Laboratorium VII - Teoria współbieżności

Weronika Szybińska, 05.12.2022

Treść zadania:

1 . Zaimplementować bufor jako aktywny obiekt (Producenci-Konsumenci)

Rozwiązanie:

Do rozwiązania tego zadnia zaimplementowanych zostało 9 klas: ActiveObject, Buffer, Servant, AddRequest, RemoveRequest, Scheduler, Future, Consumer oraz Producer. Dodatkowo stworzone zostały 2 interfejsy: IMethodRequest oraz Proxy. Są one implementowane przez klasy kolejno AddRequest i RemoveRequest oraz przez Servant. Na samym początku tworzony jest obiekt klasy ActiveObject który jako argument otrzymuje wielkość Bufora. Następnie w obiekcie tym tworzone są obiekty: Bufor o odpowiednim rozmiarze, Scheduler który odpowiada za kolejność wykonywania zadań na buforze oraz Servant, który dostaje utworzony wcześniej bufor oraz scheduler. Servant odpowiada za dodawanie do Schedulera nowych zapytań. W głownej funkcji tworzonych jest również N obiektów klasy konsument oraz M obiektów klasy Producer, które w konstruktorze dostają utworzony w ActiveObject , obiekt Proxy (Servant). Dzięki obiektowi Proxy mogą oni dodawać do kolejki Schedulera prośby o dodanie lub usunięcie obiektu z Bufora. Servant po wywołaniu metody add lub remove, tworzy odpowiedniu obiekt klasy AddRequest lub RemoveRequest, który dodawany jest do kolejki Schedulera. W przypadku metody RemoveRequest zwracany jest obiekt Future, w który opakowany jest oryginalny obiekt przekazany przez producenta. Obiekty Remove oraz Add Request mają dodatkowo zaimplementowane metody quard(), sprawdzające możliwość wykonania zadanej akcji. Scheduler wyciągając kolejny obiekt IMethodRequest z listy wywołuje metode guard() i w przypadku zwrócenia przez nią False, dodaje obiekt z powrotem na koniec kolejki przez co żadne obiekty nie są gubione. Poniżej przedstawiona jest implementacja opisana powyżej.

```
6   class ActiveObject {
7       private Buffer buffer;
8       private Scheduler scheduler;
9       private Proxy proxy;
10
11      public ActiveObject(int queueSize){
12          buffer = new Buffer(queueSize);
13          scheduler = new Scheduler();
14          proxy = new Servant(buffer, scheduler);
15          scheduler.start();
16      }
17
18      public Proxy getProxy() { return this.proxy; }
21  }
```

```java
class Buffer {
    private int bufSize;
    private Queue<Object> buffer;
    public Buffer(int bufSize){
        this.bufSize = bufSize;
        this.buffer = new LinkedList<Object>();
    }
    public void add(Object object) {
        if(!this.isFull()){
            this.buffer.add(object);
        }
    }
    public Object remove() {
        if(this.isEmpty()){
            return null;
        }
        else{
            return buffer.remove();
        }
    }
    public boolean isFull() { return buffer.size() == bufSize; }
    public boolean isEmpty() { return buffer.isEmpty(); }
}
```

f

```java
class Servant implements Proxy{
    Buffer buffer;
    Scheduler scheduler;

    public Servant(Buffer buffer, Scheduler scheduler){
        this.buffer = buffer;
        this.scheduler = scheduler;
    }

    public void add(Object object) { scheduler.addToQueue(new AddRequest(buffer, object)); }

    public Future remove(){
        Future future = new Future();
        scheduler.addToQueue(new RemoveRequest(buffer, future));
        return future;
    }
}
```

```java
interface IMethodRequest {
    boolean guard();
    void execute();
}

class AddRequest implements IMethodRequest{
    private final Buffer buffer;
    private final Object object;

    public AddRequest(Buffer buffer, Object object){
        this.buffer = buffer;
        this.object = object;
    }

    public void execute() { buffer.add(object); }

    public boolean guard() { return !buffer.isFull(); }
}
```

```java
class RemoveRequest implements IMethodRequest{
    private Buffer buffer;
    private Future future;

    public RemoveRequest(Buffer buffer, Future future){
        this.buffer = buffer;
        this.future = future;
    }

    public void execute() { future.setObject(buffer.remove()); }

    public boolean guard() { return !buffer.isEmpty(); }
}
```

```java
class Scheduler extends Thread {
    private Queue<IMethodRequest> activationQueue;

    public Scheduler() { activationQueue = new ConcurrentLinkedQueue<IMethodRequest>(); }

    public void addToQueue(IMethodRequest request) { activationQueue.add(request); }

    public void run() {
        while (true) {
            IMethodRequest iMethodRequest = activationQueue.poll();
            if (iMethodRequest != null) {
                if (iMethodRequest.guard()) {
                    iMethodRequest.execute();
                } else {
                    activationQueue.add(iMethodRequest);
                }
            }
        }
    }
}
```

```java
class Future {
    private Object object;

    public void setObject(Object object) {
        this.object = object;
    }

    public Object getObject() {
        return object;
    }

    public boolean isReady() {
        return object != null;
    }
}
```

```java
class Consumer extends Thread{
    private int id;
    private Proxy proxy;

    public Consumer(int id, Proxy proxy){
        this.id = id;
        this.proxy = proxy;
    }
    public void run(){
        while(true){
            Future result = proxy.remove();
            while(!result.isReady()){
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Consumer " + id  + " ate: " + result.getObject());
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
```

```
189     class Producer extends Thread{
190         private int id;
191         private Proxy proxy;
192         private Random rand;
193 
194         public Producer(int id, Proxy proxy){
195             this.id = id;
196             this.proxy = proxy;
197             rand = new Random();
198         }
199 
200         public void run(){
201             while(true){
202                 int temp = rand.nextInt(100);
203                 proxy.add(temp);
204                 System.out.println("Producer " + id + " added: " + temp);
205                 try {
206                     Thread.sleep(200);
207                 } catch (InterruptedException e) {
208                     e.printStackTrace();
209                 }
210             }
211         }
212     }
```

Wnioski:

Dzięki zaimplementowaniu bufora jako ActiveObject zwiększa się szybkość działania programu, gdyż metody dodawania i usuwania obiektów z bufora są wykonywane w innym wątku niż producenci i konsumenci. Dodatkowo sposób ten zapewnia odseparowanie mechanizmów współbieżnych od właściwej klasy, dzięki czemu jest ona prostsza w budowie.

Bibliografia:

https://www.topcoder.com/thrive/articles/Concurrency%20Patterns%20-%20Active%20Object%20and%20Monitor%20Object

https://en.wikipedia.org/wiki/Active_object