Teoria Współbieżności
Ćwiczenie 14

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 - kod na Upel'u.

   (a) Szkielet:

Listing 1: Starter

```
1  import org.jcsp.lang.CSProcess;
2  import org.jcsp.lang.Channel;
3  import org.jcsp.lang.One2OneChannelInt;
4  import org.jcsp.lang.Parallel;
5
6  public final class Main {
7    public static void main(String[] args) {
8      new Main();
9    }
10   public Main() {
11     final One2OneChannelInt channel = Channel.
           one2oneInt();
12     CSProcess[] procList = {
13       new Producer(channel),
14       new Consumer(channel)
15     };
16     Parallel par = new Parallel(procList);
17     par.run();
18   }
19
20 }
21
22 class Producer implements CSProcess {
23   private One2OneChannelInt channel;
24   public Producer(final One2OneChannelInt out) {
25     channel = out;
26   }
27   public void run() {
28     int item = (int)(Math.random() * 100) + 1;
29     channel.out().write(item);
30   }
31 }
32
33 class Consumer implements CSProcess {
```

```
34    private One2OneChannelInt channel;
35    public Consumer(final One2OneChannelInt in ) {
36      channel = in;
37    }
38    public void run() {
39      int item = channel.in().read();
40      System.out.println(item);
41    }
42  }
```

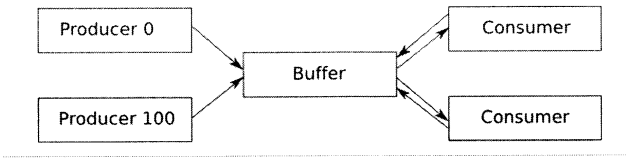
Listing 2: Podpowiedź. Poniższy kod czeka na wiadomość na kanale

```
1   public class MultiDozor implements CSProcess {
2
3     private final One2OneChannelInt in0;
4     private final One2OneChannelInt in1;
5     private final One2OneChannelInt in2;
6
7     public MultiDozor(final One2OneChannelInt in0,
          final One2OneChannelInt in1,
8       final One2OneChannelInt in2) {
9       this.in0 = in0.in();
10      this.in1 = in1.in();
11      this.in2 = in2.in();
12    }
13
14    public void run() {
15
16      final Skip skip = new Skip();
17      final Guard[] guards = {
18        in0.in(),
19        in1.(),
20        in2.(),
21        skip
22      };
23      final Alternative alt = new Alternative(
           guards);
24
25      while (true) {
26        switch (alt.priSelect()) {
27        case 0:
28          //  in0.read()
29          break;
30        case 1:
```

```
31          //  in1.read()
32          break;
33        case 2:
34          // in2.read()
35          break;
36        case 3:
37          // jesli nie ma nic na kanalach
38          break;
39        }
40      } } }
```

Do dodatkowych podpowiedzi proszę spojrzęć na link [4].

(b) Schemat komunikacji



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:

```
1  // N - rozmiar bufora;
2  [PRODUCER:: p: porcja;
3  *[true -> produkuj(p);
4  [(i:0..N-1) BUFFER(i)?JESZCZE() -> BUFFER(i)!p]
5  ]
6  ||BUFFER(i:0..N-1):: p: porcja;
7  *[true -> PRODUCER!JESZCZE() ;
8  [PRODUCER?p -> CONSUMER!p]
9  ]
10 ||CONSUMER:: p: porcja;
11 *[(i:0..N-1) BUFFER(i)?p -> konsumuj(p)] ]
```

Listing 3: Producent i konsument - rozproszony bufor **bez uwzględnienia** kolejności

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

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

Listing 4: Producent i konsumer - rozproszony bufor **z uwzględnieniem** kolejności

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/

[4] Alternative dokumentacja https://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc4/jcsp-doc/org/jcsp/lang/Alternative.html

Uwagi: sluzalec@agh.edu.pl