Teoria Współbieżności

Laboratorium 3 - Problem ograniczonego bufora i przetwarzanie potokowe.

Magdalena Pastuła

Data laboratorium: 20.10.2020

1. Zadania do wykonania.

Na laboratorium do wykonania były następujące zadania:

1. Zaimplementowanie problemu producenta i konsumenta za pomocą metod `wait()` i `notify()` dla przypadku:
    1. gdzie jest jeden producent i jeden konsument.
    2. gdzie producentów (n1) i konsumentów (n2) jest wiele i n1=n2, n1>n2 oraz n2>n1.
    3. Rozwinięcie podpunktu drugiego poprzez dodanie wywołania funkcji `sleep()` w producentach i konsumentach oraz pomiary czasów.
2. Zaimpementowanie problemu producenta i konsumenta za pomocą operacji `P()` i `V()` dla semafora przy czym:
    1. jest jeden producent i jeden konsument.
    2. jest wiele producentów i konsumentów.
3. Zaimplementowanie rozwiązania przetwarzania potokowego, gdzie rozmiar bufora jest równy 100, a poza producentem i konsumentem jest jeszcze pięć uszeregowanych procesów.

2. Koncepcja rozwiązania.

Zadanie 1

Zaimplementowanie zadanie wykorzystuje funkcje `wait()` oraz `notify()`. W przypadku, gdy ilość elementów w buforze jest maksymalna, wątek próbujący dodać kolejny element wywołuje funkcję `wait()` i czeka. Natomiast w przypadku próby wyciągnięcia z bufora elementu, gdy ten jest pusty, również skutkuje wywołaniem funkcji `wait()` i oczekiwaniem. W obydwóch przypadkach, po wykonaniu operacji wywoływana jest funkcja `notify()`.

Zadanie 2

Do wykonania tego zadania wykorzystano klasę `Semaphore`. Ogólna koncepcja rozwiązania jest podobna jak w zadaniu 1, z tym, że wywołania funkcji `wait()` oraz `notify()` zastąpiono pobieraniem oraz oddawaniem semafora.

Zadanie 3

Do wykonania tego zadania zaimplementowano dodatkową klasę `Processor`, która reprezentuje wątki przetwarzające. Każdy z wątków przetwarzających w związku z tych dokonuje tej samej operacji na danych w buforze, a mianowicie dodaje swój numer ID.
Przetwarzanie potokowe danych jest wykonywane w seriach: najpierw producent zapełnia cały bufor,

następnie każdy z wątków przetwarzający przetwarza cały bufor, a na sam koniec konsument odczytuje cały bufor. W przypadku, gdy łączna ilość danych nie jest wieloktrotnością wielkości bufora, producent dopełnia bufor symbolami `null`.
Do testowania wykorzystano model z jednym producentem, pięcioma procesorami, jednym konsumentem i buforem o długości 10. Wykorzystano mechanizm `wait()`/`notify()`.

3. Implementacja i wyniki.

Zadadnie 1.1

Implementacja bufora:

```
class Buffer {
    private LinkedList<Integer> buff = new LinkedList<>();
    private int maxSize;

    public Buffer(int size) {
        maxSize = size;
    }

    public synchronized void put(int i) {
        while (buff.size() >= maxSize) {
            try {
                wait();
            } catch (Exception ex) {}
        }
        buff.offer(i);
        notify();
    }

    public synchronized int get() {
        while (buff.size() == 0) {
            try {
                wait();
            } catch (Exception ex) { }
        }

        int val = buff.poll();
        notify();

        return val;
    }
}
```

Zadanie 1.2

Implementacja w tym zadaniu niewiele się różni od poprzedniego podpunktu.
W przypadku nierównej ilości wątków producentów i konsumentów należało również zwrócić uwagę,aby zmienić liczbę iteracji danego wątku tak, aby sumarycznie wszystkie elementy zostały wyciągnięte z bufora oraz aby konsumenci nie zawiesili się w oczekiwaniu na nowe elementy.

Zadanie 1.3

Implementacja tego zadania różni się od poprzedniego podpunktu jedynie dodaniem wywołań funkcji `sleep()` podczas dodawania i wyciągania elementów z bufora.

Zmierzone czasy wykonywania całego programu dla poszczególnych stosunków ilości wątków producentów do konsumentów (n1 - liczba producentów, n2 - liczba konsumentów):

Liczba n1,n2	Czas wykonania programu [s]
n1=1, n2=1	2,534
n1=10, n2=10	0,325
n1=10, n2=20	0,386
n1=20, n2=10	0,375

Dla każdego z powyższych przypadków sumaryczna liczba elementów była taka sama, czyli równa 100.

Zadanie 2

Implementacja klasy `Buffer`:

```
class Buffer {
    private LinkedList<Integer> buff = new LinkedList<>();
    private int maxSize;
    private Semaphore _sem = new Semaphore(1);

    public Buffer(int size) {
        maxSize = size;
    }

    public synchronized void put(int i) {
        boolean written = false;
        while (!written) {
            try {
                _sem.acquire();
                if (buff.size() < maxSize) {
                    buff.offer(i);
                    written = true;
                    System.out.println("Written " + i);
                }
                _sem.release();
            } catch (Exception e) {}
        }
    }

    public synchronized int get() {
        boolean read = false;
        int val = 0;

        while (!read) {
```

```
            try {
                _sem.acquire();
                if (buff.size() > 0) {
                    val = buff.poll();
                    read = true;
                }
                _sem.release();
            } catch (Exception e) {}
        }
        return val;
    }
}
```

Zadanie 3

Implementacja dodatkowej metody `process` klasy `Bufor`:

```
public synchronized void process(int processor_no) {
        while (stage != processor_no) {
            try {
                wait();
            } catch (Exception ex) {}
        }

        Integer val = _buff.get(indx);
        if (val != null) {
            _buff.set(indx, val+processor_no);
        }
        indx++;

        if (indx == maxSize) {
            indx = 0;
            stage++;
        }
        notifyAll();
    }
```

Zmierzony czas wykonania całego programu wyniósł 4,641 sekund.

4. Wnioski z ćwiczenia.

Zadanie 1

Czas wykonania programu w dużej mierze zależy od stosunku ilości producentów do konsumentów. Program działa najszybciej, gdy jest ich równa liczba i gdy jest ona większa od 1. W przypadku, gdy producentów jest więcej niż konsumentów, bufor zostaje szybciej zapełniony, przez co producenci potem czekają na jego rozładowanie przez konsumentów. Natomiast w momencie, gdy konsumentów jest więcej, producenci wolniej dokładaja elementów do bufora przez co konsumencji spędzają większość czasu czekając na dane.

Natomiast przypadek, gdzie jest tylko jeden prodcent i tylko jeden konument działa najwolniej, ponieważ między kolejnymi dodawaniami o odejmowaniami elementów z bufora jest duża przerwa: każdy z nich musi najpierw przejść do kolejnego kroku w pętli zewnętrznej.

Zadanie 2

Program do końca nie działa. W przypadku podpunktu jednego producenta i konsumenta oraz liczby elementów równej 10 program działa bez problemu. Natomiast w przypadku większej liczby producentów i konsumentów lub elementów, program się zawiesza. Bardzo możliwe, że jest to kwestia blokowania się wątków wzajemnie: w przypadku, gdy bufor jest pełny, pierwszeństwo dostają kolejni producenci, którzy nie mogą już dodać kolejnego elementu.

Zadanie 3.

Wykorzystane rozwiązanie problemu przetwarzania potokowego nie jest zbyt efektywne. Każdy z wątków czeka, aż poprzedni przejdzie przez cały bufor, zatem im większy bufor, tym program będzie dłużej działać. Ponadto, im większy bufor, tym większe jest prawdopodobieństwo, że nie zostanie zapełniony cały i będzie musiał być dopełniony `nullami`, co dodatkowo zwiększa czas wykonania programu.
W dużej części czas wykonania również zależy od samego mechanizmu dostępu do danych. Przykładem może być sytuacja, gdy podczas dodawania elementów do bufora przez producenta dostęp dostają najpierw pozostałe wątki. Z tego powodu w implementacji wykorzystano funkcję `notifyAll()` zamiast `notify()`, co zmniejsza blokowanie się wzajemne wątków.