

Teoria Współbieżności

Ćwiczenie 2

1 Cel ćwiczenia

Celem ćwiczenia jest zapoznanie studentów z mechanizmami synchronizacji w Java opartymi o oczekiwanie na warunek. Kolejnym celem jest rozpoznanie abstrakcyjnego problemu ograniczonego bufora przy producentach i konsumentach oraz jego symulacja przez implementację w Java. W szczególności studenci zostaną zapoznani z przetwarzaniem potokowym z buforem

2 Oczekiwanie na warunek (condition wait)

1. Oczekiwanie na spełnienie określonego warunku
2. Sposób na podział oczekujących zadań na grupy - zadania z każdej grupy czekają na spełnienie innego warunku
3. Zwyczajnie realizowane przez zmienne warunkowe (condition variables)
 - *wait*(C) (w pseudokodzie „await”)
 - *signal*(C)
4. Zmienne warunkowe skojarzone z monitorem to po prostu **nazwane kolejki** monitora.
5. Realizacja oczekiwania na warunek w Javie: **nie ma zmiennych warunkowych** - jest tylko jedna **anonimowa** kolejka *wait*
 - Oczekiwanie w pętli while (realizacja „await” w Javie)

```

1 while (! warunek) {
2     wait(); // usypiamy sie i oddajemy procesor,
           liczmy ze ktos nas obudzi
3 }

```

- Inny proces zmienia wartość zmiennej *warunek* i wykonuje *notifyAll()*

3 Ćwiczenie implementacyjne

Problem ograniczonego bufora (producentów-konsumentów)

Dany jest bufor o rozmiarze m , do którego producent może wkładać dane, a konsument pobierać. Napisz program, który zorganizuje takie działanie producenta i konsumenta, w którym zapewniona będzie własność bezpieczeństwa i żywotności.

Do symulacji użyj metod *wait()* / *notify()*. Procesy działają z różnymi prędkościami, użyj np. *TimeUnit.SECONDS.sleep(-)* / *TimeUnit.MILLISECONDS.sleep(-)*.

Wykonaj pomiar, obserwując zachowanie producentów/konsumentów.

3.1 Wersja „klasyczna”

1. dla przypadku 1 producenta i 1 konsumenta.
2. dla przypadku n_1 producentów i n_2 konsumentów:
 - (a) $n_1 > n_2$
 - (b) $n_1 = n_2$
 - (c) $n_1 < n_2$

Do realizacji bufora masz dowolność: możesz użyć tablicy, zwykłej kolejki albo *BlockingQueue* na przykład *LinkedBlockingQueue*:

```

1 import java.util.concurrent.*;
2
3 public LinkedBlockingQueue< JakasKlasa > q = new
   LinkedBlockingQueue< JakasKlasa >();
4
5 // jeden watek wsadza dane
6 q.put( instancjaJakiesKlasy ); // zapisuje do kolejki
7

```

```

8 // inny watek zbiera z kolejki
9 JakasKlasa tmp = q.take(); // czeka az moze cos sciagnac
   z kolejki

```

3.2 Przetwarzanie potokowe z buforem

1. Proces P_0 będący (tylko) producentem.
2. Proces P_{100} będący (tylko) konsumentem.
3. Procesy P_1, P_2, \dots, P_{99} będące procesami przetwarzającymi. Każdy proces otrzymuje daną wejściową od procesu poprzedniego, jego wyjście zaś jest konsumowane przez proces następny.

Możesz użyć *PipedWriter* oraz *PipedReader*.

```

1 import java.util.concurrent.*;
2
3 // pisanie do kanalu
4 private PipedWriter out = new PipedWriter();
5 out.write( jakisChar );
6
7 // czytanie z kanalu
8 public PipedReader in = new PipedReader( out );
9 char ch = (char)in.read(); // czeka az cos przeczyta

```

Starter

Listing 1: Można wykorzystać poniższy kod

```

1 class Producer extends Thread {
2     private Buffer _buf;
3
4     public void run() {
5         for (int i = 0; i < 100; ++i) {
6             _buf.put(i);
7         }
8     }
9 }
10
11 class Consumer extends Thread {
12     private Buffer _buf;

```

```
13
14     public void run() {
15         for (int i = 0; i < 100; ++i) {
16             System.out.println(_buf.get());
17         }
18     }
19 }
20
21 class Buffer {
22     public void put(int i) {
23
24     }
25
26     public int get() {
27
28     }
29 }
30
31 public class SymulacjaPK {
32     public static void main(String[] args) {
33
34     }
35 }
```

Literatura

- [1] Bruce Eckel, "Thinking in Java" - rozdział o wątkach