumer 45
Consumer 33
Consumer 86
Consumer 62
Consumer 82
Consumer 9
Producer 29
Producer 82
Consumer 39
Producer 48
Consumer 61
Consumer 89
Producer 49
Consumer 84
Producer 98
Consumer 29
Producer 91
Consumer 82
Consumer 48
Producer 24
Producer 17
Consumer 49
Consumer 98
```

Zad 1 c)

Klasa Main

```
public final class MainC
{
   public static void main (String[] args)
   {
      final int numOfBufferPlaces = 10;
      final int numOfItems = 10000;
      BufferC buff = new BufferC(numOfBufferPlaces);
      ConsumerC c = new ConsumerC(buff, numOfItems);
      ProducerC p = new ProducerC(buff, numOfItems);
      c.start();
      p.start();
   }
}
```

Klasa Consumer

```
public class ConsumerC extends Thread
{
   private BufferC buff;
   private int howMany;

   public ConsumerC (final BufferC buff, final int howMany)
   {
      this.buff = buff;
      this.howMany = howMany;
   }

   public void run()
   {
      long start = System.nanoTime();
      for (int i = 0; i < howMany; i++)
      {
         int item = buff.get();
         // System.out.println(item);
      }
      long end = System.nanoTime();
      System.out.println((end - start) / 1000000);
      System.exit(0);
   }
}
```

Klasa Producer

```
public class ProducerC extends Thread
{
   private BufferC buff;
   private int howMany;
   public ProducerC (final BufferC buff, final int howMany)
   {
      this.buff = buff;
      this.howMany = howMany;
   }

   public void run()
   {
      for (int i = 0; i < howMany; i++)
      {
         int item = (int)(Math.random()*100)+1;
         buff.put(item);
      }
   }
}
```

Klasa Buffer

```
public class BufferC {
   private int items[];
   private Semaphore freePlaces;
   private Semaphore storedItems;
   private int firstFreePlace;
   private int firstStoredItem;

   public BufferC (int numOfItems)
   {
      items = new int[numOfItems];
      freePlaces = new Semaphore(nItems);
      storedItems = new Semaphore(0);
   }

   public void put(int item)
   {
      try {
         freePlaces.acquire();
      }
      catch (InterruptedException ex) {
         ex.printStackTrace();
         System.exit(-1);
      }

      items[firstFreePlace] = item;
      firstFreePlace = (firstFreePlace + 1) % items.length;
      storedItems.release();
   }

   public int get ()
   {
      try {
         storedItems.acquire();
      }
      catch (InterruptedException ex) {
         ex.printStackTrace();
         System.exit(-1);
      }

      int item = items[firstStoredItem];
      firstStoredItem = (firstStoredItem + 1) % items.length;
      freePlaces.release();
      return item;
   }
}
```

Krótki skrypt w bashu do sprawdzenia czasu działania:

```
#!/bin/bash
for i in {1..5}
do
     java Main
done
```

Wyniki dla wersji z uwzględnianiem kolejności w milisekundach:

166

173

146

138

126

Wyniki dla wersji z uwzględnieniem kolejności w milisekundach:

196

188

282

257

254

Wyniki dla implementacji własnej w milisekundach:

15

16

23

30

17

Wnioski

Widzimy, że dla wersji z uwzględnioną kolejnością czas wykonania jest dłuższy niż dla wersji bez uwzględniania kolejności, czego mogliśmy się spodziewać. Dla wersji wykorzystującej semafory do synchronizacji czas wykonania jest dużo mniejszy (o rząd wielkości). Jest to spowodowane między innymi zastosowaniem jednego bufora zamiast wielu małych.

Bibliografia

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

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html

https://www.ibm.com/developerworks/java/library/j-csp2/