Sprawozdanie

Teoria Współbieżności

Laboratorium 3

Joanna Bryk, środa 17:50

1. Wstęp

Celem trzeciego laboratorium było zapoznanie się z problemem ograniczonego bufora (konsument-producent), wykorzystanie poznanych na poprzednich zajęciach narzędzi do synchronizacji wątków i rozwiązanie problemu przetwarzania potokowego. Należało zaimplementować program konsument-producent przy użyciu monitorów (zadanie 1.) i semaforów (zadanie 2.). Zadaniem było także rozważenie przypadków, w których liczba konsumentów i producentów była taka sama i wynosiła 1 oraz sytuacji, gdy liczby te różniły się. Następnie należało zaimplementować rozwiązanie przetwarzania potokowego, przeanalizować działanie programu i odpowiedzieć na postawione pytanie - “Od czego zależy prędkość obróbki w systemie?” (zadanie 3.).

2. Implementacja
    2.1. W zadaniu 1. implementacja programu producent-konsument opiera się na monitorach.

```
class Consumer extends Thread {
    private Buffer _buf;

    public Consumer(Buffer buffer){
        this._buf = buffer;
    }

    public void run() {
        for (int i = 0; i < 100; ++i) {
            System.out.println(this._buf.get());
            try{
                sleep(1000);
            }
            catch(InterruptedException e){
                System.out.println("ERROR");
            }
        }
    }
}

class Producer extends Thread {
    private Buffer _buf;

    public Producer(Buffer buffer){
        this._buf = buffer;
    }

    public void run() {
        for (int i = 0; i < 100; ++i) {
```

```
            _buf.put(i);
            try{
                sleep(1000);
            }
            catch(InterruptedException e){
                System.out.println("ERROR");
            }
        }
    }
}

public class Buffer {
        private final int size;
        private final List <Integer> bufferList = new ArrayList<>();

        public Buffer(int size){
                this.size = size;
        }

        public synchronized void put(int i) {
                while(this.bufferList.size() >= this.size){
                        try{
                                this.wait();
                        } catch (InterruptedException e){
                                System.out.println("ERROR");
                        }
                }
                System.out.println("Putting... " + i);
                this.bufferList.add(i);
                this.notify();
        }

        public synchronized int get() {
                while(this.bufferList.size() == 0){
                        try{
                                this.wait();
                        } catch (InterruptedException e){
                                System.out.println("ERROR");
                        }
                }
                int val = this.bufferList.remove(bufferList.size()-
1);
                System.out.println("Getting an item... " + val);
                this.notify();
                return val;
        }
}

public class PKmon {
    public static void main(String[] args) {

        Buffer buffer = new Buffer(10);

        int n1 = Integer.parseInt(args[0]); //liczba producentów
        int n2 = Integer.parseInt(args[1]); //liczba konsumentów

        long endTime, startTime;

        if (n1 > 1 && n2 > 1) {

            startTime = System.nanoTime();
```

```
            Producer[] producers = new Producer[n1];
            Consumer[] consumers = new Consumer[n2];

            for (int i = 0; i < n1; i++) {
                producers[i] = new Producer(buffer);
                producers[i].start();
            }

            for (int i = 0; i < n2; i++) {
                consumers[i] = new Consumer(buffer);
                consumers[i].start();
            }

            try {
                for (int i = 0; i < n1; i++) {
                    producers[i].join();
                }
                for (int i = 0; i < n2; i++) {
                    consumers[i].join();
                }
            } catch (InterruptedException e) {
                System.out.println("ERROR");
            }

            endTime = System.nanoTime();

        } else {

            startTime = System.nanoTime();

            Producer producer = new Producer(buffer);
            Consumer consumer = new Consumer(buffer);

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

            try {
                producer.join();
                consumer.join();
            } catch (InterruptedException e) {
                System.out.println("ERROR");
            }

           endTime = System.nanoTime();
        }

        System.out.println("Time = " + (endTime - startTime));
    }
}
```

W implementacji powyżej rozmiar buforu wynosi 10 a liczba konsumentów i producentów są argumentami programu podawanymi przy jego uruchomieniu. Kod programu powyżej zawiera już metodę *sleep()* (zadanie 1.c).

Koncepcja rozwiązania jest oczywista – przy użyciu funkcji *wait()* obsłużona została zarówno sytuacja, kiedy wątek próbuje dodać element, gdy bufor jest już przepełniony

jak i przypadek, kiedy w buforze nic nie ma, a konsument chce pobrać liczbę. Kiedy wykonanie tych operacji jest możliwe, wywoływana jest funkcja *notify()*.

Zadanie polegało na obserwacji, co dzieje się w programie przy danej liczbie wątków (n1 – liczba konsumentów, n2 – liczba producentów). W przypadku gdy liczba konsumentów jest taka sama jak liczba producentów, program działa poprawnie (np. n1 = n2 = 1, n1 = n2 = 5). Inne sytuacje możemy podzielić na dwie grupy:

- Więcej producentów niż konsumentów (n1 = 10, n2 = 2) – producent czeka, próbując dodać liczbę do bufora, jednak jest za mało konsumentów, że by zrobić miejsce.
- Więcej konsumentów niż producentów (n1 = 2, n2 = 10) – w tym przypadku konsumentów jest za dużo, więc to oni czekają, aby funkcja *get()* mogła zostać zrealizowana.

Przeprowadziłam także eksperyment co się stanie, kiedy funkcja *wait()* zostanie wywołana w klasie *Producer* lub klasie *Consumer*. W obu tych przypadkach używana była głównie jedna z komórek bufora, ponieważ albo producent zapisywał cały bufor zanim konsument się „obudził", albo też odwrotnie - konsument pobierał z pierwszej komórki, do której producent wpisywał w dużych odstępach czasu.

2.2. Następnie do tego samego problemu użyłam semaforów. Wykorzystałam implementacje z wcześniejszego laboratorium.

```
class Buffer {
    private final int size;
    private final LinkedList<Integer> bufferList = new LinkedList<>();
    private final Semaphore notAvailable;
    private final Semaphore available;
    private final BinarySemaphore binSem;


    public Buffer(int size) {
        this.size = size;
        this.available = new Semaphore(size);
        this.notAvailable = new Semaphore(0);
        this.binSem = new BinarySemaphore(true);
    }

    public void put(int i) {
        this.available.P();

        this.binSem.P();
        this.bufferList.add(i);
        this.binSem.V();

        this.notAvailable.V();
    }


    public int get() {
        this.notAvailable.P();

        this.binSem.P();
        int val = this.bufferList.removeFirst();
        this.binSem.V();
```

```
        this.available.V();
        return val;
    }
}
```

W programie działają 3 semafory – 2 semafory licznikowe i 1 semafor binarny. Semafor licznikowy *available* odpowiada za kontrolę, ile miejsc w buforze jest wolnych (na początku zmienna ta ma wartość 10) z kolei *notAvailable* za pilnowanie, aby żaden konsument nie pobrał z bufora, kiedy nie ma tam elementów. Semafor binarny pilnuje dostępu do bufora (*bufferList*).

W przypadku, gdy mamy tyle samo producentów co konsumentów program nie zawiesza się, wypisywane komunikaty sugerują poprawne działanie. Kiedy natomiast producentów/konsumentów jest więcej niż drugiej grupy wątków, sytuacja jest analogiczna jak w zadaniu 1.

2.3. Poniżej znajduje się implementacja przetwarzania potokowego. Zdecydowałam się na użycie monitorów. W klasie bufor dodałam metodę *process*, która odpowiada za przeprowadzanie transformacji elementów bufora, a także zmienne pomocnicze (*lastAdded, lastTaken, items, lastProcessed*). Dodatkowo dodałam klasę *BufforItem*, która oprócz wartości przechowuje informację, ile razy ten element bufora został przetworzony. Dzięki niej procesy przetwarzające będą zmieniać wartość w odpowiedniej kolejności.

Działanie programu jest podobne jak w zadaniu 1. Producent „czeka", kiedy elementów jest zbyt dużo a następnie wyszukuje pierwszego indeksu, w którym nie ma żadnego elementu. Procesy przetwarzające działają, kiedy elementów jest więcej niż 0 (pętla *while (items <= 0)* i funkcja *wait()*). Każdy proces ma zapisany ostatni indeks, z którego element przetwarzał (tablica *lastProcessed*). Wykonując operacje, zmienia także atrybut *processed* obiektowi *item* (*increaseProcess()*). Konsument natomiast czeka aż element zostanie odpowiednią liczbę razy przetworzony a następnie pobiera element z listy, wypisuje go oraz zmniejsza zmienną *items*.

```
public class Pipeline {
    public static void main(String[] args) {
        int n = 3;
        int size = 10;

        Buffer buffer = new Buffer(size,n);

        LinkedList<Thread> threads = new LinkedList<>();

        threads.add(new Producer(buffer));
        for(int i=0; i<n; i++){
            threads.add(new ProcessThread(buffer,i, size));
        }
        threads.add(new Consumer(buffer));

        for(Thread thread: threads){
            thread.start();
```

```
        }
        for(Thread thread: threads){
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }
}
public class BufferItem {
    private int processed;
    private int val;

    public BufferItem(int val){
        this.val = val;
        this.processed = 0;
    }

    public int getProcessed(){
        return processed;
    }
    public int getVal(){
        return val;
    }

    public void setVal(int val){
        this.val = val;
    }

    public synchronized void increaseProcess(){
        processed+=1;
    }
}
public class Buffer {

    private final int size;
    private final ArrayList<BufferItem> bufferList;
    private final int processThreads;
    private int lastAdded = 0;
    private int lastTaken = 0;
    private int items = 0;
    private final int[] lastProcessed;


    public Buffer(int size, int processThreads) {

        this.lastProcessed = new int[processThreads];
        this.size = size;
        this.processThreads = processThreads;
        this.bufferList = new ArrayList<>(size);

        for (int i = 0; i < size; i++) {
            bufferList.add(i, null);
        }
        for (int i = 0; i < processThreads; i++) {
            lastProcessed[i] = 0;
        }

    }
```

```
    public synchronized void put(int i) {
        while (items >= size) {
            System.out.println("Wanting to add.. number of items = "
+ items);
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("ERROR");
            }
        }
        //szuka miejsca do wstawienia
        for (int idx = lastAdded; ; idx = (idx + 1) % size) {
                if (bufferList.get(idx) == null) {
                System.out.println("Adding to " + idx);
                BufferItem item = new BufferItem(i);
                lastAdded = idx;
                bufferList.set(lastAdded, item);
                items++;
                break;
            }
        }
        notify();

    }

    public synchronized void process(int processID) throws
InterruptedException {
        while (items <= 0) {
            System.out.println("Waiting for producer to put....");
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("ERROR");
            }
        }
        int index = lastProcessed[processID];
        BufferItem item = bufferList.get(index);
        while (item == null || processID != item.getProcessed()) {
            item = bufferList.get(index);
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("ERROR");
            }
        }
        int newVal = transform(processID, item.getVal());
        item.setVal(newVal);
        item.increaseProcess();
        bufferList.set(index, item);
        System.out.println("New value at " + index + " val = " +
item.getVal() + " processID " + processID);
        lastProcessed[processID] = (index + 1) % size;
        notify();

    }

    public synchronized BufferItem get() throws InterruptedException
{
        while (items <= 0) {
            try {
```

```
                wait();
            } catch (InterruptedException e) {
                System.out.println("ERROR");
            }
        }

        BufferItem item = bufferList.get(lastTaken);
        while (item == null || item.getProcessed() != processThreads)
{
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("ERROR");
            }
        }
        bufferList.set(lastTaken, null);
        items--;
        lastTaken = (lastTaken + 1) % this.size;

        notify();
        return item;
    }

    public int transform(int ID, int val) {
        if (ID == 0) {
            val = val + 21;
        } else if (ID % 2 == 0) {
            val *= 50;
        } else {
            val = val % 6;
        }
        return val;
    }
}
```

3. Podsumowanie
    3.1. Monitory są odpowiednimi narzędziami do rozwiązania problemu ograniczonego bufora. Jest to prosty mechanizm, gwarantujący poprawne działanie programu konsument-producent. Należy jednak pamiętać, że może dojść do sytuacji, w której zbyt dużo producentów „wkłada” coś do bufora przy małej liczbie konsumentów (których już nie ma, aby wyciągać te elementy) bądź też na odwrót. Wtedy program zawiesza się. Nie zdarza się to, gdy liczba konsumentów i producentów jest taka sama.
    3.2. Semafory także sprawdziły się i odpowiednio dysponowały dostępem wątków do wspólnego bufora. Jednak ograniczenia co do liczby konsumentów i producentów są podobne jak w punkcie 3.1.
    3.3. Przetwarzanie potokowe jest narzędziem, które pozwala odpowiednio rozdzielić operacje, tak aby wykonywały się jednocześnie przez różne procesy. Może to ułatwiać organizację programu, jednak to rozwiązanie niesie ze sobą pewne minusy. Problem polega na tym, że czas wykonania całego programu zależy bardzo od czasu poszczególnych procesów a także kolejności w jakiej będą się one wykonywać. Może się okazać, że jeden proces, który wykonuje się

bardzo długo będzie blokował kolejne, uniemożliwiając im pracę – wtedy czas działania programu wydłuży się znacząco.