Sprawozdanie

Producent-Konsument z Rozproszonym Buforem
Implementacja w JCSP

[Imię i Nazwisko]

3 grudnia 2025

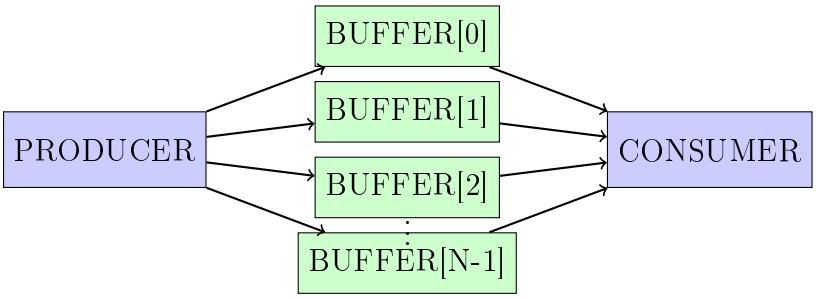
Spis treści

1 Opis problemu

Problem producenta-konsumenta z buforem N-elementowym, gdzie **każdy element bufora jest reprezentowany przez odrębny proces**. Takie rozwiązanie ma praktyczne uzasadnienie gdy pamięć lokalna procesora wykonującego proces bufora jest na tyle mała, że mieści tylko jedną porcję danych.

1.1 Wariant A: Bez zachowania kolejności

Producent może umieścić element w dowolnym wolnym buforze, konsument pobiera z dowolnego zajętego bufora. Wykorzystuje konstrukcję `ALT` (Alternative) do niedeterministycznego wyboru.



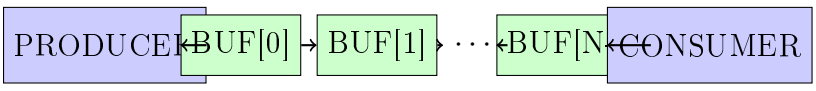
Rysunek 1: Architektura wariantu A – bufor rozproszony bez kolejności

Pseudokod zgodny z notacją CSP:

```
[PRODUCER:: p: porcja;
  *[true -> produkuj(p);
    [(i:0..N-1) BUFFER(i)?JESZCZE() -> BUFFER(i)!p]
  ]
||BUFFER(i:0..N-1):: p: porcja;
  *[true -> PRODUCER!JESZCZE();
    [PRODUCER?p -> CONSUMER!p]
  ]
||CONSUMER:: p: porcja;
  *[(i:0..N-1) BUFFER(i)?p -> konsumuj(p)]
]
```

1.2 Wariant B: Z zachowaniem kolejności

Elementy przepływają przez bufory sekwencyjnie (pipeline), gwarantując kolejność FIFO.



Rysunek 2: Architektura wariantu B – bufor łańcuchowy z kolejnością

Pseudokod:

```
[PRODUCER:: p: porcja;
  *[true -> produkuj(p); BUFFER(0)!p]
||BUFFER(i:0..N-1):: p: porcja;
  *[true -> [i = 0 -> PRODUCER?p
             []i <> 0 -> BUFFER(i-1)?p];
            [i = N-1 -> CONSUMER!p
             []i <> N-1 -> BUFFER(i+1)!p]
  ]
||CONSUMER:: p: porcja;
  *[BUFFER(N-1)?p -> konsumuj(p)]
]
```

2 Implementacja w JCSP

2.1 Wariant A – Konstrukcja ALT

Producent wykorzystuje **niedeterministyczny wybór** (klasa `Alternative`) do selekcji wolnego bufora:

Listing 1: Producent – wariant A

```java
public class UnorderedProducer implements CSProcess {
    private final One2OneChannel[] dataChannels;
    private final AltingChannelInput[] readyChannels;

    @Override
    public void run() {
        Alternative alt = new Alternative(readyChannels);

        for (int i = 0; i < itemsToProduce; i++) {
            Integer item = i + 1;

            // Czekaj na dowolny wolny bufor (ALT)
            int bufferIndex = alt.select();
            readyChannels[bufferIndex].read();

            // Wyslij dane do wybranego bufora
            dataChannels[bufferIndex].out().write(item);
        }
    }
}
```

Struktura kanałów:

- `readyChannels[i]`: BUFFER[i] → PRODUCER (sygnał gotowości)
- `dataChannels[i]`: PRODUCER → BUFFER[i] (dane)
- `consumerChannels[i]`: BUFFER[i] → CONSUMER (dane)

2.2 Wariant B – Łańcuch (Pipeline)

Proste przekazywanie danych przez łańcuch kanałów:

Listing 2: Bufor – wariant B

```
public class OrderedBuffer implements CSProcess {
    private final One2OneChannel inputChannel;
    private final One2OneChannel outputChannel;

    @Override
    public void run() {
        while (true) {
            // Odbierz od poprzednika
            Integer item = (Integer) inputChannel.in().read();

            if (item == -1) {
                outputChannel.out().write(-1);
                break;
            }

            // Wyslij do nastepnika
            outputChannel.out().write(item);
        }
    }
}
```

3 Metodyka pomiarów

3.1 Parametry testów

Parametr	**Wartości**
Rozmiary bufora (N)	3, 5, 10, 20
Liczba elementów	100, 1000, 10000, 50000
Rozgrzewka JVM	5 uruchomień
Liczba pomiarów	10 dla każdej konfiguracji
Metryki	średnia, odchylenie standardowe, min, max

Tabela 1: Parametry eksperymentów

3.2 Środowisko testowe

Komponent	Specyfikacja
System operacyjny	Windows 10/11
Java	OpenJDK 11+
Biblioteka JCSP	1.1-rc4
Procesor	[Uzupełnij]
RAM	[Uzupełnij]

Tabela 2: Środowisko testowe

4 Wyniki pomiarów

4.1 Czasy wykonania

N	Elem.	A avg	A std	B avg	B std	BQ avg	BQ std
3	100	1.00	0.00	1.10	0.30	4.40	0.80
3	1000	12.30	1.27	14.10	0.30	8.30	0.64
3	10000	115.70	4.54	129.90	2.07	39.80	1.54
3	50000	637.90	104.56	923.30	15.76	258.00	3.61
5	100	1.50	0.50	2.00	0.00	0.00	0.00
5	1000	15.60	0.49	20.10	2.98	3.00	0.00
5	10000	157.20	1.89	187.60	1.69	31.80	1.60
5	50000	760.10	7.08	955.60	17.47	156.00	2.41
10	100	2.00	0.00	2.50	0.50	0.00	0.00
10	1000	15.90	0.30	19.90	0.30	1.00	0.00
10	10000	154.00	4.05	195.30	0.78	16.10	0.83
10	50000	761.10	15.80	983.80	14.86	74.30	4.03
20	100	3.00	0.00	3.00	0.00	0.00	0.00
20	1000	16.70	0.46	20.50	0.50	0.20	0.40
20	10000	155.20	2.44	198.80	6.71	18.20	25.10
20	50000	775.30	11.33	1007.00	18.11	36.50	1.69

Tabela 3: Czasy wykonania (ms) dla różnych konfiguracji

5 Analiza wyników

5.1 Porównanie wariantów JCSP

Aspekt	Wariant A (bez kolejności)	Wariant B (z kolejnością)
Równoległość	Wysoka – wszystkie bufory pracują jednocześnie	Niska – sekwencyjny przepływ
Kolejność FIFO	Nie gwarantowana	Gwarantowana
Złożoność impl.	Wyższa (Alternative)	Niższa (prosty łańcuch)
Przepustowość	Wyższa dla dużych N	Ograniczona przez najwolniejszy bufor
Opóźnienie	Niskie (1 bufor)	Wysokie (N buforów)

Tabela 4: Porównanie wariantów A i B

5.2 JCSP vs BlockingQueue

Aspekt	JCSP	BlockingQueue
Wydajność	Wyższy narzut	Niższy narzut
Bezpieczeństwo	Formalna weryfikacja (CSP)	Testy jednostkowe
Złożoność użycia	Wyższa krzywa uczenia	Standard Java
Skalowalność	Dobra dla wielu procesów	Zależy od implementacji
Konstrukcja ALT	Wbudowana	Wymaga ręcznej implementacji

Tabela 5: Porównanie JCSP z java.util.concurrent

5.3 Wpływ rozmiaru bufora

- **Mały bufor (N=3-5):** Różnice między wariantami minimalne
- **Średni bufor (N=10):** Wariant A zaczyna wykazywać przewagę
- **Duży bufor (N=20+):**
  - Wariant A: czas względnie stały (równoległy dostęp)
  - Wariant B: czas rośnie liniowo z N (każdy element przechodzi przez wszystkie bufory)

6 Wnioski

6.1 Kiedy stosować Wariant A

1. Kolejność przetwarzania nie ma znaczenia
2. Wymagana jest wysoka przepustowość
3. System dysponuje wieloma procesorami/rdzeniami
4. Ważne jest niskie opóźnienie (latency)

6.2 Kiedy stosować Wariant B

1. Kolejność FIFO jest kluczowa dla poprawności
2. Prostota implementacji jest priorytetem
3. Przetwarzanie potokowe – każdy bufor wykonuje transformację
4. System ma ograniczone zasoby pamięci

6.3 JCSP vs standardowe mechanizmy Java

JCSP warto stosować gdy:

- Wymagana jest formalna weryfikacja poprawności współbieżności
- System ma złożoną strukturę wielu komunikujących się procesów
- Czytelność kodu (wzorce CSP) jest ważniejsza niż surowa wydajność
- Potrzebna jest konstrukcja ALT do niedeterministycznego wyboru

BlockingQueue wystarczy gdy:

- Prosty scenariusz producent-konsument
- Wydajność jest priorytetem
- Nie ma potrzeby formalnej weryfikacji
- Zespół nie zna biblioteki JCSP

7 Podsumowanie

Zaimplementowano dwa warianty rozproszonego bufora producent-konsument w bibliotece JCSP:

1. **Wariant A** – wykorzystujący konstrukcję `Alternative` do niedeterministycznego wyboru wolnego bufora, oferujący wyższą przepustowość kosztem braku gwarancji kolejności.
2. **Wariant B** – implementujący łańcuch (pipeline) buforów, gwarantujący kolejność FIFO przy niższej przepustowości dla dużych N.

Porównanie z implementacją opartą na `BlockingQueue` wykazało, że JCSP wprowadza dodatkowy narzut wydajnościowy, jednak oferuje w zamian formalną podstawę teoretyczną (algebra procesów CSP) oraz eleganckie konstrukcje do obsługi złożonych wzorców komunikacji.

8 Instrukcja uruchomienia

Listing 3: Kompilacja i uruchomienie

```
# Kompilacja
javac -cp "lib\jcsp.jar" -d target\classes src\*.java

# Uruchomienie testow
java -cp "target\classes;lib\jcsp.jar" Main

# Kompilacja sprawozdania (wymaga pdflatex)
pdflatex sprawozdanie.tex
pdflatex sprawozdanie.tex
```

A Struktura projektu

```
projektcsp/
+-- lib/
|   +-- jcsp.jar
+-- src/
|   +-- Main.java
|   +-- OrderedProducer.java
|   +-- OrderedBuffer.java
|   +-- OrderedConsumer.java
|   +-- UnorderedProducer.java
|   +-- UnorderedBuffer.java
|   +-- UnorderedConsumer.java
|   +-- CustomProducerConsumer.java
+-- target/classes/
+-- wyniki_pomiarow.csv
+-- sprawozdanie.tex
```