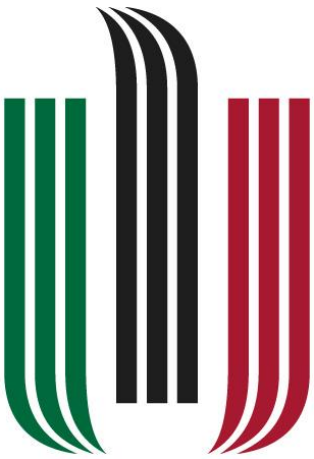
Laboratorium III

Oczekiwanie na warunek (condition wait)

Dominik Marek

21 października 2024



AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

1.Zadania

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().
    a) dla przypadku 1 producent/1 konsument
    b) dla przypadku n1 producentów/n2 konsumentów (n1>n2, n1=n2, n1<n2)
    c) 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

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 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.

2.Rozwiązania

Zadanie 1.

1. W celu zrealizowanie standardowego problemu producenta-konsumenta wykorzystuję następujące klasy:
    - *interface IBuffer* - interfejs bufora implementowany przez klasy Buffer oraz PVBuffer, wykorzystywany w drugiej części zadania. Dzięki niemu klasy Producer i Consumer mogą korzystać z buforów obu rodzajów, bez konieczności ponownej ich implementacji.
    - *class Buffer* – klasa implementująca bufor. Posiada dwie metody put() i get() wykorzystujące mechanizmy synchronized, wait(), i notifyAll() do kontrolowania dostępu do bufora, zapewniając bezpieczne działanie wielowątkowe.
    - *class Consumer i Producer* – klasy reprezentujące odpowiednio jednego konsumenta i producenta. Każda z nich w pętli wywołuję odpowiednią metodę na obiekcie bufora, aby włożyć lub pobrać element z bufora.
    - *class Main* – główna klasa programu, w której tworzymy instancję bufora i uruchamiamy program dla zadanej liczby konsumentów i producentów.

```
public interface IBuffer {

   int get(int Id);

   void put(int value, int Id);
}
```

```
class Buffer  implements IBuffer{
    private boolean is_available = false;
    private int data;

    @Override
    public  synchronized void put(int i,int producerId) {
        while (is_available) {
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }

        this.data = i;
        is_available = true;

        notifyAll();
        System.out.println("Producer number " + producerId + " produce " +
this.data);

    }

    @Override
    public synchronized int get(int consumerId) {
        while (!is_available) {
            try {
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }


        is_available = false;
        notifyAll();
        System.out.println("Consumer number " + consumerId + " consume " +
this.data);
        return this.data;
    }
}
```

```
class Producer extends Thread {
    private IBuffer _buf;
    private int id;

    public Producer(IBuffer buf, int producerId) {
        this._buf = buf;
        this.id = producerId;
    }

    public void run() {
```

```
        for (int i = 0; i < 100; ++i) {
            _buf.put(i, this.id);
            try {
                  sleep(500);
            } catch (InterruptedException e) {
              throw new RuntimeException(e);
            }


        }
    }
}
```

```
class Consumer extends Thread {
    private IBuffer _buf;
    private int id;


    public Consumer(IBuffer buf, int consumerId) {
        _buf = buf;
        this.id = consumerId;
    }


    public void run() {
        for (int i = 0; i < 100; ++i) {
            _buf.get(this.id);
            try {
                  sleep(500);
             } catch (InterruptedException e) {
               throw new RuntimeException(e);
             }


        }
    }
}
public class Main {

    public static void main(String[] args) {
        Buffer buf = new Buffer();
        int numberOfConsumer = 2;
        int numberOfProducers = 2;


        for (int i = 0; i < numberOfProducers; i++) {
            Producer producer = new Producer(buf, i);
            producer.start();
        }

        for (int i = 0; i < numberOfConsumer; i++) {
            Consumer consumer = new Consumer(buf, i);
            consumer.start();
        }

    }

}
```

2.Przebieg programu dla zadanej liczby producentów n_1 i konsumentów n_2:

1.

Dla jednego producenta i konsumenta pogram zakończy się po tym jak producent wyprodukuje 100 jednostek danych, które pobrane przez konsumenta.

2.

a) $n_1 = n_2$

Dla jednakowej liczby producentów i konsumentów system jest dobrze zbilansowany; producenci i konsumenci w miarę równomiernie dzielą zasoby. Gdy jeden producent wstawia dane do bufora, konsument może je pobrać. Gdy każdy producent wyprodukuje 100 jednostek danych, a każdy konsument skonsumuje dokładnie taką samą ilość program zakończy swoje działanie.

b) $n_1 > n_2$

W przypadku większej producentów gdy konsumenci zakończą swoją pracę (po pobraniu 100 * n_2 jednostek danych), producenci nadal będą próbowali produkować kolejne jednostki, ale będą stale czekać na opróżnienie bufora przez konsumentów, co nigdy nie nastąpi. To doprowadzi do zawieszenia programu, ponieważ producenci będą stale oczekiwać na notifyAll() bez odpowiedzi.

c) $n_1 < n_2$

W sytuacji gdy występuje przewaga konsumentów podobnie jak przypadku opisanym powyżej po zakończeniu wszystkich operacji przez producentów, tylko konsumenci będą oczekiwać na zapełnienie pustego bufora przez producentów, co już nie będzie miał miejsca. W związku z tym, pozostali konsumenci będą „zawieszeni" w oczekiwaniu na notifyAll(), którego nigdy nie otrzymają, ponieważ producenci zakończyli pracę.

3.

Dodanie metody sleep(500) do metody run() w klasach Producenta i Konsumenta po wykonaniu operacji na buforze pozwolił podczas wykonywania programu zaobserwować na przemienne wykładanie ,wyprodukowanych przez Producenta, danych do bufora oraz pobieranie ich przez konsumenta.

Zadanie 2.

W drugiej części zadania należało zrealizować problem producenta -konsumenta ale tym razem wykorzystując operacje P() i V() na semaforach. Klasa PVBuffer implementuje bufor z dostępem synchronizowanym za pomocą semaforów, umożliwiający bezpieczną współpracę między producentami i konsumentami. Semafor empty kontroluje dostęp producentów, uniemożliwiając wstawianie danych, gdy bufor jest pełny, podczas gdy semafor full pozwala konsumentom na pobieranie danych tylko wtedy, gdy są one dostępne. Metody put() i get() używają odpowiednio acquire() i release(), aby producent i konsument mogli naprzemiennie korzystać z bufora, zapewniając bezpieczeństwo danych i unikanie blokad. Dzięki temu, że klasa PVBuffer

implementuje wcześniej wspomniany interfejs IBuffer teraz w klasie Main wystarczy zmienić typu Bufora i program będzie działał.

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

public class PVBuffer implements IBuffer {

    private Semaphore empty = new Semaphore(1);
    private Semaphore full = new Semaphore(0);
    private int data;

    public PVBuffer() {}

    @Override
    public void put(int data, int producerId) {

        try {
            this.empty.acquire();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        this.data = data;
        this.full.release();
        System.out.println("Producer number " + producerId + " produce " +
this.data);

    }
   @Override
    public int get(int consumerId) {
        try {
            this.full.acquire();
        } catch (InterruptedException e) {
           Thread.currentThread().interrupt();
        }

        this.empty.release();
        System.out.println("Consumer number " + consumerId + " consume " +
this.data);
        return this.data;
    }

}
```

2.Przebieg programu:

- ***n1 = n2 = 1 (jeden producent i jeden konsument)***

  W takim przypadku kod działa tak, że jeden producent wstawia dane, a jeden konsument pobiera te dane. Program jest stabilny, ponieważ producent i konsument naprzemiennie uzyskują dostęp do bufora.

- **n1 > 1, n2 > 1 (wielu producentów i wielu konsumentów)**

W przypadku większej liczby producentów i konsumentów, semafory empty i full nadal zapewniają, że dostęp do bufora jest zsynchronizowany. Kolejni producenci i konsumenci będą czekać na możliwość uzyskania dostępu, gdy bufor będzie pełny lub pusty. Program może mieć dowolną liczbę producentów i konsumentów, ale ze względu na pojedynczy bufor, tylko jeden producent może dodać, a jeden konsument pobrać dane w danym momencie. Podobnie jak we wcześniejszej części zadania jeśli będziemy mieli równą liczbę producentów i konsumentów to program zakończy się gdy ostatni wyprodukowany przez producenta element zostanie pobrany przez ostatniego oczekujące konsumenta. W innych przypadkach podobnie jak wcześniej producenci bądź konsumenci będą czekać na opróżnienie bądź dodanie danych do bufora, które jednak nie wystąpi.

Zadanie 2.

1. Definicja przetwarzania potokowego (pipeline processing)

Przetwarzanie potokowe to technika optymalizacji obliczeń, w której zadanie jest dzielone na mniejsze etapy (procesy), a każda z tych faz działa równolegle lub w sposób współbieżny, przetwarzając różne dane w tym samym czasie. Podobnie jak w przypadku linii produkcyjnej, dane przechodzą przez kolejne etapy przetwarzania, gdzie każdy etap wykonuje swoją operację na danych i przekazuje je do kolejnej fazy. W architekturach komputerowych przetwarzanie potokowe jest stosowane w celu zminimalizowania opóźnień i zwiększenia wydajności. Procesy w przetwarzaniu potokowym działają równolegle, co pozwala na jednoczesne przetwarzanie wielu porcji danych, co skutkuje zwiększeniem przepustowości systemu.

2.Kod programu:

```
public class PipelineBuffer {
    private static class Buffer {
        private int[] productionLine;

        public Buffer(int size) {
            this.productionLine = new int[size];
            for (int i = 0; i < size; i++) {
                productionLine[i] = -1;
            }
        }

        public synchronized void startProcess(int cell, int processorId)
throws InterruptedException {
            while (productionLine[cell] != processorId - 1) {
                wait();
            }
            System.out.println("I got the cell " + cell + " process " +
processorId);
        }

        public synchronized void endProcess(int cell, int processorId) {
            productionLine[cell] = processorId;
            notifyAll();
        }

        public int getSize() {
```

```
            return productionLine.length;
        }
    }

    private static class Consumer extends Processor {

        public Consumer(Buffer buffer, int id) {
            super(buffer, id);
        }

        @Override
        public void method(int i) {
            super.method(i);
            System.out.println("Consumer have eaten cell nr " + i);
        }

    }
    private static class Producent extends Processor {

        public Producent(Buffer buffer, int id) {
            super(buffer, id);
        }

        public void method(int i) {
            super.method(i);
            System.out.println("Producent touched cell nr " + i);
        }
    }

    private static class Processor implements Runnable {
        private Buffer buffer;
        private int id;

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

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

        public void method(int i) {
            try {
                buffer.startProcess(i, id);
                Thread.sleep((int) Math.floor(Math.random() * 1000));
                buffer.endProcess(i, id);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        Buffer buffer = new Buffer(100);
        Thread producent = new Thread(new Producent(buffer, 0));
        Thread[] processors = new Thread[5];          for (int i = 1; i <=
5; i++) {
            processors[i - 1] = new Thread(new Processor(buffer, i));
```

```
        }
        Thread consumer = new Thread(new Consumer(buffer, 6));

        producent.start();
        for (int i = 1; i <= 5; i++) {
            processors[i - 1].start();
        }
        consumer.start();
    }
```

3. Opis przedstawionego rozwiązania

Kod realizuje przetwarzanie potokowe dla systemu składającego się z:

- ***1 producenta***, który inicjuje przetwarzanie danych,
- ***5 procesów przetwarzających dane***, każdy z nich działa sekwencyjnie,
- ***1 konsumenta***, który finalizuje przetwarzanie, odbierając dane na końcu potoku.

Cały proces przetwarzania odbywa się na buforze o rozmiarze 100 jednostek, które przechodzą przez wszystkie fazy przetwarzania (producent -> procesory -> konsument). Każda jednostka bufora jest przetwarzana krok po kroku przez producenta, następnie przez 5 procesorów (które są uszeregowane), a na końcu trafia do konsumenta.

Struktura kodu:

1. **Klasa Buffer:**
   - Bufor (productionLine) to tablica o rozmiarze 100, której każda komórka reprezentuje etap przetwarzania konkretnej jednostki danych.
   - **startProcess()**: Metoda odpowiedzialna za rozpoczęcie przetwarzania przez dany procesor. Używa mechanizmu monitorów (synchronized, wait(), notifyAll()), aby kontrolować dostęp do bufora. Procesor może rozpocząć przetwarzanie komórki tylko wtedy, gdy poprzedni procesor zakończył przetwarzanie tej samej komórki.
   - **endProcess()**: Po zakończeniu przetwarzania przez dany procesor, metoda aktualizuje stan przetwarzania komórki i powiadamia inne wątki (notifyAll()), że przetwarzanie się zakończyło.

2. **Processor:**

   - Klasa Processor implementuje metodę **`run()`**, która jest wymagana przez interfejs **`Runnable`.** Dzięki temu obiekty tej klasy można uruchomić jako wątki.

   - Każdy procesor (w tym producent i konsument) działa w sposób współbieżny. Wątek przetwarza każdą jednostkę (komórkę) bufora sekwencyjnie, wykonując metodę `method()` na każdej komórce.

   - **Metoda `method()`** odpowiada za:

- Wywołanie **`startProcess()`,** co oznacza, że procesor sprawdza, czy poprzedni etap przetwarzania (procesor o niższym ID) zakończył swoją pracę na danej komórce bufora.
- Symulację czasu przetwarzania za pomocą **`Thread.sleep()`,** co reprezentuje operację, którą wykonuje procesor.
- Wywołanie **`endProcess()`**, które oznacza, że procesor zakończył swoją pracę na danej komórce i może sygnalizować innym procesom (kolejnym procesorom), że mogą rozpocząć swoje przetwarzanie.

3. **Producent:**

- Producent jest specjalnym rodzajem procesora, którego zadaniem jest rozpoczęcie przetwarzania każdej komórki w buforze. Producent ma ID 0, co oznacza, że działa jako pierwszy w przetwarzaniu potokowym.
- Producent zapisuje dane w każdej komórce i sygnalizuje, że przetwarzanie tej komórki może być kontynuowane przez pierwszy procesor pośredni (procesor o ID 1).
- Po zakończeniu przetwarzania każdej komórki, producent wyświetla komunikat "Producent touched cell nr", co wskazuje, że komórka została przetworzona przez producenta i przekazana dalej w potoku.

4. **Consumer:**

- Consumer działa jako ostatni etap przetwarzania (ID 6) i ma za zadanie odebrać przetworzone dane z bufora. Oznacza to, że konsument może przetwarzać komórkę dopiero wtedy, gdy wszystkie procesory (1-5) zakończyły swoje operacje na tej komórce.
- Konsum**ent** finalizuje przetwarzanie każdej komórki, wyświetlając komunikat "Consumer have eaten cell nr", co oznacza, że dane w tej komórce zostały przetworzone i są gotowe do dalszej obróbki lub wykorzystania poza potokiem.

5. Procesory pośrednie (ID 1-5)

Procesory pośrednie działają w podobny sposób do producenta i konsumenta, jednak pełnią rolę etapów pośrednich w przetwarzaniu potokowym. Każdy procesor działa w ściśle ustalonej kolejności, wykonując swoje zadania na komórkach bufora dopiero po zakończeniu przetwarzania przez poprzedni procesor.

6. Synchronizacja:

- Wątki producenta, procesorów i konsumenta współdzielą bufor i używają mechanizmu **monitorów** (synchronized, wait(), notifyAll()), aby zagwarantować, że operacje na danych odbywają się w odpowiedniej kolejności.
- W przypadku, gdy dany procesor nie może jeszcze przetwarzać danej komórki (ponieważ poprzedni procesor jej nie skończył), wątek procesora czeka, aż poprzednik zakończy swoją operację.

4. Czynniki wpływające na prędkość przetwarzania

Prędkość przetwarzania w systemie przetwarzania potokowego zależy od kilku kluczowych czynników:

1. **Czas przetwarzania przez poszczególne procesory**:
    - Każdy procesor przetwarza dane z różną szybkością (w tym przykładzie symulowany za pomocą Thread.sleep()). Jeśli jeden z procesorów jest wolniejszy od innych, staje się wąskim gardłem całego systemu, co powoduje opóźnienia w przetwarzaniu kolejnych komórek.
2. **Synchronizacja i czekanie wątków**:
    - Mechanizm wait() i notifyAll() powoduje, że wątki czekają na zakończenie przetwarzania przez poprzednie procesy. Im częściej wątki są zmuszane do czekania, tym wolniej działa system.
    - Im większa synchronizacja, tym większe opóźnienia wynikające z oczekiwania na inne wątki.
3. **Rozmiar bufora**:
    - Bufor o większym rozmiarze może pomóc w zwiększeniu przepustowości, ponieważ więcej danych może być przetwarzanych jednocześnie. W tym przykładzie bufor ma rozmiar 100, co pozwala na przetwarzanie 100 jednostek danych w różnych etapach potoku.
4. **Wydajność sprzętu**:
    - Równoległe działanie wielu wątków wymaga odpowiedniej ilości zasobów systemowych (CPU, RAM). Jeśli system nie ma wystarczającej liczby rdzeni lub zasobów do obsługi równoległych procesów, może to znacząco wpłynąć na wydajność przetwarzania.

5. Wnioski

W przedstawionym rozwiązaniu zastosowano przetwarzanie potokowe, które jest efektywnym sposobem na optymalizację systemów wielowątkowych. Zastosowanie monitorów w Javie umożliwiło zapewnienie synchronizacji między wątkami, co zagwarantowało poprawną sekwencję operacji na danych.

Jednak prędkość przetwarzania w tym systemie jest silnie zależna od czasu przetwarzania przez poszczególne procesory oraz od mechanizmów synchronizacyjnych, które mogą prowadzić do nieefektywności, jeśli jeden z procesów staje się wąskim gardłem.

Aby zwiększyć wydajność systemu, można rozważyć:

- Zoptymalizowanie czasu przetwarzania w poszczególnych etapach,
- Zwiększenie rozmiaru bufora,

- Zmniejszenie czasu oczekiwania na synchronizację wątków, np. za pomocą bardziej zaawansowanych technik synchronizacji lub równoległości.

3.Bibliografia

- https://www.dz5.pl/ti/java/java_skladnia.pdf ~Jacek Rumiński, Język Java – rozdział o wątkach
- http://brinch-hansen.net/papers/1999b.pdf ~ Per Brinch Hansen
- https://www.artima.com/insidejvm/ed2/threadsynch.html ~Bill Venners, Inside the Java Virtual Machine - Charter 20 Thread Synchronization
- https://docs.oracle.com/en/java/javase/19/index.html
- https://java-design-patterns.com/patterns/pipeline/#programmatic-example-of-pipeline-pattern-in-java
- https://en.wikipedia.org/wiki/Pipeline_(computing)