Laboratorium III – Teoria współbieżności

Weronika Szybińska, 04.11.2022

Treść zadania:

1. Przy pomocy metod wait()/notify().
   a. dla przypadku 1 producent/1 konsument
   b. dla przypadku n1 producentów/n2 konsumentów (n1>n2, n1=n2, n1<n2)
   c. wprowadzić wywołanie metody sleep() i wykonać pomiary, obserwując zachowanie producentów/konsumentów
2. Przy pomocy operacji P()/V() dla semafora:
   a. n1=n2=1
   b. n1>1, n2>1

Opis rozwiązania:

1. Do wykonania pierwszego zadnia wykorzystany został szkielet dostarczony wraz z treścią zadania. Zmodyfikowane zostały w nim klasy Buffer oraz PKmon

```
33  class Buffer {
34      private final int[] table;
35      private int putInx = 0;
36      private int getInx = 0;
37      private int counter = 0;
38
39      public Buffer(int size){
40          this.table = new int[size];
41      }
42
43      public synchronized void put(int i) {
44          while (counter == table.length) {
45              try {
46                  wait();
47              } catch (InterruptedException e) {
48                  e.printStackTrace();
49              }
50          }
51          table[putInx] = i;
52          putInx ++;
53          if(putInx == table.length){
54              putInx = 0;
55          }
56          counter++;
57          notify();
58      }
```

```
60      public synchronized int get(){
61          while(counter == 0){
62              try {
63                  wait();
64              } catch (InterruptedException e) {
65                  e.printStackTrace();
66              }
67          }
68          int var = table[getInx];
69          table[getInx] = -1;
70          getInx ++;
71          if(getInx == table.length){
72              getInx = 0;
73          }
74          counter --;
75          notify();
76          return var;
77
78      }
79  }
```

Przy konstrukcji nowego obiektu klasy Buffer, podawany jest rozmiar tego bufora, czyli wielkość tablicy w której przechowywane będą dane. Zmienna counter kontroluje ile w danym momencie jest zajętego miejsca w buforze. Gdy counter równa się rozmiarowi bufora, obiekt Producer musi czekać, aż Consumer zwolni miejsce pobierając dane. Odwrotna sytuacja ma miejsce gdy counter równa się 0, wtedy obiekt Consumer musi czekać, aż w buforze pojawią się dane do pobrania. Buffer kontroluje również indeksy pod które Producer wkłada kolejne dane, oraz z których Consumer je pobiera.

Dla podpunktu 1a, obiekt PKmon został zaimplementowany następująco.

```
84  public class PKmon {
85      public static void main(String[] args) throws InterruptedException {
86          Buffer buffer = new Buffer( size: 8);
87
88          Producer producer = new Producer(buffer);
89          Consumer consumer = new Consumer(buffer);
90
91          producer.start();
92          consumer.start();
93
94          producer.join();
95          consumer.join();
96      }
97  }
```

Po włączeniu programu zwrócone zostały następujące wyniki: 0, 1, 2, … 99, czyli dane pobierane były w kolejności w której zostały umieszczone przez producenta. Na podstawie tego można wnioskować że program działa poprawnie.

Do wykonania podpunktu b zadania 1, klasa PKmon została zmodyfikowana, aby można było określić ilość producentów i konsumentów.

W przypadku, gdy n1 = n2, (n1>1, n2>1), program zwracał dane w odpowiedniej ilości, lecz w losowej kolejności.

W przypadku, gdy liczba producentów była mniejsza niż liczba konsumentów, program zwracał poszczególne wartości w ilości równej n1, lecz nie kończył się, gdyż konsumenci czekali na kolejne dane do pobrania.

W przypadku, gdy liczba producentów była większa, poszczególne wartości były zwracane przez konsumenta w losowych ilościach oraz tak jak w poprzednim przypadku program nie kończył się sam, gdyż producenci czekali na zwolnienie się miejsca w buforze, aby włożyć tam kolejne dane.

Następnie wykonując podpunkt c w metodach run obu klas została dodana metoda sleep(). Zmiana ta spowodowała, że gdy n1=n2 oraz n1>0 i n2>0, program zwracał wartości w prawidłowej kolejności. Dla przypadków, gdy n1>n2 lub n1<n2, wprowadzenie metody sleep() nie spowodowało żadnych zmian w zwracanych wynikach.

2. W zadaniu drugim zmodyfikowany został kod wykonany na potrzeby poprzednich laboratoriów. Użyto klasy Semafor do synchronizowania działania producentów i konsumentów.

```
class Semafor {
    private int _czeka;

    public Semafor(int availableResources) {
        _czeka = availableResources;
    }


    public synchronized void P() throws InterruptedException {
        while(_czeka <= 0){
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        _czeka -= 1;
    }

    public synchronized void V() throws InterruptedException {
        _czeka += 1;
        try {
            notify();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

```
77  class Buffer2 {
78      private final int[] table;
79      private final Semafor semafor1;
80      private final Semafor semafor2;
81      private int putInx = 0;
82      private int getInx = 0;
83
84      public Buffer2(int maxSize, Semafor semafor1, Semafor semafor2){
85          this.semafor1 = semafor1;
86          this.semafor2 = semafor2;
87          table = new int[maxSize];
88      }
89      public synchronized void  put(int i) throws InterruptedException
90          semafor1.P();
91          table[putInx] = i;
92          putInx ++;
93          if(putInx == table.length){
94              putInx = 0;
95          }
96          semafor2.V();
97      }
98
99      public int get() throws InterruptedException {
100         semafor2.P();
101         int var = table[getInx];
102         table[getInx] = -1;
103         getInx ++;
104         if(getInx == table.length){
105             getInx = 0;
106         }
107         semafor1.V();
108         return var;
109     }
110 }
```

Dzięki użyciu semaforów licznikowych, nie było potrzeby dodatkowego liczenia ile w danym momencie jest miejsca w buforze, tak jak w przypadku zadania 1, gdyż miejsce ograniczał sam semafor.

W przypadku podpunktu a, gdy ilość producentów oraz konsumentów była równa 1, program zwracał jak w przypadku podpunktu 1a, wartości w kolejności ich wkładania do bufora przez producenta.

W przypadku podpunktu b, powtórzyła się sytuacja z podpunktu 1b. Gdy ilość producentów i konsumentów była sobie równa program zwracał odpowiednie wartości i kończył się sam. Gdy

ilości te się różniły program nie kończył się sam, gdyż czekał na zakończenie działania producentów lub konsumentów w zależności od ich ilości.

WNIOSKI:

W obu zadaniach udało się zaimplementować przetwarzanie potokowe danych. Działały one jednak tylko w przypadku równej ilości producentów oraz konsumentów. Gdy ilości te różniły się, przetwarzanie potokowe blokowało się, gdyż programy czekały w nieskończoność na swoją kolej.

BIBLIOGRAFIA:

https://home.agh.edu.pl/~funika/tw/lab3/

http://www.w3big.com/pl/html/thread-procon.html#gsc.tab=0

https://pl.wikipedia.org/wiki/Problem_producenta_i_konsumenta