Análisis y diseño de software

dit
UPM

Tema 3: Concurrencia /clásicos /java

José A. Mañas

11.2.2017

método

1. identifique el estado
   - campos privados del monitor
2. identifique métodos de test and set
   - métodos synchronized
3. programe
   - las condiciones de espera

     ```
     while (condición(...)) wait();
     ```
   - programe las condiciones de notificación

     ```
     if (condición(...)) notifyAll();
     ```

ejemplos

- semaphore
  - synchronized start
  - wait for all
- producer – consumer
- object pool
- readers – writers
- smokers
- philosophers

semaphore

- se trata de implementar un semáforo usando el patrón monitor

```
public class MySemaphore {
    private ...;

    public synchronized void acquire(int n) {
        ...   ...   ...
    }

    public synchronized void release(int n) {
        ...   ...   ...
    }
```

semaphore

- se trata de implementar un semáforo usando el patrón monitor

```
public class MySemaphore {
    private int permits = 0;

    public synchronized void acquire(int n) {
        while (permits < n)
            waiting();
        permits -= n;
    }

    public synchronized void release(int n) {
        permits += n;
        notifyAll();
    }
```

semaphore: synchro start

```
public class SynchStart {
    private static final int N_THREADS = 20;

    public static void main(String[] args)
            throws InterruptedException {
        Semaphore semaphore = new Semaphore(0);
        for (int i = 0; i < N_THREADS; i++) {
            Thread thread = new MyThread(semaphore);
            thread.start();
        }
        System.out.println("wait ...");
        Thread.sleep(5000);
        semaphore.release(N_THREADS);
    }
```

semaphore: synchro start

```
class MyThread
        extends Thread {
    private final Semaphore semaphore;

    MyThread(Semaphore semaphore) {
        this.semaphore = semaphore;
    }

    public void run() {
        try {
            semaphore.acquire(1);
            System.out.println("run");
        } catch (InterruptedException ignored) {
        }
    }
}
```

semaphore: wait for all

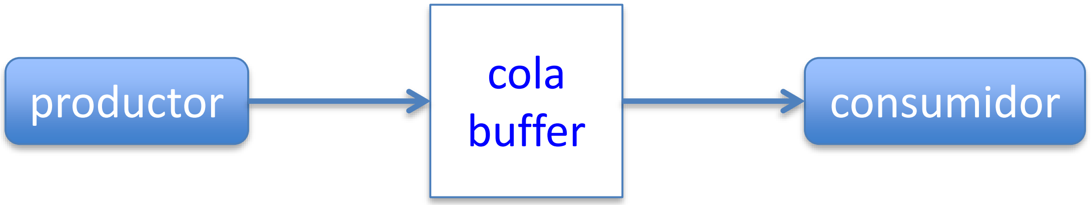
```
public class JoinAll {
    private static final int N_THREADS = 20;

    public static void main(String[] args)
            throws InterruptedException {
        Semaphore semaphore = new Semaphore(0);
        for (int i = 0; i < N_THREADS; i++) {
            Thread thread = new MyThread(semaphore);
            thread.start();
        }
        System.out.println("wait ...");
        semaphore.acquire(N_THREADS);
        System.out.println("... done");
    }
```

semaphore: wait for all

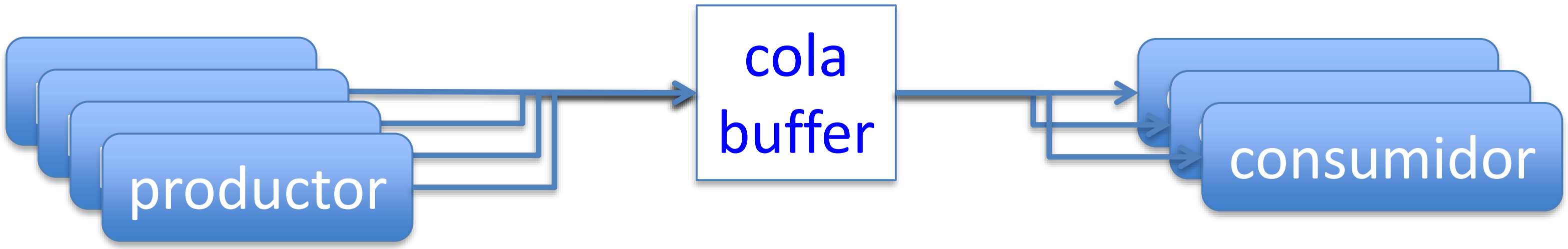
```
class MyThread
        extends Thread {
    private final Semaphore semaphore;

    MyThread(Semaphore semaphore) {
        this.semaphore = semaphore;
    }

    public void run() {
        System.out.println("run");
        semaphore.release(1);
    }
}
```

ejemplo: productor - consumidor

- uno o más *threads* producen datos a su ritmo
- uno o más *threads* consumen datos a su ritmo
- hay que sincronizarlos
  - – parar a los productores cuando hay demasiados datos
    - espera a meter
  - – parar a los consumidores cuando no hay datos
    - espera a sacar



escenario



Buffer versión 1

```
public class Buffer<E> {
    private final int max;
    private final List<E> queue;

    public Buffer1(int max) {
        this.max = max;
        queue = new ArrayList<>(max);
    }
```

```
    public synchronized void put(E s) {
        ...   ...   ...
    }

    public synchronized E take() {
        ...   ...   ...
    }
```

Buffer versión 1

```
public class Buffer<E> {
   private final int max;
   private final List<E> queue;

   public Buffer1(int max) {
      this.max = max;
      queue = new ArrayList<>(max);
   }
```

```
public synchronized void put(E s) {
   while (queue.size() >= this.max)
      waiting();
   queue.add(s);
   notifyAll();
}

public synchronized E take() {
   while (queue.isEmpty())
      waiting();
   notifyAll();
   return queue.remove(0);
}
```

Buffer versión 2

```
public class Buffer<E> {
    private final List<E> queue;

    private final Semaphore haySitio;
    private final Semaphore hayDatos;
    private final Semaphore mutex;

    public Buffer2(int size) {
        queue = new ArrayList<>(size);

        haySitio = new Semaphore(size);
        hayDatos = new Semaphore(0);
        mutex = new Semaphore(1);
    }
```

```
public void put(E s) throws InterruptedException {
        haySitio.acquire();
        mutex.acquire();
        try {
            queue.add(s);
        } finally {
            mutex.release();
            hayDatos.release();
        }
    }

    public E take() throws InterruptedException {
        hayDatos.acquire();
        mutex.acquire();
        try {
            return queue.remove(0);
        } finally {
            mutex.release();
            haySitio.release();
        }
    }
```

errores (fragilidad)

```
public void put(E s) throws InterruptedException {
      mutex.acquire();
      haySitio.acquire();
      try {
         queue.add(s);
      } finally {
         mutex.release();
         hayDatos.release();
      }
   }
```

```
   public E take() throws InterruptedException {
      mutex.acquire();
      hayDatos.acquire();
      try {
         return queue.remove(0);
      } finally {
         mutex.release();
         haySitio.release();
      }
   }
```

java.util.concurrent.BlockingQueue

```
public interface BlockingQueue<E> ... {

   /**
     * Inserts the specified element into this queue, waiting if necessary
     * for space to become available.
    */
    void put(E e) throws InterruptedException;

   /**
     * Retrieves and removes the head of this queue, waiting if necessary
     * until an element becomes available.
     *
     * @return the head of this queue
     * @throws InterruptedException if interrupted while waiting
     */
    E take() throws InterruptedException;
```

object pool

```
public abstract class Pool<E> {
    private final BlockingQueue<E> queue;

    public Pool(int size) {
        this.queue = new ArrayBlockingQueue<E>(size);
        for (int i = 0; i < size; i++)
            queue.add(create());
    }

    protected abstract E create();

    public E get() throws InterruptedException {
        return queue.take();
    }

    public void put(E data) throws InterruptedException {
        queue.put(data);
    }
}
```

readers-writers

- situations in which many threads try to access the same shared resource at one time. Some threads may read and some may write, with the constraint that no process may access the share for either reading or writing, while another process is in the act of writing to it. (In particular, it *is* allowed for two or more readers to access the share at the same time.)

readers-writers

```
public class TestSuite {
    private static final int N_READERS = 10;
    private static final int N_WRITERS = 3;

    public static void main(String[] args) {
        RW_Monitor monitor = new RW_Monitor();
        for (int a = 0; a < N_READERS; a++) {
            ReaderAgent agent = new ReaderAgent(a, monitor);
            agent.start();
        }
        for (int a = 0; a < N_WRITERS; a++) {
            WriterAgent agent = new WriterAgent(a, monitor);
            agent.start();
        }
    }
}
```

readers-writers

```
public class ReaderAgent extends Thread {
    private final int id;
    private final RW_Monitor monitor;

    public ReaderAgent(int id, RW_Monitor
        this.id = id;
        this.monitor = monitor;
    }

    @Override
    public void run() {
        while (true) {
            System.out.println(id + " working");
            monitor.openReading();
            System.out.println(id + " reading");
            nap(1000);
            monitor.closeReading();
        }
    }
```

```
public class WriterAgent extends Thread {
    private final int id;
    private final RW_Monitor monitor;

    public WriterAgent(int id, RW_Monitor monitor) {
        this.id = id;
        this.monitor = monitor;
    }

    @Override
    public void run() {
        while (true) {
            System.out.println(id + " working");
            monitor.openWriting();
            System.out.println(id + " WRITING");
            nap(1000);
            monitor.closeWriting();
        }
    }
```

readers-writers

```
public class RW_Monitor {
   private int nReadersIn = 0;
   private int nWritersIn = 0;
   private int nWritersWaiting = 0;
```

estado que debe ser protegido
consistencia de los 3 campos

- puede haber varios lectores, sin ningún escritor
- puede haber un escritor, sin ningún lector

podemos añadir condiciones como que

- **si hay un escritor esperando, tiene prioridad**
- los escritores entran por orden FIFO
- si los lectores llevan mucho esperando, toman la prioridad
- ...

readers-writers

```
synchronized void openReading() {
     while (nWritersIn > 0 || nWritersWaiting > 0)
        waiting();
     nReadersIn++;
  }

  synchronized void closeReading() {
     nReadersIn--;
     notifyAll();
  }
```

```
synchronized void openWriting() {
     nWritersWaiting++;
     while (nReadersIn > 0 || nWritersIn > 0)
        waiting();
     nWritersWaiting--;
     nWritersIn++;
  }

synchronized void closeWriting() {
     nWritersIn--;
     notifyAll();
  }
```

smokers

- Assume a cigarette requires three ingredients to make and smoke: tobacco, paper, and matches. There are three smokers around a table, each of whom has an infinite supply of *one* of the three ingredients — one smoker has an infinite supply of tobacco, another has paper, and the third has matches.
- There is also a non-smoking agent who enables the smokers to make their cigarettes by arbitrarily (non-deterministically) selecting two of the supplies to place on the table. The smoker who has the third supply should remove the two items from the table, using them (along with their own supply) to make a cigarette, which they smoke for a while. Once the smoker has finished his cigarette, the agent places two new random items on the table. This process continues forever.

smokers

```
public class TestSuite {
    public static void main(String[] args) {
        SmokersMonitor monitor = new SmokersMonitor();
        Smoker smoker1 = new Smoker(1, monitor, PAPER, MATCHES);
        Smoker smoker2 = new Smoker(2, monitor, TOBACCO, MATCHES);
        Smoker smoker3 = new Smoker(3, monitor, TOBACCO, PAPER);
        smoker1.start();
        smoker2.start();
        smoker3.start();
        for (; ; ) {
            monitor.put(Ingredient.random());
            nap(1000);
        }
    }
```

```
public enum Ingredient {
    TOBACCO, PAPER, MATCHES;

    public static Ingredient random() {
        Ingredient[] values = values();
        int random = (int) (Math.random() * values.length);
        return values[random];
    }
```

smokers

```
public class Smoker
        extends Thread {
    private final int id;
    private final SmokersMonitor monitor;
    private final Ingredient need1;
    private final Ingredient need2;

    public Smoker(int id, SmokersMonitor monitor, Ingredient need1, Ingredient need2) {
        this.id = id;
        this.need1 = need1;
        this.need2 = need2;
        this.monitor = monitor;
    }
```

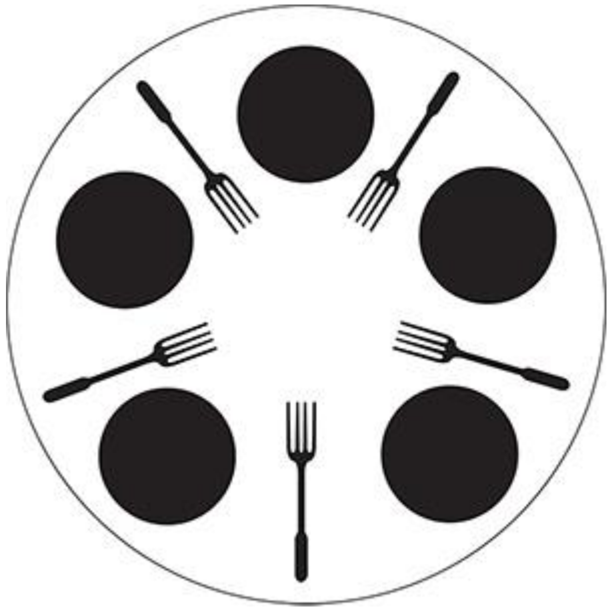
```
    @Override
    public void run() {
        while (true) {
            monitor.need(need1, need2);
            System.out.println(id + " smoking");
        }
    }
```

smokers

```
public class SmokersMonitor {
    private Set<Ingredient> available = new HashSet<>();

    public synchronized void put(Ingredient ingredient) {
        available.add(ingredient);
        notifyAll();
    }

    public synchronized void need(Ingredient need1, Ingredient need2) {
        while (!available.contains(need1) || !available.contains(need2))
            waiting();
        available.remove(need1);
        available.remove(need2);
    }
```

dining philosophers

- Cinco filósofos se sientan alrededor de una mesa y pasan su vida cenando y pensando. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato. Para comer los fideos son necesarios dos tenedores y cada filósofo sólo puede tomar los que están a su izquierda y derecha.
  Si cualquier filósofo toma un tenedor y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda tomar el otro tenedor, para luego empezar a comer.



dining philosophers

```
public enum Fork {
    F1, F2, F3, F4, F5
}
```

```
public class TestSuite {
    public static void main(String[] args) {
        PhilosophersMonitor monitor = new PhilosophersMonitor();
        for (Fork fork : Fork.values())
            monitor.put(fork);

        Philosopher philo1 = new Philosopher(1, monitor, F1, F2);
        Philosopher philo2 = new Philosopher(2, monitor, F2, F3);
        Philosopher philo3 = new Philosopher(3, monitor, F3, F4);
        Philosopher philo4 = new Philosopher(4, monitor, F4, F5);
        Philosopher philo5 = new Philosopher(4, monitor, F5, F1);
        philo1.start();
        philo2.start();
        philo3.start();
        philo4.start();
        philo5.start();
    }
}
```

dining philosophers

```
public class Philosopher extends Thread {
    private final int id;
    private final PhilosophersMonitor monitor;
    private final Fork need1;
    private final Fork need2;

    public Philosopher(int id, PhilosophersMonitor monitor, Fork need1, Fork need2) {
        this.id = id;
        this.need1 = need1;
        this.need2 = need2;
        this.monitor = monitor;
    }
```

```
    @Override
    public void run() {
        while (true) {
            System.out.println(id + " thinking");
            nap(1000);
            monitor.need(need1, need2);
            System.out.println(id + " eating");
            nap(1000);
            monitor.put(need1);
            monitor.put(need2);
        }
    }
```

dining philosophers

```
public class PhilosophersMonitor {
    private Set<Fork> available = new HashSet<>();

    public synchronized void put(Fork fork) {
        available.add(fork);
        notifyAll();
    }

    public synchronized void need(Fork need1, Fork need2) {
        while (!available.contains(need1) || !available.contains(need2))
            waiting();
        available.remove(need1);
        available.remove(need2);
    }
```