Teroria Współbieżności

Laboratoium 3

Andrzej Ratajczak

1. Problem ograniczonego bufora (producentów-konsumentów)

a) wait() / notify() / nKonsumentów = nProducentów = 1

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

    public Producer(Buffer buf) {
        _buf = buf;
    }

    public void run() {
        for (int i = 0; i < 100; ++i) {
            _buf.put(i);
        }
    }
}

class Consumer extends Thread {
    private Buffer _buf;

    public Consumer(Buffer buf) {
        _buf = buf;
    }

    public void run() {
        for (int i = 0; i < 100; ++i) {
            System.out.println(_buf.get());
        }
    }
}

class Buffer {

    private int _size;
    private int iterator = -1;
    private int[] buffer;

    public Buffer(int size) {
        _size = size;
        buffer = new int[size];
    }

    private boolean canPut() {
        return iterator + 1 < _size;
    }
```

```
    public synchronized void put(int i) {
        while (!canPut()) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("Ooops, something went wrong " +
                                   e.getMessage());
            }
        }
        iterator++;
        buffer[iterator] = i;
        notify();
    }

    private boolean canGet() {
        return iterator >= 0;
    }

    public synchronized int get() {
        while (!canGet()) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("Ooops, something went wrong " +
                                   e.getMessage());
            }
        }
        int k = buffer[iterator];
        iterator--;
        notify();
        return k;
    }
}

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

        Buffer buffer = new Buffer(4);

        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("Ooops, something went wrong " + e.getMessage());
        }
    }
}
```

Kod źródłowy 1. Impelementacja zadanego problemu przy użyciu systemowych mechanizmów.

Efekt nie jest przypadkowy i łatwy do przewidzenia. Każdy wątek produkuje/konsumuje do/z ograniczonego bufora. Systemowy monitor pilnuje tak aby tylko jeden wątek mógł naraz operować na buforze.

b) wait() / notify() / nKonsumentów > 1 ∧ nProducentów > 1

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

        Buffer buffer = new Buffer(4);



        int pSize = 4;
        int cSize = 4;

        Producer[] producer = new Producer[pSize];
        Consumer[] consumer = new Consumer[cSize];

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

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

        try {
            for(int i = 0; i < pSize; i++) {
                producer[i].join();
            }

            for(int i = 0; i < cSize; i++) {
                consumer[i].join();
            }
        } catch(InterruptedException e) {
            System.out.println("Ooops, something went wrong " + e.getMessage());
        }

    }
}
```

Kod źródłowy 2. Zmiany w kodzie źródłowym 1, tak aby zrealizować wymagania zadania.

Powyższy kod pozwala stworzyć większą liczbę wątków produkujących i konsumujących. Program dla takiej samej wartości nProducentów = nKonsumwntów zachowuje się bardzo podobnie do zadania z pkt a), jednakże jeżeli stworzymy różną ilość wątków dochodzi do zakleszczenia powodem jest implementacja działania wątków producentów/konsumentów: zostały zafixowane na wykonanie swoich akcji 100 razy, zanim się poprawnie zakończą. Jeżeli jest więcej konsumentów, to zabraknie producentów, którzy by wpisali coś do bufora i konsumerzy będą oczekiwali na zapłenienie bufora, z kolei kiedy brakuje konsumentów, może nastąpić do zapełnienia bufora (przy założeniu że bufor < (nProducentów – nKonsumentów)*100), producenci nie będą mieli gdzie wpisać wartościu które zamieżają wyprodukować.

c) wait() / notify() / sleep

Do konsumenta dodałem instrukcję sleep(1000) po odebraniu wartości z bufora. Zaowocowało to tym, że bufor praktycznie zawsze był zapełniony, i efektywnie używano tylko jednej komórki bufora. Z kolei kiedy intrukcję sleep() użyłem w przypadku producenta, praktycznie nigdy nie udało się producentowi wpisać więcej jak jednej wartości do bufora, zanim ta by została

skonsumowana. Żeby był sens używać bufora należy zagwarantować podobny czas działania wątków konsumentów i producentów.

d) P() / V() / nKonsumentów = nProducentów = 1

Do implementacji zadania wykorzystam semafor z poprzedniego laboratorium.

```
class Semafor {
    private boolean _stan = true;
    private int _czeka = 0;

    public Semafor(boolean stan) {
        _stan = stan;
    }

    public synchronized void P() {
        _czeka++;
        while(_stan == false) {
            try{
                this.wait();
            } catch(InterruptedException e) {
                System.out.println("Ooops, something went wrong " +
                                   e.getMessage());
            }
        }
        _czeka--;
        _stan = false;
    }

    public synchronized void V() {
        if(_czeka > 0) {
            this.notify();
        }
        _stan = true;
    }
}
```

Kod źródłowy 3. Implementacja semafora binarnego.

Następnie zamieniam mechanizm monitora systemowego na własną implementację semafora.

```
class Buffer {

    private int _size;
    private int iterator = -1;
    private int[] buffer;
    Semafor semafor = new Semafor(true);

    public Buffer(int size) {
        _size = size;
        buffer = new int[size];
    }

    private boolean canPut() {
        return iterator + 1 < _size;
    }

    public void put(int i) {
        semafor.P();
        while (!canPut()) {
            semafor.V();
            semafor.P();
        }
```

```
        iterator++;
        buffer[iterator] = i;
        semafor.V();
    }

    private boolean canGet() {
        return iterator >= 0;
    }

    public int get() {
        semafor.P();
        while (!canGet()) {
            semafor.V();
            semafor.P();
        }
        int res = buffer[iterator];
        iterator--;
        semafor.V();
        return res;
    }
}
```

Kod źródłowy 4. Klasa bufora z sekcjami krytycznymi zabezpieczonymi semaforem.

Rezultat taki sam jak dla programu używającego bloku synchronized

e) P() / V() / nKonsumentów > 1 ∧ nProducentów > 1

Powyższy problem to połączenie zadania b) oraz d). Rezultat działania i wnioski są takie same jak w zadaniu b)

2. Przetwarzanie potokowe z buforem

Zaimplementować rozwiązanie przetwarzania potokowego (Przykładowe zalozenia: bufor rozmiaru 100, 1 producent, 1 konsument, 5 uszeregowanych procesów przetwarzających.)

Do implementacji powyższego problemu potrzebuję specjalnej struktury bufora, która będzie posiadała sekcję krytyczną na każdej komórce.

```
class BufferCell {
    private int payload;
    private int expectedPriority = 0;
    public synchronized void operateOn(
                  int streamPriority, Function<Integer, Integer> func) {
        while (streamPriority != expectedPriority) {
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println("Ooops, something went wrong " +
                                   e.getMessage());
            }
        }
        payload = func.apply(payload);
        expectedPriority++;
        notifyAll();
    }
}
```

Kod źródłowy 5. Kod komórki bufora, która przechowuje dane, udostępnia API pozwalające zmienić stan komórki, kotnroluje priorytet procesu który powinien przetwarzać komórkę.

Teraz przygotowuję klasę bufora która będzie zarządzała naszymi komórkami.

```
class Buffer {

    private int _size;
    BufferCell[] bufferCells;
    public Buffer(int size) {
        _size = size;
        bufferCells = new BufferCell[size];
        for(int i = 0; i < size; i++) {
            bufferCells[i] = new BufferCell();
        }
    }

    public void execute(int i, int streamPriority, Function<Integer, Integer>
func) {
        bufferCells[i].operateOn(streamPriority, func);
    }

    public int get_size() {
        return _size;
    }
}
```

Kod źródłowy 6. Implementacja bufora do przetwarzania potokowego.

Teraz potrzebuję klasy reprezentującej wątek który realizuje zadania procesu przetwarzającego (w szczególności producent, konsument).

```
class ProducerConsumer extends Thread {
    private Buffer _buf;
    private int streamPriority;
    private Function<Integer, Integer> _func;
    public ProducerConsumer(Buffer buf, int sp, Function<Integer,Integer> func){
        _buf = buf;
        streamPriority = sp;
        _func = func;
    }

    public void run() {
        for (int i = 0; i < _buf.get_size(); ++i) {
            _buf.execute(i, streamPriority, _func);
        }
    }
}
```

Kod źródłowy 7. Klasa wątku operującego na buforze.

Na koniec implementacja metody main aby wystartować nasz program.

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

        Buffer buffer = new Buffer(100);
        ArrayList<Function<Integer, Integer>> functions = new
                                                    ArrayList<>(Arrays.asList(
            x -> {
                Random rn = new Random();
                int randomNumber = rn.nextInt(100) + 1;
                System.out.println("Producing number: " + randomNumber);
                return randomNumber;
            },
            x -> x / 2,
```

```
                x -> x * 5,
                x -> x % 13,
                x -> x + 2,
                x -> x << 2,
                x -> {
                    System.out.println("Consuming number: " + x);
                    return x;
                }
            ));
            int countOfWorkers = functions.size();
            ProducerConsumer[] workers = new ProducerConsumer[countOfWorkers];

            for(int i = 0; i < countOfWorkers; i++) {
                workers[i] = new ProducerConsumer(buffer, i, functions.get(i));
                workers[i].start();
            }

            try {
                for(int i = 0; i < countOfWorkers; i++) {
                    workers[i].join();
                }
            } catch (InterruptedException e) {
                System.out.println("Ooops, something went wrong " + e.getMessage());
            }
        }
    }
```

Kod źródłowy 8. Implmentacja main.

Założeniem zadania było brak użycia kolejki fifo. Aby zagwarantować kolejność wykonywania zadań wprowadziłem priorytet (kolejności od 0 do N). Należy pamiętać, że jako pierwsza instrukcja, w tablicy funkcji powinna znaleźć się funkcja producenta, a ostatnia funkcja konsumenta.

Od czego zależy prędkość obróbki w tym systemie?

Powyższa implementacja podatna jest na efekt konwoju. Jeżeli będzie jakaś funkcja, która będzie miała bardzo długi czas wykonania, to pozostałe procesy zdążą dogonić wolny proces i za każdym razem będą oczekiwały na zwolnienie monitora, aby same mogły zacząć pracę. Rozwiązaniem tego jest uszeregowanie procesów, tak aby szybsze operacje były wcześniej wykonywane (oczywiście w miarę możliwości).