Laboratorium 3 - Problem ograniczonego bufora

Piotr Karamon

28.10.2024r.

Treści zadań

Zadanie 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
   (a) dla przypadku 1 producent/1 konsument
   (b) dla przypadku n_1 producentów/n2 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) $n_1 > 1$, $n_2 > 1$

Zadanie 2

Przetwarzanie potokowe z buforem

- Bufor o rozmiarze N - wspólny dla wszystkich procesów!
- Proces A będący producentem.
- Proces Z będący konsumentem.
- Procesy B, C, ..., Y będące procesami przetwarzającymi. Kaidy proces otrzymuje dana wejściowa 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 ja w miejscu.
- Procesy działają z rożnymi 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.

Zadanie 1

Rozwiązanie przy użyciu `wait` i `notify`

Rozwiązanie wykorzystuje mechanizm monitorów do synchronizacji producentów i konsumentów, współdzielących ograniczony bufor. W metodzie `put` producent dodaje element, ale czeka (`wait()`), gdy bufor jest pełny. Metoda `get` pozwala konsumentowi pobrać element, czekając, gdy bufor jest pusty. Po każdej operacji `put` lub `get` wywołanie `notifyAll()` budzi inne wątki, zapewniając płynność działania producentów i konsumentów.

Kod klasy `BufferUsingMonitor`

```
class BufferUsingMonitor {
    private final int capacity;
    private final LinkedList<Integer> buffer = new LinkedList<>();

    public BufferUsingMonitor(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void put(int i) {
        while (buffer.size() >= capacity) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        buffer.add(i);
        notifyAll();
    }

    public synchronized int get() {
        while (buffer.isEmpty()) {
```

```
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        int element = buffer.removeFirst();
        notifyAll();
        return element;

    }

    public int size() {
        return buffer.size();
    }
}
```

Kod producenta oraz konsumenta jest bardzo prosty. Dostają oni bufor oraz liczbę iteracji które mają wykonać.

```
class Producer extends Thread {
    private final BufferUsingSemaphores buffer;
    private final int numberOfIterations;

    public Producer(BufferUsingSemaphores buffer, int numberOfIterations) {
        this.buffer = buffer;
        this.numberOfIterations = numberOfIterations;
    }


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

class Consumer extends Thread {
    private final BufferUsingSemaphores buffer;
    private final int numberOfIterations;

    public Consumer(BufferUsingSemaphores buffer, int numberOfIterations) {
        this.buffer = buffer;
        this.numberOfIterations = numberOfIterations;
    }

    public void run() {
        for (int i = 0; i < numberOfIterations; ++i) {
            System.out.println(buffer.get());
        }
    }
}
```

W celu łatwego testowania rożnych wariantów wprowadzamy metodę `run()`, która przyjmuję liczbę producentów oraz konsumentów i wykonuje eksperyment. Pojemność bufora jest stała i wynosi 10.

```
public static void run(int producers, int consumers) {
    int products = producers * 20;
    int producersIterations = products / producers;
    int consumersIterations = products / consumers;

    System.out.printf(
        "Producers = %s, Consumers = %s, Total Products = %s %n", producers, consumers,
        ↪ products);
    System.out.printf(
        "One producers produces = %s items, One consumer consumes = %s items%n",
         producersIterations, consumersIterations);


    BufferUsingMonitor buffer = new BufferUsingMonitor(10);
    List<? extends Thread> threads = Stream.concat(
        IntStream.range(0, producers).mapToObj(i -> new Producer(buffer,
        ↪ producersIterations)),
        IntStream.range(0, consumers).mapToObj(i -> new Consumer(buffer,
        ↪ consumersIterations)))
       .toList();

    threads.forEach(Thread::start);
    try {
        for (Thread t : threads) {
            t.join();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("All done");
    System.out.println("Buffer size: " + buffer.size());
    System.out.println();
}
```

W metodzie `main` testujemy nasz program dla różnych ilości producentów i konsumentów.

```
public static void main(String[] args) {
    run(1, 1);
    run(5, 2);
    run(2, 5);
    run(5, 5);
}
```

Wynik działania programu:

```
Producers = 1, Consumers = 1, Total Products = 20
One producers produces = 20 items, One consumer consumes = 20 items
All done
Buffer size: 0

Producers = 5, Consumers = 2, Total Products = 100
One producers produces = 20 items, One consumer consumes = 50 items
All done
Buffer size: 0

Producers = 2, Consumers = 5, Total Products = 40
One producers produces = 20 items, One consumer consumes = 8 items
All done
Buffer size: 0

Producers = 5, Consumers = 5, Total Products = 100
One producers produces = 20 items, One consumer consumes = 20 items
All done
Buffer size: 0
```

Wszystkie wątki poprawnie się zakończyły, nie nastąpiło zakleszczenie, ani wyścig, wszystkie elementy bufora zostały skonsumowane. Program zatem działa poprawnie.

Dodajemy teraz wywołania metody `sleep()` oraz `println()` w celu obserwacji i pomiaru czasu działania programu. Wywołania `println()` są umieszczone w metodach bufora. Natomiast wywołania `sleep()` są umieszczone w każdej iteracji producenta i konsumenta. Wątki śpią po 20ms.

Wynik wykonania zmodyfikowanego programu(z racji bardzo dużej ilości linii tylko pierwsze są pokazane w celu ukazania tendencji):

```
Producers = 1, Consumers = 1, Total Products = 20
One producers produces = 20 items, One consumer consumes = 20 items
Produced: 0
Consumed: 0
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
Produced: 6
Consumed: 6
Produced: 7
Consumed: 7
Produced: 8
Consumed: 8
Produced: 9
Consumed: 9
Produced: 10
Consumed: 10
Produced: 11
Consumed: 11
Produced: 12
Consumed: 12
Produced: 13
Consumed: 13
Produced: 14
Consumed: 14
Produced: 15
Consumed: 15
Produced: 16
Consumed: 16
Produced: 17
Consumed: 17
Produced: 18
Consumed: 18
Produced: 19
Consumed: 19
All done
Buffer size: 0
```

```
Producers = 5, Consumers = 2, Total Products = 100
One producers produces = 20 items, One consumer consumes = 50 items
Produced: 0
Produced: 0
Produced: 0
Consumed: 0
Produced: 0
Consumed: 0
Produced: 0
Produced: 1
Produced: 1
Produced: 1
Consumed: 0
Produced: 1
Consumed: 0
Produced: 1
Produced: 2
Produced: 2
Produced: 2
Consumed: 0
Produced: 2
Consumed: 1
Produced: 2
Produced: 3
Consumed: 1
...
All done
Buffer size: 0
```

```
Producers = 2, Consumers = 5, Total Products = 40
One producers produces = 20 items, One consumer consumes = 8 items
Produced: 0
Consumed: 0
Produced: 0
Consumed: 0
Produced: 1
Consumed: 1
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 2
Consumed: 2
Produced: 4
Consumed: 4
Produced: 3
Consumed: 3
Produced: 5
Consumed: 5
Produced: 4
Consumed: 4
Produced: 6
...
All done
Buffer size: 0
```

```
Producers = 5, Consumers = 5, Total Products = 100
One producers produces = 20 items, One consumer consumes = 20 items
Produced: 0
Consumed: 0
Produced: 0
Consumed: 0
Produced: 0
Produced: 0
Produced: 0
Consumed: 0
Consumed: 0
Consumed: 0
Produced: 1
Consumed: 1
Produced: 1
Consumed: 1
Produced: 1
Consumed: 1
Produced: 1
Consumed: 1
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
...
All done
Buffer size: 0
```

Jak widzimy program nadal działa poprawnie. Pomiędzy wykonaniem konsumentów oraz producentów następuje naturalny przeplot we wszystkich przypadkach. Konsumenci pobierają dane z bufora chwilę, po włożeniu ich przez producentów.

W celu analizy czasowej naszego programu skorzystamy z funkcji `runTimed()` pobiera ona liczbę produktów do przetworzenia ,ilość producentów, konsumentów oraz czas opóźnienia, osobny dla producentów i konsumentów.

```
public static void runTimed(
        int products,
        int producers,
        int producersDelay,
        int consumers,
        int consumersDelay) {
    int producersIterations = products / producers;
    int consumersIterations = products / consumers;

    System.out.println("Products = " + products);
    System.out.printf("Producers = %s, Consumers = %s, Total Products = %s %n",
    ↪  producers, consumers, products);
    System.out.printf("One producers produces = %s items, One consumer consumes = %s
    ↪  item%n",
            producersIterations, consumersIterations);

    BufferUsingMonitor buffer = new BufferUsingMonitor(10);
    List<? extends Thread> threads = Stream.concat(
                    IntStream
                            .range(0, producers)
                            .mapToObj(i -> new Producer(producersDelay, buffer,
                            ↪  producersIterations)),
                    IntStream
                            .range(0, consumers)
                            .mapToObj(i -> new Consumer(consumersDelay, buffer,
                            ↪  consumersIterations)))
            .toList();

    long startTime = System.currentTimeMillis();

    threads.forEach(Thread::start);
    try {
        for (Thread t : threads) {
            t.join();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    long endTime = System.currentTimeMillis();
    System.out.println("Time taken: " + (endTime - startTime) + "ms");
    System.out.println();
}
```

Uruchamiamy funkcję dla następujących przypadków:

```
int delay = 3;

System.out.println("Producer is as fast as consumer");

runTimed(1000, 1, delay, 1, delay);
runTimed(1000, 5, delay, 2, delay);
```

```
runTimed(1000, 2, delay, 5, delay);
runTimed(1000, 5, delay, 5, delay);

System.out.println("Producer is 3x slower than consumer");

runTimed(1000, 1, delay * 3, 1, delay);
runTimed(1000, 5, delay * 3, 2, delay);
runTimed(1000, 2, delay * 3, 5, delay);
runTimed(1000, 5, delay * 3, 5, delay);
```

Wynik programu:

```
Producer is as fast as consumer
Products = 1000
Producers = 1, Consumers = 1, Total Products = 1000
One producers produces = 1000 items, One consumer consumes = 1000 item
Time taken: 3619ms

Products = 1000
Producers = 5, Consumers = 2, Total Products = 1000
One producers produces = 200 items, One consumer consumes = 500 item
Time taken: 1891ms

Products = 1000
Producers = 2, Consumers = 5, Total Products = 1000
One producers produces = 500 items, One consumer consumes = 200 item
Time taken: 1859ms

Products = 1000
Producers = 5, Consumers = 5, Total Products = 1000
One producers produces = 200 items, One consumer consumes = 200 item
Time taken: 801ms
```

```
Producer is 3x slower than consumer
Products = 1000
Producers = 1, Consumers = 1, Total Products = 1000
One producers produces = 1000 items, One consumer consumes = 1000 item
Time taken: 9646ms

Products = 1000
Producers = 5, Consumers = 2, Total Products = 1000
One producers produces = 200 items, One consumer consumes = 500 item
Time taken: 1957ms

Products = 1000
Producers = 2, Consumers = 5, Total Products = 1000
One producers produces = 500 items, One consumer consumes = 200 item
Time taken: 4813ms

Products = 1000
Producers = 5, Consumers = 5, Total Products = 1000
One producers produces = 200 items, One consumer consumes = 200 item
Time taken: 1927ms
```

W przypadku gdy producent oraz konsument pracują jednym tempem widzimy:

- 1 producent oraz 1 konsument potrzebują 3619ms by przetworzyć dane
- 2 producentów oraz 5 konsumentów potrzebują 1859ms by przetworzyć dane, jest to bardzo bliskie $\frac{3619}{2}$, pomimo tego że mamy do dyspozycji aż 7 wątków, to przyśpieszenie jest zaledwie podwójne.
- W przypadku 5 producentów oraz 2 konsumentów mamy prawie identyczne wyniki jak wyżej.
- Przyśpieszenie bliskie 5x uzyskujemy dopiero w przypadku 5 producentów oraz 5 konsumentów.
- Oznacza to, że ilość producentów oraz konsumentów powinna być starannie dobrana, bo w przeciwnym wypadku, niektóre wątki stają się tak naprawdę nadmiarowe. Zamiast 5 producentów i 2 konsumentów moglibyśmy użyć 2 producentów w celu uzyskania tej samej prędkości.

W przypadku gdy producent jest 3x wolniejszy niż konsument:

- W poprzednim przypadku czas dla przypadków 5/2 i 2/5 był identyczny. W tym natomiast widać bardzo znaczną różnicę. Czas produkcji jest o wiele wyższy od czasu

konsumpcji, co oznacza, że potrzebujemy więcej producentów w celu zapewnienia szybkości systemu.

Wnioski: System z wykorzystaniem metod `wait()` i `notify()` skutecznie synchronizuje producentów i konsumentów, umożliwiając bezpieczny dostęp do bufora bez ryzyka zakleszczeń czy wyścigów. Mechanizm monitorów pozwala producentom i konsumentom czekać na odpowiednie warunki (np. dostępność miejsca lub danych w buforze), co gwarantuje, że wszystkie elementy zostaną prawidłowo przetworzone. Wyniki eksperymentów pokazały, że wydajność systemu zależy od odpowiedniego doboru liczby producentów i konsumentów – zbyt duża lub zbyt mała liczba wątków może prowadzić do nadmiarowości lub niedostatecznego wykorzystania zasobów, co wpływa na szybkość przetwarzania danych w systemie.

Rozwiązanie przy użyciu semaforów

W celu synchronizacji używamy trzech semaforów. Semafor `mutex` zapewnia wyłączny dostęp do bufora, podczas gdy semafory zliczające `empty` i `full` kontrolują dostępne miejsca i liczbę elementów w buforze. Metoda `put(int i)` blokuje producenta, gdy bufor jest pełny, uzyskuje dostęp do bufora, dodaje element i zwiększa semafor `full`. Z kolei metoda `get()` czeka na dostępność elementów, usuwa je z bufora i zwiększa semafor `empty`.

Rozwiązanie implementuje w klasie `BufforUsingSemaphores`

```
class BufferUsingSemaphores {
    private final int capacity;
    private final LinkedList<Integer> buffer = new LinkedList<>();
    private final BinarySemaphore mutex = new BinarySemaphore();
    private final CountingSemaphore empty;
    private final CountingSemaphore full;

    public BufferUsingSemaphores(int capacity) {
        this.capacity = capacity;
        empty = new CountingSemaphore(capacity);
        full = new CountingSemaphore(0);
    }

    public void put(int i) {
        empty.P();
        mutex.P();

        buffer.add(i);

        mutex.V();
        full.V();

    }

    public int get() {
        full.P();
        mutex.P();
```

```
        int result = buffer.removeFirst();

        mutex.V();
        empty.V();
        return result;
    }

    public int size() {
        return buffer.size();
    }
}
```

Zmieniamy klasy `Producer` i `Consumer` tak by korzystały właśnie z tego bufora. Testujemy nasz bufor dokładnie dla tych samych przypadków co poprzednio.

Wynik:

```
Producers = 1, Consumers = 1, Total Products = 20
One producers produces = 20 items, One consumer consumes = 20 item
All done
Buffer size: 0

Producers = 5, Consumers = 2, Total Products = 100
One producers produces = 20 items, One consumer consumes = 50 item
All done
Buffer size: 0

Producers = 2, Consumers = 5, Total Products = 40
One producers produces = 20 items, One consumer consumes = 8 item
All done
Buffer size: 0

Producers = 5, Consumers = 5, Total Products = 100
One producers produces = 20 items, One consumer consumes = 20 item
All done
Buffer size: 0
```

Jak widać zachowanie programu jest poprawne po wymianie implementacji bufora.

Wnioski: Do synchronizacji producentów i konsumentów można użyć: metod `wait()` i `notify()` oraz semaforów. Oba mechanizmy umożliwiają bezpieczny, zsynchronizowany dostęp do współdzielonego bufora, eliminując zakleszczenia i wyścigi.

Zadanie 2

Przedstawiony problem jest rozwinięciem problemu producentów-konsumentów. Po szybkiej analizie problemu można dojść do następujących wniosków:

- Chcemy by producent mógł wpisywać dane do bufora, dopóki są w nim wolne miejsca. Producenta nie obchodzą wątki przetwarzające, jedynie interesuję go ile jest wolnych miejsc w buforze. Po wpisaniu do bufora powinien w jakiś sposób powiadomić wątki przetwarzające o nowym elemencie.
- Gdy producent wpiszę do bufora jeden element, to wtedy ciąg operacji dla niego jest jasno określony. Wątki przetwarzające będą po kolei przetwarzać element i zapisywać nową wartość w komórce. To oznacza, że wątki przetwarzające nie powinny wymagać posiadania mutexa na cały bufor, bo byłoby to po prostu mało efektywne. Zamiast tego powinny przekazywać sobie możliwość modyfikacji elementu.
- Wątek przetwarzający po transformacji jednego elementu nie powinien czekać na całkowite przetworzenie tego elementu, powinien on móc przejść do elementu następnego.

Idea rozwiązania:

- `empty` jest semaforem licznikowym i zlicza on ilość wolnych miejsc, jest opuszczany przez producenta, a podnoszony przez konsumenta
- `mutex` jest semaforem binarnym, który jest używany przez producenta oraz konsumenta by nie nastąpił wyścig przy zmienianiu rozmiaru bufora (zmiana znaczników `head` i `tail`).
- `transformersIndexes` przechowuje informacje o tym który element powinien zostać przetworzony przez odpowiadający wątek przetwarzający.
- Każdy wątek przetwarzający oraz konsument mają odpowiadający semafor licznikowy. Semafory początkowo mają wartość zero. Producent gdy dodaje element podnosi semafor pierwszego wątku przetwarzającego. To pozwala wątkowi przetwarzającemu na dokonanie transformacji na jednym elemencie. Gdy tego dokona podnoszony jest semafor kolejnego wątku w potoku.
- Na końcu potoku dane zbiera konsument, podnosząc semafor `empty` informuje producenta o zwolnionym miejscu. Konsument w przedstawionym rozwiązaniu tworzy listę zebranych elementów weryfikacji wyniku.

Kod bufora:

```
class Buffer {
    private final int capacity;
    private final BinarySemaphore mutex;
    private final CountingSemaphore empty;
    private final List<CountingSemaphore> transformers;
```

```
    private final int[] transformersIndexes;

    private final int[] buffer;
    private int bufferHead = 0;
    private int bufferTail = 0;

    public Buffer(int capacity, int transformersCount) {
        this.capacity = capacity;
        buffer = new int[capacity];
        mutex = new BinarySemaphore();
        empty = new CountingSemaphore(capacity);
        transformers = IntStream
            .range(0, transformersCount + 1)
            .mapToObj(i -> new CountingSemaphore(0))
            .toList();
        transformersIndexes = new int[transformersCount];
    }

    public void add(int x) {
        empty.P();
        mutex.P();

        buffer[bufferHead] = x;
        bufferHead = (bufferHead + 1) % capacity;

        mutex.V();
        transformers.getFirst().V();
    }

    public void transform(int transformerIndex, Function<Integer, Integer>
    ↪   transformation) {
        transformers.get(transformerIndex).P();

        int elementIndex = transformersIndexes[transformerIndex];
        buffer[elementIndex] = transformation.apply(buffer[elementIndex]);
        transformersIndexes[transformerIndex] = (elementIndex + 1) % capacity;

        transformers.get(transformerIndex + 1).V();
    }

    public int get() {
        transformers.getLast().P();
        mutex.P();

        int result = buffer[bufferTail];
        bufferTail = (bufferTail + 1) % capacity;

        mutex.V();
        empty.V();
        return result;
    }
```

```
}
```

Kod producenta:

```
class Producer implements Runnable {
    private final Buffer buffer;

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

    @Override
    public void run() {
        for (int i = 0; i < 500 i++) {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            buffer.add(i);
        }
    }
}
```

Kod wątków przetwarzających:

```
class Transformer implements Runnable {
    private final Buffer buffer;
    private final Function<Integer, Integer> transformation;
    private final int transformerIndex;
    private final int delayMs;

    public Transformer(
            int transformerIndex,
            Buffer buffer,
            Function<Integer, Integer> transformation,
            int delayMs) {
        this.buffer = buffer;
        this.transformerIndex = transformerIndex;
        this.transformation = transformation;
        this.delayMs = delayMs;
    }

    @Override
    public void run() {
        for (int i = 0; i < 500; i++) {
            try {
                Thread.sleep(delayMs);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            buffer.transform(transformerIndex, transformation);
        }
```

```
    }
}
```

Kod konsumenta:

```
class Consumer implements Runnable {
    public final List<Integer> results = new ArrayList<>();
    private final Buffer buffer;

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

    @Override
    public void run() {
        for (int i = 0; i < 500; i++) {
            try {
                Thread.sleep(15);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            int result = buffer.get();
            results.add(result);
        }
    }
}
```

Kod testujący:

```
Buffer buffer = new Buffer(100, 5);

var producer = new Producer(buffer);
var transformer1 = new Transformer(0, buffer, x -> x + 1, 3);
var transformer2 = new Transformer(1, buffer, x -> x * 2, 3);
var transformer3 = new Transformer(2, buffer, x -> x * 3, 1);
var transformer4 = new Transformer(3, buffer, x -> x - 100, 10);
var transformer5 = new Transformer(4, buffer, x -> x + 10_000, 5);
var consumer = new Consumer(buffer);

List<Integer> expectedResults = IntStream.range(0, 500).map(
        x -> (x + 1) * 2 * 3 - 100 + 10_000
).boxed().toList();

var threads = Stream.of(
        producer,
        transformer1,
        transformer2,
        transformer3,
        transformer4,
        transformer5,
        consumer
).map(Thread::new).toList();
```

```
threads.forEach(Thread::start);
try {
    for (Thread thread : threads) {
        thread.join();
    }
} catch (InterruptedException e) {
    e.printStackTrace();
}


System.out.println("All done");
System.out.println("Results are equal: " + consumer.results.equals(expectedResults));
```

Wynik:

```
All done
Results are equal: true
```

Stworzone współbieżne rozwiązanie przetwarzania potokowego z buforem dało identyczny wynik co rozwiązanie sekwencyjne.

Prędkość obróki w takim systemie zależy od najwolniejszego wątku w potoku. Jeden wątek w potoku może całkowicie zmienić prędkość naszego systemu, bo kolejne procesy muszą czekać na zakończenie jego prac. Wynika to z faktu, że operacje przetwarzające dla danego elementu muszą się wykonywać sekwencyjnie.

Wnioski: Dzięki zastosowaniu semaforów, możliwe jest przetwarzanie potokowe z ograniczonym buforem. Stworzony program pozwala na łatwą zmianę struktury potoku(ilości wątków przetwarzających oraz ich funkcje). Szybkość takiego systemu jest mocno ograniczona z racji, tego że wiele operacji musi być wykonanych w bardzo określonej kolejności.

Bibliografia

- Bill Venners: *Inside the Java Virtual Machine Chapter 20*