Sprawozdanie nr 3 - Tomasz Szewczyk

Problem ograniczonego bufora

Problem

Częstym wzorcem pojawiającym się przy projektowaniu systemów wielowątkowym jest problem producenta i konsumenta. Architektura programu zakłada N producentów oraz M konsumentów. Producenci tworzą dane które następnie mają zostać przetworzone przez konsumentów. Żadna konkretna porcja danych nie musi trafić do konkretnego konsumenta oraz nie wiadomo w jakim tempie producenci produkują dane, a konsumenci je konsumują. Elementem łączącym producentów i konsumentów może być bufor, który jednak musi być odpowiednio synchronizowany.

Zadanie

Zadanie polegało na zaimplementowaniu bardzo prostego producenta i konsumenta oraz buforów synchronizowanych na dwa sposoby: pierwszy bufor jest synchronizowany za pomocą standardowego mechnizmu monitorów dostępnego w maszynie wirtualnej języka Java, drugi bufor jest synchronizowany napisanym wcześniej semaforem. Następnie przeprowadzono testy prawidłowego działania systemu w sytuacjach równej liczby producentów i konsumentów, większej liczby producentów niż konsumentów oraz mniejszej liczby producentów niż konsumentów.

```
package tomaszszewczyk.lab3;

interface AbstractBuffer {
    void clear();

    void put(String x);

    String get();
}

public class lab3 {
    public static void main(String[] args) throws InterruptedException {

        test(new Buffer());
        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() {
        for (int i = 0; i < count; i++) {
            myBuffer.put(Integer.toString(i));

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

class Consumer extends Thread {

    private AbstractBuffer myBuffer;
    private int count;

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

    public void run() {
        for (int i = 0; i < count; i++) {
            System.out.println(myBuffer.get());
```

```
        }
    }
}

class Buffer implements AbstractBuffer {
    private String[] buf;
    private int start = 0;
    private int stop = 0;
    private BufState state;

    Buffer() {
        buf = new String[100];
        state = BufState.RELEASED;
    }

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

    public synchronized void put(String x) {
        while (state == BufState.ACQUIRED) {
            try {
                wait();
            } catch (InterruptedException ie) {
                System.out.println(ie.getMessage());
            }
        }
        state = BufState.ACQUIRED;

        buf[stop] = x;
        stop += 1;

        state = BufState.RELEASED;
        notify();
    }

    public synchronized String get() {
        while (state == BufState.ACQUIRED || start == stop) {
            try {
                wait();
            } catch (InterruptedException ie) {
                System.out.println(ie.getMessage());
            }
        }
```

```
            state = BufState.ACQUIRED;

            String result = buf[start];
            start += 1;

            state = BufState.RELEASED;
            notify();

            return result;
        }

        private enum BufState {ACQUIRED, RELEASED}
    }

    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();
        }

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

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

        @Override
        public String get() {
            semaphore.acquire();
            while (start == stop) {
                semaphore.release();
                semaphore.acquire();
            }
```

```
        String result = buf[start];
        start += 1;
        semaphore.release();
        return result;
    }
}

class Semaphore {

    private SemState state;
    private int waiting;
    Semaphore() {
        this.state = SemState.RELEASED;
        this.waiting = 0;
    }

    synchronized void acquire() {
        waiting++;
        while (state == SemState.ACQUIRED) {
            try {
                wait();
            } catch (InterruptedException ie) {
                System.out.println(ie.getMessage());
            }
        }
        waiting--;
        state = SemState.ACQUIRED;
    }

    synchronized void release() {
        if (waiting > 0) {
            notify();
        }
        state = SemState.RELEASED;
    }

    private enum SemState {ACQUIRED, RELEASED}
}
```