SPRAWOZDANIE IV

TEORIA WSPÓŁBIEŻNOŚCI

Java Concurrency Utilities

Producenci i konsumenci z losowa ilością pobieranych i wstawianych porcji

DAWID BIAŁKA

DATA LABORATORIUM 27.10.2020

DATA ODDANIA 03.11.2020

Producenci i konsumenci z losowa ilością pobieranych i wstawianych porcji

- Bufor o rozmiarze 2M
- Jest *m* producentów i *n* konsumentów
- Producent wstawia do bufora losowa liczbę elementów (nie więcej niż M)
- Konsument pobiera losowa liczbę elementów (nie więcej niż M)
- Zaimplementować przy pomocy monitorów Javy oraz mechanizmów Java Concurrency Utilities
- Przeprowadzic porownanie wydajnosci vs. rozne parametry, zrobic wykresy i je skomentowac

Koncepcja

W przypadku, gdy dany konsument lub producent na raz może umieścić lub pobrać więcej niż jeden element może wystąpić sytuacja, że producent nie będzie w stanie umieścić wszystkich elementów, które chce, bo nie ma miejsca albo konsument chce wyciągnąć więcej, niż jest w buforze. Może do doprowadzić do takiego stanu, gdzie konsument, który ma duże zapotrzebowanie (na raz chce zabrać dużo elementów) nigdy się ich nie doczeka, ponieważ będzie wiele konsumentów z małym zapotrzebowaniem, które będą cały czas zabierać produkt. Tak samo z producentami. Dlatego musimy wprowadzić pewne ulepszenia do poprzedniego algorytmu.

```
monitor BUFOR;
const M = ?;
var buf: array[1..2*M] of element;  {bufor 2M-elementowy»
   jest: integer;   {liczba porcji w buforze}
  PIERWSZYPROD, RESZTAPROD, PIERWSZYKONS, RESZTAKONS: condition;
export procedure WSTAW(ile: integer; p: porcja);
begin
 if not empty(PIERWSZYPROD) then wait(RESZTAPROD);
            {inny producent też już czeka}
 while 2*M - jest < ile do wait(PIERWSZYPROD);
            {czekanie na "ile" wolnych miejsc}
 jest := jest + ile;
 do_bufora(ile,p);
 signal(RESZTAPROD);  {pierwszy z reszty będzie pierwszym}
 signal(PIERWSZYKONS) {pierwszy konsument sprawdzi stan}
end; {WSTAW}

export procedure POBIERZ(ile: integer; var p: porcja);
begin
 if not empty(PIERWSZYKONS) then wait(RESZTAKONS);
            {inny konsument już czeka}
 while jest < ile do wait(PIERWSZYKONS);
            {czekanie na żądaną liczbę porcji}
 jest := jest - ile;
 z_bufora(ile,p);
 signal(RESZTAKONS);  {pierwszy z reszty będzie pierwszym}
 signal(PIERWSZYPROD) {pierwszy producent sprawdzi stan}
end; {POBIERZ}
begin
 jest := 0
end; {BUFOR}
```

Jeśli nikt nie czeka na warunku PIERWSZYPROD to sprawdzamy, czy zmieścimy się z naszym produktem, aby go wstawić. Jeśli tak, to wszystko jest w porządku. Jeśli nie to czekamy na warunku PIERWSZYPROD tak długo, aż w buforze będzie wystarczają liczba wolnych miejsc. W tym czasie żaden inny producent nie może nić wkładać do bufora bo zostanie zatrzymany na warunku RESZTAPROD (mamy wątek, który czeka na wolne miejsce). Wtedy konsumenci cały czas będą pobierać elementy aż w końcu zwolni się miejsce na te elementy, które chce umieścić wątek czekający na PIERWSZYPROD. Z tego powodu, że nie możemy umieścić lub pobrać na raz więcej niż M elementów, a bufor ma 2M miejsc to nie wystąpi sytuacja, kiedy jednocześnie producent czeka na zwolnienie elementów a konsument na dołożenie elementów. Analogicznie wygląda sytuacja dla konsumentów.

Wydajność będziemy mierzyć sprawdzając ile czasu dana konfiguracja m i n się wykonywała. Gdy mamy różną liczbę konsumentów i producentów i losową liczbę elementów wkładanych i pobieranych może dojść do sytuacji, kiedy producenci nadal chcą umieszczać elementy w buforze a wszyscy konsumenci już zakończyli pobieranie i żaden konsument nie pozostał żywy (np. gdy jest więcej producentów lub gdy producenci łącznie wylosowali większą liczbę elementów do włożenia niż konsumenci do pobrania). Jeśli taka sytuacja nastąpi, to kończymy działanie całego programu.

Będziemy wykorzystywać mechanizm ReentrantLock z java.util.concurrent.locks.ReentrantLock i Condition z java.util.concurrent.locks.Condition.

Na obiekcie ReentrantLock możemy dodać warunki, na których można czekać i budzić wątki czekający na danym warunku. Aby wykonywać działania na warunku skojarzonym z danym ReentrantLockiem musimy najpierw ten Lock zająć metodą lock(). Tylko jeden wątek na raz może zająć ReentrantLocka.

Wyniki zapisujemy w pliku tekstowym.

Implementacja i wyniki

```
public class Main {

    public static void main(String[] args) {

        int m = Integer.parseInt(args[0]) * 10;
        int n = Integer.parseInt(args[1]) * 10;
        ArrayList<Producer> producers = new ArrayList<>();
        ArrayList<Consumer> consumers = new ArrayList<>();

        final ReentrantLock lock = new ReentrantLock();
        final Condition processesFinished  = lock.newCondition();
        lock.lock();

        Buffer buffer = new Buffer(120);

        for(int i=0; i<n; i++) {
            Consumer p = new Consumer(buffer, processesFinished, lock);
            p.start();
            consumers.add(p);
        }

        for(int i=0; i<m; i++) {
            Producer p = new Producer(buffer, processesFinished, lock);
            p.start();
            producers.add(p);
        }

        long start = System.nanoTime();

        try {
            processesFinished.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        long finish = System.nanoTime();

        long timeElapsed = finish - start;

        producers.forEach(Thread::interrupt);
        consumers.forEach(Thread::interrupt);

        System.out.println(m + " " + n + " " + timeElapsed/1000000);

        try {
            FileWriter myWriter = new FileWriter("results.txt", true);
            myWriter.write(String.valueOf(timeElapsed/1000000) + "\n");
            myWriter.close();
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }

        lock.unlock();

    }
}
```

```java
public class Buffer {

    private List<Integer> bufferList = new ArrayList<>();
    private final int maxSize;

    final ReentrantLock lock = new ReentrantLock();
    final Condition firstProducer  = lock.newCondition();
    final Condition firstConsumer = lock.newCondition();
    final Condition restProducers = lock.newCondition();
    final Condition restConsumers = lock.newCondition();

    public Buffer(int M) {
        this.maxSize = 2 * M;
    }

    public void put(ArrayList<Integer> items) throws InterruptedException{
        lock.lock();
        try {
            if(lock.hasWaiters(firstProducer)) restProducers.await();

            while(bufferList.size() + items.size() > maxSize)
                firstProducer.await();

            bufferList.addAll(items);
            System.out.println("Adding " + items.size() + " elements");

            restProducers.signal();
            firstConsumer.signal();

        } finally {
            lock.unlock();
        }

    }

    public void get(int size) throws InterruptedException {
        lock.lock();
        try {
            if(lock.hasWaiters(firstConsumer)) restConsumers.await();
            while(bufferList.size() < size)
                firstConsumer.await();

            for(int i=0; i<size; i++) {
                bufferList.remove(bufferList.size() - 1);
            }
            System.out.println("Getting " + size + " elements");

            restConsumers.signal();
            firstProducer.signal();

        } finally {
            lock.unlock();
        }
    }

    public int getMaxSize() {
        return maxSize;
    }
}
```

```
public class Producer extends Thread {
    private Buffer _buf;
    private Condition cond;
    private ReentrantLock mainLock;

    public Producer(Buffer buffer, Condition cond, ReentrantLock mainLock) {
        this._buf = buffer;
        this.cond = cond;
        this.mainLock = mainLock;
    }

    public void run() {
        for (int i = 0; i < 100; i++) {
            ArrayList<Integer> items = new ArrayList<>();
            int length = (int) (Math.random() * (_buf.getMaxSize() / 2 - 1));
            for(int j = 0; j<length; j++ ){
                items.add((int) (Math.random() * 10));
            }
            try {
                _buf.put(items);
            } catch (InterruptedException e) {
                return;
            }
        }
        mainLock.lock();
        cond.signal();
        mainLock.unlock();
    }
}
```

```
class Consumer extends Thread {
    private Buffer _buf;
    private Condition cond;
    private ReentrantLock mainLock;

    public Consumer(Buffer buffer, Condition cond, ReentrantLock mainLock) {
        this._buf = buffer;
        this.cond = cond;
        this.mainLock = mainLock;
    }


    public void run() {
        for (int i = 0; i < 100; i++) {
            int length = (int) (Math.random() * (_buf.getMaxSize() / 2 - 1));
            try {
                _buf.get(length);
            } catch (InterruptedException e) {
                return;
            }
        }
        mainLock.lock();
        cond.signal();
        mainLock.unlock();
    }
}
```

Skyrpt w bashu, którym uruchamiamy nasz program dla różnych konfiguracji m i n.

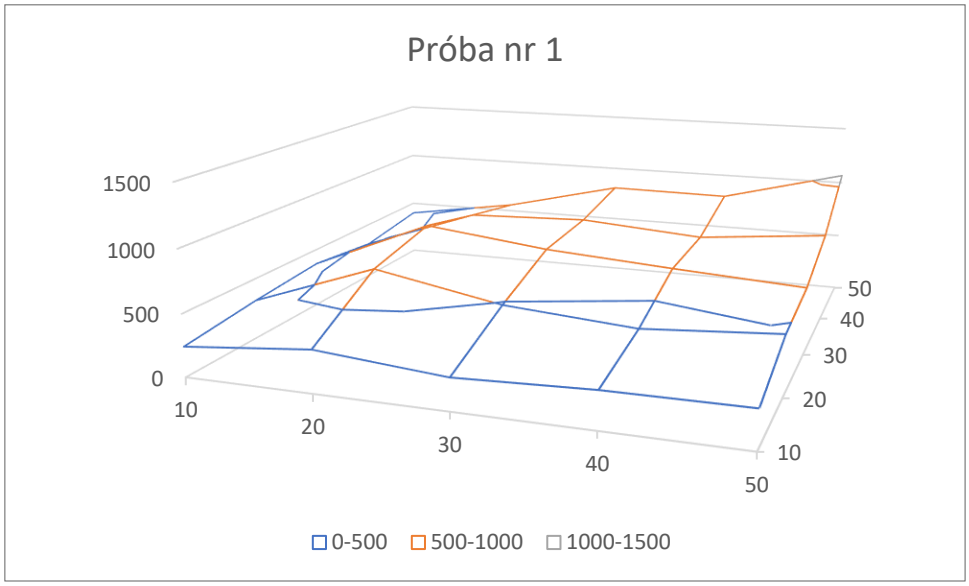
```
#!/bin/bash
for i in {1..5}
do
    for j in {1..5}
    do
        java Main "$i" "$j"
    done
done
```

Czas wykonania w milisekundach dla danej konfiguracji dla trzech prób:

Próba nr 1:

n/m	10	20	30	40	50
10	243	339	254	293	299
20	317	679	489	412	485
30	372	802	681	611	550
40	312	675	710	634	735
50	399	560	807	797	1066

n – konsumenci, m – producenci

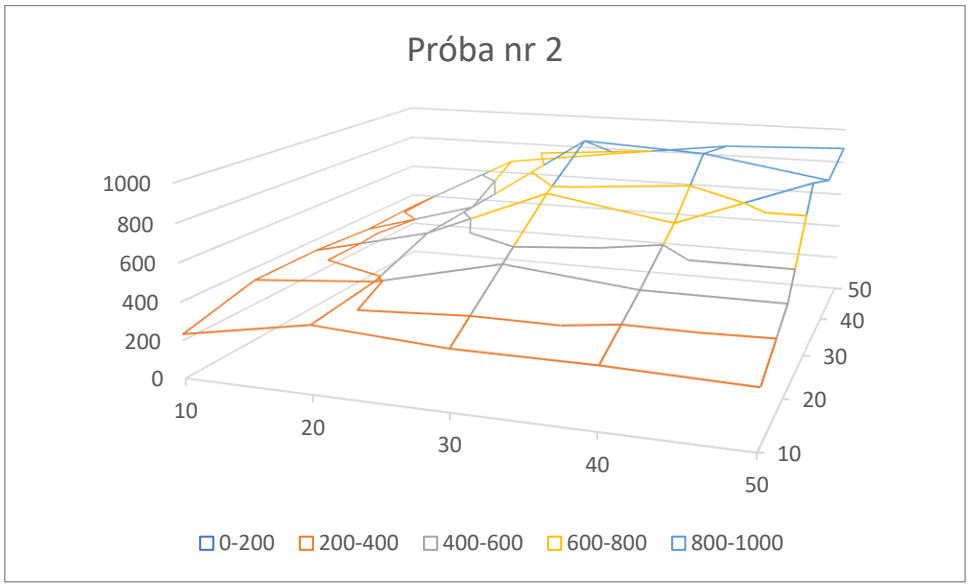


Oś x – producenci, Oś y – konsumenci

Próba nr 2.

n/m	10	20	30	40	50
10	233	355	315	316	300
20	333	389	547	478	479
30	335	496	777	664	929
40	311	507	961	926	815
50	322	676	771	859	885

n – konsumenci, m – producenci

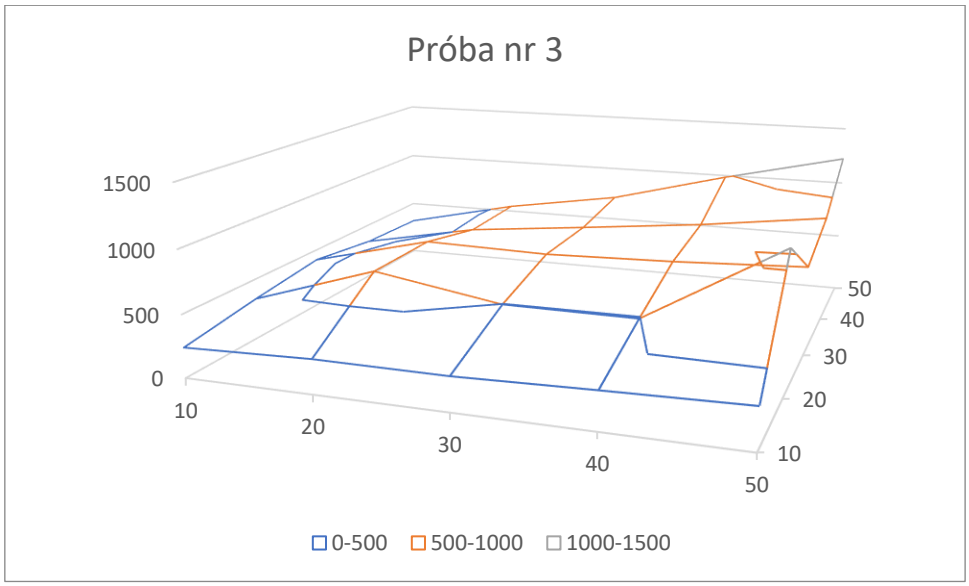


Oś x – producenci, Oś y – konsumenci

Próba nr 3.

n/m	10	20	30	40	50
10	241	272	269	297	321
20	335	664	498	495	1121
30	411	663	643	673	722
40	339	538	643	752	889
50	319	548	714	984	1223

n – konsumenci, m – producenci



Oś x – producenci, Oś y – konsumenci

Wnioski

Na podstawie powyższych wykresów widzimy, że ogólnie wraz ze wzrostem liczby konsumentów lub producentów czas wykonania rośnie. Największy wzrost jest przy liczbie konsumentów i producentów równej 50. Gdy liczba producentów lub konsumentów jest mała, to widzimy, że dla danej wartości liczby konsumentów wzrost liczby producentów nie zmienia w dużym stopniu czasu wykonania. Podobnie jest dla danej wartości liczby producentów i wzroście liczby producentów. Mamy do wykonania większą liczbę operacji, występuje więcej sytuacji, gdzie wątki czekają itd.

Jednak ciężko jest uzależnić czas wykonania tylko od tych dwóch parametrów. Wszystkie te wyniki są uzyskane poprzez wymuszenie zamknięcia programu (wątki konsumentów skończyły się szybciej niż producentów lub wątki producentów się skończyły szybciej niż konsumentów). W zależności od tego, jak losowane były liczby elementów wkładanych lub wyciąganych program mógł kończyć się bardzo szybko lub trwać dłużej (wylosowane liczby elementów były małe, rzadziej występowało czekanie itd.). Ogólnie nie mamy zależności, że dla danej liczby producentów wzrost liczby konsumentów zmniejsza czas trwania lub przy wzroście liczby producentów dla danej liczby konsumentów czas trwania rośnie (co wydaję się intuicyjne). Tak mogłoby się stać gdybyśmy kończyli program tylko w sytuacji, gdzie nie ma już aktywnych producentów (wszystkie zadania wykonane) a konsumenci byliby aktywni tak długo jak tylko jest jakikolwiek aktywny producent.

Bibliografia

Z. Weiss, T. Gruźlewski, Programowanie współbieżne i rozproszone. WNT, Warszawa 1993. ss. 83-84

https://stackoverflow.com/questions/3777772/how-to-stop-a-thread-in-a-threadpool/14969342#14969342

https://docs.oracle.com/javase/1.5.0/docs/guide/concurrency/overview.html

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/Condition.html