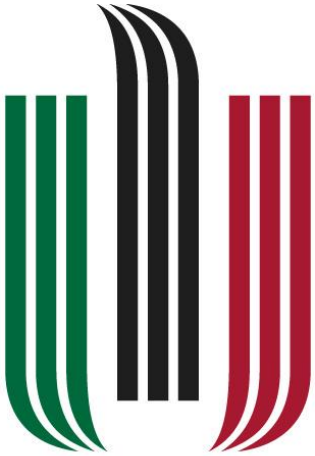
Laboratorium XIII

Communicating sequential processes

Dominik Marek

27 grudnia 2024



AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

1..Zadania

1. Proszę przeanalizować przykładowe rozwiązanie klasycznej postaci problemu producentów i konsumentów, zapisane z użyciem JCSP.

a) Szkielet:

Listing 1. Kod szkieletu

```
public Producer (final One2OneChannelInt out)

      { channel = out;
      } // constructor

    public void run ()
      { int item = (int)(Math.random()*100)+1;
        channel.write(item);
      } // run

  } // class Producer
```

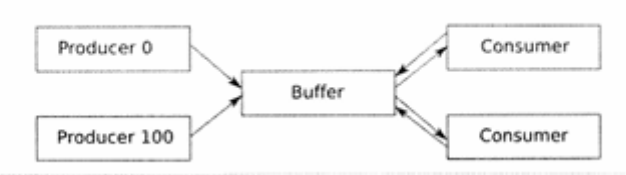
Listing 2. Kod szkieletu

```
/** Consumer class:
  * reads one int from input channel, displays it,
  * then terminates.
  */
public class Consumer implements CSProcess
  { private One2OneChannelInt channel;

    public Consumer (final One2OneChannelInt in)

      { channel = in;
      } // constructor


    public void run ()
      {   int item = channel.read();
          System.out.println(item);
      } // run

  } // class Consumer
```

Listing 3. Kod szkieletu

```
/** Main program class for Producer/Consumer example.
  * Sets up channel, creates one of each process
  * then executes them in parallel, using JCSP.
  */
public final class PCMain
  {
    public static void main (String[] args)
      { new PCMain();
      } // main

    public PCMain ()
      { // Create channel object
        final One2OneChannelInt channel =
          new One2OneChannelInt();

        // Create and run parallel construct
        // with a list of processes

        CSProcess[] procList =
          { new Producer(channel),
            new Consumer(channel) };
        // Processes

        Parallel par =
          new Parallel(procList); // PAR construct
        par.run(); // Execute processes in parallel
      } // PCMain constructor

  } // class PCMain
```

b) Schemat komunikacji



c) Przykładowe pełne rozwiązanie (Załącznik 1)

2.

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. Pseudokod:

Listing 4. Producent i konsument - rozproszony bufor **bez uwzględnienia** kolejności

```
// N – rozmiar bufora;
[PRODUCER:: p: porcja;
*[true -> produkuj(p);
[(i:0..N–1) BUFFER(i)?JESZCZE() -> BUFFER(i)!p]
]
||BUFFER(i:0..N–1):: p: porcja;
*[true -> PRODUCER!JESZCZE() ;
[PRODUCER?p -> CONSUMER!p]
]
||CONSUMER:: p: porcja;
*[(i:0..N–1) BUFFER(i)?p -> konsumuj(p)]
]
```

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

Listing 5. Producent i konsumer - rozproszony bufor **z uwzględnieniem** kolejności

```
// N – rozmiar bufora;
[PRODUCER:: p: porcja;
*[true -> produkuj(p); BUFFER(0)!p]
||BUFFER(i:O..N–1):: p: porcja;
*[true -> [i = 0 -> PRODUCER?p
[]i <> 0 -> BUFFER(i–1)?p];
[i = N–1 -> CONSUMER!p
[]i <> N–1 -> BUFFER(i+1)!p]
]
|| CONSUMER:: p: porcja;
*[BUFFER(N–1)?p -> konsumuj(p)]
]
```

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

2.Rozwiązania

a)

1. Kod klasy Producenta

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

public class Producer implements CSProcess {
    public One2OneChannelInt[] out;
    public One2OneChannelInt[] buffer;
    public int size;

    public Producer(One2OneChannelInt[] out, One2OneChannelInt[] buffer,
int size) {
        this.out = out;
        this.buffer = buffer;
        this.size = size;
    }

    public void run() {
        final Guard[] guards = new Guard[buffer.length];

        for (int idx = 0; idx < out.length; idx++) {
            guards[idx] = buffer[idx].in();
        }

        Alternative alternative = new Alternative(guards);

        for (int idx = 0; idx < size; idx++) {
            int index = alternative.select();
            buffer[index].in().read();
            out[index].out().write((int) ((Math.random() * 100) + 1));
        }
    }

}
```

2. Kod klasy Konsumenta

```
import org.jcsp.lang.Alternative;
import org.jcsp.lang.CSProcess;
```

```
import org.jcsp.lang.Guard;
import org.jcsp.lang.One2OneChannelInt;

public class Consumer implements CSProcess {
    public One2OneChannelInt[] in;
    public int size;

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

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

        final Guard[] guards = new Guard[in.length];

        for (int idx = 0; idx < in.length; idx++) {
            guards[idx] = in[idx].in();
        }

        Alternative alternative = new Alternative(guards);

        for (int idx = 0; idx < size; idx++) {
            int index = alternative.select();
            int item = in[index].in().read();
            System.out.println(index + ": " + item);
        }

        long end = System.nanoTime();

        System.out.println((end - start) + " nanoseconds");
        System.exit(0);
    }
}
```

3.Kod Bufora

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

public class Buffer implements CSProcess {
    public One2OneChannelInt producer, consumer, still;

    public Buffer(One2OneChannelInt prod, One2OneChannelInt cons,
One2OneChannelInt still) {
        this.consumer = cons;
        this.producer = prod;
        this.still = still;
    }

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

4. Kod klasy Main

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

```
import org.jcsp.lang.Parallel;
import org.jcsp.lang.StandardChannelIntFactory;

public class Main {
    static final int buffers = 20;
    static final int items = 100000;

    public static void main(String[] args) {
        StandardChannelIntFactory factory = new
StandardChannelIntFactory();

        One2OneChannelInt[] channelProducer =
factory.createOne2One(buffers);
        One2OneChannelInt[] channelBuffer = factory.createOne2One(buffers);
        One2OneChannelInt[] channelConsumer =
factory.createOne2One(buffers);

        CSProcess[] processes = new CSProcess[buffers + 2];

        processes[0] = new Producer(channelProducer, channelBuffer, items);
        processes[1] = new Consumer(channelConsumer, items);

        for (int idx = 2; idx < buffers + 2; idx++) {
            processes[idx] = new Buffer(channelProducer[idx - 2],
channelConsumer[idx - 2], channelBuffer[idx - 2]);
        }

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

Numer iteracji	*Czas wykonania programu [ns]*
1	3667171100
2	3736502700
3	3682526100
4	3690747500
5	3687940500

b)

1. Kod klasy Producenta

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

public class Producer implements CSProcess {
    public One2OneChannelInt out;
    public int size;

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

    public void run() {
        for (int idx = 0; idx < size; idx++) {
            int item = (int) (Math.random() * 100) + 1;
            out.out().write(item);
        }
    }

}
```

2. Kod klasy Konsumenta

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

public class Consumer implements CSProcess {
    public One2OneChannelInt in;
    public int size;

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

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

        for (int idx = 0; idx < size; idx++) {
            int item = in.in().read();
            System.out.println(item);
        }

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

3. Kod klasy Bufora

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

public class Buffer implements CSProcess {
    public One2OneChannelInt in, out;

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

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

4. Kod klasy Main

```
import org.jcsp.lang.CSProcess;
import org.jcsp.lang.One2OneChannelInt;
import org.jcsp.lang.Parallel;
import org.jcsp.lang.StandardChannelIntFactory;

public class Main {
    static final int buffers = 20;
    static final int items = 100000;

    public static void main(String[] args) {
        StandardChannelIntFactory factory = new
StandardChannelIntFactory();
        One2OneChannelInt[] channels = factory.createOne2One(buffers + 1);
        CSProcess[] processes = new CSProcess[buffers + 2];

        processes[0] = new Producer(channels[0], items);
        processes[1] = new Consumer(channels[buffers], items);

        for (int idx = 2; idx < buffers + 2; idx++) {
            processes[idx] = new Buffer(channels[idx - 2], channels[idx -
1]);
        }

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

Numer iteracji	*Czas wykonania programu [ns]*
1	5195412400
2	4835384100
3	5370982200
4	5283137900
5	4955108000

c)

1.Kod klasy Producenta

```
public class Producer extends Thread {
    private Buffer buff;
    private int size;
    public Producer (final Buffer buff, final int size)
    {
        this.buff = buff;
        this.size = size;
    }
    @Override
    public void run()
    {
        for (int i = 0; i < this.size; i++)
        {
            int item = (int)(Math.random()*100)+1;
            buff.put(item);
        }
    }
}
```

2. Kod klasy Konsumenta

```
public class Consumer extends Thread {
    private Buffer buff;
    private int size;

    public Consumer(final Buffer buff, final int size) {
        this.buff = buff;
        this.size = size;
    }

    @Override
    public void run() {
        long start = System.nanoTime();
        for (int i = 0; i < size; i++) {
            int item = buff.get();
            System.out.println(item);
        }
        long end = System.nanoTime();
        System.out.println((end - start) + "ns");
        System.exit(0);
    }
}
```

3. Kod bufora

```
import java.util.concurrent.Semaphore;
public class Buffer {
    private int items[];
    private Semaphore freePlaces;
    private Semaphore storedItems;
    private int firstFreePlace;
```

```
    private int firstStoredItem;
    public Buffer (int nItems)
    {
        items = new int[nItems];
        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;
    }
}
```

4. Kod klasy Main

```
public final class Main {
    public static void main(String[] args) {
        final int nBufferPlaces = 10;
        final int nItems = 10000;
        Buffer buff = new Buffer(nBufferPlaces);
        Consumer c = new Consumer(buff, nItems);
        Producer p = new Producer(buff, nItems);
        c.start();
        p.start();
    }
}
```

Numer iteracji	*Czas wykonania programu [ns]*
1	189261700
2	144984400
3	139648700
4	161613400
5	166062300

Wnioski:

Wariant drugi (z zachowaną kolejnością) z pewnością będzie wolniejszy od wariantu pierwszego. W tym wariancie każdy wyprodukowany element jest przekazywany przez kanał wielokrotnie (n razy, gdzie n to liczba buforów). Natomiast w wariancie pierwszym każdy element jest przekazywany przez kanał tylko dwa razy (plus jeden pusty komunikat). Przy większym rozmiarze buforów różnica w wydajności między tymi dwoma rozwiązaniami powinna się jeszcze bardziej zwiększyć. Rozwiązanie wykorzystujące podstawowe mechanizmy synchronizacji będzie zdecydowanie szybsze (nawet o rząd wielkości) od rozwiązań opartych na braku współdzielonej pamięci. Z drugiej strony, zastosowanie jednego dużego bufora, zamiast wielu małych, również ma wpływ na wydajność.

3.Bibliografia

- https://www.ibm.com/developerworks/java/library/j-csp2/
- https://en.wikipedia.org/wiki/JCSP
- http://www.cs.kent.ac.uk/projects/ofa/jcsp/
- http://www.jcsp.ie/