

Laboratorium Nr 3
"Problem ograniczonego bufora i Przetwarzanie potokowe z
buforem"
Radosław Kopeć
20.10.2020

I. Zadanie 1

1. Treść zadania

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

Zrealizować program:

1. Przy pomocy metod `wait()/notify()`.
 - a. dla przypadku 1 producent/1 konsument
 - b. dla przypadku n_1 producentów/ n_2 konsumentów ($n_1 > n_2$, $n_1 = n_2$, $n_1 < n_2$)
 - c. wprowadzić wywołanie metody `sleep()` i wykonać pomiary, obserwując zachowanie producentów/konsumentów
2. Przy pomocy operacji `P()/V()` dla semafora:
 - a. $n_1 = n_2 = 1$
 - b. $n_1 > 1$, $n_2 > 1$

2. Koncepcja rozwiązania

Buffer będzie zawierał tablice boolean, będzie ona mówić czy można z danego pola w odpowiadającej jej tablicy intów czytać czy pisać (true/false). Podczas operacji put w buforze będzie sprawdzany warunek czy można pisać, jeśli tak to będzie szukane odpowiednie pole do wpisania. Następnie pole to będzie wypełniane wartością, a w tablicy boolean odpowiednie pole będzie ustawione na wartość określającą możliwości czytania. Czytający wątek zostanie obudzony metoda `notifyAll()`. Jeśli nie będzie pola do wpisania, wątek zostanie uśpiony. Obudzi się on dopiero gdy jakiś czytelnik wywoła `notifyAll()`. Analogiczne jest działanie czytelnika z tym że sprawdza on warunek możliwości czytania.

3. Implementacja i wyniki:
zmienne i konstruktor Bufora:

```
private int[] buffer;  
private boolean[] canWrite;  
private final int size;  
  
public Buffer(int size){  
    this.size = size;  
    buffer = new int[size];  
    canWrite = new boolean[size];  
  
    for (int i =0;i < size; i++){  
        buffer[i]= 0;  
        canWrite[i] = true;  
    }  
}
```

metoda sprawdzająca czy można czytać:

```
/** return false if all fields of buffor are not avaiable to write */  
/** It checks is at least one true in canWrite */  
private boolean canWrite(){  
    for (int i = 0; i< size; i++) {  
        if(canWrite[i]){  
            return true;  
        }  
    }  
    return false;  
}
```

metoda znajdujaca pole do wpisania wartosci:

```
private int findFieldToWrite(){  
    for(int i=0;i<size;i++){  
        if(canWrite[i]){  
            return i;  
        }  
    }  
    throw new IllegalStateException("at least one files should be free");  
}
```

Analogiczne metody do czytania:

```
private int findFieldToRead(){
    for(int i=0;i<size;i++){
        if(!canWrite[i]){
            return i;
        }
    }
    throw new IllegalStateException("at least one files should be free");
}

/** return false if all fields of buffor are not avaiable to write */
/** It checks is at least one false in canWrite */
private boolean canRead(){
    for (int i = 0; i < size; i++) {
        if(!canWrite[i]){
            return true;
        }
    }
    return false;
}
```

metoda put:

```
public synchronized void put(int value) {

    while (!canWrite()) {

        try {
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }

    int index = findFieldToWrite();
    buffer[index] = value;
    canWrite[index] = false;
    notifyAll();
}
```

Metoda get:

```
public synchronized int get() {  
    while (!canRead()) {  
        try {  
            wait();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
  
    int index = findFieldToRead();  
    canWrite[index] = true;  
    notifyAll();  
    return buffer[index];  
}  
}
```

Jest to uniwersalne rozwiązanie dla podpunktu a i b.
Oto metody testujące i rezultaty:

Podpunkt A:

```
2  
3 ▶ public class PodpunktA {  
4 ▶     public static void main(String[] args) {  
5         Buffer buffer = new Buffer( size: 10);  
6         Producer producer = new Producer(buffer);  
7         Consumer consumer = new Consumer(buffer);  
8  
9         producer.start();  
10        consumer.start();  
11    }  
12 }  
13
```

Wynik wywołania:

```
C:\Users\Radek\.jdk\openjdk-14.0.
0
10
1
11
13
14
15
16
12
17
19
20
21
22
23
24
18
2
3
4
5
6
7
8
9
```

Nie zmieścił się pełny wynik.

Podpunkt B:

```
2
3 ▶ public class PodpunktB {
4 ▶     public static void main(String[] args) {
5         Buffer buffer = new Buffer( size: 10);
6         int n1 = 5;
7         int n2 = 5;
8         for(int i=0;i<n1 ;i++){
9             Producer producer = new Producer(buffer);
10            producer.start();
11        }
12        for(int i=0;i<n2 ;i++){
13            Consumer consumer = new Consumer(buffer);
14            consumer.start();
15        }
16    }
17 }
18
19 |
```

Rezultat wywołania:

```
0
12
11
15
0
17
10
0
16
14
13
1
2
0
1
20
3
19
18
3
2
```

Nie zmieścił się pełny wynik.

Podpunkt C kod:

Producer:

```
public void run() {
    for (int i = 0; i < 100; ++i) {
        try {
            sleep( millis: 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        _buf.put(i);
    }
}
```

Konsumer:

```
public void run() {
    for (int i = 0; i < 100; ++i) {
        try {
            sleep( millis: 2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(_buf.get());
    }
}
```

Buffer:

```
int index = findFieldToWrite();
buffer[index] = value;
canWrite[index] = false;
notifyAll();
System.out.print("Produkuje: [ " );
for (int v: buffer
    ) {
    System.out.print(" " + v + ",");
}
System.out.println("]");
}
```

```
}

int index = findFieldToRead();
canWrite[index] = true;
notifyAll();
System.out.println("Konsumuje: " + buffer[index]);
return buffer[index];
}
```

Program testujący:

```
public class PodpunktC {
    public static void main(String[] args) {
        Buffer buffer = new Buffer( size: 10);
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);

        producer.start();
        consumer.start();
    }
}
```

Rezultat wywołania:

```
C:\Users\Radek\.jdk\openjdk-14.0.1\bin\java.exe "-javaagent
Produkuje: [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,]
Konsumuje: 0
0
Produkuje: [ 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,]
Produkuje: [ 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,]
Konsumuje: 1
1
Produkuje: [ 3, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,]
Produkuje: [ 3, 2, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0,]
Konsumuje: 3
3
Produkuje: [ 5, 2, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0,]
Produkuje: [ 5, 2, 4, 6, 0, 0, 0, 0, 0, 0, 0, 0,]
Konsumuje: 5
5
Produkuje: [ 7, 2, 4, 6, 0, 0, 0, 0, 0, 0, 0, 0,]
Produkuje: [ 7, 2, 4, 6, 8, 0, 0, 0, 0, 0, 0, 0,]
Konsumuje: 7
7
Produkuje: [ 9, 2, 4, 6, 8, 0, 0, 0, 0, 0, 0, 0,]
Produkuje: [ 9, 2, 4, 6, 8, 10, 0, 0, 0, 0, 0, 0,]
Konsumuje: 9
9
Produkuje: [ 11, 2, 4, 6, 8, 10, 0, 0, 0, 0, 0, 0,]
Produkuje: [ 11, 2, 4, 6, 8, 10, 12, 0, 0, 0, 0, 0,]
Konsumuje: 11
11
Produkuje: [ 13, 2, 4, 6, 8, 10, 12, 0, 0, 0, 0, 0,]
|
```


Po jakimś czasie:

```
17
Produkuje: [ 19, 2, 4, 6, 8, 10, 12, 14, 16, 18,]
Konsumuje: 19
19
Produkuje: [ 20, 2, 4, 6, 8, 10, 12, 14, 16, 18,]
Konsumuje: 20
20
Produkuje: [ 21, 2, 4, 6, 8, 10, 12, 14, 16, 18,]
Konsumuje: 21
21
Produkuje: [ 22, 2, 4, 6, 8, 10, 12, 14, 16, 18,]
Konsumuje: 22
22
Produkuje: [ 23, 2, 4, 6, 8, 10, 12, 14, 16, 18,]
Konsumuje: 23
23
Produkuje: [ 24, 2, 4, 6, 8, 10, 12, 14, 16, 18,]
Konsumuje: 24
24
Produkuje: [ 25, 2, 4, 6, 8, 10, 12, 14, 16, 18,]
```

4. Wnioski

Podpunkt a:

Oczywiście wyniki nie zmieściły się na screen-shocie, jednak są one poprawne. Dla podpunktu a: jeden producer i jeden konsumer, mamy w wyniku ciąg unikalnych liczb od 0 - 99.

Podpunkt b:

Dla podpunktu b liczby powtarzają się każda 5 razy, jest to spowodowane tym że mamy 5 producentów i 5 konsumerów. Wyniki również są poprawne.

Podpunkt c:

W wynikach możemy zauważyć producer stopniowo zajmuje pola buffera. Można też zauważyć po krótkim czasie, że cały buffer jest wypełniony. Konsumer zaczyna konsumować i zwalniając jedno pole, natychmiast jest wrzucany od niego nowy produkt.

5. Bibliografia

<http://home.agh.edu.pl/~funika/tw/lab3/>

I. Zadanie 2

1. Koncepcja rozwiązania

Jako że na poprzednich laboratoriach nauczyliśmy się implementować semaforey i mutexy to w tym ćwiczeniu wykorzystam wbudowaną klasę Semaphore z biblioteki java.util.concurrent. Wykorzystamy trzy semaforey. Jeden będzie określał ile pól można odczytać, na początku będzie on ustawiony na 0. Drugi będzie określał ile jest wolnych pól do zapisu, na początek będzie on ustawiony na rozmiar bufora. Trzeci semafor natomiast będzie pilnował dostępu do tablicy boolean związanej z każdym polem bufora. Jej działanie jest analogiczne jak w poprzednim punkcie.

canWrite[1] = true oznacza, że można pisać do pola buffer[1], natomiast canWrite[1] = false, oznaczałoby, że można z tego pola czytać. Za każdym razem jak pisarz zapisze coś do bufora zwiększy licznik semafora do czytania, dając możliwość wątkom czytającym odblokować się. Gdy buffer się zapełni, semafor do pisania będzie miał wartość 0. Pisarz zatrzyma się wtedy na wejściu do semafora do pisania. Czytelnik odczytując jedną wartość z bufora poinformuje o tym pisarza zwiększając wartość semafora do pisania o jeden.

2. Implementacja i wyniki:

Zmienne i konstruktor:

```
class Buffer {  
  
    private final int[] buffer;  
    private final boolean[] canWrite;  
    private final int size;  
    private final Semaphore readSemaphore;  
    private final Semaphore writeSemaphore;  
    private final Semaphore indexSemaphore;  
  
    public Buffer(int size){  
        this.size = size;  
        buffer = new int[size];  
        canWrite = new boolean[size];  
  
        for (int i =0; i < size; i++){  
            buffer[i]= 0;  
            canWrite[i] = true;  
        }  
        writeSemaphore = new Semaphore( permits: size-1);  
        indexSemaphore = new Semaphore( permits: 1);  
        readSemaphore = new Semaphore( permits: 0);  
    }  
}
```

Funkcja znajdująca wolne pole do pisania:

```
private int findFieldToWrite(){
    for(int i=0;i<size;i++){
        if(canWrite[i]){
            canWrite[i] = false;
            return i;
        }
    }
    throw new IllegalStateException("at least one field should be free to write");
}
```

Funkcja znajdująca wolne pole do czytania:

```
private int findFieldToRead(){
    for(int i=0;i<size;i++){
        if(!canWrite[i]){
            canWrite[i] = true;
            return i;
        }
    }
    throw new IllegalStateException("at least one field should be free to read");
}
```

Operacja put:

```
public void put(int value) {
    try {
        writeSemaphore.acquire();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    try {
        indexSemaphore.acquire();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    int index = findFieldToWrite();

    indexSemaphore.release();

    buffer[index] = value;

    readSemaphore.release();
}
```

Operacja get:

```
public int get(){  
  
    try {  
        readSemaphore.acquire();  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
  
    try {  
        indexSemaphore.acquire();  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
  
    int index = findFieldToRead();  
  
    indexSemaphore.release();  
    int result = buffer[index];  
    writeSemaphore.release();  
  
    return result;  
}
```

Podpunkt A:

```
package producersconsumers.zad2;  
  
public class PodpunktA {  
    public static void main(String[] args) {  
        Buffer buffer = new Buffer( size: 10);  
        Producer producer = new Producer(buffer);  
        Consumer consumer = new Consumer(buffer);  
  
        producer.start();  
        consumer.start();  
    }  
}
```

Wyniki wywołania:

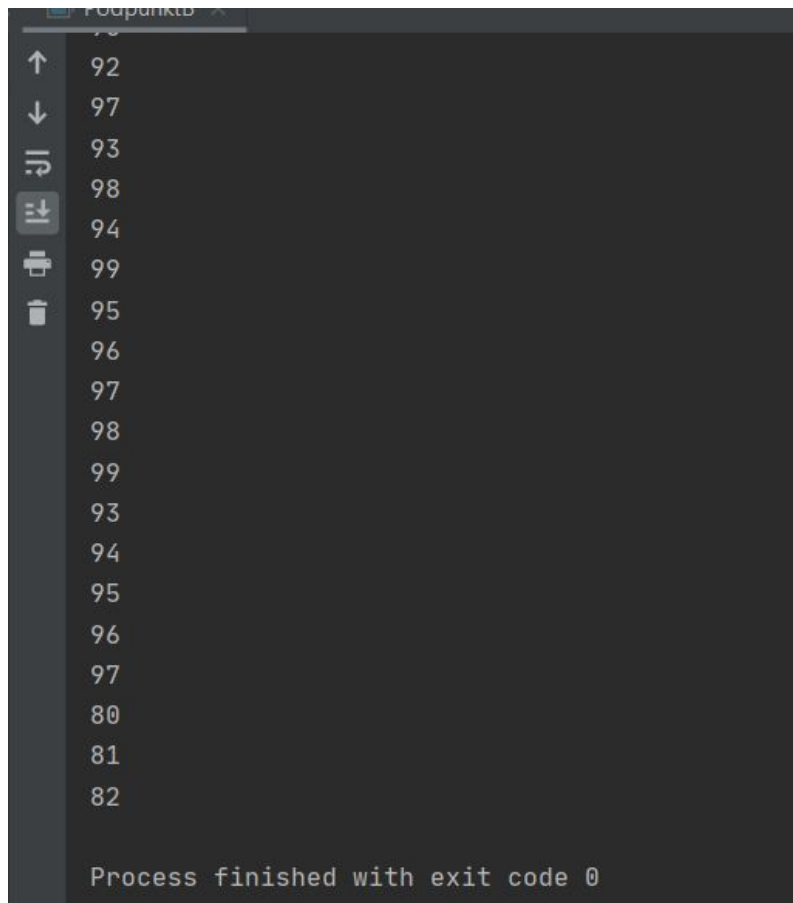
```
C:\Users\Radek\.jdk\openjdk-14.0.1\bin\java.exe "-javaagent:C:\Program F
0
9
1
2
3
10
14
15
16
17
18
19
20
```

Podpunkt B:

```
zad2\buffer.java x zad2\PodpunktB.java x PodpunktA.java x zad2\Produ
package producersconsumers.zad2;

public class PodpunktB {
    public static void main(String[] args) {
        Buffer buffer = new Buffer( size: 10);
        int n1 = 5;
        int n2 = 5;
        for(int i=0;i<n1 ;i++){
            Producer producer = new Producer(buffer);
            producer.start();
        }
        for(int i=0;i<n2 ;i++){
            Consumer consumer = new Consumer(buffer);
            consumer.start();
        }
    }
}
```

Wyniki wywołania:



```
↑ 92
↓ 97
↶ 93
↷ 98
⇅ 94
🖨 99
🗑 95
96
97
98
99
93
94
95
96
97
80
81
82

Process finished with exit code 0
```

3. Wnioski

Podpunkt a:

Oczywiście wyniki nie zmieściły się na screenshocie, jednak są one poprawne. Dla podpunktu a (jeden producer i jeden konsumer) mamy w wyniku ciąg unikalnych liczb od 0 - 99.

Podpunkt b:

Dla podpunktu b liczby powtarzają się każda 5 razy, jest to spowodowane tym że mamy 5 producentów i 5 konsumentów. Wyniki również są poprawne.

Ogólne:

Implementacja problemu producentów i konsumentów z użyciem semaforów przyszła mi dużo łatwiej, kod jest mniejszy i czytelniejszy.

4. Bibliografia

<http://home.agh.edu.pl/~funika/tw/lab3/>

I. Zadanie 2 Przetwarzanie potokowe

1. Treść zadania

- Bufor o rozmiarze N - wspólny dla wszystkich procesów!
- Proces A będący producentem.
- Proces Z będący konsumentem.
- Procesy B, C, ..., Y będące procesami przetwarzającymi.

Każdy proces otrzymuje dane wejściowe od procesu poprzedniego, jego wyjście zaś jest konsumowane przez proces następny. Procesy przetwarzają dane w miejscu, po czym przechodzą do kolejnej komórki bufora i znowu przetwarzają ją w miejscu. Procesy działają z różnymi prędkościami.

Uwaga:

1. W implementacji nie jest dozwolone korzystanie/implementowanie własnych kolejek FIFO, należy używać tylko mechanizmu monitorów lub semaforów !
2. Zaimplementować rozwiązanie przetwarzania potokowego (Przykładowe założenia: bufor rozmiaru 100, 1 producent, 1 konsument, 5 uszeregowanych procesów przetwarzających.) Od czego zależy prędkość obróbki w tym systemie ? Rozwiązanie za pomocą semaforów lub monitorów (dowolnie). Zrobić sprawozdanie z przetwarzania potokowego.

2. Koncepcja rozwiązania

Nadajemy każdemu wątkowi (spośród N wątków) w pipeline numer. Wyjątkowymi wątkami są producent nr 0, oraz konsument nr N . Pomiedzy nimi znajdują się wątki przetwarzające o numerach kolejno 1, 2, ..., $N-1$. Dla mnie wątek przetwarzający będzie dodawał do liczby w buforze wartość 1. Ważna tutaj jest kolejność, aby każda liczba została przetworzona przez kolejne wątki zaczynając od 0, kończąc na N .
Po pierwsze: zmodyfikuję klasy Producer i Consumer tak aby przyjmowały swój numer.

Po drugie: Dodam klasę `ProcessBetweenProducerAndConsumer`, która będzie obrazować wewnętrzne procesy pipelineu.

Działanie klasy `Buffer`.

Aby zapewnić kolejność w przetwarzaniu stworzymy tablicę `"whoCanModify"` typu `int`. Tablica ta będzie tej samej długości co `buffer`, będzie ona przetrzymywać numer wątku który ma prawo dostępu do danego pola `buffera`. Na początku będzie ona wyzerowana, co oznacza że tylko pisarz może coś zrobić z polami `buffera`.

Za każdym razem jak wątek wykona operacje na danym polu `buffera`, zmieni wartość odpowiedniego pola w tablicy `whoCanModify` na numer kolejnego wątku, który może przetwarzać. W szczególności wątek `N` przekaże prawo dostępu do pola wątkowi `0`, aby `buffer` się nie zakleszczył.

`Buffer` przechowywać będzie również mutexy dla każdego z wątków. Będą one symbolizować ilość operacji w buforze do przerobienia przez dany wątek. Dzięki nim możliwa będzie synchronizacja. Wątek który przetworzy swoje pole powiadomi następny wątek poprzez inkrementację jego mutexu.

3. Implementacja i wyniki:

Producer:

```
class Producer extends Thread {
    private final Buffer _buf;
    private final int number;
    public Producer(Buffer buffer, int number) {
        super();
        this._buf = buffer;
        this.number = number;
    }

    public void run() {

        for (int i = 0; i < 100; ++i) {

            _buf.put(i, number, nextProcess: number+1);

        }

    }
}
```


Konsumer:

```
class Consumer extends Thread {
    private final Buffer _buf;
    private final int number;
    public Consumer(Buffer buffer,int number){
        super();
        this._buf = buffer;
        this.number =number;
    }

    public void run() {
        for (int i = 0; i < 100; ++i) {

            System.out.println(_buf.get(number, nextProcess: 0));

        }
    }
}
```

Klasa wątków przetwarzających:

```
package pipelining;

class ProcessBetweenProducerAndConsumer extends Thread {
    private final Buffer _buf;
    private final int number;
    public ProcessBetweenProducerAndConsumer(Buffer buffer,int number){
        super();
        this._buf = buffer;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 100; ++i) {
            _buf.calculate(number, nextProcess: number+1);
        }
    }
}
```

Buffer:

```
5      class Buffer {
6
7          public final int[] buffer;
8          private final int[] whoCanModify;
9          private final Semaphore[] countOfOperations;
10         private final int size;
11         // Semaphore for whoCanModify table
12         private final Semaphore indexSemaphore;
13
14         public Buffer(int size, int processesCount, int firstProcessNumber){
15             this.size = size;
16             buffer = new int[size];
17             whoCanModify = new int[size];
18             countOfOperations = new Semaphore[size];
19             for(int i=0;i< processesCount;i++){
20                 if(i == firstProcessNumber){
21                     countOfOperations[i] = new Semaphore(size);
22                 }
23                 else {
24                     countOfOperations[i] = new Semaphore( permits: 0);
25                 }
26             }
27             for (int i =0;i < size; i++){
28                 buffer[i]= 0;
29                 whoCanModify[i] = 0;
30             }
31
32             indexSemaphore = new Semaphore( permits: 1);
33         }
34     }
```

```
private int findFieldForSpecificNumber(int processNumber){
    try {
        indexSemaphore.acquire();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    for(int i=0;i<size;i++){
        if(whoCanModify[i] == processNumber){
            indexSemaphore.release();
            return i;
        }
    }
    indexSemaphore.release();
    throw new IllegalStateException("at least one field should be free to modify");
}
```

```

private void giveToTheNextProcess(int index,int whoIsNext){
    try {
        indexSemaphore.acquire();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    whoCanModify[index] = whoIsNext;
    countOfOperations[whoIsNext].release();
    indexSemaphore.release();
}

```

```

public void put(int value,int processNumber, int nextProcess) {
    try {
        countOfOperations[processNumber].acquire();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    int index = findFieldForSpecificNumber(processNumber);
    buffer[index] = value;
    giveToTheNextProcess(index,nextProcess);
}

```

```

public void calculate(int processNumber, int nextProcess){
    try {
        countOfOperations[processNumber].acquire();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    int index = findFieldForSpecificNumber(processNumber);
    buffer[index] = buffer[index] + 1;
    giveToTheNextProcess(index,nextProcess);
}

```

```

public int get(int processNumber,int nextProcess){

    try {
        countOfOperations[processNumber].acquire();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    int index = findFieldForSpecificNumber(processNumber);
    int result = buffer[index];
    giveToTheNextProcess(index,nextProcess);

    return result;
}
}

```

Program testuacy:

```

> public class TestPipeline {
>     public static void main(String[] args) {

//         We numerate processes from producer to consumer
        int countOfProcessesInPipeline = 10;
//         init buffer for 12 threads
        Buffer buffer = new Buffer( size: 15, processesCount: 10 + 2, firstProcessNumber: 0);

        ArrayList<Thread> threads = new ArrayList<>();

//         create Producer
        threads.add(new Producer(buffer, number: 0));
//         create threads between producer and consumer
        for(int i=0;i<countOfProcessesInPipeline; i++){
            threads.add(new ProcessBetweenProducerAndConsumer(buffer, number: i+1));
        }
//         create consumer
        threads.add(new Consumer(buffer, number: countOfProcessesInPipeline + 1));

//         run pipeline
        for(Thread thread: threads){
            thread.start();
        }
    }
}

```

```

19 // create threads between producer and consumer
20 for(int i=0; i<countOfProcessesInPipeline; i++){
21     threads.add(new ProcessBetweenProducerAndConsumer(buffer, number: i+1));
22 }
23 // create consumer
24 threads.add(new Consumer(buffer, number: countOfProcessesInPipeline + 1));
25 // run pipeline
26 for(Thread thread: threads){
27     thread.start();
28 }
29 // wait for threads
30 for(Thread thread: threads){
31     try {
32         thread.join();
33     } catch (InterruptedException e) {
34         e.printStackTrace();
35     }
36 }
37 // At the end of the buffer we should see
38 // 10 to 109, because each thread add 1 to the number
39
40 System.out.println(Arrays.toString(buffer.buffer));
41 }
42 }
43

```

Wyniki:

```

TestPipeline x
C:\Users\Radek\.jdk\openjdk-14.0.1\bin\java.exe "-javaagent:C:\Program Files
10
11
12
13
14
15
16
17
24
18
25
26
27

```

```

5 97
2 105
106
107
108
109
99
100
101
102
103
104
[105, 106, 107, 108, 109, 99, 100, 94, 95, 96, 97, 101, 102, 103, 104]

```


4. Wnioski

- Uzyskaliśmy spodziewane rezultaty, producent wyprodukował liczby od 0-99. Następnie wątki pośredniczące dodały do nich 1. I na koniec konsument wyświetla je na ekran.
- W przetwarzaniu potokowym istotna jest kolejność przetwarzania.

Od czego zależy prędkość obróbki w tym systemie ?

Gdy jeden wątek spowalnia działanie w przetwarzaniu potokowym, cały proces zostaje spowolniony. Prędkość obróbki w tym systemie zależy więc od najwolniejszego wątku w pipeline.

Cytat wikipedii:

"Bloki-następniki są zależne od pracy (danych i niezawodności) swoich, niekoniecznie bezpośrednich, bloków-poprzedników - jeśli np. pierwszy blok wpadnie w nieskończoną pętlę, to żaden z następników nigdy nie wykona swojej pracy."

5. Bibliografia

<http://home.agh.edu.pl/~funika/tw/lab3/>

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

Radosław Kopeć 305333