Teoria Współbieżności Lab5 - sprawozdanie

Autor: Michał Flak

Ćwiczenie 1 - Przetwarzanie potokowe z buforem

1. Bufor o rozmiarze N - wspólny dla wszystkich procesów!
2. Proces A będący producentem.
3. Proces Z będący konsumentem.
4. Procesy B, C, ..., Y będące procesami przetwarzającymi. Każdy proces otrzymuje dana wejściową od procesu poprzedniego, jego wyjście jest konsumowane przez proces następny.
5. Procesy przetwarzają dane w miejscu, po czym przechodzą do kolejnej komórki bufora i znowu przetwarzają ja w miejscu.

Wykonanie

Wnioskując z pomocniczego schematu kodu, założyłem, że dopuszczalna jest realizacja na wątkach - nie procesach.

Chodzi o zrealizowanie przetwarzania potokowego - analogiczne do przetwarzania potokowego w architekturze procesorów. Każde z pól bufora powinno być odwiedzone raz przez każdy wątek. Operacje na polach bufora powinny być atomowe - nie mogą na jednym polu działać dwa wątki naraz. Wątki powinny zachować kolejność - nie mogą się wyprzedzać.

W celu zobrazowania działania, użyto typu String jako pola bufora. Każdy z procesów będzie modyfikował napotkany string, dopisując do niego swój identyfikator. Konsument będzie wypisywał powstały string na standardowe wyjście.

Oczekujemy więc wyjścia, dla N (rozmiar bufora) = 5, M (ilość wątków) = 10:

```
ind0thr1thr2thr3thr4thr5thr6thr7thr8
ind1thr1thr2thr3thr4thr5thr6thr7thr8
ind2thr1thr2thr3thr4thr5thr6thr7thr8
ind3thr1thr2thr3thr4thr5thr6thr7thr8
ind4thr1thr2thr3thr4thr5thr6thr7thr8
```

Poszczególne części programu zostały zrealizowane nastepująco:

Bufor

Bufor oparty został na tablicy `String`-ów. Każdemu polu bufora został stworzony odpowiadający `Object` którym blokujemy dostęp do tego pola, oraz `AtomicIntegerArray currentAllowedUser`, gwarantujący poprawną kolejność przetwarzania przez wątki.

Bufor zasadniczo zawiera trzy metody:

- `void insert(int threadId, int i, String value)` Wywoływana przez producenta o threadId = 0, wstawia wartość `value` na miejscu `i`, po czym inkrementuje `currentAllowedUser[i]`, przygotowując pole na wizytę pierwszego wątku przetwarzającego. Wybudza też wątki czekające na dostęp do tego pola.
- `void modifyInPlace(int threadId, int i, BufferModifier lambda)` Wywoływana przez wątki przetwarzające o threadId od 1 do M-2, akceptuje lambdę o następującym typie: `String modify(String before)`. Wykonuje tą funkcję z argumentem w postaci wartości pola bufora na miejscu `i`, po czym zamienia tą wartość w buforze na wartość zwróconą przez funkcję. Lambda została zastosowana w celu utrzymania kontroli nad synchronizacją w klasie `Buffer` - bez wyciekania abstrakcji do klasy `Converter`. Jak wyżej, wybudza oczekujących na dostęp do pola.
- `String retrieve(int threadId, int i)` Wywoływana przez konsumenta o threadId = M-1, zwraca wartość bufora w polu `i`, po czym ustawia `currentAllowedUser[i]` na 0 - tym samym zwalnia pole

bufora producentowi do ponownego wykorzystania. Jak wyżej, wybudza oczekujących na dostęp do pola.

Każda z tych metod przed podjęciem działania sprawdza, czy argument `threadId` zgadza się z wartością `currentAllowedUser` odpowiadającą polu. Jeśli nie - czeka, aż inny wątek o mniejszym ID zakończy pracę nad polem i inkrementuje `currentAllowedUser`.

Kod bufora:

```
class Buffer {
    private int _size;
    private Object[] bufferLocks;
    private String[] content;
    private AtomicIntegerArray currentAllowedUser; //id of current allowed thread

    public interface BufferModifier {
        String modify(String before);
    }

    public int get_size() {
        return _size;
    }

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

        bufferLocks = new Object[size];
        content = new String[size];
        currentAllowedUser = new AtomicIntegerArray(size);
        for (int i = 0; i < size; i++) {
            bufferLocks[i] = new Object();
        }
    }

    //Used by producer
    public void insert(int threadId, int i, String value) {
        synchronized (bufferLocks[i]){
            while (threadId != currentAllowedUser.get(i)) {
                try {
                    bufferLocks[i].wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            content[i] = value;
            currentAllowedUser.incrementAndGet(i);
            bufferLocks[i].notifyAll();
        }
    }

    //Made in order to keep control over locks within Buffer class
    public void modifyInPlace(int threadId, int i, BufferModifier lambda) {
        synchronized (bufferLocks[i]){
            while (threadId != currentAllowedUser.get(i)) {
                try {
```

```
                    bufferLocks[i].wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            String previous = content[i];
            content[i] = lambda.modify(previous);
            currentAllowedUser.incrementAndGet(i);
            bufferLocks[i].notifyAll();
        }
    }

    //Used by consumer
    public String retrieve(int threadId, int i) {
        synchronized (bufferLocks[i]){
            while (threadId != currentAllowedUser.get(i)) {
                try {
                    bufferLocks[i].wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            var result = content[i];
            currentAllowedUser.set(i, 0);
            bufferLocks[i].notifyAll();
            return result;
        }
    }
}
```

Main

Tutaj tworzony jest bufor, wątki oraz nadawane są im ich ID.

```
public class Main {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        int bufferSize = 1000;
        Buffer buffer = new Buffer(bufferSize);
        int noOfThreads = 100;

        //create producer
        Thread tp = new Thread(new Producer(buffer, 0));
        tp.start();

        //create converters
        List<Thread> threadList = new ArrayList<>();
        for (int i = 1; i < noOfThreads - 1; i++) {
            Thread t = new Thread(new Converter(buffer, i));
            threadList.add(t);
            t.start();
        }

        //create consumer
```

```
        Thread tc = new Thread(new Consumer(buffer, noOfThreads - 1));
        tc.start();

        //join threads
        try {
            tp.join();
            for (Thread t : threadList) t.join();
            tc.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        long time = System.currentTimeMillis() - start;
        System.out.println("czas: " + time);
    }
}
```

Producent

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

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

    public void run() {
        for (int i = 0; i < _buf.get_size(); i++) {
            _buf.insert(id, i, "ind" + i);
        }
    }
}
```

Konwerter

```
class Converter extends Thread {
    private Buffer _buf;
    private int id;

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

    public void run() {
        for (int i = 0; i < _buf.get_size(); i++) {
            _buf.modifyInPlace(id, i, (str) -> {
                return str + "thr" + id;
            });
        }
    }
}
```

Konsument

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

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

    public void run() {
        for (int i = 0; i < _buf.get_size(); i++) {
            System.out.println(_buf.retrieve(id, i));
        }
    }
}
```

Działanie

Program działa zgodnie z oczekiwaniami. Dla wspomnianego przypadku output jest zgodny z oczekiwanym:

```
ind0thr1thr2thr3thr4thr5thr6thr7thr8
ind1thr1thr2thr3thr4thr5thr6thr7thr8
ind2thr1thr2thr3thr4thr5thr6thr7thr8
ind3thr1thr2thr3thr4thr5thr6thr7thr8
ind4thr1thr2thr3thr4thr5thr6thr7thr8
czas: 6
```

Działa również dla dużo większych liczb (N=1000, M=100):

```
ind0thr1thr2thr3thr4[...]thr92thr93thr94thr95thr96thr97thr98
ind1thr1thr2thr3thr4[...]thr92thr93thr94thr95thr96thr97thr98
[...]
ind999thr1thr2thr3thr4[...]thr92thr93thr94thr95thr96thr97thr98
czas: 96
```

Wnioski

Podając buforowi informację na temat ilości wątków moglibyśmy ograniczyć liczbę locków oraz stanów *z* rozmiaru bufora *do* ilości wątków, zmniejszając zużycie pamięci.

Sukcesem okazał się brak wycieku abstrakcji synchronizacji wątków z bufora - wszystko jest w jednej klasie.

Ćwiczenie 2 - Producenci i konsumenci z losowa ilością pobieranych i wstawianych porcji

1. Bufor o rozmiarze 2M
2. Jest m producentów i n konsumentów
3. Producent wstawia do bufora losowa liczbę elementów (nie więcej niż M)
4. Konsument pobiera losowa liczbę elementów (nie więcej niż M)

Bufor

Zdecydowałem się na bufor złożony z:

- tablicy intów

- pozycji “głowicy” czytającej
- pozycji “głowicy” zapisującej
- obiektu `writeLock`, który zapewnia atomowość modyfikacji `writehead` oraz atomowość całej operacji zapisu

Bufor można dość łatwo uczynić cyklicznym, tym samym zapewniając nieskończoną pracę programu, rozpatrzyłem jednak przypadek prostszy - bufora o skończonej długości.

Zasadnicze metody bufora:

- `void insert(int elems[])` Wstawia tablicę elementów do bufora poczynając od pozycji `writeHead`. Zajmuje `writeLock` na całą operację zapisu. Na koniec działania woła potencjalnych konsumentów czekających na dane za pomocą `writeLock.notifyAll()`. W przypadku, kiedy elementów do wstawienia jest więcej niż wynosi pojemność bufora, wstawia tylko tyle ile się mieści.
- `int[] retrieve(int requestedSize)` Zwraca wycinek bufora od pozycji `readHead` do `readHead + requestedSize`. Zajmuje `writeLock` na chwilkę, celem sprawdzenia, czy jest wystarczająca liczba elementów. Jeśli nie ma, woła `writeLock.wait()` aż zostaną one wyprodukowane. Jeśli żądana ilość elementów jest większa niż znajdująca się w buforze, zwracane są wszystkie pozostałe elementy w buforze (od `readHead` do końca).

Kod bufora:

```
public class Buffer {
    private int _size;
    private int[] content;
    private int readHead = 0;
    private Integer writeHead = 0;
    private Object writeLock;

    public int get_size() {
        return _size;
    }

    public Buffer(int size) {
        this._size = size;
        content = new int[size];
        writeLock = new Object();
    }

    //Used by producer
    public void insert(int elems[]) {
        int i = 0;
        //take writeLock in order to ensure write is atomic
        //ie. writeHead actually corresponds to
        //the number of elems in array
        synchronized (writeLock) {
            while (writeHead < _size && i < elems.length)
            {
                content[writeHead] = elems[i];
                i++;
                writeHead++;
            }

            //notify readers waiting for new elements
            writeLock.notifyAll();
```

```
        }
    }

    //Used by consumer
    public int[] retrieve(int requestedSize) {
        int truncatedRequestedSize =
                requestedSize + readHead > _size ? //requested more than in buffer?
                        _size - readHead : // TRUE: return remaining;
                        requestedSize; // FALSE : return requested;

        synchronized (writeLock) {
            while(truncatedRequestedSize + readHead > writeHead) {
                try {
                    writeLock.wait(); //wait until enough elements are in the buffer
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }

        //enough elements in buffer, return:
        var result = Arrays.copyOfRange(content, readHead, readHead + truncatedRequestedSize);
        readHead += truncatedRequestedSize;
        return result;
    }
}
```

Main

Tworzy bufor, producenta i konsumenta. Zapewnia ten sam generator losowy dla producenta i konsumenta celem uniknięcia tych samych wyników dwóch generatorów utworzonych w tym samym czasie przez te wątki.

```
public class Main {

    public static void main(String[] args) {
        var buffer = new Buffer(100);
        var generator = new Random();

        var pThread = new Producer(buffer, generator);
        var cThread = new Consumer(buffer, generator);

        pThread.start();
        cThread.start();

        try {
            pThread.join();
            cThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Producent

Produkuje elementy zgodnie z poleceniem, aż zapełni bufor. W jednej partii są produkowane liczby jak: [0,1,2,3,4,...]

```
class Producer extends Thread {
    private Buffer _buf;
    private Random generator;

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

    public void run() {
        System.out.println("Started producing");
        int elementsToAdd = _buf.get_size();
        int maxBatchSize = elementsToAdd / 2;

        while(elementsToAdd > 0) {
            //generate between 1 and maxBatchSize
            var currentSize = generator.nextInt(maxBatchSize-1) + 1;
            if(currentSize > elementsToAdd) currentSize = elementsToAdd;
            int[] arr = new int[currentSize];
            for(int i = 0; i < currentSize; i++) {
                arr[i] = i;
            }
            _buf.insert(arr);
            elementsToAdd -= currentSize;
        }
        System.out.println("Finished producing");
    }
}
```

Konsument

Konsumuje elementy aż do opróżnienia bufora, zgodnie z poleceniem.

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

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

    public void run() {
        System.out.println("Started consuming");
        int elementsToConsume = _buf.get_size();
        int maxBatchSize = elementsToConsume / 2;

        while(elementsToConsume > 0) {
            //generate between 1 and maxBatchSize
            var currentSize = generator.nextInt(maxBatchSize-1) + 1;
```

```
            if(currentSize > elementsToConsume) currentSize = elementsToConsume;
            System.out.println(Arrays.toString(_buf.retrieve(currentSize)));
            elementsToConsume -= currentSize;
        }
        System.out.println("Finished consuming");
    }
}
```

Wyniki

Przykładowe uruchomienie dla rozmiaru bufora = 100:

```
Started producing
Started consuming
Finished producing
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 0, 1, 2]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
[9, 10, 11, 12, 13, 14, 15]
[16, 17, 18, 19, 20, 21, 22, 23, 24, 0, 1, 2, 3, 4]
[5, 6, 7, 8, 9, 10, 11, 12, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 2
[26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 0, 1, 2, 3]
Finished consuming
```

Zgodne z oczekiwaniami. Liczby są uporządkowane w zakresach produkowanych przez producenta, te zakresy jednak nie pokrywają się z tymi konsumowanymi przez konsumenta - działają one niezależnie. Wypisywana jest cała zawartość bufora. Program się nie zakleszcza.