Laboratorium 3

Problem producenta-konsumenta

Danylo Knapp

Teoria Współbieżności

Wydział Informatyki
Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie
20.10.23

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()`
   - dla przypadku 1 producent/1 konsument
   - dla przypadku `n1` producentów/`n2` konsumentów (`n1 > n2`, `n1 = n2`, `n1 < n2`)
   - wprowadzić wywołanie metody `sleep()` i wykonać pomiary, obserwując zachowanie producentów/konsumentów
2. przy pomocy operacji `P()`/`V()` dla semafora:
   - `n1 = n2 = 1`
   - `n1 > 1`, `n2 > 1`

Uwaga: W implementacji nie jest dozwolone korzystanie/implementowanie własnych kolejek FIFO, należy używać tylko mechanizmu monitorów lub semaforów!

1.2 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. Każdy proces otrzymuje daną wejsciową 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: Zaimplementować rozwiązanie przetwarzania potokowego (Przykładowe założenia: bufor rozmiaru 100, 1 producent, 1 konsument, 5 uszeregowanych procesow przetwarzajacych). Od czego zalezy predkosc obrobki w tym systemie? Rozwiązanie za pomocą semaforów lub monitorów (dowolnie). *Zrobić sprawozdanie z przetwarzania potokowego.*

2 Rozwiązanie

Przed aktualnym rozpoczęciem rozwiązania warto najpierw przypomnieć, na czym dokładnie polega problem producentów-konsumerów (Producer-consumer problem).

Problem producenta i konsumenta - klasyczny informatyczny problem synchronizacji. W problemie występują dwa rodzaje procesów: producent i konsument, którzy dzielą wspólny zasób – bufor – dla produkowanych (i konsumowanych) jednostek. Zadaniem producenta jest wytworzenie produktu, umieszczenie go w buforze i rozpoczęcie pracy od nowa. W tym samym czasie konsument ma pobrać produkt z bufora. Problemem jest taka synchronizacja procesów, żeby producent nie dodawał nowych jednostek gdy bufor jest pełny, a konsument nie pobierał gdy bufor jest pusty.

Struktura rozwiązań poszczególnych zadań wygląda następująco:

```
src/main/java/pl/edu/agh/tw/knapp
    simplebuff
    pipeline
    Buffer.java
    RandomSleeper.java
    WorkerThread.java
```

gdzie `simplebuff` - katalog zawierający implementację pierwszego zadania, `pipeline` - katalog zawierający implementację drugiego zadania (przetwarzanie potokowe).

Implementacja pozostałych klas:

`Buffer:`

Interfejs bufora.

```java
// Buffer.java

package pl.edu.agh.tw.knapp;

import java.util.Optional;

public interface Buffer<T> {
    boolean put(T val);
    Optional<T> get();
}
```

- Metoda `put` zwraca `true`, jeżeli wartość została poprawnie dodana
- Metoda `get` zwraca `Optional`, który jest pusty jeżeli wystąpił jakiś błąd

`RandomSleeper:`

Klasa służąca do uśpienia wątku na pewien czas, losowany z przedziału `[delayMinMs, delayMaxMs)`.

```java
// RandomSleeper.java

package pl.edu.agh.tw.knapp;

import java.util.Random;

public class RandomSleeper {
    private final Random delayRandom = new Random();
    private final long delayMinMs;
    private final long delayMaxMs;

    public RandomSleeper(long delayMinMs, long delayMaxMs) {
        this.delayMinMs = delayMinMs;
        this.delayMaxMs = delayMaxMs;
    }

    public void sleep() throws InterruptedException {
        if (delayMinMs == 0 && delayMaxMs == 0)
```

```
            return;
        var delay = delayRandom.nextLong(delayMinMs, delayMaxMs);
        Thread.sleep(delay);
    }
}
```

`WorkerThread`:

Klasa nadrzędna dla producentów i konsumentów. Zawiera referencję na bufor, udostępnia funkcję umożliwiającą randomowe uśpienie wątku oraz funkcję służącą do logowania.

```
// WorkerThread.java

package pl.edu.agh.tw.knapp;

public class WorkerThread<T> extends Thread {
    protected final Buffer<T> buff;
    private final RandomSleeper randomSleeper;

    public WorkerThread(Buffer<T> buff, long delayMinMs, long delayMaxMs) {
        this.buff = buff;
        randomSleeper = new RandomSleeper(delayMinMs, delayMaxMs);
    }

    public WorkerThread(Buffer<T> buff) {
        this(buff, 0, 0);
    }

    protected void randomDelay() throws InterruptedException {
        randomSleeper.sleep();
    }

    protected void log(Object o) {
        System.out.printf("[%s id %s] %s\n", getClass().getSimpleName(), getId(), o);
    }
}
```

2.1 Problem ograniczonego bufora

Ze względu na to, że sprawozdanie z tej części nie jest wymagane, poniżej ogólnie opiszę całą implementację. Analizy niektórych wyników dokonam w rozdziale *Wyniki*.

Od razu warto dodać, że uwagę

> W implementacji nie jest dozwolone korzystanie/implementowanie własnych kolejek FIFO, należy używać tylko mechanizmu monitorów lub semaforów!

rozumiem w taki sposób, że zabronione jest korzystanie ze wszystkich możliwych kolejek (w tym FIFO) i należy używać tylko i wyłącznie mechanizmu semaforów lub monitorów, z czego można wywnioskować że:

- Nie wolno korzystać ani z żadnych list ani tablic w celu przechowywania danych wyprodukowanych przez producenta
- Problem musi zostać zaimplementowany wyłącznie w oparciu o mechanizmy synchronizacji

z czego wynika, że bufor ma być o rozmiarze `1`. W tym przypadku rzeczywiście nie potrzebujemy żadnych kolejek i całe rozwiązanie może zostać zaimplementowane jedynie w oparciu o mechanizmy synchronizacji.

Struktura rozwiązania wygląda następująco:

```
src/main/java/pl/edu/agh/tw/knapp/simplebuff
    ConditionMonitor.java
    Consumer.java
    Main.java
    MonitorBuffer.java
    Producer.java
    SemaphoreBuffer.java
```

Krótki opis poszczególnych klas:

- `ConditionMonitor`: klasa przypominająca `std::condition_variable` z języka C++, lecz jest zaimplementowana w oparciu o Javowe monitory. Jest wykorzystywana przez `MonitorBuffer`;
- `Consumer`: implementacja konsumenta. *Próbuje* pobrać 100 razy dane z buforu. W przypadku powodzenia wypisuje pobraną wartość, w przypadku niepowodzenia wypisuje odpowiedni komunikat. Uwaga: niepowodzenie może wystąpić jeżeli żaden producent nie zapisał do bufora jakiejś wartości w ciągu określonego czasu. Wtedy konsument stwierdza, że "transmisja" jest zakończona i kończy swoje działanie. Ten mechanizm jest niezbędny dla niektórych przypadków, np. gdy liczba konsumentów jest większa od liczby producentów: w celu uniknięcia zawieszenia programu z powodu oczekiwania nowych danych nadanych przez producenta, musimy skorzystać z wyżej opisanego mechanizmu;
- `Main`: klasa zawierająca niektóre "testy", bardziej szczegółowo jest opisana poniżej;
- `MonitorBuffer`: implementacja bufora w oparciu o Javowe monitory;
- `Producer`: producent. *Próbuje* zapisać 100 różnych wartości do buforu. W przypadku niepowodzenia wypisuje odpowiedni komunikat. Niepowodzenie może zostać spowodowane tym, że już żaden konsument nie próbuje odczytać wartości z buforu. Ten mechanizm został zaimplementowany z przyczyn opisanych podczas omówienia konsumenta, tylko tym razem liczba producentów może być większa od liczby konsumentów;
- `SemaphoreBuffer`: implementacja bufora w oparciu o semafory;

Klasa `Main` zawiera następujące testy dla obu implementacji bufora:

- 1 producent 1 konsument
- 1 producent 10 konsumentów
- 10 producentów 1 konsument
- 100 producentów 100 konsumentów

Program również został przetestowany dla 5000 producentów i 5000 konsumentów, ale ten przypadek nie został zawarty w implementacji z tego powodu, iż wymaga dość dużej ilości pamięci RAM.

Uwaga: omówienie wyników znajduje się w odpowiednim rozdziale.

To zadanie może zostać uruchomione korzystając z polecenia

```
./gradlew run -Pmain=pl.edu.agh.tw.knapp.simplebuff.Main
```

2.2 Przetwarzanie potokowe z buforem

Przetwarzanie potokowe z buforem wymaga zaimplementowania następujących mechanizmów:

1. Bufor o określonym rozmiarze - jest wspólny dla wszystkich wątków, a więc musi być *thread-safe*
2. Producent - produkuje dane i zamieszcza je w buforze
3. Konsument - pobiera dane z bufora i przekazuje je do przetwarzania
4. *Pipe* (rura) - przetwarza otrzymane na wejściu dane i przekazuje je dalej: albo do następnej rury, albo do callbacku

Schemat przetwarzania potokowego wygląda w następujący sposób:

```
Producer --> Consumer
                |
              Pipe 1
                |
              Pipe 2
                |
               ...
                |
              Pipe N --> Callback
```

W trakcie wykonania zadania, struktura rozwiązania przybrała następującą postać:

```
src/main/java/pl/edu/agh/tw/knapp/pipeline
    Box.java
    Consumer.java
    Main.java
    PipeAction.java
    Pipe.java
    Producer.java
    SemaphoreBuffer.java
    ThreadedPipe.java
```

Już wiemy, na czym polega działanie bufora, producenta i konsumenta, a więc skupmy się na tym, czym jest i jak działa **pipe**.

`Pipe` jest klasą abstrakcyjną, zawierającą metody umożliwiające sterowanie przetwarzaniem danych. Jest zaimplementowana następująco:

```java
// Pipe.java

package pl.edu.agh.tw.knapp.pipeline;

public abstract class Pipe<T> {
    private Pipe<T> next;
    private PipeAction<T> action;

    public Pipe<T> then(Pipe<T> next) {
```

```
        this.next = next;
        action = null;
        return next;
    }

    public void action(PipeAction<T> action) {
        this.action = action;
        next = null;
    }

    public abstract void close();

    public void closeAll() {
        close();

        if (next != null) {
            next.closeAll();
        }
    }

    public void submitAsync(T value) {
        throw new RuntimeException("Not implemented");
    }

    public void submit(T value) {
        submitValue(value, Pipe::submit);
    }

    protected void submitValue(T value, NextAction<T> nextAction) {
        var result = onSubmit(value);

        if (next != null) {
            nextAction.onNext(next, result);
        } else if (action != null) {
            action.onAction(result);
        }
    }

    protected abstract T onSubmit(T value);

    protected interface NextAction<T> {
        void onNext(Pipe<T> next, T value);
    }
}
```

- Metoda `then` umożliwia tworzenie tzw. pipline'u: "sklejania" kilku pipe'ów, tzn. na wejście podanego jako argument pipe'u zostanie przekazany wynik (wyjście) danego pipe'u (jak to dokładnie wygląda zostanie pokazane poniżej)

- Metoda `action` jest metodą końcową (terminalną), tzn. jako argument przyjmuje callback, do którego zostanie przekazana wartość po przetworzeniu przez daną rurę (`Pipe N` na schemacie)
- Metoda abstrakcyjna `close` zamyka daną rurę - rura nie będzie już w stanie przetwarzać nowych porcji danych
- Metoda `closeAll` zamyka daną rurę oraz wszystkie inne rury niżej w hierarchii
- Metoda `submit` służy do przekazania danych na wejście danej rury. Jest to funkcja blokująca: blokuje dany wątek dopóki dane nie zostaną przetworzone przez daną rurę oraz wszystkie pozostałe rury niżej w hierarchii
- Metoda `submitAsync` działa podobnie do `submit`, lecz w sposób asynchroniczny, tzn. dostaje dane na wejściu i umieszcza je w kolejce "do przetwarzania". Po przetworzeniu, przekazuje je w sposób asynchroniczny do kolejnej rury lub do callbacku. Nie jest zaimplementowana

`ThreadedPipe` jest klasą dziedziczącą po `Pipe` i implementującą przetwarzanie asynchroniczne korzystając z `java.util.concurrent.ExecutorService`, a mianowicie jest użyta metoda `Executors.newSingleThreadExecutor()`. `ExecutorService` jest użyty z powodu chęci podniesienia czytelności kodu, bo choć implementowanie własnego `ThreadPool` nie jest specjalnie skomplikowane, lecz w sposób oczywisty kompikuje analizę rozwiązania ze względu na ilość kodu.

Oczywiście, możemy pozbyć się dodatkowej kolejki (tzn. zrezygnować z `ExecutorService`) korzystając z mechanizmu podobnego do tego, co został użyty do implementacji klasy `SemaphoreBuffer` / `MonitorBuffer` z zadania 1, ale w takim razie nawet metoda `submitAsync` może okazać się częściowo blokującą: szybkość obróbki danych zostanie uzależniona również od zajętości (`i + 1`)-ej rury, która z kolei zostanie uzależniona od (`i + 2`)-iej rury itd. Z czego wynika, że szybkość obróbki danych w rurze o indeksie `i` zostanie uzależniona od rur o indeksach `i + 1`, `i + 2`, ..., `n`. Ze względów wydajnościowych chcielibyśmy takiej sytuacji uniknąć, a więc musimy dołożyć kolejną kolejkę, a w naszym przypadku - skorzystać z `ExecutorService`. Przy takiej implementacji, szybkość obróbki nowych danych w `i`-tej rurze zostanie uzależniona tylko od szybkości nadchodzenia tych danych, od liczby danych już oczekujących na obróbkę, oraz od własnej szybkości przetwarzania jednej porcji danych przez daną rurę.

```java
// ThreadedPipe.java

package pl.edu.agh.tw.knapp.pipeline;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public abstract class ThreadedPipe<T> extends Pipe<T> {
    private final ExecutorService executorService = Executors.newSingleThreadExecutor();

    public ThreadedPipe() {
        // empty
    }

    @Override
    public void close() {
        executorService.shutdown();
    }
```

```
    @Override
    public void submitAsync(T value) {
        executorService.submit(() -> submitValue(value, Pipe::submitAsync));
    }

    @Override
    public void submit(T value) {
        var future = executorService.submit(() -> super.submit(value));

        try {
            future.get();
        } catch (InterruptedException | ExecutionException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Klasa `Box` została zaimplementowana ze względu na następującą uwagę:

> Procesy przetwarzają dane w miejscu, po czym przechodzą do kolejnej komórki bufora i znowu przetwarzają ją w miejscu

interpretuję to tak, że żaden proces (wątek, pipe) nie powinien tworzyć kopii danych otrzymanych na wejściu. Zamiast tego, musi operować na już istniejących danych (mówiąc w kontekście Javy: referencja na obiekt wczytany przez konsumenta musi pozostać taka sama przez cały czas przetwarzania tego obiektu przez poszczególne pipe'y).

Biorąc pod uwagę to, iż przetwarzać będziemy `Integer`, musimy zadbać o zachowanie w/w warunku.

```
// Box.java

package pl.edu.agh.tw.knapp.pipeline;

public class Box<T> {
    private T value;

    public Box() {
        // empty
    }

    public Box(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
```

```
        this.value = value;
    }

    @Override
    public String toString() {
        return "Box {" + value + '}';
    }
}
```

- Producent tworzy instancję klasy `Box` i zapisuje ją do bufora
- Konsument odczytuje i przekazuje do pipe'ów
- Poszczególne pipe'y modyfikują wartość zawartą w tym obiekcie zamiast tworzenia nowego obiektu

Klasa `Main` zawiera przykładowe korzystanie ze stworzonego mechanizmu:

```
package pl.edu.agh.tw.knapp.pipeline;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        var buffer = new SemaphoreBuffer<Box<Integer>>(100);
        var producer = new Producer(buffer, 0, 50);
        var consumer = new Consumer(buffer, 0, 100);

        PipeAction<Box<Integer>> resultAction = result -> {
            System.out.println(result);

            if (result.getValue() == 105) {
                consumer.pipe().closeAll();
            }
        };

        consumer.pipe()
                .then(newAddPipe(1))
                .then(newAddPipe(1))
                .then(newAddPipe(1))
                .then(newAddPipe(1))
                .then(newAddPipe(2))
                .action(resultAction);

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

        consumer.join();
        producer.join();
    }

    /**
     * A simple wrapper for `Consumer#newPipe` that returns
```

```
     * an additive pipe
     * @return A newly created pipe by the `Consumer` class.
     */
    private static Pipe<Box<Integer>> newAddPipe(int value) {
        return Consumer.newPipe(v -> {
            v.setValue(v.getValue() + value);
            return v;
        }, 0, 150);
    }

    /**
     * A simple wrapper for `Consumer#newPipe` that returns
     * a multiplicative pipe
     * @return A newly created pipe by the `Consumer` class.
     */
    private static Pipe<Box<Integer>> newMulPipe(int value) {
        return Consumer.newPipe(v -> {
            v.setValue(v.getValue() * value);
            return v;
        }, 0, 150);
    }
}
```

Jak można łatwo zauważyć, w tym przykładzie korzystając z `5` pipe'ów zwiększamy wartość wyprodukowaną przez producenta o `6`.

Widać również, iż producent produkuje nowe dane co *0-50ms*, konsument pobiera dane co *0-100ms*, a szybkość obróbki danych przez poszczególne pipe'y wynosi *0-150ms*.

Warto dodać, że aby po wykonaniu wszystkich operacji aplikacja się nie zawiesiła, musimy zadbać o zamknięcie wszystkich pipe'ów, co jest pokazane wewnątrz callback'u `resultAction`.

Oczywiście, możemy tworzyć bardziej zaawansowane pipeline'y, np. możemy skorzystać z rury mnożącej, która jest tworzona przez metodę `newMulPipe`.

> Uwaga: wyniki zostały omówione w rozdziale *Wyniki*
>
> To zadanie może zostać uruchomione korzystając z polecenia
>
> `./gradlew run -Pmain=pl.edu.agh.tw.knapp.pipeline.Main`

3 Wyniki

W tym rozdziale są umieszczone wyniki działania poszczególnych zadań.

3.1 Problem ograniczonego bufora

Poniżej w skrócie zostaną omówione niektóre wyniki.

3.1.1 1 producent 10 konsumentów

Na wyjściu dostajemy następujący wynik:

```
******* producers = 1, consumers = 10, buffer: SemaphoreBuffer *******
[Consumer id 22] 0
[Consumer id 22] 1
[Consumer id 31] 2
[Consumer id 30] 3
[Consumer id 23] 4
[Consumer id 27] 5
[Consumer id 26] 6
[...]
[Consumer id 27] 94
[Consumer id 25] 95
[Consumer id 24] 96
[Consumer id 23] 97
[Consumer id 26] 98
[Consumer id 30] 99
[Consumer id 28] Buffer#get: end reached
[Consumer id 31] Buffer#get: end reached
[Consumer id 29] Buffer#get: end reached
[Consumer id 22] Buffer#get: end reached
[Consumer id 27] Buffer#get: end reached
[Consumer id 24] Buffer#get: end reached
[Consumer id 25] Buffer#get: end reached
[Consumer id 23] Buffer#get: end reached
[Consumer id 26] Buffer#get: end reached
[Consumer id 30] Buffer#get: end reached
```

Interpretacja wyniku: ze względu na to, że w tym przypadku mamy tylko 1 producenta, który wyprodukował 100 wartości od `0` do `99`, i 10 konsumentów, każdy z których próbuje odczytać 100 wartości, w sposób oczywisty dochodzimy do pewnego problemu, a mianowicie dla konsumentów zabrakło danych. Właśnie z tego powodu widzimy 10 komunikatów od 10 różnych konsumentów, że bufor już jest pusty (został przekroczony czas oczekiwania na kolejną wartość).

3.1.2 10 producentów, 1 konsument

```
******* producers = 10, consumers = 1, buffer: SemaphoreBuffer *******
[Consumer id 22] 0
[Consumer id 22] 0
[Consumer id 22] 0
[Consumer id 22] 1
[...]
[Consumer id 22] 10
[Consumer id 22] 9
[Consumer id 22] 9
[Consumer id 22] 10
[Producer id 26] Buffer#put error
[Producer id 29] Buffer#put error
[Producer id 23] Buffer#put error
[Producer id 24] Buffer#put error
```

```
[Producer id 28] Buffer#put error
[Producer id 25] Buffer#put error
[Producer id 30] Buffer#put error
[Producer id 31] Buffer#put error
[Producer id 32] Buffer#put error
[Producer id 27] Buffer#put error
```

Interpretacja wyniku: otrzymane komunikaty są spowodowane tym, iż producentów jest więcej od konsumentów, a więc wyprodukowanych wartości jest więcej niż konsument jest w stanie przetworzyć.

3.2 Przetwarzanie potokowe z buforem

Po uruchomieniu zaprezentowanego przykładu na wyjściu otrzymujemy:

```
[Producer id 22] Started
[Consumer id 23] Started
Box {6}
Box {7}
Box {8}
[...]
Box {24}
Box {25}
Box {26}
[Producer id 22] Done
Box {27}
Box {28}
Box {29}
[...]
Box {73}
Box {74}
Box {75}
[Consumer id 23] Buffer#get: end reached
[Consumer id 23] Done
Box {76}
Box {77}
Box {78}
[...]
Box {103}
Box {104}
Box {105}
```

Z czego można wyciągnąć następujące wnioski:

- Producent skończył swoje działanie jako pierwszy
- Konsument skończył swoje działanie zanim wszystkie dane zostały przetworzone
- Dane wyprodukowane przez producenta (`0..99`) zostały powiększone o `6` (`6..105`)
- Aplikacja zakończyła swoje działanie po przetworzeniu i wypisaniu ostatniej obrobionej liczby (`105`)

4 Wnioski

- Problem producentów-konsumentów (producer-consumer problem, bounded-buffer problem) da się rozwiązać korzystając z różnych mechanizmów synchronizacji, m.in. semaforów i monitorów
- Przetwarzanie potokowe umożliwia niezależną wielowątkową obróbkę porcji danych, co ma pozytywny wpływ na wydajność
- Pipeline (tzw. "rurociąg") składa się z łańcucha elementów przetwarzających (procesów, wątków, koprocedur, funkcji itp.), ułożonych tak, że wyjście każdego elementu jest wejściem następnego
- Pipeline jest to łańcuch, składający się ze źródła, operacji pośrednich oraz operacji końcowej. W zaprezentowanej implementacji źródłem jest rura początkowa, tj. `Consumer#pipe`, operacje pośrednie to są poszczególne pipe'y (inkrementujący, mnożący), operacją końcową jest callback, prezentowany przez interfejs `PipeAction`
- Szybkość obróbki *nowych* danych w `i`-tej rurze jest uzależniona tylko od szybkości nadchodzenia tych danych, od liczby danych już oczekujących na obróbkę, oraz od własnej szybkości przetwarzania jednej porcji danych przez daną rurę
- Zaczynając od Java 8, w celu potokowego przetwarzania danych, możemy również korzystać z Java Stream API

5 Bibliografia

1. Materiały do laboratorium
2. Wikipedia - Problem producenta i konsumenta
3. Wikipedia - Producer-consumer problem
4. Java 17 Docs - ExecutorService
5. Baeldung - A Guide to the Java ExecutorService
6. Baeldung - The Java 8 Stream API Tutorial