1. Cel ćwiczenia

Celem niniejszego ćwiczenia jest zapoznanie się z koncepcją CSP oraz wykonanie rozwiązania problemu ograniczonego bufora z użyciem oprogramowania realizującego koncepcję CSP.

2. Wprowadzenie teoretyczne

2.1. Wprowadzenie do ćwiczenia

Teoria komunikujących się sekwencyjnych procesów (CSP) C.A.R. Hoare'a [1] dostarcza formalne podejście do opisu współbieżności i zbiór technik projektowania współbieżnych programów. W założeniu procesy współbieżne nie mają wspólnej przestrzeni adresowej. Proces CSP może być traktowany jako szczególny rodzaj obiektu typu aktor, w którym:

- procesy nie mają interfejsu metod ani metod, które można wywołać z zewnętrz. Metod zatem nie można wywoływać z wątków. Tak więc nie ma potrzeby jawnego blokowania;
- procesy komunikują tylko za pomocą czytania i zapisywania danych poprzez kanały;
- procesy nie mają tożsamości, a więc do nich nie można jawnie się odwoływać. Jednakże kanały umożliwiają komunikację z dowolnym procesem na drugim końcu kanału;
- procesy nie muszą pracować w pętli w nieskończoność odbierając komunikaty. Mogą pisać i czytać komunikaty na różnych kanałach, jeśli zachodzi taka potrzeba.

Kanał CSP może być rozumiany jako szczególny rodzaj kanału, przy czym:

- kanały 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.

3. Plan/ program ćwiczenia

Pakiet JCSP, opracowany na University of Kent [2], to platforma wykonawcza dla programów współbieżnych w Javie, która wspiera konstrukcje CSP reprezentowane przez interfejsy, klasy i metody, w tym:

- interfejsy **ChannelInput** (wsparcie dla odczytu), **ChannelOutput** (wsparcie dla zapisu) i **Channel** (obsługuje obydwie czynności) działają na argumentach typu **Object**, ale specjalne wersje przewidziane są też dla argumentów typu **int**. Główna klasa to **One2OneChannel**, która wspiera obsługę jednego czytelnika i jednego pisarza.

- interfejs **CSProcess** opisuje procesy wspierając tylko metodę **run**. Klasy **Parallel** and **Sequence** (i inne) mają konstruktory, które przyjmują tablice innych obiektów **CSProcess** i tworzą złożone obiekty (kompozyty).
- operator wyboru **[]** jest obsługiwany za pośrednictwem klasy **Alternative**. Konstruktor przyjmuje tablice z elementami typu **Guard**. **Alternative** wspiera metodę **select**, zwraca ona indeks wskazujący, który z nich może (i powinien) być wybrany. Metoda **fairSelect** działa w ten sam sposób, ale zapewnia dodatkowe gwarancje sprawiedliwości - wybiera sprawiedliwie spośród wszystkich gotowych alternatyw.
- dodatkowe środki programistyczne w JCSP to **timer** (który wykonuje odłożone zapisy i może być również używany do określenia time-out'u w **Alternative**), **Generate** (generuje sekwencje liczb), **Skip** (która nic nie robi - jedna z prymityw CSP), i klasy, które umożliwiają interakcję poprzez GUI.

3.1. 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

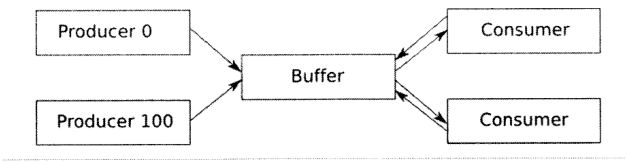
```
/** Consumer class:
  * reads one int from input channel, displays it,
  * then terminates.
  */
public class Consumer implements CSProcess
  { private One2OneChannelInt channel;

    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 w 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.
   d) napisać sprawozdanie dotyczące rozwiązania problemu oraz przeprowadzić analizę wyników eksperymentów.

4. Sposób oceny / uzyskania zaliczenia

- zadania należy robić na zajęciach (oddawanie na danych zajęciach lub na następnych).
- przy spóźnionym (więcej niż 1 tydzień) oddawaniu odejmuje się **0.5 pkt**.
- za aktywność można dostać plusy, które mogą potem podwyższyć ocenę końcową
- przy określeniu oceny brane są pod uwagę: poprawność rozwiązania (50%), argumentacja poprawności (20%), omówienie aspektów wydajnościowych (10%), redakcja sprawozdania (20%).

Literatura

[1] Hoare, C. A. R. Communicating Sequential Processes, Prentice Hall, 1985
[2] JCSP web site: `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`
[3] `http://www.ibm.com/developerworks/java/library/j-csp2/`

Załączniki

1. Przykładowe rozwiązanie klasycznego problemu producentów-konsumentów z użyciem JCSP (Autor: George Wells `G.Wells@ru.ac.za`)

A bounded-buffer (producer-consumer) program using JCSP's message passing

The full version of the solution is shown below, with all the intermediate stages incorporated. The first class shown here is the **producer**. This is fairly straightforward, generating 100 random numbers then sending -1 to indicate that the producer is finished.

```
import jcsp.lang.*;

/** Producer class: produces 100 random integers and sends them on
  * output channel, then sends -1 and terminates.
  * The random integers are in a given range [start...start+100)
  */
public class Producer2 implements CSProcess
  { private One2OneChannelInt channel;
    private int start;

    public Producer2 (final One2OneChannelInt out, int start)
      { channel = out;
        this.start = start;
      } // constructor

    public void run ()
      { int item;
        for (int k = 0; k < 100; k++)
          { item = (int)(Math.random()*100)+1+start;
            channel.write(item);
          } // for
        channel.write(-1);
        System.out.println("Producer" + start + " ended.");
      } // run

  } // class Producer2
```

The **consumer** is a little more complicated. This needs to signal to the buffer process that it is ready to read an item. This is due to the fact that the JCSP "alternative" mechanism only works with "input" channels — there is no direct way of telling whether an output channel is ready to be read. The consumer "signals" the buffer that it is ready-to-read by writing to a second channel, called req in the class below. The buffer process then uses this "request" channel with a guarded alternative construct in order to tell when the consumers are ready to read/consume more data.

```
import jcsp.lang.*;

/** Consumer class: reads ints from input channel, displays them,
then
  * terminates when a negative value is read.
  */
public class Consumer2 implements CSProcess
  { private One2OneChannelInt in;
    private One2OneChannelInt req;

    public Consumer2 (final One2OneChannelInt req, final
One2OneChannelInt in)
      { this.req = req;
        this.in = in;
      } // constructor

    public void run ()
      {  int item;
         while (true)
           { req.write(0); // Request data - blocks until data is
available
             item = in.read();
             if (item < 0)
               break;
             System.out.println(item);
           } // for
         System.out.println("Consumer ended.");
      } // run

  } // class Consumer2
```

The **buffer** is the most complex part of the system. Note how it uses an Alternative construct with the input channels (from the two producers, and the "request" channels from the two consumers) to respond to whichever of the four processes is ready (assuming that space and/or data is available). Note too how the "**clean termination**" is handled, by counting down the number of negative values read from the producers and written to the consumers.

There are a number of other possible solutions to this, and over the years some people have come up with some very ingenious methods using the "select" methods of the Alternative class which take a boolean array of preconditions. Note that using the fairSelect() method with a Skip is somewhat problematic, as it ensures that the skip option gets a fair chance (it is always "ready") even when a producer or consumer may be ready. This is not optimally efficient. Similarly, using a priSelect is not a good idea here, as it will prioritise either the producers or the consumers, depending on how you have ordered your guard array.

```
import jcsp.lang.*;

/** Buffer class: Manages communication between Producer2
  * and Consumer2 classes.
  */
public class Buffer implements CSProcess
  { private One2OneChannelInt[] in; // Input from Producer
    private One2OneChannelInt[] req; // Request for data from
Consumer
    private One2OneChannelInt[] out; // Output to Consumer
    // The buffer itself
    private int[] buffer = new int[10];
    // Subscripts for buffer
    int hd = -1;
    int tl = -1;

    public Buffer (final One2OneChannelInt[] in, final
One2OneChannelInt[] req, final One2OneChannelInt[] out)
      { this.in = in;
        this.req = req;
        this.out = out;
      } // constructor

    public void run ()
      { final Guard[] guards = { in[0], in[1], req[0], req[1] };
        final Alternative alt = new Alternative(guards);
        int countdown = 4; // Number of processes running
        while (countdown > 0)
          { int index = alt.select();
            switch (index)
              { case 0:
                case 1: // A Producer is ready to send
                    if (hd < tl + 11) // Space available
                      { int item = in[index].read();
                        if (item < 0)
                          countdown--;
                        else
                          { hd++;
                            buffer[hd%buffer.length] = item;
                          }
                      }
                    break;
                case 2:
```

```
                case 3: // A Consumer is ready to read
                    if (tl < hd) // Item(s) available
                      { req[index-2].read(); // Read and discard
request
                        tl++;
                        int item = buffer[tl%buffer.length];
                        out[index-2].write(item);
                      }
                    else if (countdown <= 2) // Signal consumer to
end
                      { req[index-2].read(); // Read and discard
request
                        out[index-2].write(-1); // Signal end
                        countdown--;
                      }
                    break;
              } // switch
          } // while
        System.out.println("Buffer ended.");
      } // run

  } // class Buffer
```

Finally, the **main program** is relatively straightforward, simply needing to create the necessary processes and channels and connect them all together appropriately.

```
import jcsp.lang.*;

/** Main program class for Producer/Consumer example.
  * Sets up channels, creates processes then
  * executes them in parallel, using JCSP.
  */
public final class PCMain2
  {
    public static void main (String[] args)
      { new PCMain2();
      } // main

    public PCMain2 ()
      { // Create channel objects
        final One2OneChannelInt[] prodChan = { new
One2OneChannelInt(), new One2OneChannelInt() }; // Producers
        final One2OneChannelInt[] consReq = { new
One2OneChannelInt(), new One2OneChannelInt() }; // Consumer requests
        final One2OneChannelInt[] consChan = { new
One2OneChannelInt(), new One2OneChannelInt() }; // Consumer data

        // Create parallel construct
        CSProcess[] procList = { new Producer2(prodChan[0], 0),
                                 new Producer2(prodChan[1], 100),
                                 new Buffer(prodChan, consReq,
consChan),
                                 new Consumer2(consReq[0],
consChan[0]),
                                 new Consumer2(consReq[1],
consChan[1]) }; // Processes
        Parallel par = new Parallel(procList); // PAR construct
        par.run(); // Execute processes in parallel
      } // PCMain constructor

  } // class PCMain2
```