Teoria współbieżności

Laboratorium 3

Problem producenta i konsumenta

Albert Gierlach

24.10.2020

1. Zadanie 1

Dany jest bufor, do którego producent może wkładać dane, a konsument pobierać. Napisać program, który zorganizuje takie działanie producenta i konsumenta, w którym zapewniona będzie własność bezpieczeństwa i żywotności.

Zrealizować program przy pomocy metod wait()/notify().

a. dla przypadku 1 producent/1 konsument

b. dla przypadku n1 producentów/n2 konsumentów (n1>n2, n1=n2, n1<n2)

c. wprowadzić wywołanie metody sleep() i wykonać pomiary, obserwując zachowanie producentów / konsumentów

2. Koncept rozwiązania

Rozwiązanie będzie opierać się na współdzielonym buforze (o określonej wielkości), do którego będą miały dostęp wątki konsumentów i producentów. Każdy z tych wątków otrzyma dostęp do bufora oraz liczbę iteracji, która określa ilość operacji dodania lub pobrania produktu z bufora. Warto zauważyć, że w takim podejściu ilość produktów dodanych musi być równa ilości produktów odebranych. W tym celu wykorzystałem algorytm najmniejszej wspólnej wielokrotności, ponieważ zapewni on, każdy wątek dostanie całkowitą liczbę produktów oraz ilość produktów do odebrania będzie równa ilości produktów do wyprodukowania.

3. Implementacja oraz wyniki

```
class Producer extends Thread {
    private final Buffer _buf;
    private final int _iters;

    public Producer(Buffer _buf, int iters) {
        this._buf = _buf;
        this._iters = iters;
    }

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

```
class Consumer extends Thread {
    private final Buffer _buf;
    private final int _iters;

    public Consumer(Buffer _buf, int iters) {
        this._buf = _buf;
        this._iters = iters;
    }

    public void run() {
        for (int i = 0; i < this._iters; ++i) {
            _buf.get();
        }
    }
}

class Buffer {
    private final LinkedList<Integer> _buf = new LinkedList<>();
    private final int MAX_ITEMS_IN_BUFFER = 10;

    public int getProductsNum(){
        return this._buf.size();
    }

    public synchronized void put(int i) {
        while(this._buf.size() >= MAX_ITEMS_IN_BUFFER){
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        this._buf.add(i);
//        System.out.println("Producing " + i);
        this.notifyAll();
    }

    public synchronized int get() {
        while(this._buf.isEmpty()){
            try {
                this.wait();
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
        }

        int v = this._buf.removeFirst();
//        System.out.println("Consuming " + v);
        this.notifyAll();
        return v;
    }
}
```

Kod testujący podpunkty a oraz b:

```
public class PKmon {
    public static void main(String[] args) {
        runCase(1, 1); // n1 = n2 = 1
        runCase(5, 2); // n1 > n2
        runCase(5, 5); // n1 = n2 = 5
        runCase(2, 5); // n1 < n2
    }

    public static void runCase(int producersNum, int consumersNum){
        System.out.println(
                "Producers: " + producersNum +
                " | Consumers: " + consumersNum
        );
        var lcm = lcm(producersNum, consumersNum);
        var products = lcm * 10;
        var producerIterations = products / producersNum;
        var consumerIterations = products / consumersNum;

        System.out.println("Each producer will produce "
                + producerIterations + " products");
        System.out.println("Each consumer will consume "
                + consumerIterations + " products");

        var buffer = new Buffer();
        var threadList = new ArrayList<Thread>();
        IntStream.range(0, producersNum)
                .forEach(i -> threadList.add(
                        new Producer(buffer, producerIterations)
                ));
```

```
        IntStream.range(0, consumersNum)
                .forEach(i -> threadList.add(
                        new Consumer(buffer, consumerIterations)
                ));
        threadList.forEach(Thread::start);

        threadList.forEach(t -> {
            try{
                t.join();
            }catch (InterruptedException e){
                e.printStackTrace();
            }
        });

        System.out.println("All threads finished!");
        System.out.println("Number of products in queue: " + buffer.getProductsNum());
        System.out.println("");
    }

    private static int lcm(int a, int b) {
        return a * (b / gcd(a, b));
    }

    private static int gcd(int a, int b) {
        while (b > 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
}
```

Wyniki uruchomienia programu wyglądają następująco:

```
Producers: 1 | Consumers: 1
Each producer will produce 10 products
Each consumer will consume 10 products
All threads finished!
Number of products in queue: 0

Producers: 5 | Consumers: 2
```

```
Each producer will produce 20 products
Each consumer will consume 50 products
All threads finished!
Number of products in queue: 0

Producers: 5 | Consumers: 5
Each producer will produce 10 products
Each consumer will consume 10 products
All threads finished!
Number of products in queue: 0

Producers: 2 | Consumers: 5
Each producer will produce 50 products
Each consumer will consume 20 products
All threads finished!
Number of products in queue: 0
```

Wyniki pokazują, że program działa poprawnie. Wszystkie produkty zostały przetworzone oraz nie doszło do zakleszczeń ani wyścigów. Poniżej zaprezentowano akcje jakie wykonują poszczególne wątki (wariant a - jeden konsument oraz producent):

```
Producing 0
Producing 1
Producing 2
Producing 3
Producing 4
Producing 5
Producing 6
Producing 7
Producing 8
Producing 9
Consuming 0
Consuming 1
Consuming 2
Consuming 3
Consuming 4
Consuming 5
Consuming 6
Consuming 7
Consuming 8
Consuming 9
```

Możemy zauważyć pewną tendencję dotyczącą zajmowania monitora - wątek zwalniający monitor powiadamia wątki czekające, ale nie zdążają one zająć monitora, ponieważ wątek powiadamiający już go zajął - stąd wynikają serie produkcji oraz konsumpcji.

Podpunkt c wymaga modyfikacji klas konsumenta oraz producenta tak, aby zasymulować pracę tych wątków. W tym celu do metod *run()* zostały dodane wywołania metod *sleep()* z losowym czasem. Zmiany wyglądają następująco:

```
class Producer extends Thread {
    private final Buffer _buf;
    private final int _iters;
    private final Random random = new Random();

    public Producer(Buffer _buf, int iters) {
        this._buf = _buf;
        this._iters = iters;
    }

    public void run() {
        for (int i = 0; i < this._iters; ++i) {
            _buf.put(i);
            try {
                Thread.sleep(random.nextInt(500));
            } catch (InterruptedException ignored) {

            }
        }
    }
}

class Consumer extends Thread {
    private final Buffer _buf;
    private final int _iters;
    private final Random random = new Random();

    public Consumer(Buffer _buf, int iters) {
        this._buf = _buf;
        this._iters = iters;
    }

    public void run() {
        for (int i = 0; i < this._iters; ++i) {
```

```
                _buf.get();
                try {
                    Thread.sleep(random.nextInt(500) + 100);
                } catch (InterruptedException ignored) {

                }
            }
        }
    }
```

Wykonanie programu trwa dłużej, ale wyniki pozostały bez zmian - bufor jest pusty na końcu oraz nie dochodzi do żadnych błędów. Zmiany są widoczne, gdy zostaną wypisane poszczególne operacje:

```
Producing 0
Consuming 0
Producing 0
Consuming 0
Producing 1
Consuming 1
Producing 1
Consuming 1
Producing 2
Producing 3
Producing 2
Consuming 2
Producing 4
Consuming 3
Consuming 2
Consuming 4
Producing 3
Producing 5
Producing 6
Consuming 3
Producing 4
Consuming 5
Producing 7
Producing 5
Consuming 6
Producing 6
Consuming 4
Consuming 7
```

```
Producing 7
Producing 8
Producing 9
Producing 8
Consuming 5
Consuming 6
Consuming 7
Producing 9
Consuming 8
Consuming 9
Consuming 8
Consuming 9
```

Jak można zauważyć program działa bardziej "naturalnie" - konsumenci pobierają produkt zaraz po dodaniu go do bufora.

4. Wnioski

Wbudowane w Jave monitory pozwalają osiągnąc efekt zmiennych warunkowych. Używanie ich jest stosunkowo proste i pozwala na budowanie mechanizmów zapewniających obsługę współdzielonych zasobów.

5. Zadanie 2

Dany jest bufor, do którego producent może wkładać dane, a konsument pobierać. Napisać program, który zorganizuje takie działanie producenta i konsumenta, w którym zapewniona będzie własność bezpieczeństwa i żywotności.

Zrealizować program przy pomocy operacji P()/V() dla semafora:

a. dla przypadku 1 producent / 1 konsument (n1 = n2 = 1)

b. dla przypadku n1 producentów/n2 konsumentów (n1 > 1, n2 > 1)

6. Koncept rozwiązania

Rozwiązanie będzie bazować na kodzie z poprzedniego zadania, zmieniona zostanie jedynie implementacja klasy *Buffer* tak, aby korzystała z dwóch semaforów zliczających oraz jednego semafora binarnego.

- Pierwszy semafor zliczający będzie informował o tym, ile jest obecnie produktów w buforze. Producent będzie podnosił ten semafor za każdym razem jak doda produkt, a konsument będzie go opuszczał po przetworzeniu produktu
- Drugi semafor zliczający będzie informował o tym, ile pozostało jeszcze miejsca w buforze. Konsument będzie podnosił semafor, a producent opuszczał po dodaniu produktu
- Semafor binarny będzie używany do wzajemnego wykluczania dostępu do struktury bufora

Kod testujący został zmieniony tak, aby korzystać z nowej klasy bufora *Buffer2*.

7. Implementacja oraz wyniki

```
class Buffer2 {
    private final LinkedList<Integer> _buf = new LinkedList<>();
    private final CountingSemaphore notEmpty;
    private final CountingSemaphore slotsAvailable;
    private final BinarySemaphore mutex;
    private final int MAX_ITEMS_IN_BUFFER = 10;

    public Buffer2() {
        this.slotsAvailable = new CountingSemaphore(MAX_ITEMS_IN_BUFFER);
        this.notEmpty = new CountingSemaphore(0);
        this.mutex = new BinarySemaphore(true);
    }
```

```
    public int getProductsNum(){
        return this._buf.size();
    }

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

        this.mutex.P();
        this._buf.add(i);
        this.mutex.V();

//        System.out.println("Producing " + i);
        this.notEmpty.V();
    }

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

        this.mutex.P();
        int v = this._buf.removeFirst();
        this.mutex.V();
//        System.out.println("Consuming " + v);

        this.slotsAvailable.V();
        return v;
    }
}
```

Kod testujący:

```
    public static void main(String[] args) {
        runCase(1, 1); // n1 = n2 = 1
        runCase(5, 2); // n1, n2 > 1
        runCase(2, 5); // n1, n2 > 1
    }
```

Wyniki działania testów:

```
Producers: 1 | Consumers: 1
Each producer will produce 10 products
```

```
Each consumer will consume 10 products
All threads finished!
Number of products in queue: 0

Producers: 5 | Consumers: 2
Each producer will produce 20 products
Each consumer will consume 50 products
All threads finished!
Number of products in queue: 0

Producers: 2 | Consumers: 5
Each producer will produce 50 products
Each consumer will consume 20 products
All threads finished!
Number of products in queue: 0
```

---

8. Wnioski

Przedstawione wyniki ukazują, że wynik działania programu jest identyczny, jak w wersji z metodami *wait()* i *notify()*. Podczas działania programu nie doszło do sytuacji wyścigu. Używanie semaforów jest znacznie prostsze niż zmiennych warunkowych zbudowanych na wyżej wymienionych metodach.

9. Zadanie 3

Zaimplementowac rozwiazanie przetwarzania potokowego (Przykladowe zalozenia: bufor rozmiaru 100, 1 producent, 1 konsument, 5 uszeregowanych procesow przetwarzajacych.) Od czego zalezy predkosc obrobki w tym systemie? Rozwiązanie za pomocą semaforów lub monitorów (dowolnie).

10. Koncept rozwiązania

Rozwiązanie sprowadza się do utworzenia $n + 1$ semaforów zliczających (gdzie n to liczba wątków przetwarzających). Na początku semafory będą opuszczone. Gdy producent doda produkt do bufora podniesie on semafor dla pierwszego wątku przetwarzającego i będzie kontynuował dalsze przetwarzanie. Wątek pierwszy opuści swój semafor i przetworzy produkt, a następnie podniesie semafor następnego wątku itd. Klasy producenta, konsumenta oraz przetwarzacza symulują pracę poprzez wywołanie metody *sleep()*.

Potok operacji użyty w zadaniu:

- Producent dodaje do bufora kolejne liczby
- Liczba zostaje przemnożona przez dwa
- Do liczby zostaje dodana wartość 7
- Liczba zostaje zamieniona na napis, a napis zostaje powielony
- Do napisu zostaje dodany prefiks
- Do napisu zostaje dodany sufiks
- Konsument pobiera finalną wartość (w tym przypadku napis)

11. Implementacja oraz wyniki

W zadaniu użyłem implementacji semafora, którą dostarcza Java. Implementacja została zrealizowana w taki sposób, aby była łatwo rozszerzalna oraz dodanie nowej funkcji przekształcającej było bardzo proste. Kod testujący polega na porównaniu wartości pobranych przez wątek konsumeta oraz wartości wyliczone bezpośrednio (nie przez wątki).

```
class Buffer3 {
    private final Object[] buffer;
    private final int bufferSize;
    private final Semaphore[] semaphores;
```

```
private final int transformOperationsNum;

public Buffer3(int bufSize, int transformThreads) {
    buffer = new Object[bufSize];
    bufferSize = bufSize;
    transformOperationsNum = transformThreads + 1; // because of consumer
    semaphores = new Semaphore[transformOperationsNum];
    IntStream.range(0, transformOperationsNum).forEach(i -> {
        semaphores[i] = new Semaphore(0);
    });
}

public int getBufferSize() {
    return bufferSize;
}

public void put(int i) {
    buffer[i] = i;
    semaphores[0].release();
}

public void transform(int i,
                      int operationIndex,
                      Function<Object, Object> transformFunction
) {
    try {
        semaphores[operationIndex].acquire();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    buffer[i] = transformFunction.apply(buffer[i]);

    semaphores[operationIndex + 1].release();
}

public Object get(int i) {
    try {
        semaphores[transformOperationsNum - 1].acquire();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

```
        return buffer[i];
    }
}

class PipeThread extends Thread {
    private final Function<Object, Object> transformFunction;
    private final Buffer3 _buf;
    private final Random random = new Random();
    private final int operationID;

    public PipeThread(Buffer3 buffer,
                      int operationIndex,
                      Function<Object, Object> func)
    {
        transformFunction = func;
        _buf = buffer;
        operationID = operationIndex;
    }

    @Override
    public void run() {
        IntStream.range(0, _buf.getBufferSize()).forEach(i -> {
            try {
                Thread.sleep(random.nextInt(300));
            } catch (InterruptedException ignored) {
            }

            _buf.transform(i, operationID, transformFunction);
        });
    }
}

class Producer2 extends Thread {
    private final Buffer3 _buf;
    private final Random random = new Random();

    public Producer2(Buffer3 buffer) {
        _buf = buffer;
    }

    public void run() {
        IntStream.range(0, _buf.getBufferSize()).forEach(i -> {
            try {
```

```
                Thread.sleep(random.nextInt(300));
            } catch (InterruptedException ignored) {
            }

            _buf.put(i);
        });
    }
}

class Consumer2 extends Thread {
    private final Buffer3 _buf;
    private final Random random = new Random();
    private final List<String> results = new LinkedList<>();

    public Consumer2(Buffer3 buffer) {
        _buf = buffer;
    }

    public void run() {
        IntStream.range(0, _buf.getBufferSize()).forEach(i -> {
            try {
                Thread.sleep(random.nextInt(300));
            } catch (InterruptedException ignored) {
            }

            Object obj = _buf.get(i);
//            System.out.println(obj.toString());
            results.add(obj.toString());
        });
    }

    public List<String> getResults() {
        return results;
    }
}

public class Pipe {
    public static void main(String[] args) {
        IntStream.range(0, 10).forEach(i -> runCase());
    }

    public static void runCase(){
        var BUFFER_SIZE = 100;
```

```
Function<Integer, Integer> multByTwo = i -> (2 * i);
Function<Integer, Integer> addSeven = i -> (i + 7);
Function<Integer, String> duplicateString = i -> (i.toString() + i.toString());
Function<String, String> addPrefix = s -> ("p" + s);
Function<String, String> addSufix = s -> (s + "s");

List<Function> transforms = Arrays.asList(
        multByTwo,
        addSeven,
        duplicateString,
        addPrefix,
        addSufix
);

var buffer = new Buffer3(BUFFER_SIZE, transforms.size());

List<Thread> threadList = new LinkedList<>();
threadList.add(new Producer2(buffer));
IntStream.range(0, transforms.size()).forEach(i -> {
    threadList.add(new PipeThread(buffer, i, transforms.get(i)));
});
var consumer = new Consumer2(buffer);
threadList.add(consumer);

threadList.forEach(Thread::start);
threadList.forEach(t -> {
    try {
        t.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});

var calculatedResults = consumer.getResults();
var expectedResults =
        IntStream
                .range(0, BUFFER_SIZE)
                .mapToObj(i -> {
                    List<Object> tempList = new ArrayList<>(transforms.size());
                    tempList.add(i);
                    transforms.forEach(t ->{
                        Object o = tempList.get(tempList.size() - 1);
```

```
                                tempList.add(t.apply(o));
                            });

                            return tempList.get(tempList.size() - 1).toString();
                        })
                        .collect(Collectors.toList());

        Collections.sort(calculatedResults);
        Collections.sort(expectedResults);
        var res = calculatedResults.equals(expectedResults);
        System.out.println(res ? "Equals" : "Not equals");
    }
}
```

Wynik wykonania programu:

```
Equals
Equals
Equals
Equals
Equals
Equals
Equals
Equals
Equals
Equals
```

Powyższy listing pokazuje, że potok działa poprawnie.

12. Wnioski

Przeprowadzone eksperymenty ukazują, że przetwarzanie potokowe jest potężnym narzędziem pozwalającym rozdzielać odpowiedzialności do osobnych jednostek wykonawczych, dzięki czemu zarządzanie nimi znacznie się upraszcza. Odpowiadając na postawione w zadaniu pytanie - "Od czego zalezy predkosc obrobki w tym systemie?", wnioskuję, że prędkość przetwarzania zależy od każdego z wątku. Każdy wątek może stać się tzw. *wąskim gardłem*, ponieważ jeśli wszystkie wątki wykonują swoje operacje błyskawicznie, ale jeden z nich wykonuje bardzo intensywne obliczenia to cały łańcuch przetwarzania będzie spowolniony. Jest to niewątpliwa wada takiego rozwiązania.