# Ejercicios de concurrencia

josé a. mañas

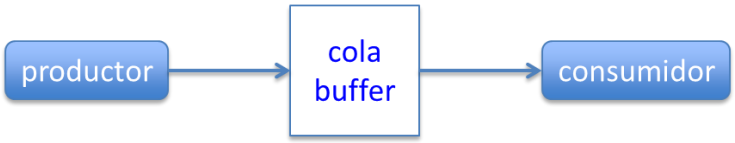
19.4.2018

## Contenido

# 1 Productores y consumidores

Problema clásico de concurrencia.

Uno o más *threads* producen datos a un ritmo diferente que otro u otros *threads* que los consumen. La cola coordina a productores y consumidores, de forma que acompasen su actividad.




Hay múltiples variantes para organizar ese objeto intermedio.

En las soluciones que siguen se presentan diferentes formas de implementar el Buffer, dentro del siguiente esquema de clases

**class Escenario**

```
public class Escenario {
    public static void main(String[] args) {
        Buffer<Character> buffer = new Buffer...<Character>(3);
        Thread productor1 = new Productor(buffer, '0');
        Thread productor2 = new Productor(buffer, 'a');
        Thread consumidor = new Consumidor(buffer);
        productor1.start();
        productor2.start();
        consumidor.start();
    }
}
```

**interface Buffer<E>**

```
public interface Buffer<E> {
    void put(E x)
            throws InterruptedException;

    E get()
            throws InterruptedException;
}
```

**class Productor**

```
import java.util.Random;

public class Productor extends Thread {
    private final Random random = new Random();
    private final Buffer<Character> buffer;
    private final char c;
```

```
    public Productor(Buffer<Character> buffer, char c) {
        this.buffer = buffer;
        this.c = c;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                buffer.put((char) (c + i));
                Thread.sleep(random.nextInt(5) * 1000);
            } catch (InterruptedException ignore) {
            }
        }
    }
}
```

**class Consumidor**

```
import java.util.Random;

public class Consumidor extends Thread {
    private final Random random = new Random();
    private final Buffer<Character> buffer;

    public Consumidor(Buffer<Character> buffer) {
        this.buffer = buffer;
    }

    public void run() {
        while (true) {
            try {
                Character msg = buffer.get();
                Thread.sleep(random.nextInt(2) * 1000);
            } catch (InterruptedException ignore) {
            }
        }
    }
}
```

## 1.1 Solución 1: wait & notify

La clase Buffer se encarga de parar y arrancar al productor según la disponibilidad de espacio para retener datos y al consumidor según haya datos disponibles:

**class Buffer1<E> implements Buffer<E>**

```
public class Buffer1<E>
        implements Buffer<E> {
    private final List<E> data;
    private final int SIZE;

    public Buffer1(int size) {
        this.SIZE = size;
        data = new ArrayList<E>();
    }

    public synchronized void put(E x)
            throws InterruptedException {
        while (data.size() >= SIZE)
            wait();
        data.add(x);
        notifyAll();
    }

    public synchronized E get()
            throws InterruptedException {
        while (data.isEmpty())
            wait();
        notifyAll();
        return data.remove(0);
    }
```

## 1.2 Solución 2: BlockingQueue

La librería de java tiene una clase BlockingQueue que hace exactamente el papel descrito. En este ejemplo, simplemente la envolvemos para ajustar los nombres de los métodos.

**class Buffer4<E> implements Buffer<E>**

```
public class Buffer4<E>
      implements Buffer<E> {
   private final BlockingQueue<E> queue;

   public Buffer4(int size) {
      queue = new LinkedBlockingQueue<E>(size);
   }

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

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

## 1.3 Pruebas

Probar programas concurrentes siempre requiere una dosis de imaginación. En lo que sigue vamos a imponer una serie de aserciones o predicados que deben cumplirse en todo momento y, si no se cumplen, es que algo ha ido mal. Se trata de pruebas de seguridad (safety). Como siempre en temas de pruebas, no pasar una prueba es un fallo detectado, pero pasar todas las pruebas no garantiza la corrección del programa.

**Buffer desbordado**

El tamaño del buffer no puede superar el SIZE máximo.

```
if (data.size() > SIZE)
    throw new IllegalStateException("buffer desbordado");
```

Verificarlo en el método put() parece obvio. Pero verificarlo en el método get() puede detectar alguna irregularidad.

```
    public synchronized E get()
            throws InterruptedException {
        if (data.size() > SIZE)
            throw new IllegalStateException("buffer desbordado");
        while (data.isEmpty())
            wait();
        notifyAll();
        return data.remove(0);
    }
```

Podemos provocar el error para verificar que el test funciona.

El método Math.random() devuelve un número entre 0.0 y 1.0 uniformemente distribuido. De esta forma, la siguiente función devuelve TRUE el x% de las veces

```
      private boolean p(double x) { return 100 * Math.random() < x; }
```

y ahora podemos provocar un desbordamiento el 10% de las veces

```
      public synchronized void put(E x)
              throws InterruptedException {
          if (p(90))
              while (data.size() >= SIZE)
                  wait();
          data.add(x);
          notifyAll();
      }
```

**Salen todos los que entran y solamente los que entran**

Podemos fijar el conjunto de datos que entran, que debe ser exactamente igual al conjunto de datos que salen.

```
public class Test {
    public static void main(String[] args)
            throws InterruptedException {
        Random random = new Random();
        Set<Integer> data = new HashSet<>();
        for (int i = 0; i < 100; i++)
            data.add(random.nextInt(1000));

        Buffer<Integer> buffer = new Buffer1<>(10);

        TestProducer producer = new TestProducer(buffer, data);
        TestConsumer consumer = new TestConsumer(buffer, data);

        producer.start();
        consumer.start();

        producer.join();
        consumer.join();
        System.out.println("test ok");
```

```
    }
}
```

Teniendo unas tareas tuneadas para producir y consumir

```
public class TestProducer
        extends Thread {
    private final Buffer<Integer> buffer;
    private final Set<Integer> data;

    public TestProducer(Buffer<Integer> buffer, Set<Integer> data) {
        this.buffer = buffer;
        this.data = data;
    }

    @Override
    public void run() {
        try {
            for (Integer n : data) {
                buffer.put(n);
                Thread.sleep(100);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("producer ok");
    }
}
```

```
public class TestConsumer extends Thread {
    private final Buffer<Integer> buffer;
    private final Set<Integer> expected;

    public TestConsumer(Buffer<Integer> buffer, Set<Integer> expected) {
        this.buffer = buffer;
        this.expected = expected;
    }

    @Override
    public void run() {
        try {
            Set<Integer> received = new HashSet<>();
            while (received.size() < expected.size()) {
                Integer x = buffer.get();
                if (received.contains(x))
                    throw new IllegalStateException("dato suplicado");
                if (!expected.contains(x))
                    throw new IllegalStateException("dato inesperado");
                received.add(x);
            }
            if (!buffer.isEmpty())
                throw new IllegalStateException("buffer no vacio");
            System.out.println("consumer ok");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

En el buffer hemos añadido una función para testar si está vacío

interface Buffer<E>

```
boolean isEmpty();
```

class Buffer1<E> implements Buffer<E>

```
public synchronized boolean isEmpty() {
    return data.isEmpty();
}
```

class Buffer4<E> implemens Buffer<E>

```
public boolean isEmpty() {
    return queue.isEmpty();
}
```

# 2 Tarea en background

Se trata de lanzar en segundo plano una actividad costosa mientras el programa principal sigue trabajando normalmente. Cuando llegue al punto en que necesita el resultado de la actividad, tendrá que esperar a que esté lista.

```
BgTask1 task = new BgTask1("task");
Thread thread = new Thread(task);
thread.start();

try {
    Nap.sleep(1000, 5000);
    Date date = new Date();
    System.out.println("main: " + date);
    String result = task.getResult();
    System.out.println(result);
} finally {
    task.quit();
}
```

La clase BgTask tiene esta interfaz

```
public class BgTask1
        implements Runnable {

    /**
     * Constructor.
     * @param name nombre de la tarea.
     */
    public BgTask1(String name) {    }

    /**
     * Para terminar.
     */
    public synchronized void quit() {    }

    /**
     * Para conocer el resultado. Espera a que esté listo.
     * @return resultado.
     */
    public synchronized String getResult() {    }

    @Override
    public void run() {    }
}
```

## 2.1 Solución

Usaremos una variable boolean READY para saber si el cálculo ha terminado, bien (con un resultado) o mal (sin él). Y usaremos una variable RESULT para poder devolver el resultado cuando el usuario lo requiera. Por último, usaremos una variable RUNNING para detener el proceso principal abruptamente, antes de que calcule el resultado.

```java
public class BgTask1
        implements Runnable {
    private String name;
    private boolean running;
    private boolean ready = false;
    private String result = null;

    public BgTask1(String name) {
        this.name = name;
    }

    /**
     * Para terminar.
     */
    public synchronized void quit() {
        running = false;
    }

    /**
     * Para conocer el resultado. Espera a que esté listo.
     *
     * @return resultado.
     */
    public synchronized String getResult() {
        while (!ready)
            waiting();
        return result;
    }

    private synchronized void setResult(String s) {
        result = s;
        ready = true;
        notifyAll();
    }

    @Override
    public void run() {
        running = true;
        while (running) {
            Nap.sleep(1000);
            if (Math.random() < .1) {
                setResult(name + ": " + new Date());
                return;
            }
        }
        setResult(null);
    }

    private void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

# 3 Tareas en Segundo plano

Parecido al ejercicio anterior, pero desde el programa principal lanzamos varias tareas y esperamos a que alguna de ellas termine.

Ahora, el programa principal no puede estar esperando por una tarea concreta, sino que recurriremos a un agente intermedio que espera a que alguna tarea anuncie que ha terminado, evento de terminación, avisando a su vez al programa principal.

O sea, visto desde el programa principal

```
List<BgTask2> task2List = new ArrayList<>();
Listener listener = new Listener();

for (int i = 0; i < N_TASKS; i++) {
    BgTask2 task = new BgTask2("task " + i);
    task2List.add(task);
}
for (BgTask2 task : task2List) {
    task.setListener(listener);
    Thread thread = new Thread(task);
    thread.start();
}

try {
    Nap.sleep(2000, 5000);
    Date date = new Date();
    System.out.println("main: " + date);
    String result = listener.getResult();
    System.out.println(result);
} finally {
    for (BgTask2 task : task2List)
        task.quit();
}
```

El agente de escucha tiene esta interfaz

```
public class Listener {

    /**
     * Alguien anuncia un resultado.
     * @param result resultado.
     */
    public synchronized void publish(String result) {    }

    /**
     * Para obtener el resultado.
     * Espera a que haya alguno disponible.
     * @return resultado.
     */
    public synchronized String getResult() {    }
```

Y la tarea en segundo plano se limita a anunciarle al listener que ha terminado:

```
public class BgTask2
        implements Runnable {
    private String name;
    private Listener listener;
    private boolean running;

    public BgTask2(String name) {
        this.name = name;
    }

    /**
     * Para anunciar el resultado cuando esté listo.
     *
     * @param listener agente
     */
    public void setListener(Listener listener) {
        this.listener = listener;
    }

    /**
     * Para terminar.
     */
    public synchronized void quit() {
        running = false;
    }

    private void setResult(String result) {
        if (listener != null)
            listener.publish(result);
    }

    @Override
    public void run() {
        running = true;
        while (running) {
            Nap.sleep(1000);
            if (Math.random() < .05) {
                setResult(name + ": " + new Date());
                return;
            }
        }
    }
}
```

Ejercicio: programe el listener.

## 3.1 Solución

```
public class Listener {
   private boolean ready = false;
   private String result = null;

   public synchronized void publish(String result) {
      this.result = result;
```

```
      ready = true;
      notifyAll();
   }

   public synchronized String getResult() {
      while (!ready)
         waiting();
      return result;
   }

   private void waiting() {
      try { wait(); } catch (InterruptedException ignored) {       }
   }
}
```

# 4 Lectores y escritores (readers and writers)

Escenario clásico donde varios agentes compiten por un recurso común, pero de forma asimétrica. Concretamente, podemos pensar en un dato compartido que algunos quieren leer y otros escribir.

- Cada operación de lectura o escritura debe ser atómica.
- Además, queremos permitir varias lecturas simultáneas;
- pero cuando alguien escribe el acceso debe ser exclusivo: sólo 1 escritor y ningún lector.

Si NR es el número de lectores (readers) y NW es el número de escritores (writers), debe cumplirse siempre este predicado

$$(NR > 0 \Rightarrow NW = 0) \ \& \ (NW > 0 \Rightarrow NR = 0) \ \& \ (NW \leq 1)$$

Usaremos este objeto monitor para organizar la concurrencia

```
public class Monitor {

    // pide permiso para leer; espera a que se le conceda
    public void openReading() {
    }

    // he terminado de leer; que pase otro
    public void closeReading() {
    }

    // pide permiso para escribir; espera a que se le conceda
    public void openWriting() {
    }

    // he terminado de escribir; que pase otro
    public void closeWriting() {
    }
}
```

Variantes

1. solo se admite una thread al tiempo dentro de la zona crítica

2. se admiten varios lectores a la vez

3. los escritores tienen prioridad: si hay un escritor esperando, les lectores no entran

4. los escritores tienen prioridad sobre los lectores, pero además entran respetando el orden de llegada

## 4.1 Soluciones

### 4.1.1 Variante 1: solo una thread

Variables de estado:

**boolean busy = false**

Lleva cuenta de si hay alguien dentro.

```
public class RW_Monitor_1 {
    private boolean busy = false;

    public synchronized void openReading() {
        while (!canRead())
            waiting();
        busy = true;
    }

    private boolean canRead() {
        if (busy) return false;
        return true;
    }

    public synchronized void closeReading() {
        busy = false;
        notifyAll();
    }

    public synchronized void openWriting() {
        while (!canWrite())
            waiting();
        busy = true;
    }

    private boolean canWrite() {
        if (busy) return false;
        return true;
    }

    public synchronized void closeWriting() {
        busy = false;
        notifyAll();
    }

    private synchronized void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

La solución es correcta: no se viola el invariante citado en el enunciado.

### 4.1.2 Variante 2: varios lectores a la vez

Intentamos optimizar el sistema de forma que se puedan realizar varias lecturas a la vez, pero las escrituras sean solitarias.

Variables de estado:

**int nReaders = 0**

Lleva cuenta del número de lectores dentro.

**int nWriters = 0**

Lleva cuenta del número de escritores dentro.

```
public class RW_Monitor_2 {
    private int nReaders = 0;
    private int nWriters = 0;

    public synchronized void openReading() {
        while (!canRead())
            waiting();
        nReaders++;
    }

    private boolean canRead() {
        if (nWriters > 0) return false;
        return true;
    }

    public synchronized void closeReading() {
        nReaders--;
        notifyAll();
    }

    public synchronized void openWriting() {
        while (!canWrite())
            waiting();
        nWriters++;
    }

    private boolean canWrite() {
        if (nReaders > 0) return false;
        if (nWriters > 0) return false;
        return true;
    }

    public synchronized void closeWriting() {
        nWriters--;
        notifyAll();
    }
```

```
    private synchronized void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

La solución es correcta: no se viola el invariante citado en el enunciado.

La solución corre el riesgo de que los lectores monopolicen el monitor y los escritores se queden esperando largo tiempo (*unfair*). De hecho, puede ser que solicitudes de lectura posteriores a una solicitud de escritura se adelanten a la escritura, lo que suena, cuanto menos, raro.

### 4.1.3 Variante 3: los escritores tienen prioridad

Variables de estado:

**int nReaders = 0**

Lleva cuenta del número de lectores dentro.

**int nWriters = 0**

Lleva cuenta del número de escritores dentro.

**int nWaitingWriters = 0**

Lleva cuenta del número de escritores esperando a entrar.

```
public class RW_Monitor_3 {
    private int nReaders = 0;
    private int nWriters = 0;

    private int waitingWriters = 0;

    public synchronized void openReading() {
        while (!canRead())
            waiting();
        nReaders++;
    }

    private boolean canRead() {
        if (nWriters > 0) return false;
        if (waitingWriters > 0) return false;
        return true;
    }

    public synchronized void closeReading() {
        nReaders--;
        notifyAll();
    }
```

```
    public synchronized void openWriting() {
        waitingWriters++;
        while (!canWrite())
            waiting();
        waitingWriters--;
        nWriters++;
    }

    private boolean canWrite() {
        if (nReaders > 0) return false;
        if (nWriters > 0) return false;
        return true;
    }

    public synchronized void closeWriting() {
        nWriters--;
        notifyAll();
    }

    private synchronized void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

La solución es correcta: no se viola el invariante citado en el enunciado.

La solución no corre el riesgo de que los lectores monopolicen el monitor y los escritores se queden esperando largo tiempo (*fair*). En cuanto hay una petición de escritura, las lecturas se detienen (para ser precisos, no se admiten más lectores). Se corre el riesgo de que los escritores monopolicen el monitor y no permitan lecturas (*unfair*).

Por último, cabe mencionar el problema de que las escrituras no ocurran en el orden de llegada, sino que un escritor más reciente se adelanta a un escritor más antiguo.

### 4.1.4 Variante 4: los escritores tienen prioridad y entran en orden

Variables de estado:

**int nReaders = 0**

Lleva cuenta del número de lectores dentro.

**int nWriters = 0**

Lleva cuenta del número de escritores dentro.

**List<Thread> nWaitingWriters = new ArrayList<Thread>**

Lleva cuenta ordenada de los escritores esperando a entrar.

```
public class RW_Monitor_4 {
    private int nReaders = 0;
    private int nWriters = 0;

    private List<Thread> waitingWriters = new ArrayList<>();

    public synchronized void openReading() {
        while (!canRead())
            waiting();
        nReaders++;
    }

    private boolean canRead() {
        if (nWriters > 0) return false;
        if (waitingWriters.size() > 0) return false;
        return true;
    }

    public synchronized void closeReading() {
        nReaders--;
        notifyAll();
    }

    public synchronized void openWriting() {
        Thread me = Thread.currentThread();
        waitingWriters.add(me);
        while (!canWrite(me))
            waiting();
        waitingWriters.remove(me);
        nWriters++;
    }

    private boolean canWrite(Thread me) {
        if (nReaders > 0) return false;
        if (nWriters > 0) return false;
        if (waitingWriters.get(0) != me) return false;
        return true;
    }

    public synchronized void closeWriting() {
        nWriters--;
        notifyAll();
    }

    private synchronized void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

La solución es correcta: no viola el invariante.

La solución prioriza a los escritores frente a los lectores.

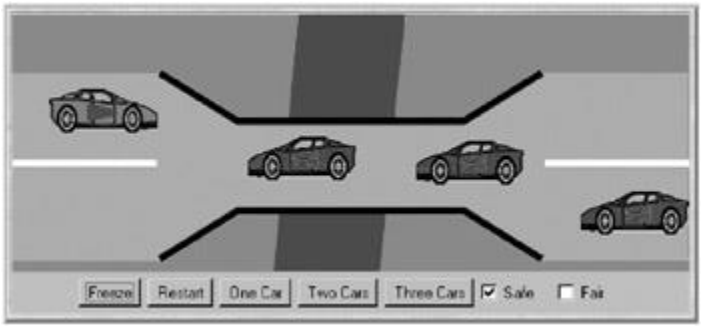
La solución respeta el orden de llegada de los escritores.

Puede ser una solución poco equitativa (*unfair*) en el sentido de que los lectores queden relegados indefinidamente mientras los escritores escriben. Desde el punto de vista de concurrencia es *unfair*. Desde el punto de vista de ingeniería, es un sistema infra dimensionado.

# 5 Carril único (single lane bridge)

Se trata de una carretera de doble dirección que llega a un puente estrecho por el que solamente cabe un coche.

Hay múltiples variantes de este escenario: un estrechamiento, un túnel, una vía de tren compartida, etc.




En general, buscaremos un monitor que controle las entradas y salidas por uno y otro lado:

```
public class NB_Monitor {

    public void entraN() {
    }

    public void entraS() {
    }

    public void saleN() {
    }

    public void saleS() {
    }
}
```

Hay un invariante que debe satisfacerse siempre:

> No hay dos coches circulando simultáneamente en direcciones opuestas.

Versión 1. Solo hay 1 coche circulando.

Versión 2. Se permite que haya varios coches en el puente circulando en la misma dirección.

Versión 3. Hay coches especiales que tienen prioridad. Piense en una ambulancia. Si llega una ambulancia, pasa en cuanto sea físicamente posible; o sea, se detiene el flujo en sentido contrario hasta que haya pasado la ambulancia.

Versión 4. Como la versión 2, pero si hay coches esperando en ambas direcciones, se alternan las entradas (por turnos).

Versión 5. Como en la versión 2, pero se respeta el orden de llegada.

## 5.1 Soluciones

### 5.1.1 Versión 1

Solo se permite 1 coche circulando.

Variables de estado:

**boolean busy = false**

true si hay un coche en el puente

```
public class NB_Monitor_v1 extends NB_Monitor {
    private boolean busy = false;

    public synchronized void entraN() {
        while (busy)
            waiting();
        busy = true;
    }

    public synchronized void entraS() {
        while (busy)
            waiting();
        busy = true;
    }

    public synchronized void saleN() {
        busy = false;
        notifyAll();
    }

    public synchronized void saleS() {
        busy = false;
        notifyAll();
    }

    private synchronized void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

La solución es correcta: se respeta el invariante.

### 5.1.2 Versión 2

Se permite que haya varios coches en el puente circulando en la misma dirección.

Variables de estado:

**int nNS = 0**

número de coches en el puente circulando de Norte a Sur.

**int nSN = 0**

número de coches en el puente circulando de Sur a Norte.

**invariante**

```
private void invariant() {
    if (nNS > 0 && nSN > 0)
        throw new IllegalStateException();
}
```

```
public class NB_Monitor_v2 extends NB_Monitor {
    private int nNS = 0;     // no. viajando de N a S
    private int nSN = 0;     // no. viajando de S a N

    private void invariant() {
        if (nNS > 0 && nSN > 0)
            throw new IllegalStateException();
    }

    public synchronized void entraN() {
        while (nSN > 0)
            waiting();
        nNS++;
        invariant();
    }

    public synchronized void entraS() {
        while (nNS > 0)
            waiting();
        nSN++;
        invariant();
    }

    public synchronized void saleN() {
        nSN--;
        invariant();
        notifyAll();
    }

    public synchronized void saleS() {
        nNS--;
        invariant();
        notifyAll();
    }

    private synchronized void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

La solución es correcta: se respeta el invariante.

La solución puede ser poco equitativa (*unfair*). Si siempre hay coches intentando circular en una dirección, nunca liberan el puente y un coche en dirección contraria puede verse retrasado indefinidamente.

### 5.1.3 Versión 3

Hay coches especiales que tienen prioridad. Piense en una ambulancia. Si llega una ambulancia, pasa en cuanto sea físicamente posible; o sea, se detiene el flujo en sentido contrario hasta que haya pasado la ambulancia.

Variables de estado:

**int nNS = 0**

número de coches en el puente circulando de Norte a Sur.

**int nSN = 0**

número de coches en el puente circulando de Sur a Norte.

**int nwN= 0**

número de ambulancias esperando en la entrada Norte

**int nwS= 0**

número de ambulancias esperando en la entrada Sur

**invariante**

```
private void invariant() {
    if (nNS > 0 && nSN > 0)
        throw new IllegalStateException();
}
```

```
public class NB_Monitor_v3 extends NB_Monitor {
    private int nNS = 0;
    private int nSN = 0;
    private int npN = 0;
    private int npS = 0;

    public synchronized void entraN() {
        while (nSN > 0 || npS > 0)
            waiting();
        nNS++;
    }

    public synchronized void entraPrioritarioN() {
        npN++;
        while (nSN > 0)
            waiting();
        npN--;
        nNS++;
    }
```

```
    public synchronized void entraS() {
        while (nNS > 0 || npN > 0)
            waiting();
        nSN++;
    }

    public synchronized void entraPrioritarioS() {
        npS++;
        while (nNS > 0)
            waiting();
        npS--;
        nSN++;
    }

    public synchronized void saleN() {
        nSN--;
        notifyAll();
    }

    public synchronized void saleS() {
        nNS--;
        notifyAll();
    }

    private synchronized void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

La solución es correcta: se respeta el invariante.

La solución puede ser poco equitativa (*unfair*). Si siempre hay coches intentando circular en una dirección, nunca liberan el puente y un coche en dirección contraria puede verse retrasado indefinidamente.

### 5.1.4 Versión 4

Como la versión 2, pero si hay coches esperando en ambas direcciones, se alternan las entradas (por turnos).

Variables de estado:

**int nNS = 0**

número de coches en el puente circulando de Norte a Sur.

**int nSN = 0**

número de coches en el puente circulando de Sur a Norte.

**int nwN= 0**

número de coches esperando en la entrada Norte

**int nwS= 0**

número de coches esperando en la entrada Sur

**boolean turnoN**

true si toca a los de la entrada norte; false si les toca a los del sur

**invariante**

```
private void invariant() {
    if (nNS > 0 && nSN > 0)
        throw new IllegalStateException();
}
```

```
public class NB_Monitor_v4 extends NB_Monitor {
    private int nNS = 0;
    private int nSN = 0;
    private int nwN = 0;
    private int nwS = 0;
    private boolean turnoN;

    public synchronized void entraN() {
        nwN++;
        while (nSN > 0 || (turnoN == false && nwS > 0))
            waiting();
        nwN--;
        turnoN = false;
        nNS++;
    }

    public synchronized void entraS() {
        nwS++;
        while (nNS > 0 || (turnoN == true && nwN > 0))
            waiting();
        nwS--;
        turnoN = true;
        nSN++;
    }

    public synchronized void saleN() {
        nSN--;
        notifyAll();
    }

    public synchronized void saleS() {
        nNS--;
        notifyAll();
    }

    private synchronized void waiting() {
        try {
```

```
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

La solución es correcta: se respeta el invariante.

La solución es perfectamente equitativa (*fair*). A cambio de esa equidad, corremos el riesgo de utilizar el puente de forma poco eficiente ya que, si hay coches esperando en ambas entradas, se alternan rigurosamente y no se permite que varios coches pasen simultáneamente.

### 5.1.5 Versión 5

Como en la versión 2, pero se respeta el orden de llegada.

Variables de estado:

**int nNS = 0**

número de coches en el puente circulando de Norte a Sur.

**int nSN = 0**

número de coches en el puente circulando de Sur a Norte.

**List<Thread> queue = new ArrayList<>()**

cola FIFO de coches, según van llegando a alguna de las entradas, Norte o Sur

**invariante**

```
private void invariant() {
    if (nNS > 0 && nSN > 0)
        throw new IllegalStateException();
}
```

```
public class NB_Monitor_v5 extends NB_Monitor {
    private int nNS = 0;
    private int nSN = 0;
    private List<Thread> queue = new ArrayList<>();

    public synchronized void entraN() {
        queue.add(Thread.currentThread());
        while (cocheEntraNorte() == false)
            waiting();
        queue.remove(0);
        nNS++;
    }

    private boolean cocheEntraNorte() {
        // requisito inexcusable
        if (nSN > 0) return false;
```

```
        // politica opcional
        Thread me = Thread.currentThread();
        if (!queue.get(0).equals(me)) return false;

        return true;
    }

    public synchronized void entraS() {
        queue.add(Thread.currentThread());
        while (cocheEntraSur() == false)
            waiting();
        queue.remove(0);
        nSN++;
    }

    private boolean cocheEntraSur() {
        // requisito inexcusable
        if (nNS > 0) return false;

        // politica opcional
        Thread me = Thread.currentThread();
        if (!queue.get(0).equals(me)) return false;

        return true;
    }

    public synchronized void saleN() {
        nSN--;
        notifyAll();
    }

    public synchronized void saleS() {
        nNS--;
        notifyAll();
    }

    private synchronized void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

La solución es correcta: se respeta el invariante.

La solución es perfectamente equitativa (*fair*) y eficiente: se permite que varios coches pasen simultáneamente.

# 6 Intercambiador (Exchanger<V>)

Dos agentes A y B se sincronizan en 1 punto. A le pasa un dato a B; B le pasa un dato a A.

```
public class Exchanger<V> {

    public synchronized V exchange(V x) {
      TO DO
    }
}
```

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Exchanger.html

> A synchronization point at which threads can pair and swap elements within pairs. Each thread presents some object on entry to the exchange method, matches with a partner thread, and receives its partner's object on return. An Exchanger may be viewed as a bidirectional form of a SynchronousQueue. Exchangers may be useful in applications such as genetic algorithms and pipeline designs.

**Escenario de prueba**

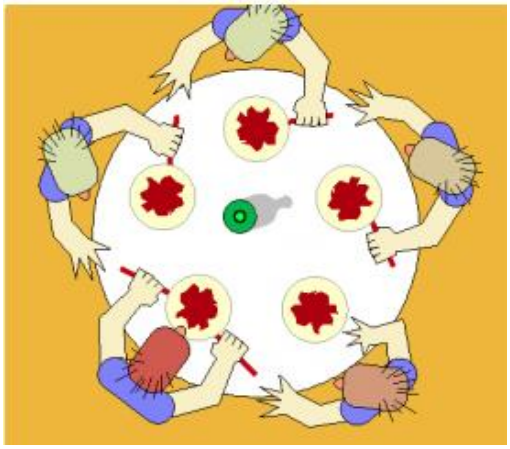
```
public static void main(String[] args) {
    int N = 5;
    Exchanger<Integer> exchanger =
        new Exchanger<Integer>();

    for (int id = 0; id < N; id++) {
        Agente agente = new Agente(id + 1, exchanger);
        agente.start();
    }
}
```

```
class Agente extends Thread {
    private final int id;
    private int n;
    private final Exchanger<Integer> exchanger;

    public Agente(int id, Exchanger<Integer> exchanger) {
        this.id = id;
        this.n = 1000 * id;
        this.exchanger = exchanger;
    }

    @Override
    public void run() {
        while (true) {
            nap();
            int m = exchanger.exchange(this.n);
            System.out.printf(
               "agente %d: recibe %d%n", id, m);
            n++;
        }
    }
}
```

## 6.1 Solución

# 7 Filósofos (philosophers)

Es un problema clásico que se suele plantear en cursos de sistemas operativos, pero que se aplica a situaciones donde unos pocos recursos se comparten entre unas pocas tareas y los recursos se asignan poco a poco llevando a situaciones de bloqueo (*deadlock*).



Se presenta como N filósofos (típicamente, N = 5) que comparten N tenedores. Cada filósofo normalmente está pensando y cuando le entra hambre coge su tenedor izquierdo, luego el derecho, como un poco, devuelve los tenedores y sigue pensando.

Hay un riesgo de interbloqueo: si todos los filósofos cogen su tenedor izquierdo simultáneamente, todos se quedan bloqueados esperando por su tenedor derecho, que está ocupado.

Básicamente, este es el comportamiento de un filósofo:

```
public class Philosopher
        extends Thread {
    private final int id;
    private final Chopstick need1;
    private final Chopstick need2;

    public Philosopher(int id, Chopstick need1, Chopstick need2) {
        this.id = id;
        this.need1 = need1;
        this.need2 = need2;
    }

    @Override
    public void run() {
        while (true) {
            set(THINKING);

            need1.take();
            need2.take();
            set(EATING);
            need1.put();
            need2.put();
        }
    }
}
```

Como la situación de bloqueo es poco probable, podemos forzarla usando una barrera de sincronización de todos los filósofos. Como barrera podemos recurrir a la clase CyclicBarrier de java.util.concurrent, o implementar una barrera sencilla sobre la marcha

```
public class Barrier {
    private final int needed;
    private int ready;

    public Barrier(int needed) {
        this.needed = needed;
        this.ready = 0;
    }

    public synchronized void await() {
        ready++;
        if (ready < needed) {
            waiting();
        } else {
            ready = 0;
            notifyAll();
        }
    }

    private void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

```
@Override
public void run() {
    while (true) {
        ball.set(THINKING);
        Nap.random(1000, 3000);

        barrier.await();
        need1.take();
        Nap.sleep(500);
        need2.take();

        ball.set(EATING);
        Nap.random(1000, 3000);
        need1.put();
        need2.put();
    }
}
```

Es un ejercicio más para pensar que para dar con la solución perfecta.

## 7.1 Soluciones

### 7.1.1 Variante 1

Buscaremos una solución donde cada tenedor es un recurso independiente y cada filósofo los va cogiendo según necesita.

```
public class Chopstick {
    private Lock door = new ReentrantLock();

    public void take() {
        door.lock();
        System.out.println("take " + this);
    }

    public void put() {
        System.out.println("put  " + this);
        door.unlock();
    }
}
```

Es una propuesta que se presta a interbloqueo (*deadlock*).

### 7.1.2 Variante 1 modificada

Se puede evitar el interbloqueo si la reserva de recursos se hace ordenadamente; es decir, si establecemos una relación de orden, cualquiera, entre recursos y los usuarios los reservan en orden, siempre en el mismo orden.

En java, podemos usar la función hash() que siempre devuelve el mismo entero para el mismo objeto y devuelve enteros diferentes para diferentes objetos, y los enteros satisfacen una relación de orden.

```
while (true) {
    set(THINKING);

    barrier.await();
    if (need1.hashCode() < need2.hashCode()) {
        need1.take();
        need2.take();
    } else {
        need2.take();
        need1.take();
    }

    set(EATING);
    need1.put();
    need2.put();
}
```

La solución no impide que haya circunstancias, fortuitas o provocadas, que impidan a un filósofo comer alguna vez y, por tanto, podría sufrir de inanición (*starving*). Esto ocurre si los filósofos adyacentes siempre acaban cogiendo los tenedores y el filósofo intermedio nunca los logra.

### 7.1.3 Variante 1 modificada

Esta es muy específica del problema que nos incumbe. Se trata de poner un portero y limitar el número máximo de toma de recursos. Basta limitar a N-1 el número de filósofos que pueden coger algún tenedor para impedir que se bloqueen entre todos.

Cada recurso sigue auto gestionándose.

Para controlar el número máximo de threads en la zona crítica, la solución más simple es usar un semáforo inicializado a 1 menos que el número de filósofos:

```
Semaphore semaphore = new Semaphore(4);
```

```
while (true) {
    set(THINKING);
    barrier.await();
    try { semaphore.acquire(); } catch (Exception ignored) { }
    need1.take();
    need2.take();
    semaphore.release();
    set(EATING);
    need1.put();
    need2.put();
}
```

La solución impide el interbloqueo (*deadlock*).

La solución no impide que haya circunstancias, fortuitas o provocadas, que impidan a un filósofo comer alguna vez y, por tanto, podría sufrir de inanición (*starving*). Esto ocurre si los filósofos adyacentes siempre acaban cogiendo los tenedores y el filósofo intermedio nunca los logra.

### 7.1.4 Variante 2

Agruparemos la adquisición de recursos en una única transacción, bajo control de un monitor centralizado de los recursos ocupados. O se obtienen todos los recursos necesarios o se espera, evitando la situación de reservas bloqueantes.

```
public class Chopstick {
}
```

```
public class Monitor {
    private Set<Chopstick> busy = new HashSet<>();

    public synchronized void take(Chopstick c1, Chopstick c2) {
        while (busy.contains(c1) || busy.contains(c2))
```

```
            waiting();
        busy.add(c1);
        busy.add(c2);
    }

    public synchronized void put(Chopstick c1, Chopstick c2) {
        busy.remove(c1);
        busy.remove(c2);
        notifyAll();
    }

    private void waiting() {
        try {
            wait();
        } catch (InterruptedException ignored) {
        }
    }
}
```

```
public class Philosopher
        extends Thread {
    private final int id;
    private Monitor monitor;
    private final Chopstick need1;
    private final Chopstick need2;

    public Philosopher(int id, Monitor monitor,
                       Chopstick need1, Chopstick need2) {
        this.id = id;
        this.monitor = monitor;
        this.need1 = need1;
        this.need2 = need2;
    }

    @Override
    public void run() {
        while (true) {
            set(THINKING);
            barrier.await();
            monitor.take(need1, need2);
            ball.set(EATING);
            monitor.put(need1, need2);
        }
    }
}
```

La solución impide el interbloqueo (*deadlock*).

La solución no impide que haya circunstancias, fortuitas o provocadas, que impidan a un filósofo comer alguna vez y, por tanto, podría sufrir de inanición (*starving*). Esto ocurre si los filósofos adyacentes siempre acaban cogiendo los tenedores y el filósofo intermedio nunca los logra.

# 8 Eventos (event-driven broadcasting)

Se trata de difundir simultáneamente un cierto mensaje a todas las threads que estén esperándolo.

```
public class Broadcaster {

    // esperamos a que se publique el mensaje
    public synchronized Object get() {
        return null;
    }

    // el mensaje se public para todos los que esperan
    public synchronized void signal(Object msg) {
    }
}
```

Un escenario de uso podría ser este

```
public static void main(String[] args) {
    Broadcaster broadcaster = new Broadcaster();

    for (int id = 100; id < 150; id++) {
        Thread thread = new Agent(id, broadcaster);
        thread.start();
    }
    Nap.sleep(1000);
    broadcaster.signal("1");

    for (int id = 200; id < 250; id++) {
        Thread thread = new Agent(id, broadcaster);
        thread.start();
    }
    Nap.sleep(1000);
    broadcaster.signal("2");
}
```

```
    class Agent extends Thread {
        private final int id;
        private final Broadcaster broadcaster;

        public Agent(int id, Broadcaster broadcaster) {
            this.id = id;
            this.broadcaster = broadcaster;
        }

        @Override
        public void run() {
            Object msg = broadcaster.get();
            System.err.println(id + ": " + msg + ": " + new Date());
```

```
        }
    }
```

## 8.1 Solución

```
public class Broadcaster {
    private Object msg;

    public synchronized Object get() {
        waiting();
        return msg;
    }

    public synchronized void signal(Object msg) {
        this.msg = msg;
        notifyAll();
    }

    private void waiting() {
        try {
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

# 9 Copia local de un sitio web

Se trata de visitar una dirección web y copiar su contenido en un fichero local. A continuación, miramos si la página tiene hiperenlaces y los descargamos de forma recursiva.

Para realizar esta tarea de descarga usaremos un conjunto de threads (thread pool) y una cola de URLs pendientes, de forma que cada tarea coge una url, la descarga, mete en la cola las url siguientes y vuelve a empezar. Así hasta que no hay más que hacer.

Las tareas tienen que compartir la cola de URL pendientes y deben llevar un registro compartido de las url que se les asignan de forma que no se descargue la misma url 2 veces, ni concurrentemente (sería un caos) ni más tarde (sería un desperdicio).

El programa principal puede tener este aspecto, donde hoy no nos preocupa tanto el manejo de java como la gestión de la información compartida.

```
public class WebCloner {
    private static final int N_ROBOTS = 1;

    public static void main(String[] args)
            throws MalformedURLException {
        URL webRoot = new URL(
            "http://www.dit.upm.es/~pepe/doc/adsw/ejercicio4/");
        URL seed = new URL(webRoot, "index.html");
        File localRoot = new File("C:/tmpp/kk");

        Queue<URL> queue = new Queue<>();
        queue.put(seed);

        Registry<String> registry = new Registry<>();

        for (int i = 0; i < N_ROBOTS; i++) {
            Worker worker = new Worker(localRoot, webRoot,
                                       queue, registry);
            worker.start();
        }
    }
}
```

La clase auxiliar que se encarga del trabajo puntual

```
public class Worker extends Thread {
    private File localRoot;
    private File webRootPath;
    private final Queue<URL> queue;
    private final Registry<String> registry;

    public Worker(File localRoot, URL webRoot,
            Queue<URL> queue, Registry<String> registry) {
        this.localRoot = localRoot;
        this.webRootPath = getLocalPath(webRoot);
```

```
        this.queue = queue;
        this.registry = registry;
    }
```

```
    public void run() {
        while (true) {
            URL next = queue.take();
            if (next == null)
                break;
            download(next);
            queue.done();
        }
    }
```

```
    public void download(URL url) {
        try {
            String protocol = url.getProtocol();
            if (!(protocol.equalsIgnoreCase("http") ||
                  protocol.equalsIgnoreCase("https")))
                return;
            File dst = getLocalPath(url);
            if (!isSubDirectory(webRootPath, dst))
                return;
            if (!registry.isNew(dst.getCanonicalPath()))
                return;
            dst.getParentFile().mkdirs();

            FileOutputStream fos = new FileOutputStream(dst);
            ReadableByteChannel rbc =
Channels.newChannel(url.openStream());
            fos.getChannel().transferFrom(rbc, 0, Long.MAX_VALUE);
            fos.close();

            URLConnection connection = url.openConnection();
            String content = connection.getContentType();
            if (!content.equalsIgnoreCase("text/html"))
                return;

            FileReader reader = new FileReader(dst);
            StringBuilder htmlPage = new StringBuilder();
            char[] buffer = new char[4 * 1024];
            while (true) {
                int n = reader.read(buffer);
                if (n < 0)
                    break;
                htmlPage.append(buffer, 0, n);
            }

            for (String link : parseHref(htmlPage.toString()))
                queue.put(new URL(url, link));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
    private static ArrayList<String> parseHref(String HTMLPage) {
        Pattern linkPattern =
          Pattern.compile(
                "<a[^>]+href=[\"']?([^\"'>]+)[\"']?[^>]*>(.+?)</a>",
                Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
        Matcher pageMatcher = linkPattern.matcher(HTMLPage);
        ArrayList<String> links = new ArrayList<String>();
        while (pageMatcher.find())
            links.add(pageMatcher.group(1));
        return links;
    }
```

```
    private File getLocalPath(URL url) {
        File dst0 = new File(localRoot, url.getHost());
        return new File(dst0, url.getPath());
    }
```

```
    private boolean isSubDirectory(File base, File child)
            throws IOException {
        base = base.getCanonicalFile();
        child = child.getCanonicalFile();

        File parent = child;
        while (parent != null) {
            if (base.equals(parent))
                return true;
            parent = parent.getParentFile();
        }
        return false;
    }
```

## 9.1 Información compartida – Zonas críticas

```
/**
 * Cola de tareas a realizar.
 * No tiene tamano maximo.
 * Lleva cuenta de si hay threads que se han llevado tarea
 * y aun no han terminado (pueden encontrar nueva tarea.
 * @param <E> de un cierto tipo.
 */
public class Queue<E> {
    private List<E> queue = new ArrayList<>();
    private int busy= 0;
```

```
    /**
     * Mete una tarea pendiente de tipo E.
     * @param e
     */
```

```
    public void put(E e) {

    }
```

```
    /**
     * Saca la primera tarea pendiente.
     * Espera si no hay nada en la cola y hay tareas trabajando.
     * @return siguiente tarea.
     */
    public E take() {

    }
```

```
    /**
     * Una tarea avisa de que ha terminado
     * y ya no va a encolar mas cosas.
     */
    public void done() {

    }
```

```
/**
 * Registro de tareas que se le han enconmendado a algun thread.
 * @param <T> Tipo de tarea.
 */
public class Registry<T> {
    private Set<T> set = new HashSet<>();

    /**
     * Cheque si alguien se esta haciendo o se ha hecho cago de algo.
     * @param t objeto a contrastar.
     * @return true si nadie se lo ha adjudicado antes.
     */
    public boolean isNew(T t) {

    }
}
```

## 9.2 Solución

### 9.2.1 Queue<E> - Cola de tareas pendientes

```
/**
 * Cola de tareas a realizar.
 * No tiene tamano maximo.
```

```java
 * Lleva cuenta de si hay threads que se han llevado tarea y aun no han
terminado (pueden encontrar nueva tarea.
 * @param <E> de un cierto tipo.
 */
public class Queue<E> {
    private List<E> queue = new ArrayList<>();
    private int busy= 0;
```

```java
    /**
     * Mete una tarea pendiente de tipo E.
     * @param e
     */
    public synchronized void put(E e) {
        System.out.println("put " + e);
        queue.add(e);
        notifyAll();
    }
```

```java
    /**
     * Saca la primera tarea pendiente.
     * Espera si no hay nada en la cola y hay tareas trabajando.
     * @return siguiente tarea.
     */
    public synchronized E take() {
        while (queue.isEmpty() && busy > 0)
            waiting();
        if (queue.isEmpty())
            return null;
        busy++;
        System.out.println("take " + queue.get(0));
        return queue.remove(0);
    }
```

```java
    /**
     * Una tarea avisa de que ha terminado
     * y ya no va a encolar mas cosas.
     */
    public synchronized void done() {
        busy--;
        notifyAll();
    }
```

### 9.2.2 Registry<T> - Registro de tareas asignadas

```java

/**
 * Registro de tareas que se le han enconmendado a algun thread.
 * @param <T> Tipo de tarea.
 */
public class Registry<T> {
```

```
    private Set<T> set = new HashSet<>();

    /**
     * Cheque si alguien se esta haciendo o se ha hecho cago de algo.
     * @param t objeto a contrastar.
     * @return true si nadie se lo ha adjudicado antes.
     */
    public synchronized boolean isNew(T t) {
        if (set.contains(t))
            return false;
        set.add(t);
        return true;
    }
}
```