Laboratorium 7

Łukasz Wala
AGH, Wydział Informatyki, Elektroniki i Telekomunikacji
Teoria Współbieżności 2022/23

Kraków, 3 grudnia 2022

1 Treść zadania

1. Pracownik powinien implementować samą kolejkę (bufor) oraz dodatkowe metody (czyPusty etc.), które pomogą w implementacji strażników. W klasie tej powinna być tylko funkcjonalność, ale nie logika związana z synchronizacją.

2. Dla każdej metody aktywnego obiektu powinna być specjalizacja klasy MethodRequest. W tej klasie m.in. zaimplementowana jest metoda guard(), która oblicza spełnienie warunków synchronizacji (korzystając z metod dostarczonych przez Pracownika).

3. Proxy wykonuje się w wątku klienta, który wywołuje metodę. Tworzenie Method request i kolejkowanie jej w Activation queue odbywa się również w wątku klienta. Servant i Scheduler wykonują się w osobnym (oba w tym samym) wątku.

2 Implementacja

```
class BufferFuture {
    private Object object;

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

    public Object getObject(){
        return object;
    }

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

```
    }
}

interface MethodRequest {
    boolean guard();
    void call();
}

class AddRequest implements MethodRequest {
    private BufferServant buffer;
    private BufferFuture future;
    private Object object;

    public AddRequest(BufferServant buffer, BufferFuture future, Object object) {
        this.buffer = buffer;
        this.object = object;
        this.future = future;
    }

    @Override
    public void call() {
        buffer.add(object);
        future.setObject(true);
    }

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

class RemoveRequest implements MethodRequest {
    private BufferServant buffer;
    private BufferFuture future;

    public RemoveRequest(BufferServant buffer, BufferFuture future) {
        this.buffer = buffer;
        this.future = future;
    }

    @Override
    public void call() {
        future.setObject(buffer.remove());
    }

    @Override
```

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

class BufferServant {
    private int bufSize;
    private Queue<Object> buffer;

    public BufferServant(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();
    }
}

class Scheduler extends Thread {
    private Queue<MethodRequest> queue = new ConcurrentLinkedQueue<MethodRequest>();

    public void enqueue(MethodRequest request) {
        queue.add(request);
    }

    public void run() {
        while (true) {
            MethodRequest request = queue.poll();

            if (request != null) {
```

```
                if (request.guard()) request.call();
                else queue.add(request);
            }
        }
    }
}

class Proxy {
    BufferServant buffer;
    Scheduler scheduler;

    public Proxy(BufferServant buffer, Scheduler scheduler){
        this.buffer = buffer;
        this.scheduler = scheduler;
    }

    public BufferFuture add(Object object) {
        BufferFuture future = new BufferFuture();
        scheduler.enqueue(new AddRequest(buffer, future, object));
        return future;
    }

    public BufferFuture remove() {
        BufferFuture future = new BufferFuture();
        scheduler.enqueue(new RemoveRequest(buffer, future));
        return future;
    }
}

class ActiveObject {
    private BufferServant buffer;
    private Scheduler scheduler;
    private Proxy proxy;

    public ActiveObject(int queueSize){
        buffer = new BufferServant(queueSize);
        scheduler = new Scheduler();
        proxy = new Proxy(buffer, scheduler);
        scheduler.start();
    }

    public Proxy getProxy(){
        return this.proxy;
    }
}
```

```
class Producer extends Thread{
    private Proxy proxy;

    public Producer(Proxy proxy){
        this.proxy = proxy;
    }

    public void run(){
        while(true){
            int value = ThreadLocalRandom.current().nextInt(100);
            BufferFuture result = proxy.add(value);

            while(!result.isReady()){
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    System.exit(0);
                }
            }

            System.out.println("Producer " + Thread.currentThread()
            + " added: " + value);

            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                System.exit(0);
            }
        }
    }
}

class Consumer extends Thread{
    private Proxy proxy;

    public Consumer(Proxy proxy){
        this.proxy = proxy;
    }

    public void run(){
        while(true){
            BufferFuture result = proxy.remove();

            while(!result.isReady()){
                try {
                    Thread.sleep(500);
```

```
                } catch (InterruptedException e) {
                    System.exit(0);
                }
            }

            System.out.println("Consumer " + Thread.currentThread()
            + " removed: " + result.getObject());

            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.exit(0);
            }
        }
    }
}

class Main {
    public static void main(String[] args) throws InterruptedException {
        int producerNo = 5;
        int consumerNo = 5;
        int bufferSize = 10;

        ActiveObject activeObject = new ActiveObject(bufferSize);
        Proxy proxy = activeObject.getProxy();

        ExecutorService service = Executors.newFixedThreadPool(producerNo + consumerNo);

        for (int i=0; i<producerNo; ++i) {
            service.submit(new Producer(proxy));
        }

        for (int i=0; i<consumerNo; ++i) {
            service.submit(new Consumer(proxy));
        }

        service.shutdown();
        while (!service.awaitTermination(24L, TimeUnit.HOURS))
            ;
    }
}
```

3 Wnioski

Wzorzec obiektu aktywnego pozwala na skuteczne współbieżne wykonywanie metod obiektu. Oddziela się w nim proces wywołania metody od jej wykonania, co znacznie ułatwia proces zrównoleglania. Dzięki temu też wątek wywołujący może podczas oczekiwania na wynik wykonywać inne czynności. Jako obiekt aktywny można skutecznie zaimplementować np. bufor, który nie steruje dostępem do siebie przez wątki bezpośrednio, robi to "otoczka", m.in. Scheduler, składająca się na obiekt aktywny.

4 Bibliografia

1. Dokumentacja języka Java - docs.oracle.com
2. R.G. Lavender, D.C. Schmidt: *Active Object*