

# Teoria Współbieżności - Sprawozdanie 3

Maciej Brzeżawski

Październik 2024

## 1 Treść zadania

### 1.1 Problem ograniczonego bufora (producentów-konsumentów)

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()`. Kod szkieletu:
  - dla przypadku 1 producent/1 konsument,
  - dla przypadku  $n_1$  producentów/ $n_2$  konsumentów (gdzie  $n_1 > n_2$ ,  $n_1 = n_2$ ,  $n_1 < n_2$ ),
  - wprowadzić wywołanie metody `sleep()` i wykonać pomiary, obserwując zachowanie producentów/konsumentów.
2. przy pomocy operacji `P()/V()` dla semafora:
  - $n_1 = n_2 = 1$ ,
  - $n_1 > 1$ ,  $n_2 > 1$ .

### 1.2 Przetwarzanie potokowe z buforem

Zaimplementować rozwiązanie przetwarzania potokowego (Przykładowe założenia: bufor rozmiaru 100, 1 producent, 1 konsument, 5 uszeregowanych procesów przetwarzających.) Od czego zależy predkosc obrobki w tym systemie ? Rozwiązanie za pomocą semaforów lub monitorów (dowolnie). Zrobić sprawozdanie z przetwarzania potokowego.

## 2 Implementacja problemu ograniczonego bufora

### 2.1 Przy pomocy metod wait()/notify()

#### 2.1.1 1 producent/1 konsument

```
class Buffer {
    private int data;
    private boolean available = false;

    public synchronized void put(int value) {
        while (available) { // Czekaj, aż bufor będzie pusty
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        data = value;
        available = true;
        notifyAll(); // Powiadamia konsumenta, że bufor ma dane
    }

    public synchronized int get() {
        while (!available) { // Czekaj, aż bufor będzie pełny
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        available = false;
        notifyAll(); // Powiadamia producenta, że bufor jest pusty
        return data;
    }
}

class Producer extends Thread {
    private final Buffer buffer;

    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
```

```

        for (int i = 0; i < 100; ++i) {
            buffer.put(i);
            System.out.println("Producent włożył: " + i);
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

class Consumer extends Thread {
    private final Buffer buffer;

    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        for (int i = 0; i < 100; ++i) {
            int value = buffer.get();
            System.out.println("Konsument pobrał: " + value);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

public class PKmon {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);

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

```

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Progr  
Konsument pobrał: 0  
Producent włożył: 0  
Producent włożył: 1  
Konsument pobrał: 1  
Producent włożył: 2  
Konsument pobrał: 2  
Producent włożył: 3  
Konsument pobrał: 3  
Producent włożył: 4  
Konsument pobrał: 4  
Producent włożył: 5  
Konsument pobrał: 5  
Producent włożył: 6  
Konsument pobrał: 6  
Producent włożył: 7  
Konsument pobrał: 7  
Producent włożył: 8  
Konsument pobrał: 8  
Producent włożył: 9  
Konsument pobrał: 9  
Producent włożył: 10  
Konsument pobrał: 10  
Producent włożył: 11  
Konsument pobrał: 11  
Producent włożył: 12  
Konsument pobrał: 12  
Producent włożył: 13  
Producent włożył: 14  
Konsument pobrał: 13  
Konsument pobrał: 14  
Producent włożył: 15
```

tw3 > src > main > java > org > example > PKmon.java > Buffer

Rysunek 1: Wynik działania programu dla przypadku 1 producent/1 konsument

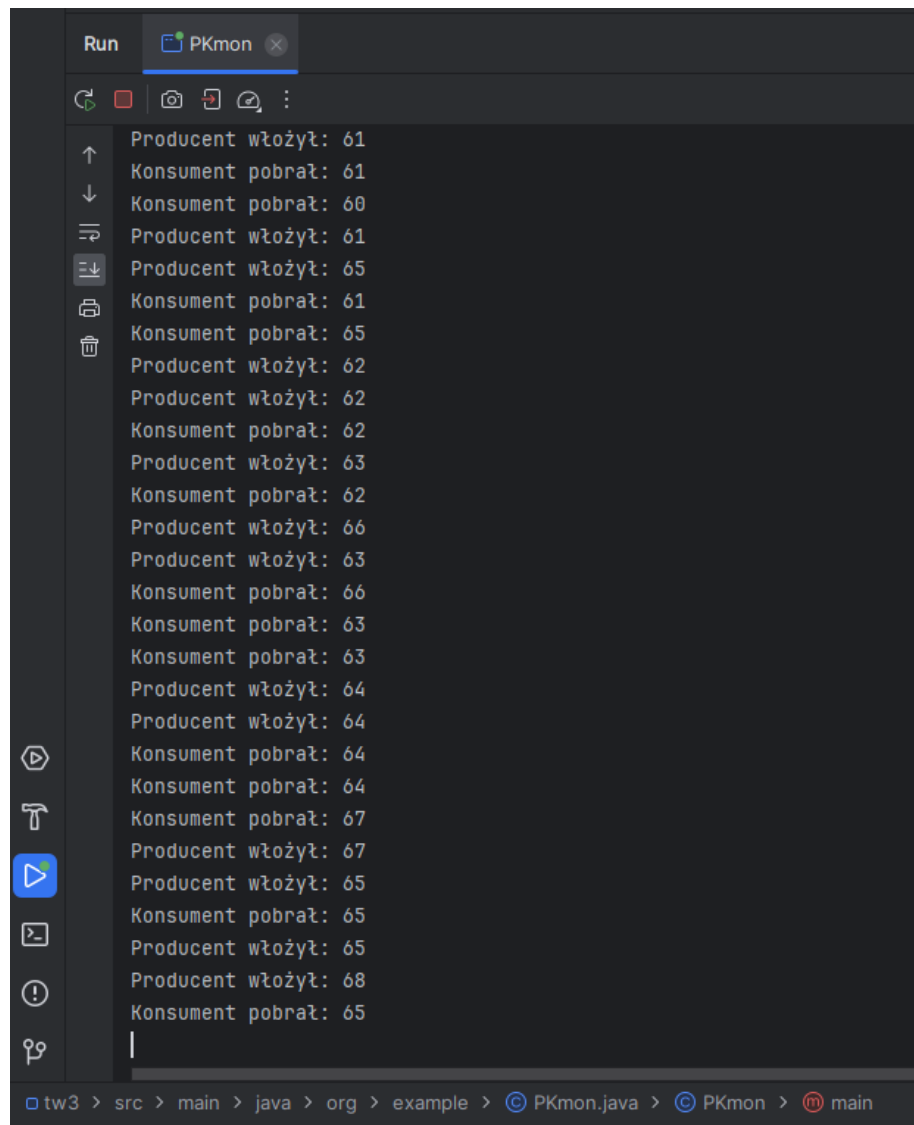
### 2.1.2 n1 producentów/n2 konsumentów (n1>n2, n1=n2, n1<n2)

#### 1. n1>n2

```
public class PKmon {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        int n1 = 3; // liczba producentów
        int n2 = 2; // liczba konsumentów

        for (int i = 0; i < n1; i++) {
            new Producer(buffer).start();
        }
        for (int i = 0; i < n2; i++) {
            new Consumer(buffer).start();
        }
    }
}
```

Program zawiesza się, ponieważ użycie pojedynczej zmiennej available i metod wait()/notifyAll() nie zapewnia właściwej synchronizacji dla więcej niż jednego producenta i jednego konsumenta. Gdy kilku producentów i konsumentów jednocześnie próbuje korzystać z bufora, może dojść do sytuacji, w której producenci czekają na wait() z powodu ustawienia available na true przez inny wątek, podczas gdy konsumenci pozostają w stanie oczekiwania na notifyAll() – co skutkuje zakleszczeniem. Brak wieloelementowego bufora i niewłaściwe sterowanie stanem available prowadzi do sytuacji, gdzie wątki nie mogą kontynuować pracy, czekając na siebie nawzajem.



```
Run PKmon
Producent włożył: 61
Konsument pobrał: 61
Konsument pobrał: 60
Producent włożył: 61
Producent włożył: 65
Konsument pobrał: 61
Konsument pobrał: 65
Producent włożył: 62
Producent włożył: 62
Konsument pobrał: 62
Producent włożył: 63
Konsument pobrał: 62
Producent włożył: 66
Producent włożył: 63
Konsument pobrał: 66
Konsument pobrał: 63
Konsument pobrał: 63
Producent włożył: 64
Producent włożył: 64
Konsument pobrał: 64
Konsument pobrał: 64
Konsument pobrał: 67
Producent włożył: 67
Producent włożył: 65
Konsument pobrał: 65
Producent włożył: 65
Producent włożył: 68
Konsument pobrał: 65
|
```

tw3 > src > main > java > org > example > PKmon.java > PKmon > main

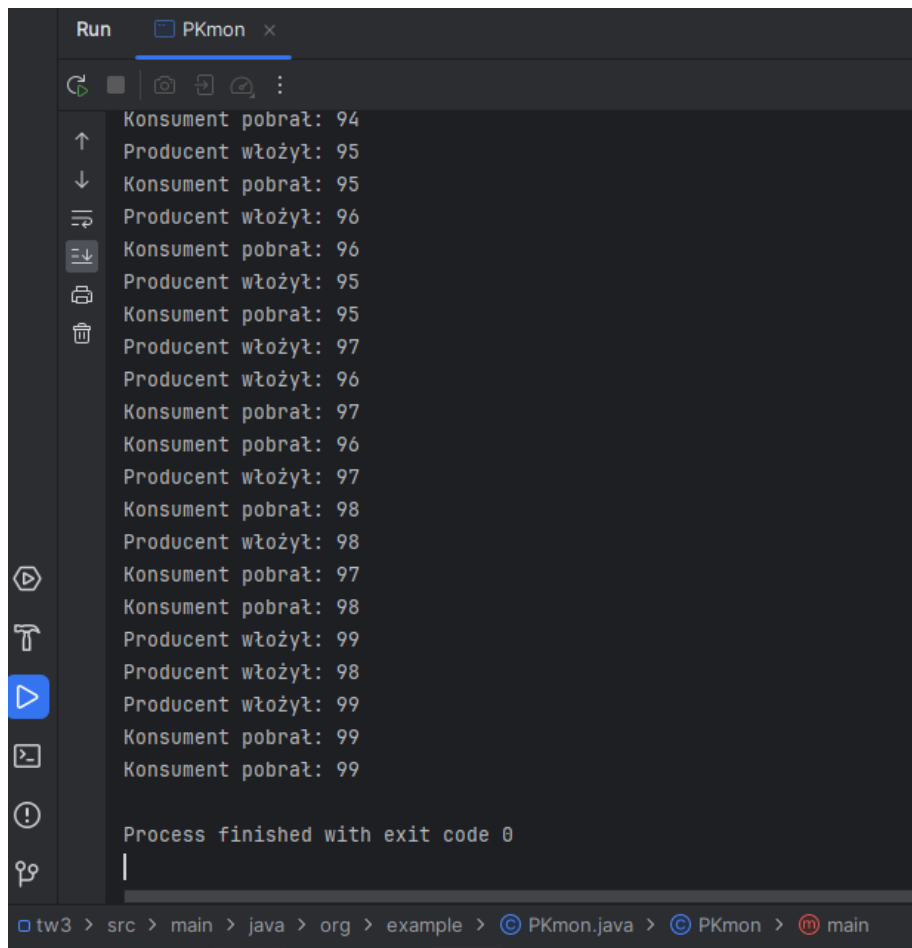
Rysunek 2: Wynik działania programu dla przypadku  $n1 > n2$

## 2. $n1=n2$

```
public class PKmon {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        int n1 = 2; // liczba producentów
        int n2 = 2; // liczba konsumentów

        for (int i = 0; i < n1; i++) {
            new Producer(buffer).start();
        }
        for (int i = 0; i < n2; i++) {
            new Consumer(buffer).start();
        }
    }
}
```

W przypadku, gdy liczba producentów i konsumentów jest równa ( $n1=2$ ,  $n2=2$ ), program działa poprawnie, ponieważ liczba wątków produkujących dane równoważy się z liczbą wątków je konsumujących. Synchronizacja oparta na jednej zmiennej `available` jest wystarczająca, aby każdy z producentów i konsumentów miał dostęp do bufora w przewidywalny sposób, unikając sytuacji, w której któryś z wątków pozostaje w nieskończoność w stanie oczekiwania. Zrównoważenie liczby producentów i konsumentów pomaga w zachowaniu ciągłości dostępu do bufora, minimalizując ryzyko zakleszczeń i gwarantując własności żywotności oraz bezpieczeństwa.



```
Run PKmon x
Konsument pobrał: 94
Producent włożył: 95
Konsument pobrał: 95
Producent włożył: 96
Konsument pobrał: 96
Producent włożył: 95
Konsument pobrał: 95
Producent włożył: 97
Producent włożył: 96
Konsument pobrał: 97
Konsument pobrał: 96
Producent włożył: 97
Konsument pobrał: 98
Producent włożył: 98
Konsument pobrał: 97
Konsument pobrał: 98
Producent włożył: 99
Producent włożył: 98
Producent włożył: 99
Konsument pobrał: 99
Konsument pobrał: 99

Process finished with exit code 0
```

tw3 > src > main > java > org > example > PKmon.java > PKmon > main

Rysunek 3: Wynik działania programu dla przypadku  $n1=n2$



### 3. $n1 < n2$

```
public class PKmon {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        int n1 = 2; // liczba producentów
        int n2 = 3; // liczba konsumentów

        for (int i = 0; i < n1; i++) {
            new Producer(buffer).start();
        }
        for (int i = 0; i < n2; i++) {
            new Consumer(buffer).start();
        }
    }
}
```

W przypadku, gdy liczba konsumentów jest większa niż liczba producentów ( $n1 < n2$ ), program pozornie dochodzi do końca, ale nie kończy swojego działania. Wynika to z faktu, że wszyscy producenci zakończyli wstawianie danych, podczas gdy konsumentom pozostaje oczekiwanie na nowe dane, które już nie nadejdą. W efekcie, gdy bufor jest pusty, konsumenci wchodzi w stan oczekiwania (`wait()`) i pozostają w nim na stałe, ponieważ żaden producent nie wywoła już `notifyAll()`, by ich obudzić. Powoduje to sytuację, w której część konsumentów pozostaje w nieskończoność w stanie oczekiwania, co uniemożliwia poprawne zakończenie programu.

```
Run PKmon x
Producent włożył: 95
Konsument pobrał: 95
Producent włożył: 96
Konsument pobrał: 96
Producent włożył: 96
Konsument pobrał: 96
Producent włożył: 97
Konsument pobrał: 97
Producent włożył: 97
Producent włożył: 98
Konsument pobrał: 97
Producent włożył: 98
Konsument pobrał: 98
Konsument pobrał: 98
Producent włożył: 99
Konsument pobrał: 99
Producent włożył: 99
Konsument pobrał: 99
|
Process finished with exit code 130
tw3 > src > main > java > org > example > PKmon.java > PKmon
```

Rysunek 4: Wynik działania programu dla przypadku  $n1 < n2$

## 2.2 Przy pomocy operacji P()/V() dla semafora

### 2.2.1 n1=n2=1

```
import java.util.concurrent.Semaphore;

class Buffer {
    private int data;
    private final Semaphore empty = new Semaphore(1); // Reprezentuje puste miejsce
    private final Semaphore full = new Semaphore(0);  // Reprezentuje pełny bufor

    public void put(int value) {
        try {
            empty.acquire(); // P() na pustym
            data = value;
            System.out.println("Producent włożył: " + value);
            full.release(); // V() na pełnym, informuje konsumenta
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public int get() {
        try {
            full.acquire(); // P() na pełnym
            int value = data;
            System.out.println("Konsument pobrał: " + value);
            empty.release(); // V() na pustym, informuje producenta
            return value;
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return -1;
        }
    }
}

class Producer extends Thread {
    private final Buffer buffer;

    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        for (int i = 0; i < 100; ++i) {
            buffer.put(i);
        }
    }
}
```

```

        }
    }
}

class Consumer extends Thread {
    private final Buffer buffer;

    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        for (int i = 0; i < 100; ++i) {
            buffer.get();
        }
    }
}

public class PKmon {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);

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

```

```
Run PKmon x
Konsument pobrał: 89
Producent włożył: 90
Konsument pobrał: 90
Producent włożył: 91
Konsument pobrał: 91
Producent włożył: 92
Konsument pobrał: 92
Producent włożył: 93
Konsument pobrał: 93
Producent włożył: 94
Konsument pobrał: 94
Producent włożył: 95
Konsument pobrał: 95
Producent włożył: 96
Konsument pobrał: 96
Producent włożył: 97
Konsument pobrał: 97
Producent włożył: 98
Konsument pobrał: 98
Producent włożył: 99
Konsument pobrał: 99

Process finished with exit code 0

tw3 > src > main > java > org > example > PKmon.java
```

Rysunek 5: Wynik działania programu dla przypadku  $n1=n2=1$

### 2.2.2 $n1 > 1, n2 > 1$

```
import java.util.concurrent.Semaphore;

class Buffer {
    private int data;
    private final Semaphore empty = new Semaphore(1); // Początkowo bufor jest pusty
    private final Semaphore full = new Semaphore(0); // Początkowo bufor nie jest pełny
    private final Semaphore mutex = new Semaphore(1); // Sekcja krytyczna

    public void put(int value) {
        try {
            empty.acquire(); // P() na pustym - czeka, aż bufor będzie pusty
            mutex.acquire(); // Zabezpiecza dostęp do sekcji krytycznej
            data = value;
            System.out.println("Producent włożył: " + value);
            mutex.release(); // Uwalnia dostęp do sekcji krytycznej
            full.release(); // V() sygnalizuje konsumentowi, że bufor jest pełny
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public int get() {
        try {
            full.acquire(); // P() na pełnym - czeka, aż bufor będzie pełny
            mutex.acquire(); // Zabezpiecza dostęp do sekcji krytycznej
            int value = data;
            System.out.println("Konsument pobrał: " + value);
            mutex.release(); // Uwalnia dostęp do sekcji krytycznej
            empty.release(); // V() sygnalizuje producentowi, że bufor jest pusty
            return value;
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return -1;
        }
    }
}

class Producer extends Thread {
    private final Buffer buffer;

    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }
}
```

```

        public void run() {
            for (int i = 0; i < 100; ++i) {
                buffer.put(i);
            }
        }
    }

    class Consumer extends Thread {
        private final Buffer buffer;

        public Consumer(Buffer buffer) {
            this.buffer = buffer;
        }

        public void run() {
            for (int i = 0; i < 100; ++i) {
                buffer.get();
            }
        }
    }

    public class PKmon {
        public static void main(String[] args) {
            Buffer buffer = new Buffer();

            int n1 = 3; // Liczba producentów
            int n2 = 2; // Liczba konsumentów

            for (int i = 0; i < n1; i++) {
                new Producer(buffer).start();
            }
            for (int i = 0; i < n2; i++) {
                new Consumer(buffer).start();
            }
        }
    }
}

```





### 3 Przetwarzanie potokowe z buforem

```
import java.util.concurrent.Semaphore;

class Buffer {
    private final int[] data = new int[100];
    private int index = 0; // wskaźnik bieżącej pozycji w buforze
    private final Semaphore[] stages; // Semaforey dla każdego etapu przetwarzania
    private final Semaphore mutex = new Semaphore(1); // Zabezpieczenie sekcji krytycznej

    public Buffer(int stagesCount) {
        // Inicjujemy semaforey dla każdego etapu przetwarzania (rozmiar stagesCount + 1)
        stages = new Semaphore[stagesCount + 1];
        for (int i = 0; i <= stagesCount; i++) {
            stages[i] = new Semaphore(i == 0 ? 100 : 0);
        }
    }

    public void put(int value) {
        try {
            stages[0].acquire(); // Oczekuje na wolne miejsce w buforze
            mutex.acquire();
            data[index] = value;
            System.out.println("Producent włożył: " + value + " na pozycję " + index);
            mutex.release();
            stages[1].release(); // Odblokowuje pierwszy etap przetwarzania
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public void process(int stage) {
        try {
            stages[stage].acquire(); // Oczekuje na odblokowanie poprzedniego procesu
            mutex.acquire();
            data[index] += stage; // Przykładowe przetwarzanie - modyfikacja danych
            System.out.println("Proces " + stage + " przetworzył dane na pozycji " + index);
            mutex.release();
            stages[stage + 1].release(); // Odblokowuje kolejny etap
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public void get() {
        try {
```

```

        stages[stages.length - 1].acquire(); //Czeka, aż dane będą gotowe do konsumpcji
        mutex.acquire();
        int value = data[index];
        System.out.println("Konsument pobrał: " + value + " z pozycji " + index);
        index = (index + 1) % 100; // Przechodzi do następnej pozycji w buforze
        mutex.release();
        stages[0].release(); // Odblokowuje producenta, aby dodał nowe dane
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
}

class Producer extends Thread {
    private final Buffer buffer;

    public Producer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        for (int i = 0; i < 1000; ++i) { // Symulacja ciągłej produkcji danych
            buffer.put(i);
            try {
                Thread.sleep(50); // Symulacja różnej prędkości działania
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

class Processor extends Thread {
    private final Buffer buffer;
    private final int stage;

    public Processor(Buffer buffer, int stage) {
        this.buffer = buffer;
        this.stage = stage;
    }

    public void run() {
        while (true) {
            buffer.process(stage);
            try {
                Thread.sleep(100); // Symulacja różnej prędkości działania
            }
        }
    }
}

```

```

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

class Consumer extends Thread {
    private final Buffer buffer;

    public Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    public void run() {
        while (true) {
            buffer.get();
            try {
                Thread.sleep(75); // Symulacja różnej prędkości działania
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

public class PipelineProcessing {
    public static void main(String[] args) {
        Buffer buffer = new Buffer(5); // Inicjalizacja bufora z 5 etapami przetwarzania

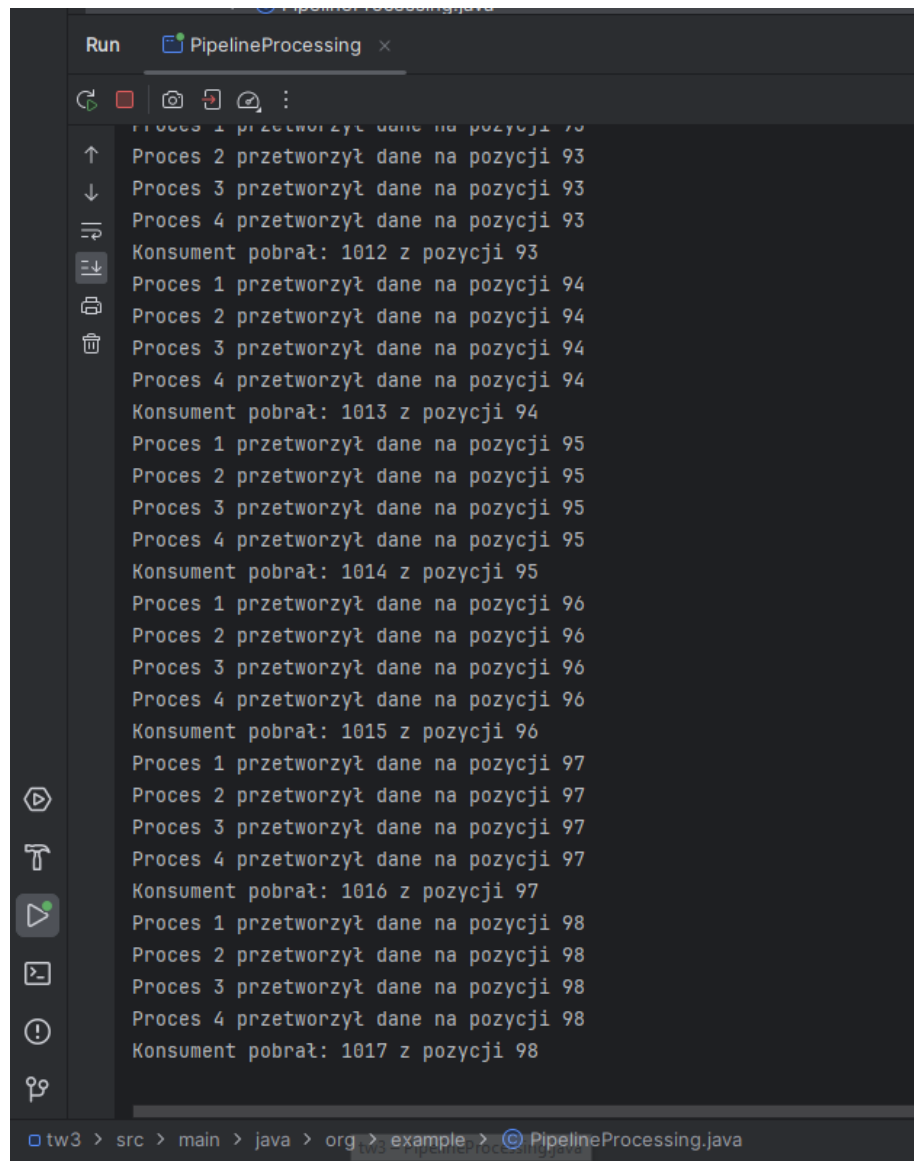
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);

        producer.start();

        // Uruchamiamy procesy przetwarzające
        for (int i = 1; i <= 5; i++) {
            new Processor(buffer, i).start();
        }

        consumer.start();
    }
}

```



```
Run PipelineProcessing x
Proces 1 przetworzył dane na pozycji 93
Proces 2 przetworzył dane na pozycji 93
Proces 3 przetworzył dane na pozycji 93
Proces 4 przetworzył dane na pozycji 93
Konsument pobrał: 1012 z pozycji 93
Proces 1 przetworzył dane na pozycji 94
Proces 2 przetworzył dane na pozycji 94
Proces 3 przetworzył dane na pozycji 94
Proces 4 przetworzył dane na pozycji 94
Konsument pobrał: 1013 z pozycji 94
Proces 1 przetworzył dane na pozycji 95
Proces 2 przetworzył dane na pozycji 95
Proces 3 przetworzył dane na pozycji 95
Proces 4 przetworzył dane na pozycji 95
Konsument pobrał: 1014 z pozycji 95
Proces 1 przetworzył dane na pozycji 96
Proces 2 przetworzył dane na pozycji 96
Proces 3 przetworzył dane na pozycji 96
Proces 4 przetworzył dane na pozycji 96
Konsument pobrał: 1015 z pozycji 96
Proces 1 przetworzył dane na pozycji 97
Proces 2 przetworzył dane na pozycji 97
Proces 3 przetworzył dane na pozycji 97
Proces 4 przetworzył dane na pozycji 97
Konsument pobrał: 1016 z pozycji 97
Proces 1 przetworzył dane na pozycji 98
Proces 2 przetworzył dane na pozycji 98
Proces 3 przetworzył dane na pozycji 98
Proces 4 przetworzył dane na pozycji 98
Konsument pobrał: 1017 z pozycji 98
tw3 > src > main > java > org > example > PipelineProcessing.java
```

Rysunek 7: Wynik działania programu dla przetwarzania potokowego z buforem

Program realizuje przetwarzanie potokowe w buforze o rozmiarze 100, gdzie producent dodaje dane, procesy przetwarzające kolejno je modyfikują, a konsument pobiera wynik końcowy. Synchronizacja za pomocą semaforów zapewnia, że każdy proces czeka na zakończenie poprzedniego, utrzymując prawidłowy przepływ danych. Program kończy się, gdy producent zasygnalizuje brak nowych danych, a wszystkie procesy zakończą pracę.

## 4 Bibliografia

Abraham Silberschatz, Peter B. Galvin, Greg Gagne. *Operating System Concepts*. Wiley, 2020. ISBN: 978-1119800361.

Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006. ISBN: 978-0321349606.

Andrew S. Tanenbaum, Herbert Bos. *Modern Operating Systems*. Pearson, 2015. ISBN: 978-0133591620.

Oracle. *The Java Tutorials - Concurrency*. Oracle, 2023.

Maurice Herlihy, Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2020. ISBN: 978-0124159501.

A. S. Tannenbaum, M. Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson, 2007. ISBN: 978-0132392273.

William Stallings. *Operating Systems: Internals and Design Principles*. Pearson, 2018. ISBN: 978-0134670959.