Asignatura 780014
Programación Avanzada

TEMA 3 – CONCURRENCIA DE MEMORIA COMÚN

Concurrencia de memoria común

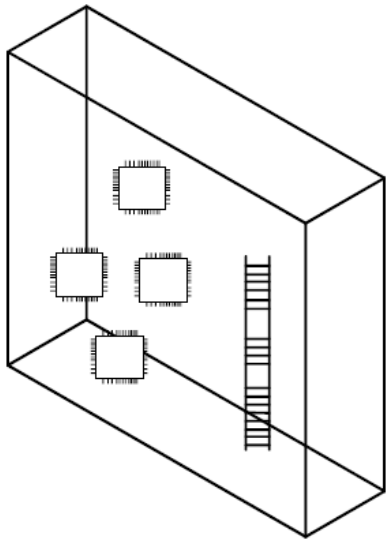
- Objetivo del tema:
  - Introducir los problemas y soluciones básicos de la implementación de concurrencia en sistemas de memoria común

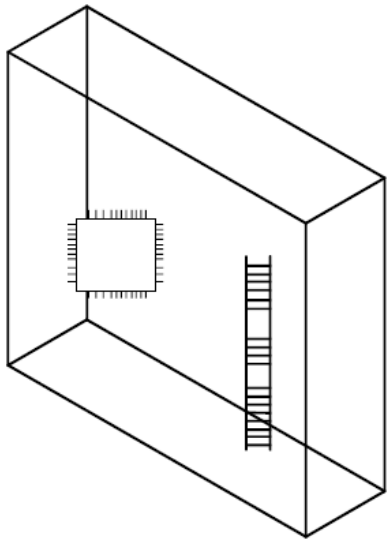
Índice

1. Concurrencia de memoria común
2. Coordinación
3. Concurrencia y Java
4. Algoritmos para Exclusión Mutua
5. Soluciones no algorítmicas
   - Locks y Conditions

Concurrencia de memoria común

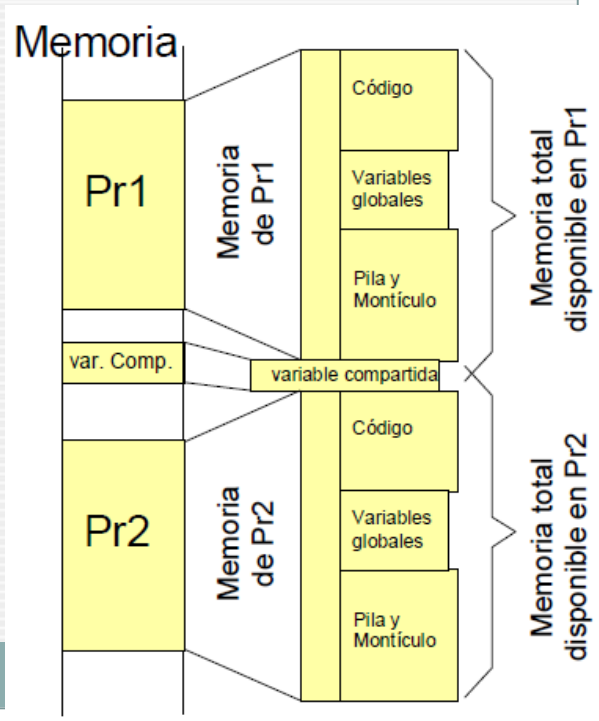
- Programación concurrente de memoria común:
  - Cuando tenemos una **única memoria** con:
    - El Sistema Operativo (SO)
    - Todos los procesos
    - Todos los datos
    - Todos los recursos
  - Podemos tener concurrencia
    - Con 1 procesador
    - Con n procesadores





Concurrencia de memoria común

- Cada proceso tiene su propio espacio de direcciones
  - Lo asigna y protege el SO
  - Sin acceso al de otros procesos
  - El SO ofrece zonas de memoria compartida
    - Variables compartidas
    - Utilizadas para la coordinación
      - Comunicación
      - Sincronización
  - Concurrencia a nivel de SO que permite explicar el funcionamiento
    - Similar en el caso de hilos

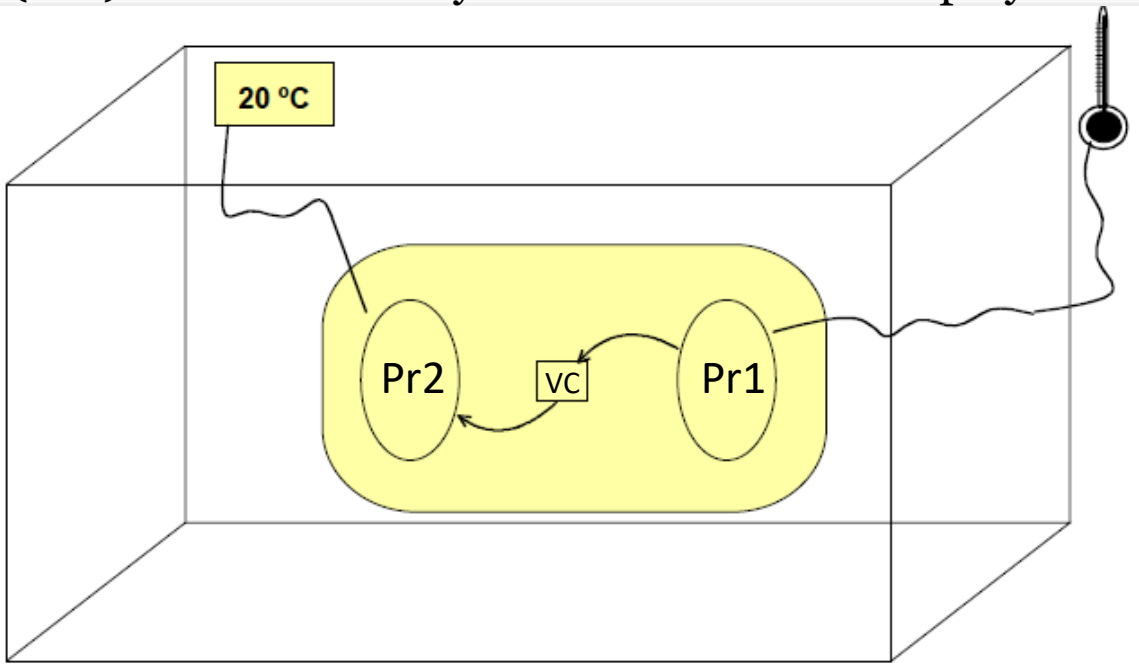


Coordinación

- Definición de **coordinación**:
  - "Conjunto de secuencias ejecutadas simultáneamente que cooperan para un objetivo común"
- Es la clave para un programa concurrente
- Se consigue usando variables compartidas
  - Dos modelos puros
    - **Comunicación** = intercambio de datos en variables compartidas sin controles
    - **Sincronización** = detención y reanudación de procesos, y su relación temporal
  - Un modelo mixto
    - **Comunicación sincronizada** = intercambio de información con control del momento en que se realiza para asegurar su integridad

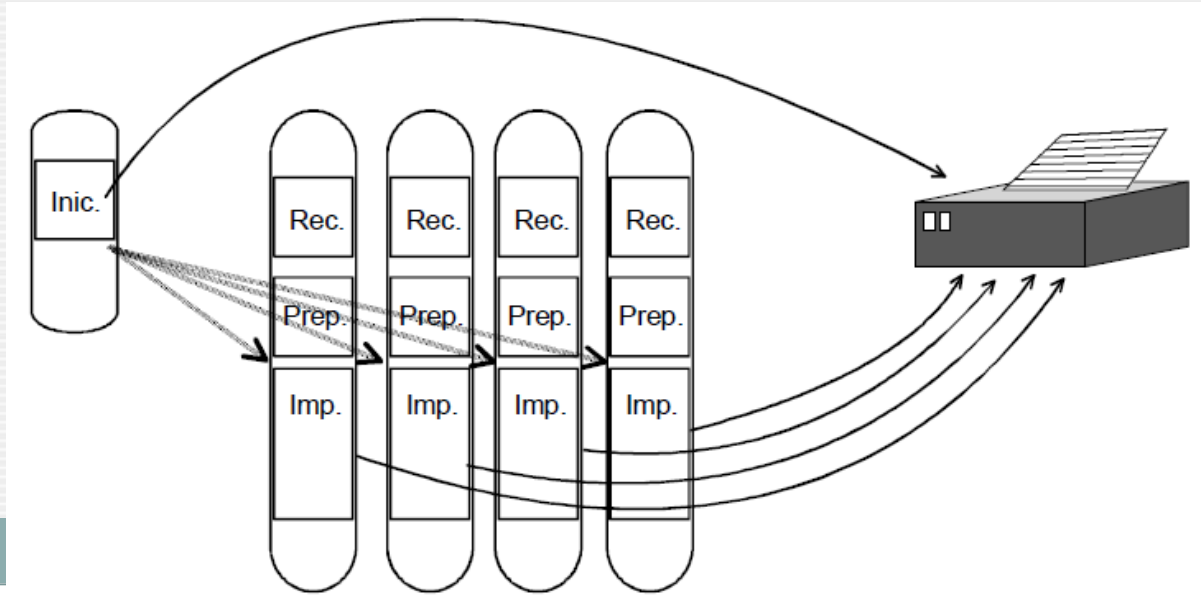
Coordinación: Ejemplos

- **Comunicación pura**
  - Dos procesos intercambiando información no crítica usando una variable compartida
    - Uno (Pr1) toma la temperatura en un sensor y la escribe en la variable compartida (VC)
    - El otro (Pr2) lee dicho valor y lo muestra en un "display"



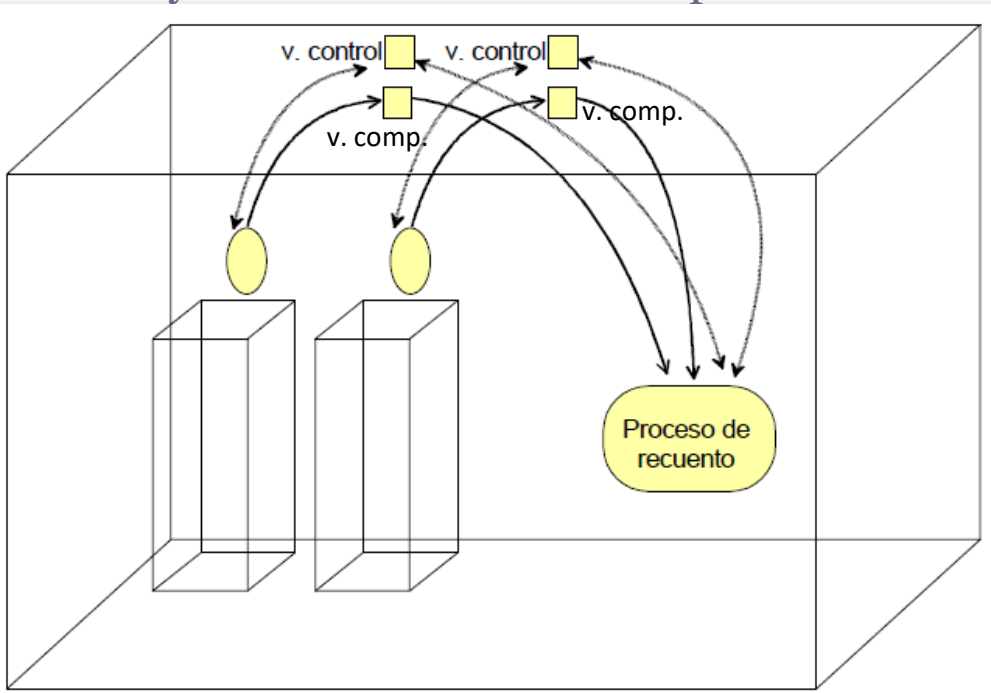
Coordinación: Ejemplos

- **Sincronización pura**
  - Un proceso inicializa una impresora (10 seg.) y envía una señal de "impresora lista" a los otros
  - Los otros procesos tardan entre 2 y 15 seg. en preparar cada documento para imprimirlo
  - Es más eficiente que todos los procesos empiecen a la vez y realicen sus tareas específicas para luego esperar a la impresora



Coordinación: Ejemplos

- **Comunicación sincronizada**
  - Un conjunto de cabinas de votación y un sistema central de recuento
  - Cada cabina esta controlada por un proceso y el sistema central por otro
  - El valor de cada voto se pone en una variable compartida por cada cabina y además tendremos variables adicionales de control que nos informen de si cada voto es nuevo o ya ha sido contabilizado por el sistema central

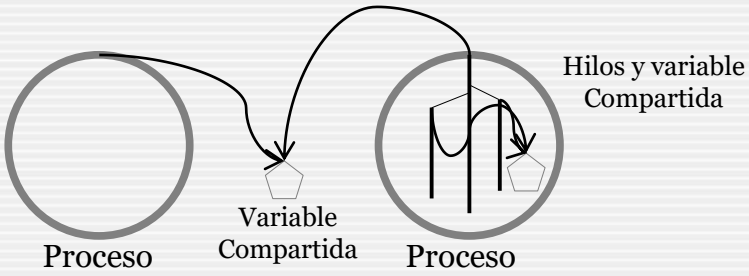


Coordinación: Problema

- La coordinación mediante variables compartidas:
  - Pone en riesgo de inconsistencia dichas variables (condiciones de carrera)
    - La solución a este riesgo es **garantizar la exclusión mutua** entre procesos cuando acceden a la misma variable compartida
- Definiciones:
  - **Exclusión mutua**
    - Sólo un hilo puede acceder a la vez
  - **Sección crítica** (SC)
    - Fragmento de código donde la corrección de un programa se ve comprometida por el uso de variables compartidas
- Solución:
  - Asegurar la ejecución en exclusión mutua de toda SC sobre una misma variable compartida

Concurrencia y Java

- Los modelos de coordinación necesitan variables compartidas
  - En concurrencia con procesos, proporcionadas por SO
  - En concurrencia con hilos, cualquier variable del programa



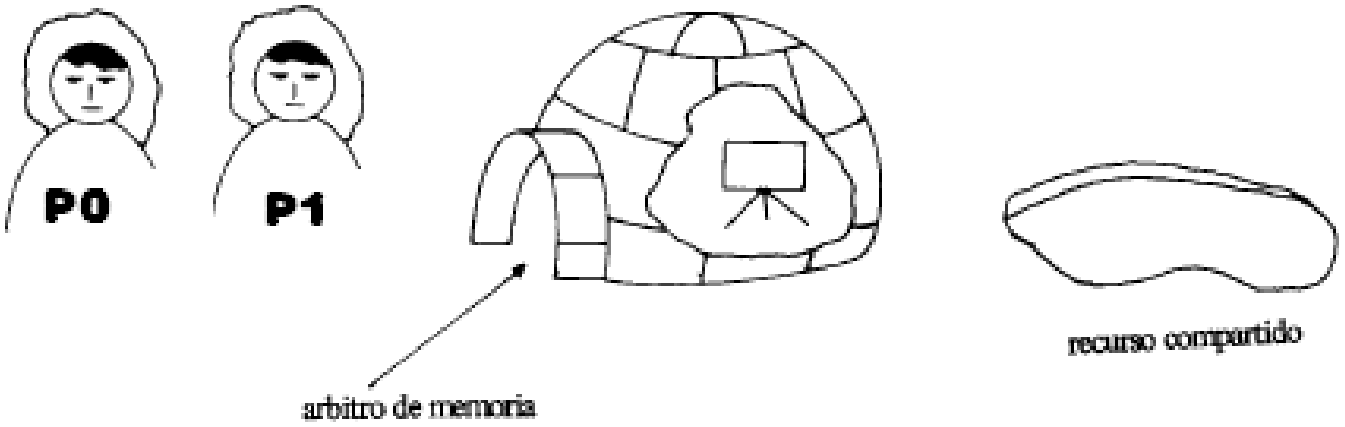
- Hilos en Java
  - Usaremos **atributos** de objetos para realizar la coordinación
    - Crearemos objetos que todos los hilos a coordinar "conocen"
    - Tendremos atributos en esos objetos compartidos que:
      - Se usarán como variables compartidas
      - Deberán ser protegidos

Concurrencia y Java: Elementos

- Utilizamos varios elementos **básicos**
  - Objetos tipo hilo (clase `Thread` / interfaz `Runnable`)
  - Objetos con atributos para coordinación
    - Modificador `synchronized` para asegurar la exclusión mutua
    - Combinaciones de modificadores de acceso (`private`, `public`, `protected`)
      - Usando atributos `private` y accediendo a ellos con un método
  - Crearemos comportamientos avanzados mediante algoritmos
- Utilizaremos elementos **avanzados** del lenguaje
  - De la "JSR 166: Concurrency Utilities" (J2SE 5.0, 2004)

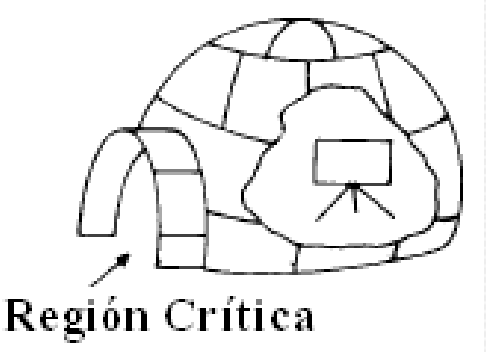
Algoritmos para exclusión mutua: Problema

- **Problema**:
  - Dos **procesos** (esquimales) comparten un **recurso** (un agujero hecho en el hielo) que necesitan utilizar (pescar). Por su naturaleza (tamaño) el **acceso** a dicho recurso debe ser **exclusivo**
  - Escribir el **algoritmo** que represente el comportamiento de los esquimales utilizando variables compartidas (que representen el estado del recurso crítico) para resolver el problema de exclusión mutua planteado



Algoritmos para exclusión mutua: 1º

- **Primera aproximación**
  - Una pizarra dentro de un **iglú** (por su acceso y tamaño sólo puede haber un esquimal dentro => Región Crítica) será la **variable compartida**
    - Esta pizarra indica el **turno** (a quién le toca pescar)
  - Cuando el esquimal P0 quiera ir a pescar, entrará en el iglú y mirará la pizarra:
    - Si hay un 1, se va a dar una vuelta por ahí
    - Si hay un 0, se va a pescar
    - Cuando vuelve de pescar, entra y pone un 1
  - P1 hace lo mismo, cambiando el 0 por el 1



Algoritmos para exclusión mutua: 1º

Primera aproximación en JAVA

```
public class Main {
    public static void main(String[] args) {
        Igloo igloo=new Igloo();
        Esquimal cero=new Esquimal(0,igloo);
        Esquimal uno=new Esquimal(1,igloo);
        cero.start();
        uno.start();
    }
}
```

```
public class Igloo {
    private int turno=0;
    public synchronized int miraTurno(){
        return turno;
    }
    public synchronized void cambiaTurno(int id){
        turno=(id+1)%2;
    }
}
```

```
public class Esquimal extends Thread {
    private int id;
    private Igloo ig;
    public Esquimal(int id, Igloo ig){
        this.id=id;      this.ig=ig;
    }
    public void run(){
        while (true){
            while(ig.miraTurno()!=id){ //Pasear
              try{sleep(10+(int)(20.*Math.random()));}
                 catch(Exception e){}
            } //Ya es mi turno y pesco:
            System.out.println("Esq. "+id+" pescando");
            try{sleep(100+(int)(200.*Math.random()));}
               catch(Exception e){}
            ig.cambiaTurno(id); //Cambio el turno
        }
    }
}
```

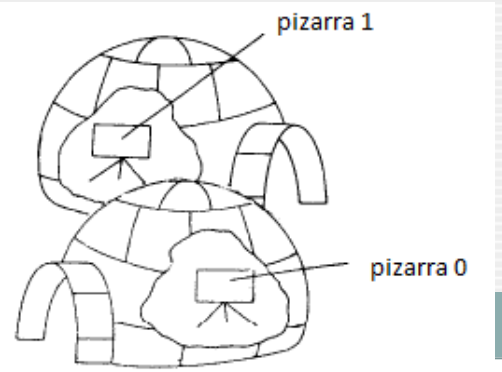
Algoritmos para exclusión mutua: 1º

- **Primera aproximación. Problema: Alternancia**
  - Se consigue la exclusión mutua a cambio de un **funcionamiento muy rígido**: pesca P0, pesca P1, pesca P0, ….
  - Si un proceso quiere usar el recurso crítico, no podrá si no es su turno: tendrá que esperar a que sea usado por el otro proceso
    - Se desaprovecha la CPU

```
run:
Esq. 0 pescando
Esq. 1 pescando
Esq. 0 pescando
Esq. 1 pescando
Esq. 0 pescando
Esq. 1 pescando
…
```

Algoritmos para exclusión mutua: 2º

- **Segunda aproximación**
  - Para evitar la alternancia, se usan **dos iglús** (**dos variables compartidas**): uno para cada esquimal
  - Dentro de cada iglú hay una pizarra en la que su propietario indica "pescando" o "no pescando"
  - Si un esquimal quiere pescar, va al iglú del otro y mira la pizarra:
    - Si está pescando, se va a dar una vuelta
    - Sino, va a su propio iglú, indica que está "pescando" y se va a pescar



Algoritmos para exclusión mutua: 2º

Segunda aproximación en JAVA

```
public class Main {
    public static void main(String[] args) {
        Igloo ig[]=new Igloo[2];
        ig[0]=new Igloo(false); ig[1]=new Igloo(false);
        Esquimal cero=new Esquimal(0,ig);
        Esquimal uno=new Esquimal(1,ig);
        cero.start();   uno.start();
    }
}
```

```
public class Igloo {
    private boolean pescando;
    public Igloo(boolean b) { pescando=b;  }
    public synchronized void pescar(){
        pescando=true; } //Se va a pescar
    public synchronized void noPescar(){
        pescando=false; } //Vuelve de pescar
    public synchronized boolean estaPescando(){
        return pescando;  }
}
```

```
public class Esquimal extends Thread {
    private int yo, tu;
    private Igloo[] ig=new Igloo[2];
    public Esquimal(int id, Igloo[] ig){
        this.yo=id; tu=(id+1)%2; this.ig=ig;  }
    public void run(){
        while (true){
            while(ig[tu].estaPescando()){ // Pasear
                try{sleep(1+(int)(2.*Math.random()));} …
            }
            ig[yo].pescar(); //Escribo que quiero pescar
            if(ig[tu].estaPescando())
  System.out.println("P"+yo+" y P"+tu+" pescando");
            try{sleep(1+(int)(2.*Math.random()));} …
            ig[yo].noPescar();
            try{sleep(1+(int)(2.*Math.random()));} …
        }
    }
}
```

Algoritmos para exclusión mutua: 2º

- **Segunda aproximación. Problema: Falta de exclusión mutua**
  - Puede ocurrir que P0 vaya a ver si P1 está pescando. Ve que no y se vuelve a su iglú, para apuntar que se va a pescar
  - Pero mientras estaba en el iglú de P1, éste fue a mirar al de P0 y vio que no estaba pescando
  - Vuelve a su iglú y apunta que se va a pescar
  - ¡**No hay exclusión mutua** en relación con el recurso crítico!

```
run:
P1 y P0 pescando
```

Algoritmos para exclusión mutua: 3º

- **Tercera aproximación**
  - La situación anterior se ha producido porque:
    - 1º se mira en el iglú del vecino
    - 2º se anota en el propio iglú la situación
  - Para evitarlo, actuaremos a la inversa; es decir, cuando un proceso quiera utilizar el recurso:
    - 1º indica en su propio iglú que quiere hacerlo
    - 2º se esperará hasta que quede libre
  - El programa que satisface este algoritmo es el siguiente:

Algoritmos para exclusión mutua: 3º

Tercera aproximación en JAVA

```
public class Main {
    public static void main(String[] args) {
        Igloo ig[]=new Igloo[2];
        ig[0]=new Igloo(false);
        ig[1]=new Igloo(false);
        Esquimal cero=new Esquimal(0,ig);
        Esquimal uno=new Esquimal(1,ig);
        cero.start();   uno.start(); } }
```

```
public class Igloo {
    private boolean pescando;
    public Igloo(boolean b){ pescando=b; }
    public synchronized void pescar(){
        pescando=true; } // Se va a pescar
    public synchronized void noPescar(){
        pescando=false; } // Vuelve de pescar
    public synchronized boolean estaPescando(){
        return pescando;
    }
}
```

```
public class Esquimal extends Thread {
    private int yo, tu;
    private Igloo[] ig=new Igloo[2];
    public Esquimal(int id, Igloo[] ig){
        this.yo=id; tu=(id+1)%2; this.ig=ig;     }
    public void run(){
        while (true){
            ig[yo].pescar(); // Escribo que quiero pescar
            while(ig[tu].estaPescando()){ // Pasear
                try{sleep(1+(int)(2.*Math.random()));} …
            }
            System.out.println("Esq."+yo+" pescando");
            try{sleep(1+(int)(2.*Math.random()));} …
            ig[yo].noPescar();
            try{sleep(1+(int)(2.*Math.random()));} …
        }
    }
}
```

Algoritmos para exclusión mutua: 3º

- **Tercera aproximación. Problema: Interbloqueo**
  - Puede darse la siguiente situación:
    1. Po pone pizarra-0 a 'pescando'
    2. P1 pone pizarra-1 a 'pescando'
    3. Po comprueba que la condición de su `while (ig[1].estaPescando())` es cierta y se queda ejecutando el bucle
    4. P1 comprueba que la condición de su `while (ig[0].estaPescando())` es cierta y se queda ejecutando el bucle
  - Cada proceso espera que el otro cambie el valor de su pizarra y ambos se quedan en una espera infinita (deadlock)

```
run:
Esq.0 pescando
Esq.1 pescando
GENERACIÓN INTERRUMPIDA (total time: 2 minutes 56 seconds)
```

Algoritmos para exclusión mutua: 4º

- **Cuarta aproximación**
  - Para solucionar este problema de interbloqueo y espera infinita introduciremos un tratamiento de **cortesía**:
    - Cuando un proceso ve que el otro quiere utilizar el recurso, le **cede** el paso cortésmente
  - El programa que implementa este algoritmo es el siguiente:

Algoritmos para exclusión mutua: 4º

Cuarta aproximación en JAVA

```
public class Main {
    public static void main(String[] args) {
        Igloo ig[]=new Igloo[2];
        ig[0]=new Igloo(false);
        ig[1]=new Igloo(false);
        Esquimal cero=new Esquimal(0,ig);
        Esquimal uno=new Esquimal(1,ig);
        cero.start();    uno.start();  }  }
```

```
public class Igloo {
    private boolean pescando;
    public Igloo(boolean b){ pescando=b;   }
    public synchronized void pescar(){
        pescando=true; } // Se va a pescar
     public synchronized void noPescar(){
        pescando=false; } //Vuelve de pescar
     public synchronized boolean estaPescando(){
        return pescando;    }
}
```

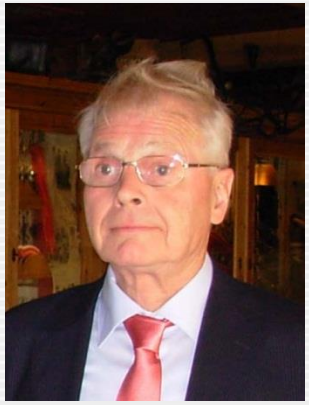
```
public class Esquimal extends Thread {
    private int yo,tu; private Igloo[] ig=new Igloo[2];
    public Esquimal(int id, Igloo[] ig){
        this.yo=id; tu=(id+1)%2;  this.ig=ig; }
    public void run(){
        while (true){
            ig[yo].pescar(); //Escribo que quiero pescar
            while(ig[tu].estaPescando()){
                ig[yo].noPescar(); // Cedo el paso
                try{sleep(100+(int)(200.*Math.random()));}
                    catch(Exception e){}
                ig[yo].pescar(); } // Apunto para pescar
            }
            System.out.println("Esq."+yo+" pescando");
            try{sleep(100+(int)(200.*Math.random()));}
                catch(Exception e){}
            ig[yo].noPescar();
            try{sleep(100+(int)(200.*Math.random()));}
               catch(Exception e){}        }    } }
```

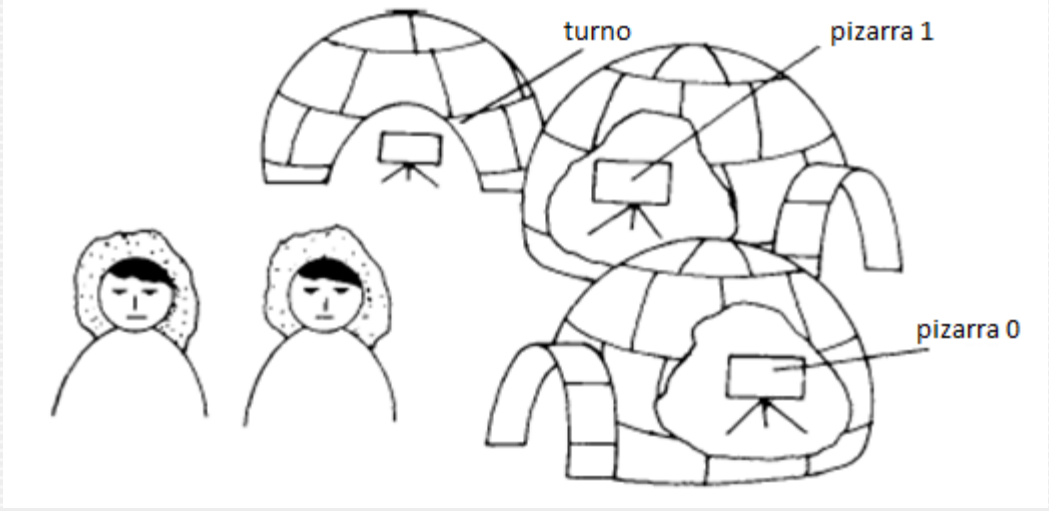
Algoritmos para exclusión mutua: 4º

- **Cuarta aproximación. Problema: Espera indefinida**
  - En esta solución, cuando P_i intenta acceder al recurso y comprueba que el otro también lo desea:
    - **cede** su turno (pone pizarra-i a "no pescando")
    - **espera** un tiempo
    - y luego **reclama** su derecho a utilizar el recurso
  - Este tratamiento de cortesía puede conducir a que los procesos se queden de manera indefinida cediéndose mutuamente el paso
  - Es muy difícil de reproducir, pero esta solución no asegura que se acceda al recurso en un tiempo finito (inanición)

Algoritmos para exclusión mutua: 5º

- Algoritmo de Dekker
  - El último problema pendiente se resuelve:
    - Con **dos pizarras** (= 4ª aproximación)
    - Y un **turno** (= 1ª aproximación)
  - En caso de conflicto, el valor del turno determina a quién se le concede el acceso al recurso





Algoritmos para exclusión mutua: 5º

Algoritmo de Dekker en JAVA

```
public class Main
{
    public static void main(String[] args)
    {
        Igloo4 ig[]=new Igloo4[2];
        ig[0]=new Igloo4(false);
        ig[1]=new Igloo4(false);
        Igloo1 igt=new Igloo1();
        Esquimal cero=new Esquimal(0,ig,igt);
        Esquimal uno=new Esquimal(1,ig,igt);
        cero.start();    uno.start();
    }
}
```

```
public class Igloo1
{
// Versión 1ª
}
```

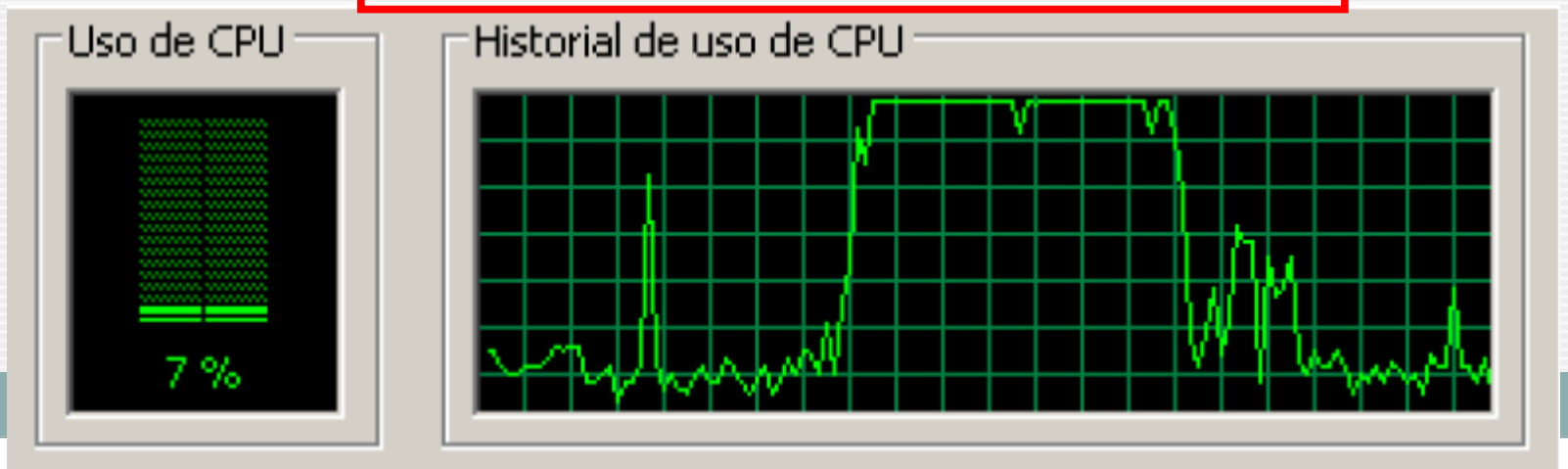
```
public class Igloo4
{
// Versión 4ª
}
```

```
public class Esquimal extends Thread {
    private int yo,tu;  private Igloo4[] ig=new Igloo4[2];
    private Igloo1 igt;
    public Esquimal(int id, Igloo4[] ig, Igloo1 igt){
        this.yo=id; tu=(id+1)%2; this.ig=ig; this.igt=igt; }
    public void run(){
        while (true){
            ig[yo].pescar(); // Escribo que quiero pescar
            while(ig[tu].estaPescando()){
                ig[yo].noPescar(); // Cortesía: cedo el paso
                while(igt.miraTurno()==tu) // Paseo
                   try{sleep(100+(int)(200.*Math.random()));}…
                ig[yo].pescar(); //Apunto para pescar
            }
            System.out.println("Esq."+yo+" pescando");
            try{sleep(100+(int)(200.*Math.random()));} …
            ig[yo].noPescar();
            igt.cambiaTurno(yo);
            try{sleep(100+(int)(200.*Math.random()));} …} }}
```

Algoritmos para exclusión mutua: problema

- Obtener la exclusión mutua mediante algoritmos
  - Con un algoritmo depurado (Dekker) se puede obtener, evitando interbloqueo, inanición y otros problemas
  - Sigue quedando pendiente el problema de la **espera activa**
    - Esperar por algo comprobando continuamente y ocupando el procesador

```
...
while(igt.miraTurno()==tu)  // Paseo
      try{sleep(100+(int)(200.*Math.random()));}…
...
```

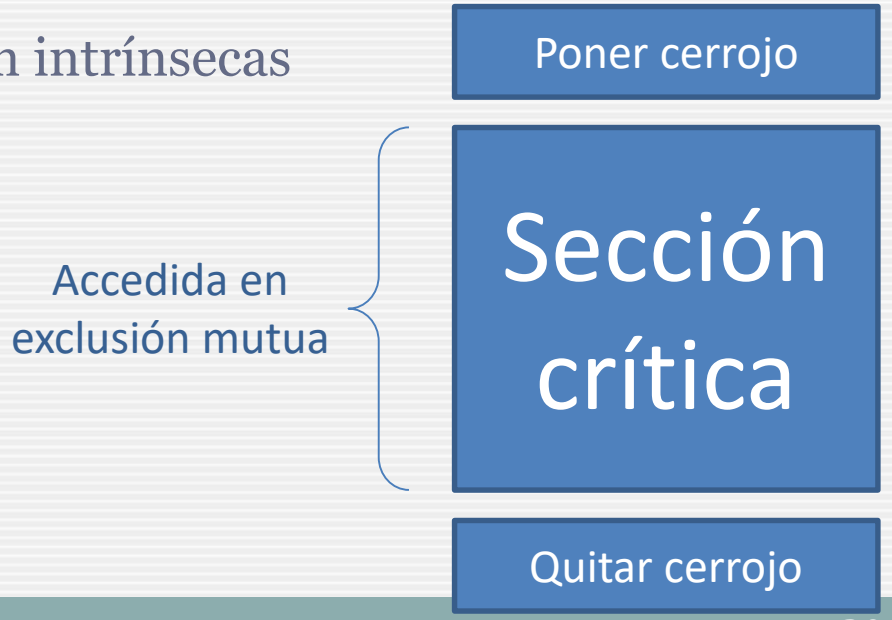


Soluciones no algorítmicas

- Para resolver los problemas sin algoritmos complejos, necesitamos ayuda del HW
  - Por ejemplo:
    - Inhibición de interrupciones
    - Instrucciones máquina especiales
      - Ejecutadas en un sólo ciclo: (leer y escribir), (leer y comprobar), …
  - Estas soluciones siguen teniendo espera activa
- Para evitar la espera activa necesitamos al SO
  - Semáforos, Regiones Críticas y otros mecanismos
  - Nuevos estados de espera

Locks (cerrojos)

- Los cerrojos nos permiten acceder a un recurso (sección crítica) en exclusión mutua
- Dos tipos:
  - **Explícitos**: las operaciones de "poner" y "quitar" el cerrojo se indican expresamente
  - **Implícitos**: las operaciones son intrínsecas



Locks implícitos y explícitos

- El bloqueo de los hilos en Java se implementó con:
  - Cláusula `synchronized` = lock **implícito** (se verá más adelante)
- El paquete `java.util.concurrent.locks` proporciona locks **explícitos** con:
  - Alto rendimiento
  - Misma semántica que `synchronized`
  - Soportando timeout cuando se intenta adquirir un lock
  - Soportando múltiples variables de condición por cada lock
  - Soportando bloqueos no limitados léxicamente
  - Soportando interrupción de hilos que esperan para adquirir un lock
- Los locks explícitos se introducen en Java 5.0 (2004)

Locks explícitos

- Definidos con la clase `ReentrantLock`
  - Funcionalidad descrita en **interfaz Lock**
- La interfaz Lock define un conjunto de operaciones abstractas de **adquisición** y **liberación** de un lock
- A diferencia del lock implícito (se verá más adelante), la interfaz Lock ofrece diferentes formas de toma de un lock:
  - Incondicional
  - No bloqueante
  - Temporizado
  - Interrumpible
- Todas las **operaciones** de adquisición y liberación de un Lock son **explícitas**

Locks explícitos

- public **interface Lock**
  - Constructor:
    - Lock lock = new ReentrantLock(); //Crea un lock (se maneja a través de la interfaz)
  - Métodos
    - void **lock**(); // Adquiere el lock
    - void **lockInterruptibly**(); //Lo adquiere si el thread no está en estado interrumpido
    - boolean **tryLock**(); //Adquiere el lock sólo si está libre en tiempo de invocación
    - boolean **tryLock**(long timeout, TimeUnit unit); //Adquiere el lock si queda libre en el // tiempo especificado y el thread no está interrumpido
    - void **unlock**(); //Libera el lock
    - Condition **newCondition()**; //Devuelve una nueva condición asociada a este lock

Locks explícitos

- Sección crítica protegida con un Lock
  - La **liberación** debe hacerse en una sentencia **`finally`**
    - Hay que prever la posibilidad de una excepción, y en este caso el lock debe liberarse explícitamente también

```
Lock control = new ReentrantLock();
...
control.lock();
try {
    // Actualiza el objeto protegido por el lock (Sección Crítica)
}
catch (Exception e) {
    // Atiende excepciones, si es necesario
}
finally {
    control.unlock();
}
```

Locks explícitos: ejemplo

- Ejemplo de secuencia (visto en tema 2) con cerrojos

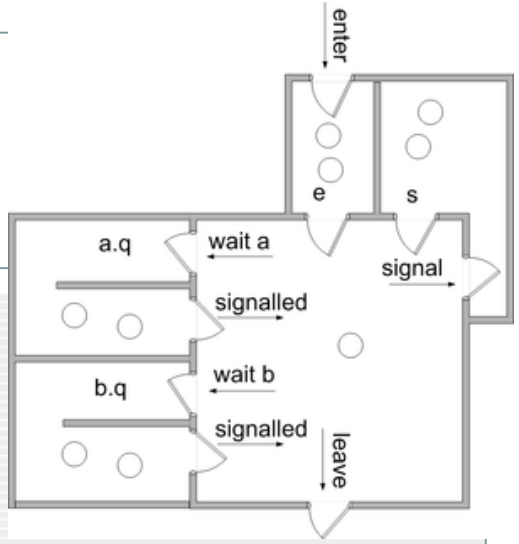
```
public class Secuencia {
    private int valor=0;
    Lock lock=new ReentrantLock();
    public int getSiguiente() {
        lock.lock();
        try {
            valor++;
            return valor;
            //Devuelve un único valor
        } finally {
            lock.unlock();
        } } }
```

```
public class Main {
    public static void main(String[] args) {
        Secuencia sec=new Secuencia();
        for(int i=0; i<100; i++) {
            Hilo hilo=new Hilo(i, sec);
            hilo.start();
        }
    }
}
```

```
public class Hilo extends Thread {
    private Secuencia sec;
    private int id;
    Hilo(int id, Secuencia sec) {
        this.id=id;     this.sec=sec;
    }
    public void run() {
        System.out.println("Hilo "+id+": secuencia="+sec.getSiguiente());
    }
}
```

```
run:
Hilo 1: secuencia=1
Hilo 0: secuencia=2
Hilo 3: secuencia=3
…
Hilo 93: secuencia=99
Hilo 99: secuencia=100
BUILD SUCCESSFUL (total time: 0 seconds)
```

Condition



- Los **locks** por sí mismos sólo permiten resolver problemas de **comunicación**: necesitamos algo para resolver problemas de **sincronización**
  - Para ello, vamos a usar objetos **Condition**
- Un objeto Condition está estructuralmente ligado a un objeto Lock
  - Se puede crear un Condition invocando el método **newCondition()** sobre un objeto Lock:
    - Condition miCondicion = miLock.newCondition();
- El objeto Condition sólo puede ser **utilizado por un thread que** previamente **haya tomado el Lock** al que pertenece
  - Cada objeto Condition representará una condición de espera

Condition

- Interface Condition
  - Es implementada por variables de condición asociadas a un ReentrantLock

```
public interface Condition {
    void await();  //El thread espera hasta ser signalled o interrupted
    boolean await(long time, TimeUnit unit) // El thread espera el tiempo especificado o a
                                            // ser signalled o interrupted
    long awaitNanos (long nanosTimeout)  //Ídem, con el tiempo en nanosegundos
    void awaitUninterruptibly()  // El current thread espera hasta ser signalled
    boolean awaitUntil(Date deadline)  // El thread espera hasta ser signalled o
                                       // interrupted, o hasta la hora especificada
    void signal()  // Despierta a un thread que esté esperando por ese Condition
    void signalAll()  // Despierta a todos los threads que estén esperando por ese Condition
}
```

Condition: ejemplo Productor-Consumidor

- 4 clases:
  - Main: crea el buffer, el productor y el consumidor, y los lanza
  - Buffer: objeto compartido con operaciones de insertar y extraer
  - Productor: genera un elemento y lo inserta en el buffer
  - Consumidor: consume un elemento y lo elimina del buffer

```
public class Main {
    public static void main(String[] args) {
        Buffer buf = new Buffer(10);
        Productor prod = new Productor(buf, "Productor");
        Consumidor cons = new Consumidor(buf, "Consumidor");
        prod.start();
        cons.start();
    }
}
```

Condition: ejemplo Productor-Consumidor

```
public class Productor extends Thread {
    Buffer buf;
    String id;
    public Productor(Buffer buf, String id) {
        this.buf = buf;
        this.id = id;
    }
    public void run() {
        Object msg;
        for (int i = 1; i <= 20; i++) {
            try {
                sleep(500 + (int) (200 * Math.random()));
                msg = (Object) (id + " - " + i);
                buf.insertar(msg);
                System.out.println("Produzco: "+msg);
            } catch (InterruptedException e) {}
        }
    }
}
```

```
public class Consumidor extends Thread {
    Buffer buf;
    String id;
    public Consumidor(Buffer buf, String id) {
        this.buf = buf;
        this.id = id;
    }
    public void run() {
        Object msg;
        for (int i = 1; i <= 20; i++) {
            try {
                sleep(3000 + (int) (200 * Math.random()));
                msg=buf.extraer();
                System.out.println("Consumo: "+msg);
            } catch (InterruptedException e) {}
        }
    }
}
```

Condition: ejemplo Productor-Consumidor

```
public class Buffer  {
    private Object[] buf;
    private int in = 0, out = 0, numElem = 0, maximo = 0;
    private Lock control = new ReentrantLock();
    private Condition lleno = control.newCondition();
    private Condition vacio = control.newCondition();
    public Buffer(int max) {
        this.maximo = max;
        buf = new Object[max];
    }
    public void insertar(Object obj) throws InterruptedException {
        control.lock();
        while (numElem==maximo) { //Buffer lleno
            lleno.await();
        }
        try { buf[in] = obj;
            numElem++;
            in = (in + 1) % maximo;
            vacio.signal(); //Buffer ya no está vacío
        } finally { control.unlock(); }
    }
```

```
public Object extraer() throws InterruptedException {
        control.lock();
        while (numElem==0) { //Buffer vacío
            vacio.await();
        }
        Object obj;
        try {
            obj = buf[out];
            buf[out] = null;
            numElem = numElem - 1;
            out = (out + 1) % maximo;
            lleno.signal(); //Buffer ya no está lleno
            return (obj);
        } finally { control.unlock(); }
    }
}
```

Locks de lectura/escritura

- La interfaz **Lock** garantiza la **exclusión mutua** en el bloque que protege
  - No hace diferencia entre procesos de escritura y procesos de lectura
  - En el problema de lectores-escritores, la exclusión mutua debería diferenciar las situaciones:
    - **escritor/escritor** y **escritor/lector** que son **incompatibles**
    - **lector/lector** que sí son **compatibles**
- La interfaz **ReadWriteLock** diferencia **dos tipos de lock**: uno para **escritura** (*writer*) y otro para **lectura** (*reader*)

```
public interface ReadWriteLock {
    Lock readLock(); // Devuelve el cerrojo (lock) usado para lectura
    Lock writeLock(); // Devuelve el cerrojo (lock) usado para escritura
}
```

- La clase **ReentrantReadWriteLock** implementa esta interfaz
  - ReadWriteLock lock = new ReentrantReadWriteLock();

Locks de lectura/escritura: ejemplo

- Problema de lectores/escritores con ReentrantReadWriteLock

```
public class Agenda {
    private HashMap<String, String> agenda;
    private ReadWriteLock lock = new ReentrantReadWriteLock();
    private Lock r = lock.readLock();
    private Lock w = lock.writeLock();
    public Agenda(HashMap agenda) {
        this.agenda = agenda;
    }

    public void escribir(String clave, String valor) {
        w.lock();
        try {
            agenda.put(clave, valor);
        } finally {
            w.unlock();
        }
    }
```

```
    public String leer(String clave) {
        r.lock();
        try {
            String valor = agenda.get(clave);
            return valor;
        } finally {
            r.unlock();
        }
    }
}
```

Locks de lectura/escritura: ejemplo

```
public class Lector extends Thread
{
    Agenda agenda;
    String id;

    public Lector(Agenda a, String id)
    {
        agenda = a;
        this.id = id;
    }

    public void run()
    {
       for (int i = 1; i <= 20; i++)
       {
         try
         {
           sleep(100 + (int) (200 * Math.random()));
           String valor = agenda.leer(i+"");
           System.out.println("Lectura: "+i+"->"+valor);
         }
         catch (InterruptedException e) {}
       }
    }
}
```

```
public class Escritor extends Thread
{
    Agenda agenda;
    String id;

    public Escritor(Agenda a, String id)
    {
        agenda = a;
        this.id = id;
    }

    public void run()
    {
       for (int i = 1; i <= 20; i++)
       {
          try
          {
             sleep(100 + (int) (200 * Math.random()));
             agenda.escribir(i+"", id+i);
             System.out.println("Escribo: "+i+"->"+id+i);
          }
          catch (InterruptedException e) {}
       }
    }
}
```

Locks de lectura/escritura: ejemplo

- Posible clase para lanzar el ejemplo
  - 3 lectores y 1 escritor

```
public class Main
{
    public static void main(String[] args)
    {
        HashMap a = new HashMap();
        Agenda agenda = new Agenda(a);
        Lector lector1 = new Lector(agenda, "lec1");
        Lector lector2 = new Lector(agenda, "lec2");
        Lector lector3 = new Lector(agenda, "lec3");
        Escritor escritor = new Escritor(agenda, "esc1");
        lector1.start();
        lector2.start();
        lector3.start();
        escritor.start();
    }
}
```

```
run:
Lectura: 1->null
Escribo: 1->esc1-1
Lectura: 1->esc1-1
Lectura: 1->esc1-1
Lectura: 2->null
Escribo: 2->esc1-2
Lectura: 2->esc1-2
…
Lectura: 20->null
Lectura: 18->esc1-18
Escribo: 20->esc1-20
Lectura: 17->esc1-17
Lectura: 19->esc1-19
Lectura: 18->esc1-18
Lectura: 20->esc1-20
Lectura: 19->esc1-19
Lectura: 20->esc1-20
BUILD SUCCESSFUL (total time: 4 seconds)
```

Ejercicios

- 1.- Ejecutar los códigos utilizados en esta presentación.
- 2.- Crear un programa en Java que simule el problema del sensor de la temperatura visto en esta presentación utilizando únicamente Locks explícitos. [Comunicación]
- 3.- Crear un programa en Java que simule el problema anterior, pero utilizando un objeto compartido (buffer de tamaño 10) donde se vayan depositando las temperaturas. Deben utilizarse Locks y Conditions. [Comunicación sincronizada]
- 4.- Crear un programa en Java que simule el problema de las cabinas de votación con un sistema central de recuento, tal cual se ha planteado en esta presentación. Deben utilizarse Locks y Conditions. [Comunicación sincronizada]