Sprawozdanie nr 4 - Tomasz Szewczyk

Mechanizmy synchronizacji w Java

Problem

Java udostępnia gotowe mechanizmy synchronizacji takie jak między innymi: zmienne atomowe, zamki, zmienne warunkowe czy semafory.

Zadanie

Problem producentów i konsumentów z poprzedniego sprawozdanie należało przepisać za pomocą standardowych mechanizmów sychronizacji dostarczanych razem z jezykiem Java: semaforów oraz zmiennych warunkowych.

```java
package tomaszszewczyk.lab4;

import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

interface AbstractBuffer {
    void clear();

    void put(String x) throws InterruptedException;

    String get() throws InterruptedException;
}

public class lab4 {
    public static void main(String[] args) throws InterruptedException {

        test(new BufferWithConditionals());
        test(new BufferWithSemaphore());
    }

    static private void test(AbstractBuffer myBuffer) throws InterruptedException {
        {
            myBuffer.clear();
            Producer producer = new Producer(myBuffer, 100);
            Consumer consumer = new Consumer(myBuffer, 100);

            System.out.println("p == c == 1");
            producer.start();
            consumer.start();

            consumer.join();
            producer.join();
        }

        {
            myBuffer.clear();
            Producer producer = new Producer(myBuffer, 75);
            Consumer consumer1 = new Consumer(myBuffer, 25);
            Consumer consumer2 = new Consumer(myBuffer, 25);
            Consumer consumer3 = new Consumer(myBuffer, 25);

            System.out.println("p < c");

            producer.start();
            consumer1.start();
            consumer2.start();
            consumer3.start();

            consumer1.join();
            consumer2.join();
            consumer3.join();
            producer.join();
        }

        {
            myBuffer.clear();
            Producer producer1 = new Producer(myBuffer, 25);
            Producer producer2 = new Producer(myBuffer, 25);
            Producer producer3 = new Producer(myBuffer, 25);
            Consumer consumer = new Consumer(myBuffer, 75);

            System.out.println("p > c");

            producer1.start();
            producer2.start();
            producer3.start();
            consumer.start();

            consumer.join();
            producer1.join();
            producer2.join();
            producer3.join();
```

```
        }
    }
}

class Producer extends Thread {

    private AbstractBuffer myBuffer;
    private int count;

    Producer(AbstractBuffer newBuffer, int newCount) {
        myBuffer = newBuffer;
        count = newCount;
    }

    public void run() {
        try {
            for (int i = 0; i < count; i++) {
                myBuffer.put(Integer.toString(i));

                try {
                    sleep(20, 0);
                } catch (Exception e) {
                    System.out.println(e.getMessage());
                }
            }
        } catch (Exception e) {
            System.out.println(e.toString());
        }
    }
}

class Consumer extends Thread {

    private AbstractBuffer myBuffer;
    private int count;

    Consumer(AbstractBuffer newBuffer, int newCount) {
        myBuffer = newBuffer;
        count = newCount;
    }

    public void run() {
        try {
            for (int i = 0; i < count; i++) {
                System.out.println(myBuffer.get());
            }
        } catch (Exception e) {
            System.out.println(e.toString());
        }
    }
}

class BufferWithConditionals implements AbstractBuffer {
    private String[] buf;
    private int start = 0;
    private int stop = 0;
    private Lock lock = new ReentrantLock();
    private Condition notEmpty = lock.newCondition();
    private Condition notFull = lock.newCondition();

    BufferWithConditionals() {
        buf = new String[100];
    }

    public void clear() {
        lock.lock();
        try {
            buf = new String[100];
            start = 0;
            stop = 0;
            notFull.signal();
        } finally {
            lock.unlock();
        }
    }

    @Override
    public void put(String x) throws InterruptedException {
        lock.lock();
        try {
            if (stop == 100)
                notFull.await();
            buf[stop] = x;
            stop += 1;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    @Override
    public String get() throws InterruptedException {
        lock.lock();
        String result;
        try {
            if (start == stop)
                notEmpty.await();
            result = buf[start];
            start += 1;
        } finally {
            lock.unlock();
        }

        return result;
    }
}

class BufferWithSemaphore implements AbstractBuffer {
    private String[] buf;
    private int start = 0;
    private int stop = 0;
    private Semaphore semaphore;

    BufferWithSemaphore() {
        buf = new String[100];
        semaphore = new Semaphore(1);
    }

    public void clear() {
        buf = new String[100];
        start = 0;
        stop = 0;
    }

    @Override
    public void put(String x) throws InterruptedException {
        semaphore.acquire();
        buf[stop] = x;
        stop += 1;
        semaphore.release();
    }

    @Override
    public String get() throws InterruptedException {
        semaphore.acquire();
        while (start == stop) {
            semaphore.release();
            semaphore.acquire();
        }
        String result = buf[start];
        start += 1;
        semaphore.release();
        return result;
    }
}
```